Python单元测试框架

作者: Steve Purcell, <stephen_purcell at yahoo dot com>

翻译: Heiz, <heiz dot yuan at gmail dot com>

项目网站: http://pyunit.sourceforge.net/

目录

概况

Python单元测试框架（The Python unit testing framework），简称为PyUnit， 是Kent Beck和Erich Gamma这两位聪明的家伙所设计的 JUnit 的Python版本。 而JUnit又是Kent设计的Smalltalk测试框架的Java版本。它们都是各自语言的标准测试框架。

此文档仅阐述针对Python的单元测试PyUnit的设计与使用。如需单元测试框架基本设计的背景 信息，请查阅Kent的原始文章"Simple Smalltalk Testing: With Patterns"。

自从 Python2.1 版本后，PyUnit成为 Python标准库的一部分。

以下内容默认您已经了解Python。我觉得Python 非常简单易学而且让人欲罢不能。

系统要求

PyUnit可以在Python 1.5.2及更高版本上运行。

作者已经在Linux（Redhat 6.0和6.1以及Debian Potato）和Python 1.5.2, 2.0和2.1上对PyUnit 进行了测试。而且PyUnit已知可以在其它操作系统平台上工作，如Windows和Mac。如果您在 任何系统平台或Python版本中遇到麻烦，请让我知道。

如需了解在JPython和Jython中使用PyUnit的细节，请阅读 在JPython和Jython中使用PyUnit 部分。

使用PyUnit构建自己的测试

安装

编写测试所需的类可以在“unittest”模块中找到。此模块是Python 2.1和更高版本的标准 库的一部分。如果你在使用更早版本的Python，你应该从单独的PyUnit发布中获得此模块。

为使此模块能在你的代码中正常工作，你只需确保包含“unittest.py”文件的目录 在你的Python搜索路径中。为此，你可以修改环境变量“$PYTHONPATH”或将此文件 放入当前Python搜索路径中的某一个目录中，比如在Redhat Linux系统中的 /usr/lib/python1.5/site-packages目录。

注意，你只有完成此项工作才能运行PyUnit所自带的例子，除非你将“unittest.py”复制到 例子目录。

测试用例介绍

单元测试是由一些测试用例（Test Cases）构建组成的。测试用例是被设置用来检测正确性的 单独的场景。在PyUnit中，unittest模块中的TestCase 类代表测试用例。

TestCase类的实例是可以完全运行测试方法和可选的设置 （set-up）以及清除（tidy-up）代码的对象。

TestCase实例的测试代码必须是自包含的，换言之，它可以单独运行或与其它任意数量的测试用例共同运行。

创建一个简单测试用例

通过覆盖runTest方法即可得到最简单的测试用例子类以运行一些测试代码：

```
import unittest


class DefaultWidgetSizeTestCase(unittest.TestCase):
    def runTest(self):
        widget = Widget("The widget")
        assert widget.size() == (50,50), 'incorrect default size'
```

注意：为进行测试，我们只是使用了Python内建的“assert”语句。如果在测试用例运行时断言（assertion）为假，AssertionError异常会被抛出，并且测试框架会认为测试用例失败。其它非“assert”检查所抛出的异常会被测试框架认为是“errors”。（参见"更多关于测试条件"）

运行测试用例的方法会在后面介绍。现在我们只是通过调用无参数的构造器（constructor）来创建一个测试用例的实例：

```
testCase = DefaultWidgetSizeTestCase()
```

复用设置代码：创建固件

现在，这样的测试用例数量巨大且它们的设置需要很多重复性工作。在上面的测试用例中，如若在100个Widget测试用例的每一个子类中都创建一个“Widget”，那会导致难看的重复。

幸运的是，我们可以将这些设置代码提取出来并放置在一个叫做setUp的钩子方法（hook method）中。测试框架会在运行测试时自动调用此方法：

```
import unittest


class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")


class DefaultWidgetSizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        assert self.widget.size() == (50,50), 'incorrect default size'
```

```
class WidgetResizeTestCase(SimpleWidgetTestCase):
    def runTest(self):
        self.widget.resize(100,150)
        assert self.widget.size() == (100,150), \
                'wrong size after resize'
```

如果setUp方法在测试运行时抛出异常，框架会认为测试遇到了错误并且 runTest不会被执行。

类似的，我们也可以提供一个tearDown方法来完成在runTest运行之后的清理工作：

```
import unittest

class SimpleWidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")
    def tearDown(self):
        self.widget.dispose()
        self.widget = None
```

如果setUp执行成功， 那么无论runTest是否成功，tearDown方法都将被执行。

Such a working environment for the testing code is termed a *fixture.* 这个测试代码的运行环境被称为*固件(fixture，译者注：此为暂定译法，意为固定的构件或方法)*。

包含多个测试方法的测试用例类

很多小型测试用例经常会使用相同的固件。在这个用例中，我们最终从SimpleWidgetTestCase继承产生很多仅包含一个方法的类，如 DefaultWidgetSizeTestCase。这是很耗时且不被鼓励的，因此，沿用JUnit的风格，PyUnit提供了一个更简便的方法：

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget("The widget")
    def tearDown(self):
        self.widget.dispose()
        self.widget = None
    def testDefaultSize(self):
```

```
        assert self.widget.size() == (50,50), 'incorrect default size'
    def testResize(self):
        self.widget.resize(100,150)
        assert self.widget.size() == (100,150), \
               'wrong size after resize'
```

在这个用例中，我们没有提供runTest方法，而是两个不同的测试方法。类实例将创建和销毁各自的self.widget并运行某一个test方法。当创建类实例时，我们必须通过向构造器传递方法的名称来指明哪个测试方法将被运行：

```
defaultSizeTestCase = WidgetTestCase("testDefaultSize")
resizeTestCase = WidgetTestCase("testResize")
```

将测试用例聚合成测试套件

测试用例实例可以根据它们所测试的特性组合到一起。PyUnit为此提供了一个机制叫做”测试套件“（test suite)。它由unittest模块中的TestSuite类表示：

```
widgetTestSuite = unittest.TestSuite()
widgetTestSuite.addTest(WidgetTestCase("testDefaultSize"))
widgetTestSuite.addTest(WidgetTestCase("testResize"))
```

我们稍后会看到，在每个测试模块中提供一个返回已创建测试套件的可调用对象，会是一个使测试更加便捷的好方法：

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase("testDefaultSize"))
    suite.addTest(WidgetTestCase("testResize"))
    return suite
```

甚至可写成:

```
class WidgetTestSuite(unittest.TestSuite):
    def __init__(self):
        unittest.TestSuite.__init__(self,map(WidgetTestCase,
                                              ("testDefaultSize",
                                               "testResize")))
```

(诚然，第二种方法不是为胆小者准备的)

因为创建一个包含很多相似名称的测试方法的TestCase子类是一种很常见的模式，所以unittest模块提供一个便捷方法，makeSuite，来创建一个由测试用例类内所有测试用例组成的测试套件：

```
suite = unittest.makeSuite(WidgetTestCase,'test')
```

需要注意的是，当使用makeSuite方法时，测试套件运行每个测试用例的顺序是由测试方法名根据Python内建函数cmp所排序的顺序而决定的。

嵌套测试套件

我们经常希望将一些测试套件组合在一起来一次性的测试整个系统。这很简单，因为多个TestSuite可以被加入进另一个TestSuite，就如同多个TestCase被加进一个TestSuite中一样：

```
suite1 = module1.TheTestSuite()
suite2 = module2.TheTestSuite()
alltests = unittest.TestSuite((suite1, suite2))
```

在发布的软件包中的“examples”目录中，"alltests.py”提供了使用嵌套测试套件的例子

测试代码放置位置

你可以将测试用例定义与被测试代码置于同一个模块中（例如“widget.py”），但是将测试代码放置在单独的模块中（如“widgettests.py”）会有一些优势：

测试模块可以从命令行单独执行

测试代码可以方便地从发布代码中分离
少了在缺乏充足理由的情况下为适应被测试代码而更改测试代码的诱惑
相对于被测试代码，测试代码不应该被频繁的修改
被测试代码可以更方法的进行重构
既然C语言代码的测试应该置于单独的模块，那何不保持这个一致性呢？
如果测试策略改变，也无需修改被测试源代码

交互式运行测试

我们编写测试的主要目的是运行它们并检查我们的软件是否工作正常。测试框架使用“TestRunner”类来为运行测试提供环境。最常用的TestRunner是TextTestRunner，它可以以文字方式运行测试并报告结果：

```
runner = unittest.TextTestRunner()
runner.run(widgetTestSuite)
```

TextTestRunner默认将输出发送到sys.stderr，但是你可以通过向它的构造器传递一个不同的

类似文件（file-object）对象来改变默认方式。

如需在Python解释器会话中运行测试，这样使用TextTestRunner是一个理想的方法。

从命令行运行测试

unittest模块包含一个main方法，可以方便地将测试模块转变为可以运行测试的脚本。main 使用unittest.TestLoader类来自动查找和加载模块内测试用例。

因此，如果你之前已经使用test*惯例对测试方法进行命名，那么你就可以将以下代码插入测试模块的结尾：

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

这样，当你从命令行执行你的测试模块时，其所包含的所有测试都将被运行。使用“-h”选项运行模块可以查看所有可用的选项。

如需从命令行运行任意测试，你可以将unittest模块作为脚本运行，并将所需执行的测试套件中的测试用例名称作为参数传递给此脚本：

```
% python unittest.py widgettests.WidgetTestSuite
```

or

```
% python unittest.py widgettests.makeWidgetTestSuite
```

你还可以在命令行指明特定的测试（方法）来执行。如要运行“listtests”模块中的TestCase类的子类 'ListTestCase'（参见发布软件包中的“examples”子目录），你可以执行以下命令：

```
% python unittest.py listtests.ListTestCase.testAppend
```

“testAppend”是测试用例实例将要执行的测试方法的名称。你可以执行以下代码来创建ListTestCase类实例并执行其所包含的所有“test*”测试方法：

```
% python unittest.py listtests.ListTestCase
```

在用户界面窗口运行测试

你还可以使用图形化窗口运行你的测试。它是用Tkinter编写的。在多数平台上，这个窗口工具与Python是捆绑在一起发布的。它看上去和JUnit窗口很相似。

你只需运行以下命令来使用测试运行窗口：

```
% python unittestgui.py
```

or

```
% python unittestgui.py widgettests.WidgetTestSuite
```

这里需要注意的是，所输入的测试名称必须是一个可以返回TestCase或TestSuite类实例的对象名称，不可以是事先创建好的测试名称，因为每个测试必须在每次运行是重新创建。

使用窗口测试会因为更新那些窗口而带来额外的时间开销。在我系统上，每一千个测试，它会多花七秒钟。你的消耗可能会不同。

为测试编写文档

通常当测试运行时，TestRunner将显示其名称。这个名称是由测试用例类名和所运行的测试方法名组成的。

但是如果你为测试方法提供了doc-string，则当测试运行时，doc-string的第一行将被显示出来。这为编写测试文档提供了一个很便捷的机制：

```
class WidgetTestCase(unittest.TestCase):
    def testDefaultSize(self):
        """Check that widgets are created with correct default size"""
        assert self.widget.size() == (50,50), 'incorrect default size'
```

更多关于测试条件

我之前建议过应使用Python内建断言机制来检查测试用例中的条件，而不应使用自己编写的替代品，因为assert更简单，简明且为大家所熟悉。

但是值得注意的是，如果在运行测试的同时Python优化选项被打开（生成“.pyo"字节码文件），那么assert语句将会被跳过，使得测试用例变得无用。

我为那些需要使用Python优化选项的用户编写了一个assert_方法并添加进TestCase类内。它的功能和内建的assert相同且不会被优化删除，但是使用较麻烦且所输出错误信息帮助较小：

```
def runTest(self):
    self.assert_(self.widget.size() == (100,100), "size is wrong")
```

我还在TestCase类中提供了failIf和failUnless两个方法：

```
def runTest(self):
    self.failIf(self.widget.size() <> (100,100))
```

测试方法还可以通过调用fail方法使得测试立即失败：

```
def runTest(self):
    ...
    if not hasattr(something, "blah"):
```

```
        self.fail("blah missing")
        # or just 'self.fail()'
```

测试相等性

最常用的断言是测试相等性。如果断言失败，开发者通常希望看到实际错误值。

TestCase包含一对方法assertEqual和assertNotEqual用于此目的（如果你喜欢，你还可以使用别名：failUnlessEqual 和 failIfEqual）：

```
    def testSomething(self):
        self.widget.resize(100,100)
        self.assertEqual(self.widget.size, (100,100))
```

测试异常

测试经常希望检查在某个环境中是否出现异常。如果期待的异常没有抛出，测试将失败。这很容易做到：

```
    def runTest(self):
        try:
            self.widget.resize(-1,-1)
        except ValueError:
            pass
        else:
            fail("expected a ValueError")
```

通常，预期异常源（译者注：将抛出异常的代码）是一个可调用对象；为此，TestCase有一个assertRaises方法。此方法的前两个参数是应该出现在“except”语句中的异常和可调用对象。剩余的参数是应该传递给可调用对象的参数。

```
    def runTest(self):
        self.assertRaises(ValueError, self.widget.resize, -1, -1)
```

通过PyUnit复用旧测试代码

一些用户希望将已有的测试代码不需转变为TestCase子类而直接从PyUnit中运行。

为此，PyUnit提供了一个FunctionTestCase类。这个TestCase子类可以用来包装已有测试函数 。

设置和清理函数也可以选择性地被包装。

对于以下测试函数：

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    ...
```

我们可以创建一个等同的测试用例实例：

```
testcase = unittest.FunctionTestCase(testSomething)
```

如果有附加的设置和清理方法需要由测试用例调用，可以如下操作：

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

在JPython和Jython中使用PyUnit

虽然PyUnit主要是为“C” Python所编写，你仍然可以用Jython编写PyUnit测试，来测试你的Java或Jython软件。这比用Jython编写JUnit测试更可取。PyUnit也可以正确的与Jython前期版本，Jython 1.0和1.1协同工作。

当然，Java不包含TK GUI接口，所以PyUnit的基于TKinter的GUI是不能在Jython下工作的，但是基于文本的接口是可以正常工作的。

要在Jython中使用PyUnit的文本接口，只需简单的将标准C Python库模块文件‘traceback.py', 'linecache.py', 'stat.py' 和 'getopt.py'复制到可以被JPython引用到的位置上。你可以在任何C Python发布版中找到这些文件。（这是针对C Python 1.5.x版本的标准库，可能对其它版本Python不适用）

现在你完全可以像在C Python中那样编写你的PyUnit测试了。

注意事项

断言

参见 "更多关于测试条件" 部分所述注意事项。

内存使用

当异常在测试套件运行过程中被抛出时，因此产生的追溯（traceback）对象将被保存，以使失败信息可以在测试运行结束后被格式化输出。除了简便性，这样做的另一个优点就是未来的GUI TestRunner可以在后期查看保存在追溯对象中的本地和全局变量。

一个可能的副作用就是，当运行一个失败频率很高的测试套件时，为保存所有这些追溯对象而需要的内存使用量将成为一个问题。当然，如果很多测试是失败的，内存的消耗也只是你的问题中最微不足道的一个。

使用条款

你可以依据Python所使用的自由条款来自由的使用，更改和重新发布此软件。我只要求我的名字，email地址和项目URL保留在代码和随行文档中，给予我作为原作者的尊重。

我编写此软件的初衷是为改进世界上软件质量而贡献微薄之力；我不求金钱回报。（这不是说我不欢迎赞助）

未来计划

一个关键的未来计划是将TK GUI和IDLE IDE整合在一起，欢迎加入！

除此之外，我没有要扩展此模块功能的庞大计划。我使PyUnit尽可能的简单（希望不能再简单了）因为我相信一些常用的辅助性的模块，比如日志文件比较，最好还是由测试编写者自行编写。

更新与社区

新闻，更新以及更多信息可以在项目网站获得。

欢迎各种评论，建议和错误报告；只需给我发送电子邮件或者这个非常小量的邮件列表并发表你的评论。现在有大量的PyUnit使用者，他们都有智慧与大家分享。

鸣谢

Many thanks to Guido and his disciples for the Python language. In tribute, I have written the following *haiku* (or '*pyku*', if you will):

Guido van Rossum

'Gawky Dutchman' gave birth to
Beautiful Python

I gratefully acknowledge the work of Kent Beck and Erich Gamma for their work on JUnit, which made the design of PyUnit a no-brainer.

Thanks also to Tim Voght; I discovered after I had implemented PyUnit that he had also

implemented a 'pyunit' module as part of his 'PyWiki' WikiWikiWeb clone. He graciously gave me the go-ahead to submit my version to the community at large.

Many thanks to those who have written to me with suggestions and questions. I've tried to add appropriate credits in the CHANGES file in the download package.

Particular thanks to J¨|rôme Marant, who packaged PyUnit for Debian.

相关信息

PyUnit网站

Python网站
"Simple Smalltalk Testing: With Patterns" 作者： Kent Beck
JUnit网站
XProgramming.com -- 极限编程主页
ExtremeProgramming 于 WikiWikiWeb
PyWiki -- Python WikiWikiWeb 由 Tim Voght复制

关于作者

Steve Purcell is just a programmer at heart, working independently writing, applying and teaching Open Source software.

He recently acted as Technical Director for a Web/WAP start-up, but spends most of his time architecting and coding large Java systems whilst counterproductively urging his Java-skilled colleagues to take up Python instead.