**多任务的概念**

什么叫“多任务”呢？简单地说，就是操作系统可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听MP3，一边在用Word赶作业，这就是多任务，至少同时有3个任务正在运行。还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。

现在，多核CPU已经非常普及了，但是，即使过去的单核CPU，也可以执行多任务。由于CPU执行代码都是顺序执行的，那么，单核CPU是怎么执行多任务的呢？

答案就是操作系统轮流让各个任务交替执行，任务1执行0.01秒，切换到任务2，任务2执行0.01秒，再切换到任务3，执行0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于CPU的执行速度实在是太快了，我们感觉就像所有任务都在同时执行一样。

真正的并行执行多任务只能在多核CPU上实现，但是，由于任务数量远远多于CPU的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

**注意：**

* 并发：指的是任务数多余cpu核数，通过操作系统的各种任务调度算法，实现用多个任务“一起”执行（实际上总有一些任务不在执行，因为切换任务的速度相当快，看上去一起执行而已）
* 并行：指的是任务数小于等于cpu核数，即任务真的是一起执行的

## 线程

#### python的thread模块是比较底层的模块，python的threading模块是对thread做了一些包装的，可以更加方便的被使用

### 1. 使用threading模块

#### 单线程执行

#coding=utf-8

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()

运行结果：

#### 多线程执行

#coding=utf-8

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

运行结果：

#### 说明

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. 当调用start()时，才会真正的创建线程，并且开始执行

### 2. 主线程会等待所有的子线程结束后才结束

#coding=utf-8

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

#coding=utf-8

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)

## 

## 线程-注意点

#### 1. 线程执行代码的封装

通过上一小节，能够看出，通过使用threading模块能完成多任务的程序开发，为了让每个线程的封装性更完美，所以使用threading模块时，往往会定义一个新的子类class，只要继承threading.Thread就可以了，然后重写run方法

示例如下：

#coding=utf-8

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()

t.start()

### 说明

* 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。

#### 2. 线程的执行顺序

#coding=utf-8

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()

执行结果：(运行的结果可能不一样，但是大体是一致的)

I'm Thread-1 @ 0

I'm Thread-2 @ 0

I'm Thread-5 @ 0

I'm Thread-3 @ 0

I'm Thread-4 @ 0

I'm Thread-3 @ 1

I'm Thread-4 @ 1

I'm Thread-5 @ 1

I'm Thread-1 @ 1

I'm Thread-2 @ 1

I'm Thread-4 @ 2

I'm Thread-5 @ 2

I'm Thread-2 @ 2

I'm Thread-1 @ 2

I'm Thread-3 @ 2

#### 说明

从代码和执行结果我们可以看出，多线程程序的执行顺序是不确定的。当执行到sleep语句时，线程将被阻塞（Blocked），到sleep结束后，线程进入就绪（Runnable）状态，等待调度。而线程调度将自行选择一个线程执行。上面的代码中只能保证每个线程都运行完整个run函数，但是线程的启动顺序、run函数中每次循环的执行顺序都不能确定。

### 3. 总结

1. 每个线程默认有一个名字，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。

**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()

运行结果:

---线程创建之前g\_num is 100---

----in work1, g\_num is 103---

----in work2, g\_num is 103---

**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()

运行结果:

----in work1--- [11, 22, 33, 44]

----in work2--- [11, 22, 33, 44]

**总结：**

* 在一个进程内的所有线程共享全局变量，很方便在多个线程间共享数据
* 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）

## 多线程-共享全局变量问题

### 多线程开发可能遇到的问题

假设两个线程t1和t2都要对全局变量g\_num(默认是0)进行加1运算，t1和t2都各对g\_num加10次，g\_num的最终的结果应该为20。

但是由于是多线程同时操作，有可能出现下面情况：

1. 在g\_num=0时，t1取得g\_num=0。此时系统把t1调度为”sleeping”状态，把t2转换为”running”状态，t2也获得g\_num=0
2. 然后t2对得到的值进行加1并赋给g\_num，使得g\_num=1
3. 然后系统又把t2调度为”sleeping”，把t1转为”running”。线程t1又把它之前得到的0加1后赋值给g\_num。
4. 这样导致虽然t1和t2都对g\_num加1，但结果仍然是g\_num=1

#### 测试1

import threading

import time

g\_num = 0

def work1(num):

global g\_num

for i in range(num):

g\_num += 1

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

def work2(num):

global g\_num

for i in range(num):

g\_num += 1

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

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

t1 = threading.Thread(target=work1, args=(100,))

t1.start()

t2 = threading.Thread(target=work2, args=(100,))

t2.start()

while len(threading.enumerate()) != 1:

time.sleep(1)

print("2个线程对同一个全局变量操作之后的最终结果是:%s" % g\_num)

#### 运行结果：

---线程创建之前g\_num is 0---

----in work1, g\_num is 100---

----in work2, g\_num is 200---

2个线程对同一个全局变量操作之后的最终结果是:200

#### 测试2

import threading

import time

g\_num = 0

def work1(num):

global g\_num

for i in range(num):

g\_num += 1

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

def work2(num):

global g\_num

for i in range(num):

g\_num += 1

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

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

t1 = threading.Thread(target=work1, args=(1000000,))

t1.start()

t2 = threading.Thread(target=work2, args=(1000000,))

t2.start()

while len(threading.enumerate()) != 1:

time.sleep(1)

print("2个线程对同一个全局变量操作之后的最终结果是:%s" % g\_num)

#### 运行结果：

---线程创建之前g\_num is 0---

----in work1, g\_num is 1088005---

----in work2, g\_num is 1286202---

2个线程对同一个全局变量操作之后的最终结果是:1286202

### 结论

* 如果多个线程同时对同一个全局变量操作，会出现资源竞争问题，从而数据结果会不正确

**同步的概念**

同步就是协同步调，按预定的先后次序进行运行。如:你说完，我再说。

"同"字从字面上容易理解为一起动作

其实不是，"同"字应是指协同、协助、互相配合。

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执行，再将结果给A;A再继续操作。

**解决线程同时修改全局变量的方式**

对于上一小节提出的那个计算错误的问题，可以通过线程同步来进行解决

思路，如下:

1. 系统调用t1，然后获取到g\_num的值为0，此时上一把锁，即不允许其他线程操作g\_num
2. t1对g\_num的值进行+1
3. t1解锁，此时g\_num的值为1，其他的线程就可以使用g\_num了，而且是g\_num的值不是0而是1
4. 同理其他线程在对g\_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

**互斥锁**

当多个线程几乎同时修改某一个共享数据的时候，需要进行同步控制

线程同步能够保证多个线程安全访问竞争资源，最简单的同步机制是引入互斥锁。

互斥锁为资源引入一个状态：锁定/非锁定

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()

# 释放

mutex.release()

**注意：**

* 如果这个锁之前是没有上锁的，那么acquire不会堵塞
* 如果在调用acquire对这个锁上锁之前 它已经被 其他线程上了锁，那么此时acquire会堵塞，直到这个锁被解锁为止

**使用互斥锁完成2个线程对同一个全局变量各加100万次的操作**

import threading

import time

g\_num = 0

def test1(num):

global g\_num

for i in range(num):

mutex.acquire() # 上锁

g\_num += 1

mutex.release() # 解锁

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

def test2(num):

global g\_num

for i in range(num):

mutex.acquire() # 上锁

g\_num += 1

mutex.release() # 解锁

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

# 创建一个互斥锁

# 默认是未上锁的状态

mutex = threading.Lock()

# 创建2个线程，让他们各自对g\_num加1000000次

p1 = threading.Thread(target=test1, args=(1000000,))

p1.start()

p2 = threading.Thread(target=test2, args=(1000000,))

p2.start()

# 等待计算完成

while len(threading.enumerate()) != 1:

time.sleep(1)

print("2个线程对同一个全局变量操作之后的最终结果是:%s" % g\_num)

运行结果：

---test1---g\_num=1909909

---test2---g\_num=2000000

2个线程对同一个全局变量操作之后的最终结果是:2000000

可以看到最后的结果，加入互斥锁后，其结果与预期相符。

**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

## 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。

尽管死锁很少发生，但一旦发生就会造成应用的停止响应。下面看一个死锁的例子

#coding=utf-8

import threading

import time

class MyThread1(threading.Thread):

def run(self):

# 对mutexA上锁

mutexA.acquire()

# mutexA上锁后，延时1秒，等待另外那个线程 把mutexB上锁

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

time.sleep(1)

# 此时会堵塞，因为这个mutexB已经被另外的线程抢先上锁了

mutexB.acquire()

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

mutexB.release()

# 对mutexA解锁

mutexA.release()

class MyThread2(threading.Thread):

def run(self):

# 对mutexB上锁

mutexB.acquire()

# mutexB上锁后，延时1秒，等待另外那个线程 把mutexA上锁

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

time.sleep(1)

# 此时会堵塞，因为这个mutexA已经被另外的线程抢先上锁了

mutexA.acquire()

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

mutexA.release()

# 对mutexB解锁

mutexB.release()

mutexA = threading.Lock()

mutexB = threading.Lock()

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

t1 = MyThread1()

t2 = MyThread2()

t1.start()

t2.start()

运行结果：

此时已经进入到了死锁状态，可以使用ctrl-c退出

### 2. 避免死锁

* 程序设计时要尽量避免（银行家算法）
* 添加超时时间等

### 附录-银行家算法

[背景知识]

一个银行家如何将一定数目的资金安全地借给若干个客户，使这些客户既能借到钱完成要干的事，同时银行家又能收回全部资金而不至于破产，这就是银行家问题。这个问题同操作系统中资源分配问题十分相似：银行家就像一个操作系统，客户就像运行的进程，银行家的资金就是系统的资源。

[问题的描述]

一个银行家拥有一定数量的资金，有若干个客户要贷款。每个客户须在一开始就声明他所需贷款的总额。若该客户贷款总额不超过银行家的资金总数，银行家可以接收客户的要求。客户贷款是以每次一个资金单位（如1万RMB等）的方式进行的，客户在借满所需的全部单位款额之前可能会等待，但银行家须保证这种等待是有限的，可完成的。

例如：有三个客户C1，C2，C3，向银行家借款，该银行家的资金总额为10个资金单位，其中C1客户要借9各资金单位，C2客户要借3个资金单位，C3客户要借8个资金单位，总计20个资金单位。某一时刻的状态如图所示。

对于a图的状态，按照安全序列的要求，我们选的第一个客户应满足该客户所需的贷款小于等于银行家当前所剩余的钱款，可以看出只有C2客户能被满足：C2客户需1个资金单位，小银行家手中的2个资金单位，于是银行家把1个资金单位借给C2客户，使之完成工作并归还所借的3个资金单位的钱，进入b图。同理，银行家把4个资金单位借给C3客户，使其完成工作，在c图中，只剩一个客户C1，它需7个资金单位，这时银行家有8个资金单位，所以C1也能顺利借到钱并完成工作。最后（见图d）银行家收回全部10个资金单位，保证不赔本。那麽客户序列{C1，C2，C3}就是个安全序列，按照这个序列贷款，银行家才是安全的。否则的话，若在图b状态时，银行家把手中的4个资金单位借给了C1，则出现不安全状态：这时C1，C3均不能完成工作，而银行家手中又没有钱了，系统陷入僵持局面，银行家也不能收回投资。

综上所述，银行家算法是从当前状态出发，逐个按安全序列检查各客户谁能完成其工作，然后假定其完成工作且归还全部贷款，再进而检查下一个能完成工作的客户，......。如果所有客户都能完成工作，则找到一个安全序列，银行家才是安全的。

## 案例：多任务版udp聊天器

**说明**

* 编写一个有2个线程的程序
* 线程1用来接收数据然后显示
* 线程2用来检测键盘数据然后通过udp发送数据

**要求**

1. 实现上述要求
2. 总结多任务程序的特点

**参考代码:**

import socket

import threading

def send\_msg(udp\_socket):

"""获取键盘数据，并将其发送给对方"""

while True:

# 1. 从键盘输入数据

msg = input("\n请输入要发送的数据:")

# 2. 输入对方的ip地址

dest\_ip = input("\n请输入对方的ip地址:")

# 3. 输入对方的port

dest\_port = int(input("\n请输入对方的port:"))

# 4. 发送数据

udp\_socket.sendto(msg.encode("utf-8"), (dest\_ip, dest\_port))

def recv\_msg(udp\_socket):

"""接收数据并显示"""

while True:

# 1. 接收数据

recv\_msg = udp\_socket.recvfrom(1024)

# 2. 解码

recv\_ip = recv\_msg[1]

recv\_msg = recv\_msg[0].decode("utf-8")

# 3. 显示接收到的数据

print(">>>%s:%s" % (str(recv\_ip), recv\_msg))

def main():

# 1. 创建套接字

udp\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

# 2. 绑定本地信息

udp\_socket.bind(("", 7890))

# 3. 创建一个子线程用来接收数据

t = threading.Thread(target=recv\_msg, args=(udp\_socket,))

t.start()

# 4. 让主线程用来检测键盘数据并且发送

send\_msg(udp\_socket)

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

main()

## 进程以及状态

### 1. 进程

程序：例如xxx.py这是程序，是一个静态的

进程：一个程序运行起来后，代码+用到的资源 称之为进程，它是操作系统分配资源的基本单元。

不仅可以通过线程完成多任务，进程也是可以的

### 2. 进程的状态

工作中，任务数往往大于cpu的核数，即一定有一些任务正在执行，而另外一些任务在等待cpu进行执行，因此导致了有了不同的状态

* 就绪态：运行的条件都已经慢去，正在等在cpu执行
* 执行态：cpu正在执行其功能
* 等待态：等待某些条件满足，例如一个程序sleep了，此时就处于等待态

## 进程的创建-multiprocessing

multiprocessing模块就是跨平台版本的多进程模块，提供了一个Process类来代表一个进程对象，这个对象可以理解为是一个独立的进程，可以执行另外的事情

### 1. 2个while循环一起执行

# -\*- coding:utf-8 -\*-

from multiprocessing import Process

import time

def run\_proc():

"""子进程要执行的代码"""

while True:

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

time.sleep(1)

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

p = Process(target=run\_proc)

p.start()

while True:

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

time.sleep(1)

#### 说明

* 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

### 2. 进程pid

# -\*- coding:utf-8 -\*-

from multiprocessing import Process

import os

import time

def run\_proc():

"""子进程要执行的代码"""

print('子进程运行中，pid=%d...' % os.getpid()) # os.getpid获取当前进程的进程号

print('子进程将要结束...')

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

print('父进程pid: %d' % os.getpid()) # os.getpid获取当前进程的进程号

p = Process(target=run\_proc)

p.start()

### 3. Process语法结构如下：

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

* target：如果传递了函数的引用，可以任务这个子进程就执行这里的代码
* args：给target指定的函数传递的参数，以元组的方式传递
* kwargs：给target指定的函数传递命名参数
* name：给进程设定一个名字，可以不设定
* group：指定进程组，大多数情况下用不到

Process创建的实例对象的常用方法：

* start()：启动子进程实例（创建子进程）
* is\_alive()：判断进程子进程是否还在活着
* join([timeout])：是否等待子进程执行结束，或等待多少秒
* terminate()：不管任务是否完成，立即终止子进程

Process创建的实例对象的常用属性：

* name：当前进程的别名，默认为Process-N，N为从1开始递增的整数
* pid：当前进程的pid（进程号）

### 4. 给子进程指定的函数传递参数

# -\*- coding:utf-8 -\*-

from multiprocessing import Process

import os

from time import sleep

def run\_proc(name, age, \*\*kwargs):

for i in range(10):

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

print(kwargs)

sleep(0.2)

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

p = Process(target=run\_proc, args=('test',18), kwargs={"m":20})

p.start()

sleep(1) # 1秒中之后，立即结束子进程

p.terminate()

p.join()

运行结果:

子进程运行中，name= test,age=18 ,pid=45097...

{'m': 20}

子进程运行中，name= test,age=18 ,pid=45097...

{'m': 20}

子进程运行中，name= test,age=18 ,pid=45097...

{'m': 20}

子进程运行中，name= test,age=18 ,pid=45097...

{'m': 20}

子进程运行中，name= test,age=18 ,pid=45097...

{'m': 20}

### 5. 进程间不同享全局变量

# -\*- coding:utf-8 -\*-

from multiprocessing import Process

import os

import time

nums = [11, 22]

def work1():

"""子进程要执行的代码"""

print("in process1 pid=%d ,nums=%s" % (os.getpid(), nums))

for i in range(3):

nums.append(i)

time.sleep(1)

print("in process1 pid=%d ,nums=%s" % (os.getpid(), nums))

def work2():

"""子进程要执行的代码"""

print("in process2 pid=%d ,nums=%s" % (os.getpid(), nums))

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

p1 = Process(target=work1)

p1.start()

p1.join()

p2 = Process(target=work2)

p2.start()

#### 运行结果:

in process1 pid=11349 ,nums=[11, 22]

in process1 pid=11349 ,nums=[11, 22, 0]

in process1 pid=11349 ,nums=[11, 22, 0, 1]

in process1 pid=11349 ,nums=[11, 22, 0, 1, 2]

in process2 pid=11350 ,nums=[11, 22]

## 进程、线程对比

### 功能

* 进程，能够完成多任务，比如 在一台电脑上能够同时运行多个QQ
* 线程，能够完成多任务，比如 一个QQ中的多个聊天窗口

### 定义的不同

* 进程是系统进行资源分配和调度的一个独立单位.
* 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

* 一个程序至少有一个进程,一个进程至少有一个线程.
* 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。
* 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
* 线线程不能够独立执行，必须依存在进程中
* 可以将进程理解为工厂中的一条流水线，而其中的线程就是这个流水线上的工人

### 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。

## 进程间通信-Queue

Process之间有时需要通信，操作系统提供了很多机制来实现进程间的通信。

### 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，首先用一个小实例来演示一下Queue的工作原理：

#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())

运行结果:

False

True

消息列队已满，现有消息数量:3

消息列队已满，现有消息数量:3

消息1

消息2

消息3

##### 说明

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

我们以Queue为例，在父进程中创建两个子进程，一个往Queue里写数据，一个从Queue里读数据：

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()

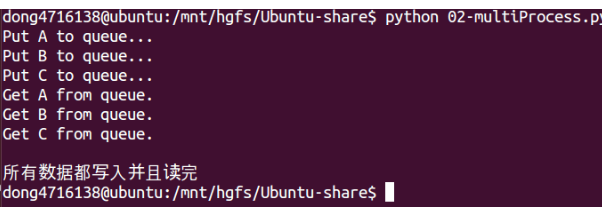
pr.join()

# 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')

运行结果：



## 进程池Pool

当需要创建的子进程数量不多时，可以直接利用multiprocessing中的Process动态成生多个进程，但如果是上百甚至上千个目标，手动的去创建进程的工作量巨大，此时就可以用到multiprocessing模块提供的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，才会用之前的进程来执行新的任务，请看下面的实例：

# -\*- coding:utf-8 -\*-

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-----")

运行结果:

----start----

0开始执行,进程号为21466

1开始执行,进程号为21468

2开始执行,进程号为21467

0 执行完毕，耗时1.01

3开始执行,进程号为21466

2 执行完毕，耗时1.24

4开始执行,进程号为21467

3 执行完毕，耗时0.56

5开始执行,进程号为21466

1 执行完毕，耗时1.68

6开始执行,进程号为21468

4 执行完毕，耗时0.67

7开始执行,进程号为21467

5 执行完毕，耗时0.83

8开始执行,进程号为21466

6 执行完毕，耗时0.75

9开始执行,进程号为21468

7 执行完毕，耗时1.03

8 执行完毕，耗时1.05

9 执行完毕，耗时1.69

-----end-----

multiprocessing.Pool常用函数解析：

* 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；
* 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；

## 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()，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.

下面的实例演示了进程池中的进程如何通信：

# -\*- coding:utf-8 -\*-

# 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 "itcast":

q.put(i)

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

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

q = Manager().Queue() # 使用Manager中的Queue

po = Pool()

po.apply\_async(writer, (q,))

time.sleep(1) # 先让上面的任务向Queue存入数据，然后再让下面的任务开始从中取数据

po.apply\_async(reader, (q,))

po.close()

po.join()

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

运行结果:

(11095) start

writer启动(11097),父进程为(11095)

reader启动(11098),父进程为(11095)

reader从Queue获取到消息：i

reader从Queue获取到消息：t

reader从Queue获取到消息：c

reader从Queue获取到消息：a

reader从Queue获取到消息：s

reader从Queue获取到消息：t

(11095) End

## 应用：文件夹copy器（多进程版）

import multiprocessing

import os

import time

import random

def copy\_file(queue, file\_name,source\_folder\_name, dest\_folder\_name):

"""copy文件到指定的路径"""

f\_read = open(source\_folder\_name + "/" + file\_name, "rb")

f\_write = open(dest\_folder\_name + "/" + file\_name, "wb")

while True:

time.sleep(random.random())

content = f\_read.read(1024)

if content:

f\_write.write(content)

else:

break

f\_read.close()

f\_write.close()

# 发送已经拷贝完毕的文件名字

queue.put(file\_name)

def main():

# 获取要复制的文件夹

source\_folder\_name = input("请输入要复制文件夹名字:")

# 整理目标文件夹

dest\_folder\_name = source\_folder\_name + "[副本]"

# 创建目标文件夹

try:

os.mkdir(dest\_folder\_name)

except:

pass # 如果文件夹已经存在，那么创建会失败

# 获取这个文件夹中所有的普通文件名

file\_names = os.listdir(source\_folder\_name)

# 创建Queue

queue = multiprocessing.Manager().Queue()

# 创建进程池

pool = multiprocessing.Pool(3)

for file\_name in file\_names:

# 向进程池中添加任务

pool.apply\_async(copy\_file, args=(queue, file\_name, source\_folder\_name, dest\_folder\_name))

# 主进程显示进度

pool.close()

all\_file\_num = len(file\_names)

while True:

file\_name = queue.get()

if file\_name in file\_names:

file\_names.remove(file\_name)

copy\_rate = (all\_file\_num-len(file\_names))\*100/all\_file\_num

print("\r%.2f...(%s)" % (copy\_rate, file\_name) + " "\*50, end="")

if copy\_rate >= 100:

break

print()

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

main()

**协程**

# 迭代器

迭代是访问集合元素的一种方式。迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

## 1. 可迭代对象

我们已经知道可以对list、tuple、str等类型的数据使用for...in...的循环语法从其中依次拿到数据进行使用，我们把这样的过程称为遍历，也叫**迭代**。

**但是，是否所有的数据类型都可以放到for...in...的语句中，然后让for...in...每次从中取出一条数据供我们使用，即供我们迭代吗？**

>>> for i in 100:

... print(i)

...

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'int' object is not iterable

>>>

# int整型不是iterable，即int整型不是可以迭代的

# 我们自定义一个容器MyList用来存放数据，可以通过add方法向其中添加数据

>>> class MyList(object):

... def \_\_init\_\_(self):

... self.container = []

... def add(self, item):

... self.container.append(item)

...

>>> mylist = MyList()

>>> mylist.add(1)

>>> mylist.add(2)

>>> mylist.add(3)

>>> for num in mylist:

... print(num)

...

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'MyList' object is not iterable

>>>

# MyList容器的对象也是不能迭代的

我们自定义了一个容器类型MyList，在将一个存放了多个数据的MyList对象放到for...in...的语句中，发现for...in...并不能从中依次取出一条数据返回给我们，也就说我们随便封装了一个可以存放多条数据的类型却并不能被迭代使用。

我们把可以通过for...in...这类语句迭代读取一条数据供我们使用的对象称之为可迭代对象（Iterable）\*\*。

## 2. 如何判断一个对象是否可以迭代

可以使用 isinstance() 判断一个对象是否是 Iterable 对象：

In [50]: from collections import Iterable

In [51]: isinstance([], Iterable)

Out[51]: True

In [52]: isinstance({}, Iterable)

Out[52]: True

In [53]: isinstance('abc', Iterable)

Out[53]: True

In [54]: isinstance(mylist, Iterable)

Out[54]: False

In [55]: isinstance(100, Iterable)

Out[55]: False

## 3. 可迭代对象的本质

我们分析对可迭代对象进行迭代使用的过程，发现每迭代一次（即在for...in...中每循环一次）都会返回对象中的下一条数据，一直向后读取数据直到迭代了所有数据后结束。那么，在这个过程中就应该有一个“人”去记录每次访问到了第几条数据，以便每次迭代都可以返回下一条数据。我们把这个能帮助我们进行数据迭代的“人”称为**迭代器(Iterator)**。

可迭代对象的本质就是可以向我们提供一个这样的中间“人”即迭代器帮助我们对其进行迭代遍历使用。

可迭代对象通过\_\_iter\_\_方法向我们提供一个迭代器，我们在迭代一个可迭代对象的时候，实际上就是先获取该对象提供的一个迭代器，然后通过这个迭代器来依次获取对象中的每一个数据.

那么也就是说，一个具备了\_\_iter\_\_方法的对象，就是一个可迭代对象。

>>> class MyList(object):

... def \_\_init\_\_(self):

... self.container = []

... def add(self, item):

... self.container.append(item)

... def \_\_iter\_\_(self):

... """返回一个迭代器"""

... # 我们暂时忽略如何构造一个迭代器对象

... pass

...

>>> mylist = MyList()

>>> from collections import Iterable

>>> isinstance(mylist, Iterable)

True

>>>

# 这回测试发现添加了\_\_iter\_\_方法的mylist对象已经是一个可迭代对象了

## 4. iter()函数与next()函数

**list、tuple等都是可迭代对象，我们可以通过iter()函数获取这些可迭代对象的迭代器。然后我们可以对获取到的迭代器不断使用next()函数来获取下一条数据。**iter()函数实际上就是调用了可迭代对象的\_\_iter\_\_方法。

>>> li = [11, 22, 33, 44, 55]

>>> li\_iter = iter(li)

>>> next(li\_iter)

11

>>> next(li\_iter)

22

>>> next(li\_iter)

33

>>> next(li\_iter)

44

>>> next(li\_iter)

55

>>> next(li\_iter)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

>>>

**注意，当我们已经迭代完最后一个数据之后，再次调用next()函数会抛出StopIteration的异常，来告诉我们所有数据都已迭代完成，不用再执行next()函数了。**

## 5. 如何判断一个对象是否是迭代器

可以使用 isinstance() 判断一个对象是否是 Iterator 对象：

In [56]: from collections import Iterator

In [57]: isinstance([], Iterator)

Out[57]: False

In [58]: isinstance(iter([]), Iterator)

Out[58]: True

In [59]: isinstance(iter("abc"), Iterator)

Out[59]: True

## 6. 迭代器Iterator

通过上面的分析，我们已经知道，迭代器是用来帮助我们记录每次迭代访问到的位置，当我们对迭代器使用next()函数的时候，迭代器会向我们返回它所记录位置的下一个位置的数据。实际上，在使用next()函数的时候，调用的就是迭代器对象的\_\_next\_\_方法（Python3中是对象的\_\_next\_\_方法，Python2中是对象的next()方法）。**所以，我们要想构造一个迭代器，就要实现它的\_\_next\_\_方法**。但这还不够，python要求迭代器本身也是可迭代的，所以我们还要为迭代器实现\_\_iter\_\_方法，而\_\_iter\_\_方法要返回一个迭代器，迭代器自身正是一个迭代器，所以迭代器的\_\_iter\_\_方法返回自身即可。

**一个实现了\_\_iter\_\_方法和\_\_next\_\_方法的对象，就是迭代器。**

class MyList(object):

"""自定义的一个可迭代对象"""

def \_\_init\_\_(self):

self.items = []

def add(self, val):

self.items.append(val)

def \_\_iter\_\_(self):

myiterator = MyIterator(self)

return myiterator

class MyIterator(object):

"""自定义的供上面可迭代对象使用的一个迭代器"""

def \_\_init\_\_(self, mylist):

self.mylist = mylist

# current用来记录当前访问到的位置

self.current = 0

def \_\_next\_\_(self):

if self.current < len(self.mylist.items):

item = self.mylist.items[self.current]

self.current += 1

return item

else:

raise StopIteration

def \_\_iter\_\_(self):

return self

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

mylist = MyList()

mylist.add(1)

mylist.add(2)

mylist.add(3)

mylist.add(4)

mylist.add(5)

for num in mylist:

print(num)

## 7. for...in...循环的本质

for **item** in **Iterable** 循环的本质就是先通过iter()函数获取可迭代对象Iterable的迭代器，然后对获取到的迭代器不断调用next()方法来获取下一个值并将其赋值给item，当遇到StopIteration的异常后循环结束。

## 8. 迭代器的应用场景

我们发现迭代器最核心的功能就是可以通过next()函数的调用来返回下一个数据值。如果每次返回的数据值不是在一个已有的数据集合中读取的，而是通过程序按照一定的规律计算生成的，那么也就意味着可以不用再依赖一个已有的数据集合，也就是说不用再将所有要迭代的数据都一次性缓存下来供后续依次读取，这样可以节省大量的存储（内存）空间。

举个例子，比如，数学中有个著名的斐波拉契数列（Fibonacci），数列中第一个数为0，第二个数为1，其后的每一个数都可由前两个数相加得到：

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

现在我们想要通过for...in...循环来遍历迭代斐波那契数列中的前n个数。那么这个斐波那契数列我们就可以用迭代器来实现，每次迭代都通过数学计算来生成下一个数。

class FibIterator(object):

"""斐波那契数列迭代器"""

def \_\_init\_\_(self, n):

"""

:param n: int, 指明生成数列的前n个数

"""

self.n = n

# current用来保存当前生成到数列中的第几个数了

self.current = 0

# num1用来保存前前一个数，初始值为数列中的第一个数0

self.num1 = 0

# num2用来保存前一个数，初始值为数列中的第二个数1

self.num2 = 1

def \_\_next\_\_(self):

"""被next()函数调用来获取下一个数"""

if self.current < self.n:

num = self.num1

self.num1, self.num2 = self.num2, self.num1+self.num2

self.current += 1

return num

else:

raise StopIteration

def \_\_iter\_\_(self):

"""迭代器的\_\_iter\_\_返回自身即可"""

return self

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

fib = FibIterator(10)

for num in fib:

print(num, end=" ")

## 9. 并不是只有for循环能接收可迭代对象

除了for循环能接收可迭代对象，list、tuple等也能接收。

li = list(FibIterator(15))

print(li)

tp = tuple(FibIterator(6))

print(tp)

# 生成器

## 1. 生成器

利用迭代器，我们可以在每次迭代获取数据（通过next()方法）时按照特定的规律进行生成。但是我们在实现一个迭代器时，关于当前迭代到的状态需要我们自己记录，进而才能根据当前状态生成下一个数据。为了达到记录当前状态，并配合next()函数进行迭代使用，我们可以采用更简便的语法，即**生成器(generator)。生成器是一类特殊的迭代器**。

## 2. 创建生成器方法1

要创建一个生成器，有很多种方法。第一种方法很简单，只要把一个列表生成式的 [ ] 改成 ( )

In [15]: L = [ x\*2 for x in range(5)]

In [16]: L

Out[16]: [0, 2, 4, 6, 8]

In [17]: G = ( x\*2 for x in range(5))

In [18]: G

Out[18]: <generator object <genexpr> at 0x7f626c132db0>

In [19]:

创建 L 和 G 的区别仅在于最外层的 [ ] 和 ( ) ， L 是一个列表，而 G 是一个生成器。我们可以直接打印出列表L的每一个元素，而对于生成器G，我们可以按照迭代器的使用方法来使用，即可以通过next()函数、for循环、list()等方法使用。

In [19]: next(G)

Out[19]: 0

In [20]: next(G)

Out[20]: 2

In [21]: next(G)

Out[21]: 4

In [22]: next(G)

Out[22]: 6

In [23]: next(G)

Out[23]: 8

In [24]: next(G)

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

StopIteration Traceback (most recent call last)

<ipython-input-24-380e167d6934> in <module>()

----> 1 next(G)

StopIteration:

In [25]:

In [26]: G = ( x\*2 for x in range(5))

In [27]: for x in G:

....: print(x)

....:

0

2

4

6

8

In [28]:

## 3. 创建生成器方法2

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的 for 循环无法实现的时候，还可以用函数来实现。

我们仍然用上一节提到的斐波那契数列来举例，回想我们在上一节用迭代器的实现方式：

class FibIterator(object):

"""斐波那契数列迭代器"""

def \_\_init\_\_(self, n):

"""

:param n: int, 指明生成数列的前n个数

"""

self.n = n

# current用来保存当前生成到数列中的第几个数了

self.current = 0

# num1用来保存前前一个数，初始值为数列中的第一个数0

self.num1 = 0

# num2用来保存前一个数，初始值为数列中的第二个数1

self.num2 = 1

def \_\_next\_\_(self):

"""被next()函数调用来获取下一个数"""

if self.current < self.n:

num = self.num1

self.num1, self.num2 = self.num2, self.num1+self.num2

self.current += 1

return num

else:

raise StopIteration

def \_\_iter\_\_(self):

"""迭代器的\_\_iter\_\_返回自身即可"""

return self

注意，在用迭代器实现的方式中，我们要借助几个变量(n、current、num1、num2)来保存迭代的状态。现在我们用生成器来实现一下。

In [30]: def fib(n):

....: current = 0

....: num1, num2 = 0, 1

....: while current < n:

....: num = num1

....: num1, num2 = num2, num1+num2

....: current += 1

....: yield num

....: return 'done'

....:

In [31]: F = fib(5)

In [32]: next(F)

Out[32]: 1

In [33]: next(F)

Out[33]: 1

In [34]: next(F)

Out[34]: 2

In [35]: next(F)

Out[35]: 3

In [36]: next(F)

Out[36]: 5

In [37]: next(F)

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

StopIteration Traceback (most recent call last)

<ipython-input-37-8c2b02b4361a> in <module>()

----> 1 next(F)

StopIteration: done

在使用生成器实现的方式中，我们将原本在迭代器\_\_next\_\_方法中实现的基本逻辑放到一个函数中来实现，但是将每次迭代返回数值的return换成了yield，此时新定义的函数便不再是函数，而是一个**生成器**了。简单来说：只要在def中有yield关键字的 就称为 生成器

此时按照调用函数的方式( 案例中为F = fib(5) )使用生成器就不再是执行函数体了，而是会返回一个生成器对象（ 案例中为F ），然后就可以按照使用迭代器的方式来使用生成器了。

In [38]: for n in fib(5):

....: print(n)

....:

1

1

2

3

5

In [39]:

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

In [39]: g = fib(5)

In [40]: while True:

....: try:

....: x = next(g)

....: print("value:%d"%x)

....: except StopIteration as e:

....: print("生成器返回值:%s"%e.value)

....: break

....:

value:1

value:1

value:2

value:3

value:5

生成器返回值:done

In [41]:

## 总结

* 使用了yield关键字的函数不再是函数，而是生成器。（使用了yield的函数就是生成器）
* yield关键字有两点作用：
  + 保存当前运行状态（断点），然后暂停执行，即将生成器（函数）挂起
  + 将yield关键字后面表达式的值作为返回值返回，此时可以理解为起到了return的作用
* 可以使用next()函数让生成器从断点处继续执行，即唤醒生成器（函数）
* Python3中的生成器可以使用return返回最终运行的返回值，而Python2中的生成器不允许使用return返回一个返回值（即可以使用return从生成器中退出，但return后不能有任何表达式）。

## 4. 使用send唤醒

我们除了可以使用next()函数来唤醒生成器继续执行外，还可以使用send()函数来唤醒执行。使用send()函数的一个好处是可以在唤醒的同时向断点处传入一个附加数据。

例子：执行到yield时，gen函数作用暂时保存，返回i的值; temp接收下次c.send("python")，send发送过来的值，c.next()等价c.send(None)

In [10]: def gen():

....: i = 0

....: while i<5:

....: temp = yield i

....: print(temp)

....: i+=1

....:

#### 使用send

In [43]: f = gen()

In [44]: next(f)

Out[44]: 0

In [45]: f.send('haha')

haha

Out[45]: 1

In [46]: next(f)

None

Out[46]: 2

In [47]: f.send('haha')

haha

Out[47]: 3

In [48]:

#### 使用next函数

In [11]: f = gen()

In [12]: next(f)

Out[12]: 0

In [13]: next(f)

None

Out[13]: 1

In [14]: next(f)

None

Out[14]: 2

In [15]: next(f)

None

Out[15]: 3

In [16]: next(f)

None

Out[16]: 4

In [17]: next(f)

None

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

StopIteration Traceback (most recent call last)

<ipython-input-17-468f0afdf1b9> in <module>()

----> 1 next(f)

StopIteration:

#### 使用\_\_next\_\_()方法（不常使用）

In [18]: f = gen()

In [19]: f.\_\_next\_\_()

Out[19]: 0

In [20]: f.\_\_next\_\_()

None

Out[20]: 1

In [21]: f.\_\_next\_\_()

None

Out[21]: 2

In [22]: f.\_\_next\_\_()

None

Out[22]: 3

In [23]: f.\_\_next\_\_()

None

Out[23]: 4

In [24]: f.\_\_next\_\_()

None

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

StopIteration Traceback (most recent call last)

<ipython-input-24-39ec527346a9> in <module>()

----> 1 f.\_\_next\_\_()

StopIteration:

# 协程

协程，又称微线程，纤程。英文名Coroutine。

### 协程是啥

协程是python个中另外一种实现多任务的方式，只不过比线程更小占用更小执行单元（理解为需要的资源）。 为啥说它是一个执行单元，因为它自带CPU上下文。这样只要在合适的时机， 我们可以把一个协程 切换到另一个协程。 只要这个过程中保存或恢复 CPU上下文那么程序还是可以运行的。

通俗的理解：在一个线程中的某个函数，可以在任何地方保存当前函数的一些临时变量等信息，然后切换到另外一个函数中执行，注意不是通过调用函数的方式做到的，并且切换的次数以及什么时候再切换到原来的函数都由开发者自己确定

### 协程和线程差异

在实现多任务时, 线程切换从系统层面远不止保存和恢复 CPU上下文这么简单。 操作系统为了程序运行的高效性每个线程都有自己缓存Cache等等数据，操作系统还会帮你做这些数据的恢复操作。 所以线程的切换非常耗性能。但是协程的切换只是单纯的操作CPU的上下文，所以一秒钟切换个上百万次系统都抗的住。

### 简单实现协程

import time

def work1():

while True:

print("----work1---")

yield

time.sleep(0.5)

def work2():

while True:

print("----work2---")

yield

time.sleep(0.5)

def main():

w1 = work1()

w2 = work2()

while True:

next(w1)

next(w2)

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

main()

运行结果：

----work1---

----work2---

----work1---

----work2---

----work1---

----work2---

----work1---

----work2---

----work1---

----work2---

----work1---

----work2---

...省略...

# greenlet

为了更好使用协程来完成多任务，python中的greenlet模块对其封装，从而使得切换任务变的更加简单

### 安装方式

使用如下命令安装greenlet模块:

sudo pip3 install greenlet

#coding=utf-8

from greenlet import greenlet

import time

def test1():

while True:

print "---A--"

gr2.switch()

time.sleep(0.5)

def test2():

while True:

print "---B--"

gr1.switch()

time.sleep(0.5)

gr1 = greenlet(test1)

gr2 = greenlet(test2)

#切换到gr1中运行

gr1.switch()

## 运行效果

---A--

---B--

---A--

---B--

---A--

---B--

---A--

---B--

...省略...

# gevent

greenlet已经实现了协程，但是这个还的人工切换，是不是觉得太麻烦了，不要捉急，python还有一个比greenlet更强大的并且能够自动切换任务的模块gevent

其原理是当一个greenlet遇到IO(指的是input output 输入输出，比如网络、文件操作等)操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。

由于IO操作非常耗时，经常使程序处于等待状态，有了gevent为我们自动切换协程，就保证总有greenlet在运行，而不是等待IO

### 安装

pip3 install gevent

### 1. gevent的使用

import gevent

def f(n):

for i in range(n):

print(gevent.getcurrent(), i)

g1 = gevent.spawn(f, 5)

g2 = gevent.spawn(f, 5)

g3 = gevent.spawn(f, 5)

g1.join()

g2.join()

g3.join()

运行结果

<Greenlet at 0x10e49f550: f(5)> 0

<Greenlet at 0x10e49f550: f(5)> 1

<Greenlet at 0x10e49f550: f(5)> 2

<Greenlet at 0x10e49f550: f(5)> 3

<Greenlet at 0x10e49f550: f(5)> 4

<Greenlet at 0x10e49f910: f(5)> 0

<Greenlet at 0x10e49f910: f(5)> 1

<Greenlet at 0x10e49f910: f(5)> 2

<Greenlet at 0x10e49f910: f(5)> 3

<Greenlet at 0x10e49f910: f(5)> 4

<Greenlet at 0x10e49f4b0: f(5)> 0

<Greenlet at 0x10e49f4b0: f(5)> 1

<Greenlet at 0x10e49f4b0: f(5)> 2

<Greenlet at 0x10e49f4b0: f(5)> 3

<Greenlet at 0x10e49f4b0: f(5)> 4

可以看到，3个greenlet是依次运行而不是交替运行

### 2. gevent切换执行

import gevent

def f(n):

for i in range(n):

print(gevent.getcurrent(), i)

#用来模拟一个耗时操作，注意不是time模块中的sleep

gevent.sleep(1)

g1 = gevent.spawn(f, 5)

g2 = gevent.spawn(f, 5)

g3 = gevent.spawn(f, 5)

g1.join()

g2.join()

g3.join()

运行结果

<Greenlet at 0x7fa70ffa1c30: f(5)> 0

<Greenlet at 0x7fa70ffa1870: f(5)> 0

<Greenlet at 0x7fa70ffa1eb0: f(5)> 0

<Greenlet at 0x7fa70ffa1c30: f(5)> 1

<Greenlet at 0x7fa70ffa1870: f(5)> 1

<Greenlet at 0x7fa70ffa1eb0: f(5)> 1

<Greenlet at 0x7fa70ffa1c30: f(5)> 2

<Greenlet at 0x7fa70ffa1870: f(5)> 2

<Greenlet at 0x7fa70ffa1eb0: f(5)> 2

<Greenlet at 0x7fa70ffa1c30: f(5)> 3

<Greenlet at 0x7fa70ffa1870: f(5)> 3

<Greenlet at 0x7fa70ffa1eb0: f(5)> 3

<Greenlet at 0x7fa70ffa1c30: f(5)> 4

<Greenlet at 0x7fa70ffa1870: f(5)> 4

<Greenlet at 0x7fa70ffa1eb0: f(5)> 4

### 3. 给程序打补丁

from gevent import monkey

import gevent

import random

import time

def coroutine\_work(coroutine\_name):

for i in range(10):

print(coroutine\_name, i)

time.sleep(random.random())

gevent.joinall([

gevent.spawn(coroutine\_work, "work1"),

gevent.spawn(coroutine\_work, "work2")

])

运行结果

work1 0

work1 1

work1 2

work1 3

work1 4

work1 5

work1 6

work1 7

work1 8

work1 9

work2 0

work2 1

work2 2

work2 3

work2 4

work2 5

work2 6

work2 7

work2 8

work2 9

from gevent import monkey

import gevent

import random

import time

# 有耗时操作时需要

monkey.patch\_all() # 将程序中用到的耗时操作的代码，换为gevent中自己实现的模块

def coroutine\_work(coroutine\_name):

for i in range(10):

print(coroutine\_name, i)

time.sleep(random.random())

gevent.joinall([

gevent.spawn(coroutine\_work, "work1"),

gevent.spawn(coroutine\_work, "work2")

])

运行结果

work1 0

work2 0

work1 1

work1 2

work1 3

work2 1

work1 4

work2 2

work1 5

work2 3

work1 6

work1 7

work1 8

work2 4

work2 5

work1 9

work2 6

work2 7

work2 8

work2 9

## 进程、线程、协程对比

#### 请仔细理解如下的通俗描述

* 有一个老板想要开个工厂进行生产某件商品（例如剪子）
* 他需要花一些财力物力制作一条生产线，这个生产线上有很多的器件以及材料这些所有的 为了能够生产剪子而准备的资源称之为：进程
* 只有生产线是不能够进行生产的，所以老板的找个工人来进行生产，这个工人能够利用这些材料最终一步步的将剪子做出来，这个来做事情的工人称之为：线程
* 这个老板为了提高生产率，想到3种办法：
  1. 在这条生产线上多招些工人，一起来做剪子，这样效率是成倍増长，即单进程 多线程方式
  2. 老板发现这条生产线上的工人不是越多越好，因为一条生产线的资源以及材料毕竟有限，所以老板又花了些财力物力购置了另外一条生产线，然后再招些工人这样效率又再一步提高了，即多进程 多线程方式
  3. 老板发现，现在已经有了很多条生产线，并且每条生产线上已经有很多工人了（即程序是多进程的，每个进程中又有多个线程），为了再次提高效率，老板想了个损招，规定：如果某个员工在上班时临时没事或者再等待某些条件（比如等待另一个工人生产完谋道工序 之后他才能再次工作） ，那么这个员工就利用这个时间去做其它的事情，那么也就是说：如果一个线程等待某些条件，可以充分利用这个时间去做其它事情，其实这就是：协程方式

#### 简单总结

1. 进程是资源分配的单位
2. 线程是操作系统调度的单位
3. 进程切换需要的资源很最大，效率很低
4. 线程切换需要的资源一般，效率一般（当然了在不考虑GIL的情况下）
5. 协程切换任务资源很小，效率高
6. 多进程、多线程根据cpu核数不一样可能是并行的，但是协程是在一个线程中 所以是并发

## 并发下载器

### 并发下载原理

from gevent import monkey

import gevent

import urllib.request

# 有耗时操作时需要

monkey.patch\_all()

def my\_downLoad(url):

print('GET: %s' % url)

resp = urllib.request.urlopen(url)

data = resp.read()

print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([

gevent.spawn(my\_downLoad, 'http://www.baidu.com/'),

gevent.spawn(my\_downLoad, 'http://www.itcast.cn/'),

gevent.spawn(my\_downLoad, 'http://www.itheima.com/'),

])

运行结果

GET: http://www.baidu.com/

GET: http://www.itcast.cn/

GET: http://www.itheima.com/

111327 bytes received from http://www.baidu.com/.

172054 bytes received from http://www.itheima.com/.

215035 bytes received from http://www.itcast.cn/.

从上能够看到是先发送的获取baidu的相关信息，然后依次是itcast、itheima，但是收到数据的先后顺序不一定与发送顺序相同，这也就体现出了异步，即不确定什么时候会收到数据，顺序不一定

### 实现多个视频下载

from gevent import monkey

import gevent

import urllib.request

#有IO才做时需要这一句

monkey.patch\_all()

def my\_downLoad(file\_name, url):

print('GET: %s' % url)

resp = urllib.request.urlopen(url)

data = resp.read()

with open(file\_name, "wb") as f:

f.write(data)

print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([

gevent.spawn(my\_downLoad, "1.mp4", 'http://oo52bgdsl.bkt.clouddn.com/05day-08-%E3%80%90%E7%90%86%E8%A7%A3%E3%80%91%E5%87%BD%E6%95%B0%E4%BD%BF%E7%94%A8%E6%80%BB%E7%BB%93%EF%BC%88%E4%B8%80%EF%BC%89.mp4'),

gevent.spawn(my\_downLoad, "2.mp4", 'http://oo52bgdsl.bkt.clouddn.com/05day-03-%E3%80%90%E6%8E%8C%E6%8F%A1%E3%80%91%E6%97%A0%E5%8F%82%E6%95%B0%E6%97%A0%E8%BF%94%E5%9B%9E%E5%80%BC%E5%87%BD%E6%95%B0%E7%9A%84%E5%AE%9A%E4%B9%89%E3%80%81%E8%B0%83%E7%94%A8%28%E4%B8%8B%29.mp4'),

])