Eva_J

程序媛

python之路——线程

简介

回到顶部

操作系统线程理论

回到顶部

线程概念的引入背景

进程

之前我们已经了解了操作系统中进程的概念，程序并不能单独运行，只有将程序装载到内存中，系统为它分配资源才能运行，而这种执行的程序就称之为进程。程序和进程的区别就在于：程序是指令的集合，它是进程运行的静态描述文本；进程是程序的一次执行活动，属于动态概念。在多道编程中，我们允许多个程序同时加载到内存中，在操作系统的调度下，可以实现并发地执行。这是这样的设计，大大提高了CPU的利用率。进程的出现让每个用户感觉到自己独享CPU，因此，进程就是为了在CPU上实现多道编程而提出的。

有了进程为什么要有线程

进程有很多优点，它提供了多道编程，让我们感觉我们每个人都拥有自己的CPU和其他资源，可以提高计算机的利用率。很多人就不理解了，既然进程这么优秀，为什么还要线程呢？其实，仔细观察就会发现进程还是有很多缺陷的，主要体现在两点上：

- 进程只能在一个时间干一件事，如果想同时干两件事或多件事，进程就无能为力了。
- 进程在执行的过程中如果阻塞，例如等待输入，整个进程就会挂起，即使进程中有些工作不依赖于输入的数据，也将无法执行。

昵称： Eva_J
园龄： 4年9个月
粉丝： 4193
关注： 7
+加关注

< 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)

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)

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)

如果这两个缺点理解比较困难的话，举个现实的例子也许你就清楚了：如果把我们上课的过程看成一个进程的话，那么我们要做的是耳朵听老师讲课，手上还要记笔记，脑子还要思考问题，这样才能高效的完成听课的任务。而如果只提供进程这个机制的话，上面这三件事将不能同时执行，同一时间只能做一件事，听的时候就不能记笔记，也不能用脑子思考，这是其一；如果老师在黑板上写演算过程，我们开始记笔记，而老师突然有一步推不下去了，阻塞住了，他在那边思考着，而我们呢，也不能干其他事，即使你想趁此时思考一下刚才没听懂的一个问题都不行，这是其二。

现在你应该明白了进程的缺陷了，而解决的办法很简单，我们完全可以让听、写、思三个独立的过程，并行起来，这样很明显可以提高听课的效率。而实际的操作系统中，也同样引入了这种类似的机制——线程。

线程的出现

60年代，在OS中能拥有资源和独立运行的基本单位是进程，然而随着计算机技术的发展，进程出现了很多弊端，一是由于进程是资源拥有者，创建、撤消与切换存在较大的时空开销，因此需要引入**轻型进程**；二是由于对称多处理机（SMP）出现，**可以满足多个运行单位**，而多个进程并行开销过大。

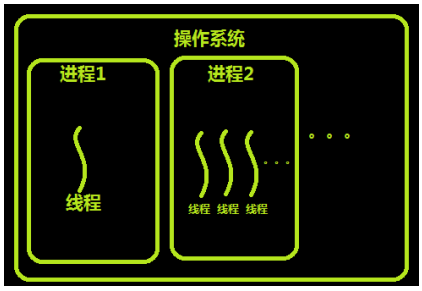
因此在80年代，出现了**能独立运行的基本单位**——线程（Threads）。

注意：进程是资源分配的最小单位,线程是CPU调度的最小单位.

每一个进程中至少有一个线程。

回到顶部

进程和线程的关系



线程与进程的区别可以归纳为以下4点：

1）地址空间和其它资源（如打开文件）：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。

2）通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

3）调度和切换：线程上下文切换比进程上下文切换要快得多。

4）在多线程操作系统中，进程不是一个可执行的实体。

*通过漫画了解线程进城

回到顶部

线程的特点

在多线程的操作系统中，通常是在一个进程中包括多个线程，每个线程都是作为利用CPU的基本单位，是花费最小开销的实体。线程具有以下属性。

1）轻型实体

线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证独立运行的资源。

线程的实体包括程序、数据和TCB。线程是动态概念，它的动态特性由线程控制块TCB（Thread Control Block）描述。

TCB包括以下信息：
（1）线程状态。
（2）当线程不运行时，被保存的现场资源。
（3）一组执行堆栈。
（4）存放每个线程的局部变量主存区。
（5）访问同一个进程中的主存和其它资源。
用于指示被执行指令序列的程序计数器、保留局部变量、少数状态参数和返回地址等的一组寄存器和堆栈。

2）独立调度和分派的基本单位。

在多线程OS中，线程是能独立运行的基本单位，因而也是独立调度和分派的基本单位。由于线程很"轻"，故线程的切换非常迅速且开销小（在同一进程中的）。

3）共享进程资源。

线程在同一进程中的各个线程，都可以共享该进程所拥有的资源，这首先表现在：所有线程都具有相同的进程id，这意味着，线程可以访问该进程的每一个内存资源；此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。由于同一个进程内的线程共享内存和文件，所以线程之间互相通信不必调用内核。

4）*可并发执行。*

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 employee 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

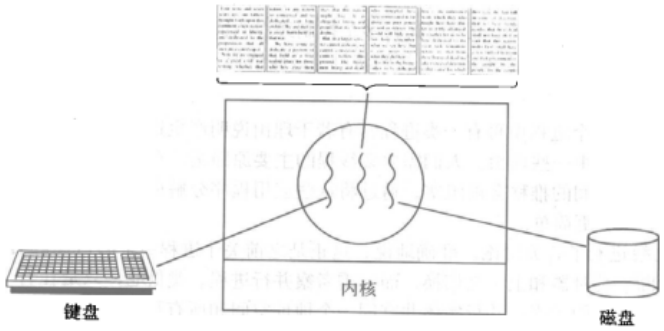
阅读排行榜

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)

在一个进程中的多个线程之间，可以并发执行，甚至允许在一个进程中所有线程都能并发执行；同样，不同进程中的线程也能并发执行，充分利用和发挥了处理机与外围设备并行工作的能力。

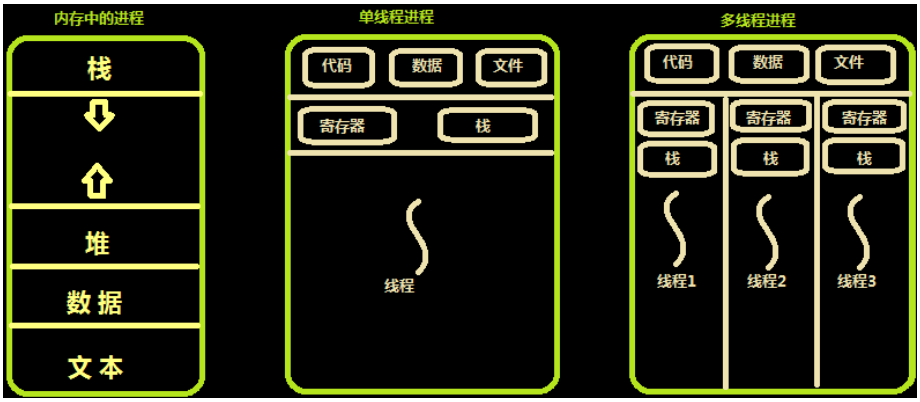
回到顶部

使用线程的实际场景



开启一个字处理软件进程，该进程肯定需要办不止一件事情，比如监听键盘输入，处理文字，定时自动将文字保存到硬盘，这三个任务操作的都是同一块数据，因而不能用多进程。只能在一个进程里并发地开启三个线程,如果是单线程，那就只能是，键盘输入时，不能处理文字和自动保存，自动保存时又不能输入和处理文字。

内存中的线程



多个线程共享同一个进程的地址空间中的资源，是对一台计算机上多个进程的模拟，有时也称线程为轻量级的进程。

而对一台计算机上多个进程，则共享物理内存、磁盘、打印机等其他物理资源。多线程的运行也多进程的运行类似，是cpu在多个线程之间的快速切换。

不同的进程之间是充满敌意的，彼此是抢占、竞争cpu的关系，如果迅雷会和QQ抢资源。而同一个进程是由一个程序员的程序创建，所以同一进程内的线程是合作关系，一个线程可以访问另外一个线程的内存地址，大家都是共享的，一个线程干死了另外一个线程的内存，那纯属程序员脑子有问题。

类似于进程，每个线程也有自己的堆栈，不同于进程，线程库无法利用时钟中断强制线程让出CPU，可以调用thread_yield运行线程自动放弃cpu，让另外一个线程运行。

线程通常是有益的，但是带来了不小程序设计难度，线程的问题是：

1. 父进程有多个线程，那么开启的子线程是否需要同样多的线程

2. 在同一个进程中，如果一个线程关闭了文件，而另外一个线程正准备往该文件内写内容呢?

因此，在多线程的代码中，需要更多的心思来设计程序的逻辑、保护程序的数据。

回到顶部

用户级线程和内核级线程（了解）

线程的实现可以分为两类：用户级线程(User-Level Thread)和内核线线程(Kernel-Level Thread)，后者又称为内核支持的线程或轻量级进程。在多线程操作系统中，各个系统的实现方式并不相同，在有的系统中实现了用户级线程，有的系统中实现了内核级线程。

用户级线程

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)

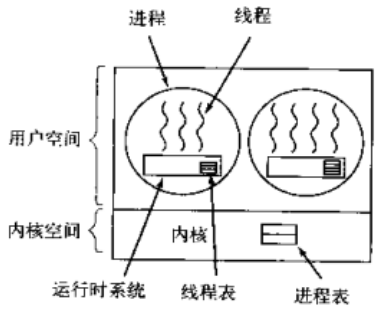
评论排行榜

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)

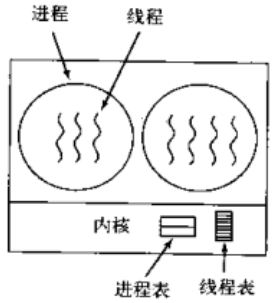
内核的切换由用户态程序自己控制内核切换,不需要内核干涉，少了进出内核态的消耗，但不能很好的利用多核Cpu。



在用户空间模拟操作系统对进程的调度，来调用一个进程中的线程，每个进程中都会有一个运行时系统，用来调度线程。此时当该进程获取cpu时，进程内再调度出一个线程去执行，同一时刻只有一个线程执行。

内核级线程

内核级线程:切换由内核控制，当线程进行切换的时候，由用户态转化为内核态。切换完毕要从内核态返回用户态；可以很好的利用smp，即利用多核cpu。windows线程就是这样的。



用户级与内核级线程的对比

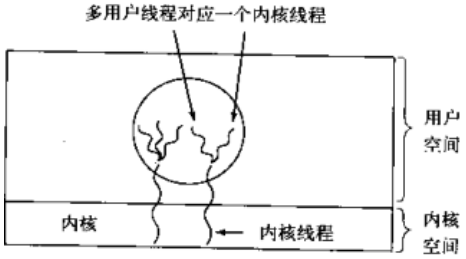
```
内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的。
用户级线程的创建、撤消和调度不需要OS内核的支持，是在语言（如Java）这一级处理的；而内核支持线程的创建、撤消和调度都需O
用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。
在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内
用户级线程的程序实体是运行在用户态下的程序，而内核支持线程的程序实体则是可以运行在任何状态下的程序。
```

```
优点：当有多个处理机时，一个进程的多个线程可以同时执行。
缺点：由内核进行调度。
```

```
优点：
线程的调度不需要内核直接参与，控制简单。
可以在不支持线程的操作系统中实现。
创建和销毁线程、线程切换代价等线程管理的代价比内核线程少得多。
允许每个进程定制自己的调度算法，线程管理比较灵活。
线程能够利用的表空间和堆栈空间比内核级线程多。
同一进程中只能同时有一个线程在运行，如果有一个线程使用了系统调用而阻塞，那么整个进程都会被挂起。另外，页面失效也会产生同样
缺点：
资源调度按照进程进行，多个处理机下，同一个进程中的线程只能在同一个处理机下分时复用
```

混合实现

用户级与内核级的多路复用，内核同一调度内核线程，每个内核线程对应n个用户线程



linux操作系统的 NPTL

历史
在内核2.6以前的调度实体都是进程，内核并没有真正支持线程。它是能过一个系统调用clone()来实现的，这个调用创建了一份调用进程

很显然，为了改进LinuxThread必须得到内核的支持，并且需要重写线程库。为了实现这个需求，开始有两个相互竞争的项目：IBM启动

NPTL最开始在redhat linux 9里发布，现在从RHEL3起内核2.6起都支持NPTL，并且完全成了GNU C库的一部分。

设计
NPTL使用了跟LinuxThread相同的办法，在内核里面线程仍然被当作是一个进程，并且仍然使用了clone()系统调用(在NPTL库里调用)

NPTL也是一个1*1的线程库，就是说，当你使用pthread_create()调用创建一个线程后，在内核里就相应创建了一个调度实体，在lin

除NPTL的1*1模型外还有一个m*n模型，通常这种模型的用户线程数会比内核的调度实体多。在这种实现里，线程库本身必须去处理可能

回到顶部

线程和python

回到顶部

理论知识

全局解释器锁GIL

Python代码的执行由Python虚拟机(也叫解释器主循环)来控制。Python在设计之初就考虑到要在主循环中，同时只有一个线程在执行。虽然 Python 解释器中可以“运行”多个线程，但在任意时刻只有一个线程在解释器中运行。

对Python虚拟机的访问由全局解释器锁(GIL)来控制，正是这个锁能保证同一时刻只有一个线程在运行。

在多线程环境中，Python 虚拟机按以下方式执行：

a、设置 GIL；

b、切换到一个线程去运行；

c、运行指定数量的字节码指令或者线程主动让出控制(可以调用 time.sleep(0))；

d、把线程设置为睡眠状态；

e、解锁 GIL；

d、再次重复以上所有步骤。

在调用外部代码(如 C/C++扩展函数)的时候，GIL将会被锁定，直到这个函数结束为止(由于在这期间没有Python的字节码被运行，所以不会做线程切换)编写扩展的程序员可以主动解锁GIL。

python线程模块的选择

Python提供了几个用于多线程编程的模块，包括thread、threading和Queue等。thread和threading模块允许程序员创建和管理线程。thread模块提供了基本的线程和锁的支持，threading提供了更高级别、功能更强的线程管理的功能。Queue模块允许用户创建一个可以用于多个线程之间共享数据的队列数据结构。

避免使用thread模块，因为更高级别的threading模块更为先进，对线程的支持更为完善，而且使用thread模块里的属性有可能会与threading出现冲突；其次低级别的thread模块的同步原语很少(实际上只有一个)，而threading模块则有很多；再者，thread模块中当主线程结束时，所有的线程都会被强制结束掉，没有警告也不会有正常的清除工作，至少threading模块能确保重要的子线程退出后进程才退出。

thread模块不支持守护线程，当主线程退出时，所有的子线程不论它们是否还在工作，都会被强行退出。而threading模块支持守护线程，守护线程一般是一个等待客户请求的服务器，如果没有客户提出请求它就在那等着，如

果设定一个线程为守护线程，就表示这个线程是不重要的，在进程退出的时候，不用等待这个线程退出。

threading模块

multiprocess模块的完全模仿了threading模块的接口，二者在使用层面，有很大的相似性，因而不再详细介绍（官方链接）

回到顶部

线程的创建Threading.Thread类

线程的创建

```
from threading import Thread
import time
def sayhi(name):
    time.sleep(2)
    print('%s say hello' %name)

if __name__ == '__main__':
    t=Thread(target=sayhi,args=('egon',))
    t.start()
    print('主线程')
```

```
from threading import Thread
import time
class Sayhi(Thread):
    def __init__(self,name):
        super().__init__()
        self.name=name
    def run(self):
        time.sleep(2)
        print('%s say hello' % self.name)


if __name__ == '__main__':
    t = Sayhi('egon')
    t.start()
    print('主线程')
```

多线程与多进程

```
from threading import Thread
from multiprocessing import Process
import os

def work():
    print('hello',os.getpid())

if __name__ == '__main__':
    #part1:在主进程下开启多个线程,每个线程都跟主进程的pid一样
    t1=Thread(target=work)
    t2=Thread(target=work)
    t1.start()
    t2.start()
    print('主线程/主进程pid',os.getpid())

    #part2:开多个进程,每个进程都有不同的pid
    p1=Process(target=work)
    p2=Process(target=work)
    p1.start()
    p2.start()
    print('主线程/主进程pid',os.getpid())
```

```
from threading import Thread
from multiprocessing import Process
import os

def work():
    print('hello')

if __name__ == '__main__':
    #在主进程下开启线程
    t=Thread(target=work)
    t.start()
    print('主线程/主进程')
    '''
    打印结果:
    hello
    主线程/主进程
    '''

    #在主进程下开启子进程
    t=Process(target=work)
    t.start()
    print('主线程/主进程')
    '''
    打印结果:
    主线程/主进程
    hello
    '''
```

```
from  threading import Thread
from multiprocessing import Process
import os
def work():
    global n
    n=0

if __name__ == '__main__':
    # n=100
    # p=Process(target=work)
    # p.start()
    # p.join()
    # print('主',n) #毫无疑问子进程p已经将自己的全局的n改成了0,但改的仅仅是它自己的,查看父进程的n仍然为100


    n=1
    t=Thread(target=work)
    t.start()
    t.join()
    print('主',n) #查看结果为0,因为同一进程内的线程之间共享进程内的数据
```

同一进程内的线程共享该进程的数据?

练习： 多线程实现socket

```
#_*_coding:utf-8_*_
#!/usr/bin/env python
import multiprocessing
import threading

import socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind(('127.0.0.1',8080))
s.listen(5)
```

```
def action(conn):
    while True:
        data=conn.recv(1024)
        print(data)
        conn.send(data.upper())

if __name__ == '__main__':

    while True:
        conn,addr=s.accept()


        p=threading.Thread(target=action,args=(conn,))
        p.start()
```

```
#_*_coding:utf-8_*_
#!/usr/bin/env python


import socket

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

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

    s.send(msg.encode('utf-8'))
    data=s.recv(1024)
    print(data)
```

Thread类的其他方法

```
Thread实例对象的方法
  # isAlive(): 返回线程是否活动的。
  # getName(): 返回线程名。
  # setName(): 设置线程名。

threading模块提供的一些方法:
  # threading.currentThread(): 返回当前的线程变量。
  # threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线
  # threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。
```

```
from threading import Thread
import threading
from multiprocessing import Process
import os

def work():
    import time
    time.sleep(3)
    print(threading.current_thread().getName())


if __name__ == '__main__':
    #在主进程下开启线程
    t=Thread(target=work)
    t.start()

    print(threading.current_thread().getName())
    print(threading.current_thread()) #主线程
    print(threading.enumerate()) #连同主线程在内有两个运行的线程
```

```
    print(threading.active_count())
    print('主线程/主进程')

    '''
    打印结果:
    MainThread
    <_MainThread(MainThread, started 140735268892672)>
    [<_MainThread(MainThread, started 140735268892672)>, <Thread(Thread-1, started 123145307557888)>]
    主线程/主进程
    Thread-1
    '''
```

```
from threading import Thread
import time
def sayhi(name):
    time.sleep(2)
    print('%s say hello' %name)

if __name__ == '__main__':
    t=Thread(target=sayhi,args=('egon',))
    t.start()
    t.join()
    print('主线程')
    print(t.is_alive())
    '''
    egon say hello
    主线程
    False
    '''
```

守护线程

无论是进程还是线程，都遵循：守护xx会等待主xx运行完毕后被销毁。需要强调的是：运行完毕并非终止运行

```
#1.对主进程来说，运行完毕指的是主进程代码运行完毕
#2.对主线程来说，运行完毕指的是主线程所在的进程内所有非守护线程统统运行完毕，主线程才算运行完毕
```

```
#1 主进程在其代码结束后就已经算运行完毕了（守护进程在此时就被回收），然后主进程会一直等非守护的子进程都运行完毕后回收子进
#2 主线程在其他非守护线程运行完毕后才算运行完毕（守护线程在此时就被回收）。因为主线程的结束意味着进程的结束，进程整体的资
```

```
from threading import Thread
import time
def sayhi(name):
    time.sleep(2)
    print('%s say hello' %name)

if __name__ == '__main__':
    t=Thread(target=sayhi,args=('egon',))
    t.setDaemon(True) #必须在t.start()之前设置
    t.start()

    print('主线程')
    print(t.is_alive())
    '''
    主线程
    True
    '''
```

```
from threading import Thread
import time
def foo():
```

```
    print(123)
    time.sleep(1)
    print("end123")

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


t1=Thread(target=foo)
t2=Thread(target=bar)

t1.daemon=True
t1.start()
t2.start()
print("main-------")
```

回到顶部

锁

锁与GIL

同步锁

```
from threading import Thread
import os,time
def work():
    global n
    temp=n
    time.sleep(0.1)
    n=temp-1
if __name__ == '__main__':
    n=100
    l=[]
    for i in range(100):
        p=Thread(target=work)
        l.append(p)
        p.start()
    for p in l:
        p.join()

    print(n) #结果可能为99
```

```
import threading
R=threading.Lock()
R.acquire()
'''
对公共数据的操作
'''
R.release()
```

```
from threading import Thread,Lock
import os,time
def work():
    global n
    lock.acquire()
    temp=n
    time.sleep(0.1)
    n=temp-1
    lock.release()
if __name__ == '__main__':
```

```
    lock=Lock()
    n=100
    l=[]
    for i in range(100):
        p=Thread(target=work)
        l.append(p)
        p.start()
    for p in l:
        p.join()

    print(n) #结果肯定为0，由原来的并发执行变成串行，牺牲了执行效率保证了数据安全
```

```
#不加锁:并发执行,速度快,数据不安全
from threading import current_thread,Thread,Lock
import os,time
def task():
    global n
    print('%s is running' %current_thread().getName())
    temp=n
    time.sleep(0.5)
    n=temp-1


if __name__ == '__main__':
    n=100
    lock=Lock()
    threads=[]
    start_time=time.time()
    for i in range(100):
        t=Thread(target=task)
        threads.append(t)
        t.start()
    for t in threads:
        t.join()

    stop_time=time.time()
    print('主:%s n:%s' %(stop_time-start_time,n))

'''
Thread-1 is running
Thread-2 is running
......
Thread-100 is running
主:0.5216062068939209 n:99
'''


#不加锁:未加锁部分并发执行,加锁部分串行执行,速度慢,数据安全
from threading import current_thread,Thread,Lock
import os,time
def task():
    #未加锁的代码并发运行
    time.sleep(3)
    print('%s start to run' %current_thread().getName())
    global n
    #加锁的代码串行运行
    lock.acquire()
    temp=n
    time.sleep(0.5)
    n=temp-1
    lock.release()

if __name__ == '__main__':
    n=100
    lock=Lock()
    threads=[]
    start_time=time.time()
    for i in range(100):
        t=Thread(target=task)
        threads.append(t)
        t.start()
    for t in threads:
```

```
        t.join()
    stop_time=time.time()
    print('主:%s n:%s' %(stop_time-start_time,n))

'''
Thread-1 is running
Thread-2 is running
......
Thread-100 is running
主:53.294203758239746 n:0
'''

#有的同学可能有疑问:既然加锁会让运行变成串行,那么我在start之后立即使用join,就不用加锁了啊,也是串行的效果啊
#没错:在start之后立刻使用jion,肯定会将100个任务的执行变成串行,毫无疑问,最终n的结果也肯定是0,是安全的,但问题是
#start后立即join:任务内的所有代码都是串行执行的,而加锁,只是加锁的部分即修改共享数据的部分是串行的
#单从保证数据安全方面,二者都可以实现,但很明显是加锁的效率更高.
from threading import current_thread,Thread,Lock
import os,time
def task():
    time.sleep(3)
    print('%s start to run' %current_thread().getName())
    global n
    temp=n
    time.sleep(0.5)
    n=temp-1


if __name__ == '__main__':
    n=100
    lock=Lock()
    start_time=time.time()
    for i in range(100):
        t=Thread(target=task)
        t.start()
        t.join()
    stop_time=time.time()
    print('主:%s n:%s' %(stop_time-start_time,n))

'''
Thread-1 start to run
Thread-2 start to run
......
Thread-100 start to run
主:350.6937336921692 n:0 #耗时是多么的恐怖
'''

)
```

死锁与递归锁

进程也有死锁与递归锁，在进程那里忘记说了，放到这里一切说了额

所谓死锁： 是指两个或两个以上的进程或线程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程，如下就是死锁

```
from threading import Lock as Lock
import time
mutexA=Lock()
mutexA.acquire()
mutexA.acquire()
print(123)
mutexA.release()
mutexA.release()
```

解决方法，递归锁，在Python中为了支持在同一线程中多次请求同一资源，python提供了可重入锁RLock。

这个RLock内部维护着一个Lock和一个counter变量，counter记录了acquire的次数，从而使得资源可以被多次require。直到一个线程所有的acquire都被release，其他的线程才能获得资源。上面的例子如果使用RLock代替Lock，则不会发生死锁：

```
from threading import RLock as Lock
import time
mutexA=Lock()
mutexA.acquire()
mutexA.acquire()
print(123)
mutexA.release()
mutexA.release()
```

典型问题：科学家吃面

```
import time
from threading import Thread,Lock
noodle_lock = Lock()
fork_lock = Lock()
def eat1(name):
    noodle_lock.acquire()
    print('%s 抢到了面条'%name)
    fork_lock.acquire()
    print('%s 抢到了叉子'%name)
    print('%s 吃面'%name)
    fork_lock.release()
    noodle_lock.release()

def eat2(name):
    fork_lock.acquire()
    print('%s 抢到了叉子' % name)
    time.sleep(1)
    noodle_lock.acquire()
    print('%s 抢到了面条' % name)
    print('%s 吃面' % name)
    noodle_lock.release()
    fork_lock.release()

for name in ['哪吒','egon','yuan']:
    t1 = Thread(target=eat1,args=(name,))
    t2 = Thread(target=eat2,args=(name,))
    t1.start()
    t2.start()
```

```
import time
from threading import Thread,RLock
fork_lock = noodle_lock = RLock()
def eat1(name):
    noodle_lock.acquire()
    print('%s 抢到了面条'%name)
    fork_lock.acquire()
    print('%s 抢到了叉子'%name)
    print('%s 吃面'%name)
    fork_lock.release()
    noodle_lock.release()

def eat2(name):
    fork_lock.acquire()
    print('%s 抢到了叉子' % name)
    time.sleep(1)
    noodle_lock.acquire()
    print('%s 抢到了面条' % name)
    print('%s 吃面' % name)
    noodle_lock.release()
    fork_lock.release()

for name in ['哪吒','egon','yuan']:
    t1 = Thread(target=eat1,args=(name,))
    t2 = Thread(target=eat2,args=(name,))
```

```
    t1.start()
    t2.start()
```

回到顶部

线程队列

queue队列 ： 使用import queue，用法与进程Queue一样

queue is especially useful in threaded programming when information must be exchanged safely between multiple threads.

class queue.Queue(***maxsize=0***) **#先进先出**

```
import queue

q=queue.Queue()
q.put('first')
q.put('second')
q.put('third')

print(q.get())
print(q.get())
print(q.get())
'''
结果(先进先出)：
first
second
third
'''
```

class queue.LifoQueue(***maxsize=0***) **#last in fisrt out**

```
import queue

q=queue.LifoQueue()
q.put('first')
q.put('second')
q.put('third')

print(q.get())
print(q.get())
print(q.get())
'''
结果(后进先出)：
third
second
first
'''
```

class queue.PriorityQueue(***maxsize=0***) **#存储数据时可设置优先级的队列**

```
import queue

q=queue.PriorityQueue()
#put进入一个元组，元组的第一个元素是优先级(通常是数字，也可以是非数字之间的比较)，数字越小优先级越高
q.put((20,'a'))
q.put((10,'b'))
q.put((30,'c'))

print(q.get())
print(q.get())
print(q.get())
'''
结果(数字越小优先级越高，优先级高的优先出队)：
```

```
(10, 'b')
(20, 'a')
(30, 'c')
'''
```

```
Constructor for a priority queue. maxsize is an integer that sets the upperbound limit on the number

The lowest valued entries are retrieved first (the lowest valued entry is the one returned by sorted(

exception queue.Empty
Exception raised when non-blocking get() (or get_nowait()) is called on a Queue object which is empty

exception queue.Full
Exception raised when non-blocking put() (or put_nowait()) is called on a Queue object which is full.

Queue.qsize()
Queue.empty() #return True if empty
Queue.full() # return True if full
Queue.put(item, block=True, timeout=None)
Put item into the queue. If optional args block is true and timeout is None (the default), block if n

Queue.put_nowait(item)
Equivalent to put(item, False).

Queue.get(block=True, timeout=None)
Remove and return an item from the queue. If optional args block is true and timeout is None (the def

Queue.get_nowait()
Equivalent to get(False).

Two methods are offered to support tracking whether enqueued tasks have been fully processed by daemo

Queue.task_done()
Indicate that a formerly enqueued task is complete. Used by queue consumer threads. For each get() us

If a join() is currently blocking, it will resume when all items have been processed (meaning that a

Raises a ValueError if called more times than there were items placed in the queue.

Queue.join() block直到queue被消费完毕
```

回到顶部

Python标准模块--concurrent.futures

https://docs.python.org/dev/library/concurrent.futures.html

```
#1 介绍
concurrent.futures模块提供了高度封装的异步调用接口
ThreadPoolExecutor：线程池，提供异步调用
ProcessPoolExecutor：进程池，提供异步调用
Both implement the same interface, which is defined by the abstract Executor class.

#2 基本方法
#submit(fn, *args, **kwargs)
异步提交任务

#map(func, *iterables, timeout=None, chunksize=1)
取代for循环submit的操作

#shutdown(wait=True)
相当于进程池的pool.close()+pool.join()操作
wait=True，等待池内所有任务执行完毕回收完资源后才继续
wait=False，立即返回，并不会等待池内的任务执行完毕
但不管wait参数为何值，整个程序都会等到所有任务执行完毕
submit和map必须在shutdown之前

#result(timeout=None)
```

取得结果

#add_done_callback(fn)
回调函数

done()
判断某一个线程是否完成

cancle()
取消某个任务

```
#介绍
The ProcessPoolExecutor class is an Executor subclass that uses a pool of processes to execute calls

class concurrent.futures.ProcessPoolExecutor(max_workers=None, mp_context=None)
An Executor subclass that executes calls asynchronously using a pool of at most max_workers processes


#用法
from concurrent.futures import ThreadPoolExecutor,ProcessPoolExecutor

import os,time,random
def task(n):
    print('%s is runing' %os.getpid())
    time.sleep(random.randint(1,3))
    return n**2

if __name__ == '__main__':

    executor=ProcessPoolExecutor(max_workers=3)

    futures=[]
    for i in range(11):
        future=executor.submit(task,i)
        futures.append(future)
    executor.shutdown(True)
    print('+++>')
    for future in futures:
        print(future.result())
```


```
#介绍
ThreadPoolExecutor is an Executor subclass that uses a pool of threads to execute calls asynchronousl
class concurrent.futures.ThreadPoolExecutor(max_workers=None, thread_name_prefix='')
An Executor subclass that uses a pool of at most max_workers threads to execute calls asynchronously.

Changed in version 3.5: If max_workers is None or not given, it will default to the number of process

New in version 3.6: The thread_name_prefix argument was added to allow users to control the threading
```

#用法
与ProcessPoolExecutor相同

```
from concurrent.futures import ThreadPoolExecutor,ProcessPoolExecutor

import os,time,random
def task(n):
    print('%s is runing' %os.getpid())
    time.sleep(random.randint(1,3))
    return n**2
```

```
if __name__ == '__main__':

    executor=ThreadPoolExecutor(max_workers=3)

    # for i in range(11):
    #     future=executor.submit(task,i)

    executor.map(task,range(1,12)) #map取代了for+submit
```

```
from concurrent.futures import ThreadPoolExecutor,ProcessPoolExecutor
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 parse_page(res):
    res=res.result()
    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)
    # for url in urls:
    #     p.apply_async(get_page,args=(url,),callback=pasrse_page)
    # p.close()
    # p.join()

    p=ProcessPoolExecutor(3)
    for url in urls:
        p.submit(get_page,url).add_done_callback(parse_page) #parse_page拿到的是一个future对象obj，需要用
```


Eva_J
关注 - 7
粉丝 - 4193
+加关注

支持 Markdown

提交评论 退出 订阅评论

[Ctrl+Enter快捷键提交]

相关博文：
· Python之路：线程池
· Python之路——多线程
· python之路——多线程
· python之路-------之线程
· Python之路——线程池
» 更多推荐...

最新 IT 新闻:
· 消息人士：商汤科技考虑在完成最新一轮融资后在科创板上市
· 《王者荣耀》曜FMVP皮肤“云鹰飞将”来了：飞檐走壁 如履平地
· 苏宁推49元换电池 覆盖200余款机型 已卖出超10000单
· 华为最高档天才少年：刚毕业年薪201万元 全球仅4人
· 美国强迫收购TikTok、限45天、还要中间费！李开复：做法不可思议
» 更多新闻...