

Eva\_J

程序媛

# python之路——进程

昵称: Eva\_J  
园龄: 4年9个月  
粉丝: 4193  
关注: 7  
[+加关注](#)

## 阅读目录

- 理论知识
  - 操作系统背景知识
  - 什么是进程
  - 进程调度
  - 进程的并发与并行
  - 同步\异步\阻塞\非阻塞
  - 进程的创建与结束
- 在python程序中的进程操作
  - multiprocess模块
  - 进程的创建和multiprocess.Process
  - 进程同步控制 —— 锁
  - 进程间通信 —— 队列
  - 进程间的数据共享 —— multiprocessing.Manager
  - 进程池和multiprocess.Pool

< 2020年8月 >						
日	一	二	三	四	五	六
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

## 搜索

找找看

谷歌搜索

## 常用链接

我的随笔  
我的评论  
我的参与  
最新评论  
我的标签  
[更多链接](#)

## 我的标签

[go\(1\)](#)  
[python\(1\)](#)  
[回到顶部](#)

## 随笔分类

[python\\_Django\(4\)](#)  
[python基础语法\(7\)](#)  
[python面向对象\(4\)](#)  
[python网络编程\(1\)](#)  
[python线程进程与协程\(6\)](#)  
[回到顶部](#)

## 随笔档案

[2020年3月\(1\)](#)  
[2019年3月\(1\)](#)  
[2018年7月\(1\)](#)  
[2018年1月\(1\)](#)  
[2017年8月\(1\)](#)  
[2017年4月\(1\)](#)  
[2017年3月\(1\)](#)

## 理论知识

### 操作系统背景知识

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

[回到顶部](#)

## 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。

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。

☐

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）、数据区域（data region）  
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的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律引

☐

从理论角度看，是对正在运行的程序过程的抽象；  
从实现角度看，是一种数据结构，目的在于清晰地刻画动态系统的内在规律，有效管理和调度进入计算机系统主存储器运行的程序。

☐

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，程序不能发生

☐

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。

**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。**

[回到顶部](#)

## 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（进

☐

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。但其对长

☐

2016年6月(2)  
2016年4月(1)  
2016年3月(1)  
2016年2月(1)  
2016年1月(10)  
2015年12月(6)  
2015年11月(6)

## 文章分类

flask(1)  
go(3)  
mysql(7)  
python宣讲专用课件(3)  
python之路(16)  
数据库相关(7)  
周末班(3)

## 友链

银角大王  
学霸yuan先生  
冷先生

## 最新评论

1. Re:多表查询  
查询每个部门最新入职的那位员工: select post,emp\_name from empl  
oyee where hire\_date in (select  
max(hire\_date) from e...  
--仰望夜空
2. Re:mysql索引原理  
写的很细，很好，赞  
  
--15927797249
3. Re:python——有一种线程池叫  
做自己写的线程池  
武sir的版本有完整的代码和例子吗？  
或者原链接.....  
  
--littlesnaka
4. Re:git操作备忘  
sf~  
  
--Chris2357
5. Re:python之路——博客目录  
停更了1年多了呢  
  
--Chris2357

## 阅读排行榜

1. python之路——博客目录(145038)
2. python——赋值与深浅拷贝(33035)
3. python——SQL基本使用(30716)

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, 则调度程序在轮转法中, 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队列。

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, 而且如果并未指明进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较

(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低。

(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完

(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

进程的并行与并发

**并行**：并行是指两者同时执行，比如赛跑，两个人都在不停的往前跑；（资源够用，比如三个线程，四核的CPU）

**并发**：并发是指资源有限的情况下，两者交替轮流使用资源，比如一段路(单核CPU资源)同时只能过一个人，A走一段后，让给B，B用完继续给A，交替使用，目的是提高效率。

**区别**：

**并行**是从微观上，也就是在一个精确的时间片刻，有不同的程序在执行，这就要求必须多个处理器。

**并发**是从宏观上，在一个时间段上可以看出是同时执行的，比如一个服务器同时处理多个session。

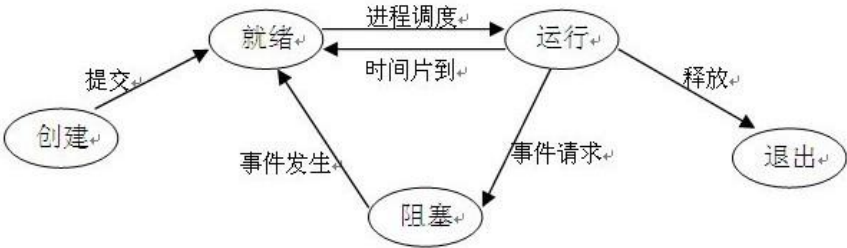
[回到顶部](#)

同步异步阻塞非阻塞

[回到顶部](#)

状态介绍

进程三态状态装换图



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。

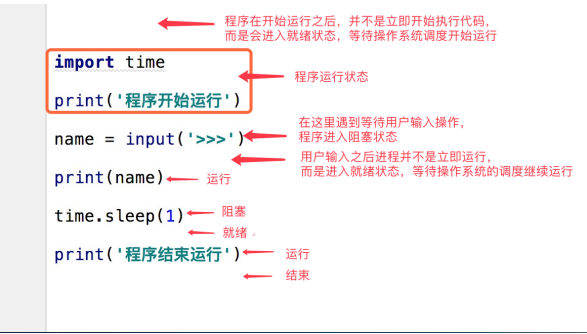
- 4. python——django使用mysql数据库 (一) (23337)
- 5. 前端开发的正确姿势——各种文件的目录结构规划及引用(21500)

评论排行榜

- 1. python之路——博客目录(32)
- 2. python——赋值与深浅拷贝(16)
- 3. python——进程基础(9)
- 4. python的类和对象——进阶篇(8)
- 5. python\_控制台输出带颜色的文字方法(8)

推荐排行榜

- 1. python之路——博客目录(63)
- 2. python——赋值与深浅拷贝(35)
- 3. python3.7导入gevent模块报错的解决方案(6)
- 4. python\_控制台输出带颜色的文字方法(5)
- 5. python——挖装饰器祖坟事件(5)



同步和异步

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，失败都失败，两个任务的状态可以保持一致。

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了。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。

比如我去银行办理业务，可能会有两种方式：

第一种：选择排队等候；

第二种：选择取一个小纸条上面有我的号码，等到排到我这一号时由柜台的人通知我轮到我去办理业务了；

第一种：前者（排队等候）就是同步等待消息通知，也就是我要一直在等待银行办理业务情况；

第二种：后者（等待别人通知）就是异步等待消息通知。在异步消息处理中，等待消息通知者（在这个例子中就是等待办理业务的人）往往注

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，那相反，有的人喜欢在银行办理这些业务的时候一边打电话发发短信一边等待，这样的状态就是非阻塞的，因为他（等待者）没有阻塞在这

注意：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有没有了。如果把打电话和观察排队

同步/异步与阻塞/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
- 如果在银行等待办理业务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离开银行做其它的事情，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有没有了，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，于是他告诉大堂经理说，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。

[回到顶部](#)

## 进程的创建与结束

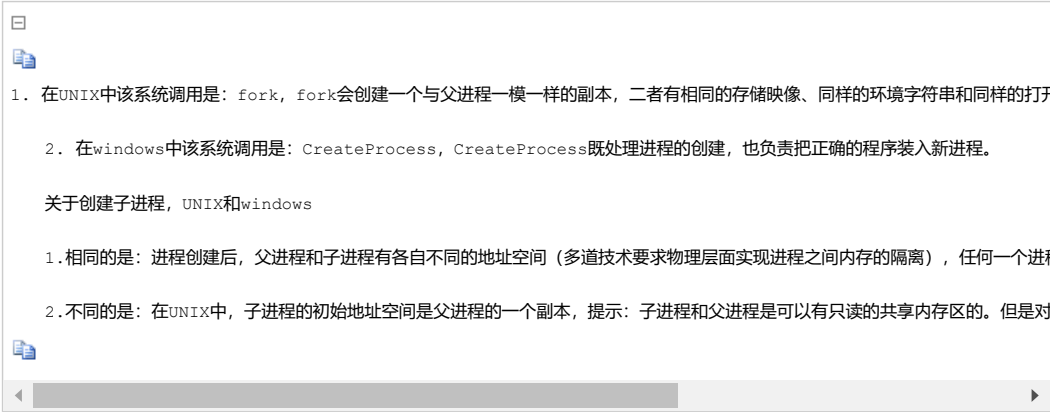
### 进程的创建

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。



### 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）

[回到顶部](#)

## 在python程序中的进程操作

之前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。

[回到顶部](#)

### 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。

[回到顶部](#)

### multiprocess.process模块

## 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

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

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



```
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  
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
```



```
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字节的字符串。这个键的用途是为涉及网络连接的底层进程间
```



在Windows操作系统中由于没有`fork`（linux操作系统中创建进程的机制），在创建子进程的时候会自动 `import` 启动它的这个文件，而

## 使用process模块创建进程

在一个python进程中开启子进程，start方法和并发效果。



```
import time  
from multiprocessing import Process  
  
def f(name):  
    print('hello', name)  
    print('我是子进程')  
  
if __name__ == '__main__':  
    p = Process(target=f, args=('bob',))  
    p.start()  
    time.sleep(1)  
    print('执行主进程的内容了')
```



```
import time  
from multiprocessing import Process  
  
def f(name):  
    print('hello', name)  
    time.sleep(1)  
    print('我是子进程')
```



```

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    #p.join()
    print('我是父进程')

```



```

import os
from multiprocessing import Process

def f(x):
    print('子进程id : ', os.getpid(), '父进程id : ', os.getppid())
    return x*x

if __name__ == '__main__':
    print('主进程id : ', os.getpid())
    p_lst = []
    for i in range(5):
        p = Process(target=f, args=(i,))
        p.start()

```



进阶，多个进程同时运行（注意，子进程的执行顺序不是根据启动顺序决定的）



```

import time
from multiprocessing import Process

def f(name):
    print('hello', name)
    time.sleep(1)

if __name__ == '__main__':
    p_lst = []
    for i in range(5):
        p = Process(target=f, args=('bob',))
        p.start()
        p_lst.append(p)

```



```

import time
from multiprocessing import Process

def f(name):
    print('hello', name)
    time.sleep(1)

if __name__ == '__main__':
    p_lst = []
    for i in range(5):
        p = Process(target=f, args=('bob',))
        p.start()
        p_lst.append(p)
        p.join()
    # [p.join() for p in p_lst]
    print('父进程在执行')

```



```
import time
from multiprocessing import Process

def f(name):
    print('hello', name)
    time.sleep(1)

if __name__ == '__main__':
    p_lst = []
    for i in range(5):
        p = Process(target=f, args=('bob',))
        p.start()
        p_lst.append(p)
    # [p.join() for p in p_lst]
    print('父进程在执行')
```



除了上面这些开启进程的方法，还有一种以继承Process类的形式开启进程的方式

```
import os
from multiprocessing import Process

class MyProcess(Process):
    def __init__(self, name):
        super().__init__()
        self.name = name
    def run(self):
        print(os.getpid())
        print('%s 正在和女主播聊天' % self.name)

p1 = MyProcess('wupeiqi')
p2 = MyProcess('yuanhao')
p3 = MyProcess('nezha')

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

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

print('主线程')
```



进程之间的数据隔离问题

```
from multiprocessing import Process

def work():
    global n
    n = 0
    print('子进程内: ', n)

if __name__ == '__main__':
    n = 100
    p = Process(target=work)
    p.start()
    print('主进程内: ', n)
```



## 守护进程



会随着主进程的结束而结束。

#### 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: daemon 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)

p = Myprocess('哪吒')
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(10) # 在sleep时查看进程id对应的进程ps -ef|grep id
print('主')
```

```
from multiprocessing import Process

def foo():
    print(123)
    time.sleep(1)
    print("end123")

def bar():
    print(456)
    time.sleep(3)
    print("end456")

p1 = Process(target=foo)
p2 = Process(target=bar)

p1.daemon = True
p1.start()
p2.start()
time.sleep(0.1)
print("main-----") #打印该行则主进程代码结束,则守护进程p1应该被终止. #可能会有p1任务执行的打印信息123,因为主进程打印
```

#### socket聊天并发实例

```
from socket import *
from multiprocessing import Process

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, client_addr):
    while True:
```

```

try:
    msg=conn.recv(1024)
    if not msg:break
    conn.send(msg.upper())
except Exception:
    break

if __name__ == '__main__': #windows下start进程一定要写到这下面
    while True:
        conn,client_addr=server.accept()
        p=Process(target=talk,args=(conn,client_addr))
        p.start()

```



```

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'))

```



## 多进程中的其他方法



```

from multiprocessing import Process
import time
import random

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

    def run(self):
        print('%s正在和网红脸聊天' %self.name)
        time.sleep(random.randrange(1,5))
        print('%s还在和网红脸聊天' %self.name)

p1=Myprocess('哪吒')
p1.start()

p1.terminate() #关闭进程,不会立即关闭,所以is_alive立刻查看的结果可能还是存活
print(p1.is_alive()) #结果为True

print('开始')
print(p1.is_alive()) #结果为False

```



```

1 class Myprocess(Process):
2     def __init__(self,person):
3         self.name=person # name属性是Process中的属性,标示进程的名字
4         super().__init__() # 执行父类的初始化方法会覆盖name属性
5         #self.name = person # 在这里设置就可以修改进程名字了
6         #self.person = person #如果不想覆盖进程名,就修改属性名称就可以了
7     def run(self):
8         print('%s正在和网红脸聊天' %self.name)
9         # print('%s正在和网红脸聊天' %self.person)
10        time.sleep(random.randrange(1,5))

```

```
11         print('%s正在和网红脸聊天' %self.name)
12         # print('%s正在和网红脸聊天' %self.person)
13
14
15 p1=Myprocess('哪吒')
16 p1.start()
17 print(p1.pid)      #可以查看子进程的进程id
```

[回到顶部](#)

## 进程同步(multiprocess.Lock)

### 锁 —— multiprocessing.Lock

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。

当多个进程使用同一份数据资源的时候，就会引发数据安全或顺序混乱问题。

```
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(3):
        p=Process(target=work,args=(i,))
        p.start()
```

```
# 由并发变成了串行,牺牲了运行效率,但避免了竞争
import os
import time
import random
from multiprocessing import Process,Lock

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

if __name__ == '__main__':
    lock=Lock()
    for i in range(3):
        p=Process(target=work,args=(lock,i))
        p.start()
```

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，来看看数据安全性的重要性。

```
#文件db的内容为: {"count":1}
#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
def search():
    dic=json.load(open('db'))
```

```

print('\033[43m剩余票数%s\033[0m' % dic['count'])

def get():
    dic=json.load(open('db'))
    time.sleep(0.1) #模拟读数据的网络延迟
    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'))
        print('\033[43m购票成功\033[0m')

def task():
    search()
    get()

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

```

☐

📄

#文件db的内容为: {"count":5}  
 #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

def search():
    dic=json.load(open('db'))
    print('\033[43m剩余票数%s\033[0m' % dic['count'])

def get():
    dic=json.load(open('db'))
    time.sleep(random.random()) #模拟读数据的网络延迟
    if dic['count'] >0:
        dic['count']-=1
        time.sleep(random.random()) #模拟写数据的网络延迟
        json.dump(dic,open('db','w'))
        print('\033[32m购票成功\033[0m')
    else:
        print('\033[31m购票失败\033[0m')

def task(lock):
    search()
    lock.acquire()
    get()
    lock.release()

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

```

📄

📄

#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, 但牺牲了速度却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multiprocess

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, 往往可以获得更好



[回到顶部](#)

# 进程间通信——队列（multiprocess.Queue）

## 进程间通信

IPC(Inter-Process Communication)

## 队列

### 概念介绍

创建共享的进程队列，Queue是多进程安全的队列，可以使用Queue实现多进程之间的数据传递。

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定实现。

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，还需要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False，同q.get\_nowait()方法。

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QueueFull异常。

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目。

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间，队列可能已经非空。

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）。。

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关闭队列。

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程调用。

### 代码实例

'''

multiprocessing模块支持进程间通信的两种主要形式：管道和队列

都是基于消息传递实现的，但是队列接口

'''

from multiprocessing import Queue

q=Queue(3)

#put ,get ,put\_nowait,get\_nowait,full,empty

q.put(3)

https://www.cnblogs.com/Eva-J/articles/8253549.html

13/25

```

q.put(3)
q.put(3)
# q.put(3)    # 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(3) # 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()) #满了

print(q.get())
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

```

上面这个例子还没有加入进程通信，只是先来看看队列为我们提供的方法，以及这些方法的使用和现象。

```

import time
from multiprocessing import Process, Queue

def f(q):
    q.put([time.asctime(), 'from Eva', 'hello']) #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

if __name__ == '__main__':
    q = Queue() #创建一个Queue对象
    p = Process(target=f, args=(q,)) #创建一个进程
    p.start()
    print(q.get())
    p.join()

```

上面是一个queue的简单应用，使用队列q对象调用get函数来取得队列中最先进入的数据。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__':
    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()
```



## 生产者消费者模型

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。

### 基于队列实现生产者消费者模型

```
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')
```



此时的问题是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。

解决方案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(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))
    q.put(None) #发送结束信号
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()
    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，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')
```



### JoinableQueue([maxsize])

创建可连接的共享进程队列。这就像是一个Queue对象，但队列允许项目的使用者通知生产者项目已经被成功处理。通知进程是使用共享的信号和条件变量来实现的。



JoinableQueue的实例q除了与Queue对象相同的方法之外，还具有以下方法：

q.task\_done()

使用者使用此方法发出信号，表示q.get()返回的项目已经被处理。如果调用此方法的次数大于从队列中删除的项目数量，将引发ValueError。

q.join()

生产者将使用此方法进行阻塞，直到队列中所有项目均被处理。阻塞将持续到为队列中的每个项目均调用q.task\_done()方法为止。下面的例子说明如何建立永远运行的进程，使用和处理队列上的项目。生产者将项目放入队列，并等待它们被处理。



```

from multiprocessing import Process,JoinableQueue
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))
        q.task_done() #向q.join()发送一次信号,证明一个数据已经被取走了

def producer(name,q):
    for i in range(10):
        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))
    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。
```

```

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
    c2.daemon=True

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

    p1.join()
    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

```

[回到顶部](#)

## 进程之间的数据共享

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。

**但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。**

**以后我们会尝试使用数据库来解决现在进程之间的数据共享问题。**

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

```

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)

```

# 进程池和multiprocess.Pool模块

## 进程池

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，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。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.Pool模块

### 概念介绍

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

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

1 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。  
2 '''需要强调的是：此操作并不会在所有池工作进程中并行执行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  
3  
4 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。  
5 '''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cal  
6  
7 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  
8  
9 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

1 方法apply\_async()和map\_async () 的返回值是AsyncResult的实例obj。实例具有以下方法  
2 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  
3 obj.ready():如果调用完成，返回True  
4 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  
5 obj.wait([timeout]):等待结果变为可用。  
6 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

### 代码实例

#### 进程池和多进程效率对比

p.map进程池和进程效率测试

#### 同步和异步

import os,time  
from multiprocessing import Pool  
  
def work(n):  
 print('%s run' %os.getpid())  
 time.sleep(3)  
 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
        # 但不管该任务是否存在阻塞,同步调用都会在原地等着

    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
        # 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()
    p.join()
    for res in res_l:
        print(res.get()) #使用get来获取apply_async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

```



## 练习



```

#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。

### 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。主进程则调用一个函数

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的过



```

from multiprocessing import Pool
import requests
import json
import os

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

def parse_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=parse_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

'''
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...',...}]
...
```



```
import re
from urllib.request import urlopen
from multiprocessing import Pool

def get_page(url,pattern):
    response=urlopen(url).read().decode('utf-8')
    return pattern,response

def parse_page(info):
    pattern,page_content=info
    res=re.findall(pattern,page_content)
    for item in res:
        dic={
            'index':item[0].strip(),
            'title':item[1].strip(),
            'actor':item[2].strip(),
            'time':item[3].strip(),
        }
        print(dic)
if __name__ == '__main__':
    regex = r'<dd>.*?<.*?class="board-index.*?>(\d+)</i>.*?title="(.*?)".*?class="movie-item-info".*?'
    pattern1=re.compile(regex,re.S)

    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()
```



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
```

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

参考资料  
<http://www.cnblogs.com/linhaifeng/articles/6817679.html>  
<https://www.jianshu.com/p/1200fd49b583>  
<https://www.jianshu.com/p/aed6067eeac9>

好文要顶 关注我 收藏该文





Eva\_J  
关注 - 7  
粉丝 - 4193

+加关注

6

1

posted @ 2018-01-19 08:32 Eva\_J 阅读(17440) 评论(13) 编辑 收藏

发表评论

- #1楼 2018-07-08 17:31 | 潇洒劫个妞

厉害

支持(0) 反对(0)
- #2楼 2018-10-23 17:48 | XXXXXXXXxyyy

插个眼, nice!

支持(0) 反对(0)
- #3楼 2018-11-06 10:51 | Mr.暖阳

老师你这个示例代码不能查看: p.map进程池和进程效率测试

支持(1) 反对(0)
- #4楼 2018-11-20 11:26 | 清风不问烟雨

老师, 这个爬虫的代码运行时报错。提示是最后的i.get()的地方报错, 错误如下: multiprocessing.pool.MaybeEncodingError: Error sending result: '<multiprocessing.pool.ExceptionWithTraceback object at 0x000001D65D829C88>'. Reason: 'TypeError("cannot serialize '\_io.BufferedReader' object",)' 不知道是怎么引起的。

支持(2) 反对(0)
- #5楼 2018-12-05 18:59 | wangspy

声音好听的老师, 你好! Manager例子中的代码(去掉lock部分) 我用pycharm2017.3版本 始终不能出现错误的情况

支持(0) 反对(0)
- #6楼 2019-01-09 17:29 | linux超



回复 引用

美女老师, 问你个问题 为什么函数外部会打印两次呢? 且进程id不一样一次是子进程的一次是主进程的id, 求解答





- #7楼 [楼主] 2019-01-10 16:39 | Eva\_J

回复 引用

@_linux超			
这是一个综合性很强的问题，需要你对windows操作系统启动子进程以及python的模块都有了解。 windows在启动子进程的时候会将主进程文件倒入到子进程中。导入模块就相当于执行这个模块中的代码，所以第一个print会在主进程中执行一次，又在被导入的过程中在子进程中又执行了一次。		支持(0) 反对(0)	
#8楼	2019-03-02 09:05   <a href="#">Tank-Li</a>	回复	引用
用多进程的方式实现socketserver的效果,不能在子进程中有input(),否则会报错; 但是在多线程里面就可以.目前只知现象不知原因,求老师解答. 么么哒~~比心		支持(0) 反对(0)	
#9楼	2019-04-08 10:58   <a href="#">二哈的博客</a>	回复	引用
看海峰老师的讲解再看你的博客来学习		支持(0) 反对(0)	
#10楼	2019-05-10 13:37   <a href="#">运维小学生</a>	回复	引用
mark		支持(0) 反对(0)	
#11楼	2019-08-13 11:29   <a href="#">只会玩辅助</a>	回复	引用
老师你的第二种创建进程的方式不用放在__main__里面吗		支持(0) 反对(0)	
#12楼	2019-08-26 15:31   <a href="#">changxin7</a>	回复	引用
@_linux超 操作系统和系统内部的优化算法有关系,还有电脑的运行速度也有关系		支持(0) 反对(0)	
#13楼	2020-02-28 09:32   <a href="#">Bravo_Jack</a>	回复	引用
请问下为什么进程池的apply方法是同步执行,就能够立刻获取结果呢?		支持(0) 反对(0)	

发表评论

编辑 预览

B    

支持 Markdown

提交评论

退出 订阅评论

- [Ctrl+Enter快捷键提交]
- 【推荐】超50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库
  - 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
  - 【推荐】有道智云周年庆，API服务大放送，注册即送100元体验金！
  - 【推荐】30+视频&10+案例纵横文件与IO领域 | Java开发者高级应用站



相关博文：

- Python之路： 进程、线程
- python之路-进程、线程
- Python之路-进程
- Python之路-进程
- python之路-进程
- » 更多推荐...

最新 IT 新闻：

- 美国强迫收购TikTok、限45天、还要中间费！李开复：做法不可思议
- 联发科回应与华为签订采购大单：不评论单一客户讯息
- 苹果回应Siri专利案：对诉讼感到失望，最高法院认证机构表示未侵权
- 互联网免费时代终结 全面付费用户被套路
- 2020胡润全球独角兽榜上的企业，有多少能留到明年
- » 更多新闻...