Python之线程

Python**之线程**

线程

本节目录

一 背景知识

1.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。

2.有了进程为什么还要线程

```
#什么是线程：
#指的是一条流水线的工作过程，关键的一句话：一个进程内最少自带一个线程，其实进程根本不能执行，进程不是执行单位，是资源的
#线程才是执行单位
#进程：做手机屏幕的工作过程,刚才讲的
#我们的py文件在执行的时候，如果你站在资源单位的角度来看，我们称为一个主进程，如果站在代码执行的角度来看，它叫做主线程，

#进程vs线程
#1 同一个进程内的多个线程是共享该进程的资源的，不同进程内的线程资源肯定是隔离的
#2 创建线程的开销比创建进程的开销要小的多


#并发三个任务：1启动三个进程：因为每个进程中有一个线程，但是我一个进程中开启三个线程就够了
#同一个程序中的三个任务需要执行，你是用三个进程好 ，还是三个线程好？
#例子：
    # pycharm 三个任务：键盘输入  屏幕输出  自动保存到硬盘
    #如果三个任务是同步的话，你键盘输入的时候，屏幕看不到
    #咱们的pycharm是不是一边输入你边看啊，就是将串行变为了三个并发的任务
    #解决方案：三个进程或者三个线程，哪个方案可行。如果是三个进程，进程的资源是不是隔离的并且开销大，最致命的就是资源隔
```

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。

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。

3.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。

在传统操作系统中，每个进程有一个地址空间，而且默认就有一个控制线程

线程顾名思义，就是一条流水线工作的过程，一条流水线必须属于一个车间，一个车间的工作过程是一个进程

车间负责把资源整合到一起，是一个资源单位，而一个车间内至少有一个流水线

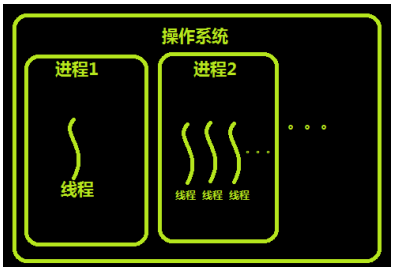
流水线的工作需要电源，电源就相当于cpu

所以，**进程只是用来把资源集中到一起（进程只是一个资源单位，或者说资源集合），而线程才是**cpu**上的执行单位。**

多线程（即多个控制线程）的概念是，在一个进程中存在多个控制线程，多个控制线程共享该进程的地址空间，相当于一个车间内有多条流水线，都共用一个车间的资源。

例如，北京地铁与上海地铁是不同的进程，而北京地铁里的13号线是一个线程，北京地铁所有的线路共享北京地铁所有的资源，比如所有的乘客可以被所有线路拉。

二 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。（就类似进程中的锁的作用）

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（现在咱们用的系统基本都是多线程的操作系统），进程不是一个可执行的实体，

真正去执行程序的不是进程，是线程，你可以理解进程就是一个线程的容器。

三 线程的特点

先简单了解一下线程有哪些特点，里面的堆栈啊主存区啊什么的后面会讲，大家先大概了解一下就好啦。

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，只是有一些必不可少的、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）。

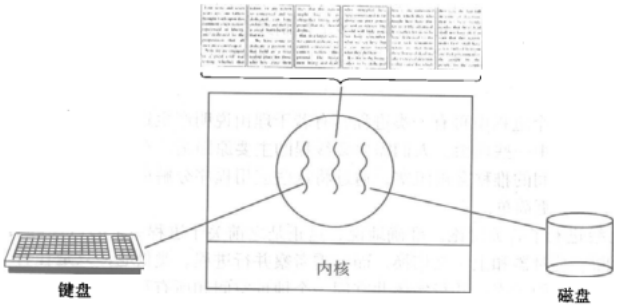
3）共享进程资源。

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。

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。

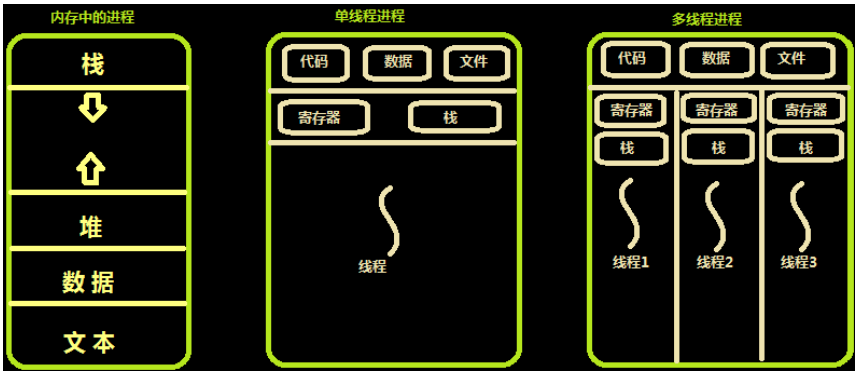
四 线程的实际应用场景



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。

之前我们将的socket是不是通过多进程去实现过呀，如果有500个人同时和我聊天，那我是不是要起500进程啊，能行吗？不好，对不对，那么怎么办，我就可以开几个进程，然后每个进程里面开多个线程来处理多个请求和通信。再举例：我用qq是一个进程，然后我和一个人聊天的时候，是不是还可以去接收别人给我发的消息啊，这个是不是并行的啊，就类似我一个进程开了多个线程来帮我并发接收消息。

五 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?

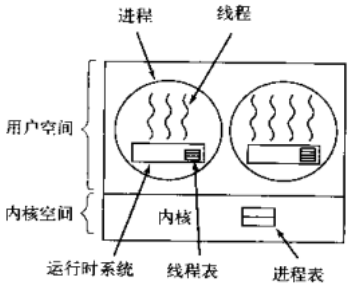
因此，在多线程的代码中，需要更多的心思来设计程序的逻辑、保护程序的数据。

六 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。

1.用户级线程

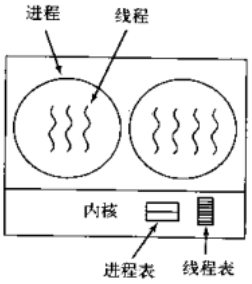
内核的切换由用户态程序自己控制内核切换,不需要内核干涉，少了进出内核态的消耗，但不能很好的利用多核Cpu。



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。

2.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。



3.用户级和内核级线程的对比

1 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。
2 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
3 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。
4 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
5 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

内核级线程的优缺点：

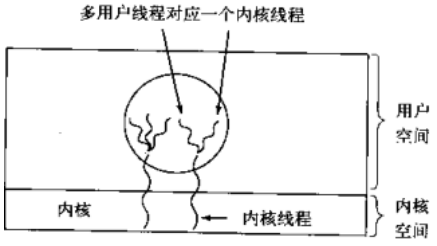
优点：当有多个处理机时，一个进程的多个线程可以同时执行。
缺点：由内核进行调度。

用户级线程的优缺点：

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
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

3.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，用户和内核都能感知到的线程，用户创建一个线程，那么操作系统内核也跟着创建一个线程来专门执行你用户的这个线程。



在linux操作系统上也实现了这种混合的方式NPTL，看下面的介绍。

4.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

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

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linu

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

七 python与线程

1.全局解释器锁GIL（用一下threading模块之后再来看~~）

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。

2.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。

就像我们熟悉的time模块，它比其他模块更加接近底层，越是接近底层，用起来越麻烦，就像时间日期转换之类的就比较麻烦，但是后面我们会学到一个datetime模块，提供了更为简便的时间日期处理方法，它是建立在time模块的基础上来的。又如socket和socketserver（底层还是用的socket）等等，这里的threading就是thread的高级模块。

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。

八 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）

我们先简单应用一下threading模块来看看并发效果：

```
import time
from threading import Thread
#多线程并发，是不是看着和多进程很类似
def func(n):
    time.sleep(1)
    print(n)

#并发效果，1秒打印出了所有的数字
for i in range(10):
    t = Thread(target=func,args=(i,))
    t.start()
```

1.线程创建

```
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=('太白',))
    t.start()
    print('主线程')
```

```
import time
from threading import Thread
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('太白')
```

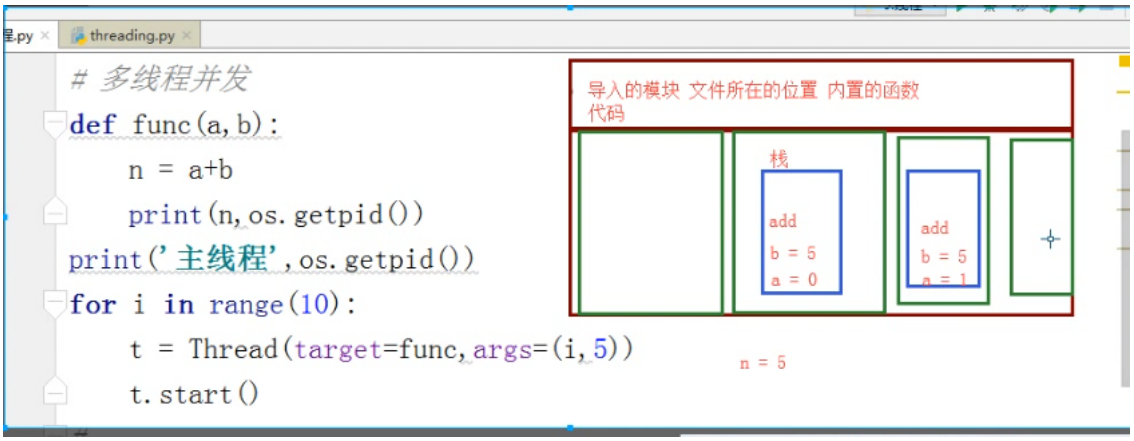
```
    t.start()
    print('主线程')
```

2.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())
```

那么哪些东西存在进程里，那些东西存在线程里呢?

进程：导入的模块、执行的python文件的文件所在位置、内置的函数、文件里面的这些代码、全局变量等等，然后线程里面有自己的堆栈（类似于一个列表，后进先出）和寄存器，里面存着自己线程的变量，操作（add）等等，占用的空间很小。




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

```
import os
import time
def work():
    print('hello')

if __name__ == '__main__':
    s1 = time.time()
    #在主进程下开启线程
    t=Thread(target=work)
    t.start()
    t.join()
    t1 = time.time() - s1
    print('进程的执行时间：',t1)
    print('主线程/主进程')
    '''
    打印结果:
    hello
    进程的执行时间：  0.0
    主线程/主进程
    '''

    s2 = time.time()
    #在主进程下开启子进程
    t=Process(target=work)
    t.start()
    t.join()
    t2 = time.time() - s2
    print('线程的执行时间：', t2)
    print('主线程/主进程')
    '''
    打印结果:
    hello
    线程的执行时间：  0.5216977596282959
    主线程/主进程
    '''
```

```
from  threading import Thread
from multiprocessing import Process
import os
def work():
    global n  #修改全局变量的值
    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()  #必须加join，因为主线程和子线程不一定谁快，一般都是主线程快一些，所有我们要等子线程执行完毕才能看出效果
    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
```

```
# 通过一个global就实现了全局变量的使用，不需要进程的IPC通信方法
```

在这里我们简单总结一下：

进程是最小的内存分配单位

线程是操作系统调度的最小党委

线程被CPU执行了

进程内至少含有一个线程

进程中可以开启多个线程

开启一个线程所需要的时间要远小于开启一个进程

多个线程内部有自己的数据栈，数据不共享

全局变量在多个线程之间是共享的

3.多线程实现socket（练习）

```
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)
        msg = input('服务端输入:') #在多线程里面可以使用input输入内容，那么就可以实现客户端和服务端的聊天了，多进程不能输入
        conn.send(bytes(msg,encoding='utf-8'))

if __name__ == '__main__':

    while True:
        conn,addr=s.accept()
        p=threading.Thread(target=action,args=(conn,))
        p.start()
```

讲一讲代码

```
mport 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)
```

在socket通信里面是不是有大量的I/O啊，recv、accept等等，我们使用多线程效率更高，因为开销小。

4.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程。
  # 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())#主线程对象
    print(threading.current_thread().getName()) #主线程名称
    print(threading.current_thread().ident) #主线程ID
    print(threading.get_ident()) #主线程ID
    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, started 14104)>
    MainThread
    14104
    14104
    [<_MainThread(MainThread, started 14104)>, <Thread(Thread-1, started 17976)>]
```

```
    2
    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=('太白',))
    t2=Thread(target=sayhi,args=('alex',))
    t.start()
    t2.start()
    t.join()  #因为这个线程用了join方法，主线程等待子线程的运行结束

    print('主线程')
    print(t.is_alive())  #所以t这个线程肯定是执行结束了，结果为False
    print(t2.is_alive()) #有可能是True，有可能是False，看子线程和主线程谁执行的快
    '''
    egon say hello
    主线程
    False
    '''
```

5.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
```

详细解释

```
#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
#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
```


```
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=('taibai',))
    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
from multiprocessing import Process
import time
def func1():
    while True:
        print(666)
        time.sleep(0.5)
def func2():
    print('hello')
    time.sleep(3)

if __name__ == '__main__':
    # t = Thread(target=func1,)
    # t.daemon = True  #主线程结束，守护线程随之结束
    # # t.setDaemon(True) #两种方式，和上面设置守护线程是一样的
    # t.start()
    # t2 = Thread(target=func2,) #这个子线程要执行3秒，主线程的代码虽然执行完了，但是一直等着子线程的任务执行完毕，主线
    # 但是主线程的的守护线程t1还在执行，说明什么，说明我的主线程还没有完毕，只不过是代码执行完了，一直等着子线程t2执行完
    # t2.start()
    # print('主线程代码执行完啦！')
    p = Process(target=func1,)
    p.daemon = True
    p.start()

    p2 = Process(target=func2,)
    p2.start()
    time.sleep(1) #让主进程等1秒，为了能看到func1的打印效果
    print('主进程代码执行完啦！') #通过结果你会发现，如果主进程的代码运行完毕了，那么主进程就结束了，因为主进程的守护进程p
```

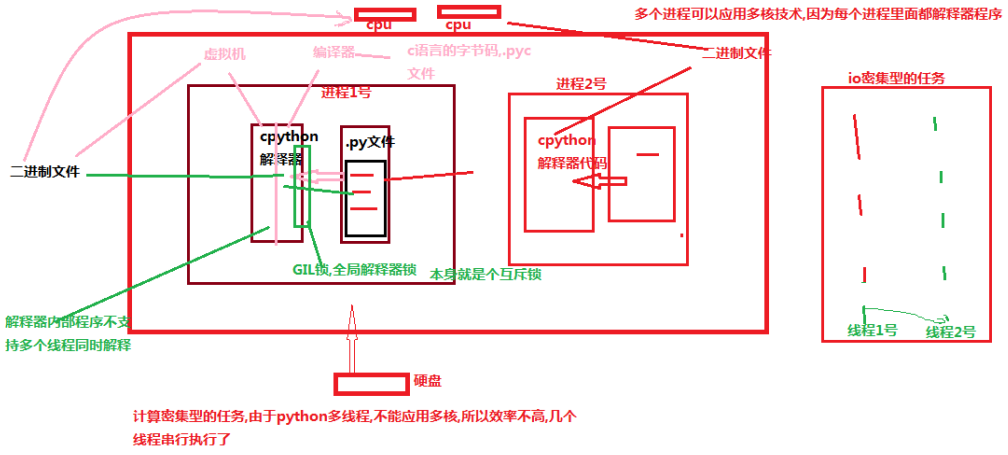
今天的内容就到这里啦，同学们整理整理前面的内容吧
~~~~~~

九 锁

1.GIL锁（Global Interpreter Lock）

首先，一些语言（java、c++、c）是支持同一个进程中的多个线程是可以应用多核CPU的，也就是我们会听到的现在4核8核这种多核CPU技术的牛逼之处。那么我们之前说过应用多进程的时候如果有共享数据是不是会出现数据不安全的问题啊，就是多个进程同时一个文件中去抢这个数据，大家都把这个数据改了，但是还没来得及去更新到原来的文件中，就被其他进程也计算了，导致数据不安全的问题啊，所以我们是不是通过加锁可以解决啊，多线程大家想一下是不是
~~~~~~

一样的，并发执行就是有这个问题。但是python最早期的时候对于多线程也加锁，但是python比较极端的（在当时电脑cpu确实只有1核）加了一个GIL全局解释锁，是解释器级别的，锁的是整个线程，而不是线程里面的某些数据操作，每次只能有一个线程使用cpu，也就说多线程用不了多核，但是他不是python语言的问题，是CPython解释器的特性，如果用Jpython解释器是没有这个问题的，Cpython是默认的，因为速度快，Jpython是java开发的，在Cpython里面就是没办法用多核，这是python的弊病，历史问题，虽然众多python团队的大神在致力于改变这个情况，但是暂没有解决。（这和解释型语言（python，php）和编译型语言有关系吗？？？待定！，编译型语言一般在编译的过程中就帮你分配好了，解释型要边解释边执行，所以为了防止出现数据不安全的情况加上了这个锁，这是所有解释型语言的弊端？？）



但是有了这个锁我们就不能并发了吗？当我们的程序是偏计算的，也就是cpu占用率很高的程序（cpu一直在计算），就不行了，但是如果你的程序是I/O型的（一般你的程序都是这个）（input、访问网址网络延迟、打开/关闭文件读写），在什么情况下用的到高并发呢（金融计算会用到，人工智能（阿尔法狗），但是一般的业务场景用不到，爬网页，多用户网站、聊天软件、处理文件），I/O型的操作很少占用CPU，那么多线程还是可以并发的，因为cpu只是快速的调度线程，而线程里面并没有什么计算，就像一堆的网络请求，我cpu非常快速的一个一个的将你的多线程调度出去，你的线程就去执行I/O操作了，

详细的GIL锁介绍：链接：https://www.cnblogs.com/clschao/articles/9705317.html

2.同步锁

三个需要注意的点：
#1.线程抢的是GIL锁，GIL锁相当于执行权限，拿到执行权限后才能拿到互斥锁Lock，其他线程也可以抢到GIL，但如果发现Lock仍然

#2.join是等待所有，即整体串行，而锁只是锁住修改共享数据的部分，即部分串行，要想保证数据安全的根本原理在于让并发变成串行

#3. 一定要看本小节最后的GIL与互斥锁的经典分析

GIL VS Lock

机智的同学可能会问到这个问题，就是既然你之前说过了，Python已经有一个GIL来保证同一时间只能有一个线程来执行了，为什么

首先我们需要达成共识：锁的目的是为了保护共享的数据，同一时间只能有一个线程来修改共享的数据

然后，我们可以得出结论：保护不同的数据就应该加不同的锁。

最后，问题就很明朗了，GIL 与Lock是两把锁，保护的数据不一样，前者是解释器级别的（当然保护的就是解释器级别的数据，比如

过程分析：所有线程抢的是GIL锁，或者说所有线程抢的是执行权限

线程1抢到GIL锁，拿到执行权限，开始执行，然后加了一把Lock，还没有执行完毕，即线程1还未释放Lock，有可能线程2抢到GI

```
既然是串行，那我们执行

t1.start()

t1.join

t2.start()

t2.join()

这也是串行执行啊，为何还要加Lock呢，需知join是等待t1所有的代码执行完，相当于锁住了t1的所有代码，而Lock只是锁住一部
```

详解：

```
因为Python解释器帮你自动定期进行内存回收，你可以理解为python解释器里有一个独立的线程，每过一段时间它起wake up做一次全
```

看一段代码：解释为什么要加锁，如果下面代码中work函数里面的那个time.sleep(0.005)，我的电脑用的这个时间片段，每次运行都呈现不同的结果，我们可以改改时间试一下。

```
from threading import Thread,Lock
import os,time
def work():
    global n
    # lock.acquire() #加锁
    temp=n
    time.sleep(0.1) #一会将下面循环的数据加大并且这里的时间改的更小试试
    n=temp-1
    # time.sleep(0.02)
    # n = n - 1
    '''如果这样写的话看不出来效果，因为这样写就相当于直接将n的指向改了，就好比从10，经过1次减1之后，n就直接指向了9，速度
    # lock.release()
if __name__ == '__main__':
    lock=Lock()
    n=100
    l=[]
    # for i in range(10000):  #如果这里变成了10000，你在运行一下看看结果
    for i in range(100):  #如果这里变成了10000，你在运行一下看看结果
        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
```

上面这个代码示例，如果循环次数变成了10000，在我的电脑上就会出现不同的结果，因为在线程切换的那个time.sleep的时间内，有些线程还没有被切换到，也就是有些线程还没有拿到n的值，所以计算结果就没准了。

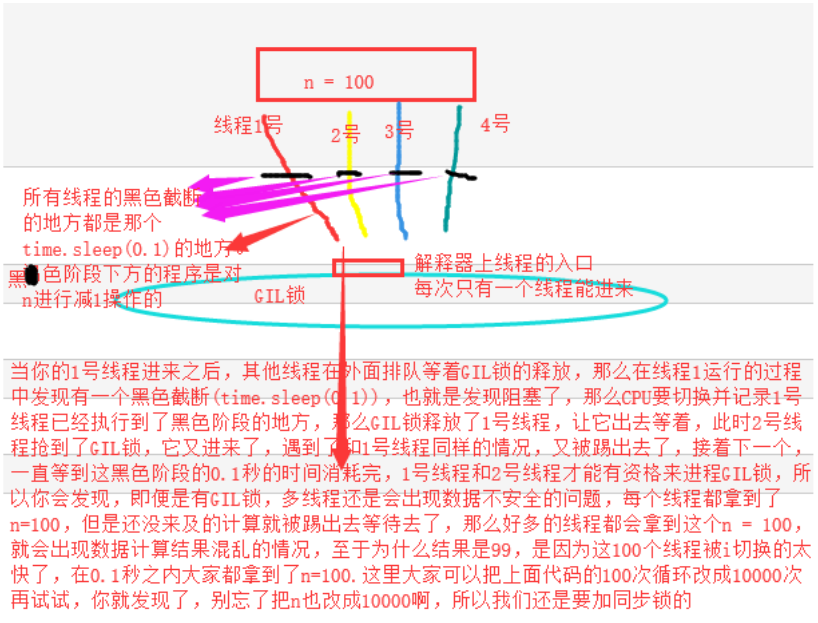
锁通常被用来实现对共享资源的同步访问。为每一个共享资源创建一个 Lock**对象，当你需要访问该资源时，调用**acquire**方法来获取锁对象（如果其它线程已经获得了该锁，则当前线程需等待其被释放），待资源访问完后，再调用**release**方法释放锁：**

```
import threading

R=threading.Lock()

R.acquire() ##R.acquire()如果这里还有一个acquire，你会发现，程序就阻塞在这里了，因为上面的锁已经被拿到了并且还没有释放的
'''
对公共数据的操作
'''
R.release()
```

通过上面的代码示例1，我们看到多个线程抢占资源的情况，可以通过加锁来解决，看代码：

```
from threading import Thread,Lock
import os,time
def work():
    global n
    lock.acquire() #加锁
    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
```

看上面代码的图形解释：



分析：
    #1.100个线程去抢GIL锁，即抢执行权限
    #2. 肯定有一个线程先抢到GIL（暂且称为线程1），然后开始执行，一旦执行就会拿到lock.acquire()
    #3. 极有可能线程1还未运行完毕，就有另外一个线程2抢到GIL，然后开始运行，但线程2发现互斥锁lock还未被线程1释放，于是阻
    #4.直到线程1重新抢到GIL，开始从上次暂停的位置继续执行，直到正常释放互斥锁lock，然后其他的线程再重复2 3 4的过程

```
#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
'''
```

3.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，进程的死锁和线程的是一样的，而且一般情况下进程之间是数据不共享的，不需要加锁，由于线程是对全局的数据共享的，所以对于全局的数据进行操作的时候，要加锁。

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()
```

```
from threading import Thread,Lock
import time
mutexA=Lock()
mutexB=Lock()

class MyThread(Thread):
    def run(self):
        self.func1()
        self.func2()
    def func1(self):
        mutexA.acquire()
        print('\033[41m%s 拿到A锁>>>\033[0m' %self.name)
        mutexB.acquire()
        print('\033[42m%s 拿到B锁>>>\033[0m' %self.name)
        mutexB.release()
        mutexA.release()

    def func2(self):
        mutexB.acquire()
        print('\033[43m%s 拿到B锁???\033[0m' %self.name)
        time.sleep(2)
        #分析：当线程1执行完func1，然后执行到这里的时候，拿到了B锁，线程2执行func1的时候拿到了A锁，那么线程2还要继续执行

        mutexA.acquire()
        print('\033[44m%s 拿到A锁???\033[0m' %self.name)
        mutexA.release()

        mutexB.release()

if __name__ == '__main__':
    for i in range(10):
        t=MyThread()
        t.start()

'''
Thread-1 拿到A锁>>>
Thread-1 拿到B锁>>>
Thread-1 拿到B锁???
```

```
Thread-2 拿到A锁>>>
然后就卡住，死锁了
'''
```

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：

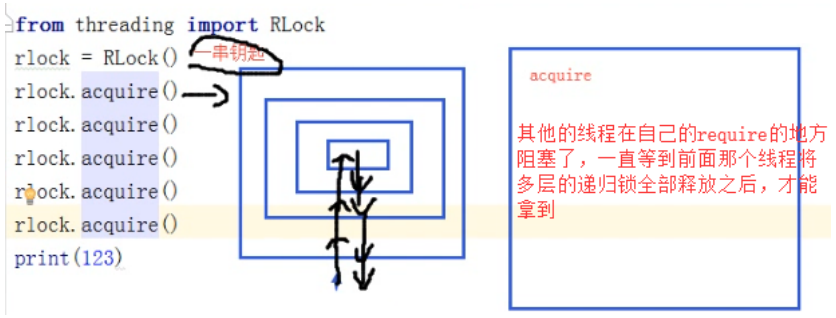
```
from threading import RLock as Lock
import time
mutexA=Lock()
mutexA.acquire()
mutexA.acquire()
print(123)
mutexA.release()
mutexA.release()
```

典型问题：科学家吃面 ， 看下面代码示例：

```
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 ['taibai','egon','wulaoban']:
    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 ['taibai','wulaoban']:
    t1 = Thread(target=eat1,args=(name,))
    t1.start()
for name in ['alex','peiqi']:
    t2 = Thread(target=eat2,args=(name,))
    t2.start()
```

递归锁大致描述：　　当我们的程序中需要两把锁的时候，你就要注意，别出现死锁，最好就去用递归锁。



十 信号量

同进程的一样

Semaphore管理一个内置的计数器，
每当调用acquire()时内置计数器-1；
调用release() 时内置计数器+1；
计数器不能小于0；当计数器为0时，acquire()将阻塞线程直到其他线程调用release()。

实例：(同时只有5个线程可以获得semaphore,即可以限制最大连接数为5)：

示例代码

大家还记得信号量和进程池的区别吗，线程也有线程池，和信号量也是那点区别

互斥锁与信号量推荐博客：http://url.cn/5DMsS9r

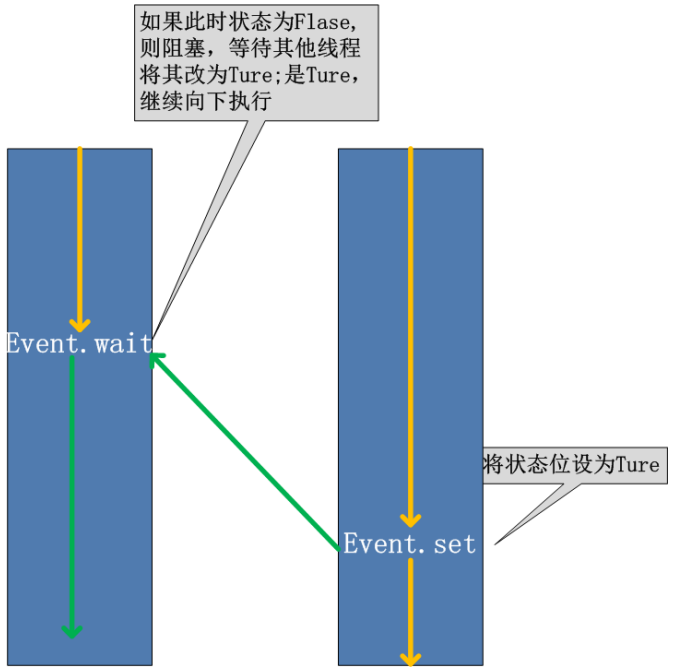
十一 事件

同进程的一样

线程的一个关键特性是每个线程都是独立运行且状态不可预测。如果程序中的其 他线程需要通过判断某个线程的状态来确定自己下一步的操作,这时线程同步问题就会变得非常棘手。为了解决这些问题,我们需要使用threading库中的Event对象。 对象包含一个可由线程设置的信号标志,它允许线程等待某些事件的发生。在 初始情况下,Event对象中的信号标志被设置为假。如果有线程等待一个Event对象, 而这个Event对象的标志为假,那么这个线程将会被一直阻塞直至该标志为真。一个线程如果将一个Event对象的信号标志设置为真,它将唤醒所有等待这个Event对象的线程。如果一个线程等待一个已经被设置为真的Event对象,那么它将忽略这个事件, 继续执行

事件的基本方法：

```
event.isSet()：返回event的状态值；
event.wait()：如果 event.isSet()==False将阻塞线程；
event.set()： 设置event的状态值为True，所有阻塞池的线程激活进入就绪状态， 等待操作系统调度；
event.clear()：恢复event的状态值为False。
```



还记得我们进程那里的事件用的什么例子吗，是不是红绿灯啊，这次我们不讲红绿灯的例子了，换个新的！

例如，有多个工作线程尝试链接MySQL，我们想要在链接前确保MySQL服务正常才让那些工作线程去连接MySQL服务器，如果连接不成功，都会去尝试重新连接。那么我们就可以采用threading.Event机制来协调各个工作线程的连接操作

MySQL是啥呢？简单说一下：

```
mysql就是一个数据库，存数据用的东西，它就像一个文件夹，里面存着很多的excel表格，我们可以在表格里面写数据，存数据。但是
```

我们先模拟一个场景：

首先起两个线程：

第一个线程的用处：连接数据库，那么我这个线程需要等待一个信号，告诉我我们之间的网络是可以连通的。

第二个线程的用处：检测与数据库之间的网络是否联通，并发送一个可联通或者不可联通的信号。

```
from threading import Thread,Event
import threading
import time,random
def conn_mysql():
    count=1
    while not event.is_set():
        if count > 3:
            raise TimeoutError('链接超时') #自己发起错误
        print('<%s>第%s次尝试链接' % (threading.current_thread().getName(), count))
        event.wait(0.5) #
        count+=1
    print('<%s>链接成功' %threading.current_thread().getName())


def check_mysql():
    print('\033[45m[%s]正在检查mysql\033[0m' % threading.current_thread().getName())
    t1 = random.randint(0,3)
    print('>>>>',t1)
    time.sleep(t1)
    event.set()
if __name__ == '__main__':
    event=Event()
    check = Thread(target=check_mysql)
    conn1=Thread(target=conn_mysql)
    conn2=Thread(target=conn_mysql)

    check.start()
    conn1.start()
    conn2.start()
```

十二 条件Condition（了解）

使得线程等待，只有满足某条件时，才释放n个线程，看一下大概怎么用就可以啦~~

```
import time
from threading import Thread,RLock,Condition,current_thread

def func1(c):
    c.acquire(False) #固定格式
    # print(1111)

    c.wait()  #等待通知,
    time.sleep(3)  #通知完成后大家是串行执行的，这也看出了锁的机制了
    print('%s执行了'%(current_thread().getName()))

    c.release()

if __name__ == '__main__':
    c = Condition()
    for i in range(5):
```

```
        t = Thread(target=func1,args=(c,))
        t.start()

    while True:
        num = int(input('请输入你要通知的线程个数:'))
        c.acquire() #固定格式
        c.notify(num)  #通知num个线程别等待了，去执行吧
        c.release()

#结果分析:
# 请输入你要通知的线程个数:3
# 请输入你要通知的线程个数:Thread-1执行了 #有时候你会发现的你结果打印在了你要输入内容的地方，这是打印的问题，没关系，不
# Thread-3执行了
# Thread-2执行了
```

十三 定时器（了解）

定时器，指定n秒后执行某个操作，这个做定时任务的时候可能会用到。


```
import time
from threading import Timer,current_thread #这里就不需要再引入Timer
import threading
def hello():
    print(current_thread().getName())
    print("hello, world")
    # time.sleep(3) #如果你的子线程的程序执行时间比较长，那么这个定时任务也会乱，当然了，主要还是看业务需求
t = Timer(10, hello)  #创建一个子线程去执行后面的函数
t.start()  # after 1 seconds, "hello, world" will be printed
# for i in range(5):
#     t = Timer(2, hello)
#     t.start()
#     time.sleep(3) #这个是创建一个t用的时间是2秒，创建出来第二个的时候，第一个已经过了两秒了，所以你的5个t的执行结果基本

print(threading.active_count())
print('主进程',current_thread().getName())
```

十四 线程队列

线程之间的通信我们列表行不行呢，当然行，那么队列和列表有什么区别呢?

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 #不需要通过threading模块里面导入，直接import queue就可以了，这是python自带的
#用法基本和我们进程multiprocess中的queue是一样的
q=queue.Queue()
```

```
q.put('first')
q.put('second')
q.put('third')
# q.put_nowait() #没有数据就报错，可以通过try来搞
print(q.get())
print(q.get())
print(q.get())
# q.get_nowait() #没有数据就报错，可以通过try来搞
'''
结果(先进先出):
first
second
third
'''
```

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

先进后出示例代码

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

优先级队列示例代码

这三种队列都是线程安全的，不会出现多个线程抢占同一个资源或数据的情况。

十五 Python标准模块--concurrent.futures

到这里就差我们的线程池没有讲了，我们用一个新的模块给大家讲，早期的时候我们没有线程池，现在python提供了一个新的标准或者说内置的模块，这个模块里面提供了新的线程池和进程池，之前我们说的进程池是在multiprocessing里面的，现在这个在这个新的模块里面，他俩用法上是一样的。

为什么要将进程池和线程池放到一起呢，是为了统一使用方式，使用threadPollExecutor和ProcessPollExecutor的方式一样，而且只要通过这个concurrent.futures导入就可以直接用他们两个了

```
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
```

```
import time
import os
import threading
from concurrent.futures import ThreadPoolExecutor,ProcessPoolExecutor

def func(n):
    time.sleep(2)
    print('%s打印的：'%(threading.get_ident()),n)
    return n*n
tpool = ThreadPoolExecutor(max_workers=5) #默认一般起线程的数据不超过CPU个数*5
# tpool = ProcessPoolExecutor(max_workers=5) #进程池的使用只需要将上面的ThreadPoolExecutor改为ProcessPoolExecutor
#异步执行
t_lst = []
for i in range(5):
    t = tpool.submit(func,i) #提交执行函数,返回一个结果对象，i作为任务函数的参数 def submit(self, fn, *args, **kwargs)：可以
    t_lst.append(t)  #
    # print(t.result())
    #这个返回的结果对象t，不能直接去拿结果，不然又变成串行了，可以理解为拿到一个号码，等所有线程的结果都出来之后，我们再
tpool.shutdown() #起到原来的close阻止新任务进来 + join的作用，等待所有的线程执行完毕
print('主线程')
for ti in t_lst:
    print('>>>>',ti.result())

# 我们还可以不用shutdown()，用下面这种方式
# while 1:
#     for n,ti in enumerate(t_lst):
#         print('>>>>', ti.result(),n)
#     time.sleep(2) #每个两秒去去一次结果，哪个有结果了，就可以取出哪一个，想表达的意思就是说不用等到所有的结果都出来再去

#结果分析： 打印的结果是没有顺序的，因为到了func函数中的sleep的时候线程会切换，谁先打印就没准儿了，但是最后的我们通过结
# 37220打印的： 0
# 32292打印的： 4
# 33444打印的： 1
# 30068打印的： 2
# 29884打印的： 3
# 主线程
# >>>> 0
# >>>> 1
# >>>> 4
# >>>> 9
# >>>> 16
```

ProcessPoolExecutor的使用：

```
只需要将这一行代码改为下面这一行就可以了，其他的代码都不用变
tpool = ThreadPoolExecutor(max_workers=5) #默认一般起线程的数据不超过CPU个数*5
# tpool = ProcessPoolExecutor(max_workers=5)

你就会发现为什么将线程池和进程池都放到这一个模块里面了，用法一样
```

```
from concurrent.futures import ThreadPoolExecutor,ProcessPoolExecutor
import threading
import os,time,random
def task(n):
    print('%s is runing' %threading.get_ident())
    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)

    s = executor.map(task,range(1,5)) #map取代了for+submit
    print([i for i in s])
```

```
import time
import os
import threading
from concurrent.futures import ThreadPoolExecutor,ProcessPoolExecutor

def func(n):
    time.sleep(2)
    return n*n

def call_back(m):
    print('结果为：%s'%(m.result()))

tpool = ThreadPoolExecutor(max_workers=5)
t_lst = []
for i in range(5):
    t = tpool.submit(func,i).add_done_callback(call_back)
```

```
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obj.result()拿到
```