Broadview
www.broadview.com.cn

模式

王翔 孙逊 著

现及扩展

设计模式C#版）

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

模式工程化实现及扩展

王翔　孙逊　著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

设计模式不是一门适合空谈的技术，它来自于开发人员的工程实践又服务于工程实践。

本书并不是一本面向入门者的读物，因为它需要结合工程实践介绍如何发现模式灵感、如何应用模式技术。不过作为一本介绍设计模式的书，它并不需要读者对于庞大的.NET Framework 有深入了解，因为扩展主要是结合 C#语法完成的，配合书中的实例，相信读者不仅能够熟练应用设计模式技术，也能令自己的 C#语言上一个台阶。

为了降低学习门槛，本书第一部分除了介绍面向对象设计原则外，还充实了一些 C#语言的介绍，但这些内容并不是枯燥的讲解，读者可以在阅读中通过一系列动手练习尽快吸收这些理论，并将它们转化为自己的技能。本书最后一部分的“GOF 综合练习”把各种设计模式进行了一次集中展示，目的是让读者把分散的模式知识融合在一起，能够将书本知识真正用于改善一个“准”生产型模块的实现。

本书内容生动，示例贴近中型、大型项目实践，通过一个个“四两拨千斤”的示例练习可以让读者有一气读完的兴趣。

图书在版编目（CIP）数据

模式——工程化实现及扩展：设计模式 C#版 / 王翔，孙逊著. —北京：电子工业出版社，2012.4
ISBN 978-7-121-15639-7

Ⅰ. ①模… Ⅱ. ①王… ②孙… Ⅲ. ①C 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2011）第 280710 号

策划编辑：张春雨
责任编辑：付　睿
特约编辑：赵树刚
印　　刷：
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编 100036
开　　本：787×1092　1/16　印张：25.5　字数：653 千字
印　　次：2012 年 4 月第 1 次印刷
印　　数：4000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

出版说明

丛书内容

本系列在《设计模式——基于 C#的工程化实现及扩展》的基础上充实完成，本系列此版本面向 C#开发者和 Java 开发者提供如下 4 册图书：

- 模式——工程化实现及扩展（设计模式 C# 版）。
- 模式——工程化实现及扩展（架构模式 C# 版）。
- 模式——工程化实现及扩展（设计模式 Java 版）。
- 模式——工程化实现及扩展（架构模式 Java 版）。

其中，Java 版本采用 Oracle Java 7 编写，面向 Java 平台的专业人员；而 C#版本采用 Microsoft C# 4 编写，面向.NET 平台的专业人员。

设计模式版本仅介绍 GOF 23 个设计模式的内容，示例情景也主要面向进程内的类型设计；而架构模式版本则从分布式应用角度介绍部分典型的架构模式、数据访问模式、信息安全模式，尽管类型不同，但均为架构所必须的，所以统一纳入架构模式分册，其示例情景一般也存在跨进程调用，采用的也是分布式组件技术。

丛书定位

不管是本系列的 Java 版本还是 C#版本，是设计模式版本还是架构模式版本，**它都不是面向入门者的读物。**

它的内容是针对中高级开发人员安排的，而部分章节后面的“自我检验”环节，则是针对希望进一步挑战模式技术的软件架构师设计的。

内容要点

本系列的重点不是阐述设计模式 GOF 和架构模式本身，这些内容应该从 *Design Patterns : Elements of Reusable Object-Oriented Software* 获得。本系列面向的是模式的实践者。它假设读者已经熟悉 Java 或 C#语法，能熟练运用 Java 5/6/7 和 C# 3/4 语法特性，并能充分发掘其中的实践价值，确保工作成果在 JVM 和 CLR&DLR 平台上更简洁、明快地实现。本系列还假设读者对 Java 和 C#的编译器有所了解，对

于 Byte Code 或 MSIL 有一定的基础，对于一些要点内容可以从 Byte Code 或 MSIL 层面对交付成果进行评析。

在没有其他说明的情况下，本系列各分册所指的《设计模式》是 *Design Patterns : Elements of Reusable Object-Oriented Software*，即翻译为《设计模式：可复用面向对象软件的基础》这本书。

对于那些已经融入语言或开发平台的模式，例如，GOF 中的原型模式、命令模式、策略模式、观察者模式、迭代器模式、访问者模式和架构模式中的 MVC 模式、管道过滤器模式等，我们将尽量采用语言自身的机制完成，读者可能甚至看不到明显的“模式”痕迹，模式的经典实现仅大体介绍模式意图后简要带过。毕竟，工程中“暴殄天物”不值得提倡。

UML 标注特点

为了简化类图的篇幅，本系列在所有属性方法（Java 中表示访问 Field 的 getXxx/setXxx 方法，C#中表示 get/set 的 Property）中都采用简略方式定义，而且命名统一采用 Pascal Case 风格。比如图 F1-1 所示类图，等价的 Java 代码和 C#代码如下：

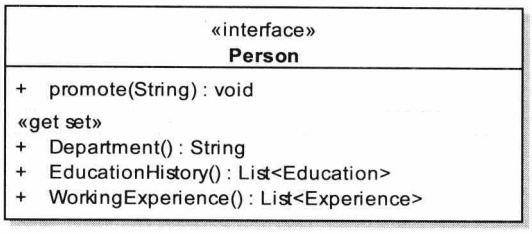

图 F1-1 示例类图

Java

```
public interface Person {
    void promote(String newTitle);
    public String getDepartment();
    public void setDepartment();
    public List<Education> getEducationHistory();
    public void setEducationHistory(List<Education> educationHistory);
    public List<Experience> getWorkingExperience();
    public void setWorkingExperience(List<Experience> workingExperience);
}
```

C#

```
public interface IPerson
{
   void Promote(string newTitle);
   string Department { get; set; }
```

```
    IList<Education> EducationHistory { get; set; }
    IList<Experience> WorkingExperience { get; set; }
}
```

示例特点

首先，所有示例代码的实现，保持简洁风格，不过，考虑到设计模式要解决“变化”问题，所以示例在保持简洁的同时，也适当照顾到“应变”的要求。

由于本书很多介绍都要结合代码，为了尽量减少篇幅，书中的代码基本都做了精简：

- 许多接口和类型定义都采用了很不规范的单行书写方式。
- 注释一般也进行了裁剪。
- 对于与展示实现思路无关的代码、配置文件、示例数据也简略带过。

例如，下面的 Java 代码和 C#代码。如果读者平时负责代码复查或者对代码的书写方式很敏感，笔者对读者可能会感到的不畅致歉。

Java

```
/**抽象定义部分*/
interface NamedState{ String getName();}

abstract class NamedStateBase implements NamedState{
    protected String name;

    protected NamedStateBase(String name){ this.name = name;}
    public String getName(){return this.name;}
}

/**具体类型*/
class ClosedState extends NamedStateBase{…}
class OpeningState extends NamedStateBase{…}
class OpenedState extends NamedStateBase{…}
```

C#

```
/// 具体策略类型
class GetMinSortStrategy : IStrategy{…}
class FirstDataStrategy : IStrategy{…}
class GetMaxStrategy : IStrategy{…}

/// 需要采用可替换策略执行的对象
public class Context : IContext
{
    public IStrategy Strategy { get; set; }

    /// 执行对象依赖于策略对象的操作方法
     public int GetData(int[] data)
     {

        if (strategy == null) throw new NullReferenceException("strategy");
```

```
        if (data == null) throw new ArgumentNullException("data");
        return strategy.PickUp(data);
    }
}
```

另外，在每个模式的实现方式上，针对不同的扩展主题本系列将始终沿用这样的优先级编码实现：

- Java 和 C#语法特性。
- 官方 Java SE JDK 和.NET Framework 的框架内容。
- 个别情况考虑集成官方资助的开源框架。
- 对于其他第三方提供的开源框架，不列入或不引入本系列的实现示例。

本文中“官方”对 Java 指 Oracle，对 C#指微软。在没有其他说明的情况下，本系列各分册所指的《.NET 设计规范》是微软 MSDN 文档中发布的.NET 设计规范 *Design Guidelines for Developing Class Libraries*（*http://msdn.microsoft.com/en-us/library/ms229042.aspx*），而《Java 语言编码规范》指的是 Oracle 发布的 *Code Conventions for the Java Programming Language*（*http://www.oracle.com/technetwork/java/codeconvtoc-136057.html*）。

如何阅读

本系列图书每章都会先对模式的经典介绍做简要评述，说明其主要意图和典型实现示例，然后会结合语言逐步扩充其使用情景，介绍典型的实现技巧。

为了便于读者使用，建议读者按照如下顺序阅读本系列各分册图书：

1）了解经典的面向对象设计原则。

2）在熟悉所用编程语言的基础上，投入一定的时间阅读“重新研读 Java 语言”或“重新研读 C#语言”两章，了解工程化代码与“玩具代码”的实现差异。

3）阅读各章对模式经典内容的介绍，了解为什么需要这个模式、它主要解决什么问题、经典的实现是怎样的。

4）阅读每章的扩展部分，熟悉不同示例环境下如何用更工程化的方式实现扩展要求。为了节省篇幅，书中代码只是代码片段节选，建议读者直接运行示例代码，这样体会更深。

5）分析和体会单元测试的结果。

6）逐步结合自己的项目修改示例代码，真正将书中的“工程化实现和扩展”融入实际工作中。

为了便于读者考查自己对于特定模式的掌握程度，本版本在部分章节增加了一个“自我检验”的练习环节，目的是通过一些准案例的介绍，让读者灵活应用该章的知识内容，独立思考解决方案。由于解决方案会不断调整和优化，所有的参考答案会发布在笔者的个人博客，解决方案内容也会根据与大家交流的结果不断优化、动态更新。希望这种方式读者能够喜欢。下面是笔者的个人博客，欢迎读者常来看看：

- CSDN http://blog.csdn.net/firevision/
- 博客园 http://www.cnblogs.com/callwangxiang/
- ITEye http://callwangxiang.iteye.com/

最后，祝大家有更好的阅读体验。

前　言

如同每个人都有其个性一样，每种开发语言也有自己的个性。

项目中，我们固然可以机械地将一种语言的开发经验套用到另一种语言，但效果不一定好，因为

- 语言有自己的短处：用短处去实现不仅费时费力，结果也不理想。
- 语言有自己的长处：但为了沿用以前的经验削足适履，没有用到语言的精要，结果暴殄天物。

相信读者也发现了，用一个语言写 Hello World 是一回事，写一个应用是一回事，写好一个应用则完全是另一回事，这就是工程化代码和“玩具”代码的区别。教科书上的知识落实到工程上时不能按图索骥，需要考虑开发语言和目标环境，设计模式也不例外。

也许读者会觉得本书很多实现方式与《设计模式》介绍的内容不一致，但别忘了《设计模式》一书出版至今已近 20 年，其间无论是开发语言还是技术平台已经“换了人间”，GOF 23 个模式的思想不仅影响着我们，更影响着走在技术前沿的语言设计者、平台设计者。他们也在工作中潜移默化地把模式思想融入自己的工作成果。作为用户，如果我们“推却”别人的盛情，所有事情都从“车轮”做起，多少有点不经济。

作为本系列的 C#设计模式分册，我试图用最 C#的方式将自己对于设计模式的理解呈献给读者，而且实现上务求简洁、直接。结构上，本书分为以下 5 个部分。

- 第一部分，预备知识

包括面向对象设计原则中“面向类”的部分、C#语言面向对象扩展特性，以及 Java 和 C#语法特性的简单对比。

- 第二部分，创建型模式

主要介绍如何创建对象，如何将客户程序与创建过程的“变化”有效隔离。

- 第三部分，结构型模式

从静态结构出发，分析导致类型结构相互依赖的原因，通过将静态变化因素抽象、封装为独立对象的办法，梳理对象结构关系。

- 第四部分，行为型模式

从动态机制出发，分析导致类型调用过程的依赖因素，通过将调用关系、调用过程抽象、封装为独立对象的办法，削弱调用过程中的耦合关系。

- 第五部分，GOF 综合练习

为了便于读者从整体上体会模式化设计思路和实现技巧，这部分通过一个综合性的示例向读者展示如何发现变化、抽象变化、应用模式并最终结合.NET Framework 平台特性加以实现的过程。

不管读者之前对于模式是否有所尝试，我希望读者不妨浏览这章，毕竟模式思想转化为模式设计思路，再转化为模式应用技巧是一个渐进的过程，必须实际动手才会加深印象，然后才可能进一步开阔思路。本章示例设计上变化因素较多，需要三类模式的综合运用，务求能起到抛砖引玉的效果。

感谢多年培养、帮助我的领导和同事们，多年富有挑战、共同拼搏的项目经历使我能够完成这本书。

感谢我和我妻子共同的父母，您们一直给予我无私的关心和照顾，还教会我学会从生活中发掘无穷的技术灵感。

最后，感谢我挚爱的妻子，你给予我直面挑战、战胜挑战的信心和力量。

不过，受到开发年限和项目经验的限制，本书在很多地方难免会有疏漏和不足之处，希望能够听到读者的批评和建议。

王翔

示例项目使用说明

解决方案（Solution）的创建和整理

1. 创建 Solution

如果读者是项目的技术负责人，开始编码、设计具体的项目前，首先要做的就是建立 Solution（而不是具体的 Project），并从整体上规划开发成果的布局，如图 A1-1 所示。

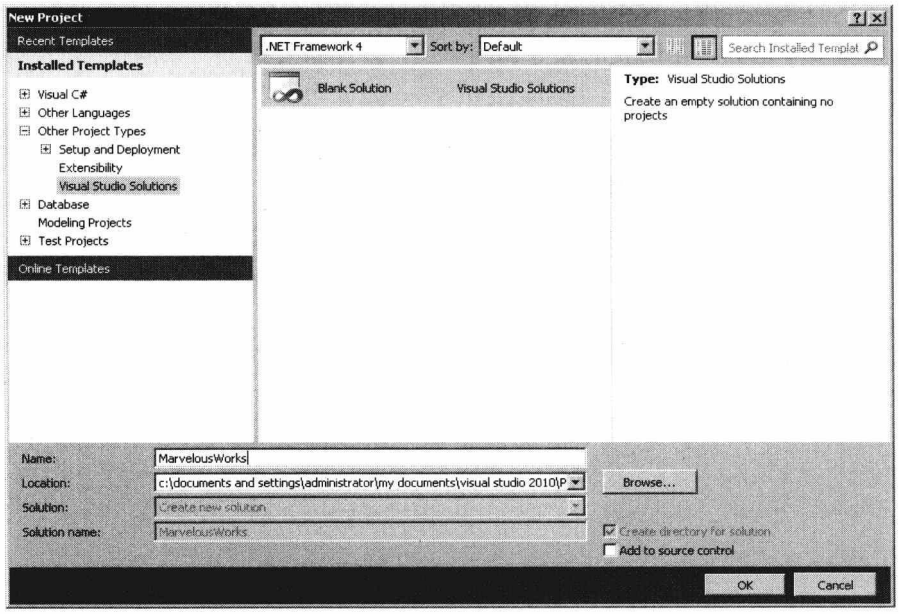

图 A1-1　在 Visual Studio 中建立 Solution

2. 定义 Solution Folder

接着，需要通过 Solution Folder 整体上规划项目过程产品和交付成果的布局，本书采用如图 A1-2 至图 A1-4 所示的规划。

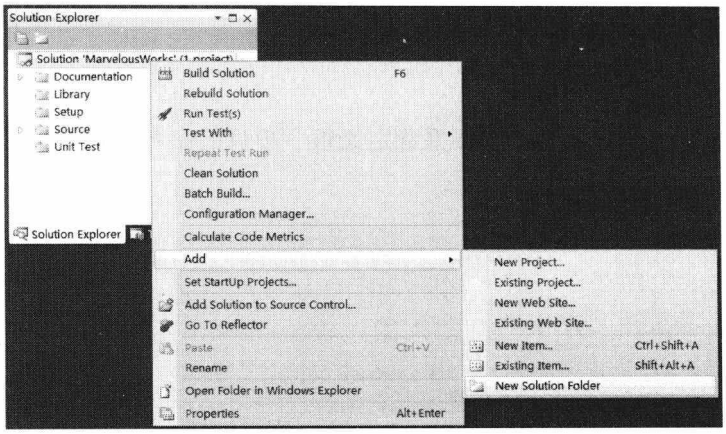

图 A1-2 建立新的 Solution Folder

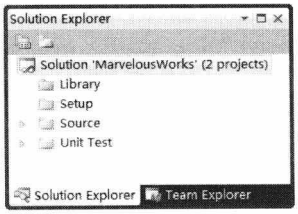

图 A1-3 本书示例的 Solution Folder 的布局

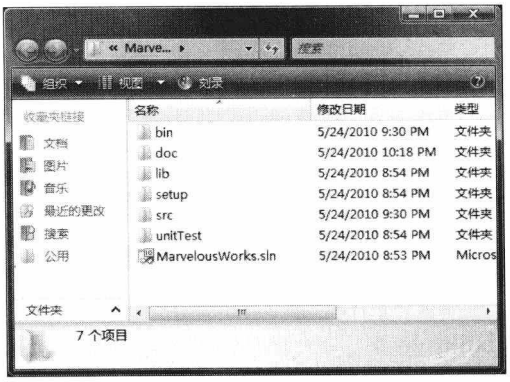

图 A1-4 实际项目文件夹布局

其中：

- Library 对应文件夹 lib，存放需要引用的外部 Assembly。
- Setup 对应文件夹 setup，存放执行环境所需的各类脚本、工具软件等。
- Source 对应文件夹 src，存放源代码。
- Unit Test 对应文件夹 unitTest，存放单元测试程序。

此外，doc 文件夹用于保存文档。

3. 定义项目命名空间和编译目标目录

完成 Solution Folder 的定义后，需要妥善定义每个项目的命名空间，如图 A1-5 所示。

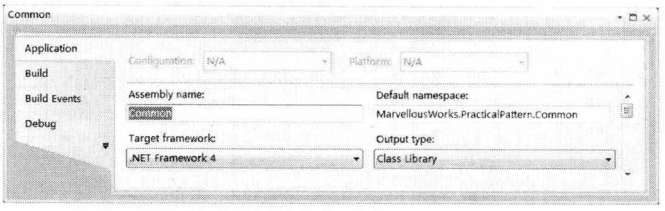

图 A1-5 定义命名空间

此外，为了便于其他程序引用，建议在建立项目时定义编译目标目录。本书中，所有生产代码编译结果统一置于 bin 文件夹下，如图 A1-6 所示。单元测试项目一律采用二进制方式引用 bin 文件下的 Assembly，单元测试项目采用默认的编译目标目录。

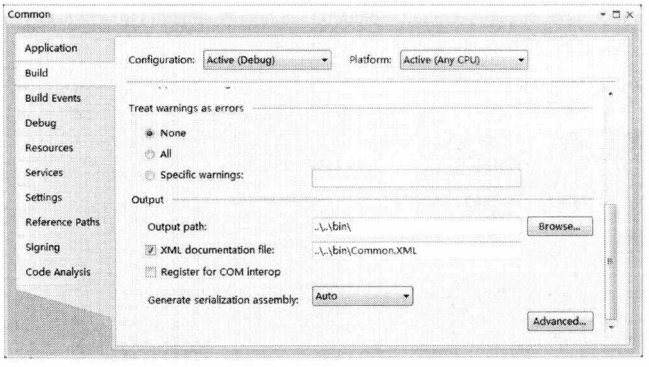

图 A1-6 生产代码项目统一编译到 bin 文件夹下

由于示例 Solution 并没有写 build 脚本，为了确保单元测试程序所需的 Assembly 可以预先编译完成，建议下载代码后第一次执行前，先进行如下编译工作：

（1）选择 Solution 下的 Source 文件夹，单击鼠标右键，在弹出的快捷菜单中选择 Rebuild 命令。

（2）再选择 Unit Test 文件夹，单击鼠标右键，在弹出的快捷菜单中选择 Rebuild 命令。

4．准备单元测试环境

由于本文重在介绍各模式的工程化实现，建议读者采用单元测试跟踪（Trace）的方式学习每个示例，而不是通过建立 Console 甚至是 WinForm、WebForm 项目，因为后者往往需要浪费不少时间去做与模式实现和使用无关的事情。

此外，为了准备一个最轻巧、整洁的单元测试环境，本书单元测试项目并没有通过 Visual Studio 的项目模板创建，而是采用 Class Library 项目引用必要的 Assembly 手工搭建，单元测试通过 TestDriven.NET 调试（该工具可以从 http://www.testdriven.net/下载）。

下面介绍单元测试的准备过程。

5．建立 Class Library 项目

C#

```
using System;
usingSystem.Collections.Generic;
usingSystem.Linq;
namespaceMarvellousWorks.PracticalPattern.Common
{
publicstaticclassEnumerableHelper
    {
///<summary>
/// Execute Action for each element in <see cref="IEnumerable<T>"/>
///</summary>
publicstaticvoidForEach<T>(thisIEnumerable<T> sequence, Action<T> action)
        {
if (sequence == null)
thrownewArgumentNullException("sequence");
if (action == null)
thrownewArgumentNullException("action");
foreach (var item in sequence)
                action(item);
        }
    }
}
```

6．手工建立 Unit Test 项目

通过 Class Library 项目手工建立单元测试环境，如图 A1-7 所示。

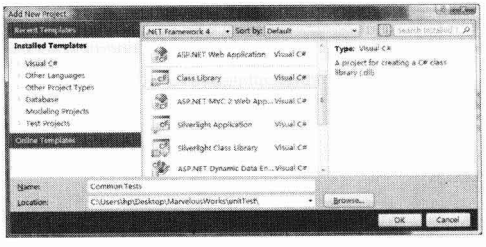

图 A1-7　通过 Class Library 项目手工建立单元测试环境

同前，建立单元测试项目后首先要修改其命名空间，确保测试程序与测试目标程序对应关系明确。此外，为了加载单元测试类型，需要引用额外的 Assembly，如图 A1-8 所示。

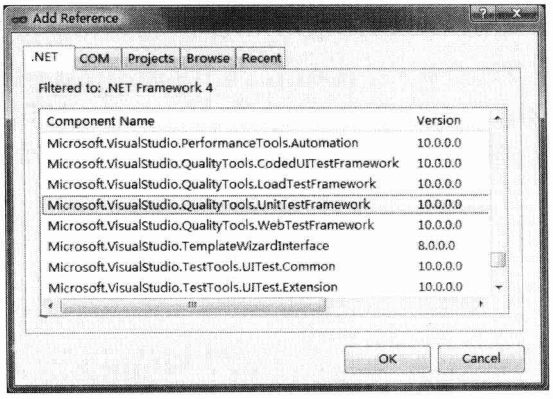

图 A1-8 引用微软的 Unit Test 框架

这样，一个普通的类库项目就可以通过 TestDriven.NET 跟踪和执行单元测试了。

7．编写测试用例

接着，从 bin 文件夹下引用目标 Assembly 并编写单元测试用例。

对于采用测试驱动开发的项目，其单元测试在目标类库完成前就可以完成很多内容，这里为了示例在目标类库完成后才开始编写单元测试用例代码。

C# Unit Test

```
using System;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Diagnostics;
usingMicrosoft.VisualStudio.TestTools.UnitTesting;
usingMarvellousWorks.PracticalPattern.Common;
namespaceCommon.Tests
{
    [TestClass]
publicclassEnumerableHelperFixture
    {
        [TestMethod]
publicvoidTestForEach()
        {
string[] data = { "A", "B", "CDE", "FG"};
int[] lens = newint[data.Length];
int i = 0;
EnumerableHelper.ForEach<string>(data,
delegate(string item)
                {
                    lens[i++] = item.Length;
Trace.WriteLine(lens[i - 1]);
```

```
            });
Assert.AreEqual<int>(string.Join("", data).Length, lens.Sum());
        }
    }
}
```

通过 TestDriven.NET 执行单元测试，如图 A1-9 所示。

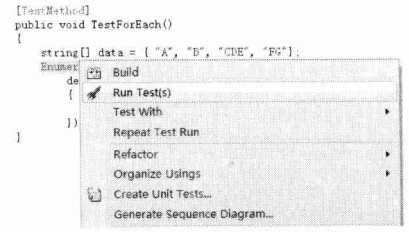

图 A1-9 通过 TestDriven.NET 执行单元测试

测试结果如图 A1-10 所示。

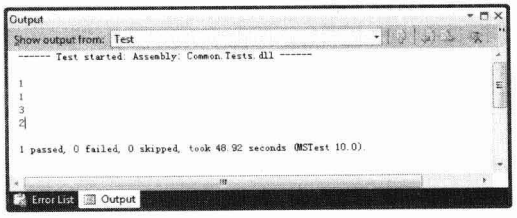

图 A1-10 Output 窗口显示的单元测试追踪信息

类似地，如果需要调试，则可以采用 Debugger 进行单元测试，如图 A1-11 所示。

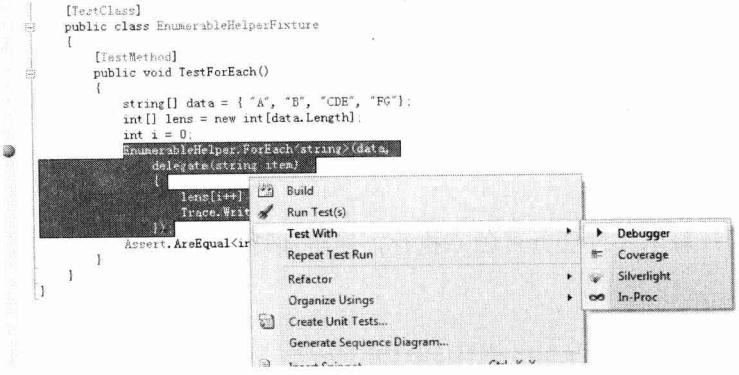

图 A1-11 对单元测试程序进行跟踪调试

8. 提供并发触发机制

尽管绝大部分单元测试可以在单线程环境下完成，但在项目中我们经常需要验

证一些对象在多线程环境下的表现。虽然启动并行处理的方法很多，但本书采用一个简单的方法，代码如下：

C#

```
publicstaticvoid Parallel(paramsThreadStart[] actions)
{
     Thread[] threads = actions.Select(a =>newThread(a)).ToArray();
  threads.ForEach(t =>t.Start());
     threads.ForEach(t =>t.Join());
}
```

下面是对这个测试支持方法自身的测试。

C# Unit Test

```
[TestClass]
publicclassTestHarnessFixture
{
IList<string> recorder = newList<string>();

    [TestMethod]
publicvoidTestParallelExecution()
    {
var start = DateTime.Now;
TestHarness.Parallel
          (
              ()=> { Thread.Sleep(2000); recorder.Add("A");},
              () => { Thread.Sleep(3000); recorder.Add("B"); },
              () => { Thread.Sleep(1500); recorder.Add("C"); }
          );
var end = DateTime.Now;

// max execution time span
Assert.AreEqual<int>(3, (end - start).Seconds);

// execution sequence
Assert.AreEqual<string>("C", recorder[0]);
Assert.AreEqual<string>("A", recorder[1]);
Assert.AreEqual<string>("B", recorder[2]);
    }
}
```

测试结果表明：每个并行任务可独立执行，但整体耗时是由其中执行最久的任务决定的。