# **并发编程**

| **Tedu Python 教学部** |
| --- |
| Author：吕泽 |
| Days：6天 |

* [多任务编程](#%E5%A4%9A%E4%BB%BB%E5%8A%A1%E7%BC%96%E7%A8%8B)
* [进程（process）](#%E8%BF%9B%E7%A8%8Bprocess)
  + [进程理论基础](#%E8%BF%9B%E7%A8%8B%E7%90%86%E8%AE%BA%E5%9F%BA%E7%A1%80)
* [基于fork的多进程编程](#%E5%9F%BA%E4%BA%8Efork%E7%9A%84%E5%A4%9A%E8%BF%9B%E7%A8%8B%E7%BC%96%E7%A8%8B)
  + [fork使用](#fork%E4%BD%BF%E7%94%A8)
  + [进程相关函数](#%E8%BF%9B%E7%A8%8B%E7%9B%B8%E5%85%B3%E5%87%BD%E6%95%B0)
  + [孤儿和僵尸](#%E5%AD%A4%E5%84%BF%E5%92%8C%E5%83%B5%E5%B0%B8)
  + [群聊聊天室](#%E7%BE%A4%E8%81%8A%E8%81%8A%E5%A4%A9%E5%AE%A4)
* [multiprocessing 模块创建进程](#multiprocessing-%E6%A8%A1%E5%9D%97%E5%88%9B%E5%BB%BA%E8%BF%9B%E7%A8%8B)
  + [进程创建方法](#%E8%BF%9B%E7%A8%8B%E5%88%9B%E5%BB%BA%E6%96%B9%E6%B3%95)
  + [进程池实现](#%E8%BF%9B%E7%A8%8B%E6%B1%A0%E5%AE%9E%E7%8E%B0)
* [进程间通信（IPC）](#%E8%BF%9B%E7%A8%8B%E9%97%B4%E9%80%9A%E4%BF%A1ipc)
  + [管道通信(Pipe)](#%E7%AE%A1%E9%81%93%E9%80%9A%E4%BF%A1pipe)
  + [消息队列](#%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97)
  + [共享内存](#%E5%85%B1%E4%BA%AB%E5%86%85%E5%AD%98)
  + [本地套接字](#%E6%9C%AC%E5%9C%B0%E5%A5%97%E6%8E%A5%E5%AD%97)
  + [信号量（信号灯集）](#%E4%BF%A1%E5%8F%B7%E9%87%8F%E4%BF%A1%E5%8F%B7%E7%81%AF%E9%9B%86)
* [线程编程（Thread）](#%E7%BA%BF%E7%A8%8B%E7%BC%96%E7%A8%8Bthread)
  + [线程基本概念](#%E7%BA%BF%E7%A8%8B%E5%9F%BA%E6%9C%AC%E6%A6%82%E5%BF%B5)
  + [threading模块创建线程](#threading%E6%A8%A1%E5%9D%97%E5%88%9B%E5%BB%BA%E7%BA%BF%E7%A8%8B)
  + [线程对象属性](#%E7%BA%BF%E7%A8%8B%E5%AF%B9%E8%B1%A1%E5%B1%9E%E6%80%A7)
  + [自定义线程类](#%E8%87%AA%E5%AE%9A%E4%B9%89%E7%BA%BF%E7%A8%8B%E7%B1%BB)
* [同步互斥](#%E5%90%8C%E6%AD%A5%E4%BA%92%E6%96%A5)
  + [线程间通信方法](#%E7%BA%BF%E7%A8%8B%E9%97%B4%E9%80%9A%E4%BF%A1%E6%96%B9%E6%B3%95)
  + [线程同步互斥方法](#%E7%BA%BF%E7%A8%8B%E5%90%8C%E6%AD%A5%E4%BA%92%E6%96%A5%E6%96%B9%E6%B3%95)
    - [线程Event](#%E7%BA%BF%E7%A8%8Bevent)
    - [线程锁 Lock](#%E7%BA%BF%E7%A8%8B%E9%94%81-lock)
  + [死锁及其处理](#%E6%AD%BB%E9%94%81%E5%8F%8A%E5%85%B6%E5%A4%84%E7%90%86)
* [python线程GIL](#python%E7%BA%BF%E7%A8%8Bgil)
* [进程线程的区别联系](#%E8%BF%9B%E7%A8%8B%E7%BA%BF%E7%A8%8B%E7%9A%84%E5%8C%BA%E5%88%AB%E8%81%94%E7%B3%BB)
  + [区别联系](#%E5%8C%BA%E5%88%AB%E8%81%94%E7%B3%BB)
  + [使用场景](#%E4%BD%BF%E7%94%A8%E5%9C%BA%E6%99%AF)
  + [要求](#%E8%A6%81%E6%B1%82)
* [并发网络通信模型](#%E5%B9%B6%E5%8F%91%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1%E6%A8%A1%E5%9E%8B)
  + [常见模型分类](#%E5%B8%B8%E8%A7%81%E6%A8%A1%E5%9E%8B%E5%88%86%E7%B1%BB)
  + [基于fork的多进程网络并发模型](#%E5%9F%BA%E4%BA%8Efork%E7%9A%84%E5%A4%9A%E8%BF%9B%E7%A8%8B%E7%BD%91%E7%BB%9C%E5%B9%B6%E5%8F%91%E6%A8%A1%E5%9E%8B)
    - [实现步骤](#%E5%AE%9E%E7%8E%B0%E6%AD%A5%E9%AA%A4)
  + [基于threading的多线程网络并发](#%E5%9F%BA%E4%BA%8Ethreading%E7%9A%84%E5%A4%9A%E7%BA%BF%E7%A8%8B%E7%BD%91%E7%BB%9C%E5%B9%B6%E5%8F%91)
    - [实现步骤](#%E5%AE%9E%E7%8E%B0%E6%AD%A5%E9%AA%A4-1)
  + [@@扩展：集成模块完成多进程/线程网络并发](#%E6%89%A9%E5%B1%95%E9%9B%86%E6%88%90%E6%A8%A1%E5%9D%97%E5%AE%8C%E6%88%90%E5%A4%9A%E8%BF%9B%E7%A8%8B%E7%BA%BF%E7%A8%8B%E7%BD%91%E7%BB%9C%E5%B9%B6%E5%8F%91)
  + [ftp 文件服务器](#ftp-%E6%96%87%E4%BB%B6%E6%9C%8D%E5%8A%A1%E5%99%A8)
* [IO并发](#io%E5%B9%B6%E5%8F%91)
  + [IO 分类](#io-%E5%88%86%E7%B1%BB)
    - [阻塞IO](#%E9%98%BB%E5%A1%9Eio)
    - [非阻塞IO](#%E9%9D%9E%E9%98%BB%E5%A1%9Eio)
  + [IO多路复用](#io%E5%A4%9A%E8%B7%AF%E5%A4%8D%E7%94%A8)
    - [select 方法](#select-%E6%96%B9%E6%B3%95)
  + [@@扩展: 位运算](#%E6%89%A9%E5%B1%95-%E4%BD%8D%E8%BF%90%E7%AE%97)
    - [poll方法](#poll%E6%96%B9%E6%B3%95)
    - [epoll方法](#epoll%E6%96%B9%E6%B3%95)
  + [协程技术](#%E5%8D%8F%E7%A8%8B%E6%8A%80%E6%9C%AF)
    - [基础概念](#%E5%9F%BA%E7%A1%80%E6%A6%82%E5%BF%B5)
    - [扩展延伸@标准库协程的实现](#%E6%89%A9%E5%B1%95%E5%BB%B6%E4%BC%B8%E6%A0%87%E5%87%86%E5%BA%93%E5%8D%8F%E7%A8%8B%E7%9A%84%E5%AE%9E%E7%8E%B0)
    - [第三方协程模](#%E7%AC%AC%E4%B8%89%E6%96%B9%E5%8D%8F%E7%A8%8B%E6%A8%A1)
  + [HTTPServer v2.0](#httpserver-v20)

## **多任务编程**

1. 意义： 充分利用计算机多核资源，提高程序的运行效率。
2. 实现方案 ：多进程 ， 多线程
3. 并行与并发

**并发** ： 同时处理多个任务，内核在任务间不断的切换达到好像多个任务被同时执行的效果，实际每个时刻只有一个任务占有内核。

**并行** ： 多个任务利用计算机多核资源在同时执行，此时多个任务间为并行关系。

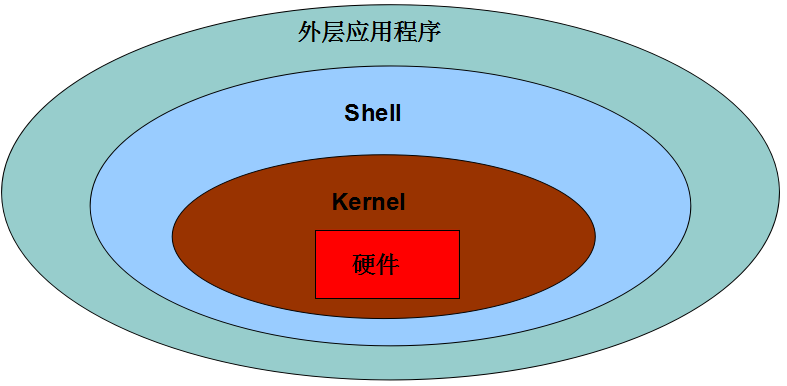
## **进程（process）**

### **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。

1. 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



1. 进程基本概念

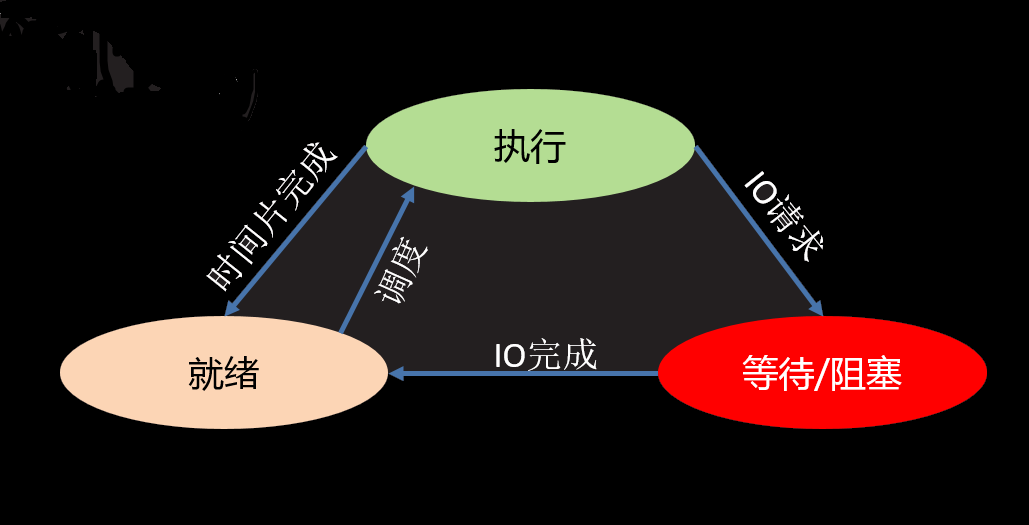
* **cpu时间片**：如果一个进程占有cpu内核则称这个进程在cpu时间片上。
* **PCB(进程控制块)**：在内存中开辟的一块空间，用于存放进程的基本信息，也用于系统查找识别进程。
* **进程ID（PID）**： 系统为每个进程分配的一个大于0的整数，作为进程ID。每个进程ID不重复。

Linux查看进程ID ： ps -aux

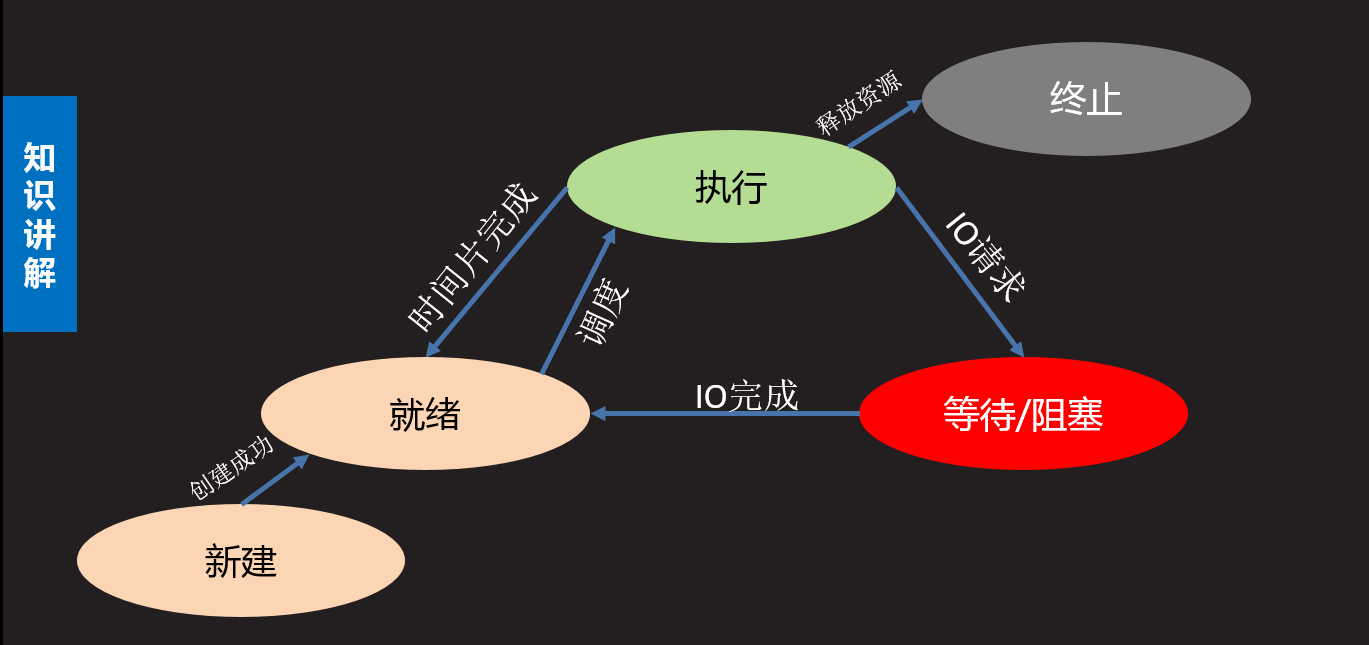
* **父子进程** ： 系统中每一个进程(除了系统初始化进程)都有唯一的父进程，可以有0个或多个子进程。父子进程关系便于进程管理。

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

D 等待态

T 等待态(暂停)

Z 僵尸

< 有较高优先级

N 优先级较低

+ 前台进程

s 会话组组长

l 有多线程的

* 进程的运行特征  
  【1】 进程可以使用计算机多核资源  
  【2】 进程是计算机**分配资源**的最小单位  
  【3】 进程之间的运行互不影响，各自独立  
  【4】 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

## **基于fork的多进程编程**

### **fork使用**

代码示例：day01/fork.py  
代码示例：day01/fork1.py

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：day01/get\_pid.py  
代码示例：day01/exit.py

**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，一般为0退出

**sys.exit([status])**  
功能：退出进程  
参数：整数 表示退出状态，默认为0  
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**。

1. **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。

1. **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：day01/wait.py

```

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

pid,status = **os.waitpid**(pid,option)

功能： 在父进程中处理子进程退出状态

参数： pid -1 表示等待任意子进程退出

>0 表示等待指定的子进程退出

option 0 表示阻塞等待

WNOHANG 表示非阻塞

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：day01/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**

代码示例：day01/signal\_处理僵尸.py

**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

### **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

## **multiprocessing 模块创建进程**

### **进程创建方法**

代码示例：day2/process1.py  
代码示例：day2/process2.py  
代码示例：day2/process3.py

#### 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

#### 基本接口使用

**Process()**类

功能 ： 创建进程对象

参数 ： target 绑定要执行的目标函数(必须写)

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（eg:input()函数，会报EOF异常）

#### 进程对象属性

代码示例：day2/process\_attr.py

[p.name](http://p.name/)获取进程名称，可以更改

p.pid 获取对应子进程的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()

### **进程池实现(进程复用)**

代码示例：day2/pool.py

#### **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**

管道通信 ，消息队列，共享内存 ，本地套接字 ，信号量 ，

### **管道通信(Pipe)**

代码示例：day2/pipe.py

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

1. 实现方法

from multiprocessing import Pipe

**fd1,fd2 = Pipe(duplex = True)**

功能: 创建管道

参数：默认为True 表示双向管道（指的是管道两边都可以进行读写操作）

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：day2/queue\_0.py

**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(阻塞状态)

**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

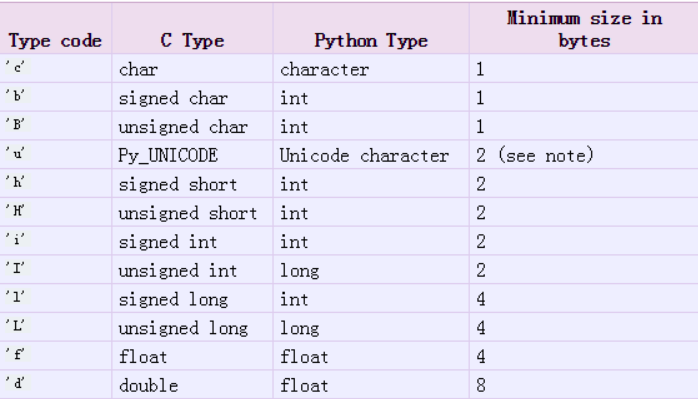
q.empty() 判断队列是否为空

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：day3/value.py  
代码示例：day3/array.py

1. 通信原理：在内存中开辟一块空间，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

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' 'f' 'c'

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

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**

### **本地套接字**

代码示例：day2/unix\_recv.py  
代码示例：day2/unix\_send.py

1. 功能 ： 用于本地两个程序之间进行数据的收发
2. 套接字文件 ：用于本地套接字之间通信时，进行数据传输的介质。
3. 创建本地套接字流程

【1】 创建本地套接字

sockfd = socket(AF\_UNIX,SOCK\_STREAM)

【2】 绑定本地套接字文件

sockfd.bind(file)

【3】 监听，接收客户端连接，消息收发

listen()-→accept()-→recv(),send()

**注意点：**

如果父进程中打开文件，创建进程通信对象或者创建套接字，子进程从父进程内存空间获取这些内容，那么父子进程对该对象的操作会有一定的属性关联。

### **信号量（信号灯集）**

代码示例：day2/sem.py

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。

1. 实现方法

from multiprocessing import Semaphore

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(1/20)**  
   【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：day3/thread1.py  
代码示例：day3/thread2.py

【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：day3/thread\_attr.py

[t.name](http://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：day3/myThread.py

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**

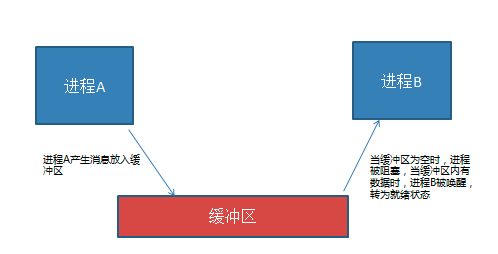
线程间使用**全局变量**进行通信(同一进程里线程间的通信)

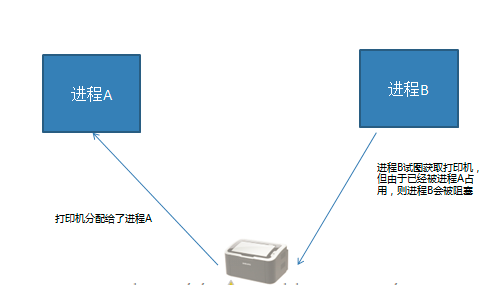
1. **共享资源争夺**

* 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。

1. **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：day3/thread\_event.py

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：day3/thread\_lock.py

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

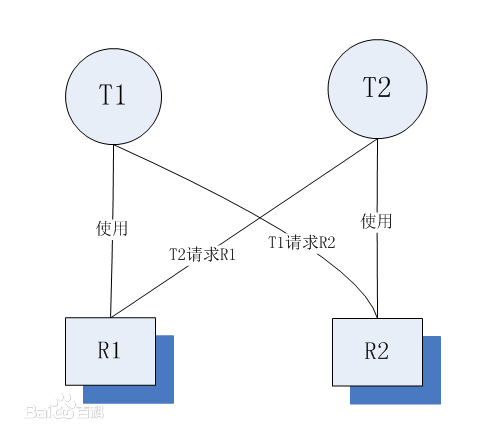
...

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ss下去。此时称系统处于死锁状态或系统产生了死锁。



1. 死锁产生条件

代码示例: day4/dead\_lock.py  
代码示例: day4/dead\_lock1.py

**1.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。

**2.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。

* **使用定时锁**
* 使用**重入锁RLock()**,用法同Lock。RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次require。直到一个线程所有的acquire都被release，其他的线程才能获得资源。

## **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#）

1. 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. 进程空间多里，数据互不干扰，有**专门通信方法**；线程使用**局部变量**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**：多中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. **难以程度**：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。

### **面试要求**

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**

### **常见模型分类**

#### **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．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。

#### **３．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。

### **基于fork的多进程网络并发模型**

代码实现: day4/fork\_server.py

#### **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: day4/thread\_server.py

#### **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

### **@@扩展：集成模块完成多进程/线程网络并发**

代码实现: day4/sock\_server.py

1. 使用方法

import socketserver

通过模块提供的不同的类的组合完成多进程或者多线程，tcp或者udp的网络并发模型

1. 常用类说明

TCPServer 创建tcp服务端套接字  
UDPServer 创建udp服务端套接字

StreamRequestHandler 处理tcp客户端请求  
DatagramRequestHandler 处理udp客户端请求

ForkingMixIn 创建多进程并发  
ForkingTCPServer ForkingMixIn + TCPServer  
ForkingUDPServer ForkingMixIn + UDPServer

ThreadingMixIn 创建多线程并发  
ThreadingTCPServer ThreadingMixIn + TCPServer  
ThreadingUDPServer ThreadingMixIn + UDPServer

1. 使用步骤

【1】 创建服务器类，通过选择继承的类，决定创建TCP或者UDP，多进程或者多线程的并发服务器模型。

【2】 创建请求处理类，根据服务类型选择stream处理类还是Datagram处理类。重写handle方法，做具体请求处理。

【3】 通过服务器类实例化对象，并绑定请求处理类。

【4】 通过服务器对象，调用serve\_forever()启动服务

## **IO并发**

### **IO 分类**

IO分类：阻塞IO ，非阻塞IO，IO多路复用，异步IO（协程）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（立即设置为非阻塞）

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 （等待一段时间设置为非阻塞）

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: day5/block\_io.py

### **ftp 文件服务器**

代码实现: day4/ftp

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多路复用**

1. 定义

同时监控多个IO事件(前提：IO互不影响)，当哪个IO事件准备就绪就执行哪个IO事件(计算机内核可以获取)。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。

1. 具体方案

select方法 ： windows linux unix  
poll方法： linux unix  
epoll方法： linux

#### **select 方法**

代码实现: day5/select\_server.py

**import select**

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(eg:accept(),recv())

wlist 列表 存放关注的**要主动处理**的IO事件(eg:sendto(),recv())

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

可以用按位运算可以知道某一类属性

运算符号：

& 按位与

| 按位或

^ 按位异或

<< 左移

>> 右移

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: day5/poll\_server.py

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）　　　返回值：１

POLLOUT 写IO事件 (wlist) 　返回值：４

POLLERR 异常IO （xlist） 　返回值：８

POLLHUP 断开连接 　返回值：１６

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

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: day5/epoll\_server.py

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 poll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。是为非抢占式多任务产生子程序的计算机组件。协程允许不同入口点在不同位置暂停或开始，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栈帧，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**

代码示例：day5/async\_test.py

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的协成调。

* 同步是指完成事务的逻辑，先执行第一个事务，如果阻塞了，会一直等待，直到这个事务完成，再执行第二个事务，顺序执行。
* 异步是和同步相对的，异步是指在处理调用这个事务的之后，不会等待这个事务的处理结果，直接处理第二个事务去了，通过状态、通知、回调来通知调用者处理结果。

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: day6/greenlet\_tast.py

* 安装 ： sudo pip3 install greenlet
* 函数

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

1. gevent模块

示例代码: day6/gevent\_test.py  
示例代码: day6/gevent\_server.py

* 安装：sudo pip3 instll gevent
* 函数

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**

day6/httpserver2.0.py

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】 将数据内容形参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

# code

## 五月九日（day01）

### １．fork的演示１

**import** os

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

pid = os.fork()

**if** pid < 0: *# 这里的pid为fork函数的返回值，并不是系统给的pid值*

print(**"Create process failed"**)

**elif** pid == 0:

print(pid)

sleep(3)

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

**else**:

print(pid)

sleep(1)

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

print(**"Fork test over"**)

*# 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*

### ２．fork的演示２

**import** os

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

a = 1

pid = os.fork()

**if** pid < 0:

print(**"Error"**)

**elif** pid == 0:

print(**"Child process"**)

print(**"a = %d"** % a)

a = 10000

**else**:

print(**"Parent process"**)

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

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

**"""**

**=========================**

**Parent process**

**a: 1**

**All a= 1**

**Child process**

**a = 1**

**All a= 10000**

**# 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**

**"""**

*# 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*

### ３．获取进程PID

**import** os

pid = os.fork()

**if** pid < 0:

print(**"Error"**)

**elif** pid == 0:

print(**"Child PID"**, os.getpid()) *# 获取一个进程的PID值*

print(**"Get Parent PID"**, os.getppid()) *# 获取父进程的PID号*

**else**:

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

print(**"Get Child PID"**, pid)

**"""**

**Parent PID 12290**

**Get Child PID 12291**

**Child PID 12291**

**Get Parent PID 12290**

**"""**

### ４．退出进程

**import** os

**import** sys

*# os.\_exit(1)# 进程退出*

sys.exit(**"进程退出"**) *# 可以打印字符串*

print(**"Process exit"**)

### ５．僵尸进程的演示以及wait的处理

*# 僵尸进程的演示以及wait的处理*

**import** os

pid = os.fork()

**if** pid < 0:

print(**"Error"**)

**elif** pid == 0:

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

os.\_exit(2) *# 系统会＊256*

**else**:

*# pid,status=os.wait() # 等待处理僵尸*

pid, status = os.waitpid(-1, os.WNOHANG) *# 非阻塞*

print(**"pid:"**, pid) *# 退出的子进程的PID*

print(**"status:"**, os.WEXITSTATUS(status)) *# 子进程退出状态*

**while True**:

**pass**

**"""**

**pid: 0**

**status: 0**

**Child process: 12734**

**"""**

### ６．**创建二级子进程处理僵尸**

*# 创建二级子进程防止僵尸进程*

**import** os

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

**def** f1():

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

sleep(2)

print(**"写代码......"**)

**def** f2():

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

sleep(1)

print(**"测代码......"**)

pid = os.fork()

**if** pid < 0:

print(**"Error"**)

**elif** pid == 0:

p = os.fork() *# 二级子进程*

**if** p == 0:

f2() *# 二级子进程执行*

**else**:

os.\_exit(0)

**else**:

os.wait()

f1()

**"""**

**测代码......**

**测代码......**

**写代码......**

**测代码......**

**写代码......**

**测代码......**

**测代码......**

**写代码......**

**写代码......**

**"""**

### ７．**通过信号处理子进程退出**

*# 信号方法处理僵尸*

**import** os

**import** signal

*# 处理子进程退出*

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

pid = os.fork()

**if** pid < 0:

**pass**

**elif** pid == 0:

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

**else**:

**while True**:

**pass**

*#Child PID: 14080*

### ８．**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

１．技术点分析

＊转发模型，客户端－－＞服务端－－＞转发给其他客户端

＊网络模型，UDP通讯

＊保存用户信息　　[(name,addr),()] {name:addr}

＊收发关系处理：采用多进程分别进行收发操作

２．结构设计

＊采用什么样的封装结构：都可以，这里采用函数封装

＊编写一个模块，测试一个功能

＊注意注释和结构的设计

３．分析功能模块，制定具体编写流程

＊搭建网络连接

＊进入聊天室

客户端：

＊输入姓名

＊将姓名发送给服务器

＊接收服务器返回的结果

＊如果不允许则重复输入姓名

服务端：

＊接收姓名

＊判断姓名是否存在

＊将结果给客户端

＊如果允许进入聊天室则增加用户信息

＊通知其他用户

＊聊天

客户端：

＊创建新的进程

＊一个进程循环发送消息

＊一个进程循环接收消息

服务端：

＊接收请求，判断请求类型

＊将消息转发给其他用户

＊退出聊天室

客户端：

＊输入quit或者ctrl+c退出

＊将请求发送给服务器

＊结束进程

＊接收端收到EXIT退出进程

服务端：

＊接收消息

＊将退出消息告知其他人

　＊给该用户发送“EXIT”

　　＊删除用户

＊管理员消息

４．协议

＊如果允许进入聊天室，服务端发送'ＯＫ'

＊如果不允许进入聊天室，服务端发送　不允许的原因

＊请求类别

L-->进入聊天室

C-->聊天信息

Q-->退出聊天室

＊用户存储结构：{name:addr ......}

＊客户端如果输入quit或ctrl+c，点击esc表示退出

**代码实现：**

**服务端：**

*"""*

*群聊聊天室服务端*

*Char room*

*env:python3.6*

*socket fork 练习*

*"""*

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

**import** os, sys

*# 服务器地址*

ADDR = (**'0.0.0.0'**, 3189)

*# 存储用户信息*

user = {}

*# 进入聊天室*

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

**if** name **in** user **or "管理员" in** name **or "quit" in** name:

s.sendto(**"该用户已存在或违规"**.encode(), addr)

**return**

s.sendto(**b'OK'**, addr)

*# 循环通知其他人*

msg = **"欢迎%s进入聊天室"** % name

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

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

*# 将用户加入*

user[name] = addr

*# 聊天*

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

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

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

**if** i != name:

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

*# 退出*

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

msg = **"%s退出了聊天室"** % name

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

**if** i != name:

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

**else**:

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

*# 将用户删除*

**del** user[name]

*# 接收各种客户端请求*

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

**while True**:

data, addr = s.recvfrom(1024)

msg = data.decode().split(**' '**)

*# 区分请求类型*

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

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

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

text = **' '**.join(msg[2:]) *# 假如内容有空格*

do\_char(s, msg[1], text)

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

do\_quit(s, msg[1])

*# 创建网络连接*

**def** main():

*# 套接字*

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

s.bind(ADDR)

pid = os.fork()

**if** pid < 0:

**return**

*# 发送管理员消息*

**elif** pid == 0:

**while True**:

msg = input(**"管理员消息："**)

msg = **"C 管理员消息** **"** + msg

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

**else**:

*# 请求处理*

do\_request(s) *# 消息的收发处理*

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

main()

**客户端：**

*"""*

*群聊聊天室客户端*

*"""*

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

**import** os, sys

*# 服务器地址*

ADDR = (**'176.140.6.137'**, 3189)

*# 循环发送消息*

**def** send\_msg(s, name):

**while True**:

**try**:

test = input() *# 消息内容*

**except** KeyboardInterrupt:

test = **'quit'**

*# 退出聊天室*

**if** test == **'quit'**:

msg = **"Q"** + name

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

sys.exit(**"退出聊天室"**)

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

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

*# 循环接收消息*

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

**while True**:

data, addr = s.recvfrom(2048)

*# 服务端发送EXIT表示让客户端退出*

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

sys.exit()

print(data.decode())

*# 创建网络连接*

**def** main():

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

**while True**:

name = input(**"请输入姓名："**)

msg = **"L "** + name

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

*# 等待回应*

data, addr = s.recvfrom(1024)

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

print(**"你已进入聊天室"**)

**break**

**else**:

print(data.decode())

*# 创建新的进程*

pid = os.fork()

**if** pid < 0:

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

**elif** pid == 0:

send\_msg(s, name)

**else**:

recv\_msg(s)

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

main()

## 五月十日（day02）

### １．**进程创建方法**

#### １．１演示（创建单个进程）

*# multiprocess事例*

*# 可以方便的创建多个子进程*

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

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

a = 1 *# 全局变量*

**def** fun():

print(**"子进程开始执行"**)

**global** a

a = 10000

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

*# sleep(3)*

print(**"子进程执行完毕"**)

*# 创建进程对象*

p = mp.Process(target=fun)

*# 启动进程*

p.start() *# 执行子进程*

sleep(2) *# 执行父进程*

print(**"父进程执行ing"**)

*# 回收进程*

p.join()

print(**"parent a"**, a)

**"""**

**子进程开始执行**

**a = 10000**

**子进程执行完毕**

**父进程执行ing**

**parent a 1**

**"""**

#### １．２演示（创建多个进程）

*# 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():

sleep(2)

print(**"睡觉"**)

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

**def** th3():

sleep(4)

print(**"撩妹"**)

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

things = [th1, th2, th3]

jobs = []

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

p = Process(target=th)

jobs.append(p)

p.start()

*# 防止僵尸进程产生*

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

i.join()

**"""**

**睡觉**

**1727 --- 1729**

**吃饭**

**1727 --- 1728**

**撩妹**

**1727 --- 1730**

**"""**

#### １．３演示（带参数的函数）

**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..."**)

*# p = Process(target=worker,args=(2,'Baron')) # 给target函数位置传参(元组)*

*# p = Process(target=worker,kwargs={'name':"Abby",'sec':2}) # 给target函数键值传参(字典)*

p = Process(target=worker, args=(2,), kwargs={**'name'**: **"Abby"**}) *# 两个一起使用（元组＋字典）*

p.start()

p.join()

**"""**

**I'm Abby**

**I'm working...**

**I'm Abby**

**I'm working...**

**I'm Abby**

**I'm working...**

**"""**

### ２．进程对象属性

*# 进程对象属性*

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

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

**def** tm():

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

sleep(2)

print(ctime())

p = Process(target=tm, name=**'Mr.zuo'**)

p.daemon = **True** *# 设置父子进程的退出关系 父进程一退出，所有子进程也会退出*

p.start()

print(**'Name:'**, p.name) *# 获取名称，可以更改*

print(**'PID:'**, p.pid) *# 获取PID*

print(**'is Alive:'**, p.is\_alive()) *# 查看子进程是否在生命周期*

*# p.join()*

**"""**

**Name: Mr.zuo**

**PID: 2849**

**is Alive: True**

**"""**

### ３．进程池实现

*# 进程池实现*

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

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

*# 进程池事件*

**def** worker(msg):

sleep(2)

print(msg)

*# 创建进程池*

pool = Pool(2) *# 不写数据会按系统默认(4个)分配进程数量*

*# 向进程池添加事件*

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

msg = **'Hello %d'** % i

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

*# 关闭进程池*

pool.close()

*# 回收进程池*

pool.join()

**"""**

**Hello 0**

**Hello 1**

**Hello 3**

**Hello 2**

**Hello 4**

**Hello 5**

**Hello 6**

**Hello 7**

**Hello 8**

**Hello 9**

**"""**

### ４．管道通信

*# 管道通信(Pipe)*

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

**import** os, time

*# 创建管道*

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

**def** fun(name):

time.sleep(3)

*# 向管道写入内容*

fd2.send({name: os.getpid()})

jobs = []

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

p = Process(target=fun, args=(i,))

jobs.append(p)

p.start()

*# 父进程读取*

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

*# 读取管道*

data = fd1.recv()

print(data)

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

i.join()

**"""**

**{1: 4298}**

**{0: 4297}**

**{2: 4299}**

**{3: 4300}**

**{4: 4301}**

**"""**

### ５．消息队列

*# 消息队列通信*

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

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

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

*# 创建消息队列*

q = Queue(3)

**def** request():

**for** i **in** range(20):

x = randint(0, 100)

y = randint(0, 100)

q.put((x, y)) *# 存入*

**def** handle():

**while True**:

sleep(0.5)

**try**:

x, y = q.get(timeout=3) *# 取出*

**except**:

**break**

**else**:

print(**"%d + %d = %d"** % (x, y, x + y))

p1 = Process(target=request)

p2 = Process(target=handle)

p1.start()

p2.start()

p1.join()

p2.join()

**"""**

**97 + 4 = 101**

**40 + 50 = 90**

**8 + 71 = 79**

**92 + 42 = 134**

**70 + 58 = 128**

**81 + 6 = 87**

**66 + 33 = 99**

**78 + 33 = 111**

**2 + 96 = 98**

**......**

**"""**

### ６．共享内存

#### ６．１Array

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

*# 创建共享内存*

*# 共享内存开辟５个整数列表空间*

*# shm = Array('i',5)*

*# 共享内存初始化数据[1,2,3]*

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

*# 字节串*

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

**def** fun():

*# 共享内存对象可迭代*

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

print(i)

*# 修改共享内存*

*# shm[1]=1000*

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

p = Process(target=fun)

p.start()

p.join()

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

print(i, end=**' '**)

*# 通过value属性访问字节串*

print(shm.value)

**"""**

**b'h'**

**b'e'**

**b'l'**

**b'l'**

**b'o'**

**b'H' b'e' b'l' b'l' b'o' b'Hello'**

**"""**

#### ６．２Value

*# 共享内存*

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

**import** time

**import** random

*# 创建共享内存*

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

*# 操作共享内存*

**def** man():

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

time.sleep(0.2)

money.value += random.randint(1,1000)

**def** girl():

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

time.sleep(0.15)

money.value -= random.randint(100,800)

m = Process(target= man)

g = Process(target=girl)

m.start()

g.start()

m.join()

g.join()

*# 获取共享内存值*

print(**"本月余额："**,money.value)

**"""**

**本月余额：** **7013 #数据不一定**

**"""**

### ７．本地套接字

#### ７．１本地套接字接收端

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

**import** os

*# 确定本地套接字文件*

sock\_file = **"./sock"** *# 两个程序之间通过创建的一个sock套接字文件对数据进行收发*

*# 判断文件是否存在，存在就删除*

**if** os.path.exists(sock\_file):

os.remove(sock\_file)

*# 创建本地套接字*

sockfd=socket(AF\_UNIX, SOCK\_STREAM)

*# 绑定本地套接字*

sockfd.bind(sock\_file)

*# 监听*

sockfd.listen(3)

**while True**:

c,addr=sockfd.accept()

**while True**:

data = c.recv(1024)

**if not** data:

**break**

print(data.decode())

c.close()

sockfd.close()

#### ７．２本地套接字发送端

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

**import** os

*# 确认两边使用相同的套接字文件*

sock\_file = **"./sock"** *# 两个程序之间通过创建的一个sock套接字文件对数据进行收发*

sockfd = socket(AF\_UNIX, SOCK\_STREAM)

sockfd.connect(sock\_file)

*# 循环发送*

**while True**:

msg = input(**">>"**)

**if not** msg:

**break**

sockfd.send(msg.encode())

sockfd.close()

### ８．信号量（信号灯集）

*# 信号量（信号灯集）*

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

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

**import** os

*# 创建信号量*

*# 表示**服务程序最多允许３个进程同时执行事件*

sem = Semaphore(3)

**def** handle():

print(**"%d 想执行事件"** % os.getpid())

*# 想执行事件必须获取信号量*

sem.acquire()

print(**"%d 开始执行操作"** % os.getpid())

sleep(3)

print(**"%d 完成操作"** % os.getpid())

sem.release() *# 增加信号量*

jobs = []

*#* ５*个进程请求执行事件*

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

p = Process(target=handle)

jobs.append(p)

p.start()

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

i.join()

print(sem.get\_value()) *# 获取信号量*

**"""**

**6790 想执行事件**

**6790 开始执行操作**

**6794 想执行事件**

**6794 开始执行操作--------->Semaphore(3)**

**6792 想执行事件　　　　　　　服务程序最多允许３个进程同时执行事件**

**6792 开始执行操作**

**6791 想执行事件　--------->等待执行**

**6793 想执行事件　--------->等待执行**

**6790** 完成操作　　**--------->完成一个**

**6791** 开始执行操作**--------->执行一个**

**6794 完成操作**

**6792 完成操作**

**6793 开始执行操作**

**6791 完成操作**

**6793 完成操作**

**3**

**"""**

### ９．**exercise**

1. multiprocess创建两个进程，同时复制一个文件的上下两半部分，各自复制到一个新的文件里

代码实现：

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

**import** os

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

filename = **"test"**

*# 　获取文件大小*

size = os.path.getsize(filename)

*# 　父进程打开文件*

*# 　文件偏移量会相互影响*

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

*# 　复制上半部分*

**def** top():

sleep(1)

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

n = size // 2

fw = open(**"top"**, **'wb'**)

fw.write(f.read(n))

f.close()

fw.close()

*# 　下半部分*

**def** bot():

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

fw = open(**'bot'**, **'wb'**)

f.seek(size // 2, 0)

**while True**:

data = f.read(1024)

**if not** data:

**break**

fw.write(data)

f.close()

fw.close()

t = Process(target=top)

b = Process(target=bot)

t.start()

b.start()

t.join()

b.join()

## 五月十三日（day03）

### １．**线程编程（Thread）--threading模块创建线程**

* + 1. *"""*

*线程事例一*

*"""*

**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(5):

sleep(2)

print(os.getpid(), **"播放：爱的供养"**) *# pid一样,说明都同属同一个线程*

*# 创建线程对象*

t = threading.Thread(target=music)

t.start()

*# 主线程任务*

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

sleep(3)

print(os.getpid(), **"播放：学猫叫"**)

t.join()

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

**"""**

**a= 1**

**17533 播放：爱的供养**

**17533 播放：学猫叫**

**17533 播放：爱的供养**

**17533 播放：爱的供养**

**17533 播放：学猫叫**

**17533 播放：爱的供养**

**17533 播放：学猫叫**

**17533 播放：爱的供养**

**Main thread a: 10000**

**"""**

### ２．线程的传参

* + 1. *# 线程的传参*

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

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

*# 含有参数的线程函数*

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

print(**"线程函数传参"**)

sleep(sec)

print(**"%s 线程执行完毕"** % 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)

t.start()

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

i.join()

**"""**

**线程函数传参**

**线程函数传参**

**线程函数传参**

**线程函数传参**

**线程函数传参**

**T0 线程执行完毕**

**T1 线程执行完毕**

**T3 线程执行完毕**

**T2 线程执行完毕**

**T4 线程执行完毕**

**"""**

### ３．线程的属性

* + 1. *# 线程属性*

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

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

**def** fun():

sleep(3)

print(**"线程属性测试"**)

t=Thread(target=fun,name=**'Mr.zuo'**)

*# 主线程退出分支线程也退出*

t.setDaemon(**True**)

t.start()

*# 线程名称*

t.setName(**'tarena'**)*# 设置线程名称(也可以在参数传入修改)*

print(**'Thread name:'**,t.getName())*# 获取线程名称*

*# 线程生命周期*

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

*#Thread name: tarena*

*#is alive: True 没有t.start(),所以周期为True*

### ４．自定义线程类

* + 1. **from** threading **import** Thread

**class** ThreadClass(Thread):

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

self.attr = attr

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

*# 多个方法配合实现具体功能*

**def** f1(self):

print(**"步骤1"**)

**def** f2(self):

print(**"步骤2"**)

**def** run(self):

self.f1()

self.f2()

t = ThreadClass(**"xxx"**)

t.start() *# 自动运行run方法*

t.join()

**"""**

**步骤1**

**步骤2**

**"""**

### ５．**线程同步互斥方法****－－－线程Event**

* + 1. **from** threading **import** Thread, Event

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

s = **None** *# 全局变量*

e = Event() *# 创建Event对象*

**def** 座山雕():

print(**"杨子荣前来拜山头"**)

**global** s

s = **"天王盖地虎"**

e.set() *# 共享资源操作完毕　　阻塞等待e被set*

t = Thread(target=座山雕)

t.start()

print(**"口令"**)

e.wait(3) *# 阻塞等待*

**if** s == **"天王盖地虎"**:

print(**"宝塔镇河妖"**)

print(**"来了老弟"**)

**else**:

print(**"弄死他"**)

t.join()

### ６．**线程同步互斥方法****－－－线程锁 Lock**

* + 1. **from** threading **import** Thread, Lock

a = b = 0

lock = Lock() *# 创建锁对象*

**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: *# with代码块结束自动解锁*

a += 1

b += 1

t.join()

### ７．**死锁及其处理－－－事例一**

* + 1. *# 死锁事例一*

**import** time

**import** threading

*#交易类*

**class** Account:

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

self.id=\_id *# 用户*

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

*# 查看账户金额*

**def** get\_balance(self):

**return** self.balance

*# 转账函数*

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

*# 上锁成功返回True*

**if** from\_.lock.acquire(): *# 锁住自己的账户*

from\_.withdraw(amount) *# 自己账户金额减少*

time.sleep(0.5) *# 造成死锁*

**if** to.lock.acquire(): *# 对方账户上锁*

to.deposit(amount)*# 对方账户金额增加*

to.lock.release() *# 对方账户解锁*

from\_.lock.release() *# 自己账户解锁*

print(**"转账成功"**)

*# 创建两个账户（两个线程）*

Abby = Account(**'Abby'**,5000,threading.Lock())

Levi = Account(**'Levi'**,3000,threading.Lock())

*# 转账*

t01 = threading.Thread(target=transfer,args=(Abby,Levi,1500))

t02 = threading.Thread(target=transfer,args=(Levi,Abby,1000))

t01.start()

time.sleep(2) *# 处理死锁(确保同一时刻只有一个线程在运行时)*

t02.start()

t01.join()

t02.join()

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

print(**'Levi:'**,Levi.get\_balance())

### ８．**死锁及其处理－－－事例二**

* + 1. *# 死锁事例二*

*# 重复上锁造成死锁*

**from** threading **import** Thread,RLock

**import** time

num = 0 *# 共享资源*

lock = RLock() *# 解决方法,使用重载锁,可以允许重复上锁*

**class** MyThread(Thread):

**def** fun01(self):

**global** num

**with** lock:

num -= 1

**def** fun02(self):

**global** num

**if** lock.acquire():

num += 1

**if** num >5: *# 当条件num大于５时，本身上锁的情况下*

self.fun01()*# 又会调用fun01的with lock上锁，从而造成死锁*

print(**"Num = "**,num)

lock.release()

**def** run(self):

time.sleep(2)

self.fun02()

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

t = MyThread()

t.start()

t.join()

### **９．练习**

* + 1. *# 练习*

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

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

**class** MyThread(Thread):

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

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

self.target=target

self.args=args

self.kwargs=kwargs

self.name=name

**def** run(self):

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

*#--------------------------------*

*#通过完成上面的MyThread类让整个程序可以正常执行*

*#当调用start和player作为一个线程功能函数运行*

*#注意,函数名称和参数并不确定,player只是测试函数*

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

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

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

sleep(sec)

t = MyThread(target=player,args=(3,),kwargs={**'song'**:**'凉凉'**},name=**'happy'**)

t.start()

t.join()

*#Playing 凉凉:Mon May 13 19:41:31 2019*

*#Playing 凉凉:Mon May 13 19:41:34 2019*

### **１０．ｅｘｅｒｃｉｓｅ**

#### １０．４．多进程

**from** test **import** \*

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

**import** time

jobs = []

st = 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(**"Process io:"**, time.time() - st)

*# Process io: 4.49703574180603*

#### １０．１．编写程序完成效率测试

*"""*

*编写程序完成效率测试*

*使用单线程执行计算密集型函数10次，记录时间，执行io密集型函数10次，记录时间*

*使用10个线程分别执行计算密集型函数１次，记录时间，执行io密集型函数１次，记录时间*

*使用10个进程执行计算密集型函数１次，记录时间，执行io密集型函数１次，记录时间*

*"""*

*#　计算密集型函数* *x y 传入１，１*

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

c = 0

**while** c < 7000000:

c += 1

x += 1

y += 1

*#　io密集型*

**def** io():

write()

read()

**def** write():

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

**for** i **in** range(1500000):

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

f.close()

**def** read():

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

lines = f.readlines()

f.close()

#### １０．２．单线程

**from** test **import** \*

**import** time

st = time.time()

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

*# count(1,1)*

io()

print(**"Single IO:"**,time.time() - st)

*# Single IO: 4.304931640625*

#### １０．３．多线程

**from** test **import** \*

**import** threading

**import** time

jobs = []

st = time.time()

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

t = threading.Thread(target=io)

jobs.append(t)

t.start()

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

i.join()

print(**"Thread io:"**, time.time() - st)

*# Thread io: 8.935524225234985*

## 五月十四日（day04）

### １．**基于fork的多进程网络并发模型**

#### １．１服务端

*"""*

*基于fork的多进程网络并发模型*

*重点代码*

*"""*

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

**import** os,sys

**import** signal

*# 创建监听套接字*

HOST = **"0.0.0.0"**

PORT = 8888

ADDR = (HOST,PORT)

s = socket()*# TCP套接字*

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

s.bind(ADDR)

s.listen(3)

*# 僵尸进程的处理*

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

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

*# 客户端处理函数*

**def** handle(c):

print(**"客户端："**,c.getpeername())

**while True**:

data = c.recv(1024)

**if not** data:

**break**

print(data.decode())

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

c.close()

*# 循环等待客户端连接*

**while True**:

**try**:

c,addr = s.accept()

**except** KeyboardInterrupt:

sys.exit(**"服务器退出"**)

**except** Exception **as** e:*# 其他异常打印出来*

print(e)

**continue**

*# 创建子进程处理客户端请求*

pid = os.fork()

**if** pid == 0:

s.close()*# 子进程不需要c*

handle(c)*# 具体处理客户端请求*

os.\_exit(0)

*# 父进程实际只用来处理客户端的连接*

**else**:

c.close()*# 父进程不需要c*

#### １．２客户端（２．３通用）

*"""*

*TCP套接字客户端*

*重点代码*

*"""*

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

*# 创建套接字*

sockfd = socket()

*# 请求连接*

server\_addr = (**'176.140.6.137'**, 8888) *# 访问本机主机IP地址　端口*

sockfd.connect(server\_addr)

*# 消息收发*

**while True**:

data = input(**"消息："**)

**if not** data: *# 结束*

**break**

sockfd.send(data.encode()) *# 发送消息*

data = sockfd.recv(1024) *# 接收消息*

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

*# 关闭套接字*

sockfd.close()

### ２．基于threading的多线程网络并发

*# 基于threading的多线程网络并发 （服务端） 重点代码*

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

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

**import** sys

*# 创建监听套接字*

HOST = **"0.0.0.0"**

PORT = 3189

ADDR = (HOST, PORT)

s = socket()

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

s.bind(ADDR)

s.listen(3)

*# 客户端处理函数*

**def** handle(c):

print(**"Connect from"**, c.getpeername())

**while True**:

data = c.recv(1024)

**if not** data:

**break**

print(data.decode())

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

c.close()

*# 循环等待客户端连接*

**while True**:

**try**:

c, addr = s.accept()

**except** KeyboardInterrupt:

sys.exit(**"退出服务器"**)

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

print(e)

**continue**

*# 创建新的线程处理客户端请求*

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

t.setDaemon(**True**) *# 如果用join退出子线程主线程也会退出*

t.start()

### ３．基于multiprocessing的多进程网络并发

*# 基于multiprocessing的多进程网络并发 重点代码*

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

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

**import** sys

**import** signal

*# 创建监听套接字*

HOST = **"0.0.0.0"**

PORT = 3189

ADDR = (HOST, PORT)

s = socket()

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

s.bind(ADDR)

s.listen(3)

*# 僵尸进程的处理*

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

*# 客户端处理函数*

**def** handle(c):

print(**"Connect from"**, c.getpeername())

**while True**:

data = c.recv(1024)

**if not** data:

**break**

print(data.decode())

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

c.close()

*# 循环等待客户端连接*

**while True**:

**try**:

c, addr = s.accept()

**except** KeyboardInterrupt:

sys.exit(**"退出服务器"**)

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

print(e)

**continue**

*# 创建新的线程处理客户端请求*

p = Process(target=handle, args=(c,))

p.daemon = **True** *# 如果用join退出子线程主线程也会退出*

p.start()

### ４．**ftp 文件服务器**

#### **４．１．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

#### **４．２．思**路**分析**

１．技术分析点

＊并发模型　　多线程并发模型

＊数据传输　　tcp传输

２．结构设计

＊客户端发起请求，打印请求提示界面

＊文件传输功能封装为类

３．功能分析

＊网络搭建

＊查看文件库信息

＊下载文件

＊上传文件

＊客户端退出

4.协议

L 表示请求文件列表

Q 表示退出

G 表示下载

P 表示上传

#### **４．３．服务端代码实现**

*"""*

*ftp　文件服务器*

*并发网络功能训练*

*"""*

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

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

**import** os

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

*# 　全局变量*

HOST = **'0.0.0.0'**

PORT = 8080

ADDR = (HOST, PORT)

FTP = **"/home/tarena/ftp/"** *# 文件库路径*

*# 将客户端请求功能封装为类*

**class** FtpServer:

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

self.connfd = connfd

self.path = FTP\_PATH

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

*# 　获取文件列表*

files = os.listdir(self.path)

**if not** files:

self.connfd.send(**"该文件类别为空"**.encode())

**return**

**else**:

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

sleep(0.1)

fs = **''**

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

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

os.path.isfile(self.path + file):

fs += file + **'\n'**

self.connfd.send(fs.encode())

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

**try**:

fd = open(self.path + filename, **'rb'**)

**except** Exception:

self.connfd.send(**'文件不存在'**.encode())

**return**

**else**:

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

sleep(0.1)

*# 　发送文件内容*

**while True**:

data = fd.read(1024)

**if not** data: *# 文件结束*

sleep(0.1)

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

**break**

self.connfd.send(data)

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

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

self.connfd.send(**"该文件已存在"**.encode())

**return**

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

fd = open(self.path + filename, **'wb'**)

*# 接收文件*

**while True**:

data = self.connfd.recv(1024)

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

**break**

fd.write(data)

fd.close()

*# 客户端请求处理函数*

**def** handle(connfd):

*# 　选择文件夹*

cls = connfd.recv(1024).decode()

FTP\_PATH = FTP + cls + **'/'**

ftp = FtpServer(connfd, FTP\_PATH)

**while True**:

*# 接受客户端请求*

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

*# 　如果客户端断开返回ｄａｔａ为空*

**if not** data **or** data[0] == **'Q'**:

**return**

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

ftp.do\_list()

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

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

ftp.do\_get(filename)

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

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

ftp.do\_put(filename)

*# 网络搭建*

**def** main():

*# 　创建套接字*

sockfd = socket()

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

sockfd.bind(ADDR)

sockfd.listen(5)

print(**"Listen the port 8080..."**)

**while True**:

**try**:

connfd, addr = sockfd.accept()

**except** KeyboardInterrupt:

print(**"退出服务程序"**)

**return**

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

print(e)

**continue**

print(**"连接的客户端："**, addr)

*# 创建新的线程处理客户端请求*

client = Thread(target=handle, args=(connfd,))

client.setDaemon(**True**) *# 如果用join退出子线程主线程也会退出*

client.start()

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

main()

#### **４．４．客户端代码实现**

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

**import** sys

**import** time

*# 将客户端请求功能封装为类 具体功能*

**class** FtpClient:

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

self.sockfd = sockfd

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

self.sockfd.send(**b'L'**) *# 发送请求*

*# 等待回复*

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

*# 　ｏｋ表示请求成功*

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

*# 　接收文件列表*

data = self.sockfd.recv(4096)

print(data.decode())

**else**:

print(data)

**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()

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

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

*# 　接收内容写入文件*

**while True**:

data = self.sockfd.recv(1024)

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

**break**

fd.write(data)

fd.close()

**else**:

print(data)

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

**try**:

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

**except** Exception:

print(**"没有该文件"**)

**return**

*# 　发送请求*

filename = filename.split(**'/'**)[-1] *# 切割路径*

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:

time.sleep(0.1)

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

**break**

self.sockfd.send(data)

f.close()

**else**:

print(data)

*# 　发起请求*

**def** request(sockfd):

ftp = FtpClient(sockfd)

**while True**:

print(**"\n==========命令选项==========="**)

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

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

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

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

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

cmd = input(**"输入命令:"**)

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

ftp.do\_list()

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

ftp.do\_quit()

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

filename = cmd.strip().split(**' '**)[-1] *# 先去空格再取文件名*

ftp.do\_get(filename)

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

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

ftp.do\_put(filename)

*# 　网络链接*

**def** main():

*# 　服务器地址*

ADDR = (**'127.0.0.1'**, 8080)

sockfd = socket()

**try**:

sockfd.connect(ADDR)

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

print(**"链接服务器失败"**)

**return**

**else**:

print(**"""**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**data file image**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**"""**)

cls = input(**"请选择文件种类："**)

**if** cls **not in** [**'data'**, **'file'**, **'image'**]:

print(**"Sorry input Error!!"**)

**return**

**else**:

sockfd.send(cls.encode())

request(sockfd) *# 发送具体请求*

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

main()

## 五月十五日（day05）

### １．**IO并发－－套接字非阻塞示例**

*"""*

*套接字非阻塞示例*

*"""*

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

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

*# 日志文件*

f = open(**'log.txt'**, **'a+'**)

*# ｔｃｐ套接字(阻塞较多)*

sockfd = socket()

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

sockfd.listen(3)

*# 设置套接字为非阻塞*

*# sockfd.setblocking(False)*

*# 超时检测*

sockfd.settimeout(3)

**while True**:

print(**'Waiting for connect...'**)

**try**:

connfd, addr = sockfd.accept()

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

*# 每隔２ｓ写入一条日志*

sleep(2)

f.write(**'%s: %s\n'** % (ctime(), e))

f.flush()

**else**:

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

print(data)

*# Waiting for connect...*

*# Waiting for connect...*

*# Waiting for connect...*

*# Waiting for connect ...*

*# Waiting for connect...*

*# Waiting for connect...*

**"""**

**log.txt日志内容:**

**Wed May 15 18:59:55 2019: timed out**

**Wed May 15 19:00:00 2019: timed out**

**Wed May 15 19:00:05 2019: timed out**

**Wed May 15 19:00:10 2019: timed out**

**Wed May 15 19:00:15 2019: timed out**

**Wed May 15 19:00:20 2019: timed out**

**"""**

### ２．**IO多路复用－－ｓｅｌｅｃｔ方法实现多个客户端的通信**

**服务端实现代码：**

*"""*

*io多路复用select实现多个客户端通信*

*"""*

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

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

*# 设置套接字为关注IO*

s = socket()

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

s.bind((**'0.0.0.0'**, 9999))

s.listen(3)

*# 设置关注io*

rlist = [s]

wlist = []

xlist = []

**while True**:

*# 监控io的发生*

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

*# 遍历三个返回值列表，判断哪个io发生*

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

*# 如果套接字就绪则处理连接*

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

c, addr = r.accept()

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

rlist.append(c) *# 循环加入新的关注io*

**else**:

data = r.recv(1024)

**if not** data:

rlist.remove(r)

r.close()

**continue**

print(data.decode())

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

wlist.append(r) *# 主动处理这个ｉｏ*

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

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

wlist.remove(w)

**for** x **in** xs:

**pass**

**客户端实现代码：**

*"""*

*TCP套接字客户端*

*重点代码*

*"""*

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

*# 创建套接字*

sockfd = socket()

*# 请求连接*

server\_addr = (**'176.140.6.137'**, 9999) *# 访问本机主机IP地址　端口*

sockfd.connect(server\_addr)

*# 消息收发*

**while True**:

data = input(**"消息："**)

**if not** data: *# 结束*

**break**

sockfd.send(data.encode()) *# 发送消息*

data = sockfd.recv(1024) *# 接收消息*

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

*# 关闭套接字*

sockfd.close()

### ３．ｐｏｌｌ**方法实现多个客户端的通信**

**服务端实现代码：**

*"""*

*poll多路复用*

*次重点*

*"""*

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

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

*# 　设置套接字为关注ＩＯ*

s = socket()

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

s.bind((**'0.0.0.0'**, 9999))

s.listen(5)

*# 　创建ｐｏｌｌ*

p = poll()

*# 　建立查找字典* *{fileno: io\_obj} 文件描述符*

fdmap = {s.fileno(): s}

*# 　设置关注ＩＯ*

p.register(s, POLLIN | POLLERR)

print(**'Waiting connect...'**)

*# 　循环监控ｉｏ事件发生*

**while True**:

events = p.poll() *# 阻塞等待ＩＯ发生*

print(events)

*# 遍历列表处理ＩＯ*

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

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

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

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

*# 　添加新的关注事件*

p.register(c, POLLIN | POLLHUP)

fdmap[c.fileno()] = c

*# elif event & POLLHUP: #　客户端断开*

*# print("客户端退出")*

*# p.unregister(fd) #　取消关注*

*# fdmap[fd].close()*

*# del fdmap[fd] #　从字典删除*

**elif** event & POLLIN: *# 客户端发消息*

data = fdmap[fd].recv(1024)

*# 断开发生时data得到空此时POLLIN也会就绪*

**if not** data:

print(**"客户端退出"**)

p.unregister(fd) *# 取消关注*

fdmap[fd].close()

**del** fdmap[fd]

**continue**

print(data.decode())

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

客户端代码同２．

### ４．ｅｐｏｌｌ**方法实现多个客户端的通信**

**服务端实现代码：**

*"""*

*epoll多路复用*

*次重点*

*"""*

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

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

*# 　设置套接字为关注ＩＯ*

s = socket()

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

s.bind((**'0.0.0.0'**, 9999))

s.listen(5)

*# 　创建eｐｏｌｌ*

ep = epoll()

*# 　建立查找字典* *{fileno: io\_obj}*

fdmap = {s.fileno(): s}

*# 　设置关注ＩＯ*

ep.register(s, EPOLLIN | EPOLLERR)

print(**'Waiting connect...'**)

*# 　循环监控ｉｏ事件发生*

**while True**:

events = ep.poll() *# 阻塞等待ＩＯ发生*

print(events)

*# 遍历列表处理ＩＯ*

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

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

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

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

*# 　添加新的关注事件*

ep.register(c, EPOLLIN | EPOLLHUP | EPOLLET)

fdmap[c.fileno()] = c

**elif** event & EPOLLIN: *# 客户端发消息*

data = fdmap[fd].recv(1024)

*# 断开发生时data得到空此时EPOLLIN也会就绪*

**if not** data:

ep.unregister(fd) *# 取消关注*

fdmap[fd].close()

**del** fdmap[fd]

**continue**

print(data.decode())

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

客户端代码同２．

### ５．拓展知识－－协程技术

**import** asyncio

**import** time

now = **lambda**: time.time()

**async def** do\_work(x):

print(**'Waiting:'**, x)

**await** asyncio.sleep(x) *# 协程跳转*

**return "Done after %s s"** % x

start = now()

*# 协程对象*

cor1 = do\_work(1)

cor2 = do\_work(2)

cor3 = do\_work(3)

*# 将协程对象生成一个可轮循跳转的对象列表*

tasks = [asyncio.ensure\_future(cor1),

asyncio.ensure\_future(cor2),

asyncio.ensure\_future(cor3)]

*# 得到轮循对象调用run启动协程执行*

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(asyncio.wait(tasks))

print(**'Time:'**, now() - start)

**"""**

**Waiting: 1**

**Waiting: 2**

**Waiting: 3**

**Time: 3.0019924640655518**

**"""**

## 五月十六日（day06）

### １．ｇｒｅｅｎｌｅｔ模块－－第三方协程模

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

**def** test1():

print(**'执行test1'**)

gr2.switch()

print(**'结束test1'**)

gr2.switch()

**def** test2():

print(**'执行test2'**)

gr1.switch()

print(**'结束test2'**)

*# gr1.switch()*

*# 将函数变成协程函数*

gr1 = greenlet(test1)

gr2 = greenlet(test2)

gr1.switch() *# 执行协程test1*

**"""**

**执行test1**

**执行test2**

**结束test1**

**结束test2**

**"""**

### ２．ｇｅｖｅｎｔ模块

**import** gevent

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

print(**'Running foo...\n'**, a, b)

gevent.sleep(3)

print(**'Foo again'**)

**def** bar():

print(**'Running foo...\n'**)

gevent.sleep(3)

print(**'Bar again'**)

*# 将函数封装为协程，遇到gevent阻塞自动执行*

f = gevent.spawn(foo, **'hello'**, **'kitty'**)

g = gevent.spawn(bar)

gevent.joinall([f, g]) *# 阻塞等待协程执行完毕*

**"""**

**Running foo...**

**hello kitty**

**Running foo...**

**Foo again**

**Bar again**

**"""**

### ３．ｍｏｎｋｅｙ脚本

*"""*

*gevent协程演示*

*"""*

**import** gevent

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

monkey.patch\_all()

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

**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.bind((**'0.0.0.0'**, 3189))

s.listen(5)

**while True**:

c, addr = s.accept()

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

*# handle(c) # 循环方案*

gevent.spawn(handle, c) *# 协程方案*

s.close()

**"""**

**浏览器输入：IP:PORT**

**Connect from ('176.140.6.137', 45294)**

**GET / HTTP/1.1**

**Host: 176.140.6.137:3189**

**Connection: keep-alive**

**Upgrade-Insecure-Requests: 1**

**User-Agent: Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/73.0.3683.103 Safari/537.36**

**Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3**

**Accept-Encoding: gzip, deflate**

**Accept-Language: zh-CN,zh;q=0.9**

**"""**

### ４．**HTTPServer v2.0**

**服务端代码实现：**

*"""*

*http server v2.0*

*＊io 并发处理(连接多个客户端)*

*＊基本的request解析*

*＊使用类封装*

*"""*

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

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

*# 将具体http server功能封装*

**class** HttpServer:

**def** \_\_init\_\_(self, server\_address, static\_dir):

*# 添加属性*

self.server\_address = server\_address

self.static\_dir = static\_dir

self.rlist = [] *# 变成属性变量，这样每个函数就都能使用了*

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)

**def** bind(self):

self.sockfd.bind(self.server\_address)

self.ip = self.server\_address[0]

self.port = self.server\_address[1]

*# 启动服务*

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

self.sockfd.listen(5)

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

self.rlist.append(self.sockfd)

*# 循环监听*

**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()

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

self.rlist.append(c)

**else**:

*# 处理浏览器请求*

self.handle(r)

*# 处理客户端请求*

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

*# 接收http请求*

request = connfd.recv(4096)

**if not** request: *# 防止客户端断开*

self.rlist.remove(connfd)

connfd.close()

**return**

*# 请求解析*

request\_line = request.splitlines()[0].decode() *# 将request按行分割取第一行*

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

print(connfd.getpeername(), **':'**, info)

*# info　分为访问网页和其他*

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

self.get\_html(connfd, info)

**else**:

self.get\_data(connfd, info)

self.rlist.remove(connfd)

connfd.close()

*# 处理网页*

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

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

*# 网页文件*

filename = self.static\_dir + **'/index.html'**

**else**:

filename = self.static\_dir + info

**try**:

fd = open(filename) *# 打开网页文件*

**except** Exception:

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

responseHeaders += **'\r\n'**

responseBody = **"<h1>Sorry,Not Found the page</h1>"**

**else**:

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

responseHeaders += **'\r\n'** *# 换行*

responseBody = fd.read()

**finally**:

response = responseHeaders + responseBody

connfd.send(response.encode())

*# 其他情况*

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

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

responseHeaders += **'\r\n'**

responseBody = **"<h1>Waiting httpserver 3.0</h1>"**

response = responseHeaders + responseBody

connfd.send(response.encode())

*# 如何使用HttpServer类*

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

*# 用户自己决定：地址，内容*

server\_address = (**'0.0.0.0'**, 8888)

static\_dir = **"./static"** *# 网页存放位置/路径(当前路径的static文件夹)*

httpd = HttpServer(server\_address, static\_dir) *# 生成实例对象*

httpd.server\_forever() *# 启动服务(浏览器输入ip:port/网页即可启动服务)*