# 第二阶段

目录

[进程、线程、协程](#_Toc17089_WPSOffice_Level1) [2](#_Toc17089_WPSOffice_Level1)

[网络编程](#_Toc13405_WPSOffice_Level1) [5](#_Toc13405_WPSOffice_Level1)

[一、协议(网络协议)概念](#_Toc15816_WPSOffice_Level1) [5](#_Toc15816_WPSOffice_Level1)

[二、OSI模型和TCP / IP模型](#_Toc29025_WPSOffice_Level1) [5](#_Toc29025_WPSOffice_Level1)

[三、TCP/IP四层模型\*\*](#_Toc8185_WPSOffice_Level1) [6](#_Toc8185_WPSOffice_Level1)

[四、tcp / udp 传输协议](#_Toc19766_WPSOffice_Level1) [6](#_Toc19766_WPSOffice_Level1)

[1. tcp 和 udp 的特征](#_Toc13405_WPSOffice_Level2) [6](#_Toc13405_WPSOffice_Level2)

[2. tcp 和 udp 的区别](#_Toc15816_WPSOffice_Level2) [6](#_Toc15816_WPSOffice_Level2)

[3. tcp的应用场景](#_Toc29025_WPSOffice_Level2) [7](#_Toc29025_WPSOffice_Level2)

[4. udp的应用场景](#_Toc8185_WPSOffice_Level2) [7](#_Toc8185_WPSOffice_Level2)

[五、网络套接字编程](#_Toc6881_WPSOffice_Level1) [7](#_Toc6881_WPSOffice_Level1)

[1. 网络编程目的](#_Toc19766_WPSOffice_Level2) [7](#_Toc19766_WPSOffice_Level2)

[2. 套接字的分类：](#_Toc6881_WPSOffice_Level2) [7](#_Toc6881_WPSOffice_Level2)

[3. tcp套接字编程的基本步骤：](#_Toc28783_WPSOffice_Level2) [7](#_Toc28783_WPSOffice_Level2)

[4. udp 套接字编程的基本步骤：](#_Toc14341_WPSOffice_Level2) [7](#_Toc14341_WPSOffice_Level2)

[1. tcp粘包的产生条件](#_Toc15540_WPSOffice_Level2) [8](#_Toc15540_WPSOffice_Level2)

[2. tcp粘包的原因](#_Toc16322_WPSOffice_Level2) [8](#_Toc16322_WPSOffice_Level2)

[3. tcp粘包的处理](#_Toc22777_WPSOffice_Level2) [8](#_Toc22777_WPSOffice_Level2)

[1) 每次发送后加一个特殊结尾标志，接收端通过标志进行判断](#_Toc13405_WPSOffice_Level3) [8](#_Toc13405_WPSOffice_Level3)

[2) 发送一个数据结构](#_Toc15816_WPSOffice_Level3) [8](#_Toc15816_WPSOffice_Level3)

[3) 每次发送中间有一个短暂的延迟](#_Toc29025_WPSOffice_Level3) [8](#_Toc29025_WPSOffice_Level3)

[六、HTTP协议](#_Toc28783_WPSOffice_Level1) [8](#_Toc28783_WPSOffice_Level1)

[1. HTTP协议的概念](#_Toc25885_WPSOffice_Level2) [8](#_Toc25885_WPSOffice_Level2)

[2. HTTP的用途](#_Toc15499_WPSOffice_Level2) [8](#_Toc15499_WPSOffice_Level2)

[3. HTTP特点：](#_Toc31007_WPSOffice_Level2) [8](#_Toc31007_WPSOffice_Level2)

[1) 应用层协议，传输层使用tcp传输](#_Toc8185_WPSOffice_Level3) [8](#_Toc8185_WPSOffice_Level3)

[2) 简单灵活，和多种语言对接方便](#_Toc19766_WPSOffice_Level3) [8](#_Toc19766_WPSOffice_Level3)

[3) 无状态协议，不记录用户的通信内容](#_Toc6881_WPSOffice_Level3) [8](#_Toc6881_WPSOffice_Level3)

[4) 成熟稳定 http1.1](#_Toc28783_WPSOffice_Level3) [8](#_Toc28783_WPSOffice_Level3)

[1. HTTP协议的使用过程（工作模式）](#_Toc6005_WPSOffice_Level2) [9](#_Toc6005_WPSOffice_Level2)

[1. HTTP的请求Request](#_Toc10799_WPSOffice_Level2) [9](#_Toc10799_WPSOffice_Level2)

[2. HTTP的响应Response](#_Toc11009_WPSOffice_Level2) [9](#_Toc11009_WPSOffice_Level2)

[多进程编程](#_Toc14341_WPSOffice_Level1) [10](#_Toc14341_WPSOffice_Level1)

[1.通俗的讲：我们可以将一个没有在运行态的二进制文件叫做程序](#_Toc15540_WPSOffice_Level1) [10](#_Toc15540_WPSOffice_Level1)

[2.进程的状态：](#_Toc16322_WPSOffice_Level1) [10](#_Toc16322_WPSOffice_Level1)

[优缺点：](#_Toc22777_WPSOffice_Level1) [12](#_Toc22777_WPSOffice_Level1)

[进程间通信的示例：](#_Toc25885_WPSOffice_Level1) [13](#_Toc25885_WPSOffice_Level1)

[进程池：](#_Toc15499_WPSOffice_Level1) [14](#_Toc15499_WPSOffice_Level1)

[协程：](#_Toc31007_WPSOffice_Level1) [15](#_Toc31007_WPSOffice_Level1)

[多任务编程](#_Toc6005_WPSOffice_Level1) [17](#_Toc6005_WPSOffice_Level1)

[多线程：](#_Toc10799_WPSOffice_Level1) [17](#_Toc10799_WPSOffice_Level1)

[多线程实现方式：](#_Toc11009_WPSOffice_Level1) [18](#_Toc11009_WPSOffice_Level1)

[多核CPU：](#_Toc14074_WPSOffice_Level1) [20](#_Toc14074_WPSOffice_Level1)

[死锁：](#_Toc31596_WPSOffice_Level1) [30](#_Toc31596_WPSOffice_Level1)

[如何避免死锁：](#_Toc31219_WPSOffice_Level1) [32](#_Toc31219_WPSOffice_Level1)

[1.银行家算法。](#_Toc14074_WPSOffice_Level2) [32](#_Toc14074_WPSOffice_Level2)

[2.添加超时时间。](#_Toc31596_WPSOffice_Level2) [32](#_Toc31596_WPSOffice_Level2)

[mysql](#_Toc19454_WPSOffice_Level1) [33](#_Toc19454_WPSOffice_Level1)

[RE](#_Toc6506_WPSOffice_Level1) [37](#_Toc6506_WPSOffice_Level1)

[正则表达式模式](#_Toc8491_WPSOffice_Level1) [37](#_Toc8491_WPSOffice_Level1)

[正则表达式修饰符 - 可选标志](#_Toc18893_WPSOffice_Level1) [39](#_Toc18893_WPSOffice_Level1)

[正则表达式实例](#_Toc8291_WPSOffice_Level1) [39](#_Toc8291_WPSOffice_Level1)

[re.match函数](#_Toc15002_WPSOffice_Level1) [40](#_Toc15002_WPSOffice_Level1)

[re.match与re.search的区别](#_Toc9148_WPSOffice_Level1) [43](#_Toc9148_WPSOffice_Level1)

[检索和替换](#_Toc19446_WPSOffice_Level1) [43](#_Toc19446_WPSOffice_Level1)

[A46G8HFD1134](#_Toc18333_WPSOffice_Level1) [44](#_Toc18333_WPSOffice_Level1)

[12 32 43 3](#_Toc31219_WPSOffice_Level2) [48](#_Toc31219_WPSOffice_Level2)

[正则表达式对象](#_Toc23060_WPSOffice_Level1) [48](#_Toc23060_WPSOffice_Level1)

[re.RegexObject](#_Toc11028_WPSOffice_Level1) [48](#_Toc11028_WPSOffice_Level1)

[re.MatchObject](#_Toc5335_WPSOffice_Level1) [48](#_Toc5335_WPSOffice_Level1)

[正则表达式修饰符 - 可选标志](#_Toc8058_WPSOffice_Level1) [49](#_Toc8058_WPSOffice_Level1)

[GitHub 代码协同管理工具](#_Toc19776_WPSOffice_Level1) [49](#_Toc19776_WPSOffice_Level1)

[非关系型数据库（MongoDB）](#_Toc10894_WPSOffice_Level1) [52](#_Toc10894_WPSOffice_Level1)

## 进程、线程、协程

#### **进程**：

CPU密集型，它是计算机内核分配的最小单位（一个程序运行就是他一个进程）

#### **线程**：

IO密集型，调度执行的最小单位，也叫执行路径，不能独立存在，依赖进程存在一个进程至少有一个线程，叫主线程，而多个线程共享内存(数据共享，共享全局变量)，从而极大地提高了程序的运行效率。

应对大量的任务需要多进程完成,需要频繁创建和销毁进程的情况，我们可以创建

#### **进程池**：

创建一定量的进程作为进程池,用来处理事件,事件处理完成后进程不退出,而是继续等待处理其他事件,直到所有待处理事件处理完成后统一销毁,增加了进程的重复利用,降低资源消耗

当需要创建的子进程数量不多时，可以直接利用multiprocessing中的Process动态成生多个进程，但如果是上百甚至上千个目标，手动的去创建进程的工作量巨大，此时就可以用到multiprocessing模块提供的Pool方法。

初始化Pool时，可以指定一个最大进程数，当有新的请求提交到Pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到指定的最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来执行，请看下面的实例：

from multiprocessing import Pool  
import os,time,random  
   
def worker(msg):  
 t\_start = time.time()  
 print("%s开始执行,进程号为%d"%(msg,os.getpid()))  
 #random.random()随机生成0~1之间的浮点数  
 time.sleep(random.random()\*2)   
 t\_stop = time.time()  
 print(msg,"执行完毕，耗时%0.2f"%(t\_stop-t\_start))  
   
po=Pool(3) #定义一个进程池，最大进程数3  
for i in range(0,10):  
 #Pool.apply\_async(要调用的目标,(传递给目标的参数元祖,))  
 #每次循环将会用空闲出来的子进程去调用目标  
 po.apply\_async(worker,(i,))  
   
print("----start----")  
po.close() #关闭进程池，关闭后po不再接收新的请求  
po.join() #等待po中所有子进程执行完成，必须放在close语句之后  
print("-----end-----")

运行结果

----start----  
0开始执行,进程号为21466  
1开始执行,进程号为21468  
2开始执行,进程号为21467  
0 执行完毕，耗时1.01  
3开始执行,进程号为21466  
2 执行完毕，耗时1.24  
4开始执行,进程号为21467  
3 执行完毕，耗时0.56  
5开始执行,进程号为21466  
1 执行完毕，耗时1.68  
6开始执行,进程号为21468  
4 执行完毕，耗时0.67  
7开始执行,进程号为21467  
5 执行完毕，耗时0.83  
8开始执行,进程号为21466  
6 执行完毕，耗时0.75  
9开始执行,进程号为21468  
7 执行完毕，耗时1.03  
8 执行完毕，耗时1.05  
9 执行完毕，耗时1.69  
-------end-----------

#### **协程**：

是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。 协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快

**并发**:同时处理多个任务,内核在任务间不段地切换,达到好像是多个任务同时被执行的这种效果,实际上每一时刻只有一个任务在占有CPU

**并行**:一起执行,多个任务利用计算机的多核资源同时执行,此时多个任务之间是并行关系

#### **孤儿进程**

父进程先于子进程退出,此时子进程就是孤儿进程(会被系统进程进行收养)

#### **僵尸进程(Z)**

1.子进程先于父子进程退出,而且父进程没处理子进程的退出行为,此时子进程为僵尸进程

2.大量僵尸进程的后果: 虽然进程结束,但是会存留部分进程信息在内存中,耗费系统资源

3.如何避免僵尸进程:

1.让父进程处理子进程 pid, status = os.wait() 06*Z*wait.py

功能:在父进程中阻塞,等待子进程退出并处理

返回值: pid:子进程PID号 status:子进程的退出状态(0正常退出,1异常退出)

2.二级子进程

让一级子进程先退出,二级子进程变为了孤儿进程被收养,父进程处理一级子进程的退出(避免一级

子进程为僵尸)

父进程(回收子进程) 事件1-->

子进程(创建二级进程马上结束)-->

二级子进程(变为孤儿进程,被系统进程收养)事件2

## **网络编程**

### **一、协议(网络协议)概念**

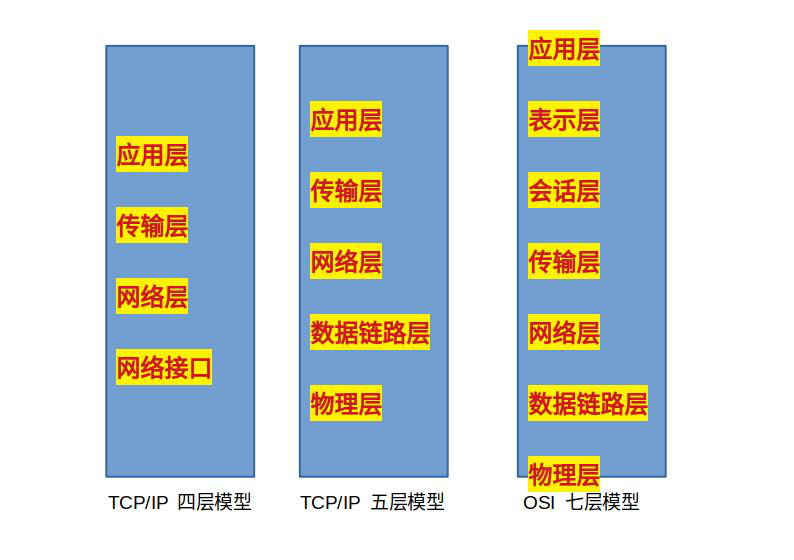
在网络通信中，各方必须遵守的规定。包括建立什么样的连接，消息结构等。

### **二、OSI模型和TCP / IP模型**

OSI七层模型：应用层、表示层、会话层、传输层、网络层、数据链路层、物理层

TCP/IP四层模型：应用层、传输层、网络层、数据链路层、物理层

TCP/IP五层模型：应用层、传输层、网络层、网络接口



### 三、TCP/IP四层模型\*\*

在四层模型中常用的协议：

应用层：FTP HTTP TFTP DNS SMTP

传输层：TCP UDP

网络层：IP ICMP IGMP

网络接口层（物理层）：

### **四、tcp / udp 传输协议**

#### 1. tcp 和 udp 的特征

#### 2. tcp 和 udp 的区别

1) 流式套接字采用字节流的方式传输数据，而数据报套接字以数据报形式传输。

2) tcp会产生粘包现象，udp消息是有边界的不会粘包。

3) tcp传输是建立在连接的基础上，保证传输的可靠性，而udp一次接受一个数报, 不保证完整性。

4) tcp依赖listen accept建立连接，udp不用。

5) tcp收发消息使用recv send，udp用recvfrom sendto。

#### 3. tcp的应用场景

文件传输，网页获取，邮件发送，信息聊天

#### 4. udp的应用场景

网络视频，群聊，发送广播

### **五、网络套接字编程**

1. 网络编程目的

2. 套接字的分类：

tcp套接字 和 udp 套接字

3. tcp套接字编程的基本步骤：

> 服务端：  
 >  
 > 1）导入模块，创建套接字对象  
 >  
 > 2）将套接字对象绑定IP端口号  
 >  
 > 3）将套接字对象设置为监听状态  
 >  
 > 4）等待客户端连接  
 >  
 > 5）消息的收发  
 >  
 > 6）关闭套接字  
 >  
 > 客户端：  
 >  
 > 1）导入模块，创建套接字对象  
 >  
 > 2）发送连接请求  
 >  
 > 3）消息的收发  
 >  
 > 4）关闭套接字

4. udp 套接字编程的基本步骤：

> 服务端：  
 >  
 > 1）导入模块，创建套接字对象  
 >  
 > 2）将套接字对象绑定IP端口号  
 >  
 > 3）消息的收发  
 >  
 > 4）关闭套接字  
 >  
 > 客户端：  
 >  
 > 1）导入模块，创建套接字对象  
 >  
 > 2）消息的收发  
 >  
 > 3）关闭套接字

1. tcp粘包的产生条件
2. tcp粘包的原因

* 1) 发送端需要等缓冲区满才发送出去，造成粘包 2) 接收方不及时接收缓冲区的包，造成多个包接收

1. tcp粘包的处理

1) 每次发送后加一个特殊结尾标志，接收端通过标志进行判断

2) 发送一个数据结构

3) 每次发送中间有一个短暂的延迟

### **六、HTTP协议**

1. HTTP协议的概念

超文本传输协议，应用层协议

2. HTTP的用途

网页的获取，基于网站的数据传输基于http协议的数据传输。

3. HTTP特点：

1) 应用层协议，传输层使用tcp传输

2) 简单灵活，和多种语言对接方便

3) 无状态协议，不记录用户的通信内容

4) 成熟稳定 http1.1

1. HTTP协议的使用过程（工作模式）

使用http双方均遵循http协议规定发送接收消息体

请求方，根据协议组织请求内容发送给对方

服务方，收到内容按照协议解析

服务方，将恢复内容按照协议组织发送给请求方

请求方，收到回复根据协议解析

1. HTTP的请求Request

* 1) 请求格式组成：
* \*请求行
* GET / HTTP/1.1
* 请求种类 请求内容 请求版本
* \*请求头
* 对请求内容的具体描述
* Accept: text/html
* Accept-Encoding: gzip, deflate, br
* Accept-Language: zh\_CN,zh;q=0.9
* Cache-Control: max-age=0
* Connection: keep-alive
* \*空行
* \*请求体
* 提交具体的请求参数
* 2) 常见的请求方式种类：
* GET 获取网络资源
* POST 提交一定的附加数据，得到返回结果
* PUT 更新服务器资源
* HEAD 获取响应头
* DELETE 删除服务器资源
* CONNECT 预留
* TRACE 测试
* OPTIONS 获取服务器性能

1. HTTP的响应Response

* 1) 响应格式:
* \*响应行
* 反馈具体的响应情况
* HTTP/1.1 200 OK
* 版本信息 响应码 附加信息
* \*响应头
* 对响应信息的具体描述
* Cache-Control:private.py
* Connect: Keep-Alive
* \*空行
* \*响应体
* 将客户想要的内容进行返回
* 2) 常见响应码：
* 1xx 提示信息 表示请求已经接受
* 2xx 响应成功
* 3xx 响应需要重新请定向
* 4xx 客户端错误，请求页面不存在
* 5xx 服务器错误

## 多进程编程

1.通俗的讲：我们可以将一个没有在运行态的二进制文件叫做程序

将在运行态的二进制文件叫做进程。程序只有一个，但是程序运行起来可能会产生多个进程。

2.进程的状态：

工作中，任务往往大于CPU的核心数，也就是一定有一些任务在执行，另外一些任务在等待CPU的唤醒去执行。因此导致出现了不同的状态。

就绪态： 执行的条件已经满足，等待CPU执行。

运行态：CPU正在执行其功能。

等待态：等待条件满足，例如一个程序的sleep.此时就处于等待态。

import time   
import threading  
import multiprocess  
  
def test1():  
 while True:  
 print('1-----')  
 time.sleep(1)  
   
def test2():  
 while True:  
 print('2------')  
 time.sleep(1)  
   
def main():  
 p1 = multiprocessing.Process(target=test1) # 创建第一个进程对象  
 p2 = multiprocessing.Process(target=test2) # 创建第二个进程对象  
 p1.start（）  
 p2.start()  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

子进程会复制主进程的代码以及资源，但是他独有的是进程id,进程最大的特点是在子进程没有修改父进程的代码的情况下，会进行对父进程的代码共享。

#### 进程、线程对比

进程 能够完成多任务，比如，在一台电脑上能够同时运行多个QQ。进程是操作系统调度的单位。

线程 能够完成多任务，比如 一个QQ中的多个聊天窗口。线程是资源分配的单位。

线程是不能够独立去执行，必须存在于进程中。

我们可以将进程理解我工厂中的流水线，而其中的线程就是这个流水线上的工人。线程使用进程中资源。

一个程序至少有一个进程，一个进程进程至少有一个线程。

进程在执行的过程中拥有独立的内存单元，而多个线程共享内存，从而极大的提高了程序的运行效率。

### 优缺点：

线程和进程在使用上各有优缺点，线程的执行开销小，但不利于资源的管理和保护，而进程正好相反。

#### 进程间的通信：

线程之间可以共享数据，但是进程是独立去开辟空间去运行。也就意味着他们之间就需要一种机制去通信。

socket是进程之间通信的一种方式。简单的说就是进程在不同的机器上。

其实也可以通过文件操作进行进程间的通信。只不过效率比较低。通过对硬盘的操作进行读取数据。这样效率远远低于在内存中的效率。

队列（**Queue**）是解决进程间的通信的一种方式。队列特点（先进先出）。

import multiprocessing  
  
# 创建队列,队列的长度为3  
q = multiprocessiong.Queue(3)  
# 往队列中放元素，可以放任意类型的元素  
q.put('123')  
q.put(123)  
q.put([123,234,4546])  
# 此时队列的长度已经达到3 再往里边添加元素，就会阻塞在此，等待队列空间的释放  
  
# 从队列中取元素  
q.get() # '123'  
q.get() # 123  
q.get() # [123,234,4546]  
# 此时队列中已经为空，再去拿元素的时候就会出现阻塞，因为队列中没有元素可以获取。  
  
# 判断队列的状态  
q.full()  
# 如果队列是满的状态就是 True 否则为False  
# 判断对列是否为空的状态  
q.empty()   
# 如果队列是空返回值 True 否则为False

### 进程间通信的示例：

# 多进程之间通过Queue 进行通信  
import multiprocessing  
  
def download\_from\_web(q):  
 """  
 模拟从网上下载音乐  
 """  
 data = [11,22,33,44]  
 # 向队列中写入数据  
 for temp in data:  
 q.put(temp)  
 print('---下载器已经完成了数据下载并且存入到队列中')  
   
def datascience\_data(q):  
 """  
 模拟获取数据进行音乐播放  
 """  
 waitting\_data = list()  
 while True:  
 data = q.get()  
 waitting\_data.append(data)  
   
 # 判断里边的元素是否为空  
 if q.empty():  
 break  
 print('音乐播放完毕：',waitting\_data)  
  
def main():  
 # 创建一个队列  
 q = multiprocessing.Queue() # 如果不写参数，此时系统会根据你电脑的内存自己设置阈值  
 # 创建多个进程，将队列的引用当做实参进行传递到  
 p1 = multiprocessing.Process(target=download\_from\_web，args=(q,))   
 p2 = multiprocessing.Process(target=datascience\_data，args=(q,))   
 p1.start()  
 p2.start()  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

### 进程池：

当创建的子进程数量不多的时候，我们可以使用multiprocessing中的Process动态生成多个进程，但如果是上百甚至上千个目标，手动去创建工作量巨大。此时就可以用到multiprocessing模块提供的Pool方法。

初始化Pool时候，可以指定一个最大的进程数，当有新的请求提交到Pool中时，如果还没有满，那么就会创建一个新的进程用来执行该请求，单如果池中的进程数已经叨叨了指定的最大值，那么该请求就会等待，直到进程池中有进程结束，才会用之前的进程来执行新的任务，请看下面的实例：

# 进程池代码演示：  
from multiprocessing import Pool  
import os,time,random  
  
  
def worker(msg):  
 t\_start = time.time()  
 print('开始执行，进程号为：%d'%（os.getpid()）)  
 # 生成0-2之间的随机数字  
 time.sleep(random.random()\*2)  
 t\_stop = time.time()  
 print(msg,'执行完毕，耗时%0.2f'%（t\_stop - t\_start）)  
   
po = Pool(3)  
for i in range(0,10):  
 # Pool().apply\_async（需要调用的目标，（传递给目标的参数元祖））  
 # 每次循环会空闲出来的子进程去调用目标  
 po.apply\_async(worker,(i,))  
   
print('----START----')  
po.close() # 关闭进程池，关闭后不在接收新的请求  
po.join() # 等待po中所有的子进程执行完毕，必须放在close语句之后  
print('----END-----')

### 协程：

协程的最大特点就是能够利用时间的等待去做执行其他的事情。协程依赖于线程，线程依赖于进程，

import time  
import gevent  
  
def f1(n):  
 for i in range(n):  
 print(gevent.getcurent(),i)  
 gevent.sleep(0.5)  
   
def f2(n):  
 for i in range(n):  
 print(gevent.getcurent(),i)  
 gevent.sleep(0.5)  
  
def f3(n):  
 for i in range(n):  
 print(gevent.getcurent(),i)  
 gevent.sleep(0.5)  
   
print('----1-----')  
g1 = gevent.spawn(f1,5)  
print('----2-----')  
g1 = gevent.spawn(f2,5)  
print('----3-----')  
g1 = gevent.spawn(f3,5)  
print('----4-----')  
g1.join()  
g2.join()  
g3.join()

解释：在使用gevent的时候，程序在等待gevent.sleep()的时候会进行阻塞等待，此时就会转而执行没有阻塞的f2.f2阻塞之后回去执行f3.这样就可以提高线程的执行效率。但是我们在之前写的代码可能有很多的耗时阻塞的方法。如果需要每一行都要改的话，那将是一个很大的工作量。如何有办法进行避免呢？

import time   
import gevent   
from gevent import monkey  
  
# 此方法的作用是将本模块中的所有的耗时操作都将转化为gevent的方法  
# 例如 sleep 或者recv等，不用我们人为的去修改gevent内部所特有的延时  
monkey.patch\_all()  
  
def f1(n):  
 for i in range(n):  
 print(gevent.getcurent(),i)  
 time.sleep(0.5)  
   
def f2(n):  
 for i in range(n):  
 print(gevent.getcurent(),i)  
 time.sleep(0.5)  
  
def f3(n):  
 for i in range(n):  
 print(gevent.getcurent(),i)  
 time.sleep(0.5)  
  
print('----1-----')  
g1 = gevent.spawn(f1,5)  
print('----2-----')  
g1 = gevent.spawn(f2,5)  
print('----3-----')  
g1 = gevent.spawn(f3,5)  
print('----4-----')  
g1.join()  
g2.join()  
g3.join()

更加简洁的方式：

import time   
import random  
import gevent   
from gevent import monkey  
  
def coroutine\_work(coroutine\_name)：  
 for i in range(10):  
 print(coroutine\_name,i)  
 time.sleep(random.random())  
   
# 将所有的任务放在gevent的joinall列表中、此时就可以等待回收所有的任务  
gevent.joinall([  
 gevent.spawn(coroutine\_work,'work1'),  
 gevent.spawn(coroutine\_work,'work2'),  
])

简单总结：

1.进程是资源分配的单位。

2.线程是操作系统的调度单位。

3.进程切换需要的资源最大，效率低。

4.线程切换需要的资源一般，但是效率一般。（在不考虑GIL的情况下）

5.协程切换任务资源很小，效率高。

6.多进程切换任务资源小，效率高。

7.多进程、多线程根据CPU的核数不一样可能是并行的。但是协程是在同一个线程中，是并发。

## 多任务编程

### 多线程：

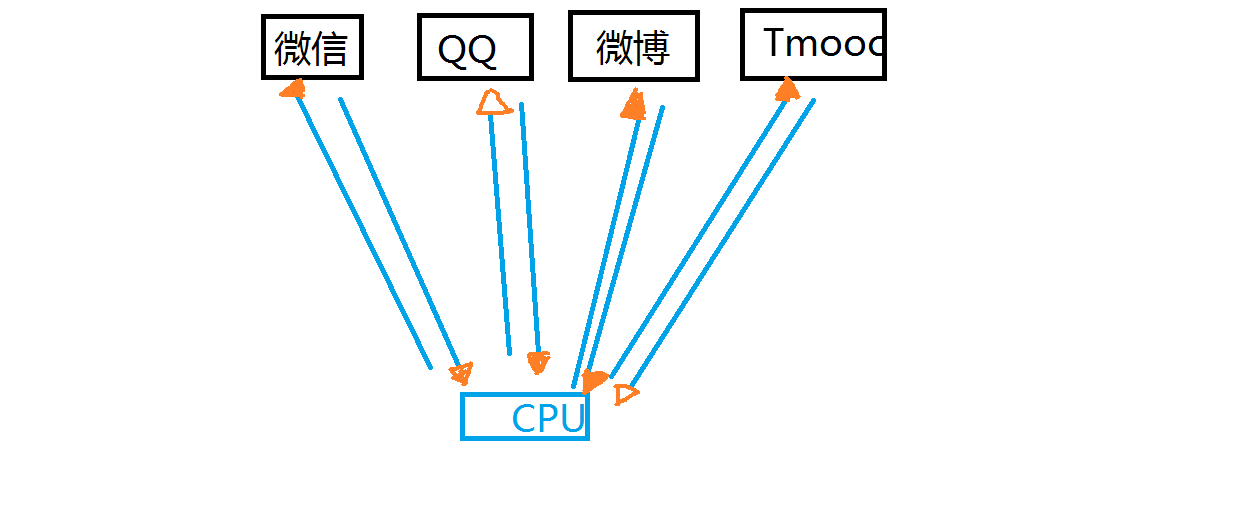
线程其实就是不同的应用去抢占CPU的时间片。

# 单进程  
# 在同时有多任务的时候，我们的不同的代码块只能一块一块的去执行。  
  
def sing():  
 """定义一个唱歌的函数"""  
 for i in range(5):  
 print('唱歌')  
   
def dance():  
 """定义一个跳舞的函数"""  
 for i in range(5):  
 print('跳舞')  
  
def main():  
 sing()  
 dance()  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

### 多线程实现方式：

用多线程的时候，就会有一种假象，好像是多个任务同时进行：其实是多个应用在抢占CPU的时间片，在很短的时间内进行，各个应用的切换，由于CPU的计算效率很高，就会给人造成一种，多个应用同时都在运行的错觉。造成假的'一起'运行。并且线程的运行是没有先后顺序的。

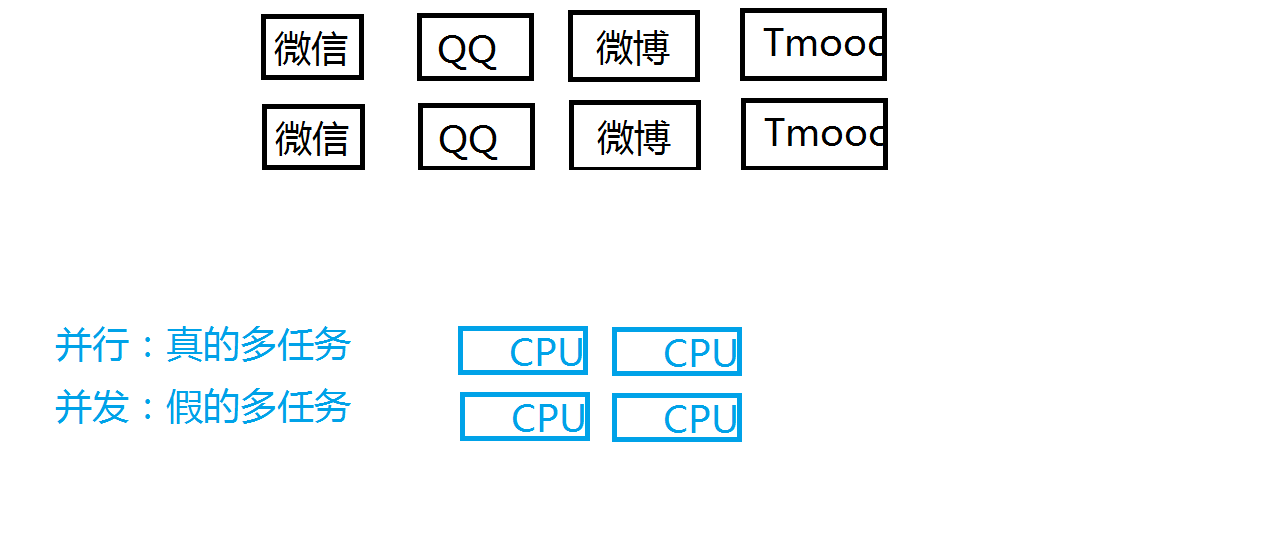
# 多线程：  
import threading  
  
def sing():  
 """定义一个唱歌的函数"""  
 for i in range(5):  
 print('唱歌')  
   
def dance():  
 """定义一个跳舞的函数"""  
 for i in range(5):  
 print('跳舞')  
  
def main():  
 t1 = threading.Thread(target=sing)  
 t2 = threading.Thread(target=dance)  
 t1.start()  
 t2.start()  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()



### 多核CPU：

并行: 当CPU的核数，大于任务数的时候，一定是并行，因为每个任务都能够分配到CPU的时间片。

并发：当CPU的核数，小于任务数的时候，一定是并发，因为每个任务都要去抢占CPU的时间片。



真正的并行执行只能在多核CPU上执行，但是由于任务数量远远多于CPU的核心数，所以操作系统会自动把很多任务轮流调度分配到每个核心上执行。

**python的thread模块是一个比较底层的模块，python的threading 模块是对thread进行了一系列封装，可以更加简便的使用**

使用threading模块创建多线程：

import threading  
  
def say\_sorry():  
 """用来道歉的函数"""  
 print('对不起，我错了。')  
   
def main():  
 for i in range(5):  
 t = threading.Thread(target=say\_sorry)  
 t.start()  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

#### 查看线程数

import threading  
import time  
  
def test1():  
 for i in range(5):  
 print('-----test1---%s'%i)  
 time.sleep(1)  
   
def test2():  
 for i in range(10):  
 print('-----test2---%s'%i)  
 time.sleep(1)  
  
def main():  
 t1 = threading.Thread(target=test1)  
 t2 = threading.Thread(target=test2)  
 t1.start()  
 t2.start()  
   
 while True:  
 print(thread.enumerate())   
 # 让所有的线程都同时启动，如果所有的子线程都挂掉的话，就可以终止主线程  
 if len(threading.enumerate()) <= 1:  
 break

#### 通过类来创建线程(创建线程的第二种方式)

建议：如果一个线程要实现多个事件，可以将其写成类，通过类方法之间的互相调用就可以实现。但必须调用的开始是run方法。

# coding:utf-8  
import threading  
import time  
  
class MyThread(threading.Thread):  
 def run(self): #通过继承threading.Thread,但必须继承run方法，  
 for i in range(3): # run方法里边的内容就是线程执行的代码块  
 time.sleep(1)  
 msg = 'i"m '+self.name + '@' +str(i)  
 print(msg)  
   
# 如果要实现多个事件，可以继续在类中写其他的方法，然后通过run方法互相调用，  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 t = MyThread()   
 t.start()

#### 多线程共享全局变量。

多个线程之间可以共享全局变量。

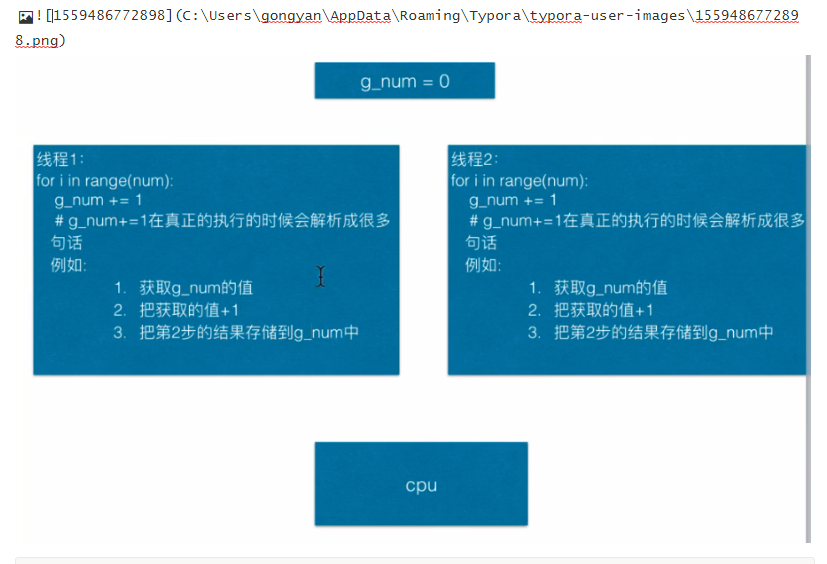
创建两个线程，线程1用来修改全局变量，线程2用来访问全局变量。主线程来访问全局变量：

import threading  
import time  
  
# 定义一个全局变量  
g\_num = 100  
  
def test1():  
 global g\_num  
 g\_num += 1  
 print('-----in tes1 g\_num = %d------'%g\_num)  
   
def test2():  
 print('-----in tes2 g\_num = %d------'%g\_num)  
   
def main():  
 t1 = threading.Thread(target=test1)  
 t2 = threading.Thread(target=test2)  
 t1.start()  
 time.sleep(1)  
   
 t2.start()  
 time.sleep(1)  
   
 print('-----in main thread g\_num = %d------'%g\_num)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()  
   
# 输出结果是：  
-----in tes1 g\_num = 101------  
-----in tes2 g\_num = 101------  
-----in main thread g\_num = 101------

#### 多线程共享全局变量（带有参数）：

import threading  
import time  
  
def test1(temp):  
 temp.append(33)  
 print('-----in tes1 temp = %s------'%str(temp))  
  
def test2(temp):  
 print('-----in tes2 temp = %s------'%str(temp))  
   
g\_mums = [11,22]  
def main():  
 # target指定子线程去执行的函数代码块  
 # args 调用函数的时候传递的参数  
 t1 = threading.Thread(target=test1，args=(g\_nums,))  
 t2 = threading.Thread(target=test2，args=(g\_nums,))  
 t1.start()  
 time.sleep(1)  
   
 t2.start()  
 time.sleep(1)  
   
 print('-----in main thread temp = %s------'%str(g\_num))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()  
   
# 运行的结果：  
-----in tes1 temp = [11,22,33]------  
-----in tes2 temp = [11,22,33]------  
-----in main thread temp = [11,22,33]------

#### 多线程贡献全局变量带来的问题：



import time  
import threading  
  
# 定义全局变量  
g\_num = 0  
  
def test1(num):  
 global g\_num  
 for i in range(num):  
 g\_num += 1  
 print('-----in test1 g\_num=',g\_num)  
   
def test2(num):  
 global g\_num  
 for i in range(num):  
 g\_num += 1  
 print('------in test2 g\_num=',g\_num)  
   
def main():  
 t1 = threading.Thread(target=test1, args=(100,))  
 t2 = threading.Thread(target=test2, args=(100,))  
   
 t1.start()  
 t2.start()  
   
 #等待上面的两个线程执行完毕  
 print('-----in main Thread g\_num='，g\_num)  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()  
# 当计算量小的时候不会出现竞争CPU时间片问题。  
# 如果数据量大的时候会出现竞争CPU时间片，不会按照预期的加法进行。

**解决线程同时修改全局变量的问题，可以通过线程锁来实现。当线程1对全局变量进行修改的时候，我们可以给全局变量上锁，等线程1工作完之后，再将锁进行释放。在上锁的过程中，不允许其他线程操作全局变量。为解决竞争CPU的时间片的问题，可以使用互斥锁。**

import time  
import threading  
  
# 定义全局变量  
g\_num = 0  
  
def test1(num):  
 global g\_num  
 singleLock.acquire()  
 for i in range(num):  
 g\_num += 1  
 singLock.release()  
 print('-----in test1 g\_num=',g\_num)  
   
def test2(num):  
 global g\_num  
 singleLock.acquire()  
 for i in range(num):  
 g\_num += 1  
 singLock.release()  
 print('------in test2 g\_num=',g\_num)  
  
# 创建一个互斥锁，默认是没有上锁的  
singleLock = threading.Lock()  
  
def main():  
 t1 = threading.Thread(target=test1, args=(100,))  
 t2 = threading.Thread(target=test2, args=(100,))  
   
 t1.start()  
 t2.start()  
   
 #等待上面的两个线程执行完毕  
 print('-----in main Thread g\_num='，g\_num)  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

### 死锁：

在多个线程之间多个资源进行共享的时候。如果两个线程分别占有一部分资源并且同时等待对方的资源，就会造成死锁。

尽管死锁很少发生，但一旦繁盛就会造成应用停止响应。

import time   
import threading  
  
class Mythread1(threading.Thread):  
 def run(self):  
 # 对mytextA进行上锁  
 mytextA.acquire()  
 print(self.name+'do1 ------up-----')  
 time.sleep(1)  
   
 # mytextB进行上锁，但此时会阻塞。  
 mytextB.acquire()  
 print(self.name+'do1 ------up-----')  
 mytextB.release()  
   
 mytextA.release()  
   
class Mythread2(threading.Thread):  
 def run(self):  
 # 对mytextA进行上锁  
 mytextB.acquire()  
 print(self.name+'do2 ------up-----')  
 time.sleep(1)  
   
 # mytextB进行上锁，但此时会阻塞。  
 mytextA.acquire()  
 print(self.name+'do2 ------up-----')  
 mytextA.release()  
   
 mytextB.release()  
  
   
mytextA = threading.Lock()  
mytextB = threading.Lock()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 t1 = MyThread1()  
 t2 = MyThread2()  
 t1.start()  
 t2.start()  
# 执行结果：  
Thread1 do1--------up----  
Thread2 do2--------up----  
..........卡死

此代码在运行的时候会出现，线程1对A进行上锁，线程2对B进行上锁，此时互相都在等待对方的全局变量的释放。那么此程序就会一直阻塞。

### 如何避免死锁：

1.银行家算法。

2.添加超时时间。

#### 银行家算法

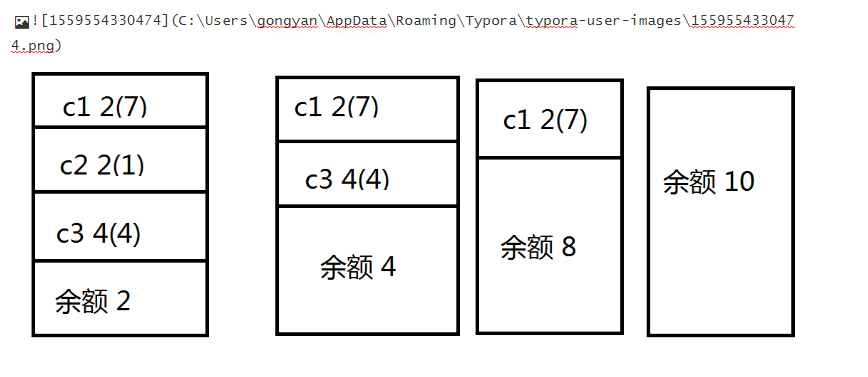
[背景知识]

一个银行家如何将一定数目的资金安全的借给若干个客户，使这些客户既能借到钱完成要干的事，同时银行家又能收回资金而不至于破产，这就是银行家问题。这个问题同操作系统中资源分配问题十分相似。银行家就像一个操作系统 ，客户就像运行进程，银行家的资金就是系统的资源。

[问题的描述]

一个银行家拥有一定的数量的资金，若干个客户要贷款，每个客户必须在一开始声明他所需要的贷款的总额。若该客户贷款总额不超过银行家的资金总数，银行家可以接受客户的要求，客户贷款是以每次一个资金单位（如1万人民币）方式进行，客户在借满所需的全部单位款额之前可能会等待，但银行家必须保证这些等待是有限的，可完成的。

例如：有三个客户C1,C2,C3，向银行家借款， 该银行家的资金总额为10个资金单位，其中C1客户要借9个资金单位，C2客户要借3个资金单位，C3客户要借8个b资金单位，总计20个资金单位，某一时刻的状态如图所示：



在有三个用户，C1,C2,C3.三个用户，他们需要借钱9,3,8个资金单位。但是银行现在只有10个资金单位，为了不影响银行盈利，但是又能留存客户。第一次借钱个C1是2个资金单位。C2是2个自己单位。C3是4个资金单位。银行为了盈利并且能够快速的收回资金，从余额中再拿出1个资金单位给C2,此时余额为1。C2借款单位满足。  
 过一段时间，C2的资金回收。现在银行余额为4个资金单位。此时可以满足C3客户的需求。  
 过一段时间后。C3资金回收。现在银行余额为8个资金单位，此时可以满足C1客户的需求。

## mysql

安装服务端：sudo apt-get install mysql-server  
安装客户端：sudo apt-get install mysql-client  
# Ubuntu安装软件 sudo apt-get update # 更新  
sudo apt-get -f install # 修复依赖关系  
sudo apt-get install # 软件包  
sudo /etc/inti.d/mysql start # 启动服务端  
sudo /etc/init.d/mysql stop #　停止服务端  
sudo /etc/init.d/mysql restart # 重启服务端  
sudo /etc/init.d/mysql status # 查看当前状态  
sudo /etc/init.d/mysql reload # 重新加载配置文件  
  
mysql -hlocalhost -uroot -p123456　　# 连接服务器  
  
create database 库名 [character set utf8]; # 创建库  
show databases;　　　　　　　　　　　 　 # 查看已有的库  
show create database 库名; 　　　　　 # 查看创建库的语句(字符集)  
use 库名; 　　　　　　　　　　　　　 # 切换库、进入库　　  
select database(); 　　　　　　　　　 # 查看当前所在库  
show tables; 　　　　　　　　　　　　　 # 查看库中已有表  
drop database 库名;　　　　　　　　　　 # 删除库  
  
# 创建表  
create table 表名(字段名1　数据类型,  
字段名2　数据类型, ...字段名n 数据类型)character set utf8;   
  
show create table 表名; # 查看创建表的语句(字符集)  
desc 表名; # 查看表结构  
drop table; # 删除表  
创建虚拟表:  
create view sss as select user.username,xinxi.city from user,xinxi where user.id=xinxi.id;  
删除虚拟表:drop view sss;  
  
# 插入表记录  
insert into 表名 values (值1),(值2),(值3)...;   
insert into 表名 (字段名1,字段名2...) values(值1),(值2)...;   
# 查询表记录  
select \* from 表名 [where 条件];   
select (字段名1,字段名2...) from 表名 [where 条件];   
# 表字段操作  
alter table 表名 add 字段名 数据类型; # 默认在表尾增加  
alter table 表名 add 字段名 数据类型 first; # 在表头增加  
alter table 表名 add 字段名 数据类型 after 字段名; # 在字段名后增加  
alter table 表名 drop 字段名; # 删除字段  
alter table 表名 modify 字段名 新的数据类型 # 修改字段数据类型  
alter table 表名 change 原字段名 新字段名 (数据类型不变) # 修改字段名  
  
delete from 表名 where 条件; # 删除表记录(一定得有where条件，否则删除所有记录)  
update 表名 set 字段1 = 值1,字段2 = 值2,...where 条件; # 更新表记录，一定得有where条件，否则该字段值全部更新  
  
select ... from表名group by () where/having (比较条件), order by () limit m,n ; # 筛选条件顺序  
  
# 普通索引index  
创建表时:index (字段名１), index (字段名２)  
# 已有表时:create index 索引名(一般索引名和字段名相同) on 表名(字段名)  
# 查看方式有两种  
desc 表名; show index from 表名\G;  
# 删除普通索引:drop index 索引名 on 表名;  
  
# 唯一索引unique  
# 创建表时： unique (字段名１), unique (字段名２)  
# 已有表时: cretate unique index 索引名(索引名一般和字段名相同) on 表名;  
# 查看和删除同普通索引  
  
# 主键primary key  
# 创建表时: id int ptimary key auto\_increment  
# 已有表时:alter table 表名 add primary key(字段名);# 先增加主键、再修改自增长属性  
alter table 表名 modify id int auto\_increment;  
# 设置自增长属性的起始值  
# 创建表时  
create table 表名(  
id int primary key auto\_increment,  
...)auto\_increment = 100;  
# 已有表时：alter table 表名 auto\_increment=100;  
# 主键自增长属性的修改和删除  
alter table 表名 modify id int;# 先修改自增长属性、再删除主键  
alter table 表名 drop primary key;  
  
# 外键 foreign key   
# 创建外键 主表从表数据类型要一致  
foreign key (参考字段) references 主表(被参考字段名)   
on delete cascade   
on update cascade;  
# 已有表时：alter table 表名 add foreign key (参考字段) references (被参考字段)  
on delete cascade  
on update cascade;  
# 删除外键 从表跟着主表更新  
show create table 表名;# 先查看外键名，紧跟CONSTRAINT后面即是  
alter table drop foreign key 外键名; # 再删除外键  
  
# 数据导入  
load data infile "文件名" # 此处的文件名必须是绝对路径  
into table 表名  
fields terminated by '分隔符'  
lines terminated by '\n';  
# 导入步骤  
# １、先把要导入的文件放到数据库默认的搜索路径。默认搜索路径查看方法：  
show variables like "secure\_file\_priv";  
# 执行复制命令: cp 文件路径(绝对路径) 默认搜索路径(/var/lib/mysql-files/)  
# 2、创建库、表(都必须是utf8)  
create database 库名 character set utf8;  
use 库名;  
create table 表名(  
字段名 数据类型,  
)character set utf8;  
# 3、执行导入语句  
load data infile "/var/lib/mysql-files/文件名"  
into table 表名  
fields terminated by '分分隔符'  
lines terminated by '\n';  
  
# 数据的导出 （路径必须为数据库指定的默认搜索绝对路径）   
select ... from 表名  
into outfile "文件名"  
fields terminated by '分隔符'  
lines terminated by '\n';  
  
# 表的复制：create table 表名 select ...from 表名;  
# 只复制表结构：create table 表名 select \* from 表名 where false;  
# 嵌套查询：select ... from 表名 where 字段名 运算符 (select ...)  
# 多表查询  
select ... from 表1,表2;(笛卡尔积，不加where条件)  
select ... from 表1,表2 where条件;  
# 内连接：select ... from 表1 inner join 表2 on 条件;  
# 外连接  
# 左连接：select ... from 表1 left join 表2 on 条件;  
# 右连接：select ... from 表1 right join 表2 on 条件;  
  
# 备份库  
# 备分库命令为: mysqldump -uroot -p --all-databases > 新库名.sql;  
--all-databases # 备份所有库  
库名　　　　　　# 备份一个库  
-B 库1,库2 # 备份几个库  
库名 表1,表 # 备份指定表  
# 数据恢复 （如果要恢复的库不存在时则必须先创建空库）  
# 命令：mysql -uroot -p MOSHOU < MOSHOU.sql  
# 恢复所有库命令：mysql -uroot -p --one-database MOAHOU < all.sql;  
  
# pymysql使用流程  
# 1、建立连接  
conn = pymysql.connect  
(host = "localhost",  
user = "root",  
passwd = "123456",  
db = "MOSHOU",  
charset = "utf8",  
port = 3306)  
# 2、创建游标对象  
cur = conn.cursor()  
# 3、执行sql语句  
cur.execute('...')  
# 4、提交到数据库执行  
conn.commit()  
# 5、关闭游标  
cur.close()  
# 6、断开连接  
conn.close()

## RE

### 正则表达式模式

|  |  |
| --- | --- |
| 模式 | 描述 |
| ^ | 匹配字符串的开头 |
| $ | 匹配字符串的末尾。 |
| . | 匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。 |
| [...] | 用来表示一组字符,单独列出：[amk] 匹配 'a'，'m'或'k' |
| [^...] | 不在[]中的字符：abc 匹配除了a,b,c之外的字符。 |
| re\* | 匹配0个或多个的表达式。 |
| re+ | 匹配1个或多个的表达式。 |
| re? | 匹配0个或1个由前面的正则表达式定义的片段，非贪婪方式 |
| re{ n} | 匹配n个前面表达式。例如，"o{2}"不能匹配"Bob"中的"o"，但是能匹配"food"中的两个o。 |
| re{ n,} | 精确匹配n个前面表达式。例如，"o{2,}"不能匹配"Bob"中的"o"，但能匹配"foooood"中的所有o。"o{1,}"等价于"o+"。"o{0,}"则等价于"o\*"。 |
| re{ n, m} | 匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式 |
| a| b | 匹配a或b |
| (re) | 匹配括号内的表达式，也表示一个组 |
| (?imx) | 正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。 |
| (?-imx) | 正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。 |
| (?: re) | 类似 (...), 但是不表示一个组 |
| (?imx: re) | 在括号中使用i, m, 或 x 可选标志 |
| (?-imx: re) | 在括号中不使用i, m, 或 x 可选标志 |
| (?#...) | 注释. |
| (?= re) | 前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。 |
| (?! re) | 前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功。 |
| (?> re) | 匹配的独立模式，省去回溯。 |
| \w | 匹配数字字母下划线 |
| \W | 匹配非数字字母下划线 |
| \s | 匹配任意空白字符，等价于 [\t\n\r\f]。 |
| \S | 匹配任意非空字符 |
| \d | 匹配任意数字，等价于 [0-9]。 |
| \D | 匹配任意非数字 |
| \A | 匹配字符串开始 |
| \Z | 匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。 |
| \z | 匹配字符串结束 |
| \G | 匹配最后匹配完成的位置。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。 |
| \B | 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。 |
| \n, \t, 等。 | 匹配一个换行符。匹配一个制表符, 等 |
| \1...\9 | 匹配第n个分组的内容。 |
| \10 | 匹配第n个分组的内容，如果它经匹配。否则指的是八进制字符码的表达式。 |

### 正则表达式修饰符 - 可选标志

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR(|) 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志：

|  |  |
| --- | --- |
| 修饰符 | 描述 |
| re.I | 使匹配对大小写不敏感 |
| re.L | 做本地化识别（locale-aware）匹配 |
| re.M | 多行匹配，影响 ^ 和 $ |
| re.S | 使 . 匹配包括换行在内的所有字符 |
| re.U | 根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B. |
| re.X | 该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。 |

### 正则表达式实例

#### 字符匹配

|  |  |
| --- | --- |
| 实例 | 描述 |
| python | 匹配 "python". |

#### 字符类

|  |  |
| --- | --- |
| 实例 | 描述 |
| [Pp]ython | 匹配 "Python" 或 "python" |
| rub[ye] | 匹配 "ruby" 或 "rube" |
| [aeiou] | 匹配中括号内的任意一个字母 |
| [0-9] | 匹配任何数字。类似于 [0123456789] |
| [a-z] | 匹配任何小写字母 |
| [A-Z] | 匹配任何大写字母 |
| [a-zA-Z0-9] | 匹配任何字母及数字 |
| [^aeiou] | 除了aeiou字母以外的所有字符 |
| [^0-9] | 匹配除了数字外的字符 |

#### 特殊字符类

|  |  |
| --- | --- |
| 实例 | 描述 |
| . | 匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。 |
| \d | 匹配一个数字字符。等价于 [0-9]。 |
| \D | 匹配一个非数字字符。等价于 0-9。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。 |
| \S | 匹配任何非空白字符。等价于 \f\n\r\t\v。 |
| \w | 匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9\_]'。 |
| \W | 匹配任何非单词字符。等价于 'A-Za-z0-9\_'。 |

### re.match函数

re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match()就返回none。

**函数语法**：

re.match(pattern, string, flags=0)

函数参数说明：

|  |  |
| --- | --- |
| 参数 | 描述 |
| pattern | 匹配的正则表达式 |
| string | 要匹配的字符串。 |
| flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。参见：[正则表达式修饰符 - 可选标志](https://www.runoob.com/python3/python3-reg-expressions.html#flags) |

匹配成功re.match方法返回一个匹配的对象，否则返回None。

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

|  |  |
| --- | --- |
| 匹配对象方法 | 描述 |
| group(num=0) | 匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。 |
| groups() | 返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。 |

#### 实例

#!/usr/bin/python   
import re   
print(re.match('www', 'www.runoob.com').span()) # 在起始位置匹配   
print(re.match('com', 'www.runoob.com')) # 不在起始位置匹配

以上实例运行输出结果为：

(0, 3)  
None

#### 实例

#!/usr/bin/python3  
import re  
   
line = "Cats are smarter than dogs"  
# .\* 表示任意匹配除换行符（\n、\r）之外的任何单个或多个字符  
matchObj = re.match( r'(.\*) are (.\*?) .\*', line, re.M|re.I)  
   
if matchObj:  
 print ("matchObj.group() : ", matchObj.group())  
 print ("matchObj.group(1) : ", matchObj.group(1))  
 print ("matchObj.group(2) : ", matchObj.group(2))  
else:  
 print ("No match!!")

以上实例执行结果如下：

matchObj.group() : Cats are smarter than dogs  
matchObj.group(1) : Cats  
matchObj.group(2) : smarter

#### re.search方法

re.search 扫描整个字符串并返回第一个成功的匹配。

函数语法：

re.search(pattern, string, flags=0)

函数参数说明：

|  |  |
| --- | --- |
| 参数 | 描述 |
| pattern | 匹配的正则表达式 |
| string | 要匹配的字符串。 |
| flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。参见：[正则表达式修饰符 - 可选标志](https://www.runoob.com/python3/python3-reg-expressions.html#flags) |

匹配成功re.search方法返回一个匹配的对象，否则返回None。

|  |  |
| --- | --- |
| 匹配对象方法 | 描述 |
| group(num=0) | 匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。 |
| groups() | 返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。 |

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

#### 实例

#!/usr/bin/python3  
   
import re  
   
print(re.search('www', 'www.runoob.com').span()) # 在起始位置匹配  
print(re.search('com', 'www.runoob.com').span()) # 不在起始位置匹配

以上实例运行输出结果为：

(0, 3)  
(11, 14)

#### 实例

#!/usr/bin/python3  
   
import re  
   
line = "Cats are smarter than dogs";  
   
searchObj = re.search( r'(.\*) are (.\*?) .\*', line, re.M|re.I)  
   
if searchObj:  
 print ("searchObj.group() : ", searchObj.group())  
 print ("searchObj.group(1) : ", searchObj.group(1))  
 print ("searchObj.group(2) : ", searchObj.group(2))  
else:  
 print ("Nothing found!!")

以上实例执行结果如下：

searchObj.group() : Cats are smarter than dogs  
searchObj.group(1) : Cats  
searchObj.group(2) : smarter

### re.match与re.search的区别

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

#### 实例

#!/usr/bin/python3  
   
import re  
   
line = "Cats are smarter than dogs";  
   
matchObj = re.match( r'dogs', line, re.M|re.I)  
if matchObj:  
 print ("match --> matchObj.group() : ", matchObj.group())  
else:  
 print ("No match!!")  
   
matchObj = re.search( r'dogs', line, re.M|re.I)  
if matchObj:  
 print ("search --> matchObj.group() : ", matchObj.group())  
else:  
 print ("No match!!")

以上实例运行结果如下：

No match!!  
search --> matchObj.group() : dogs

### 检索和替换

Python 的re模块提供了re.sub用于替换字符串中的匹配项。

语法：

re.sub(pattern, repl, string, count=0, flags=0)

参数：

* pattern : 正则中的模式字符串。
* repl : 替换的字符串，也可为一个函数。
* string : 要被查找替换的原始字符串。
* count : 模式匹配后替换的最大次数，默认 0 表示替换所有的匹配。
* flags : 编译时用的匹配模式，数字形式。

前三个为必选参数，后两个为可选参数。

#### 实例

#!/usr/bin/python3  
import re  
   
phone = "2004-959-559 # 这是一个电话号码"  
   
# 删除注释  
num = re.sub(r'#.\*$', "", phone)  
print ("电话号码 : ", num)  
   
# 移除非数字的内容  
num = re.sub(r'\D', "", phone)  
print ("电话号码 : ", num)

以上实例执行结果如下：

电话号码 : 2004-959-559   
电话号码 : 2004959559

#### repl 参数是一个函数

以下实例中将字符串中的匹配的数字乘于 2：

##### 实例

#!/usr/bin/python  
   
import re  
   
# 将匹配的数字乘于 2  
def double(matched):  
 value = int(matched.group('value'))  
 return str(value \* 2)  
   
s = 'A23G4HFD567'  
print(re.sub('(?P<value>\d+)', double, s))

执行输出结果为：

A46G8HFD1134

#### compile 函数

compile 函数用于编译正则表达式，生成一个正则表达式（ Pattern ）对象，供 match() 和 search() 这两个函数使用。

语法格式为：

re.compile(pattern[, flags])

参数：

* pattern : 一个字符串形式的正则表达式
* flags 可选，表示匹配模式，比如忽略大小写，多行模式等，具体参数为：
  + re.I 忽略大小写
* re.L 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境
* re.M 多行模式
* re.S 即为' . '并且包括换行符在内的任意字符（' . '不包括换行符）
* re.U 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库
* re.X 为了增加可读性，忽略空格和' # '后面的注释

##### 实例

import re  
pattern = re.compile(r'\d+') # 用于匹配至少一个数字  
m = pattern.match('one12twothree34four') # 查找头部，没有匹配  
print m  
None  
m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配，没有匹配  
print m  
None  
m = pattern.match('one12twothree34four', 3, 10) # 从'1'的位置开始匹配，正好匹配  
print m # 返回一个 Match 对象  
<\_sre.SRE\_Match object at 0x10a42aac0>  
m.group(0) # 可省略 0  
'12'  
m.start(0) # 可省略 0  
3  
m.end(0) # 可省略 0  
5  
m.span(0) # 可省略 0  
(3, 5)

##### 实例

在上面，当匹配成功时返回一个 Match 对象，其中：

* group([group1, …]) 方法用于获得一个或多个分组匹配的字符串，当要获得整个匹配的子串时，可直接使用 group() 或 group(0)；
* start([group]) 方法用于获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引），参数默认值为 0；
* end([group]) 方法用于获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引+1），参数默认值为 0；
* span([group]) 方法返回 (start(group), end(group))。

再看看一个例子：

##### 实例

>>>import re  
>>>pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I) # re.I 表示忽略大小写  
>>>m = pattern.match('Hello World Wide Web')  
print (m) # 匹配成功，返回一个 Match 对象  
<\_sre.SRE\_Match object at 0x10bea83e8>  
>>>m.group(0) # 返回匹配成功的整个子串  
'Hello World'  
>>>m.span(0) # 返回匹配成功的整个子串的索引  
(0, 11)  
>>>m.group(1) # 返回第一个分组匹配成功的子串  
'Hello'  
>>>m.span(1) # 返回第一个分组匹配成功的子串的索引  
(0, 5)  
>>>m.group(2) # 返回第二个分组匹配成功的子串  
'World'  
>>>m.span(2) # 返回第二个分组匹配成功的子串索引  
(6, 11)  
>>>m.groups() # 等价于 (m.group(1), >>>m.group(2), ...)  
('Hello', 'World')  
>>>m.group(3) # 不存在第三个分组  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
IndexError: no such group

#### findall

在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果没有找到匹配的，则返回空列表。

**注意：** match 和 search 是匹配一次 findall 匹配所有。

语法格式为：

findall(string[, pos[, endpos]])

参数：

* **string** 待匹配的字符串。
* **pos** 可选参数，指定字符串的起始位置，默认为 0。
* **endpos** 可选参数，指定字符串的结束位置，默认为字符串的长度。

查找字符串中的所有数字：

##### 实例

import re  
   
pattern = re.compile(r'\d+') # 查找数字  
result1 = pattern.findall('runoob 123 google 456')  
result2 = pattern.findall('run88oob123google456', 0, 10)  
   
print(result1)  
print(result2)

输出结果：

['123', '456']  
['88', '12']

#### re.finditer

和 findall 类似，在字符串中找到正则表达式所匹配的所有子串，并把它们作为一个迭代器返回。

re.finditer(pattern, string, flags=0)

参数：

|  |  |
| --- | --- |
| 参数 | 描述 |
| pattern | 匹配的正则表达式 |
| string | 要匹配的字符串。 |
| flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。参见：[正则表达式修饰符 - 可选标志](https://www.runoob.com/python3/python3-reg-expressions.html#flags) |

##### 实例

import re   
it = re.finditer(r"\d+","12a32bc43jf3")   
for match in it:   
 print (match.group() )

输出结果：

12   
32   
43   
3

#### re.split

split 方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

re.split(pattern, string[, maxsplit=0, flags=0])

参数：

|  |  |
| --- | --- |
| 参数 | 描述 |
| pattern | 匹配的正则表达式 |
| string | 要匹配的字符串。 |
| maxsplit | 分隔次数，maxsplit=1 分隔一次，默认为 0，不限制次数。 |
| flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。参见：[正则表达式修饰符 - 可选标志](https://www.runoob.com/python3/python3-reg-expressions.html#flags) |

##### 实例

>>>import re  
>>> re.split('\W+', 'runoob, runoob, runoob.')  
['runoob', 'runoob', 'runoob', '']  
>>> re.split('(\W+)', ' runoob, runoob, runoob.')   
['', ' ', 'runoob', ', ', 'runoob', ', ', 'runoob', '.', '']  
>>> re.split('\W+', ' runoob, runoob, runoob.', 1)   
['', 'runoob, runoob, runoob.']  
   
>>> re.split('a\*', 'hello world') # 对于一个找不到匹配的字符串而言，split 不会对其作出分割  
['hello world']

## 正则表达式对象

### re.RegexObject

re.compile() 返回 RegexObject 对象。

### re.MatchObject

group() 返回被 RE 匹配的字符串。

* **start()** 返回匹配开始的位置
* **end()** 返回匹配结束的位置
* **span()** 返回一个元组包含匹配 (开始,结束) 的位置

### 正则表达式修饰符 - 可选标志

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR(|) 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志：

|  |  |
| --- | --- |
| 修饰符 | 描述 |
| re.I | 使匹配对大小写不敏感 |
| re.L | 做本地化识别（locale-aware）匹配 |
| re.M | 多行匹配，影响 ^ 和 $ |
| re.S | 使 . 匹配包括换行在内的所有字符 |
| re.U | 根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B. |
| re.X | 该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。 |

## GitHub 代码协同管理工具

一. 为什么使用代码管理工具  
 1. 防止代码丢失,做备份  
 2. 做代码的版本控制  
 3. 方便多人开发进行分支管理  
  
二. git有什么特点  
  
 1. git是一个开源的分布式代码管理工具  
 2. git可以更好的支持分支,方便多人协同  
 3. git多用于linux unix操作系统,可以脱网操作  
 4. git数据传输速度更快,工作模式更多  
  
三. git 安装配置  
 1. 安装 : sudo apt-get install git  
 2. 配置用户和邮箱  
 [1] 配置命令 : git config   
 命令选项 --system 配置作用域系统所有用户  
 --global 配置作用域当前用户所有项目  
 [2] 配置用户名:  
 git config --global user.name Abby  
 [3] 配置邮箱  
 git config --global user.email zhao@123.com  
  
四. git基本操作  
 1. 初始化git仓库  
 git init  
 2. 查看分支状态  
 git status  
 3. 文件提交到暂存区  
 git add [files]  
 取消暂存操作  
 git rm --cached [files]  
 4. 将文件保存到本地仓库  
 git commit -m 'some message'  
 5. 查看commit日志  
 git log  
 6. 版本控制  
 git reset --hard [HEAD^ or commit\_id]  
 \* HEAD^ 表示退回到上一个版本  
 commit\_id 表示退回到指定版本  
 查看操作历史记录  
 git reflog  
 7. 其他命令  
 git diff file 比较工作区和仓库文件的区别  
 git checkout -- file 放弃工作区修改  
 git rm files 将文件删除添加到暂存区记录  
 git mv files 将文件移动添加到暂存区记录  
  
五. 标签管理  
 1. 什么是标签 : 即在重要的工作里程碑(版本)位置添加的一个工作状态的保存标记.  
 2. 标签命令  
 [1] 创建标签  
 git tag v1.0 [commit\_id] -m 'some message'  
 \* 默认在最新的状态添加标签,也可以指定一个commit版本  
 [2] 查看标签  
 git tag 查看标签列表  
 git show v1.0 查看标签具体信息  
 [3] 去往某个标签版本  
 git reset --hard v0.9  
 [4] 删除标签  
 git tag -d v2.0  
作业: 熟练命令,注册github账号 (github.com)  
  
  
六. 分支操作  
 1. 分支即每个人获取原有代码,在此基础上创建自己的工作环境,单独开发,不会影响其他分支操作,开发完成再统一合并.  
 2. 分支命令  
 [1] 查看分支  
 git branch  
 \* 前面带'\*'的分支为当前正在工作分支  
 [2] 创建分支  
 git branch [branch\_name]  
 \* 基于某个分支创建新的分支,则新分支会有原分支所有内容.  
 [3] 切换分支  
 git checkout [branch\_name]  
 \* git checkout -b [branck\_name] 可以创建并切换到新建分支  
 [4] 合并分支工作  
 git merge [branch\_name]  
 \* 如果将B分支合并到A分支,则合并操作需要在A分支中进行  
 注意 :   
 1. 在创建分支时,要确保原分支是干净的状态  
 2. 在和并分只时,如果多个分支操作同一个文件会产生冲突,需要自行解决  
 [5] 删除分支  
 git branch -d [branch\_name]  
 \* 如果分支没有被合并则需要用-D删除  
  
七. 远程仓库  
 1. 远程仓库: 远程主机上的仓库,实际git是分布式的,每一台主机的git结构都类似,只是把其他主机上的git仓库称之为远程仓库.  
 2. 创建共享仓库  
 [1] 创建目录,设置目录属主  
 mkdir gitrepo  
 chown tarena:tarena gitrepo  
 [2] 将该文件夹设置文可共享的git仓库  
 git init --bare ftp.git  
 [3] 设置git配置文件夹属主  
 chown -R tarena:tarena ftp.git  
 3. 向远程上传项目  
 [1] 连接远程主机  
 git remote add origin tarena@127.0.0.1:/home/tarena/gitrepo/ftp.git  
 [2] 向远程主机上传代码  
 git push -u origin master  
 \* -u 表示本地仓库分支与远程建立连接,这样以后再推送代码直接 git push即可  
 [3] 从远程仓库获取项目  
 git clone tarena@127.0.01:/home/tarena/gitrepo/ftp.git  
 \* clone 获取项目时会自动和远程仓库建立连接  
 [4] 从远程仓库获取代码  
 git pull  
八. github操作  
 1. github是一个社区网站,拥有全球最多的开源项目.git是该网站的唯一管理工具.  
 2. 获取项目   
 [1] 左上角搜索,选择想要的项目  
 [2] 使用clone 命令获取  
 3. 新建仓库  
 [1] 右上角 "+" 下拉,选择新建仓库  
 [2] 输入仓库具体信息  
 [3] 根据提示上传本地代码到仓库 (上传代码到远程过程)

## 非关系型数据库（MongoDB）

#创建集合:db.createCollection("acct")  
#向集合中插入数据:db.acct.insert({acct\_no:"62234500001",acct\_name:"Jerry",balance:5000.00})  
#查询:db.acct.find().pretty()#带格式 db.acct.find()#不带格式  
#删除集合: db.acct.drop()  
#重命名集合名称:db.student.renameCollection("stu")  
#插入一条文档操作:db.acct.insert({acct\_no:"6222345000002", acct\_name:"Tom", acct\_type:2, balance:12345.67})  
#插入多条文档操作  
db.acct.insert([  
 {acct\_no:"6222345000003", acct\_name:"Jack", acct\_type:1, balance:125.67},  
 {acct\_no:"6222345000004", acct\_name:"Rain", acct\_type:1, balance:99999.67},])  
#save实现插入  
db.acct.save({"\_id":ObjectId("5c10cffc35e4229055592fae"),acct\_no:"622345000006",acct\_name:"Emma",acct\_type:2,banlance:7777})  
#带条件查询:db.acct.find({acct\_name:"Emma",acct\_type:2}).pretty()  
#查找一笔账户类型为1的数据:db.acct.findOne({acct\_type:1})   
#指定\_id域不显示:db.acct.find({},{\_id:0})  
#指定不显示\_id域,显示acct\_no,acct\_name域 db.acct.find({条件},{显示的域})  
db.acct.find({},{\_id:0,acct\_no:1,acct\_name:1})  
#比较运算符  
et:查找acct\_type != 1的账号 db.acct.find({acct\_type:{$ne:1}})  
et:查找acct\_type == 1的账号 db.acct.find({acct\_type:{$eq:1}})  
et:查找balance > 5000的所有账户 db.acct.find({balance:{$gt:5000}})  
et:查找balance < 5000的所有账户 db.acct.find({balance:{$lt:5000}})  
et:查找balance <= 5000的所有账户 db.acct.find({balance:{$lte:5000}})  
et:查找balance >= 5000的所有账户 db.acct.find({balance:{$gte:5000}})  
#逻辑运算符  
et:查找balance > 99999.67,acct\_type == 1的账户信息  
 db.acct.find({$and:[{acct\_type:1},{balance:{$gt:99999.67}}]})  
et:查找acct\_type == 1 或 acct\_type ==2 的账户信息  
 db.acct.find({$or:[{acct\_type:1},{acct\_type:2}]})  
et：查找balance不是大于5000,即balance <= 5000 的账户信息  
 db.acct.find({balance:{$not:{$gt:5000}}})  
 查询用户名不是"Jerry"的所有账户  
 db.acct.find({acct\_name:{$not:{$eq:"Jerry"}}})  
et:查找既不是"Jack"也不是"Tom"的账户信息  
 db.acct.find({$nor:[{acct\_name:"Jack"},{acct\_name:"Tom"}]})  
#插入一条信息:db.student.insert({name:'Irris', age:22, socre:[70,77,88,99], sex:1})  
#查找含有sex的所有数据  
db.student.find({sex:{$exists:true}})#存在 db.student.find({sex:{$exists:false}})#不存在  
$mod: 根据余数查找 et:查找年龄为奇数的学生信息 db.student.find({age:{$mod:[2,1]}})  
$type: 查找name位字符串型的所有数据 db.student.find({name:{$type:2}})  
#查找函数  
1.distinct:去重,查看某个域的取值范围,et:查看acct集合中acct\_type一共有多少种 db.acct.distinct('acct\_type')  
2.limit: 显示查找结果的前面n条 limit(m.n) et:查看acct集合前面3笔的数据 db.acct.find().limit(3)  
3.skip:跳过查找结果前面的n笔数据 et:跳过前2笔 db.acct.find({acct\_type:1}).skip(2)  
4.sort:对查询结果排序 语法格式:sort({域名称,1/-1}) 1.升序 2.降序 没有这个域不会参加排序  
 et:对所有余额进行降序排列 db.acct.find().sort({balance:-1})  
 et:查找所有账户按照账户类型和余额升序排列 db.acct.find().sort({acct\_type:1,balance:1})  
5.count():统计查询结果数量 et:对账户数量进行统计 db.acct.find().count()  
6.pretty():将find查询结果格式化显示et:格式化显示查询结果 db.acct.find().pretty()  
#文档删除 格式:db.collectionName.remove({query}, justOne)  
 et:删除1笔账户类型为1的数据 db.acct.remove({acct\_type:1},true)  
#文档的修改  
 1.格式:db.collectionName.update(query,update,upsert,multi)  
 参数:  
 -query:数据筛选条件  
 -update: 修改操作符,表示要修改的数据  
 -upsert:bool值, 指定在没有找到满足条件数据的时候,要不要插入,true--插入,false--插入  
 -multi:bool值,指定是否修改多笔数据 true--所有满足所有条件的数据,false--值修改第一条  
 2.修改操作符:  
 -$set:修改某个账户余额  
 et:db.acct.update({acct\_no:"622345000006"}, {$set:{balance:10000.00}}, false,false)  
 et:如果修改不存在的域,则创建  
 db.acct.update({acct\_no:"6222345000002"},  
 {$set:{balance:10000.00,reg\_date:Date()}}, false,false)  
 -$unset:删除某个域 删除某个账户的reg\_date域 使用unset将某个域设置位""  
 db.acct.update({acct\_no:"6222345000002"}, {$unset:{reg\_date:''}})  
 -$rename:修改域名称  
 首先,在acct集合增加一个域revoke\_date(注销日期)  
 db.acct.update({}, {$set:{revoke\_date:Date()}}, false, true)  
 修改域名称remove\_date修改成close\_date  
 db.acct.update({},{$rename:{revoke\_date:'close\_date'}},false,true)  
 -$inc:加法修改器,完成对域的值加减计算  
 et:为某个账户余额增加1000元  
 db.acct.update({acct\_no:"6222345000002"}, {$inc:{balance:1000}}, false,false)  
 -$mul: 乘法修改器,完成对域的值乘法计算  
 et:为每个账户余额增加2%的利息  
 db.acct.update({},{$mul:{balance:1.02}},false,true)  
 \*$inc$mul作用值可以是整数小数正数负数  
 -$min:用指定的域的值和某个值比较,取其中较小者,如果大于指定值则改为指定值, 小于指定值则不做修  
 et:用来限定某个域的最大值  
 db.student.update({name:"Dekie"},{$min:{socre:75}},false,false)  
 -$max:用指定的域的值和某个值比较,取其中较大者, 如果小于指定值则改为指定值, 大于指定值则不做修改  
 et:用来限定某个域的最小值 将低于60分的成绩修改成60  
 db.student.update({},{$max:{socre:60}},false,true)  
 -$push: 向数组中添加一个值  
 et:向student集合中,socre(成绩)域添加一名成绩  
 db.student.update({name:"Dekie"}, {$push:{socre:44}}, false, false)  
 -$pushAll:向语句中添加多个值  
 et：向student集合中,添加两门成绩  
 db.student.update({name:"Dekie"}, {$pushAll:{socre:[22,33]}}, false, false)  
 -$pull:从数组中删除某个值,  
 et:从student 结合中,删除33的这个成绩  
 db.student.update({}, {$pull:{socre:33}}, false, false)  
 -$pullAll: 从数组中删除多项值  
 et:从student 结合中,删除22,44的这个成绩  
 db.student.update({}, {$pullAll:{socre:[22,44]}}, false, false)  
 -$each: 逐一取出器中得知进行操作 $each迭代操作  
 et:利用$push, $each组合向数组添加多个值  
 db.student.update({name:"Dekie"}, {$push:{socre:{$each:[77,88,99]}}}, false, false)  
 -$position: 指定数据的插入位置,搭配$push, $each使用 #值不能为负数  
 et:将某个值插入到数组的头部,将插入位置指定为0  
 db.student.update({name:"Dekie"}, {$push:{socre:{$each:[100],$position:0}}}, false, false)  
 $sort:对数组的内容进行排序(1为升序, 2为降序)  
 et:对某个学生的成绩进行排序  
 db.student.update({name:"Dekie"}, {$push:{socre:{$each:[],$sort:1}}}, false, false)  
 -$pop: 弹出数组中的一项 (1表示弹出最后一项, -1表示弹出第一项)  
 et:db.student.update({name:"Dekie"}, {$pop:{socre:-1}}, false, false)  
 -$addToSet: 向数组中添加一个值, 但是不能添加已有内容  
 et:db.student.update({name:"Dekie"}, {$addToSet:{socre:98}}, false, false)  
 通过索引访问数组:域.索引值  
 et:查找socre域第一项值为70的数据 \*域和索引的引号不能少  
 db.student.find({"socre.0":70})  
#内嵌文档引用方式:  
 外部文档.内部文档.域名称use test  
 db.acct.update({acct\_no : '6222345000003'},  
 {$set:{cust\_info:{tel:'13456212345',email:'15234@qw.com',addr:'BJ'}}},false,false)  
 et:根据客户电话查找  
 -格式:db.collectionName.find({'内部文档名.域名称'})  
 db.acct.find({'cust\_info.tel':'13456212345'},{\_id:0})  
 \*域的口号不能省略  
 et:修改某个客户的电话(cust\_info.tel)  
 -格式:db.collectionName.update({},{$set:{'内部文档名.域名称':'变更后的内容'}})  
 db.acct.update({acct\_no : '6222345000003'},{$set:{'cust\_info.tel':'1111111111'}})  
#索引:索引是建立文档所在位置的查找清单,使用索引可以方便的进行快速查找,减少遍历次数,提高查找效率  
 \*全表扫描:将集合中所有数据逐一取出进行条件匹配  
 -索引注意事项:  
 +建立适当的索引,索引不是越多越好  
 +当数据库很小时不适合创建索引  
 +如果数据修改, 索引表也要随之修改, 否则索引就没有意义  
 -查看和创建索引  
 +查看:db.collectionName.getIndexes()  
 +创建:db.collectionNmae.createIndex({})  
 et:创建name为升序索引 db.student.createIndex({name:1}) 创建升序索引  
 +删除:db.collectionName.dropIndex(索引名称)  
 \*如果不带索引名称,则会删除所有索引 db.collectionName.dropIndexes()  
 -符合索引:根据多个域创建索引  
 et:在acct集合上,根据acct\_name,acct\_no,创建索引  
 db.acct.createIndex({acct\_no:1, acct\_name:1})  
 -唯一索引:创建唯一索引的域要求所有值不能重查询效率高 MongoDB中,通过unique指定创建唯一索引  
 et:在acct集合的acct\_no上创建唯一索引  
 db.acct.createIndex({acct\_no:1},{unique:true})  
 -稀疏索引  
 -稀疏索引(间隙索引):如果文档的域不存在,则不对该文档创建索引sparse: true  
 db.collectionName.createIndex({}, {sparse: true})  
 et:在orders 集合上fax域创建稀疏索引  
 db.orders.createIndex({fax:1},{sparse:true})  
#聚合操作  
 1.聚合:对文档的高级筛选,统计 如求和,求平均值,求最大/最小值  
 2.聚合函数: db.collectionName.aggregate()  
 -功能:完成聚合操作  
 -参数:聚合条件,配合聚合操作符使用  
 -返回:返回聚合结果  
 3.常用聚合操作  
 -$group: 分组聚合,需要和统计操作符$sum配合使用  
 et:根据acct\_type统计各类账户数量  
 db.acct.aggregate({$group:{'\_id':'$acct\_type', num:{$sum:1}}})  
 # group后面只能使用'\_id'; $sum后面参数只能是1; ‘$acct\_type’中'$'不能少  
 类似于SQL中:select acct\_type, count(\*) from acct group by acct\_type  
 -$sum:对某个域求和  
 et:统计各类型账户余额求和 下根据账户类型分类,再进行求和 total是查询结果域的名称  
 db.acct.aggregate({$group:{'\_id':'$acct\_type',total:{$sum:'$balance'}}})  
 类似于SQL中: select acct\_type,sum(balance) from acct group by acct\_type  
 et:统计所有账户的和 不统计分组依据  
 db.acct.aggregate({$group:{'\_id':'',total:{$sum:'$balance'}}})  
 -max:求最大值  
 et:找出每一类账户中,余额最大的账户  
 db.acct.aggregate({$group:{'\_id':'$acct\_type',ls:{$max:'$balance'}}})  
 et:找有所有账户中余额最大的值  
 db.acct.aggregate({$group:{'\_id':'',ls:{$max:'$balance'}}})  
 -min:求最小值  
 et:找出每一类账户中,余额最小的账户  
 db.acct.aggregate({$group:{'\_id':'$acct\_type',ls:{$min:'$balance'}}})  
 et:找有所有账户中余额最小的值  
 db.acct.aggregate({$group:{'\_id':'',ls:{$min:'$balance'}}})  
 -$avg:求平均值  
 et:求每类账户平均值  
 db.acct.aggregate({$group:{'\_id':'$acct\_type',pj:{$avg:'$balance'}}})  
 et:求账户平均值  
 db.acct.aggregate({$group:{'\_id':'',pj:{$avg:'$balance'}}})  
 -$project:从集合中,查询指定的域(类似于关系数据库的投影运算) 用于格式化的显示文档  
 et:从acct集合中查询acct\_no,acct\_name显示,其他域不显示  
 db.acct.aggregate({$project:{'\_id':0,AcctNo:'$acct\_no',AcctName:'$acct\_name'}})  
 \* AcctNo,AcctName 为显示查询结果在的别名 '','$'不可省略  
 -$match:对数据过滤,匹配数据  
 et：查询 账户余额小于11000的数据  
 db.acct.aggregate({$match:{balance:{$lt:11000}}})  
 -$sort:排序 利用聚合进行排序 1表示正序排序;-1表示倒叙排序  
 et:查询所有账户,按照账户余额,倒序排列  
 db.acct.aggregate({$sort:{balance:-1}})  
 -$first:第一个  
 -$limit:  
 -$skip:跳过  
 -$last:最后一个  
 -管道聚合:指的是将多个聚合操作合并到一起完成, 将上一个聚合的结果交给下一个聚合作为操作对象操作  
 格式:db.collectionName.aggregate([{聚合1},{聚合2}...])  
 et:首先对账号按照类型进行筛选,然后在进行排序  
 db.acct.aggregate([{$match:{acct\_type:1}},{$sort:{acct\_no:-1}}])  
 \*acct\_type:1中表示账户类型的值 acct\_no:-1表示将序排序  
#创建固定集合  
 -语法: db.createCollection(collectionName, {capped:true, size:1000, max:1000})  
 -参数:  
 collectionName: 集合名称  
 capped:true 表示创建固定集合  
 size: 集合的大小, 单位字节(Byte)  
 max: 最多存放多少文档  
 -et:创建一个最大能存储3个数据的固定集合  
 db.createCollection('log', {capped: true, size:1000, max:3})  
 测试:插入3笔数据,每笔都存入,当插入第4笔数据时,第一笔将被覆盖  
 db.log.insert([{name:'Emme',oper:'update'}, {name:'LZ',oper:'derf'},  
 {name:'Lkis',oper:'update'}])  
 db.log.insert({name:'Tome',oper:'ins'})