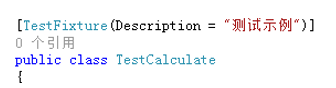
**前言：**

  本文主要是介绍了Nunit的基本使用，其中参详了很多已有的文章，由于最近要使用其进行测试，所以对网上的文章做了下整理，同时加入了一些自己的实践。

**NUnit的属性**

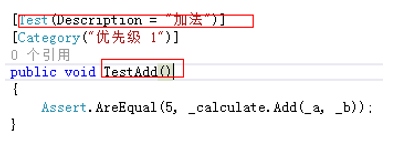
**TestFixture**

它标记一个类包含测试，申明该类是用来测试的。一般用在class的定义之前；



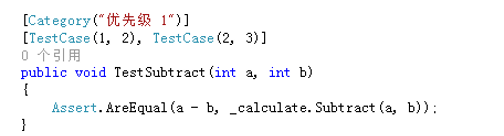
**Test**

         一般是放在method之前，表示对该方法的测试，中添加Description参数可以给我们测试的功能添加描述信息



**TestCase(arguments)**

    属性标记有参数无返值方法为测试方法(泛型方法一样标记),想要多次测试可用逗号隔开([TestCase(1,2), TestCase(2,3)])



TestCaseAttribute还提供一些列其他属性：

　　　　Description：描述测试用例详情

　　　　ExpectedException：指定期望抛出的异常

　　　　ExpectedExceptionName：指定期望抛出异常的全名

　　　　ExpectedMessage：指定期望抛出异常的信息

　　　　Explict：设置为true标记测试用例为显示执行，用Reason来诠释缘由

　　　　Ignore：设置为true用于指定忽略测试用例，用Reason来诠释缘由

　　　　IgnoreReason：指定忽略测试用例及缘由，等同于同时使用Ignore && Reason。注意：本特性会在以后版本中移除

　　　　MatchType：**MessageMatch**枚举类型，指定测试期望信息（详情见[ExpectedExceptionAttribute](file:///C:\\Program%20Files\\NUnit%202.6.3\\doc\\exception.html)）

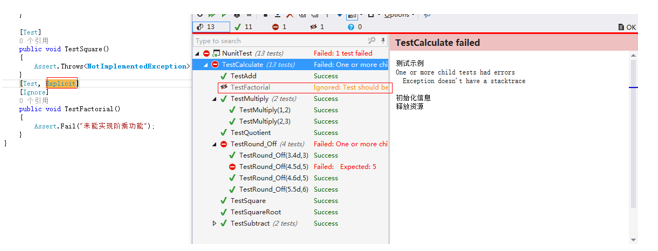
　　　　Reason：指定不执行测试用例的缘由，结合Ignore或Explicit特性使用

　　　　Result：指定测试方法期望的返回值，返回值类型必须是可兼容的

　　　　TestName：为测试用例指定一个名称，如果不指定会根据测试方法及参数自动生成。

**Explicit**

   属性标记测试方法需要在UI界面显式执行，如果不想对某个方法进行单元测试,只是在它被选中时才进行测试的话,可以调用该特性。

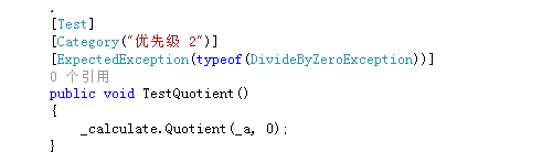


**Ignore**

    属性标记一个测试方法或一个测试类被忽略，如果测试类被忽略，其内中的测试方法也会被忽略。

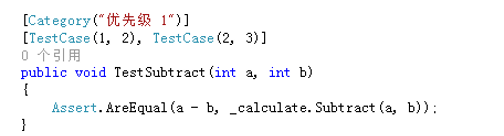
**ExpectedException(Type)**

属性标记测试方法在运行时抛出一个期望的异常，如果是则测试通过，否则不通过



**Category("")**

属性标记用于将测试分类(便于只测试需要的类别)，可在方法与类上进行标记，在NUnit-GUI界面的Categories选项卡中对要参与参数的类别进行选择，Run时仅测试该类别的测试。如果均不选，则默认全部测试



**TestFixtureSetUp**

属性标记方法为类级别设置(初始化)方法，在整个测试类中执行一次初始化，所有的测试方法共享初始化数据，标记此属性的方法可以简单的将其想象为一个初始化器，就行类的构造函数一样

**TestFixtureTearDown**

属性标记方法为类级别拆卸方法，在整个测试类中执行一次拆卸.当测试类中的所有测试方法执行完成，就会执行拆卸方法，用于清除数据、释放资源，可以将其想象为一个析够函数，在测试完成后对测试过程中使用的资源进行回收

**TearDown**

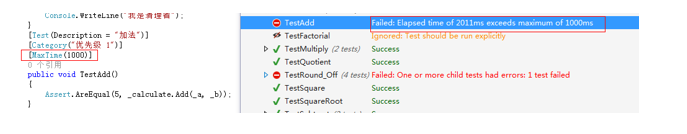
属性标记方法为函数级别的拆卸方法,在执行完每个测试方法后，执行该拆卸方法。一个测试类可以仅有一个TearDown/Setup/TestFixtureSetUp/TestFixtureTearDown方法。如果有多个定义，测试类也会编译成功，但是测试时不会运行这些标记过的方法

**SetUp**

属性标记方法为函数级别的设置方法,在执行每个测试方法前，执行该设置方法

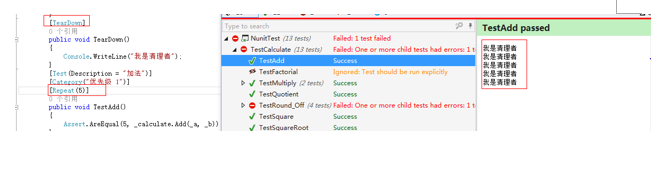
**[Maxtime]/[Timeout]**

属性标记测试用例的最大执行时间，前者超时时不取消测试，而后者会强行中断，用法如：[Test, Maxtime(2000)],[Test, Timeout(2000)]。



**Repeat**

属性标记测试方法重复执行多少次，如：[Test, Repeat(5)]。

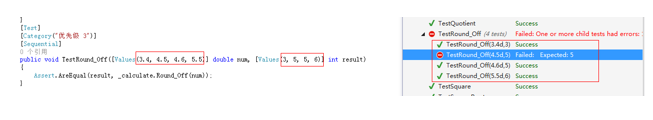


**[RequiresMTA]/[RequiresSTA]/[RequiresThread]**

属性标记测试用例必须的在多线程、单线程、独立的线程状态下运行

**Values**

属性标记测试用例的参数，以参数的形式传入一组值，NUnit会把这组值分解成相应数量的子测试。当测试用例的2个参数都使用[Values]进行标记，NUnit默认生成2组数量乘积的用例，需要使用[Sequential]标记测试用例才能按顺序生成一一对应的n(n=2组中最大数组长度)个子测试用例



**ValueSource**

属性标记测试用例的参数，指定参数的数据源来自哪里，在使用[ValueSource]指定数据源时，该数据源必须实现了IEnumerable接口，数据源可以是属性、无参方法、实例或静态成员

**Nunit常用类和方法**

**1、Assert(断言)：如果断言失败，方法将没有返回，并且报告一个错误。**

**1）、测试二个参数是否相等**

Assert.AreEqual;

Assert.AreEqual;

**2）、测试二个参数是否引用同一个对象**

Assert.AreSame;

Assert.AreNotSame;

**3）、测试一个对象是否被一个数组或列表所包含**

Assert.Contains;

**4）、测试一个对象是否大于另一个对象**

Assert.Greater;

**5）、测试一个对象是否小于另一个对象**

Assert.Less;

**6）、类型断言：**

Assert.IsInstanceOfType;

Assert.IsAssignableFrom；

**7）、条件测试：**

Assert.IsTrue;

Assert.IsFalse;

Assert.IsNull;

Assert.IsNotNull;

Assert.IsNaN;用来判断指定的值是否为数字。

Assert.IsEmpty;

Assert.IsNotEmpty;

Assert.IsEmpty;

Assert.IsNotEmpty;

**8）、其他断言：**

Assert.Fail;方法为你提供了创建一个失败测试的能力，这个失败是基于其他方法没有封装的测试。对于开发你自己的特定项目的断言，它也很有用。

Assert.Pass;强行让测试通过

**2、字符串断言(StringAssert)：提供了许多检验字符串值的有用的方法**

StringAssert.Contains;

StringAssert.StartsWith;

StringAssert.EndsWith;

StringAssert.AreEqualIgnoringCase;

**3、CollectionAssert类**

CollectionAssert.AllItemsAreInstancesOfType;集合中的各项是否是某某类型的实例

CollectionAssert.AllItemsAreNotNull:集合中的各项均不为空

CollectionAssert.AllItemsAreUnique;集合中的各项唯一

CollectionAssert.AreEqual;两个集合相等

CollectionAssert.AreEquivalent;两个集合相当

CollectionAssert.AreNotEqual;两个集合不相等

CollectionAssert.AreNotEquivalent;两个集合不相当

CollectionAssert.Contains;

CollectionAssert.DoesNotContain;集合中不包含某对象

CollectionAssert.IsSubsetOf:一个集合是另外一个集合的子集

CollectionAssert.IsNotSubsetOf:一个集合不是另外一个集合的子集

CollectionAssert.IsEmpty;集合为空

CollectionAssert.IsNotEmpty;集合不为空

CollectionAssert.IsOrdered;集合的各项已经排序

**4、FileAssert**

FileAssert.AreEqual;

FileAssert.AreNotEqual;

**5、DirectoryAssert**

DirectoryAssert.AreEqual;

DirectoryAssert.AreNotEqual;

DirectoryAssert.IsEmpty;

DirectoryAssert.IsNotEmpty;

DirectoryAssert.IsWithin;

DirectoryAssert.IsNotWithin;

**Nunit测试工具下载：**

**ReSharper集成：**

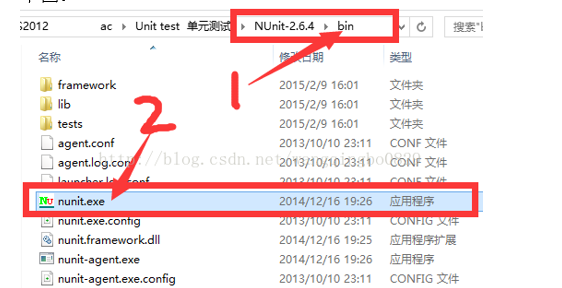
 <http://www.jetbrains.com/resharper/download/>

**Nunit工具：**

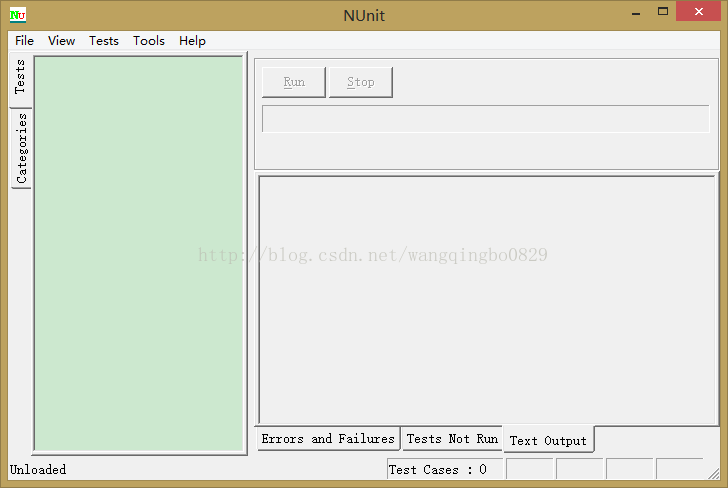
<https://github.com/nunit/nunitv2/releases/tag/2.6.4>可根据自己的需要进行下载

**Nunit工具使用**

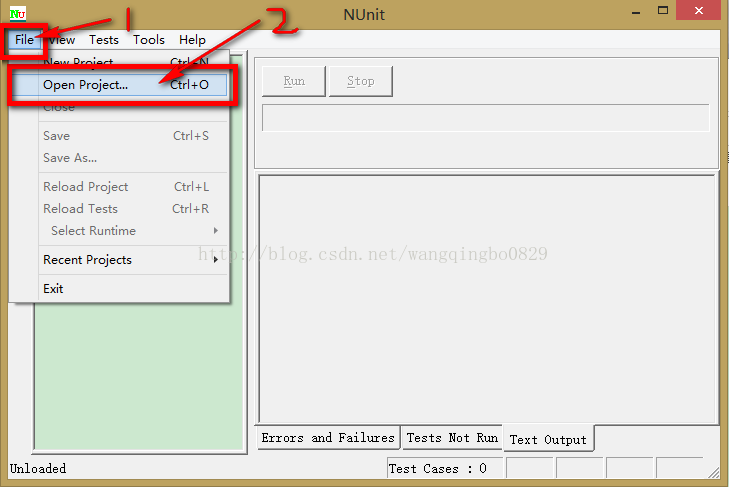
1.下载解压后，得到NUnit-2.6.4文件夹，打开文件夹中的bin文件，如下图：



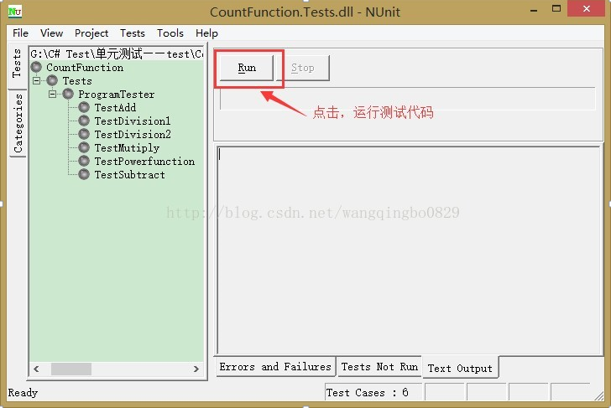
2.选中nunit.exe发送快捷方式到桌面，双击刚发到桌面上“nunit.exe-快捷方式”图标，进入Nunit工具运行界面，如图：



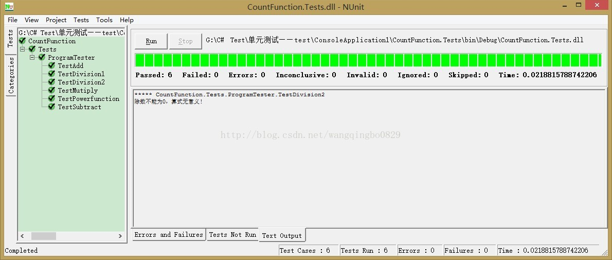
3如图，通过菜单中File的下拉列表中的“Open Project Ctrl+O”项，找到为**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "_blank)**而写的测试项目的.dll文件（.exe文件也可），添加到NUnit中。



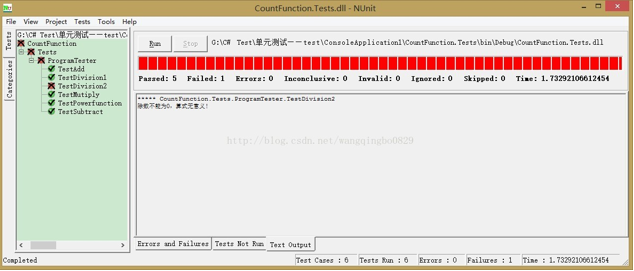
4.下图便是加载后的展示。直接单价Run按钮，运行测试代码。



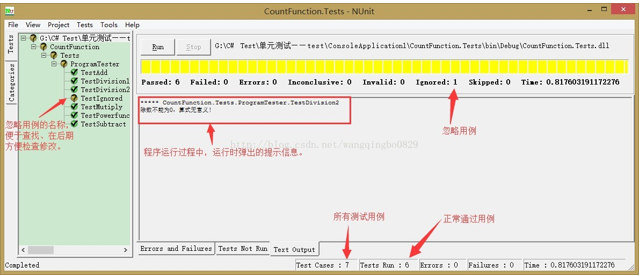
5.用例测试，完全成功通过的示例图为：



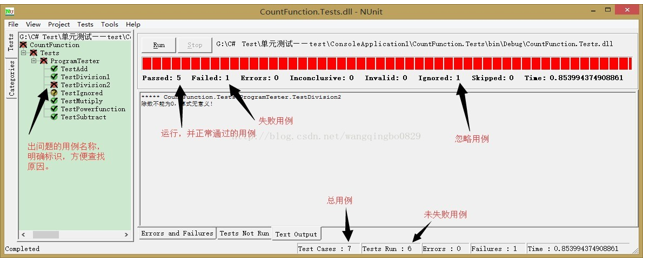
6.存在测试失败时，图示：



7.存在被忽略用例，不存在出错用例时：

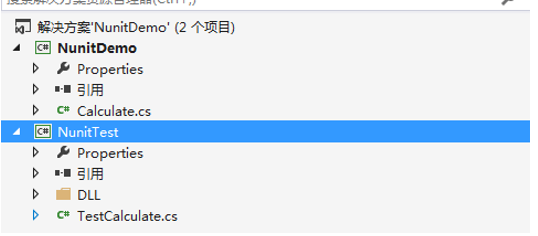


8.混合用例（同时存在忽略用例+出错用例）：

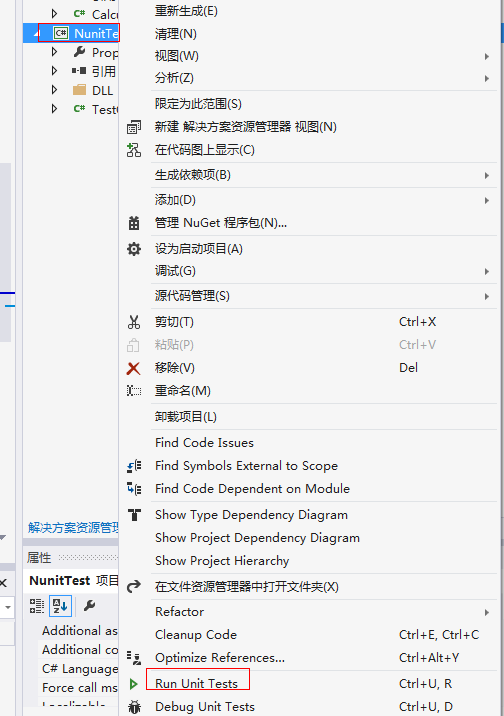


**ReSharper使用单元测试**

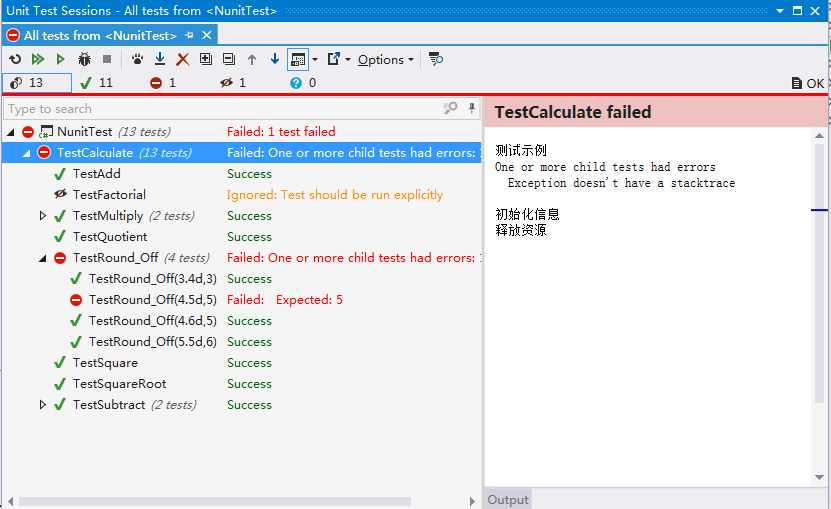
　　1.以下是一个需要测试的类库和一个 Nunit测试单元。



      2.右键 Nunit测试单元，选择Run Nunit tests即可



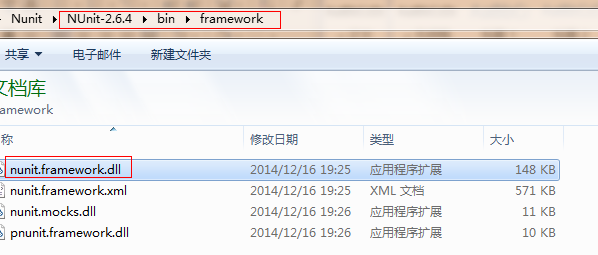
 3.选择之后就会看到如下界面：



这个界面的使用和Nunit Ui界面的使用类似，不多讲。同时你可能看到有一个Dubug Nunit Tests 选项，他的功能就类似代码调试，可以逐步调试测试用例调用的过程，记得打断点。

**注意**

  在测试单元需要引入 这个包，这个包的来源可以在你下载的Nunit工具中找到。



**学习资料：**

<http://www.cnblogs.com/kim01/default.aspx?page=3>这个是对官网章节的翻译，当然你也可以直接去官网查看。

# [单元测试与Nunit的基本使用](http://www.cnblogs.com/zwt-blog/p/5788222.html)

**一、单元测试是什么**

单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，一般来说，要根据实际情况去判定其具体含义，如C语言中单元指一个函数，C#里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

单元测试（模块测试）是开发者编写的一小段代码，用于检验被测代码的一个很小的、很明确的功能是否正确。通常而言，一个单元测试是用于判断某个特定条件（或者场景）下某个特定函数的行为。

**二、为什么需要单元测试**

在我们现在的编程思维中一直都是编码=>编译=>调试，一直循环，直到要处理的功能完成，每一个功能完成都是如此，且有的功能是严重依赖于上一个功能。在如此处理中存在几个问题。

1. 编译通过后，运行程序出现的bug难以定位。
2. 修改一个bug，容易引进其他bug。
3. Bug越到后期发现，越难以修改。
4. 后期系统的复杂性，导致代码难以修改和重构，使得系统难以维护。
5. 开发人员常认为编译功过，进行了几次手工测试就等于测试通过（认为详细的测试是测试人员的工作，非开发人员的工作）。
6. 在完全依赖外部系统的情况下，难以进行有效的测试。
7. 手工测试效率低下，针对性不强，测试不能重用。

有了单元测试在开发过程中起到的作用。

1. 大大节约了测试和修改的时间，有效且便于测试各种情况。
2. 能快速定位bug（每一个测试用例都是具有针对性）。
3. 能使开发人员重新审视需求和功能的设计（难以单元测试的代码，就需要重新设计）。
4. 强迫开发者以调用者而不是实现者的角度来设计代码，利于代码之间的解耦。
5. 自动化的单元测试能保证回归测试的有效执行。
6. 使代码可以放心修改和重构。
7. 测试用例，可作为开发文档使用（测试即文档）。
8. 测试用例永久保存，支持随时测试。

既然单元测试有这些好处，为什么我们不去用呢。可以归纳为以下几个理由。

1. 对单元测试存在的误解，如：单元测试属于测试工作，应该由测试人员来完成，所以单元测试不属于开发人员的职责范围。答：虽然单元测试虽然叫做"测试"，但实际属于开发范畴，应该由开发人员来做，而开发人员也能从中受益。
2. 没有真正意识到单元测试的收益，认为写单元测试太费时，不值得。

答：在开发时越早发现bug，就能节省更多的时间，降低更多的风险。单元测试先期要编写测试用例，是需要多耗费些时间，但是后面的调试、自测，都可以通过单元测试处理，不用手工一遍又一遍处理。实际上总时间被减少了。

1. 项目经理或技术主管没有要求写单元测试，所以不用写。

答：写单元测试应该成为开发人员的一种本能，开发本身就应该包含单元测试。

1. 不知道有单元测试这回事，不知道如何用。经过这篇文档的说明，就基本知道如何处理单元测试。

**结论：**

只进行手工测试，只是临时性的单元测试，代码测试覆盖率要超过70%都很困难，未覆盖的代码可能遗留大量的细小的错误，这些错误还会互相影响，当bug暴露出来的时候难于调试，大幅度提高后期测试和维护成本。可以说，进行充分的单元测试，是提高软件质量，降低开发成本的必由之路。

要进行充分的单元测试，应专门编写测试代码，并与产品代码隔离。比较简单的办法是为产品工程建立对应的测试工程，为每个类建立对应的测试类，为每个函数（很简单的除外）建立测试函数。

单元测试是由程序员自己来完成，最终受益的也是程序员自己。可以这么说，程序员有责任编写功能代码，同时也就有责任为自己的代码编写单元测试。执行单元测试，就是为了证明这段代码的行为和我们期望的一致。

对于程序员来说，如果养成了对自己写的代码进行单元测试的习惯，不但可以写出高质量的代码，而且还能提高编程水平。

**三、单元测试工具。**

在.Net平台有三种单元测试工具，分别为MS Test、NUnit、Xunit.Net。

1.MS Test为微软产品，集成在Visual Studio 2008+工具中。

2.NUnit为.Net开源测试框架（采用C#开发），广泛用于.Net平台的单元测试和回归测试中,官方网址([www.nunit.org](http://www.nunit.org))。

3.XUnit.Net为NUnit的改进版。

（以下主要讲解NUnit的使用，会了NUnit其他2个测试工具也能快速熟悉）。

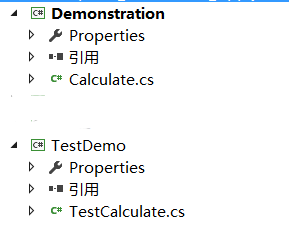
任何xUnit工具都使用断言进行条件的判断，NUnit自然也不例外，与其它的xUnit（如JUnit、phpUnit、pythonUnit）相比，由于大量使用了Generic、Attribute等语言特征，NUnit提供了更为方面、灵活的测试方法，下面先介绍一下断言。

NUnit一共有五个断言类，分别是Assert、StringAssert、FileAssert、DirectoryAssert、CollectionAssert，它们都在NUnit.Framework命名空间，其中Assert是常用的，而另外四个断言类，顾名思义，分别对应于字符串的断言、文件的断言、目录的断言、集合的断言。理论上，仅Assert类就可以完成所有条件的判断，然而，如果合理的运用后面的四个断言，将使代码更加简洁、美观，也更加便于理解和维护。

**四、NUnit的使用。**

本处演示所使用的NUnit版本为2.6.4，若要使用最新版可以去官网下载。

首先创建一个类库项目（也可以是其他项目），然后创建一个Test+类库名称的项目（也可以是项目名称+Test），用于代表是测试工程。如下图：



Demonstration项目中含有一个计算功能类，对应的测试项目含有一个测试计算类，一个计算功能类中方法可能需要多个测试用例来完成检测。如下展示出了2个类的代码：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifView Code

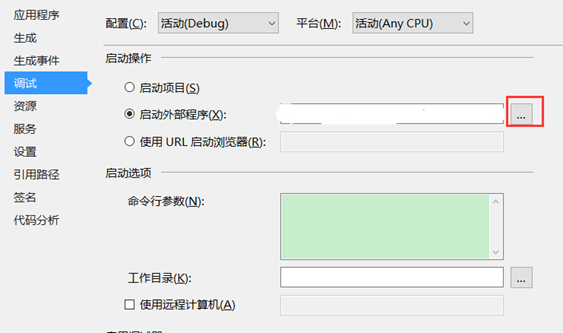
在粗略看了代码后，下面就详细说明相应的测试标记（属性）的用法。

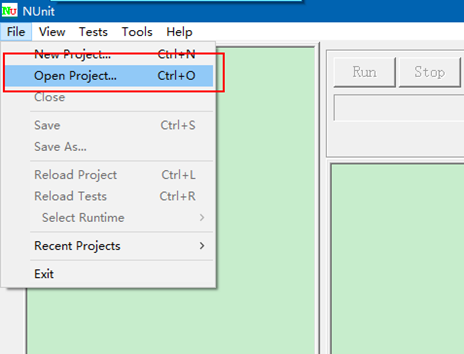
1. [TestFixture(arguments)]属性标记类为测试类，若没有填写参数，则测试类必须含有无参构造函数，否则需要相应的有参构造函数。也可以多个测试[TestFixture(1), TestFixture("a")]
2. [Test]属性标记方法为测试方法，中添加Description参数可以给我们测试的功能添加描述信息。
3. [TestCase(arguments)]属性标记有参数无返值方法为测试方法(泛型方法一样标记),想要多次测试可用逗号隔开([TestCase(1,2), TestCase(2,3)])。
4. [TestCase(arguments,Result = value)属性标记带参数与返回值的方法为测试方法，执行的时候把预期的返回值也告诉NUnit，如果返回值不对，测试同样无法通过。
5. [Suite]（测试套件,仅对属性与索引器标记有效）：可以将多个测试类组合到一起，同时执行多个测试。本版本的开发人员的一个信念就是减少这个的需要，可以使用[Category]来替代它。
6. [Explicit]属性标记测试方法需要在UI界面显式执行，如果不想对某个方法进行单元测试,只是在它被选中时才进行测试的话,可以调用该特性。
7. [Ignore]属性标记一个测试方法或一个测试类被忽略，如果测试类被忽略，其内中的测试方法也会被忽略。
8. [ExpectedException(Type)]属性标记测试方法在运行时抛出一个期望的异常，如果是则测试通过，否则不通过。
9. [Category("")]属性标记用于将测试分类(便于只测试需要的类别)，可在方法与类上进行标记，在NUnit-GUI界面的Categories选项卡中对要测试种类进行添加，Run时仅测试该类别的测试。
10. [TestFixtureSetUp]属性标记方法为类级别设置(初始化)方法，在整个测试类中执行一次初始化，所有的测试方法共享初始化数据。
11. [TestFixtureTearDown]属性标记方法为类级别拆卸方法，在整个测试类中执行一次拆卸.当测试类中的所有测试方法执行完成，就会执行拆卸方法，用于清除数据、释放资源。
12. [TearDown]属性标记方法为函数级别的拆卸方法,在执行完每个测试方法后，执行该拆卸方法。一个测试类可以仅有一个TearDown/Setup/TestFixtureSetUp/TestFixtureTearDown方法。如果有多个定义，测试类也会编译成功，但是测试时不会运行这些标记过的方法。
13. [SetUp]属性标记方法为函数级别的设置方法,在执行每个测试方法前，执行该设置方法。
14. 每执行一次Run，就是new一个新的实例在测试。
15. [Maxtime]/[Timeout]属性标记测试用例的最大执行时间，前者超时时不取消测试，而后者会强行中断，用法如：[Test, Maxtime(2000)],[Test, Timeout(2000)]。
16. [Repeat]属性标记测试方法重复执行多少次，如：[Test, Repeat(100)]。
17. [RequiresMTA]/[RequiresSTA]/[RequiresThread]属性标记测试用例必须的在多线程、单线程、独立的线程状态下运行。
18. [Values]属性标记测试用例的参数，以参数的形式传入一组值，NUnit会把这组值分解成相应数量的子测试。当测试用例的2个参数都使用[Values]进行标记，NUnit默认生成2组数量乘积的用例，需要使用[Sequential]标记测试用例才能按顺序生成一一对应的n(n=2组中最大数组长度)个子测试用例。
19. [ValueSource]属性标记测试用例的参数，指定参数的数据源来自哪里，在使用[ValueSource]指定数据源时，该数据源必须实现了IEnumerable接口，数据源可以是属性、无参方法、实例或静态成员。

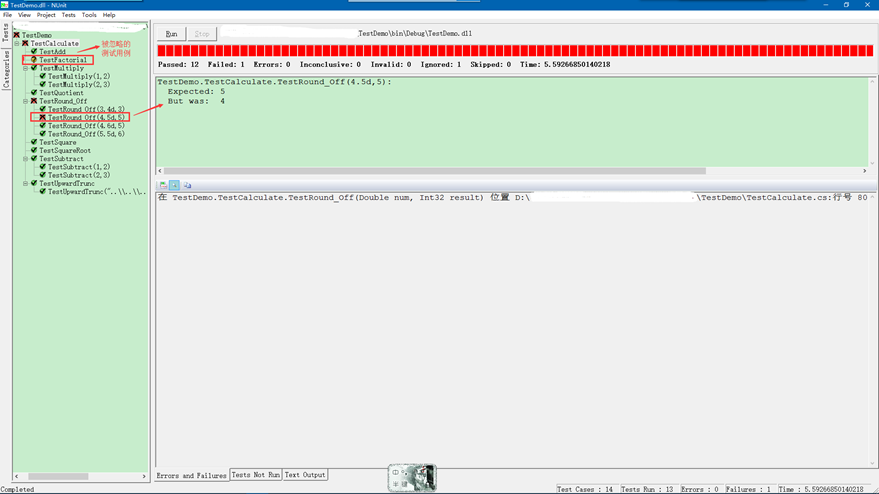
更多属性标记与详细说明，可以查阅NUnit官网提供的说明文档。一个方法的测试可能要写很多个测试用例,这都是正常的，如果一个测试用例包含多个断言，那些紧跟失败断言的断言都不会执行，因为通常每个测试方法最好只有一个断言。

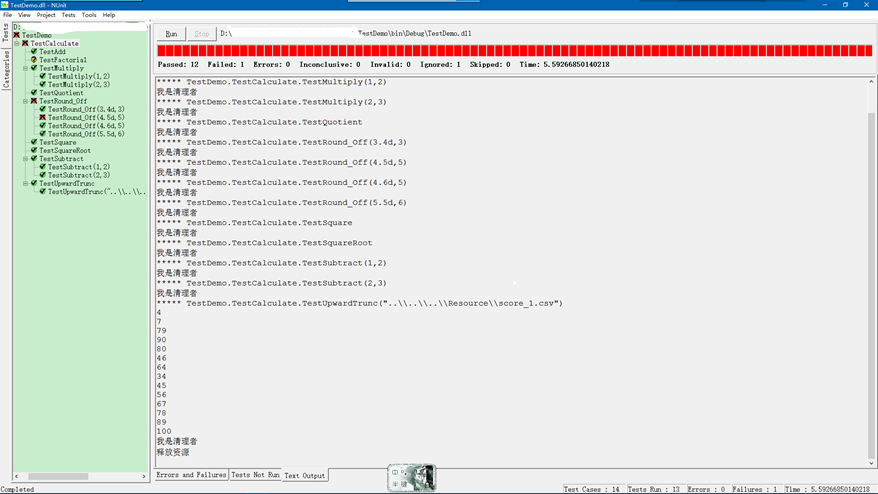
在运行单元测试时有3种方式分别为：

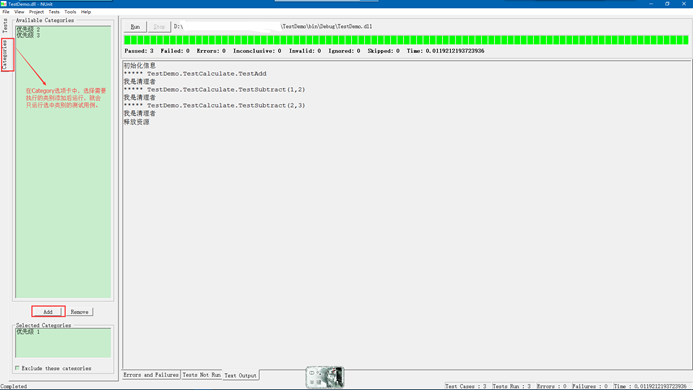
1. 把测试工程的属性=>调试=>启动外部程序，设置为NUnit运行程序。在启用调试时会启动NUnit界面程序，但NUnit界面没有测试用例的信息，需要自己添加在File=>Open Project->文件资源管理器，找你的测试工程类库或程序添加即可。点击Run运行，根据选中的节点运行该节点下所有的子测试用例（该测试可进行调试）。如下图：











以上的图片展示了运行错误界面和运行输出界面。在测试用例的节点中绿色'√'代表通过，黄色'√'代表忽略，红色'×'代表失败。

1. 直接启动NUnit界面程序，在File=>Open Project->文件资源管理器，添加测试工程类库或程序,点击相应的节点进行Run测试，NUnit会根据类库或程序生成更新，自动更新界面中测试用例节点,但运行的测试用例不能进行调试。效果图与①中的效果一样。
2. 在Visual Studio 2010+的IDE中以插件的方式集成NUnit测试工具，直接在测试工程中点击鼠标右键，运行测试即可。或者在VS菜单栏的测试中运行NUnit测试。集成与运行效果图在"第五节"中展示。

**五、Nunit常用类和方法**

1、Assert(断言)：如果断言失败，方法将没有返回，并且报告一个错误。

1）、测试二个参数是否相等

Assert.AreEqual;

Assert.AreEqual;

2）、测试二个参数是否引用同一个对象

Assert.AreSame;

Assert.AreNotSame;

3）、测试一个对象是否被一个数组或列表所包含

Assert.Contains;

4）、测试一个对象是否大于另一个对象

Assert.Greater;

5）、测试一个对象是否小于另一个对象

Assert.Less;

6）、类型断言：

Assert.IsInstanceOfType;

Assert.IsAssignableFrom；

7）、条件测试：

Assert.IsTrue;

Assert.IsFalse;

Assert.IsNull;

Assert.IsNotNull;

Assert.IsNaN;用来判断指定的值是否为数字。

Assert.IsEmpty;

Assert.IsNotEmpty;

Assert.IsEmpty;

Assert.IsNotEmpty;

8）、其他断言：

Assert.Fail;方法为你提供了创建一个失败测试的能力，这个失败是基于其他方法没有封装的测试。对于开发你自己的特定项目的断言，它也很有用。

Assert.Pass;强行让测试通过

2、字符串断言(StringAssert)：提供了许多检验字符串值的有用的方法

StringAssert.Contains;

StringAssert.StartsWith;

StringAssert.EndsWith;

StringAssert.AreEqualIgnoringCase;

3、CollectionAssert类

CollectionAssert.AllItemsAreInstancesOfType;集合中的各项是否是某某类型的实例

CollectionAssert.AllItemsAreNotNull:集合中的各项均不为空

CollectionAssert.AllItemsAreUnique;集合中的各项唯一

CollectionAssert.AreEqual;两个集合相等

CollectionAssert.AreEquivalent;两个集合相当

CollectionAssert.AreNotEqual;两个集合不相等

CollectionAssert.AreNotEquivalent;两个集合不相当

CollectionAssert.Contains;

CollectionAssert.DoesNotContain;集合中不包含某对象

CollectionAssert.IsSubsetOf:一个集合是另外一个集合的子集

CollectionAssert.IsNotSubsetOf:一个集合不是另外一个集合的子集

CollectionAssert.IsEmpty;集合为空

CollectionAssert.IsNotEmpty;集合不为空

CollectionAssert.IsOrdered;集合的各项已经排序

4、FileAssert

FileAssert.AreEqual;

FileAssert.AreNotEqual;

5、DirectoryAssert

DirectoryAssert.AreEqual;

DirectoryAssert.AreNotEqual;

DirectoryAssert.IsEmpty;

DirectoryAssert.IsNotEmpty;

DirectoryAssert.IsWithin;

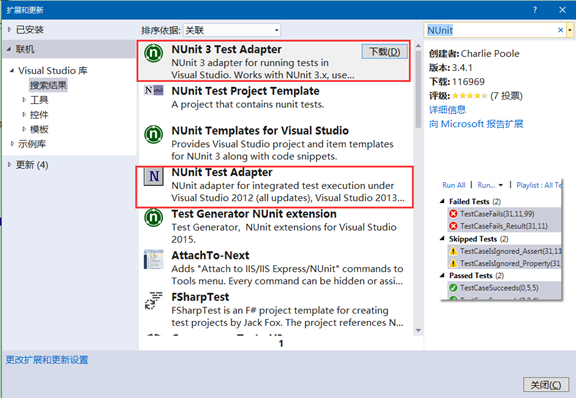
DirectoryAssert.IsNotWithin;

**六、NUnit集成到VS中的使用。**

在使用NUnit-GUI处理运行测试用例，是不是感觉比较麻烦，还要使用外部的NUnit应用程序，有没有简单点的最好能够跟VS开发工具紧密结合的方式来进行NUnit单元测试呢？答案是肯定的，有2种方式。

1.我们在VS中选择工具菜单栏下的扩展和更新，选择联机并在搜索框中输入NUnit。出现如下图的信息，有2个版本的Nunit适配器，分别为NUnit 3.x（最新版为3.4.1）和NUnit 2.x（最新版为2.6.4）,都支持Visual Studio 2012+。若想在VS2010中集成，需要安装NUnit 2.6.4安装包(可在官网下载)与VS2010 NUnit整合插件(下载地址：

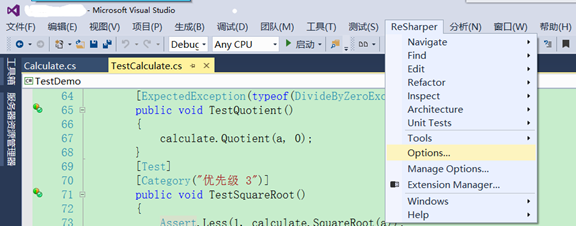
http://visualstudiogallery.msdn.microsoft.com/c8164c71-0836-4471-80ce-633383031099)，下载安装完毕就能在 VS2010 的视图=>其他窗口中看到 Visual Nunit(或使用快捷键Ctrl + F7），打开该视图，将之拖到合适的位置。

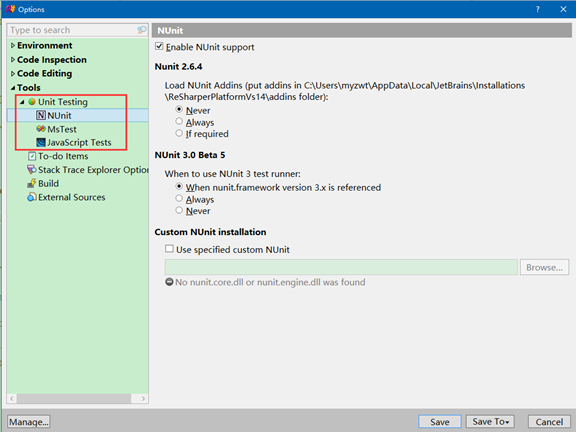


下载安装NUnit Test Adapter后关闭VS，重启一下就好了，我们打开类库项目中的TestCalculate类，在右键弹出的菜单中点击运行测试。运行结束后，会在左侧的测试资源管理器当中显示本次操作的结果。

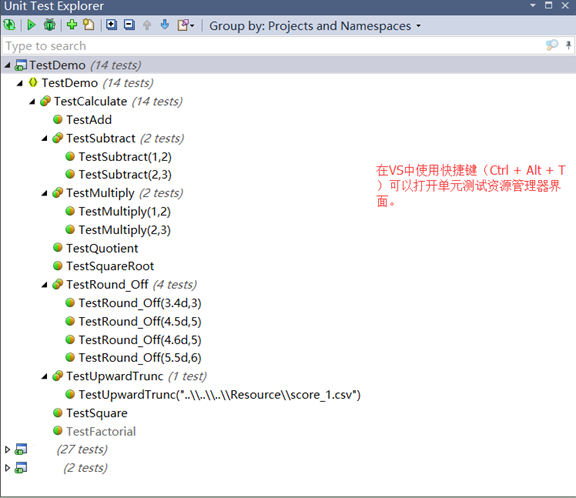


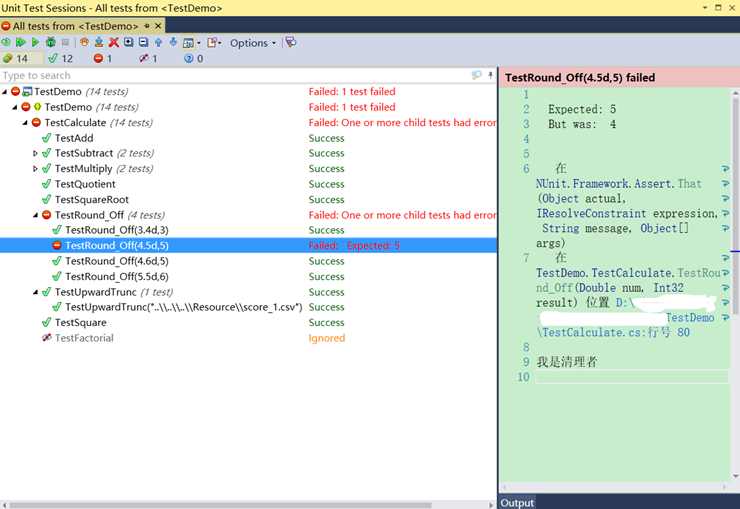
2.通过ReSharper工具处理NUnit的单元测试，在VS2010+中安装了ReSharper开发插件，ReSharper内中自带支持NUnit与MS Test这2个单元测试工具，只要你的测试工程中引用了相应的单元测试类库（如nunit.Framework.dll）、以及含有测试用例。通过鼠标右键或快捷键(Ctrl + T,R)，就可以运行单元测试，也可以进行单元测试调试，ReSharper选项图与运行效果如下图。











**七、后续**

上面列出只能单元测试的基本使用，未能说明对Mock等其他功能的使用，也没有解释对难以单元测试的代码进行重新设计的说明，需要后期深入了解才能列出相应的文档说明。能够更好的使用单元测试才能更好的使用TDD（测试驱动开发）来开展项目，TDD测试驱动开发是测试先行（此测试是单元测试）、是极限编程的一个重要特点，它以不断的测试推动代码的开发，既简化了代码，同时也保证了软件指令，另一方面说编写的测试用例将成为重要文档（可以作为SDK提供给开发者，测试即文档）。

-----------------以上内容是根据博客园其他博客的说明与Nunit官方文档，以及自己测试使用，进行了整理说明。----------------------------

# [单元测试2](http://www.cnblogs.com/zwt-blog/p/5881415.html)

**单元测试的定义：**一个单元测试是一段自动化的代码，这段代码调用被测试的工作单元，之后对这个单元的单个最终结果的某些假设进行检验。单元测试几乎都是用单元测试框架编写的。单元测试容易编写，能快速运行。单元测试可靠、可读，并且可维护。只要产品代码不发生变化，单元测试的结果是稳定的。

成功进行TDD的三种核心技能：知道如何编写优秀的测试、在编码前编写测试、以及良好的测试设计。

**测试命名和位置的基本规则**

|  |  |
| --- | --- |
| 测试对象 | 测试方创建的对象 |
| 项目 | 创建一个名为"项目名+.UnitTests"的测试项目 |
| 类 | 对应被测试项目中的一个类，创建一个名为[ClassName]Tests的类 |
| 工作单元（一个方法，或者几个方法组成的一个逻辑组，或者几个类） | 对应被测试项目中的一个类，创建一个如下命名的测试方法：[UnitOfWorkName]\_[ScenarioUnderTest]\_[ExpectedBGheavior]。如果整个工作单元就是一个方法，工作单元名就可以很简单，就是这个方法名；如果工作单元是一个包含多个方法或类的用例，工作单元名就可能比较抽象，如：UserLogin、RemoveUser或Startup。你可以从方法名开始，之后逐渐过渡到比较抽象的工作单元名。如何使用方法名，要确保这些方法是公共的，否者它们不能真正代表一个工作单元的起点。 |

测试方法名称的三部分：

1. UnitOfWorkName 被测试的方法、一组方法或者一组类
2. Scenario 测试进行的假设条件，例如"登入失败""无效用户"或"密码正确"。你可以用测试场景描述传给公开方法的参数，或者单元测试进行时系统的初始状态，例如："系统内存不足""无用户存在"或"用户已经存在"。
3. ExpectedBehavior 在测试场景指定的条件下，你对被测试方法行为的预期。测试方法的行为有三种可能的结果：返回一个值（一个真实值，或者一个异常），改变系统状态（例如在系统中添加了一个用户，导致在下一次登入时系统的行为发生变化），或调用一个第三方系统（例如一个外部的Web服务）。

在我们对IsValidLogFileName方法进行测试中，场景是你给方法传入一个有效的文件名，预期行为是方法返回一个值true。我们可以把这个测试的方法命名为IsValidFileName\_BadExtension\_ReturnFalse()。

你应该把测试代码放在产品代码项目中吗？还是应该把测试代码单独放在另一个测试相关的项目里呢？我通常选择把测试和产品代码分开，这样可以使测试相关的所有其他任务更容易进行，而且，在产品代码中包含测试代码容易导致复杂的条件编译设置，还会带来其他的问题，降低代码的可读性，因此很多人都不喜欢这种做法。

**一个单元测试通常主要包含三个行为：**

1. 准备（Arrange）对象，创建对象，进行必要的设置；
2. 操作（Act）对象；
3. 断言（Assert）某件事情是预期的。

下面是一段简单的代码，分别为被测试代码与测试单元，测试单元包含了以上全部三个行为，其中断言部分使用了NUnit框架提供的Assert类。

[复制代码](javascript:void(0);)

public class LogAnalyzer

{

public bool IsValidLogFileName(string fileName)

{

if (string.IsNullOrEmpty(fileName))

{

throw new ArgumentException("filename has to be provided");

}

if (fileName.EndsWith(".SLF")//此中故意丢失!运算符与忽略大小写，就是为了测试其存在缺陷。

{

return false;

}

return true;

}

}

[Test]

public void IsValidLogFileName\_BadExtension\_ReturnsFalse()

{

LogAnalyzer analyzer = new LogAnalyzer();//三部分行为“A-A-A”，都隔一行就是便于区分与阅读

bool result = analyzer.IsValidLogFileName("filewithbadextension.foo");

Assert.False(result);

}

[复制代码](javascript:void(0);)

如上，当我们需要使用多个文件名来测试单元有效性时，难道要写多个测试方法吗。肯定不是，可以使用TestCase属性标记，该属性与更多属性的详细说明上一节已经列出。如下代码：

[复制代码](javascript:void(0);)

[TestCase("filewithgoodextension.SLF",true)]

[TestCase("filewithgoodextension.slf",true)]

[TestCase("filewithbadextension.foo",false)]

public void IsValidLogFileName\_VariousExtensions\_ChecksThem(string file, bool expected)

{

LogAnalyzer analyzer = new LogAnalyzer();

bool result = analyzer.IsValidLogFileName(file);

Assert.AreEqual(expected, result);

}

[复制代码](javascript:void(0);)

检测预期的异常，一个常见的场景是：保证当异常应该抛出时，被测试的方法能够抛出正确的异常。

假设传入一个空文件名的时候，你的方法应该抛出一个ArgumentException异常，如果代码在这种情况下没有抛出异常，你的测试就应该失败，代码上面已列出。对此有两种测试方法，让我们先来看不应该用的那种，因为这种方法很流行，而且曾经是做这种测试的唯一方法。使用ExpectedException属性标记测试异常。代码如下：

[复制代码](javascript:void(0);)

[Test]

[ExpectedException(typeof(ArgumentException),ExpectedMessage = "filename has to be provided")]

public void IsValidLogFileName\_EmptyFileName\_ThrowsException()

{

LogAnalyzer la = MakeAnalyzer();

la.IsValidLogFileName(string.Empty);

}

private LogAnalyzer MakeAnalyzer()

{

return new LogAnalyzer();

}

[复制代码](javascript:void(0);)

在这段代码中没有使用Assert调用，[ExceptedException]属性内部包含断言，为什么说不应该使用这种方法呢？因为这个属性基本上是告诉测试运行器把这整个方法包在一个大的try-catch块里，如果没有东西"捕捉"到，就认为测试失败。这种做法有一个很大的问题，就是你不知道是哪一行代码抛出的这个异常。实际上，如果构造函数有问题，抛出了一个异常，你的测试也会通过，而构造函数是绝对不应该抛出异常的，这样的话，使用这个属性，测试结果有可能是不真实。所以尽量不要用这种方法。

NUnit提供了一个更新的API：Assert。Catch<T>(delegate)，以下是使用Assert.Catch编写的代码：

[复制代码](javascript:void(0);)

[Test]

public void IsValidLogFileName\_EmptyFileName\_Throws()

{

LogAnalyzer la = MakeAnalyzer();

var ex = Assert.Catch<ArgumentException>(() => la.IsValidLogFileName(""));

StringAssert.Contains("filename has to be provided", ex.Message);

}

[复制代码](javascript:void(0);)

Assert.Catch函数返回Lambda内抛出的异常实例，你可以在之后的代码中对这个异常对象的消息进行断言。

使用StringAssert，它包含能够简化字符串测试的辅助方法，使用这个类可以提高代码可读性。

没有用Assert.AreEqual进行全字符串相等断言，而是使用StringAssert.Contains断言消息包含你寻找的字符串。随着时间的变化，当代码中加入新功能后，字符串经常会发生变化，经常会包含额外的换行符以及你不关心的多余信息，使用StringAssert.Contains可以使测试更容易维护，否则就不得不对这个测试进行修复。

使用这种方法测试结果的可能性就比较小了，因此我推荐使用Assert.Catch而不是[ExpectedException]。

**测试系统状态的改变而非返回值**

基于状态的测试（也称为状态验证）通过检查被测试系统极其协作方（依赖物）在被测试方法执行后行为的改变，判定被测试方法是否正确工作。

考虑对LogAnalyzer类的基于状态的简单测试，引入一个新的属性WasLastFilenameValid，这个属性记录IsValidLogFileName方法的上次调用成功与否。代码如下：

[复制代码](javascript:void(0);)

public class LogAnalyzer

{

public bool WasLastFileNameValid { get; set; }

public bool IsValidLogFileName(string fileName)

{

WasLastFileNameValid = false;

if (string.IsNullOrEmpty(fileName))

{

throw new ArgumentException("filename has to be provided");

}

if (!fileName.EndsWith(".SLF",StringComparison.CurrentCultureIgnoreCase))

{

return false;

}

WasLastFileNameValid = true;

return true;

}

}

[TestCase("badfile.foo", false)]

[TestCase("goodfile.slf", true)]

public void IsValidLogFileName\_WhenCalled\_ChangesWasLastFileNameValid(string file, bool expected)

{

LogAnalyzer la = MakeAnalyzer();

la.IsValidLogFileName(file);

Assert.AreEqual(expected, la.WasLastFileNameValid);

}

[复制代码](javascript:void(0);)

如你在以上代码中所见，LogAnlyzer记住了最后一次验证的结果，因为WasLastFileNameValid的值依赖另一个方法先调用，所以无法通过编写一个获得方法返回值的测试来检测它的功能。需要单独的状态属性进行断言。

以上内容根据《单元测试的艺术----第二版》进行整理的（其内容主要讲解编写优秀的测试）。