

# 1. Python 进程和线程（包含两者区别）

几乎所有的操作系统都支持同时运行多个任务，每个任务通常是一个程序，每一个运行中的程序就是一个**进程**，即进程是应用程序的执行实例。现代的操作系统几乎都支持多进程并发执行。

注意，并发和并行是两个概念，并行指在同一时刻有多条指令在多个处理器上同时执行；并发是指在同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。

例如，程序员一边开着开发工具在写程序，一边开着参考手册备查，同时还使用电脑播放音乐.....除此之外，每台电脑运行时还有大量底层的支撑性程序在运行.....这些进程看上去像是在同时工作。

但事实的真相是，对于一个 CPU 而言，在某个时间点它只能执行一个程序。也就是说，只能运行一个进程，CPU 不断地在这些进程之间轮换执行。那么，为什么用户感觉不到任何中断呢？

这是因为相对人的感觉来说，CPU 的执行速度太快了（如果启动的程序足够多，则用户依然可以感觉到程序的运行速度下降了）。所以，虽然 CPU 在多个进程之间轮换执行，但用户感觉到好像有多个进程在同时执行。

**线程**是进程的组成部分，一个进程可以拥有多个线程。在多线程中，会有一个主线程来完成整个进程从开始到结束的全部操作，而其他的线程会在主线程的运行过程中被创建或退出。

当进程被初始化后，主线程就被创建了，对于绝大多数的应用程序来说，通常仅要求有一个主线程，但也可以在进程内创建多个顺序执行流，这些顺序执行流就是线程。

当一个进程里只有一个线程时，叫作**单线程**。超过一个线程就叫作**多线程**。

每个线程必须有自己的父进程，且它可以拥有自己的堆栈、程序计数器和局部变量，但不拥有系统资源，因为它和父进程的其他线程共享该进程所拥有的全部资源。线程可以完成一定的任务，可以与其他线程共享父进程中的共享变量及部分环境，相互之间协同完成进程所要完成的任务。

多个线程共享父进程里的全部资源，会使得编程更加方便，需要注意的是，要确保线程不会妨碍同一进程中的其他线程。

线程是独立运行的，它并不知道进程中是否还有其他线程存在。线程的运行是抢占式的，也就是说，当前运行的线程在任何时候都可能被挂起，以便另外一个线程可以运行。

多线程也是并发执行的，即同一时刻，**Python** 主程序只允许有一个线程执行，这和全局解释器锁有关系，后续会做详细介绍。

一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发运行。

从逻辑的角度来看，多线程存在于一个应用程序中，让一个应用程序可以有多个执行部分同时执行，但操作系统无须将多个线程看作多个独立的应用，对多线程实现调度和管理以及资源分配，线程的调度和管理由进程本身负责完成。

简而言之，进程和线程的关系是这样的：操作系统可以同时执行多个任务，每一个任务就是一个进程，进程可以同时执行多个任务，每一个任务就是一个线程。

## 2. Python 创建线程（2 种方式）详解

[Python](#) 中，有关线程开发的部分被单独封装到了模块中，和线程相关的模块有以下 2 个：

- `_thread`：是 Python 3 以前版本中 `thread` 模块的重命名，此模块仅提供了低级别的、原始的线程支持，以及一个简单的锁。功能比较有限。正如它的名字所暗示的（以 `_` 开头），一般不建议使用 `thread` 模块；
- `threading`：Python 3 之后的线程模块，提供了功能丰富的多线程支持，推荐使用。

本节就以 `threading` 模块为例进行讲解。Python 主要通过两种方式来创建线程：

1. 使用 `threading` 模块中 `Thread` 类的构造器创建线程。即直接对类 `threading.Thread` 进行实例化创建线程，并调用实例化对象的 `start()` 方法启动线程。
2. 继承 `threading` 模块中的 `Thread` 类创建线程类。即用 `threading.Thread` 派生出一个新的子类，将新建类实例化创建线程，并调用其 `start()` 方法启动线程。

### 调用 `Thread` 类的构造器创建线程

`Thread` 类提供了如下的 `__init__()` 构造器，可以用来创建线程：

```
__init__(self, group=None, target=None, name=None, args=(), kwargs=None, *, daemon=None)
```

此构造方法中，以上所有参数都是可选参数，即可以使用，也可以忽略。其中各个参数的含义如下：

- `group`：指定所创建的线程隶属于哪个线程组（此参数尚未实现，无需调用）；
- `target`：指定所创建的线程要调度的目标方法（最常用）；
- `args`：以元组的方式，为 `target` 指定的方法传递参数；
- `kwargs`：以字典的方式，为 `target` 指定的方法传递参数；
- `daemon`：指定所创建的线程是否为后代线程。

这些参数，初学者只需记住 `target`、`args`、`kwargs` 这 3 个参数的功能即可。

下面程序演示了如何使用 `Thread` 类的构造方法创建一个线程：

```
1. import threading
2. #定义线程要调用的方法，*add 可接收多个以非关键字方式传入的参数
3. def action(*add):
4.     for arc in add:
5.         #调用 getName() 方法获取当前执行该程序的线程名
6.         print(threading.current_thread().getName() + " " + arc)
7. #定义为线程方法传入的参数
8. my_tuple = ("http://c.biancheng.net/python/", \
9.             "http://c.biancheng.net/shell/", \
10.            "http://c.biancheng.net/java/")
11. #创建线程
```

```
12. thread = threading.Thread(target = action, args =my_tuple)
```

有关 Thread 类提供的和线程有关的方法，可阅读 [Python Thread 手册](#)，由于不是本节重点，这里不再进行详细介绍。

由此就创建好了一个线程。但是线程需要手动启动才能运行，threading 模块提供了 start() 方法用来启动线程。因此在上面程序的基础上，添加如下语句：

```
1. thread.start()
```

再次执行程序，其输出结果为：

```
Thread-1 http://c.biancheng.net/python/  
Thread-1 http://c.biancheng.net/shell/  
Thread-1 http://c.biancheng.net/java/
```

可以看到，新创建的 thread 线程（线程名为 Thread-1）执行了 action() 函数。

默认情况下，主线程的名字为 MainThread，用户启动的多个线程的名字依次为 Thread-1、Thread-2、Thread-3、...、Thread-n 等。为了使 thread 线程的作用更加明显，可以继续在上面程序的基础上添加如下代码，让主线程和新创建线程同时工作：

```
1. for i in range(5):  
2.     print(threading.current_thread().getName())
```

再次执行程序，其输出结果为：

```
MainThreadThread-1 http://c.biancheng.net/python/  
  
MainThreadThread-1 http://c.biancheng.net/shell/  
  
MainThreadThread-1 http://c.biancheng.net/java/  
  
MainThread  
MainThread
```

可以看到，当前程序中有 2 个线程，分别为主线程 MainThread 和子线程 Thread-1，它们以并发方式执行，即 Thread-1 执行一段时间，然后 MainThread 执行一段时间。通过轮流获得 CPU 执行一段时间的方式，程序的执行在多个线程之间切换，从而给用户一种错觉，即多个线程似乎同时在执行。

如果程序中不显式创建任何线程，则所有程序的执行，都将由主线程 MainThread 完成，程序就只能按照顺序依次执行。

## 继承 Thread 类创建线程类

通过继承 Thread 类，我们可以自定义一个线程类，从而实例化该类对象，获得子线程。

需要注意的是，在创建 Thread 类的子类时，必须重写从父类继承得到的 run() 方法。因为该方法即为要创建的子线程执行的方法，其功能如同第一种创建方法中的 action() 自定义函数。

下面程序，演示了如何通过继承 Thread 类创建并启动一个线程：

```
1. import threading  
2.  
3. #创建子线程类，继承自 Thread 类
```

```
4.  class my_Thread(threading.Thread):
5.      def __init__(self, add):
6.          threading.Thread.__init__(self)
7.          self.add = add
8.          # 重写 run() 方法
9.      def run(self):
10.         for arc in self.add:
11.             #调用 getName() 方法获取当前执行该程序的线程名
12.             print(threading.current_thread().getName() + " " + arc)
13.
14. #定义为 run() 方法传入的参数
15. my_tuple = ("http://c.biancheng.net/python/", \
16.             "http://c.biancheng.net/shell/", \
17.             "http://c.biancheng.net/java/")
18. #创建子线程
19. mythread = my_Thread(my_tuple)
20. #启动子线程
21. mythread.start()
22. #主线程执行此循环
23. for i in range(5):
24.     print(threading.current_thread().getName())
```

程序执行结果为：

```
MainThreadThread-1 http://c.biancheng.net/python/

MainThreadThread-1 http://c.biancheng.net/shell/

MainThreadThread-1 http://c.biancheng.net/java/

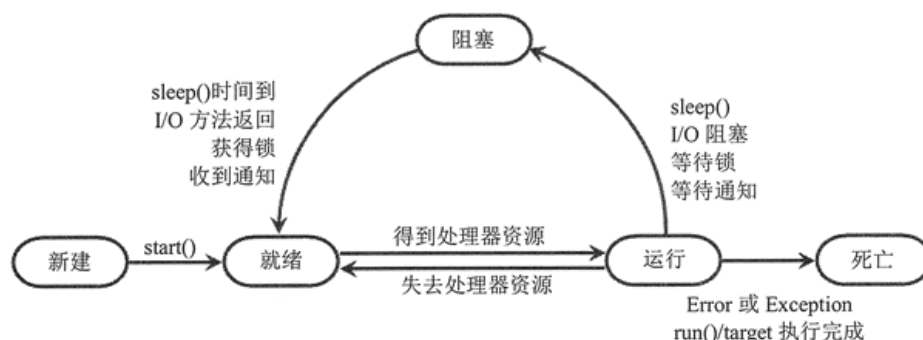
MainThread
MainThread
```

此程序中，子线程 Thread-1 执行的是 run() 方法中的代码，而 MainThread 执行的是主程序中的代码，它们以快速轮换 CPU 的方式在执行。

### 3. Python 线程的生命周期（新建、就绪、运行、阻塞和死亡）

《[Python 创建线程](#)》一节中，介绍了 2 种创建线程的方法，通过分析线程的执行过程我们得知，当程序中包含多个线程时，CPU 不同一直被特定的线程霸占，而是轮流执行各个线程。

那么，CPU 在轮转执行线程过程中，线程都经历了什么呢？线程从创建到消亡的整个过程，可能会历经 5 种状态，分别是新建、就绪、运行、阻塞和死亡，如图 1 所示。



#### 线程的新建和就绪状态

无论是通过 Thread 类直接实例化对象创建线程，还是通过继承自 Thread 类的子类实例化创建线程，新创建的线程在调用 start() 方法之前，不会得到执行，此阶段的线程就处于新建状态。

从图 1 可以看出，只有当线程刚刚创建，且未调用 start() 方法时，该线程才处于新建状态，而一旦线程调用 start() 方法之后，线程将无法再回到新建状态。

当位于新建状态的线程调用 start() 方法后，该线程就转换到就绪状态。所谓就绪，就是告诉 CPU，该线程已经可以执行了，但是具体什么时候执行，取决于 CPU 什么时候调度它。换句话说，如果一个线程处于就绪状态，只能说明此线程已经做好了准备，随时等待 CPU 调度执行，并不是说执行了 start() 方法此线程就会立即被执行。

值得注意的是，start() 方法只能由处于新建状态的线程调用，而一旦调用 start() 方法，线程状态就会由新建状态转为就绪状态。这意味着，每一个线程最多只能调用一次 start() 方法。如果多次调用，则 [Python](#) 解释器将抛出 RuntimeError 异常。

另外，线程由新建状态转到就绪状态，只有一个办法，就是调用 start() 方法。有读者可能会问，直接调用 Thread 类构造方法中 target 参数指定的函数，或者直接调用 Thread 子类中的 run() 实例方法，不可以吗？当然不可以，这 2 种方法是可以执行目标代码，但是由主线程 MainThread 负责执行，而不是由新建的子线程负责执行。

原因很简单，一方面 Python 解释器会将它们看做是普通的函数调用和类方法调用。另一方面，由于新建的线程属于新建状态而不是就绪状态，因此不会得到 CPU 的调度。

#### 线程的运行和阻塞状态

当位于就绪状态的线程得到了 CPU，并开始执行 target 参数执行的目标函数或者 run() 方法，就表明当前线程处于运行状态。

但如果当前有多个线程处于就绪状态（等待 CPU 调度）时，处于运行状态的线程将无法一直霸占 CPU 资源，为了使其它线程也有执行的机会，CPU 会在一定时间内强制当前运行的线程让出 CPU 资源，以供其他线程使用。而对于获得 CPU 调度却没有执行完毕的线程，

就会进入阻塞状态。

目前几乎所有的桌面和服务器操作系统，都采用的是抢占式优先级调度策略。即 CPU 会给每一个就绪线程一段固定时间来处理任务，当该时间用完后，系统就会阻止该线程继续使用 CPU 资源，让其他线程获得执行的机会。而对于具体选择那个线程上 CPU，不同的平台采用不同的算法，比如先进先出算法（FIFO）、时间片轮转算法、优先级算法等，每种算法各有优缺点，适用于不同的场景。除此之外，如果处于运行状态的线程发生如下几种情况，也将会由运行状态转到阻塞状态：

- 线程调用了 `sleep()` 方法；
- 线程等待接收用户输入的数据；
- 线程试图获取某个对象的同步锁（后续章节会详细讲解）时，如果该锁被其他线程所持有，则当前线程进入阻塞状态；
- 线程调用 `wait()` 方法，等待特定条件的满足；

以上几种情况都会导致线程阻塞，只有解决了线程遇到的问题之后，该线程才会有阻塞状态转到就绪状态，继续等待 CPU 调度（如图 1 所示）。以上 4 种可能发生线程阻塞的情况，解决措施分别如下：

- `sleep()` 方法规定的时间已过；
- 线程接收到了用户输入的数据；
- 其他线程释放了该同步锁，并由该线程获得；
- 调用 `set()` 方法发出通知；

以上涉及到的线程方法以及它们的含义和用法，会在后续章节做详细讲解。

## 线程死亡状态

对于获得 CPU 调度却未执行完毕的线程，它会转入阻塞状态，待条件成熟之后继续转入就绪状态，重复争取 CPU 资源，直到其执行结束。执行结束的线程将处于死亡状态。

线程执行结束，除了正常执行结束外，如果程序执行过程发生异常（Exception）或者错误（Error），线程也会进入死亡状态。对于处于死亡状态的线程，有以下 2 点需要注意：

- 主线程死亡，并不意味着所有线程全部死亡。也就是说，主线程的死亡，不会影响子线程继续执行；反之也是如此。
- 对于死亡的线程，无法再调用 `start()` 方法使其重新启动，否则 Python 解释器将抛出 `RuntimeError` 异常。

## 4. Python Thread join()用法详解

前面章节中，我们讲解了如何通过 Thread 类创建并启动一个线程，当时给读者用如下的程序进行演示：

```
1. import threading
2. #定义线程要调用的方法，*add 可接收多个以非关键字方式传入的参数
3. def action(*add):
4.     for arc in add:
5.         #调用 getName() 方法获取当前执行该程序的线程名
6.         print(threading.current_thread().getName() + " " + arc)
7. #定义为线程方法传入的参数
8. my_tuple = ("http://c.biancheng.net/python/", \
9.             "http://c.biancheng.net/shell/", \
10.            "http://c.biancheng.net/java/")
11. #创建线程
12. thread = threading.Thread(target = action, args =my_tuple)
13. #启动线程
14. thread.start()
15. #主线程执行如下语句
16. for i in range(5):
17.     print(threading.current_thread().getName())
```

程序执行结果为（不唯一）：

```
Thread-1 http://c.biancheng.net/python/MainThread
Thread-1 http://c.biancheng.net/shell/MainThread
Thread-1 http://c.biancheng.net/java/MainThread
MainThread
MainThread
```

可以看到，我们用 Thread 类创建了一个线程（线程名为 Thread-1），其任务是执行 action() 函数。同时，我们也给主线程 MainThread 安排了循环任务（第 16、17 行）。通过前面的学习我们知道，主线程 MainThread 和子线程 Thread-1 会轮流获得 CPU 资源，因此该程序的输出结果才会向上面显示的这样。

但是，如果我们想让 Thread-1 子线程先执行，然后再让 MainThread 执行第 16、17 行代码，该如何实现呢？很简单，通过调用线程对象的 join() 方法即可。

join() 方法的功能是在程序指定位置，优先让该方法的调用者使用 CPU 资源。该方法的语法格式如下：

```
thread.join([timeout])
```

其中，thread 为 Thread 类或其子类的实例化对象；timeout 参数作为可选参数，其功能是指定 thread 线程最多可以霸占 CPU 资源的时间（以秒为单位），如果省略，则默认直到 thread 执行结束（进入死亡状态）才释放 CPU 资源。

举个例子，修改上面的代码，如下所示：

```

1.  import threading
2.  #定义线程要调用的方法，*add 可接收多个以非关键字方式传入的参数
3.  def action(*add):
4.      for arc in add:
5.          #调用 getName() 方法获取当前执行该程序的线程名
6.          print(threading.current_thread().getName() + " " + arc)
7.  #定义为线程方法传入的参数
8.  my_tuple = ("http://c.biancheng.net/python/", \
9.             "http://c.biancheng.net/shell/", \
10.            "http://c.biancheng.net/java/")
11. #创建线程
12. thread = threading.Thread(target = action, args =my_tuple)
13. #启动线程
14. thread.start()
15. #指定 thread 线程优先执行完毕
16. thread.join()
17. #主线程执行如下语句
18. for i in range(5):
19.     print(threading.current_thread().getName())

```

程序执行结果为：

```

Thread-1 http://c.biancheng.net/python/
Thread-1 http://c.biancheng.net/shell/
Thread-1 http://c.biancheng.net/java/
MainThread
MainThread
MainThread
MainThread
MainThread
MainThread

```

程序中第 16 行的位置，thread 线程调用了 join() 方法，并且没有指定具体的 timeout 参数值。这意味着如果程序想继续往下执行，必须先执行完 thread 线程。



## 5. Python daemon 守护线程详解

前面不只一次提到，当程序中拥有多个线程时，主线程执行结束并不会影响子线程继续执行。换句话说，只有程序中所有线程全部执行完毕后，程序才算真正结束。

下面程序演示了包含 2 个线程的程序执行流程：

```
1.  import threading
2.
3.  #主线程执行如下语句
4.  for i in range(5):
5.      print(threading.current_thread().getName())
6.
7.  #定义线程要调用的方法，*add 可接收多个以非关键字方式传入的参数
8.  def action(*add):
9.      for arc in add:
10.         #调用 getName() 方法获取当前执行该程序的线程名
11.         print(threading.current_thread().getName() + " " + arc)
12. #定义为线程方法传入的参数
13. my_tuple = ("http://c.biancheng.net/python/", \
14.             "http://c.biancheng.net/shell/", \
15.             "http://c.biancheng.net/java/")
16. #创建线程
17. thread = threading.Thread(target = action, args =my_tuple)
18. #启动线程
19. thread.start()
```

程序执行结果为：

```
MainThread
MainThread
MainThread
MainThread
MainThread
Thread-1 http://c.biancheng.net/python/
Thread-1 http://c.biancheng.net/shell/
Thread-1 http://c.biancheng.net/java/
```

显然，只有等 MainThread 和 Thread-1 全部执行完之后，程序才执行结束。

除此之外，Python 还支持创建另一种线程，称为**守护线程**（或**后台线程**）。此类线程的特点是，当程序中主线程及所有非守护线程执行结束时，未执行完毕的守护线程也会随之消亡（进行死亡状态），程序将结束运行。

Python 解释器的垃圾回收机制就是守护线程的典型代表，当程序中所有主线程及非守护线程执行完毕后，垃圾回收机制也就没有再继续执行的必要了。

前面章节中，我们学习了 2 种创建线程的方式，守护线程本质也是线程，因此其创建方式和普通线程一样，唯一不同之处在于，将普通线程设为守护线程，需通过线程对象调用其 `daemon` 属性，将该属性的值该为 `True`。

并且需要注意的一点是，线程对象调用 `daemon` 属性必须在调用 `start()` 方法之前，否则 Python 解释器将报 `RuntimeError` 错误。

举个例子，下面程序演示了如何创建一个守护线程：

```
1. import threading
2. #定义线程要调用的方法，*add 可接收多个以非关键字方式传入的参数
3. def action(length):
4.     for arc in range(length):
5.         #调用 getName() 方法获取当前执行该程序的线程名
6.         print(threading.current_thread().getName()+" "+str(arc))
7. #创建线程
8. thread = threading.Thread(target = action, args =(20,))
9. #将 thread 设置为守护线程
10. thread.daemon = True
11. #启动线程
12. thread.start()
13. #主线程执行如下语句
14. for i in range(5):
15.     print(threading.current_thread().getName())
```

程序中第 10 行代码处，通过调用 `thread` 线程的 `daemon` 属性并赋值为 `True`，则该 `thread` 线程就变成了守护线程。由于该程序中除了 `thread` 守护线程就只有主线程 `MainThread`，因此只要主线程执行结束，则守护线程将随之消亡。程序执行结果为：

```
Thread-1 0
MainThread
MainThread
Thread-1 1
MainThread
Thread-1 2
MainThread
MainThread
```

## 6. Python sleep()函数用法：线程睡眠

位于 time 模块中的 sleep(secs) 函数，可以实现令当前执行的线程暂停 secs 秒后再继续执行。所谓暂停，即令当前线程进入阻塞状态，当达到 sleep() 函数规定的时间后，再由阻塞状态转为就绪状态，等待 CPU 调度。

sleep() 函数位于 time 模块中，因此在使用前，需先引入 time 模块。

sleep() 函数的语法规则如下所示：

```
time.sleep(secs)
```

其中，secs 参数用于指定暂停的秒数，

仍以前面章节创建的 thread 线程为例，下面程序演示了 sleep() 函数的用法：

```
1. import threading
2. import time
3. #定义线程要调用的方法，*add 可接收多个以非关键字方式传入的参数
4. def action(*add):
5.     for arc in add:
6.         #暂停 0.1 秒后，再执行
7.         time.sleep(0.1)
8.         #调用 getName() 方法获取当前执行该程序的线程名
9.         print(threading.current_thread().getName() + " " + arc)
10. #定义为线程方法传入的参数
11. my_tuple = ("http://c.biancheng.net/python/", \
12.             "http://c.biancheng.net/shell/", \
13.             "http://c.biancheng.net/java/")
14. #创建线程
15. thread = threading.Thread(target = action, args =my_tuple)
16. #启动线程
17. thread.start()
18.
19. #主线程执行如下语句
20. for i in range(5):
21.     print(threading.current_thread().getName())
```

程序执行结果为：

```
MainThread
MainThread
MainThread
MainThread
MainThread
Thread-1 http://c.biancheng.net/python/
```

```
Thread-1 http://c.biancheng.net/shell/  
Thread-1 http://c.biancheng.net/java/
```

可以看到，和未使用 `sleep()` 函数的输出结果相比，显然主线程 `MainThread` 在前期获得 CPU 资源的次数更多，因为 `Thread-1` 线程中调用了 `sleep()` 函数，在一定程序上会阻碍该线程获得 CPU 调度。

## 7. Python 互斥锁 ( Lock )：解决多线程安全问题

多线程的优势在于并发性，即可以同时运行多个任务。但是当线程需要使用共享数据时，也可能会由于数据不同步产生“错误情况”，这是由系统的线程调度具有一定的随机性造成的。

互斥锁的作用就是解决数据不同步问题。关于互斥锁，有一个经典的“银行取钱”问题。银行取钱的基本流程可以分为如下几个步骤：

1. 用户输入账户、密码，系统判断用户的账户、密码是否匹配。
2. 用户输入取款金额。
3. 系统判断账户余额是否大于取款金额。
4. 如果余额大于取款金额，则取款成功；如果余额小于取款金额，则取款失败。

乍一看上去，这确实就是日常生活中的取款流程，这个流程没有任何问题。但一旦将这个流程放在多线程并发的场景下，就有可能出现问题。注意，此处说的是有可能，并不是一定。也许你的程序运行了一百万次都没有出现问题，但没有出现问题并不等于没有问题！

按照上面的流程编写取款程序，并使用两个线程分别模拟两个人使用同一个账户做并发取钱操作。此处忽略检查账户和密码的操作，仅仅模拟后面三步操作。下面先定义一个账户类，该账户类封装了账户编号和余额两个成员变量。

```
1. class Account:
2.     # 定义构造器
3.     def __init__(self, account_no, balance):
4.         # 封装账户编号、账户余额的两个成员变量
5.         self.account_no = account_no
6.         self.balance = balance
```

接下来程序会定义一个模拟取钱的函数，该函数根据执行账户、取钱数量进行取钱操作，取钱的逻辑是当账户余额不足时无法提取现金，当余额足够时系统吐出钞票，余额减少。

程序的主程序非常简单，仅仅是创建一个账户，并启动两个线程从该账户中取钱。程序如下：

```
1. import threading
2. import time
3. import Account
4.
5. # 定义一个函数来模拟取钱操作
6. def draw(account, draw_amount):
7.     # 账户余额大于取钱数目
8.     if account.balance >= draw_amount:
9.         # 吐出钞票
10.        print(threading.current_thread().name\
11.              + "取钱成功! 吐出钞票:" + str(draw_amount))
12.    #     time.sleep(0.001)
13.    # 修改余额
```

```

14.         account.balance -= draw_amount
15.         print("\t 余额为: " + str(account.balance))
16.     else:
17.         print(threading.current_thread().name\
18.             + "取钱失败! 余额不足!")
19. # 创建一个账户
20. acct = Account.Account("1234567", 1000)
21. # 模拟两个线程对同一个账户取钱
22. threading.Thread(name='甲', target=draw, args=(acct, 800)).start()
23. threading.Thread(name='乙', target=draw, args=(acct, 800)).start()

```

先不要管程序中第 12 行被注释掉的代码，上面程序是一个非常简单的取钱逻辑，这个取钱逻辑与实际的取钱操作也很相似。

多次运行上面程序，很有可能都会看到如图 1 所示的错误结果。



```

Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\mengma\Desktop\1.py =====
甲取钱成功! 吐出钞票:800
>>> 乙取钱成功! 吐出钞票:800

        余额为: 200        余额为: -600

```

图 1 线程安全问题

如图 1 所示的运行结果并不是银行所期望的结果（不过有可能看到正确的运行结果），这正是多线程编程突然出现的“偶然”错误因为线程调度的不确定性。

假设系统线程调度器在注释代码处暂停，让另一个线程执行（为了强制暂停，只要取消程序中注释代码前的注释即可）。取消注释后，再次运行程序，将总可以看到如图 1 所示的错误结果。

问题出现了，账户余额只有 1000 元时取出了 1600 元，而且账户余额出现了负值，远不是银行所期望的结果。虽然上面程序是人为地使用 `time.sleep(0.001)` 来强制线程调度切换，但这种切换也是完全可能发生的（100000 次操作只要有 1 次出现了错误，那就是由编程错误引起的）。

## Python 互斥锁同步线程

之所以出现如图 1 所示的错误结果，是因为 `run()` 方法的方法体不具有线程安全性，程序中有两个并发线程在修改 `Account` 对象，而且系统恰好在注释代码处执行线程切换，切换到另一个修改 `Account` 对象的线程，所以就出现了问题。

为了解决这个问题，Python 的 `threading` 模块引入了互斥锁（`Lock`）。`threading` 模块提供了 `Lock` 和 `RLock` 两个类，它们都提供了如下两个方法来加互斥锁和释放互斥锁：

1. `acquire(blocking=True, timeout=-1)`：请求对 `Lock` 或 `RLock` 加锁，其中 `timeout` 参数指定加锁多少秒。
2. `release()`：释放锁。

`Lock` 和 `RLock` 的区别如下：

- `threading.Lock`：它是一个基本的锁对象，每次只能锁定一次，其余的锁请求，需等待锁释放后才能获取。
- `threading.RLock`：它代表可重入锁（`Reentrant Lock`）。对于可重入锁，在同一个线程中可以对它进行多次锁定，也可以多次释放。如果使用 `RLock`，那么 `acquire()` 和 `release()` 方法必须成对出现。如果调用了 `n` 次 `acquire()` 加锁，则必须调用 `n` 次 `release()` 才能释放锁。

由此可见，`RLock` 锁具有可重入性。也就是说，同一个线程可以对已被加锁的 `RLock` 锁再次加锁，`RLock` 对象会维持一个计数器来追踪 `acquire()` 方法的嵌套调用，线程在每次调用 `acquire()` 加锁后，都必须显式调用 `release()` 方法来释放锁。所以，一段被锁保护的方法可以调用另一个被相同锁保护的方法。

`Lock` 是控制多个线程对共享资源进行访问的工具。通常，锁提供了对共享资源的独占访问，每次只能有一个线程对 `Lock` 对象加锁，线程在开始访问共享资源之前应先请求获得 `Lock` 对象。当对共享资源访问完成后，程序释放对 `Lock` 对象的锁定。

在实现线程安全的控制中，比较常用的是 `RLock`。通常使用 `RLock` 的代码格式如下：

```
class X:

    #定义需要保证线程安全的方法

    def m () :

        #加锁

        self.lock.acquire()

        try :

            #需要保证线程安全的代码

            #...方法体

        #使用 finally 块来保证释放锁

        finally :

            #修改完成，释放锁

            self.lock.release()
```

使用 `RLock` 对象来控制线程安全，当加锁和释放锁出现在不同的作用范围内时，通常建议使用 `finally` 块来确保在必要时释放锁。

通过使用 `Lock` 对象可以非常方便地实现线程安全的类，线程安全的类具有如下特征：

- 该类的对象可以被多个线程安全地访问。
- 每个线程在调用该对象的任意方法之后，都将得到正确的结果。
- 每个线程在调用该对象的任意方法之后，该对象都依然保持合理的状态。

总的来说，不可变类总是线程安全的，因为它的对象状态不可改变；但可变对象需要额外的方法来保证其线程安全。例如，上面的 Account 就是一个可变类，它的 self.account\_no 和 self.\_balance（为了更好地封装，将 balance 改名为 \_balance）两个成员变量都可以被改变，当两个线程同时修改 Account 对象的 self.\_balance 成员变量的值时，程序就出现了异常。下面将 Account 类对 self.balance 的访问设置成线程安全的，那么只需对修改 self.balance 的方法增加线程安全的控制即可。

将 Account 类改为如下形式，它就是线程安全的：

```
1.  import threading
2.  import time
3.
4.  class Account:
5.      # 定义构造器
6.      def __init__(self, account_no, balance):
7.          # 封装账户编号、账户余额的两个成员变量
8.          self.account_no = account_no
9.          self._balance = balance
10.         self.lock = threading.RLock()
11.
12.         # 因为账户余额不允许随便修改，所以只为 self._balance 提供 getter 方法
13.         def getBalance(self):
14.             return self._balance
15.         # 提供一个线程安全的 draw() 方法来完成取钱操作
16.         def draw(self, draw_amount):
17.             # 加锁
18.             self.lock.acquire()
19.             try:
20.                 # 账户余额大于取钱数目
21.                 if self._balance >= draw_amount:
22.                     # 吐出钞票
23.                     print(threading.current_thread().name\
24.                           + "取钱成功！吐出钞票：" + str(draw_amount))
25.                     time.sleep(0.001)
26.                     # 修改余额
27.                     self._balance -= draw_amount
28.                     print("\t 余额为：" + str(self._balance))
29.             else:
30.                 print(threading.current_thread().name\
31.                       + "取钱失败！余额不足！")
```



```
32.         finally:
33.             # 修改完成，释放锁
34.             self.lock.release()
```

上面程序中的定义了一个 RLock 对象。在程序中实现 draw() 方法时，进入该方法开始执行后立即请求对 RLock 对象加锁，当执行完 draw() 方法的取钱逻辑之后，程序使用 finally 块来确保释放锁。

程序中 RLock 对象作为同步锁，线程每次开始执行 draw() 方法修改 self.balance 时，都必须先对 RLock 对象加锁。当该线程完成对 self.balance 的修改，将要退出 draw() 方法时，则释放对 RLock 对象的锁定。这样的做法完全符合“加锁→修改→释放锁”的安全访问逻辑。

当一个线程在 draw() 方法中对 RLock 对象加锁之后，其他线程由于无法获取对 RLock 对象的锁定，因此它们同时执行 draw() 方法对 self.balance 进行修改。这意味着，并发线程在任意时刻只有一个线程可以进入修改共享资源的代码区（也被称为临界区），所以在同一时刻最多只有一个线程处于临界区内，从而保证了线程安全。

为了保证 Lock 对象能真正“锁定”它所管理的 Account 对象，程序会被编写成每个 Account 对象有一个对应的 Lock（就像一个房间有一个锁一样）。

上面的 Account 类增加了一个代表取钱的 draw() 方法，并使用 Lock 对象保证该 draw() 方法的线程安全，而且取消了 setBalance() 方法（避免程序直接修改 self.balance 成员变量），因此线程执行体只需调用 Account 对象的 draw() 方法即可执行取钱操作。

下面程序创建并启动了两个取钱线程：

```
1. import threading
2. import Account
3.
4. # 定义一个函数来模拟取钱操作
5. def draw(account, draw_amount):
6.     # 直接调用 account 对象的 draw() 方法来执行取钱操作
7.     account.draw(draw_amount)
8. # 创建一个账户
9. acct = Account.Account("1234567", 1000)
10. # 模拟两个线程对同一个账户取钱
11. threading.Thread(name='甲', target=draw, args=(acct, 800)).start()
12. threading.Thread(name='乙', target=draw, args=(acct, 800)).start()
```

上面程序中代表线程执行体的 draw() 函数无须自己实现取钱操作，而是直接调用 account 的 draw() 方法来执行取钱操作。由于 draw() 方法已经使用 RLock 对象实现了线程安全，因此上面程序就不会导致线程安全问题。

多次重复运行上面程序，总可以看到如图 2 所示的运行结果。



```
*Python 3.6.2 Shell*
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\mengma\Desktop\1.py =====
>>> 甲取钱成功! 吐出钞票:800
      余额为: 200
乙取钱失败! 余额不足!
Ln: 5 Col: 4
```

图 2 使用互斥锁保证线程安全

可变类的线程安全是以降低程序的运行效率作为代价的，为了减少线程安全所带来的负面影响，程序可以采用如下策略：

- 不要对线程安全类的所有方法都进行同步，只对那些会改变竞争资源（竞争资源也就是共享资源）的方法进行同步。例如，上面 Account 类中的 account\_no 实例变量就无须同步，所以程序只对 draw() 方法进行了同步控制。
- 如果可变类有两种运行环境，单线程环境和多线程环境，则应该为该可变类提供两种版本，即线程不安全版本和线程安全版本。在单线程环境中使用线程不安全版本以保证性能，在多线程环境中使用线程安全版本。

## 8.什么是死锁，如何避免死锁（4 种方法）

当两个线程相互等待对方释放资源时，就会发生死锁。Python 解释器没有监测，也不会主动采取措施来处理死锁情况，所以在进行多线程编程时应该采取措施避免出现死锁。

一旦出现死锁，整个程序既不会发生任何异常，也不会给出任何提示，只是所有线程都处于阻塞状态，无法继续。

死锁是很容易发生的，尤其是在系统中出现多个同步监视器的情况下，如下程序将会出现死锁：

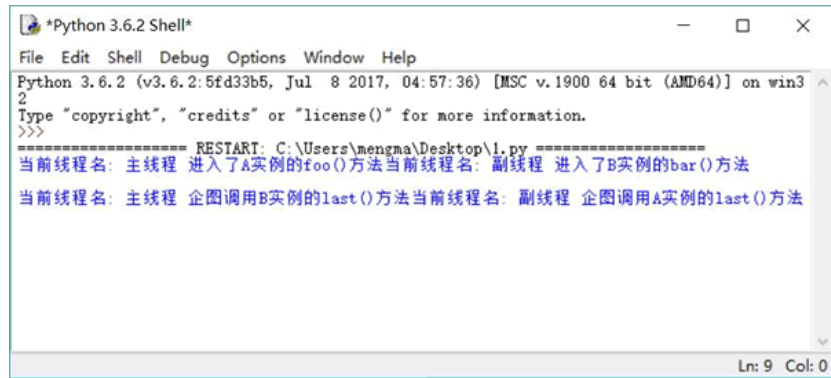
```
1.  import threading
2.  import time
3.
4.  class A:
5.      def __init__(self):
6.          self.lock = threading.RLock()
7.      def foo(self, b):
8.          try:
9.              self.lock.acquire()
10.             print("当前线程名: " + threading.current_thread().name\
11.                 + " 进入了 A 实例的 foo()方法" )      # ①
12.             time.sleep(0.2)
13.             print("当前线程名: " + threading.current_thread().name\
14.                 + " 企图调用 B 实例的 last()方法")    # ③
15.             b.last()
16.         finally:
17.             self.lock.release()
18.     def last(self):
19.         try:
20.             self.lock.acquire()
21.             print("进入了 A 类的 last()方法内部")
22.         finally:
23.             self.lock.release()
24. class B:
25.     def __init__(self):
26.         self.lock = threading.RLock()
27.     def bar(self, a):
28.         try:
29.             self.lock.acquire()
```

```

30.         print("当前线程名: " + threading.current_thread().name\
31.               + " 进入了 B 实例的 bar()方法" )    # ②
32.         time.sleep(0.2)
33.         print("当前线程名: " + threading.current_thread().name\
34.               + " 企图调用 A 实例的 last()方法" )    # ④
35.         a.last()
36.     finally:
37.         self.lock.release()
38.
39.     def last(self):
40.         try:
41.             self.lock.acquire()
42.             print("进入了 B 类的 last()方法内部")
43.             finally:
44.                 self.lock.release()
45.
46. a = A()
47. b = B()
48.
49. def init():
50.     threading.current_thread().name = "主线程"
51.     # 调用 a 对象的 foo() 方法
52.     a.foo(b)
53.     print("进入了主线程之后")
54.
55. def action():
56.     threading.current_thread().name = "副线程"
57.     # 调用 b 对象的 bar() 方法
58.     b.bar(a)
59.     print("进入了副线程之后")
60.
61. # 以 action 为 target 启动新线程
62. threading.Thread(target=action).start()
63.
64. # 调用 init()函数
65. init()

```

运行上面程序，将会看到如图 1 所示的效果。



```
*Python 3.6.2 Shell*
File Edit Shell Debug Options Window Help
Python 3.6.2 (v3.6.2:5fd33b5, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\mengma\Desktop\l.py =====
当前线程名: 主线程 进入了A实例的foo()方法
当前线程名: 副线程 进入了B实例的bar()方法
当前线程名: 主线程 企图调用B实例的last()方法
Ln: 9 Col: 0
```

图 1 死锁效果

从图 1 中可以看出，程序既无法向下执行，也不会抛出任何异常，就一直“僵持”着。究其原因，是因为上面程序中 A 对象和 B 对象的方法都是线程安全的方法。

程序中有两个线程执行，副线程的线程执行体是 action() 函数，主线程的线程执行体是 init() 函数（主程序调用了 init() 函数）。其中在 action() 函数中让 B 对象调用 bar() 方法，而在 init() 函数中让 A 对象调用 foo() 方法。

图 1 显示 action() 函数先执行，调用了 B 对象的 bar() 方法，在进入 bar() 方法之前，该线程对 B 对象的 Lock 加锁（当程序执行到 ② 号代码时，副线程暂停 0.2s）；CPU 切换到执行另一个线程，让 A 对象执行 foo() 方法，所以看到主线程开始执行 A 实例的 foo() 方法，在进入 foo() 方法之前，该线程对 A 对象的 Lock 加锁（当程序执行到 ① 号代码时，主线程也暂停 0.2s）。

接下来副线程会先醒过来，继续向下执行，直到执行到 ④ 号代码处希望调用 A 对象的 last() 方法（在执行该方法之前，必须先对 A 对象的 Lock 加锁），但此时主线程正保持着 A 对象的 Lock 的锁定，所以副线程被阻塞。

接下来主线程应该也醒过来了，继续向下执行，直到执行到 ③ 号代码处希望调用 B 对象的 last() 方法（在执行该方法之前，必须先对 B 对象的 Lock 加锁），但此时副线程没有释放对 B 对象的 Lock 的锁定。

至此，就出现了主线程保持着 A 对象的锁，等待对 B 对象加锁，而副线程保持着 B 对象的锁，等待对 A 对象加锁，两个线程互相等待对方先释放锁，所以就出现了死锁。

死锁是不应该在程序中出现的，在编写程序时应该尽量避免出现死锁。下面有几种常见的方式用来解决死锁问题：

1. 避免多次锁定。尽量避免同一个线程对多个 Lock 进行锁定。例如上面的死锁程序，主线程要对 A、B 两个对象的 Lock 进行锁定，副线程也要对 A、B 两个对象的 Lock 进行锁定，这就埋下了导致死锁的隐患。
2. 具有相同的加锁顺序。如果多个线程需要对多个 Lock 进行锁定，则应该保证它们以相同的顺序请求加锁。比如上面的死锁程序，主线程先对 A 对象的 Lock 加锁，再对 B 对象的 Lock 加锁；而副线程则先对 B 对象的 Lock 加锁，再对 A 对象的 Lock 加锁。这种加锁顺序很容易形成嵌套锁定，进而导致死锁。如果让主线程、副线程按照相同的顺序加锁，就可以避免这个问题。
3. 使用定时锁。程序在调用 acquire() 方法加锁时可指定 timeout 参数，该参数指定超过 timeout 秒后会自动释放对 Lock 的锁定，这样就可以解开死锁了。
4. 死锁检测。死锁检测是一种依靠算法机制来实现的死锁预防机制，它主要是针对那些不可能实现按序加锁，也不能使用定时锁的场景的。

## 9. Python condition 实现线程通信（详解版）

当线程在系统中运行时，线程的调度具有一定的透明性，通常程序无法准确控制线程的轮换执行，如果有需要，Python 可通过线程通信来保证线程协调运行。

假设系统中有两个线程，这两个线程分别代表存款者和取钱者，现在假设系统有一种特殊的要求，即要求存款者和取钱者不断地重复存款、取钱的动作，而且要求每当存款者将钱存入指定账户后，取钱者就立即取出该笔钱。不允许存款者连续两次存钱，也不允许取钱者连续两次取钱。

为了实现这种功能，可以借助于 **Condition** 对象来保持协调。使用 **Condition** 可以让那些已经得到 **Lock** 对象却无法继续执行的线程释放 **Lock** 对象，**Condition** 对象也可以唤醒其他处于等待状态的线程。

将 **Condition** 对象与 **Lock** 对象组合使用，可以为每个对象提供多个等待集（wait-set）。因此，**Condition** 对象总是需要有对应的 **Lock** 对象。从 **Condition** 的构造器 `__init__(self, lock=None)` 可以看出，程序在创建 **Condition** 时可通过 `lock` 参数传入要绑定的 **Lock** 对象；如果不指定 `lock` 参数，在创建 **Condition** 时它会自动创建一个与之绑定的 **Lock** 对象。

**Condition** 类提供了如下几个方法：

- `acquire([timeout])/release()`：调用 **Condition** 关联的 **Lock** 的 `acquire()` 或 `release()` 方法。
- `wait([timeout])`：导致当前线程进入 **Condition** 的等待池等待通知并释放锁，直到其他线程调用该 **Condition** 的 `notify()` 或 `notify_all()` 方法来唤醒该线程。在调用该 `wait()` 方法时可传入一个 `timeout` 参数，指定该线程最多等待多少秒。
- `notify()`：唤醒在该 **Condition** 等待池中的单个线程并通知它，收到通知的线程将自动调用 `acquire()` 方法尝试加锁。如果所有线程都在该 **Condition** 等待池中等待，则会选择唤醒其中一个线程，选择是任意性的。
- `notify_all()`：唤醒在该 **Condition** 等待池中等待的所有线程并通知它们。

本例程序中，可以通过一个旗标来标识账户中是否已有存款，当旗标为 `False` 时，表明账户中没有存款，存款者线程可以向下执行，当存款者把钱存入账户中后，将旗标设为 `True`，并调用 **Condition** 的 `notify()` 或 `notify_all()` 方法来唤醒其他线程。

当存款者线程进入线程体后，如果旗标为 `True`，就调用 **Condition** 的 `wait()` 方法让该线程等待。当旗标为 `True` 时，表明账户中已经存入了钱，取钱者线程可以向下执行，当取钱者把钱从账户中取出后，将旗标设为 `False`，并调用 **Condition** 的 `notify()` 或 `notify_all()` 方法来唤醒其他线程；当取钱者线程进入线程体后，如果旗标为 `False`，就调用 `wait()` 方法让该线程等待。

本程序为 **Account** 类提供了 `draw()` 和 `deposit()` 两个方法，分别对应于该账户的取钱和存款操作。因为这两个方法可能需要并发修改 **Account** 类的 `self.balance` 成员变量的值，所以它们都使用 **Lock** 来控制线程安全。除此之外，这两个方法还使用了 **Condition** 的 `wait()` 和 `notify_all()` 来控制线程通信。

```
1.  import threading
2.
3.  class Account:
4.      # 定义构造器
5.
6.      def __init__(self, account_no, balance):
7.
8.          # 封装账户编号、账户余额的两个成员变量
9.
10.         self.account_no = account_no
11.
12.         self._balance = balance
13.
14.         self.cond = threading.Condition()
15.
16.         # 定义代表是否已经存钱的旗标
17.
18.         self._flag = False
```

```
13.     # 因为账户余额不允许随便修改, 所以只为 self._balance 提供 getter 方法
14.     def getBalance(self):
15.         return self._balance
16.     # 提供一个线程安全的 draw() 方法来完成取钱操作
17.     def draw(self, draw_amount):
18.         # 加锁, 相当于调用 Condition 绑定的 Lock 的 acquire()
19.         self.cond.acquire()
20.         try:
21.             # 如果 self._flag 为假, 表明账户中还没有人存钱进去, 取钱方法阻塞
22.             if not self._flag:
23.                 self.cond.wait()
24.             else:
25.                 # 执行取钱操作
26.                 print(threading.current_thread().name
27.                       + " 取钱:" + str(draw_amount))
28.                 self._balance -= draw_amount
29.                 print("账户余额为: " + str(self._balance))
30.                 # 将标识账户是否已有存款的旗标设为 False
31.                 self._flag = False
32.                 # 唤醒其他线程
33.                 self.cond.notify_all()
34.         # 使用 finally 块来释放锁
35.         finally:
36.             self.cond.release()
37.     def deposit(self, deposit_amount):
38.         # 加锁, 相当于调用 Condition 绑定的 Lock 的 acquire()
39.         self.cond.acquire()
40.         try:
41.             # 如果 self._flag 为真, 表明账户中已有人存钱进去, 存钱方法阻塞
42.             if self._flag:                # ①
43.                 self.cond.wait()
44.             else:
45.                 # 执行存款操作
46.                 print(threading.current_thread().name\
47.                       + " 存款:" + str(deposit_amount))
48.                 self._balance += deposit_amount
```

```

49.         print("账户余额为: " + str(self._balance))
50.
51.         # 将表示账户是否已有存款的旗标设为 True
52.         self._flag = True
53.
54.         # 唤醒其他线程
55.         self.cond.notify_all()
56.
57.         # 使用 finally 块来释放锁
58.         finally:
59.             self.cond.release()

```

上面程序使用 Condition 的 wait() 和 notify\_all() 方法进行控制，对存款者线程而言，当程序进入 deposit() 方法后，如果 self.\_flag 为 True，则表明账户中已有存款，程序调用 Condition 的 wait() 方法被阻塞；否则，程序向下执行存款操作，当存款操作执行完成后，系统将 self.\_flag 设为 True，然后调用 notify\_all() 来唤醒其他被阻塞的线程。如果系统中有存款者线程，存款者线程也会被唤醒，但该存款者线程执行到 ① 号代码处时再次进入阻塞状态，只有执行 draw() 方法的取钱者线程才可以向下执行。同理，取钱者线程的运行流程也是如此。

程序中的存款者线程循环 100 次重复存款，而取钱者线程则循环 100 次重复取钱，存款者线程和取钱者线程分别调用 Account 对象的 deposit()、draw() 方法来实现。主程序可以启动任意多个“存款”线程和“取钱”线程，可以看到所有的“取钱”线程必须等“存款”线程存钱后才可以向下执行，而“存款”线程也必须等“取钱”线程取钱后才可以向下执行。主程序代码如下：

```

1.  import threading
2.  import Account
3.
4.  # 定义一个函数，模拟重复 max 次执行取钱操作
5.  def draw_many(account, draw_amount, max):
6.      for i in range(max):
7.          account.draw(draw_amount)
8.
9.  # 定义一个函数，模拟重复 max 次执行存款操作
10. def deposit_many(account, deposit_amount, max):
11.     for i in range(max):
12.         account.deposit(deposit_amount)
13.
14. # 创建一个账户
15. acct = Account.Account("1234567", 0)
16.
17. # 创建、并启动一个“取钱”线程
18. threading.Thread(name="取钱者", target=draw_many,
19.                   args=(acct, 800, 100)).start()
20.
21. # 创建、并启动一个“存款”线程
22. threading.Thread(name="存款者甲", target=deposit_many,
23.                   args=(acct, 800, 100)).start();
24.
25. threading.Thread(name="存款者乙", target=deposit_many,
26.                   args=(acct, 800, 100)).start()

```



```
22.     threading.Thread(name="存款者丙", target=deposit_many,  
23.         args=(acct , 800, 100)).start()
```

运行该程序，可以看到存款者线程、取钱者线程交替执行的情形，每当存款者向账户中存入 800 元之后，取钱者线程就立即从账户中取出这笔钱。存款完成后账户余额总是 800 元，取钱结束后账户余额总是 0 元。

运行该程序，将会看到如图 1 所示的结果。

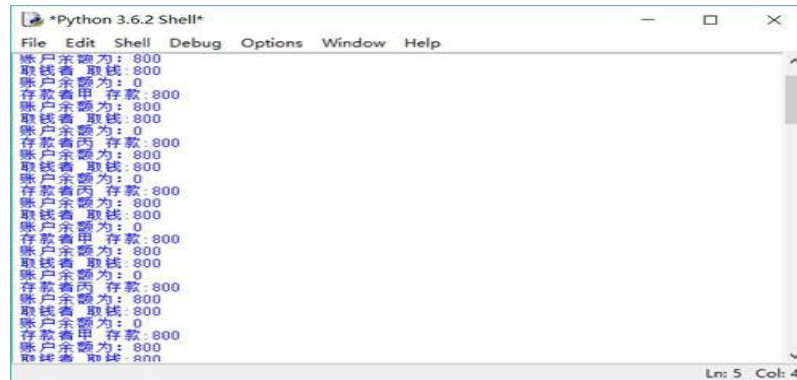


图 1 线程通信的效果

从图 1 中可以看出，3 个存款者线程随机地向账户中存钱，只有 1 个取钱者线程执行取钱操作。只有当取钱者线程取钱后，存款者线程才可以存钱；同理，只有等存款者线程存钱后，取钱者线程才可以取钱。

图 1 显示程序最后被阻塞无法继续向下执行。这是因为 3 个存款者线程共有 300 次尝试存钱操作，但 1 个取钱者线程只有 100 次尝试取钱操作，所以程序最后被阻塞。

如图 1 所示的阻塞并不是死锁，对于这种情况，取钱者线程已经执行结束，而存款者线程只是在等待其他线程来取钱而已，并不是等待其他线程释放同步监视器。不要把死锁和程序阻塞等同起来。

# 10. Python Queue 队列实现线程通信

queue 模块下提供了几个阻塞队列，这些队列主要用于实现线程通信。在 queue 模块下主要提供了三个类，分别代表三种队列，它们的主要区别就在于进队列、出队列的不同。

关于这三个队列类的简单介绍如下：

1. queue.Queue(maxsize=0)：代表 FIFO（先进先出）的常规队列，maxsize 可以限制队列的大小。如果队列的大小达到队列的上限，就会加锁，再次加入元素时就会被阻塞，直到队列中的元素被消费。如果将 maxsize 设置为 0 或负数，则该队列的大小就是无限制的。
2. queue.LifoQueue(maxsize=0)：代表 LIFO（后进先出）的队列，与 Queue 的区别就是出队列的顺序不同。
3. PriorityQueue(maxsize=0)：代表优先级队列，优先级最小的元素先出队列。

这三个队列类的属性和方法基本相同，它们都提供了如下属性和方法：

- Queue.qsize()：返回队列的实际大小，也就是该队列中包含几个元素。
- Queue.empty()：判断队列是否为空。
- Queue.full()：判断队列是否已满。
- Queue.put(item, block=True, timeout=None)：向队列中放入元素。如果队列已满，且 block 参数为 True（阻塞），当前线程被阻塞，timeout 指定阻塞时间，如果将 timeout 设置为 None，则代表一直阻塞，直到该队列的元素被消费；如果队列已满，且 block 参数为 False（不阻塞），则直接引发 queue.FULL 异常。
- Queue.put\_nowait(item)：向队列中放入元素，不阻塞。相当于在上一个方法中将 block 参数设置为 False。
- Queue.get(item, block=True, timeout=None)：从队列中取出元素（消费元素）。如果队列已满，且 block 参数为 True（阻塞），当前线程被阻塞，timeout 指定阻塞时间，如果将 timeout 设置为 None，则代表一直阻塞，直到有元素被放入队列中；如果队列已空，且 block 参数为 False（不阻塞），则直接引发 queue.EMPTY 异常。
- Queue.get\_nowait(item)：从队列中取出元素，不阻塞。相当于在上一个方法中将 block 参数设置为 False。

下面以普通的 Queue 为例介绍阻塞队列的功能和用法。首先用一个最简单的程序来测试 Queue 的 put() 和 get() 方法。

```
1. import queue
2.
3. # 定义一个长度为 2 的阻塞队列
4. bq = queue.Queue(2)
5. bq.put("Python")
6. bq.put("Python")
7. print("1111111111")
8. bq.put("Python") # ① 阻塞线程
9. print("2222222222")
```

上面程序先定义了一个大小为 2 的 Queue，程序先向该队列中放入两个元素，此时队列还没有满，两个元素都可以被放入。当程序试图放入第三个元素时，如果使用 put() 方法尝试放入元素将会阻塞线程，如上面程序中 ① 号代码所示。

与此类似的是，在 Queue 已空的情况下，程序使用 get() 方法尝试取出元素将会阻塞线程。

在掌握了 Queue 阻塞队列的特性之后，在下面程序中就可以利用 Queue 来实现线程通信了。

```
1. import threading
2. import time
```

```

3. import queue
4.
5. def product(bq):
6.     str_tuple = ("Python", "Kotlin", "Swift")
7.     for i in range(99999):
8.         print(threading.current_thread().name + "生产者准备生产元组元素!")
9.         time.sleep(0.2);
10.        # 尝试放入元素, 如果队列已满, 则线程被阻塞
11.        bq.put(str_tuple[i % 3])
12.        print(threading.current_thread().name \
13.              + "生产者生产元组元素完成!")
14.    def consume(bq):
15.        while True:
16.            print(threading.current_thread().name + "消费者准备消费元组元素!")
17.            time.sleep(0.2)
18.            # 尝试取出元素, 如果队列已空, 则线程被阻塞
19.            t = bq.get()
20.            print(threading.current_thread().name \
21.                  + "消费者消费[ %s ]元素完成!" % t)
22.        # 创建一个容量为 1 的 Queue
23.        bq = queue.Queue(maxsize=1)
24.        # 启动 3 个生产者线程
25.        threading.Thread(target=product, args=(bq, )).start()
26.        threading.Thread(target=product, args=(bq, )).start()
27.        threading.Thread(target=product, args=(bq, )).start()
28.        # 启动一个消费者线程
29.        threading.Thread(target=consume, args=(bq, )).start()

```

上面程序启动了三个生产者线程向 Queue 队列中放入元素, 启动了三个消费者线程从 Queue 队列中取出元素。本程序中 Queue 队列的大小为 1, 因此三个生产者线程无法连续放入元素, 必须等待消费者线程取出一个元素后, 其中的一个生产者线程才能放入一个元素。

运行该程序, 将会看到如图 1 所示的结果。



```
*Python 3.6.2 Shell*
File Edit Shell Debug Options Window Help
Thread-4消费者准备消费元组元素！ Thread-2生产者准备生产元组元素！
Thread-4消费者消费[ Python ]元素完成！ Thread-1生产者生产元组元素完成！
Thread-4消费者准备消费元组元素！ Thread-1生产者准备生产元组元素！
Thread-4消费者消费[ Swift ]元素完成！ Thread-3生产者生产元组元素完成！
Thread-4消费者准备消费元组元素！ Thread-3生产者准备生产元组元素！
Thread-4消费者消费[ Kotlin ]元素完成！ Thread-2生产者生产元组元素完成！
Thread-4消费者准备消费元组元素！ Thread-2生产者准备生产元组元素！
Thread-4消费者消费[ Kotlin ]元素完成！ Thread-1生产者生产元组元素完成！
Thread-4消费者准备消费元组元素！ Thread-1生产者准备生产元组元素！
|
Ln: 5 Col: 4
```

图 1 使用 Queue 控制线程通信

从图 1 可以看出，三个生产者线程都想向 Queue 中放入元素，但只要其中一个生产者线程向该队列中放入元素之后，其他生产者线程就必须等待，等待消费者线程取出 Queue 队列中的元素。

# 11. Python Event 实现线程通信

**Event** 是一种非常简单的线程通信机制，一个线程发出一个 Event，另一个线程可通过该 Event 被触发。

Event 本身管理一个内部旗标，程序可以通过 Event 的 `set()` 方法将该旗标设置为 `True`，也可以调用 `clear()` 方法将该旗标设置为 `False`。程序可以调用 `wait()` 方法来阻塞当前线程，直到 Event 的内部旗标被设置为 `True`。

Event 提供了如下方法：

- `is_set()`：该方法返回 Event 的内部旗标是否为 `True`。
- `set()`：该方法将会把 Event 的内部旗标设置为 `True`，并唤醒所有处于等待状态的线程。
- `clear()`：该方法将 Event 的内部旗标设置为 `False`，通常接下来会调用 `wait()` 方法来阻塞当前线程。
- `wait(timeout=None)`：该方法会阻塞当前线程。

下面程序示范了 Event 最简单的用法：

```
1. import threading
2. import time
3.
4. event = threading.Event()
5. def cal(name):
6.     # 等待事件，进入等待阻塞状态
7.     print('%s 启动' % threading.currentThread().getName())
8.     print('%s 准备开始计算状态' % name)
9.     event.wait()    # ①
10.    # 收到事件后进入运行状态
11.    print('%s 收到通知了.' % threading.currentThread().getName())
12.    print('%s 正式开始计算!' % name)
13. # 创建并启动两条，它们都会①号代码处等待
14. threading.Thread(target=cal, args=('甲',)).start()
15. threading.Thread(target=cal, args=('乙',)).start()
16. time.sleep(2)    #②
17. print('-----')
18. # 发出事件
19. print('主线程发出事件')
20. event.set()
```

上面程序以 `cal()` 函数为 `target`，创建并启动了两个线程。由于 `cal()` 函数在 ① 号代码处调用了 Event 的 `wait()`，因此两个线程执行到 ① 号代码处都会进入阻塞状态；即使主线程在 ② 号代码处被阻塞，两个子线程也不会向下执行。

直到主程序执行到最后一行，程序调用了 Event 的 `set()` 方法将 Event 的内部旗标设置为 `True`，并唤醒所有等待的线程，这两个线程才能向下执行。

运行上面程序，将看到如下输出结果：

```
Thread-1 启动
甲 准备开始计算状态
Thread-2 启动
乙 准备开始计算状态
-----
主线程发出事件
Thread-1 收到通知了.
Thread-2 收到通知了.
甲 正式开始计算！
乙 正式开始计算！
```

上面程序还没有使用 Event 的内部旗标，如果结合 Event 的内部旗标，同样可实现前面的 Account 的生产者-消费者效果：存钱线程（生产者）存钱之后，必须等取钱线程（消费者）取钱之后才能继续向下执行。

Event 实际上优点类似于 Condition 和旗标的结合体，但 Event 本身并不带 Lock 对象，因此如果要实现线程同步，还需要额外的 Lock 对象。

下面是使用 Event 改写后的 Account：

```
1.
2. import threading
3.
4. class Account:
5.     # 定义构造器
6.     def __init__(self, account_no, balance):
7.         # 封装账户编号、账户余额的两个成员变量
8.         self.account_no = account_no
9.         self._balance = balance
10.        self.lock = threading.Lock()
11.        self.event = threading.Event()
12.        # 因为账户余额不允许随便修改，所以只为 self._balance 提供 getter 方法
13.        def getBalance(self):
14.            return self._balance
15.        # 提供一个线程安全的 draw() 方法来完成取钱操作
16.        def draw(self, draw_amount):
17.            # 加锁
18.            self.lock.acquire()
19.            # 如果 Event 内部旗标为 True，表明账户中已有人存钱进去
20.            if self.event.is_set():
21.                # 执行取钱操作
22.                print(threading.current_thread().name
```

```
23.         + " 取钱:" + str(draw_amount))
24.
25.     self._balance -= draw_amount
26.
27.     print("账户余额为: " + str(self._balance))
28.
29.     # 将 Event 内部旗标设为 False
30.
31.     self.event.clear()
32.
33.     # 释放加锁
34.
35.     self.lock.release()
36.
37.     # 阻塞当前线程阻塞
38.
39.     self.event.wait()
40.
41. else:
42.
43.     # 释放加锁
44.
45.     self.lock.release()
46.
47.     # 阻塞当前线程阻塞
48.
49.     self.event.wait()
50.
51. def deposit(self, deposit_amount):
52.
53.     # 加锁
54.
55.     self.lock.acquire()
56.
57.     # 如果 Event 内部旗标为 False, 表明账户中还没有人存钱进去
58.
59.     if not self.event.is_set():
60.
61.         # 执行存款操作
62.
63.         print(threading.current_thread().name\
64.
65.             + " 存款:" + str(deposit_amount))
66.
67.         self._balance += deposit_amount
68.
69.         print("账户余额为: " + str(self._balance))
70.
71.         # 将 Event 内部旗标设为 True
72.
73.         self.event.set()
74.
75.         # 释放加锁
76.
77.         self.lock.release()
78.
79.         # 阻塞当前线程阻塞
80.
81.         self.event.wait()
82.
83.     else:
84.
85.         # 释放加锁
86.
87.         self.lock.release()
88.
89.         # 阻塞当前线程阻塞
90.
91.         self.event.wait()
```

# 12 . Python 线程池及其原理和使用（超级详细）

系统启动一个新线程的成本是比较高的，因为它涉及与操作系统的交互。在这种情形下，使用线程池可以很好地提升性能，尤其是当程序中需要创建大量生存期很短暂的线程时，更应该考虑使用线程池。

线程池在系统启动时即创建大量空闲的线程，程序只要将一个函数提交给线程池，线程池就会启动一个空闲的线程来执行它。当该函数执行结束后，该线程并不会死亡，而是再次返回到线程池中变成空闲状态，等待执行下一个函数。

此外，使用线程池可以有效地控制系统中并发线程的数量。当系统中包含有大量的并发线程时，会导致系统性能急剧下降，甚至导致 [Python](#) 解释器崩溃，而线程池的最大线程数参数可以控制系统中并发线程的数量不超过此数。

## 线程池的使用

线程池的基类是 `concurrent.futures` 模块中的 `Executor`，`Executor` 提供了两个子类，即 `ThreadPoolExecutor` 和 `ProcessPoolExecutor`，其中 `ThreadPoolExecutor` 用于创建线程池，而 `ProcessPoolExecutor` 用于创建进程池。

如果使用线程池/进程池来管理并发编程，那么只要将相应的 `task` 函数提交给线程池/进程池，剩下的事情就由线程池/进程池来搞定。

`Executor` 提供了如下常用方法：

- `submit(fn, *args, **kwargs)`：将 `fn` 函数提交给线程池。`*args` 代表传给 `fn` 函数的参数，`*kwargs` 代表以关键字参数的形式为 `fn` 函数传入参数。
- `map(func, *iterables, timeout=None, chunksize=1)`：该函数类似于全局函数 `map(func, *iterables)`，只是该函数将会启动多个线程，以异步方式立即对 `iterables` 执行 `map` 处理。
- `shutdown(wait=True)`：关闭线程池。

程序将 `task` 函数提交（`submit`）给线程池后，`submit` 方法会返回一个 `Future` 对象，`Future` 类主要用于获取线程任务函数的返回值。由于线程任务会在新线程中以异步方式执行，因此，线程执行的函数相当于一个“将来完成”的任务，所以 Python 使用 `Future` 来代表。

实际上，在 [Java](#) 的多线程编程中同样有 `Future`，此处的 `Future` 与 `Java` 的 `Future` 大同小异。

`Future` 提供了如下方法：

- `cancel()`：取消该 `Future` 代表的线程任务。如果该任务正在执行，不可取消，则该方法返回 `False`；否则，程序会取消该任务，并返回 `True`。
- `cancelled()`：返回 `Future` 代表的线程任务是否被成功取消。
- `running()`：如果该 `Future` 代表的线程任务正在执行、不可被取消，该方法返回 `True`。
- `done()`：如果该 `Future` 代表的线程任务被成功取消或执行完成，则该方法返回 `True`。
- `result(timeout=None)`：获取该 `Future` 代表的线程任务最后返回的结果。如果 `Future` 代表的线程任务还未完成，该方法将会阻塞当前线程，其中 `timeout` 参数指定最多阻塞多少秒。
- `exception(timeout=None)`：获取该 `Future` 代表的线程任务所引发的异常。如果该任务成功完成，没有异常，则该方法返回 `None`。
- `add_done_callback(fn)`：为该 `Future` 代表的线程任务注册一个“回调函数”，当该任务成功完成时，程序会自动触发该 `fn` 函数。

在用完一个线程池后，应该调用该线程池的 `shutdown()` 方法，该方法将启动线程池的关闭序列。调用 `shutdown()` 方法后的线程池不再接收新任务，但会将以前所有的已提交任务执行完成。当线程池中的所有任务都执行完成后，该线程池中的所有线程都会死亡。

使用线程池来执行线程任务的步骤如下：

1. 调用 `ThreadPoolExecutor` 类的构造器创建一个线程池。



2. 定义一个普通函数作为线程任务。
3. 调用 `ThreadPoolExecutor` 对象的 `submit()` 方法来提交线程任务。
4. 当不想提交任何任务时，调用 `ThreadPoolExecutor` 对象的 `shutdown()` 方法来关闭线程池。

下面程序示范了如何使用线程池来执行线程任务：

```
1.  from concurrent.futures import ThreadPoolExecutor
2.
3.  import threading
4.
5.  # 定义一个准备作为线程任务的函数
6.  def action(max):
7.
8.      my_sum = 0
9.
10.     for i in range(max):
11.
12.         print(threading.current_thread().name + ' ' + str(i))
13.
14.         my_sum += i
15.
16.     return my_sum
17.
18. # 创建一个包含 2 条线程的线程池
19. pool = ThreadPoolExecutor(max_workers=2)
20.
21. # 向线程池提交一个 task, 50 会作为 action() 函数的参数
22. future1 = pool.submit(action, 50)
23.
24. # 向线程池再提交一个 task, 100 会作为 action() 函数的参数
25. future2 = pool.submit(action, 100)
26.
27. # 判断 future1 代表的任务是否结束
28. print(future1.done())
29.
30. time.sleep(3)
31.
32. # 判断 future2 代表的任务是否结束
33. print(future2.done())
34.
35. # 查看 future1 代表的任务返回的结果
36. print(future1.result())
37.
38. # 查看 future2 代表的任务返回的结果
39. print(future2.result())
40.
41. # 关闭线程池
42. pool.shutdown()
```

上面程序中，第 13 行代码创建了一个包含两个线程的线程池，接下来的两行代码只要将 `action()` 函数提交（`submit`）给线程池，该线程池就会负责启动线程来执行 `action()` 函数。这种启动线程的方法既优雅，又具有更高的效率。

当程序把 `action()` 函数提交给线程池时，`submit()` 方法会返回该任务所对应的 `Future` 对象，程序立即判断 `future1` 的 `done()` 方法，

该方法将会返回 False (表明此时该任务还未完成)。接下来主程序暂停 3 秒, 然后判断 future2 的 done() 方法, 如果此时该任务已经完成, 那么该方法将会返回 True。

程序最后通过 Future 的 result() 方法来获取两个异步任务返回的结果。

读者可以自己运行此代码查看运行结果, 这里不再演示。

当程序使用 Future 的 result() 方法来获取结果时, 该方法会阻塞当前线程, 如果没有指定 timeout 参数, 当前线程将一直处于阻塞状态, 直到 Future 代表的任务返回。

## 获取执行结果

前面程序调用了 Future 的 result() 方法来获取线程任务的返回值, 但该方法会阻塞当前主线程, 只有等到线程任务完成后, result() 方法的阻塞才会被解除。

如果程序不希望直接调用 result() 方法阻塞线程, 则可通过 Future 的 add\_done\_callback() 方法来添加回调函数, 该回调函数形如 fn(future)。当线程任务完成后, 程序会自动触发该回调函数, 并将对应的 Future 对象作为参数传给该回调函数。

下面程序使用 add\_done\_callback() 方法来获取线程任务的返回值:

```
1.  from concurrent.futures import ThreadPoolExecutor
2.  import threading
3.  import time
4.
5.  # 定义一个准备作为线程任务的函数
6.  def action(max):
7.
8.      my_sum = 0
9.
10.     for i in range(max):
11.
12.         print(threading.current_thread().name + ' ' + str(i))
13.
14.         my_sum += i
15.
16.     return my_sum
17.
18. # 创建一个包含 2 条线程的线程池
19. with ThreadPoolExecutor(max_workers=2) as pool:
20.
21.     # 向线程池提交一个 task, 50 会作为 action() 函数的参数
22.
23.     future1 = pool.submit(action, 50)
24.
25.     # 向线程池再提交一个 task, 100 会作为 action() 函数的参数
26.
27.     future2 = pool.submit(action, 100)
28.
29.     def get_result(future):
30.
31.         print(future.result())
32.
33.     # 为 future1 添加线程完成的回调函数
34.
35.     future1.add_done_callback(get_result)
```

```

22.     # 为 future2 添加线程完成的回调函数
23.     future2.add_done_callback(get_result)
24.     print('-----')

```

上面主程序分别为 future1、future2 添加了同一个回调函数，该回调函数会在线程任务结束时获取其返回值。

主程序的最后一行代码打印了一条横线。由于程序并未直接调用 future1、future2 的 result() 方法，因此主线程不会被阻塞，可以立即看到输出主线程打印出的横线。接下来将会看到两个新线程并发执行，当线程任务执行完成后，get\_result() 函数被触发，输出线程任务的返回值。

另外，由于线程池实现了上下文管理协议 (Context Manage Protocol)，因此，程序可以使用 with 语句来管理线程池，这样即可避免手动关闭线程池，如上面的程序所示。

此外，Executor 还提供了一个 map(func, \*iterables, timeout=None, chunksize=1) 方法，该方法的功能类似于全局函数 map()，区别在于线程池的 map() 方法会为 iterables 的每个元素启动一个线程，以并发方式来执行 func 函数。这种方式相当于启动 len(iterables) 个线程，并收集每个线程的执行结果。

例如，如下程序使用 Executor 的 map() 方法来启动线程，并收集线程任务的返回值：

```

1.  from concurrent.futures import ThreadPoolExecutor
2.  import threading
3.  import time
4.
5.  # 定义一个准备作为线程任务的函数
6.  def action(max):
7.
8.      my_sum = 0
9.
10.     for i in range(max):
11.
12.         print(threading.current_thread().name + ' ' + str(i))
13.
14.         my_sum += i
15.
16.     return my_sum
17.
18. # 创建一个包含 4 条线程的线程池
19. with ThreadPoolExecutor(max_workers=4) as pool:
20.
21.     # 使用线程执行 map 计算
22.
23.     # 后面元组有 3 个元素，因此程序启动 3 条线程来执行 action 函数
24.
25.     results = pool.map(action, (50, 100, 150))
26.
27.     print('-----')
28.
29.     for r in results:
30.
31.         print(r)

```

上面程序使用 map() 方法来启动 3 个线程（该程序的线程池包含 4 个线程，如果继续使用只包含两个线程的线程池，此时将有一个任务处于等待状态，必须等其中一个任务完成，线程空闲出来才会获得执行的机会），map() 方法的返回值将会收集每个线程任务的返回结果。

运行上面程序，同样可以看到 3 个线程并发执行的结果，最后通过 results 可以看到 3 个线程任务的返回结果。

通过上面程序可以看出，使用 `map()` 方法来启动线程，并收集线程的执行结果，不仅具有代码简单的优点，而且虽然程序会以并发方式来执行 `action()` 函数，但最后收集的 `action()` 函数的执行结果，依然与传入参数的结果保持一致。也就是说，上面 `results` 的第一个元素是 `action(50)` 的结果，第二个元素是 `action(100)` 的结果，第三个元素是 `action(150)` 的结果。

## 13. Python threading Local()函数用法：返回线程局部变量

前面讲过，当多线程操作同一公共资源时，如果涉及到修改该资源的操作，为了避免数据不同步可能导致的错误，需要使用互斥锁机制。

其实，除非必须将多线程使用的资源设置为公共资源，[Python](#) threading 模块还提供了一种可彻底避免数据不同步问题的方法，即本节要介绍的 local() 函数。

使用 local() 函数创建的变量，可以被各个线程调用，但和公共资源不同，各个线程在使用 local() 函数创建的变量时，都会在该线程自己的内存空间中拷贝一份。这意味着，local() 函数创建的变量看似全局变量（可以被各个线程调用），但各线程调用的都是该变量的副本（各调用各的，之间并无关系）。

可以这么理解，使用 threading 模块中的 local() 函数，可以为各个线程创建完全属于它们自己的变量（又称线程局部变量）。正是由于各个线程操作的是属于自己的变量，该资源属于各个线程的私有资源，因此可以从根本上杜绝发生数据同步问题。

下面程序示范了 threading local() 函数的用法：

```
1.  import threading
2.
3.  # 创建全局 ThreadLocal 对象:
4.  local = threading.local()
5.
6.  def process():
7.      # 3. 获取当前线程关联的 resource:
8.      res = local.resource
9.      print (res + "http://c.biancheng.net/python/")
10.
11. def process_thread(res):
12.     # 1. 为当前线程绑定 ThreadLocal 的 resource:
13.     local.resource = res
14.     # 2. 线程的调用函数不需要传递 res 了，可以通过全局变量 local 访问
15.     process()
16.
17. t1 = threading.Thread(target=process_thread, args=('t1 线程',))
18. t2 = threading.Thread(target=process_thread, args=('t2 线程',))
19. t1.start()
20. t2.start()
```

程序执行结果为：

```
t1 线程 http://c.biancheng.net/python/
t2 线程 http://c.biancheng.net/python/
```

下面带领读者分析一下此程序：

1) 首先，在全局作用域范围内，调用 `local()` 函数会生成一个 `ThreadLocal` 对象（有关该类型，初学者不用关心），由此生成的 `local` 变量即为公共资源，它可以在程序的任意线程中被调用。

2) 随后，我们创建了 2 个子线程 `t1` 和 `t2`，它们都负责执行 `process_thread()` 函数，该函数中，我们在全局 `local` 变量的基础上，定义了一个 `resource` 变量，该变量即为线程局部变量，即哪个线程调用该函数，都会将 `resource` 变量拷贝一份并存储在自己的存储空间中。因此，`t1` 和 `t2` 线程会各自拥有一份 `resource` 变量的副本。

注意，虽然变量名相同，但是值不同，`t1` 线程传递的 `res` 参数为 "`t1` 线程"，而 `t2` 线程传递的 `res` 参数值为 "`t2` 线程"。

3) 但各个线程拷贝 `resource` 变量完成后，随即执行 `process()` 函数，在该函数中，我们可以使用 `ThreadLocal` 类型的 `local` 全局变量，调用各个线程自己的 `resource` 变量。

需要说明的一点是，该程序中，只用 `ThreadLocal` 类型的对象 `local` 创建了一个 `resource` 变量，根据实际场景需要，我们完全可以创建任意个线程局部变量。

通过分析上面的样例，不难看出，使用 `local()` 函数的好处，至少有以下 2 点：

1. 各个线程操作的都是自己的私有资源，不会涉及到数据同步问题；
2. 由于 `local()` 函数的返回值位于全局作用域，无论在程序什么位置，都可以随时调用，很方便。

## 总结

无论是使用线程局部变量，还是使用互斥锁机制，其根本目的是为了解决多线程访问公共资源时可能发生的数据同步问题。互斥锁机制实现的出发点是，在各线程仍使用公共资源的前提下，想办法控制各个线程对该资源的同时访问；而线程局部变量则另辟蹊径，直接令多线程操作各自的私有资源，从根本上杜绝了同时访问所带来的数据同步问题。

需要说明的一点是，线程局部变量的解决方案，并不能完全替代互斥锁同步机制。同步机制是为了同步多个线程对公共资源的并发访问，是多个线程之间进行通信的有效方式；而线程局部变量则从根本上避免了多个线程之间对共享资源（变量）的竞争。

那么，这两种解决方案该如何选择呢？简单来说，如果多线程之间需要共享资源（如多人操作同一银行账户的例子），就使用互斥锁同步机制；反之，如果仅是为了解决多线程之间的共享资源冲突，则推荐使用线程局部变量。

## 14. Python Timer 定时器：控制函数在特定时间执行

Thread 类有一个 Timer 子类，该子类可用于控制指定函数在特定时间内执行一次。例如如下程序：

```
1. from threading import Timer
2.
3. def hello():
4.     print("hello, world")
5. # 指定 10 秒后执行 hello 函数
6. t = Timer(10.0, hello)
7. t.start()
```

上面程序使用 Timer 控制 10s 后执行 hello 函数。

需要说明的是，Timer 只能控制函数在指定时间内执行一次，如果要使用 Timer 控制函数多次重复执行，则需要再执行下一次调度。

如果程序想取消 Timer 的调度，则可调用 Timer 对象的 cancel() 函数。例如，如下程序每 1s 输出一当前时间：

```
1. from threading import Timer
2. import time
3.
4. # 定义总共输出几次的计数器
5. count = 0
6. def print_time():
7.     print("当前时间: %s" % time.ctime())
8.     global t, count
9.     count += 1
10.    # 如果 count 小于 10，开始下一次调度
11.    if count < 10:
12.        t = Timer(1, print_time)
13.        t.start()
14. # 指定 1 秒后执行 print_time 函数
15. t = Timer(1, print_time)
16. t.start()
```

上面程序开始运行后，程序控制 1s 后执行 print\_time() 函数。print\_time() 函数中的代码会进行判断，如果 count 小于 10，程序再次使用 Timer 调度 1s 后执行 print\_time() 函数，这样就可以控制 print\_time() 函数多次重复执行。

在上面程序中，由于只有当 count 小于 10 时才会使用 Timer 调度 1s 后执行 print\_time() 函数，因此该函数只会重复执行 10 次。

# 15. Python schedule 任务调度及其用法

使用 Timer 定时器有一个弊端，即只能控制线程在指定时间内执行一次任务，如果想实现每隔一段时间就执行一次，需要借助循环结构。

实际上，Python 还提供有一个更强大的、可用来定义执行任务调度的 sched 模块，该模块中含有一个 scheduler 类，可用来执行更复杂的任务调度。

scheduler 类常用的构造方法如下：

```
scheduler(timefunc=time.monotonic, delayfunc=time.sleep)
```

可以向该构造方法中传入 2 个参数（当然也可以不提供，因为都有默认值），分别表示的含义如下：

- timefunc：指定生成时间戳的函数，默认使用 time.monotonic 来生成时间戳；
- delayfunc：在未达到指定时间前，通过该参数可以指定阻塞任务执行的函数，默认采用 time.sleep() 函数来阻塞程序。

另外，scheduler 类中还提供有一些方法，表 1 罗列了常用的一些。

表 1 scheduler 类常用方法	
方法格式	功能
scheduler.enter(delay, priority, action, argument=(), kwargs={})	在 time 规定的时间后，执行 action 参数指定的函数，其中 argument 和 kwargs 负责为 action 指定的函数传参，priority 参数执行要执行任务的等级，当同一时间点有多个任务需要执行时，等级越高（priority 值越小）的任务会优先执行。该函数会返回一个 event，可用来取消该任务。
scheduler.cancel(event)	取消 event 任务。注意，如果 event 参数执行的任务不存在，则会引发 ValueError 错误。
scheduler.run(blocking=True)	运行所有需要调度的任务。如果调用该方法的 blocking 参数为 True，该方法将会阻塞线程，直到所有被调度的任务都执行完成。

下面程序示范了 scheduler 类的用法。

```
1. import threading
2. from sched import scheduler
3.
4. def action(arg):
5.     print(arg)
6.
7. #定义线程要调用的方法，*add 可接收多个以非关键字方式传入的参数
8. def thread_action(*add):
9.     #创建任务调度对象
10.    sche = scheduler()
11.    #定义优先级
12.    i = 3
13.    for arc in add:
14.        # 指定 1 秒后执行 action 函数
```



```
15.         sche.enter(1, i, action, argument=(arc,))
16.         i = i - 1
17.         #执行所有调度的任务
18.         sche.run()
19.
20.         #定义为线程方法传入的参数
21.         my_tuple = ("http://c.biancheng.net/python/", \
22.                     "http://c.biancheng.net/shell/", \
23.                     "http://c.biancheng.net/java/")
24.         #创建线程
25.         thread = threading.Thread(target = thread_action, args =my_tuple)
26.         #启动线程
27.         thread.start()
```

程序执行结果为：

```
http://c.biancheng.net/java/
http://c.biancheng.net/shell/
http://c.biancheng.net/python/
```

注意，以上输出结果是在执行程序 1 秒后逐个输出的。

上面程序中，创建了 thread 子线程去执行 thread\_action() 函数，在该函数中使用 scheduler 类调度了 3 个任务，这 3 个任务都指定的是 1 秒后执行，其优先级分别为 3、2、1。

由于是在同一时间执行这 3 个任务，因此优先级的设定决定了谁先执行、谁后执行。显然优先级为 1 的任务优先执行，优先级为 3 的最后执行。因此上面程序执行结果中字符串的输出顺序恰好和 my\_tuple 元组中的顺序是相反的。

## 16. Python os.fork()方法：创建新进程

前面章节一直在介绍如何使用多线程实现并发编程，其实 Python 还支持多进程编程。

要知道，每个 Python 程序在执行时，系统都会生成一个新的进程，该进程又称**父进程**（或**主进程**）。在此基础上，Python os 模块还提供有 fork() 函数，该函数可以在当前程序中再创建出一个进程（又称**子进程**）。

也就是说，程序中通过引入 os 模块，并调用其提供的 fork() 函数，程序中会拥有 2 个进程，其中父进程负责执行整个程序代码，而通过 fork() 函数创建出的子进程，会从创建位置开始，执行后续所有的程序（包含创建子进程的代码）。

**注意，os.fork() 函数在 Windows 系统上无效，只在 UNIX 及类 UNIX 系统上（包括 UNIX、Linux 和 Mac OS X）效。**

fork() 方法的语法格式为：

```
pid = os.fork()
```

其中，pid 作为函数的返回值，主进程和子进程都会执行该语句，但主进程执行 fork() 函数得到的 pid 值为非 0 值（其实是子进程的进程 ID），而子进程执行该语句得到的 pid 值为 0。因此，pid 常常作为区分父进程和子进程的标志。

在大多数操作系统中，都会为执行的进程配备唯一的 ID 号，os 模块提供了 getpid() 和 getppid() 函数，可分别用来获取当前进程的 ID 号和父进程的 ID 号。

下面程序演示了 os.fork() 方法的具体使用：

```
1. import os
2. print('父进程 ID =', os.getpid())
3. # 创建一个子进程，下面代码会被两个进程执行
4. pid = os.fork()
5. print('当前进程 ID =', os.getpid(), " pid=", pid)
6. #根据 pid 值，分别为子进程和父进程布置任务
7. if pid == 0:
8.     print('子进程, ID=', os.getpid(), " 父进程 ID=", os.getppid())
9. else:
10.    print('父进程, ID=', os.getpid(), " pid=", pid)
```

程序输出结果为：

```
父进程 ID = 2884
当前进程 ID = 2884 pid= 2885
父进程, ID= 2884 pid= 2885
当前进程 ID = 2885 pid= 0
子进程, ID= 2885 父进程 ID= 2884
```

从输出结果可以看到，当前程序在执行时，系统生成了进程号为 2884 的进程，该进程负责执行当前程序中的所有代码。与此同时，程序第 4 行创建了进程号为 2885 的子进程，该进程将执行第 4 行开始（包括第 4 行）后续的所有代码。

注意，程序第 7 行代码的 if 判断语句，通过判断 pid 值是否为 0，分别为父进程和 fork() 函数创建的子进程布置了不同的执行任务，即子进程负责执行 if 代码块，而父进程则负责执行 else 代码块。

# 17. Python Process 创建进程（2 种方法）详解

前面介绍了使用 `os.fork()` 函数实现多进程编程，该方法最明显的缺陷就是不适用于 Windows 系统。本节将介绍一种支持 [Python](#) 在 Windows 平台上创建新进程的方法。

Python multiprocessing 模块提供了 Process 类，该类可用在 Windows 平台上创建新进程。和使用 Thread 类创建多线程方法类似，使用 Process 类创建多进程也有以下 2 种方式：

- 1. 直接创建 Process 类的实例对象，由此就可以创建一个新的进程；
- 2. 通过继承 Process 类的子类，创建实例对象，也可以创建新的进程。注意，继承 Process 类的子类需重写父类的 `run()` 方法。

不仅如此，Process 类中也提供了一些常用的属性和方法，如表 1 所示。

表 1 Python Process 类常用属性和方法	
属性名或方法名	功能
<code>run()</code>	第 2 种创建进程的方式需要用到，继承类中需要对方法进行重写，该方法中包含的是新进程要执行的代码。
<code>start()</code>	和启动子线程一样，新创建的进程也需要手动启动，该方法的功能就是启动新创建的线程。
<code>join([timeout])</code>	和 thread 类 <code>join()</code> 方法的用法类似，其功能是在多进程执行过程，其他进程必须等到调用 <code>join()</code> 方法的进程执行完毕（或者执行规定的 <code>timeout</code> 时间）后，才能继续执行；
<code>is_alive()</code>	判断当前进程是否还活着。
<code>terminate()</code>	中断该进程。
<code>name</code> 属性	可以为该进程重命名，也可以获得该进程的名称。
<code>daemon</code>	和守护线程类似，通过设置该属性为 <code>True</code> ，可将新建进程设置为“守护进程”。
<code>pid</code>	返回进程的 ID 号。大多数操作系统都会为每个进程配备唯一的 ID 号。

接下来将——对创建进程的 2 种方法做详细的讲解。

## 通过 Process 类创建进程

和使用 thread 类创建子线程的方式非常类似，使用 Process 类创建实例化对象，其本质是调用该类的构造方法创建新进程。Process 类的构造方法格式如下：

```
def __init__(self,group=None,target=None,name=None,args=(),kwargs={})
```

其中，各个参数的含义为：

- `group`：该参数未进行实现，不需要传参；
- `target`：为新建进程指定执行任务，也就是指定一个函数；
- `name`：为新建进程设置名称；
- `args`：为 `target` 参数指定的参数传递非关键字参数；
- `kwargs`：为 `target` 参数指定的参数传递关键字参数。

下面程序演示了如何用 Process 类创建新进程。

```

1.  from multiprocessing import Process
2.  import os
3.  print("当前进程 ID: ", os.getpid())
4.
5.  # 定义一个函数，准备作为新进程的 target 参数
6.  def action(name, *add):
7.      print(name)
8.      for arc in add:
9.          print("%s --当前进程%d" % (arc, os.getpid()))
10. if __name__ == '__main__':
11.     # 定义为进程方法传入的参数
12.     my_tuple = ("http://c.biancheng.net/python/", \
13.                 "http://c.biancheng.net/shell/", \
14.                 "http://c.biancheng.net/java/")
15.     # 创建子进程，执行 action() 函数
16.     my_process = Process(target = action, args = ("my_process 进程", *my_tuple))
17.     # 启动子进程
18.     my_process.start()
19.     # 主进程执行该函数
20.     action("主进程", *my_tuple)

```

程序执行结果为：

```

当前进程 ID：12980
主进程
http://c.biancheng.net/python/ --当前进程 12980
http://c.biancheng.net/shell/ --当前进程 12980
http://c.biancheng.net/java/ --当前进程 12980
当前进程 ID：12860
my_process 进程
http://c.biancheng.net/python/ --当前进程 12860
http://c.biancheng.net/shell/ --当前进程 12860
http://c.biancheng.net/java/ --当前进程 12860

```

需要说明的是，通过 `multiprocessing.Process` 来创建并启动进程时，程序必须先判断 `if __name__ == '__main__':`，否则运行该程序会引发异常。

此程序中有 2 个进程，分别为主进程和我们创建的新进程，主进程会执行整个程序，而子进程不会执行 `if __name__ == '__main__'` 中包含的程序，而是先执行此判断语句之外的所有可执行程序，然后再执行我们分配让它的任务（也就是通过 `target` 参数指定的函数）。

## 通过 Process 继承类创建进程

和使用 `thread` 子类创建线程的方式类似，除了直接使用 `Process` 类创建进程，还可以通过创建 `Process` 的子类来创建进程。

需要注意的是，在创建 Process 的子类时，需在子类内容重写 run() 方法。实际上，该方法所起到的作用，就如同第一种创建方式中 target 参数执行的函数。

另外，通过 Process 子类创建进程，和使用 Process 类一样，先创建该类的实例对象，然后调用 start() 方法启动该进程。下面程序演示如何通过 Process 子类创建一个进程。

```
1.  from multiprocessing import Process
2.  import os
3.  print("当前进程 ID: ", os.getpid())
4.
5.  # 定义一个函数，供主进程调用
6.  def action(name,*add):
7.      print(name)
8.      for arc in add:
9.          print("%s --当前进程%d" % (arc, os.getpid()))
10.
11. #自定义一个进程类
12. class My_Process(Process):
13.     def __init__(self, name,*add):
14.         super().__init__()
15.         self.name = name
16.         self.add = add
17.     def run(self):
18.         print(self.name)
19.         for arc in self.add:
20.             print("%s --当前进程%d" % (arc, os.getpid()))
21.
22. if __name__ == '__main__':
23.     #定义为进程方法传入的参数
24.     my_tuple = ("http://c.biancheng.net/python/", \
25.                 "http://c.biancheng.net/shell/", \
26.                 "http://c.biancheng.net/java/")
27.
28.     my_process = My_Process("my_process 进程",*my_tuple)
29.     #启动子进程
30.     my_process.start()
31.     #主进程执行该函数
32.     action("主进程",*my_tuple)
```

程序执行结果为：

```
当前进程 ID：22240
主进程
http://c.biancheng.net/python/ --当前进程 22240
http://c.biancheng.net/shell/ --当前进程 22240
http://c.biancheng.net/java/ --当前进程 22240
当前进程 ID：18848
my_process 进程
http://c.biancheng.net/python/ --当前进程 18848
http://c.biancheng.net/shell/ --当前进程 18848
http://c.biancheng.net/java/ --当前进程 18848
```

显然，该程序的运行结果与上一个程序的运行结果大致相同，它们只是创建进程的方式略有不同而已。

推荐读者使用第一种方式来创建进程，因为这种方式不仅编程简单，而且进程直接包装 `target` 函数，具有更清晰的逻辑结构。

# 18. Python 设置进程启动的 3 种方式

前面章节中，已经详解介绍了 2 种创建进程的方法，即分别使用 `os.fork()` 和 `Process` 类来创建进程。其中：

- 使用 `os.fork()` 函数创建的子进程，会从创建位置处开始，执行后续所有的程序，主进程如何执行，则子进程就如何执行；
- 而使用 `Process` 类创建的进程，则仅会执行 `if "__name__"=="__main__"` 之外的可执行代码以及该类构造方法中 `target` 参数指定的函数（使用 `Process` 子类创建的进程，只能执行重写的 `run()` 方法）。

实际上，`Python` 创建的子进程执行的内容，和启动该进程的方式有关。而根据不同的平台，启动进程的方式大致可分为以下 3 种：

1. `spawn`：使用此方式启动的进程，只会执行和 `target` 参数或者 `run()` 方法相关的代码。`Windows` 平台只能使用此方法，事实上该平台默认使用的也是该启动方式。相比其他两种方式，此方式启动进程的效率最低。
2. `fork`：使用此方式启动的进程，基本等同于主进程（即主进程拥有的资源，该子进程全都有）。因此，该子进程会从创建位置起，和主进程一样执行程序中的代码。注意，此启动方式仅适用于 `UNIX` 平台，`os.fork()` 创建的进程就是采用此方式启动的。
3. `forkserver`：使用此方式，程序将会启动一个服务器进程。即当程序每次请求启动新进程时，父进程都会连接到该服务器进程，请求由服务器进程来创建新进程。通过这种方式启动的进程不需要从父进程继承资源。注意，此启动方式只在 `UNIX` 平台上有效。

总的来说，使用类 `UNIX` 平台，启动进程的方式有以上 3 种，而使用 `Windows` 平台，只能选用 `spawn` 方式（默认即可）。

在了解以上 3 种进程启动方式的基础上，我们还需要知道手动设置进程启动方式的方法，大致有以下 2 种。

1) `Python multiprocessing` 模块提供了一个 `set_start_method()` 函数，该函数可用于设置启动进程的方式。需要注意的是，该函数的调用位置，必须位于所有与多进程有关的代码之前。

例如，下面代码演示了如何显式设置进程的启动方式：

```
1. import multiprocessing
2. import os
3. print("当前进程 ID: ",os.getpid())
4.
5. # 定义一个函数，准备作为新进程的 target 参数
6. def action(name,*add):
7.     print(name)
8.     for arc in add:
9.         print("%s --当前进程%d" % (arc,os.getpid()))
10. if __name__=='__main__':
11.     #定义为进程方法传入的参数
12.     my_tuple = ("http://c.biancheng.net/python/",\
13.                 "http://c.biancheng.net/shell/",\
14.                 "http://c.biancheng.net/java/")
15.     #设置进程启动方式
16.     multiprocessing.set_start_method('spawn')
17.
18.     #创建子进程，执行 action() 函数
```

```

19.     my_process = multiprocessing.Process(target = action, args = ("my_process 进程",*my_tuple))
20.     #启动子进程
21.     my_process.start()

```

程序执行结果为：

```

当前进程 ID：24500
当前进程 ID：17300
my_process 进程
http://c.biancheng.net/python/ --当前进程 17300
http://c.biancheng.net/shell/ --当前进程 17300
http://c.biancheng.net/java/ --当前进程 17300

```

注意，由于此程序中进程的启动方式为 spawn，因此该程序可以在任意（Windows 和类 UNIX 上都可以）平台上执行。

2) 除此之外，还可以使用 multiprocessing 模块提供的 get\_context() 函数来设置进程启动的方法，调用该函数时可传入 "spawn"、"fork"、"forkserver" 作为参数，用来指定进程启动的方式。

需要注意的一点是，前面在创建进程是，使用的 multiprocessing.Process() 这种形式，而在使用 get\_context() 函数设置启动进程方式时，需用该函数的返回值，代替 multiprocessing 模块调用 Process()。

例如，下面程序演示了如何使用 get\_context() 函数设置进程启动：

```

1.  import multiprocessing
2.  import os
3.  print("当前进程 ID: ",os.getpid())
4.
5.  # 定义一个函数，准备作为新进程的 target 参数
6.  def action(name,*add):
7.      print(name)
8.      for arc in add:
9.          print("%s --当前进程%d" % (arc,os.getpid()))
10. if __name__ == '__main__':
11.     #定义为进程方法传入的参数
12.     my_tuple = ("http://c.biancheng.net/python/",\
13.                 "http://c.biancheng.net/shell/",\
14.                 "http://c.biancheng.net/java/")
15.     #设置使用 fork 方式启动进程
16.     ctx = multiprocessing.get_context('spawn')
17.
18.     #用 ctx 代替 multiprocessing 模块创建子进程，执行 action() 函数
19.     my_process = ctx.Process(target = action, args = ("my_process 进程",*my_tuple))
20.     #启动子进程

```



```
21.         my_process.start()
```

程序执行结果为：

```
当前进程 ID : 18632
当前进程 ID : 16700
my_process 进程
http://c.biancheng.net/python/ --当前进程 16700
http://c.biancheng.net/shell/ --当前进程 16700
http://c.biancheng.net/java/ --当前进程 16700
```

以上仅演示了在 Windows 平台上设置进程启动方式的效果，有兴趣的读者可自行尝试选择类 UNIX 平台测试其他启动进程的方式。

## 19.多进程编程和多线程编程优缺点

多进程编程和多线程编程，都可以使用并行机制来提升系统的运行效率。二者的区别在于运行时所占的内存分布不同，多线程是共用一套内存的代码块区间；而多进程是各用一套独立的内存区间。

多进程的优点是稳定性好，一个子进程崩溃了，不会影响主进程以及其余进程。基于这个特性，常常会用多进程来实现守护服务器的功能。

多进程编程也有不足，即创建进程的代价非常大，因为操作系统要给每个进程分配固定的资源，并且操作系统对进程的总数会有一定的限制，若进程过多，操作系统调度都会存在问题，会造成假死状态。

多线程编程的优点是效率较高一些，适用于批处理任务等功能；不足之处在于，任何一个线程崩溃都可能造成整个进程的崩溃，因为它们共享了进程的内存资源池。

既然多线程编程和多进程编程各有优缺点，因此它们分别适用于不同的场景。比如说，对于计算密集型的任务，多进程效率会更高一下；而对于 IO 密集型的任务（比如文件操作，网络爬虫），采用多线程编程效率更高。为什么是这样呢？

其实也不难理解。对于 IO 密集型操作，大部分消耗时间其实是等待时间，在等待时间中，Python 会释放 GIL 供新的线程使用，实现了线程间的切换；相反对于 CPU 密集型代码，2 个 CPU 干活肯定比一个 CPU 快很多。

在大型的计算机集群系统中，通常都会将多进程程序分布运行在不同的计算机上协同工作。而每一台计算机上的进程内部，又会由多个线程来并行工作。

注意，对于任务数来说，无论是多进程编程或者多线程编程，其进程数或线程数都不能太多：

- 对于多进程编程来说，操作系统在切换任务时，会有一系列的保护现场措施，这要花费相当多的系统资源，若任务过多，则大部分资源都被用做干这些了，结果就是所有任务都做不好；
- 多线程编程也不是线程个数越多效率越高，通过下面的公式可以计算出线程数量最优的一个参考值。

$$\text{最佳线程数量} = \frac{\text{线程等待时间} + \text{线程CPU时间}}{\text{线程CPU时间}} * \text{CPU数量}$$

# 20.Python 使用进程池管理进程

和选用线程池来关系多线程类似，当程序中设置到多进程编程时，Python 提供了更好的管理多个进程的方式，就是使用进程池。

进程池可以提供指定数量的进程给用户使用，即当有新的请求提交到进程池中时，如果池未满，则会创建一个新的进程用来执行该请求；反之，如果池中的进程数已经达到规定最大值，那么该请求就会等待，只要池中有进程空闲下来，该请求就能得到执行。

Python multiprocessing 模块提供了 Pool() 函数，专门用来创建一个进程池，该函数的语法格式如下：

```
multiprocessing.Pool( processes )
```

其中，processes 参数用于指定该进程池中包含的进程数。如果进程是 None，则默认使用 os.cpu\_count() 返回的数字（根据本地的 cpu 个数决定，processes 小于等于本地的 cpu 个数）。

注意，Pool() 函数只是用来创建进程池，而 multiprocessing 模块中表示进程池的类是 multiprocessing.pool.Pool 类。该类中提供了一些和操作进程池相关的方法，如表 1 所示。

表 1 multiprocessing 模块 Pool 类常用方法	
方法名	功能
apply( func[, args[, kwds]] )	将 func 函数提交给进程池处理。其中 args 代表传给 func 的位置参数，kwds 代表传给 func 的关键字参数。该方法会被阻塞直到 func 函数执行完成。
apply_async( func[, args[, kwds[, callback[, error_callback]]]] )	这是 apply() 方法的异步版本，该方法不会被阻塞。其中 callback 指定 func 函数完成后的回调函数，error_callback 指定 func 函数出错后的回调函数。
map( func, iterable[, chunksize] )	类似于 Python 的 map() 全局函数，只不过此处使用新进程对 iterable 的每一个元素执行 func 函数。
map_async( func, iterable[, chunksize[, callback[, error_callback]]] )	这是 map() 方法的异步版本，该方法不会被阻塞。其中 callback 指定 func 函数完成后的回调函数，error_callback 指定 func 函数出错后的回调函数。
imap( func, iterable[, chunksize] )	这是 map() 方法的延迟版本。
imap_unordered( func, iterable[, chunksize] )	功能类似于 imap() 方法，但该方法不能保证所生成的结果（包含多个元素）与原 iterable 中的元素顺序一致。
starmap( func, iterable[, chunksize] )	功能类似于 map() 方法，但该方法要求 iterable 的元素也是 iterable 对象，程序会将每一个元素解包之后作为 func 函数的参数。
close()	关闭进程池。在调用该方法之后，该进程池不能再接收新任务，它会把当前进程池中的所有任务执行完成后再关闭自己。
terminate()	立即中止进程池。
join()	等待所有进程完成。

下面程序演示了进程池的创建和使用。

```
1.  from multiprocessing import Pool
2.
3.  import time
4.
5.  def action(name='http://c.biancheng.net'):
6.      print(name, ' --当前进程: ', os.getpid())
7.      time.sleep(3)
8.  if __name__ == '__main__':
```

```

9.      #创建包含 4 条进程的进程池
10.     pool = Pool(processes=4)
11.     # 将 action 分 3 次提交给进程池
12.     pool.apply_async(action)
13.     pool.apply_async(action, args=('http://c.biancheng.net/python/', ))
14.     pool.apply_async(action, args=('http://c.biancheng.net/java/', ))
15.     pool.apply_async(action, kwds={'name': 'http://c.biancheng.net/shell/'})
16.     pool.close()
17.     pool.join()

```

程序执行结果为：

```

http://c.biancheng.net --当前进程：14396
http://c.biancheng.net/python/ --当前进程：5432
http://c.biancheng.net/java/ --当前进程：11080
http://c.biancheng.net/shell/ --当前进程：10944

```

除此之外，我们可以使用 with 语句来管理进程池，这意味着我们无需手动调用 close() 方法关闭进程池。例如：

```

1.  from multiprocessing import Pool
2.  import time
3.  import os
4.
5.  def action(name='http://c.biancheng.net'):
6.      time.sleep(3)
7.      return (name+' --当前进程: %d'%os.getpid())
8.  if __name__ == '__main__':
9.      #创建包含 4 条进程的进程池
10.
11.     with Pool(processes=4) as pool:
12.         adds = pool.map(action, ('http://c.biancheng.net/python/', 'http://c.biancheng.net/java/',
13.                                  'http://c.biancheng.net/shell/'))
14.         for arc in adds:
15.             print(arc)

```

程序执行结果为：

```

http://c.biancheng.net/python/ --当前进程：24464
http://c.biancheng.net/java/ --当前进程：22900
http://c.biancheng.net/shell/ --当前进程：23324

```

# 21.Python 进程间通信的 2 种实现方法（ Queue 和 Pipe ）

在讲解多线程时，介绍了 3 种实现线程间通信的机制，同样 [Python](#) 也提供了多种实现进程间通信的机制，主要有以下 2 种：

- 1. Python multiprocessing 模块下的 Queue 类，提供了多个进程之间实现通信的诸多方法；
- 2. Pipe，又被称为“管道”，常用于实现 2 个进程之间的通信，这 2 个进程分别位于管道的两端。

接下来将对以上 2 种方式的具体实现做详细的讲解。

## Queue 实现进程间通信

前面讲解了使用 Queue 模块中的 Queue 类实现线程间通信，但要实现进程间通信，需要使用 multiprocessing 模块中的 Queue 类。

简单的理解 Queue 实现进程间通信的方式，就是使用了操作系统给开辟的一个队列空间，各个进程可以把数据放到该队列中，当然也可以从队列中把自己需要的信息取走。

Queue 类提供了诸多实现进程间通信的方法，表 1 罗列了常用的一些方法。

表 1 Python multiprocessing Queue 类常用方法	
方法名	功能
put( obj[,block=True[,timeout=None]])	将 obj 放入队列，其中当 block 参数设为 True 时，一旦队列被写满，则代码就会被阻塞，直到有进程取走数据并腾出空间供 obj 使用。timeout 参数用来设置阻塞的时间，即程序最多在阻塞 timeout 秒之后，如果还是没有空闲空间，则程序会抛出 queue.Full 异常。
put_nowait(obj)	该方法的功能等同于 put(obj, False)。
get([block=True[,timeout=None]])	从队列中取数据并返回，当 block 为 True 且 timeout 为 None 时，该方法会阻塞当前进程，直到队列中有可用的数据。如果 block 设为 False，则进程会直接做取数据的操作，如果取数据失败，则抛出 queue.Empty 异常（这种情形下 timeout 参数将不起作用）。如果手动 timeout 秒数，则当前进程最多被阻塞 timeout 秒，如果到时依旧没有可用的数据取出，则会抛出 queue.Empty 异常。
get_nowait()	该方法的功能等同于 get(False)。
empty()	判断当前队列空间是否为空，如果为空，则该方法返回 True；反之，返回 False。

下面程序演示了如何使用 Queue 类实现多进程之间的通信。

```
1. import multiprocessing
2.
3. def processFun(queue, name):
4.     print(multiprocessing.current_process().pid,"进程放数据：",name)
5.     #将 name 放入队列
6.     queue.put(name)
7. if __name__ == '__main__':
8.     # 创建进程通信的 Queue
9.     queue = multiprocessing.Queue()
```

```
10.     # 创建子进程
11.     process = multiprocessing.Process(target=processFun, args=(queue, "http://c.biancheng.net/python/"))
12.     # 启动子进程
13.     process.start()
14.     #该子进程必须先执行完毕
15.     process.join()
16.     print(multiprocessing.current_process().pid, "取数据：")
17.     print(queue.get())
```

程序执行结果为：

```
27100 进程放数据： http://c.biancheng.net/python/
24188 取数据：
http://c.biancheng.net/python/
```

## Pipe 实现进程间通信

Pipe 直译过来的意思是“管”或“管道”，该种实现多进程编程的方式，和实际生活中的管（管道）是非常类似的。通常情况下，管道有 2 个口，而 Pipe 也常用来实现 2 个进程之间的通信，这 2 个进程分别位于管道的两端，一端用来发送数据，另一端用来接收数据。

使用 Pipe 实现进程通信，首先需要调用 multiprocessing.Pipe() 函数来创建一个管道。该函数的语法格式如下：

```
conn1, conn2 = multiprocessing.Pipe( [duplex=True] )
```

其中，conn1 和 conn2 分别用来接收 Pipe 函数返回的 2 个端口；duplex 参数默认为 True，表示该管道是双向的，即位于 2 个端口的进程既可以发送数据，也可以接受数据，而如果将 duplex 值设为 False，则表示管道是单向的，conn1 只能用来接收数据，而 conn2 只能用来发送数据。

另外值得一提的是，conn1 和 conn2 都属于 PipeConnection 对象，它们还可以调用表 2 所示的这些方法。

表 2 Pipe 对象可调用的方法

方法名	功能
send(obj)	发送一个 obj 给管道的另一端，另一端使用 recv() 方法接收。需要说明的是，该 obj 必须是可序列化的，如果该对象序列化之后超过 32MB，则很可能会引发 ValueError 异常。
recv()	接收另一端通过 send() 方法发送过来的数据。
close()	关闭连接。
poll([timeout])	返回连接中是否还有数据可以读取。
send_bytes(buffer[, offset[, size]])	发送字节数据。如果没有指定 offset、size 参数，则默认发送 buffer 字节串的全部数据；如果指定了 offset 和 size 参数，则只发送 buffer 字节串中从 offset 开始、长度为 size 的字节数据。通过该方法发送的数据，应该使用 recv_bytes() 或 recv_bytes_into 方法接收。
recv_bytes([maxlength])	接收通过 send_bytes() 方法发送的数据，maxlength 指定最多接收的字节数。该方法返回接收到的字节数据。
recv_bytes_into(buffer[, offset])	功能与 recv_bytes() 方法类似，只是该方法将接收到的数据放在 buffer 中。

下面程序演示了如何使用 Pipe 管道实现 2 个进程之间通信：

```
1. import multiprocessing
2.
3. def processFun(conn, name):
4.     print(multiprocessing.current_process().pid, "进程发送数据: ", name)
5.     conn.send(name)
6. if __name__ == '__main__':
7.     #创建管道
8.     conn1, conn2 = multiprocessing.Pipe()
9.     # 创建子进程
10.    process = multiprocessing.Process(target=processFun, args=(conn1, "http://c.biancheng.net/python/"))
11.    # 启动子进程
12.    process.start()
13.    process.join()
14.
15.    print(multiprocessing.current_process().pid, "接收数据: ")
16.    print(conn2.recv())
```

程序执行结果为：

```
28904 进程发送数据： http://c.biancheng.net/python/
28880 接收数据：
http://c.biancheng.net/python/
```

## 22. Python Futures 并发编程详解

无论哪门编程语言，并发编程都是一项很常用很重要的技巧。例如，爬虫就被广泛应用在工业界的各个领域，我们每天在各个网站、各个 App 上获取的新闻信息，很大一部分便是通过并发编程版的爬虫获得。

正确合理地使用并发编程，无疑会给程序带来极大的性能提升。因此，本节就带领大家一起学习 Python 中的 Futures 并发编程。首先，先带领大家从代码的角度来理解并发编程中的 Futures，并进一步来比较其与单线程的性能区别。

假设有这样一个任务，要下载一些网站的内容并打印，如果用单线程的方式，它的代码实现如下所示（为了突出主题，对代码做了简化，忽略了异常处理）：

```
1.  import requests
2.  import time
3.
4.  def download_one(url):
5.      resp = requests.get(url)
6.      print('Read {} from {}'.format(len(resp.content), url))
7.
8.  def download_all(sites):
9.      for site in sites:
10.         download_one(site)
11.
12.  def main():
13.      sites = [
14.          'http://c.biancheng.net',
15.          'http://c.biancheng.net/c',
16.          'http://c.biancheng.net/python'
17.      ]
18.      start_time = time.perf_counter()
19.      download_all(sites)
20.      end_time = time.perf_counter()
21.      print('Download {} sites in {} seconds'.format(len(sites), end_time - start_time))
22.
23.  if __name__ == '__main__':
24.      main()
```

输出结果为：

```
Read 52053 from http://c.biancheng.net
Read 30718 from http://c.biancheng.net/c
Read 34470 from http://c.biancheng.net/python
Download 3 sites in 0.3537296 seconds
```



注意，此程序中，requests 模块需单独安装，可通过执行 `pip install requests` 命令进行安装。  
这种方式应该是最直接也最简单的：

- 先是遍历存储网站的列表；
- 然后对当前网站执行下载操作；
- 等到当前操作完成后，再对下一个网站进行同样的操作，一直到结束。

可以看到，总共耗时约 0.35s。单线程的优点是简单明了，但是明显效率低下，因为上述程序的绝大多数时间都浪费在了 I/O 等待上。程序每次对一个网站执行下载操作，都必须等到前一个网站下载完成后才能开始。如果放在实际生产环境中，我们需要下载的网站数量至少是以万为单位的，不难想象，这种方案根本行不通。

接着再来看多线程版本的代码实现：

```
1. import concurrent.futures
2. import requests
3. import threading
4. import time
5.
6. def download_one(url):
7.     resp = requests.get(url)
8.     print('Read {} from {}'.format(len(resp.content), url))
9.
10. def download_all(sites):
11.     with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
12.         executor.map(download_one, sites)
13.
14. def main():
15.     sites = [
16.         'http://c.biancheng.net',
17.         'http://c.biancheng.net/c',
18.         'http://c.biancheng.net/python'
19.     ]
20.     start_time = time.perf_counter()
21.     download_all(sites)
22.     end_time = time.perf_counter()
23.     print('Download {} sites in {} seconds'.format(len(sites), end_time - start_time))
24.
25. if __name__ == '__main__':
26.     main()
```

运行结果为：

```
Read 52053 from http://c.biancheng.net
Read 30718 from http://c.biancheng.net/c
Read 34470 from http://c.biancheng.net/python

Download 3 sites in 0.1606366 seconds
```

可以看到，总耗时是 0.2s 左右，效率一下子提升了很多。

注意，虽然线程的数量可以自己定义，但是线程数并不是越多越好，因为线程的创建、维护和删除也会有一定的开销，所以如果设置的很大，反而可能会导致速度变慢。我们往往需要根据实际的需求做一些测试，来寻找最优的线程数量。

上面两段代码中，多线程版本和单线程版的主要区别在于如下代码：

```
1. with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
2.     executor.map(download_one, sites)
```

这里创建了一个线程池，总共有 5 个线程可以分配使用。executor.map() 与前面所讲的 Python 内置的 map() 函数类似，表示对 sites 中的每一个元素并发地调用函数 download\_one()。

在 download\_one() 函数中使用的 requests.get() 方法是线程安全的，在多线程的环境下也可以安全使用，不会出现条件竞争（多个线程同时竞争使用同一资源）的情况。

当然，也可以用并行的方式去提高程序运行效率，只需要在 download\_all() 函数中做出下面的变化即可：

```
1. with futures.ThreadPoolExecutor(workers) as executor
2.     #=>
3. with futures.ProcessPoolExecutor() as executor:
```

这部分代码中，函数 ProcessPoolExecutor() 表示创建进程池，使用多个进程并行的执行程序。不过，这里通常省略参数 workers，因为系统会自动返回 CPU 的数量作为可以调用的进程数。

但是，并行的方式一般用在 CPU heavy 的场景中，因为对于 I/O heavy 的操作，多数时间都会用于等待，相比于多线程，使用多进程并不会提升效率。反而很多时候，因为 CPU 数量的限制，会导致其执行效率不如多线程版本。

## 什么是 Futures？

Python Futures 模块，位于 concurrent.futures 和 asyncio 中，它们都表示带有延迟的操作。Futures 会将处于等待状态的操作包裹起来放到队列中，这些操作的状态随时可以查询，当然它们的结果（或是异常）也能够操作完成后被获取。

通常来说，用户不用考虑如何去创建 Futures，这些 Futures 底层都会帮我们处理好，唯一要做的只是去设定这些 Futures 的执行。比如，Futures 中的 Executor 类，当执行 executor.submit(func) 时，它便会安排里面的 func() 函数执行，并返回创建好的 future 实例，以便之后查询调用。

这里再介绍一些常用的函数。比如 Futures 中的方法 done()，表示相对应的操作是否完成，返回 True 表示完成；返回 False 表示没有完成。不过要注意的是，done() 是非阻塞的，会立即返回结果。相对应的 add\_done\_callback(fn)，则表示 Futures 完成后，相对应的参数函数 fn 会被通知并执行调用。

Futures 中还有一个重要的函数 result()，它表示当 future 完成后，返回其对应的结果或异常。而 as\_completed(fs)，则是针对给定的 future 迭代器 fs，在其完成后返回完成后的迭代器。

所以，上述例子也可以写成下面的形式：

```
1. import concurrent.futures
2. import requests
3. import time
4.
5. def download_one(url):
6.     resp = requests.get(url)
7.     print('Read {} from {}'.format(len(resp.content), url))
8.
9. def download_all(sites):
10.     with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
11.         to_do = []
12.         for site in sites:
13.             future = executor.submit(download_one, site)
14.             to_do.append(future)
15.
16.         for future in concurrent.futures.as_completed(to_do):
17.             future.result()
18. def main():
19.     sites = [
20.         'http://c.biancheng.net',
21.         'http://c.biancheng.net/c',
22.         'http://c.biancheng.net/python'
23.     ]
24.     start_time = time.perf_counter()
25.     download_all(sites)
26.     end_time = time.perf_counter()
27.     print('Download {} sites in {} seconds'.format(len(sites), end_time - start_time))
28.
29. if __name__ == '__main__':
30.     main()
```

运行结果为：

```
Read 52053 from http://c.biancheng.net
Read 34470 from http://c.biancheng.net/python
Read 30718 from http://c.biancheng.net/c
```

Download 3 sites in 0.2275894 seconds

此程序中，首先调用 `executor.submit()`，将下载每一个网站的内容都放进 `future` 队列 `to_do` 等待执行。然后是 `as_completed()` 函数在 `future` 完成后便输出结果。

不过，这里要注意，`future` 列表中每个 `future` 完成的顺序和它在列表中的顺序并不完全一致。到底哪个先完成、哪个后完成，取决于系统的调度和每个 `future` 的执行时间。

## 23. Python Asyncio 并发编程详解

本节继续学习 Python 并发编程的另一种实现方式，也就是 Asyncio 并发编程。

我们知道，使用多线程和普通的单线程相比，其运行效率会有极大的提高。但不得不说，多线程虽然有诸多优势，也存在一定的局限性：

- 多线程运行过程中容易被打断，还可能出现多个线程同时竞争同一资源的情况；
- 多线程切换本身存在一定的损耗，线程数不能无限增加，因此如果 I/O 操作非常频繁，多线程很有可能满足不了高效率、高质量的需求。

为了解决这些问题，Asyncio 并发编程应运而生。

在详细介绍 Asyncio 之前，要先搞清楚什么是同步，什么是异步。所谓同步，是指操作一个接一个地执行，下一个操作必须等上一个操作执行完成之后才能开始执行；而异步是指不同操作间可以相互交替执行，如果其中地某个操作被堵塞，程序并不会等待，而是会找出可执行的操作继续执行。

为了更好地区分同步和异步，这里举个例子，假设公司要我们做一份报表，并以邮件的方式提交，则分别以同步和异步的方式完成的过程如下：

- 如果按照同步的方式，应先向软件中输入各项数据，接下来等报表生成，再写邮件提交；
- 如果按照异步的方式，向软件中输出各项数据后，会先写邮件，等待报表生成后，暂停写邮件的工作去查看生成的报表，确认无误后在写邮件直到发送完毕。

了解了同步和异步（以及它们之间的区别）之后，接下来正式开始介绍 Asyncio。

### 什么是 Asyncio

事实上，Asyncio 和其他 Python 程序一样，是单线程的，它只有一个主线程，但可以进行多个不同的任务。这里的任务，指的就是特殊的 future 对象，我们可以把它类比成多线程版本里的多个线程。

这些不同的任务，被一个叫做事件循环（Event Loop）的对象所控制。所谓事件循环，是指主线程每次将执行序列中的任务清空后，就去事件队列中检查是否有等待执行的任务，如果有则每次取出一个推到执行序列中执行，这个过程是循环往复的。

为了简化讲解这个问题，可以假设任务只有两个状态：，分别是预备状态和等待状态：

- 预备状态是指任务目前空闲，但随时待命准备运行；
- 等待状态是指任务已经运行，但正在等待外部的操作完成，比如 I/O 操作。

在这种情况下，事件循环会维护两个任务列表，分别对应这两种状态，并且选取预备状态的一个任务（具体选取哪个任务，和其等待的时间长短、占用的资源等等相关）使其运行，一直到这个任务把控制权交还给事件循环为止。

当任务把控制权交还给事件循环对象时，它会根据其是否完成把任务放到预备或等待状态的列表，然后遍历等待状态列表的任务，查看他们是否完成：如果完成，则将其放到预备状态的列表；反之，则继续放在等待状态的列表。而原先在预备状态列表的任务位置仍旧不变，因为它们还未运行。

这样，当所有任务被重新放置在合适的列表后，新一轮的循环又开始了，事件循环对象继续从预备状态的列表中选取一个任务使其执行...如此周而复始，直到所有任务完成。

值得一提的是，对于 Asyncio 来说，它的任务在运行时不会被外部的一些因素打断，因此 Asyncio 内的操作不会出现竞争资源（多个线程同时使用同一资源）的情况，也就不需要担心线程安全的问题了。

## 如何使用 Asyncio

讲完了 Asyncio 的原理，下面结合具体的代码来看一下它的用法。还是以下载网站内容为例，用 Asyncio 的实现代码（省略了异常处理的一些操作）如下：

```
1.  import asyncio
2.  import aiohttp
3.  import time
4.
5.  async def download_one(url):
6.      async with aiohttp.ClientSession() as session:
7.          async with session.get(url) as resp:
8.              print('Read {} from {}'.format(resp.content_length, url))
9.
10. async def download_all(sites):
11.     tasks = [asyncio.ensure_future(download_one(site)) for site in sites]
12.     await asyncio.gather(*tasks)
13.
14. def main():
15.     sites = [
16.         'http://c.biancheng.net',
17.         'http://c.biancheng.net/c',
18.         'http://c.biancheng.net/python'
19.     ]
20.     start_time = time.perf_counter()
21.
22.     loop = asyncio.get_event_loop()
23.     try:
24.         loop.run_until_complete(download_all(sites))
25.     finally:
26.         loop.close()
27.
28.     end_time = time.perf_counter()
29.     print('Download {} sites in {} seconds'.format(len(sites), end_time - start_time))
30.
31. if __name__ == '__main__':
32.     main()
```

运行结果为：

```
Read 52053 from http://c.biancheng.net
Read 30718 from http://c.biancheng.net/c
Read 34470 from http://c.biancheng.net/python
Download 3 sites in 0.12174049999999999 seconds
```

注意，此程序运行前，需确保已安装好 aiohttp 模块，此模块可直接执行 `pip install aiohttp` 命令安装。

上面程序中，`Async` 和 `await` 关键字是 `Asyncio` 的最新写法，表示这个语句（函数）是非阻塞的，正好对应前面所讲的事件循环的概念，即如果任务执行的过程需要等待，则将其放入等待状态的列表中，然后继续执行预备状态列表里的任务。

另外在主函数中，第 22-26 行代码表示拿到事件循环对象，并运行 `download_all()` 函数，直到其结束，最后关闭这个事件循环对象。

值得一提的，如果读者使用 Python 3.7 及以上版本，则 22-26 行代码可以直接用 `asyncio.run(download_all(sites))` 来代替。

至于 `Asyncio` 版本的函数 `download_all()`，和之前多线程版本有很大的区别：

1. 这里的 `asyncio.ensure_future(coro)` 表示对输入的协程 `coro` 创建一个任务，安排它的执行，并返回此任务对象。可以看到，这里对每一个网站的下载，都创建了一个对应的任务。

注意，Python 3.7+ 版本之后，可以使用 `asyncio.create_task(coro)` 等效替代 `asyncio.ensure_future(coro)`。

2. `asyncio.gather()` 表示在事件循环对象中运行 `aws` 序列的所有任务。

可以看到，其输出结果显示用时只有 0.12s，比之前的多线程版本效率更高，充分体现其优势。

当然，除了例子中用到的这几个函数，`Asyncio` 还提供了很多其他的用法，你可以查看 [Python 事件循环官方文档](#) 进行了解。

## Asyncio 有缺陷吗？

通过以上的学习，明显看到了 `Asyncio` 的强大。但是，任何一种方案都不是完美的，都存在一定的局限性，`Asyncio` 同样如此。

实际工作中，想用好 `Asyncio`，特别是发挥其强大的功能，很多情况下必须得有相应的 Python 库支持。前面章节在学习多线程编程中使用的是 `requests` 库，但本节使用的是 `aiohttp` 库，原因在于 `requests` 库并不兼容 `Asyncio`，而 `aiohttp` 库兼容。`Asyncio` 软件库的兼容性问题，在 Python3 的早期一直是个大问题，但是随着技术的发展，这个问题正逐步得到解决。

另外，使用 `Asyncio` 时，因为在任务调度方面有了更大的自主权，写代码时就得更加注意，不然很容易出错。

## 24. Python GIL 全局解释器锁详解（深度剖析）

通过前面的学习，我们了解了 Python 并发编程的特性以及什么是多线程编程。其实除此之外，Python 多线程还有一个很重要的知识点，就是本节要讲的 GIL。

GIL，中文译为全局解释器锁。在讲解 GIL 之前，首先通过一个例子来直观感受一下 GIL 在 Python 多线程程序运行的影响。

首先运行如下程序：

```
1. import time
2. start = time.clock()
3. def Countdown(n):
4.     while n > 0:
5.         n -= 1
6. Countdown(100000)
7. print("Time used:", (time.clock() - start))
```

运行结果为：

```
Time used: 0.0039529000000000005
```

在我们的印象中，使用多个（适量）线程是可以加快程序运行效率的，因此可以尝试将上面程序改成如下方式：

```
1. import time
2. from threading import Thread
3. start = time.clock()
4. def Countdown(n):
5.     while n > 0:
6.         n -= 1
7. t1 = Thread(target=CountDown, args=[100000 // 2])
8. t2 = Thread(target=CountDown, args=[100000 // 2])
9. t1.start()
10. t2.start()
11. t1.join()
12. t2.join()
13. print("Time used:", (time.clock() - start))
```

运行结果为：

```
Time used: 0.006673
```

可以看到，此程序中使用了 2 个线程来执行和上面代码相同的工作，但从输出结果中可以看到，运行效率非但没有提高，反而降低了。



如果使用更多线程进行尝试，会发现其运行效率和 2 个线程效率几乎一样（本机器测试使用 4 个线程，其执行效率约为 0.005）。这里不再给出具体测试代码，有兴趣的读者可自行测试。

是不是和你猜想的结果不一样？事实上，得到这样的结果是肯定的，因为 GIL 限制了 Python 多线程的性能不会像我们预期的那样。

那么，什么是 GIL 呢？GIL 是 CPython 解释器（平常称为 Python）中的一个技术术语，中文译为全局解释器锁，其本质上类似操作系统的 Mutex。GIL 的功能是：在 CPython 解释器中执行的每一个 Python 线程，都会先锁住自己，以阻止别的线程执行。

当然，CPython 不可能容忍一个线程一直独占解释器，它会轮流执行 Python 线程。这样一来，用户看到的就是“伪”并行，即 Python 线程在交替执行，来模拟真正并行的线程。

有读者可能会问，既然 CPython 能控制线程伪并行，为什么还需要 GIL 呢？其实，这和 CPython 的底层内存管理有关。

CPython 使用引用计数来管理内容，所有 Python 脚本中创建的实例，都会配备一个引用计数，来记录有多少个指针来指向它。当实例的引用计数的值为 0 时，会自动释放其所占的内存。

举个例子，看如下代码：

```
>>> import sys
>>> a = []
>>> b = a
>>> sys.getrefcount(a)
3
```

可以看到，a 的引用计数值为 3，因为有 a、b 和作为参数传递的 getrefcount 都引用了一个空列表。

假设有两个 Python 线程同时引用 a，那么双方就都会尝试操作该数据，很有可能造成引用计数的条件竞争，导致引用计数只增加 1（实际应增加 2），这造成的后果是，当第一个线程结束时，会把引用计数减少 1，此时可能已经达到释放内存的条件（引用计数为 0），当第 2 个线程再次视图访问 a 时，就无法找到有效的内存了。

所以，CPython 引进 GIL，可以最大程度上规避类似内存管理这样复杂的竞争风险问题。

## Python GIL 底层实现原理

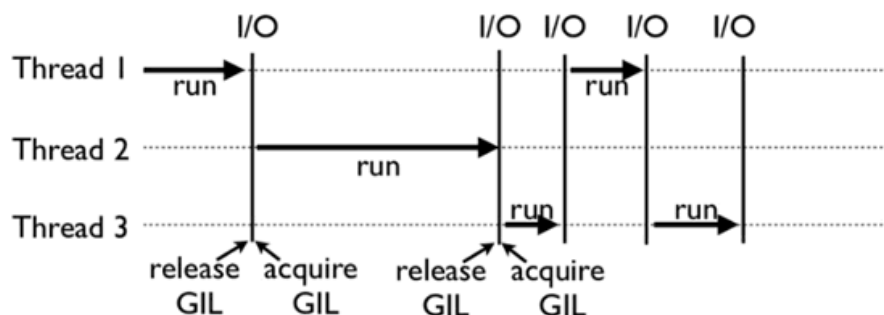


图 1 GIL 工作流程示意图

上面这张图，就是 GIL 在 Python 程序的工作示例。其中，Thread 1、2、3 轮流执行，每一个线程在开始执行时，都会锁住 GIL，以阻止别的线程执行；同样的，每一个线程执行完一段后，会释放 GIL，以允许别的线程开始利用资源。

读者可能会问，为什么 Python 线程会去主动释放 GIL 呢？毕竟，如果仅仅要求 Python 线程在开始执行时锁住 GIL，且永远不去释放 GIL，那别的线程就都没有运行的机会。其实，CPython 中还有另一个机制，叫做间隔式检查（`check_interval`），意思是 CPython 解释器会去轮询检查线程 GIL 的锁住情况，每隔一段时间，Python 解释器就会强制当前线程去释放 GIL，这样别的线程才能有执行的机会。

注意，不同版本的 Python，其间隔式检查的实现方式并不一样。早期的 Python 是 100 个刻度（大致对应了 1000 个字节码）；而

Python 3 以后，间隔时间大致为 15 毫秒。当然，我们不必细究具体多久会强制释放 GIL，读者只需要明白，CPython 解释器会在一个“合理”的时间范围内释放 GIL 就可以了。

整体来说，每一个 Python 线程都是类似这样循环的封装，来看下面这段代码：

```
1.  for (;;) {
2.      if (--ticker < 0) {
3.          ticker = check_interval;
4.          /* Give another thread a chance */
5.          PyThread_release_lock(interpreter_lock);
6.          /* Other threads may run now */
7.          PyThread_acquire_lock(interpreter_lock, 1);
8.      }
9.      bytecode = *next_instr++;
10.     switch (bytecode) {
11.         /* execute the next instruction ... */
12.     }
13. }
```

从这段代码中可以看出，每个 Python 线程都会先检查 ticker 计数。只有在 ticker 大于 0 的情况下，线程才会去执行自己的代码。

## Python GIL 不能绝对保证线程安全

注意，有了 GIL，并不意味着 Python 程序员就不用去考虑线程安全了，因为即便 GIL 仅允许一个 Python 线程执行，但别忘了 Python 还有 check interval 这样的抢占机制。

比如，运行如下代码：

```
1.  import threading
2.  n = 0
3.  def foo():
4.      global n
5.      n += 1
6.  threads = []
7.  for i in range(100):
8.      t = threading.Thread(target=foo)
9.      threads.append(t)
10. for t in threads:
11.     t.start()
12. for t in threads:
```

```
13.     t.join()
14.  print(n)
```

执行此代码会发现，其大部分时候会打印 100，但有时也会打印 99 或者 98，原因在于 `n+=1` 这一句代码让线程并不安全。如果去翻译 `foo` 这个函数的字节码就会发现，它实际上是由下面四行字节码组成：

```
>>> import dis
>>> dis.dis(foo)
LOAD_GLOBAL      0 (n)
LOAD_CONST      1 (1)
INPLACE_ADD
STORE_GLOBAL     0 (n)
```

而这四行字节码中间都是有可能被打断的！所以，千万别以为有了 GIL 程序就不会产生线程问题，我们仍然需要注意线程安全。

## 25. 深度解析 Python 垃圾回收机制（超级详细）

我们知道，目前的计算机都采用的是图灵机架构，其本质就是用一条无限长的纸带，对应今天的存储器。随后在工程学的推演中，逐渐出现了寄存器、易失性存储器（内存）以及永久性存储器（硬盘）等产品。由于不同的存储器，其速度越快，单位价格也就越昂贵，因此，妥善利用好每一寸告诉存储器的空间，永远是系统设计的一个核心。

Python 程序在运行时，需要在内存中开辟出一块空间，用于存放运行时产生的临时变量，计算完成后，再将结果输出到永久性存储器中。但是当数据量过大，或者内存空间管理不善，就容易出现内存溢出的情况，程序可能会被操作系统终止。

而对于服务器这种用于永不中断的系统来说，内存管理就显得更为重要了，不然很容易引发内存泄漏。

这里的内存泄漏是指程序本身没有设计好，导致程序未能释放已不再使用的内存，或者直接失去了对某段内存的控制，造成了内存的浪费。

那么，对于不会再用到的内存空间，Python 是通过什么机制来管理的呢？其实在前面章节已大致接触过，就是引用计数机制。

### Python 引用计数机制

在学习 Python 的整个过程中，我们一直在强调，Python 中一切皆对象，也就是说，在 Python 中你用到的一切变量，本质上都是类对象。

那么，如何知道一个对象永远都不能再使用了呢？很简单，就是当这个对象的引用计数值为 0 时，说明这个对象永不再用，自然它就变成了垃圾，需要被回收。

举个例子：

```
1.  import os
2.  import psutil
3.
4.  # 显示当前 python 程序占用的内存大小
5.  def show_memory_info(hint):
6.      pid = os.getpid()
7.      p = psutil.Process(pid)
8.
9.      info = p.memory_full_info()
10.     memory = info.uss / 1024. / 1024
11.     print('{} memory used: {} MB'.format(hint, memory))
12. def func():
13.     show_memory_info('initial')
14.     a = [i for i in range(1000000)]
15.     show_memory_info('after a created')
16.
17. func()
18. show_memory_info('finished')
```

输出结果为：

```
initial memory used: 47.19140625 MB
after a created memory used: 433.91015625 MB
finished memory used: 48.109375 MB
```

注意，运行此程序之前，需安装 psutil 模块（获取系统信息的模块），可使用 pip 命令直接安装，执行命令为 `$pip install psutil`，如果遇到 Permission denied 安装失败，请加上 sudo 重试。

可以看到，当调用函数 func() 且列表 a 被创建之后，内存占用迅速增加到了 433 MB，而在函数调用结束后，内存则返回正常。这是因为，函数内部声明的列表 a 是局部变量，在函数返回后，局部变量的引用会注销掉，此时列表 a 所指代对象的引用计数为 0，Python 便会执行垃圾回收，因此之前占用的大量内存就又回来了。

明白了这个原理后，稍微修改上面的代码，如下所示：

```
1. def func():
2.     show_memory_info('initial')
3.     global a
4.     a = [i for i in range(1000000)]
5.     show_memory_info('after a created')
6.
7. func()
8. show_memory_info('finished')
```

输出结果为：

```
initial memory used: 48.88671875 MB
after a created memory used: 433.94921875 MB
finished memory used: 433.94921875 MB
```

上面这段代码中，global a 表示将 a 声明为全局变量，则即使函数返回后，列表的引用依然存在，于是 a 对象就不会被当做垃圾回收掉，依然占用大量内存。

同样，如果把生成的列表返回，然后在主程序中接收，那么引用依然存在，垃圾回收也不会被触发，大量内存仍然被占用着：

```
1. def func():
2.     show_memory_info('initial')
3.     a = [i for i in derange(1000000)]
4.     show_memory_info('after a created')
5.     return a
6.
7. a = func()
8. show_memory_info('finished')
```

输出结果为：

```
initial memory used: 47.96484375 MB
after a created memory used: 434.515625 MB
finished memory used: 434.515625 MB
```

以上最常见的几种情况，下面由表及里，深入看一下 Python 内部的引用计数机制。先来分析一段代码：

```
1. import sys
2. a = []
3. # 两次引用，一次来自 a，一次来自 getrefcount
4. print(sys.getrefcount(a))
5. def func(a):
6.     # 四次引用，a，python 的函数调用栈，函数参数，和 getrefcount
7.     print(sys.getrefcount(a))
8. func(a)
9. # 两次引用，一次来自 a，一次来自 getrefcount，函数 func 调用已经不存在
10. print(sys.getrefcount(a))
```

输出结果为：

```
2
4
2
```

注意，`sys.getrefcount()` 函数用于查看一个变量的引用次数，不过别忘了，`getrefcount` 本身也会引入一次计数。另一个要注意的是，在函数调用发生的时候，会产生额外的两次引用，一次来自函数栈，另一个是函数参数。

```
1. import sys
2. a = []
3. print(sys.getrefcount(a)) # 两次
4. b = a
5. print(sys.getrefcount(a)) # 三次
6. c = b
7. d = b
8. e = c
9. f = e
10. g = d
11. print(sys.getrefcount(a)) # 八次
```

输出结果为：

```
2
3
8
```

分析一下这段代码，a、b、c、d、e、f、g 这些变量全部指代的是同一个对象，而 `sys.getrefcount()` 函数并不是统计一个指针，而是要统计一个对象被引用的次数，所以最后一共会有 8 次引用。

理解引用这个概念后，引用释放是一种非常自然和清晰的思想。相比 C 语言中需要使用 `free` 去手动释放内存，Python 的垃圾回收在这里可以说是省心省力了。

不过，有读者还是会好奇，如果想手动释放内存，应该怎么做呢？方法同样很简单，只需要先调用 `del a` 来删除一个对象，然后强制调用 `gc.collect()` 即可手动启动垃圾回收。例如：

```
1. import gc
2.
3. show_memory_info('initial')
4. a = [i for i in range(1000000)]
5. show_memory_info('after a created')
6. del a
7. gc.collect()
8. show_memory_info('finish')
9. print(a)
```

输出结果为：

```
initial memory used: 48.1015625 MB
after a created memory used: 434.3828125 MB
finish memory used: 48.33203125 MB
NameError Traceback (most recent call last)
<ipython-input-12-153e15063d8a> in <module>
    11
    12 show_memory_info('finish')
---> 13 print(a)
NameError: name 'a' is not defined
```

是不是觉得垃圾回收非常简单呢？这里再问大家一个问题：引用次数为 0 是垃圾回收启动的充要条件吗？还有没有其他可能性呢？

其实，引用计数是其中最简单的实现，引用计数并非充要条件，它只能算作充分非必要条件，至于其他的可能性，下面所讲的循环引用正是其中一种。

## 循环引用

首先思考一个问题，如果有两个对象，之间互相引用，且不再被别的对象所引用，那么它们应该被垃圾回收吗？

举个例子：

```
1. def func():
2.     show_memory_info('initial')
```

```
3.     a = [i for i in range(1000000)]
4.     b = [i for i in range(1000000)]
5.     show_memory_info('after a, b created')
6.     a.append(b)
7.     b.append(a)
8.
9.     func()
10.    show_memory_info('finished')
```

输出结果为：

```
initial memory used: 47.984375 MB
after a, b created memory used: 822.73828125 MB
finished memory used: 821.73046875 MB
```

程序中，a 和 b 互相引用，并且作为局部变量在函数 func 调用结束后，a 和 b 这两个指针从程序意义上已经不存在，但从输出结果中看到，依然有内存占用，这是为什么呢？因为互相引用导致它们的引用数都不为 0。

试想一下，如果这段代码出现在生产环境中，哪怕 a 和 b 一开始占用的空间不是很大，但经过长时间运行后，Python 所占用的内存一定会变得越来越大，最终撑爆服务器，后果不堪设想。

有读者可能会说，互相引用还是很容易被发现的呀，问题不大。可是，更隐蔽的情况是出现一个引用环，在工程代码比较复杂的情况下，引用环真不一定能被轻易发现。那么应该怎么做呢？

事实上，Python 本身能够处理这种情况，前面刚刚讲过，可以显式调用 gc.collect() 来启动垃圾回收，例如：

```
1.     import gc
2.
3.     def func():
4.         show_memory_info('initial')
5.         a = [i for i in range(1000000)]
6.         b = [i for i in range(1000000)]
7.         show_memory_info('after a, b created')
8.         a.append(b)
9.         b.append(a)
10.
11.    func()
12.    gc.collect()
13.    show_memory_info('finished')
```

输出结果为：



```
initial memory used: 49.51171875 MB
after a, b created memory used: 824.1328125 MB
finished memory used: 49.98046875 MB
```

事实上，Python 使用标记清除（mark-sweep）算法和分代收集（generational），来启用针对循环引用的自动垃圾回收。

先来看标记清除算法。我们先用图论来理解不可达的概念。对于一个有向图，如果从一个节点出发进行遍历，并标记其经过的所有节点；那么，在遍历结束后，所有没有被标记的节点，我们就称之为不可达节点。显而易见，这些节点的存在是没有任何意义的，自然的，我们就需要对它们进行垃圾回收。

当然，每次都遍历全图，对于 Python 而言是一种巨大的性能浪费。所以，在 Python 的垃圾回收实现中，标记清除算法使用双向链表维护了一个数据结构，并且只考虑容器类的对象（只有容器类对象才有可能产生循环引用）。

而分代收集算法，则是将 Python 中的所有对象分为三代。刚刚创立的对象是第 0 代；经过一次垃圾回收后，依然存在的对象，便会依次从上一代挪到下一代。而每一代启动自动垃圾回收的阈值，则是可以单独指定的。当垃圾回收器中新增对象减去删除对象达到相应的阈值时，就会对这一代对象启动垃圾回收。

事实上，分代收集基于的思想是，新生的对象更有可能被垃圾回收，而存活更久的对象也有更高的概率继续存活。因此，通过这种做法，可以节约不少计算量，从而提高 Python 的性能。

由于篇幅有限，这里不再对标记清除算法和分代收集算法的具体实现所详细介绍，有兴趣的读者可自行百度搜索。