# 线程和进程

程序：保存在计算机中的代码

进程：运行起来的程序即为进程，从操作系统中获得了资源

线程：程序运行时代码执行的进度，cpu调度的最小单位

并发：利用时间片，同时运行

并行：多个cpu同时进行

串行：单个cpu依次运行

# 进程

## 1/fork

ret = os.fork()

if ret>0:

父进程

else:

子进程

Os.fork会返回2个值一个0，一个大于0，其中大于零的时父进程，等于0的时子进程，父类进程不会等待子进程结束

## 2/process

from multiprocessing import Process

import time

def test():

for i in range(5):

print("---test---")

time.sleep(1)

p = Process(target=test, args=(100,))#这里一定要加,

p.start() #让这个进程开始执行test函数里的代码

p.join()

默认不会等待子进程结束，可设置参数

**.join([time])** 主进程等待子进程结束，或等待多少秒

**.start()** 启动进程实例

**.run()** 如果没有target方法，则默认调用的方法，这个方法在multiprocessing.process中，可以进行重写

## 3/进程池

from multiprocessing import Pool

import os

import random

import time

def worker(num):

for i in range(5):

print("===pid=%d==num=%d="%(os.getpid(), num))

time.sleep(1)

#3表示 进程池中对多有3个进程一起执行

pool = Pool(3)

for i in range(10):

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

#注意：如果添加的任务数量超过了进程池中进程的个数的话，那么不会导致添加不进入

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

#pool.apply(worker, (i,)) 阻塞

pool.close()#关闭进程池，相当于不能够再次添加新任务了

pool.join()

注意：

1/主进程默认**不会等待**进程池中的任务执行完后才结束，而是当主进程的任务做完之后立马结束

2/添加join使主进程等待子进程

## 4/通信

进程之间数据是不共享的，需要进行通信才可以数据共享

**队列->使用process**

From multiprocessing import Queue

Q = Queue(3)#最多存储3条数据

Q.put(“xx”)#添加数据

Q.get()#吐出数据

Q.full()#判断数据是否放满

不同进程之间需要传递Queue对象，如：

from multiprocessing import Process, Queue

def write(q):

**#q是父类的队列对象**

q.put(value)

def read(q):

**#q是父类的队列对象**

value = q.get(True)

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,))

**队列->使用pool**

1/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()，否则会得到一条如下的错误信息：

RuntimeError: Queue objects should only be shared between processes through inheritance.

# 线程

## 1/Thread

.join()

->主线（进程）等待子线程结束，才回到进程。可设置时间

--------------------------------------------------------------------------------

thread模块比较底层，一般使用threading模块

from threading import Thread

t = Thread( target=test，args=(,) ) #test函数名称

t.start() #启动线程

# t.setDaemon(True) # 将t设置为守护进程，主进程结束后，t也会结束

# t.join() # 让主线程 等待 线程test 线程结束，再继续下面的代码，默认主线程不等待

通过类调用--不常用

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) #name属性中保存的是当前线程的名字

print(msg)

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

t = MyThread() #初始化类的时候自动调用

t.start() # start会调用run方法

## 2/互斥锁 | 线程同步

多个线程对同一个资源进行竞争

**from threading import** Thread, **Lock**

import time

g\_num = 0 #实验用计数

**total\_lock = Lock() #创建一把互斥锁，这个锁默认是没有上锁的**

def test1():

global g\_num

#这个线程和ｔｅｓｔ2线程都在抢着　对这个锁　进行上锁，如果有１方成功的上锁，那么导致另外

#一方会堵塞（一直等待）到这个锁被解开为止

**total\_lock.acquire() #上锁**

for i in range(1000000):

g\_num += 1

#运行完对锁进行释放，不然其他线程无法获取资源

**total\_lock.release() # 解锁**

def test2():

global g\_num

total\_lock**.acquire() #上锁**

for i in range(1000000):

g\_num += 1

total\_lock**.release() #解锁**

p1 = Thread(target=test1)

p1.start()

p2 = Thread(target=test2)

p2.start()

--------------------------------------

**from threading import** **Lock #导入锁**

**mutex = Lock() #初始化锁**

**mutex.acquire([timeout]) #加锁**

**mutex.release() #释放锁**

注意：加锁的代码越少约好

## 3/线程池

**1、维护指定数量的线程**

from concurrent.futures import TheadPoolExecutor

excutor = ThreadPoolExecutor(max\_workers=2) # 线程池中有2个

task\_1 = excutor.submit(fn, 2, name, ...)

task\_1.done() # task\_1任务状态，True完成，false未完成

# 取消任务，运行中的程序不能取消，可以取消线程池中未运行的线程

task\_1.cancel()

**2、主线程等待子线程完成**

from concurrent.futures import wait，ALL\_COMPLETED

# 等待所有子线程完成后，主线程结束

wait([task\_1, task\_2], return\_when=ALL\_COMPLETED)

**3、监控是哪一个线程完成**

all\_task = [task\_1, task\_2]

for task in **as\_completed(all\_task)**:

print(task.result()) #线程返回结果

## 4/线程通信

通过把数据信息放到另外的程序中进行通信

from queue import Queue

if \_\_name\_\_ == “\_\_main\_\_”:

# Queue(maxsize=10) 设置队列长度

# 有最大长度，队列放满了，队列就会阻塞

topic\_queue = Queue() #队列1

anthor\_queue = Queue() #队列2

topic\_queue.put(“body”) #向队列中放一个值

topic\_queue.get() #取一个值

topic\_queue.get\_nowait()

topic\_queue.put\_nowait()

# 三、补充

使用队列，栈，线程池缓存数据，处理与生产数据速度不匹配，解耦作用

Import queue import Queue #线程中是使用的队列，与进程中的对类不同

#线程中类似session.localhost的东西，保存数据，实现数据共享

Import threading from local

Local\_p = local() # 创建

Local\_p.one = “xxxx” # 这是一个对象，可以存放函数

# 四、GIL锁

GIL(Global Interpreter Lock全局解释锁),python设计之初，为数据安全（GIL会保证字节码的安全）考虑，每个CPU在同一时间只能执行一个线程。

**GIL特性：**

**#** cpython有GIL，pypy没有GIL

1、python2根据代码行数释放，python3根据时间片释放

2、遇到IO释放GIL锁

由此，python对**高IO系统**，基本没有影响，这个时候会**并发**执行**（利用不了多cpu的特性）**，但对**密集计算**的进程会有很多影响，这时应使用**多进程，**但进程与进程之间通信比较复杂

## 通过C函数调用

使用C语言进行函数的调用，跳过gil锁，涉及到调用第三方函数

GCC编译成动态库文件

Gcc xxx.c -shared -o -libxxx.so

from ctypes import \*

from threading import Thread

#加载动态库

lib = cdll.LoadLibrary("./libdeadloop.so")

#创建一个子线程，让其执行ｃ语言编写的函数，此函数是一个死循环

t = Thread(target=lib.DeadLoop)

t.start()