前言

本文是阅读了《单元测试之道》一书后的笔记，也 是公司安排本人进行单元测试培训的材料，原文是一个 Powerpoint，故修改了下，并针对 Visual studio 2005 自带的单元测试做的一个整理，将其奉献出来，目的是供需要了解和学习单元测试的朋友们阅读。如有错误望指出。

什么是单元测试？

单元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。例如，你可能把一个很大的 值放入一个有序 list 中去，然后确认该值出现在 list 的尾部。或者，你可能会从字符串中删除匹配某种模式的字符，然后确认字符串确实不再包含这些 字符了。

执行单元测试，是为了证明某段代码的行为确实和 开发者所期望的一致。

为什么需要单元测试？

当编写项目的时刻，如果我们假设底层的代码是正 确无误的，那么先是高层代码中使用了底层代码；然后这些高层代码又被更高层的代码所使用，如此往复。当基本的底层代码不再可靠时，那么必需的改动就无法只 局限在底层。虽然你可以修正底层的问题，但是这些对底层代码的修改必然会影响到高层代码。

于是，一个对底层代码的修正，可能会导致对几乎所有代码的一连串 改动，从而使修改越来越多，也越来越复杂。从而使整个项目也以失败告终。

而单元测试的核心内涵：这个简单有效的技术就是 为了令代码变得更加完美。

什么是断言

Assertion（断言），它是一个简单的方法调用，用于判断某个语句是否为真。

例如：

```
public void IsTrue(bool condtion){
 if(!condition) abort();
}
```

应用则为：

```
int a=2;
IsTrue(a==2);
```

还可以编写更多的特定数据类型的断言。

计划你的单元测试

当我们编写了一个如下的函数，它用于查找 list 中的最大值：static int Largest(int[] list);

所能想到的测试如下：

输 入	预 期结果
7,8,9	9
8,9,7	9
9,7,8	9
7,9,8,9	9
1	1
-9,-8,-7	-7
null	Exception

创建单元测试

在解决方案资源管理器中右击某个测试项目，或在 Visual Studio 代码编辑器中，右击要测试的命名空间、类或方法并选择“创建单元测试”。

VsUnit 的各种断言

Assert

在测试方法中，可以调用任意数量的 Assert 类方法，如 Assert.AreEqual ()。Assert 类有很多方法可供选择，其中许多方法具有若干重载。

CollectionAssert

使用 CollectionAssert 类可比较对象集合，也可验证一个或多个集合的状态。

StringAssert

使用 StringAssert 类可对字符串进行比较。此类包含各种有用的方法，如 StringAssert.Contains、StringAssert.Matches 和 StringAssert.StartsWith。

AssertFailedException

只要测试失败，就会引发 AssertFailedException 异常。如果测试超时，引发意外的异常，或包含生成了 Failed 结果的 Assert 语句，则该测试失败。

AssertInconclusiveException (无结 果的)

只要测试生成的结果为 Inconclusive，就会引发 AssertInconclusiveException。通常，向仍在处理的测试添加 Assert.Inconclusive 语句可指示该测试尚未准备好，不能运行。

UnitTestAssertException

编写新的 Assert 异常类时使该类从基类 UnitTestAssertException 进行继承，可更方便地将异常标识为断言失败而非从测试或产品代码引发的意外异常。

ExpectedExceptionAttribute

如果希望开发代码中的某方法引发异常，又想用测 试方法来验证是否真的在该方法中引发了异常，则请用 ExpectedExceptionAttribute 属性来修饰测试方法。

如：

```
[TestMethod]
[ExpectedException(typeof(ArgumentException),
    "userID 为 NULL 的异常检测.")]
public void NullUserIdInConstructor()
{
    LogonInfo logonInfo = new LogonInfo(null, "P@ss0word");
```

}

单元测试的属性

除了单元测试方法的 [**TestMethod**()] 属性及其包容类的 [**TestClass**()] 属性之外，可使用其他属性启用特定的单元测试功能。在这些属性中，最主要的属性有 [**TestInitialize**()] 和 [**TestCleanup**()]。使用标记有 [**TestInitialize**()] 的方法对将要在其中运行单元测试的环境的各个方面进行准备；这样做的目的在于为单元测试的运行建立已知的状态。例如，可以使用 [**TestInitialize**()] 方法复制、更改或创建测试中将要使用的某些数据文件。

在运行完某个测试后，可通过标记有 [**TestCleanup**()] 的方法将环境返回到已知状态；这可能意味着需要删除文件夹中的文件，或将某个数据库返回到已知状态。例如，在测试了订单录入应用程序中使用的某个方法后， 可将库存数据库重置为初始状态。此外，建议您在 [**TestCleanup**()] 或 **ClassCleanup** 方法中使用清除代码，而不要在终结器方法（**~Constructor**）中使用此代码。从终结器方 法引发的异常不会被捕捉到，并且会导致无法预料的结果。

用于建立调用顺序的 属性

对于程序集:

在加载程序集之后以及卸载程序集之前，将调用 **AssemblyInitialize** 和 **AssemblyCleanup**。

对于类 :

在加载类之后以及卸载类之前，将调用 **ClassInitialize** 和 **ClassCleanup**。

对于测试方法 ：

在每个测试方法加载以及卸载之前，将调用 **TestInitialize** 和 **TestCleanup**

什么是 VsUnit 的 TestContext 类

测试上下文类的属性存储有关当前测试运行的信 息。例如，TestContext.DataRow 和 TestContext.DataConnection 属性包含测试用于数据驱动的单元测试的信息。

其他

用于对测试进行标识和排序的属性

测试配置类

用于生成报告的属性

用于专用访问器的类

测试哪些内容：Right-BICEP

Right----------结果是否正确？

B----------是否所有的边界条件都是正确的？

I----------能查一下反向关联吗？

C----------能用其他手段交叉检查一下结果吗？

E----------你是否可以强制错误条件发生？

P----------是否满足性能要求？

RIGHT:结果是否正确

如果代码能运行正确，如何才知道它是正确的呢？

1、使用更明确的设计文档

2、真实环境数据

3、????

B:边界条件

尽可能的至少各种特殊或者意外的情况，测试程序 是否能正常工作，如：

l 完 全伪造或者不一致的输入数据，如叫做“(*@Q!&#?±的文件。

l 格 式错误的数据，如错误格式的邮件地址

l 空 值或不完整的值

l 一 些与意料中的合理值相去甚远的值，如年纪为 10000

l 如 果要求是一个不允许出现重复数值的 list，但传入一个有重复数值的 list

l 要 求是一个有序的 list，但传入一个无序的 list

l 处 理的顺序是错误的，或者与期望的次序不一致。如未登录系统就尝试打印。

B:边界条件的-CORRECT

Conformance（一致性）值是否和预期的一致

Ordering（顺序性）值是否应该的那样有序或者无序

Range（区间性）值是否位于合理范围

Reference（依赖性）代码是否引用了一些代码本身控制范围之外的资源

Existence（存在性）值是否存在（是否非空，非零，在集合中等等）

Cardinality（基数性）是否恰好有足够的值

Time（相对或绝对的时间性）所有事情的发生是否有序？是否在正确的时间？是否恰好及时？

I:检查反向关联

通常一些结果可以使用反向的逻辑关系来验证它们 是否正确，如：计算 a *b 的函数，测试方法如下：

```
Public void UsingInverse(){
```

```
 double x = MyMath.AB(4,4);
 Assert.AreEqual<double>(x,4*4);
}
```

C:使用其他手段实现交叉检查

计算一个结果可以存在多个算法 ,同一个算法可以使用稳定的版本来校验新改进的版本，如：

```
Public void UsingStd(){
    double number = 23214.01;
    double result1 = MyMath.旧方 法(number1);
    double result2 = MyMath.新方 法(number1);
    Assert.AreEqual<double>(result1,result2);
}
```

E:强制产生错误条件

真实运行环境各种出乎意料的事情都可能发生，如 断电，断网等等。在测试中模拟这些情况可以使用 Mock 对象来实现。

P:性能特性

如数据采集的功能，在十个网站上进行采集工作很 正常，那么在 1000 个网站上或更多的网站上进行采集它的速度如何？是否写个单元测试？

如何才是一个好的单元测试

好的测试应该具有的品质是：**A-TRIP**（合称）

Automatic 自动化

Thorough 彻底的

Repeatable 可重复

Independent 独立的

Professional 专业的

Automatic 自动化

调用自动化

检查结果自动化

Thorough 彻底的

测试所有可能会出现问题的情况，一个极端是，对 于每行代码、代码可能到达的分支 ,每个可能抛出的异常等等 ,都可以作为测试的对象。另一个极端是，你仅仅测试最可能的情况---边界条件、残缺和畸形的数 据等等。然而这些都基于项目需求的决策问题。

这些所说的归纳为“代码覆盖率”。

Repeatable 可重复

测试应该独立于所有其他的测试，而且必须独立于 周围的环境。目标只有一个 ,就是测试应该能够以任意的顺序一次又一次的运行 ,并且产生相同的结果。如果结果不同，则存在 BUG。

Independent 独立 的

编写测试时，确保一次只测试了一样东西，但并不 表示一个 TestMethod 内只能使用一个 Assert，而是一个测试函数应该专注于产品代码中的一个函数，或者组合起来并共同提供某个特性的一组函 数。

Professional 专业 的

测试代码必须同产品代码相同的风格来编写。这意 味着你需要抽取出共同且重复的代码，并把它们放到一个功能类之中，从而可以复用；单元测试的代码一样讲究------维护封装，DRY 原则，降低耦合。

何时需要 Mock 对象

l 真 实对象具有不可确定的行为(产生不可预测的结果，如股票的行情)

l 真 实对象很难被创建(比如具体的 web 容器)

l 真 实对象的某些行为很难触发(比如网络错误)

l 真 实情况令程序的运行速度很慢

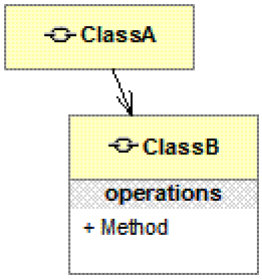
l 真 实对象有用户界面

l 测 试需要询问真实对象它是如何被调用的(比如测试可能需要验证某个回调函数是否被调用了)

l 真 实对象实际上并不存在(当需要和其他开发小组，或者新的硬件系统打交道的时候，这是一个普遍的问题)

Mock 对象的三个步骤

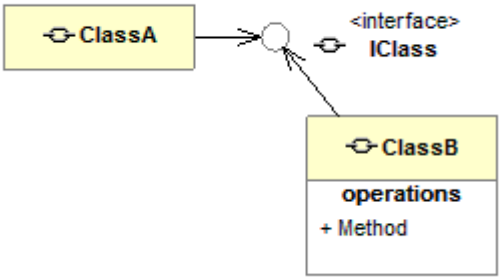
1. 原型



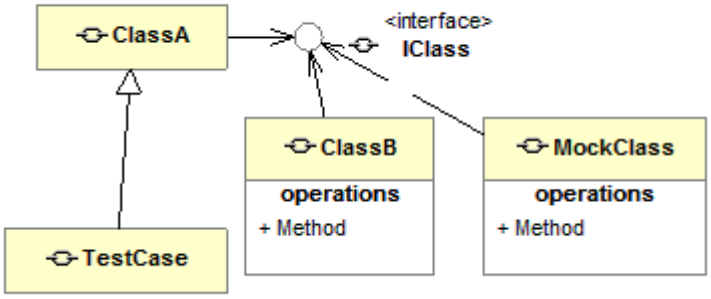
ClassA 调用 ClassB 的 Method()

2. 使用 一个接口来描述这个对象

ClassA 通过接口调用 ClassB 的 Method



3. 以测 试为目的，在 mock 对象中实现这个接口



全文完

简介

最新发布的 Visual Studio Test System (VSTS) 包含了一套用于 Visual Studio Team Test 的完整功能。Team Test 是 Visual Studio 集成的单元测试框架，它支持：

- 测试方法存根 (stub) 的代码生成。
- 在 IDE 中运行测试。
- 合并从数据库中加载的测试数据。
- 测试运行完成后，进行代码覆盖分析。

另 外，Team Test 包含了一套测试功能，可以同时支持开发人员和测试人员。

在本文中，我们准备演练如何创建 Team Test 的单元测试。我们从一个简单的示例程序集开始，然后在该程序集中生成单元测试方法存根。这样可以为 Team Test 和单元测试的新手读者提供基本的语法和代码，同时也很好地介绍了如何快速建立测试项目的结构。然后，我们转到使用测试驱动开发 (test driven development, TDD) 方法，即在写产品代码前先写单元测试。

Team Test 的一个关键特点是从数据库中加载测试数据，然后将其用于测试方法。在演示基本的单元测试后，我们描述如何创建测试数据并集成到测试中。

本 文中使用的示例项目包含一个 LongonInfo 类，它封装了与登录相关的数据（例如用户名和密码）以及一些关于数据的简单的验证规则。最终的类如下图 1 所示。

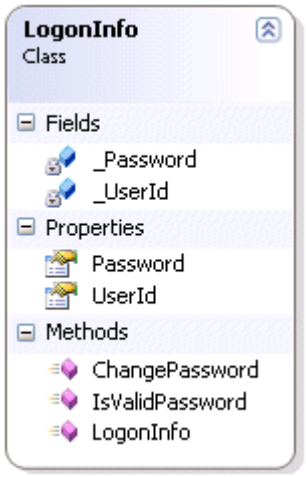


图 1. 最终的 LogonInfo 类

请注意所有的测试代码 位于一个单独的项目。这是有道理的，产品代码应该尽可能少的受测试代码影响，所以我们不想在产品代码的程序集中嵌入测试代码。

返 回页首

开始

首先，我们创建一个名为“VSTSDemo”的类库项目。默认情况 下，**为方案创建目录(Create directory for solution)** 复选框 被选中。保留此选项可以使我们在 VSTSDemo 项目的同一层目录创建测试项目。相反，如果不选中此选项，Visual Studio 2005 会将测试项目放在 VSTSDemo 项目的子目录中。测试项目遵循 Visual Studio 在解决方案文件路径的子目录中创建额外项目的规定。

创 建初始的 VSTSDemo 项目后，我们使用 Visual Studio 的解决方案资源管理器将 Class1.cs 文件重命名为 *LogonInfo.cs*， 这样类名也会被更新为 LogonInfo。然后我们修改构造函数以接受两个字符串参数：userId 和 password。 一旦构造函数的签名被声明，我们就可以为构造函数生成测试。

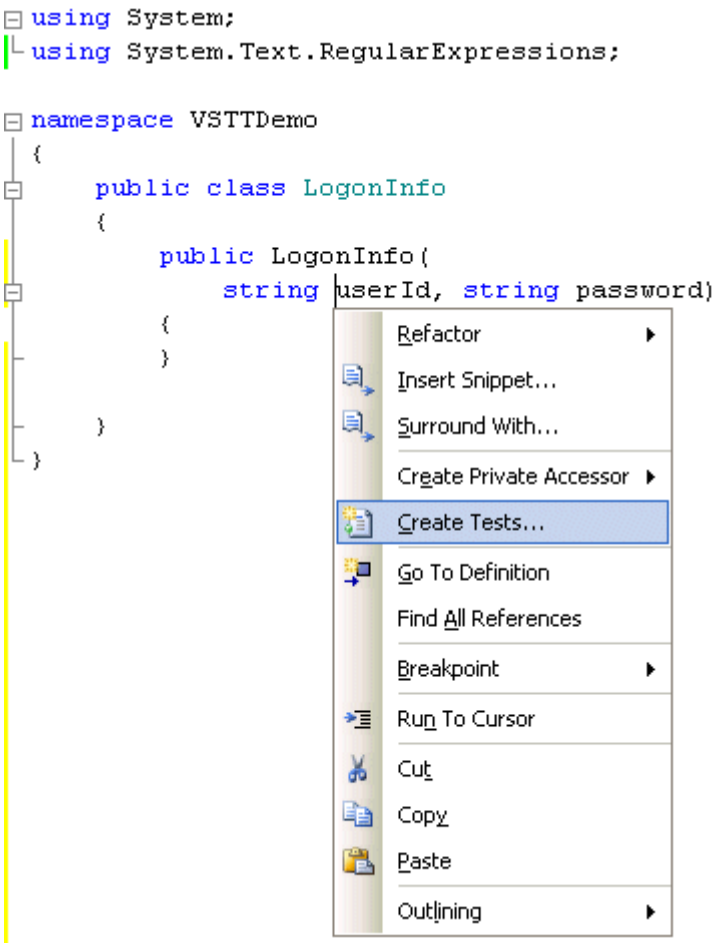


图 2. LongonInfo 构造函数的上下文菜单的“创建测试…” (Create Tests...) 菜单项

返 回页首

创建测试

在开始编写 LogonInfo 的 任何实现之前，我们遵循 TDD 实践的规则，首先编写测试。TDD 在 Team Test 中并不是必需的，但最好在本文的剩余部分遵循 TDD。右键单击 LogonInfo()构造函数，然后选择“创建测试…”菜单项(如图 2 所示)。这样会出现一个对话框，可以在不同的项目中生成单元测试（如图 3 所示)。默认情况下,项目设置的输出 (Output) 选项是一个新的 Visual Basic 项目，但是也可以选择 C# 和 C++ 测试项目。在本文中，我们选择 Visual C#，然后单击 OK 按 钮，接着输入项目名 *VSTSDemo.Test*。测试项目名称。

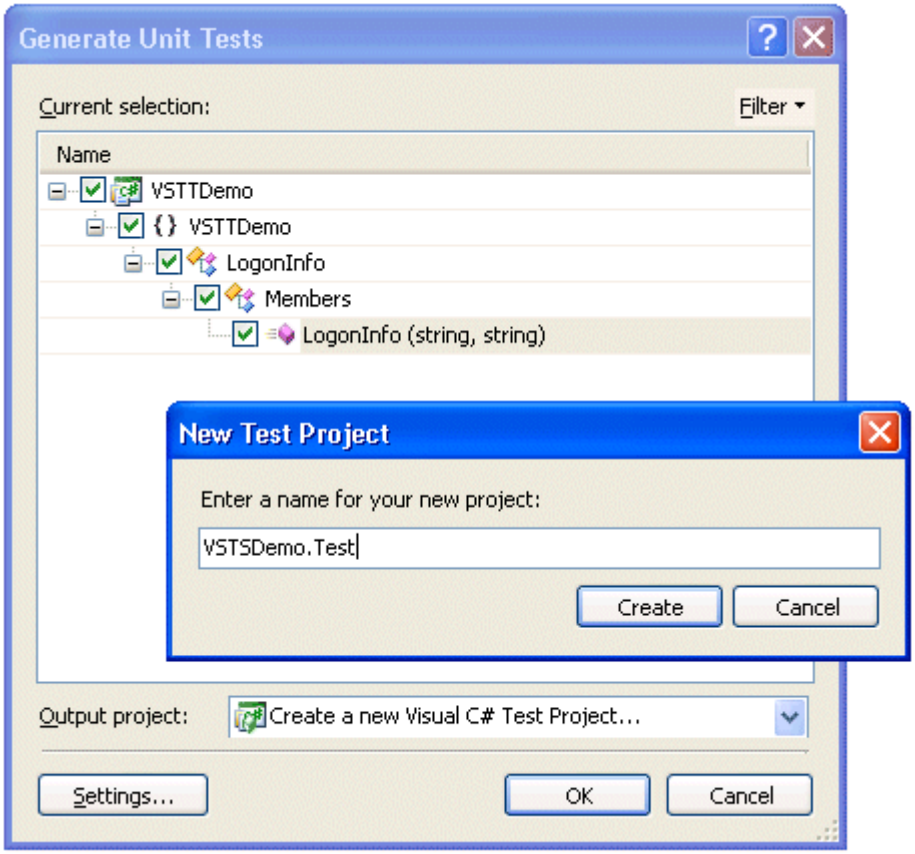


图 3. 生成单元测试对话框

生成的测试项目包含四个与测试相关的文件。

文 件名	目的
AuthoringTest.txt	提 供关于创建测试的说明，包括向项目增加其他测试的说明。
LogonInfoTest.cs	包 含了用于测试 LogonInfo()的生成测试，以及测试初始化和测试清除的方法。
ManualTest1.mht	提 供了一个模板，可以填入手工测试的指令。
UnitTest1.cs	一 个空的单元测试类架构，用于放入另外的单元测试。

因 为我们不打算对该项目进行手工测试,并且由于已经有了一个单元测试文件,我们将删除 ManualTest1.mht 和 UnitTest1.cs。

除 了一些默认的文件，生成的测试项目还包含了对 Microsoft.VisualStudio.QualityTools.UnitTestFramework 和 VSTSDemo 项目的引用。前者是测试引擎运行单元测试需要依赖的测试框架程序集，后者是对我们需要测试的目标程序集的项目引用。

默认情况下，生成的测试 方法是包含以下实现的占位符：

清单 1. 生成的测试方法：ConstructorTest(),位 于 VSTSDemo.Test.LogonInfoTest

```
/// <summary>
   ///This is a test class for VSTTDemo.LogonInfo and is intended
   ///to contain all VSTTDemo.LogonInfo Unit Tests
   ///</summary>
   [TestClass()]
   public class LogonInfoTest
   {
      // ...

      /// <summary>
      ///A test case for LogonInfo (string, string)
      ///</summary>
      [TestMethod()]
      public void ConstructorTest()
      {
         string userId = null; // TODO: Initialize to an appropriate
value

         string password = null; // TODO: Initialize to an
appropriate value

         LogonInfo target = new LogonInfo(userId, password);

         // TODO: Implement code to verify target
         Assert.Inconclusive(
            "TODO: Implement code to verify target");
      }
}
```

确切的生成代码会根据测试目标的方法类型和签名不同而有所不同。例如，向导会为私有成员函数的测试生成反射代码。在这种特别的情况下， 我们需要专门用于公有构造函数测试的代码。

关于 Team Test ，有两个重要的特性。首先，作为测试的方法由 TestMethodAttribute 属 性指定，另外，包含测试方法的类有 TestClassAttribute 属性。这些属性都可以在 Microsoft.VisualStudio.QualityTools.UnitTesting.Framework 命名空间中找到。Team Test 使用反射机制在测试程序集中搜索所有由 TestClass 修饰的类，然后查找由 TestMethodAttribute 修 饰的方法来决定执行的内容。另外一个重要的由执行引擎而不是编译器验证的标准是，测试方法的签名必须是无参数的实例方法。因为反射搜索 TestMethodAttribute， 所以测试方法可以使用任意的名字。

测试方法 ConstructorTest()首先实例化目标 LongonInfo 类，然后断言测试是非决定性的(使用 Assert.Inconclusive())。当测试运行时，Assert.Inconclusive()说 明了它可能缺少正确的实现。在我们的示例中，我们更新 ConstructorTest()方法，让它检查用户名和密码的初 始化，如下所示。

清单 2. 更新的 ConstructorTest()实 现

```
/// <summary>
      ///A test case for LogonInfo (string, string)
      ///</summary>
      [TestMethod()]
      public void ConstructorTest()
      {
         string userId = "IMontoya";

         string password = "P@ssw0rd";

         LogonInfo logonInfo = new LogonInfo(userId, password);

         Assert.AreEqual<string>(userId, logonInfo.UserId,
            "The UserId was not correctly initialized.");
         Assert.AreEqual<string>(password, logonInfo.Password,
            "The Password was not correctly initialized.");
      }
```

请注意我们的检查使用 Assert.AreEqual<T>() 方法完成。Assert 方 法也支持没有泛型的 AreEqual()，但是泛型版本几乎总是首选，因为它会在编译时验证类型匹配 — 在 CLR 支持泛型前，这种错误在单元测试框架中非常普遍。

因为 UserID 和 Password 的实例域还没有创建，我们需要回头将其添加到 **LogonInfo** 类中，以便 VSTTDemo.Test 项目可以编译。

即使我们还没有一个有效的实现， 让我们开始运行测试。如果我们遵循 TDD 方法，我们就应该直到测试证明我们需要这样的代码时才去编写产品代码。我们仅在建立项目结构时违背此原则，但是一旦项目建立后，就可以容易地始终遵循 TDD 方法。

返 回页首

运行测试

要运行项目中的所有测试，只需要运行测试项目。要实现这一点， 我们需要右键单击解决方案资源管理器的 VSTSDemo.Test 项目，选择**设置为启动项目**(Set as StartUp Project)。接着，使用菜单项**调试->启动(F5)** 或者**调试->开始执行（不调试）(Ctrl+F5)** 开 始运行测试。

这时出现测试结果窗口，列出项目中的所有测试。因为我们的项目只包含一个测试，因此只列出了一个测试。开始的时候，测试会处于 挂起的状态，但是一旦测试完成，结果将是我们意料中的失败（如图 4 所示）。

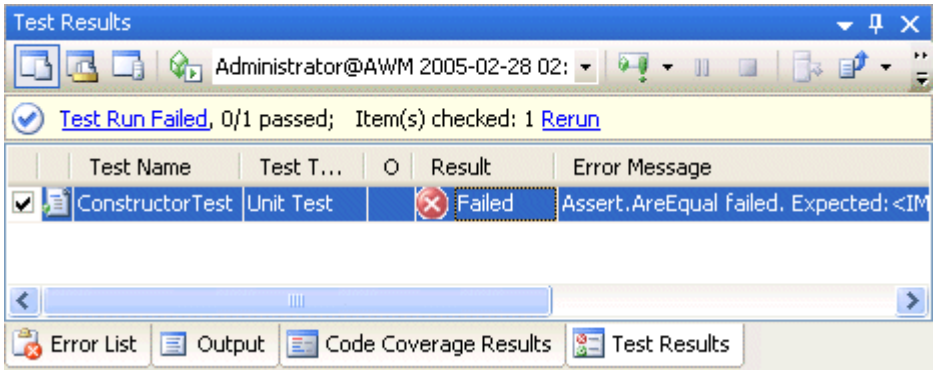


图 4. 执行所有测试后的测试结果窗口

图 4 显示了**测试结果 (Test Results)** 窗口。这个特别的屏幕快照除了默认的列外，还显示了错误信息。您可以在列头上单击右键并选择菜单项**增加/删 除列…**以增加或者删除列。

如果要查看测试的额外细节，我们可以选定测试并双击，打开“ConstructorTest[Results]”窗口，如图 5 所示。

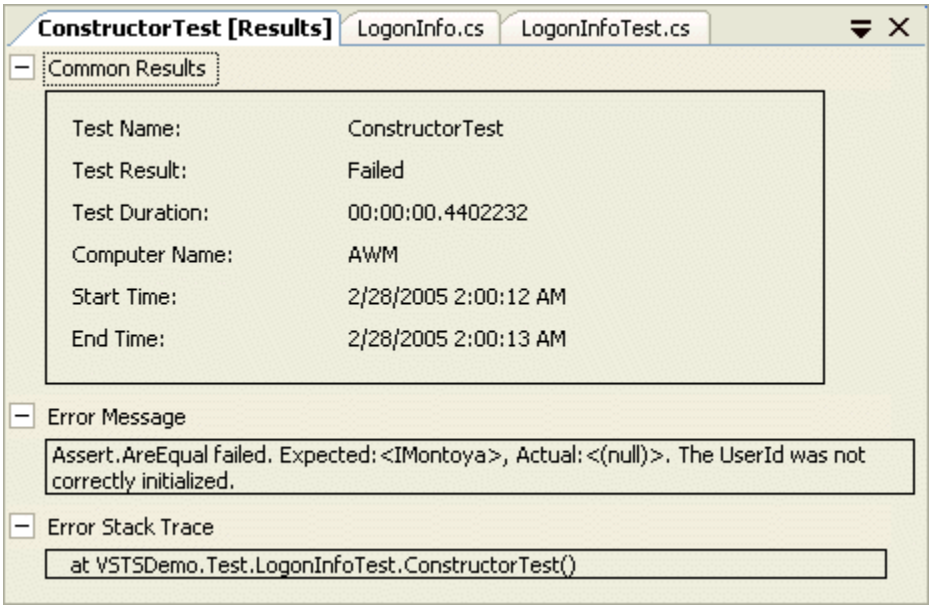


图 5. 详细的测试 ConstructorTest [Results]窗 口

另外，我们可以右键单击单个测试，然后选择**打开测试**(Open Test) 菜 单项，进入测试代码。因为我们已经知道问题在于 LogonInfo 构造函数的实现，我们可以去那里编写初始化 UserID 和 Password 字段的代码，使用传入的参数对它们进行初始化。重新运行测试以验证测试现在可以通过。

返 回页首

检查异常

下一步是创建 LongonInfo 类，以提 供对 UserID 和 password 的一些验证。不幸的是，UserID 和 Password 字段是公共的，这意 味着它们没有提供任何封装来确保它们有效。但是在我们将其转换为属性并提供验证前，让我们编写一些测试来验证任何实现的结果都是正确的。

我 们首先来编写一个测试，防止空值 (null) 或空字符串赋值给 UserID。预期结果是，如果空值传送给构造函数，会引发一个 ArgumentException 异 常。测试代码如清单 3 所示。

清单 3. 使用 ExpectedExceptionAttribute 对 异常情况进行测试

```
[TestMethod]
      [ExpectedException(typeof(ArgumentException),
         "A userId of null was inappropriately allowed.")]
      public void NullUserIdInConstructor()
      {
```

```
        LogonInfo logonInfo = new LogonInfo(null, "P@ssOword");
    }

    [TestMethod]
    [ExpectedException(typeof(ArgumentException),
        "A empty userId was inappropriately allowed.")]
    public void EmptyUserIdInConstructor()
    {
        LogonInfo logonInfo = new LogonInfo("", "P@ssOword");
    }
```

请注意对于 ArgumentException 没有 try-catch 代码块的显式测试。不过，两个测试都包含另外一个属性 ExpectedException，它接受一个类型参数，以及一个可选的错误信 息，用于在没有引发异常时显示。当这个单元测试执行时，测试框架会显式地监视引发的 ArgumentException 异 常，如果方法没有引发这个异常，测试将失败。运行这些测试会证明我们还没有对 UserID 做任何验证检查；因此，测试会失败，因为没有引发预期的异常。

有了失败的测试，现在可以回到产品代码进行更新来提供测试需要检查的功能。在 这个例子中，我们将 UserID 字段转换为属性，并提供验证检查（清单 4）。

清单 4. 在 LogonInfo 类 中验证 UserId

```
public class LogonInfo
    {
        public LogonInfo(string userId, string password)
        {
            this.UserId = userId;
            this.Password = password;
        }

        private string _UserId;
        public string UserId
        {
            get { return _UserId; }
            private set
            {
                if (value == null || value.Trim() == string.Empty)
                {
                    throw new ArgumentException(
                        "Parameter userId may not be null or blank.");
                }
                _UserId = value;
            }
```

```
        }

        // ...
}
```

属性的实现使用了 C# 2.0 的功能，其中 getter 和 setter 的访问权限不一致。setter 的实现标识为私有，而 getter 实现为公有。这样 UserID 就不能在 LogonInfo 类外被修改了（除非通过反射机制）。

一旦增加了验证，我们可以重新运行测试来验证实现是正确的。我们运行所有的三个测试来验证 UserID 字段转换为属性的重构过程没有产生任何意外的错误。单元测试的真正价值在代码修改的时候才真正有所体现。一套单元测试可以保证我们在维护和改进代码的时候 没有破坏代码。

返 回页首

从数据库中加载测试数据

对于 LogonInfo 类的 下一次修改，我们将提供一个方法来改变密码。该方法接受旧密码和新密码作为参数。另外，我们会验证密码符合某种复杂性需求。确切的说，我们将保证密码符合 Windows Active Directory 的默认需求，即包含以下四种类型字符中的三种：

- 大写字母
- 小 写字母
- 标点符号
- 数字

另外，我们将检查密码最少包含 6 个字符，最多包含 255 个字符。

和之前一样，我们在编写实现前先为密码复杂性需求编写测试。但是显然，我们需要提供一个测试值的 大集合用于验证实现。我们不是为每个测试用例创建一个单独的测试，也不是创建一个循环来调用一系列的测试用例，我们将创建一个数据驱动测试，它从数据库中 取出所需的数据。

返 回页首

测试视图 (Test View) 窗口

首先我们定义一个名为 ChangePasswordTest() 的新测试。定义后，从菜单项**测试->查看和创建测试(Test->View and Author Tests)**为测试方法打开**测试视图**窗口，如图 6 所示：

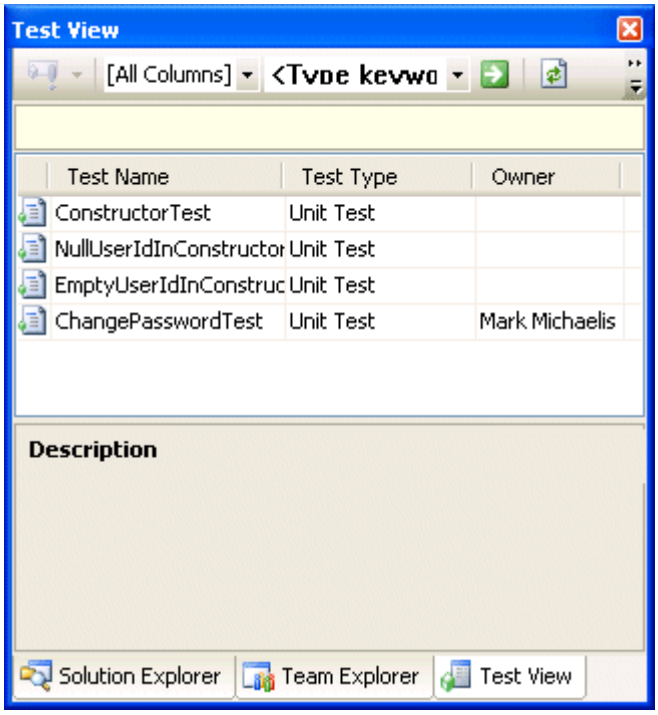


图 6. 测试视图 (Test view) 窗口

测试视图 窗口可用来运行指定的测试和浏览测试的特定属性。通过增加额外的列(右键单击列头并选择添加/删除列…),我们可以排序并根据偏好查看测试。有些列来自修 饰测试的属性。例如,添加 OwnerAttribute 将在所有者列显示测试的所有者。其它元数据属性(如 DescriptionAttribute)也 可以使用。这些属性都可以在 Microsoft.VisualStudio.QualityTools.UnitTesting.Framework 命名空间中找到。如果没有显式的属性存在,那么我们可以使用自由形式的 TestPropertyAttribute 来为特 别的测试方法增加名一值对。

没有对应列的属性可以在一个测试的属性窗口中显示(选择一个测试,在右键上下文菜单中单击**属性**)。 它包含了指定数据连接字符串和用于载入测试数据的表名的属性。显然,为了指定有效值,我们需要一个数据库连接。

返 回页首

增加一个测试数据库

从服务器资源管理器窗口,我们可以使用**创建新 的 SQL Server 数据库**(Create new SQL Server Database) 菜单项。但是要小心这种方法,如果我们要在其它计算机上执行测试的话,我们要保证在一台服务器上创建数据库, 其它可能执行测试的计算机必须能够访问该服务器 — 例如一台用于构建的计算机。

另外一个选择是仅仅增加一个数据库文件。使用**项目->增 加新项…** (Project ->Add new item...) 允许向项目插入一个 SQL 数据库文件。这种方法使测试

数据和测试项目保持在一起。缺点是如果数据库变得很大，我们就不想这么做，而宁可提供全局的数据源。

对 于本项目中的数据，我们创建一个名为 *VSTSDemo.mdf* 的本地项目数据库文件。为了向文件加入测试数据，我们使用菜 单**工具->连接到数据库 (Tools ->Connect to Database)**，然后指定 VSTSDemo.mdf 文件。然后，从服务器资源管理器窗口我们可以使用设计器加入一个新的表 **LongonInfoTest**。清单 5 显示了该表的定义。

清单 5. LogonInfoTestData SQL 脚本

```
CREATE TABLE dbo.LogonInfoTest
   (
   UserId nchar(256) NOT NULL PRIMARY KEY CLUSTERED,
   Password nvarchar(256) NULL,
   IsValid bit NOT NULL
   )  ON [PRIMARY]
GO
```

保存表后，我们可以将其打开，然后输入不同的非法密码，如下表所示。

UserId	Password	IsValid
Humperdink	P@w0d	False
IMontoya	p@ssword	False
Inigo.Montoya	P@ssw0rd	False
Wesley	Password	False

返 回页首

将数据与测试关联

一旦完成表的创建，我们需要将其与测试 InvalidPasswords()联 系起来。从测试 InvalidPasswords 的属性窗口，我们填写**数据连接字符串(Data Connection String)** 和**数据表名 (Data Table Name)** 属性。这样做将使用附加的属性 DataSourceAttribute 和 DataTableNameAttribute 更 新测试。最终的方法 ChangePasswordTest()在清单 6 中显示。

清单 6. 用于数据驱动测试的测试代码

```
enum Column
      {
         UserId,
```

```
            Password,
            IsValid
        }

        private TestContext testContextInstance;

        /// <summary>
        ///Gets or sets the test context which provides
        ///information about and functionality for the
        ///current test run.
        ///</summary>
        public TestContext TestContext
        {
            get
            {
                return testContextInstance;
            }
            set
            {
                testContextInstance = value;
            }
        }

        [TestMethod]
        [Owner("Mark Michaelis")]
        [TestProperty("TestCategory", "Developer"),
        DataSource("System.Data.SqlClient",
            "Data Source=.\\SQLEXPRESS;AttachDbFilename=\"<Path to the
sample .mdf file>";Integrated Security=True",
            "LogonInfoTest",
            DataAccessMethod.Sequential)]
        public void ChangePasswordTest()
        {

            string userId =
                (string)TestContext.DataRow[(int)Column.UserId];
            string password =
                (string)TestContext.DataRow[(int)Column.Password];
            bool isValid =
                (bool)TestContext.DataRow[(int)Column.IsValid];

            LogonInfo logonInfo = new LogonInfo(userId, "P@ssw0rd");

            if (!isValid)
```

```
        {
            Exception exception = null;
            try
            {
                logonInfo.ChangePassword(
                    "P@ssw0rd", password);
            }
            catch (Exception tempException)
            {
                exception = tempException;
            }
            Assert.IsNotNull(exception,
                "The expected exception was not thrown.");
            Assert.AreEqual<Type>(
                typeof(ArgumentException), exception.GetType(),
                "The exception type was unexpected.");
        }
        else
        {
            logonInfo.ChangePassword(
                "P@ssw0rd", password);
            Assert.AreEqual<string>(password, logonInfo.Password,
                "The password was not changed.");
        }
    }
```

清单 6 第一个需要注意的地方是增加了 DataSourceAttribute 属性，它指明了连接字 符串、表名和访问顺序。在这个清单中，我们使用数据库文件名标识数据库。这样的优点是该文件和测试项目一起迁移，假设它可能会被移动到一个相对的路径。

第 二个注意的地方是 TestContext.DataRow 调用。TestContext 是在 我们运行创建测试向导时由生成器提供的属性，它在运行时由测试执行引擎自动赋值，这样我们就可以在测试中访问跟测试环境关联的数据。如图 7 所示。

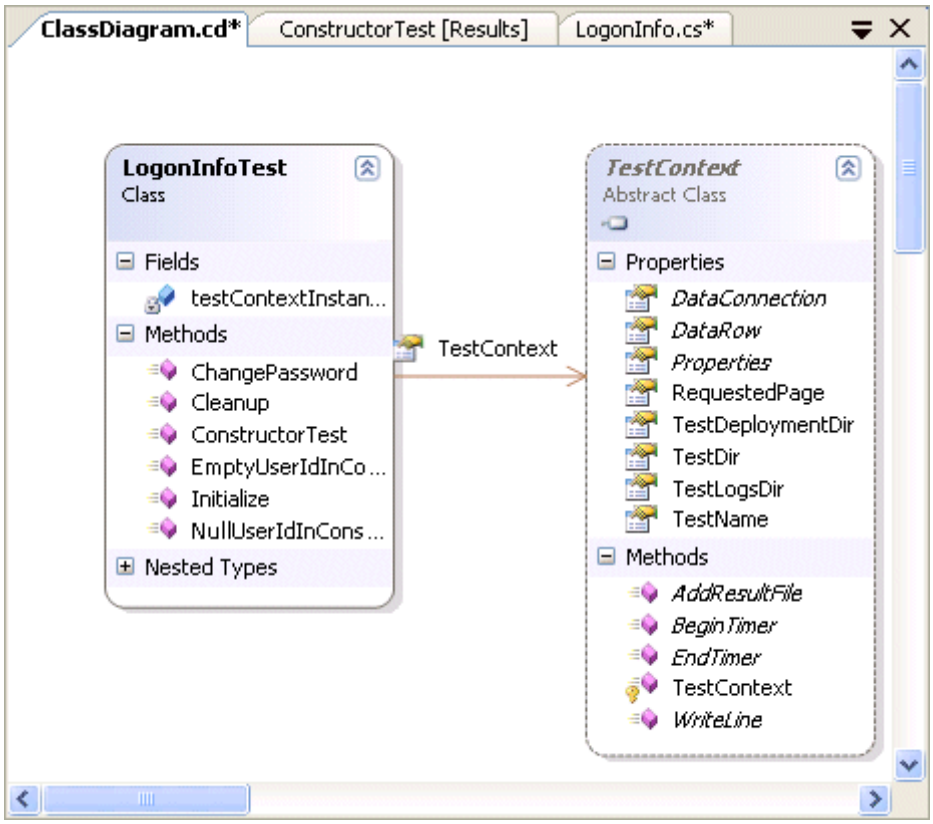


图 7. TestContext 关联

如图 7 所示，TestContext 提 供了 TestDirectory 和 TestName 数据，以及 BeginTimer()和 EndTimer()方 法。对 ChangePasswordTest()方法最有意义的是 DataRow 属性。因 为 ChangePasswordTest()方法由 DataSourceAttribute 修 饰，该属性指定的表返回每个记录时，该方法都会被调用一次。这就使测试代码使用运行中的测试的数据，而且对插入 LongonInfoTest 表的每条记录重复执行测试。如果表包含四条记录，那么测试将会分别执行四次。

使用这样的数据驱动测试方法，可以很容易的提供额外的测试数 据，而不需要编写任何代码。一旦需要额外的测试用例，我们需要做的就是向 LongonInfoTest 表增加关联的数据。尽管我们可以创建两个独立的测试来使用单独的表分别测试有效和无效数据，这个特定的例子合并了这些测试来显示稍微复杂的数据测试实例。

返 回页首

实现和重构目标方法

现在我们已经有了测试，是时候为测试编写实现了。使 用 C# 重构工具，我们可以右键单击 ChangePassword()方法调用，选择菜单项 GenerateMethodStub，然后对于生成的方法提供实现，一旦我们成功地运行了使用所有测试数据的测试，我们也可以开始重构代码了，LogonInfo 类的最终实现 如清单 7 所示。

清单 7. LogonInfo 类

```
using System;
using System.Text.RegularExpressions;

namespace VSTTDemo
{
   public class LogonInfo
   {
      public LogonInfo(string userId, string password)
      {
         this.UserId = userId;
         this.Password = password;
      }

      private string _UserId;
      public string UserId
      {
         get { return _UserId; }
         private set
         {
            if (value == null || value.Trim() == string.Empty)
            {
               throw new ArgumentException(
                  "Parameter userId may not be null or blank.");
            }
            _UserId = value;
         }
      }

      private string _Password;
      public string Password
      {
         get { return _Password; }
         private set
         {
            string errorMessage;
            if (!IsValidPassword(value, out errorMessage))
            {
               throw new ArgumentException(
                  errorMessage);
            }
            _Password = value;
         }
```

```
    }

    public static bool IsValidPassword(string value,
       out string errorMessage)
    {
       const string passwordSizeRegex = "(?=^.{6,255}$)";
       const string uppercaseRegex = "(?=.*[A-Z])";
       const string lowercaseRegex = "(?=.*[a-z])";
       const string punctuationRegex = @"(?=.*\d)";
       const string upperlowernumericRegex = "(?=.*[^A-Za-z0-9])";

       bool isValid;
       Regex regex = new Regex(
          passwordSizeRegex +
          "(" + punctuationRegex + uppercaseRegex + lowercaseRegex
+
             "|" + punctuationRegex + upperlowernumericRegex +
lowercaseRegex +
             "|" + upperlowernumericRegex + uppercaseRegex +
lowercaseRegex +
             "|" + punctuationRegex + uppercaseRegex +
upperlowernumericRegex +
          ")^.*");

       if (value == null || value.Trim() == string.Empty)
       {
          isValid = false;
          errorMessage = "Password may not be null or blank.";
       }
       else
       {
          if (regex.Match(value).Success)
          {
             isValid = true;
             errorMessage = "";
          }
          else
          {
             isValid = false;
             errorMessage = "Password does not meet the complexity
requirements.";
          }
       }
       return isValid;
```

```
        }

        public void ChangePassword(
            string oldPassword, string newPassword)
        {
            if (oldPassword == Password)
            {
                Password = newPassword;
            }
            else
            {
                throw new ArgumentException(
                    "The old password was not correct.");
            }
        }
    }
}
```

返 回页首

代码覆盖

单元测试的一个关键度量是决定在单元测试运行时测试了多少代 码。该度量称为代码覆盖，Team Test 包含了一个代码覆盖工具，可以详细解释被执行代码的百分率，并突出显示哪些代码被执行，那些没有被执行。该功能如图 8 所示。

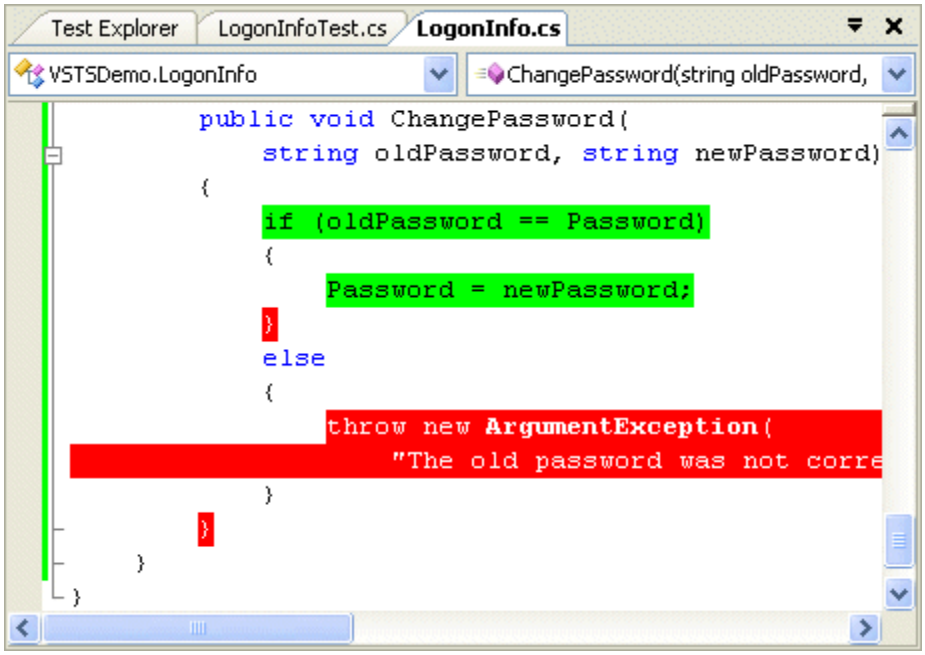


图 8. 突出显示代码覆盖

图 8 显示了运行所有单元测试后的代码覆盖的突出显示情况。红色突出显示说明了我们有产品代码没有运行任何单元测试，这说明我们编写这些代码时未遵循 TDD 原则，即在编写实现前先提供测试。

返 回页首

初始化和清除测试

一般来说，测试类不仅包含独立的测试方法，还包含了不 同的对测试进行初始化和清除的方法。实际上，创建测试向导在创建 VSTSDemo.Test 项目时，将一些这样的方法添加到类 LongonInfoTest 中，见清单 8。

清单 8. 最终的 LogonInfoTest 类

```
using VSTTDemo;
using Microsoft.VisualStudio.QualityTools.UnitTesting.Framework;
using System;

namespace VSTSDemo.Test
{
   /// <summary>
   ///This is a test class for VSTTDemo.LogonInfo and is intended
   ///to contain all VSTTDemo.LogonInfo Unit Tests
   ///</summary>
   [TestClass()]
   public class LogonInfoTest
   {

      private TestContext testContextInstance;

      /// <summary>
      ///Gets or sets the test context which provides
      ///information about and functionality for the
      ///current test run.
      ///</summary>
      public TestContext TestContext
      {
         get
         {
            return testContextInstance;
         }
         set
         {
            testContextInstance = value;
```

```
        }
    }

    /// <summary>
    ///Initialize() is called once during test execution before
    ///test methods in this test class are executed.
    ///</summary>
    [TestInitialize()]
    public void Initialize()
    {
        //  TODO: Add test initialization code
    }

    /// <summary>
    ///Cleanup() is called once during test execution after
    ///test methods in this class have executed unless
    ///this test class' Initialize() method throws an exception.
    ///</summary>
    [TestCleanup()]
    public void Cleanup()
    {

        //  TODO: Add test cleanup code
    }


    // ...

    [TestMethod]
    // ...
    public void ChangePasswordTest()
    {
    // ...
    }

}
}
```

用于对测试进行设置和清除的方法分别由属性 TestInitializeAttribute 和 TestCleanupAttribute 修 饰。在每个这样的方法中，我们可以加入额外的代码，它们将会在每个测试前或者测试后运行。这意味着在每次对应于 LongonInfoTest 表的记录的 ChangePasswordTest()执行前，Initialize() 和 Cleanup() 都会被执行，每次 NullUserIdInConstructor 和 EmptyUserIdInConstructor 执 行时也会发生同样的情况。这样的方法可以用于

向数据库中插入默认的数据，然后在测试完成时清除插入的数据。例如，我们可以做到在 Initialize()中 开始一个事务，然后在清除时回滚同一个事务，这样一来，如果测试方法使用相同的连接时，数据状态会在每次测试执行完成时恢复原状。类似地，测试文件也可以 这样处理。

在调试期间，TestCleanupAttribute 修饰的方法可能由于调试器在清除的代码执行 前终止运行。由于这个原因，最好在设置测试期间检查清除情况，并在需要时在设置测试前执行清除代码。关于初始化和清除的其它可用的测试属性有 AssemblyInitializeAttribute/AssemblyCleanupAttribute 和 ClassInitializeAttribute/ClassCleanupAttribute。程序集相关的属性对整个程序集运行一 次，而类相关的属性对一个特定的测试类的加载运行一次。

返 回页首

最佳实践

在结束前我们回顾几种单元测试的最佳实践。首先，TDD 是非常有价值的实践。在所有现有的开发方法中，TDD 可能是多年来根本上改进开发且投资成本最小的一种。每个 QA 工程师都会告诉您，开发人员在没有相应的测试前不会写出成功的软件。有了 TDD，实践是在实现前编写测试，并且理想情况是，编写的测试可以成为无需人工参与执行的构建脚本的一部分。需要训练来开始养成习惯，但一旦建立习惯后， 不使用 TDD 方法编码就像开车时不系安全带一样。

对于测试本身，有一些额外的原则可以帮助成功进行测试：

- 避 免测试产生依赖性，这样测试需要按照特定的顺序执行。每个测试都应该是自治的。
- 使用测试初始化代码验证测试清除已经成功 执行，如果没有则在执行测试前重新执行清除。
- 在编写任何产品代码的实现前编写测试。
- 对 于产品代码中的每个类创建一个测试类。这样可以简化测试的组织，并可以容易地选择在何处放置每个测试。
- 使用 Visual Studio 生成初始化的测试项目。这样可以大大减少手工设置测试项目并与产品项目关联的步骤。
- 避免 创建其他依赖计算机的测试，例如依赖特定的目录路径的测试。
- 创建模拟对象 (mock object) 来测试接口。模拟对象通常在需要验证 API 符合所需功能的测试项目中实现。
- 在继续创建新的测试前验证所有测试运行成 功。这样可以保证在破坏代码后立刻进行修正。
- 可以最大化无需人工参与执行的测试代码。在依赖于手工测试前，必须完全肯定 无法采用合理的无需人工参与执行的测试方案。

返 回页首

小结

总的来说，VSTS 的单元测试功能本身很好理解。而且尽管本文没有提到，它还可以通过自定义执行引擎进行扩展。此外，它包含了代码覆盖分析的功能，这对于评价测试的全面性非 常有用。通过使用 VSTS，您可以将测试数目和 bug 数目或编写的代码数量进行关联比较。这为项目的运行状况提供了很好的指标。

本文 介绍了 Team Test 产品中的基本单元测试功能，也探讨了关于数据驱动测试的一些更加高级的功能。通过开始实践对代码进行单元测试，您会为产品的整个生命期建立一套宝贵的测试 集。Team Test 通过与 Visual Studio 的强大集成和其它 VSTS 产品线，使这一切变得容易。

Mark Michaelis 在 Itron 公司担任软件架构师和讲师。他曾经对几个微软的产品设计进行检查，包括 C# 和 VSTS。现在他正在撰写另外一本有关 C# 的书，Essential C# (Addison Wesley)。不使用计算机时，他会陪伴家人，进行户外运动，或者进行环球旅行。Mark Michaelis 住在 Spokane, WA。您可以通过 mark@michaelis.net 和他联系或者访问他的网络日志：http://mark.michaelis.net。

转 到原英文页面

翻译者 Luke 是微软公司的软件工程师，习惯使用 C++和 C#开发应用程序。闲暇时间他喜欢音乐，旅游和怀旧游戏，并 且愿意帮助 MSDN 翻译更多的文章和其他开发者共享。可以通过 ecaijw@msn.com 联系他。