12.2 判断线程是否已经启动¶

问题¶

你已经启动了一个线程，但是你想知道它是不是真的已经开始运行了。

解决方案¶

线程的一个关键特性是每个线程都是独立运行且状态不可预测。如果程序中的其他线程需要通过判断某个线程的状态来确定自己下一步的操作，这时线程同步问题就会变得非常棘手。为了解决这些问题，我们需要使用 `threading` 库中的 `Event` 对象。 `Event` 对象包含一个可由线程设置的信号标志，它允许线程等待某些事件的发生。在初始情况下，event 对象中的信号标志被设置为假。如果有线程等待一个 event 对象，而这个 event 对象的标志为假，那么这个线程将会被一直阻塞直至该标志为真。一个线程如果将一个 event 对象的信号标志设置为真，它将唤醒所有等待这个 event 对象的线程。如果一个线程等待一个已经被设置为真的 event 对象，那么它将忽略这个事件，继续执行。 下边的代码展示了如何使用 `Event` 来协调线程的启动：

```
from threading import Thread, Event
import time

# Code to execute in an independent thread
def countdown(n, started_evt):
    print('countdown starting')
    started_evt.set()
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create the event object that will be used to signal startup
started_evt = Event()

# Launch the thread and pass the startup event
print('Launching countdown')
t = Thread(target=countdown, args=(10,started_evt))
t.start()

# Wait for the thread to start
started_evt.wait()
print('countdown is running')
```

当你执行这段代码，“countdown is running” 总是显示在 “countdown starting” 之后显示。这是由于使用 event 来协调线程，使得主线程要等到 `countdown()` 函数输出启动信息后，才能继续执行。

讨论¶

event 对象最好单次使用，就是说，你创建一个 event 对象，让某个线程等待这个对象，一旦这个对象被设置为真，你就应该丢弃它。尽管可以通过 `clear()` 方法来重置 event 对象，但是很难确保安全地清理 event 对象并对它重新赋值。很可能会发生错过事件、死锁或者其他问题（特别是，你无法保证重置 event 对象的代码会在线程再次等待这个 event 对象之前执行）。如果一个线程需要不停地重复使用 event 对象，你最好使用 `Condition` 对象来代替。下面的代码使用 `Condition` 对象实现了一个周期定时器，每当定时器超时的时候，其他线程都可以监测到：

```
import threading
import time

class PeriodicTimer:
    def __init__(self, interval):
        self._interval = interval
        self._flag = 0
        self._cv = threading.Condition()

    def start(self):
        t = threading.Thread(target=self.run)
        t.daemon = True

        t.start()

    def run(self):
        '''
        Run the timer and notify waiting threads after each interval
        '''
        while True:
            time.sleep(self._interval)
            with self._cv:
                self._flag ^= 1
                self._cv.notify_all()

    def wait_for_tick(self):
        '''
        Wait for the next tick of the timer
        '''
        with self._cv:
            last_flag = self._flag
            while last_flag == self._flag:
                self._cv.wait()

# Example use of the timer
ptimer = PeriodicTimer(5)
ptimer.start()

# Two threads that synchronize on the timer
def countdown(nticks):
    while nticks > 0:
        ptimer.wait_for_tick()
        print('T-minus', nticks)
        nticks -= 1

def countup(last):
    n = 0
    while n < last:
        ptimer.wait_for_tick()
        print('Counting', n)
        n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()
```

event对象的一个重要特点是当它被设置为真时会唤醒所有等待它的线程。如果你只想唤醒单个线程，最好是使用信号量或者 `Condition` 对象来替代。考虑一下这段使用信号量实现的代码：

```
# Worker thread
def worker(n, sema):
    # Wait to be signaled
    sema.acquire()

    # Do some work
    print('Working', n)

# Create some threads
sema = threading.Semaphore(0)
nworkers = 10
for n in range(nworkers):
    t = threading.Thread(target=worker, args=(n, sema,))
    t.start()
```

运行上边的代码将会启动一个线程池，但是并没有什么事情发生。这是因为所有的线程都在等待获取信号量。每次信号量被释放，只有一个线程会被唤醒并执行，示例如下：

```
>>> sema.release()
Working 0
>>> sema.release()
Working 1
>>>
```

编写涉及到大量的线程间同步问题的代码会让你痛不欲生。比较合适的方式是使用队列来进行线程间通信或者每个把线程当作一个Actor，利用Actor模型来控制并发。下一节将会介绍到队列，而Actor模型将在12.10节介绍。