TURING 图灵程序设计丛书 微软技术系列

Addison Wesley

More Effective C#中文版

改善C#程序的50个具体办法

[美] Bill Wagner 著
陈黎夫 译

MVP Microsoft Most Valuable Professional

人民邮电出版社
POSTS & TELECOM PRESS

“身为C#设计组的成员，我很少能够从C#书中学到什么新东西，本书则是个例外，它很好地将特定的代码和深入的分析结合了起来。……这些富有洞察力的、充满远见的内容会给你日后学习C#很大的启发和帮助。”

——Mads Torgersen，微软公司Visual C#项目经理

“Bill Wagner为C#开发人员撰写了一部精彩绝伦的图书，其中介绍了大量C#最佳实践。……若想成为C#开发的顶级高手，那么没有什么资料比Bill Wagner的这本书更好了。Bill非常智慧、深刻，富有经验和技巧。若能将这本书中给出的建议应用到你的代码中，定会大大提高你的工作质量。”

——Charlie Calvert，微软公司Visual C#社区项目经理

More Effective C#中文版

改善C#程序的50个具体办法

C#语言已经成为目前功能最强大的通用语言之一，近几年的几次升级更是令C#编程世界发生了极大的改变。本书是*Effective C#*的续作，秉承了*Effective*经典系列的卓越风格，用真实的代码示例，通过清晰、贴近实际和简明的阐述，以条目形式为广大程序员提供了凝聚业界经验结晶的专家建议。

本书中，著名.NET专家Bill Wagner围绕C# 2.0和3.0中的新特性给出了实用的建议，涉及泛型、多线程编程、设计实践、C# 3.0语言增强、LINQ、可空类型等主题，讲述了如何在开发中使用这些新语言特性，如何避免误用造成的影响。书中每个条目自成一体，针对使用C#时的某个特定问题，帮助你以最佳的方式切换至C# 3.0。通过阅读本书，读者完全可以举一反三，将其中许多建议应用到自己的日常编程工作中。

- C#语言顶级高手的秘籍
- 业界资深专家智慧和经验的结晶
- 理论和实践的完美结合

Addison Wesley
www.informit.com

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-21570-3

ISBN 978-7-115-21570-3
定价：49.00元

TURING 图灵程序设计丛书 微软技术系列

More Effective C#中文版

改善C#程序的50个具体办法

[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

More Effective C# 中文版：改善 C# 程序的 50 个具体办法 /（美）瓦格纳（Wagner, B.）著；陈黎夫译．—北京：人民邮电出版社，2010.1
（图灵程序设计丛书）
书名原文：More Effective C#: 50 Specific Ways to Improve Your C#
ISBN 978-7-115-21570-3

Ⅰ.①M… 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（2009）第178752号

内 容 提 要

本书延续了 *Effective* 系列图书的风格，针对 C# 2.0 和 C# 3.0 中添加的新特性给出了实用的建议。书中的 50 个条目自成一体且又丝丝相扣，这些条目按照泛型、多线程开发、C# 设计模式、C# 3.0 语言增强、LINQ 以及杂项等主题分为 6 类，将特定的代码和深入的分析有机地结合了起来，能够帮助你以最佳的方式从 C# 1.x 切换至 C# 3.0。当你通读全书之后，会发现不只得到了一条条独立的建议，还学到了如何以优雅的方式用 C# 进行程序设计。

本书适合具有 C# 编程经验的 .NET 开发人员阅读。

图灵程序设计丛书

More Effective C#中文版：改善C#程序的50个具体办法

◆ 著 [美] Bill Wagner
译 陈黎夫
责任编辑 傅志红
执行编辑 陈兴璐

◆ 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
北京隆昌伟业印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：19.25
字数：315千字 2010年1月第1版
印数：1－3 000册 2010年1月北京第1次印刷

著作权合同登记号 图字：01-2009-5718号

ISBN 978-7-115-21570-3

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

本书赞誉

“本书就像一盏明灯，照亮了C# 3.0中很多不为人知的角落。它不仅介绍了如何做，还解释了这样做的原因，让读者学习到很多经过实践检验的语言新特性用法，包括LINQ、泛型以及多线程等。若你确实需要使用C#语言开发程序，那么本书是必不可少的。”

——Bill Craun，Ambassador Solutions公司首席咨询师

“本书创造了一个让你能够和Bill Wagner并肩思考、工作的机会。Bill在本书中充分展示了他在C#上的造诣，给出了很多编程方面的有效建议，值得每个Visual C#开发者去学习。本书并没有停留在泛泛描述C#语法上，而是真正教会你使用C#语言。”

——Peter Ritchie，微软公司MVP: Visual C#

“本书是Bill Wagner前一本书的很好续作。其中对C# 3.0和LINQ的介绍非常及时！”

——Tomas Restrepo，微软公司MVP: Visual C++, .NET和Biztalk Server

“作为C#设计组的成员，很少有书能够让我从中学到什么新东西。本书则是个例外。它很好地将特定的代码和深入的分析结合了起来。本书提供了一系列非常有用的建议。当你通读全书之后，会发现不只得到了一条条独立的建议，还学到了如何能够以优雅的方式用C#进行程序设计。虽然你可以根据需要挑选

某个条目阅读，但我仍强烈建议你通读全书——至少不要跳过每一章前面的介绍部分。这一富有洞察力的、充满远见的内容会对你日后的C#学习给予很大的启发和帮助。”

——Mads Torgersen，微软公司Visual C#项目经理

“Bill Wagner为C#开发人员撰写了一本精彩绝伦的图书，其中介绍了大量C#最佳实践。本书再次确立了他在C#社区中的重要地位。我们大都知道如何使用C#，同时也期待有人能给出提高的建议，让我们更上一层楼。若想成为C#开发的顶级高手，那么没有什么资料要比Bill Wagner的这本书更好了。Bill非常智慧、深刻，富有经验和技巧。若能将这本书中给出的建议应用到你的代码中，定会大大提高你的工作质量。”

——Charlie Calvert，微软公司Visual C#社区项目经理

前　言

自从Anders Hejlsberg在2005年专业开发者大会上第一次演示LINQ（Language-Integrated Query，语言集成查询）以来，C#编程世界就被彻底地改变了。LINQ的出现为C#语言带来了几个令人着迷的新特性：扩展方法、局部变量类型推断、lambda表达式、匿名类型、对象初始化器以及集合初始化器。C# 2.0也为LINQ的出现打下了坚实的基础，添加了包括泛型、迭代器、静态类、可空类型、属性访问器权限以及匿名委托等新功能。但即使在非LINQ的使用环境中，这些语言特性也有大显身手之处——毕竟还有很多非数据访问的编程任务。

本书针对C# 2.0和C# 3.0中添加的新特性给出了实用的建议，也包含了在我的上本图书*Effective C#: 50 Specific Ways to Improve Your C#*（Addison-Wesley，2004）[①]中没有提到的高级特性。本书中的条目主要针对那些正在使用C# 3.0编写程序的开发人员。书中着重介绍了泛型技术，这是C# 2.0和C# 3.0中众多新特性的基石。本书并没有将条目按照语言特性组织起来，而是根据新特性最善于解决的编程问题来编排条目的。

与Effective Software Development丛书中的其他图书一样，本书中的每个条目的建议都自成一体，针对使用C#时的某个特定问题。这些条目能够帮助你以最佳的方式从C# 1.x切换至C# 3.0。

泛型是C# 3.0中所有新特性的基础。虽然只有第1章专门介绍了泛型，但你会发现泛型技术也是几乎每个条目中的不可分割的一部分。在阅读完本书之后，你定会熟悉并喜欢上泛型以及元编程（metaprogramming）概念。

① 中译本《Effective C#中文版》，李建忠译，人民邮电出版社2007年出版。——编者注

当然，本书中的很大一部分篇幅都用来讨论了如何使用C# 3.0以及LINQ查询语法。不过不管你是否将其用在查询数据源上，C# 3.0所添加的众多语言新特性均非常有用。语言上的改变非常巨大，LINQ又是引起改变的主要原因，它们都需要专门的章进行介绍。LINQ和C# 3.0将深刻影响你编写C#代码的方式，而本书则会让这个过渡更加平稳简单。

读者对象

本书是为那些使用C#进行软件设计的专业开发人员所编写的。本书假定你已对C# 2.0和C# 3.0有了一定的了解。Scott Meyers告诉我说，*Effective*系列图书应该作为开发人员针对某一主题学习的进阶参考资料。因此，本书并没有泛泛介绍任何有关语言的新特性，而是着重阐述如何将这些新特性应用到正在开发的软件中。你将会学到，何时该在开发中使用这些新语言特性，以及如何避免误用所造成的问题。

除了对C#语言新特性有一定了解之外，你还应该对组成.NET Framework的主要组件有所了解，包括.NET CLR（Common Language Runtime）、.NET BCL（Base Class Library）以及JIT（Just In Time）编译器等。本书并没有涉及.NET 3.0组件，例如WCF（Windows Communication Foundation）、WPF（Windows Presentation Foundation）以及WF（Windows Workflow Foundation）等。不过其中介绍的各种用法同样适用于上述各个组件以及其他的.NET Framework组件。

内容介绍

泛型是自C# 1.1以来所有C#语言新功能的基础。第1章首先介绍了如何用泛型替代`System. Object`和类型强制转换，随后讨论了一些高级主题，包括约束、泛型的特化、方法约束以及向后兼容性等。其中介绍的几种技术都使用泛型让你更清晰地表达出设计意图。

多核处理器已经普及，同时计算机的核心数量也在不停增加。这也就意味着每个C#开发人员都需要对C#多线程编程有着足够的理解。即便一章的篇幅

不足以让你变成专家，但第2章中的建议仍旧会对你开发多线程应用程序有所帮助。

第3章介绍了如何用C#语言实现常用的设计。其中将介绍用C#语言特性表达意图的最好方法。你将学到如何使用延迟求值，如何创建可组合的接口，以及如何避免由于公共接口中各种语言元素所带来的混乱。

第4章讨论了如何借助C# 3.0的语言增强来解决编程中遇到的困难，包括如何使用扩展方法来分离契约和实现，如何有效地使用C#闭包，以及如何使用匿名类型等内容。

第5章介绍了LINQ及其查询语法，包含了编译器如何将查询关键字映射到方法调用的方法，如何区分委托和表达式树（以及需要时在二者之间进行转换），以及如何在需要单一值（scalar value）时处理查询等。

第6章介绍了如何定义部分类，使用可空类型，以及在使用数组参数时避免协变和逆变问题等内容。

示例代码

本书中给出的示例代码不是完整的程序，而是小块的代码片断，但已足够说明问题。在有些示例中，方法名称就表明了该方法要完成的任务，例如`AllocateExpensiveResource()`。这样你无需阅读整篇的代码即可快速领悟到其表达的含义，从而应用到你的开发中。在省略了方法实现时，方法名称就表明了该方法的用途。

在所有的代码片断中，均假设引入了如下命名空间：

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

若是使用到了其他命名空间中的类型，我将在类型中显式给出命名空间。

在本书的前3章中，我会尽可能地使用C# 2.0和C# 3.0中的新语法，即使这些新语法并不是必需的。在第4章和第5章中，则假设你已经熟悉了3.0的新语法。

建议和反馈

我已经仔细审校过本书，但若是你确信找到了错误，请通过电子邮件联系我：bill.wagner@srtsolutions.com。本书勘误将发布至http://srtsolutions.com/blogs/MoreEffectiveCSharp之上。

致谢

最近，一位同事问起我写完一本书之后的感觉。我说有一种类似发布了一个软件产品一样的满足和轻松。尽管写书工作量很大，但它确实也带来很多的成就感。与成功发布软件产品一样，完成一本书需要很多人的共同努力，这些人理应得到感谢。

在2004年撰写*Effective C#*时，我很荣幸成为Effective Software Development丛书的一名作者。现在这本涉及大量C#语言变化的*More Effective C#*继续成为该丛书中的一员，则更让我受宠若惊。撰写本书的想法源于我在2005年的PDC大会上与Curt Johnson和Joan Murray的一顿晚餐，那时我已被Hejlsberg和其他C#团队成员的演示介绍深深震撼。从那时开始，我就开始关注C#语言的变化，并仔细思考这些变化将为C#开发者带来怎样的影响。

当然，在我有能力和自信给出关于所有这些新特性的建议之前，仍旧需要花费时间来使用这些功能，并与同事、客户以及社区中的其他开发者讨论各种不同的使用方式。在万事俱备之后，我便正式开始撰写本书了。

我非常幸运拥有一支优秀的技术审校队伍。他们向我介绍新的主题，修改现有的建议，并挑出了早期草稿中出现的很多错误。Bill Craun、Wes Dyer、Nick Paldino、Tomas Restrepo和Peter Ritchie都提供了详细的技术反馈意见，让本书更加实用。Pavin Podila还审校了本书的WPF部分，保证了其正确性。

在写作的过程中，我和社区以及C#开发团队的成员讨论了很多想法。安阿伯.NET开发组、大湖区.NET用户组、大兰辛地区用户组、西密歇根.NET用户组和托莱多.NET用户组中的成员都充当了本书各个条目的早期审校者。此外，CodeMash的参会者也帮我决定了各个条目的取舍。特别是Dustin Campbell、Jay Wren和Mike Woelmer，他们和我讨论了很多想法。Mads Torgersen、Charlie Calvert和Eric Lippert也曾帮我澄清了本书中的很多条目。特别值得一提的是，Charlie Calvert帮我更好地将工程师的思维用写作者的表达方法变成文字。若是没有这些讨论，本书将远远无法清晰明了，且可能会错过很多重要的概念。

从头到尾两次经历了Scott Meyers的完整审校流程之后，我可以闭着眼推荐他经手的每一本书。Scott Meyers虽不是C#专家，但他的天分和对图书质量的认真负责让我非常佩服。他对本书的审读意见非常多，逐一解答确认也花费了很长时间，但这保证了书稿的质量。

在本书出版的整个过程中，Joan Murray的作用无可替代。作为编辑，她总能帮我想到所有的一切。在我需要监督的时候给出提醒，为我找到优秀的审校者，并帮助我把本书的想法变为大纲、再到草稿、直到现在你手中的成书。还有Curt Johnson，他们让我与Addison-Wesley出版社的合作非常愉快。

最后一步是与文字编辑的配合。Betsy Hardinger帮助将工程师写出的枯燥文字转变成了生动的书面英语，而并没有损害技术上的准确性。在她编辑之后，本书的语言变得更加地流畅。

当然，撰写图书也要花费大量的时间。在这段时间中，Dianne Marsh（SRT Solutions的另一个所有人）让公司保持正常运转。而最大的牺牲则来自于我的家庭，在撰写本书时，我无暇悉心顾及他们的感受。最真诚的感谢送给Marlene、Lara、Sarah和Scott，感谢你们再次全力支持我的投入。

目　录

第1章

使用泛型 1

毋庸置疑，C# 2.0所添加的泛型（generic）特性极大地影响了开发人员编写代码的方法和方式。很多文章和论文均详细分析过泛型相对于从前版本中C#集合类的优势，这些分析都是正确的。与使用传统的弱类型集合（依赖于System.Object）相比，使用泛型集合能够保证编译期类型安全，并提高应用程序的执行效率。

不过，有些文章和论文可能会让你有这样的想法：泛型仅仅在集合这个上下文中才有用武之地。但事实并非如此。泛型还能够用于很多其他场合中，例如创建接口、事件处理程序以及常用算法等。

也有一些讨论将C#的泛型和C++的模板进行对比，这类讨论通常的结论是某一种要好于另一种。虽然将C#泛型和C++模板比较能够帮助你理解泛型的语法，不过其作用也就仅此而已。有些使用方法在C++模板中显得更加自然，而有些则在C#泛型中略胜一筹。不过，若一味沉浸在探究哪一种“更好”之中，那么最终只会让你迷失于这无谓的争辩中，失去了比较的本意（可以参考稍后将介绍的条目2）。为C#语言添加泛型支持需要修改C#编译器、JIT（Just In Time）编译器以及CLR（Common Language Runtime，公共语言运行时）。其中，C#编译器根据C#代码生成以微软中间语言（Microsoft Intermediate Language，MSIL或IL）表示的泛型类型定义。而JIT编译器则会把泛型类型定义与一系列的类型参数组合起来，从而创建出封闭的泛型类型。CLR将在运行时同时支持上述两种概念。

泛型类型定义有利也有弊。有些时候，将代码转换为泛型写法可以减少程序的大小。而有些时候反而会事与愿违。这取决于程序中使用的类型参数

的个数，以及创建出的封闭泛型类型的个数。

泛型类定义能够完整地编译为MSIL类型。对于任何满足约束的类型参数，泛型类型中包含的代码必须保证完全合法。所有类型参数已经明确给出的泛型类型叫做*封闭泛型类型*（closed generic type），而若是仅给出了部分类型参数，那么这种泛型叫做*开放泛型类型*（open generic type）。

IL中的泛型可以看作是某个实际类型定义的一部分。IL为初始化某个完整的泛型类型实例预留了占位符。JIT编译器将在运行时生成机器代码时补全该封闭泛型类型的定义。这样的处理方法自然带来了一个矛盾：其劣势在于多个封闭泛型类型会增大处理代码的开销，而优势则体现在存储数据的时间/空间开销会减少。

不同的封闭泛型类型可能会导致代码生成不同的最终运行时形式。在创建多个封闭泛型类型时，JIT编译器和CLR均会对过程进行优化，以便降低对内存的压力。IL形式的程序集将被加载至内存中的数据页。只有在JIT编译器将IL转换成机器指令之后，生成的机器码才会被放置于只读的代码页中。

无论是否泛型，每个类型都会执行上述过程。对于非泛型类型，类的IL和其生成的机器码之间是一对一的关系。而泛型的出现则让转换的过程变得略加复杂。在JIT对泛型类进行转换时，JIT编译器将检查当前的类型参数，并根据该信息生成特定的指令。JIT编译器将会对该过程进行一系列的优化，以便让不同类型参数能够使用同样的机器码。首先也是最重要的，JIT编译器将专门为所有引用类型生成泛型类的一个机器码版本。

如下的实例化代码在运行时的机器码均完全相同。

```
List <string> stringList = new List<string>();
List<Stream> OpenFiles = new List<Stream>();
List<MyClassType> anotherList = new List<MyClassType>();
```

C#编译器将在编译期保证类型安全，有了这种保证之后，JIT编译器即可生成更加优化的机器码。

而若是某个封闭泛型类型中使用了至少一个值类型的类型参数，那么JIT编译器将采用不同的策略。在这种情况下，JIT编译器将为不同的类型参数创建不

同版本的机器指令。因此，下面的三种封闭泛型类型所生成的机器码将各不相同。

```
List<double> doubleList = new List<double>();
List<int> markers = new List<int>();
List<MyStruct> values = new List<MyStruct>();
```

虽然看上去比较有意思，不过这和我们开发者又有什么关系呢？答案就是，使用多个引用类型参数的泛型类型并不会影响程序的内存占用，因为其被JIT编译后只生成一份代码。不过若是封闭泛型类型中包含值类型作为参数，那么其JIT编译后的代码则会各不相同。我们再来深入挖掘一下上述过程，看看其带来的影响。

当运行时需要JIT编译一个泛型定义（泛型方法或泛型类），且至少有一个类型参数为值类型时，那么该过程可以分为两个步骤。首先，编译器将创建一个新的IL类，用来表示该封闭泛型类型。例如，在泛型定义中将T用int或其他某种值类型替换。随后，JIT将把该代码编译成x86指令。这两个步骤非常有必要，因为JIT并不是在某个类加载时就为其生成完整的x86指令，而是仅在类中的每个方法被第一次调用时才开始编译的。这样，框架有必要在IL代码上先执行一个替换的步骤，随后再像普通类定义一样按需编译。

这也就意味着运行时的额外内存占用将分为如下两个部分：一是为每种用值类型作为参数的封闭泛型类型保存一份IL定义的副本，二是为每种用值类型作为参数的封闭类型保存一份所调用方法的机器码的副本。

不过这个使用值类型作为泛型参数的做法也有它的好处：避免了对于值类型的装箱和拆箱操作，这样也就降低了值类型的代码/数据所占用的空间。此外，类型安全可以由编译器保证，也就让框架不必忙于进行运行时检查，进一步降低了代码量并提高了程序的性能。不仅如此，与创建泛型类相比，创建泛型方法将有助于降低为支持不同实例而需要额外生成的IL代码量（将在条目8中介绍）。只有实际用到的方法才会被实例化。非泛型类中定义的泛型方法将不会被JIT编译。

本章将介绍泛型的种种使用方法，以及创建泛型类型或方法的一些技巧，从而节约时间并有助于创建具有更高复用性的组件。本章也将介绍如何把.NET 1.x的类型（使用System.Object）迁移至使用泛型的.NET 2.0类型。

条目1：使用1.x框架API的泛型版本

.NET平台的头两个版本并不支持泛型。因此只能基于System.Object来编码，然后通过必要的运行时检查保证程序的正确性（一般来讲，将会检查该对象运行时类型是不是System.Object的某个特定的子类型）。甚至在.NET Framework中也大量存在着这种设计，因为框架本身也需要为开发人员提供一系列的底层组件库。

毫无疑问，System.Object是所有类型的最终基类。因此，使用它就意味着程序可以在此处使用任意的类型。不过编译器对类型的了解也就到此为止了，我们必须小心翼翼地对待每一行代码，使用我们组件的其他开发人员也需要非常仔细。无论是把System.Object作为参数，还是作为返回值，都难免在运行时得到意外的类型输入，随即不可避免地导致运行时错误。

在泛型出现之后，这样的日子就一去不复返了。若是你有过使用从前版本.NET的经历，那么定会非常赞同用最新的泛型版本来替代原有的传统类和接口。将System.Object用泛型的类型参数替代能够极大地提高代码质量。为什么呢？因为泛型类型的约束让人很难传入错误类型的参数。

如果上述程序健壮性的优势无法说服你从原始的System.Object迁移到泛型代码，那么性能则会是第二个理由。.NET 1.1强制使用最基础的System.Object类型，只有在使用之前才将其动态强制转换为需要的类型。1.1版本的任何类/接口都需要在进行值类型和System.Object转换的过程中执行装箱/拆箱的操作。根据实际情况，这个操作有可能会给性能带来巨大的影响。虽然这仅仅影响到了值类型，不过正如前面提到过的那样，1.1版本的弱类型系统需要我们在程序中添加很多检查性质的代码，以便保证参数和返回值类型的正确。哪怕检查结果没有任何问题，这些冗繁的步骤也会带来额外的性能开销。而在失败时的开销会更大：包括栈审核（stack walk）和转换异常时的栈解退（stack unwinding）、运行时定位catch语句等。总体说来，弱类型系统将带来各种各样的麻烦，从性能低下直至程序异常终止等。

开始学习泛型的第一步就是对.NET Framework 2.0有充分的了解。显然，我们可以从System. Collections.Generics命名空间入手，紧接着的还有System.Collections.ObjectModel命名空间。每个System.Collections命名空间的原有类都有一个新的、改进了的版本，位于System.Collections.Generics中。例如ArrayList的替代品是List<T>，Stack由Stack<T>代替，Hashtable由Dictionary<K,V>代替，Queue由Queue<T>代替等。此外，System.Collections.Generics命名空间中还提供了几个新的集合，例如SortedList<T>和LinkedList<T>等。

这些类的引入自然也带了新的泛型接口。我们同样可以从System.Collections.Generics命名空间入手。原有的IList接口由IList<T>进行了扩展。所有基于集合的接口都进行了升级：IDictionary<K,V>替换了IDictionary，IEnumerable<T>扩展了IEnumerable，IComparer<T> 替换了IComparer，ICollection<T>替换了ICollection等。

注意到上面谨慎地区分了“扩展”和“替换”两个词。这是因为很多泛型接口都从其非泛型的版本继承而来，为原有的功能“扩展”出了特定于类型的版本。而有些原有的接口并没有被包含在新接口的签名中。出于种种原因，新的接口方法签名不能与原有的接口保持一致。此时，新版本接口则无法继承于原接口。

.NET Framework 2.0还添加了一个IEquatable<T>接口，降低了因为重写System.Object.Equals方法而导致潜在错误的可能性。

```
public interface IEquatable<T>
{
    bool Equals(T other);
}
```

无论何时重写了System.Object.Equals方法，你都应该同时也添加对该接口的支持。

若需要对定义在其他类库中的类型进行比较，那么还可以使用System.Collections.Generics命名空间中给出的另一个泛型接口：IEquality-Comparer<T>。该接口包含两个方法：Equals和GetHashCode。

```
public interface IEqualityComparer<T>
{
    int Equals( T x, T y);
    int GetHashCode(T obj);
}
```

你可以为任何第三方类型创建一个辅助类，让其实现IEqualityComparer<T>接口。该类有些类似于1.1版本中的IHashCodeProvider，允许开发人员创建类型安全的相等比较功能，而无需再使用原有的基于System.Object的版本。不过在绝大多数时间内，你并不需要完整地实现IEqualityComparer<T>接口，而是可以直接使用EqualityComparer<T>类的Default属性。例如，我们可以提供如下的EmployeeComparer类，让其继承自EqualityComparer<T>，并使用它来判断两个定义与其他类库中的Employee对象是否相等。

```
public class EmployeeComparer : EqualityComparer<Employee>
{
    public override bool Equals(Employee x, Employee y)
    {
        return EqualityComparer<Employee>.Default.Equals(x, y);
    }

    public override int GetHashCode(Employee obj)
    {
        return EqualityComparer<Employee>.Default.
            GetHashCode(obj);
    }
}
```

Default属性将检查类型参数T。若T实现了IEquatable<T>，那么Default将返回一个使用了该泛型接口的IEqualityComparer<T>。若没有实现，那么Default将返回一个使用了System.Object的Equals()和GetHashCode()方法的IEqualityComparer<T>。这样，EqualityComparer<T>即可保证提供最佳的实现方法。

上述这些指出了泛型类型的一个重要特性：越是基础的算法逻辑，例如等同性比较，越有可能需要一个泛型类型定义。在编写有几个变量的基础算法时，我们应该考虑充分借助泛型类型定义所提供的编译期类型检查功能。

接着就来详细介绍，我们将从System命名空间开始学习.NET Framework 2.0中其他的泛型类。之所以需要这样的介绍，主要考虑到两个原因。首先，若你有一段时间的C#经验，那么定会习惯于没有这些类的日子。因此需要改变原有的习惯，充分利用到泛型带来的好处。其次，框架内建的类就提供了很多非常不错的使用泛型解决问题的示例。

前面曾经提到过，System.Collections.Generic命名空间中包含了一个IComparer接口的升级版本：IComparer<T>。System命名空间中也包含了一个泛型的比较接口：IComparable<T>。

```
public interface IComparable<T>
{
    int CompareTo(T other);
}
```

与之类似的老版本的IComparable接口如下。

```
// 1.1版本中IComparable的签名
public interface IComparable
{
    int CompareTo(object other);
}
```

这种典型的实现很清晰地表明了泛型版本的优势。下面是一段来自我的前一本书（*Effective C#*）中条目9的代码。

```
// Customer结构（值类型）中的一段代码
public int CompareTo(object right)
{
    if (!(right is Customer))
        throw new ArgumentException("Argument not a customer",
            "right");
    Customer rightCustomer = (Customer)right;
    return Name.CompareTo(rightCustomer.Name);
}
```

与依赖System.Object的IComparable相比，IComparable<T>版本的代码非常简单。

```
public int CompareTo(Customer right)
{
    return Name.CompareTo(right.Name);
}
```

使用泛型接口，你会由类型安全得到4个优势。注意IComparable和IComparable<T>实现上的差异。IComparable<T>显得更加简洁，因为IComparable中手工实现的运行时类型检查在IComparable<T>中是由编译器实现的。这样，只需少量代码即可实现原本的功能。泛型版本也更加高效，因为其中无需过多的错误检查。此外，泛型版本还避免了所有的装箱/拆箱以及类型转换操作。且泛型版本也不会抛出任何运行时异常。在泛型版本中，非泛型版本中所有可能出现的运行时异常均可由编译器捕获。

当然，有些时候System.Object仍旧是不可缺少的。程序中可能需要比较属性相似、但类型不同（或不在一个继承链中）的两个对象。在这种情况下，则应该实现传统的接口以及最新的泛型接口（参见本章条目10）。假设我们原有的发货系统必须和一个第三方的电子商务系统进行集成。两个系统均有订单（Order）的概念，不过二者却并没有任何继承关系。

```
namespace ThirdPartyECommerceSystem
{
    public class Order
    {
        // 细节省略
    }
}
```

这时即可修改我们的系统，并使用传统的接口，从而支持比较两种Order对象。

```
namespace InternalShippingSystem
{
    public class Order : IEquatable<Order>,
        IComparable<Order>
    {
        #region IEquatable<Order> Members
```

```
        public bool Equals(Order other)
        {
            // 省略
            return true;
        }
        #endregion

        #region IComparable<Order> Members
        public int CompareTo(Order other)
        {
            // 省略
            return 0;
        }
        #endregion

        public override bool Equals(object obj)
        {
            if (obj is Order)
                return this.Equals((Order)obj);
            else if (obj is ThirdPartyECommerceSystem.Order)
                return this.Equals
                    ((ThirdPartyECommerceSystem.Order)obj);
                throw new ArgumentException(
                    "Object type not supported", "obj");
        }
        public bool Equals(ThirdPartyECommerceSystem.Order
            other)
        {
            bool equal = true;
            // 具体检验步骤省略
            return equal;
        }
    }
}
```

传统的接口能够支持此类表示同样信息而类型不同的对象之间的比较。虽然这并不是个故意要添加至系统中的设计，不过当你不得不处理来自不同提供者的类库中的相似对象时，这也就是能想到的最好方法了。

仔细察看上述代码，可以注意到System.Object.Equals()的重写实现中抛出了运行时错误，而不是编译期错误。

无论何时，只要能将System.Object替换成适当的类型，那么即可获得编译期类型安全以及一定的性能提升。可以看到，在几乎所有情况下，IComparable<T>都要比IComparable更加有优势。实际上，几乎所有的使用System.Object的1.x接口都有了相对应的泛型版本。唯一一个明显的反例是ICloneable，它并没有与之匹配的泛型版本。因为在设计中并不鼓励实现该接口。请参考Krzysztof Cwalina和Brad Abrams的*Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*（Addison-Wesley, 2005），221-222页。

.NET Framework 2.0中的一个很常见的泛型应用是可空泛型类型（nullable generic type）。可空类型是由两个互为补充的类型实现的：Nullable<T>结构和静态的Nullable类。Nullable<T>结构是所有需要表示空值的值类型的统一包装（参见第6章条目45）。目前，你只要了解可空类型和其实际的值类型实例是通过泛型类型联系起来的就够了。

为了能够支持可空值类型，.NET基础类库添加了静态的Nullable类和Nullable<T>泛型结构。Nullable类也包含了一些泛型方法，用来处理可空类型。例如，Nullable.GetUnderlyingType(Type t)方法将返回该Nullable<T>中实际对象的类型。换句话说，如下的定义可以用来比较两个可空类型，并判断其是否相等。

```
Nullable.Compare<T> (Nullable<T> left, Nullable<T> right);
```

总体上，上述类提供了C#编译器所需要的实现可空类型的各种功能。

.NET Framework同样也添加了一些泛型的委托，用来实现一些常见的模式。例如，若想用一个回调来访问某个泛型集合中的每个元素，那么可以使用foreach循环配合IEnumerable<T>实现。

```
List<MyType> theList = new List<MyType>();
foreach (MyType thing in theList)
    thing.DoSomething();
```

这里可以将DoSomething替换成匹配System.Action委托的任意函数。

```
public delegate void Action<T>(T obj);
```

用一个简单的泛型函数即可访问到集合中的每个元素，并执行特定的操作。

```
public static void EnumerateAll<T>(IEnumerable<T>
theCollection,
    Action<T> doIt)
{
    foreach (T thing in theCollection)
        doIt(thing);
}
```

当然，在C# 3.0中，使用扩展方法看起来更加漂亮。

```
public static void EnumerateAll<T>(this IEnumerable<T>
theCollection,
    Action<T> doIt)
{
    foreach (T thing in theCollection)
        doIt(thing);
}
```

若想对一系列没有实现IComparable<T>的对象进行排序，那么可以提供一个匹配System.Comparison委托的委托。

```
delegate int Comparison<T>( T x, T y);
```

该委托已经用于List.Sort<T>(Comparison<T> comparison)中，这是Sort()方法的一个重载，允许指定自定义的比较方法。

还有两个委托可以用在以类型安全的方式对集合中的对象进行转换。System.Converter委托能够将一个输入的对象转换为期待的输出。

```
public delegate TOutput Converter<TInput, TOutput>(TInput
    input);
```

该委托支持用一个泛型方法将一系列的某种类型对象转换成一系列的另一种对象。

```
public IEnumerable<TOutput> Transform<TInput, TOutput>(
    IEnumerable<TInput> theCollection,
    Converter<TInput, TOutput> transformer)
{
    foreach (TInput source in theCollection)
        yield return transformer(source);
}
```

Transform包含两个类型参数：TInput和TOutput。它们分别表示输入和输出的类型。System.Converter使用同样的命名规则来描述两个类型。

有些时候，程序需要逐一检查集合中的每个对象。这时可以创建一个使用System.Predicate委托的方法。

```
delegate bool Predicate<T>(T obj)
```

只需略加修改，即可过滤出满足指定条件的元素，并组成新的集合。

```
public IEnumerable<T> Test<T> (IEnumerable<T> theCollection,
Predicate<T> test)
{
    foreach (T source in theCollection)
        if ( test( source ) )
            yield return source;
}
```

泛型还可以简化事件相关的代码。在.NET 1.1中，我们需要首先创建继承于EventArgs的类，然后创建委托定义，最后才是提供匹配委托签名的事件定义。虽然不难，不过步骤却很繁琐。为了解决这个问题，.NET Framework 2.0中提供了一个事件处理程序的泛型定义。

```
public delegate void EventHandler<TEventArgs>(
    object sender, TEventArgs args)
    where TEventArgs: EventArgs
```

这个定义能够让你省去大多数用自定义代码来创建委托的重复工作。例如，如下的委托和事件的定义

```
public delegate void MyEventHandler(object sender,
    MyEventArgs args);
public event MyEventHandler OnRaiseMyEvent;
```

可以用如下这种新方法代替。

```
public event EventHandler<MyEventArgs> OnRaiseMyEvent;
```

虽然是个小小的改进，不过在需要定义很多事件时，也会集腋成裘。

这些就是System命名空间中提供的主要泛型接口。但还有一个重要的类型没有提到：System.ComponentModel.BindingList<T>。在.NET 1.x中创建一个实现了IBindingList的类型是个非常枯燥又麻烦的工作。而且，其中大多数代码都十分相似，且与列表中实际的类型并没有太大关系。但IBindingList、ICancelAddNew和CurrencyManager等类型之间需要配合使用，因此出错的机率往往会很大。例如，若想为某个表示员工的类创建一个实现IBindingList接口的类，那么难免需要数页的代码。而在.NET 2.0中，却得到的极大的简化。

```
System.ComponentModel.BindingList<Employee>
```

哪一种更加简单呢？显而易见。

本条目介绍了.NET Framework 2.0中主要的泛型类。可以看到使用泛型意味着编写更少的错误检查代码，因为编译器将为你验证参数和返回值的类型。泛型能够带来很多优势，所以你必须熟悉并学会使用这些泛型定义。

最后的问题就是，如何处理现有的.NET 1.1代码。我建议你尽早将现有代码升级成为泛型版本，不过不必专门为此安排单独的工作。已经能够正常运行的代码并不需要修改，不过一旦升级完毕，那么就可以借助编译器发现潜在以及将来可能出现的bug。当你接手了一个维护非泛型类的大项目时，可以先从将其转换为泛型版本开始，借助编译器帮你找到可能遇到的问题。

这个条目着重于让你改变原有的使用非泛型API编写应用程序的习惯，转向最新的泛型版本。泛型能够让你的组件更易于使用，同时避免很多错误。编译器能够帮你完成更多的类型检查。使用内建的泛型委托定义，我们还能够使用到C# 3.0的更多增强特性，例如让编译器进行类型推断等，从而让你全力投入到对算法和逻辑的思考中。越早将非泛型代码切换到泛型版本，你就能够越

早地享受到最新语言特性所带来的便利。

条目2：恰到好处地定义约束

类型参数的约束指出了能完成该泛型类工作的类必须具有的行为。若是某一类型无法满足约束，那么自然无法用于该泛型类型中。不过这也意味着，每次在泛型类型中引入新的约束，都会给该类型的使用者增加更多的工作。实际情况各不相同，因此并没有万能的解决方案，不过太过极端总归是不好的。若是不给出任何约束，那么则必须在运行时进行过多检查，比如使用强制转换、反射并抛出运行时异常等来保证程序的正确性。而若是约束过多，那么也会让类的使用者觉得麻烦。因此你要找到那个恰到好处的中间点，精确地对类型参数给出约束，不多也不少。

约束能让编译器了解某个类型参数更具体的信息，而不仅限于极为笼统的System.Object。在创建泛型类型时，C#编译器将要为泛型类型的定义生成合法的IL。而在进行编译时，虽然编译器对今后可能用来替换类型参数的具体类型了解甚少，但仍需要生成合法的程序集。若是不添加任何约束，那么编译器只能假设这些类型仅具有最基本的特性，即System.Object中定义的方法。编译器无法猜测出你对类型的假设，唯一能够确认的就是该类型继承于System.Object。（因此我们无法创建不安全的泛型，也无法将指针作为类型参数。）我们知道，System.Object的功能非常有限，因此若是使用到了任何非System.Object的功能，编译器均会抛出异常。你甚至都无法使用最基础的构造函数new T()，因为若某个类型仅提供了有参数的构造函数，那么该默认构造函数将会被隐藏。

因此，约束能够让编译器和用户充分了解到我们对泛型类型参数的假设。约束让编译器认识到，某个泛型类型并不仅仅拥有System.Object里公共接口的功能。这能让编译器得到如下两个方面的便利。首先，有助于泛型类型的编写：编译器将会认为该泛型类型参数具有约束所定义的各种功能。其次，

编译器还能保证使用该泛型类型的用户所指定的类型参数一定会满足约束的条件。

如果不使用约束，那么势必要执行大量的强制转换以及运行时检验工作。例如，如下的泛型方法并没有对T声明任何约束，因此在使用相应的方法之前必须对是否实现了IComparable<T>接口进行检查。

```
// 没有约束
public bool AreEqual<T>(T left, T right)
{
    if (left == null)
        return right == null;

    if (left is IComparable<T>)
    {
        IComparable<T> lval = left as IComparable<T>;
        return lval.CompareTo(right) == 0;
    }
    else // failure
    {
        throw new ArgumentException(
            "Type does not implement IComparable<T>",
            "left");
    }
}
```

而若是规定T必须实现了IComparable<T>接口，那么上一个方法将会变得非常简单。

```
public bool AreEqual<T>(T left, T right)
    where T : IComparable<T>
{
    return left.CompareTo(right) == 0;
}
```

第二个版本将可能抛出的运行时错误改为了编译期错误。只需更少的代码，即可让编译器帮你预防第一个版本中必须重新编码来避免的运行时错误。若是没有了约束，那么则无法如此清晰地给出错误提示。考虑到这些，我们应该明确指定泛型类型的约束。否则若使用者误解的话，那么该泛型类就可能会

被误用、产生异常或其他运行时错误。误用的情况很常见，因为你的类的使用者若是想了解其用法，则只能够阅读文档。而本身就是开发人员的你，自然能够想象到这些文档在开发者眼中的地位[①]。但使用约束即可让编译器保证程序的正确性，降低运行时错误的数量以及误用的可能性。

不过定义了太多的约束也不是件好事。对泛型类型参数的约束越多，也就表示泛型类能够适用的场合越少。因此，尽管添加必要的约束是必不可少的，但也需要尽可能地让约束足够宽松。

最小化约束的方法有很多种。其中最常见的一种是，确保泛型类型不要求其不需要的功能。以IEquatable<T>为例，这是个很常用的接口，且创建新类型时也经常用到。我们可以重写AreEqual方法，让其调用Equals方法。

```
public static AreEqual<T>(T left, T right)
{
    return left.Equals(right);
}
```

在上述代码中，若AreEqual<T>()定义在一个带有IEquatable<T>约束的泛型类中，那么AreEqual将调用IEquatable<T>.Equals。否则，C#编译器则不会假设具体类型一定会实现IEquatable<T>，因此唯一可用的Equals()方法就是System.Object.Equals()。

上述示例可以看到C#泛型和C++模板之间的主要区别。在C#中，编译器仅能使用约束中给出的信息来生成IL。即使为某个特定的实例指定的类型拥有更好的方法，也不会在运行时使用，除非在该泛型类型编译时就明确地给出了限定。

如果实现了IEquatable<T>接口的话，那么显然使用其中定义的方法来判断等同性才是最合适的。它可以避免在运行时检查对象是否提供了System.Object.Equals()的重写，也可以避免对于值类型的装箱/拆箱操作。从性能角度考虑，IEquatable<T>也能省去调用虚方法所带来的一点开销。

① 作者暗指开发人员不乐于阅读程序/组件的开发设计文档。——译者注

因此，为类型提供IEquatable<T>是个不错的做法。但是否有必要将其作为约束而规定下来呢？如果已经有了个很不错的System.Object.Equals方法(虽然性能上可能略显不足)，那么别人是否还一定需要实现IEquatable<T>呢？本书推荐的做法是，如果能够使用IEquatable<T>的话，那么则尽量使用，否则也要支持使用其Equals()方法完成比较。这可以通过根据要支持的类型的不同而创建内部的重载方法来实现。这也是本条目开始部分曾给出的AreEqual()方法的做法。这种做法需要我们更多的工作，不过你会看到如何根据特定类型的功能去选择调用最合适的方法，且这样做也并不给其他开发人员带来任何的额外工作。

有些时候添加的约束会极大地限制类的应用范围，这时则需要权衡该限制是否为必须存在的，并考虑在缺少某一接口或基类时程序的写法。也就是将限制放宽，不做强制要求。这时，在编写代码时，则需要考虑类型参数有可能会提供额外的功能，但也有可能不提供。这种做法的实例在Equatable<T>和Comparable<T>中都可以见到。

你也可以将该技术应用到其他有泛型和非泛型接口的约束中，例如IEnumerable和IEnumerable<T>。

另外一个需要留心的地方就是默认构造函数约束（default constructor constraint)。有时我们可以将new()约束用default()调用来替换。后者是C#中的一个新操作符，用来将变量初始化成其默认值。该操作符将把值类型的所有位设置为0，并把引用类型设置为null。因此用default()替代new()往往会引起类约束或值约束。需要注意的是，对于引用类型，default()和new()的语义是完全不同的。

你会经常看到泛型类中使用了default()获得类型参数的默认值。下面的这个方法将搜索集合中第一个满足指定条件的对象。若存在该对象，则返回。否则将返回一个默认值。

```
public static T FirstOrDefault<T>(this IEnumerable<T> sequence,
    Predicate<T> test)
{
```

```
        foreach (T value in sequence)
            if (test(value))
                return value;

        return default(T);
    }
```

将该方法与下面的这个对比。可以看到下面的工厂方法用来创建一个类型为T的新对象。若该工厂方法返回null，那么该方法将返回由默认构造函数给出的值。

```
public delegate T FactoryFunc<T>();
public static T Factory<T>(FactoryFunc<T> makeANewT)
    where T : new()
{
    T rVal = makeANewT();
    if (rVal == null)
        return new T();
    else
        return rVal;
}
```

使用了default()的方法并不需要任何约束。而调用了new T()的方法则必须给出new()约束。且考虑到检验空值的方式，值类型和引用类型的行为将完全不同。值类型不可能为空。因此if语句的子句将永远不会执行。但即使其在内部检查了空值，Factory<T>仍可配合值类型使用。因为若T为值类型的话，JIT编译器（用来将T替换成特定类型）将自动去除该空值检验语句。

需要特别注意的是new()、struct和class约束。前面的例子说明，添加此类约束也就意味着对对象的创建方式给出了假定，包括该对象的默认值是均为0还是空引用，是否能够在该泛型类内部创建泛型类型参数实例等。在最理想的情况下，我们应该尽力避免使用上述三种约束。因此在你“自然”地下决定（“我当然需要调用new T()了”）之前，应该考虑一下是否有替代的做法（例如使用default(T)）。仔细考虑你在不经意之间作出的假设，将那些不是真正必要的都丢掉。

若想将你的假设告知给你的泛型类型的使用者，则需要给出约束。不过，

指定越多的约束也就意味着类的使用范围越窄。我们的最终目标是让创建出的泛型类型能够尽可能地应用到更多的场景中。因此需要在约束保证的安全性和约束给他人带来的使用上的不便之间找到平衡点。一方面努力降低所需要的假设的数量，另一方面也要把必需的假设以约束的形式确定下来。

条目 3：运行时检查泛型参数的类型并提供特定的算法

只要指定不同的类型参数，即可重用泛型。给出新的类型参数意味着得到一个带有类似功能的全新类型。

这是个很不错的功能，能够大大节省代码。不过有些时候，过多地使用泛型也可能无法让一些算法达到最优化的解决方案。C#语言的规则考虑到了这一点。当类型参数提供了更为强大的功能时，我们可以用特定的代码充分使用这些功能。不过这并不一定需要创建一个新的泛型类型，并指定不同的约束来实现。要知道泛型的实例化是基于对象的编译期类型而不是运行时类型的。因此，若不考虑到这一点，那么将会错过很多本可以提高性能的机会。

例如，若我们需要编写一个类，用来将一个序列中的元素反序。

```
public sealed class ReverseEnumerable<T> : IEnumerable<T>
{
    private class ReverseEnumerator : IEnumerator<T>
    {
        int currentIndex;
        IList<T> collection;

        public ReverseEnumerator(IList<T> srcCollection)
        {
            collection = srcCollection;
            currentIndex = collection.Count;
        }

        #region IEnumerator<T> Members
        public T Current
```

```
        {
            get { return collection[currentIndex]; }
        }
        #endregion

        #region IDisposable Members
        public void Dispose()
        {
            // 不需要实现
            // 无需实现保护的Dispose()方法，因为该类型为密封类
        }
        #endregion

        #region IEnumerator Members
        object System.Collections.IEnumerator.Current
        {
            get { return this.Current; }
        }

        public bool MoveNext()
        {
            return --currentIndex >= 0;

        }

        public void Reset()
        {
            currentIndex = collection.Count;
        }
        #endregion
    }

    IEnumerable<T> sourceSequence;
    IList<T> originalSequence;

    public ReverseEnumerable(IEnumerable<T> sequence)
    {
        sourceSequence = sequence;
    }
```

```
    #region IEnumerable<T> Members
    public IEnumerator<T> GetEnumerator()
    {
        // 创建原有序列的副本，以便反序操作
        if (originalSequence == null)
        {
            originalSequence = new List<T>();
            foreach (T item in sourceSequence)
                originalSequence.Add(item);
        }
        return new ReverseEnumerator(originalSequence);
    }
    #endregion

    #region IEnumerable Members
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
    {
        return this.GetEnumerator();
    }
    #endregion
}
```

这种实现仅仅做了非常少的约束。ReverseEnumerable构造函数只不过假设其输入参数支持IEnumerable<T>而已。而IEnumerable<T>并不支持随机访问其中的元素。因此，我们只能使用ReverseEnumerator<T>.GetEnumerator()中给出的方法来对元素进行反序。即在第一次调用构造函数时，遍历整个输入序列并复制出来。然后使用内嵌类来反序遍历该序列。

这个方法没什么问题，且若是实际输入的集合确实不支持随机访问元素的话，这也是仅有的一种反序方法。不过实际上，这段代码却有些丑陋。日常中经常使用的很多集合均支持随机访问，考虑到这些，上述实现显然非常低效。若是输入的序列本身就支持IList<T>，那么再对整个序列进行一次完整的复制也就毫无意义。因此，考虑到很多实现了IEnumerable<T>接口的集合也实现了IList<T>，上述代码还有较大的改进空间。

我们仅需要修改ReverseEnumerable<T>类的构造函数即可。

```
public ReverseEnumerable(IEnumerable<T> sequence)
{
    sourceSequence = sequence;
    // 若序列没有实现IList<T>,
    // 那么originalSequence为空，可以正常执行
    originalSequence = sequence as IList<T>;
}
```

为什么不直接给出一个新的构造函数，让其接受IList<T>呢？若编译期的类型就为IList<T>，那自然没有问题。不过在另外一些情况下有可能还不够，例如，某参数的编译期类型为IEnumerable<T>，而在运行时却实现了IList<T>。为了能够处理这种情况，我们需要同时提供运行时检查和编译期重载。

```
public ReverseEnumerable(IEnumerable<T> sequence)
{
    sourceSequence = sequence;
    // 若序列没有实现IList<T>,
    // 那么originalSequence为空，可以正常执行
    originalSequence = sequence as IList<T>;
}

public ReverseEnumerable(IList<T> sequence)
{
    sourceSequence = sequence;
    originalSequence = sequence;
}
```

与IEnumerable<T>相比，IList<T>在效率上更胜一筹。但也不必强迫使用者一定提供IList <T>的高级功能，不过若是确实提供了的话，自然也要做到物尽其用。

上述修改已经能够覆盖到绝大多数情况，不过仍有一些集合实现了ICollection<T>，但却没有实现IList<T>。对于这种情况，该代码仍旧不够高效。我们看一下ReverseEnumerable <T>.GetEnumerator()方法。

```
public IEnumerator<T> GetEnumerator()
{
    // 创建原有序列的副本，以便反序操作
    if (originalSequence == null)
    {
        originalSequence = new List<T>();
        foreach (T item in sourceSequence)
            originalSequence.Add(item);
    }
    return new ReverseEnumerator(originalSequence);
}
```

若集合实现了ICollection<T>的话，那么首先创建源序列的副本将不必要地降低程序的效率。下面这个方法就添加了Count属性，可以用于初始化最终的存储变量。

```
public IEnumerator<T> GetEnumerator()
{
    // 创建原有序列的副本，以便反序操作
    if (originalSequence == null)
    {
        if (sourceSequence is ICollection<T>)
        {
            ICollection<T> source = sourceSequence
               as ICollection<T>;
            originalSequence = new List<T>(source.Count);
        }
        else
            originalSequence = new List<T>();
        foreach (T item in sourceSequence)
            originalSequence.Add(item);
    }
    return new ReverseEnumerator(originalSequence);
}
```

这里给出的代码类似于List<T>的构造函数，将会根据输入序列创建出一个列表对象。

```
List<T>(IEnumerable<T> inputSequence);
```

不过在结束本条目之前，还要说明这种实现的一个负面影响。注意到ReverseEnumerable<T>中的检验均为运行时检查。这也就造成了运行时一定的额外开销。但好在大多数情况下，这些额外开销要比复制整个元素集合的开销小得多。

或许你会觉得我们对ReverseEnumerable<T>的考虑已经很周全了。不过仍有一个疏忽：string类。string类提供了随机访问其中字符串的方法，就像IList<char>一样，不过却并没有实现IList<char>。因此，需要我们在该泛型类中添加更多的特别代码。下面的代码给出了嵌入到ReverseEnumerable<T>中的ReverseStringEnumerator类。可以看到其构造函数中使用了string的Length属性，其他方法均和ReverseEnumerator<T>中的类似。

```
private class ReverseStringEnumerator : IEnumerator<char>
{
    private string sourceSequence;
    private int currentIndex;

    public ReverseStringEnumerator(string source)
    {
        sourceSequence = source;
        currentIndex = source.Length;
    }

    #region IEnumerator<char> Members
    public char Current
    {
        get { return sourceSequence[currentIndex]; }
    }
    #endregion

    #region IDisposable Members
    public void Dispose()
    {
       // 无需实现
    }
    #endregion
```

```
    #region IEnumerator Members
    object System.Collections.IEnumerator.Current
    {
        get { return sourceSequence[currentIndex]; }
    }

    public bool MoveNext()
    {
        return --currentIndex >= 0;
    }

    public void Reset()
    {
        currentIndex = sourceSequence.Length;
    }
    #endregion
}
```

最后，ReverseEnumerable<T>.GetEnumerator()需要检查参数的类型，并创建合适的枚举器（enumerator）类型。

```
public IEnumerator<T> GetEnumerator()
{
    // string是个特例：
    if (sourceSequence is string)
    {
        // 注意该转换，
        // 因为T在编译期可能不是char
        return new ReverseStringEnumerator
            (sourceSequence as string)
            as IEnumerator<T>;
    }

    // 创建原有序列的副本，以便反序操作
    if (originalSequence == null)
    {
        if (sourceSequence is ICollection<T>)
        {
            ICollection<T> source = sourceSequence
                as ICollection<T>;
```

```
                originalSequence = new List<T>(source.Count);
            } else
                originalSequence = new List<T>();
            foreach (T item in sourceSequence)
                originalSequence.Add(item);
        }
        return new ReverseEnumerator(originalSequence);
    }
```

和前面介绍过的一样，这样做的目的是在泛型类型中隐藏所有的特殊实现。它自然会多出一些工作，因为string类本身的特殊性需要一个专门的内部类来处理。

注意，在使用ReverseStringEnumerator时，GetEnumerator()的实现中进行了一次强制转换。在编译期，T可能为任何类型，自然有可能不是char。不过这个强制转换却是安全的，因为只有序列为string，也就是T为char的时候才可能执行到这部分代码。这个做法不会有什么问题，因为ReverseStringEnumerator隐藏在了外部的类中，不会污染公共接口。可以看到，泛型的存在并不能让我们将对类型的维护完全交给编译器负责，有时仍需手工作一些特殊处理。

上述简单示例演示了如何让泛型类型在尽可能宽泛的约束情况下，仍旧能够根据实际类型可能会提供的高级功能来有针对性地优化实现过程。这样即可同时照顾到重用和性能两方面，并找到其最佳的契合点。

条目4：使用泛型强制编译期类型推断

模式的作用就在于标准化那些尚未被语言或框架直接支持的常见算法逻辑。遵循模式即可解决一类常见的问题。提供模式是为了在无法进行代码层面的重用时，也能给出一定程度上编程思想的重用。若使用模式无法得到最优化方案，那是因为很难从模式的实现代码中分离开所有的特殊部分。

泛型类能够帮助你实现很多常用的、可重用的模式。模式的本质在于，

它必定包含一系列通用的算法和代码，支持应用于应用程序中各种特定类型上。泛型无法应用于每个模式之上，不过却能极大地减少实现某个模式所需要的代码。

.NET Framework中的集合类就在若干个实现了枚举器模式（enumerator pattern）的方法中演示这种功能。List<T>Find (Predicate<T> match)将找到集合中第一个满足指定条件的元素。List类还提供了一些类似的方法，例如FindAll、TrueForAll和ForEach等。当然，这些方法的具体实现并不是什么惊天地、泣鬼神之笔。但这种设计原则却非常有启发性，因为我们能够从中学到何时可以使用同样的模式。现有的集合类均包含相应的遍历其中每个元素的算法。客户端代码仅需要提供针对集合中每个元素的操作方法即可。

基于这样的模式，.NET Framework设计者并不需要预先考虑使用者将会对集合中的元素进行何种操作。相反，它提供了访问每个元素的泛型方法，并支持在其中针对当前元素调用使用者给出的特定函数。遍历所有元素的理由可能有很多，例如查找、检查值或进行转换等。

.NET开发者允许你根据需要自行定义检查或断言的具体实现。实际上，这些断言也可以以内联匿名方法的方式给出，请参考条目18（第3章）。

遵循同样的方式，你可以以断言的形式定义合适的委托或事件，进而实现很多设计模式。今后再次需要同样模式的时候，即可直接重用这些泛型实现了。

接下来将介绍两个示例，演示如何创建出易于重用且能避免误用的模式。其中较为简单的一个是使用XML序列化器（Serializer）对对象进行序列化。而其非泛型的版本将允许传入任何的类型，像下面这样。

```
public static class XmlPersistenceManager
{
    public static object LoadFromFile(Type typeToLoad,
        string filePath)
    {
        XmlSerializer factory = new XmlSerializer(typeToLoad);
```

```
            if (File.Exists(filePath))
            {
                using (TextReader r = new StreamReader(filePath))
                {
                    object rVal = factory.Deserialize(r);
                    return rVal;
                }
            }
            return default(object);
        }

        public static void SaveToFile(string filePath, object obj)
        {
            Type theType = obj.GetType();
            XmlSerializer factory = new XmlSerializer(theType);

            using (TextWriter w = new StreamWriter(filePath,
                false))
            {
                factory.Serialize(w, obj);
            }
        }
    }
```

这当然能实现我们的要求，不过却并不够易用。使用该类型的开发者必须指定Type参数，而无法通过推断得出。

嗯，经过一段时间后，我们需要序列化另一种类型了。这两个方法的应用非常广泛——有时是同样的类型，而有时是不同的类型。问题虽然不大，但上述代码中仍存在一些缺点。首先是类型安全：每次调用LoadFromFile时，都需要对返回值进行类型强制转换或转换。虽然这时该强制转换是难免的，不过我们当然希望能尽可能地降低执行此类操作的频率。

代码中还有一处较为低效。即每次调用这些方法时都会创建一个新的XmlSerializer。在应用程序中，这样将会创建出过多的XmlSerializer对象。虽然框架本身已经尽可能地降低了创建对象所花费的开销，但代码本身不应该肆意挥霍框架的努力，随意地创建并销毁不必要的临时对象。考虑到这些，你可能会按照这样修改代码。

```
// 错误做法。在缓存XmlSerializer时存在问题
public static class XmlPersistenceManager
{
    // 创建后就缓存XmlSerializer
    private static XmlSerializer factory;

    public static object LoadFromFile(Type typeToLoad,
        string filePath)
    {
        if (factory == null)
            factory = new XmlSerializer(typeToLoad);
        if (File.Exists(filePath))
        {
            using (TextReader r = new StreamReader(filePath))
            {
                object rVal = factory.Deserialize(r);
                return rVal;
            }
        }
        return null;
    }

    public static void SaveToFile(string filePath, object obj)
    {
        Type theType = obj.GetType();
        if (factory == null)
            factory = new XmlSerializer(theType);

        using (TextWriter w = new StreamWriter(filePath,
            false))
        {
            factory.Serialize(w, obj);
        }
    }
}
```

看上去不错：创建XmlSerializer一次，然后缓存起来以备后续使用。这甚至还能通过简单的单元测试。不过实际上却并非如此，若用该XmlPersistenceManager序列化超过一种类，就会出现问题。每个XmlSerializer均与一种特定的Type绑定，也只能序列化这种类型。若是让其再操作另一种Type

实例，那么将抛出异常。

当然，你也可以修改上述代码，让其保存一个与各种不同Type绑定的XmlSerializer的散列表。不过这也将带来更多的代码，这些工作其实是可以由编译器和JIT帮你完成的。（虽然也可以让XmlSerializer实例支持多个类型，不过它并不是这里想要的答案。）

不难看出，其实这就是复制了完全一样的算法，然后进行了一些替换而已。这不正是编译器处理泛型的方法吗？因此可以这样修改。

```
// 这里的缓存没有任何问题
// 因为对于每一种类型T，均缓存了一个专门的泛型实例
public static class GenericXmlPersistenceManager<T>
{
    // 创建后就缓存XmlSerializer:
    private static XmlSerializer factory;

    public static T LoadFromFile(string filePath)
    {
        if (factory == null)
            factory = new XmlSerializer(typeof (T));
        if (File.Exists(filePath))
        {
            using (TextReader r = new StreamReader(filePath))
            {
                T rVal = (T)factory.Deserialize(r);
                return rVal;
            }
        }
        return default(T);
    }

    public static void SaveToFile(string filePath, T data)
    {
        if (factory == null)
            factory = new XmlSerializer(typeof(T));

        using (TextWriter w = new StreamWriter(filePath,
            false))
```

```
        {
            factory.Serialize(w, data);
        }
    }
}
```

接下来，可能会要考虑将XML节点序列化到一个输出流中。只需略加修改，即可让所有使用者都能受益。

```
public static class GenericXmlPersistenceManager<T>
{
    // 创建后就缓存XmlSerializer:
    private static XmlSerializer factory;

    public static T LoadFromFile(string filePath)
    {
        if (File.Exists(filePath))
        {
            using (XmlReader inputStream = XmlReader.Create(
                filePath))
            {
                return ReadFromStream(inputStream);
            }
        }
        return default(T);
    }

    public static void SaveToFile(string filePath, T data)
    {
        using (XmlWriter writer = XmlWriter.Create(filePath))
        {
            AddToStream(writer, data);
        }
    }

    public static void AddToStream(
        System.Xml.XmlWriter outputStream,  T data)
    {
        if (factory == null)
            factory = new XmlSerializer(typeof(T));
```

```
            factory.Serialize(outputStream, data);
        }
        public static T ReadFromStream(
            System.Xml.XmlReader inputStream)
        {
            if (factory == null)
                factory = new XmlSerializer(typeof(T));
            T rVal = (T)factory.Deserialize(inputStream);
            return rVal;
        }
    }
```

这种做法的另一个好处就是，编译器能够推断出保存方法的类型参数。指定了类型参数的加载方法将只把该参数用于返回的类型，因此开发者仍需要给加载方法传入类型参数。这个泛型的XML序列化器的另外一个优势在于，若是对其进行了修改或增强，所有使用到该类型的实现均会立竿见影看到效果。

你会发现，在实现某一算法逻辑时，我们很多时候需要同时知晓传入参数的类型。这时，通常可以创建一个泛型的实现，将方法的类型参数抽象到泛型参数中。随后编译器即可根据泛型参数创建出所需要类型。

条目5：确保泛型类型支持可销毁对象

约束能够为你和你的用户完成两件事情。首先，约束能将运行时错误转换成编译期错误。其次，约束为你的用户清楚地给出了实例化类型参数所必须实现的功能。不过约束却不能限制类型参数不能做什么。好在大多数情况下，我们无需关心除了必要功能之外，类型参数还附带了哪些额外功能。不过若是类型参数实现了IDisposable，那泛型类型内则还需要一些额外的工作。

有关这个问题的实际例子将会比较有说服力，因此这里将假设一个简单场景，演示这个问题是如何发生的，以及如何在代码中修正该问题。当某个泛型方法需要在其中的某个方法中创建并使用类型参数的实例时，就可能发生该问题。

```
public interface IEngine
{
    void DoWork();
}

public class EngineDriver<T> where T : IEngine, new()
{
    public void GetThingsDone()
    {
        T driver = new T();
        driver.DoWork();
    }
}
```

若是T实现了IDisposable，那么这里将有可能造成资源泄露。无论何时初始化本地变量T之后，我们都需要检查T是否实现了IDisposable。如果确有实现的话，那么则有必要在使用后销毁。

```
public void GetThingsDone()
{
    T driver = new T();
    using (driver as IDisposable)
    {
        driver.DoWork();
    }
}
```

若你从未见过如此使用using语句，那么定会有些困惑，但这样做的确是合法的。编译期将为上述代码分配一个本地变量，存放将driver强制转换为IDisposable的结果。若T没有实现IDisposable，那么该本地变量将为null。这时，编译器将不会调用Dispose()。而若是T的确实现了IDisposable，那么编译器将在该代码块的结束位置生成对Dispose()方法的调用。

这样就让问题变得简单起来：只需将初始化本地类型参数的代码用using语句包装起来即可。同时需要使用类似前面的转换代码，因为T可能没有实现IDisposable。

若是泛型类中需要实例化某个类型参数，并将其作为成员变量，那么情况将变得复杂一些。这时，泛型类本身将引用一个可能实现了IDisposable的对象。

这也就意味着泛型类本身必须也要实现IDisposable。随后检查这些资源是否实现了IDisposable，如果实现了的话，那么有必要销毁其占用的资源。

```
public sealed class EngineDriver<T> : IDisposable
    where T : IEngine, new()
{
    // 创建的代价较高，因此初始化为空
    private T driver;
    public void GetThingsDone()
    {
        if (driver == null)
            driver = new T();
        driver.DoWork();
    }

    #region IDisposable Members
    public void Dispose()
    {
        IDisposable resource = driver as IDisposable;

        if (resource != null)
        {
            resource.Dispose();
        }
        // 多次调用Dispose没问题
    }
    #endregion
}
```

这样，泛型类一下子变得臃肿了起来。不但添加了IDisposable的实现，还让该类变成了密封类（添加sealed关键字）。若不想这样做，那么则需要实现完整的IDisposable模式，以便让派生类也能使用到你的Dispose()方法（参见Cwalina与Abrams合著的*Framework Design Guidelines*，248-261页[①]）。而

① 中译本《.NET设计规范》，葛子昂译，人民邮电出版社2006年出版。此处对应译本第232-245页。

直接让该类无法被继承则省去了这些工作，不过同时也限制了该类的使用。

此外，注意到该类的当前实现并不能保证不会重复调用driver的Dispose()方法。实现了IDisposable的类型必须支持多次调用Dispose()。因为T并没有class约束，因此在离开Dispose方法之前，并不能将driver设置为null。（值类型无法设置为null。）

在实际开发中，我们一般可以通过修改泛型类的接口来避免此类设计。例如，将Dispose的调用移到泛型类之外，同时去掉new()约束，让变量在外部创建好之后再传递到泛型类中。

```
public class EngineDriver<T> where T : IEngine
{
    private T driver;
    public EngineDriver(T driver)
    {
        this.driver = driver;
    }

    public void GetThingsDone()
    {
        driver.DoWork();
    }
}
```

当然，先前代码中的注释说过创建T对象的开销很大，而上面的一段代码则忽略了这个考虑。最终如何解决该问题还取决于应用程序设计中的其他各种因素。不过有一点是确定的：无论实例化了哪个泛型类型参数对象，都需要小心这些类型可能实现了IDisposable接口。因此在编写代码时必须足够小心，避免可能出现的资源泄露问题。

有时，你可以通过重构并让泛型类中不再创建实例来解决这个问题。而对于必须要使用到局部变量的时候，则要提供必要的销毁代码。若是泛型类还需要支持对实现了IDisposable的泛型参数进行延迟创建的话，那么将会带来更多的工作，但这也是必不可少的。

条目6：使用委托定义类型参数上的方法约束

乍看上去，C#的约束机制显得有些过于简单：仅支持指定一个基类、接口、类或结构和一个无参数的构造函数。还有很多无法实现，例如无法指定静态方法（包括操作符），也不能指定其他的构造函数。虽然从某一角度看来，语言定义的约束也能够满足所有契约的需要。例如定义IFactory<T>接口支持创建T对象，定义IAdd<T>接口实现T对象之间的相加并使用T中定义的静态的+操作符（或使用其他方法将T对象相加）。不过这显然不是很理想，这会让设计变得晦涩难懂。

接下来就以Add()为例。若你的泛型类型需要T支持Add()方法，那么需要完成若干项任务：首先创建IAdd<T>接口，随后基于该接口开始设计。这似乎没什么问题，不过却给每个想要使用该泛型类的开发人员都添加了新的工作：创建实现IAdd<T>的类，定义IAdd<T>中的方法，最后为泛型类定义指定封闭泛型类。为了调用一个方法，不得不让开发者创建一个符合API签名的类，这显然会让其他开发者觉得困惑不解，加大使用难度。

好在还有更好的方法——指定一个匹配泛型类将要调用的委托的签名。这并不会给泛型类的开发人员带来任何额外的工作，反而却能够为泛型类型的使用者省下大量的时间。

接下来的代码就演示了如何在某个泛型类里的某个方法中将两个T类型的对象相加起来。你甚至不必自行给出委托的定义，System.Func<T1, T2, TOutput>委托正好能够满足所需要的签名。下面的这个泛型方法就将两个对象加了起来。

```
public static class Example
{
    public static T Add<T>(T left, T right,
        Func<T, T, T> AddFunc)
    {
        return AddFunc(left, right);
    }
}
```

在泛型类型需要调用AddFunc()时，开发者可以使用类型推断和lambda表达式来定义将要使用的方法。使用lambda表达式调用Add泛型方法的代码如下。

```
int a = 6;
int b = 7;
int sum = Example.Add(a, b, (x, y) => x + y);
```

C#编译器能够从lambda表达式(等同于匿名委托)中推断出类型和返回值。若你对lambda表达式的语法还不熟悉，也可以使用匿名委托来调用Add()。

```
int sum2 = Example.Add(a,b, delegate(int x, int y)
{
    return x + y;
});
```

无论采用哪种方法，C#编译器均会在包含这段代码的类中创建一个私有静态方法，用来返回两个int整数的和。方法名由编译器生成。编译器同样将创建一个Func<T,T,T>委托对象，并将其指向编译器前面生成的方法。最终，编译器即可将委托传递给Example.Add()方法。

前面使用lambda语法给出了委托的定义，也演示了为何应该用这种方法创建基于委托的接口契约。虽然示例本身显得有些简单，不过其要表达出的想法却非常重要。即若用接口定义约束显得比较笨重，那么可以使用方法签名和委托来满足实际的需要。随后再把该委托的一个实例添加到泛型方法的参数列表中。这样，使用该类的开发者即可用lambda表达式定义该方法，既减少了代码量，又显得更加清晰。使用该类的开发者可以用lambda表达式给出方法的所需功能，且并不需要任何额外的代码来支持这种基于接口的约束。

很多时候，我们需要使用基于委托的契约来创建操作于一系列元素之上的算法。设想需要编写这样一段代码，将来自不同机械探头上的采样组合起来，把两个序列组合成一个单一的、包含一个个点的序列。

表示点的类型如下。

```
public class Point
{
    public double X
    {
        get;
        private set;
    }
    public double Y
    {
        get;
        private set;
    }
    public Point(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

我们从设备中得到的数据为两个List<double>序列。因此需要找到一种方法，能够基于每一对连续的X和Y重复地调用Point(double,double)构造函数。Point是一个不可变类型，因此无法先调用其默认的构造函数，然后再设定其X和Y属性。也无法在约束中为构造函数指定参数。解决的办法就是创建一个接受两个参数、并返回一个Point的委托。这样的委托同样可以在.NET Framework 3.5中找到。

```
delegate TOutput Func<T1, T2, TOutput>(T1 arg1, T2 arg2);
```

在本实例中，T1和T2均为同样的double类型。创建输出序列的泛型方法将如下所示。

```
public static IEnumerable<TOutput> Merge<T1, T2, TOutput>
    (IEnumerable<T1> left, IEnumerable<T2> right,
    Func<T1, T2, TOutput> generator)
{
    IEnumerator<T1> leftSequence = left.GetEnumerator();
    IEnumerator<T2> rightSequence = right.GetEnumerator();
    while (leftSequence.MoveNext() && rightSequence.MoveNext())
```

```
        {
            yield return generator(leftSequence.Current,
                rightSequence.Current);
        }
    }
```

该Merge方法将逐一枚举输入序列中的元素，对于每一对元素，都会调用generator委托，以便返回刚刚创建的Point对象（参见第3章条目19）。从该委托的签名中可以看出，其表示的方法将根据两个不同的输入构造出输出。注意，上述Merge的定义并不需要两个输入为同样的类型。使用同一个方法即可创建出不同类型的键值对，只要传入不同的委托即可。

可以按照如下方法调用Merge。

```
double[] xValues = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
double[] yValues = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

List<Point> values = new List<Point>(
    Utilities.Merge(xValues, yValues,
    (x, y) => new Point(x, y)));
```

这里再次使用了lambda表达式语法，这与匿名委托语法的功能完全相同。

```
List<Point> values2 = new List<Point>(
    Utilities.Merge(xValues, yValues,
    delegate(double x, double y)
    {
        return new Point(x, y);
    }));
```

与前面一样，编译器将生成一个私有静态方法，随后将创建委托对象并指向该方法，最后将该委托对象传递给Merge()方法。

一般情况下，泛型类型中将要调用的所有方法均可由某个委托所代替。前面两个示例就分别包含了一个委托，用来在泛型方法中调用。哪怕是在多个地方需要该委托方法，也可以使用同样的模式。对于创建的泛型类，完全可以将委托作为其类型参数。随后在实例化该类时，即可将委托保留在该类的一个成

员变量中。

接下来的这个简单示例就将这样把一个委托“缓存”起来，随后从流中读取字符，并用该委托将其解析为一个Point对象。首先，我们要为Point类添加一个构造函数，支持从流中读取字符。

```
public Point(System.IO.TextReader reader)
{
    string line = reader.ReadLine();
    string[] fields = line.Split(',');
    if (fields.Length != 2)
        throw new InvalidOperationException(
            "Input format incorrect");
    double value;
    if (!double.TryParse(fields[0], out value))
        throw new InvalidOperationException(
            "Could not parse X value");
    else
        X = value;

    if (!double.TryParse(fields[1], out value))
        throw new InvalidOperationException(
            "Could not parse Y value");
    else
        Y = value;
}
```

创建该集合类需要一些技巧。我们无法用约束来确保泛型类型一定提供了带有参数的构造函数。不过这个工作却可以借助于一个方法来实现。定义一个委托，让其从流中创建一个T对象。

```
public delegate T CreateFromStream<T>(TextReader reader);
```

接下来编写容器类，这个容器类的构造函数接受一个该委托类型的实例作为参数。

```
public class InputCollection<T>
{
    private List<T> thingsRead = new List<T>();
    private readonly CreateFromStream<T> readFunc;
```

```
    public InputCollection(CreateFromStream<T> readFunc)
    {
        this.readFunc = readFunc;
    }

    public void ReadFromStream(TextReader reader)
    {
        thingsRead.Add(readFunc(reader));
    }

    public IEnumerable<T> Values
    {
        get { return thingsRead; }
    }
}
```

在实例化InputCollection时，需要提供该委托。

```
InputCollection<Point> readValues = new
    InputCollection<Point>(
    (inputStream) => new Point(inputStream));
```

这个示例非常简单，你甚至可以用非泛型的版本来实现。不过这种技术则可以让你在构建泛型时，指定一些无法通过普通约束来限制的行为。

通常，最好的设计是使用类约束或接口约束来指定需要的约束。.NET基础类库（Base Class Library，BCL）中有很多现成的例子，例如需要让泛型参数必须实现IComparable<T>、IEquatable<T>或IEnumerable<T>等。这是个非常不错的选择，因为上述接口较为常见，且用于很多算法中。此外，这些接口也极为明确地表达了它们的含义：实现了IComparable<T>的类型表示其对象之间支持排序关系。实现了IEquatable<T>的类型表示其对象之间支持等同性比较。

不过，若你需要仅为某个特定的泛型方法或类创建自定义的接口契约，那么也可以使用委托将该契约声明成方法的约束，这将会让类型的使用者更加方便。这样，你的泛型类型将更易于使用，其调用代码也会易于理解。无论是为了指定操作符、静态方法、委托类型或是某种创建规则，都可以用泛型接口来

表达这种约束，并通过创建实现了这些接口的辅助方法类型来满足约束的条件。不要让任何无法由基本约束直接支持的语义上的约束限制了你的设计。

条目7：不要为基类或接口创建泛型的特殊实现

引入泛型方法将让编译器对重载的解析变得非常复杂。每个泛型方法的类型参数都可以任意替换。如果稍有疏忽，程序的行为将变得极其古怪。在创建泛型类或方法时，必须保证让使用者能够尽可能地理解你的设计意图，安全地使用你的代码。因此需要非常小心对重载的解析，还要了解哪个方法将会更好地匹配开发者的原本意图。

查看如下代码，猜测一下它的输出。

```
public class MyBase
{
}

public interface IMessageWriter
{
    void WriteMessage();
}

public class MyDerived : MyBase, IMessageWriter
{
    #region IMessageWriter Members
    void IMessageWriter.WriteMessage()
    {
        Console.WriteLine("Inside MyDerived.WriteMessage");
    }
    #endregion
}

public class AnotherType : IMessageWriter
{
    #region IMessageWriter Members
    public void WriteMessage()
```

```
    {
        Console.WriteLine("Inside AnotherType.WriteMessage");
    }
    #endregion
}

class Program
{
    static void WriteMessage(MyBase b)
    {
        Console.WriteLine("Inside WriteMessage(MyBase)");
    }

    static void WriteMessage<T>(T obj)
    {
        Console.Write("Inside WriteMessage<T>(T):  ");
        Console.WriteLine(obj.ToString());
    }

    static void WriteMessage(IMessageWriter obj)
    {
        Console.Write(
            "Inside WriteMessage(IMessageWriter):  ");
        obj.WriteMessage();
    }

    static void Main(string[] args)
    {
        MyDerived d = new MyDerived();
        Console.WriteLine("Calling Program.WriteMessage");
        WriteMessage(d);
        Console.WriteLine();

        Console.WriteLine(
            "Calling through IMessageWriter interface");
        WriteMessage((IMessageWriter)d);
        Console.WriteLine();
        Console.WriteLine("Cast to base object");
```

```
            WriteMessage((MyBase)d);
            Console.WriteLine();

            Console.WriteLine("Another Type test:");
            AnotherType anObject = new AnotherType();
            WriteMessage(anObject);
            Console.WriteLine();

            Console.WriteLine("Cast to IMessageWriter:");
            WriteMessage((IMessageWriter)anObject);
        }
    }
```

其中的一些注释可能会对猜出答案有所帮助，不过你应该在查看输出之前尽力去思考一下。我们必须理解泛型方法是如何对方法的解析规则产生影响的。一般来讲，泛型匹配的优先级较高，因此其输出结果可能会让你非常意想不到。如下就是输出结果。

```
Calling Program.WriteMessage
Inside WriteMessage<T>(T):  Item14.MyDerived

Calling through IMessageWriter interface
Inside WriteMessage(IMessageWriter):
    Inside MyDerived.WriteMessage

Cast to base object
Inside WriteMessage(MyBase)

Another Type test:
Inside WriteMessage<T>(T):  Item14.AnotherType

Cast to IMessageWriter:
Inside WriteMessage(IMessageWriter):
    Inside AnotherType.WriteMessage
```

第一个测试体现了一个很重要的概念：对于一个派生于MyBase的对象来说，WriteMessage <T>(T obj)要比WriteMessage(MyBase b)在重载匹配上更加优先。因为通过将T替换成MyDerived，编译器即可完成一个精确的匹配，而WriteMessage(MyBase)则还需要一次隐式转换。于是在这里泛型方法

占得了上风。在接触C# 3.0里Queryable和Enumerable类型中的扩展方法时，你也会看到这个重要概念。即使与接受基类作为参数的方法比较，泛型方法也总会是最好的匹配。

接下来的两个测试演示了如何通过显式调用转换过程（转换至MyBase或IMessageWriter类型）来控制该行为。最后的两个测试说明了即使不在类型继承体系中，对接口的实现也会出现同样的行为。

名称解析的规则非常有趣，甚至能够成为让你在技术人员聚会上出彩的话题。但我们真正的目的是让开发者和编译器对“最佳匹配”的理解趋于一致。这样才不会给使用者造成太多的困惑。

当你想支持某一类及其所有的派生类时，基于基类创建泛型并不是个好的选择。基于接口也是如此。不过数值类型却没有问题，因为整型和浮点型数之间并不存在继承关系。如条目2所述，我们通常都会找到一些原因，为不同的值类型给出特定版本的方法。例如，.NET Framework就为Enumerable.Max<T>和Enumerable.Min<T>等类似方法为所有不同的数值类型均提供了专门的方法。不过最好仍是使用编译器，而不是添加运行时检查来判断类型。这也正是使用泛型的首要目标。

```
// 不是最好的解决方案，使用了运行时类型检查
static void WriteMessage<T>(T obj)
{
    if (obj is MyBase)
        WriteMessage(obj as MyBase);
    else if (obj is IMessageWriter)
        WriteMessage((IMessageWriter)obj);
    else
    {
        Console.Write("Inside WriteMessage<T>(T):  ");
        Console.WriteLine(obj.ToString());
    }
}
```

若仅有少数几个条件需要检查，那么上述代码不会有什么问题。它确实将所有丑陋的行为对类型的使用者隐藏了起来，当然也带来了一些运行时的开销。该泛型方法需要首先检查参数的类型，然后判断是否存在着比编译器的默认分配更好的重载。只有在确保存在着更好的重载时，才可以考虑使用这种技术，同时也需要对其性能进行分析，看看是否还有更好的方法重写该类库并避免问题。

当然，这里并不是说你绝不应该为某种实现创建专门的重载方法。条目3演示了当具体类型还提供了高级功能时，如何用更好的方法实现算法逻辑。此时，条目3中的代码直接创建了一个反向的迭代器。注意，条目3中的代码并没有泛型来进行名称解析。每个构造函数均正确地表达了不同的语义，让其他代码可以随时调用。不过，若想为泛型方法的某个特定类型进行特殊的实例化的话，那么则需要为该类型及其所有的派生类给出必要的代码。若想为某个接口类型创建泛型的特例的话，也需要为实现了该接口的所有类型提供适当的特例实现。

条目8：尽可能使用泛型方法，除非需要将类型参数用于实例的字段中

我们会很容易地被泛型类型的定义限制住。而一般情况下，只需在非泛型类型中提供几个泛型方法就足够表达设计者的意图了。之所以这样做，是因为C#编译器必须根据给出的约束为整个泛型类生成合法的IL。且给出的约束也必须满足整个类的需要，而包含了泛型方法的辅助类则可以为每个方法指定不同的约束。有了这些不同的约束，编译器即可更容易地找到合适的重载，类型的使用者也会觉得更加清晰。

此外，类型参数仅仅需要满足将要用到的泛型方法的约束即可。而对于泛型类，类型参数则必须满足定义在整个类上的所有约束。在随后对该类进行扩展时，你会发现定义在类级别的约束将会让你越来越无法放开手脚。这里可以给出一个简单的规则：如果你需要类型级别的数据成员，特别是涉及类型参数的成员变量，那么则使用泛型类。对于一切其他情况，均使用泛型方法。

我们来看一个简单的示例，其中包含泛型的Min和Max方法。

```
public static class Utils<T>
{
    public static T Max(T left, T right)
    {
        return Comparer<T>.Default.Compare(left, right) < 0 ?
            right : left;
    }

    public static T Min(T left, T right)
    {
        return Comparer<T>.Default.Compare(left, right) < 0 ?
            left : right;
    }
}
```

第一眼看上去似乎没什么问题。可以比较数字：

```
double d1 = 4;
double d2 = 5;
double max = Utils<double>.Max(d1, d2);
```

也可以比较字符串：

```
string foo = "foo";
string bar = "bar";
string sMax = Utils<string>.Max(foo, bar);
```

嗯，似乎不错，可以回家了。不过这个类的使用者却并不觉得那么顺手。前面的代码中，每次调用均需要显式声明类型参数。这是因为使用了泛型类，而不是泛型方法。除了这个麻烦之外，还有一些深层次的问题。很多内建类型已经提供了Max和Min方法，例如Math.Max()和Math.Min()就支持所有的数值类型。而泛型类的实现则总是调用Comparer<T>的实现。虽然结果上没什么问题，不过却导致程序必须在运行时检查类型是否实现了IComparer<T>，随后再调用合适的方法。

很自然，我们希望让程序能够自动地选择最好的方法。而若是通过在非泛型类中提供泛型方法来实现，那么将会容易很多。

```
public static class Utils
{
    public static T Max<T>(T left, T right)
    {
        return Comparer<T>.Default.Compare(left, right) < 0 ?
            right : left;
    }

    public static double Max(double left, double right)
    {
        return Math.Max(left, right);
    }
    // versions for other numeric types elided

    public static T Min<T>(T left, T right)
    {
        return Comparer<T>.Default.Compare(left, right) < 0 ?
            left : right;
    }
    public static double Min(double left, double right)
    {
        return Math.Min(left, right);
    }
    // versions for other numeric types elided
}
```

这样，Utils不再是泛型类了。而是提供了几个Min和Max的重载。这些特定的方法要比泛型版本更加高效（参见条目3）。此外，使用这也无需显式指定类型参数。

```
double d1 = 4;
double d2 = 5;
double max = Utils.Max(d1, d2);

string foo = "foo";
string bar = "bar";
string sMax = Utils.Max(foo, bar);
```

```
double? d3 = 12;
double? d4 = null;
double? Max2 = Utils.Max(d3, d4).Value;
```

若是能够找到与类型参数匹配的重载，那么编译器将调用该重载。否则将调用泛型的版本。此外，若稍后又对Uitls类进行了扩展，为更多类型添加了重载，那么编译器也能够立即找到。

并不是仅有静态的辅助类才应该使用泛型方法而不是泛型类。比如下面的这个简单类，用来构建一个由逗号分割的条目列表。

```
public class CommaSeparatedListBuilder
{
    private StringBuilder storage = new StringBuilder();

    public void Add<T>(IEnumerable<T> items)
    {
        foreach (T item in items)
        {
            if (storage.Length > 0)
                storage.Append(", ");
            storage.Append("\"");
            storage.Append(item.ToString());
            storage.Append("\"");
        }
    }

    public override string ToString()
    {
        return storage.ToString();
    }
}
```

可以看到，上述类允许程序向列表中添加各种不同的类型。每次使用新类型时，编译器都会生成一个新的Add<T>版本。而若是将类型参数应用到类声明之上，那么每一个CommaSeparatedListBuilder都只能支持一个类型。每种做法都是合法的，不过其语意却截然不同。

这个例子非常简单，我们完全可以将类型参数用System.Object代替。不过其要表达的概念却值得我们学习：通过在非泛型类中创建一个总的泛型方法来支持若干个不同的特定方法。类本身并没有在其私有字段中使用到T，而仅在其公共API中将T作为了参数。在方法参数中使用不同的类型并不需要重新创建一个新的类型。

显然，并不是所有的泛型算法都支持用泛型方法代替泛型类。不过其中确有一些简单的规范可以遵循。如下的两种情况中必须使用泛型类。第一，类本身需要存放类型参数对象作为其内部状态。（例如集合类型。）第二，类实现了泛型接口。除了这两种情况，其他时候我们大都可以使用非泛型类，并在其中提供泛型方法。这样也会为稍后更新算法留有更大的空间。

再次查看前面的示例。可以看到第二个Utils类并不需要调用者显式指定泛型方法的类型参数。有可能的话，我们应该尽量使用这种方法来提供API，原因有如下几种。首先，其简化了调用过程。在不指定类型参数时，编译器将自动选择最合适的方法。这样也给作为类库开发者的你更多的改进空间。其次，若稍后你针对某种特定的类型提供了更好的实现，那么调用者也将自动调用这个新的实现。而若是这些方法需要调用者显式指定类型参数，那么哪怕你提供了更好的实现，程序仍旧会调用原始的泛型版本。

条目9：使用泛型元组代替out和ref参数

很多开发者都会遇到的一个常见问题就是，如何为逻辑上返回多个项目的方法设计签名。有些人会使用ref或out参数。不过定义泛型元组来返回多个不相关值则是个更好的做法。这里，元组（tuple）的概念就是*n*个元素的组合。

之所以说这是个更好的方法，有如下几个原因。首先考虑不可变性（immutability）。若是使用ref参数，那么将很难创建不可变的对象。

下面的这个类就将员工（Employee）表示为不可变类型。

```
public class Employee
{
    private readonly string firstName;
    private readonly string lastName;
    private readonly decimal salary;
    public string FirstName
    {
        get { return firstName; }
    }
    public string LastName
    {
        get { return lastName; }
    }
    public decimal Salary
    {
        get { return salary; }
    }
    public Employee(string firstName, string lastName,
        decimal salary)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.salary = salary;
    }
    public override string ToString()
    {
        return string.Format("{1}, {0} salary: {2}",
            lastName, firstName, salary);
    }
}
```

我们可以从控制台中读取员工的信息，并创建员工对象。

```
string last = Console.ReadLine();
string first = Console.ReadLine();
string salary = Console.ReadLine();
Employee emp = new Employee(first, last,
    decimal.Parse(salary));
```

很自然地，若是输入的工资（salary）参数不符合格式，那么decimal.Parse将抛出异常。这时则需要使用TryParse。

```
string last = Console.ReadLine();
string first = Console.ReadLine();
string salaryString  = Console.ReadLine();
decimal salary = 0;
bool okSalary = decimal.TryParse(salaryString, out salary);
if (okSalary)
{
    Employee emp = new Employee(first, last, salary);
    Console.WriteLine(emp);
}
```

即使对于这个最简单的例子，也需要添加不少的代码。若是程序的逻辑再复杂一些，那么问题将变得更加严重。

此外，ref和out也让程序难以实现多态。考虑如下存在继承关系的Person和Employee定义。

```
public class Person
{
    private readonly string firstName;
    private readonly string lastName;
    public string FirstName
    {
        get { return firstName; }
    }
    public string LastName
    {
        get { return lastName; }
    }
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
public class Employee : Person
{
    private readonly decimal salary;
    public decimal Salary
    {
```

```
        get { return salary; }
    }
    public Employee(string firstName, string lastName,
        decimal salary) :
        base (firstName, lastName)
    {
        this.salary = salary;
    }
    public override string ToString()
    {
        return string.Format("{1}, {0} salary: {2}",
            LastName, FirstName, salary);
    }
}
```

随后，有人添加了一个新的方法，用来修改个人的姓氏。

```
static void ChangeName(ref Person p, string newLastName)
{
    p = new Person(p.FirstName, newLastName);
}
```

ChangeName()无法接受Employee对象，因为该代码无法执行成功。该方法将把一个Employee对象（或其他派生类型的对象）转换成一个Person对象，这个过程中将会丢失信息。当然，你也不能只为Employee提供一个方法，因为这个方法无法接收其基类。使用ref参数将意味着你必须为每一个希望支持的类型均提供一个重载。因为被ref修饰参数不支持隐式地类型转换。在使用ref时，无法使用派生类（或基类）来替换所需的基类参数。而方法的返回值却支持这种隐式类型转换，因为方法可以返回其签名中声明的类型的派生类型对象。借助于一些简单的局部变量类型推断，开发者可以很容易地使用到继承体系中的各种类型。

其效果非常类似于条目47（见第6章）中描述的数组参数问题。唯一的不同是在ref参数的问题里，编译器对类型的检查更加严格了一些。

前面的两个示例均只有一个ref参数，且没有返回值。对于这种情况，我们可以很容易地修改方法的签名，让其返回适当的类型。

```
static Person ChangeName(Person p, string newLastName)
    { // 省略}
static Employee ChangeName(Employee p, string newLastName)
    { // 省略}
```

若是有两个逻辑上的返回值，那么问题将变得复杂起来。在这种情况下，我们可以用泛型定义一个元组，一次性返回包含所有需要支持的字段。下面就是一个支持两个返回值的泛型元组定义。

```
public struct Tuple<T1, T2> : IEquatable<Tuple<T1, T2>>
{
    private readonly T1 first;
    public T1 First
    {
       get { return first; }
    }

    private readonly T2 second;
    public T2 Second
    {
        get { return second; }
    }

    public Tuple(T1 f, T2 s)
    {
        first = f;
        second = s;
    }
    // IEquatable<Tuple<T1, T2>>的实现省略
}
```

可以看到，上述类的定义非常类似于System.Collections.Generic.KeyValuePair泛型类型。不过其本意却完全不同，因此这里使用了一个新的类型。此外，这个类是一个很典型的元组：包含了两个元素。你也可以继续扩展该技术，让其支持多个字段。

该元组可以用于任意需要返回两个元素的方法，你会自然地想到一个扩展，即让其支持更多数量的元素。借助于使用该Tuple类，即可避免使用ref和out参数来实现多个逻辑方法。这样也让客户端代码更加简洁。

例如，下面的这个方法演示了使用该元组结构返回最近的城市及其温度信息。

```
public static Tuple<string, decimal> FindTempForNearestCity
    (string soughtCity)
{
    string city = "algorithmElided";
    decimal temp = decimal.MinValue; // 真的好冷
    return new Tuple<string, decimal>(city, temp);
}
```

调用该方法之前，也需要准备一个同样的结构。

```
Tuple<string, decimal> weather =
    FindTempForNearestCity("NearHere");
```

可能你并不希望此类Tuple的定义扰乱了类型的真正含义，我也不希望这样。好在C#语言的设计者也考虑到了这个问题，允许我们使用using语句为任意的封闭泛型类型声明一个别名。

```
using CityTemperature = Tuple<string, decimal>;
```

现在，该方法的签名显得清晰多了。

```
public static CityTemperature FindTempForNearestCity
    (string soughtCity)
{
    string city = "algorithmElided";
    decimal temp = decimal.MinValue; // 真的好冷
    return new CityTemperature(city, temp);
}
```

方法调用以及赋值语句也显得更加清晰。

```
CityTemperature weather = FindTempForNearestCity("NearHere");
```

也许你会觉得这只不过是重新调整一下方法的签名而已。不过当你逐渐熟悉C# 3.0，并开始使用其中引入的函数式编程结构时，就会发现这个技术的重要性。接受out和ref参数的方法无法与其他方法良好地组合在一起。而返回单一值（无论多复杂）的方法则能很好地进行组合。

ref和out参数表明方法中可以创建出符合声明类型的对象。多态表明派生类总是可以替代基类。不过，当方法中需要创建出返回值对象时，上述说法还包含了一些新的含义。当调用代码期待基类对象时，很多情况下方法中都会创建出派生类型对象。总的说来，上述两条规则意味着，out和ref参数无法真正地使用到多态。若是改为使用泛型元组返回多个值，那么你的算法逻辑势必能够更易于使用。

条目 10：在实现泛型接口的同时也实现传统接口

到目前为止，本章中的条目均介绍了泛型所带来的巨大优势。若是能够忽略掉.NET和C#在支持泛型之前的所有历史，那该有多好啊！不过开发人员的工作却不仅仅如此简单，原因有很多。若是你的类库能够在支持泛型接口的同时，也对传统的非泛型接口提供支持，那么显然将变得更加有用。这个建议可以应用到三个地方：第一，类及其支持的接口；第二，公共属性；第三，需要序列化的元素。

我们先来看看为什么要继续支持那些非泛型接口。随后研究一下如何在支持那些传统接口的同时，也能鼓励用户使用最新的泛型版本。我们以一个Name类为例，该类用来在应用程序中存放人们的姓名。

```
public class Name :
    IComparable<Name>,
    IEquatable<Name>
{
    public string First
    {
        get;
        set;
    }

    public string Last
    {
        get;
```

```
        set;
    }
    public string Middle
    {
        get;
        set;
    }

    #region IComparable<Name> Members
    public int CompareTo(Name other)
    {
        if (other == null)
            return 1; // Any non-null object > null.
        int rVal = Comparer<string>.Default.Compare
            (Last, other.Last);
        if (rVal != 0)
            return rVal;
        rVal = Comparer<string>.Default.Compare
            (First, other.First);
        if (rVal != 0)
            return rVal;
        return Comparer<string>.Default.Compare(Middle,
            other.Middle);
    }
    #endregion

    #region IEquatable<Name> Members
    public bool Equals(Name other)
    {
        if (Object.ReferenceEquals(other, null))
            return false;
        // 语义上等同于使用EqualityComparer<string>.Default
        return Last == other.Last &&
            First == other.First &&
            Middle == other.Middle;
    }
    #endregion

    // 省略其他细节
}
```

所有核心的等同性和排序比较功能都使用泛型(类型安全)的版本实现的。此外，可以看到代码中将空值检查从CompareTo()延迟到了默认的字符串比较器中。这样节约了代码的同时也提供了同样的语义。

不过这个泛型实现却无法与.NET 1.x的代码配合使用。此外，你还可能需要将来自于不同系统却表示同一种概念的不同类型集成起来。假设你从一家厂商中购买了一套电子商务系统，又从另外一家厂商购买了一套配送系统。两个系统都有订单的概念：Store.Order和Shipping.Order。这难免需要你创建出一套等同性比较的逻辑。泛型无法很好地支持这个需求。要进行的是不同类型之间的比较。此外，你还可能需要将两种不同的Order对象存放至同一个集合中。泛型类型同样无法支持。

这时，你需要一个使用System.Object来检查对象等同性的方法，其代码可能如下所示。

```
public static bool CheckEquality(object left, object right)
{
    if (left == null)
        return right == null;
    return left.Equals(right);
}
```

用两个person对象调用CheckEquality()方法将得到预料之外的结果。CheckEquality()并不会调用IEquatable<Name>.Equals()，而是会调用System.Object.Equals()!这样就会导致错误，因为System.Object.Equals()使用的是引用的比较，而重写的IEquatable<T>. Equals比较的则是值。

如果可以修改CheckEquality()的话，那么可以为其创建一个泛型版本，让其调用正确的方法。

```
public static bool CheckEquality<T>(T left, T right)
    where T : IEquatable<T>
{
    if (left == null)
        return right == null;

    return left.Equals(right);
}
```

当然，若CheckEquality()不在你的控制当中，例如在第三方类库，甚至.NET BCL中，那么则无法使用这种解决方案。此时必须重写传统的Equals方法，并在其中调用IEquatable<T>.Equals方法。

```
public override bool Equals(object obj)
{
    if (obj.GetType() == typeof(Name))
        return this.Equals(obj as Name);
    else return false;
}
```

修改之后，几乎所有的检查Name类型等同性的方法都能正常使用了。注意，在将obj参数转换为Name对象之前，代码首先检查了obj的类型。也许你会觉得这一条检查是多余的，因为若obj无法转换为Name类型的话，as操作符将返回null。这个假设没有考虑到一些条件：as操作符将调用用户自定义的转换，但这却并不是你期待的行为。此外，若obj为Name的派生类，那么as操作符将返回一个Name对象的指针。那么，即使两个对象中属于Name类型的那部分相同，其本质上也并不相等。

接下来，重写了Equals则意味着同时应该重写GetHashCode。

```
public override int GetHashCode()
{
    int hashCode = 0;
    if (Last != null)
        hashCode ^= Last.GetHashCode();
    if (First != null)
        hashCode ^= First.GetHashCode();
    if (Middle != null)
        hashCode ^= Middle.GetHashCode();
    return hashCode;
}
```

通过这样对公共API的扩展之后，类型即可与1.x代码配合使用了。

若是希望完整地考虑到方方面面，那么还需要处理几个操作符。实现IEquality<T>意味着同时实现了==和!=操作符。

```
public static bool operator ==(Name left, Name right)
{
    if (left == null)
        return right == null;
    return left.Equals(right);
}
public static bool operator !=(Name left, Name right)
{
    if (left == null)
        return right != null;
    return !left.Equals(right);
}
```

对于等同性检查而言，这就足够了。不过Name类还实现了IComparable<T>接口，你在实践中也会遇到等同性比较中同样的情况。实现传统的IComparable接口需要很多代码。而既然已经写好了这部分算法，那么不如直接添加IComparable接口的实现，随后再编写其他的方法。

```
public class Name :
    IComparable<Name>,
    IEquatable<Name>,
    IComparable
{
    #region IComparable Members
    int IComparable.CompareTo(object obj)
    {
        if (obj.GetType() != typeof(Name))
            throw new ArgumentException(
                "Argument is not a Name object");
        return this.CompareTo(obj as Name);
    }
    #endregion
    // 其他细节省略
}
```

注意，传统的接口使用了显式接口实现。这样即可保证使用者不会因为不小心调用到了这个版本，确保使用者尽可能使用推荐的泛型版本。在通常使用中，编译器将会选择泛型方法而不是显式的接口方法。仅当调用方法将对象转换成传统接口的类型（IComparable）时，编译器才会调用该接口中的成员。

当然，实现了IComparable<T>则表示该对象之间支持排序关系，因此同样需要实现小于（<）和大于（>）两个操作符。

```
public static bool operator <(Name left, Name right)
{
    if (left == null)
        return right != null;
    return left.CompareTo(right) < 0;
}
public static bool operator >(Name left, Name right)
{
    if (left == null)
        return false;
    return left.CompareTo(right) < 0;
}
```

对于该Name类型而言，因为其同时定义了排序关系和等同关系，那么还应该实现<=和>=操作符。

```
public static bool operator <=(Name left, Name right)
{
    if (left == null)
        return true;
    return left.CompareTo(right) <= 0;
}
public static bool operator >=(Name left, Name right)
{
    if (left == null)
        return right == null;
    return left.CompareTo(right) >= 0;
}
```

需要理解的是，排序关系和等同关系二者并不相关。你可以定义实现了等同性比较的对象，但没有定义排序关系。也可以定义实现了排序关系的对象，但不支持等同性比较。

前面的代码均或多或少地实现了Equatable<T和Comparer<T>的语义。这些类型中的Default属性均检查了类型参数T是否实现了针对该类型的等同性或排序关系比较方法。若确有实现，那么将调用这些方法。否则将使用System.Object中的默认版本。

这里通过等同性和排序关系来演示了新旧接口风格之间的不兼容情况。这类不兼容性也会体现在其他地方。IEnumerable<T>继承于IEnumerable。不过带有完整功能的集合接口却没有这样的关系，即ICollection<T>并没有继承于ICollection，IList<T>也没有继承于IList。不过由于IList<T>和ICollection<T>均继承于IEnumerable<T>，所以两个接口也都可以支持传统的IEnumerable。

大多数情况下，添加对传统接口的支持只不过是为类添加带有合适签名的方法而已。正如我们在IComparable<T>和IComparable中看到的那样，你应该以显式的方式来实现IComparable传统接口，以便鼓励用户使用新版本的代码。Visual Studio等工具也为创建这些接口方法提供了向导支持。

.NET Framework 2.0种提供了很多新的接口和类型。这些接口和类型均在很大程度上提高了应用程序和类库中的类型安全性。不过在你使用它们的同时，也不要忘记并不是世界上的所有人也都在使用它们。虽然可以通过显式接口实现来避免可能出现的误用，但也应该尽力支持那些有着同样功能的传统接口。

第2章

C#中的多线程

摩尔定律已经改变了。计算机仍然在不断变快，不过所倚赖的不再是时钟频率，而是更多的处理器核心。大势所趋，开发者也愈发频繁地处理多线程编程的任务。

多线程编程更加困难，且很容易出错。很多难以察觉的bug都出现在线程的切换过程中。除非仔细检查程序中的每一行代码，并分析线程切换时的各种情况，否则很容易引入潜在的问题。也许有一天，线程的切换发生在了你测试时没有覆盖到的地方，从而导致了程序的崩溃。这样，编写正确的程序变得更加困难，验证程序正确性的难度也大大提高。因此，虽然多线程程序更为流行了，但开发难度仍旧要比单线程程序大得多。

本章并不会让你成为多线程编程的专家，不过其中给出的条目却都是.NET多线程程序设计中的一些常见的建议和规则。若想完整地学习多线程技术的方方面面，我推荐阅读Joe Duffy的*Concurrent Programming on Windows Vista: Architecture, Principles, and Patterns*（Addison-Wesley, 2008）。在取得以上共识后，下面我们先来看看多线程编程相对于单线程编程所增加的挑战。

简单地将单线程程序转为并行执行将导致很多问题。我们来看一个简单的银行账户的定义。

```
public class BankAccount
{
    public string AccountNumber
    {
        get;
```

```
        private set;
    }

    public decimal Balance
    {
        get;
        private set;
    }

    public BankAccount(string accountNumber)
    {
        AccountNumber = accountNumber;
    }

    public void MakeDeposit(decimal amount)
    {
        Balance += amount;
    }
    public decimal MakeWithdrawal(decimal amount)
    {
        if (Balance > amount)
        {
            Balance -= amount;
            return amount;
        }
        return 0M;
    }
}
```

如此简单的一段代码，无需过多检查即可保证其正确性。不过在多线程环境中却并非如此。为什么呢？因为上述代码中包含了很多潜在的竞争条件。存款和取款的方法实际上均由几个不同的操作组成。+=操作符首先将当前的账户余额从内存中读取出来并保存于寄存器中。随后CPU将执行相加操作。随后，新的余额才会被写回至内存中。

问题在于，在多核心的处理器上，应用程序中的多个线程可能同时运行于多个核心上。这样，不同的线程就有可能交替地对同一块内存地址进行读写，进而造成数据错误。考虑如下的场景。

(1) 线程A开始存入10 000美元的操作。

(2) 线程A获取到当前的余额为2 000美元。

(3) 线程B开始存入4 000美元的操作。

(4) 线程B获取到当前的余额为2 000美元。

(5) 线程B计算出新的余额为6 000美元。

(6) 线程A计算出新的余额为12 000美元。

(7) 线程A将余额12 000美元保存。

(8) 线程B将余额6 000美元保存。

这样，这种线程之间的交替操作就导致了从前单线程情况下不会发生的错误。

之所以会出现这样的竞争条件，是因为该类中没有为可能引起副作用的操作提供任何同步机制。Deposit()和Withdrawal()方法都会产生副作用：它们均修改了当前的状态，并返回新的值。这两个方法依赖于调用方法时系统的当前状态。例如，若是账户中的余额不足，那么取款操作将失败。没有副作用的方法一般均不需要过多的同步。此类方法不依赖于当前状态，因此在方法执行过程中，即使当前状态发生了改变，也不会影响到其执行结果。

修复这个问题非常简单，只需添加一些锁定即可。

```
public void MakeDeposit(decimal amount)
{
    lock (syncHandle)
    {
        Balance += amount;
    }
}
public decimal MakeWithdrawal(decimal amount)
{
    lock (syncHandle)
    {
        if (Balance > amount)
        {
            Balance -= amount;
            return amount;
        }
    }
    return 0M;
}
```

这样似乎解决了问题，不过却可能导致死锁。例如，若某个客户拥有两个银行账户，一个储蓄账户，一个支票账户。客户可能需要进行转账操作——从一个账户中提款，并存入另一个账户中。逻辑上，这是个单一的操作。不过在实现上，它却包含了一系列的操作。首先从一个账户中取款（这本身也是个多步的操作），随后存入另一个账户（也是个多步操作）中。这似乎不会有什么问题：在从一个账户中取款时锁定该账户，然后获取第二个账户的锁定，并执行存款操作。

不过若是多个线程在同时执行转账操作，那么就可能发生死锁。死锁将会发生的情况是，两个线程互相持有另一个线程完成工作所需要的锁。这样，无论等待多长时间，情况都不会有所转变。应用程序看起来就像是崩溃了。实际上并没有崩溃，不过却在等待着一些永远都不会发生的事情。

另一种比死锁略微好一点的情况是活锁。活锁涉及一种较为复杂的锁定机制，这种机制把对同一块数据的读取和写入区分对待。这种机制允许让多个读取线程同时检查一块数据，不过同时仅允许一个写入线程修改该数据。此外，当某一写入线程在修改数据时，也不允许读取线程读取该数据。活锁将发生于这样的情况下：不停有读取线程在检查某块数据，而让写入线程无法插入到其中。这样，该数据实际上就变为了只读的。

没有什么好的方法可以避免此类问题。多线程编程本身就很复杂，所有操作的复杂性也有了很大的提高。不过多线程编程前景广阔，因此每个C#开发人员至少都需要对多线程技术有一些基本的了解。

.NET Framework在很多地方都使用到了多线程。例如在Web应用程序和Web服务中，每个请求都由一个专门的ASP.NET工作线程负责。remoting类库也用同样的方式处理每个请求。有些计时器的事件处理程序运行于新的线程之上。WCF（Windows Communication Foundation）类库也使用了多个线程。你还可以在调用Web服务时采用异步的方式。

你总会用到上述这些技术。因此若想对.NET有更深入的理解，那么必须了解一些多线程技术。

条目 11：使用线程池而不是创建线程

你无法知道应用程序中需要的线程的最佳数量。你的应用程序可能运行于拥有多个核心的计算机上，不过无论你今天假设将会有多少个核，6个月后都十有八九会不正确。此外，你也无法控制CLR为完成其自身工作（例如垃圾收集器等）所创建的线程的数量。在服务器应用程序中，例如ASP.NET或WCF服务，每个请求都由一个不同的线程处理。作为应用程序或类库开发者的我们，将很难优化目标系统上的线程数量。但.NET线程池却能够获取到所有的必要信息，来优化指定系统上活动线程的数量。此外，即使你创建了过多的任务或线程，那么线程池也能用队列把无法及时处理的请求保存起来，直至有线程释放出来。

.NET线程池能够替你完成很多线程资源的管理工作。当应用程序开始执行重复的后台任务，且并不需要经常与这些任务交互时，使用.NET线程池管理这些资源将会让程序的性能更佳。

调用`QueueUserWorkItem`方法即可让线程池来为你管理资源。在其中添加项目之后，该项目将在有空余线程时得以执行。根据正在运行的任务的数量和线程池的大小，项目可能会立即执行，或等待直至有空余的线程出现。线程池由每个处理器中一定数量的就绪线程和一系列I/O读取线程组成。具体的数字因版本的不同而不同。在开始向队列中插入新的任务时，线程池可能会创建更多的线程，这也取决于当前内存以及其他资源的可用情况。

这里并没有仔细解释线程池的实现，因为线程池本身是用来减轻我们的工作，并让框架去帮我们分担的。简而言之，线程池中的线程数量将在可用线程数量和最小化已分配但尚未使用的资源之间自动平衡。在添加了工作项之后，若当前还有可用线程，那么就会开始执行。线程池的工作是保证尽可能快地提供出可用线程。不过对于你来说，只要发起请求，就不必再担心其内部处理机制了。

线程池同样也会自动管理任务结束后的维护工作。当任务结束之后，线程并不会被销毁，而是返回到可用状态，以便执行其他任务。稍后，该线程可能会被线程池分配去处理另外的任务。接下来的任务并不一定和前面的任务一样，可能为应用程序需要的任何一个较耗时的方法。只需调用QueueUserWorkItem并传入需要的方法，线程池将自动为你管理好所有的一切。

线程池还能够用另一种方法帮你管理运行于其他线程中的任务。所有QueueUserWorkItem使用的线程池中的线程均为后台线程（background thread）。这也就意味着你并不需要在应用程序退出之前手工清理资源。若是应用程序在这些后台线程仍在运行时就退出了，那么系统将停止这些后台任务，并释放所有与应用程序相关的资源。你只需要在退出应用程序之前确保停止了所有非后台线程即可。不过若没有做到这一点，那么应用程序很可能在不做任何事情的时候还占用着资源。

从另一个角度考虑，因为后台线程将在没有任何警告的情况下被终止，因此需要小心其中对系统资源的访问方式，免得当应用程序终止时，让系统停留在不稳定的状态中。很多情况下，当线程终止时，运行时将在该线程上抛出ThreadAbortException异常。而若是应用程序在尚有后台线程运行时突然终止，那么后台线程将不会收到任何通知，而只是被立即终止。因此，若某个线程可能会让系统资源处于不稳定状态中，那么则不要使用后台线程。好在这种情况并不多见。

系统将管理线程池中活动线程的数量。线程池将根据当前可用的系统资源数量来适当地执行任务。若系统负荷已经接近极限，那么线程池将暂停开启新任务。而若是系统并不繁忙，那么线程池将立即开始新任务。我们无需手工编写负载均衡的逻辑，线程池将为你做好这一切。

或许你会认为，同时执行的任务的最佳数量就等于计算机上的CPU内核数量。这虽然不算是个最差的策略，不过却过于简单了一些，基本上也不会是最好的答案。等待时间、对除了CPU之外其他资源的争夺以及其他你无法控制的

线程均会影响到应用程序中最佳的线程数量。若创建的线程过少，那么将无法完全利用到系统资源，白白浪费了计算能力。而若是线程数量过多，那么计算机将花费过多的时间在线程调度上，进而影响到线程的实际执行时间，反倒降低了整体效率。

为了给出一些通用的原则，我编写了一个使用“亚历山大港的希罗”（Hero of Alexandria）提出的海伦算法计算平方根的小程序。这些原则均比较通用，因为每种算法都有它自己的特性。在这个程序中，核心算法比较简单，且并不需要与其他线程进行交互。

算法的开始将作一次猜测，取得某个数字的平方根，比如猜测1。若想找到下一个近似值，只需计算出当前猜测和原输入与当前猜测的商的平均值。例如，若想得到10的平方根，首先猜测为1，接下来的猜测为[1 +(10/1)]/2，即5.5。重复上述步骤，直至得到符合条件的正确值即可。其代码如下。

```
public static class Hero
{
    private const double TOLERANCE = 1.0E-8;
    public static double FindRoot(double number)
    {
        double guess = 1;
        double error = Math.Abs(guess * guess - number);

        while (error > TOLERANCE)
        {
            guess = (number / guess + guess) / 2.0;
            error = Math.Abs(guess * guess - number);
        }
        return guess;
    }
}
```

为了比较线程池、手工创建线程和单线程三种情况下应用程序的性能，这里我还给出了相应的测试程序，其中将重复多次进行该计算。

```
private static double OneThread()
{
```

```
        Stopwatch start = new Stopwatch();
        start.Start();
        for (int i = LowerBound; i < UpperBound; i++)
        {
            double answer = Hero.FindRoot(i);
        }
        start.Stop();
        return start.ElapsedMilliseconds;
    }

    private static double ThreadPoolThreads(int numThreads)
    {
        Stopwatch start = new Stopwatch();
        using (AutoResetEvent e = new AutoResetEvent(false))
        {
            int workerThreads = numThreads;

            start.Start();
            for (int thread = 0; thread < numThreads; thread++ )
                System.Threading.ThreadPool.QueueUserWorkItem(
                    (x) =>
                    {
                        for (int i = LowerBound;
                            i < UpperBound; i++)
                        {
                            // 进行计算
                            if (i % numThreads == thread)
                            {
                                double answer = Hero.FindRoot(i);
                            }

                        }

                        // 减少计数器的值
                        if (Interlocked.Decrement(
                            ref workerThreads) == 0)
                        {
                            // 设置事件
                            e.Set();
                        }
                    });
```

```
            // 等待信号
            e.WaitOne();

            // 跳出
            start.Stop();
            return start.ElapsedMilliseconds;
        }
    }

    private static double ManualThreads(int numThreads)
    {
        Stopwatch start = new Stopwatch();
        using (AutoResetEvent e = new AutoResetEvent(false))
        {
            int workerThreads = numThreads;

            start.Start();
            for (int thread = 0; thread < numThreads; thread++)
            {
                System.Threading.Thread t = new Thread(
                    () =>
                    {
                        for (int i = LowerBound;
                            i < UpperBound; i++)
                        {
                            // 进行计算
                            if (i % numThreads == thread)
                            {
                                double answer = Hero.FindRoot(i);
                            }
                        }
                        // 减少计数器的值
                        if (Interlocked.Decrement(
                            ref workerThreads) == 0)
                        {
                            // 设置事件
                            e.Set();
```

```
                    }
                });
            t.Start();
        }
        // 等待信号
        e.WaitOne();

        // 跳出
        start.Stop();
        return start.ElapsedMilliseconds;
    }
}
```

单线程版本非常简单。而两个多线程版本均使用了lambda表达式语法（参考第1章条目6）来定义后台线程将要执行的操作。当然，如条目6中所述，也可以将lambda表达式替换成匿名委托。

```
System.Threading.ThreadPool.QueueUserWorkItem(
    delegate(object x)
    {
        for (int i = LowerBound; i < UpperBound; i++)
        {
            // 进行计算
            if (i % numThreads == thread)
            {
                double answer = Hero.FindRoot(i);
            }

        }

        // 减少计数器的值
        if (Interlocked.Decrement(
            ref workerThreads) == 0)
        {
            // 设置事件
            e.Set();
        }
    });
```

若使用显式的方法并显式创建委托的话，需要增加很多代码。因为外部方法中定义的很多局部变量（重置事件、线程数和当前线程的索引）都会被用于内部的后台线程中。而C#编译器则会自动为使用了lambda表达式的内联方法创建一个闭包（参见第4章条目33和第5章条目41），省去了我们的工作。此外，注意labmda表达式语法也可以用于多语句的方法，而不仅限于单一的表达式中。

主程序为三个版本的算法均统计了时间，这样可以看到使用线程给算法带来的影响。图2-1即为统计结果。从该示例中，我们可以学到一些东西。首先，与线程池线程相比，手工创建线程的开销更大。若创建了10个以上的线程，那么过多的线程将成为最主要的性能瓶颈。即使在这个并不需要太多等待时间的算法中，其影响也显而易见。

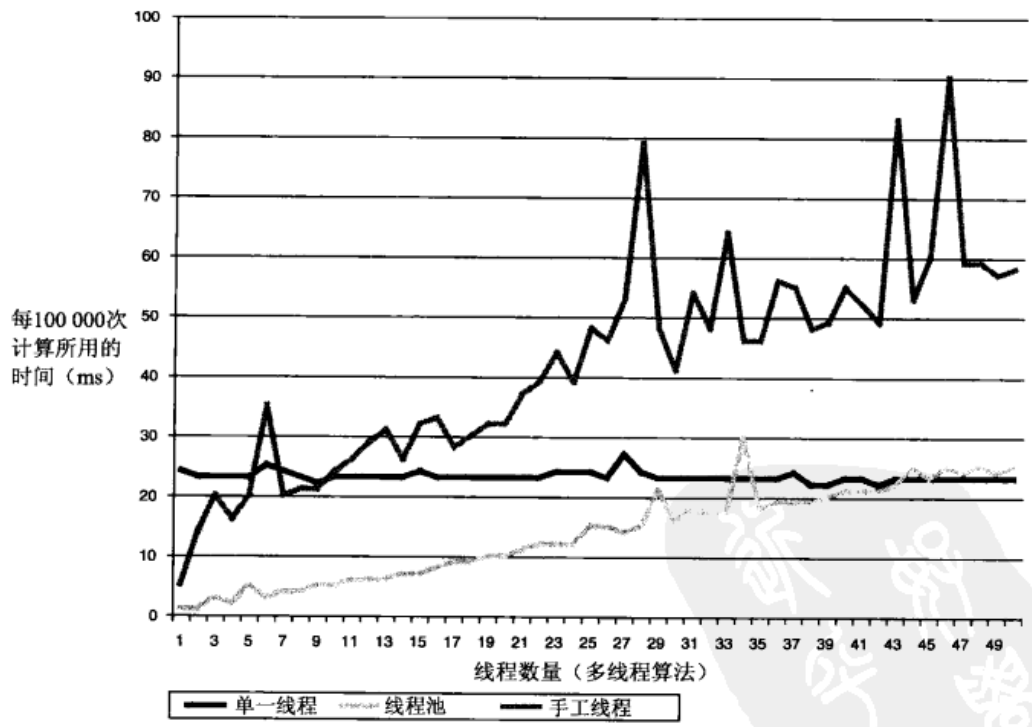

图2-1 单线程、使用System.Threading.Thread和使用System.Threading.ThreadPool.QueueUserWorkItem程序的计算时间结果。*y*轴为在一台双核笔记本电脑上执行100 000次计算所花费的时间（单位为ms）

若使用线程池，那么必须添加40个以上的项目，才会看到额外开销逐渐占据了支配地位。而这只是在一台双核的笔记本电脑上而已。对于那些服务器级别的计算机，更多的核心将支持更多的并发线程。让线程数量多于内核数量通常将带来更好的结果。不过，其实际效果非常依赖于应用程序本身，还依赖于应用程序线程花费在等待资源上的时间。

之所以线程池实现要优于手工创建线程，主要有两个因素。首先，线程池将重用那些被释放了的线程。而在手工创建线程时，必须为每个任务创建一个全新的线程。线程的创建和销毁所花费的时间要高于.NET线程池管理所带来的开销。

第二，线程池将为你管理活动线程的数量。若创建了过多的线程，那么系统将挂起一部分，直到有足够的资源执行。QueueUserWorkItem则将工作交给线程池中接下来的一个可用线程，并帮你完成一定的线程管理工作。若应用程序线程池中所有的线程均被占用，那么线程池也会挂起任务，直至出现可用线程。

随着多核处理器的一天天普及，你会越来越频繁遇到多线程的应用程序。若你在开发.NET服务器端应用程序，例如WCF、ASP.NET或.NET远程处理，那么则已经开始与多线程打交道了。这些.NET子系统均使用线程池来管理线程，因此你也应该采用同样的做法。线程池能够降低额外开销，进而提高性能。此外，.NET线程池也能够帮你更好地管理当前用于执行工作的活动线程数量。

条目12：使用BackgroundWorker实现线程间通信

条目11演示了使用ThreadPool.QueueUserWorkItem来执行多个后台任务。该API非常易于使用，因为大多数的线程管理工作都交给了框架和底层的操作系统来完成。其中的很多功能也能够重用，因此若在应用程序中需要使用到后台线程来执行任务的话，那么QueueUserWorkItem将是你首要考虑使用的工具。但对于将要执行的后台任务，QueueUserWorkItem有着几个假设。若是程序的实际需求不符合这些假设，那么还是需要额外的工作。不过不是直

接使用System.Threading.Thread来创建线程，而是使用System.ComponentModel.BackgroundWorker。BackgroundWorker不仅构造于ThreadPool之上，还为线程间通信提供了很多支持。

其中需要处理的最重要的问题就是在WaitCallback中抛出的异常。WaitCallback即为实际执行任务的后台线程，若是该方法中有任何异常抛出，那么系统将终止应用程序，而不仅仅是终止该后台线程。这个行为和其他后台线程API的行为一致，不过处理的难点在于，QueueUserWorkItem并没有提供任何内建的错误处理机制。

此外，QueueUserWorkItem也没有提供实现后台线程和前台线程之间交互的内建支持——你不能检查完成情况、跟踪进度、暂停任务或是取消任务。若程序需要这些功能，那么则要使用建立在QueueUserWorkItem功能之上的BackgroundWorker组件。

BackgroundWorker组件包含了System.ComponentModel.Component的功能，用来提供设计时的支持。不过在没有设计器支持时，BackgroundWorker在代码中也非常有用。实际上，我在使用BackgroundWorker的大多数情况时，都不是在Windows窗体中。

BackgroundWorker的最简单使用方法是创建一个符合委托签名的方法，将该方法附加到BackgroundWorker的DoWork事件上，然后调用BackgroundWorker的RunWorkerAsync()方法。

```
BackgroundWorker backgroundWorkerExample =
    new BackgroundWorker();
backgroundWorkerExample.DoWork += new
    DoWorkEventHandler(backgroundWorkerExample_DoWork);
backgroundWorkerExample.RunWorkerAsync();

// 其他位置:
void backgroundWorkerExample_DoWork(object sender,
    DoWorkEventArgs e)
{
    // 方法的内容省略
}
```

在这个模式中，BackgroundWorker提供的功能和ThreadPool.QueueUserWorkItem完全一致。BackgroundWorker将使用ThreadPool执行其后台任务，其内部也会用到QueueUserWorkItem。

BackgroundWorker的强大之处在于，其整套框架已经为一些常见的场景提供了支持。例如，BackgroundWorker使用事件在前台和后台线程之间通信。当前台线程发起一个请求时，BackgroundWorker将触发后台线程上的DoWork事件。随后DoWork事件的处理程序将读取其参数，并开始相应的执行工作。

在后台线程任务结束（即DoWork事件处理函数执行完成）之后，BackgroundWorker将在前台线程中触发RunWorkerCompleted事件，如图2-2所示。这样，前台线程即可根据需要在后台线程结束时执行必要的后续处理。

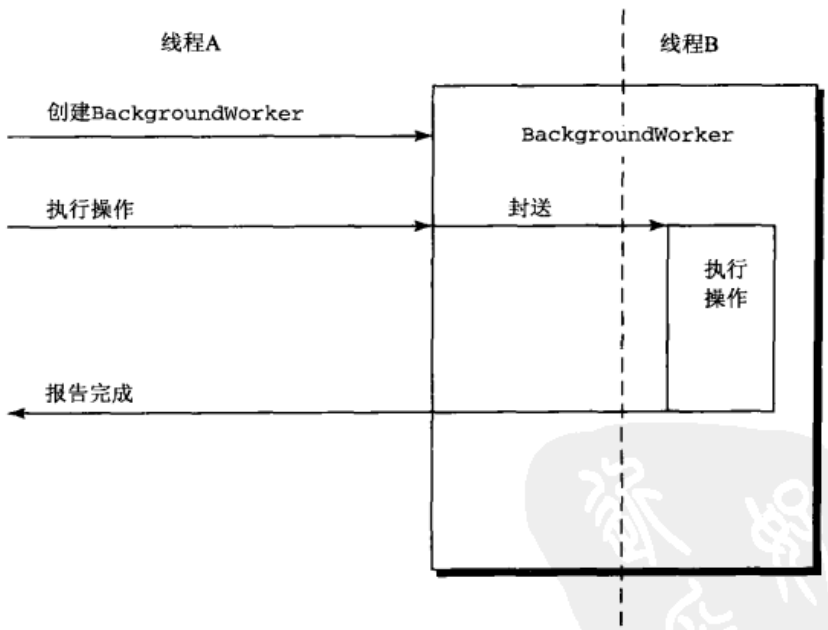

图2-2　BackgroundWorker类能够在后台线程执行完毕时触发前台线程中的事件处理程序。只要注册了完成事件，那么在DoWork委托完成执行之后，BackgroundWorker就会触发该事件

除了BackgroundWorker触发的事件，也可以用属性来维护控制前台线程和后台线程间的交互。WorkerSupportsCancellation属性让Background-

Worker知道后台线程是否能够中止该操作并退出。WorkerReportsProgress属性则会告知BackgroundWorker对象，每过一段时间后台线程将会通知前台线程其执行进度，如图2-3所示。此外，BackgroundWorker还能将取消请求从前台线程转发给后台线程。这样，后台线程即可检查CancellationPending标记，并根据需要停止执行。

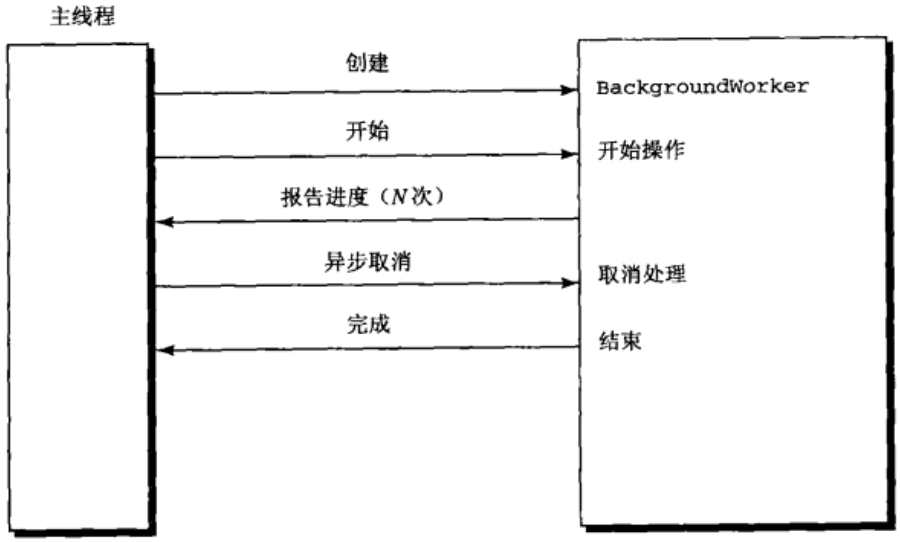

图2-3 BackgroundWorker支持使用多个事件来取消执行任务，向前台线程报告执行进度以及错误报告。BackgroundWorker定义了线程间通信的协议，并在需要的时候通过事件来支持通信机制。若想报告执行进度，那么后台线程必须触发定义在BackgroundWorker中的事件。前台线程的代码也必须支持此类事件，提供必要的事件处理程序

此外，BackgroundWorker还拥有内建的协议来支持报告后台线程中发生的错误。在条目11中，我曾经解释过异常不能从某个线程抛到另一个线程中。若是后台线程中抛出了异常，且没有被捕获，那么该线程将被终止。不仅如此，前台线程也不会收到任何有关后台线程已被终止的通知。BackgroundWorker通过在DoWorkEventArgs中添加Error属性，并将异常信息保存在其中来解决了这个问题。后台线程可以捕获所有的异常，并将其设置到Error属性中。（需要注意的是，这是仅有的几个需要捕获所有异常的场景之一。）随后在后台线程返回时，前台线程即可在事件处理程序中处理该异常。

前面曾提到过，我经常在非Form类中使用BackgroundWorker，甚至在非

Windows窗体中，例如服务或Web服务等。不过有几点注意之处。当BackgroundWorker检测到其正运行于Windows窗体程序中，且该窗体为可见，那么ProgressChanged和RunWorkerCompleted事件将通过转发控制以及Control.BeginInvoke被转发给GUI线程中（参见本章条目16）。在其他情况下，这些委托只是运行于线程池中的某个空闲线程上。在条目16中你将看到，这个行为可能会影响到接收到事件的顺序。

最后一点，因为BackgroundWorker构建于QueueUserWorkItem之上，所以我们可以使用BackgroundWorker来处理多个后台请求。通过检查BackgroundWorker的IsBusy属性即可判断BackgroundWorker当前是否在执行任务。若需要同时执行多个后台任务，那么可以创建多个BackgroundWorker对象。这些BackgroundWorker对象均会共享同一个线程池，因此其实际的执行效果和QueueUserWorkItem一样。这样就需要保证事件处理程序要访问到正确的线程，以便保证后台线程和前台线程之间通信的正确性。

BackgroundWorker支持创建后台任务时的很多常用模式。借助于BackgroundWorker，我们即可提高代码的重用性，根据需要使用这些模式，而并不用手工定义前后台线程之间的通信协议。

条目13：让lock()作为同步的第一选择

线程之间需要进行通信。我们需要提供一种安全的方式，让应用程序中的线程能够发送并接收数据。不过，在线程间共享数据可能会引发同步问题，导致数据完整性方面的错误。因此，必须保证每一块共享数据的当前状态都是一致的。实现该目标需要使用同步原语（synchronization primitive）来保护对同享数据的访问。同步原语能够保证在完成一系列必要操作之前，当前线程不会被打断。

.NET BCL中提供了很多不同的同步原语，均可用来保证共享数据的同步。不过仅有一对——Monitor.Enter()和Monitor.Exit()——在C#语言上得到了原生支持。Monitor.Enter()和Monitor.Exit()共同组成了一个临界

区。临界区在保证同步方面的应用非常广泛，因此语言本身就对其提供了支持——lock()语句。这样，我们应该尽可能遵循语言设计者的意愿，让lock()作为保证同步的第一选择。

原因很简单：编译器生成的代码永远是一致的，而开发者则可能会犯错误。C#语言引入了lock关键字来控制多线程程序的同步操作。lock语句将生成与正确使用Monitor.Enter()和Monitor.Exit()同样的代码。此外，该关键字更见简单，且能够自动生成所需的可以安全处理异常的代码。

不过，在两种情况下，Monitor也能提供两种lock()无法实现的功能。首先，lock必须使用在同一个上下文中。也就是说，在使用lock时，你无法在一个上下文中进入锁定却在另一个上下文中退出。例如，无法在某个方法中进入Monitor，然后在该方法中的某个lambda表达式中退出（参见第5章条目41）。其次，Monitor.Enter支持制定一个超时时间，这一点将在稍后介绍。

按如下方式使用lock语句即可锁定某一引用类型。

```
public int TotalValue
{
    get
    {
        lock(syncHandle)
        {
            return total;
        }
    }
}

public void IncrementTotal()
{
    lock (syncHandle)
    {
        total++;
    }
}
```

lock语句将独占地锁定某一对象，并确保在锁定被释放之前其他线程无法

再次锁定。上述使用lock()的示例代码将生成与下面使用Monitor.Enter()和Monitor.Exit()代码同样的IL。

```
public void IncrementTotal()
{
    object tmpObject = syncHandle;
    System.Threading.Monitor.Enter(tmpObject);
    try
    {
        total++;
    }
    finally
    {
        System.Threading.Monitor.Exit(tmpObject);
    }
}
```

为了避免常见错误，lock语句还提供了很多检查，例如检查被锁定对象为引用类型。而Monitor.Enter则并没有包含这些检查。如下这段使用lock()的代码无法通过编译。

```
public void IncrementTotal()
{
    lock (total) // 编译期错误：无法锁定值类型
    {
        total++;
    }
}
```

不过如下代码则可以编译通过。

```
public void IncrementTotal()
{
    // 并没有真正锁定住total
    // 锁住的是total的一个装箱对象
    Monitor.Enter(total);
    try
    {
        total++;
    }
    finally
```

```
        {
            // 会抛出异常
            // 释放了包含total的另一个装箱对象
            Monitor.Exit(total);
        }
    }
```

之所以Monitor.Enter()可以通过编译，是因为其签名接受的是System.Object对象，因此程序将把total装箱成对象传入。这样，Monitor.Enter()实际上锁定的是total的一个装箱对象，因此埋下了一个潜在的bug。假设第一个线程进入到了IncrementTotal()中并获取了锁定。然后在对total进行操作时，第二个线程也调用了IncrementTotal()。这时，第二个线程依旧可以成功地获取锁定，因为total可被装箱成另外一个对象。第一个线程获取了total的一个装箱，第二个线程则获取了total的另一个装箱。可以看到，这样做不但增加了代码，也没有实现保证同步所需的要求。

这段代码中还有一个bug：在任意一个线程尝试释放total上的锁时，Monitor.Exit()均会抛出SynchronizationLockException异常。这是因为调用Monitor.Exit()时total又被装箱到了另外的一个新对象中，因为Monitor.Exit()接受的也是System.Object类型。在释放锁时，释放的对象和开始时锁定的对象并不是同一个，自然也就导致Monitor.Exit()调用失败并抛出异常。

或许有人聪明一些，想出了这样的做法。

```
public void IncrementTotal()
{
    // 同样无法正常执行：
    object lockHandle = total;
    Monitor.Enter(lockHandle);
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(lockHandle);
    }
}
```

虽然这段代码不会抛出异常，不过也不能保证共享数据的同步。每次调用IncrementTotal()时，均会为total创建一个新的装箱，并锁定该对象。这样，每个线程都能立即获取到所需要的锁，但是却没有锁定到任何共享的资源上。导致的结果就是，虽然每个线程都不会阻塞，但total却无法保持一致。

lock还能预防一些比较易于忽视的问题。Enter()和Exit()是两个独立的调用，很容易写错，导致获取和释放的是两个不同的对象。这将导致SynchronizationLockException。而若是需要锁定多个对象，那么也很有可能在临界区结束时释放了错误的对象。

lock语句还能够自动地生成能够安全处理异常的代码，而这些往往是开发者所忽视的。此外，它生成的代码也要比Monitor.Enter()和Monitor.Exit()更加高效，因为其只需要对目标对象进行一次求值。因此在默认情况下，我们应该在C#程序中尽可能地使用lock语句来保证同步性。

不过，lock语句所生成的MSIL也存在着局限：Monitor.Enter()将在获取到锁之前永远等待下去。这样就可能造成死锁。在大规模的企业系统中，访问关键资源的策略应该更加小心仔细、趋于保守。这时，使用Monitor.TryEnter()即可给出一个等待的超时时间，并给出无法访问到关键资源时的处理方法。

```
public void IncrementTotal()
{
    if (!Monitor.TryEnter(syncHandle, 1000)) // 等待1s
        throw new PreciousResourceException
            ("Could not enter critical section");
    try
    {
        total++;
    }
    finally
    {
        Monitor.Exit(syncHandle);
    }
}
```

还可使用泛型类对其进行简单包装。

```
public sealed class LockHolder<T> : IDisposable
    where T : class
{
    private T handle;
    private bool holdsLock;

    public LockHolder(T handle, int milliSecondTimeout)
    {
        this.handle = handle;
        holdsLock = System.Threading.Monitor.TryEnter(
            handle, milliSecondTimeout);
    }

    public bool LockSuccessful
    {
        get { return holdsLock; }
    }

    #region IDisposable Members
    public void Dispose()
    {
        if (holdsLock)
            System.Threading.Monitor.Exit(handle);
        // 不要重复释放
        holdsLock = false;
    }
    #endregion
}
```

随后这样使用该泛型类。

```
object lockHandle = new object();

using (LockHolder<object> lockObj = new LockHolder<object>
    (lockHandle, 1000))
{
    if (lockObj.LockSuccessful)
    {
        // 具体操作省略
    }
```

```
}
// 在此处析构
```

之所以C#开发团队为Monitor.Enter()和Monitor.Exit()添加了语言级别的支持（即lock语句），是因为这是一种最常用的同步机制。编译器所做的额外检查也能让你更容易地编写出要求保证同步的代码。因此对于大多数C#应用程序来讲，lock()都是保证同步的最佳选择。

不过lock并不是同步的唯一选择。实际上，若是需要同步地访问某个值类型，或替换某个引用类型，那么System.Threading.Interlocked类型即可直接支持对象上的单一操作。System.Threading.Interlocked提供了一系列方法，可用来访问共享数据，并保证在其他线程访问该数据之前就完成上一次操作。Interlocked还能帮你预防操作共享数据时将会遇到的一些常见的同步问题。

例如如下方法：

```
public void IncrementTotal()
{
    total++;
}
```

在这样的实现中，多线程访问可能会造成数据的不一致。因为自增操作符并不是单一的一条机器指令。total变量的值需要首先从内存读入到寄存器中，然后在寄存器中自增，最后再从寄存器写回到内存中的特定位置里。若是另外一个线程在第一个线程之后再次读取该变量，此时第一个线程已经在寄存器中完成了增加但尚未写回内存，就会造成数据的不一致。

假设两个线程几乎在同时调用了IncrementTotal。线程A读取了total的值为5。此时，活动线程切换到了线程B。于是线程B读取到了5，自增，然后把6写回到total中。这时活动线程又切换回了线程A。线程A将在寄存器中将数值自增到6，然后写回到total中。这样，虽然IncrementTotal()被调用了两次（线程A和线程B），不过结果是total仅仅自增了一次。此类问题很难发现，因为只有在非常凑巧的时候才会发生这类交叉访问的情况。

虽然可以使用lock()来保证同步，不过还有一种更好的办法。Interlocked类提供了一个专门的InterlockedIncrement方法来解决这个问题。按照如下方法重写IncrementTotal，即可保证自增操作不会被打断，两次自增操作均会成功。

```
public void IncrementTotal()
{
    System.Threading.Interlocked.Increment(ref total);
}
```

Interlocked类还提供了另外一些处理内建类型的方法。例如Interlocked.Decrement()能够自减某个值，Interlocked.Exchange()能够将变量的值交换成新的值，并将原始值返回。你可以用Interlocked.Exchange()来设定一个新的状态，并将从前的状态返回。例如，若需要将最后一个访问某资源的用户ID保存起来，即可调用Interlocked.Exchange()来保存当前的用户ID，并同时获取到前一个访问的用户ID。

Interlocked还提供了CompareExchange()方法，用来读取某个共享的数据，随后判断若其与某一值相同的话，则赋以新值，否则不作任何操作。两种情况下CompareExchange都会返回从前的值。在下一节中，条目14将演示如何使用CompareExchange来在类中创建一个私有的锁对象。

同步原语中并不只包含Interlocked和lock()。Monitor类还提供了Pulse和Wait方法，可用来实现消费者/生产者模型。在很多线程读取某一资源，且很少线程修改该资源时，可以使用ReaderWriterLockSlim实现该设计。ReaderWriterLockSlim对早先版本的ReaderWriterLock做了一些改进，因此在开发中应该选用ReaderWriterLockSlim。

对于大多数同步问题，都可以先看看Interlocked是否能够满足你的需要。很多单一的操作都可以用它来实现。否则应尽可能地使用lock()语句。只有在确实需要某些特定的锁实现时，再考虑使用别的方法。

条目14：尽可能地减小锁对象的作用范围

在编写并发程序时，我们需要选择最合适的同步原语。应用程序中对同步原语使用得越多，也就越难以避免发生死锁或失锁等并发上的错误。这是个规模的问题：需要检查的地方越多，也就越难发现某个特定的错误。

在面向对象编程中，我们使用私有成员变量来尽可能减少（不是移除，而是减少）发生状态变化的位置的数量。在并发程序中，同样也应该尽可能地减小用来实现同步对象的作用范围。

若从上述角度来看，两种广泛应用的锁方式均不满足要求。lock(this)和lock(TypeOf (MyType))都使用了公共实例来创建锁对象。

若是你像下面这样编写代码：

```
public class LockingExample
{
    public void MyMethod()
    {
        lock (this)
        {
            // 省略
        }
    }
    // 省略
}
```

再假如你的某个客户——叫他亚历山大好了——说他需要锁住一个对象。于是亚历山大这样写了代码：

```
LockingExample x = new LockingExample();
lock (x)
    x.MyMethod();
```

此类锁定策略很容易就造成了死锁。客户代码在LockingExample对象上获取了锁，而MyMethod中却又尝试在同一个对象上获取锁。虽然这里的问题很容

易看出来，不过也许某一天，另一个线程又在其他什么地方锁定了该LockingExample对象。这时发生的死锁将很难找到其原因。

我们需要改变锁定的策略，你可以采用下面将要介绍的三种方法。

第一种方法是，若你需要保护整个一个方法，那么可以使用MethodImplAttribute属性来指定该方法是同步的。

```
[MethodImpl(MethodImplOptions.Synchronized)]
public void IncrementTotal()
{
    total++;
}
```

不过显然，这种需求并不常见。

第二种方法是，强制所有的开发者都必须仅锁定当前的类型或当前的对象，即使用lock(this)或lock(MyType)。若是每个人都遵守该规则的话，那么也不会出现什么问题。不过若想达到这个目标，需要你的程序的每个使用者都清楚地了解并遵守这个规则，即只能够锁定当前对象或当前类型，而不能是别的对象。这种理想的假设显然不够现实。

第三种则是最好的办法。你可以在类中创建一个同步对象，专门用来保护访问共享的资源。该同步对象是一个私有成员变量，因此无法在类型之外访问到。你也可以保证该同步对象为私有，且不能被任何非私有属性访问。这样即可确保锁定语句安全地锁定到指定的对象上。

通常，我们会创建一个System.Object对象作为同步对象。随后在类中访问需要保护的成员时即可锁定该同步对象。但在创建同步对象时需要小心，不要因为线程的交替执行创建出了多个同步对象的副本。Interlocked类的CompareExchange方法能够验证某个值，并在必要时替换成新值。我们可以使用该方法来保证类型中仅分配了一个同步对象。

下面就是第三种做法的最简单实现。

```
private object syncHandle = new object();

public void IncrementTotal()
{
    lock (syncHandle)
    {
        // 代码省略
    }
}
```

或许你会发现，程序并不需要经常地锁定，因此只要在需要锁定时创建同步对象即可。这时，创建锁定对象可采用如下的一种非常巧妙方式。

```
private object syncHandle;

private object GetSyncHandle()
{
    System.Threading.Interlocked.CompareExchange(
        ref syncHandle, new object(), null);
    return syncHandle;
}

public void AnotherMethod()
{
    lock (GetSyncHandle())
    {
        // 代码省略
    }
}
```

syncHandle用来控制类中对共享资源的访问。私有的GetSyncHandle()方法将返回同步对象。而不能被打断的CompareExchange调用则保证了程序仅会创建出一个同步对象。CompareExchange首先比较syncHandle和null，若syncHandle的当前值为null，那么CompareExchange将创建一个新的对象，并将其指派给syncHandle。

这种做法适用于实例方法中的任何一种锁定，不过静态方法又该如何实现呢？仍旧使用类似的方法，不过创建的是一个静态的同步对象，从而让该类的所有实例都共用一个同步对象。

当然，你可以在方法（属性访问器或索引器也可）内的任意一段代码上创建同步区块。不过不管怎样，你都应该尽可能地减少被锁定的代码。

```
public void YetAnotherMethod()
{
    DoStuffThatIsNotSynchronized();
    int val = RetrieveValue();
    lock (GetSyncHandle())
    {
        // 代码省略
    }
    DoSomeFinalStuff();
}
```

若是在lambda表达式中使用锁，那么必须小心处理。C#编译器将在lambda表达式外面创建一个闭包。该闭包和C# 3.0支持的延迟执行模型让开发者很难判断锁定何时才能结束，也就更容易发生死锁情况。因为开发者可能无法判断某段代码是否位于某个锁定区域内。

在结束这个话题之前，还有另外两个锁定相关的建议。若你发现需要在某个类中创建不同的锁对象，那么或许应该考虑将该类拆分成多个类。因为该类做的事情太多了。若是需要保护对某些变量的访问，同时也需要用其他的锁来保护类中其他的变量，那么则非常建议你将类拆成具有不同责任的类。若每个类都是一个独立的单元，那么将更容易保证一致性。每个拥有共享数据（将由不同线程访问或修改）的类都应该仅用一个同步对象来保证其一致性。

在考虑锁定时，应选择一个外部不可见的私有字段。不要锁定公共对象，因为锁定公共对象要求所有的开发者都永远遵循同样的规范，且非常容易导致死锁。

条目 15：避免在锁定区域内调用外部代码

有些情况下，问题的出现是因为没有进行足够的锁定。不过当你再创建新的锁时，接下来却又可能发生死锁。当两个线程各持有了一个资源，同时也在等待对方的资源时，死锁自然难以避免。在.NET Framework中，有一种特殊情况是两个线程的执行几乎不分前后，同时进行。这时，虽然只有一个资源被锁定，但仍可能发生死锁。（条目16将介绍这种情况。）

我们已经介绍了一种避免这个问题的最简单方法：条目13使用一个私有的数据成员作为同步对象，从而锁定了共享的数据。不过即使这样，仍有可能会导致死锁。若是在某段同步区域中调用了外部代码，那么另外的线程也有可能引发死锁。

例如，使用如下代码来处理一个后台操作。

```
public class WorkerClass
{
    public event EventHandler<EventArgs> RaiseProgress;
    private object syncHandle = new object();

    public void DoWork()
    {
        for(int count = 0; count < 100; count++)
        {
            lock (syncHandle)
            {
                System.Threading.Thread.Sleep(100);
                progressCounter++;
                if (RaiseProgress != null)
                    RaiseProgress(this, EventArgs.Empty);
            }
        }
    }

    private int progressCounter = 0;
    public int Progress
```

```
        {
            get

            {
                lock (syncHandle)
                    return progressCounter;
            }
        }
    }
```

RaiseProgress()方法将通知所有的事件监听程序。所有的监听程序都可以注册并处理该事件。在多线程程序中，一个典型的事件处理程序将如下所示。

```
static void engine_RaiseProgress(object sender, EventArgs e)
{
    WorkerClass engine = sender as WorkerClass;
    if (engine != null)
        Console.WriteLine(engine.Progress);
}
```

程序运行不会出现什么问题，不过这也是只因为幸运而已。之所以没有问题，是因为事件处理程序运行于后台线程中。

不过，假设该程序是一个Windows窗体应用程序，且你需要让事件处理程序在UI层上执行（参见条目16）。这可以使用Control.Invoke来实现。不仅如此，Control.Invoke还将阻塞原有线程，直到目标委托执行完毕为止。但这也不会出现什么问题——我们的操作运行于另一个线程上，因此不会导致死锁。

而第二个重要的操作却导致了整个程序的死锁。为了获取进度的详细情况，事件处理程序使用了engine对象。不过其Progress访问器目前却是运行在另一个线程上，因此无法获取同样的锁。

Progress访问器锁定了该同步对象。在本地上下文中，这没什么问题，不过实际并非如此。UI线程将需要锁定这个已经在后台线程中被锁定的同步对象。不过后台线程却也处于阻塞状态中，等待事件处理程序返回，同时后台线程已经锁定了该同步对象。这就难以避免地发生了死锁。

表2-1给出了调用栈。可以看到查出此类问题并不容易。调用栈中，在第一个锁定和第二次尝试锁定之间有8种方法。而且，这些线程的交替执行均在框架内部发生，在真正发生问题时，你甚至无法看到这些详细的信息。

表2-1　用来更新窗体显示的前后台线程执行过程的调用栈

方　法	线　程
DoWork	BackgroundThread
raiseProgress	BackgroundThread
OnUpdateProgress	BackgroundThread
engine_OnUpdateProgress	BackgroundThread
Control.Invoke	BackgroundThread
UpdateUI	UIThread
Progress (property access)	UIThread (deadlock)

核心问题是代码需要再次获取一个锁。因为你不知道控件外部的代码将如何工作，所以应该尽量避免在同步区域内调用外部代码。在这个示例中，这就意味着你必须在同步区域之外触发进度报告事件。

```
public void DoWork()
{
    for(int count = 0; count < 100; count++)
    {
        lock (syncHandle)
        {
            System.Threading.Thread.Sleep(100);
            progressCounter++;
        }
        if (RaiseProgress != null)
            RaiseProgress(this, EventArgs.Empty);
    }
}
```

既然你已经看到了问题的所在，那么现在有必要让你完全理解调用未知代码所可能给应用程序带来的影响了。显然，触发公开的访问事件将调用外部代码。使用以参数形式传入或通过公开API设定的委托将调用外部代码。使用以参数形式传入的lambda表达式也可能会调用到外部代码（参见第5章条目40）。

此类外部代码很容易发现，不过还有一个未知却并不那么容易找到：虚方法。虚方法调用的可能是派生类的重写版本，而这个重写的类中则可能调用任

意的方法，也就有可能导致死锁。

无论具体是什么情况，问题的成因都是相似的。你的类首先获取了一个锁。随后在同步区域内，调用了外部代码。该外部代码有可能会最终调用回你的类中，甚至在另外的线程上。你无法确保这些外部代码不做任何有害的事情。因此则必须从源头上杜绝：不要在代码的同步区域内调用外部代码。

条目 16：理解 Windows 窗体和 WPF 中的跨线程调用

若你曾开发过Windows窗体程序，可能会注意到有时事件处理程序将抛出InvalidOperationException异常，信息为“跨线程调用非法：在非创建控件的线程上访问该控件”。这种Windows窗体应用程序中跨线程调用时的一个最为奇怪的行为就是，有些时候它没什么问题，可有些时候却会出现问题。在WPF（Windows Presentation Foundation）中，这个行为有所改变。WPF中跨线程调用将永远不会成功。不管怎样，至少这能让你在开发过程中更容易地找到问题的所在。

在Windows窗体中，解决方法是首先检查Control.InvokeRequired属性，若Control.InvokeRequired属性为true，那么调用ControlInvoke()。在WPF中，可以使用System.Windows.Threading.Dispatcher中的Invoke()和BeginInvoke()方法。这两种情况中都发生了很多事情，你也同样有别的选择。这两个API为你做了很多事情，不过在某些情况下仍有可能会失败。因为这些方法将用来处理跨线程调用，因此若是没有正确使用（甚至是正确使用但没有完全理解其行为）的话，也有可能会导致竞争条件的出现。

无论是Windows窗体还是WPF，问题的成因都很简单：Windows控件使用的是组件对象模型（Component Object Model，COM）单线程单元（Single-threaded Apartment，STA）[①]模型，因为其底层的控件是单元线程（apartment-threaded）

① 简而言之，STA的程序每个线程都拥有自己独立的资源，这些资源无法被别的线程访问到。而.NET应用程序默认使用的是多线程单元（Multi-threaded Apartment，MTA）模型。——译者注

的。此外，很多控件都用消息泵（message pump）来完成操作。因此，这种模型就需要所有调用该控件的方法都和创建该控件的方法位于同一个线程上。Invoke、BeginInvoke和EndInvoke调度方法都需要在正确的线程上调用。两种模型的底层代码非常相似，因此这里将以Windows窗体的API为例。不过当调用方法有所区别时，我将同时给出两个版本。其具体的做法非常复杂，但仍需要深入了解。

首先，我们来看一段简单的泛型代码，能够让你在遇到此种情况时得到一定的简化。匿名委托让仅在一处使用的小方法更加易于编写。不过，匿名委托却并不能与接受System.Delegate类型的方法（例如Control.Invoke）配合使用。因此，你需要首先定义一个非抽象的委托类型，随后在使用Control.Invoke时传入。

```
private void OnTick(object sender, EventArgs e)
{
    Action action = () =>
        toolStripStatusLabel1.Text =
            DateTime.Now.ToLongTimeString();
    if (this.InvokeRequired)
        this.Invoke(action);
    else
        action();
}
```

C# 3.0大大简化了上述代码。System.Core.Action委托定义了一类专门的委托类型，用来表示不接受任何参数并返回void的方法。lambda表达式也能够更加简单地定义方法体。但若你仍旧需要支持C# 2.0，那么需要编写如下的代码。

```
delegate void Invoker();
private void OnTick20(object sender, EventArgs e)
{
    Action action = delegate()
    {
        toolStripStatusLabel1.Text =
            DateTime.Now.ToLongTimeString();
    };
```

```
    if (this.InvokeRequired)
        this.Invoke(action);
    else
        action();
}
```

在WPF中，则需要使用控件上的System.Threading.Dispatcher对象来执行封送操作。

```
private void UpdateTime()
{
    Action action = () => textBlock1.Text =
        DateTime.Now.ToString();
    if (System.Threading.Thread.CurrentThread !=
        textBlock1.Dispatcher.Thread)
    {
        textBlock1.Dispatcher.Invoke
            (System.Windows.Threading.DispatcherPriority.Normal,
            action);
    }
    else
    {
        action();
    }
}
```

这种做法让事件处理程序的实际逻辑变得更加模糊，让代码难以阅读和维护。这种做法还需要引入一个委托定义，仅仅用来满足方法的签名。

使用一小段泛型代码即可改善这种情况。下面的这个ControlExtensions静态类所包含的泛型方法适用于调用不超过两个参数的委托。再添加一些重载即可支持更多的参数。此外，其中的方法还可使用委托定义来调用目标方法，既可以直接调用，也可以通过Control.Invoke的封送。

```
public static class ControlExtensions
{
    public static void InvokeIfNeeded(this Control ctl,
        Action doit)
    {
        if (ctl.InvokeRequired)
```

```
            ctl.Invoke(doit);
        else
            doit();
    }

    public static void InvokeIfNeeded<T>(this Control ctl,
        Action<T> doit, T args)
    {
        if (ctl.InvokeRequired)
            ctl.Invoke(doit, args);
        else
            doit(args);
    }
}
```

在多线程环境中使用InvokeIfNeeded能够很大程度上简化事件处理程序的代码。

```
private void OnTick(object sender, EventArgs e)
{
    this.InvokeIfNeeded(() => toolStripStatusLabel1.Text =
        DateTime.Now.ToLongTimeString());
}
```

对于WPF控件，也可以创建出一系列类似的扩展。

```
public static class WPFControlExtensions
{
    public static void InvokeIfNeeded(
        this System.Windows.Threading.DispatcherObject ctl,
        Action doit,
        System.Windows.Threading.DispatcherPriority priority)
    {
        if (System.Threading.Thread.CurrentThread !=
            ctl.Dispatcher.Thread)
        {
            ctl.Dispatcher.Invoke(priority,
                doit);
        }
        else
        {
            doit();
```

```
        }
    }
    public static void InvokeIfNeeded<T>(
        this System.Windows.Threading.DispatcherObject ctl,
        Action<T> doit,
        T args,
        System.Windows.Threading.DispatcherPriority priority)
    {
        if (System.Threading.Thread.CurrentThread !=
            ctl.Dispatcher.Thread)
        {
            ctl.Dispatcher.Invoke(priority,
                doit, args);
        }
        else
        {
            doit(args);
        }
    }
}
```

WPF版本没有检查InvokeRequired，而是检查了当前线程的标识，并于将要进行控件交互的线程进行比较。DispatcherObject是很多WPF控件的基类，用来为WPF控件处理线程之间的分发操作。注意，在WPF中还可以指定事件处理程序的优先级。这是因为WPF应用程序使用了两个UI线程。一个线程用来专门处理UI呈现，以便让UI总是能够及时呈现出动画等效果。你可以通过指定优先级来告诉框架哪类操作对于用户更加重要：要么是UI呈现，要么是处理某些特定的后台事件。

这段代码有几个优势。虽然使用了匿名委托定义，不过事件处理程序的核心仍位于事件处理程序中。与直接使用Control.IsInvokeRequired或ControlInvoke相比，这种做法更加易读且易于维护。在ControlExtensions中，使用了泛型方法来检查InvokeRequired或是比较两个线程，这也就让使用者从中解脱了起来。若是代码仅在单线程应用程序中使用，那么我也不会使用这些方法。不过若是程序最终可能在多线程环境中运行，那么不如使用上面这种更加完善的处理方式。

若想支持C# 2.0，那么还要做一些额外的工作。主要在于无法使用扩展方法和lambda表达式语法。这样，代码将变得有些臃肿。

```
// 定义必要的Action:
public delegate void Action;
public delegate void Action<T>(T arg);
// 3个和4个参数的Action定义省略

public static class ControlExtensions
{
    public static void InvokeIfNeeded(Control ctl, Action doit)
    {
        if (ctl.InvokeRequired)
            ctl.Invoke(doit);
        else
            doit();
    }

    public static void InvokeIfNeeded<T>( Control ctl,
        Action<T> doit, T args)
    {
        if (ctl.InvokeRequired)
            ctl.Invoke(doit, args);
        else
            doit(args);
    }
}

// 其他位置:

private void OnTick20(object sender, EventArgs e)
{
    ControlExtensions.InvokeIfNeeded(this, delegate()
    {
        toolStripStatusLabel1.Text =
          DateTime.Now.ToLongTimeString();
    });
}
```

在将这个方法应用到事件处理程序之前，我们来仔细看看InvokeRequired和Control.Invoke所做的工作。这两个方法并非没有什么代价，也不建议将这种模式应用到各处。Control.InvokeRequired用来判断当前代码是运行于创建该控件的线程之上，还是运行于另一个线程之上。若是运行于另一个线程之上，那么则需要使用封送。大多数情况下，这个属性的实现还算简单：只要检查当前线程的ID，并与创建该控件的线程ID进行比较即可。若二者匹配，那么则无需Invoke，否则就需要Invoke。这个比较并不需要花费太多时间，WPF版本的这类扩展方法也是执行了同样的检查。

不过其中还有一些边缘情况。若需要判断的控件还没有被创建，在父控件已创建好，正在创建子控件时就可能发生这个情况。那么此时，虽然C#对象已经存在，不过其底层的窗口句柄仍旧为null。此时也就无法进行比较，因此框架本身将花费一定代价来处理这种情况。框架将沿着控件树向上寻找，看看是否有上层控件已被创建。若是框架能够找到一个创建好了的窗体，那么该窗体将作为封送窗体。这是一个非常合理的假设，因为父控件将要负责创建子控件。这种做法可以保证子控件将会与父控件在同一个线程上创建。找到合适的父控件之后，框架即可执行同样的检查，比较当前线程的ID和创建该父控件的线程的ID。

不过，若是框架无法找到任何一个已创建的父窗体，那么则需要找到一些其他类型的窗体。若在层次体系中无法找到可用的窗体，那么框架将开始寻找*暂存窗体*（parking window），暂存窗体让你不会被某些Win32 API奇怪的行为所干扰。简而言之，有些对窗体的修改（例如修改某些样式）需要销毁并重新创建该窗体。暂存窗体就是用来在父窗体被销毁并重新创建的过程中用来临时保存其中的控件的。在这段时间内，UI线程仅运行于暂存窗体中。

在WPF中，得益于Dispatcher类的使用，上述很多过程都得到了简化。每个线程都有一个Dispatcher。在第一次访问某个控件的Dispatcher时，类库将察看该线程是否已经拥有了Dispatcher。若已经存在，那么直接返回。如果没有的话，那么将创建一个新的Dispatcher对象，并关联在控件及其所在的线程之上。

不过这其中仍旧有可能存在着漏洞和发生失败。有可能所有的窗体，包括暂存窗体都没有被创建。在这种情况下，InvokeRequired将返回false，表示无需将调用封送到另一个线程上。这种情况可能会比较危险，因为这个假设可能是错误的，但框架也仅能做到如此了。任何需要访问窗体句柄的方法都无法成功执行，因为现在还没有任何窗体。此外，封送也自然会失败。若是框架无法找到任何可以封送的控件，自然也无法将当前调用封送到UI线程上。于是框架选择了一个可能在稍后出现的失败，而不是当前会立即出现的失败。幸运的是，这种情况在实际中非常少见。不过在WPF中，Dispatcher还是包含了额外的代码来预防这种情况。

总结一下InvokeRequired的相关内容。一旦控件创建完成，那么InvokeRequired的效率将会不错，且也能保证安全。不过若是目标控件尚未被创建，那么InvokeRequired则可能会耗费比较长的时间。而若是没有创建好任何控件，那么InvokeRequired则可能要相当长的时间，同时其结论也无法保证正确。但虽然Control.InvokeRequired有可能耗时较长，也比非必要地调用Control.Invoke要高效得多。且在WPF中，很多边缘情况都得到了优化，性能要比Windows窗体的实现提高不少。

接下来看看Control.Invoke的执行过程。（Control.Invoke的执行非常复杂，因此这里将仅做简要介绍。）首先，有一个特殊情况是虽然调用了Invoke方法，不过当前线程却和控件的创建线程一样。这是个最为简单的特例，框架将直接调用委托。即当InvokeRequired返回false时仍旧调用Control.Invoke()将会有微小的损耗，不过仍旧是安全的。

在真正需要调用Invoke时会发生一些有趣的情况。Control.Invoke能够通过将消息发送至目标控件的消息队列来实现跨线程调用。Control.Invoke还创建了一个专门的结构，其中包含了调用委托所需要的所有信息，包括所有的参数、调用栈以及委托的目标等。参数均会被预先复制出来，以避免在调用目标委托之前被修改（记住这是在多线程的世界中）。

在创建好这个结构并添加到队列中之后，Control.Invoke将向目标对象

发送一条消息。Control.Invoke随后将在等待UI线程处理消息并调用委托时组合使用旋转等待（spin wait）和休眠。这部分的处理包含了一个重要的时间问题。当目标控件开始处理Invoke消息时，它并不会仅仅执行一个委托，而是处理掉队列中所有的委托。若你使用的是Control.Invoke的同步版本，那么不会看到任何效果。不过若是混合使用了Control.Invoke和Control.BeginInvoke，那么行为将有所不同。这部分内容将在稍后继续介绍，目前需要了解的是，控件的WndProc将在开始处理消息时处理掉每一个等待中的Invoke消息。对于WPF，可控制的要多一些，因为可以指定异步操作的优先级。你可以让Dispatcher将消息放在队列中时给出三种优先级：(1)基于系统或应用程序的当前状况；(2)使用普通优先级；(3)高优先级。

当然，这些委托中可能会抛出异常，且异常无法跨线程传递。因此框架将把对委托的调用用try/catch包围起来并捕获所有的异常。随后在UI线程完成处理之后，其中发生的异常将被复制到专门的数据结构中，供原线程分析。

在UI线程处理结束之后，Control.Invoke将察看UI线程中抛出的所有异常。如果确有异常发生，那么将在后台线程中重新抛出。若没有异常，那么将继续进行普通的处理。可以看到，调用一个方法的过程并不简单。

Control.Invoke将在执行封送调用时阻塞后台线程，虽然实际上在多线程环境中运行，不过仍旧让人觉得是同步的行为。

不过这可能不是你所期待的。很多时候，你希望让工作线程触发一个事件之后继续进行下面的操作，而不是同步地等待UI。这时则应该使用BeginInvoke。该方法的功能和Control.Invoke基本相同，不过在向目标控件发送消息之后，BeginInvoke将立即返回，而不是等待目标委托完成。BeginInvoke支持发送消息（可能在稍后才会处理）后立即返回到调用线程上。你可以根据需要为ControlExtensions类添加相应的异步方法，以便简化异步跨线程UI调用的操作。虽然与前面的那些方法相比，这些方法带来的优势不那么明显，不过为了保持一致，我们还是在ControlExtensions中给出。

```
public static void QueueInvoke(this Control ctl, Action doit)
{
    ctl.BeginInvoke(doit);
}

public static void QueueInvoke<T>(this Control ctl,
    Action<T> doit, T args)
{
    ctl.BeginInvoke(doit, args);
}
```

QueueInvoke并没有在一开始检查InvokeRequired。这是因为即使当前已经运行于UI线程之上，你仍可能想要异步地调用方法。BeginInvoke()就实现了这个功能。Control.BeginInvoke将消息发送至目标控件，然后返回。随后目标控件将在其下一次检查消息队列时处理该消息。若是在UI线程中调用的BeginInvoke，那么实际上这并不是异步的：当前操作后就会立即执行该调用。

这里我忽略了BeginInvoke所返回的Asynch结果对象。实际上，UI更新很少带有返回值。这会大大简化异步处理消息的过程。只需简单地调用BeginInvoke，然后等待委托在稍后的某个时候执行即可。但编写委托方法时需要格外小心，因为所有的异常都会在跨线程封送中被默认捕获。

在结束这个条目之前，我再来简单介绍一下控件的WndProc。当WndProc接收到了Invoke消息之后，将执行InvokeQueue中的每一个委托。若是希望按照特定的顺序处理事件，且你还混合使用了Invoke和BeginInvoke，那么可能会在时间上出现问题。可以保证的是，使用Control. BeginInvoke或Control.Invoke调用的委托将按照其发出的顺序执行。BeginInvoke仅仅会在队列中添加一个委托。不过稍后的任意一个Control.Invoke调用均会让控件开始处理队列中所有的消息，包括先前由BeginInvoke添加的委托。“稍后的某一时间”处理委托意味着你无法控制“稍后的某一事件”到底是何时。“现在”处理委托则意味着应用程序先执行所有等待的异步委托，然后处理当前的这一个。很有可能的是，某个由BeginInvoke发出的异步委托将在Invoke委托调用之前改变了程序的状态。因此需要小心地编写代码，确保在委托中重新

检查程序的状态，而不是依赖于调用Control.Invoke时传入的状态。

简单举例，如下版本的事件处理程序很难显示出那段额外的文字。

```
private void OnTick(object sender, EventArgs e)
{
    this.InvokeAsynch(() => toolStripStatusLabel1.Text =
        DateTime.Now.ToLongTimeString());
    toolStripStatusLabel1.Text += "  And set more stuff";
}
```


这是因为第一个修改会被暂存于队列中，随后在开始处理接下来的消息时才会修改文字。而此时，第二条语句已经给标签添加了额外的文字[①]。

Invoke和InvokeRequired为你默默地做了很多的工作。这些工作都是必需的，因为Windows窗体控件构建于STA模型之上。这个行为在最新的WPF中依旧存在。在所有最新的.NET Framework代码之下，原有的Win32 API并没有什么变化。因此这类消息传递以及线程封送仍旧可能导致意料之外的行为。你必须对这些方法的工作原理及其行为有着充分的理解。

① 这样，第一条语句中的时间就会覆盖掉第二条语句中的"And set more stuff"。——译者注

第3章

C#设计实践

3

软件设计方法通常与使用的编程语言无关。不过，为了能够编写出实际运行的应用程序，则必须使用编程语言将设计表达出来。在包括C#的各种语言中，表达同样的设计都有很多种不同的做法。若想提高软件的质量，那么则需要用最佳的方式来实现设计。这样才能让其他开发者更快理解你的意图，让软件易于维护并扩展。本章就将介绍如何用C#语言来最佳地实现一些常见的设计。

条目17：为序列创建可组合的API

我们的代码中经常会使用到循环。在很多程序中，操作一个序列的项目要比操作单一的项目更加常见。因此诸如foreach、for和while之类的关键字也会经常出现。于是，我们往往会创建出专门的方法，来接受一个集合作为输入，然后在方法中检查或修改其中的项，最后将操作过的集合返回。

问题在于，这类基于整个集合之上的操作策略的执行效率并不高。因为我们一般都不会在集合元素上仅执行一种操作，而更可能是需要在源集合与最终结果间执行多个操作。在这样的实现中，程序不得不创建很多集合（可能会相当大）来存放中间状态。且只有在上一步彻底完成之后才能开始进行下一步。此外，这样的设计意味着每个操作都需要重新遍历整个集合，这也影响了程序的执行效率。

另外一种做法是在一个循环中完成所有的操作，在一次迭代中生成最终的

结果。这种循环一次的做法提高了程序的执行效率，也降低了程序的内存占用，因为不需要保存每一个操作的中间结果。但它却牺牲了可重用性——若是将操作步骤分开的话，那么每一步的转换可以很容易地重用到其他算法中。

好在C#迭代器能让我们在操作序列的方法中根据调用者的需要一个一个地处理并返回元素。C# 2.0添加了yield return语句，用来支持以序列的方式返回对象集合。这类迭代器方法接受一个序列（IEnumerable<T>）作为参数，同时也返回一个序列（另一个IEnumerable<T>）。通过使用yield return语句，迭代器方法无需为序列中的所有元素都分配空间。这类方法将在必要时从输入序列中请求一个元素，且在调用者需要时才生成输出序列中的下一个元素。

这种用IEnumerable<T>作为输入和输出参数的方法是一种全新的理念，以至于很多开发者并没有足够地理解。不过这个做法却能带来很多好处。例如，你会很自然地用最佳的方法编写代码，让模块之间能够方便地组合起来，以提升可重用性。此外，你也可以仅用一次迭代就对元素进行多个操作，从而提高了程序的执行效率。每个迭代方法都将在调用者请求第N个元素时才开始执行代码，而不是预先就处理好。与传统方式相比，这个延迟执行模型（参见第5章条目37）能让算法使用更少的存储空间，并支持更自由的逻辑组合。且随着类库的不断演化，将来你甚至可以将不同的操作分配给不同的CPU核心来处理，进一步提高效率。此外，方法体内部通常不会对将要操作的对象的类型有过多的限制，因此完全可以使用泛型来进一步提高代码的可重用性。

为了能够演示迭代方法带来的好处，我们来看一个简单的示例。下面的这个方法接受一个整型数组作为参数，并将其中的重复元素过滤掉，随后输出：

```
public static void Unique(IEnumerable<int> nums)
{
    Dictionary<int, int> uniqueVals =
        new Dictionary<int, int>();

    foreach (int num in nums)
    {
        if (!uniqueVals.ContainsKey(num))
```

```
            {
                uniqueVals.Add(num, num);
                Console.WriteLine(num);
            }
        }
    }
```

这个方法很简单，但别人却无法重用其中的逻辑。在实际开发中，过滤集合中重复元素的操作极其有用，会经常重用在程序的其他部分中。

而若是按照如下方法编写该方法。

```
public static IEnumerable<int> Unique(IEnumerable<int> nums)
{
    Dictionary<int, int> uniqueVals = new
        Dictionary<int, int>();
    foreach (int num in nums)
    {
        if (!uniqueVals.ContainsKey(num))
        {
            uniqueVals.Add(num, num);
            yield return num;
        }
    }
}
```

那么，上述Unique方法将返回一个序列，其中包含了过滤掉重复之后的元素。下面就是其使用方法。

```
foreach (int num in Unique(nums))
    Console.WriteLine(num);
```

似乎并没有什么好处——甚至第二种做法还要更加低效一些，不过事实并非如此。下面我们就来在Unique方法中添加一些跟踪语句，让你看到其中的不同。

如下是修改后的Unique方法。

```
public static IEnumerable<int> Unique(IEnumerable<int> nums)
{
    Dictionary<int, int> uniqueVals =
        new Dictionary<int, int>();
    Console.WriteLine("\tEntering Unique");
```

```
        foreach (int num in nums)
        {
            Console.WriteLine("\tevaluating {0}", num);
            if (!uniqueVals.ContainsKey(num))
            {
                Console.WriteLine("\tAdding {0}", num);
                uniqueVals.Add(num, num);
                yield return num;
                Console.WriteLine
                    ("\tReentering after yield return");
            }
        }
        Console.WriteLine("\tExiting Unique ");
    }
```

下面是其输出。

```
        Entering Unique
        evaluating 0
        Adding 0
0
        Reentering after yield return
        evaluating 3
        Adding 3
3
        Reentering after yield return
        evaluating 4
        Adding 4
4
        Reentering after yield return
        evaluating 5
        Adding 5
5
        Reentering after yield return
        evaluating 7
        Adding 7
7
        Reentering after yield return
        evaluating 3
        evaluating 2
        Adding 2
2
```

```
        Reentering after yield return
        evaluating 7
        evaluating 8
        Adding 8
8
        Reentering after yield return
        evaluating 0
        evaluating 3
        evaluating 1
        Adding 1
1
        Reentering after yield return
        Exiting Unique
```

yield return语句在这里功不可没：该语句返回了一个值，同时还保留了内部迭代中的当前位置以及当前的状态。这样就得到了一个操作于整个序列上的方法：方法的输入和输出都是迭代器。在其内部，迭代能够在跟踪输入序列位置的同时，继续向输出序列返回下一个元素。这是一个持续方法，它将保留当前的状态，并在下次执行时从该位置继续开始。

这样让Unique()方法变成一个持续方法能带来两个好处。第一，它可以支持对每一个元素进行延迟求值。第二，也是更重要的，延迟执行能够让各个不同方法很方便地组合起来，这个优势很难用传统的foreach循环来实现。

注意，输入元素的类型（这里是整数）并没有对Unique()的逻辑有所影响。因此自然可以将其改写成泛型方法。

```
public static IEnumerable<T> UniqueV3<T>
    (IEnumerable<T> sequence)
{
    Dictionary<T, T> uniqueVals = new Dictionary<T, T>();
    foreach (T item in sequence)
    {
        if (!uniqueVals.ContainsKey(item))
        {
            uniqueVals.Add(item, item);
```

```
                yield return item;
            }
        }
    }
```

在需要将多个这种迭代方法组合起来时，迭代方法的强大之处将尽显无遗。例如你需要最终的输出过滤掉重复元素，并对每个元素取平方。平方操作很简单。

```
public static IEnumerable<int> Square(IEnumerable<int> nums)
{
    foreach (int num in nums)
        yield return num * num;
}
```

调用时只要将两个方法嵌套起来即可。

```
foreach (int num in Square(Unique(nums)))
    Console.WriteLine("Number returned from Unique: {0}", num);
```

无论需要调用多少个不同的迭代方法，程序只会循环一次。若用伪代码表示，那么算法的执行过程将如图3-1所示。

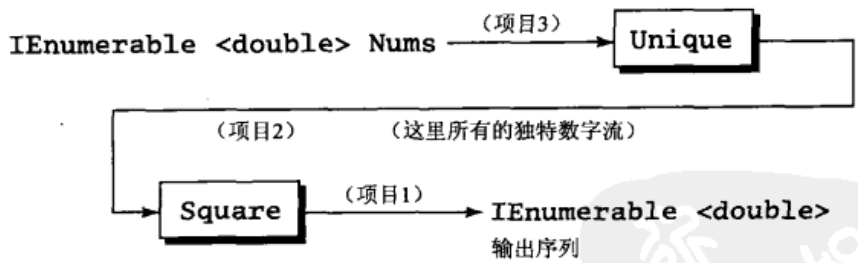

图3-1　项目将从一系列迭代方法中通过，当每个迭代方法都可以开始接受下一个项目时，该项目将从源序列中进入管道中。在同一时间，仅有一个元素处于处理的某一环节中

图3-1中的代码演示了多个迭代方法的强大组合能力。这些迭代方法仅需要遍历一次源序列，即可完成所有的工作。而作为对比，传统的实现却需要在每种操作中都遍历一次源序列。

在编写接受序列作为输入和输出的迭代方法时，还可以进行一些变化。例如，可以将两个序列组成一个新的序列。

```
public static IEnumerable<string> Join(
    IEnumerable<string> first,
    IEnumerable<string> second)
{
    using (IEnumerator<string> firstSequence =
        first.GetEnumerator())
    {
        using (IEnumerator<string> secondSequence =
            second.GetEnumerator())
        {
            while (firstSequence.MoveNext() &&
                secondSequence.MoveNext())
            {
                yield return string.Format("{0} {1}",
                    firstSequence.Current,
                    secondSequence.Current);
            }
        }
    }
}
```

如图3-2所示，Join将把来自两个输入序列中的各一个元素连接起来，并组成一个新的序列。Join也可以改为泛型写法，虽然要比Unique略微麻烦一些（参见第1章条目6）。

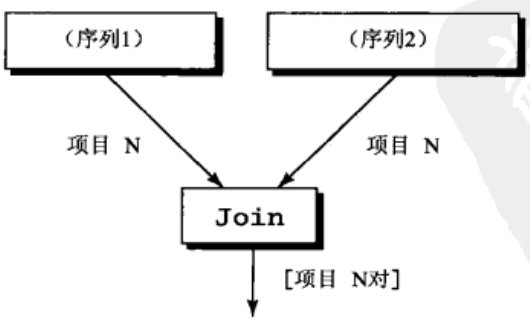

图3-2　Join方法将从两个不同的序列中分别取出一个元素。每次其调用者请求输出时，Join将从两个输入中各取出一个元素，将其连接起来，随后传送到输出序列中

Square()迭代方法演示了方法可以修改源序列中的元素。Unique()迭代方法演示了方法可以修改源序列本身：只有每个元素的第一个副本才会被返回。不过实际上，迭代方法并不会改变源序列，而是会生成一个新的序列作为输出。不过若序列中包含的是引用类型，那么其仍可能被修改。

将这些迭代方法组合起来，就会类似于孩子们的那种滑轨玩具。在一开始，你可以一个一个地放下小球，随后小球将陆续经过滑道和障碍，在一路上触发各种各样的操作。小球们并不会在每个障碍处聚集起来，头一个小球可能会远远领先于最后一个。而每个迭代方法都会对输入序列中的元素进行操作，并在输出序列中添加新元素。单一地看，迭代方法能够实现的功能很少。不过这些方法均基于一个输入和输出流，因此可以很容易地将其组合起来。若你已经有了足够的迭代方法，那么只需将其恰当地组合起来，即可实现一些非常复杂的算法。

条目18：将遍历和操作、谓词以及函数分开

前一条中介绍了使用yield return让方法操作于序列之上，而不是单个的数据类型。在实践的过程中，你会发现此类代码可以分为两大类：一种将修改序列中的元素，另一种将针对序列中的元素执行某个操作（action）。例如，你可能会需要仅获取序列中的那些满足指定条件的元素，也可能要间隔*N*个元素进行取样或跳过一组元素。

后一种操作与前一种操作完全不同。无论是要为数据生成各种报表，汇总特定的数据，还是修改集合中元素的属性，这种枚举遍历的模式均和将要执行的操作无关，因此两件事情应该分开处理。若将二者放在一起，那么不但增加了耦合度，还可能会造成代码的重复。

之所以很多开发者选择将很多操作写在一个方法中，是因为需要自定义的部分在逻辑中隐藏的比较深。若想实现这种位于算法内部的自定义逻辑，你可以将自定义的部分以方法调用或函数对象传入到算法中。在C#中，这可

以使用委托来实现。随着泛型和匿名委托的出现，在C# 2.0中我们可以相对容易地实现此类操作，从而支持各种自定义操作。在接下来的示例中，我将同时演示匿名委托语法（C# 2.0）以及C# 3.0中更加巧妙的lambda表达式语法。

匿名委托有两个常见的用法：函数（function）和操作（action）。此外，函数还有一个特例：谓词（predicate）。谓词是一个返回布尔值的方法，用来判断序列中的某个元素是否满足某个条件。这类委托都能对集合中的元素进行特定的操作，其应用范围非常广泛，考虑到这些，.NET框架中内建了这些委托的签名：Action<T>、Func<T、TResult>和Predicate<T>。

```
namespace System
{
    public delegate bool Predicate<T>( T obj);
    public delegate void Action<T>( T obj);
    public delegate TResult Func<T, TResult>(T arg);
}
```

例如，List<T>.RemoveAll()方法能够接受一个谓词。下面的这个调用就将列表中所有的“5”移除。

```
// 使用匿名委托语法的RemoveAll
myInts.RemoveAll(
    delegate(int collectionMember)
    {
        return collectionMember == 5;
    });

// 使用lambda表达式语法的RemoveAll
myInts.RemoveAll((collectionMember) => collectionMember == 5);
```

List<T>.RemoveAll()将在内部调用传入的委托方法（以匿名方法的形式给出），将其逐一应用到列表中的每个项目上。若该委托返回true，那么当前元素将被移除。（实际的实现要更复杂一些，因为RemoveAll()为了能保证在枚举的过程中不修改原始的列表，在实现中还使用到了内部的存储。当然，

这些都属于实现细节。）

操作方法也会陆续应用到集合中的每个项目上，List<T>.ForEach()就是个很好的实例。下面的这段代码就将集合中的每个整数输出到控制台上。

```
// 使用匿名委托语法的ForEach
myInts.ForEach(delegate(int collectionMember)
{
    Console.WriteLine(collectionMember);
});

// 使用lambda表达式语法的ForEach
myInts.ForEach((collectionMember) =>
    Console.WriteLine(collectionMember));
```

虽然看上去有些无聊，不过这段代码中体现的想法却可以应用到任意的场景中。匿名委托可用来定义将要执行的操作，随后ForEach方法将逐一把该匿名方法应用到集合中的每个元素上。

有了这两类方法，我们即可使用这些技术将复杂的操作应用到集合上。下面来看另一个示例，其中通过使用谓词和操作节约了很多代码。

这个筛选器方法使用了Predicate来执行测试。Predicate方法定义了哪个对象能够通过过滤，哪个将被过滤掉。根据条目17的建议（本章前面部分），我们可以创建一个泛型的过滤器，返回包含满足某种条件的元素的序列。

```
public static IEnumerable<T> Filter<T>
    (IEnumerable<T> sequence,
    Predicate<T> filterFunc)
{
    if (filterFunc == null)
        throw new ArgumentNullException
            ("Predicate must not be null");
    foreach (T item in sequence)
```

```
        if (filterFunc(item))
            yield return item;
}
```

输入序列中的每个元素都将通过Predicate方法的检验。若Predicate返回ture，那么该元素将输出到输出序列中。任何开发人员都可以针对某个类型编写检验用的方法，随后即可配合筛选器使用。

例如，如下代码即可对源序列进行采样，即在每条过*N*个元素后返回一个元素。

```
public static IEnumerable<T> EveryNthItem<T>(
    IEnumerable<T> sequence, int period)
{
    int count = 0;
    foreach (T item in sequence)
        if (++count % period == 0)
            yield return item;
}
```

这样即可将上述筛选器应用到某个序列之上，对其进行采样。

操作类型的委托可以跟任何枚举模式配合使用。这里，我们将创建一个转换方法，用来通过调用给定的方法将现有的序列转换成新的序列。

```
public delegate T Transformer<T>(T element);
public static IEnumerable<T> Transform<T>(
    IEnumerable<T> sequence, Transformer<T> method)
{
    foreach( T element in sequence)
        yield return method(element);
}
```

使用如下的方法调用Transform，即可将源序列中的每一个元素转换为其平方值。

```
// 使用匿名委托语法的Transform
foreach (int i in Transform(myInts, delegate(int value)
    {
        return value * value;
```

```
    }))
    Console.WriteLine(i);
// 使用lambda表达式语法的Transform
foreach (int i in Transform(myInts, (value)=> value * value))
    Console.WriteLine(i);
```

Transform方法并不强制你一定返回同样类型的元素。你可以让Transform返回其他类型的元素。

```
public delegate Tout Transformer<Tin,Tout>(Tin element);
public static IEnumerable<Tout> Transform<Tin,Tout>(
    IEnumerable<Tin> sequence, Transformer<Tin,Tout> method)
{
    foreach (Tin element in sequence)
        yield return method(element);
}
```

随后即可按照如下方式调用。

```
// 匿名委托语法
foreach (string s in Transform(myInts, delegate(int value)
{
    return value.ToString();
}))
    Console.WriteLine(s);

// lambda表达式语法
foreach (string s in Transform(myInts, (value)
    => value.ToString()))
    Console.WriteLine(s);
```

在条目17中可以看到，编写或使用这些方法并不难。这样做的核心理念是将遍历序列和对元素进行操作这两个步骤分开。这种使用匿名委托或lambda表达式创建的代码片断能够和很多不同的模式配合，并使用在各种不同的地方。你可以将各种对序列的修改封装成函数（包括其中的特例——谓词），还可以使用操作（action）委托（或类似的定义）来维护集合中的元素。

条目 19：根据需要生成序列中的元素

迭代方法的参数并不一定需要为序列。使用yield return语句可以创建出新的序列，即作为生成序列元素的工厂。但在创建时，不要一次性地创建出整个集合，而是根据需要生成序列中的元素。这样即可避免无谓地创建出调用者不需要的元素。

我们来看看一个简单示例，这段代码将生成一系列的整数。

```
static IList<int> CreateSequence(int numberOfElements,
    int startAt, int stepBy)
{
    List<int> collection =
        new List<int>(numberOfElements);
    for (int i = 0; i < numberOfElements; i++)
        collection.Add(startAt + i * stepBy);

    return collection;
}
```

这段代码可以正常使用，不过与使用yield return语句的代码相比，却存在着很多不足。第一，这种方法假设你要将结果存放在List<int>中。因此若客户需要将结果存放在其他结构，例如BindingList<int>中时，则必须再进行转换。

```
BindingList<int> data = new
    BindingList<int>(CreateSequence(100, 0, 5));
```

不过上述代码中可能会隐藏着潜在的问题。BindingList<T>的构造函数并没有复制源列表中的每一个元素，而是仍使用着这些元素的原始储存位置。若用来初始化BindingList<T>的这些元素同时还可以被其他代码访问，那么则可能会造成数据完整性错误。因为同样的数据有着多个引用。

不仅如此，一次性地创建整个列表就无法让客户代码根据特定的情况终止该过程。CreateSequence总是会生成指定数目的元素。若是需要因为分页等原因停下来，那么这个做法将无法实现。

此外，这个方法可能仅为若干个操作中的第一步（参见本章前面部分的条目17）。若是这样的话，那么该方法将成为整个管道的瓶颈：必须一次性地创建出所有的元素，然后才能开始执行下一步。

若是将上述方法改成泛型的迭代方法，那么就不会出现这些问题了。

```
static IEnumerable<int> CreateSequence(int numberOfElements,
    int startAt, int stepBy)
{
    for (int i = 0; i < numberOfElements; i++)
        yield return startAt + i * stepBy;
}
```

核心逻辑仍然没有变化：生成一系列的整数。

需要注意的是，在这个版本的代码中发生了一个变化。每次开始枚举其结果时，方法将重新生成序列中的整数。考虑到上述代码将永远生成同样的序列，所以这个变化也不会影响到程序的行为。这个版本的代码也没有假设客户需要何种类型的集合。因此若客户需要List<int>，那么可以将IEnumerable<int>传递到其构造函数中。

```
List<int> listStorage = new List<int>(
    CreateSequence(100, 0, 5));
```

需要保证的是让程序只生成一个序列的整数。按照这样的方式创建一个BindingList<int>集合。

```
BindingList<int> data = new
    BindingList<int>(CreateSequence(100,0,5).ToList());
```

上述代码看起来似乎有些低效，因为BindingList<T>已经提供了一个接受IEnumerable<T>作为参数的构造函数。不过实际上这并不低效，因为BindingList<T>将依然引用着源列表中的元素，而并没有创建新的副本。考虑到这些，程序需要使用ToList()来创建出CreateSequence所生成元素的一个新的副本，然后再让BindingList<int>去引用这个副本。

使用下面的方法，即可很容易地让CreateSequence停下来。但作为对比

的是，若你使用了CreateSequence的第一个实现，那么无论调用者是否希望其中途停止，CreateSequence均会完整地生成1 000个元素。而对于第二个实现，则将在不需要下一个元素的时候就立即停止继续创建。这样也就极大地提高程序的效率。

```
// 使用匿名委托
IEnumerable<int> sequence = CreateSequence(10000, 0, 7).
    TakeWhile(delegate(int num) { return num < 1000; });

// 使用lambda表达式
IEnumerable<int> sequence = CreateSequence(10000, 0, 7).
    TakeWhile((num) => num < 1000);
```

当然，你也可以使用别的条件来判断是否应该停止生成枚举序列，比如用户是否希望继续、从另外的地方获取输入，或是其他什么方法。这种枚举式的方法允许程序在枚举的任意时刻停止。也就是说，在客户代码每次需要使用一个元素时，迭代方法才会开始生成这个元素。

不过，只有在代码中实际创建了序列时，你才应该使用这种做法。若是集合已经存在，那么再用迭代的方式将其中的元素逐一返回并不会带来什么好处。

```
// 其中的各个值已经在其他位置给出
private List<string> labels = new List<string>();

// 没有意义，创建了另一个由编译器生成的枚举类
public IEnumerable<string> LabelsBad()
{
    foreach (string label in labels)
        yield return label;
}

// List<string>已经支持枚举器了
public IEnumerable<string> LabelsBetter()
{
    return labels;
}
```

所有的集合类型都会支持IEnumerable<T>，因此没有必要再画蛇添足。

一般来讲，我们最好仅在序列的使用者开始请求其元素时再开始实际的生成操作。这样，若是使用者仅需要一部分的元素，那么即可省下创建额外元素所用的时间。这部分时间可能很少，但若是创建元素的代价很高，那么时间也可能会很多。无论何种情况，根据需要生成序列中的元素总会让代码更加优雅。

条目20：使用函数参数降低耦合

开发人员通常都会选择最熟悉的语言特性来描述组件之间的契约。对于大多数开发者来说，一般会使用基类或借口来定义其他类型所需要的方法，然后根据这些接口编写代码。通常来说这没什么问题，不过使用函数参数则能够让其他开发者在使用你的组件和类库时更容易一些。使用函数参数意味着你的组件无需负责提供类型所需要的具体处理逻辑，而是将其抽象了出来，交给调用者实现。

我们都已熟悉了通过接口或抽象类来实现分离。不过有些时候，定义并实现接口仍旧显得过于笨重。虽然这样做符合了传统面向对象的理念，不过其他的技术则可以实现更简单的API。使用委托一样可以创建出契约，同时也降低了客户的代码量。

我们的目的是尽可能地将你的工作与客户使用者的代码分离开来，降低二者之间的依赖。如果不这样做的话，那么将给你和你的使用者都带来不少的困难。你的代码越是依赖于其他代码，那么就越难以单元测试或在其他地方重用。从另一方面考虑，你的代码越是需要客户代码遵守某类特定的模式，那么客户也会承受越多的约束。

使用函数参数即可降低你的组件和其使用者之间的耦合程度。不过，任何一种方法都会带来一定的代价。若你想强制地将结合非常紧密的组件和调用者代码分离开来，那么则可能会给你带来很多的工作，却只会让调用者得到些许的收

益。需要根据实际在二者之间进行平衡。此外，若想实现更松散的耦合——使用委托或是其他通信机制——也意味着你需要对编译器提供的某些检查进行特殊处理。

在现有的各种选择中，你可能会使用基类来定义与客户代码之间的契约。对于客户代码来说，这是最简单的一种方法。二者的契约很清晰：从某个特定的基类中继承，实现已知的抽象/虚方法即可。此外，你还可以为抽象基类提供一些通用的逻辑，这样其使用者就不必每次都要重写这部分代码了。

从你的组件方面来看，这种做法同样可以节省一些代码。你可以假设客户一定会实现某些特定的行为，因为若派生类没有实现所有的抽象方法，那么编译器将不会让其通过检查。虽然无法保证客户代码给出的实现的正确性，不过你至少知道客户代码中存在这样的方法。

不过，让客户代码实现某个基类是一种最为严格的约束。例如，所有的.NET语言都仅支持类之间的单继承关系。因此要求客户代码继承于一个基类将会带来很大的限制，它强制地给出了必须遵守的继承体系，且没有任何方法可以绕开。

使用接口作为契约要比依赖于基类显得宽松一些。一般来讲，我们会创建一个接口，并要求客户代码实现该接口。这个方法所创建的关系类似于使用基类所创建的关系。二者只有两个较大的不同：第一，使用接口并不会限制客户代码的继承体系。不过第二点就是，你也无法为客户代码提供任何公共的逻辑。

不过，这两种解决方案可能都会显得有些冗余。你真的需要使用到接口吗？或是用一种更加松散的方式，例如定义个委托作为签名会更好一些呢？

在条目17中，你已经看到了一些示例。例如List.RemoveAll()方法就接受了一个委托类型Predicate<T>。

```
void List<T>.RemoveAll (Predicate<T> match);
```

当然，.NET Framework的设计者也可以通过定义接口来实现该方法。

```
// 带来不恰当的额外耦合。
public interface IPredicate<T>
{
    bool Match(T soughtObject);
}
public class List<T>
{
    public void RemoveAll(IPredicate<T> match)
    {
        // 省略
    }
    // 其他API省略
}
// 第二个版本的使用方式相当复杂：
public class MyPredicate : IPredicate<int>
{
    public bool Match(int target)
    {
        return target < 100;
    }
}
```

看看条目17中的实现，你会发现这比将其定义在List<T>中要简单很多。通常，若是使用委托等更为松散的方式来定义契约的话，那么其他开发者使用起来将会更加容易。

之所以用委托而不是接口来定义契约，是因为委托并不是类型的基本属性。这与方法的个数无关——很多.NET Framework中的接口都只包含一个方法，例如IComparable<T>和IEquatable<T>等都是不错的定义。实现这些接口则意味着你的类型拥有了某些特定的属性：支持相互之间进行比较或是等同性判断。不过实现了这个假定的IPredicate<T>却并没有说明类型的特定属性。对于那些单个API来说，定义一个方法也就足够了。

通常，在你开始考虑定义接口或创建基类时，或许也可以使用函数参数来与泛型方法配合起来。条目17中给出了一个Join方法，用来合并两个序列。

```
public static IEnumerable<string> Join(
    IEnumerable<string> first,
    IEnumerable<string> second)
{
    using (IEnumerator<string> firstSequence =
        first.GetEnumerator())
    {
        using (IEnumerator<string> secondSequence =
            second.GetEnumerator())
        {
            while (firstSequence.MoveNext() &&
                secondSequence.MoveNext())
            {
                yield return string.Format("{0} {1}",
                    firstSequence.Current,
                    secondSequence.Current);
            }
        }
    }
}
```

在这里，你可以创建一个泛型方法，并使用函数参数来构造输出序列。

```
public static IEnumerable<TResult> Join<T1, T2,
TResult>(IEnumerable<T1> first,
    IEnumerable<T2> second, Func<T1, T2, TResult> joinFunc)
{
    using (IEnumerator<T1> firstSequence =
         first.GetEnumerator())
    {
        using (IEnumerator<T2> secondSequence =
            second.GetEnumerator())
        {
            while (firstSequence.MoveNext() &&
                secondSequence.MoveNext())
            {
                yield return joinFunc(firstSequence.Current,
                    secondSequence.Current);
            }
        }
    }
}
```

随后，调用者必须给出joinFunc的实现：

```
IEnumerable<string> result = Join(first, second, (one, two) =>
    string.Format("{0} {1}", one, two));
```

若是调用者还没有使用到C# 3.0，那么也可以用匿名委托来代替lambda表达式：

```
IEnumerable<string> result = Join(first, second,
    delegate(string one, string two)
    {
        return string.Format("{0} {1}", one, two);
    });
```

这样就更加降低了Join方法及其调用者之间的耦合。

条目19中的CreateSequence方法也能够这样修改。该CreateSequence方法用来创建一个整数的序列。将其修改为泛型方法，随后即可使用函数参数来给出生成序列的具体逻辑。

```
public static IEnumerable<T> CreateSequence<T>
    (int numberOfElements, Func<T> generator)
{
    for (int i = 0; i < numberOfElements; i++)
        yield return generator();
}
```

若想和原有代码的功能保持一致，那么调用代码要这样编写：

```
int startAt = 0;
int nextValue = 5;
IEnumerable<int> sequence = CreateSequence(1000,
    () => startAt += nextValue);
```

或者使用匿名委托的语法：

```
IEnumerable<int> sequence = CreateSequence(1000,
    delegate()
    {
        return startAt += nextValue;
    });
```

有些时候，你会需要在序列的每一个元素上都执行某个操作，最后返回一个汇总的数据。例如，下面这个方法将统计序列中所有整数的和。

```
public static int Sum(IEnumerable<int> nums)
{
    int total = 0;
    foreach (int num in nums)
    {
        total += num;
    }
    return total;
}
```

这时即可将该方法实现为一个泛型的累加器，其中将Sum的算法提取出来，用一个委托定义来代替。

```
public static T Sum<T>(IEnumerable<T> sequence, T total,
    Func<T,T, T> accumulator)
{
    foreach (T item in sequence)
    {
        total = accumulator(total, item);
    }
    return total;
}
```

按照如下方式调用。

```
int total = 0;
total = Sum(sequence, total, (sum, num) => sum + num);
```

或者使用匿名委托调用。

```
total = Sum(sequence, total, delegate(int sum, int num)
{
    return sum + num;
});
```

Sum仍旧有着不少限制。在上面的写法中，其必须使用与序列中元素、返回值和初始值同样的类型。而我们可能需要使用一些不同的类型。

```
List<Employee> peeps = new List<Employee>();
// 所有的员工对象都已经在其他位置添加完成
// 计算工资总数：
decimal totalSalary = Sum(peeps, 0M, (person, sum) =>
    sum + person.Salary);
```

我们来对Sum方法进行少量的修改，允许序列中的元素和累加的结果使用不同的类型。

```
public static TResult Sum<T, TResult>(IEnumerable<T>
sequence,
    TResult total,
    Func<T,TResult, TResult> accumulator)
{
        .
    foreach (T item in sequence)
    {
        total = accumulator(item, total);
    }
    return total;
}
```

使用函数参数能够很方便地将算法与特定的数据类型分离开来。不过，它在降低耦合的同时，也难免增加了一定的工作，因为需要处理组件之间通信中可能发生的错误。例如，若代码中定义了事件。那么在触发事件之前则必须确保该事件成员不为空。客户代码可能还没有创建过事件处理程序。在使用委托创建接口时也需要执行类似的检验。你需要考虑若是客户传入了一个空的委托，那么怎样才是正确的行为呢？应该是异常，还是正确的默认行为？若客户代码传入的委托抛出了异常，那么你能否捕获？如果能的话，应该怎样处理呢？

最后，在你准备从继承切换到委托来定义异常时，必须理解这种作法将和持有对象或接口的引用一样，有着相同的运行时耦合。若你的对象保存了传入的委托以备稍后调用，那么这个对象现在就控制了委托中对象的生命周期。这就可能延长了此类对象的生命周期。这和先让一个对象引用另一个对象（通过存放对接口或基类的引用），然后再使用的情况没什么不同。但在阅读代码时却更难发现。

在定义组件和其他客户代码的通信契约时，默认的选择仍是接口。抽象基类则能够提供一些默认的公共实现，让客户代码无需重复编写。而为方法定义

委托则提供了最大的灵活性，但也意味着你得到的支持会更少。总地看来，就是用更多的工作换来了更好的灵活性。

条目 21：让重载方法组尽可能清晰、最小化且完整

为方法创建越多的重载，也就越可能造成二义性。甚至编译器也无法找到最佳的方法重载。更严重的是，即使你进行了某些看似无关紧要的修改，也可能让代码调用了另外一个重载，进而导致意料之外的错误。

很多时候，使用较少的重载要更加容易一些。我们的目标应该是创建恰到好处的重载：既能够让开发者足够使用，又不要过多以至于让API难以理解，也让编译器无法找到最好的匹配。

重载中的二义性越多，那么其他开发者则会越加难以使用到类型推断等C#的最新功能。带有二义性的方法越多，编译器也就越难以判断哪个方法才是最合适的。

C#语言规范描述了用来判断最佳匹配方法的规范。作为C#开发者，你应该对这些规范有所理解。若你正在开发API，那么则应该对其有更深入的理解。你的一个重要职责就是让编译器在解析API时尽可能地不出现二义性导致的错误。更重要的是不要误导使用者，让其误解编译器将要选用的重载方法。

C#编译器可能会使用一个相当复杂的算法来判断是否有合适的重载，如果有的话，哪个才是更加合适的。当类型仅包含非泛型方法时，你会很容易地找到最佳的重载方法。不过随着不断添加各种变量，那么情况将变得越来越糟糕，也会越来越可能造成二义性。

几个条件会改变编译器解释这些方法的方式。解析的过程将受到参数的数量以及参数的类型的影响。是否需要考虑泛型方法，是否考虑接口方

法，以及是否考虑引入到当前的上下文中的扩展方法都会影响到重载的解析。

编译器能够在几个不同的位置搜寻可选的方法。在找到所有的候选方法之后，就会挑选出最佳的一个。若没有最佳的方法，或最佳方法多于一个，那么将导致编译错误。不过编译错误反倒最好，因为无法通过编译的代码自然无法发布。相反，最差的情况是，你和编译器对于应该选择哪个重载方法有着不同的理解，但编译器却不会为你而改变，因此自然埋下问题的隐患，导致期待之外的结果。

首先，需要保证的是带有相同名字的方法应该执行同样的操作。例如同一个类中的两个名为Add()方法应该实现同样的功能。若方法之间存在着语义上的不同，那么应该选用不同的名字。例如，你绝不应该编写如下的代码。

```
public class Vector
{
    private List<double> values = new List<double>();

    // 将某个值添加到内部的列表中
    public void Add(double number)
    {
        values.Add(number);
    }

    // 将一些值累加到序列中的元素上
    public void Add(IEnumerable<double> sequence)
    {
        int index = 0;
        foreach (double number in sequence)
        {
            if (index == values.Count)
                return;
            values[index++] += number;
        }
    }
}
```

两个Add()看似都很合理，不过却不应该将其放在同一个类中。不同的重载方法应该提供不同的参数列表，而不是提供不同的行为。

若是编译器调用了非你所期待的重载，那么遵守这个规则能够降低程序发生错误的可能性。若是两个方法执行的是同样的操作，那么无论调用哪个都不会有问题，不是吗？

当然，带有不同参数列表的重载方法之间的执行效率一般会有所不同。即使各个方法执行的是同样的工作，你也应该调用最为合适的一个。作为类的设计者，你完全可以很好地控制代码，避免二义性的发生。

若是方法有着类似的参数，且编译器需要在方法之间进行选择，那么就会发生二义性。最简单的情况是，每个重载均只有一个参数。


```
public void Scale(short scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(int scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(float scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(double scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}
```

提供了这些重载之后，即可避免了任何可能出现的二义性。所有的数值类型都列了出来（省略了Decimal，因为Decimal到double需要进行显式转换），这样编译器即可轻松地选择到最佳的版本。若你有一些C++背景，那么也许会建议使用泛型方法来改写。不过这样不行，因为C#泛型并不能实现C++模板中的此类操作。在C#泛型中，你不能假设类型参数中可以包含二义方法或操作符，而必须使用约束来给出你的期待（参见第1章条目1）。或者你会考虑使用委托来定义方法的约束（参见第1章条目6）。不过这样做只是将问题从一个地方移动到了另一个地方，即转移到了指定类型参数和委托的位置。

回到主题上来，若是遗漏了某些重载的话，比如：

```
public void Scale(float scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}

public void Scale(double scaleFactor)
{
    for (int index = 0; index < values.Count; index++)
        values[index] *= scaleFactor;
}
```

那么对于short和double类型，这样的代码则会存在二义性。从short到float以及从short到double均存在着隐式的转换。那么编译器将如何选择呢？若是没办法选择，那么你还要强迫开发者给出显式转换。这里，编译器将会选择float的重载。不过，很多开发者却无法一眼给出标准的答案。因此要尽量避免这样的问题：在给方法创建多个重载时，要确保大多数开发者能够立即判断出编译器将会选择哪个来调用。实现这个目的的最好方法就是提供一系列完整的方法重载。

单参数的方法其实非常简单，多个参数的方法将更加难以理解。下面的这两个方法就各自提供了一套参数列表。

```
public class Point
{
    public double X
    {
        get;
        set;
    }
    public double Y
    {
        get;
        set;
    }

    public void Scale(int xScale, int yScale)
    {
        X *= xScale;
        Y *= yScale;
    }

    public void Scale(double xScale, double yScale)
    {
        X *= xScale;
        Y *= yScale;
    }
}
```

那么，若是传入了int和float会怎样呢？传入int和long会怎样呢？

```
Point p = new Point { X = 5, Y = 7 };
// 注意第二个参数的类型是long
p.Scale(5, 7L); // 将调用Scale(double,double)
```

这时，两个参数中仅有一个能够完全符合参数列表的要求。而且对于另外一个参数来说，也没有提供隐式的转换操作，因此将无法选择出合适的重载。不过有些开发者可能不清楚这些，只能猜测代码将会调用哪个方法。

不过等等——重载方法的选择还能变得更复杂。我们可以在当前的情况中再添加一些不确定，看看将发生什么。若是基类中存在的方法要比派生类中的方法更加合适，那么又该如何选择呢？

```
public class Point
{
    // 省略了前面曾给出过的代码
    public void Scale(int scale)
    {
        X *= scale;
        Y *= scale;
    }
}
public class Point3D : Point
{
    public double Z
    {
        get;
        set;
    }

    // 没有override或new关键词。而是不同的参数类型
    public void Scale(double scale)
    {
        X *= scale;
        Y *= scale;
        Z *= scale;
    }
}
Point3D p2 = new Point3D { X = 1, Y = 2, Z = 3 };
p2.Scale(3);
```

这段代码中犯了很多的错误。首先，若要进行重写的话，Point应该将Scale()声明为虚方法。不过编写重写方法的人——我们叫她Kaitlyn——却并没有这样做：她创建了一个新的方法（而并没有隐藏掉基类的实现）。这样，她的类型的使用者将会调用到错误的方法。编译器在作用区域中找到两个方法，然后自然会判断Point.Scale(int)是最好的匹配（通过参数的类型）。这样，通过创建两个签名冲突的方法，Kaitlyn不经意间造成了这样的二义性。

若你想通过添加泛型方法来捕获所有没有覆盖到的情况，那么将会让情况变得更加糟糕。

```
public static class Utilities
{
    // 对double类型，选择Math.Max:
    public static double Max(double left, double right)
    {
        return Math.Max(left, right);
    }

    // float和int等类型将在这里处理:
    public static T Max<T>(T left, T right)
        where T : IComparable<T>
    {
        return (left.CompareTo(right) > 0 ? left : right);
    }
}
double a1 = Utilities.Max(1,3 );
double a2 = Utilities.Max(5.3, 12.7f);
double a3 = Utilities.Max(5, 12.7f);
```

第一个调用为Max<int>实例化了一个泛型方法，第二个调用使用了Max(double, double)，而第三个调用则为Max<float>实例化了一个泛型方法。这是因为对于泛型方法来说，总会有一个类型能够实现完美的匹配，而不需要任何的转换。若是编译器能够针对所有的类型参数找到合适的实例，那么这将成为最好的匹配方法。事实就是这样，即使别的方法能够通过隐式转换调用，泛型方法仍旧要略胜一筹。

但事情还远不止这么简单。扩展方法同样也将被纳入在重载匹配的考虑范围之内。若是某个扩展方法要比类型的成员方法更加匹配，那么又会如何呢？“更加匹配”意味着该方法的参数列表更加适合于调用者提供的参数。答案是，若扩展方法匹配得更好，即各个参数都不需要转换，那么编译器将选择该扩展方法而不是需要隐式类型转换的成员方法。但是若考虑到泛型方法强大的“适应能力”（总是能创造出最合适的匹配），那么扩展方法基本上不会在方法匹配中占得先机。

可以看到，编译器为了找到合适的方法，将会查询很多位置。方法存在的位置越多，那么候选方法也就越多，自然也会更容易产生二义性。虽然编

译器能够明确地判断出选择哪个方法，不过你的用户可能会产生疑惑。若是20个开发者中只有一个能够说出将会调用哪个重载，那么显然，你的API太过于复杂了。用户应该能够一眼就看出编译器将选择哪个重载，否则你的类库就需要改进。

若想为用户提供完整的功能，那么只要创建最少的、满足要求的重载即可。不要再添加新的了，否则只会增加类库的复杂性，而不会给使用者带来什么帮助。

条目22：定义方法后再重载操作符

每一种面向对象语言都对操作符重载提供了不同的支持。有些语言允许你重载几乎所有的操作符，而有些则根本不允许重载。C#在二者之间取了个折中：允许重载部分（但不是全部）的操作符。总体说来，C#语言设计者对于重载采取了一种较为宽松的策略：对于有些可以被重载的操作符，可能你永远都找不到合适的理由去提供重载。

不同的语言对待操作符重载的做法各不相同，通用语言规范（Common Language Specification，CLS）则采取了一种有趣的做法。重载过的操作符并没有CLS的内建支持，而是将每个操作符都映射到了一个特殊的方法上。这种做法就允许那些本来不支持重载操作符的语言也能够调用定义在支持重载的语言中的操作符。

这些重载操作符的名称可能并不友好。若是你在C#中重载了==操作符，那么其他语言中可能需要通过op_Equality方法来调用。有些语言甚至不支持外部调用其底层的重载操作符。例如，C#就不允许调用用户自定义的op_Assign方法。哪怕是在其他支持这个方法的语言定义了该操作符，C#编译器仍旧会执行简单的逐字节复制。

这类语言之间的不配合意味着，我们不应该仅仅依赖于操作符重载来创建公共接口。因为使用了你的类型的.NET开发者可能永远无法访问到这些操作

符。即使能够访问，也可能需要通过op_这类的方法调用。因此，我们应该使用CLS兼容的方式来定义公共接口，随后再根据需要，为了方便某些语言的使用者而添加合适的操作符重载。在本条目的接下来部分，我将介绍常见的可以重载的操作符及其适用场景。

其中最简单的一个就是重载==操作符。若你的类型重写了System.Object.Equals方法或实现了IEquatable<T>接口，那么则应该重载==操作符。C#语言规范规定，若重载了==操作符，那么必须同时重载!=操作符并重写System.Object.GetHashCode()方法。这三个方法必须按照同样的语义来判断对象的等同性，虽然语言本身无法强制实现。因此必须由开发人员来保证这些方法的正确性。即若是重载了==操作符，则必须同时实现IEquatable<T>接口，并重写System.Object.Equals()方法。

接下来的一系列常被重载的操作符是比较操作符。若类型实现了IComparable<T>接口，那么则应该重载<和>操作符。若你的类型还实现了IEquatable<T>接口，那么也应该重载<=和>=操作符。如==操作符一样，C#语言规范要求你必须成对地重载这些操作符。重载<操作符则必须重载>操作符，重载<=操作符则必须重载>=操作符。

注意在上面两个描述中，只有使用标准.NET BCL接口定义与某些操作符同样功能的操作后，才应该重载相应的操作符。这正是该条目中最重要的一点：只要你定义了重载操作符，那么也必须提供实现了同样行为的公共API。否则，对于那些使用不支持操作符重载语言的开发者来说，这个类型将显得不够友好。

除了等同性检查和比较操作之外，其他地方将很少用到重载操作符。在这些情况下，何时应该重载操作符并不会表现得那么清晰。

若你的类型表示的是数值数据，那么可以考虑重载算术操作符。此外，还需要为那些不支持操作符重载的语言提供额外的接口。重载了+操作符则意味着应该创建一个Add()方法。（例如，System.String中的Concat()方法与+所表示的语义相同。）

若是先定义需要支持的方法，然后根据这些方法重载操作符，那么重载的过程将会简单很多。例如，若你创建了Add()和Subtract()两个方法，那么则应该重载+和-两个操作符。若你创建了带有不同参数的多个Add()方法，那么则应该定义多个+操作符的重载。此外，因为加法满足交换律，所以+操作符也必须支持两种参数顺序。若A+B是合法的，那么B+A也必须是合法的。哪怕A和B是不同的类型（除了System.String），A+B和B+A也必须生成同样的结果。

因此，若是先定义了需要支持的方法，那么将会更加容易地判断某个操作表达式是否合法。你的类型需要同时支持单目和双目操作符吗？例如，是否支持A+(-B)表达式呢？若想支持该表达式，那么则必须提供-单目操作符。这样，类的使用者也会认为该表达式的结果和A-B一致。这也就意味着需要定义某个类似Negate()的方法，以便支持-单目操作符。若是从定义方法开始，那么是否需要定义Negate()呢？如果不需要的话，那么则意味着你无需支持此类操作符。

在C#中定义算术操作符比较简单，因为语言本身提供了限制。语言不支持重载算术赋值操作符（例如+=和-=等），但却会使用相应的算术运算符帮你完成，因此此类操作依然可以正常使用。

其他的操作符将很少有机会被重载，包括逻辑操作符、true操作符和false操作符等。

在支持操作符重载的语言中，提供必要的重载能够让代码的语法更加直观。不过，并不是每一种.NET语言都支持操作符重载。因此你应该从开始定义CLS兼容的公共接口开始，随后再考虑为那些支持操作符重载的语言提供必要的重载。最常见的需要重载的操作符包括==和!=，实现了IEquatable<T>接口后就应该重载这些操作符。很多C#开发人员都习惯于使用这些操作符，且认为其表现将会和IEquatable<T>.Equals()方法一致。若是你的类型实现了IComparable<T>接口，那么也应该重载<、>、<=和>=操作符。

若你的类型表示的是数值数据，那么应该考虑重载算术操作符。从实现你要支持的方法开始，随后添加必要的操作符重载，以便为那些使用支持操作符重载的语言（例如C#）的开发者提供便利。不过在任何情况下都需要保证的是，即使去掉了所有的操作符重载，你的类型依然可用。考虑到使用不支持操作符重载语言的开发者将无法访问到这些功能，所以在你提供重载操作符之前，先要定义必要的方法。

条目 23：理解事件是如何增加对象间运行时耦合的

事件似乎能够完全地去掉你的类和其需要通知的类型之间的耦合。因此，你会经常地提供对外发布的事件，随后让事件的任意订阅者监听这些事件。这样，你的类对其订阅者一无所知，也对订阅者没有任何限制。任何代码都可以订阅这些事件，并在事件触发时执行任意的操作。

不过事情并不像你想象中的那么简单。基于事件的API仍旧存在着耦合方面的问题。例如，有些事件参数包含着状态标记，允许客户控制你的类执行相应的操作。

```
public class WorkerEngine
{
    public event EventHandler<WorkerEventArgs> OnProgress;
    public void DoLotsOfStuff()
    {
        for (int i = 0; i < 100; i++)
        {
            SomeWork();
            WorkerEventArgs args = new WorkerEventArgs();
            args.Percent = i;
            EventHandler<WorkerEventArgs> progHandler =
                OnProgress;
            if (progHandler  != null)
            {
                progHandler(this, args);
            }
```

```
                if (args.Cancel)
                    return;
            }
        }
        private void SomeWork()
        {
            // 省略
        }
    }
```

这样，所有该事件的订阅者都会被耦合起来。假设有多个订阅者同时订阅了一个事件。那么，可能一个订阅者需要取消，而另一个则希望能继续执行下去。前面的这个定义不能防止发生这样的事情。因此，多个订阅者加上可变的事件参数就导致了订阅链上的最后一个订阅者覆盖了所有的其他订阅者的操作。你没有办法让事件只有一个订阅者，也没有办法保证你就是最后一个订阅者。因此解决方法只能是让事件参数中的标志位一旦被设置，就再不能被关闭。

```
public class WorkerEventArgs : EventArgs
{
    public int Percent
    {
        get;
        set;
    }
    public bool Cancel
    {
        get;
        private set;
    }
    public void RequestCancel()
    {
        Cancel = true;
    }
}
```

在这里我们通过修改公共接口解决了问题，不过这种做法却不一定能够适用于任何情况。若你能够确定仅有一个订阅者，那么或许可以选择另外一种通

信机制。例如，定义接口然后调用其实现方法。或者接收一个委托让外部代码传入。随后，这个唯一的调用者即可决定是否要支持更多的订阅者，以及如何协调各个订阅者的取消操作。

在运行时，事件源和事件订阅者之间还存在着另一种的耦合。事件源将保留一个委托，用来引用到事件订阅者。这样，事件订阅者对象的生命周期则会至少与事件源对象的生命周期一致。只要事件源还保留着该引用，且事件源依旧可达，那么事件的订阅者也会保持可达，无法被垃圾收集。这样，即使事件订阅者已经没有任何用处了，因为事件源保留了其引用，所以也无法将其垃圾收集。

因此，事件订阅者需要修改其析构的模式，在Dispose()方法中取消订阅该事件。否则，订阅者对象将不会被销毁，这也是运行时耦合带来的代价。虽然看上去已经最小化了编译期依赖，不过运行时耦合仍旧会造成代价。

基于事件的通信降低了类型之间的静态关联，不过却在运行时为事件源和事件订阅者带来了耦合。事件天生的多播特性让所有的订阅者不得不遵守同样的协议，来给事件源以反馈。在事件模型中，事件源保留着所有订阅者的引用，因此订阅者要么在析构时取消订阅事件，要么只能等待事件源被销毁时一起销毁。此外，事件源在析构时也必须释放所有关联了的事件处理程序。在考虑使用事件时，必须为此给出合理的设计。

条目 24：仅声明非虚的事件

与C#类型中的其他成员一样，我们也可以声明虚（virtual）事件。或许你会想，虚事件和其他虚成员没什么区别。不过，因为你可以使用类似字段的语法来声明事件，随后使用add和remove事件处理程序的语法，所以事情变得不那么简单。你会难免地在基类和派生类之间编写出有问题的事件处理程序。这甚至会导致一些很难被发现的严重bug。

我们可以修改一下前面例子中的WorkerEngine，用一个基类来定义基本的事件机制。

```
public abstract class WorkerEngineBase
{
    public virtual event EventHandler<WorkerEventArgs>
        OnProgress;

    public void DoLotsOfStuff()
    {
        for (int i = 0; i < 100; i++)
        {
            SomeWork();
            WorkerEventArgs args = new WorkerEventArgs();
            args.Percent = i;
            EventHandler<WorkerEventArgs> progHandler =
                OnProgress;
            if (progHandler  != null)
            {
                progHandler(this, args);
            }
            if (args.Cancel)
                return;
        }
    }

    protected abstract void SomeWork();
}
```

编译器将创建私有字段，以及公共add和remove方法。这些生成的代码将类似如下所示。注意其中在事件处理程序上应用了Synchronized属性（参见第2章条目13）。

```
private EventHandler<WorkerEventArgs> progressEvent;

public virtual event EventHandler<WorkerEventArgs> OnProgress
{
    [MethodImpl(MethodImplOptions.Synchronized)]
     add
```

```
        {
            progressEvent += value;
        }
        [MethodImpl(MethodImplOptions.Synchronized)]
        remove
        {
            progressEvent -= value;
        }
    }
```

因为该私有字段是有编译器生成的，所以你无法通过代码访问到。唯一访问该事件的方法就是通过其公开的声明。这种限制同样也存在于派生的事件上，你无法通过代码访问到其底层的私有字段。不过由于编译器能够访问到其生成的字段，因此编译器可以通过合适的代码来重写事件。实际上，派生事件将隐藏基类中的事件，派生类所做的工作与原始版本完全一致。

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        // 省略
    }
}
```

但在添加了override事件后，将会破坏掉代码。

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        Thread.Sleep(50);
    }
    // 被破坏。隐藏了基类中的私有事件字段
    public override event
        EventHandler<WorkerEventArgs> OnProgress;
}
```

这个重写事件的声明意味着在订阅该事件时，基类中的隐藏私有字段将不会得到赋值。其他类型订阅的是派生类中的事件，且派生类中没有办法触发该事件。

因此，若基类使用的是以类似字段方式声明的事件，那么重写该事件将隐藏基类中表示该事件的字段。基类中的代码可以触发该事件，不过却没有任何响应，因为订阅者关注的都是派生类的事件。这与派生类使用类似字段还是类似属性方式来声明事件无关。派生类将隐藏基类中的事件，所以基类中触发的事件自然不会调用到任何订阅者的代码。

派生类仅在使用add/remove访问器时才能正常工作。

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        Thread.Sleep(50);
    }
    public override event
        EventHandler<WorkerEventArgs> OnProgress
    {
        add
        {
            base.OnProgress += value;
        }
        remove
        {
            base.OnProgress -= value;
        }
    }
    // 重要：仅有基类才能触发该事件
    // 派生类无法直接触发该事件
    // 若派生类有必要触发事件，那么基类必须提供专门的保护（protected）方法
}
```

若是基类以类似属性的方式声明事件，则不会出现什么问题。

我们需要修改基类，让其包含一个受保护的事件字段，随后派生类即可修改基类中的变量了。

```
public abstract class WorkerEngineBase
{
    protected EventHandler<WorkerEventArgs> progressEvent;

    public virtual event
        EventHandler<WorkerEventArgs> OnProgress
    {
       [MethodImpl(MethodImplOptions.Synchronized)]
        add
        {
            progressEvent += value;
        }
        [MethodImpl(MethodImplOptions.Synchronized)]
        remove
        {
            progressEvent -= value;
        }
    }

    public void DoLotsOfStuff()
    {
        for (int i = 0; i < 100; i++)
        {
            SomeWork();
            WorkerEventArgs args = new WorkerEventArgs();
            args.Percent = i;
            EventHandler<WorkerEventArgs> progHandler =
                progressEvent;
            if (progHandler  != null)
            {
                progHandler(this, args);
            }
            if (args.Cancel)
                return;
        }
    }

    protected abstract void SomeWork();
}
```

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        // 省略
    }
    // 没问题。访问基类中的事件字段
    public override event
        EventHandler<WorkerEventArgs> OnProgress
    {
       [MethodImpl(MethodImplOptions.Synchronized)]
        add
        {
            progressEvent += value;
        }
        [MethodImpl(MethodImplOptions.Synchronized)]
        remove
        {
            progressEvent -= value;
        }
    }
}
```

不过，这段代码依旧限制了派生类的实现——派生类无法以类似字段的语法来声明事件。

```
public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        // 省略
    }
    // 破坏。其私有字段隐藏了基类的实现
    public override event
        EventHandler<WorkerEventArgs> OnProgress;
}
```

为了解决这个问题，你可以采取两种做法。第一，在创建虚事件时，不要使用字段方式的语法，无论是在基类还是在派生类中都不要使用。另外一个解决方法是创建一个虚方法，在定义虚事件后调用该方法触发。所有的派生类都必须重写该触发事件的方法以及虚事件。

```
public abstract class WorkerEngineBase
{
    public virtual event
        EventHandler<WorkerEventArgs> OnProgress;

    protected virtual WorkerEventArgs
        RaiseEvent(WorkerEventArgs args)
    {
        EventHandler<WorkerEventArgs> progHandler =
            OnProgress;
        if (progHandler  != null)
        {
            progHandler(this, args);
        }
        return args;
    }

    public void DoLotsOfStuff()
    {
        for (int i = 0; i < 100; i++)
        {
            SomeWork();
            WorkerEventArgs args = new WorkerEventArgs();
            args.Percent = i;
            RaiseEvent(args);
            if (args.Cancel)
                return;
        }
    }

    protected abstract void SomeWork();
}

public class WorkerEngineDerived : WorkerEngineBase
{
    protected override void SomeWork()
    {
        Thread.Sleep(50);
    }
```

```
    public override event
        EventHandler<WorkerEventArgs> OnProgress;

    protected override WorkerEventArgs
        RaiseEvent(WorkerEventArgs args)
    {
        EventHandler<WorkerEventArgs> progHandler =
            OnProgress;
        if (progHandler != null)
        {
            progHandler(this, args);
        }
        return args;
    }
}
```

不过，回顾一下代码，你会看到定义虚事件并没有带来任何的好处。触发事件的虚方法是你在派生类中唯一一个可以控制事件触发行为的地方。通过重写事件本身所能实现的功能都可以通过重写触发事件的方法来实现：你可以手工遍历每个委托，你可以为每个订阅者如何修改事件参数提供不同的处理过程，你甚至可以取消将要发出的事件。

第一眼看来，事件似乎为你的类和其他需要与其通信的类提供了一种松散耦合的接口。不过若是创建了虚事件，那么则难免同时为事件源和监听者之间建立了编译期和运行时的耦合。为了能让你的虚事件正常工作，我们还不得不添加一些额外的代码，不过这些代码往往反倒说明了你根本不需要定义虚事件。

条目25：使用异常来报告方法的调用失败

任何无法完成预定工作的方法都应该通过抛出异常来报告失败。错误代码容易被忽略，且分散各处的错误检查会污染正常的代码逻辑。不过异常不应该用于控制正常的逻辑流程。这就意味着，我们需要尽可能地保证在执行正常操作时，类库的公共方法不会频繁地抛出异常。运行时的异常很耗时，编写能够

良好处理异常的代码也比较困难。若你的API一定要调用者同时给出很多try/catch的话，那么这个类库显然不够健壮。

异常是失败报告机制的一种首选做法，因为与错误报告机制相比，它有着很多优势。返回的错误代码是方法签名的一部分，且经常需要携带一些非错误的信息。而与之对应的异常则仅有一个目的：报告调用失败。因为异常是一个类型，且你可以派生出自定义的异常类型，所以程序可以使用异常来携带足够说明该错误的信息。

返回的错误代码必须由方法调用者来处理。而作为对比，抛出异常将从调用栈中向上传递，直到找到合适的catch子句为止。这就让开发者能够将错误的发生和处理之处在调用栈上隔开来，处理方法更加自由。在这种分离的处理中不会丢失任何信息，因为异常对象已经将其完整地封装了起来。

此外，你无法忽略任意一个异常。若是程序没有合适的catch子句，那么抛出异常将终止应用程序的执行。在处理异常之前，程序无法继续运行，因为这会导致数据错误。

异常可用来报告方法调用的失败，但这并不代表任何无法完成预定操作的方法都必须通过抛出异常来结束，因为并不是所有的失败都是个异常。例如，若文件存在的话，那么File.Exists()将返回true，否则返回false；而若是文件不存在的话，那么File.Open()将抛出异常。区别很简单：File.Exists()本身就是用来判断文件是否存在的，即使文件不存在，方法也会成功执行。而只有当文件存在、当前用户可以读取文件、且当前进程可以打开文件以便读取时，File.Open()才算成功执行。对于File.Exists()来说，即使它告诉你的结果并不是你所期待的，但该方法仍旧完成了它的任务。而对于File.Open()来说，若文件不存在，那么方法执行将失败，因为你无法进行后续的读取操作。得到非期待中的结果并不代表着失败——方法的执行很成功，因为它给出了你请求的信息。

如何给方法命名将会很大程度上影响到方法成功与否的判断。执行操作的

方法应该有着清晰的命名，给出其中哪些操作一定要完成。作为对比，用来检查某些操作的方法应该在名称中体现出检查的含义。此外，提供用来检查的方法有助于降低异常出现的几率，让客户代码无需依赖异常来控制程序流程。编写异常安全的代码比较困难，此外，异常也比普通方法调用更加耗时。因此你应该提供相应的方法，允许用户在执行可能会失败的操作之前先进行检查。这样会让程序更加安全，且若是没有预先检查的话，仍旧可以通过抛出异常来给出失败信息。

在你需要编写可能会抛出异常的方法时，应该同时提供检查是否会抛出异常的方法。随后在内部实现中，你也可以使用这个检查方法来预先判断，并在判断失败时抛出异常。

假设你的某个工作类将在一些部件没有到位时执行失败。那么若API仅包含有调用的代码，而没有检查工具，则使用用者只能这样编写代码。

```
// 不推荐这样写
DoesWorkThatMightFail worker = new DoesWorkThatMightFail();
try
{
    worker.DoWork();
}
catch (WorkerException e)
{
    ReportErrorToUser(
        "Test Conditions Failed. Please check widgets");
}
```

因此，你应该补充一些公共方法，来让开发者能够在调用前检查是否满足条件。

```
public class DoesWorkThatMightFail
{
    public bool TryDoWork()
    {
        if (!TestConditions())
            return false;
        Work(); // 仍有可能由于失败抛出异常，不过可能性很小
        return true;
```

```
    }

    public void DoWork()
    {
        Work(); // 在失败时将抛出异常
    }

    private bool TestConditions()
    {
        // 实现省略
        // 这里检查是否满足条件
        return true;
    }

    private void Work()
    {
        // 省略
        // 执行实际操作
    }
}
```

这个模式需要你编写4个方法：2个公共方法和2个私有方法。TryDoWork()方法将验证所有的输入参数以及内部的对象状态，随后将调用Work()方法执行任务。而DoWork()方法则直接调用Work()方法，抛出可能出现的异常。这个模式在.NET中非常常用，因为抛出异常需要一定的代价，所以开发者可能希望通过预先检查来降低这部分开销。

添加了这几个额外方法之后，使用者即可更清晰地预先检查条件了。

```
if (!worker.TryDoWork())
{
    ReportErrorToUser
        ("Test Conditions Failed. Please check widgets");
}
```

在实际中，预先检查条件还可以包含更多的验证，例如检查参数以及内部状态等。这个模式常用在某个工作类将接受非信任的输入（例如用户输入、

文件输入或从未知代码传来的参数）时。这类失败比较常见，且可以在应用程序中给出解决办法。因此有必要提供一定的处理机制，让其不至于抛出异常。注意，我并没有保证Work()不会抛出任何异常。即使在检查了参数之后，仍旧可能出现其他预料之外的失败。这类失败应该用异常来报告，即使是在TryDoWork()方法中。

当你的方法无法完成预定任务时，那么有必要抛出合适的异常。不过，因为异常并不应该用来控制程序逻辑，所以你应该同时提供检查的方法，以便开发者在调用可能抛出异常的方法之前进行检查。

条目26：确保属性的行为与数据类似

属性存在着两面性。从外部看上去它就像是简单的数据元素。而在内部，则以方法的形式实现。这种两面性可能会让你创建出让用户感到迷惑的属性。使用你的类型的开发者会假设其中的属性会与直接访问数据成员一样。若你的属性不符合这样的假设，那么用户则可能产生误会。属性给人的感觉应该是和直接访问数据成员一样。

若是能够正确地使用属性来描述数据成员，那么即可符合客户开发者的期待。首先，客户开发者希望相连的两次调用属性的get访问器将会得到同样的结果。

```
int someValue = someObject.ImportantProperty;
Debug.Assert(someValue == someObject.ImportantProperty);
```

当然，多线程环境下该假设可能不会成立——无论是使用属性还是字段。不过其他情况下，重复地访问同样一个属性理应得到同样的结果。

此外，类型的使用者也不会期待属性的访问器做很多工作。调用某个属性的getter不应该影响到花费很多的时间。类似地，属性的set访问器可能会执行一定的验证，不过也不应该太过耗时。

为什么使用者会有这样的想法？这是因为他们将属性看作是一种数据。开发者喜欢在循环中重复地访问属性，在.NET集合类中也是如此。例如在使用for循环遍历数组时，你会重复地读取数组的Length属性。

```
for (int index = 0; index < myArray.Length; index++)
```

数组中的元素越多，那么访问Length属性的次数也就越多。若是每次访问该属性的时候都需要重新计算一下元素的个数，那么每个循环都会是平方量级的时间复杂度，于是没有人会轻易地使用循环了。

满足客户开发人员的期待并不难。首先，你可以使用隐式属性。*隐式属性*的包装非常少，仅仅包装了编译器自动生成的一个成员而已。其行为与直接访问字段最为相似。实际上，因为属性的访问器如此简单，所以经常会被编译器优化成内联的实现。若是以隐式属性的方式来提供属性，那么一定会符合客户的期待。

不过，若是你的属性还包含了其他的非默认行为，那么通常也不会造成什么问题。我们一般会在属性的setter中进行一定的验证，这不会影响到用户的体验。早些时候我曾经给出了LastName属性的这样一个setter实现。

```
public string LastName
{
    // 省略了getter
    set
    {
        if (string.IsNullOrEmpty(value))
            throw new ArgumentException(
            "last name can't be null or blank");
        lastName = value;
    }
}
```

这段额外的验证代码并没有破坏掉属性的任何行为。执行效率很高，同时又保证了对象的有效性。

此外，属性的get访问器通常会在返回之前执行一些计算工作。假设你有一个Point类，其中包含一个属性，表示该点与原点之间的距离。

```
public class Point
{
    public int X
    {
        get;
        set;
    }
    public int Y
    {
        get;
        set;
    }
    public double Distance
    {
        get { return Math.Sqrt(X * X + Y * Y); }
    }
}
```

这个简单的计算距离操作很快即可完成，用户也不会由于这样的实现而影响到性能。不过，若是Distance确实成为了瓶颈，那么也可以在第一次计算时将其缓存起来。当然，这也意味着在Point的其他属性发生变化时，必须重新计算Distance。（或者将Point改为不可变类型。）

```
public class Point
{
    private int xValue;
    public int X
    {
        get { return xValue; }
        set
        {
            xValue = value;
            distance = default(double?);
        }
    }
    private int yValue;
    public int Y
    {
        get { return yValue; }
```

```
            set
            {
                yValue = value;
                distance = default(double?);
            }
        }
        private double? distance;
        public double Distance
        {
            get
            {
                if (!distance.HasValue)
                    distance = Math.Sqrt(X * X + Y * Y);
                return distance.Value;
            }
        }
    }
```

若计算通过属性getter返回的值非常耗时，那么或许应该重新考虑一下你的设计。

```
// 不好的属性设计。getter需要长时间的执行
public class MyType
{
    // 省略
    public string ObjectName
    {
        get { return RetrieveNameFromRemoteDatabase(); }
    }
}
```

用户通常不会认为访问某个属性还需要从远程存储介质中获取数据。因此你需要修改这个公共接口来符合用户的预期。类型之间各不相同，因此实现必须依赖于类型的使用模式。这里，将其缓存起来则是个不错的办法。

```
// 一个可行的解决方案：一次获取，随后缓存
public class MyType
{
    // 省略
    private string objectName;
```

```
        public string ObjectName
        {
            get
            {
                if (objectName == null)
                    objectName = RetrieveNameFromRemoteDatabase();
                return objectName;
            }
        }
    }
```

若ObjectName只是偶尔需要，那么上述解决方案自然比较合适。因为它省去了在不需要该值时的获取操作，不过该属性的第一个调用者则需要承受比较大的开销。若程序中经常使用到ObjectName属性，且将其缓存起来也能满足要求的话，我们完全可以在构造函数中加载，并在需要时返回缓存中的数据。前一段代码也假定ObjectName允许被缓存。若程序的其他部分或其他程序有可能在同时修改远程的存储，那么就不可以使用这样的设计。

这种从远程数据库中获取数据，随后将修改提交回数据库的操作非常普遍。因此可以将这些操作以方法的形式提供出来，给出恰当的名字，从而满足用户的预期。下面就是另一个版本的MyType，它也不会给用户造成误导。

```
// 更好的解决方案：使用方法来管理缓存过的数据
public class MyType
{
    public void LoadFromDatabase()
    {
        objectName = RetrieveNameFromRemoteDatabase();
        // 其他字段省略
    }
    public void SaveToDatabase()
    {
        SaveNameToRemoteDatabase(objectName);
        // 其他字段省略
    }
```

```
    // 省略
    private string objectName;
    public string ObjectName
    {
        get { return objectName; }
        set { objectName = value; }
    }
}
```

自然，在这样修改之后，你完全可以使用隐式属性来实现ObjectName。

```
// 用隐式属性来实现
public class MyType
{
    public void LoadFromDatabase()
    {
        ObjectName = RetrieveNameFromRemoteDatabase();
        // 其他字段省略
    }

    public void SaveToDatabase()
    {
        SaveNameToRemoteDatabase(ObjectName);
        // 其他字段省略
    }

    // 省略

    public string ObjectName
    {
        get;
        set;
    }
}
```

不仅get访问器有可能超出客户开发者的意料，setter中也可能发生类似的情况。假设ObjectName是个可读写的属性。若setter直接将值写回了远程数据库，那么同样会让客户开发者无所适从。

```
// 不好的属性设计：setter太过于耗时
public class MyType
{
    // 省略
    private string objectName;
    public string ObjectName
    {
        get
        {
            if (objectName  == null)
                objectName = RetrieveNameFromRemoteDatabase();
            return objectName;
        }
        set
        {
            objectName = value;
            SaveNameToRemoteDatabase(objectName);
        }
    }
}
```

这里，setter所作的额外工作打破了使用者的几个固有假设。客户开发者不会想到一个setter将会调用远程数据库，其执行时间将大大超乎预期。写回数据库的过程也有可能出现很多问题，最终导致失败。

对于客户开发者来说，属性和方法是两种截然不同的特性。客户开发者希望访问属性可以很快完成，并认为属性表示了对象的当前状态。无论是在行为还是在执行性能上，他们都会希望属性与数据字段相似。若是你的属性不满足这些假设，那么则可能要修改现有实现，通过方法来表述那些不该有属性表示的操作。这样才能让属性回归到其本意，即作为了解对象状态的窗口。

条目27：区分继承和组合

面向对象语言中的一个很重要的组成部分就是继承。因此，很多开发人员都会无意识地在设计中大量使用到继承。不过很不幸，继承并不能够应用于每

一个设计中。继承将在两个类型之间创建极为紧密的耦合：派生类包含了基类中的所有成员。且由于派生类是基类的一个实例，因此派生类的对象能够应用于任何需要基类对象的地方。

继承表示了两种类型之间的一种非常牢固的关系。如果这并不是你设计的本意，那么则应该选用其他的方式，例如组合。组合允许外部对象仅公开内部对象的某些指定方法，而无须在公共和受保护的接口上实现如此紧密的耦合。

无论是类型之间的继承还是组合，都能够让一个类重用另一个类的实现。开发者通常会选用继承，因为一旦类型继承自另一个类型，那么也就自动拥有了基类所有的公共成员。


```
public class MyDerived : MyBase
{
    // MyBase的每个公共方法都成为了MyDerived的公共API
}
```

若是使用组合的话，那么必须重新实现基类的所有公共接口，然后再将实现委托给基类处理。

```
public class MyOuterClass
{
    private MyInnerClass implementation =
        new MyInnerClass();

    public void SampleImplMethod()
    {
        implementation.SampleImplMethod();
    }
    // 为MyInnerClass的每个公共方法重复类似的实现
}
```

这种显式地将方法委托给内部类的做法让你可以有选择地控制将哪些内部类的方法公开出来。而继承则总是公开基类中所有的公共成员。作为对比，组合允许你手工选择哪个内部类的方法将被公开在外部类中。除非显式添加，

否则内部类的方法将不会成为外部类的公共接口。

你的类的接口同样也不会随着内部类的接口变化而变化。若是使用继承，那么修改基类也就意味着修改了你的派生类。而在组合的实现中，你必须手工修改外部类，才能将内部类的新方法公开出来。若是内部类由第三方提供的话，那么这个做法将格外有用。你的用户在得到最新组件的同时，并不需要对其现有代码进行修改。

当然，你会发现使用组合则无法享受到多态带来的优势。你无法在需要MyInnerClass时传入MyOuterClass代替。不过若使用一些间接方法的话，也能实现类似的行为。你可以创建一个接口，来支持那些需要的方法。随后即可让外部类实现该接口。

```
internal interface IContract
{
    void SampleImplMethod();
    // 省略了其他方法
}

internal class MyInnerClass : IContract
{
    public void SampleImplMethod()
    {
        // 省略
    }
}

public class MyOuterClass : IContract
{
    private MyInnerClass impl = new MyInnerClass();
    // 其他省略

    public void SampleImplMethod()
    {
        impl.SampleImplMethod();
    }
}
```

调用者现在是通过IContract接口来访问MyOuterClass。这种接口和类之间的分离实现了足够的粒度，并降低了公共接口之间的耦合。你可以在接口中仅声明调用者所需要的成员。而其实现类以及外部类则可以提供更多的公共成员，不过此时调用者依赖的是接口。这和继承并不一样：MyOuterClass并不能传入到需要MyInnerClass参数的方法中。

组合还允许外部类在中包含多个内部类。你可以在外部类中通过自由组合来提供各种不同的功能。

```
internal class InnerClassOne
{
    internal void SampleImplMethodOne();
}

internal class InnerClassTwo
{
    internal void SampleImplMethodTwo();
}

public class MyOuterClass
{
    private InnerClassOne implOne = new InnerClassOne();
    private InnerClassTwo implTwo = new InnerClassTwo();

    public void SampleImplMethodOne()
    {
        implOne.SampleImplMethodOne();
    }
    public void SampleImplMethodTwo()
    {
        implTwo.SampleImplMethodTwo();
    }
}
```

当然，若从上述实现中抽象出接口，那么外部类即可用于那些需要这些特定接口的地方。

```
public interface IContractOne
{
    void SampleImplMethodOne();
    // 省略了其他方法
}

public interface IContractTwo
{
    void SampleImplMethodTwo();
    // 省略了其他方法
}

public class MyOuterClass : IContractOne, IContractTwo
{
    // 其他省略
}
```

此外，这种包含还允许你在运行时修改内部类。我们可以通过检查各种运行时条件，来选择并创建出某个合适的内部类。若想这样做，那么各种内部类必须实现同样的接口。这样即可以多态的方式处理不同的内部类。

```
public interface IContract
{
    void SomeMethod();
}

internal class InnerTypeOne : IContract
{
    public void SomeMethod()
    {
        // 省略
    }
}

internal class InnerTypeTwo : IContract
{
    public void SomeMethod()
    {
        // 省略
    }
}
```

```
public class MyOuterClass : IContract
{
    private readonly IContract impl;
    public MyOuterClass(bool flag)
    {
        if (flag)
            impl = new InnerTypeOne();
        else
            impl = new InnerTypeTwo();
    }

    public void SomeMethod()
    {
        impl.SomeMethod();
    }
}
```

这个概念可以根据需要进行扩展，从而充分满足你的需求。程序完全可以在多个内部类之间切换。若情况有所变化，你也可以通过创建新的内部对象来修改其行为。这并不会带来太多的工作，因为你不必修改类型对外的接口。这里可以看作借用了重载的概念，在各个内部类之间执行自定义的分发操作。

上述这些示例主要查看了两种策略之间的不同。随着时间的推移，组合会越来越体现出它的优势，因为它降低了派生类中的接口耦合。这是因为派生类将自动包含所有基类中的公共方法。若基类开发者添加了新的方法，那么你的类型也会自动得到这些方法，不管你是否真的需要。使用组合则可以由你自己决定公开出哪些方法。虽然必须手工编写代码，但不至于泄露你的类型的公共接口。

在总结之前，别误以为这个条目是让你不要使用继承。若对象模型恰好反映了Is A（是一个）关系，那么继承将能提供更好的重用性，并节省大量的代码。此外，若能够良好地建模，也会让后续开发者更容易理解你的代码。.NET BCL在Windows窗体和Web窗体类库中大量使用了继承，因为继承模型对于UI控件来说非常合适。而在.NET BCL的其他部分，继承则不是如此常用，开发

者更经常选择的是组合。

我们应该在设计时考虑更多的选择，灵活地使用组合和继承。在你需要让类型重用其他类型的实现时，应该使用组合。若类型表示的是Is A关系，那么继承则是更好的选择。若想能够公开内部对象的实现，组合需要更多的工作，不过得到的好处就是能够更精确地控制你的类型以及将要使用的类型之间的关系。而使用继承则意味着在任何情况下，你的派生类型都是基类的一个特例。

第4章

C# 3.0语言增强 4

C# 3.0语言中添加了一些全新的特性。虽然很多特性的本意是用来支持LINQ（Language Integrated Query，语言集成查询），但这些特性仍旧能够用于很多非LINQ查询的场景中。本章就将介绍这些语言特性，解释如何使用这些新技术来解决开发中的问题，并列举出使用中常见的一些误区。

条目28：使用扩展方法增强现有接口

扩展方法让C#开发者可以为接口定义行为。你可以在接口中仅定义最小化的功能，然后通过扩展方法来增强该接口的功能。此外，我们不仅仅能够定义API，还可以用其增强现有行为。

System.Linq.Enumerable类中包含了很多实例。System.Enumerable中为IEnumerable <T>类型定义个超过50种扩展方法，例如Where、OrderBy、ThenBy和GroupInto等。以IEnumerable<T>的扩展方法的形式定义这些方法有着很大的优势。首先，这些功能均不需要修改现有已经实现了IEnumerable<T>的类型。实现IEnumerable<T>的类也无需担负更多的责任，仍旧是实现GetEnumerator()就够了。且IEnumerator<T>也仍只需提供Current、MoveNext()和Reset()的实现。随后，通过创建扩展方法，C#编译器即可保证所有的集合类型均能支持查询操作。

你也可以使用同样的模式。例如，IComparable<T>遵守了一种由C语言流

传至今的习惯。如果left < right，那么left.CompareTo(right)的返回值将小于0。若left > right，那么left.CompareTo(right)的返回值将大于0。若left和right相等，那么left.CompareTo(right)将返回0。这个模式相当常用，因此人们都比较熟悉，不过这并不代表它就是个很好的做法。若是能写成类似left.LessThan(right)或left.GreaterThanEqual(right)这样，那么可读性将有很大提高。有了扩展方法之后，这个想法很容易就能实现。

```
public static class Comparable
{
   public static bool LessThan<T>(this T left, T right)
        where T : IComparable<T>
    {
        return left.CompareTo(right) < 0;
    }

    public static bool GreaterThan<T>(this T left, T right)
        where T : IComparable<T>
    {
        return left.CompareTo(right) < 0;
    }

    public static bool LessThanEqual<T>(this T left, T right)
        where T : IComparable<T>
    {
        return left.CompareTo(right) <= 0;
    }

    public static bool GreaterThanEqual<T>(this T left, T right)
        where T : IComparable<T>
    {
        return left.CompareTo(right) <= 0;
    }
}
```

这样，若添加了合适的using语句，那么所有实现了IComparable<T>的类看上去都有了三个额外的方法。实现者仍仅需要提供CompareTo方法，随后其使用者即可使用那些额外的方法了。

在对待应用程序中创建的接口时，你也应该遵循同样的模式。即仅定义

满足需要所必需的功能，随后再使用扩展方法添加辅助的功能。与那些庞大的接口相比，使用扩展方法能让实现接口更加简单，同时也拥有完善的功能。

通过这样将接口和扩展方法组合起来，我们即可为接口的方法提供默认的实现。这种做法让类能够重用那些基于接口的实现。在你需要定义接口时，可以查看一下能够用现有接口成员间接实现的方法。这些方法就应该以扩展方法的形式实现，进而被所有的接口实现者重用。

若是某个类需要定义一个方法，而这个类所实现的接口中也给出了同名的扩展方法，那么也许会导致一些意料之外的行为。虽然根据方法解析的规则，类中的方法将会被优先调用，不过这个解释发生在编译期。若是程序中通过接口来调用该方法，那么将会调用到接口上的扩展方法，而不是定义在类型中的版本。

我们来看一个简单实例。下面的这个接口用来在对象上做一个标记。

```
public interface IFoo
{
    int Marker
    {
        get;
        set;
    }
}
```

你可以用一个扩展方法来让标记的值自增。

```
public static class FooExtensions
{
    public static void NextMarker(this IFoo thing)
    {
        thing.Marker += 1;
    }
}
```

在代码中，可以这样使用该扩展方法。

```
private static void UpdateMarker(IFoo item)
{
    item.NextMarker();
}

public class MyType : IFoo
{
    #region IFoo Members
    public int Marker
    {
        get;
        set;
    }
    #endregion

    // 省略
}

// 其他位置
MyType t = new MyType();
UpdateMarker(t); // t.Marker == 1
```

一段时间之后，另一个开发者创建了一个新类型，并实现了一个自己的NextMarker方法（语义与原有扩展方法不同）。这样，MyType就有了另一个版本的NextMarker实现。

```
// MyType的第二个版本
public class MyType : IFoo
{
    public int Marker
    {
        get;
        set;
    }

    public void NextMarker()
    {
        Marker += 5;
    }
}
```

随后，应用程序的行为将发生很大的变化。例如，如下代码将把Marker改写成5。

```
MyType t = new MyType();
t.NextMarker(); // t.Marker == 5
```

你无法彻底避免这个问题，不过可以尽力降低它的影响。这个示例只是用来演示不好的做法。在真正的代码中，扩展方法应该与同名的类方法有着同样的语义和行为。若你能针对某个特定类型提供更好、更有效的方法，那么则应该以同名方法的形式给出。但必须保证其行为是一致的，这样才不会影响到程序的正确性。

当你发现在某个设计中，需要很多类同时实现一个接口时，那么可以考虑在接口中仅定义最小的成员。然后以扩展方法的形式提供其他的方法。这就让日后将要实现该接口的开发者省去大量的工作，并得到很多的便利。

条目 29：使用扩展方法增强现有类型

应用程序中经常会使用到现有的泛型类型。比如我们会创建某些特定的集合：List<int>、Dictionary<EmployeeID, Employee>等。之所以创建了这些集合，是因为应用程序需要特定类型的集合，且需要使用为这些特定的类型定义专门的行为。若希望尽可能不影响到现有的系统，我们可以通过为这些特定类型创建扩展方法来实现。

在System.Linq.Enumerable中，你可以看到这个模式。条目28也介绍了这种通过为Enumerable<T>编写扩展方法来提供常用操作的模式。此外，Enumerable类型中还包含有一系列方法，专门操作于实现了IEnumerable<T>的某些特定类型之上。例如，一些算术类型的方法仅应用在了数值序列（IEnumerable<int>、IEnumerable<double>、IEnumerable<long>和IEnumerable<float>）之上。下面就是专门为IEnumerable<int>提供的几个扩展方法。

```
public class Enumerable
{
    public static int Average(this IEnumerable<int>
        sequence);
    public static int Max(this IEnumerable<int> sequence);
    public static int Min(this IEnumerable<int> sequence);
    public static int Sum(this IEnumerable<int> sequence);

    // 其他方法省略
}
```

了解了这个模式之后，即可将其应用到你的项目中，为现有类型提供扩展。假设你正在开发一个电子商务应用程序，如你需要通过电子邮件发送优惠券给一系列客户，那么方法的签名可能会如下所示。

```
public static void SendEmailCoupons(this
    IEnumerable<Customer>
    customers, Coupon specialOffer);
```

类似地，你还可以找到在过去的一个月内没有任何订单的客户。

```
public static IEnumerable<Customer> LostProspects(
    this IEnumerable<Customer> targetList);
```

若不使用扩展方法，那么则需要从某个指定的泛型类型中派生出一个新类型。这样，上述这两个操作于Customer集合的方法将要这样实现。

```
public class CustomerList : List<Customer>
{
    public void SendEmailCoupons(Coupon specialOffer);
    public static IEnumerable<Customer> LostProspects();

}
```

当然，这样做没有什么问题，不过与IEnumerable<Customer>上的扩展方法相比，这种做法存在着更多的限制。方法签名的不同就是其中的一个原因。扩展方法使用IEnumerable <Customer>作为参数，而派生类则基于List<Customer>，即强制要求使用一种特定的存储模型。因此，这样的做法

将无法与一系列迭代方法组合使用（参见第3章条目17）。这样做会让方法的使用者受到不必要的约束，也就是误用了继承。

另一个推荐使用扩展方法来实现的原因是各个查询之间组合的方式。例如，LostProspects()可以按照如下的方式编写。

```
public static IEnumerable<Customer> LostProspects(
    IEnumerable<Customer> targetList)
{
    IEnumerable<Customer> answer =
        from c in targetList
        where DateTime.Now - c.LastOrderDate >
            TimeSpan.FromDays(30)
        select c;
    return answer;
}
```

条目34将介绍为什么我们推荐在查询中使用lambda表达式，而不是方法。以扩展方法的形式实现这些功能意味着这些方法可以以lambda表达式的方式重用在查询中。这样即可重用整个查询，而不是仅重用其where子句中的谓词。

回顾一下你编写的应用程序或类库，你会发现很多地方均使用了自定义的类型来存放数据。你应该检查这些类型，并判断这些类型在逻辑上真正需要的是哪些方法。随后最好能够以扩展方法的形式，将这些功能添加到该类型实现的泛型接口之上。因为若是要让类型本身包含所有行为的话，那么就要不得不将一个泛型的实例修改成复杂的专门类型。此外，这种实现方式能够在最大程度上降低存储模型和功能实现之间的耦合。

条目 30：推荐使用隐式类型局部变量

C#中添加了隐式类型局部变量来支持匿名类型。另外一个隐式类型局部变量的出现原因是，有一些查询的结果是IQueryable<T>，而另一些则是IEnumerable<T>。若将IQueryable<T>强制转换为IEnumerable<T>集合，

那么将失去所有IQueryProvider（参见第5章条目42）带来的特别优化。使用var还能够加深开发者对代码的理解。因为与JobsQueuedByRegion这样顾名思义的变量名字相比，变量的类型Dictionary<int, Queue<string>>实在不能带来太多额外的帮助。

我非常喜欢使用var来声明局部变量，因为这样可以让开发者始终关注在程序的最重要部分（其语义），而不是某个变量的类型。开发者输入更多的代码并不代表类型安全就得到了更好的保证。在很多情况下，IQueryable和IEnumerable之间的区别根本不会给开发者更多的帮助。不仅如此，若是你明确地告诉了编译器变量的类型，那么反倒有可能在稍后修改查询时导致类型方面的错误（参见第5章条目42）。很多情况下使用隐式类型局部变量要更好一些，因为编译器将会帮你选择出更好的类型。但在另一些情况下，过度使用var也会降低代码的可读性。使用隐式类型变量还有可能会导致一些难以察觉的转型bug。

我们先从可读性开始。很多时候，在初始化语句中，局部变量的类型就能体现得很清楚。

```
var foo = new MyType();
```

任何一个合格的开发者都能立即判断出变量foo的类型。类似地，大多数工厂方法也很清晰。

```
var foo = AnotherType.CreateObject();
```

不过在某些情况下，返回值的类型也许并不能从方法命中看到。

```
var foo = someObject.DoSomeWork(anotherParameter);
```

当然，这个例子或许有些刻意。且代码中的方法的名字均应该尽量反映出其返回值。不过即使在这个刻意的例子中，一个良好的变量名也能让大多数开发者理解代码的含义。

```
var HighestSellingProduct =
    someObject.DoSomeWork(anotherParameter);
```

虽然没有任何类型信息，不过大多数开发者仍旧能够看出其类型为Product。

根据DoSomeWork方法签名的不同，HighestSellingProduct可能并不是个Product。也许是Product的一个派生类，甚至是Product所实现的接口。编译器将根据DoSomeWork方法的签名来设定HighestSellingProduct的具体类型。即使运行时的类型不是Product也没有问题。当编译期类型和运行时类型有所冲突时，编译器总会占得先机。除非强制转换，否则无法得到其实际的类型。

这样，我们就开始遇到了var所引发的可读性问题。使用var来声明某个接受方法返回值的变量将有可能让开发者难以理解你的代码。人们在阅读代码时会假设一个类型。在运行时，这个类型可能是对的。但编译器却无法检查对象的运行时类型。编译器只能检查编译期类型，并根据方法声明来推断该局部变量的类型。这里有所不同的是，变量的类型将由编译器给出。若你自己声明了类型，那么其他开发者可以看到该类型。而作为对比，若是编译器选择了类型，那么开发者不一定能够与编译器的判断保持一致。因此，若是以这样的方式编写代码，那么开发者和编译器将可能对变量的类型有着不同的判断。这样就会导致出现一些本可以避免的问题。

在配合内建的数值类型使用隐式类型局部变量时，也会出现一些问题。内建的数值类型之间有很多的转换关系，可分为两类：拓宽转换，例如float到double，这种转换总是安全的。而另一种缩小转换，例如从long到int，则可能会丢失精度。因此，若是显式地声明了所有数值变量的类型，那么将会对使用的类型有更多的控制，也让编译器能够在进行某些危险转换时给出警告。

考虑如下代码。

```
static void Main(string[] args)
{
    var f = GetMagicNumber();
    var total = 100 * f / 6;
    Console.WriteLine("Type: {0}, Value: {1}",
```

```
            total.GetType().Name, total);
    }
```

那么total是什么类型的呢？这取决于GetMagicNumber的返回类型。下面有5种输出，每一种都是由GetMagicNumber的不同声明造成的。

```
Declared Type: Double, Value: 166.666666666667
Declared Type: Single, Value: 166.6667
Declared Type: Decimal, Value: 166.6666666666666666666666667
Declared Type: Int32, Value: 166
Declared Type: Int64, Value: 166
```

之所以类型有所不同，是因为编译器在推断f的类型时的处理过程影响到了推断出的total的类型。编译器将把f的类型设定为GetMagicNumber()的返回值类型。因为计算total的公式已经固定了下来，因此编译器可以把该公式与f的类型放在一起考虑，随后在计算的过程中使用适合f类型的方式。这也就导致了不同的类型将得到不同的答案。

这并不是语言本身的问题。C#编译器总会按照你的请求完成工作。若是使用了局部变量推断，那么则意味着你相信编译器要比你更了解该类型。随后，编译器将基于赋值语句的右半部分尽力作出判断。需要注意的是，在配合内建数值类型使用时必须格外小心，因为数值类型之间存在着很多的隐式转换。此外，考虑到不同的数值类型有着不同的精度，所以不单是可读性受到了影响，结果的正确性也可能无法保证。

当然，并不是因为使用var才导致了这个问题。问题的成因在于GetMagicNumber()的返回值类型难以判断，且还需要考虑内建的转换。即使在方法中不使用f变量，仍旧会出现同样的问题。

```
static void Main(string[] args)
{
    var total = 100 * GetMagicNumber() / 6;
    Console.WriteLine("Type: {0}, Value: {1}",
        total.GetType().Name, total);
}
```

这里的问题在于，开发者不能看到GetMagicNumber()的返回值的实际类型，因此也无法判断将要执行哪些数值转换。

我们来将其与明确指定期待的GetMagicNumber()的返回类型比较一下。这样，编译器即可告知你的假设是否正确。若从GetMagicNumber()的返回值类型到f的声明类型之间存在着隐式转换，那么不会出现什么问题。例如将f声明为decimal，GetMagicNumber()返回int。不过若是无法进行隐式转换的话，你将得到一个编译错误，进而就必须修改类型。通过这样的提醒，让你不得以地去查看代码，从而理解需要进行的转换操作。

前面这个例子演示了局部变量类型推断有可能会让代码的维护者难以理解。编译器总是按照同样的方式工作，并能够检查类型。不过，开发者却并不能很容易地看出将使用哪条规则和哪个转换。这样，局部变量类型推断自然将阻碍人们对代码的理解。

局部变量类型推断并没有改变C#静态类型的本质。为什么呢？首先，你需要理解局部变量类型推断和动态类型并不是一个概念。用var声明的变量并不是动态的，只不过其类型将由赋值语句右边表达式的类型来决定。虽然你没有告诉编译器使用哪个类型，但编译器将做出自己的推断。因此若你期待中的类型与编译器选择的类型不一致，那么代码就会出现问题。

不过，有些时候编译器要比你更加聪明一些。例如如下这段代码，将从数据库中取得姓名以A开头的顾客的名字。

```
public IEnumerable<string> FindCustomersStartingWith
    (string start)
{
    IEnumerable<string> q =
        from c in db.Customers
        select c.ContactName;

    var q2 = q.Where(s => s.StartsWith(start));
    return q2;
}
```

这段代码有着很严重的性能问题。原来的那个包含了所有客户姓名的查询，被开发人员错误地声明成了IEnumerable<string>。由于该查询将配

合数据库使用，因此其实际类型为IQueryable<string>。而若是强制将其转换为IEnumerable<string>，那么你将失去很多有用的信息。IQueryable<T>继承自IEnumerable<string>，因此编译器不会给出任何警告。但随后在执行接下来的过滤时，将使用Enumerable.Where而不是Queryable.Where。在这种情况下，编译器本可以选择出比你指定的类型（IEnumerable<string>）更好的类型（IQueryable <string>）。若是没有从IQueryable<string>的隐式转换，那么编译器将给出错误。不过由于IQueryable<T>继承自IEnumerable<string>，所以编译器将允许这个转换，最终导致性能受到影响。

第二个查询没有调用Queryable.Where，而是调用了Enumerable.Where。这在性能上有着很大的影响。在条目38（第5章）中，你会看到IQueryable能够将多个查询表达式树组合成单一的操作，并可以在远程数据库服务器上一次执行完毕。不过在这里，第二部分查询（where子句）将把输入看作IEnumerable<string>。这个变化产生的影响很大，因为现在只有第一部分查询会执行在远程服务器上。你将得到数据库中所有的客户姓名。随后第二条语句（where子句）将在本地检查该完整的客户姓名列表，然后才能得到并返回满足条件的姓名。

将其与下面这个版本对比。

```
public IEnumerable<string> FindCustomersStartingWith
    (string start)
{
    var q =
        from c in db.Customers
        select c.ContactName;

    var q2 = q.Where(s => s.StartsWith(start));
    return q2;
}
```

这里，q是一个IQueryable<string>。编译器能够根据查询的数据源推断出返回值的类型。第二条语句与第一条组合起来，为其添加了where子句，构成

了一个新的、更加复杂的表达式树。仅在调用者开始访问查询结果时，程序才开始获取实际的数据。过滤查询结果的表达式将传递给数据源执行，因此返回的序列将仅包含那些满足条件的姓名。这样做不但大大降低了网络流量，查询也变得更加高效。

这个非常神奇的变化是因为q被声明（由编译器）成了IQueryable<string>，而不是IEnumerable<string>。扩展方法不能为虚方法，因此运行时分发不可用。扩展方法是静态方法，因此编译器将基于变量的编译期类型而不是运行时类型来选择最佳的方法。这里也不会出现延迟绑定功能。即使运行时类型包含了能够匹配该方法调用的成员，编译器对其也一无所知，自然不会选择调用该方法。

值得注意的是，扩展方法中可以检查其参数的运行时类型。因此扩展方法可以根据其参数的运行时类型创建不同的实现。实际上，当传入的参数实现了IList<T>或ICollection<T>接口时，Enumerable.Reverse()正是使用了这种方法来增强了性能（参见第1章条目3）。

作为开发者，你必须判断让编译器声明变量的类型是否会影响代码的可读性。若无法一眼看出某个局部变量的类型，且这一点影响其他人理解代码，那么最好显式给出该变量的类型。不过在很多情况下，代码均能很好地表达出变量的含义。在前面的一个示例中，所有人都知道q是一个包含了姓名（恰好为字符串类型）的序列。其初始化语句已经很清晰地表明了语义。这一点常见于表示查询表达式的变量中。在变量的语义非常明确时，你可以使用var。而在前面某些代码中，当初始化表达式无法给出明确的语义，必须使用显式类型声明时，则应该避免使用var。

简而言之，除非开发者（包括一段时间后的你自己）需要看到变量的类型才能理解代码，否则最好用var来声明局部变量。本条目的标题也是“推荐”，而不是“总是”。我还建议你显式声明所有数值变量的类型（int、float、double等），而不要使用var。此外，在泛型中也应该使用类型参数（例如T和TResult），而不是var。除此之外的所有情况中都可以使用var。输入更多的类型信息——

以便显式给出类型——并不能保证代码的类型安全或提高可读性。若是指定的类型不恰当的话，反而可能会影响程序执行的性能。

条目31：使用匿名类型限制类型的作用域

在需要简单的数据容器用来存放一些中间结果时，我们应该考虑使用匿名类型。匿名类型能够大大节约你的工作，且编译器能够生成一些你无法做到的代码来支持额外的功能。此外，匿名类型也不会带来很多人想像中的代码量膨胀。考虑到了这些，匿名类型自然应该得到更广泛的应用。只有在需要为类型添加行为，或是指定其名字，以便能让其作为方法参数或类型成员时再考虑编写专门的类型。

我们先从节约代码开始。例如这样一条赋值语句：

```
var aPoint = new { X = 5, Y = 67 };
```

这条语句告诉了编译器几件事情。首先，语句说明你需要一个内部（internal）密封（sealed）类。该类型是个不可变类型，拥有两个只读的属性：X和Y。

这样，编译器将为你生成类似如下的代码。

```
internal sealed class AnonymousMumbleMumble
{
    private readonly int x;

    public int X
    {
        get { return X; }
    }

    private readonly int y;
    public int Y
    {
        get { return y; }
    }
```

```
    public AnonymousMumbleMumble(int xParm, int yParm)
    {
        x = xParm;
        y = yParm;
    }
}
```

编译器将通过继承于泛型的Tuple类型（参见第1章条目9）来定义实际的类型，并给出你需要的属性名称。

与手工编写相比，我更希望让编译器替我完成，因为这会带来很多好处。最简单的一条是，编译器的速度更快。手工无法在输入表达式的同时就创建好这个新类型的定义。第二，编译器将为重复的任务生成完全一样的定义。作为开发者，我们经常会不小心遗忘某些东西。这段代码非常简单，因此不太可能会发生什么错误，但仍旧无法完全避免。而编译器则不会犯此类的错误。第三，让编译器生成代码将减少需要手工维护的代码量。开发人员无需阅读这些代码，也无需猜想为什么要编写、代码做了什么以及何处使用了它之类的问题。因为编译器生成了这些代码，因此不需要开发者费心思考太多。

使用匿名类型中最大的不足是你无法知道该类型的名字。因为你无法指定类型的名称，因此匿名类型不能作为方法的参数或返回值。不过仍旧有办法可以操作单一的对象或包含匿名类型的序列。我们可以在方法内部编写处理匿名类型的方法或表达式。这时，处理的过程必须以lambda表达式或匿名委托的形式来给出，这样即可在创建匿名类型的方法中定义处理操作。若使用包含函数参数的泛型方法，那么可以创建配合匿名方法使用的方法。例如，使用这样的一个转换方法就能够将x和y的值乘以2：

```
static T Transform<T>(T element, Func<T, T> transformFunc)
{
    return transformFunc(element);
}
```

这样，即可将匿名类型传递给Transform方法。

```
var aPoint = new { X = 5, Y = 67 };
var anotherPoint = Transform(aPoint, (p) =>
    new { X = p.X * 2, Y = p.Y * 2});
```

自然，复杂的算法需要更加复杂的lambda表达式，且可能需要调用很多个泛型方法。不过其原理和前面的简单示例一样。这也正是将匿名类型作为存放内部计算结果的优势。匿名类型的作用域仅在创建该类型的方法中。匿名类型无法将某个算法第一阶段的结果存放起来，并传递给第二阶段继续使用。通过泛型方法和lambda表达式，我们可以在匿名类型的作用域（本方法内）中对其进行任意的转换操作。

此外，因为临时结果是存放在匿名类型中的，因此这些类型不会污染应用程序的命名空间。你完全可以让编译器来创建这些简单类型，而无需强迫让其他人在理解这些类型之后才能理解你的程序。匿名类型的作用域仅在其被声明的方法中，因此匿名类型也会让其他开发者清楚地看到该类型仅在这一个方法中被用到。

你会注意到在前面我介绍编译器如何定义匿名类型时使用了一些较为模糊的说法。在你需要匿名类型时，编译器将生成“类似如下的代码”。实际上，编译器添加的一些功能是你无法手工实现的。匿名类型是不可变的类型，同时能支持对象初始化器（object initializer）语法。若手工编写不可变类型的话，那么必须让构造函数能够初始化类型中的每个字段或属性。而这样手工编写的不可变类型却无法支持对象初始化器语法，因为属性没有提供可被外部访问的setter。不过在创建匿名类型的实例时，你却能够且必须使用对象初始化器语语法。编译器将创建一个公共构造函数来设定每个属性，然后再在构造函数中设定每个属性。

例如，若你这样初始化一个对象：

```
var aPoint = new { X = 5, Y = 67 };
```

那么编译器将会翻译成这样。

```
AnonymousMumbleMumble aPoint = new AnonymousMumbleMumble (5, 67);
```

若想让不可变类型同时也支持对象初始化器语法，那么只能使用匿名类型。手工编写的类型无法得到此类编译器的支持。

最后需要说明的是，使用匿名类型并不会带来太大的运行时开销。你也许会猜想，每次创建匿名类型时，编译器都要重新定义一个新的类型。好在编译器要更加聪明一些。即当你创建相同的匿名类型时，编译器将重用从前使用过的那个类型。

这里有必要更加准确地说明一下编译器对待不同位置中同样匿名类型的处理方法。不过前提条件肯定是这些匿名类型均定义在同一个程序集中。

此外，若想让编译器将两个匿名类型认为是相同的，那么这两个类型的各个属性名称及其类型必须一致，且属性的声明顺序也必须相同。下面的两个声明将生成两个不同的匿名类型。

```
var aPoint = new { X = 5, Y = 67 };
var anotherPoint = new { Y = 12, X = 16 };
```

若是属性的声明顺序不同，那么将会创建出两个不同的匿名类型。若想使用同一个匿名类型，那么必须保证的是在任何时候创建对象时，所有属性的声明顺序都完全相同。

在结束这个条目之前，还有必要提及一个特别的情况。因为匿名类型的等同性是靠着其中几个属性值的比较得来的，所以匿名类型可以用来作为组合键使用。例如，若想将所有的客户按照其销售人员和邮政编码这两个属性联合起来分组，那么可以使用如下查询：

```
var query = from c in customers
            group c by new { c.SalesRepresentative,
                c.ZipCode };
```

该查询将生成一个字典，其键为SalesRepresentative和ZipCode的组合，值为Customer列表。

匿名类型并不像你看到的那样难以理解，若能合理使用的话，也不会影响代码的可读性。一般来讲，若是你需要保留一些中间结果，且这些结果可以用不可变类型表示的话，那么应该使用匿名类型。而若是你还需要为这些类型添加行为，那么此时则应该考虑创建专门的类型来表示这些数据。编译器将为匿

名类型自动生成符合规范的代码。通过匿名类型，程序也能很清楚地告诉其他开发者，这个类型仅会在当前方法上下文及其调用的泛型方法中用到。

条目32：为外部组件创建可组合的API

对于定义在其他类库中的方法，你可以使用扩展方法来扩展其组合能力。所谓可组合的方法，是指该方法属于输入的某个参数的成员方法，并在该方法内部进行转换后返回计算结果。可组合方法能够很容易地按照特定的顺序组合起来，从而完成指定的算法需求。

不幸的是，很多.NET Framework API并没有按照这样的方法设计。例如TryParse方法，将返回一个布尔值和解析后的值。TryParse返回了两个信息：解析是否成功以及解析后的值。因为你必须在执行下一步操作之前检查解析是否成功，所以TryParse不能和其他方法组合使用。

下面这一小段程序将读取一个包含数值数据的、由逗号分割的文件（CSV），并返回一个数组的列表，其中的每个数组将包含文件中每一行的各个数值。

```
private static IEnumerable<int[]>
    ParseNumbersInCsv(TextReader src)
{
    List<int[]> values = new List<int[]>();
    string line = src.ReadLine();
    while (line != null)
    {
        List<int> lineOfValues = new List<int>();
        string[] fields = line.Split(',');
        foreach (var s in fields)
        {
            int dataValue = 0;
            bool success = int.TryParse(s, out dataValue);
            // 没有检查是否组合成功，因此用0表示失败
            lineOfValues.Add(dataValue);
        }
        values.Add(lineOfValues.ToArray());
        line = src.ReadLine();
```

```
        }
        return values;
    }
```

上述程序并没有什么问题，不过其实现方式却不那么漂亮，且无法很好地与其他部分组合。此外，这种命令式的实现也让你损失了其他的一些优势。例如若文件非常大，且其中可能包含了某个标记值，表示此时可以停止继续处理。那么若想满足新的需求，则需要修改方法并添加参数和逻辑，进而增加了整个方法的复杂性。选择了命令式的方式，也就意味着若想修改方法，那么同时也改变了方法执行的功能。

我们接下来所要做的就是修改该API，让其支持各种操作之间的组合，其中有些操作还允许调用者传入不同的表达式来实现不同的行为。我们先从修改原有方法内部的将字符串转为可空整数的逻辑开始。

```
public static int? DefaultParse(this string input)
{
    int answer;
    return (int.TryParse(input, out answer))
        ? answer : default(int?);
}
```

也许还要提供一个变体，对于不合法的输入返回默认值。

```
public static int DefaultParse(this string input,
    int defaultValue)
{
    int answer;
    return (int.TryParse(input, out answer))
        ? answer : defaultValue;
}
```

这样，DefaultParse即可用于解析输入文字的一行之上。

```
public static IEnumerable<int> ParseLine(this string line,
    int defaultValue)
{
    string[] fields = line.Split(',');
    foreach (string s in fields)
        yield return (s.DefaultParse(defaultValue));
}
```

继续提供一个ParseLine方法，返回IEnumerable<int?>。

```
public static IEnumerable<int?> ParseLine(this string line)
{
    string[] fields = line.Split(',');
    foreach (string s in fields)
        yield return (s.DefaultParse());
}
```

同时再编写一个扩展方法，逐一返回每一行。

```
public static IEnumerable<string> EatLines
    (this TextReader reader)
{
    string line = reader.ReadLine();
    while (null != line)
    {
        yield return line;
        line = reader.ReadLine();
    }
}
```

有了这些API之后，就可以用一个简单的查询生成一系列的行，其中每一行都包含了一个序列的元素。

```
var values = from l in src.EatLines()
             select l.ParseLine(0);
```

上面这个查询返回的是IEnumerable<IEnumerable<int>>。若想使用可空int的话，那么使用下面的查询将返回IEnumerable<IEnumerable<int?>>。

```
var values = from l in src.EatLines()
             select l.ParseLine();
```

若还有其他的用处，这些方法也可很容易地进行扩展。例如，比如文件中有一个特殊的字符串来标出输入已终止，那么按照如下方式修改，即可让程序读到该终止符时停止继续解析。

```
var values = (from l in src.EatLines()
              select l.ParseLine()).TakeWhile(
              (lineOfValues) =>
              !lineOfValues.Contains(default(int?)));
```

之所以可以很容易地实现这个需求，是因为使用扩展方法重写了API签名，将其改为了可组合的方式。在你的代码中也可以使用同样的组合功能。注意，虽然方法的签名有所变化，但其语意却仍保持一致。在创建扩展方法时必须考虑到这些，不要让调用者误解扩展方法的行为。

扩展方法若是改变了现有API的语义，那么将会导致一系列的问题。首先，改变了方法签名的语义。例如，实例方法无法调用在空引用之上，因为.NET运行时将直接抛出空引用异常，而不是调用你的代码。你无法预先判断这种情况并作出其他反应。这样，当你为某个类型添加某个扩展方法，且该扩展方法处理了空引用的情况，那么就会出现潜在的问题。因为在编译器解析方法调用时，扩展方法的优先级最低，所以若是稍后其他开发人员为该类型添加了同名的实例方法，那么该实例方法将被优先调用，导致原来正常执行的代码抛出空引用异常。

可以看到，最常见的一种滥用扩展方法的情况就是让扩展方法处理空对象。

```
// 错误做法。不要修改处理空引用的语义
public static int StorageLength(this string target)
{
    if (target == null)
        return 0;
    else
        return target.Length;
}
```

该方法的使用者将困惑于其行为。因为.NET运行时将首先检查空对象指针，若确实为空的话，将保证在调用任何实例方法之前均抛出空引用异常。而这个扩展方法却可以正常处理空值，这也自然让代码变得难以理解、维护以及扩展。

同样应该避免的是，编写一些很有可能会失败的扩展方法。这种情况经常出现在对.NET集合接口进行扩展的时候。比如下面这个方法就添加AddRange()来扩展了ICollection<T>接口。

```
public static void AddRange<T>(this ICollection<T> coll,
    IEnumerable<T> range)
{
    foreach (T item in range)
        coll.Add(item);
}
```

若实现了该ICollection的类型恰好支持Add操作，那么不会出现什么问题。不过若是在数组上调用该方法，编译器则会抛出异常。

```
string[] sample = { "one", "two", "three", "four", "five", };
// 将抛出NotSupportedException异常
sample.AddRange(range);
```

大多数实现了ICollection<T>接口、并支持Add方法的类型均提供了其自己的AddRange实现。因此这个扩展方法对增强功能方面没有太多帮助，反倒让方法的调用者可能会得到异常。因此，编写这些有可能失败的扩展方法并没有太大的意义。

若是你的扩展方法的性能远低于用户的期盼，那么也会造成使用者的不满意。下面的这个用来反转IList<T>接口集合的扩展方法是个很不错的实现。

```
// 很好的实现，不会影响到性能
public static IEnumerable<T> Reverse<T>(this IList<T> sequence)
{
    for (int index = sequence.Count-1; index >= 0; index--)
        yield return sequence[index];
}
```

而同样操作与IEnumerable<T>之上的扩展方法，性能上则与上面的版本有着天壤之别。

```
// 不好的实现，创建副本影响了性能
public static IEnumerable<T> Reverse<T>
    (this IEnumerable<T> sequence)
{
    IList<T> temp = new List<T>(sequence);
    return temp.Reverse();
}
```

IEnumerable<T>的版本将需要额外占用与原有集合同样大小的内存，且执行时间也较长。大多数用户将（错误地）认为Reverse的性能是线性的。若是操作于IList<T>之上，情况的确如此。不过若是将该操作应用到一个太过泛化的接口之上，以至于你不得不使用一种比较低效且复杂的实现，那么将会误导用户，因为他们会以为你的做法很高效。

此外，因为实现了IList<T>或ICollection<T>的类型同时也实现了IEnumerable<T>，那么考虑到编译器解析方法调用的规则，你的实现将变得更加低效。因为只有在编译期类型实现了IList<T>时，编译器才会调用IList<T>版本的Reverse()方法。若是编译期类型声明为IEnumerable<T>，哪怕运行时类型实现了IList<T>，也会调用该低效的版本。

System.Linq.Enumerable类型提供了Reverse方法，接受IEnumerable<T>参数。.NET Framework的开发者通过检查参数的运行时类型来避免了这个问题。若是类型实现了IList<T>或ICollection<T>，那么将使用更快以及更少内存占用的算法。

通过扩展方法，我们可以很好地增强现有的API签名，提高其组合性，以便配合复杂的表达式使用。不过在创建扩展方法增强现有API组合能力时，需要保证扩展方法不会隐藏原有API抛出的异常，也不会降低原有API的性能。

条目 33：避免修改绑定变量

下面这一小段代码演示了在闭包中使用外部的变量，随即又在外部修改这些变量时情况。

```
int index = 0;
Func<IEnumerable<int>> sequence =
    () => Utilities.Generate(30, () => index++);

index = 20;
foreach (int n in sequence())
    Console.WriteLine(n);
Console.WriteLine("Done");
```

```
index = 100;
foreach (int n in sequence())
    Console.WriteLine(n);
```

这段代码将打印出20～50一串数字，随后是100～130一串数字，你也许会对此感到惊讶。不过在本条目的稍后部分，我将给出编译器为其生成的代码，并解释出现这种情况的原因。这个行为确实有它存在的意义，某些时候可以很好地利用。

为了将你的查询表达式转换成可执行代码，C#编译器做了很多工作。虽然C#语言中提供了很多全新的功能，不过这些功能均可被编译成与.NET CLR 2.0兼容的IL指令。查询语法依赖于新的程序集，但却不依赖于任何新的CLR功能。C#编译器将把查询和lambda表达式转换成静态委托、实例委托或闭包。编译器将根据lambda表达式中的代码选择一种实现方式。选择哪一种方式依赖于lambda表达式的主体（body）部分。这看上去似乎是一些语言上的实现细节，但它却会显著地影响到你的代码。编译器选择何种实现将可能导致代码的行为发生微妙的变化。

并不是所有的lambda表达式都会生成同样结构的代码。对于编译器来说，最简单的一种是为如下形式的代码生成委托。

```
int[] someNumbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
IEnumerable<int> answers = from n in someNumbers
                           select n * n;
```

编译器将使用静态委托来实现该n * n的lambda表达式，其生成的代码就像手工编写的一样。

```
private static int HiddenFunc(int n)
{
    return (n * n);
}
private static Func<int, int> HiddenDelegateDefinition;

// 使用方法
int[] someNumbers = new int[] { 0, 1, 2, 3, 4, 5,
    6, 7, 8, 9, 10 };
if (HiddenDelegateDefinition == null)
```

```
{
    HiddenDelegateDefinition  = new
        Func<int, int>(HiddenFunc);
}
IEnumerable<int> answers =
    someNumbers.Select<int, int>( HiddenDelegateDefinition);
```

该lambda表达式的主体部分并没有访问任何实例变量或是局部变量。lambda表达式仅仅访问了它的参数。对于这种情况，C#编译器将创建一个静态方法，作为委托的目标。这也是编译器执行的最简单的一种处理方式。若是表达式可以通过私有的静态方法实现，那么编译器将生成该私有的静态方法以及相应的委托定义。对于前面简单例子中的情况以及仅访问了类型中静态变量的表达式，编译器都会采取这样的做法。

这个示例lambda表达式仅是委托所指向的方法调用中的最简单情况。接下来的一种同样较为简单的情况是，lambda表达式需要访问类型的实例变量，但无需访问外层方法中的局部变量。

```
public class ModFilter
{
    private readonly int modulus;

    public ModFilter(int mod)
    {
        modulus = mod;
    }

    public IEnumerable<int> FindValues(
        IEnumerable<int> sequence)
    {
        return from n in sequence
               where n % modulus == 0 // 新添加的表达式
               select n * n; // 与前面的例子一样
    }
}
```

在这种情况下，编译器将为表达式创建一个实例方法来包装该委托。其基本概念和前一种情况一致，不过这里使用了实例方法，以便读取并修改当前对象的状态。与静态委托的例子一样，这里编译器将把lambda表达式转化成你早

已熟悉的代码。其中包含委托的定义以及方法调用。

```
// 与之等同的代码
public class ModFilter
{
    private readonly int modulus;

    // 新添加的方法
    private bool WhereClause(int n)
    {
        return ((n % this.modulus) == 0);
    }

    // 原有方法
    private static int SelectClause(int n)
    {
        return (n * n);
    }

    // 原有委托
    private static Func<int, int> SelectDelegate;

    public IEnumerable<int> FindValues(
        IEnumerable<int> sequence)
    {
        if (SelectDelegate == null)
        {
            SelectDelegate = new Func<int, int>(SelectClause);
        }
        return sequence.Where<int>(
            new Func<int, bool>(this.WhereClause)).
            Select<int, int>(SelectClause);
    }
    // 其他方法省略
}
```

这样，若是lambda表达式中的代码访问了对象实例中的成员变量，那么编译器将生成实例方法来表示lambda表达式中的代码。这并没有什么奇特之处——编译器省去了你的一些代码输入工作，不过这也就是其带来的所有好处了，其实现原理依旧是普通的方法调用。

不过若是lambda表达式中访问到了外部方法中的局部变量或是方法参数，那么编译器将帮你完成很多工作。这里就会用到闭包。编译器将生成一个私有的嵌套类型，以便为局部变量实现闭包。局部变量必须传入到实现了lambda表达式主体部分的委托里。此外，所有由该lambda表达式执行的对这些局部变量所作的修改都必须能够在外部访问到。Anders Hejlsberg、Mads Torgersen、Scott Wiltamuth和Peter Golde的*The C# Programming Language, Third Edition*（Microsoft Corporation，2009）一书在7.14.4.1节中介绍了这个行为。当然，代码中内层和外层中共享的可能不止有一个变量。也可能有不止一个的查询表达式。

我们来修改一下该实例方法，让其访问一个局部变量。

```
public class ModFilter
{
    private readonly int modulus;

    public ModFilter(int mod)
    {
        modulus = mod;
    }

    public IEnumerable<int> FindValues(
        IEnumerable<int> sequence)
    {
        int numValues = 0;
        return from n in sequence
               where n % modulus == 0 // 新添加的表达式
               // select子句访问了局部变量
               select n * n / ++numValues;
    }
    // 其他方法省略
}
```

注意，select子句需要访问numValues这个局部变量。编译器为了创建这个闭包，需要使用嵌套类型来实现你所需要的行为。下面就是编译器在这种情况下生成的代码。

```
// 与之等同的代码
public class ModFilter
{
    private sealed class Closure
    {
        public ModFilter outer;
        public int numValues;

        public int SelectClause(int n)
        {
            return ((n * n) / ++this.numValues);
        }
    }

    private readonly int modulus;

    public ModFilter(int mod)
    {
        this.modulus = mod;
    }

    private bool WhereClause(int n)
    {
        return ((n % this.modulus) == 0);
    }

    public IEnumerable<int> FindValues
        (IEnumerable<int> sequence)
    {
        Closure c = new Closure();
        c.outer = this;
        c.numValues = 0;
        return sequence.Where<int>
            (new Func<int, bool>(this.WhereClause))
            .Select<int, int>(
                new Func<int, int>(c.SelectClause));
    }
}
```

在上面这段代码中，编译器专门创建了一个嵌套类，用来容纳所有将在lambda表达式中访问或修改的变量。实际上，这些局部变量将完全被嵌套类的字段所代替。lambda表达式内部的代码以及表达式外部（但仍在当前方法内）

的代码访问的均是同一个字段。lambda表达式中的逻辑也被编译成了内部类中的一个方法。

对于lambda表达式中将要用到的外部方法的参数，编译器也会以对待局部变量的方式实现：编译器将把这些参数复制到表示该闭包的嵌套类中。

回到本条目开始的那个示例中，这时你应该理解了这种看似怪异的行为了。变量index在传入闭包后，但在查询开始执行之前曾被外部代码修改。也就是说，你修改了闭包的内部状态，然后还期待其能够回到从前的状态开始执行，这显然不可能实现。

考虑到延迟执行中的交互以及编译器实现闭包的方式，修改查询与外部代码之间的绑定变量将可能会引发错误的行为。因此，我们应该尽量避免在方法中修改那些将要传入到闭包中，并将在闭包中使用的变量。

条目 34：为匿名类型定义局部函数

很多开发者对匿名类型的认识还不够全面。没错，匿名类型非常易于使用，常用于实现那些短时间存在的、并不在应用程序逻辑中起支配地位的类型。这是因为匿名类型的生命周期无法跨越包含该类型的方法。因此，很多开发人员认为匿名类型并不好用，因为其无法在多个方法之间传递。

这样的说法并不准确。你可以为匿名类型编写泛型方法。不过若是这样做的话，你就不能[①]在泛型方法中处理任何特殊的元素或是编写任何专门的逻辑。

下面就是一个简单的示例，该方法将返回集合中与待查找对象相等的所有元素。

```
static IEnumerable<T> FindValue<T>(IEnumerable<T> enumerable,
    T value)
{
    foreach (T element in enumerable)
```

① 原文为“需要”，而根据作者稍后给出的示例代码，程序确实无法对泛型方法的类型参数作任何形式上的假设。——译者注

```
        {
            if (element.Equals(value))
            {
                yield return element;
            }
        }
    }
```

这个方法即可配合匿名类型使用。

```
static void Main(string[] args)
{
    IDictionary<int, string> numberDescriptionDictionary =
        new Dictionary<int, string>()
    {
        {1,"one"},
        {2, "two"},
        {3, "three"},
        {4, "four"},
        {5, "five"},
        {6, "six"},
        {7, "seven"},
        {8, "eight"},
        {9, "nine"},
        {10, "ten"},
    };
    List<int> numbers = new List<int>()
        { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    var r = from n in numbers
            where n % 2 == 0
            select new { Number = n,
                Description =
                    numberDescriptionDictionary[n] };
    r = from n in FindValue(r, new
        { Number = 2, Description = "two" })
        select n;
}
```

该FindValue()方法并不了解任何该匿名类型的信息，它只是个泛型类型而已。

当然，这类简单的函数也仅能实现这些。若想编写方法来使用匿名类型中的特定属性，那么则需要创建并使用高阶函数（higher-order function）。高阶函

数是指那些接受函数作为参数，或者将函数作为返回值的函数。在操作匿名类型时，高阶函数能够接受函数作为参数的特性非常有用。这样，使用高阶函数和泛型，即可跨越多个方法使用匿名类型。比如下面这个查询。

```
Random randomNumbers = new Random();
var sequence = (from x in Utilities.Generator(100,
                    () => randomNumbers.NextDouble() * 100)
                let y = randomNumbers.NextDouble() * 100
                select new { x, y }).TakeWhile(
                point => point.x < 75);
```

该查询以TakeWhile()方法结束。TakeWhile()方法的签名如下。

```
public static IEnumerable<TSource> TakeWhile<TSource>
    (this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

注意上述TakeWhile的签名，该方法将返回IEnumerable<TSource>并接受一个IEnumerable<TSource>参数。在我们的示例中，TSource是一个表示了一对X和Y的匿名类型。Func<TSource, bool>则表示了一个接受TSource作为参数的方法。

这个技术让你能够创建一系列的、操作匿名类型的类库。这种依赖于泛型方法的查询表达式可以配合匿名类型使用。因为lambda表达式与匿名类型声明在同一作用域内，所以能够了解该匿名类型的所有信息。编译器将创建私有的嵌套类，并将该匿名类型的实例在各个方法中传递。

下面这段代码就创建了一个匿名类型，随后在多个泛型方法中使用了该类型。

```
var sequence = (from x in Funcs.Generator(100,
                    () => randomNumbers.NextDouble() * 100)
                let y = randomNumbers.NextDouble() * 100
                select new { x, y }).TakeWhile(
                point => point.x < 75);

var scaled = from p in sequence
             select new {x = p.x * 5, y = p.y * 5};
var translated = from p in scaled
                 select new { x = p.x - 20, y = p.y - 20};
```

```
var distances = from p in translated
                let distance = Math.Sqrt(
                    p.x * p.x + p.y * p.y)
                where distance < 500.0
                select new { p.x, p.y, distance };
```

这其中并没有什么值得惊讶的地方。不过是编译器生成了委托，然后调用了这些委托而已。编译器均会为这些查询方法生成专门的方法，然后将匿名类型作为参数传入。编译器将分别为每个方法创建与之绑定的委托，随后使用委托作为查询方法的参数。

随着程序功能的不断发展，这些代码难免会超过你的控制，算法可能被不停复制，最终导致出现大量的重复代码。因此，接下来我们对上述代码进行修改，以便在需要更多功能时，仍旧保持代码简单、模块化且易于扩展。

一种做法是将某些代码抽取出来，封装成简单的方法，同时也作为可重用的块。我们可以对现有算法进行重构，将其改写成泛型方法，接受lambda表达式来执行算法所需要的特定操作。

几乎下面所有的方法都是用来实现将一种类型映射成另一种类型。有一些甚至更加简单，只是将原有对象映射成同类型的另一个对象而已。

```
// 在另一个类中定义Map方法
public static IEnumerable<TResult> Map<TSource, TResult>(this
IEnumerable<TSource> source,
    Func<TSource, TResult> mapFunc)
{
    foreach (TSource s in source)
        yield return mapFunc(s);
}

// 用法
var sequence = (from x in Funcs.Generator(100,
                    () => randomNumbers.NextDouble() * 100)
                let y = randomNumbers.NextDouble() * 100
                select new { x, y }).TakeWhile(
                point => point.x < 75);
```

```
var scaled = sequence.Map(p => new {x = p.x * 5,
    y = p.y * 5});
var translated = scaled.Map(p => new { x = p.x - 20,
    y = p.y - 20});
var distances = translated.Map(p => new { p.x, p.y,
    distance = Math.Sqrt(p.x * p.x + p.y * p.y) });
var filtered = from location in distances
               where location.distance < 500.0
               select location;
```

这种技术可以让算法在尽可能避免了解匿名类型具体细节的情况下仍旧被提取出来。所有的匿名类型都只支持IEquatable<T>。因此你只能使用System.Object的公共成员以及IEquatable<T>的成员。这样做并没有改变现有情况，不过你应该意识到若想在方法之间传递匿名类型，那么必须也要同时传递操作该类型的方法。

按照同样的思路，你会发现原方法的某一部分也可能会在其他地方重复使用。这时，同样应该将可重用的代码片断提取出来，封装成另一个泛型方法，以便在不同位置调用。

不过，我们应该在合理范围内小心地使用这种技术。对于那些将在很多算法中起到核心作用的类型，不应该用匿名类型实现。当你越来越多地发现使用了同样的匿名类型，且对该类型进行了越来越多的操作时，就该考虑将其转换为某种具体类型来实现了。最终的结论似乎有些武断，不过若是在超过三个主要算法中使用到了同一个匿名类型，那么则应该将其改写成具体类型。同理，若你发现需要创建过于复杂的lambda表达式来使用匿名类型，那么这也是一个你开始创建具体类型的信号。

匿名类型是一种轻量级的类型，仅包含可被读写的属性，用来存放简单的数值。你可以基于这些简单类型创建算法，也可以使用lambda表达式和泛型方法类维护匿名类型。与通过创建私有嵌套类来限制类型的作用域类似，匿名类型也可以将类型的作用范围限制在指定的方法中。通过使用泛型和高阶函数，我们也可以在使用匿名类型的同时仍旧保证代码的模块化。

条目35：不要在不同命名空间中声明同名的扩展方法

在本章的前面部分（条目28和条目29），我介绍了为接口或类型创建扩展方法的三个理由：为接口添加默认的实现、为封闭的泛型类型创建行为以及创建可组合的接口。不过，扩展方法并不总是设计上的首选。在上述这些情况下，我们对现有类型的定义进行了增强，不过此类增强并没有从根本上改变类型的行为。

条目22介绍了通过扩展方法来为最小化设计的接口提供常见操作的默认实现。或许你也会考虑使用同样的方法来增强你的类型，甚至准备创建多个版本的扩展，随后通过修改引入的命名空间在各个版本之间进行切换。不要这样做。扩展方法能够为实现了某个接口的类型提供默认的实现。不过，对类类型进行扩展还有一些更好的方法。滥用扩展方法将很快导致出现一系列混乱冲突的方法，极大地增加了维护成本。

我们先从一个演示误用扩展方法的例子开始。例如一个其他类库中的Person类型。

```
public sealed class Person
{
    public string FirstName
    {
        get;
        set;
    }
    public string LastName
    {
        get;
        set;
    }
}
```

你可能会需要通过扩展方法来让该类型支持将其表示的人名输出到控制台。

```
// 不好的开始
// 使用扩展方法扩展类型
```

```
namespace ConsoleExtensions
{
    public static class ConsoleReport
    {
        public static string Format(this Person target)
        {
            return string.Format("{0,20}, {1,15}",
                target.LastName, target.FirstName);
        }
    }
}
```

上述扩展方法很容易如下实现：

```
static void Main(string[] args)
{
    List<Person> somePresidents =
        new List<Person>{
            new Person{
                FirstName = "George",
                LastName = "Washington" },
            new Person{
                FirstName = "Thomas",
                LastName = "Jefferson" },
            new Person{
                FirstName = "Abe",
                LastName = "Lincoln" }
        };

    foreach (Person p in somePresidents)
        Console.WriteLine(p.Format());
}
```

这看上去没有什么问题，不过需求总是在改变。某天之后程序可能会需要以XML的格式输出，那么某人或许想这样编写代码。

```
// 更差的作法
// 带有二义性的扩展方法，定义于不同的命名空间中
namespace XmlExtensions
{
    public static class XmlReport
    {
```

```
        public static string Format(this Person target)
        {
            return new XElement("Person",
                new XElement("LastName", target.LastName),
                new XElement("FirstName", target.FirstName)
                ).ToString();
        }
    }
}
```

在源文件中切换using语句即可切换输出的格式，但这是一种对扩展方法的误用，让类型的扩展变得难以维护。若开发者使用了错误的命名空间，那么程序的行为也将发生变化。若是开发人员没有引入任何一个扩展所在的命名空间，那么程序甚至不能通过编译。若是在不同的方法中需要使用两个命名空间中的扩展，那么则需要将类定义拆分到不同的文件中，以便符合要求。同时引入两个命名空间将会因为引用的二义性导致编译错误。

考虑到这些，我们显然要通过另一种方法来实现该功能。扩展方法强迫你只能根据编译期类型来完成方法分发。通过命名空间来切换将要使用的方法也让该策略更加难以维护。

这种功能并不基于你要扩展的类型：将Person对象进行输出至控制台或XML并不是Person类型的职责，而更可以看作是使用Person的外部环境的责任。

扩展方法应该用于增强一个现有类型的自然功能。你应该仅仅使用扩展方法来为类型添加逻辑上本应该属于该类型的功能。条目28和条目29介绍了两种增强接口和封闭类型的技术。查看这两个条目中的示例，你会发现从类型使用者的角度来看，这些扩展的功能均属于类型的一部分。

而与这里的示例相比较，这些扩展并不像是Person类型的一部分。Format方法只是操作了Person类型，而在使用Person类型的开发者眼中，该方法并不属于Person类型。

这些方法就其本身来讲是合法的，不过却应该以普通静态方法的形式实现，放在专门的辅助类中。实际上，这些方法应该放在同一个类中，并有不同的方法名。

```
public static class PersonReports
{
    public static string FormatAsText(Person target)
    {
        return string.Format("{0,20}, {1,15}",
            target.LastName, target.FirstName);
    }
    public static string FormatAsXML(Person target)
    {
        return new XElement("Person",
            new XElement("LastName", target.LastName),
            new XElement("FirstName", target.FirstName)
            ).ToString();
    }
}
```

这个类同时包含了两个静态方法，其不同的名字很清楚地反映了各个方法的用处。这样，既可以为使用者提供必要的功能，也不会给公共接口带来二义性。若是需要的话，每个开发者都可以使用到任意一个方法。这并不需要在不同的命名空间中提供同样的方法签名，进而造成二义性。牢记这一点非常重要，因为很少有开发者会意识到修改using语句将可能会影响到程序的行为。他们也许会想到这可能导致编译期错误，不过不会想到会导致运行时错误。

当然，若是修改这些方法的名称，那么又可以将其作为扩展方法了。但这样做其实也不会带来什么好处，因为这些方法不是对类型本身功能的扩展，而仅仅是使用这些类型而已。但不管怎样，因为名称不会冲突，所以你可以将两个方法放在同一个命名空间的同一个类中，从而避免了前面例子中的陷阱。

对于某种特定类型的所有扩展方法，我们应该将其作为一个完整的集合考

虑。扩展方法不应该在命名空间上进行重载。无论何时，当你发现需要创建多个同名的扩展方法时，要立即停下来。要么修改方法的签名，要么考虑创建传统的静态方法，以便避免发生由编译器根据using语句选择不同重载所导致的二义性。

第5章

使 用 LINQ

5

C#3.0之所以添加了很多语言上的新特性，其很主要的一个原因是为了支持LINQ。这些新语言特性是为了支持延迟查询，将查询转换成LINQ to SQL需要的SQL语句以及对不同数据源进行操作的统一语法。第4章介绍了如何在非数据查询中使用这些新的语言特性。而本章将专注于将这些功能用于查询数据之上，无论其底层的数据源是什么。

LINQ的核心目标之一是，无论何种数据源均提供统一的查询语法。不过，虽然其查询语法能够应用于各种数据源之上，但你仍需要用各种方法来实现将查询与实际数据源连接起来的查询提供器（query provider）。只有理解了这些不同的行为，才能更加透明地操作多种数据源。如果需要的话，甚至可以自行创建数据提供器。

条目 36：理解查询表达式与方法调用之间的映射

LINQ构建于两个概念之上：一种查询语言和一系列将查询语言转换成方法调用的实现。C#编译器将把用查询语言编写的查询表达式转换成方法调用。

每个查询表达式都可以映射成一个或多个方法调用。你应该从两个方面理解这种映射。从类的使用者角度看，应该理解的是查询表达式仅仅是方法调用而已。例如，where子句将被转换成名为Where()的方法，并传入合适的参数。

而从类的设计者角度看，你应该了解由基本框架所提供的这些方法的默认实现，并判断是否可以为你的类型提供更好的专门实现。如果不能的话，那么就直接使用类库提供的实现。不过，若你打算创建一种更好的实现，则必须完全理解了从查询表达式到方法调用之间的转换原理，从而确保你的自定义方法能够处理所有的转换过程。对于某些查询表达式来说，其转换的方式很明显。而对于某些较为复杂的表达式，其转换过程则不是那么直接。

完整的查询表达式模式（query expression pattern）包含了11个方法。下面就是Anders Hejlsberg、Mads Torgersen、Scott Wiltamuth和Peter Golde在The *C# Programming Language, Third Edition*（Microsoft Corporation，2009）中7.15.3节给出的这些方法的定义（已由微软公司授权转载）。

```
delegate R Func<T1,R>(T1 arg1);
delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);
class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);
    public C<U> Select<U>(Func<T,U> selector);
    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);
    public C<V> Join<U,K,V>(C<U> inner,
        Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector,
        Func<T,U,V> resultSelector);
    public C<V> GroupJoin<U,K,V>(C<U> inner,
        Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector,
        Func<T,C<U>,V> resultSelector);
    public O<T> OrderBy<K>(Func<T,K> keySelector);
    public O<T> OrderByDescending<K>(Func<T,K> keySelector);
    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
```

```
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);
    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

.NET基础类库为该模式提供了两种通用的参考实现。System.Linq.Enumerable使用了IEnumerable<T>上的扩展方法来实现查询表达式模式，而System.Linq.Queryable则提供了类似的一系列IQueryable<T>上的扩展方法，让查询提供器能够将查询转换成另一种格式的语法。（例如，LINQ to SQL就能将查询表达式转换成SQL查询，并由SQL数据库引擎执行。）作为类的使用者，大多数的查询都会使用这两种查询中的一种来实现。

其次，作为类的编写者，你可以创建一个实现了IEnumerable<T>或IQueryable<T>（或继承自IEnumerable<T>或IQueryable<T>的封闭泛型类型）的数据源，这样你的类型就自然实现了查询表达式模式。这是因为你的类型可以直接使用定义在基本类库中的扩展方法。

在继续深入之前，你必须理解C#语言并没有强迫限制查询表达式模式上的语义。我们完全可以创建一个符合某个查询方法签名的方法，然后在其中定义任意的操作。编译器无法验证你的Where方法是否符合查询表达式模式的语义，唯一能够保证的就是语法上的正确性。这个行为类似于对接口的实现。例如，你可以在接口的实现方法中做任何事情，无论其是否符合用户的期待。

不过，这并不代表你可以随心所欲地给出实现。若你准备实现任何一个查询表达式模式方法，那么应该保证其行为在语法和语义两方面都与参考实现一致。除了性能上的差别之外，调用者不应该察觉出调用你的实现和参考实现在行为上有什么不同。

从查询表达式到方法调用的转换是一个复杂的迭代过程。编译器将重复地将表达式转换成方法调用，直到成功转换了所有的表达式为止。此外，编译器在转换时也有一个特定顺序，不过接下来不会按照这个顺序来介绍。编译器采用这种顺序有助于简化其实现，该顺序可以在C#规范中找到。而本条目的介绍顺序则更易于读者理解。接下来，我将使用一些小巧简单的示例来解释一些转换。

在下面的这个查询中，我会介绍where、select以及范围变量。

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var smallNumbers = from n in numbers
                   where n < 5
                   select n;
```

表达式from n in numbers将把范围变量n绑定到numbers中的每个元素上。where子句中定义的过滤条件将被转换成Where方法的调用。where n < 5表达式将被翻译成如下语句。

```
numbers.Where((n) => n < 5);
```

Where只不过是个过滤器。Where的输出是输入序列的一个满足指定断言的元素的子集。输入和输出序列中包含的元素类型必须相同，且Where方法内部不应该修改输入序列中的元素（由用户给出的谓词可能会修改元素，不过查询表达式模式并不应该为此负责）。

Where方法既可以用可被numbers访问到的实例方法实现，也可以使用符合numbers类型的扩展方法提供。在这个示例中，numbers是一个包含了int元素的数组。因此，方法调用中的n即为一个整数。

Where是从查询表达式转换成方法调用时最简单的一个例子。在继续下去之前，我们来深入了解一下其工作原理，以及它对于转换来说意味着什么。编译器将在任何重载解析以及类型绑定操作之前完成从查询表达式到方法调用的转换过程。编译器在将某个查询表达式转换成方法调用时，还不知道是否真的有合适的对应方法供下一步选择。因此，编译器此时并不检查类型，也不会

搜索待选的扩展方法，只是简单地将查询表达式转换成方法的调用。在所有的查询都被转换成方法调用语法之后，编译器才开始进行查找待选方法以及选择最佳匹配方法的工作。

接下来，我们可以在查询中添加一个select表达式。select子句将被转换成Select方法。但在某些特殊情况下，Select方法也可以被优化去掉。前面的这个查询就是一个退化选择（degenerate select），仅用来查询出范围变量。退化选择查询可以被优化去掉，因为输出序列和输入序列并不相同。该示例查询包含一个where子句，破坏了输入序列和输出序列之间的一对一关系。因此，该查询转换后的最终方法调用将类似如下所示。

```
var smallNumbers = numbers.Where(n => n < 5);
```

select子句是冗余的，因此就被优化去掉了。这样做不会出现什么问题，因为select操作在另一个表达式（此处即为where）的直接结果上。

若是select没有操作与另一个表达式的直接结果上，那么则不能被优化去掉。例如如下查询：

```
var allNumbers = from n in numbers select n;
```

该查询将被转换成如下的方法调用：

```
var allNumbers = numbers.Select(n => n);
```

在LINQ中，select经常用来将一种元素转换或投影成另一种不同的元素或类型。下面这个查询就修改了返回值。

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var smallNumbers = from n in numbers
                   where n < 5
                   select n * n;
```

你也可以将输入序列转换成不同的类型。

```
int [] numbers = {0,1,2,3,4,5,6,7,8,9};
var squares = from n in numbers
              select new { Number = n, Square = n * n};
```

select子句将映射到一个满足查询表达式模式签名的Select方法。

```
var squares = numbers.Select(n =>
    new { Number = n, Square = n * n});
```

Select能够将输入类型转换成输出类型。正确实现的Select方法必须为每个输入元素生成一个且仅为一个的输出元素。此外，正确实现的Select方法也不能修改输入序列中的元素。

结束了这个简单的查询表达式之后，我们来看看一些较为复杂的转换。

排序关系将映射到OrderBy或ThenBy方法上，或是OrderByDescending或ThenByDescending方法上。比如如下的查询：

```
var people = from e in employees
             where e.Age > 30
             orderby e.LastName, e.FirstName, e.Age
             select e;
```

将被转换成：

```
var people = employees.Where(e => e.Age > 30).
    OrderBy(e => e.LastName).
    ThenBy(e => e.FirstName).
    ThenBy(e => e.Age);
```

注意，在查询表达式模式的定义中，ThenBy将操作于一个OrderBy和ThenBy的返回序列之上。这些序列中包含了特殊的标记，以便ThenBy可以在前一个排序字段相同的前提下继续操作根据当前排序字段排序一段子元素。

若是将orderby子句作为一个独立的子句给出，那么转换的过程将完全不同。下面的这个查询将会把整个序列按照LastName排序，然后再把整个序列按照FirstName排序，最后仍然是把整个序列按照Age排序。

```
// 非正常做法。将整个序列排序了3次
var people = from e in employees
             where e.Age > 30
             orderby e.LastName
             orderby e.FirstName
             orderby e.Age
             select e;
```

若是以分开的形式编写，那么即可指定某个orderby子句使用降序排列。

```
var people = from e in employees
             where e.Age > 30
             orderby e.LastName descending
             thenby e.FirstName
             thenby e.Age
             select e;
```

OrderBy方法将创建一个不同类型的序列作为输出，因此thenby子句即可更加高效地执行，并保证整个查询的正确性。ThenBy无法操作于未经排序的序列上，而只能操作在已被排序的序列上（示例中为O<T>）。序列中的子区间已经排序过，且被标记出来。若你需要实现自己的OrderBy和ThenBy方法，那么必须遵守同样的规则。你需要标记出每个已经排序过的子区间，随后thenby子句才能正常工作。ThenBy方法需要接受OrderBy或ThenBy方法的输出，随后对每个子区间进行排序。

上面所有有关OrderBy和ThenBy的内容均适用于OrderByDescending和ThenByDescending。实际上，若你的类型需要上述这些方法中的某一种，那么一般也要同时实现所有的4种。

剩下的表达式转换过程均要多个步骤才能完成。这些查询要么涉及分组，要么带有多个from子句，因此需要延续的处理过程。包含延续处理过程的查询表达式将首先被转换成嵌套查询，随后再将嵌套查询转换为方法调用。下面就是一个需要延续处理的查询。

```
var results = from e in employees
              group e by e.Department into d
              select new { Department = d.Key,
              Size = d.Count() };
```

在开始任何其他转换之前，延续处理将被转换成嵌套查询。

```
var results = from d in
    from e in employees group e by e.Department
    select new { Department = d.Key, Size = d.Count()};
```

创建好嵌套查询之后，即可继续转换成如下的方法调用。

```
var results = employees.GroupBy(e => e.Department).
    Select(d => new { Department = d.Key, Size = d.Count()});
```

上述查询中的GroupBy将返回一个序列。而查询表达式语法中的另一种GroupBy方法将返回一个序列的组，其中每个组都包含了一个键以及该键对应的元素列表。

```
var results = from e in employees
              group e by e.Department into d
              select new { Department = d.Key,
              Employees = d.AsEnumerable()};
```

该查询将被转换成如下的方法调用。

```
var results2 = employees.GroupBy(e => e.Department).
    Select(d => new { Department = d.Key,
        Employees = d.AsEnumerable()});
```

GroupBy方法将生成一个包含了键值对的序列。键为组的选择器，值为当前组中的各个元素。不管怎样，输出的每个键值对中的值一定会包含一系列由输入元素创建的、属于该组的元素。

最后需要了解的方法是SelectMany、Join和GroupJoin。这三个方法比较复杂，因为它们将操作于多个输入序列之上。这些方法将在多个序列上进行遍历，然后将这些序列按照某种条件展平，变成一个输出序列。其中，SelectMany将在两个源序列上执行一个交叉连接，例如如下查询：

```
int[] odds = {1,3,5,7};
int[] evens = {2,4,6,8};
var pairs = from oddNumber in odds
            from evenNumber  in evens
            select new {oddNumber, evenNumber,
            Sum=oddNumber+evenNumber};
```

其生成的序列包含了16个元素：

```
1,2, 3
1,4, 5
```

```
1,6, 7
1,8, 9
3,2, 5
3,4, 7
3,6, 9
3,8, 11
5,2, 7
5,4, 9
5,6, 11
5,8, 13
7,2, 9
7,4, 11
7,6, 13
7,8, 15
```

包含了多个from子句的查询表达式将被转换成对SelectMany方法的调用。前面的这个查询就将转换成如下的SelectMany方法。

```
int[] odds = { 1, 3, 5, 7 };
int[] evens = { 2, 4, 6, 8 };
var values = odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new { oddNumber, evenNumber,
    Sum = oddNumber + evenNumber });
```

SelectMany的第一个参数是一个函数，用来将第一个源序列中的每个元素映射到第二个源序列中的元素之上。第二个参数（输出选择器）将根据两个序列中的每一对映射来投影出结果元素。

SelectMany()将迭代第一个序列。对于第一个序列中的每个元素，该方法都会迭代第二个序列，随后根据这一对映射来生成结果值。随后将在那个展开了的、包含了来自两个序列中的每一对映射过的数据的序列上调用输出选择器。SelectMany的一个可行的实现如下所示。

```
static IEnumerable<TOutput> SelectMany<T1, T2, TOutput>(
    this IEnumerable<T1> src,
    Func<T1, IEnumerable<T2>> inputSelector,
    Func<T1, T2, TOutput> resultSelector)
{
    foreach (T1 first in src)
```

```
        {
            foreach (T2 second in inputSelector(first))
                yield return resultSelector(first, second);
        }
    }
```

方法中首先开始迭代第一个输入序列。随后对于第一个序列中的每个元素，均会遍历第二个输入序列。这很重要，因为第二个序列上的输入选择器可能会要依赖于第一个序列中的当前元素。随后，在生成了一对元素之后，即可在其上应用输出选择器。

若是查询还包含了更多的表达式，SelectMany无法一次创建出最终结果，那么SelectMany将会先创建出包含了来自每个序列中的单个元素的元组(tuple)。这个包含了元组的序列将用于稍后处理。例如，若对上述查询进行修改：

```
int[] odds = { 1, 3, 5, 7 };
int[] evens = { 2, 4, 6, 8 };
var values = from oddNumber in odds
             from evenNumber in evens
             where oddNumber > evenNumber
             select new { oddNumber, evenNumber,
             Sum = oddNumber + evenNumber };
```

那么该查询将生成如下的SelectMany调用。

```
odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new {oddNumber, evenNumber});
```

随后，完整的查询将被转换成如下语句。

```
var values = odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new { oddNumber, evenNumber }).
    Where(pair => pair.oddNumber > pair.evenNumber).
    Select(pair => new {
        pair.oddNumber,
        pair.evenNumber,
        Sum = pair.oddNumber + pair.evenNumber });
```

在编译器将多个from子句转换成SelectMany调用时，可以看到SelectMany的另一个很有意思的特性，即该方法可以很好地与自身组合使用。多于两个的from子句将生成不止一个的SelectMany()方法调用。第一个SelectMany()的结果能够作为第二个SelectMany()的输入，这样即可生成一个三元组。该三元组包含了三个序列中元素的各种组合。比如如下这个查询：

```
var triples = from n in new int[] { 1, 2, 3 }
              from s in new string[] { "one", "two",
                  "three" }
              from r in new string[] { "I", "II", "III" }
              select new { Arabic = n, Word = s, Roman = r };
```

将被转换成如下的方法调用：

```
var numbers = new int[] {1,2,3};
var words = new string[] {"one", "two", "three"};
var romanNumerals = new string[] { "I", "II", "III" };
var triples = numbers.SelectMany(n => words,
    (n, s) => new { n, s}).
    SelectMany(pair => romanNumerals,
    (pair,n) =>
        new { Arabic = pair.n, Word = pair.s, Roman = n });
```

可以看到，只要使用更多的SelectMany()，即可将三个序列扩展到任意个序列。这些示例也说明了SelectMany将会在查询中使用匿名类型。SelectMany()返回的结果就是一个包含了匿名类型的序列。

接下来将介绍剩下的两个需要理解的转换：Join和GroupJoin。这两个方法都应用于连接表达式中。若是join表达式包含了into子句，那么将会使用GroupJoin。若是不包含into子句，那么将使用Join。

例如下面这个没有into子句的连接：

```
var numbers = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var labels = new string[] { "0", "1", "2", "3", "4", "5" };
var query = from num in numbers
            join label in labels on num.ToString() equals
                label
            select new { num, label };
```

将转换成如下代码：

```
var query = numbers.Join(labels, num => num.ToString(),
    label => label, (num, label) => new { num, label });
```

而into子句将创建出一个包含了细分后结果的列表：

```
var groups = from p in projects
             join t in tasks on p equals t.Parent
                 into projTasks
             select new { Project = p, projTasks };
```

该查询将转换成GroupJoin调用：

```
var groups = projects.GroupJoin(tasks,
    p => p, t => t.Parent, (p, projTasks) =>
        new { Project = p, TaskList = projTasks });
```

将所有表达式转换成方法调用的完整过程非常复杂，且通常要多个步骤才能完成。

不过好消息是，在大多数情况下，你可以相信编译器能够正确地完成转换的工作。且由于你的类型实现了IEnumerable<T>，那么类型的使用者均能得到期待中的正确行为。

但在有些时候，你会不得不创建一些自定义的实现了查询表达式模式的方法。或许因为你的集合总会按照某个字段保持有序状态，这时即可将OrderBy方法短路掉。或许你的类型能够公开出包含列表的列表，这时GroupBy和GroupJoin可能会有更高效的实现。

还有可能出现的是，你想要创建自己的LINQ提供器，因此需要实现整个模式。若是这样的话，那么必须首先理解每个查询方法的行为，并知道实现的具体细节。因此在开始创建你自己的实现之前，请查看上述示例，确保你完全理解了每个查询方法的行为。

我们编写的很多自定义类型都是用来描述某类集合类型数据的。因此，使用该类型的开发者往往会期待能够像操作其他集合类型一样，使用内建的查询语法来操作你的类型。只要你的集合实现了IEnumerable<T>接口，那么也就自然满足了这个需求。不过根据实际情况的不同，你的类型或许能够改进默认

的实现。在你着手改进之前，首先要保证你的类型在各方面都符合了查询表达式模式的要求。

条目 37：推荐使用延迟求值查询

在定义查询时，我们不会立即得到查询返回的数据并生成序列。定义的仅是为了得到查询结果所要执行的步骤，在开始迭代查询结果时，查询才会开始真正执行。这也就意味着，每次开始执行查询时，都会从头开始执行查询的每个步骤。这通常是正确的行为，即每次遍历都会得到全新的结果，也叫做延迟求值（lazy evaluation）。不过有时候这并不是你想要的做法。你会希望一次性地将集合生成，然后可以任意读取，这叫做主动求值（eager evaluation）。

当你在编写需要多次遍历结果的查询时，就应该考虑选择何种行为。是只需要数据的一个快照，还是先给出得到结果所必需的操作步骤，然后在需要时再获取当前数据呢？

对于你所熟悉的操作方式来说，这个概念将带来很大的改变。通常来讲，你会认为代码将立即执行。不过对于LINQ查询而言，我们只不过是将查询代码插入到了某个方法中而已，这些查询代码将会在稍后运行。不仅如此，若该LINQ提供器使用的表达式树而非委托，那么该表达式树还能在稍后添加新的表达式，从而构造出更加复杂的查询。

我们来通过一个示例来解释延迟求值和主动求值之间的不同。下面的这段代码将生成一个序列，随后迭代该序列三次。在每次遍历之间都会有个暂停。

```
private static IEnumerable<TResult>
    Generate<TResult>(int number, Func<TResult> generator)
{
    for (int i = 0; i < number; i++)
        yield return generator();
}

private static void LazyEvaluation()
{
    Console.WriteLine("Start time for Test One: {0}",
```

```
            DateTime.Now);
        var sequence = Generate(10, () => DateTime.Now);
        Console.WriteLine("Waiting....\tPress Return");
        Console.ReadLine();

        Console.WriteLine("Iterating...");
        foreach (var value in sequence)
            Console.WriteLine(value);

        Console.WriteLine("Waiting....\tPress Return");
        Console.ReadLine();
        Console.WriteLine("Iterating...");
        foreach (var value in sequence)
            Console.WriteLine(value);
    }
```

下面是该程序的一段示例输出。

```
Start time for Test One: 11/18/2007 6:43:23 PM
Waiting....    Press Return

Iterating...
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
11/18/2007 6:43:31 PM
Waiting....    Press Return

Iterating...
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
11/18/2007 6:43:42 PM
```

在延迟求值的示例中，注意到每次迭代时序列都会重新生成（注意输出中的时间戳）。序列变量并没有暂存住创建的元素，其保存的是创建该序列的表达式树。你可以尝试执行一下该代码，调试到每个查询中并查看表达式是何时开始求值的。这也是了解LINQ查询执行时机的最直观的方法。

使用这个特点，我们可以基于现有的查询编写新的查询。程序没有必要先获取第一个查询的结果，然后再接下来对其进一步处理。而是可以在不同的步骤中共同构造一个查询，然后仅执行最终查询一次。例如，若希望修改查询，让其以国际标准时间的形式输出。

```
var sequence1 = Generate(10, () => DateTime.Now);
var sequence2 = from value in sequence1
                select value.ToUniversalTime();
```

序列1和序列2共享的是其构造的方式，而不是数据。序列2并不需要遍历序列1的结果并对其修改，而是直接执行用来生成序列1的代码，随后再执行用来生成序列2的代码。若你在不同的时间迭代两个序列，那么将会看到二者的结果完全不同。序列2包含的并不是经过转换后的序列1中的元素，其包含的是全新的值。序列2并没有先生成一个序列的时间，然后再将整个序列转换成国际标准时间，而是直接生成了所需要的国际标准时间。

由于其延迟求值的本质，查询表达式可以操作在无限长度的序列之上。在这种情况下，我们可以让查询从序列中的第一个元素开始，直到找到合适的元素后才停止。而在另一些情况中，有一些查询表达式必须先得到完整的序列，然后才能计算出其输出。理解这些瓶颈所在的位置将有助于编写更加高效的查询，不至于影响程序的性能。此外，了解这些也会让你尽可能地避免对整个序列进行遍历，避免在程序中创建瓶颈。

考虑下面这个小程序。

```
static void Main(string[] args)
{
    var answers = from number in AllNumbers()
                  select number;
```

```
        var smallNumbers = answers.Take(10);
        foreach (var num in smallNumbers)
            Console.WriteLine(num);
    }

    static IEnumerable<int> AllNumbers()
    {
        int number = 0;
        while (number < int.MaxValue)
        {
            yield return number++;
        }
    }
```

这个程序就演示了前面提到过的那种情况，即某个程序可能并不需要完整的序列。该方法的输出只是0~9这10个数字，虽然AllNumbers()将会生成一个无限的序列。（当然，这个方法最终将会溢出，不过需要等待很长时间。）

之所以这段程序执行很快，是因为它并不需要整个序列。Take()方法将返回序列中的前*N*个元素，因此不会有什么不妥。

不过若是按照如下方式编写查询，那么程序将永远运行下去[①]。

```
class Program
{
    static void Main(string[] args)
    {
        var answers = from number in AllNumbers()
                      where number < 10
                      select number;

        foreach(var num in answers)
            Console.WriteLine(num);
    }
}
```

之所以会永远运行下去，是因为该查询需要检查序列中的每个元素，并判断其是否符合要求。实现同样逻辑的这个版本的程序会需要整个序列。

① 直到发生溢出为止。——译者注

有几个查询操作符需要整个序列才能完成所需的操作。Where将使用整个序列。Orderby需要整个序列都可用。Max和Min也需要整个序列。若是不逐一检查序列中的每个元素，那么这些操作便无法完成。因此当你需要此类功能时，也必须使用这些方法。

你应该预先考虑到使用那些需要整个序列的方法所带来的后果。可以看到，若序列长度可能为无限的话，那么则必须避免使用需要整个序列的方法。第二，即使序列并不是无限的，也要将过滤序列的查询方法放在查询的最早执行。若查询的头几步就能从集合中去掉一些元素，那么这也会提高接下来的查询执行的性能。

例如，下面的两个查询将生成同样的结果，不过第二个查询可能会执行得更快一些。一些良好实现的提供器会自动对上述查询进行优化，让两个查询都能高效完成。不过在LINQ to Objects的实现（由System.Linq.Enumerable提供）中，所有的产品都会被读取出来并排序，然后再对产品序列进行过滤。

```
// 在过滤之前排序
var sortedProductsSlow =
    from p in products
    orderby p.UnitsInStock descending
    where p.UnitsInStock > 100
    select p;

// 在排序之前过滤
var sortedProductsFast =
    from p in products
    where p.UnitsInStock > 100
    orderby p.UnitsInStock descending
    select p;
```

注意，第一个查询先对整个序列进行了排序，然后过滤掉了那些存货不足100件的产品。而第二个查询则先对序列进行了过滤，然后再对过滤的结果（可能比原始序列小很多）进行排序。很多时候，了解某个方法是否需要整个序列将导致程序性能方面的巨大差异。因此我们需要熟知哪些方法需要整个序列，并尽量在查询表达式的最后部分再执行此类操作。

至此，我已经给出了很多使用延迟求值的理由。在大多数情况下，延迟求值是最好的方法。不过在有些情况下，我们确实需要数据在某个时间点上的快照。你可以使用两个方法来立即生成序列，并将其存放在容器中：ToList()和ToArray()。这两个方法都将执行查询，并将结果分别存放在List<T>或数组中。

这些方法有两个重要的用途。首先，通过强制查询立即执行，这些方法能够得到数据当前的快照。因为这样的查询将立即执行，而不是等到你开始对结果序列开始遍历时。此外，你也可以用ToList()或ToArray()加载那些需要多次访问，但不太可能会发生变化的数据。随后即可将其缓存起来，已备稍后再次使用。

在大多数情况下，延迟求值都要比主动求值更加灵活且易于使用。在那些确实需要主动求值的情况下，你可以使用ToList()或ToArray()方法来强制执行查询并保存其执行结果。不过除非特别有必要使用主动求值，否则最好使用延迟求值。

条目 38：推荐使用 lambda 表达式而不是方法

这条建议看似有些违反常识，因为使用lambda表达式将会造成lambda表达式主体部分的代码重复。你会经常发现在重复输入一些小块的逻辑。例如下面的这段代码就将同样的逻辑重复了几次。

```
var allEmployees = FindAllEmployees();

// 找到第一个员工
var earlyFolks = from e in allEmployees
                 where e.Classification ==
                     EmployeeType.Salary
                 where e.YearsOfService > 20
                 where e.MonthlySalary < 4000
                 select e;

// 找到最近来的人
var newest = from e in allEmployees
             where e.Classification == EmployeeType.Salary
```

```
                where e.YearsOfService < 2
                where e.MonthlySalary < 4000
                select e;
```

当然，你可以将多个Where的调用改写成仅调用Where一次，然后把所有的条件写在一起。不过两种表示方法之间并没有多少差别。考虑到查询组合（参见第3章条目17），以及简单的where谓词将被内联，二者的性能不会有什么差别。

或许你会考虑将重复出现的lambda表达式重构到专门的方法中，以便重用。最后的代码可能会如下所示。

```
// 提取成方法
private static bool LowPaidSalaried(Employee e)
{
    return e.MonthlySalary < 4000 &&
        e.Classification == EmployeeType.Salary;
}

// 在其他地方
var allEmployees = FindAllEmployees();
var earlyFolks = from e in allEmployees
                 where LowPaidSalaried(e) &&
                 e.YearsOfService > 20
                 select e;

// 找到最近来的人
var newest = from e in allEmployees
             where LowPaidSalaried(e) && e.YearsOfService < 2
             select e;
```

这是个很简单的示例，因此看不出有什么大的变化。不过感觉似乎已经好些了。现在，若是员工的分级系统或是低工资标准需要改变的话，只需要在一处修改即可。

很不幸的是，这种代码的重构方式反倒降低了其可重用性。实际上，第一种方法的可重用性要比第二种方法高一些。这是由lambda表达式的求值、解析以及最终执行的方式所决定的。若是你和大多数开发者一样，那么定会对那些复制的代码深恶痛绝，并会竭力除去这些重复。这样看来，第二个抽取出方法的版本更加简单。它仅包含了逻辑的一个副本，且在需要时也便于修改。看上去似乎是个很不错的软件工程实践。

不过这种想法也是错误的。编译器会将查询表达式中的lambda表达式转换成委托来执行。其他类型也可以根据lambda表达式来构造表达式树、解析该表达式并在其他环境中执行该表达式。LINQ to Objects使用委托来执行，而LINQ to SQL则使用表达式树来实现。

LINQ to Objects将在本地数据源上执行查询，这类本地数据源通常是个泛型集合。其实现将会构造一个包含了lambda表达式中逻辑的匿名委托，然后执行这段代码。LINQ to Objects扩展方法使用IEnumerable<T>作为输入序列。

而LINQ to SQL则使用表达式树来容纳查询语句。表达式树能够在逻辑上表示一个查询。LINQ to SQL将解析该表达式树，并根据其结构创建合适的T-SQL语句，随后将直接发送给数据库处理。

这个处理的过程需要LINQ to SQL引擎解析整个表达式树，并将每个逻辑操作替换成与之等同的SQL语句。所有的方法都将被替换成一个Expression.MethodCall节点。而LINQ to SQL引擎无法将方法调用转换成SQL语句，因此会直接抛出异常。LINQ to SQL处理器不会尝试执行多条查询语句，获取多个数据到客户端，更不能在客户端进行后期连接处理。

若你正在开发类库，且计划让其能够配合任意数据源使用，那么必须预料到这种情况。需要保证代码可以正确地配合任意数据源使用。这就意味着lambda表达式必须作为内联代码分开编写，这样你的类库才能正常工作。

当然，这并不意味着你可以在类库中任意复制代码。而是让你在需要使用查询表达式以及lambda表达式时，在应用程序中创建不同的构件块。比如前面的例子，即可通过创建更大一些的重用块来改进。

```
private static IQueryable<Employee> LowPaidSalariedFilter
    (this IQueryable<Employee> sequence)
{
    return from s in sequence
           where s.Classification == EmployeeType.Salary &&
```

```
            s.MonthlySalary < 4000
            select s;
}

// 在其他地方
var allEmployees = FindAllEmployees();

// 找到第一个员工
var salaried = allEmployees.LowPaidSalariedFilter();

var earlyFolks = salaried.Where(e => e.YearsOfService > 20);

// 找到最近来的人
var newest = salaried.Where(e => e.YearsOfService < 2);
```

当然，不是所有的查询都能如此简单地修改成这样。你需要在方法调用链上找到那一部分可重用的处理逻辑，并将其抽取出来，从而避免重复出现的lambda表达式。条目17（第3章）中介绍了枚举方法直到开始遍历集合中元素时才开始真正执行。记住这一点，即可创建出一系列包含常用lambda表达式的小方法，用来构造最终的查询。这些小方法必须接受序列作为参数，且必须使用yield return关键字来返回序列。

按照同样的模式，你也可以通过构造支持在远程执行的新表达式树来实现IQueryable的枚举器。这里，查找指定员工的表达式树可以在其真正执行之前与其他表达式组成一个新的查询。随后，IQueryProvider对象（例如LINQ to SQL数据源）将处理这个完整的查询，而不会让其中的一部分在本地执行。

这样，即可将这一系列小方法组合起来，构造出应用程序中需要的大型查询。这样做的好处是避免了代码重复，即解决了本条目开始部分的问题。同时也将代码更好地组织了起来，以便在构造完查询并即将实际执行时创建一棵表达式树，随后执行完整的查询。

在复杂查询中重用lambda表达式的最有效的一种方法就是，为这些操作封闭泛型类型的查询创建扩展方法。找到较低工资员工的方法就是一个这样做的例子。该方法接受一个员工序列作为输入，并返回另一个经过过滤的员工序列。在实际应用中，你还应该同时提供一个接受IEnumerable<Employee>参数的

重载。这样即可同时支持LINQ to SQL和LINQ to Objects两种风格的实现。

通过将这些包含了lambda表达式的小方法恰当地组合起来，即可构造出所需要的最终查询。你的代码也可以同时配合IEnumerable<T>和IQueryable<T>使用。此外，这样做也不会影响到对IQueryable<T>实现所需要的表达式树的求值。

条目 39：避免在函数或操作中抛出异常

若你的函数或操作需要操作一个序列的对象，且在处理的过程中抛出了异常，那么将会很难恢复到抛出异常前的状态。我们不知道已经处理了多少个元素，也不知道应该怎样回滚，因此根本无法返回到程序的先前状态。

考虑如下这段代码，用来给每个员工增加百分之五的薪水。

```
var allEmployees = FindAllEmployees();
allEmployees.ForEach(e => e.MonthlySalary *= 1.05M);
```

可是有一天，这个程序在运行时抛出了异常。很有可能抛出异常的位置并不在第一个或是最后一个员工上。因此一些员工得到了加薪，而另一些则没有。这样，程序将很难恢复到从前的状态。程序还能让数据回到一致的状态吗？一旦丢失了程序的状态，那么除了人工检查之外，将没有办法重新找回状态信息。

上述代码修改元素的方式导致了发生这样的问题。这段代码并没有遵循强异常安全保证（strong exception guarantee）的规则。若是运行时遇到了错误，你无法得知具体发生了什么，没有发生什么。

若是能够保证当方法无法完成时，程序的状态不会发生变化，那么即可避免此类情况的发生。有几种方法可以实现这个需求，每一种都有它自身的优势和风险。

在开始讨论其风险之前，我们先来深入解释一下这样做的目的。不是所有的方法都会遇到此类问题。很多方法仅仅检查了序列中的元素，但并没有修改这些元素。下面的这个方法就逐一查看了每个人的薪水，并返回薪水的总数。

```
decimal total = allEmployees.Aggregate(0M,
    (sum, emp) => sum + emp.MonthlySalary);
```

对于这类并不会修改序列中元素的方法，我们并不需要太过小心。在应用程序中，你会发现大多数的方法都不会修改序列。不过回到第一个方法中，即给每个员工加薪百分之五。若想遵守强异常安全保证，那么又应该如何修改该方法呢？

第一种也是最简单的一种，就是重写前面以lambda表达式给出的操作方法，让其永远不会抛出异常。很多时候，在开始修改序列中各个元素之前，预先验证其每一种失败的条件并不困难（参见第3章条目25）。你需要小心地定义函数和谓词，以便其能够满足所有情况下的需求，包括错误情况。若是可以跳过将会导致异常的元素，那么可以采用这个做法。在提升工资的例子中，假设所有的异常都是因为提升了那些已经不在该公司工作、但仍旧存放在公司员工数据库中员工的工资造成的。那么此时，跳过这些员工则是正确的行为。可以按照如下进行修改。

```
allEmployees.FindAll(
    e => e.Classification == EmployeeType.Active).
    ForEach(e => e.MonthlySalary *= 1.05M);
```

这是最简单的一种修复问题，保证算法一致性的做法。若你可以保证让操作方法或lambda表达式不会抛出任何异常，那么这种做法将是你的最佳选择。

不过，有些时候你不能保证处理方法不会抛出异常。这时则必须采取一些代价更加高昂的处理方式。在编写算法时，就应该考虑到抛出异常后的处理方法。这就意味着首先在原数据的副本上执行操作，随后仅在操作成功执行之后再将其替换原有的序列。例如，若你觉得终究无法预先判断出所有导致异常的情况，那么应该按照如下方式重写代码。

```
var updates = (from e in allEmployees
               select new Employee
               {
                   EmployeeID = e.EmployeeID,
                   Classification = e.Classification,
                   YearsOfService = e.YearsOfService,
```

```
                    MonthlySalary = e.MonthlySalary *= 1.05M
                }).ToList();
allEmployees = updates;
```

可以看到这些改变所带来的代价。首先，代码量大大增加。这就造成了更多的工作——更多需要维护、需要理解的代码。此外，应用程序的性能也有所下降。这段代码为所有的员工创建了一份副本，在修改完成后又用这个副本替换了原有的员工列表。若员工列表比较庞大，那么这将造成一个巨大的性能瓶颈。在替换原有员工列表之前，程序必须复制每一个员工的信息。但这样做的好处是，该方法即可针对非法的员工对象自由地抛出异常，外部代码也可以根据需要处理这些情况。

不过这种修改还有一个问题：这样做是否有意义取决于其使用方法。这个版本限制了你用多个函数将各种操作组合起来的能力。这段代码将缓存完整的列表，也就意味着它的修改并不能与其他只要一次遍历列表的操作组合使用。每个转换都变成了命令式的操作。在实际应用中，你可以将所有的危险转换操作都放在一个方法中。在该方法中先缓存列表，接下来执行这些危险的操作，最后替换原有序列。这样，既能够保留现有API的组合功能，又得到了强异常安全保证。

在实际中，这意味着让查询表达式返回新的序列，而不是修改原有序列中的元素。这样，每个组合而成的查询在正确执行之后都应该构造出原有列表的副本，除非处理过程的某一步抛出了异常。

组合查询改变了编写安全处理异常的代码的方式。若你的函数或操作抛出异常的话，那么可能没有办法保证数据的一致性。你不知道多少个元素已经被处理过，也不知道应该怎样操作才能将数据恢复到原始状态。不过，返回新的元素（而不是修改现有元素）则能够让你在尝试完成所有的操作的同时，也不影响到程序的任何状态。

对于所有将要修改元素且可能抛出异常的方法，都应该遵守这个规范。这同样也适用于多线程环境中。这时，若是lambda表达式中可能会抛出异常，那么问题将更加难以发现。在最后一步时，若所有操作都顺利完成，那么应该完

整替换原有的列表。

条目 40：区分早期执行和延迟执行

声明式代码（declarative code）是解释性的，它定义了将要完成什么工作。而*命令式代码*（imperative code）则一步一步地给出了完成指定工作需要的各个步骤。两种代码风格都是合法的，也都能写出不错的程序。不过若是将二者混合起来，那么将有可能导致无法预测的行为。

命令式代码均会预先计算出所有的参数，然后调用方法。如下这一行代码就演示了用命令式代码分步骤完成某项工作的做法。

```
object answer = DoStuff(Method1(),
    Method2(),
    Method3());
```

在运行时，这行代码将按照如下顺序执行。

(1) 调用Method1，生成DoStuff()所需要的第一个参数。
(2) 调用Method2，生成DoStuff()所需要的第二个参数。
(3) 调用Method2，生成DoStuff()所需要的第三个参数。
(4) 使用计算得到的三个参数调用DoStuff。

你应该很熟悉这种代码风格。先计算出需要的所有参数，然后一次性地传递给待调用方法。其代码包含了一系列描述步骤，遵循这种步骤才能得到期待中的结果。

而lambda表达式和查询表达式所引入的*延迟执行*（deferred execution）则完全地改变了这个流程，甚至完全颠覆了你的想法。下面的这行代码看似与前一个例子相同，不过很快你就会发现其中的不同。

```
object answer = DoStuff(() => Method1(),
    () => Method2(),
    () => Method3());
```

在运行时，这行代码将按照如下顺序执行。

(1) 调用DoStuff()，传入可以调用Method1、Method2和Method3的lambda表达式。

(2) 在DoStuff中，仅在需要Method1的执行结果时，才会调用Method1。

(3) 在DoStuff中，仅在需要Method2的执行结果时，才会调用Method2。

(4) 在DoStuff中，仅在需要Method3的执行结果时，才会调用Method3。

(5) Method1、Method2和Method3可能会以任意的顺序调用，并调用任意多次（也可能是零次）。

注意，只有确实需要某个方法的计算结果时，才会调用该方法。二者的不同显而易见，因此若将两种编程方式混合使用的话，将有可能造成严重的问题。

从外部看来，只要方法不会做额外的操作，那么方法调用都可以用其返回值替代，反之亦然。在我们的例子中，两种策略下DoStuff()对方法并没有什么影响。DoStuff()将返回同样的值，两种策略也都是正确的。若是某个方法对于同样的输入总是返回同样的计算结果，那么该方法的返回值则总是能被对该方法的调用所替代，反之亦然。

不过若是将应用程序作为一个整体来看，那么这两行代码则有可能发生显著的不同。命令式模型总会调用所有的三个方法。三个方法中可能进行的额外操作也会且仅会发生一次。作为对比，声明式模型可能会也可能不会调用某个甚至所有的三个方法，也可能多次执行同一个方法。这就是(1)调用方法并将其返回值传入另一个方法，与(2)将方法以委托形式传入并让外部方法来调用委托之间的区别。在多次运行同一个应用程序时，你可能会得到不同的结果，这取决于方法的具体实现。

最新添加的lambda表达式、类型推断以及枚举器等特性让我们很容易地在类型中使用到函数式编程的概念。你可以编写高阶函数，让其接受函数作为参数，或将函数返回给调用者。从某一角度而言，这不会成为问题：一个真正的函数及其返回值总会是可以互相代替的。而在实际中，函数往往会做一些额外操作，而这些额外操作则有可能会导致程序运行结果的不确定。

若数据和方法之间是可以互换的，那么你会选择哪个呢？此外更为重要的是，何时应该选择哪个？二者最主要的区别就是，数据必须预先准备好，

而方法则可以延迟求值。当你需要预先求出数据时，则应该先运行方法，用其返回值作为数据，而不是用函数式的方式将方法本身传递过去。

选择使用哪种方式的最重要判断依据是，方法调用时是否会产生副作用。副作用可能发生在函数内部以及其返回值的可变性上。条目37（本章前面部分）中演示了一个执行结果依赖于当前时间的查询。是缓存其调用结果，还是以函数参数的形式使用该查询，这个选择将会影响到该查询的返回值。可以看到，若像示例中的那样，函数本身将产生副作用，那么程序的行为也就会依赖于执行该函数的时间。

有一些技术可以降低早期求值（early execution）和延迟求值之间的差异。纯粹的不可变类型无法被修改，因此也不会改变程序中其他的状态，自然不会带来副作用。在前面的示例中，若Method1、Method2和Method3都是某个不可变类型的成员，那么早期求值和延迟求值语句的行为应该会完全一致。

很多示例并没有演示接受参数的情况，不过若是某个延迟求值的方法需要接受参数，那么参数必须也为不可变的，这样才能让早期求值和延迟求值的执行结果保持一致。

因此，决定选择早期求值和延迟求值的关键在于你想要实现的语义。当且仅当对象和方法是不可变的，你才能够将某个值替换成计算出该值的方法，反之亦然。（“不可变的方法”是指那些不会修改任何全局状态，例如执行I/O操作、修改全局变量或与其他进程通信的方法。）若对象或方法是可变的，那么在早期求值和延迟求值之间切换则有可能会改变程序的行为。本条目的接下来部分将假设在早期求值和延迟求值之间切换不会影响到程序的行为，并基于这样的假设分析两种策略之间其他的不同之处。

另一个考虑之处是输入和输出变量所占用的空间以及计算输出所花费的代价。例如，若是在使用π时Math.PI才开始计算，那么程序依然可以正常工作。对于外部而言，π的值和其计算是可以互相替代的。不过程序将因此变慢，因为计算π需要时间。而一个名为CalculatePrimeFactors(int)的方法当然也可以把所有整数的所有因数存放在查找表中。不过这样的话，存放查找表所

带来的内存耗费要远远大于在需要时计算对性能的影响。

在实际开发中，一般不会遇到如此极端的情况。正确的解决方案也不会如此地直观清晰。除了分析比较计算代价和存储代价之外，还需要考虑你将如何使用某个特定方法的执行结果。在某些情况下，早期执行特定的查询也有它存在的意义。而在另一些情况下，只是偶尔会用到这些中间结果。若能够确保代码不会产生副作用，且早期求值和延迟求值都能得到正确的结果，那么则可以根据两个方法的性能来选择最佳的策略。你可以尝试使用一下，比较其不同，然后作出最好的选择。

但在另一些情况下，你会发现将两种策略混合使用才是最好的做法。有时，适当的缓存将极大提高效率。对于这种情况，可以使用委托来返回缓存中的数据。

```
MyType cache = Method1();
object answer = DoStuff(() => cache,
    () => Method2(),
    () => Method3());
```

最后一个考虑之处是该方法是否需要操作远程的数据存贮介质。这对于LINQ to SQL来说格外重要。每个LINQ to SQL查询在开始时都是延迟查询：这些方法（而不是数据）起到的是参数的作用。有一些方法中的工作将会在远程数据库引擎中完成，而有一些表示本地工作的方法必须在将查询提交给数据库之前完成。LINQ to SQL将解析表达式树，并在将查询发送给数据库之前，将所有本地方法调用都替换成该方法的调用结果。为了实现这些，LINQ to SQL必须保证这些方法不依赖于那些正在处理的输入序列[①]（参见本章条目37和条目38）。

在把所有本地方法调用都替换成与之等同的方法返回值之后，LINQ to SQL将把查询表达式转换成SQL语句，发送给数据库并执行。通过这样的过程，LINQ to SQL即可支持通过一系列表达式或代码来创建查询，最终将其替换成SQL语句。这样做提高了性能，也降低了网络带宽的占用。LINQ to SQL让C#

① 即方法内部的逻辑不会操作LINQ to SQL将要从数据库中获取的实体对象。——译者注

开发者不必精通T-SQL，其他的各种LINQ提供器也能类似地简化开发人员的工作。

不过，之所以可以实现这些，是因为在这些情况下代码和数据可以互换。在LINQ to SQL中，若是本地方法的参数是不变的，且不依赖于输入序列，那么本地方法就能被替换成其返回值。当然，LINQ to SQL类库本身也提供了很多功能，以便将表达式树转换成专门组织的结构，并最终转换成T-SQL语句。

此刻，当你在编写C#程序时，应该可以判断出用数据作为参数和用函数作为参数这两种方法是否会给程序带来不同的行为。若是二者均可以采用的话，那么则应该判断哪一种才是更好的选择。若空间占用不大，那么直接传入数据可能会好一些。不过在另一些情况中，输入或输出空间可能会非常大，且你也并不需要完整的数据，那么则可以考虑将算法本身作为参数传入到需要的方法中。若无法确定的话，那么可以将算法作为参数传入，因为如果有确实的需要，函数的使用者完全可以先对传入的函数进行主动求值，然后再使用计算出的结果。

条目 41：避免在闭包中捕获昂贵的外部资源

闭包将创建出包含绑定变量的对象。这些绑定变量的生命周期可能会大大出乎你的意料，且不一定会符合你的期待。作为开发者，我们习惯于用一种很简单的方法来判断局部变量的生命周期：在变量来到声明该变量的代码区域中时，其生命周期正式开始；而当该代码区域结束后，变量的生命周期也宣告终止。在离开所在的代码区域之后，该局部变量即可被垃圾收集。我们正是使用这样的假设来管理资源使用以及对象生命周期的。

闭包和其中捕获的外部变量改变了这些规则。当你在闭包中捕获外部变量时，那么除非最后一个引用了该变量的委托离开了作用域，否则该变量所引用的对象不会终止其生命周期。在某些情况下，这类对象的生命周期将会更长。在闭包和其中捕获的变量被暴露到方法之外时，还可能被客户代码的闭包和委

托访问到。这些委托和闭包还可以进一步被其他代码访问，如此进行下去。最终，访问你的委托的代码将变得无穷无尽，以至于无法判断何时你的闭包和委托才会变得不可达。这也就意味着，若你暴露了使用了被捕获变量的委托，那么你将无法控制这些局部变量将何时离开其作用域。

不过好消息是，通常而言我们并不需要关注这个行为。托管且并没有持有昂贵资源的的局部变量将在稍后被垃圾收集，就像普通的变量一样。若这些额外的开销仅仅体现在内存占用上，那么其实并没有什么好担心的。

不过有些变量却持有着昂贵的资源。其类型实现了IDisposable，且需要在使用后显式清理。因此，你可能会在遍历集合之前就过早地清理掉了这些资源，也可能并没有及时关掉文件或连接，以至于无法再次访问到文件（因为已经被打开了）。

条目33（第4章）演示了C#编译器是如何生成委托并在闭包中捕获局部变量的。在该条目中，我们将分析捕获持有昂贵资源的局部变量时发生的情况。本条目将介绍如何管理这些资源，并且能在被捕获变量生命周期超过你期待时避免常见的陷阱。

比如这段代码：

```
int counter = 0;
IEnumerable<int> numbers =
    Extensions.Generate(30, () => counter++);
```

将生成类似如下的代码：

```
private class Closure
{
    public int generatedCounter;
    public int generatorFunc()
    {
        return generatedCounter ++;
    }
}

// 使用
Closure c = new Closure();
c.generatedCounter = 0;
```

```
IEnumerable<int> sequence = Extensions.Generate(30, new
Func<int>(c.generatorFunc));
```

这段代码很值得玩味。该隐藏内嵌类型的成员将绑定到Extensions.Generate所使用的委托上。这样就会影响到隐藏对象的生命周期，进而影响到该对象成员可以被垃圾收集的时期。查看如下代码。

```
public IEnumerable<int> MakeSequence()
{
    int counter = 0;
    IEnumerable<int> numbers = Extensions.Generate(30,
        () => counter++);
    return numbers;
}
```

在这段代码中，返回值使用了由闭包绑定的委托。因为返回值需要该委托，所以委托的生命周期就越过了该方法。表示绑定变量的对象的生命周期也被延长了。对象因为委托实例可达而继续保持可达，委托实例可达是因为它是返回值的一部分。此外，考虑到对象本身可达，所以其所有的成员也是可达的。

C#编译器将生成类似如下的代码。

```
public static IEnumerable<int> MakeSequence()
{
    Closure c = new Closure();
    c.generatedCounter = 0;
    IEnumerable<int> sequence = Extensions.Generate(30,
        new Func<int>(c.generatorFunc));
    return sequence;
}
```

注意到该序列包含了一个委托，而委托又引用到了一个绑定到c的方法，其中c是闭包中初始化的一个局部变量。这样，局部变量c的生命周期就将长于方法的生命周期。

通常情况下，这样并不会导致什么问题。不过在两种情况下则可能有所不妥。第一种情况涉及了IDisposable。考虑如下代码，其中从CSV输入流中读取了数字，然后将其作为一个二维序列（序列中包含序列）返回。每个内层的序列都包含了CSV中某一行中的数字。这段代码使用到了条目28（第4章）中的一些扩展方法。

```
public static IEnumerable<string> ReadLines(
    this TextReader reader)
{
    string txt = reader.ReadLine();
    while (txt != null)
    {
        yield return txt;
        txt = reader.ReadLine();
    }
}

public static int DefaultParse(this string input,
    int defaultValue)
{
    int answer;
    return (int.TryParse(input, out answer))
        ? answer : defaultValue;
}

public static IEnumerable<IEnumerable<int>>
    ReadNumbersFromStream(TextReader t)
{
    var allLines = from line in t.ReadLines()
                   select line.Split(',');
    var matrixOfValues = from line in allLines
                         select from item in line
                                select item.DefaultParse(0);
    return matrixOfValues;
}
```

可以这样使用：

```
TextReader t = new StreamReader("TestFile.txt");
var rowsOfNumbers = ReadNumbersFromStream(t);
```

我们知道，该查询仅在需要访问下一个值时，才会开始生成该值。ReadNumbersFromStream()并没有将所有的数据都放在内存中，而是根据需要再从流中加载。上面的这两行代码并没有开始实际读取文件，当程序在稍后开始遍历rowsOfNumbers时，才会开始打开文件并读取文件。

不过在稍后的某次代码审查中，Alexander指出你并没有显式地关闭该文本文件。他可能是因为察觉了资源泄露，或是希望读取同样一个文件时得到了错误才发现了这个问题。因此，你按照如下方式修复了问题。但这并没有抓住事物的本质。

```
IEnumerable<IEnumerable<int>> rowOfNumbers;
using (TextReader t = new StreamReader("TestFile.txt"))
    rowOfNumbers = ReadNumbersFromStream(t);
```

你会很兴奋地测试一下，期待其问题的解决，不过在几行之后，程序却抛出了异常。

```
IEnumerable<IEnumerable<int>> rowOfNumbers;
using (TextReader t = new StreamReader("TestFile.txt"))
    rowOfNumbers = ReadNumbersFromStream(t);

foreach (var line in rowOfNumbers)
{
    foreach (int num in line)
        Console.Write("{0}, ", num);
    Console.WriteLine();
}
```

怎么会这样呢？提示说在关闭后还会继续访问该文件。迭代的过程抛出了ObjectDisposedException异常。这是因为C#编译器将TextReader绑定到了读取文件并解析的委托之上。这一系列的代码仅仅是通过arrayOfNums表达了出来，还没有真正进行什么操作。既没有开始读取流，也没有开始解析。这也是将资源交给调用者管理所造成的问题之一。若调用者没有充分理解资源的生命周期，那么将可能会导致资源泄露或破坏程序的功能。

正确的修改方法很简单。移动几行代码，让程序在使用了arrayOfNums之后再将文件关闭掉即可。

```
using (TextReader t = new StreamReader("TestFile.txt"))
{
    var arrayOfNums = ReadNumbersFromStream(t);

    foreach (var line in arrayOfNums)
    {
```

```
            foreach (var num in line)
                Console.Write("{0}, ", num);
            Console.WriteLine();
        }
    }
```

解决方法并不难，不过不是所有的问题都会如此简单。这样的策略将导致出现很多重复代码，这是开发人员不想看到的。因此接下来我们来看看有没有什么更通用的解决方案。前面这段代码之所以不再出现问题，是因为在关闭文件之前就使用完了该数组。

不过若是按照这样的方式组织代码，那么将很难找到合适的关闭文件位置。这个API需要在一个地方打开文件，然后在另外一处才能关闭。该API的原有使用方式是这样的。

```
using (TextReader t = new StreamReader("TestFile.txt"))
    return ReadNumbersFromFile(t);
```

但这样的话就无法正常地关闭文件。文件在这里打开，不过在调用栈的某处，终究需要关闭该文件。但又是在何处呢？你不知道，因为这并不在你的代码中，而是在调用栈的别处。这不在你的控制之下，在需要关闭时，开发者甚至都不知道文件名称，也丢失了该流的引用，以至于根本无法关闭。

另一个简单的解决方案是使用一个专门的方法来打开文件，读取内容并返回序列。类似如下代码。

```
public static IEnumerable<string> ParseFile(string path)
{
    using (StreamReader r = new StreamReader(path))
    {
        string line = r.ReadLine();
        while (line != null)
        {
            yield return line;
            line = r.ReadLine();
        }
    }
}
```

这个方法使用了与条目17（第3章）中一样的延迟执行模型。不过其很重

要的一点是，只有在读取了所有的元素之后（不管是什么时候），StreamReader才会被释放。在遍历过返回序列后，该文件对象肯定会被关闭。下面有个简单的小例子可以作为解释。

```
class Generator : IDisposable
{
    private int count;
    public int GetNextNumber()
    {
        return count++;
    }

    #region IDisposable Members
    public void Dispose()
    {
        Console.WriteLine("Disposing now ");
    }
    #endregion
}
```

Generator类实现了IDisposable接口，不过仅仅用来说明捕获IDisposable对象时将要发生的事情。下面是其简单的应用。

```
var query = (from n in SomeFunction()
             select n).Take(5);

foreach (var s in query)
    Console.WriteLine(s);


Console.WriteLine("Again");
foreach (var s in query)
    Console.WriteLine(s);
```

上述代码的输出如下：

```
0
1
2
3
4
Disposing now
```

```
Again
0
1
2
3
4
Disposing now
```

Generator如我们期待的那样按时被释放，即完成第一次迭代之后。无论是完成迭代，还是在迭代中停了下来，Generator都会被释放。

不过这里还存在着一个问题。注意到输出中包含了两次“Disposing now”。这是因为代码遍历了两次序列，因此也让Generator释放了两次。对于Generator类来说这并不是问题，因为重复释放只不过是多输出了一些而已。不过若是对于文件，那么在第二次遍历的时候就会抛出异常。因为在第一次遍历结束后，StreamReader就会被释放，所以在第二次遍历尝试访问已经被释放的StreamReader时，自然无法正常执行。

若应用程序可能需要对IDisposable对象进行多次遍历，那么则要使用另一种不同的方式。可能应用程序需要读取多个值，然后在算法执行的过程中用不同的方式来处理。若是这样的话，即可考虑将该算法或多个算法以委托的形式传入到读取并处理文件的方法中。

你还可以使用泛型来编写这个方法，并支持从外部传入对读取到的数据的处理方式。这样即可在释放文件之前就完成了需要的操作。改写后的代码将类似如下所示。

```
// 使用方法：参数为文件
// 以及针对文件每一行将要执行的操作
ProcessFile("testFile.txt",
    (arrayOfNums) =>
    {
        foreach (IEnumerable<int> line in arrayOfNums)
        {
            foreach (int num in line)
                Console.Write("{0}, ", num);
            Console.WriteLine();
        }
```

```
            // Make the compiler happy by returning something:
            return 0;
        }
);

// 声明委托类型
public delegate TResult ProcessElementsFromFile<TResult>(
    IEnumerable<IEnumerable<int>> values);

// 读取文件并使用委托处理每一行的方法
public static TResult ProcessFile<TResult>(string filePath,
    ProcessElementsFromFile<TResult> action)
{
    using (TextReader t = new StreamReader(filePath))
    {
        var allLines = from line in t.ReadLines()
                       select line.Split(',');

        var matrixOfValues = from line in allLines
                             select from item in line
                                    select
                                        item.DefaultParse(0);
        return action(matrixOfValues);
    }
}
```

看上去似乎有些复杂，不过其使用方法却非常简单，并支持各种不同的操作。例如，若你想找到文件中最大的数字。

```
var maximum = ProcessFile("testFile.txt",
    (arrayOfNums) =>
        (from line in arrayOfNums
            select line.Max()).Max());
```

这样，对文件流的使用被完整地封装到了ProcessFile中。你所希望找到的是一个数值，将会由lambda表达式返回。通过这样修改，即可让那些昂贵的资源（这里是文件流）在函数内部创建并释放，你也无需在闭包中添加此类昂贵的成员。

闭包中捕获昂贵资源所导致的其他问题并不是那么严重，不过仍有可能影响到程序的性能。考虑如下方法。

```
IEnumerable<int> ExpensiveSequence()
{
    int counter = 0;
    IEnumerable<int> numbers = Extensions.Generate(30,
        () => counter++);

    Console.WriteLine("counter: {0}", counter);

    ResourceHog hog = new ResourceHog();
    numbers = numbers.Union(
        hog.SequenceGeneratedFromResourceHog(
        (val) => val < counter));
    return numbers;
}
```

与其他闭包类似，在延迟执行模型中，上述lambda表达式中的代码将稍后执行。这也就意味着ResourceHog将超过该方法的生命周期，生存到客户代码遍历完序列之后。此外，若ResourceHog没有被释放，那么将存活到其所有的引用都变得不可达，然后垃圾收集器才会清理该资源。

若这变成了性能瓶颈，那么可以重新组织一下查询，将ResourceHog生成数字改为主动求值。这样ResourceHog即可被立即清理。

```
IEnumerable<int> ExpensiveSequence()
{
    int counter = 0;
    IEnumerable<int> numbers = Extensions.Generate(30,
        () => counter++);

    Console.WriteLine("counter: {0}", counter);

    ResourceHog hog = new ResourceHog();
    IEnumerable<int> mergeSequence =
        hog.SequenceGeneratedFromResourceHog(
            (val) => val < counter).ToList();
```

```
        numbers = numbers.Union(mergeSequence);
        return numbers;
    }
```

这个示例很清楚，因为代码并不复杂。若你的算法非常复杂，那么将很难把普通资源和昂贵资源分离开来。根据创建闭包的方法中算法的复杂程度，可能会很难将闭包中各个绑定变量中不同的资源分开。下面的这个方法就在闭包中捕获了三个不同的局部变量。

```
private static IEnumerable<int> LeakingClosure(int mod)
{
    ResourceHogFilter filter = new ResourceHogFilter();
    CheapNumberGenerator source = new CheapNumberGenerator();
    CheapNumberGenerator results = new CheapNumberGenerator();

    double importantStatistic = (from num in
                                 source.GetNumbers(50)
                                 where
                                     filter.PassesFilter(num)
                                 select num).Average();

    return from num in results.GetNumbers(100)
           where num > importantStatistic
           select num;
}
```

第一眼看上去，这似乎没什么问题。ResourceHog用来生成重要的统计信息。起作用范围被限制在方法中，且在方法结束后就会被垃圾回收。

不过实际上，这个方法执行的并不像你想象中的那么好。

C#将为每个代码作用区域创建一个内嵌类，用来实现闭包。最终的查询语句——用来返回大于重要统计信息的数字——需要使用闭包来容纳绑定变量（importantStatistic）。在该方法的开始部分，筛选器用来在闭包中创建importantStatistic。这也就意味着筛选器将被复制到实现该闭包的内嵌类中。return语句返回的类型又使用了内嵌类的实例来实现where子句。这样，实现闭包的内嵌类就泄露到了方法的外面。通常情况下这无伤大雅。不过若是ResourceHogFilter占用了昂贵的资源，那么就会影响到程

序的性能。

若想修复该问题，你可将方法分为两个部分，让编译器创建成两个闭包类。

```
private static IEnumerable<int> NotLeakingClosure(int mod)
{
    var importantStatistic = GenerateImportantStatistic();

    CheapNumberGenerator results = new CheapNumberGenerator();
    return from num in results.GetNumbers(100)
           where num > importantStatistic
           select num;
}

private static double GenerateImportantStatistic()
{
    ResourceHogFilter filter = new ResourceHogFilter();
    CheapNumberGenerator source = new CheapNumberGenerator();

    return (from num in source.GetNumbers(50)
            where filter.PassesFilter(num)
            select num).Average();
}
```

“等一下，”你可能会说，“GenerateImportantStatistic中的return语句中包含了生成统计信息的查询，闭包仍旧会泄露。”但实际上是不会的。Average方法将需要整个序列（本章条目40）。遍历操作发生在GenerateImportantStatistic中，且仅返回了平均值。因此在该方法返回后，包含了ResourceHogFilter对象的闭包即可被很快被回收。

之所以这样重写该方法，是因为在编写带有多个逻辑上闭包的方法时，还可能出现更多的问题。虽然你会认为编译器将创建多个闭包，不过编译器实际上仅会创建一个，并用其处理整个作用域内所有的lambda表达式。你会留意将要被返回出去的那个表达式，但可能会认为其他表达式不会有任何影响。而实际上其他的表达式也会产生影响。因为编译器对一个作用域只会创建一个类，并用其处理其中所有的闭包，所有闭包中使用到的所有成员都会被放入到该类

中。例如如下这个小方法：

```
public IEnumerable<int> MakeAnotherSequence()
{
    int counter = 0;

    IEnumerable<int> interim = Extensions.Generate(30,
        () => counter++);
    Random gen = new Random();

    IEnumerable<int> numbers = from n in interim
                               select gen.Next() - n;
    return numbers;
}
```

MakeAnotherSequence()包含了两个查询。第一个查询生成了从0~29的整数序列。第二个查询将使用随机数发生器对该序列进行修改。C#编译器会生成一个私有类来实现该闭包，其中包含了counter和gen。调用了MakeAnotherSequence()的代码将访问到该生成类的一个实例，自然也包含了counter和gen这两个局部变量。需要注意的是编译器并没有创建两个内嵌类，而仅创建了一个。该嵌套类的实例最终将被传递给调用者。

与闭包中操作发生的时机相关的还有一个问题。比如下面的示例：

```
private static void SomeMethod(ref int i)
{
    //...
}
private static void DoSomethingInBackground()
{
    int i = 0;
    Thread thread = new Thread(delegate()
        { SomeMethod(ref i); });
    thread.Start();
}
```

在该示例中，我们捕获了一个变量并在两个线程中使用。此外，代码还让两个线程都通过引用来访问该变量。虽然我很想完整地解释运行该示例后i的变化，不过实际上没人知道将会发生什么。两个线程都可以检查并修改i，同时

也取决于哪个线程执行得更快一些，每个线程都能够在任何时间修改该变量。

在使用查询表达式时，编译器将为整个方法中的所有表达式创建一个闭包。该闭包类型的一个对象可能会被方法返回，例如返回的是类型中某个实现了集合接口的成员。这样，该对象将生存在系统中，直到不再有任何对象引用到它。这样就可能会造成很多问题。若被封装到闭包中的某个字段实现了IDisposable，那么可能会引发程序的错误。若是复制某个字段需要较长时间，那么则可能带来性能问题。无论怎样，你都需要理解当方法返回了由闭包创建的对象时，该闭包将包含所有需要用来执行计算的变量。因此，你要确认程序的确需要用到这些变量，或者，若不能确定的话，也要让闭包能够尽快地自动清理掉这些额外占用的资源。

条目 42：区分 IEnumerable 和 IQueryable 数据源

IQueryable<T>和IEnumerable<T>这两个API的签名非常相似。IQueryable<T>继承自IEnumerable<T>。因此你可能会认为这两个接口是可以互换的。很多情况下确实如此，因为它们就是这样设计的。作为对比，我们知道序列和序列虽然有其共性，不过却并不总是可以互相代替使用的。IQueryable<T>和IEnumerable<T>的行为有所不同，且其性能方面也可能会天差地别。例如，下面的这两个查询语句则完全不同。

```
var q =
    from c in dbContext.Customers
    where c.City == "London"
    select c;
var finalAnswer = from c in q
                  orderby c.Name
                  select c;
// 省略掉遍历finalAnswer的代码
var q =
    (from c in dbContext.Customers
    where c.City == "London"
    select c).AsEnumerable();
```

```
var finalAnswer = from c in q
                  orderby c.Name
                  select c;

// 省略掉遍历finalAnswer的代码
```

这两个查询能返回同样的结果，不过其实现方式则大相径庭。第一个查询使用了LINQ to SQL的处理方式，基于IQueryable构造。而第二个查询则将数据库中的对象强制转换成IEnumerable序列，且很多操作都是在本地完成的。这种方法组合使用了延迟求值和LINQ to SQL中的IQueryable<T>。

在开始遍历查询结果时，LINQ to SQL类库将会根据其每个查询语句构造出结果。在这个例子中，LINQ to SQL将发出对数据库的请求，并在数据库中用SQL语句来处理where和orderby子句。

在第二种情况下，将第一个查询的结果转为IEnumerable<T>序列意味着接下来的操作将用LINQ to Objects实现（具体实现中使用了委托）。第一条语句将向数据库发出请求，获取所有位于伦敦的客户。第二条语句将把第一条调用的返回集合排序，该排序操作是在本地进行的。

对于这些不同，你必须非常小心，因为很多查询在使用IQueryable功能时要比使用IEnumerable高效很多。此外，因为IQueryable和IEnumerable处理查询表达式的做法各不相同，因此你会发现在一处工作正常的查询，迁移到另一处却会无法运行。

IQueryable和IEnumerable处理查询表达式的做法有着根本上的差异。这是因为二者使用的类型截然不同。Enumerable<T>扩展方法使用委托来实现lambda表达式，并应用了查询表达式中出现的函数参数。而Queryable<T>则使用表达式树来处理这些元素。表达式树是一种专门的特殊结构，用来组织并存放查询中所有的逻辑。Enumerable<T>查询必须在本地执行。lambda表达式将被编译成方法，随后立即在本地计算机中执行。这就意味着我们需要首先把所有的数据都加载到本地应用程序中，这样将不得不传输更多的数据，随后还要将没有用到的数据丢掉。

而作为对比，IQueryable处理方式将首先解析表达式树。在解析完成之后，Queryable处理方式将会把表达式树中的逻辑转换成能够被当前提供器理解的形式，随后在靠近数据实际存放位置的地方执行。这样就大大降低了数据的传输量，并从总体上提高了系统的性能。不过，在使用IQueryable接口并依赖Queryable<T>实现序列时，会对将要转换成查询表达式的代码有一定的限制。

正如我在本章前面部分条目37中介绍过的那样，IQueryable并不能解析任意的方法，因为这将意味着无穷无尽多种逻辑。IQueryable能够理解一系列操作符以及一系列确定的方法，这是由.NET Framework实现的。若你的查询中还包含了其他的方法调用，那么可能会不得不使用IEnumerable的实现。

```
private bool isValidProduct(Product p) {
    return p.ProductName.LastIndexOf('C') == 0;
}
// 没有问题
var q1 =
    from p in dbContext.Products.AsEnumerable()
    where isValidProduct(p)
    select p;
// 在遍历查询结果时，将会抛出异常
var q2 =
    from p in dbContext.Products
    where isValidProduct(p)
    select p;
```

第一个查询没有问题，因为LINQ to Objects将使用委托以方法调用的形式实现查询。AsEnumerable()将把查询结果强制放在本地存储中，随后where子句即使用LINQ to Objects执行。而第二个查询将抛出异常，因为LINQ to SQL使用的是IQueryable<T>实现[①]。LINQ to SQL中的IQueryProvider将把查询转换成T-SQL。随后把该T-SQL发送至远程数据库引擎，接下来数据库引擎将在该上下文中执行该SQL语句（参见本章前面部分的条目38）。IQueryable<T>实现能够带来一个优势，那就是程序传输的数据量会大大减少。

考虑到性能和强壮性之间的平衡，你可以通过将查询结果显式转换成

① 而LINQ to SQL无法将isValidProduct()方法内部的逻辑转换为合适的SQL语句。——译者注

IEnumerable<T>来避免异常。不过这样做的劣势在于，LINQ to SQL将从数据库中返回dbContext.Products中所有的数据。此外，查询的其他部分都是在本地执行的。因为IQueryable<T>继承自IEnumerable<T>，所以调用isValidProduct()方法将不会出现问题。

这看上去不错，而且也非常简单。不过却强制所有需要使用isValidProduct()方法的代码都回到了IEnumerable<T>序列之上。即使开发人员使用的数据源支持IQueryable<T>，你也必须要求他将所有的数据都加载在本地程序的内存中，然后在本地进行处理，这样才能返回结果。

虽然偶尔编写这样的方法，将其放在那些不常用的类型中也无妨，不过不要用这种方式来对待IEnumerable<T>和IQueryable<T>。虽然IEnumerable<T>和IQueryable<T>对外部而言的功能类似，但其本质上是用来表示不同实现的，这意味着你应该根据所使用的数据源来选择合适的实现。在实际开发中，你当然会知道数据源实现了IQueryable<T>，或是仅支持IEnumerable<T>。若数据源支持IQueryable的话，你应该确保使用IQueryable的操作方式。

不过，有时对于同一个类型，你会需要让类必须同时支持IEnumerable<T>和IQueryable<T>上的查询。

```
public static IEnumerable<Product>
    ValidProducts(this IEnumerable<Product> products)
{
    return from p in products
           where p.ProductName.LastIndexOf('C') == 0
           select p;
}

// 没问题，因为LINQ to SQL提供器支持
public static IQueryable<Product>
ValidProducts(this IQueryable<Product> products)
{
    return from p in products
           where p.ProductName.LastIndexOf('C') == 0
           select p;
}
```

不过上述代码的重复度很高。使用AsQueryable()尝试将IEnumerable<T>转换成IQueryable<T>即可避免这个重复。

```
public static IEnumerable<Product>
    ValidProducts(this IEnumerable<Product> products)
{

    return from p in products.AsQueryable()
           where p.ProductName.LastIndexOf('C') == 0
           select p;
}
```

AsQueryable()将察看序列的运行时类型。若序列为IQueryable，那么将把序列作为IQueryable返回。而若是运行时类型为IEnumerable，那么AsQueryable()将会使用LINQ to Objects创建一个包装，从而实现IQueryable，然后再翻回该包装。虽然得到的仍是Enumerable，不过却用IQueryable引用包装了起来。

这样看来，使用AsQueryable()能够带来最大化的优势。已经实现了IQueryable的序列可以使用其高效的实现，而仅实现了IEnumerable的序列仍旧可以正常工作。当客户代码传递给你IQueryable序列时，你的代码将正确地使用Queryable<T>的方法，并假设其支持表达式树以及外部执行。而若是你操作的序列仅支持IEnumerable<T>，那么运行时也会降级使用IEnumerable的处理方法。

注意到该版本的代码仍旧使用了一个方法调用：string.LastIndexOf()。不过这个方法可以被LINQ to SQL类库正确解析，因此可以应用于LINQ to SQL查询中。不过，每个提供器都有它自己独特的功能，因此不要认为每个IQueryProvider实现都能支持这个方法。

IQueryable<T>和IEnumerable<T>看似提供了同样的功能。其所有的不同都源于二者对查询模式的不同实现。你需要确保程序声明的查询结果类型符合数据源的性质。查询方法均是静态绑定的，因此正确声明查询变量的类型才能得到正确的行为。

条目 43：使用 Single()和 First()来明确给出对查询结果的期待

第一眼看上去，LINQ类库似乎只能用来操作序列。不过实际上，LINQ也提供了几个方法能够从查询中得到单一的元素。这些方法各有不同，其各自的功能能够让你根据需要表达返回单一值的程序的意图。

Single()仅会返回序列中唯一的一个元素。若没有元素或有多个元素，那么Single()将抛出异常。这是一个非常强的约束，且如果执行失败的话，你应该立即找到原因。当你的查询确定要仅返回一个元素时，则应该适用Single()。这个方法非常清晰地表达了你的假设：查询只会返回一个元素。虽然在不满足条件时该方法会失败，不过这个失败会在第一时间发生，且不会造成数据错误。这个失败将帮你迅速诊断并纠正问题。此外，应用程序的数据也不会因为在稍后使用到了错误的数据而被破坏，因为假设一旦不成立，查询将立即失败，程序也无法继续运行。

```
var somePeople = new List<Person>{
    new Person {FirstName = "Bill", LastName = "Gates"},
    new Person { FirstName = "Bill", LastName = "Wagner"},
    new Person { FirstName = "Bill", LastName = "Johnson"}};

// 因为序列中包含多个元素，因此将抛出异常
var answer = (from p in somePeople
              where p.FirstName == "Bill"
              select p).Single();
```

此外，与其他查询不同的是，这个查询将在检查结果前就抛出异常。Single()将立即对查询进行求值，并返回该单一的元素。下面这个查询也将失败，并抛出同样的异常（虽然异常信息不同）。

```
var answer = (from p in somePeople
              where p.FirstName == "Larry"
              select p).Single();
```

这里，代码仍旧假设又且仅有一个结果存在。当实际情况不符合假设时，Single()将抛出InvalidOperationException异常。

若查询可能返回零个或一个元素，那么可以使用SingleOrDefault()。不过若是序列中包含了多于一个元素，那么SingleOrDefault()仍会抛出异常。使用SingleOrDefault()则意味着你假设查询表达式中返回的元素不会超过一个。

```
var answer = (from p in somePeople
              where p.FirstName == "Larry"
              select p).SingleOrDefault();
```

该查询将返回null（引用类型的默认值），表示数据源中没有符合该查询的元素。

当然，有时你也会期待得到多个元素，但仅获取其中的某一个。这时可以考虑使用First()或FirstOrDefault()。这两个方法都将返回序列中的第一个元素。若是序列为空的话，那么FirstOrDefault()将返回默认值[①]。下面的这个查询将返回进球数最多的前锋，不过若是所有的前锋都没有进球的话，那么将返回null。

```
// 没问题，将返回null
var answer = (from p in Forwards
              where p.GoalsScored > 0
              orderby p.GoalsScored
              select p).FirstOrDefault();
// 若序列中没有元素的话，将抛出异常
var answer2 = (from p in Forwards
              where p.GoalsScored > 0
              orderby p.GoalsScored
              select p).First();
```

不过有时我们需要的不是第一个元素，有几种做法可以实现这个需求。例如先对元素进行排序，让你想要的元素变成第一个。（也可以让其变成最后一个，然后获取最后的一个元素，但这可能耗时较长。）

若你明确地知道要得到序列中的第几个元素，那么可以使用Skip和First来获取该指定元素[②]。下面这个查询就返回了排名第三的最佳射手。

```
var answer = (from p in Forwards
              where p.GoalsScored > 0
```

① 而First()将抛出异常。——译者注

② 使用ElementAt()方法可以直接得到序列中指定索引的元素，且只要一个方法调用，让代码更加简单。——译者注

```
orderby p.GoalsScored
select p).Skip(2).First();
```

这里我使用了First()而不是Take()来强调程序仅需要一个元素，而不是一个包含了一个元素的序列。注意，因为这里使用了First()而不是FirstOrDefault()，所以编译器将假设至少有三个前锋有过进球。

不过，当你需要获取序列中某个特定位置的元素时，那么一般情况下都可以通过修改查询本身来实现。还有没有什么别的属性可以使用了呢？你的序列是否实现了IList<T>并支持索引操作呢？能否修改一下算法，让其仅返回那个你想要的元素呢？或许其他的一些做法能让你更简单明了地找到想要的结果。

很多查询的意图就是返回单一的元素。当你仅需要单一元素时，那么最好可以直接返回该元素，而不是包含了该元素的序列。使用Single()意味着你需要一个且仅一个元素。SingleOrDefault()意味着零个或一个元素。First和Last意味着将从序列的头部或尾部得到一个元素。若是你还需要使用其他的方法，那么很有可能是因为你的查询本身还不够好。这样的代码将不利于其他开发者使用，也不利于日后的维护。

条目 44：推荐保存 Expression<>而不是 Func<>

在条目42（本章前面部分）中，曾经简要介绍了例如LINQ to SQL等LINQ提供器是如何在执行查询并将查询转换成原生格式之前进行检查的。而LINQ to Objects则通过将lambda表达式编译成方法，并用委托访问这些方法来实现的查询。这种方式没有太多高深之处，除了通过委托访问之外。

LINQ to SQL（以及其他的查询提供器）是通过System.Linq.Expressions.Expression对象表示的查询表达式来实现其功能的。Expression是一个抽象基类，表示一个表达式。Expression的一个派生类是System.Linq.Expressions.Expression<TDelegate>，其中TDelegate是一个委托类型。Expression<TDelegate>用一种特殊的数据结构来表示一个lambda表达式。

我们可以使用Body、NodeType以及Parameters等参数来对其进行分析。此外，还可以使用Expression<TDelegate>.Compile()方法将其编译成委托。

这就让Expression<TDelegate>要比Func<T>更具有普适性。简单说来，Func<T>只是一个能够被调用的委托。而Expression<TDelegate>则可被察看修改，也可以被编译成委托并调用。

在你需要保存lambda表达式时，那么将其存放为Expression<T>将会更加灵活。你不会因此丢掉任何功能，而只是要在调用之前编译一下而已。

```
Expression<Func<int, bool>> compound = val =>
    (val % 2 == 1) && (val > 300);
Func<int, bool> compiled = compound.Compile();
Console.WriteLine(compiled(501));
```

Expression类提供了多种用来查看表达式中逻辑的方法。通过查看表达式树，即可分析其具体的实现逻辑。C#开发团队提供了查看分析表达式树的一个参考实现，并随Visual Studio 2008一同发布，即Expression Tree Visualizer。该示例包含源代码，能够查看表达式树中每个节点的类型，并显示其内容。该程序将递归地访问每个节点的子节点，这也是我们在访问并修改节点时应该采用的方法。

与函数和委托相比，操作表达式和表达式树要更好一些，因为表达式提供了很多的功能：例如，你可以将Expression转换成Func，可以遍历表达式树（意味着可以在同时创建一个修改后的表达式树）。你甚至可以用Expression在运行时生成新的算法，而这对于Func来说则非常困难。

这样，我们还可以在稍后通过代码将表达式树合并起来。这样的话，你就构造了一棵包含了多个子句的表达式树。随后即可在需要时调用Compile()，得到相应的委托。

下面就是一种将两个表达式组成一个更大表达式的方法。

```
Expression<Func<int, bool>> IsOdd = val => val % 2 == 1;
Expression<Func<int, bool>> IsLargeNumber = val => val > 300;

InvocationExpression callLeft = Expression.Invoke(IsOdd,
Expression.Constant(5));
```

```
InvocationExpression callRight = Expression.Invoke(
    IsLargeNumber,
    Expression.Constant(5));

BinaryExpression Combined =
    Expression.MakeBinary(ExpressionType.And,
    callLeft, callRight);

// 转换成强类型表达式
Expression<Func<bool>> typeCombined =
    Expression.Lambda<Func<bool>>(Combined);

Func<bool> compiled = typeCombined.Compile();
bool answer = compiled();
```

这段代码首先创建了两个小的表达式，并将其组合成一个表达式。随后将该组合后的表达式编译并执行。若你熟悉CodeDom或是Reflection.Emit的话，那么会发现Expression的API提供了类似的元编程（metaprogramming）能力。你可以根据需要访问表达式，创建新表达式，将表达式编译成委托并执行该表达式。

操作表达式树并不容易，因为表达式是不可变的，所以想构造一棵修改后的表达式树需要做很多细致的工作。你要在遍历表达式树的每个节点时，要么将其复制到新树上，要么将现有节点替换成能生成同样结果的新表达式。市面上已经存在着一些作为演示的、开源的访问表达式树的实现，因此这里不必再给出一个版本。若有兴趣的话，可以用“expression tree visitor”作为关键字在因特网上搜索。

System.Linq.Expressions命名空间提供了非常丰富的语法，可以用来在运行时构造算法。使用这些组件即可构造出表达式，进而组成表达式树。下面这段代码的逻辑和前面的例子一样，但却是在代码中构造的lambda表达式。

```
// 该lambda表达式有一个参数
ParameterExpression parm = Expression.Parameter(
    typeof(int), "val");
// 使用几个整数常量
ConstantExpression threeHundred = Expression.Constant(300,
    typeof(int));
ConstantExpression one = Expression.Constant(1, typeof(int));
ConstantExpression two = Expression.Constant(2, typeof(int));
```

```
// 创建 (val > 300)
BinaryExpression largeNumbers =
    Expression.MakeBinary(ExpressionType.GreaterThan,
    parm, threeHundred);

// 创建 (val % 2)
BinaryExpression modulo = Expression.MakeBinary(
    ExpressionType.Modulo,
    parm, two);
// 使用modulo创建((val % 2) == 1)
BinaryExpression isOdd = Expression.MakeBinary(
    ExpressionType.Equal,
    modulo, one);
// 使用isOdd 和largeNumbers创建((val % 2) == 1) && (val > 300),
BinaryExpression lambdaBody =
    Expression.MakeBinary(ExpressionType.AndAlso,
    isOdd, largeNumbers);

// 使用lambda表达式主体和参数创建val => (val % 2 == 1) && (val > 300)
LambdaExpression lambda = Expression.Lambda(lambdaBody, parm);

// 编译
Func<int, bool> compiled = lambda.Compile() as
    Func<int, bool>;
// 执行
Console.WriteLine(compiled(501));
```

确实，使用Expression来构建逻辑当然要比直接用Func<>给出定义要复杂得多。此类元编程技术属于高级主题，并不会在日常工作中经常用到。

即使你自己不需要构建或修改表达式，你所使用的类库也会有可能用到这些功能。因此，若需要将lambda表达式传递给未知类库，那么最好使用Expression<>而不是Func<>，因为这些类库可能会需要使用表达式树来将你的算法转换成另一种形式。LINQ to SQL等IQueryProvider就会执行此类操作。

此外，使用表达式而不是委托也可能会给你带来某些意想不到的方便。这样说的理由同样很简单：你总可以将表达式转换成委托，反之则不行。

你会发现用委托来表示lambda表达式非常简单，从概念上来讲确实如此。委托可以被执行，大多数C#开发人员都理解这个概念，且委托也基本能提供给你所需要的所有功能。不过，若是你的类型使用的是表达式并将表达式提供给不在你控制之下的其他对象，或者若你需要将表达式组合成一些更为复杂的结构，那么则可以考虑用Expression<>来代替Func<>。.NET Framework提供了一系列的API，支持你在运行时根据需要修改表达式，并在修改后执行。

第6章 杂 项

无论怎样组织，总是有一些条目无法归类。不过这些杂项内容同样非常重要，其中包含了你每天都要用到的各种最佳实践。遵循这些建议能够让代码更加易于使用、理解，也便于日后的扩展。

条目45：最小化可空类型的可见范围

与非可空类型相比，可空类型需要更多的检查。若是能够尽可能地使用非可空类型，并将可空类型限制在算法内部使用，那么将让程序的含义更加清晰。可空类型为非可空类型添加了一种丢失或不可用的状态。很多时候，这种可空类型的使用方式有些类似于从前的用某个特别的值来标记出丢失值的做法。

基于可空类型编程要比基于非可空类型复杂很多。可空类型意味着额外的检查：若该值为空，那么应该如何处理呢？答案多种多样，不过对于每个算法来说，空值都应该有确定的处理方式。我们的目标是将对空值的额外检查限制在尽可能小的范围内。这样，客户代码就可以假设你的类库处理了所有的空值情况。

可空类型可以用于需要用到空值数据的算法中。在条目34（第4章）中，曾经给出了一个扩展方法，用来解析字符串并返回可空类型。

```
public static int? DefaultParse(this string input)
{
```

```
    int answer;
    return (int.TryParse(input, out answer))
        ? answer : default(int?);
}
```

若是输入的字符串无法解析成整数，那么该方法将返回一个不包含任何值的可空整数。从该方法的上下文中来看，确实没有什么更好的方法。输入的字符串可能由用户输入、从文件中读取或从未知的源中获取，因此很有可能无法解析。不过最终在某个位置，程序会知道无法解析时应该进行的操作，这时就应该将可空类型替换成正确的默认值。若是没有默认值，且空值意味着失败的话，那么此时则应该抛出异常。这样，我们就将可空类型的使用范围限制在了那些你不知道空值的具体含义的这部分代码中。外部代码不应该看到这些由可空类型引入的额外复杂性。

可空类型为各种常见的值类型操作增加了语义。比如，可空数值类型提供了类似浮点数和NaN之间的语义。所有涉及NaN的比较都将返回false。

```
double d = 0;
Console.WriteLine(d > double.NaN);              // false
Console.WriteLine(d < double.NaN);              // false
Console.WriteLine(double.NaN < double.NaN);     // false
Console.WriteLine(double.NaN == double.NaN);    // false
```

可空类型的行为与其基本一致。与NaN不同的是，可空类型支持空值之间的等同性比较。

此外，涉及空值数据的计算的结果仍旧是空值，就像NaN对浮点数运算的影响一样。

```
// 使用可空类型
int? nullableOne = default(int?);
int? nullableTwo = 0;
int? nullableThree = default(int?);

Console.WriteLine(nullableOne < nullableTwo);       // false
Console.WriteLine(nullableOne > nullableTwo);       // false
```

```
Console.WriteLine(nullableOne == nullableThree);   // true

// 使用NaN
double d = 0;
Console.WriteLine(d + double.NaN);  // NaN
Console.WriteLine(d - double.NaN);  // NaN
Console.WriteLine(d * double.NaN);  // NaN
Console.WriteLine(d / double.NaN);  // NaN
```

涉及可空数值类型的操作和涉及NaN的数值类型操作的行为完全相同。所有涉及空值数值类型的计算结果也是空值。

```
int? nullableOne = default(int?);
int? nullableTwo = default(int);
int? nullableThree = default(int?);

Console.WriteLine((
    nullableOne + nullableTwo).HasValue);   // false
Console.WriteLine(
    (nullableOne - nullableTwo).HasValue);   // false
Console.WriteLine(
    (nullableOne * nullableThree).HasValue); // false
```

这个行为（涉及空值的表达式的结果为空）意味着你会经常需要为这些操作定义默认值。这可以通过Nullable<T>.GetValueOrDefault()实现，不过C#语言提供了一种语法上的便利，即空值合并运算符（null coalescing operator，??）。若该可空类型中包含值，那么该运算符将返回其值，否则将返回操作符右侧的值。如下两行代码所执行的操作相同。

```
var result1 = NullableObject ?? 0;
var result2 = NullableObject.GetValueOrDefault(0);
```

可以将该操作符应用于任何使用到可空类型的表达式中。

```
int? nullableOne = default(int?);
int? nullableTwo = default(int);
int? nullableThree = default(int?);

int answer1 = (nullableOne ?? 0) + (nullableTwo ?? 0);
Console.WriteLine(answer1); // output is 0
```

```
int answer2 = (nullableOne ?? 0) - (nullableTwo ?? 0);
Console.WriteLine(answer2); // output is 0

int answer3 = (nullableOne ?? 1) * (nullableThree ?? 1);
Console.WriteLine(answer3); // output is 1
```

若是代码中充斥着可空类型，那么还会带来另外一些问题。例如，序列化可空类型非常危险，如下代码片断：

```
int? f = default(int?);

XmlSerializer x = new XmlSerializer(typeof(int?));
StringWriter t = new StringWriter();
x.Serialize(t, f);
Console.WriteLine(t.ToString());
```

将生成如下XML文档：

```
<int xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xsi:nil="true" />
```

注意，上述XML元素的类型是int，而该元素却没有包含任何有关该int可能为空的信息。可以看到，该信息就这样丢失了。

这样将导致如下看上去正确的代码抛出NullReferenceException异常。

```
string storage = t.ToString();
StringReader s = new StringReader(storage);
int f2 = (int)x.Deserialize(s); // f2不能为null
Console.WriteLine(f2);
```

出现了该行为的原因非常简单。首先保存了一个可空的整数，该整数可能是任何一个整数值以及空值。不过在将其重新读取回来的时候，却隐式地将int?转换成了int。这样，空值就变得不合法了，于是代码自然会抛出异常。

在需要序列化XML文档的时候，包含了可空数值的类型也会产生同样的问题。这些成员元素将被序列化成值类型，且其值为空。而根据XML文档构造对象的工具则会构造出int，而不是可空int。在空值的情况下，使用这些

工具凡序列化此类对象将导致同样的问题。

在使用System.Object中定义的虚方法，或是使用底层值类型实现的接口时，可空类型同样会带来更加复杂的行为。这个操作将导致装箱，将内部的值类型转换为System.Object或接口的指针。考虑下面这两个转换操作，将可空int转换成字符串。

```
int? defaultNullable = default(int?);
string s = defaultNullable.ToString();
Console.WriteLine(s);
string s2 = ((object)defaultNullable).ToString();
Console.WriteLine(s2);
```

defaultNullable.ToString()将生成空字符串""作为其字符串表示。这没有什么问题：若可空int中不包含值，那么自然也没有其相应的字符串表示形式。不过，第二个转换则会抛出NullReferenceException异常。这是因为该转换会尝试将可空类型中包含的值类型装箱。而该可空类型却恰好为空，因此该转型将返回null。虽然这确实是正确的行为，不过它需要一些时间来理解。其行为和前面的调用不同，因为前面的调用了<int>.ToString()。而第二个则装箱了可空类型包含的值，其结果为null，随后在空对象上调用了object.ToString()。

每个可空类型都可以被隐式地转换成其包含的非可空类型，因此同样的问题将经常出现。例如，你可以在两个可空数值类型上使用IComparable<T>接口。

```
int? one = 1;
int? two = 2;

IComparable<int> oneIC = one;
if (oneIC.CompareTo(two.Value) > 0)
    Console.WriteLine("Greater");
else
    Console.WriteLine("Not Greater");
```

不过这里将涉及一次装箱操作，因此若将包含了空值的可空类型转换成非空类型，并调用该类型上的接口，那么将得到NullReferenceException

异常。

```
int? empty = default(int?);

IComparable<int> emptyIC = empty;
// 抛出NullReferenceException异常
if (emptyIC.CompareTo(two.Value) > 0)
    Console.WriteLine("Greater");
else
    Console.WriteLine("Not Greater");
```

这非常合理。该空的可空类型并没有包含值，因此基于该值进行的操作自然会报错。

上面所有演示可空类型的例子都使用的是数值类型。这并不是因为数值类型有什么特别之处，而是因为其语义上比较简单，且与其他结构相比，数值类型的操作更加简单。不过，在使用可空的结构时，上述各个情况也会发生。当然，结构上并没有自动定义算术操作符，但其他方法却依然存在。访问为空的可空结构的值将导致NullReferenceException异常。这个转换是隐式的，因此编译器不会提示你其中可能出现的危险。

在设计API时，应该尽力保证其易于使用，且不容易被误用。可空类型将让情况变得更加复杂。若能够避免在公共接口或存储模型中公开可空类型，那么将让组件更易于使用。这并不是说可空类型不能出现在你的API中。本条目的第一个例子就演示了一个返回nullable<int>的方法，用来将输入字符串转换成整数。若是可空类型确实表示了你正在操作的类型，那么自然应该使用。在组件内部，可空的结构则通常很适用于算法中。不管怎样，最小化可空对象的作用范围将让你的类型更易于使用，且不易于被误用。

这样，在使用可空类型时，应该小心检查在可空对象和非空对象之间的转换操作。若你将这些可空类型作为公共接口公开出去，那么也就是在要求你的用户像你一样理解这些规则。这也许确实是正确的行为，不过你应该让其成为显式的选择，而不应该是后台隐式的操作。

条目 46：为部分类的构造函数、修改方法以及事件处理程序提供部分方法

C#语言的开发团队之所以添加了部分类（partial class）特性，是为了让代码生成器能够生成类型的一部分，然后开发者可以在在另一个代码文件中为同一个类添加各种自定义逻辑。但不幸的是，这种分离并不能全面满足复杂情况下的各种应用模式。很常见的情况是，开发人员需要在代码生成器生成的成员中添加代码。这些成员包括构造函数、生成代码中定义的事件处理程序，以及生成代码中定义的修改方法（mutator method）等。

我们的目的是让使用代码生成器的开发者无需修改这些自动生成的代码。另一方面，若你在使用由工具创建的代码，那么则绝不应该修改这些生成的代码。因为这样做将破坏掉与代码生成器之间的关系，也会让稍后的继续开发变得更加困难。

在某些情况下，编写部分类有些类似于设计API。作为开发者或者作为代码生成工具的作者，我们所编写的代码将会被另一方使用（要么是开发者，要么是代码生成工具）。这就像是两个开发者共同编写着一个类，不过其约束却非常严格。两个开发者不能彼此沟通，且均无法修改对方的代码。这就意味着你需要在代码中为对方提供很多挂钩。这些挂钩应该以部分方法的形式实现。另一方的开发者既可以选择实现部分方法提供的挂钩，也可以选择不实现。

代码生成器将为这些扩展点定义部分方法。部分方法允许开发人员在另外一个代码文件中的部分类中给出方法的实现。编译器将查看完整的类定义，随后，如果定义了部分方法，那么编译器将生成对这些方法的调用。若是没有实现某些部分方法，那么编译器将把对其的调用移除。

因为部分方法可能成为类的一部分，也可能不是类的一部分，所以语言本身对部分方法的签名给出了一定的限制：部分方法的返回值必须为void，部分方法不能为抽象或虚方法，且部分方法也不能用来实现接口方法。其参数列表也不能包含任何out参数，因为编译器无法初始化这些out参数。若是方法没

有实现的话，自然也不能给出返回值。此外，部分方法还应该是私有的。

对于三种类型成员，你应该添加部分方法来让用户监视或修改类的行为：修改方法、事件处理程序以及构造函数。

*修改方法*是指那些将要修改类中对外可见的状态的方法。从部分方法以及部分类的角度来看，我们可以将其理解为任何状态的变化。另外一个代码文件中的部分类实现依然是类的一部分，因此自然应该能够完全控制到类的所有内部状态。

修改方法中应该为类的另一个实现方提供两个部分方法：第一个方法应该在修改之前调用，用来让另一方能够进行合法性验证甚至拒绝本次修改。第二个方法应该在修改之后调用，从而让另一方可以响应这个状态的变化。

例如，工具生成的核心代码类似如下形式。

```
// 这是代码生成器生成的代码
public partial class GeneratedStuff
{
    private int storage = 0;

    public void UpdateValue(int newValue)
    {
        storage = newValue;
    }
}
```

你应该在变化之前以及之后添加挂钩。这样即可让另一方修改或响应该变化。

```
// 这是代码生成器生成的代码
public partial class GeneratedStuff
{
    private struct ReportChange
    {
        public readonly int OldValue;
        public readonly int NewValue;

         public ReportChange(int oldValue, int newValue)
        {
            OldValue = oldValue;
```

```
            NewValue = newValue;
        }
    }

    private class RequestChange
    {
        public ReportChange Values
        {
            get;
            set;
        }
        public bool Cancel
        {
            get;
            set;
        }
    }

    partial void ReportValueChanging(RequestChange args);
    partial void ReportValueChanged(ReportChange values);

    private int storage = 0;

    public void UpdateValue(int newValue)
    {
        // 预先检查修改
        RequestChange updateArgs = new RequestChange
        {
            Values = new ReportChange(storage, newValue)
        };
        ReportValueChanging(updateArgs);
        if (!updateArgs.Cancel) // 如果可以修改
        {
            storage = newValue; // 进行修改
            // 报告修改
            ReportValueChanged(new ReportChange(
                storage, newValue));
        }
    }
}
```

如果没有提供两个部分方法的话，那么UpdateValue()编译后将按照如下代码编译。

```
public void UpdateValue(int newValue)
{
    RequestChange updateArgs = new RequestChange {
        Values = new ReportChange(this.storage, newValue)
    };
    if (!updateArgs.Cancel)
    {
        this.storage = newValue;
    }
}
```

这个挂钩允许开发者验证或响应任何变化。

```
// 这是手工编写的代码
public partial class GeneratedStuff
{
    partial void ReportValueChanging(
        GeneratedStuff.RequestChange args)
    {
        if (args.Values.NewValue < 0)
        {
            Console.WriteLine("Invalid value: {0}, canceling",
                args.Values.NewValue);
            args.Cancel = true;
        }
        else
            Console.WriteLine("Changing {0} to {1}",
                args.Values.OldValue,
                args.Values.NewValue);

    }
    partial void ReportValueChanged(
        GeneratedStuff.ReportChange values)
    {
        Console.WriteLine("Changed {0} to {1}",
            values.OldValue, values.NewValue);
    }
}
```

这里，我们通过一个取消标记来让开发者可以取消某个修改操作。你也可以通过抛出异常来取消操作。若取消操作将由调用者执行，那么抛出异常将会更好一些。否则推荐使用布尔型的取消标记，因为这样更加轻量一些。

此外，注意到哪怕没有调用ReportValueChanged()，程序仍旧创建了RequestChange对象。RequestChange对象的构造函数中可能有任意的逻辑，因此编译器无法在不改变UpdateValue()语义的情况下移除该构造函数调用。为了能够让另一方能够创建额外对象来验证并响应变化，这是必须要付出的代价。

找到类中所有的公共修改方法非常简单，不过要记住属性的公共set访问器也属于修改方法。若没有提供相应的挂钩的话，那么另一方将无法验证或响应属性的变化。

接下来则需要在构造函数中为用户代码提供挂钩。生成代码和用户提供的代码都无法控制外部将会调用哪个构造函数。因此，代码生成器必须能够在外部调用生成的构造函数时调用用户自定义的逻辑。下面就是对前面的GeneratedStuff类型的扩展。

```
// 为用户自定义逻辑提供挂钩
partial void Initialize();

public GeneratedStuff() :
    this(0)
{
}

public GeneratedStuff(int someValue)
{
    this.storage = someValue;
    Initialize();
}
```

注意到Initialize()方法将在构造的最后一步调用。这就让手工代码有机会检查当前对象的状态，进行必要的修改，或是在不符合某些要求时抛出异常。需要保证的是你不会调用Initialize()两次，且必须在生成代码的每个

构造函数中都调用到该Initialize()方法。而手工代码中的构造函数则决不应该调用它自己的Initialize()。而是应该显示调用生成代码中的某个构造函数，这样才能保证生成代码中的初始化工作也能顺利完成。

最后，若是生成代码订阅了某个事件，那么在处理事件时也应该以扩展方法的形式提供挂钩。若是该事件还包含了状态或取消标记的话，那么这一点将更加重要。因为手工代码也需要能够修改事件的状态或是取消标记。

部分类和部分方法提供了一种将同一个类中的生成代码和手工代码分开的机制。借助于本条目中给出的这种扩展，你可以做到不对生成代码进行任何修改。或许你已经用到了Visual Studio或其他工具生成的代码。在开始着手修改这些生成代码之前，首先你要检查一下生成代码中是否提供了部分方法定义，从而让你在自己的代码文件中完成所需的工作。更加重要的是，若你是代码生成器的开发者，那么必须要以扩展方法的形式提供完整的一系列扩展点，以便支持客户代码的扩展逻辑。若是没有提供足够的扩展，那么结果将难以预料，使用者可能不得不放弃你的代码生成器。

条目47：仅在需要parms数组时才使用数组作为参数

使用数组参数将有可能让代码遇到一些意料之外的问题。更好的方式是为方法传递集合参数或可变个数的参数。

数组的一些特殊的性质让我们可以编写出看上去有着严格类型检查，不过却会在运行时失败的方法。你可以很容易地绕过所有的编译期类型检查。下面的代码就会在为ReplaceIndices方法中的parms参数的第一个元素赋值时抛出ArrayTypeMismatchException异常。

```
static void Main(string[] args)
{
    string[] labels = new string[] { "one", "two",
        "three", "four", "five" };
```

```
        ReplaceIndices(labels);
    }

    static private void ReplaceIndices(object[] parms)
    {
        for (int index = 0; index < parms.Length; index++)
            parms[index] = index;
    }
```

之所以出现这样的问题，是因为数组是作为输入参数传入的。你无需将数组的精确类型传入到方法中。此外，虽然数组是通过传值的方式传入的，不过其内容却可以指向引用类型。在方法中，某些对数组元素的修改可能会让其类型不能与原数组的元素类型匹配。当然，上面的这个例子非常明显，你会认为你永远不会写出这样的代码。不过继续看看接下来的这几个类继承关系。

```
class B
{
    public static B Factory()
    {
        return new B();
    }

    public virtual void WriteType()
    {
        Console.WriteLine("B");
    }
}

class D1 : B
{
    public static new B Factory()
    {
        return new D1();
    }

    public override void WriteType()
    {
        Console.WriteLine("D1");
    }
}
```

```
class D2 : B
{
    public static new B Factory()
    {
        return new D2();
    }

    public override void WriteType()
    {
        Console.WriteLine("D2");
    }
}
```

若能正确使用的话，那么不会出现什么问题。

```
static private void FillArray(B[] array, Func<B> generator)
{
    for (int i = 0; i < array.Length; i++)
        array[i] = generator();
}

// 其他位置
B[] storage = new B[10];
FillArray(storage, () => B.Factory());
FillArray(storage, () => D1.Factory());
FillArray(storage, () => D2.Factory());
```

若是与派生类型之间存在着一些不匹配，那么同样将抛出ArrayTypeMismatchException异常。

```
B[] storage = new D1[10];
// All three calls will throw exceptions:
FillArray(storage, () => B.Factory());
FillArray(storage, () => D1.Factory());
FillArray(storage, () => D2.Factory());
```

此外，因为数组不支持逆变（contravariance），因此在写入数组元素时，即使看上去合情合理，仍旧会得到编译错误。

```
static void FillArray(D1[] array)
{
    for (int i = 0; i < array.Length; i++)
        array[i] = new D1();
```

```
}
B[] storage = new B[10];
// 产生编译错误 CS1503 (argument mismatch)
// 即使D对象能够放在B数组中
FillArray(storage);
```

若是作为ref参数传递数组的话，那么情况还会更加复杂。在方法中你可以创建派生类，但却不能创建基类。不过数组中的元素仍旧是错误的类型。

将参数标记为接口类型，并使用类型安全的序列即可避免此类问题。对于类型T，输入参数应该为IEnumerable<T>类型。这个策略即可保证方法中无法修改输入序列，因为IEnumerable<T>并没有提供修改集合的方法。另外一种方法是以基类的形式传入类型，这同样可以避免让API修改集合。

若你确实需要修改集合，那么最好不要修改输入参数，而是返回一个新的修改过的序列（参见第3章条目17）。在生成序列时，可以为类型T返回IEnumerable<T>。

在另一些情况中，你需要为方法传入可变个数的选项。这时就应该使用数组参数，不过应该为其添加params关键词。params数组允许方法的使用者为方法传入可变个数的参数。比较下面这两个方法：

```
// 普通数组
private static void WriteOutput1(object[] stuffToWrite)
{
    foreach (object o in stuffToWrite)
        Console.WriteLine(o);
}
// Params数组
private static void WriteOutput2(params object[]
    stuffToWrite)
{
    foreach (object o in stuffToWrite)
        Console.WriteLine(o);
}
```

可以看到，在编写方法以及方法中检查数组中各元素的过程中，二者没有什么不同。不过调用这两个方法的语法却各不相同。

```
WriteOutput1(new string[]
    { "one", "two", "three", "four", "five" });
WriteOutput2("one", "two", "three", "four", "five");
```

若是调用者不想指定任何一个可选的参数，那么就会看出区别。params数组版本可以在无参数时调用。

```
WriteOutput2();
```

而使用普通数组的方法则无法通过编译。

```
WriteOutput1(); // 无法通过编译
```

即使提供null，也会抛出异常。

```
WriteOutput1(null); // 抛出空参数异常
```

必须这样丑陋地调用：

```
WriteOutput1(new object[] { });
```

不过params数组的做法仍称不上完美，因为即使params数组也会遇到类型不协调的问题。不过这个问题一般不大会发生。首先，编译器将生成存放该参数数组的空间，并传入到你的方法中，因此修改一个由编译器生成的数组并不会有什么意义。调用方法也不会得到返回结果。此外，编译器将自动为数组选择正确的类型。若想出现这种异常，方法调用者的代码必须相当不合常理——首先要创建一个真正的数组，其类型与将调用的方法参数的类型不同。然后还要将其作为params数组来调用方法。虽然这也是可能的，不过为了避免这个问题，系统本身已经做了足够多的努力。

并不是说数组永远不应该用于方法参数中，不过这样做却可能造成两种类型的错误。数组类型的不明确可能造成运行时错误，且数组的引用特性将允许被调用者中可以替换调用者的对象。即使你的方法保证不会出现此类问题，但方法的签名却暗示了这个问题出现的可能性。使用你的方法的开发者会产生疑问：调用该方法安全吗？是不是应该创建一个临时的副本再调用？在使用数组

作为方法参数时，你基本上总能找到更好的方法。若参数表示一个序列，那么使用IEnumerable<T>或IEnumerable<T>的某个合适的实例。若参数表示的是可变的集合，那么修改方法签名，让其接受输入序列，修改后再创建输出序列。若参数表示的一系列选项，那么使用params数组。对于任何一种情况，我们总是可以找到一种更好且更安全的接口。

条目 48：避免在构造函数中调用虚方法

在构造对象的过程中，虚方法的行为会显得有些奇怪。在构造函数执行完毕之前，对象并没有被完整创造出来。同时，虚方法此时的表现也不一定会如你所愿。例如如下的这个简单的程序。

```
class B
{
    protected B()
    {
        VFunc();
    }

    protected virtual void VFunc()
    {
        Console.WriteLine("VFunc in B");
    }
}

class Derived : B
{
    private readonly string msg = "Set by initializer";

    public Derived(string msg)
    {
        this.msg = msg;
    }

    protected override void VFunc()
    {
        Console.WriteLine(msg);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Derived d = new Derived("Constructed in main");
    }
}
```

将输出那一句话呢？“Constructed in main”、“VFunc in B”还是“Set by initializer”？有C++开发经验的开发者会说是“VFunc in B”。有些C#开发者会说是“Constructed in main”。不过正确的答案却是“Set by initialize”。

基类构造函数调用了一个虚方法，这个虚方法定义在了基类中，但却在派生类中被重写。在运行时，派生类中的方法将被调用，且对象的运行时类型为Derived。C#语言会认为派生对象已经完全可用，因为在进入到任何一个构造函数中时，所有的成员变量都被初始化过了。毕竟，所有的变量初始化器都已被执行。代码中已经有过机会初始化所有的变量，不过这并不代表所有的变量都被初始化成了你所期待的值。这是因为仅仅变量初始化器得以执行过，而派生类构造函数还没有机会进行其初始化工作。

不管怎样，在构造对象是调用虚方法将会导致某些不一致性的情况发生。C++语言设计者在设计该功能时，会调用将要构造的对象的运行时类型中的虚方法。在对象创建时，就确定下来其运行时类型。

这样做确有其道理。首先，将要创建的对象的类型为Derived，因此所有的方法都应该调用Derived中重写过的版本。在这一方面，C++并不一致：对象的运行时类型会根据当前正在执行的构造函数而发生变化。其次，这样设计的语言特性避免了一个问题，即在当前类型为抽象基类时，虚方法实现中可以避免使用空方法指针，看看下面这个可变基类。

```
abstract class B
{
  protected B()
  {
    VFunc();
  }
```

```
    protected abstract void VFunc();
}

class Derived : B
{
    private readonly string msg = "Set by initializer";

    public Derived( string msg )
    {
        this.msg = msg;
    }

    protected override void VFunc()
    {
        Console.WriteLine( msg );
    }

    static void Main()
    {
        Derived d = new Derived( "Constructed in main" );
    }

}
```

这段程序可以通过编译，因为B对象还没有被创建，所以派生对象必须提供VFunc()的实现。之所以C#的策略是调用与运行时类型匹配的VFunc()版本，是因为这是避免在构造函数中调用抽象方法时抛出运行时异常的唯一办法。有经验的C++开发者会预料到使用同样构造方式可能出现的运行时异常。在C++中，在B构造函数中调用VFunc()将会让程序崩溃。

不过，这个简单的例子还是展示了C#这种策略的一个陷阱。msg变量是不可变的，在对象生命周期中应该有着同样的值。不过在构造函数还没完成其工作的这一小段时间之内，该变量可能会有不同的值：一个由初始化器设置，另一个则由构造函数中给出。在一般情况下，派生类的所有变量都应该在其默认状态中，或者由初始化器设置，或者是系统默认值。这些变量当然不会有你所期待的值，因为派生类的构造函数还没有执行。

仅在最严格的条件下，构造函数中调用虚方法才不会出现问题，即派生类

必须在变量初始化器中初始化过所有的变量。满足这个规则的对象很少见：大多数构造函数都会接受一些参数，用来设定对象的内部状态。因此可以这样说，在构造函数中调用虚方法则必须让所有的派生类都定义一个默认的构造函数，且还不能定义其他的构造函数。这是一个相当严格的限制。你真的希望所有使用你的基类的开发者都遵守这个规则吗？我不这样认为，因为这样实在不会带来什么好处，却引出了很多不便。实际上，这种情况是如此地少见，以至于和Visual Studio配合使用的FxCop和Static Code Analyzer工具都会给出相应的提示。

条目49：考虑为大型对象使用弱引用

无论怎样尽力，我们总是会使用到某些需要大量内存的对象。有些时候，这些内存并不需要经常访问。或许你需要从一个大文件中查找某个特定值，或者算法需要一个较大的查询表。这时，你也许会采取两种不太好的做法。第一种是创建一个本地变量，然后在每次执行该算法是都生成这一大块垃圾。第二种则是创建一个成员变量，在很长一段时间内均占用着这一大块内存。很多时候这两种做法都不是非常妥当。

好在我们还有一个选择：创建弱引用。弱引用的对象和垃圾对象差不多。程序会告诉垃圾收集器该对象可以被回收，不过在回收之前你仍旧有一个引用，可以用来在需要的时候访问到该对象。若能合理使用该策略的话，弱引用将帮助你和垃圾收集器协同工作而不是针锋相对，从而共同优化内存使用。

假设你的某个算法偶尔会需要到一个1 000×1 000的数组。那么你会创建一个类来存放该数组。

```
class MyLargeClass
{
    private int[,] matrix = new int[1000, 1000];
    private int matrixXDimension;
    private int matrixYDimension;

    // 省略
}
```

在任何环境中，这个一百万个元素的数组的代价都非常高昂。在进行分析之后，你发现只是偶尔会使用到该数组，而且分配并释放这个偶尔才会使用到的MyLargeClass对象耗费了很多程序时间。此时就应该考虑创建一个弱引用来指向单一的一个MyLargeClass对象，并在需要时随时重用。

使用弱引用非常简单。只需创建一个WeakReference对象，传入期待的对象，随后即可将强引用设置为null。

```
WeakReference w = new WeakReference (myLargeObject);
myLargeObject = null;
```

此时，系统将会认为myLargeObject所指向的对象可以被垃圾收集。若垃圾收集器在此时运行，那么将会回收该对象。不过若是在垃圾收集器运行之前你再次需要该对象，那么只需从该WeakReference对象中重新获取即可。

```
myLargeObject = w.Target as MyLargeClass;
if (myLargeObject == null)
    myLargeObject = new MyLargeClass();
```

若是对象已经被回收，那么Target属性将返回null。此时你将不得不重新创建该对象。这样就将很多优化的工作交给了运行时来处理。在对象仍旧存在时，可以容易地重用，同时也让系统能够在需要时回收这块内存。若将弱引用重新变成强引用，那么其内容以及状态仍会保持不变。不过这并不是使用弱引用的原因。如果创建对象的状态比较昂贵，那么应该使用强引用。WeakReference是为了那些内存占用较大的对象而设立的。

这是最简单的场景，不过很多时候现实并不是那么简单。很少有对象会完全与外界隔离。任何一个大型的对象都毫无疑问地包含了对其他对象的引用。垃圾收集器在处理弱引用对象中包含的对象时也会非常小心。例如MyLargeClass还有其他的一些成员变量。

```
class MyLargeClass
{
    private int[,] matrix;
    private string reallyLongMessage;
```

```
        private int matrixXDimension;
        private int matrixYDimension;

        // 省略
    }
```

MyLargeClass包含了一个数组的值类型、一个引用类型和两个值类型。若MyLargeClass变成了垃圾，那么数组和字符串都会成为垃圾收集器回收的目标，这当然没有什么问题。不过若MyLargeClass对象存在弱引用的话，那么上述代码还需要一定的修改。若是垃圾收集器将数组和字符串释放的话，那么很有可能会导致问题。实际上，垃圾收集器确保了这里不会出现任何问题，但这需要一些额外的工作。在接下来的介绍中，我极大地简化了垃圾收集器的实际处理过程，不过也足够理解。毕竟我们不是要编写垃圾收集器，而仅仅是使用。

在为对象创建弱引用时，实际上创建的是一棵弱引用树：目标对象的每一个可达对象都被标记为弱引用。垃圾收集器在标记正在使用的对象时还有一个额外的步骤：在标记所有的强引用对象之后，将会标记所有的弱引用对象。随后执行清理工作。若垃圾收集器发现没有足够的内存，那么垃圾收集器将会回收那些仅通过弱引用可达的对象。

从我们的角度看来，在添加弱引用时发生了两件事情。首先，垃圾收集器将找到所有占用的内存。随后，这些由弱引用指向的对象将被当成是垃圾中的可被再利用部分。与那些完全不可达的垃圾相比，这些对象被回收的几率更小一些。结果就是，弱引用对象的状态介于完全可用和完全不可用之间。

对于实现了IDisposable接口的弱引用以及对象来说，还有一些其他的考虑。此类对象并不适合作为弱引用，因为你不知道何时调用这些对象上的Dispose()方法。对于一个已经被释放的对象来说，其重用并不容易，因为其中重要的资源已经被释放了。此外，你也不能在弱引用上调用Dispose()。简而言之，你不知道何时应该在弱引用对象上调用Dispose()方法。

不过等一下，还要考虑终结器（finalizer）。终结器会带来更多的问题，它会引入“长”弱引用和“短”弱引用的概念。短弱引用（short weak reference）将在其目标对象不存在时立即返回null。此时表示该对象要么已被回收，要么已被终结。而长弱引用（long weak reference）将在其目标对象仍在内存中时依旧返回该对象。即使对象已被终结，其仍旧存在于内存中。这样，长弱引用对象有可能会返回一个已经被终结过的对象指针。对于这个已被终结过的对象，能够进行的操作少得可怜，因此我从未在实际中用过长弱引用对象。

在实际应用中，弱引用能帮你管理应用程序中的那些大型且仅会偶尔使用的对象。弱引用能够很好地配合非IDisposable对象使用。按照定义，这些类型不包含终结器。有了这些约束，弱引用即可让垃圾收集器很好地管理内存需要。不过若是在其他情况中使用，那么则必须非常小心。

最后必须给出一些值得注意的警告：弱引用可能会提高算法的效率，不过也会对垃圾收集器带来不小的性能影响。考虑到这些影响，在程序中使用弱引用也许会让程序变得更慢。因此在考虑使用弱引用之前，先尝试优化一下算法，让其尽可能少地占用内存（参见第5章条目37）。仅在这些方法都无济于事时再考虑使用弱引用。随后应该对使用弱引用之前和之后的程序性能进行测试，并仔细比较其中的差别。

条目 50：使用隐式属性表示可变但不可序列化的数据

属性的语法让你能够更清晰地表达出设计的含义。从C# 2.0开始，我们即可为属性的getter和setter指定不同的访问权限。在C# 3.0中，还可以使用隐式属性，它同样可以和属性访问修饰符良好配合。

在为类添加可访问的数据时，属性访问器通常只是简单地包装了类的数据字段。这时即可使用隐式属性来提高代码的可读性。

```
public string Name
{
    get;
    set;
}
```

编译器将自动生成属性封装的支持字段，并指派一个名称。你也可以使用属性的setter来修改该字段的值。因为支持字段的名称是编译器生成的，因此即使在类型的内部也要通过属性访问器来修改，而不能直接设置。这并没有什么问题，调用属性访问器也能实现同样的功能，且因为编译器生成的访问器中仅包含一条赋值语句，因此该访问器通常会被内联。从性能角度考虑，访问隐式属性和直接访问其支持字段的运行时行为完全一致。

与显式属性一样，隐式属性也支持为getter和setter指定不同的访问权限。你可以根据需要为getter或setter给出更严格的访问限制。

```
public string Name
{
    get;
    protected set;
}
// 或者
public string Name
{
    get;
    internal set;
}
// 或者
public string Name
{
    get;
    protected internal set;
}
// 或者
public string Name
{
    get;
    private set;
}
```

隐式属性将会依然遵循与前面版本中普通属性（带有支持字段的属性）一样的模式。其优势在于提高了生产率，也让代码更易于阅读。隐式属性声明让人能够很清楚地看出代码的意图，且也不会生成过多臃肿的代码，以至将代码的核心逻辑淹没其中。

因为隐式属性将生成和显式属性同样的代码，所以我们当然也可以用这种语法来定义虚属性、重写虚属性或实现定义在接口中的属性。

在创建隐式虚属性时，派生类无法访问到编译器生成的字段。不过，重写属性中仍可以使用base属性来访问到基类中的getter和setter，就像在虚方法中的其他操作一样。

```
public class BaseType
{
    public virtual string Name
    {
        get;
        protected set;
    }
}

public class DerivedType : BaseType
{
    public override string Name
    {
        get { return base.Name; }
        protected set
        {
            if (!string.IsNullOrEmpty(value))
                base.Name = value;
        }
    }
}
```

使用隐式属性将带来两个额外的好处。在日后需要将隐式属性替换成专门的显式实现时（可能因为需要验证数据或其他原因），类对外仍旧是二进制兼容的，且验证逻辑也会仅出现在一处。

在早先版本的C#语言中，大多数开发者在类内部都将直接访问其中的支持字段。这样的代码让验证以及错误检查逻辑分散在了各处。而每次修改隐式属性封装的支持字段时，都会调用该属性的setter访问器（可能是私有的）。随后在需要时，可以容易地将隐式属性访问器转换为显式的，并在其中添加一切必要的验证逻辑。

```
// 原始版本
public class Person
{
    public string FirstName
    {
        get;
        set;
    }
    public string LastName
    {
        get;
        set;
    }
    public override string ToString()
    {
        return string.Format("{0} {1}", FirstName, LastName);
    }
}

// 后续的更新版本，添加了验证逻辑
public class Person
{
    private string firstName;
    public string FirstName
    {
        get
        {
            return firstName;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "First name cannot be null or empty");
```

```
            firstName = value;
        }
    }
    private string lastName;
    public string LastName
    {
        get
        {
            return lastName;
        }
        private set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Last name cannot be null or empty");
            lastName = value;
        }
    }
    public override string ToString()
    {
        return string.Format("{0} {1}", FirstName, LastName);
    }
}
```

这样就将所有的验证操作都放在了统一的位置。由于我们均是通过属性访问器来访问支持字段，因此每次访问都会得到必要的验证。

上述介绍也许会让你觉得隐式属性在各方面都是最好的选择，不过在创建不可变类型时，它存在着一定的限制。因为即使在构造函数中，也必须使用属性访问器来设置属性的值。这样就要让其支持字段在任何时候都支持被修改。属性访问器无法分辨出在构造过程中设置以及在稍后使用时设置的区别。这就意味着支持字段并没有带有initonly标记[①]。此外，你还可以编写任意的修改方法（但要受到访问修饰符的限制），并用其修改你的类型。考虑到这些，在创建不可变类型时，应该使用显式成员变量。

① C++/CLI的关键词。静态的initonly字段只可以被静态构造函数赋值，非静态的initonly字段只可以被实例的构造函数赋值。它类似于C#中的readonly。——译者注

创建真正不可变类型的唯一办法是手工编写属性及其支持字段。在JIT层面上，隐式属性并不支持真正的不可变类型。从客户代码角度来看，某个类型可能是不可变的，不过运行时却无法保证隐式属性支持字段在构造后不发生变化。

隐式属性还有一个重要的限制。在应用了Serializable属性的类型中不能使用隐式属性。因为持久化存储的格式需要依赖于编译器生成的字段的名称，而该字段的名称则无法保证永远不变。任何对该类的修改都有可能会改变这个自动生成的字段的名称。

除了这两个限制之外，隐式属性节省了开发者的时间，增强了代码的可读性，其实现特性也非常有益于引导开发者将所有的字段修改验证放在统一的位置。清晰的代码自然更有助于日后的维护。

索　引

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y