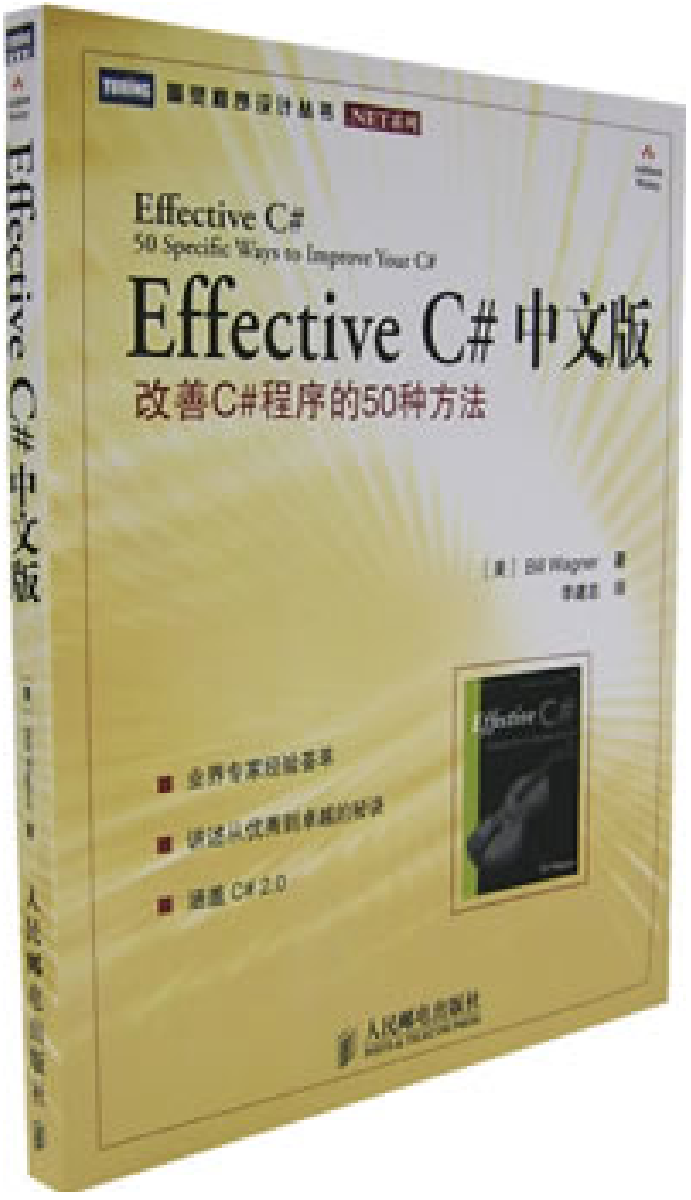
Effective C# 中文版 改善C#程序的50种方法



内容提要

本书围绕一些关于 C#和.NET 的重要主题，包括 C#语言元素、.NET 资源管理、使用 C#表达设计、创建二进制组件和使用框架等，讲述了最常见的 50 个问题的解决方案，为程序员提供了改善 C#和.NET 程序的方法。本书通过将每个条款构建在之前的条款之上，并合理地利用之前的条款，来让读者最大限度地学习书中的内容，为其在不同情况下使用最佳构造提供指导。

本书适合各层次的 C#程序员阅读，同时可以推荐给高校教师（尤其是软件学院教授 C#/.NET 课程的老师），作为 C#双语教学的参考书。

作者简介

Bill wagner 是世界知名的. NET 专家，微软 C#领域的 MVP，并荣获微软 Regional Director 称号。他是著名软件咨询公司 SRT Solutions 的创始人，有 20 多年软件开发经验，曾经领导了众多成功的 Windows 平台产品的开发。他是微软开发社区的活跃人物，长期担任 MSDN Magazine、ASP. NET Pro、Visual Studio Magazine 等技术杂志的专栏作者。他的 blog 是 http://www.srtsolutions.com/public/blog/20574，可以通过 wwagner@SR7Solutions.com 与他联系。

编辑推荐

业界专家经验荟萃，讲述从优秀到卓越的秘诀，涵盖 C#2.0。

“一直以来，读者们总在不停地问我，什么时候写 Effective C#？本书的出版使我如释重负。令人高兴的是，我本人已经从阅读 Bill 的著作中获益良多，相信读者也会和我一样。”

——Scott Meyers，Effective C++作者，世界级面向对象技术专家

C#与 C++、Java 等语言的相似性大大降低了学习难度。但是，C#所具有的大量独特的特性和实现细节。有时又会使程序员适得其反：他们往往根据既有经验，错误地选用了不恰当的技术。从而导致各种问题。与此同时，随着数年来 C#的广泛应用，业界在充分利用 C#的强大功能编写快速、高效和可靠的程序方面也积累了丰富的最佳实践。

本书秉承了 Scott Meyers 的 Effective C++和 Joshua Bloch 的 Effective Java 所开创的伟大传统. 用真实的代码示例，通过清晰、贴近实际和简明的阐述，以条款格式为广大程序员提供凝聚了业界经验结晶的专家建议。

本书中，著名.NET 专家 Bill Wagner 就如何高效地使用 C#语言和.NET 库。围绕 C#语言元素、.NET 资源管理、使用 C#表达设计、创建二进制组件和使用框架等重要主题，讲述了如何在不同情况下使用最佳的语言构造和惯用法，同时避免常见的性能和可靠性问题。其中许多建议读者都可以举一反三。立即应用到自己的日常编程工作中去。

前言

本书就如何高效使用 C#语言和.NET 库，为程序员们提供了一些实用的建议。本书由 50 个关键条款（也可看作是 50 个小主题）组成，这些主题反映了我（及其他 C#顾问）和 C#开发人员共事时遇到的最常见问题。

与很多 C#开发人员一样，我是在从事 10 多年 C++开发之后开始使用 C#的。在本书中，讨论了哪些情况下遵循 C++实践可能会在使用 C#时引发的问题。有一些使用 C#的开发人员有着深厚的 Java 背景，他们可能会发现有些变化相当明显。因为从 Java 到 C#，一些最佳实践发生了改变，我建议 Java 开发者要格外注意有关值类型的论述（参见第 1 章）。此外，.NET 垃圾收集器和 JVM 垃圾收集器的行为方式也不尽相同（参见第 2 章）。

本书中的条款汇集了我最常提供给开发者的建议。虽然并非所有条款都是通用的，但大多数条款都可以很容易地应用到日常的编程工作中。这些条款涵盖了对属性（条款 1）、条件编译（条款 4）、常量性类型（条款 7）、相等判断（条款 9）、ICloneable（条款 27）和 new 修饰符（条款 29）的论述。我的经验是，在大多数情况下，减少开发时间和编写出色的代码应该是程序员的主要目标。某些科学和工程应用程序最重视的可能是系统的整体性能。对其他应用程序而言，凡事都应该围绕可伸缩性展开。对于不同的目标，可能会找到某些情况下比较重要（或不太重要）的信息。针对这一问题，我设法对各种目标进行了详细的解释说明。书中关于 readonly 和 const（条款 2）、可序列化的类型（条款 25）、CLS 兼容（条款 30）、Web 方法（条款 34）和 DataSet（条款 41）的讨论针对某些特定的设计目标。这些目标在相应的条款中有清楚的说明，这样读者就可以在特定的情况下决定最适用的做法。

虽然本书中的每个条款都是独立的，但是这些条款是围绕一些重要的主题（如 C#语法、资源管理和对象及组件设计）组织起来的，理解这一点非常重要。这并非无心之举。我的目的就是通过将每个条款构建在之前的条款之上，并合理地利用之前的条款，来让读者最大限度地学习书中的内容。尽管如此，大家仍然不要忘了举一反三。对于特定的问题，本书也可以作为一个理想的查询工具。

要记住的是，本书并不是 C#语言的教程或指南，也不是为了教授大家 C#语法或结构。我的目标是为大家在不同的情况下使用什么语言构造最好提供指导。

本书面向的读者

本书是为专业的开发人员，也就是那些在日常工作中使用 C#的程序员们编写的。本书的阅读前提是读者有面向对象的编程经验，并且至少用过一种 C 系列语言（C、C++、C#或 Java）。有 Visual Basic 6 背景的开发人员在阅读本书之前，应该先熟悉 C#语法和面向对象设计。

另外，读者应该在.NET 的重要领域有一些经验：Web Services、ADO.NET、Web Forms 和 Windows Forms。

为了充分利用本书，大家应该理解.NET 环境处理程序集的方式、微软中间语言（MSIL）和可执行代码。C#编译器生成的程序集会包含 MSIL，我经常将其简写为 IL。加载程序集的时候， JIT（Just In Time）编译器会将 MSIL 转换为机器可执行的代码。C#编译器确实会执行一些优化，但是 JIT 编译器会负责处理很多更高效的优化，如内联。在书中，我对各种优化所涉及的过程进行了说明。这种两阶段的编译过程对于在不同情形下哪种构造的表现最佳有着很重要的影响。

本书内容

第 1 章“C#语言元素”讨论的是 C#语法元素和 System.Object 的核心方法，System.Object 是编写每一个类型都要涉及的。声明、语句、算法和 System.Object 接口，这些都是编写 C#代码时必须时刻记住的主题。此外，与值类型和引用类型之间的区别直接相关的条款也都在本章。根据使用的是引用类型（类）还是值类型（结构），很多条款内容都有一些不同。在深入阅读本书之前，我强烈建议大家先阅读有关值类型和引用类型的讨论（条款 6~8）。

第 2 章“.NET 资源管理”涵盖了 C#和.NET 的资源管理问题。大家会学习如何针对.NET 管理的执行环境优化资源分配和使用模式。是的，.NET 垃圾收集器使我们的工作简单了很多。内存管理是环境的职责，而非开发人员的职责。但是，我们的行为对垃圾收集器在应用程序中的执行效果会产生大的影响。而且，尽管内存不是我们的问题，但管理非内存资源仍然是我们的职责，后者可以通过 IDisposable 进行处理。在这里，大家可以学习.NET 中资源管理的最佳做法。

第 3 章“使用 C#表达设计”从 C#的角度讲解了面向对象设计。C#提供了丰富的工具供我们使用。有时候，相同的问题可以用很多不同的方法解决：使用接口、委托、事件或者特性和反射。选用哪一种方式，对系统将来的可维护性会产生很大的影响。选择最佳的设计表示可以帮助程序员们更容易地使用类型。最自然的表示会使我们的意图更加清晰。这样，类型就会比较容易使用，而且不容易误用。第 3 章中的条款集中讲解了我们所做的设计决定，以及各种 C#惯用法最适用的场合。

第 4 章“创建二进制组件”讲解了组件和语言互操作性。大家将学习如何在不牺牲 C#功能的情况下，编写可被其他.NET 语言使用的组件。还将学习如何将类细分成组件，来升级应用程序的某些部分。我们应该能在不重新发布整个应用程序的情况下发布组件的新版本。

第 5 章“使用框架”讲解了.NET 框架未充分使用的部分。我看到很多开发人员非常希望创建自己的软件，而不是使用已经构建好的软件。这可能是由.NET 框架的体积造成的，也可能因为框架是全新的。这些条款涵盖了框架中那些我曾见过开发人员做重复劳动、而非使用业已存在的功能的部分。通过学习更高效地使用框架，大家可以节省宝贵的时间。

第 6 章“杂项讨论”以不适合其他分类的条款以及对未来的展望作为全书的结尾。有关 C# 2.0、标准、异常安全（exception-safe）的代码、安全和互操作的信息，都可以在这里找到。

关于条款

我写这些条款是为了向大家提供编写 C#软件的简洁明了的建议。书中有一些指导方针是通用的，因为它们会影响程序的正确性，如正确初始化数据成员（参见第 2 章）。有一些指导方针不是很容易理解，并且在.NET 社区中引发过很多争论，如是否使用 ADO.NET DataSet。我个人认为使用它们可以节省很多时间（参见条款 41），其他一些专业的程序员，同时也是我非常尊敬的程序员，对此并不同意。它其实取决于我们正在构建的软件性质。我的立场是尽量节省时间。如果是编写大量在基于.NET 和基于 Java 的系统之间传输信息的软件，DataSet 就是个糟糕的主意。在整本书中，我为所做的全部建议都给出了理由。如果其理由并不适用于你碰到的情况，那就不要采纳书中的建议。当建议是普遍适用时，我通常会省略其显而易见的理由：如果不这样做，程序就起不了作用。

版式和代码约定

写编程语言图书的一个困难之处在于，语言设计者用一些英文单词表示非常特殊的新含义，这就导致了一些很难理解的句子。“Develop interfaces with interfaces”就是一个例子。因此，我在使用语言关键字时，都采用了代码体。

本书使用了很多相关的 C#术语。在提到类型的成员时，它是指可以成为类型的一部分的任何定义：方法、属性、字段、索引器、事件、枚举或者委托。当应用的只是一种定义时，我会使用一个更加具体的术语。对于书中的许多术语，大家可能熟悉，也可能还不熟悉。当这些术语第一次在正文中出现时，它们会以楷体形式表现，并给出定义。

本书的范例都是简短、专注的代码段，以示范特定条款中的建议。列出它们是为了强调遵循建议的好处。它们并不是可以加入到读者当前程序中的完整范例。大家不能简单地复制代码清单，然后编译它们。所有代码清单我都省略了很多细节。在所有情况下，我们都预设已存在如下常用的 using 语句：

```
using System;
using System.IO;
using System.Collections;
using System.Data;
```

当使用不太常见的命名空间时，我会确保让读者看到相关的命名空间。简短的范例会使用完全限定类名称，而长的范例则会包含不太常用的 using 语句。

范例中的代码也比较随意。例如，当显示下列代码时：

```
string s1 = GetMessage();
```

如果和论述的内容无关，我可能不会显示 GetMessage()例程的主体。当我省略代码时，读者可以假定缺失的方法做的是一些明显且合理的事情。我这样做的目的是为了让我们把焦点聚在特殊的主题上。通过省略和主题无关的代码，我们的注意力就不会分散。这样还能让各个条款保持简短，以使大家能够在短时间内完成学习。

关于 C# 2.0

我之所以对新的 C# 2.0 版本所言甚少，有两个原因。首先，本书中的大部分建议也同样适用于 C# 2.0。虽然 C# 2.0 是一个非常重大的升级版本，但它是建立在 C# 1.0 基础之上的，且并没有让现如今的建议失效。对于最佳实践有可能发生变化的地方，我已经在文中给出了说明。

第二个原因是现在编写新的 C# 2.0 功能的高效用法还为时过早。本书的内容是基于我以及我的同事使用 C# 1.0 的已有经验。我们对于 C#2.0 中的新功能还没有足够的经验，因而也就不了解能够应用到日常任务中的最佳做法。当在书中编写 C# 2.0 新功能的高效用法的时机还未成熟时，我并不想误导读者。

建议、反馈及获取本书的更新内容

本书内容基于我的经验以及和同事们的交流。如果读者有不同的经验，或者有任何疑问或意见，我愿洗耳恭听。请通过电子邮件和我联系：wwagner@srtsolutions.com。我会把这些意见放在网上，作为本书的延伸。登录 www.srtsolutions.com/EffectiveCSharp，可以看到当前的讨论。

致谢

虽然写作似乎是一件孤独的事情，但本书却是一大群人的成果。我非常幸运，认识两位出色的编辑 Stephane Nakib 和 Joan Murray。Stephane Nakib 第一次联系我为 Addison Wesley 写作是在一年多以前。我当时心存疑虑，因为书店里到处都是.NET 和 C#方面的书籍。在 C# 2.0 面世了足够的时间，可以大书特书之前，我一直看不出为 C#和.NET 再写一本参考书、教程或编程书籍的必要性。我们讨论过好几个想法，话题总是回到写一本有关 C#最佳实践的图书。在进行这些讨论的过程中，Stephane 告诉我，Scott Meyers 开始着手主编一个 Effective 系列，其风格延续了他的 Effective C++系列图书。我买的 Scott 的三本书都被我翻得非常破旧。我还将它们推荐给了我认识的每一位专业 C++程序员。他的写作风格清晰而简明。每一个建议条款都有过硬的理由。Effective 丛书是很好的资源，而且其体例使得读者很容易就能记住其中的建议。我认识很多 C++开发人员，他们复印了书的目录，并把它钉在书房的墙上，不断提醒自己。Stephane 一提到写作 Effective C#的想法，我就欣然接受了这个机会。这本书把我曾为 C#开发人员给出的所有建议收录在一起。我很荣幸能成为该系列图书的一个作者。和 Scott 一起工作让我学到了很多东西。我真心希望本书能够像 Scott 的书提高了我的 C++技巧那样，帮助大家提高 C#应用技巧。

Stephane 帮助落实了写作 Effective C#的想法，她审读了提纲和草稿，并在该书的早期写作过程中给予了充分的支持。当她抽身离开的时候，Joan Murray 接管了这个项目，并毫无倦怠地负责了原稿的写作管理。Ebony Haight 作为编辑助理，在整个过程中提供了不间断的帮助。Krista Hansing 完成了所有编辑和转换编程行话的工作。Christy Hackerd 完成了所有把 Word 文档转变为成书的工作。

书中如有错误，应由我来负责。出色的审稿团队修改了绝大多数的错误、冗长和表述不清的问题。最值得一提的是，Brian Noyes、Rob Steel、Josh Holmes 和 Doug Holland 使得最终的正文比初期的草稿更加正确和有用。另外，还要感谢安阿伯计算机学会、大湖区.NET 用户组、Greater Lansing 用户组和西密歇根州.NET 用户组的所有成员，他们听取了有关这些条款的议论，并提供了出色的反馈。

特别要提到的是，Scott Meyers 的参与对本书的最终版本有着巨大的积极影响。和他讨论本书的早期草稿，使得我更加明白为什么自己会把 Effective C++丛书用得破旧不堪。再小的问题，也逃不过他的眼睛。

我要感谢 MyST Technology Partners（myst-technology.com）的 Andy Seidl 和 Bill French。我使用了一个基于 MyST 的安全博客网址向审稿人公布了各个条款的早期草稿。之后我们向公众公开了部分站点，以便大家能够以在线的格式看到本书的部分内容。登录 www.srtsolutions.com/EffectiveCSharp，可以阅读在线版本。

到目前为止，我已经为杂志写了好几年的文章，我要在此感谢那个将我引进门的人：Richard Hale Shaw。他在自己参与创办的杂志《Visual C++开发者》上邀请我这个未经检验的作者开设了一个专栏。如果没有他的帮助，我不会发现自己对写作的热爱。没有他最初给我的帮助，我也不会有机会为 Visual Studio 杂志、C# Pro 或 ASP.NET Pro 撰稿。

一路走来，我幸运地和不同杂志的很多出色的编辑共过事。我想把他们的名字全都列在这里，但空间不允许。有一个人非提不可，那就是 Elden Nelson。我享受与他共事的所有时光，他对我的写作风格产生了很大的积极影响。

我的业务伙伴 Josh Holmes 和 Dianne Marsh，他们容忍了我对公司业务的有限参与，而让我把时间用来写作本书。他们还帮助审阅了我的原稿、想法和条款中的思想。

在整个漫长的写作过程中，我的父母 Bill 和 Alice Wagner“做事要有始有终”的忠告，成为我最终完成本书的唯一原因。

最后也是最重要的，我要感谢我的家人 Marlene、Lara、Sarah 和 Scott。写书会占用大量的业余时间。在我为本书付出了所有时间之后，他们表现出来的却是始终如一的耐心。

第一章 C#语言元素

为什么程序已经可以正常工作了，我们还要改变它们呢？答案就是我们可以让它们变得更好。我们常常会改变所使用的工具或者语言，因为新的工具或者语言更富生产力。如果固守旧有的习惯，我们将得不到期望的结果。对于 C#这种和我们已经熟悉的语言（如 C++或 Java）有诸多共通之处的新语言，情况更是如此。人们很容易回到旧的习惯中去。当然，这些旧的习惯绝大多数都很好，C#语言的设计者们也确实希望我们能够利用这些旧习惯下所获取的知识。但是，为了让 C#和公共语言运行库（Common Language Runtime，CLR）能够更好地集成在一起，从而为面向组件的软件开发提供更好的支持，这些设计者们不可避免地需要添加或者改变某些元素。本章将讨论那些在 C#中应该改变的旧习惯，以及对应的新的推荐做法。

原则 1：始终能的使用属性(property)，而不是可直接访问的 Data Member

Always use properties instead of accessible data members.

在 C#里，Property 已经晋升为一类公民。如果你的类里还有 Public 的变量，Stop! 如果你还在手写 get and set 方法,Stop! Property 在不破坏你类的封装的情况下,仍可以把类的 data member 变成 public interface 的一部分。访问 Property 的方式和访问 data member 的方式一样,但 Property 是用 methods 实现的。

有些类的成员只能用 data 最好的表示，比如：你一个客户的名字，一个点的坐标，等等。而 Property 就是用来欺骗使用你类的客户，让它们错误的认为它们在访问你类的 public 变量。你还可以通过 Property 的实现方法来控制 Property 的访问。

.Net Framework 假定你使用 Property 来让外界访问你类里想让外界访问到的 data member（也就是 public data member）。实际上也是这样的，因为.Net 的 data binding 只支持 Property，而不支持 public data member 的访问。Data binding 的目的就是把一个 object 的 Property 绑定到一个用户界面的 control 上，web control 或者 windows form control. Data binding 是通过 reflection 来实现的，如下例：

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

这段 code 就是把 textBoxCity 的 Text Property 绑定到 address 这个 object 的 City Property 上。如果你把 address 的 City Property 改成 public data member，这段 code 是不会运行的。因为.Net Framework Class Library 的设计者不支持你的这种行为，他们认为 public data member 是非常不好的行为和习惯，所以他们不会支持，他们想让你遵从正确的 Object Oriented 设计方法。Data binding 也不会去找 get and set methods，所以一定要用 Property，而不是传统的 get and set methods.

你也许要说，data binding 只适用于那些含有要显示在用户界面的元素的类。但实际情况并不是这样，对于你所有的类，都要使用 Property 而不是 public data member。因为当有新的需求时，通过修改 Property 的实现方法来适应这个新的需求，要比在你的程序里修改所有的 public data member 去适应这个需求容易太多了。比如说你以前定义了一个类 customer，现在你发现由于当初的粗心没有强制 customer 姓名不能为空，如果你使用了 Property，你可以非常轻松的添加一个检查机制，如下面这段 code:

```
public class Customer
{
    private string _name;
    public string Name
```

```
        {
            get
            {
                return _name;
            }
            set
            {
                if ((value == null) || (value.Length == 0))
                {
                    throw new ArgumentException("Name can    not be blank", "Name");
                }
                _name = value;
            }
        }
        //...
    }
```

如果你使用了 public data member，你就要找遍你的程序，在每个地方都修改，那样就很愚蠢了。而且浪费了无数青春好时光。

因为 Property 是用 methods 实现的，所以添加 multi-threaded 的支持是非常方便的。比如想要添加同步访问的支持：

```
    public string Name
    {
        get
        {
            lock(this)
            {
                return _name;
            }
        }
        set
        {
            lock(this)
            {
                _name = value;
            }
        }
    }
```

因为 Property 是用 methods 实现的，所以它拥有 methods 所拥有的一切。Property 可以被定义为 virtual：

```
    public class Customer
    {
        private string _name;
        public virtual string Name
        {
            get
            {
```

```
            return _name;
        }
        set
        {
            _name = value;
        }
    }
    //...
}
```

显而易见，你也可以把 Property 扩展为 abstract，甚至成为 interface 的一部分。

```
public interface INameValuePair
{
    object Name
    {
        get;
    }
    object Value
    {
        get;
        set;
    }
}
```

你当然也可以扩展出 const 和 nonconst 版本的 interface。

```
public interface IConstNameValuePair
{
    object Name
    {
        get;
    }
    object Value
    {
        get;
    }
}
public interface INameValuePair
{
    object Value
    {
        get;
        set;
    }
}
//usage:
public class Stuff : IConstNameValuePair, INameValuePair
{
```

```
        private string _name;
        private object _value;
        #region IConstNameValuePair Members
        public object Name
        {
            get
            {
                return _name;
            }
        }
        object IConstNameValuePair.Value
        {
            get
            {
                return _value;
            }
        }
        #endregion
        #region INameValuePair Members
        public object Value
        {
            get
            {
                return _value;
            }
            set
            {
                _value = value;
            }
        }
        #endregion
    }
```

如前所述，Property 是访问内部数据的 method 的扩展，它拥有 member function 的一切特性。

因为实现 Property 访问的方法 get and set 是独立的两个 method，在 C# 2.0 中，你可以给它们定义不同的访问级别，来更好的控制类成员的可见性，如下例：

```
    public class Customer
    {
        private string _name;
        public virtual string Name
        {
            get
            {
                return _name;
            }
            protected set
```

```
            {
                _name = value;
            }
        }
        //...
    }
```

Property 的语法已经超越了单纯的 data field。如果你的类包含 indexed item，你可以使用 indexer(参数化的 Property)，你可以创建一个可返回一个序列元素的 Property，如下例：

```
    public int this[int index]
    {
        get
        {
              return _theValues[index];
        }
        set
        {
              _theValues[index] = value;
        }
    }
    //usage:
    int val = MyObject[i];
```

indexer 和单元素 Property 有着相同的特性。一维的 indexer 可以用于 data binding，二维和多维的 indexer 可以用来实现其他的数据结构，比如 map 和 dictionary：

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

多维的 indexer 的每个 axis 上的数据类型可以相同，也可以不同：

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

```
        {
            return ComputeValue(x, name);
        }
    }
```

所有的 indexer 都必须也只能用 this 来定义，所以参数表相同的 indexer，每个类最多只能有一个。

因为使用 Property 和 data member 对于数据访问的 code 没有什么区别，比如：

```
public class Customer
{
    public string Name;
    //...
}
```

在这个类中使用了 public data member，数据访问的 code 如下：

```
string name = CustomerOne.Name;
CustomerOne.Name = "customer name";
```

你也许会想，如果在以后的修改中，用 Property 来代替 public data member 是可行的，因为数据访问的 code 相同，但实际上这是行不通的。确实，访问 Property 和访问 data member 的 code 是相同的，但 Property 不是 data，访问 Property 所产生的 IL code 和数据访问的 IL code 是不一样的。所以访问 Property 和访问 data member 只具有 code 兼容性，而不具有 binary 的兼容性。如果有兴趣，你可以使用 Reflector (http://www.aisto.com/roeder/dotnet/)来分析使用 Property 和 public data member 的类。

你会发现在使用 Property 的类中，存在.property directive，这个 directive 定义了 Property 的类型以及 get and set 实现方法。Get and set 都被标注为 hidebysig, specialname。也就是说它们不能被 C#源代码直接调用，它们也不是正是的类型定义。你只能通过 Property 来访问它们。

C#的编译器会根据类的情况(是用 Property 还是 data member)来自动产生不同的 IL code。如上所述，访问 Property 和访问 data member 只具有 code 兼容性，而不具有 binary 的兼容性。所以，如果你改变最初的设计，用 Property 来代替 public data member 的话，你必须重新编译整个程序。这使得升级已经部署的程序或 assembly 是非常的麻烦。

那么两种实现谁的效率更好呢？Property 确实不会比 public data member 快，但也不一定会慢。因为 JIT 对 Property 的存取方法 set and get 进行 inline 的优化。这时，Property 和 public data member 的效率是一样的。即使 Property 的存取方法没有被 inline 优化，它和 public data member 的效率差别也只是一个可以忽略的 function call。只有在很少的情况下，这种差别才可以被测量出来。

总而言之，当你想让你类内部的数据被外界访问到时(不管是 public 还是 protected)，一定要用 Property。对于序列和字典，使用 indexer。你类的 data member 永远应该是 private，绝无例外。使用 Property，你可以得到如下好处：

1．Data binding 支持

2．对于需求变化有更强的适应性，更方便的修改实现方法

记住，现在多花 1 分钟使用 Property，会在你修改程序以适应设计变化时，为你节约 n 小时。

原则 2：为你的常量选择 readonly 而不是 const

Prefer readonly to const

对于常量，C#里有两个不同的版本：运行时常量和编译时常量。

因为他们有不同的表现行为，所以当你使用不当时，将会损伤程序性能或者出现错误。

两害相权取其轻，当我们不得不选择一个的时候，我们宁可选择一个运行慢一点但正确的那一个，而不是运行快一点但有错误的那个。基于这个理由，你应该选择运行时常量而不是编译时常量(译注：这里隐藏的说明了编译时常量效率更高，但可能会有错误)。

编译时常量更快更直接，但在可维护性上远不及运行时常量。保留编译时常量是为了满足那些对性能要求克刻，且随着程序运行时间的过去，其值永远不发生改变的常量使用的(译注：这说明编译时常量是可以不被 C#采用的，但考虑到性能问题，还是做了保留)。

你可以用关键字 readonly 来声明(declare)一个运行时常量，编译时常量是用关键字 const 声明的。

```
//Compile time constant:
public cocnst int _Millennium = 2000;
//Runtime constant:
public static readonly int _ThisYear = 2004;
```

编译时常量与运行时常量不同之处表现在如何对他们的访问上。

一个编译时常量会被目标代码中的值直接取代。下面的代码：

```
if(myDateTime.Year == _Millennium)
```

会与下面写的代码编译成完全相同的 IL 代码：

```
if(myDateTime.Year == 2000)
```

运行时常量的值是在运行时确定的。当你引用一个只读常量时(read-only)IL 会为你引用一个运行时常量的变量，而不是直接使用该值。

当你任意的使用其中一个常量时，这些区别就在一些限制上表现出来。编译时常量只能是基本类型(primitive types)(built-in integral and floating-poing types),枚举或者是字符串。这些就是你只能给运行时常量在初始化时赋值的类型。这些基本类就是可以被编译器在编译 IL 代码时直接用真实的值所取代的数据类型。下面的代码块(construct)不能通过编译。你不能用 new 运算符初始化一个编译时常量，即使这个数据类型是值类型。

```
//Does not complie, use readonly instead:
private const DateTime _classCreation = new DateTime(2000,1,1,0,0,0);
```

编译时常量仅限于数字和字符串。只读变量，也就是运行时常量，在构造函数(constructor)执行完成后它们是不以能被修改的。但只读变量是所有不同的，因为他们是在运行时才赋值的。当你使用运行时常量时，你有更大的可伸缩性。有一点要注意的是，运行时常量可以是任何类型的数据。而且你必须在构造函数里对他们初始化，或者你可以用任何一个初始化函数来完成。你可以添加一个 DateTime 结构的只读变量(--运行时常量)，但你不能添加一个 DateTime 结构的(编译时)常量。

你可以把每一个实例(的常量)指定为只读的，从而为每一个类的实例存放不同的值。与编译时常量不同的是，它只能是静态的。

只读数据最重要的区别是他们在运行时才确定值。当你使用只读变量时，IL 会为你产生一个对只读变量引用，而不是直接产生数值。随着时间的推移，这个区别在(系统)维护上有深远的潜在影响。

编译时常量生成的 IL 代码就跟直接使用数值时生成的 IL 是一样的，即使是在跨程序集时：一个程序集里的编译时常量在另一个程序集会保留着同样的值(译注：这里说的不是很清楚，看后面的这个例子可能会更清楚一些)。

编译时常量和运行时常量的赋值方法对运行时的兼容性有所影响。

假设你已经在程序集 Infrastructure 中同时定义了一个 const 和一个 readonly 变量：

```
public class UserfulValues
{
    public static readonly int StartValue = 5;
    public const int EndValue = 10;
}
```

同时，在另一个程序集(译注：这个程序集认为是我们做测试的应用程序的程序集，下面所说的应用程序的程序集都是指的这个程序集)中，你引用了这些值：

```
for(int i=UserfulValues.StartValue;i<UserfulValues.EndValue;i++)
{
    Console.WriteLine("value is {0}",i);
}
```

如果你运行这个简单测试程序，你可以看到下面明显的结果：

value is 5

value is 6

...

value is 9

过后，你又为程序集 Infrastructure 发布了个新的版本，并做了如下的修改：

```
public class UserfulValues
{
    public static readonly int StartValue = 105;
    public const int EndValue = 120;
}
```

你单独的发布了程序集 Infrastructure 而没有全部编译你的程序，你希望得到下面的：

value is 105

value is 106

...

value is 119

事实上，你什么也得不到。上面的循环已经是用 105 开始而用 10 来结束。C#编译器(在编译时)把常量用 10 来代替应用程序的程序集中的使用，而不是用常量 EndValue 所存储的值。而常量 StartValue 的值，它是被申明为只读的，它可以在运行时重新读取该常量的值。因此，应用程序的程序集可以在不用重新编译的情况下使用新的数据，简单的编译一下 Infrastructure 程序集，然后重新布署安装一下，就足够让你的客户可能使用这些新的数据了。更新的编译时常量应该看成是接口的变化。你必须重新编译所有引用到编译时常量的代码。更新的运行时常量则可以当成是实现的改变，这于在客户端已经存在的二进制代码是兼容的。用 MSIL 解释一下前面的那个循环里发生了什么：

```
IL_0000: ldsfld int32 Chapter1.UserfulValues::StartValue
IL_0005: stloc.0
IL_0006: br.s IL_001c
IL_0008: ldstr "value is {0}"
IL_000d: ldloc.0
IL_000e: box [mscrolib]System.Int32
IL_0013: call void [mscrolib]System.Console::WriteLine(string,object)
IL_0018: ldloc.0
IL_0019: ldc.i4.1
IL_001a: add
IL_001b: stloc.0
IL_001c: ldloc.0
IL_001d: ldc.i4.s 10
IL_001f: blt.s IL_0008
```

从 MSIL 命令清单的最上面一行你可以看到，StartValue(的值)是动态载入的。

但是，在 MSIL 命令的最后，结束条件是把值 10 当成硬代码(hard-coded)使用的。

另一方面，有些时候你也须要为某些值使用编译时常量。例如：考虑一个须要识别不同版本的续列化情形。用来标识一个特殊版本号的常量应该是一个编译时常量，它们决不会发生改变。而当前版本号则应该是一个运行时常量，在不同的版本发布后会有所改变。

```
private const int VERSION_1_0 = 0x0100;
private const int VERSION_1_1 = 0x0101;
private const int VERSION_1_2 = 0x0102;
```

```
//major release;
private const int VERSION_2_0 = 0x0200;
//Chech for the current version:
private static readonly int CURRENT_VERSION = VERSION_2_0;
```

在每次存盘时，你用运行常量来保存当前版本号。

```
//Read fom persistent storage, check stored version against complie-time constant:
protected MyType(SerializationInfo info, StreamingContext cntxt)
{
    int storedVersion = info.GetInt32("VERSION");
    switch(storedVersion){
    case VERSION_2_0:
        readVersion2(info,cntxt);
        break;
    case VERSION_1_1:
        readVersion1(info,cntxt);
        break;
    //etc.
    }
}
//Write the current version:
[SecurityPermissionAttribute(SecurityAction.Demand,SerializationFormatter = true)]
void ISerializable.GetObjectData(SerializationInfo inf,StreamingContext cxt)
{
    //use runtime constant for currnet version
    inf.AddValue("VERSION",CURRENT_VERSION);
    //write remaining delements...
}
```

最后一个有利的原因而使我们要使用编译时常量，就是它的性能。比起运行时常量，已知的编译时常量可以更直接有效的被访问。然而，性能上的功效是甚微的，并且应该与可伸缩性的降低进行一个权衡。Be sure to profile performace differences before giveing up the flexibility.

const 的值必须在编译时被确定，(它们可以是)：属性参数，枚举定义，以及一小部份你认为应该定义一个值且该值不能在不同的版本发布时发生改变的常量。

无论如何，宁愿选择伸缩性更强的运行时常量。

原则 3：选择 is 或者 as 操作符而不是做强制类型转换

Prefer the is or as Operators to Casts

C#是一个强数据类型语言。好的编程实践意味着当可以避免从一种数据类型强制转化为另种数据类型时，我们应该尽我们的所能来避免它。但在某些时候，运行时类型检测是不可避免的。在 C#里，大多数时候你要为调用函数的参数使用 System.Object 类型，因为 Framwork 已经为我们定义了函数的原型。你很可能要试图把那些类型进行向下转化为其它类型的接口或者类。你有两个选择：用 as 运算符，或者，采用旧式的 C 风格，强制转换。(不管是哪一种，)你还必须对变量进行保护：你可以试着用 is 进行转换，然而再用 as 进行转换或者强制转换。

无论何时，正确的选择是用 as 运算符进行类型转换。因为比起盲目的强制转换它更安全，而且在运行时效率更高。用 as 和 is 运算符进行转换时，并不是对所有的用户定义的类型都能完成的。它们只在运行时类型和目标类型匹配的时候，转换才能成功。它们决不会构造一个新的对象来满足(转化)要求。

看一个例子。你写了一段代码，要转换一个任意类型的对象实例到一个 MyType 类型的实例。你是这样写代码的：

```
object o = Factory.GetObject();
// Version one:
MyType t = o as MyType;
if (t != null)
{
    // work with t, it's a MyType.
} else
{
    // report the failure.
}
```

或者你这样写：

```
object o = Factory.GetObject();
// Version two:
try
{
    MyType t;
    t = (MyType) o;
    if (t != null)
    {
        // work with T, it's a MyType.
    } else
    {
        // Report a null reference failure.
    }
}
catch
{
    // report the conversion failure.
}
```

你会同意第一种写法更简单更容易读。它没有 try/catch 结构，所以你可以同时避免(性能)开销和(多写)代码。我们注意到，强制转换的方法为了检测转换是否把一个 null 的对象进行强制转换，而不得不添加一个捕获异常的结构。null 可以被转换为任意的引用类型，但 as 操作符就算是转化一个 null 的引用时，也会(安全的)返回一个 null。所以当你用强制类型转换时，就得用一个 try/catch 结构来捕获转换 null 时的异常。用 as 进行转换的时就，就只用简单的检测一下转化后的实例不为 null 就行了。

(译注：被转换对象和转换后的结果都有可能为 null，上面就是对这两种 null 进行了说明，注意区分。强制转换是不安全的，可能会有异常抛出，因此要用 try/catch 结构来保证程序正常运行，而 as 转换是安全的，不会有异常抛出，但在转换失败后，其结果为 null)

强制转换与 as 转换最大的区别表现在如何对待用户定义类型的转换。

与其它运算不一样，as 和 is 运算符在运行时要检测转换目标的类型。如果一个指定对象不是要求转换的类型，或者它是从要求转换类型那里派生的，转换会失败。另一方面，强制转换可以用转换操作把一个对象转换成要求的类型。这还包括对内置数据(built-in numberic)类型的转换。强制转换一个 long 到一个 short 可能会丢失数据。

同样的问题也隐藏在对用户定义类型的转换上。考虑这样的类型：

```
public class SecondType
```

```
{
    private MyType _value;
    // other details elided
    // Conversion operator.
    // This converts a SecondType to
    // a MyType, see item 29.
    public static implicit operator MyType(SecondType t)
    {
        return t._value;
    }
}
```

假设代码片段中开始的 Factory.GetObject()函数返回的是 SecondType 类型的数据:

```
object o = Factory.GetObject();
// o is a SecondType:
MyType t = o as MyType; // Fails. o is not MyType
if (t != null)
{
    // work with t, it's a MyType.
}
else
{
    // report the failure.
}
// Version two:
try
{
    MyType t1;
    t = (MyType) o; // Fails. o is not MyType
    if (t1 != null)
    {
        // work with t1, it's a MyType.
    }
    else
    {
        // Report a null reference failure.
    }
}
catch
{
    // report the conversion failure.
}
```

两种转换都失败了。但是我告诉过你，强制转化可以在用户定义的类型上完成。你应该想到强制转化会成功。你是对的--(如果)它们跟像你想的一样是会成功的。但是转换失败了，因为你的编译器为对象 o 产生的代码是基于编译时类型。而对于运行时对象 o，编译器什么也不知道，它们被视为 System.Obejct 类型。编译器认为，不存在 System.Object 类型到用户类型 MyType 的转换。它检测了 System.Object 和 MyType 的定义。缺少任意的用

户定义类型转换，编译器(为我们)生成了用于检测运行时对象 o 的代码，并且检测它是不是 MyType 类型。因为对象 o 是 SecondType 类型，所以失败了。编译器并不去检测实际运行时对象 o 是否可以被转换为 MyType 类型。

如果你使用下面的代码段，你应该可以成功的完成从 SecondType 到 MyType 的转换：

```
object o = Factory.GetObject();
// Version three:
SecondType st = o as SecondType;
try
{
    MyType t;
    t = (MyType) st;
    if (t != null)
    {
        // work with T, it's a MyType.
    } else
    {
        // Report a null reference failure.
    }
}
catch
{
    // report the failure.
}
```

你决不应该写出如此糟糕的代码，但它确实解决了一个很常见的难题。尽管你决不应该这样写代码，但你可以写一个函数，用一个 System.Object 参数来完成正确的转换：

```
object o = Factory.GetObject();
DoStuffWithObject(o);
private void DoStuffWithObject(object o2)
{
    try
    {
        MyType t;
        t = (MyType) o2; // Fails. o is not MyType
        if (t != null)
        {
            // work with T, it's a MyType.
        }
        else
        {
            // Report a null reference failure.
        }
    }
    catch
    {
        // report the conversion failure.
    }
```

```
}
```

记住，对一个用户定义类型的对象，转换操作只是在编译时，而不是在运行时。在运行时存在介于 o2 和 MyType 之间的转换并没有关系，(因为)编译器并不知道也不关心这些。这样的语句有不同的行为，这要取决于对 st 类型的申明：

```
t = (MyType) st;
```

(译注：上面说的有些模糊。为什么上面的代码可能会有不同的行为的呢？不同的什么行为呢？主要就是：上面的这个转化，是在编译时还是在运行时!如果 st 是用户定义的类型，那么上面的转换是在编译时。编译器把 st 当成为 System.Object 类型来编译生成的 IL 代码，因此在运行时是无法把一个 Object 类型转化为 MyType 类型的。解决办法就是前面提到的方法,多加一条语句，先把 Object 类型转换为 SecondType，然后再强制转化为 MyType 类型。但是如果 st 是内置类型，那么它的转换是在运行时的，这样的转化或许会成功，看后面的说明。因此，类似这样的代码：MyType m_mytype = (m_secondType as SecondType) as MyType;是不能通过编译的，提示错误是无法在编译时把 SecondType 转化为 MyType，即使是重写了转换操作符。)

但下面的转换只会有一种行为，而不管 st 是什么类型。

所以你应该选择 as 来转换对象，而不是强制类型转换。实际上，如果这些类型与继承没有关系，但是用户自己定义的转换操作符是存在的，那么下面的语句转换将会得到一个编译错误：t = st as MyType;现在你应该明白要尽可能的使用 as，下面我们来讨论不能使用 as 的时候。as 运算符对值类型是无效的，下面的代码无法通过编译：

```
object o = Factory.GetValue();
int i = o as int; // Does not compile.
```

这是因为整形(ints)数据是值类型，并且它们永远不会为 null。当 o 不是一个整形的时候，i 应该取什么值呢？不管你选择什么值，它都将是一个无效的整数。因此，as 不能使用(在值类型数据上)。你可以坚持用强制转化：

```
object o = Factory.GetValue();
int i = 0;
try {
    i = (int) o;
}
catch
{
    i = 0;
}
```

但是你并没有必要这样坚持用强制转换。你可以用 is 语句取消可能因转换引发的异常：

```
object o = Factory.GetValue();
int i = 0;
if (o is int)
     i = (int) o;
```

(译注：is 和 as 一样，都是类型转换安全的，它们在任何时候都不会在转换时发生异常，因此可以先用 is 来安全的判断一下数据类型。与 as 不同的时，is 只是做类型检测并返回逻辑值，不做转换。)

如果 o 是其它可转化为整形的类型(译注：但 o 并不是真正的整形)，例如 double，那么 is 运算操作会返回 false。对于 null，is 总是返回 false。

is 只应该在你无法用 as 进行转换时使用。

另外，这是无意义的冗余：

```
// correct, but redundant:
object o = Factory.GetObject();
MyType t = null;
if (o is MyType)
  t = o as MyType;
```

如果你写下面的代码，那么跟上面一样，都是冗余的：

```
// correct, but redundant:
object o = Factory.GetObject();
MyType t = null;
if ((o as MyType) != null)
    t = o as MyType;
```

这都是低效且冗余的。如果你使用 as 来转换数据，那么用 is 来做检测是不必要的。只用检测返回类型是否为 null 就行了，这很简单。

现在，你已经知道 is,as 和强制转换之间的区别了。而在 foreach 的循环中，是使用的哪一种转换呢？

```
public void UseCollection(IEnumerable theCollection)
{
    foreach (MyType t in theCollection)
        t.DoStuff();
}
```

foreach 循环是用强制转换来完成把一个对象转换成循环可用的类型。上面的循环代码与下面手写的代码(hand-coded)是等效的：

```
public void UseCollection(IEnumerable theCollection)
{
    IEnumerator it = theCollection.GetEnumerator();
    while (it.MoveNext())
    {
        MyType t = (MyType) it.Current;
        t.DoStuff();
    }
}
```

foreach 须要用强制转换来同时支持对值类型和引用类型的转换。通过选择强制转化，foreach 循环就可以采用一样的行为，而不用管(循环)目标对象是什么类型。不管怎样，因为 foreach 循环是用的强制转换，因些它可能会产生 BadCastExceptions 的异常。

因为 IEnumberator.Current 返回一个 System.Obejct 对象，该对象没有(重写)转换操作符，所以它们没有一个满足(我们在上面做的)测试。

(译注：这里是说，如果你用一个集合存放了 SecondType，而你又想用 MyType 来对它进行 foreach 循环，那么转换是失败的，原因是在循环时，并不是用 SecondType，而是用的 System.Object，因此，在 foreach 循环里做的转换与前面说的：MyType t = (MyType) o;是一样的错误，这里的 o 是 SecondType，但是是以 System.Object 存在。)

正如你已经知道的，一个存放了 SecondType 的集合是不能在前面的函数 UseCollection 中使用循环的，这会是失败的。用强制转换的 foreach 循环不会在转换时检测循环集合中的对象是否具有有效的运行时类型。它只检测由 IEnumerator.Current 返回来的 System.Object 是否可转换为循环中使用的对象类型，这个例子中是 MyType 类型。

最后，有些时候你想知道一个对象的精确类型，而不仅仅是满足当前可转换的目标类型。as 运算符在为任何一个从目标类型上派生的对象进行转换时返回 true。GetType()方法取得一个对象的运行时对象，它提供了比 is 和 as 更严格的(类型)测试。GetType()返回的类型可以与指定的类型进行比较(，从而知道它是不是我们想要的类型)。

再次考虑这个函数：

```
public void UseCollection(IEnumerable theCollection)
{
    foreach (MyType t in theCollection)
```

```
            t.DoStuff();
        }
```

如果你添加了一个派生自 MyType 的新类 NewType，那么一个存放了 NewType 类型对象的集合可以很好的在 UseCollection 函数中工作。

```
        public class NewType : MyType
        {
            // contents elided.
        }
```

如果你想要写一个函数，使它对所有 MyType 类型的实例都能工作，上面的方法是非常不错的。如果你只想写一个函数，只对精确的 MyType 对象有效，那你必须用精确的类型比较。这里，你可以在 foreach 循环的内部完成。大多数时候，认为运行时确定对象的精确类型是很重要的，是在为对象做相等测试的时候。在大多数其它的比较中，由 is 和 as 提供的.isinst(译注：IL 指令)比较，从语义上讲已经是正确的了。

好的面向对象实践告诉我们，你应该避免类型转换，但有些时候我没别无选择。当你无法避免转换时，用(C#)语言为我们提供的 is 和 as 运算符来清楚的表达你的意思吧。不同方法的强制转换有不同的规则。从语义上讲，is 和 as 在绝大多转换上正确的，并当目标对象是正确的类型时， 它们总是成功的。应该选择这些基本指令来转换对象，至少它们会如你所期望的那样成功或者失败；而不是选择强制类型转换，这转换会产生一些意想不到的副作用。

原则 4：用条件属性而不是#if

Use Conditional Attributes Instead of #if

使用#if/#endif 块可以在同样源码上生成不同的编译(结果)，大多数 debug 和 release 两个版本。但它们决不是我们喜欢用的工具。由于#if/#endif 很容易被滥用，使得编写的代码难于理解且更难于调试。程序语言设计者有责任提供更好的工具，用于生成在不同运行环境下的机器代码。C#就提供了条件属性(Conditional attribute)来识别哪些方法可以根据环境设置来判断是否应该被调用。

(译注：属性在 C#里有两个单词，一个是 property 另一个是 attribute，它们有不是的意思，但译为中文时一般都是译为了属性。property 是指一个对象的性质，也就是 Item1 里说的属性。而这里的 attribute 指的是.net 为特殊的类，方法或者 property 附加的属性。可以在 MSDN 里查找 attribute 取得更多的帮助，总之要注意：attribute 与 property 的意思是完全不一样的。)

这个方法比条件编译#if/#endif 更加清晰明白。编译器可以识别 Conditional 属性，所以当条件属性被应用时，编译器可以很出色的完成工作。条件属性是在方法上使用的，所以这就使用你必须把不同条件下使用的代码要写到不同的方法里去。当你要为不同的条件生成不同的代码时，请使用条件属性而不是#if/#endif 块。

很多编程老手都在他们的项目里用条件编译来检测先决条件(per-conditions)和后续条件(post-conditions)。

(译注：per-conditions，先决条件，是指必须满足的条件，才能完成某项工作，而 post-conditions，后续条件，是指完成某项工作后一定会达到的条件。例如某个函数，把某个对象进行转化，它要求该对象不能为空，转化后，该对象一定为整形，那么：per-conditions 就是该对象不能为空，而 post-conditions 就是该对象为整形。例子不好，但可以理解这两个概念。)

你可能会写一个私有方法来检测所有的类及持久对象。这个方法可能会是一个条件编译块，这样可以使它只在 debug 时有效。

```
        private void CheckState()
        {
            // The Old way:
        #if DEBUG
            Trace.WriteLine("Entering CheckState for Person");
            // Grab the name of the calling routine:
```

```
        string methodName = new StackTrace().GetFrame(1).GetMethod().Name;
        Debug.Assert(_lastName != null, methodName, "Last Name cannot be null");
        Debug.Assert(_lastName.Length > 0, methodName, "Last Name cannot be blank");
        Debug.Assert(_firstName != null, methodName, "First Name cannot be null");
        Debug.Assert(_firstName.Length > 0, methodName, "First Name cannot be blank");
        Trace.WriteLine("Exiting CheckState for Person");
    #endif
    }
```

使用#if 和#endif 编译选项(pragmas)，你已经为你的发布版(release)编译出了一个空方法。这个 CheckState()方法会在所有的版本(debug 和 release)中调用。而在 release 中它什么也不做，但它要被调用。因此你还是得为例行公事的调用它而付出小部份代价。

不管怎样，上面的实践是可以正确工作的，但会导致一个只会出现在 release 中的细小 BUG。下面的就是一个常见的错误，它会告诉你用条件编译时会发生什么：

```
    public void Func()
    {
        string msg = null;
    #if DEBUG
        msg = GetDiagnostics();
    #endif
        Console.WriteLine(msg);
    }
```

这一切在 Debug 模式下工作的很正常，但在 release 下却输出的为空行。release 模式很乐意给你输出一个空行，然而这并不是你所期望的。傻眼了吧，但编译器帮不了你什么。你的条件编译块里的基础代码确实是这样逻辑。一些零散的#if/#endif 块使你的代码在不同的编译条件下很难得诊断(diagnose)。

C#有更好的选择：这就是条件属性。用条件属性，你可以在指定的编译环境下废弃一个类的部份函数， 而这个环境可是某个变量是否被定义，或者是某个变量具有明确的值。这一功能最常见的用法就是使你的代码具有调试时可用的声明。.Net 框架库已经为你提供了了基本泛型功能。这个例子告诉你如何使用.net 框架库里的兼容性的调试功能，也告诉你条件属性是如何工作的以及你在何时应该添加它：

当你建立了一个 Person 的对象时，你添加了一个方法来验证对象的不变数据(invariants)：

```
    private void CheckState()
    {
        // Grab the name of the calling routine:
        string methodName = new StackTrace().GetFrame(1).GetMethod().Name;
        Trace.WriteLine("Entering CheckState for Person:");
        Trace.Write("\tcalled by ");
        Trace.WriteLine(methodName);
        Debug.Assert(_lastName != null, methodName, "Last Name cannot be null");
        Debug.Assert(_lastName.Length > 0, methodName, "Last Name cannot be blank");
        Debug.Assert(_firstName != null, methodName, "First Name cannot be null");
        Debug.Assert(_firstName.Length > 0, methodName, "First Name cannot be blank");
        Trace.WriteLine("Exiting CheckState for Person");
    }
```

这这个方法上，你可能不必用到太多的库函数，让我简化一下。这个 StackTrace 类通过反射取得了调用方法的的名字。这样的代价是昂贵的，但它确实很好的简化了工作，例如生成程序流程的信息。这里，断定了 CheckState 所调用的方法的名字。被判定(determining)的方法是 System.Diagnostics.Debug 类的一部份，或者是

System.Diagnostics.Trace 类的一部份。Degbug.Assert 方法用来测试条件是否满足，并在条件为 false 时会终止应用程序。剩下的参数定义了在断言失败后要打印的消息。Trace.WriteLine 输出诊断消息到调试控制台。因此，这个方法会在 Person 对象不合法时输出消息到调试控制台，并终止应用程序。你可以把它做为一个先决条件或者后继条件，在所有的公共方法或者属性上调用这个方法。

```
public string LastName
{
    get
    {
        CheckState();
        return _lastName;
    }
    set
    {
        CheckState();
        _lastName = value;
        CheckState();
    }
}
```

在某人试图给 LastName 赋空值或者 null 时，CheckState 会在第一时间引发一个断言。然后你就可以修正你的属性设置器，来为 LastName 的参数做验证。这就是你想要的。

但这样的额外检测存在于每次的例行任务里。你希望只在调试版中才做额外的验证。这时候条件属性就应运而生了：

```
[Conditional("DEBUG")]
private void CheckState()
{
    // same code as above
}
```

Conditional 属性会告诉 C#编译器，这个方法只在编译环境变量 DEBUG 有定义时才被调用。同时，Conditional 属性不会影响 CheckState()函数生成的代码，只是修改对函数的调用。如果 DEBGU 标记被定义，你可以得到这：

```
public string LastName
{
    get
    {
        CheckState();
        return _lastName;
    }
    set
    {
        CheckState();
        _lastName = value;
        CheckState();
    }
}
```

如果不是，你得到的就是这：

```
public string LastName
{
    get
    {
        return _lastName;
    }
    set
    {
        _lastName = value;
    }
}
```

不管环境变量的状态如何，CheckState()的函数体是一样的。这只是一个例子，它告诉你为什么要弄明白.Net 里编译和 JIT 之间的区别。不管 DEBUG 环境变量是否被定义，CheckState()方法总会被编译且存在于程序集中。这或许看上去是低效的，但这只是占用一点硬盘空间，CheckState()函数不会被载入到内存，更不会被 JITed(译注：这里的 JITed 是指真正的编译为机器代码)，除非它被调用。它存在于程序集文件里并不是本质问题。这样的策略是增强(程序的)可伸缩性的，并且这样只是一点微不足道的性能开销。你可以通过查看.Net 框架库中 Debug 类而得到更深入的理解。在任何一台安装了.Net 框架库的机器上，System.dll 程序集包含了 Debug 类的所有方法的代码。由环境变量在编译时来决定是否让由调用者来调用它们。

你同样可以写一个方法，让它依懒于不只一个环境变量。当你应用多个环境变量来控制条件属性时，他们时以 or 的形式并列的。例如，下面这个版本的 CheckState 会在 DEBUG 或者 TRACE 为真时被调用：

```
[ Conditional("DEBUG"),  Conditional("TRACE") ]
private void CheckState()
```

如果要产生一个 and 的并列条件属性，你就要自己事先直接在代码里使用预处理命令定义一个标记：

```
#if (VAR1 && VAR2)
#define BOTH
#endif
```

是的，为了创建一个依懒于前面多个环境变量的条件例程(conditional routine)，你不得不退到开始时使用的#if 实践中了。#if 为我们产生一个新的标记，但避免在编译选项内添加任何可运行的代码。

Conditional 属性只能用在方法的实体上，另外，必须是一个返回类型为 void 的方法。你不能在方法内的某个代码块上使用 Conditional，也不能在一个有返回值的方法上使用 Conditional 属性。取而代之的是，你要细心构建一个条件方法，并在那些方法上废弃条件属性行为。你仍然要回顾一下那些具有条件属性的方法，看它是否对对象的状态具有副作用。但 Conditional 属性在安置这些问题上比#if/#endif 要好得多。在使用#if/#endif 块时，你很可能错误的移除了一个重要的方法调用或者一些配置。

前面的例子合用预先定义的 DEBUG 或者 TRACE 标记，但你可以用这个技巧，扩展到任何你想要的符号上。Conditional 属性可以由定义标记来灵活的控制。你可以在编译命令行上定义，也可以在系统环境变量里定义，或者从源代码的编译选择里定义。

使用 Conditional 属性可以比使用#if/#endif 生成更高效的 IL 代码。在专门针对函数时，它更有优势，它会强制你在条件代码上使用更好的结构。编译器使用 Conditional 属性来帮助你避免因使用#if/#endif 而产生的常见的错误。条件属性比起预处理，它为你区分条件代码提供了更好的支持。

原则 5：始终提供 ToString()

Always Provide ToString()

在.Net 世界里，用得最多的方法之一就是 System.Object.ToString()了。你应该为你所有的客户写一个"通情达理"的类(译注：这里是指这个类应该对用户友好)。要么，你就迫使所用类的用户，去使用类的属性并添加一些

合理的易读的说明。这个以字符串形式存在，关于你设计的类的说明，可以很容易的向你的用户显示一些关于对象的信息到：Windows Form 里，Web Form 里，控制台输出。这些字符说明可以用于调试。你写的任何一种类型，都应该合理的重写这个方法。当你设计更多的复杂的类型时，你应该实现应变能力更强的 IFormattable.ToString(). 承认这个：如果你不重写(override)这个常规的方法，或者只是写一个很糟糕的，你的客户将不得不为你修正它。

System.Object 版的 ToString()方法只返回类型的名字。这并没有太多有用的信息：“Rect”,“Point”,“Size”并不会如你所想的那样显示给你的用户。但那只是在你没有为你的类重写 ToString()方法时得到的。你只用为你的类写一次，但你的客户却会使用很多次。当你设计一个类时，多添加一点小小的工作，就可以在你或者是其他人每次使用时得到回报。

让我们来考虑一个简单的需求：重写 System.Object.ToString()方法。你所设计的每一个类型都应该重写 ToString()方法，用来为你的类型提供一些最常用的文字说明。考虑这个 Customer 类以及它的三个成员(fields)(译注：一般情况，类里的 fields 译为成员，这是面向对象设计时的概念，而在与数据库相关的地方，则是指字段)：

```
public class Customer
{
    private string _name;
    private decimal _revenue;
    private string _contactPhone;
}
```

默认继承自 System.Object 的 ToString()方法会返回"Customer"。这对每个人都不会有太大的帮助。就算 ToString()只是为了在调试时使用，也应该更灵活(sophisticated)一些。你重写的 ToString()方法应该返回文字说明，更像是你的用户在使用这个类一样。在 Customer 例子中，这应该是名字：

```
public override string ToString()
{
    return _name;
}
```

如果你不遵守这一原则里的其它意见，就按照上面的方法为你所定义的所有类型重写该方法。它会直接为每个人省下时间。

当你负责任的为 Object.ToString()方法实现了重写时，这个类的对象可以更容易的被添加到 Windows Form 里，Web Form 里，或者打印输出。 .NET 的 FCL 使用重载的 Object.ToString()在控件中显示对象：组合框，列表框，文本框，以及其它一些控件。如果你一个 Windows Form 或者 Web Form 里添加一个 Customer 对象的链表，你将会得到它们的名字(以文本)显示出来(译注：而不是每个对象都是同样的类型名)。

Syste.Console.WriteLine()和 System.String.Formate()在内部(实现的方法)是一样的。任何时候，.Net 的 FCL 想取得一个 customer 的字符串说明时，你的 customer 类型会提供一个客户的名字。一个只有三行的简单函数，完成了所有的基本需求。

这是一个简单的方法，ToString()还可以以文字(输出的方法)满足很多用户自定义类型的需求。但有些时候，你的要求可能会更多。前面的 customer 类型有三个成员：名字，收入和联系电话。对 System.Object.ToString()(译注：原文这里有误，掉了 Object)的重写只使用了_name。你可以通过实现 IFormattable(这个接口)来弥补这个不足。这是一个当你需要对外输出格式化文本时使用的接口。IFormattable 包含一个重载版的 ToString()方法，使用这个方法，你可以为你的类型信息指定详细的格式。这也是一个当你要产生并输出多种格式的字符串时要使用的接口。customer 类就是这种情况，用户将希望产生一个报表，这个报表包含了已经表格化了的用户名和去年的收入。IFormattable.ToString()方法正合你意，它可以让用户格式化输出你的类型信息。这个方法原型的参数上一包含一个格式化字符串和一个格式化引擎：

```
string System.IFormattable.ToString(string format, IFormatProvider formatProvider)
```

你可以为你设计的类型指定要使用的格式字符串。你也可以为你的格式字符串指定关键字符。在这个 customer 的例子中，你可以完全可以用 n 来表示名字，r 表示收入以及 p 来表示电话。这样一来，你的用户就可以随意的组合指定信息，而你则须要为你的类型提供下面这个版本的的 IFormattable.ToString():

```
#region IFormattable Members
// supported formats:
// substitute n for name.
// substitute r for revenue
// substitute p for contact phone.
// Combos are supported:   nr, np, npr, etc
// "G" is general.
string System.IFormattable.ToString(string format,
  IFormatProvider formatProvider)
{
    if (formatProvider != null)
    {
        ICustomFormatter fmt = formatProvider.GetFormat(this.GetType()) as
ICustomFormatter;
        if (fmt != null)
            return fmt.Format(format, this, formatProvider);
    }
    switch (format)
    {
        case "r":
            return _revenue.ToString();
        case "p":
            return _contactPhone;
        case "nr":
            return string.Format("{0,20}, {1,10:C}", _name, _revenue);
        case "np":
            return string.Format("{0,20}, {1,15}", _name, _contactPhone);
        case "pr":
            return string.Format("{0,15}, {1,10:C}", _contactPhone, _revenue);
        case "pn":
            return string.Format("{0,15}, {1,20}", _contactPhone, _name);
        case "rn":
            return string.Format("{0,10:C}, {1,20}", _revenue, _name);
        case "rp":
            return string.Format("{0,10:C}, {1,20}", _revenue, _contactPhone);
        case "nrp":
            return string.Format("{0,20}, {1,10:C}, {2,15}", _name, _revenue,
_contactPhone);
        case "npr":
            return string.Format("{0,20}, {1,15}, {2,10:C}", _name, _contactPhone,
_revenue);
        case "pnr":
            return string.Format("{0,15}, {1,20}, {2,10:C}", _contactPhone, _name,
_revenue);
        case "prn":
```

```
                return string.Format("{0,15}, {1,10:C}, {2,15}", _contactPhone, _revenue,
    _name);
            case "rpn":
                return string.Format("{0,10:C}, {1,15}, {2,20}", _revenue, _contactPhone,
    _name);
            case "rnp":
                return string.Format("{0,10:C}, {1,20}, {2,15}", _revenue, _name,
    _contactPhone);
            case "n":
            case "G":
            default:
                return _name;
        }
    }
    #endregion
```

(译注：上面的做法显然不合理，要是我的对象有 10 个成员，这样的组合是会让人疯掉的。推荐使用正则表达式来完成这样的工作，正则表达式在处理文字时的表现还是很出色的。)

添加了这样的函数后，你就让用户具有了可以这样指定 customer 数据的能力：

```
IFormattable c1 = new Customer();
Console.WriteLine("Customer record: {0}", c1.ToString("nrp", null));
```

任何对 IFormattable.ToString()的实现都要指明类型，但不管你在什么时候实现 IFormattation 接口，你都要注意处理大小写。首先，你必须支持能用格式化字符：“G”。其次，你必须支持两个空格式化字符：""和 null。当你重载 Object.ToString()这个方法时，这三个格式化字符应该返回同样的字符串。.Net 的 FCL 经常用 null 来调用 IFormattable.ToString()方法，来取代对 Object.ToString()的调用，但在少数地方使用格式符"G"来格式化字符串，从而区别通用的格式。如果你添加了对 IFormattable 接口的支持，并不再支持标准的格式化，你将会破坏 FCL 里的字符串的自动(隐式)转换。

IFormattable.ToString()的第二个参数是一个实现了 IFormatProvider 接口的对象。这个对象为用户提供了一些你没有预先设置的格式化选项(译注：简单一点，就是你可以只实现你自己的格式化选项，其它的默认由它来完成)。如果你查看一下前面 IFormattable.ToString()的实现，你就会毫不犹豫的拿出不计其数的，任何你喜欢的格式化选项，而这些都是的格式化中所没有的。支持人们容易阅读的输出是很自然的事，但不管你支持多少种格式，你的用户总有一天会想要你预先没想到的格式。这就为什么这个方法的前几行要检察实现了 IFormatProvider 的对象，并把 ICustomFormatter 的工作委托给它了。

让我们把(讨论的)焦点从类的作者转移到类的使用者上来。你发现你想要的格式化不被支持。例如，你有一个一组客户，他们的名字有的大于 20 个字符，并且你想修改格式化选项，让它支持 50 个字符长的客户名。这就是为什么 IFormatProvider 接口要存在。你可以设计一个实现了 IFormatProvider 的类，并且让它同时实现 ICustomFormatter 接口用于格式化输出。IFormatProvider 接口定义了一个方法：GetFormat()。这个方法返回一个实现了 ICustomFormatter 接口的对象。由 ICustomFormatter 接口的指定方法来完成实际的格式化工作。下面这一对(接口)实现了对输出的修改，让它可以支持 50 个字符长的用户名：

```
// Example IFormatProvider:
public class CustomFormatter : IFormatProvider
{
    #region IFormatProvider Members
    // IFormatProvider contains one method.
    // This method returns an object that
    // formats using the requested interface.
```

```
        // Typically, only the ICustomFormatter
        // is implemented
        public object GetFormat(Type formatType)
        {
            if (formatType == typeof(ICustomFormatter))
                return new CustomerFormatProvider();
            return null;
        }
        #endregion
        // Nested class to provide the
        // custom formatting for the Customer class.
        private class CustomerFormatProvider : ICustomFormatter
        {
            #region ICustomFormatter Members
            public string Format(string format, object arg, IFormatProvider formatProvider)
            {
                Customer c = arg as Customer;
                if (c == null)
                    return arg.ToString();
                return string.Format("{0,50}, {1,15}, {2,10:C}", c.Name, c.ContactPhone,
    c.Revenue);
            }
            #endregion
        }
    }
```

GetFormat()方法取得一个实现了 ICustomFormatter 接口的对象。而 ICustomFormatter.Format()方法，则根据用户需求负责实际的格式化输出工作。这个方法把对象转换成格式化的字符串。你可以为 ICustomFormatter.Format()定义格式化字符串，因此你可以按常规指定多重格式。FormatProvider 就是一个由 GetFormat()方法取得的 IFormatProvider 对象。

为了满足用户的格式化要求，你必须用 IFormatProvider 对象明确的调用 string.Format()方法:

```
Console.WriteLine(string.Format(new CustomFormatter(), "", c1));
```

你可以设计一个类，让它实现 IFormatProvider 和 ICustomFormatter 接口，再实现或者不实现 IFormattable 接口。因此，即使这个类的作者没有提供合理的 ToStrying 行为，你可以自己来完成。当然，从类的外面来实现，你只能访问公共属性成数据来取得字符串。实现两个接口，IFormatProvider 和 IcustomFormatter，只做一些文字输出，并不需要很多工作。但在.Net 框架里，你所实现的指定的文字输出在哪里都可以得到很好的支持。

所以，再回到类的作者上来。重写 Object.ToString()，为你的类提供一些说明是件很简单的事。你每次都应该为你的类型提供这样的支持。而且这应该是对你的类型最显而易见的，最常用的说明。在一些极端情况下，你的格式化不能支持一些过于灵活的输出时，你应该借用 IFormattable 接口的优势。它为你的类型进行自定义格式化输出提供了标准方法。如果你放弃这些，你的用户将失去用于实现自定义格式化的工具。这些解决办法须要写更多的代码，并且因为你的用户是在类的外面的，所以他们无法检查类的里面的状态。

最后，大家注意到你的类型的信息，他们会明白输出的文字。尽可能以简单的方式的提供这样的信息吧：为你的所有类型重写 ToString()方法。

原则 6：区别值类型数据和引用类型数据

Distinguish Between Value Types and Reference Types

值类型数据还是引用类型数据？结构还是类？什么你须要使用它们呢？这不是 C＋＋，你可以把所有类型都定义为值类型，并为它们做一个引用。这也不是 Java，所有的类型都是值类型。你在创建每个类型实例时，你必须决定它们以什么样的形式存在。这是一个为了取得正确结果，必须在一开始就要面对的重要决定。（一但做也决定）你就必须一直面对这个决定给你带来的后果，因为想在后面再对它进行改动，你就不得不在很多细小的地方强行添加很多代码。当你设计一个类型时，选择 struct 或者 class 是件简单的小事情，但是，一但你的类型发生了改变，对所有使用了该类型的用户进行更新却要付出(比设计时)多得多的工作。

这不是一个简单的非此及彼的选择。正确的选择取决于你希望你的新类型该如何使用。值类型不具备多态性，但它们在你的应用程序对数据的存取却是性能有佳；引用类型可以有多态性，并且你还可以在你的应用程序中为它们定义一些表现行为。考虑你期望给你的类型设计什么样的职能，并根据这些职能来决定设计什么样的类型。结构存储数据，而类表现行为。

因为很多的常见问题在 C++以及 Javaj 里存在，因此.Net 和 C#对值类型和引用类型的做了区分。在 C++里，所有的参数和返回值都是以值类型的进行传递的。以值类型进行传递是件很有效率的事，但不得不承受这样的问题：对象的浅拷贝(partial copying)(有时也称为 slicing object)。如果你对一个派生的对象 COPY 数据时，是以基类的形式进行 COPY 的，那么只有基类的部分数据进行了 COPY。你就直接丢失了派生对象的所有信息。即使时使用基类的虚函数。

而 Java 语言呢，在放弃了值类型数据后，或多或少有些表现吧。Java 里，所有的用户定义类型都是引用类型，所有的参数及返回数据都是以引用类型进行传递的。这一策略在(数据)一致性上有它的优势，但在性能上却有缺陷。让我们面对这样的情况，有些类型不是多态性的--它们并不须要。Java 的程序员们为所有的变量准备了一个内存堆分配器和一个最终的垃圾回收器。他们还须要为每个引用变量的访问花上额外的时间，因为所有的变量都是引用类型。在 C#里，你或者用 struct 声明一个值类型数据，或者用 class 声明一个引用类型数据。值类型数据应该比较小，是轻量级的。引用类型是从你的类继承来的。这一节将练习用不同的方法来使用一个数据类型，以便你给掌握值类型数据和引用类型数据之间的区别。

我们开始了，这有一个从一个方法上返回的类型：

```
private MyData _myData;
public MyData Foo()
{
 return _myData;
}
// call it:
MyData v = Foo();
TotalSum += v.Value;
```

如果 MyData 是一个值类型，那么回返值会被 COPY 到 V 中存起来。而且 v 是在栈内存上的。然而，如果 MyData 是一个引用类型，你就已经把一个引用导入到了一个内部变量上。同时，

你也违犯了封装原则(见原则 23)。

或者，考虑这个变量：

```
private MyData _myData;
public MyData Foo()
{
 return _myData.Clone() as MyData;
}
// call it:
MyData v = Foo();
```

```
TotalSum += v.Value;
```

现在，v 是原始数据_myData 的一个 COPY。做为一个引用类型，两个对象都是在内存堆上创建的。你不会因为暴露内部数据而遇到麻烦。取而代之的是你会在堆上建立了一个额外的数据对象。如果 v 是局部变量，它很快会成为垃圾，而且 Clone 要求你在运行时做类型检测。总而言之，这是低效的。

以公共方法或属性暴露出去的数据应该是值类型的。但这并不是说所有从公共成员返回的类型必须是值类型的。对前面的代码段做一个假设，MyData 有数据存在，它的责任就是保存这些数据。

但是，可以考虑选择下面的代码段：

```
private MyType _myType;
public IMyInterface Foo()
{
 return _myType as IMyInterface;
}
// call it:
IMyInterface iMe = Foo();
iMe.DoWork();
```

变量_myType 还是从 Foo 方法返回。但这次不同的是，取而代之的是访问返回值的内部数据，通过调用一个定义好了的接口上的方法来访问对象。你正在访问一个 MyType 的对象，而不是它的具体数据，只是使用它的行为。该行为是 IMyInterface 展示给我们的，同时，这个接口是可以被其它很多类型所实现的。做为这个例子，MyType 应该是一个引用类型，而不是一个值类型。MyType 的责任是考虑它周围的行为，而不是它的数据成员。

这段简单的代码开始告诉你它们的区别：值类型存储数据，引用类型表现行为。现在我们深入的看一下这些类型在内存里是如何存储的，以及在存储模型上表现的性能。考虑下面这个类：

```
public class C
{
  private MyType _a = new MyType();
  private MyType _b = new MyType();
  // Remaining implementation removed.
}
C var = new C();
```

多少个对象被创建了？它们占用多少内存？这还不好说。如果 MyType 是值类型，那么你只做了一次堆内存分配。大小正好是 MyType 大小的 2 倍。然而，如果 MyType 是引用类型，那么你就做了三次堆内存分配：一次是为 C 对象，占 8 字节(假设你用的是 32 位的指针)(译注：应该是 4 字节，可能是笔误)，另 2 次是为包含在 C 对象内的 MyType 对象分配堆内存。之所以有这样不同的结果是因为值类型是以内联的方式存在于一个对象内，相反，引用类型就不是。每一个引用类型只保留一个引用指针，而数据存储还须要另外的空间。

为了理解这一点，考虑下面这个内存分配：

```
MyType [] var = new MyType[ 100 ];
```

如果 MyType 是一个值类型数据，一次就分配出 100 个 MyType 的空间。然而，如果 MyType 是引用类型，就只有一次内存分配。每一个数据元素都是 null。当你初始化数组里的每一个元素时，你要上演 101 次分配工作--并且这 101 次内存分配比 1 次分配占用更多的时间。分配大量的引用类型数据会使堆内存出现碎片，从而降低程序性能。如果你创建的类型意图存储数据的值，那么值类型是你要选择的。

采用值类型数据还是引用类型数据是一个很重要的决定。把一个值类型数据转变为类是一个深层次的改变。考虑下面这种情况：

```
public struct Employee
{
  private string  _name;
  private int     _ID;
```

```
  private decimal _salary;
  // Properties elided
  public void Pay(BankAccount b)
  {
    b.Balance += _salary;
  }
}
```

这是个很清楚的例子，这个类型包含一个方法，你可以用它为你的雇员付薪水。时间流逝，你的系统也公正的在运行。接着，你决定为不同的雇员分等级了：销售人员取得拥金，经理取得红利。你决定把这个 Employee 类型改为一个类：

```
public class Employee
{
  private string  _name;
  private int     _ID;
  private decimal _salary;
  // Properties elided
  public virtual void Pay(BankAccount b)
  {
    b.Balance += _salary;
  }
}
```

这扰乱了很多已经存在并使用了你设计的结构的代码。返回值类型的变为返回引用类型。参数也由原来的值传递变为现在的引用传递。下面代码段的行为将受到重创：

```
Employee e1 = Employees.Find("CEO");
e1.Salary += Bonus; // Add one time bonus.
e1.Pay(CEOBankAccount);
```

就是这个一次性的在工资中添加红利的操作，成了持续的提升。曾经是值类型 COPY 的地方，如今都变成了引用类型的引用。编译器很乐意为你做这样的改变，你的 CEO 更是乐意这样的改变。另一方面，你的 CEO 将会给你报告 BUG。

你还是没能改变对值类型和引用类型的看法，以至于你犯下这样的错误还不知道：它改变了行为!

出现这个问题的原因就是因为 Employee 已经不再遵守值类型数据的的原则。

另外，定义为 Empolyee 的保存数据的元素，在这个例子里你必须为它添加一个职责：为雇员付工资。职责是属于类范围内的事。类可以被定义多态的，从而很容易的实现一些常见的职责；而结构则不充许，它应该仅限于保存数据。

在值类型和引用类型间做选择时，.Net 的说明文档建议你把类型的大小做为一个决定因素来考虑。而实际上，更多的因素是类型的使用。简单的结构或单纯的数据载体是值类型数据优秀的候选对象。事实表明，值类型数据在内存管理上有很好的性能：它们很少会有堆内存碎片，很少会有垃圾产生，并且很少间接访问。

(译注：这里的垃圾，以及前面提到过的垃圾，是指堆内存上"死"掉的对象，用户无法访问，只等着由垃圾回收器来收集的对象，因此认为是垃圾。在.net 里，一般说垃圾时，都是指这些对象。建议看一下.net 下垃圾回收器的管理模型)

更重要是：当从一个方法或者属性上返回时，值类型是 COPY 的数据。这不会有因为暴露内部结构而存在的危险。But you pay in terms of features. 值类型在面向对象技术上的支持是有限的。你应该把所有的值类型当成是封闭的。你可以建立一个实现了接口的值类型，但这须要装箱，原则 17 会给你解释这会带来性能方面的损失。把值类型就当成是一个数据的容器吧，不再感觉是 OO 里的对象。

你创建的引用类型可能比值类型要多。如果你对下面所有问题回答 YES，你应该创建值类型数据。把下面的问

题与前面的 Employee 例子做对比：

1、类型的最基本的职责是存储数据吗？

2、它的属性上有定义完整的公共接口来访问或者修改数据成员吗？

3、我对类型决不会有子类自信吗？

4、我对类型决不会有多太性自信吗？

把值类型当成一个低层次的数据存储类型，把应用程序的行为用引用类型来表现。

你会在从类暴露的方法那取得安全数据的 COPY。你会从使用内联的值类型那里得到内存使用高率的好处。并且你可以用标准的面向对象技术创建应用程序逻辑。当你对期望的使用拿不准时，使用引用类型。

==========

小结：这一原则有点长，花的时间也比较多一点，本想下班后，两三个小时就搞定的，因为我昨天已经翻译了一些的，结果，还是一不小心搞到了 11 点。

最后说明一个，这一原则还是没有说明白什么是引用类型什么是值类型。当然，用 class 说明的类型一定是引用类型，用 struct 说明的是值类型。还要注意其它一些类型的性质：例如：枚举是什么类型？委托是什么类型？事件呢？

原则 7：选择恒定的原子值类型数据

Prefer immutable automic value type

恒定类型(immutable types)其实很简单，就是一但它们被创建，它们(的值)就是固定的。如果你验证一些准备用于创建一个对象的参数，你知道它在验证状态从前面的观点上看。你不能修改一个对象的内部状态使之成为无效的。在一个对象被创建后，你必须自己小心翼翼的保护对象，否则你不得不做错误验证来禁止改变任何状态。恒定类型天生就具有线程完全性的特点：多访问者可同时访问相同的内容。如果内部状态不能修改，那么就不能给不同的线程提供查看不一致的数据视图的机会。恒定类型可以从你的类上安全的暴露出来。调用者不能修改对象的内部状态。恒定类型可以很好的在基于哈希代码的集合上工作。以 Object.GetHashCode()方法返回的值，对同一个实例是必须相同的(参见原则 10)，而这正是恒定类型总能成功的地方。

并不是所有的类型都能成为恒定类型的。如果它可以，你需要克隆一个对象用于修改任何程序的状态了。这就是为什么同时推荐使用恒定类型和原子类型数据了。把你的对象分解为自然的单一实体结构。一个 Address 类型就是的，它就是一个简单的事，由多个相关的字段组成。改变其中一个字段就很可能意味着修改了其它字段。一个客户类型不是一个原子类型，一个客户类型可能包含很多小的信息块：地址，名字，一个或者多个电话号码。任何一个互不关联的信息块都可以改变。一个客户可能会在不搬家的情况下改变电话号码。而另一个客户可能在搬了家的情况下保留原来的电话号码。还有可能，一个客户改变了他(她)的名字，而没有搬家也没有改电话号码。一个客户类型就不是原子类型；它是由多个不同的恒定的组成部份构成的：地址，名字，以及一个成对出现的电话号码集合。原子类型是单一实体：你很自然的用原子类型来取代实体内容。这一例外会改变它其中的一个组成字段。

下面就是一个典型的可变地址类的实现：

```
// Mutable Address structure.
public struct Address
{
  private string  _line1;
  private string _line2;
  private string  _city;
  private string _state;
  private int    _zipCode;
  // Rely on the default system-generated
  // constructor.
  public string Line1
```

```
  {
    get { return _line1; }
    set { _line1 = value; }
  }
  public string Line2
  {
    get { return _line2; }
    set { _line2 = value; }
  }
  public string City
  {
    get { return _city; }
    set { _city= value; }
  }
  public string State
  {
    get { return _state; }
    set
    {
      ValidateState(value);
      _state = value;
    }
  }
  public int ZipCode
  {
    get { return _zipCode; }
    set
    {
      ValidateZip(value);
      _zipCode = value;
    }
  }
  // other details omitted.
}
// Example usage:
Address a1 = new Address();
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111 ;
// Modify:
a1.City = "Ann Arbor"; // Zip, State invalid now.
a1.ZipCode = 48103; // State still invalid now.
a1.State = "MI"; // Now fine.
```

内部状态的改变意味着它很可能违反了对象的不变性，至少是临时的。当你改变了 City 这个字段后，你就使

a1 处于无效状态。城市的改变使得它与洲字段及以区码字段不再匹配。代码的有害性看上去还不足以致命，但这对于多线程程序来说只是一小部份。在城市变化以后，洲变化以前的任何内容转变，都会潜在的使另一个线程看到一份矛盾的数据视图。

Okay，所以你不准备去写多线程程序。你仍然处于困境当中。想象这样的问题，区代码是无效的，并且设置抛出了一个异常。你只是完成了一些你想做的事，可你却使系统处于一个无效的状态当中。为了修正这个问题，你须要在地址类里面添加一个相当大的内部验证码。这个验证码应该须要相当大的空间，并且很复杂。为了完全实现期望的安全性，当你修改多个字段时，你须要在你的代码块周围创建一个被动的数据 COPY。线程安全性可能要求添加一个明确的线程同步用于检测每一个属性访问器，包括 set 和 get。总而言之，这将是一个意义重大的行动--并且这很可能在你添加新功能时被过分的扩展。

取而代之，把 address 结构做为一个恒定类型。开始把所有的字段都改成只读的吧：

```
public struct Address
{
  private readonly string  _line1;
  private readonly string  _line2;
  private readonly string  _city;
  private readonly string  _state;
  private readonly int    _zipCode;
  // remaining details elided
}
```

你还要移除所有的属性设置功能：

```
public struct Address
{
  // ...
  public string Line1
  {
    get { return _line1; }
  }
  public string Line2
  {
    get { return _line2; }
  }
  public string City
  {
    get { return _city; }
  }
  public string State
  {
    get { return _state; }
  }
  public int ZipCode
  {
    get { return _zipCode; }
  }
}
```

现在，你就拥有了一个恒定类型。为了让它有效的工作，你必须添加一个构造函数来完全初始化 address 结构。

这个 address 结构只须要额外的添加一个构造函数，来验证每一个字段。一个拷贝构造函数不是必须的，因为赋值运算符还算高效。记住，默认的构造函数仍然是可访问的。这是一个默认所有字符串为 null，ZIP 代码为 0 的地址结构：

```
public struct Address
{
  private readonly string  _line1;
  private readonly string  _line2;
  private readonly string  _city;
  private readonly string  _state;
  private readonly int    _zipCode;
  public Address(string line1,
    string line2,
    string city,
    string state,
    int zipCode)
  {
    _line1 = line1;
    _line2 = line2;
    _city = city;
    _state = state;
    _zipCode = zipCode;
    ValidateState(state);
    ValidateZip(zipCode);
  }
  // etc.
}
```

在使用这个恒定数据类型时，要求直接用不同的调用来一顺的修改它的状态。你更宁愿创建一个新的对象而不是去修改某个实例：

```
// Create an address:
Address a1 = new Address("111 S. Main",
  "", "Anytown", "IL", 61111);
// To change, re-initialize:
a1 = new Address(a1.Line1,
  a1.Line2, "Ann Arbor", "MI", 48103);
```

a1 的值是两者之一：它的原始位置 Anytown，或者是后来更新后的位置 Ann Arbor。你再不用像前面的例子那样，为了修改已经存在的地址而使对象产生临时无效状态。这里只有一些在构造函数执行时才存在的临时状态，而在构造函数外是无法访问内部状态的。很快，一个新的地址对象很快就产生了，它的值就一直固定了。这正是期望的安全性：a1 要么是默认的原始值，要么是新的值。如果在构造对象时发生了异常，那么 a1 保持原来的默认值不变。

(译注：为什么在构造时发生异常不会影响 a1 的值呢？因为只要构造函数没有正确返回，a1 都只保持原来的值。因为是那是一个赋值语句。这也就是为什么要用构造函数来实现对象更新，而不是另外添加一个函数来更新对象，因为就算用一个函数来更新对象，也有可能更新到一半时，发生异常，也会使得对象处于不正确的状态当中。大家可以参考一下.Net 里的日期时间结构，它就是一个典型的恒定常量例子。它没有提供任何的对单独年，月，日或者星期进行修改的方法。因为单独修改其中一个，可能导致整个日期处于不正确的状态：例如你把日期单独的修改为 31 号，但很可能那个月没有 31 号，而且星期也可能不同。它同样也是没提供任何方法来同时设置所以参数，

读了条原则后就明白为什么了吧。参考一下 DateTime 结构，可以更好的理解为什么要使用恒定类型。注：有些书把 immutable type 译为不变类型。)

为了创建一个恒定类型，你须要确保你的用户没有任何机会来修改内部状态。值类型不支持派生类，所以你不必定义担心派生类来修改它的内部状态。但你须要注意任何在恒定类型内的可变的引用类型字段。当你为这些类型实现了构造函数后，你须要被动的把可变的引用类型 COPY 一遍(译注：被动 COPY，defensive copy，文中应该是指为了保护数据，在数据赋值时不得不进行的一个 COPY，所以被认为是"防守"拷贝，我这里译为：被动拷贝，表示拷贝不是自发的，而是不得以而为之的)。

所有这些例子，都是假设 Phone 是一个恒定的值类型，因为我们只涉及到值类型的恒定性：

```
// Almost immutable: there are holes that would
// allow state changes.
public struct PhoneList
{
  private readonly Phone[] _phones;
  public PhoneList(Phone[] ph)
  {
    _phones = ph;
  }
  public IEnumerator Phones
  {
    get
    {
      return _phones.GetEnumerator();
    }
  }
}
Phone[] phones = new Phone[10];
// initialize phones
PhoneList pl = new PhoneList(phones);
// Modify the phone list:
// also modifies the internals of the (supposedly)
// immutable object.
phones[5] = Phone.GeneratePhoneNumber();
```

这个数组是一个引用类型。PhoneList 内部引用的数组，引用了分配在对象外的数组存储空间上。开发人员可以通过另一个引用到这个存储空间上的对象来修改你的恒定结构。为了避免这种可能，你须要对这个数组做一个被动拷贝。前面的例子显示了可变集合的弊端。如果电话类型是一个可变的引用类型，它还会有更多危害存在的可能。客户可以修改它在集合里的值，即使这个集合是保护，不让任何人修改。这个被动的拷贝应该在每个构造函数里被实现，而不管你的恒定类型里是否存在引用对象：

```
// Immutable: A copy is made at construction.
public struct PhoneList
{
  private readonly Phone[] _phones;
  public PhoneList(Phone[] ph)
  {
    _phones = new Phone[ ph.Length ];
    // Copies values because Phone is a value type.
```

```
        ph.CopyTo(_phones, 0);
    }
    public IEnumerator Phones
    {
      get
      {
        return _phones.GetEnumerator();
      }
    }
}
Phone[] phones = new Phone[10];
// initialize phones
PhoneList pl = new PhoneList(phones);
// Modify the phone list:
// Does not modify the copy in pl.
phones[5] = Phone.GeneratePhoneNumber();
```

当你返回一个可变类型的引用时，也应该遵守这一原则。如果你添加了一个属性用于从 PhoneList 结构中取得整个数组的链表，这个访问器也必须实现一个被动拷贝。详情参见原则 23。

这个复杂的类型表明了三个策略，这是你在初始化你的恒定对象时应该使用的。这个 Address 结构定义了一个构造函数，让你的客户可以初始化一个地址，定义合理的构造函数通常是最容易达到的。

你同样可以创建一个工厂方法来实现一个结构。工厂使得创建一个通用的值型数据变得更容易。.Net 框架的 Color 类型就是遵从这一策略来初始化系统颜色的。这个静态的方法 Color.FromKnownColor()和 Color.FromName()从当前显示的颜色中拷贝一个给定的系统颜色，返回给用户。

第三，你可以为那些需要多步操作才能完成构造函数的恒定类型添加一个伴随类。.Net 框架里的字符串类就遵从这一策略，它利用了伴随类 System.Text.StringBuilter。你是使用 StringBuliter 类经过多步操作来创建一个字符串。在完成了所有必须步骤生成一个字符串类后，你从 StringBuilter 取得了一个恒定的字符串。

(译注：.net 里的 string 是一但初始化，就不能再修改，对它的任何改动都会生成新的字符串。因此多次操作一个 string 会产生较多的垃圾内存碎片，你可以用 StringBuliter 来平衡这个问题。)

恒定类型是更简单，更容易维护的。不要盲目的为你的每一个对象的属性创建 get 和 set 访问器。你对这些类型的第一选择是把这些数存储为恒定类型，原子类型。从这些实体中，你可以可以容易的创建更多复杂的结构。

==========

小结：翻译了几篇原则，有些句子确实很难理解，自己也感觉翻译的七不像八不像的。如果读者遇到这样的一些不清楚的句子，可以跳过去，或者看原文。感觉实在是能力有限。

而且，对于书中的内容，我也并不是完全清楚，很多东西我自己也是在学习。所以添加的一些译注也不见得就是完全正确的。例如这一原则中的 DateTime 结构，它是不是一个恒定类型，我不敢确定，但从我读了这一原则后，加上我对 DataTime 以及这一原则的理解，觉得这个 DateTime 结构确实就是这一原则的实例。后面的原则我大概翻阅了一下，有的深有的浅，后期的翻译也会是有些艰难的，但不管怎样，我都会尽我最大的能力，尽快翻译完所有原则。

原则 8：确保 0 对于值类型数据是有效的

Ensure That 0 Is a Valid State for Value Types

.Net 系统默认所有的对象初始化时都为 0。这并没有提供一个方法来预防其他程序员创建的值类型数据的实例在初始化是都是 0。请让你的数据类型默认值也是 0。

一个特殊情况是在枚举类型数据中。决不要创建一个不包括 0 在内的枚举类型。所有的枚举类型都是从

System.ValueType 派生的。枚举类型的值是从 0 开始的，但你可以改变这一行为：

```
public enum Planet
{
  // Explicitly assign values.
  // Default starts at 0 otherwise.
  Mercury = 1,
  Venus = 2,
  Earth = 3,
  Mars = 4,
  Jupiter = 5,
  Saturn = 6,
  Neptune = 7,
  Uranus = 8,
  Pluto = 9
}
Planet sphere = new Planet();
```

sphere 此时的值就是 0，而这并不是一个有效的值。枚举类型的取值限制在所有列举的值中，任何依懒这一(普通)事实的代码都将无法工作。当你为你的枚举类型创建你自己的取值时，请确保 0 是当中的一个。如果你的枚举类型采用的是以位(bit)模式，把 0 定义为其它属性不存在时的取值。

按照现在的情况,你迫使用户必须精确的初始化值：

```
Planet sphere = Planet.Mars;
```

这将使包含(Planet)这一类型的其它类型很难创建：

```
public struct ObservationData
{
  Planet   _whichPlanet; //what am I looking at?
  Double  _magnitude; // perceived brightness.
}
```

创建一个新 ObservationData 实例的用户会创建一个不合法的 Planet 成员：

```
ObservationData d = new ObservationData();
```

最后创建的 ObservationData 的成员_magnitude 的值是 0，这是合理的。但_whichPlanet 却是无效的。你须要让 0 也是有效的(状态)。如果可能，选择把 0 做为一个最好的默认。Planet 枚举类型没有一个明确的默认值，无论用户是否任意的选择一些行星，这都不会给人留下好的感觉。当你陷入这样的情况时，使用 0 做为一个非初始化的值，这也是在后面可以更新的：

```
public enum Planet
{
  None = 0,
  Mercury = 1,
  Venus = 2,
  Earth = 3,
  Mars = 4,
  Jupiter = 5,
  Saturn = 6,
  Neptune = 7,
  Uranus = 8,
  Pluto = 9
```

```
}
Planet sphere = new Planet();
```

此时，sphere 具有一个(默认)值 None。为 Planet 枚举类型添加的这个非初始化的默认值，对 ObservationData 结构。最新创建的 ObservationData 对象的目标上具有 None 和一个数值 0。添加一个清晰的构造函数让用户为你的类型的所有字段明白的初始化：

```
public struct ObservationData
{
  Planet   _whichPlanet; //what am I looking at?
  Double  _magnitude; // perceived brightness.
  ObservationData(Planet target,
    Double mag)
  {
    _whichPlanet = target;
    _magnitude = mag;
  }
}
```

但请记住，默认的构造函数还是可访问的，而且是结构的部份。用户还是可以创建一个系统初始化的变量，而你无法阻止它。

在结束枚举类型转而讨论其它类型之前，你须要明白几个用于标记的特殊枚举类型规则。枚举类型在使用 Flags 特性时，必须把 None 的值设置为 0：

```
[Flags]
public enum Styles
{
  None = 0,
  Flat = 1,
  Sunken = 2,
  Raised = 4,
}
```

很多开发人员使用枚举标记和位运算操作 AND 进行运行，0 值会与位标记产生严重的问题。下面这个实验如果 Flat 的值是 0 时，是决不会成功的：

```
if ((flag & Styles.Flat) != 0) // Never true if Flat == 0.
  DoFlatThings();
```

如果你遇到 Flags，确保 0 对它来说是有效的，并且这就着：“对所有缺少的标记。”

另一个很常见的初始化问题就是值类型中包含了引用类型。字符串是一个常见的例子：

```
public struct LogMessage
{
  private int _ErrLevel;
  private string _msg;
}
LogMessage MyMessage = new LogMessage();
```

MyMessage 包含了一个_msg 为 null 的引用字段。这里没有办法强行使用另一个不同的初始化方法，但你利用属性来局部化这个问题。你创建一个属性向所用的用户暴露_Msg 的值。添加一个业务逻辑，使得当字符串为 null 引用是，用空 串来取而代之：

```
public struct LogMessage
```

```
{
  private int _ErrLevel;
  private string _msg;
  public string Message
  {
    get
    {
      return (_msg != null) ?
        _msg : string.Empty;
    }
    set
    {
      _msg = value;
    }
  }
}
```

(译注：我个人觉得这里违反了原则一。当对两个实例进行赋值 COPY 时，会出现，你明明使用了 a=b 的运行，但实际上 a!=b 的结果。可以参见原则 1。)

在你自己的数据类型内部，你应该添加这样的一个属性。做了这样的局部处理后，null 引用在某一位置做了验证。当调用是在你的程序集内时，Message 的访问器基本上是可以很好的内联的。你将会取得高效低错的代码。

系统为所有的值类型数据初始化为 0，而没有办法防止用户在创建一个值类型实例时，给所有的值类型都赋值为 0。如果可能，把 0 设置为自然的默认值。特殊情况下，使用 Flags 特性的枚举类型必须确保 0 是所有缺省标记的值。

原则 9：明白几个相等运算之间的关系

Understand the Relationships Among ReferenceEquals(), static Equals(), instance Equals(), and operator==

明白 ReferenceEquals(), static Equals(), instance Equals(), 和运算行符==之间的关系。

当你创建你自己的类型时(不管是类还是结构)，你要定义类型在什么情况下是相等的。C#提供了 4 个不同的方法来断定两个对象是否是相等的：

```
public static bool ReferenceEquals
  (object left, object right);
public static bool Equals
  (object left, object right);
public virtual bool Equals(object right);
public static bool operator==(MyClass left, MyClass right);
```

这种语言让你可以为上面所有的 4 种方法创建自己的版本。But just because you can doesn't mean that you should.你或许从来不用重新定义前面两个方法。你经常遇到的是创建你自己实例的 Equals()方法，来为你的类型定义语义；或者你偶而重载运==运算符，但这只是为了考虑值类型的性能。幸运的是，这 4 个方法的关系，当你改变其中一个时，会影响到其它的几个。是的，须要 4 个方法来完整的测试对象是否完全相等。但你不用担心，你可以简单的搞定它们。

和 C#里其它大多数复杂元素一样，这个(对相等的比较运算)也遵守这样的一个事实：C#充许你同时创建值类型和引用类型。两个引用类型的变量在引用同一个对象时，它们是相等的，就像引用到对象的 ID 一样。两个值类型的变量在它们的类型和内容都是相同时，它们应该是相等的。这就是为什么相等测试要这么多方法了。

我们先从两个你可能从来不会修改的方法开始。Object.ReferenceEquals()在两个变量引用到同一个对象时返回 true，也就是两个变量具有相同的对象 ID。不管比较的类型是引用类型还是值类型的，这个方法总是检测对象 ID，而不是对象内容。是的，这就是说当你测试两个值类型是否相等时，ReferenceEquals()总会返回 false，即使你是比较同一个值类型对象，它也会返回 false。这里有两个装箱，会在原则 16 中讨论。(译注：因为参数要求两个引用对象，所以用两个值类型来调用该方法，会先使两个参数都装箱，这样一来，两个引用 对象自然就不相等了。)

```
int i = 5;
int j = 5;
if (Object.ReferenceEquals(i, j))
  Console.WriteLine("Never happens.");
else
  Console.WriteLine("Always happens.");
if (Object.ReferenceEquals(i, i))
  Console.WriteLine("Never happens.");
else
  Console.WriteLine("Always happens.");
```

你或许决不会重新定义 Object.ReferenceEquals()，这是因为它已经确实实现了它自己的功能：检测两个变量的对象 ID(是否相同)。

第二个可能从来不会重新定义的方法是静态的 Object.Equals()。这个方法在你不清楚两个参数的运行类型时什么时，检测它们是否相等。记住：C#里 System.Object 是一切内容的最终基类。任何时候你在比较两个变量时，它们都是 System.Object 的实例。因此，在不知道它们的类型时，而等式的改变又是依懒于类型的，这个方法是怎样来比较两个变量是否相等的呢？答案很简单：这个方法把比较的职责委交给了其中一个正在比较的类型。静态的 Object.Equals()方法是像下面这样实现的：

```
public static bool Equals(object left, object right)
{
  // Check object identity
  if (left == right)
    return true;
  // both null references handled above
  if ((left == null) || (right == null))
    return false;
  return left.Equals (right);
}
```

这个示例代码展示的两个方法是我还没有讨论的：操作符==()和实例的 Equals()方法。我会详细的解释这两个，但我还没有准备结束对静态的 Equals()的讨论。现在，我希望你明白，静态的 Equals()是使用左边参数实例的 Equals()方法来断定两个对象是否相等。

与 ReferenceEquals()一样，你或许从来不会重新定义静态的 Object.Equals()方法，因为它已经确实的完成了它应该完成的事：在你不知道两个对象的确切类型时断定它们是否是一样的。因为静态的 Equals()方法把比较委托给左边参数实例的 Equals()，它就是用这一原则来处理另一个类型的。

现在你应该明白为什么你从来不必重新定义静态的 ReferenceEquals()以及静态的 Equals()方法了吧。现在来讨论你须要重载的方法。但首先，让我们先来讨论一下这样的一个与相等相关的数学性质。你必须确保你重新定义的方法的实现要与其它程序员所期望的实现是一致的。这就是说你必须确保这样的一个数学相等性质：相等的自反性，对称性和传递性。自反性就是说一个对象是等于它自己的，不管对于什么类型，a==a 总应该返回 true；对称就是说，如果有 a==b 为真，那么 b==a 也必须为真；传递性就是说，如果 a==b 为真，且 b==c 也为真，那么 a==c 也必须为真，这就是传递性。

现在是时候来讨论实例的 Object.Equals()函数了，包括你应该在什么时候来重载它。当默认的行为与你的类

型不一致时，你应该创建你自己的实例版本。Object.Equals()方法使用对象的 ID 来断定两个变量是否相等。这个默认的 Object.Equals()函数的行为与 Object.ReferenceEquals()确实是一样的。但是请注意，值类型是不一样的。System.ValueType 并没有重载 Object.Equals()，记住，System.ValueType 是所有你所创建的值类型(使用关键字 struct 创建)的基类。两个值类型的变量相等，如果它们的类型和内容都是一样的。ValueType.Equals()实现了这一行为。不幸的是，ValueType.Equals()并不是一个高效的实现。ValueType.Equals()是所有值类型的基类(译注：这里是说这个方法在基类上进行比较)。为了提供正确的行为，它必须比较派生类的所有成员变量，而且是在不知道派生类的类型的情况下。在 C#里，这就意味着要使用反射。正如你将会在原则 44 里看到的，对反射而言它们有太多的不利之处，特别是在以性能为目标的时候。

相等是在应用中经常调用的基础结构之一，因此性能应该是值得考虑的目标。在大多数情况下，你可以为你的任何值类型重载一个快得多的 Equals()。简单的推荐一下：在你创建一个值类型时，总是重载 ValueType.Equals()。

你应该重载实例的 Equals()函数，仅当你想改变一个引用类型所定义的(Equals()的)语义时。.Net 结构类库中大量的类是使用值类型的语义来代替引用类型的语义。两个字符中对象相等，如果它们包含相同的内容。两个 DataRowViewc 对象相等，如果它们引用到同一个 DataRow。关键就是，如果你的类型须要遵从值类型的语义(比较内容)而不是引用类型的语义(比较对象 ID)时，你应该自己重载实例的 Object.Equals()方法。

好了，现在你知道什么时候应该重载你自己的 Object.Equals()，你应该明白怎样来实现它。值类型的比较关系有很多装箱的实现，装箱在原则 17 中讨论。对于用户类型，你的实例方法须要遵从原先定义行为(译注：前面的数学相等性质)，从而避免你的用户在使用你的类时发生一些意想不到的行为。这有一个标准的模式：

```
public class Foo
{
  public override bool Equals(object right)
  {
    // check null:
    // the this pointer is never null in C# methods.
    if (right == null)
      return false;
    if (object.ReferenceEquals(this, right))
      return true;
    // Discussed below.
    if (this.GetType() != right.GetType())
      return false;
    // Compare this type's contents here:
    return CompareFooMembers(
      this, right as Foo);
  }
}
```

首先，Equals()决不应该抛出异常，这感觉不大好。两个变量要么相等，要么不等；没有其它失败的余地。直接为所有的失败返回 false，例如 null 引用或者错误参数。现在，让我们来深入的讨论这个方法的细节，这样你会明白为什么每个检测为什么会在那里，以及那些方法可以省略。第一个检测断定右边的对象是否为 null，这样的引用上没有方法检测，在 C#里，这决不可能为 null。在你调用任何一个引用到 null 的实例的方法之前，CLR 可能抛出异常。下一步的检测来断定两个对象的引用是否是一样的，检测对象 ID 就行了。这是一个高效的检测，并且相等的对象 ID 来保证相同的内容。

接下来的检测来断定两个对象是否是同样的数据类型。这个步骤是很重要的，首先，应该注意到它并不一定是 Foo 类型，它调用了 this.GetType()，这个实际的类型可能是从 Foo 类派生的。其次，这里的代码在比较前检测了对象的确切类型。这并不能充分保证你可以把右边的参数转化成当前的类型。这个测试会产生两个细微的 BUG。考虑下面这个简单继承层次关系的例子：

```
public class B
{
  public override bool Equals(object right)
  {
    // check null:
    if (right == null)
      return false;
    // Check reference equality:
    if (object.ReferenceEquals(this, right))
      return true;
    // Problems here, discussed below.
    B rightAsB = right as B;
    if (rightAsB == null)
      return false;
    return CompareBMembers(this, rightAsB);
  }
}
public class D : B
{
  // etc.
  public override bool Equals(object right)
  {
    // check null:
    if (right == null)
      return false;
    if (object.ReferenceEquals(this, right))
      return true;
    // Problems here.
    D rightAsD = right as D;
    if (rightAsD == null)
      return false;
    if (base.Equals(rightAsD) == false)
      return false;
    return CompareDMembers(this, rightAsD);
  }
}
//Test:
B baseObject = new B();
D derivedObject = new D();
// Comparison 1.
if (baseObject.Equals(derivedObject))
  Console.WriteLine("Equals");
else
  Console.WriteLine("Not Equal");
// Comparison 2.
```

```
if (derivedObject.Equals(baseObject))
  Console.WriteLine("Equals");
else
  Console.WriteLine("Not Equal");
```

在任何可能的情况下，你都希望要么看到两个 Equals 或者两个 Not Equal。因为一些错误，这并不是先前代码的情形。这里的第二个比较决不会返回 true。这里的基类，类型 B，决不可能转化为 D。然而，第一个比较可能返回 true。派生类，类型 D，可以隐式的转化为类型 B。如果右边参数以 B 类型展示的成员与左边参数以 B 类型展示的成员是同等的，B.Equals()就认为两个对象是相等的。你将破坏相等的对称性。这一架构被破坏是因为自动实现了在继承关系中隐式的上下转化。

当你这样写时，类型 D 被隐式的转化为 B 类型：

baseObject.Equals(derived)

如果 baseObject.Equals()在它自己所定义的成员相等时，就断定两个对象是相等的。另一方面，当你这样写时，类型 B 不能转化为 D 类型，

derivedObject.Equals(base)

B 对象不能转化为 D 对象，derivedObject.Equals()方法总是返回 false。如果你不确切的检测对象的类型，你可能一不小心就陷入这样的窘境，比较对象的顺序成为一个问题。

当你重载 Equals()时，这里还有另外一个可行的方法。你应该调用基类的 System.Object 或者 System.ValueType 的比较方法，除非基类没有实现它。前面的代码提供了一个示例。类型 D 调用基类，类型 B，定义的 Equals()方法，然而，类 B 没有调用 baseObject.Equals()。它调用了 Systme.Object 里定义的那个版本，就是当两个参数引用到同一个对象时它返回 true。这并不是你想要的，或者你是还没有在第一个类里的写你自己的方法。

原则是不管什么时候，在创建一个值类型时重载 Equals()方法，并且你不想让引用类型遵从默认引用类型的语义时也重载 Equals(),就像 System.Object 定义的那样。当你写你自己的 Equals()时，遵从要点里实现的内容。重载 Equals()就意味着你应该重写 GetHashCode()，详情参见原则 10。

解决了三个，最后一个：操作符==()，任何时候你创建一个值类型，重新定义操作符==()。原因和实例的 Equals()是完全一样的。默认的版本使用的是引用的比较来比较两个值类型。效率远不及你自己任意实现的一个，所以，你自己写。当你比较两个值类型时，遵从原则 17 里的建议来避免装箱。

注意，我并不是说不管你是否重载了实例的 Equals()，都还要必须重载操作符==()。我是说在你创建值类型时才重载操作符==()。.Net 框架里的类还是期望引用类型的==操作符还是保留引用类型的语义。

C#给了你 4 种方法来检测相等性，但你只须要考虑为其中两个提供你自己的方法。你决不应该重载静态的 Object.ReferenceEquals()和静态的 Object.Equals()，因为它们提供了正确的检测，忽略运行时类型。你应该为了更好的性能而总是为值类型实例提供重载的 Equals()方法和操作符==()。当你希望引用类型的相等与对象 ID 的相等不同时，你应该重载引用类型实例的 Equals()。简单，不是吗？

原则 10：明白 GetHashCode()的缺陷

Understand the Pitfalls of GetHashCode()

这是本书中唯一一个被一整个函数占用的原则，你应该避免写这样的函数。GetHashCode()仅在一种情况下使用：那就是对象被用于基于散列的集合的关键词，如经典的 HashTable 或者 Dictionary 容器。这很不错，由于在基类上实现的 GetHashCode()存在大量的问题。对于引用类型，它可以工作，但高效不高；对于值类型，基类的实现经常出错。这更糟糕。你自己完全可以写一个即高效又正确的 GetHashCode()。没有那个单一的函数比 GetHashCode()讨论的更多，且令人困惑。往下看，为你解释困惑。

如果你定义了一个类型，而且你决不准备把它用于某个容器的关键词，那就没什么事了。像窗体控件，网页控

件，或者数据库链接这样的类型是不怎像要做为某个任何的关键词的。在这些情况下，什么都不用做了。所有的引用类型都会得到一个正确的散列值，即使这样效率很糟糕。值类型应该是恒定的(参见原则 7)，这种情况下，默认的实现总是工作的，尽管这样的效率也是很糟糕的。在大多数情况下，你最好完全避免在类型的实例上使用 GetHashCode()。

然而，在某天你创建了一个要做为 HashTable 的关键词来使用的类型，那么你就须要重写你自己的 GetHashCode()的实现了。继续看，基于散列(算法)的集合用散列值来优化查找。每一个对象产生一个整型的散列值，而该对象就存储在基于这个散列值的"桶"中。为了查找某个对象，你通过它的散列值来找到这个(存储了实际对象的)"桶"。在.Net 里，每一对象都有一个散列值，它是由 System.Object.GetHashCode()断定的。任何对 GetHashCode()的重写都必须遵守下面的三个规则：

1、如果两个对象是相等的(由操作符==所定义)，那么它们必须产生相同的散列值。否则，无法通过散列值在容器中找到对象。

2、对于任意对象 A，A.GetHashCode()必须是实例不变的。不管在 A 上调用了什么方法，A.GetHashCode()必须总是返回同样的散列值。这就保证在某个"桶"中的对象始终是在这个"桶"中。

3、对于任意的输入，散列函数总是产生产生一个介于整型内的随机分布。这会让你在一个基于散列的容器取得好的效率。

为一个类型写一个正确且高效的散列函数来满足上面的三条，要对该类型有广泛的认识。System.Object 和 System.ValueType 的默认版本并不具备什么优势。这些版本必须为你的特殊类型提供默认的行为，而同时它们对这些特殊的类型又并不了解。Object.GetHashCode()是使用 System.Object 内在的成员来产生散列值。每个对象在产生时指定一个唯一的值来做为对象关键词，(这个值)以整型来存储。这些关键词从 1 开始，在每次有任何新的对象产生时逐渐增加。对象的 ID 字段在 System.Object 的构造函数进行设置，并且今后再也不能修改。Object.GetHashCode()就是把这个值当成给定对象的散列值来返回。

(译注：注意这里说的是默认的引用类型，其它情况就不是这样的了。)

现在我们根据上面的三个原则来验证 Object.GetHashCode()。如果两个对象是相等的，Object.GetHashCode()返回同样的散列值，除非你重写了操作符==。System.Object 这个版本的==操作符是检测对象的 ID。GetHashCode()返回对象内部的 ID 字段，它是好的。然而，如果你提供了自己的==版本，你就必须同时提供你自己版本的 GetHashCode()，从而保证遵守了前面说的第一条规则。相等的细节参见原则 9。

第二条规则已经遵守了：一个对象创建后，它的散列值是不能改变的。

第三条规则，对所有的输入，在整型内进行随机分布，这并没有被支持。这个数字序列并不是整型上的随机分布，除非你创建了大量的对象。Object.GetHashCode()所产生的散列值主要集中在尽可能小的整型范围内。

这就是说这个 Object.GetHashCode()是正确的，但并不高效。如果你在你定义的引用类型上创建一个散列表，这个默认从 System.Object 上继承的行为是工作的，但是比较慢。当你创建准备用于散列关键词的引用类型时，你应该为你的特殊类型重写 GetHashCode()，从而提供更好的在整型范围上随机分布的散列值。

在讲解如何重写你自己的 GetHashCode()之前，这一节来验证 ValueType.GetHashCode()是否也遵守上面的三条规则。System.ValueType 重写了 GetHashCode()，为所有的值类型提供默认的行为。这一版本从你所定义的类型的第一个字段上返回散列。考虑这个例子：

```
public struct MyStruct
{
  private string   _msg;
  private int      _id;
  private DateTime _epoch;
}
```

从 MyStruct 对象上返回的散列值是从该对象的_msg 成员上生成的。下面的代码片断总是返回 true:

```
MyStruct s = new MyStruct();
return s.GetHashCode() == s._msg.GetHashCode();
```

规则 1 表示，两个相等的对象(用操作符==定义的)必须返回相同的散列值。这一规则被大多数值类型遵守着，

但你可以破坏它， just as you could with for reference types. ValueType 的操作符==()与其它成员一起来比较结构的第一个字段。这是满足第一条规则的。只要在任何时候你所重写的==操作符用第一个字段，这将可以正常工作。任何不以第一个字段断定相等的结构将违反这一规则，从而破坏 GetHashCode().

第二条规则表示，散列值必须是实例不变的。这一规则只有当结构的第一个成员字段是恒定类型时才被遵守。如果第一个字段的值发生了改变，那么散列值也会发生改变。这就破坏了这规则。是的，如果你的结构的实例对象在它的生存期内改变了结构的第一个字段，那么 GetHashCode()就破坏了这一规则。这也就是另一个原因，你最好的原则就是把值类型设置为恒定类型(参见原则 7)。

第三个规则依懒于第一个字段以及是如何使用它的。如果你的第一个字段能保证产生一个在整型范围上的随机分布，并且第一个字段的分布能复盖结构的所有其它值，那么这个结构就很好的保证了一个均衡的分布(译注：就是说结构的第一个字段可以唯一的决定一个实例)。然而，如果第一个字段经常具有相同的值，那么这一规则也会被破坏。考虑对前面的结构做一个小的修改：

```
public struct MyStruct
{
  private DateTime _epoch;
  private string   _msg;
  private int      _id;
}
```

如果_epoch 字段设置的是当前日期(不包含时间)，所有在同一给定日期里创建的对象具有相同的散列值。这防碍了在所有散列值中进行均衡的分布。

概括一个默认的行为，Object.GetHashCode()可以正确的在引用类型上工作，尽管它不是必须保证一个高效的分布。(如果你有一个对 Object 的==操作符的重载，你会破坏 GetHashCode())。ValueType.GetHashCode()仅在你的结构的第一个字段是只读的时候才能正确工作。而当你的结构的第一个字段的值，复盖了它所能接受的输入的有意义的子集时，ValueType.GetHashCode()就可以保证一个高效的散列值。

如果你准备创建一个更好的散列值，你须要为你的类型建立一些约束。再次验证上面的三条规则，现在我们来实现一个可工作的 GetHashCode()。

首先，如果两个对象相等，就是由操作符==所定义的，它们必须返回同样的散列值。类型的任何承担散列值的属性或者数据值也必须参与相等比较。显然，这就意味着同样用于相等比较的的属性也用于散列值的生成。然而很可能所有与相等比较的属性，并不用于散列值的计算。System.ValueType 的默认行为就是这样的，也就是说它经常违反规则 3。同样的数据元素应该参同时参与两个运算(比较和散列)。

第二条规则就是 GetHashCode()返回的值必须是实例不变的。想象你已经了一个引用类型，Customer:

```
public class Customer
{
  private string _name;
  private decimal _revenue;
  public Customer(string name)
  {
    _name = name;
  }
  public string Name
  {
    get { return _name; }
    set { _name = value; }
  }
  public override int GetHashCode()
  {
```

```
    return _name.GetHashCode();
  }
}
```

假设你运行下面的代码片断：

```
Customer c1 = new Customer("Acme Products");
myHashMap.Add(c1, orders);
// Oops, the name is wrong:
c1.Name = "Acme Software";
```

c1 在某些地方会丢失散列映射。当你把 c1 放在映射中时，散列值是由字符串“Acme Products”来保证的。当你把客户的名字修改为“Acme Software”后，散列值也发生了改变。现在是由新的名字：“Acme Software”来保证的了。c1 存储在一个由“Acme Products”决定的“桶”内，而它不应该存在于由“Acme Software”决定的“桶”内。你将会在你自己的集合中丢失这个客户。丢失的原因就是散列值不是实例不变的。你在对象存储后，改变了正确的“桶”。

前面的情形只有当 Customer 是引用类型时才出现。而当是值类型时，会有不同的错误行为，而且同样是带来麻烦的。如果 Customer 是一个值类型，c1 的一个拷贝会存在在散列映射中。因为装箱与拆箱很会拷贝数据。这并不是很可靠，当你在把一个值类型对象添加到集合之后，你还可以修改值类型的成员数据。

唯一安置好规则 2 的方法就是，定义一个散列函数，它依懒于对象的一些不变的属性来返回散列值。System.Object 是通过不变的对象 ID 来遵守这一规则的。System.ValueType 希望你的类型的第一个字段不发生改变。除了把你的类型设计为恒定类型以外，你没有更好的方法了。当你准备定义一个在散列容器中当关键词使用的值类型时，它必须是一个恒定的类型。如果不遵守劝告，你的用户就可以把你的类型做为一个关键词在散列表中使用，进而找到一个方法破坏散列表。 更正 Customer 类，你可以修改它，使客户名成为一个恒定的：

```
public class Customer
{
  private readonly string _name;
  private decimal _revenue;
  public Customer(string name) :
    this (name, 0)
  {
  }
  public Customer(string name, decimal revenue)
  {
    _name = name;
    _revenue = revenue;
  }
  public string Name
  {
    get { return _name; }
  }
  // Change the name, returning a new object:
  public Customer ChangeName(string newName)
  {
    return new Customer(newName, _revenue);
  }
  public override int GetHashCode()
  {
```

```
    return _name.GetHashCode();
  }
}
```

使名字成为恒定的类型后，你要怎样才能修改一个客户对象的名字呢：

```
Customer c1 = new Customer("Acme Products");
myHashMap.Add(c1,orders);
// Oops, the name is wrong:
Customer c2 = c1.ChangeName("Acme Software");
Order o = myHashMap[ c1 ] as Order;
myHashMap.Remove(c1);
myHashMap.Add(c2, o);
```

你已经移除了原来的客户，修改了名字，然后添加一个新的客户对象到散列表中。这看上去比原来的更麻烦，但它可以正确工作。先前的版本充许程序员写一些不正确的代码。通过强制使用恒定属性来生成散列值后，你就增加了正确的行为。你的用户就不会出错了。是的，这个版本可以更好的工作。你迫使开发人员写更多的代码，但这是因为只有这样才能写正确的代码。请确保参与散列运算的数据成员是恒定的。

第三条规则是说，GetHashCode()应该对所有的输入随机的生成一个分布在整型范围内的值。这一需求依懒于你实际创建的类型。如果有一个神奇的公式存在，那它应该在 System.Object 中实现了，并且这一原则(译注：这里说的是全文这一原则)也将不存在了。一个通用而且成功的算法就是 XOR(异或)运算，对一个类型内的所有字段的散列再进行异或后返回。如果你的类型里包含一些持久字段，计算时应该排除它们。

GetHashCode()具有很特殊的要求：相等的对象必须产生相等的散列值，并且散列值必须是对象不变的，并且是均衡的高效分布。所有这些只有对恒定类型才能满足(译注：本文前面已经说过了，.Net 框架中的 System.Object.GetHashCode()其实并不满足均衡高效分布这一规则)。对于其它类型，就交给默认的行为吧，知道它的缺点就行了。

原则 11：选择 foreach 循环

Prefer foreach Loops

C#的 foreach 语句是从 do，while，或者 for 循环语句变化而来的，它相对要好一些，它可以为你的任何集合产生最好的迭代代码。它的定义依懒于.Net 框架里的集合接口，并且编译器会为实际的集合生成最好的代码。当你在集合上做迭代时，可用使用 foreach 来取代其它的循环结构。检查下面的三个循环：

```
int [] foo = new int[100];
// Loop 1:
foreach (int i in foo)
  Console.WriteLine(i.ToString());
// Loop 2:
for (int index = 0;  index < foo.Length;  index++)
  Console.WriteLine(foo[index].ToString());
// Loop 3:
int len = foo.Length;
for (int index = 0;  index < len;  index++)
  Console.WriteLine(foo[index].ToString());
```

对于当前的 C#编译器(版本 1.1 或者更高)而言，循环 1 是最好的。起码它的输入要少些，这会使你的个人开发效率提提升。(1.0 的 C#编译器对循环 1 而言要慢很多，所以对于那个版本循环 2 是最好的。) 循环 3，大多数 C 或者 C++程序员会认为它是最有效的，但它是最糟糕的。因为在循环外部取出了变量 Length 的值，从而阻碍了 JIT 编译器将边界检测从循环中移出。

C#代码是安全的托管代码里运行的。环境里的每一块内存，包括数据的索引，都是被监视的。稍微展开一下，循环 3 的代码实际很像这样的：

```
// Loop 3, as generated by compiler:
int len = foo.Length;
for (int index = 0;  index < len;  index++)
{
  if (index < foo.Length)
    Console.WriteLine(foo[index].ToString());
  else
    throw new IndexOutOfRangeException();
}
```

C#的 JIT 编译器跟你不一样，它试图帮你这样做了。你本想把 Length 属性提出到循环外面，却使得编译做了更多的事情，从而也降低了速度。CLR 要保证的内容之一就是：你不能写出让变量访问不属于它自己内存的代码。在访问每一个实际的集合时，运行时确保对每个集合的边界(不是 len 变量)做了检测。你把一个边界检测分成了两个。

你还是要为循环的每一次迭代做数组做索引检测，而且是两次。循环 1 和循环 2 要快一些的原因是因为，C#的 JIT 编译器可以验证数组的边界来确保安全。任何循环变量不是数据的长度时，边界检测就会在每一次迭代中发生。(译注：这里几次说到 JIT 编译器，它是指将 IL 代码编译成本地代码时的编译器，而不是指将 C#代码或者其它代码编译成 IL 代码时的编译器。其实我们可以用不安全选项来迫使 JIT 不做这样的检测，从而使运行速度提高。)

原始的 C#编译器之所以对 foreach 以及数组产生很慢的代码，是因为涉及到了装箱。装箱会在原则 17 中展开讨论。数组是安全的类型，现在的 foreach 可以为数组生成与其它集合不同的 IL 代码。对于数组的这个版本，它不再使用 IEnumerator 接口，就是这个接口须要装箱与拆箱。

```
IEnumerator it = foo.GetEnumerator();
while(it.MoveNext())
{
  int i = (int) it.Current; // box and unbox here.
  Console.WriteLine(i.ToString());
}
```

取而代之的是，foreach 语句为数组生成了这样的结构：

```
for (int index = 0;  index < foo.Length;  index++)
  Console.WriteLine(foo[index].ToString());
```

(译注：注意数组与集合的区别。数组是一次性分配的连续内存，集合是可以动态添加与修改的，一般用链表来实现。而对于 C#里所支持的锯齿数组，则是一种折衷的处理。)

foreach 总能保证最好的代码。你不用操心哪种结构的循环有更高的效率：foreach 和编译器为你代劳了。

如果你并不满足于高效，例如还要有语言的交互。这个世界上有些人(是的，正是他们在使用其它的编程语言)坚定不移的认为数组的索引是从 1 开始的，而不是 0。不管我们如何努力，我们也无法破除他们的这种习惯。.Net 开发组已经尝试过。为此你不得不在 C#这样写初始化代码，那就是数组从某个非 0 数值开始的。

```
// Create a single dimension array.
// Its range is [ 1 .. 5 ]
Array test = Array.CreateInstance(typeof(int),
new int[ ]{ 5 }, new int[ ]{ 1 });
```

这段代码应该足够让所有人感到畏惧了(译注：对我而言，确实有一点)。但有些人就是很顽固，无认你如何努力，他们会从 1 开始计数。很幸运，这是那些问题当中的一个，而你可以让编译器来"欺骗"。用 foreach 来对 test 数组进行迭代：

```
foreach(int j in test)
```

```
  Console.WriteLine (j);
```

foreach 语句知道如何检测数组的上下限，所以你应该这样做，而且这和 for 循环的速度是一样的，也不用管某人是采用那个做为下界。

对于多维数组，foreach 给了你同样的好处。假设你正在创建一个棋盘。你将会这样写两段代码：

```
private Square[,] _theBoard = new Square[ 8, 8 ];
// elsewhere in code:
for (int i = 0; i < _theBoard.GetLength(0); i++)
  for(int j = 0; j < _theBoard.GetLength(1); j++)
    _theBoard[ i, j ].PaintSquare();
```

取而代之的是，你可以这样简单的画这个棋盘：

```
foreach(Square sq in _theBoard)
  sq.PaintSquare();
```

(译注：本人不赞成这样的方法。它隐藏了数组的行与列的逻辑关系。循环是以行优先的，如果你要的不是这个顺序，那么这种循环并不好。)

foreach 语句生成恰当的代码来迭代数组里所有维数的数据。如果将来你要创建一个 3D 的棋盘，foreach 循环还是一样的工作，而另一个循环则要做这样的修改：

```
for (int i = 0; i < _theBoard.GetLength(0); i++)
  for(int j = 0; j < _theBoard.GetLength(1); j++)
    for(int k = 0; k < _theBoard.GetLength(2); k++)
      _theBoard[ i, j, k ].PaintSquare();
```

(译注：这样看上去虽然代码很多，但我觉得，只要是程序员都可以一眼看出这是个三维数组的循环，但是对于 foreach，我看没人一眼可以看出来它在做什么！个人理解。当然，这要看你怎样认识，这当然可以说是 foreach 的一个优点。)

事实上，foreach 循环还可以在每个维的下限不同的多维数组上工作(译注：也就是锯齿数组)。 我不想写这样的代码，即使是为了做例示。但当某人在某时写了这样的集合时，foreach 可以胜任。

foreach 也给了你很大的伸缩性，当某时你发现须要修改数组里底层的数据结构时，它可以尽可能多的保证代码不做修改。我们从一个简单的数组来讨论这个问题：

```
int [] foo = new int[100];
```

假设后来某些时候，你发现它不具备数组类(array class)的一些功能，而你又正好要这些功能。你可能简单把一个数组修改为 ArrayList:

```
// Set the initial size:
ArrayList foo = new ArrayList(100);
```

任何用 for 循环的代码被破坏：

```
int sum = 0;
for (int index = 0;
  // won't compile: ArrayList uses Count, not Length
  index < foo.Length;
  index++)
  // won't compile: foo[ index ] is object, not int.
  sum += foo[ index ];
```

然而，foreach 循环可以根据所操作的对象不同，而自动编译成不同的代码来转化恰当的类型。什么也不用改。还不只是对标准的数组可以这样，对于其它任何的集合类型也同样可以用 foreach.

如果你的集合支持.Net 环境下的规则，你的用户就可以用 foreach 来迭代你的数据类型。为了让 foreach 语句认为它是一个集合类型，一个类应该有多数属性中的一个：公开方法 GetEnumerator()的实现可以构成一个集合类。明确的实现 IEnumerable 接口可以产生一个集合类。实现 IEnumerator 接口也可以实现一个集合类。foreach

可以在任何一个上工作。

foreach 有一个好处就是关于资源管理。IEnumerable 接口包含一个方法：GetEnumerator()。foreach 语句是一个在可枚举的类型上生成下面的代码，优化过的：

```
IEnumerator it = foo.GetEnumerator() as IEnumerator;
using (IDisposable disp = it as IDisposable)
{
  while (it.MoveNext())
  {
    int elem = (int) it.Current;
    sum += elem;
  }
}
```

如果断定枚举器实现了 IDisposable 接口，编译器可以自动优化代码为 finally 块。但对你而言，明白这一点很重要，无论如何，foreach 生成了正确的代码。

foreach 是一个应用广泛的语句。它为数组的上下限自成正确的代码，迭代多维数组，强制转化为恰当的类型(使用最有效的结构)，还有，这是最重要的，生成最有效的循环结构。这是迭代集合最有效的方法。这样，你写出的代码更持久(译注：就是不会因为错误而改动太多的代码)，第一次写代码的时候更简洁。这对生产力是一个小的进步，随着时间的推移会累加起来。

第二章 .Net 资源管理

.NET Resource Management

一个简单的事实：.Net 应用程序是在一个托管的环境里运行的，这个环境和不同的设计器有很大的冲突，这就才有了 Effective C#。极大限度上的讨论这个环境的好处，须要把你对本地化环境的想法改变为.Net CLR。也就意味着要明白.Net 的垃圾回收器。在你明白这一章里所推荐的内容时，有必要对.Net 的内存管理环境有个大概的了解。那我们就开始大概的了解一下吧。

垃圾回收器(GC)为你控制托管内存。不像本地运行环境，你不用负责对内存泄漏，不定指针，未初始化指针，或者一个其它内存管理的服务问题。但垃圾回收器前不是一个神话：你一样要自己清理。你要对非托管资源负责，例如文件句柄，数据链接，GDI+对象，COM 对象，以及其它一些系统对象。

这有一个好消息：因为 GC 管理内存，明确的设计风格可以更容易的实现。循环引用，不管是简单关系还是复杂的网页对象，都非常容易。GC 的标记以及严谨的高效算法可以检测到这些关系，并且完全的删除不可达的网页对象。GC 是通过对从应用程序的根对象开始，通过树形结构的"漫游"来断定一个对象是否可达的，而不是强迫每个对象都保持一些引用跟踪，COM 就是这样的。DataSet 就是一个很好的例子，展示了这样的算法是如何简化并决定对象的所属关系的。DataSet 是一个 DataTable 的集合，而每一个 DataTable 又是 DataRow 的集合，每一个 DataRow 又是 DataItem 的集合，DataColum 定义了这些类型的关系。这里就有一些从 DataItem 到它的列的引用。而同时，DataTime 也同样有一个引用到它的容器上，也就是 DataRow。DataRow 包含引用到 DataTable，最后每个对象都包含一个引用到 DataSet。

(译注：作者这里是想说：你看，这么复杂的引用关系，GC 都可以轻松的搞定，你看 GC 是不是很强大？)

如果这还不够复杂，那可以创建一个 DataView，它提供对经过过滤后的数据表的顺序访问。这些都是由 DataViewManager 管理的。所有这些贯穿网页的引用构成了 DataSet。释放内存是 GC 的责任。因为.Net 框架的设计者让你不必释放这些对象，这些复杂的网页对象引用不会造成问题。没有必须关心这些网页对象的合适的释放顺序，这是 GC 的工作。GC 的设计结构可以简化这些问题，它可以识别这些网页对象就是垃圾。在应用程序结束了对 DataSet 的引用后，没有人可以引用到它的子对象了(译注：就是 DataSet 里的对象再也引用不到了)。因此，网页里还有没有对象循环引用 DataSet，DataTables 已经一点也不重要了，因为这些对象在应用程序都已经不能

被访问到了，它们是垃圾了。

垃圾回收器在它独立的线程上运行，用来从你的程序里移除不使用的内存。而且在每次运行时，它还会压缩托管堆。压缩堆就是把托管堆中活动的对象移到一起，这样就可以空出连续的内存。图 2.1 展示了两个没有进行垃圾回收时的内存快照。所有的空闲内存会在垃圾回收进行后连续起来。

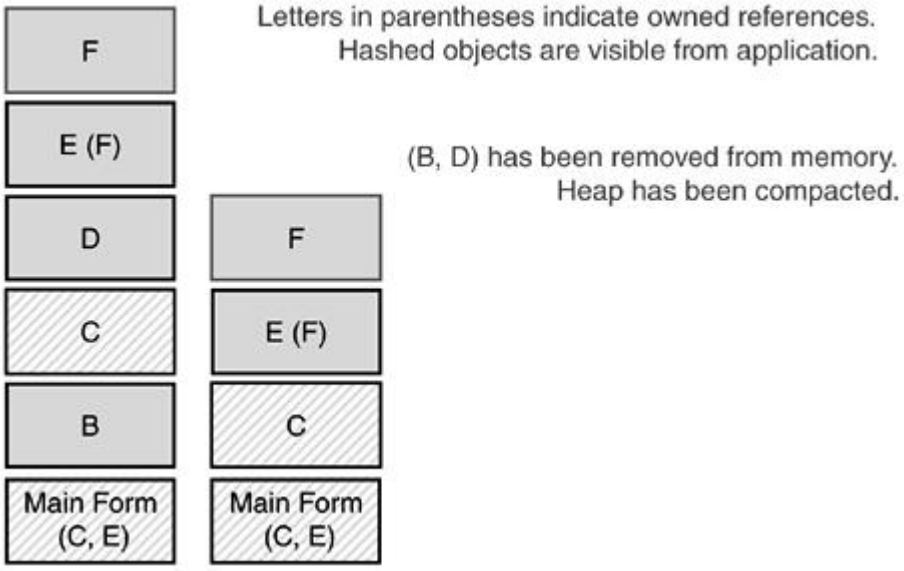


图 2.1 垃圾回收器不仅仅是移动不使用的内存，还移除动其它的对象，从而压缩使用的内存，让出最多的空闲内存。

正如你刚开始了解的，垃圾回收器的全部责任就是内存管理。但，所有的系统资源都是你自己负责的。你可以通过给自己的类型定义一个析构函数，来保证释放一些系统资源。析构函数是在垃圾回收器把对象从内存移除前，由系统调用的。你可以，也必须这样来释放任何你所占用的非托管资源。对象的析构函数有时是在对象成为垃圾之后调用的，但是在内存归还之前。这个非确定的析构函数意味着在你无法控制对象析构与停止使用之间的关系(译注：对象的析构与对象的无法引用是两个完全不同的概念。关于 GC，本人推荐读者参考一下 Jeffrey 的".Net 框架程序设计(修订版)"中讨论的垃圾回收器)。对 C++来说这是个重大的改变，并且这在设计上有一个重大的分歧。有经验的 C++程序员写的类总在构造函数内申请内存并且在析构函数中释放它们：

```
// Good C++, bad C#:
class CriticalSection
{
public:
  // Constructor acquires the system resource.
  CriticalSection()
  {
    EnterCriticalSection();
  }
  // Destructor releases system resource.
  ~CriticalSection()
  {
    ExitCriticalSection();
  }
};
// usage:
void Func()
```

```
{
  // The lifetime of s controls access to
  // the system resource.
  CriticalSection s;
  // Do work.
  //...
  // compiler generates call to destructor.
  // code exits critical section.
}
```

这是一种很常见的 C++风格，它保证资源无异常的释放。但这在 C#里不工作，至少，与这不同。明确的析构函数不是.Net 环境或者 C#的一部份。强行用 C++的风格在 C#里使用析构函数不会让它正常的工作。在 C#里，析构函数确实是正确的运行了，但它不是即时运行的。在前面那个例子里，代码最终在 critical section 上，但在 C#里，当析构函数存在时，它并不是在 critical section 上。它会在后面的某个未知时间上运行。你不知道是什么时候，你也无法知道是什么时候。

依懒于析构函数同样会导致性能上的损失。须要析构的对象在垃圾回收器上放置了一剂性能毒药。当 GC 发现某个对象是垃圾但是须要析构时，它还不能直接从内存上删除这个对象。首先，它要调用析构函数，但析构函数的调用不是在垃圾回收器的同一个线程上运行的。取而代之的是，GC 不得不把对象放置到析构队列中，让另一个线程让执行所有的析构函数。GC 继续它自己的工作，从内存上移除其它的垃圾。在下一个 GC 回收时，那些被析构了的对象才会再从内存上移除。图 2.2 展示了三个内存使用不同的 GC 情况。注意，那些须要析构的对象会待在内存里，直到下一次 GC 回收。

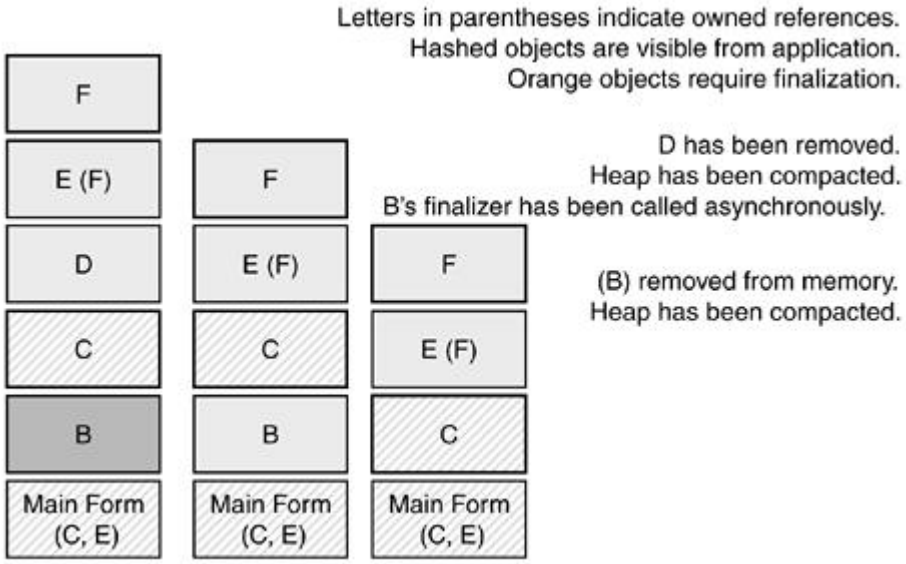


图 2.2 这个顺序展示了析构函数在垃圾回收器上起的作用。对象会在内存里存在的时间更长，须要启动另一个线程来运行垃圾回收器。

这用使你相信：那些须要析构的对象在内存至少多生存一个 GC 回收循环。但，我是简化了这些事。实际上，因为另一个 GC 的介入(译注：其实只有一个 GC，作者是想引用回收代的问题。)，使得情况比这复杂得多。.Net 回收器采用"代"来优化这个问题。代可以帮助 GC 来很快的标识那些看上去看是垃圾的对象。所以从上一次回后开始创建的对象称为第 0 代对象，所有那些经过一次 GC 回收后还存在的对象称为第 1 代对象。所有那些经过 2 次或者 2 次以上 GC 回收后还存在的对象称为第 2 代对象(译注：因为目前 GC 只支持 3 代对象，第 0 代到第 2 代，所以最多只有第 2 代对象，如果今后 GC 支持更多的代，那么会出现更代的对象，.Net 1.1 与 2.0 都只支持 3 代，这是 MS 证实比较合理的数字)。

分代的目的就是用来区分临时变量以及一些应用程序的全局变量。第 0 代对象很可能是临时的变量。成员变量，以及一些全局变量很快会成为第 1 代对象，最终成为第 2 代对象。

GC 通过限制检测第 1 以及第 2 代对象来优化它的工作。每个 GC 循环都检测第 0 代对象。粗略假设个 GC 会超过 10 次检测来检测第 0 代对象，而要超过 100 次来检测所有对象。再次考虑析构函数的开销：一个须要析构函数的对象可能要比一个不用析构函数的对象在内存里多待上 9 个 GC 回收循环。如果它还没有被析构，它将会移到第 2 代对象。在第 2 代对象中，一个可以生存上 100 个 GC 循环直到下一个第 2 代集合(译注：没理解，不知道说的什么)。

结束时，记得一个垃圾回收器负责内存管理的托管环境的最大好处：内存泄漏，其它指针的服务问题不在是你的问题。非内存资源迫使你要使用析构函数来确保清理非内存资源。析构函数会对你的应用程序性能产生一些影响，但你必须使用它们来防止资源泄漏(译注：请注意理解非内存资源是什么，一般是指文件句柄，网络资源，或者其它不能在内存中存放的资源)。通过实现 IDisposable 接口来避免析构函数在垃圾回收器上造成的性能损失。接下来的具体的原则将会帮助你更有效的使用环境来开发程序。

原则 12：选择变量初始化而不是赋值语句

Prefer Variable Initializers to Assignment Statements

(译注：根据我个人对文章的理解，我把 initializer 译为：初始化器，它是指初始化语法，也就是在一个类里声明变量的同时，直接创建实例值的方法。

例：object m_o = new object();如果这段代码不在任何函数内，但在一个类里，它就是一个初始化器，而不管你是把它放在类的开始还以结尾。)

一些类经常不只一个构造函数。时间一长，就难得让它的成员变量以及构造函数进行同步了。最好的确保这样的事不会发生的方法就是：在声明就是的时间就直接初始化，而不是在每个构造函数内进行赋值。而且你应该使用初始化器语法同时为静态的和实例的变量进行初始化。

在 C#里，当你声明一个变量时就自然的构造了这个成员变量。直接赋值：

```
public class MyClass
{
  // declare the collection, and initialize it.
  private ArrayList _coll = new ArrayList();
}
```

忽略你最终会给 MyClass 添加多少个构造函数，_coll 会正确的初始化。编译器会产生一些代码，使得在你的任何一个构造函数调用前，都会初始化你声明的实例变量。当你添加一个新的构造函数时，_coll 就给你初始化了。当你添加了一个新的变量，你不用在所有的构造函数里添加初始化代码；直接在声明的地方对它进行初始化就行了。同样重要的是：如果你没有明确的声明任何一个构造函数，编译会默认的给你添加一个，并且把所有的变量初始化过程都添加到这个构造函数里。

初始化器更像是一个到构造函数的方便的快捷方法。初始化生成的代码会放置在类型的构造函数之前。初始化会在执行类型的基类的构造函数之前被执行，并且它们是按你声明的先后关系顺序执行的。

使用初始化器是一个最简单的方法，在你的类型里来避免使用一些没有赋值的变量，但这并不是很好。下面三种情况下，你不应该使用初始化器语法。首先就是，如果你是初始化一个对象为 0，或者为 null。系统默认会在你任何代码执行前，为所有的内容都初始化为 0。系统置 0 的初始化是基于底层的 CPU 指令，对整个内存块设置。你的任何其它置 0 的初始化语句是多余的。C#编译器忠实的添加额外的指令把内存设置为 0。这并没有错，只是效率不高。事实上，如果是处理值类型数据，这是很不值的：

```
MyValType _MyVal1;  // initialized to 0
MyValType _MyVal2 = new MyValType(); // also 0
```

两条语句都是把变量置为 0。第一个是通过设置包含_MyVal1 的内存来置 0;而第二个是通过 IL 指令 initobj，这对变量_MyVal2 会产生装箱与拆箱操作。这很要花一点额外的时间(参见原则 17)。

第二个低效率的是在你为一个对象添加两个构造函数时会产生。你使用初始化器初始化变量，而所有的构造函数也对这些变量进行了初始化。这个版本的 MyClass 两个不同的 ArrayList 对象在它的构造函数内：

```
public class MyClass
{
  // declare the collection, and initialize it.
  private ArrayList _coll = new ArrayList();
  MyClass()
  {
  }
  MyClass(int size)
  {
    _coll = new ArrayList(size);
  }
}
```

当你创建一个新的 MyClass 对象时，特别指定集合的大小，你创建了两个数组列表。其中一个很快成为垃圾对象。初始化器在所有的构造函数之前会执行，构造函数会创建第 2 个数组列表。编译器产生了这个的一个版本，当然这是你决不会手动写出来的。(参见原则 14 来使用一个恰当的方法来解决这个问题)

```
public class MyClass
{
  // declare the collection, and initialize it.
  private ArrayList _coll;
  MyClass()
  {
    _coll = new ArrayList();
  }
  MyClass(int size)
  {
    _coll = new ArrayList();
    _coll = new ArrayList(size);
  }
}
```

最后一个原因要把初始化放到构造函数里就是促使异常的捕获。你不能在初始化器中使用 try 块，任何在构造时因成员变量产生的异常可能衍生到对象的外面。你无法试图在你的类里来捕获它。你应该把那些初始化代码移到构造函数里，这样你就可以捕获异常从而保证你的代码很友好(参见原则 45)。

变量初始化器是一个最简单的方法，在忽略构造函数时来保证成员变量被正确的初始化。初始化器在所有的构造函数之前被执行。使用这样的语法意味着当你在为后来发布的版本中添加了构造函数时，不会忘记添加恰当的初始化到构造函数里。当构造函数与初始化生成同样的成员对象时，就使用初始化器。阅读简单而且易于维护。

原则 13：用静态构造函数初始化类的静态成员

Initialize Static Class Members with Static Constructors

(译注：initializer 在上文中译为了“初始化器”，实在不好听，本文中全部改译为：“预置方法”)

你应该知道，在一个类型的任何实例初始化以前，你应该初始化它的静态成员变量。在里 C#你可以使用静态的预置方法和静态构造函数来实现这个目的。一个类的静态构造函数是一个与众不同的，它在所有的方法，变量或

者属性访问前被执行。你可以用这个函数来初始化静态成员变量，强制使用单件模式，或者实现其它任何在类型的实例可用前应该完成的工作。你不能用任何的实例构造函数，其它特殊的私有函数， 或者任何其它习惯方法来初始化一个 变量(译注：编译器就不让你这样做，所以你不用担心这样的问题)。

和实例的预置方法一样，你可以把静态的预置方法做为静态构造函数可替代的选择。如果须要简单的分配一个静态成员，就直接使用初始化语法。当你有更复杂的逻辑来初始化静态成员变量时，就创建一个静态构造函数：

```
public class MySingleton
{
  private static readonly MySingleton _theOneAndOnly =
    new MySingleton();
  public static MySingleton TheOnly
  {
    get
    {
      return _theOneAndOnly;
    }
  }
  private MySingleton()
  {
  }
  // remainder elided
}
```

可以用下面的方法简单的实现单件模式，实际上你在初始化一个单件模式时可能有更复杂的逻辑：

```
public class MySingleton
{
  private static readonly MySingleton _theOneAndOnly;
  static MySingleton()
  {
    _theOneAndOnly = new MySingleton();
  }
  public static MySingleton TheOnly
  {
    get
    {
      return _theOneAndOnly;
    }
  }
  private MySingleton()
  {
  }
  // remainder elided
}
```

同样，和实例的预置方法一样，静态的预置方法在静态的构造函数调用前执行。并且，你的静态预置方法在基类的静态构造函数执行前被执行。

当应用程序第一次装载你的数据类型时，CLR 自动调用静态构造函数。你只能定义一个静态构造函数，并且不能有参数。因为静态构造函数是 CLR 调用的，你必须十分注意异常的产生。如果在静态构造函数里产生了异常，CLR

将会直接终止你的应用程序。正因为异常，静态构造函数常常代替静态预置方法。如果你使用静态预置方法，你自己不能捕获异常。做为一个静态的构造，你可以这样(参见原则 45)：

```
static MySingleton()
{
  try {
    _theOneAndOnly = new MySingleton();
  } catch
  {
    // Attempt recovery here.
  }
}
```

静态预置方法和静态构造函数为你的类提供了最清爽的方法来初始化静态成员。与其它语言不同，它们被添加到 C#语言中，是初始化静态成员的两个不同的特殊位置。

原则 14：使用构造函数链

Utilize Constructor Chaining

写构造函数是一个反复的工作。很多开发人员都是先写一个构造函数，然后复制粘贴到其它的构造函数里，以此来满足类的一些重载接口。希望你不是这样做的，如果是的，就此停止吧。有经验的 C++程序可能会用一个辅助的私有方法，把常用的算法放在里面来构造对象。也请停止吧。当你发现多重构造函数包含相同的逻辑时，取而代之的是把这些逻辑放在一个常用的构造函数里。你可以得避免代码的重复的好处，并且构造函数初始化比对象的其它代码执行起来更高效。C#编译器把构造函数的初始化识别为特殊的语法，并且移除预置方法中重复的变量和重复的基类构造函数。结果就是这样的，你的对象最终执行最少的代码来合理的初始化对象。你同样可以写最少的代码来把负责委托给一个常用的构造函数。构造函数的预置方法充许一个构造函数调用另一个构造函数。这是一个简单的例子：

```
public class MyClass
{
  // collection of data
  private ArrayList _coll;
  // Name of the instance:
  private string  _name;
  public MyClass() :
    this(0, "")
  {
  }
  public MyClass(int initialCount) :
    this(initialCount, "")
  {
  }
  public MyClass(int initialCount, string name)
  {
    _coll = (initialCount > 0) ?
      new ArrayList(initialCount) :
      new ArrayList();
    _name = name;
```

```
  }
}
```

C#不支持带默认值的参数，C++是很好的解决这个问题的(译注：C++可以让参数有默认的值，从而有效的减少函数的重载)。你必须重写每一个特殊的构造函数。对于这样的构造函数，就意味着大量的代码重复工作。可以使用构造函数链来取代常规的方法。下面就是一些常规的低效率的构造函数逻辑：

```
public class MyClass
{
  // collection of data
  private ArrayList _coll;
  // Name of the instance:
  private string  _name;
  public MyClass()
  {
    commonConstructor(0, "");
  }
  public MyClass(int initialCount)
  {
    commonConstructor(initialCount, "");
  }
  public MyClass(int initialCount, string Name)
  {
    commonConstructor(initialCount, Name);
  }
  private void commonConstructor(int count,
    string name)
  {
    _coll = (count > 0) ?
      new ArrayList(count) :
      new ArrayList();
    _name = name;
  }
}
```

这个版本看上去是一样的，但生成的效率远不及对象的其它代码。为了你的利益，编译器为构造函数添加了一些代码。添加了一些代码来初始化所有的变量(参见原则 12)。它还调用了基类的构造函数。当你自己写一些有效的函数时，编译器就不会添加这些重复的代码了。第二个版本的 IL 代码和下面写的是一样的：

```
// Not legal, illustrates IL generated:
public MyClass()
{
  private ArrayList _coll;
  private string  _name;
  public MyClass()
  {
    // Instance Initializers would go here.
    object(); // Not legal, illustrative only.
    commonConstructor(0, "");
```

```
  }
  public MyClass (int initialCount)
  {
    // Instance Initializers would go here.
    object(); // Not legal, illustrative only.
    commonConstructor(initialCount, "");
  }
  public MyClass(int initialCount, string Name)
  {
    // Instance Initializers would go here.
    object(); // Not legal, illustrative only.
    commonConstructor(initialCount, Name);
  }
  private void commonConstructor(int count,
    string name)
  {
    _coll = (count > 0) ?
      new ArrayList(count) :
      new ArrayList();
    _name = name;
  }
}
```

如果你用第一个版本写构造函数，在编译看来，你是这样写的：

```
// Not legal, illustrates IL generated:
public MyClass()
{
  private ArrayList _coll;
  private string  _name;
  public MyClass()
  {
    // No variable initializers here.
    // Call the third constructor, shown below.
    this(0, ""); // Not legal, illustrative only.
  }
  public MyClass (int initialCount)
  {
    // No variable initializers here.
    // Call the third constructor, shown below.
    this(initialCount, "");
  }
  public MyClass(int initialCount, string Name)
  {
    // Instance Initializers would go here.
    object(); // Not legal, illustrative only.
    _counter = initialCount;
```

```
    _name = Name;
  }
}
```

不同之处就是编译器没有生成对基类的多重调用，也没有复制实例变量到每一个构造函数内。实际上基类的构造函数只是在最后一个构造函数里被调用了，这同样很重要：你不能包含更多的构造函数预置方法。在这个类里，你可以用 this()把它委托给另一个方法，或者你可以用 base()调用基类的构造。但你不能同时调用两个。

还不清楚构造函数预置方法吗？那么考虑一下只读的常量，在这个例子里，对象的名字在整个生命期内都不应该改变。这就是说，你应该把它设置为只读的。如果使用辅助函数来构造对象就会得到一个编译错误：

```
public class MyClass
{
  // collection of data
  private ArrayList _coll;
  // Number for this instance
  private int       _counter;
  // Name of the instance:
  private readonly string  _name;
  public MyClass()
  {
    commonConstructor(0, "");
  }
  public MyClass(int initialCount)
  {
    commonConstructor(initialCount, "");
  }
  public MyClass(int initialCount, string Name)
  {
    commonConstructor(initialCount, Name);
  }
  private void commonConstructor(int count,
    string name)
  {
    _coll = (count > 0) ?
      new ArrayList(count) :
      new ArrayList();
    // ERROR changing the name outside of a constructor.
    _name = name;
  }
}
```

C++程序会把这个_name 留在每一个构造函数里，或者通常是在辅助函数里把它丢掉。C#的构造函数预置方法提供了一个好的选择，几乎所有的琐碎的类都包含不只一个构造函数，它们的工作就是初始化对象的所有成员变量 。这是很常见的，这些函数在理想情况下有相似的共享逻辑结构。使用 C#构造预置方法来生成这些常规的算法，这样就只用写一次也只执行一次。

这是 C#里的最后一个关于对象构造的原则，是时候复习一下，一个类型在构造时的整个事件顺序了。你须要同时明白一个对象的操作顺序和默认的预置方法的顺序。你构造过程中，你应该努力使所有的成员变量只精确的初始化一次。最好的完成这个目标的方法就是尽快的完成变量的初始化。这是某个类型第一次构造一个实例时的顺序：

1、静态变量存储位置 0。
2、静态变量预置方法执行。
3、基类的静态构造函数执行。
4、静态构造函数执行。
5、实例变量存储位置 0。
6、实例变量预置方法执行。
7、恰当的基类实例构造函数执行。
8、实例构造函数执行。

后续的同样类型的实例从第 5 步开始，因为类的预置方法只执行一次。同样，第 6 和第 7 步是优化了的，它可以让编译器在构造函数预置方法上移除重复的指令。

C#的编译器保证所有的事物在初始化使用同样的方法来生成。至少，你应该保证在你的类型创建时，对象占用的所有内存是已经置 0 的。对静态成员和实例成员都是一样的。你的目标就是确保你希望执行的初始化代码只执行一次。使用预置方法来初始化简单的资源，使用构造函数来初始化一些具有复杂逻辑结构的成员。同样，为了减少重复尽可能的组织调用其它的构造函数。

原则 15：使用 using 和 try/finally 来做资源清理

Utilize using and TRy/finally for Resource Cleanup

使用非托管资源的类型必须实现 IDisposable 接口的 Dispose()方法来精确的释放系统资源。.Net 环境的这一规则使得释放资源代码的职责是类型的使用者，而不是类型或系统。因此，任何时候你在使用一个有 Dispose()方法的类型时，你就有责任来调用 Dispose()方法来释放资源。最好的方法来保证 Dispose()被调用的结构是使用 using 语句或者 try/finally 块。

所有包含非托管资源的类型应该实现 IDisposable 接口，另外，当你忘记恰当的处理这些类型时，它们会被动的创建析构函数。如果你忘记处理这些对象，那些非内存资源会在晚些时候，析构函数被确切调用时得到释放。这就使得这些对象在内存时待的时间更长，从而会使你的应用程序会因系统资源占用太多而速度下降。

幸运的是，C#语言的设计者精确的释放资源是一个常见的任务。他们添加了一个关键字来使这变得简单了。

假设你写了下面的代码：

```
public void ExecuteCommand(string connString,
  string commandString)
{
  SqlConnection myConnection = new SqlConnection(connString);
  SqlCommand mySqlCommand = new SqlCommand(commandString,
    myConnection);
  myConnection.Open();
  mySqlCommand.ExecuteNonQuery();
}
```

这个例子中的两个可处理对象没有被恰当的释放：SqlConnection 和 SqlCommand。两个对象同时保存在内存里直到析构函数被调用。(这两个类都是从 System.ComponentModel.Component 继承来的。)

解决这个问题的方法就是在使用完命令和链接后就调用它们的 Dispose:

```
public void ExecuteCommand(string connString,
  string commandString)
{
  SqlConnection myConnection = new SqlConnection(connString);
  SqlCommand mySqlCommand = new SqlCommand(commandString,
    myConnection);
```

```
  myConnection.Open();
  mySqlCommand.ExecuteNonQuery();
  mySqlCommand.Dispose();
  myConnection.Dispose();
}
```

这很好，除非 SQL 命令在执行时抛出异常，这时你的 Dispose()调用就永远不会成功。using 语句可以确保 Dispose()方法被调用。当你把对象分配到 using 语句内时，C#的编译器就把这些对象放到一个 try/finally 块内：

```
public void ExecuteCommand(string connString,
  string commandString)
{
  using (SqlConnection myConnection = new
    SqlConnection(connString))
  {
    using (SqlCommand mySqlCommand = new
      SqlCommand(commandString,
      myConnection))
    {
      myConnection.Open();
      mySqlCommand.ExecuteNonQuery();
    }
  }
}
```

当你在一个函数内使用一个可处理对象时，using 语句是最简单的方法来保证这个对象被恰当的处理掉。当这些对象被分配时，会被编译器放到一个 try/finally 块中。下面的两段代码编译成的 IL 是一样的：

```
SqlConnection myConnection = null;
// Example Using clause:
using (myConnection = new SqlConnection(connString))
{
  myConnection.Open();
}

// example Try / Catch block:
try {
  myConnection = new SqlConnection(connString);
  myConnection.Open();
}
finally {
  myConnection.Dispose();
}
```

(译注：就我个人对 try/catch/finally 块的使用经验而言，我觉得上面这样的做法非常不方便。可以保证资源得到释放，却无法发现错误。关于如何同时抛出异常又释放资源的方法可以参考一下其它相关资源，如 Jeffrey 的.Net 框架程序设计,修订版)

如果你把一个不能处理类型的变量放置在 using 语句内，C#编译器给出一个错误，例如：

```
// Does not compile:
// String is sealed, and does not support IDisposable.
```

```
using(string msg = "This is a message")
  Console.WriteLine(msg);
```

using 只能在编译时，那些支持 IDispose 接口的类型可以使用，并不是任意的对象：

```
// Does not compile.
// Object does not support IDisposable.
using (object obj = Factory.CreateResource())
  Console.WriteLine(obj.ToString());
```

如果 obj 实现了 IDispose 接口，那么 using 语句就会生成资源清理代码，如果不是，using 就退化成使用 using(null)，这是安全的，但没有任何作用。如果你对一个对象是否应该放在 using 语句中不是很确定，宁可为了更安全：假设要这样做，而且按前面的方法把它放到 using 语句中。

这里讲了一个简单的情况：无论何时，当你在某个方法内使用一个可处理对象时，把这个对象放在 using 语句内。现在你学习一些更复杂的应用。还是前面那个例子里须要释放的两个对象：链接和命令。前面的例子告诉你创建了两个不同的 using 语句，一个包含一个可处理对象。每个 using 语句就生成了一个不同的 try/finally 块。等效的你写了这样的代码：

```
public void ExecuteCommand(string connString,
  string commandString)
{
  SqlConnection myConnection = null;
  SqlCommand mySqlCommand = null;
  try
  {
    myConnection = new SqlConnection(connString);
    try
    {
      mySqlCommand = new SqlCommand(commandString,
      myConnection);
      myConnection.Open();
      mySqlCommand.ExecuteNonQuery();
    }
    finally
    {
      if (mySqlCommand != null)
        mySqlCommand.Dispose();
    }
  }
  finally
  {
    if (myConnection != null)
      myConnection.Dispose();
  }
}
```

每一个 using 语句生成了一个新的嵌套的 try/finally 块。我发现这是很糟糕的结构，所以，如果是遇到多个实现了 IDisposable 接口的对象时，我更愿意写自己的 try/finally 块：

```
public void ExecuteCommand(string connString,
  string commandString)
```

```
{
  SqlConnection myConnection = null;
  SqlCommand mySqlCommand = null;
  try {
    myConnection = new SqlConnection(connString);
    mySqlCommand = new SqlCommand(commandString,
      myConnection);
    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();
  }
  finally
  {
    if (mySqlCommand != null)
      mySqlCommand.Dispose();
    if (myConnection != null)
      myConnection.Dispose();
  }
}
```

(译注：作者里的判断对象是否为 null 是很重要的，特别是一些封装了 COM 的对象，有些时候的释放是隐式的，当你再释放一些空对象时会出现异常。例如：同一个 COM 被两个不同接口的变量引用时，在其中一个上调用了 Dispose 后，另一个的调用就会失败。在.Net 里也要注意这样的问题，所以要判断对象是否为 null)

然而，请不要自作聪明试图用 as 来写这样的 using 语句：

```
public void ExecuteCommand(string connString,
  string commandString)
{
  // Bad idea. Potential resource leak lurks!
  SqlConnection myConnection =
    new SqlConnection(connString);
  SqlCommand mySqlCommand = new SqlCommand(commandString,
      myConnection);
      using (myConnection as IDisposable)
      using (mySqlCommand as IDisposable)
      {
        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
      }
}
```

这看上去很清爽，但有一个狡猾的(subtle)的 bug。 如果 SqlCommand()的构造函数抛出异常，那么 SqlConnection 对象就不可能被处理了。你必须确保每一个实现了 IDispose 接口的对象分配在在 using 范围内，或者在 try/finally 块内。否则会出现资源泄漏。

目前为止，你已经学会了两种最常见的情况。无论何时在一个方法内处理一个对象时，使用 using 语句是最好的方法来确保申请的资源在各种情况下都得到释放。当你在一个方法里分配了多个(实现了 IDisposable 接口的)对象时，创建多个 using 块或者使用你自己的 try/finally 块。

对可处理对象的理解有一点点细微的区别。有一些对象同时支持 Disponse 和 Close 两个方法来释放资源。SqlConnection 就是其中之一，你可以像这样关闭 SqlConnection：

```
public void ExecuteCommand(string connString,
  string commandString)
{
  SqlConnection myConnection = null;
  try {
    myConnection = new SqlConnection(connString);
    SqlCommand mySqlCommand = new SqlCommand(commandString,
      myConnection);
    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();
  }
  finally
  {
    if (myConnection != null)
      myConnection.Close();
  }
}
```

这个版本关闭了链接，但它确实与处理对象是不一样的。Dispose 方法会释放更多的资源，它还会告诉 GC，这个对象已经不再须要析构了(译注：关于 C#里的析构，可以参考其它方面的书籍)。Dispose 会调用 GC.SuppressFinalize()，但 Close()一般不会。结果就是，对象会到析构队列中排队，即使析构并不是须要的。当你有选择时，Dispose()比 Colse()要好。你会在原则 18 里学习更更精彩的内容。

Dispose()并不会从内存里把对象移走，对于让对象释放非托管资源来说是一个 hook。这就是说你可能遇到这样的难题，就是释放一个还在使用的对象。不要释放一个在程序其它地方还在引用的对象。

在某些情况下，C#里的资源管理比 C++还要困难。你不能指望确定的析构函数来清理你所使用的所有资源。但垃圾回收器却让你更轻松，你的大从数类型不必实现 IDisposable 接口。在.Net 框架里的 1500 多个类中，只有不到 100 个类实现了 IDisposable 接口。当你使用一个实现了 IDisposeable 接口的对象时，记得在所有的类里都要处理它们。你应该把它们包含在 using 语句中，或者 try/finally 块中。不管用哪一种，请确保每时每刻对象都得到了正确的释放。

原则 16：垃圾最小化

Minimize Garbage

垃圾回收器对内存管理表现的非常出色，并且它以非常高效的方法移除不再使用的对象。但不管你怎样看它，申请和释放一个基于堆内存的对象总比申请和释放一个不基于堆内存的对象要花上更多的处理器时间。你可以给出一些严重的性能问题，例如应用程序在某个方法内分配过量的引用对象。

你不应该让垃圾回收器超负荷的工作，为了程序的效率，你可以使用一些简单的技巧来减少垃圾回收器的工作。所有的引用类型，即使是局部变量，都是在堆上分配的。所有引用类型的局部变量在函数退出后马上成为垃圾，一个最常见的"垃圾"做法就是申请一个 Windows 的画图句柄：

```
protected override void OnPaint(PaintEventArgs e)
{
  // Bad. Created the same font every paint event.
  using (Font MyFont = new Font("Arial", 10.0f))
  {
    e.Graphics.DrawString(DateTime.Now.ToString(),
      MyFont, Brushes.Black, new PointF(0,0));
```

```
  }
  base.OnPaint(e);
}
```

OnPaint()函数的调用很频繁的，每次调用它的时候，都会生成另一个 Font 对象，而实际上它是完全一样的内容。垃圾回收器每次都须要清理这些对象。这将是难以置信的低效。

取而代之的是，把 Font 对象从局部变量提供为对象成员，在每次绘制窗口时重用同样的对象：

```
private readonly Font _myFont =
  new Font("Arial", 10.0f);
protected override void OnPaint(PaintEventArgs e)
{
  e.Graphics.DrawString(DateTime.Now.ToString(),
    _myFont, Brushes.Black, new PointF(0,0));
  base.OnPaint(e);
}
```

这样你的程序在每次 paint 事件发生时不会产生垃圾，垃圾回收器的工作减少了，你的程序运行会稍微快一点点。当你把一个实现了 IDisposable 接口的局部变量提升为类型成员时，例如字体，你的类同样也应该实现 IDisposable 接口。原则 18 会给你解释如何正确的完成它。

当一个引用类型(值类型的就无所谓了)的局部变量在常规的函数调用中使用的非常频繁时，你应该把它提升为对象的成员。那个字体就是一个很好的例子。只有常用的局部变量频繁访问时才是很好的候选对象，不是频繁调用的就不必了。你应该尽可能的避免重复的创建同样的对象，使用成员变量而不是局部变量。

前面例子中使用的静态属性 Brushes.Black，演示了另一个避免重复创建相似对象的技术。使用静态成员变量来创建一些常用的引用类型的实例。考虑前面那个例子里使用的黑色画刷，每次当你要用黑色画刷来画一些东西时，你要在程序中创建和释放大量的黑色画刷。前面的一个解决方案就是在每个期望黑色画刷的类中添加一个画刷成员，但这还不够。程序可能会创建大量的窗口和控件，这同样会创建大量的黑色画刷。.Net 框架的设计者预知了这个问题，他们为你创建一个简单的黑色画刷以便你在任何地方都可以重复使用。Brushes 对象包含一定数量的静态 Brush 对象，每一个具有不同的常用的颜色。在内部，Brushes 使用了惰性算法来，即只有当你使用时才创建这些对象。一个简单的实现方法：

```
private static Brush _blackBrush;
public static Brush Black
{
  get
  {
    if (_blackBrush == null)
      _blackBrush = new SolidBrush(Color.Black);
      return _blackBrush;
  }
}
```

当你第一次申请黑色画刷时，Brushes 类就会创建它。然而 Brushes 类就保留一个单一的黑色画刷的引用句柄，当你再次申请时它就直接返回这个句柄。结果就是你只创建了一个黑色画刷并且一直在重用它。另外，如果你的应用程序不须要一个特殊的资源，一个柠檬绿(lime green)的画刷就可能永远不会创建。框架提供了一个方法来限制对象，使得在满足目标的情况下使用最小的对象集合。学会在你的应用程序里使用这样的技巧。

你已经学会了两种技术来最小化应用程序的(对象)分配数量，正如它承担它自己的任务一样。你可以把一个经常使用的局部变量提升为类的成员变量，你可以提供一个类以单件模式来存储一些常用的给定对象的实例。最后一项技术还包括创建恒定类型的最终使用值。System.String 类就是一个恒定类型，在你创建一个字符串后，它的内容就不能更改了。当你编写代码来修改这些串的内容时，你实际上是创建了新的对象，并且让旧的串成为了垃圾。

这看上去是清白的例子：

```
string msg = "Hello, ";
msg += thisUser.Name;
msg += ". Today is ";
msg += System.DateTime.Now.ToString();
```

这实际上低效的如果你是这样写：

```
string msg = "Hello, ";
// Not legal, for illustration only:
string tmp1 = new String(msg + thisUser.Name);
string msg = tmp1; // "Hello " is garbage.
string tmp2 = new String(msg + ". Today is ");
msg = tmp2; // "Hello <user>" is garbage.
string tmp3 = new String(msg + DateTime.Now.ToString());
msg = tmp3;// "Hello <user>. Today is " is garbage.
```

字符串 tmp1,tmp2,tmp3 以及最原始的 msg 构造的(“Hello”)，都成了垃圾。+=方法在字符串类上会生成一个新的对象并返回它。它不会通过把字符链接到原来的存储空间上来修改结果。对于先前这个例子，给一个简单的构造例子，你应该使用 string.Format()方法：

```
string msg = string.Format ("Hello, {0}. Today is {1}",
  thisUser.Name, DateTime.Now.ToString());
```

对于更多的复杂的字符串操作，你应该使用 StringBuilter 类：

```
StringBuilder msg = new StringBuilder("Hello, ");
msg.Append(thisUser.Name);
msg.Append(". Today is ");
msg.Append(DateTime.Now.ToString());
string finalMsg = msg.ToString();
```

StringBuilder 也一个(内容)可变的字符串类，用于生成恒定的字符串对象。在你还没有创建一个恒定的字符串对象前，它提供了一个有效的方法来存储可变的字符串。更重要的是，学习这样的设计习惯。当你的设计提倡使用恒定类型时(参见原则 7)，对于一些要经过多次构造后才能最终得到的对象，可以考虑使用一些对象生成器来简化对象的创建。它提供了一个方法让你的用户来逐步的创建(你设计的)恒定类型，也用于维护这个类型。

(译注：请理解作者的意图，只有当你使用恒定类型时才这样，如果是引用类型，就不一定非要使用对象生成器了。而且注意恒定类型的特点，就是一但创建就永远不能改变，所有的修改都会产生新的实例，string 就是一个典型的例子，它是一个恒定的引用类型；还有 DateTime 也是一个，它是一个恒定的值类型。)

垃圾回收器在管理应用程序的内存上确实很高效。但请记住，创建和释放堆对象还是很占时间的。避免创建大量的对象，也不要创建你不使用的对象。也要避免在局部函数上多次创建引用对象。相反，把局部变量提供为类型成员变量，或者把你最常用的对象实例创建为静态对象。最后，考虑使用可变对象创建器来构造恒定对象。

原则 17：装箱和拆箱的最小化

Minimize Boxing and Unboxing

值类型是数据的容器，它们不具备多太性。另一方面就是说，.Net 框架被设计成单一继承的引用类型，System.Object，在整个继承关系中做为根对象存在。设计这两种类型的目的是截然不同的，.Net 框架使用了装箱与拆箱来链接两种不同类型的数据。装箱是把一个值类型数据放置在一个无类型的引用对象上，从而使一个值类型在须要时可以当成引用类型来使用。拆箱则是额外的从“箱”上拷贝一份值类型数据。装箱和拆箱可以让你在须要使用 System.Object 对象的地方使用值类型数据。但装箱与拆箱操作却是性能的强盗，在些时候装箱与拆箱会产生一些临时对象，它会导致程序存在一些隐藏的 BUG。应该尽可能的避免使用装箱与拆箱。

装箱可以把一个值类型数据转化也一个引用类型，一个新的引用对象在堆上创建，它就是这个“箱子”，值类型的数据就在这个引用类型中存储了一份拷贝。参见图 2.3，演示了装箱的对象是如何访问和存储的。箱子中包含一份这个值类型对象的拷贝，并且复制实现了已经装箱对象的接口。当你想从这个箱子中取回任何内容时，一个值类型数据的拷贝会被创建并返回。这就是装箱与拆箱的关键性概念：对象的一个拷贝存放到箱子中，而不管何时你再访问这个箱子时，另一个拷贝又会被创建。

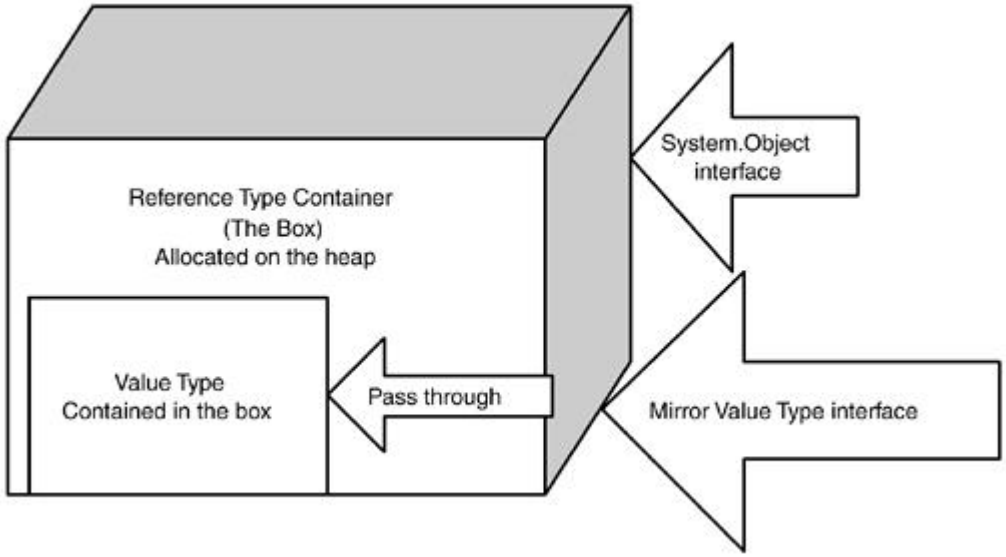


图 2.3，值类型数据在箱子中。把一个值类型数据转化成一个 System.Object 的引用，一个无名的引用类型会被创建。值类型的数据就存储在这个无名的引用对象中，所有的访问方法都要通过这个箱子才能到达值类型数据存储的地方。

最阴险的地方是这个装箱与拆箱很多时候是自动完成的!当你在任何一个期望类型是 System.Object 的地方使用值类型数据时，编译器会生成装箱与拆箱的语句。另外，当你通过一个接口指针来访问值类型数据时，装箱与拆箱也会发生。当你装箱时不会得到任何警告，即使是最简单的语句也一样。例如下面这个：

```
Console.WriteLine("A few numbers:{0}, {1}, {2}",
  25, 32, 50);
```

使用重载的 Console.WriteLine 函数须要一个 System.Object 类型的数组引用，整型是值类型，所以必须装箱后才能传给重载的 WriteLine 方法。唯一可以强制这三个整数成为 System.Object 对象的方法就是把它们装箱。另外，在 WriteLine 内部，通过调用箱子对象上的 ToString()方法来到达箱子内部。某种意义上讲，你生成了这样的结构：

```
int i =25;
object o = i; // box
Console.WriteLine(o.ToString());
```

在 WriteLine 内部，下面的执行了下面的代码：

```
object o;
int i = (int)o; // unbox
string output = i.ToString();
```

你可能自己从来不会写这样的代码，但是，却让编译器自动从一个指定的类型转化为 System.Object，这确实是你做的。编译器只是想试着帮助你，它想让你成功(调用函数)，它也很乐意在必要时候为你生成装箱和拆箱语句，从而把一个值类型数据转化成 System.Object 的实例。为了避免这么挑剔的惩罚，在使用它们来调用 WriteLine 之前，你自己应该把你的类型转化成字符串的实例。

```
Console.WriteLine("A few numbers:{0}, {1}, {2}",
 25.ToString(), 32.ToString(), 50.ToString());
```

(译注：注意，在自己调用 ToString 方法时，还是会在堆上创建一个引用实例，但它的好处是不用拆箱，因为对象已经是一个引用类型了。)

这段代码使用已知的整数类型，而且值类型再也不会隐式的转化为 System.Object 类型。这个常见的例子展示了避免装箱的第一个规则：注意隐式的转化为 System.Object，如果可以避免，值类型不应该被 System.Object 代替。

另一个常见情况就是，在使用.Net 1.x 的集合时，你可能无意的把一个值类型转化成 System.Object 类型。任何时候，当你添加一个值类型数据到集合时中，你就创建了一个箱子。任何时候从集合中移出一个对象时，你得到的是箱子里的一个拷贝。从箱子里取一个对象时，你总是要创建一个拷贝。这会在应用程序中产生一些隐藏的 BUG。编译器是不会帮你查找这些 BUG 的。这都是装箱惹的祸。让我们开始创建一个简单的结构，可以修改其中一个字段，并且把它的一些实例对象放到一个集合中：

```
public struct Person
{
  private string _Name;
  public string Name
  {
    get
    {
      return _Name;
    }
    set
    {
      _Name = value;
    }
  }
  public override string ToString()
  {
    Return _Name;
  }
}
// Using the Person in a collection:
ArrayList attendees = new ArrayList();
Person p = new Person("Old Name");
attendees.Add(p);
// Try to change the name:
// Would work if Person was a reference type.
Person p2 = ((Person)attendees[ 0 ]);
p2.Name = "New Name";
// Writes "Old Name":
Console.WriteLine(
  attendees[ 0 ].ToString());
```

Person 是一个值类型数据，在存储到 ArrayList 之前它被装箱。这会产生一个拷贝。而在移出的 Persone 对象上通过访问属性做一些修改时，另一个拷贝被创建。而你所做的修改只是针对的拷贝，而实际上还有第三个拷贝通过 ToString()方法来访问 attendees[0]中的对象。

正因为这以及其它一些原因，你应该创建一些恒定的值类型(参见原则 7)。如果你非要在集合中使用可变的值类型，那就使用 System.Array 类，它是类型安全的。

如果一个数组不是一个合理的集合，以 C#1.x 中你可以通过使用接口来修正这个错误。尽量选择一些接口而不是公共的方法，来访问箱子的内部去修改数据：

```
public interface IPersonName
{
  string Name
  {
    get; set;
  }
}
struct Person : IPersonName
{
  private string _Name;
  public string Name
  {
    get
    {
      return _Name;
    }
    set
    {
      _Name = value;
    }
  }
  public override string ToString()
  {
    return _Name;
  }
}
// Using the Person in a collection:
ArrayList attendees = new ArrayList();
Person p = new Person("Old Name");
attendees.Add(p); // box
// Try to change the name:
// Use the interface, not the type.
// No Unbox needed
((IPersonName)attendees[ 0 ]).Name = "New Name";
// Writes "New Name":
Console.WriteLine(
  attendees[ 0 ].ToString()); // unbox
```

装箱后的引用类型会实现原数据类型上所有已经实现的接口。这就是说，不用做拷贝，你可以通过调用箱子上的 IPersonaName.Name 方法来直接访问请求到箱子内部的值类型数据。在值类型上创建的接口可以让你访问集合里的箱子的内部，从而直接修改它的值。在值类型上实现的接口并没有让值类型成为多态的，这又会引入装箱的惩罚(参见原则 20)。

在 C#2.0 中对泛型简介中，很多限制已经做了修改(参见原则 49)。泛型接口和泛型集合会时同处理好集合与接口的困境。在那之前，我们还是要避免装箱。是的，值类型可以转化为 System.Object 或者其它任何的接口引用。

这些转化是隐式的，使得发现它们成为繁杂的工作。这些也就是环境和语言的规则，装箱与拆箱操作会在你不经意时做一些对象的拷贝，这会产生一些 BUG。同样，把值类型多样化处理会对性能有所损失。时刻注意那些把值类型转化成 System.Object 或者接口类型的地方：把值类型放到集合里，调用定义参数为 System.Object 类型的方法，或者强制转化为 System.Object。能够避免就尽量避免!

原则 18：实现标准的处理(Dispose)模式

Implement the Standard Dispose Pattern

我们已经讨论过，处理一个占用了非托管资源对象是很重要的。现在是时候来讨论如何写代码来管理这些类占用的非内存资源了。一个标准的模式就是利用.Net 框架提供的方法处理非内存资源。你的用户也希望你遵守这个标准的模式。也就是通过实现 IDisposable 接口来释放非托管的资源，当然是在用户记得调用它的时候，但如果用户忘记了，析构函数也会被动的执行。它是和垃圾回收器一起工作的，确保在一些必要时候，你的对象只会受到因析构函数而造成的性能损失。这正是管理非托管资源的好方法，因此有必要彻底的弄明白它。

处在类继承关系中顶层的基类应该实现 IDisposable 接口来释放资源。这个类型也应该添加一个析构函数，做为最后的被动机制。这两个方法都应该是用虚方法来释放资源，这样可以让它的派生类重载这个函数来释放它们自己的资源。派生类只有在它自己须要释放资源时才重载这个函数，并且一定要记得调用基类的方法。

开始时，如果你的类使用了非内存资源，则一定得有一个析构函数。你不能指望你的用户总是记得调用 Dispose 方法，否则当他们忘记时，你会丢失一些资源。这或许是因为他们没有调用 Dispose 的错误，但你也有责任。唯一可以确保非内存资源可以恰当释放的方法就是创建一个析构函数。所以，添加一个析构函数吧!

当垃圾回收器运行时，它会直接从内存中移除不用析构的垃圾对象。而其它有析构函数的对象还保留在内存中。这些对象被添加到一个析构队列中，垃圾回收器会起动一个线程专门来析构这些对象。当析构线程完成它的工作后，这些垃圾对象就可以从内存中移除了。就是说，须要析构的对象比不须要析构的对象在内存中待的时间要长。但你没得选择。如果你是采用的这种被动模式，当你的类型占用非托管资源时，你就必须写一个析构函数。但目前你还不用担心性能问题，下一步就保证你的用户使用更加简单，而且可以避免因为析构函数而造成的性能损失。

实现 IDisposable 接口是一个标准的模式来告诉用户和进行时系统：你的对象占有资源而且必须及时的释放。IDisposable 接口只有一个方法：

```
public interface IDisposable
{
  void Dispose();
}
```

实现 IDisposable.Dispose()方法有责任完成下面的任务：

1、感知所有的非托管资源。

2、感知所有的托管资源(包括卸载一些事件)。

3、设置一个安全的标记来标识对象已经被处理。如果在已经处理过的对象上调用任何方法时，你可以检验这个标记并且抛出一个 ObjectDisposed 的异常。

4、阻止析构。你要调用 GC.SuppressFinalize(this)来完成最后的工作。

通过实现 IDisposable 接口，你写成了两件事：第一就是提供了一个机制来及时的释放所有占用的托管资源(译注：这里就是指托管资源，当实现了这个接口后，可以通过调用 Dispose 来立即释放托管资源)，另一个就是你提供了一个标准的模式让用户来释放非托管资源。这是十分重要的，当你在你的类型上实现了 IDisposable 接口以后，用户就可以避免析构时的损失。你的类就成了.Net 社区中表现相当良好的成员。

但在你创建的机制中还是存在一些漏洞。如何让一个派生类清理自己的资源，同时还可以让基类很好的再做资源清理呢？(译注：因为调用 Dispose 方法时，必须调用基类的 Dispose，当然是在基类有这个方法时。但前面说过，我们只有一个标记来标识对象是否处理过，不管先调用那个，总得有一个方法不能处理这个标记，而这就存在隐患) 如果基类重载了析构函数，或者自己添加实现了 IDisposable 接口，而这些方法又都是必须调用基类的方法的；否则，基类无法恰当的释放资源。同样，析构和处理共享了一些相同的职责：几乎可以肯定你是复制了析构方

法和处理方法之间的代码。正如你会在原则 26 中学到的，重载接口的方法根本没有如你所期望的那样工作。Dispose 标准模式中的第三个方法，通过一个受保护的辅助性虚函数，制造出它们的常规任务并且挂接到派生类来释放资源。基类包含接口的核心代码， 派生类提供的 Dispose()虚函数或者析构函数来负责清理资源：

```
protected virtual void Dispose(bool isDisposing);
```

重载的方法同时完成析构和处理必须提供的任务，又因为它是虚函数，它为所有的派生类提供函数入口点。派生类可以重载这个函数，提供恰当的实现来释放它自己的资源，并且调用基类的函数。当

isDisposing 为 true 时你可能同时清理托管资源和非托管资源，当 isDisposing 为 false 时你只能清理非托管资源。两种情况下，都可以调用基类的 Dispose(bool)方法让它去清理它自己的资源。

当你实现这样的模式时，这里有一个简单的例子。MyResourceHog 类展示了 IDisposable 的实现，一个析构函数，并且创建了一个虚的 Dispose 方法：

```
public class MyResourceHog：IDisposable
{
  // Flag for already disposed
  private bool _alreadyDisposed = false;
  // finalizer:
  // Call the virtual Dispose method.
  ~MyResourceHog()
  {
    Dispose(false);
  }
  // Implementation of IDisposable.
  // Call the virtual Dispose method.
  // Suppress Finalization.
  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(true);
  }
  // Virtual Dispose method
  protected virtual void Dispose(bool isDisposing)
  {
    // Don't dispose more than once.
    if (_alreadyDisposed)
      return;
    if (isDisposing)
    {
      // TODO: free managed resources here.
    }
    // TODO: free unmanaged resources here.
    // Set disposed flag:
    _alreadyDisposed = true;
  }
}
```

如果派生类有另外的清理任务，就让它实现 Dispose 方法：

```
public class DerivedResourceHog：MyResourceHog
```

```
{
  // Have its own disposed flag.
  private bool _disposed = false;
  protected override void Dispose(bool isDisposing)
  {
    // Don't dispose more than once.
    if (_disposed)
      return;
    if (isDisposing)
    {
      // TODO: free managed resources here.
    }
    // TODO: free unmanaged resources here.
    // Let the base class free its resources.
    // Base class is responsible for calling
    // GC.SuppressFinalize()
    base.Dispose(isDisposing);
    // Set derived class disposed flag:
    _disposed = true;
  }
}
```

注和意，派生类和基类都有一个处理状态的标记，这完全是被动的。重制的标记掩盖了在处理时任何可能发生的错误，而且是单一的类型处理，而不是处理构成这个对象的所有类型。(译注：就是基类与子类各自标记一个，互不影响。)

你应该被动的写处理方法和析构函数，处理对象可能以任何顺序发生，你可能会遇到这种情况：你的类中某个成员在你调用 Dispose 方法以前已经被处理过了。你没有看到这种情况是因为 Dispose()方法是可以多次调用的。如果在一个已经被处理过的对象上调用该方法，就什么也不发生。析构函数也有同样的规则。任何对象的引用存在于内存中时，你不用检测 null 引用。然而，你引用的对象可能已经处理掉了，或者它已经析构了。

这就引入用了一个非常重要的忠告：对于任何与处理和资源清理相关的方法，你必须只释放资源！不要在处理过程中添加其它任何的任务。你在处理和清理中添加其它任务时，可能会在对象的生存期中遇到一些严重而繁杂的问题。对象在你创建它时出生，在垃圾回收器认领它时死亡。你可以认为当你的程序不能再访问它们时，它们是睡眠的。你无法访问对象，无法调用对象的方法。种种迹象表明，它们就像是死的。但对象在宣布死亡前，析构函数还有最后一气。析构函数什么也不应该做，就是清理非托管资源。如果析构函数通过某些方法让对象又变得可访问，那么它就复活了。(译注：析构函数不是用户调用的，也不由.Net 系统调用，而是在由 GC 产生的额外线程上运行的)它又活了，但这并不好。即使是它是从睡眠中唤醒的。这里有一个明显的例子：

```
public class BadClass
{
  // Store a reference to a global object:
  private readonly ArrayList _finalizedList;
  private string _msg;
  public BadClass(ArrayList badList, string msg)
  {
    // cache the reference:
    _finalizedList = badList;
    _msg = (string)msg.Clone();
```

```
  }
  ~BadClass()
  {
    // Add this object to the list.
    // This object is reachable, no
    // longer garbage. It's Back!
    _finalizedList.Add(this);
  }
}
```

当一个 BadClass 对象的析构函数执行时，它把自己的一个引用添加到了全局的链表中。这使得它自己又是可达的，它就又活了。前面向你介绍的这个方法会遇到一些让人畏缩的难题。对象已经被析构了，所以垃圾回收器从此相信再也不用调用它的析构函数了。如果你实际要析构一个可达对象，这将不会成功。其次，你的一些资源可能不再有用。GC 不再从内存上移除那些只被析构队列引用的对象，但它们可能已经析构了。如果是这样，它们很可能已经不能使用了。(译注：也就是说利用上面的那个方法让对象复活后，很有可能对象是不可用的。)尽管 BadClass 所拥有的成员还在内存里，它们像是可以被析构或者处理，但 C#语言没有一个方法可以让你控制析构的次序，你不能让这样的结构可靠的运行。不要尝试。

我还没有看到这样的代码：用这样明显的方式来复活一个对象，除非是学术上的练习。但我看过这样的代码，析构函数试图完成一些实质的工作，最后还通过析构函数的调用把引用放到对象中，从而把自己复活。析构函数里面的代码看上去是精心设计的，另外还有处理函数里的。再检查一遍，这些代码是做了其它事情，而不是释放资源!这些行为会为你的应用程序在后期的运行中产生很多 BUG。删除这些方法，确保析构函数和 Dispose()方法除了清理资源外，什么也不做。

在托管环境里，你不用为每一个创建的类写析构函数；只有须要释放一些使用的非托管资源时才添加，或者你的类所包含的成员有实现了 IDisposable 接口的时候也要添加。即使如此，你也只用实现 IDisposable 接口完成所有的功能就行了，不用析构函数。否则，你会限制你的派生类实现实现标准的 Dispose 习惯。 遵守这个我所讲叙的标准的 Dispose 习惯。这会让你的程序生活变得轻松，也为你的用户，也为那些从你的类创建派生类的人。

第三章 用 C#表达你的设计

Expressing Designs with C#

C#语言为你的设计介绍了一种新的语法，你可以选择这种技术让你的设计与其它的开发人员进行交流，这些开发人员可以维护，扩展或者是使用你设计的软件。C#的所有类型都是生存在.Net 环境下的，这个环境对于所有类型的兼容性做了一些好的假设。但如果你违反了这些假设，你就增加了类型不能正确工作的可能性。

这些原则不是那些已经出版了的软件设计技术的概要，相反，这些原则醒目的给出了怎样用不同的 C#语言特性来表达最好的软件设计意图。C#语言的设计者添加了一些语言特性，用来更清楚的表达现代的软件设计习惯。具体的语言特性有什么杰出的地方是很微妙的，并且你经常会有很多可选的方法。更多的选择在一开始可能只是更好的方案，但随后，语言特性杰出的地方会展示出来，你会发现你有必要增强已经存在的程序。确保你能很好的理解这些原则，并且小心的应用它们，在你创建自己的系统时，注意那些最有可能增强(设计)的机会。

一些语法的改变会给你一些新的词汇，用来描述一些你每天都要使用的习惯。属性，索引器，事件，以及代理就是例子。它们与类和接口是不同的：类定义类型，接口声明行为。基类声明类型，并且为一些放在一起的相关类型定义一些常用的行为。因为垃圾回收器，另一些设计习惯已也经改变了；还有一些改变，是因为大多数变量是引用类型。

这一章里推荐的原则会帮助你为你的设计选择一些最自然的表达方法。这将让你能设计出更容易维护，更容易扩展，并且更容易使用的软件。

原则 19：选择定义和实现接口，而不是继承

Prefer Defining and Implementing Interfaces to Inheritance

抽象类在类的继承中提供了一个常规的“祖先”。一个接口描述了一个可以被其它类型实现的原子级泛型功能。各有千秋，却也不尽相同。接口是一种合约式设计：一个类型实现了某个接口的类型，就必须实现某些期望的方法。抽象类则是为一个相关类的集合提供常规的抽象方法。这些都是老套的东西了：它是这样的，继承就是说它是某物(is a,)，而接口就是说它有某个功能(behaves like.)！这些陈词滥调已经说了好久了，因为它们提供了说明，同时在两个结构上描述它们的不同：基类是描述对象是什么，接口描述对象有某种行为。

接口描述了一组功能集合，或者是一个合约。你可以在接口里创建任何的占位元素(placeholder，译注：就是指先定义，后面再实现的一些内容)：方法，属性，索引器以及事件。任何实现类型这个接口的类型必须为接口里的每个元素提供具体的内容。你必须实现所有的方法，提供全部属性访问器，索引器，以及定义接口里的所有事件。你在接口里标记并且构造了可重用的行为。你可以把接口当成参数或者返回值，你也可以有更多的机会重用代码，因为不同的类型可以实现相同的接口。更多的是，比起从你创建的基类派生，开发人员可以更容易的实现接口。(译注：不见得!)

你不能在接口里提供任何成员的具体实现，无论是什么，接口里面都不能实现。并且接口也不能包含任何具体的数据成员。你是在定义一个合约，所有实现接口的类型都应该实现的合约。

抽象的基类可以为派生类提供一些具体的实现，另外也描述了一些公共的行为。你可以更详细的说明数据成员，具体方法，实现虚函数，属性，事件以及索引器。一个基类可以只提供部份方法的实现，从而只提供一些公共的可重用的具体实现。抽象类的元素可以是虚的，抽象的，或者是非虚的。一个抽象类可以为具体的行为提供一个可行的实现，而接口则不行。

重用这些实现还有另一个好处：如果你在基类中添加一个方法，所有派生类会自动隐式的增加了这个方法。这就是说，基类提供了一个有效的方法，可以随时扩展几个(派生)类型的行为：就是向基类添加并实现方法，所有派生类会立即具有这些行为。而向一个接口添加一个方法，所会破坏所有原先实现了这个接口的类。这些类不会包含新的方法，而且再也通不过编译。所有的实现者都必须更新，要添加新的方法。

这两个模式可以混合并重用一些实现代码，同时还可以实现多个接口。System.Collections.CollectionBase 就是这样的一个例子，它个类提供了一个基类。你可以用这个基类你的客户提供一些.Net 缺少的安全集合。例如，它已经为你实现了几个接口：IList, ICollection,和 IEnumerable。另外，它提供了一个受保护的方法，你可以重载它，从而为不同的使用情况提供自己定义的行为。IList 接口包含向集合中添加新对象的 Insert()方法。想自己更好的提供一个 Insert 方法的实现，你可以通过重载 CollectionBase 类的 OnInsert()或 OnInsertCcomplete()虚方法来处理这些事件：

```
public class IntList : System.Collections.CollectionBase
{
  protected override void OnInsert(int index, object value)
  {
    try
    {
      int newValue = System.Convert.ToInt32(value);
      Console.WriteLine("Inserting {0} at position {1}",
        index.ToString(), value.ToString());
        Console.WriteLine("List Contains {0} items",
        this.List.Count.ToString());
    }
    catch(FormatException e)
    {
      throw new ArgumentException(
```

```
        "Argument Type not an integer",
        "value", e);
      }
    }
    protected override void OnInsertComplete(int index,
      object value)
    {
      Console.WriteLine("Inserted {0} at position {1}",
        index.ToString(), value.ToString());
      Console.WriteLine("List Contains {0} items",
        this.List.Count.ToString());
    }
  }
  public class MainProgram
  {
    public static void Main()
    {
      IntList l = new IntList();
      IList il = l as IList;
      il.Insert(0,3);
      il.Insert(0, "This is bad");
    }
  }
```

前面的代码创建了一个整型的数组链表，而且使用 IList 接口指针添加了两个不同的值到集合中。通过重载 OnInsert()方法，IntList 类在添加类型时会检测类型，如果不是一个整数时，会抛出一个异常。 基类给你实现了默认的方法，而且给我们提供了机会在我们自己的类中实现详细的行为。

CollectionBase 这个基类提供的一些实现可以直接在你的类中使用。你几乎不用写太多的代码，因为你可以利用它提供的公共实现。但 IntList 的公共 API 是通过 CollectionBase 实现接口而来的：IEnumerable,ICollection 和 IList 接口。CollectionBase 实现了你可以直接使用的接口。

现在我们来讨论用接口来做参数和返回值。一个接口可以被任意多个不相关的类型实现。比起在基类中编码，实现接口的编码可以在开发人员中提供更强的伸缩性。因为.Net 环境中强制使用单继承的，这使得实现接口这一方法显得很重要。

下面两个方法完成了同样的任务：

```
public void PrintCollection(IEnumerable collection)
{
  foreach(object o in collection)
  Console.WriteLine("Collection contains {0}",
    o.ToString());
}
public void PrintCollection(CollectionBase collection)
{
  foreach(object o in collection)
  Console.WriteLine("Collection contains {0}",
    o.ToString());
```

```
}
```

第二个方法的重用性远不及第一个。Array，ArrayList，DataTable，HashTable，ImageList 或者很多其它的集合类无法使用第二个方法。让方法的参数使用接口，可以让程序具有通用性，而且更容易重用。

用接口为类定义 API 函数同样可以取得很好的伸缩性。例如，很多应用程序使用 DataSet 与你的应用程序进行数据交换。假设这一交流方法是不变的，那这太容易实现了：

```
public DataSet TheCollection
{
  get { return _dataSetCollection; }
}
```

然而这让你在将来很容易遇到问题。某些情况下，你可能从使用 DataSet 改为暴露一个 DataTable，或者是使用 DataView，甚至是使用你自己定义的对象。任何的改变都会破坏这些代码。当然，你可能会改变参数类型，但这会改变你的类的公共接口。修改一个类的公共接口意味着你要在一个大的系统中修改更多的内容；你须要修改所有访问这个公共接口的地方。

紧接着的第二个问题麻烦问题就是：DataSet 类提供了许多方法来修改它所包含的数据。类的用户可能会删除表，修改列，甚至是取代 DataSet 中的所有对象。几乎可以肯定这不是你想要的。幸运的是，你可以对用户限制类的使用功能。不返回一个 DataSet 的引用，你就须要返回一个期望用户使用的接口。DataSet 支持 IListSource 接口，它用于数据绑定：

```
using System.ComponentModel;
public IListSource TheCollection
{
  get { return _dataSetCollection as IListSource; }
}
```

IListSource 让用户通过 GetList()方法来访问内容，它同时还有 ContainsListCollection 属性，因此用户可以修改全部的集合结构。使用 IListSource 接口，在 DataSet 里的个别对象可以被访问，但 DataSet 的所有结构不能被修改。同样，调用者不能使用 DataSet 的方法来修改可用的行为，从而在数据上移动约束或者添加功能。

当你的类型以类的方式暴露一些属性时，它就暴露了这个类的全部接口。使用接口，你可以选择只暴露一部分你想提供给用户使用的方法和属性。以前在类上实现接口的详细内容，在后来是可以修改的(参见原则 23)。

另外，不相关的类型可以实现同样的接口。假设你在创建一个应用程序。用于管理雇员，客户和卖主。他们都不相关，至少不存在继承关系。但他们却共享着某些功能。他们都有名字，而且很有可能要在一些 Windows 控件中显示他们的名字：

```
public class Employee
{
  public string Name
  {
    get
    {
      return string.Format("{0}, {1}", _last, _first);
    }
  }
  // other details elided.
}
public class Customer
{
  public string Name
  {
```

```
    get
    {
      return _customerName;
    }
  }
  // other details elided
}
public class Vendor
{
  public string Name
  {
    get
    {
      return _vendorName;
    }
  }
}
```

Eyployee,Customer 和 Vendor 类不应该共享一个基类。但它们共享一些属性：姓名(正如前面显示的那样)，地址，以及联系电话。你应该在一个接口中创建这些属性：

```
public interface IContactInfo
{
  string Name { get; }
  PhoneNumber PrimaryContact { get; }
  PhoneNumber Fax { get; }
  Address PrimaryAddress { get; }
}
public class Employee : IContactInfo
{
  // implementation deleted.
}
```

对于不的类型使用一些通用的功能，接口可以简化你的编程任务。Customer, Employee, 和 Vendor 使用一些相同的功能，但这只是因为你把它们放在了接口上。

使用接口同样意味着在一些意外情况下，你可以减少结构类型拆箱的损失。当你把一个结构放到一个箱中时，这个箱可以实现结构上的所有接口。当你用接口指针来访问这个结构时，你不用结构进行拆箱就可以直接访问它。这有一个例子，假设这个结构定义了一个链接和一些说明：

```
public struct URLInfo : IComparable
{
  private string URL;
  private string description;
  public int CompareTo(object o)
  {
    if (o is URLInfo)
    {
      URLInfo other = (URLInfo) o;
```

```
      return CompareTo(other);
    }
    else
      throw new ArgumentException(
        "Compared object is not URLInfo");
  }
  public int CompareTo(URLInfo other)
  {
    return URL.CompareTo(other.URL);
  }
}
```

你可以为 URLInfo 的对象创建一个有序表，因为 URLInfo 实现了 IComparable 接口。URLInfo 结构会在添加到链表中时被装箱，但 Sort()方法不须要拆箱就可以调用对象的 CompareTo()方法。你还须要对参数(other)进行拆箱，但你在调用 IComparable.CompareTo()方法时不必对左边的对象进行拆箱。

基类可以用来描述和实现一些具体的相关类型的行为。接口则是描述一些原子级别的功能块，不相关的具体类型都可以实现它。接口以功能块的方法来描述这些对象的行为。如果你明白它们的不同之处，你就可以创建出表达力更强的设计，并且它们面对修改是有很加强的伸缩性的。类的继承可以用来定义一些相关类型。通过实现一些接口来暴露部份功能来访问这些类型。

原则 20：明辨接口实现和虚函数重载的区别

Distinguish Between Implementing Interfaces and Overriding Virtual Functions

粗略的看一下，感觉实现接口和虚函数重载是一样的。你定义了一些对象，但是这些对象是在另一个类型里申明的。你被第一感觉骗了，实现接口与虚函数重载是完全不同的。在接口里定义的成员默认情况下，是根本不存在实际内容的。

派生类不能重载基类中的接口成员。接口可以隐式的实现，就是把它们从类的公共接口中隐藏。它们的概念是不同的而且使用也是不同的。

但你可以这样的实现接口：让你的派生类可以修改你的实现。你只用对派生类做一个 Hook 就行了。(译注：相信写过 C++程序的人就知道 hook 是什么意思，而且我也实在想不到把 hook 译成什么比较好，所以就直接用 hook 这个原词了，就像 bug 一样。)

为了展示它们的不同之处，试着做一个简单的接口以及在一个类中实现它：

```
interface IMsg
{
  void Message();
}
public class MyClass : IMsg
{
  public void Message()
  {
    Console.WriteLine("MyClass");
  }
}
```

Message()方法是 MyClass 的公共接口，Message 同样可以用一个接口指针 IMsg 来访问。现在让我们来一点繁杂的，添加一个派生类：

```
public class MyDerivedClass : MyClass
```

```
{
  public new void Message()
  {
    Console.WriteLine("MyDerivedClass");
  }
}
```

注意到，我添加了一个关键字 new 在 Message 方法上，用于区别前面的一个 Message(参见原则 29)。MyClass.Message()不是虚函数，派生类可以不提供重载版本。MyDerived 类创建了一个新的 Message 方法，但这个方法并不是重载 MyClass.Message:它隐藏了原来的方法。而且，MyClass.Message 还是可以通过 IMsg 的引用来访问：

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // prints "MyDerivedClass".
IMsg m = d as IMsg;
m.Message(); // prints "MyClass"
```

接口方法不是虚的，当你实现一个接口时，你就要在详细的相关类型中申明具体的实现内容。

但你可能想要创建接口，在基类中实现这些接口而且在派生类中修改它们的行为。这是可以办法到的。你有两个选择，如果不访问基类，你可以在派生类中重新实现这个接口：

```
public class MyDerivedClass : MyClass, IMsg
{
  public new void Message()
  {
    Console.WriteLine("MyDerivedClass");
  }
}
```

添加的 IMsg 让你的派生类的行为发生了改变，以至 IMsg.Message 现在是在派生类上使用的：

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // prints "MyDerivedClass".
IMsg m = d as IMsg;
m.Message(); // prints "MyDerivedClass"
```

派生类上还是须要在 MyDerivedClass.Message()方法上添加关键字 new，这还是有一点隐患(参见原则 29)。基类还是可以通过接口引用来访问：

```
MyDerivedClass d = new MyDerivedClass();
d.Message(); // prints "MyDerivedClass".
IMsg m = d as IMsg;
m.Message(); // prints "MyDerivedClass"
MyClass b = d;
b.Message(); // prints "MyClass"
```

唯一可以修正这个问题的方法是修改基类，把接口的申明修改为虚函数：

```
public class MyClass : IMsg
{
  public virtual void Message()
  {
    Console.WriteLine("MyClass");
  }
}
```

```
public class MyDerivedClass：MyClass
{
  public override void Message()
  {
    Console.WriteLine("MyDerivedClass");
  }
}
```

MyDerivedClass 以及其它所有从 MyClass 派生的类可以申明它们自己的 Message()方法。这个重载的版本每次都会调用：通过 MyDerivedClass 的引用，通过 IMsg 接口的引用，或者直接通过 MyClass 的引用。

如果你不喜欢混杂的虚函数概念，那就对 MyClass 的定义做一个小的修改：

```
public abstract class MyClass, IMsg
{
  public abstract void Message();
}
```

是的，你可以用一个抽象方法来实现一个接口。通过申明一个接口内的抽象的方法，你可以让你的所有派生都必须实现这个接口。现在，IMsg 接口成为了 MyClass 的一个组成部份，你的每一个派生类都必须实现它。

隐式接口实现，可以让你在一个类上隐藏公共的接口成员方法，而且也实现了这个接口。它在实现接口和虚函数重载上绕了几个圈。当有多个合适的函数版本时，你可以利用隐式接口的实现来限制用户的编码。在原则 26 讲到的 IComparable 习惯会详细的讨论这一点。

实现接口让我们有更多的选择，用于创建和重载虚函数。你可以创建隐秘的实现，虚的实现，或者抽象关联到派生类。你可以精确的决定，你的派生类如何以及何时，修改接口的默认实现。接口方法不是虚方法，而是一个独立的约定!

原则 21：用委托来表示回调

Express Callbacks with Delegates

我：“儿子，到院子里除草去，我要看会书。”

斯科特：“爸，我已经打扫过院子了。”

斯科特：“爸，我已经把草放在除草机上了。”

斯科特：“爸，除草机不能启动了。”

我：“让我来启动它。”

斯科特：“爸，我做好了。”

这个简单的交互展示了回调。我给了我儿子一个任务，并且他可以报告状态来(重复的)打断我。而当我在等待他完成任务的每一个部份时，我不用阻塞我自己的进程。他可以在有重要(或者事件)状态报告时，可以定时的打断我，或者向我询求帮助。回调就是用于异步的提供服务器与客户之间的信息反馈。它们可能在多线程中，或者可能是简单的提供一个同步更新点。在 C#里是用委托来表示回调的。

委托提供了一个类型安全的回调定义。尽管委托大多数是为事件使用的，但这不应该是 C#语言中唯一使用这一功能的地方。任何时候，如果你想在两个类之间进行通信，而你又期望比使用接口有更少的偶合性，那么委托是你正确的选择。委托可以让你在运行确定(回调)目标并且通知用户。委托就是包含了某些方法的引用。这些方法可以是静态方法，也可以是实例方法。使用委托，你可以在运行时确定与一个或者多个客户对象进行交互。

多播委托包含了添加在这个委托上的所有单个函数调用。有两点要注意的：它不是异常安全的，并且返回值总是委托上最后一个函数调用后返回的值。

在多播委托调用的内部，每一个目标都会成功的调用。委托不会捕获任何的异常，也就是说，在委托链中抛出的任何异常都会终止委托链的继续调用。

在返回值上也存在一个简单的问题。你可以定义委托有返回值或者是 void。你可能会写一个回调函数来检测用

户的异常中断：

```
public delegate bool ContinueProcessing();
public void LengthyOperation(ContinueProcessing pred)
{
  foreach(ComplicatedClass cl in _container)
  {
    cl.DoLengthyOperation();
    // Check for user abort:
    if (false == pred())
      return;
  }
}
```

在单委托上这是工作的，但在多播委托上却是有问题的：

```
ContinueProcessing cp = new ContinueProcessing (
  CheckWithUser);
cp += new ContinueProcessing(CheckWithSystem);
c.LengthyOperation(cp);
```

从委托的调用上返回的值，其实是它的最后一个函数的调用上返回的值。其它所有的的返回值都被忽略。即，从 CheckWithUser()返回的断言被忽略。

你可以自己手动的设置两个委托来调用两个函数。你所创建的每一个委托都包含有一个委托链。直接检测这个委托链，并自己调用每一个委托：

```
public delegate bool ContinueProcessing();
public void LengthyOperation(ContinueProcessing pred)
{
  bool bContinue = true;
  foreach(ComplicatedClass cl in _container)
  {
    cl.DoLengthyOperation();
    foreach(ContinueProcessing pr in
      pred.GetInvocationList())
      bContinue &= pr();
    if (false == bContinue)
      return;
  }
}
```

这时，我已经定义好了程序的语义，因此委托链上的每个委托必须返回真以后，才能继续调用。

委托为运行时回调提供了最好的方法，用户简单的实现用户对类的需求。你可以在运行时确定委托的目标。你可以支持多个用户目标，这样，用户的回调就可以用.Net 里的委托实现了。

原则 22：用事件定义对外接口

Define Outgoing Interfaces with Events

可以用事件给你的类型定义一些外部接口。事件是基于委托的，因为委托可以提供类型安全的函数签名到事件句柄上。加上大多数委托的例子都是使用事件来说明的，以至于开发人员一开始都认为委托与事件是一回事。在原则 21 里，我已经展示了一些不在事件上使用委托的例子。在你的类型与其它多个客户进行通信时，为了完成它们的

行为，你必须引发事件。

一个简单的例子，你正在做一个日志类，就像一个信息发布机一样在应用程序里发布所有的消息。它接受所有从程序源发布的消息，并且把这些消息发布到感兴趣的听众那里。这些听众可以是控制台，数据库，系统日志，或者是其它的机制。就可以定义一个像下面这样的类，当消息到达时来引发事件：

```
public class LoggerEventArgs : EventArgs
{
  public readonly string Message;
  public readonly int Priority;
  public LoggerEventArgs (int p, string m)
  {
    Priority = p;
    Message = m;
  }
}
// Define the signature for the event handler:
public delegate void AddMessageEventHandler(object sender,
  LoggerEventArgs msg);
public class Logger
{
  static Logger()
  {
    _theOnly = new Logger();
  }
  private Logger()
  {
  }
  private static Logger _theOnly = null;
  public Logger Singleton
  {
    get
    {
      return _theOnly;
    }
  }
  // Define the event:
  public event AddMessageEventHandler Log;
  // add a message, and log it.
  public void AddMsg (int priority, string msg)
  {
    // This idiom discussed below.
    AddMessageEventHandler l = Log;
    if (l != null)
      l (null, new LoggerEventArgs(priority, msg));
  }
}
```

AddMsg 方法演示了一个恰当的方法来引发事件。临时的日志句柄变量 是很重要的，它可以确保在各种多线程的情况下，日志句柄也是安全的。如果没有这个引用的 COPY，用户就有可能在 if 检测语句和正式执行事件句柄之间移除事件句柄。有了引用 COPY，这样的事情就不会发生了。

我还定义了一个 LoggerEventArgs 来保存事件和消息的优先级。委托定义了事件句柄的签名。而在 Logger 类的内部，事件字段定义了事件的句柄。编译器会认为事件是公共的字段，而且会为你添加 Add 和 Remove 两个操作。生成的代码与你这样手写的是一样的：

```
public class Logger
{
  private AddMessageEventHandler _Log;
  public event AddMessageEventHandler Log
  {
    add
    {
      _Log = _Log + value;
    }
    remove
    {
      _Log = _Log - value;
    }
  }
    public void AddMsg (int priority, string msg)
    {
      AddMessageEventHandler l = _Log;
      if (l != null)
        l (null, new LoggerEventArgs (priority, msg));
    }
  }
}
```

C#编译器创建 Add 和 Remove 操作来访问事件。看到了吗，公共的事件定义语言很简洁，易于阅读和维护，而且更准确。当你在类中添加一个事件时，你就让编译器可以创建添加和移除属性。你可以，而且也应该，在有原则要强制添加时自己手动的写这些句柄。

事件不必知道可能成为监听者的任何资料，下面这个类自动把所有的消息发送到标准的错误设备(控制台)上：

```
class ConsoleLogger
{
  static ConsoleLogger()
  {
    logger.Log += new AddMessageEventHandler(Logger_Log);
  }
  private static void Logger_Log(object sender,
    LoggerEventArgs msg)
  {
    Console.Error.WriteLine("{0}:\t{1}",
      msg.Priority.ToString(),
      msg.Message);
  }
```

```
}
```

另一个类可以直接输出到系统事件日志：

```
class EventLogger
{
  private static string eventSource;
  private static EventLog logDest;
  static EventLogger()
  {
    logger.Log +=new AddMessageEventHandler(Event_Log);
  }
  public static string EventSource
  {
    get
    {
      return eventSource;
    }
    set
    {
      eventSource = value;
      if (! EventLog.SourceExists(eventSource))
        EventLog.CreateEventSource(eventSource,
          "ApplicationEventLogger");
      if (logDest != null)
        logDest.Dispose();
      logDest = new EventLog();
      logDest.Source = eventSource;
    }
  }
  private static void Event_Log(object sender,
    LoggerEventArgs msg)
  {
    if (logDest != null)
      logDest.WriteEntry(msg.Message,
        EventLogEntryType.Information,
        msg.Priority);
  }
}
```

事件会在发生一些事情时，通知任意多个对消息感兴趣的客户。Logger 类不必预先知道任何对消息感兴趣的对象。

Logger 类只包含一个事件。大多数 windows 控件有很多事件，在这种情况下，为每一个事件添加一个字段并不是一个可以接受的方法。在某些情况下，一个程序中只实际上只定义了少量的事件。当你遇到这种情况时，你可以修改设计，只有在运行时须要事件时在创建它。

(译注：作者的一个明显相思就是，当他想说什么好时，就决不会，或者很少说这个事情的负面影响。其实事件对性能的影响是很大的，应该尽量少用。事件给我们带来的好处是很多的，但不要海滥用事件。作者在这里没有明说事件的负面影响。)

扩展的 Logger 类有一个 System.ComponentModel.EventHandlerList 容器，它存储了在给定系统中应该引发的事件对象。更新的 AddMsg()方法现在带一个参数，它可以详细的指示子系统日志的消息。如果子系统有任何的监听者，事件就被引发。同样，如果事件的监听者在所有感兴趣的消息上监听，它同样会被引发：

```
public class Logger
{
  private static System.ComponentModel.EventHandlerList
    Handlers = new System.ComponentModel.EventHandlerList();
  static public void AddLogger(
    string system, AddMessageEventHandler ev)
  {
    Handlers[ system ] = ev;
  }
  static public void RemoveLogger(string system)
  {
    Handlers[ system ] = null;
  }
  static public void AddMsg (string system,
    int priority,  string msg)
  {
    if ((system != null) && (system.Length > 0))
    {
      AddMessageEventHandler l =
        Handlers[ system ] as AddMessageEventHandler;
      LoggerEventArgs args = new LoggerEventArgs(
        priority, msg);
      if (l != null)
        l (null, args);
      // The empty string means receive all messages:
      l = Handlers[ "" ] as AddMessageEventHandler;
      if (l != null)
        l(null, args);
    }
  }
}
```

这个新的例子在 Event HandlerList 集合中存储了个别的事件句柄，客户代码添加到特殊的子系统中，而且新的事件对象被创建。然后同样的子系统需要时，取回同样的事件对象。如果你开发一个类包含有大量的事件实例，你应该考虑使用事件句柄集合。当客户附加事件句柄时，你可以选择创建事件成员。在.Net 框架内部，System.Windows.Forms.Control 类对事件使用了一个复杂且变向的实现，从而隐藏了复杂的事件成员字段。每一个事件字段在内部是通过访问集合来添加和移除实际的句柄。关于 C#语言的这一特殊习惯，你可以在原则 49 中发现更多的信息。

你用事件在类上定义了一个外接的接口：任意数量的客户可以添加句柄到事件上，而且处理它们。这些对象在编译时不必知道是谁。事件系统也不必知道详细就可以合理的使用它们。在 C#中事件可以减弱消息的发送者和可能的消息接受者之间的关系，发送者可以设计成与接受者无关。事件是类型把动作信息发布出去的标准方法。

原则 23：避免返回内部类对象的引用

Avoid Returning References to Internal Class Objects

你已经知道，所谓的只读属性就是指调用者无法修改这个属性。不幸运的是，这并不是一直有效的。如果你创建了一个属性，它返回一个引用类型，那么调用者就可以访问这个对象的公共成员，也包括修改这些属性的状态。例如：

```
public class MyBusinessObject
{
  // Read Only property providing access to a
  // private data member:
  private DataSet _ds;
  public DataSet Data
  {
    get
    {
      return _ds;
    }
  }
}
// Access the dataset:
DataSet ds = bizObj.Data;
// Not intended, but allowed:
ds.Tables.Clear(); // Deletes all data tables.
```

任何 MyBusinessObject 的公共客户都可以修改你的内部 dateset。你创建的属性用来隐藏类的内部数据结构，你提供了方法，让知道该方法的客户熟练的操作数据。因此，你的类可以管理内部状态的任何改变。然而，只读属性对于类的封装来说开了一个后门。当你考虑这些问题时，它并不是一个可读可写属性，而是一个只读属性。

欢迎来到一个精彩的基于引用的系统，任何返回引用的成员都会返回一个对象的句柄。你给了调用者一个接口的句柄，因此调用者修改这个对象的某个内部引用时，不再需要通过这个对象。

很清楚，你想防止这样的事情发生。你为你的类创建了一个接口，同时希望用户使用这个接口。你不希望用户在不明白你的意图时，访问并修改对象的内部状态。你有四个策略来保护你的内部数据结构不被无意的修改：值类型，恒定类型，接口和包装(模式)。

值类型在通过属性访问时，是数据的拷贝。客户对类的拷贝数据所做的任何修改，不会影响到对象的内部状态。客户可以根据需求随意的修改拷贝的数据。这对你的内部状态没有任意影响。

恒定类型，例如 System.String，也是安全的。你可以返回一个字符串，或者其它恒定类型。恒定类型的安全性告诉你，没有客户可以修改字符串。你的内部状态是安全的。

第三个选择就是定义接口，从而允许客户访问内部成员的部份功能(参见原则 19)。当你创建一个自己的类时，你可以创建一些设置接口，用来支持对类的子对象进行设置。通过这些接口来暴露一些功能函数，你可以尽可能的减少一些对数据的无意修改。客户可以通过你提供的接口访问类的内部对象，而这个接口并不包含这个类的全部的功能。在 DataSet 上暴露一个 IListsource 接口就是这种策略，可以阻止一些有想法的程序员来猜测实现这个接口的对象，以及强制转换。这样做和程序员付出更多的工作以及发现更多的 BUG 都是自找的(译注：这一句理解可能完全不对，读者可以自行参考原文：But programmers who go to that much work to create bugs get what they deserve.)。

System.Dataset 类同时也使用了最后一种策略：包装对象。DataViewManager 类提供了一种访问 DataSet 的方法，而且防止变向的方法来访问 DataSeto 类：

```
public class MyBusinessObject
```

```
{
  // Read Only property providing access to a
  // private data member:
  private DataSet _ds;
  public DataView this[ string tableName ]
  {
    get
    {
      return _ds.DefaultViewManager.
        CreateDataView(_ds.Tables[ tableName ]);
    }
  }
}
// Access the dataset:
DataView list = bizObj[ "customers" ];
foreach (DataRowView r in list)
  Console.WriteLine(r[ "name" ]);
```

DataViewManager 创建 DataView 来访问 DataSet 里的个别数据表。DataViewManager 没有提供任何方法来修改 DataSet 里的数据表。每一个 DataView 可以被配置为许可修改个别数据元素，但客户不能修改数据表，或者数据表的列。读/写是默认的，因此客户还是可以添加，修改，或者删除个别的数据条目。

在我们开始讨论如何创建一个完全只读的数据视图时以前，让我先简单的了解一下你应该如何响应公共用户的修改。这是很重要的，因为你可能经常要暴露一个 DataView 给 UI 控件，这样用户就可以编辑数据(参见原则 38)。确信你已经使用过 Windows 表单的数据绑定，用来给用户提供对象私有数据编辑。DataSet 里的 DataTable 引发一些事件，这样就可以很容易的实现观查者模式：你的类可以响应其它客户的任何修改。DataSet 里的 DataTable 对象会在数据表的任何列以及行发生改变时引发事件。ColumnChanging 和 RowChanging 事件会在编辑的数据提交到 DataSet 前被引发。而 ColumnChanged 和 RowChanged 事件则是在修改提交后引发。

任何时候，当你期望给公共客户提供修改内部数据的方法时，都可以扩展这样的技术，但你要验证而且响应这些改变。你的类应该对内部数据结构产生的事件做一些描述。事件句柄通过更新这些内部的状态来验证和响应改变。

回到原来的问题上，你想让客户查看你的数据，但不许做任何的修改。当你的数据存储在一个 DataSet 里时，你可以通过强制在 DataTable 上创建一个 DataView 来防止任何的修改。DataView 类包含一些属性，通过定义这些属性，可以让 DataView 支持在实际的表上添加，删除，修改甚至是排序。你可以在被请求的 DataTable 上使用索引器，通过创建一个索引器来返回一个自定义的 DataView：

```
public class MyBusinessObject
{
  // Read Only property providing access to a
  // private data member:
  private DataSet _ds;
  public IList this[ string tableName ]
  {
    get
    {
      DataView view =
        _ds.DefaultViewManager.CreateDataView
        (_ds.Tables[ tableName ]);
      view.AllowNew = false;
```

```
        view.AllowDelete = false;
        view.AllowEdit = false;
        return view;
      }
    }
  }
  // Access the dataset:
      IList dv = bizOjb[ "customers" ];
      foreach (DataRowView r in dv)
        Console.WriteLine(r[ "name" ]);
```

这个类的最后一点摘录(的代码)通过访问 IList 接口引用，返回这个实际数据表上的视图。你可以在任何的集合上使用 IList 接口，并不仅限于 DataSet。你不应该只是简单的返回 DataView 对象。用户可以再次简单的取得编辑，添加/删除的能力。你返回的视图已经是自定义的，它不许可在列表的对象上做任何的修改。返回的 IList 指针确保客户没有像 DataView 对象里赋于的修改权利。

从公共接口上暴露给用户的引用类型，可以让用户修改对象内部成员，而不用访问该对象。这看上去不可思议，也会产生一些错误。你须要修改类的接口，重新考虑你所暴露的是引用而不是值类型。如果你只是简单的返回内部数据，你就给了别人机会去访问内部成员。你的客户可以调用成员上任何可用的方法。你可以通过暴露接口来限制一些内部私有数据访问，或者包装对象。当你希望你的客户可以修改你的内部数据时，你应该实现你自己的观察者模式，这样你的对象可以验证修改或者响应它们。

原则 24：选择申明式编程而不是命令式编程

Prefer Declarative to Imperative Programming

与命令式编程相比，申明式编程可以用更简单，更清楚的方法来描述软件的行为。申明式编程就是说用申明来定义程序的行为，而不是写一些指令。在 C#里，也和其它大多数语言一样，你的大多数程序都是命令式的：在程序中写一个方法来定义行为。在 C#中，你在编程时使用特性就是申明式编程。你添加一个特性到类，属性，数据成员，或者是方法上，然后.Net 运行时就会为你添加一些行为。这样申明的目的就是简单易用，而且易于阅读和维护。

让我们以一个你已经使用过的例子开始。当你写你的第一个 ASP.Net Web 服务时，向导会生成这样的代码：

```
[WebMethod]
public string HelloWorld()
{
  return "Hello World";
}
```

VS.net 的 Web 服务向导添加了[WebMethod]特性到 HelloWorld()方法上，这就定义了 HelloWorld 是一个 web 方法。ASP.net 运行时会为你生成代码来响应这个特性。运行时生成的 Web 服务描述语言(WSDL)文档，也就是包含了对 SOAP 进行描述的文档，调用 HelloWorld 方法。ASP.net 也支持运行时发送 SOAP 请求 HelloWorld 方法。另外，ASP.net 运行时动态的生成 HTML 面页，这样可以让你在 IE 里测试你的新 Web 服务。而这些全部是前面的 WebMethod 特性所响应的。这个特性申明了你的意图，而且运行时确保它是被支持的。使用特性省了你不少时间，而且错误也少了。

这并不是一个神话，ASP.net 运行时使用反射来断定类里的哪些方法是 web 服务，当它们发现这些方法时，ASP.net 运行时就添加一些必须的框架代码到这些方法上，从而使任何添加了这些代码的方法成为 web 方法。

[WebMethod] 特性只是.Net 类库众多特性之一，这些特性可能帮助你更快的创建正确的程序。有一些特性帮助你创建序列化类型(参见原则 25)。正如你在原则 4 里看到的，特性可以控制条件编译。在这种情况以下其它一些情况下，你可以使用申明式编程写出你所要的更快，更少错误的代码。

你应该使用.Net 框架里自带的一些特性来申明你的意图，这比你自己写要好。因为这样花的时间少，更简单，

而且编译器也不会出现错误。

如果预置的特性不适合你的需求，你也可以通过定义自己的特性和使用反射来使用申明式编程结构。做为一个例子，你可以创建一个特性，然而关联到代码上，让用户可以使用这个特性来创建默认可以排序的类型。一个例子演示了如何添加这个特性，该特性定义了你想如何在一个客户集合中排序：

```
[DefaultSort("Name")]
public class Customer
{
  public string Name
  {
    get { return _name; }
    set { _name = value; }
  }
  public decimal CurrentBalance
  {
    get { return _balance; }
  }
  public decimal AccountValue
  {
    get
    {
      return calculateValueOfAccount();
    }
  }
}
```

DefaultSort 特性，Nane 属性，这就暗示了任何 Customer 的集合应该以客户名字进行排序。DefaultSort 特性不是.Net 框架的一部份，为了实现它，你创建一个 DefaultSortAttribute 类：

```
[AttributeUsage(AttributeTargets.Class |
  AttributeTargets.Struct)]
public class DefaultSortAttribute：System.Attribute
{
  private string _name;
  public string Name
  {
    get { return _name; }
    set { _name = value; }
  }
  public DefaultSortAttribute(string name)
  {
    _name = name;
  }
}
```

同样，你还必须写一些代码，来对一个集合运行排序，而该集合中的元素是添加了 DefaultSort 特性的对象。你将用到反射来发现正确的属性，然后比较两个不同对象的属性值。一个好消息是你只用写一次这样的代码。

下一步，你要写一个实现了 IComparer 接口的类。(在原则 26 中会详细的充分讨论比较。) ICompare 有一个 CompareTo()方法来比较两个给定类型的对象，把特性放在实现了 IComparable 的类上，就可以定义排序顺序

了。构造函数对于通用的比较，可以发现默认的排序属性标记，而这个标记是基于已经比较过的类型。Compare 方法对任何类型的两个对象进行排序，使用默认的排序属性：

```
internal class GenericComparer : IComparer
{
  // Information about the default property:
  private readonly PropertyDescriptor _sortProp;
  // Ascending or descending.
  private readonly bool _reverse = false;
  // Construct for a type
  public GenericComparer(Type t) :
    this(t, false)
  {
  }
  // Construct for a type
  // and a direction
  public GenericComparer(Type t, bool reverse)
  {
    _reverse = reverse;
    // find the attribute,
    // and the name of the sort property:
    // Get the default sort attributes on the type:
    object [] a = t.GetCustomAttributes(
      typeof(DefaultSortAttribute),false);
    // Get the PropertyDescriptor for that property:
    if (a.Length > 0)
    {
      DefaultSortAttribute sortName = a[ 0 ] as   DefaultSortAttribute;
      string name = sortName.Name;
      // Initialize the sort property:
      PropertyDescriptorCollection props =
        TypeDescriptor.GetProperties(t);
      if (props.Count > 0)
      {
        foreach (PropertyDescriptor p in props)
        {
          if (p.Name == name)
          {
            // Found the default sort property:
            _sortProp = p;
            break;
          }
        }
      }
    }
  }
```

```
  // Compare method.
  int IComparer.Compare(object left,
    object right)
  {
    // null is less than any real object:
    if ((left == null) && (right == null))
      return 0;
    if (left == null)
      return -1;
    if (right == null)
      return 1;
    if (_sortProp == null)
    {
      return 0;
    }
    // Get the sort property from each object:
    IComparable lField =
      _sortProp.GetValue(left) as IComparable;
    IComparable rField =
      _sortProp.GetValue(right) as IComparable;
    int rVal = 0;
    if (lField == null)
      if (rField == null)
        return 0;
      else
        return -1;
    rVal = lField.CompareTo(rField);
    return (_reverse) ? -rVal : rVal;
  }
}
```

这个通用的比较对任何 Customers 集合可以进行排序，而这个 Customers 是用 DefaultSort 特性申明了的：

```
CustomerList.Sort(new GenericComparer(
  typeof(Customer)));
```

实现 GenericComparer 的代码利用了一些高级的技术，使用反射(参见原则 43)。但你必须写一遍这样的代码。从这个观点上看，你所要做的就是添加空上属性到其它任何类上，然而你就可以对这些对象的集合进行能用的排序了。如果你修改了 DefaultSort 特性的参数，你就要修改类的行为。而不用修改所有的算法。

这种申明式习惯是很有用的，当一个简单的申明可以说明你的意图时，它可以帮助你避免重复的代码。再参考 GenericComparer 类，你应该可以为你创建的任何类型，写一个不同的(而且是是直接了当的)排序算法。这种申明式编程的好处就是你只用写一次能用的类型，然后就可以用一个简单的申明为每个类型创建行为。关键是行为的改变是基于单个申明的，不是基于任何算法的。GenericComparer 可以在任何用 DefaultSort 特性修饰了的类型上工作，如果你只须要在程序里使用一两次排序功能，就按常规简单的方法写吧。然而，如果你的程序对于同样的行为，可能须要在几十个类型上实现，那么能用的算法以及申明式的解决方案会省下很多时间，而且在长时间的运行中也是很有力的。你不应该为 WebMethod 特性写代全部的代码，你应该把这一技术展开在你自己的算法上。原则 42 里讨论了一个例子：如何使用特性来建立一个附加命令句柄。其它的例子可能还包括一些在定义附加包建立动态的 web UI 面页时的其它内容。

申明式编程是一个很有力的工具，当你可以使用特性来表明你的意图时，你可以通过使用特性，来减少在大量类似的手写算法中出现逻辑错误的可能。申明式编程创建了更易于阅读，清晰的代码。这也就意味着不管是现在还是将来，都会少出现错误。如果你可以使用.Net 框架里定义的特性，那就直接使用。如果不能，考虑选择创建你自己的特性，这样你可以在将来使用它来创建同样的行为。

原则 25: 让你的类型支持序列化

Prefer Serializable Types

对象的持久是类型的一个核心功能。这是一个在你忽略对它的支持以前，没有人会注意到的基本元素之一。 如果你的类型不能恰当的支持序列化，那么对于把你类的做为基类或者成员的开发人员来说，你会给他们增加很多的工作量。当你的类型不支持序列化时，他们不得不围绕这工作，自己添加实现这个标准的功能。而对于不能访问类的私有成员的开发人来说，恰当的实现你的类型的序列化是不太可能的。如果你的类型不支持序列化，那么对于你的用户来说，想再要实现实它是很困难或者根本就不可能的事。

取而代之的是，为你的实际类型添加序列化。对于那些不用承载 UI 元素，窗口，或者表单的类型来说这是有实际意义的。感觉有额外的工作是没有理由的，.Net 的序列化是很简单的，以至于你没有任意的借口说不支持它。在多数情况下，添加 Serializable 特性就足够了：

```
[Serializable]
public class MyType
{
  private string _label;
  private int _value;
}
```

只添加一个 Serializable 特性就足以让它可以序列化，是因为这它的成员都是可序列化的：string 和 int 都是.Net 序列化支持的。无论是否可能，都要给类型添加序列化支持是很重要的，原因在你添加另一个类做为新类的成员时就很明显了：

```
[Serializable]
public class MyType
{
  private string      _label;
  private int         _value;
  private OtherClass  _object;
}
```

这里的 Serializable 特性只有在 OtherClass 也支持序列化时才有效。如果 OtherClass 不支持序列化，那么你在序列化 MyType 时，因为 OtherClass 对象也在里面，你会得到一个运行时错误。这只是因为对 OtherClass 的内部结构不清楚，而使序列化成为不可能。

.Net 的序列化是把类中所有成员变量保存到输出流中。另外，.Net 的序列化还支持任意的对象图(object graph)：即使你的对象上有一个循环引用，serialize 和 deserialize 方法都只会为你的实际对象读取和储存一次。当一些 web 对象反序列化了以后，.Net 序列化框架也可以创建这些 web 对象的引用。你创建的任何与 web 相关的对象，在对象图序列化以后，你都可以正确的保存它们。最后一个要注意的地方是 Serializable 特性同时支持二进制和 SOAP 序列化。这一原则里的所有技术都支持这两种序列化格式。但是要记住：只有当所有类型的对象图都支持序列化时才能成功。这就是为什么要让所有的类型都支持序列化显得很重要了。一但你放过了一个类，你就轻意的给对象图开了个后门，以至于所有使用这个类的人，想要序列化对象图时变得更加困难。不久以后，他们就会发现不得不自己写序列化代码了。

添加 Serializable 特性是一个最简单的技术来支持对象的序列化。但最简单的方案并不是总是正确的方案。有时候，你并不想序列化对象的所有成员：有些成员可能只存在于长期操作的缓存中，还有一些对象可能占用着一些

运行时资源，而这些资源只能存在于内存中。你同样可以很好的使用特性来控制这些问题。添加 NonSerialized 特性到任何你不想让它序列化的数据成员上。这给它们标上了不用序列化的标记：

```
Serializable]
public class MyType
{
  private string _label;
  [NonSerialized]
  private int _cachedValue;
  private OtherClass  _object;
}
```

你，做为类的设计者，非序列化成员给你多添加了一点点工作。在序列化过程中，序列化 API 不会为你初始化非序列化成员。因为类型的构造函数没有被调用，所以成员的初始化也不会被执行。当你使用序列化特性时，非序列成员就保存着系统默认值：0 或者 null。当默认的 0 对初始化来说是不正确的，那么你须要实现 IDeserializationCallback 接口，来初始化这些非序列化成员。框架会在整个对象图反序列化以后，调用这个方法。这时，你就可以用它为所有的非序列化成员进行初始化了。因为整个对象图已经载入，所以你的类型上的所有方法的调用及成员的使用都是安全的。不幸的是，这不是傻瓜式的。在整个对象图载入后，框架会在对象图中每个实现了 IDeserializationCallback 接口的对象上调用 OnDeserialization 方法。对象图中的其它任何对象可以在 OnDeserialization 正在进行时调用对象的公共成员。如果它们抢在了前面，那么你的非序列化成员就是 null 或者 0。顺序是无法保证的，所以你必须确保你的所有公共成员，都能处理非序列化成员还没有初始化的这种情况。

到目前为止，你已经知道为什么要为所有类型添加序列化了：非序列化类型会在要序列化的对象中使用时带来更多的麻烦事。你也学会了用特性来实现最简单的序列化方法，还包括如何初始化非序列化成员。

序列化了对象有方法在程序的不同版本间生存。(译注：这是一个很重要的问题，因为.Net 里的序列化不像 C++ 那样，你可以轻松的自己控制每一个字节的数据，因此版本问题成了序列化中经常遇到的一个问题。) 添加序列化到一个类型上，就意味着有一天你要读取这个对象的早期版本。Serializable 特性生成的代码，在对象图的成员被添加或者移除时会抛出异常。当你发现你自己已经要面对多版本问题时，你就需要在序列化过程中负出更多的操作：使用 ISerializable 接口。这个接口定义了一些 hook 用于自定义序列化你的类型。ISerializable 接口里使用的方法和存储与默认的序列化方法和储存是一致的，这就是说，你可以使用序列化特性。如果什么时候有必要提供你自己的扩展序列化时，你可以再添加对 ISerializable 接口的支持。

做一个为例子：考虑你如何来支持 MyType 的第 2 个版本，也就是添加了另一个字段到类中时。简单的添加一个字段都会产生一个新的类型，而这与先前已经存在磁盘上的版本是不兼容的：

```
[Serializable]
public class MyType
{
  private string _label;
  [NonSerialized]
  private int _value;
  private OtherClass  _object;
  // Added in version 2
  // The runtime throws Exceptions
  // with it finds this field missing in version 1.0
  // files.
  private int  _value2;
}
```

你实现 ISerializable 接口来支持对这个行为的处理。ISerializable 接口定义了一个方法，但你必需实现两个。ISerializable 定义了 GetObjectData()方法，这是用于写数据到流中。另外，如果你必须提供一个序列析构函数

从流中初始化对象：

```
private MyType(SerializationInfo info,
  StreamingContext cntxt);
```

下面的序列化构造函数演示了如何从先前的版本中读取数据，以及和默认添加的 Serializable 特性生成的序列化保持供一致，来读取当前版本中的数据：

```
using System.Runtime.Serialization;
using System.Security.Permissions;
[Serializable]
public sealed class MyType : ISerializable
{
  private string _label;
  [NonSerialized]
  private int _value;
  private OtherClass  _object;
  private const int DEFAULT_VALUE = 5;
  private int  _value2;
  // public constructors elided.
  // Private constructor used only by the Serialization
      framework.
  private MyType(SerializationInfo info,
    StreamingContext cntxt)
  {
    _label = info.GetString("_label");
    _object = (OtherClass)info.GetValue("_object", typeof
      (OtherClass));
    try {
      _value2 = info.GetInt32("_value2");
    } catch (SerializationException e)
    {
      // Found version 1.
      _value2 = DEFAULT_VALUE;
    }
  }
  [SecurityPermissionAttribute(SecurityAction.Demand,
    SerializationFormatter =true)]
  void ISerializable.GetObjectData (SerializationInfo inf,
    StreamingContext cxt)
  {
    inf.AddValue("_label", _label);
    inf.AddValue("_object", _object);
    inf.AddValue("_value2", _value2);
  }
}
```

序列化流是以键/值对应的方法来保存每一个元素的。默认的特性生成的代码是以变量名做为键来存储值。当你添加了 ISerializable 接口后，你必须匹配键名以及变量顺序。这个顺序就是在类中定义时的顺序。(顺便说一句，

这实际上就是说重新排列类中的变量名或者重新给变量命名，都会破坏对已经创建了的文件的兼容性。)

同样，我已经要求过 SerializationFormatter 的安全许可。如果不实行恰当的保护，对于你的类来说，GetObjectData()可能存在安全漏洞。恶意代码可能会产生一个 StreamingContext，从而可以用 GetObjectData()方法从对象中取得值，或者不断修改版本而取得另一个 SerializationInfo，或者重新组织修改的对象。这就许可了恶意的开发者来访问对象的内部状态，在流中修改它们，然而发送一个修改后的版本给你。对 SerializationFormatter 进行许可要求可以封闭这个安全漏洞。这样可以确保只有受信任的代码才能恰当的访问类的内部状态(参见原则 47)。

但在使用 ISerializable 接口时有一个弊端，你可以看到，我很早就让 MyType 成为密封(sealed)的，这就强制让它只能成为叶子类(leaf class)。在基类实现 ISerializable 接口就隐式的让所有派生类也序列化。实现 ISerializable 就意味关所有派生类必须创建受保护构造函数以及反序列化。另外，为了支持非密封类，你必须在 GetObjectData()方法创建 hook，从而让派生类可以添加它们自己的数据到流中。编译器不会捕获任何这样的错误，当从流中读取派生类时，因缺少恰当的构造构造函数会在运行时抛出异常。缺少 hook 的 GetObjectData()方法也意味着从派生类来的数据不会保存到文件中。当然也不会有错误抛出。所以我要推荐：在叶类中实现 Serializable。

我没有说这，因为它不工作：为了派生类的序列化，你的基类必须支持序列化。修改 MyType ，让它成为了一个可序列化的基类，你要把序列化构造函数修改为 protected，然后创建一个虚方法，这样派生类就可以重载它并存储它们的数据。

```
using System.Runtime.Serialization;
using System.Security.Permissions;
[Serializable]
public class MyType：ISerializable
{
  private string _label;
  [NonSerialized]
  private int _value;
  private OtherClass  _object;
  private const int DEFAULT_VALUE = 5;
  private int  _value2;
  // public constructors elided.
  // Protected constructor used only by the Serialization
      framework.
  protected MyType(SerializationInfo info,
    StreamingContext cntxt)
  {
    _label = info.GetString("_label");
    _object = (OtherClass)info.GetValue("_object", typeof
      (OtherClass));
    try {
      _value2 = info.GetInt32("_value2");
    } catch (SerializationException e)
    {
      // Found version 1.
      _value2 = DEFAULT_VALUE;
    }
  }
```

```
  [ SecurityPermissionAttribute(SecurityAction.Demand,
    SerializationFormatter =true) ]
  void ISerializable.GetObjectData(
    SerializationInfo inf,
    StreamingContext cxt)
  {
    inf.AddValue("_label", _label);
    inf.AddValue("_object", _object);
    inf.AddValue("_value2", _value2);
    WriteObjectData(inf, cxt);
  }
  // Overridden in derived classes to write
  // derived class data:
  protected virtual void
    WriteObjectData(
    SerializationInfo inf,
    StreamingContext cxt)
  {
  }
}
```

一个派生类应该提供它自己的序列化构造函数，并且重载 WriteObjectData 方法：

```
public class DerivedType : MyType
{
  private int _DerivedVal;
  private DerivedType (SerializationInfo info,
    StreamingContext cntxt) :
      base(info, cntxt)
  {
      _DerivedVal = info.GetInt32("_DerivedVal");
  }
  protected override void WriteObjectData(
    SerializationInfo inf,
    StreamingContext cxt)
  {
    inf.AddValue("_DerivedVal", _DerivedVal);
  }
}
```

从流中写入和读取值的顺序必须保持一致。我相信先读写基类的数据应该简单一些，所以我就这样做了。如果你写的代码不对整个继承关系进行精确的顺序序列化，那么你的序列化代码是无效的。

.Net 框架提供了一个简单的方法，也是标准的算法来支持对象的序列化。如果你的类型须要持久，你应该遵守这个标准的实现。如果你的类型不支持序列化，那化其它使用这个类的类也不能序列。为了让使用类的客户更加方便，尽可能的使用默认序列化特性，并且在默认的特性不满足时要实现 ISerializable 接口。

原则 26：用 IComparable 和 IComparer 实现对象的顺序关系

Implement Ordering Relations with IComparable and IComparer

你的类型应该有一个顺序关系，以便在集合中描述它们如何存储以及排序。.Net 框架为你提供了两个接口来描述对象的顺序关系：IComparable 和 IComparer。IComparable 为你的类定义了自然顺序，而实现 IComparer 接口的类可以描述其它可选的顺序。你可以在实现接口时，定义并实现你自己关系操作符(<，>，<=，>=)，用于避免在运行时默认比较关系的低效问题。这一原则将讨论如何实现顺序关系，以便.Net 框架的核心可以通过你定义的接口对你的类型进行排序。这样用户可以在些操作上得更好的效率。

IComparable 接口只有一个方法：CompareTo()，这个方法沿用了传统的 C 函数库里的 strcmp 函数的实现原则：如果当前对象比目标对象小，它的返回值小于 0；如果相等就返回 0；如果当前对象比目标对象大，返回值就大于 0。IComparable 以 System.Object 做为参数，因此在使用这个函数时，你须要对运行时的对象进行检测。每次进行比较时，你必须重新解释参数的类型：

```
public struct Customer : IComparable
{
  private readonly string _name;
  public Customer(string name)
  {
    _name = name;
  }
  #region IComparable Members
  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);
  }
  #endregion
}
```

关于实现比较与 IComparable 接口的一致性有很多不太喜欢的地方，首先就是你要检测参数的运行时类型。不正确的代码可以用任何类型做为参数来调用 CompareTo 方法。还有，正确的参数还必须进行装箱与拆箱后才能提供实际的比较。每次比较都要进行这样额外的开销。在对集合进行排序时，在对象上进行的平均比较次数为 N x log(N)，而每次都会产生三次装箱与拆箱。对于一个有 1000 个点的数组来说，这将会产生大概 20000 次的装箱与拆箱操作，平均计算：N x log(n) 有 7000 次，每次比较有 3 次装箱与拆箱。因此，你必须自己找个可选的比较方法。你无法改变 IComparable.CompareTo()的定义，但这并不意味着你要被迫让你的用户在一个弱类型的实现上也要忍受性能的损失。你可以重载 CompareTo()方法，让它只对 Customer 对象操作：

```
public struct Customer : IComparable
{
  private string _name;
  public Customer(string name)
  {
    _name = name;
  }
  #region IComparable Members
```

```
  // IComparable.CompareTo()
  // This is not type safe. The runtime type
  // of the right parameter must be checked.
  int IComparable.CompareTo(object right)
  {
    if (! (right is Customer))
      throw new ArgumentException("Argument not a customer",
        "right");
    Customer rightCustomer = (Customer)right;
    return CompareTo(rightCustomer);
  }
  // type-safe CompareTo.
  // Right is a customer, or derived from Customer.
  public int CompareTo(Customer right)
  {
    return _name.CompareTo(right._name);
  }
  #endregion
}
```

现在，IComparable.CompareTo()就是一个隐式的接口实现，它只能通过 IComparable 接口的引用才能调用。你的用户则只能使用一个类型安全的调用，而且不安全的比较是不可能访问的。下面这样无意的错误就不能通过编译了：

```
Customer c1;
Employee e1;
if (c1.CompareTo(e1) > 0)
  Console.WriteLine("Customer one is greater");
```

这不能通过编译，因为对于公共的 Customer.CompareTo(Customer right)方法在参数上不匹配，而 IComparable. CompareTo(object right)方法又不可访问，因此，你只能通过强制转化为 IComparable 接口后才能访问：

```
Customer c1;
Employee e1;
if ((c1 as IComparable).CompareTo(e1) > 0)
  Console.WriteLine("Customer one is greater");
```

当你通过隐式实现 IComparable 接口而又提供了一个类型安全的比较时，重载版本的强类型比较增加了性能，而且减少了其他人误用 CompareTo 方法的可能。你还不能看到.Net 框架里 Sort 函数的所有好处，这是因为它还是用接口指针(参见原则 19)来访问 CompareTo()方法，但在已道两个对象的类型时，代码的性能会好一些。

我们再对 Customer 结构做一个小的修改，C#语言可以重载标准的关系运算符，这些应该利用类型安全的 CompareTo()方法：

```
public struct Customer : IComparable
{
  private string _name;
  public Customer(string name)
  {
    _name = name;
```

```
    }
    #region IComparable Members
    // IComparable.CompareTo()
    // This is not type safe. The runtime type
    // of the right parameter must be checked.
    int IComparable.CompareTo(object right)
    {
      if (! (right is Customer))
        throw new ArgumentException("Argument not a customer",
          "right");
      Customer rightCustomer = (Customer)right;
      return CompareTo(rightCustomer);
    }
    // type-safe CompareTo.
    // Right is a customer, or derived from Customer.
    public int CompareTo(Customer right)
    {
      return _name.CompareTo(right._name);
    }
    // Relational Operators.
    public static bool operator < (Customer left,
      Customer right)
    {
      return left.CompareTo(right) < 0;
    }
    public static bool operator <=(Customer left,
      Customer right)
    {
      return left.CompareTo(right) <= 0;
    }
    public static bool operator >(Customer left,
      Customer right)
    {
      return left.CompareTo(right) > 0;
    }
    public static bool operator >=(Customer left,
      Customer right)
    {
      return left.CompareTo(right) >= 0;
    }
    #endregion
  }
```

所有客户的顺序关系就是这样：以名字排序。不久，你很可能要创建一个报表，要以客户的收入进行排序。你还是需要 Custom 结构里定义的普通的比较机制：以名字排序。你可以通过添加一个实现了 IComparer 接口的类来完成这个新增的需求。IComparer 给类型比较提供另一个标准的选择，在.Net FCL 中任何在 IComparable 接

口上工作的函数，都提供一个重载，以便通过接口对对象进行排序。因为你是 Customer 结构的作者，你可以创建一个新的类(RevenueComparer)做为 Customer 结构的一个私有的嵌套类。它通过 Customer 结构的静态属性暴露给用户：

```
public struct Customer : IComparable
{
  private string _name;
  private double _revenue;
  // code from earlier example elided.
  private static RevenueComparer _revComp = null;
  // return an object that implements IComparer
  // use lazy evaluation to create just one.
  public static IComparer RevenueCompare
  {
    get
    {
      if (_revComp == null)
        _revComp = new RevenueComparer();
      return _revComp;
    }
  }
  // Class to compare customers by revenue.
  // This is always used via the interface pointer,
  // so only provide the interface override.
  private class RevenueComparer : IComparer
  {
    #region IComparer Members
    int IComparer.Compare(object left, object right)
    {
      if (! (left is Customer))
        throw new ArgumentException(
          "Argument is not a Customer",
          "left");
      if (! (right is Customer))
        throw new ArgumentException(
          "Argument is not a Customer",
          "right");
      Customer leftCustomer = (Customer) left;
      Customer rightCustomer = (Customer) right;
      return leftCustomer._revenue.CompareTo(
        rightCustomer._revenue);
    }
    #endregion
  }
}
```

最后这个版本的 Customer 结构，包含了 RevenueComparer 类，这样你就可以以自然顺序-名字，对对象

进行排序；还可有一个选择就是用这个暴露出来的，实现了 IComparer 接口的类，以收入对客户进行排序。如果你没有办法访问 Customer 类的源代码，你还可以提供一个 IComparer 接口，用于对它的任何公共属性进行排序。只有在你无法取得源代码时才使用这样的习惯，同时也是在.Net 框架里的一个类须要不同的排序依据时才这样用。

这一原则里没有涉及 Equals()方法和==操作符(参见原则 9)。排序和相等是很清楚的操作，你不用实现一个相等比较来表达排序关系。 实际上，引用类型通常是基于对象的内容进行排序的，而相等则是基于对象的 ID 的。在 Equals()返回 false 时，CompareTo()可以返回 0。这完全是合法的，相等与排序完全没必要一样。

(译注：注意作者这里讨论的对象，是排序与相等这两种操作，而不是具体的对象，对于一些特殊的对象，相等与排序可能相关。)

IComparable 和 IComparer 接口为类型的排序提供了标准的机制，IComparable 应该在大多数自然排序下使用。当你实现 IComparable 接口时，你应该为类型排序重载一致的比较操作符(<, >, <=, >=)。IComparable.CompareTo()使用的是 System.Object 做为参数，同样你也要重载一个类型安全的 CompareTo()方法。IComparer 可以为排序提供一个可选的排序依据，这可以用于一些没有给你提供排序依据的类型上，提供你自己的排序依据。

原则 27：避免使用 ICloneable

Avoid ICloneable

ICloneable 看上去是个不错的主意：为一个类型实现 ICloneable 接口后就可以支持拷贝了。如果你不想支持拷贝，就不要实现它。

但你的对象并不是在一个"真空"的环境中运行，但考虑到对派生类的些影响，最好还是对 ICloneable 支持。一但某个类型支持 ICloneable, 那么所有的派生类都必须保持一致，也就是所有的成员必须支持 ICloneable 接口或者提供一种机制支持拷贝。最后，支持深拷贝的对象，在创建设计时如果包含有网络结构的对象，会使拷贝很成问题。ICloneable 也觉察到这个问题，在它的官方定义中有说明：它同时支持深拷贝和浅拷贝。浅拷贝是创建一个新的对象，这个新对象对包含当前对象中所有成员变量的拷贝。如果这些成员变量是引用类型的，那么新的对象与源对象包含了同样的引用。而深拷贝则可以很好的拷贝所有成员变量，引用类型也被递归的进行了拷贝。对于像整型这样的内置类型，深拷贝和浅拷贝是一样的结果。哪一种是我们的类型应该支持的呢？这取决于类型本身。但同时在一个类型中混用深拷贝和浅拷贝会导致很多不一致的问题。一但你涉及到 ICloneable 这个问题，这样的混用就很难解脱了。大多数时候，我们应该完全避免使用 ICloneable，让类更简单一些。这样使用和实现都相对简单得多。

任何只以内置类型做为成员的值类型不必支持 ICloneable; 用简单的赋值语句对结构的所有值进行拷贝比 Clone()要高效得多。Clone()方法必须对返回类型进行装箱，这样才能强制转化成一个 System.Object 的引用。而调用者还得再用强制转化从箱子中取回这个值。我知道你已经有足够的能力这样做，但不要用 Clone()函数来取代赋值语句。

那么，当一个值类型中包含一个引用类型时又会怎样呢？最常见的一种情况就是值类型中包含一个字符串：

```
public struct ErrorMessage
{
  private int errCode;
  private int details;
  private string msg;
 // details elided
}
```

字符串是一个特殊情况，因为它是一个恒定类。如果你指定了一个错误消息串，那么所有的错误消息类都引用到同一个字符串上。而这并不会导致任何问题，这与其它一般的引用类型是不一样的。如果你在任何一个引用上修改了 msg 变量，你会就为它重新创建了一个 string 对象(参见原则 7)。

(译注：string 确实是一个很有意思的类，很多 C++程序员对这个类不理解，也很有一些 C#程序对它不理解，导致很多的低效，甚至错误问题。应该好好的理解一下 C#里的 string(以及 String 和 StringBulider 之间的关系)

这个类，这对于学好 C#是很有帮助的。因为这种设计思想可以沿用到我们自己的类型中。)

一般情况，如果一个结构中包含了一个任意的引用类型，那么拷贝时的情况就复杂多了。这也是很少见的，内置的赋值语句会对结构进行浅拷贝，这样两个结构中的引用变量就引用到同个一个对象上。如果要进行深拷贝，那么你就必须对引用类型也进行拷贝，而且还要知道该引用类型上是否也支持用 Clone()进行深拷贝。不管是哪种情况，你都不用对值类型添加对 ICloneable 的支持，赋值语句会对值类型创建一个新的拷贝。

一句概括值类型：没有任何理由要给一个值类型添加对 ICloneable 接口的支持! 好了，现在让我们再看看引用类型。引用类型应该支持 ICloneable 接口，以便明确的给出它是支持深拷贝还是浅拷贝。明智的选择是添加对 ICloneable 的支持，因为这样就明确的要求所有派生类也必须支持 ICloneable。看下面这个简单的继承关系：

```
class BaseType : ICloneable
{
  private string _label = "class name";
  private int [] _values = new int [ 10 ];
  public object Clone()
  {
    BaseType rVal = new BaseType();
    rVal._label = _label;
    for(int i = 0; i < _values.Length; i++)
      rVal._values[ i ] = _values[ i ];
    return rVal;
  }
}
class Derived : BaseType
{
  private double [] _dValues = new double[ 10 ];
  static void Main(string[] args)
  {
    Derived d = new Derived();
    Derived d2 = d.Clone() as Derived;
    if (d2 == null)
      Console.WriteLine("null");
  }
}
```

如果你运行这个程序，你就会发现 d2 为 null。虽然 Derived 是从 BaseType 派生的，但从 BaseType 类继承的 Clone()函数并不能正确的支持 Derived 类：它只拷贝了基类。BaseType.Clone()创建的是一个 BaseType 对象，不是派生的 Derived 对象。这就是为什么程序中的 d2 为 null 而不是派生的 Derived 对象。即使你克服了这个问题，BaseType.Clone()也不能正确的拷贝在 Derived 类中定义的_dValues 数组。一但你实现了 ICloneable, 你就强制要求所有派生类也必须正确的实现它。实际上，你应该提供一个 hook 函数，让所有的派生类使用你的拷贝实现(参见原则 21)。在拷贝时，派生类可以只对值类型成员或者实现了 ICloneable 接口的引用类型成员进行拷贝。对于派生类来说这是一个严格的要求。在基类上实现 ICloneable 接口通常会给派生类添加这样的负担，因此在密封类中应该避免实现 ICloneable 接口。

因此，当整个继承结构都必须实现 ICloneable 时，你可以创建一个抽象的 Clone()方法，然后强制所有的派生类都实现它。

在这种情况下，你需要定义一个方法让派生类来创建基类成员的拷贝。可以通过定义一个受保护的构造函数来实现：

```
class BaseType
```

```
{
  private string _label;
  private int [] _values;
  protected BaseType()
  {
    _label = "class name";
    _values = new int [ 10 ];
  }
  // Used by devived values to clone
  protected BaseType(BaseType right)
  {
    _label = right._label;
    _values = right._values.Clone() as int[ ] ;
  }
}
sealed class Derived : BaseType, ICloneable
{
  private double [] _dValues = new double[ 10 ];
  public Derived ()
  {
    _dValues = new double [ 10 ];
  }
  // Construct a copy
  // using the base class copy ctor
  private Derived (Derived right) :
    base (right)
  {
    _dValues = right._dValues.Clone()
      as double[ ];
  }
  static void Main(string[] args)
  {
    Derived d = new Derived();
    Derived d2 = d.Clone() as Derived;
    if (d2 == null)
      Console.WriteLine("null");
  }
  public object Clone()
  {
    Derived rVal = new Derived(this);
    return rVal;
  }
}
```

基类并不实现 ICloneable 接口；通过提供一个受保护的构造函数，让派生类可以拷贝基类的成员。叶子类，应该都是密封的，必要它应该实现 ICloneable 接口。基类不应该强迫所有的派生类都要实现 ICloneable 接口，但

你应该提供一些必要的方法，以便那些希望实现 ICloneable 接口的派生类可以使用。

ICloneable 接口有它的用武之地，但相对于它的规则来说，我们应该避免它。对于值类型，你不应该实现 ICloneable 接口，应该使用赋值语句。对于引用类型来说，只有在拷贝确实有必要存在时，才在叶子类上实现对 ICloneable 的支持。基类在可能要对 ICloneable 进行支持时，应该创建一个受保护的构造函数。总而言之，我们应该尽量避免使用 ICloneable 接口。

原则 28：避免转换操作

Avoid Conversion Operators

转换操作是一种等代类型(Substitutability)间操作转换操作。等代类型就是指一个类可以取代另一个类。这可能是件好事：一个派生类的对象可以被它基类的一个对象取代，一个经典的例子就是形状继承。先有一个形状类，然后派生出很多其它的类型：长方形，椭圆形，圆形以及其它。你可以在任何地方用图形状来取代圆形，这就是多态的等代类型。这是正确的，因为圆形就是一个特殊的形状。当你创建一个类时，明确的类型转化是可以自动完成的。正如.Net 中类的继承，因为 System.Object 是所有类型的基类，所以任何类型都可以用 System.Obejct 来取代。同样的情况，你所创建的任何类型，也应该可以用它所实现的接口来取代，或者用它的基类接口来取代，或者就用基类来取代。不仅如此，C#语言还支持很多其它的转换。

当你为某个类型添加转换操作时，就等于是告诉编译器：你的类型可以被目标类所取代。这可能会引发一些潜在的错误，因为你的类型很可能并不能被目标类型所取代(译注：这里并不是指继承关系上的类型转换，而是 C#语言许可我们的另一种转换，请看后文)。它所的副作用就是修改了目标类型的状态后可能对原类型根本无效。更糟糕的是，如果你的转换产生了临时对象，那么副作用就是你直接修改了临时对象，而且它会永久丢失在垃圾回收器。总之，使用转换操作应该基于编译时的类型对象，而不是运行时的类型对象。用户可能须要对类型进行多样化的强制转换操作，这样的实际操作可能产生不维护的代码。

你可以使用转换操作把一个未知类型转化为你的类型，这会更加清楚的表现创建新对象的操作(译注：这样的转换是要创建新对象的)。转换操作会在代码中产生难于发现的问题。假设有这样一种情况，你创建了如图 3.1 那样的类库结构。椭圆和圆都是从形状类继承下来的，尽管你相信椭圆和圆是相关的，但还是决定保留这样的继承关系。这是因为你不想在继承关系中使用非抽象叶子类，这会在从椭圆类上继承圆类时，有一些不好实现的难题存在。然而，你又意识到每一个圆形应该是一个椭圆，另外某些椭圆也可能是圆形。

(图 3.1)

(译注：这一原则中作者所给出的例子不是很恰当，而且作者也在前面假设了原因，因此请读者不要对这个例子太钻牛角尖，理解作者所在表达的思想就行了，相信在你的 C#开发中可能也会遇到类似的转换问题，只是不太可能从圆形转椭圆。)

这将导致你要添加两个转换操作。因为每一个圆形都是一个椭圆，所以要添加隐式转换从一个圆形转换到新的椭圆。隐式转换会在一个类要求转化为另一个类时被调用。对应的，显示转化就是程序员在代码中使用了强制转换操作符。

```
public class Circle : Shape
{
  private PointF _center;
  private float _radius;
  public Circle() :
    this (PointF.Empty, 0)
  {
  }
  public Circle(PointF c, float r)
  {
    _center = c;
```

```
    _radius = r;
  }
  public override void Draw()
  {
    //...
  }
  static public implicit operator Ellipse(Circle c)
  {
    return new Ellipse(c._center, c._center,
      c._radius, c._radius);
  }
}
```

现在你就已经实现了隐式的转换操作，你可以在任何要求椭圆的地方使用圆形。而且这个转换是自动完成的：

```
public double ComputeArea(Ellipse e)
{
  // return the area of the ellipse.
}
// call it:
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
ComputeArea(c);
```

我只是想用这个例子表达可替代类型：一个圆形已经可以代替一个可椭圆了。ComputeArea 函数可以在替代类型上工作。你很幸运，但看下面这个例子：

```
public void Flatten(Ellipse e)
{
  e.R1 /= 2;
  e.R2 *= 2;
}
// call it using a circle:
Circle c = new Circle(new PointF (3.0f, 0), 5.0f);
Flatten(c);
```

这是无效的，Flatten()方法要求一个椭圆做为参数，编译器必须以某种方式把圆形转化为椭圆。确实，也已经实现了一个隐式的转换。而且你转换也被调用了，Flatten()方法得到的参数是从你的转换操作中创建的新的椭圆对象。这个临时对象被 Flatten()函数修改，而且它很快成为垃圾对象。正是因为这个临时对象，Flatten()函数产生了副作用。最后的结果就是这个圆形对象，c，根本就没有发生任何改变。从隐式转换修改成显示转换也只是强迫用户调用强制转换而以：

```
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
Flatten((Ellipse) c);
```

原先的问题还是存在。你想让用户调用强制转换为解决这个问题，但实际上还是产生了临时对象，把临时对象进行变平(flatten)操作后就丢掉了。原来的圆，c，还是根本没有被修改过。取而代之的是，如果你创建一个构造函数把圆形转换成椭圆，那么操作就很明确了：

```
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
Flatten (new Ellipse(c));
```

相信很多程序员一眼就看的出来，在前面的两行代码中传给 Flatten()的椭圆在修改后就丢失了。他们可能会通过跟踪对象来解决这个问题：

```
Circle c = new Circle(new PointF(3.0f, 0), 5.0f);
```

```
// Work with the circle.
// ...
// Convert to an ellipse.
Ellipse e = new Ellipse(c);
Flatten(e);
```

通过一个变量来保存修改(变平)后的椭圆，通过构造函数来替换转换操作，你不会丢失任何功能：你只是让创建新对象的操作更加清楚。(有经验的 C++程序可能注意到 C#的隐式转化和显示转换都没有调用构造函数。在 C++中，只有明确的使用 new 操作符才能创建一个新的对象时，其它时候不行。而在 C#的构造函数中不用明确的使用关键字。)

从类型里返回字段的转换操作并不会展示类型的行为，这会产生一些问题。你给类型的封装原则留下了几个严重的漏洞。通过把类型强制转化为其它类型，用户可以访问到类型的内部变量。这正是原则 23 中所讨论的所有原因中最应该避免的。

转换操作提供了一种类型可替代的形式，但这会给代码引发一些问题。你应该已经明白所有这些内容：用户希望可以合理的用某种类型来替代你的类型。当这个可替代类型被访问时，你就让用户在临时对象上工作，或者内部字段取代了你创建的类。随后你可能修改了临时对象，然后丢掉。因为这些转换代码是编译器产生的，因此这些潜在的 BUG 很难发现。应该尽量避免转换操作。

原则 29：仅在对基类进行强制更新时才使用 new 修饰符

Use the new Modifier Only When Base Class Updates Mandate It

你可以用 new 修饰符来重新定义一个从基类中继承来的非虚成员。你可以这样做，但并不意味着需要这样做。重新定义非虚方法会导致方法含意的混乱。如果两个相关的类是继承关系，那么很多开发人员可能会立即假设两段代码块是做完全相同的事情，而且他们也会这么认为：

```
object c = MakeObject();
// Call through MyClass reference:
MyClass cl = c as MyClass;
cl.MagicMethod();
// Call through MyOtherClass reference:
MyOtherClass cl2 = c as MyOtherClass;
cl2.MagicMethod();
```

一但使用了 new 修饰符以后，问题就完全不一样了：

```
public class MyClass
{
  public void MagicMethod()
  {
    // details elided.
  }
}
public class MyOtherClass : MyClass
{
  // Redefine MagicMethod for this class.
  public new void MagicMethod()
  {
    // details elided
  }
```

```
}
```

这样的实际操作会让很多开发人员迷惑。因为当你在同一个对象上调用相同的函数时，一定希望它们执行同样的代码。但实际上是，一但你用不同的引用来调用同名的函数，它们的行为是不一样的，这感觉非常糟糕。它们是不一致的。一个 MyOtherClass 类型的对象所表现的行为会因为你引用的方式不一样而有所不同。这就是 new 修饰符用在非虚成员上的后果。其实这只是让你在类的名字空间中添加了一个不同的方法(虽然它们的函数名是相同的)。

非虚方法是静态绑定的，不管哪里的代码，也不管在哪里引用，MyClass.MagicMethod() 总是严格的调用类中所定义的函数。并不会在运行时在派生类中查找不同的版本。另一方面，虚函数动态的。运行时会根据不同的类型对象调用不同的版本。

建议大家避免使用 new 修饰符来重新定义非虚函数，这并不要太多的解释，就像推荐大家在定义一个基类时应该用虚方法一样。一个类库的设计者应该按照合某种约设计虚函数。也就表示你期望任何派生类都应该修改虚函数的实现。虚函数的集合就相当于是定义了一个行为的集合，这些行为是希望在派生中重新实现的。设计默认的虚函数就是说派生可以修改类中的所有虚的行为。这确实是说你不想考虑所有派生类可能要修改行为的分歧问题。相反，你可以把时间花在考虑把什么样的方法以及属性设计成多态的。当然，只有它们是虚行为的时候才能这样做。不要考虑这样会限制类的用户。相反，应该认为这是给类型的用户定义行为提供了一个入口向导。

有且只有一种情况要使用 new 修饰符，那就是把类集成到一个已经存在的基类上时，而这个基类中已经使用了存在的方法名，这时就要使用 new 了(译注：就是说基类与派生类都已经存在了，是后来添加的继承关系，结果在添加继承关系时，发现两个类中使用了同样的方法名，那么就可以在派生类中添加一个 new 来解决这个问题)。因为有些代码已经依懒于类的方法名，或者已经有其它程序集在使用这个方法。例如你在库中创建了下面的类，使用了在另一个库中定义的 BaseWidget：

```
public class MyWidget：BaseWidget
{
  public void DoWidgetThings()
  {
    // details elided.
  }
}
```

你完成了你的 widget, 而且用户可以使用它。然而你却发现 BaseWidget 公司发布了一个新的版本。而这正是你所渴望的，于是你立即购买并编译你的 MyWidget 类。结果失败了，因为 BaseWidget 的家伙们已经添加了他们自己的 DoWidgetThings 方法：

```
public class BaseWidget
{
  public void DoWidgetThings()
  {
    // details elided.
  }
}
```

这是个难题，你的基类中隐藏了一个方法，而这又是在你的类的名字空间中。有两个方法解决这个问题，一个就是修改你的类中的方法名：

```
public class MyWidget：BaseWidget
{
  public void DoMyWidgetThings()
  {
    // details elided.
  }
```

```
}
```

或者使用 new 修饰符：

```
public class MyWidget：BaseWidget
{
  public new void DoWidgetThings()
  {
    // details elided.
  }
}
```

如果你可以拿到所有使用 MyWidget 类的源代码，那么你应该选择修改方法名，因为这对于今后的运行会更简单。然而，如果你已经向全世界的人发布了 MyWidget 类，这会迫使所有用户来完成这个众多的改变。这正是 new 修饰符容易解决的问题，你的用户不用修改 DoWidgetThings()方法而继续使用它。没有人会调用到 BaseWidget.DoWidgetThings()方法，因为(对于派生类而言)它们根本不存在。在更新一个基类时，如果发现它与先前申明的成员发生了冲突，可以用 new 修饰符来解决这个问题。

当然，在某些时候，你的用户可能想调用基类的 Widget.DoWidgetThings()方法，这时你又回到了原来的问题上：两个方法看上去是一样的，但其实是不同的。考虑到 new 修饰长期存在的歧意问题，有时候，还是在短期上麻烦一下，修改方法名为上策。(译注：长痛不如短痛。呵呵)

new 修饰符必须小心谨慎的使用。如果它是有歧意的，你就在类上创建了个模糊的方法。这只有在特殊情况下才使用，那就是升级基类时与你的类产生冲突时。即使在这种情况下，也应该小心的使用它。最重要的是，其它任何时候都不要用它。

第四章 创建基于二进制的组件

Creating Binary Components

随着类的数量增加，创建二进制的组件就变得很平常了：你想分离部分功能。所有不同的是，二进制组件可以让你独立的保存这些离散的功能。创建的组件程序集可以方便的共享逻辑，方便交叉语言 编程，以及方便布置。

在.Net 程序就是组件包，每一个程序什么样可以随时更新和发布。你应该已经知道，基于程序集之间的应用程序是多么容易更新呀，程序集之间的偶合是多么好呀！最小偶合就是说更多的是减少了程序集之间复杂的依懒关系。同时也意味着你可以简单的更新小块新 的程序集。这一章就是关于创建简单易用，容易布置，以及容易更新的程序集。

.Net 环境中的应用程序可以由多样的二进制组件组成。随后，你可以独立的更新这些组件，也就是可以在一个多程序集应用程序中安装与更新一个程序集。但你必须明白一件事，那就是 CLR 是如何发现和载入这些程序集的。在你创建这些二进制组件时，你也就必须创建符合这些明确期望的组件。接下来就介绍这一概念。

CLR 并不会在程序启动时加载全部的引用程序集。更合适的作法是当运行时须要程序的成员时，加载器才解决程序集的引用问题。这可能是调用一个方法或者访问数据，加载器先查找引用的程序集，然后加载它，然后 JIT 编译须要的 IL。

当 CLR 须要加载一个程序集时，先是断定那个文件要加载。程序集的元数据包含了对所有其它程序集的引用记录，这个记录有强名称和弱名称之分。一个强名称(程序集)由四部分组成：程序名，版本号，语言属性(译注：这里的语言范指区域性特定的信息，如关联的语言、子语言、国家/地区、日历和区域性约定)，以及公开密钥。如果程序集不是一个强名称程序，那么就只记录了程序集名。如果你的程序是一个强名称程序集，那么你的程序集就不太可能被一些恶意的组件(程序集)所取代。强程序集还可以让你用配置文件对组件的版本进行配置，是新的，还是先前的。

确定正确的程序集以后，CLR 接来就断定这个程序集是否已经在当前的应用程序中加载。如果是，就用原来的。如果不是，CLR 就继承查找程序集。如果程序是强名称的，CLR 就先在全局程序缓存(GAC)中查找，如果不在 GAC

中，加载器就检测代码目录(codebase directory，译注，这里只是译为代码目录，其实并不是源代码所在的目录)中的配置文件，如果当前代码目录存在，那就是唯一被搜索程的目录了。如果在代码目录中没有找到所要的程序集，那么加载就失败。

如果没有直接的代码目录，加载器会搜索预先设定的目录集：

* 应用程序目录。也就是与主应用程序集同在的位置。

* 语言目录。这是一个在应用程序目录下的子目录，这个目录与当前语言名匹配。

* 程序集子目录。这是一个与程序集同名的子目录，这两个可以组合成这样： [语言]/[程序集名]

* 私有的运行目录(binPath)。这是一个在应用程序配置文件中定义的私有目录。它同样可以和语言和程序集名组合： [bin 目录]/[程序集名], 或者[bin 目录]/[语言], 还可以是：[bin 目录]/[语言]/[程序集名]。

从这个讨论中你应该明白三个内容：第一，只有强名称程序集才能放到 GAC 中。其次，你可以通过配置文件来修改默认的行为，从而升级程序中个别的强名称程序集。第三，强名称程序集可以阻止对程序集的恶意篡改，从而提供更安全的应用程序。

了解 CLR 是如何加载程序集的，有利于在实际操作中考虑如何创建和更新组件。首先，你应该使用强名称程序集，把所有的元数据信息都记录下来。当你在 VS.net 中创建一个项目时，你应该把 assemblyInfo.cs 文件中的所有属性都填满，包括完整的版本号。这会让你在后面的升级中更简单的。VS.net 在 assemblyInfo.cs 中创建三个不同的部份，首先就是信息报告：

```
[assembly: AssemblyTitle("My Assembly")]
[assembly: AssemblyDescription
  ("This is the sample assembly")]
#if DEBUG
[assembly: AssemblyConfiguration("Debug")]
#else
[assembly: AssemblyConfiguration("Release")]
#endif
[assembly: AssemblyCompany("My company")]
[assembly: AssemblyProduct("It is part of a product")]
[assembly: AssemblyCopyright("Insert legal text here.")]
[assembly: AssemblyTrademark("More legal text")]
[assembly: AssemblyCulture("en-US")]
```

最后一条，AssemblyCulture 只针对本地化程序集。如果你的程序不包含任何的本地化资源，就空着。否则就应该遵从 RFC1766 标准填写语言描述信息。

接下来的部份就是版本号，VS.net 是这样记录的：

```
[assembly: AssemblyVersion("1.0.*")]
```

AssemblyVersion 包含 4 个部份：主版本号.副版本号.编译号.修订号，星号就是告诉编译器用当前时间来填写编译号和修订号。编译号就是从 2000 年 1 月 1 号起的天数，而修订号则是从当天凌晨起的秒数除以 2。这样的算法可以确保修订号是不断递增的：每次编译后的号码都会比前一次高。(译注：我有一点不明白的，就是如果我把本机时间修改了呢？或者两台机器上的时间不一致，会有什么问题呢？当然，这并不会有什么太大的问题。)

对于这一版本号的算法，好消息是两个编译的版本不会有完全相同的版本号。坏消息就是在你编译并发布后，要记住这个版本号。我个人比较喜欢让编译器给我生成编译和修订号。通过记录发布时的编译号，我就知道最后的版本号了。我从来不会忘记在我发布一个新程序集时修要改它的版本号。但也有例外，COM 组件是在你每次编译后自动注册的。如果还让编译器自己生成编译号后然后注册它，那么很快就让注册表里填满了无用的信息。

最后一部份就是强名称信息：

```
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

应该考虑为所有的程序集都创建强名称，强名称程序集可以防止别人的篡改，而且可以为单个应用程序独立更新个别程序集。然而，你应该在 ASP.net 应用程序中避免使用强程序集；局部安装的强名称程序集不会正确的加载。同样，强名称程序集必须用 AllowPartiallyTrustedCallers 特性进行修饰，否则它们不能被非强名称程序集访问(参见原则 47)。

实际在更新一个组件时，公共的和受保护的接口部份必须在 IL 语言层上是兼容的。也就是说不能删除方法，不能修改参数，不能修改返回值。简单的说，没有一个组件愿意在引用你的组件后要重新编译。

你可以添加配置信息来修改引用的程序集。配置信息可以存储在不同的位置，这取决于你想如何更新组件。如果是一个简单的应用程序，你可以创建一个应用程序配置文件，然后把它存放在应用程序目录就行了。如果是配置所以使用同一个组件的应用程序，你可以在 GAC 中创建一个发布策略文件。最后，如果是全局的修改，你可以修改 machine.config 文件，这个文件在.Net 进行时的配置目录里(参见原则 37)。

实际操作中，你可能从来不用修改 machine.config 文件来更新你的程序集。这个文件包含了基于整台机器的信息。你可以通过应用程序配置文件来更新单个应用程序配置，或者使用一个发布策略来更新多个程序公用的组件。

这是一个 XML 配置文件，它描述了存在的版本信息和升级后的信息：

```
<dependentAssembly>
  <assemblyIdentity name="MyAssembly"
    publicKeyToken="a0231341ddcfe32b" culture="neutral" />
  <bindingRedirect oldVersion="1.0.1444.20531"
    newVersion="1.1.1455.20221" />
</dependentAssembly>
```

你可以通过这个配置文件来标识程序集，旧版本，以及升级后的版本。当你安装和更新了程序集后，你就要更新或者创建一个恰当的配置文件，然后程序程序就可以使用新的版本了。

如果你的软件是一个程序集的集合：你希望个别的更新它们。通过一次更新一个程序集，你须要做一些预先的工作，那就是第一次安装时应该包含一些必要的支持升级的信息。

原则 30：选择与 CLS 兼容的程序集

Prefer CLS-Compliant Assemblies

.Net 运行环境是语言无关的：开发者可以用不同的.Net 语言编写组件。而且在实际开发中往往就是这样的。你创建的程序集必须是与公共语言系统(CLS)是兼容的，这样才能保证其它的开发人员可以用其它的语言来使用你的组件。

CLS 的兼容至少在公共命名上要与互用性靠近。CLS 规范是一个所有语言都必须支持的最小操作子集。创建一个 CLS 兼容的程序集，就是说你创建的程序集的公共接口必须受 CLS 规范的限制。这样其它任何满足 CLS 规范的语言都可以使用这个组件。然而，这并不是说你的整个程序都要与 CLS 的 C#语言子集相兼容。

为了创建 CLS 兼容的程序集，你必须遵从两个规则：首先，所以参数以及从公共的和受保护的成员上反回的值都必须是与 CLS 兼容的。其次，其它不与 CLS 兼容的公共或者受保护成员必须存在 CLS 兼容的同意对象。

第一个规则很容易实现：你可以让编译来强制完成。添加一个 CLSCompliant 特性到程序集上就行了：

```
[ assembly: CLSCompliant(true) ]
```

编译器会强制整个程序集都是 CLS 兼容的。如果你编写了一个公共方法或者属性，它使用了一个与 CLS 不兼容的结构，那么编译器会认为这是错误的。这非常不错，因为它让 CLS 兼容成了一个简单的任务。在打开与 CLS 兼容性后，下面两个定义将不能通过编译，因为无符号整型不与 CLS 兼容：

```
// Not CLS Compliant, returns unsigned int:
public UInt32 Foo()
{
  return _foo;
```

```
}
// Not CLS compliant, parameter is an unsigned int.
public void Foo2(UInt32 parm)
{
}
```

记住，创建与 CLS 兼容的程序集时，只对那些可以在当前程序集外面可以访问的内容有效。Foo 和 Foo2 在定义为公共或者受保护时，会因与 CSL 不兼容而产生错误。然而如果 Foo 和 Foo2 是内部的，或者是私有的，那么它们就不会被包含在要与 CLS 兼容的程序集中；CLS 兼容接口只有在把内容向外部暴露时才是必须的。

那么属性又会怎样呢？它们与 CLS 是兼容的吗？

```
public MyClass TheProperty
{
  get { return _myClassVar; }
  set { _myClassVar = value; }
}
```

这要视情况而定，如果 MyClass 是 CLS 兼容的，而且表明了它是与 CLS 兼容的，那么这个属性也是与 CLS 兼容的。相反，如果 MyClass 没有标记为与 CLS 兼容，那么属性也是与 CLS 不兼容的。就意味着前面的 TheProperty 属性只有在 MyClass 是在与 CLS 兼容的程序集中是，它才是与 CLS 兼容的。

如果你的公共的或者受保护的接口与 CLS 是不兼容的，那么你就不能编译成 CLS 兼容的程序集。作为一个组件的设计者，如果你没有给程序集标记为 CLS 兼容的，那么对于你的用户来说，就很难创建与 CLS 兼容的程序集了。他们必须隐藏你的类型，然后在 CLS 兼容中进行封装处理。确实，这样可以完成任务，但对于那些使用组件的程序员来说不是一个好方法。最好还是你来努力完成所有的工作，让程序与 CLS 兼容：对于用户为说，这是可以让他们的程序与 CLS 兼容的最简单的方法。

第二个规则是取决与你自己的：你必须确保所有公共的及受保护的操作是语言无关的。同时你还要保证你所使用的多态接口中没有隐藏不兼容的对象。

操作符重载这个功能，有人喜欢有人不喜欢。同样，也并不是所有的语言都支持操作符重载的。CLS 标准对于重载操作符这一概念即没有正面的支持也没有反正的否定。取而代之是，它为每个操作符定义了一了函数：op_equals 就是=操作符所对应的函数名。op_addis 是重载了加号后的函数名。当你重载了操作符以后，操作符语法就可以在支持操作符重载的语言中使用。如果某些开发人员使用的语言不支持操作符重载时，他们就必须使用 op_这样的函数名了。如果你希望那些程序员使用你的 CLS 兼容程序集，你应该创建更多的方便的语法。介此，推荐一个简单的方法：任何时候，只要重载操作运算符时，再提供一个等效的函数：

```
// Overloaded Addition operator, preferred C# syntax:
public static Foo operator+(Foo left, Foo right)
{
  // Use the same implementation as the Add method:
  return Foo.Add(left, right);
}
// Static function, desirable for some languages:
public static Foo Add(Foo left, Foo right)
{
  return new Foo (left.Bar + right.Bar);
}
```

最后，注意在使用多态的接口时，那些非 CLS 的类型可能隐藏在一些接口中。最容易出现的就是在事件的参数中。这会让你创建一些 CLS 不兼容的类型，而在使用的地方却是用与 CLS 兼容的基类。

假设你创建了一个从 EventArgs 派生的类：

```
internal class BadEventArgs : EventArgs
```

```
{
  internal UInt32 ErrorCode;
}
```

这个 BadEventArgs 类型就是与 CLS 不兼容的，你不可能在其它语言中写的事件句柄上使用这个参数。但多态性却让这很容易发生。你只是申明了事件参数为基类：EventArgs:

```
// Hiding the non-compliant event argument:
public delegate void MyEventHandler(
  object sender, EventArgs args);
public event MyEventHandler OnStuffHappens;
// Code to raise Event:
BadEventArgs arg = new BadEventArgs();
arg.ErrorCode = 24;
// Interface is legal, runtime type is not:
OnStuffHappens(this, arg);
```

以 EventArgs 为参数的接口申明是与 CLS 兼容的，然而，实际取代参数的类型是与 CLS 不兼容的。结果就是一些语言不能使用。

最后以如何实现 CLS 兼容类或者不兼容接口来结束对 CLS 兼容性的讨论。兼容性是可以实现的，但我们可以更简单的实现它。明白 CLS 与接口的兼容同样可以帮助你完整的理解 CLS 兼容的意思，而且可以知道运行环境是怎样看待兼容的。

这个接口如果是定义在 CLS 兼容程序集中，那么它是 CLS 兼容的：

```
[ assembly:CLSCompliant(true) ]
public interface IFoo
{
  void DoStuff(Int32 arg1, string arg2);
}
```

你可以在任何与 CLS 兼容的类中实现它。然而，如果你在与没有标记与 CLS 兼容的程序集中定义了这个接口，那么这个 IFoo 接口就并不是 CLS 兼容的接口。也就是说，一个接口只是满足 CLS 规范是不够的，还必须定义在一个 CSL 兼容的程序集中时才是 CLS 兼容的。原因是编译器造成的，编译器只在程序集标记为 CLS 兼容时才检测 CLS 兼容类型。相似的，编译器总是假设在 CLS 不兼容的程序集中定义的类型实际上都是 CLS 不兼容的。然而，这个接口的成员具有 CLS 兼容性标记。即使 IFoo 没有标记为 CLS 兼容，你也可以在 CLS 兼容类中实现这个 IFoo 接口。这个类的客户可以通过类的引来访问 DoStuff，而不是 IFoo 接口的引用。

考虑这个简单的参数：

```
public interface IFoo2
{
  // Non-CLS compliant, Unsigned int
  void DoStuff(UInt32 arg1, string arg2);
}
```

一个公开实现了 IFoo2 接口的类，与 CLS 是不兼容的。为了让一个类即实现 IFoo2 接口，同时也是 CLS 兼容的，你必须使用清楚的接口定义：

```
public class MyClass: IFoo2
{
  // explicit interface implementation.
  // DoStuff() is not part of MyClass's public interface
  void IFoo2.DoStuff(UInt32 arg1, string arg2)
```

```
  {
    // content elided.
  }
}
```

MyClass 有一个与 CLS 兼容的接口，希望访问 IFoo2 接口的客户必须通过访问与 CLS 不兼容的 IFoo2 接口指针。

兼容了吗？不，还没。创建一个 CLS 兼容类型要求所有的公共以及受保护接口都只包含 CLS 兼容类型。这就是说，某个类的基类也必须是 CLS 兼容的。所有实现的接口也必须是 CLS 兼容的。如果你实现了一个 CLS 不兼容的接口，你必须实现明确的接口定义，从而在公共接口上隐藏它。

CLS 兼容性并没有强迫你去使用最小的公共名称来实现你的设计。它只是告诉你应该小心使用程序集上的公共的接口。以于任何公共的或者受保护的类，在构造函数中涉及的任何类型必须是 CLS 兼容的，这包含：

*基类

*从公共或者受保护的方法和属性上返回的值

*公共及受保护的方法和索引器的参数

*运行时事件参数

*公共接口的申明和实现

编译器会试图强制兼容一个程序集。这会让提供最小级别上的 CLS 兼容变得很简单。再稍加小心，你就可以创建一个其它语言都可以使用的程序集了。编译器的规范试图确保不用牺牲你所喜欢的语言的结构就可以尽可能的与其它语言兼容。你只用在接口中提供可选的方案就行了。

CLS 兼容性要求你花点时间站在其它语言上来考虑一下公共接口。你不必限制所有的代码都与 CLS 兼容，只用避免接口中的不兼容结构就行了。通用语言的可操作性值得你花点时间。

原则 31：选择小而简单的函数

Prefer Small, Simple Functions

做为一个有经验的程序员，不管你在使用 C#以前是习惯用什么语言的，我们综合了几个可以让你开发出有效代码的实际方法。有些时候，我们在先前的环境中所做的努力在.Net 环境中却成了相反的。特别是在你试图手动去优化一些代码时尤其突出。你的这些行为往往会阻止 JIT 编译器进行最有效的优化。你的以性能为由的额外工作，实际上产生了更慢的代码。你最好还是以你最清楚的方法写代码，其它的让 JIT 编译器来做。最常见的一个例子就是预先优化，你创建一个很长很复杂的函数，本想用它来避免太多的函数调用，结果会导致很多问题。实际操作时，提升这样一个函数的逻辑到循环体中对.Net 程序是有害的。这与你的真实是相反的，让我们来看一些细节。

这一节介绍一个简单的内容，那就是 JIT 编译器是如何工作的 。.Net 运行时调用 JIT 编译器，用来把由 C#编译器生成的 IL 指令编译成机器代码。这一任务在应用程序的运行期间是分步进行的。JIT 并不是在程序一开始就编译整个应用程序，取而代之的是，CLR 是一个函数接一个函数的调用 JIT 编译器。这可以让启动开销最小化到合理的级别，然而不合理的是应用程序保留了大量的代码要在后期进行编译。那些从来不被调用的函数 JIT 是不会编译它的。你可以通过让 JIT 把代码分解成更多的小块，从而来最小化大量无关的代码，也就是说小而多的函数比大而少的函数要好。考虑这个人为的例子：

```
public string BuildMsg(bool takeFirstPath)
{
  StringBuilder msg = new StringBuilder();
  if (takeFirstPath)
  {
    msg.Append("A problem occurred.");
    msg.Append("\nThis is a problem.");
```

```
    msg.Append("imagine much more text");
  } else
  {
    msg.Append("This path is not so bad.");
    msg.Append("\nIt is only a minor inconvenience.");
    msg.Append("Add more detailed diagnostics here.");
  }
  return msg.ToString();
}
```

在 BuildMsg 第一次调用时，两个选择项就都编译了。而实际上只有一个是须要的。但是假设你这样写代码：

```
public string BuildMsg(bool takeFirstPath)
{
  if (takeFirstPath)
  {
    return FirstPath();
  } else
  {
    return SecondPath();
  }
}
```

因为函数体的每个分支被分解到了独立的小函数中，而 JIT 就是须要这些小函数，这比前面的 BuildMsg 调用要好。确实，这个例子只是人为的，而且实际上它也没什么太特别的。但想想，你是不是经常写更"昂贵"的例子呢：一个 if 语句中是不是每个片段中都包含了 20 或者更多的语句呢？你的开销就是让 JIT 在第一次调用它的时候两个分支都要编译。如果一个分支不像是错误条件，那到你就招致了本可以简单避免的浪费。小函数就意味着 JIT 编译器只编译它要的逻辑，而不是那些沉长的而且又不会立即使用的代码。对于很长的 switch 分支，JIT 要花销成倍的存储，因此把每个分支的内容定义成内联的要比分离成单个函数要好。

JIT 编译器可以更简单的对小而简单的函数进行可登记(enregistration)处理。可登记处理是指进程选择哪些局部变量可以被存储到寄存器中，而这比存储到堆栈中要好。创建少的局部变量可以能 JIT 提供更好的机会把最合适的候选对象放到寄存器中。这个简单的控制流程同样会影响 JIT 编译能否如期的进行变量注册。如果函数只有一个循环，那么循环变量就很可能被注册。然而，当你在一个函数中使用过多的循环时，对于变量注册，JIT 编译器就不得不做出一些困难的决择。简单就是好，小而简单的函数很可能只包含简单几个变量，这样可以让 JIT 很容易优化寄存器的使用。

JIT 编译器同样决定内联方法。内联就是说直接使用函数体而不必调用函数。考虑这个例子：

```
// readonly name property:
private string _name;
public string Name
{
  get
  {
    return _name;
  }
}
// access:
```

```
string val = Obj.Name;
```

相对函数的调用开销来说，属性访问器实体包含更少数的指令：对于函数调用，要先在寄存器中存储它的状态，然后从头到尾执行，接着存储返回结果。这还不谈如果有参数时，把参数压到堆栈上还要更多的工作。如果你这样写，这会产生更多的机器指令：

```
string val = Obj._name;
```

当然，你应该不会这样做，因为你已经明白最好不要创建公共数据成员(参见原则 1)。JIT 编译器明白你即须要效率也须要简洁，所以它会内联属性访问器。JIT 会在以速度或者大小为目标(或者两个同时要求)时，内联一些方法，用函数体来取代函数的调用会让它更有利。一般情况不用为内联定义额外的规则，而且任何已经实现的内联在将来都可能被改变。另外，内联函数并不是你的职责。正好 C#语言没有提供任何关键字让你暗示编译器说你想内联某个函数。实际上，C#编译器也不支持任何暗示来让 JIT 编译进行内联。你可以做的就是确保你的代码尽可能的清楚，尽可能让 JIT 编译器容易的做出最好的决定。我的推荐现在就很熟悉了：越小的方法越有可能成为内联对象。请记住：任何虚方法或者含有 try/catch 块的函数都不可能成为内联的。

内联修改了代码正要被 JIT 的原则。再来考虑这个访问名字属性的例子：

```
string val = "Default Name";
if (Obj != null)
  val = Obj.Name;
```

JIT 编译器内联了属性访问器，这必然会在相关的方法被调用时 JIT 代码。

你没有责任来为你的算法决定最好的机器级别上的表现。C#编译器以及 JIT 编译器一起为你完成了这些。C#编译器为每个方法生成 IL 代码，而 JIT 编译器则把这些 IL 代码在目标机器上翻译成机器指令。并不用太在意 JIT 编译器在各种情况下的确切原则；有这些时间可以开发出更好的算法。取而代之的，你应该考虑如何以一种好的方式表达你的算法，这样的方式可以让开发环境的工具以最好的方式工作。幸运的是，这些你所考虑的这些原则(译注：JIT 工作原则)已经成为优秀的软件开发实践。再强调一次：使用小而简单的函数。

记住，你的 C#代码经过了两步才编译成机器可执行的指令。C#编译器生成以程序集形式存在的 IL 代码。而 JIT 编译器则是在须要时，以每个函数为单元生成机器指令(当内联调用时，或者是一组方法)。小函数可以让它非常容易被 JIT 编译器分期处理。小函数更有可能成为内联候选对象。当然并不是足够小才行：简单的控制流程也是很重要的。函数内简单的控制分支可以让 JIT 以容易的寄存变量。这并不是只是写清晰代码的事情，也是告诉你如何创建在运行时更有效的代码。

原则 32：选择小而内聚的程序集

Prefer Smaller, Cohesive Assemblies

这一原则实际应该取这个名字："应该创建大小合理而且包含少量公共类型的程序集"。但这太沉长了，所以就以我认为最常见的错误来命名：开发人员总是把所有的东西，除了厨房里水沟以外(译注：夸张说法，kitchen sink 可能是个口语词，没能查到是什么意思，所以就直译了。)，都放到一个程序集。这不利于重用其中的组件，也不利于系统中小部份的更新。很多以二进制组件形式存在的小程序集可以让这些都变得简单。

然而这个标题对于程序集的内聚来说也很醒目的。程序集的内聚性是指概念单元到单个组件的职责程度。聚合组件可以简单的用一句话概括，你可以从很多.Net 的 FCL 程序集中看到这些。有两个简单的例子：System.Collections 程序集就是负责为相关对象的有序集提供数据结构，而 System.Windows.Forms 程序集则提供 Windows 控件类的模型。Web form 和 Windows Form 在不同的程序集中，因为它们不相关。你应该用同样的方式，用简单的一句话来描述你的程序集。不要玩花样：一个 MyApplication 程序集提供了你想要的一切内容。是的，这也是简单的一句，但这也太刁懒了吧，而且你很可能在 My2ndApplication(我想你很可能会要重用到其中的一些内容。这里"其中的一些内容"应该放到一个独立的程序集中。)程序集并不须要使用所有的功能。

你不应该只用一个公共类来创建一个程序程序集。应该有一个折衷的方法，如果你太偏激，创建了太多的程序集，你就失去了使用封装的一些好处：首先就是你失去了使用内部类型的机会，内部类型是在一个程序集中与封装(打包)无关的公共类(参见原则 33)(译注：简单的说，内部类型就是只能在一个公共的程序集中访问类，程序集以外限

制访问)。JIT 编译器可以在一个程序集内有很的内联效率，这比起在多程序集中穿梭效率要高得多。这就是说，在一个程序集中放置一些相关的类型对你是有好处的。我们的目标就是为我们的组件创建大小最合适的程序集。这一目标很容易实现，就是一个组件应该只有一个职责。

在某些情况下，一个程序集就是类的二进制表现形式，我们用类来封装算法和存储数据。只有公共的接口才能成为"官方"的合约，也就是只有公共接口才能被用户访问。同样，程序集为相关类提供二进制的包，在这个程序集以外，只有公共和受保护的类是可见的。工具类可以是程序集的内部类。确实，它们对于私有的嵌套类来说它们应该具有更更宽的访问范围，但你有一个机制可以共享程序集内部通用的实现，而不用暴露这个实现给所有的用户。那就是封装相关类，然后从程序集中分离成多个程序。

其实，使用多程序集可以让很多不同布署选项变得很简单。考虑一个三层应用程序，一部份程序以智能客户端的形式在运行，而另一部份则是在服务器上运行。你在客户端上提供了一些验证原则，用于确保用户反馈的数据输入和修改是正确的。而在服务器上你又要重复这些原则，而且复合一些验证以保证验证更严格。而这些在服务器端的业务原则应该是一个完整的集合，而在每个客户端上只是一个子集。

确实，你也可以通过重用源文件来为客户端和服务器的业务原则创建不同的程序集，但这对你的布署机制来说会成为一个复杂的问题。当你更新这些业务原则时，你就有两个安装要完成。相反，你可以从严格的服务器端验证中分离一部分验证，封装成不同的程序集放置到客户端。这样，你就重用封装成程序集的二进制对象。这比起重用代码或者资源，重新编译成多个程序集要好得多。

做为一个程序，应该是一个包含相关功能的组织结构库。这已经是大家熟悉的了，但在实际操作中却很难实现。实际上，对于一个分布式应用程序，你可能不能提前知道哪些类应该同时分布到服务器和客户端上。即使可能，服务端和客户端的功能也有可能是流动的；你将来很有可能要面临两边都要处理的地步。通过尽可能能的让程序集小，你就有可能更简单的重新布署服务器和客户端。程序集是应用程序的二进制块，对于一个工作的应用程序来说，很容易添加一个新的组件插件。如果你不小心出了什么错误，创建过多的程序集要比个别很太的程序要容易处理得多。

我经常程序集和二进制组件类似的看作是 Lego。你可以很容易的抽出一个 Lego 然后用另一个代替。同样的，对于有相同接口的程序集来说，你应该可以很容易的把它抽出来然后用一个新的来替换。而且程序其它部份应该可以继续像往常一样运行。这和 Lego 有点像，如果你的所有参数和返回值都是接口，那么任何一个程序集就可以很容易的用另一个有相同接口的来代替(参见原则 19)。

更小的程序集同样可以让你对程序启动时的开销进行分期处理。更大的程序要花上更多的 CUP 时间来加载，以及更多的时间来编译必须的 IL 到机器指令。应该只在启动时 JIT 一些必须的内容，而程序集是整个载入的，而且 CLR 要为程序集中的每个方法保存一个存根。

稍微休息一下，而且确保我们不会走到极端。这一原则是确保你不会创建出单个单片电路的程序，而是创建基于二进制的整体系统，而且是可重用的组件。不要参考这一原则而走到另一个极端。一个基于太多小程序集的大型应用程序的开销是相关的。如果你的程序使用了太多的程序集，那么在程序集之间的穿梭会产生更多的开销。在加载更多的程序集并转化 IL 为机器指令时，CLR 的加载器有一点额外的工作要完成，那就是调整函数入口地址。

同样，以程序集之间穿梭时，安全性检查也会成为一个额外的开销。同一个程序集中的所有的代码具有相同的信任级别(并不是同样的访问级别，而是可信级别)。 无论何时，只要代码访问超出了一个程序集，CLR 都要完成一些安全验证。程序花在程序集间穿梭的时间越少，相对程序的效率就更高。

这些与性能相关的说明并没有一个是劝阻你把一个大程序集分离成小程序集的。性能的损失是其次的，C# 和.Net 的设计是以组件为核心思想的，更好的伸缩性通常更有价值。

那么，你决定一个程序集中放多少代码或者多少类呢？更重要的是，你是如何决定哪些代码应该在一个程序集中？这很大程度上取决于实际的应用程序，因此这并没有一个确论。我这里有一个推荐：通过观察所有的公共类开始，用一个公共基类合并这些类到一个程序集中。然后添加一些工具类到这个程序集中，这些工具类主要是负责提供所有相关类的功能。把相关的公共接口封装到一个独立的程序集中。最后一步，查看那些在应用程序中横向访问的对象，这些是有可能成为广泛使用的工具程序集的候选对象，它们可能会包含在应用程序的工具库中。

最后的结果就是，你的组件只在一个简单的相关集合中，这个集合中只有一些必须的公共类，以及一些工具类来支持它们。这样，你就创建了一个足够小的程序集，而且很容易从更新和重用中得到好处，同时也在最小化多个程序集相关的开销。一个设计好的内聚组件可以用一句话来概括。例如，"Common.Storage.dll 用管理所有离线

用户数据缓存以及用户设置。”就描述了一低内聚的组件。相反，做两个组件：“Common.Data.dll 管理离线数据缓存。Common.Settings.dll 管理用户设置。” 当你把它们分开后，你可能还要使用一个第三方组件：“Common.EncryptedStorage.dll 为本地加密存储管理文件系统 IO”，这样你就可以独立的更新这三个组件了。

小，是一个相对的条件。Mscorlib.dll 就大概有 2MB，System.Web. RegularExpressions.dll 却只有 56KB。但它们都满足小的核心设计目标，重用程序集：它们都包含相关类和接口的集合。绝对大小的不同应该根据功能的不同来决定：mscorlib.dll 包含了所有应用程序中要使用的最底层的类。而 System.Web.RegularExpressions.dll 却很特殊，它只包含一些在 Web 控件中要使用的正则表达式类。这就创建了两种不同类型的组件：一个就是小，而大的程序集则是集中在特殊的功能上，广泛应用的程序集包含通用的功能。不论哪种情况，应该它们尽可能合理的小，直到不能再小。

原则 33：限制类型的访问

Limit Visibility of Your Types

并不是所有的人都须要知道所有的事。也不是所有的类型须要是公共的。对于每个类型，在满足功能的情况下，应该尽可能的限制访问级别。而且这些访问级别往往比你想像的要少得多。在一个私有类型上，所有的用户都可以通过一个公共的接口来访问这个接口所定义的功能。

让我们回到最根本的情况上来：强大的工具和懒惰的开发人员。VS.net 对于他们来说是一个伟大的高产工具。我用 VS.net 或者 C# Builder 轻松的开发我所有的项目，因为它让我更快的完成任务。其中一个加强的高产工具就是让你只用点两下按钮，一个类就创建了，当然如果这正是我想要的话。VS.net 为我们创建的类就是这样的：

```
public class Class2
{
  public Class2()
  {
    //
    // TODO: Add constructor logic here
    //
  }
}
```

这是一个公共类，它在每个使用我的程序集的代码块上都是可见的。这样的可见级别太高了，很多独立存在的类都应该是内部(internal)的。你可以通过在已经存在的类里嵌套一个受保护的或者私有的类来限制访问。 越低的访问级别，对于今后的更新整个系统的可能性就越少。越少的地方可以访问到类型，在更新时就越少的地方要修改。

只暴露须要暴露的内容，应该通过尝试在类上实现公共接口来减少可见内容。你应该可以在.Net 框架库里发现使用 Enumerator 模式的例子，System.ArrayList 包含一个私有类，ArrayListEnumerator, 而就是它只实现了 IEnumerator 接口：

```
// Example, not complete source
public class ArrayList: IEnumerable
{
  private class ArraylistEnumerator : IEnumerator
  {
    // Contains specific implementation of
    // MoveNext(), Reset(), and Current.
  }
  public IEnumerator GetEnumerator()
  {
    return new ArrayListEnumerator(this);
```

```
  }
// other ArrayList members.
}
```

对于我们这样的使用者来说，不须要知道 ArrayListEnumerator 类，所有你须要知道的，就是当我们在 ArrayList 对象上调用 GetEnumerator 函数时，你所得到的是一个实现了 IEnumerator 接口的对象。而具体的实现则是一个明确的类。.Net 框架的设计者在另一个集合类中使用了同样的模式：哈希表(Hashtable)包含一个私有的 HashtableEnumerator, 队列(Queue)包含一个 QueueEnumerator, 等等。私有的枚举类有更多的优势。首先，ArrayList 类可以完全取代实现 IEnumerator 的类型，而且你已经成为一个贤明的程序员了，不破坏任何内容。其实，枚举器类不须要是 CLS 兼容的，因为它并不是公共的(参见原则 30)。而它的公共接口是兼容的。你可以使用枚举器而不用知道实现的类的任何细节问题。

创建内部的类是经常使用的用于限制类型可见范围的概括方法。默认情况下，很多程序员都总是创建公共的类，从来不考虑其它方法。这是 VS.net 的事。我们应该取代这种不加思考的默认，我们应该仔细考虑你的类型会在哪些地方使用。它是所有用户可见的？或者它主要只是在一个程序集内部使用？

通过使用接口来暴露功能，可以让你更简单的创建内部类，而不用限制它们在程序集外的使用(参见原则 19)。类型应该是公共的呢？或者有更好的接口聚合来描述它的功能？内部类可以让你用不同的版本来替换一个类，只要在它们实现了同样的接口时。做为一个例子，考虑这个电话号码验证的问题：

```
public class PhoneValidator
{
  public bool ValidateNumber(PhoneNumber ph)
  {
    // perform validation.
    // Check for valid area code, exchange.
    return true;
  }
}
```

几个月过后，这个类还是可以很好的工作。当你得到一个国际电话号码的请求时，前面的这个 PhoneValidator 就失败了。它只是针对 US 的电话号码的。你仍然要对 US 电话号码进行验证，而现在，在安装过程中还要对国际电话号码进行验证。与其粘贴额外的功能代码到一个类中，还不如了断减少两个不同内容耦合的做法，直接创建一个接口来验证电话号码：

```
public interface IPhoneValidator
{
  bool ValidateNumber(PhoneNumber ph);
}
```

下一步，修改已经存在的电话验证，通过接口来实现，而且把它做为一个内部类：

```
internal class USPhoneValidator : IPhoneValidator
{
  public bool ValidateNumber(PhoneNumber ph)
  {
    // perform validation.
    // Check for valid area code, exchange.
    return true;
  }
}
```

最后，你可以为国际电话号码的验证创建一个类：

```
internal class InternationalPhoneValidator : IPhoneValidator
```

```
{
  public bool ValidateNumber(PhoneNumber ph)
  {
    // perform validation.
    // Check international code.
    // Check specific phone number rules.
    return true;
  }
}
```

为了完成这个实现，你须要创建一个恰当的类，这个类基于电话号码类型类，你可以使用类厂模式实现这个想法。在程序集外，只有接口是可见的。而实际的类，就是这个为世界不同地区使用的特殊类，只有在程序集内是可见的。你可以为不同的区域的验证创建不同的验证类，而不用再系统里的其它程序集而烦扰了。

你还可以为 PhoneValidator 创建一个公共的抽象类，它包含通用验证的实现算法。用户应该可以通过程序集的基类访问公共的功能。在这个例子中，我更喜欢用公共接口，因为即使是同样的功能，这个相对少一些。其他人可能更喜欢抽象类。不管用哪个方法实现，在程序集中尽可能少的公开类。

这些暴露在外的公共类和接口就是你的合约：你必须保留它们。越多混乱的接口暴露在外，将来你就越是多的直接受到限制。越少的公共类型暴露在外，将来就越是有更多的选择来扩展或者修改任何的实现。

原则 34：创建大容量的 Web API

Create Large-Grain Web APIs

交互协议的开销与麻烦就是对数据媒体的如何使用。在交互过程中可能要不同的使用媒体，例如在交流中要不同的使用电话号码，传真，地址，和电子邮件地址。让我们再回头来看看上次的订购目录，当你用电话订购时，你要回答售货员的一系列问题：

“你可以把第一项填一下吗？”

“这一项的号码是 123-456”

"您想订购多少呢？"

"三件"

这样的问题一直要问到销售人员填写完所有的信息为止，例如还要知道你的订购地址，信用卡信息，运送地址，以及其它一些必须的信息来完成这比交易。在电话上完成这样一来一回的讨论还是令人鼓舞的。因为你不会是一个人长时间的自言自语，而且你也不会长时间忍受销售人员是否还要哪里的安静状态。

与传真订购相比，你要填写整个订购文档，然后把整个文档发给公司。一个文件一次性传输完成，你不用很填写产品编号，发传真，然后填写地址，然后再传真，填写信用卡号，然后再发传真。

这里演示了一个定义糟糕的 web 方法接口会遇到的常见缺陷。当你使用 web 服务，或者.Net 远程交互时，你必须记住：最昂贵的开销是在两台远程机器之间进行对象传输时出现。你不应该只是通过重新封装一下原来在本地计算机上使用的接口来创建远程 API。虽然这样是可以工作的，但效率是很低的。

这就有点类似是用电话的方式来完成用传真订购的任务。你的应用程序大部份时间都在每次向信道上发送一段数据后等待网络。使用越是小块的 API，应用程序在等待服务器数据返回的时间应用比就更高。

相反，我们在创建基于 web 的接口时，应该把服务器与客户端的一系列对象进行序列化，然后基于这个序列化后的文档进行传输。你的远程交流应该像用传真订购时使用的表单一样：客户端应该有一个不与服务器进行通信的扩展运行时间段。这时，当所用的信息已经填写完成时，用户就可以一次性的提交这个文档到服务器上。服务器上还是做同样的事情：当服务器上返回到客户上的信息到达时，客户的手头上就得到了完成订购任务必须的所有信息。

比喻说我们要粘贴一个客户订单，我们要设计一个客户的订购处理系统，而且它要与中心服务器和桌面用户通过网络访问信息保持一致。系统其中的一个类就是客户类。如果你忽略传输问题，那么客户类可能会像这样设计，这充许用户取回或者修改姓名，运输地址，以及账号信息：

```
public class Customer
{
  public Customer()
  {
  }
  // Properties to access and modify customer fields:
  public string Name
  {
    // get and set details elided.
  }
  public Address shippingAddr
  {
    // get and set details elided.
  }
  public Account creditCardInfo
  {
    // get and set details elided.
  }
}
```

这个客户类不包含远程调用的 API，在服务器和客户之间调用一个远程的用户会产生严重的交通阻塞：

```
// create customer on the server.
Customer c = new Server.Customer();
// round trip to set the name.
c.Name = dlg.Name.Text;
// round trip to set the addr.
c.shippingAddr = dlg.Addr;
// round trip to set the cc card.
c.creditCardInfo = dlg.credit;
```

相反，你应该在本机创建一个完整的客户对象，然后等用户填写完所有的信息后，再输送这个客户对象到服务器：

```
// create customer on the client.
Customer c = new Customer();
// Set local copy
c.Name = dlg.Name.Text;
// set the local addr.
c.shippingAddr = dlg.Addr;
// set the local cc card.
c.creditCardInfo = dlg.credit;
// send the finished object to the server. (one trip)
Server.AddCustomer(c);
```

这个客户的例子清楚简单的演示了这个问题：在服务器与客户端之间一来一回的传输整个对象。但为了写出高效的代码，你应该扩展这个简单的例子，应该让它包含正确的相关对象集合。在远程请求中，使用对象的单个属性就是使用太小的粒子(译注：这里的粒子就是指一次交互时所包含的信息量)。但，对于每次在服务器与客户之间传输来说，一个客户实例可能不是大小完全正确的粒子。

让我们来再扩展一下这个例子，让它更接近现实设计中会遇到的一些问题，我们再对系统做一些假设。这个软

件主要支持一个拥有 1 百万客户的在线卖主。假设每个用户有一个订购房子的主要目录，平均一点，去年有 15 个订单。

每个电话接线员使用一台机器轮班操作，而且不管电话订单者是否回答电话，他们都要查找或者创建这条订单记录。你的设计任务是决定大多数在客户和服务器之间传输的高效对象集合。

你一开始可能消除一些显而易见的选择，例如取回每一个客户以及每次的订单信息是应该明确禁止的：1 百万客户以及 15 百万(1 千 5 百万)订单记录显然是太大了而不应该反回到做一个客户那里去。这样很容易在另一个用户上遇到瓶颈问题。在每次可能要更新数据时，都会给服务器施加轰炸式打击，你要发送一个包含 15 百万对象的请求。当然，这只是一次事务，但它确实太低效了。

相反，考虑如何可以最好的取回一个对象的集合，你可以创建一个好的数据集合代理，处理一些在后来几分钟一定会使用的对象。一个接线员回复一个电话，而且可能对某个客户有兴趣。在电话交谈的过程中，接线员可能添加或者移除订单，修改订单，或者修改一个客户的账号信息。明显的选择就是取回一个客户，以及这个用户的所有订单。服务器上的方法可能会是这样的：

```
public OrderData FindOrders(string customerName)
{
  // Search for the customer by name.
  // Find all orders by that customer.
}
```

对的吗？传送到客户而且客户已经接收到的订单很可能在客户机上是不须要的。一个更好的做法就是为每个请求的用户只取回一条订单。服务器的方法可能修改成这个样子：

```
public OrderData FindOpenOrders(string customerName)
{
  // Search for the customer by name.
  // Find all orders by that customer.
  // Filter out those that have already
  // been received.
}
```

这样你还是要让客户机为每个电话订单创建一个新的请求。有一个方法来优化通信吗？比下载用户包含的所有订单更好的方法。我们会在业务处理中添加一些新的假设，从而给你一些方法。假设呼叫中心是分布的，这样每个工作组收到的电话具有不同的区号。现在你就可以修改你的设计了，从而对交互进行一个不小的优化。

每个区域的接线员可能在一开始轮班时，就取回并且更新客户以及订单信息。在每次电话后，客户应用程序应该把修改后的数据返回到服务上，而且服务器应该响应上次客户请求数据以后的所有修改。结果就是，在每次电话后，接线员发送所有的修改，这些修改包含这个组中其它接线员所做的所有修改。这样的设计就是说，每一个电话只有一次会话，而且每一个接线员应该在每次回复电话时，手里有数据集合访问权。这样服务器上可能就有两个这样的方法：

```
public CustomerSet RetrieveCustomerData(
  AreaCode theAreaCode)
{
  // Find all customers for a given area code.
  // Foreach customer in that area code:
    // Find all orders by that customer.
    // Filter out those that have already
    // been received.
  // Return the result.
```

```
}
public CustomerSet UpdateCustomer(CustomerData
  updates, DataTime lastUpdate, AreaCode theAreaCode)
{
  // First, save any updates, marking each update
  // with the current time.
  // Next, get the updates:
  // Find all customers for a given area code.
  // Foreach customer in that area code:
    // Find all orders by that customer that have been
    // updated since the last time. Add those to the result.
  // Return the result.
}
```

但这样可能还是要浪费一些带宽。当每个已知客户每天都有电话时，最后一个设计是最有效。但这很可能是不对的。如果是的，那么你的公司应该在客户服务上存在很大的问题，而这个问题应该用软件是无法解决的。

如何更进一步限制传输大小呢，要求不增加会话次数和及服务器的响应延时？你可以对数据库里的一些准备打电话的客户进行一些假设。你可以跟踪一些统计表，然后可以发现，如果一些客户已经有 6 个月没有订单了，那么他们很可能就不会再有订单了。这时你就应该在那一天的一开始就停止取回这些客户以及他们的订单。这可以收缩传输的初始大小，你同样可以发现，很多客户在通过一个简短电话下了订单过后，经常会再打电话来询问上次订单的事。因此，你可以修改订单列表，只传输最后的一些订单而不是所有的订单。这可能不用修改服务器上的方法签名，但这会收缩传输给客户上的包的大小。

这些假设的讨论焦点是要给你一些关于远程交互的想法：你减少两机器间的会话频率和会话时数据包的大小。这两个目标是矛盾的，你要在这两者中做一个平衡的选择。你应该取两个极端的中点，而不是错误的选择过大，或者过小的会话。

第五章 和 Framework 一起工作

Working with the Framework

我的同事，也是我的朋友Martin Shoemaker研究一个很严肃的问题，那就是:“我必须写这样的.Net代码吗？”答案是，也希望是：不。你应该使用你手头上有的，也是你会用的工具来帮助你写代码。

.Net 框架是一个很丰富的类库，你对框架学习的越多，你自己要写的代码就越少。框架库会帮你完成很多工作。这一章就告诉你一些.Net 框架里最常用的一些技术。当你在.Net 框架中有多个选择时，这一章中的一些原则会帮助你选择最好的。你可以使用一些已经存在的东西来写你自己的类和算法，而不应该是与它们相抵触。

这一章中的一些原则反映一些算法和类，当一些开发者可以轻松的使用.Net 框架的时候，这是他们决心要自己写的。他们之所以要自己写这些，这是因为，有些时候，.Net 框架确实不清楚他们想要什么。这时，我会告诉你如何来扩展已经存在的核心功能。还有一些时候，因为他们不清楚核心时如何工作的；也有时候是因为他们对性能有过高的要求。

即使是使用.框架里所有可用的工具，也有很多的开发人员宁可创建他们自己的工具。千万别写这些代码，特别是已经有人完成了的。

原则 35：选择重写函数而不是使用事件句柄

Prefer Overrides to Event Handlers

很多.Net 类提供了两种不同的方法来控制一些系统的事件。那就是，要么添加一个事件句柄；要么重写基类的

虚函数。为什么要提供两个方法来完成同样的事情呢？其实很简单，那就是因为不同的情况下要调用为的方法。在派生类的内部，你应该总是重写虚函数。而对于你的用户，则应该限制他们只使用句柄来响应一些不相关的对象上的事件。

例如你很了一个很不错的 Windows 应用程序，它要响应鼠标点下的事件。在你的窗体类中，你可以选择重写 OnMouseDown()方法：

```
public class MyForm：Form
{
  // Other code elided.
  protected override void OnMouseDown(
    MouseEventArgs e)
  {
    try {
      HandleMouseDown(e);
    } catch (Exception e1)
    {
      // add specific error handling here.
    }
    // *almost always* call base class to let
    // other event handlers process message.
    // Users of your class expect it.
    base.OnMouseDown(e);
  }
}
```

或者你可以添加一个事件句柄：

```
public class MyForm：Form
{
  // Other code elided.
  public MyForm()
  {
    this.MouseDown += new
      MouseEventHandler(this.MouseDownHandler);
  }
  private void MouseDownHandler(object sender,
    MouseEventArgs e)
  {
    try {
      HandleMouseDown(e);
    } catch (Exception e1)
    {
      // add specific error handling here.
    }
  }
}
```

前面一些方法要好一些，如果在事件链上有一个句柄抛出了一个异常，那么其它的句柄都不会再被调用(参见原

则 21)。一些“病态”的代码会阻止系统调用事件上的句柄。通过重写受保护的虚函数，你的控制句柄会就先执行。基类上的虚函数有责任调用详细事件上的所有添加的句柄。这就是说，如果你希望事件上的句柄被调用(而且这是你最想完成的)，你就必须调用基类。而在一些罕见的类中，你希望取代基类中的默认事件行为，这样可以让事件上的句柄都不被执行。你不去保证所所的事件句柄都将被调用，那是因为一些“病态”事件句柄可能会引发一些异常，但你可以保证你派生类的行为是正确的。

使用重载比添加事件句柄更高效。我已经在原则 22 中告诉过你，System.Windows.Forms.Control 类是如何世故的使用任命机制来存储事件句柄，然后映射恰当的句柄到详细的事件上。这种事件机制要花上更多的处理器时间，那是因为它必须检测事件，看它是否有事件句柄添加在上面。如果有，它就必须迭代整个调用链表。方法链表中的每个方法都必须调用。断定有哪些事件句柄在那里，还要对它们进行运行时迭代，这与只调用一个虚函数来说，要花上更多的执行时间。

如果这还不足以让你决定使用重载，那就再看看这一原则一开始的链表。那一个更清楚？如果重载虚函数，当你在维护这个窗体时，只有一个函数要检查和修改。而事件机制则有两个地方要维护：一个就是事件句柄，另一就是事件句柄上的函数。任何一个都可能出现失败。就一个函数更简单一些。

OK，我已经给出了所有要求使用重载而不是事件句柄的原因。.Net 框架的设计者必须要添加事件给某人，对吗？当然是这样的。就你我们剩下的内容一个，他们太忙了而没时间写一些没人使用的代码。重写只是为派生类提供的，其它类必须使用事件机制。例如，你经常添加一个按钮点击事件到一个窗体上。事件是由按钮触发的，但是由窗体对象处理着事件。你完全可以在这个类中定义一个用户的按钮，而且重写这个点击句柄，但这对于只是处理一个事件来说花上了太多的代码。不管怎样，问题都是交给你自己的类了：你自己定义的按钮还是在点击时必须与窗体进行通信。显然应该用事件来处理。因此，最后，你只不过是创建了一个新类来向窗体发送事件(译注：其实我们完全可以创建这个类不用发事件给窗体就可以完成回调的，只是作者习惯的说什么好就一味的否定其它。但不管怎样，重写一个按钮来重载函数确实不是很值。)。 相对前面一种方法，直接在窗体事件添加句柄要简单得多。这也就是为什么.Net 框架的设计者把事件放在窗体的最前面。

另一个要使用事件的原因就是，事件是在运行时处理的。使用事件有更大的伸缩性。你可以在一个事件上添加多个句柄，这取决于程序的实际环境。假设你写了一个绘图程序，根据程序的状态，鼠标点下时应该画一条线，或者这它是要选择一个对象。当用户切换功能模式时，你可以切换事件句柄。不同的类，有着不同的事件句柄，而处理的事件则取决于应用程序的状态。

最后，对于事件，你可以把多个事件句柄挂到同样的事件上。还是想象同样的绘图程序，你可能在 MouseDown 事件上挂接了多个事件句柄。第一个可能是完成详细的功能，第二个可能是更新状态条或者更新一些可访问的不同命令。不同的行为可以在同一事件上响应。

当你有一个派生类中只有一个函数处理一个事件时，重载是最好的方法。这更容易维护，今后也会更正确，而且更高效。而应该为其它用户保留事件。因此，我们应该选择重写基类的实现而不是添加事件句柄。

原则 36：利用.Net 运行时诊断

Leverage .NET Runtime Diagnostics

当有问题发生时，它们往往并不是在实验的时候发生的，机器有轻松调试的工具。在很多实际情况中，你不好修正的问题总是发生在用户的机器上，那里没有调试环境，也没有好的方法计算出问题的情况。在实际情况中，有经验的开发人员会创建一个方法，让系统在运行时捕获尽可能多的信息。.Net 框架已经包含一些类集合，利用这些集合，你可以做一些通用的调试。而且这些类可以在运行时或者编译时进行配置。如果你利用它们，你就可以轻松的发现在实际运行时的问题。使用框架里已经存在的代码，你可以发送一条诊断信息到一个文件，或者到调试终端。另外，你还可以为你的产品指定特殊的调试输出级别。你应该尽快的在你的开发环境中使用这些功能，以确保你可以利用这些输出信息来修正在实际运行中没有预料到的一些问题。不要自己写诊断库除非你已经明白框架已经提供了哪些。

System.Diagnostics.Debug, System.Diagnostics.Trace 和 System.Diagnostics.EventLog 类提供了你在运行程序时要创建诊断信息的所有工具。前面两个类功能是基本上是一样的。不同之外是 Trace 类是由预处理

符 TRACE 控制的，而 Debug 类则是由 DEBUG 预处理符控制的。当你用 VS.net 开发一个项目时，TRACE 符号是同时在调试版和发布版中定义的。你可以为所有的发布版使用 Trace 类来创建诊断信息。EventLog 类提供了一个入口，通过这个入口，你的程序可以写一些系统日志。EventLog 类不支持运行时配置，但你可以把它封装到一个统一的简单接口中。

你可以在运行时控制诊断输出，.Net 框架使用一个应用程序配置文件来控制变化多样的运行时设置。这个是一个 XML 文件，在主应用程序运行时的目录中。这个文件与应用程序同名，但添加了一个.config 后缀。务更制块例如 MyApplication.exe 可能会有一个 MyApplication.exe.config 的 XML 文件来控制它。所所有的配置信息包含在一个 configuration 节点中：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

.Net 框架使用预定义的关键字来控制框架中一些类的行为。另外，你可以定义你自己的配置关键字和值。

你可以组合输出开关和 Trace.WriteLineIf()方法来控制应用程序的输出。你可以在应用程序外以默认的方式关闭这个输出，以便应用程序得到最好的性能。当发现问题时，你可以打开这个输出用于诊断和修正在实际中遇到的问题。WriteLineIf()只有在表达式为真时才输出：

```
bool _printDiagnostics = true;
Trace.WriteLineIf(_printDiagnostics,
  "Printing Diagnostics Today", "MySubSystem");
```

你所创建的输出开关用于控制输出的级别，一个输出开关可以是由应用程序配置文件定义的变量，可以是五种状态之一：关闭(Off)，错误(Error)，警告(Warning)，信息(Info)和详细(Verbose)。这些状态是环境的一部份，而且它们的值可以是从 0 到 4。这样你就可能为所有的子系统信息创建一个控制。 定义一个输出开关类然后初始化它就可以创建一个开关了：

```
static private TraceSwitch librarySwitch = new
  TraceSwitch("MyAssembly",
  "The switch for this assembly");
```

第一个参数是开关显示的名字，第二个参数是描述。这样，在运行时可以在应用程序配置文件中配置它们的值。下面就把 librarySwitch 设置成 Info：

```
<system.diagnostics>
  <switches>
    <add name="MyAssembly" value="3" />
  </switches>
</system.diagnostics>
```

如果你编辑了这个配置文件中开关的值，那么就修改了所有由那个开关控制的输出语句。

另一个任务：你须要配置你的输出到什么地方去。 默认是一个链接到 Trace 类上的监听者：一个 DefaultTraceListener 对象。DefaultTraceListener 发送信息到调试器，而且在它的失败方法(断言失败时调用)会打印一些诊断信息然后终止程序。在产品发布环境中，你不可能看到这样的信息。但你可是以配置不同的监听对象到产品发布环境中：那就是在应用程序的配置文件中添加监听者。下面就添加了一个 TextWriterTraceListener 到应用程序中：

```
<system.diagnostics>
  <trace autoflush="true" indentsize="0">
    <listeners>
      <add name="MyListener"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="MyListener.log"/>
```

```
    </listeners>
  </trace>
</system.diagnostics>
```

TextWriterTraceListener 把所有的诊断信息到打印到一个 MyListener.log 文件中。名字属性指定了监听者的名字，类型指定了作者监听对象的类型，它必须是从 System.Diagnostics.TraceListener 派生下来的。只有在极少数情况下你才创建自己的监听类，那就是你觉得.Net 框架的监听类不够用。initializeData 的值是一个字符串，用于传给对象的构造函数。而 TextWriterTraceListeners 把它用于文件名。

你可以小做一个扩展，让它可以在应用中每个部署的程序集上都可以简单的使用。对于每个程序集，添加一个类来跟踪程序集创建的诊断：

```
internal class MyAssemblyDiagnostics
{
  static private TraceSwitch myAssemblySwitch =
    new TraceSwitch("MyAssembly",
    "The switch for this assembly");
  internal static void Msg(TraceLevel l, object o)
  {
    Trace.WriteLineIf(myAssemblySwitch.Level >= l,
      o, "MyAssembly");
  }
  internal static void Msg(TraceLevel l, string s)
  {
    Trace.WriteLineIf(myAssemblySwitch.Level >= l,
      s, "MyAssembly");
  }
  // Add additional output methods to suit.
}
```

MyAssemblyDiagnostices 类根据一个开关来为这个程序集创建诊断信息。为了创建信息，调用按常规调用重载的 Msg 的任何一个就行了：

```
public void Method1()
{
  MyAssemblyDiagnostics.Msg(TraceLevel.Info,
    "Entering Method1.");
  bool rVal = DoMoreWork();
  if(rVal == false)
  {
    MyAssemblyDiagnostics.Msg(TraceLevel.Warning,
      "DoMoreWork Failed in Method1");
  }
  MyAssemblyDiagnostics.Msg(TraceLevel.Info,
    "Exiting Method1.");
}
```

利用一个全局的开关，你还可以组件特殊的程序集开关，来控制整个应用程序的输出：

```
internal static void Msg(TraceLevel l, object o)
{
  Trace.WriteLineIf (librarySwitch.Level >= l ||
    globalSwitch.Level >= l,
    o, "MyLibrary");
}
internal static void Msg(TraceLevel l, string s)
{
  Trace.WriteLineIf(librarySwitch.Level >= l ||
    globalSwitch.Level >= l,
    s, "MyLibrary");
}
```

这样，你就可以在应用程序上诊断信息，而且更友好的控制个别库文件的输出。在应用程序的任何地方，你都可以设置应用程序级的诊断到错误级，从而发现错误。当你有一个独立的问题时，你可以通过提高这个库的输出级别，从而精确的发现问题的源头。

在实际环境中，对于已经布署的应用程序，诊断库对于程序诊断和维护是必须的。但你自己不必写这些诊断库：.Net FCL 已经完成了核心的功能。尽可能完全的使用它们，然后在满足特殊要求时扩展它们。这样，即使是在产品发布的环境中也可以捕获所有的问题。

原则 37：使用标准的配置机制

Use the Standard Configuration Mechanism

我们要寻求一种避免直接写代码的应用程序配置和信息设置方法，我们已经创建了多种不同的策略来存储配置信息。而我们是要寻求一种正确的方法，我们要不断提高和改我们的想法，关于哪里是放置这些信息的好地方。INI 文件？这是 Windows3.1 做的事，配置信息的结构是受限制的，而且在文件名上可能还会与其它程序程序相冲突。注册表？是的，是这个正确的想法，但它也有它的限制。乱七八糟的程序可能会通过在注册表里写一些错误信息来严重破坏计算机。正因为写注册表存在危险，一个应用程序必须有管理员权限来写注册表的一部份。你的所有用户都会是以具有修改注册表权利的管理员身份在运行吗？希望不是，如果你使用注册表，而你的用户不是以管理员身份运行的，在试图读写注册表时，将会得到一个异常和错误。

谢天谢地，还有很多更好的方法来存储设置信息，这样你的程序可以根据用户的选择不同适应不同的行为，例如安装参数，机器设置，或者其它任何事情。.Net 框架提供了一个标准的设置位置，这样你的程序可以使用它来存储配置信息。这些存储位置是由应用程序特别指定的，而且当程序执行的机器上的用户被限制了权限时一样可以有效的工作。

只读的信息是属于配置文件的，XML 文件控制应用程序中不同类型的行为；定义的结构表指明了所有的元素和属性，而这些都是.NET FCL 从配置文件中分析出来的。 这些元素控制一些设置，例如正在使用那个框架版本，支持的调试级别(参见原则 36)，以及程序集的搜索路径。有一个节点你是必须要明白的，那就是 appSettings 部份，它可以同时应用与 web 应用程序和桌面应用程序。运行程序在启动时读取这一节点的信息，它加载所有的关键字和值到一个属于应用程序的名字值集合(NameValueCollection)中。这是你自己程序的一部份，你可以添加任何程序须要的值来控制程序行为。当修改配置文件时，也就修改了程序行为。

对于使用配置文件来说，ASP.Net 应用程序比桌面应用程序的伸缩性稍灵活一点。每个个虚拟目录可以有一个自己的配置文件，这个文件被每个虚拟目录依次读取，而每个虚拟目录也就对应一个 URL 的一部分。The most local wins. 例如，这个 URL：http://localhost/MyApplication/SubDir1/SubDir2/file.aspx 可能被 4 个不同的配置文件所控制。machine.config 最先读取，其次是在 MyApplication 中的 web.config 文件，接着是在 SubDir1 和

SubDir2 中的 web.config 文件。而它们每一个都可以修改前一个配置文件设置的值，或者是添加自己键/值对。你可以通过这种配置继承方式，来配置一个全局应用程序的参数选择，而且可以限制一些私有资源的访问。web 应用程序在不同的虚拟目录中有不同的配置。

在桌面应用程序中，对于每个应用程序域只有一个应用程序程序配置文件。.Net 运行时在载入每个可执行文件时，为它创建一个默认的应用程序域，然后读取一个预先军定义的配置文件到这个应用程序域中。默认的配置文件在与应用程序运行时的同一个目录中，而且就以<应用程序名>.<扩展名>.config 来命名的。例如：MyApp.exe 可能就有一个名为 MyApp.exe.config 的配置文件。appsettings 部份可以用于创建你自己的键/值对到应用程序中。

配置文件是存储一些控制程序行为的信息的最好的地方。但你可能很快会发现，应用程序没有 API 来写配置文件信息。配置文件不是用于存储任何有序设置的地方。不要急着写注册表，也不要自己乱写。这里有一个更好的方法让你配置桌面应用程序。

你可能须要定义配置文件的格式，而且把配置文件放到正确的地方。通过在全局设置上定义一些设置结构和添加公共的读写属性，你可以很简单的存储和取回这些设置：

```
[ Serializable() ]
public struct GlobalSettings
{
  // Add public properties to store.
}
```

XML 序列化来存储你的设置：

```
XmlSerializer ser = new XmlSerializer(
  typeof(GlobalSettings));
TextWriter wr = new StreamWriter("data.xml");
ser.Serialize(wr, myGlobalSettings);
wr.Close();
```

使用 XML 格式就意味着你的设置可以很容易的阅读，很容易的解析，以及很容易的去调试。如果须要，你可以对这些用户设置进行加密存储。这只是一个使用 XML 序列化的例子，不是对象持久序列化(参见原则 25)。XML 序列化存储文件，不是整个对象树。配置设置以及用户设置一般不会包含网状对象，而且 XML 序列化是一个简单的文件格式。

最后一个问题就是，应该在哪里存储这些信息。你应该在三个不同的地方放置配置信息文件。选择哪一个要根据配置的使用情况：全局，单用户，或者单用户且单机器。这三个位置可以通过调用 System.Environment.GetFolderPath() 而取得。你应该在 GetFolderPath()返回的路径后添加上应用程序的详细目录。请格外小心的在所有用户或者机器范围上填写信息。这样做要在目标机器是取得一些特权。

Environment.SpecialFolder.CommonApplicationData 返回存储信息的目录，这一目录是被机器上的所有用户所共享的。如果在一台机上使用的是默认安装，GetFolderPath(SpecialFolder.CommonApplicationData)会返回 C:\Documents and Settings\All Users\Application Data。存储在这一目录的的设置应该是被机器上的所有用户所使用的。当你要在这里创建信息时，让安装程序给你做或者以管理员模式进行。不应该在这里写一些用户级(译注：users 级是 windows 里的一个用户组，权利比管理员小。)的程序数据。偶然可能会让你的应用程序在用户机上没有足够的权限来访问。

Environment.SpecialFolders.ApplicationData 返回当前用户的路径，而且在网络上被所有机器共享的。在默认安装中，GetFolderPath(SpecialFolders.ApplicationData)返回 C:\Documents and Settings\<用户名>\Application Data。每个用户有他(或她)自己的应用程序数据目录。当用户登录到一个域是，使用这个列举进入到共享网络上，而且在网络上包含了用户的全局设置。存储在这里的数据只由当前用户使用，不管是从网络上的哪台机器登录过来的。

Environment.SpecialFolders.LocalApplicationData 返回一个特殊的目录，该目录队了存储设置信息以外，

同时也是一个用户的私人目录，它只属于从这台机器上登录的用户。一般 GetFolderPath(SpecialFolders.LocalApplicationData)返回：C:\Documents and Settings\<用户名>\Local Settings\Application Data

这三个不同的位置可以让你存储每个人的设置信息，给定用户的信息，或者是给定用户并给定机器的信息。具体的使用哪一个取决于应用程序。但考虑一些明显的例子：数据库链接字符串是一个全局设置，它应该存在在通用应用程序数据(Common Application Data) 目录中。一个用户的工作内容应该存在在应用程序数据(Application Data)目录中，因为它只取决于用户。窗口的位置信息应该在本地应用程序数据(Local Application Data)目录中。因为它们取决于机器上的用户的属性(不的机器可能有不同的分辨率)。

应该有一个特殊的目录，它为所有应用程序的所有用户设置存储，描述顶层的目录结构。这里，你须要在顶层目录结构下创建子目录。.Net 框架的 System.Windows.Application 类定义了一些属性，这些属性可以为你创建一些通用的配置路径。Application.LocalAppDataPath 属性返回 GetFolderPath(SpecialFolders.CommonApplicationData)+"\\CompanyName\\ProductName\\ProductVersion"的路径。类似的，Application.UserDataPath 和 Application.LocalUserDataPath 产生位于用户数据和本地数据目录下的路径名，这一目录包括公司，应用程序，以及版本号。如果你组合这些位置，你就可以为你自己公司的所有应用程序创建一个配置信息，或者为某一程序的所有版本，或者是特殊版本。

注意到了，这些目录中我没有提到过应用程序目录，就是在 Program Files 下的目录。你决不应该在 Program Files 或者在 Windows 系统目录以及子目录里写数据。这些目录要更高的特权，因此你并不能指望你的用户有权利来写这些数据。

当在哪里存储应用程序的设置数据成为一个很重要的问题，就像每个企业级用户到家庭用户所担心的机器安全问题一样时，把信息放在正确的位置就意味着对于使用你的应用程序的用户来说没有折衷的办法。你还是要给用户提供私人的感觉，用.Net 的顺序组合正确的位置，这样可以很容易的给每个用户一种私有的感觉，而且不用折衷安全问题。

原则 38：使用和支持数据绑定

Utilize and Support Data Binding

有经验的 Windows 程序员一定对写代码从一个控件上取值，以及把值存储到控件上很熟悉：

```
public Form1 : Form
{
  private MyType myDataValue;
  private TextBox textBoxName;
  private void InitializeComponent()
  {
    textBoxName.Text = myDataValue.Name;
    this.textBoxName.Leave += new
      System.EventHandler(this.OnLeave);
  }
  private void OnLeave(object sender, System.EventArgs e)
  {
    myDataValue.Name = textBoxName.Text;
  }
}
```

这太简单了，正如你知道的，重复代码。之所以不喜欢这样重复代码，就是因为应该有更好的方法。是的，.Net 框架支持数据绑定，它可以把一个对象的属性映射到控件的属性上：

```
textBoxName.DataBindings.Add ("Text",myDataValue, "Name");
```

上面的代码就把 textBoxName 控件的“Text”属性上绑定了 MyDataValue 对象的"Name"属性。在内部有两个对象，绑定管理(BindingManager)和流通管理(CurrencyManager)，实现了在控件与数据源之间的传输实现。你很可能已经见过为种结构的例子，特别是在 DataSet 和 DataGrid 之间的。你也很可能已经做过数据绑定的例子。你很可能只在表面上简单的使用过从数据绑定上得到的功能。你可以通过高效的数据绑定避免写重复的代码。

关于数据绑定的完整处理方案可能至少要花上一本书来说明，要不就是两本。Windows 应用程序和 Web 应用程序同时都支持数据绑定。比写一个完整的数据绑定论述要强的是，我确实想让你记住数据绑定的核心好处。首先，使用数据绑定比你自己写代码要简单得多。其次，你应该在对文字元素通过属性来显示时，尽可能的使用它，它可以很好的绑定。第三，在 Windows 窗体中，可以同步的对绑定在多控件上的数据，进行相关数据源的检测。

例如，假设只要在数据不合法时，要求将文字显示为红色，你可能会写这样的代码：

```
if (src.TextIsInvalid)
{
  textBox1.ForeColor = Color.Red;
} else
{
  textBox1.ForeColor = Color.Black;
}
```

这很好，但只要在文字源发生改变时，你要随时调用这段代码。这可能是在用户编辑了文字，或者是在底层的数据源发生改变时。这里有太多的事件要处理了，而且很多地方你可能会错过。但，使用数据绑定时，在 src 对象上添加一个属性，返回恰当的前景颜色就行了。

另一个逻辑可能是要根据文字消息的状态，来设置值可变化为恰当颜色的值：

```
private Color _clr = Color.Black;
public Color ForegroundColor
{
  get
  {
    return _clr;
  }
}
private string _txtToDisplay;
public string Text
{
  get
  {
    return _txtToDisplay;
  }
  set
  {
    _txtToDisplay = value;
    UpdateDisplayColor(IsTextValid());
  }
}
private void UpdateDisplayColor(bool bValid)
{
```

```
  _clr = (bValid) ? Color.Black : Color.Red;
}
```

简单的添加绑定到文本框里就行了：

```
textBox1.DataBindings.Add ("ForeColor",
src, "ForegroundColor");
```

当数据绑定配置好以后，textBox1 会根据内部源对象的值，用正确的颜色来绘制文本。这样，你就已经大大减少了从源数据到控件的数据来回传输。不再须要对不同地方显示不同颜色来处理很多事件了。你的数据源对象保持对属性的正确显示进行跟踪，而表单控件对数据绑定进行控制。

通过这个例子，我演示了 Windows 表单的数据绑定，同样的在 web 应用程序中也是一样的原则：你可以很好的绑定数据源的属性到 web 控件的属性上：

```
<asp:TextBox id=TextBox1 runat="server"
  Text="<%# src.Text %>"
  ForeColor="<%# src.ForegroundColor %>">
```

这就是说，当你创建一个应用程序在 UI 上显示的类型时，你应该添加一些必须的属性来创建和更新你的 UI，以便用户在必要时使用。

当你的对象不支持你要的属性时怎么办呢？那就把它封装成你想要的。看这样的数据结构：

```
public struct FinancialResults
{
  public decimal Revenue
  {
    get { return _revenue; }
  }
  public int NumberOfSales
  {
    get { return _numSales; }
  }
  public decimal Costs
  {
    get { return _cost;}
  }
  public decimal Profit
  {
    get { return _revenue - _cost; }
  }
}
```

要求你在一个表单上以特殊的格式信息来显示这些，如果收益为负，你必须以红色来显示收益。如果薪水小于 100，你应该用粗体显示。如果开销在 10 千(1 万)以上，你也应该用粗体显示。创建 FinancialResults 结构的开发者没有添加 UI 功能到这个结构上。这很可能是正确的选择，FinancialResults 应该限制它的功能，只用于存储实际的值。你可以创建一个新类型，包含 UI 格式化属性，以及在 FinancialResults 结构中的原始的存储属性：

```
public struct FinancialDisplayResults
{
  private FinancialResults _results;
  public FinancialResults Results
  {
    get { return _results; }
```

```
  }
  public Color ProfitForegroundColor
  {
    get
    {
      return (_results.Profit >= 0) ?
        Color.Black : Color.Red;
    }
  }
  // other formatting options elided
}
```

这样，你就创建了一个简单的数据结构来帮助你所包含的数据结构来进行数据绑定：

```
// Use the same datasource. That creates one Binding Manager
textBox1.DataBindings.Add ("Text", src, "Results.Profit");
textBox1.DataBindings.Add ("ForeColor",src,”ProfitForegroundColor");
```

我已经创建了一个只读的属性，用于访问核心的财政数据结构。这种构造在你试图支持对数据的读写操作时不能工作，FinancialResults 结构是值类型，这就是说获取访问器不提供对存储空间的访问，它只是返回一个拷贝。这样的方式很乐意返回一个拷贝，而这样的拷贝并不能在数据绑定中进行修改。然而，如果你试图对数据进行编辑时，FinancialResults 类应该是一个类，而不是一个结构(参见原则 6)。做为一个引用类型，你的获取访问器返回一个内部存储的引用，而且可以被用户编辑。内部的结构应该须要对存储的数据发生改变时做出响应。FinancialResults 应该触发事件来告诉其它代码这一状态的改变。

有一个很重要的事情要记住：把数据源用在同一表单中的所有相关控件上。使用 DataMember 属性来区别每个控件显示的属性。你可以像这样写绑定过程：

```
// Bad practice: creates two binding managers
textBox1.DataBindings.Add ("Text",src.Results, "Profit");
textBox1.DataBindings.Add ("ForeColor",src,"rofitForegroundColor");
```

这会创建两个绑定管理者，一个为 src 对象，另一个为 src.Results 对象。每个数据源由不同的绑定管理者控制，如果你想让绑定管理者在数据源发生改变时，更新所有的属性，你须要确保数据源是一致的。

你几乎可以在所有的 Windows 控件和 web 控件上使用数据绑定。在控件里显示的值，字体，只读状态，甚至是控件控件的位置，都可以成为绑定操作的对象。我的建议是创建类或者结构，包含一些用户要求的，以某种样式显示的数据。这些数据就是用于更新控件。

另外，在简单控件中，数据绑定经常出现在 DataSet 和 DataGrids 中。这非常有用，你把 DataGrid 绑定到 DataSet 上，然后 DataSet 中所有的值就显示了。如果你的 DataSet 有多个表，你甚至还可以在多个表中间进行导航。这不是很好吗？

好了，下面的问题就是如果你的数据集不包含你想显示的字段时该怎么办。这时，你必须添加一个列到 DataSet 中，这一列计算一些 UI 中必须的值。如果值可以用 SQL 表达式计算，那么 DataSet 可以为你完成。下面的代码就添加了一个列到 Employees 数据表中，用于显示格式化了名字：

```
DataTable dt = data.Tables[ "Employees" ];
dt.Columns.Add("EmployeeName",
  typeof(string),
   "lastname + ', ' + firstname");
```

通过添加列到 DataSet 中，你可以添加这些列到 DataGrid 上。你所创建的对象层，是在数据存储对象的最项层上，用于创建数据表现层给你的用户。

到目前为止，这一原则里所使用的都是 string 类型，.net 框架可以处理字符到数字的转化：它试图转化用户

的输入到恰当的类型。如果失败，原始的值会恢复。这是可以工作的，但用户完全没的反馈信息，他们的输出被安静的忽略了。你可以通过处理绑定过程中的转化事件来添加反馈信息。这一事件在绑定管理者从控件上更新值到数据源时发生。ParseEventArgs 包含了用户输入的文字 ，以及它所期望被转化的类型。你可以捕获这一事件，其后完成你自己的通知，也可以修改数据并且用你自己的值来更新数据：

```
private void Form1_Parse(object sender, ConvertEventArgs e)
{
  try {
    Convert.ToInt32 (e.Value);
  } catch
  {
    MessageBox.Show (
     string.Format("{0} is not an integer",
       e.Value.ToString()));
    e.Value = 0;
  }
}
```

你可能还要处理 Format 事件，这一个 HOOK，可以在数据从数据源到控件时格式化数据。你可以修改 ConvertEventArgs 的 Value 字段来格式化必须显示的字符串。

.Net 提供了通用的框架，可以让你支持数据绑定。你的工作就是为你的应用程序和数据提供一些特殊的事件句柄。Windows 表单和 Web 表单以及子系统都包含了丰富的数据绑定功能。框架库已经包含了所有你须要的工具，因此，你的 UI 代码应该真实的描述数据源和要显示的属性，以及在把这些元素存储到数据源时须要遵守的规则。你应该集中精力创建数据类型，用于描述显示的参数，然后 Winform 以及 Webform 的数据绑定完成其它的。不应该在把数据从用户控件到数据源之间进行传输时写相关的代码(译注：指用数据绑定，而不用其它的方法)。不管怎样，数据必须从你的业务对象关联到 UI 控件上与用户进行交互。通过创建类型层以及使用数据绑定的概念，你就可以少写很多代码。.Net 框架已经 同时在 Windows 和 Web 应用程序中为你处理了传输的工作。

原则 39：使用.Net 验证

Use .NET Validation

用户的输入可能是多种多样的：你必须在交互式的控件中尽可能的验证输入。写一些用户输入验证可能很做作，而且也有出错的可能，但还是很有必要的。不能太相信用户的输入，用户可能会输入任何内容导致异常发生，进而进行 SQL 注入式攻击。我们不希望任何类似这样的事情发生。你应该了解足够的信息来怀疑用户的输入。很好，每个人都应该这样做，这也就是为什么.Net 框架已经扩展了这样的功能，你可以使用这些功能从而使自己的代码编写工作减到最小，因为我们要对用户输入的每一块数据都要进行验证。

.Net 框架提供了不同的机制来验证用户的输入，分别可以用在 Web 和 Windows 应用程序中。Web 应用程序应该在浏览器上进行数据验证，一般是使用 JavaScript。一些验证控件在 HTML 面而中生成一些 JS 代码，这对你的用户来说是很有效的：在对每一项输入时，他们不用每次返回数据到服务上。这些 Web 控件是使用正则表达式的扩展功能来完成对用户输入的验证，这些验证可以在页面提交到服务器之间完成。即使如此，你还是要在服务器上做一些额外的验证，以免受到程序式的攻击。Windows 就用程序使用不同的模式。用户的输入可以直接在应用程序中用 C#代码来验证。所有的 Windows 控件都是可验证的，当你想通知用户的非法输入时。一般的模式是使用属性访问时的异常来指示非法的输入。UI 控件捕获这些异常然后显示错误给用户。

你可以使用 5 个 web 控件来处理 ASP.net 应用程序中的大多数验证任务。这 5 个控件都是由属性来控制这些要验证的特殊的字段。RequiredFieldValidator 强制用户在给定字段中输入一个值，RangeValidator 要求特殊的字段提供的值在给定范围内，这个范围可是一个数的大小，也可以是一个字符串的长度。CompareValidator 可以让你构造一个验证规则来验证表单上两个同的控件。这三个控件都很简单。最后两个控件提供了强大的功能，可

以让你根据你想要求的方法进行验证。RegularExpression 验证使用与此同时表达式来验证用户的输入。如果与比较返回匹配，输入的就是合法的。正则表达式是很有用的语言。你可以为你所有的实际情况创建正则表达式。VS.net 包含了一些验证的表达式，这可以帮助你开始学习它。这有一些帮助你学习更多正则表达式的有用资料，而且我强烈鼓励你学习它。但我不能跑题而不给你提供一些最常用的构造。表 5.1 显示了最常用的一些正则表达式元素，你可能会在你的应用程序中用来验证输入：

表 5.1 常用的正则表达式

构造 含意

[a-z] 匹配单个小写字符。括号内的字符集中的任何字符与单个字符匹配。

\d 任何数字。

^,$ ^表示串的开始, $表示结束。

\w 匹配任何单词.这是[A-Za-z0-9]简写。

(?NamedGroup\d{4,16}) 显示两个不同的常用元素，?NamedGroup 定义了一个特殊的变量来引用匹配。{4,16}匹配前面的构造至少 4 次最多 16 次。这一模式匹配一个至少包含 4 个但不超过 16 个数字的字符串。如果匹配存在，那么结果会存储在 NamedGroup 中以便后面使用。

(a|b|c) 匹配 a 或 b 或 c。 用竖线分开的是选择操作：输入的可是其中的任何一个。

(?(NamedGroup)a|b) 可选的。这与 C#里的三元操作等效，也就是说，如果 NamedGroup 存在，匹配 a,否则匹配 b.

(译注，关于正则表达式这里只是简单的说明了一下。觉得作者在这里写正则表达式很是不伦不类，即不全也不精。)

使用这些及正则表达式的构造，你可以发现你可以验证用户提交给你的任何内容。如果正则表达式还不够，你还可以通过从 CustomValidator 派生一个新在类添加你自己的验证。这是一个不小的工作，而且我尽可能的避免它。当你用 C#写了一服务器函数来验证数据后，还要用 ECMAscript 写一个客户端的验证函数。我讨厌同样的事做两遍，而且我也尽可能的避免用 ECMAscript 写任何内容，所以，我喜欢粘贴正则表达式式。

例如，这有一个正则表达式，用于验证 US 的电话号码。它接受区号用括号括起来的，或者没有括号的，然后就是区号和号码之间的空格，交换局号(exchange)，以及号码。区号和交换局号之间的横线也是可选的：

((\(\s*\d{3}\s*\))|(\d{3}))-?\s*\d{3}\s*-\s*\d{4}

通过查验每一个组的表达式，这样的逻辑是很清楚的：

((\(\s*\d{3}\s*\))|(\d{3}))-?

这和区号匹配，它充许(XXX)或者 XXX 的形式，其中 XXX 是三个数字。任何在数字周围的空白字符是充许的。最后两个字符，-和？，是许可但不要求一个横线。

剩下的部份用于匹配电话的 XXX-XXXX 部份。\s 匹配任意的空白，\d{3}匹配三个数字，\s*-\s*匹配一个围绕在数字边上的空白字符。最后，\d{4}精确匹配 4 个数字。

windows 验证工作方法小有不同，你没有预先的验证分析。相反，你要写一个事件句柄到 System.Windows.Forms.Control.Validating 事件上，或者，如果你创建了你自己的控件，重载 OnValidating 方法(参见原则 35)。下面是一个标准的方法：

```
private void textBoxName_Validating(object sender,
  System.ComponentModel.CancelEventArgs e)
{
  string error = null;
```

```
  // Perform your test
  if (textBoxName.Text.Length == 0)
  {
    // If the test fails, set the error string
    // and cancel the validation event.
    error = "Please enter a name";
    e.Cancel = true;
  }
  // Update the state of an error provider with
  // the correct error text. Set to null for no
  // error.
  this.errorProviderAll.SetError(textBoxName, error);
}
```

你有几个小工作要完成，以确保没有不合法的输入愉愉的混过去了。每一个控件包含一个 CausesValidation 属性，这个属性决定这个控件是否参与验证。一般情况，你应该让所有控件的这一属性为真，除非是 Cancel 按钮。如果你忘记了，用户还必须输出正确的值以后才能取消对话框。第二个小任务是添加 OK 句柄来强制验证所有的控件。验证只有在用户访问和离开控件时触发。如果用户打开了一个窗口，然后马上点 OK，你的所有验证代码都不会执行。为了修正这个，你要添加 OK 按钮句柄，来访问所有的控件，然后强制验证它们。下面两个常规方法显示了如何正确的完成任务。递归方法处理控件以及它所包含的控件：Tab 页面，控件组以及控件面板：

```
private void buttonOK_Click(object sender,
  System.EventArgs e)
{
  // Validate everyone:
  // Here, this.DialogResult will be set to
  // DialogResult.OK
  ValidateAllChildren(this);
}
private void ValidateAllChildren(Control parent)
{
  // If validation already failed, stop checking.
  if(this.DialogResult == DialogResult.None)
    return;
  // For every control
  foreach(Control c in parent.Controls)
  {
    // Give it focus
    c.Focus();
    // Try and validate:
    if (!this.Validate())
    {
      // when invalid, don't let the dialog close:
      this.DialogResult = DialogResult.None;
      return;
    }
    // Validate children
```

```
        ValidateAllChildren(c);
      }
    }
```

这些代码可以处理大多数情况。一个特殊的快捷应用就是 DataGrid/DataSet 的组合。在设计时指定 ErrorProvider 的 DataSource 以及 DataMember 属性：

```
ErrProvider.DataSource = myDataSet;
ErrProvider.DataMember = "Table1";
```

或者在运行时，调用 BindToDataAndErrors 方法来同时设置：

```
ErrProvider.BindToDataAndErrors(myDataSet, "Table1");
```

错误会在设置 DataRow.RowError 属性以及调用 DataRow.SetColumnError 方法时显示特殊的错误。ErrorProvider 会在 DataGrid 的原始的行上的特殊单元格里显示红色的警告图标。

大概的了解(whirlwind tour)了一下.net 框架里的控件验证，这可能对你很有帮助，在很多应用程序中，你都可以创建出你所须要的高效验证。用户的输入不能完全信任：用户可能会出现错误，而且有时会有一些恶意的用户试图破坏你的应用程序。通过.Net 框架已经提供的服务，你可以减少你自己的代码编写工作。验证所有用户的输入，但要使用已经提供了的高效工具。

原则 40：根据需求选择集合

Match Your Collection to Your Needs

“哪种集合是最好的？”答案是：“视情况而定。” 不同的集合有不同的性能，而且在不同的行为上有不同的优化。.Net 框架支持很多类似的集合：链表，数组，队列，栈，以及其它的一些集合。C#支持多维的数组，它的性能与一维的数组和锯齿数组都有所不同。.Net 框架同样包含了很多特殊的集合，在你创建你自己的集合类之前，请仔细参阅这些集合。你可以发现很多集合很快，因为所有的集合都实现了 ICollection 接口。在说明文档中列出了所有实现了 ICollection 接口的集合，你将近有 20 多个集合类可用。

为了选择适合你使用的集合，你须要考虑在该集合上使用的操作。为了创建一个有伸缩性的程序，你须要使用一些实现了接口的集合类，这样当你发现某个假设的集合不正确时，你可以用其它不同的集合来代替它(参见原则 19)。

.Net 框架有三种不同类型的集合：数组，类似数组的集合，以及基于散列值的集合。数组是最简单的也是一般情况下最快的集合，因此我们就从它开始。这也是你经常要使用到的集合类。

你的第一选择应该经常是 System.Array 类，确切的说，应该是一个数组类型的类。 选择数组类的首要原因，也是最重要的原因是数组是类型安全的。所有其它集合都是存储的 System.Object 引用，直到 C#2.0 才引入了范型(参见原则 49)。当你申明一个数组时，编译器为你的类型创建一个特殊的 System.Array 派生类。例如：这样创建了申明并创建了一个整型的数组：

```
private int [] _numbers = new int[100];
```

这个数组存储的是整型，而不是 System.object 的引用。这很重要，因为你在一个数组上添加，访问，删除值类型时，可以避免装箱与拆箱操作的性能损失(参见原则 17)。上面的初始化操作创建了一个一维的数组，里面存放了 100 个整数。所有被数组占用的内存都被清 0 了。值类型数组都是 0，而引用数组都是 null。所有在数组里的元素可以用索引访问：

```
int j = _numbers[ 50 ];
```

另外，访问数组时还可以用 foreach 迭代，或者使用枚举器：

```
foreach (int i in _numbers)
  Console.WriteLine(i.ToString());
// or:
IEnumerator it = _numbers.GetEnumerator();
```

```
while(it.MoveNext())
{
  int i = (int) it.Current;
  Console.WriteLine(i.ToString());
}
```

如果准备存储单一次序的对象，你应该使用数组。但实际上你的数据结构往往比这复杂的多。这很快诱使我们回到了 C 风格上的锯齿数组，那就是一个数组里存储另一个数组。有些时候这确实是你想要的，每个外层元素都是内层的一个数组：

```
public class MyClass
{
  // Declare a jagged array:
  private int[] [] _jagged;
  public MyClass()
  {
    // Create the outer array:
    _jagged = new int[5][];
    // Create each inner array:
    _jagged[0] = new int[5];
    _jagged[1] = new int[10];
    _jagged[2] = new int[12];
    _jagged[3] = new int[7];
    _jagged[4] = new int[23];
  }
}
```

每个内层的一维数组可以有同不的大小。在需要不同大小的数组时可以使用锯齿数据。使用锯齿数组的缺点是在列方面上的访问是底效的。检查一个每行上有三个列的锯齿数组时，每次访问都要做两个检测 。在行 0 列 3 上的元素和在行 1 列 3 的元素没有任何关系。只有多维数组才可在以列方面上的访问有一定的效率。过去，C 和 C++程序使用二维或多维数组时是把它们映射到一个一维数组上。对于老式的 C 和 C++程序员，这样的标记应该很清楚：

```
double num = MyArray[ i * rowLength + j ];
```

这个世界上的其它人可能更喜欢这样：

```
double num = MyArray[ i, j ];
```

但，C 和 C++其实并不支持多维数组。C#可以，而且对于多维数组的语法：当你创建一个真实的多维数组时，对于编译器和你来说意义都是很清楚的。创建多维数组时只用在熟悉的一维数组标记上扩展一下就行了：

```
private int[ , ] _multi = new int[ 10, 10 ];
```

前面的申明创建了一个二维数组，而且是 10X10 的 100 个元素。在多维数组上的每一维的长度总是一致的。编译器利用这一特性，可以创建更高效的代码。初始化锯齿数组时须要多次使用初始语句。而在我前面的例子里(译注：指这里：private int[] [] _jagged;)，你须要 5 个语句。更大的数组或者更多维的数组须要更多的初始代码，而且你要手动编写。然而 ，多维数组在初始化时只要指定多少维就行了。此外，多维数组在初始化元素时更有效。以于值类型数组在初始化时是直接在有效的数组索引上包含该值，所有的内容就是 0。而引用类型的数组则是 null。数组的数组在内层上都包含 null 引用。

访问多维数组时比访问锯齿数组要快得多，特殊是在列或者对角线上访问时。编译器在数组的每个维上是使用的指针算法。锯齿数组则要为每个一维数组查找正确的(指针引用)值。

多维数组可以当成数组在很多情况下使用。假设你创建一个棋盘游戏，你可能须要在一个格子上创建一个 64 个方格的棋盘：

```
private Square[ , ] _theBoard = new Square[ 8, 8 ];
```

这个初始化创建了数组来存储方格，假设这些方格是引用类型，这些方格自己还没有创建，而且每个元素还是 null。为了初始化每个元素，你还要检测数组上的每一维元素：

```
for (int i = 0; i < _theBoard.GetLength(0); i++)
  for(int j = 0; j < _theBoard.GetLength(1); j++)
    _theBoard[ i, j ] = new Square();
```

你你还有更灵活的方法来访问多维数组，你可以给定个别的元素索引来访问该元素：

```
Square sq = _theBoard[ 4, 4 ];
```

如果你要迭代整个集合，你还可以使用迭代器：

```
foreach(Square sq in _theBoard)
  sq.PaintSquare();
```

与锯齿数组相比，你可能要这样写：

```
foreach(Square[] row in _theBoard)
  foreach(Square sq in row)
    sq.PaintSquare();
```

锯齿数组里的每一个新的维引用了另一个 foreach 语句。然而，对于一个多维数组，每一个 foreach 语句都要产生代码来检测数组每个维上的界线。foreach 语句在每个维上生成特殊的代码来迭代数组。foreach 循环生成的代码与你这样手写是一样的：

```
for (int i = _theBoard.GetLowerBound(0);
  i <= _theBoard.GetUpperBound(0); i++)
  for(int j = _theBoard.GetLowerBound(1);
    j <= _theBoard.GetUpperBound(1); j++)
    _theBoard[ i, j ].PaintSquare();
```

考虑在内层循环上所有对 GetLowerBound 和 GetUpperBound 的调用，这看上去是认低效的，但这确实是很高效的结构。JIT 编译器对数组类是很了解的，它们会缓存数组界线，而且可以识别内层的迭代界线从而省略它们(参见原则 11)。

数组类的两个主要的不利因素可能会让你考虑使用.Net 框架里的其它集合类。第一个影响就是改变数组的大小：数组不能动态的改变大小。如果你须要修改任意数组及任意维数的大小，你必须创建一个新的数组，然后拷贝所有已经存在的元素到新的数组里。改变大小要花时间：因为新的数组要分配，而且所有的元素要拷贝到新的数组里。尽管拷贝和移动在托管堆上并不是像使用 C 和 C++的年代那样开销昂贵，但还是很花时间。更重要的是，它会让数据使用陈旧。考虑下面的代码：

```
private string [] _cities = new string[ 100 ];
public void SetDataSources()
{
  myListBox.DataSource = _cities;
}
public void AddCity(string CityName)
{
  String[] tmp = new string[ _cities.Length + 1 ];
  _cities.CopyTo(tmp, 0);
  tmp[ _cities.Length ] = CityName;
  _cities = tmp; // swap the storage.
}
```

即使在调用 AddCity 之后，列表框的数据源还是使用拷贝前的旧数据。你新添加的城市根本没有在列表框中显示。

ArrayList 类是在数组上的一个高度抽象类。ArrayList 集合混合了一维数组和链表的语义。你可以在 ArrayList 使用迭代，而且你可以调整它的大小。ArrayList 基本上把所有责任都委托给了它所包含了数组上，这就是说 ArrayList 类与数组类在性能上很接近。它的最大好处就是在你不清楚你的集合具体要多大时，ArrayList 类要比数组类好用得多。ArrayList 可以随时扩展和收缩。但你还是要在移动和拷贝元素上有性能损失，但这些算法的代码已经写好了而且经过了测试。因为内部存储的数组已经封装在 ArrayList 对象内，因此陈旧数据的情况就不存在了：用户的指针是指向 ArrayList 对象而不是内部的数组。.Net 框架里的 ArrayList 集合有点类似 C++中标准库里的向量类。

队列和栈类在 System.Array 类上提供了特殊的接口。这些特殊的接口是自定义的先进先出的队列接口和后进先出的栈接口。时刻记住，这些集合的内部存储都是基于数组的。在修改集合的大小时同样会得到性能的损失。

(译注：对于这样的一些集合，在初始化时都有一个大小的参数，应该尽可能的给定集合的容量大小，这里的大小并不是一次性申明和使用的，而是集合中的元素小于这个大小时，不会有移动和拷贝元素的性能损失。因此合理的选择这些集合的容量大小是很关键的。队列的默认大小是 32，而栈的默认大小是 10，ArrayList 默认是 0。而它们的构造函数都有一个重载版本可以指定容量。)

.Net 中的集合没有包含链表(linked list)结构。垃圾回收器的高效使得列表(list)结构在实际使用是占用时间最小，因此也让它成为了最佳选择。如果你确实须要链表的行为，你有两个选择。如果你因为希望经常添加和删除里面的元素而选择了列表，那么使用 null 引用的字典类。简单的存储关键字，你就可以使用 ListDictionary 类了，而且它用键/值对应方式实现了单向链表。或者你可以选择 HybridDictionary 类，这是一个为小集合使用了 ListDictionary 的类，而且对于大集合会转化到 HashTable 上来。这些集合以及很多其它内容都在 System.Collections.Specialized 名字空间里。然而，如果因为用户想控制次序而你想使用列表结构，你可以使用 ArrayList 集合。ArrayList 可以在任何位置上执行插入操作，即使它的内部是使用的数组存储方式。

另外两个类支持基于字典的集合：SortedList 和 Hashtable。它们都是包含键/值对应的。SortedList 以键进行排序，而 Hashtable 则没有。Hashtable 在给定的键上进行查找时更快，但 SortedList 提供了基于键的有序迭代。Hashtable 通过键的散列值进行查找。如果散列值是高效的，它的查找时间是一个常量，O(1)。有序列表是使用的折半查找算法来查找关键值。这中一个对数的算法操作 ：O(ln n)。

最后，就是 BitArray 类。正如它的名字所说的，它存储位数值。BitArray 是以整数的数组来存储数据的。每个整型存储一个 32 位的二进制值。这让 BitArray 类是压缩的，但它还是会降低性能。BitArray 上的每个 get 和 set 操作要在存储的整数上完成位操作，这要查找另外的 31 个位。BitArray 包含的一些方法在很多值类型上实现了立即的 Boolean 操作：OR, XOR, AND, 和 NOT。这些方法用 BitArray 做为参数，而且 BitArray 可以用于快速的多位 mask 操作，BitArray 是一个为位操作进行优化了的容器；经常使用 mask 操作时，要存储这些位标记的数据时应该使用它。但不要用它灰替换一般用图和 Boolean 值数组。

对于数组类的概念而言，在 C#的 1.x 发布版中，没有一个集合类是强类型的。它们都是存储的对象引用。C#的范型将会在所有这些拓扑上包含新的版本。这会是一个最好的方法来创建类型安全的集合。同时，目前的 System.Collections 名字空间中包含了抽象的基类，利用这些基类，你可以在类型不安全的集合上创建你自己的类型安全的接口：CollectionBase 和 ReadOnlyCollectionBase 提供了列表和向量结构的基类。DictionaryBase 提供了键/值对应的基类。DictionaryBase 类是建立在 Hashtable 上的，它的性能与 Hashtable 是一致的。

(译注：在 C#1.x 中，我们其实可以合建类型安全的集合，当然这只是利用编译器一些特性，例如：

```
public class MyObject
{
}
public class MyObjectCollection : ArrayList
{
 [ObsoleteAttribute("Please use specify object.",true)]
 public override int Add(object value)
```

```
 {
  return base.Add (value);
 }
 public int Add(MyObject value)
 {
  return base.Add (value);
 }
 new public MyObject this[int index]
 {
  get
  {
   return base[index] as MyObject;
  }
  set
  {
   base[index] = value;
  }
 }
}
```

这样就可以简单的验证集合的类型了，当然，本质上还是类型不安全的，这只是一个折衷的办法。

)

任何时候，你的类型包含集合时，你可能想要把这些集合在你的类上暴露给你的用户。你有两个方法：使用索引或者使用 IEnumerable 接口。记住，在这一原则的一开始，我就说明过，你可以直接使用[]标记来访问一个数组里的元素，而且你可以用 foreach 来迭代数组里的任何元素。

你可以为你的类创建多维的索引，以 C++里你可能须要重载[]操作。而在 C#里，你可以创建多维索引：

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

添加对索引器的支持就是说你的类型包含一个集合。这也就是说你应该支持 IEnumerable 接口。IEnumerable 提供了标准的机制来迭代所有的集合元素：

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

GetEnumerator 方法返回一个实现了 IEnumerator 接口的对象。IEnumerator 接口支持集合的历遍：

```
public interface IEnumerator
{
  object Current
  { get; }
```

```
  bool MoveNext();
  void Reset();
}
```

另外，在使用 IEnumerable 接口时，如果你的类型模型是数组，你应该考虑 IList 和 ICollection 接口。如果你的类型模型是字典，你应该考虑实现 IDictionary 接口。你可以自己创建对这些功能强大的接口的实现，而且你可能要花上更多的几页代码来解释如何实现。但这里有一个简单的解决方案：在你创建自己的特殊集合类时从 CollectionBase 或者 DictionaryBase 派生你的类。

让我们回顾一下今天的内容，最好的集合取决于你对操作的实现，以及应用程序对空间和时间的目标要求。在大多数情况下，数组提供了最高效的容器。C#里扩展的多维数组，可以简单的实现多维的结构，而且没有性能的损失。当我们的应用程序须要对元素进行更灵活的添加和移动操作时，使用众多集合中更活越的一个就行了。最后，当你创建自己的集合模型类时，不论什么时候都实现索引器和 IEnumerable 接口。

原则 41：选择 DataSet 而不是自定义的数据结构

Prefer DataSets to Custom Structures

因为两个原则，把 DataSet 的名声搞的不好。首先就是使用 XML 序列化的 DataSet 与其它的非.Net 代码进行交互时不方便。如果在 Web 服务的 API 中使用 DataSet 时，在与其它没有使用.Net 框架的系统进行交互时会相当困难。其次，它是一个很一般的容器。你可以通过欺骗.Net 框架里的一些安全类型来错误 DataSet。但在现代软件系统中，DataSet 还可以解决很多常规的问题。如果你明白它的优势，避免它的缺点，你就可以扩展这个类型了。

DataSet 类设计出来是为了离线使用一些存储在相关数据库里的数据。你已经知道它是用来存储 DataTable 的，而 DataTable 就是一个与数据库里的结构在行和列上进行匹配的内存表。或许你已经看到过一些关于 DataSet 支持在内部的表中建立关系的例子。甚至还有可能，你已经见过在 DataSet 里验证它所包含的数据，进行数据约束的例子。

但不仅仅是这些，DataSet 还支持 AcceptChanges 和 RejectChanges 方法来进行事务处理，而且它们可以做为 DiffGrams 存储，也就是包含曾经修改过的数据。多个 DataSet 还可以通过合并成为一个常规的存储库。DataSet 还支持视图，这就是说你可以通过标准的查询来检测数据里的部份内容。而且视图是可以建立在多个表上的。

然而，有些人想开发自己的存储结构，而不用 DataSet。因 DataSet 是一个太一般的容器，这会在性能上有所损失。一个 DataSet 并不是一个强类型的存储容器，其实存储在里面的对象是一个字典。而且在里的表中的列也是字典。存储在里的元素都是以 System.Object 的引用形式存在。这使得我们要这样写代码：

```
int val = (int)MyDataSet.Tables[ "table1" ].
  Rows[ 0 ][ "total" ];
```

以 C#强类型的观点来看，这样的结构是很麻烦的。如果你错误使用 table1 或者 total 的类型，你就会得到一个运行时错误。访问里面的数据元素要进行强制转化。而这样的麻烦事情是与你访问里面的元素的次数成正比的，与其这样，我们还真想要一个类型化的解决方法。那就让我们来试着写一个 DataSet 吧，基于这一点，我们想要的是：

```
int val = MyDataSet.table1.Rows[ 0 ].total;
```

当你看明白了类型化的 DataSet 内部的 C#实现时，就会知道这是完美的。它封装了已经存在的 DataSet，而且在弱类型的访问基础上添加了强类型访问。你的用户还是可以用弱类型 API。但这并不是最好的。

与它同时存在的，我会告诉你我们放弃了多少东西。我会告诉你 DataSet 类里面的一些功能是如何实现的，也就是在我们自己创建的自定义集合中要使用的。你可能会觉得这很困难，或者你觉得我们根本用上不同 DataSet 的所有功能，所以，代码并不会很长。OK，很好，我会写很长的代码。

假设你要创建一个集合，用于存储地址。每一个独立的元素必须支持数据绑定，所以你我创建一个具有下面公共属性的结构：

```
public struct AddressRecord
{
  private string _street;
  public string Street
  {
    get { return _street; }
    set { _street = value; }
  }
  private string _city;
  public string City
  {
    get { return _city; }
    set { _city = value; }
  }
  private string _state;
  public string State
  {
    get { return _state; }
    set { _state = value; }
  }
  private string _zip;
  public string Zip
  {
    get { return _zip; }
    set { _zip = value; }
  }
}
```

下面，你要创建这个集合。因为我们要类型安全的集合，所以我们要从 CollectionsBase 派生：

```
public class AddressList : CollectionBase
{
}
```

CollectionBase 支持 IList 接口，所以你可以使用它来进行数据绑定。现在，你就发现了你的第一个问题：如果地址为空，你的所有数据绑定行就失败了。而这在 DataSet 里是不会发生的。数据绑定是由基于反射的迟后绑定代码组成的。控件使用反射来加载列表里的第一个元素，然后使用反射来决定它的类型以及这个类型上的所有成员属性。这就是为什么 DataGrid 可以知道什么列要添加。它会在集合中的第一个元素上发现所有的公共属性，然后显示他们。当集合为空时，这就不能工作了。你有两种可能来解决这个问题。第一个方法有点丑，但是一个简单的方法：那就是不充许有空列表存在。第二个好一些，但要花点时间：那就是实现 ITypedList 接口。ITypedList 接口提供了两个方法来描述集合中的类型。GetListName 返回一个可读的字符串来描述这个列表。GetItemProperties 则返回 PropertyDescriptors 列表，这是用于描述每个属性的，它要格式化在表格里的：

```
public class AddressList : CollectionBase
{
  public string GetListName(
    PropertyDescriptor[ ] listAccessors)
```

```
  {
    return "AddressList";
  }
  public PropertyDescriptorCollection
    GetItemProperties(
    PropertyDescriptor[ ] listAccessors)
  {
    Type t = typeof(AddressRecord);
    return TypeDescriptor.GetProperties(t);
  }
}
```

这稍微好一点了，现在你你已经有一个集合可以支持简单的数据绑定了。尽管，你失去了很多功能。下一步就是要实现数据对事务的支持。如果你使用过 DataSet，你的用户可以通过按 Esc 键来取消 DataGrid 中一行上所有的修改。例如，一个用户可能输入了错误的城市，按了 Esc，这时就要原来的值恢复过来。DataGrid 同样还支持错误提示。你可以添加一个 ColumnChanged 事件来处理实际列上的验证原则。例如，州的区号必须是两个字母的缩写。使用框架里的 DataSet，可以这样写代码：

```
ds.Tables[ "Addresses" ].ColumnChanged +=new
  DataColumnChangeEventHandler(ds_ColumnChanged);
private void ds_ColumnChanged(object sender,
  DataColumnChangeEventArgs e)
{
  if (e.Column.ColumnName == "State")
  {
    string newVal = e.ProposedValue.ToString();
    if (newVal.Length != 2)
    {
      e.Row.SetColumnError(e.Column,
        "State abbreviation must be two letters");
      e.Row.RowError = "Error on State";
    }
    else
    {
      e.Row.SetColumnError(e.Column,
        "");
      e.Row.RowError = "";
    }
  }
}
```

为了在我们自己定义的集合上也实现这样的概念，我们很要做点工作。你要修改你的 AddressRecord 结构来支持两个新的接口，IEditableObject 和 IDataErrorInfo。IEditableObject 为你的类型提供了对事务的支持。IDataErrorInfo 提供了常规的错误处理。为了支持事务，你必须修改你的数据存储来提供你自己的回滚功能。你可能在多个列上有错误，因此你的存储必须包含一个包含了每个列的错误集合。这是一个为 AddressRecord 做的更新的列表：

```
public class AddressRecord : IEditableObject, IDataErrorInfo
```

```
{
    private struct AddressRecordData
    {
      public string street;
      public string city;
      public string state;
      public string zip;
    }
    private AddressRecordData permanentRecord;
    private AddressRecordData tempRecord;
    private bool _inEdit = false;
    private IList _container;
    private Hashtable errors = new Hashtable();
    public AddressRecord(AddressList container)
    {
      _container = container;
    }
    public string Street
    {
      get
      {
        return (_inEdit) ? tempRecord.street :
          permanentRecord.street;
      }
      set
      {
        if (value.Length == 0)
          errors[ "Street" ] = "Street cannot be empty";
        else
        {
          errors.Remove("Street");
        }
        if (_inEdit)
          tempRecord.street = value;
        else
        {
          permanentRecord.street = value;
          int index = _container.IndexOf(this);
          _container[ index ] = this;
        }
      }
    }
    public string City
    {
      get
```

```
  {
    return (_inEdit) ? tempRecord.city :
      permanentRecord.city;
  }
  set
  {
    if (value.Length == 0)
      errors[ "City" ] = "City cannot be empty";
    else
    {
      errors.Remove("City");
    }
    if (_inEdit)
      tempRecord.city = value;
    else
    {
      permanentRecord.city = value;
      int index = _container.IndexOf(this);
      _container[ index ] = this;
    }
  }
}
public string State
{
  get
  {
    return (_inEdit) ? tempRecord.state :
      permanentRecord.state;
  }
  set
  {
    if (value.Length == 0)
      errors[ "State" ] = "City cannot be empty";
    else
    {
      errors.Remove("State");
    }
    if (_inEdit)
      tempRecord.state = value;
    else
    {
      permanentRecord.state = value;
      int index = _container.IndexOf(this);
      _container[ index ] = this;
    }
```

```
  }
}
public string Zip
{
  get
  {
    return (_inEdit) ? tempRecord.zip :
      permanentRecord.zip;
  }
  set
  {
    if (value.Length == 0)
      errors["Zip"] = "Zip cannot be empty";
    else
    {
      errors.Remove ("Zip");
    }
    if (_inEdit)
      tempRecord.zip = value;
    else
    {
      permanentRecord.zip = value;
      int index = _container.IndexOf(this);
      _container[ index ] = this;
    }
  }
}
public void BeginEdit()
{
  if ((! _inEdit) && (errors.Count == 0))
    tempRecord = permanentRecord;
  _inEdit = true;
}
public void EndEdit()
{
  // Can't end editing if there are errors:
  if (errors.Count > 0)
    return;
  if (_inEdit)
    permanentRecord = tempRecord;
  _inEdit = false;
}
public void CancelEdit()
{
  errors.Clear();
```

```
    _inEdit = false;
  }
  public string this[string columnName]
  {
    get
    {
      string val = errors[ columnName ] as string;
      if (val != null)
        return val;
      else
        return null;
    }
  }
  public string Error
  {
    get
    {
      if (errors.Count > 0)
      {
        System.Text.StringBuilder errString = new
          System.Text.StringBuilder();
        foreach (string s in errors.Keys)
        {
          errString.Append(s);
          errString.Append(", ");
        }
        errString.Append("Have errors");
        return errString.ToString();
      }
      else
        return "";
    }
  }
}
```

花了几页的代码来支持一些已经在 DataSet 里实现的了的功能。实际上，这还不能像 DataSet 那样恰当的工作。例如，交互式的添加一个新记录到集合中，以及支持事务所要求的 BeginEdit, CancelEdit, 和 EndEdit 等。 你要在 CancelEdit 调用时检测一个新的对象而不是一个已经修改了的对象。CancelEdit 必须从集合上移除这个新的对象，该对象应该是上次调用 BeginEdit 时创建的。对于 AddressRecord 来说，还有很多修改要完成，而且一对事件还要添加到 AddressList 类上。

最后，就是这个 IBindingList 接口。这个接口至少包含了 20 个方法和属性，用于控件查询列表上的功能描述。你必须为只读列表实现 IBindingList 或者交互排序，或者支持搜索。在你取得内容之前就陷于层次关系和导航关系中了。我也不准备为上面所有的代码添加任何例子了。

几页过后，再问问你自己，还准备创建你自己的特殊集合吗？或者你想使用一个 DataSet 吗？除非你的集合是一个基于某些算法，对性能要求严格的集合，或者必须有轻便的格式，就要使用自己的 DataSet，特别是类型化的 DataSet。这将花去你大量的时间，是的，你可以争辩说 DataSet 并不是一个基于面向对象设计的最好的例子。类

型化的 DataSet 甚至会破坏更多的规则。但，使用它所产生的代码开发效率，比起自己手写更优美的代码所花的时间，这只是其中一小部份

原则 42：使用特性进行简单的反射

Utilize Attributes to Simplify Reflection

当你创建了一个与反射相关的系统时，你应该为你自己的类型，方法，以及属性定义一些自己的特性，这样可以让它们更容易的被访问。自定义的特性标示了你想让这些方法在运行时如何被使用。特性可以测试一些目标对象上的属性。测试这些属性可以最小化因为反射时可能而产生的类型错误。

假设你须要创建一个机制，用于在运行时的软件上添加一个菜单条目到一个命令句柄上。这个须要很简单：放一个程序集到目录里，然后程序可以自己发现关于它的一些新菜单条目以及新的菜单命令。这是利用反射可以完成的最好的工作之一：你的主程序须要与一些还没有编写的程序集进行交互。这个新的插件同样不用描述某个集合的功能，因为这可以很好的用接口来完成编码。

让我们为创建一个框架的插件来开始动手写代码吧。你须要通过 Assembly.LoadFrom() 函数来加载一个程序，而且要找到这个可能提供菜单句柄的类型。然后须要创建这个类型的一个实例对象。接着还要找到这个实例对象上可以与菜单命令事件句柄的申明相匹配的方法。完成这些任务之后，你还须要计算在菜单的什么地方添加文字，以及什么文字。

特性让所有的这些任务变得很简单。通过用自己定义的特性来标记不同的类以及事件句柄，你可以很简单的完成这些任务：发现并安装这些潜在的命令句柄。你可以使用特性与反射来协作，最小化一些在原则 43 中描述的危险事情。

第一个任务就是写代码，发现以及加载插件程序集。假设这个插件在主执行程序所在目录的子目录中。查找和加载这个程序集的代码很简单：

```
// Find all the assemblies in the Add-ins directory:
string AddInsDir = string.Format("{0}/Addins",
  Application.StartupPath);
string[] assemblies = Directory.GetFiles(AddInsDir, "*.dll");
foreach (string assemblyFile in assemblies)
{
  Assembly asm = Assembly.LoadFrom(assemblyFile);
  // Find and install command handlers from the assembly.
}
```

接下来，你须要把上面最后一行的注释替换成代码，这些代码要查找那些实现了命令句柄的类并且要安装这些句柄。加载完全程序集之后，你就可以使用反射来查找程序集上所有暴露出来的类型，使用特性来标识出哪些暴露出来的类型包含命令句柄，以及哪些是命令句柄的方法。下面是一个添加了特性的类，即标记了命令句柄类型：

```
// Define the Command Handler Custom Attribute:
[AttributeUsage(AttributeTargets.Class)]
public class CommandHandlerAttribute : Attribute
{
  public CommandHandlerAttribute()
  {
  }
}
```

这个特性就是你须要为每个命令标记的所有代码。总是用 AttributeUsage 特性标记一个特性类，这就是告诉其它程序以及编译器，在哪些地方这个特性可以使用。前面这个例子表示 CommandHandlerAttribute 只能在类上使用，它不能应用在其它语言的元素上。

你可以调用 GetCustomAttributes 来断定某个类是否具有 CommandHandlerAttribute 特性。只有具有该特性的类型才是插件的候选类型 ：

```
// Find all the assemblies in the Add-ins directory:
string AddInsDir = string.Format("{0}/Addins", Application.StartupPath);
string[] assemblies = Directory.GetFiles(AddInsDir, "*.dll");
foreach (string assemblyFile in assemblies)
{
  Assembly asm = Assembly.LoadFrom(assemblyFile);
  // Find and install command handlers from the assembly.
  foreach(System.Type t in asm.GetExportedTypes())
  {
    if (t.GetCustomAttributes(
      typeof(CommandHandlerAttribute), false).Length > 0)
    {
      // Found the command handler attribute on this type.
      // This type implements a command handler.
      // configure and add it.
    }
    // Else, not a command handler. Skip it.
  }
}
```

现在，让我们添加另一个新的特性来查找命令句柄。一个类型应该可以很简单的实现好几个命令句柄，所以你可以定义新的特性，让插件的作者可以把它添加到命令句柄上。这个特性会包含一参数，这些参数用于定义新的菜单命令应该放在什么地方。每一个事件句柄处理一个特殊的命令，而这个命令应该在菜单的某个特殊地方。为了标记一个命令句柄，你要定义一个特性，用于标记一个属性，让它成为一个命令句柄，并且申明菜单上的文字以及父菜单文字。DynamicCommand 特性要用两个参数来构造：菜单命令文字以及父菜单的文字。这个特性类还包含一个构造函数，这个构造函数用于为菜单初始化两个字符串。这些内容同样可以使用可读可写的属性：

```
[AttributeUsage(AttributeTargets.Property) ]
public class DynamicMenuAttribute：System.Attribute
{
  private string _menuText;
  private string _parentText;
  public DynamicMenuAttribute(string CommandText,
    string ParentText)
  {
    _menuText = CommandText;
    _parentText = ParentText;
  }
  public string MenuText
  {
    get { return _menuText; }
    set { _menuText = value; }
  }
  public string ParentText
  {
```

```
    get { return _parentText; }
    set { _parentText = value; }
  }
}
```

这个特性类已经做了标记，这样它只能被应用到属性上。而命令句柄必须在类中以属性暴露出来，用于提供给命令句柄来访问。使用这一技术，可以让程序在启动的时候查找和添加命令句柄的代码变得很简单。

现在你创建了这一类型的一个对象：查找命令句柄，以及添加它们到新的菜单项中。你可以把特性和反射组合起来使用，用于查找和使用命令句柄属性，对对象进行推测：

```
// Expanded from the first code sample:
// Find the types in the assembly
foreach(Type t in asm.GetExportedTypes())
{
  if (t.GetCustomAttributes(
    typeof(CommandHandlerAttribute), false).Length > 0)
  {
    // Found a command handler type:
    ConstructorInfo ci =
      t.GetConstructor(new Type[0]);
    if (ci == null) // No default ctor
      continue;
    object obj = ci.Invoke(null);
    PropertyInfo [] pi = t.GetProperties();
    // Find the properties that are command
    // handlers
    foreach(PropertyInfo p in pi)
    {
      string menuTxt = "";
      string parentTxt = "";
      object [] attrs = p.GetCustomAttributes(
        typeof (DynamicMenuAttribute), false);
      foreach (Attribute at in attrs)
      {
        DynamicMenuAttribute dym = at as
          DynamicMenuAttribute;
        if (dym != null)
        {
          // This is a command handler.
          menuTxt = dym.MenuText;
          parentTxt = dym.ParentText;
          MethodInfo mi = p.GetGetMethod();
          EventHandler h = mi.Invoke(obj, null)
            as EventHandler;
          UpdateMenu(parentTxt, menuTxt, h);
        }
```

```
      }
    }
  }
}
private void UpdateMenu(string parentTxt, string txt,
  EventHandler cmdHandler)
{
  MenuItem menuItemDynamic = new MenuItem();
  menuItemDynamic.Index = 0;
  menuItemDynamic.Text = txt;
  menuItemDynamic.Click += cmdHandler;
  //Find the parent menu item.
  foreach (MenuItem parent in mainMenu.MenuItems)
  {
    if (parent.Text == parentTxt)
    {
      parent.MenuItems.Add(menuItemDynamic);
      return;
    }
  }
  // Existing parent not found:
  MenuItem newDropDown = new MenuItem();
  newDropDown.Text = parentTxt;
  mainMenu.MenuItems.Add(newDropDown);
  newDropDown.MenuItems.Add(menuItemDynamic);
}
```

现在你将要创建一个命令句柄的示例。首先，你要用 CommandHandler 特性标记类型，正如你所看到的，我们习惯性的在附加特性到项目上时，在名字上省略 Attribute：

Now you'll build a sample command handler. First, you tag the type with the CommandHandler attribute. As you see here, it is customary to omit Attribute from the name when attaching an attribute to an item:

```
[ CommandHandler ]
public class CmdHandler
{
  // Implementation coming soon.
}
```

在 CmdHandler 类里面，你要添加一个属性来取回命令句柄。这个属性应该用 DynamicMenu 特性来标记：

```
[DynamicMenu("Test Command", "Parent Menu")]
public EventHandler CmdFunc
{
  get
  {
```

```
    if (theCmdHandler == null)
      theCmdHandler = new System.EventHandler
        (this.DynamicCommandHandler);
    return theCmdHandler;
  }
}
private void DynamicCommandHandler(
  object sender, EventArgs args)
{
  // Contents elided.
}
```

就是这了。这个例子演示了你应该如何使用特性来简化使用反射的程序设计习惯。你可以用一个特性来标记每个类型，让它提供一个动态的命令句柄。当你动态的载入这个程序集时，可以更简单的发现这个菜单命令句柄。通过应用 AttributeTargets (另一个特性)，你可以限制动态命令句柄应用在什么地方。这让从一个动态加载的程序集上查找类型的困难任务变得很简单：你确定从很大程度上减少了使用错误类型的可能。这还不是简单的代码，但比起不用特性，还算是不错的。

特性可以申明运行的意图。通过使用特性来标记一个元素，可以在运行时指示它的用处以及简化查找这个元素的工作。如何没有特性，你须要定义一些命名转化，用于在运行时来查找类型以及元素。任何命名转化都会是发生错误的起源。通过使用特性来标记你的意图，就把大量的责任从开发者身上移到了编译器身上。特性可以是只能放置在某一特定语言元素上的，特性同样也是可以加载语法和语义信息的。

你可以使用反射来创建动态的代码，这些代码可以在实际运行中进行配置。设计和实现特性类，可以强制开发者为申明一些类型，方法，以及属性，这些都是可以被动态使用的，而且减少潜在的运行时错误。也就是说，让你增加了创建让用户满足的应用程序的机会。

原则 43：请勿滥用反射

Don't Overuse Reflection

创建二进制的组件时，同时也意味着你要使用迟后绑定和反射来查找你所须要的具有特殊功能代码。反射是一个很有力的工具，而且它让你可以写出可动态配置的软件。使用反射，一个应用程序可以通过添加新的组件来更新功能，而这些组件是在软件最开始发布时没有的。这是有利的。

这一伸缩性也带来了一些复杂的问题，而且复杂问题的增加又会增加出现其它问题的可能。当你使用反射时，你是围绕着 C#的安全类型。然而，成员调用的参数和返回值是以 System.Object 类型存在的。你必须在运行时确保这些类型是正确的。简单的说，使用反射可以让创建动态的程序变得很容易，但同时也让程序出现错误变得很容易。通常，简单的思考一下，你就可以通过创建一系列接口集合来最小化或者移除反射，而这些接口集合应该表达你对类型的假设。

反射给了你创建类型实例的功能，以及在对象上调用成员方法，以及访问对象上的成员数据。这听上去就跟每天的编程任务是一样的。确实是这样的，对于反射，并没有什么新奇的：它就是动态创建其它的二进制组件。大多数情况下，你并不须要像反射这样的伸缩功能，因为有其它可选的更易维护的方案。

让我们从创建一个给定类型的实例开始，你可以经常使用一个类厂来完成同样的任务。考虑下面的代码，它通过使用反射，调用默认的构造函数创建了一个 MyType 的实例：

```
// Usage:Create a new object using reflection:
Type t = typeof(MyType);
MyType obj = NewInstance(t) as MyType;
```

```
// Example factory function, based on Reflection:
object NewInstance(Type t)
{
  // Find the default constructor:
  ConstructorInfo ci = t.GetConstructor(new Type[ 0 ]);
  if (ci != null)
    // Invoke default constructor, and return
    // the new object.
    return ci.Invoke(null);
  // If it failed, return null.
  return null;
}
```

代码通过反射检测了类型，而且调用了默认的构造函数来创建了一个对象。如果你须要在运行时创建一个预先不知道任何信息的类型实例，这是唯一的选择。这是一段脆弱的代码，它依懒于默认的构造函数的存在。而且在你移除了 MyType 类型的默认构造函数时仍然是可以通过编译的。你必须在运行时完成检测，而且捕获任何可能出现的异常。一个完成同样功能的类厂函数，在构造函数被移除时是不能通过编译的：

```
public MyType NewInstance()
{
  return new MyType();
}
```

(译注：其实 VS.Net 会给我们添加默认的构造函数，所以上面的两个方法都是可以编译，而且可以正确运行的。本人做过测试。但如果给构造函数添加访问限制，那么可以让类厂无法构造对象而产生编译时错误。)

你应该使用静态的类厂函数来取代依懒于反射的实例创建方法。如果你须要实例对象使用迟后数据绑定，那么应该使用类厂函数，而且使用相关的特性来标记它们(参见原则 42)。

另一个反射的潜在的用处就是访问类型的成员。你可以使用成员名和类型在运行时来调用实际的函数：

```
// Example usage:
Dispatcher.InvokeMethod(AnObject, "MyHelperFunc");
// Dispatcher Invoke Method:
public void InvokeMethod (object o, string name)
{
  // Find the member functions with that name.
  MemberInfo[] myMembers = o.GetType().GetMember(name);
  foreach(MethodInfo m in myMembers)
  {
    // Make sure the parameter list matches:
    if (m.GetParameters().Length == 0)
      // Invoke:
     m.Invoke(o, null);
  }
}
```

在上面的代码中，进行是错误被屏避。如果类型名字打错了，这个方法就找不到。就没有方法被调用。

这还只是一个简单的例子。要创建一个灵活的 InvokeMethod 版本，须要从 GetParameters()方法上返回的参数列表中，检测所有出现的参数类型。这样的代码是很沉长的，而且很糟糕以至于我根本就不想浪费地方来演示。

反射的第三个用处就是访问数据成员。代码和访问成员函数的很类似：

```
// Example usage:
```

```
object field = Dispatcher.RetrieveField (AnObject, "MyField");
// elsewhere in the dispatcher class:
public object RetrieveField (object o, string name)
{
  // Find the field.
  FieldInfo myField = o.GetType().GetField(name);
  if (myField != null)
    return myField.GetValue(o);
  else
    return null;
}
```

和方法调用一样，使用反射来取回一个数据成员，要在一个字段上通过名字来调用类型查询，看它是否与请求的字段名相匹配。如果发现一个，就可以使用 FieldInfo 结构来返回值。这个构造在.Net 框架里是很常见的。数据绑定就是利用反射来查找这些标记了绑定操作的属性。在这种情况下，数据绑定的动态性质超过了它的开销。(译注：也就是说值得使用反射进行动态绑定。)

因此，如果反射是一个如此痛苦的事情，你就须要找一个更好更简单的可选方案。你有三个选择：首先就是使用接口。你可以为任何你所期望的类，结构来定义接口(参见原则 19)。这可能会使用更清楚的代码来取代所有的反射代码：

```
IMyInterface foo = obj as IMyInterface;
if (foo != null)
{
  foo.DoWork();
  foo.Msg = "work is done.";
}
```

如果你用标记了特性的类厂函数来合并接口，几乎所有的你所期望于反射的解决方案都变得更简单：

```
public class MyType : IMyInterface
{
  [FactoryFunction]
  public static IMyInterface
    CreateInstance()
  {
    return new MyType();
  }
  #region IMyInterface
  public string Msg
  {
    get
    {
      return _msg;
    }
    set
    {
      _msg = value;
    }
```

```
  }
  public void DoWork()
  {
    // details elided.
  }
  #endregion
}
```

把这段代码与前面的基于反射的方案进行对比。即使这只是简单的例子，但还有在某些弱类型上使用所有的反射 API 时有精彩之处：返回类型已经是类型化的对象。而在反射上，如果你想取得正确的类型，你须要强制转换。这一操作可能失败，而且在继承上有危险。而在使用接口时，编译器提供的强类型检测显得更清楚而且更易维护。

反射应该只在某些调用目标不能清楚的用接口表示时才使用。.Net 的数据绑定是在类型的任何公共属性上可以工作，把它限制到定义的接口上可能会很大程度上限制它的使用。菜单句柄的例子充许任何函数(不管是实例的还是静态的)来实现命令句柄，使用一个接口同样会限制这些功能只能是实例方法。FxCop 和 NUnit (参见原则 48)都扩展了反射的使用，它们使用反射，是因为它们遇到的的现实的问题是最好用它来处理的。FxCopy 检测所有的代码来评估它们是否与已经的原则矛盾。这须要使用反射。NUnit 必须调用你编译的测试代码。它使用反射来断定哪些你已经写的代码要进行单元测试。对于你可能要写的测试代码，可能是一个方法集合，但接口是不能表达它们的。NUnit 使用特性来发现测试以及测试案例来让它的工作更简单(参见原则 42)。

当你可以使用接口策划出你所期望调用的方法和属性时，你就可以拥有一个更清楚，更容易维护的系统。反射是一个在数据以后绑定上功能强大的工具。.Net 框架使用它实现对 Windows 控件和 Web 控件的数据绑定。然而，很多常规情况下很少用，而是使用类厂，委托，以及接口来创建代码，这可以产生出更容易维护的系统。

原则 44：创建应用程序特定的异常类

Create Complete Application-Specific Exception Classes

异常是一种的报告错误的机制，它可以在远离错误发生的地方进行处理错误。所有关于错误发生的的信息必须包含在异常对象中。在错误发生的过程中，你可能想把底层的错误转化成详细的应用程序错误，而且不丢失关于错误的任何信息。你须要仔细考虑关于如何在 C#应用程序中创建特殊的异常类。第一步就是要理解什么时候以及为什么要创建新的异常类，以及如何构造继承的异常信息。当开发者使用你的库来写 catch 语句时，他们是基于特殊的进行时异常在区别为同的行为的。每一个不同的异常类可以有不同的处理要完成：

```
try {
  Foo();
  Bar();
} catch(MyFirstApplicationException e1)
{
  FixProblem(e1);
} catch(AnotherApplicationException e2)
{
  ReportErrorAndContinue(e2);
} catch(YetAnotherApplicationException e3)
{
  ReportErrorAndShutdown(e3);
} catch(Exception e)
{
  ReportGenericError(e);
```

```
}
finally
{
  CleanupResources();
}
```

不同的 catch 语句可以因为不同的运行时异常而存在。你，做为库的作者，当异常的 catch 语句要处理不同的事情时，必须创建或者使用不同的异常类。如果不这样，你的用户就只有唯一一个无聊的选择。在任何一个异常抛出时，你可以挂起或者中止应用程序。这当然是最少的工作，但样是不可能从用户那里赢得声望的。或者，他们 可以取得异常，然后试着断定这个错误是否可以修正：

```
try {
  Foo();
  Bar();
} catch(Exception e)
{
  switch(e.TargetSite.Name)
  {
    case "Foo":
      FixProblem(e);
      break;
    case "Bar":
      ReportErrorAndContinue(e);
      break;
    // some routine called by Foo or Bar:
    default:
      ReportErrorAndShutdown(e);
      break;
  }
} finally
{
  CleanupResources();
}
```

这远不及使用多个 catch 语句有吸引力，这是很脆弱的代码：如果只是常规的修改了名字，它就被破坏了。如果你移动了造成错误的函数调用，放到了一个共享的工具函数中，它也被破坏了。在更深一层的堆栈上发生异常，就会使这样的结构变得更脆弱。

在深入讨论这一话题前，让我先附带说明两个不能做承诺的事情。首先，异常并不能处理你所遇到的所有异常。这并不是一个稳固指导方法，但我喜欢为错误条件抛出异常，这些错误条件如果不立即处理或者报告，可能会在后期产生更严重的问题。例如，数据库里的数据完整性的错误，就应该生产一个异常。这个问题如果忽略就只会越发严重。而像在写入用户的窗口位置失败时，不太像是在后来会产生一系列的问题。返回一个错误代码来指示失败就足够了。

其次，写一个抛出(throw)语句并不意味会在这个时间创建一个新的异常类。我推荐创建更多的异常，而不是只有少数几个常规的自然异常：很从人好像在抛出异常时只对 System.Exception 情有独钟。可惜只只能提供最小的帮助信息来处理调用代码。相反，考虑创建一些必须的异常类，可以让调用代码明白是什么情况，而且提供了最好的机会来恢复它。

再说一遍：实际上要创建不同的异常类的原则，而且唯一原因是让你的用户在写 catch 语句来处理错误时更简单。查看分析这些错误条件，看哪些可以放一类里，成为一个可以恢复错误的行为，然后创建指定的异常类来处理

这些行为。你的应用程序可以从一个文件或者目录丢失的错误中恢复过来吗？它还可以从安全权限不足的情况下恢复吗？网络资源丢失又会怎样呢？对于这种遇到不同的错误，可能要采取不同的恢复机制时，你应该为不同的行为创建新的异常类。

因此，现在你应该创建你自己的异常类了。当你创建一个异常类时，你有很多责任要完成。你应该总是从 System.ApplicationException 类派生你的异常类，而不是 System.Exception 类。对于这个基类你不用添加太多的功能。对于不同的异常类，它已经具有可以在不同的 catch 语句中处理的能力了。

但也不要从异常类中删除任何东西。ApplicationException 类有四个不同的构造函数：

```
// Default constructor
public ApplicationException();
// Create with a message.
public ApplicationException(string);
// Create with a message and an inner exception.
public ApplicationException(string, Exception);
// Create from an input stream.
protected ApplicationException(
  SerializationInfo, StreamingContext);
```

当你创建一个新的异常类时，你应该创建这个四构造函数。不同的情况调用不同的构造方法来构造异常。你可以委托这个工作给基类来实现：

```
public class MyAssemblyException :
  ApplicationException
{
  public MyAssemblyException() :
    base()
  {
  }
  public MyAssemblyException(string s) :
    base(s)
  {
  }
  public MyAssemblyException(string s,
    Exception e) :
    base(s, e)
  {
  }
  protected MyAssemblyException(
    SerializationInfo info, StreamingContext cxt) :
    base(info, cxt)
  {
  }
}
```

构造函数须要的这个异常参数值得讨论一下。有时候，你所使用的类库之一会发生异常。调用你的库的代码可能会取得最小的关于可能修正行为的信息，当你简单从你使用的异常上传参数时：

```
public double DoSomeWork()
{
```

```
  // This might throw an exception defined
  // in the third party library:
  return ThirdPartyLibrary.ImportantRoutine();
}
```

当你创建异常时，你应该提供你自己的库信息。抛出你自己详细的异常，以及包含源异常做为它的内部异常属性。你可以提供你所能提供的最多的额外信息：

```
public double DoSomeWork()
{
  try {
    // This might throw an exception defined
    // in the third party library:
    return ThirdPartyLibrary.ImportantRoutine();
  } catch(Exception e)
    {
      string msg =
        string.Format("Problem with {0} using library",
          this.ToString());
      throw new DoingSomeWorkException(msg, e);
    }
  }
}
```

这个新的版本会在问题发生的地方创建更多的信息。当你已经创建了一个恰当的 ToString()方法时(参见原则 5)，你就已经创建了一个可以完整描述问题发生的异常对象。更多的，一个内联异常显示了产生问题的根源：也就是你所使用的第三方库里的一些信息。

这一技术叫做异常转化，转化一个底的层异常到更高级的异常，这样可以提供更多的关于错误的内容。你越是创建多的关于错误的额外的信息，就越是容易让它用于诊断，以及可能修正错误。通过创建你自己的异常类，你可能转化底层的问题到详细的异常，该异常包含所详细的应用程序信息，这可以帮助你诊断程序以及尽可能的修正问题。

希望你的应用程序不是经常抛出异常，但它会发生。如果你不做任何详细的处理，你的应用程序可能会产生默认的.Net 框架异常，而不管是什么错误在你调用的方法里发生。提供更详细的信息将会让你以及你的用户，在实际应用中诊断程序以及可能的修正错误大有帮助。当且仅当对于错误有不同的行为要处理时，你才应该创建不同的异常类。你可以通过提供所有基类支持的构造函数，来创建全功能的异常类。你还可以使用 InnerException 属性来承载底层错误条件的所有错误信息。

第六章 杂项

Miscellaneous

有些内容不合适专门做一个目录，但这并不是说它们不重要。对于每个人来说，理解代码的安全访问策略是很重要的，就像明白异常处理策略一样。其它的一些推荐资料是关于经常变化的一些东西，因为 C#本身也是一门在发展的语言，要与最新的标准和资讯进行交流。这些变化值得注意一下，而且要为这些变化做准备，它们会在以后溶入到你的工作中。

原则 45：选择强异常来保护程序

Prefer the Strong Exception Guarantee

当你抛出异常时，你就在应用程序中引入了一个中断事件。而且危机到程序的控制流程。使得期望的行为不能发生。更糟糕的是，你还要把清理工作留给最终写代码捕获了异常的程序员。而当一个异常发生时，如果你可以从你所管理的程序状态中直接捕获，那么你还可以采取一些有效的方法。谢天谢地，C#社区不须要创建自己的异常安全策略，C++社区里的人已经为我们完成了所有的艰巨的工作。以 Tom Cargill 的文章开头："异常处理：一种错误的安全感觉，" 而且 Herb Sutter，Scott Meyers，Matt Austern，Greg Colvin 和 Dave Abrahams 也在后继写到这些。C++社区里的大量已经成熟的实践可以应用在 C#应用程序中。关于异常处理的讨论，一直持续了 6 年，从 1994 年到 2000 年。他们讨论，争论，以及验证很多解决困难问题的方法。我们应该在 C#里利用所有这些艰难的工作。

Dave Abrahams 定义了三种安全异常来保证程序：基本保护，强保证，以及无抛出保证。Herb Sutter 在他的 Exceptional C++(Addison-Wesley, 2000)一书讨论了这些保证。基本保证状态是指没有资源泄漏，而且所有的对象在你的应用程序抛出异常后是可用的。强异常保证是创建在基本保证之上的，而且添加了一个条件，就是在异常抛出后，程序的状态不发生改变。无抛出保证状态是操作决对不发生失败，也就是从在某个操作后决不会发生异常。强异常保证在从异常中恢复和最简单异常之间最平衡的一个。基本保证一般在是.Net 和 C#里以默认形式发生。运行环境处理托管内存。只有在一种情况下，你可能会有资源泄漏，那就是在异常抛出时，你的程序占有一个实现了 IDisposable 接口的资源对象。原则 18 解释了如何在对面异常时避免资源泄漏。

强异常保证状态是指，如果一个操作因为某个异常中断，程序维持原状态不改变。不管操作是否完成，都不修改程序的状态，这里没有折衷。强异常保证的好处是，你可以在捕获异常后更简单的继续执行程序，当然也是在你遵守了强异常保证情况下。任何时候你捕获了一个异常，不管操作意图是否已经发生，它都不应该开始了，而且也不应该做任何修改。这个状态就像是你还没有开始这个操作行为一样。

很多我所推荐的方法，可以更简单的帮助你来确保进行强异常保证。你程序使用的数据元素应该存为一个恒定的类型(参见原则 6 和原则 7)。如果你组并这两个原则，对程序状态进行的任何修改都可以在任何可能引发异常的操作完成后简单的发生。常规的原则是让任何数据的修改都遵守下面的原则：

1、对可能要修改的数据进行被动式的拷贝。

2、在拷贝的数据上完成修改操作。这包括任何可能异常异常的操作。

3、把临时的拷贝数据与源数据进行交换。 这个操作决不能发生任何异常。

做为一个例子，下面的代码用被动的拷贝方式更新了一个雇员的标题和工资 ：

```
public void PhysicalMove(string title, decimal newPay)
{
  // Payroll data is a struct:
  // ctor will throw an exception if fields aren't valid.
  PayrollData d = new PayrollData(title, newPay,
    this.payrollData.DateOfHire);
  // if d was constructed properly, swap:
  this.payrollData = d;
}
```

有些时候，这种强保证只是效率很低而不被支持，而且有些时候，你不能支持不发生潜在 BUG 的强保证。开始的那个也是最简单的那个例子是一个循环构造。当上面的代码在一个循环里，而这个循环里有可能引发程序异常的修改，这时你就面临一个困难的选择：你要么对循环里的所有对象进行拷贝，或者降低异常保证，只对基本保证提供支持。这里没有固定的或者更好的规则，但在托管环境里拷贝堆上分配的对象，并不是像在本地环境上那开销昂贵。在.Net 里，大量的时间都花在了内存优化上。我喜欢选择支持强异常保证，即使这意味要拷贝一个大的容器：获得从错误中恢复的能力，比避免拷贝获得小的性能要划算得多。在特殊情况下，不要做无意义的拷贝。如果某个异常在任何情况下都要终止程序，这就没有意义做强异常保证了。我们更关心的是交换引用类型数据会让程序产生

错误。考虑这个例子：

```
private DataSet _data;
public IListSource MyCollection
{
  get
  {
    return _data;
  }
}
public void UpdateData()
{
  // make the defensive copy:
  DataSet tmp = _data.Clone() as DataSet;
  using (SqlConnection myConnection =
    new SqlConnection(connString))
  {
    myConnection.Open();
    SqlDataAdapter ad = new SqlDataAdapter(commandString,
      myConnection);
    // Store data in the copy
    ad.Fill(tmp);
    // it worked, make the swap:
    _data = tmp;
  }
}
```

这看上去很不错，使用了被动式的拷贝机制。你创建了一个 DataSet 的拷贝，然后你就从数据库里攫取数据来填充临时的 DataSet。最后，把临时存储交换回来。这看上去很好，如果在取回数据中发生了任何错误，你就相当于没有做任何修改。

这只有一个问题：它不工作。MyCollection 属性返回的是一个对_data 对象的引用(参见原则 23)。所有的类的使用客户，在你调用了 UpdateData 后，还是保持着原原来数据的引用。他们所看到的是旧数据的视图。交换的伎俩在引用类型上不工作，它只能在值类型上工作。因为这是一个常用的操作，对于 DataSets 有一个特殊的修改方法：使用 Merge 方法：

```
private DataSet _data;
public IListSource MyCollection
{
  get
  {
    return _data;
  }
}
public void UpdateData()
{
  // make the defensive copy:
  DataSet tmp = new DataSet();
  using (SqlConnection myConnection =
```

```
    new SqlConnection(connString))
  {
    myConnection.Open();
    SqlDataAdapter ad = new SqlDataAdapter(commandString,
      myConnection);
    ad.Fill(tmp);
    // it worked, merge:
    _data.Merge(tmp);
  }
}
```

合并修改到当前的 DataSet 上，就让所有的用户保持可用的引用，而且内部的 DataSet 内容已经更新。

在一般情况下，你不能修正像这样的引用类型交换，然后还想确保用户拥有当前的对象拷贝。交换工作只对值类型有效，如果你遵守原则 6，这应该是足够了。

最后，也是最严格的，就是无抛出保证。无抛出保证听起来很优美，就像是：一个方法是无抛出保证，如果它保证总是完成任务，而且不会在方法里发生任何异常。在大型程序中，对于所有的常规问题并不是实用的。然而，如果在一个小的范围上，方法必须强制无抛出保证。析构和处理方法就必须保证无异常抛出。在这两种情况下，抛出任何异常会引发更多的问题，还不如做其它的选择。在析构时，抛出异常中止程序就不能进一步的做清理工作了。

如果在处理方法中抛出异常，系统现在可能有两个异常在运行系统中。.Net 环境丢失前面的一个异常然后抛出一个新的异常。你不能在程序的任何地方捕获初始的异常，它被系统干掉了。这样，你又如何能从你看不见的错误中恢复呢？

最后一个要做无抛出保证的地方是在委托对象上。当一个委托目标抛出异常时，在这个多播委托上的其它目标就不能被调用了。对于这个问题的唯一方法就是确保你在委托目标上不抛出任何异常(译注：我不造成这种做法，而且谁又能保证在委托目标上不 出现异常呢？)。让我们再重申一下：委托目标(包括事件句柄)应该不抛出异常。这样做就意味关引发事件的代码不应该参与到强异常保证中。但在这里，我将要修改这一建议。原则 21 告诉你可以调用委托，因此你可以从一个异常中恢复。想也想得到，并不是每个人都这样做，所以你应该在委托句柄上抛出异常。只在要你在委托上不抛出异常，并不意味着其它人也遵守这一建议，在你自己的委托调用上，不能指望它是无抛出的保证。这主是被动式程序设计：你应该尽可能做原最好，因为其他程序可能做了他们能做的最坏的事。

异常列在应用程序的控制流程上引发了一系列的改变，在最糟糕的情况下，任何事情都有可能发生，或者任何事情也有可能不发生。在异常发生时，唯一可以知道哪些事情发生，哪些事情没有发生的方法就是强制强异常保证。当一个操作不管是完成还是没有完成时都不做任何修改。构造和 Dispose()以及委托目标是特殊的情况，而且它们应该在不充许任何异常逃出环境的情况下完成任务。最后一句话：小心对引用类型的交换，它可能会引发大量潜在的 BUG。

原则 46：最小化与其它非托管代码的交互

Minimize Interop

在开发设计.Net 时，MS 所做的最聪明的修改之一就是他们意识到，如果没有办法整合已经存在的代码到新的.Net 环境中，那没没有人会接受这个新的平台。MS 知道，如果没有办法来利用已经存在的代码，这将阻止大家接受它。与其它非托管代码的交互是可以工作了，但这是可交互唯一可以拿来说一下的有利的地方。对于所有的交互策略，当操作流程在本地代码和托管代码之间的边界上进行传送时，都要求强制提供一些 编组的信号。同时，交互策略强迫开发人员必须手动的申明所有调用参数(译注：这里是说你根本不知道参数的数据类型，很多时间你都只能以 int32 的方式传递所有参数，例如文件句柄，字符串指针等几乎是所有的参数，都只有一个 int32 也就是 IntPtr 类型进行处理，当然这里认为是是在 32 位机器上。)。最后，CLR 还不能完成对托管代码和本地代码的边界上进行数据传递时的优化。忽略采用本地代码或者 COM 对象时得到的好处吧，没有什么比这更好的了(译注：我本人强烈反对这一原则。C++，COM 在目前来说，绝对有它生存的优势，我觉得应该充分利用这些优势，而不应该忽略它

们)。但事实是交互并不是总能工作的，很多时候我们还是在要已经存在的应用程序中添加新的功能，提高而且更新已经存在的工具，或者在其它的地方要完成一个新的托管应用程序与旧的应用程序交互。使用一些交互在实际应用中只会是减缓对旧系统的更替。所以，有明白不同的交互策略之间有什么开销是很重要的。这些开销要同时花在开发计划以及运行时性能中。有些，最后的选择是重写旧代码。还有一些时候，你须要选择正确的交互策略。

在我讨论这个对你有用的交互策略之前，我须要花一段来讨论放弃(just throw it out)策略。第五章，与.Net 框架一起工作，向你展示了一些.Net 里已经为你创建好了的类和技术，你可以直接使用或者派生。为你你想的很多，你可以确定一些类和你一些代码算法，并且全部用 C#重写。剩下存在的代码可以被.Net 框架里已经存在的可能功能性的派生来取代。这并不会总是在任何地方，任何时候都可以工作的，但这确实是一个经过认真考虑过的迁移策略。整个第 5 章都推荐使用"throw it out"策略。这一原则就专注于交互，而它确实是件痛苦的事情。

接下来，让我们假设你已经决定重写全部代码并不实际。一些不同的策略要求你从.Net 中访问本地代码。你须要明白在本地代码和托管代码的边界上传递数据时的开销的低效。在使用交互时有三个开销。首先就是数据集群处理，这在托管堆和本地代码堆之间进行数据传递时发生。其次就是在托管代码和非托管代码进行交互时的大量数据吞吐时的开销。你以及你的用户要承担这些开销。第三个开销就只是你自己的了：你要在这个混合的开发环境中添加很多工作来实现交互。这也是最糟糕的一个，所以你的设计应该决定最小化这样的开销。

让我们开始讨论交互时在性能上的开销，以及如何最小化这些开销。数据集群是最大的一个因数，就像是网络服务或者远程操作一样，你须要尽可能使用笨重的(chunky)API 而不是小巧的(chatty)API(译注：数据集群是指你没有办法即时的与本地代码进行交互，而有一个延时，这个延时就使用数据堆集起来一起处理，这样就使得你应该尽可能少的与本地代码进行交互，而要选择一些一次可以处理较多数据的 API)。你可以用不同的方法来完成与非托管代码的交互。你可以重新修改已经存在的非托管代码来创建一个笨重的 API，更适合交互的 API。常规的 COM 应用中是申明很多属性，这样客户可以设置并修改 COM 对象内部的状态或者行为。每次的设置属性都会集群数据，而且不得不穿越边界。(而且每在穿越交互边界时也会有 thunks。)这非常的低效，不幸的是，COM 对象或者非托管库可能不受你控制。当这种情况发生时，你须要完成更麻烦的工作。这时，你可以创建一个轻量级的 C++库，通过使用你所须要的 chunkier API 来暴露类型的功能。这就要增加你的开发时间了(也就是第三个开销)。

当你封装一个 COM 对象时，确保你修改的数据类型已经在本地代码一托管代码之间提供了最好的数据集群策略。有些类型可以很好的比其它类型进行集群，试着限制用于在本地代码和托管代码之间进行传递的数据类型，尽量使用 blittable 数据。blittable 是指托管代码和本地代码都一样使用的类型。数据内容可以直接拷贝而不用管对象的内部的结构。某些情况下，非托管代码可能使用托管代码的代码。下面列出了 blittable 类型：

System.Byte
System.SByte
System.Int16
System.UInt16
System.Int32
System.UInt32
System.Int64
System.UInt64
System.UIntPtr

另外，任何的 blittable 类型的一维数组也是 blittable 类型。最后，任何格式化的包含 blittable 类型的也是 blittable 类型。一个格式化的类型可以是一个用 StructLayoutAttribute 明确定义了数据层次的结构，

```
[ StructLayout(LayoutKind.Sequential) ]
public struct Point3D
{
  public int X;
  public int Y;
  public int Z;
}
```

当你在托管代码和非托管代码之间，只使用 blittable 类型时，你就最小化了多少必须拷贝的信息呀!你同样也优化了任何必须发生的拷贝操作。

如果在数据拷贝时，你不能限制数据类型让它成为 blittable 类型，你可以使用 InAttribute 和 OutAttribute 来进行控制。也 COM 类似，这些特性控制数据拷贝的方法。In/Out 参数双向拷贝，In 参数以及 Out 参数是一次拷贝。确保你应用严格限制的 In/Out 组合，来避免更多不必须拷贝。

最后，你可以通过申明如何集群数据来提高性能。对于字符串来说这是最常见的。集群字符串默认是使用 BSTRs。这是一个安全的策略，但这也是最低效的。你可以通过修改默认的集群格式减少额外的拷贝操作，可以使用 MarshalAs 特性来修改集群方式。下面申明了字符串的集群使用 LPWStr 或者 wchar*:

```
public void SetMsg(
  [ MarshalAs(UnmanagedType.LPWStr) ] string msg);
```

这有一个关于处理托管和非托管层上数据的轶事：数据被拷贝然后在托管和非托管类型之间进行传输。你有三个方法业最小化拷贝操作。首先就是通过限制参数和返回值，让它们是 blittable 类型。这应该是首选的。当你不能这样做时，应用 In 和 Out 特性来最小化必须完成的拷贝和传输操作。最后一个优化就是，很多类型可以不只一种集群方式，你应该选择最优化的一种。

现在让我们转到如何在托管的非托管组件中转移程序控制。你有三种选择：COM 交互，平台调用(P/Invoke)，以及托管 C++。每种方法有它自己的优势和劣势。

COM 交互是最简单的方法来使用已经存在的 COM 组件。但 COM 交互也是在.Net 中和本地代码交互时最低效的一种方式。除非你的 COM 组件已经有很重要的利益，否则不要这样做。不要看也不要想这种方式。如果你没有 COM 组件而要使用这种方法就意味着你要把 COM 和交互原则学的一样好。没时间让你开始理解 IUnknown(译注：COM 原理中最基本的接口，任何 COM 都实现了这样的接口)。那些这样做的人都试着从我们的内存中尽快的清理它们。使用 COM 交互同样意味着在运行时你要为 COM 子系统承担开销。你同样还要考虑，在不同的 CLR 对象的生命期管理和 COM 版本的对象生命期管理之间又意味看什么。你可以服从 CLR 的原则，这就是说做一个你所导入的 COM 对象有一个析构函数，也就是在 COM 接口上调用的 Release()。或者你可以自己清楚的使用 ReleaseCOMObject()来释放 COM 对象。第一种方法会给应用程序引入运行时的低效(参见原则 15)。而第二个在你的程序里又是头疼的事。使用 ReleaseCOMObject ()就意味看你要深入到管理细节上，而这些 CLR 的 COM 交互已经帮我们完成了。你已经了解了，而且你认你明白最好的。CLR 请求所有不同，而且它要释放 COM 对象，除非你正确的告诉它，你已经完成了。这是一个极大的鬼计，因为 COM 专家程序员都是在每个接口上调用 Release()，而你的托管代码是以对象处理的。简单的说，你必须知道什么接口已经在对象上添加了 AddRef，而且只释放这些(译注：COM 的引用非常严格，每个引用都会添加一个 AddRef，释放时必须明确的给出 Release()，而且必须成对使用，而在.Net 对 COM 进行封装后，很多时候就是这个引用不匹配而出现资源泄漏)。就让 CLR 来管理 COM 的生命期，你来承担性能损失就行了。你是一个烦忙的开发人员，想在.Net 中混合 COM 资源到托管环境里，你要学习的太多了(也就是第三个开销)。

第二个选择是使用 P/Invoke。这是一个更高效的方法来调用 Win32 的 API，因为你避免了在上层与 COM 打交道。而坏消息是，你须要为你使用的每个 P/Invoke 方法手写这些接口。越是调用多的方法，越是多的申明必须手写。这种 P/Invoke 申明就是告诉 CRL 如何访问本地方法。这里额外的解释一下为什么每个一个关于 P/Invoke 的例子里(也包括下面这个)都使用 MessageBox:

```
public class PInvokeMsgBox
{
  [ DllImport("user32.dll") ]
  public static extern int MessageBoxA(
    int h, string m, string c, int type);
  public static int Main()
  {
```

```
        return MessageBoxA(0,
          "P/InvokeTest",
          "It is using Interop", 0);
      }
    }
```

另一个使用 P/Invoke 的主要缺点是，这并不是设计成面向对象的语言。如果你须要导入 C++库，你必须在你的导入申明中指明封装名。假设取代 Win32 的 MessageBox API，你想访问 MFC 的 C++DLL 里的另外两个 AfxMessageBox 方法。你须要为其中一个方法创建一个 P/Invoke 申明：

```
?AfxMessageBox@@YGHIII@Z
?AfxMessageBox@@YGHPBDII@Z
```

这两个申明名是与下面的两个方法匹配的：

```
int AfxMessageBox(LPCTSTR lpszText,
  UINT nType, UINT nIDHelp);
int AFXAPI AfxMessageBox(UINT nIDPrompt,
  UINT nType, UINT nIDHelp);
```

即使是在重写少数几个方法之后，你很快就明白这不是一个高产的方法来提供交互功能。简单的说，使用 P/Invoke 只能访问 C 风格的 Win32 方法(在开发上要开销更多的时间)。

最后一种选择就是在 Microsoft C++编译器上使用/CLR 开关来混合托管的非托管代码。如果你编译你所有的本地代码使用/CLR，你就创建了一个基于 MSIL 的库，该库使用本地堆来存储所有的数据。这就是说，这样的 C++库不能直接被 C#调用。你必须在你所熟悉的代码上创建一个托管的 C++库，用于在托管和非托管类型之间创建一个桥梁，提供在托管和非托管的堆之间的数据集群支持。这样的 C++库包含托管类，这些数据成员是基于托管堆的。这些类同样包含对本地对象的引用：

```
// Declare the managed class:
public __gc class ManagedWrapper : public IDisposable
{
private:
  NativeType* _pMyClass;
public:
  ManagedWrapper() :
    _pMyClass(new NativeType())
  {
  }
  // Dispose:
  virtual void Dispose()
  {
    delete _pMyClass;
    _pMyClass = NULL;
    GC::SuppressFinalize(this);
  }
  ~ManagedWrapper()
  {
    delete _pMyClass;
  }
```

```
  // example property:
  __property System::String* get_Name()
  {
    return _pMyClass->Name();
  }
  __property void set_Name(System::String* value)
  {
    char* tmp   = new char [ value->Length + 1 ];
    for (int i = 0 ; i < value->Length; i++)
      tmp[ i ] = (char)value->Chars[ i ];
    tmp[ i ] = 0;
    _pMyClass->Name(tmp);
    delete [] tmp;
  }
  // example method:
  void DoStuff()
  {
    _pMyClass->DoStuff();
  }
  // other methods elided...
}
```

再说一次，这并不是一个像以前使用过的高产的程序开发工具。这只是代码重复，而且整个目的都要在托管和非托管数据之间进行集群数据和 thunking。优势是，你可以从你的本地代码中暴露出来的方法和属性上完成控制。而劣势是你必须手写一部份.Net 代码以及一部份 C++代码。在这两种代码之间转换很容易让你发生错误。你还不能忘记删除非托管对象。托管对象不是你 要负责的。这会让你的开发进度降下来，因为要不断的检测这是否正确。

使用 /CLR 开关听上去像是一个神话，但对于所有的交互来说，这并不是一个神弹(magic bullet)。C++里的模板和异常处理与 C#是大不相同的。写的很好很高效的 C++并不一写能转化成最好的 MSIL 结构。更重要是，编译 C++代码时的 /CLR 开关并不做确认。正如我前面所说的，这样的代码是使用本地堆：它访问本地内存。而 CLR 并不能验证这些代码是安全的。调用这些代码的程序必须确保有安全许可来访问不安全代码。虽然如此，/CLR 策略还是最好的一个方法，在.Net 中利用已经存在的 C++代码(不是 COM 对象)。你的程序不会招致 thunking 开销，而为你的 C++库现在并不在 MSIL 中，MSIL 不是本地的 CPU 指令。

交互操作是件头疼的事。在你使用交互之间，认真的考虑写本地应用程序。这经常是简单而且很快的。不幸的是，对于很多的开发人员来说，交互是必须的。如果你有已经存在的用其它语言写的 COM 对象，使用 COM 交互。如果你有已经存在的 C++代码，使用 /CLR 开关并托管 C++来提供最好的策略来方法已经存在的本地代码。选择最省时间的一个策略，这可能就是"just thro it out"策略。

原则 47：选择安全的代码

Prefer Safe Code

.Net 运行时已经设计好了，一些怀有恶意的代码不能渗透到远程计算机上并执行。目前一些分部式系统依懒于从远程机器上下载和执行代码。如果你可以通过 Internet 或者以太网来发布你的软件，或者直接从 web 上运行，但你须要明白 CRL 在你的程序集中的一些限制。如果 CLR 不是完全相信一个程序集，它会限制一些的行为。这些调用代码要有访问安全认证(CAS)。从另一方面来说，CLR 强制要求基于角色的安全认证，这样这些代码才能或者不能在基于一个特殊的角色帐号下运行。

安全违例是运行时条件，编译器不能强制它们。幸运的是，它们绝不会在你的开发机器上出现，而且你所编译的代码从你自己的硬件上加载，这就是说，它有更高的信任级别。讨论所有潜在的.Net 安全模型可以足足写上几本书，但你可以了解合理行为的一小部份，这样可以让你的程序集与.Net 的安全模式更容易的交互。这些推荐只有在你创建一个组件程序库时，或者是开发一些通过网络发布的组件和程序集是才可以参考应用。

通过这个讨论，你应该记住.Net 是一个托管的环境。这个环境保证有一个明确的安全环境。在安装时可以用.Net 的配置策略来管理安全策略。大多数.Net 框架库是在安装时对配置策略是安全信任的。它会查明安全问题，这就是说 CLR 可以检测 IL 而且确保它不会有什么潜在的危险行为，例如直接访问原始内存。它不会在访问本地资源时要求特殊的安全权限进行断言。你应该试着遵守同样的检测，如果你的代码不须要任何的安全权限，就应该避免使用 CAS 的 API 来对断定访问权限，否则你所做的只是降低程序性能。

你要使用 CAS 的 API 来访问一些受保护的资源，而这些资源是要求增加的特权的。很多通用的受保护资源是非托管的内存和文件系统。其它一些受保护的资源还包括数据库，网络端口，windows 注册表，以及打印子系统。在每种情况下，如果调用代码没有足够的许可，试着访问这些资源都会引发一个异常。而且，访问这些资源可能引发运行时建立一个安全栈上的询访，以确保当前栈上的所有的程序集有恰当的许可。让我们看一下内存以及文件系统，讨论安全系统和机密问题中最实际的一些问题。

不管什么时候，你都可以通过创建恰当的安全程序集来避免非托管内存访问。一个安全的程序集，也就是一个不用使用任何指针来访问其它非托管，或者托管的堆内存。不管你是否知道，你所创建的所有 C#代码几乎都是安全的。除非你在 C#编译器上打开了不安全的编译开关/unsafe，否则你所创建的都是安全代码(译注：就算打开了开关也不是说就编译成不安全代码了，还要看你的代码是怎样写的。)。/unsafe 充许用户使用 CLR 不进行的验证的指针。

要使用不安全代码的原因很少，特别是一常规的任务中。指向原始内存的指针比要检测的安全的引用快一些。在一些经典的数组中，它们可能要快上 10 倍以上。但当你使用不安全结构时，要明白任何的不安全代码都会影响整个程序集。当你创建不安全块时，应该考虑把这些算法独立到一个程序信中(参见原则 32)。这样可以在整个程序上限制不安全代码的影响。如果它是独立的，只有实际调用它的访问者才会受到影响。其它剩下的，你还是可以在更严格的环境中使用安全机制。你可能还须要不安全代码来处理一些须要直接指针的 P/Invoke 或者 COM 接口。同样的推荐：独立它。不安全代码只会影响它自己的小程序集，不再有其它的。

对于访问的建议很简单：只要可能，都应该避免访问非托管内存。

接下来的安全核心就是文件系统。程序要存储数据。从 Internet 上下载回来的代码，文件系统中的大多数地方都不能访问，否则会有很大的安全漏洞。是的，完全不许访问文件系统就很难创建能使用的程序。通过使用独立存储可以解决这一问题。独立存储可以穿越基于程序集而独立的虚拟的目录，以及应用程序域，以及当前的用户。选择性的，你可以使用更一般的独立存储虚拟目录，该目录是基于程序集或者当前用户的。

实际上，受信任的程序集可以访问他们自己特殊的独立存储区域，但不能是文件系统的其它地方。独立的存储目录是隐藏在其它程序集以及其它用户中的。你可以使用 System.IO.IsolatedStorage 名字空间中的类来访问独立的存储。IsolatedStorageFile 类包含的方法可以很简单的访问 System.IO.File 类。实际上，它是从 System.IO.FileStream 类派生下来的。写内容到独立存储的代码几乎与写内容到任何文件里是一样的：

```
IsolatedStorageFile iso =
  IsolatedStorageFile.GetUserStoreForDomain();
IsolatedStorageFileStream myStream = new
  IsolatedStorageFileStream("SavedStuff.txt",
  FileMode.Create, iso);
StreamWriter wr = new StreamWriter(myStream);
// several wr.Write statements elided
wr.Close();
```

读操作也是完全和其它使用文件 I/O 相似的：

```
IsolatedStorageFile isoStore =
  IsolatedStorageFile.GetUserStoreForDomain();
string[] files = isoStore.GetFileNames("SavedStuff.txt");
if (files.Length > 0)
{
  StreamReader reader = new StreamReader(new
    IsolatedStorageFileStream("SavedStuff.txt",
    FileMode.Open,isoStore));
  // Several reader.ReadLines() calls elided.
  reader.Close();
}
```

你可以独立存储来持久大小合适的数据元素，这些元素可以被代码部分信任，用于从一个安全分离的本地磁盘上的某个地方存储和载入信息。.Net 环境为每个程序定义和限制了独立存储的大小。这可以预防一些恶意的代码占用磁盘空间，让系统就得不可用。独立存储对于其它程序和其它用户来说是不可见的。也就是说，它不应用于要管理员手动操作才能布署或者配置设置的情况。即使这它是隐藏的，然而，独立存储对于从受信任的用户那来里来的非托管代码来说也是不受保护的。不要用独立存储来存储一些高度机密的内容，除非你的程序给它加过密。

在文件系统中创建一个可能要许可安全策略的程序集时，要独立存储流的内容。当你的程序集可能在 web 上运行，或者可能被运行在 web 上的代码访问时，应该考虑使用独立存储。

你可能须要正确的使用一个受保护的资源。一般情况下，访问这些资源要指出你的程序要被完全信任。唯一可选的就是完全避免使用这些受保护的资源。例如，考虑 windows 的注册表，如果你和程序须要访问注册表，你必须安装你的程序到最终用户的机器上，这样才能有必须的权限来访问注册表。你想简单点，从 web 上运行的程序是不能建立注册表的修改的。安全策略就应该是这样的。

.Net 的安全模型意味着你的程序的行为是要权得进行核对的。注意你的程序所要求的权利，而且试着最小化它们。不必要求你不使用的权利。你的程序集越是少的要求受保护资源，那么它们就越是可以保证安全策略异常不抛出。避免使用机密资源，如果可能，要考虑其它可选方案。当你在某个算法上确实须要更高安全的许可时，应该独立这些代码到它们自己的程序集中。

原则 48：了解更多的工具和资源

Learn About Tools and Resources

对于 C#以及.Net 来说这是激动人心的时候。这些工具目前还是比较新的，整个社区都在学习如何使用这些工具。一些资源可以帮助你提高你的知识，以及为.Net 和 C#创建一个更大的知识社区。这些工具是我每天都向 C#开发人员推荐的。关于 C#实践的全部内容还在写作当中，跟进它们而且不断了解相关的内容。

第一个应该在每一个 C#开发人员的工具箱的工具是 NUnit，它可以在 www.nunit.org 网站上找到。NUnit 是一个自动进行单元测试的工具，功能和 JUnit 很像。和其它大多数开发人员一样，我讨厌写测试代码并且自己测试。NUnit 让这些进程都变得很高效，在你有规律的使用这些工具后，可以保证你会习惯测试你所有的 C#类。不管什么时候当我创建了一个类库工程时，我都会添加一个 NUnit 测试工程，而且把自动生成的测试做为一部分添加进来。我添加一个以创建和运行测试的配置，这样可以在每次编译时进行测试。然后，我可以转换活动的配置来控制是否要让单元测试做为正规程序的一部份存在。默认情况下，我运行它们。当我须要进行 UI 测试时，我会转换到另一个配置上。

在附带的使用 NUnit 时，你可以通过检测 NUnit 的源代码学到一些有意思的技术。NUnit 使用一些高级的反射习惯来加载和测试你的程序集。它使用特性来查找测试包，测试用例，以及每个测试用例的期望结果(参见原则 42)。这是一个非常不错的例子，可以告诉你如何使用这些技术来创建可以自己动态配置的工具，而且它可以广泛的应用。

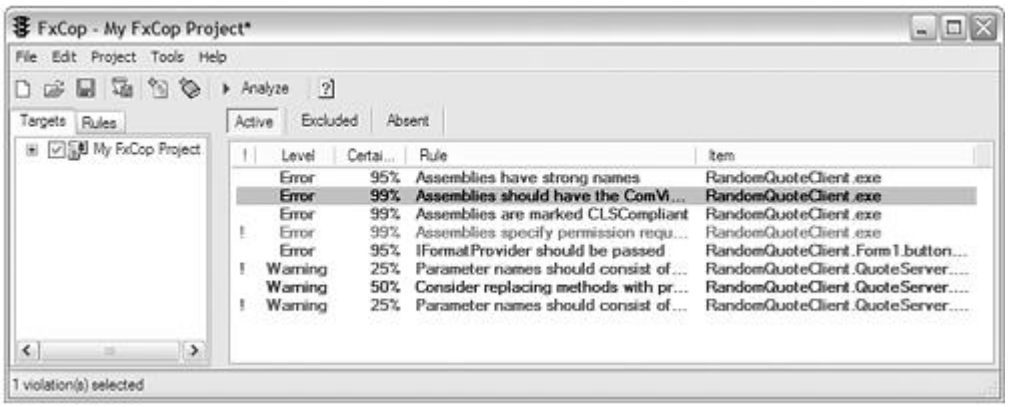
接来下是 FXCop，这是一个免费的工具，可以从 GotDotNet(www.gotdotnet.com)上得到。FXCop 分析你的程序集里的 IL，看它是否与实践的原则相违背，以及报告这些违例的地方。每一个原则都有一个可靠的公制规

范，以及使用这一原则的原因。如本书里所有推荐的原则一样，一些文档中有一个关于某一原则的简短理由。你可以断定这些实际的问题是否遵守这些建议。同样你也可以配置是否把每一个原则都应用到项目中。我并不赞成 FXCop 中的一些原则，而且我在本书前面已经说明了原因。然而，像 NUnit 一样，FXCop 可以成为你创建的正规程序的一部分。每次编译后，可以有一个编译后步骤，可以用 FXCop 来分析你选择的原则。图 6.1 展示了一个从 FXCop 里输出的例子。尽管一些推荐并不是我喜欢的(例如有一个是让每一个程序集应该是让 COM 见的)，但它确实是一个有用的工具，因为它让你思考很多你已经默认的决定。

图 6.1,FXCop 分析的一个项目：

Figure 6.1. FXCop analyzing a project.

[View full size image]



ILDasm 是一个 IL 反汇编器，在本书不同的地方，我已经演示了一些 IL 代码，它们就是编译器为不同的 C#结构生成的。尽管我不相信很多人会在有高级语言存在的情况下，还选择写 IL 代码，但你应该熟悉它。知道从不同的 C#结构上生成的 IL 代码，可以帮助你成为一个更好的开发者。你可以为你自己的程序集检测 IL 代码，或者是.Net 框架里的程序集。这就是使用 ILDAsm，而且它是和.Net 框架的 SDK 一起发布的。IL 对于所有开发者来说都是可用的。ILDAsm 可以让你看到你的程序集的中间语言。不管怎样，这是一个好的方法来学习.Net 框架程序集，这也是得到的原始资料。

这些只是你正式工具箱中的一部份，但拥有这些工具只是提高你技能的一个方面。大量在线的资源以及交流社区可以让你参与和学习，以及增加你自己的 C#和.net 框架知识。首先也是最重要的就是 GotDotNet 网站 (www.gotdotnet.com)，这是.Net 组的官方网站。C#小组在 MSDN 上有一个站点，目前是在 msdn.microsoft.com/vcsharp/ (它偶然在 MSDN 网站有变动而被重新组织)。如果你的工作主要是基于网络的，试着访问 www.asp.net，这是为 ASP.Net 组提供的。如果你的工作主要是基于 Windows Form 的，试着看看 www.windowsforms.net，这是 Windows Form 组的官方网站。这些网站包含很多常规编程的引用和实现，这些可能是你的应用程序中想要的。它们都是还源文件的组件，所以你可以检测和修改这些，让它们成为你想要的。最后也是最重要的位置应该要了解就是在 MS 模式和实践的网页。这个网页目前在 www.microsoft.com/resources/practices/，从这个地方，你可以查到一些常用的设计模式以及一些最好的模式的初始代码。

而且这个地方经常更新一些新的例子的代码以及库，这可能帮助你解决常规的编程问题。在写这些时，你已经可以使用 10 个不同的应用程序块来实现一些常规的程序要求，我确信当你阅读到这些时，这些地方已经有更多的内容了。

我还要推荐一些 C#组的 FAQ 的订阅：http://blogs.msdn.com/csharpfaq，附带的在这个上，有几个 C#组的成员用博客讨论一些 C#问题。你可以在这里找到最新的列表：
http://msdn.microsoft.com/vcsharp/team/blogs/

如果你学习更多的而且对语言和环境想得到更深入的了解，你可以检测共享的 CLI(code-named rotor)。这包含.net 框架以及 C#编译器的一些核心内容。你可阅读这些资料来对 C#语言的每一个功能和.Net 框架得到更深

入的理解。并不是所有的.net 商业框架都有可用的共享资料：例如，特殊的 Windows 代码并没有发布共享代码。然而，这些已经发布了的子集，同样够你学习更多的关于 CLR 和 C#语言内部的东西。

C#编译器已经和共享的 CLI 资料一起发布，这是用 C++写的，它做为底层的 CLR 代码存在。你须要对 C++有很深的背景知识，以及对编译器设计有清楚的认识才能很好的理解它。现代的语言编译器是复杂的软件块，但 CLR 资料是一个有用的工具，来理解.Net 框架的核心功能是如何实现的。

这里只是给出了一个简单的列表， 我只是在众多资料中介绍了一个表面。很多资料你都可以多 MS 上得到，或者其它在线网站，或者是书。你越是多的使用这些工具，你就可以得到越多的知识。整个.Net 以及 C#社区是在前进的，因为它发展的很快，这些列出的资源也可能不断的在改变。你可以自己学习和自己写稿。

原则 49：为 C#2.0 做好准备

Prepare for C# 2.0

C#2.0，在 2005 年已经可以使用了，它有一些主要的新功能。这样使得目前使用的一些最好的实际经验可能会有所改变，这也会随着下一代工具的发布而修改。尽管目前你还可以不使用这些功能，但你应该这些做些准备。

当 Visual Studio .net2005 发布后，会得到一个新的开发环境，升级的 C#语言。附加到这门语言上的内容确实让你成为更有工作效率的开发者：你将可以写更好重用的代码，以及用几行就可以写出更高级的结构。总而言之，你可以更快的完成你的工作。

C#2.0 有四个大的新功能：范型，迭代，匿名方法，以及部分类型。这些新功能的主要目的就是增强你，做为一个 C#开发的开发效率。这一原则会讨论其中的三个新功能，以及为什么你要为些做准备。与其它新功能相比，范型在对你如何开发软件上有更大的影响。范型并不是 C#的特殊产物。为了实现 C#的范型，MS 已经扩展了 CLR 以及 MS 的中间语言(MSIL)。C#，托管 C++，以及 VB.Net 都将可以使用范型。J#也将可以使用这些功能。

范型提供了一种"参数的多太"，对于你要利用同一代码来创建一系列相似的类来说，这是一个很神奇的方法。当你为范型参数供一个特殊的类型，编译器就可以生成不同版本的类。你可以使用范型来创建算法，这些算法与参数化的结构相关的，它们在这些结构上实行算法。你可以在.net 的 Collection 名字空间中找到很多候选的范型：HashTables, ArrayList，Queu，以及 Stack 都可以存储不同的对象而不用管它们是如何实现的。这些集合对于 2.0 来说都是很好的范型候选类型，这在 System.Collections.Generic 存在范型，而且这些对于目前的类来说都一个副本。C#1.0 是存储一个对 System.Obejct 类型的引用，尽管当前的设计对于这一类型来说是可重用的，但它有很多不足的地方，而且它并不是一个类型安全的。考虑这些代码：

```
ArrayList myIntList = new ArrayList();
myIntList.Add(32);
myIntList.Add(98.6);
myIntList.Add("Bill Wagner");
```

这编译是没问题的，但这根本无法表明你的意思。你是真的想设计这样一个容器，用于存储总完全不同的元素吗？或者你是想在一个受到限制的语言上工作吗？这样的实践意味着当你移除集合里的元素时，你必须添加额外的代码来决定什么样的对象事先已经存在这样的集合中。不管什么情况，你须要从 System.Object 强制转化这些元素到实际的你要的类型。

这还不只，当你把它们放到 1.0 版(译注：是 1.0，不是 1.1)的集合中时，值类型的开销更特殊。任何时候，当你放到个值类型数据到集合中时，你必须对它进行装箱。而当你在从集合中删除它时，你又会再开销一次。这些损失虽然小，但对于一个有几千元素的大集合来说，这些开销就很快的累积起来了。通过为每种不同的值类型生成特殊的代码，范型已经消除了这些损失。

如果你熟悉 C++的模板，那么对于 C#的范型就不存在什么问题了，因为这些从语法上讲是非常相似的。范型的内部的工作，不管它是怎产的，却是完全不同的。让我们看一些简单的例子来了解东西是如何工作的，以及它是如何实现的。考虑下面某个类的部份代码：

```
public class List
{
```

```
  internal class Node
  {
    internal object val;
    internal Node next;
  }
  private Node first;

 public void AddHead(object t)
  {
    // ...
  }
  public object Head()
  {
    return first.val;
  }
}
```

这些代码在集合中存储 System.Object 的引用，任何时候你都可以使用它，在你访问集合是，你必须添加强制转换。但使用 C#范型，你可以这样定义同样的类：

```
public class List < ItemType >
{
  private class Node < ItemType >
  {
    internal ItemType val;
    internal Node < ItemType > next;
  }
  private Node < ItemType > first;
  public void AddHead(ItemType t)
  {
    // ...
  }
  public ItemType Head()
  {
    return first.val;
  }
}
```

你可以用对象来代替 ItemType, 这个参数类型是用于定义类的。C#编译器在实例化列表时，用恰当的类型来替换它们。例如，看一下这样的代码：

```
List < int > intList = new List < int >();
```

MSIL 可以精确的确保 intList 中存储的是且只是整数。比起目前你所实现的集合(译注：这里指 C#1.1 里的集合)，创建的范型有几个好处，首先就是，如果你试图把其它任何不是整型的内容放到集合中时，C#的编译器会给出一个编译错误，而现今，你须要通过测试运行时代码来附加这些错误。

在 C#1.0 里，你要承担装箱和拆箱的一些损失，而不管你是从集合中移出或者是移入一个值类型数据，因为它们都是以 System.Object 的引用形式存在的。使用范型，JIT 编译器会为集合创建特殊的实例，用于存储实际的值类型。这样，你就不用装箱或者拆箱了。还不只这些，C#的设计者还想避免代码的膨胀，这在 C++模板里是相关的。为了节约空间，JIT 编译器只为所有的引用类型生成一个版本。这样可以取得一个速度和空间上的平衡，对每

个值类型(避免装箱)会有一个特殊的版本呢，而且引用类型共享单个运行时的版本用于存储 System.Object (避免代码膨胀)。在这些集合中使用了错误的引用时，编译器还是会报告错误。

为了实现范型，CLR 以及 MSIL 语言经历了一些修改。当你编译一个范型类时，MSIL 为每一个参数化的类型预留了空间。考虑下面两个方法的申明 MSIL：

To implement generics, the CLR and the MSIL language undergo some changes. When you compile a generic class, MSIL contains placeholders for each parameterized type. Consider these two method declarations in MSIL:

```
.method public AddHead (!0 t) {
 }
.method public !0 Head () {
}
```

!0 就是一个为一个类型预留的，当一个实际的实例被申明和创建时，这个类型才创建。这就有一种替换的可能：

```
.method public AddHead (System.Int32 t) {
 }
.method public System.Int32 Head () {
}
```

类似的，变化的实例包含特殊的类。前面的为整型的申明就变成了为样：

```
.locals (class List<int>)
newobj void List<int>::.ctor ()
```

这展示了 C#编译器以及 JIT 编译是如何为一个范型而共同工作的。C#编译器生成的 MSIL 代码为每一个类型预留了一个空间，JIT 编译器在则把这些预留的类型转换成特殊的类型，要么是为所有的引用类型用 System.Object，或者对值类型言是特殊的值类型。每一个范型的变量实例化后会带有类型信息，所以 C#编译器可以强制使用类型安全检测。

范型的限制定义可能会对你如何使用范型有很大的影响。记住，在 CLR 还没有加载和创建这些进行时实例时，用于范型运行时的特殊实例是还没有创建的。为了让 MISL 可以让所有的范型实例都成为可能，编译器须要知道在你的范型类中使用的参数化类型的功能。C#是强制解决这一问题的。在参数化的类型上强制申明期望的功能。考虑一个二叉树的范型的实现。二叉树以有序方式存储对象，也就是说，二叉树可以只存储实现了 IComparable 的类型。你可以使用约束来实现这一要求：

```
public class BinaryTree < ValType > where
  ValType : IComparable < ValType >
{
}
```

使用这一定义，使用 BinaryTree 的实例，如何使用了一个没有实现 IComparable 接口的类型时是不能通过编译的。你可以指明多个约束。假设你想限制你的 BinaryTree 成为一个支持 ISerializable 的对象。你只用简单的添加更多的限制就行了。注意这些接口以及限制可以在范型上很好的使用：

```
public class BinaryTree < ValType > where
      ValType : IComparable < ValType > ,
  ValType : ISerializable
{
}
```

你可以为每个个实例化的类型指明一个基类以及任何数量的接口集合。另外，你可以指明一个类必须有一个无参数的构造函数。

限制同样可以提供一些更好的好处：编译器可以假设这些在你的范型类中的对象支持指定列表中的某些特殊的

接口(或者是基类方法)。如何不使用任何限制时，编译器则只假设类型满员 System.Object 中定义的方法。你可能须要添加强制转换来使用其它的方法，不管什么时候你使用一个不在 System.Object 对象里的方法时，你应该在限制集合是写下这些需求。

约束指出了另一个要尽量使用接口的原因(参见原则 19)：如果你用接口来定义你的方法，它会让定义约束变得很简单。

迭代也是一个新的语法，通常习惯上用于少代码。想像你创建一些特殊的新容器类。为了支持你的用户，你须要在集合上创建一些方法来支持逆转这些集合以及运行时对象。

目前，你可能通过创建一个实现 IEnumerator 了的类来完成这些。IEnumerator 包含两个方法，Reset 和 MoveNextand，以及一个属性：Current。另外，你须要添加 IEnumerable 来列出集合上所有实现了的接口，以及它的 GetEnumerator 方法为你的集合返回一个 IEnumerator。在你写写完了以后，你已经写了一个类以及至少三个额外的函数，同时在你的主类里还有一些状态管理和其它方法。为了演示这些，目前你须要写这样一页的代码，来处理列表的枚举：

```
public class List : IEnumerable
{
  internal class ListEnumerator : IEnumerator
  {
    List theList;
    int pos = -1;
    internal ListEnumerator(List l)
    {
      theList = l;
    }
    public object Current
    {
      get
      {
        return theList [ pos ];
      }
    }
    public bool MoveNext()
    {
      pos++;
      return pos < theList.Length;
    }
    public void Reset()
    {
      pos = -1;
    }
  }
  public IEnumerator GetEnumerator()
  {
    return new ListEnumerator(this);
  }
  // Other methods removed.
```

```
}
```

在这一方面上，C#2.0 用 yield 关键字添加了新的语法，这让在写这些迭代时变得更清楚。对于前面的代码，在 C#2.0 里可是样的：

```
public class List
{
  public object iterate()
  {
    int i=0;
    while (i < theList.Length ())
      yield theList [ i++ ];
  }
  // Other methods removed.
}
```

yield 语句让你只用 6 行代码足足替换了近 30 行代码。这就是说，BUG 少了，开发时间也少了，以及少的代码维护也是件好事。

在内部，编译器生成的 MSIL 与目前这 30 行代码是一致的。编译器为你做了这些，所以你不用做 。编译器生成的类实现了 IEnumerator 接口，而且添加了你要支持的接口到列表上。

最后一个新功能就是部分类型。部分类型让你要吧把一个 C#类的实现分开到多个文件中。你可能很少这样做，如果有，你自己可以在日常的开发中，使用这一功能来创建多源文件。MS 假设这一修改是让 C#支持 IDE 以及代码生成器。目前，你可以在你的类中使用 region 来包含所以 VS.net 为你生成的代码。而将来(译注：指 C#2.0)，这些工具可以创建部份类而且取代这些代码到分开的文件中。

使用这一功能，你要为你的类的申明添加一个 partial 关键字：

```
public partial class Form1
{
  // Wizard Code:
  private void InitializeComponent()
  {
    // wizard code...
  }
}
// In another file:
public partial class Form1
{
  public void Method ()
  {
    // etc...
  }
}
```

部分类型有一些限制。类只与源相关的，不管是一个文件还是多个源文件，它们所生成的 MSIL 代码没有什么两样。你还是要编译一个完整类的所有的文件到同样的程序集中，而且没有什么自动的方法来确保你已经添加了一个完整类的所有源文件到你的编译项目中。当你把一个类的定义从一文件分开到多个文件时，你可能会以引发很多问题，所以建议你只用 IDE 生成部分类型功能。这包含 form，正如我前面介绍的那样。VS.Net 同样为 DataSet(参见原则 41)也生成部分类型，还有 web 服务代理，所以你可以添加你自己的成员到这些类中。

我没有太多的谈到关于 C#2.0 的功能，因为添加的与目前的编码有一些冲突。你可以使用它，通过范型让你自己的类型变得简单，而定义接口可以描述行为：这些接口可以做为约束。新的迭代语法可以提供更高效的方法来实现枚举。你可以通过这一新语法，快速简单的取代嵌套枚举。然而，用户扩展类可能不会是简单的取代。现在开发你自己的代码，在显而易见的地方利用这些功能，而且在用 C#2.0 升级你已经存在的代码时，它会变得更容易，工作量也会变得最少。

原则 50：了解 ECMA 标准

Learn About the ECMA Standard

ECMA 标准是 C#语言所有功能的官方说明。ECMA-334 定义了 C#语言 1.0 的标准，你可以从 The C# Programming Language 这本书上学习 C#2.0 的计划(译注：现在已经不是计划了)，这本书的作者是 Anders Hejlsberg, Scott Wiltamuth, 和 Peter Golde (Addison-Wesley, 2003)。这本书是一个语言手册，而不是指南。它详细说明了这门语言书面定义的每一个功能。每一种语言都只一种标记，可以让你更加明白每一种语言的功能。当我还在写这本书的时候，我还经常把这书放在我的桌子上参考。

如果你认真的 C#程序员，你应该明白这门语言，包括在不同功能后面的基本原理。如果在你工作的时候，你知道在什么时候应用每一个功能，它就会让你的工作更容易。你可以更好的理解隐藏在不同语言表达式后的每一个不同细节。

对于 C#的附带内容，你应该彻底的明白公共运行时(CLR)。CLR 以及公共语言基础(CLI)标准在 ECMA-335 中有定义,这也是 CLR 标准。做为 C#,这还是 1.0 的标准。公共语言基础标记标准这一书(The Common Language Infrastructure Annotated Standard)，由 ames Miller 和 Susann Ragsdale (Addison-Wesley, 2003)所著，解说了 CLI 的 2.0 版本。这是一本包括公共语言子系统(CLS)的使用手册，这会帮助理解 CLS 遵从性(译注：前几天才看到 MSDN 上把 compliance 翻译为遵从性，而我一直理解为兼容性，当然兼容性有其它的词，很多时候是理解的问题。)的背后原则。这同样可以帮助你明白.Net 运行时和基础的 ECMA 标准。

C#和 CLR 委员会还在发布工作文档，用于讨论的发展 C#语言的 2.0 版本以及 CLR。这些讨论对于明白 C#今后会如何发展以及改变是很有价值的。

另处，深刻理解当前的标准以及增加的意图可以帮助你创建经得起时间考虑的代码。通过理解这些可能添加到语言以及进行环境上的功能，你可以在创建软件时立于一个有利的位置上，而且可以持续更久。你可以预料到一些可能因某些必然因素而做出的修改。

软件设计随时在改变，C#也会发展和改变，很可能过时候就会在 2.0 上发展几个版本。这是一个工具，你可以每天都利用它，至少是大多数日子。学习更多的官方说明，而且总是站在这些内容的最项上。