TURING 图灵程序设计丛书

Addison Wesley

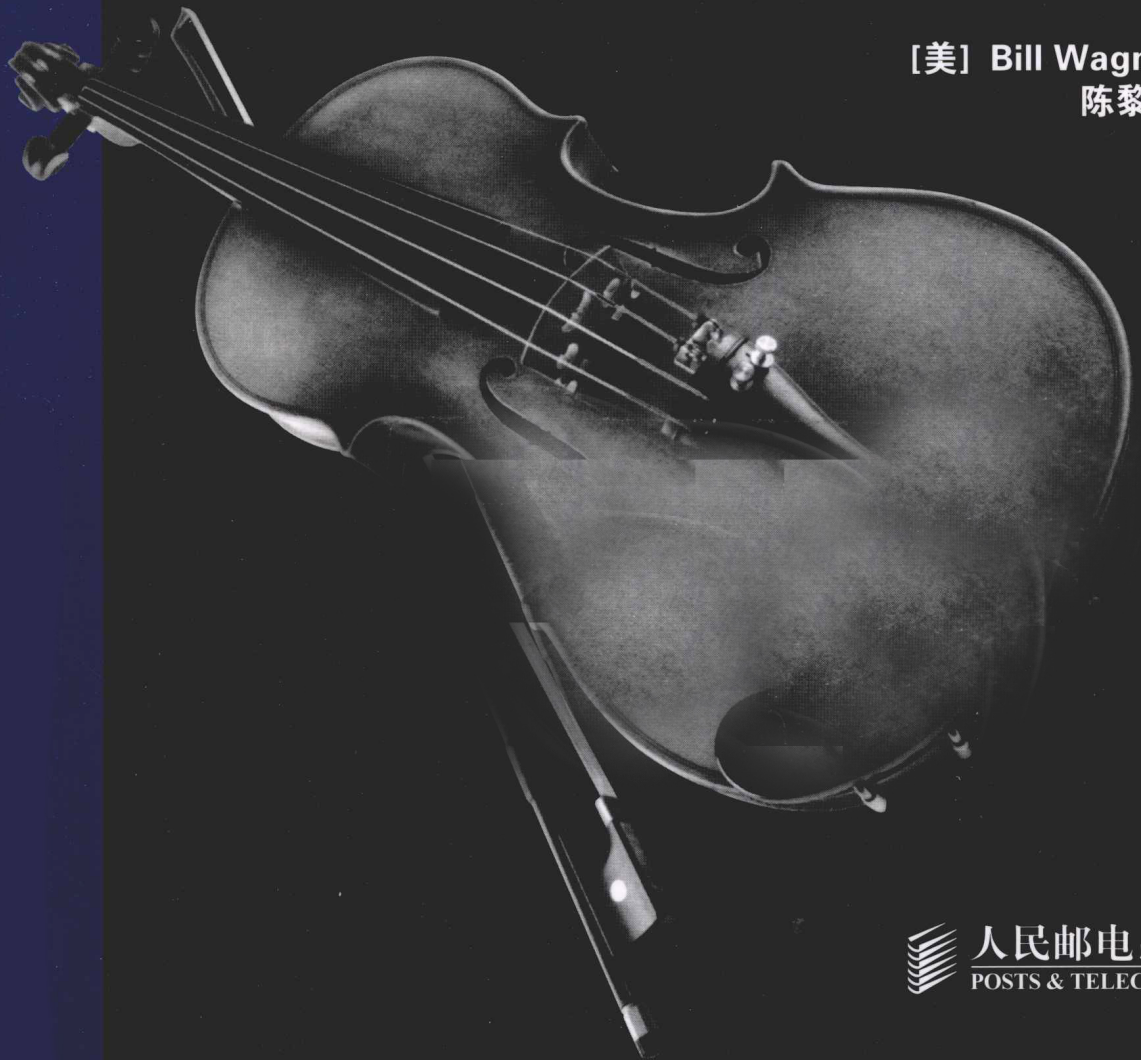
Effective C#

50 Specific Ways to Improve Your C# Second Edition

C#高效编程

改进C#代码的50个行之有效的办法

（第2版）



[美] Bill Wagner 著
陈黎夫 译

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Effective C#
50 Specific Ways to Improve Your C# Second Edition

C#高效编程
改进C#代码的50个行之有效的办法
（第2版）

[美] Bill Wagner 著
陈黎夫 译

人民邮电出版社
北京

图书在版编目（CIP）数据

C#高效编程 ：改进C#代码的50个行之有效的办法 ：第2版 /（美）瓦格纳（Wagner,B.）著 ；陈黎夫译. -- 北京 ：人民邮电出版社，2010.12
（图灵程序设计丛书）
书名原文：Effective C# ：50 Specific Ways to Improve Your C#
ISBN 978-7-115-24041-5

Ⅰ. ①C… Ⅱ. ①瓦… ②陈… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2010)第209314号

内 容 提 要

本书围绕C#语言元素、.NET资源管理、使用C#表达设计、创建二进制组件和使用框架等主题，针对50个常见问题给出了详实的解决方案，并就提升编程效率给出了合理建议。书中内容共分6章，分别讲述了实际编程中不可或缺的习惯用法，如何更好地配合开发环境以期在优化之前解决问题，如何用C#语言良好地表达设计意图，BCL、Parallel Task Library的最常见用法和使用框架过程中常用的技巧，如何在C#中使用动态特性，以及一些对创建强壮、易于维护的程序来说非常重要的难以归类的主题。

本书适合所有C#程序员，也可供高等院校相关专业师生参考。

图灵程序设计丛书

C#高效编程：改进C#代码的50个行之有效的办法（第2版）

◆ 著 [美] Bill Wagner
译 陈黎夫
责任编辑 朱 巍
执行编辑 毛倩倩

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
中国铁道出版社印刷厂印刷

◆ 开本：800×1000 1/16
印张：17.5
字数：370千字 第1版
印数：1－3 000册 2010年12月北京第1次印刷

著作权合同登记号 图字：01-2010-4022号

ISBN 978-7-115-24041-5

定价：49.00元

读者服务热线：(010)51095186 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

引　言

与2004年本书第一版出版时相比，2010年的C#社区已经有了很大的变化。越来越多的开发人员开始使用C#，很多人已经将C#作为他们的首选开发语言。这些人并不是半路出家，也没有从其他语言中带来什么不好的习惯。C#社区成员的背景也日益多样化，有刚刚毕业的学生，也有有着几十年经验的专家。

在过去的5年中，C#语言本身也在不断进步。我们现在已经用惯了的泛型、lambda表达式、LINQ等很多特性，在本书第一版时都还不存在。C# 4.0更是添加了一批新功能。不过，虽然C#语言发展的脚步从未停歇，但很多C# 1.*x*版本中的建议仍然没有过时。事后看来，C#的变化显得非常自然，顺理成章地逐步扩展了C# 1.0的功能。新版本的语言提供了新的解决问题方式，但却不会影响从前的惯用做法。

在组织第二版的时候，我同时考虑了C#语言的变化和C#社区的变化。本书并不会平淡地介绍C#的变化过程，而是会告诉你如何使用当前版本的C#语言。第二版中删除掉的条目都是与当前版本C#不相关的主题。添加的新条目涉及了语言和框架的新功能，还包含社区在编写C#软件过程中总结出的经验教训。总而言之，这本书中的一系列建议将帮你更有效、更专业地使用C# 4.0。

本书覆盖到了C# 4.0，但却不是简单的语言新特性介绍。作为Effective Software Development系列丛书[①]之一，本书的主要目的是介绍如何用这些特性帮你解决每天都要遇到的实际问题。书中的很多条目在C# 3.0或更早一些的版本中依旧适用。

① 此套丛书旨在解决程序员日常工作中遇到的实际问题。除了本书，*Effective Perl*中文版也即将由人民邮电出版社出版。——编者注

读者对象

本书是为那些以C#为主要开发语言的专业开发者编写的。本书假设你已经熟悉了C#的语法和语言特性。第二版还假设你理解了C# 4.0中新添加的语法，以及早期版本中的语法。本书并没有包含语言特性的入门介绍，而是着重于阐述如何将当前版本C#中的各个特性集成到你的每日开发工作中。

除了语言特性之外，本书还假设读者对CLR（Common Language Runtime，公共语言运行时）环境和JIT（Just-In-Time ）编译器有一定的了解。

内容介绍

语言构造几乎是每个C#程序中都不可避免的。第1章就介绍了这些语言习惯。你在日常工作中经常会用到这些习惯，估计都听腻了。在创建每个类型、实现每个算法时，这些内容都是必不可少的。

在托管环境中开发并不代表环境应该负责所有的事情。你仍需要配合环境，创建出合乎性能需求的程序。这并不仅仅涉及性能测试和性能调优。第2章介绍的条目能帮你更好地和环境配合，以期在开始优化之前就解决问题。

很多时候，我们的代码更需要让人理解，而不是仅用来满足编译器的要求。编译器所关心的只是程序是否合法，但项目的协作者也需要理解代码表示的意图。第3章就介绍了如何用C#语言良好地表达出设计意图。解决一个问题总是有多种方法，而第3章给出的建议将帮你选择一个最易于表达设计意图的做法。

C#并不是一个庞大的语言，但其背后却有着丰富的框架库支持。第4章介绍了将用在核心算法中的.NET BCL（Base Class Library，基础类库），还介绍了使用框架过程中常用的一些技巧。多核处理器是日后发展的趋势，Parallel Task Library让.NET平台下的多线程程序开发迈出了坚实的一步。这一章介绍了Parallel Task Library的最常见用法。

第5章介绍了如何在C#中使用动态特性。C#是一种强类型的静态语言。不过，越来越多的程序都会同时使用动态类型和静态类型。C#在提供了动态编程的种种功能的同时，也没有丢失静态类型所带来的优势。第5章将介绍如何使用动态功能，以及如何避免动态类型泄漏

到整个程序之外。

第6章介绍了一些难以归类的主题。这些主题在对创建强壮、易于维护和扩展的程序时会经常用到。

代码规范

开发工具中的代码都有着色，因此在书中也不该只看到黑白的样子。虽然无法实现如同在流行IDE中阅读代码那样的体验，但是我仍旧尽力让你能更容易理解书中的代码。本书中，代码中着重强调的部分将加底纹标出。

在书中提供代码仍需要一些让步，主要是篇幅和易读性方面的问题。书中尽力将代码压缩至足够说明问题的一小段，这通常也意味着类或者方法的其他部分将被省略，包括一些错误恢复代码。公有方法应该验证参数以及其他输入，但这里限于篇幅则通常会省略。类似的省略还包含方法调用时的验证以及复杂算法中常用的try/finally子句等。

这里也假设读者能够找到例子中用到的常用命名空间。或者，你也可以假设每一段代码都引入了如下的using语句：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Dynamic;
using System.Threading;
```

此外，本书还使用了#region/#endregion表示接口的实现。虽然这并不是必需的，且有些人并不喜欢这类风格，但这种做法的确能使读者在静态的文字中更明显地找出接口的实现。所有其他的做法都不够标准，且需要更多的篇幅。

提供反馈

虽然我和审校者已经尽全力地保证本书的质量，但文字和代码中仍有可能包含错误。若你确信找到了错误，请给我发邮件（bill.wagner@srtsolutions.com）。本书的勘误将发布在http://srtsolutions.com/blogs/effectivecsharp之上。本书以及《More Effective C#中文版——改善

C#程序的50个具体办法》中的很多条目都是与其他C#开发者沟通后做的结论。若你对这些条目存在疑问或建议，也请联系我。我的博客http://srtsolutions.com/blogs/billwagner中还有更多相关主题的讨论。

致谢

很多人为本书做出了贡献，我向他们致谢。我为身处这样一个充满惊喜的C#社区而感到荣幸。C# Insiders邮件列表中的每个人（不管是不是在微软公司）的讨论和想法都让本书变得更加完善。

我必须特别感谢C#社区中几位对本书给出了最直接帮助的人，他们将想法变成了一条条成文的建议。正是有了与Charlie Calvert、Eric DeCarufel、Justin Etheredge、Marc Gravell、Mike Gold和Doug Holland的讨论才有了这一版中的很多新想法。

关于Parallel Task Library及其在C#中的运用，我与Stephen Toub和Michael Wood通过e-mail进行了讨论并获益匪浅。

本书的技术审校团队非常出色。Jason Bock、Claudio Lassala和Tomas Petricek完整审阅了本书的文字和示例程序，保证了质量。他们审校得非常认真、全面，没有人能比他们做得更好了。此外，他们提出的建议还帮我用更好的方式表述了某些主题。

Addison-Wesley公司的团队是出版行业的梦之队。Joan Murray是一位出色的编辑和项目经理，保证了整个项目的顺利进行。Joan Murray和我的工作都离不开Olivia Basegio。正是有了她的帮助，本书无论是封面设计还是内文质量都得到了保证。Curt Johnson和Brandon Prebynski对技术内容的市场运作也非常成功。本书的所有载体形式都饱含Curt和Brandon的辛勤劳动。Geneil Breeze通读了本书，帮助在一些地方做了进一步的解释，并澄清了一些地方的措辞用语。

能够为Scott Meyer编写这个系列图书之一是我的荣幸。Scott Meyer审校了每一篇文章，并给出了评价和建议。Scott Meyer思维缜密，虽然并不熟悉C#，不过以其在软件行业的经验，指出了本书中一些没有表述清楚的条目，以及某些不合适的条目。他的反馈从来都是无价之宝，对本书的反馈亦不例外。

SRT Solutions公司的很多咨询师也给出了宝贵的建议。无论是经验丰富的高手，还是初出茅庐的年轻人，SRT Solutions眼光卓越的人们从不吝惜表达自己的意见。与Ben Barefield、Dennis Burton、Marina Fedner、Alex Gheith、Darrell Hawley、Chris Marinos、Dennis Matveyev、Anne Marsan、Dianne Marsh、Charlie Sears、Patrick Steele、Mike Woelmer和Jay Wren的无数次谈话给我带来了很多的想法和例子。后续的一些讨论还帮我修订了一些文字，也帮我在各个条目中做出了取舍。

我的家人也一如既往地支持我。他们给了我很多自由时间，让我能专心完成本书。我的孩子Lara、Sarah和Scott容忍我躲在家里写书而不参加他们的活动。我的妻子Marlene忍受我长时间外出撰稿和寻找写作素材。没有了他们的支持，我不可能完成这本书，更不可能让本书达到如此令人满意的程度。

目　录

第1章 C#语言习惯

为什么程序已经可以正常工作了，还要继续修改呢？答案就是我们还能让程序变得更好。如果你总是墨守成规，那么将永远体会不到新技术带来的优势。对于C#这种和我们已经熟悉的语言（如C++或Java等）有很多类似之处的新语言，情况更是如此。C#也是一种用大括号组织代码块的语言，因此人们很容易重拾他们熟悉的习惯。但这会阻碍你学到C#的种种精妙之处。自2001年第一个正式版本以来，C#一直在不断改进。其第一个版本与C++和Java尚有许多共同之处，但如今已渐行渐远。若你是从其他语言转到C#。那么应掌握必要的C#语言习惯，让其更好地配合你的工作。本章将讨论那些在C#中应该改变的旧习惯，以及与其对应的推荐的新做法。

条目1 使用属性而不是可访问的数据成员

属性一直是C#语言中的一等公民。自1.0版本以来，C#对属性进行了一系列的增强，让其表达能力不断提高。你甚至可以为setter和getter指定不同的访问权限。隐式属性也极大降低了声明属性时的工作量，不会比声明数据成员麻烦多少。若你仍然在类型中声明公有成员，或是仍在手工编写set或get之类的方法，那么快停下来吧。属性允许将数据成员作为公共接口的一部分暴露出去，同时仍旧提供面向对象环境下所需要的封装。属性这个语言元素可以让你像访问数据成员一样使用，但其底层依旧使用方法实现。

类型的某些成员确实非常适合作为数据，例如某个客户的名称，某个点的x、y坐标或上

一年度的收入等。而属性则让你可以创建出类似于数据访问，但实际上却是方法调用的接口，自然也可以享受到方法调用的所有好处。客户代码访问属性时，就像是在访问公有的字段。不过其底层使用方法实现，其中可以自由定义属性访问器的行为。

.NET Framework假设你会对公有数据成员使用属性。实际上，.NET Framework中的数据绑定类仅支持属性，而不支持公有数据成员。对于所有的数据绑定类库均是如此，包括WPF、Windows Forms和Silverlight。数据绑定会将某个对象的一个属性和某个用户界面控件相互关联起来。数据绑定机制将使用反射来找到类型中的特定属性：

```
textBoxCity.DataBindings.Add("Text",
    address, "City");
```

这段代码将textBoxCity控件的Text属性绑定到了address对象的City属性上。公有的数据成员并不推荐使用，因此Framework Class Library设计器也不支持其实现绑定。这样的设计也保证了你必须选择合适的面向对象技术。

确实，数据绑定只是用在用户界面逻辑中会使用到的类中。但这并不意味着属性仅应该用在UI逻辑中，其他类和结构中也应使用属性。在日后产生新的需求或行为时，属性更易于修改。例如，你会很快有这样的想法，客户对象不应该有空白的名称。若你使用了公有属性来封装Name，那么只要修改一处即可。

```
public class Customer
{
    private string name;
    public string Name
    {
        get { return name; }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                "Name");
            name = value;
        }
        // More Elided.
    }
}
```

若是使用了公有的数据成员，那么就需要查找每一处设置客户名称的代码并逐一修复。这将花费大量的时间。

因为属性是使用方法来实现的，所以添加多线程支持也非常简单。很容易即可在属性的get和set访问器中作出如下的修改，从而支持对数据的同步访问：

```
public class Customer
{
    private object syncHandle = new object();

    private string name;
    public string Name
    {
        get
        {
            lock (syncHandle)
                return name;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                  "Name");
            lock (syncHandle)
                name = value;
        }
    }
    // More Elided.
}
```

属性可以拥有方法的所有语言特性。例如，属性可以为虚的（virtual）：

```
public class Customer
{
    public virtual string Name
    {
        get;
        set;
    }
}
```

注意，上述例子中使用了C# 3.0中的隐式属性语法。使用属性来封装私有字段是一个常用的模式。通常而言，我们并不需要验证属性的getter或setter逻辑。因为语言本身提供了简

化的隐式属性语法，力求尽量降低开发人员的输入工作，即将一个简单的字段暴露成属性。编译器将为你创建一个私有的成员字段，并自动生成最简单的get和set访问器的逻辑。

你还可以将属性声明为抽象的（abstract），以类似隐式属性语法的形式将其定义在接口中。下面的例子就将属性定义在了一个泛型接口中。需要注意的是，虽然其语法和隐式属性完全相同，但是编译器却不会自动地生成任何实现。接口只是定义了一个契约，强制所有实现了该接口的类型都必须满足。

```
public interface INameValuePair<T>
{
    string Name
    {
        get;
    }

    T Value
    {
        get;
        set;
    }
}
```

属性是一种全功能的、第一等的语言元素，能够以方法调用的形式访问或修改内部数据。成员函数中可以实现的功能均可在属性中实现。

属性的访问器将作为两个独立的方法编译到你的类型中。在C#中，你可以为get和set访问器制定不同的访问权限。这样即可更精妙地控制作为属性暴露出来的数据成员的可见性：

```
public class Customer
{
    public virtual string Name
    {
        get;
        protected set;
    }
    // remaining implementation omitted
}
```

上述属性语法的表达含义远远超出了简单数据字段的范畴。若类型需要包含并暴露出可索引的项目，那么可以使用索引器（即支持参数的属性）。若想返回序列中的项，创建一个属性会是个不错的做法：

```
public int this[int index]
{
    get { return theValues[index]; }
    set { theValues[index] = value; }
}

// Accessing an indexer:
int val = someObject[i];
```

索引器和单一条目属性有着同样的语言支持：它们都是作为方法实现的，因此可以在索引器内部实现任意的验证或计算逻辑。索引器也可为虚的或抽象的，可声明在接口中，可以为只读或读写。一维且使用数字作为参数的索引器也可参与数据绑定。使用非整数参数的索引器可用来定义图和字典：

```
public Address this[string name]
{
    get { return adressValues[name]; }
    set { adressValues[name] = value; }
}
```

C#中支持多维数组，类似地，我们也可以创建多维索引器，每一个维度上可以使用同样或不同的类型：

```
public int this[int x, int y]
{
    get { return ComputeValue(x, y); }
}

public int this[int x, string name]
{
    get { return ComputeValue(x, name); }
}
```

需要注意的是，所有的索引器都使用this关键字声明。C#不支持为索引器命名。因此，类型中每个不同的索引器都必须有不同的参数列表，以免混淆。几乎属性上的所有特性都能应用到索引器上。索引器也可为虚的或抽象的，可以对setter和getter给出不同的访问限制，不过却不能像属性那样创建隐式索引器。

属性的功能很强大，是个不错的改进。但你是不是还在想能不能先用数据成员来实现，而在稍后需要其他各种功能的时候再改成属性呢？这看似是个不错的策略，不过实际上却行不通。考虑如下这个类的定义：

```
// using public data members, bad practice:
public class Customer
{
    public string Name;

    // remaining implementation omitted
}
```

这个类描述一个客户（Customer），包含了一个名称（Name）。你可以使用熟悉的成员表示方式获取或设置该名称：

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

看似简单直观，你也会认为若是日后将Name改成属性，那么代码也可以无需修改保持正常。但这个答案并不是完全正确的。属性仅仅是访问时类似于数据成员，这是语法所实现的目的。不过属性并不是数据，属性的访问和数据的访问将会生成不同的MSIL（Microsoft Intermediate Language，微软中间语言）指令。

虽然属性和数据成员在源代码层次上是兼容的，不过在二进制层面上却大相径庭。这也就意味着，若将某个公有的数据成员改成了与之等同的共有属性，那么就必须重新编译所有用到该公有数据成员的代码。C#把二进制程序集作为一等公民看待。该语言本身的一个目标就是支持发布某个单一程序集时，不需要更新整个的应用程序。而这个将数据成员改为属性的简单操作却破坏掉了二进制兼容型，也就会让更新单一程序集变得非常困难。

若是查看属性生成的IL，那么你或许会想比较一下属性和数据成员的性能。属性当然不会比数据成员访问快，不过也不会比其慢多少。JIT编译器将内联一些方法调用，包括属性访问器。当JIT编译器内联了属性访问器时，数据成员和属性的访问效率即可持平。即使某个属性访问器没有被内联，其性能差距也实在是微乎其微，仅仅一次函数调用之别而已。只有在某些极端情况下，二者的差距才会有所影响。

在调用方来看，属性虽然是方法，但它和数据却有着类似的概念。这会使你的调用者对属性有着一些潜意识的认识。例如，调用者会把属性访问当成是数据的访问。不管怎样，二者看上去很像。属性访问器应该满足这些潜意识的预期。get访问器不应该有可被观察到的副作用。set访问器会修改状态，用户应该可以看到调用后带来的改变。

调用者也会对属性访问器的性能有着一定的预期。属性的访问就像是访问一个数据字