本节任务：理解并发，并行，同步，异步，会写多线程的异步操作代码

并发编程（并发，并行，同步，异步）

通俗理解并发编程中的相关核心概念

核心概念：进程、线程和互斥锁

- CPU的作用
  - 计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。
  - CPU的核数（多核计算机，大部分情况下也只是用了一核cup）
    - 假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，**单核CPU一次只能运行一个任务**。这个任务是什么呢?
- **进程** 就好比工厂的车间，它代表CPU所能处理的单个任务。

- 任意时刻，CPU总是运行一个进程，其他进程处于非运行状态。
- **基于车间来聊：**
  - 一个车间里，可以有很多工人。他们协同完成一个任务。
  - **线程** 就好比车间里的工人。**一个进程可以包括多个线程**。
- 车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征**一个进程的内存空间是被线程共享的**，每个线程都可以使用这些共享内存。
- 基于进程空间可以被线程共享的角度---思考：
  - 每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。那么如何实现呢?
  - 一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫**"互斥锁"，其作用是防止多个线程同时读写某一块内存区域**。

进程

什么是进程

- 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。
- 在操作系统中，每启动一个应用程序其实就是OS开启了一个进程且为进程分类对应的内存/资源，应用程序的执行也就是进程在执行。
- 狭义定义：一个正在运行的应用程序在操作系统中被视为一个进程
- 举例： 我们有py1文件中和py2文件，两个文件运行起来后是两个进程。

进程调度

- 提问：
    - 进程就是计算机中正在运行的一个程序或者软件，并且在上述工厂案例中，我们说单个CPU一次只能运行一个任务，那么你有没有在电脑上一边聊微信一边听音乐一边打游戏的场景啊？why?
        - 是因为CPU在交替运行多个进程。

- 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，这个调度也不是随机进行的，而是需要遵循一定的法则，由此就有了进程的调度算法。
    - 目前已实现的调度算法有：先来先服务（FCFS）调度算法、短作业优先调度算法和时间片轮转法。不过被公认的一种比较好的进程调度算法是"时间片轮转法"。

"时间片轮转法"调度算法的实施过程如下所述。
(1) os会创建多个就绪队列存储进程，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，以此类推。并且该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先级愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍
(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，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成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地排队等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列。
(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os正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的服务，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

并发与并行

- 通过进程之间的调度，也就是进程之间的切换，我们用户感知到的好像是两个视频文件同时在播放，或者音乐和游戏同时在进行，那就让我们来看一下什么叫做并发和并行。
- 无论是并行还是并发，在用户看来都是'同时'运行的，不管是进程还是线程，都只是一个任务而已，真实干活的是cpu，而一个cpu同一时刻只能执行一个任务。
- **并行**：同时运行，只有具备多个cpu才能实现并行
- **并发**：是伪并行，即看起来是同时运行。

举例说明

你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也不支持并行。
你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。
你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。

总结

并发的关键是你有处理多个任务的能力，不一定要同时。

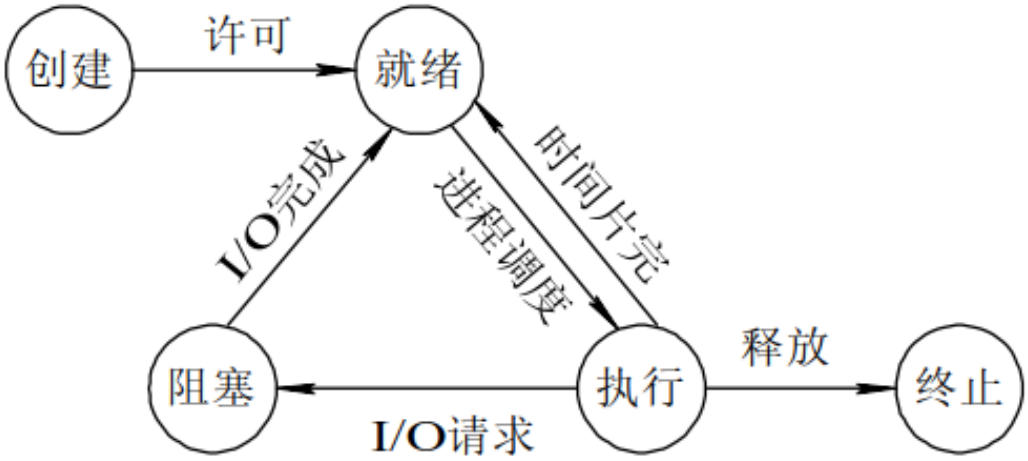
并行的关键是你有同时处理多个任务的能力。

所以它们最关键的点就是：是否是『同时』。

进程的状态

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、阻塞和终止。

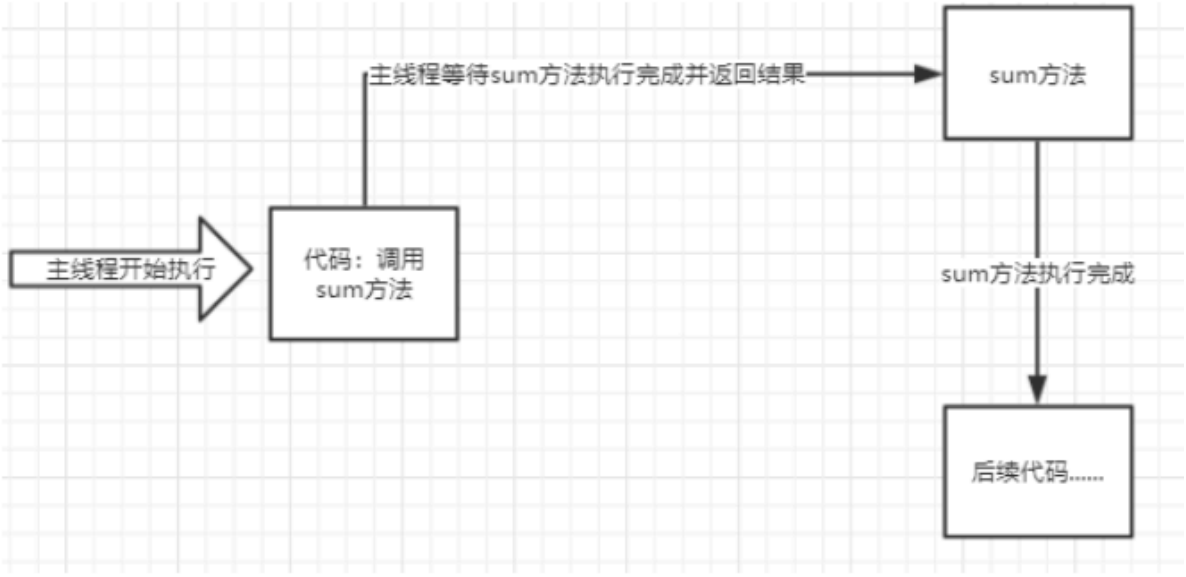
- 就绪(Ready)状态
  - 进程已经准备好，已分配到所需资源/内存。
- 执行/运行（Running）状态
  - 进程处于就绪状态被调度后，进程进入执行状态
- 阻塞(Blocked)状态（耗时操作）
  - 正在执行的进程由于某些事件（I/O请求,input,申请缓存区失败）而暂时无法运行，进程受到阻塞,则进入就绪状态等待系统调用
    - 网络请求，input等
- 终止状态
  - 进程结束，或出现错误，或被系统终止，进入终止状态。无法再执行



同步和异步

- 举个例子来说，你去商场买手机的时候正好口渴了。
  - **同步**的意思就是说，你和店员说你看上这部手机了，然后店员去仓库拿货（买手机任务阻塞），你在店里等待店员回来后再去买水喝（阻塞结束再干喝水的事）。
  - **异步**呢，异步的意思就是在店员去仓库拿货的时候买手机任务阻塞），你趁机去买水喝，然后喝完水后，刚好店员也带着你的新手机回来了。（利用阻塞的时间干后面的事）
- 使用方法的调用来举例：
  - **同步方法**调用一旦开始，调用者必须等到方法调用返回后，才能继续该方法后续的行为代码。

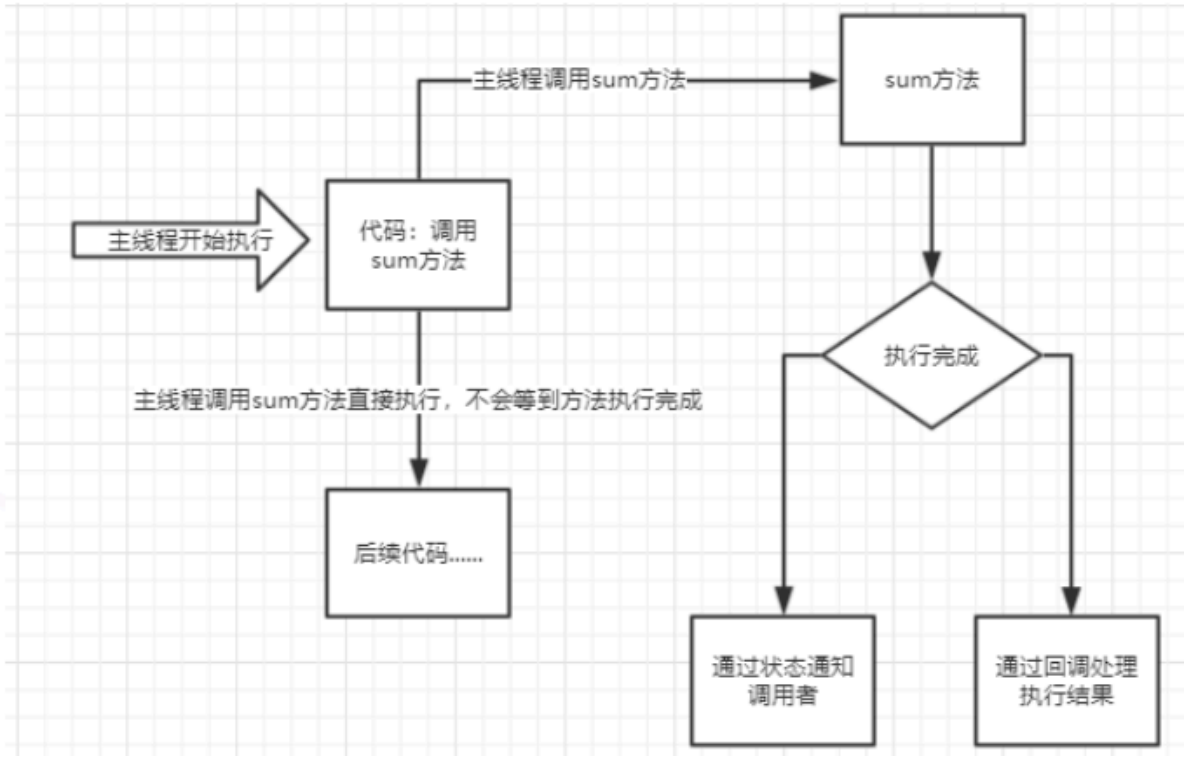
同步执行



同步执行当调用方法执行完成后并返回结果，才能执行后续代码

- **异步方法**调用更像一个消息传递，一旦调用开始，该方法调用就会立即返回，调用者就可以继续后续的操作。而异步方法通常会在另外一个线程/进程中，“真实”地执行着。整个过程，不会阻碍调用者的工作(让另一个人在等店员取货回来，你去买水喝)

异步执行



注意：同步和异步针对是cup遇到阻塞操作时，所产生的不同行为！

思考：异步操作是基于并行的还是基于并发的?

- 异步可以是基于并行的也可以是基于并发的，但是大部分情况下是基于并发的。
    - 基于并发是指，在任务开始执行的时候，发生了阻塞则启动另一个进程/线程执行方法后序的操作，而当前的进程/线程执行该方法任务阻塞的操作。则当前进程/线程和启动的另一个新的进程/线程是基于cpu的调度算法，调度执行的。
    - 基于并行是指，在多核情况下，如果应用程序执行过程中设计到的计算量特别大，则相关的运算操作启动的进程/线程会在另一个cup中启动，这样可以实现真正的并行。

下面我们就一起来学习，可以实现异步的具体操作：进程、线程和协程！

Python进程的实现

multiprocessing包

multiprocess是python中管理进程的包。 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，提供的子模块非常多。

Process模块

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。

之前我们说过，运行一个py文件就相当于启动了一个进程，这个进程我们成为**"主进程"**

而在主进程对应的py文件中，可以通过Process模块创建另一个进程，这个进程是基于主进程创建的，因此可以被称为**"子进程"**

当有了两个进程后，我们其实就可以实现**异步机制**了！

具体实现过程：

1.导入模块：from multiprocessing import Process

2.基于Process创建一个子进程对象(当前运行的整个py文件表示主进程)，然后可以基于target参数将外部的一个函数注册到该子进程中

3.基于start()方法启动创建好的子进程

```
from multiprocessing import Process
def func():
    print('我是绑定给子进程的一组任务！')

if __name__ == '__main__':
    print('主进程开始执行！')
    #创建一个进程p，给该进程绑定一组任务
    p = Process(target=func)
    #启动创建好的进程
    p.start()

    print('主进程执行结束！')
```

- 如何手动给注册在子线程中的函数传递指定的参数?
  - 通过args传递参数

```
from multiprocessing import Process
def func(num1,num2):
    print('我是绑定给子进程的一组任
务！',num1,num2)

if __name__ == '__main__':
    print('主进程开始执行！')
    #创建一个进程p，给该进程绑定一组任务
    p = Process(target=func,args=(123,456))
    #启动创建好的进程
    p.start()

    print('主进程执行结束！')
```

- 使用进程实现异步效果：
  - 同步效果：

```
import time
def get_request(url):
    print('正在请求网址的数据: ',url)
    time.sleep(2)
    print('请求结束:',url)

if __name__ == "__main__":
    start = time.time()
    urls =
['www.1.com','www.2.com','www.3.com']
    for url in urls:
        get_request(url)
    print('总耗时: ',time.time()-start)
```

- 异步效果:

```
import time
from multiprocessing import Process
def get_request(url):
    print('正在请求网址的数据: ',url)
    time.sleep(2)
    print('请求结束:',url)

if __name__ == "__main__":
    urls =
['www.1.com','www.2.com','www.3.com']
    for url in urls:
        #创建了三个进程，表示三组任务
        p =
Process(target=get_request,args=(url,))
        p.start()
```

- join方法的使用（了解）
  - join是需要让子进程调用的，主进程一定会等待调用了join的子进程结束后，主进程在结束！

```
import time
from multiprocessing import Process
def get_request(url):
    print('正在请求网址的数据: ',url)
    time.sleep(2)
    print('请求结束:',url)
```

```
if __name__ == "__main__":
    start = time.time()
    urls =
['www.1.com','www.2.com','www.3.com']
    p_list = [] #存储创建好的子进程
    for url in urls:
        #创建子进程
        p = Process(target=get_request,args=
(url,))
        p_list.append(p)
        #p.join() #一定不要这么写
        #启动子进程
        p.start()
    for pp in p_list:#pp就是列表中的每一个子进程
        pp.join() #是的每一个子进程都执行了join操
作
        #意味着：主进程需要等待所有执行了join操作的
子进程结束后再结束


    print('总耗时：',time.time()-start)
```

```
#观察下述代码出现的问题是什么？（了解）
from multiprocessing import Process
import time
```

```
ticketNum = 10 #全部的车票
def func(num):
    print('我是子进程，我要购买%d张票！'%num)
    global ticketNum
    ticketNum -= num
    time.sleep(2)

if __name__ == '__main__':
    p = Process(target=func,args=(3,))
    p.start()
    #主进程在子进程结束之后在结束
    p.join() #只有当子进程结束后，join的调用结束，才会执行join后续的操作
    print('目前剩余车票数量为:',ticketNum) #输出结果依然是10
    #进程和进程之间是完全独立。两个进程对应的是两块独立的内存空间，每一个进程只可以访问自己内存空间里的数据。
```

- 如果主进程的查询结果是在2s中后才出现的，则join生效了。但是查询结果为什么是这样的呢?
  - 首先，ticketNum = 10这个变量是存在于主进程中的，然后再func函数中ticketNum则是将全局变量ticketNum的值拷贝到了子进程中的ticketNum变量中，因此在func中的减法操作只能作用在子进程的变量中。最终，最后一行主进程打印的ticketNum则是

原来主进程未发生变量的值。

- 如何解决？(自己可以尝试文件共享)
  - 进程通信机制，管道，信号量等(没必要掌握，日后用不到)
- 继续思考：一个子进程函数的返回值如何被主进程获取?
- **总结：进程之间的数据是隔离的，也就是数据不共享**

守护进程（了解）

那么如果有一天我们的需求是我的主进程结束了，由我主进程创建的那些子进程必须跟着结束，怎么办？守护进程就来了！

主进程创建守护进程后：

其一：守护进程会在主进程代码执行结束后就终止

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

注意：主进程代码运行结束，守护进程随即终止

```
import time
from multiprocessing import Process
def get_request(url):
    print('正在请求网址的数据：',url)
```

```
    time.sleep(2)
    print('请求结束:',url)

if __name__ == "__main__":
    start = time.time()
    p = Process(target=get_request,args=
('www.1.com',))
    # 将当前p这个子进程设置为了守护进程
    p.daemon = True #该操作必须放置在子进程启动操作之
前
    p.start()

    print('主进程执行结束')
```

进程同步(锁)(了解)

通过刚刚的学习，我们千方百计实现了程序的异步，让多个任务可以同时在几个进程中并发处理，但是它们之间的运行没有顺序，一旦开启也不受我们控制。

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

def work(n):
    print('%s: %s is running' %(n,os.getpid()))
```

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

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

尽管并发编程让我们能更加充分的利用计算机的资源，但是也给我们带来了新的问题：进程之间数据不共享,但是共享同一套文件系统,所以访问同一个文件是没有问题的，要是对同一文件进行读写操作呢？要知道共享带来的是竞争，竞争带来的结果就是错乱，如何控制，就是加锁处理。

加锁流程：

1.导包：from multiprocessing import Lock

2.加锁：lock.acquire()

3.解锁：lock.release()

接下来，我们以模拟抢票为例，来看看数据安全的重要性。

```
#在当前目录下创建一个文件（db）
#文件db的内容：{"count":1}表示的是余票数量
from multiprocessing import Process
import time,json,random
def search():#查询db文件中的余票数量
```

```
    fp = open('./db.txt','r')
    dic = json.load(fp) #反序列化，将文件中的json数据转成python字典对象
    print('剩余车票数量为:',dic['count'])

def get(): #负责抢票，一次只能购买一张票
    fp = open('./db.txt', 'r')
    dic = json.load(fp)
    time.sleep(0.1)
    if dic['count'] > 0:#还有剩余车票
        time.sleep(0.2)
        dic['count'] -= 1 #一次只能购买一张票
        time.sleep(0.1)
        #购买车票后，余票数量发生了变化，将最新的余票数量在写回到db文件中进行存储
        json.dump(dic,open('./db.txt','w'))
        print('购票成功! ')
def task():
    search() #先查询
    get() #后购买

if __name__ == '__main__':
    for i in range(3):#创建三个子进程
        p = Process(target=task)
        p.start()
```

加锁后：

```
from multiprocessing import Process
import time,json,random
from multiprocessing import Lock #进程锁
#mac下添加下两行代码，其他系统不需要
import multiprocessing
multiprocessing.set_start_method('fork')

def search():#查询db文件中的余票数量
    fp = open('./db.txt','r')
    dic = json.load(fp) #反序列化，将文件中的json数据转成python字典对象
    print('剩余车票数量为:',dic['count'])

def get(): #负责抢票，一次只能购买一张票
    fp = open('./db.txt', 'r')
    dic = json.load(fp)
    time.sleep(0.1)
    if dic['count'] > 0:#还有剩余车票
        time.sleep(0.2)
        dic['count'] -= 1 #一次只能购买一张票
        time.sleep(0.1)
        #购买车票后，余票数量发生了变化，将最新的余票数量在写回到db文件中进行存储
        json.dump(dic,open('./db.txt','w'))
        print('购票成功! ')
def task(lock):
    lock.acquire() #上锁
```

```
    search() #先查询
    get() #后购买
    lock.release() #解锁

if __name__ == '__main__':
    lock = Lock() #创建了一把进程锁
    for i in range(3):#创建三个子进程
        p = Process(target=task,args=(lock,))
        p.start()
```

- 注意：加锁的代价就是使得程序失去了异步效果！

线程

基本概念

线程：线程是操作系统能够进行运算调度的最小单位（车间里的工人），它被包含在进程之中，线程是进程中的实际运作单位。一个进程中可以并发多个线程，每条线程并行执行不同的任务。

注意：

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

3.每一个进程中至少会包含有一个线程，该线程叫做"主线程"

思考：多线程可以实现并行吗?

2. 在CPU资源比较充足的时候，一个进程内的多线程，可以被分配到不同的CPU资源，这就是通过多线程实现并行。
3. 至于多线程实现的是并发还是并行？上面所说，所写多线程可能被分配到一个CPU内核中执行，也可能被分配到不同CPU执行，分配过程是操作系统所为，不可人为控制。所有，如果有人问我我所写的多线程是并发还是并行的？我会说，都有可能。
4. 不管并发还是并行，都提高了程序对CPU资源的利用率，最大限度地利用CPU资源。

Python实现线程

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。

由于更高级别的threading模块更为先进，对线程的支持更为完善，因此推荐大家使用该模块！

Threading模块

- 我们先简单应用一下threading模块来看看并发效果：

```
from threading import Thread
def func(num):
    print('num的值是:',num)

if __name__ == '__main__':
    #创建好了一个子线程（在主线程中创建）
    t = Thread(target=func,args=(1,))
    t.start()
```

```
from threading import Thread
import time
def func(num):
    time.sleep(1)
    print('num的值是:',num)

if __name__ == '__main__':
    for i in range(3):
        #创建好了一个子线程（在主线程中创建）
        t = Thread(target=func,args=(1,))
        t.start()
```

- join()方法

```
from threading import Thread
import time
class MyThread(Thread):
    def __init__(self):
        super().__init__()

    def run(self):
        print('当前子线程正在执行')
        time.sleep(2)
        print('当前子线程执行结束')

if __name__ == '__main__':
    start = time.time()

    ts = []
    for i in range(3):
        t = MyThread() #创建线程对象
        t.start() #启动线程对象
        ts.append(t)
    for t in ts:
        t.join()
    print('总耗时:',time.time()-start)
```

- 线程内存数据共享：

```
from threading import Thread
import time
def work():
    global n
    n = 0 #将全局变量修改为了0
if __name__ == '__main__':
    n = 1 #全局变量
    t = Thread(target=work)
    t.start()
    print(n) #在进程中输出全局变量的值就是线程修改后的结果为0
```

- 守护线程
  - 无论是进程还是线程，都遵循：守护xx会在主xx运行完毕后被销毁，不管守护xx时候被执行结束。

```
from threading import Thread
import time
def work():
    time.sleep(1)
    print('子线程正在执行! ')
if __name__ == '__main__':
    t = Thread(target=work)
    t.daemon = True #当前的子线程设置为了守护线
程
    t.start()
    print('主线程结束! ')
```

- 多线程使用:

```
from threading import Thread
import time

def run():
    print('当前子线程正在执行')
    time.sleep(2)
    print('当前子线程执行结束')

if __name__ == '__main__':
    start = time.time()

    ts = []
    for i in range(3):
```

```
        t = Thread(target=run)
        t.start() #启动线程对象
        ts.append(t)
    for t in ts:
        t.join()
    print('总耗时:',time.time()-start)
```

线程的GIL锁（大致了解）

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技术的厉害之处。

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，并发执行也会有这个数据安全的问题。如何解决呢?

但是python最早期的时候对于多线程也加锁，但是python比较极端的加了一个GIL全局解释锁，锁的是整个线程，而不是线程里面的某些数据操作，也就是说每次只能有一个线程使用cpu，也就说多线程用不了多核实现并行。

但是这个并不是python语言的问题，是CPython解释器的特性，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。

GIL介绍

在同一个进程中只有一个线程可以获取cpu的使用权限，那么其他的线程就必须等待该线程的cpu使用权消失后才能使用cpu,即使多个线程直接不会相互影响，在同一个进程下也只有一个线程使用cpu，这样的机制称为全局解释器锁（GIL）。每一个 Python 线程，在 CPython 解释器中执行时，都会先锁住自己的线程，阻止别的线程执行。

GIL的优点：

1、避免了大量的加锁解锁的繁琐操作

2、使数据更加安全，解决多线程间的数据完整性和状态同步

缺点：

多核处理器的效果退化成单核处理器，只能并发不能并行

多线程和多进程的使用选择

计算密集型案例：多进程效率高

os.cpu_count()：查看计算机的核数

```
from multiprocessing import Process
from threading import Thread
import time,os
def word(): #计算密集型任务
    res = 1
    for i in range(1000000):
        res *= i

if __name__ == '__main__':
    print('当前计算机的核数为：',os.cpu_count())
    start = time.time()
    ps = []
    for i in range(100):
        #使用多进程来处理计算密集型任务（利用多核优势【并行】）
        p = Process(target=word) #执行耗时1s
        # p = Thread(target=word) #执行耗时2s

        p.start()
        ps.append(p)
    for p in ps:
        p.join()
```

```
    print('总耗时: ',time.time()-start)
```

密集IO型：多线程效率高

```
from multiprocessing import Process
from threading import Thread
import time,os
def word(): #IO密集型任务
    time.sleep(2)
    print('--------------------')

if __name__ == '__main__':
    print('当前计算机的核数为: ',os.cpu_count())
    start = time.time()
    ps = []
    for i in range(999):
        # p = Process(target=word) #执行耗时11s
        p = Thread(target=word) #执行耗时2s

        p.start()
        ps.append(p)
    for p in ps:
        p.join()
    print('总耗时: ',time.time()-start)
```

- 计算密集型任务使用多进程，因为可以享用多核的优势

- io密集型任务使用多线程

同步锁

```
#未上锁
from threading import Thread,Lock
import time,random
def work():
    global n
    temp = n
    time.sleep(random.random())
    n = temp - 1

if __name__ == '__main__':
    n = 10 #全局变量
    ts = []
    for i in range(10):
        t = Thread(target=work)
        t.start()
        ts.append(t)
    for t in ts:
        t.join()

    print('全局变量n的值为：',n)
```

```
#上锁
from threading import Thread,Lock
```

```
import time,random
def work():
    global n
    lock.acquire() #上锁
    temp = n
    time.sleep(random.random())
    n = temp - 1
    lock.release() #解锁

if __name__ == '__main__':
    n = 10 #全局变量
    ts = []
    lock = Lock()
    for i in range(10):
        t = Thread(target=work)
        t.start()
        ts.append(t)
    for t in ts:
        t.join()

    print('全局变量n的值为：',n)
```

```
#下述代码必须会

from multiprocessing.dummy import Pool #导入了线程池模块
```

```
import time
from threading import Thread
start = time.time()
def get_requests(url):
    print('正在爬取数据')
    time.sleep(2)
    print('数据爬取结束')

urls =
['www.1.com','www.2.com','www.3.com','www.4.com'
,'www.5.com']
ts = []
for url in urls:
    t = Thread(target=get_requests,args=(url,))
    t.start()
    ts.append(t)
for t in ts:
    t.join()

print('总耗时:',time.time()-start)
```

线程池

线程预先被创建并放入线程池中，同时处理完当前任务之后并不销毁而是被安排处理下一个任务，因此能够避免多次创建线程，从而节省线程创建和销毁的开销，能带来更好的性能和系统稳定性。

```
from multiprocessing.dummy import Pool #导入了线程池模块
import time
urls =
['www.1.com','www.2.com','www.3.com','www.4.com','www.5.com']
def get_reqeust(url):
    print('正在请求数据: ',url)
    time.sleep(2)
    print('请求结束:',url)
start = time.time()
#创建一个线程池,开启了5个线程
pool = Pool(5)
#可以利用线程池中三个线程不断的去处理5个任务
pool.map(get_reqeust,urls)
#get_reqeust函数调用的次数取决urls列表元素的个数
#get_requests每次执行都会接收urls列表中的一个元素作为参数

print('总耗时: ',time.time()-start)
```

```
pool.close() #释放线程池
```

协程（重要！）

- 我们知道，无论是多进程还是多线程，在遇到IO阻塞时都会被操作系统强行剥夺走CPU的执行权限(使得cup执行其他操作，其他操作可能是我们程序的其他部分，也可能是其他的应用程序)，我们自己程序的执行效率因此就降低了下来。
- 解决这一问题的关键在于：
  - 我们自己从自己的应用程序级别检测到IO阻塞，然后使得cpu切换到我们自己程序的其他部分/任务执行（这里的任务指的是当前我们自己程序表示的进程或线程中的某一组操作/子程序），这样可以把我们程序的IO阻塞降到最低，我们的程序处于就绪态就会增多，以此来迷惑操作系统，操作系统便以为我们的程序是IO阻塞比较少的程序，从而会尽可能多的分配CPU给我们，这样也就达到了提升程序执行效率的目的。
    - 通俗理解：
      - 一个线程/进程可以表示一组指定行为的操作，这个操作可以由多个执行步骤组成，这些执行步骤有的是阻塞操作有的非阻塞操作，那么，当cpu执行当前进程/线程的时候遇到了阻塞的执行步骤的时候，如果不对其处理，则包含当前执行步骤的进程/线程就会被挂起进入到阻塞状态，且交出cpu的使用权

（cpu就被别人抢走了）。那么如果遇到阻塞的执行步骤，我们的程序可以检测出它是阻塞的，且可以将cpu切换到我们自己程序其他非阻塞的执行步骤时，则包含这些执行步骤的进程/线程就不会进入到阻塞状态，从而减少进程/线程的阻塞状态，增加就绪状态（牢牢抢占cup）极大幅度提升程序执行的效率。
        - 因此，有了协程后，在单进程或者单线程的模式下，就可以大幅度提升程序的运行效率了！
- 在python3.5之后新增了asyncio模块，可以帮我们检测IO（只能是网络IO【HTTP连接就是网络IO操作】），实现应用程序级别的切换（**异步IO**）。
- 接下来让我们来了解下协程的实现，从 Python 3.4 开始，Python 中加入了协程的概念，但这个版本的协程还是以生成器对象为基础的，在 Python 3.5 则增加了 asyncio，使得协程的实现更加方便。首先我们需要了解下面几个概念：
    - 特殊函数：
        - 在函数定义前添加一个async关键字，则该函数就变为了一个特殊的函数！
        - 特殊函数的特殊之处是什么？
            - 1.特殊函数被调用后，函数内部的程序语句（函数体）没有被立即执行
            - 2.特殊函数被调用后，会返回一个协程对象

- 协程：
  - 协程对象，特殊函数调用后就可以返回/创建了一个协程对象。
  - 协程对象 == 特殊的函数 == 一组指定形式的操作
    - 协程对象 == 一组指定形式的操作
- 任务：
  - 任务对象就是一个高级的协程对象。高级之处，后面讲，不着急！
  - 任务对象 == 协程对象 == 一组指定形式的操作
    - 任务对象 == 一组指定形式的操作
- 事件循环：
  - 事件循环对象（Event Loop）,可以将其当做是一个容器，该容器是用来装载任务对象的。所以说，让创建好了一个或多个任务对象后，下一步就需要将任务对象全部装载在事件循环对象中。
  - 思考：为什么需要将任务对象装载在事件循环对象?
    - 当将任务对象装载在事件循环中后，启动事件循环对象，则其内部装载的任务对象对应的相关操作就会被立即执行。

```
import asyncio
import time
#特殊的函数
```

```
async def get_request(url):
    print('正在请求的网址是:',url)
    time.sleep(2)
    print('请求网址结束! ')
    return 123
#创建了一个协程对象
c = get_request('www.1.com')
#创建任务对象
task = asyncio.ensure_future(c)
#创建事件循环对象
loop = asyncio.get_event_loop()
#将任务对象装载在loop对象中且启动事件循环对象
loop.run_until_complete(task)
```

- 任务对象对比协程对象的高级之处重点在于:
  - 可以给任务对象绑定一个回调函数!
  - 回调函数有什么作用?
    - 回调函数就是回头调用的函数，因此要这么理解，当任务对象被执行结束后，会立即调用给任务对象绑定的这个回调函数!

```
import asyncio
import time
#特殊的函数
async def get_request(url):
    print('正在请求的网址是:',url)
```

```
    time.sleep(2)
    print('请求网址结束！')
    return 123
#回调函数的封装:必须有一个参数
def t_callback(t):
    #参数t就是任务对象
    # print('回调函数的参数t是：',t)
    # print('我是任务对象的回调函数！')
    data = t.result()#result()函数就可以返回特殊函数内部的返回值
    print('我是任务对象的回调函数！,获取到特殊函数的返回值为:',data)
#创建协程对象
c = get_request('www.1.com')
#创建任务对象
task = asyncio.ensure_future(c)
#给任务对象绑定回调函数:add_done_callback的参数就是回调函数的名字
task.add_done_callback(t_callback)

#创建事件循环对象
loop = asyncio.get_event_loop()
loop.run_until_complete(task)
```

- 多任务的协程

```
import asyncio
```

```
import time
start = time.time()
urls = [

 'www.1.com','www.2.com','www.3.com'
]
async def get_request(url):
    print('正在请求: ',url)
    time.sleep(2)
    print('请求结束:',url)
#有了三个任务对象和一个事件循环对象
if __name__ == '__main__':
    tasks = []
    for url in urls:
        c = get_request(url)
        task =
asyncio.ensure_future(c)
        tasks.append(task)
    #将三个任务对象，添加到一个事件循环对象
中
    loop = asyncio.get_event_loop()

 loop.run_until_complete(asyncio.wai
t(tasks))

    print('总耗时:',time.time()-
start)
```

- 出现两个问题：
  - 1.没有实现异步效果
  - 2.wait()是什么意思？
- wait()函数：
  - 给任务列表中的每一个任务对象赋予一个可被挂起的权限！当cpu执行的任务对象遇到阻塞操作的时候，当前任务对象就会被挂起，则cup就可以执行其他任务对象，提高整体程序运行的效率！
  - 挂起任务对象：让当前正在被执行的任务对象交出cpu的使用权，cup就可以被其他任务组抢占和使用，从而可以执行其他任务组。
  - **注意：特殊函数内部，不可以出现不支持异步模块的代码，否则会中断整个异步效果！**
- await关键字：挂起发生阻塞操作的任务对象。在任务对象表示的操作中，凡是阻塞操作的前面都必须加上await关键字进行修饰！
- 完整的实现了，多任务的异步协程操作

```
import asyncio
import time
start = time.time()
urls = [
```

```
 'www.1.com','www.2.com','www.3.com'
]
async def get_request(url):
    print('正在请求: ',url)
    # time.sleep(2) #time模块不支持异步
    await asyncio.sleep(2)
    print('请求结束:',url)
#有了三个任务对象和一个事件循环对象
if __name__ == '__main__':
    tasks = []
    for url in urls:
        c = get_request(url)
        task =
asyncio.ensure_future(c)
        tasks.append(task)
    #将三个任务对象，添加到一个事件循环对象
中
    loop = asyncio.get_event_loop()

 loop.run_until_complete(asyncio.wai
t(tasks))

    print('总耗时:',time.time()-
start)
```

- 真正的将多任务的异步协程作用在爬虫中
  - 需求：爬取自己服务器中的页面数据，并将其进行数据解析操作
  - aiohttp:是一个基于网络请求的模块，功能和requests相似，但是，requests是不支持异步的，而aiohttp是支持异步的模块。
    - 环境安装：pip install aiohttp
    - 具体用法：
      - 1.先写大致加购

```
    with
aiohttp.ClientSession() as
sess:
        #基于请求对象发起请求
        #此处的get是发起get请求,
常用参数:
url,headers,params,proxy
        #post方法发起post请求，常
用参数：url,headers,data,proxy
        #发现处理代理的参数和
requests不一样（注意），此处处理代
理使用proxy='http://ip:port'
         with sess.get(url=url)
as response:
            page_text =
response.text()
            #text():获取字符串形
式的响应数据
            #read()：获取二进制形
式的响应数据
            return page_text
```

- 2.在第一步的基础上补充细节
  - 在每一个with前加上async关键字
  - 在阻塞操作前加上await关键字

- 完整代码：

```
async def
get_request(url):
    #requests是不支持异步的
模块
    # response = await
requests.get(url=url)
    # page_text =
response.text
    #创建请求对象（sess）
    async with
aiohttp.ClientSession()
as sess:
        #基于请求对象发起请
求
        #此处的get是发起get
请求，常用参数：
url,headers,params,proxy
        #post方法发起post请
求，常用参数：
url,headers,data,proxy
        #发现处理代理的参数
和requests不一样（注意），此
处处理代理使用
proxy='http://ip:port'
```

```
            async with await
sess.get(url=url) as
response:
                page_text =
await response.text()
                #text():获取字
符串形式的响应数据
                #read(): 获取
二进制形式的响应数据
                return
page_text
```

- 多任务异步爬虫的完整代码实现：

```
import requests
import asyncio
import time
from lxml import etree
import aiohttp
start = time.time()
urls = [
    'http://127.0.0.1:5000/bobo',
    'http://127.0.0.1:5000/jay',
    'http://127.0.0.1:5000/tom'
]
#该任务是用来对指定url发起请求，获取响应数据
async def get_request(url):
```

```
    #requests是不支持异步的模块
    # response = await
requests.get(url=url)
    # page_text = response.text
    #创建请求对象（sess）
    async with aiohttp.ClientSession()
as sess:
        #基于请求对象发起请求
        #此处的get是发起get请求，常用参数：
url,headers,params,proxy
        #post方法发起post请求，常用参数：
url,headers,data,proxy
        #发现处理代理的参数和requests不一样
（注意），此处处理代理使用
proxy='http://ip:port'
        async with await
sess.get(url=url) as response:
            page_text = await
response.text()
            #text():获取字符串形式的响应数
据
            #read()：获取二进制形式的响应数
据
            return page_text
def parse(t):#回调函数专门用于数据解析
    #获取任务对象请求到的页面源码数据
    page_text = t.result()
```

```
    tree = etree.HTML(page_text)
    a =
tree.xpath('//a[@id="feng"]/@href')[0]
    print(a)

tasks = []
for url in urls:
    c = get_request(url)
    task = asyncio.ensure_future(c)
    task.add_done_callback(parse)
    tasks.append(task)
loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(ta
sks))

print('总耗时:',time.time()-start)
```