7.10 带额外状态信息的回调函数¶

问题¶

你的代码中需要依赖到回调函数的使用(比如事件处理器、等待后台任务完成后的回调等)， 并且你还需要让回调函数拥有额外的状态值，以便在它的内部使用到。

解决方案¶

这一小节主要讨论的是那些出现在很多函数库和框架中的回调函数的使用——特别是跟异步处理有关的。 为了演示与测试，我们先定义如下一个需要调用回调函数的函数：

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

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

实际上，这段代码可以做任何更高级的处理，包括线程、进程和定时器，但是这些都不是我们要关心的。 我们仅仅只需要关注回调函数的调用。下面是一个演示怎样使用上述代码的例子：

```
>>> def print_result(result):
...     print('Got:', result)
...
>>> def add(x, y):
...     return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Got: helloworld
>>>
```

注意到 `print_result()` 函数仅仅只接受一个参数 `result` 。不能再传入其他信息。 而当你想让回调函数访问其他变量或者特定环境的变量值的时候就会遇到麻烦。

为了让回调函数访问外部信息，一种方法是使用一个绑定方法来代替一个简单函数。 比如，下面这个类会保存一个内部序列号，每次接收到一个 `result` 的时候序列号加1：

```
class ResultHandler:

    def __init__(self):
        self.sequence = 0

    def handler(self, result):
        self.sequence += 1
        print('[{}] Got: {}'.format(self.sequence, result))
```

使用这个类的时候，你先创建一个类的实例，然后用它的 `handler()` 绑定方法来做为回调函数：

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
```

```
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Got: helloworld
>>>
```

第二种方式，作为类的替代，可以使用一个闭包捕获状态值，例如：

```
def make_handler():
    sequence = 0
    def handler(result):
        nonlocal sequence
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
    return handler
```

下面是使用闭包方式的一个例子：

```
>>> handler = make_handler()
>>> apply_async(add, (2, 3), callback=handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler)
[2] Got: helloworld
>>>
```

还有另外一个更高级的方法，可以使用协程来完成同样的事情：

```
def make_handler():
    sequence = 0
    while True:
        result = yield
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
```

对于协程，你需要使用它的 `send()` 方法作为回调函数，如下所示：

```
>>> handler = make_handler()
>>> next(handler) # Advance to the yield
>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

讨论¶

基于回调函数的软件通常都有可能变得非常复杂。一部分原因是回调函数通常会跟请求执行代码断开。 因此，请求执行和处理结果之间的执行环境实际上已经丢失了。如果你想让回调函数连续执行多步操作， 那你就必须去解决如何保存和恢复相关的状态信息了。

至少有两种主要方式来捕获和保存状态信息，你可以在一个对象实例(通过一个绑定方法)或者在一个闭包中保存它。 两种方式相比，闭包或许是更加轻量级和自然一点，因为它们可以很简单的通过函数来构造。 它们还能自动捕获所有被使用到的变量。因此，你无需去担心如何去存储额外的状态信息(代码中自动判定)。

如果使用闭包，你需要注意对那些可修改变量的操作。在上面的方案中， `nonlocal` 声明语句用来指示接下来的变量会在

回调函数中被修改。如果没有这个声明，代码会报错。

而使用一个协程来作为一个回调函数就更有趣了，它跟闭包方法密切相关。 某种意义上来讲，它显得更加简洁，因为总共就一个函数而已。 并且，你可以很自由的修改变量而无需去使用 `nonlocal` 声明。 这种方式唯一缺点就是相对于其他Python技术而言或许比较难以理解。 另外还有一些比较难懂的部分，比如使用之前需要调用 `next()` ，实际使用时这个步骤很容易被忘记。 尽管如此，协程还有其他用处，比如作为一个内联回调函数的定义(下一节会讲到)。

如果你仅仅只需要给回调函数传递额外的值的话，还有一种使用 `partial()` 的方式也很有用。 在没有使用 `partial()` 的时候，你可能经常看到下面这种使用lambda表达式的复杂代码：

```
>>> apply_async(add, (2, 3), callback=lambda r: handler(r, seq))
[1] Got: 5
>>>
```

可以参考7.8小节的几个示例，教你如何使用 `partial()` 来更改参数签名来简化上述代码。