12.1 启动与停止线程¶

问题¶

你要为需要并发执行的代码创建/销毁线程

解决方案¶

`threading` 库可以在单独的线程中执行任何的在 Python 中可以调用的对象。你可以创建一个 `Thread` 对象并将你要执行的对象以 target 参数的形式提供给该对象。 下面是一个简单的例子：

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

# Create and launch a thread
from threading import Thread
t = Thread(target=countdown, args=(10,))
t.start()
```

当你创建好一个线程对象后，该对象并不会立即执行，除非你调用它的 `start()` 方法（当你调用 `start()` 方法时，它会调用你传递进来的函数，并把你传递进来的参数传递给该函数）。Python中的线程会在一个单独的系统级线程中执行（比如说一个 POSIX 线程或者一个 Windows 线程），这些线程将由操作系统来全权管理。线程一旦启动，将独立执行直到目标函数返回。你可以查询一个线程对象的状态，看它是否还在执行：

```
if t.is_alive():
    print('Still running')
else:
    print('Completed')
```

你也可以将一个线程加入到当前线程，并等待它终止：

```
t.join()
```

Python解释器直到所有线程都终止前仍保持运行。对于需要长时间运行的线程或者需要一直运行的后台任务，你应当考虑使用后台线程。 例如：

```
t = Thread(target=countdown, args=(10,), daemon=True)
t.start()
```

后台线程无法等待，不过，这些线程会在主线程终止时自动销毁。 除了如上所示的两个操作，并没有太多可以对线程做的事情。你无法结束一个线程，无法给它发送信号，无法调整它的调度，也无法执行其他高级操作。如果需要这些特性，你需要自己添加。比如说，如果你需要终止线程，那么这个线程必须通过编程在某个特定点轮询来退出。你可以像下边这样把线程放入一个类中：

```
class CountdownTask:
    def __init__(self):
        self._running = True
```

```
    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running and n > 0:
            print('T-minus', n)
            n -= 1
            time.sleep(5)

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
t.start()
c.terminate() # Signal termination
t.join()      # Wait for actual termination (if needed)
```

如果线程执行一些像I/O这样的阻塞操作，那么通过轮询来终止线程将使得线程之间的协调变得非常棘手。比如，如果一个线程一直阻塞在一个I/O操作上，它就永远无法返回，也就无法检查自己是否已经被结束了。要正确处理这些问题，你需要利用超时循环来小心操作线程。 例子如下：

```
class IOTask:
    def terminate(self):
        self._running = False

    def run(self, sock):
        # sock is a socket
        sock.settimeout(5)      # Set timeout period
        while self._running:
            # Perform a blocking I/O operation w/ timeout
            try:
                data = sock.recv(8192)
                break
            except socket.timeout:
                continue
            # Continued processing
            ...
        # Terminated
        return
```

讨论¶

由于全局解释锁（GIL）的原因，Python 的线程被限制到同一时刻只允许一个线程执行这样一个执行模型。所以，Python 的线程更适用于处理I/O和其他需要并发执行的阻塞操作（比如等待I/O、等待从数据库获取数据等等），而不是需要多处理器并行的计算密集型任务。

有时你会看到下边这种通过继承 `Thread` 类来实现的线程：

```
from threading import Thread

class CountdownThread(Thread):
    def __init__(self, n):
        super().__init__()
        self.n = n
    def run(self):
        while self.n > 0:
```

```
        print('T-minus', self.n)
        self.n -= 1
        time.sleep(5)

c = CountdownThread(5)
c.start()
```

尽管这样也可以工作，但这使得你的代码依赖于 `threading` 库，所以你的这些代码只能在线程上下文中使用。上文所写的那些代码、函数都是与 `threading` 库无关的，这样就使得这些代码可以被用在其他的上下文中，可能与线程有关，也可能与线程无关。比如，你可以通过 `multiprocessing` 模块在一个单独的进程中执行你的代码：

```
import multiprocessing
c = CountdownTask(5)
p = multiprocessing.Process(target=c.run)
p.start()
```

再次重申，这段代码仅适用于 CountdownTask 类是以独立于实际的并发手段（多线程、多进程等等）实现的情况。