单元测试技术

广州盛原成科技有限公司

提　纲

1. 单元测试基础
2. 破除依赖-存根
3. 交互测试-模拟对象
4. 隔离（模拟）框架
5. 优秀单元测试的支柱

一、单元测试基础

什么是单元测试

从调用系统的一个公共方法到产生一个测试可见的最终结果，期间这个系统发生的行为总称为一个**工作单元**。我们通过系统的公共API和行为就可以观察到一个可见的**最终结果**，无需查看系统的内部状态。一个最终结果可以是以下任何一种形式。

- 被调用的公共方法返回一个值（一个返回值不为空的函数）。
- 在方法调用的前后，系统的状态或行为有可见的变化，这种变化无需查询私有状态即可判断。（例如：一个以前不存的用户可以登录系统，或者一个状态机系统的属性发生变化。）
- 调用了一个不受测试控制的第三方系统，这个第三方系统不反回任何值，或者返回值都被忽略。（例如：调用一个第三方日志系统，这个系统不是你编写的，而且你也没有源代码。）

定义

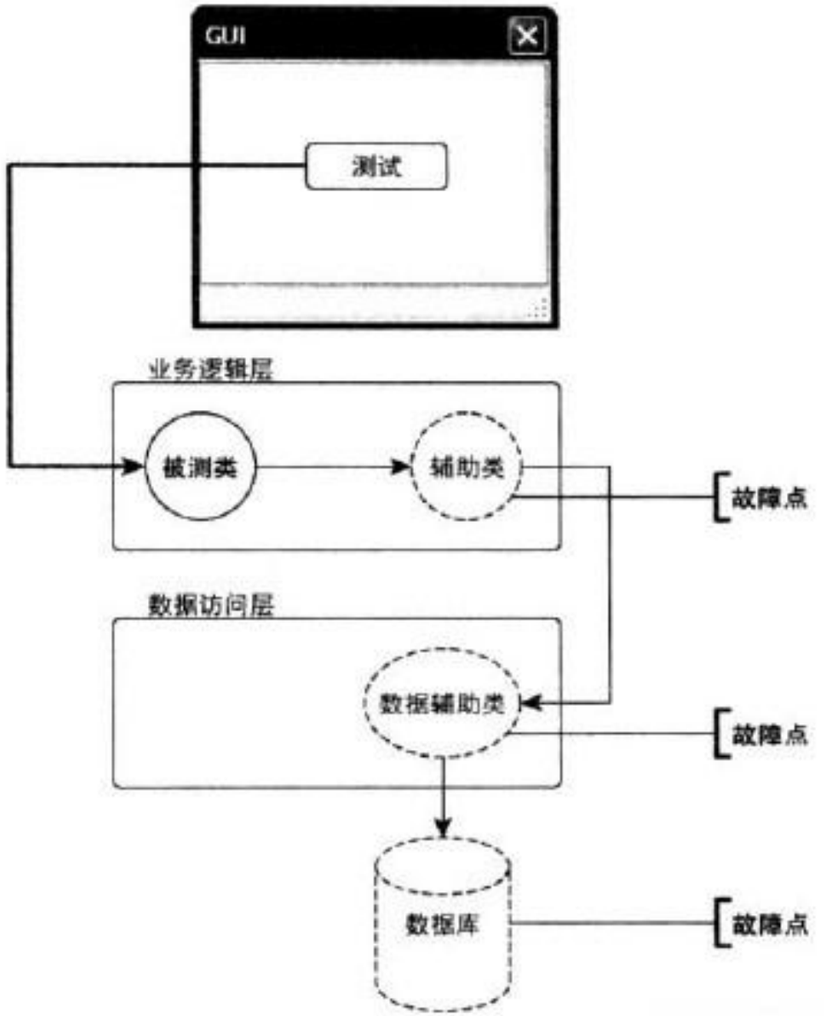
一个单元测试是一段 **自动化的代码** ，这段代码 **调用一个工作单元** ，之后对这个单元的单个 **最终结果** 的某些假设 **进行检验**。

一个单元测试的范围，可以小到一个方法，大到多个类。

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，能够快速运行。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。

与集成测试的区别

集成测试是对一个工作单元进行的、使用该单元的一个或多个真实依赖（例如时间、网络、数据库、线程或随机数生成器等）而对被测试的工作单元没有完全的控制的测试。



总的来说，集成测试会使用真实依赖物，而单元测试则把被测试单元和其依赖物隔离开，以保证单元测试结果高度稳定，还可以轻易控制和模拟被测试单元行为的任何方面。

单元测试框架

单元测试框架为开发者提供了类库，包括：

- 可以继承的基础类或接口；
- 用来标记测试方法的属性；
- 提供特殊断言方法的断言类用于验证代码；
- 提供一个**测试运行器**，用于发现代码中的测试、自动运行测试、运行期间显示状态、用命令行自动化，查看测试运行结果。

通常单元测试框架都以他们支持的语言的开头字母加上Unit作为名字，它们统称为xUnit框架。

C++的单元测试框架是CppUnit，Java的是Junit，.Net的有MSTest,NUnit,xUnit.net。不是所有的单元测试框架都这么命名，但是大部分如此。

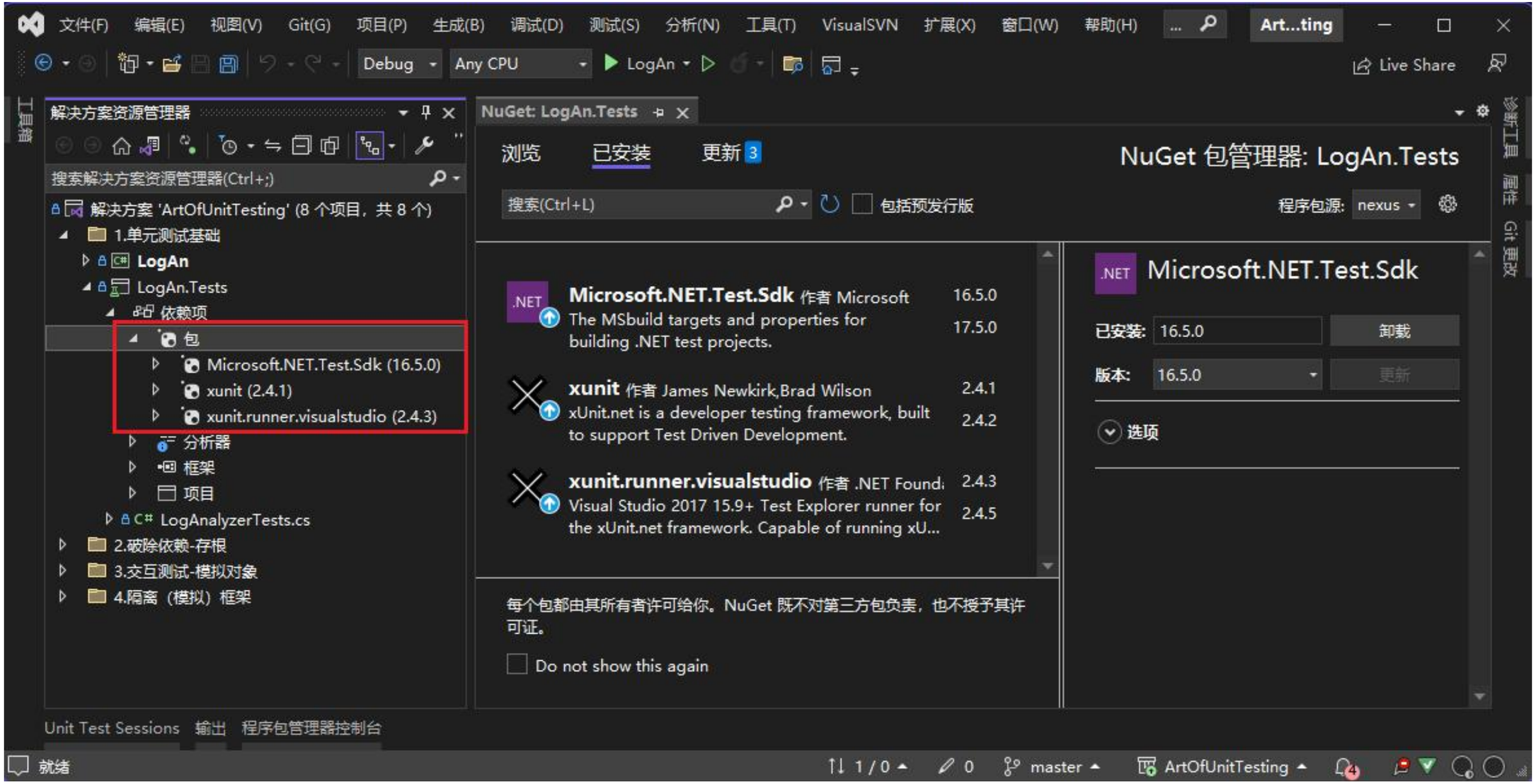
我们将使用.Net单元测试框架xUnit.net，它可以简化我们编写和运行测试，以及获得结果的工作。

.NET中xUnit对比

MSTest	NUnit	xUnit.net	Comments
[TestMethod]	[Test]	[Fact]	标记为测试方法
[TestClass]	[TestFixture]	n/a	包含有测试方法的类，在xUnit.net中无需标记，它会查找程序集中所有的public的测试方法
[ExpectedException]	Assert.Catch<Exception>	Assert.Throws<Exception>	期望抛出异常
[TestInitialize]	[SetUp]	Constructor	在每个测试方法执行之前用于初始化的方法
[TestCleanup]	[TearDown]	IDisposable.Dispose	在每个测试方法执行之后用于结束的方法
[ClassInitialize]	[TestFixtureSetUp]	IUseFixture<T>	在所有测试方法执行之前用于初始化的方法
[ClassCleanup]	[TestFixtureTearDown]	IUseFixture<T>	在所有测试方法执行之后用于结束的方法
[Ignore]	[Ignore]	[Fact(Skip="reason")]	临时忽略被标记的方法
[Timeout]	[Timeout]	[Fact(Timeout=n)]	用于指定被测试方法的最大执行时间（单位毫秒），如果超过指定时间则会被标记为测试失败
[DataSource]	[TestCase]	[Theory], [XxxData]	数据测试
AreEqual	AreEqual	Equal	相等比较
AreNotEqual	AreNotEqual	NotEqual	不相等比较
AreNotSame	AreNotSame	NotSame	不相同比较
AreSame	AreSame	Same	相同比较

安装单元测试框架

作为一名.NET程序员，如何在VS中安装xUnit.net并能够在VS中直接运行测试呢?



测试命名和位置的基本规则

测试对象	测试方创建对象
项目	创建一个名为【ProjectUnderTest】.Tests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	对每个工作单元，创建一个如下命名的测试方法： 【UnitOfWorkName】_【ScenarioUnderTest】_【ExpectedBehavior】

- UnitOfWorkName：被测试的方法，一组方法或者一组类。
- Scenario：测试进行的假设条件，例如“登录失败”“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“用户已存在”。
- ExpectedBehavior：在测试场景指定的条件下，你对被测试方法的预期。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，导致在下一次登录时系统的行为发生变化）。
  - 调用一个第三方系统（例如一个外部的Web服务）。

单元测试的三个行为

首先，我们要明确如何编写测试代码，一般来说，一个单元测试通常包含三个行为：



LogAn 项目介绍

LogAn （Log And Notificaition）

场景：公司有很多内部产品，用于在客户场地监控公司的应用程序。所有这些监控产品都会写日志文件，日志文件存放在一个特定的目录中。日志文件的格式是公司自己制定的，无法用现有的第三方软件进行解析。现在的任务是：实现一个产品，对这些日志文件进行分析，在其中搜索特定的情况和事件，这个产品就是LogAn。找到特定的情况和事件后，这个产品应该通知相关的人员。

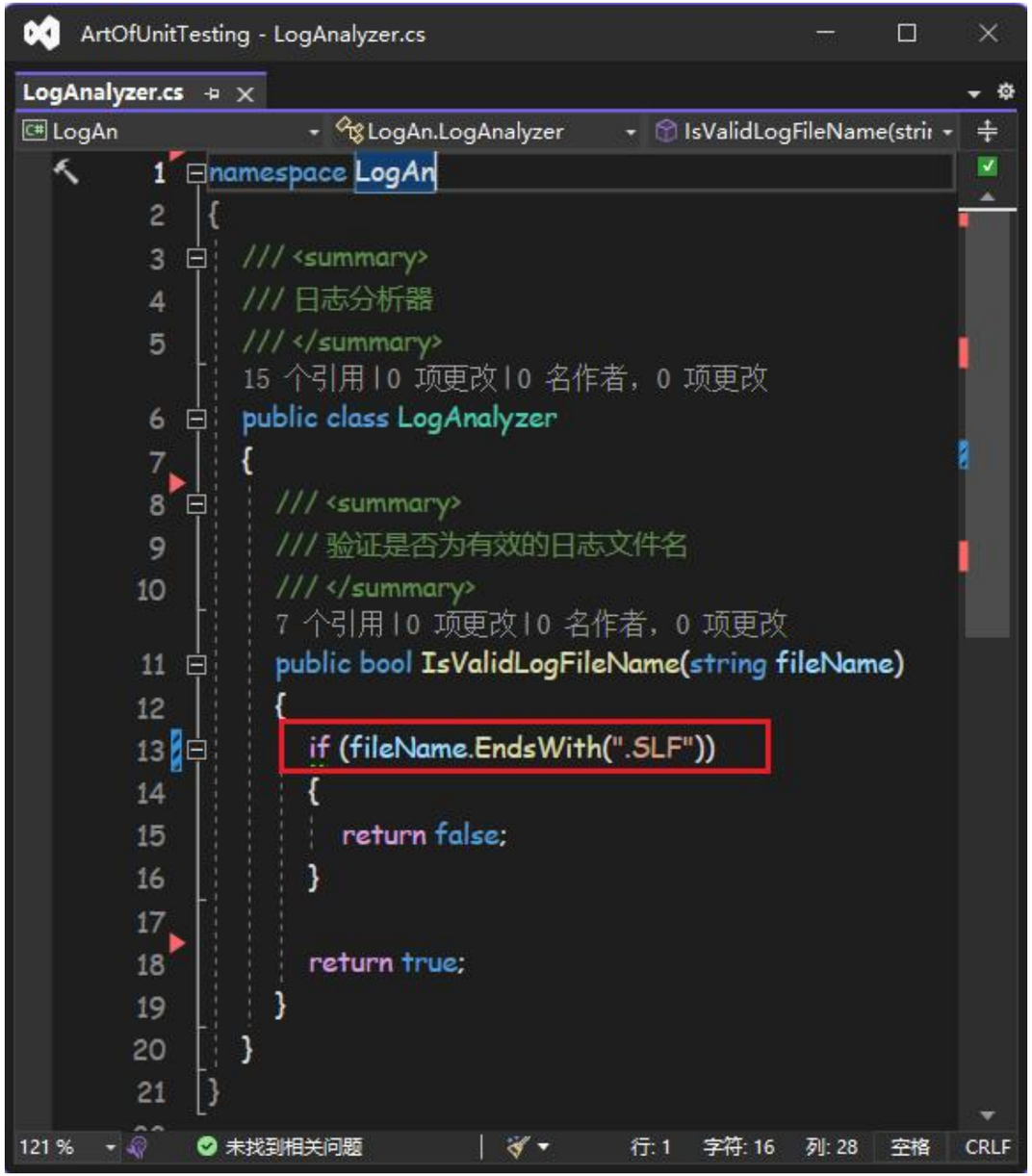
在本次的单元测试实践中，我们会一步一步编写测试来验证LogAn的解析、事件识别以及通知功能。首先，我们需要了解使用xUnit.net来编写单元测试。

编写第一个测试

（1）我们的测试从以下这个LogAnalyzer类开始，这个类暂时只有一个方法IsValidLogFileName：这个方法检查文件扩展名，据此判断一个文件是不是有效的日志文件。这里在if中故意去掉了一个!运算符，因此这个方法就包含了一个Bug-当文件名以.SLF结尾时会返回false，而不是返回true。这样，我们就能看到测试失败时在测试运行期中显示什么内容。

（2）新建一个类库项目，命名为LogAn.Tests（被测试项目名为LogAn）。添加一个类，取名为LogAnalyzerTests.cs

（3）在LogAnalyzerTests类中新增一个测试方法，取名为IsValidFileName_BadExtension_ReturnsFalse()



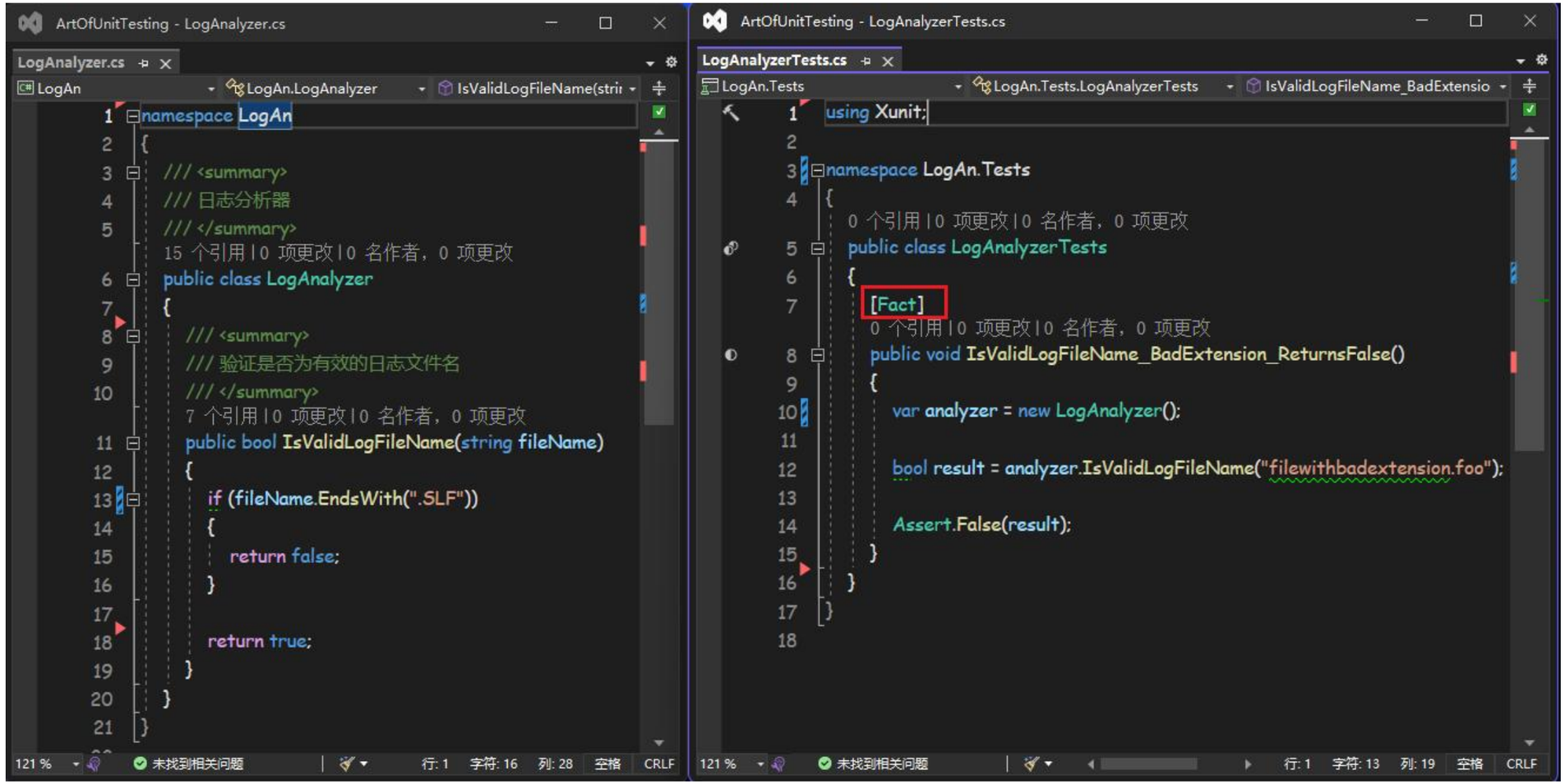
编写第一个测试

根据单元测试的三个行为，我们可以编写出以下的测试方法：（其中断言部分使用了xUnit.net框架提供的Assert类）

（4）[Fact]加在一个方法上，标识这个方法是一个需要调用的自动化测试；

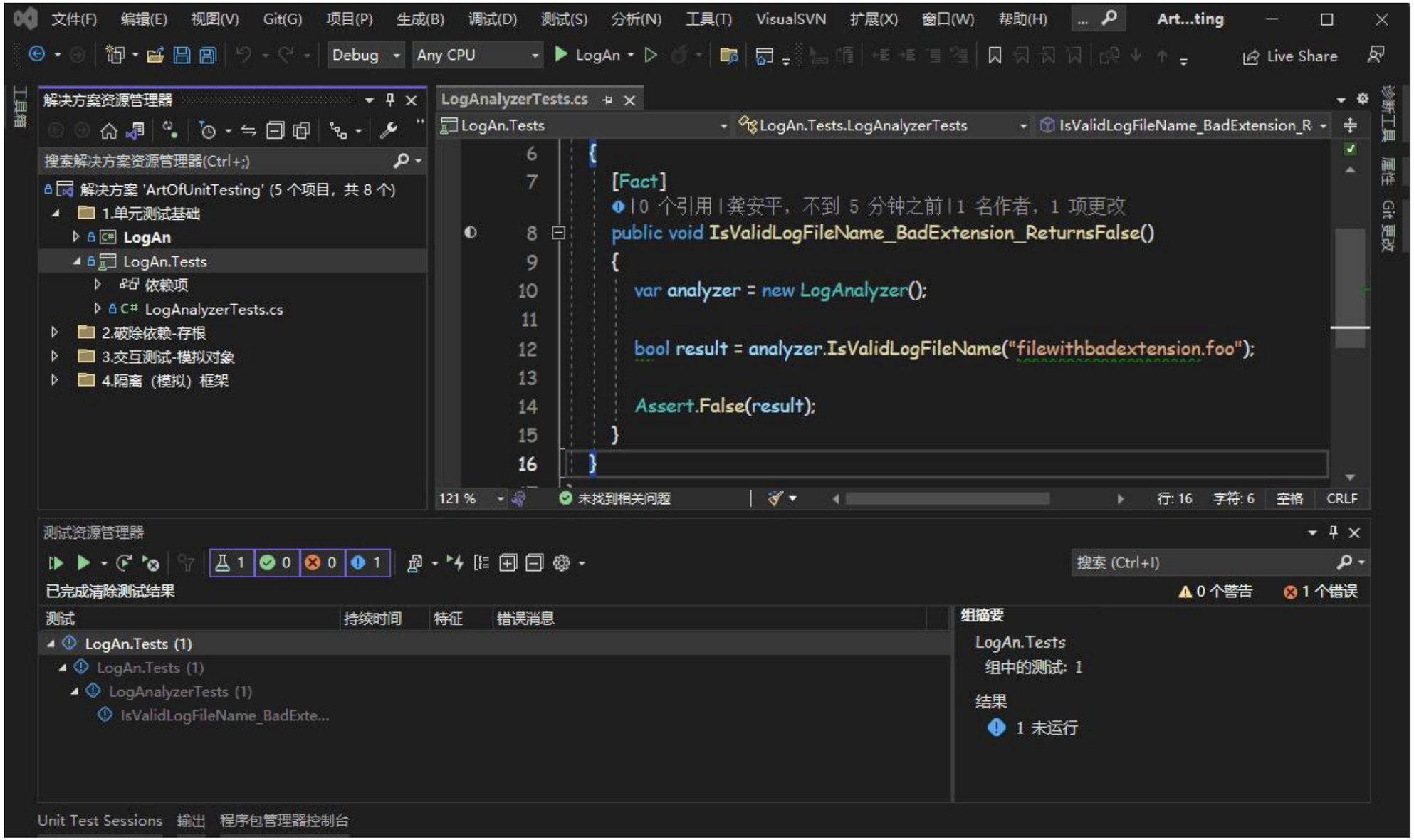
（5）编写好测试代码之后，点击"测试"->"运行"->"所有测试"

（6）然后，点击"测试"->"窗口"->"测试窗口管理器"，你会看到以下场景



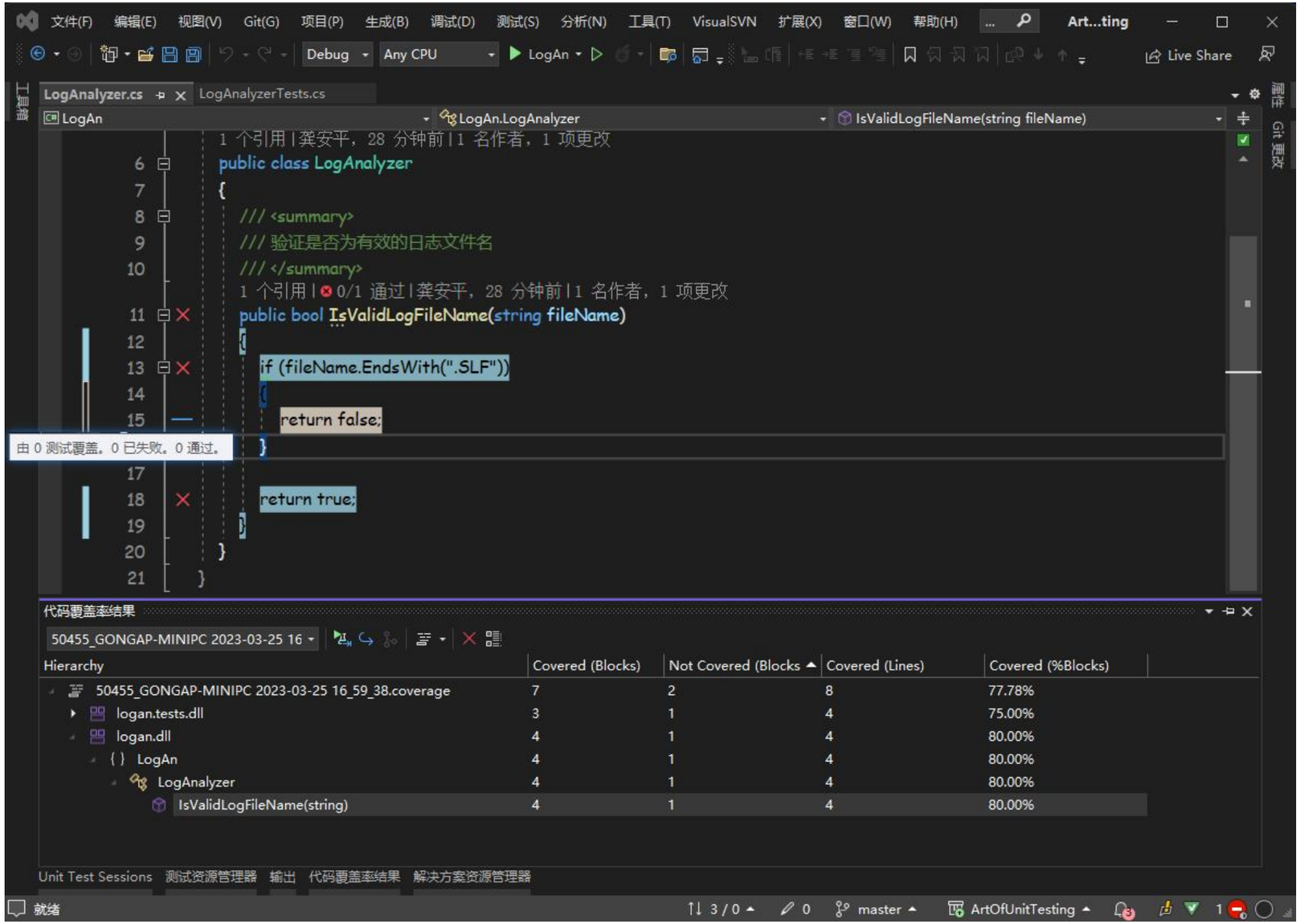
运行第一个测试

从下图可以看出，我们得测试方法并没有通过，我们期望（Expected）的结果是False，而实际（Actual）的结果却是True。



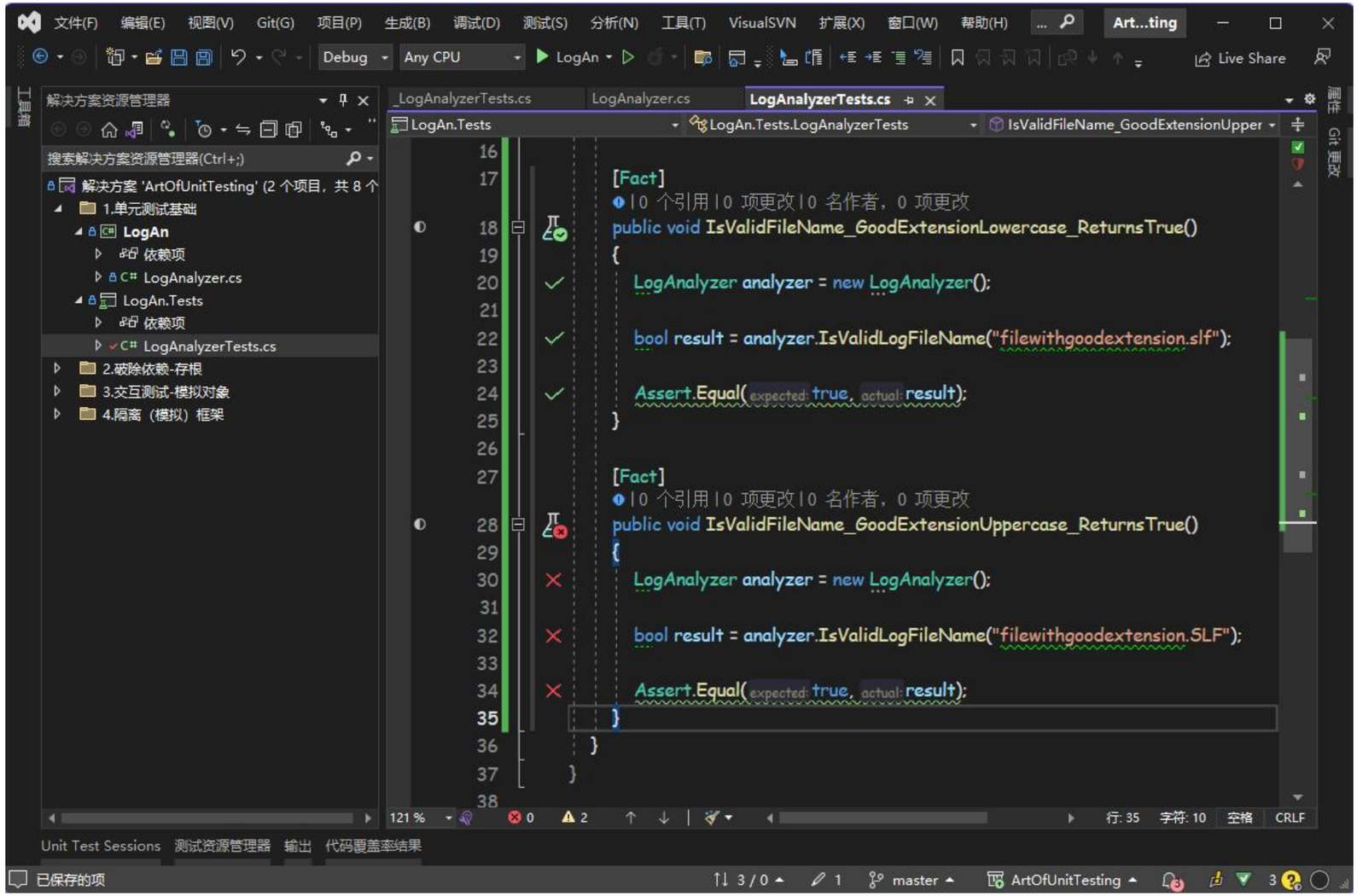
继续添加测试方法

（7）通常在进行单元测试时我们会考虑到**代码覆盖率**，点击"测试"->"分析代码覆盖率"->"所有测试"，你可以看到以下结果：**80%**



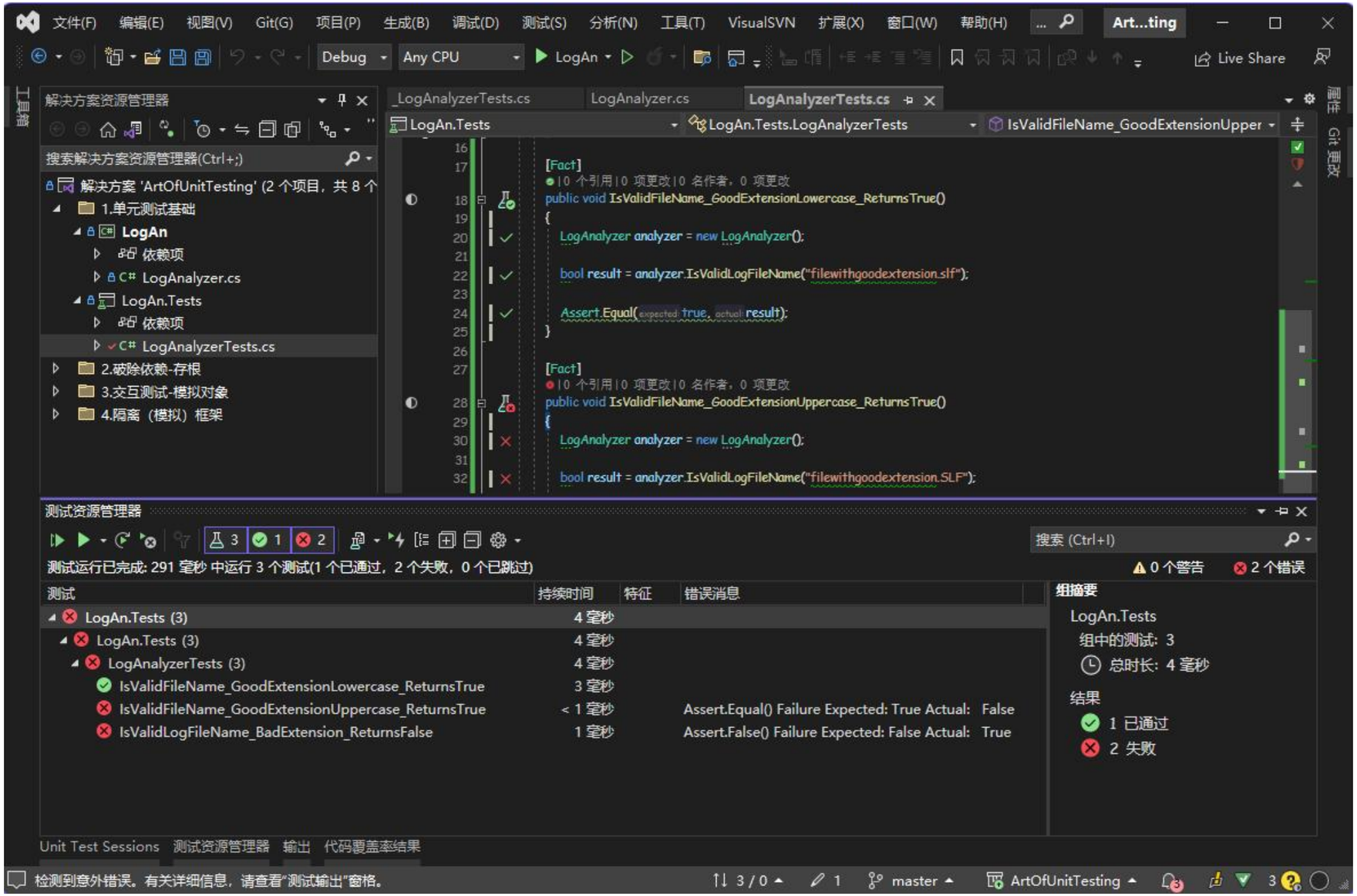
提高我们的代码覆盖率

（8）这时，我们需要想出完善的测试策略来覆盖所有的情况，因此我们添加一些测试方法来提高我们的代码覆盖率。这里我们添加两个方法，一个测试大写文件扩展名，一个测试小写文件扩展名：



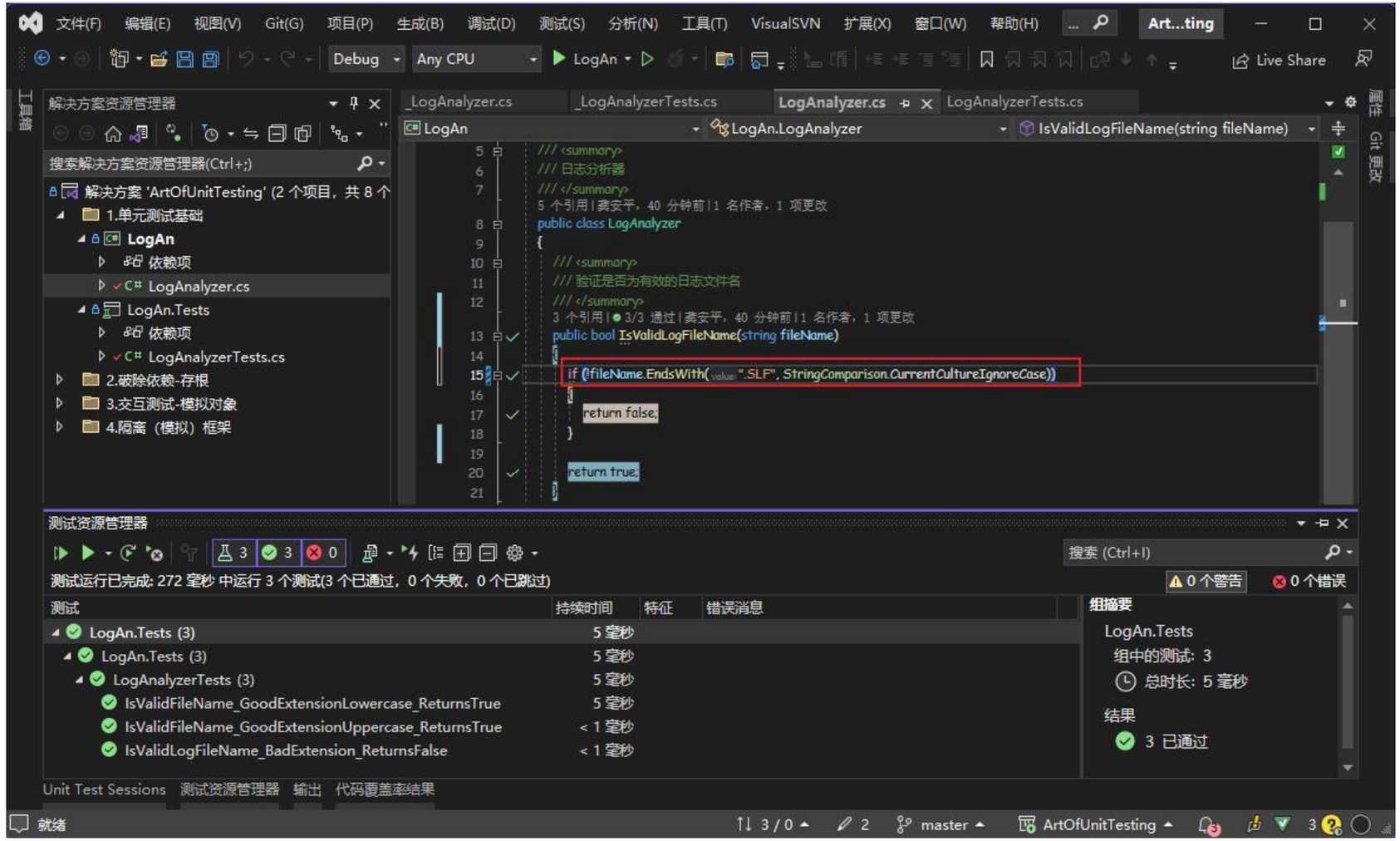
提高我们的代码覆盖率

这时测试结果如下图所示：



需要修改源代码

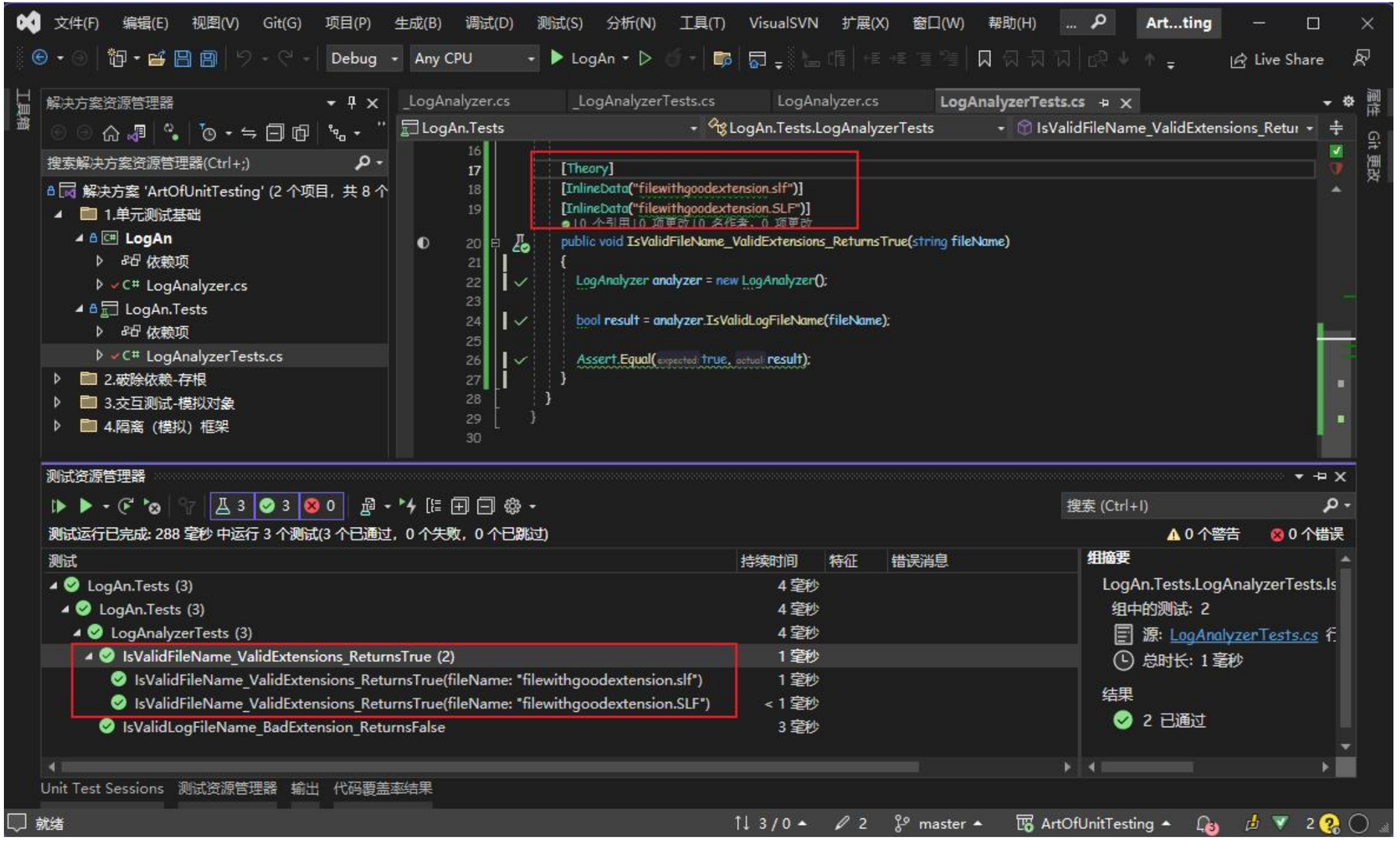
（9）为了让所有的测试都能通过，这时我们需要修改源代码，改用大小写不敏感的字符串匹配：



这时，我们再来运行一下所有的测试（也可以选择 运行未通过的测试）来看下**由红到绿**的快感。单元测试的理念很简单：只有所有的测试都通过，继续前行的绿灯才会亮起。哪怕只有一个测试失败了，进度条上都会亮起红灯，显示你的系统（或者测试）出现了问题。

参数化重构单元测试

xUnit.net中有个叫做 参数化测试（Parameterized Tests）的功能，我们可以借助[Theory]标签特性来重构我们的单元测试，可以看到，借助[InlineData]特性，测试数目没有改变，但是测试代码却变得更易维护，更加易读。



检验预期的异常

很多时候，我们的方法中会抛出一些异常，这时如果我们的测试也应该做一些修改。在xUnit.net中，提供了一个API：**Assert.Throws<T>(delegate)**

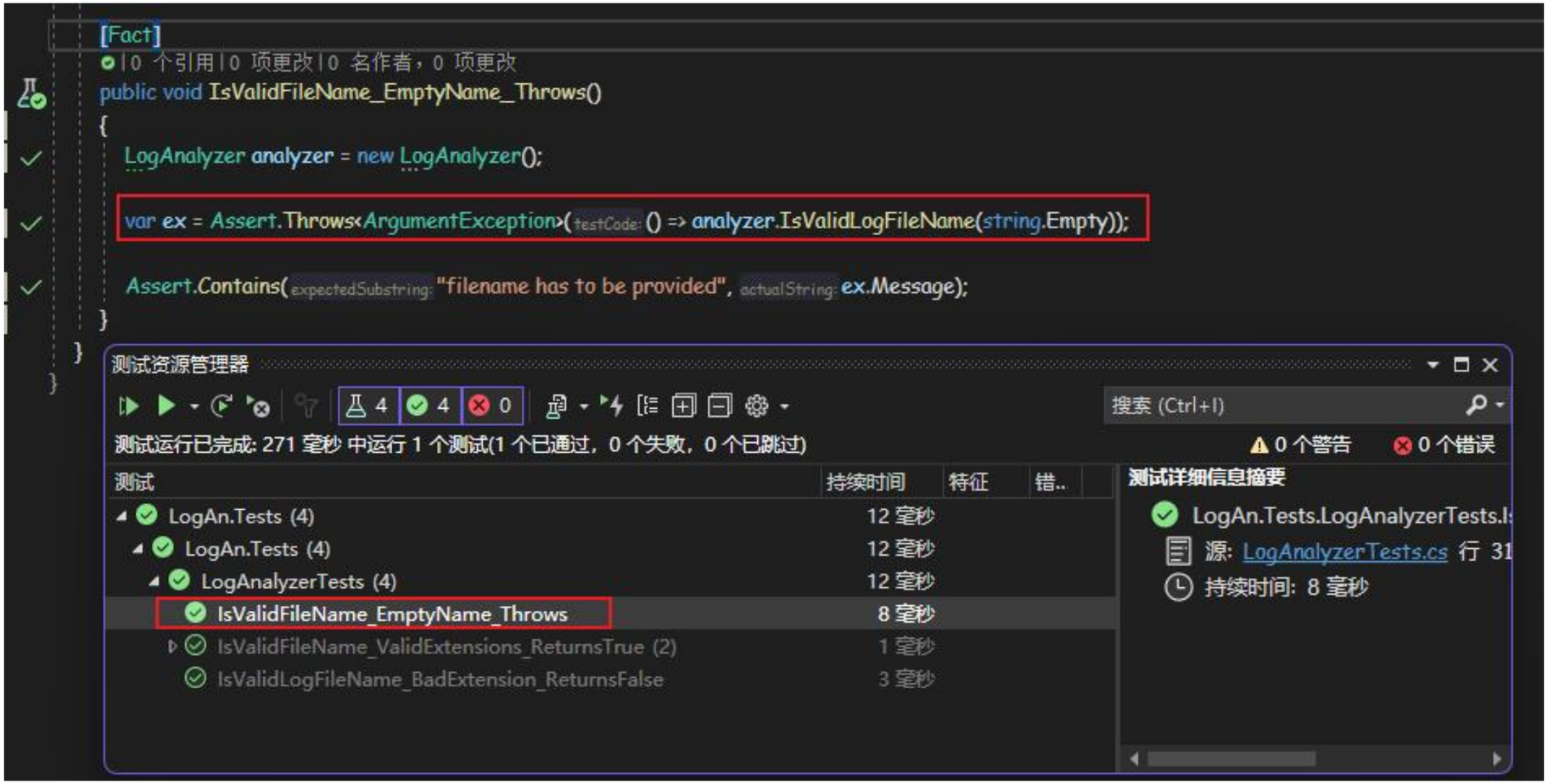
首先，我们修改一下被测试的方法，

增加一行判断文件名是否为空的代码：

```
3   namespace LogAn
4   {
5       /// <summary>
6       /// 日志分析器
7       /// </summary>
        5 个引用 | 龚安平，1 天前 | 1 名作者，1 项更改
8       public class LogAnalyzer
9       {
10          /// <summary>
11          /// 验证是否为有效的日志文件名
12          /// </summary>
            3 个引用 | 3/4 通过 | 龚安平，1 天前 | 1 名作者，1 项更改
13          public bool IsValidLogFileName(string fileName)
14          {
15              if (string.IsNullOrEmpty(fileName))
16              {
17                  throw new ArgumentException(message: "filename has to be provided");
18              }
19
20              if (!fileName.EndsWith(value: ".SLF", StringComparison.CurrentCultureIgnoreCase))
21              {
22                  return false;
23              }
24
25              return true;
26          }
27      }
28  }
29
```

检验预期的异常

然后，我们新增一个测试方法，使用Assert.Catch来检测异常是否一致：



测试系统状态的改变

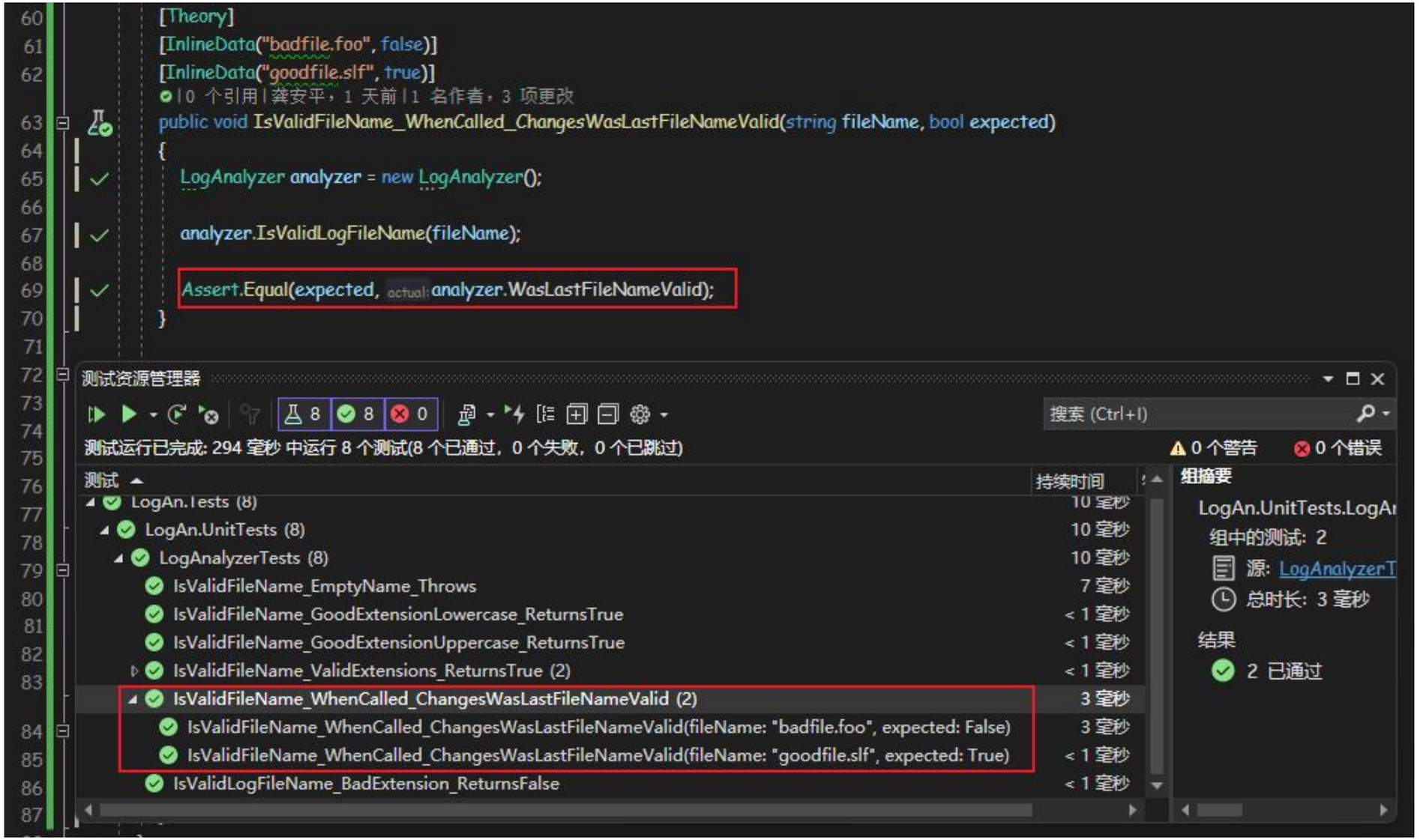
此前我们得测试都有返回值，而很多要测试的方法都没有返回值，而只是改变对象中的某些状态，我们又该如何测试呢？

首先，我们修改`IsValidLogFileName`方法，**增加一个状态属性**：

```
7   namespace LogAn
8   {
9       /// <summary>
10      /// 日志分析器
11      /// </summary>
        14 个引用 | 龚安平，1 天前 | 1 名作者，1 项更改
12      public class LogAnalyzer
13      {
            3 个引用 | 0/2 通过 | 龚安平，1 天前 | 1 名作者，1 项更改
14          public bool WasLastFileNameValid { get; set; }
15
16          /// <summary>
17          /// 验证是否为有效的日志文件名
18          /// </summary>
            6 个引用 | 0/8 通过 | 龚安平，1 天前 | 1 名作者，1 项更改
19          public bool IsValidLogFileName(string fileName)
20          {
21              // 改变系统状态
22              WasLastFileNameValid = false;
23
24              if (string.IsNullOrEmpty(fileName))
25              {
26                  throw new ArgumentException(message: "filename has to be provided");
27              }
28
29              if (!fileName.EndsWith(value: ".SLF", StringComparison.CurrentCultureIgnoreCase))
30              {
31                  return false;
32              }
33
34              // 改变系统状态
35              WasLastFileNameValid = true;
36
37              return true;
38          }
39      }
40  }
```

测试系统状态的改变

其次，我们编写一个测试，对系统状态进行断言：



目前为止，我们的单元测试都还很简单也还比较顺利。但是，如果我们要测试的方法依赖于一个**外部资源**，如文件系统、数据库、Web服务或者其他难以控制的东西，那又该如何编写测试呢？为了解决这些问题，我们需要创建测试**存根**、**伪对象**及**模拟对象**．下面我们会开始接触这些核心技术，借助存根**破除依赖**，使用模拟对象进行**交互测试**，使用隔离框架支持**适应**未来和可用性的功能。

二、破除依赖-存根

定义

- **外部依赖项**：一个外部依赖项是系统中的一个对象，被测试代码与这个对象发生交互，但你不能控制这个对象。（常见的外部依赖项包括：文件系统、线程、内存以及时间等）
- **存根**：一个存根（Stub）是对系统中存在的一个依赖项（或者协作者）的可控制的替代物。通过使用存根，你在测试代码时无需直接处理这个依赖项。
- **接缝**：接缝（Seam）是代码中可以插入不同功能的地方。通过重构添加接缝以使用伪对象。

为何使用存根

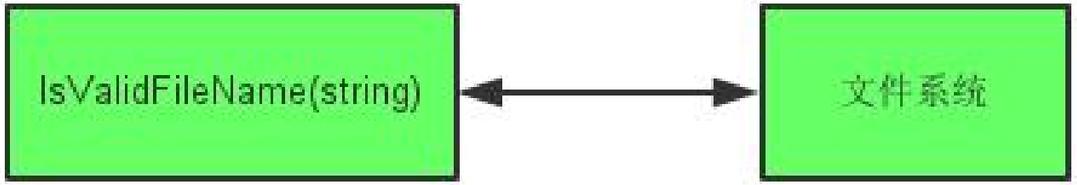
当我们要测试的对象依赖另一个你**无法控制**（或者还未实现）的对象，这个对象可能是Web服务、系统时间、线程调度或者很多其他东西。那么重要的问题来了：你的测试代码不能控制这个依赖的对象向你的代码返回什么值，也不能控制它的行为（例如你想摸你一个异常）。因此，这种情况下你可以使用**存根**。

发现项目中的外部依赖

继续上一篇中的LogAn案例，假设我们的IsValidLogFilename方法会首先读取配置文件，如果配置文件说支持这个扩展名，就返回true：

```
public bool IsValidLogFileName(string fileName)
{
    // 读取配置文件
    // 如果配置文件说支持这个扩展名，则返回true
}
```

那么问题来了：一旦测试依赖于文件系统，我们进行的就是集成测试，会带来所有与集成测试相关的问题—运行速度较慢，需要配置，一次测试多个内容等。



换句话说，尽管代码本身的逻辑是完全正确的，但是这种依赖可能导致测试失败。

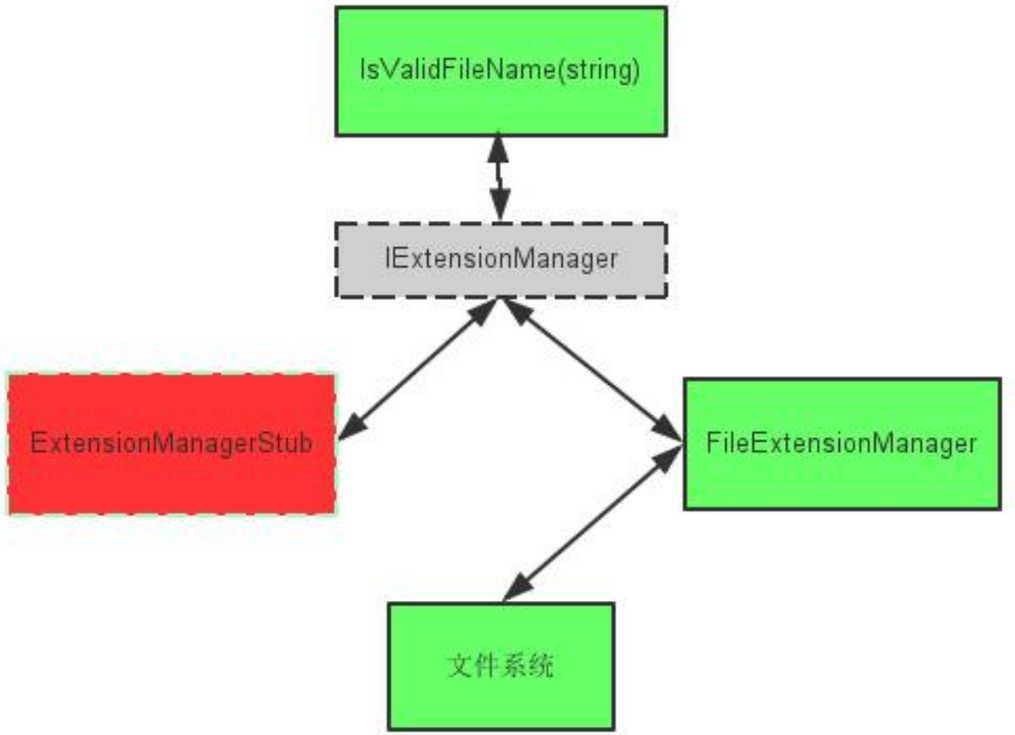
避免项目中的直接依赖

想要破除直接依赖，可以参考以下两个步骤：

（1）找到被测试对象使用的外部接口或者API；

（2）把这个接口的底层实现替换成你能控制的东西；

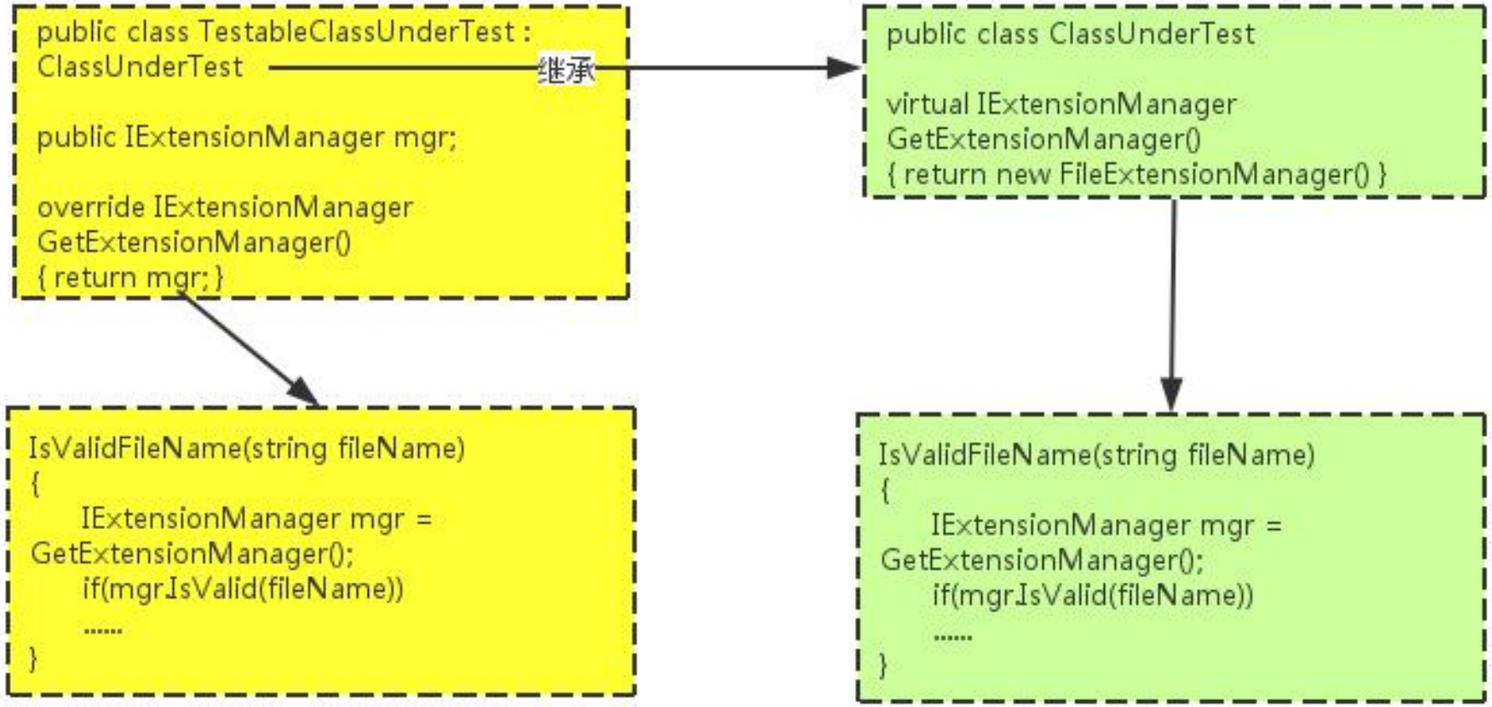
对于我们的LogAn项目，我们要做到替代实例不会访问文件系统，这样便破除了文件系统的依赖性。因此，我们可以引入一个间接层来避免对文件系统的直接依赖。访问文件系统的代码被隔离在一个FileExtensionManager类中，这个类之后将会被一个存根类替代，如下图所示：



在上图中，我们引入了存根 ExtensionManagerStub 破除依赖，现在我们的代码不应该知道也不会关心它使用的扩展管理器的内部实现。

抽取和重写

抽取和重写是一项强大的技术，可直接替换依赖项，实现起来快速干净，可以让我们编写更少的接口、更多的虚函数。



工作单元可能有三种最终结果，目前为止，我们编写过的测试只针对前两种：**返回值**和**改变系统状态**。现在，我们来了解如何测试第三种最终结果-**调用第三方对象**。

三、交互测试-模拟对象

定义

- **模拟对象** 是系统中的伪对象，它可以验证被测试对象是否按预期的方法调用了这个伪对象，因此导致单元测试通过或失败。通常每个测试最多有一个模拟对象。

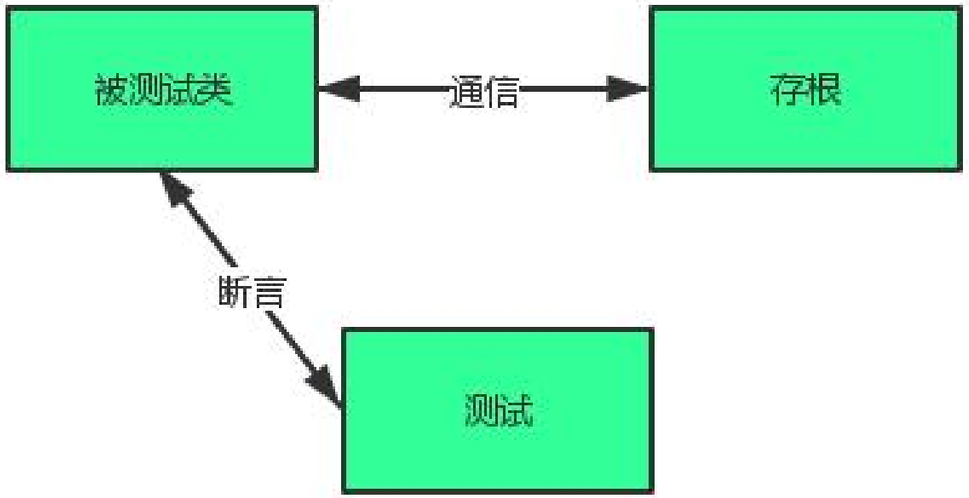
- **伪对象** 是通用术语，可以描述一个存根或者模拟对象（手工或者非手工编写）,因为存根和模拟对象看上去都很像真实对象。一个伪对象究竟是存根还是模拟对象**取决于它当前测试中的使用方法**：如果这个伪对象用来检验一个交互（**对其进行断言**），它就是**模拟对象**，否则就是**存根**。

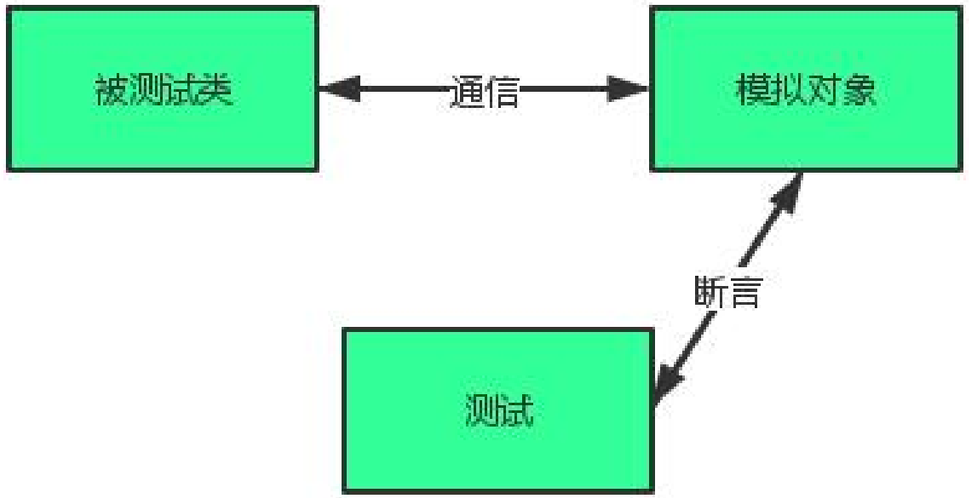
模拟对象与存根的区别

模拟对象和存根之间的区别很小，但二者之间的区别非常微妙，但又很重要。二者最根本的区别在于：

存根不会导致测试失败，而模拟对象可以。

下图展示了存根和模拟对象之间的区别，可以看到测试会使用模拟对象验证测试是否失败。



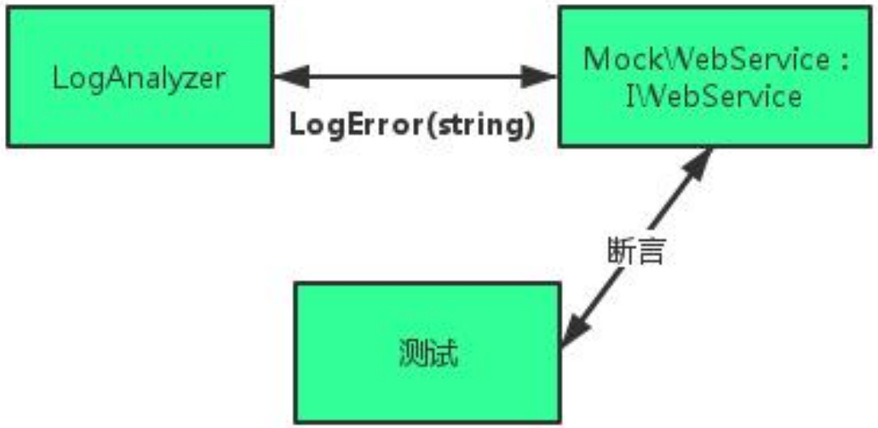


第一个手工模拟对象

创建和使用模拟对象的方法与使用存根类似，只是模拟对象比存根多做一件事：它保存通讯的历史记录，这些记录之后用于预期（Expection）验证。

假设我们的被测试项目LogAnalyzer需要和一个外部的Web Service交互，每次LogAnalyzer遇到一个过短的文件名，这个Web Service就会收到一个错误消息。遗憾的是，要测试的这个Web Service还没有完全实现。就算实现了，使用这个Web Service也会导致测试时间过长。

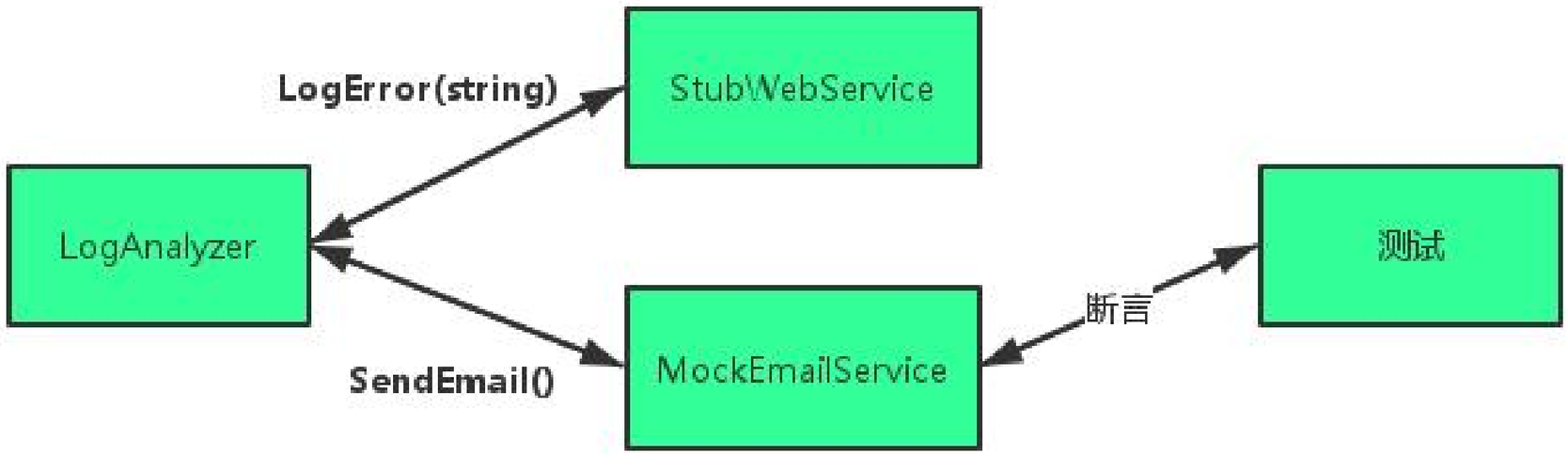
因此，我们需要重构设计，创建一个新的接口，之后用于这个接口创建模拟对象。这个接口只包括我们需要调用的Web Service方法。



测试会创建一个FakeWebService记录LogAnalyzer发送的消息，之后对FakeWebService进行断言

同时使用模拟对象和存根

我们可以在测试代码中使用存根替换Web Service来模拟异常，然后模拟邮件服务来检查调用。测试的内容是LogAnalyzer与其他对象的交互。



每个测试应该只测试一件事情，**测试中应该也最多只有一个模拟对象**。

一个测试只能指定工作单元三种最终结果中的一个，不然的话天下大乱。

四、隔离（模拟）框架

为何使用隔离框架

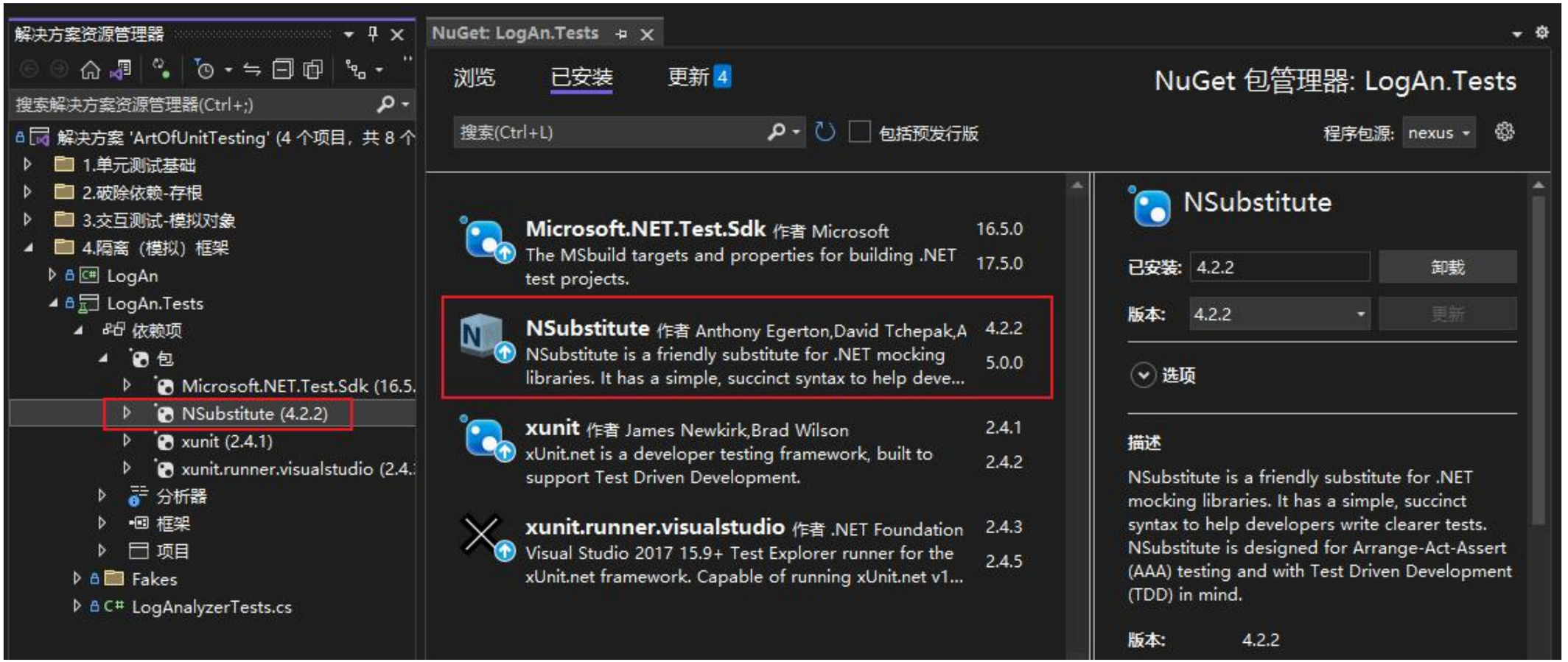
对于复杂的交互场景，可能手工编写模拟对象和存根就会变得很不方便，因此，我们可以借助隔离框架来帮我们在运行时**自动生成存根和模拟对象**。

一个隔离框架是一套可编程的*API*，使用这套*API*创建伪对象比手工编写容易得多，快得多，而且简洁得多。

隔离框架的主要功能就在于帮我们生成动态伪对象，动态伪对象是运行时创建的任何存根或者模拟对象，它的创建不需要手工编写代码（硬编码）。

关于NSubstitute隔离框架

Nsubstitute是一个开源的框架，源码是C#实现的。你可以在这里获得它的源码：
https://github.com/nsubstitute/NSubstitute
NSubstitute 采用的是Arrange-Act-Assert测试模式，你只需要告诉它应该如何工作，然后断言你所期望接收到的请求，就大功告成了。因为你有更重要的代码要编写，而不是去考虑是需要一个Mock还是一个Stub。在.NET项目中，我们仍然可以通过NuGet来安装NSubsititute：



使用NSubstitute模拟对象

NSub是一个受限框架，它最适合为接口创建伪对象。我们继续以前的例子，来看下面一段代码，它是一个手写的伪对象FakeLogger，它会检查日志调用是否正确执行。此处我们没有使用隔离框架。

```
7 个引用 | 0 项更改 | 0 名作者，0 项更改
public interface ILogger
{
    4 个引用 | 0/1 通过 | 0 项更改 | 0 名作者，0 项更改
    void LogError(string message);
}

3 个引用 | 0 项更改 | 0 名作者，0 项更改
public class LogAnalyzer
{
    private ILogger log;
    1 个引用 | 0/1 通过 | 0 项更改 | 0 名作者，0 项更改
    public LogAnalyzer(ILogger log)
    {
        this.log = log;
    }

    2 个引用 | 0/1 通过 | 0 项更改 | 0 名作者，0 项更改
    public int MinNameLength
    {
        get; set;
    }

    1 个引用 | 0/1 通过 | 0 项更改 | 0 名作者，0 项更改
    public void Analyze(string fileName)
    {
        if (fileName.Length < MinNameLength)
        {
            log.LogError( message: string.Format("Filename too short : {0}", fileName));
        }
    }
}
```

```
/// <summary>
/// 手写的伪对象FakeLogger
/// </summary>
/// <seealso cref="LogAn.ILogger" />
1 个引用 | 0 项更改 | 0 名作者，0 项更改
public class FakeLogger : ILogger
{
    public string LastError;
    4 个引用 | 1/1 通过 | 0 项更改 | 0 名作者，0 项更改
    public void LogError(string message)
    {
        LastError = message;
    }
}

0 个引用 | 龚安平，1 天前 | 1 名作者，1 项更改
public partial class LogAnalyzerTests
{
    [Fact]
    | 0 个引用 | 龚安平，1 天前 | 1 名作者，1 项更改
    public void Analyze_TooShortFileName_CallLogger()
    {
        var logger = new FakeLogger();
        LogAnalyzer analyzer = new LogAnalyzer(logger);
        analyzer.MinNameLength = 6;

        analyzer.Analyze( fileName: "a.txt");

        Assert.Contains( expectedSubstring: "too short", actualString: logger.LastError);
    }
```

使用NSubstitute模拟对象

现在我们看看如何使用NSub伪造一个对象，换句话说，之前我们手动写的FakeLogger在这里就不用再手动写了：

```
9       [Fact]
        0 个引用 | 龚安平，1 天前 | 1 名作者，1 项更改
10      public void Analyze_TooShortFileName_CallLogger()
11      {
12          // 创建模拟对象，用于测试结尾的断言
13          ILogger logger = Substitute.For<ILogger>();
14          LogAnalyzer analyzer = new LogAnalyzer(logger);
15          analyzer.MinNameLength = 6;
16
17          analyzer.Analyze(fileName: "a.txt");
18
19          // 使用NSub API设置预期字符串
20          logger.Received().LogError(message: "Filename too short : a.txt");
21      }
```

需要注意的是：

- ILogger接口自身并没有这个Received方法；
- NSub命名空间提供了一个扩展方法Received，这个方法可以断言在测试中调用了伪对象的某个方法；
- 通过在LogError()前调用Received()，其实是NSub在询问伪对象的这个方法是否调用过。

使用Nsubstitute模拟值

如果接口的方法返回不为空，如何从实现接口的动态伪对象返回一个值呢?

我们可以借助NSub强制方法返回一个值:



使用Nsubstitute模拟值

如果我们不想关心方法的参数，即无论参数是什么，方法应该总是返回一个价值，这样的话测试会更容易维护，因此我们可以借助NSub的参数匹配器：



`Arg.Any<Type>`称为参数匹配器，在隔离框架中被广泛使用，控制参数处理。

使用Nsubstitute模拟值

如果我们需要模拟一个异常，也可以借助NSub来解决：

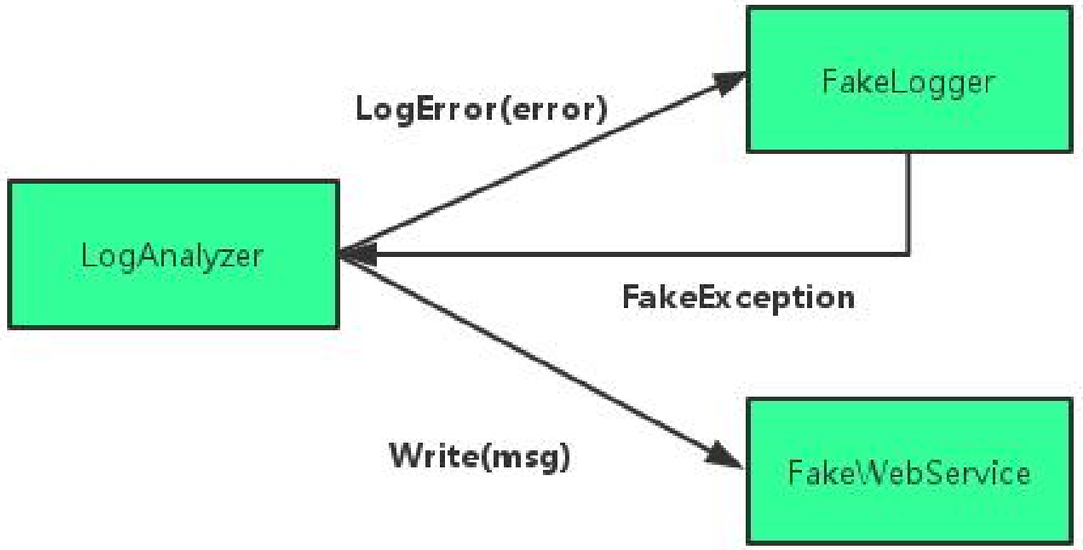


这里，使用了Assert.Throws验证被测试方法确实抛出了一个异常。When和Do两个方法顾名思义代表了什么时候发生了什么事，发生了事之后要触发其他什么事。需要注意的是，这里When方法必须使用Lambda表达式。

同时使用模拟对象和存根

这里我们在一个场景中结合使用两种类型的伪对象：一个用作存根，另一个用作模拟对象。

继续前面的一个例子，LogAnalyzer要使用一个MailServer类和一个WebService类，这次需求有变化：如果日志对象抛出异常，LogAnalyzer需要通知Web服务，
如下图所示：

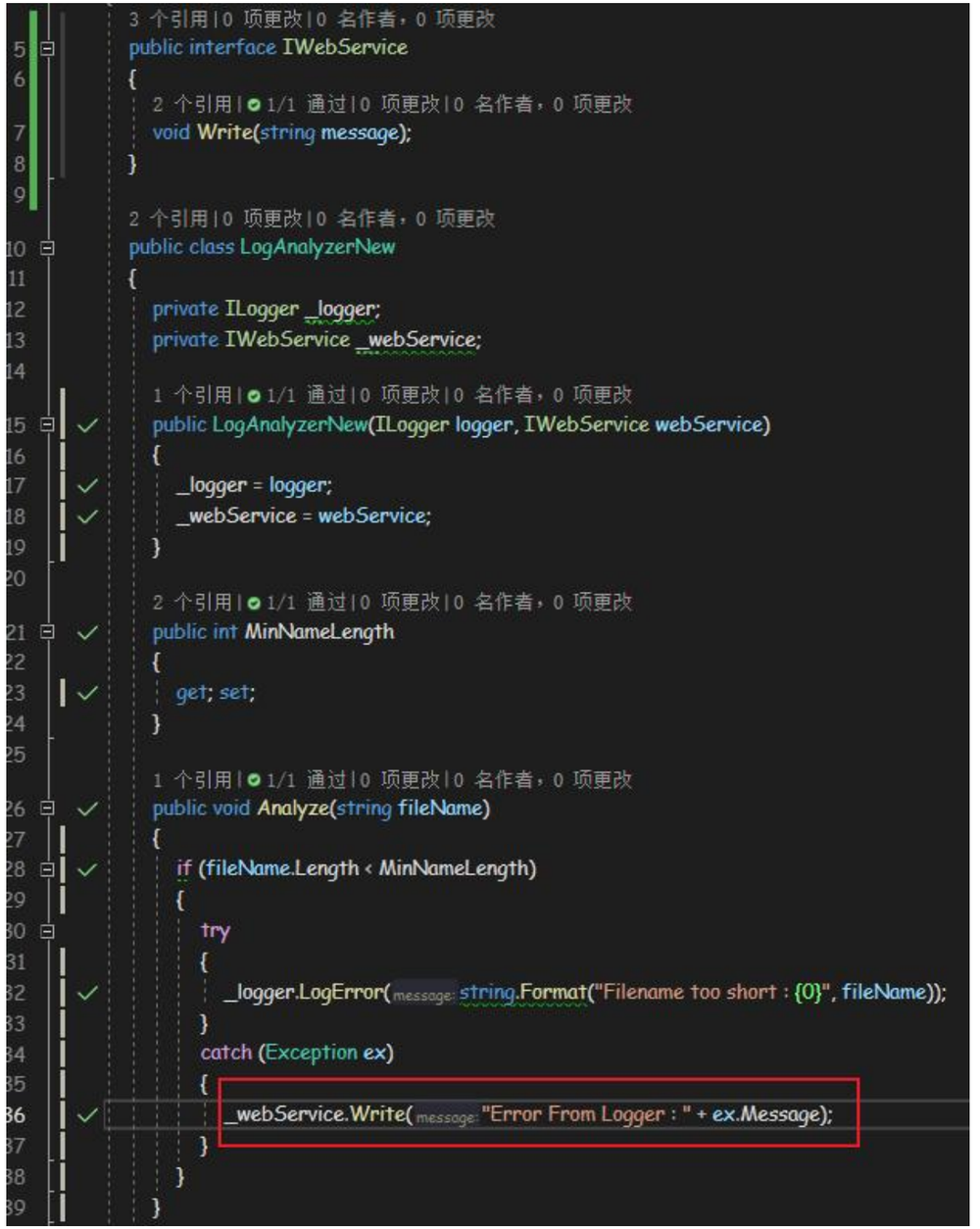


同时使用模拟对象和存根

我们需要确保的是：如果日志对象抛出异常，

LogAnalyzer会把这个问题通知WebService。

下面是被测试类的代码：



```
3 个引用 | 0 项更改 | 0 名作者，0 项更改
public interface IWebService
{
    2 个引用 | 1/1 通过 | 0 项更改 | 0 名作者，0 项更改
    void Write(string message);
}

2 个引用 | 0 项更改 | 0 名作者，0 项更改
public class LogAnalyzerNew
{
    private ILogger _logger;
    private IWebService _webService;

    1 个引用 | 1/1 通过 | 0 项更改 | 0 名作者，0 项更改
    public LogAnalyzerNew(ILogger logger, IWebService webService)
    {
        _logger = logger;
        _webService = webService;
    }

    2 个引用 | 1/1 通过 | 0 项更改 | 0 名作者，0 项更改
    public int MinNameLength
    {
        get; set;
    }

    1 个引用 | 1/1 通过 | 0 项更改 | 0 名作者，0 项更改
    public void Analyze(string fileName)
    {
        if (fileName.Length < MinNameLength)
        {
            try
            {
                _logger.LogError(message: string.Format("Filename too short : {0}", fileName));
            }
            catch (Exception ex)
            {
                _webService.Write(message: "Error From Logger : " + ex.Message);
            }
        }
    }
```

同时使用模拟对象和存根

现在我们借助NSubstitute进行测试：

```
59      // 同时使用模拟对象和存根
60      [Fact]
        0 个引用 | 龚安平，1 天前 | 1 名作者，1 项更改
61      public void Analyze_LoggerThrows_CallsWebService()
62      {
63          var mockWebService = Substitute.For<IWebService>();
64          var stubLogger = Substitute.For<ILogger>();
65
66          // 无论输入什么都抛出异常
67          stubLogger.When(substituteCall: logger => logger.LogError(message: Arg.Any<string>())) // WhenCalled<ILogger>
68              .Do(callbackWithArguments: info => { throw new Exception(message: "fake exception"); });
69
70          var analyzer = new LogAnalyzerNew(stubLogger, mockWebService);
71          analyzer.MinNameLength = 10;
72          analyzer.Analyze(fileName: "short.txt");
73
74          //验证在测试中调用了Web Service的模拟对象，调用参数字符串包含 "fake exception"
75          mockWebService.Received().Write(message: Arg.Is<string>(s :string => s.Contains("fake exception")));
76      }
```

这里我们不需要手工实现伪对象，但是代码的可读性已经变差了，因为有一堆Lambda表达式，不过它也帮我们避免了在测试中使用方法名字符串。

五、优秀单元测试的支柱

编写可靠的测试

一个可靠的测试能让你觉得自己对事态了如指掌，能够从容应对。以下是一些指导原则和技术：

- **决定何时删除或修改测试**
- **避免测试中的逻辑**
- **只测试一个关注点**
- **把单元测试和集成测试分开**
- **用代码审查确保代码覆盖率**

编写可维护性的测试

可维护性是大多数开发者在编写单元测试时面对的核心问题之一。为此我们需要：

- **只测试公共契约**
- **删除重复测试（去除重复代码）**
- **实施测试隔离**

测试隔离的基本概念是：**一个测试应该总是在它自己的小世界中运行，与其他类似或不同的工作的测试隔离，甚至不知道其他测试的存在。**

编写可读性的测试

不可读的测试几乎没有任何意义，它是我们向项目的下一代开发者讲述的故事，帮助开发者理解一个应用程序的组成及其开端。

- **单元测试命名：**这个前面我们讨论过，应该包括三部分：被测试方法名_测试场景_预期行为，如果开发人员都是用这种规范，其他的开发人员就能很容易进入项目，理解测试。

- **变量命名：**通过合理命名变量，你可以确保阅读测试的人可以尽快地理解你要验证什么（相对于理解产品代码中你想要实现什么）。

- **有意义的断言：**只有当测试确实需要，并且找不到别的办法使测试更清晰时，你才应该编写定制的断言信息。编写好的断言信息就像编写好的异常信息，一不小心就会犯错，使读者产生误解，浪费他们的时间。

- **断言和操作分离：**为了可读性，请不要把断言和方法调用写在同一行。

总结

总结

本次培训我们学习了：

- **单元测试的概念**：如何编写单元测试，如何在VS中应用xUnit进行单元测试。
- **单元测试的核心技术**：存根、模拟对象以及隔离框架。使用存根可以帮助我们破除依赖，模拟对象与存根的区别主要在于存根不会导致测试失败，而模拟对象则可以。要辨别你是否使用了存根，最简单的方法是：存根永远不会导致测试失败，测试总是对被测试类进行断言。使用隔离框架，测试代码会更加易读、易维护，重点是可以帮助我们节省不少时间编写模拟对象和存根。
- **优秀单元测试具有三大支柱**：可读性、可维护性与可靠性，它们相辅相成。

如果别人能读懂你的测试，就能理解和维护测试，如果测试能够通过，它们也会信任测试。一旦实现这个目标，你就能知道系统是否正常工作，具有了处理变更和在需要时修改代码的能力；

本培训的示例代码：点此下载

谢　谢！