**进程**

**<1>进程的创建-fork**

**1. 进程 VS 程序:**

编写完毕的代码，在没有运行的时候，称之为**程序**

正在运行着的代码，就成为**进程**

进程，除了包含代码以外，还有需要运行的环境等，所 以和程序是有区别的

**2. fork( )**

Python的os模块封装了常见的系统调用，其中就包括 fork，可以在Python程序中轻松创建子进程：

import os

# 注意，fork函数，只在Unix/Linux/Mac上运行，windows不可以

pid = os.fork()

if pid == 0:

print('哈哈1')

else:

print('哈哈2')

程序执行到os.fork()时，操作系统会创建一个新的进 程（子进程），然后复制父进程的所有信息到子进程中

然后父进程和子进程都会从fork()函数中得到一个返 回值，在子进程中这个值一定是0，而父进程中是子进程 的 id号

**3. getpid()、getppid()**

import os

rpid = os.fork()

if rpid<0:

print("fork调用失败。")

elif rpid == 0:

print("我是子进程（%s），我的父进程是（%s） "%(os.getpid(),os.getppid()))

x+=1

else:

print("我是父进程（%s），我的子进程是（%s） "%(os.getpid(),rpid))

print("父子进程都可以执行这里的代码")

**<2>多进程修改全局变量**

#coding=utf-8

import os

import time

num = 0

# 注意，fork函数，只在Unix/Linux/Mac上运行，windows不可以

pid = os.fork()

if pid == 0:

num+=1

print('哈哈1---num=%d'%num)

else:

time.sleep(1)

num+=1

print('哈哈2---num=%d'%num)

多进程中，每个进程中所有数据（包括全局变量）都各有拥有一份，互不影响。

**<3>多次fork问题**

#coding=utf-8

import os

import time

# 注意，fork函数，只在Unix/Linux/Mac上运行，windows不可以

pid = os.fork()

if pid == 0:

print('哈哈1')

else:

print('哈哈2')

pid = os.fork()

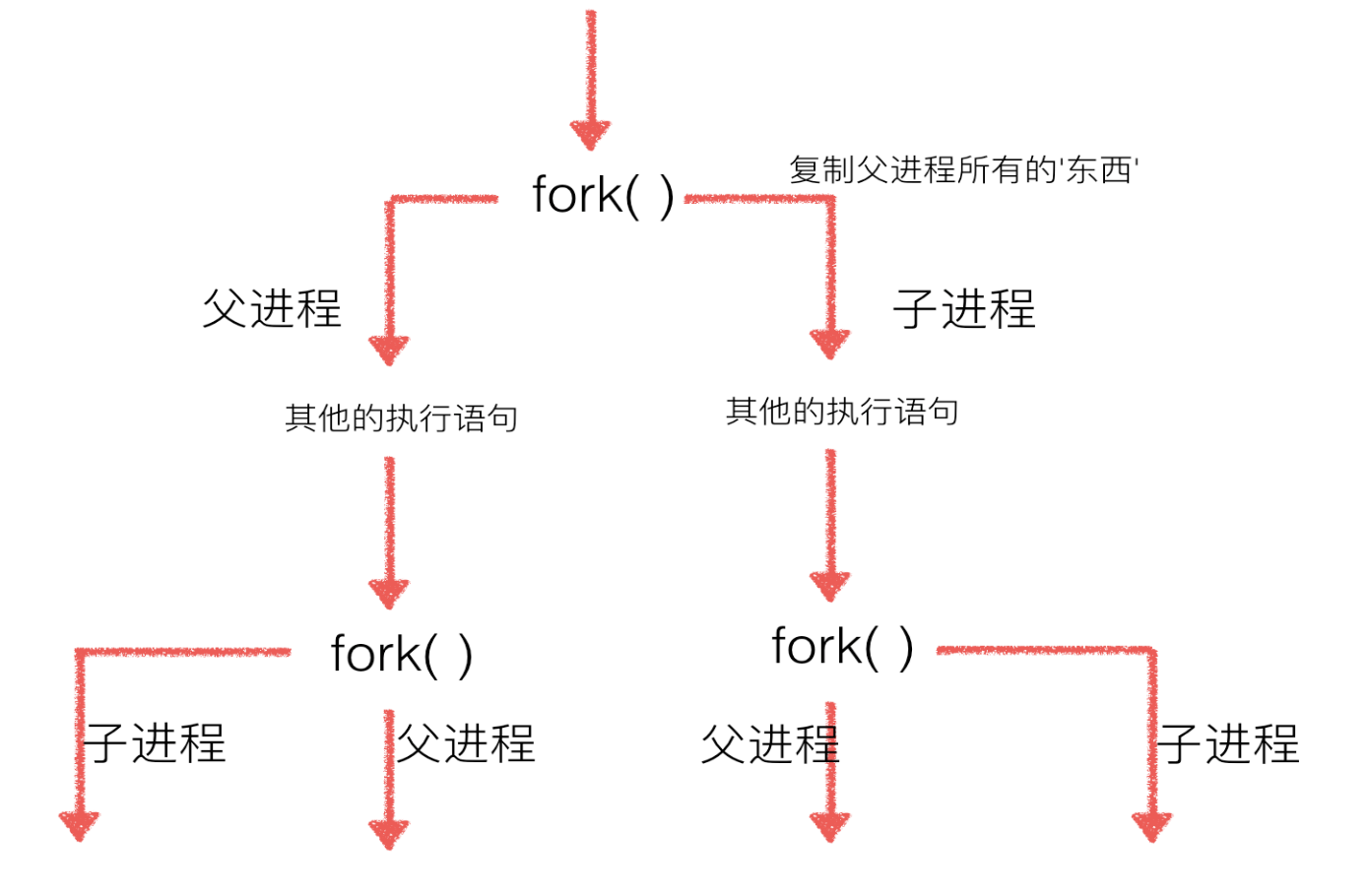
if pid == 0:

print('哈哈3')

else:

print('哈哈4')

time.sleep(1)



父进程、子进程执行顺序没有规律，完全取决于操作系统的调度算法

**<4> multiprocessing**

multiprocessing模块是跨平台版本的多进程模块。multiprocessing模块提供了一个Process类来代表一个进程对象。

#coding=utf-8

from multiprocessing import Process

import os

# 子进程要执行的代码

def run\_proc(name):

print('子进程运行中，name= %s ,pid=%d...' % (name, os.getpid()))

if \_\_name\_\_=='\_\_main\_\_':

print('父进程 %d.' % os.getpid())

p = Process(target=run\_proc, args=('test',))

print('子进程将要执行')

p.start()

p.join()

print('子进程已结束')

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。

join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

**<5> Process语法结构**

Process([group [, target [, name [, args [, kwargs]]]]])

target：表示这个进程实例所调用对象；

args：表示调用对象的位置参数**元组**；

kwargs：表示调用对象的关键字参数字典；

name：为当前进程实例的别名；

group：大多数情况下用不到；

**<5> Process类常用方法**

is\_alive()：判断进程实例是否还在执行；

join([timeout])：是否等待进程实例执行结束，或等待多少秒；

start()：启动进程实例（创建子进程）；

run()：如果没有给定target参数，对这个对象调用start()方法时，就将执行对象中的run()方法；

terminate()：不管任务是否完成，立即终止；

Process类常用属性：

name：当前进程实例别名，默认为Process-N，N为从1开 始递增的整数；

pid：当前进程实例的PID值；

**<6>进程池Pool**

multiprocessing模块提供Pool方法，初始化Pool时，可以指定一个最大进程数，当有新的请求提交到Pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到指定的最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来执行。

from multiprocessing import Pool

import os,time,random

def worker(msg):

t\_start = time.time()

print("%s开始执行,进程号为%d"%(msg,os.getpid()))

#random.random()随机生成0~1之间的浮点数

time.sleep(random.random()\*2)

t\_stop = time.time()

print(msg,"执行完毕，耗时%0.2f"%(t\_stop-t\_start))

po=Pool(3) #定义一个进程池，最大进程数3

for i in range(0,10):

#Pool.apply\_async(要调用的目标,(传递给目标的参数元祖,))

#每次循环将会用空闲出来的子进程去调用目标

po.apply\_async(worker,(i,))

print("----start----")

po.close() #关闭进程池，关闭后po不再接收新的请求

po.join() #等待po中所有子进程执行完成，必须放在close语句之后

print("-----end-----")

**<7>multiprocessing.Pool常用函数：**

apply\_async(func[, args[, kwds]]) ：使用非阻塞方式调用func（并行执行，堵塞方式必须等待上一个进程退出才能执行下一个进程），args为传递给func的参数列表，kwds为传递给func的关键字参数列表；

apply(func[, args[, kwds]])：使用阻塞方式调用func

close()：关闭Pool，使其不再接受新的任务；

terminate()：不管任务是否完成，立即终止；

join()：主进程阻塞，等待子进程的退出， 必须在close或terminate之后使用；

**<8>apply堵塞式:**

from multiprocessing import Pool

import os,time,random

def worker(msg):

t\_start = time.time()

print("%s开始执行,进程号为%d"%(msg,os.getpid()))

#random.random()随机生成0~1之间的浮点数

time.sleep(random.random()\*2)

t\_stop = time.time()

print(msg,"执行完毕，耗时%0.2f"%(t\_stop-t\_start))

po=Pool(3) #定义一个进程池，最大进程数3

for i in range(0,10):

po.apply(worker,(i,))

print("----start----")

po.close() #关闭进程池，关闭后po不再接收新的请求

po.join() #等待po中所有子进程执行完成，必须放在close语句之后

print("-----end-----")

**<9>进程间通信-Queue**

**1. Queue的使用**

可以使用multiprocessing模块的Queue实现多进程之间 的数据传递，Queue本身是一个消息列队程序。

#coding=utf-8

from multiprocessing import Queue

q=Queue(3) #初始化一个Queue对象，最多可接收三条put消息

q.put("消息1")

q.put("消息2")

print(q.full()) #False

q.put("消息3")

print(q.full()) #True

#因为消息列队已满下面的try都会抛出异常，第一个try会等待2 秒后再抛出异常，第二个Try会立刻抛出异常

try:

q.put("消息4",True,2)

except:

print("消息列队已满，现有消息数量:%s"%q.qsize())

try:

q.put\_nowait("消息4")

except:

print("消息列队已满，现有消息数量:%s"%q.qsize())

#推荐的方式，先判断消息列队是否已满，再写入

if not q.full():

q.put\_nowait("消息4")

#读取消息时，先判断消息列队是否为空，再读取

if not q.empty():

for i in range(q.qsize()):

print(q.get\_nowait())

初始化Queue()对象时（例如：q=Queue()），若括号中没 有指定最大可接收的消息数量，或数量为负值，那么就代表可 接受的消息数量没有上限（直到内存的尽头）；

Queue.qsize()：返回当前队列包含的消息数量；

Queue.empty()：如果队列为空，返回True，反之False ；

Queue.full()：如果队列满了，返回True,反之False；

Queue.get([block[, timeout]])：获取队列中的一条消 息，然后将其从列队中移除，block默认值为True；

1）如果block使用默认值，且没有设置timeout（单位 秒），消息列队如果为空，此时程序将被阻塞（停在读取状态）， 直到从消息列队读到消息为止，如果设置了timeout，则会等 待timeout秒，若还没读取到任何消息，则抛出"Queue.Empty" 异常；

2）如果block值为False，消息列队如果为空，则会立刻 抛出"Queue.Empty"异常；

Queue.get\_nowait()：相当Queue.get(False)；

Queue.put(item,[block[, timeout]])：将item消息写 入队列，block默认值为True；

1）如果block使用默认值，且没有设置timeout（单位 秒），消息列队如果已经没有空间可写入，此时程序将被阻塞 （停在写入状态），直到从消息列队腾出空间为止，如果设置 了timeout，则会等待timeout秒，若还没空间，则抛出 "Queue.Full"异常；

2）如果block值为False，消息列队如果没有空间可写入， 则会立刻抛出"Queue.Full"异常；

Queue.put\_nowait(item)：相当Queue.put(item, False)；

**2. Queue实例**

from multiprocessing import Process, Queue

import os, time, random

# 写数据进程执行的代码:

def write(q):

for value in ['A', 'B', 'C']:

print 'Put %s to queue...' % value

q.put(value)

time.sleep(random.random())

# 读数据进程执行的代码:

def read(q):

while True:

if not q.empty():

value = q.get(True)

print 'Get %s from queue.' % value

time.sleep(random.random())

else:

break

if \_\_name\_\_=='\_\_main\_\_':

# 父进程创建Queue，并传给各个子进程：

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

# 启动子进程pw，写入:

pw.start()

# 等待pw结束:

pw.join()

# 启动子进程pr，读取:

pr.start()

# pr进程里是死循环，无法等待其结束，只能强行终止:

print ''

print '所有数据都写入并且读完'

**3. 进程池中的Queue**

如果要使用Pool创建进程，就需要使用 multiprocessing.Manager()中的Queue()，而不是 multiprocessing.Queue()。

#修改import中的Queue为Manager

from multiprocessing import Manager,Pool

import os,time,random

def reader(q):

print("reader启动(%s),父进程为 (%s)"%(os.getpid(),os.getppid()))

for i in range(q.qsize()):

print("reader从Queue获取到消 息：%s"%q.get(True))

def writer(q):

print("writer启动(%s),父进程为 (%s)"%(os.getpid(),os.getppid()))

for i in "dongGe":

q.put(i)

if \_\_name\_\_=="\_\_main\_\_":

print("(%s) start"%os.getpid())

q=Manager().Queue() #使用Manager中的Queue来初始化

po=Pool()

#使用阻塞模式创建进程，这样就不需要在reader中使用死循 环了，可以让writer完全执行完成后，再用reader去读取

po.apply(writer,(q,))

po.apply(reader,(q,))

po.close()

po.join()

print("(%s) End"%os.getpid())

**线程**

**<1>多线程-threading**

**1. 使用threading模块**

**单线程执行**

import time

def saySorry():

print("亲爱的，我错了，我能吃饭了吗？")

time.sleep(1)

if \_\_name\_\_ == "\_\_main\_\_":

for i in range(5):

saySorry()

**多线程执行**

import threading

import time

def saySorry():

print("亲爱的，我错了，我能吃饭了吗？")

time.sleep(1)

if \_\_name\_\_ == "\_\_main\_\_":

for i in range(5):

t = threading.Thread(target=saySorry)

t.start() #启动线程，即让线程开始执行

1、可以明显看出使用了多线程并发的操作，花费时间要短很多

2、创建好的线程，需要调用start()方法来启动

**2. 主线程会等待所有的子线程结束后才结束**

import threading

from time import sleep,ctime

def sing():

for i in range(3):

print("正在唱歌...%d"%i)

sleep(1)

def dance():

for i in range(3):

print("正在跳舞...%d"%i)

sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

print('---开始---:%s'%ctime())

t1 = threading.Thread(target=sing)

t2 = threading.Thread(target=dance)

t1.start()

t2.start()

#sleep(5) # 屏蔽此行代码，试试看，程序是否会立马结束？

print('---结束---:%s'%ctime())

**3. 查看线程数量**

import threading

from time import sleep,ctime

def sing():

for i in range(3):

print("正在唱歌...%d"%i)

sleep(1)

def dance():

for i in range(3):

print("正在跳舞...%d"%i)

sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

print('---开始---:%s'%ctime())

t1 = threading.Thread(target=sing)

t2 = threading.Thread(target=dance)

t1.start()

t2.start()

while True:

length = len(threading.enumerate())

print('当前运行的线程数为：%d'%length)

if length<=1:

break

sleep(0.5)

**<2> threading注意点**

**1. 线程执行代码的封装**

import threading

import time

class MyThread(threading.Thread):

def run(self):

for i in range(3):

time.sleep(1)

#name属性中保存的是当前线程的名字

msg = "I'm "+self.name+' @ '+str(i)

print(msg)

if \_\_name\_\_ == '\_\_main\_\_':

t = MyThread()

t.start()

python的threading.Thread类有一个run方法，用于定 义线程的功能函数，可以在自己的线程类中覆盖该方法。而创 建自己的线程实例后，通过Thread类的start方法，可以启 动该线程，交给python虚拟机进行调度，当该线程获得执行 的机会时，就会调用run方法执行线程。(创建进程也可继承 Process类，重写run方法)

**2. 线程的执行顺序**

import threading

import time

class MyThread(threading.Thread):

def run(self):

for i in range(3):

time.sleep(1)

msg = "I'm "+self.name+' @ '+str(i)

print(msg)

def test():

for i in range(5):

t = MyThread()

t.start()

if \_\_name\_\_ == '\_\_main\_\_':

test()

**多线程程序的执行顺序是不确定的**

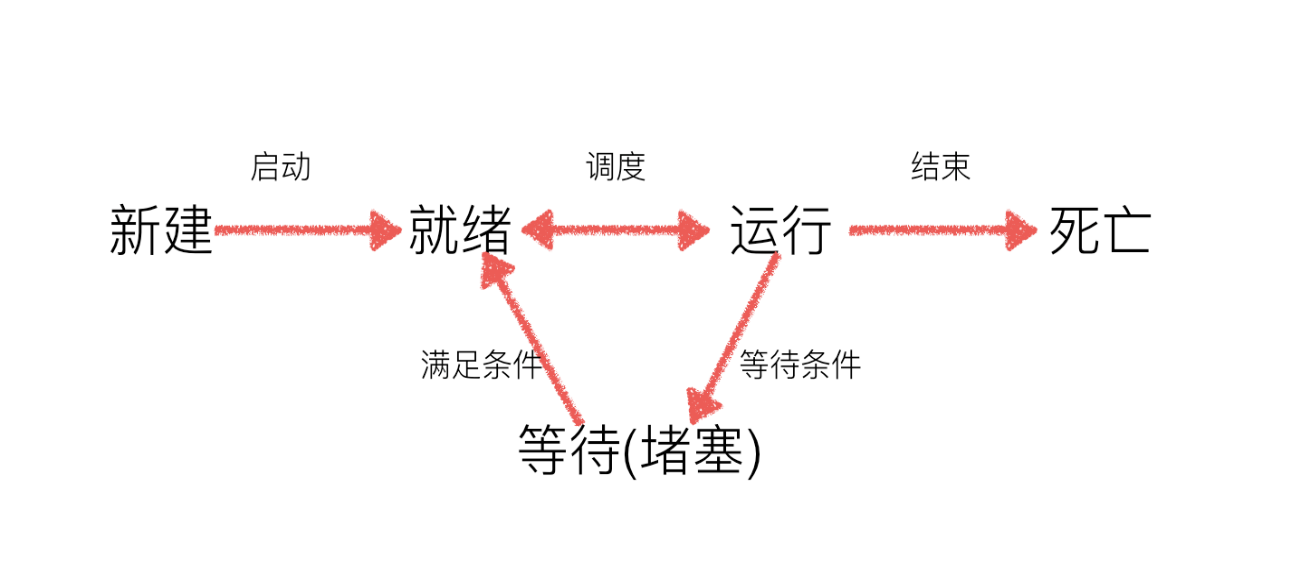
**3. 总结**

1、每个线程一定会有一个名字，尽管上面的例子中没有指 定线程对象的name，但是python会自动为线程指定一个名 字。

2、当线程的run()方法结束时该线程完成。

3、无法控制线程调度程序，但可以通过别的方式来影响线 程调度的方式。

4、线程的几种状态



**<3>多线程-共享全局变量**

from threading import Thread

import time

g\_num = 100

def work1():

global g\_num

for i in range(3):

g\_num += 1

print("----in work1, g\_num is %d---"%g\_num)

def work2():

global g\_num

print("----in work2, g\_num is %d---"%g\_num)

print("---线程创建之前g\_num is %d---"%g\_num)

t1 = Thread(target=work1)

t1.start()

#延时一会，保证t1线程中的事情做完

time.sleep(1)

t2 = Thread(target=work2)

t2.start()

**列表当做实参传递到线程中**

from threading import Thread

import time

def work1(nums):

nums.append(44)

print("----in work1---",nums)

def work2(nums):

#延时一会，保证t1线程中的事情做完

time.sleep(1)

print("----in work2---",nums)

g\_nums = [11,22,33]

t1 = Thread(target=work1, args=(g\_nums,))

t1.start()

t2 = Thread(target=work2, args=(g\_nums,))

t2.start()

1、在一个进程内的所有线程共享全局变量，能够在不适用 其他方式的前提下完成多线程之间的数据共享（这点要比多 进程要好）

2、缺点就是，线程是对全局变量随意遂改可能造成多线程 之间对全局变量的混乱（即线程非安全）

**<4>进程VS线程**

**定义的不同：**

1、进程是系统进行资源分配和调度的一个独立单位。

2、线程是进程的一个实体,是CPU调度和分派的基本单位, 它是比进程更小的能独立运行的基本单位.线程自己基本上 不拥有系统资源,只拥有一点在运行中必不可少的资源(如程 序计数器,一组寄存器和栈),但是它可与同属一个进程的其 他的线程共享进程所拥有的全部资源。

**区别：**

**1、**一个程序至少有一个进程,一个进程至少有一个线程.

**2、**线程的划分尺度小于进程(资源比进程少)，使得多线程 程序的并发性高。

**3、**进程在执行过程中拥有独立的内存单元，而多个线程共 享内存，从而极大地提高了程序的运行效率

**4、**线程不能够独立执行，必须依存在进程中

**优缺点：**

线程和进程在使用上各有优缺点：线程执行开销小，但不 利于资源的管理和保护；而进程正相反。

**<5> 同步**

**1. 多线程开发可能遇到的问题**

from threading import Thread

import time

g\_num = 0

def test1():

global g\_num

for i in range(1000000):

g\_num += 1

print("---test1---g\_num=%d"%g\_num)

def test2():

global g\_num

for i in range(1000000):

g\_num += 1

print("---test2---g\_num=%d"%g\_num)

p1 = Thread(target=test1)

p1.start()

p2 = Thread(target=test2)

p2.start()

print("---g\_num=%d---"%g\_num)

问题产生的原因就是没有控制多个线程对同一资源的访 问，对数据造成破坏，使得线程运行的结果不可预期。这种现 象称为“线程不安全”。

**2. 什么是同步**

同步就是协同步调，按预定的先后次序进行运行。如进程、 线程同步，可理解为进程或线程A和B一块配合，A执行到一 定程度时要依靠B的某个结果，于是停下来，示意B运行;B 依言执行，再将结果给A;A再继续操作。

**3. 解决问题的思路**

1、系统调用t1，然后获取到num的值为0，此时上一把 锁，即不允许其他现在操作num

2、对num的值进行+1

3、解锁，此时num的值为1，其他的线程就可以使用num 了，而且是num的值不是0而是1

4、同理其他线程在对num进行修改时，都要先上锁，处理 完后再解锁，在上锁的整个过程中不允许其他线程访问，就保 证了数据的正确性。

**<6> 互斥锁**

某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”，其他的线程才能再次锁定该资源。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

threading模块中定义了Lock类，可以方便的处理锁定：

#创建锁

mutex = threading.Lock()

#锁定

mutex.acquire([blocking])

#释放

mutex.release()

其中，锁定方法acquire可以有一个blocking参数。

1、如果设定blocking为True，则当前线程会堵塞，直到获 取到这个锁为止（如果没有指定，那么默认为True）

2、如果设定blocking为False，则当前线程不会堵塞

from threading import Thread, Lock

import time

g\_num = 0

def test1():

global g\_num

for i in range(1000000):

#True表示堵塞 即如果这个锁在上锁之前已经被上锁了， 那么这个线程会在这里 一直等待到解锁为止

#False表示非堵塞，即不管本次调用能够成功上锁，都不 会卡在这,而是继续执行下面的代码

mutexFlag = mutex.acquire(True)

if mutexFlag:

g\_num += 1

mutex.release()

print("---test1---g\_num=%d"%g\_num)

def test2():

global g\_num

for i in range(1000000):

mutexFlag = mutex.acquire(True) #True表示堵塞

if mutexFlag:

g\_num += 1

mutex.release()

print("---test2---g\_num=%d"%g\_num)

#创建一个互斥锁

#这个所默认是未上锁的状态

mutex = Lock()

p1 = Thread(target=test1)

p1.start()

p2 = Thread(target=test2)

p2.start()

print("---g\_num=%d---"%g\_num)

**上锁解锁过程：**

当一个线程调用锁的acquire()方法获得锁时，锁就进入 “locked”状态。

每次只有一个线程可以获得锁。如果此时另一个线程试图 获得这个锁，该线程就会变为“blocked”状态，称为“阻塞”，直 到拥有锁的线程调用锁的release()方法释放锁之后，锁进入 “unlocked”状态。

线程调度程序从处于同步阻塞状态的线程中选择一个来 获得锁，并使得该线程进入运行（running）状态。

**总结：**

锁的好处：

确保了某段关键代码只能由一个线程从头到尾完整地 执行

锁的坏处：

阻止了多线程并发执行，包含锁的某段代码实际上只 能以单线程模式执行，效率就大大地下降了

由于可以存在多个锁，不同的线程持有不同的锁，并试 图获取对方持有的锁时，可能会造成死锁

**<7> 多线程-非共享数据**

**在多线程开发中，全局变量是多个线程都共享的数据，而局部变量等是各自线程的，是非共享的。**

import threading

import time

class MyThread(threading.Thread):

# 重写 构造方法

def \_\_init\_\_(self,num,sleepTime):

threading.Thread.\_\_init\_\_(self)

self.num = num

self.sleepTime = sleepTime

def run(self):

self.num += 1

time.sleep(self.sleepTime)

print('线程(%s),num=%d'%(self.name, self.num))

if \_\_name\_\_ == '\_\_main\_\_':

mutex = threading.Lock()

t1 = MyThread(100,5)

t1.start()

t2 = MyThread(200,1)

t2.start()

import threading

from time import sleep

def test(sleepTime):

num=1

sleep(sleepTime)

num+=1

print('---(%s)--num=%d'%(threading.current\_thread(), num))

t1 = threading.Thread(target = test,args=(5,))

t2 = threading.Thread(target = test,args=(1,))

t1.start()

t2.start()

**<8> 死锁**

**1. 死锁**

在线程间共享多个资源的时候，如果两个线程分别占有一 部分资源并且同时等待对方的资源，就会造成死锁。

import threading

import time

class MyThread1(threading.Thread):

def run(self):

if mutexA.acquire():

print(self.name+'----do1---up----')

time.sleep(1)

if mutexB.acquire():

print(self.name+'----do1---down----')

mutexB.release()

mutexA.release()

class MyThread2(threading.Thread):

def run(self):

if mutexB.acquire():

print(self.name+'----do2---up----')

time.sleep(1)

if mutexA.acquire():

print(self.name+'----do2---down----')

mutexA.release()

mutexB.release()

mutexA = threading.Lock()

mutexB = threading.Lock()

if \_\_name\_\_ == '\_\_main\_\_':

t1 = MyThread1()

t2 = MyThread2()

t1.start()

t2.start()

**2. 避免死锁**

**1、**程序设计时要尽量避免（银行家算法）

**2、**添加超时时间等

**<9> 同步应用（多线程有序执行）**

from threading import Thread,Lock

from time import sleep

class Task1(Thread):

def run(self):

while True:

if lock1.acquire():

print("------Task 1 -----")

sleep(0.5)

lock2.release()

class Task2(Thread):

def run(self):

while True:

if lock2.acquire():

print("------Task 2 -----")

sleep(0.5)

lock3.release()

class Task3(Thread):

def run(self):

while True:

if lock3.acquire():

print("------Task 3 -----")

sleep(0.5)

lock1.release()

#使用Lock创建出的锁默认没有“锁上”

lock1 = Lock()

#创建另外一把锁，并且“锁上”

lock2 = Lock()

lock2.acquire()

#创建另外一把锁，并且“锁上”

lock3 = Lock()

lock3.acquire()

t1 = Task1()

t2 = Task2()

t3 = Task3()

t1.start()

t2.start()

t3.start()

**<10> 生产者与消费者模式**

**1. 队列**

先进先出

**2. 栈**

先进后出

Python的Queue模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出)队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列PriorityQueue。这些队列都实现了锁原语（可以理解为原子操作，即要么不做，要么就做完），能够在多线程中直接使用。可以使用队列来实现线程间的同步。

import threading

import time

#python2中

from Queue import Queue

#python3中

# from queue import Queue

class Producer(threading.Thread):

def run(self):

global queue

count = 0

while True:

if queue.qsize() < 1000:

for i in range(100):

count = count +1

msg = '生成产品'+str(count)

queue.put(msg)

print(msg)

time.sleep(0.5)

class Consumer(threading.Thread):

def run(self):

global queue

while True:

if queue.qsize() > 100:

for i in range(3):

msg = self.name + '消费了 '+queue.get()

print(msg)

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

queue = Queue()

for i in range(500):

queue.put('初始产品'+str(i))

for i in range(2):

p = Producer()

p.start()

for i in range(5):

c = Consumer()

c.start()

**3. Queue的说明**

1、对于Queue，在多线程通信之间扮演重要的角色

2、添加数据到队列中，使用put()方法

3、从队列中取数据，使用get()方法

4、判断队列中是否还有数据，使用qsize()方法

**4. 生产者消费者模式的说明**

**为什么要使用生产者和消费者模式：**

在线程世界里，生产者就是生产数据的线程，消费者就是 消费数据的线程。在多线程开发当中，如果生产者处理速度很 快，而消费者处理速度很慢，那么生产者就必须等待消费者处 理完，才能继续生产数据。同样的道理，如果消费者的处理能 力大于生产者，那么消费者就必须等待生产者。为了解决这个 问题于是引入了生产者和消费者模式。

**什么是生产者消费者模式：**

生产者消费者模式是通过一个容器来解决生产者和消费 者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通 过阻塞队列来进行通讯，所以生产者生产完数据之后不用等 待消费者处理，直接扔给阻塞队列，消费者不找生产者要数 据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲 区，平衡了生产者和消费者的处理能力。

这个阻塞队列就是用来给生产者和消费者解耦的。纵观大 多数设计模式，都会找一个第三者出来进行解耦。

**<11> ThreadLocal**

**1. 使用函数传参的方法**

def process\_student(name):

std = Student(name)

# std是局部变量，但是每个函数都要用它，因此必须传进 去：

do\_task\_1(std)

do\_task\_2(std)

def do\_task\_1(std):

do\_subtask\_1(std)

do\_subtask\_2(std)

def do\_task\_2(std):

do\_subtask\_2(std)

do\_subtask\_2(std)

**2. 使用全局字典的方法**

global\_dict = {}

def std\_thread(name):

std = Student(name)

# 把std放到全局变量global\_dict中：

global\_dict[threading.current\_thread()] = std

do\_task\_1()

do\_task\_2()

def do\_task\_1():

# 不传入std，而是根据当前线程查找：

std = global\_dict[threading.current\_thread()]

...

def do\_task\_2():

# 任何函数都可以查找出当前线程的std变量：

std = global\_dict[threading.current\_thread()]

...

**3. 使用ThreadLocal的方法**

import threading

# 创建全局ThreadLocal对象:

local\_school = threading.local()

def process\_student():

# 获取当前线程关联的student:

std = local\_school.student

print('Hello, %s (in %s)' % (std, threading.current\_thread().name))

def process\_thread(name):

# 绑定ThreadLocal的student:

local\_school.student = name

process\_student()

t1 = threading.Thread(target= process\_thread, args=('dongGe',), name='Thread-A')

t2 = threading.Thread(target= process\_thread, args=(' 老王',), name='Thread-B')

t1.start()

t2.start()

t1.join()

t2.join()

全局变量local\_school就是一个ThreadLocal对象，每 个Thread对它都可以读写student属性，但互不影响。你可 以把local\_school看成全局变量，但每个属性如 local\_school.student都是线程的局部变量，可以任意读写 而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

可以理解为全局变量local\_school是一个dict，不但可 以用local\_school.student，还可以绑定其他变量，如 local\_school.teacher等等。

ThreadLocal最常用的地方就是为每个线程绑定一个数据 库连接，HTTP请求，用户身份信息等，这样一个线程的所有 调用到的处理函数都可以非常方便地访问这些资源。

**4. 小结**

一个ThreadLocal变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。ThreadLocal解决了参数在一个线程中各个函数之间互相传递的问题

**<12> 异步**

1、同步调用就是你 喊 你朋友吃饭 ，你朋友在忙 ，你就一直在那等，等你朋友忙完了 ，你们一起去

2、异步调用就是你 喊 你朋友吃饭 ，你朋友说知道了 ，待会忙完去找你 ，你就去做别的了。

from multiprocessing import Pool

import time

import os

def test():

print("---进程池中的进程---pid=%d,ppid=%d-- "%(os.getpid(),os.getppid()))

for i in range(3):

print("----%d---"%i)

time.sleep(1)

return "hahah"

def test2(args):

print("---callback func--pid=%d"%os.getpid())

print("---callback func--args=%s"%args)

pool = Pool(3)

pool.apply\_async(func=test,callback=test2)

time.sleep(5)

print("----主进程-pid=%d----"%os.getpid())

**<13> GIL问题**

在python中，GIL相当于一把大锁，控制线程访问数据，因此python中的多线程，其实等同于单线程。可以通过使用其他语言改写特殊的代码，来解决GIL的问题。