7.11 内联回调函数¶

问题¶

当你编写使用回调函数的代码的时候，担心很多小函数的扩张可能会弄乱程序控制流。 你希望找到某个方法来让代码看上去更像是一个普通的执行序列。

解决方案¶

通过使用生成器和协程可以使得回调函数内联在某个函数中。 为了演示说明，假设你有如下所示的一个执行某种计算任务然后调用一个回调函数的函数(参考7.10小节)：

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

接下来让我们看一下下面的代码，它包含了一个 `Async` 类和一个 `inlined_async` 装饰器：

```
from queue import Queue
from functools import wraps

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

def inlined_async(func):
    @wraps(func)
    def wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while True:
            result = result_queue.get()
            try:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue.put)
            except StopIteration:
                break
    return wrapper
```

这两个代码片段允许你使用 `yield` 语句内联回调步骤。比如：

```
def add(x, y):
    return x + y

@inlined_async
def test():
    r = yield Async(add, (2, 3))
    print(r)
    r = yield Async(add, ('hello', 'world'))
    print(r)
    for n in range(10):
        r = yield Async(add, (n, n))
        print(r)
    print('Goodbye')
```

如果你调用 `test()` ，你会得到类似如下的输出：

```
5
helloworld
0
```

```
2
4
6
8
10
12
14
16
18
Goodbye
```

你会发现，除了那个特别的装饰器和 `yield` 语句外，其他地方并没有出现任何的回调函数(其实是在后台定义的)。

讨论¶

本小节会实实在在的测试你关于回调函数、生成器和控制流的知识。

首先，在需要使用到回调的代码中，关键点在于当前计算工作会挂起并在将来的某个时候重启(比如异步执行)。 当计算重启时，回调函数被调用来继续处理结果。`apply_async()` 函数演示了执行回调的实际逻辑， 尽管实际情况中它可能会更加复杂(包括线程、进程、事件处理器等等)。

计算的暂停与重启思路跟生成器函数的执行模型不谋而合。 具体来讲，`yield` 操作会使一个生成器函数产生一个值并暂停。 接下来调用生成器的 `__next__()` 或 `send()` 方法又会让它从暂停处继续执行。

根据这个思路，这一小节的核心就在 `inline_async()` 装饰器函数中了。 关键点就是，装饰器会逐步遍历生成器函数的所有 `yield` 语句，每一次一个。 为了这样做，刚开始的时候创建了一个 `result` 队列并向里面放入一个 `None` 值。 然后开始一个循环操作，从队列中取出结果值并发送给生成器，它会持续到下一个 `yield` 语句， 在这里一个 `Async` 的实例被接受到。然后循环开始检查函数和参数，并开始进行异步计算 `apply_async()` 。 然而，这个计算有个最诡异部分是它并没有使用一个普通的回调函数，而是用队列的 `put()` 方法来回调。

这时候，是时候详细解释下到底发生了什么了。主循环立即返回顶部并在队列上执行 `get()` 操作。 如果数据存在，它一定是 `put()` 回调存放的结果。如果没有数据，那么先暂停操作并等待结果的到来。 这个具体怎样实现是由 `apply_async()` 函数来决定的。 如果你不相信会有这么神奇的事情，你可以使用 `multiprocessing` 库来试一下， 在单独的进程中执行异步计算操作，如下所示：

```
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Run the test function
    test()
```

实际上你会发现这个真的就是这样的，但是要解释清楚具体的控制流得需要点时间了。

将复杂的控制流隐藏到生成器函数背后的例子在标准库和第三方包中都能看到。 比如，在 `contextlib` 中的 `@contextmanager` 装饰器使用了一个令人费解的技巧， 通过一个 `yield` 语句将进入和离开上下文管理器粘合在一起。另外非常流行的 `Twisted` 包中也包含了非常类似的内联回调。