python 多线程 -threading 模块

threading 是我们常用的用于 python 多线程的模块，其功能更加丰富。下面我们就来开始学习这个模块。

同样的，我这里声明一样我使用的版本是 python2.7，不同版本直接可能存在差异。

老规矩，使用 help() 函数获取帮助文档，看看里面有什么内容。

threading 模块中提供了一个 thread 的类，注意不要和 thread 模块搞混了，两者差别还是很大的。 thread 这个类可以实例化一个对象，每个对象代表一个线程，可以调用其中的 run() 方法来开启一个线程的运行。而且，既然他是一个类，我们也可以派生一个子类，实现多种不同的线程创建方式。下面我们来看看如何使用这个类来实现多线程。

1.创建一个 Thread 的实例，传给它一个函数。

某处抠来的代码示例：

复制代码

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import threading
from time import sleep, ctime

loops = [4, 2]


def loop(nloop, nsec):
    print ' 开始线程 ', nloop, ' 于', ctime()
    sleep(nsec)
    print 'loop 函数 ', nloop, ' 完成于 ', ctime()


def main():
    print ' 开始主线程： ', ctime()
    threads = []            # 一个用于储存线程对象的列表
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread(target=loop, args=(i, loops[i]))      # 每次循环创建一个 Thread 的实例
        threads.append(t)      # 将新创建的对象放到一个列表中

    for i in nloops:
        threads[i].start()     # 每次循环运行一个线程
```

```
    for i in nloops:
        threads[i].join()      #  等待子线程的完成

    print ' 主线程完成：   ', ctime()

if __name__ == '__main__':
    main()
```

复制代码

输出：

首先我们来看看如何创建一个 Thread 的实例，先看看其构造函数接受哪些参数：

__init__(self, group=None, target=None, name=None, args=(), kwargs=None, verbose=None)

self：实例本身，特定格式，这里忽略。

group：应该是 None，这里是留给未来的扩展，当 ThreadGroup 类实现的时候。我们不用理它。

target：一个可调用的对象，可以是一个函数，也可以是一个类。如果是一个类的时候，将调用类内部的 __call__ 方法。

name： 用字符串表示线程的名字。 在默认的情况下， 遵从 "Thread-N" 的格式， 其中 N 是一个十进制的数字，从 1 开始。可以为多个线程设定一样的名字。

args： 是一个元祖，以位置参数的方式传给被调用对象。

kwargs：是一个字典，以关键字参数的方式传给对象。

verbose：未有说明，试了一下，设置了改参数的值后将输出详细的说明，但是设置的值对这个输出结果的影响未知， 据我观察输出的顺序多是随机的， 也就是这里有值就行， 下面是其中一次的输出结果：

一般情况下用不到，这里了解下就行。

注意：我们在 thread 模块中说过， thread.start_new_thread() 函数不接受关键字传参，而这里的 threading.Thread 类则建议全部用关键字传参。

知道了如何创建 thread 对象之后，我们再来看看其内部有什么方法：

1. getName(self)

获取线程的名字，就是初始函数中的 name 的值。同样的，我们可以使用实例的 name 属性来获取。注意这里是类里面的方法，所以 self 不是要给的参数。

2. setName(self, name)

为线程设定名字 name ，和初始函数中的设定一样，不过可以用来重命名线程名。

3. start(self)

开始线程的运行，每个线程对象只有调用最多一次。它将调用被调用对象的 run() 方法，并控制各个对象独立运行。也就是说被调用的对象必要要有 run() 方法，在使用 Thread 类来实例化对象的时候，因为 Thread 中已经有了 run() 方法了，所以可以不用理。但是，在基础 Thread 创建子类的时候，一般我们要重写子类的 run() 方法。

当一个线程对象重复调用 start() 方法时，将触发 RuntimeError 异常。

4. run(self)

方法表示线程是活跃的，也就是要运行线程。我们可以在子类中重写这个方法。

它和 start() 的差别在于： start() 更像是一个管理器，它负责调用对象的 run() 方法，而 run() 方法更像是一个入口函数。如果单独运行 run() 方法，则只是运行了一个线程，而运行 start() 则会运行多个 run() 方法，并管理这些线程。

它们的关系更像：

这只是一个比喻理解，并不保证学术性上的正确。

所以， run 更像是个人行为，而 start 更像是一个团队，而团队中有很多的个人。一个人同时只能做一件事，而一个团队可以为每个人分配任务，那么这个团队同时就能做很多事情了。

我们再用代码来演示一下：

复制代码

```
loops = [4, 2]


def loop(nloop, nsec):
```

```
    print ' 开始线程  ', nloop, ' 于', ctime()
    sleep(nsec)
    print 'loop   函数 ', nloop, ' 完成于  ', ctime()


def main():
    print ' 开始主线程：    ', ctime()
    threads = []              #  一个用于储存线程对象的列表
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread(target=loop, args=(i, loops[i]))        #  每次循环创建一个     Thread 的
实例
        threads.append(t)       #  将新创建的对象放到一个列表中

    for i in nloops:
        threads[i].run()     #  使用对象的    run  方法

    print ' 主线程完成：    ', ctime()

if __name__ == '__main__':
    main()
```

复制代码

可以看到又变成了单线程行为了。

5. isAlive(self)/is_alive

is_alive = isAlive(self)，功能相同。

判断线程是否还活动着。　在线程调用了　run()　方法之后，　而线程又没运行结束时，　返回 True，否则返回　False。（模块中的方法　threading.enumerate()　将返回所有活动线程对象，并将这些对象放在一个列表中，本质就是调用这个方法来判断线程是否活动。　　）

6. join(self, timeout=None)

阻塞主线程，直到调用该方法的子线程运行完毕或者超时。　　timeout　表示超时时间，可以是一个数，例如整数，小数，分数，表示超时的时间，单位是秒。返回值为　　None，可以

在 join 超时之后调用 isAlive 确认线程是否结束。 如果线程还活动， 说明 join 触发了超时，此时你可以继续调用 join 或者做其他处理。 当 timeout 没有给或者为 None 的时候， 将阻塞直到调用此方法的子线程结束。一个线程可以调用多次 join 方法。

为一个未开始的线程调用 join 将触发 RuntimeError 异常，当然函数内部主动触发的另算。另外，如果试图使当前线程陷入死循环式的调用，也会触发该异常。

在开始的示例中我们看到了使用 join 的情况，现在我们来看看不使用 join 的情况：

复制代码

```
loops = [4, 2]


def loop(nloop, nsec):
    print ' 开始线程 ', nloop, ' 于', ctime()
    sleep(nsec)
    print 'loop 函数 ', nloop, ' 完成于 ', ctime()


def main():
    print ' 开始主线程： ', ctime()
    threads = []
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread(target=loop, args=(i, loops[i]))
        threads.append(t)
    for i in nloops:
        threads[i].start()

    print ' 主线程完成： ', ctime()

if __name__ == '__main__':
    main()
```

复制代码

看一下其输出：

可以看见主线程早就完成了，但是 threading 不同于 thread 模块，它还进行了收尾工作，也就是剩下的线程都运行完了。

阻塞主线程的意义更大是在于： 主线程需要子线程处理后的结果， 所以需要等待子线程

输出结果。

7. setDaemon(self, daemonic)

设置守护线程，若 setDaemon(True) ，则代表将线程转入后台执行。

必须在线程开始之前调用设置，否则触发 RuntimeError 异常。它的默认值继承于创建子线程的线程，在主线程中创建时，主线程属于非守护线程，所以其子线程的默认值也是 False。如果将某个线程设为后台进行，那么 python 的主线程在结束的时候不会进行收尾工作。

也就是说一旦主线程结束，整个进程也就结束了，那么子线程也就结束了。其实说是守护线程，但其重要程度反而是最低的。所以有些书也将其称为“不太重要的线程”。

代码示例：

复制代码

```
loops = [10, 5]


def loop(nloop, nsec):
    print '开始线程 ', nloop, '于', ctime()
    sleep(nsec)
    print 'loop 函数 ', nloop, '完成于 ', ctime()



def main():
    print '开始主线程： ', ctime()
    threads = []            # 一个用于储存线程对象的列表
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread(target=loop, args=(i, loops[i]))    # 每次循环创建一个 Thread 的实例
        threads.append(t)    # 将新创建的对象放到一个列表中
    threads[0].setDaemon(True)    # 将第一个线程设为守护线程
    for i in nloops:
        threads[i].start()    # 每次循环运行一个线程
print '主线程完成： ', ctime()

if __name__ == '__main__':
    main()
```

复制代码

可以看到程序结束的时候，线程 0 并没有结束。

注意：对守护线程使用 join 还是有效的。

8. isDaemon(self)

判断一个线程是否是守护线程，返回布尔值。

2. 创建一个 Thread 的实例，传给它一个可调用的类

前面我们都是使用一个函数作为 target，除此之外，我们还能以一个可调用的类为目标。而所谓可调用的类，其实就是在类里面实现了 __call__ 方法，下面看代码实例：

复制代码

```
loops = [4, 2]


def loop(nloop, nsec):
    print ' 开始线程 ', nloop, ' 于', ctime()
    sleep(nsec)
    print 'loop 函数 ', nloop, ' 完成于 ', ctime()


class ThreadFunc(object):
    def __init__(self, func, args):
        self.func = func
        self.args = args

    def __call__(self):    # 关键是要实现这个方法
        self.func(self.args[0], self.args[1])    # apply(self.func, self.args) 也有这种写法


def main():
    print ' 开始主线程： ', ctime()
    threads = []            # 一个用于储存线程对象的列表
    nloops = range(len(loops))
    for i in nloops:
        t = threading.Thread(target=ThreadFunc(func=loop, args=(i, loops[i])),)    # 每次循环创建一个 Thread 的实例，目标是一个类
        threads.append(t)    # 将新创建的对象放到一个列表中
```

```
for i in nloops:
    threads[i].start()    # 每次循环运行一个线程

for i in nloops:
    threads[i].join()    # 等待子线程的完成

print ' 主线程完成：  ', ctime()

if __name__ == '__main__':
    main()
```

复制代码

其核心就是通过调用类中的 __call__ 方法，该方法又调用了传入的函数对象。

3.创建一个 Thread 的子类

这种方法更加通用，这种方法的关键是在子类内部重写 run 方法。

代码示例：

复制代码

```
loops = [4, 2]


def loop(nloop, nsec):
    print ' 开始线程 ', nloop, ' 于', ctime()
    sleep(nsec)
    print 'loop 函数 ', nloop, ' 完成于 ', ctime()

class MyThread(threading.Thread):
    def __init__(self, func, args):
        threading.Thread.__init__(self)    # 调用父类的构造函数
        self.func = func
        self.args = args

    def run(self):
        apply(self.func, self.args)

def main():
    print ' 开始主线程：  ', ctime()
```

```
    threads = []            #  一个用于储存线程对象的列表
    nloops = range(len(loops))
    for i in nloops:
        t = MyThread(func=loop, args=(i, loops[i]))     #  使用我们自己的类来新建对象
        threads.append(t)      #  将新创建的对象放到一个列表中
    for i in nloops:
        threads[i].start()     #  每次循环运行一个线程

    for i in nloops:
        threads[i].join()      #  等待子线程的完成

    print ' 主线程完成：  ', ctime()

if __name__ == '__main__':
    main()
```

复制代码

一般推荐用这种方法，因为我们可以在类中实现其他方法，例如我并不在目标函数中输出结果，而是将结果作为返回值，而 run 方法得到这个返回值后，将其变成一个属性，这样只有我调用这个属性的时候，才会输出结果。

代码示例：

复制代码

```
loops = [4, 2]

def loop(nloop, nsec):
    return nloop + nsec        #  返回两个参数的和

class MyThread(threading.Thread):
    def __init__(self, func, args):
        threading.Thread.__init__(self)     #  调用父类的构造函数
        self.func = func
        self.args = args

    def run(self):
        self.result = apply(self.func, self.args)       #  调用函数的结果作为一个属性

def main():
    print ' 开始主线程：  ', ctime()
    threads = []            #  一个用于储存线程对象的列表
```

```
    nloops = range(len(loops))
    for i in nloops:
        t = MyThread(func=loop, args=(i, loops[i]))      #  使用我们自己的类来新建对象
        threads.append(t)       #  将新创建的对象放到一个列表中
    for i in nloops:
        threads[i].start()     #  每次循环运行一个线程

    for i in nloops:
        threads[i].join()       #  等待子线程的完成

    for i in nloops:
        print ' 执行结果为：   ', threads[i].result         #  打印该对象的属性

    print ' 主线程完成：   ', ctime()

if __name__ == '__main__':
    main()
```

复制代码

可以看出这种方法的扩展能力更强，所以一般推荐这种方法。

当然最终的选择还是看个人需求。

FUNCTIONS

1. Lock = allocate_lock(...)

和 thread 模块中的 allocate_lock() 方法一样，也是用来锁定线程的。返回一个锁对象（LockType），该对象在 thread 模块中详细讲过了，这里不再多说。

2. RLock(*args, **kwargs)

这是一个工厂函数，返回一个可重入的锁对象。和 Lock 的最大区别就是，该对象可以被重复的 acquire() ，也就是多次加锁。所以解锁的时候也要解锁多次。

3. Condition(*args, **kwargs)

其实可以把它看做是一个高级的锁，比 Lock，RLock 更加高级，且功能更多，它能实现复杂的线程同步功能。

threadiong.Condition 在内部维护一个琐对象（默认是 RLock），可以在创建 Condigtion 对象的时候把琐对象作为参数传入。Condition 也提供了 acquire，release 方法，其含义与琐的 acquire, release 方法一致，其实它只是简单的调用内部琐对象的对应的方法而已。

除此之外 Condition 还提供了如下方法 (特别要注意：这些方法只有在占用琐 (acquire) 之后才能调用，否则将会报 RuntimeError 异常。)

Condition.wait([timeout])

wait 方法释放内部所占用的琐，同时线程被挂起，直至接收到通知被唤醒或超时（如果提供了 timeout 参数的话）。当线程被唤醒并重新占有琐的时候，程序才会继续执行下去。

Condition.notify()

唤醒一个挂起的线程（如果存在挂起的线程）。注意：notify() 方法不会释放所占用的琐。

Condition.notifyAll()/Condition.notify_all()

唤醒所有挂起的线程（如果存在挂起的线程）。注意：方法不会释放所占用的琐。

为了方便大家理解，我去网上找了一个例子：戳这里

这个例子是一个捉迷藏的游戏，来具体介绍 threading.Condition 的基本使用。

假设这个游戏由两个人来玩，一个藏 (Hider)，一个找 (Seeker)。游戏的规则如下： 1. 游戏开始之后，Seeker 先把自己眼睛蒙上，蒙上眼睛后，就通知 Hider；2. Hider 接收通知后开始找地方将自己藏起来，藏好之后，再通知 Seeker 可以找了； 3. Seeker 接收到通知之后，就开始找 Hider。Hider 和 Seeker 都是独立的个体，在程序中用两个独立的线程来表示，在游戏过程中，两者之间的行为有一定的时序关系，我们通过 Condition 来控制这种时序关系。

复制代码

```
class Seeker(threading.Thread):
    def __init__(self, cond, name,):
        threading.Thread.__init__(self)
        self.cond = cond
        self.name = name

    def run(self):
        time.sleep(1)      # 确保先运行 Hider 中的方法
        self.cond.acquire()        # 2 加锁
        print self.name + ': 我已经把眼睛蒙上了 '
        self.cond.notify()    # 4 唤醒一个被挂起的线程，因为本线程并没有挂起，所以唤醒
```

的是下面的

```
        self.cond.wait()        # 5 挂起，等待别的线程唤醒
        print self.name + ':  我找到你了   ~_~'
        self.cond.notify()    # 7 唤醒别的休眠线程
        self.cond.release()        # 8 释放锁
        print self.name + ':  我赢了 '    # 9 结束


class Hider(threading.Thread):
    def __init__(self, cond, name):
        threading.Thread.__init__(self)
        self.cond = cond
        self.name = name

    def run(self):
        self.cond.acquire()        # 1 加锁
        self.cond.wait()        # 3 释放对琐的占用，同时线程挂起在这里，      直到被 notify 并重
新占有琐
        print self.name + ':  我已经藏好了，你快来找我吧    '
        self.cond.notify()    # 6 唤醒一个被休眠的线程
        self.cond.wait()        # 7 挂起当前线程
        self.cond.release()        # 8 被唤醒后释放锁
        print self.name + ':  被你找到了，哎  ~~~'      # 9 结束

cond = threading.Condition()
seeker = Seeker(cond, 'seeker')
hider = Hider(cond, 'hider')
seeker.start()
hider.start()
```

复制代码

这个例子可能和原文中的不一样，因为原文中写反了，同时我也稍微修改了一下代码来适应我的 2.7 版本，还加了些注释。

总体思路就是使用了一个锁来控制多个线程，可以使原本独立的相关线程实现通信。

4. Event(*args, **kwargs)

Event（事件）实现与 Condition 类似的功能，不过比 Condition 简单一点。它通过维护内部的标识符来实现线程间的同步问题。

同样的， Event 这个工程函数还有以下方法可以使用：

Event.wait([timeout])

堵塞线程，直到 Event 对象内部标识位被设为 True 或超时（如果提供了参数 timeout ）。

Event.set()

将标识位设为 Ture。

Event.clear()

将标识位设为 False。

Event.isSet()

判断标识位是否为 Ture。

5. enumerate()

返回一个列表，列表中的元素是当前还活动的线程对象。列表中包括了守护线程，由 current_thread() 方法创建的虚拟线程，和主线程。但不包括已经结束的线程和尚未开始运行的线程。

6. activeCount()/active_count

其中 active_count = activeCount() ，是同样功能的两个函数。

返回当前还活动着的线程的数量，这个数量等于 enumerate() 方法获得的列表的长度。

7. currentThread()/current_thread

current_thread = currentThread()

返回当前线程对象， 对应于被调用线程的控制者。 如果控制线程不是由 threading 模块创建的，则返回一个虚拟线程对象，该对象是实现了有限的功能。 8. setprofile(func)

为所有的由 threading 模块创建的线程设置一个配置函数， 这个函数通过 sys.setprofile() 实现，在线程调用 run 方法之前调用。

9. settrace(func)

为所有的由 threading 模块创建的线程设置一个跟踪函数，这个函数通过 sys.settrace() 实现，在线程调用 run 方法之前调用。

10. stack_size([size]) -> size

返回创建新线程时使用的线程堆栈大小，祥见 thread 模块中的说明。

11. Timer(*args, **kwargs)

是一个工厂函数，该函数返回一个 Timer 对象。该对象会在指定的时间过后调用某个函数。

```
t = Timer(30.0, f, args=[], kwargs={})      # 30 秒后调用函数 f
t.start()      # 开始计时
t.cancel()      # 停止计时，如果还没有开始调用目标函数的话
```

12. Semaphore(*args, **kwargs)

是一个工厂函数，返回一个信号量对象。

该方法实现了对公共资源或者临界区的访问的控制。对象内维护着一个计数器，指定可同时访问资源或者进入临界区的线程数。每次有一个线程获得信号机时，计数器 -1。若计数器为 0，其他线程就停止访问信号机，直到另一个线程释放信号机。（也就是为 0 时阻塞）

该对象内部的方法有：

release()

释放信号量，每调用一次 release()，计数器加 1. 信号量类的 release()方法增加计数器的值并且唤醒其他线程。

acquire()

每调用一次 acquire()，计数器减 1。

计算器的总值为：初始值（默认为 1）+ release()调用次数 - acquire()调用次数。

这个方法多实现对线程数量的控制

13. BoundedSemaphore(*args, **kwargs)

“有限” (bounded) 信号量类，可以确保 release()方法的调用次数不能超过给定的初始信号量数值 (value 参数)。

复制代码

```
def printthread(n):
    # 进程函数
    print n, "--> 进程创建 ", time.ctime()
    time.sleep(3)
    print n, "--> 进程结束 ", time.ctime()
    sem.release()


if __name__ =='__main__':
    maxThread = 2
    sem = threading.BoundedSemaphore(maxThread)        # 限制最大的线程数为 2
    for a in range(4):
        sem.acquire()
        threading.Thread(target=printthread, args=(a,)).start()    # 不做限制的或会创建 4个线程
    print "All thread has create,Wait for all thread exit."
    for a in range(maxThread):
        sem.acquire()    # 在子线程完成之前阻塞主线程
    print "All thread exit"
```