

## 线程与进程

线程与进程是操作系统里面的术语，简单来讲，每一个应用程序都有一个自己的进程。操作系统会为这些进程分配一些执行资源，例如内存空间等。在进程中，又可以创建一些线程，他们共享这些内存空间，并由操作系统调用，以便并行计算。

我们都知道现代操作系统比如 Mac OS X，UNIX，Linux，Windows 等可以同时运行多个任务。打个比方，你一边在用浏览器上网，一边在听敲代码，一边用 Markdown 写博客，这就是多任务，至少同时有 3 个任务正在运行。当然还有很多任务悄悄地在后台同时运行着，只是桌面上没有显示而已。对于操作系统来说，一个任务就是一个进程（Process），比如打开一个浏览器就是启动一个浏览器进程，打开 PyCharm 就是一个启动了一个 PtCharm 进程，打开 Markdown 就是启动了一个 Md 的进程。

虽然现在多核 CPU 已经非常普及了。可是由于 CPU 执行代码都是顺序执行的，这时候我们就会有疑问，单核 CPU 是怎么执行多任务的呢？

其实就是操作系统轮流让各个任务交替执行，任务 1 执行 0.01 秒，切换到任务 2 ，任务 2 执行 0.01 秒，再切换到任务 3 ，执行 0.01秒……这样反复执行下去。表面上看，每个任务都是交替执行的，但是，由于 CPU的执行速度实在是太快了，我们肉眼和感觉上没法识别出来，就像所有任务都在同时执行一样。

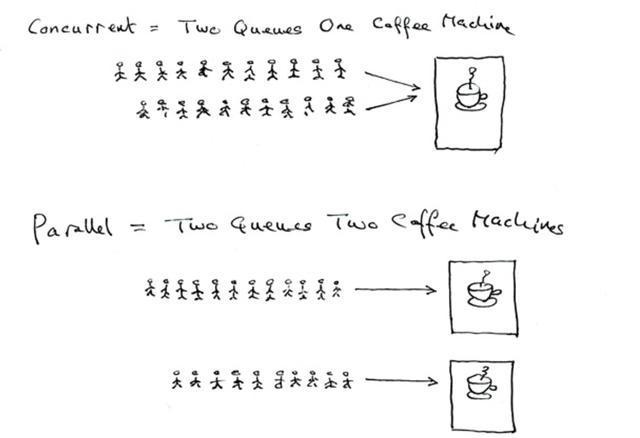
真正的并行执行多任务只能在多核 CPU 上实现，但是，由于任务数量远远多于 CPU 的核心数量，所以，操作系统也会自动把很多任务轮流调度到每个核心上执行。

**并发与并行**

**并发是指一个处理器同时处理多个任务。**

**并发(concurrency)：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的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。  
并行是指多个处理器或者是多核的处理器同时处理多个不同的任务。   
并发是逻辑上的同时发生（simultaneous），而并行是物理上的同时发生。**

**并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。**  
**来个比喻：并发是一个人（CPU）同时吃三个馒头，而并行是三个人（CPU）同时吃三个馒头。**



有些进程不仅仅只是干一件事的啊，比如浏览器，我们可以播放时视频，播放音频，看文章，编辑文章等等，其实这些都是在浏览器进程中的子任务。在一个进程内部，要同时干多件事，就需要同时运行多个“子任务”，我们把进程内的这些“子任务”**称为线程（Thread）**。

由于每个进程至少要干一件事，所以，一个进程至少有一个线程。当然，一个进程也可以有多个线程，多个线程可以同时执行，多线程的执行方式和多进程是一样的，也是由操作系统在多个线程之间快速切换，让每个线程都短暂地交替运行，看起来就像同时执行一样。

那么在 Python 中我们要同时执行多个任务怎么办？

有两种解决方案：

一种是启动多个进程，每个进程虽然只有一个线程，但多个进程可以一块执行多个任务。

还有一种方法是启动一个进程，在一个进程内启动多个线程，这样，多个线程也可以一块执行多个任务。

当然还有第三种方法，就是启动多个进程，每个进程再启动多个线程，这样同时执行的任务就更多了，当然这种模型更复杂，实际很少采用。

总结一下就是，多任务的实现有3种方式：

* 多进程模式
* 多线程模式
* 多进程+多线程模式

同时执行多个任务通常各个任务之间并不是没有关联的，而是需要相互通信和协调，有时，任务 1 必须暂停等待任务 2 完成后才能继续执行，有时，任务 3 和任务 4 又不能同时执行，所以，多进程和多线程的程序的复杂度要远远高于我们前面写的单进程单线程的程序。

因为复杂度高，调试困难，所以，不是迫不得已，我们也不想编写多任务。但是，有很多时候，没有多任务还真不行。想想在电脑上看电影，就必须由一个线程播放视频，另一个线程播放音频，否则，单线程实现的话就只能先把视频播放完再播放音频，或者先把音频播放完再播放视频，这显然是不行的。

## 多线程编程

其实创建线程之后，线程并不是始终保持一个状态的，其状态大概如下：

* new 创建
* Runnable 就绪。等待调度
* Running 运行
* Blocked 阻塞，阻塞可能在 Wait Locked Sleeping
* Dead 消亡

线程有着不同的状态，也有不同的类型。大致可分为：

* 主线程
* 子线程
* 守护线程（后台线程）
* 前台线程

### 线程的创建

Python 提供两个模块进行多线程的操作，分别是 thread和 threading

前者是比较低级的模块，用于更底层的操作，一般应用级别的开发不常用。

#!/usr/bin/env python2

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

import time

import threading

class **MyThread**(threading.Thread):

def **run**(*self*):

for i in range(5):

print *"thread{0}, @number:{1}"*.format(*self*.name, i)

time.sleep(1)

def **main**():

print *"Start main threading"*

threads = [MyThread() for i in range(3)]

#创建三个线程

for t in threads:

t.start()

print *"End Main threading"*

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

main()

执行结果：

Start main threading

threadThread-1, @number:0

threadThread-2, @number:0

threadThread-3, @number:0

End Main threading

threadThread-2, @number:1

threadThread-3, @number:1

threadThread-1, @number:1

threadThread-2, @number:2

threadThread-3, @number:2

threadThread-1, @number:2

threadThread-2, @number:3

threadThread-3, @number:3

threadThread-1, @number:3

threadThread-2, @number:4

threadThread-3, @number:4

threadThread-1, @number:4

### 线程合并（join方法）

上面的示例打印出来的结果来看，主线程结束后，子线程还在运行。那么我们需要主线程要等待子线程运行完后，再退出，要怎么办呢？

这时候，就需要用到 join方法了。

在上面的例子，新增一段代码，具体如下：

#!/usr/bin/env python2

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

import time

import threading

class **MyThread**(threading.Thread):

def **run**(*self*):

for i in range(5):

print *"thread{0}, @number:{1}"*.format(*self*.name, i)

time.sleep(1)

def **main**():

print *"Start main threading"*

threads = [MyThread() for i in range(3)]

#创建三个线程

for t in threads:

t.start()

for t in threads:

t.join()

print *"End Main threading"*

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

main()

执行结果：

Start main threading

threadThread-1, @number:0

threadThread-2, @number:0

threadThread-3, @number:0

threadThread-1, @number:1threadThread-3, @number:1threadThread-2, @number:1

threadThread-1, @number:2threadThread-3, @number:2threadThread-2, @number:2

threadThread-3, @number:3

threadThread-1, @number:3

threadThread-2, @number:3

threadThread-3, @number:4

threadThread-2, @number:4threadThread-1, @number:4

End Main threading

从打印的结果，可以清楚看到，相比上面示例打印出来的结果，主线程是在等待子线程运行结束后才结束的。

### 线程同步与互斥锁

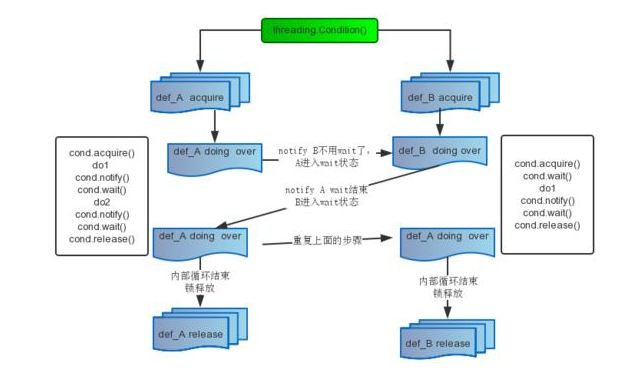
使用线程加载获取数据，通常都会造成数据不同步的情况。当然，这时候我们可以给资源进行加锁，也就是访问资源的线程需要获得锁才能访问。

参考Python专题——常用的Python模块

### Condition 条件变量

实用锁可以达到线程同步，但是在更复杂的环境，需要针对锁进行一些条件判断。Python 提供了 Condition 对象。使用 Condition 对象可以在某些事件触发或者达到特定的条件后才处理数据，Condition 除了具有 Lock 对象的 acquire 方法和 release 方法外，还提供了 wait 和 notify 方法。线程首先 acquire 一个条件变量锁。如果条件不足，则该线程 wait，如果满足就执行线程，甚至可以 notify 其他线程。其他处于 wait 状态的线程接到通知后会重新判断条件。

其中条件变量可以看成不同的线程先后 acquire 获得锁，如果不满足条件，可以理解为被扔到一个（ Lock 或 RLock ）的 waiting 池。直达其他线程 notify 之后再重新判断条件。不断的重复这一过程，从而解决复杂的同步问题。



### 线程间通讯

从一个线程向另一个线程发送数据最安全的方式可能就是使用 queue 库中的队列了。创建一个被多个线程共享的 Queue对象，这些线程通过使用 put()和 get() 操作来向队列中添加或者删除元素。

Python 还提供了 Event 对象用于线程间通信，它是由线程设置的信号标志，如果信号标志位真，则其他线程等待直到信号接触。

Event 对象实现了简单的线程通信机制，它提供了设置信号，清楚信号，等待等用于实现线程间的通信。

### 后台线程

默认情况下，主线程退出之后，即使子线程没有 join。那么主线程结束后，子线程也依然会继续执行。如果希望主线程退出后，其子线程也退出而不再执行，则需要设置子线程为后台线程。Python 提供了 setDeamon方法。

## 多进程编程