第17章多任务编程

计算机的核心是CPU,基于计算机多核的环境下，为了能提高程序执行的效率，Pythoii提 出了 3种实现多任务编程的方式：多进程、多线程、协程。实际上，多任务就是用户可以在同 一时间内运行多个应用程序，也指应用程序可以在同一时间内运行多个任务，多任务编程是影 响程序性能的重要因素。接下来，本章将为大家介绍多任务编程的知识。

*17.1*多任务的实现噱理

操作系统可以同时运行多个任务。例如，计算机在同一时刻既可以使用浏览器上网，又可 以使用酷我播放音乐，还能挂着QQ工具聊天……所有的这些都体现了多任务运行的现象。

现在,多核CPU已经很普及了。即使使用过去的单核CPU,依然能同时运行多个任务,它是如 何办到的呢？答案是操作系统轮流让每个任务交替执行。例如，任务1执行（HHs以后，再切换 到任务2执行（W的……以此重复执行下去，原理如图17』所示。由于CPU运行的速度实在太 快了，任务切换的速度我们的肉眼根本感知不到，所以从表面上来看，所有的任务是在同时运 行的。

C^) (wj (ma) qZj

1 2 3 4 5 6 7 8 单位,

E ] 「 I j E I [ 十宀

QQ 微信 邮箱 …

任务：I （aff） 顷直）fw） （~^） |

} m *\「I*—8 单％"

单核cpu ~ QQ™-I

图17』单核CPU调度任务

图1項 中要执行的任务有QQ、微信、邮箱、百度、酷我等，由于单核CPU每个时间点 只能运行一个任务，所以按照时间片轮转的方式（即任务切换一遍后又重复切换），让每个任 务执行2监（举例值）的时间「从而形成多个任务同时运行的假象。

实际上真正地并行执行多'个任务，只能在多核CPU上完成。不过，计算机要执行的任务 数量非常庞大，远远大于CPU内核的数量.所以操作系统也会自动把任务轮流调度到每个核 心上执行。接下来，使用一张图来分析多核CPU调度任务的原理，如图 以2所示。



任务:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 4 | 5 | 6 | 7 | 8单位：］1S | |
| QQ |  | 邮箱 | 酷我 |  |  |  |  |
| 微信 |  | 百度 |  |  |  |  |  |

CPU单核i

CPU单核2

图17-2双核CPU调度任务

图17・2中，双核CPU相当于计算机有两个单核，每个单核负责执行一个任务，这样就能 同时执行两个任务。当这一批的任务运行一段时间（2応）以后,会再分配另外一批任务继续运行, 这样就真正地形成了多任务的同时运行。

上述延伸出两个概念：并发和并行。从宏观角度上来讲，两者都能够同时处理多个任务， 但是两者又有区别。并行是指两个或者多个任务在同一时刻发生，而并发是指两个或者多个任 务在同一时间间隔内发生。比如在某个时间段中膏有若干个程序都处于已启动运行到运行完毕 之间的状态，某个时刻只能有一个程序运行，这种现象就是并发。

这里面的“同时” “并发”只是一种宏观上的感受，实际上从微观层面看只是进程/线程 的轮换执行，由于切换的时间非常短，所以产生了一起执行的感觉。

W兰进程奇绿

要实现并发，首先需要操作系统的支持。现在的操作系统大部分都是多任务操作系统，可 以“同时”执行多个任务。多任务可以在进程或线程的层面执行，接下来我们先来认识一下进程。

程序是写出来没有被执行的代码，它是一个没有生命的实体，只有处理器赋予程序生命时, 即程序代码被操作系统运行起来，它才能成为一个活动的实体，我们称其为进程。

进程是一个进行中的程序，它不只是程序的代码，还包括当前的活动，通过程序计数器的 值和寄存器的内容来表示。

每个应用程序都有一个自己的进程，操作系统会为这个进程分配地址空间。一般情况下， 包括文本区域、数据区域和堆栈。其中，文本区域用于存储处理器执行的代码，数据区域用来 存储变量和进程执行期间使用的动态分配的内存，堆栈区用来存储活动过程调用的指令和本地 变量。

进程具有以下一些特点：

•动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生，动态 消亡的。

•并发性：任何进程都可以同其他进程一起并发执行。

•独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位。

•异步性？由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不 可预知的速度向前推进。

17.2.2還繼齣撅态 「. 「「「「 . • .

在操作系统的内核中，进程可以标记为"新建"(created)、"就绪” (ready)、"运行” (running ). “阻塞” (blocked)、“挂起” (suspend)和“终止” (terminated)等状态。 这些状态的切换过程如下：

-当程序从存储介质中加载到内存中，进程的状态变为“新建”。

•进程被创建以后，进程调度器会自动把进程的状态设置成“就绪” o此时，进程等待调 度器做上下文切换。只要处理器处于空闲时，调度器会将进程加载到处理器中，进程的 状态变为“运行”，处理器开始执行这个进程的命令。

•如果进程需要等待某个资源(如用户输入等)，进程的状态会被标记为“阻塞”状态。 当进程获得了等待的资源以后，它的状态又会变回“就绪”O

•当内存中的所有进程都处于“阻塞”状态时，系统会把其中一个进程设置为“挂起”状态, 然后将其在内存中的数据保存到磁盘中*，*这样能释放空间给其他进程使用。

•只要进程执行完成或者被操作系统终止，它就会从内存中被移除或者设置为“终止”状态。 为了让大家更好地理解这些状态的转换关系,接下来通过一张图来进行分析,如图17-3 所示。

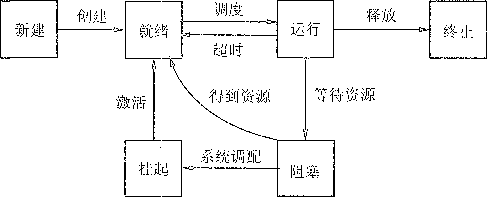


图17-3进程状态的切换

了解进程以后，如何在程序中创建进程呢？在UNIX/Linux操作系统中，坎模块封装了 fb『k()函数供系统调用，可以很轻松地创建进程。

函数的定义格式如下：

os o fork()

执行上述函数会创建一个新的进程(子进程)。与普通函数不同，调用一次这个函数会返 回两次结果，这是因为操作系统自动把当前的进程(称为父进程)复制一份(称为子进程)， 然后分别在父进程和子进程中返回。这个值在子进程中永远返回0,而在父进程中则返回子进 程的IDO

需要注意的是，Windows系统中不能使用forkQ函数。

17.3/1 WforkO

使用fo『k()函数创建子进程，示例如下:

import os

import time process = os a fork() if process==0:

#创建子进程

#子进程走if分支

父进程走else分支

print ( v 子进程--\* )

time sleep (2) print('一 父进程 一 °)

else :

while True:

time.sleep(2)

在上述示例中，调用fork()函数创建了一个子进程，并且把返回的结果赋值给process0若 process的值为0,则打印“子进程”，反之则打印“父进程"。程序运行后成为一个父进程, 父进程在接收到fork命令以后生成一个子进程，就是说此时有两条进程执行接下来的程序。

子进程会返回0给process,所以当子进程执行时会走if语句，父进程会返回大于0的值 给process,所以当父进程执行时会走else语句。由于现在有两个进程在运行程序,所以程序一 直在同时输岀两个分支的打印语句。

程序输出的结果如下：

一父进程一

一子进程一

一子进程一

一父进程一

一子进程一

那么,父进程和子进程哪个先执行呢?子进程和父进程执行的顺序是不固定的,它完全交 由操作系统调度，这个调度除了时间片以外，还有可能受到调度优先级或者其他因素的影响。 (冷多学一招：获取当前迸程的ID

父进程既可以创建子进程，又需要管理子进程。为了便于管理子进程，父进程需要记录每 个子进程的ID,这个表示当前进程的D)可以通过os.getpid()函数进行获取。另外，子进程若 想要获取父进程的ID,则可以调用o&gdppid()函数进行获取。

获取子进程和父进程ID的示例如下：

import os

process = os . fork () # 创建子进程 if process==0:

print (\*我是子进程-%d7父进程是%d! % (os .getpid () , os .getppid ())) #获取父进程的ID

print (，我是父进程-%dz子进程是%d\* % (os .getpid () zprocess) ) #获取当前线程的iD

程序运行的结果如下：

我是父进程-2497,子进程是2498

我是子进程-2498,父进程是2497

(泞注意：

运行程序，若父进程负责执行的任务已经完成了，则程序会直接退出，不会因为子进程没 有结束而出现等待的情况。

龙德魇f◎『女0圖數劎篷缪命琴襪體，

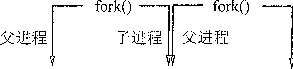
如果在程序中多次使用到fo『k()函数，那么会以怎样的规律来创建新的子进程呢？无论是 父进程还是子进程，只要调用fork()函数，就在这个进程的基础上再增加一条进程，这个进程 会成为父进程，而新增加的进程会成为子进程。

为了让读者更好地理解，接下来，通过一个案例来讲解如何使用fwk()函数来创建多个子 进程，具体代码如下。

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | import os | | |
| 2 | process=os.fork（） | # | 创建子进程 |
| 3 | if process==0: | # | 子进程走这个分支 |
| 4 | print （ T 进程 1' ） |  |  |
| 5 | e 1. s e : | # | 父进程走这个分支 |
| 6 | print （'进程 2 ' ） |  |  |
| 7 | process=os.fork（） | # | 再次创建子进程 |
| 8 | if process==0: |  |  |
| 9 | print （ ,进程 3 \* ） |  |  |
| 10 | else : |  |  |
| 11 | print （ \* 进程 4 \* ） |  |  |

程序在运行以后,只有一个进程从上而下的执行代码,这个进程通常被称为主进程。在第 2行创建了新的子进程,这时下面的程序会交由两个进程执行,如图17・4所示。父进程会执行 else分支的语句*，*子进程会执行if分支的语句。

接着，在第7行又创建了新的进程。此时，运行程序的每条线(进程)各自再拆分为两条线, 也就是说现在有4个进程运行程序，如图17・5所示。此时，两个父进程执行Es©语句，另外两 个子进程执行if语句。因此，程序会输出两遍打印的内容。

进程

—fbrk()—

父进程 子进程

子进程

图17-5进程数量变化(由2变成4 )

进程

fork()—-

父进程 子进程

图17-4进程数量变化(由1变成2 )

程序运行的结果如下:

2 4 1 3 4 3  
程程程程程程  
也彩迷迷迷迷

Wo4通过Process龚剑建进程(跨驱膏)

之前在介绍fok()函数时，提到它不能够在Windows系统中使用，显而易见，这点对 Python的跨平台使用有着极大的局限性。因此a Pythoii提供了更加好用的多进程模块multipro­cessing, 该模块是基于fork机制的,而且支持跨平台操作。这个模块中有个Process类,专门 负责实现与进程相关的功能。

不过,在Windows系统中使用multiprocessing模块时，必须使用“if ―name\_== 的方式运行程序。

17.4J SH Fr©cess饗寰例劎還逓體

Process类对象代表着一个进程，其构造函数的语法格式如下:

class multiprocessing.Process(group=Nonef target=Nonez name=None7 args=(), kwargs={}, \*f daemon^None)

上述函数的参数代表的含义如下:

® group:通常为None,仅仅是为了兼容threading .Thread (线程)。

。target：可调用对象，由mn()方法调用。一般情况下，函数会传给该参数，交给子进程运行。

® name：当前进程的名称。若没有指定，Process-! ? Process-2?…类似这样的名字, 数字会从1开始递增。

* args： tai闵调用的位置参数元组。

® kwargs： target调用的关键字参数字典。

* daemon：进程的守护标志。若提供该参数，则将会设置为True或Falseo若没有提供(默 认为None) 9则该标志将从创建的进程中继承。

创建好进程后，要怎样启动这个进程执行任务呢？调用就酒()方法可以启动进程，该方法 的语法格式如下:

start()

为T让读者更好地理解,下面使用Process类创建子进程，并且调用start()方法启动进程， 具体代码如下：

from multiprocessing import Process # 导入模块

import time

import os

def run\_proc () : #指定子进程负责的任务

while True:

print ( 1子进程运行:pid=%d\* %os . getpid.())

time e sleep(1)

if name ==， main , :

process = Process (target=run\_proc) # 创建子进程

process . start () #启动进程,让这个进程开始执行run\_proc函数的代码

while True:

print ( ,父进程运行:pid.=%d \* %os . getpid ())

time。sleep(1)

上述示例中,声明了一个runjproc()函数,作为子进程待执行的任务。

在if—name ==，—main ，语句中，创建了一个Process实例，并为其指定了任务函数。 一旦调用start()函数启动进程，就会执行这个函数中的代码。

程序运行的结果如下：

父进程运行：pid=2401 子进程运行:pid=2402

子进程运行:pid=2402

父进程运行：pid=2401

17.4.2儼魇』◎饭。為灌Bi歩體體 '

使用fo『k()函数创建子进程时，主进程的结束跟子进程是没有任何关系的。但是，若使用 Process类创建子进程，主进程会等待所有的子进程运行完以后才结束。

示例如下：

from multiprocessing import Process

import time

def test () : #子进程要执行的函数

for i in range(5):

print("--test--") time.sleep(1)

if name == ? main !:

process = Process (target=test) #创建一个子进程,为其指定任务函数为test process . start () #启动子进程,开始执行test函数里的代码

print('--main-™ ,)

上述示例中，定义了任务函数於St。在该函数中，使用fOAill打印了 5次输出语句，并且 每打印1次就睡眠1S。由此说明，子进程执行任务要耗费大于或者等于5S的时间。

在if name == ! main J语句中，程序由主进程运行。在运行期间，创建了一个子进 程，并且调用start()启动子进程，输出表示主进程执行完的打印语句。一旦输岀了这条语句， 就表示主进程执行完成。

程序运行的结果如下：

--main--

--test--

--test--

--test--

从输岀的结果可以看岀，主进程优先执行完任务。不过，主进程并没有立即结束程序，而 是等待所有的子进程结束它才结束。

主进程在执行某条语句时，一直在等待某个条件发生。如果该条件不发生，就会一直卡在 这个地方等待，我们把这种现象称为阻塞。

问题是这么多的子进程，每个子进程执行任务的时长是不确定的，主进程应该阻塞多长时 间合适呢？这时，我们需要让这些进程间同步运行。进程间的同步是指，在多进程的环境下， 进程是并发执行的。因这些进程受到一定条件的制约，而互相发送消息，互相合作、互相等待， 使得每个进程按照一定的速度执行的过程。

为此，多进程模块中提供了 join。方法实现进程间的同步。一旦某个子进程调用了 join。 •方法，就会使主进程阻塞，一直到被调用的进程运行结束或者超时。

join方法的语法格式如下:

join ( [time5out])

上述格式中,timeoirt表示超时时长。若设置timeout的值为None,则主进程将会一直阻塞， 直到调用join()方法的子进程结束;若设置timeout为正数，则会阻塞主进程timeout秒。

需要注意的是，上述方法终止或者方法超时，都会返回None,可以通过检查进程的緜他 code属性以确定是否终止。

使用join()方法同步进程的示例如下：

from multiprocessing import Process

import time

def test () : #子进程要执行的函数

for i in range(5):

print time.sleep(1)

if name == \* main !:

process^Process (target=test) #创建一个子进程,为其指定任务函数为test

process . start () #启动子进程,开始执行test函数里的代码 process “ join(2) print(!--main--!)

上述示例中，定义了任务函数test。在该函数中，使用打印输出5遍语句，每打印1 次就睡眠Is。因此须子进程执行完任务要花费5 s左右。

if \_\_name\_ =语句由主进程运行。在运行期间，首先创建一个子进程，执行 此st任务。然后，调用join方法设置主进程阻塞等待子进程2 s的时间，一旦超过2私主进程 就打印语句。

程序运行的结果如下：

—0 —

——main——

—— 2 ——

—3 —

多学一招：Process类其他的属性和方法

Process类还提供了如下一些属性和方法:

® is\_alive()方法：判断进程对象是否还在执行。

* run()方法：如果在创建进程时，没有« target参数绑定的任务函数，那么这个进程对 象调用start()方法时,将执行该对象的run()方法。
* terminate()方法：无论任务是否执行完成，都会立即终止程序。
* name属性:当前进程的名称,*默认为*Process-N, N为从1开始递增的整数。
* pid属性：当前进程的PID值。

17.4.3禳遊Process罄釁劍邇字遙釋

定义一个继承自Process的子类，每次创建这个自定义类的实例时，就等同于实例化一个 进程对象。

为了让读者更好地理解，接下来，定义一个继承自Process的子类，该类用来计算子进程 执行某个任务要耗费多长的时间。具体代码如下言

from multiprocessing import Process import time import os class Process\_Class(Process): def init (selfinterval):

定义继承自Process的子类

Process. init (self)

完成父类的初始化

间隔秒数

重写了 Process类的run ()方法

self.interval = interval

def run(self):

print ("子进程(%s)开始执行,父进程为(%s)"% (os .getpid() z os .getppid ())) time\_start = time . time () # 返回开始时间的时间戳

time o sleep(self.interval) time\_stop = time . time () # 返回结束时间的时间戳

print (" (%s)执行结束,耗时％ 0.2f 秒"% (os» getpid ()*尸* time\_stop - time\_start))

上述示例中，声明了一个自定义的进程类Process\_Class9 一旦创建了该类的进程对象，就 能够统计该进程执行任务所耗费的时间。

在\_init\_()方法中*，*先完成了父类成员的初始化，再定义了一个表示间隔秒数的interval 属性。

为了记录进程在启动和结束的时间差,需要重写从Process类继承的start()方法。由于该 方法会隐式地调用nm()方法，所以可以重写nin()方法。一旦调用s£art()方法启动进程，就会 自动调用重写的run()方法。

在nm()方法中，记录了进程启动的时间。当程序睡眠了 interval秒后,再次记录进程结束 的时间。

使用上述自定义的进程类Process\_Class,创建一个子进程来执行任务,具体代码如下:

if name ==" main ": print ("当前程序进程(%s) "%os . getpid ()) process = Process\_Class(2) #对一个不包含target参数的Process类执行start ()方法，

#就会运行这个类中的run ()方法，所以这里会执行process . run () process•start()

程序执行的结果如下:

当前程序进程(2827)

子进程(2 82 8)开始执行，父进程为(2827 )

(2828)执行结束，耗时2。00秒

\* 17o5进程池撅暈剑建进程

当要创建的进程数量不多时，可以利用Process类动态地生成多个进程。如果要启动大量 的子进程，显然手动创建多个进程的方式是不可取的，这个工作量是非常巨大的。为此，多进 程模块multiprocessing提供了 Pool (进程池)类，可以批量地创建子进程。

创建Pool类的进程池，可使用如下构造方法：

Pool ( [processes[f initializer[e initargs[, maxtasksperchild[, con­text] ]]]])

上述方法的参数所表示的含义如下:

® processes:使用的工作进程数量。*若设为*None,则会使用os.cpu\_count()返回的数量。

® initializer：若该参数不为None,则每个工作进程将在启动时调用initialize^\*initargs)。

* maxtasksperchild：工作进程退出之前可以完成的任务数，并在完成之后使用新的进程来 替代原进程，以使得闲置的资源被释放。如果该参数设为默认值None,那么只要进程 池存在，工作进程就会一直存活。

® context：用在制定工作进程启动时的上下文。

接下来，使用上述方法创建进程池对象，示例如下：

from multiprocessing import Pool

pool=Pool(10)

上述代码创建一个进程池，指定了最大的进程数量为10。

进程池的内部维护了一个进程序列。当使用进程池时，会到进程池中获取一个工作进程。 如果进程池中的进程数量没有达到最大值，那么会创建一个新的进程来执行任务；如果进程池 序列中没有可供使用的进程，那么程序会等待，直到进程池中有可用的进程为止。

[気？注意：进程池的特点是先创建一定数量的进程(无论用不用)。当程序要使用进程时， *闻擀阳盛妒'*

就直接从进程池中取，用完以后再把进程还给进程池，从而增大了进程的重复使用率。

17SJ 遴糧艇離飓鑑說瀑掘僵翳

创建线程池实例后，可以调用apply\_async()方法给进程添加任务，其语法格式如下： apply\_async(func[, args[, kwds[, callbackerror\_callback]]]]) 上述方法的参数代表的含义如下:

@ fonc：异步执行的函数名称。

* args和kwds： fonc函数的参数。

• callback：回调函数。

® error\_callback：程序失败后指定的回调函数。

上述方法会返回一个AsyncResult类对象。如果callback被指定,则callback可以接收一个 参数后被调用。当返回的AsyncResult类对象准备好回调时会调用callback,调用失败时，则用 error\_callback替换callback, callback应该被立即完成，否则处理结果的线程会被阻塞。

上述方法负责异步的执行任务函数，而且禁止任何阻塞操作。意思就是不用等待当前进程 执行完毕,随时根据系统调度切换进程。

接下来，通过一个案例来演示如何使用进程池非阻塞地添加任务，示例如下：

from multiprocessing import Pool # 从模块中导入 Pool 类

import os

import random

impozrt time

def worker(number):

for i in range(2):

print(1--pid=%d--number=%d'%(os o getpid()r number)) time*。*sleep(1)

*if* \_name == \* main \*:

#创建进程池,指定最大进程数量为2

pool==Pool(2)

for i in range(5):

print(''%i)

pool. apply\_async (worker, (ir ) ) # 添加任务

上述示例中，定义了任务函数worker0在该函数中，通过foHn循环打印两遍语句，并且 每次都要睡眠1 s的时间。

创建了一个至多只有两条进程的进程池，接着给这些进程添加了5个任务，每个任务至少 要花费2 s的时间完成。

程序运行的结果如下：

从运行结果可以看出9即使进程池还未完成执行，在主进程结束后整个程序会退岀。 apply\_async()方法返回的是一个AsyncResult类的对象，该类中有个get()方法，用于返回 收到的结果。虽然apply\_async()方法是非阻塞的，但是get()方法却是阻塞的,因此,我们可 以使用get()阻塞主进程。

如果对apply\_async()方法返回的结果不感兴趣，这时就可以在主进程中使用pooLcloseQ 与pool.join()方法来防止主进程退出。

在上述示例的末尾增加语句，具体如下：

#关闭进程池

pool.。close () pool.j oin()

#主进程等待所有子进程执行完毕

340 w Pjthoo囊戯鑑麗：映嚳尊Pyth©o

上述两个方法的作用如下：

* close()方法；等待所有进程结束后，才关闭进程池。调用close。方法后，就不能再继续 添加新的进程了。
* join()方法：主进程等待所有子进程执行完毕，必须在close()或於rmina\*()之后调用。 再次运行程序，程序运行的结果如下：

—— 0 ——

—1--

—— 2 ——

——3——

—— 4 ——

-~pid=25 95--number=0 --pid=25 96---number=l ~~pid=2 5 95--number=0 ~-pid=25 96---number=l --pid=2596~-number=2 --pid=2595--number=3 --pid=2596-~number=2 -~pid=2595--number=3 --pid=2596~-number=4 --pid=25 96--number:=4

从运行结果可以看出，程序安排了五个任务，进程池中只有两个子进程执行任务，虽然进 程的数量小于任务的数量，但是程序并没有出现堵塞。

实际上，这两个进程会选取前两个任务执行，其余的任务都处于等待的状态。当这 两个任务做完后，会从其余未完成的任务中再取两个任务，交由子进程中执行，如图174 所示。

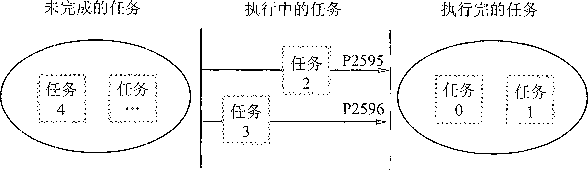


图17-6进程池执行的原理

Python另提供了 apply()方法(从23版以后就不建议再使用了)用来添加任务，主进程会 阻塞于该方法。其语法格式如下：

apply(func[, args[, kwds]])

上述方法中,使用args和kwds参数调用函数.结果返回前会一直阻塞，而且ftmc函数仅 仅被进程池中的一个进程运行。由于这些原因，apply\_async()方法比apply。方法更适合并发 执行。

把173]的示例中添加任务的方法换成apply()方法，具体示例如下:

time.sleep (1)

if name == \* main ': pool==Pool(2)

for i in range(5):

print('--%d~-1%i)

pool.apply(workerz (i,))

程序运行的结果为：

一一o——

~-pid=2870--number=0

--pid=2870--number=0

]

-~pid=2871--number=l

--pid=2871~-number=l

——2 —-

--pid=2870--number=2

-~pid^2870~-number=2

—一3一

--pid=2871--number=3

~-pid=2871--number=3

—4 —

-~pid=2870--number=4

-™pid=287 0---number=4

from multiprocessing import Pool # 从模块中导入 Pool 类

import os

import random

import time def worker (number) : # 进程池指定的函数

for i in range(2):

print('--pid=%d--number=%d'%(os.getpid()z number))

#创建进程池，指定最大进程数量为2

#添加同步函数

从运行的结果可以看出，每个任务都是按照顺序执行的，从第1个任务开始，这个任务运 行完才能轮到下一个任务。由此表明，阻塞式执行任务就是要按照顺序一个接着一个来。

:17o 6適过娜帅iro(匏豔模块刨建进程

从Python 2.4开始，Python引入了 subprocess模块来管理子进程,以取代一些旧模块的方 法，如os.system,它不仅可以调用外部的命令作为子进程，而且可以联系到子进程的输入/输 出/错误管道，获取相关的返回信息。

subprocess模块能执行外部程序，它有如下两种常用的方式创建进程:

B-it： 函数

subprocess模块中定义了多个创建子进程的函数，这些函数分别以不同的方式创建子进程， 为此我们要根据需求从中选择一个使用，具体如下：

(1 )调用subprocess.call()函数运行夕卜部命令

call()函数用于执行一个外部命令，该方法会返回执行的状态码：成功(0 )或错误(非0 )。

(2 )调用 subprocess. check\_call()函数处理错误

通过call()函数会返回一个状态码，这时可以调用check\_call()函数来检测命令的执行结果。 若没有成功,则会返回subprocess.CalledProcessError异常。

(3 )调用subprocess. check\_output()函数捕获输出结果

通过call()函数启动的进程「其标准输入输出会绑定到父进程，调用程序无法获取命令的 输出结果，这时可以调用check\_output()函数来捕获输出。

篥二神:通过Popen类创建进程

实际上，上面的几个函数都是基于Popm()的封装，这些封装的目的在于更容易使用子进程。 当我们想要更个性化需求时，就要转向Popen类，该类生成的对象用来代表子进程。

接下来，本节将围绕着subprocess模块的类和函数进行介绍。

call()函数创建进程，其语法格式如下

subprocess call(args7 \*z stdin=Nonef stdout=Nonez stderr=Nonez shell=Falsef timeout=None)

上述函数会执行args描述的命令，它会一直等到子进程运行结束，并且返回retumcode (子 进程的退出状态)属性。

关于上述函数的参数含义如下:

• args：可以是字符串或者程序参数的序列。如果传递单一的字符串，则shell必须为 True,或者该字符串必须只是简单地指明要执行程序的名称且不带任何参数。

e stdin, stdout, stderr：分别指定程序的标准输入、标准输出和标准错误文件的句柄。

e shell：指定是否通过shell执行指定的命令，默认为屈屁。在Linux系统下，若shell设 为False,贝Popen对象调用os.execvp()来执行args指定的程序;若shell设为True, 且弑gs是字符串9 Popen直接调用系统的shell来执行理s指定的程序；若血11设为 True,且args是个序列，则args的第一项是定义程序命令的字符串，其他项是调用系统 shell时的附加参数。

大多数典型的场景下，这些参数可以安全地保持为默认值。具体示例如下：

>>> import subprocess

>>> code = subprocess.call ( [\* Is 1, 1-a 1])

#跟shell中：Ls -a命令显示的结果\_样

>>> code

0

>>> subprocess.call ( J exit 15 zshell=True)

1

』乙鼠2 ®h p©p®o

subprocess模块中定义了 Popen类，可以使用该类创建进程，并与进程进行复杂的交互。 Pop/类提供如下构造函数，用于创建进程执行系统命令，其语法格式如下：

subprocess o Popen(args, bufsize=-l, executable=None, stdin=None?

stdout=Nonez stderr=Nonef preexec\_fn=None,

close\_fds=True r shell=False z cwd=None f env=None f universal\_newlines=Falsez startupinfo=None7 creat ionflags=0 z restore\_signals=Truer start\_new\_session：=Falser pass\_fds=())

通过上述语法中的参数可以详细定制子进程的环境，不过位置参数args是必须要传人的。 上述函数中的参数，具体含义如下：

窗取gs参数：可以是字符串或者序列类型，用于指定进程的可执行文件及其参数。如果是 序列类型，第一个元素通常是可执行文件的路径，也可以显式地使用executeable#» 指定可执行文件的路径。

° bufsize参数:指定缓冲区的大小。0表ZK不缓冲,1表示按行缓冲，其他正数表示使用 该大小的缓冲，负数表示完全缓冲。

* executable参数：指定一个可执行的程序，一般使用雎s参数设置所要运行的程序。
* Stdiil参数：指定要执行程序的标准输入，默认为键盘。
* stdout参数：指定要执行程序的标准输出，默认为屏幕。
* stderr参数：指定程序的标准错误输出，默认为屏幕。

。preexec\_fh参数:只在UNIX平台下有效,用于指定一个可执行对象,将在子进程运行 之前被调用。

* close\_fds参数：在Windows平台下，若该参数设置为Th©则子进程不会继承任何句柄。 注意：不能设置该参数为Trus并同时通过设置stdin、sfdout或者stderr重定向标准句柄。

. • shell参数？如果设为True,程序将通过shell执行。

* cwd参数：指定子进程运行的工作目录。
* SV参数：指定进程的环境变量。若eiiv为None,子进程的环境变量将从父进程中继承。
* universal\_newlines参数：指定是否使用统一的文本换行符。如果设为True,无论是

UNIX系统下的“\n”，Windows系统下的“\M , Python统一把所有的这些终止符当 作处理。

* startupinfo和creationflags参数：只在Windows系统下有效。这两个参数被传递给底层 的CreateProcess()函数,用于设置进程的一些属性,如主窗口的外观和进程的优先级等。
* restore\_signals参数:如果该参数为True (默认值)，Python将设置为SIG\_IGN的 所有信号在exec之前的子进程中还原到SIG\_DFLO目前包括SIGPIPE. SIGXFZ和 SIGXFSZ 信号。

® start\_new\_session参数:若设为True,则在执行子进程之前,将在子进程中进行setsid() 系统调用。

。pass\_fds参数：可选的文件描述符序列，用于在父进程和子进程之间保持打开状态。若 设置了该参数，则必须让close\_fcis为Trueo

使用Popen类创建子进程的示例如下:

import subprocess

#创建子进程

process child = subprocess.Popen(?ping -c4 www.itcast.cn1,shell==True)

在上述示例中，使用Pops类的构造函数创建了一个进程。其中，在函数中传入的第1个 参数为表示可执行文件的路径的字符串，第2个参数表示通过shell执行程序。

17oT通« Qweue实现进程剛的通儔

每个进程间是需要进行通信的。例如，子进程完成下载后，通知主进程下载完毕。Python 的multiprocessing模块包装了底层的机制，提供了 Queue、Pipes等多种方式来交换数据，其中 Pipes用来在两个进程间通信，Queue用来在多个进程间实现通信，这里只介绍Queue (队列) 的使用。

17.7J 金屬靈最德夢命躍體廓罪裳寥

在多进程中歹每个进程所拥有的数据(包括全局变量)都是独有的歹进程间互不影响。 示例如下：

import os

import time global\_number=100 process=os.fork() if process == 0:

#声明一个全局变量

#创建一个子进程

#子进程走if语句

#让全局变量的值加1

global\_number += 1

print(1--process-1—global\_number=%d5 %global\_number)

else :

time. *sleep* (2) #让程序休眠2秒,这时子进程已经改变了全局变量的值

print(\*--process-2--global\_number=%d'%global\_number)

上述示例中，在子进程的if分支中，对这个全局变量进行了修改，为了保证这个改变在父 进程运行以前，所以让父进程休眠了 2s的时间。

程序运行的结果如下：

-~process-l--global\_number=l01 一一process-2--global\_number=100

从运行的结果可以看出，全局变量在两个进程中的获取到的结果是不一样的。产生这种情 况，原因在于进程间的数据是独立的，互不干扰的，使得进程跟进程之间资源是不共享的。

1X7^2 應隔 Qoeo® OISWSHfiB

每个进程有着自己独立的数据空间，通常不能与其他进程共享。要想在进程间实现资源共 享,可以使用multiprocessoring模块的Queue类(表示队列)完成。

Queue类用来在进程间传递消息，它提供了一个数据的中转站，一个进程往Queue里存数据, 另一个进程可以从Queue里将该数据取出。为此，Queue提供了如下两个方法分别用于插入和 读取数据：

(1 ) put()方法

put()方法用于插入数据，其语法格式如下:

Queue.put(itemz block=Truez timeout=None)

用于把item插入队列中。如果block True (默认值)，并且timeout为正值，则方法会

阻塞timeout指定的时间，直到该队列有剩余的空间，如果超时，就会抛出QueteFiiH异常； 如果block为False,但是该Queue已经满了,会立即抛出Queue.Full异常。

(2 ) get()方法

get()方法用于读取数据*，*其语法格式如下:

Queue。get(block=Truez timeout=None)

从队列读取并且删除一个元素。如果block*为*True (默认值)，并且timeout为正值,那 么在等待时间内没有取到任何元素，会抛出Queue.Empty异常；如果block False,有两种 情况存在，如果Queue有一个值可用，则立即返回该值，如果队列为空，则立即抛出Qiwu巳 Empty异常c

为了大家更好地理解，接下来，通过示例来演示如何使用Queue类在进程间传递数据(一 个子进程负责插入数据，另一个子进程负责读取数据)，具体如下：

import multiprocessing def

def

if

writer(queue):

try:

queue .put (\* 我是数据 s f block=False)

except:

pass

reader (queue) : #从队列中读取数据

try:

print(queue.get(block=False))

except:

pass

\_name ==" main ": q=multiprocessing.Queue() #创建用作插入数据的进程

调用put方法插入数据到队列中

调用get方法从队列中读取数据

创建队列

writer\_process = multiprocessing.Process(target=writerf args=(qf)) writer\_process . start () # 开启进程

#创建甬作读取数据的进程

reader\_process = multiprocessing.Process(target=readerz args=(q,)) reader\_process . start () # 开启进程

reader\_process.j oin() writer.j oin()

程序运行的结果如下：

插入进程

我是数据

从程序运行的结果看出，一个进程读取到了另一个进程、小、心口 读取进程 写入的数据，表明使用队列能实现两个进程间的通信，原理如 图17.7所示。

数: 据:

队列

图17-7进程间的通信

:17O8线程逾缩

线程跟进程有些相似，有时被称作轻量级的进程。但不同的是，所有的线程运行在同一个 进程中，共享相同的运行环境。

17.8J ff忽是囊釋 ^

线程9有时被称为轻量级进程(lightweight process9 LWP),是程序执行流的最小单元。一 个标准的线程由线程ID、当前指令指针(PC)、寄存器集合和堆栈组成。此外，线程是进程 中的一个实体歹是被系统独立调度和分派的基本单位.线程自己不独立拥有系统资源，但它可 与同属一个进程的其他线程共享该进程所拥有的全部资源。

一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线 程之间的相互制约，致使线程在运行中呈现出间断性。

每一个应用程序都至少有一个进程和一个线程，线程是程序中一个单一的顺序控制流程。 在单个程序中同时运行多个线程完成不同的被划分成一块一块的工作，称为多线程。

举个例子，某个厂家要生成某个产品，为此要在生产基地建设很多厂房，每个厂房又要分 配多条流水生产线。这些厂房配合把整个产品生产岀来，单个厂房内的所有流水线将这个厂房 负责的部件生产出来。每个厂房拥有自己的材料库，厂房内的生产线共享这些材料，而每一个 厂家要实现生产必须拥有至少一个厂房一条生产线，那么这个厂家就是某个应用程序，每个厂 房就是一个进程，每条生产线都是一个线程。

1晃底2鳗雑爾號态

跟进程相似，线程在创建之后,并不是始终保持着一种状态,其状态大致可以分为如下几种： e New：新建。新创建的线程经过初始化以后，进入就绪状态。

° Runnable：就绪。等待系统调度，被调度以后进入运行状态。

° Running：运行。

。Blocked：阻塞。解除阻塞以后，会进入到就绪状态重新等待调度。

-奂讪 消亡。线程方法执行完毕返回或者异常终止。

这些状态之间是可以相互转换的，如图17屹

新建 部消亡

所示。

线程由运行状态转换成阻塞状态，可能有如

满足条件\ /等待条件

下3种情况：

。同步：线程中获取同步锁，但是资源已经

图17-8线程状态的转换

被其他线程锁定时，进入Locked状态，直到该资源可以获取为止。

•睡眠：线程执行sleep()或join()方法后，线程都会进入到休眠状态。两者的区别在于， 血汹)方法会等待一定的时间，而join。方法会等待子线程执行完。当然，join()方法可 以指定超时时长，而等待一定的时间。

•等待：线程中执行wait()方法后，线程进入Waiting状态，等待其他线程的通知。

线程可以分为如下几种类型：

。主线程。程序启动时，系统会创建并立刻运行一个线程，该线程通常叫做程序的主线程 (MainThread)。每个进程至少都有一个主线程，它通常是最后关闭的。

主线程的重要性体现在两方面：一是产生其他子线程的线程，二是必须最后完成执行，比 如执行各种关闭动作。

•子线程。在程序中创建的其他线程，相对于主线程来说就是这个主线程的子线程。

•守护线程（后台线程）。守护线程，是指在程序运行时在后台提供一种通用服务的线程， 比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的 部分。守护线程是一个独立于主程序的线程，它会一直运行，不会因为主程序的终止而 结束。

•前台线程。相对于守护线程的其他线程称为前台线程。

前台线程和后台线程具体有什么区别呢？在主线程执行的过程中，前台线程和后台线程都 会进行。当主线程执行完毕以后，后台线程不论成功与否，主线程和后台线程均停止；而主线 程会等待前台线程执行完成后,程序才会终止。

*；*X?多学一招：进程和线程的区别

*\*曲裕时*

实际上，进程和线程都是实现多任务的一种方式。例如，在同一台计算机上能同时运行多 个QQ （进程）9 一个QQ可以打开多个聊天窗口（线程）。两者间有着各自的特色，具体如下：

—定义方面：进程是程序在某个数据集合上的一次进行活动，线程是进程中的一个执行路径。 ®角色方面 在支持线程机制的系统中，进程是系统资源分配的单位，线程是系统调度的单位。

\*资源共享方面：进程间不能共享资源，而线程共享所在进程的地址空间和其他资源。同时， 线程有自己的栈和栈指针，程序计数器等寄存器。

•独立性方面：进程有自己独立的地址空间，而线程没有，线程必须依赖于进程存在，线 程间共享地址空间。

®性能方面：线程的划分尺度小于进程（资源比进程少），执行开销小，使得多线程的并 发性比较高，但是不利于资源的管理和保护，而进程却恰恰相反。

17O9剑建线程

Python提供了两个模块进行多线程的操作:thread和fh冀adiiig。前者是比较低级的模块， 用于更低层的操作，一般应用级别的开发不常用，后者则封装了更多高级的接口，类似于Java 的多线程风格，所以绝大多数情况下会选择使用后者。

17SJ 應阑TIhnsarf襲寰嬲寥鳗覆

在模块threading中定义了 Thread类，专门负责管理线程。启用一个线程就是把一个函数 传入并创建Thread实例，然后调用start（）方法开始执行。其构造函数的语法格式如下：

class threading.Thread（group==Nonef target=None^ name=Noner

args=（）e kwargs={}7 \* F daemon=None）

应始终使用关键字参数调用此构造函数。参数代表的含义如下:

*臨*group参数：线程组。目前没有实现，库引用中提示必须是None0

。target #B：要执行的方法。

® name参数：线程的名称。 ’

* args/ kwargs参数：要传入方法的参数。args是目标调用的参数元组，默认为()；kwargs 是目标调用的关键字参数的字典，默认为｛｝。
* daemon参数:线程的守护标志。若提供该参数，则将会设置为Tme或Falseo若没有提 供(默认为None),则该标志将从创建的线程中继承。

有了线程以后，通常需要调用线程对象的关tDaanon()方法把线程设置为守护线程(默认为 前台线程)。在主线程执行完以后，如果还有其他非守护线程，则主线程不会退出，会被无限挂起, 所以必须将线程声明为守护线程，这样线程运行完以后，整个程序不用等待就可以退出了。

要想真正地运行线程，需要调用start。方法启动。接下来，通过两个示例来区分未设置守 护线程和设置守护线程的不同，具体如下」

示例衆设置

import threading #导入多线程模块

import time

def test () : #子线程要执行的任务

time。sleep(1)

print ( ?子线程运行,线程名称为:%s \* %threading. currentThread () . getName ()) time•sleep(1)

for i in range(4):

#创建孑线程，并扌旨定线程要执行的坨$七函数 thread=threading。Thread(target=test)

# thread. setDaemon (True) # 未设置守护线程

thread. start () # 启动线程

print ( ?主进程结束？)

在上述示例中，循环创建了4个子线程，每个子线程都要花费近两秒钟的时间执行任务, 一直处于阻塞或者等待被调度的状态，所以子线程会在主线程后面执行完成。

上述示例运行的结果为辭

主进程结束 子线程运行, 子线程运行, 子线程运行, 子线程运行,

线程名称为: 线程名称为: 线程名称为: 线程名称为:

Thread-2

Thread-1

Thread-3

Thread-4

从运行结果可以看出，当主线程执行完以后，会等待所有的前台线程执行完再终止程序。 新的线程的生命周期为整个任务函数，只要任务执行完以后就会消亡。另外，创建线程的顺序 跟执行顺序无关，执行顺序是无法确定的，主要靠操作系统的调度决定。

示例二：设置守护线程

把示例一中用于设置守护线程的代码取消注释，程序运行的结果如下：

主进程结束

从运行的结果可以看出，程序没有打印后台线程的信息。这是因为当主线程执行完毕后, 无论后台线程成功与否,主线程和后台线程均停止。

(/注意：

任何进程默认会启动一个线程，该线程称为主线程，主线程又可以启动新的线程。

Python的threading模块中有个current\_thread()函数,它永远返回当前线程的实例。主线 程实例的名字叫MainThread,子线程的名字在创建时可以指定，若没有指定名字，Python就会 自动给线程命名为Thread-1, Thread-2,…

憊廳Thread琴鬟莫觐褰爨褸

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。在创建了自己的线程实例后，通过Thread类的湖rt()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mn()方法执行线程。

示例如下:

import threading

import time

class MyThread(threading.Thread): def run (self):

for i in range(3): time.sleep(1)

message= ?我是 5 +self . name+ \* @ 1 +str (i) print(message)

if name == f main ?: mythread=MyThread() mythread.start()

#自定义继承自Thread.的子类

#线程被CPU调度后自动执行该方法

# name属性保存的是当前线程的名字

#创建自定义的线程对象

#启动线程

在上述示例中，当执行到sleep()语句时，线程将被阻塞，一直到睡眠结束须线程重新进入 到就绪状态，等待调度。每个线程一定会有一个名字，尽管上述示例中没有设定线程对象的名字， 但是Python会自动为线程指定一个名字。

程序运行的结果为：

我是 Thread-1 @ 0

我是 Thread-1 @ 1

我是 Thread-1 @ 2

17S.3 ilBjoioOBSlfWfi

Pythcm提供了 join()方法，使得一个线程可以等待另一个线程执行结束后再继续运行。这 个方法还可以设定一个timeout参数，避免无休止的等待。因为两个线程顺序完成，看起来像 一个线程，所以称为线程的合并。

join()方法的定义格式如下:

j oin(timeout=None)

上述方法将会等待线程终止，这将阻塞调用的线程，直到调用该方法的线程正常或者通过 未处理的异常终止，或者到可选的超时发生。

只要调用了子线程的join。方法，主线程就会被子线程所阻塞，直到子线程执行完毕再轮 到主线程执行。在1791的示例中让子线程阻塞主线程，具体如下：

import threading #导入多线程模块

import time

def test () : #子线程要执行的任务

time-sleep(1)

print ( ,子线程运行,线程名称为：%s 1 %threading. cur rent Thread () . getName ()) time.sleep(1)

for i in range(4):

#创建于线程，并指定线程要执行的"st函数

thread = threading.Thread(target=test)

thread. start () # 启动线程

thread.j oin()

print ( ,主进程结束!)

程序运行的结果如下:

子线程运行, 子线程运行, 子线程运行, 子线程运行, 主进程结束

线程名称为： 线程名称为: 线程名称为: 线程名称为:

Thread-1

Thread-2

Thread-3

Thread-4

17ol0解决銭程典事鬢源产生胸间题

与进程不同，多个线程是共享全局变量的，这在一定程度上减少了程序的开销。凡事有利 必有弊，多个线程访问同一份资源，很有可能造成数据混乱（不同步）的情况，为此，Python 提供了互斥锁进行处理。本节将为大家介绍涉及的知识点。

17J0J 饑疆漿蔓金嵐靈最

现在有如下需求：定义一个全局变量g\_num?设置初始值为100。创建两个线程分别用于 设置和获取该全局变量的值。那么，获取到的值会在设置以后发生变化吗？

示例如下：

from threading import Thread

import time

number = 100 #声明全局变量

def set\_number():

global number

for i in range (3):

number += 1 #重新设置全局变量的值

print ( \* - -set\_n umber : number 值为 %d' % number) def get\_\_number ():

global number

print ( \* --get\_number : number 值为 %d\* %number) # 获取全局变量的值 print ( J number 初始值：%d5 %number) thread\_one = Thread (target=set\_number) # 创建设置值的线程

thread one.start()

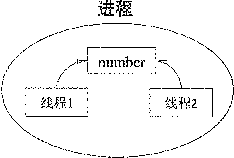
time, sleep (1) #延时一会火保证貝线程中的事情能做完

thread\_two = Thread (target=get\_number) # 创建获取值的线程

thread\_two。start()

为了保证用于设置值的线程比获取值的线程执行的早，所以使用血ep()方法睡眠延时。 程序运行的结果如下：

number 初始值:100

—set\_number : number 值为 103 一get\_number : number 值为 103

从运行的结果可以看岀，获取到的值是设置以后的值， 代表着线程是共享全局变量的。程序在执行时形成一个进 程，全局变量被放置在这个进程所占用的空间中，同时两 个子线程都属于这个进程，为此能共享这个全局变量，如 图17-9所示。一旦某个线程对该变量做出了修改，则另一

个线程会看到这个变化。 图17=9线程间共享全局变量原理图

17J0.2爾间塗魁變量噩勰灘矚不圖霽觸间靂

线程之所以比进程轻量，其中一个原因就是线程共享内存。也就是各个线程可以平等地 访问内存的数据，如果在短时间“同时并行”读取修改内存的数据，很可能造成数据不同步的 问题。

接下来，使用一个例子引出多线程访问同一份资源的隐患。例如，有个全局变量的值为0, 分别让两个线程给这个变量累加值，每个线程累加1 000 000次(数值越大，问题出现得更明显)o

示例如下:

from threading import Thread import time global\_value = 0 def test\_one():

global global\_value for i in range(1000000): global\_value += 1

print('test\_one--global\_value: def test\_two():

global global\_value for i in range(1000000): global\_\_value += 1

print(f test\_two-"global\_ value:

#导入多线程模块

#声明全局变量

#表示线程1累加值的函数

#让全局变量的值加1

%d'%global\_value)

#表示线程2累加值的函数

#让全局变量的值加1 %d? %global\_\_value)

thread\_one = Thread (target=test\_one) # 线程 1 thread.\_one。start ()

thread.\_two = Thread (target=test\_two) # 线程 2 thread\_two。start()

time.sleep(2) print(5 global\_value: %d1%global\_value) 在上述示例中，线程1累加全局变量的值1 000 000次，使得该变量的值应该从。变成

1 000 000 9而线程2再次累加1 000 00()次，使得该变量的值从1 000 000变为2 000 000 ,按照

逻辑来讲这样是没有问题的，是期望得到的结果。

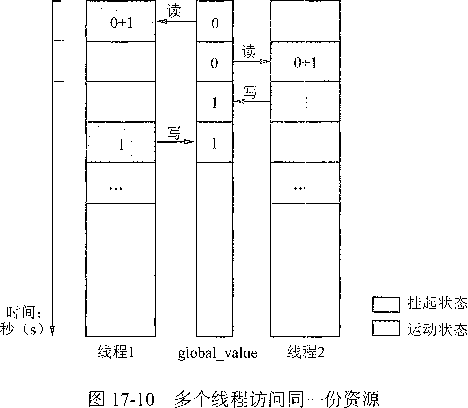
但是，程序运行的结果如下：

test\_one--global\_value: 1250721

test\_two--global\_value: 1439276

global\_value: 1439276

从运行结果可以看出，最终的值与期望的值相差很大，为什么会岀现这个问题呢？究其原 因，每个线程负责的任务可以拆分两部分：一为取变量的值，另一为给变量赋新值，主要是因 为每个线程运行的时间有限，很有可能当CPU让线程1执行完累加值还未赋值时，就被剔除 呈挂起状态，所以当线程2获取值时，访问到的仍然是未累加前的值。为了大家更好地理解， 接下来通过一张图来分析资源抢夺的原因，如图17J0所示。



1X10.3 aMBWIf ( L©©k) I?澳數疇來圖霽齣祠飄

当多个线程都对某项数据进行修改时，需要进行同步控制。线程同步能够保证多个线程安 全地访问竞争资源，其中最简单的同步机制就是引入互斥锁。互斥锁为资源引入一个状态：锁 定或者非锁定。当某个线程要更改共享的数据时，先将其锁定，此时资源的状态为“锁定”， 其他线程不能进行更改，直到该线程释放资源，将资源的状态变成“非锁定” 9其他的线程才 能再次锁定该资源。为此，使用互斥锁每次只能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。

Python的threading模块中定义了 Lock类,可以方便地处理锁定，具体可分为如下3个步骤:

第1步：

创建锁对象，可通过如下示例完成。

mutex = threading.Lock()

第2步：

锁定资源，可通过如下示例完成。

mutex.acquire([blocking])

在示例中，锁定方法有个blocking参数。如果设定blocking True (默认)，则当前线程 会堵塞，直到获取到这个锁位置；如果设定blocking \* False,则当前线程不会堵塞。

第3步：

释放资源，可通过如下示例完成。

mutex <, release ()

例如，在17J02小节的示例中，使用互斥锁锁定和释放资源呀具体如下：

from threading import Threeid^ Lock

number = 0

def test\_one():

global number

for i in range(1000000):

mutex.acquire()

number += 1 mutex.release() print('test\_one--number: %d \* %number)

def test\_two(): global number for i in range(1000000): mutex.acquire() number += 1 mutex。release()

print(5 test\_two--number: %d , %number)

mutex = Lock()

threa.d\_one = Thread (target=:test\_one)

threa.d\_one . start ()

threa.d\_two = Thread (target==test\_two)

threa.d\_two . start ()

print(\* number: %d \* %number)

上述示例中使用互斥锁锁住了变动的数据。起初两个线程都抢着对这个锁进行上锁，直到 有一方成功上锁，则另外一方会堵塞到这个锁解开为止。只要锁开了，就让所有因这个锁被锁 而堵塞的线程继续抢着上锁。

程序运行的结果如下：

number: 52359

test\_ one--number: 1810700

test\_two--number: 2000000

程序运行的最终结果跟期望的值一样，表明互斥锁成功地解决了数据不同步的问题。

囑宥爵令饋遭鐵瓠鐵餉祠題

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，但是一旦发生就会造成应用停止响应。

为了大家更好地理解，接下来使用一个示例，在示例中有两个自定义的线程类，每个线程 类使用锁来锁定资源，并且相互间引用了对方的锁资源。具体如下：

import threading

import time

class MyThreadOne (threading . Thread) : # 自定义一个线程类

def run (self) : #线程启动时自动调用该方法

if mutex\_a . acquire () : # 对 mutex\_a 上锁

print(self.name + \*: dol-up--\*)

time.sleep(1) if mutex\_b.acquire():

|  |  |  |
| --- | --- | --- |
| mutex\_b。release() | # | 对mutex\_b解锁 |
| mutex\_a.release() | # | 对mutex\_\_a解锁 |
| class MyThreadTwo(threading.Thread): | # | 自定义一不线程类 |
| def run(self): | # | 线程启动时自动调用该方法 |
| if mutex\_b.acquire(): | # | 对mutex\_\_b上锁 |

#对mutex\_b上锁

print(self.name + \*: do1-down--1)

print(self.name + \*: do2-up--?)

time.sleep(1)

if mutex \_a . acquire () : # 对 mutex\_\_a 上锁

print (self . name + 5 : do2-d.own-- 5 )

|  |  |  |  |
| --- | --- | --- | --- |
| mutex\_a.release() | | #  # | 对mutex\_a解锁 对mutex\_\_b解锁 |
|  | mutex\_b.release() |
| mutex\_a = | threading.Lock() | # | 创建锁对篆a |
| mutex\_b = | threading.Lock() | # | 创建锁对象b |
| if name\_ | \_ == \* main ?: |  |  |
| thread | \_one = MyThreadOne() | # | 创建一个线程thread\_one |
| thread | \_two = MyThreadTwo() | # | 创建一个线程thread\_two |

thread\_one.start()

thread\_two。start()

当调用上述两个线程的就art()方法时，这些线程分别执行各自的run()方法。假设线程 thread\_one先得到调度，会对mutex\_a进行上锁，然后又换到thread\_two获得执行权，会对 mutex\_b进行上锁。当线程thread\_one要对mutex\_b上锁时,发现线程thread\_two对mutex b 已经上锁,此时只能等待mutex\_b解锁;而线程threadjwo要对mutex\_a上锁时,同样发现线 程thread one已经对mutex a上锁,此时也只能等mutex a解锁。

这样.程序岀现死锁会一直处于阻塞状态，只能主动结束程序：

Thread-1: dol-up--

Thread-2: do2-up~- I //光标等待输出

为了避免像上述产生死锁的问题，因此，可以在锁定资源时设定超时时间，只要超过了设

定的时间，线程将不会阻塞重新恢复运行。

在上述示例中，锁后对mutexb上锁时 省略N行...

为其设定超时时长为2秒，具体代码如下。

#对mutex\_a上锁

#对mutex\_b上锁,超时时长为2秒

if mutex\_a»acquire():

print(self.name + s: dol~up~™ 5) time.sleep(1)

if mutex\_b.acquire(timeout=2): 省略N行

程序在阻塞了两秒钟后结束，并输岀如下信息：

Thread-1: dol-up-~

Thread-2 : d.o2-up-~

Thread-2: do2-down~-

从上述运行结果可以看出,线程Thread-1对mutex\_a进行上锁,之后线程Thread^对

进行上锁。两个线程休眠1秒钟，都处于阻塞状态等待对方解锁。由于mutRx\_b设置 超时时间为2,表明Thread-1仅仅会阻塞两秒钟,之后会继续向下执行程序,从而对mutex\_a 进行解锁。此时，Thread-2发现mutex\_建已经解锁，它会对mutex\_务进行上锁，并执行输岀语句。

17J0.5廁蠶入鐡(^L©©k)餉邇魇

在程序中，即使只有一个Lock实例，也有可能会造成死锁的问题。例如，在同一个线程 中连续两次调用acquire方法,示例代码如下:

|  |  |
| --- | --- |
| class MyThread(threading.Thread): | #自定义一个线程类 |
| def run(self): | #线程启动时自动调用该方法 |
| mutex.acquire()  mutex•acquire()  mutex o release () mutex = threading.Lock() if \_name == \* main \* : thread = MyThread.() threads start() | #对mutex上锁  #再次对mutex上锁  #对mutex解锁  #创建锁对象a  #创建一个线程thread |

运行程序，程序一直处于等待状态，没有岀现结束。

为了能在同一个线程中多次请求同一份资源,Python提供了可重入锁(RXock)。RLock 内部维持着一个Lock和一个confer变量，这个变量会记录上锁的次数，从而使得资源可以被 多次需求，直到一个线程所有上的锁都解除了，其他线程才能获得资源。

示例如下：

import threading

| time*。*sleep(1) mutex.acquire() mutex.release() mutex。release() | #对mutex上锁  #对mutex解锁  #对mutex解锁 |
| --- | --- |
| print (” 线程 %s 解锁 n % self . name)  print ("主线程开始")  mutex = threading»RLock()  threads = [MyThread() for i in range (2)] for tureed in threads: | #创建可重入锁 #创建两个线程 |
| tureedostart() | #启动线程 |

print ("主线程结束° )

上述示例中，在每个线程中都对同一份资源进行了多次锁定和解锁，并且上锁和解锁的次 数是相互对应的。

程序运行的结果如下：

主线程开始

线程Thread-1上锁

|  |  |
| --- | --- |
| import time  class MyThread(threading.Thread): | #自定义线程类 |
| def run(self): | #启动线程后自动调用的方法 |
| if mutex.acquire(): | #对mutex上锁 |

print ("线程 %s 上锁"%self . name)

主线程结束

线程Thread-1解锁

线程Thread-2上锁

线程Thread-2解锁

("多学一招：多线程使用非共享变量

在访问全局变量时，如果希望只获取其值，而不做任何修改，则无须用到互斥锁。要是在 多个线程中使用到非共享的变量，线程间会产生怎样的结果呢？

接下来,创建执彳亍test\_one函数的线程thread\_one,执行test\_two函数的线程thread\_\_two,在 两个函数的内部各自声明具有相同名称的局部变量，进行一系列的操作后查看结果，具体如下：

from threading import Thread, import time def test\_one():

number = 100

number += 1 print(test\_one: time.sleep(2) print('test\_one: def test\_two():

time"sleep(1)

number = 100 print(test\_two:

#定义局部变量

number=%d \* %number)

# 保证thread\_two运行

number=%d' %number) # 查看。11111}3女 的值是否被 thread\_two 改变

#保证thread\_one先执行

#定义同名的局部变量,给number赋值 number=%d? %number)

thread\_one = Thread(target=test\_one) thread\_one。start() thread\_two = Thread(target=test\_two) thread\_two.start()

程序运行的结果如下:

test\_one: number=101

test\_two: number=100

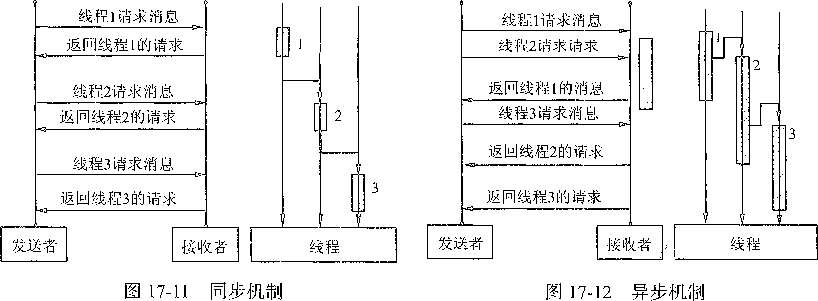
test\_one: number=101

从运行的结果•可以看出，number的值始终与thread\_one线程运行的结果保持一致，并没有 因为th冀Mi\_two的重新赋值发生改变。为此，在两个线程中变量是各自的，相互之间没有任何 影响。当创建一个线程后，这个函数的所有内存空间是这个线程独有的，当另一个线程执行这 个函数时，重新创建一份内存空间，所以这两个内存空间的变量没有任何关系。

I IToll戮程的同步应朋

所谓同步，就是在发岀一个“调用”时，在没有得到结果之前，该“调用”就不返回。但 是一旦调用返回，就得到返回值了。换句话说，就是由“调用者”主动等待这个“调用”的结果。

而异步则是相反，“调用”在发出之后，这个调用就直接返回了，所以没有返回结果。换 句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在“调用’’发出后，“被 调用者”通过状态、通知来通知调用者，或通过回调函数处理这个调用。为了能更好地理解， 接下来通过两张图来描述异步和同步的不同，如图1741和图17J2所示。



为此，Pythoii的多线程模块中提供了条件变量(Condition类)和队列(Qu\*类)，实现 对复杂线程同步问题的支持。

17J11 SaSftWB ( C©ndlion )黨嬲爨體圖歩

Condition类提供对复杂线程同步问题的支持，Condition类称为条件变量，它允许一个或 多个线程等待，直到被另一个线程通知。条件变量通常与一个锁关联，当需要在多个Condition 对象中共享一个锁时，可以传递一个Lock/FtLock实例给构造函数，语法格式如下：

class threading.Condition(lock^None)

若给出lock参数，则它必须是Lock或者RLock对象，并将其作为底层锁。否则将自己生 成一个新的RLock对象作为底层锁。

可以认为,Condition对象维护了 一个锁(Lock/RLock)以及一个waiting池,线程通过 acquire()方法获得条件变量，然后判断条件：

羯如果不满足条件，则调用wait()方法等待，线程会释放Condition内部的锁并进入 blocked状态,同时在waiting池中记录这个线程。

•如果条件满足，则进行一些处理改变条件后，通过调用noti以()方法通知其他线程，这 些处于等待状态的线程接到通知后会重新判断条件。

演示条件变量同步的经典问题是生产者与消费者问题：有一群生产者(PmduceQ和一群 消费者(Consume)通过一个市场来交互产品。生产者的“策略”是如果市场上剩余的产品少 于1 000个，那么就生产100个产品放到市场上；而消费者的“策略”是如果市场上剩余产品 的数量多于100个，那么就消费3个产品。用Condition解决生产者与消费者问题的代码如下：

|  |  |  |  |
| --- | --- | --- | --- |
| import threading  import time | | | |
| class Producer(threading.Thread): def run(self): | | # | 代表生产者的线程 |
|  | global count while True: | # | 市场上产品的剩余数量 |
|  | if condition.acquire (): | # | 上锁 |
|  | if count > 1000: | # | 判断产品剩余数量是否大于1000 |
|  | condition.wait() | # | 满足,放入到waiting池中等待 |

else :

#不满足，再生产100个

count += 100 message = self»name+J print(message) condition.notify() condition.release() time。sleep(1)

class Consumer(threading.Thread): def run(self):

global count while True: if condition.acquire():

if count < 100: condition.wait() else :

count -= 3

message = self.name+

生成100个,总数量为:1 + str (count)

#通知其他线程

#释放锁

#让线程休眠:Ls

#代表消费者的线程

#上锁

#判断产品剩余数量是否小于100

#满足，放入waiting池中等待

Thread-5

Thread-3

Thread-6

Thread-7

Thread-1

Thread-4

Thread-5

肖肖肖肖史肖肖 、、/ 、、/ 、、丿、、/ 卢一 、、/*7*

费费费费成费费

/ r f / 个个个个30个个

总数量为: 总数量为: 总数量为: 总数量为: ,总数量方 总数量为: 总数量为:

10 07

1004

1001

998

:1098

1095

1092

|  |  |  |  |
| --- | --- | --- | --- |
| print(message) | |  |  |
| condition。notify() | | | #通知其他线程 |
| condition.release() |  |  | #释放锁 |
| time。sleep(1) | |  | #让线程休眠Is |
| count = 500 |  |  | #初始产品数量 |
| condition = threading.. Condition () # | | 创建条-件变量对象，使用RLock对象作为底层锁 | |
| if name == 1 main ?: |  |  |  |
| for i in range(2): |  |  |  |
| producer = Producer() | | # | 创建两个生产者流水线 |
| producer。start() |  | # | 启动线程，自动调用run ()方法 |
| for i in range(5): |  |  |  |
| consumer = Consumer() | | # | 创建两个消费者销售线 |
| consumer。start() |  | # | 启动线程，自动调用run ()方法 |
| 程序运行的结果如下： |  |  |  |
| Thread-1生成100个*丫*总数量为 | :600 |  |  |
| Thread-2生成100个，总数量为 | :700 |  |  |
| Thread-3消费3个,总数量为: | 697 |  |  |
| Thread-4消费3个,总数量为: | 694 |  |  |
| Thread-5消養3个,总数量为: | 691 |  |  |
| Thread-6消费3个,总数量为: | 68 8 |  |  |
| Thread-7消養3个'总数量为: | 685 |  |  |

#不满足，再消费3个

消费3个,总数量为:\* +str (count)

1 私街卷

上节使用条件变量同步机制解决了生产者与消费者的问题。让我们来考虑一种更为复杂的 场景：产品是各不相同的。这时，只通过记录一个数量显然不够，还需要记录每个产品的细节, 很容易想到需要用一个容器将这些产品记录下来。

Python的queue模块中提供了同步的、线程安全的队列类,包括FIFO (先进先出)队列 QueueLIFO (先进后出)队列LifoQueue,以及优先级队列PriorityQueueo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。

要通过队列实现线程间的同步，可使用如下方法：

° Queue(maxsize=O):构造一个先进先出队列。maxsize用于指定队列的长度,为0时表示 队列长度无限制。

° Queue.qsize()：返回队列的大小。

° Queue.emptyQ:当队列为空时返回True,否则返回Falseo

° Queue.full():当队列已满时返回True,否则返回Falseo

° Queue.put(item, block=True, timeout=None):将 item 放入 Queue 的末尾,item 必须存在。 block参数默认为Trw,表示当队列满时，会等待队列给出可用位置，当为False时表示 非阻塞，此时若队列已满，会引发queueTull异常。可选参数timeout表示阻塞设置的时间? 超岀该时间后，若队列无法给岀放入血m的位置，贝0引发queueTull异常。

° Queue.get(block=True, timeout=None):移除并返回队列头部的一个值。可选参数block 默认为True,表示获取值时如果队列为空，则阻塞；若为屈檢，表示不阻塞。若此时 队列为空，则会引发queue.Empty异常。可选参数timeout 阻塞设置的时间，过后，

若队列为空，则引发queue.Empty异常。

用FIFO队列实现上述生产者与消费者问题的代码如下：

|  |  |
| --- | --- |
| import threading | |
| import time  from queue import Queue class Producer(threading„Thread): | #代表生产者的线程 |
| def run(self): global queue count *= 0* | #初始的产品数量 |
| while True:  for i in range(100):  if queue.qsize() > 1000 : | #若队列的大小比1000 *k,*不做任何操作 |
| pass  else :  count += 1  message = ,生成产品 | \* +str(count) |
| queue .,put (message) | #把新生产的产品放到队列 |
| print(message) time。sleep (1)  class Consumer(threading„Thread): | #代表消费者的线程 |
| def run(self):  global queue while True:  for i in range (3): if queue-qsize() < 100: | #若队列的大小比100小，不做任何操作 |
| pass  else :  #从队列中销售产品  message = self.name +，消费了 足 +queueget () | |
| print(message) |  |

360 Pyfthoo 寰臆飜鹳 Python

time.sleep(1)

queue = Queue () #创建队列对象

if name == ' main ':

for i in range(500): queue .put ('初始产品1 +str (i) ) #往队列中放入初始产品500个

for i in range (2):

producer = Producer() producer.start() for i in range (5):

consumer = Consumer() consumer.start()

在上述示例中，使用先入先出的队列放置初始产品和新生产的产品，并且满足条件后从队 列中获取待销售的产品，使得产品的生产和销售始终维持这种预订的状态(市场上的产品数量 少于1 000个就生产，大于100个就消费)。

程序运行的结果为:

生成产品1

生成产品2

生成产品99

生成产品100

Thread-3消费了初始产品0

Thread-3消费了初始产品1

Thread™?消费了初始产品13 Thread-7消费了初始产品14 生成产品101

生成产品102

生成产品199

生成产品2 0 0

Thread-4消费了初始产品15

Thread-4消费了初始产品16

Thread-7消费了初始产品29 生成产品2 01

生成产品202

(片多学一招:什么是生产者一消费者模式？

生产-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 彼此之间不直接通信，而通过阻塞队列来进行通信，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。

*R多*学一招：异步应用

异步的标准元素是回调函数(Callback,后来衍生出Promise/Defered概念),主线程发起一 个异步任务，让其自己到一边去工作，当其完成后，会通.过执行预先指定的回调函数完成后续任务,

然后返回主线程。在异步任务执行过程中，主线程无须等待和阻塞,可以继续处理其他任务。 这里，我们以进程为例，介绍如何^使用回调函数实现异步编程。

from multiprocessing import Pool

import time

import os

def test\_one () : #子进程负责执行的任务函数

print (° 进程池中的进程:pid~%d, ppid~%dn% (os . getpid () z os.getppid())) for i in range(3):

print ( \* \* %i)

time。sleep(1) return \* haha s

def test\_two (args) : # 回调函数

print(1一一callback func： pid=%d一一1%os.getpid ()) print(\*--callback func: args=%s--!%args)

pool = Pool (3) #创建进程池

#为进程池添加功能函数和回调函数

pool。apply\_async(func=testf callback=test\_two)

time。sleep(5)

print (' 一 主进程:pid=%d~- ' %os . getpid ())

在示例中，通过进程池来开启多个任务，并且为进程池指定了功能函数和回调函数。 程序运行的结果为：

进程池中的进程：pid-1756, ppid-1755

—0 —

—1 —

—2 —

--callback func: pid=1755--

--callback func: args=haha-~-

--主进程：pid=1755--

从上述结果可以看出，在执行完给子进程分配的任务以后，程序继续执行指定的回调函数， 返回到主进程中由它自己来负责执行。

*:11.*12使网事滞(E與踰)实现线程通僞

很多时候，线程之间会有互相通信的需要。常见的情形是次要线程为主要线程执行特定的任 务，在执行过程中需要不断报告执行的进度情况。前面的Condition类已经涉及到了线程间的通 信(threading.Condition的notify()方法),事实上,更为通用的方式是使用threading.Event对象。

threading.Event可以使一个线程等待其他线程的通知，其内置了一个标志，初始值为 False,线程通过wMt()方法进入等待状态，直到另一个线程调用皿()方法将内置标志设置为 True时，Event通知所有等待状态的线程恢复运行。此外，还可以通过isSet()方法查询Event 对象内置状态的当前值。

具体示例如下:

import threading

import random

import time

class MyThread (threading. Thread.) : # 自定义线程类

def init (self,thread\_name,event):

threadingThread» init (self z name=thread\_name) self.thread\_event = event

|  |  |  |
| --- | --- | --- |
| def run(self): | | |
| print (” 线程 %s 准备"% self . name) self。thread\_event。wait()  print ("%s 运行时％ self . name) | # | 线程主动进入等待状态 |
| sinal = threading.Event() for i in range (10):  thread = MyThread(str(i)7sinal) thread。start() | # | 创建事件对象 |
| sinal.set() | # | 重置內置标志的状态 |

程序运行的结果为：

线程0准备

线程1准备

线程2准备

线程3准备

线程4准备

线程5准备

线程6准备

线程7准备

线程8准备

线程9准备

0运行

2运行

6运行

7运行

9运行

4运行

8运行

5运行

1运行

3运行

Wo 13后侖线程

默认情况下，主线程在退出时会等待所有子线程的结束。如果希望主线程不等待子线程， 而是在退岀时自动结束所有的子线程，就需要设置子线程为后台线程(daemon) ?具体是通过 调用Thread类的setDaemon()方法。

示例如下:

import threading import random import time class MyThread(threading.Thread):

def run(self):

wait\_time = random.randrange(1z 10) print ("%s 要等％d 秒"% (self . namez wait\_time)) time。sleep(wait\_time) print ("%s 结束"%self . name)

if name == ° main n :

print (，主线程等待结束...?)

for i in range(5):

thread = MyThread() thread.setDaemon(True) thread.start() print ( ,主线程结束...\* )

程序运行的结果为：

。秒秒秒秒秒 〕5 9 8 8 8 柬等等等等等 .

Thread-1

Thread-2

Thread-3

Thread-4

Thread-5

主线程结束

（注意:join（）方法使得线程可以等待另一个线程的运行,而setDaemonQ方法使得线程在 结束时可以不等待子线程，它们都可以改变线程之间的运行顺序。

17； 14协程

协程，又称微线程（或者纤程，英文名Coroutine）。协程的作用是在执行函数A时，可 以随时中断去执行函数然后中断函数B继续执行函数A（可以自由切换）。不过，这个过 程并不是函数的调用，看似是像多个线程执行任务，然而协程却只有一个线程执行。接下来， 本节将带领大家学习协程的知识。

与线程相比，协程具有如下优势：

。执行效率极高。协程切换并不是线程间的切换，它是由程序自身所控制的，没有切换线 程的开销，所以跟多线程相比，线程的数量越多，协程性能的优势就更明显。

-不用锁机制。由于只有一个线程，就不存在同时写变量的冲突，在控制共享资源时不需 要加锁，所以执行效率会高很多。

17J4.2

在Python 2°x中，支持协程的模块并不多，比较常用的就是gevento gevent是一个第三方库*，* 通过greenlet实现协程，它的用法具体如下:

t基赤思想

当遇到10操作（如访问网络）时.一个greenlet （小型的独立伪线程）会自动切换到其他 的greenlet,只要K）操作执行完成，就会在适当的时候切换回来继承执行。

由于IO操作是非常耗时的，程序如果经常处于等待的状态，那么一定会降低程序执行的效 率。使用gweirt库为我们自动切换协程,，保证了程序中始终有greenlet在运行，而不是等待KX

2.靈装配置

可使用如下命令安装gevent第三方库：

pip install gevent

1. 使用说ia

gevent常用的方法如下:

• monkey：使一些阻塞的模块变得不阻塞。

® gevent.spawn()函数:启动协程。

® gevent.joinall()函数:停止协程。

1. 具体案例

下面通过一个简单的爬虫的案例，帮助大家认识协程的使用。

使用协程的示例如下：

import gevent from gevent import monkey monkey.patch\_all() import urll.ib2 def get\_body(i): print ("开始 %d"%i) urllib2•urlopen("[http://www.itcast.cn](http://www.itcast.cnH)[H](http://www.itcast.cnH)) print (° %dn%i)

tasks = [gevent. spawn (get\_bodyz i) for i in range (3) ] # 启动协程

gevent. j oinall (tasks) # 停止协程

上述示例中，分别调用spawn()和joinall ()函数来启动和停止协程。

程序运行的结果为：

0

始始始束束束 #-#■#-结结结

1

2

2

0

1

从运行结果可以看出，执行get\_body()函数的顺序应该是先输出“开始”，然后执行到 urlopenQ时碰到10堵塞，则会启动切换运行下一个程序，直到urlopen()返回结果再执行“结束”。 也就是说，程序并没有等待请求网站返回结果，而选择直接跳过，等待执行完毕以后再回来获 取返回值。注意，在此过程中只有一个线程在执行。

17J4.3 Python

Python 3.x中支持协程的模块同样包括gevent,用法跟2.x版本的一样。此外,在3.4版本 以后又引入了 asyncio标准库，这个库直接内置了对异步IO的支持。asyncio库的异步操作，需 要在coroutine中通过yield from完成。

具体示例如下:：

import asyncio Qasyncio.coroutine def test(i):

print(Htest\_l %d"%i)

.result=yield from asyncio.sleep (1)

print(°test\_2 %d"%i)

loop=asyncio。get\_event\_loop()

tasks=[test(i) for i in range(5)]

loop.run\_until\_complete(asyncio.wait (tasks)) loop。close()

在上述示例中切@asyncioxoroutine会把生成器标记为coroutine类型，然后把这个 tine扔到事件循环中执行。

程序运行的结果为：

test 1 1

|  |  |  |
| --- | --- | --- |
| test | 1 | 4 |
| test | 1 | 0 |
| test | 1 | 2 |
| test | 1 | 3 |
| test | 2 | 1 |
| test | 2 | 4 |
| test | 2 | 0 |
| test | 2 | 2 |
| test | 2 | 3 |

从上述结果可以看出，与gevent库达到的效果一样，同样在遇到K)操作时进行切换(先 输出test\_\_l 9再输出test\_2 )。首先，程序先打印出test\_\_l 9然后yield from调用另外一个生成器, 由于asyncio.sleep()也是一个coroutine,所以线程不会等待，而是直接中断并且执行下一个消 息循环，当其返回结果时，线程就可以从yield from中拿到返回值(此处为None)，然后执行 下一行语句。

注意：这里可以把asyncio.sleep(l)看成是一个耗时1 $的K)操作。在此期间，主线程并 未等待,而是去执行时间循环(Eventlvoop )中其他可以执行的coroutine,为此能实现并发执行。

为了简化并且更好地标识异步10操作，从Python 35版本开始引入了新的语法async和 awaito要使用这两个新的语法，需要做到如下两个替换：

® 扌巴 @asyncio.coroutine 替换为 asynco

•把 yield from 替换为 await o

把上述示例中的这两个部分按照上述要求进行替换，其余均不变，具体如下：

import asyncio

async def test(i):

print(°test\_l %d"%i)

await asyncio.sleep(1)

print(°test\_2 %d"%i) loop=:asyncio。get\_event\_\_loop () tasks=[test(i) for i in range(5)] loop.run\_until\_complete(asyncio.wait(tasks)) loop » close()

程序运行的结果与上述示例的结果一样。