14.2 在单元测试中给对象打补丁¶

问题¶

你写的单元测试中需要给指定的对象打补丁， 用来断言它们在测试中的期望行为（比如，断言被调用时的参数个数，访问指定的属性等）。

解决方案¶

`unittest.mock.patch()` 函数可被用来解决这个问题。 `patch()` 还可被用作一个装饰器、上下文管理器或单独使用，尽管并不常见。 例如，下面是一个将它当做装饰器使用的例子：

```
from unittest.mock import patch
import example

@patch('example.func')
def test1(x, mock_func):
    example.func(x)       # Uses patched example.func
    mock_func.assert_called_with(x)
```

它还可以被当做一个上下文管理器：

```
with patch('example.func') as mock_func:
    example.func(x)     # Uses patched example.func
    mock_func.assert_called_with(x)
```

最后，你还可以手动的使用它打补丁：

```
p = patch('example.func')
mock_func = p.start()
example.func(x)
mock_func.assert_called_with(x)
p.stop()
```

如果可能的话，你能够叠加装饰器和上下文管理器来给多个对象打补丁。例如：

```
@patch('example.func1')
@patch('example.func2')
@patch('example.func3')
def test1(mock1, mock2, mock3):
    ...

def test2():
    with patch('example.patch1') as mock1, \
         patch('example.patch2') as mock2, \
         patch('example.patch3') as mock3:
    ...
```

讨论¶

`patch()` 接受一个已存在对象的全路径名，将其替换为一个新的值。 原来的值会在装饰器函数或上下文管理器完成后自动恢复回来。 默认情况下，所有值会被 `MagicMock` 实例替代。例如：

```
>>> x = 42
>>> with patch('__main__.x'):
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

不过，你可以通过给 `patch()` 提供第二个参数来将值替换成任何你想要的：

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

被用来作为替换值的 `MagicMock` 实例能够模拟可调用对象和实例。 他们记录对象的使用信息并允许你执行断言检查，例如：

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>

>>> m.upper.return_value = 'HELLO'
>>> m.upper('hello')
'HELLO'
>>> assert m.upper.called

>>> m.split.return_value = ['hello', 'world']
>>> m.split('hello world')
['hello', 'world']
>>> m.split.assert_called_with('hello world')
>>>

>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>
```

一般来讲，这些操作会在一个单元测试中完成。例如，假设你已经有了像下面这样的函数：

```
# example.py
from urllib.request import urlopen
import csv

def dowprices():
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=sl1')
    lines = (line.decode('utf-8') for line in u)
    rows = (row for row in csv.reader(lines) if len(row) == 2)
    prices = { name:float(price) for name, price in rows }
    return prices
```

正常来讲，这个函数会使用 `urlopen()` 从Web上面获取数据并解析它。 在单元测试中，你可以给它一个预先定义好的数据集。下面是使用补丁操作的例子:

```
import unittest
from unittest.mock import patch
import io
import example

sample_data = io.BytesIO(b'''\
"IBM",91.1\r
"AA",13.25\r
"MSFT",27.72\r
\r
''')

class Tests(unittest.TestCase):
    @patch('example.urlopen', return_value=sample_data)
    def test_dowprices(self, mock_urlopen):
        p = example.dowprices()
        self.assertTrue(mock_urlopen.called)
        self.assertEqual(p,
                         {'IBM': 91.1,
                          'AA': 13.25,
                          'MSFT' : 27.72})

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

本例中，位于 `example` 模块中的 `urlopen()` 函数被一个模拟对象替代， 该对象会返回一个包含测试数据的 `BytelO()`.

还有一点，在打补丁时我们使用了 `example.urlopen` 来代替 `urllib.request.urlopen` 。 当你创建补丁的时候，你必须使用它们在测试代码中的名称。 由于测试代码使用了 `from urllib.request import urlopen` ,那么 `dowprices()` 函数 中使用的 `urlopen()` 函数实际上就位于 `example` 模块了。

本节实际上只是对 `unittest.mock` 模块的一次浅尝辄止。 更多更高级的特性，请参考 官方文档