*"""  
 进程就是运行中的应用程序，应用程序是动态的，进程是动态的  
 每一个进程至少对应一个线程，每个进程都来完成一件任务每个线程完成进程中的一件子任务，线程就是进程内部的一个执行单元  
 每个进程都有一个独立的内存空间，而进程内的所有线程共享内存空间  
  
 不管是单核cpu还是多核cpu都是支持多任务的，他们执行0.02秒就进行切换执行其他的任务，这样反复交替的执行，让我们感觉是在同时执行，  
这种方式叫做并发（多个任务交替执行），多个任务同时执行成为并行，如果要实现并行，那就要有多核cpu并且每个核上只有一个任务（任务数小于  
cpu的核数）  
"""***"""  
自动创建与启动的进程  
import time  
time.sleep(20) #在python中这个方法是按照秒数进行计算的  
当我们在PyCharm运行文件时，PyCharm会自动创建一个名为python的进程，随着py文件运行结束，自动创建并启动的新进程也随之结束  
创建并启动子进程中的进程成为父进程  
获得进程和父进程的id  
import time, os  
print(os.getpid()) #打印当前进程的id getpid:get process id  
print(os.getppid()) #打印当前进程id的父进程的id getppid:get parent process id  
time.sleep(20) #在python中这个方法是按照秒数进行计算的  
  
使用multiprocessing类中的current\_process()获取当前进程中的对象，通过对象来访问id  
import time, os  
import multiprocessing  
print(multiprocessing.current\_process().pid) #multiprocessing.current\_process()获得当前进程的实例对象，然后在获取pid  
print(os.getpid()) #打印当前进程的id getpid:get process id  
print(os. ()) #打印当前进程id的父进程的id getppid:get parent process id  
time.sleep(20)  
运行结果：  
13320  
13320  
7088  
"""  
"""  
标准库模块提供的类对象multiprocessing提供的Process用于表示进程  
第一种方式  
 使用类对象Process创建并启动进程的第一种方式  
 主要有一下几步：  
 根据类对象Process创建一个实例对象  
 调用进程实例对象的方法start()来启动一个进程  
 start()调用了Process内部的run()然后run调用了action（）方法（会在一个新创建并启用的  
代码：  
from multiprocessing import current\_process, Process  
import time  
print('父进程开始：(%d - %s)' % (current\_process().pid, current\_process().name))  
def do\_sth(args1, args2):  
 print('子进程启动：（%d - %s）' % (current\_process().pid, current\_process().name))  
 print('args1 = %d,args2 = %d' % (args1, args2))  
 print('子进程结束：（%d - %s）' % (current\_process().pid, current\_process().name))  
pc = Process(target=do\_sth, args=(1,2), name='do\_sth')  
if \_\_name\_\_ == '\_\_main\_\_':  
 pc.start()  
time.sleep(3)  
print('父进程开始：(%d - %s)' % (current\_process().pid, current\_process().name))  
运行结果：  
父进程开始：(10668 - MainProcess)  
父进程开始：(15940 - do\_sth)  
父进程开始：(10668 - MainProcess)  
父进程开始：(15940 - do\_sth)  
子进程启动：（15940 - do\_sth）  
args1:1,args22  
子进程结束：（15940 - do\_sth）  
"""  
"""  
Process的特殊方法分析  
def \_\_init\_\_(self,  
 group: Any = ...,   
 target: Optional[Callable] = ...,   
 name: Optional[str] = ...,   
 args: Iterable[Any] = ...,   
 kwargs: Mapping[Any, Any] = ...,   
 \*,  
 daemon: Optional[bool] = ...) -> None: ...  
group：用于执行进程实例对象所属的进程组，默认不属于任何进程组  
target：用于指定方法run（）调用的函数，默认没有函数被调用  
name:用于指定创建进程实例对象的名称，默认名称是Process-数字  
args:用于指定target位置参数，用元组表示，默认是空元组，默认不接受任何参数  
kwargs:用于指定target位置关键字参数，用字典表示，默认不接受任何参数  
"""  
"""  
第二种方式  
 自定义继承自Process的类对象，然后重写\_\_init\_\_和run  
 根据自定义；欸对象创建进程实例对象  
 调用特殊方法start()启动进程（会自动调用方法run()）  
   
 （相比于第一种方法，相当于是把target中的函数体转移到了方法run方法中的方法体转移到了一个方法中）  
代码：  
from multiprocessing import Process, current\_process  
import time  
print('父进程启动（%d - %s）' % (current\_process().pid, current\_process().name))  
class MyProcess(Process):  
 def \_\_init\_\_(self, name, args): #name还是要返回给父类中去修改，而接受的参数args则是赋值到self对象中  
 super().\_\_init\_\_(name = name) #注意这里必须调用父类中的\_\_init\_\_进行初始化，否否则会报错  
 self.args = args  
 def run(self):  
 print('子进程启动（%d - %s）' % (current\_process().pid, current\_process().name))  
 print('args1 = %s, args2 = %s' % (self.args[0], self.args[1]))  
 print('子进程结束（%d - %s）' % (current\_process().pid, current\_process().name))  
  
mp = MyProcess(name = 'MyProcess', args = (1,8))  
if \_\_name\_\_ == '\_\_main\_\_':  
 mp.start() #当启动时会自动调用run()方法  
time.sleep(3)  
print('父进程启动（%d - %s）' % (current\_process().pid, current\_process().name))  
运行结果：  
父进程启动（18912 - MainProcess）  
父进程启动（15080 - MyProcess）  
父进程启动（18912 - MainProcess）  
父进程启动（15080 - MyProcess）  
子进程启动（15080 - MyProcess）  
args1 = 1, args2 = 8  
子进程结束（15080 - MyProcess）  
"""  
"""  
子进程创建与启动之fork()  
fork() 注意：在Windows系统上是不能使用fork（）的  
返回值的三种情况：   
 小于0：复制子进程失败  
 0：处在子进程中  
 大于0：当前在父进程中，并且返回值就是子进程的id  
import os  
try:  
 pid = os.fork()  
except (OSError,AttributeError):  
 print('你的操作系统是不支持调用函数fork()的')  
 exit()  
  
if pid < 0:  
 print('复制子进程失败')  
elif pid == 0:  
 print('我只子进程：%d, 我的父进程是%d' % (os.getpid(), os.getppid()))  
else:  
 print('我是父进程%d, 我的子进程是%d' % (os.getpid(), pid))  
"""  
"""  
自动创建并启动的父线程  
任何进程都会创建并启动一个线程，该线程被称为父线程  
父（主）线程的默认名称是MAainThread  
代码：  
import time, threading  
#current\_thread()用于获得当前线程实例对象（自动创建并启动的父线程）  
print('自动创建并启动了父线程',threading.current\_thread().getName())  
time.sleep(20)  
运行结果：  
自动创建并启动了父线程 MainThread  
"""  
"""  
标准库模块提供的类对象threading提供的Thread用于表示线程  
第一种方式  
 使用类对象Thread创建并启动进程的第一种方式  
 主要有一下几步：  
 根据类对象Thread创建一个实例对象  
 调用进程实例对象的方法start()来启动一个进程  
 start()调用了Thread内部的run()然后run调用了action（）方法（会在一个新创建并启用的  
代码：  
from threading import Thread, current\_thread  
import time  
print('父线程启动（%s）' % (current\_thread().getName()))  
class MyThread(Thread):  
 def \_\_init\_\_(self, name, args): #name还是要返回给父类中去修改，而接受的参数args则是赋值到self对象中  
 super().\_\_init\_\_(name = name) #注意这里必须调用父类中的\_\_init\_\_进行初始化，否否则会报错  
 self.args = args  
 def run(self):  
 print('子线程启动（%s）' % (current\_thread().getName()))  
 time.sleep(20)  
 print('args1 = %s, args2 = %s' % (self.args[0], self.args[1]))  
 print('子线程结束（%s）' % (current\_**thread**().getName()))  
  
mp = MyThread(name = 'MyProcess', args = (1,8))  
mp.start() #当启动时会自动调用run()方法  
time.sleep(3)  
print('父线程启动（%s）' % (current\_thread().getName()))  
运行结果：  
父线程启动（MainThread）  
子线程启动（MyProcess）  
父线程启动（MainThread）  
args1 = 1, args2 = 8  
子线程结束（MyProcess）  
"""  
"""  
def \_\_init\_\_(self, group=None, target=None, name=None,  
 args=(), kwargs=None, \*, daemon=None):  
group：用于执行进程实例对象所属的进程组，默认不属于任何进程组  
target：用于指定方法run（）调用的函数，默认没有函数被调用  
name:用于指定创建进程实例对象的名称，默认名称是Thread-数字  
args:用于指定target位置参数，用元组表示，默认是空元组，默认不接受任何参数  
kwargs:用于指定target位置关键字参数，用字典表示，默认不接受关键字参数  
daemon：用于指定线程对象是否是守护线程，默认不是守护线程  
"""  
"""  
第二种方式  
 自定义继承自Thread的类对象，然后重写\_\_init\_\_和run  
 根据自定义；欸对象创建进程实例对象  
 调用特殊方法start()启动进程（会自动调用方法run()）  
代码：  
from threading import Thread, current\_thread  
import time  
print('父进程启动（%s）' % (current\_thread().getName()))  
class MyThread(Thread):  
 def \_\_init\_\_(self, name, args): #name还是要返回给父类中去修改，而接受的参数args则是赋值到self对象中  
 super().\_\_init\_\_(name = name) #注意这里必须调用父类中的\_\_init\_\_进行初始化，否否则会报错  
 self.args = args  
 def run(self):  
 print('子进程启动（%s）' % (current\_thread().getName()))  
 print('args1 = %s, args2 = %s' % (self.args[0], self.args[1]))  
 print('子进程结束（%s）' % (current\_thread().getName()))  
mp = MyThread(name = 'MyProcess', args = (1,8))  
mp.start() #当启动时会自动调用run()方法  
time.sleep(3)  
print('父进程启动（%s）' % (current\_thread().getName()))  
运行结果：  
父进程启动（MainThread）  
子进程启动（MyProcess）  
args1 = 1, args2 = 8  
子进程结束（MyProcess）  
父进程启动（MainThread）  
"""  
"""  
多进程执行的不确定性  
默认情况下，多个进程都是不确定的，这完全取决于系统的调度  
多进程线行的不确定性  
默认情况下，多个线程都是不确定的，这完全取决于系统的调度  
"""  
"""  
守护父进程的子进程  
可以在调用进程实例对象的方法start将属性daemon值设置为True从而将进程设置为守护进程  
守护进程是为了守护父进程而存在的子进程，当父进程结束时，那这个守护进程就没有存在的意义，所以会随着父进程的结束而立刻结束  
代码：  
from multiprocessing import Process, current\_process  
import time  
print('父进程开始：%s: %d' % (current\_process().name, current\_process().pid))  
def do\_sth():  
 print('子进程开始：%s: %d' % (current\_process().name, current\_process().pid))  
 time.sleep(2)  
 print('子进程结束：%s: %d' % (current\_process().name, current\_process().pid))  
if \_\_name\_\_ == '\_\_main\_\_':  
 pc = Process(target = do\_sth, name = 'MyProcess')  
 pc.daemon = True # 设置为守护进程  
 pc.start()  
print('父进程结束：%s: %d' % (current\_process().name, current\_process().pid))  
运行结果：  
父进程开始：MainProcess: 11224  
子进程开始：MainProcess: 11224  
子进程结束：MainProcess: 11224  
父进程开始：MainProcess: 11224  
"""  
"""  
守护父线程的子线程  
代码：  
from threading import Thread, current\_thread  
import time  
print('父线程开始：%s' % (current\_thread().getName()))  
def do\_sth():  
 print('子线程开始：%s' % (current\_thread().getName()),'\n')  
 time.sleep(2)  
 print('子线程结束：%s' % (current\_thread().getName()))  
if \_\_name\_\_ == '\_\_main\_\_':  
 th = Thread(target = do\_sth,daemon = True)  
 th.start()  
print('父线程结束：%s' % (current\_thread().getName()))  
运行结果：  
父线程开始：MainThread  
子线程开始：Thread-1  
父线程结束：MainThread   
"""  
"""  
 在父进程中创建并启动子进程后，可以调用方法join()，这样子进程会把父进程阻塞，父进程会等子进程执行完之后再从被阻塞的地方继续执行。  
代码：  
from multiprocessing import Process, current\_process  
import time  
print('父进程开始：%s: %d' % (current\_process().name, current\_process().pid))  
def do\_sth():  
 print('子进程开始：%s: %d' % (current\_process().name, current\_process().pid))  
 time.sleep(2)  
 print('子进程结束：%s: %d' % (current\_process().name, current\_process().pid))  
if \_\_name\_\_ == '\_\_main\_\_':  
 pc = Process(target = do\_sth, name = 'MyProcess')  
 pc.start()  
 pc.join(1) #会把父进程阻塞  
print('父进程结束：%s: %d' % (current\_process().name, current\_process().pid))  
"""  
"""  
在父线程中创建并启动子线程后，可以调用方法join()，这样子线程会把父进程阻塞，父线程会等子线程执行完之后再从被阻塞的地方继续执行。  
from threading import Thread, current\_thread  
import time  
print('父线程开始：%s' % (current\_thread().getName()))  
def do\_sth():  
 print('子线程开始：%s' % (current\_thread().getName()),'\n')  
 time.sleep(2)  
 print('子线程结束：%s' % (current\_thread().getName()))  
if \_\_name\_\_ == '\_\_main\_\_':  
 th = Thread(target = do\_sth,daemon = True)  
 th.start()  
 th.join()  
print('父线程结束：%s' % (current\_thread().getName()))  
"""  
  
  
  
  
  
  
  
  
  
  
"""  
进程池Pool：管理进程的数目是确定的  
 如果并发的任务数过多，一次性创建并启动大量的进程会给计算机带来很大压力，那么就可以使用进程池对创建的进程进行限制和管理  
 标准库multiprocessing中提供了一个类对象Pool，用于表示进程池，进程池中所能容纳的进程数目可以在创建Pool实例对象进行指定，如果不指定，  
那默认就是cpu的核数  
代码：  
from multiprocessing import Pool  
import time, random  
print('父进程启动')  
def do\_sth(i):  
 print('子进程%d启动' % i)  
 start = time.time()  
 time.sleep(random.random() \* 10)  
 end = time.time()  
 print('子进程%d结束，耗时%.2f秒' % (i, end - start))  
#将进程池所能容纳的最大进程数指定为3  
if \_\_name\_\_ == '\_\_main\_\_':  
 pp = Pool(3)  
for i in range(1, 11):  
 #与start类似，不同的是创建并启动的是由进程池管理的子进程  
 if \_\_name\_\_ == '\_\_main\_\_':  
 pp.apply\_async(do\_sth, args=(i,)) # apply\_async就相当于start do\_sth是位置参数而不是关键字参数  
#调用方法join()之前必须先调用方法close（）调用方法close()之后就不能让进程池在管理新的进程了  
if \_\_name\_\_ == '\_\_main\_\_':  
 pp.close()  
 # 父进程被阻塞  
 # 进程池所有的子进程完成之后，父进程再从被阻塞的地方执行  
 pp.join()  
print('父进程结束')  
运行结果：  
父进程启动  
父进程启动  
父进程结束  
父进程启动  
父进程结束  
父进程启动  
父进程结束  
子进程1启动  
子进程2启动  
子进程3启动  
子进程1结束，耗时2.84秒  
子进程4启动  
子进程3结束，耗时6.53秒  
子进程5启动  
子进程2结束，耗时9.72秒  
子进程6启动  
子进程5结束，耗时5.80秒  
子进程7启动  
子进程4结束，耗时9.56秒  
子进程8启动  
子进程7结束，耗时0.27秒  
子进程9启动  
子进程9结束，耗时1.80秒  
子进程10启动  
子进程10结束，耗时0.22秒  
子进程8结束，耗时4.78秒  
子进程6结束，耗时8.40秒  
父进程结束  
  
总结：程序运行后会同时创建并启动3个子进程，第四个子进程要等前面3个中的某一个执行结束后才会创建并启动  
"""  
"""  
线程池ThreadPool：管理进程的数目是确定的  
 如果并发的任务数过多，一次性创建并启动大量的进程会给计算机带来很大压力，那么就可以使用线程池对创建的进程进行限制和管理  
 在第三方库threadPool中提供了一个类对象ThreadPool，用于表示线程池，线程池中所能容纳的进程数目可以在创建Pool实例对象进行指定，如果不指定，  
那默认就是cpu的核数  
代码：  
from threadpool import ThreadPool, makeRequests  
import time,random  
args\_list = []  
print('父线程开始')  
for i in range(1, 11):  
 args\_list.append(i)  
tp = ThreadPool(3)  
def do\_sth(i):  
 start = time.time()  
 time.sleep(random.random() \* 10)  
 end = time.time()  
 print('子进程%s结束用时%.2f' % (i, end - start))  
#makeRequests做请求,创建需要线程池处理的任务  
request = makeRequests(do\_sth,args\_list)  
#将需要线程池处理的任务全部交个线程池，此后就会创建并启动由线程池管理的子线程  
for req in request:  
 tp.putRequest(req)  
#父线程被阻塞，线程池管理的所有子线程执行完之后，父线程再从被阻塞的地方继续执行  
tp.wait()  
print('父线程结束')  
运行结果：  
父线程开始  
子进程3结束用时8.24  
子进程2结束用时9.64  
子进程1结束用时9.98  
子进程4结束用时5.59  
子进程5结束用时4.28  
子进程7结束用时2.23  
子进程8结束用时5.12  
子进程6结束用时9.67  
子进程10结束用时1.30  
子进程9结束用时4.97  
父线程结束  
"""  
"""  
 进程池ProcessPoolExecutor  
 进程池ProcessPoolExecutor：管理进程的数目是确定的  
 标准库模块concurrent.futures中提供了一个类对象ProcessPoolExecutor，特用于表示进程池，与Pool相比ProcessPoolExecutor的功能很  
强大  
 类对象ProcessPoolExecutor遵守了上下文管理协议，所以可以使用with语句，这样在离开运行时上下文时，会自动调用方法shutdown(wait =  
True)  
那默认就是cpu的核数  
"""  
"""  
代码：  
from concurrent.futures import ProcessPoolExecutor  
import time, random  
print('父进程启动')  
def do\_sth(i):  
 print('子进程%d启动' % i)  
 start = time.time()  
 time.sleep(random.random() \* 10)  
 end = time.time()  
 print('子进程%d结束，耗时%.2f秒' % (i, end - start))  
#将进程池所能容纳的最大进程数指定为3  
ppe = ProcessPoolExecutor(max\_workers = 3)  
#将需要今晨吃处理的任务全部交给进程池，此后创建并启动由进程池管理的子进程  
for i in range(1, 11):  
 #与start类似，不同的是创建并启动的是由进程池管理的子进程  
 if \_\_name\_\_ == '\_\_main\_\_':  
 ppe.submit(do\_sth, i) # apply\_async就相当于start do\_sth是位置参数而不是关键字参数  
  
# 父进程被阻塞  
# 进程池所有的子进程完成之后，父进程再从被阻塞的地方执行  
ppe.shutdown(wait = True)  
print('父进程结束')  
"""  
"""  
代码：  
from concurrent.futures import ProcessPoolExecutor  
import time, random  
print('父进程启动')  
def do\_sth(i):  
 print('子进程%d启动' % i)  
 start = time.time()  
 time.sleep(random.random() \* 10)  
 end = time.time()  
 print('子进程%d结束，耗时%.2f秒' % (i, end - start))  
with ProcessPoolExecutor(max\_workers = 3) as ppe:  
 if \_\_name\_\_ == '\_\_main\_\_':  
 ppe.map(do\_sth, range(1, 11)) #将序列range中的所有元素都放到do\_sth中创建一个进程  
print('父进程结束')  
"""  
"""  
代码：  
from concurrent.futures import ProcessPoolExecutor  
import time, random  
print('父进程启动')  
def do\_sth(i):  
 print('子进程%d启动' % i)  
 start = time.time()  
 time.sleep(random.random() \* 10)  
 end = time.time()  
 print('子进程%d结束，耗时%.2f秒' % (i, end - start))  
with ProcessPoolExecutor(max\_workers = 3) as ppe:  
 for i in range(1,11):  
 if \_\_name\_\_ == '\_\_main\_\_':  
 ppe.submit(do\_sth, i)  
print('父进程结束')  
"""  
"""  
进程池ProcessPoolExecutor(中)  
 方法submit（）返回值是一个Fiture实例对象表示子进程所调用的那个函数的执行  
 方法result()是一个同步方法,直到函数执行完毕result()才会返回。  
"""  
"""  
代码：  
from concurrent.futures import ProcessPoolExecutor  
import time  
def do\_sth(i):  
 time.sleep(2)  
 return i \*\* i  
with ProcessPoolExecutor(max\_workers = 3) as ppe:  
 if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(1,11):  
 future = ppe.submit(do\_sth, i)  
 # result()是同步的，就是阻塞。得到的值就是do\_sth的返回值，直到do\_sth（）执行之后才会返回  
 print(future.result())  
"""  
"""  
如果还想实现同步还想获取方法do\_sth中的返回值那就可以先创建一个列表，把值存储到列表中。最后再打印  
代码：  
from concurrent.futures import ProcessPoolExecutor  
import time  
def do\_sth(i):  
 time.sleep(2)  
 return i \*\* i  
with ProcessPoolExecutor(max\_workers = 3) as ppe:  
 objs = []  
 if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(1,11):  
 future = ppe.submit(do\_sth, i)  
 # result()是异步的，就是阻塞。得到的值就是do\_sth的返回值，直到do\_sth（）执行之后才会返回  
 print(future)  
 objs.append(future)  
for i in objs:  
 print(i.result())  
"""  
"""  
标准库模块concurrent.futures还提供了两个函数  
def wait(fs, timeout=None, return\_when=ALL\_COMPLETED):  
该函数用于阻塞父进程，以等待指定的Future实例对象，直到等到所指定的条件  
 参数fs用于执行要等待的Future对象序列  
 参数timeout用于执行等待最长时间，如果不执行则为None，则一直等待  
 参数return\_when用于指定函数何时返回，有三种取值FIRST\_COMPLETED,FIRST\_EXCEPTION,ALL\_COMPLETED,分别表示，当第一个Future实例  
对象完成或已被取消时就返回，当第一个Future实例对象抛出异常时就返回，当所有的Future已经完成或者取消时就返回  
wait()返回值是一个由两个集合组成的一个元组，其中第一个集合表示已经完成或者已被取消的所有Future对象，第二个表示包含了没有完成并且被  
取消的Future对象  
"""  
"""  
代码：  
from concurrent.futures import ProcessPoolExecutor,wait,ALL\_COMPLETED,FIRST\_COMPLETED,FIRST\_EXCEPTION,as\_completed  
import time,random  
def do\_sth(i):  
 time.sleep(random.random() \* 10)  
 return i \* i  
  
ppe = ProcessPoolExecutor(max\_workers = 3)  
objs = []  
for i in range(1,5):  
 if \_\_name\_\_ == '\_\_main\_\_':  
 future = ppe.submit(do\_sth, i)  
 objs.append(future)  
#(done, not\_done) = wait(objs, return\_when=ALL\_COMPLETED) #done表示所有完成的， not\_done表示所有没有完成的  
#(done, not\_done) = wait(objs, return\_when=FIRST\_COMPLETED) #done表示所有完成的， not\_done表示所有没有完成的  
#(done, not\_done) = wait(objs, return\_when=FIRST\_EXCEPTION) #done表示所有完成的， not\_done表示所有没有完成的  
print(done)  
print(not\_done)  
as\_completed()  
"""  
"""  
 def as\_completed(fs, timeout=None):  
 参数fs用于执行要等待的Future对象序列  
 参数timeout用于指定超时时间，如果指定为None或者不指定那就不会超时  
该函数用于将指定的Future实例对象序列转换为一个迭代器对象，当序列中的任意一个Future实例对象已经完成或已被取消时都会被yield，这样，通过  
遍历得到的迭代器，就可以在任意一个Future实例对象已经完成或者取消时立即做一些处理，比如调用方法result（）得到结果  
 返回值是一个Future实例对象的迭代器  
"""  
"""  
代码：  
from concurrent.futures import ProcessPoolExecutor,wait,ALL\_COMPLETED,FIRST\_COMPLETED,FIRST\_EXCEPTION,as\_completed  
import time,random  
def do\_sth(i):  
 time.sleep(random.random() \* 10)  
 return i \* i  
ppe = ProcessPoolExecutor(max\_workers = 3)  
objs = []  
for i in range(1,5):  
 if \_\_name\_\_ == '\_\_main\_\_':  
 future = ppe.submit(do\_sth, i)  
 objs.append(future)  
future\_iterator = as\_completed(objs)  
for future in future\_iterator:  
 print(future.result())  
运行结果：  
1  
4  
16  
9  
"""  
"""  
 进程池ThreadPoolExecutor  
 进程池ThreadPoolExecutor：管理进程的数目是确定的  
 标准库模块concurrent.futures中提供了一个类对象ThreadPoolExecutor，特用于表示进程池，与Pool相比ThreadPoolExecutor的功能很  
强大  
 类对象ThreadPoolExecutor遵守了上下文管理协议，所以可以使用with语句，这样在离开运行时上下文时，会自动调用方法shutdown(wait =  
True)  
那默认就是cpu的核数  
"""  
"""  
代码：  
from concurrent.futures import ThreadPoolExecutor  
import time  
def do\_sth(i):  
 time.sleep(2)  
 return i \*\* i  
with ThreadPoolExecutor(max\_workers = 3) as ppe:  
 objs = []  
 if \_\_name\_\_ == '\_\_main\_\_':  
 for i in range(1,11):  
 future = ppe.submit(do\_sth, i)  
 # result()是异步的，就是阻塞。得到的值就是do\_sth的返回值，直到do\_sth（）执行之后才会返回  
 print(future)  
 objs.append(future)  
for i in objs:  
 print(i.result())  
"""  
  
  
  
  
  
  
  
  
  
"""  
全部变量在多个进程中不能共享  
"""  
"""  
代码：  
from multiprocessing import Process  
num = 18  
def do\_sth():  
 global num  
 num += 1  
p = Process(target=do\_sth)  
if \_\_name\_\_ == '\_\_main\_\_':  
 p.start()  
 p.join()  
print(num)  
运行结果：  
18  
18  
总结：在子进程中修改全局变量对父进程没有影响，因为子进程对父进程中的全局变量做了一个拷贝，子进程与父进程中的num是完全不同的两个变量  
每个进程都有独立的内存空间，而进程空间是相互独立的。因此，全局变量在多个进程中能共享  
"""  
"""  
线程内的全局变量可以共享  
"""  
"""  
代码：  
from threading import Thread  
num = 18  
def do\_sth():  
 global num  
 num += 1  
p = Thread(target=do\_sth)  
if \_\_name\_\_ == '\_\_main\_\_':  
 p.start()  
 p.join()  
print(num)  
运行结果：  
19  
总结：因此，全局变量在多个线程中可以共享  
"""  
"""  
 多线程执行是不确定的，导致多线程共享数据的结果是不可预期的，不安全的  
"""  
"""  
from threading import Thread  
num = 0  
def do\_sth():  
 global num  
 for i in range(1000000):  
 #就相当于num = num +1、   
 #首先计算num + 1，存入临时变量中欧，然后像临时变量赋值给num  
 #m = m + 1   
 #m = m + 1   
 #线程1 执行完m+1然后调度到线程2  
 #线程2 执行完m+1然后调度到线程1  
 #线程1 将1赋值给m 现在m = 1  
 #线程2 将1赋值给m 依然是m = 1  
 num += 1   
t1 = Thread(target=do\_sth)  
t2 = Thread(target=do\_sth)  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
print(num)  
运行结果：  
1130625  
"""  
"""  
线程间内存共享  
共享内存：指的就是在内存中开辟一个空间专门用来进程之间的通信，多个进程可以访问这个共享内存  
共享内存是实现通信的常用方式之一，它允许多个进程直接访问同一块内存空间  
共享内存必须是ctypes的。ctypes是与c语言兼容的一种数据类型。  
"""  
"""  
Value():返回值表示一个数值，typecode\_or\_type用于执行数值的类型码或者是类型  
Array()：返回值用于表示一个数组，typecode\_or\_type用于指定数组中元素的类型码或者ctypes类型，size\_or\_initializer用于指定数组的长度、  
或python中的序列  
"""  
"""  
代码：  
from multiprocessing import Value, Array, Process  
import ctypes  
#在共享内存中创建了一个ctypes对象  
num = Value('d', 2.3)  
#num = Value(ctypes.c\_float, 2.3)  
#创建了一个表示数组的ctypes对象  
arr = Array('i', range(1, 5))  
#arr = Array(ctypes.c\_int, range(1,5))  
def do\_sth(num, arr):  
 num.value = 5.6  
 for i in range(len(arr)):  
 arr[i] = -arr[i]  
p1 = Process(target=do\_sth,args=(num,arr))  
if \_\_name\_\_ == '\_\_main\_\_':  
 p1.start()  
 p1.join()  
print(num.value)  
print(arr[:])  
运行结果：  
2.3  
[1, 2, 3, 4]  
5.6  
[-1, -2, -3, -4]  
"""  
"""  
 多进程执行是不确定的，导致多进程共享数据的结果是不可预期的，不安全的  
"""  
"""  
代码：  
from multiprocessing import Process, Value, Array  
import ctypes  
num = Value(ctypes.c\_int, 0)  
def do\_sth(num):  
 for i in range(10000):  
 #就相当于num = num +1、  
 #首先计算num + 1，存入临时变量中欧，然后像临时变量赋值给num  
 #m = m + 1  
 #m = m + 1  
 #线程1 执行完m+1然后调度到线程2  
 #线程2 执行完m+1然后调度到线程1  
 #线程1 将1赋值给m 现在m = 1  
 #线程2 将1赋值给m 依然是m = 1  
 num.value += 1  
p1 = Process(target=do\_sth, args=(num,))  
p2 = Process(target=do\_sth, args=(num,))  
if \_\_name\_\_ == '\_\_main\_\_':  
 p1.start()  
 p2.start()  
 p1.join()  
 p2.join()  
print(num.value)  
运行结果：  
0  
0  
11175  
"""  
  
  
  
  
  
  
  
  
  
  
"""  
 多线程同步lock()  
 为了保证多个线程能够安全的操作数据，必须保证一个线程操作数据的时候，其他的线程都不能操作  
 同理一个线程A在操作共享数据前必须显示图获得锁，从而给相关代码上锁，线程A获得锁之后锁的状态就变味了"locked"，如果另外一个线程试图获得锁  
线程b就会变为"blocked"并且被添加到锁等待池，只能等待获得锁的线程A在释放锁之后变为"unlocked"，线程调度程序再从锁等待池中处于状态"blocked"  
的线程中选择一个来获得锁，获得锁之后线程的状态变为"running"，由于只有一把锁。无论有所少个线程同一时刻，最多只有一个线程能获得该锁，这样就确保了  
共享数据的相关代码，只能有一个线程从头到尾完整的执行，从而就确保了多个线程共享数据总是安全的  
 但是包含锁的相关代码，只能以单线程模式执行，因此效率大大降低了。  
"""  
"""  
 标准库threading中提供了一个Lock，表示锁，以实现线程同步，同步就意味着等待和阻塞  
 为了保证获得锁的线程用完后一定要释放锁，可以将操作共享数据的相关代码方法try - finally语句块中  
"""  
"""  
代码:  
from threading import Thread, Lock  
lock = Lock()  
num = 0  
def do\_sth():  
 global num  
 for i in range(1000000):  
 lock.acquire() #上锁  
 try:  
 num += 1  
 finally:  
 lock.release() #下锁  
t1 = Thread(target=do\_sth)  
t2 = Thread(target=do\_sth)  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
print(num)  
运行结果：  
2000000  
"""  
"""  
因为lock实现了上下文管理协议，所以可以使用with语句。而且会自动上锁下锁  
from threading import Thread, Lock  
lock = Lock()  
num = 0  
def do\_sth():  
 global num  
 for i in range(1000000):  
 with lock:  
 num += 1  
t1 = Thread(target=do\_sth)  
t2 = Thread(target=do\_sth)  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
print(num)  
运行结果：  
2000000  
"""  
"""  
进程锁：  
"""  
"""  
import os,time,random  
from multiprocessing import Process, Lock, Value  
num = Value('i', 0)  
lock = Lock()  
def work(lock,num):  
 with lock:  
 for i in range(10000):  
 num.value += 1  
if \_\_name\_\_ == '\_\_main\_\_':  
 p=Process(target=work,args=(lock,num))  
 p1=Process(target=work,args=(lock,num))  
 p.start()  
 p1.start()  
 p.join()  
 p1.join()  
print(num.value)  
"""  
  
"""  
多线程之死锁  
 如果有多个共享数据，在多个线程操作这个共享数据时，如果有两个线程通过枷锁站有一部分数据。并且同时等待对方释放锁，这样就会出现死锁  
避免死锁的算法中最具代表性的是Dijkstra提出的银行家算法  
"""  
"""  
代码：  
from threading import Thread, Lock, current\_thread  
num1 = 0  
num2 = 0  
l1 = Lock()  
l2 = Lock()  
def fun1():  
 global num2, num1  
 try:  
 l1.acquire()  
 print('在方法fun1中，l1锁中：', current\_thread().getName())  
 num1 += 1  
 try:  
 l2.acquire()  
 print('在方法fun1中，l2锁中：', current\_thread().getName())  
 num2 += 1  
 except:  
 l2.release()  
 except:  
 l1.release()  
  
def fun2():  
 global num2, num1  
 try:  
 l2.acquire()  
 print('在方法fun2中，l2锁中：', current\_thread().getName())  
 num1 += 1  
 try:  
 l1.acquire()  
 print('在方法fun2中，l1锁中：', current\_thread().getName())  
 num2 += 1  
 except:  
 l1.release()  
 except:  
 l2.release()  
def do\_sth():  
 fun1()  
 fun2()  
  
thlist = []  
for i in range(100):  
 t = Thread(target=do\_sth)  
 if \_\_name\_\_ == '\_\_main\_\_':  
 t.start()  
 thlist.append(t)  
for i in thlist:  
 if \_\_name\_\_ == '\_\_main\_\_':  
 i.join()  
运行结果：  
在方法fun1中，l1锁中： Thread-1  
在方法fun1中，l2锁中： Thread-1  
"""  
"""  
在标准库模块threading中还提供了用于表示类对象Rlook（Reentrant Lock:表示可重入锁），不同的是，同一个线程中多次调用方法acquire（）  
不会造成死锁。  
在Rlook的内部，维护了一个Look的计数器counter。counter记录了acquire（）的次数  
def acquire(self, blocking=True, timeout=-1):  
 me = get\_ident()  
 if self.\_owner == me:  
 self.\_count += 1  
 return 1  
 rc = self.\_block.acquire(blocking, timeout)  
 if rc:  
 self.\_owner = me  
 self.\_count = 1  
 return rc  
当线程第一次调用方法acquite（）获得锁时，锁的拥有者被保存，同时计数器counter被初始化为1，再次调用方法acquire（）首先判断调用者  
是不是锁的拥有者，如果是计数器counter+1  
 def release(self):  
 if self.\_owner != get\_ident(): #判断调用者是否是锁的拥有者  
 raise RuntimeError("cannot release un-acquired lock")  
 self.\_count = count = self.\_count - 1  
 if not count:  
 self.\_owner = None  
 self.\_block.release()  
当调用方法release会先判断调用者是否是锁的拥有者，如果是，计数器counter减一，如果计数器counter变为0，则将锁的拥有者变为None，然后释放锁  
Rlook()就相当于上了多把锁，但是上了多少吧锁，就得解开多少把锁，注意上锁和解锁一定要成对出现  
Rlook()也遵守了上下文管理协议  
"""  
"""  
from threading import Thread, RLock  
num1 = 0  
num2 = 0  
rlock = RLock()  
def do\_sth():  
 rlock.acquire()  
 try:  
 fun1()  
 fun2()  
 finally:  
 rlock.release()  
def fun1():  
 global num1  
 with rlock:  
 num1 += 1  
def fun2():  
 global num2  
 with rlock:  
 num2 += 1  
thlist = []  
for i in range(10):  
 t = Thread(target=do\_sth)  
 if \_\_name\_\_ == '\_\_main\_\_':  
 t.start()  
 thlist.append(t)  
for i in thlist:  
 if \_\_name\_\_ == '\_\_main\_\_':  
 i.join()  
print(num1)  
print(num2)  
"""  
"""  
同样的multiprocessing也提供了一个Rlock()方法用来  
"""  
"""  
代码：  
from multiprocessing import Process, RLock, Array  
import ctypes  
array = Array(ctypes.c\_int,[0, 0])  
rlock = RLock()  
def do\_sth(rlock, array):  
 rlock.acquire()  
 try:  
 fun1(rlock, array)  
 fun2(rlock, array)  
 finally:  
 rlock.release()  
def fun1(rlock, array):  
 with rlock:  
 array[0] += 1  
def fun2(rlock, array):  
 with rlock:  
 array[1] += 1  
thlist = []  
for i in range(10):  
 p = Process(target=do\_sth, args=(rlock, array))  
 if \_\_name\_\_ == '\_\_main\_\_':  
 p.start()  
 thlist.append(p)  
for item in thlist:  
 if \_\_name\_\_ == '\_\_main\_\_':  
 item.join()  
print(array[0])  
print(array[1])  
"""  
"""  
多线程同步之Condition  
用于表示触发带触发条件的锁，帮助我们处理多线程间复杂的问题。可以允许一个或者多个线程等待触发条件，直到另外一个线程的通知  
"""  
"""  
代码：  
from threading import Condition, Thread  
import time  
cond = Condition()  
class MyThread1(Thread):  
 def \_\_init\_\_(self, name):  
 super().\_\_init\_\_(name=name)  
 def run(self) -> None:  
 cond.acquire()  
 print('线程%s说：干啥呢兄弟' % self.name)  
 cond.notify() #给MyThread2发通知  
 cond.wait() #通知完之后等待  
 time.sleep(2)  
 print('线程%s说：那脚丫子香不香啊' % self.name)  
 cond.notify() #给MyThread2发通知  
 cond.wait()  
 time.sleep(2)  
 print('线程%s说：有那么香么' % self.name)  
 cond.notify() # 给MyThread2发通知  
 cond.release()  
class MyThread2(Thread):  
 def \_\_init\_\_(self, name):  
 super().\_\_init\_\_(name=name)  
 def run(self) -> None:  
 cond.acquire()  
 time.sleep(2)  
 print('线程%s说：没事啊！抠脚丫子呢' % self.name)  
 cond.notify() # 给MyThread1发通知  
 cond.wait() # 通知完之后等待  
 time.sleep(2)  
 print('线程%s说：那倍香啊' % self.name)  
 cond.notify() # 给MyThread1发通知  
 cond.wait() # 通知完之后等待  
 time.sleep(2)  
 print('线程%s说：那必须香啊' % self.name)  
 cond.release()  
MyThread1('Thread1').start()  
MyThread2('Thread2').start()  
"""  
"""  
多进程中依然可以使用Condition  
"""  
"""  
生产者和消费者问题：  
假设有一群生产者（Producer）和一群消费者（Consumer）通过市场来交换产品  
生产者的策略是：如果市场剩余的产品少于20个，那就生产4个产品放到市场上  
消费者的策略是：如果市场上剩余的产品多于10个，那就从市场上消费3个产品  
"""  
"""  
from threading import Condition,Thread  
import time  
cond = Condition()  
num = 0 #表示所有的产品数量  
class Producer(Thread): #用来表示生产者  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def run(self) -> None:  
 global num  
 cond.acquire()  
 while True:  
 time.sleep(2)  
 if num < 20:  
 num += 4  
 print('生产者生产了4个产品，产品数为:', num)  
 if num >= 20:  
 cond.notify()  
 cond.wait()  
  
class Consumer(Thread): #用来表示消费者  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
  
 def run(self) -> None:  
 global num  
 cond.acquire()  
 while True:  
 time.sleep(2)  
 if num > 10:  
 num -= 3  
 print('消费者消费了3个产品，产品数量为：', num)  
 if num <= 10:  
 cond.notify()  
 cond.wait()  
  
Producer().start()  
Consumer().start()  
"""  
"""  
from threading import Thread, Lock  
import time  
lock = Lock()  
num = 0 #表示所有的产品数量  
  
def Producer\_Consumer(name):  
 global num  
 with lock:  
 if num < 20:  
 num += 4  
 print('生产者生产了4个产品，产品数为:', num)  
 time.sleep(2)  
 if num > 10:  
 num -= 3  
 print('消费者消费了3个产品，产品数量为：', num)  
 time.sleep(2)  
t1 = Thread(target=Producer\_Consumer, args=('Producer',))  
t2 = Thread(target=Producer\_Consumer, args=('Consumer',))  
while True:  
 t1.start()  
 t2.start()  
"""  
"""  
from threading import Thread  
import time  
num = 0  
def do\_sth():  
 global num  
 while True:  
 time.sleep(1)  
 if num < 20:  
 num += 4  
 print('生产者生产了4个产品，产品数为:', num)  
 if num > 10:  
 num -= 3  
 print('消费者消费了3个产品，产品数量为：', num)  
t1 = Thread(target=do\_sth)  
t2 = Thread(target=do\_sth)  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
"""  
"""  
多线程同步之Semaphore  
标准库模块threading中提供了一个类对象Semaphore，用于表示信号量，它可以帮助我们控制并发线程的最大数量，从而实现多线程之间的同步  
Semaphore也遵守了上下文管理协议，所以可以使用with  
"""  
"""  
代码：  
from threading import Semaphore, Thread  
import time, random  
se = Semaphore(3)  
class MyThread(Thread):  
 def run(self) -> None:  
 se.acquire()  
 print('%s获得资源' % self.name)  
 time.sleep(random.random() \* 10)  
 se.release()  
  
for i in range(10):  
 MyThread().start()  
  
from threading import Semaphore, Thread  
import time, random  
se = Semaphore(3)  
class MyThread(Thread):  
 def run(self) -> None:  
 with se:  
 print('%s获得资源' % self.name)  
 time.sleep(random.random() \* 10)  
for i in range(10):  
 MyThread().start()  
"""  
"""  
如果acquire()的数量超过release()，那Semaphore的初始值将会超过，所以可以引用BoundedSemaphore（有边界的Semaphore）,一旦程序中出现  
这种bug就会抛出ValueError异常  
"""  
"""  
from threading import BoundedSemaphore, Thread  
import time, random  
se = BoundedSemaphore(3)  
class MyThread(Thread):  
 def run(self) -> None:  
 se.acquire()  
 print('%s获得资源' % self.name)  
 time.sleep(random.random() \* 10)  
 se.release()  
  
for i in range(10):  
 MyThread().start()  
"""  
"""  
multiprocessing中也设置的有BoundedSemaphore  
from multiprocessing import BoundedSemaphore  
bs = BoundedSemaphore(3)  
bs.acquire()  
bs.release()  
bs.release()  
"""  
"""  
多线程同步之Event  
标准库模块threading中提供了一个Event，也可以实现多线程间的同步，Event实现了一个内部标志，通过改变内部标志的值，可以让一个线程  
给其他处于阻塞状态的线程发送一个事件信号,从而唤醒这些线程转为运行状态  
 set()用于将内部标志设置为True  
 is\_set()用于判断内部标志是否被设置为True  
 clear()将内部标志设置为False  
 wait()当内部标志为False的时候调用方法的线程会被阻塞，直到另外一个线程调用了方法set()将内部标志设置为True将被阻塞的线程才  
 会转为运行状态  
"""  
"""  
from threading import Event, Thread, current\_thread  
import time  
event = Event()  
def do\_sth():  
 print('%s开始等待：' % current\_thread().getName())  
 event.wait()  
 print('%s结束等待：' % current\_thread().getName())  
  
for i in range(3):  
 Thread(target=do\_sth).start()  
time.sleep(2)  
event.set()  
"""  
"""  
multiprocessing中也提供了Event  
 from multiprocessing import Event, Process, current\_process  
 import time  
 event = Event()  
 def do\_sth():  
 print('%s开始等待：' % current\_process().name)  
 event.wait()  
 print('%s结束等待：' % current\_process().name)  
   
 for i in range(3):  
 if \_\_name\_\_ == '\_\_main\_\_':  
 Process(target=do\_sth).start()  
 time.sleep(2)  
 event.set()  
"""  
  
  
  
  
"""  
管道：（半双工（单向）：全装工（双向））  
标准库模块multiprocessing提供了一个函数Pipe，返回值是一个元组，元组中包含两个都西昂，分别表示管道两端的链接  
调用函数Pipe()时  
 如果不传入参数或者传入参数为True时，管道工作方式为双向（全双工）  
 如果传入参数为Flase，管道工作方式为单项（半双工），第一个链接对象只能接收对象，第二个连接对象只能发送数据  
对于管道两端的连接对象，主要有两个方法  
send(self, obj):用于将参数obj所指定的对象发送到管道  
recv(self)：用于从管道中接收对象，如果没有被接受的对象，程序就会被阻塞  
"""  
"""  
from multiprocessing import Pipe  
conn1, conn2 = Pipe()  
conn1.send('conn1第一次发送的数据')  
conn1.send('conn2第二次发送的数据')  
conn2.send('conn1第一次发送的数据')  
conn2.send('conn2第二次发送的数据')  
  
print(conn1.recv())  
print(conn1.recv())  
print(conn2.recv())  
print(conn2.recv())  
"""  
"""  
from multiprocessing import Pipe  
c1, c2 = Pipe(False) #注意：这里只能由c2发送c1接收  
c2.send('c2第一次发送的')  
c2.send('c2第二次发送的')  
print(c1.recv())  
print(c1.recv())  
"""  
"""  
通过管道在进程见发送和接收数据  
所以如果想要实现进程之间的通信，管道也是常用的方式之一  
"""  
"""  
代码：  
from multiprocessing import Pipe, Process, current\_process  
import time, random  
print('父进程%s开始' % current\_process().name)  
cr, cs = Pipe(False)  
def process\_send(conn):  
 for i in range(11):  
 conn.send(i)  
 print('发送数据：', i)  
 conn.send(None)  
 print('发送数据：None')  
def process\_recv(conn):  
 while True:  
 item = conn.recv()  
 if item == None:  
 print('接收到数据None，退出')  
 break  
 print('接收到数据：', item)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 p1 = Process(target=process\_send, args=(cs,)).start()  
 p2 = Process(target=process\_recv, args=(cr,)).start()  
 p1.join()  
 p2.join()  
  
print('父进程%s结束' % current\_process().name)  
"""  
"""  
进程之间的通信之Manager也是常见的实现方式之一，与共享内存相比，Manager更加灵活，因为Manager支持多种类型，此外Manager也可以通过网络被  
不同计算机上的进程所共享，但是Manager要比共享内存慢  
"""  
"""  
from multiprocessing import Process, Manager  
def do\_sth(d, l):  
 d[0] = '0'  
 d[1] = '1'  
 d[2] = '2'  
 l.reverse()  
if \_\_name\_\_ == '\_\_main\_\_':  
 manager = Manager()  
 #通过Manafer来创建一个用于进程之间的通信  
 d = manager.dict()  
 l = manager.list(range(10))  
 p = Process(target=do\_sth,args=(d, l))  
 p.start()  
 p.join()  
 print(d)  
 print(l)  
"""  
"""  
def do\_sth():  
 while True:  
 pass  
do\_sth()  
单进程或者单线程沾满了八核cpu的一核  
"""  
"""  
from multiprocessing import Process  
def do\_sth():  
 while True:  
 pass  
if \_\_name\_\_ == '\_\_main\_\_':  
 Process(target=do\_sth).start()  
 Process(target=do\_sth).start()  
do\_sth()  
三个进程沾满了八核cpu中的三核，因此多进程可以实现并行（同时处理多个任务）从而发挥最大优势  
"""  
"""  
from threading import Thread  
def do\_sth():  
 while True:  
 pass  
if \_\_name\_\_ == '\_\_main\_\_':  
 Thread(target=do\_sth).start()  
 Thread(target=do\_sth).start()  
do\_sth()  
三个线程并没有沾满八核cpu的三核只是沾满了一个并行，因此多线程并不能实现并行（同时处理多个任务），而只是能实现并发（交替处理多个任务）  
"""  
"""  
我们编写的Python代码，是通过Python解释器来执行的，通常使用的Python解释器是官方提供的CPython。CPython，中有一个GIL（Global Interpreter   
Lock）其作用相当于Lock，任何线程在执钱必须获得GIL，一个线程在获得GIL后其他线程就不能执行，直到线程GIL释放后。因此，GIL保证了同一时  
刻只有一个线程可以执行，从而导致Python中的多线程不能实现并行。  
  
"""  
"""  
定时器线程Timer Timer是Thread的子类  
如果想要在指定的时间片段之后执行子线程，可以使用Timer，用于表示定时器线程，也是通过start来启动线程  
定时器只是执行一次，如果需要每隔一段时间指定一次。可以在函数的内部再次创建与启动定时器  
"""  
"""  
代码：  
from threading import Timer  
import time  
def do\_sth():  
 global timer  
 print('Hello word')  
 timer = Timer(3, do\_sth)  
 timer.start() #然后每隔三秒钟启动一次（打印Hello word）  
  
timer = Timer(2, do\_sth) #隔两秒钟启动  
timer.start()  
time.sleep(10)  
timer.cancel()  
print('父线程结束')  
运行结果：  
Hello word  
Hello word  
Hello word  
父线程结束  
"""  
"""  
为什么需要ThreadLocal  
多线程共享的全局变量是不安全的，那局部变量呢？  
局部变量只归某个线程私有，其他线程是无法访问的。但是，在线程内操作局部变量也存在问题：  
如果线程内有多个函数都需要访问多个局部变量，则需要将这些局部变量都作为实参分别传递给这些函数。  
这样，传递参数就会很麻烦。  
"""  
"""  
代码：  
from threading import Thread,current\_thread  
def do\_sth(arg1, arg2, arg3):  
 local\_var1 = arg1  
 local\_var2 = arg2  
 local\_var3 = arg3  
 fun1(local\_var1, local\_var2, local\_var3)  
 fun2(local\_var1, local\_var2, local\_var3)  
 fun3(local\_var1, local\_var2, local\_var3)  
def fun1(local\_var1, local\_var2, local\_var3):  
 print('%s:%s - %s - %s' % (current\_thread().name, local\_var1, local\_var2, local\_var3))  
def fun2(local\_var1, local\_var2, local\_var3):  
 print('%s:%s - %s - %s' % (current\_thread().name, local\_var1, local\_var2, local\_var3))  
def fun3(local\_var1, local\_var2, local\_var3):  
 print('%s:%s - %s - %s' % (current\_thread().name, local\_var1, local\_var2, local\_var3))  
t = Thread(target=do\_sth, args=('1','2','3'))  
t1 = Thread(target=do\_sth, args=('4','5','6'))  
t.start()  
t1.start()  
运行结果：  
Thread-1:1 - 2 - 3  
Thread-1:1 - 2 - 3  
Thread-1:1 - 2 - 3  
Thread-2:4 - 5 - 6  
Thread-2:4 - 5 - 6  
Thread-2:4 - 5 - 6  
如果需要传入100个参数将会很麻烦  
"""  
"""  
什么是ThreadLocal  
ThreadLocal是一个全局变量，用来存放各个线程的局部变量。ThreadLocal中会维护‘某个线程 - 某个局部变量名字 - 该局部变量的值’的映射。  
(所以在各个线程中是不会混淆的)  
在线程中将局部变量写入ThreadLocal语法格式：threadlocal.局部变量名 = 局部变量值  
在线程中从ThreadLocal中读取局部变量的语法格式为：threadlocal.局部变量名  
"""  
  
from** threading **import** Thread,current\_thread, local  
thread\_local = local()  
**def** do\_sth(arg1, arg2, arg3):  
 thread\_local.local\_var1 = arg1  
 thread\_local.local\_var2 = arg2  
 thread\_local.local\_var3 = arg3  
 fun1()  
 fun2()  
 fun3()  
**def** fun1():  
 print(**'%s:%s - %s - %s'** % (current\_thread().name, thread\_local.local\_var1, thread\_local.local\_var2, thread\_local.local\_var3))  
**def** fun2():  
 print(**'%s:%s - %s - %s'** % (current\_thread().name, thread\_local.local\_var1, thread\_local.local\_var2, thread\_local.local\_var3))  
**def** fun3():  
 print(**'%s:%s - %s - %s'** % (current\_thread().name, thread\_local.local\_var1, thread\_local.local\_var2, thread\_local.local\_var3))  
t = Thread(target=do\_sth, args=(**'1'**,**'2'**,**'3'**))  
t1 = Thread(target=do\_sth, args=(**'4'**,**'5'**,**'6'**))  
t.start()  
t1.start()