MATLAB 中文论坛技术专栏　系列

MATLAB 单元测试框架

MATLAB 中文论坛　出品

目　　录

0.1 什么是框架

从逻辑上来说，框架 (Framework)，是一个比面向对象和设计模式更加复杂的结构，但读者不用担心，虽然框架在结构上比模式要复杂，但是学习起来要比设计模式简单得多。我们这里介绍的不是关于如何设计框架，而是介绍如何利用现成的框架为工程计算服务，理解设计模式不是使用框架的前提，甚至不用理解面向对象，也可以享受框架给我们工程计算带来的便利。

设计模式教给我们的是编程的指导思想，没有现成的代码可以直接套用，模式每次的使用，都要通过重新编程来实现；而框架，是包装好的即时可以使用的代码，可以直接的反复被使用。设计模式处理的是软件程序设计中的局部的行为，而框架处理的是更大系统。模式是组成框架的基石，框架的设计和实现包含中多种模式。设计模式的应用范围很广，而框架通常限定了应用范围，比如：单元测试框架保证我们在算法开发的同时能够保证已有的程序功能不会退化，而性能测试框架保证算法性能不退化，方便的比较不同算法的性能。

0.2 基于函数的单元测试的构造

在附录**??**中介绍 inputParser 的时候，我们通过不断改进 getArea 函数对输入参数的处理方法，引入这样一个观点：一个可靠的科学工程计算项目必须有一套测试系统，才能防止开发的过程中算法退化，工程项目的推进必须在算法开发和算法测试之间不断迭代完成。在附录**??**的最后, 还根据直觉提出了一个测试系统所应该有的基本功能。在本章中，我们将学习 MATLAB 从 R2013a 开始提供的测试解决方案：MATLAB 单元测试 (MATLAB Unit Tests)。MATLAB 单元测试框架可以接受不同格式的测试文件，本书介绍两种，一种是基于函数 (Function-Based) 的，另种基于类文件 (Class-Based) 如图1所示，先介绍基于函数的单元测试。

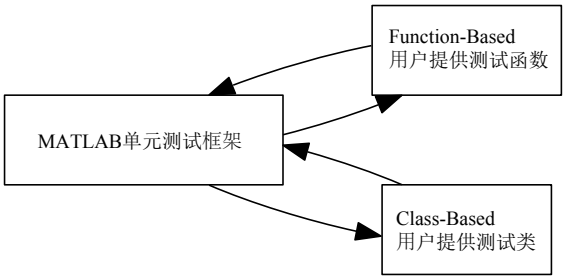


图 1 单元测试 Framework 和两个风格的单元测试

MATLAB 基于函数的单元测试构造很简单，用户通过一个主测试函数和若干局部测试函数[①] (Local Function) 来组织各个测试。而测试的运行则交给 MATLAB 的单元测试 Framework 去完成。

主测试函数和局部测试函数看上去和普通的 MATLAB 函数没有区别，如图2所示，只是命名上有一些规定而已，这些特殊的规定是为了 Framework 可以和测试函数契合而规定的。

① 也叫做测试点

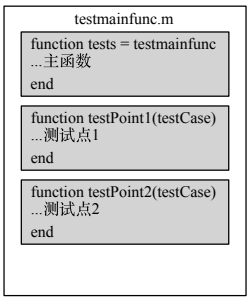


图 2　简单的主测试函数和若干局部的测试函数构成的一个单元测试

命名规则如下：主函数的名称由用户任意指定，和其他的 MATLAB 函数文件一样，该文件的名称需要和函数的名称的相同. (如果主函数的名称是`testmainfunc`, 该文件名称则是`testmainfunc.m`)。在主函数中, 必须调用一个叫做`functiontests` 的函数, 搜集该函数中的所有局部函数，产生一个包含这些局部函数的函数局部的测试矩阵并返回给 Framework，如下所示：

testmainfunc.m

```
function tests = testmainfunc
  tests = functiontests(localfunctions);    % 主测试函数中必须要有这个命令
end
...
```

其中`localfunctions` 是一个 MATLAB 函数，用来返回所有局部函数的函数句柄。

局部函数的命名必须以`test` 开头，局部函数只接受一个输入参数，即测试对象, 即下面例子中的形参 `testCase`

testmainfunc.m

```
...
function testPoint1(testCase)     % 只接受一个输入参数
    testCase.verifyEqual(.....);
end

function testPoint2(testCase)    % 只接受一个输入参数
    testCase.verifyEqual(.....);
end
...
```

其中 testCase 由单元测试 Framework 提供, 即 Framework 将自动的调用该函数，并且提供 testCase 参数。

按照规定，要运行单元测试中的所有测试，必须调用 runtests 函数

command line

```
>> runtests('testmainfunc.m')
```

下面用我们用基于函数的单元测试来给 getArea 函数的构造其单元测试。

0.3 getArea 函数的单元测试: 版本 I

首先给主测试文件起个名字叫做 testGetArea，该名字是任意的，为了便于理解名字里面通常包含 test, 并包含要测试的主要函数的名字:

testGetArea.m

```
function tests = testGetArea
  tests = functiontests(localfunctions);
end
```

在该主函数中，localfunctions 将搜集所有的局部函数，构造函数句柄数组并返回测试矩阵。这里自然会有一个问题，这个 tests 句柄数组将返回给谁，这就要了解 Framework 是如何和测试相互作用的。如图3所示，整个测试从 `runtests('testmainfunc.m')` 命令开始，命令函数，Framework 将首先调用 testGetArea 的主函数，得到所有的局部函数的函数句柄，如空心箭头线段所示，然后 Framework 再负责调用每一个测试局部函数，并且把 testCase 当做参数提供给每个局部函数, 如虚线线段所示。我们可以把 Framework 想象成一个流水线，用户只需要通过`runtests('testmainfunc.m')` 把"testmainfunc.m" 放到流水线上并且打开开关" 就可以了。它是 MATLAB 的类 `matlab.unittest.FunctionTestCase` 的对象。

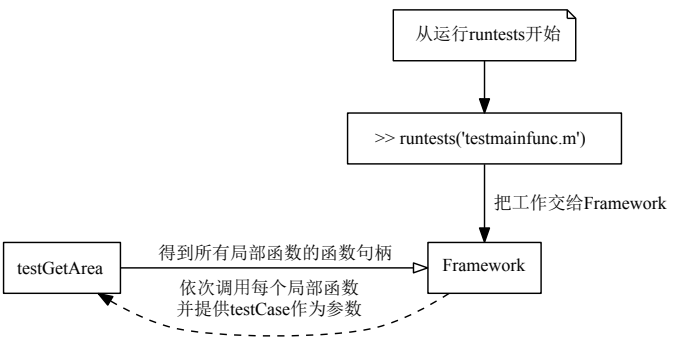


图 3 单元测试 Framework 和测试函数的相互作用

返回的 testCase 是类 `matlab.unittest.FunctionTestCase` 的对象，有很多成员验证方法可以提供给用户调用，回忆我们的第一版的 getArea 函数如下, 要求函数接受两个参数，并且都是数值类型:

第一版的 getArea 函数

```
function a = getArea(wd,ht)

p = inputParser;

p.addRequired('width', @isnumeric); % 检查输入必须是数值型的
p.addRequired('height',@isnumeric);

p.parse(wd,ht);
```

```
  a = p.Results.width*p.Results.height;  % 从 Results 处取结果
end
```

我们先给这个 getArea 写第一个测试点，确保测试 getArea 函数在接受两个参数的时候，能给出正确的答案

testGetArea.m

```
function tests = testGetArea
    tests = functiontests(localfunctions);
end
% 添加了第一个测试点
function testTwoInputs(testCase)
    testCase.verifyTrue(getArea(10,22)==220,'!=220');
    testCase.verifyTrue(getArea(3,4)==12,'!=12');
end
```

我们给 testGetArea.m 添加一个局部函数叫做 testTwoInputs, 按照规定，该局部函数的名字要以 test 开头，后面的名字要能够尽量反应该测试点的实际测试的内容。verifyTrue 是一个 testCase 对象所支持的方法，它用来验证其第一个参数，作为一个表达式，是否为真。verifyTrue 的第二个参数接受字符串，在测试失败时提供诊断提示。

一个很常见的问题是: getArea 是一个极其简单的函数，内部的工作就是把两个输入相乘，在这里验证getArea(10,22) == 220 真的有必要吗? 请读者记住这个问题，它是单元测试的精要之一。

下面我们来运行这个测试:

command line

```
>> results =runtests('testGetArea')
Running testGetArea
.
Done testGetArea
----------
results =    % 测试返回 matlab.unittest.TestResult 对象
  TestResult with properties:
            Name: 'testGetArea/testTwoInputs'
          Passed: 1
          Failed: 0
      Incomplete: 0
        Duration: 0.0018
Totals:
   1 Passed, 0 Failed, 0 Incomplete.
```

```
   0.0018203 seconds testing time.
```

测试返回一个 matlab.unittest.TestResult 对象，其中包括运行测试的结果，不出意料我们的函数通过了这轮简单的测试。

如果函数没有通过测试，比如我们故意要验证一个错误的结果： getArea(10,22) ==0

testGetArea.m

```
function tests = testGetArea
    tests = functiontests(localfunctions);
end
function testTwoInputs(testCase)
    testCase.verifyTrue(getArea(10,22)==0,'Just A Test'); % 故意让验证失败
end
```

Framework 将给出详尽的错误报告, 其中 Test Diagnostic 栏目中报告的就是 verifyTrue 函数中的第二个参数所提供的诊断信息。

command line

```
>> results =runtests('testGetArea')
Running testGetArea
================================================================================
Verification failed in testGetArea/testTwoInputs.  % 验证失败
    ----------------
    Test Diagnostic:           % 诊断信息
    ----------------
    Just A Test


    ---------------------
    Framework Diagnostic:
    ---------------------
    verifyTrue failed.       % 验证函数 verifyTrue 出错
    --> The value must evaluate to "true".
                             % 验证的表达式 getArea(10,22)==0 的值应该为 true
    Actual logical:
             0               % 表达式的实际值为 false
    ------------------
    Stack Information:
    ------------------
    In testGetArea.m (testTwoInputs) at 6 % 测试点 testTwoPoints 出错
================================================================================
.
```

```
Done testGetArea

---------
Failure Summary:                % 测试简报
     Name                       Failed  Incomplete  Reason(s)
    ========================================================================
     testGetArea/testTwoInputs    X                 Failed by verification.
                              % 出错的测试点名称
results =
  TestResult with properties:

          Name: 'testGetArea/testTwoInputs'
        Passed: 0               % 零个测试点通过
        Failed: 1               % 一个测试点出错
    Incomplete: 0
      Duration: 0.0342
Totals:
   0 Passed, 1 Failed, 0 Incomplete.
   0.03422 seconds testing time.
```

我们再添加一个负面测试，回忆第一版的函数 getArea 不支持单个参数，如下：

command line

```
>> getArea(10)          % 如预期报错 调用少一个参数
Error using getArea
Not enough input arguments.
>> [a b] = lasterr      % 调用 lasterr 得到 error ID
a =
Error using getArea1 (line 6)
Not enough input arguments.
b =
MATLAB:minrhs
```

我们还利用 lasterr 函数得到了这个错误的 Error ID, 这个 Error ID 将在负面测试中用到。

下面是这个负面测试，验证在只有一个输入的情况下，getArea 函数能够如预期报错。我们给测试添加一个新的测试点，叫做 `testTwoInputsInvalid`

testGetArea.m

```
function tests = testGetArea
   tests = functiontests(localfunctions);
end
```

```
function testTwoInputs(testCase)
    testCase.verifyTrue(getArea1(10,22)==220,'!=220');
    testCase.verifyTrue(getArea1(3,4)==12,'!=12');
end
% 添加了第 2 个测试点
function testTwoInputsInvalid(testCase)
    testCase.verifyError(@()getArea1(10),'MATLAB:minrhs');
end
```

在testTwoInputsInvalid 中，我们使用了测试对象的 verifyError 成员函数，它的第一个参数是函数句柄，即要执行的语言（会出错的语句），第二个参数是要验证的 MATLAB 错误的 Error ID, 就是我们前面用 lasterr 函数得到的信息。verifyError 内部还有 try 和 catch, 可以运行函数句柄，捕捉到错误，并且把 Error ID 和第二个参数做比较。

再举一个例子，我们先在 getArea 函数中规定所有的输入必须是数值类型，所以如果输入的是字符串，getArea 将报错，先再命令行中实验一下，以便得到 Error ID:

在命令行中得到 Error ID

```
>> getArea1('10',22)
Error using getArea1 (line 6)
The value of 'width' is invalid. It must satisfy the function: isnumeric.
>> [a b] = lasterr
a =
Error using getArea1 (line 6)
The value of 'width' is invalid. It must satisfy the function: isnumeric.
b =
MATLAB:InputParser:ArgumentFailedValidation     % 这个 Error ID 是我们需要的
```

然后再把这个负面测试添加到 testGetArea 中去。

testGetArea.m

```
function tests = testGetArea
tests = functiontests(localfunctions);
end

function testTwoInputs(testCase)
    testCase.verifyTrue(getArea1(10,22)==220,'!=220');
    testCase.verifyTrue(getArea1(3,4)==12,'!=12');
end

function testTwoInputsInvalid(testCase)
    testCase.verifyError(@()getArea1(10),'MATLAB:minrhs');
    testCase.verifyError(@()getArea1('10',22),...    % 新增的 test
```

```
                        'MATLAB:InputParser:ArgumentFailedValidation')
end
```

运行一遍，一个正面测试，一个负面测试都全部通过。

command line

```
>> runtests('testGetArea')
Running testGetArea
..
Done testGetArea

---------
ans =
  1x2 TestResult array with properties:
    Name
    Passed
    Failed
    Incomplete
    Duration
Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.0094501 seconds testing time.
```

0.4 getArea 函数的单元测试: 版本 II 和版本 III

回忆 getArea 函数的开发[①]，第二个版本我们给 getArea 添加了可以处理单个参数的能力，并且把 inputParser 和 validateAttributes 联合起来使用。新的函数在原来的基础上可以应付如下的新的情况

command line

```
>> getArea(10)    % 正确处理了单个参数的情况
ans =
   100

>> getArea(10,0)  % 如预期检查出第二个参数的错误，并给出提示
Error using getArea (line 37)
The value of 'height' is invalid. Expected input number 2, height, to be nonzero.

>> getArea(0,22)  % 如预期检查出第一个参数的错误，并给出提示
Error using getArea (line 37)
The value of 'width' is invalid. Expected input number 1, width, to be nonzero.
```

① 参见附录??

在开发完这第二个版本的函数之后，我们首先运行了一下已经有的 testGetArea 测试，发现之前添加的一个测试点，验证函数在接受一个参数时会报错的情况已不再适用，因为我们已经开始支持单参数的功能了，所以要去掉它，随着程序算法的不断开发，修改或删除已有的测试是很常见的

testGetArea.m

```
  ...
% testCase.verifyError(@()getArea1(10),'MATLAB:minrhs');  需要去掉这个测试
  ...
```

去掉不再适用的测试之后，我们继续给单元测试添加新的测试点，首先添加一个 Postive 测试点，确保 getArea 函数接受单一参数计算结果正确

确保单一参数计算正确

```
function tests = testGetArea
   ... 从略

 function testOneInput(testCase)
      testCase.verifyTrue(getArea2(10) ==100,'!=100');
      testCase.verifyTrue(getArea2(22) ==484,'!=484');
 end
```

再添加一个 Negative 测试点，确保 getArea 函数会处理输入是零的情况

保证不接受零输入

```
function tests = testGetArea
   ... 从略

 function testTwoInputsZero(testCase)
     testCase.verifyError(@()getArea(10,0),'MATLAB:expectedNonZero');
     testCase.verifyError(@()getArea(0,22),'MATLAB:expectedNonZero');
 end
```

然后调用

command line

```
>> runtests('testGetArea')
...
```

每次运行这个命令，会运行之前所有的测试点和新的测试点，这也就保证了对新添加的算法没有破坏以前有的功能。我们前面问了一个问题：验证`getArea(10,22) == 220` 真的有必要吗。其必要性之一，也是单元测试功能之一：即这个验证其实是对 getArea 能正确处理两个参数的能力的一个历史记录。因为我们在不停的算法开发中，很难保证不会偶然破坏一些以前的什么功能，但是只要有这条测试在, 无论我们对 getArea 函数做怎样翻天复地的修改，

只要一运行测试，都会验证这条历史记录，确保我们没有损坏已经有的功能，换句话说，新的函数是向后兼容的。对于一个科学工程计算系统来说，一个函数会被用在很多不同的地方，向后兼容让我们放心的继续开发新的功能，而不用担心是否要去检查所有其它使用该函数的地方。所以从这个角度说：单元测试是算法开发的堡垒，算法的开发应该以单元测试来步步为营，在确保算法没有退化的基础上开发新的内容。话说回来，为了让这个版本的 getArea 能够顺利运行，我们确实去掉了一个对单一参数报错的测试，因为函数开始支持这种功能了，这种做法和我们说以单元测试步步为营并不矛盾，如果新的算法导致旧的测试失败，我们要根据实际情况，酌情决定是修改算法还是修改测试。

在 getArea 的第三个版本中，我们给函数添加了两个可选的参数:shape 和 units, 并且它们的顺序可以相互颠倒的。新的函数可以应付如下的情况：

command line

```
>> getArea(10,22,'shape','square','units','m') % 接受两对 name-value pair
ans =            %--name  value  --name   value
     area: 220
    shape: 'square'
    units: 'm'

>> getArea(10,22,'units','m','shape','square')  % 变化了参数的位置
ans =
     area: 220
    shape: 'square'
    units: 'm'


>> getArea(10,22,'units','m')                   % 仅仅提供 unit 参数
ans =
     area: 220
    shape: 'rectangle'
    units: 'm'
```

为其添加的新的测试点如下：

testGetArea

```
function tests = testGetArea
  ... 从略

 function testFourInputs(testCase) % 记录可以支持四个参数的情况
    actStruct = getArea5(10,22,'shape','square','unit','m');
    expStruct = struct('area',220,'shape','square','units','m');
```

```
    testCase.verifyEqual(actStruct,expStruct,'structs not equal');

    actStruct = getArea5(10,22,'unit','m','shape','square');
    expStruct = struct('area',220,'shape','square','units','m');
    testCase.verifyEqual(actStruct,expStruct,'structs not equal');
 end


 function testThreeInputs(testCase) % 记录可以支持三个参数的情况
    actStruct = getArea5(10,22,'units','m');
    expStruct = struct('area',220,'shape','rectangle','units','m');
    testCase.verifyEqual(actStruct,expStruct,'structs not equal');
 end
```

在 testFourInputs 中，我们从 getArea 函数那里先得到一个结构体，命名叫做 actStruct(实际值) 然后准备了一个结构体叫做expStruct(期望值)，然后把用 verifyEqual 方法来作比较在 testThreeInputs 中，我们调换的第三和第四个参数的位置，确保结果依然是我们预期的。

0.5 测试的准备和清理工作: Test Fixtures

本节介绍单元测试框架中另一个很重要的概念叫做 Test Fixtures。所谓 Fixtures, 就是”固定的装置“，而 Test Fixtures 指的是每次测试中固定的要做的工作。假设我们要给图形处理的一系列算法写测试，这些算法需要图像数据作为输入，所以在测试之前，我们需要先载入图像数据，按照上节的例子，单元测试看上去是这样的。

testImgProcess

```
 function tests = testImgProcess(  )
    tests = functiontests(localfunctions);
end


function testOp1(testCase)
    img = imread('testimg.tif');    % 载入图像
    Op1(img);
    % ... rest of the work
end

function testOp2(testCase)
    img = imread('testimg.tif');   % 载入图像
```

```
    Op2(img);
    % ... rest of the work
end
```

可以观察到，在每个测试点的一开始，都有同样的准备工作，就是打开一个图像。在单元测试中，这叫做 Test Fixtures, 即每个测试的固定的共同准备工作。如果这个测试函数中有很多这样的测试点，每次都要重复的调用 imread 操作很麻烦。对于这样的准备工作，我们可以把它们放在一个叫做 setup 的局部函数中，该函数统一地在每个测试点的开始之前被调用。这样就不用在每个测试点中都包括一个 imread 的调用了。新的测试看上去是这样的：

使用 setup 和 teardown

```
function tests = testImgProcess(  )
    tests = functiontests(localfunctions);
end

function setup(testCase)
    testCase.TestData.img = imread('corn.tif');
   % 其它的准备工作
end
function teardown(testCase)
   % 其他清理工作
end
function testOp1(testCase)
    newImg = Op1(testCase.TestData.img);    % 直接使用对象 testCase 的属性 TestData
    % ... rest of the work
end

function testOp2(testCase)
    newImg = Op2(TestCase.TestData.img);h
    % ... rest of the work
end
```

这里需要注意 img 在各个测试点中的传递方式，在 setup 方法中，我们打开一个文件，并把数据动态地添加到 testCase 对象的 TestData 结构体上，这个对象是一个 Handle 对象，在之后的每个局部测试点中，我们可以通过`testCase.TestData.img` 来访问这个数据。setup 中还可以放其他的准备工作，比如创建一个临时的文件夹放置临时的数据等待。对应的 teardown 函数中用来存放每个局部测试点运行完毕之后的清理工作，比如清除临时文件夹。

setup 和 teardown 方法在每个局部测试点的开始和结束后运行，所以如果该主测试文件有两个测试点，那么 setup 和 teardown 各被运行了两次，流程如图所示：

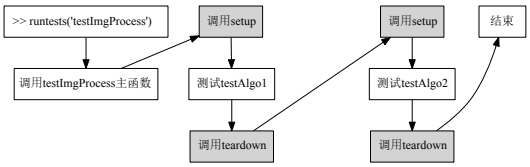


图 4 setup 和 teardown 方法在每个局部测试点的开始和结束后运行

如果还有一些准备和清理工作只需要开始和结束的时候各运行一次，那么可以把他们放到 setupOnce 和 teardownOnce 中去，比如我们要验证一些算法，而给该算法提供的数据来自数据库，在运行算法测试之前，要先连接数据库，在测试结束之后，要关闭和数据库的连接，这样的工作就符合 setupOnce 和 teardownOnce 的范畴，如下所示：

使用 setupOnce teardownOnce 来管理对数据库的连接

```
function tests = testAlgo(  )
    tests = functiontests(localfunctions);
end

function setupOnce(testCase)
    testCase.TestData.conn = connect_DB('testdb'); % 一个假想的连接数据库的函数
end
function teardownOnce(testCase)
    disconnect_DB();
end
function testAlgo1(testCase)
    % retrieve data and do testing
end

function testAlgo2(testCase)
    % retrieve data and do testing

end
```

setupOnce 和 teardownOnce 方法仅仅在整个测试开始和结束时运行一次，流程如图5所示

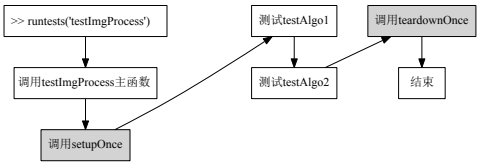


图 5 setupOnce 和 teardownOnce 方法仅仅在整个测试开始和结束是运行一次

setupOnce,teardownOnce 和 setup,teardown 也可以联合起来使用，如图14所示：

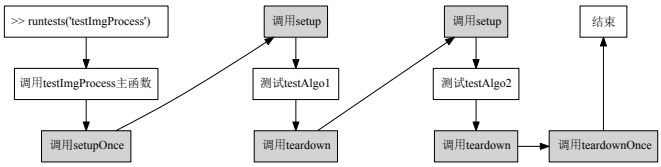


图 6　setupOnce,teardownOnce 和 setup,teardown 联合起来使用

0.6 验证方法: Types of Qualification

在0.3节中我们提到, 如下的测试点中:

testGetArea.m

```
function tests = testGetArea
    tests = functiontests(localfunctions);
end
% 添加了第一个测试点
function testTwoInputs(testCase)
    testCase.verifyTrue(getArea(10,22)==220,'!=220');
    testCase.verifyTrue(getArea(3,4)==12,'!=12');
end
```

参数 testCase 是类`matlab.unittest.FunctionTestCase` 的对象，由 Framework 提供，该类有很多成员验证方法可以提供给用户调用,比如前几节用到的`verifyTrue` 和`verifyError`，这个两个验证方法最常见。全部的验证方法下表所示:

表 1 Type of Qualifications 验证函数

验证方法	验证	典型使用
verifyTrue	表达式值为真	`testCase.verifyTrue(expr,msg)`
verifyFalse	表达式值为假	`testCase.verifyFalse(expr,msg)`
verifyEqual	两个输入的表达式相同	`testCase.verifyEqual(expr1,expr2,msg)`
verifyNotEqual	两个输入的表达式不同	`testCase.verifyNotEqual(expr1,expr2,msg)`
verifySameHandle	两个 handle 指向同一个对象	`testCase.verifySameHandle(h1,h2,msg)`
verifyNotSameHanle	两个 handle 指向不同对象	`testCase.verifyNotSameHandle(h1,h2,msg)`
verifyReturnsTrue	函数句柄执行返回结果为真	`testCase.verifyReturnsTrue(fh,msg)`
verifyFail	无条件产生一个错误	`testCase.verifyFail(msg)`
verifyThat	表达式值满足某条件	`testCase.verifyThat(5, IsEqualTo(5), '')`
verifyGreatThan	大于	`testCase.verifyGreaterThan(3,2)`
verifyGreaterThanOrEqual	大于等于	`testCase.verifyGreateThanOrEqual(3,2)`
verifyLessThan	小于	`testCase.verifyLessThan(2,3)`
verifyLessThanOrEqual	小于等于	`testCase.verifyLessThanOrEqual(2,3)`
verifyClass	表达式的类型	`testCase.verifyClass(value,className)`
verifyInstanceOf	对象类型	`testCase.verifyInstanceOf(derive,?Base)`
verifyEmpty	表达式为空	`testCase.verifyEmpty(expr,msg)`
verifyNotEmpty	表达式非空	`testCase.verifyNotEmpty(expr,msg)`
verifySize	表达式尺寸	`testCase.verifySize(expr,dims)`
verifyLength	表达式长度	`testCase.verifyLength(expr,len)`
verifyNumElements	表达式中元素的总数	`testCase.verifyNumElements(expr,value)`
verifySubstring	表达式中含有字串	`testCase.verifySubstring('thing','th')`
verifyMatches	字串匹配	`testCase.verifyMatches('Another', 'An')`
verifyError	句柄的执行抛出指定错误	`testCase.verifyError(fh,id,msg)`
verifyWarning	句柄的执行抛出指定警告	`testCase.verifyWarning(fh,id,msg)`
verifyWarningFree	句柄的执行没有警告	`testCase.verifyWarningFree(fh)`

除了 verify 系列的函数，MATLAB 单元测试还提供 assume 系列，assert 系列和 fatalAssert 系列的验证函数，也就是说，上面每一个 verify 函数，都有一个对应的 assume,assert

和 fatalAssert 函数。比如除了 verifyTrue, 还有 assumeTrue,assertTrue,fatalAssertTrue 三个验证方法。

assume 系列的验证方法一般用来验证一些测试是否满足某些先决条件，如果满足，测试继续，如果不满足，则过滤掉这个测试，但是不产生错误。比如下面的测试点，如果测试者的意图是：在 Windows 平台下才执行，没有必要在其它平台下执行

tFoo.m

```
function tests = tFoo
    tests = functiontests(localfunctions);
end

function testSomething_PC(testCase)
    testCase.assumeTrue(ispc,'only run in PC');    % 如果这个测试点在其它平台运行，
                                                   % 则显示 Incomplete
    % ....
end
```

如果我们在 MAC 下运行这个测试，则显示

```
>> runtests('tFoo')
Running tFoo
================================================================================
tFoo/testSomething_PC was filtered.
    Test Diagnostic: only run in PC
    Details
================================================================================
.
Done tFoo
----------

Failure Summary:

     Name                    Failed  Incomplete  Reason(s)
    =====================================================================
     tFoo/testSomething_PC               X       Filtered by assumption.
                                                 该测试被过滤掉了
ans =
  TestResult with properties:

          Name: 'tFoo/testSomething_PC'
```

```
        Passed: 0
        Failed: 0
    Incomplete: 1
      Duration: 0.0466

Totals:
   0 Passed, 0 Failed, 1 Incomplete.
   0.046577 seconds testing time.
```

assert 系列的验证方法也是用来验证一些测试是否满足某些先决条件，如果满足，测试继续，如果不满足，则过滤掉这个测试，并且产生错误。但是它不会影响其余的测试点。比如下面这个例子，testSomething 测试点中，我们要求该测试的先决条件是数据库必须先被连接，如果没有连接，那么没有必要进行余下的测试，并且 testA 的测试结果显示失败。但是这个失败将不会影响 testB 测试点的运行

```
function tests = tFoo
    tests = functiontests(localfunctions);
end

function testA(testCase)
    testCase.assertTrue(isConnected(),'database must be connected!')
    % 其它测试内容
end

function testB(testCase)
    testCase.verifyTrue(1==1,'');
end
```

运行这个测试，显示如下

command line

```
  >> runtests('tFoo')
Running tFoo
================================================================================
Assertion failed in tFoo/testA and it did not run to completion.
    ----------------
    Test Diagnostic:
    ----------------
```

```
    database must be connected!
    ---------------------
    Framework Diagnostic:
    ---------------------
    assertTrue failed.
    --> The value must evaluate to "true".

    Actual logical:
             0
    ------------------
    Stack Information:
    ------------------
    In /Users/iamxuxiao/Documents/MATLAB/tFoo.m (testA) at 6
================================================================================
..
Done tFoo
----------

Failure Summary:

     Name        Failed  Incomplete  Reason(s)
    =====================================================
     tFoo/testA    X         X       Failed by assertion.

Totals:
   1 Passed, 1 Failed, 1 Incomplete.
   0.036008 seconds testing time.
```

最后，fatalAssert 系列的验证方法, 顾名思义，就是如果失败，立即停止结束所有的测试。如果还有未运行的测试点，则不再运行它们，例子从略。

0.7 测试方法论和用测试驱动开发

0.7.1 开发流程概述

在前节的基础上，本节将抽象的讨论 MATLAB 常见的开发流程，引入用测试驱动开发的思想。先概述一下常见的开发工作流程。最简单也是最常见的工作流程是：先用代码实现一个功能，然后在命令行测试该代码是否达到预期目的，如果达到了，则该函数放到更大的工程项目中去使用，然后不再去更新，如图所示：

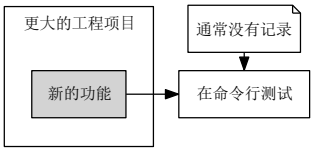


图 7　最简单最常见的工作流程

如果比较复杂的功能，在写好的代码放入更大的工程项目之前，我们通常需要在命令行中反复的测试各个方面的功能，方便起见，我们通常还会写一个专门测试的脚本，比如新的函数如果叫做 op1, 通常习惯会写一个 script1.m 来一次性测试 op1 的所有的功能。测试完毕之后，把 op1 函数放入工程项目中, 而该 script1.m 脚本，通常因为没有很好的管理方式，则难免遗忘在某个文件夹中，或遗忘在工程项目的最上层目录里面，最终被清理掉。

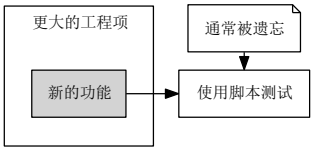


图 8　用脚本测试

本节我们将引入的工作流程是：开发一个复杂的功能，从开发最简单的部分开始，循序渐进的完成更复杂的需求，并且在此同时引入该功能配套的单元测试文件，测试和开发同步进行，测试和要测试的代码共生在同一个目录下。即使要测试的内容被加入的更大的项目之中，我们还是保留这个测试，单元测试本身也是工程项目中的一部分。

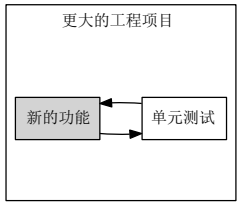


图 9　单元测试是工程项目的一部分

测试还是多人合作项目中不可缺少缺少的环节。比如 A 和 B 共同开发一个项目两人分别负责该项目中的不同部分，他们的工作项目依赖相互调用，甚至有少量的重叠，即有可能要修改对方的代码。那么如何保证 A 在修改 B 的代码的时候不会破坏 B 已有的功能呢，这就要依靠 B 写的测试代码了。在 A 修改完代码之后，但在 A 提交代码到 Repository 之前，A 必须在本地的工程项目中运行所有的测试，这些测试确保 A 不会意外的破坏 B 的代码的

已有的功能，所以 B 的测试也起到了保护自己代码的作用，因为它起到了对他人的约束作用。

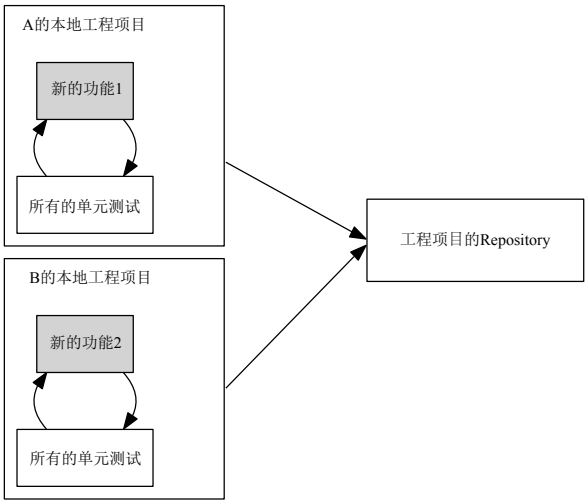


图 10　提交之前必须运行所有的测试

前面我们提出一个问题：如下测试点里验证显而易见的`getArea(10,22) == 220` 真的有必要吗?

testGetArea.m

```
    ...
function testTwoInputs(testCase)
    testCase.verifyTrue(getArea(10,22)==220,'!=220');
    ...
end
...
```

从另一角度看：有必要。因为单元测试其实是程序最好的文档。因为我们不可能给每一个函数都写文档，或者在函数里面都写清楚详细的注释。天长日久之后，即使有注释也许因为遗忘而很难看懂。当我们要回忆一个函数，一个功能如何使用的时候，最快的办法不是去读它的实现代码或者注释，而是去查找工程项目中其它的地方是如何使用这个功能的。但是如果工程项目过于复杂，这也不会是一件容易的事情。如果有了这个函数的单元测试，因为这个单元测试是仅仅关于这一个功能的，那么我们会很容易就通过单元测试就可以了解这个函数的功能是什么。所以`getArea(10,22) == 220` 不但是一个历史的记录，记录这个函数要实现的功能，还是该函数最好的说明文档，为了让这个说明文档以后阅读起来更加的清晰，我们还必要错误的提示信息写得更加详细一些，比如上面的测试点可可以这样改写

错误提示信息其实是 `getArea` 文档的一部分

```
    ...
function testTwoInputs(testCase)
```

```
    testCase.verifyTrue(getArea(10,22)==220,'given width and height, ...
                                            should return 10*22=220');
    ...
end
...
```

前面所讨论的开发模式，测试总是作为主要功能的辅助，还有一种流行的开发模式，测试的地位和要测试的代码的地位是不相上下，这种测试和开发的工作流程，叫做用测试驱动 (Test Driven Development), 也值得我们了解一下。我们先前的这些工作流程无一例外都是先写算法，然后补上测试代码；读者有没有想过可不可以先写测试，再写函数的实现呢。为什么要这样开发，这样开发有什么好处，我们将举例说明。

0.7.2 用测试驱动开发：fibonacci 例

假设一个教编程的老师给学生布置了一道 MATLAB 程序，要求写一个计算 Fibonacci 数列的函数。已知，Fibonacci 函数定义如下：

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$

当 $n = 1;2$ 时 $F_1 = F_2 = 1$。并且规定 $n = 0$ 时，$F_0 = 0$。要求除了计算正确以外，还必须能正确的处理各种非法的输入，比如输入是非整数，负数或者字符串的情况。

所谓以测试驱动做开发就得到程序的需求之后，在这里即老师的作业要求，先写测试的代码，再写程序。比如根据老师的要求，很容易就写出该函数要满足的条件的一个清单

- □ `fibonacci(0)= 0`
- □ `fibonacci(1)= 1`
- □ `fibonacci(2)= 1`
- □ `fibonacci(3)= 2 ;  fibonacci(4)= 3`
- □ `fibonacci(1.5)` 报错
- □ `fibonacci(-1)` 报错
- □ `fibonacci('a')` 报错

根据这些条件，我们可以很容易的写出两个测试点，一个是正面测试，一个负面测试

testFib.m

```
function tests = testFib(  )
    tests = functiontests(localfunctions);
end


function testValidInputs(testCase)
    % fibonacci function only accepts integer
    testCase.verifyTrue(fibonacci(int8(0))  ==0, 'f(0) Error');
    testCase.verifyTrue(fibonacci(int16(1)) ==1, 'f(1) Error');
```

```
    testCase.verifyTrue(fibonacci(int32(2)) ==1, 'f(2) Error');
    testCase.verifyTrue(fibonacci(uint8(3)) ==2, 'f(3) Error');
    testCase.verifyTrue(fibonacci(uint16(4))==3, 'f(4) Error');
    testCase.verifyTrue(fibonacci(uint32(5))==5, 'f(4) Error');
end

function testInvalidInputs(testCase)
    testCase.verifyError(@()fibonacci(1.5),'MATLAB:invalidType');
    testCase.verifyError(@()fibonacci(-1), 'MATLAB:invalidType');
    testCase.verifyError(@()fibonacci('a'),'MATLAB:invalidType');
end
```

其中第一个测试点尝试了各种不同的整数类型，第二个测试点确保在输入非法的情况下，函数要抛出错误.[①] 。讨论如何实现函数 fibonacci 之前，我们先讨论一下先写测试给我们带来的好处：

- 先写测试有助于我们对函数的设计，即使用和行为。在这些测试中，其实罗列了函数的各种的使用情况，甚至还包括出错的 ErrorID。我们先设计的函数的" 外观"，即它接受什么样的输入，如何返回结果，这在使用中是很重要的，程序的开发者，站在了一个使用者的角度去设计了这个函数，这将更加有利于我们设计出友好的函数。
- 每个测试点中的测试都是极其简单的，仅仅测试函数一个小的方面，这样很容易看懂，该测试文件是 fibonacci 函数的极好的说明文件。

该 fibonacci 的计算有两种方法，递归和非递归，我们先设计递归的版本, 如下：

fibonacci.m

```
function result = fibonacci( n )
    validateattributes(n,{'int8', 'int16', 'int32', 'int64', 'uint8',...
        'uint16', 'uint32', 'uint64'},{'>=',0});
     if n==0
         result = 0;
     elseif n <= 1
         result = 1;
     else
         result = fibonacci(n-1) + fibonacci(n-2);
     end
end
```

① 第二个参数是 validateattributes 函数在输入非法是的常见的 Error ID. 其实我们这里已经认定函数要使用 validateattributes 来检查输入的类型了

运行 runtests 无误, 我们完成了清单中的所有内容。

☑ `fibonacci(0)= 0`

☑ `fibonacci(1)= 1`

☑ `fibonacci(2)= 1`

☑ `fibonacci(3)= 2 ;  fibonacci(4)= 3`

☑ `fibonacci(1.5)` 报错

☑ `fibonacci(-1)` 报错

☑ `fibonacci('a')` 报错

有了测试，即函数需要满足的需求，我们就放心的改进这个函数了，现在假设老师第二天布置了一个新的任务，要求用非递归的方式实现这个函数[①]，于是我们可以在原函数的基础上，把函数修改成:

fibonacci.m

```
function result = fibonacci( n )
    validateattributes(n,{'int8', 'int16', 'int32', 'int64', 'uint8',...
        'uint16', 'uint32', 'uint64'},{'>=',0});

    if  n==1 || n==2
        result =1;
    else
        y1 = 1;
        y2 = 1;
        for iter = 3: n      % 这里用累加的方式代替递归
            result = y1 + y2;
            y2=y1;
            y1=result;
        end
    end
end
```

写完这个新的实现之后，我们首先运行一遍已有的测试，检查一下新的改动有没有破坏以前的功能

command line

```
>> runtests('testFib')
Running testFib
================================================================================
Error occurred in testFib/testValidInputs and it did not run to completion.
```

① 这里不是改进，而是设计一个新的算法

```
    --------------
    Error Details:   % 错误 !!
    --------------
    Output argument "result" (and maybe others) not assigned during call to
    "fibonacci".
    Error in testFib>testValidInputs (line 6)
        testCase.verifyTrue(fibonacci(uint8(0))==0,'f(0) Error');
================================================================================
..
Done testFib
__________
Failure Summary:
     Name                      Failed  Incomplete  Reason(s)
    =====================================================
     testFib/testValidInputs    X         X       Errored.
ans =
  1x2 TestResult array with properties:
    Name
    Passed
    Failed
    Incomplete
    Duration
Totals:
   1 Passed, 1 Failed, 1 Incomplete.
   0.024965 seconds testing time.
```

有一个测试没有通过，第 11 行提示 f(0) 的结果错了，这就是单元测试的最重要作用之一，防止算法的退化。检查发现，原来是因为函数在 n=0 时没有返回值，于是添加一个 result=0 作为默认返回，新函数如下：

fibonacci

```
function result = fibonacci( n )
    validateattributes(n,{'int8', 'int16', 'int32', 'int64', 'uint8',...
        'uint16', 'uint32', 'uint64'},{'>=',0});

    result =0 ;        % 默认的返回值
    if  n==1 || n==2
        result =1;
    else
```

```
        y1 = 1;
        y2 = 1;
        for iter = 3: n      % 这里用累加的方式代替递归
            result = y1 + y2;
            y2=y1;
            y1=result;
        end
    end
end
```

再次运行 `runtests('testfib')`，所有测试通过。回顾以测试驱动的这种模式，关键在于我们在编写函数之前，先列出了函数要满足的条件，再写测试，最后再写实现的代码。先编写测试的好处是有利于我们在一开始就站在用户的角度去使用这个 API。总的来说开发流程如下：

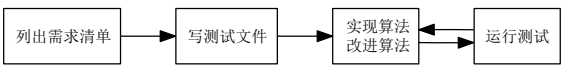


图 11　测试在算法的重构和改进过程中提供保障

0.7.3　用测试驱动开发：算符重载和量纲分析

本节重用第**??**节量纲的例子来介绍用测试驱动开发，但这里的重点不是如何设计量纲系统[①]，而是示例如何在工作中从需求出发，先完成测试代码，然后再编写实际的生产代码，并且如此循环往复的开发。

众所周知，工程科学计算中，一般物理量都是有单位的，比如：速度单位是米每秒 LT^{-1}，由长度基本量纲和时间基本量纲构成。加速度单位是米每秒平方，其量纲是 LT^{-2}。加速度乘以质量得到力，单位牛顿，其量纲是 MLT^{-1}：

$$F = ma$$

单位和量纲的运算遵从量纲法则：只有量纲相同的物理量，才能彼此相加、相减。也就是说我们不可以把速度和加速度相加：

$$? = v + a$$

工程科学计算中，如果不小心对不同单位的物理量做了加减运算，不但结果是错误的，而且应用到实际也可能会带来危险。所以有必要建立这样的一个单位和量纲系统，在计算的过程中可以携带单位，计算得到的结果也是有单位的物理量，并且在不小心对不同单位的物理做了加减运算时，该系统能够终止计算并且提示错误。

国际标准量纲制规定了物理量的基本量纲是：质量，长度，时间，电荷，温度，密度和物质的量，一个物理量的量纲可以用一个整数数组类表示，为了构造一个量纲系统，首先需

① 所以会省略一些设计细节，完整例子请参考重载章节

要一个量纲类，该类的构造函数要能接受 1×7 的数组作为输入，并且我们规定，该构造函数只接受 1×7 的行向量，不接受元胞，不接受列向量：

- □ `Dimension([1,0,0,0,0,0,0])` 表示质量基本量纲
- □ `Dimension([0,1,0,0,0,0,0])` 表示长度基本量纲
- □ `Dimension([0,0,1,0,0,0,0])` 表示时间基本量纲
- □ `Dimension({1,2})` 报错
- □ `Dimension([2,2,2,2,2,2,2,2,2])` 报错
- □ `Dimension([0;0;1;0;0;0;0])` 报错

根据这些条件，我们构造一个测试文件叫做 tDimensionExample，很容易写出两个关于构造函数的测试点，一个正面测试[①]，一个负面测试。

tDimensionExample

```
function tests = tDimensionExample()
tests = functiontests(localfunctions);
end

function testConstructor(testCase)
 Dimension([1,0,0,0,0,0,0]);   % 确保构造质量量纲对象无误
 Dimension([0,1,0,0,0,0,0]);   % 确保构造长度量纲对象无误
 Dimension([0,0,1,0,0,0,0]);   % 确保构造时间量纲对象无误
end

function testCtor_negative(testCase)
 testCase.verifyError(@() Dimension({1,2}), 'MATLAB:invalidType');
 testCase.verifyError(@() Dimension([2,2,2,2,2,2,2,2,2]), 'MATLAB:incorrectSize');
 testCase.verifyError(@() Dimension([0;0;1;0;0;0;0]), 'MATLAB:incorrectSize');
end
```

注意在这里写完测试的时候，我们甚至没有开始写 Dimension 类的具体实现，其实这里已经完成了基本的 Dimension 类的构造函数的设计，包括有非法输入时该抛出什么错误。这样的方法好处是迫使我们先从使用者的角度去考虑，有助于设计出友好的函数。

满足上面条件的 Dimension 类如下，在算符重载章节已有详细介绍，这里不再赘述：

Dimension.m

```
1 classdef Dimension
2     properties
3         value
4     end
5     methods
6         function obj = Dimension(input)
```

① 简单起见，没有使用 verify 系的函数，只要可以构造基本量纲对象，MATLAB 不出错，就算通过测试。

```
7           validateattributes(input,{'numeric'},{'size',[1 7]});
8           obj.value = input;
9       end
10   end
11 end
```

该设计很容易就通过了两个测试点的测试

command line

```
>> runtests('tDimensionExample.m')
Running tDimensionExample
..
Done tDimensionExample

---------
ans =
  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.014111 seconds testing time.
```

这是 Dimension 类实现的一个阶段性的成就，现在我们可以放心的继续开发这个类了。Dimension 类的重要功能就是对算术运算的进行单位检查：比如加减运算只能在相同量纲的物理量之间进行，那么加减法要满足的需求清单如下：

- □ 加减运算的正面测试

```
t1 = Dimension([0,0,1,0,0,0,0]) ;
t2 = Dimension([0,0,1,0,0,0,0]) ;
t3  = t1 + t2; % 结果量纲是 [0,0,1,0,0,0,0]
```

- □ 加减运算的负面测试

```
t1 = Dimension([0,0,0,0,0,0,0]) ;
l1 = Dimension([0,1,0,0,0,0,0]) ;
x  = t1 - l1;  % 应该报错。
```

根据这些需求，我们给已有的测试文件 tDimension 添加两个测试点

tDimensionExample

```
function tests = tDimensionExample()
tests = functiontests(localfunctions);
end

... 省略 testConstructor 和 testCtor_negative 测试点

function testplus(testCase)
 t1=Dimension([0,0,1,0,0,0,0]) ;
 t2=Dimension([0,0,1,0,0,0,0]) ;
 t3=t1+t2;
 testCase.verifyTrue(t3.unit==[0,0,1,0,0,0,0],...
                     'dimension changed for addition');
end

function testInvalidDimOp(testCase)
 t1=Dimension([0,0,0,0,0,0,0]) ;
 t2=Dimension([0,1,0,0,0,0,0]) ;
 testCase.verifyError(@()t1+t2,'Dimension:DimensionMustBeTheSame');
end
```

具体的必须重载 Dimension 类的 plus 和 minus 算符:

Dimension.m

```
1  classdef Dimension
2      % value object, can do compare directly
3      properties
4          value    % dimension  value
5      end
6      methods
7          .... 其余略
8          function newunit = plus(o1,o2)
9              isequalassert(o1,o2);
10             newunit = o1;     % 结果的量纲等于 o1 的量纲或者 o2 的量纲
11         end
12
13         function newunit = minus(o1,o2)
14             isequalassert(o1,o2);
15             newunit = o1;   % 结果的量纲等于 o1 的量纲或者 o2 的量纲
```

```
16          end
17  end
18
19  % 类 Dimension 的局部函数
20  function isequalassert(v1,v2)
21      if isequal(v1,v2)
22      else
23          error('Dimension:DimensionMustBeTheSame','');
24      end
25  end
```

说明如下

- 第 1 行中，我们把该 Dimension 类设计成了 Value 类，这是为了方便第 21 行直接对两个输入的量纲进行比较.[①]
- 第 9 和第 14 行中，我们调用了类的局部的函数[②] isequalassert，在每次进行计算之前，检查运算的量纲是否相同，如果不同，则报错。
- 量纲的加减运算，结果量纲不变，所以第 10，15 行直接构造一个新的 Dimension 对象，等于原来的量纲值，作为结果返回。

再运行一遍所有的测试，确保我们新添的函数没有破坏已有的功能：

command line

```
>> runtests('tDimensionExample.m')
Running tDimensionExample

.... 部分输出从略

Totals:
   4 Passed, 0 Failed, 0 Incomplete.
   0.020766 seconds testing time.
```

目前为止，Dimension 的功能是一点一点加入类的定义中的，我们通过逐步的提出需求，设计 API 写出测试，来指导我们一步一步稳扎稳打的实现这个 Dimension 类。下面我们继续考虑新的功能：Dimension 类只是一个表示单位的类，不包括物理量的实际值。上面的设计加法和减法也仅仅限于单位之间的加减法。为了表示物理量的实际的值，我们还需要一个 Quantity 类，该类既包括物理量的值，也包括物理量的 Dimension。该类需要满足如下简单构造：提供实际值和量纲对象做为构造函数的输入，返回 Quantity 对象：

- 2 千克的质量物理量表示成

① handle 类对象之间的比较具有不同的意义

② 请参考相关章节

```
m1 = Quantity(2,Dimension([1,0,0,0,0,0,0]));
```

□ 加速度为 3 的物理量表示成

```
a1 = Quantity(3,Dimension([0,1,-2,0,0,0,0])) ;
```

□ 我们应该可以对 Quantity 对象进行加减法, 如下计算应该得到 6 千克的质量:

```
m1 = Quantity(2,Dimension([1,0,0,0,0,0,0]));
m2 = Quantity(4,Dimension([1,0,0,0,0,0,0]));
m3 = m1 + m2 ;                          % 结果 m3 表示 6 千克的质量
```

这里篇幅有限，只提出几个最有代表性的需求，然后构造测试点，实际应用中，需求越详细，测试点越多，最后得到的生产代码也就越健壮。新添加一个 Quantity 的构造函数的测试点

tDimensionExample.m

```
....
function testQuantityCtor(testCase)
    m1 = Quantity(2,Dimension([1,0,0,0,0,0,0]));
    m2 = Quantity(4,Dimension([1,0,0,0,0,0,0]));
    a1 = Quantity(3,Dimension([0,1,-2,0,0,0,0]));
    m3 = m1 + m2;
    testCase.verifyTrue(m3.unit ==[1,0,0,0,0,0,0],'unit unchanged for addition');
    testCase.verifyTrue(m3.value==6,'2kg + 4kg = 6kg');
end
```

和以往一样，在没有写一行 Quantity 的代码之前，我们就先设计了它的构造函数所接受的输入的形式，还设计它有两个属性，一个叫做 unit, 一个叫做 value。在 Dimension 类的设计的基础上，容易写出 Quantity 类的:

Quantity.m

```
1  classdef Quantity
2      properties
3          value
4          unit
5      end
6      methods
7          function obj = Quantity(value,unit)
8              obj.value = value;
9              obj.unit = unit;
10         end
11         function results = plus(o1,o2)
12             results = Quantity(o1.value+o2.value,o1.unit+o2.unit);
```

```
13          end
14          function results = minus(o1,o2)
15              results = Quantity(o1.value-o2.value,o1.unit-o2.unit);
16          end
17      end
18  end
```

其中第 12 行和第 15 行的第 2 个参数把对 Quantity 的单位的加减法的计算转到了 Dimension 类的加减法计算函数上去。最后再运行一次测试文件确保无误，结果从略。

完成了对加法和减法的设计，并且通过了所有加法和减法的测试点。这些测试点是对已有的功能的描述和锁定，在这个基础上，我们再讨论乘法和除法的需求，量纲运算法则对乘除法的规定是：复合量纲是其它的量纲的幂积，具体需求表示如下:

□ Dimension 对象之间的除法

```
s1 = Dimension([0,1,0,0,0,0,0]) ;  % length
t1 = Dimension([0,0,1,0,0,0,0]) ;  % time
v1  = s1/t1    ;                   % 结果是速度 量纲是 [0,1,-1,0,0,0,0]
s2  = v1 * t1  ;                   % 结果是长度 量纲是 [1,0,0,0,0,0,0]
```

□ Dimension 对象之间的乘法

```
m1 = Dimension([1,0,0,0,0,0,0])  ;  % mass
a1 = Dimension([0,1,-2,0,0,0,0]) ;  % acceleration
f1 = m1 * a1 ;                 % 结果单位是力 量纲是 [1,1,-2,0,0,0,0]
```

□ Quantity 对象之间的除法

```
s1 = Quantity(2, Dimension([0,1,0,0,0,0,0])) ;  % 2 米
t1 = Quantity(4, Dimension([0,0,1,0,0,0,0])) ;  % 4 秒
v1  = s1/t1    ;             % 结果值 0.5 单位是速度 量纲 [0,1,-1,0,0,0,0]
```

□ Quantity 对象之间的乘法

```
m1 = Quantity(3,Dimension([1,0,0,0,0,0,0]))  ;  % 3 千克
a1 = Quantity(5,Dimension([0,1,-2,0,0,0,0])) ;  % 5 米每秒平方
f1 = m1 * a1 ;                 % 结果值 15 单位是牛 量纲 [1,1,-2,0,0,0,0]
```

添加两个测试点对应 Dimension 类和 Quantity 类的乘除操作

tDimensionExample.m

```
...
function testQuantityMultiplyDivide(testCase)
s1 = Quantity(2,Dimension([0,1,0,0,0,0,0]));
```

```
t1=  Quantity(4,Dimension([0,0,1,0,0,0,0]));

v1 = s1/t1;
testCase.verifyEqual(v1.value,0.5);
testCase.verifyEqual(v1.unit,Dimension([0,1,-1,0,0,0,0]));

s2 = v1 * t1;
testCase.verifyEqual(s1.value,2);
testCase.verifyEqual(s2.unit,Dimension([0,1,0,0,0,0,0]));

m1 = Quantity(3,Dimension([1,0,0,0,0,0,0]));
a1=  Quantity(5,Dimension([0,1,-2,0,0,0,0]));
f1 = m1 * a1;
testCase.verifyEqual(f1.value,15);
testCase.verifyEqual(f1.unit,Dimension([1,1,-2,0,0,0,0]));
end

function testDimensionMultiplyDivide(testCase)
s1 = Dimension([0,1,0,0,0,0,0]);
t1=  Dimension([0,0,1,0,0,0,0]);
v1 = s1/t1;
testCase.verifyEqual(v1,Dimension([0,1,-1,0,0,0,0]));

m1 = Dimension([1,0,0,0,0,0,0]);
a1=  Dimension([0,1,-2,0,0,0,0]);
f1=m1 * a1;
testCase.verifyEqual(f1,Dimension([1,1,-2,0,0,0,0]));
end
...
```

除了可以用 Quantity 的构造函数来生产 Quantity 对象，根据我们计算的书写习惯，我们还期望 Quantity 类还支持其它的构造对象的方法。比如 8 秒钟这个物理量，可以用 Quantity 类的构造函数来构造：

```
t1 = Quantity(8, Dimension([0,0,1,0,0,0,0]))
```

当然我们还知道 8 秒钟，其实还可以通过 4 秒钟 $\times 2$, 或者 8 $\times$ 一个单位时间秒来表示，这里的 2 和 8 都是标量，或者普通的数字。该需求反映在代码上，就是要求下面的计算都能返回 value 为 8，unit 是`Dimension([0,0,1,0,0,0,0])` 的 Quantity 对象，需求如下：

□ Quantity 类还支持其它的对象的方法

```
% Quantity 类对象和 scalar 的乘法得到 Qunatity 对象
t1 = Quantity(2,Dimension([0,0,1,0,0,0,0]) * 4
t2 = 4 * Quantity(2,Dimension([0,0,1,0,0,0,0])

% Dimension 类对象和 scalar 的乘法得到 Qunatity 对象
t3 = 8 * Dimension([0,0,1,0,0,0,0]
t4 = Dimension([0,0,1,0,0,0,0] * 8
```

和 scalar 乘除法对应的单元测试可以写作

tDimensionExample.m

```
...
function testMixProduct(testCase)
t1 = Quantity(2,Dimension([0,0,1,0,0,0,0]))*4;
t2=  4 * Quantity(2,Dimension([0,0,1,0,0,0,0]));

testCase.verifyEqual(t1.value,t2.value);
testCase.verifyEqual(t1.unit,t2.unit);

t3 = Dimension([0,0,1,0,0,0,0])*8;
t4=  8 * Dimension([0,0,1,0,0,0,0]);

testCase.verifyEqual(t2.value,t3.value);
testCase.verifyEqual(t1.unit,t4.unit);

end
```

目前为止我们对这个量纲系统的需求描述完毕，根据这些需求，我们给 Dimension 的类新添了第 21 到 48 行，其完整设计如下：

Dimension.m

```
1  classdef Dimension
2      % value object, can do compare directly
3      properties
4          value    % dimension  value
5      end
6      methods
7          function obj = Dimension(input)
8              validateattributes(input,{'numeric'},{'size',[1 7]});
```

```
9           obj.value = input;
10      end
11
12      function newunit = plus(o1,o2)
13          isequalassert(o1,o2);
14          newunit = o1;
15      end
16
17      function newunit = minus(o1,o2)
18          isequalassert(o1,o2);
19          newunit = o1;
20      end
21      function newObj = mtimes(o1,o2)
22           validateattributes(o1,{'numeric','Dimension'},{});
23           validateattributes(o2,{'numeric','Dimension'},{});
24           if isnumeric(o1)  && isa(o2,'Dimension')
25                newObj = Quantity(o1,o2);
26           elseif isnumeric(o2)  && isa(o1,'Dimension')
27                newObj = Quantity(o2,o1) ;
28           elseif isa(o1,'Dimension') && isa(o2,'Dimension')
29                newObj = Dimension(o1.value + o2.value);
30           else
31              % will not reach here, due to dispatch rules
32           end
33      end
34
35      function newObj = mrdivide(o1,o2)
36           validateattributes(o1,{'numeric','Dimension'},{});
37           validateattributes(o2,{'numeric','Dimension'},{});
38           if isnumeric(o1)  && isa(o2,'Dimension')
39                newObj = Quantity(o1,Dimension(-o2.value));
40           elseif isnumeric(o2)  && isa(o1,'Dimension')
41                newObj = Quantity(1/o2,o1) ;
42           elseif isa(o1,'Dimension') && isa(o2,'Dimension')
43                newObj = Dimension(o1.value - o2.value);
44           else
45              % will not reach here, due to dispatch rules
```

```
46              end
47
48          end
49      end
50  end
51
52  function isequalassert(v1,v2)
53      if isequal(v1,v2)
54      else
55          error('Dimension:DimensionMustBeTheSame','');
56      end
57  end
```

Dimension 类设计说明如下

- 第 22，23，36，37 行确保做乘除运算时，运算符要么是两个 Dimension 对象

```
v1  = s1/t1    ;
s2  = v1 * t1  ;
```

要么是 Dimension 对象和简单的标量

```
t3 = 8 * Dimension([0,0,1,0,0,0,0]
t4 = Dimension([0,0,1,0,0,0,0] * 8
```

- mtimes 从第 24 行起，根据运算符 o1 或者 o2 是否是 scalar，区别对待。规定，只有在两个运算符都是 Dimension 对象时，返回的结果才是 Dimension 对象，Dimension 对象和 scalar 的计算结果是 Quantity 对象。
- mrdivide 的设计原理和 mtimes 类似

我们对 Quantity 类新添了 17-45 行，其完整设计如下：

Dimension.m

```
1   classdef Quantity
2       properties
3           value
4           unit
5       end
6       methods
7           function obj = Quantity(value,unit)
8               obj.value = value;
9               obj.unit = unit;
10          end
```

```
        function results = plus(o1,o2)
            results = Quantity(o1.value+o2.value,o1.unit+o2.unit);
        end
        function results = minus(o1,o2)
            results = Quantity(o1.value-o2.value,o1.unit-o2.unit);
        end
        function newObj = mtimes(o1,o2)
            [o1,o2] = converter_helper(o1,o2);
            newObj = Quantity(o1.value*o2.value,o1.unit*o2.unit);
        end
        function newObj = mrdivide(o1,o2)
            [o1,o2] = converter_helper(o1,o2);
            newObj = Quantity(o1.value/o2.value,o1.unit/o2.unit);
        end
    end
end

% convert input into Quantity object
function [o1,o2] = converter_helper(o1,o2)
validateattributes(o1,{'numeric','Quantity','Dimension'},{});
validateattributes(o2,{'numeric','Quantity','Dimension'},{});

% convert numeric input into Quantity object
if isnumeric(o1)
    o1 = Quantity(o1,Dimension([0,0,0,0,0,0,0]));
elseif isnumeric(o2)
    o2= Quantity(o2,Dimension([0,0,0,0,0,0,0]));
end

% convert dimension input into Quantity object
if isa(o1,'Dimension')
    o1 = Quantity(1,o1);
elseif isa(o2,'Dimension')
    o2= Quantity(1,o2);
end
end
```

Quantity 类说明如下

- 第 18，22 行对输入进行预处理，调用局部函数 converter_helper 确保两个参数们统统都是 Quanit 对象，再分别对其 value 和 unit 部分做计算。第 19, 23 行返回新的 Quantity 对象。
- 局部函数 converter_helper 从第 29 行开始，第 30，31 行 validateattributes 限制操作数只能是简单的数字,Quantity 或者 Dimension 对象。如果有任何一个输入是简单的数字, 第 34 到 38 行把它转成 Quantity 对象，量纲为 Scalar。如果有任何一个输入是 Dimension 对象，第 41 到 45 行把它转成 Quantity 对象，其值是 1，其 Dimension 不变.

最后验证所有的测试点通过测试

command line

```
>> runtests('tDimensionExample.m')
Running tDimensionExample
........
Done tDimensionExample

----------

ans =
  1x8 TestResult array with properties:
    Name
    Passed
    Failed
    Incomplete
    Duration

Totals:
   8 Passed, 0 Failed, 0 Incomplete.
   0.068013 seconds testing time.
```

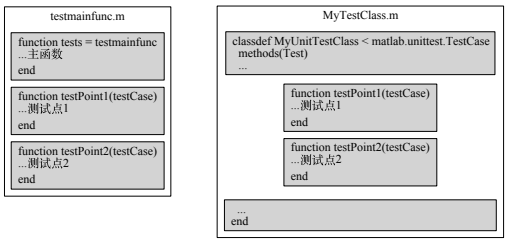
本节的量纲系统的例子较 fibonacci 的例子更加的复杂，Dimension 和 Quantity 类的设计有方方面面的细节，很难想象实现者能从一开始就对类的各个方面的设计成竹在胸。在实际工程项目中，工程的复杂度要大的多，要求实现者能一开始就能有一个完整的设计也是不现实的。由测试驱动的开发本质上提供了一种从局部开始，步步为营稳扎稳打的解决问题的流程。先写单个测试点迫使设计者从小处着手，把大的问题分解成小的问题，把每个小的问题解决好，并且每个解决的小问题，都能通过测试点中记录下来，以此来锁定已有的成绩。流程如下:

图 12　测试于开发之间的不断迭代和往复

需要说明的是，这并不代表已有的测试点必须是一成不变的，在实际应用中，如果一个新的功能不得已会造成已有的测试的失败，我们要权衡之后再决定是修改生产代码还是修改测试。以测试为驱动的开发是一种开发风格，这种风格注重从局部 API 的测试开始设计和完成。但这和我们的对工程项目进行高屋建瓴的框架设计不冲突。从局部的 API 的测试入手，有助于设计出更简单的框架。而且，单元测试能够保证大系统中的每一个小系统都能健壮的准确无误的工作，这样我们才更有信心对大的系统进行改良和重构。

0.8 基于类的单元测试

单元测试 Framework 除了可以运行 Function-Based 的测试，还可以运行 Class-Based 的测试，如图2所示，和 Function-Based 的测试相比，Class-Based 的测试只是把测试点用类的格式写了出来，其实内容都一样，都交给同一个 Framework 去处理。使用 Function-Based 测试的好处是：书写迅速简单，不需要面向对象的基础。使用 Class-Based 的测试的好处是：可以使用单元测试 Framework 中包括面向对象的更多高级的功能。



(a) 基于函数的单元测试　　(b) 基于类的单元测试

图 13　两种风格的单元测试在格式上的比较：(左)Function-Based(右)Class-Based

0.8.1 getArea 函数的基于类的单元测试

假设我们要测试的函数是第**??**节中，包含 inputParser 的 getArea 函数。

getArea

```
function a = getArea(width,varargin)

  p = inputParser;
  p.addRequired('width',@(x)validateattributes(x,{'numeric'},...
                                        {'nonzero'},'getArea','width',1));



  defaultheight = width;
  p.addOptional('height',defaultheight,@(x)validateattributes(x,{'numeric'},...
                                        {'nonzero'},'getArea','height',2));

  p.parse(width,varargin{:});

  a = p.Results.width*p.Results.height;
end
```

一个简单的 Class-Based 单元测试类如下, 其中第一个测试点验证两个输入的面积的计算，第二个测试点验证当一个参数缺省时，算法返回的是正方形的面积。

MyUnitTestClass 最初的定义

```
classdef MyUnitTestClass < matlab.unittest.TestCase
   methods(Test)
      function testTwoInputs(testCase)
          testCase.verifyTrue(getArea(10,22)==220,'Must return area of 10 X 22');
      end
      function testOneInput(testCase)
          testCase.verifyTrue(getArea(10)==100,'Must return area of a square');
          testCase.verifyTrue(getArea(22)==484,'Must return area of a square');
      end
   end
end
```

Class-Based 单元测试本身是一个类，所以可以声明一个该测试类的对象

Command Line

```
>> o = MyUnitTestClass
o =
  MyUnitTestClass with no properties.
```

通过 methods 函数可以查询单元测试 Framework 提供的验证方法，详述参见第0.6节

Command Line

```
>> methods(o)
Methods for class MyUnitTestClass:
MyUnitTestClass                fatalAssertEmpty
addTeardown                    fatalAssertEqual
applyFixture                   fatalAssertError
assertClass                    fatalAssertFail
assertEmpty                    fatalAssertFalse
assertEqual                    fatalAssertGreaterThan
assertError                    fatalAssertGreaterThanOrEqual
assertFail                     fatalAssertInstanceOf
assertFalse                    fatalAssertLength
assertGreaterThan              fatalAssertLessThan
assertGreaterThanOrEqual       fatalAssertLessThanOrEqual
...
```

调用 run 方法就可以运行所有的测试点，下面的结果表示，testTwoInput 和 testOneInput 都通过了测试。

Command Line

```
>> o.run()
Running MyUnitTestClass
..
Done MyUnitTestClass
```

```
----------
ans =
  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.0044918 seconds testing time.
```

runtests 函数也可以用来运行所有测试点：

Command Line

```
>> runtests('MyUnitTestClass');
Running MyUnitTestClass
..
Done MyUnitTestClass
```

0.8.2 MVC GUI 的基于类的单元测试

回顾第**??**, 第**??**和第**??**中的 GUI 程序的开发过程，程序写完后，为了验证，需要手动的点击 GUI 上的 button，然后观察 GUI 上的显示，来判断工作是否正常，这显然不是一个可靠持久的验证方法。本节举例介绍如何用 Class-Based 单元测试来程序化地验证第**??**, 第**??**和第**??**中的 MVC 代码和其产生的界面。

Class-Based 单元测试让我们能够写出面向对象风格的单元测试，比如我们可以把 MVC 的三个对象做为测试类的属性，那么所有的测试点都可以共享同一个 GUI 的界面，这样也自然解决了数据和图形 handle 在测试点之间传递的问题。

' 添加了 Fixture'

```
classdef MVCUnitTest < matlab.unittest.TestCase
    properties
        view                        % MVC 对象是整个测试的属性
        model
        controller
    end
    methods(TestClassSetup)         % 整个测试开始时运行一次
        function createMVC(testCase)
            testCase.model = Model(500);          % 初始化 model 对象
            testCase.view = View(testCase.model;)% 初始化 view 对象
            testCase.view.hfig.Visible = 'off';  % 图像设置为不可见
            testCase.controller = testCase.view.controlObj;
```

```
        end
    end
    methods(TestClassTeardown)   % 整个测试结束时运行一次
        function closeFigure(testCase)
            close(testCase.view.hfig);
        end
    end
end
```

在运行测试之前，我们首先需要构造 MVC 对象，这就是 createMVC 方法中的内容，该方法位于 methods(TestClassSetup) 的 block 中，表示该方法仅在整个测试的开始时执行一次。在 createMVC 方法中，账户初值设置为 500，并且把图像的设置为不可见，让其仅存在于后台[①]。在测试结束之后，我们需要做清理工作，即 closeFigure 函数，负责把隐藏的图像关闭，它放在 methods(TestClassTeardown) 的 block 中，该 block 中的方法只在整个测试结束时运行一次。TestClassSetup 和 TestClassTeardown 用来标记测试中的 Test Fixture，它们和第0.5中提到的 setupOnce 和 teardownOnce 功能类似。

第一个测试点验证 MVC 中的 Model，测试点以类方法的形式存在，要放在 Methods(Test) 的 block 中，首先调用 model 对象的 deposit 方法，存入 50 元，然后验证账户余额是 550，再在 550 的基础上取出 100 元，验证账户余额 450。

```
' 加入了第一个测试点 testModel'
classdef MVCUnitTest < matlab.unittest.TestCase
    properties
        view
        model
        controller
    end
    methods(TestClassSetup)
        function createMVC(testCase)
           testCase.model = Model(500);
           testCase.view = View(testCase.model);
           testCase.view.hfig.Visible = 'off';
           testCase.controller = testCase.view.controlObj;
        end
    end
    methods(TestClassTeardown)
        function closeFigure(testCase)
            close(testCase.view.hfig);
        end
    end
```

① 这样做的原因是，通常测试都是在后台自动运行的，如果自动运行的程序总是弹出窗口，会影响用户的其它工作流程，所以测试中，最好把 GUI 要产生的 Figure 设置成隐藏

```
    methods(Test)
        function testModel(testCase)                              % 测试 Model 的工作
            testCase.model.deposit(50);                           % 存 50
            testCase.verifyEqual(testCase.model.balance,550); % 验证余额 550
            testCase.model.withdraw(100);                         % 取 100
            testCase.verifyEqual(testCase.model.balance,450); % 验证余额 450
        end
    end
end
```

声明一个测试类对象，运行测试，通过。

“Command Line”

```
>> o = MVCUnitTest ;
>> o.run
Running MVCUnitTest
.
Done MVCUnitTest
----------

ans =
  TestResult with properties:

          Name: 'MVCUnitTest/testModel'
        Passed: 1
        Failed: 0
    Incomplete: 0
      Duration: 0.8636
       Details: [1x1 struct]

Totals:
   1 Passed, 0 Failed, 0 Incomplete.
   0.86362 seconds testing time.
```

增加第二个测试点，验证 View 的更新正常，这里需要考虑一个问题，该测试要不要接着上一个测试点的账户余额继续，比如 testModel 测试结束之后账户余额是 450，如果该测试点再存入 10 元，我们可以检查 View 上的显示是 460 元，但其实这不是一个好方法，设计各个测试点的一个基本原则是各个测试点之间没有相关性，因为我们无法始终保证各个测试点之间的有固定的运行顺序，或者有时候我们只希望运行各别的测试点。我们可以通过加入一个 Test Fixture，叫做 TestMethodSetup，来保证每个测试点运行之前，账户余额都被恢复到初始状态，即 500 元。和 TestMethodSetup 对称的 Test Fixture 叫做 TestMethodTeardown，它在每个测试点完成之后运行一次，这个例子中，没有什么工作需要放到 TestMethodTeardown 中。

' 加入了 TestMethodSetup Fixture'

```
classdef MVCUnitTest < matlab.unittest.TestCase
    properties
        view
        model
        controller
    end
    methods(TestClassSetup)
        function createMVC(testCase)
           testCase.model = Model(500);
           testCase.view = View(testCase.model);
           testCase.view.hfig.Visible = 'off';
           testCase.controller = testCase.view.controlObj;
        end
    end
    methods(TestClassTeardown)
        function closeFigure(testCase)
            close(testCase.view.hfig);
        end
    end
    methods(TestMethodSetup)
        function resetBalance(testCase)     %  每次测试开始前运行
           testCase.model.balance = 500;    %  账户余额复位 500
        end
    end
    methods(TestMethodTeardown)
        function some_cleaning_up(testCase) % 每次测试结束之后运行
                                            % 如有必要，做清理工作
        end
    end
    methods(Test)
        function testModel(testCase)
            testCase.model.deposit(50);
            testCase.verifyEqual(testCase.model.balance,550);
            testCase.model.withdraw(100);
            testCase.verifyEqual(testCase.model.balance,450);
        end
        function testView(testCase)          % 测试 View 的显示
            testCase.model.deposit(10);      % 存 10 元
            % 验证 GUI 上的显示 510
            testCase.verifyEqual(testCase.view.balanceBox.String,'510');
            testCase.model.withdraw(30);     % 取 30 元
            % 验证 GUI 上的显示 480
```

```
            testCase.verifyEqual(testCase.view.balanceBox.String,'480');
        end

    end
end
```

在 testView 测试点中，首先在 model 处存入 10 元，由于 View 对象监听了 model 对象的 balanceChanged 事件，所以它会自动把显示更新成账户中余额，通过检查编辑框对象的 String 属性可以验证。然后再从 model 处取出 30 元，验证 GUI 界面上做出相应的变化，再次运行[①]，测试通过。

"Command Line"

```
>> o.run
Running MVCUnitTest
..
Done MVCUnitTest
----------

ans =

  1x2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   0.94354 seconds testing time.
```

添加第三个测试验证 controller 的功能：通过设置编辑框中 String 属性，模拟用户在输入框内键入数字，然后在测试中调用 button 的回调函数：drawbutton 和 depositbutton，最后验证 View 处的更新正常。

" 在类中添加新的测试点 testController"

```
classdef MVCUnitTest < matlab.unittest.TestCase
...
        function testController(testCase)
            testCase.view.numBox.String = '30';                    % 模拟用户输入
```

① 如果你使用的 MATLAB 支持 Auto Update(第??节)，那么更新过类定义之后（这里是添加了类方法），无需 clear class，可以直接运行 o.run，低于 R2014b 版本 MATLAB 需要 clear classes 让类的定义更新。

```
            testCase.controller.callback_drawbutton();                    % 直接调回调函数
            testCase.verifyEqual(testCase.view.balanceBox.String,'470'); % 验证 GUI 显示

            testCase.view.numBox.String = '50';
            testCase.controller.callback_depositbutton();
            testCase.verifyEqual(testCase.view.balanceBox.String,'520');
        end
...
```

添加第四个测试，验证 View 对象中的控件上确实存在回调函数，并且工作正常。在这个测试中，我们通过 get 函数直接获得 button 对象的 callback 函数，然后直接执行它们，这相当于用户用鼠标的点击 button，然后验证 view 中显示正常。

" 在类中添加新的测试点 testCallBack"

```
classdef MVCUnitTest < matlab.unittest.TestCase
...
     function testCallBack(testCase)
            testCase.view.numBox.String = '10';                           % 模拟用户输入
            callback = get(testCase.view.drawButton,'callback');          % 获得回调函数
            callback();                                                   % 调用回调函数
            testCase.verifyEqual(testCase.view.balanceBox.String,'490');  % 验证 GUI 显示

            testCase.view.numBox.String = '20';
            callback = get(testCase.view.depositButton,'callback');
            callback();
            testCase.verifyEqual(testCase.view.balanceBox.String,'510');
        end
...
```

目前为止，我们一共给该 Class-Based 的单元测试添加了四个测试点，和一个 Test Fixture，它们的运行顺序是：在整个测试开始时，先运行 TestClassSetup，每个测试点开始和结束时，分别运行 TestMethodSetup，TestMethodTeardown，整个测试结束后，运行 TestClassTeardown。

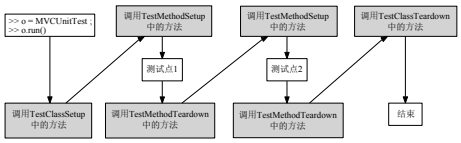


图 14　Class-Based 测试中 Fixture 和测试点的运行顺序

本节通过一个 Class-Based 单元测试，介绍了程序化测试 GUI 界面的基本思路：即通过程序的方式模拟用户的输入点击，最后通过验证对象的内部的状态来检测结果。

作者简介

徐潇

MathWorks 开发部 MATLAB 架构 C++ 高级软件工程师。计算物理学博士，研究方向为电子结构计算、密度泛函算法开发；计算机硕士，研究方向为图像处理。2004 年，开始使用 MATLAB，在科研编程中遇到了开发大型程序难以维护的困难，花了很多时间用于改进程序但效果总不尽如人意。2009 年，开始使用 MATLAB 面向对象编程，发现工程进度被迅速加快，于是萌生了写一本介绍 MATLAB 面向对象编程书的念头，这就是 **«MATLAB 面向对象设计编程：从入门到设计模式 »** 的起源。2011 年，在美国取得博士学位之后入职 MathWorks，从理科科研工作者和多年的 MATLAB 爱好者，成为一名 MATLAB 语言的设计开发和实现的软件工程师。2016 年，作者在 MATLAB 中文论坛开辟了技术专栏，和大家分享最新的行业应用技术和 MATLAB 编程理念，旨在推动软件工程中的现代手段在 MATLAB 科学工程计算项目中的使用，帮助科学家和工程师们更有效地解决复杂的科研问题。本书凝结了作者多年的科研和工作经验以及对 MATLAB 语言的理解，希望能对各种规模的科学工程计算项目的 MATLAB 使用者有所启发。

更多 MATLAB 中文论坛技术文章

用MATLAB构造可靠的大型工程项目系列

(1) MATLAB 性能测试框架
(2) 物联网专题——Internet of Things
- Thingspeak 能为我们做什么
- 如何用 Browser 向 Thingspeak 推送数据
- 如何用 Postman 向 Thingspeak 获取和推送数据
- 如何在 Thingspeak 中添加 MATLAB Plot
- 如何使用 Javascript 获得 Thingspeak 频道数据
- 如何在 Thingspeak 中使用 JS Plugin
- NodeMCU 基础篇
- NodeMCU 和 Thingspeak
- 如用 Thingspeak 和 IFTTT 让物联网硬件发邮件
- 如何用 Thingspeak 控制联网的硬件

(3) MATLAB 中文论坛常见问题归纳
(4) 为什么要基于模型设计?
(5) MATLAB 单元测试
(6) 对函数的输入进行检查和解析
(7) MATLAB 映射表数据结构
(8) MATLAB table 数据结构
(9) ...