Python之进程

Python之进程

进程

本节目录

一 背景知识

顾名思义，进程即正在执行的一个过程。进程是对正在运行程序的一个抽象。

进程的概念起源于操作系统，是操作系统最核心的概念，也是操作系统提供的最古老也是最重要的抽象概念之一。操作系统的其他所有内容都是围绕进程的概念展开的。

所以想要真正了解进程，必须事先了解操作系统，点击进入

PS：即使可以利用的cpu只有一个（早期的计算机确实如此），也能保证支持（伪）并发的能力。将一个单独的cpu变成多个虚拟的cpu（多道技术：时间多路复用和空间多路复用+硬件上支持隔离），没有进程的抽象，现代计算机将不复存在。

必备的理论基础：

```
#一 操作系统的作用：
    1：隐藏丑陋复杂的硬件接口，提供良好的抽象接口
    2：管理、调度进程，并且将多个进程对硬件的竞争变得有序

#二 多道技术：
    1.产生背景：针对单核，实现并发
    ps：
    现在的主机一般是多核，那么每个核都会利用多道技术
    有4个cpu，运行于cpu1的某个程序遇到io阻塞，会等到io结束再重新调度，会被调度到4个
    cpu中的任意一个，具体由操作系统调度算法决定。

    2.空间上的复用：如内存中同时有多道程序
    3.时间上的复用：复用一个cpu的时间片
      强调：遇到io切，占用cpu时间过长也切，核心在于切之前将进程的状态保存下来，这样
          才能保证下次切换回来时，能基于上次切走的位置继续运行
```

二 什么是进程

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。我们自己在python文件中写了一些代码，这叫做程序，运行这个python文件的时候，这叫做进程。

狭义定义：进程是正在运行的程序的实例（an instance of a computer program that is being executed）。
广义定义：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单

元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

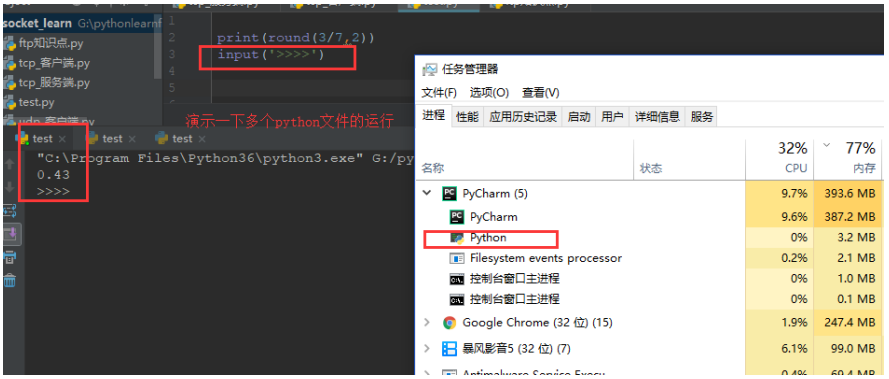
举例： 比如py1文件中有个变量a=1，py2文件中有个变量a=2，他们两个会冲突吗？不会的，是不是，因为两个文件运行起来后是两个进程，操作系统让他们在内存上隔离开，对吧。

第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）（python的文件）、数据区域
第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行之），它才能成为一个活动
进程是操作系统中最基本、重要的概念。是多道程序系统出现后，为了刻画系统内部出现的动态情况，描述系统内部各道程序的活动规

动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生，动态消亡的。
并发性：任何进程都可以同其他进程一起并发执行
独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位；
异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进
结构特征：进程由程序、数据和进程控制块三部分组成。
多个不同的进程可以包含相同的程序：一个程序在不同的数据集里就构成不同的进程，能得到不同的结果；但是执行过程中，程序不能

程序是指令和数据的有序集合，其本身没有任何运行的含义，是一个静态的概念。
而进程是程序在处理机上的一次执行过程，它是一个动态的概念。
程序可以作为一种软件资料长期存在，而进程是有一定生命期的。
程序是永久的，进程是暂时的。
举例：就像qq一样，qq是我们安装在自己电脑上的客户端程序，其实就是一堆的代码文件，我们不运行qq，那么他就是一堆代码程序，

注意：同一个程序执行两次，就会在操作系统中出现两个进程，所以我们可以同时运行一个软件，分别做不同的事情也不会混乱。比如打开暴风影音，虽然都是同一个软件，但是一个可以播放苍井空，一个可以播放饭岛爱。



三 进程调度

要想多个进程交替运行，操作系统必须对这些进程进行调度，这个调度也不是随即进行的，而是需要遵循一定的法则，由此就有了进程的调度算法。

先来先服务（FCFS）调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。FCFS算法比较有利于长作业

短作业（进程）优先调度算法（SJ/PF）是指对短作业或短进程优先调度的算法，该算法既可用于作业调度，也可用于进程调度。但其对

时间片轮转(Round Robin，RR)法的基本思路是让每个进程在就绪队列中的等待时间与享受服务的时间成比例。在时间片轮转法中，需
显然，轮转法只能用来调度分配一些可以抢占的资源。这些可以抢占的资源可以随时被剥夺，而且可以将它们再分配给别的进程。
在轮转法中，时间片长度的选取非常重要。首先，时间片长度的选择会直接影响到系统的开销和响应时间。如果时间片长度过短，则调
在轮转法中，加入到就绪队列的进程有3种情况：
一种是分给它的时间片用完，但进程还未完成，回到就绪队列的末尾等待下次调度去继续执行。
另一种情况是分给该进程的时间片并未用完，只是因为请求I/O或由于进程的互斥与同步关系而被阻塞。当阻塞解除之后再回到就绪
第三种情况就是新创建进程进入就绪队列。
如果对这些进程区别对待，给予不同的优先级和时间片从直观上看，可以进一步改善系统服务质量和效率。例如，我们可把就绪队

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指
而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种
(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降
(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果

对于多级反馈队列，windows不太清楚，但是在linux里面可以设置某个进程的优先级，提高了有限级有可能就会多执行几个时间片。

四 并发与并行

通过进程之间的调度，也就是进程之间的切换，我们用户感知到的好像是两个视频文件同时在播放，或者音乐和游戏同时在进行，那就让我们来看一下什么叫做并发和并行

无论是并行还是并发，在用户看来都是'同时'运行的，不管是进程还是线程，都只是一个任务而已，真是干活的是cpu，cpu来做这些任务，而一个cpu同一时刻只能执行一个任务

并发：是伪并行，即看起来是同时运行。单个cpu+多道技术就可以实现并发，（并行也属于并发）

你是一个cpu，你同时谈了三个女朋友，每一个都可以是一个恋爱任务，你被这三个任务共享要玩出并发恋爱的效果，
应该是你先跟女友1去看电影，看了一会说：不好，我要拉肚子，然后跑去跟第二个女友吃饭，吃了一会说：那啥，我去趟洗手间，然后

并行：并行：同时运行，只有具备多个cpu才能实现并行

将多个cpu必须成高速公路上的多个车道，进程就好比每个车道上行驶的车辆，并行就是说，大家在自己的车道上行驶，会不影响，同时

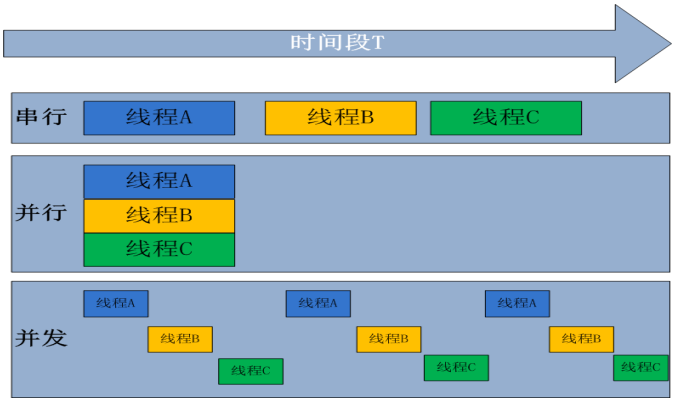
单核下，可以利用多道技术，多个核，每个核也都可以利用多道技术（**多道技术是针对单核而言的**）

有四个核，六个任务，这样同一时间有四个任务被执行，假设分别被分配给了cpu1，cpu2，cpu3，cpu4，

一旦任务1遇到I/O就被迫中断执行，此时任务5就拿到cpu1的时间片去执行，这就是单核下的多道技术

而一旦任务1的I/O结束了，操作系统会重新调用它(**需知进程的调度、分配给哪个cpu运行，由操作系统说了算**)，可

能被分配给四个cpu中的任意一个去执行



所有现代计算机经常会在同一时间做很多件事，一个用户的PC（无论是单cpu还是多cpu），都可以同时运行多个任务（一个任务可以理解为一个进程）。

启动一个进程来杀毒（360软件）

启动一个进程来看电影（暴风影音）

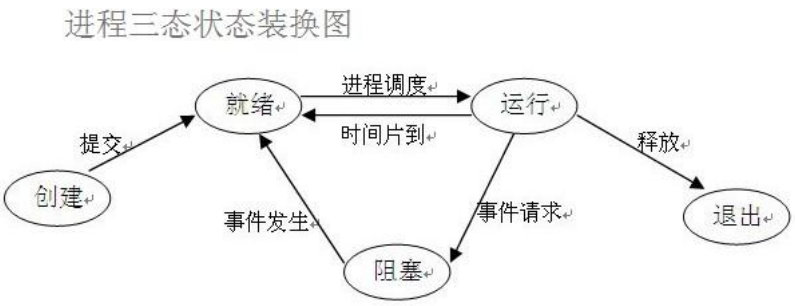
启动一个进程来聊天（腾讯QQ）

所有的这些进程都需被管理，于是一个支持多进程的多道程序系统是至关重要的

多道技术概念回顾：内存中同时存入多道（多个）程序，cpu从一个进程快速切换到另外一个，使每个进程各自运行几十或几百毫秒，这样，虽然在某一个瞬间，一个cpu只能执行一个任务，但在1秒内，cpu却可以运行多个进程，这就给人产生了并行的错觉，即伪并行，以此来区分多处理器操作系统的真正硬件并行（多个cpu共享同一个物理内存）

五 同步\异步\阻塞\非阻塞（重点）

1.进程状态介绍



在了解其他概念之前，我们首先要了解进程的几个状态。在程序运行的过程中，由于被操作系统的调度算法控制，程序会进入几个状态：就绪，运行和阻塞。

（1）就绪(Ready)状态

当进程已分配到除CPU以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态。

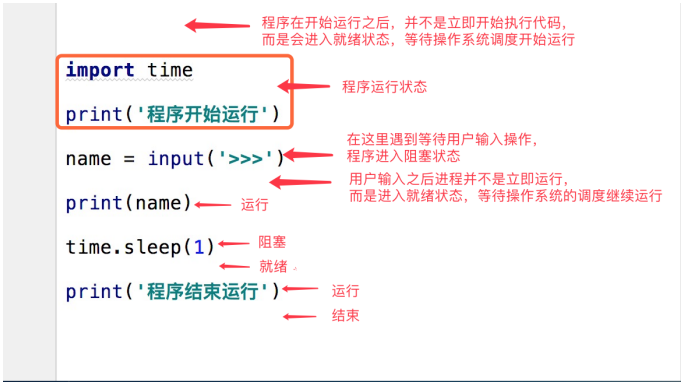
（2）执行/运行（Running）状态当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为执行状态。

（3）阻塞(Blocked)状态正在执行的进程，由于等待某个事件发生而无法执行时，便放弃处理机而处于阻塞状态。引起进程阻塞的事件可有多种，例如，等待I/O完成、申请缓冲区不能满足、等待信件(信号)等。

事件请求：input、sleep、文件输入输出、recv、accept等

事件发生：sleep、input等完成了

时间片到了之后有回到就绪状态，这三个状态不断的在转换。



2.同步异步

所谓同步就是一个任务的完成需要依赖另外一个任务时，只有等待被依赖的任务完成后，依赖的任务才能算完成，这是一种可靠的任务序列 。要么成功都成功

所谓异步是不需要等待被依赖的任务完成，只是通知被依赖的任务要完成什么工作，依赖的任务也立即执行，只要自己完成了整个任务就算完成
至于被依赖的任务最终是否真正完成，依赖它的任务无法确定， 所以它 是不可靠的 任务序列 。

比如我们去楼下的老家肉饼吃饭，饭点好了，取餐的时候发生了一些同步异步的事情。
同步：我们都站在队里等着取餐，前面有个人点了一份肉饼，后厨做了很久，但是由于同步机制，我们还是要站在队里等着前面那个人
异步：我们点完餐之后，点餐员给了我们一个取餐号码，跟你说，你不用在这里排队等着，去找个地方坐着玩手机去吧，等饭做好了，

3.阻塞与非阻塞

阻塞和非阻塞这两个概念与程序（线程）等待消息通知(无所谓同步或者异步)时的状态有关。也就是说阻塞与非阻塞主要是程序（线程）等待消息通知时的状态角度来说的

继续上面的那个例子，不论是排队还是使用号码等待通知，如果在这个等待的过程中，等待者除了等待消息通知之外不能做其它的事情
相反，有的人喜欢在等待取餐的时候一边打游戏一边等待，这样的状态就是非阻塞的，因为他(等待者)没有阻塞在这个消息通知上，而是

4.同步/异步 与 阻塞和非阻塞

1. **同步阻塞形式**

效率最低。拿上面的例子来说，就是你专心排队，什么别的事都不做。

2. **异步阻塞形式**

如果在排队取餐的人 采用的是异步的方式去等待消息被触发（通知） ，也就是领了一张小纸条，假如在这段时间里他不能做其它的事情，就在那坐着等着，不能玩游戏等，那么很显然，这个人被阻塞在了这个等待的操作上面；

异步操作是可以被阻塞住的，只不过它不是在处理消息时阻塞，而是在等待消息通知时被阻塞。

3. **同步非阻塞形式**

实际上是效率低下的。

想象一下你一边打着电话一边还需要抬头看到底队伍排到你了没有，如果把打电话和观察排队的位置看成是程序的两个操作的话， 这个程序需要在这两种不同的行为之间来回的切换 ，效率可想而知是低下的。

4. **异步非阻塞形式**

效率更高，

因为打电话是你(等待者)的事情，而通知你则是柜台(消息触发机制)的事情， 程序没有在两种不同的操作中来回切换 。

比如说，这个人突然发觉自己烟瘾犯了，需要出去抽根烟，于是他告诉点餐员说，排到我这个号码的时候麻烦到外面通知我一下，那么他就没有被阻塞在这个等待的操作上面，自然这个就是异步+非阻塞的方式了。

很多人会把同步和阻塞混淆，是因为很多时候同步操作会以阻塞的形式表现出来，同样的，很多人也会把异步和非阻塞混淆，因为异步操作一般都不会在真正的IO操作处被阻塞。

六 进程的创建、结束与并发的实现（了解）

1.进程的创建

但凡是硬件，都需要有操作系统去管理，只要有操作系统，就有进程的概念，就需要有创建进程的方式，一些操作系统只为一个应用程序设计，比如微波炉中的控制器，一旦启动微波炉，所有的进程都已经存在。

而对于通用系统（跑很多应用程序），需要有系统运行过程中创建或撤销进程的能力，主要分为4中形式创建新的进程

1. 系统初始化（查看进程linux中用ps命令，windows中用任务管理器，前台进程负责与用户交互，后台运行的进程与用户无关，运行在后台并且只在需要时才唤醒的进程，称为守护进程，如电子邮件、web页面、新闻、打印）

2. 一个进程在运行过程中开启了子进程（如nginx开启多进程，os.fork,subprocess.Popen等）

3. 用户的交互式请求，而创建一个新进程（如用户双击暴风影音）

4. 一个批处理作业的初始化（只在大型机的批处理系统中应用）

无论哪一种，新进程的创建都是由一个已经存在的进程执行了一个用于创建进程的系统调用而创建的：

1. 在UNIX中该系统调用是：fork，fork会创建一个与父进程一模一样的副本，二者有相同的存储映像、同样的环境字符串和同样的打开文件（在shell解释器进程中，执行一个命令就会创建一个子进程）

2. 在windows中该系统调用是：CreateProcess，CreateProcess既处理进程的创建，也负责把正确的程序装入新进程。

关于创建的子进程，UNIX和windows

1.相同的是：进程创建后，父进程和子进程有各自不同的地址空间（**多道技术要求物理层面实现进程之间内存的隔离**），任何一个进程的在其地址空间中的修改都不会影响到另外一个进程。

2.不同的是：在UNIX中，子进程的初始地址空间是父进程的一个副本，提示：子进程和父进程是可以有只读的共享内存区的。但是对于windows系统来说，从一开始父进程与子进程的地址空间就是不同的。

2.进程的结束

1. 正常退出（自愿，如用户点击交互式页面的叉号，或程序执行完毕调用发起系统调用正常退出，在linux中用exit，在windows中用ExitProcess）

2. 出错退出（自愿，python a.py中a.py不存在）

3. 严重错误（非自愿，执行非法指令，如引用不存在的内存，1/0等，可以捕捉异常，try...except...）

4. 被其他进程杀死（非自愿，如kill -9）

3.进程并发的实现（了解）

进程并发的实现在于，硬件中断一个正在运行的进程，把此时进程运行的所有状态保存下来，为此，操作系统维护一张表格，即进程表（process table），每个进程占用一个进程表项（这些表项也称为进程控制块）

进程管理	存储管理	文件管理
寄存器 程序计数器 程序状态字 堆栈指针 进程状态 优先级 调度参数 进程ID 父进程 进程组 信号 进程开始时间 使用的CPU时间 子进程的CPU时间 下次报警时间	正文段指针 数据段指针 堆栈指针	根目录 工作目录 文件描述符 用户ID 组ID

该表存放了进程状态的重要信息：程序计数器、堆栈指针、内存分配状况、所有打开文件的状态、帐号和调度信息，以及其他在进程由运行态转为就绪态或阻塞态时，必须保存的信息，从而保证该进程在再次启动时，就像从未被中断过一样。

==========

上面的内容都是进程的一些理论基础，下面的内容是python中进程的应用实战

==========

今天的内容就到这个地方吧，同学们好好整理一下~~~~~~~~~~~~~~~~

通过上面内容的学习，我们已经了解了很多进程相关的理论知识，了解进程是什么应该不再困难了，刚刚我们已经了解了，运行中的程序就是一个进程。所有的进程都是通过它的父进程来创建的。因此，运行起来的python**程序也是一个进程，那么我们也可以在程序中再创建进程。多个进程可以实现并发效果，也就是说，当我们的程序中存在多个进程的时候，在某些时候，就会让程序的执行速度变快。以我们之前所学的知识，并不能实现创建进程这个功能，所以我们就需要借助**python**中强大的模块。**

七 multiprocess模块

仔细说来，multiprocess不是一个模块而是python中一个操作、管理进程的包。 之所以叫multi是取自multiple的多功能的意思,在这个包中几乎包含了和进程有关的所有子模块。由于提供的子模块非常多，为了方便大家归类记忆，我将这部分大致分为四个部分：创建进程部分，进程同步部分，进程池部分，进程之间数据共享。重点强调：进程没有任何共享状态，进程修改的数据，改动仅限于该进程内，但是通过一些特殊的方法，可以实现进程之间数据的共享。

1.process模块介绍

process模块是一个创建进程的模块，借助这个模块，就可以完成进程的创建。

```
Process([group [, target [, name [, args [, kwargs]]]]])，由该类实例化得到的对象，表示一个子进程中的任务（尚未启动）

强调：
1. 需要使用关键字的方式来指定参数
2. args指定的为传给target函数的位置参数，是一个元组形式，必须有逗号
```

我们先写一个程序来看看：

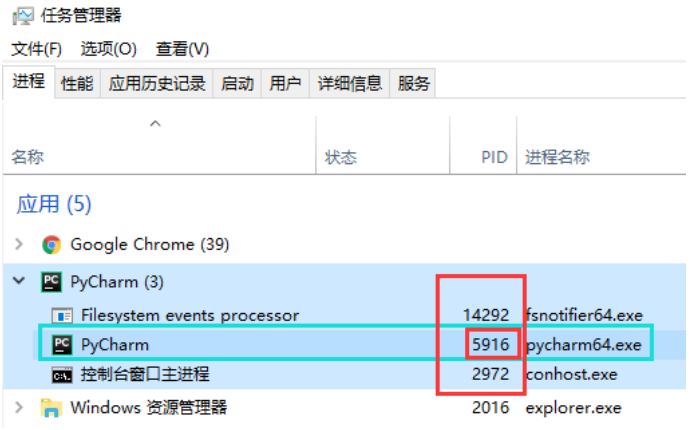
```
#当前文件名称为test.py# from multiprocessing import Process
#
# def func():
#     print(12345)
#
```

```
# if __name__ == '__main__': #windows 下才需要写这个，这和系统创建进程的机制有关系，不用深究，记着windows下要写就好啦
#     #首先我运行当前这个test.py文件，运行这个文件的程序，那么就产生了进程，这个进程我们称为主进程
#
#     p = Process(target=func,) #将函数注册到一个进程中，p是一个进程对象，此时还没有启动进程，只是创建了一个进程对象。并
#     p.start() #告诉操作系统，给我开启一个进程，func这个函数就被我们新开的这个进程执行了，而这个进程是我主进程运行过程中
#     print('*' * 10) #这是主进程的程序，上面开启的子进程的程序是和主进程的程序同时运行的，我们称为异步
```

上面说了，我们通过主进程创建的子进程是异步执行的，那么我们就验证一下，并且看一下子进程和主进程(也就是父进程)的ID号（讲一下pid和ppid，使用pycharm举例），来看看是否是父子关系。

```
import time
import os

#os.getpid() 获取自己进程的ID号
#os.getppid() 获取自己进程的父进程的ID号

from multiprocessing import Process

def func():
    print('aaaa')
    time.sleep(1)
    print('子进程>>',os.getpid())
    print('该子进程的父进程>>',os.getppid())
    print(12345)

if __name__ == '__main__':
    #首先我运行当前这个文件，运行的这个文件的程序，那么就产生了主进程

    p = Process(target=func,)
    p.start()
    print('*' * 10)
    print('父进程>>',os.getpid())
    print('父进程的父进程>>',os.getppid())

#加上time和进程号给大家看一看结果：
#********** 首先打印出来了出进程的程序，然后打印的是子进程的，也就是子进程是异步执行的，相当于主进程和子进程同时运行着，
#父进程>> 3308
#父进程的父进程>> 5916 #我运行的test.py文件的父进程号，它是pycharm的进程号，看下面的截图

#aaaa
#子进程>> 4536
#该子进程的父进程>> 3308 #是我主进程的ID号，说明主进程为它的父进程

#12345
```

打开windows下的任务管理器，**看**pycharm**的**pid**进程号，是我们上面运行的**test.py**这个文件主进程的父进程号：**



看一个问题，说明linux和windows两个不同的操作系统创建进程的不同机制导致的不同结果：

```
import time
import os
from multiprocessing import Process

def func():
    print('aaaa')
    time.sleep(1)
    print('子进程>>',os.getpid())
    print('该子进程的父进程>>',os.getppid())
    print(12345)

print('太白老司机~~~~') #如果我在这里加了一个打印，你会发现运行结果中会出现两次打印出来的太白老司机，因为我们在主进程中
#其实是因为windows开起进程的机制决定的，在linux下是不存在这个效果的，因为windows使用的是process方法来开启进程，他就
要加上if __name__ == '__main__':，否则会出现子进程中运行的时候还开启子进程，那就出现无限循环的创建进程了，就报错了
```

一个进程的生命周期：如果子进程的运行时间长，那么等到子进程执行结束程序才结束，如果主进程的执行时间长，那么主进程执行结束程序才结束，实际上我们在子进程中打印的内容是在主进程的执行结果中看不出来的，但是pycharm帮我们做了优化，因为它会识别到你这是开的子进程，帮你把子进程中打印的内容打印到了显示台上。

如果说一个主进程运行完了之后，我们把pycharm关了，但是子进程还没有执行结束，那么子进程还存在吗？这要看你的进程是如何配置的，如果说我们没有配置说我主进程结束，子进程要跟着结束，那么主进程结束的时候，子进程是不会跟着结束的，他会自己执行完，如果我设定的是主进程结束，子进程必须跟着结束，那么就不会出现单独的子进程（孤儿进程）了，具体如何设置，看下面的守护进程的讲解。比如说，我们将来启动项目的时候，可能通过cmd来启动，那么我cmd关闭了你的项目就会关闭吗，不会的，因为你的项目不能停止对外的服务，对吧。

Process类中参数的介绍：

```
参数介绍：
1 group参数未使用，值始终为None
2 target表示调用对象，即子进程要执行的任务
3 args表示调用对象的位置参数元组，args=(1,2,'egon',)
4 kwargs表示调用对象的字典,kwargs={'name':'egon','age':18}
5 name为子进程的名称
```

给要执行的函数传参数：

```
def func(x,y):
    print(x)
    time.sleep(1)
    print(y)

if __name__ == '__main__':

    p = Process(target=func,args=('姑娘','来玩啊！'))#这是func需要接收的参数的传送方式。
    p.start()
    print('父进程执行结束！')

#执行结果：
父进程执行结束！
姑娘
来玩啊！
```

Process类中各方法的介绍：

```
1 p.start()：启动进程，并调用该子进程中的p.run()
2 p.run():进程启动时运行的方法，正是它去调用target指定的函数，我们自定义类的类中一定要实现该方法
3 p.terminate():强制终止进程p，不会进行任何清理操作，如果p创建了子进程，该子进程就成了僵尸进程，使用该方法需要特别小心这
4 p.is_alive():如果p仍然运行，返回True
5 p.join([timeout]):主线程等待p终止（强调：是主线程处于等的状态，而p是处于运行的状态）。timeout是可选的超时时间，需要强
```

join方法的例子：

让主进程加上join的地方等待（也就是阻塞住），等待子进程执行完之后，再继续往下执行我的主进程，好多时候，我们主进程需要子进程的执行结果，所以必须要等待。join感觉就像是将子进程和主进程拼接起来一样，将异步改为同步执行。

```
def func(x,y):
    print(x)
    time.sleep(1)
    print(y)

if __name__ == '__main__':

    p = Process(target=func,args=('姑娘','来玩啊！'))
    p.start()
    print('我这里是异步的啊！') #这里相对于子进程还是异步的
    p.join() #只有在join的地方才会阻塞住，将子进程和主进程之间的异步改为同步
    print('父进程执行结束！')

#打印结果：
我这里是异步的啊！
姑娘
来玩啊！
父进程执行结束！
```

怎么样开启多个进程呢？for循环。并且我有个需求就是说，所有的子进程异步执行，然后所有的子进程全部执行完之后，我再执行主进程，怎么搞？看代码

```
#下面的注释按照编号去看，别忘啦！
import time
import os
from multiprocessing import Process

def func(x,y):
    print(x)
    # time.sleep(1) #进程切换：如果没有这个时间间隔，那么你会发现func执行结果是打印一个x然后一个y，再打印一个x一个y，不
    print(y)

if __name__ == '__main__':

    p_list= []
    for i in range(10):
        p = Process(target=func,args=('姑娘%s'%i,'来玩啊！'))
        p_list.append(p)
        p.start()

    [ap.join() for ap in p_list] #4、这是解决办法，前提是我们的子进程全部都已经去执行了，那么我在一次给所有正在执行的子进程加

        # p.join() #1、如果加到for循环里面，那么所有子进程包括父进程就全部变为同步了，因为for循环也是主进程的，循环第一次的
        #2、如果我不想这样的，也就是我想所有的子进程是异步的，然后所有的子进程执行完了再执行主进程
    #p.join() #3、如果这样写的话，多次运行之后，你会发现会出现主进程的程序比一些子进程先执行完，因为我们p.join()是对最后一
                    #完了，那么就会先打印主进程的内容了，这个cpu调度进程的机制有关系，因为我们的电脑可能只有4个cpu，我的
            #程给cpu去执行，这里也解释了我们打印出来的子进程中的内容也是没有固定顺序的原因，因为打印结果也需要调用cpu，可
    print('不要钱~~~~~~~~~~~~~~~~！')
```

模拟两个应用场景：1、同时对一个文件进行写操作 2、同时创建多个文件

```
import time
import os
import re
from multiprocessing import Process
#多进程同时对一个文件进行写操作
def func(x,y,i):
    with open(x,'a',encoding='utf-8') as f:
        print('当前进程%s拿到的文件的光标位置>>%s'%(os.getpid(),f.tell()))
        f.write(y)

#多进程同时创建多个文件
# def func(x, y):
#     with open(x, 'w', encoding='utf-8') as f:
#         f.write(y)

if __name__ == '__main__':

    p_list= []
```

```
    for i in range(10):
        p = Process(target=func,args=('can_do_girl_lists.txt','姑娘%s'%i,i))
        # p = Process(target=func,args=('can_do_girl_info%s.txt'%i,'姑娘电话0000%s'%i))
        p_list.append(p)
        p.start()

    [ap.join() for ap in p_list] #这就是个for循环，只不过用列表生成式的形式写的
    with open('can_do_girl_lists.txt','r',encoding='utf-8') as f:
        data = f.read()
        all_num = re.findall('\d+',data) #打开文件，统计一下里面有多少个数据，每个数据都有个数字，所以re匹配一下就行了
        print('>>>>>',all_num,'.....%s'%(len(all_num)))
    #print([i in in os.walk(r'你的文件夹路径')])
    print('不要钱~~~~~~~~~~~~~~~~~~~！ ')
```

Process类中自带封装的各属性的介绍

```
1 p.daemon：默认值为False，如果设为True，代表p为后台运行的守护进程，当p的父进程终止时，p也随之终止，并且设定为True后
2 p.name:进程的名称
3 p.pid：进程的pid
4 p.exitcode:进程在运行时为None、如果为-N，表示被信号N结束(了解即可)
5 p.authkey:进程的身份验证键,默认是由os.urandom()随机生成的32字符的字符串。这个键的用途是为涉及网络连接的底层进程间通
```

2.Process类的使用

注意：在windows中Process()必须放到# if __name__ == '__main__':下

```
Since Windows has no fork, the multiprocessing module starts a new Python process and imports the calling module.
If Process() gets called upon import, then this sets off an infinite succession of new processes (or until your machine runs
This is the reason for hiding calls to Process() inside

if __name__ == "__main__"
since statements inside this if-statement will not get called upon import.
由于Windows没有fork，多处理模块启动一个新的Python进程并导入调用模块。
如果在导入时调用Process（），那么这将启动无限继承的新进程（或直到机器耗尽资源）。
这是隐藏对Process（）内部调用的原，使用if __name__ == “__main __”，这个if语句中的语句将不会在导入时被调用。
```

进程的创建**第二种方法（继承）**

```
class MyProcess(Process): #自己写一个类，继承Process类
    #我们通过init方法可以传参数，如果只写一个run方法，那么没法传参数，因为创建对象的是传参就是在init方法里面，面向对象的时
    def __init__(self,person):
        super().__init__()
        self.person=person
    def run(self):
        print(os.getpid())
        print(self.pid)
        print(self.pid)
```

```
        print('%s 正在和女主播聊天' %self.person)
    # def start(self):
    #     #如果你非要写一个start方法，可以这样写，并且在run方法前后，可以写一些其他的逻辑
    #     self.run()
if __name__ == '__main__':
    p1=MyProcess('Jedan')
    p2=MyProcess('太白')
    p3=MyProcess('alexDSB')

    p1.start() #start内部会自动调用run方法
    p2.start()
    # p2.run()
    p3.start()


    p1.join()
    p2.join()
    p3.join()
```

进程之间的数据是隔离的：

```
#我们说进程之间的数据是隔离的，也就是数据不共享，看下面的验证
from multiprocessing import Process
n=100 #首先我定义了一个全局变量，在windows系统中应该把全局变量定义在if __name__ == '__main__'之上就可以了
def work():
    global n
    n=0
    print('子进程内: ',n)

if __name__ == '__main__':
    p=Process(target=work)
    p.start()
    p.join() #等待子进程执行完毕，如果数据共享的话，我子进程是不是通过global将n改为0了，但是你看打印结果，主进程在子进程
    print('主进程内: ',n)

#看结果：
# 子进程内：  0
# 主进程内：  100
```

练习：我们之前学socket的时候，知道tcp协议的socket是不能同时和多个客户端进行连接的，(这里先不考虑socketserver那个模块)，对不对，那我们自己通过多进程来实现一下同时和多个客户端进行连接通信。

服务端代码示例：（注意一点：通过这个是不能做qq聊天的，因为qq聊天是qq的客户端把信息发给另外一个qq的客户端，中间有一个服务端帮你转发消息，而不是我们这样的单纯的客户端和服务端对话，并且子进程开启之后咱们是没法操作的，并且没有为子进程input输入提供控制台，所有你再在子进程中写上了input会报错，EOFError错误，这个错误的意思就是你的input需要输入，但是你输入不了，就会报这个错误。而子进程的输出打印之类的，是pycharm做了优化，将所有子进程中的输出结果帮你打印出来了，但实质还是不同进程的。）

```
from socket import *
from multiprocessing import Process
```

```
def talk(conn,client_addr):
    while True:
        try:
            msg=conn.recv(1024)
            print('客户端消息>>',msg)
            if not msg:break
            conn.send(msg.upper())
            #在这里有同学可能会想，我能不能在这里写input来自己输入内容和客户端进行对话？朋友，是这样的，按说是可以的，但是
        except Exception:
            break

if __name__ == '__main__': #windows下start进程一定要写到这下面
    server = socket(AF_INET, SOCK_STREAM)
    # server.setsockopt(SOL_SOCKET, SO_REUSEADDR,1）# 如果你将如果你将bind这些代码写到if __name__ == '__main__'这行
    server.bind(('127.0.0.1', 8080))
    #有同学可能还会想，我为什么多个进程就可以连接一个server段的一个ip和端口了呢，我记得当时说tcp的socket的时候，我是不能
    server.listen(5)
    while True:
        conn,client_addr=server.accept()
        p=Process(target=talk,args=(conn,client_addr))
        p.start()
```

客户端代码示例：

```
from socket import *

client=socket(AF_INET,SOCK_STREAM)
client.connect(('127.0.0.1',8080))


while True:
    msg=input('>>: ').strip()
    if not msg:continue

    client.send(msg.encode('utf-8'))
    msg=client.recv(1024)
    print(msg.decode('utf-8'))
```

上面我们通过多进程实现了并发，但是有个问题

```
每来一个客户端，都在服务端开启一个进程，如果并发来一个万个客户端，要开启一万个进程吗，你自己尝试着在你自己的机器上开启
解决方法：进程池，本篇博客后面会讲到，大家继续学习呀
```

Process对象的其他方法或属性（简单了解一下就可以啦）

```
#进程对象的其他方法一:terminate,is_alive
from multiprocessing import Process
```

```
import time
import random

class Piao(Process):
    def __init__(self,name):
        self.name=name
        super().__init__()

    def run(self):
        print('%s is 打飞机' %self.name)
        # s = input('???') #别忘了再pycharm下子进程中不能input输入，会报错EOFError: EOF when reading a line，因为子进程中没
        time.sleep(2)
        print('%s is 打飞机结束' %self.name)

if __name__ == '__main__':
    p1=Piao('太白')
    p1.start()
    time.sleep(5)
    p1.terminate()#关闭进程,不会立即关闭,有个等着操作系统去关闭这个进程的时间,所以is_alive立刻查看的结果可能还是存活，但是
    print(p1.is_alive()) #结果为True
    print('等会。。。。')
    time.sleep(1)
    print(p1.is_alive()) #结果为False
```

```
from multiprocessing import Process
import time
import random
class Piao(Process):
    def __init__(self,name):
        # self.name=name
        # super().__init__() #Process的__init__方法会执行self.name=Piao-1,
        #                    #所以加到这里,会覆盖我们的self.name=name

        #为我们开启的进程设置名字的做法
        super().__init__()
        self.name=name

    def run(self):
        print('%s is piaoing' %self.name)
        time.sleep(random.randrange(1,3))
        print('%s is piao end' %self.name)

p=Piao('egon')
p.start()
print('开始')
print(p.pid) #查看pid
```

僵尸进程与孤儿进程（简单了解 一下就可以啦）

```
参考博客：http://www.cnblogs.com/Anker/p/3271773.html
```

一：僵尸进程（有害）
　　僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进

我们知道在unix/linux中，正常情况下子进程是通过父进程创建的，子进程在创建新的进程。子进程的结束和父进程的运行是一个异步

因此，UNIX提供了一种机制可以保证父进程可以在任意时刻获取子进程结束时的状态信息：
1、在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。但是仍然为其保留一定的信息（包括进程号
2、直到父进程通过wait / waitpid来取时才释放. 但这样就导致了问题，如果进程不调用wait / waitpid的话，那么保留的那段信息就不

　　任何一个子进程(init除外)在exit()之后，并非马上就消失掉，而是留下一个称为僵尸进程(Zombie)的数据结构，等待父进程处理。

二：孤儿进程（无害）

　　孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号

　　孤儿进程是没有父进程的进程，孤儿进程这个重任就落到了init进程身上，init进程就好像是一个民政局，专门负责处理孤儿进程的

我们来测试一下（创建完子进程后，主进程所在的这个脚本就退出了，当父进程先于子进程结束时，子进程会被init收养，成为孤儿进程

```
import os
import sys
import time

pid = os.getpid()
ppid = os.getppid()
print 'im father', 'pid', pid, 'ppid', ppid
pid = os.fork()
#执行pid=os.fork()则会生成一个子进程
#返回值pid有两种值：
#    如果返回的pid值为0，表示在子进程当中
#    如果返回的pid值>0，表示在父进程当中
if pid > 0:
    print 'father died..'
    sys.exit(0)

# 保证主线程退出完毕
time.sleep(1)
print 'im child', os.getpid(), os.getppid()
```

执行文件，输出结果：

```
im father pid 32515 ppid 32015
father died..
im child 32516 1
```

看，子进程已经被pid为1的init进程接收了，所以僵尸进程在这种情况下是不存在的，存在只有孤儿进程而已，孤儿进程声明周期结束自

三：僵尸进程危害场景：

　　例如有个进程，它定期的产 生一个子进程，这个子进程需要做的事情很少，做完它该做的事情之后就退出了，因此这个子进程的生

四：测试

```
#1、产生僵尸进程的程序test.py内容如下

#coding:utf-8
from multiprocessing import Process
import time,os

def run():
    print('子',os.getpid())
```

```
if __name__ == '__main__':
    p=Process(target=run)
    p.start()

    print('主',os.getpid())
    time.sleep(1000)


#2、在unix或linux系统上执行
[root@vm172-31-0-19 ~]# python3  test.py &
[1] 18652
[root@vm172-31-0-19 ~]# 主 18652
子 18653

[root@vm172-31-0-19 ~]# ps aux |grep Z
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root     18653  0.0  0.0      0     0 pts/0    Z    20:02   0:00 [python3] <defunct> #出现僵尸进程
root     18656  0.0  0.0 112648   952 pts/0    S+   20:02   0:00 grep --color=auto Z

[root@vm172-31-0-19 ~]# top #执行top命令发现1zombie
top - 20:03:42 up 31 min,  3 users,  load average: 0.01, 0.06, 0.12
Tasks:  93 total,   2 running,  90 sleeping,   0 stopped,   1 zombie
%Cpu(s):  0.0 us,  0.3 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1016884 total,    97184 free,    70848 used,   848852 buff/cache
KiB Swap:        0 total,        0 free,        0 used.   782540 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
root      20   0   29788   1256    988 S  0.3  0.1   0:01.50 elfin


#3、
等待父进程正常结束后会调用wait / waitpid去回收僵尸进程
但如果父进程是一个死循环，永远不会结束，那么该僵尸进程就会一直存在，僵尸进程过多，就是有害的
解决方法一：杀死父进程
解决方法二：对开启的子进程应该记得使用join，join会回收僵尸进程
参考python2源码注释
class Process(object):
    def join(self, timeout=None):
        '''
        Wait until child process terminates
        '''
        assert self._parent_pid == os.getpid(), 'can only join a child process'
        assert self._popen is not None, 'can only join a started process'
        res = self._popen.wait(timeout)
        if res is not None:
            _current_process._children.discard(self)

join方法中调用了wait，告诉系统释放僵尸进程。discard为从自己的children中剔除

解决方法三：http://blog.csdn.net/u010571844/article/details/50419798
```


3.守护进程

之前我们讲的子进程是不会随着主进程的结束而结束，子进程全部执行完之后，程序才结束，那么如果有一天我们的需求是我的主进程结束了，由我主进程创建的那些子进程必须跟着结束，怎么办？守护进程就来了！

主进程创建守护进程

其一：守护进程会在主进程代码执行结束后就终止

其二：守护进程内无法再开启子进程,否则抛出异常：AssertionError: daemonic processes are not allowed to have children

注意：进程之间是互相独立的，主进程代码运行结束，守护进程随即终止

```
import os
import time
from multiprocessing import Process

class Myprocess(Process):
    def __init__(self,person):
        super().__init__()
        self.person = person
    def run(self):
        print(os.getpid(),self.name)
        print('%s正在和女主播聊天' %self.person)
        time.sleep(3)
if __name__ == '__main__':
    p=Myprocess('太白')
    p.daemon=True #一定要在p.start()前设置,设置p为守护进程,禁止p创建子进程,并且父进程代码执行结束,p即终止运行
    p.start()
    # time.sleep(1) # 在sleep时linux下查看进程id对应的进程ps -ef|grep id
    print('主')
```

4.进程同步（锁）

通过刚刚的学习，我们千方百计实现了程序的异步，让多个任务可以同时在几个进程中并发处理，他们之间的运行没有顺序，一旦开启也不受我们控制。尽管并发编程让我们能更加充分的利用IO资源，但是也给我们带来了新的问题：进程之间数据不共享,但是共享同一套文件系统,所以访问同一个文件,或同一个打印终端,是没有问题的，而共享带来的是竞争，竞争带来的结果就是错乱，如何控制，就是加锁处理。

```
import os
import time
import random
from multiprocessing import Process

def work(n):
    print('%s: %s is running' %(n,os.getpid()))
    time.sleep(random.random())
    print('%s:%s is done' %(n,os.getpid()))

if __name__ == '__main__':
    for i in range(5):
        p=Process(target=work,args=(i,))
        p.start()

# 看结果：通过结果可以看出两个问题：问题一：每个进程中work函数的第一个打印就不是按照我们for循环的0-4的顺序来打印的
#问题二：我们发现，每个work进程中有两个打印，但是我们看到所有进程中第一个打印的顺序为0-2-1-4-3，但是第二个打印没有按
#问题的解决方法，第二个问题加锁来解决，第一个问题是没有办法解决的，因为进程开到了内核，有操作系统来决定进程的调度，我们
# 0: 9560 is running
# 2: 13824 is running
# 1: 7476 is running
# 4: 11296 is running
```

```
# 3: 14364 is running

# 2:13824 is done
# 1:7476 is done
# 0:9560 is done
# 3:14364 is done
# 4:11296 is done
```

```
#由并发变成了串行,牺牲了运行效率,但避免了竞争
from multiprocessing import Process,Lock
import os,time
def work(n,lock):
    #加锁，保证每次只有一个进程在执行锁里面的程序，这一段程序对于所有写上这个锁的进程，大家都变成了串行
    lock.acquire()
    print('%s: %s is running' %(n,os.getpid()))
    time.sleep(1)
    print('%s:%s is done' %(n,os.getpid()))
    #解锁，解锁之后其他进程才能去执行自己的程序
    lock.release()
if __name__ == '__main__':
    lock=Lock()
    for i in range(5):
        p=Process(target=work,args=(i,lock))
        p.start()

#打印结果:
# 2: 10968 is running
# 2:10968 is done
# 0: 7932 is running
# 0:7932 is done
# 4: 4404 is running
# 4:4404 is done
# 1: 12852 is running
# 1:12852 is done
# 3: 980 is running
# 3:980 is done

#结果分析：（自己去多次运行一下，看看结果，我拿出其中一个结果来看）通过结果我们可以看出，多进程刚开始去执行的时候，每次
```

上面这种情况虽然使用加锁的形式实现了顺序的执行，但是程序又重新变成串行了，这样确实会浪费了时间，却保证了数据的安全。

接下来，我们以模拟抢票为例，来看看数据安全的重要性。

```
#注意：首先在当前文件目录下创建一个名为db的文件
#文件db的内容为：{"count":1}，只有这一行数据,并且注意，每次运行完了之后，文件中的1变成了0，你需要手动将0改为1，然后在
#注意一定要用双引号，不然json无法识别
#并发运行，效率高，但竞争写同一文件，数据写入错乱
from multiprocessing import Process,Lock
import time,json,random
```

```
#查看剩余票数
def search():
    dic=json.load(open('db')) #打开文件，直接load文件中的内容，拿到文件中的包含剩余票数的字典
    print('\033[43m剩余票数%s\033[0m' %dic['count'])

#抢票
def get():
    dic=json.load(open('db'))
    time.sleep(0.1)      #模拟读数据的网络延迟，那么进程之间的切换，导致所有人拿到的字典都是{"count": 1}，也就是每个人都拿到
    if dic['count'] >0:
        dic['count']-=1
        time.sleep(0.2)  #模拟写数据的网络延迟
        json.dump(dic,open('db','w'))
        #最终结果导致，每个人显示都抢到了票，这就出现了问题~
        print('\033[43m购票成功\033[0m')

def task():
    search()
    get()

if __name__ == '__main__':
    for i in range(3): #模拟并发100个客户端抢票
        p=Process(target=task)
        p.start()

#看结果分析：由于网络延迟等原因使得进程切换，导致每个人都抢到了这最后一张票
# 剩余票数1
# 剩余票数1
# 剩余票数1
# 购票成功
# 购票成功
# 购票成功
```

```
#注意：首先在当前文件目录下创建一个名为db的文件
#文件db的内容为：{"count":1}，只有这一行数据,并且注意，每次运行完了之后，文件中的1变成了0，你需要手动将0改为1，然后在
#注意一定要用双引号，不然json无法识别
#加锁保证数据安全，不出现混乱
from multiprocessing import Process,Lock
import time,json,random

#查看剩余票数
def search():
    dic=json.load(open('db')) #打开文件，直接load文件中的内容，拿到文件中的包含剩余票数的字典
    print('\033[43m剩余票数%s\033[0m' %dic['count'])

#抢票
def get():
    dic=json.load(open('db'))
    time.sleep(0.1)      #模拟读数据的网络延迟，那么进程之间的切换，导致所有人拿到的字典都是{"count": 1}，也就是每个人都拿到
    if dic['count'] >0:
        dic['count']-=1
        time.sleep(0.2)  #模拟写数据的网络延迟
        json.dump(dic,open('db','w'))
        #最终结果导致，每个人显示都抢到了票，这就出现了问题~
        print('\033[43m购票成功\033[0m')
    else:
```

```
        print('sorry，没票了亲！')
def task(lock):
    search()
    #因为抢票的时候是发生数据变化的时候，所有我们将锁加加到这里
    lock.acquire()
    get()
    lock.release()
if __name__ == '__main__':
    lock = Lock() #创建一个锁
    for i in range(3): #模拟并发100个客户端抢票
        p=Process(target=task,args=(lock,)) #将锁作为参数传给task函数
        p.start()

#看结果分析：只有一个人抢到了票
# 剩余票数1
# 剩余票数1
# 剩余票数1
# 购票成功   #幸运的人儿
# sorry，没票了亲!
# sorry，没票了亲!
```

进程锁总结：

```
#加锁可以保证多个进程修改同一块数据时，同一时间只能有一个任务可以进行修改，即串行的修改，没错，速度是慢了，但牺牲了速
虽然可以用文件共享数据实现进程间通信，但问题是：
1.效率低（共享数据基于文件，而文件是硬盘上的数据）
2.需要自己加锁处理

#因此我们最好找寻一种解决方案能够兼顾：1、效率高（多个进程共享一块内存的数据）2、帮我们处理好锁问题。这就是mutiproce
队列和管道都是将数据存放于内存中
队列又是基于（管道+锁）实现的，可以让我们从复杂的锁问题中解脱出来，
我们应该尽量避免使用共享数据，尽可能使用消息传递和队列，避免处理复杂的同步和锁问题，而且在进程数目增多时，往往可以获得

IPC通信机制（了解）：IPC是intent-Process Communication的缩写，含义为进程间通信或者跨进程通信，是指两个进程之间进行数
```

第二天进程的学习就到这里啦~~~~~~

5.队列（推荐使用）

进程彼此之间互相隔离，要实现进程间通信（IPC），multiprocessing模块支持两种形式：队列和管道，这两种方式都是使用消息传递的。队列就像一个特殊的列表，但是可以设置固定长度，并且从前面插入数据，从后面取出数据，先进先出。

```
Queue([maxsize]) 创建共享的进程队列。
参数 : maxsize是队列中允许的最大项数。如果省略此参数，则无大小限制。
底层队列使用管道和锁实现。
```

先看下面的代码示例，然后再看方法介绍。

queue的方法介绍

```
q = Queue([maxsize])
```

创建共享的进程队列。maxsize是队列中允许的最大项数。如果省略此参数，则无大小限制。底层队列使用管道和锁定实现。另外，还
Queue的实例q具有以下方法：

q.get([block [,timeout]])
返回q中的一个项目。如果q为空，此方法将阻塞，直到队列中有项目可用为止。block用于控制阻塞行为，默认为True. 如果设置为Fal

q.get_nowait( )
同q.get(False)方法。

q.put(item [, block [,timeout]])
将item放入队列。如果队列已满，此方法将阻塞至有空间可用为止。block控制阻塞行为，默认为True。如果设置为False，将引发Que

q.qsize()
返回队列中目前项目的正确数量。此函数的结果并不可靠，因为在返回结果和在稍后程序中使用结果之间，队列中可能添加或删除了项

q.empty()
如果调用此方法时 q为空，返回True。如果其他进程或线程正在往队列中添加项目，结果是不可靠的。也就是说，在返回和使用结果之

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

queue的其他方法（了解）

q.close()
关闭队列，防止队列中加入更多数据。调用此方法时，后台线程将继续写入那些已入队列但尚未写入的数据，但将在此方法完成时马上

q.cancel_join_thread()
不会再进程退出时自动连接后台线程。这可以防止join_thread()方法阻塞。

q.join_thread()
连接队列的后台线程。此方法用于在调用q.close()方法后，等待所有队列项被消耗。默认情况下，此方法由不是q的原始创建者的所有进

我们看一些代码示例：

```
from multiprocessing import Queue
q=Queue(3) #创建一个队列对象，队列长度为3

#put ,get ,put_nowait,get_nowait,full,empty
q.put(3)  #往队列中添加数据
q.put(2)
q.put(1)
# q.put(4)  # 如果队列已经满了，程序就会停在这里，等待数据被别人取走，再将数据放入队列。
        # 如果队列中的数据一直不被取走，程序就会永远停在这里。
try:
    q.put_nowait(4) # 可以使用put_nowait，如果队列满了不会阻塞，但是会因为队列满了而报错。
except: # 因此我们可以用一个try语句来处理这个错误。这样程序不会一直阻塞下去，但是会丢掉这个消息。
    print('队列已经满了')

# 因此，我们再放入数据之前，可以先看一下队列的状态，如果已经满了，就不继续put了。
print(q.full()) #查看是否满了，满了返回True，不满返回False
```

```
print(q.get())  #取出数据
print(q.get())
print(q.get())
# print(q.get()) # 同put方法一样，如果队列已经空了，那么继续取就会出现阻塞。
try:
    q.get_nowait(3) # 可以使用get_nowait，如果队列满了不会阻塞，但是会因为没取到值而报错。
except: # 因此我们可以用一个try语句来处理这个错误。这样程序不会一直阻塞下去。
    print('队列已经空了')

print(q.empty()) #空了
```

```
#看下面的队列的时候，按照编号看注释
import time
from multiprocessing import Process, Queue

# 8. q = Queue(2) #创建一个Queue对象，如果写在这里，那么在windows还子进程去执行的时候，我们知道子进程中还会执行这个
def f(q):
    # q = Queue() #9. 我们在主进程中开启了一个q，如果我们在子进程中的函数里面再开一个q，那么你下面q.put('姑娘，多少钱~')
    q.put('姑娘，多少钱~')  #4.调用主函数中p进程传递过来的进程参数 put函数为向队列中添加一条数据。
    # print(q.qsize()) #6.查看队列中有多少条数据了

def f2(q):
    print('》》》》》》》》')
    print(q.get())  #5.取数据
if __name__ == '__main__':
    q = Queue() #1.创建一个Queue对象
    q.put('小鬼')

    p = Process(target=f, args=(q,)) #2.创建一个进程
    p2 = Process(target=f2, args=(q,)) #3.创建一个进程
    p.start()
    p2.start()
    time.sleep(1) #7.如果阻塞一点时间，就会出现主进程运行太快，导致我们在子进程中查看qsize为1个。
    # print(q.get()) #结果：小鬼
    print(q.get()) #结果：姑娘，多少钱~
    p.join()
```

接下来看一个稍微复杂一些的例子：

```
import os
import time
import multiprocessing

# 向queue中输入数据的函数
def inputQ(queue):
    info = str(os.getpid()) + '(put):' + str(time.asctime())
    queue.put(info)

# 向queue中输出数据的函数
```

```
def outputQ(queue):
    info = queue.get()
    print ('%s%s\033[32m%s\033[0m'%(str(os.getpid()), '(get):',info))

# Main
if __name__ == '__main__':
    #windows下，如果开启的进程比较多的话，程序会崩溃，为了防止这个问题，使用freeze_support()方法来解决。知道就行啦
    multiprocessing.freeze_support()
    record1 = []   # store input processes
    record2 = []   # store output processes
    queue = multiprocessing.Queue(3)

    # 输入进程
    for i in range(10):
        process = multiprocessing.Process(target=inputQ,args=(queue,))
        process.start()
        record1.append(process)

    # 输出进程
    for i in range(10):
        process = multiprocessing.Process(target=outputQ,args=(queue,))
        process.start()
        record2.append(process)

    for p in record1:
        p.join()

    for p in record2:
        p.join()
```

队列是进程安全的：同一时间只能一个进程拿到队列中的一个数据，你拿到了一个数据，这个数据别人就拿不到了。

下面我们来看一个叫做生产者消费者模型的东西：

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

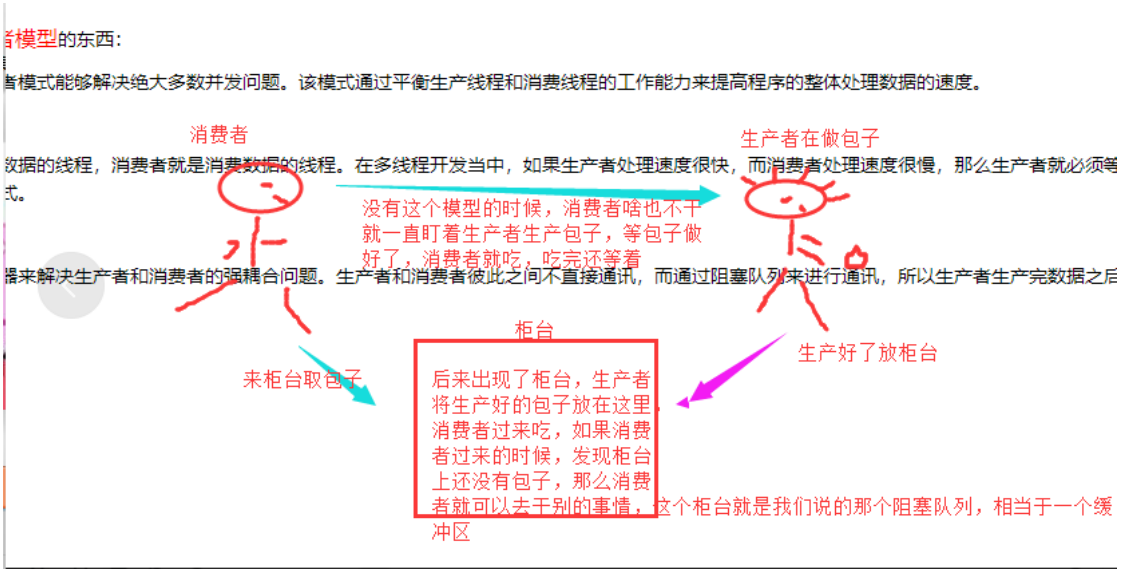
为什么要使用生产者和消费者模式

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

什么是生产者消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力，并且我可以根据生产速度和消费速度来均衡一下多少个生产者可以为多少个消费者提供足够的服务，就可以开多进程等等，而这些进程都是到阻塞队列或者说是缓冲区中去获取或者添加数据。

通俗的解释：看图说话。。背景有点乱，等我更新~~



那么我们基于队列来实现一个生产者消费者模型，代码示例：

```
from multiprocessing import Process,Queue
import time,random,os
def consumer(q):
    while True:
        res=q.get()
        time.sleep(random.randint(1,3))
        print('\033[45m%s 吃 %s\033[0m' %(os.getpid(),res))

def producer(q):
    for i in range(10):
        time.sleep(random.randint(1,3))
        res='包子%s' %i
        q.put(res)
        print('\033[44m%s 生产了 %s\033[0m' %(os.getpid(),res))

if __name__ == '__main__':
    q=Queue()
    #生产者们:即厨师们
    p1=Process(target=producer,args=(q,))

    #消费者们:即吃货们
    c1=Process(target=consumer,args=(q,))

    #开始
    p1.start()
    c1.start()
    print('主')
```

```
#生产者消费者模型总结

    #程序中有两类角色
        一类负责生产数据（生产者）
```

```
    一类负责处理数据（消费者）

#引入生产者消费者模型为了解决的问题是：
    平衡生产者与消费者之间的工作能力，从而提高程序整体处理数据的速度

#如何实现：
    生产者<-->队列<——>消费者
#生产者消费者模型实现类程序的解耦和
```

通过上面基于队列的生产者消费者代码示例，我们发现一个问题：主进程永远不会结束，原因是：生产者p在生产完后就结束了，但是消费者c在取空了q之后，则一直处于死循环中且卡在q.get()这一步。

解决方式无非是让生产者在生产完毕后，往队列中再发一个结束信号，这样消费者在接收到结束信号后就可以break出死循环


```
from multiprocessing import Process,Queue
import time,random,os
def consumer(q):
    while True:
        res=q.get()
        if res is None:break #收到结束信号则结束
        time.sleep(random.randint(1,3))
        print('\033[45m%s 吃 %s\033[0m' %(os.getpid(),res))

def producer(q):
    for i in range(5):
        time.sleep(random.randint(1,3))
        res='包子%s' %i
        q.put(res)
        print('\033[44m%s 生产了 %s\033[0m' %(os.getpid(),res))
    q.put(None) #在自己的子进程的最后加入一个结束信号
if __name__ == '__main__':
    q=Queue()
    #生产者们:即厨师们
    p1=Process(target=producer,args=(q,))

    #消费者们:即吃货们
    c1=Process(target=consumer,args=(q,))

    #开始
    p1.start()
    c1.start()

    print('主')
```

注意：结束信号None，不一定要由生产者发，主进程里同样可以发，但主进程需要等生产者结束后才应该发送该信号

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

```
import time,random,os
def consumer(q):
    while True:
        res=q.get()
        if res is None:break #收到结束信号则结束
        time.sleep(random.randint(1,3))
        print('\033[45m%s 吃 %s\033[0m' %(os.getpid(),res))

def producer(q):
    for i in range(2):
        time.sleep(random.randint(1,3))
        res='包子%s' %i
        q.put(res)
        print('\033[44m%s 生产了 %s\033[0m' %(os.getpid(),res))

if __name__ == '__main__':
    q=Queue()
    #生产者们:即厨师们
    p1=Process(target=producer,args=(q,))

    #消费者们:即吃货们
    c1=Process(target=consumer,args=(q,))

    #开始
    p1.start()
    c1.start()

    p1.join() #等待生产者进程结束
    q.put(None) #发送结束信号
    print('主')
```

但上述解决方式，在有多个生产者和多个消费者时，由于队列我们说了是进程安全的，我一个进程拿走了结束信号，另外一个进程就拿不到了，还需要多发送一个结束信号，有几个取数据的进程就要发送几个结束信号，我们则需要用一个很low的方式去解决

```
from multiprocessing import Process,Queue
import time,random,os
def consumer(q):
    while True:
        res=q.get()
        if res is None:break #收到结束信号则结束
        time.sleep(random.randint(1,3))
        print('\033[45m%s 吃 %s\033[0m' %(os.getpid(),res))

def producer(name,q):
    for i in range(2):
        time.sleep(random.randint(1,3))
        res='%s%s' %(name,i)
        q.put(res)
        print('\033[44m%s 生产了 %s\033[0m' %(os.getpid(),res))



if __name__ == '__main__':
    q=Queue()
    #生产者们:即厨师们
    p1=Process(target=producer,args=('包子',q))
    p2=Process(target=producer,args=('骨头',q))
```

```
    p3=Process(target=producer,args=('泔水',q))

    #消费者们:即吃货们
    c1=Process(target=consumer,args=(q,))
    c2=Process(target=consumer,args=(q,))

    #开始
    p1.start()
    p2.start()
    p3.start()
    c1.start()

    p1.join() #必须保证生产者全部生产完毕,才应该发送结束信号
    p2.join()
    p3.join()
    q.put(None) #有几个消费者就应该发送几次结束信号None
    q.put(None) #发送结束信号
    print('主')
```

其实我们的思路无非是发送结束信号而已，有另外一种队列提供了这种机制

JoinableQueue([maxsize])

```
#JoinableQueue([maxsize])：这就像是一个Queue对象，但队列允许项目的使用者通知生成者项目已经被成功处理。通知进程是使用

  #参数介绍：
   maxsize是队列中允许最大项数，省略则无大小限制。
   #方法介绍：
   JoinableQueue的实例p除了与Queue对象相同的方法之外还具有：
   q.task_done()：使用者使用此方法发出信号，表示q.get()的返回项目已经被处理。如果调用此方法的次数大于从队列中删除项目的
   q.join():生产者调用此方法进行阻塞，直到队列中所有的项目均被处理。阻塞将持续到队列中的每个项目均调用q.task_done（）方
```


```
from multiprocessing import Process,JoinableQueue
import time,random,os
def consumer(q):
    while True:
        res=q.get()
        # time.sleep(random.randint(1,3))
        time.sleep(random.random())
        print('\033[45m%s 吃 %s\033[0m' %(os.getpid(),res))
        q.task_done() #向q.join()发送一次信号,证明一个数据已经被取走并执行完了

def producer(name,q):
    for i in range(10):
        # time.sleep(random.randint(1,3))
        time.sleep(random.random())
        res='%s%s' %(name,i)
        q.put(res)
        print('\033[44m%s 生产了 %s\033[0m' %(os.getpid(),res))
    print('%s生产结束'%name)
    q.join() #生产完毕，使用此方法进行阻塞，直到队列中所有项目均被处理。
    print('%s生产结束~~~~~~'%name)
```

```
if __name__ == '__main__':
    q=JoinableQueue()
    #生产者们:即厨师们
    p1=Process(target=producer,args=('包子',q))
    p2=Process(target=producer,args=('骨头',q))
    p3=Process(target=producer,args=('泔水',q))

    #消费者们:即吃货们
    c1=Process(target=consumer,args=(q,))
    c2=Process(target=consumer,args=(q,))
    c1.daemon=True #如果不加守护，那么主进程结束不了，但是加了守护之后，必须确保生产者的内容生产完并且被处理完了，所有
    c2.daemon=True

    #开始
    p_l=[p1,p2,p3,c1,c2]
    for p in p_l:
        p.start()

    p1.join() #我要确保你的生产者进程结束了，生产者进程的结束标志着你生产的所有的人任务都已经被处理完了
    p2.join()
    p3.join()
    print('主')

    # 主进程等--->p1,p2,p3等---->c1,c2
    # p1,p2,p3结束了,证明c1,c2肯定全都收完了p1,p2,p3发到队列的数据
    # 因而c1,c2也没有存在的价值了,不需要继续阻塞在进程中影响主进程了。应该随着主进程的结束而结束,所以设置成守护进程就可以
```

6.管道（了解）

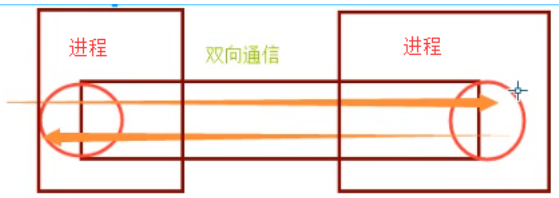
进程间通信（IPC）方式二：管道（不推荐使用，了解即可），会导致数据不安全的情况出现，后面我们会说到为什么会带来数据 不安全的问题。

```
#创建管道的类:
Pipe([duplex]):在进程之间创建一条管道，并返回元组（conn1,conn2）,其中conn1，conn2表示管道两端的连接对象，强调一点：必
#参数介绍:
dumplex:默认管道是全双工的，如果将duplex射成False，conn1只能用于接收，conn2只能用于发送。
#主要方法:
    conn1.recv():接收conn2.send(obj)发送的对象。如果没有消息可接收，recv方法会一直阻塞。如果连接的另外一端已经关闭，那么
    conn1.send(obj):通过连接发送对象。obj是与序列化兼容的任意对象
 #其他方法:
conn1.close():关闭连接。如果conn1被垃圾回收，将自动调用此方法
conn1.fileno():返回连接使用的整数文件描述符
conn1.poll([timeout]):如果连接上的数据可用，返回True。timeout指定等待的最长时限。如果省略此参数，方法将立即返回结果。如

conn1.recv_bytes([maxlength]):接收c.send_bytes()方法发送的一条完整的字节消息。maxlength指定要接收的最大字节数。如果进
conn.send_bytes(buffer [, offset [, size]]):通过连接发送字节数据缓冲区，buffer是支持缓冲区接口的任意对象，offset是缓冲区中

conn1.recv_bytes_into(buffer [, offset]):接收一条完整的字节消息，并把它保存在buffer对象中，该对象支持可写入的缓冲区接口（
```

```
from multiprocessing import Process, Pipe
```

```
def f(conn):
    conn.send("Hello 妹妹") #子进程发送了消息
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe() #建立管道，拿到管道的两端，双工通信方式，两端都可以收发消息
    p = Process(target=f, args=(child_conn,)) #将管道的一段给子进程
    p.start() #开启子进程
    print(parent_conn.recv()) #主进程接受了消息
    p.join()
```

```
conn1, conn2 = Pipe()
conn1.send('123456')
print(conn2.recv())
```



应该特别注意管道端点的正确管理问题。**如果是生产者或消费者中都没有使用管道的某个端点，就应将它关闭。**这也说明了为何在**生产者中关闭了管道的输出端，在消费者中关闭管道的输入端。**如果忘记执行这些步骤，程序可能在消费者中的recv（）操作上挂起（就是阻塞）。管道是由操作系统进行引用计数的，必须在所有进程中关闭管道的相同一端就会能生成EOFError异常。因此，在生产者中关闭管道不会有任何效果，除非消费者也关闭了相同的管道端点。

```
from multiprocessing import Process, Pipe

def f(parent_conn,child_conn):
    #parent_conn.close() #不写close将不会引发EOFError
    while True:
        try:
            print(child_conn.recv())
        except EOFError:
            child_conn.close()
            break

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(parent_conn,child_conn,))
    p.start()
    child_conn.close()
    parent_conn.send('hello')
    parent_conn.close()
    p.join()
```

主进程将管道的两端都传送给子进程，子进程和主进程共用管道的两种报错情况，都是在recv接收的时候报错的：

1.主进程和子进程中的管道的相同一端都关闭了，出现EOFError；

2.如果你管道的一端在主进程和子进程中都关闭了，但是你还用这个关闭的一端去接收消息，那么就会出现OSError；

所以你关闭管道的时候，就容易出现问题，需要将所有只用这个管道的进程中的两端全部关闭才行。当然也可以通过异常捕获（try：except EOFerror）来处理。

虽然我们在主进程和子进程中都打印了一下conn1一端的对象，发现两个不再同一个地址，但是子进程中的管道和主进程中的管道还是可以通信的，因为管道是同一套，系统能够记录。

我们的目的就是关闭所有的管道，那么主进程和子进程进行通信的时候，可以给子进程传管道的一端就够了，并且用我们之前学到的，信息发送完之后，再发送一个结束信号None，那么你收到的消息为None的时候直接结束接收或者说结束循环，就不用每次都关闭各个进程中的管道了。

```
from multiprocessing import Pipe,Process

def func(conn):
    while True:
        msg = conn.recv()
        if msg is None:break
        print(msg)

if __name__ == '__main__':
    conn1,conn2 = Pipe()
    p = Process(target=func,args=(conn1,))
    p.start()
    for i in range(10):
        conn2.send('约吧')
    conn2.send(None)
```

```
from multiprocessing import Process,Pipe

def consumer(p,name):
    produce, consume=p
    produce.close()
    while True:
        try:
            baozi=consume.recv()
            print('%s 收到包子:%s' %(name,baozi))
        except EOFError:
            break

def producer(seq,p):
    produce, consume=p
    consume.close()
    for i in seq:
        produce.send(i)

if __name__ == '__main__':
    produce,consume=Pipe()

    c1=Process(target=consumer,args=((produce,consume),'c1'))
    c1.start()


    seq=(i for i in range(10))
    producer(seq,(produce,consume))

    produce.close()
```

```
    consume.close()

    c1.join()
    print('主进程')
```

```
关于管道会造成数据不安全问题的官方解释：
    The two connection objects returned by Pipe() represent the two ends of the pipe. Each connection object has send() ar
        由Pipe方法返回的两个连接对象表示管道的两端。每个连接对象都有send和recv方法（除其他之外）。注意，如果两个进程（或
```

多个消费者竞争会出现数据不安全的问题的解决方案：加锁

管道可以用于双工通信，通常利用在客户端/服务端中使用的请求/响应模型，或者远程过程调用，就可以使用管道编写与进程交互的程序，像前面将网络通信的时候，我们使用了一个叫subprocess的模块，里面有个参数是pipe管道，执行系统指令，并通过管道获取结果。

7.数据共享（了解）

展望未来，基于消息传递的并发编程是大势所趋

即便是使用线程，推荐做法也是将程序设计为大量独立的线程集合

通过消息队列交换数据。这样极大地减少了对使用锁定和其他同步手段的需求，还可以扩展到分布式系统中

进程间应该尽量避免通信，即便需要通信，也应该选择进程安全的工具来避免加锁带来的问题，应该尽量避免使用本节所讲的共享数据的方式，以后我们会尝试使用数据库来解决进程之间的数据共享问题。

进程之间数据共享的模块之一Manager模块：


```
进程间数据是独立的，可以借助于队列或管道实现通信，二者都是基于消息传递的
虽然进程间数据独立，但可以通过Manager实现数据共享，事实上Manager的功能远不止于此

A manager object returned by Manager() controls a server process which holds Python objects and allows other processes

A manager returned by Manager() will support types list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore,
```

多进程共同去处理共享数据的时候，就和我们多进程同时去操作一个文件中的数据是一样的，不加锁就会出现错误的结果，进程不安全的，所以也需要加锁

```
from multiprocessing import Manager,Process,Lock
def work(d,lock):
    with lock: #不加锁而操作共享的数据,肯定会出现数据错乱
        d['count']-=1

if __name__ == '__main__':
    lock=Lock()
    with Manager() as m:
        dic=m.dict({'count':100})
```

```
    p_l=[]
    for i in range(100):
        p=Process(target=work,args=(dic,lock))
        p_l.append(p)
        p.start()
    for p in p_l:
        p.join()
    print(dic)
```

总结一下，进程之间的通信：队列、管道、数据共享也算

下面要讲的信号量和事件也相当于锁，也是全局的，所有进程都能拿到这些锁的状态，进程之间这些锁啊信号量啊事件啊等等的通信，其实底层还是socekt，只不过是基于文件的socket通信，而不是跟上面的数据共享啊空间共享啊之类的机制，我们之前学的是基于网络的socket通信，还记得socket的两个家族吗，一个文件的一个网络的，所以将来如果说这些锁之类的报错，可能你看到的就是类似于socket的错误，简单知道一下就可以啦~~~

工作中常用的是锁，信号量和事件不常用，但是信号量和事件面试的时候会问到，你能知道就行啦~~~

8.信号量（了解）


```
互斥锁同时只允许一个线程更改数据，而信号量Semaphore是同时允许一定数量的线程更改数据 。
假设商场里有4个迷你唱吧，所以同时可以进去4个人，如果来了第五个人就要在外面等待，等到有人出来才能再进去玩。
实现：
信号量同步基于内部计数器，每调用一次acquire()，计数器减1；每调用一次release()，计数器加1.当计数器为0时，acquire()调用被阻
信号量与进程池的概念很像，但是要区分开，信号量涉及到加锁的概念
```

比如大保健：提前设定好，一个房间只有4个床（计数器现在为4），那么同时只能四个人进来，谁先来的谁先占一个床（acquire，计数器减1），4个床满了之后（计数器为0了），第五个人就要等着，等其中一个人出来（release，计数器加1），他就去占用那个床了。

```
from multiprocessing import Process,Semaphore
import time,random

def go_ktv(sem,user):
    sem.acquire()
    print('%s 占到一间ktv小屋' %user)
    time.sleep(random.randint(0,3)) #模拟每个人在ktv中待的时间不同
    sem.release()

if __name__ == '__main__':
    sem=Semaphore(4)
    p_l=[]
    for i in range(13):
        p=Process(target=go_ktv,args=(sem,'user%s' %i,))
        p.start()
        p_l.append(p)

    for i in p_l:
        i.join()
    print('============》')
```

9.事件（了解）

python线程的事件用于主线程控制其他线程的执行，事件主要提供了三个方法 set、wait、clear。

事件处理的机制：全局定义了一个“Flag”，如果“Flag”值为 False，那么当程序执行 event.wait 方法时就会阻塞，如果“Flag”值为Tr

clear：将“Flag”设置为False
set：将“Flag”设置为True

```
from multiprocessing import Process,Semaphore,Event
import time,random

e = Event() #创建一个事件对象
print(e.is_set())  #is_set()查看一个事件的状态，默认为False，可通过set方法改为True
print('look here！')
# e.set()          #将is_set()的状态改为True。
# print(e.is_set())#is_set()查看一个事件的状态，默认为False，可通过set方法改为Tr
# e.clear()        #将is_set()的状态改为False
# print(e.is_set())#is_set()查看一个事件的状态，默认为False，可通过set方法改为Tr
e.wait()           #根据is_set()的状态结果来决定是否在这阻塞住，is_set()=False那么就阻塞，is_set()=True就不阻塞
print('give me！！')

#set和clear  修改事件的状态 set-->True   clear-->False
#is_set     用来查看一个事件的状态
#wait       依据事件的状态来决定是否阻塞 False-->阻塞  True-->不阻塞
```

```
from multiprocessing import Process, Event
import time, random

def car(e, n):
    while True:
        if not e.is_set():  # 进程刚开启，is_set()的值是Flase，模拟信号灯为红色
            print('\033[31m红灯亮\033[0m，car%s等着' % n)
            e.wait()    # 阻塞，等待is_set()的值变成True，模拟信号灯为绿色
            print('\033[32m车%s 看见绿灯亮了\033[0m' % n)
            time.sleep(random.randint(2,4))
            if not e.is_set():   #如果is_set()的值是Flase，也就是红灯，仍然回到while语句开始
                continue
            print('车开远了,car', n)
            break

# def police_car(e, n):
#     while True:
#         if not e.is_set():# 进程刚开启，is_set()的值是Flase，模拟信号灯为红色
#             print('\033[31m红灯亮\033[0m，car%s等着' % n)
#             e.wait(0.1) # 阻塞，等待设置等待时间，等待0.1s之后没有等到绿灯就闯红灯走了
#             if not e.is_set():
#                 print('\033[33m红灯,警车先走\033[0m，car %s' % n)
#             else:
#                 print('\033[33;46m绿灯，警车走\033[0m，car %s' % n)
#         break
```

```
def traffic_lights(e, inverval):
    while True:
        time.sleep(inverval)
        if e.is_set():
            print('######', e.is_set())
            e.clear()  # ---->将is_set()的值设置为False
        else:
            e.set()    # ---->将is_set()的值设置为True
            print('**********',e.is_set())


if __name__ == '__main__':
    e = Event()
    for i in range(10):
        p=Process(target=car,args=(e,i,))  # 创建10个进程控制10辆车
        time.sleep(random.random(1, 3))    #车不是一下子全过来
        p.start()

    # for i in range(5):
    #     p = Process(target=police_car, args=(e, i,))  # 创建5个进程控制5辆警车
    #     p.start()

    #信号灯必须是单独的进程，因为它不管你车开到哪了，我就按照我红绿灯的规律来闪烁变换，对吧
    t = Process(target=traffic_lights, args=(e, 5))  # 创建一个进程控制红绿灯
    t.start()

    print('预备~~~~开始！！！')
```

八 进程池和mutiprocess.Poll

为什么要有进程池?进程池的概念。

在程序实际处理问题过程中，忙时会有成千上万的任务需要被执行，闲时可能只有零星任务。那么在成千上万个任务需要被执行的时候，我们就需要去创建成千上万个进程么？首先，创建进程需要消耗时间，销毁进程（空间，变量，文件信息等等的内容）也需要消耗时间。第二即便开启了成千上万的进程，操作系统也不能让他们同时执行，维护一个很大的进程列表的同时，调度的时候，还需要进行切换并且记录每个进程的执行节点，也就是记录上下文（各种变量等等乱七八糟的东西，虽然你看不到，但是操作系统都要做），这样反而会影响程序的效率。因此我们不能无限制的根据任务开启或者结束进程。就看我们上面的一些代码例子，你会发现有些程序是不是执行的时候比较慢才出结果，就是这个原因，那么我们要怎么做呢?

在这里，要给大家介绍一个进程池的概念，定义一个池子，在里面放上固定数量的进程，有需求来了，就拿一个池中的进程来处理任务，等到处理完毕，进程并不关闭，而是将进程再放回进程池中继续等待任务。如果有很多任务需要执行，池中的进程数量不够，任务就要等待之前的进程执行任务完毕归来，拿到空闲进程才能继续执行。也就是说，池中进程的数量是固定的，那么同一时间最多有固定数量的进程在运行。这样不会增加操作系统的调度难度，还节省了开闭进程的时间，也一定程度上能够实现并发效果

multiprocess.Poll模块

创建进程池的类：如果指定numprocess为3，则进程池会从无到有创建三个进程，然后自始至终使用这三个进程去执行所有任务（高级一些的进程池可以根据你的并发量，搞成动态增加或减少进程池中的进程数量的操作），不会开启其他进程，提高操作系统效率，减少空间的占用等。

概念介绍:

```
Pool([numprocess  [,initializer [, initargs]]]):创建进程池
```

numprocess:要创建的进程数，如果省略，将默认使用cpu_count()的值
initializer：是每个工作进程启动时要执行的可调用对象，默认为None
initargs：是要传给initializer的参数组

p.apply(func [, args [, kwargs]]):在一个池工作进程中执行func(*args,**kwargs),然后返回结果。
'''需要强调的是：此操作并不会在所有池工作进程中并执行func函数。如果要通过不同参数并发地执行func函数，必须从不同线程调用

p.apply_async(func [, args [, kwargs]]):在一个池工作进程中执行func(*args,**kwargs),然后返回结果。
'''此方法的结果是AsyncResult类的实例，callback是可调用对象，接收输入参数。当func的结果变为可用时，将理解传递给callback。

p.close():关闭进程池，防止进一步操作。如果所有操作持续挂起，它们将在工作进程终止前完成

P.jion():等待所有工作进程退出。此方法只能在close（）或teminate()之后调用

方法apply_async()和map_async（）的返回值是AsyncResul的实例obj。实例具有以下方法
obj.get():返回结果，如果有必要则等待结果到达。timeout是可选的。如果在指定时间内还没有到达，将引发一场。如果远程操作中引
obj.ready():如果调用完成，返回True
obj.successful():如果调用完成且没有引发异常，返回True，如果在结果就绪之前调用此方法，引发异常
obj.wait([timeout]):等待结果变为可用。
obj.terminate()：立即终止所有工作进程，同时不执行任何清理或结束任何挂起工作。如果p被垃圾回收，将自动调用此函数

```
import time
from multiprocessing import Pool,Process

#针对range(100)这种参数的
# def func(n):
#     for i in range(3):
#         print(n + 1)

def func(n):
    print(n)
    # 结果:
    #     (1, 2)
    #     alex
def func2(n):
    for i in range(3):
        print(n - 1)
if __name__ == '__main__':
    #1.进程池的模式
    s1 = time.time()  #我们计算一下开多进程和进程池的执行效率
    poll = Pool(5) #创建含有5个进程的进程池
    # poll.map(func,range(100)) #异步调用进程，开启100个任务,map自带join的功能
    poll.map(func,[(1,2),'alex']) #异步调用进程，开启100个任务,map自带join的功能
    # poll.map(func2,range(100))  #如果想让进程池完成不同的任务，可以直接这样搞
    #map只限于接收一个可迭代的数据类型参数，列表啊，元祖啊等等，如果想做其他的参数之类的操作，需要用后面我们要学的方法
    # t1 = time.time() - s1
```

```
    #
    # #2.多进程的模式
    # s2 = time.time()
    # p_list = []
    # for i in range(100):
    #     p = Process(target=func,args=(i,))
    #     p_list.append(p)
    #     p.start()
    # [pp.join() for pp in p_list]
    # t2 = time.time() - s2
    #
    # print('t1>>',t1) #结果：0.5146853923797607s 进程池的效率高
    # print('t2>>',t2) #结果：12.092015027999878s
```

有一点，map是异步执行的，并且自带close和join

一般约定俗成的是进程池中的进程数量为CPU的数量，工作中要看具体情况来考量。

实际应用代码示例：

同步与异步两种执行方式：

```
import os,time
from multiprocessing import Pool

def work(n):
    print('%s run' %os.getpid())
    time.sleep(1)
    return n**2

if __name__ == '__main__':
    p=Pool(3) #进程池中从无到有创建三个进程,以后一直是这三个进程在执行任务
    res_l=[]
    for i in range(10):
        res=p.apply(work,args=(i,)) # 同步调用，直到本次任务执行完毕拿到res，等待任务work执行的过程中可能有阻塞也可能没有
                                    # 但不管该任务是否存在阻塞，同步调用都会在原地等着
        res_l.append(res)
    print(res_l)
```


```
import os
import time
import random
from multiprocessing import Pool

def work(n):
    print('%s run' %os.getpid())
    time.sleep(random.random())
    return n**2
```

```
if __name__ == '__main__':
    p=Pool(3) #进程池中从无到有创建三个进程,以后一直是这三个进程在执行任务
    res_l=[]
    for i in range(10):
        res=p.apply_async(work,args=(i,)) # 异步运行，根据进程池中有的进程数，每次最多3个子进程在异步执行，并且可以执行不
                                          # 返回结果之后，将结果放入列表，归还进程，之后再执行新的任务
                                          # 需要注意的是，进程池中的三个进程不会同时开启或者同时结束
                                          # 而是执行完一个就释放一个进程，这个进程就去接收新的任务。
        res_l.append(res)

    # 异步apply_async用法：如果使用异步提交的任务，主进程需要使用join，等待进程池内任务都处理完，然后可以用get收集结果
    # 否则，主进程结束，进程池可能还没来得及执行，也就跟着一起结束了
    p.close() #不是关闭进程池，而是结束进程池接收任务，确保没有新任务再提交过来。
    p.join()  #感知进程池中的任务已经执行结束，只有当没有新的任务添加进来的时候，才能感知到任务结束了，所以在join之前必须
    for res in res_l:
        print(res.get()) #使用get来获取apply_aync的结果,如果是apply,则没有get方法,因为apply是同步执行,立刻获取结果,也根本无需
```

```
#一：使用进程池（异步调用,apply_async）
#coding: utf-8
from multiprocessing import Process,Pool
import time

def func(msg):
    print( "msg:", msg)
    time.sleep(1)
    return msg

if __name__ == "__main__":
    pool = Pool(processes = 3)
    res_l=[]
    for i in range(10):
        msg = "hello %d" %(i)
        res=pool.apply_async(func, (msg, ))  #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去
        res_l.append(res)
        # s = res.get() #如果直接用res这个结果对象调用get方法获取结果的话，这个程序就变成了同步，因为get方法直接就在这里等
    print("==============================>") #没有后面的join，或get，则程序整体结束，进程池中的任务还没

    pool.close() #关闭进程池，防止进一步操作。如果所有操作持续挂起，它们将在工作进程终止前完成
    pool.join()  #调用join之前，先调用close函数，否则会出错。执行完close后不会有新的进程加入到pool,join函数等待所有子进程

    print(res_l) #看到的是<multiprocessing.pool.ApplyResult object at 0x10357c4e0>对象组成的列表,而非最终的结果,但这一步
    for i in res_l:
        print(i.get()) #使用get来获取apply_aync的结果,如果是apply,则没有get方法,因为apply是同步执行,立刻获取结果,也根本无需g

#二：使用进程池（同步调用,apply）
#coding: utf-8
from multiprocessing import Process,Pool
import time

def func(msg):
    print( "msg:", msg)
    time.sleep(0.1)
    return msg

if __name__ == "__main__":
    pool = Pool(processes = 3)
```

```
    res_l=[]
    for i in range(10):
        msg = "hello %d" %(i)
        res=pool.apply(func, (msg, ))   #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去
        res_l.append(res) #同步执行，即执行完一个拿到结果，再去执行另外一个
    print("==============================>")
    pool.close()
    pool.join()   #调用join之前，先调用close函数，否则会出错。执行完close后不会有新的进程加入到pool,join函数等待所有子进程结
    
    print(res_l) #看到的就是最终的结果组成的列表
    for i in res_l: #apply是同步的，所以直接得到结果，没有get()方法
        print(i)
```

进程池版的socket**并发聊天代码示例：**

```
#Pool内的进程数默认是cpu核数，假设为4（查看方法os.cpu_count()）
#开启6个客户端，会发现2个客户端处于等待状态
#在每个进程内查看pid，会发现pid使用为4个，即多个客户端公用4个进程
from socket import *
from multiprocessing import Pool
import os

server=socket(AF_INET,SOCK_STREAM)
server.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
server.bind(('127.0.0.1',8080))
server.listen(5)

def talk(conn):
    print('进程pid: %s' %os.getpid())
    while True:
        try:
            msg=conn.recv(1024)
            if not msg:break
            conn.send(msg.upper())
        except Exception:
            break

if __name__ == '__main__':
    p=Pool(4)
    while True:
        conn,*_=server.accept()
        p.apply_async(talk,args=(conn,))
        # p.apply(talk,args=(conn,client_addr)) #同步的话，则同一时间只有一个客户端能访问
```

```
from socket import *

client=socket(AF_INET,SOCK_STREAM)
client.connect(('127.0.0.1',8080))
```

```
while True:
    msg=input('>>: ').strip()
    if not msg:continue

    client.send(msg.encode('utf-8'))
    msg=client.recv(1024)
    print(msg.decode('utf-8'))
```

发现：并发开启多个客户端，服务端同一时间只有4个不同的pid，只能结束一个客户端，另外一个客户端才会进来.

同时最多和4个人进行聊天，因为进程池中只有4个进程可供调用，那有同学会问，我们这么多人想同时聊天怎么办，又不让用多进程，进程池也不能开太多的进程，那咋整啊，后面我们会学到多线程，到时候大家就知道了，现在你们先这样记住就好啦

然后我们再提一个回调函数

需要回调函数的场景：进程池中任何一个任务一旦处理完了，就立即告知主进程：我好了额，你可以处理我的结果了。主进程则调用一

我们可以把耗时间（阻塞）的任务放到进程池中，然后指定回调函数（主进程负责执行），这样主进程在执行回调函数时就省去了I/O的

```
import os
from multiprocessing import Pool

def func1(n):
    print('func1>>',os.getpid())
    print('func1')
    return n*n

def func2(nn):
    print('func2>>',os.getpid())
    print('func2')
    print(nn)
    # import time
    # time.sleep(0.5)
if __name__ == '__main__':
    print('主进程：',os.getpid())
    p = Pool(5)
    #args里面的10给了func1，func1的返回值作为回调函数的参数给了callback对应的函数，不能直接给回调函数直接传参数，他只能
    # for i in range(10,20): #如果是多个进程来执行任务，那么当所有子进程将结果给了回调函数之后，回调函数又是在主进程上执行
    #     p.apply_async(func1,args=(i,),callback=func2)
    p.apply_async(func1,args=(10,),callback=func2)

    p.close()
    p.join()

#结果
# 主进程：  11852  #发现回调函数是在主进程中完成的，其实如果是在子进程中完成的，那我们直接将代码写在子进程的任务函数fun
# func1>> 17332
# func1
# func2>> 11852
# func2
```

```
# 100
```

回调函数在写的时候注意一点，回调函数的形参执行有一个，如果你的执行函数有多个返回值，那么也可以被回调函数的这一个形参接收，接收的是一个元祖，包含着你执行函数的所有返回值。

使用进程池来搞爬虫的时候，最耗时间的是请求地址的网络请求延迟，那么如果我们在将处理数据的操作加到每个子进程中，那么所有在进程池后面排队的进程就需要等更长的时间才能获取进程池里面的执行进程来执行自己，所以一般我们就将请求作成一个执行函数，通过进程池去异步执行，剩下的数据处理的内容放到另外一个进程或者主进程中去执行，将网络延迟的时间也利用起来，效率更高。

requests这个模块的get方法请求页面，就和我们在浏览器上输入一个网址然后回车去请求别人的网站的效果是一样的。安装requests模块的指令：在cmd窗口执行pip install requests。


```
import requests
response = requests.get('http://www.baidu.com')
print(response)
print(response.status_code) #200正常，404找不到网页，503等5开头的是人家网站内部错误
print(response.content.decode('utf-8'))
```


```
from multiprocessing import Pool
import requests
import json
import os

def get_page(url):
    print('<进程%s> get %s' %(os.getpid(),url))
    respone=requests.get(url)
    if respone.status_code == 200:
        return {'url':url,'text':respone.text}

def pasrse_page(res):
    print('<进程%s> parse %s' %(os.getpid(),res['url']))
    parse_res='url:<%s> size:[%s]\n' %(res['url'],len(res['text']))
    with open('db.txt','a') as f:
        f.write(parse_res)


if __name__ == '__main__':
    urls=[
        'https://www.baidu.com',
        'https://www.python.org',
        'https://www.openstack.org',
        'https://help.github.com/',
        'http://www.sina.com.cn/'
    ]

    p=Pool(3)
    res_l=[]
    for url in urls:
        res=p.apply_async(get_page,args=(url,),callback=pasrse_page)
        res_l.append(res)
```

```
    p.close()
    p.join()
    print([res.get() for res in res_l]) #拿到的是get_page的结果,其实完全没必要拿该结果,该结果已经传给回调函数处理了

'''
打印结果:
<进程3388> get https://www.baidu.com
<进程3389> get https://www.python.org
<进程3390> get https://www.openstack.org
<进程3388> get https://help.github.com/
<进程3387> parse https://www.baidu.com
<进程3389> get http://www.sina.com.cn/
<进程3387> parse https://www.python.org
<进程3387> parse https://help.github.com/
<进程3387> parse http://www.sina.com.cn/
<进程3387> parse https://www.openstack.org
[{'url': 'https://www.baidu.com', 'text': '<!DOCTYPE html>\r\n...',...}]
'''
```

```
from multiprocessing import Pool
import time,random
import requests
import re

def get_page(url,pattern):
    response=requests.get(url)
    if response.status_code == 200:
        return (response.text,pattern)

def parse_page(info):
    page_content,pattern=info
    res=re.findall(pattern,page_content)
    for item in res:
        dic={
            'index':item[0],
            'title':item[1],
            'actor':item[2].strip()[3:],
            'time':item[3][5:],
            'score':item[4]+item[5]

        }
        print(dic)
if __name__ == '__main__':
    pattern1=re.compile(r'<dd>.*?board-index.*?>(\d+)<.*?title="(.*?)".*?star.*?>(.*?)<.*?releasetime.*?>(.*?)<.*?integer.*

    url_dic={
        'http://maoyan.com/board/7':pattern1,
    }

    p=Pool()
    res_l=[]
    for url,pattern in url_dic.items():
        res=p.apply_async(get_page,args=(url,pattern),callback=parse_page)
        res_l.append(res)
```

```
    for i in res_l:
        i.get()

    # res=requests.get('http://maoyan.com/board/7')
    # print(re.findall(pattern,res.text))
```

如果在主进程中等待进程池中所有任务都执行完毕后，再统一处理结果，则无需回调函数

```
from multiprocessing import Pool
import time,random,os

def work(n):
    time.sleep(1)
    return n**2
if __name__ == '__main__':
    p=Pool()

    res_l=[]
    for i in range(10):
        res=p.apply_async(work,args=(i,))
        res_l.append(res)

    p.close()
    p.join() #等待进程池中所有进程执行完毕

    nums=[]
    for res in res_l:
        nums.append(res.get()) #拿到所有结果
    print(nums) #主进程拿到所有的处理结果,可以在主进程中进行统一进行处理
```

进程池和信号量的区别：

进程池是多个需要被执行的任务在进程池外面排队等待获取进程对象去执行自己，而信号量是一堆进程等待着去执行一段逻辑代码。

信号量不能控制创建多少个进程，但是可以控制同时多少个进程能够执行，但是进程池能控制你可以创建多少个进程。

举例：就像那些开大车拉煤的，信号量是什么呢，就好比我只有五个车道，你每次只能过5辆车，但是不影响你创建100辆车，但是进程池相当于什么呢？相当于你只有5辆车，每次5个车拉东西，拉完你再把车放回来，给别的人拉煤用。

其他语言里面有更高级的进程池，在设置的时候，可以将进程池中的进程动态的创建出来，当需求增大的时候，就会自动在进程池中添加进程，需求小的时候，自动减少进程，并且可以设置进程数量的上线，最多为多，python里面没有。

进程池的其他实现方式： https://docs.python.org/dev/library/concurrent.futures.html