《C# 12.0 本质论》

[美] 马克・米凯利斯（Mark Michaelis） 著

周靖 译（https://bookzhou.com）

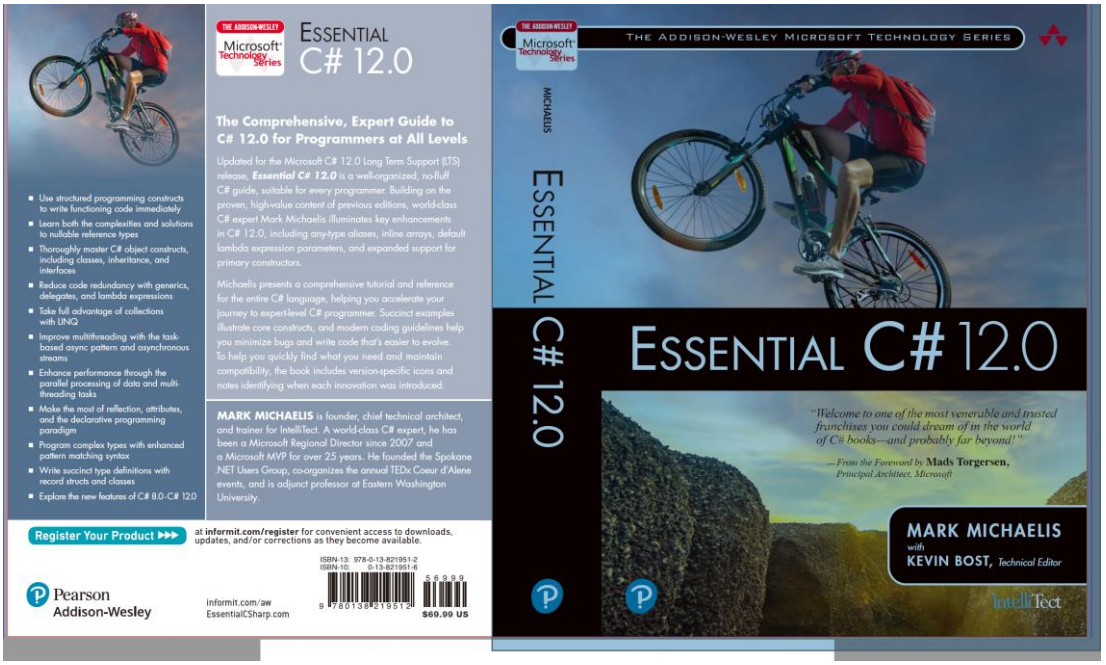
清华大学出版社

北 京

C# 12.0 本质论

（美）Mark Michaelis 著

周靖 译



全面、专业的 C# 12.0 指南，专为各种水平的程序员而设计

本书针对 C# 12.0 长期支持（LTS）版本进行了全面更新，继承以前各版之精华，由世界级 C#专家 Mark Michaelis 撰写，深入解析 C# 12.0 的主要升级功能，包括类型别名、内联数组、默认 Lambda 表达式参数以及对许多主要语言构造的扩展支持。

Michaelis 非常全面地讲解了 C#语言，助你快速成为 C#高级程序员。书中以简洁的例子阐释核心概念，并强调现代设计规范，助你减少错误，编写易于进化的代码。

Mark Michaelis 是创新软件工程和咨询公司 IntelliTect 的创始人，任公司首席技术架构师和培训师。作为一位世界级的 C#专家，他自 2007 年以来一直是 Microsoft Regional Director，是超过 25 年的 Microsoft MVP，是斯波坎.NET 用户组的创始人，是年度 TEDx 科尔德蓝活动的共同组织者，而且是东华盛顿大学（EWU）的兼职教授。

- 立即使用结构化编程构造编写可以实际工作的代码
- 了解可空引用类型的复杂性及解决方案
- 全面掌握 C#对象构造，包括类、继承和接口
- 利用泛型、委托和 Lambda 表达式减少代码冗余
- 通过 LINQ 充分利用集合
- 通过基于任务的异步模式和异步流改进多线程处理

- 通过并行数据处理和多线程任务来提升性能
- 充分利用反射、属性和宣告式编程范式
- 用增强的模式匹配语法来编程复杂的类型
- 通过 `record struct` 和 `record class` 来编写简洁的类型定义
- 探索 C# 8.0～C# 12.0 的新特性

献给我的家人：Elisabeth、Benjamin、Hanna 和 Abigail。感谢你们的理解和支持，容忍我在写作过程中的长时间缺席，尤其是在你们最需要我的时刻。我由衷感激你们的包容和鼓励！

同时，我也要向我的朋友们以及 IntelliTect 的团队致以深深的感激。在我利用工作时间进行写作时，感谢你们无私地接手我的工作，协助优化内容，以及确保本书的代码库运行顺畅。特别感谢你们的努力和才智，使得 EssentialCSharp.com 这个项目得以实现和成功运行。

译者序

自 2007 年《C# 2.0 本质论》面世以来，我便与这本书以及作者马克·米凯利斯（Mark Michaelis）结下了不解之缘。除去未参与《C# 6.0 本质论》和《C# 8.0 本质论》的翻译，其余译本均出自我的笔下。

马克是位不折不扣的实干家。作为铁人三项运动员和技术领域的佼佼者，他深入探究事物本质的能力令人敬佩。对于任何问题，他都能深究其表面之下的本质。任何问题他都能做到不仅知其然，还知所以然。这种深度思考体现在本书中，诸多知识点被巧妙且紧密地联系起来。最开始不明白的问题，一气呵成读下去会有恍然大悟的感觉。正如微软首席架构师 Mads Torgersen 在本书的推荐序中所言，“一样东西通过了 Mark 的测试，就没什么好担心的了！”

马克的《C#本质论》系列版本，总让我想起罗素。作为哲学家、数学家和逻辑学家，罗素的行文向来字字珠玑，具有较强的感染力，很容易引起读者的共鸣，给读者带来许多启发。在《罗素回忆录：来自记忆力的肖像》中，他介绍了自己是如何写作的：“我希望能够用最少的词把每件事情说得一清二楚。我肯花时间设法找出最简洁的方式把某些事情毫不含糊地表达出来，为此，我往往不惜牺牲追求美学上优点的一切企图。”在他 21 岁之前，罗素希望自己的写作风格够接近于约翰·米尔的风格，因为后者有值得自己效仿的句型结构和主题拓展方式。经过种种尝试，罗素终于醒悟，意识到对华丽词藻和张扬写作风格的模仿会诱发一定程度的虚伪性，认识到所有模仿都是危险的，由此，他总结出三大简单的写作准则：其一，如果可以使用一个简单的词，就永远不要使用一个复杂的词；其二，如果想要做一个包含大量必要条件在内的说明，那么尽量把这些必要条件放在不同的句子里分别说清楚；其三，不要让句子的开头引导读者得出一个与结尾有抵触的结论。

由此联想到马克的这本书，优秀的专业技术类图书，其表述方式和措辞首先考虑的是以读者为本，而不是满篇都是只有少数博学之士才能看懂的行话或术语。在这次翻译《C# 12.0 本质论》的过程中，我有颇多这样的感受。真正的技术大师，并不会一味地追求形式化、科学化和精致化而导致专业知识远离大部分读者，直到竖起不可逾越的篱笆。真正优秀的作者，是像马克这样的，他们拒绝向读者介绍未经过自己检验的东西，他们拒绝伪装，从来不会故意用高深莫测且让大家云里雾里的话术来凸显自己的专家身份。

在我看来，《C# 12.0 本质论》实际上完成了一项非常困难的任务。前面几章的内容很易于被刚入门的开发者理解。而在后期的章节中，作者毫不藏私，将自己二十多年来对 C#语言的理解倾囊以授，为有经验的开发人员提供了挖掘 C#潜能所需要的技术细节。马克是组织内容的高手。从第 1 章起，他就成功赢得了许多读者（甚至是高手）的信任。与此同时，全书的内容还做到了通俗易懂且没有半句废话的程度。

这一版是历史上改动最大的一版。针对 C# 12.0 的新特性，内容编排有了很大变化。感谢

框架和语言的进步，以前实现比较起来繁琐的代码，现在变得更简洁了。当然结果就是，全书几乎所有代码和相关内容都要重新设计。作为译者，加上错过了上一版，所以面对的基本是一本全新的书。

下面简单总结了中文版的一些特色：

- 思维导图中的编号对应二级小节编号
- 在“语言对比”中删除了涉及 Visual Basic 的几乎所有内容
- 中文版本地化了所有代码，包括注释和 UI
- 近 150 条“译注”
- 对原书（纸质版）进行了大量勘误，并反馈回作者（100 条以上），译者主页也提供了英文版勘误

说到本书源代码，不得不说这一版的呈现方式是最完美的。本书在 GitHub 上有专门的项目（*https://github.com/IntelliTect/EssentialCSharp*），读者可以随时下载最新代码并在 Visual Studio 2022 中打开。中文版读者则可以访问译者主页（*https://bookzhou.com*）或 GitHub 项目（*https://github.com/transbot/EssentialCSharp*），获取配套的中文版资源。所有代码的注释和 UI 均已中文本地化。如果访问 GitHub 有困难，可以直接从译者主页下载。另外，务必查看源代码中的 README.md 文件了解用法。

当涉及到 C#编程，不得不提一下这个领域的“三剑客”，我亲自翻译了它们。首先，《深入 CLR》第 4 版（原《CLR via C#》），为你提供了一个高瞻远瞩的视角，深入解析了运行时和框架——C#语言的根基。语言的每一项设计都是基于这个核心架构构建的。其次，本书《C# 12.0 本质论》，它如同一个全面且深入的语言百科全书，覆盖了 C#语言的方方面面。最后，《Visual C#从入门到精通》第 10 版虽然同样涵盖了语言的基础知识，但更多地是从 GUI 编程的角度帮助你理解 C#编程。注意，这三本书均通过了 Visual Studio 2022 的测试。

感谢作者马克• 米凯利斯(Mark Michaelis)，他是一位非常有激情和活力的技术专家。翻译过程中，他热情、耐心地解释我所提出的问题，并虚心、坦诚地采纳了我提出的修改意见。还要感谢我的家人，尤其是女儿周子衿（Ava Zhou），她总能从一些新奇的角度来帮我重新认识这个世界。

衷心希望每一位读者都能通过这本书，踏上一段愉快而激动人心的 C#之旅！

——周靖，2024

推荐序

您手上是 C#最值得尊敬、最权威的参考书之一，作者为此付出了非凡努力！Mark Michaelis 的《C#本质论》系列多年来一直是畅销经典。而我刚认识 Mark 的时候，这本书还处于萌芽阶段。

2005 年 LINQ（语言集成查询）公布时，我才刚加入微软，正好见证了 PDC 会议上令人激动的公开发布。虽然我对技术本身几乎没有什么贡献，但它的宣传造势我可是全程参加了。那时人人都在谈论它，宣传小册子满天飞。那是 C#和.NET 的大日子，至今依然令人难忘。

但会场供人上手实作的区域却相当安静，那儿的人可以按部就班地试验处于预览阶段的技术。我就是在那儿遇见 Mark 的。不用说，他一点儿没有“按部就班”的意思。他在做自己的试验，梳理文档，和别人沟通，忙着鼓捣自己的东西。

作为 C#社区的新人，我感觉自己在那次会议上见了许多人。但老实说，当时太混乱了。唯一记得清的就是 Mark。因为当我问他是否喜欢这个新技术时，他不像别人那样马上开始滔滔不绝，而是非常冷静地说：“还不确定，要自己搞一搞才知道。”他希望完整地理解并消化一种技术，之后才将自己的理论告知于人。

所以不像我本来设想的那样发生一次快餐式的对话。相反，我们的对话相当坦诚、颇有营养。像这样的交流好多年都没有发生了。新技术的细节、造成的后果和存在的问题都涉及到了。对我们这些语言设计者，Mark 是最有价值的社区成员。他非常聪明，会打破砂锅问到底，能深刻理解一种技术对于真正的开发人员的影响。但是，最根本的原因可能还是他的坦诚，他从不惧怕说出自己的想法。一样东西通过了 Mark 的测试，就没什么好担心的了！

这些特质也使 Mark 成为一名出色的技术作家。他的文字直指技术的本质，向读者提供最完整的信息，没有废话，能敏锐地看出技术的真正价值和问题。没人能像这位大师一样帮你正确理解 C# 12.0。

请好好享用本书！

——Mads Torgersen，微软首席架构师

目录

前言

在软件工程的发展历史中，用于编写计算机程序的方法经历了几次思维模式的重大转变。每种思维模式都以前一种为基础，宗旨都是增强代码的组织，并降低复杂性。本书将带领你体验相同的思维模式转变过程。

本书开始几章指导你学习**顺序编程结构**。在这种编程结构中，语句按执行顺序写。该结构的问题在于，随着需求的增加，复杂性也呈指数级增加。为了降低复杂性，代码块被移到方法中，形成了**结构化编程模型**。在这种模型中，可以从一个程序中的多个位置调用同一个代码块，无需复制。但是，即使有这种结构，程序还是很快就会变得臃肿不堪，需进一步抽象。所以，在此基础上，人们又提出了**面向对象编程**的概念，这将在第 6 章开始讨论。在此之后，你将继续学习其他编程方法，比如基于接口的编程和 LINQ（以及它促使集合 API 发生的改变），并最终学习通过特性（attributes）进行初级的声明性编程[①]（第 18 章）。

本书有以下三个主要职能。

- 全面讲述 C#语言，其内容已远远超过了一本简单的教程，为你进行高效率软件开发打下坚实基础。
- 对于已熟悉了 C#的读者，本书探讨了一些较为复杂的编程思想，并深入讨论了语言最新版本（C# 12.0 和.NET 8）的新功能。
- 它是你永远的案头参考——即便在你精通了这种语言之后。

成功学习 C#的关键在于，要尽可能快地开始编程。不要等自己成为一名理论“专家”之后才开始写代码。所以，不要犹豫，马上开始写程序吧。作为迭代开发[②]思想的追随者，我希望即使一名刚开始学习编程的新手，在第 2 章结束时也能动手写基本的 C#代码。

许多主题本书没有讨论。你在本书找不到 ASP.NET、Entity Framework、Mauri、智能客户端开发以及分布式编程等主题。虽然这些主题与.NET 有关，但它们都值得专门用一本书来讲解。幸好，本书重点在于 C#以及*基类库*中的类型。本书将帮助你打下一个坚实

---

① 译注：与声明性编程或者宣告式编程（declarative programming）对应的是命令式编程；前者表述问题，后者实际解决问题。

② 译注：简单地说，迭代开发是指分周期、分阶段进行一个项目，以增量方式逐渐对其进行改进的过程。

的 C#语言基础，在上述任何高级领域继续深入时，都会有一种游刃有余的感觉。

本书面向的读者

写作本书时，我面临的一个挑战是如何在持续吸引高级开发人员眼球的同时，不至于因为使用了 assembly、link、chain、thread 和 fusion 等大量字眼而打击到初学者的信心，否则许多人会以为这是一本讲冶金而不是程序设计的书。[①]本书的主要读者是已经有一定编程经验，并想多学一种语言来“傍身”的开发人员。不过，我还是精心地编排了本书的内容，使之对于各种层次的开发人员都有足够大的价值。

- **初学者**：如果你是编程新手，本书帮助你从入门级程序员过渡成为 C#开发人员，消除以后在面临任何 C#编程任务时的害怕心理。本书不但教你语法，还教你养成良好的编程习惯，为将来的编程生涯打下良好基础。
- **熟悉结构化编程的程序员**：学习外语最好的方法就是“沉浸法”。[②]类似地，学习一门计算机语言最好的方法就是边动手边学习，而不是一直“纸上谈兵”。基于这个前提，本书最开始的内容是那些熟悉结构化编程的开发人员很容易上手的。到第 5 章结束时，这些开发人员应该可以开始写基本的控制流程序。然而，要成为真正的 C#开发人员，记住语法只是第一步。为了从简单程序过渡到企业级开发，C#开发人员必须本能地从对象及其关系的角度来思考问题。为此，第 6 章开始介绍类和面向对象开发。对于 C，COBOL 和 FORTRAN 等结构化编程语言，虽然它们仍在发挥作用，但作用会越来越小。所以，软件工程师们应逐渐开始了解面向对象开发。C#是进行这一思维模式转变的理想语言，因为它本来就是基于“面向对象开发”这一中心思想来设计的。
- **熟悉“基于对象”和“面向对象”理念的开发人员**：C++，Java，Python，TypeScript，Visual Basic 和 Java 程序员都可归于此类。对于分号和大括号，他们可是一点儿都不陌生！简单浏览一下第 1 章的代码，你会发现，从核心上讲，C#类似于你熟知的 C 和 C++风格的语言。
- **C#专家**：对于已经精通 C#的人，本书可供你参考不太常见的语法。此外，对于

[①] 译注：上述每个单词在计算机和冶金领域都有专门的含义，所以作者用它们开了一个玩笑。例如，assembly 既是“程序集”，也是“装配件”；thread 既是“线程”，也是“螺纹”。

[②] 译注：沉浸法，即 immersion approach，是指想办法让学习者泡到一个全外语的环境中，比如孤身一人在国外生活或学习。

在其他地方强调较少的一些语言细节以及微妙之处，我提出了自己的见解。最重要的是，本书提供了编写可靠和易维护代码的指导原则及模式。你教别人学 C#时，本书也颇有助益。从 C# 3.0 到 C# 12.0 最重要的一些增强包括：

1. 字符串插值（第 2 章）
2. 隐式类型的变量（第 3 章）
3. 元组（第 3 章）
4. 可空引用类型（第 3 章）
5. 模式匹配（第 4 章）
6. 扩展方法（第 6 章）
7. 分部方法（第 6 章）
8. 默认接口成员（第 8 章）
9. 匿名类型（第 12 章）
10. 泛型（第 12 章）
11. Lambda 语句和表达式（第 13 章）
12. 表达式树（第 13 章）；
13. 标准查找操作符（第 15 章）
14. 查询表达式（第 16 章）；
15. 动态编程（第 18 章）；
16. 用任务编程库（TPL）和 async 进行多线程编程（第 20 章）；
17. 用 PLINQ 进行并行查询处理（第 21 章）；
18. 并发集合（第 22 章）。

考虑到许多人还不熟悉这些主题，本书围绕它们展开了详细讨论。涉及高级 C#开发的还有“指针”这一主题，该主题将在第 23 章讨论。即使是有经验的 C#开发人员，也未必能很透彻地理解这一主题。

本书特色

本书是语言参考书，遵循核心 C#语言规范。为了帮助读者理解各种 C#构造，书中用大量例子演示了每一种特性，而且为每个概念都提供了相应的指导原则和最佳实践，以确保代码能顺利编译、避免留下隐患，并获得最佳的可维护性。

为增强可读性，所有代码均进行了特殊的格式化，而且每章内容都在最开始用一幅思维导图来概括。

网站

本书的交互式网站位于 *https://essentialcsharp.com*，提供了在线章节（英文版），并支持全文搜索。网站不久后还会上线对许多代码清单进行交互式代码编辑和客户端编译的功能。这有助于使你专注于语言本身，而不会因为安装或 `dotnet` 设置问题而分心。

源代码下载

除了 EssentialCSharp.com 网站之外，所有英文版和中文版源代码都可以从 GitHub 获取，网址是：

- *https://github.com/IntelliTect/EssentialCSharp*（英文版）
- *https://github.com/transbot/EssentialCSharp*（中文版，含中文注释和 UI 本地化）；如果访问有困难，也可以直接从译者主页（*https://bookzhou.com*）获取

这样你就可以下载或在本地克隆，方便查看代码或者进行修改，并实际体验效果。动手实作，这是学习语言细节的一种很好的方式。

C#设计规范

本书新版本最重大的改进之一就是增加了大量“设计规范”，下面是第 17 章的例子。

设计规范

DO ensure that equal objects have equal hash codes

要确保相等的对象有相等的散列码。

DO ensure that the hash code of an object never changes while it is in a hash table.

要确保对象在散列表中的时候散列码永不变化。

DO ensure that the hashing algorithm quickly produces a well-distributed hash.

要确保散列算法快速生成良好分布的散列码。

DO ensure that the hashing algorithm is robust in any possible object state.

要确保散列算法在任何可能的对象状态中的健壮性。

一名知道语法的程序员，和一名能因地制宜写出最高效代码的专家，区分两者的关键就是这些设计规范。专家不仅能让代码通过编译，还遵循最佳实践，降低出现 bug 的机率，并使代码的维护变得更容易。设计规范强调了一些关键原则，开发时务必注意。

示例代码

本书大多数代码都能在公共语言基础结构（Common Language Infrastructure，CLI）的任何实现上运行，但重点还是.NET 实现。很少使用平台或厂商特有的库，除非需要解释只和那些平台相关的重要概念（例如，解释如何正确处理 Windows 单线程 UI）。任何 C# 8.0，9.0，10.0，11.0 或 12.0 特有的代码都在书末的 C# 版本索引中进行了明确标注。

下面是一个示例代码清单。

代码清单 1.21　为代码添加注释

```
public class CommentSamples
{
    public static void Main()
    {
        string firstName;  // 用于存储名字的变量
        string lastName;   // 用于存储姓氏的变量

        Console.WriteLine("嘿，你！");

        Console.Write /* 不换行 */ ("请输入你的名字：");
        firstName = Console.ReadLine();

        Console.Write /* 不换行 */ ("请输入你的姓氏：");
        lastName = Console.ReadLine();

        /* 使用字符串插值在控制台上显示问候语*/
        Console.WriteLine($"你的全名是{ firstName } { lastName }。");
        // 这是程序清单
        // 的结尾
    }
}
```

下面解释了具体的格式：

- 所有代码都进行了彩色标注（黑白印刷的书页会呈现不同的灰度）
- 省略号表示无关代码已省略。

```
// ...
```

- 某些代码行添加了底纹，表明这是对上个代码清单的修改，或者强调当前正在讲解的某个主题，如下例所示。

代码清单 2.23　错误；string 是不可变的

```
Console.Write("输入文本: ");
string text = Console.ReadLine();

// UNEXPECTED: text 并没有转换为全大写
text.ToUpper();

Console.WriteLine(text);
```

- 某些代码清单后列出了相应的控制台输出。由用户输入的内容加粗，如下例所示。

输出 1.7

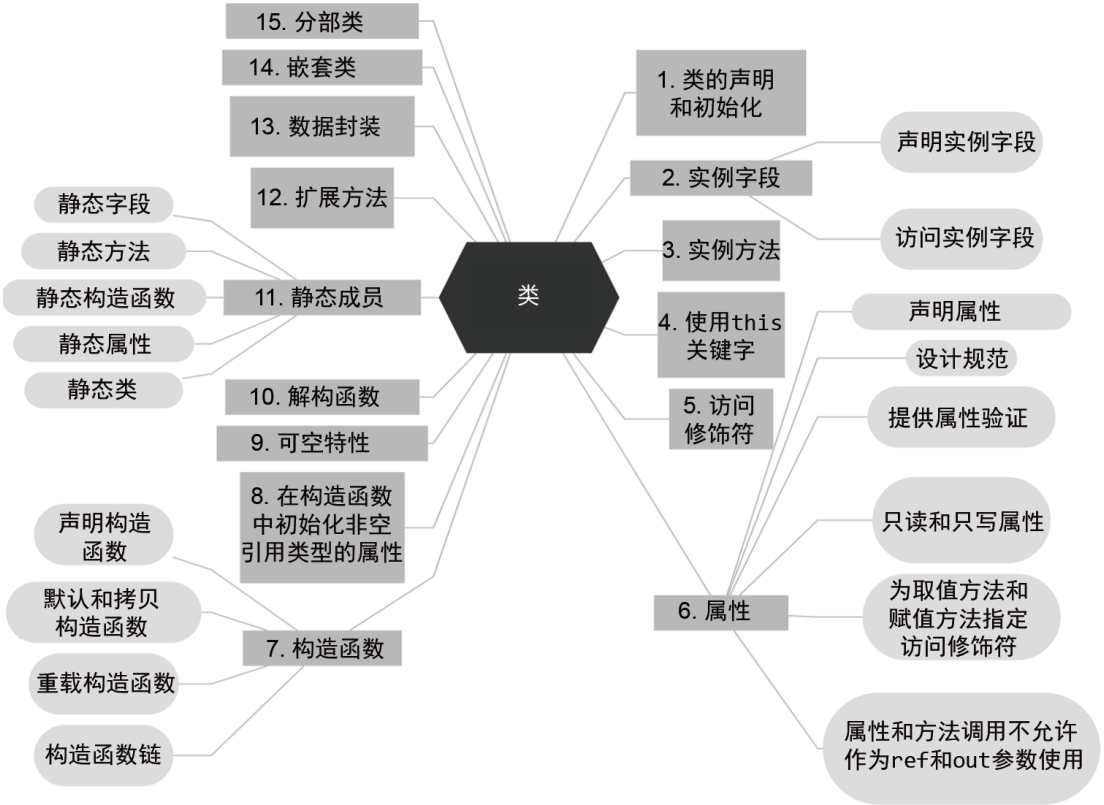
```
嘿，你!
请输入你的名字: Inigo
请输入你的姓氏: Montoya
你的全名是 Inigo Montoya。
```

虽然我也可以在书中提供完整代码方便你复制，但这样会分散注意力。因此，你需要在自己的程序中修改示例代码。书中的代码主要省略了错误检查，比如异常处理。

本书中文版读请访问译者主页 *https://bookzhou.com* 获取示例代码和更多配套资源。

思维导图

每章开头都有一幅“思维导图”。它是本章内容的一个提纲，为读者提供对每章内容的快速参考。下面是一个例子（摘自第 6 章）。



思维导图中心显示了每一章的主题，所有高级主题都围绕这个中心展开。利用思维导图，读者可方便地搭建自己的知识体系，可以从一个主题出发，更清楚地理解其周边的各个具体概念，避免中途纠缠于一些不相干的枝节问题。

分类解说

根据自己的编程水平，利用书中的标志来帮助自己轻松找到适合自己的内容。

- **初学者主题**：特别为刚入门的程序员提供的定义或解释。
- **高级主题**：使有经验的开发人员将注意力放在他们最关心的内容上。
- **注意**：“注意”框强调了读者应注意的要点。
- **语言对比**：分散于正文中的“语言对比”描述了 C#和其他语言的关键差异，为具有其他语言背景的读者提供 C#指引。

本书内容组织

总的来说，软件工程的宗旨就是管理复杂性。本书基于该宗旨来组织内容。第 1 章～第 5

章介绍结构化编程，学习这些内容后，可以立即开始写一些功能简单的代码。第 6 章～第 10 章介绍了 C#的面向对象构造，新手应在完全理解了这几章的内容之后，再开始接触本书其余部分更高级的主题。第 12 章～第 14 章介绍更多用于降低复杂性的构造，讲解了当今几乎所有程序都要用到的通用设计模式。理解了它们之后，才能更轻松地理解如何通过反射和特性来进行动态编程。后续章节将广泛运用它们来实现线程处理和互操作性。

本书最后专门用一章（第 24 章）讲解 CLI。这一章在开发平台的背景下讲述 C#语言。之所以要放到最后，是因为它非 C#特有，而且不涉及语法和编程风格问题。不过，本章适合在任何时候阅读，或许最恰当的时机是在阅读了第 1 章之后。

下面是每一章的内容提要。（**加黑**的标题表明那一章含有 C# 10.0～C# 12.0 的内容。）

- ***第1章——C#概述***：本章在展示了用C#写的“Hello World”程序之后对其进行细致分析。目的是让读者熟悉C#程序的“外观和感觉”，并理解如何编译和调试自己的程序。另外，还简单描述了C#程序的执行上下文及其中间语言（Intermediate Language，IL）。

- *第2章——数据类型*：任何有用的程序都要处理数据，本章介绍了C#的基元数据类型。

- *第3章——深入数据类型*：本章深入讲解数据类型的两大类别：值类型和引用类型。然后讲解了隐式类型的局部变量、元组、可空修饰符以及C# 8.0引入的可空引用类型。最后深入讨论了基元数组结构。

- *第4章——操作符和控制流*：计算机最擅长重复性操作，为了利用该能力，需要知道如何在程序中添加循环和条件逻辑。本章还讨论了C#操作符、数据转换和预处理器指令。

- ***第5章——方法和参数***：本章详细讨论了方法及其参数，其中包括传值、传引用和通过out参数返回数据。从C# 4.0开始支持默认参数，本章解释了具体如何使用。

- ***第6章——类***：前面已学过了类的基本构成元素，本章将将这些构造合并为具有完整功能的类型。类是面向对象技术的核心，它定义了对象模板。

- *第7章——继承*：继承是许多开发人员的基本编程手段，C#更是提供了一些独特构造，比如new修饰符。本章讨论了继承语法的细节，其中包括重写（overriding）。

- ***第8章——接口***：本章讨论如何利用接口来定义类之间的“可进行版本控制的交互契约”（versionable interaction contract）。C#同时包含显式和隐式接口成员实现，可以实现一个额外的封装等级，这是其他大多数语言所不支持的。本章最后用一节讨论了

接口版本控制问题，并强调了C# 8.0的引入的“默认接口成员”的作用。

- ***第9章——结构和记录***：C# 9.0为结构引入了记录（`record`）的概念，并在C# 10中把它扩展到值类型（即`record struct`）。尽管定义引用类型的情况更普遍，但有时确实需要定义行为与C#内置的基元类型相似的值类型。本章介绍如何创建自定义结构（`struct`），并强调了它的一些要特别注意的地方。

- *第10章——良好形式的类型*：本章讨论了更高级的类型定义，解释如何实现操作符，比如+和转型操作符，并描述如何将多个类封装到一个库中。另外，还演示了如何定义命名空间和XML注释，并讨论了如何基于垃圾回收机制来设计令人满意的类。

- *第11章——异常处理*：本章延伸讨论第5章引入的异常处理机制，描述了如何利用异常层次结构创建自定义异常。另外，还强调了异常处理的一些最佳实践。

- *第12章——泛型*：泛型或许是C# 1.0最缺少的功能。本章全面讨论自C# 2.0引入的泛型机制。此外，C# 4.0增加了对协变和逆变的支持，本章将在泛型背景中探讨它们。

- ***第13章——委托和Lambda表达式***：正是因为委托，才使C#与其前身语言（C和C++等）有了显著不同，它定义了在代码中处理事件的模式。这几乎完全消除了写轮询例程的必要。Lambda表达式是使C# 3.0的LINQ成为可能的关键概念。通过本章的学习，将知道Lambda表达式是在委托的基础上构建起来的，它提供了比委托更优雅和简洁的语法。本章内容是第14章讨论的集合API的基础。

- *第14章——事件*：封装起来的委托（称为事件）是公共语言运行时（Common Language Runtime，CLR）的核心构造。本章还讲解了自C# 2.0引入的另一个特性，即匿名方法。

- *第15章——支持标准查询操作符的集合接口*：通过讨论新的`Enumerable`类的扩展方法，向你介绍C# 3.0引入的一些简单而强大的改变。`Enumerable`类造就了全新的集合API，即“标准查询操作符”，本章对其进行了详细讨论。

- ***第16章——使用查询表达式的LINQ***：如只使用标准查询操作符，会形成让人难以辨认的长语句。但查询表达式提供了一种类似SQL风格的语法，有效地解决了该问题。本章会详细讨论这种表达式。

- *第17章——构建自定义集合*：构建用于操纵业务对象的自定义API时，经常都需要创建自定义集合。本章讨论了具体如何做。还介绍了能使自定义集合的构建变得更简单的上下文关键字。

- *第18章——反射、特性和动态编程*：20世纪80年代末，程序结构的思维模式发生了根本性的变化，面向对象的编程是这个变化的基础。类似地，特性（attributes）使声明性编程和嵌入元数据成为可能，因而引入了一种新的思维模式。本章探讨了特性的方方面面，并讨论了如何通过反射机制来获取它们。本章还讨论了如何通过基类库（Base Class Library，BCL）中的序列化框架来进行文件的输入输出。C# 4.0新增了`dynamic`关键字，能将所有类型检查都移至运行时进行，因而极大扩展了C#的能力。

- *第19章——多线程处理*：大多数现代程序都要求用线程执行长时间运行的任务，同时确保对并发事件的快速响应。随着程序变得越来越复杂，必须采取其他措施来保护这些高级环境中的数据。多线程应用程序的编写比较复杂。本章讨论了如何操纵线程（包括如何取消）以及如何在任务上下文中进行异常处理。

- *第20章——基于任务的异步模式编程*：本章深入探讨了基于任务的异步模式以及相应的`async`/`await`语法。它极大地简化了多线程编程。另外，还讲解了C# 8.0引入的异步流的概念。

- *第21章——并行迭代*：为了优先性能，一个简单的方式是使用`Parallel`对象或并行LINQ（PLINQ）库在数据上进行并行迭代。

- *第22章——线程同步*：本章对前几章的内容进行扩展，演示了如何利用一些内建线程处理模式来简化对多线程代码的显式控制。

- ***第23章——平台互操作性和不安全的代码***：必须意识到，C#仍然是相对年轻的一种语言，现有的许多代码都是用其他语言写成的。为了用好这些现有代码，C#通过P/Invoke提供了对互操作性（调用非托管代码）的支持。此外，C#允许使用指针，也允许执行直接内存操作。虽然使用了指针的代码要求特殊权限才能运行，但它具有与C风格的API完全兼容的能力。

- *第24章——公共语言基础结构(CLI)*：事实上，C#被设计成一种在CLI的顶部工作的最有效的编程语言。本章讨论了C#程序与底层“运行时”及其规范的关系。

希望本书成为你打磨 C#专业技能的一个极好的资源。另外，在你能熟练使用 C#后，本书仍然可以作为你的案头参考书，在遇到自己不熟悉的领域时参考。

— Mark Michaelis
IntelliTect.com/mark，mark.michaelis.net

X (前Twitter): @Intellitect, @MarkMichaelis

致谢

世上没有任何一本书是作者单枪匹马就能呈现到你面前的，在此，我要向此过程中帮助过我的所有人致以衷心感谢。排名不分先后，但家人必然第一。考虑到这已经是本书的第 8 版，你可以想象在过去 18 年里，我的家人为了让我写作牺牲了多少（更不用说之前的书了）。在 Banjamin、Hanna 和 Abigail 眼中，爸爸经常因为此书而无暇顾及他们，但 Elisabeth 承受得更多。家里大事小事全靠她一个人，她独自承担家庭的重任。（2017 年在外面度假时，好几天我都在“闭门造书”，他们更想去海边。）我感到万分抱歉，辛苦你们了！

在写作《C# 12.0 本质论》的过程中，我非常感激 Pearson 允许 IntelliTect 在 EssentialCSharp.com 网站上托管手稿。然而，如果没有 IntelliTect 团队的支持，我永远无法独立完成这项工作。我非常幸运能与一群出色的软件工程师一起工作，他们能在我缺席的情况下自主地创造出如此卓越的成果。我们花费了许多、许多小时来解析手稿、格式化它、编辑内容，并同时创建网站和出版版本。非常感谢以下为本书这一版的实现做出贡献的人：Benjamin Michaelis，Kevin Bost，Daniel Olvera，Jenny Curry，Mikaella Croskrey，Grant Woods，Zack Ward，Josh Willis，Andrew Scott，Anré “Ray” Tanner，Artem Chetverikov，Casey White，Austen Frostad，Tom Clark，Joseph Riddles，John Evans，Kelsey McMahon，Casey Schadewitz，Cameron J. Osborn，Nicole Glidden 和 Elizabeth Pauley。特别感谢 Kevin Bost 和我的儿子 Benjamin Michaelis。除了在代码上的贡献，他们还帮助管理整个 Web 工具项目，使网站变成现实。

我与 Kevin Bost 自 2013 年以来一直在 IntelliTect 工作，他的软件开发能力令我惊叹不已。他不仅精通 C#，在其他许多开发技术上也都是级别 10 的专家。因为这一切及更多原因，我今年邀请 Kevin Bost 以正式技术编辑的身份审阅这本书。真的非常感谢他，他对全书内容提出了许多独到的见解和改进。某些问题自初版以来一直存在，而其他人都没有看到。他对细节的关注，加上对卓越的坚持，使这本《C# 12.0 本质论》更有资格成为一本经典的 C#书籍。

Eric 带给我无尽的惊喜。他对 C#的精通程度让人敬畏，我非常欣赏他的精细修改，尤其是在追求术语精准性方面的不懈努力。在本书第 2 版中，他对与 C# 3.0 相关的章节进行了深入且广泛的改进。那一版中，我唯一的遗憾是没有让他审阅所有章节。但现在，这个遗憾已不复存在。Eric 认真且专注地审阅了《Essential C# 4.0》的每个章节，而在《Essential C# 5.0》和《Essential C# 6.0》中，他更是担任了共同作者的角色。从《Essential C# 7.0》直至《Essential C# 12.0》，我非常感激他担任技术编辑的贡献。谢谢你，Eric！我无法想象还有

谁能比你做得更好。正是有了你，这本书才真正完成了从“优秀”到“卓越”的飞跃。

就像 Eric 之于 C# ，很少有人像 Stephen Toub 那样对.NET Framework 多线程处理有如此深刻的理解。Stephen 专门审阅了（嗯，第三次了）重写的关于多线程的两章，并重点检查了 C# 5.0 的 `async` 支持。谢谢你，Stephen！

多年来，其他许多技术编辑对每一章都进行了详尽审查，以确保技术的准确性。我经常对这些人依然能够发现的微妙错误感到惊讶：Paul Bramsman，Kody Brown，Andrew Comb，Ian Davis，Doug Dechow，Gerard Frantz，Dan Haley，Thomas Heavey，Anson Horton，Brian Jones，Shane Kercheval，Angelika Langer，Neal Lundby，John Michaelis，Jason Morse，Nicholas Paldino，Jason Peterson，Jon Skeet，Michael Stokesbary，Robert Stokesbary 和 John Timney。

感谢 Pearson/Addison-Wesley 出版社的所有人在与我合作时的耐心，容忍我经常把注意力放在手稿之外的其他事物上。还要感谢 Malobika Chakraborty 和 Julie Nahil 在从提案到制作整个过程中对我的帮助。

作者简介

Mark Michaelis 是创新软件工程和咨询公司 IntelliTect 的创始人，任公司首席技术架构师和培训师。Mark 经常在全世界飞来飞去，在各种各样的会议上提供领导力或技术方面的演讲，并代表微软或其他客户进行演讲。他还撰写了大量文章和书籍，是东华盛顿大学（EWU）的兼职教授，是斯波坎.NET 用户组的创始人，也是年度 TEDx 科尔德蓝活动的共同组织者。

作为一位世界级的 C#专家，Mark 自 2007 年以来一直是 Microsoft Regional Director，还是超过 25 年的 Microsoft MVP。

Mark 拥有伊利诺伊大学哲学专业文学学士学位和伊利诺伊理工大学计算机硕士学位。

他不是痴迷于计算机，就是忙于陪伴家人或者玩壁球（2016 年暂停铁人三项训练）。他居住在华盛顿州的斯波坎，他和妻子伊丽莎白有三个孩子：本杰明、汉娜和阿比盖尔。

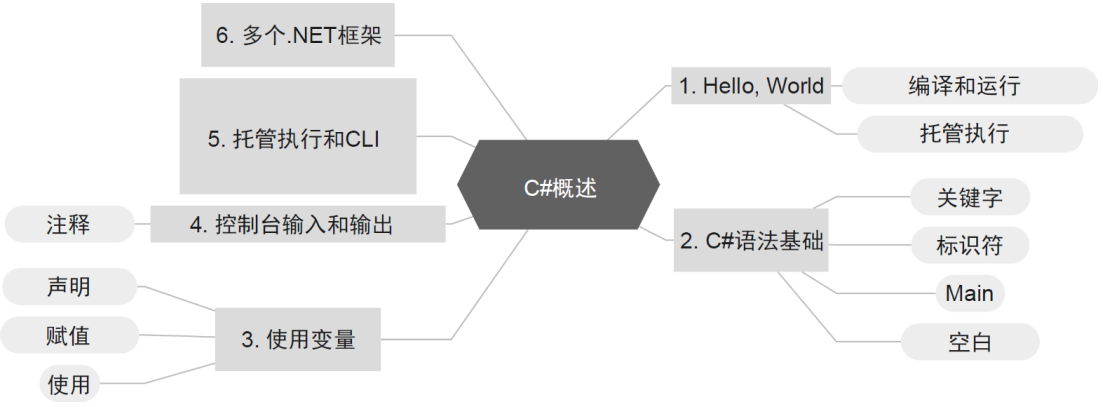
技术编辑简介

Kevin Bost 是一位成功的 Microsoft MVP，同时也是 IntelliTect 的高级软件架构师。他在构

建几个创新产品方面发挥了重要作用，其中包括 System.CommandLine，Moq.AutoMocker 和 ShowMeTheXAML。在工作之余，Kevin 经常在线上（youtube.keboo.dev）为其他开发人员提供指导，并维护着流行的 Material Design in XAML 工具箱（*http://materialdesigninxaml.net/*）。他还喜欢桌游、极限飞盘和骑摩托车。

第 1 章 C#概述

C#编程语言可以为各种操作系统（平台）开发软件组件和应用程序，包括移动设备、游戏主机、Web 应用、物联网（Internet of Things，IoT）、微服务和桌面应用程序等。此外，C#是免费的；事实上，它还完全开源，你可以查看、修改、重新分发，并向开源社区贡献你所做的任何改进。C#还是一种成熟的语言，基于作为前身的 C 风格语言（C、C++和 Java）的功能而设计[①]，所以有经验的程序员都能很快熟悉它。



本章通过一个传统的 HelloWorld 程序来介绍 C#，重点是 C#语法基础，包括如何定义一个 C#程序入口。通过本章的学习，你将熟悉 C#的语法风格和结构，并能开始写最简单的 C#程序。在讨论 C#语法基础之前，将简单介绍一下托管执行环境，并解释 C#程序在运行时是如何执行的。最后，我们会讨论变量声明、控制台输入/输出以及基本的 C#代码注释机制。

1.1 Hello, World

学习新语言最好的办法就是立即动手写代码。第一个例子是经典的 HelloWorld 程序，它在屏幕上显示一些文本。代码清单 1.1 展示了完整程序，我们将在之后的小节完成它的编译和运行。[②]

代码清单 1.1　用 C#编写的 HelloWorld 程序

[①] 第一次 C#设计会议在 1998 年召开。

[②] 如果不知道 Inigo Montoya 是谁，请找《公主新娘》（The Princess Bride）这部电影看看。

```
Console.WriteLine("你好，我叫 Inigo Montoya。");
```

注意，之所以能写如此简单的一行代码，需要用到 C# 9.0 引入的一个特性——**顶级语句**（top-level statements）。稍后会在代码清单 1.6 中展示另一种选择，那可以说是一种更典型的代码清单。有关顶级语句的更多信息，请参见第 5 章。

> **初学者主题：基本编译术语**
>
> **编译器**的作用类似于译员，代码从一种语言转换为另一种语言。通常，编译器负责将高级语言（如 C#）转换为计算机能直接理解的低级语言。在 C#的情况下，编译器输出的是一种**中间语言**（Intermediate Language，IL），到时会由第二个编译器将其翻译为机器语言。本章稍后的“托管执行和 CLI”一节会进一步讨论这个问题。**源代码**简单来说就是构成程序的文本，所以代码清单 1.1 就是这个 HelloWorld 程序的源代码。“代码”一词通常可与“源代码”互换使用。

1.1.1 创建、编辑、编译和运行 C#源代码

C#代码写好后还需要编译和运行。取决于你的操作系统，此时要决定下载哪个编译器。通常，这些实现被打包成**软件开发包**（Software Development Kit，SDK），其中包括编译器、运行时执行引擎、“运行时”能访问的语言可访问功能框架（参见本章后面的 “应用程序编程接口”一节）以及可能与 SDK 捆绑的其他工具（比如供自动化生成的生成引擎）。由于不同的平台使用了不同的编译器，而且 C#自 2000 年便公之于众了（参见本章后面的“多个.NET 框架”一节），所以我们目前有多种选择。

操作系统不同，安装指令也有所区别。有鉴于此，建议访问 *https://www.microsoft.com/net/download* 获取具体的下载和安装指令。先选好操作系统，再依据目标操作系统选择要下载的 SDK。此外，也可以选择使用哪个（哪些）.NET 实现（有时称为.NET 框架）。虽然我们可以在这里提供更多详细信息，但.NET 下载站点针对每种支持的组合都提供了最新的指令。虽然有多个框架版本可选，但最简单的还是下载默认的那个，它对应于最新的完全发布版本（显示为“长期支持”或 LTS）。

有许多源代码编辑工具可供选择，包括最基本的 Windows 记事本、Mac/macOS TextEdit 和 Linux vi 等。但是，建议选择一个稍微高级点的工具，它至少应支持语法彩色标注。支持 C#的任何代码编辑器都可以。如果还没有特别喜欢的，那么推荐开源编辑器 Visual Studio Code（*https://code.visualstudio.com*）。如图 1.1 所示，为了在 Visual Studio Code 中使用 C#，需要安装相应的 C#扩展。另外，如果在 Windows 或 Mac 上工作，那么还可以考虑 Microsoft Visual Studio 2022（或更高版本），详情请访问 *https://www.visualstudio.com*。所有这些都是免费的。

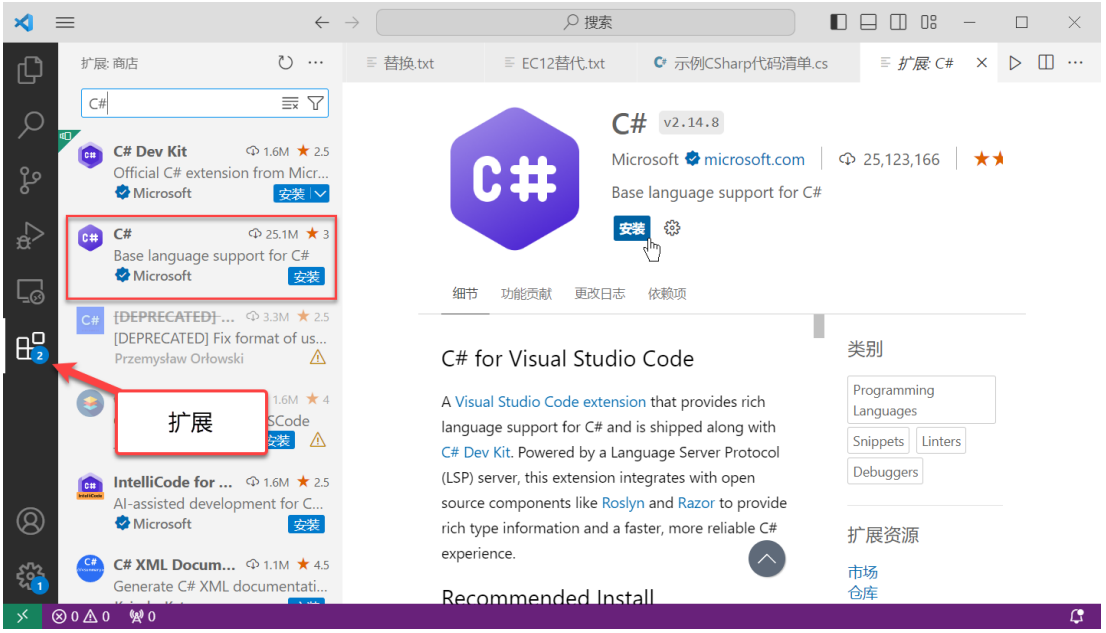


图 1.1 为 Visual Studio Code 安装 C#扩展模块

稍后就会提供这两种编辑器的操作指示。首先，我们依赖于命令行接口（CLI）工具 **dotnet CLI** 来创建 C#程序，并用它编译和运行程序。对于 Windows 和 Mac，我们将重点放在平台专用的 Visual Studio 2022 版本上。

使用 Dotnet CLI

Dotnet 命令 `dotnet` 是 Dotnet 命令行接口（或称 Dotnet CLI），可用于生成 C#程序的初始代码库，并编译和运行程序。[1]注意，这里的 CLI 代表“命令行接口”（Command-Line Interface）。为避免和“公共语言基础结构”（Common Language Infrastructure）的简称 CLI 混淆，本书在提到 Dotnet CLI 时都会附加 Dotnet 前缀。没有 Dotnet 前缀的 CLI 才是“公共语言基础结构”。安装好之后，首先验证一下可以在终端上执行 `dotnet` 命令。

以下是在 Windows，macOS 或 Linux 上创建、编译和执行 HelloWorld 程序的指示：

1. 打开终端（在 Microsoft Windows 上打开命令提示符，在 Mac/macOS 上打开 Terminal 应

[1] 这个工具大约是与 C# 7.0 同期发布的，取代了直接使用 C#编译器（csc.exe）进行编译的传统方式。

用，还可以执行 pwsh 命令来使用跨平台命令行接口 PowerShell[①]）。

2. 在想要存放代码的地方新建一个目录。作为本书的第一个例子，考虑使用 HelloWorld 或 EssentialCSharp/HelloWorld 这样的目录名。在命令行上执行：

```
mkdir HelloWorld
```

3. 导航到新目录，使之成为终端的当前位置。

```
cd HelloWorld
```

4. 在 HelloWorld 目录中执行 dotnet new console 命令来生成程序基架（或称“项目”）。随后会生成几个文件，最主要的就是 Program.cs 和项目文件 HelloWorld.csproj。

```
dotnet new console
```

5. 运行生成的程序。以下命令会编译并运行由 dotnet new console 命令创建的默认 Program.cs 程序。程序内容与代码清单 1.1 相似，只是输出变成了“Hello, World!”。

```
dotnet run
```

6. 虽然没有显式请求应用程序编译（或生成），但 dotnet run 命令在执行时已经隐式执行了 dotnet build 命令。

7. 编辑 Program.cs 文件，使代码与代码清单 1.1 一致。如果用 Visual Studio Code 打开并编辑 Program.cs，那么会体验到支持 C#的编辑器的好处，代码会用彩色标注不同类型的构造。要用 Visual Studio Code 打开并编辑，请执行以下命令。

```
code Program.cs
```

8. 也可以像输出 1.1 那样，一直在 Windows 命令行上操作。

9. 重新运行程序。

```
dotnet run
```

输出 1.1 还原了上述步骤。[②]

输出 1.1

```
1>
2> mkdir HelloWorld
3> cd HelloWorld
4> dotnet new console
```

[①] *https://github.com/PowerShell/PowerShell*

[②] 加粗的是由用户输入的内容。

```
已成功创建模板“控制台应用”。

正在处理创建后操作...
正在还原 F:\Work\Essential CSharp 12.0(原书第 8 版)\代码\HelloWorld\HelloWorld.csproj:
  正在确定要还原的项目…
  已还原 F:\Work\Essential CSharp 12.0(原书第 8 版)\代码\HelloWorld\HelloWorld.csproj (用时 298
ms)。
  已成功还原。
5> dotnet run
Hello, World!
6> echo Console.WriteLine("你好，我叫 Inigo Montoya。"); > Program.cs
7> dotnet run
你好，我叫 Inigo Montoya。
```

使用 Visual Studio 2022(Windows 或 Mac)

在 Visual Studio 2022 中的操作相似，只是现在不用命令行了，而是直接使用一个集成开发环境（IDE）。由于有菜单可供操作，所以不必一切都依赖于命令行。

1. 启动 Visual Studio 2022。

2. 单击“创建新项目”。如果没有显示启动窗口，那么可以选择“文件”|“启动窗口”来显示它，或者直接选择“文件”丨“新建”|“项目”（Ctrl+Shift+N）打开“创建新项目”对话框。

3. 在搜索框（Alt+S）中输入“控制台应用”，并选择“控制台应用”，如图 1.2 所示。如果还安装了其他语言，那么可以选择 C#来缩小搜索范围。选好模板后，单击“下一步”。

图 1.2 “创建新项目”对话框

4. 在“项目名称”框中输入 **HelloWorld**。为“位置”选择你的工作目录。如图 1.3 所示。完成后，单击“下一步”。

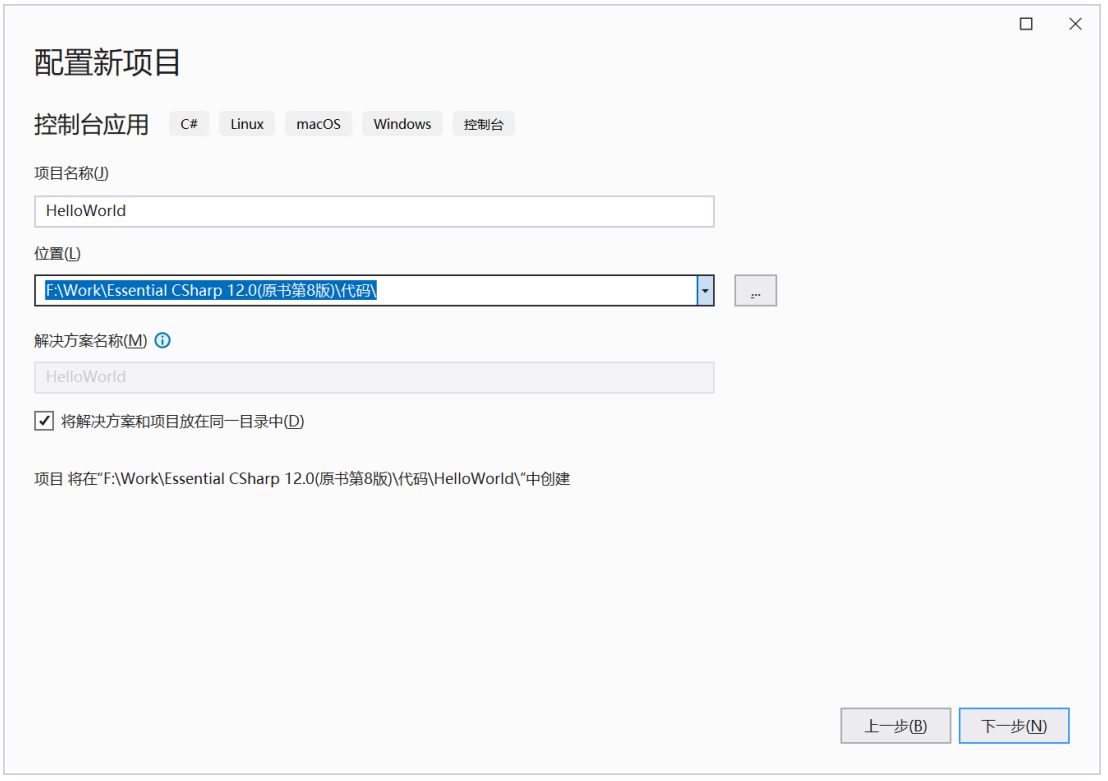


图 1.3 “配置新项目”对话框

5. 在“其他信息”对话框中，从“框架”下拉列表中选择“.NET 8.0 (长期支持)”。注意，请保持“不使用顶级语句”框的非勾选状态。单击“创建”。

6. 项目创建好后会打开 Program.cs 文件供编辑，如图 1.4 所示。

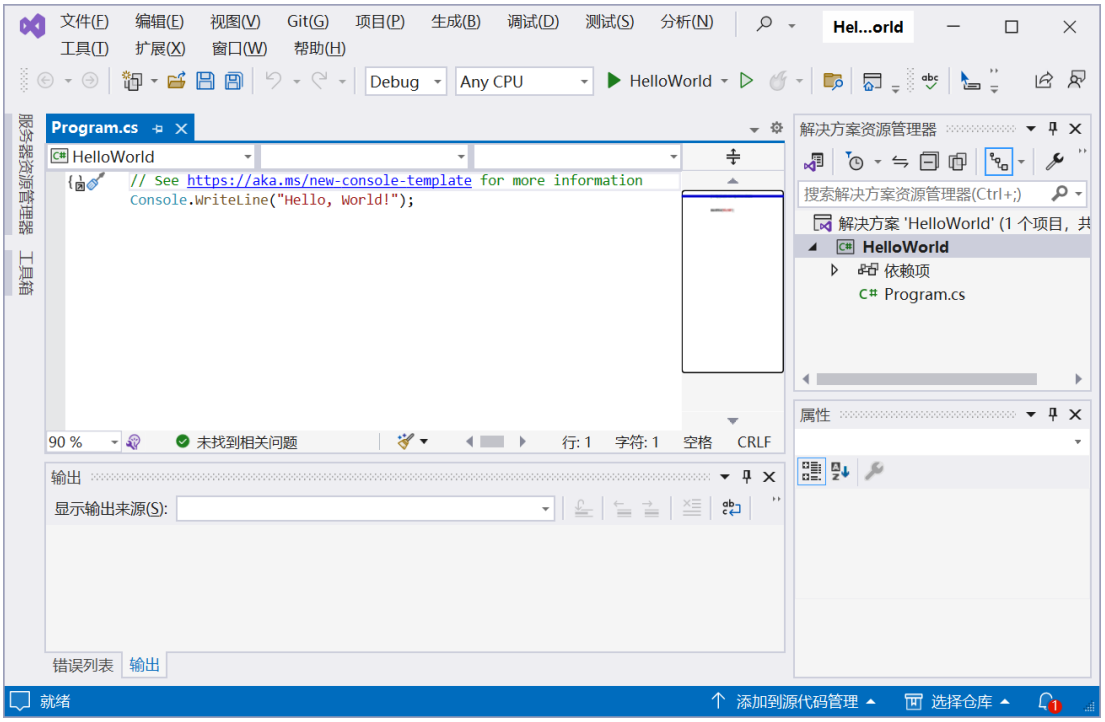


图 1.4 编辑 Program.cs 文件

7. 选择“调试”|“开始执行(不调试)”（Ctrl+F5）来生成并运行程序。会显示如输出 1.2 所示的终端窗口，只是第一行暂时为默认的“Hello, World! ”。

8. 将 Program.cs 修改成代码清单 1.1 的样子，重新执行步骤 7，获得如输出 1.2 所示的结果。

输出 1. 2

```
你好，我叫 Inigo Montoya。

...\HelloWorld.exe (进程 30456)已退出，代码为 0。
按任意键关闭此窗口. . .
```

1.1.2 理解项目

无论 Dotnet CLI 还是 Visual Studio 都会自动创建几个文件。第一个是名为 Program.cs 的 C# 文件。虽然可以选择任何名称，但一般都用 Program 这一名称作为控制台程序的起点。.cs 是所有 C#文件的标准扩展名，也是编译器默认要编译成最终程序的扩展名。为了使用代码清单 1.1 中的代码，可以打开 Program.cs 文件，并将其内容替换成代码清单 1.1 的。保存更新之前，注意代码清单 1.1 和默认生成的代码相比，唯一的区别是多了一条注释，并修改了双引号内的文本。

创建 C#项目时，会自动创建一个称为**项目文件**的配置文件。应用程序类型和.NET 框架不同，项目文件的内容也不同。但是，它至少会指出要生成（build）什么应用程序类型（控制台、Web、库等等）、支持什么.NET 框架、编译器设置以及启动应用程序需要什么设置。除此之外，还会指出代码的其他依赖项（称为库）。例如，代码清单 1.2 为上一节创建的简单.NET 控制台应用程序列出了项目文件的内容。

代码清单 1.2　示例. NET 控制台项目文件

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

在代码清单 1.2 中，注意应用程序类型被标识为一个.NET 8.0（`net8.0`）控制台应用（`Exe`）。其他两个元素 `Nullable`（C# 8.0）和 `ImplicitUsings`（C# 10.0）将分别在第 3 章的“声明允许为 `null` 的类型”一节和第 5 章的“`using` 指令”一节讨论。其他所有设置（比如要编译哪些 C#文件）则沿用默认值。例如，和项目文件同一目录（或子目录）中的所有*.cs 文件都会包含到编译中。[①]

1.1.3 编译和执行

执行 `dotnet run` 时，它首先隐式执行 `dotnet build` 命令来编译代码。后者的输出是一个名为 HelloWorld.dll 的**程序集**（assembly）。[②]程序集包含一组指定程序行为方式的指令。扩展名.dll 代表“动态链接库”（Dynamic Link Library，DLL）。.NET 的所有程序集都使用.dll 扩展名，控制台程序也不例外，就像本例这样（如果在 Windows 上，那么还会创建一个.exe 文件）。.NET 应用程序的编译输出会默认放到一个子目录中，子目录名称基于项目文件中的 `TargetFramework` 设置（如代码清单 1.2 所示）。本例使用的子目录名称是./bin/Debug/net8.0/。之所以使用 Debug 这个名称，是因为默认配置就是 debug。该配置

[①] *.cs 中的星号称为通配符，可以匹配任意数量的字符，包括零个字符。本例表示任何以 .cs 结尾的文件都将被包括。如果是 i*.cs，那么将包括任何以 i 开头并以 .cs 结尾的文件。

[②] 如果用 Microsoft .NET Framework 创建控制台程序，那么编译好的代码会放到一个 HelloWorld.exe 文件中。在已安装.NET Framework 的情况下，可以直接执行该文件。

造成输出针对调试而不是性能进行优化。编译好的输出本身不能执行。相反，需要用 CLI 来寄宿（host）代码。对于.NET 应用程序，这要求 dotnet.exe（在 Linux 和 Mac 上是 dotnet）进程作为应用程序的寄宿进程。这正是为什么要用 `dotnet run` 命令来运行程序的原因。话虽如此，还是有一种方法可以生成独立的可执行文件，其中包含必要的运行时文件，因此不需要安装 dotnet 运行时（在 Windows 上，如果已安装 dotnet 运行时，那么可以直接执行.exe 文件，而不必使用 `dotnet run`）。详情请参见“高级主题：发布独立可执行文件”。

> **高级主题：发布独立可执行文件**
>
> 可以使用 `dotnet publish` 命令来输出一个独立于 `dotnet` 命令来运行的可执行文件。为此，请使用`--runtime`（或`-r`）参数运行 `dotnet publish` 命令，该参数指定了要兼容的目标平台（操作系统）。例如，在大多数 Linux 平台上，可以在与.csproj 文件相同的目录中使用 `linux-x64`。
>
> ```
> dotnet publish --runtime linux-x64
> ```
>
> 执行此命令将创建一个目录（./bin/Debug/netcoreapp3.1/linux-x64/publish/），其中包含运行 HelloWorld 控制台程序所需的全部文件，而无需首先安装 dotnet 运行时。要执行 HelloWorld 程序，调用可执行文件名即可。如果当前目录不是发布目录，那么需要指定具体路径：
>
> ```
> ./bin/Debug/netcoreapp3.1/linux-x64/publish/HelloWorld
> ```
>
> （在 Windows 上，可执行文件名将包含一个 .exe 扩展名，但在运行程序时不需要输入该扩展名。）
>
> 注意，上述命令生成的可执行文件只能在兼容 linux-x64 的平台上运行。需要为每个目标平台执行 `dotnet publish` 命令。其他平台常见的“运行时”标识包括 `win-x64` 和 `osx-x64`（完整列表请访问 *https://tinyurl.com/3zccbuj8*）。
>
> 还可以将独立可执行文件发布到单个文件中（否则可能需要差不多 200 个文件）。如果想发布到单个可执行文件，请在命令中指定`-p:PublishSingleFile=true` 开关。
>
> ```
> dotnet publish --runtime linux-x64 -p:PublishSingleFile=true
> ```
>
> 有关`-p` 参数的更多信息，请参见第 10 章。

1.1.4 使用本书源代码

本书源代码和英文版手稿都可以通过 *https://essentialcsharp.com* 访问。源代码还可以直接从 GitHub 下载，网址是 *https://github.com/IntelliTect/EssentialCSharp*。中文版源代码的网址是 *https://github.com/transbot/EssentialCSharp*（或 *https://bookzhou.com*）代码的编译和运行说明可以在同一个地方的 README.md 文件中找到。

1.2 C#语法基础

成功编译并运行 HelloWorld 程序之后，我们来分析一下代码，了解它的各个组成部分。当然，代码清单 1.1 是最简单的 C#程序，其中只有一个语句。

1.2.1 语句和语句定界符

这个语句就是 `Console.WriteLine()`，它向控制台写入一行文本。**语句**包含代码将执行的一个或多个操作，而 C#通常使用分号标识语句的结束。语句的典型用途包括声明变量、控制程序流程和调用方法等。

> **高级主题：无分号的语句**
>
> C#中的许多编程元素都以分号结束。一个不需要分号的例子是 `switch` 语句。由于 `switch` 语句总是包含大括号，所以 C#不要求它后跟分号。实际上，代码块本身也被视为语句（它们也由语句组成），不要求以分号结尾。类似地，有的编程元素（比如 `using` 指令）虽然末尾有分号但不被视为语句。

由于换行符不会分隔语句，所以可以将多个语句放到同一行上，C#编译器认为该行包含包含多个指令。例如，代码清单 1.3 在同一行写了两个语句，会分两行显示“上”和“下”。

代码清单 1.3　同一行上的多个语句

```
Console.WriteLine("上"); Console.WriteLine("下");
```

当然，也可以每个语句一行，如代码清单 1.4 所示。

代码清单 1.4　每个语句一行

```
Console.WriteLine("上");
Console.WriteLine("中");
Console.WriteLine("下");
```

C#还允许将一个语句跨越多行。同样地，C#编译器会寻找分号来识别语句的结束。例如，在代码清单 1.5 中，HelloWorld 程序的原始 `WriteLine()`语句跨越了多行。

代码清单 1.5　一个语句跨越多行

```
Console.WriteLine(
    "你好，我叫 Inigo Montoya。");
```

1.2.2 认识类和方法

在到目前为止显示的代码清单中，语句都是独立于其他任何 C#构造的，并且仅以单个文件的方式呈现。毕竟，前面展示的只是最简单的 C#程序，即一个 HelloWorld 程序。然而，程序完全可能变得非常复杂，并且可以添加各种结构来组织代码。最简单的结构是添加方法，并将这些方法放在类中。代码清单 1.6 展示了一个例子。

代码清单 1.6　带有类和方法的 HelloWorld 程序

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("你好，我叫 Inigo Montoya。");
    }
}
```

在这个代码清单中，语句放到一个名为 Main 的方法中，后者又放到一个名为 Program 的类中。

那些有 Java，C 或 C++编程经验的人会立即看到相似之处。就像 Java 一样，C#也从 C 和 C++继承了基本语法。[1]对于有这些语言背景的程序员来说，语法标点符号（如分号和大括号）、某些特色（如区分大小写）以及所用的关键字（如 `class`，`public` 和 `void`）都是非常熟悉的。

> **注意：**C#是区分大小写的语言；大小写不正确，会使代码无法成功编译。

> **语言对比：Java——文件名必须匹配类名**
>
> Java 要求文件名与类名一致。C#虽然也常遵守这一约定，却并非必须。在 C#中，一

[1] C#语言设计者直接从 C/C++规范中拿掉了他们不喜欢的特性，保留并新增了他们喜欢的。开发组还有其他语言的资深专家。

> 个文件可以包含多个类；而且从 C# 2.0 开始，一个类的代码可通过所谓的**分部类**拆分到多个文件中。

> **初学者主题：关键字**
>
> 为了帮助编译器解释代码，C#中的某些单词具有特殊地位和含义，它们称为**关键字**。编译器根据关键字的固有语法来解释程序员写的表达式。在 HelloWorld 程序中，`class`，`static` 和 `void` 均是关键字。
>
> 编译器根据关键字识别代码的结构与组织方式。由于编译器对这些单词有着严格的解释，所以只能将关键字放在特定位置。违反规则的话，编译器会报错。

1.2.3 C#关键字

关键字是其他编程语言常见的另一种构造。表 1.1 总结了 C#关键字。

表 1.1 C#关键字

abstract	add*(1)	alias*(2)	and*
args*	as	ascending*(3)	async*(5)
await*(5)	base	bool	break
by*(3)	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
descending*(3)	do	double	dynamic*(4)
else	enum	equals*(3)	event
explicit	extern	false	file*
finally	fixed	float	for
foreach	from*(3)	get*(1)	global*(2)

goto	group*(3)	if	implicit
in	init*(9)	int	interface
internal	into*(3)	is	join*(3)
let*(3)	lock	long	nameof*(6)
namespace	new	nint*(9)	not*
notnull*(8)	null	nunit*(9)	object
on*(3)	operator	or*	orderby*(3)
out	override	params	partial*(2)
private	protected	public	readonly
record*	ref	remove*(1)	required*(11)
return	sbyte	scoped*	sealed
select*(3)	set*(1)	short	sizeof
stackalloc	static	string	struct
switch	this	throw	TRUE
try	typeof	uint	ulong
unchecked	unmanaged*(7.3)	unsafe	ushort

* 这些是上下文关键字，括号中的数字（n）代表加入该上下文关键字的 C#版本。

C# 1.0 之后没有引入任何新的保留关键字，但在后续版本中，一些构造使用了上下文关键字，它们只有在特定位置才有意义，其他位置则无意义。[1]通过这种方法，即使是 C# 1.0 的代码也与后来的标准兼容。

---

[1] 例如，在 C# 2.0 设计之初，语言设计者将 `yield` 指定成关键字。在微软发布的 C# 2.0 编译器的 alpha 版本中（该版本分发给了数千名开发人员），`yield` 以一个新关键字的身份存在。但语言设计者最终选择使用 `yield return` 而非 `yield`，从而避免将 `yield` 作为新关键字。除非与 `return` 连用，否则它没有任何特殊意义。

①

1.2.4 标识符

和其他语言一样，C#用**标识符**标识程序员编码的构造。在代码清单 1.6 中，HelloWorld 和 Main 均为标识符。我们以后用标识符来引用它所标识的构造，所以开发者应分配有意义的名称，不要随性而为。

> **注意**：好的程序员总能选择简洁而有意义的名称，这使代码更容易理解和重用。

清晰和一致是如此重要，以至于“框架设计准则”（*https://tinyurl.com/4ccecwys*）建议不要在标识符中使用单词缩写②，甚至不要使用不被广泛接受的首字母缩写词。即使被广泛接受（如 HTML），使用时也要一致。不要忽而这样用，忽而那样用。为避免滥用，可以限制所有首字母缩写词都必须包含到术语表中。总之，要选择清晰（甚至是详细）的名称，尤其是在团队中工作，或者开发要由别人使用的库的时候。

标识符有两种基本的大小写风格。第一种风格是.NET 框架创建者所谓的 Pascal 大小写（PascalCase），它在 Pascal 编程语言中很流行，要求标识符中每个单词的首字母大写，例如 `ComponentModel`，`Configuration` 和 `HttpFileCollection`。注意，在 `HttpFileCollection` 中，由于首字母缩写词 HTTP 的长度超过两个字母，所以仅首字母大写。第二种风格是 camel 大小写（camelCase），除了第一个字母小写，其他约定一样，例如 `quotient`，`firstName`，`httpFileCollection`，`ioStream` 和 `theDreadPirateRoberts`。

设计规范

DO favor clarity over brevity when naming identifiers.

要更注重标识符的清晰而不是简短。

---

① 偶尔也有不兼容的情况，比如 C# 2.0 要求为 `using` 语句提供的对象必须实现 `IDisposable` 接口，而不能只是实现 `Dispose()`方法。还有一些少见的泛型表达式，比如 `F(G<A,B>(7))`在 C# 1.0 中代表 `F((G<A),(B>7))`，而在 C# 2.0 中代表调用泛型方法 `G<A,B>`，传递实参 7，结果传给 F。

② 译注：有两种单词缩写，一种是“Abbreviation”，比如 Professor 缩写为 Prof.；另一种是“Contraction”，比如 Doctor 缩写为 Dr.。

DO NOT use abbreviations or contractions within identifier names.

不要在标识符名称中使用单词缩写。

DO NOT use any acronyms unless they are widely accepted, and even then, only when necessary.

不要使用不被广泛接受的首字母缩写词，即使被广泛接受，非必要也不要用。

下划线虽然合法，但标识符一般不要包含下划线、连字号或其他非字母/数字字符。此外，C#不像其前辈那样使用匈牙利命名法（为名称附加类型缩写前缀）。这避免了数据类型改变时还要重命名变量，也避免了数据类型前缀经常不一致的情况。

极少数情况下，有的标识符（比如 `Main`）可能在 C#语言中具有特殊含义。

虽然命名规范看起来比较琐碎，特别是在那些有其他语言背景的人的眼中。在那些语言中，这方面的规范可能非常模糊，甚至可能完全缺失。但是，如果违反这些规范，有经验的 C#和.NET 程序员会觉得非常刺眼，会本能地觉得代码质量低或程序员经验不足。为了避免这种情况，请学习并严格遵循这些规范。

设计规范

DO capitalize both characters in two-character acronyms, except for the first word of a camelCased identifier.

要把两个字母的首字母缩写词全部大写，除非它是 camelCase 标识符的第一个单词。

DO capitalize only the first character in acronyms with three or more characters, except for the first word of a camelCased identifier.

包含三个或更多字母的首字母缩写词，仅第一个字母才**要**大写，除非该缩写词是 camelCase 标识符的第一个单词。

DO NOT capitalize any of the characters in acronyms at the beginning of a camelCased identifier.

在 camelCase 标识符最开头的部分，首字母缩写词中的所有字母都**不要**大写。

DO NOT use Hungarian notation (that is, do not encode the type of a variable in its name).

不要使用匈牙利命名法（也就是说，不要为变量名称附加类型前缀）。

> **高级主题：关键字**
>
> 虽然罕见，但关键字附加“@”前缀可以作为标识符使用。例如，可以命名局部变量`@return`。类似地（虽然不符合C#大小写规范），可以命名方法`@throw()`。
>
> 在微软的实现中，还有 4 个未文档化的保留关键字：`__arglist`，`__makeref`，`__reftype`，和`__refvalue`。它们仅在罕见的互操作情形下才需要使用，平时完全可以忽略。注意，这 4 个特殊关键字均以双下划线开头。C#设计者保留将来把这种标识符转化为关键字的权利。为安全起见，自己不要创建这样的标识符。

1.2.5 类型定义

类定义是`class <`*标识符*`> { ... }`形式的一个区域。代码清单 1.7 展示了一个例子，其中的*标识符*是`HelloWorld`。

代码清单 1.7　基本的类声明

```
public class HelloWorld
{
    // ...
}
```

类型名称（本例是`HelloWorld`）可以随便取，但根据约定，它应当使用 PascalCase 大小写风格。就本例来说，可以选择的名称包括`Greetings`、`HelloInigoMontoya`、`Hello`或者简单地称为`Program`。注意，对于包含`Main()`方法的类，`Program`是一个很好的名称。`Main()`方法的详情稍后讲述。

设计规范

DO name classes with nouns or noun phrases.

要用名词或名词短语命名类。

DO use PascalCasing for all class names.

要为所有类名使用 PascalCase 大小写风格。

程序通常包含多个类型，每个类型又包含多个方法。

> **初学者主题：什么是方法？**
>
> 从语法上说，C#方法是已命名（具名）的代码块，由一个方法声明（例如 `static void Main()`）引入，后跟一对大括号（{}），其中包含零个或多个语句。方法可以执行计算和/或操作。与书面语言中的段落相似，方法提供了结构化和组织代码的一种方式，使之更易读。更重要的是，方法可以重用，可以从多个地方调用，所以避免了代码的重复。方法声明除了引入方法并定义方法名，还要定义传入和传出方法的数据。在代码清单 1.8 中，`Main()`连同后面的`{ ... }`便是 C#方法的例子。

1.2.6 Main 方法

`Main` 方法是 C#程序的**入口点**：换言之，C#程序从 `Main` 方法开始执行。该方法以 `static void Main()`开头。在命令控制台中输入 `dotnet run` 来运行程序，程序将启动并解析 `Main` 方法的位置，然后执行其中第一个语句。如代码清单 1.8 所示。

代码清单 1.8　分解 HelloWorld 程序

```
public class Program                // 类定义的开始
{
    public static void Main()       // 方法声明
    {                               // 方法实现的开始
        Console.WriteLine(          // 此语句跨越两行
            "你好，我叫 Inigo Montoya。");
    }                               // 方法实现的结束
}                                   // 类定义的结束
```

虽然 `Main` 方法声明可以有某种程度的变化，但关键字 `static` 和方法名 `Main` 始终都是需要的（参见“高级主题：`Main` 方法声明”）。

稍后就会讲解代码清单 1.8 中以`//`开头的注释。它们的作用是标识代码清单中的不同构造。

> **高级主题：Main 方法声明**
>
> C#要求 `Main` 方法返回 `void` 或 `int`，而且要么无参，要么接收一个字符串数组。代码清单 1.9 展示了 `Main` 方法的完整声明。其中，`args` 参数是用于接收命令行参数的一个

字符串数组。但是，数组第一个元素不是程序名称（这跟 C/C++不一样），而是紧接在它之后的第一个命令行参数。要获取执行程序所用的完整命令（包括程序名），你需要使用 System.Environment.CommandLine。

代码清单 1.9　带有参数和返回类型的 Main 方法

```
public static int Main(string[] args)
{
    // ...
}
```

Main()返回的 int 称为**状态码**，用于标识程序执行是否成功。返回非零值通常意味着错误。

从 C# 7.1 开始，还为 Main 方法增加了 async/await 支持。在这种情况下，返回的应该是基于 Task 的某个类型。

注意：与 C 风格的“前辈”不同，C#的 Main 方法名使用大写 M，和 C#的 PascalCase 命名约定一致。

将 Main 方法指定为 static，意味着这是“静态”方法，可以采用*类名.方法名*的形式调用。不指定 static，“运行时”还要先对类进行实例化（在内存中分配类的一个实例），然后才能调用该方法。第 6 章有一整节的内容都是讲述静态成员的。

Main()之前的 void 表明该方法不返回任何数据（将在第 2 章进一步解释）。

和 C/C++一样，C#也用大括号封闭一个构造（比如类或者方法）的主体。例如，Main 方法主体（方法体）就是用大括号封闭起来的。在本例中，方法的主体仅一个语句。

初学者主题：什么是空白？

空白（whitespace）是一个或多个连续的格式字符（比如制表符、空格和换行符）。删除单词间的所有空白肯定会造成歧义。删除引号字符串中的任何空白也会。

1.2.7 空白

分号使 C#编译器能忽略代码中的空白。除少数特殊情况，C#允许代码随意插入空白而不改变语义。在代码清单 1.8 和代码清单 1.9 中，在语句中或语句间可以随意换行，甚至可以删除换行，这对编译器最终创建的可执行文件没有任何影响。（但是，在双引号包围的字符串中不能随意换行。）

程序员经常利用空白对代码进行缩进来增强可读性，这称为**缩排**。来看看代码清单 1.10 和代码清单 1.11 展示的两个版本的 HelloWorld 程序。虽然这两个版本看起来和原始版本颇有不同，但 C#编译器认为它们在语义上是等价的。

代码清单 1.10　不缩进

```
public class Program
{
public static void Main()
{
Console.WriteLine("你好，Inigo Montoya。");
}
}
```

代码清单 1.11　删除空白

```
public class Program{public static void Main()
{Console.WriteLine("你好，Inigo Montoya。");}}
```

> **初学者主题：用空白格式化代码**
>
> 为了增强可读性，用空白对代码进行缩排很有必要。写代码时，要遵循制订好的编码标准和约定，以增强代码的可读性。
>
> 本书约定每个大括号都单独占一行，并缩进大括号内的代码。如果一对大括号嵌套了第二对大括号，那么第二对大括号中的代码也缩进。
>
> 这不是强制性的 C#标准，只是风格偏好。

1.3 使用变量

现在，你已经见识了最基本的 C#程序。接下来，让我们学习声明局部变量。变量声明后可以赋值，可以将值替换成新值，而且可以在计算和输出等操作中使用该变量。但是，和 Python 和 JavaScript 等语言不同，C#中的变量一经声明，数据类型就不能改变。在代码清单 1.12 中，`string max` 就是变量声明。

代码清单 1.12　变量声明和赋值

```
public class MiracleMax
```

```
{
    public static void Main()
    {
        string max;     // “string”标识数据类型，
                        // “max”是变量名称。
        max = "愉快地袭击古堡吧！";
        Console.WriteLine(max);
    }
}
```

> **初学者主题：局部变量**
>
> **变量**是一个存储位置的符号名称，程序以后可以对该存储位置进行赋值和修改。**局部**意味着变量在方法或代码块（一对大括号{}）内部声明，其作用域“局部”于当前代码块。所谓“声明变量”就是定义一个变量，你需要：
>
> 1. 指定变量要包含的数据的类型；
>
> 2. 为它分配标识符，即变量名。

1.3.1 数据类型

代码清单 1.12 声明了一个 string 类型的变量。本章使用的其他常用数据类型还有 int 和 char。

- int 是 C#的 32 位整型。
- char 是字符类型，长度 16 位，足以表示无代理项的 Unicode 字符[①]。

下一章将更详细地探讨这些以及其他常见的数据类型。

> **初学者主题：什么是数据类型?**
>
> **数据类型**（或对象类型）是具有相似特征和行为的所有个体的分类。例如，animal（动物）就是一个类型，它对具有动物特征（多细胞、具有运动能力等）的所有个体（猴子、野猪和鸭嘴兽等）进行了分类。类似地，在编程语言中，类型是被赋予了相似特征的一些个体的定义。

[①] 译注：某些语言的文字编码要用两个 16 位值表示。第一个代码值称为“高位代理项”（high surrogate），第二个称为“低位代理项”（low surrogate）。在代理项的帮助下，Unicode 可以表示 100 多万个不同的字符。美国和欧洲地区很少使用代理项，东亚各国则很常用。

1.3.2 变量声明

代码清单 1.12 中的 `string max` 是**变量声明**，它声明名为 `max` 的一个 `string` 变量。还可以在同一个语句中声明多个变量。为此，首先指定数据类型，然后用逗号分隔不同标识符即可，如代码清单 1.13 所示。

代码清单 1.13　在同一个语句中声明两个变量

```
string message1, message2;
```

由于声明多个变量的语句只允许指定一次数据类型，因此所有变量都具有同一类型。

C#变量名可以使用任何字母或下划线（_）作为开头，后跟任意数量的字母、数字以及/或者下划线。但根据约定，局部变量名采用 camelCase 大小写风格（除了第一个单词，其他每个单词的首字母大写），而且不包含下划线。

设计规范

DO use camelCasing for local variable names.

要为局部变量使用 camelCase 大小写风格。

1.3.3 变量赋值

声明局部变量后，在读取它之前必须先赋值。一个办法是使用=操作符，即**简单赋值操作符**。**操作符**是一种特殊符号，标识了代码要执行的操作[①]。代码清单 1.14 演示了如何利用赋值操作符指定 `miracleMax` 和 `valerie` 变量要指向的字符串值。

代码清单 1.14　更改变量值

```
public class StormingTheCastle
{
    public static void Main()
    {
```

[①] 译注：operator 在本书统一采用“操作符”的说法，而不是“运算符”。相应地，operand 是“操作数”，而不是“运算子”。

```
        string valerie;
        string miracleMax = "愉快地袭击古堡吧！";

        valerie = "你觉得可以吗?";

        Console.WriteLine(miracleMax);
        Console.WriteLine(valerie);

        miracleMax = "这需要奇迹。";
        Console.WriteLine(miracleMax);
    }
}
```

从这个例子可以看出，既可以在声明变量的同时赋值（比如变量 miracleMax），也可在声明后用另一个语句赋值（比如变量 valerie）。要赋的值必须放在赋值操作符右侧。

运行编译好的程序，将生成如输出 1.3 所示的结果。

输出 1.3

```
>dotnet run
愉快地袭击古堡吧！
你觉得可以吗?
这需要奇迹。
```

本例特意列出了 dotnet run 命令，以后不再重复，除非需要添加额外的参数来指定程序的运行方式。

C#要求局部变量在读取前“明确赋值”；换言之，编译器必须确定它的值。此外，赋值本身作为一种操作会返回一个值。所以，C#允许在同一语句中进行多个赋值操作，如代码清单 1.15 所示。

代码清单 1.15　赋值会返回值，而该值可用于再次赋值

```
public class StormingTheCastle
{
    public static void Main()
    {
        // ...
        string requirements, miracleMax;
        requirements = miracleMax = "这需要奇迹。";
        // ...
    }
}
```

1.3.4 使用变量

赋值后就能通过变量名来引用值。因此，在 Console.WriteLine(miracleMax)语句中使用

变量 miracleMax 时，程序在控制台上显示“愉快地袭击古堡吧!”，也就是 miracleMax 当前的值。更改 miracleMax 的值，并执行相同的 Console.WriteLine(miracleMax)语句，会显示 miracleMax 的新值，即“这需要奇迹。”

> **高级主题：字符串不可变**
>
> 所有 string 类型的数据，不管是不是字符串字面值（literal）[1]，都是不可变的（不可修改）。例如，无法将字符串"Come As You Are."改成"Come As You Age."。也就是说，不能修改变量最初引用的数据，只能重新赋值，让它指向内存中的新位置。

1.4 控制台输入和输出

本章已多次使用 Console.WriteLine 将文本输出到命令控制台（或称终端）。除了能输出数据，程序还需要能接收用户输入的数据。

1.4.1 从控制台获取输入

可以使用 Console.ReadLine()方法获取控制台输入的文本。它暂停程序执行并等待用户输入。用户按 Enter 键，程序继续。Console.ReadLine()方法的输出，也称为**返回值**，就是用户输入的文本字符串。代码清单 1.16 和输出 1.4 展示了一个例子。

代码清单 1.16 使用 Console.ReadLine()

```
public class HeyYou
{
    public static void Main()
    {
        string firstName;
        string lastName;

        Console.WriteLine("嘿，你！");

        Console.Write("请输入你的名字：");
        firstName = Console.ReadLine();

        Console.Write("请输入你的姓氏：");
        lastName = Console.ReadLine();
```

[1] 译注：字面值（literal）是直接在代码中输入的值，包括数字和字符串值。也称为直接量、字面量或文字常量。

```
        }
    }
```

输出 1.4

```
嘿，你！
请输入你的名字：Inigo
请输入你的姓氏：Montoya
```

在每条提示消息后，程序都用 Console.ReadLine()方法获取用户输入并赋给一个变量。在第二个 Console.ReadLine()赋值操作完成之后，firstName 引用值"Inigo"，而 lastName 引用值"Montoya"。

将 Console.ReadLine()的结果赋给变量时，如果遇到 CS8600 警告，即“将 null 文本或可能的 null 值转换为不可为 null 类型”，那么在第 2 章之前可以安全地忽略它。或者，可以在声明 firstName 和 lastName 变量时，使用 string?而不是 string 类型名称来解决此警告。另外，还可以在项目文件（*.csproj）的 PropertyGroup 元素中，将 Nullable 元素设为 disable（<Nullable>disable</Nullable>），从而完全禁止与可空性相关的警告。

> **高级主题：Console.Read()**
>
> 除了 Console.ReadLine()，我们还可以使用 Console.Read()方法读取用户输入。但是，后者从标准输入流中读取 *下一个字符*，并返回相应的整数编码（如果输入中文，那么每个汉字算一个字符）。如果没有更多字符可供读取，就返回-1。为了获取实际字符，需要先将整数转型为字符，如代码清单 1.17 所示。
>
> **代码清单 1.17　使用 Console.Read()**
>
> ```
> int readValue;
> char character;
> Console.Write("随意输入，按 Enter 键结束：");
>
> while (true)
> {
>     readValue = Console.Read();
>     if (readValue == 13) break; // 13 是 Enter 键的编码
>     character = (char)readValue;
>     Console.Write(character);
> }
> ```
>
> 注意，除非用户按 Enter 键，否则 Console.Read()方法不会读取并返回输入。按 Enter 键之前，不会对字符进行任何处理，即使用户已经输入了多个字符。

为了读取用户的输入，还可以使用自 C# 2.0 开始引入的 Console.ReadKey()方法。和

Console.Read()方法不同的是，它返回用户的单次按键输入。可以利用它来拦截用户的按键操作，并采取相应行动，例如验证按键，或者限制用户只能按数字键等。

1.4.2 将输出写入控制台

在之前的代码清单 1.16 中，注意我们是用 Console.Write()而不是 Console.WriteLine()方法提示用户输入名字和姓氏。Console.Write()方法不在输出文本后自动添加换行符，而是将当前光标位置保持在同一行上。这样，用户输入就会与提示内容处在同一行。代码清单 1.16 的输出清楚演示了 Console.Write()的效果。

下一步是将通过 Console.ReadLine()获取的用户输入写回控制台。在代码清单 1.18 中，程序在控制台上输出了用户的全名。但是，这段代码使用了 Console.WriteLine()的一个变体，利用了从 C# 6.0 开始引入的字符串插值功能。注意，在 Console.WriteLine 调用中为字符串字面值附加了$前缀。它表明使用了字符串插值。[1]输出 1.5 是对应的输出。

代码清单 1.18　使用字符串插值来格式化

```
public class HeyYou
{
    public static void Main()
    {
        string firstName;
        string lastName;

        Console.WriteLine("嘿，你！");

        Console.Write("请输入你的名字：");
        firstName = Console.ReadLine();

        Console.Write("请输入你的姓氏：");
        lastName = Console.ReadLine();

        Console.WriteLine(
            $"你的全名是{ firstName } { lastName }。");
    }
}
```

输出 1.5

```
嘿，你！
请输入你的名字：Inigo
请输入你的姓氏：Montoya
你的全名是 Inigo Montoya。
```

[1] 译注：文档中也称为“字符串内插”。

注意，代码清单 1.18 不是先用 Write 语句输出"你的全名是"，再用 Write 语句输出 firstName，再用第三个 Write 语句输出空格，最后用 WriteLine 语句输出 lastName。相反，是用 C# 6.0 的字符串插值功能一次性输出整个字符串。字符串中的大括号被解释成表达式。编译器会求值这些表达式，转换成字符串并插入当前位置。不需要单独执行多个代码段并将结果整合成字符串，该技术允许一个步骤完成全部操作，从而增强了代码的可读性。

C# 6.0 之前则采用不同的方式，称为**复合格式化**。它要求先提供**格式字符串**来定义输出格式，如代码清单 1.19 所示。

代码清单 1.19　使用复合格式化

```
public class HeyYou
{
   public static void Main()
   {
      string firstName;
      string lastName;

      Console.WriteLine("嘿，你！");

      Console.Write("请输入你的名字：");
      firstName = Console.ReadLine();

      Console.Write("请输入你的姓氏：");
      lastName = Console.ReadLine();

      Console.WriteLine(
         "你的全名是{0} {1}。", firstName, lastName);
   }
}
```

本例使用的格式字符串是"你的全名是{0} {1}。"它为要在字符串中插入的数据标识了两个索引占位符。每个占位符都顺序对应格式字符串之后的实参。

注意，索引值是从零开始的。每个要插入的实参，或者称为**格式项**，按照与索引值对应的顺序排列在格式字符串之后。在本例中，由于 firstName 是紧接在格式字符串之后的第一个实参，所以它对应索引值 0。类似地，lastName 对应索引值 1。

注意，占位符在格式字符串中不一定按顺序出现。例如，代码清单 1.20 交换了两个索引占位符的位置，并添加了一个逗号，从而改变了（英文）姓名的显示方式（参见输出 1.6）。

代码清单 1.20　交换索引占位符和对应的变量

```
Console.WriteLine("你的全名是{1}，{0}。", firstName, lastName);
```

输出 1.6

```
嘿，你！
请输入你的名字：Inigo
请输入你的姓氏：Montoya
你的全名是 Montoya，Inigo。
```

占位符除了能在格式字符串中按任意顺序出现，同一占位符还能在一个格式字符串中多次使用。另外，还可以省略占位符。但是，每个占位符都必须有对应的实参。

> **注意：**由于自 C# 6.0 引入的字符串插值几乎肯定比复合格式化更容易理解，所以本书默认使用前者。

1.4.3 注释

本节修改代码清单 1.19 来添加**注释**。注释不会改变程序的执行，只是使代码变得更容易理解。代码清单 1.21 中展示了新代码，输出 1.7 是对应的输出。

代码清单 1.21　为代码添加注释

```
public class CommentSamples
{
    public static void Main()
    {
        string firstName;  // 用于存储名字的变量
        string lastName;   // 用于存储姓氏的变量

        Console.WriteLine("嘿，你！");

        Console.Write /* 不换行 */ ("请输入你的名字：");
        firstName = Console.ReadLine();

        Console.Write /* 不换行 */ ("请输入你的姓氏：");
        lastName = Console.ReadLine();

        /* 使用字符串插值在控制台上显示问候语*/
        Console.WriteLine($"你的全名是{ firstName } { lastName }。");
```

```
            // 这是程序清单
            // 的结尾
        }
    }
```

输出 1.7

```
嘿，你!
请输入你的名字: Inigo
请输入你的姓氏: Montoya
你的全名是 Inigo Montoya。
```

虽然插入了注释，但编译并执行，输出和以前是一样的。

程序员用注释来描述和解释自己写的代码，尤其是在语法本身难以理解的时候，或者是在另辟蹊径实现一个算法的时候。只有检查代码的程序员才需要看注释，编译器会忽略注释。因此，在生成的程序集中，看不到源代码中的注释的一丝踪影。

表 1.2 总结了 4 种不同的 C#注释。代码清单 1.21 使用了其中两种。

表 1.2 C#注释类型

注释类型	说明	示例
带分隔符的注释	正斜杠后跟一个星号，即/*，用于开始一条带分隔符的注释。结束注释是在星号后跟上一个正斜杠，即*/。这种形式的注释既可以在代码文件中跨越多行，也可嵌入一行代码中使用。如果星号出现在行首，同时又在/*和*/这两个分隔符之间，那么它们也是注释的一部分，仅用于对注释进行排版	/*注释*/
单行注释	注释也可以放在由两个连续的正斜杠构成的分隔符（//）之后。编译器将从这个分隔符开始到行末的所有文本视为注释。这种形式的注释只占一行。但可以连续使用多条单行注释，就像代码清单 1.21 最后的注释那样	//注释
XML 带分隔符的注释	以/**开头并以**/结尾的注释称为 XML 带分隔符的注释。它们具有与普通的带分隔符的注释一样的特征，只是编译器会注意到 XML 注释的存在，而且可	/**注释**/

	以把它们解析到一个单独的文本文件（API 文档）中。XML 带分隔符的注释是 C# 2.0 新增的，但它的语法完全与 C# 1.0 兼容	
XML 单行注释	XML 单行注释以///开头，并延续到行末。除此之外，编译器可将 XML 单行注释和 XML 带分隔符的注释一起存储到单独的文件中	///注释

第 10 章将更全面地讨论 XML 注释以及如何用它生成 API 文档。届时会讨论各种 XML 标记。

编程史上确实有一段时期，如果代码没有详尽的注释，都不好意思说自己是专业程序员。但时代变了。没有注释但可读性好的代码，比需要注释才能说清楚的代码更有价值。如果开发人员发现需要写注释才能说清楚一个代码块的功用，那么应考虑重构，而不是洋洋洒洒写一大堆注释。写注释来重复代码本身就讲得清楚的事情，只会变得臃肿，降低可读性，还容易过时，因为将来可能更改代码但没有来得及更新注释。

设计规范

DO NOT use comments unless they describe something that is not obvious to someone other than the developer who wrote the code.

不要使用注释，除非代码本身“一言难尽”，或者只有开发人员自己看得懂。

DO favor writing clearer code over entering comments to clarify a complicated algorithm.

要尽量写清楚的代码而不是通过注释来澄清复杂的算法。

> **初学者主题：XML**
>
> XML（Extensible Markup Language，可扩展标记语言）是一种简单而灵活的文本格式，常用于 Web 应用以及应用之间的数据交换。XML 之所以“可扩展”，是因为 XML 文档包含的是对数据进行描述的信息，也就是所谓的元数据（metadata）。下面展示了一个示例 XML 文件。
>
> ```
> <?xml version="1.0" encoding="utf-8" ?>
>
> <body>
> ```

```
        <book title="C# 12.0 本质论">
            <chapters>
                <chapter title="C#概述"/>
                <chapter title="数据类型"/>
                ...
            </chapters>
        </book>
    </body>
```

文件以 Header 元素开始，描述 XML 文件版本和字符编码方式。之后是一个主要的 book 元素。元素以尖括号中的单词开头，比如<body>。结束元素需要将同一单词放在尖括号中，并为单词添加正斜杠前缀，比如</body>。除了元素，XML 还支持属性。title="C# 12.0 本质论"就是 XML 属性的例子。注意，XML 文件包含了对数据（比如“C# 12.0 本质论 C#”、“数据类型”等）进行描述的元数据（书名、章名等）。这可能形成相当臃肿的文件，但优点是可以通过描述来帮助解释数据。

1.4.4 调试

IDE 最重要的特色之一就是对调试的支持。请在 Visual Studio 或 Visual Studio Code 中按以下步骤试验。

1. 打开代码清单 1.21，光标定位到最后有 Console.WriteLine 的那一行，按 F9 在该行激活断点。
2. 如果用的是 Visual Studio，那么选择“调试”|“开始调试”（F5）重新启动应用程序，但这次激活了调试功能。Visual Studio Code 与此相似，只是菜单变成了“运行”|“启动调试(F5)”。
3. 注意，会在断点所在行暂停执行。此时可将鼠标放到某个变量（例如 firstName）上观察它的值。
4. 还可以拖动左侧黄箭头将程序执行从当前行移动到方法内的另一行。
5. 要继续执行，选择“调试”|“继续”（F5）或者单击工具栏上的“继续”按钮。

要更多地了解调试，请访问与不同 IDE 对应的链接：

- Visual Studio：*https://learn.microsoft.com/visualstudio/debugger/debugger-feature-tour*
- Visual Studio 2022 for Mac：*https://learn.microsoft.com/visualstudio/mac/debugging*
- Visual Studio Code：*https://code.visualstudio.com/Docs/editor/debugging*

1.5 托管执行和 CLI

处理器不能直接解释程序集。.NET 程序集用的是另一种语言，即**公共中间语言**（Common

Intermediate Language，CIL），或称**中间语言**（IL）[1]。C#编译器将 C#源代码文件转换成中间语言。为了将 CIL 代码转换成处理器能理解的机器码，还要完成一个额外的步骤（通常在运行时进行）。该步骤涉及 C#程序执行的一个重要元素：VES（Virtual Execution System，虚拟执行系统）。VES 也称为“运行时”（runtime）[2]。它根据需要编译 CIL 代码，这个过程称为**即时编译**或 **JIT 编译**（just-in-time compilation）。如果代码在像“运行时”这样的一个“代理”的上下文中执行，那么就称为**托管代码**（managed code），在“运行时”的控制下执行的过程则称为**托管执行**（managed execution）。之所以称为“托管”，是因为“运行时”管理着诸如内存分配、安全性和 JIT 编译等方面，从而控制了主要的程序行为。执行时不需要“运行时”的代码则称为**本机代码**（native code）或**非托管代码**（unmanaged code）。

> **注意：**“运行时”既可能指“程序执行的时候”，也可能指“虚拟执行系统”（VES）。为明确起见，本书用“执行时”表示“程序执行的时候”，用“运行时”表示负责管理 C#程序执行的代理。

“运行时”规范包含在一个包容面更广的规范中，即 CLI（Common Language Infrastructure，公共语言基础结构）规范[3]。作为国际标准，CLI 包含了以下几方面的规范：

- VES 或“运行时”
- CIL
- 支持语言互操作性的类型系统，称为 CTS（Common Type System，公共类型系统）
- 关于如何编写可通过 CLI 兼容语言来访问的库的指导原则，这部分内容具体放在公共语言规范（Common Language Specification，CLS）中
- 使各种服务能被 CLI 识别的元数据（包括程序集的布局或文件格式规范）

在“运行时”执行引擎的上下文中运行，程序员不需要直接写代码就能使用几种服务和功能，其中包括：

---

[1] CIL 的第三种说法是 Microsoft IL (MSIL)。本书用 CIL 一词，因其是 CLI 标准所采纳的。C#程序员交流时常用 IL 一词，因为他们都假定 IL 是指 CIL 而不是其他中间语言。

[2] 译注：“运行时”（runtime）作为名词使用时一律添加引号。

[3] 参见 *The Common Language Infrastructure Annotated Standard*。（Addison-Wesley，2004）Miller, J.和 S. Ragsdale 著。

- **语言互操作性**：不同源语言间的互操作性。语言编译器将每种源语言转换成相同的中间语言（CIL）来实现这种互操作性。
- **类型安全**：检查类型间转换，确保兼容的类型才能相互转换。这有助于防范缓冲区溢出（这是造成安全漏洞的主要原因）。
- **代码访问安全性**：程序集开发者的代码有权在计算机上执行的证书。
- **垃圾回收**：一种内存管理机制，自动释放“运行时”为数据分配的空间。
- **平台可移植性**：同一程序集可在多种操作系统上运行。要实现这一点，一个显而易见的限制就是不能使用平台特有的库。所以平台依赖问题需单独解决。
- **BCL（基类库）**：提供开发者能（在所有.NET 框架中）依赖的大型代码库，使其不必亲自写这些代码。

> **注意**：本节只是简单介绍了 CLI，目的是让你熟悉 C#程序的执行环境。此外，本节还提及了本书后面会用到的一些术语。第 24 章会专门探讨 CLI 及其与 C#开发人员的关系。虽然那一章在本书的最后，但其内容实际并不依赖之前的任何一章。所以，要想多了解一下 CLI，随时都能直接翻到那一章。

CIL 和 ILDASM

前面说过，C#编译器将 C#代码转换成 CIL 代码而不是机器码。处理器只理解机器码，所以 CIL 代码必须先转换成机器码才能由处理器执行。可用 CIL 反汇编程序将程序集解构为 CIL 形式。通常使用微软特有的文件名 ILDASM 来称呼这种 CIL 反汇编程序（ILDASM 是 IL Disassembler 的简称），它能对程序集执行反汇编，将 C#编译器生成的 CIL 提取为文本。

.NET 程序集的反汇编结果比机器码更易理解。许多开发人员害怕即使别人没有拿到源代码，程序也容易被反汇编并曝光其算法。其实，无论是否基于 CLI，任何程序防止反编译唯一安全的方法就是禁止访问编译好的程序（例如只在网站上存放程序，不把它分发到用户机器）。但假如目的只是减小别人获得源代码的可能性，那么可以考虑使用一些混淆器（obfuscator）产品。这种产品会打开 IL 代码，转换成一种功能不变但更难理解的形式。这可以防止普通开发人员访问代码，使程序集难以被反编译成容易理解的代码。除非程序需要对算法进行非常高级的安全防护，否则混淆器足矣。

> **高级主题：HelloWorld.dll 的 CIL 输出**
>
> 在不同 CLI 实现中，使用 CIL 反汇编程序的命令也有所区别。具体指令可以参考 *http://itl.tc/ildasm*。代码清单 1.22 展示了为 HelloWorld 控制台程序（代码清单 1.1 展示的那个）运行 ILDASM 而生成的 CIL 代码。为了看到这个结果，请在命令行上执行 `ildasm HelloWorld.dll /out=`*输出文件名*。

代码清单 1.22　示例 CIL 输出

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.8.3928.0


// Metadata version: v4.0.30319
.assembly extern System.Runtime
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )// .?_....:
  .ver 8:0:0:0
}
.assembly extern System.Console
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )// .?_....:
  .ver 8:0:0:0
}
.assembly HelloWorld
{
  .custom instance void
[System.Runtime]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.
ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .custom instance void
[System.Runtime]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ct
or() = ( 01 00 01 00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx
63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )  // ceptionThrows.

  // --- 下列自定义特性会自动添加，不要取消注释 -------
  //  .custom instance void
[System.Runtime]System.Diagnostics.DebuggableAttribute::.ctor(valuetype
[System.Runtime]System.Diagnostics.DebuggableAttribute/DebuggingModes) =
    ( 01 00 07 01 00 00 00 00 )

  .custom instance void
 [System.Runtime]System.Runtime.Versioning.TargetFrameworkAttribute::.ctor(string)
= ( 01 00 18 2E 4E 45 54 43 6F 72 65 41 70 70 2C 56   // ....NETCoreApp,V
65 72 73 69 6F 6E 3D 76 38 2E 30 01 00 54 0E 14   // ersion=v8.0..T..
46 72 61 6D 65 77 6F 72 6B 44 69 73 70 6C 61 79   // FrameworkDisplay
4E 61 6D 65 08 2E 4E 45 54 20 38 2E 30 )// Name..NET 8.0
  .custom instance void
 [System.Runtime]System.Reflection.AssemblyCompanyAttribute::.ctor(string) =
 ( 01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void
 [System.Runtime]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) =
( 01 00 05 44 65 62 75 67 00 00 )    // ...Debug..
  .custom instance void
 [System.Runtime]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) =
( 01 00 07 31 2E 30 2E 30 2E 30 00 00 )   // ...1.0.0.0..
  .custom instance void
```

```
[System.Runtime]System.Reflection.AssemblyInformationalVersionAttribute::.ctor(str
ing) = ( 01 00 05 31 2E 30 2E 30 00 00 )    // ...1.0.0..
  .custom instance void
 [System.Runtime]System.Reflection.AssemblyProductAttribute::.ctor(string) =
( 01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .custom instance void
 [System.Runtime]System.Reflection.AssemblyTitleAttribute::.ctor(string) =
( 01 00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
  .hash algorithm 0x00008004
  .ver 1:0:0:0
}
.module HelloWorld.dll
// MVID: {C565720C-7ED0-4287-A721-A31748D11C11}
.custom instance void
[System.Runtime]System.Runtime.CompilerServices.RefSafetyRulesAttribute::.ctor(int
32) = ( 01 00 0B 00 00 00 00 00 )
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003  // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x05120000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit Program
  extends [System.Runtime]System.Object
{
  .custom instance void

[System.Runtime]System.Runtime.CompilerServices.CompilerGeneratedAttribute::.ctor(
) = ( 01 00 00 00 )
  .method private hidebysig static void  '<Main>$'(string[] args) cil managed
  {
    .entrypoint
    // 代码大小  12 (0xc)
    .maxstack  8
    IL_0000:  ldstr bytearray (60 4F 7D 59 0C FF 11 62 EB 53 49 00 6E 00 69 00   //
`O}Y...b.SI.n.i.
 67 00 6F 00 20 00 4D 00 6F 00 6E 00 74 00 6F 00   // g.o. .M.o.n.t.o.
 79 00 61 00 02 30 ) // y.a..0
    IL_0005:  call  void [System.Console]System.Console::WriteLine(string)
    IL_000a:  nop
    IL_000b:  ret
  } // end of method Program::'<Main>$'

  .method public hidebysig specialname rtspecialname
instance void  .ctor() cil managed
  {
```

```
    // 代码大小  8 (0x8)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call  instance void [System.Runtime]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  ret
  } // end of method Program::.ctor

} // end of class Program


// =============================================================

// ************ 反汇编完成 ***********************
// 警告: 创建了 Win32 资源文件 output.res
```

最开头是清单（manifest）信息。其中不仅包括被反编译的模块的全名（HelloWorld.dll），还包括它依赖的所有模块和程序集及其版本信息。

基于这样的一个 CIL 代码清单，最有趣的可能就是能相对比较容易地理解程序所做的事情，这比阅读并理解机器码（汇编程序）容易多了。上述代码出现了对`Console.WriteLine()`的显式引用。CIL 代码清单包含许多外围信息，但如果开发者想要理解 C#模块（或任何基于 CLI 的程序）的内部工作原理，但又拿不到源代码，只要作者没有使用混淆器，理解这样的 CIL 代码清单还是比较容易的。事实上，一些免费工具（比如 Red Gate Reflector，ILSpy，JustDecompile，dotPeek 和 CodeReflect）都能将 CIL 自动反编译成 C#。[①]

1.6 多个.NET 框架

本章之前说过，目前存在多种.NET 框架。微软的宗旨是在最大范围的操作系统和硬件平台上提供.NET 实现。表 1.3 列出了最主要的。

表 1.3 主要的.NET Framework 实现

实现	描述
.NET 6 和后续版本	从.NET 6 开始已经取消了 Core 后缀，以表示.NET Core 和.NET Framework 已基本统一。

[①] 译注：注意反汇编（disassemble）和反编译（decompile）的区别。反汇编得到的是汇编代码，反编译得到的是所用语言的源代码。

.NET Core	真正跨平台和开源的.NET 框架，为服务器和命令行应用提供了高度模块化的 API 集合。
Microsoft .NET Framework	最开始的.NET 框架，正逐渐被.NET 取代。
Xamarin	随着.NET 6.0 的发布，它实际已成为.NET 的遗留移动平台实现，可与 iOS 和 Android 一起使用，并支持从单一代码库开发移动应用，同时仍然可以访问本机平台 API。
Mono	最早的.NET 开源实现，是 Xamarin 和 Unity 的基础。如果是新的开发，那么用.NET Core 代替 Mono。
Unity	跨平台 2D/3D 游戏引擎，用于为游戏机、PC、移动设备和网站开发电子游戏。（Unity 引擎开创了投射到 Microsoft Hololens 增强现实的先河。）

除非特别注明，否则本书所有例子都是针对.NET 开发的。

> **注意：**本书都用“.NET 框架”指代.NET 实现所支持的框架。相反，用“Microsoft .NET Framework”指代只能在 Windows 上运行，最初由微软在 2001 年发布的.NET 框架实现。

1.6.1 应用程序编程接口

某个数据类型（比如 `Console`）提供的所有方法（常规地说是成员）定义了该类型的**应用程序编程接口**（Application Programming Interface，API）。API 定义软件如何与其他组件交互，所以它们不局限于单独一个数据类型，而是更通用。通常，是由一组数据类型的所有 API 结合起来，为某个组件集合创建 API。以.NET 为例，一个程序集中的所有类型（及其成员）构成了该程序集的 API。类似地，.NET 中的所有程序集构成了更大的 API。通常将这一组更大的 API 称为**框架**。所以，我们用“.NET 框架”一词指代由.NET 的所有程序集共同公开的 API。API 通常包含一组接口和协议（或指令），帮助你使用一系列组件进行编程。事实上，对于.NET 来说，协议本身就是.NET 程序集的执行规则。

1.6.2 C#和.NET 版本控制

在历史上，.NET 框架的开发生命周期并不总是与 C#语言一致，这造成底层.NET 框架和对应的 C#语言使用不同版本号。例如，如果使用 C# 12.0 编译器进行编译，那么默认情况下将针对.NET 8.0 进行编译。表 1.4 概述了 C#、.NET Framework/.NET Core/.NET 以及 Visual Studio 的版本对应关系。

表 1.4 C#和.NET 版本

版本	说明
C# 1.0 和.NET Framework 1.0/1.1（Visual Studio 2002 和 2003）	C#的第一个正式发行版本。微软的团队从无到有创造了一种语言，专门为.NET 编程提供支持
C# 2.0 和.NET Framework 2.0（Visual Studio 2005）	C#语言开始支持泛型，.NET Framework 2.0 新增了支持泛型的库
.NET Framework 3.0	新增一套 API 来支持分布式通信（Windows Communication Foundation，WCF）、富客户端表示（Windows Presentation Foundation，WPF）、工作流（Windows Workflow，WF）以及 Web 身份验证（Cardspaces）
C# 3.0 和.NET Framework 3.5（Visual Studio 2008）	增加对 LINQ 的支持，对集合编程 API 进行大幅改进。.NET Framework 3.5 对原有的 API 进行扩展以支持 LINQ
C# 4.0 和.NET Framework 4（Visual Studio 2010）	增加对动态类型的支持，对多线程编程 API 进行大幅改进，强调了多处理器/多核心支持。
C# 5.0 和.NET Framework 4.5（Visual Studio 2012）和 WinRT 集成	增加对异步方法调用的支持，同时不需要显式注册一个委托回调。框架的另一个改动是支持与 Windows Runtime（WinRT）的互操作性。
C# 6.0 和.NET Framework 4.6/NET Core 1.X（Visual Studio 2015）	增加字符串插值、空传播（空条件）成员访问、异常过滤器、字典初始化器和其他许多功能。

C# 7.0 和.NET Framework 4.7/NET Core 1.1/2.0（Visual Studio 2017）	增加元组、解构器、模式匹配、嵌套方法（本地函数）、返回引用（`return ref`）等功能。
C# 8.0 和.NET Framework 4.8/.NET Core 3.0	增加对可空引用类型、高级模式匹配、`using` 声明、静态本地函数、disposable `ref` struct[①]、`Range` 和索引以及异步流的支持（尽管.NET Framework 4.8 不支持最后两个）
C# 9.0 和.NET 5.0（大多数.NET Framework 功能都合并到.NET Core）	增加对记录、关系模式匹配、顶级语句、源生成器和函数指针的支持。
C# 10.0 和.NET 6.0	增加了一些简化措施，包括记录结构、全局 `using` 指令、文件范围命名空间和扩展的属性模式等。另外，新增了参数表达式等。
C# 11 和.NET 7.0	增加了对文件范围类型、泛型数学支持、结构的默认初始化、原始字符串字面值、泛型 attributes、列表模式、必需成员等的支持。
C# 12 和.NET 8.0	扩展了主构造函数的使用，不再局限于记录。可以使用 `using` 指令为任何类型创建别名。为 Lambda 表达式新增了可选参数。

自 C# 6.0 引入的最重要的一个框架功能或许就是对跨平台编译的支持。换言之，不仅能用 Windows 上运行的 Microsoft .NET Framework 编译，还能使用 Windows，Linux 和 macOS

[①] 译注：文档将 disposal 和 dispose 翻译成“释放”。将 disposable 翻译成“可释放”。这里解释一下为什么不赞成这个翻译。在英语中， 这个词的意思是“摆脱”或“除去”(get rid of)一个东西，尤其是在这个东西很难除去的情况下。之所以认为“释放”不恰当，除了和 release 一词冲突，还因为 dispose 强调了“清理”和“处置”，而且在完成(对象中包装的)资源的清理之后，对象占用的内存还暂时不会释放。所以，“dispose 一个对象”真正的意思是：清理或处置对象中包装的资源(比如它的字段引用的对象)，然后等着在一次垃圾回收之后回收该对象占用的托管堆内存(此时才释放)。为避免误解，本书保留了 dispose ，disposal 和 disposable 的原文。

上运行的.NET Core 实现来编译。这意味着同一个代码库可编译并执行在多个平台上运行的应用程序。自版本 5.0 开始重命名为.NET 的.NET Core 是一套完整的 SDK，包含了从.NET Compiler Platform（即“Roslyn”，本身在 Linux 和 macOS 上运行）到.NET“运行时”的一切，另外还提供了像 Dotnet 命令行实用程序（dotnet CLI，自 C# 7.0 引入）这样的工具。

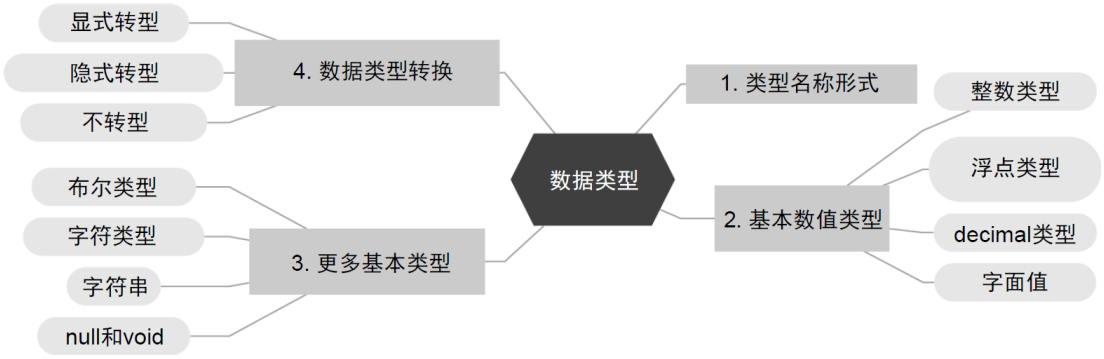
1.7 小结

本章对 C#进行了初步介绍。通过本章的学习，你熟悉了基本 C#语法。由于 C#与其他 C++风格的语言的相似性，所以本章许多内容可能都是你所熟悉的。但是，C#和托管代码确实有自己的一些独特性，比如会编译成 CIL 等。虽然并不特殊，但 C#的另一个关键特征是它完全面向对象。即使是在控制台上读取和写入数据这样的事情，也是面向对象的。面向对象是 C#的基础，这一点将贯穿全书。

下一章要探讨 C#的基本数据类型，并讨论如何将这些数据类型应用于操作数来构建表达式。

第 2 章 数据类型

从第 1 章的 HelloWorld 程序出发，你对 C#语言、它的结构、基本语法以及如何编写最简单的程序有了初步理解。本章将讨论基本 C#类型，继续巩固 C#的基础知识。



到目前为止，我们只用过少量语言内建的数据类型，而且只是一笔带过。C#有大量类型，而且可以合并类型来创建新类型。但是，C#有几种类型非常简单，是其他所有类型的基础，它们称为**预定义类型**（predefined type）或**基元类型**（primitive type）。C#语言的基元类型包括十种整数类型、两种用于科学计算的二进制浮点类型、一种用于金融计算的十进制浮点（`decimal`）类型、一种布尔类型以及一种字符类型。本章将探讨所有这些基元数据类型，还会更深入地研究第 1 章便已接触的 `string` 类型。

2.1 类型名称形式

所有内置类型基本上有三种名称形式。以 `string` 类型为例，它有完整名称（即 `System.String`）、隐含或简化名称（即 `String`）以及关键字（即 `string`）。

完整名称提供所有上下文，消除类型与其他所有类型的歧义。这是将 C#编译成的底层 CIL 代码所用的名称。完整名称由命名空间（完全限定类型名称最后一个句点之前的所有内容）后跟类型名称（名称最后一个句点之后的最后一部分）组成。如果不是为了提供一些方便，所有 C#类型都会通过其完整名称来引用。所有预定义类型本身都是基类库（BCL）的一部分。BCL 是指构成底层框架的 API 集合的名称。因此，包含在 BCL 中的类型的完整名称也是 BCL 名称。

刚才说的“方便”是指一些“捷径”，允许编译器隐式解析类型的命名空间，而不必在每个引用中都提供命名空间。一些类型具有很长的命名空间，例如 `System.Text.RegularExpressions.RegEx`。如果能避免命名空间限定符，只写一个 `RegEx`，那么必然能从中受益良多。有关这些快捷方式以及编译器如何推断命名空间的更多信息，

请参见第 5 章的“using 指令”一节。

关键字是 C#语言专门为预定义类型（基元类型）内置的“捷径”。不是预定义的类型没有关键字。例如，第 1 章的 Console 就只有完整名称（即 System.Console）和较简单的非限定名称（Console），后者的命名空间是隐含的。类似地，你定义的任何自定义类型（例如 Program）都没有关键字。

开发人员可以根据情况选择使用这三种名称形式。但是，不要变来变去。相反，最好坚持使用一种。虽然使用非限定名称与使用关键字似乎没有太大的区别，但 C#开发人员通常使用 C#关键字（如果可用）。例如，使用 string 而不是 String 或 System.String，使用 int 而不是 Int32 或 System.Int32。很少有必要在 C#代码中使用完全限定名称。而且，一般只是为了消除具有相同非限定名称的类型的歧义而使用，例如 System.Timers.Timer 和 System.Threading.Timer。

设计规范

DO use the C# keyword rather than the unqualified name when specifying a data type (for example, string rather than String).

要在指定数据类型时使用 C#关键字而不是非限定名称（例如，使用 string 而不是 String）。

DO favor consistency rather than variety within your code.

要一致而不要变来变去。

坚持一致可能和其他设计规范冲突。例如，虽然规范说要用 C#关键字取代完整名称，但有时需维护公司遗留下来的风格相反的文件（或文件库）。这时，最好与原风格保持一致，而不是强行引入新风格，造成和原来的约定不一致。但话又说回来，如原有“风格”实际是不好的编码实践，有可能造成 bug，会严重妨碍维护，那么还是应该尽量彻底纠正。

> **高级主题：using 指令和 global using 指令**
>
> 我们用于从命令行（控制台或终端）读取和写入内容的 Console 类的完整名称是 System.Console。在这个名称中，System 就是命名空间。命名空间以层次化的方式对类型进行分组。例如，通过正则表达式对文本进行解析的 RegEx 类位于 System.Text.RegularExpressions 命名空间。在这个命名空间的名称中，每个句点都代表层次结构中的不同级别。
>
> 例如，可以在文件顶部写一个 using 指令，例如 using System，从而避免为类型附加

命名空间前缀，因为编译器可以从 using 指令中推断出命名空间。

C# 10 还新增了对 global using 指令的支持，只需一次为整个项目添加 using 指令，而不必在每个使用命名空间中的文件中重复。要将 using 指令标识为全局，可以在 using 指令前加上 global，例如 global using System。

此外，C# 10 生成了一组默认的全局 using 语句，这称为隐式 using。在上一章的代码清单 1.2 中，我们为 ImplicitUsings 元素分配了 enable 值，它的作用就是启用此功能。一旦启用，所有常规 using 指令都会自动生成，包括 using System。具体地说，C#编译器会自动为项目生成一个.cs 文件，其中包含了一组全局 using 语句。

2.2 基本数值类型

C#基本数值类型都有关键字与之关联，包括整数类型、浮点类型以及 decimal 类型。decimal 是特殊的浮点类型，能存储无表示错误的很大的数字。

2.2.1 整数类型

C#支持 10 种整数类型，可以选择最恰当的一种来存储数据以避免浪费资源。表 2.1 总结了每种整型。

表 2.1 整数类型

类型	大小	范围（包括边界值）	BCL 名称	是否有符号	字面值后缀
sbyte	8 位	-128～127	System.SByte	是	
byte	8 位	0～255	System.Byte	否	
short	16 位	-32768～32767	System.Int16	是	
ushort	16 位	0～65535	System.UInt16	否	
int	32 位	-214483648～2147483647	System.Int32	是	
uint	32 位	0～4294967295	System.UInt32	否	U 或 u

long	64 位	9223372036854775808～9223372036854775807	System.Int64	是	L 或 l
ulong	64 位	0 ～ 18446744073709551615	System.UInt64	否	UL ，ul，LU 或 lu
nint	有符号 32 位或 64 位整数(视平台而定)	具体取决于代码在什么平台上执行。用 sizeof(nint)判断大小	System.IntPtr	是	
nuint	无符号 32 位或 64 位整数(视平台而定)	具体取决于代码在什么平台上执行。用 sizeof(nuint)判断大小	System.UIntPtr	否	

表 2.1（以及本节的其他表格）专门有一列给出了每种类型的完整名称；本章稍后会讲述字面值后缀问题。C#所有基元类型都有短名称和完整名称。完整名称对应 BCL（基类库）中的类型名称。该名称在所有语言中都一样，对程序集中的类型进行了唯一性标识。由于基元数据类型是其他类型的基础，所以 C#为基元数据类型的完整名称提供了短名称（或者称为缩写）。其实从编译器的角度看，两种名称完全一样，最终都生成相同的代码。事实上，检查最终生成的 CIL 代码，根本看不出源代码具体使用的名称。

> **语言对比：C++——short 数据类型**
>
> C/C++的 short 数据类型是 short int 的缩写。而 C#的 short 是一种实际存在的数据类型。

2.2.2 浮点类型(float 和 double)

浮点数精度可变。除非用分数表示时，分母恰好是 2 的整数次幂，否则用二进制浮点类型无法准确表示该数。[1]将浮点变量设为 0.1，很容易表示成 0.099 999 999 999 999 999 或者 0.100 000 000 000 000 000 1（或者其他非常接近 0.1 的数）。另外，像阿伏伽德罗常数这样

[1] 译注：例如 0.1，表示为分数是 1/10，分母 10 不是 2 的整数次幂，因此 1/10 不能用有限数量的二进制位来表示。

非常大的数字（6.02×10^{23}），即使出现误差达到 10^{8} 的表示错误，结果仍然非常接近 6.02×10^{23}，因为原始数字实在是太大了。根据定义，浮点数的精度与它所代表的数字的大小成正比。准确地说，浮点数精度由有效数位的个数决定，而不是由一个固定值（比如±0.01）决定。`double` 类型最多有 17 个有效数位，`float` 类型则最多有 9 个有效数位[①]（前提是数字不是从字符串转换而来的，参见稍后的“高级主题：浮点类型解析”）。[②]

C#支持表 2.2 所示的两种浮点数类型。二进制数转换成十进制数以便理解。虽然本书未提及其他一些浮点类型（具体可以访问 *https://learn.microsoft.com/dotnet/standard/numerics*），但它们不是内置类型，没有对应的关键字。

表 2.2 浮点类型

类型	大小	范围（包括边界值）	BCL 名称	有效数位	字面值后缀
`float`	32 位	± 1.5 × 10^{45} ～ ±3.4 x 10^{38}	`System.Single`	7	`F` 或 `f`
`double`	64 位	± 5.0 × 10^{324} ～ ±1.7 × 10^{308}	`System.Double`	15～16	`D` 或 `d`

> **高级主题：浮点类型解析**
>
> 只要在 `decimal` 类型的范围和精度限制内，任何十进制数都能完全准确地表示，不会

[①] 从.NET Core 3.0 起。另外要注意的是，所谓“有效数位”或“有效数字”（significant digits），并不是指小数位数（number of decimal digits），而是指一个数字（number）中具有意义的数位（digit），这包括小数点后的数位以及整数部分的数位（删除前导零）。本节所说的有效数位都是换算成十进制后的数位。

[②] 在.NET Core 3.0 之前，二进制数被转换为 15 个十进制数位，剩余部分则贡献给第 16 个十进制数位，如表 2.2 所示。具体地说，介于 1.7×10^{307} 和小于 1×10^{308} 之间的数字仅具有 15 个有效数位。然而，从 1×10^{308} 到 1.7×10^{308} 的数字具有 16 个有效数位。`decimal` 类型的有效数位范围与此相似。

> 出现表示错误。相反，如果用二进制浮点数表示十进制数，那么可能造成舍入错误。用任何有限数量的十进制数位表示 1/3 都无法做到精确。类似地，用任何有限数量的二进制数位表示 11/10，也无法做到精确（用二进制会表示成 1.0001100110011001101...）。两种情况都会产生某种形式的舍入错误。
>
> decimal 被表示成±N×10^k；其中 N 是 96 个二进制位的正整数，而-28 <= k <= 0。
>
> 与之相反，浮点数是±N×2^k 的任意数字。其中，N 是用固定数量的二进制位（float 是 24 位，double 是 53 位）表示的正整数，而 k 是-149～+104（float）或者-1075～+970（double）的任意整数

2.2.3 decimal 类型

C#还提供了 128 位精度的一种十进制浮点类型（参见表 2.3）。它适合大而精确的计算，尤其是金融计算。

表 2.3 decimal 类型

类型	大小	范围（包括边界值）	BCL 名称	有效数位	字面值后缀
decimal	128 位	± 1.0 × 10^{28} ～ ± 7.9228× 10^{28}	System.Decimal	28～29	M 或 m

和二进制浮点数不同，decimal 这种特殊的浮点数类型保证范围内的所有十进制数都是精确的。所以，对于 decimal 类型来说，0.1 就是 0.1，而不是近似值。不过，虽然 decimal 类型具有比浮点类型更高的精度，但它的范围较小。所以，从浮点类型转换为 decimal 类型可能发生溢出错误。此外，decimal 的计算速度稍慢（但差别不大，基本上可以忽略）。

> **高级主题： 本机大小的整数**
>
> C# 9.0 引入新的上下文关键字来表示本机大小（native-sized）的有符号和无符号整数，分别是 nint 和 nuint（参见表 2.4）。与其他数值类型不同，这两种整型的具体大小取决于代码在什么平台上执行。例如，nint 在 32 位平台上为 32 位，在 64 位平台上为 64 位。之所以设计这些类型，是为了与它们所在系统中的指针大小匹配。它们是一种更高级的类型，通常只在处理指针和底层操作系统的内存时才有用。在.NET 托管执

行上下文中处理内存时，它们是用不上的。

表 2.4 本机整数类型（具体 BCL 类型取决于上下文）

类型	大小	范围（包括边界值）	BCL 名称	有效数位	字面值后缀
nint	取决于操作系统	范围可变，但可以在运行时通过 nint.MinValue 和 nint.MaxValue 来确定	System.IntPtr	取决于操作系统	
nuint	取决于操作系统	范围可变，但可以在运行时通过 nuint.MinValue 和 nuint.MaxValue 来确定	System.UIntPtr	取决于操作系统	

请参见第 23 章更多地了解 nint 和 nuint。

2.2.4 字面值

字面值（literal value）表示源代码中的常量值。例如，如果希望用 Console.WriteLine() 输出整数值 42 和 double 值 1.618034（黄金分割比例），那么可以使用如代码清单 2.1 所示的代码，输出 2.1 展示了结果。

代码清单 2.1　指定字面值

```
Console.WriteLine(42);
Console.WriteLine(1.618034);
```

输出 2.1

```
42
1.618034
```

初学者主题：硬编码值的时候要慎重

直接将值放到源代码中称为**硬编码**（hardcoding），因为以后若是更改了值，就必须重新编译代码。因为可能会为维护带来不便，所以开发人员在硬编码值的时候必须慎重。例如，可以考虑从一个外部来源（如配置文件）中获取值。这样以后在值发生改变的时候，就不需要重新编译代码了。

默认情况下，输入带小数点的字面值，编译器会自动把它解释成 double 类型。相反，整数值（无小数点）通常默认为 int，前提是该值不是太大，以至于无法存储为一个 32 位整数。如果值太大，编译器会把它解释成 long。此外，C#编译器允许向非 int 的数值类型赋值，前提是字面值对于目标数据类型来说合法。例如，short s = 42 和 byte b = 77 都是允许的。但这一点仅对常量值成立。不使用额外的语法，b = s 就是非法的，具体参见稍后的“数据类型转换”一节。

前面说过，C#有许多数值类型。在代码清单 2.2 中，一个字面值被直接传给 WriteLine 方法。由于带小数点的值默认为 double，所以如输出 2.2 所示，结果是 1.618033988749895（原本最后两位是 48，现在变成 5），这符合我们对 double 精度的预期。

代码清单 2.2　指定 double 字面值

```
Console.WriteLine(1.6180339887498948);
```

输出 2.2

```
1.618033988749895
```

要显示具有完整精度的数字，必须将字面值显式声明为 decimal 类型，这是通过附加一个 M（或者 m）字面值后缀来实现的，如代码清单 2.3 和输出 2.3 所示。

代码清单 2.3　指定 decimal 字面值

```
Console.WriteLine(1.6180339887498948M);
```

输出 2.3

```
1.6180339887498948
```

代码清单 2.3 的输出符合预期：1.6180339887498948。注意后缀 d 表示 double，但为什么用 m 表示 decimal 呢？这是因为这种数据类型经常用于货币（monetary）计算。

可以使用 F（或 f）和 D（或 d）后缀，将字面值分别显式声明为 float 或者 double。对于整型，相应后缀是 U，L，LU 和 UL。整数字面值的类型是像下面这样确定的。

- 无后缀的数值字面值按以下顺序解析成能存储该值的第一个数据类型：int，uint，long，ulong。
- 后缀 U 的数值字面值按以下顺序解析成能存储该值的第一个数据类型：uint，ulong。
- 后缀 L 的数值字面值按以下顺序解析成能存储该值的第一个数据类型：long，ulong。

- 如果后缀是 UL 或 LU，那么直接解析成 ulong。

注意，字面值后缀不区分大小写。但一般推荐大写，以避免出现小写字母 l 和数字 1 不好区分的情况。

设计规范

DO use uppercase literal suffixes (e.g., 1.6180339887498948M).

要使用大写的字面值后缀（例如 1.6180339887498948M）

很大的数字较难辨识。为了解决可读性问题，C# 7.0 新增了对**数字分隔符**[1]的支持。如代码清单 2.4 所示，可以在书写数值字面值的时候用下划线（_）分隔。

代码清单 2.4　使用数字分隔符

```
Console.WriteLine(9_814_072_356M);
```

本例对数字进行千分位分隔，但这只是为了方便辨识，C#不要求必须这样做。可以在数字第一位和最后一位之间的任何位置添加分隔符。事实上，还可以连写多个下划线。

有的时候，可以考虑使用指数记数法，避免在小数点前后写许多个 0（不管是否使用了数字分隔符）。指数记数法要求使用 e 或 E 中缀，在中缀字母后面添加正整数或者负整数，并在字面值最后添加恰当的数据类型后缀。例如，可以将阿伏伽德罗常数作为 float 输出，如代码清单 2.5 和输出 2.4 所示。

代码清单 2.5　指数记数法

```
Console.WriteLine(6.023E23F);
```

输出 2.4

```
6.023E+23
```

[1] 译注：文档如此，“数字分隔符”（digit separator）实际应该是“数位分隔符”，用于对一个数字中的不同数位进行分隔。

> **初学者主题：十六进制记数法**
>
> 我们一般使用十进制记数法，即每个数位可用10个符号（0～9）表示。还可使用十六进制记数法，即每个数位可用 16 个符号表示：0～9，A～F（允许小写）。所以，0x000A 对应十进制值 10，而 0x002A 对应十进制值 42（2×16+10）。不过，实际的数是一样的。十六进制和十进制的相互转换不会改变数本身，改变的只是数的表示形式。
>
> 每个十六进制数位都用 4 个二进制位表示，所以一个字节可表示两个十六进制数位。

前面讨论数值字面值的时候，我们只使用了十进制值。C#还允许指定十六进制值。为值附加 0x 前缀，再添加希望使用的十六进制数字即可，如代码清单 2.6 和输出 2.5 所示。

代码清单 2.6　十六进制字面值

```
// 使用十六进制字面值显示值 42
Console.WriteLine(0x002A);
```

输出 2.5

```
42
```

注意，代码输出的仍然是 42，而不是 0x002A。

从 C# 7.0 起，还可以将数字表示成二进制值，如代码清单 2.7 所示。

代码清单 2.7　二进制字面值

```
// 使用二进制字面值显示值 42
Console.WriteLine(0b101010);
```

语法与十六进制语法相似，只是使用 0b 前缀（允许大写 B）。参见第 4 章的“初学者主题：位和字节”来了解二进制记数法以及二进制与十进制之间的转换。

注意，从 C# 7.2 起，数字分隔符可以放到代表十六进制的 x 或者代表二进制的 b 后面（称为前导数字分隔符），例如 0x_12_34_AB。

> **高级主题：将数字格式化成十六进制**

要显示数值的十六进制形式，必须使用 x 或 X 数值格式说明符。大小写决定了十六进制字母的大小写。代码清单 2.8 和输出 2.6 展示了一个例子。

代码清单 2.8　十六进制格式说明符的例子

```
// 显示"0x2A"
Console.WriteLine($"0x{42:X}");
```

输出 2.6

```
0x2A
```

注意，数值字面值（42）可以随便使用十进制或十六进制形式，结果一样。另外，在格式说明符与字符串插值表达式之间，要用冒号分隔。

高级主题：round-trip 格式化[①]

默认情况下，`Console.WriteLine(1.6180339887498948);`语句在执行后会显示 `1.618033988749895`，最后的 48 变成了 5。为了使 double 值在转换成字符串后能保持原样，可以使用格式字符串和 round-trip 格式说明符 R（或者 r）来转换它。例如，`Console.WriteLine($"{1.6180339887498948:R}");`会显示结果 `1.618033988749895`。

将 round-trip 格式说明符返回的字符串再转换回数值，肯定能获得原始值。所以如代码清单 2.9 和输出 2.7 所示，如果没有使用 round-trip 格式，两个数就不相等了。[②]

代码清单 2.9　使用 R 格式说明符进行格式化

```
const double number = 1.6180339887498948;
double result;
string text;

text = $"{number}";
result = double.Parse(text);
```

[①] 译注：在.NET 8.0 和 C# 12.0 中，可以忽略该“高级主题”。

[②] 译注：这个例子演示了如果没有使用 round-trip 格式，那么两个数就不相等了。但现在 round-trip 格式已经发生了改变，导致即使不使用 round-trip 格式，两个数也相等。这个改变来自开发团队的以下 pull request：Update the double/float formatters to return the shortest roundtrippable string。网址是 https://github.com/dotnet/coreclr/pull/22040。因此，这个例子暂时可以忽略。

```
Console.WriteLine($"{result == number}: result == number");
text = $"{number:R}";
result = double.Parse(text);
Console.WriteLine($"{result == number}: result == number");
```

输出 2.7

```
False: result == number
True: result == number
```

第一次为 text 赋值没有使用 R 格式说明符，所以 double.Parse(text)的返回值与原始数值不同。相反，在使用了 R 格式说明符之后，double.Parse(text)返回的就是原始值。

如果还不熟悉 C 风格语言的==语法，那么 result == number 在 result 等于 number 的前提下会返回 true，result != number 则相反。下一章将讨论赋值和相等性操作符。

2.3 更多基元类型

迄今为止，我们讨论的都是基元数值类型。C#还支持其他基元类型，包括 bool，char 和 string。

2.3.1 布尔类型(bool)

另一个 C#基元类型是布尔（Boolean）或条件类型 bool。它在条件语句和表达式中表示真或假，只允许 true 和 false 这两个值。bool 的 BCL 名称是 System.Boolean。例如，可以调用 string.Compare()方法并传递一个 bool 字面值 true，在不区分大小写的前提下比较两个字符串，如代码清单 2.10 所示。

代码清单 2.10　不区分大小写比较两个字符串

```
bool ignoreCase = true;
string option = "/help";
// ...
int comparison = string.Compare(option, "/Help", ignoreCase);
bool isHelpRequested = (comparison == 0);
// 显示: 是否请求帮助: True
Console.WriteLine($"是否请求帮助: {isHelpRequested}");
```

本例在不区分大小写的前提下将变量 option 的内容（"/help"）与字面值"/Help"进行比较，结果赋给 comparison（一个 int 值，为零表示比较的两个字符串相同）。

虽然理论上一个二进制位就足以容纳布尔类型的值，但 bool 实际是一个字节大小。

2.3.2 字符类型(char)

字符类型 char 表示 16 位字符，取值范围对应于 Unicode 字符集。从技术上说，char 的大小和 16 位无符号整数（ushort）一样，后者可以表示 0～65535 的值。但在 C#中，char 是一种特殊类型，在代码中也要特殊对待。

char 的 BCL 名称是 System.Char。

> **初学者主题：Unicode 标准**
>
> Unicode 是一个国际性标准，表示了大多数自然语言中的字符/字。它方便计算机系统构建本地化应用程序，为不同语言文化显示具有本地特色的字符/字。

> **高级主题：16 位不足以表示所有 Unicode 字符**
>
> 遗憾的是，不是所有 Unicode 字符都能用一个 16 位 char 表示。刚开始提出 Unicode 的概念时，它的设计者以为 16 位已经足够。但随着支持的语言越来越多，才发现当初的假定是错误的。结果是，一些 Unicode 字符要由一对称为“代理项”的 char 构成，总共 32 位（UTF-32）。

为了输入 char 字面值，需要将字符放到一对单引号中，例如'A'。所有 ANSI 键盘字符都可这样输入，包括字母、数字以及特殊符号。

但是，有的特殊字符不能直接插入源代码，需要进行特殊处理。为此，首先输入反斜杠（\）前缀，再跟随一个特殊字符代码。反斜杠和特殊字符代码统称为**转义序列**（escape sequence）。例如，\n 代表换行符，而\t 代表制表符。由于反斜杠标志转义序列开始，所以要用\\表示字面上的一个反斜杠字符。

代码清单 2.11　使用转义序列显示单引号

```
Console.WriteLine('\'');
```

表 2.5 总结了转义序列以及字符的 Unicode 编码。

表 2.5 转义字符

转义序列	字符名称	Unicode 编码
\'	单引号	\U0027
\"	双引号	\U0022
\\	反斜杠	\U005C
\0	Null	\U0000
\a	警报（系统响铃），a 是指 alert	\U0007
\b	退格	\U0008
\f	换页（Form Feed）	\U000C
\n	换行（Line Feed 或 newline）	\U000A
\r	回车（Enter）	\U000D
\t	水平制表符	\U0009
\v	垂直制表符	\U000B
\uxxxx	十六进制 Unicode 字符	\U0029
\x[n][n][n]n	十六进制 Unicode 字符（前三个占位符可选），\uxxxx 的长度可变版本	\U3A
\U00xxxxxx	Unicode 转义序列，用于创建代理项对（最大支持的序	\U00020100（𠀀）

	列是\U0010FFFF）	
\uxxxx\uxxxx	Unicode 转义序列，用于创建代理项对（支持\U0010FFFF 以上的序列）	\uD83D\uDE00（😀）

可用 Unicode 编码表示任何字符。为此，请为 Unicode 值附加\u 前缀。Unicode 字符可以使用十六进制记数法表示。例如，字母 A 除了用十进制 65 来表示，还可以表示成十六进制值 0x41。所以，无论执行 char letter=(char)65;还是执行 char letter=(char)0x41;语句，最终 letter 这个 char 变量的值都是'A'。

代码清单 2.12 显示两个 Unicode 字符来构成笑脸符号（:)），输出 2.8 展示了结果。

代码清单 2.12　使用 Unicode 编码显示笑脸符号

```
Console.Write('\u003A');
Console.WriteLine('\u0029');
```

输出 2.8

```
:)
```

2.3.3 字符串(string)

零或多个字符的有限序列称为**字符串**（string）。C#支持基元字符串类型 string，它的 BCL 名称是 System.String。对于已经熟悉了其他语言的开发者，string 的一些特点或许会出乎预料。除了第 1 章讨论的字符串字面值格式，还允许使用逐字前缀@，并允许用$前缀进行字符串插值。注意，string 是一种“不可变”（immutable）类型。从 C# 11 开始，语言还增加了对**原始字符串字面值**的支持。

字面值

为了将字面值字符串输入代码，要将文本放入一对双引号（""）内，就像 HelloWorld 程序中那样。字符串由字符构成，所以可以在字符串内嵌入字符的转义序列。

例如，代码清单 2.13 显示两行文本[①]。但这里没有使用 Console.WriteLine()，而是使用 Console.Write()来输出换行符\n。输出 2.9 展示了结果。记住，WriteLine()会在行末自

[①] 对注：本例的对白出自电影《公主新娘》。

动添加换行符，而 Write()不会。

代码清单 2.13　用字符\n 插入换行符

```
Console.Write("\"真的，你有令人混乱的智慧。\"");
Console.Write("\n\"等我做选择吧！\"\n");
```

输出 2.9

```
"真的，你有令人混乱的智慧。"
"等我做选择吧！"
```

注意，要显示的双引号也进行了转义（\"），否则会被视为定义字符串的开始与结束。

C#允许在字符串前使用@符号，指明转义序列不被处理。结果是一个**逐字字符串字面值**（verbatim string literal），它不仅将反斜杠当作普通字符，还会逐字解释所有空白字符。例如，代码清单 2.14 的三角形会在控制台上原样输出，其中包括反斜杠、换行符和缩进。输出 2.10 展示了结果。

代码清单 2.14　使用逐字字符串字面值来显示三角形

```
Console.Write(@"开始
                    /\
                   /  \
                  /    \
                 /      \
                /________\
结束");
```

输出 2.10

```
开始
                    /\
                   /  \
                  /    \
                 /      \
                /________\
结束
```

不使用@字符，这些代码甚至无法通过编译。事实上，即便将形状变成正方形，避免使用反斜杠，代码仍然不能通过编译，因为不能将换行符直接插入不以@符号开头的字符串中。

以@开头的逐字字符串唯一支持的转义序列是""，代表一个双引号，它不会终止字符串。

> **语言对比：C++ ——在编译时连接字符串**
>
> 和 C++不同，C#不自动连接（或串接、拼接，即 concatenate）字符串字面值。例如，不能像下面这样指定字符串字面值：
>
> ```
> string s = "Hello World!" " Hello World!";
> ```
>
> 相反，必须用+操作符连接。但是，如果编译器能在编译时计算结果，那么最终的 CIL 代码包含的就是一个连接好的字符串。

如果同一字符串字面值在程序集中多次出现，并赋给不同的变量，那么编译器在程序集中只定义字符串一次，而且所有变量都指向这个字符串实例。因此，如果在代码中多处插入包含大量字符的同一个字符串字面值，那么生成的程序集只会存储它的一个实例。

> **高级主题：UTF-8 字符串字面值**
>
> 本章前面提到，.NET 中的字符都是 Unicode 编码。具体而言，它们都是 UTF-16，每个字符（或符号）需要两个字节。因此，由字符构成的字符串也都是 Unicode（UTF-16）。
>
> 但是，在处理外部定义的格式时，偶尔需要使用 UTF-8 字符串。C# 11 允许在字符串的结束引号后使用后缀来指定 UTF-8 字符串字面值——`u8`。然而，这种字面值的数据类型就不是.NET 字符串了（`System.String`）。相反，数据类型是泛型`ReadOnlySpan<byte>`类型（参见 17.2.9 节）。下例展示了这样一个字符串的声明和赋值。
>
> ```
> ReadOnlySpan<byte> rates = "€汇率"u8;
> ```

字符串插值

如第 1 章所述，从 C# 6.0 起，允许在字符串中使用插值技术来嵌入表达式。语法是为字符串附加`$`前缀，并在字符串中用一对大括号嵌入表达式。在代码清单 2.15 中，`firstName` 和 `lastName` 是插入的简单变量。事实上，任何有效的 C#表达式（返回单个值的表达式）都是允许的。

代码清单 2.15　字符串插值

```
Console.WriteLine($"你的全名是: {firstName} {lastName}。");
```

从 C# 11 开始，还可以在大括号之间插入换行符（参见代码清单 2.16）。而在 C# 11 之前，换行符只能在逐字字符串中使用（参见之前打印三角形的例子）。

代码清单 2.16　在字符串插值中使用换行符

```
Console.WriteLine($"你的全名是: {
    firstName} {lastName}。");
```

注意，逐字和插值技术可以组合使用，但要先指定`$`，再指定`@`（从 C# 8.0 开始无此顺序要求），如代码清单 2.17 所示。

代码清单 2.17　组合使用逐字和插值技术

```
Console.WriteLine($@"你的全名是:
{firstName} {lastName}。");
```

由于是逐字字符串，所以会按字符串在代码中的样子分两行输出。注意，在大括号中换行是起不到换行效果的。逐字字符串的问题在于，表达式外部的所有空白现在都有了意义。因此，为了避免字符串中多余的缩进影响输出效果，应注意把它们删除（换言之，字符串应顶格排版）。为了解决这个问题，C# 11 引入了原始字符串字面值。

原始字符串字面值

C# 11 引入了原始字符串字面值。与逐字字符串字面值相似，原始字符串字面值也允许直接嵌入任意文本，包括空格、换行符、额外的引号和其他特殊字符，而无需使用转义序列。但是，两者存在一些重要的区别。代码清单 2.18 演示了最简单的单行原始字符串字面值。

代码清单 2.18　单行原始字符串字面值[①]

```
Console.WriteLine(
"""妈妈说过，"生活就像一盒巧克力..." """);
```

在这个最简单的例子中，开始和结束引号都在同一行。

注意，不需要对字符串中嵌入的引号进行转义。然而，在"..巧克力..."后面我们添加了一个空格。这是因为如果字符串以三个引号开头，就不能以四个引号结束。结束引号的计数必须与开始引号的计数匹配。如果不匹配，编译器会报错。（避免额外空格的解决方案将在稍后的多行原始字符串字面值的例子中提供。）

原始字符串字面值之所以允许用三个以上的双引号来定界原始字符串字面值，是因为这样就可以在文本中插入三个（或更多）连续的双引号字面值。例如，在原始字符串字面值的开头和结尾使用五个双引号，就可以在文本中插入四个连续的双引号，而且它们会在字面意义上被解释为四个连续的双引号。

还可以在原始字符串字面值中插值，如代码清单 2.19 所示。

代码清单 2.19　含有插值的原始字符串字面值

```
Console.WriteLine($"""你好，我叫{firstName}。{firstName} {lastName}。""");
```

但是，这里要注意一个额外的复杂性。$前缀的数量决定了容纳表达式所需的大括号数量。

---

① 这个例子出自电影《阿甘正传》。

例如，连续两个$要求每个表达式都用两个大括号括起来。代码清单 2.20 和输出 2.11 展示了一个例子，它显示了通过插值技术返回的字符串"莫比乌斯"。但是，由于使用了两个$前缀，所以 name 表达式要用两个大括号括起来，导致最后只输出一对字面意义的大括号（代码中总共使用了三对大括号）。

此外，代码 2.20 还演示了原始字符串字面值的多行版本。

代码清单 2.20　使用原始字符串字面值显示莫比乌斯环

```
string name = "莫比乌斯";
Console.Write(
    $$"""
        开始
                  ____
                 /   /\
                /   /  \
               /   /   /\
              /   /   /  \
             /   /   /\   \
            /   /   /  \   \
           /   /   /\   \   \
          /   /   /  \   \   \
         /___/___/____\   \   \
        /   {{{name}}} \   \   \
       /________________\   \   \
       \                \   \  /
        \________________\___\/
        结束
        """);
```

输出 2.11

```
开始
          ____
         /   /\
        /   /  \
       /   /   /\
      /   /   /  \
     /   /   /\   \
    /   /   /  \   \
   /   /   /\   \   \
  /___/___/____\   \   \
 /  {莫比乌斯}   \   \   \
/________________\   \   \
\                \   \  /
 \________________\___\/
结束
```

注意，在代码清单 2.20 中，尽管代码进行了缩进，但输出没有缩进。对于原始字符串字面

值来说，它最后的三个引号至关重要，因为最左侧的所有空白都会被删除。因此，在这三个引号中，第一个引号决定了输出中最左边的字符列，除了空白之外，字符串中的所有内容均不得跑到它的左边，否则编译器会发出编译警告。

另外，对于多行原始字符串字面值，起始引号集后或结束双引号集之前不能有任何字符。因此，在起始$$"""之后或者结束"""之前，出现任何文本都会导致编译错误。

使用多行原始字符串字面值，可以避免文本中出现有问题的结束引号，如代码清单 2.21 中对 mamaSaid 变量的赋值所示。

代码清单 2.21　通过多行原始字符串字面值来简化赋值

```
string firstName = "Forest";

// 单行原始字符串字面值
string lastName = """Gump""";

// 带插值的单行原始字符串字面值
string greeting =
$"""你好，我是{firstName}。{firstName} {lastName}。""";

string proposal = "你想要一块巧克力吗？"
    + "我能吃掉上百万块巧克力。";

string mamaSaid = // 多行原始字符串字面值
    """
                妈妈说过，"生活就像一盒巧克力..."
                """;

string jsonDialogue =

    // 带插值的多行原始字符串字面值

    $$"""
            {
                "quote": {
                    "character": "The MAN",
                    "dialogue": "{{greeting}}"
                 },
                "description" : "她点点头，兴趣不大。他...",
                "quote": {
                    "character": "The MAN",
                    "dialogue": "{{proposal}}"
                 },
                 "description" : "她摇摇头\"不\" 他打开盒子...",
                 "quote": {
```

```
                "character": "The MAN",
                 "dialogue": "{{mamaSaid.Replace("\"", "\\\"")}}"
            }
        }
        """;

Console.WriteLine(jsonDialogue);
```

在代码清单 2.21 中，jsonDialogue 变量的文本实际是一种称为 JSON 的特殊文本格式。文本中包含一组以层次化方式组织的名称/值对，与 JavaScript 兼容。这种格式是在为 Web 编程的时候流行起来的。但是，它现在已经成为一种最常用的数据交换手段。对于 JSON，原始字符串字面值的支持尤其关键，因为它允许将 JSON 字符串直接放入 C#程序，而无需复杂且容易出错的转义序列。

在赋给 jsonDialogue 的原始字符串字面值中，反斜杠对于 C#来说没有特殊意义。但在 JSON 中，每个字符串，包括名称和值，都被引号括起来。因此，值中的一些特殊 JSON 文本（例如"）要用一个反斜杠进行转义。然而，由于整个字符串值中的文本都是原始字符串字面值，而反斜杠对 C#来说没有特殊意义，所以反斜杠现在能在 JSON 字符串中得到保留。在 C#代码中嵌入 JSON 文本的时候，这是一项巨大的优势（在表示正则表达式时，原始字符串字面值也很有用）。

除此之外，我们在 JSON 文本中嵌入了对 C# Replace()方法的调用。下一节会讲解常见的字符串方法。

> **高级主题：理解字符串插值的内部工作原理**
>
> 字符串插值是调用 string.Format()方法的语法糖。例如以下语句：
>
> ```
> Console.WriteLine(
> $"你的全名是{firstName} {lastName}。");
> ```
>
> 它会被转换成以下形式的 C#代码：
>
> ```
> object[] args = new object[] { firstName, lastName };
> Console.WriteLine(string.Format("你的全名是{0} {1}。", args));
> ```
>
> 这就和复合字符串一样实现了某种程度的本地化支持，而且确保不会引入潜在的安全风险（因为不会在编译后通过字符串注入代码）。不过，完整的本地化支持需要与资源文件配合，此时应直接使用复合字符串或 string.Format()。如果还是在运行时解析字符串中的表达式，那么字符串插值可能无法很好地工作。

字符串方法

与 Console 类型相似，string 类型也提供了几个方法来格式化、连接（拼接）和比较字符串。

表 2.6 中的 Format()方法具有与 Console.Write()和 Console.WriteLine()方法相似的行为。区别在于，string.Format()不是在控制台窗口中显示结果，而是作为一个表达式返回结果。当然，有了字符串插值后，用到 string.Format()的机会减少了很多（本地化时还是用得着）。但在幕后，如果字符串组合不是由字符串字面值构成的，那么字符串插值编译的 CIL 结果实际是常量（C# 11 引入）以及 string.Concat()和 string.Format()调用的一个组合。

表 2.6　string 的静态方法

方法	示例
`static string string.Format(string format, ...)`	`string text, firstName, lastName;` `//...` `text = string.Format("你的全名是{0} {1}.", firstName, lastName);` `// 显示："你的全名是<firstName> <lastName>。"` `Console.WriteLine(text);`
`static string string.Concat(string str0, string str1)`	`string text, firstName, lastName;` `//...` `text = string.Concat(firstName, lastName);` `// 显示"<firstName><lastName>"，注意` `// 名字和姓氏之间无空格` `Console.WriteLine(text);`
`static int string.Compare(string str0, string str1)`	`// 1.` `string option;` `//...` `// 区分大小写的字符串比较` `int result = string.Compare(option, "/help");` `// 显示:` `// 0(相等)` `// 负数(option < /help)` `// 正数(option > /help)` `Console.WriteLine(result);` `// 2.` `string option;` `//...` `// 不区分大小写的字符串比较`

	`int result = string.Compare(` `option, "/Help", true);` `// 0(相等)` `// 负数(option < /help)` `// 正数(option > /help)` `Console.WriteLine(result);`

表 2.6 列出的都是**静态**方法。这意味着为了调用方法，需要在方法名（例如 Concat）之前附加方法所在类型的名称（例如 string）。但正如本章稍后会讲到的那样，string 类还有一些**实例**方法。实例方法不以类型名作为前缀，而是以变量名（或者其他实例引用）作为前缀。表 2.7 列出了部分实例方法和例子。

表 2.7　string 的实例方法

方法	示例
`bool StartsWith(string value)` `bool EndsWith(string value)`	`string lastName;` `//...` `bool isPhd = lastName.EndsWith("Ph.D.");` `bool isDr = lastName.StartsWith("Dr.");`
`string ToLower()` `string ToUpper()`	`string severity = "warning";` `// severity字符串全部转换成大写` `Console.WriteLine(severity.ToUpper());`
`string Trim()` `string Trim(...)` `string TrimEnd()` `string TrimStart()`	`// 1. 删除首尾空白` `username = username.Trim();` `// 2.` `string text = "indiscriminate bulletin";` `// 删除首尾的'i'和'n'` `text = text.Trim("in".ToCharArray());` `// 显示：discriminate bullet` `Console.WriteLine(text);`
`string Replace(string oldValue, string newValue)`	`string filename;` `//...` `// 删除字符串中的所有?` `filename = filename.Replace("?", "");`

字符串格式化

无论使用 string.Format()还是字符串插值来构造复杂格式的字符串，都可以通过一组丰富而复杂的格式化模式来显示数字、日期、时间、时间段等。例如，给定 decimal 类型的

price 变量，那么 string.Format("{0,20:C2}", price)或等价的插值字符串$"{price, 20:C2}"会使用默认货币格式化规则将 decimal 值转换成字符串。即添加本地货币符号，小数点后四舍五入保留两位，整个字符串在 20 个字符的域内右对齐（要想左对齐，就为 20 添加负号。另外，宽度不够只好超出）。因篇幅有限，无法详细讨论所有可能的格式字符串，请在 MSDN 文档中查阅 string.Format()，获取格式字符串的完整列表（*http://tinyurl.com/yck8j5jn*）。

要在插值或格式化的字符串中添加字面意义的左右大括号（{和}），那么可以连写两个大括号来表示。例如，插值字符串$"{{ {price:C2} }}"将生成字符串"{ $1,234.56 }"。

换行符

输出换行所需的字符由操作系统决定。Microsoft Windows 的换行符是\r 和\n 这两个字符的组合，UNIX 则是单个\n（称为 line feed）。为了消除平台之间的不一致，一个办法是使用 Console.WriteLine()在最后自动输出一个空行。为了确保跨平台兼容性，另一个办法是用 Environment.NewLine 获取当前操作系统上实际使用的换行符。换言之，Console.WriteLine("Hello World") 等价于 Console.Write("Hello World" + Environment.NewLine)。注意，在 Windows 操作系统上，Console.WriteLine()和 Console.Write(Environment.NewLine) 等价于 Console.Write("\r\n") 而非 Console.Write("\n")。总之，要依赖System.WriteLine()和Environment.NewLine而不是\n 来确保跨平台兼容。

设计规范

DO rely on Console.WriteLine() and Environment.NewLine rather than \n to accommodate Windows-specific operating system idiosyncrasies with the same code that runs on Linux and macOS.

要依赖 Console.WriteLine()和 Environment.NewLine 而不是\n 来确保跨平台兼容。

> 高级主题：C#属性
>
> 下一节提到的 Length 成员实际不是方法，因为调用时没有使用圆括号。Length 是 string 类型的**属性**（property），C#语法允许像访问成员变量（在 C#中称为**字段**）那样访问属性。换言之，属性定义了称为赋值方法（setter）和取值方法（getter）的特殊方法，但允许用字段语法访问那些方法。

> 研究属性的底层 CIL 实现，会发现它实际编译成两个方法：set_<PropertyName>和get_<PropertyName>。但这两个方法不能直接从 C#代码中访问，只能通过 C#属性构造来访问。第 6 章会更详细地讨论属性。

字符串长度

判断字符串长度（字或字符的个数）可以使用 string 的 Length 成员。该成员是只读属性。不能设置，调用时也不需要任何参数。代码清单 2.22 演示了如何使用 Length 属性，输出 2.12 是结果。

代码清单 2.22　使用 string 的 Length 成员

```
public class PalindromeLength
{
    public static void Main()
    {
        string palindrome;
        Console.Write("输入一句回文：");
        palindrome = Console.ReadLine();
        Console.WriteLine(
        $"回文\"{palindrome}\"共有"
        + $" {palindrome.Length}个字。");
    }
}
```

输出 2.12

```
输入一句回文：菜油炒油菜
回文"菜油炒油菜"共有 5 个字。
```

字符串的长度不能直接设置，只能根据字符串中的字或字符数计算得到。此外，字符串长度不能更改，因为字符串**不可变**。

字符串不可变

string 类型的一个关键特征是它不可变（immutable）。可为 string 变量赋一个全新的值，但出于性能考虑，没有提供修改现有字符串内容的机制。所以，不可能在同一个内存位置将字符串中的字母全部转换为大写。只能在其他内存位置新建字符串，让它成为旧字符串大写字母版本，旧字符串在这个过程中不会被修改，如果没人引用它，会被垃圾回收。代码清单 2.23 和输出 2.13 展示了一个例子。

代码清单 2.23　错误；string 是不可变的

```
Console.Write("输入文本: ");
string text = Console.ReadLine();

// UNEXPECTED: text 并没有转换为全大写
text.ToUpper();

Console.WriteLine(text);
```

输出 2.13

```
输入文本: This is a test of the emergency broadcast system.
This is a test of the emergency broadcast system.
```

从表面上看，text.ToUpper()似乎应该将 text 中的字符转换成全大写形式。但是，由于 string 类型不可变，所以 text.ToUpper()不会进行这样的修改。相反，text.ToUpper()会返回修改后的一个新字符串。为了使用它，需要把它保存到一个变量中，或者直接传给 Console.WriteLine()。代码清单 2.24 给出了纠正后的代码，输出 2.14 是结果。

代码清单 2.24　正确的字符串处理

```
public class Uppercase
{
    public static void Main()
    {
        string text, uppercase;
        Console.Write("输入文本: ");
        text = Console.ReadLine();

        // 返回全大写的一个新字符串
        uppercase = text.ToUpper();
        Console.WriteLine(uppercase);
    }
}
```

输出 2.13

```
输入文本: This is a test of the emergency broadcast system.
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM.
```

如果忘记字符串不可变的特点，很容易会在使用其他字符串方法时犯下和代码清单 2.23 相似的错误。

要真正更改 text 中的值，将 ToUpper()的返回值赋回 text 即可。如下例所示：

```
text = text.ToUpper();
```

System.Text.StringBuilder

如果要对字符串进行大量修改，比如经历多个步骤来构造一个长字符串，那么可以考虑使用 System.Text.StringBuilder 类型而不是 string。StringBuilder 提供了 Append()，AppendFormat()，Insert()，Remove()和 Replace()等方法。虽然 string 也提供了其中一些方法，但两者的关键区别在于，StringBuilder 的这些方法会修改 StringBuilder 实例本身，而不是每次修改都返回一个新字符串。换言之，和 string 相反，StringBuilder 是**可变**的。[1]

2.3.4 null 和 void

两个与类型相关的额外关键字是 null 和 void。null 关键字代表空值，表示变量不引用任何有效对象。void 则指出这里不存在类型或者根本不存在任何值。

null

null 也可以作为“字面值”的一种类型使用。将 null 赋给字符串变量，表明变量为“空”，不指向任何位置。事实上，甚至可以检查变量是否引用了一个 null 值。

将 null 赋给变量和根本不赋值是不一样的概念。换言之，赋值了 null 的变量已设置，而未赋值的变量未设置。因此，使用未赋值的变量会造成编译时错误。

注意，将 null 赋给 string 变量和为变量赋值""（称为空串）也是不一样的概念。null 意味着变量无任何值，而""意味着变量有一个称为“空串”（empty string）的值。这种区分相当有用。例如，编程逻辑可以将值为 null 的 homePhoneNumber 解释成“家庭电话未知”，将值为""的 homePhoneNumber 解释成“无家庭电话”。

> **高级主题：可空修饰符**
>
> 代码清单 2.25 演示了如何通过在类型声明中添加可空修饰符——一个问号（?）——将 null 赋给 int 变量。
>
> **代码清单 2. 25　将 null 赋给整型变量**
>
> ```
> public static void Main()
> ```

[1] 译注：参见《CLR via C#》第 4 版，14.3.2 节，清华大学出版社。

```
    {
        int? age;
        //...
        // 清除 age 的值
        age = null;
        // ...
    }
```

在 C# 2.0 引入可空修饰符之前，我们无法将 null 赋给除了 string（一个引用类型）之外的所有类型（值类型）的变量。（有关值类型和引用类型的更多信息，请参见第 3 章。）

此外，在 C# 8.0 之前，引用类型（如 string）默认支持 null 赋值，因此，无法使用可空修饰符来修饰一个引用类型。由于 null 反正都可以隐式赋值，因此用可空修饰符来修饰它是多余的。

高级主题： 可用引用类型

在 C# 8.0 之前，由于所有引用类型都默认可空，所以不存在可空引用类型的概念。但从 C# 8.0 开始，这一行为变得可以配置，允许使用可用修饰符（?）将引用类型声明为可空，或者保留默认的非空配置（阻止显示关于可空性的警告）。这样一来，从 C# 8.0 开始就有了**可空引用类型**的概念。带有可空修饰符的引用类型变量是可空引用类型。在已经启用可空引用类型的前提下（在*.csproj 文件中设置），将 null 赋给没有可空修饰符的引用类型变量会出现警告。

到目前为止，本书唯一涉及到的引用类型就是 string。如果已配置引用类型支持可空修饰符，那么可以声明像下面这样的一个字符串变量：

```
string? homeNumber = null;
```

为了支持可空性，请确保.csproj 文件中的 Nullable 元素已设为 enable。

void“类型”

有的时候，C#语法要求指定数据类型但不传递任何数据。例如，对于方法无返回值的情况，C#允许在数据类型的位置放一个 void 关键字。HelloWorld 程序（代码清单 1.1）的 Main 方法声明就是一个例子。在返回类型的位置使用 void，意味着该方法不返回任何数据。与此同时，编译器也不要指望会有一个值。void 本质上不是数据类型，它只是指出没有数据返回这一事实。

语言对比：C++

> 无论 C++还是 C#，void 都有两个含义：一个是指出该方法不返回任何数据，另一个是代表指向未知类型的存储位置的一个指针。C++程序经常使用 void**这样的指针类型。C#也可以使用相同的语法来表示指向未知类型的存储位置的指针。但是，这种用法在 C#中比较罕见，一般仅在需要与非托管代码库进行互操作时才会用到。

2.4 数据类型转换

考虑到各种.NET 预定义了大量类型，加上代码也能定义几乎无限数量的类型，所以类型之间的相互转换至关重要。会造成转换的最常见操作就是**转型**或**强制类型转换**（casting）。[①]

来考虑将 long 值转换成 int 的情形。long 类型能容纳的最大值是 9 223 372 036 854 775 808，int 则是 2 147 483 647。所以，转换时可能丢失数据——long 值可能大于 int 能容纳的最大值。有可能造成数据丢失（因为大小和/或精度）或引发异常（因为转换失败）的任何转换都需要执行**显式转型**。相反，不会丢失数据，而且不会引发异常（无论操作数的类型是什么）的任何转换都可以进行**隐式转型**。

2.4.1 显式转型

C#允许用转型操作符来执行转型。为此，需要在一对圆括号中指定希望变量转换成的类型，表明你已知晓在发生显式转型时可能丢失精度和数据，或者可能造成异常。代码清单 2.26 将一个 long 转换成 int，并显式告诉系统尝试这个操作。

代码清单 2.26　显式转型示例

```
long longNumber = 50918309109;
int intNumber = (int)longNumber; // (int)是转型操作符
```

程序员使用转型操作符告诉编译器：“相信我，我知道自己正在干什么。我知道值能适应目标类型。”只有程序员像这样做出明确决断，编译器才允许转换。但是，这也可能只是程序员“一厢情愿”。执行显式转换时，如果数据未能成功转换，那么“运行时”还是会引发异常。所以，要由程序员负责确保数据成功转换，或提供错误处理代码来处理转换不

---

① 译注：文档中使用的“强制类型转换”过于繁琐，本书以后都简称为“转型”。

成功的情况。

> **高级主题：checked 和 unchecked 转换**
>
> C#提供了特殊关键字来标识代码块，指出假如目标数据类型太小，以至于容不下所赋的数据，那么会发生什么情况。默认情况下，容不下的数据在赋值时会悄悄地溢出。代码清单 2.27 和输出 2.15 展示了一个例子。
>
> **代码清单 2.27　整数值溢出**
>
> ```
> // int.MaxValue 等于 2147483647
> int n = int.MaxValue;
> n = n + 1;
> Console.WriteLine(n);
> ```
>
> **输出 2.15**
>
> ```
> -2147483648
> ```
>
> 代码清单 2.27 向控制台写入值-2147483648，假装它完成了一次正确的计算。但是，将上述代码放到一个 checked 块中，或者在编译时使用 checked 选项，这个计算就会使“运行时”抛出 System.OverflowException 异常。代码清单 2.28 给出了 checked 块的语法，输出 2.16 展示了结果。
>
> **代码清单 2.28　checked 块示例**
>
> ```
> checked
> {
>     // int.MaxValue 等于 2147483647
>     int n = int.MaxValue;
>     n = n + 1;
>     Console.WriteLine(n);
> }
> ```
>
> **输出 2.16**
>
> ```
> 未处理的异常： System.OverflowException: 算术运算导致溢出。
>    在 Program.Main() 位置...Program.cs:行号 12
> ```
>
> checked 块的代码如果在运行时发生赋值溢出，那么会抛出异常。要将默认的不检查溢出更改为检查溢出，请在.csproj 文件中添加以下元素：
>
> ```
> <CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
> ```
>
> 此外，C#还支持用一个 unchecked 块来强制不执行溢出检查，块中溢出的赋值不会引

发异常，如代码清单 2.29 和输出 2.17 所示。

代码清单 2. 29　unchecked 块示例

```
unchecked
{
    // int.MaxValue 等于 2147483647
    int n = int.MaxValue;
    n = n + 1;
    Console.WriteLine(n);
}
```

输出 2. 17

```
-2147483648
```

即使编译时开启了 checked 选项，上述代码中的 unchecked 关键字也会阻止“运行时”抛出异常。

读者可能会觉得奇怪，在不检查溢出的前提下，在 int.MaxValue 上加 1 为什么会得到 -2147483648。这是二进制的回绕（wrap around）语义造成的。int.MaxValue 的二进制形式是 01111111111111111111111111111111，第一位（0）表示这是正值。递增该值触发回绕，下个值回到了 10000000000000000000000000000000，即最小的整数（int.MinValue），其中，第一位（1）表示这是负值。在 int.MinValue 上加 1 会得到 10000000000000000000000000000001（-2147483647）并如此反复。

转型操作符不是万能药，不能用它将一种类型任意转换为其他类型。编译器仍会检查转型操作的有效性。例如，long 不能转换成 bool。因为并没有定义这样的转换，所以编译器不允许。

语言对比：数值转换成布尔值

一些人可能觉得奇怪，C#居然不存在从数值类型到布尔类型的有效转型，因为这在其他许多语言中是很普遍的。C#之所以不支持这样的转换，是为了避免可能发生的歧义。例如，-1 到底对应 true 还是 false？更重要的是，如下一章要讲到的那样，这还有助于避免用户在本应使用相等操作符的时候使用了赋值操作符。例如，这样可以避免在本该写 if(x == 42){...}的时候却写了 if(x = 42){...}，编译器会对后者报错。

2.4.2 隐式转型

在某些情况下，比如从 int 类型转换成 long 类型时，不会发生精度的丢失，而且值不会

发生根本性的改变，所以代码只需指定赋值操作符，转换将**隐式**地发生。换言之，编译器判断这样的转换能正常完成。代码清单 2.30 直接使用赋值操作符实现从 int 到 long 的转换。

代码清单 2.30　隐式转型无需使用转型操作符

```
int intNumber = 31416;
long longNumber = intNumber;
```

如果愿意，在本来可以隐式转型的时候，也可以强行添加转型操作符，如代码清单 2.31 所示。

代码清单 2.31　隐式转型也使用转型操作符

```
int intNumber = 31416;
long longNumber = (long)intNumber;
```

2.4.3 不使用转型操作符的类型转换

由于未定义从字符串到数值类型的转换，因此需要使用像 Parse()这样的方法。每种数值数据类型都提供了一个 Parse()方法，允许将字符串转换成对应的数值类型。如代码清单 2.32 所示。

代码清单 2.32　使用 float.Parse()将 string 转换为数值类型

```
string text = $"{9.11E-31}";
float kgElectronMass = float.Parse(text);
```

还可以利用特殊类型 System.Convert 将一种类型转换成另一种。如代码清单 2.33 所示。

代码清单 2.33　使用 System.Convert 进行类型转换

```
string middleCText = $"{261.626}";
double middleC = Convert.ToDouble(middleCText);
bool boolean = Convert.ToBoolean(middleC);
```

但是，System.Convert 只支持少量类型，且不可扩展，允许从 bool，char，sbyte，short，int，long，ushort，uint，ulong，float，double，decimal，DateTime 和 string 转换到这些类型中的其他任何一种。

此外，所有类型都支持 ToString()方法，可以用它获得类型的字符串表示。代码清单 2.34 演示了如何使用该方法，输出 2.18 展示了结果。

代码清单 2.34　使用 ToString()转换为一个 string

```
bool boolean = true;
string text = boolean.ToString();
// 显示"True"
Console.WriteLine(text);
```

输出 2.18

```
True
```

大多数类型的 ToString()方法只是返回数据类型的名称，而不是数据的字符串表示[①]。只有在类型显式实现了 ToString()的前提下才会返回字符串表示。最后要注意，完全可以编写自定义的转换方法，“运行时”的许多类都存在这样的方法。

> **高级主题：TryParse()**
>
> 从 C# 2.0（.NET 2.0）起，所有基元数值类型都包含静态 TryParse()方法。该方法与 Parse()非常相似，只是转换失败不是引发异常，而是返回 false，如代码清单 2.35 和输出 2.19 所示。
>
> 代码清单 2.35　用 TryParse()代替引发异常
>
> ```
> double number; // 老版本 C#要求变量先声明才能作为 out 参数使用
> string input;
>
> Console.Write("输入一个数字: ");
> input = Console.ReadLine();
> if (double.TryParse(input, out number))
> {
>     // 转换正确，现在开始使用数字
>     // ...
> }
> else
> {
>     Console.WriteLine(
>     "输入的文本不是一个有效的数字。");
> }
> ```

[①] 译注：继承自终极基类 Object 的实现。

输出 2.19

```
输入一个数字：四十二
输入的文本不是一个有效的数字。
```

上述代码从输入字符串解析到的值通过一个 out 参数（本例是 number）返回。第 5 章的“输出参数(out)”一节会更详细地讨论 out 参数。

除了各种数值类型，枚举（enum）也支持 TryParse()方法。

注意，从 C# 7.0 开始，准备作为 out 参数使用的变量不用事先声明了。代码清单 2.36 展示了修改后的代码。

代码清单 2.36　TryParse()的 out 参数声明在 C# 7.0 中可以内联了

```
// 现在，作为 out 参数使用的变量不需要事先声明
// double number;
string input;
Console.Write("输入一个数字：");
input = Console.ReadLine();
if (double.TryParse(input, out double number))
{
    Console.WriteLine(
        $"输入被成功解析成数字：{number}.");
}
else
{
    // 注意：number 的作用域也延伸到这里(虽然未赋值)
    Console.WriteLine(
        "输入的文本不是一个有效的数字。");
}
Console.WriteLine(
    $"'number'目前的值是：{number}");
```

注意，是先写 out 再写数据类型。像这样定义的 number 变量除了具有 if 语句内部真假条件块中的作用域，在外部也可以使用（参见本例最后一个 Console.WriteLine 语句）。

Parse()和 TryParse()的关键区别在于，如果转换失败，那么 TryParse()不会抛出异常。string 到数值类型的转换是否成功，往往要取决于输入文本的用户。用户完全可能输入无法成功解析的数据。使用 TryParse()而不是 Parse()，就可以避免在这种情况下引发异常（我们已预见到用户会输入无效数据，而针对预见到的情况，我们应尽量避免抛出异常）。

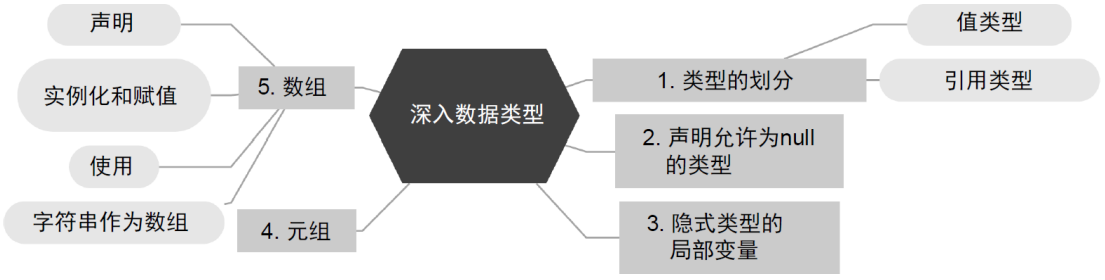
2.5 小结

即使是有经验的程序员，也要注意 C#引入的几个新垢编程构造。例如，本章探讨了用于精确金融计算的 `decimal` 类型。此外，本章还提到布尔类型 `bool` 和整型之间不允许隐式转换，目的是防止在条件表达式中误用赋值操作符。C#其他与众不同的地方还有：允许用@定义逐字字符串，用开始和结束的三个或更多双引号来定义原始字符串字面值，这两者都强迫字符串忽略转义字符；字符串插值，可以在字符串中嵌入表达式；以及 C#的 `string` 数据类型不可变。

下一章继续讨论数据类型。要讨论值类型和引用类型，还要讨论如何将数据元素组合成元组和数组。

第 3 章 深入数据类型

第 2 章讨论了所有 C#预定义类型，并简单讲述了引用类型和值类型的区别。本章继续讨论数据类型，深入解释类型划分。



此外，本章还要讨论将数据元素合并成元组的细节，这是 C# 7.0 引入的一个功能。最后，本章要讨论如何将数据分组到称为**数组**的集合中。首先，让我们深入理解值类型和引用类型。

3.1 类型的划分

一个类型要么是**值类型**，要么是**引用类型**。区别在于拷贝方式：值类型的数据总是拷贝值；而引用类型的数据总是拷贝引用。

3.1.1 值类型

除了 `string`，本书到目前为止讲述的所有预定义类型都是值类型。值类型直接包含值。换言之，变量引用的位置就是内存中实际存储值的位置。因此，将一个值赋给变量 1，再将变量 1 赋给变量 2，会在变量 2 的位置创建值的拷贝，而不是引用变量 1 的位置。这进一步造成更改变量 1 的值不会影响变量 2 的值。图 3.1 对此进行了演示。在这个图中，`number1` 引用内存中的特定位置，其中包含值 `42`。将 `number1` 的值赋给 `number2` 之后，两个变量都包含值 `42`。但修改其中任何一个变量的都不会影响另一个。

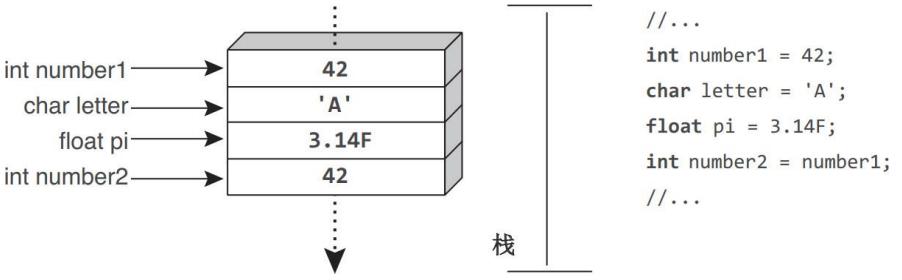


图 3.1 值类型的实例直接包含数据

类似地，将值类型的实例传给 `Console.WriteLine()`这样的方法也会生成内存拷贝。在方法内部对参数值进行的任何修改都不会影响调用函数[1]中的原始值。由于值类型需要创建内存拷贝，因此一般不要定义占用内存太多的值类型（通常应小于 16 字节）。

3.1.2 引用类型

相反，引用类型的变量存储对数据存储位置的引用，而不是直接存储数据。要去该引用所指向的位置，才能找到真正的数据。所以，为了访问数据，“运行时”要先从变量中读取内存位置，再“跳转”到包含数据的内存位置。为引用类型的变量分配实际数据的内存区域称为**堆**（heap），如图 3.2 所示。

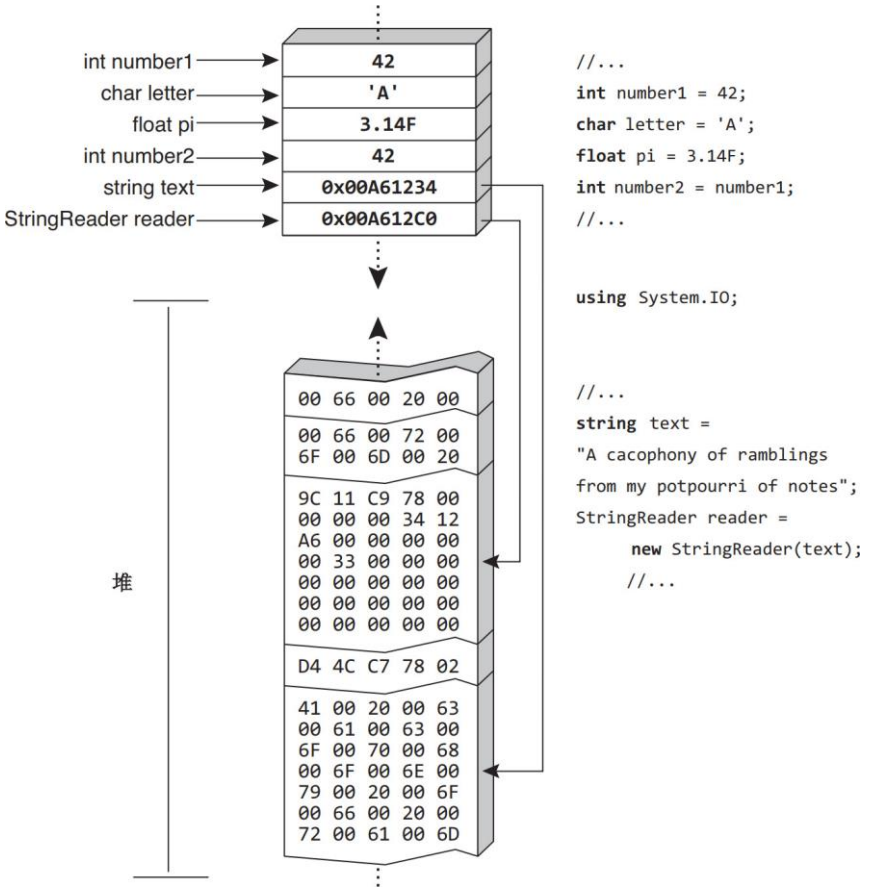


图 3.2 引用类型的实例指向堆

[1] 译注：即发出调用那个位置所在的函数，一般称为 call site。

引用类型不像值类型那样要求创建数据的内存拷贝。因此，拷贝引用类型的实例比拷贝大的值类型实例更高效。将引用类型的变量赋给另一个引用类型的变量，只会拷贝引用而不会拷贝所引用的数据。事实上，每个引用总是系统的“原生（本机）大小”。换言之，32 位系统拷贝 32 位引用，64 位系统拷贝 64 位引用，以此类推。显然，拷贝对一个大数据块的引用，速度比拷贝整个数据块快得多。

由于引用类型只拷贝对数据的引用，所以两个不同的变量可以引用相同的数据。在两个变量引用同一个对象的情况下，通过一个变量来更改对象的字段，再用另一个对象访问字段，会看到更改后的结果。无论赋值还是方法调用都会如此。因此，如果在方法内部更改引用类型的数据，那么当控制返回调用者之后，将看到更改后的结果。有鉴于此，如果对象在逻辑上是固定大小、不可变的值，就可以考虑定义成值类型。如果逻辑上是可引用、可变的东西，就应考虑定义成引用类型。

除了 `string` 和自定义类（如 `Program`），本书目前讲到的所有类型都是值类型。但在实际应用中，大多数类型都是引用类型。虽然偶尔需要自定义的值类型，但更多需要的还是自定义 的引用类型。

3.2 声明允许为 null 的类型

我们经常需要表示值“缺失”的情况。例如，在指定一个计数变量 `count` 时，如果计数未知或未赋值，那么应该在其中存储什么值呢？一种可能的解决方案是指定一个“魔法”值，例如`-1` 或 `int.MaxValue`。但是，这些都是有效的整数，所以有时会搞不清楚魔法值什么时候是正常的整数，什么时候表示缺失值。一个更可取的方法是将 `null` 赋给它，表示值无效或尚未分配的情况。在数据库编程中，`null` 属于一种“刚需”。数据库表中的列通常允许 `null` 值。但是，检索这些列，并将值赋给 C#代码中的相应变量，则可能出问题，除非 C#的数据类型也允许包含 `null`。

在声明一个类型时，可以明确允许它为 `null`[①]。为了启用可空性，在类型名称后紧跟一个**可空修饰符**（一个问号）即可。例如，`int? number = null` 将声明一个可空的 `int` 类型的变量，并将 `null` 值赋给它。遗憾的是，可空性存在一些陷阱，在启用可空性时需要特别留意。

[①] 一个技术细节是，C# 2.0 开始引入了对值类型上的可空修饰符的支持。在 C# 8.0 中，则加入了对引用类型的支持。

3.2.1 对 null 引用进行解引用

虽然允许将 null 值赋给变量是非常有价值的，但它并非毫无缺点。将 null 值复制或传递给其他变量和方法，这自然没有任何问题。但是，对一个 null 实例进行**解引用**[①]（调用其成员）将抛出一个 System.NullReferenceException 异常。例如，如果在 text 具有 null 值时调用 text.GetType()，就会抛出该异常。任何时候只要生产代码抛出了一个 System.NullReferenceException，都意味着出现了一个 bug。这个异常表明编写代码的开发人员在调用之前忘了检查 null。更糟的是，对 null 的检查要求开发人员意识到 null 值存在的可能性。因此，需要显式地执行该操作。正是因为这个原因，所以声明可空变量要求显式使用可空修饰符，而不是相反，即默认允许 null 的存在（参见稍后的“可空引用类型”小节）。换言之，当程序员选择允许变量为 null 时，他们就必须承担起一个额外的责任，即确保在任何场合下都不对值为 null 的变量进行解引用。

由于检查 null 需要用到我们尚未讨论的语句和/或操作符，所以有关如何进行 null 检查的详细信息请参见稍后的“高级主题：检查 null”。第 4 章将进行更全面的讨论。

> **高级主题：检查 null**
>
> 有许多语句和操作符可供开发人员检查 null。if 语句和 is 操作符是检查 null 最清晰的方式，如代码清单 3.1 所示。
>
> **代码清单 3.1　检查 null**
>
> ```
> public static void Main(string[] args)
> {
>     int? number = null;
>     // ...
>     if (number is null)
>     {
>         Console.WriteLine("需要为'number'提供一个值，不允许为 null。");
>     }
>     else
>     {
>         Console.WriteLine($"你在命令行提供的第一个参数包括{number}个字符/字。");
>         Console.WriteLine($"'number'的两倍是{number * 2}。");
>     }
> }
> ```
>
> if 语句检查 number 是否为 null，然后视情况采取不同的操作。虽然也可以使用相等

[①] 译注：“解引用”（dereferencing）也称为提领、用引等。

性操作符（==），但可能通过重写（overriding）来修改相等性操作符的行为；因此，这里更适合使用 is 关键字。

在处理 null 时，另一个有用的操作符是自 C# 6.0 引入的**空条件操作符**。在对可空值进行解引用之前，它首先检查值是否为 null。以 int? length = text?.Length;这个语句为例，如果 text 的值为 null，那么等号右侧的表达式会自动返回 null；否则返回存储在 text 中的字符串的长度。注意，由于从 text?.Length 返回的值可能为 null（当 text 为 null 的时候），因此必须将变量 length 声明为可空。

if 语句和空条件操作符的详情将在第 4 章进行更详细的讲解。虽然 is 关键字在第 4 章也会简单地提到，但直到第 7 章讨论模式匹配的概念时，才会进行完整讲解。

3.2.2 可空值类型

由于值类型直接引用实际的值，所以值类型天生就不能包含 null，因为根据定义，它们不能包含引用，即使是对“什么都没有（nothing）”的引用。尽管如此，在调用值类型的成员时，我们仍然使用“对值类型进行解引用”这个说法。换言之，虽然在技术上不正确，但在调用成员的时候，不管是值类型还是引用类型，都通常使用“解引用”这一术语。[①]

高级主题： 在值类型上对 null 进行解引用

从技术上说，带有可空修饰符的值类型仍然是值类型。因此，尽管它们具有 null 的行为，但在底层它们实际上并不为 null。因此，大多数时候，对表示成 null 的可空值类型进行解引用，都不会抛出跟“空引用”有关的 NullReferenceException 异常。事实上，像 HasValue、ToString()甚至与相等性相关的成员（GetHashCode()和 Equals()）都是在泛型 Nullable<T>类型上实现的。因此，它们在值代表 null 时并不会抛出“空引用”异常。相反，在对代表 null 的值类型进行解引用时，抛出的是代表“无效操作”的 InvalidOperationException 异常而不是 NullReferenceException，目的是提醒程序员在解引用之前应检查值。但是，如果在代表 null 的值类型上调用 GetType()，那么确实会抛出 NullReferenceException。之所以出现这种不一致性，是因为 GetType()不是一个虚方法，所以 Nullable<T>无法重写它的行为，只能沿用默认行为，即抛出 NullReferenceException。

3.2.3 可空引用类型

在 C# 8.0 之前，所有引用类型变量都允许为 null。遗憾的是，这个设计导致了大量程序出现了数不清的 bug。原因是为了避免空引用异常，开发人员必须意识到要检查 null，并进

[①] 可空值类型是自 C# 2.0 引入的。

行防御性编程，以避免对 null 值进行解引用。使局面进一步恶化的是，引用类型变量默认就允许为 null。未赋值的引用类型的变量默认为 null。此外，如果对未赋值的引用类型的局部变量进行解引用，那么编译器将报告错误：“使用了未赋值的局部变量'xxx'”。编译器这样做自然是无可厚非的。但是，为了避免这个问题，开发人员往往会选择最简单粗暴的解决方案，即在声明变量时无脑地赋一个 null 值，而不是在所有可能的执行路径中确保赋了一个恰当的值（参见代码清单 3.2）。换言之，开发人员很容易陷入这样的陷阱，即轻率地将声明的任何变量都初始化为null，从而防止出现上述编译器错误，并期望在解引用变量之前，代码能重新为该变量赋一个合适的值。显然，这极有可能只是一个美好的愿望。

代码清单 3.2　解引用未赋值的变量

```
#nullable enable
public static void Main()
{
    string? text;
    // ...
    // 编译错误：使用了未赋值的局部变量'text'
    System.Console.WriteLine(text.Length);
}
```

总之，引用类型默认可空，这是 System.NullReferenceException 异常频繁出现的一个主要原因。而且，编译器的行为也会误导开发人员，妨碍他们认真地、明确地采取措施来避免踏入这一陷阱。

为了解决这个问题，C#团队在 C# 8.0 中引入了**可空引用类型**的设计（这自然意味着也可以指定**不可空**的引用类型）。这一设计使引用类型跟上了值类型的脚步，也就是可以在声明时，用可空修饰符来指定是否允许为空。从 C# 8.0 开始，如果声明一个没有可空修饰符（即?）的变量，那么意味着它不允许为空。

遗憾的是，允许用可空修饰符来声明引用类型，并默认将无此修饰符的引用类型设为不允许为空，这对从早期 C#版本升级来的代码会产生重大影响。既然在 C# 7.0 以及更早的版本中，所有引用类型都允许赋值 null（例如，string text = null），那么是否意味着在 C# 8.0 以及后续版本中，以前的所有代码都会变得无法通过编译？

幸好，向后兼容是 C#团队非常注重的事情。所以，对于现有的项目，默认不会启用对可空引用类型的支持。相反，在第 1 章的 HelloWorld 程序这样的新项目中，默认会启用可空性。

有几种方法可以配置可空性，包括#nullable 指令、项目属性和命令行。在代码清单 3.2 中，我们使用#nullable 指令启用了引用类型的可用性。

```
#nullable enable
```

该指令支持 enable，disable 和 restore 这三个值，最后一个值将可空上下文还原为项目级别的设置。在代码清单 3.2 中，由于已经使用 nullable 指令启用了可空性，所以 text 能够声明为 string?类型，编译器不会发出警告。

另外，也可以通过设置项目属性来启用引用类型的可空性。如果未设置，那么项目文件（*.csproj）的项目级设置是不允许为空。要启用它，请添加一个 Nullable 项目属性，把它的值设为 enable（第 1 章的代码清单 1.2 展示了一个完整的例子）。

```
<Nullable>enable</Nullable>
```

本书所有示例代码都在项目级别启用了可空性。启用可空性的最后一种方法是使用 dotnet 命令行上的/p 参数来设置项目属性：

```
dotnet build /p:Nullable=enable
```

通过命令行为 Nullable 指定的值将覆盖项目文件（*.csproj）中设置的任何值。

3.3 隐式类型的局部变量

使用从 C# 3.0 开始引入的上下文关键字 var，我们可以声明**隐式类型的局部变量**。在声明变量的同时，如果能用类型可以确定的一个表达式来初始化它，那么就允许变量的数据类型“隐式”，无需显式指定，如代码清单 3.3 所示。

代码清单 3.3　字符串处理

```
Console.Write("输入文本: ");
var text = Console.ReadLine();

// 返回全大写的一个新字符串
var uppercase = text.ToUpper();
Console.WriteLine(uppercase);
```

与第 2 章的代码清单 2.24 相比，上述代码进行了两处修改。首先，没有将两个变量显式声明为 string 类型，而是直接声明为 var 并初始化。最终的 CIL 代码没有区别。但 var 告诉编译器根据声明时所赋的值（本例是 System.Console.ReadLine()方法的返回值）来推断数据类型。

其次，text 和 uppercase 变量都在声明时初始化。使用 var 的时候，不在声明的同时初始化会造成编译时错误。如前所述，编译器自行判断用于初始化表达式的数据的类型，并相应地声明变量，就好比程序员已经显式指定了类型。

虽然允许用 var 取代显式数据类型，但在数据类型已知的情况下，最好还是不要用 var。例如，还是应该将 text 和 uppercase 显式声明为 string。这不仅使代码更易理解，还相当于你亲自确认了等号右侧表达式返回的是你希望的数据类型。使用 var 变量时，右侧数

据类型应显而易见；否则应避免用 var 声明变量。

设计规范

AVOID using implicitly typed local variables unless the data type of the assigned value is obvious.

避免使用隐式类型的局部变量，除非所赋的值的数据类型显而易见。

> 语言对比：C++/Visual Basic/JavaScript——void*，Variant 和 var
>
> C#隐式类型的变量并不等同于 C++的 void*、Visual Basic 的 Variant 或者 JavaScript 的 var。这三种情况的变量声明都不严格，因为可以将一个不同的类型重新赋给这些变量，这类似于在 C#中将变量声明为 object 类型。相反，C#的 var 由编译器严格确定类型，确定了就不能变。另外，类型检查和成员调用都会在编译时进行验证。

3.4 元组

我们经常需要将数据元素组合到一起。例如，2022 年全球最贫穷的国家是首都位于布琼布拉（Bujumbura）的布隆迪（Burundi），人均 GDP 为 263.67 美元。利用目前讲过的编程构造，可以将上述每个数据元素存储到单独的变量中，但结果是这些数据元素相互没有任何关联。换言之，看不出 263.67 和布隆迪有什么联系。为了解决该问题，第一个方案是在变量名中使用统一的后缀或前缀，第二个方案是将所有数据合并到一个字符串中，但缺点是需要解析字符串才能处理单独的数据元素。

从 C# 7.0 开始，我们还可以使用第三个方案：**元组**（tuple）[①]，它允许在一个语句中完成所有变量的赋值，例如以下国家数据的赋值：

```
(string country, string capital, double gdpPerCapita) =
    ("布隆迪", "布琼布拉", 263.67);
```

表 3.1 总结了元组的其他语法形式。

[①] 译注：在英文中，tuple 一词源自对顺序的抽象：single，double，triple，quadruple，quintuple，*n*-tuple。

表 3.1 示例元组声明和赋值[①]

示例	说明	示例代码
1	将元组赋给单独声明的变量	`(string country, string capital, double gdpPerCapita) =` `("布隆迪", "布琼布拉", 263.67);` `Console.WriteLine(` `$@"2022 年全球最贫穷的国家是{country}, {capital}: {gdpPerCapita}");`
2	将元组赋给预声明的变量	`string country;` `string capital;` `double gdpPerCapita;` `(country, capital, gdpPerCapita) =` `("布隆迪", "布琼布拉", 263.67);` `Console.WriteLine(` `$@"2022 年全球最贫穷的国家是{country}, {capital}: {gdpPerCapita}");`
3	将元组赋给单独声明和隐式类型的变量	`(var country, var capital, var gdpPerCapita) =` `("布隆迪", "布琼布拉", 263.67);` `Console.WriteLine(` `$@"2022 年全球最贫穷的国家是{country}, {capital}: {gdpPerCapita}");`
4	将元组赋给单独声明和隐式类型的变量，但只用了一个 var	`var (country, capital, gdpPerCapita) =` `("布隆迪", "布琼布拉", 263.67);` `Console.WriteLine(` `$@"2022 年全球最贫穷的国家是{country}, {capital}: {gdpPerCapita}");`
5	声明并赋值包含具名元组项的元组，然后按名称访问元组项	`(string Name, string Capital, double GdpPerCapita) countryInfo =` `("布隆迪", "布琼布拉", 263.67);` `Console.WriteLine(` `$@"2022 年全球最贫穷的国家是{countryInfo.Name}, {countryInfo.Capital}: {countryInfo.GdpPerCapita}");`
6	声明包含具名元组项的元组，将其赋给隐式类型的	`var countryInfo =` `(Name: "布隆迪", Capital: "布琼布拉", GdpPerCapita: 263.67);`

[①] 译注：此表格中的代码已随同本书配套代码提供，详情请访问 *https://bookzhou.com*，后同。

	变量，然后按名称访问元组项	Console.WriteLine($@"2022 年全球最贫穷的国家是{countryInfo.Name}, {countryInfo.Capital}: {countryInfo.GdpPerCapita}");
7	将元组项未具名的元组赋给隐式类型的变量，通过项编号属性来访问单独的元素	var countryInfo = ("布隆迪", "布琼布拉", 263.67); Console.WriteLine($@"2022 年全球最贫穷的国家是{countryInfo.Item1}, {countryInfo.Item2}: {countryInfo.Item3}");
8	将元组项已具名的元组赋给隐式类型的变量，但还是通过项编号属性访问单独的元素	var countryInfo = (Name: "布隆迪", Capital: "布琼布拉", GdpPerCapita: 263.67); Console.WriteLine($@"2022 年全球最贫穷的国家是{countryInfo.Item1}, {countryInfo.Item2}: {countryInfo.Item3}");
9	赋值时用下划线丢弃元组的一部分（弃元）	(string name, _, double gdpPerCapita) = ("布隆迪", "布琼布拉", 263.67);
10	通过变量和属性名推断元组项名称（C# 7.1 新增）	string country = "布隆迪"; string capital = "布琼布拉"; double gdpPerCapita = 263.67; var countryInfo = (country, capital, gdpPerCapita); Console.WriteLine($@"2022 年全球最贫穷的国家是{countryInfo.country}, {countryInfo.capital}: {countryInfo.gdpPerCapita}");

前四个例子虽然右侧是元组，但左侧仍然是单独的变量，只是用**元组语法**一起赋值。在这种语法中，两个或更多元素以逗号分隔，放到一对圆括号中进行组合。（我之所以使用“元组语法”一词，是因为编译器为左侧生成的基础数据类型从技术上说并非元组。）结果是虽然右侧的值合并成元组，但在向左侧赋值的过程中，元组已被**解构**（deconstruct）它的组成部分。例 2 左边被赋值的变量是事先声明好的，但例 1，3 和 4 的变量是在元组语法中声明的。由于只是声明变量，所以命名和大小写应遵循第 1 章的设计规范，例如有一条是“**要**为局部变量使用 camelCase 大小写风格。”

注意，虽然隐式类型（var）在例 4 中用元组语法平均分配给每个变量声明，但这里的 var 绝不可以替换成显式类型（如 string）。元组的宗旨是允许每一项都有不同数据类型，所以为每一项都指定同一个显式类型名称跟这个宗旨冲突（即使类型真的一样，编译器也不允许指定显式类型）。

例 5 在左侧声明一个元组，将右侧的元组赋给它。注意元组含具名项。随后，可以引用这些名称来获取右侧元组中的值。这正是能在 `Console.WriteLine` 语句中使用 `countryInfo.Name`，`countryInfo.Capital` 和 `countryInfo.GdpPerCapita` 语法的原因。在左侧声明元组，造成多个变量被组合到单个元组变量（`countryInfo`）中。然后，可以利用元组变量来访问其组成部分。如第 4 章所述，这样的设计允许将该元组变量作为一个整体传给其他方法，而那些方法能轻松访问元组中的项。

前面说过，用元组语法定义的变量应遵守 camelCase 大小写规则。但该规则并未得到彻底贯彻。有人提倡当元组的行为和参数相似时（类似于元组语法出现之前用于返回多个值的 out 参数），这些名称应使用参数命名规则。另一个方案是 PascalCase 大小写，这是类型成员（属性、函数和公共字段，参见第 5 章和第 6 章的讨论）的命名规范。个人强烈推荐 PascalCase 规范，从而和 C#/.NET 成员标识符的大小写规范一致。但由于这并不是被广泛接受的规范，所以我在设计规范“**考虑**为所有元组项名称使用 PascalCase 大小写风格”中使用“考虑”而非“要”一词，

设计规范

DO use camelCasing for variable declarations using tuple syntax.

要为元组语法的变量声明使用 camelCase 大小写规范。

CONSIDER using PascalCasing for all tuple item names.

考虑为所有元组项名称使用 PascalCase 大小写风格

例 6 提供和例 5 一样的功能，只是右侧元组使用了具名元组项，左侧使用了隐式类型声明。但是，元组项名称会传入隐式类型变量，所以 `WriteLine` 语句仍可使用它们。当然，左侧可以使用和右侧不同的元组项名称。C#编译器允许这样做但会显示警告，指出右侧元组项名称会被忽略，因为此时左侧的优先。

不指定元组项名称，被赋值的元组变量中的单独元素仍可访问，只是名称是 `Item1`，`Item2`, ...，如例 7 所示。事实上，即便提供了自定义名称，`Item`*`X`* 名称始终都能使用，如例 8 所示。但在使用 Visual Studio 这样的 IDE 工具时， `Item`*`X`* 属性不会出现在“智能感知”的下拉列表中。这是好事，因为自己提供的名称理论上应该更好。如例 9 所示，可用下划线丢弃部分元组项的赋值，这称为一个**弃元**（discard）。

例 10 展示的元组项名称推断功能是自 C# 7.1 引入的。如本例所示，元组项名称可以根据变量名（甚至属性名）来推断。

元组是在对象中封装数据的轻量级方案，有点像你用来装杂货的购物袋。和稍后讨论的数

组不同，元组项的数据类型可以不一样，没有限制[1]，只是它们由编译器决定，不能在运行时改变。另外，元组项数量也是在编译时硬编码好的。最后，不能为元组添加自定义行为（扩展方法不在此列）。如需和封装数据关联的行为，那么应该使用面向对象编程并定义一个类，具体将在第 6 章讲述。

> **高级主题：System.ValueTuple<...>类型**
>
> 在表 3.1 的示例中，C#为赋值操作符右侧的所有元组实例生成的代码都基于一组泛型值类型（结构），例如 System.ValueTuple<T1, T2, T3>。类似地，同一组 System.ValueTuple<...>泛型值类型用于从例 5 开始的左侧数据类型。元组类型唯一包含的方法是跟比较与相等性测试有关的那些，这符合预期。
>
> 既然自定义元组项名称及其类型没有包含在 System.ValueTuple<...>定义中，为什么每个自定义元组项名称都好象是 System.ValueTuple<...>类型的成员，并能以成员的形式访问呢？让人（尤其是那些熟悉匿名类型实现的人）惊讶的是，编译器根本没有为那些和自定义名称对应的“成员”生成底层 CIL 代码，但从 C#的角度看，又似乎存在这样的成员。
>
> 对于表 3.1 的所有具名元组例子，编译器在元组剩下的作用域中显然知道那些名称。事实上，编译器（和 IDE）正是依赖该作用域通过项的名称来访问它们。 换言之，编译器查找元组声明中的项名称，并允许代码访问还在作用域中的项。也正是因为这一点，IDE 的“智能感知”不显示底层的 Item*X* 成员。它们会被忽略，替换成显式命名的项。
>
> 编译器能判断作用域中的元组项名称，这一点还好理解，但如果元组要对外公开，比如作为另一个程序集中的一个方法的参数或返回值使用（另一个程序集可能看不到你的源代码），那么会发生什么？其实对于作为 API（公共或私有）一部分的所有元组，编译器都会以“特性”（attributes）的形式将元组项名称添加到成员元数据中。例如，代码清单 3.4 展示了编译器为以下方法生成的 CIL 代码的 C#形式：
>
> ```
> public (string First, string Second) ParseNames(string fullName)
> ```
>
> **代码清单 3.4　和返回 ValueTuple 的方法的 CIL 代码对应的 C#代码（伪代码）**
>
> ```
> [return: System.Runtime.CompilerServices.TupleElementNames(
> new { "First", "Second" })]
> public System.ValueTuple<string, string> ParseNames(
> ```

[1] 但从技术上说不能是指针（第 21 章讲述指针）。

```
    string fullName)
    {
        // ...
    }
```

另外要注意，如果显式使用 System.ValueTuple<...>类型，C#就不允许使用自定义的元组项名称。所以，表 3.1 的例 8 如果将 var 替换成该类型，编译器会警告所有项的名称被忽略。

下面总结了和 System.ValueTuple<...>有关的其他注意事项：

- 共有 8 个泛型 System.ValueTuple<...>。前 7 个最大支持七元组。第 8 个是 System.ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>，可为最后一个类型参数指定另一个 ValueTuple，从而支持 *n* 元组。例如，编译器自动为 8 个参数的元组生成 System.ValueTuple<T1, T2, T3, T4, T5, T6, T7, System.ValueTuple<TSub1>>作为底层实现类型。注意，System.ValueTuple<T1>的存在只是为了补全，很少使用，因为 C#元组语法要求至少两项。
- 有一个非泛型 System.ValueTuple 类型作为元组工厂使用，提供了和所有 ValueTuple 元数[①]对应的 Create()方法。C# 7.0 以后基本用不着 Create()方法，因为像 var t1 = ("Inigo Montoya", 42)这样的元组字面值实在太好用了。
- C#程序员实际编程时完全可以忽略 System.ValueTuple 和 System.ValueTuple<T>。

还有一个元组类型是 Microsoft .NET Framework 4.5 引入的 System.Tuple<...>。当时是想把它打造成核心元组实现。但是，在C#中引入元组语法时，设计者才意识到值类型更佳，所以量身定制了 System.ValueTuple<...>，它在所有情况下都代替了 System.Tuple<...>（除非要向后兼容依赖 System.Tuple<...>的遗留 API）。

高级主题：匿名类型

C# 3.0 添加 var 的真正目的是支持**匿名类型**。匿名类型是在方法内部动态声明的数据类型，而不是通过显式的类定义来声明，如代码清单 3.5 和输出 3.1 所示。（第 15 章会深入讨论匿名类型。）

代码清单 3.5　使用匿名类型声明隐式局部变量

[①] 译注：元数的英文是 arity，源自像 unary（arity=1）、binary（arity=2）、ternary（arity=2）这样的单词。

```
var patent1 =
    new
    {
        Title = "双焦点眼镜",
        YearOfPublication = "1784"
    };
var patent2 =
new
{
    Title = "留声机",
    YearOfPublication = "1877"
};

Console.WriteLine(
    $"{patent1.Title} ({patent1.YearOfPublication})");
Console.WriteLine(
    $"{patent2.Title} ({patent2.YearOfPublication})");
```

输出 3.1

```
双焦点透镜 (1784)
留声机 (1877)
```

代码清单 3.5 演示了如何将匿名类型的值赋给一个隐式类型（var）局部变量。C# 3.0 之所以设计这种类型的操作，是为了支持连接或关联（joining 或 associating）数据类型，或者将特定类型的大小缩减至更少的数据元素。但是，自从 C# 7.0 引入元组语法后，匿名类型就几乎用不着了。

3.5 数组

第 1 章没有提到的一种特殊的变量声明是数组声明。利用数组声明，可以在单个变量中存储**同一种类型**的多个数据项，而且可以利用索引来单独访问这些数据项。C#的数组索引从零开始，所以我们说 C#数组**基于零**。

初学者主题：数组

使用数组变量声明，我们可以声明相同类型的多个数据项的集合。每一项都用名为**索引**的整数值进行唯一性标识。C#数组的第一个数据项使用索引 0 访问。由于索引基于零，应确保最大索引值比数组中的数据项总数小 1。在 C# 8.0 中，引入了“末尾索引”操作符^。例如，^1 将访问数组的最后一个元素。

初学者可以将索引想象成偏移量。第一项距数组开头的偏移量是 0，第二项的偏移量

> 是 1，以此类推。

数组是几乎所有编程语言的基本组成部分，所有开发人员都应学习。虽然 C#编程经常用到数组，初学者也确实应该掌握，但现在大多数程序都用泛型集合类型而非数组来存储数据集合。如果只是为了熟悉数组的实例化和赋值，那么可以略读下一节“数组声明”。表 3.2 列出了要注意的重点。泛型集合在第 15 章详细讲述。

表 3.2 数组的重点

说明	示例
声明： 注意数据类型后的方括号。 声明多维数组要使用逗号；逗号数量+1 = 维数。	`// 1.` `string[] languages; // 一维` `int[,] cells; // 二维`
赋值： new 关键字可以省略，但只能是在声明的时候。 如果未在声明的同时赋值，以后实例化数组时必须使用 new 关键字。 可以不指定数据类型（从 C# 3.0 开始）。但是，没有数据类型，就不能指定数组大小。 无论数组的大小还是值，都不需要是字面值；它们可以在运行时确定。 如果同时指定了值和数组大小，那么这两者必须保持一致。数组可以在没有值的情况下赋给变量，但数组中每一项的值都会被初始化为其	`// 2.` `string[] languages = {` `"C#", "COBOL", "Java",` `"C++", "TypeScript", "Pascal",` `"Python", "Lisp", "JavaScript"};` `languages = new string[]{` `"C#", "COBOL", "Java",` `"C++", "TypeScript", "Pascal",` `"Python", "Lisp", "JavaScript" };` `// 多维数组赋值和初始化` `int[,] cells = new[,]` `{` `{1, 0, 2},` `{1, 2, 0},` `{1, 2, 1}` `};` `languages = new string[9];`

默认值。 如果没有指定值，那么必须同时指定数据类型和数组大小。	
正向访问数组 数组基于零，所以第一个元素的索引是 `0`。 用方括号存储和获取数组数据。	`// 3.` `string[] languages = new[]{` `"C#", "COBOL", "Java",` `"C++", "TypeScript", "Visual Basic",` `"Python", "Lisp", "JavaScript"};` `// 从 languages 数组获取第 5 个元素(TypeScript)` `string language = languages[4];` `// 输出“TypeScript”` `Console.WriteLine(language);` `// 获取数组倒数第 3 个元素(Python)` `language = languages[^3];` `// 输出“Python”` `Console.WriteLine(language);`
逆向访问数组 从 C# 8.0 开始，还可以从数组末尾索引。例如，`^1` 对应数组最后一个元素，`^3` 对应数组倒数第三个元素。	
范围 C# 8.0 允许使用范围操作符（`..`）来标识并提取由数组元素构成的一个数组，该操作符标识的范围从起始项开始，到结束项为止（但不包括结束项）。	`string[] languages = new[]{` `"C#", "COBOL", "Java",` `"C++", "TypeScript", "Visual Basic",` `"Python", "Lisp", "JavaScript"};` `Console.WriteLine($@"^3..^0: {` `// Python, Lisp, JavaScript` `string.Join(", ", languages[^3..^0])}");` `Console.WriteLine($@"^3..: {` `// Python, Lisp, JavaScript` `string.Join(", ", languages[^3..])}");` `Console.WriteLine($@" 3..^3: {` `// C++, TypeScript, Visual Basic` `string.Join(", ", languages[3..^3])}");` `Console.WriteLine($@" ..^6: {` `// C#, COBOL, Java` `string.Join(", ", languages[..^6])}");`

此外，本章最后一节“常见数组错误”还会讲到数组的一些特点。

3.5.1 数组声明

C#用方括号声明数组变量。首先指定数组元素的类型，后跟一对方括号，再输入变量名。代码清单 3.6 声明字符串数组变量 languages。

代码清单 3.6　声明数组

```
string[] languages;
```

显然，数组声明的第一部分标识了数组中存储的元素的类型。作为声明的一部分，方括号指定了数组的**秩**（rank），或者说维数。本例声明的是一维数组。类型和维数构成了 languages 变量的数据类型。

> **语言对比：C++和 Java——数组声明**
>
> 在 C#中，作为数组声明一部分的方括号紧跟在数据类型之后，而不是在变量声明之后。这样所有类型信息都在一起，而不是像 C++和 Java 那样分散于标识符前后（Java 允许方括号出现在数据类型或变量名之后）。

代码清单 3.6 定义的是一维数组。如果在方括号中出现了逗号，那么它定义的是额外的维。例如，代码清单 3.7 为井字棋（tic-tac-toe）棋盘定义了一个二维数组。

代码清单 3.7　声明二维数组

```
//    |   |
// ---+---+---
//    |   |
// ---+---+---
//    |   |
int[,] cells;
```

代码清单 3.7 定义了一个二维数组。第一维对应从左到右的单元格，第二维对应从上到下的单元格。可以用更多的逗号来定义更多的维，数组总维数等于逗号数加 1。注意某一维上的元素数量不是变量声明的一部分。这是在创建（实例化）数组并为每个元素分配内存空间时指定的。

3.5.2 数组实例化和赋值

声明数组后，可以在一对大括号中使用以逗号分隔的数据项列表来填充它的值。代码清单3.8 声明一个字符串数组，将一对大括号中的 9 种编程语言的名称赋给它。

代码清单 3.8　声明数组的同时赋值

```
string[] languages = { "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript" };
```

列表的第一项成为数组第一个元素，第二项成为第二个，以此类推。我们用大括号定义数组字面值。只有在同一个语句中声明并赋值，才能使用代码清单 3.8 的赋值语法。如果只是声明，然后在其他地方赋值，那么必须使用 new 关键字，如代码清单 3.9 所示。

代码清单 3.9　先声明数组，以后再赋值

```
string[] languages;
languages = new string[]{ "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript" };
```

从 C# 3.0 开始，不需要在 new 后指定数组类型（string）。编译器能根据初始化列表中的数据类型推断数组类型。但方括号仍不可缺少。

C#还允许将 new 关键字作为声明语句的一部分，所以可以像代码清单 3.10 那样在声明的同时赋值。

代码清单 3.10　声明数组的同时用 new 赋值

```
string[] languages = new string[]{
   "C#", "COBOL", "Java",
   "C++", "TypeScript", "Visual Basic",
   "Python", "Lisp", "JavaScript" };
```

new 关键字的作用是指示“运行时”为数据类型分配内存，即指示它实例化数据类型（本例是数组）。

数组赋值时只要使用了 new 关键字，就可以在方括号内指定数组大小，如代码清单 3.11 所示。

代码清单 3.11　声明数组的同时用 new 关键字赋值，并指定数组大小

```
string[] languages = new string[9]{
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript" };
```

指定的数组大小必须与大括号中的元素数量匹配。另外，也可单纯分配[1]数组，但不提供初始值，如代码清单 3.12 所示。

代码清单 3.12　分配数组但不提供初始值

```
string[] languages = new string[9];
```

分配数组但不指定初始值，“运行时”会将每个元素初始化为它们的默认值，如下所示：

- 引用类型——无论是否可空（比如 string 和 string?）初始化为 null
- 可空值类型初始化为 null
- 非空数值类型初始化为 0
- bool 初始化为 false
- char 初始化为\0

非基元值类型以递归方式初始化，其每个字段都被初始化为默认值。所以，其实并不需要在使用数组前初始化它的所有元素。

由于数组大小不需要作为变量声明的一部分，所以可以在运行时指定数组大小。例如，代码清单 3.13 根据在 Console.ReadLine()调用中由用户指定的一个大小来创建数组。

代码清单 3.13　在运行时确定数组大小

```
string[] groceryList;
Console.Write("购物清单中有多少种商品？ ");
int size = int.Parse(Console.ReadLine());
groceryList = new string[size];
// ...
```

[1] 译注：在编程中说到“分配”（allocate）时，一般都是指在运行时从“堆”中动态分配一块内存。这是一个耗时、耗资源的操作。在 C#中，我们用 new 关键字告诉系统“分配”内存。

C#以类似的方式处理多维数组。每一维的大小以逗号分隔。代码清单 3.14 初始化一个没有开始落子的井字棋棋盘。

代码清单 3.14　声明二维数组

```
int[,] cells = new int[3, 3];
```

还可以像代码清单 3.15 那样，将井字棋的棋盘初始化成特定的棋子布局。

代码清单 3.15　初始化二维整数数组

```
int[,] cells = {
    {1, 0, 2},
    {1, 2, 0},
    {1, 2, 1}
};
```

在声明的多维数组中，包含三个 int[]类型的元素，每个元素的大小一样（本例凑巧也是 3）。注意，每个 int[]元素的大小必须完全一样。也就是说，像代码清单 3.16 那样的声明是无效的。

代码清单 3.16　大小不一致的多维数组会造成错误

```
// 错误，每一维的大小必须一致
int[,] cells = {
    {1, 0, 2, 0},
    {1, 2, 0},
    {1, 2},
    {1}
};
```

为了表示棋盘，并不一定需要在每个位置都使用整数。另一个办法是为每个玩家都单独提供虚拟棋盘，每个棋盘都包含一个 bool 来指出玩家选择的位置。代码清单 3.17 对应于一个三维棋盘。

代码清单 3.17　初始化三维数组

```
bool[,,] cells;
cells = new bool[2, 3, 3]
{
    // Player 1 moves
    //  X |   |
    // ---+---+---
    //  X |   |
    // ---+---+---
    //  X |   | X
    //
    {
        {true, false, false},
        {true, false, false},
        {true, false, true}
    },
    // Player 2 moves
    //    |   | O
    // ---+---+---
    //    | O |
    // ---+---+---
    //    | O |
    {
        {false, false, true},
        {false, true, false},
        {false, true, true}
    }
};
```

本例初始化棋盘，并显式指定每一维的大小。new 表达式除了指定大小，还提供了数组的字面值。bool[,,]类型的字面值被分解成两个 bool[,]类型的二维数组（大小均为 3×3）。每个二维数组都由三个 bool 数组（大小均为 3）构成。

如前所述，多维数组（这种普通的多维数组也称为“矩形数组”）每一维的大小必须一致。除此之外，还可以定义**交错数组**（jagged array），也就是由数组构成的数组。交错数组的语法稍微有别于多维数组。而且交错数组不需要具有一致的大小。所以，可以像代码清单 3.18 那样初始化交错数组。

代码清单 3.18　初始化交错数组

```
int[][] cells = {
    new [] {1, 0, 2, 0},
    new [] {1, 2, 0},
    new [] {1, 2},
    new [] {1}
};
```

交错数组不用逗号标识新维。相反，交错数组定义由数组构成的数组。代码清单 3.18 在 int[]后添加[]，表明数组元素本身是 int[]类型的数组。

注意，交错数组要求为每个内部数组都创建一个数组实例（或 null）。在前面的例子中，是使用 new 来实例化嵌套数组的内部元素。如果省略这个实例化操作，将导致编译错误。（如前所述，只要提供了值，数据类型是可以省略的。）

另外，从 C# 12.0 开始可以用集合表达式初始化数组。所以，本例像 new [] {1, 0, 2, 0} 这样的写法可以简化为 [1, 0, 2, 0]。

3.5.3 使用数组

我们使用方括号（称为**数组访问符**）访问数组元素。为了获取第一个元素，要指定 0 作为索引。代码清单 3.19 将数组变量 languages 中的第 5 个元素（索引 4）的值存储到字符串变量 language 中。

代码清单 3.19　声明并访问数组

```
string[] languages = new[]{
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript"};
// 获取 languages 数组的第五项(TypeScript)
string language = languages[4];
// 输出"TypeScript"
Console.WriteLine(language);
// 获取倒数第三项(Python)
language = languages[^3];
// 输出"Python"
Console.WriteLine(language);
```

从 C# 8.0 开始，还可以使用“末尾索引”操作符^，它相对于数组末尾来访问项。例如，用^1 来访问数组最后一项。因此，由于示例数组共有 9 项，所以为了访问数组第一个元素，需要使用^9。在代码清单 3.19 中，^3 指定将 languages 数组倒数第三项的值（"Python"）赋给 language 变量。

由于^1 对应最后一个元素，所以^0 对应超出最后一个数组元素的位置。当然，这个位置并没有元素，因此不能使用^0 来索引一个具体的元素，但它代表真正的“数组末尾”。这类似于不能用数组的长度（本例是 9）作为数组索引。此外，在索引数组时不能使用负值。

对比使用正整数从数组开头进行索引，以及使用^整数值（或返回整数值的表达式）从数组末尾进行索引，这两者之间似乎存在差异。前者从 0 开始计数访问第一个元素，而后者从^1 开始以访问最后一个元素。C#设计团队选择使用 0 来保持与 C#的前身语言（包括 C，C++和 Java）的一致性。而在从末尾进行索引时，C#参照的是 Python（其他 C 风格的语言不支持末尾索引操作符），也从 1 开始计数。然而，与 Python 不同的是，C#团队选择了^操

作符（而不是负整数），以确保在使用索引操作符访问允许负值的集合类型（非数组）时向后兼容。（^操作符在支持范围时具有额外的优势，本章稍后就会讨论。）

> **语言对比： C++/Java/JavaScript/Python——数组索引**
>
> 与其前辈（C++/Java/JavaScript/Python）一样，C#也从 0 开始索引数组。虽然这些前辈大多数不支持从末尾索引，但 Python 支持。与 Python 相似，C#从末尾索引的起始值是 1；只是末尾索引操作符用的是^，而不是-。

为了记住末尾索引操作符的工作方式，请注意使用正整数从末尾索引时，最后一个元素的索引是 length - 1，倒数第二个元素的索引是 length - 2，以此类推。从 length 中减去的那个整数就是“末尾索引”值，即^1、^2 等。此外，采用这种记忆法，你会发现对于任何一个数组元素，它正数的索引值和倒数的索引值加起来，必然正好等于数组长度。

注意，末尾索引操作符不限于字面整数值。还可以使用表达式，例如，languages[^languages.Length]将返回第一个元素。

> **注意：**在 C#中，索引从 0 开始计数以访问第一个元素，而从末尾进行索引的操作符从 ^1 开始，用于访问最后一个元素。从末尾索引时要使用^操作符，从索引为“长度-1”的元素开始。从“长度”中减去的那个整数就是末尾索引要使用的值。对于任何一个数组元素，从数组开头的索引值和从数组末尾开头的索引值加到一起，始终等于数组的长度。

还可以使用方括号语法将数据存储到数组中。代码清单 3.20 交换了"C++"和"Java"的顺序。

代码清单 3.20　交换数组中不同位置的数据

```
string[] languages = new[] {
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript" };
// 将"C++"存储到 language 变量中
string language = languages[3];
// 将"Java"赋给原本是"C++"的位置
languages[3] = languages[2];
// 将 language 的值赋给"Java"的位置
languages[2] = language;
```

多维数组的元素用每一维的索引来标识，如代码清单 3.21 所示。

代码清单 3.21　初始化二维整数数组

```
int[,] cells = {
    {1, 0, 2},
    {0, 2, 0},
    {1, 2, 1}
};
// 将井字棋的决胜落子设为玩家 1，
// 因为玩家 1 只需在第二行(索引 1)的第一列(索引 0)落子，即获胜。
cells[1, 0] = 1;
```

交错数组元素的赋值稍有不同，这是因为它必须与交错数组的声明一致。第一个索引指定“由数组构成的数组”中的一个数组。第二个索引指定具体是该数组中的哪一项（参见代码清单 3.22）。

代码清单 3.22　向交错数组的一个元素赋值

```
int[][] cells = {
    [1, 0, 2, 0],
    [1, 2, 0],
    [1, 2],
    [1]
};

// 向第二个数组([1])的第一个[0]元素赋值
cells[1][0] = 0;
```

可以像代码清单 3.23 那样获取数组长度。

代码清单 3.23　获取数组长度

```
string[] languages = new[] {
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Visual Basic",
    "Python", "Lisp", "JavaScript" };
```

```
Console.WriteLine($"数组中有{languages.Length}种语言。");
```

数组具有固定长度；除非重新创建数组，否则不能随便更改。此外，越过数组的**边界**（或长度）会造成“运行时”报错。用无效索引（指向的元素不存在）来访问（检索或者赋值）数组时就会发生这种情况。例如，在代码清单 3.24 中，用数组长度作为（正向）索引来访问数组时，会在程序运行时报错。

代码清单 3.24　访问数组时越界会抛出异常

```
string[] languages = new string[9];
// ...
// 运行时错误：索引越界 - 最后一个元素的索引应为 8
languages[4] = languages[9];
```

> **注意：**Length 成员返回的是数组元素个数，而不是最高索引值。languages 变量的 Length 成员的值是 9，而 languages 变量的最高索引是 8，那是从起点能到达的最远位置。

> **语言对比：C++ ——缓冲区溢出错误**
>
> 非托管 C++并非总是检查是否越过数组边界。这个错误不仅很难调试，而且有可能造成潜在的安全问题，也就是所谓的**缓冲区溢出**。相反，CLR 能防止所有 C#（和托管 C++）代码越界，消除了在托管代码中发生缓冲区溢出的可能。

从 C# 8.0 开始，如果直接访问元素^0，那么也会出现类似的问题。由于^1 才是最后一项，所以^0 代表数组真正的末尾——那里没有真正的元素。

为了避免在访问数组最后一个元素时越界，可以进行长度检查，以确保数组的长度大于 0，并使用^1（从 C# 8.0 开始）或 Length - 1 代替硬编码的值来访问数组中的最后一项。使用 Length 作为索引时，需要减 1 以避免出现越界错误（参见代码清单 3.25）。

代码清单 3.25　在数组索引中使用 Length - 1

```
string[] languages = new string[9];
// ...
languages[4] = languages[languages.Length - 1];
```

当然，如果一开始就有可能不存在数组实例，那么应该在访问数组之前首先检查它是否为 null。

设计规范

CONSIDER checking the array length before indexing into an array rather than assuming the length.

考虑在对数组进行索引之前检查数组的长度，而不是假设一个长度。

CONSIDER using the index from end operator (^) rather than `Length - 1` with C# 8.0 or higher.

考虑在 C# 8.0 或更高版本中使用末尾索引操作符（^），而不是使用 `Length - 1`。

Length 返回数组中元素的总数。因此，如果你有一个多维数组，比如大小为 2×3×3 的 bool cells[,,]数组，那么 Length 会返回元素总数 18。

对于交错数组，Length 返回外部数组（第一个数组）的元素数——交错数组是“数组构成的数组”，所以 Length 只作用于外层数组，只统计它的元素数（也就是它具体由多少个数组构成），而不管各内部数组总共包含了多少个元素。

3.5.4 范围

C# 8.0 引入的另一个与索引相关的功能是对数组切片（范围）的支持，即从数组中提取一部分到一个新数组。指定范围的语法是在两个索引（包括末尾索引）之间使用范围操作符（..）。代码清单 3.26 展示了一个例子。

代码清单 3.26　范围操作符的例子

```
string[] languages = [
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Swift",
    "Python", "Lisp", "JavaScript"];

// 1. C#, COBOL, Java
Console.WriteLine($@" 0..3: {string.Join(", ", languages[0..3]) }");
```

```
// 2. Python, Lisp, JavaScript
Console.WriteLine($@"^3..^0: {string.Join(", ", languages[^3..^0]) }");

// 3. C++, TypeScript, Swift
Console.WriteLine($@" 3..^3: {string.Join(", ", languages[3..^3]) }");

// 4. C#, COBOL, Java
Console.WriteLine($@" ..^6: {string.Join(", ", languages[..^6]) }");

// 5. Python, Lisp, JavaScript
Console.WriteLine($@" 6..: {string.Join(", ", languages[6..]) }");

// 6. C#, COBOL, Java, C++, TypeScript, Swift, Python, Lisp, JavaScript
Console.WriteLine($@" ..: {string.Join(", ", languages[..]) }");

// 7. C#, COBOL, Java, C++, TypeScript, Swift, Python, Lisp, JavaScript
Console.WriteLine($@" 0..^0: {string.Join(", ", languages[0..^0])}");
```

范围操作符的一个重点在于，被包括到范围中的是从第一项（含）到最后一项（不含）的所有项。因此，在代码清单 3.26 节的例 1 中，`0..3` 这个范围包括从索引 `0` 到索引 `3`（但不包括 `3`）的所有项（这里使用 `3` 来标识第四项，因为正向索引是从 `0` 开始计数）。而在例 2 的范围`^3..^0` 中，包含的将是数组的最后三项。注意，`^0` 不会因为尝试访问数组末尾之后的位置而引发错误，因为范围不会包含标记范围的最后一项。

范围起始项的索引和最后一项的索引都是可选的。如代码清单 3.26 的例 4～例 6 所示。另外例 7 所示，如果完全省略索引，那么等效于指定范围 `0..^0`。

最后要注意，和其他 C#内置类型一样，索引和范围是.NET/C#中具有“一等公民”（first-class）地位的类型，详情请参见后续的“高级主题：`System.Index` 和 `System.Range`”。它们并非只能用于数组访问。

> **注意：**范围操作符（`..`）通过指定从第一项（含）到最后一项（不含）的方式来返回一个范围。

> **高级主题：`System.Index` 和 `System.Range`**
>
> 末尾索引操作符返回的是一个 `System.Index` 类型的值。因此，完全可以在方括号的上下文之外使用索引。例如，可以声明一个索引，并将索引操作符的结果赋给它：

```
System.Index index = ^42;
```

还可以将普通的整数赋给 System.Index。System.Index 类型有两个属性，一个是 int 类型的 Value，另一个是 bool 类型的 IsFromEnd。后者显然用于指示索引是从数组的开始还是末尾计数。

此外，用于指定范围的数据类型是 System.Range。因此，可以声明一个范围并这样赋值：

```
System.Range range = ..^0;
```

甚至可以这样写：

```
System.Range range = ..;
```

System.Range 有两个属性：Start 和 End。

通过提供这些类型，C# 使你能编写支持范围和末尾索引的自定义集合（我们将在第 17 章讨论如何编写自定义集合）。

3.5.5 更多数组方法

数组提供了其他许多方法来操作数组元素，其中包括 Sort()，BinarySearch()，Reverse() 和 Clear()等，如代码清单 3.27 和输出 3.2 所示。

代码清单 3.27　更多数组方法

```
string[] languages = {
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Swift",
    "Python", "Lisp", "JavaScript"};

Array.Sort(languages);

string searchString = "COBOL";
int index = Array.BinarySearch(languages, searchString);
Console.WriteLine("未来的浪潮，" + $"{searchString}，位于索引{index}。");
Console.WriteLine();
Console.WriteLine($"{"第一个元素",-26}\t{"最后一个元素",-26}");
Console.WriteLine($"{"--------------",-26}\t{"--------------",-26}");
Console.WriteLine($"{languages[0],-26}\t{languages[^1],-26}");
Array.Reverse(languages);
Console.WriteLine($"{languages[0],-26}\t{languages[^1],-26}");

// 注意：Clear 方法不是从数组中删除所有项，
// 相反，它是将每一项设为当前类型的默认值
```

```
Array.Clear(languages, 0, languages.Length);
Console.WriteLine($"{languages[0],-26}\t{languages[^1],-26}");
Console.WriteLine($"Clear 后，数组大小是：{languages.Length}");
```

输出 3.2

```
未来的浪潮，COBOL，位于索引 2。

第一个元素                      最后一个元素
--------------                  --------------
C#                              TypeScript
TypeScript                      C#

Clear 后，数组大小是：9
```

这些方法通过 System.Array 类提供。大多数都一目了然，但要注意以下两点。

- 使用 BinarySearch()方法前要先对数组进行排序[①]。值不按升序排序，会返回不正确的索引。目标元素不存在会返回负值；在这种情况下，可以使用按位求补操作符 ~index 来返回比目标元素大的第一个元素的索引（如果有的话）。
- Clear()方法不删除数组元素，不会将长度设为零。数组大小是固定的，不能修改。所以，Clear()方法将每个元素都设为其默认值（false，0 或 null）。这解释了在调用 Clear()之后输出数组内容时，Console.WriteLine()为什么输出一个空行。

3.5.6 数组的实例成员

类似于字符串，数组也有不从数据类型而是从变量访问的实例成员。Length 就是一个例子，它通过数组变量（数组实例）来访问，而非通过类。其他常用实例成员还有 GetLength()，Rank 和 Clone()。

获取特定维的长度不是使用 Length 属性，而是使用数组的 GetLength()实例方法，调用时，需要指定返回哪一维的长度，如代码清单 3.28 和输出 3.3 所示。

代码清单 3.28　获取特定维的大小

```
bool[,,] cells;
cells = new bool[2, 3, 3];
Console.WriteLine(cells.GetLength(0)); // 显示 2
Console.WriteLine(cells.Rank); // 显示 3
```

输出 3.3

[①] 译注：事先排好序，这是所有二叉查找算法的前置要求。

```
2
3
```

第一个输出是 2，这是第一维的元素个数。另外，还可以访问数组的 Rank 成员来获取整个数组的维数（逗号数+1）。如输出 3.3 所示，cells.Rank 返回 3。

将一个数组变量赋给另一个数组变量，默认只会拷贝数组引用，不会拷贝数组中单独的元素。要创建数组的全新拷贝，需要使用数组的 Clone()方法。该方法返回数组的完整拷贝，修改这个数组拷贝不会影响原始数组。

3.5.7 字符串作为数组

string 类型的变量可以作为一个字符数组来访问。例如，可以调用 palindrome[3]来获取 palindrome 字符串的第 4 个字符。注意，由于字符串不可变，所以不能向字符串中的特定位置赋值。所以，对于 palindrome 字符串来说， palindrome[3]='a'这样的写法在 C#中是不允许的。代码清单 3.29 使用数组访问符（方括号）来判断命令行上提供的参数是不是选项（选项的第一个字符是短划线）。

代码清单 3.29 判断命令行选项

```
public static void Main(string[] args)
{
    // ...
    // 命令行输入的以空格分隔的每个参数都是一个字符串(字符数组)
    // 所有字符串构成了 args 字符串数组
    if (args[0][0] == '-')
    {
        // 该参数是选项
    }
}
```

上述代码使用了会在第 4 章讲述的 if 语句。注意，第一个数组访问符[]获取字符串数组 args 的第一个元素，第二个数组访问符则获取该字符串的第一个字符。上述代码等价于代码清单 3.30。

代码清单 3.30 判断命令行选项（简化版）

```
public static void Main(string[] args)
{
    // ...
    string arg = args[0];
    if(arg[0] == '-')
    {
```

```
            // 该参数是选项
        }
    }
```

不仅可以使用数组访问符单独访问字符串中的字符，还可以使用字符串的 ToCharArray()方法将整个字符串作为字符数组返回。然后，就可以使用 System.Array.Reverse()方法反转数组中的元素，如代码清单 3.31 和输出 3.4 所示，该程序判断字符串是不是回文。

代码清单 3.31　反转字符串

```
string reverse, palindrome;
char[] temp;

Console.Write("输入一句回文: ");
palindrome = Console.ReadLine();

// 删除空格，并转换成小写
reverse = palindrome.Replace(" ", "");
reverse = reverse.ToLower();

// 转换成字符数组
temp = reverse.ToCharArray();


// 反转数组
Array.Reverse(temp);

// 将数组转换回字符串，并检查
// 反转后的字符串是否相等
if (reverse == new string(temp))
{
    Console.WriteLine(
        $"\"{palindrome}\"是回文。");
}
else
{
    Console.WriteLine(
        $"\"{palindrome}\"不是回文。");
}
```

输出 3.4

```
输入一句回文：菜油炒油菜
"菜油炒油菜"是回文。

输入一句回文：Never Odd Or  Even
"Never Odd Or  Even"是回文。
```

这个例子使用 new 关键字向 string 的构造函数传递反转好的字符数组来创建新字符串。

3.5.8 常见数组错误

前面描述了三种类型的数组：一维、多维和交错。一些规则和特点约束着数组的声明和使用。表 3.3 总结了数组编码时的一些常见错误，有助于巩固对这些规则的了解。阅读时最好先看“常见错误”一栏的代码（先不要看错误说明和改正后的代码），看自己是否能发现错误，检验你对数组及其语法的理解。

表 3.3 常见数组编码错误

常见错误	错误说明	改正后的代码
`int numbers[];`	用于声明数组的方括号放在数据类型之后，而不是在变量标识符之后	`int[] numbers;`
`int[] numbers;` `numbers =` `  {42, 84, 168 };`	如果是在声明之后再对数组进行赋值，需要使用 new 关键字，并可选择指定数据类型	`int[] numbers;` `numbers = new int[]{` `  42, 84, 168 }`
`int[3] numbers =` `  { 42, 84, 168 };`	不能在变量声明中指定数组大小	`int[] numbers =` `  { 42, 84, 168 };`
`int[] numbers =` `  new int[];`	除非提供数组字面值，否则必须在初始化时指定数组大小	`int[] numbers =` `  new int[3];`
`int[] numbers =` `  new int[3]{}`	数组大小指定为 3，但数组字面值中没有任何元素。数组的大小必须与数组字面值中的元素个数相符	`int[] numbers =` `  new int[3]` `  { 42, 84, 168 };`
`int[] numbers =` `  new int[3];` `Console.WriteLine(` `  numbers[3]);`	数组索引起始于零。因此，最后一项的索引比数组长度小 1。注意这是运行时错误，而不是编译时错误	`int[] numbers =` `  new int[3];` `Console.WriteLine(` `  Numbers[2]);`

`int[] numbers = new int[3]; numbers[numbers.Length]= 42;`	和上一个错误相同：需要从 Length 减去 1 来访问最后一个元素。注意这是运行时错误，而不是编译时错误	`int[] numbers = new int[3]; numbers[numbers.Length-1] = 42;`
`int[] numbers; Console.WriteLine( numbers[0]);`	尚未对 numbers 数组实例化，暂时不可访问	`int[] numbers = {42, 84}; Console.WriteLine( numbers[0]);`
`int[,] numbers = { {42}, {84, 42} };`	多维数组的结构必须一致	`int[,] numbers = { {42, 168}, {84, 42} };`

3.5 小结

本章首先讨论了两种不同的类型：值类型和引用类型。它们是 C#程序员必须理解的基本概念，虽然读代码时可能看不太出来，但它们改变了类型的底层机制。

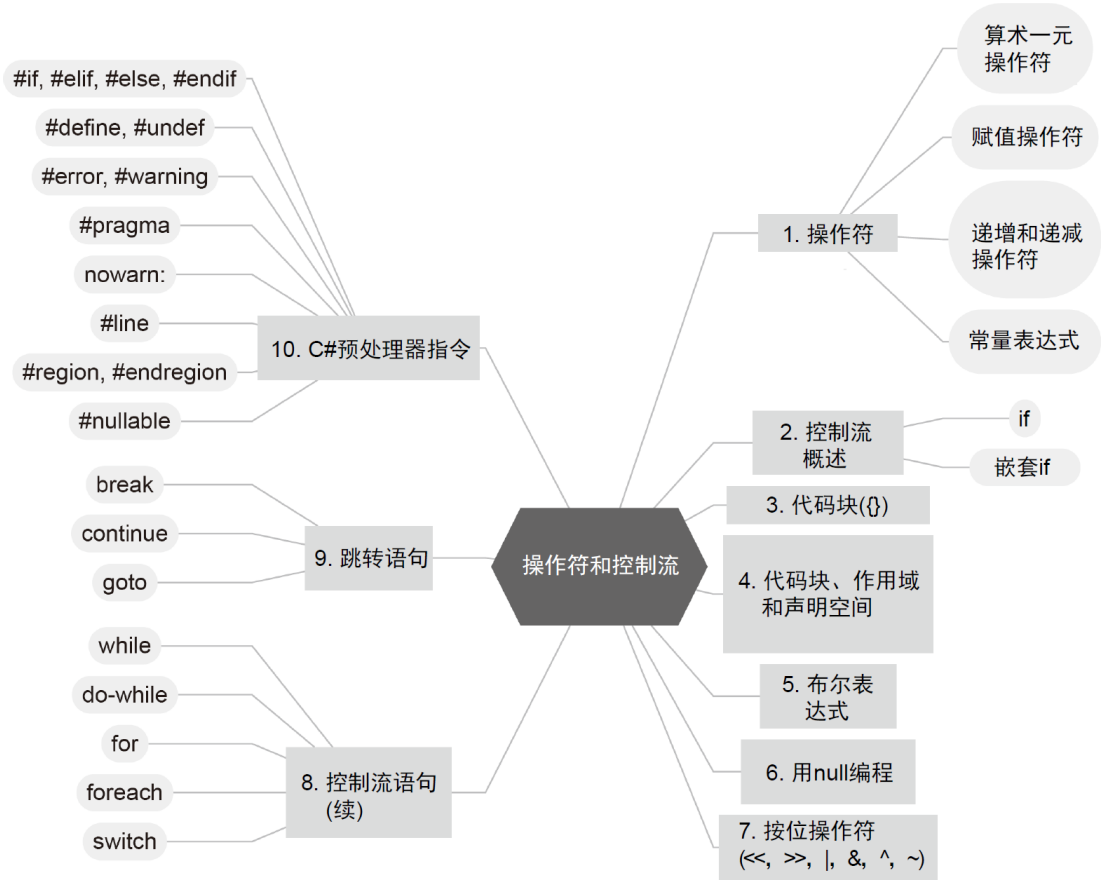
讨论数组之前，我们讨论了 C#后来引入的两种语言构造。首先讨论的是 C# 2.0 引入的可空修饰符（?），它允许值类型存储 null，并允许引用类型显式指定是否要存储 null。然后讨论的是元组，并介绍如何用 C# 7.0 引入的新语法处理元组，同时不必显式地和底层数据类型打交道。

本章最后讨论了 C#数组语法，并介绍了各种数组处理方式。许多开发者刚开始不容易熟练掌握这些语法。所以提供了一个常见错误列表，专门列出与数组编码有关的错误。

下一章将讨论表达式和控制流语句。本章最后出现过几次的 if 语句会一并讨论。

第 4 章 操作符和控制流

本章学习操作符、控制流语句和 C#预处理器。**操作符**[1]提供了对操作数执行各种计算或操作的语法。**控制流语句**控制程序的条件逻辑，或多次重复一节代码。在介绍了 `if` 控制流语句后，本章将探讨布尔表达式的概念，许多控制流语句都需要这种表达式。还会提到整数不能转换为 `bool`（显式转型也不行），并讨论了这个设计的好处。本章最后讨论了 C#预处理器指令。



4.1 操作符

第 2 章学习了预定义数据类型，本节学习如何将操作符应用于这些数据类型来执行各种计算。例如，可对声明好的变量执行计算。

---

[1] 译注：operator 在本书统一采用“操作符”的说法，而不是“运算符”。相应地，operand 是“操作数”，而不是“运算子”。

> 初学者主题：操作符
>
> **操作符**（operator）对称为**操作数**（operand）的值（或变量）执行数学或逻辑运算/操作来生成新值（称为**结果**）。例如，代码清单 4.1 有两个操作数，即数字 4 和 2，它们被减法操作符（-）组合到一起，结果赋给变量 difference（差）。
>
> 代码清单 4.1　简单的操作符例子
>
> ```
> int difference = 4 - 2;
> ```

通常将操作符划分为三大类：一元、二元和三元，分别对应一个、两个和三个操作符。此外，虽然一些操作符用符号表示，例如+，-，?.和??，但还有一些操作符采用的是关键字形式，例如 default 和 is。本节讨论最基本的一元和二元操作符，三元操作符将在本章后面简略介绍。

4.1.1 一元正负操作符(+，-)

我们有时需要改变数值的正负号。这时一元负操作符（-）就能派上用场。例如，代码清单 4.2 将当前的美国国债金额变成负值，指明这是欠款。

代码清单 4.2　指定负值

```
// 美国国债金额，最新数据请查询 https://www.usdebtclock.org/
decimal debt = -33940505930933M;
```

使用一元负操作符等价于从零减去操作数。一元正操作符（+）对值几乎没有影响[①]。它在C#语言中是多余的，只是出于对称性的考虑才加进来。

4.1.2 二元算术操作符(+，-，*，/，%)

二元操作符要求两个操作数。C#为二元操作符使用中缀记号法：操作符在左右操作数之

[①] 一元+操作符定义为获取 int，unit，nint，nuint，long，ulong，float，double 和 decimal 类型（以及这些类型的可空版本）的操作数。作用于其他类型（例如 short）时，操作数会相应地转换为上述某个类型。

间。除赋值之外的每个二元操作符的结果必须以某种方式使用（例如作为另一个表达式的操作数）。

> **语言对比：C++——光杆表达式语句**
>
> 和上面提到的规则相反， C++甚至允许像 `4+5;`这样的二元表达式作为独立语句使用。在 C#中，只有赋值、调用、递增、递减、`await` 和对象创建表达式才能作为独立语句使用。

代码清单 4.3 和输出 4.1 展示了使用二元操作符（更准确地说是二元算术操作符）的例子。算术操作符的每一边都有一个操作数，计算结果赋给一个变量。除了二元减法操作符（`-`），其他二元算术操作符还有加法（`+`）、除法（`/`）、乘法（`*`）和取余操作符（`%`，有时也称为取模或模除操作符）。

代码清单 4.3　使用二元操作符

```
Console.Write("输入分子: ");
numerator = int.Parse(Console.ReadLine());

Console.Write("输入分母: ");
denominator = int.Parse(Console.ReadLine());

quotient = numerator / denominator;  // 除法
remainder = numerator % denominator; // 取余

Console.WriteLine(
    $"{numerator} / {denominator} = 商{quotient}余{remainder}。");
```

输出 4.1

```
输入分子: 23
输入分母: 3
23 / 3 = 商 7 余 2。
```

在加了注释的两个赋值语句中，除法和取余均先于赋值发生。操作符的执行顺序取决于它们的**优先级**和**结合性**。迄今为止用过的操作符的优先级如下。

1. `*`，`/`和`%`具有最高优先级
2. `+`和`-`具有较低优先级
3. `=`在 6 个操作符中优先级最低

所以，上述语句的行为符合预期，除法和取余匀先于赋值进行。

如果忘记对二元操作符的结果进行赋值，那么会出现如输出 4.2 所示的编译错误。

输出 4.2

```
... error CS0201: 只有 assignment、call、increment、decrement 和 new 对象表达式可用作语句
```

> **初学者主题：圆括号、结合性、优先级和求值**
>
> 包含多个操作符的表达式可能让人分不清楚每个操作符的操作数。例如，在表达式 x+y*z 中，很明显表达式 x 是+操作符的操作数，z 是*操作符的操作数。但 y 是+还是*的操作数？
>
> **圆括号**清楚地将操作数与操作符关联。如果希望 y 是被加数，那么可以写(x+y)*z。如希望 y 是被乘数，那么可以写 x+(y*z)。
>
> 但是，包含多个操作符的表达式不一定非要添加圆括号。编译器能根据结合性和优先级判断执行顺序。**结合性**决定相似操作符的执行顺序，**优先级**决定不相似操作符的执行顺序。
>
> 二元操作符可以“左结合”或“右结合”，具体取决于“位于中间”的表达式是从属于左边的操作符，还是从属于右边的操作符。例如，a-b-c 被判定为(a-b)-c，而不是 a-(b-c)。这是因为减法操作符为“左结合”。C#的大多数操作符都是左结合的，赋值操作符右结合。
>
> 对于不相似操作符，要根据操作符优先级决定位于中间的操作数从属于哪一边。例如，乘法优先级高于加法，所以表达式 x+y*z 求值为 x+(y*z)而不是 (x+y)*z。
>
> 但是，通常一个好的实践是坚持用圆括号增强代码可读性，即使这有时显得“多余”。例如，在执行摄氏-华氏温度换算时，(c*9.0/5.0)+32.0 比 c*9.0/5.0+32.0 更易读，即使圆括号在这个表达式中可以省略。
>
> **设计规范**
>
> DO use parentheses to make code more readable, particularly if the operator precedence is not clear to the casual reader.
>
> **要**用圆括号增加代码的易读性，尤其是在操作符优先级不是让人一目了然的时候。
>
> 很明显，对于相邻的两个操作符，高优先级的会先于低优先级的执行。例如 x+y*z 是先乘后加，乘法结果变成了加法操作符的右操作数。但要注意，优先级和结合性只影响操作符自身的执行顺序，不影响操作数的求值顺序。

> 在 C#中，操作数总是从左向右求值。在包含三个方法调用的表达式中，比如 A()+B()*C()，首先求值 A()，然后 B()，然后 C()，然后乘法操作符决定乘积，最后加法操作符决定和。不能因为 C()是乘法操作数，A()是加法操作数，就认为 C()先于 A()调用。

> **语言对比：C++——操作数求值顺序**
>
> 和上述规则相反，C++规范允许不同的实现自行选择操作数的求值顺序。对于 A()+B()*C()这样的表达式，不同的 C++编译器可能选择以不同顺序求值函数调用，只要乘积是某个被加数即可。例如，可以选择先求值 B()，再 A()，再 C()，再乘法，最后加法。

将加法操作符用于字符串

操作符也可用于非数值类型。例如，可以使用加法操作符来连接（拼接）两个或更多字符串，如代码清单 4.4 和输出 4.3 所示。

代码清单 4.4　将二元操作符应用于非数值类型

```
short windSpeed = 67; // 风速，单位是公里/小时
Console.WriteLine(
    $"华盛顿州原来的塔科马大桥" +
    $"{Environment.NewLine}被"
    + "风速为"
    + windSpeed + "公里/小时的大风摧毁。");
```

输出 4.3

```
华盛顿州原来的塔科马大桥
被风速为 67 公里/小时的大风撕毁。
```

由于不同语言文化的语句结构迥异，所以开发者注意不要对准备本地化的字符串使用加法操作符。类似地，虽然可用字符串插值技术在字符串中嵌入表达式，但其他语言的本地化仍然要求将字符串移至某个资源文件，这使字符串插值没了用武之地。在这种情况下，复合格式化更理想（第 1 章的代码清单 1.19 展示了一个例子）。

设计规范

DO favor composite formatting over use of the addition operator for concatenating strings when localization is a possibility.

程序可能需要本地化的时候，**要**用复合格式化而不是加法操作符来连接字符串，。

在算术运算中使用字符

第 2 章在介绍 char 类型时，提到虽然 char 类型存储的是字符而不是数字，但它是整型（意味基于整数）。可以和其他整型一起参与算术运算。但是，不是基于存储的字符来解释 char 类型的值，而是基于它的基础值。例如，数字字符'3'用 Unicode 值 0x33（十六进制）表示，换算成十进制值是 51。数字字符'4'用 Unicode 值 0x34 表示，或十进制 52。如代码清单 4.5 和输出 4.4 所示，'3'和'4'这两个字符相加，得到十六进制值 0x167，即十进制 103，等价于字母字符'g'。

代码清单 4.5　将加法操作符应用于 char 数据类型

```
int n = '3' + '4';
char c = (char)n;
Console.WriteLine(c); // 输出'g'
```

输出 4.4

```
g
```

可以利用 char 类型的这个特点判断两个字符相距多远。例如，字母 f 与字母 c 有 3 个字符的距离。为了获得这个值，用字母 f 减去字母 c 即可，如代码清单 4.6 和输出 4.5 所示。

代码清单 4.6　判断两个字符之间的“距离”

```
int distance = 'f' - 'c';
Console.WriteLine(distance);
```

输出 4.5

```
3
```

浮点类型的特殊性

浮点类型 float 和 double 有一些特殊性，比如它们管理精度的方式。本节通过一些实例帮助认识浮点类型的特殊性。

float 支持 7 个有效数位，所以能准确地容纳值 7654321F 和值 0.1234567F。但这两个 float 值相加的结果会被取整为 7654321，因为小数部分加到一起，超过了 float 能容纳的 7 个有效数位。这种类型的取整有时是致命的，尤其是在执行重复性计算或相等性检查的时候（参见稍后的“高级主题：浮点类型造成非预期的不相等”）。

二进制浮点类型内部存储二进制分数而不是十进制分数。所以，一次简单的赋值就可能引发精度问题，例如 double number = 140.6F。140.6 的准确值是分数 703/5。但是，由于分母不是 2 的整数次幂，所以无法用二进制浮点数准确表示。实际分母是用 float 的 32 个二进制位能表示的最接近的一个值。

由于 double 能容纳比 float 更精确的值，所以 C#编译器实际将该表达式求值为 double number = 140.600006103516，这是最接近 140.6F 的二进制分数，但表示成 double 后，占用的内存会比 140.6F 稍大。

设计规范

AVOID binary floating-point types when exact decimal arithmetic is required; use the decimal floating-point type instead.

避免在需要准确的十进制小数算术运算时使用二进制浮点类型，改为使用 decimal 浮点类型。

> **高级主题：浮点类型造成非预期的不相等**
>
> 比较两个值是否相等，浮点类型的不准确性可能造成严重后果。有时本应相等的值被错误地判断为不相等，如代码清单 4.7 和输出 4.6 所示。
>
> **代码清单 4.7　浮点类型的不准确性造成非预期的不相等**
>
> ```
> decimal decimalNumber = 4.2M;
> double doubleNumber1 = 0.1F * 42F;
> double doubleNumber2 = 0.1D * 42D;
> float floatNumber = 0.1F * 42F;
> ```

```
// 1. 显示：4.2 != 4.2000002861023 - True
Console.WriteLine($"{decimalNumber} != {(decimal)doubleNumber1} - {decimalNumber != (decimal)doubleNumber1}");

// 2. 显示：4.2 != 4.200000286102295 - True
Console.WriteLine($"{(double)decimalNumber}       !=       {doubleNumber1}       - {(double)decimalNumber != doubleNumber1}");

// 3. 显示：(float)4.2M != 4.2000003F - True
Console.WriteLine($"(float){(float)decimalNumber}M     !=     {floatNumber}F     - {(float)decimalNumber != floatNumber}");

// 4. 显示：4.200000286102295 != 4.2 - True
Console.WriteLine($"{doubleNumber1}   !=   {doubleNumber2}   -   {doubleNumber1   != doubleNumber2}");

// 5. 显示：4.2000003F != 4.2D - True
Console.WriteLine($"{floatNumber}F   !=   {doubleNumber2}D   -   {floatNumber   != doubleNumber2}");

// 6. 显示：4.1999999809265137 != 4.2 - True
Console.WriteLine($"{(double)4.2F} != {4.2D} - {(double)4.2F != 4.2D}");

// 7. 显示：4.2F != 4.2D - True
Console.WriteLine($"{4.2F}F != {4.2D}D - { 4.2F !=  4.2D}");
```

输出 4.6

```
4.2 != 4.2000002861023 - True
4.2 != 4.200000286102295 - True
(float)4.2M != 4.2000003F - True
4.200000286102295 != 4.2 - True
4.2000003F != 4.2D - True
4.1999999809265137 != 4.2 - True
4.2F != 4.2D - True
```

可以看出，虽然某些值理论上应该相等，但由于浮点数的不准确性，它们都被错误地判断为不相等。

设计规范

AVOID using equality conditionals with binary floating-point types. Either subtract the two values and see if their difference is less than a tolerance, or use the `decimal` type.

避免对二进制浮点类型的值进行相等性测试。要么判断两个值之差是否在容差范围之内，要么使用 `decimal` 类型。

浮点类型还有其他特殊性。例如，整数除以零理论上应该报错。int 和 decimal 等数据类型确实会如此。但 float 和 double 允许结果是一个特殊值，如代码清单 4.8 和输出 4.7 所示。

代码清单 4.8　浮点数被零除的结果是 NaN

```
float n = 0f;
// 显示：NaN
Console.WriteLine(n / 0);
```

输出 4.7

```
NaN
```

在数学中，某些算术运算是“未定义”的，例如 0 除以它自己。在 C#中，float 0 除以 0 会得到一个“Not a Number”（非数字，NaN）结果。试图打印这样的一个数，实际输出的就是 NaN。类似地，获取负数的平方根（System.Math.Sqrt(-1)）也会得到 NaN。

浮点数可能溢出边界。例如，float 的上边界约为 3.4×10^{38}。一旦溢出，结果数就会存储为“正无穷大”（∞）。类似地，float 的下边界是 -3.4×10^{38}，溢出会得到“负无穷大”（-∞）。代码清单 4.9 分别生成正负无穷大，输出 4.8 展示了结果。

代码清单 4.9　溢出 float 值边界

```
// 显示：-∞（负无穷大）
Console.WriteLine(-1f / 0);

// 显示：∞（正无穷大）
Console.WriteLine(3.402823E+38f * 2f);
```

输出 4.8

```
-∞
∞
```

进一步研究浮点数，发现它能包含非常接近零、但实际不是零的值。如果值超过 float 或 double 类型的阈值，那么值可能表示成“负零”或者“正零”，具体取决于数是负还是正，并在输出中表示成-0 或者 0。

4.1.3 复合赋值操作符(+=，-=，*=，/=，%=)

第 1 章讨论了简单的赋值操作符（=），它将操作符右边的值赋给左边的变量。复合赋值操作符将常见的二元操作符与赋值操作符结合到一起，如代码清单 4.10 所示。

代码清单 4.10　常见的递增计算

```
int x = 123;
x = x + 2;
```

在这个例子中，首先计算 x + 2，结果赋回 x。由于这种形式的运算相当普遍，所以专门设计了一些**复合赋值操作符**来集成计算与赋值。例如，+=操作符使左侧的变量递增右侧的值，如代码清单 4.11 所示。

代码清单 4.11　使用+=操作符

```
int x = 123;
x += 2;
```

上述代码与代码清单 4.10 是等价的。

还有其他复合赋值操作符提供了类似的功能。例如，赋值操作符还可以和减法、乘法、除法与取余操作符合并，如代码清单 4.12 所示。

代码清单 4.12　其他复合赋值操作符

```
x -= 2;
x /= 2;
x *= 2;
x %= 2;
```

4.1.4 递增和递减操作符(++，--)

C#提供了特殊的一元操作符来实现计数器的递增和递减。**递增操作符**（++）每次使一个变量递增 1。所以，代码清单 4.13 每一行代码的作用都一样。

代码清单 4.13　递增操作符

```
spaceCount = spaceCount + 1;
```

```
spaceCount += 1;
spaceCount++;
```

类似地，**递减操作符**（--）使变量递减 1。所以，代码清单 4.14 每一行代码的作用都一样。

代码清单 4.14　递减操作符

```
lines = lines - 1;
lines -= 1;
lines--;
```

> 初学者主题：循环中的递减示例
>
> 递增和递减操作符在循环（比如稍后要讲到的 while 循环）中经常用到。例如，代码清单 4.15 使用递减操作符逆向遍历英语字母表的每个字母，结果如输出 4.9 所示。
>
> 代码清单 4.15　降序显示每个字母的 Unicode 值
>
> ```
> char current;
> int unicodeValue;
>
> // 设置 current 的初始值
> current = 'z';
>
> do
> {
>     // 获取 current 的 Unicode 值
>     unicodeValue = current;
>     Console.Write($"{current}={unicodeValue}\t");
>
>     // 继承处理英语字母表的前一个字母
>     current--;
> } while (current >= 'a');
> ```
>
> 输出 4.9
>
> ```
> z=122   y=121   x=120   w=119   v=118   u=117   t=116   s=115   r=114   q=113  p=112
> o=111   n=110   m=109   l=108   k=107   j=106   i=105   h=104   g=103   f=102  e=101
> d=100   c=99    b=98    a=97
> ```
>
> 递增和递减操作符用于控制特定操作的执行次数。本例还要注意递减操作符应用于字符（char）数据类型。只要数据类型支持“下一个值”和“上一个值”的概念，就适合使用递增和递减操作符。更多信息请参见第 10 章的“操作符重载”一节。

以前说过，赋值操作符首先计算要赋的值，再执行赋值。赋值操作符[①]返回的结果就是要赋的值。递增和递减操作符与此相似。也是计算要赋的值，再执行赋值。所以，赋值操作符可以和递增/递减操作符一起使用。但是，如果不仔细，可能得到令人困惑的结果。如代码清单 4.16 和输出 4.10 所示。

代码清单 4.16　使用后缀递增操作符

```
int count = 123;
int result;
result = count++;
Console.WriteLine($"result = {result}, count = {count}");
```

输出 4.10

```
result = 123, count = 124
```

你可能会觉得奇怪，赋给 result 的居然是 count 递增前的值。递增或递减操作符的位置决定了所赋的值是操作数求值之前还是之后的值。所以，如果希望 result 的值是递增/递减后的结果，那么需要将操作符放在想递增/递减的变量之前，如代码清单 4.17 和输出 4.11 所示。

代码清单 4.17　使用前缀递增操作符

```
int count = 123;
int result;
result = ++count;
Console.WriteLine($"result = {result}, count = {count}");
```

输出 4.11

```
result = 124, count = 124
```

本例的递增操作符出现在操作数之前，所以是先对表达式进行求值，再将结果赋给 result 变量。假定 count 为 123，那么++count 会先对 count 进行递增，结果 124 赋给 result。相反，后缀形式 count++是先将 count 的值赋给左侧的变量 result，再对 count 进行递增。

---

① 译注：所有操作符在底层都是作为操作符方法来实现的，比如 operator+(...)。

代码清单 4.18 和输出 4.12 展示了前缀和后缀操作符在行为上的差异。

代码清单 4.18　对比前缀和后缀递增操作符

```
int x = 123;
// 显示：123, 124, 125
Console.WriteLine($"{x++}, {x++}, {x}");

// x 现在的值是 125
// 显示：126, 127, 127
Console.WriteLine($"{++x}, {++x}, {x}");
// x 现在的值是 127
```

输出 4.12

```
123, 124, 125
126, 127, 127
```

在代码清单 4.18 中，递增和递减操作符相对于操作数的位置影响了表达式的结果。前缀操作符的结果是变量递增/递减之后的值，而后缀操作符的结果是变量递增/递减之前的值。在语句中使用这些操作符应该小心。若心存疑虑，最好独立使用这些操作符（自成一个语句）。这样不仅代码更易读，还可以保证不犯错。

> **语言对比：C++——不同的实现有不同的行为**
>
> 以前说过，C++的不同实现可以任意选择表达式中的操作数的求值顺序，而 C#总是从左向右。类似地，在 C++中实现递增和递减时，可按任何顺序执行递增和递减的副作用[①]。例如，在 C++中，对于 `M(x++, x++)`这样的调用，假定 x 初值是 1，那么既可以调用 `M(1,2)`，也可以调用 `M(2,1)`，具体由编译器决定。而 C#总是调用 `M(1,2)`，因为 C#做出了两点保证。第一，传给调用的实参总是从左向右计算。第二，总是先将已递增的值赋给变量，再使用表达式的值。这两点 C++都不保证。

[①] 译注：在计算机编程中，如果一个函数/方法或表达式除了生成一个值，还会造成状态的改变，就说它会造成副作用；或者说会执行一个副作用。

设计规范

AVOID confusing usages of the increment and decrement operators.

避免递增和递减操作符的使用让人迷惑。

DO be cautious when porting code between C, C++, and C# that uses increment and decrement operators; C and C++ implementations need not follow the same rules as C#.

在 C、C++和 C#之间移植使用了递增和递减操作符的代码**要**小心；C 和 C++的实现遵循的不一定是和 C#相同的规则。

> **高级主题：线程安全的递增和递减**
>
> 虽然递增和递减操作符简化了代码，但两者执行的都不是原子级别的运算。在操作符执行期间，可能发生线程上下文切换，可能造成竞争条件。可用 `lock` 语句防止出现竞争条件。但是，对于简单递增和递减运算，一个代价没有那么高的替代方案是使用由 `System.Threading.Interlocked` 类提供的线程安全方法 `Increment()`和 `Decrement()`。这两个方法依赖处理器的功能来执行快速的、线程安全的递增和递减运算（详情参见第 19 章）。

4.1.5 常量表达式和常量符号

上一章讨论了字面值，或者说直接嵌入代码的值。可以使用操作符将多个字面值合并到一个常量表达式中。根据定义，**常量表达式**是 C#编译器能在编译时求值的表达式（而不是在运行时才能求值），因其完全由常量操作数构成。然后，可以使用常量表达式初始化常量符号，从而为常量值分配一个名称（类似于局部变量为存储位置分配一个名称）。例如，可用常量表达式计算一天中的秒数，结果赋给一个常量符号，并在其他表达式中使用该符号。

代码清单 4.19 中的 `const` 关键字的作用就是声明常量符号。由于常量和“变量”相反（“常”意味着“不可变”），以后在代码中任何修改它的企图都会造成编译时错误。

代码清单 4.19　声明常量

```
const int secondsPerDay = 60 * 60 * 24;          // 每天秒数
const int secondsPerWeek = secondsPerDay * 7;    // 每周秒数
```

注意，赋给 `secondsPerWeek` 的也是常量表达式。表达式中所有操作数都是常量，编译器

能确定结果。

设计规范

DO NOT use a constant for any value that can possibly change over time. The value of pi and the number of protons in an atom of gold are constants; the price of gold, the name of your company, and the version number of your program can change.

不要用常量表示将来可能改变的任何值。π 和金原子的质子数是常量。金价、公司名和程序版本号则应该是变量。

C# 10 引入了对**常量插值字符串**的支持，允许定义含有插值的一个常量字符串，前提是插入的表达式也是常量字符串，可以在编译时完成求值（参见代码清单 4.20）。

代码清单 4.20　使用非数值类型的二元操作符

```
const string windSpeed = "67";
const string announcement = $"""
    华盛顿州原来的塔科马大桥
    被风速为{windSpeed}公里/小时的大风摧毁。
    """;
Console.WriteLine(announcement);
```

尽管代码清单 4.20 中的 announcement 是一个插值字符串，但字符串中插入的表达式的值也是常量字符串，这就使编译器能在编译时而不是执行时完成求值。另外要注意的是，windSpeed 变量的值是一个字符串而不是整数。常量字符串插值仅在插入其他常量字符串时才有效，不能插入其他数据类型——即使这些类型也是常量。这个限制是为了允许在执行时根据语言文化的差异进行字符串转换。例如，将黄金比率转换为字符串可能得到 1.6180339887 或 1,6180339887，具体取决于运行环境的语言文化。

4.2 控制流概述

本章后面的代码清单 4.46 展示了如何以一种简单方式查看一个数的二进制形式。但即便如此简单的程序，不用控制流语句也写不出来。控制流语句控制程序的执行路径。本节讨论如何基于条件检查来改变语句的执行顺序。以后还会学习如何通过循环构造来反复执行一组语句。

表 4.1 总结了所有控制流语句。注意，“常规语法结构”这一栏给出的只是常见的语句用法，没有给出完整的词法结构。其中，“嵌入语句”是除了当前标注的“语句”或“声明”之外的其他任何语句，但通常是一个代码块（即{...}）。

表 4.1 控制流语句[①]

语句	常规语法结构	示例
if 语句	**if** (*boolean-expression*) *embedded-statement*	**if** (input == "quit") { Console.WriteLine("游戏结束"); **return**; }
if 语句	**if** (*boolean-expression*) *embedded-statement* **else** *embedded-statement*	**if** (input == "quit") { System.Console.WriteLine("游戏结束"); **return**; } **else** GetNextMove();
while 语句	**while** (*boolean-expression*) *embedded-statement*	**while**(count < total) { System.Console.WriteLine($"count = {count}"); count++; }
do while 语句	**do** *embedded-statement* **while** (*boolean-expression*) ;	**do** { Console.WriteLine("输入名字:"); input = System.Console.ReadLine(); } **while**(input != "exit");
for 语句	**for** (*for-initializer;* *boolean-expression;* for-iterator) *embedded-statement*	**for** (**int** count = 1; count <= 10; count++) { Console.WriteLine($"count = {count}"); }

[①] 译注：此表格中的代码已随同本书配套代码提供，后同。

foreach 语句	**foreach** (*type identifier* **in** *expression*) *embedded-statement*	**foreach** (**char** letter **in** email) { **if**(!insideDomain) { **if** (letter == '@') { insideDomain = **true**; } **continue**; } Console.Write(letter); }
continue 语句	**continue**;	
switch 语句	**switch**(*governing-type-expression*) { ... **case** *const-expression*: *statement-list* *jump-statement* **default**: *statement-list* *jump-statement* }	**switch**(input) { **case** "exit": **case** "quit": Console.WriteLine("退出程序...."); **break**; **case** "restart": Reset(); **goto case** "start"; **case** "start": GetMove(); break; **default**: System.Console.WriteLine(input); **break**; }
break 语句	**break**;	
goto 语句	**goto** *identifier*;	
	goto case *const-expression*;	
	goto default;	

在后文的井字棋[①]程序中，用到了表 4.1 展示每个 C#控制流语句。如果有兴趣，可以直接查看第 4 章的源代码文件 TicTacToe.cs（*https://github.com/transbot/EssentialCSharp*）。程序会显示一个井字棋棋盘，提示每个玩家落子，并在每一次落子之后更新。

本章剩余部分将详细讨论每一种语句。在讨论了 **if** 语句后，会先解释代码块、作用域、

[①] 译注：有的国家和地区也将 Tic-Tac-Toe 称为“画圈打叉”游戏。我们称为“井字棋”。

布尔表达式以及按位操作符的概念，然后才会讨论其他控制流语句。由于 C#和其他语言存在很多相似性，部分读者可能发现该表格非常熟悉。这部分读者可以直接跳到“C#预处理器指令”一节，或者直接跳到本章最后的“小结”。

4.2.1 if 语句

if 语句是 C#最常见的语句之一。它对称为**条件**（condition）的**布尔表达式**（返回 true 或 false 的表达式）进行求值。条件为 true，就执行**后续语句**（consequence-statement）。if 语句可以有 else 子句，其中包含在条件求值为 false 时执行的**替代语句**（alternative-statement）。常规形式如下：

```
if (condition)
  consequence-statement
else
  alternative-statement
```

在代码清单 4.21 中，如果玩家输入 1，那么程序将显示"人机对战"；否则显示"双人对战"。

代码清单 4.21　if/else 语句示例

```
string input;

// 提示用户选择单人还是双人游戏
Console.Write($"""
            1 - 人机对战
            2 - 双人对战
            请选择:
            """
);
input = Console.ReadLine();

if (input == "1")
    // 用户选择人机对战
    Console.WriteLine("人机对战。");
else
    // 其他情况都默认双人对战(即使用户输入的不是 2)
    Console.WriteLine("双人对战。");
```

4.2.2 嵌套 if

代码有时需要多个 if 语句。代码清单 4.22 首先判断玩家是否输入了一个小于或等于 0 的数字要求退出。如果不是，就检查用户是否知道井字棋最多能走多少步。输出 4.13 展示了结果。

代码清单 4.22　嵌套 if 语句

```
int input;    // 声明一个变量来存储用户输入

Console.Write(
    "井字棋最多能走" +
    "多少步?" +
    " (输入 0 退出): ");

// int.Parse()将 ReadLine()的
// 返回值转换为 int
input = int.Parse(Console.ReadLine());

// 条件 1
if (input <= 0)
    // 输入小于等于 0
    Console.WriteLine("退出...");
else
    // 条件 2
    if (input < 9)
    // 输入小于 9
    Console.WriteLine(
        "井字棋最大步数" +
        $"大于{input}");
    else
        // 条件 3
        if (input > 9)
        // 输入大于 9
            Console.WriteLine(
                "井字棋最大步数" +
                $"小于{input}");
        // 条件 4
        else
            // 输入等于 9
            Console.WriteLine(
                "正确，井字棋最多" +
                "只能走 9 步。");
```

输出 4.13

```
井字棋最多能走多少步? (输入 0 退出): 9
正确，井字棋最多只能走 9 步。
```

假定在提示输入时，玩家输入了 9，那么执行路径如下。

- 条件 1：检查 input 是否小于 0。因为不是，所以跳到条件 2。
- 条件 2：检查 input 是否小于 9。因为不是，所以跳到条件 3。
- 条件 3：检查 input 是否大于 9。因为不是，所以跳到条件 4。
- 条件 4：显示答案正确。

代码清单 4.22 使用了嵌套 if 语句。为分清嵌套结构，代码行在排版时进行了缩进。但如第 1 章所述，空白不影响执行路径。有没有缩进和换行，代码执行起来都一样。代码清单 4.23 展示了嵌套 if 语句的另一种形式，与代码清单 4.22 等价。

代码清单 4.22　if/else 连贯格式化

```
if (input <= 0)
    Console.WriteLine("退出...");
else if (input < 9)
    Console.WriteLine(
        "井字棋最大步数" +
        $"大于{input}");
else if (input > 9)
    Console.WriteLine(
        "井字棋最大步数" +
        $"小于{input}");
else
    Console.WriteLine(
        "正确，井字棋最多" +
        "只能走 9 步。");
```

虽然后一种格式更常见，但无论哪种情况，都应选择代码最易读的格式。

上述两个代码清单的 if 语句都省略了大括号。但正如马上就要讲到的那样，这和设计规范不符。 规范提倡除了单行语句之外都使用由大括号（{}）界定的代码块。

4.3 代码块({})

在前面的示例 if 语句中，if 和 else 之后跟随了单一的 Console.WriteLine();语句，如代码清单 4.24 所示。

代码清单 4.24　没有添加代码块的 if 语句

```
if (input <= 0)
    Console.WriteLine("退出...");
```

可以使用大括号将多个语句合并成**代码块**，以便在符合条件时执行多个语句。例如代码清单 4.25 中用于计算半径的代码块，结果如输出 4.14 所示。

代码清单 4.25　跟随了代码块的 if 语句

```
double radius;  // 声明一个变量来存储半径
double area;    // 声明一个变量来存储面积

Console.Write("输入圆的半径: ");

// double.Parse 将 ReadLine()返回的结果
// 转换成一个 double
string temp = Console.ReadLine();
radius = double.Parse(temp);
if (radius >= 0)
{
    // 计算圆的面积
    area = Math.PI * radius * radius;
    Console.WriteLine(
        $"这个圆的面积是: {area:0.00}");
}
else
{
    Console.WriteLine($"{radius}不是有效半径值。");
}
```

输出 4.14

```
输入圆的半径: 3
这个圆的面积是: 28.27
```

在这个例子中，if 语句检查 radius（半径）是否为零或正数。如果是，就计算并显示圆的面积；否则显示一条消息，指出半径无效。

注意，第一个 if 之后跟随了两个语句，它们被封闭在一对大括号中。大括号将多个语句合并成代码块，可以将整个代码块视为单个语句。

如果去掉代码清单 4.25 中用于创建代码块的大括号，在布尔表达式返回 true 的前提下，只有紧接在 if 语句之后的那个语句才会执行。无论布尔表达式求值结果是什么，后续所有语句都会执行。代码清单 4.26 展示了这种无效的代码。

代码清单 4.26　依赖缩进造成无效的代码

```
if (radius >= 0)
    area = Math.PI * radius * radius;
    Console.WriteLine($"圆的面积是: {area:0.00}");
```

在 C#中，缩进仅用来增强代码的可读性。编译器会忽略它，所以上述代码在语义上等价于代码清单 4.27。

代码清单 4.27　语义上等价于代码清单 4.26

```
if (radius >= 0)
{
    area = Math.PI * radius * radius;
}
Console.WriteLine($"圆的面积是: {area:0.00}");
```

程序员必须防止此类不容易发现的错误。一种比较极端的做法是，始终在控制流语句之后包括一个代码块，即使其中只有一个语句。事实上，设计规范是除非最简单的单行 `if` 语句，否则都要添加大括号。

虽然比较少见，但也可以独立使用一个代码块，它在语义上不属于任何控制流语句。换言之，大括号可以自成一体（例如，没有对应的条件，也不在循环中），这完全合法。

在前几个代码清单中，π 值用 `System.Math` 类的 `PI` 常量表示。编程时不要硬编码 π 和 e（自然对数的底），请用 `System.Math.PI` 或 `System.Math.E`。

设计规范

AVOID omitting braces, except for the simplest of single-line `if` statements.

除非最简单的单行 `if` 语句，否则**避免**省略大括号

4.4 代码块、作用域和声明空间

代码块经常被称为**作用域**（scope），但两个术语并非完全可以互换。某个具名事物的作用域是指源代码的一个区域。可在该区域使用非限定名称（前面不加限定前缀）来引用该事物。局部变量的作用域就是封闭它的那个代码块。这正是经常将代码块称为“作用域”的原因。

人们经常混淆作用域和声明空间。**声明空间**（declaration space）是具名事物的逻辑容器。该容器中不能存在同名的两个事物。代码块不仅定义了作用域，还定义了局部变量声明空间。同一个声明空间中，不允许声明两个同名的局部变量。在声明了一个局部变量的代码块外部，没有办法用该局部变量的名称引用它；这个时候，我们说该局部变量“超出作用

域”。类似地，不能在同一个类中声明具有Main()签名的两个方法。（方法的规则有一些放宽：在同一个声明空间中，允许存在签名不同的两个同名方法。方法的签名包括它的名称和参数的数量/类型。）

简单地说，作用域决定了一个名称引用的是什么事物，而声明空间决定了同名的两个事物是否冲突。在代码清单 4.28 中（结果如输出 4.15 所示），message 是在 if 语句主体内声明的，这样就把它的作用域限制在了 if 主体内部。之后再使用它的名称时，它已经超出了作用域。要纠正错误，必须在 if 语句外部声明该变量。

代码清单 4.28　变量在其作用域外无法访问

```
string playerCount;
Console.Write("输入玩家数量(1 或 2):");
playerCount = Console.ReadLine();

if (playerCount != "1" && playerCount != "2")
{
    string message = "你输入了无效的玩家数量。";
}
else
{
    // ...
}
// 错误：message 超出作用域
Console.WriteLine(message);
```

输出 4.15

```
...
...\Program.cs(18,26): error CS0103: 当前上下文中不存在名称'message'
```

声明空间覆盖了所有子代码块。C#编译器禁止一个代码块中声明（或作为参数声明）的局部变量在其子代码块中重复声明。总之，一个变量的声明空间是当前代码块，以及它的所有子代码块。在代码清单 4.28 中，由于 playerCount 是在方法的代码块（方法主体）中声明的，所以在这个代码块的任何地方（包括子代码块）都不能再次声明。

message 这个名称则仅在 if 块中可用，在外部（包括 else 子块）不能使用。类似地，playerCount 在整个方法（包括 if 和 else 子块）中引用的都是同一个变量。

> **语言对比：C++ ——局部变量作用域**
>
> 在 C++中，对于块中声明的局部变量，它的作用域是从声明位置开始，到块尾结束。

> 声明前对局部变量的引用会失败，因为局部变量此时不在作用域内。此时若有另一个同名的事物在作用域中，C++会将名称解析成对那个事物的引用，而这可能不是你的原意。C#的规则稍有不同，对于声明局部变量的那个块，局部变量都在作用域中，但声明前引用它属于非法。换言之，此时局部变量合法存在，但使用非法。只有在声明后的位置使用才合法。这是C#为了防止像C++那样出现不容易察觉之错误的众多规则之一。

4.5 布尔表达式

if 语句中包含在圆括号内的部分是布尔表达式，称为**条件**。如代码清单 4.29 的 if 语句的条件所示。

代码清单 4.29　布尔表达式

```
if (input < 9)
{
    // 输入小于 9
    Console.WriteLine("井字棋最大步数" + $"大于{input}");
}
// ...
```

许多控制流语句都要使用布尔表达式，其关键特征在于总是求值为 true 或 false。input < 9 之所以是一个布尔表达式，是因为它返回了一个 bool 值。例如，编译器不允许将布尔表达式写成 x = 42，因为它的作用是对 x 进行赋值，并返回新值，而不是检查 x 的值是否等于 42。

> **语言对比：C++——在本该使用==的地方使用了=**
>
> C#消除了 C/C++的一个常见的编码错误。代码清单 4.30 在 C++中不会出错。
>
> **代码清单 4.30　C++允许将赋值作为条件**
>
> ```
> if (input = 9) // C++允许，但 C#不允许
>     System.Console.WriteLine("正确，井字棋最多只能走 9 步。");
> ```
>
> 虽然上述代码表面上是检查 input 是否等于 9，但正如第 1 章讲过的那样，=代表的是赋值操作符，而不是检查相等性的操作符。从赋值操作符返回的是赋给变量的值，本例就是 9。然而，作为 int 的 9 无法被判定为布尔表达式，所以是 C#编译器不允许的。C 和 C++将非零整数视为 true，将零视为 false。相反， C#要求条件必须是布尔类型，不允许整数。

4.5.1 关系操作符和相等性操作符

关系和相等性操作符判断一个值是否大于、小于或等于另一个值。表 4.2 总结了所有关系和相等性操作符。它们都是二元操作符。

表 4.2 关系和相等性操作符

操作符	说明	示例
<	小于	`input < 9;`
>	大于	`input > 9;`
<=	小于或等于	`input <= 9;`
>=	大于或等于	`input >= 9;`
==	等于	`input == 9;`
!=	不等于	`input != 9;`

和其他许多编程语言一样，C#使用相等性操作符==来测试相等性。例如，判断 `input` 是否等于 9 要使用 `input == 9`。相等性操作符使用两个等号，赋值操作符则使用一个。C#的感叹号代表 NOT，所以用于测试不等性的操作符是`!=`。

关系和相等性操作符总是生成 `bool` 值，如代码清单 4.31 所示。

代码清单 4.31　将关系操作符的结果赋给 bool 变量

```
bool result = 70 > 7;
```

井字棋程序的完整代码清单使用相等性操作符判断玩家是否退出游戏。代码清单 4.32 的布尔表达式包含一个 OR（||）逻辑操作符，下一节将详细讨论它。

代码清单 4.32　在布尔表达式中使用相等性操作符

```
if (input.Length == 0 || input == "quit")
```

```
{
    Console.WriteLine($"玩家{currentPlayer}退出!!");
}
```

4.5.2 逻辑操作符

逻辑操作符获取布尔操作数并生成布尔结果。可以使用逻辑操作符来合并多个布尔表达式，从而构造更复杂的布尔表达式。逻辑操作符包括|，||，&，&&和^，对应 OR，AND 和 XOR（异或）。OR 和 AND 的|和&版本很少用，原因稍后讨论。

OR 操作符(||)

在代码清单 4.32 中，如果玩家输入 quit，或直接按 Enter 键而不输入任何值，就认为想要退出游戏。为了允许玩家以这两种方式退出，程序使用了逻辑 OR 操作符||。

||操作符对两个布尔表达式进行求值，任何一个为 true 就返回 true，如代码清单 4.33 所示。

代码清单 4.33　使用 OR 操作符

```
if ((hourOfTheDay > 23) || (hourOfTheDay < 0))
    Console.WriteLine("你输入了无效的时间。");
```

注意，使用布尔 OR 操作符时，不一定每次都会对操作符两边的表达式进行求值。和 C#的所有操作符一样，OR 操作符从左向右求值。所以，一旦左边求值为 true，那么右边就可以忽略。换言之，如果 hourOfTheDay 的值为 33，那么(hourOfTheDay > 23)会返回 true，所以 OR 操作符会忽略右边的表达式。这种**短路求值**方式同样适合布尔 AND 操作符。注意，本例的圆括号括号并非必须，因为逻辑操作符的优先级低于关系操作符。然而，为了清晰起见，还是可以将子表达式用圆括号括起来，这使初学者更容易理解。

AND 操作符(&&)

布尔 AND 操作符&&在两个操作数求值都为 true 的前提下才返回 true。任何操作数为 false，都会返回 false。代码清单 4.34 判断当前小时数是否大于 10 而且小于 24[①]。如果同时满足这两个条件，就输出一条消息表明当前是工作时间。和 OR 操作符一样，AND 操作符也并非每次都要对右边的表达式进行求值。只要左边的表达式返回 false，那么不管右边的操作数是什么，最终结果肯定为 false，所以“运行时”会忽略右边的操作数。

[①] 程序员典型的工作时间。

代码清单 4.34　使用 AND 操作符

```
if ((10 < hourOfTheDay) && (hourOfTheDay < 24))
    Console.WriteLine("嗨哟，嗨哟，我们去工作了。");
```

XOR 操作符(^)

^符号是异或（Exclusive OR，XOR）操作符。如果应用于两个布尔操作数，那么只有在两个操作数中仅有一个为 true 的前提下，XOR 操作符才会返回 true，如表 4.3 所示。

与布尔 AND 和 OR 操作符不同，布尔 XOR 操作符不支持短路运算。它始终都要检查两个操作数，因为除非确切知道两个操作数的值，否则不能判定最终结果。注意，如果将 XOR 操作符换成!=操作符，那么还是会得到表 4.3 的结果。

表 4.3　XOR 真值表

左操作数	右操作数	结果
True	True	False
True	False	True
False	True	True
False	False	False

4.5.3　逻辑取反操作符(!)

逻辑取反操作符（!）有时也称为 NOT 操作符，作用是反转一个 bool 数据类型的值。这是一元操作符，只需一个操作数。代码清单 4.35 演示了它如何工作，输出 4.16 展示了结果。

代码清单 4.35　使用逻辑取反操作符

```
bool valid = false;
bool result = !valid;
// 显示"result = True"
Console.WriteLine($"result = {result}");
```

输出 4.16

```
result = True
```

valid 最开始的值为 false。取反操作符对 valid 的值取反，将新值赋给 result。

4.5.4 条件操作符(?:)

可以使用**条件操作符**取代 if-else 语句来选择两个值中的一个。条件操作符同时使用一个问号和一个冒号，常规格式如下：

```
condition ? consequence : alternative
```

条件操作符是三元操作符，需要三个操作数：condition，consequence 和 alternative。类似于逻辑操作符，条件操作符也采用了某种形式的短路求值。如果 condition 求值为 true，那么条件操作符只求值 consequence；否则只求值 alternative。操作符的结果是被求值的表达式。

代码清单 4.36 展示了如何使用条件操作符。该程序的完整代码清单可以参考本书配套资源中的 Chapter04\TicTacToe.cs。

代码清单 4.36　条件操作符

```
public class TicTacToe
{
    public static void Main()
    {
        // 最开始将 currentPlayer 设为玩家 1
        int currentPlayer = 1;

        // ...

        for(int turn = 1; turn <= 10; turn++)
        {
            // ...

            // 交换玩家
            currentPlayer = (currentPlayer == 2) ? 1 : 2;
        }
    }
}
```

程序的作用是交换当前玩家。它检查当前值是否为 2。这是条件语句的 condition 部分。如果结果为 true，那么条件操作符返回 consequence 值 1；否则返回 alternative 值 2。和 if 语句不同，条件操作符的结果必须赋给某个变量（或作为参数传递），不能自成语句。

设计规范

CONSIDER using an `if/else` statement instead of an overly complicated conditional expression.

考虑使用 `if/else` 语句而不是过于复杂的条件表达式。

在C# 9.0之前，语言要求条件操作符中的输出部分（`consequence`和`alternative`表达式）具有一致的类型，而且在判定类型是否一致时不会检查表达式的上下文（包括最终要赋给的目标类型）。然而，C# 9.0 引入了对目标类型（target-typed）条件表达式的支持。因此，即使 `consequence` 和 `alternative` 表达式是不同的类型（例如 `string` 和 `int`），而且两者不能隐式转换为对方，只要两个类型都能隐式转换到目标类型，那么该语句仍然是允许的。例如，编译器现在允许写 `object result = condition ? "abc" : 123;`这样的语句，因为两个条件的输出类型都能隐式转换为 `object`。

4.6 用 null 编程

如第 3 章所述，虽然 `null` 可以非常有用，但它也带来了一些挑战。其中最明显的是，在调用对象的成员，或者将值从 `null` 更改为更适合当前情况的值之前，需要检查值是否为 `null`。

尽管可以使用相等性操作符甚至关系相等性操作符来检查 `null`，但还有其他几种方法，包括 C# 7.0 对 `is null` 的支持以及 C# 10.0 对 `is not null` 的支持。除此之外，有几个操作符专门用于处理可能为 `null` 的值，其中包括空合并操作符（以及 C# 8.0 的空合并赋值操作符）和空条件操作符。甚至有一个操作符可以告诉编译器：虽然编译器自己没有把握，但你可以确定一个值不是`null`——这就是空包容操作符（一元后缀`!`操作符）。让我们从最简单的开始，即检查一个值是否为 `null`。

4.6.1 检查是否为 null

有多种方式可以检查 `null`，如表 4.4 所示。对于其中每个代码段，都假设之前已经声明好了变量：

```
string? uriString = null;
```

表 4.4 检查 `null`

说明	示例
`is null` 操作符模式匹配 `is` 操作符提供了多种检查 `null` 的方法。从 C# 7.0	`// 1.` `if (uriString is null)` `{`

开始，可以使用 is null 表达式来检查一个值是否为 null。这是检查值是否为 null 的一种非常简单明了的方式。	Console.WriteLine("Uri 为 null"); }
is not null 操作符模式匹配 类似地，C# 9.0 增加了对 is not null 的支持。如果要检查是否不为空，这就是首选方式。	// 2. if (uriString is not null) { Console.WriteLine("Uri 不为空"); }
相等/不相等 相等性和不相等性操作符适用于所有版本的 C#。 此外，用这种方式检查 null 具有较好的可读性。 这种方式的唯一缺点是可能要求重写相等性/不相等性操作符，从而可能引入轻微的性能损失。	// 3. if (uriString == null) { Console.WriteLine("Uri 为 null"); } if (uriString != null) { Console.WriteLine($"Uri 是: {uriString}"); }
is object is object 是 C# 1.0 就有的一个功能，用于判断操作数是否不为 null，等价于 is not null，但明显不如后者清晰。is object 比下面的 is {}表达式更好用，因为当操作数是非空的值类型时，它会发出警告。这很合理，既然操作数不能为 null，检查 null 的意义何在呢？	// 4. int number = 0; if ((uriString is object) // Warning CS0183:给定表达式 // 始终不为 null && (number is object)) { Console.WriteLine($"Uri 是: {uriString}"); }
is { }操作符模式匹配 C# 8.0 新增了属性模式匹配表达式<*操作数*> is { }，它提供了与 is object 几乎相同的功能。除了可读性较差，它还有一个明显的缺点，也就是为不可空的值类型表达式使用 is {}，它是不会发出警告的，但 is object 会。但是，对于不可空的值类型来说，最好是发出警告，因为检查不可空	// 5. if (uriString is { }) { Console.WriteLine($"Uri 是: {uriString}"); }

的值类型是否为 null 是没有意义的。因此，建议优先使用 is object 而不是 is { }。	
ReferenceEquals() object.ReferenceEquals()用于执行引用相等性检查。虽然对于一个如此简单操作而言，这个名字显得过长，但它适用于所有版本的 C#，且具有不允许重写的优点。所以，它始终是“名符其实”的。	`// 6.` `if (ReferenceEquals(` `  uriString, null))` `{` `  Console.WriteLine(` `  "Uri 为 null");` `}`

有这么多检查 null 值的方式，自然会带来一个问题：哪种更好？如果使用的是现代版本的 C#，那么应该首选 C# 7.0 加强的 is null 以及 C# 9.0 新增的 is not null 语法。

当然，如果必须使用 C# 6.0 或更早的版本编程，那么除了用 is object 检查非空，唯一的选择就是相等/不相等性操作符。is object 要稍微好一些，因为不可能改变 is object 表达式的行为，所以自然不可能造成性能损失（虽然很轻微）。这使 is {}几乎没了用武之地。ReferenceEquals()很少有人使用，但它允许比较未知数据类型的值，这在实现相等性操作符的自定义版本时相当有用。详情参见第 10 章的“操作符重载”一节。

表 4.4 有几行使用了模式匹配（也就是那些 is 和 is not 等），这个概念将在第 7 章的“模式匹配”一节进行更详细的介绍。

4.6.2 空合并操作符和空合并赋值操作符(??，??=)

空合并操作符??能简单地表示：“如果这个值为空，就使用另一个值”，其形式如下。

```
expression1 ?? expression2
```

空合并操作符支持短路求值。如果 expression1 不为 null，就返回 expression1 的值，另一个表达式不求值。如果 expression1 求值为 null，就返回 expression2 的值。和条件操作符不同，空合并操作符是二元操作符。

代码清单 4.37 展示了使用空合并操作符的一个例子。

代码清单 4.37　空合并操作符

```
string? fullName = GetSaveFilePath();
// ...

// 空合并操作符
```

```
string fileName = GetFileName() ?? "config.json";
string directory = GetConfigurationDirectory() ??
    GetApplicationDirectory() ??
    Environment.CurrentDirectory;

// 空合并赋值操作符
fullName ??= $"{ directory }/{ fileName }";

// ...
```

在这个例子中，如果 GetFileName()返回 null，就用空合并操作符将 fullName 设为"config.json"。如果 fileName 不为 null，那么直接将 GetFileName()的返回值赋给 fullName。

空合并操作符能完美“链接”。例如，对于表达式 x ?? y ?? z ，x 不为 null 将返回 x；否则，y 不为 null 将返回 y；否则返回 z。也就是说，从左向右选出第一个非空表达式。之前所有表达式都为空，就选择最后一个。代码清单 4.27 在对 directory 进行赋值的时候用到了这个技术。

C# 8.0 引入了空合并操作符与赋值操作符的组合，增加了**空合并赋值操作符**??=。使用这个操作符，首先判断左侧的变量是否为 null。如果是，就将右侧表达式的值赋给它。在代码清单 4.37 中，我们在对 fullName 赋值时使用了该操作符。注意，??=操作符的左操作数必须是一个变量、属性或索引器元素。

4.6.3 空条件操作符(?.)

由于经常需要在调用成员之前检查 null，所以从 C# 6.0 开始引入了**空条件操作符**?.，如代码清单 4.38 所示。[1]

代码清单 4.38　空条件操作符

```
string[]? segments = null;
string? uri = null;
// ...
int? length = segments?.Length;
// ...
if (length is not null && length != 0)
{
    uri = string.Join('/', segments!);
}
if (uri is null || length is 0)
```

[1] 这段代码其实可以用一个 **using 语句**来改进，但由于尚未讲到那里，所以暂时放弃。另外，空条件操作符也称为“空传播操作符”。

```
{
    Console.WriteLine("没有更多区段可以合并了。");
}
else
{
    Console.WriteLine(
    $"Uri: {uri}");
}
```

对于空条件操作符的这个例子（int? length = segments?.Length），在调用方法或属性（本例是 Length 属性）前，会首先检查操作数（segments）是否为 null。它在逻辑上等价于以下代码（虽然原始的语法只求值 segments 一次）：

```
int? length = (segments != null) ? (int?)segments.Length : null
```

空条件操作符的一个重点在于，它会始终生成一个可空值。在本例中，即使 string.Length 成员生成一个不可空的 int，用空条件操作符来调用 Length 也会生成一个可空 int（即 int?）。

还可以访问数组时使用空条件操作符。例如，uriString = segments?[0] 在 segments 数组不为 null 的情况下获得它的第一个元素。然而，很少使用空条件操作符来访问数组，因为它要满足以下条件：不知道操作数是否为 null，但又知道元素的数量，或者至少知道特定元素是否存在。

空条件操作符最方便的地方在于可以“链接”（不管用不用更多的空合并操作符）。例如，在以下代码中，只有在 segments 和 segments[0]都非空的前提下才会调用 ToLower()和 StartWith()。

```
segments?[0]?.ToLower().StartsWith("file:");
```

当然，在这个例子中，我们假设 segments 的元素可能为空，所以应该更准确地这样写它的声明（要求至少 C# 8.0）：

```
string?[]? segments;
```

它的意思是说，segments 数组是可空的，而且它的每个元素都是一个可空的 string。

空条件表达式链接起来后，如果第一个操作数为空，那么表达式求值会被短路，调用链中不再发生其他调用。还可以在表达式末尾再链接一个空合并操作符。这样一来，如果操作数为 null，那么就可以指定使用一个默认值。

```
string uriString = segments?[0]?.ToLower().StartsWith(
    "file:") ?? "intellitect.com";
```

注意，空合并操作符??所生成的数据类型应该是非空的（换言之，假设操作符的右操作数——本例是"intellitect.com"——非空。这很正常，因为如果右操作数也允许为空，那

么空合并操作符就没有意义了）。

但是，注意不要遗漏额外的 null 值。例如，假定（只是假定）ToLower()也返回 null，那么会发生什么？这样在调用 StartsWith()时会抛出 NullReferenceException 异常。但这并不是说一定要使用一个空条件操作符链，而是说应关注程序逻辑。在本例中，由于 ToLower()永不为空，所以不需要额外的空条件操作符。

虽然有点怪（和其他操作符行为相比），但只有在调用链最后才会生成一个可空值类型的值。结果是假如用点（.）操作符来调用 Length 的成员，那么只允许调用它的 int（而非 int?）的成员。但是，将 segments?.Length 放到圆括号中（从而先返回一个 int?），就可以在返回的 int?上调用 Nullable<T>类型的特殊成员（HasValue 和 Value）了。

换言之，segments?.Length.Value 是无法通过编译的，因为 int（Length 返回的数据类型）没有名为 Value 的成员。然而，通过改变求值优先级，修改成(segments?.Length).Value，就可以避免编译错误了，因为(segments?.Length)会先返回一个 int?，所以可以调用它的 Value 属性。

4.6.4 空包容操作符(!)

在之前的代码清单 4.38 中，注意 Join()调用在 segments 后面包含了一个感叹号。

```
uri = string.Join('/', segments!);
```

在执行上述代码的时候，我们已经将 segments?.Length 赋给 length 变量，而由于 if 语句已经验证了 length 不为 null，所以我们知道 segments 也不可能为 null。

```
int? length = segments?.Length;
// ...
if (length is not null && length != 0){ }
```

但是，编译器没有能力做同样的判断。由于 Join()要求一个非空字符串数组，当传递一个声明为可空的 segments 变量时，它会发出警告。为了避免该警告，我们可以从 C# 8.0 开始使用空包容操作符（!）。它告诉编译器，作为程序员，我们更清楚 segments 变量不可能为 null。这样一来，在编译时，编译器就知道我们比它更清楚，所以会消除警告（尽管“运行时”仍会在程序执行时检查我们的这个断言）。

注意，虽然空条件操作符会检查 segments 数组是否为 null，但它会不检查其中有多少个元素，也不会检查每个元素是否为 null。

在第 1 章中，当我们将 Console.ReadLine()的返回值给一个字符串时，遇到了警告 CS8600，即“将 null 文本或可能的 null 值转换为不可为 null 类型”。这是因为 Console.ReadLine()返回的是一个 string?，而不是一个非空字符串。事实上，Console.ReadLine()仅在输入被重定向时才有可能返回 null，而这并不是我们这些入门程序所期望的用例。不过，编译器不知道这一点。为了消除警告，我们可以为 Console.ReadLine()的返回值使用空包容操

作符，表示我们更清楚返回值不会为 null（如果真的为 null，那么也可以放心地抛出空引用异常）。

```
string text = Console.ReadLine()!;
```

> **高级主题：空条件操作符应用于委托**
>
> 空条件操作符本身已是极好的功能。与委托调用配合，它更是解决了自 C# 1.0 就存在的一个痛点。在代码清单 4.39 展示的传统方式中，我们先将 PropertyChange 事件处理程序赋给一个局部拷贝（propertyChanged），再执行空检查，非空则引发事件。这是以线程安全的方式调用（invoke）事件处理程序的最简单方式，可防范在空检查和引发事件之间发生事件被取消订阅的风险。[①]
>
> **代码清单 4.39　委托调用的传统方式**
>
> ```
> System.EventHandler propertyChanged = PropertyChanged;
> if (propertyChanged != null)
> {
>     propertyChanged(this, new System.EventArgs());
> }
> ```
>
> 但该模式并不直观，经常会有开发人员不遵守。结果就是抛出让人摸不着头脑的 NullReferenceException。幸好，空条件操作符解决了该问题。现在，代码清单 4.39 所展示的对委托值进行的 null 检查可以修改成以下更优雅的代码。
>
> ```
> PropertyChanged?.Invoke(propertyChanged(
>     this, new PropertyChangedEventArgs(nameof(Name)));
> ```
>
> 事件本质上就是委托[②]，所以通过空条件操作符和 Invoke()来调用（invoke）委托总是可行的。

[①] 译注：“调用一个委托实例”中的“调用”对应的是 invoke，理解为“唤出”更恰当。它和“在一个对象上调用方法”中的“调用”稍有不同，后者对应的是 call。在英语的语境中， invoke 和 call 的区别在于，在执行一个所有信息都已知的方法时，用 call 比较恰当。这些信息包括要引用的类型，方法的签名以及方法名。但是，在需要先“唤出”某个东西来帮你调用一个信息不明的方法时，用 invoke 就比较恰当。但是，由于两者均翻译为“调用”不会对读者的理解造成太大的困扰，所以本书仍然采用约定俗成的方式来进行翻译，只是在必要的时候附加英文原文提醒你区分。

[②] 译注：更准确地说，CLR 事件模型以委托为基础。委托本质上是一种类型， 提供了调用（invoke）回调方法的一种类型安全的方式。参见《CLR via C#》第 4 版，第 11 章。

4.7 按位操作符(<<，>>，|，&，^，~)

几乎所有编程语言都提供了一套按位操作符来处理值的二进制形式。

初学者主题：位和字节

在计算机中，所有值都表示成 1 和 0 的二进制形式，这些 1 和 0 称为**二进制位**（bit）。8 位一组称为**字节**（byte）。一个字节中每个连续的位都对应 2 的一个乘幂。其中，最右边的位对应 2^0，最左边的对应 2^7，如图 4.1 所示。

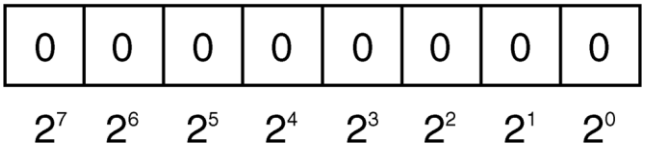


图 4.1 对应的占位值

在许多情况下，尤其是在操作低级设备或系统服务的时候，信息是以二进制数据的形式获取的。操作这些设备和服务需要处理二进制数据。

如图 4.2 所示，每个框都对应 2 的某个乘幂。字节（8 位构成的一个数）的值是设为 1 的所有位的 2 的乘幂之和。

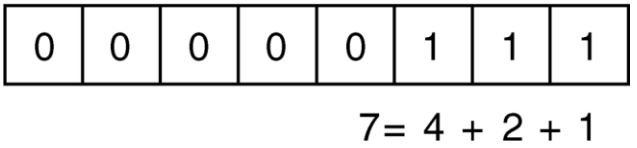


图 4.2 如何计算一个无符号字节的值

对于有符号的数，二进制转换则有很大的不同。有符号的数（`long`，`short`，`int`）使用 2 的补数记数法表示。这确保了将负数加到正数上时，上述加法运算能照常进行，就好比两个操作数都是正数一样。使用这种记数法，负数在行为上有别于正数。负数通过最左边的 1 来标识。如果最左边的位置包含 1，就要将含有 0 的位置加到一起，而不是将含有 1 的位置加到一起。每个位置都对应负的“2 的乘幂”。此外，最后还要在结果上减 1。图 4.3 对此进行了演示。

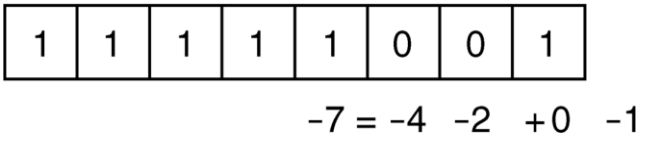


图 4.3 计算有符号字节的值

> 所以，1111 1111 1111 1111 对应-1（因为最后要减 1），1111 1111 1111 1001 对应-7，而 1000 0000 0000 0000 对应 16 位整数能容纳的最小负值。

4.7.1 移位操作符(<<，>>，<<=，>>=)

有的时候，我们需要将一个数的二进制值向右或向左移位。左移时，所有位都向左移动由操作符右侧的操作数指定的位数。左移后，在右边留下的空位由零填充。右移位操作符原理相似，只是朝相反方向移位；但是，如果是负数，左侧要填充 1 而非 0。两个移位操作符是>>和<<，分别称为**右移位**和**左移位**操作符。除此之外，还有**复合移位和赋值**操作符<<=和>>=。

例如，int 值-7 的二进制形式为 1111 1111 1111 1111 1111 1111 1111 1001。代码清单 4.40 使其右移 2 个位置，结果如输出 4.17 所示。

代码清单 4.40　使用右移位操作符

```
int x;
x = (-7 >> 2); // 11111111111111111111111111111001 变成
               // 11111111111111111111111111111110
// 输出"x = -2。"
Console.WriteLine($"x = {x}。");
```

输出 4.17

```
x = -2。
```

在这个例子中，右移位时，最右侧的位从边界处“离开”，左侧的负数位标识符也右移两个位置，腾出来的空白位置用 1 填充。最终结果是-2（再次提醒要多减一个 1）

虽然坊间传说 x << 2 比 x * 4 快，但不要将移位操作符用于乘除法。70 年代的一些 C 编译器可能确实如此，但现代微处理器都对算术运算进行了完美的优化。通过移位进行乘除令人迷惑，而且假如维护代码的人忘记移位操作符的优先级低于算术操作符，还很容易造成错误。

4.7.2 按位操作符(&，|，^)

我们有时需要对两个操作数执行逐位（bit-by-bit）的逻辑运算，比如 AND，OR 和 XOR 等，这分别是用&、|和^操作符来实现的。

> **初学者主题：理解逻辑操作符**
>
> 假定有如图 4.4 所示的两个数，按位操作符会从最左边的位开始逐位进行逻辑运算，

直到最右边的位为止。值 1 被视为 true，值 0 被视为 false。

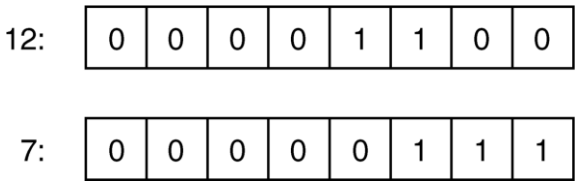


图 4.4 12 和 7 的二进制形式

所以，对图 4.4 的两个值执行按位 AND 运算，会逐位比较第一个操作数（12）和第二个操作数（7），得到二进制值 000000100，也就是十进制 4。另外，这两个值的按位 OR 运算结果是 00001111，也就是十进制 15。XOR 结果是 00001011，也就是十进制 11（记住，异或运算是指两个操作数中，有一个而且仅有一个为 true，结果才会为 true）。

代码清单 4.41 演示了如何使用这些按位操作符，结果如输出 4.18 所示。

代码清单 4. 41　使用按位操作符

```
byte and, or, xor;
and = 12 & 7;   // and = 4
or = 12 | 7;    // or = 15
xor = 12 ^ 7;   // xor = 11
Console.WriteLine( $"""
        and = { and }
        or = { or }
        xor = { xor }
        """);
```

输出 4. 18

```
and = 4
or = 15
xor = 11
```

在代码清单 4.41 中，值 7 称为**掩码**（mask），作用是通过特定的操作符表达式，公开（expose）或消除（eliminate）第一个操作数中特定的位。注意，和 AND（&&）操作符不同，&操作符两边都要求值，即使左边为 false。类似地，OR 操作符的|版本也不会进行“短路求值”。即使左边的操作数为 true，右边也要求值。总之，AND 和 OR 操作符的按位版本不进行“短路求值”。

为了将一个数转换成二进制，需要遍历它的每一位。代码清单 4.42 展示了如何将整数转换成二进制形式的字符串，输出 4.19 展示了结果。

代码清单 4.42　获取二进制形式的字符串表示

```
public class BinaryConverter
{
    public static void Main()
    {
        const int size = 64;
        ulong value;
        char bit;

        Console.Write("输入一个整数: ");
        // 使用 long.Parse()来支持负数，
        // 假设对 ulong 进行 unchecked 赋值，
        // 如果 ReadLine 返回 null，那么使用"42"作为默认输入
        value = (ulong)long.Parse(Console.ReadLine() ?? "42");

        // 将初始掩码(mask)设为 100000...0
        ulong mask = 1UL << size - 1;

        for(int count = 0; count < size; count++)
        {
            bit = ((mask & value) != 0) ? '1' : '0';
            Console.Write(bit);
            // 掩码右移 1 位
            mask >>= 1;
        }
        Console.WriteLine();
    }
}
```

输出 4.19

```
输入一个整数: 42
0000000000000000000000000000000000000000000000000000000000101010
```

注意，每次 for 循环（稍后就会讨论）都使用右移位赋值操作符（>>=）创建与变量 value 中的每一位对应的掩码。使用按位操作符&，可以判断一个特定的位是否已设置（是否设为 1）。如果掩码测试生成非零结果（测试结果为 true），就将 1 写到控制台，否则写 0。这样可真实反映一个无符号长整数（ulong）的二进制形式。

注意，(mask & value) != 0 中的圆括号是必须的，因为不相等测试的优先级高于 AND 操作符。不显式添加圆括号，就相当于 mask & (value != 0)。这没有任何意义，因为&左侧

是一个 ulong，右侧是一个 bool。

本例仅供参考，有内建的 CLR 方法 System.Convert.ToString(value, 2)可以直接执行这个转换。第二个参数指定进制（2 代表二进制，10 代表十进制，16 代表十六进制）。

4.7.3 按位复合赋值操作符(&=，|=，^=)

一点儿都不奇怪，按位操作符还可以和赋值操作符合并，即&=，|=和^=。例如，可以让变量与一个数进行 OR 运算，结果赋回初始变量，如代码清单 4.43 和输出 4.20 所示。

代码清单 4.43　使用逻辑赋值操作符

```
byte and = 12, or = 12, xor = 12;
and &= 7;   // and = 4
or |= 7;    // or = 15
xor ^= 7;   // xor = 11
Console.WriteLine( $"""
        and = {and}
        or = {or}
        xor = {xor}
        """);
```

输出 4.20

```
and = 4
or = 15
xor = 11
```

使用 fields &= mask 这样的表达式将位映射（bitmap）与掩码合并，会从 fields 中清除 mask 中没有设置的位。相反，fields &= ~mask 从 fields 中清除 mask 中已设置的位。记住，所谓“清除”一位，是指将该 bit 设为 0（置 0）。

4.7.4 按位取反操作符

按位取反操作符反转操作数的每一个二进制位，操作数可以是 int，uint，nint，nuint，long 和 ulong 类型。例如，~1 返回 1111 1111 1111 1111 1111 1111 1111 1110（或十进制-2），而~(1<<31)返回 0111 1111 1111 1111 1111 1111 1111 1111（或十进制 2 147 483 647，即 Int32.MaxValue）

4.8 控制流语句(续)

更详细地探讨了布尔表达式之后，就可以更清楚地描述 C#支持的控制流语句。有经验的程序员已熟悉了其中许多语句，所以可快速浏览本节内容，找出 C#特有的信息。尤其关注 foreach 循环，它对许多程序员来说是新的。

4.8.1 while 和 do/while 循环

目前学习的都是只执行一遍的程序。但是，计算机的关键优势之一是能多次执行相同的操作。为此，需要创建指令循环。本节讨论的第一个指令循环是 while 循环，它是最简单的条件循环。while 语句的常规形式如下：

```
while (condition)
   statement
```

其中，条件（condition）必须是布尔表达式，只要它求值为 true，作为循环主体的语句（statement）就会反复执行。条件求值为 false，就跳过循环主体，从它之后的语句执行。注意循环主体会一直执行，即使这个过程中导致条件变成 false。除非回到“循环顶部”重新求值条件，而且结果是 false，否则循环不会退出。代码清单 4.44 用一个斐波那契计算器演示了 while 语句的用法。

代码清单 4.44　while 循环示例

```
decimal current;
decimal previous;
decimal temp;
decimal input;

Console.Write("输入一个正整数:");

// decimal.Parse 将 ReadLine 的返回结果转换为一个十进制数，
// 如果 ReadLine 返回 null，就用"42"作为默认值。
input = decimal.Parse(Console.ReadLine() ?? "42");

// 将 current 和 previous 初始化为 1，这是
// 斐波那契数列前两个固定的数。
current = previous = 1;

// 判断数列中当前的斐波那契数是否
// 小于用户输入的数
while (current <= input)
{
    temp = current;
    current = previous + current;
    previous = temp; // 即使上一个语句造成 current 大于 input，
                     // 也会执行这个语句。
}

Console.WriteLine($"下一个斐波那契数是{ current }");
```

斐波那契数是**斐波那契数列**的成员，数列中所有数都是数列中前两个数之和。数列最开头

两个数固定为 1 和 1。代码清单 4.44 中提示用户输入一个正整数，使用 while 循环发现比输入的数大的第一个斐波那契数。

> **初学者主题：何时使用 while 循环**
>
> 本章剩余部分会讲到造成代码块反复执行的其他循环构造。**循环主体**或**循环体**是指 while 结构中执行的语句（通常是一个代码块）。这是因为在达成退出条件之前，循环主体代码会一直“迭代”。应该理解在什么时候选择什么循环构造。如果条件为 true 就一直执行某个操作，就选择 while。for 主要用于重复次数已知的循环，比如从 0 到 *n* 计数。do/while 类似于 while 循环，区别在于循环主体至少执行一次。

do/while 循环与 while 循环非常相似，只是最适合需要循环 1～*n* 次的情况，而且 *n* 在循环开始前无法确定。经常用这个模式提示用户输入。代码清单 4.45 是从井字棋程序中提取出来的。

代码清单 4.45　do/while 循环示例

```
// 反复提示玩家落子，直到
// 输入棋盘上的一个有效位置。
bool valid;
do
{
    valid = false;

    // 请求当前玩家落子
    Console.Write($"玩家{currentPlayer}: 输入落子:");
    string? input = Console.ReadLine();

    // 检查当前玩家的输入
    // ...

} while(!valid);
```

代码清单 4.45 在每次迭代（循环刚开始的时候）[1]将 valid 设为 false。接着，提示并获取用户输入的一个数。虽然这部分在代码中省略了，但接下来的操作是检查输入是否正确。如正确，就将 true 赋给 valid。由于代码使用 do/while 而不是 while 语句，所以至少提示

[1] 译注：每一次循环都称为一次“迭代”。

用户输入一次。

do/while 循环的常规形式如下：

```
do
   statement
while (condition);
```

和所有控制流语句一样，循环主体通常是一个代码块，以便执行多个语句。但是，完全可以将单个语句作为循环主体，但加了标签的语句（参见本章稍后的“基本 switch 语句”一节）和局部变量声明除外。

4.8.2 for 循环

for 循环反复执行代码块，直至满足指定条件。这和 while 循环差不多。区别在于，for 循环有一套内建的语法规定了如何初始化、递增以及测试一个计数器的值。该计数器称为**循环变量**。由于循环语法中专门有一个位置是为递增/递减操作保留的，所以递增/递减操作符经常作为 for 循环的一部分使用。

代码清单 4.46 展示了如何使用 for 循环显示整数的二进制形式。输出 4.21 展示了结果。

代码清单 4.46　使用 for 循环

```
const int size = 64; // 也可以从 sizeof(ulong)取得
ulong value;
char bit;

Console.Write("输入一个整数: ");
// 使用 long.Parse()来支持负数
// 假设对 ulong 进行 unchecked 赋值，
// 如果 ReadLine 返回 null，那么使用"42"作为默认输入
value = (ulong)long.Parse(Console.ReadLine() ?? "42");

// 将初始掩码(mask)设为 100000...0
ulong mask = 1UL << size - 1;
for(int count = 0; count < size; count++)
{
    bit = ((mask & value) > 0) ? '1' : '0';
    Console.Write(bit);
    // 掩码右移 1 位
    mask >>= 1;
}
```

输出 4.21

```
输入一个整数: -42
1111111111111111111111111111111111111111111111111111111111010110
```

代码清单 4.46 执行位掩码 64 次，对用户输入的数中的每一位都应用一次。for 循环头包含三个部分。第一部分声明并初始化变量 count，第二部分描述 for 循环主体的执行条件，第三部分描述如何更新循环变量。for 循环的常规形式如下：

```
for (initial ; condition ; loop)
   statement
```

下面解释了 for 循环的各个部分。

- *initial*（初始化）负责首次迭代前的初始化操作。在代码清单 4.46 中，它声明并初始化 count 变量。*initial* 表达式不一定非要声明新变量。例如，完全可以事先声明好变量，在 for 循环中只是初始化它。还可以完全省略该部分。如果在这里声明变量，其作用域仅限于 for 语句头部和主体。
- *condition*（条件）指定循环结束条件。条件为 false 就终止循环，这和 while 循环一样。只有条件求值为 true 才会执行 for 循环主体。本例在 count 大于或等于 64 时退出循环。
- *loop*（循环）表达式在每次迭代后求值。本例的循环表达式 count++会在 mask 右移位（mask >>= 1）之后、在对条件求值之前执行。第 64 次迭代时，count 递增到 64，造成条件变成 false，因而终止循环。
- *statement* 是在条件表达式为 true 时执行的“循环主体”代码。

代码清单 4.46 的 for 循环的执行步骤可用以下伪代码表示。

1. 声明 count 并将其初始化为 0。
2. 如果 count 小于 64，跳到步骤 3；否则跳到步骤 7。
3. 计算 bit 并显示它。
4. 对 mask 执行右移位。
5. count 递增 1。
6. 跳回步骤 2。
7. 继续执行循环主体之后的语句。

for 语句头部三部分均可省略。for(;;){ ... }完全有效，只要有办法从循环中退出以避免无限循环（缺失的条件默认为常量 true）。

initial 和 loop 表达式支持多个循环变量，如代码清单 4.47 和输出 4.22 所示。

代码清单 4. 47　使用多个表达式的 for 循环

```
for (int x = 0, y = 5; ((x <= 5) && (y >= 0)); y--, x++)
{
    Console.Write($"{x}{((x > y) ? '>' : '<')}{y}\t");
}
```

输出 4. 22

```
0<5     1<4     2<3     3>2     4>1     5>0
```

在这个例子中，`initial` 部分声明并初始化两个循环变量。看起来复杂，但起码像是在一个语句中声明多个局部变量，多少有点正常。`loop` 部分看起来则有点不太正常，因为它包含以逗号分隔的表达式列表，而非单一表达式。

设计规范

CONSIDER refactoring the method to make the control flow easier to understand if you find yourself writing `for` loops with complex conditionals and multiple loop variable.

如果必须写包含复杂条件和多个循环变量的 `for` 循环，那么**考虑**重构方法使控制流更容易理解。

任何 `for` 循环都可以改写成 `while` 循环。

```
{
   initial;
   while (condition)
   {
      statement;
      loop;
   }
}
```

设计规范

DO use the `for` loop when the number of loop iterations is known in advance and the “counter” that gives the number of iterations executed is needed in the loop.

事先知道循环次数，而且循环中要用到控制循环次数的“计数器”，**要**使用 `for` 循环。

DO use the `while` loop when the number of loop iterations is not known in advance and a counter is not needed.

事先不知道循环次数，而且不需要计数器，**要**使用 `while` 循环。

4.8.3 foreach 循环

C#最后一个循环语句是 foreach，它遍历数据项集合，设置循环变量来依次表示其中每一项。循环主体可以对集合中的所有数据项执行指定操作。foreach 循环的特点是每一项必然会被遍历一次：不会像其他循环那样出现计数错误，也不可能越过集合边界。

foreach 语句的常规形式如下：

```
foreach(type variable in collection)
    statement
```

下面解释了 foreach 语句的各个部分。

- *type* 是代表 collection 中每一项的 *variable* 的数据类型。可以将类型设为 var，编译器将根据集合类型推断数据项类型。
- *variable* 是一个只读变量，foreach 循环自动将 *collection* 中的下一项赋给它。该变量的作用域限于循环主体。
- *collection* 是代表多个数据项的表达式，比如数组。
- *statement* 是每次迭代都要执行的循环主体。

代码清单 4.48 和输出 4.23 展示了一个简单 foreach 循环。

代码清单 4. 48　使用 foreach 循环判断剩余落子

```
// 像下面这样硬编码初始棋盘
// ---+---+---
//  1 | 2 | 3
// ---+---+---
//  4 | 5 | 6
// ---+---+---
//  7 | 8 | 9
// ---+---+---
char[] cells = { '1', '2', '3', '4', '5', '6', '7', '8', '9' };

Console.Write("可能的落子如下所示: ");

// 输出初始可能的落子
foreach (char cell in cells)
{
    if (cell != 'O' && cell != 'X')
    {
        Console.Write($"{cell} ");
    }
}
```

输出 4.23

```
可能的落子如下所示：1 2 3 4 5 6 7 8 9
```

执行到 foreach 语句时，将 cells 数组的第一项，也就是值'1'赋给 cell 变量。然后执行 foreach 循环主体。if 语句判断 cell 的值是否等于'O'或'X'。如果两者都不是，就在控制台上输出 cell 的值。下次循环迭代，会将数组的下个值赋给 cell，并以此类推。

必须记住，foreach 循环期间禁止修改循环变量（本例是 cell）。

> **初学者主题：何时使用 switch 语句**
>
> 有的时候，我们需要在连续几个 if 语句中比较同一个值，如代码清单 4.49 的 input 变量所示。
>
> **代码清单 4.49　用 if 语句检查玩家输入**

```
// ...

// 检查当前玩家的输入
if ((input == "1") ||
    (input == "2") ||
    (input == "3") ||
    (input == "4") ||
    (input == "5") ||
    (input == "6") ||
    (input == "7") ||
    (input == "8") ||
    (input == "9"))
{
    // 根据玩家的输入保存/落子
    // ...

}
else if ((input.Length == 0) || (input == "quit"))
{
    // 重试或退出
    // ...
}
else
{
    Console.WriteLine($"""
          错误：  输入 1-9 的值。
          按 Enter 键退出。
          """);
}
```

```
// ...
```

上述代码验证用户输入的文本，确定是一步有效的井字棋落子。例如，假定 input 的值是 9，那么程序不得不执行 9 次求值。显然，更好的思路是只在一次求值之后就跳转到正确的代码。在这种情况下，应该使用 switch 语句。

4.8.4 基本 switch 语句

需要将一个值和多个常量值比较的时候，基本 switch 语句比 if 语句更容易理解，下面是它的常规形式。

```
switch (expression)
{
   case constant:
      statements
   default:
      statements
}
```

下面解释了 switch 语句的各个部分。

- *expression* 是要和不同常量比较的值。该表达式的类型决定了 switch 的“主导类型”（governing type）。允许的主导类型包括 bool、sbyte、byte、short、ushort、int、uint、nint、nuint、long、ulong、char、任何枚举（enum）类型（详情参见第 9 章）、上述所有值类型的可空版本以及 string。
- *constant* 是和主导类型兼容的任何常量表达式。
- 一个或多个 case 标签（或 default 标签），后跟一个或多个语句（称为一个 **switch 小节**，即 switch section）。在上面的“常规形式”中，只显示了两个 switch 小节。代码清单 4.50 的 switch 语句则显示了三个。
- *statements* 是在 expression 的值等于某个标签指定的 *constant* 值时执行的一个或多个语句。这组语句的结束点必须“不可到达”[①]。换言之，不能“直通”或“贯穿”到下个 switch 小节。所以，最后一个语句通常是跳转语句，比如 break，return 或 goto。

[①] 译注：C#语言规范对结束点和可到达性的解释是这样的：“每个语句都有一个结束点 (end point)。直观地讲，语句的结束点是紧跟在语句后面的那个位置。复合语句（包含嵌入语句的语句）的执行规则规定了当控制到达一个嵌入语句的结束点时所采取的操作。例如，当控制到达块中某个语句的结束点时，控制就转到该块中的下一个语句。如果执行流可能到达某个语句，则称该语句可到达 (reachable)。相反，如果某个语句不可能被执行，则称该语句不可到达 (unreachable)。”

设计规范

DO NOT use `continue` as the jump statement that exits a `switch` section. This is legal when the `switch` is inside a loop, but it is easy to become confused about the meaning of `break` in a later `switch` section.

不要使用 `continue` 作为跳转语句退出 `switch` 小节。如果 `switch` 在循环中，那么这样写虽然合法，但很容易对之后的 `switch` 小节中同样出现的 `break` 产生困惑。

`switch` 语句至少应该有一个 `switch` 小节，`switch(x){}`虽然合法，但会产生一个警告。另外，虽然一般情况下应避免省略大括号，但一个例外是应省略 `case` 和 `break` 语句的大括号，因为这两个关键字本身就指示了块的开始和结束。

代码清单 4.50 的 `switch` 语句在语义上等价于代码清单 4.49 的一系列 `if` 语句。

代码清单 4.50　将 `if` 语句替换成 `switch` 语句

```
public static bool ValidateAndMove(
        int[] playerPositions, int currentPlayer, string input)
{
    bool valid = false;

    // 检查当前玩家的输入
    switch (input)
    {
        case "1":
        case "2":
        case "3":
        case "4":
        case "5":
        case "6":
        case "7":
        case "8":
        case "9":
            // 根据玩家的输入保存/落子
            // ...
            valid = true;
            break;
        case "":
        case "quit":
            valid = true;
            break;
        default:
```

```
                // 如果和其他 case 都不匹配，表明输入无效
                Console.WriteLine(
                "错误：  输入 1-9 的值。 "
                + "按 Enter 键退出。");
                break;
        }
        return valid;
    }
```

代码清单 4.50 中的 input 是要测试的表达式。由于 input 是字符串，所以主导类型是 string。如果 input 的值是"1"、"2"、……"9"，那么落子有效（valid = true）。接着，可以更改相应的单元格，使之与当前用户的标记（X 或 O）匹配。在 switch 中遇到 break 语句，会立即跳转到 switch 语句之后的语句。

下一个 switch 小节描述如何处理空白字符串""或"quit"。input 等于这两个值之一，那么也将 valid 设为 true。如果没有其他 case 标签与测试表达式匹配，就执行 switch 的 default 小节。

> **语言对比：C++ —— switch 语句贯穿**
>
> 在 C++中，如果 switch 小节不以跳转语句结尾，控制会“贯穿”（直通）到下个 switch 小节并执行其中的代码。由于容易出错，所以 C#不允许控制从一个 switch 小节自然贯穿到下一个。C#的设计者认为这样可以更好地防止 bug，并增强代码的可读性。如果希望 switch 小节执行另一个 switch 小节中的代码，要显式使用 goto 语句来实现，详情参见稍后的“goto 语句”小节。

switch 语句有几点要注意。

- 无任何小节的 switch 语句会产生编译器警告，但语句仍能通过编译。
- 各个小节的顺序任意；default 小节不一定非要放在 switch 语句最后。甚至可以省略这个小节。
- 和其他语言不同，C#要求每个 switch 小节（包括最后一个小节）以一个跳转语句结尾（参见下一节）。这意味着 switch 小节通常以 break，return 或 goto 结尾。

C# 7.0 为 switch 语句引入了模式匹配，允许 switch 表达式使用任何数据类型，而非只能使用前面描述的有限几个。这样一来，switch 语句的使用就可以基于 switch 表达式的类型（可以在 case 标签中声明变量）。最后，模式匹配的 switch 语句支持条件表达式，所以不仅可以根据类型来标识应执行的 case 标签，还可以在 case 标签末尾使用布尔表达式来标识该标签的执行条件（称为 case guard）。第 7 章会更详细地讨论 switch 语句和表达式的模式匹配。

4.9 跳转语句

循环的执行路径可以改变。事实上，可以使用跳转语句退出循环，或者跳过一次循环迭代的剩余部分并开始下一次迭代——即使循环条件当前仍然为 true。本节介绍了让执行路径从一个位置跳转到另一个位置的几种方式。

4.9.1 break 语句

C#使用 break 语句退出循环或 switch 语句。任何时候遇到 break 语句，控制都会立即离开循环或 switch。代码清单 4.51 和输出 4.24 演示了井字棋程序的 foreach 循环。

代码清单 4.51　发现赢家就用 break 跳出循环

```
int winner = 0;
// 存储每个玩家的落子位置
int[] playerPositions = { 0, 0 };

// 硬编码的棋盘位置
//  X | 2 | O
// ---+---+---
//  O | O | 6
// ---+---+---
//  X | X | X
playerPositions[0] = 449;
playerPositions[1] = 28;

// 判断是否出现了一个赢家
int[] winningMasks = {
            7, 56, 448, 73, 146, 292, 84, 273 };

// 遍历每个致胜掩码(winning mask)，
// 判断是否有一位赢家
foreach (int mask in winningMasks)
{
    if ((mask & playerPositions[0]) == mask)
    {
        winner = 1;
        break;
    }
    else if ((mask & playerPositions[1]) == mask)
    {
        winner = 2;
        break;
    }
}
```

```
    Console.WriteLine($"玩家{winner}是赢家。");
```

输出 4.24

```
玩家 1 是赢家。
```

代码清单 4.51 发现有玩家取胜后就执行 break 语句。break 强迫它所在的循环（或 switch 语句）终止，控制转移到循环（或 switch 语句）后的下一个语句。在本例中，位比较返回 true（在当前硬编码的这个棋盘上，已经有玩家取胜）就执行 break 语句，跳出当前 foreach 循环并显示赢家。

> **初学者主题：用按位操作符处理棋子分布**
>
> 完整井字棋代码清单使用按位操作符判断哪个玩家取胜。首先，代码将每个玩家的落子位置保存到名为 playerPositions 的位映射中（用两个元素的一个数组保存两个玩家的位置）。
>
> 最开始，playerPositions 的两个位置都是 0。玩家每次落子，与落子位置对应的位都被设置。例如，假定玩家选择在单元格 3 落子，那么 shifter 设为 3 - 1。减 1 是因为 C#数组基于 0，所以应将 0 而非 1 视为第一个位置。接着，用移位操作 000000000000001 << shifter 设置 position，即与单元格 3 对应的位。shifter 当前的值是 2。最后，将当前玩家的 playerPositions 设为 0000000000000100（因为基于 0，所以还是要减 1）。代码清单 4.52 使用|=合并之前和当前的落子。
>
> **代码清单 4.52　设置与玩家每次落子对应的二进制位**
>
> ```
> int shifter;   // 要移多少位来设置一个 bit
> int position;  // 要设置的 bit
>
> // int.Parse() 将"input"转换为整数。
> // 之所以要用"int.Parse(input) - 1"，是因为
> // 数组基于零。
> shifter = int.Parse(input) - 1;
>
> // 使掩码 00000000000000000000000000000001
> // 移位单元格的位置。
> position = 1 << shifter;
>
>
> // 取当前玩家的单元格，并对它们进行
> // 按位或（OR）运算，以设置新的位置。
> // 由于 currentPlayer 要么是 1，要么是 2，
> ```

```
    // 因此要减去 1，才能将 currentPlayer
    // 用作零基数组的索引。
    playerPositions[currentPlayer - 1] |= position;
```

之后，就可以遍历与棋盘上所有取胜布局对应的每个掩码，判断当前玩家是否达成了一个取胜布局，就像代码清单 4.51 展示的那样。

4.9.2 continue 语句

循环主体可能有很多语句。如果想在符合特定条件时中断当前迭代，放弃执行剩余语句，那么可以使用 continue 语句跳到当前迭代的末尾，并开始下一次迭代。C# continue 语句允许退出当前迭代（无论剩下多少语句没有执行），并跳到循环条件。如果循环条件仍为 true，那么继续迭代。

代码清单 4.53 使用 continue 语句只显示电子邮件地址的域部分。输出 4.25 展示了结果。

代码清单 4.53　判断电子邮件地址的域

```
string email;
bool insideDomain = false;

Console.Write("输入一个电子邮件地址: ");
email = Console.ReadLine() ?? string.Empty;

Console.Write("该地址的域是: ");

// 遍历 email 地址中的每个字母
foreach(char letter in email)
{
    if(!insideDomain)
    {
        if(letter == '@')
        {
            insideDomain = true;
        }
        continue;
    }
    Console.Write(letter);
}
```

输出 4.25

```
输入一个电子邮件地址: admin@bookzhou.com
该地址的域是: bookzhou.com
```

在代码清单 4.53 中，在遇到电邮地址的域部分之前，需要一直使用 continue 语句来跳至循环末尾，开始处理电子邮件地址的下一个字符。

一般都可以用 if 语句代替 continue 语句，这样还能增强可读性。continue 语句的问题在于，它在一次迭代中提供了多个控制流，从而影响了可读性。代码清单 4.54 重写上面的例子，将 continue 语句替换成 if/else 构造来改善可读性。

代码清单 4.54　将 continue 替换成 if 语句

```
// 遍历 email 地址中的每个字母
foreach (char letter in email)
{
    if (insideDomain)
    {
        Console.Write(letter);
    }
    else
    {
        if (letter == '@')
        {
            insideDomain = true;
        }
    }
}
```

4.9.3 goto 语句

早期编程语言不像 C#这些现代语言那样具备完善的“结构化”控制流，它们要依赖简单的条件分支（if）和无条件分支（goto）语句来满足控制流的需求。这样得到的程序难以理解。许多资深程序员觉得 goto 语句在 C#中继续存在非常反常。但是，C#确实支持 goto，而且只能利用 goto 在 switch 语句中实现贯穿（直通）。在代码清单 4.55 中，如果设置了 /out 选项，就使用 goto 语句跳转到 default，/f 选项的处理与此相似。输出 4.26 展示了结果。

代码清单 4.55　演示带 goto 的 switch 语句

```
public static void Main(string[] args)
{
    bool isOutputSet = false;
    bool isFiltered = false;
    bool isRecursive = false;

    foreach(string option in args)
```

```
    {
        switch(option)
        {
            case "/out":
                isOutputSet = true;
                isFiltered = false;
                // ...
                goto default;
            case "/f":
                isFiltered = true;
                isRecursive = false;
                // ...
                goto default;
            default:
                if(isRecursive)
                {
                    // 向下递归遍历层次结构
                    Console.WriteLine("正在递归遍历...");
                    // ...
                }
                else if(isFiltered)
                {
                    // 为筛选器清单添加选项
                    Console.WriteLine("正在筛选...");
                    // ...
                }
                break;
        }
    }

    // ...
}
```

输出 4.26

```
C:\SAMPLES>Generate /out fizbottle.bin /f "*.xml" "*.wsdl"
```

要跳转到标签不是 default 的其他 switch 小节，可以使用 goto case *constant*;语法；其中，constant 是在目标标签中指定的常量。要跳转到没有和 switch 小节关联的语句，请在目标语句前添加标识符和冒号，并在 goto 语句中使用该标识符。例如，可以写标签语句 myLabel : Console.WriteLine();，然后用 goto myLabel;跳到那里。幸好，C#禁止通过 goto 跳到 goto 语句作用域外部的一个分支。只能用 goto 在当前代码块内部跳转，或者跳转到包围它的外层代码块（例如，跳出嵌套循环）。通过这个限制，C#避免了在其他语言中可能遇到的大多数滥用 goto 的情况。

一般认为使用 goto 是不“优雅”的，难以理解，而且会造成结构很差的代码。要多次或者在不同情况下执行某个代码小节，要么使用循环，要么将代码重构为方法。

设计规范

AVOID using goto.

避免使用 goto。

4.10 C#预处理器指令

控制流语句在运行时求值条件表达式。与之相反，C#预处理器在编译时调用。预处理器指令告诉 C#编译器要编译哪些代码，并指出如何处理代码中的特定错误和警告。C#预处理器指令还可以告诉 C#编译器有关代码组织的信息。

> 语言对比：C++——预处理
>
> C 和 C++等语言用一个**预处理器**（preprocessor）对代码进行整理，根据特殊的记号来执行特殊的操作。预处理器指令通常告诉编译器如何编译文件中的代码，而并不参与实际的编译过程。相反，C#编译器将预处理器指令作为对源代码执行的常规词法分析的一部分。其结果就是，C#不支持更高级的预处理器宏，它最多只允许定义常量。事实上，“预处理器”在 C++中显得很贴切，但在 C#中就属于用词不当。

每个预处理器指令都以#开头，而且必须一行写完。换行符（而不是分号）标志着预处理器指令的结束。

表 4.5 总结了所有预处理器指令。

表 4.5 预处理器指令

语句或表达式	常规语法结构	示例
#if 指令	**#if** preprocessor-expression *代码* **#endif**	**#if** CSHARP2PLUS Console.Clear(); **#endif**
#elif 指令	**#if** preprocessor-expression1 代码 **#elif** preprocessor-expression2 *代码* **#endif**	**#if LINUX** ... **#elif WINDOWS** ... **#endif**

#else 指令	**#if** *代码* **#else** *代码* **#endif**	**#if** CSHARP1 ... **#else** ... **#endif**
#define 指令	**#define** conditional-symbol	**#define** CSHARP2PLUS
#undef 指令	**#undef** conditional-symbol	**#undef** CSHARP2PLUS
#error 指令	**#error** preproc-message	**#error** Buggy implementation
#warning 指令	**#warning** preproc-message	**#warning** Needs code review
#pragma 指令	**#pragma** warning	**#pragma** warning disable CS1030
#line 指令	**#line** org-line new-line **#line default**	**#line** 467 "TicTacToe.cs" ... **#line default**
#region 指令	**#region** pre-proc-message *代码* **#endregion**	**#region** Methods ... **#endregion**
#nullable 指令	**#nullable** enable | disable | restore	**#nullable** enable ..string? text = null; **#nullable** restore

4.10.1 排除和包含代码

经常用预处理器指令控制何时以及如何包含代码。例如，要使代码兼容 C# 2.0（及以后版本）和 1.0 的编译器，可指示在遇到 1.0 编译器时排除 C# 2.0 特有的代码。井字棋程序和代码清单 4.56 对此进行了演示。

代码清单 4.56　遇到 C# 1.x 编译器就排除 C# 2.0 代码

```
#if CSHARP2PLUS
System.Console.Clear();
#endif
```

本例调用了仅 2.0 或更高版本才支持的 System.Console.Clear()方法。使用#if 和#endif 预处理器指令，这行代码只有在定义了预处理器符号 CSHARP2PLUS 的前提下才会编译。

预处理器指令的另一个应用是处理不同平台之间的差异，比如用 WINDOWS 和 LINUX #if

指令将 Windows 和 Linux 特有的 API 包围起来。开发人员经常用这些指令取代多行注释（/*...*/），因为它们更容易通过定义恰当的符号或通过搜索/替换来移除。

预处理器指令最后一个常见的用途是调试。用#if DEBUG 指令将调试代码包围起来之后，大多数 IDE 都支持在发布版中移除这些代码。IDE 默认将 DEBUG 符号用于调试编译，将 RELEASE 符号用于发布生成。

为了处理 else-if 条件，可以在#if 指令中使用#elif 指令，而不是创建两个完全独立的#if 块，如代码清单 4.57 所示。

代码清单 4.57 使用#if，#elif 和#endif 指令

```
#if LINUX
// ...
#elif WINDOWS
// ...
#endif
```

4.10.2 定义预处理器符号

可以通过两种方式定义预处理器符号。第一种是使用#define 指令，如代码清单 4.58 所示。

代码清单 4.58 #define 示例

```
#define CSHARP2PLUS
```

第二种方式是在编译时使用 define 选项，输出 4.27 演示了在命令行上的用法。

输出 4.27

```
>csc.exe -define:CSHARP2PLUS TicTacToe.cs
```

多个定义以分号分隔。使用 define 编译器选项的优点是不需要更改源代码，所以可以使用相同的源代码文件生成两套不同的二进制程序。

要取消符号定义，可以采取和使用#define 相同的方式来使用#undef 指令。

4.10.3 生成错误和警告(#error，#warning)

有时要标记代码中潜在的问题。为此，可以插入#error 和#warning 指令来分别生成错误和警告消息。代码清单 4.59 使用井字棋的例子来警告目前的代码无法防止玩家多次落子于同

一个位置。输出 4.28 展示了结果。

代码清单 4.59　用#warning 定义警告

```
#warning "允许在同一个位置多次落子。"
```

输出 4.28

```
Performing main compilation...
...\tictactoe.cs(471,16): warning CS1030: #warning: '"允许在同一个位置多次落子。"'

Build complete -- 0 errors, 1 warnings
```

包含#warning 指令后，编译器会主动发出警告，如输出 4.28 所示。可以利用这种警告来标记代码中潜在的 bug 和也许能改善的地方。它是提醒开发者任务尚未完结的好帮手。

4.10.4 关闭警告消息(#pragma)

警告指出代码中可能存在的问题，所以很有用。但有的警告可安全地忽略。C# 2.0 和之后的编译器提供了预处理器指令#pragma 来关闭或还原警告，如代码清单 4.60 所示。

代码清单 4.60　使用预处理器指令#pragma 禁用#warning 指令

```
#pragma warning disable CS1030
```

重新启用警告仍是使用#pragma 指令，只是在 warning 后添加 restore 选项，如代码清单 4.61 所示。

代码清单 4.61　使用预处理器指令#pragma 还原警告

```
#pragma warning restore CS1030
```

上述两条指令正好可以将一个特定的代码块包围起来——前提是已知该警告不适用于该代码块。

经常被禁用的警告是 CS1591。使用/doc 编译器选项生成 XML 文档，但并未注释程序中的所有公共项将显示该警告。

代码警告在本书经常出现，因为展示的代码清单通常是不完整的——也就是说，展示的是

未完全开发的初始代码片段。为了抑制这些警告（它们在示例代码场景中不相关），我们在文件中添加了#pragma 指令。表 4.6 展示了在第 4 章各个代码片段中禁用的一些警告。

表 4.6 示例警告

类别	警告
CS0168	声明了声明，但从未使用过
CS0219	变量已被赋值，但从未使用过它的值
IDE0059	不需要对变量赋值

本书源代码经常会嵌入这样的#pragma warning disable 指令，以避免因示例代码未完全开发而引起的警告。这些代码旨在阐明当前主题，并不需要完整。

注意，编译器输出 C#警告编号时，会附带一个 CS 字母前缀。在#pragma 指令中，可以不指定该前缀。要查看一个警告的编号，可以暂时屏蔽预处理器命令，让编译器显示警告即可看到。但是，有的前缀不能在#pragma 指令中省略，例如 IDE（代表 identity）。IDE0059 消息就是一个例子。代码分析器生成带有 IDE 前缀的警告以建议代码改进。如果在#pragma 指令中删除该前缀，那么所引用的就不再是同一个警告了。

4.10.5 nowarn:<warn list>选项

除了#pragma 指令，C#编译器通常还支持 nowarn:<warn list>选项。它可以获得与#pragma 相同的结果，只是它不放到源代码中，而作为编译器选项使用。除此之外，nowarn 选项会影响整个编译过程，而#pragma 指令只影响该指令所在的那个文件。如输出 4.29 所示，我们在命令行上关闭了 CS1591 警告。

输出 4.29

```
> dotnet build /p:NoWarn="0219"
```

4.10.6 指定行号(#line)

用#line 指令改变 C#编译器在报告错误或警告时显示的行号。该指令主要由自动生成 C#代码的实用程序和设计器使用。在代码清单 4.62 中，真实行号显示在最左侧。

代码清单 4.62 #line 预处理器指令

```
124 #line 113 "TicTacToe.cs"
125 #warning "允许在同一个位置多次落子。"
```

```
126 #line default
```

在上例中，使用#line 指令后，编译器会将实际发生在第 125 行的警告报告在第 113 行上发生，如输出 4.30 所示。

输出 4. 30

```
Performing main compilation...
...\tictactoe.cs(113,18): warning CS1030: #warning: '"允许在同一个位置多次落子。"'

Build complete -- 0 errors, 1 warnings
```

在#line 指令后添加 default，会反转之前的所有#line 的效果，并指示编译器报告真实的行号，而不是之前使用#line 指定的行号。

4.10.7 可视编辑器提示(#region，#endregion)

C#提供了只有在可视代码编辑器中才有用的两个预处理器指令：#region 和#endregion。像 Microsoft Visual Studio 这样的代码编辑器能搜索源代码，找到这些指令，并在写代码时提供相应的编辑器功能。C#允许用#region 指令声明代码区域。#region 和#endregion 必须成对使用，两个指令都可以选择在指令后跟随一个描述性字符串。此外，一个区域可以嵌套到另一个区域中。

代码清单 4.63 是井字棋程序的例子。

代码清单 4. 63　#region 和#endregion 预处理器指令

```
// ...
#region 显示井字棋棋盘

#if CSHARP2PLUS
System.Console.Clear();
#endif

// 显示当前棋盘
border = 0;   //  设置第一个界线，(border[0] = "|")

// 显示顶行连线
// ("\n---+---+---\n")
Console.Write(borders[2]);
foreach(char cell in cells)
{
    // 输出一个单元格值以及紧接在它后面的边框
    Console.Write($" { cell } { borders[border] }");
```

```
        // 递增到下一个边框
        border++;

        // 如果边框到 3 了，就重置为 0
        if(border == 3)
        {
            border = 0;
        }
    }

    #endregion 显示井字棋棋盘

    // ...
```

Visual Studio 检查上述代码，在编辑器左侧提供树形控件来展开和折叠由#region 和#endregion 指令界定的代码区域，如图 4.5 所示。

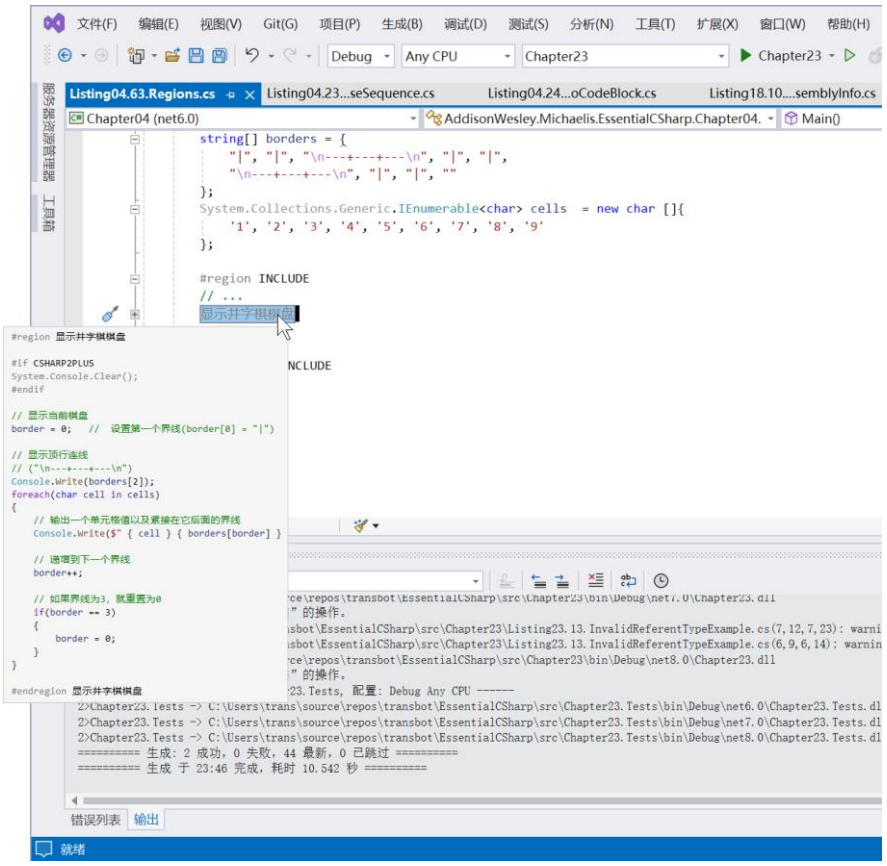


图 4.5 Microsoft Visual Studio 的折叠区域

4.10 小结

本章首先介绍了 C#赋值和算术操作符。接着讲解了如何使用操作符和 const 关键字声明常量。但是，我们并没有按顺序讲解所有 C#操作符。相反，在讨论关系和逻辑比较操作符之前，我们先讨论了 if 语句，并强调了代码块和作用域等重要概念。最后讨论的操作符是按位操作符，强调了掩码的用法。然后讨论了其他控制流语句，比如循环、switch 和 goto。本章最后讨论了 C#预处理器指令。

本章早些时候已讨论了操作符优先级，但表 4.5 的总结最全面，其中包括几个尚未讲到的。

表 4.5 操作符优先级*

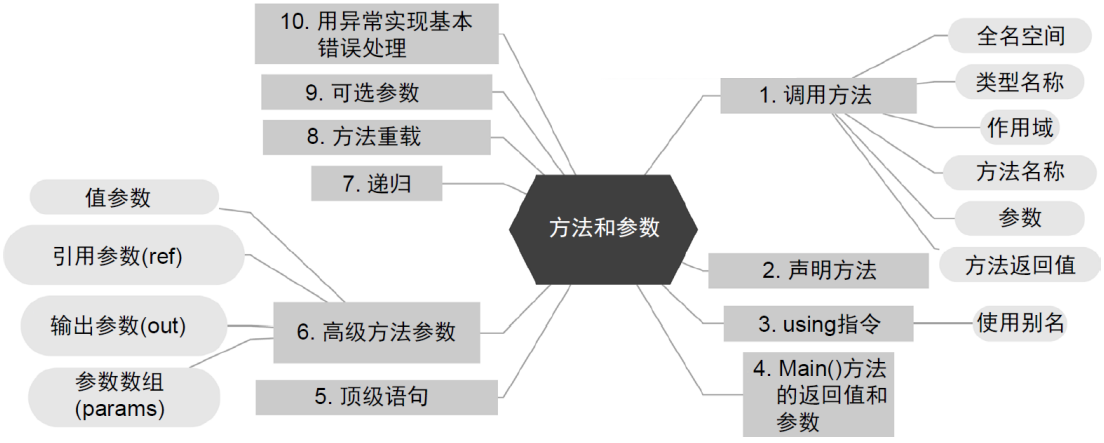
类别	操作符
主要	x.y f(x) a[x] x++ x-- new typeof(T) checked(X) unchecked(X) default(T) nameof(x) delegate{} ()
一元	+ - ! ~ ++x --x (T)x await x
乘	* / %
加	+ -
移位	<< >>
关系和类型测试	< > <= >= is as
相等性	== !=
逻辑 AND	&
逻辑 XOR	^
逻辑 OR	|
条件 AND	&&
条件 OR	||
空合并	??
条件	?:
赋值和 Lambda	= *= /= %= += -= <<= >>= &= ^= |= =>

* 各行优先级从高到低排列。

要复习第 1 章～第 4 章的内容，或许最好的办法是将井字棋程序（Chapter04\TicTacToe.cs）彻底搞清楚。通过研究该程序，可慢慢领悟如何将自己学到的东西合并成完整程序。

第 5 章 参数和方法

基于目前学到的 C#编程知识，大家应该能写一些简单、直观的程序，它们由一组语句构成，和上个世纪 70 年代的那些程序差不多。但编程技术自 70 年代以来有了长足进步，随着程序变得越来越复杂，需要新的思维模式来管理这种复杂性。“过程式”或“结构化”编程的基本思路就是提供一些语言构造对语句分组来构成单元。此外，可以通过结构化编程将数据传给一个语句分组，在这些语句执行完毕后返回结果。



除了方法定义和调用的基础知识，本章还讨论了一些更高级的概念，包括递归、方法重载、可选参数和具名参数。注意，目前和直至本章末尾讨论的都是静态方法(第 6 章详述)。

其实从第 1 章的 HelloWorld 程序起就已学习了如何定义方法。那个例子定义的是 `Main()`方法。本章将更详细地学习方法的创建，包括如何用特殊的 C#语法（`ref` 和 `out`）让参数向方法传递变量而不是值。最后会介绍一些基本的错误处理技术。

5.1 调用方法

> **初学者主题：什么是方法**
>
> 到目前为止，我们看到的几乎所有语句都是“平铺直叙”的，除了语句或代码块本身提供的分组，没有其他分组。随着程序越来复杂，这种代码很快就会变得难以维护，可读性也越来越差。
>
> **方法**（method）组合一系列语句，以执行特定操作或计算特定结果。它能为构成程序的语句提供更好的结构和组织。例如，假定要用 `Main()`方法统计某个目录下源代码的行数。我们不是在一个巨大的 `Main()`方法中写所有代码，而是提供更简短的版本，以便在必要时能专注每个方法的实现细节，如代码清单 5.1 所示。

代码清单 5.1　将语句组合成方法

```
public class Program
{
    public static void Main()
    {
        int lineCount;
        string files;

        DisplayHelpText();
        files = GetFiles();
        lineCount = CountLines(files);
        DisplayLineCount(lineCount);
    }

    // ...
}
```

在这个例子中，不是将所有语句都放到 Main()中，而是把它们划分到多个方法中。例如，程序先用一系列 Console.WriteLine()语句显示帮助文本，这些语句全部放到 DisplayHelpText()方法中。类似地，用 GetFiles()方法获取要统计行数的文件。最后调用 CountLines()方法实际统计行数，并调用 DisplayLineCount()方法显示结果。一眼就能看清楚整个程序的结构，因为方法名清楚描述了方法的作用。

初学者主题：什么是函数

函数在本质上与方法几乎相同，都用于将一系列语句组合在一起，以执行特定操作或计算特定结果。实际上，“方法”和“函数”经常被互换着使用。然而，从技术上讲，方法总是与一种类型关联，比如代码清单 5.1 中的 Program 类。相反，函数没有这样的关联。函数不一定有所属的类型。

设计规范

DO give methods names that are verbs or verb phrases.

要为方法名使用动词或动词短语。

方法提供了对相关代码进行组合的一种方式。方法通过**实参**（arguments）接收数据，实参由方法的**参数**或**形参**（parameters）定义[①]。参数是**调用者**（发出方法调用的代码）用于向被调用的方法（例如 Write()，WriteLine()，GetFiles()，CountLines()等）传递数据的变量。在代码清单 5.1 中，files 和 lineCount 分别是传给 CountLines()和 DisplayLineCount()方法的实参。方法通过**返回值**将数据返回调用者。在代码清单 5.1 中，GetFiles()方法调用的返回值被赋给 files。

首先，让我们重新讨论一下第 1 章讲过的 Console.Write()、Console.WriteLine()和 Console.ReadLine()方法。这一次要从方法调用的角度讨论，而不是强调控制台的输入和输出细节。代码清单 5.2 展示了这三个方法的应用。

代码清单 5.2　简单方法调用

```
    public static void Main()
        {
            string? firstName;  // 用于存储名字的变量
            string? lastName;   // 用于存储姓氏的变量

            Console.WriteLine("嘿，你！");

            Console.Write("请输入你的名字：");
            firstName = Console.ReadLine();

            Console.Write("请输入你的姓氏：");
            lastName = Console.ReadLine();

            /* 使用字符串插值在控制台上显示问候语*/
            Console.WriteLine($"你的全名是{ firstName } { lastName }。");
    }
```

方法调用由方法名称和实参列表构成。完全限定的方法名称包括命名空间、类型名和方法名；每部分以句点分隔。第 2 章在讲解类型名称的时候，我们提到方法也可以只通过它的完全限定名称的最后一部分来调用。

5.1.1 命名空间

命名空间是一种分类机制，用于分组功能相关的所有类型。命名空间以层次化的方式组织，

[①] 译注：以后不需要区分形参和实参时一般以“参数”代之。

级数任意，但超过 6 级就很罕见了。一般从公司名开始，然后是产品名，最后是功能领域。例如在 Microsoft.Win32.Networking 中，最外层的命名空间是 Microsoft，它包含内层命名空间 Win32，后者又包含嵌套更深的 Networking 命名空间。

主要用命名空间按功能领域组织类型，以便查找和理解这些类型。此外，命名空间还有助于防范类型名称冲突。两个都叫 Button 的类型只要在不同命名空间，比如 System.Web.UI.WebControls.Button 和 System.Windows.Controls.Button，编译器就能区分。

在代码清单 5.2 中，Console 类型位于 System 命名空间。System 命名空间包含用于执行大量基本编程活动的类型。几乎所有 C#程序都要使用 System 命名空间中的类型。表 5.1 总结了其他常用命名空间。

表 5.1 常用命名空间

命名空间	**描述**
System	包含基元类型，以及用于类型转换、数学计算、程序调用以及环境管理的类型
System.Collections.Generics	包含使用泛型的强类型集合
System.Data.Entity	设计用于从关系数据库存储和检索数据的框架，它生成代码将实体映射到表格
System.Drawing	包含用于在显示设备上绘图和进行图像处理的类型。但要注意的是，它主要用于 Windows 平台
System.IO	包含用于文件和目录处理的类型
System.Linq	包含使用“语言集成查询”（LINQ）对集合数据进行查询的类和接口
System.Text	包含用于处理字符串和各种文本编码的类型，以及在不同编码方式之间转换的类型
System.Text.RegularExpressions	包含用于处理正则表达式的类型
System.Threading	包含用于多线程编程的类型
System.Threading.Tasks	包含以任务（Task）为基础进行异步操作

	的类型
System.Windows	包含用 WPF 或 Windows UI 库（WinUI）创建富用户界面的类型，使用用 XAML 进行声明性（宣告式）UI 设计
System.Xml.Linq	使用名为 LINQ 的一种对象查询语言为 XML 处理提供基于标准的支持（参见第 15 章）

调用方法并非一定要提供命名空间。例如，如果要调用的方法与发出调用的方法在同一个命名空间，就没必要指定命名空间。本章稍后会讲解如何利用 using 指令避免每次调用方法都指定命名空间限定符。

设计规范

DO use PascalCasing for namespace names.

要为命名空间使用 PascalCase 大小写。

CONSIDER organizing the directory hierarchy for source code files to match the namespace hierarchy.

考虑组织源代码文件目录结构以匹配命名空间层次结构。

5.1.2 类型名称

调用静态方法时，只要目标方法和调用者不在同一个类型（或基类）中，就需要添加类型名称限定符（本章稍后会介绍如何用 using static 指令省略类型名称）。例如，从 HelloWorld.Main()中调用静态方法 Console.WriteLine()时，就需要添加类型名称 Console（控制台）。但和命名空间一样，如果要调用的方法是调用表达式所在类型的成员，C#就允许在调用时省略类型名称（代码清单 5.4 展示了这种方法调用的例子）。之所以不需要类型名称，是因为编译器能够根据调用位置推断类型。显然，如果编译器无法进行这样的推断，就必须将类型名称作为方法调用的一部分提供。

类型本质是对方法及其相关数据进行分组的一种方式。例如，Console 类型包含控制台常用的 Write()，WriteLine()和 ReadLine()等方法。所有这些方法都在同一个“组”中，都从属于 Console 类型。

5.1.3 作用域

上一章讲过，一个事物的“作用域”是可用非限定名称来引用它的那个区域。两个方法在同一个类型中声明，一个方法调用另一个就不需要类型限定符；因为这两个方法具有整个包容类型的作用域。类似地，类型的作用域是声明它的那个命名空间。所以，特定命名空间中的一个类型中的方法调用不需要指定该命名空间。

5.1.4 方法名称

每个方法调用都要指定一个方法名称。如前所述，它可能用、也可能不用命名空间和类型名称加以限定。方法名称之后是圆括号中的实参列表，每个实参以逗号分隔，对应于声明方法时指定的形参。

5.1.5 形参和实参

方法可以接收任意数量的形参，每个形参都具有特定数据类型。调用者为形参提供的值称为**实参**；每个实参都要和一个形参对应。例如，以下方法调用传递了三个实参。

```
System.IO.File.Copy( oldFileName, newFileName, false);
```

该方法位于 File 类，后者位于 System.IO 命名空间。方法声明为获取三个参数，第一个和第二个是 string 类型，第三个是 bool 类型。本例传递 string 变量 oldFileName 和 newFileName 来分别代表旧的和新的文件名，第三个实参是 false，指定在新文件名已经存在的前提下文件拷贝失败。

5.1.6 方法返回值

和 Console.WriteLine()相反，代码清单 5.2 中的 Console.ReadLine()没有任何参数，因为该方法声明为不获取任何参数。但这个方法有返回值。可以利用返回值将调用方法所生成的结果返回给调用者。由于 Console.ReadLine()有返回值，所以可以将返回值赋给变量 firstName。除此之外，还可以将方法的返回值作为另一个方法的实参使用，如代码清单 5.3 所示。

代码清单 5.3　将方法返回值作为实参传给另一个方法调用

```
 public static void Main()
{
    Console.Write("请输入你的名字： ");
    Console.WriteLine($"你好，{ Console.ReadLine() }! ");
}
```

代码清单 5.3 不是先为变量赋值，再在 Console.WriteLine()调用中使用这个变量。相反，是在调用 Console.WriteLine()时直接调用 Console.ReadLine()方法。运行时会先执行 Console.ReadLine()方法，返回值直接传给 Console.WriteLine()方法，而不是传给一个变量。

并非所有方法都返回数据，Console.Write()和 Console.WriteLine()就是如此。稍后会讲到，这种方法指定了 void 返回类型，好比在 HelloWorld 的例子中，Main()方法的返回类型就是 void。

5.1.7 对比语句和方法调用

代码清单 5.3 演示了语句和方法调用的差异。Console.WriteLine($"你好，{ Console.ReadLine() }! ");在一个语句中包含两了个方法调用。语句通常包含一个或多个表达式，本例的两个表达式都是方法调用。所以，我们说方法调用构成了语句的不同部分。

虽然在一个语句中包含多个方法调用能减少编码量，但不一定能增强可读性，而且很少能带来性能上的优势。开发者应该更注重代码的可读性，而不要将过多精力放在写简短的代码上。

> **注意：**通常，开发者应侧重于可读性，而不是在写更短的代码方面耗费心机。为了使代码一目了然，进而在长时间里更容易维护，可读性是关键。

5.2 声明方法

本节描述如何声明方法来包含参数或返回类型。代码清单 5.4 演示了这些概念，输出 5.1 展示了结果。

代码清单 5. 4　声明方法

```
public class IntroducingMethods
{
    public static void Main()
    {
        string firstName;
        string lastName;
        string fullName;
        string initials;

        Console.WriteLine("嘿，你！");
```

```
            firstName = GetUserInput("请输入你的名字: ");
            lastName = GetUserInput("请输入你的姓氏: ");

            fullName = GetFullName(firstName, lastName);
            initials = GetInitials(firstName, lastName);
            DisplayGreeting(fullName, initials);
        }

        static string GetUserInput(string prompt)
        {
            Console.Write(prompt);
            return Console.ReadLine() ?? string.Empty;
        }

        static string GetFullName(
              string firstName, string lastName) =>
                  $"{ firstName } { lastName }";


        static void DisplayGreeting(string fullName, string initials)
        {
            Console.WriteLine(
                $"你好，{ fullName }! 你的姓名缩写是{ initials }");
            return;
        }

        static string GetInitials(string firstName, string lastName)
        {
            return $"{ firstName[0] }. { lastName[0] }。";
        }
    }
```

输出 5.1

```
嘿，你!
请输入你的名字: Inigo
请输入你的姓氏: Montoya
你好，Inigo Montoya! 你的姓名缩写是 I. M。
```

代码清单 5.4 总共声明了 5 个方法。在 Main()方法中，先调用了 GetUserInput()，然后调用了 GetFullName()和 GetInitials()。后三个方法都返回一个值，而且都要获取实参。最后调用 DisplayGreeting()，它不返回任何数据。

> 语言对比：C++/Visual Basic——全局方法

> C#不支持全局方法；一切都必须放到一个类型声明中。这正是 `Main()`方法标记为 `static` 的原因——它等价于 C++的全局方法和 Visual Basic 的“共享”方法。出现在方法之外的语句以及出现在类之外的方法都符合这一规则，因为C#编译器为这些看似独立的语言构造生成了包围它们的方法和类。更多信息请参见本章后面的“顶级语句”一节。

> **初学者主题：用方法进行重构**
>
> 将一组语句转移到一个方法中，而不是把它们留在一个较大的方法中，这是**重构**的一种形式。重构有助于减少重复代码，因为可以从多个位置调用方法，而不必在每个位置都重复这些代码。重构还有助于增强代码的可读性。我们平时编码的时候，一个最佳实践就是经常主动检查代码，找出可以重构的地方。尤其是那些不好理解的代码块，最好把它们转移到方法中，用有意义的方法名清晰定义代码的行为。与简单地为代码块加上注释相比，重构效果更佳，因为看方法名就知道方法要做什么。
>
> 例如，代码清单 5.4 的 `Main()`方法只需扫一眼，就可以理解整个程序（暂时不用操心被调用的每个方法的实现细节）。
>
> Visual Studio 允许右击选定的一组语句，选择“快速操作和重构”（或者按 Ctrl+.）将这组语句提取到一个方法中，并在原始位置插入对新方法的调用。

5.2.1 参数声明

注意 `DisplayGreeting()`，`GetFullName()`和 `GetInitials()`方法的声明。可以在方法声明的圆括号中添加参数（形参）列表（以后在讨论泛型时会讲到，方法还可以有一个**类型参数列表**。以后，如果能根据上下文分清当前讲的是哪种参数，就直接把它们称为“参数列表”中的“参数”，也不刻意区分形参和实参）。列表中的每个参数都包含参数类型和参数名称，每个参数以逗号分隔。

大多数参数的行为和命名规范与局部变量一致。所以参数名采用 camelCase 大小写风格。另外，不能在方法中声明与参数同名的局部变量，因为这会造成同名的两个“局部变量”。

设计规范

DO use “camelCasing” for parameter names.

要为参数名使用 camelCase 大小写风格。

5.2.2 方法返回类型声明

GetUserInput()，GetFullName()和 GetInitials()方法除了定义参数，还定义了**方法返回类型**。很容易就可分辨一个方法是否有返回值，因为在声明这种方法时，会在方法名之前添加一个数据类型。上述所有方法的返回数据类型都是 string。虽然方法可指定多个参数，但返回类型只能有一个。

如 GetUserInput()和 GetInitials()方法所示，具有返回类型的方法几乎总是包含一个或多个 return 语句，以便将控制返回给调用者。return 语句以 return 关键字开头，后跟计算返回值的表达式。例如，GetInitials()方法的 return 语句是：

```
return $"{ firstName[0] }. { lastName[0] }。";
```

return 关键字后面的表达式（本例是一个插值字符串）必须兼容方法的返回类型。

如果方法有返回类型，那么它的主体不能存在“不可到达”的语句。换言之，所有执行路径都应该以一个 return 语句结尾，在此之后的语句都“不可到达”。为了保证这一点，最简单的办法就是将 return 语句作为方法的最后一个语句。但这并非绝对，return 语句并非只能在方法末尾出现。例如，方法中的 if 或 switch 语句可以包含 return 语句，如代码清单 5.5 所示。

代码清单 5.5　方法中间的 return 语句

```
public class Program
{
    // ...

    public static bool MyMethod()
    {
        string command = ObtainCommand();
        switch(command)
        {
            case "quit":
                return false;
            // ... 省略了其他 case
            default:
                return true;
        }
    }

    // ...
}
```

注意，return 语句将控制转移出 switch，所以这里不需要用 break 语句来防止非法“贯

穿”switch 小节。

在代码清单 5.5 中，方法最后一个语句不是 return 语句，而是 switch 语句。但是，编译器判断方法的每条执行路径最终都是 return 语句，所有代码都会在 return 之前执行到。所以，这样的方法是合法的，即使它不以 return 语句结尾。

如果 return 之后有“不可到达”的语句，编译器会发出警告，指出存在永远执行不到的语句。

虽然 C#允许提前返回，但为了增强代码的可读性，并使代码更易维护，应尽量确定单一的退出位置，而不是在方法的多个代码路径中散布多个 return 语句。

指定 void 作为返回类型，表示方法没有返回值。所以，无法用这种方法的返回值向变量赋值，也无法在调用位置[①]作为参数传递。void 调用只能作为语句使用。此外，return 在这种方法内部可选。如果指定 return，那么它之后不能有任何值。例如代码清单 5.4 的 Main()方法的返回值是 void，方法中没有使用 return 语句。而 DisplayGreeting()有 return 语句，但 return 之后没有添加任何值。

虽然从技术上说方法只能有一个返回类型，但返回类型可以是一个元组。因此，从 C# 7.0 起，多个值可以通过 C#元组语法打包成元组返回，如代码清单 5.6 的 GetName()方法所示

代码清单 5.6　用元组返回多个值

```
public class Program
{
    static string GetUserInput(string prompt)
    {
        Console.Write(prompt);
        return Console.ReadLine() ?? string.Empty;
    }
    static (string First, string Last) GetName()
    {
        string firstName, lastName;
        firstName = GetUserInput("请输入你的名字: ");
        lastName = GetUserInput("请输入你的姓氏: ");
        return (firstName, lastName);
    }
    public static void Main()
    {
        (string First, string Last) name = GetName();
        Console.WriteLine($"你好, { name.First } { name.Last }! ");
    }
```

[①] 译注：call site，就是发出调用的地方，可以理解成调用了一个目标方法的表达式或代码行。

```
}
```

从技术上说，它仍然只返回一个数据类型，即一个ValueTuple<string, string>。但利用元组中的每一项，实际可以返回任意多个数据类型（当然要合理）。

5.2.3 表达式主体方法

有些方法过于简单。为了简化这些方法的定义，C# 6.0 引入了**表达式主体方法**，允许用表达式代替完整方法主体。代码清单 5.4 的 GetFullName()方法就是一例：

```
static string GetFullName(
      string firstName, string lastName) =>
          $"{ firstName } { lastName }";
```

表达式主体方法不是用大括号定义方法主体，而是用=>操作符（第 13 章详述）。该操作符的结果数据类型必须与方法返回类型匹配。换言之，虽然没有显式的 return 语句，但表达式本身的返回类型必须与方法声明的返回类型匹配。

表达式体方法是完整方法主体语法的一种“语法糖”。因此，其应用应限于最简单的方法实现，例如单行表达式。

> **语言对比：C++——头文件**
>
> 和 C++不同，C#类从来不将实现与声明分开。C#不区分头文件（.h）和实现文件（.cpp）。相反，声明和实现在同一个文件中。（C#确实支持名为“分部方法”的高级功能，允许将方法的声明和实现分开。但考虑到本章的目的，我们只讨论非分部方法。）这样就不需要在两个位置维护冗余的声明信息。

5.2.4 本地函数

C# 7.0 引入了在方法内声明函数的能力。换言之，现在不是非要直接在类内部声明方法，而是可以在另一个方法内声明方法（从技术上说是函数），如代码清单 5.7 所示。

代码清单 5.7　声明本地函数

```
public static void Main()
{
    string GetUserInput(string prompt)
    {
        string? input;
        do
        {
```

```
            Console.Write(prompt + ": ");
            input = Console.ReadLine();
        }
        while(string.IsNullOrWhiteSpace(input));
        return input!;
    };

    string firstName = GetUserInput("名字");
    string lastName = GetUserInput("姓氏");
    string email = GetUserInput("电子邮件地址");

    Console.WriteLine($"{firstName} {lastName} <{email}>");
    //...
}
```

这种语言构造称为**本地函数**（local function）。之所以要以这种方式声明，是为了将该函数的作用域限制在声明它的那个方法内部。在方法的作用域之外，无法调用本地函数。此外，本地函数可以访问本地函数之前在方法中声明的局部变量。如果不想这样，还可以通过在本地函数声明的开头添加 `static` 来明确阻止这一点（参见第 13 章的“静态匿名函数”一节）。

5.3 using 指令

完全限定的名称可能很长、很笨拙。可以将一个或多个命名空间的所有类型“导入”文件，这样在使用时就不需要完全限定。

5.3.1 using 指令概述

C#支持一个只应用于当前文件（而不是整个项目）的 `using` 指令。例如，在代码清单 5.8 中，不需要为 `RegEx` 附加 System.Text.RegularExpressions 前缀。这是因为代码文件顶部有一个 `using System.Text.RegularExpressions` 指令。这个程序的结果如输出 5.2 所示。

代码清单 5.8　using 指令示例

```
// using 指令将所有类型从指定命名空间
// 导入当前代码文件。

using System.Text.RegularExpressions;

public class Program
{
    public static void Main()
    {
        const string firstName = "FirstName";
```

```
        const string initial = "Initial";
        const string lastName = "LastName";

        // 对正则表达式的详细解释超出了本书范围，
        // 访问 https://www.regular-expressions.info 了解详情。
        const string pattern = $"""
            (?<{firstName}>\w+)\s+((?<{
            initial}>\w)\.\s+)?(?<{
            lastName}>\w+)\s*
            """;

        Console.WriteLine(
            "输入你的全名 (例：Inigo T. Montoya): ");
        string name = Console.ReadLine()!;

        // 因为之前的 using 指令，所以
        // 不需要用 System.Text.RegularExpressions
        // 前缀来限定 RegEx 类型。
        Match match = Regex.Match(name, pattern);

        if (match.Success)
        {
            Console.WriteLine(
                $"{firstName}: {match.Groups[firstName]}");
            Console.WriteLine(
                $"{initial}: {match.Groups[initial]}");
            Console.WriteLine(
                $"{lastName}: {match.Groups[lastName]}");
        }
    }
}
```

输出 5.2

```
输入你的全名 (例：Inigo T. Montoya):
Inigo T. Montoya
FirstName: Inigo
Initial: T
LastName: Montoya
```

像 using System.Text 这样的 using 指令并不能避免为**嵌套命名空间**（例如 System.Text.RegularExpressions）中声明的类型省略 System.Text。例如，如果代码要访问 System.Text.RegularExpressions 命名空间中的 RegEx 类型，那么要么添加另一个 using System.Text.RegularExpressions 指令，要么将类型完全限定为 System.Text.RegularExpressions.RegEx，而不能只是写 RegularExpressions.RegEx。简单地说，using 指令不会“导入”任何嵌套命名空间中的类型。嵌套命名空间（由命名空间中的句点符号来标识）必须显式导入。

> **语言对比：Java——import 指令中的通配符**
>
> Java 允许使用通配符导入命名空间，例如：
>
> ```
> import javax.swing.*;
> ```
>
> 相反，C#不允许在 using 指令中使用通配符，每个命名空间都必须显式导入。

通常，程序如果要用到一个命名空间中的大量类型，那么就应该考虑为该命名空间使用 using 指令，避免对该命名空间中的所有类型都进行完全限定。正是因为这个原因，几乎所有文件都在顶部添加了 using System 指令。在本书剩余的部分，代码清单会经常省略 using System 指令。但其他命名空间指令都会显式地包含（已设为全局 using 的除外，稍后会详述）。

使用 using System 指令的一个有趣结果是，可以使用不同的大小写形式来表示字符串数据类型：String 或者 string。前者的基础是 using System 指令，后者使用的是 string 关键字。两者在 C#中都引用 String 数据类型，最终生成的 CIL 代码毫无区别[①]。

> **初学者主题：命名空间**
>
> 如前所述，**命名空间**是分类和分组相关类型的一种机制。在一个类型所在的命名空间中，能找到和它相关的其他类型。此外，不同命名空间中重名的两个或更多类型没有歧义。

5.3.2 隐式 using 指令

本书第 1 章讲到，C# 10.0 及更高版本的.csproj 文件支持一个 ImplicitUsings 元素。

```
<ImplicitUsings>enable</ImplicitUsings>
```

此外，在我们的源代码中，会经常引用诸如 Console 的东西，即使它的完全限定名称是 System.Console。之所以能如此简化，是因为我们已将 ImplicitUsings 元素设为 enable，从而告诉编译器自动推断命名空间，而不需要程序员提供。因此，我们可以在不指定完全限定名称的情况下使用类型标识符。将 ImplicitUsings 元素设为 enable 后，编译器会自动生成全局 using 指令，如下一节所述。

---

① 我（作者）更喜欢使用 string 关键字，但无论选择哪一种表示方法，都应在项目中保持一致。

5.3.3 全局 using 指令

搜索一个 C# 10.0（或更高版本）项目的子目录，会注意到有一个扩展名为 GlobalUsing.g.cs 的文件，通常在 obj 文件夹的子目录中，其中包含了多个全局 using 指令，如代码清单 5.9 所示。

代码清单 5.9　ImplicitUsings 设为 enable 后生成的全局 using 指令[①]

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

每一行都是一个全局 using 指令（注释行除外），它告诉编译器对于在指定的命名空间中出现的任何类型，都隐含命名空间限定符。例如，全局 using global::System 允许像 Console.WriteLine(...) 这样的写法，而不必使用完全限定形式，即 System.Console.WriteLine(...)，因为 Console 是在 System 命名空间中定义的。

当然，也可以提供自己的全局 using 声明。例如，假定经常都要使用 System.Text.StringBuilder 类（曾在第 2 章简单地提及），那么可以为 System.Text 命名空间提供一个全局 using 指令。然后，在整个项目的范围内，都可以直接使用 StringBuilder 来引用该类型（参见代码清单 5.10），同时仍然依赖对 System 命令空间的隐式 using。

代码清单 5.10　为 StringBuilder 添加 global using 指令

```
global using System.Text;
// ...
public class Program
{
    public static void Main()
    {
```

[①] 在这个代码清单中，注意在自动生成的全局 using 语句中，使用了 global::作为命名空间前缀，例如 global using global::System。本章稍后会讨论命名空间别名限定符。

```
        // new();的用法请参见第 6 章
        StringBuilder name = new();

        Console.Write("请输入你的名字: ");
        name.Append(Console.ReadLine()!.Trim());

        Console.Write("请输入你的中间名首字母: ");
        name.Append($" { Console.ReadLine()!.Trim('.').Trim() }." );

        Console.Write("请输入你的姓氏: ");
        name.Append($" { Console.ReadLine()!.Trim() }");

        Console.WriteLine($"你好，{name}! ");
    }
}
```

全局 using 指令不管在哪代码文件中出现，都会应用于整个项目。虽然 C# 允许多次使用相同的全局 using 声明，但这样做是多余的。因此，最好是将所有全局 using 指令放在一个名为 Usings.cs 的文件中。统一位置后，还方便所有参与项目的人放置或删除这些声明。

除了应用于整个项目的全局 using 指令，C#还支持仅应用于当前文件的其他 using 指令。然而，全局 using 声明必须出现在其他任何（非全局）using 指令之前，后者会在介绍了.csproj 文件的 Using 元素之后说明。

5.3.4 .csproj Using 元素

还可以在项目文件（.csproj）中使用 Using 元素来指定全局 using 声明，如代码清单 5.11 所示。

代码清单 5.11 带有 Using 元素的示例.NET 控制台项目文件

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <Using Include="System.Net" />
    <Using Static="true" Include="System.Console"/>
  </ItemGroup>
</Project>
```

要注意的一个重点是，与 ImplicitUsings 元素不同，Using 元素是 ItemGroup 元素的子元素，而不是 PropertyGroup 的元素。另外，由于为 System.Net 设置了 Using 元素，所以就

不再需要完全限定 System.Net 命名空间中的 HttpClient 类。相反，在生成项目时，编译器会生成一个全局 using System.Net 指令。本例还有一个 Using 元素包含了一个 Static 属性，我们将在下一节进一步解释。

> **高级主题：嵌套 using 指令**
>
> using 指令不仅可以在文件顶部使用，还可以在命名空间声明的顶部使用。例如，声明新命名空间 EssentialCSharp 时，可以在该声明的顶部添加 using 指令，如代码清单 5.12 所示。
>
> **代码清单 5.12　在命名空间声明中使用 using 指令**
>
> ```
> namespace EssentialCSharp
> {
>     using System;
>     class HelloWorld
>     {
>         static void Main()
>         {
>             // 因为上方的 using 指令，所以
>             // 不需要用 System 限定 Console 类。
>             Console.WriteLine("你好，我的名字是 Inigo Montoya。");
>         }
>     }
> }
> ```
>
> 在文件顶部和命名空间声明的顶部使用 using 指令的区别在于，后者的 using 指令只在当前声明的命名空间内有效。如果在 EssentialCSharp 命名空间前后声明了新的命名空间，那么新命名空间不会受别的命名空间中的 using System 指令的影响。但是，我们很少写这样的代码，尤其是根据约定，每个文件只应该有一个类型声明。

5.3.5 using static 指令

using 指令允许省略命名空间限定符来简化类型名称。而 using static 指令允许将命名空间和类型名称都省略，只需写静态成员名称。例如，using static System.Console 指令允许直接写 WriteLine()，而不必写完全限定名称 System.Console.WriteLine()。基于这个技术，代码清单 5.2 可以改写为代码清单 5.13。

代码清单 5.13　using static 指令

```
using static System.Console;
```

```
public class HeyYou
{
    public static void Main()
    {
        string? firstName;  // 用于存储名字的变量
        string? lastName;   // 用于存储姓氏的变量

        WriteLine("嘿，你！");

        Write("请输入你的名字：");
        firstName = ReadLine() ?? string.Empty;

        Write("请输入你的姓氏：");
        lastName = ReadLine() ?? string.Empty;

        WriteLine(
            $"你的全名是{firstName} {lastName}。");
    }
}
```

本例不会损失代码的可读性。WriteLine()，Write()和 ReadLine()明显与控制台的操作相关。代码显得比以往更简单、更清晰（虽然可能有争议）。

但这并非绝对，有的类定义了重叠的行为名称（方法名），例如文件和目录都提供了 Exists()方法。在定义了 using static 指令的前提下，如果直接调用 Exists()，那么无利于澄清。类似地，如果你写的类定义了行为名称重叠的成员，例如 Display()和 Write()，读者也会感到困惑。

编译器不允许这种歧义。两个成员如果具有相同签名（通过 using static 指令或者是单独声明的成员），调用它们时就会产生歧义，会造成编译错误。

注意，C# 10 或更高版本还支持全局静态 using 指令，例如：

```
global using static System.Console;
```

类似地，可以在.csproj 文件中配置全局 using static 指令，例如：

```
<Using Static="true" Include="System.Console"/>
```

这在代码清单 5.11 中已经展示过了。

5.3.6 使用别名

从 C# 12.0 开始，using 指令还允许为命名空间以及包括元组、指针（第 23 章）、数组类型和泛型类型（第 12 章）在内的任何类型设置**别名**。别名是在 using 指令起作用的范围内可以使用的替代名称。别名两个最常见的用途是消除两个同名类型的歧义以及缩写长名称。

例如在代码清单 5.14 中，CountDownTimer 别名引用了 Timers.Timer 类型。仅添加 using System.Timers 指令还不足以避免对 Timer 类型的完全限定，原因是 System.Threading 也包含一个 Timer 类型，所以在代码中直接写 Timer 会产生歧义。

代码清单 5.14　声明类型别名

```
using CountDownTimer = System.Timers.Timer;
using StartStop = (DateTime Start, DateTime Stop);

public class HelloWorld
{
    public static void Main()
    {
        CountDownTimer timer;
        StartStop startStop;
        // ...
    }
}
```

代码清单 5.14 将全新名称 CountDownTimer 作为别名。但是，也可将别名指定为 Timer，如代码清单 5.15 所示。

代码清单 5.15　声明同名的类型别名

```
// 声明别名 Timer 来引用 System.Timers.Timer，
// 以避免代码与 System.Threading.Timer 产生歧义。
using Timer = System.Timers.Timer;

public class HelloWorld
{
    public static void Main()
    {
        Timer timer;

        // ...
    }
}
```

由于 Timer 现在是别名，所以“Timer”引用不会产生歧义。这个时候，如果要引用 System.Threading.Timer 类型，那么必须完全限定或者定义不同的别名。

当然，C# 10 及更高版本还支持全局别名 using 指令，例如：

```
global using Timer = Timers.Timer;
```

和其他所有全局 using 指令一样，还可以在.csproj 文件中配置全局别名 using 指令：

```
<Using Include = "System.Timers.Timer" Alias = "Timer" />
```

5.4 Main()方法的返回值和参数

到目前为止，可执行文件（可执行体）的所有Main()方法采用的都是最简单的声明。这些Main()方法声明不包含任何参数或非 void 返回类型。但是，C#支持在执行程序时提供命令行参数，并允许从 Main()方法返回状态标识符。

“运行时”通过一个 string 数组参数将命令行参数传给 Main()。要获取传递的这些参数，访问数组就可以了，代码清单 5.16 和输出 5.3 对此进行了演示。在本例中，第一次运行程序未传递任何参数，第二次运行则传递了两个参数。程序的作用是下载指定 URL 处的资源。第一个命令行参数指定URL，第二个则指定要用什么文件名来保存下载的资源。代码从一个 switch 语句开始，根据参数数量（args.Length）来采取不同操作：

1. 如果没有两个参数，就显示一条错误消息，指出必须提供 URL 和用于保存资源的文件名；

2. 如果有两个参数，表明用户提供了 URL 和文件名。

代码清单 5.16　向 Main()方法传递命令行参数

```
public class Program
{
    public static int Main(string[] args)
    {
        int result;
        if (args?.Length != 2)
        {
            // 必须提供两个（而且只能是两个）参数，所以报错
            Console.WriteLine(
                "错误: 必须指定"
                + "URL 和文件名");
            Console.WriteLine(
                "用法: Downloader.exe <URL> <文件名>");
            result = 1;
        }
        else
        {
```

```
            string urlString = args[0];
            string fileName = args[1];
            HttpClient client = new();
            byte[] response =
                client.GetByteArrayAsync(urlString).Result;
            client.Dispose();
            File.WriteAllBytes(fileName, response);
            Console.WriteLine($"已从'{urlString}'下载'{fileName}'。");
            result = 0;
        }
        return result;
    }
}
```

输出 5.3

```
>Downloader
错误：必须指定 URL 和文件名
用法：Downloader.exe <URL> <文件名>
>Downloader https://bookzhou.com index.html
已从'https://bookzhou.com'下载'index.html'。
```

成功获取用于保存资源的文件名，就用它保存从 URL 下载的资源。否则，应显示帮助文本来指出正确用法。另外，Main()方法还会返回一个 int，而不是像往常那样返回 void。返回值对于 Main()声明来说是可选的。但如果有返回值，程序就可以将状态码返回给调用者（比如脚本或批处理文件）。根据约定，非零返回值代表出错。

虽然所有命令行参数都可以通过字符串数组传给 Main()，但有时需要从非 Main()的方法中访问那些参数。这时可以使用 System.Environment.GetCommandLineArgs()方法返回由命令行参数构成的数组。该数组和通过 Main(string[] args)传入的 args 数组唯一的区别在于，第一个元素是可执行文件名，而不是第一个命令行参数。

为了能从网上下载文件，代码清单 5.16 使用了一个 System.Net.Http.HttpClient 对象，但代码中只需写 HttpClient，因为它的命名空间已由.csproj 文件中的 ImplicitUsings 元素导入了。

> **高级主题：消除多个 Main()方法的歧义**
>
> 如果一个程序包含两个带有 Main()方法的类，那么可以指定使用哪一个作为入口点。在 Visual Studio 中，可以在解决方案资源管理器中右击项目名称，并选择“属性”。然后，在“应用程序”标签页中，可以从“启动对象”中选择用哪个类的 Main()方法来启动程序。这个操作会在.csproj 文件的 PropertyGroup 中添加一个附加的元素：
>
> ```
> <StartupObject>
> ```

```
        AddisonWesley.Michaelis.EssentialCSharp.Shared.Program
    </StartupObject>
```

在命令行上 build 时，也可以用你希望的值来设置 StartupObject 属性，例如：

```
dotnet build /p:StartupObject=AddisonWesley.Program2
```

其中，AddisonWesley.Program2 是包含所选 Main()方法的命名空间和类。事实上，.csproj 文件的 PropertyGroup 区域的任何项都可以通过这种方式在命令行上指定。

初学者主题：调用栈和调用点

代码执行时，方法可能调用其他方法，其他方法可能调用更多方法，以此类推。在代码清单 5.4 的简单情况中，Main()调用 GetUserInput()，后者调用 Console.ReadLine()，后者又在内部调用更多方法。每次调用新方法，“运行时”都创建一个“栈帧”或“活动帧”，其中包含的信息包括传给新调用的方法的实参、新调用的方法自己的局部变量以及方法返回时应该从哪里恢复等。由此形成的一系列栈帧称为**调用栈**（call stack）[①]。随着程序越来越复杂，每个方法调用另一个方法时，这个调用栈都会变大。但当调用结束时，调用栈又会发生收缩，直到调用另一个方法。我们用**栈展开**（stack unwinding）[②]一词描述从调用栈中删除栈帧的过程。栈展开的顺序通常与方法调用的顺序相反。方法调用完毕，控制会返回**调用点**（call site），也就是最初发出方法调用的位置。

5.5 顶级语句

截至 C# 9.0，C#中的所有可执行代码都必须放在类型定义和成员（例如类型内的方法）中。但是，正如代码清单 1.1 展示的那样，现在已经不需那样做了。相反，现在可以在代码文件中直接写**顶级语句**（top-level statements），这些语句独立于任何类型定义，甚至可以没有 Main()方法。但是，只允许在一个文件中写这样的语句，而且必须是程序中执行的第一批语句——等同于以前在 Main()方法中写的那些语句。事实上，如果你写顶级语句，那么编译器会自动生成一个名为 Program 的类，在该类中包装顶级语句，并将它们放入一个

---

① async 或迭代器方法除外，它们的活动记录转移到堆上。

② 译注：unwind 一般翻译成“展开”，但这并不是一个很好的翻译。wind 和 unwind 源于生活。把线缠到线圈上称为 wind；从线圈上松开称为 unwind。同样地，调用方法时压入栈帧，称为 wind；方法执行完毕弹出栈帧，称为 unwind。

Main()方法中。此外，该方法有一个上下文关键字 args，相当于 Main 方法的 string[] args 参数。

使用顶级语句，C#表面上允许语句出现在方法外部。有弱结构化语言编程背景的开发人员在刚开始接触 C#时，会发现像这样的写法非常熟悉[①]。但无论如何，最后编译时还是会这些语句移到Main()方法中。因此，最终的结果是，在底层的 CIL 中，所有语句都还是在类型定义和类型成员内。由于 C#编译器将顶级语句转移到自动生成的 Program 类所定义的 Main()方法中，所以如果尝试定义另一个名为 Program 的类，那么编译器会报错[②]。对顶级语句的另一个限制是，同一个文件中的任何类型定义都必须放在顶级语句后面。

在包含顶级语句的文件中，还可以包含所谓的**顶级方法**，这些方法也可以独立于类型定义。

在 Visual Studio 中运行某些“创建新项目”向导时，会提供一个“不使用顶级语句”选项，该选项允许你选择是使用简化的“顶级语句”版本，还是使用显式的代码结构。默认是不勾选该选项，从而生成如代码清单 1.1 所示的代码（没有显式的类或 Main 方法）。类似地，在 dotnet 命令行上，某些项目模板（例如，在执行 dotnet new Console 命令时提供的 Console 参数）通常支持一个--use-program-main 选项，可以用它禁用顶级语句，以传统方式生成代码结构。但是，顶级语句仅对具有入口点的项目可用；换言之，要具有Main()方法。编译器不允许为没有 Main()方法的程序使用顶级语句（例如类库）。

顶级语句主要是为了在写简单程序时避免不必要的套路，它降低了初学者的上手难度。在引入顶级语句之前，即使程序只有一个语句。也必须写一个类型定义和一个Main()方法。有了顶级语句，就可以避免这些套路。此外，顶级语句方便我们将看似完整的 C#代码片段嵌入到文本中，比如 Polyglot Notebooks[③]（*https://github.com/dotnet/interactive*）或者本书英文版手稿的在线版本（*https://essentialcsharp.com*）。

5.6 高级方法参数

本章之前一直是通过方法的 return 语句返回数据。本节将解释方法如何通过自己的参数返回数据，以及方法如何获取数量可变的参数。

[①] 译注：这使 C#更接近脚本语言的使用体验。

[②] 正如编译器的错误消息所提示的那样，可以用 partial 修饰符将这个新的 Program 类定义为分部类，表示这两部分定义将合并到同一个 Program 类中。详情将在第 6 章的“分部类”中解释。

[③] 译注：可以作为 Visual Studio Code 的插件安装。

5.6.1 值参数

参数默认**传值**（pass by value）。换言之，参数值会复制（拷贝）到目标参数中。以代码清单 5.17 为例，在调用 Combine()时，Main()正在使用的每个变量值都会复制给 Combine()方法的参数。输出 5.4 展示了结果。

代码清单 5.17　以传值方式传递变量

```
public class Program
{
    public static void Main()
    {
        // ...
        string fullName;
        string driveLetter = "C:";
        string folderPath = "Data";
        string fileName = "index.html";
        fullName = Combine(driveLetter, folderPath, fileName);
        Console.WriteLine(fullName);
        // ...
    }
    static string Combine(
        string driveLetter, string folderPath, string fileName)
    {
        string path;
        path = string.Format("{1}{0}{2}{0}{3}",
            Path.DirectorySeparatorChar,
            driveLetter, folderPath, fileName);
        return path;
    }
}
```

输出 5.4

```
C:\Data\index.html
```

Combine()方法返回前，即使故意将 null 值赋给 driveLetter，folderPath 和 fileName 等变量，Main()中对应的变量仍会保持其初始值不变，因为在调用方法时，只是将变量的值复制了一份给方法。调用栈在一次调用的末尾“展开”（unwind）的时候，复制的数据会被丢弃。

> 初学者主题：匹配调用者变量与参数名

在代码清单 5.17 中，调用者 Main()中的变量名与被调用方法 Combine()中的参数名是匹配的。但这只是为了增强可读性，名称是否匹配与方法调用的行为无关。被调用方法的参数和发出调用的方法的局部变量在不同的声明空间中，相互之间没有任何关系。

高级主题：比较引用类型与值类型

就本节的主题来说，传递的参数是值类型还是引用类型并不重要。相反，我们主要关心的是被调用的方法是否能将值写入调用者的原始变量。由于现在是生成原始值的拷贝，所以无论怎么更改都影响不到调用者的变量。但不管怎样，都有必要理解值类型和引用类型的变量的区别。

从名字就可以看出，对于引用类型的变量，它的值是对数据实际存储位置的引用。“运行时”如何表示引用类型变量的值，这是“运行时”的实现细节。一般都是用数据实际存储的内存地址来表示，但并非一定如此。

如果引用类型的变量以传值方式传给方法，复制的就是引用（地址）本身。这样虽然在被调用的方法中还是更改不了引用（地址）本身，但可以更改地址处的数据。

相反，对于值类型的参数，参数获得的是值的拷贝，所以被调用的方法无论怎么折腾这些拷贝，都影响不了调用者的变量。

5.6.2 引用参数(ref)

下面来看看代码清单 5.18 的例子，它调用方法来交换两个值，输出 5.5 展示了结果。

代码清单 5.18　以传引用的方式传递变量

```
string first = "你好";
string second = "再见";

Swap(ref first, ref second);


Console.WriteLine(
    $@"first = ""{first}"", second = ""{second}""");
        // ...
```

```
// 方法定义可以放到顶级语句之前，但类型定义不可以
static void Swap(ref string x, ref string y)
{
    string temp = x;
    x = y;
    y = temp;
}
```

输出 5.5

```
first = "再见", second = "你好"
```

赋给 first 和 second 的值被成功交换。这要求以**传引用**（pass by reference）的方式传递变量。比较本例的 Swap()调用与代码清单 5.17 的 Combine()调用，不难发现两者最明显的区别就是本例在参数数据类型前使用了关键字 ref，这使参数以传引用方式传递，被调用的方法可用新值更新调用者的变量。

如果被调用的方法将参数指定为 ref，调用者调用该方法时提供的实参也应该是附加了 ref 前缀的变量（而不能传递值）。这样，调用者就显式确认了目标方法可以对它接收到的任何 ref 参数进行重新赋值。此外，调用者应初始化传引用的局部变量，因为被调用的方法可能直接从 ref 参数读取数据而不先对其进行赋值。例如，在代码清单 5.18 中，temp 直接从 first 获取数据，认为 first 变量已由调用者初始化。事实上，ref 参数只是传递的变量的别名。换言之，作用只是为现有变量分配一个参数名，而非创建新变量并将实参的值拷贝给它。

> **注意：** ref 修饰符使参数引用栈上现有的一个变量，而不是创建新变量，并将参数值复制到参数中。

5.6.3 输出参数(out)

如前所述，用作 ref 参数的变量必须在传给被调用的方法前赋值（初始化），因为被调用的方法可能直接从变量中读取值。例如，上一个例子的 Swap 方法必须读写传给它的变量。但是，方法经常要获取一个变量引用，并只是向变量写入而不读取。这时更安全的做法是以传引用的方式传入一个未初始化的局部变量。

为此，代码需要用关键字 out 修饰参数类型。例如代码清单 5.19 的 TryGetPhoneButton()方法，它返回与字符对应的电话按键。运行结果如输出 5.6 所示。

代码清单 5.19　仅传出（输出）的变量

```
public static int Main(string[] args)
{
    if (args.Length == 0)
    {
        Console.WriteLine(
            "用法：ConvertToPhoneNumber.exe <一个英文短语>");
        Console.WriteLine(
            "'_'表示无标准电话按键");
        return 1;
    }

    foreach (string word in args)
    {
        foreach (char character in word)
        {
            if (TryGetPhoneButton(character, out char button))
            {
                Console.Write(button);
            }
            else
            {
                Console.Write('_');
            }
        }
    }
    Console.WriteLine();
    return 0;
}

static bool TryGetPhoneButton(char character, out char button)
{
    bool success = true;
    switch (char.ToLower(character))
    {
        case '1':
            button = '1';
            break;
        case '2':
        case 'a':
        case 'b':
        case 'c':
            button = '2';
            break;

        // ...

        case '-':
            button = '-';
            break;
        default:
```

```
                // 设置 button 来指示一个无效的值
                button = '_';
                success = false;
                break;
            }
            return success;
        }
```

输出 5.6

```
ConvertToPhoneNumber.exe BookZhou.com
26659468_266
```

在本例中，如果能成功判断与 character 对应的电话按键，TryGetPhoneButton()方法就返回 true。方法还使用 out 参数 button 返回对应的按键。

out 参数功能上与 ref 参数完全一致，唯一区别是 C#语言对别名变量的读写方式有不同的规定。如果参数被标记为 out，那么编译器会核实在方法所有正常返回的代码路径（也就是不抛出异常的路径）中，是否都对该参数进行了赋值。如果发现某个代码执行路径没有对 button 赋值，编译器就会报错，指出代码没有对 button 进行初始化。在代码清单 5.19 中，方法最后将下划线字符赋给 button，因为即使无法判断正确的电话按键，也必须对 button 进行赋值。

使用 out 参数时，一个常见的编码错误是忘记在使用前声明 out 变量。从 C# 7.0 起，可以在调用方法前以内联的形式声明 out 变量。代码清单 5.19 在 TryGetPhoneButton(character, out char button)中就使用了该功能，button 变量完全不需要事先声明。而在 C# 7.0 之前，必须先声明 button 变量，再用 TryGetPhoneButton(character, out button)调用方法。

另外，从 C# 7.0 开始允许完全放弃 out 参数。例如，你可能只想知道某字符是不是有效电话按键，而不需要实际返回对应的数值。这时就可以用下划线放弃 button 参数（称为弃元），即 TryGetPhoneButton(character, out _)。

在 C# 7.0 元组语法之前，开发人员通过声明一个或多个 out 参数来解除方法只能有一个返回类型的限制。例如，为了返回两个值，可以正常返回一个，另一个写入作为 out 参数传递的别名变量。虽然这个做法既常见也合法，但通常都有更好的方案能达到相同目的。例如，用 C# 7.0 写代码时，为了返回两个或更多值，应该首选元组语法。而在 C# 7.0 之前，可以考虑改成两个方法，每个方法返回一个值。如果非要一次返回多个，那么还是可以使用 ValueTuple 类型，只是当然不能使用 C# 7.0 语法。

> **注意：**所有正常的代码路径都必须对 out 参数赋值。

5.6.4 只读传引用(in)

C# 7.2 支持以传引用的方式传入只读值类型。不是创建值类型的拷贝并使方法能修改拷贝。相反，只读传引用造成值类型以传引用的方式传给方法。不仅不会每次调用方法都创建值类型的拷贝，而且被调用的方法不能修改值类型。换言之，其作用是在传值时减少拷贝量，同时把它标识为只读，从而增强性能。该语法要为参数添加 in 修饰符。例如：

```
int Method(in int number) { ... }
```

使用 in 修饰符，方法中对 number 的任何重新赋值操作（例如，number++）都会造成编译错误，并报告 number 只读。

5.6.5 返回引用

从 C# 7.0 开始还允许返回对变量的引用。例如，代码清单 5.20 定义了一个方法来返回图片中的第一个红眼像素。

代码清单 5.20　return ref 和 ref 局部变量声明

```
// 返回一个引用
public static ref byte FindFirstRedEyePixel(byte[] image)
{
    // 执行图像检查(也许通过机器学习)
    for (int counter = 0; counter < image.Length; counter++)
    {
        if (image[counter] == (byte)ConsoleColor.Red)
        {
            return ref image[counter];
        }
    }
    throw new InvalidOperationException("没有像素是红色的。");
}

public static void Main()
{
    byte[] image = new byte[254];
    // 加载图像
    int index = new Random().Next(0, image.Length - 1);
    image[index] =
        (byte)ConsoleColor.Red;
    Console.WriteLine(
        $"image[{index}]={(ConsoleColor)image[index]}");
    // ...

    // 获取对第一个红色像素的引用
```

```
    ref byte redPixel = ref FindFirstRedEyePixel(image);
    // 把它更新为黑色
    redPixel = (byte)ConsoleColor.Black;

    Console.WriteLine(
        $"image[{index}]={(ConsoleColor)image[redPixel]}");
}
```

如代码清单 5.20 所示，通过返回对变量的引用，调用者可以将像素更新为不同颜色。检查对数组的更新证明值现已变成黑色。

返回引用有两个重要的限制，两者都和对象生存期有关：1. 对象引用在仍被引用时不应被垃圾回收；2. 在不存在对它们的任何引用后，就不应消耗内存。为了符合这些限制，从方法返回引用时只能返回：

- 对字段或数组元素的引用
- 其他返回引用的属性或方法
- 作为参数传给“返回引用的方法”的引用

例如，FindFirstRedEyePixel()返回对一个 image 数组元素的引用，该引用是传给方法的参数。类似地，如果图片作为类的字段存储，那么可以返回对字段的引用：

```
byte[] _Image;
public ref byte[] Image { get { return ref _Image; } }
```

此外，ref 局部变量被初始化为引用一个特定变量，以后不能修改它来引用其他变量。

返回引用时要注意几点：

- 如果决定返回引用，就必须返回一个引用。以代码清单 5.20 为例，即使不存在红眼像素，也必须返回一个字节引用。找不到？那么只有抛出异常。相反， 如采取传引用参数的方式，就可以不修改参数，只是返回一个 bool 值代表成功，许多时候这样做更佳。
- 声明一个引用局部变量的同时必须初始化它。为此，需要将方法返回的引用赋给它，或者将一个变量引用赋给它。

```
ref string text; // 会报错
```

- 虽然从 C# 7.0 开始允许声明 ref 局部变量，但不允许声明 ref 字段（参见第 6 章的“实例字段”一节，更多地了解字段的声明）。

```
class Thing { ref string _Text; /* 会报错 */ }
```

- 自动实现的属性不能声明为引用类型（参见第 6 章的“属性”一节，更多地了解属性的声明）。

```
    class Thing { ref string Text { get; set; } /* 会报错 */ }
```

- 从属性返回引用是允许的。

```
class Thing { string _Text = "Inigo Montoya";
ref string Text { get { return ref _Text; } } }
```

- ref 局部变量不能用值（比如 null 或常量）来初始化。必须将返回引用的成员赋给它，或者将局部变量、字段或数组赋给它：

```
ref int number = null; ref int number = 42; // 会报错
int local = 1; ref int a = ref local; // 不会报错
```

5.6.6 参数数组(params)

到目前为止，方法的参数数量都是在声明时确定好的。但是，我们有时希望参数数量可变。以代码清单 5.17 的 Combine()方法为例，它传递了驱动器号（driveLetter）、文件夹路径（folderPath）和文件名（fileName）等参数。如果路径中包含多个嵌套的文件夹，调用者希望将额外的文件夹连接起来以构成完整路径，那么应该如何写代码呢？或许最好的办法就是为文件夹传递一个字符串数组，其中包含不同的文件夹名称。但这会使调用代码变复杂，因为需要事先构造好数组并将数组作为参数传递。

为了简化编码，C#提供了一个特殊关键字，允许在调用方法时提供数量可变的参数，而不是事先就固定好参数数量。讨论方法声明前，先注意一下 Main()中的调用代码，如代码清单 5.21 和输出 5.7 所示。

代码清单 5. 21　传递长度可变的参数列表

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string fullName;

        // ...

        // 向 Combine()传递 4 个参数
        fullName = Combine(
            Directory.GetCurrentDirectory(),
            "bin", "config", "index.html");
        Console.WriteLine(fullName);

        // ...
```

```
        // 向 Combine()传递 3 个参数
        fullName = Combine(
            Environment.SystemDirectory,
            "Temp", "index.html");
        Console.WriteLine(fullName);

        // ...

        // 向 Combine()传递一个数组
        fullName = Combine(
            new string[] {
                $"{Environment.GetFolderPath(Environment.SpecialFolder.UserProfile)}",
                "Documents", "Web", "index.html" });
        Console.WriteLine(fullName);
        // ...
    }

    static string Combine(params string[] paths)
    {
        string result = string.Empty;
        foreach (string path in paths)
        {
            result = Path.Combine(result, path);
        }
        return result;
    }
}
```

输出 5. 7

```
C:\Data\mark\bin\config\index.html
C:\WINDOWS\system32\Temp\index.html
C:\Data\HomeDir\index.html
```

第一个 Combine()调用提供了 4 个参数。第二个只提供 3 个。最后一个调用传递一个数组来作为参数。换言之，Combine()方法接受数量可变的参数，要么是以逗号分隔的字符串参数，要么是单个字符串数组。前者称为方法调用的“展开”（expanded）形式，后者称为“正常”（normal）形式。

为了获得这样的效果，Combine()方法需要：

1. 在方法声明的最后一个参数前添加 params 关键字
2. 将最后一个参数声明为数组

像这样声明了**参数数组**之后，每个参数都作为参数数组的成员来访问。Combine()方法遍历 paths 数组的每个元素并调用 System.IO.Path.Combine()。该方法自动合并路径中的不同部分，并正确使用平台特有的目录分隔符。

参数数组要注意以下几点：

- 参数数组不一定是方法唯一的参数，但必须是最后一个。由于只能放在最后，所以最多只能有一个参数数组。
- 调用者可以指定和参数数组对应的零个实参，这会造成将包含零个数据项的一个数组作为参数数组传递。
- 参数数组是类型安全的；换言之，实参的类型必须兼容参数数组的元素类型。
- 调用者可以传递一个实际的数组，而不是传递以逗号分隔的参数列表。最终生成的 CIL 代码一样。
- 如果目标方法的实现要求一个最起码的参数数量，那么请在方法声明中显式指定必须提供的参数。这样一来，一旦遗漏必须的参数，就会导致编译器报错。这样就不必依赖运行时的错误处理。例如，使用 `int Max(int first, params int[] operands)`而不是 `int Max(params int[] operands)`，确保至少有一个整数实参传给 `Max()`。

使用参数数组，可以将数量可变的多个同类型参数传给方法。本章稍后的“方法重载”一节讨论了如何支持不同类型的、数量可变的参数。

设计规范

DO use parameter arrays when a method can handle any number—including zero—of additional arguments

如果方法能处理任何数量（包括零个）额外实参，那么**要**使用参数数组。

顺便说一句，我们在这个例子中写的 `Combine()`函数是一个人为的例子。其实可以直接利用 `System.IO.Path.Combine()`函数，它已进行了重载，能支持参数数组。

5.7 递归

“递归调用方法”或者“用递归实现方法”意味着方法调用它自身。有的时候，这是解决特定问题最简单的方式（例如，汉诺塔）。代码清单 5.22 统计目录及其子目录中的所有 C# 源代码文件（*.cs）的代码行数，结果如输出 5.8 所示。

代码清单 5.22　返回目录中所有.cs 文件中的代码行数

```
using System.IO;

public static class LineCounter
```

```
{
    // 使用第一个实参作为要搜索的目录，
    // 或者默认为当前目录。
    public static void Main(string[] args)
    {
        int totalLineCount = 0;
        string directory;
        if (args.Length > 0)
        {
            directory = args[0];
        }
        else
        {
            directory = Directory.GetCurrentDirectory();
        }
        totalLineCount = DirectoryCountLines(directory);
        Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines(string directory)
    {
        int lineCount = 0;
        foreach (string file in
            Directory.GetFiles(directory, "*.cs"))
        {
            lineCount += CountLines(file);
        }

        foreach (string subdirectory in
            Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }

        return lineCount;
    }

    private static int CountLines(string file)
    {
        string? line;
        int lineCount = 0;
        // 可以使用一个 using 语句改进，但目前还没有讲到
        FileStream stream = new(file, FileMode.Open);
        StreamReader reader = new(stream);
        line = reader.ReadLine();

        while (line != null)
        {
            if (line.Trim() != "")
            {
```

```
                    lineCount++;
                }
                line = reader.ReadLine();
            }

            reader.Dispose();  // 自动关闭流
            return lineCount;
        }
    }
```

输出 5.8

```
LineCounter.exe D:\CLRviaCShap2024
2048
```

程序首先将第一个命令行参数传给 DirectoryCountLines()。如果没有在命令行传递参数，就直接使用当前目录。在方法中，我们首先遍历指定目录中的所有文件，累加每个.cs 文件包含的源代码行数。处理好该目录之后，重新将子目录传给 DirectoryCountLines()方法以处理每个子目录。同样的过程会针对每个嵌套的子目录反复进行，直到再也没有更多子目录可供处理。

不熟悉递归的读者刚开始可能觉得非常繁琐。但事实上，递归通常都是最简单的编码模式，尤其是在和文件系统这样的层次化数据打交道的时候。不过，虽然可读性不错，但一般不是最快的实现。如果必须关注性能，开发者应该为递归实现寻求一种替代方案。至于具体如何选择，通常取决于想如何在可读性与性能之间取得平衡。

> **初学者主题：无限递归错误**
>
> 用递归实现方法时，一个常见的错误是在程序执行期间发生栈溢出（stack overflow）。这通常是由于**无限递归**造成的。假如方法持续调用自身，永远抵达不了标志递归结束的位置，就会发生无限递归。必须仔细检查每个使用了递归的方法，验证递归调用是有限而非无限的。
>
> 下面以伪代码的形式展示了一个常用的递归模式：
>
> ```
> M(x)
> {
>   if x 已达最小，不可继续分解
>     返回结果
>   else
>     (1) 采取一些操作使问题变得更小
>     (2) 递归调用 M 来解决更小的问题
>     (3) 根据(1)和(2)计算结果
> ```

返回结果

不遵守这个模式就可能出错。例如，如果不能将问题变得更小，或者不能处理所有可能的“最小”情况，就会递归个不停。

5.8 方法重载

上一节的代码清单 5.22 调用 DirectoryCountLines()方法来统计*.cs 文件中的源代码行数。但是，如果要统计*.h，*.cpp，或者*.vb 文件的代码行数，DirectoryCountLines()就无能为力了。我们希望有这样一个方法，它能获取文件扩展名作为参数，同时保留现有方法定义，以便默认处理*.cs 文件。

一个类中的所有方法都必须具有唯一的签名，C#依据方法名、参数数据类型或参数数量的差异来定义唯一性。注意，*方法返回类型不计入签名*。两个方法如果仅返回类型不同，那么会造成编译错误（即使返回的是两个不同的元组）。如一个类包含两个或多个同名方法，就会发生**方法重载**。对于重载的方法，参数数量以及/或者数据类型肯定有所不同。

注意：方法的唯一性取决于方法名、参数数据类型或参数数量的差异。

方法重载是一种**操作性多态**（operational polymorphism）。如果因数据变化而造成同一个逻辑操作具有许多（“多”）形式（“态”），就会发生“多态”。以 WriteLine()方法为例，可以向它传递一个格式字符串和其他一些参数，也可以只传递一个整数。两者的实现肯定不一样。但在逻辑上，对于调用者来说，该方法的作用就是输出数据。至于方法内部具体如何实现，调用者并不关心。代码清单 5.23 是一个例子，输出 5.9 展示了结果。

代码清单 5.23　使用重载统计代码文件的行数

```
public static class LineCounter
{
    public static void Main(string[] args)
    {
        int totalLineCount;

        if (args.Length > 1)
        {
            totalLineCount = DirectoryCountLines(args[0], args[1]);
        }
        else if (args.Length > 0)
        {
```

```
            totalLineCount = DirectoryCountLines(args[0]);
        }
        else
        {
            totalLineCount = DirectoryCountLines();
        }

        Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines()
    {
        return DirectoryCountLines(
            Directory.GetCurrentDirectory());
    }

    static int DirectoryCountLines(string directory)
    {
        return DirectoryCountLines(directory, "*.cs");
    }

    static int DirectoryCountLines(
        string directory, string extension)
    {
        int lineCount = 0;
        foreach (string file in
            Directory.GetFiles(directory, extension))
        {
            lineCount += CountLines(file);
        }

        foreach (string subdirectory in
            Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }

        return lineCount;
    }

    private static int CountLines(string file)
    {
        int lineCount = 0;
        string? line;
        // 可以使用一个 using 语句改进，但目前还没有讲到
        FileStream stream = new(file, FileMode.Open);
        StreamReader reader = new(stream);
        line = reader.ReadLine();
        while (line is not null)
        {
```

```
                if (line.Trim() != "")
                {
                    lineCount++;
                }
                line = reader.ReadLine();
            }

            reader.Dispose();  // 自动关闭流
            return lineCount;
        }
    }
```

输出 5.9

```
D:\CLRviaCSharp2024>LineCounter.exe
2048
D:\CLRviaCSharp2024>LineCounter.exe .\ *.xml
1928
```

方法重载的目的是提供调用方法的多种方式。如本例所示，在 Main()中调用 DirectoryCountLines()方法时，可以选择是否传递要搜索的目录和文件扩展名。

本例修改 DirectoryCountLines()的无参版本，让它调用单一参数的 int DirectoryCountLines(string directory)。这是实现重载方法的常见模式，基本思路是：开发者只需在一个方法中实现核心逻辑，其他所有重载版本都调用那个方法。如果核心实现需要修改，在一个位置修改就可以了，不必兴师动众修改每一个实现。通过方法重载来支持可选参数时，该模式尤其有用。注意这些参数的值在编译时不能确定，不适合使用从 C# 4.0 开始引入的“可选参数”功能。

> **注意：**将核心功能放到单一方法中供其他重载方法调用，以后就只需在核心方法中修改，其他方法将自动受益。

5.9 可选参数

C# 4.0 新增了对**可选参数**的支持。声明方法时，事先将常量值赋给参数，以后调用方法时就不必再为每个参数提供实参了，如代码清单 5.24 所示。

代码清单 5.24　使用可选参数的方法

```
    public static class LineCounter
```

```
{
    public static void Main(string[] args)
    {
        int totalLineCount;
        if (args.Length > 1)
        {
            totalLineCount =
            DirectoryCountLines(args[0], args[1]);
        }
        else if (args.Length > 0)
        {
            totalLineCount = DirectoryCountLines(args[0]);
        }
        else
        {
            totalLineCount = DirectoryCountLines();
        }
        Console.WriteLine(totalLineCount);
    }

    static int DirectoryCountLines()
    {
        // ...
    }

    /*
    static int DirectoryCountLines(string directory)
    { ... }
    */

    static int DirectoryCountLines(
        string directory, string extension = "*.cs")
    {
        int lineCount = 0;
        foreach (string file in
        Directory.GetFiles(directory, extension))
        {
            lineCount += CountLines(file);
        }
        foreach (string subdirectory in
        Directory.GetDirectories(directory))
        {
            lineCount += DirectoryCountLines(subdirectory);
        }
        return lineCount;
    }
    private static int CountLines(string file)
    {
        // ...
    }
```

```
}
```

在代码清单 5.24 中，DirectoryCountLines()方法的单参数版本已被移除（注释掉），但 Main()方法似乎仍在调用该方法（指定一个参数）。如果调用时不指定 extension（扩展名）参数，就使用声明时赋给 extension 的值（本例是*.cs）。这样一来，在调用代码时就可以不为该参数传递值了。而在 C# 3.0 和更早的版本中，将不得不声明一个额外的重载版本。注意，可选参数一定要放在所有必须参数（无默认值的参数）后面。另外，默认值必须是常量或者其他能在编译时确定的值，这一点极大限制了“可选参数”的应用。例如，不能像下面这样声明方法：

```
DirectoryCountLines(
    string directory = Environment.CurrentDirectory,
    string extension = "*.cs")
```

这是由于 Environment.CurrentDirectory 不是常量。而"*.cs"是，所以 C#允许它作为可选参数的默认值。

设计规范

DO provide good defaults for all parameters where possible.

要尽量为所有参数提供好的默认值。

DO provide simple method overloads that have a small number of required parameters.

要提供简单的方法重载，必须的参数数量要少。

CONSIDER organizing overloads from the simplest to the most complex.

考虑从最简单到最复杂组织重载。

C# 4.0 新增的另一个跟方法调用有关的功能是**具名参数**。调用者可利用具名参数为一个参数显式赋值，而不是像以前那样只能依据参数顺序来决定哪个值赋给哪个参数，如代码清单 5.25 所示。

代码清单 5. 25　调用方法时指定参数名

```
public static void Main()
```

```
{
    DisplayGreeting(firstName: "Inigo", lastName: "Montoya");
}

public static void DisplayGreeting( string firstName,
    string? middleName = null,
    string? lastName = null )
{
    // ...
}
```

在代码清单 5.25 中，从 Main()中调用 DisplayGreeting()时，我们使用具名参数为一个参数显式赋值。至于剩下的两个可选参数（middleName 和 lastName），则只用具名参数为 lastName 显式提供了值，另一个保留它的默认值。

如果一个方法有大量参数，其中许多都可选（访问 Microsoft COM 库时很常见），那么具名参数语法肯定能带来不少便利。但要注意的是，它的代价是牺牲了方法接口的灵活性。过去（至少就 C#来说）参数名可自由更改，不会造成调用代码无法编译的情况。但在添加了具名参数后，参数名就成为方法接口的一部分。更改名称会导致使用具名参数的代码无法编译。

设计规范

DO treat parameter names as part of the API, and avoid changing the names if version compatibility between APIs is important.

要将参数名视为 API 的一部分；如果关心 API 之间的版本兼容性，那么应该避免改变名称。

对于有经验的 C#开发人员，这是一个令人吃惊的限制。但该限制自.NET 1.0 开始就作为 CLS 的一部分存在下来了。另外，Visual Basic 一直都支持用具名参数调用方法。所以，库开发人员应该早已养成了不更改参数名的习惯，这样才能成功地与其他.NET 语言进行互操作，不会因为版本的变化而造成自己开发的库失效。C# 4.0 只是像其他许多.NET 语言早就要求的那样，对参数名的更改进行了相同的限制。

方法重载、可选参数和具名参数这几种技术一起使用，可能很难一眼看出最终调用的是哪个方法。只有在所有参数（可选参数除外）都恰好有一个对应的实参（不管是根据名称还是位置），而且该实参具有兼容类型的情况下，才说一个调用**适用**于（兼容于）一个方法。虽然这限制了可调用方法的数量，但不足以唯一性地标识方法。为了进一步区分方法，编译器只使用调用者显式标识的参数，忽略调用者没有指定的所有可选参数。所以，假如因为其中一个方法有可选参数，造成两个方法都适用，编译器最终将选择无可选参数的方法。

高级主题：方法解析

编译器从一系列“适用”的方法中选择最终调用的方法时，依据的是哪个方法最具体。假定有两个适用的方法，每个都要求将实参隐式转换成形参的类型，那么最终选择的是形参类型最具体（派生程度最大）的方法。

例如，假定调用者传递一个 int，那么接受 double 的方法将优先于接受 object 的方法。这是由于 double 比 object 更具体。有不是 double 的 object，但没有不是 object 的 double，所以 double 更具体。

如果有多个适用的方法，但无法从中挑选出最具唯一性的，编译器就会报错，指明调用存在歧义。

例如，给定以下方法：

```
static void Method(object thing) { }
static void Method(double thing) { }
static void Method(long thing) { }
static void Method(int thing) { }
```

调用 Method(42)会解析成 Method(int thing)，因为存在从实参类型到形参类型的完全匹配。如果删除该版本，那么重载解析机制会选择 long 版本，因为 long 比 double 和 object 更具体。

C# 规范包含额外的规则来决定 byte，ushort，uint，ulong 和其他数值类型之间的隐式转换。但在写程序时，最好还是使用显式转型，方便别人理解你想调用哪个目标方法。

5.10 用异常实现基本错误处理

本节将探讨如何利用**异常处理**机制来解决错误报告的问题。方法利用异常处理将与错误相关的信息传给调用者，同时不需要使用返回值或显式提供任何参数。代码清单 5.26 略微修改了第 1 章的 HeyYou 程序（代码清单 1.18）。这次不是请求用户输入姓氏，而是请求输入年龄。输出 5.11 展示了结果。

代码清单 5. 26　将 string 转换成 int

```
public static void Main()
{
    string? firstName;
```

```
        string ageText;
        int age;

        Console.WriteLine("嘿，你！");

        Console.Write("请输入你的名字：");
        firstName = Console.ReadLine();

        Console.Write("请输入你的年龄：");
        // 假设非空
        ageText = Console.ReadLine()!;
        age = int.Parse(ageText);

        Console.WriteLine(
            $"你好，{ firstName }！你有{ age * 12 }个月大了。");
    }
```

输出 5.11

```
嘿，你！
请输入你的名字：Inigo
请输入你的年龄：42
你好，Inigo！你有 504 个月大了。
```

Console.ReadLine()的返回值存储到 ageText 变量中，然后传给 int 数据类型的 Parse()方法。该方法获取代表数字的 string 值并转换为 int。

> **初学者主题：42 作为字符串和整数**
>
> C#要求每个非空值都有一个良好定义的类型。换言之，数据不仅值很重要，它的类型也很重要。所以，字符串"42"和整数 42 完全是两码事。字符串由'4'和'2'这两个字符构成，而 int 是数值 42。

基于转换好的字符串，Console.WriteLine()语句以月份为单位打印年龄（age*12）。

但是，用户完全可能输入一个无效的整数字符串。例如，输入“四十二”会发生什么？Parse()方法不能完成这样的转换。它希望用户输入只含数字的字符串。如果 Parse()方法接收到无效值，它需要某种方式将这一事实反馈给调用者。

5.10.1 捕捉错误

为了通知调用者参数无效，int.Parse()会**抛出异常**。抛出异常会终止执行当前分支，跳到调用栈中用于处理异常的第一个代码块。

由于当前尚未提供任何异常处理，所以程序会向用户报告发生了未处理的异常。如果系统中没有注册任何调试器，那么错误信息会出现在控制台上，如输出 5.11 所示。

输出 5.11

```
嘿，你！
请输入你的名字：Inigo
请输入你的年龄：四十二

Unhandled exception. System.FormatException: The input string '四十二' was not in a correct
format.
   at System.Number.ThrowFormatException[TChar](ReadOnlySpan`1 value)
   at System.Int32.Parse(String s)
   at ... ExceptionHandling.Main() ...
```

显然，像这样的错误消息并不是特别有用。为了解决问题，需要提供一个机制对错误进行恰当的处理，例如向用户报告一条更有意义的错误消息。

这个过程称为**捕捉异常**[①]。代码清单 5.27 展示了具体的语法，输出 5.12 展示了结果。

代码清单 5.27　捕捉异常

```
public class ExceptionHandling
{
    public static int Main(string[] args)
    {
        string? firstName;
        string ageText;
        int age;
        int result = 0;

        Console.WriteLine("嘿，你！");

        Console.Write("请输入你的名字：");
        firstName = Console.ReadLine();
        Console.Write("请输入你的年龄：");
        // 假设不为空
        ageText = Console.ReadLine()!;

        try
        {
            age = int.Parse(ageText);
            Console.WriteLine(
```

[①] 译注：如果你喜欢说“捕获异常”，那么也是完全可以的。

```
                $"你好，{firstName}！你有{age * 12}个月大了。");
            }
            catch (FormatException)
            {
                Console.WriteLine(
                    $"你输入的年龄'{ageText}'不是一个有效的整数。");
                result = 1;
            }
            catch (Exception exception)
            {
                Console.WriteLine(
                    $"非预期的错误：{exception.Message}");
                result = 1;
            }
            finally
            {
                Console.WriteLine($"再见，{firstName}。");
            }

            return result;
        }
    }
```

输出 5.12

```
嘿，你！
请输入你的名字：Inigo
请输入你的年龄：四十二
你输入的年龄'四十二'不是一个有效的整数。
再见，Inigo。
```

首先用 try 块将可能抛出异常的代码（age = int.Parse()）包围起来。这个块以 try 关键字开始。try 关键字告诉编译器：开发者认为块中的代码有可能抛出异常；如果真的抛出了异常，那么某个 catch 块要尝试处理这个异常。

try 块之后必须紧跟着一个或多个 catch 块（可选择增加一个 finally 块）。catch 块（参见稍后的“高级主题：常规 catch”）可以指定异常的数据类型。只要数据类型与异常类型匹配，对应的 catch 块就会执行。但是，如果一直找不到合适的 catch 块，抛出的异常就会变成一个未处理异常，就好像没有进行异常处理一样。图 5.1 展示了最终的程序流程。

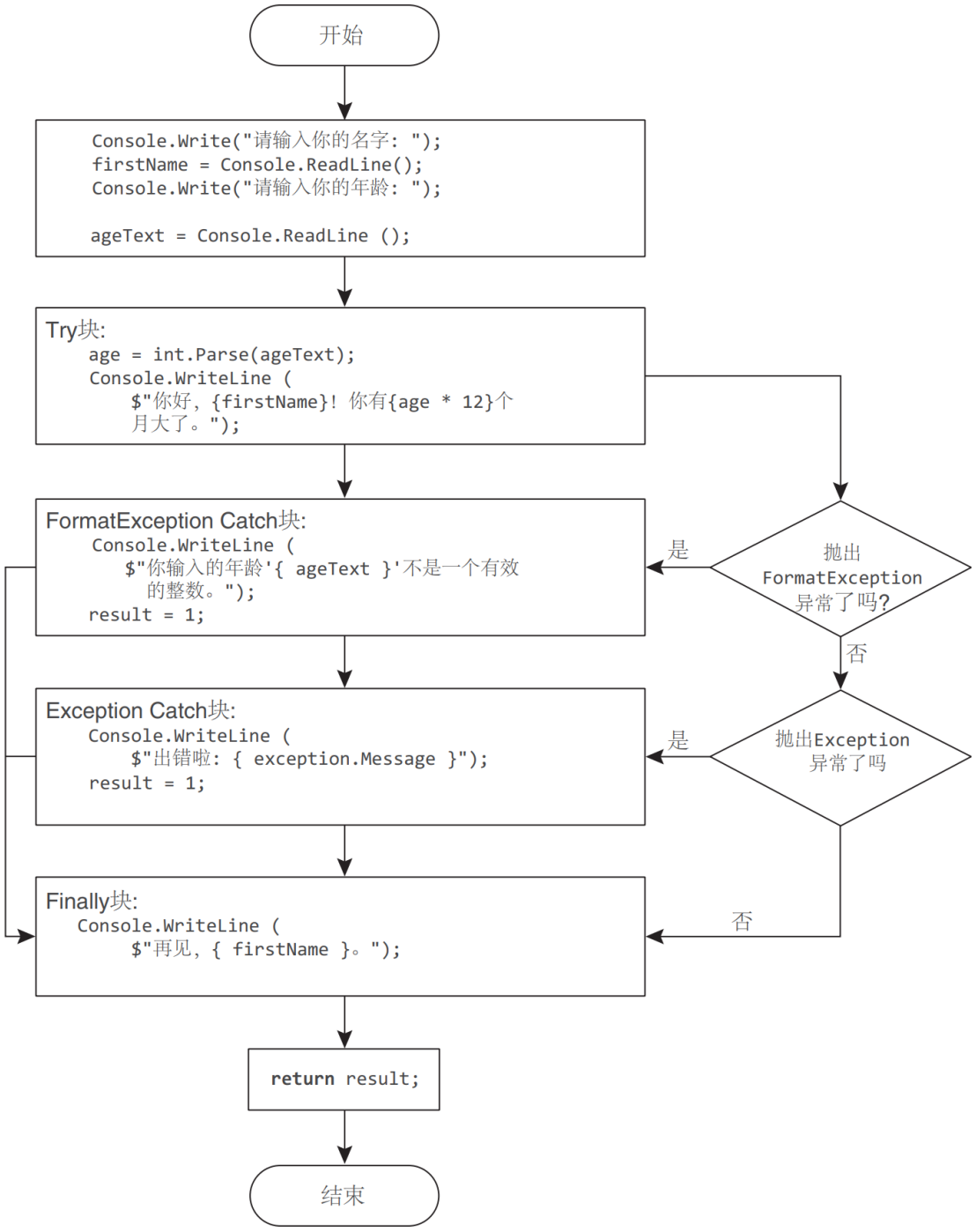


图 5.1 异常处理控制流

例如，假定为年龄输入 “四十二”，那么 `int.Parse()`会抛出 `System.FormatException` 类

型的异常，控制会跳转到后面的一系列 catch 块（FormatException 表明字符串格式不正确，无法进行解析）。由于第一个 catch 块就与 int.Parse()抛出的异常类型匹配，所以会执行这个块中的代码。但是，假如 try 块中的语句抛出的是不同类型的异常，执行的就是第二个 catch 块，因为所有异常都是 Exception 类型。

如果没有 FormatException catch 块，那么即使 int.Parse 抛出的是一个 FormatException 异常，也会执行 Exception catch 块。这是由于 FormatException 也属于（is-a）Exception 类型（FormatException 是泛化异常类 Exception 的一个更具体的实现）。

虽然 catch 块的数量随意，但处理异常的顺序不要随意。catch 块必须从最具体到最不具体排列。Exception 数据类型最不具体，所以应该放到最后。FormatException 排在第一，因为它是代码清单 5.27 所处理的最具体的异常。

不管 try 块的代码是否抛出异常，只要控制离开 try 块，最终都会执行 finally 块。finally 块的作用是提供一个最终位置，在其中放入无论是否发生异常都要执行的代码。finally 块最适合用来进行资源清理（dispose）。事实上，完全可以只写一个 try 块和一个 finally 块，而不写任何 catch 块。无论 try 块是否抛出异常，甚至无论是否写了一个 catch 块来处理异常，finally 块都会执行。代码清单 5.28 演示了一个 try/finally 块，输出 5.13 展示了结果。

代码清单 5.28　有 finally 块但无 catch 块

```
public class ExceptionHandling
{
    public static int Main()
    {
        string? firstName;
        string ageText;
        int age;
        int result = 0;

        Console.Write("请输入你的名字：");
        firstName = Console.ReadLine();

        Console.Write("请输入你的年龄：");
        // 假设不为空        //
        ageText = Console.ReadLine()!;

        try
        {
            age = int.Parse(ageText);
            Console.WriteLine(
                $"你好，{firstName}！你有{age * 12}个月大了。");
        }
```

```
            finally
            {
                Console.WriteLine($"再见，{firstName}。");
            }

            return result;
        }
    }
```

输出 5.13

```
请输入你的名字：Inigo
请输入你的年龄：四十二
Unhandled exception. System.FormatException: The input string '四十二' was not in a correct
format.
   at System.Number.ThrowFormatException[TChar](ReadOnlySpan`1 value)
   at System.Int32.Parse(String s)
   at ...ExceptionHandling.Main() in ...
再见，Inigo。
```

细心的读者能看出蹊跷。“运行时”是先报告未处理的异常，再运行 `finally` 块。这种行为有何道理可言？

首先，该行为完全合法，因为对于未处理的异常，“运行时”的行为是它自己的实现细节，任何行为都合法！“运行时”之所以选择这个特定的行为，是因为它知道在运行 `finally` 块之前，异常就已经是未处理的了。“运行时”已经检查了调用栈上的所有栈帧，发现没有任何一个关联了能和抛出的异常匹配的 `catch` 块。

一旦“运行时”发现未处理的异常，就会检查是否在机器上安装了调试器，因为用户可能是软件开发人员，正要对这种错误进行分析。如果是，就允许用户在运行 `finally` 块*之前*将调试器与进程连接。没有安装调试器，或者用户主动选择拒绝调试，默认行为就是在控制台上打印未处理的异常，再看是否有任何 `finally` 块可供运行。注意，由于这是“实现细节”，所以“运行时”并非一定要运行 `finally` 块，它完全可以选择做其他的。

> **注意：**如果在进程退出之前发生未处理的异常，`finally` 块的执行顺序，以及它是否执行，是由“运行时”的实现定义的。

设计规范

AVOID explicitly throwing exceptions from `finally` blocks. (Implicitly thrown exceptions resulting from method calls are acceptable.)

避免从 finally 块显式抛出异常（因方法调用而隐式抛出的异常除外）。

DO favor try/finally and avoid using try/catch for cleanup code.

要优先使用 try/finally 而不是 try/catch 块来实现资源清理代码。[①]

DO throw exceptions that describe what exceptional circumstance occurred, and if possible, how to prevent it.

要在抛出的异常中描述异常为什么发生。顺带说明如何防范就更好了。

> **高级主题：Exception 类继承**
>
> 从 C# 2.0 起，所有异常都派生自 System.Exception 类（从其他语言抛出的异常类型如果不是从 System.Exception 派生，会自动由一个是的对象“封装”）。所以，它们都可以用 catch(System.Exception exception)块进行处理（本章的例子因为使用了 5.5 节描述的顶级语句，所以可以省略 System 前缀）。但是，更好的做法是写专门的 catch 块来处理更具体的派生类型（例如 FormatException），从而获取有关异常的具体信息，有的放矢地进行处理，避免使用大量条件逻辑来判断具体发生了什么类型的异常。
>
> 这正是 C#规定 catch 块必须从“最具体”到“最不具体”排列的原因。例如，用于捕捉 Exception 的 catch 语句不能出现在捕捉 FormatException 的 catch 语句之前，因为 FormatException 较 Exception 具体。否则，永远执行不到 FormatException catch 块。

一个方法可以抛出许多异常类型，表 5.2 总结了一些较常见的。

[①] 译注：作者的意思是说，要优先将资源清理代码放在 finally 块中，而不是放在其他地方。

表 5.2 常见异常类型

异常类型(省略了 System 前缀)	**描述**
Exception	所有异常的基类，是最“基本”的异常；其他所有异常类型都从它派生
ArgumentException	传给方法的参数无效
ArgumentNullException	不应该为 null 的参数为 null
ApplicationException	避免用这个。最开始是想区分系统异常和应用程序异常。貌似合理，实际不好使
FormatException	实参类型不符合形参规范
IndexOutOfRangeException	试图访问不存在的数组或其他集合元素
InvalidCastException	无效的类型转换
InvalidOperationException	出现非预期的情况，应用程序不再处于有效工作状态
NotImplementedException	虽然找到了对应的方法签名，但该方法尚未完全实现，经常用它在写代码的时候创建 to-do 事项
NullReferenceException	引用为空，没有指向一个实例
ArithmeticException	发生无效数学运算，其中不包括被零除（因为 C#搞了一个 NaN）
ArrayTypeMismatchException	试图将类型有误的元素存储到数组中
StackOverflowException	发生非预期的深递归，栈溢出了

> **高级主题：常规 catch**
>
> 我们可以指定一个无参的 catch 块，如代码清单 5.29 所示。

代码清单 5.29　常规 catch 块

```
// 上一个 catch 子句已捕获所有异常
#pragma warning disable CS1058
// ...
        try
        {
            age = int.Parse(ageText);
            Console.WriteLine(
                $"你好，{firstName}！你有{age * 12}个月大了。");
        }
        catch(FormatException exception)
        {
            Console.WriteLine(
                $"你输入的年龄'{ageText}'不是一个有效的整数。");
            result = 1;
        }
        catch(Exception exception)
        {
            Console.WriteLine(
                $"非预期的错误：{exception.Message}");
            result = 1;
        }
        catch
        {
            Console.WriteLine("非预期的错误！");
            result = 1;
        }
        finally
        {
            Console.WriteLine($"再见，{firstName}。");
        }
```

没有指定数据类型的 catch 块称为**常规 catch 块**，等价于获取 object 数据类型的 catch 块，即 catch(object exception){...}。它的作用与上一个 Exception catch 块完全相同，所以正常情况下会显示一个警告：“上一个 catch 子句已捕获所有异常”。为了屏蔽这个警告，我们在代码中添加了#pragma warning disable 指令。

由于所有类最终都从 object 派生，所以没有数据类型（无参）的 catch 块必须放到最后。

常规 catch 块很少使用，因为没办法捕捉有关异常的任何信息。此外，C#也不支持抛出一个 object 类型的异常，只有使用 C++这样的语言写的库才允许任意类型的异常。

从 C# 2.0 起的行为稍微有别于之前的版本：如果遇到用另一种语言写的代码，而且它会抛出不是从 Exception 类派生的异常，那么该异常对象会被封装到一个 System.Runtime.CompilerServices.RuntimeWrappedException 中，后者从 Exception

> 派生。换言之，在C#程序集中，所有异常（无论它们是否从 Exception 派生）都会表现得像是从 Exception 派生的。
>
> 结果就是，捕捉 Exception 的 catch 块会捕捉之前的块没有捕捉到的所有异常，同时，Exception catch 块之后的一个常规 catch 块永远得不到调用。所以，从 C# 2.0 开始，假如在捕捉 Exception 的 catch 块之后添加了一个常规 catch 块，编译器就会报告一条警告消息，指出常规 catch 块永远不会执行。

设计规范

AVOID general catch blocks and replace them with a catch of Exception.

避免使用常规（无参）catch 块，用捕捉 Exception 的 catch 块代替。

AVOID catching exceptions for which the appropriate action is unknown. It is better to let an exception go unhandled than to handle it incorrectly.

避免捕捉无法从中完全恢复的异常。这种异常与其不正确地处理，还不如保持未处理状态。

5.10.2 使用 throw 语句报告错误

C#允许开发人员从代码中抛出异常，代码清单 5.30 和输出 5.14 对此进行了演示。

代码清单 5. 30　抛出异常

```
public static void Main()
{
    try
    {
        Console.WriteLine("开始执行");
        Console.WriteLine("抛出异常");
        throw new Exception("任意异常");
        Console.WriteLine("结束执行");
    }
    catch(FormatException exception)
    {
        Console.WriteLine(
            "已抛出一个 FormatException 异常");
```

```
        }
        // Catch 1
        catch(Exception exception)
        {
            Console.WriteLine(
                $"非预期的错误: { exception.Message }");
            // 跳转到 Post Catch
        }

        // Post Catch
        Console.WriteLine(
            "正在关闭...");
    }
```

输出 5.14

```
开始执行
抛出异常
非预期的错误: 任意异常
正在关闭...
```

如代码清单 5.30 的注释所示，抛出异常会使执行从异常的抛出点跳转到与抛出的异常类型兼容的第一个 catch 块。本例是由第二个 catch 块（Catch 1）处理抛出的异常，它在屏幕上输出一条错误消息。由于没有 finally 块，所以在执行了 Catch 1 的 Console.WriteLine() 方法后，会执行 try/catch 块后的 Console.WriteLine()语句（Post Catch）。

抛出异常需要有 Exception 的实例。代码清单 5.30 使用关键字 new，后跟异常的数据类型，从而创建了这样的实例。大多数异常类型都允许在抛出该类型的异常时传递消息，以便在发生异常时获取消息。

有的时候，catch 块能捕捉异常，但不能正确或完整地处理。这时可以让该 catch 块重新抛出异常，具体是使用一个独立 throw 语句，不要在它后面指定任何异常，如代码清单 5.31 所示。

代码清单 5.31　重新抛出异常

```
// ...
catch (Exception exception)
{
    Console.WriteLine(
        "重新抛出非预期的异常: "
        + $"{ exception.Message }");

    throw;
}
```

```
    // ...
```

注意，代码清单 5.31 中的 `throw` 语句是“空”的，没有指定 `exception` 变量所引用的异常。区别在于，`throw;`保留了异常中的“调用栈”信息，而 `throw exception;`将那些信息替换成当前调用栈信息。而调试时一般都需要知道原始调用栈。

设计规范

DO prefer using an empty `throw` when catching and rethrowing an exception so as to preserve the call stack.

要在捕捉并重新抛出异常时使用空的 `throw` 语句，以便保留调用栈。

DO report execution failures by throwing exceptions rather than returning error codes.

要通过抛出异常而不是返回错误码来报告执行失败。

DO NOT have public members that return exceptions as return values or an `out` parameter. Throw exceptions to indicate errors; do not use them as return values to indicate errors.

不要让公共成员将异常作为返回值或者 `out` 参数。抛出异常来指明错误；不要把它们作为返回值来指明错误。

AVOID catching and logging an exception before rethrowing. Instead, allow the exception to escape until it can be handled appropriately.

避免在重新抛出前捕捉和记录异常。要允许异常逃脱（传播），直至它被正确处理。

报告空参数异常

在启用了可空引用类型的情况下，编译器会努力识别有可能将可空参数作为非空参数传递的情况。如果有可能将 `null` 值赋给非空参数，那么编译器会发出警告，指出：“将 `null` 文本或可能的 `null` 值转换为不可为 `null` 类型。”只要没禁用这样的警告，那么可以放心地依赖编译器来捕获大多数可能的情况。但是，如果调用者在你的控制范围之外（例如，来

自你没有参与编写的库），没有启用可空引用类型，主动忽略了警告，或者从 C# 7.0 或更早的版本调用方法，那么尽管参数声明为非空，仍然无法阻止传递 null 参数。因此，最佳实践是检查任何公共的非空引用类型，确保它们不为 null。在.NET 6.0 中，可以使用 if (is null)来做这个检查。

```
if (is null) throw new ArgumentNullException(...)
```

利用空合并操作符（4.6.2 节），在单个语句中就能完成赋值或抛出异常的整套操作。如果参数值为 null，那么可以使用 throw ArgumentNullException 表达式，如代码清单 5.32 所示。

代码清单 5.32　通过抛出 ArgumentNullException 进行参数验证

```
httpsUrl = httpsUrl ??
    throw new ArgumentNullException(nameof(httpsUrl));
fileName = fileName??
    throw new ArgumentNullException(nameof(fileName));

// ...
```

在.NET 7.0 中，可以使用 ArgumentNullException.ThrowIfNull()方法，如代码清单 5.33 所示。

代码清单 5.33　使用 ArgumentNullException.ThrowIfNull()进行参数验证

```
ArgumentNullException.ThrowIfNull(httpsUrl);
ArgumentNullException.ThrowIfNull(fileName);

// ...
```

ArgumentNullException.ThrowIfNull()方法内部抛出的还是 ArgumentNullException。

设计规范

DO verify that non-null reference types parameters are not null and throw an ArgumentNullException when they are.

要验证非空引用类型的参数不为 null，并在其为 null 时抛出 ArgumentNullException。

DO use `ArgumentException.ThrowIfNull()` to verify values are null in .NET 7.0 or later.

要在.NET 7.0 或更高版本中使用 `ArgumentException.ThrowIfNull()`验证值是否为 `null`。

其他参数验证技术

显然，方法参数可能还有其他许多形式的约束。例如，可能要求一个字符串参数不能为空串，不能只由空白字符（例如，制表符）组成，或者必须具有“HTTPS”作为前缀。代码清单 5.34 用一个完整的 `DownloadSSL()`方法来演示了这些参数验证技术。

代码清单 5. 34　自定义参数验证

```
public class Program
{
    public static int Main(string[] args)
    {
        int result = 0;
        if (args.Length != 2)
        {
            // 必须指定两个（而且只能是两个）参数；报错
            Console.WriteLine(
                "错误：必须指定"
                + "URL 和文件名");
            Console.WriteLine(
                "用法：Downloader.exe <URL> <文件名>");
            result = 1;
        }
        else
        {
            DownloadSSL(args[0], args[1]);
        }
        return result;
    }

    private static void DownloadSSL(string httpsUrl, string fileName)
    {
#if !NET7_0_OR_GREATER
    httpsUrl = httpsUrl?.Trim() ??
        throw new ArgumentNullException(nameof(httpsUrl));
    fileName = fileName ??
        throw new ArgumentNullException(nameof(fileName));
    if (fileName.Trim().Length == 0)
    {
        throw new ArgumentException(
            $"{nameof(fileName)}不能为空或者仅由空白字符构成");
```

```
    }
#else
        ArgumentException.ThrowIfNullOrEmpty(httpsUrl = httpsUrl?.Trim()!);
        ArgumentException.ThrowIfNullOrEmpty(fileName = fileName?.Trim()!);
#endif
        if (!httpsUrl.ToUpper().StartsWith("HTTPS"))
        {
            throw new ArgumentException("URL 必须以'HTTPS'开头。");
        }

        HttpClient client = new();
        byte[] response =
            client.GetByteArrayAsync(httpsUrl).Result;
        client.Dispose();
        File.WriteAllBytes(fileName!, response);
        Console.WriteLine($"已从'{httpsUrl}'下载'{fileName}'。");
    }
}
```

使用.NET 7.0 或更高版本，可以依赖 ArgumentException.ThrowIfNullOrEmpty()方法来同时检查 null 和空白（empty）字符串（即""）。另外，如果在调用 ThrowIfNullOrEmpty()时配合使用 string.Trim()方法，那么在参数内容仅由空白字符串构成时，也可以抛出异常。（不过，系统默认的异常消息不会指出输入的是空白字符。）.NET 6.0 或更早版本的等效代码显示在#if 指令中。

如果 null、空字符串和空白字符均验证通过，代码清单 5.34 就用一个 if 语句来检查“HTTPS”前缀。如果验证失败，生成的代码将抛出 ArgumentException，其中包含描述问题的自定义消息。

nameof 操作符简介

参数未通过验证时应抛出异常，通常是 ArgumentException 或 ArgumentNullException 类型的异常。这两种异常都接受一个名为 paramName 的 string 参数，用于标识无效参数的名称。在代码清单 5.32 中，我们使用自 C# 6.0 引入的 nameof 操作符作为该参数。nameof 操作符接收一个标识符，例如 httpsUrl 变量，并返回该名称的字符串表示，本例即为"httpsUrl"。

使用 nameof 操作符的优势在于，如果标识符名称发生更改，重构工具自动更改传给 nameof 的实参。如果没有使用重构工具，代码将无法编译，迫使开发人员手动更改实参。这是一个很好的设计，因为先前的值可能无效。结果是 nameof 甚至会检查拼写错误。由此得出的设计规范是：向 ArgumentException 和 ArgumentNullException 等异常传递 paramName 实参时，要使用 nameof 操作符。更多信息请参见第 18 章。

设计规范

DO use `nameof(value)` (which resolves to `"value"`) for the `paramName` argument when creating `ArgumentException()` or `ArgumentNullException()` type exceptions. (`value` is the implicit name of the parameter on property setters.)

创建 `ArgumentException` 或 `ArgumentNullException` 类型的异常时，**要**使用 `nameof(value)`（解析为 `"value"`）作为 `paramName` 实参。（`value` 是属性 setter 的隐式参数名称。）

避免使用异常处理来处理预料之中的情况

开发人员应避免为预料之中的情况或正常控制流抛出异常。例如，开发人员应事先料到用户可能在输入年龄时输入无效文本[①]。因此，不要用异常来验证用户输入的数据。相反，应该在尝试转换前对数据进行检查（甚至可以考虑从一开始就防止用户输入无效数据）。异常是专为跟踪例外的、事先没有预料到的、而且可能造成严重后果的情况而设计的。为本来就预料到的情况使用异常，会造成代码难以阅读、理解和维护。

以第 2 章使用的 `int.Parse()`方法为例，它将字符串转换为整数。在那一章的例子中，即使用户输入的不是数字，代码也会执行转换。而“用户输入的不是数字”是一种我们应该事先预料到的情况。`Parse()`方法的问题在于，为了确定转换是否成功，唯一的办法就是强制转换，然后在不起作用时捕捉异常。由于抛出异常是一种相对昂贵的操作，所以最好是在不进行异常处理的前提下尝试转换。为此，最好的办法是使用某个 `TryParse()`方法，比如 `int.TryParse()`。它需要使用 out 关键字，因为 `TryParse()`返回的是一个 `bool`，而不是转换后的值。代码清单 5.35 演示了如何使用 `int.TryParse()`进行转换。

代码清单 5.35　用 `int.TryParse()`来转换

```
if (int.TryParse(ageText, out int age))
{
    Console.WriteLine(
        $"你好，{ firstName }！" +
        $"你有{age * 12}个月大了。");
}
else
```

---

① 通常，开发者必须假定用户会采取非预期的行为，所以应该进行防御性编程，提前为所有想得到的“愚蠢用户行为”制订对策。

```
{
    Console.WriteLine(
        $"你输入的年龄'{ageText}'不是一个有效的整数。");
}
```

有了 TryParse()方法，处理从字符串向数值的转换就不必兴师动众地使用 try/catch 块了。

之所以要避免为事先能预料到的情况使用异常，另一个原因是性能。和大多数语言一样，C#在抛出异常时会产生些许性能损失——相较于大多数操作都是纳秒级的速度，它可能造成毫秒级的延迟。人们平常注意不到这个延迟——除非异常没有得到处理。例如，执行代码清单 5.26 的程序，并输入一个无效年龄（例如，“四十二"），由于异常没有得到处理，所以当“运行时”在环境中搜索有一个可以加载的调试器时，你会感觉到明显延迟。幸好，程序都已经在关闭了，性能好坏也无谓了。

设计规范

DO NOT use exceptions for handling normal, expected conditions; use them for exceptional, unexpected conditions.

不要用异常处理正常的、预期的情况；用它们处理异常的、非预期的情况。

5.11 小结

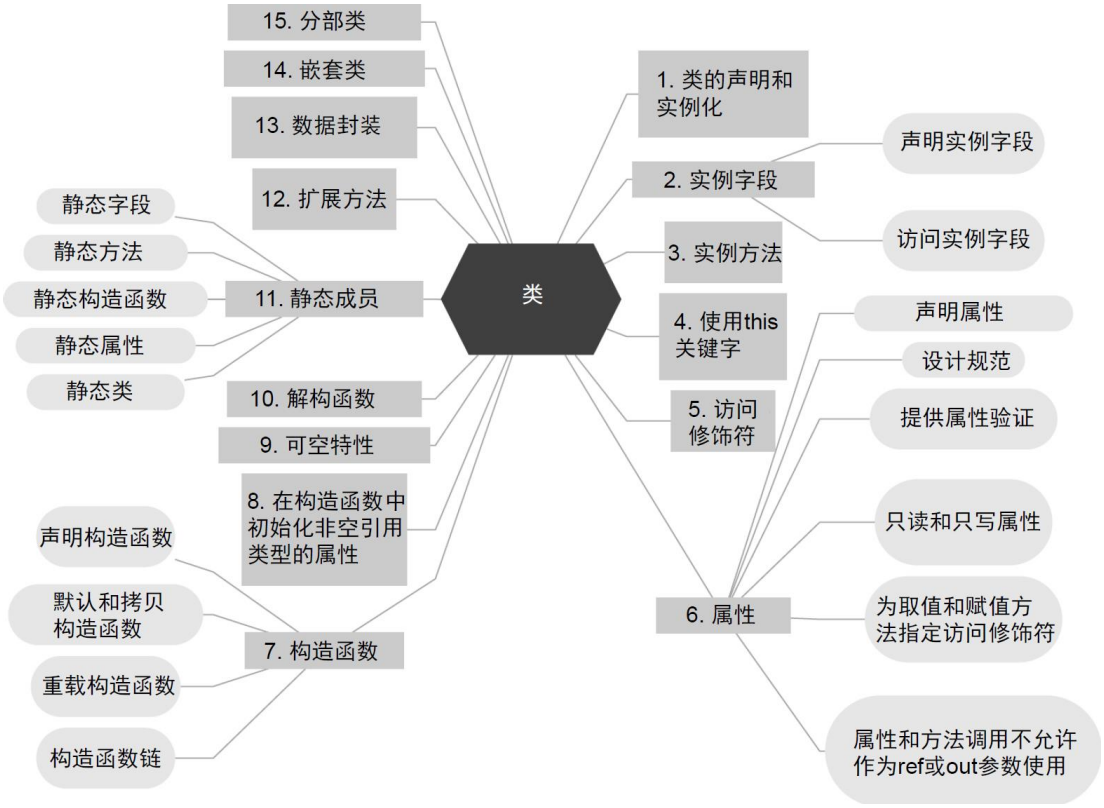
本章详细解释了方法的声明和调用，包括如何为方法参数使用关键字 out 和 ref 来传递/返回变量，而不是用 return 语句从方法中返回。除了方法声明，本章还介绍了基本的异常处理机制。

为了写出容易理解的代码，要利用“方法”这种基本编程单元。但不要在一个方法中包含大量语句，而应当学会用方法为代码“分段”，一个方法通常不应超过 10 行代码。将较大的任务分解成多个较小的子任务来重构代码，使代码更容易理解和维护。

下一章将讨论类，解释它如何将方法（行为）和字段（数据）封装为一个整体。

第 6 章 类

第 1 章简单介绍了如何声明一个名为 HelloWorld 的新类。第 2 章介绍了 C#内置的基元类型。在学习了控制流以及如何声明方法之后，接着就可以学习如何定义自己的类型了。这是任何 C#程序的核心构造。正是由于 C#支持类以及基于类来创建对象，所以我们说 C#是一种面向对象的编程语言。



本章介绍 C#面向对象编程的基础知识，重点放在**类**的定义上。你可以将类理解成对象的**模板**。

在面向对象编程中，之前学过的所有结构化的、基于控制流的编程构造仍然适用。但是，将那些构造封装到类中，可以创建出更大、更有条理而且更容易维护的程序。

面向对象编程的一个核心优势是不必从头创建新程序，而是可以将现有的一系列对象组装到一起，用新功能扩展类，或者添加更多的类。

还不熟悉面向对象编程的读者应阅读“初学者主题”获得对它的初步了解。“初学者主题”以外的内容将着重讨论如何使用 C#进行面向对象编程，并假定读者已熟悉了面向对象的概念。

本章重点在于 C#如何通过类、属性、访问修饰符、方法等构造来支持封装。着重讨论的是前三种，因为方法已在第 5 章讨论。掌握这些基础知识之后，第 7 章将讨论如何通过面向对象编程实现继承和多态性。

> **初学者主题：面向对象编程（OOP）**
>
> 如今，为了成功地进行编程，关键在于提供恰当的组织和结构，以满足大型应用程序的复杂需求。**面向对象编程**（Object-Oriented Programming，OOP）能很好地实现该目标。有多好呢？可以这样说，开发人员一旦熟悉了面向对象编程，除非写一些极为简单程序，否则很难再回到结构化编程了。
>
> 面向对象编程最基本的构造是**类**（class）。类相当于一种编程抽象、模型或模板，通常对应现实世界的一个概念。例如，`OpticalStorageMedia`（光学存储媒体）类可能有一个 `Eject()`方法，用于从播放机弹出光盘。在这个例子中，`OpticalStorageMedia` 类是对现实世界中的“CD/DVD 播放机”对象的一种编程抽象。
>
> 类呈现出了面向对象编程的三个主要特征——封装、继承和多态性。
>
> **封装**
>
> **封装**（encapsulation）旨在隐藏细节。必要时仍可访问细节，但通过巧妙地封装细节，大的程序变得更容易理解，数据不会因为不慎而被修改，代码也变得更容易维护（因为对一处代码进行修改所造成的影响被限制在封装的范围之内）。方法就是封装的一个例子。虽然可以将代码从方法中拿出，把它们直接嵌入调用者的代码中，但将特定的代码重构成方法，能享受到封装所带来的好处。
>
> **继承**
>
> 来考虑这个例子：DVD 是光学存储媒体的一种类型。它具有特定的存储容量，能容纳一部数字电影。CD 也是光学存储媒体的一个类型，但它具有不同特征。CD 上的版权保护有别于 DVD 的版权保护，两者存储容量也不同。无论 CD 还是 DVD，它们都有别于硬盘、U 盘和软盘（现在还有人知道这个东西吗？）。虽然所有这些都是“存储媒体”，但分别具有不同的特征——即使一些基本功能也是不同的，比如所支持的文件系统，以及媒体的实例是只读还是可读/可写。
>
> 面向对象编程中的继承允许在这些相似但又不同的物件之间建立“属于”（is a）关系。我们可以合理地认为，DVD 和 CD 都“属于”存储媒体。因而，它们都具有存储能力。类似地，CD 和 DVD 都“属于”光学存储媒体，后者又“属于”存储媒体。
>
> 为上面提到的每种存储媒体类型都定义一个类，就可以得到一个类层次结构，它由一系列“属于”关系构成。例如，可以将基类型（所有存储媒体都从它派生）定义成

StorageMedia（存储媒体）。CD、DVD、硬盘、U 盘和软盘都属于 StorageMedia。但 CD 和 DVD 不一定要直接从 StorageMedia 派生。相反，它们可以从中间类型 OpticalStorageMedia（光学存储媒体）派生。我们可以用一张 UML（Unified Modeling Language，统一建模语言）风格的类关系图来查看类层次结构，如图 6.1 所示。

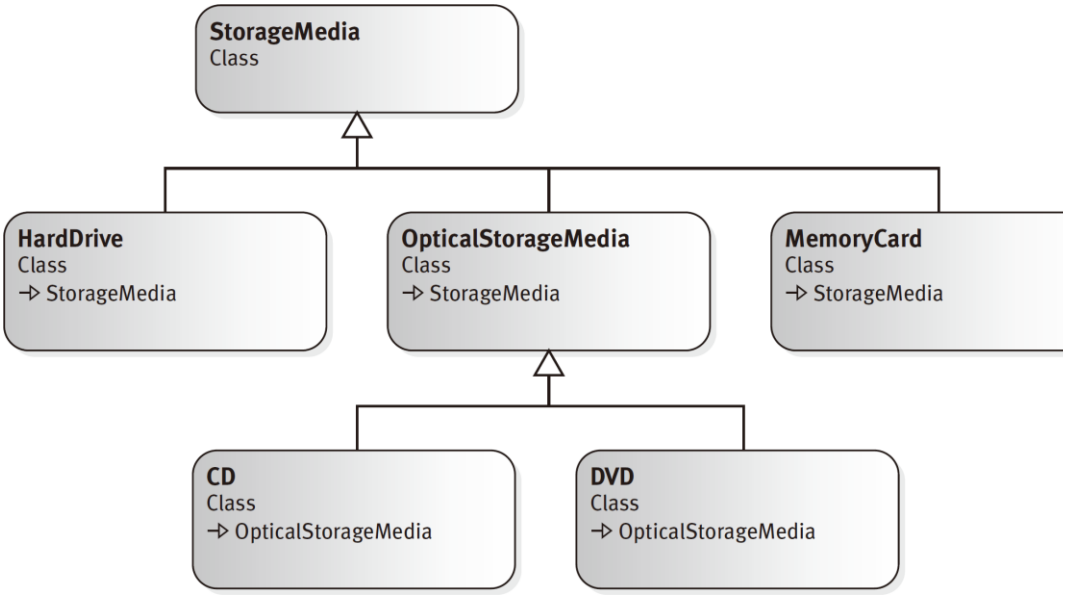


图 6.1 类层次结构

继承关系至少涉及两个类，一个是另一个更具体的版本。图 6.1 中的 HardDrive 是更具体的 StorageMedia。反之不成立，因为 StorageMedia 的一个实例并非肯定是 HardDrive。如图 6.1 所示，继承涉及的类可能不止两个。

更具体的类型称为**派生类型**或**子类型**。更常规的类型称为**基类型**或者**超类型**。也经常将基类型称为“父”类型，将派生类型称为它的“子”类型。虽然这种说法很常见，但会带来混淆。“子”毕竟不是“父”的一种！本书将采用“派生类型”和“基类型”的说法。

为了从一个类型**派生**或**继承**，需对那个类型进行**特化**（specialize）。这意味着要对基类型进行自定义，为满足特定需求而调整它。基类型可能包含所有派生类型都适用的实现细节。

继承最关键的一点是所有派生类型都继承了基类型的成员。可以在派生类型中修改基类型的成员，但无论如何，派生类型除了自己显式添加的成员，还包含了基类型的成员。

> 可用派生类型以一致性的层次结构组织类。在这个层次结构中，派生类型比它们的基类型更特别。
>
> **多态性**
>
> **多态性**（polymorphism）这个词由一个表示“多”（poly）的词和一个表示“态”（morph）的词构成。讲到对象时，多态性意味着一个方法或类型可具有多种形式的实现。
>
> 假定有一个媒体播放机，它既能播放音乐 CD，也能播放包含 MP3 歌曲的 DVD。但 `Play()`方法的具体实现会随着媒体类型的变化而变化。在一个音乐 CD 对象上调用 `Play()`方法，或者在一张音乐 DVD 上调用 `Play()`方法，都能播放出音乐，因为每种类型都理解自己具体如何“播放”。媒体播放机唯一知道的就是公共基类型 `OpticalStorageMedia`（光学存储媒介）以及它定义了 `Play()`方法签名的事实。多态性使不同类型能自己照料一个方法的实现细节，因为多个派生类型都包含了该方法，每个派生类型都共享同一个基类型（或接口），后者也包含了相同的方法签名。

6.1 类的声明和实例化

为了定义类，我们首先写关键字 `class`，再后跟一个标识符，如代码清单 6.1 所示。

代码清单 6.1　定义类

```
// 定义员工类
public class Employee
{
}
```

该类的所有代码都放到类声明之后的大括号中。虽然并非必须，但一般应该将每个类都放到它自己的文件中，并用类名对文件进行命名。这样可以更轻松地找到定义了一个特定类的代码。

设计规范

AVOID place more than one class in a single source file.

避免在一个源代码文件中放多个类。

DO name the source file with the name of the public type it contains.

源代码文件名**要**和它所包含的公共类型的名称一致。

定义好新类后，就可以像使用框架内置的类那样使用它了。换言之，现在可以声明该类型的变量，或者定义方法来接收该类型的参数。代码清单 6.2 对此进行了演示。

代码清单 6.2　声明类类型的变量

```
public class Program
{
    public static void Main()
    {
        Employee employee1, employee2;
        // ...
    }

    public static void IncreaseSalary(Employee employee)
    {
        // 加薪
    }
}
```

> **初学者主题：对象和类**
>
> 在非正式场合，类和对象这两个词经常互换着使用。但是，对象和类具有截然不同的含义。类是模板，定义了对象在实例化时看起来像什么样子。所以，对象是类的**实例**。类就像模具，定义了零件的样子。对象就是用这个模具创建的零件。从类创建对象的过程称为**实例化**，因为对象是类的实例。

现在，我们已经定义了一个新的**类类型**（class type），接着可以实例化该类型的对象。效仿它的前身语言，C#也使用 new 关键字实例化对象（参见代码清单 6.3）。

代码清单 6.3　类的实例化

```
public class Program
{
```

```
    public static void Main()
    {
        Employee employee1 = new Employee();
        Employee employee2;
        employee2 = new();

        IncreaseSalary(employee1);
        IncreaseSalary(employee2);
    }
}
```

毫不奇怪，声明和赋值既能在一个语句中完成，也能分别用不同的语句完成。

和本书以前使用的基元数据类型（如 int）不同，不能用字面值指定一个 Employee。相反，要用 new 操作符指示“运行时”为 Employee 对象分配内存、实例化对象，并返回对实例的引用。

虽然可以在 new 后面指定数据类型（Employee），但从 C# 9.0 开始，只要编译器能从赋值语句的左侧推断出类型，那么这个数据类型就可以省略。在本例中，编译器可以确定目标类型是 Employee，并因而推断 new 表达式是想要实例化一个 Employee ，所以允许在调用构造函数时不指定类型。C# 9.0 将这个功能称为“目标类型的 new 表达式”（target-typed new expressions）。

话虽如此，但假如无法从一行代码中清楚地看出目标类型，那么开发人员还是应该慎用这个功能。例如，突兀地写一行 text = new();语句，可能看不出具体的数据类型应该是什么。而且，即使能推断出是 string ，但万一是 System.Text.StringBuilder 类型呢？后者也很合理。因此，当数据类型不明显时，应避免使用目标类型的 new 表达式。

设计规范

AVOID target-typed new expressions when the data type of the constructor is not obvious.

避免当构造函数的数据类型不明显时使用“目标类型的 new 表达式”。

CONSIDER use target-typed new expressions when the data type of the constructor is obvious

考虑在构造函数的数据类型明显时使用“目标类型的 new 表达式”。

虽然有专门的 new 操作符分配内存，但没有一个对应的操作符回收内存。相反，“运行时”会在对象变得无人引用之后的某个时间自动回收内存。这个工作具体是由**垃圾回收器**完成的。它判断哪些对象不再由其他活动对象引用，然后自行安排一个时间回收对象占用的内

存。因此，我们无法在编译时确定会在程序的什么位置回收并归还内存。[1]

在当前这个简单的例子中，并没有什么数据或方法与 Employee 关联。像这样的对象其实是没有什么用处的。下一节重点讲述如何为对象添加数据。

> **初学者主题：封装（第一部分）：对象将数据和方法组合到一起**
>
> 假定接收到一叠写有员工名字的索引卡、一叠写有员工姓氏的索引卡以及一叠写有他们工资的索引卡，那么除非确定每一叠卡片都按相同顺序排列，否则这些索引卡没有什么作用。即使符合这个条件，也很难使用上面的数据，因为要判断一个人的全名，需要搜索两叠卡片。更糟的是，如果遗失其中一叠卡片，就没有办法再将名字、姓氏和工资关联起来了。在这种情况下，我们需要的是一叠员工卡片，每个员工的数据都组合到一张卡片中。换言之，要将名字、姓氏和工资**封装**到一起。
>
> 日常生活中的封装是将一系列物品装入封套。类似地，面向对象编程将方法和数据装入对象。这提供了所有类**成员**（类的数据和方法）的一个分组，使它们不再需要单独处理。不需要将名字、姓氏和工资作为三个单独的参数传给方法。相反，可以在调用时传递对一个员工对象的引用。一旦被调用的方法接收到对象引用，就可以向对象发送消息（例如在对象上调用像 AdjustSalary()——调薪——这样的方法），以执行特定的操作。

> **语言对比：C++——delete 操作符**
>
> 程序员应将 new 操作符视为实例化对象的调用，而不是分配内存的调用。在堆上分配的对象和在栈上分配的对象都支持 new 操作符，这进一步强调了 new 不是关于内存分配的，也不是关于是否有必要进行回收的。
>
> 所以，C#不需要 C++中的 delete 操作符。内存分配和回收是“运行时”的细节。这使开发人员可以将注意力更多地放在业务逻辑上。然而，虽然“运行时”会管理内存，但它不会管理其他资源，比如数据库连接、网络端口等。和 C++不同，C#不支持**隐式确定性资源清理**（implicit deterministic resource cleanup，也就是在编译时确定的位置进行隐式对象析构）。幸好，C#通过 using 语句来支持**显式确定性资源清理**

[1] 译注：《CLR via C#》第 4 版（清华大学出版社）对“运行时”的垃圾回收机制进行了很好的解释，详情参见它的第 21 章。

（explicit deterministic resource cleanup），通过**终结器**（finalizer）来支持**隐式非确定性资源清理**（implicit nondeterministic resource cleanup）。

6.2 实例字段

面向对象设计的一个核心是对数据进行分组以建立特定结构。本节将讨论如何在 Employee 类中添加数据。在 OOP 术语中，在类中存储数据的变量称为**成员变量**。这个术语在 C#中很好理解，但更标准、更符合规范的术语是**字段**。它是与包容类型[①]关联的具名存储单元。**实例字段**是在类的级别上声明的变量，用于存储与对象（实例）关联的数据。

6.2.1 声明实例字段

代码清单 6.4 对 Employee 进行了修改，在其中包含了三个字段：FirstName，LastName 和 Salary。

代码清单 6.4　声明字段

```
public class Employee
{
    public string FirstName; // 名字
    public string LastName;  // 姓氏
    public string? Salary;    // 工资
}
```

添加好字段后，就可以随同每个 Employee 实例存储一些基本数据。本例添加访问修饰符 public 作为字段前缀。为字段添加 public 前缀，意味着可以从 Employee 之外的其他类访问该字段中的数据（参见本章稍后的“访问修饰符”一节）。

和局部变量声明一样，在字段声明中包含了字段所引用的数据类型。此外，还可以在声明的同时初始化字段，如代码清单 6.5 的 Salary 字段所示。

代码清单 6.5　在声明的同时初始化字段

```
// 暂时禁用“不可为 null 的字段未初始化”警告，因为代码尚未完成
```

[①] 译注：包容类型（containing type）其实就是声明该字段的那个类型。

```
#pragma warning disable CS8618
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary = "不够";
}
```

字段命名和编码的设计规范将在稍后介绍了 C#“属性”和“构造函数”之后给出。现在只需知道，不仅代码清单 6.5，还有后续的几个代码清单，它们都*不*符合规范。事实上，你会频繁看到以下警告：

- CS0649：字段从未赋值，将始终具有默认值 null。
- CS8618：不可为 null 的字段未初始化。考虑声明为可以为 null。

在本例中，由于 FirstName 和 LastName 没有初始化，所以会触发 CS8618 警告。

为了避免分心，这些警告会在本书配套的源代码中忽略，并事实上通过#pragma 指令禁用，直到本章后面把所有这些概念讲清楚。

6.2.2 访问实例字段

可以设置和获取（检索）字段中的数据。注意，没有 static 修饰符的字段意味着它是实例字段。实例字段只能从其包容类的实例（对象）中访问，无法直接从类中访问（换言之，不创建实例就不能访问）。

代码清单 6.6 对 Program 类进行了更新，并展示了它如何使用 Employee 类。输出 6.1 展示了结果。

代码清单 6.6　访问字段

```
public class Program
{
    public static void Main()
    {
        Employee employee1 = new();
        Employee employee2;
        employee2 = new();

        employee1.FirstName = "Inigo";
        employee1.LastName = "Montoya";
        employee1.Salary = "太少了";
        IncreaseSalary(employee1);
```

```
        Console.WriteLine(
           $"{
            employee1.FirstName } {
            employee1.LastName }: {
            employee1.Salary }");
        // ...
    }

    public static void IncreaseSalary(Employee employee)
    {
        employee.Salary = "勉强过活";
    }
}
```

输出 6.1

```
Inigo Montoya: 勉强过活
```

代码清单 6.6 实例化两个 Employee 对象，这和之前的例子一样。接着设置每个字段，调用 IncreaseSalary()来更改工资，然后显示与 employee1 引用的对象关联的每个字段。

注意，首先必须指定要操作哪个 Employee 实例。所以，在对字段进行赋值和访问（取值）时，要添加 employee1 变量（Employee 类的一个实例）作为字段名的前缀。

6.3 实例方法

在 Main()中调用 WriteLine()方法并对姓名进行格式化，这其实是笨办法。更好的办法是在 Employee 类中提供方法专门进行格式化。将功能修改成由 Employee 提供，而不是作为 Program 的成员，这符合类的封装原则。为什么不把与员工姓名相关的方法放到包含姓名数据的类中呢？

代码清单 6.7 演示了如何创建这样的一个方法。

代码清单 6.7　从包容类内部访问字段

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;

    public string GetName()
    {
        return $"{ FirstName } { LastName }";
```

```
    }
}
```

和第 5 章的同名方法相比，这里的 GetName()没有太多特别之处，只是该方法现在访问对象中的字段，而非访问局部变量。此外，方法声明没有用 static 来标记。本章稍后会讲到，静态方法不能直接访问类的实例字段。相反，必须先获得类的实例才能调用实例成员——无论该实例成员是方法还是字段。

添加了 GetName()方法后，就可以更新 Program.Main()来使用它，如代码清单 6.8 和输出 6.2 所示。

代码清单 6.8　从包容类外部访问字段

```
public class Program
{
    public static void Main()
    {
        Employee employee1 = new();
        Employee employee2;
        employee2 = new();

        employee1.FirstName = "Inigo";
        employee1.LastName = "Montoya";
        employee1.Salary = "太少了";
        IncreaseSalary(employee1);
        Console.WriteLine(
            $"{employee1.GetName()}: {employee1.Salary}");
        // ...
    }
    // ...
}
```

输出 6.2

```
Inigo Montoya: 勉强过活
```

6.4 使用 this 关键字

在类的实例成员内部，我们可以获取对该类的引用。C#允许用关键字 this 显式指出当前访问的字段或方法是包容类的实例成员。调用任何实例成员时，this 都是隐含的，它返回对象本身的实例。来看看代码清单 6.9 中的 SetName()方法。

代码清单 6.9　使用 this 显式标识字段的所有者

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;
    public string GetName()
    {
        return $"{FirstName} {LastName}";
    }

    public void SetName(
        string newFirstName, string newLastName)
    {
        this.FirstName = newFirstName;
        this.LastName = newLastName;
    }
}
```

本例使用关键字 this 指出字段 FirstName 和 LastName 是类的实例成员。

虽然可以为所有本地类成员引用都添加 this 前缀，但设计规范是若非必要就不要在代码中“添乱”。所以， this 关键字只在必要时才应使用。本章后面的代码清单 6.12 是必须使用 this 的例子。代码清单 6.9 和 6.10 则不是。在代码清单 6.9 中，舍弃 this 不会改变代码的含义。而代码清单 6.10 可以修改字段命名规范，同时遵循参数的命名约定，从而避免局部变量与字段之间的歧义。

> **初学者主题：依靠编码样式避免歧义**
>
> 在代码清单 6.9 的 SetName()方法中，事实上没有必要使用 this 关键字，因为 FirstName 显然有别于 newFirstName。但是，假如参数不叫做 newFirstName，而叫做 FirstName（使用 PascalCase 风格的大小写规范），那么会发生什么情况？如代码清单 6.10 所示。

代码清单 6.10　使用 this 避免歧义

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;

    public string GetName()
    {
        return $"{ FirstName } { LastName }";
    }
```

```
        // 警告：参数名使用了 PascalCase 大小写风格，
        // 应改为 camelCase 大小写风格。
        public void SetName(string FirstName, string LastName)
        {
            this.FirstName = FirstName;
            this.LastName = LastName;
        }
    }
```

在这个例子中，要引用 FirstName 字段就必须显式指明它所在的 Employee 对象。this 就好比在 Program.Main()方法中使用的 employee1 变量前缀（参见代码清单 6.8），标识了要在其上调用 SetName()方法的那个对象。

代码清单 6.10 不符合 C#命名规范，即参数要像局部变量那样使用 camelCase 大小写风格（除了第一个单词，其他每个单词首字母大写）。违反这个规范，可能造成难以发现的 bug，因为将字段 FirstName 赋给参数 FirstName，代码仍能编译并运行。为了避免该问题，最好是为参数和局部变量采用和字段不同的命名规范。本章稍后会演示该规范的实际应用。

代码清单 6.9 和代码清单 6.10 中的 GetName()方法没有使用 this 关键字，它确实可有可无。但是，假如存在与字段同名的局部变量或参数（参见代码清单 6.10 的 SetName()方法），那么省略 this 将访问局部变量或参数而非字段。这时 this 就是必须的。

还可以使用 this 关键字显式访问类的方法。例如，可以在 SetName()方法内使用 this.GetName()输出新赋值的姓名（参见代码清单 6.11 和输出 6.3）。

代码清单 6.11　this 作为方法名前缀

```
public class Employee
{
    // ...
    public string GetName()
    {
        return $"{FirstName} {LastName}";
    }

    public void SetName(string newFirstName, string newLastName)
    {
        this.FirstName = newFirstName;
        this.LastName = newLastName;
        Console.WriteLine(
```

```
            $"姓名更改为'{this.GetName()}'");
    }
}

public class Program
{
    public static void Main()
    {
        Employee employee = new();
        employee.SetName("Inigo", "Montoya");
        // ...
    }
    // ...
}
```

输出 6.3

```
姓名更改为'Inigo Montoya'
```

有的时候，我们需要使用 this 传递对当前对象的引用。如代码清单 6.12 中的 Save()方法所示。

代码清单 6.12　在方法调用中传递 this

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;

    public void Save()
    {
        DataStorage.Store(this);
    }
}

public class DataStorage
{
    // 将 Employee 对象写入一个以员工姓名命名的文件
    public static void Store(Employee employee)
    {
        // ...
    }
}
```

Save()方法调用 DataStorage 类的 Store()方法。但是，需要向 Store()方法传递准备进行持久化存储的 Employee 对象。这是使用关键字 this 来完成的，它传递了正在其上调用 Save()方法的那个 Employee 对象实例。

存储和加载文件

在 DataStorage 内部，Store()方法的实现要用到 System.IO 命名空间中的类，如代码清单 6.13 所示。在 Store()内部，首先要实例化一个 FileStream 对象，将它与一个对应员工全名的文件关联。FileMode.Create 参数的意思是说，如果不存在名为<名字><姓氏>.dat 的文件，那么就新建一个；如果存在，就覆盖它。接着，创建一个 StreamWriter 对象，以便将文本写入 FileStream（相当于向文件写入）。数据用 WriteLine()方法写入，这跟向控制台写入没有太大区别。

代码清单 6.13　将数据持久化存储到文件

```
public class DataStorage
{
    // 将 Employee 对象写入一个以员工姓名命名的文件；
    // 这里未显示错误处理的情况。
    public static void Store(Employee employee)
    {
        // 使用<名字><姓氏>.dat 作为文件名来实例化一个 FileStream。
        // FileMode.Create 将强制创建一个新文件，或者覆盖一个已存在的文件。
        // 注意：这段代码可以通过使用 using 语句来改进——我们目前尚未讲到的一种构造。
        FileStream stream = new(
            employee.FirstName + employee.LastName + ".dat",
            FileMode.Create);

        // 创建 StreamWriter 类型的对象 writer，
        // 以便将文本写入 FileStream 类型的对象 stream。
        StreamWriter writer = new(stream);

        // 开始写入与员工实例关联的所有数据
        writer.WriteLine(employee.FirstName);
        writer.WriteLine(employee.LastName);
        writer.WriteLine(employee.Salary);

        // 对 StreamWriter 及其流进行资源清理(dispose)
        writer.Dispose();  // 会自动关闭流
    }
    // ...
}
```

写入完成后，应关闭 FileStream 和 StreamWriter[①]，避免它们在等待垃圾回收期间处于“不确定性打开”状态。上述代码未包含任何错误处理机制，所以如果中途抛出异常，那么可能会造成 Dispose()方法得不到调用的情况。[②]

文件加载过程与存储过程相似，如代码清单 6.14 和输出 6.4 所示。

代码清单 6.14　从文件获取数据

```
public class Employee
{
    // ...
}

public class DataStorage
{
    // ...

    public static Employee Load(string firstName, string lastName)
    {
        Employee employee = new();

        // 使用<名字><姓氏>.dat 作为文件名来实例化一个 FileStream。
        // FileMode.Open 将打开一个已存在的文件；不存在会报错。
        FileStream stream = new(
            firstName + lastName + ".dat", FileMode.Open);

        // 创建一个 StreamReader，以便从文件中读取文本
        StreamReader reader = new(stream);

        // 读取文件的每一行，并将其赋给关联的属性
        employee.FirstName = reader.ReadLine() ??
```

[①] 译注：在 StreamWriter 对象上调用 dispose()，会自动在 FileStream 对象上调用同样的方法，确保先将数据写入文件再关闭文件流。两者的依赖关系请参见《CLR via C#》的 21.3.2 节。

[②] 译注：文档将 disposal 和 dispose 翻译成“释放”。这里解释一下为什么不赞成这个翻译。在英语中，这个词的意思是“摆脱”或“除去”(get rid of)一个东西，尤其是在这个东西很难除去的情况下。之所以认为“释放”不恰当，除了和 release 一词冲突，还因为 dispose 强调了“清理”和“处置”，而且在完成(对象中包装的)资源的清理之后，对象占用的内存还暂时不会释放。所以，“dispose 一个对象”真正的意思是：清理或处置对象中包装的资源(比如它的字段引用的对象)，然后等着在一次垃圾回收之后回收该对象占用的托管堆内存(此时才释放)。为避免误解，本书尽量保留 dispose 和 disposal 的原文，或者说成“资源清理”。

```
                throw new InvalidOperationException(
                    "FirstName 不能为 null");
            employee.LastName = reader.ReadLine() ??
                throw new InvalidOperationException(
                    "LastName 不能为 null");
            employee.Salary = reader.ReadLine();

            // 对 StreamReader 及其流进行资源清理(dispose)
            reader.Dispose();  // 会自动关闭流

            return employee;
        }
    }

    public class Program
    {
        public static void Main()
        {
            Employee employee1;

            Employee employee2 = new();
            employee2.SetName("Inigo", "Montoya");
            employee2.Save();

            // 保存后修改 employee2
            IncreaseSalary(employee2);

            // 从保存的 employee2 版本中，将数据加载到 employee1
            employee1 = DataStorage.Load("Inigo", "Montoya");

            Console.WriteLine(
                $"{employee1.GetName()}: {employee1.Salary}");
        }
        // ...
    }
```

输出 6.4

```
姓名更改为'Inigo Montoya'
Inigo Montoya:
```

代码清单 6.14 展示了和存储相反的过程，使用 StreamReader 而非 StreamWriter。同样地，一旦数据读取完毕，就在 StreamReader 上调用 Dispose()方法进行资源清理。

输出 6.4 没有在 Inigo Montoya:之后显示任何工资信息，这是因为在本例中，是在调用了 Save()后才调用 IncreaseSalary()将 Salary 设为"勉强过活"。换言之，从磁盘文件没有

读取到加薪后的数据。[1]

注意，Main()是在一个员工实例上调用 Save()来保存新员工的数据。但是，为一个新员工加载数据时，我们调用的是 DataStorage.Load()。需要加载一个员工的数据时，通常还不存在可在其中加载（数据）的员工实例。因此，在这种情况下使用 Employee 类的某个实例方法并不可取。除了在 DataStorage 类上调用 Load()，另一个思路是直接为 Employee 类添加一个静态 Load()方法（详情参见本章后面的“静态成员”一节），这样就可以调用 Employee.Load()——注意，是直接在 Employee 类上调用，而不是在它的某个实例上调用。

6.5 访问修饰符

本章之前声明字段时，曾为字段声明添加关键字 public 作为前缀。public 是一种**访问修饰符**，它标识了所修饰成员的封装级别。C#支持 5 种访问修饰符：public，private，protected，internal 和 protected internal。本节介绍前两个。

> **初学者主题：封装（第二部分）：信息隐藏**
>
> 除了对数据和方法进行分组，封装的另一个重要作用是隐藏对象的数据以及行为的内部细节。方法在某种程度上也能做到这一点，因为在方法外部，调用者看见的只有方法的声明，内部实现则看不到。但是，面向对象编程更进一步，它能控制类成员在类外部的可视程度。类外部不可见的成员称为**私有成员**。
>
> 在面向对象编程中，封装的作用不仅仅是对数据和行为进行分组，它还能使类内部的工作机制不被暴露。这降低了调用者对数据进行不恰当修改的几率，同时防止类的使用者依赖类的内部实现来写自己程序（将来若实现发生了变化，那么程序也不得不跟着修改）。

访问修饰符的作用是提供封装。public 显式指定可以从 Employee 类的外部访问被它修饰的字段。例如，可以从 Program 类中访问那些字段。

但是，假定 Employee 类要包含一个代表密码的 Password 字段，那么应该如何设计？这时应允许在一个 Employee 对象上调用 Logon()方法来验证密码，但不应允许从类的外部访问 Employee 对象的 Password 字段。

---

[1] 译注：要在 employee1 中反映 employee2 加薪后的工资，在 IncreaseSalary(employee2);后紧接着调用一次 employee2.Save();即可。

为了隐藏 Password 字段，禁止从它的包容类的外部访问，应使用 private 访问修饰符代替 public，如代码清单 6.15 所示。这样就无法在 Program 类中访问 Password 字段了。

代码清单 6.15　使用 private 访问修饰符

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string? Salary;
    // 像这样直接使用解密的密码仅供演示，平时不推荐。
    // 未初始化；稍后会解释“构造函数”
    private string Password;
    private bool IsAuthenticated;

    public bool Logon(string password)
    {
        if (Password == password)
        {
            IsAuthenticated = true;
        }
        return IsAuthenticated;
    }

    public bool GetIsAuthenticated()
    {
        return IsAuthenticated;
    }
    // ...
}

public class Program
{
    public static void Main()
    {
        Employee employee = new();

        employee.FirstName = "Inigo";
        employee.LastName = "Montoya";

        // ...

        // 错误：Password 是私有的，所以不能从类的外部访问
        Console.WriteLine(
           "密码 = {0}", employee.Password);
    }
    // ...
```

```
}
```

虽然代码清单 6.15 没有演示使用 private 来修饰方法，但实际也是可以的。

注意，如果不为类成员添加访问修饰符，默认就是 private。也就是说，成员默认私有。想要公共的成员必须显式指定。

6.6 属性

上一节演示了如何使用 private 关键字封装密码，禁止从类的外部访问。但是，这种形式的封装通常过于严格。例如，可能希望字段在外部只读，但内部可以更改。又例如，可能希望允许对类中的一些数据执行写入操作，但需要验证对数据的更改。再例如，可能希望动态构造数据。为了满足所有这些需求，传统方式是将字段标记为私有，再提供取值和赋值方法（getter 和 setter）来访问和修改数据。代码清单 6.16 将 FirstName 和 LastName 都更改为私有字段。每个字段的公共取值和赋值方法则用于访问和更改它们的值。

代码清单 6.16　声明取值和赋值方法

```
public class Employee
{
    private string FirstName;
    // FirstName getter
    public string GetFirstName()
    {
        return FirstName;
    }
    // FirstName setter
    public void SetFirstName(string newFirstName)
    {
        if (newFirstName != null && newFirstName != "")
        {
            FirstName = newFirstName;
        }
    }
    private string LastName;
    // LastName getter
    public string GetLastName()
    {
        return LastName;
    }
    // LastName setter
    public void SetLastName(string newLastName)
    {
        if (newLastName != null && newLastName != "")
```

```
        {
            LastName = newLastName;
        }
    }
    // ...
}
```

遗憾的是，这一更改会影响 `Employee` 类的可编程性。无法再用赋值操作符来设置类中的数据。另外，只能调用方法来存取数据。

6.6.1 声明属性

考虑到经常都会用到这种编程模式，所以 C#的设计者决定为它提供显式的语法支持。这种语法称为**属性**（property），如代码清单 6.17 和输出 6.5 所示。

代码清单 6.17　定义属性

```
public class Program
{
    public static void Main()
    {
        Employee employee = new();
        // 调用 FirstName 属性的 setter(赋值方法)
        employee.FirstName = "Inigo";
        // 调用 FirstName 属性的 getter(取值方法)
        System.Console.WriteLine(employee.FirstName);
    }
}
public class Employee
{
    // FirstName 属性
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
        set
        {
            _FirstName = value;
        }
    }
    private string _FirstName;
    // ...
}
```

输出 6.5

```
Inigo
```

在代码清单 6.17 中，最引人注目的不是属性本身，而是 Program 类的代码。现在其实已经没有 FirstName 和 LastName 字段了，但这一点从 Program 类本身看不出来。访问员工名字和姓氏所用的代码根本没有改变。仍然可以使用简单的赋值操作符对姓或名进行赋值，例如 employee.FirstName = "Inigo"。

属性的关键在于，它提供了从编程角度看类似于字段的 API。但是，实际并不存在这样的字段。属性声明看起来和字段声明一样，但跟随在属性名之后的是一对大括号，要在其中添加属性的实现。属性的实现由两个可选的部分构成。其中，get 标志属性的取值方法（getter），直接对应代码清单 6.16 定义的 GetFirstName()和 GetLastName()方法。访问 FirstName 属性需调用 employee.FirstName。类似地，set 标志属性的赋值方法（setter），它实现了字段赋值语法，如下所示。，

```
employee.FirstName = "Inigo";
```

属性的定义使用了三个上下文关键字。其中，get 和 set 关键字分别标识属性的取值和赋值部分。此外，赋值方法可用 value 关键字引用赋值操作的右侧部分。所以，当 Program.Main()调用 employee.FirstName = "Inigo"时，赋值方法中的 value 被自动设为 "Inigo"，该值可以赋给_FirstName 字段。代码清单 6.17 的属性实现是最常见的。调用取值方法时，比如 Console.WriteLine(employee2.FirstName)，会获取字段（_FirstName）的值并将其写入控制台。

从 C# 7.0 起，还可以使用**表达式主体成员**来声明属性的取值和赋值方法，如代码清单 6.18 所示。

代码清单 6.18　用表达式主体成员定义属性

```
public class Employee
{
    // FirstName 属性
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
        set
        {
            _FirstName = value;
```

```
        }
    }

    private string _FirstName;
    // LastName 属性
    public string LastName
    {
        get => _LastName;
        set => _LastName = value;
    }
    private string _LastName;
    // ...
}
```

代码清单 6.18 用两种不同的语法实现属性，但这只是为了演示，实际编程时请统一。

6.6.2 自动实现的属性

从 C# 3.0 起属性语法有了简化版本。在属性中声明支持字段（比如上例的_FirstName），并用取值方法和赋值方法来获取和设置该字段——由于这是十分常见的设计，而且代码非常简单（参考 FirstName 和 LastName 的实现就知道了），所以现在允许在声明属性时不添加取值或赋值方法，也不声明任何支持字段。一切都自动实现。代码清单 6.19 展示了如何用简化的语法定义 Title 和 Manager 属性，输出 6.6 是结果。

代码清单 6.19　自动实现的属性

```
public class Program
{
    public static void Main()
    {
        Employee employee1 =
            new();
        Employee employee2 =
            new();

        // 调用 FirstName 属性的取值方法(setter)
        employee1.FirstName = "Inigo";

        // 调用 FirstName 属性的赋值方法(getter)
        System.Console.WriteLine(employee1.FirstName);

        // 向自动实现的属性赋值
        employee2.Title = "电脑发烧友";
        employee1.Manager = employee2;
```

```
        // 打印 employee1 的经理的 Title
        System.Console.WriteLine(employee1.Manager.Title);
    }
}

public class Employee
{
    // FirstName 属性
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
        set
        {
            _FirstName = value;
        }
    }
    private string _FirstName;

    // LastName 属性
    public string LastName
    {
        get => _LastName;
        set => _LastName = value;
    }
    private string _LastName;

    // Title 属性
    public string? Title { get; set; }

    // Manager 属性
    public Employee? Manager { get; set; }

    public string? Salary { get; set; } = "不够";
    // ...
}
```

输出 6.6

```
Inigo
电脑发烧友
```

自动实现的属性简化了写法，也使代码更易读。此外，如果未来需添加一些额外的代码，比如要在赋值方法中对值进行验证，那么虽然要修改现在的属性声明来包含实现，但调用它们的代码不必进行任何修改。

关于自动实现的属性，最后要注意从 C# 6.0 开始，可以像代码清单 6.19 最后一行那样初始化：

```
public string? Salary { get; set; } = "不够";
```

在 C# 6.0 之前，只能通过方法（包括构造函数，本章稍后会讲到）来初始化属性。但现在可以用字段初始化那样的语法在声明的同时初始化自动实现的属性。

6.6.3 属性和字段的设计规范

由于还可以写显式的赋值和取值方法而不是属性，所以有时会疑惑该用属性还是方法。一般原则是方法代表行动，而属性代表数据。属性旨在简化对简单数据的访问。调用属性的代价不应比访问字段高出太多。

至于命名，注意在代码清单 6.19 中，属性名是 `FirstName`，它的支持字段名变成了 `_FirstName`。其实就是添加了下划线前缀的 PascalCase 大小写正式工。对于为属性提供支持的私有字段，其他常见的命名规范还有`_firstName` 和 `m_FirstName`（延续自 C++的命名规范，`m` 代表 member variable，即成员变量）。还可以像局部变量那样采用 camelCase 大小写规范[①]。不过，应尽量避免 camelCase 大小写，因为局部变量和参数也经常采用这种大小写，可能造成名称的重复。另外，为符合封装原则，属性的支持字段不应声明为 `public`。

不管私有字段使用哪一种命名方案，属性都要使用 PascalCase 大小写规范。因此，属性应使用 `LastName` 和 `FirstName` 等形式的名词、名词短语或形容词。事实上，属性和类型同名的情况也不罕见，例如某个 `Person` 对象中的 `Address` 类型的 `Address` 属性。

设计规范

DO use properties for simple access to simple data with simple computations.

要使用属性，通过简单的计算，对简单的数据进行简单的访问。

AVOID throwing exceptions from property getters.

避免从属性取值方法抛出异常。

[①] 我个人更喜欢`_FirstName`，下划线就足够了，名称前的 `m` 太多余。另外，使用与属性名称相同的大小写规范，Visual Studio 代码模板扩展工具中就可以只设置一个字符串，而不必为属性名和字段名各设一个。

DO preserve the original property value if the property throws an exception.

要在属性抛出异常时保留原始属性值。

CONSIDER using the same casing on a property's backing field as what used in the property, distinguishing the backing field with an "_" prefix.

考虑为支持字段和属性使用相同的大小写风格，为支持字段附加"_"前缀来予以区分。

DO name properties using a noun, noun phrase, or adjective.

要使用名词、名词短语或形容词命名属性。

CONSIDER giving a property the same name as its type.

考虑让属性和它的类型同名。

AVOID naming fields with camelCase.

避免用 camelCase 大小写风格命名字段。

DO favor prefixing Boolean properties with "Is," "Can," or "Has," when that practice adds value.

如果有意义的话，**要**为 Boolean 属性附加"Is"，"Can"或"Has"前缀。

DO declare all instance fields as `private` (and expose them via a property).

要将所有实例字段声明为 `private`（并通过属性来公开）。

DO name properties with PascalCase.

要用 PascalCase 大小写风格命名属性。

DO favor automatically implemented properties over fields.

要优先使用自动实现的属性而不是字段。

DO favor automatically implemented properties over using fully expanded ones if there is no additional implementation logic.

如果没有额外的实现逻辑，那么**要**优先使用自动实现的属性而不是自己写属性的完整版本。

6.6.4 提供属性验证

在代码清单 6.20 中，注意 Employee 的 Initialize()方法使用属性而不是字段进行赋值。虽然并非必须如此，但这样做的结果是，无论在类的内部还是外部，属性的赋值方法中的任何验证都会得到调用。例如，假定更改 LastName 属性，在把 value 赋给_LastName 之前检查它是否为 null 或空字符串，那么会发生什么？记住，之所以要进行 null 检查，是因为即使数据类型是非空字符串，但调用者可能已禁用可空引用类型，或者该方法可能是从 C# 7.0 或更早版本中调用的（那时还没有可空引用类型）。

代码清单 6.20　实现属性验证

```
public class Employee
{
    // ...
    public void Initialize(
        string newFirstName, string newLastName)
    {
        // 使用 Employee 类的属性
        FirstName = newFirstName;
        LastName = newLastName;
    }
    // LastName property
    public string LastName
    {
        get => _LastName;
        set
        {
            // ...
            // 验证对 LastName 的赋值
            ArgumentException.ThrowIfNullOrEmpty(value = value?.Trim()!);
            // ...
            _LastName = value;
        }
    }
    private string _LastName;
    // ...
}
```

在新的实现中，如果为 LastName 赋了无效的值（要么从同一个类的另一个成员赋值，要么在 Program.Main()内直接向 LastName 赋值），代码就会抛出异常。拦截赋值，并通过字段风格的 API 对参数进行验证，这是属性的优点之一。

一个好的实践是只从属性的实现中访问属性的支持字段。换言之，要一直使用属性，不要直接调用字段。许多时候，即使在属性所在的类中，也不应该从属性实现的外部访问其支

持字段。如果坚持这个实践，以后一旦为属性增添验证代码，那么整个类就能马上利用这个逻辑，而不必大费周章地修改。[1]

虽然很少见，但确实能在赋值方法中对 value 进行赋值。如代码清单 6.20 所示，我们调用 value.Trim()来移除新姓氏值左右的空白字符。

设计规范

AVOID accessing the backing field of a property outside the property, even from within the containing class.

避免从属性外部（即使是从属性所在的类中）访问属性的支持字段。

6.6.5 只读和只写属性

可以移除属性的取值方法或赋值方法来改变属性的可访问性。只有赋值方法的属性是只写属性，这种情况较罕见。类似地，只提供取值方法会得到只读属性；任何赋值企图都会造成编译错误。例如，为了使 Id 只读，可以像代码清单 6.21 那样编码。

代码清单 6.21　C# 6.0 之前定义只读属性的方式

```
public class Program
{
    public static void Main()
    {
        Employee employee1 = new();
        employee1.Initialize(42);

        // 错误：无法为属性或索引器'Employee.Id'赋值；它是只读的
        employee1.Id = "490";
    }
}

public class Employee
{
    public void Initialize(int id)
    {
```

[1] 本章后面会讲到，一个例外是在字段被标记为只读时。此时只能在构造函数中设置值。从 C# 6.0 开始，可以直接对只读属性赋值，完全用不着只读字段了。

```
        // 使用字段，因为 Id 属性没有 setter；它是只读的
        _Id = id.ToString();
    }

    // ...
    // Id 属性声明
    public string Id
    {
        get => _Id;
        // 未提供 setter
    }
    private string _Id;
}
```

代码清单 6.21 从 Employee 的 Initialize()方法（而不是属性）中对字段赋值（_Id = id）。就像 Program.Main()中展示的那样，通过属性来赋值会造成编译错误。

从 C# 6.0 开始支持**只读自动实现属性**，如下所示。

```
public bool[] Cells { get; } = new bool[9];
```

只读自动实现属性的一个重点在于，和只读字段一样，编译器要求通过一个自动属性初始化器（或通过构造函数）来初始化。上例使用的是一个初始化器（初始化列表[①]），但稍后就会讲到，也可以在构造函数中对 Cells 进行赋值。

由于设计规范是不要从属性外部访问支持字段，所以程序员应该无脑使用只读自动实现属性。唯一例外是在字段和属性类型不匹配的时候。例如字段是 int 类型，只读属性是 double 类型。

设计规范

DO create read-only properties if the property value should not be changed.

如果属性值不变，**要**创建只读属性。

DO create read-only automatically implemented properties, rather than read-only properties with a backing field if the property value should not be changed.

如果属性值不变，**要**创建只读自动实现的属性，而不是只读属性加支持字段。

[①] 译注：文档中翻译为“初始值设定项”。

6.6.6 计算属性

有的时候，我们甚至根本不需要支持字段。相反，可以让属性的取值方法返回一个计算好的值，同时让赋值方法解析值，并可选择将值持久存储到其他成员字段中。代码清单 6.22 的 Name 属性就是一个例子，输出 6.7 展示了结果。

代码清单 6.22　定义计算属性

```
public class Program
{
    public static void Main()
    {
        Employee employee1 = new();

        employee1.Name = "Inigo Montoya";
        System.Console.WriteLine(employee1.Name);
        // ...
    }
}
public class Employee
{
    // ...
    // FirstName 属性
    public string FirstName
    {
        get
        {
            return _FirstName;
        }
        set
        {
            _FirstName = value;
        }
    }
    private string _FirstName;

    // LastName 属性
    public string LastName
    {
        get => _LastName;
        set => _LastName = value;
    }
    private string _LastName;
    // ...

    // Name 属性
```

```
    public string Name
    {
        get
        {
            return $"{FirstName} {LastName}";
        }
        set
        {
            // ...
            ArgumentException.ThrowIfNullOrEmpty(value = value?.Trim()!);
            // ...
            // 将所赋的值拆分为名字和姓氏
            string[] names;
            names = value.Split(new char[] { ' ' });
            if (names.Length == 2)
            {
                FirstName = names[0];
                LastName = names[1];
            }
            else
            {
                // 如果没有赋全名，就抛出异常
                throw new System.ArgumentException(
                    $"所赋的值'{value}'无效。",
                    nameof(value));
            }
        }
    }

    public string Initials => $"{FirstName[0]} {LastName[0]}";
    // ...
}
```

输出 6.7

```
Inigo Montoya
```

Name 属性的取值方法将 FirstName 和 LastName 属性的返回值（两个 string）连接到一起。事实上，所赋的姓名并没有真正存储下来。向 Name 属性赋值时，右侧的值会解析成名字和姓氏部分。

6.6.7 取值和赋值方法的访问修饰符

如前所述，一个好的实践是不要从属性外部访问其字段，否则为属性添加的验证逻辑或其他逻辑可能失去意义。

在属性的实现中，我们可以为 get 或 set 部分指定访问修饰符（但不能为两者都指定）[1]，从而覆盖在声明属性时指定的访问修饰符。代码清单 6.23 展示了一个例子。

代码清单 6.23　为赋值方法指定访问修饰符

```
public class Program
{
    public static void Main()
    {
        Employee employee1 = new();
        employee1.Initialize(42);
        // 错误：无法为属性或索引器'Employee.Id'赋值；它是只读的
        employee1.Id = "490";
    }
}

public class Employee
{
    public void Initialize(int id)
    {
        // 设置 Id 属性
        Id = id.ToString();
    }

    // ...
    // Id 属性声明
    public string Id
    {
        get => _Id;
        private set => _Id = value;
    }
    private string _Id;
}
```

为赋值方法指定 private 修饰符，属性对于除 Employee 的其他类来说就是只读的。在 Employee 类内部，属性可读/可写，所以可以在构造函数中对属性进行赋值。为取值或赋值方法指定访问修饰符时，要注意该访问修饰符的“限制性”一定要比应用于整个属性的访问修饰符更“严格”。例如，将属性声明为较严格的 private，但将它的赋值方法声明为较宽松的 public，就会发生编译错误。

[1] 这个功能是从 C# 2.0 开始引入的。C# 1.0 不允许为属性的取值和赋值方法指定不同的封装级别。换言之，不能为属性创建一个公共取值方法和一个私有赋值方法，使外部类只能对属性进行只读访问，而允许类内的代码向属性写入。

设计规范

DO apply appropriate accessibility modifiers on implementations of getters and setters on all properties.

要为所有属性的取值和赋值方法应用适当的可访问性修饰符。

DO NOT provide set-only properties or properties with the setter having broader accessibility than the getter.

不要提供只写属性，也不要让赋值方法的可访问性比取值方法更宽松。

6.6.8 属性和方法调用不允许作为 ref 或 out 参数值

C#允许属性像字段那样使用，只是不允许作为 ref 或 out 参数值传递。ref 和 out 参数内部要将内存地址传给目标方法。但是，由于属性可能没有支持字段，也有可能只读或只写，所以不可能传递存储地址。同样的道理也适用于方法调用。如果需要将属性或方法调用作为 ref 或 out 参数值传递，那么必须先将值拷贝到变量再传递该变量。方法调用结束后，再将变量的值赋回属性。

> **高级主题：属性的内部工作机制**
>
> 代码清单 6.24 证明取值方法和赋值方法在 CIL 代码中以 get_FirstName()和 set_FirstName()的形式出现。
>
> **代码清单 6.24　属性的 CIL 代码**
>
> ```
> // ...
>
> .field private string _FirstName
> .method public hidebysig specialname instance string
>         get_FirstName() cil managed
>    {
>         // Code size       12 (0xc)
>         .maxstack  1
>         .locals init (string V_0)
>         IL_0000:  nop
>         IL_0001:  ldarg.0
>         IL_0002:  ldfld      string Employee::_FirstName
>         IL_0007:  stloc.0
> ```

```
    IL_0008:  br.s IL_000a

    IL_000a:  ldloc.0
    IL_000b:  ret
  } // End of method Employee::get_FirstName

  .method public hidebysig specialname instance void
          set_FirstName(string 'value') cil managed
  {
    // Code size       9 (0x9)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldarg.1
    IL_0003:  stfld      string Employee::_FirstName
    IL_0008:  ret
  } // End of method Employee::set_FirstName

  .property instance string FirstName()
  {
    .get instance string Employee::get_FirstName()
    .set instance void Employee::set_FirstName(string)
  } // End of property Employee::FirstName

  // ...
```

除了外观与普通方法无异，注意属性在 CIL 中也是一种显式的构造。如代码清单 6.25 所示，取值方法和赋值方法由 CIL 属性调用，而 CIL 属性是 CIL 代码中的一种显式构造。因此，语言和编译器并非总是依据一个命名惯例来解释属性。相反， 正是由于反正最后都会回归 CIL 属性，所以编译器和代码编辑器能随便提供自己的特殊语法。

代码清单 6.25　属性是 CIL 的显式构造

```
.property instance string FirstName()
{
  .get instance string Program::get_FirstName()
  .set instance void Program::set_FirstName(string)
} // End of property Program::FirstName
```

注意，在代码清单 6.24 中，作为属性一部分的取值方法和赋值方法包含了 specialname 元数据。IDE（比如 Visual Studio）根据该修饰符在“智能感知”（IntelliSense）中隐藏成员。

> 自动实现的属性在 CIL 中看起来和显式定义支持字段的属性几乎完全一样。C#编译器在 IL 中生成名为<*PropertyName*>k_BackingField 的字段。该字段应用了名为 System.Runtime.CompilerServices.CompilerGeneratedAttribute 的特性（参见第 18 章）。无论取值还是赋值方法都用同一个特性修饰。

6.7 构造函数

现在，我们已经为类添加了用于存储数据的字段，接着应考虑数据的有效性。如代码清单 6.6 所示，可以使用 new 操作符实例化对象。但这样可能创建包含无效数据的员工对象。

实例化 employee1 后得到的是姓名和工资尚未初始化的 Employee 对象。在该代码清单中，是在实例化员工之后，立即对尚未初始化的字段进行赋值。但假如忘了初始化，编译器也不会发出警告。结果是得到含有无效姓名的 Employee 对象。（从技术上说，从 C# 8.0 开始，非空（不可为 null）的引用类型会触发一个警告，并建议将数据类型切换为可空类型，以避免默认为 null 的情况。尽管如此，为了避免实例化其字段包含无效数据的对象，仍然需要进行初始化。）

6.7.1 声明主构造函数

为了解决该问题，必须提供一种方式在创建对象时指定必须的数据。这是用构造函数来实现的，如代码清单 6.26 所示。

代码清单 6.26　定义主构造函数

```
// Employee 构造函数
public class Employee(string firstName, string lastName)
{
    public string FirstName { get; set; } = firstName;
    public string LastName { get; set; } = lastName;
    public string? Salary { get; set; } = "不够";
    // ...
}
```

对于**主构造函数**（primary constructor），注意是在类型声明后添加了一个方法签名。这提供了作用域局部于类的变量（称为**位置参数**）。而且如代码清单 6.26 所示，这些变量可以作为属性初始化器来赋值。类的任何实例成员都能访问主构造函数的变量，所以允许将 firstName 和 lastName 分别赋给 FirstName 和 LastName 属性。

“运行时”会调用构造函数来初始化对象实例。本例的构造函数获取名字和姓氏作为参数，

允许程序员在实例化 Employee 对象时指定这些参数的值。代码清单 6.27 演示了如何调用构造函数。

代码清单 6.27　调用构造函数

```
public class Program
{
    public static void Main()
    {
        Employee employee;
        employee = new("Inigo", "Montoya");
        employee.Salary = "太少了";
        System.Console.WriteLine("{0} {1}: {2}",
            employee.FirstName,
            employee.LastName,
            employee.Salary);
    }
    // ...
}
```

注意，new 操作符返回对实例化好的对象的一个引用。另外，已移除了名字和姓氏的初始化代码，因为现在是在构造函数内部初始化。但是，由于本例没有在构造函数内部初始化 Salary，所以对工资进行赋值的代码还是保留下来了。

> **高级主题：new 操作符的实现细节**
>
> new 操作符内部和构造函数是像下面这样交互的。new 操作符从内存管理器获取“空白”内存，调用指定的构造函数，将对“空白”内存的引用作为一个隐式的 this 参数传给构造函数。构造函数链剩余的部分开始执行，在构造函数之间传递引用。这些构造函数都没有返回类型（表现得像是返回 void）。构造函数链上的执行结束后，new 操作符返回内存引用。现在，该引用指向的内存处于已初始化好的形式。

6.7.2 定义构造函数

上一节解释了如何定义主构造函数。我们还可以单独定义构造函数，与类型声明分开。如代码清单 6.28 所示，为了定义（非主）构造函数，可以创建一个没有返回类型的方法，其方法名与类型名相同。即使在构造函数的声明或实现中没有指定返回类型或 return 语句，调用构造函数（通过 new 操作符）仍然会返回类型的实例。

代码清单 6.28　定义构造函数

```
public class Employee
{
    // Employee 构造函数
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? Salary { get; set; } = "不够";
    // ...
}
```

开发人员应注意到既在声明中又在构造函数中赋值的情况。如果属性或字段在声明的同时赋值（比如代码清单 6.28 中的 `string? Salary { get; set; } = "不够"`，或者代码清单 6.5 中的 `public string? Salary = "不够"`），那么只有在这个赋值发生之后，构造函数内部的赋值才会发生。所以，最终生效的是构造函数内部的赋值，它会覆盖声明时的赋值。如果不细心，很容易就会以为对象实例化后保留的是声明时所赋的属性或字段值。所以，有必要考虑一种编码风格，避免同一个类中既在声明时赋值，又在构造函数中赋值。

6.7.3 默认和拷贝构造函数

必须注意，一旦像之前的例子那样显式添加了构造函数，在 `Main()`中实例化 `Employee` 就必须指定名字和姓氏。代码清单 6.29 的代码无法编译。

代码清单 6.29　没有默认构造函数了

```
public class Program
{
    public static void Main()
    {
        Employee employee;

        // 错误：没有获取 0 个参数的 Employee 方法重载
        employee = new Employee();

        // ...
    }
}
```

如果类没有显式定义的构造函数，C#编译器会在编译时自动添加一个。该构造函数不获取参数，称为**默认构造函数**。一旦为类显式添加了构造函数，C#编译器就不再自动提供默认构造函数。因此，在定义了有参的 `Employee(string firstName, string lastName)` 构造函数之后，编译器不再添加无参的默认构造函数 `Employee()`。虽然可以手动添加一个，但会再度允许构造没有指定员工姓名的 `Employee` 对象。

我们其实没有必要依赖编译器提供的默认构造函数。程序员任何时候都可以显式定义默认构造函数，比如用它将某些字段初始化成特定值。无参构造函数就是默认构造函数。

拷贝构造函数（copy constructor）[1]只获取一个参数，该参数必须具有包容类型。下面展示了一个例子。

```
public Employee(Employee original)
{
    // 在不同员工对象之间拷贝属性
}
```

利用这种构造函数，我们可以方便地克隆一个对象实例，创建内容一样的新实例。

6.7.4 对象初始化器

为了初始化对象中所有可以访问的字段和属性，可以使用一个称为**对象初始化器**（object initializer）的概念。具体地说，调用构造函数创建对象时，可在后面的一对大括号中添加成员初始化列表。每个成员的初始化操作都是一个赋值操作，等号左边是可以访问的字段或属性，右边是要赋的值。如代码清单 6.30 所示。

代码清单 6.30　调用对象初始化器

```
public class Program
{
    public static void Main()
    {
        Employee employee = new("Inigo", "Montoya")
           { Title = "电脑发烧友", Salary = "不够" };
        // ...
    }
```

[1] 译注：众所周知，微软文档喜欢把一切 copy 翻译为“复制”，这有点不符合来自 C++等语言的程序员的习惯。所以，在文档中看到“复制构造函数”时，请自行脑补为“拷贝构造函数”。

```
    }
```

注意，使用对象初始化器时要遵守相同的构造函数规则。这实际只是一种语法糖，最终生成的 CIL 代码和创建对象实例后单独用语句对字段及属性进行赋值无异。C#代码中的成员初始化顺序决定了在 CIL 中调用构造函数后的属性和字段赋值顺序。

总的来说，当构造函数退出时，所有属性都应初始化成合理的默认值。此外，利用属性的赋值方法的验证逻辑，可以制止将无效数据赋给属性。但是，偶尔一个或多个属性的值可能导致同一个对象的其他属性暂时包含无效值。这时应暂缓为无效状态抛出异常，直到对象实际使用这些相关属性时再抛出。

设计规范

DO provide sensible defaults for all properties, ensuring that defaults do not result in a security hole or significantly inefficient code.

要为所有属性提供有意义的默认值，确保默认值不会造成安全漏洞或显著影响代码执行效率。

DO allow properties to be set in any order even if this results in a temporarily invalid object state.

要允许属性以任意顺序设置，即使这会造成对象短时处于无效状态。

> 高级主题：集合初始化器
>
> 从 C# 3.0 开始引入了**集合初始化器**（collection initializer）的概念，它采用和对象初始化器相似的语法，用于在集合实例化期间向集合项赋值。它借用数组语法，在创建集合的同时初始化集合中的每一项。例如，为了初始化由 `Employee` 对象构成的一个列表，可在构造函数调用之后的一对大括号中指定每一项，如代码清单 6.31 所示。
>
> 代码清单 6. 31　调用集合初始化器
>
> ```
> public class Program
> {
>     public static void Main()
>     {
>         List<Employee> employees = new()
> ```

```
                {
                    new("Inigo", "Montoya"),
                    new("Kevin", "Bost")
                };
            // ...
        }
    }
```

像这样为新集合实例赋值，编译器生成的代码会按顺序实例化每个对象，并自动使用`Add()`方法把它们添加到集合。

6.7.5 仅初始化的赋值函数

对象初始化器允许在对象初始化期间指定成员的值。但是，所有只读属性都不能像这样设置，因为只有在对象构造期间，才能设置只读属性（这种属性只有 getter），对象初始化器是在此之后运行的。为了解决这个问题，C# 9.0 新增了对仅**初始化赋值函数**（init-only setter）的支持。它们可以在对象初始化器中设置，但在此之后不能设置。代码清单 6.32 演示了如何使用这种 setter。

代码清单 6.32　仅初始化的赋值函数（init-only setter）

```
public class Employee
{
    public Employee(int id, string name)
    {
        Id = id;
        Name = name;
        Salary = null;
    }

    // ...
    public int Id { get; }
    public string Name { get; }

    public string? Salary
    {
        get => _Salary;
        init => _Salary = value;  // init-only setter
    }
    private string? _Salary;
}
```

```
public class Program
{
    public static void Main()
    {
        Employee employee = new(42, "Inigo Montoya")
        {
            Salary = "非常充足"
        };

        // 错误：属性或索引器'Employee.Salary'不能在初始化结束后赋值
        employee.Salary = "够了";
    }
}
```

注意，虽然可以在对象初始化器中设置 `Salary` 的值，但设置好后就不能修改。那个值毕竟是只读的。

> 高级主题：终结器
>
> **构造函数**（constructor）定义了在类的实例化过程中发生的事情。为了定义在对象销毁过程中发生的事情，C#提供了**终结器**（finalizer）。和 C++的**析构器**或**析构函数**（destructor）不同，终结器不是在对一个对象的所有引用都消失后马上运行。相反，终结器在对象被判定“不可到达”之后的某个不确定的时间执行。具体地说，垃圾回收器会在一次垃圾回收过程中识别出带有终结器的对象。但是，它不是立即回收这些对象，而是把它们添加到一个**终结队列**中。一个独立的线程遍历终结队列中的每个对象，调用其终结器，然后将其从队列中删除，使其再次可供垃圾回收器处理。第 10 章深入讨论了这个过程以及资源清理的主题。

6.7.6 重载构造函数

构造函数可以重载。可以同时存在多个构造函数，只要参数数量和类型不同即可。如代码清单 6.33 所示，可以提供一个构造函数，除了获取员工姓名还获取员工 ID；再提供一个构造函数只获取员工 ID。

代码清单 6.33　重载构造函数

```
public class Employee
{
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
```

```
    public Employee(
        int id, string firstName, string lastName)
    {
        Id = id;
        FirstName = firstName;
        LastName = lastName;
    }

    // FirstName 和 LastName 在 Id 属性的 setter 中设置
    #pragma warning disable CS8618
    public Employee(int id) => Id = id;
    #pragma warning restore CS8618

    private int _Id;
    public int Id
    {
        get => _Id;
        private set
        {
            // 查找员工姓名...
            // ...
        }
    }
    // ...
    public string FirstName { get; set; }
    // ...
    public string LastName { get; set; }
    public string? Salary { get; set; } = "不够";

    // ...
}
```

这样一来，当 Program.Main()根据姓名来实例化员工对象时，既可只传递员工 ID，也可同时传递姓名和 ID。例如，创建新员工时调用同时获取姓名和 ID 的构造函数，而从文件或数据库加载现有员工时调用只获取 ID 的构造函数。

和方法重载一样，通过提供多个版本的构造函数，我们平时可以传递少量参数来支持简单情况，传递额外的参数来支持复杂情况。应优先使用可选参数而不是重载，以便在 API 中清楚地看出“默认”属性的默认值。例如，构造函数签名 Person(string firstName, string lastName, int? age = null)清楚地指明了假如 Person 的年龄未指定，就默认为 null。

还要注意的是，从 C# 7.0 开始支持构造函数的**表达式主体成员**实现，例如：

```
// FirstName 和 LastName 在 Id 属性的 setter 中设置
```

```
#pragma warning disable CS8618
public Employee(int id) => Id = id;
```

在本例中，我们是调用 `Id` 属性以实现对 `FirstName` 和 `LastName` 的赋值。遗憾的是，编译器未检测到会在 `Id` 属性内部发生的这个赋值，并且从 C# 8.0 开始，会发出一个警告建议将这些属性标记为可空。由于我们实际上会设置这些属性，所以主动禁用了警告。

设计规范

DO use the same name for constructor parameters (camelCase) and properties (PascalCase) if the constructor parameters are used to simply set the property.

如果构造函数的参数只是用于设置属性，那么构造函数参数（camelCase）**要**使用和属性（PascalCase）相同的名称，区别仅在于首字母的大小写。

DO provide constructor optional parameters or constructor overloads that initialize properties with good defaults.

要为构造函数提供可选参数，或者提供重载构造函数，用好的默认值初始化属性。

6.7.7 构造函数链：使用 `this` 调用另一个构造函数

注意，代码清单 6.33 对 `Employee` 对象进行初始化的代码在好几个地方重复，所以必须在多个地方维护。虽然本例的代码量较小，但完全可以从一个构造函数中调用另一个构造函数，以避免重复输入代码。这称为**构造函数链**，是用**构造函数初始化器**（constructor initializers）来实现的。构造函数初始化器会在执行当前构造函数的实现之前，判断要调用另外哪一个构造函数，如代码清单 6.34 所示。

代码清单 6.34　从一个构造函数中调用另一个

```
public class Employee
{
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
    public Employee(
        int id, string firstName, string lastName)
        : this(firstName, lastName)
    {
```

```
        Id = id;
    }

    // FirstName 和 LastName 在 Id 属性的 setter 中设置
    #pragma warning disable CS8618
    public Employee(int id)
    {
        Id = id;
        // 查找员工姓名...
        // ...

        // 注意: 成员构造函数不能以内联方式显式调用
        // this(id, firstName, lastName);
    }
    #pragma warning restore CS8618

    public int Id { get; private set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? Salary { get; set; } = "Not Enough";
    // ...
}
```

针对相同对象实例，为了从一个构造函数中调用同一个类的另一个构造函数，C#语法是在一个冒号后添加 `this` 关键字，再添加被调用构造函数的参数列表。本例是获取三个参数的构造函数调用获取两个参数的构造函数。但是，我们平时一般采用相反的调用模式，即参数最少的构造函数调用参数最多的，为未知参数传递默认值。

> **初学者主题：集中初始化**
>
> 如代码清单 6.34 所示，在 `Employee(int id)`构造函数的实现中不能调用 `this(id, firstName, lastName)`，这是因为该构造函数没有 `firstName` 和 `lastName` 这两个参数。要将所有初始化代码都集中到一个方法中，必须创建单独的方法，如代码清单 6.35 所示。
>
> **代码清单 6.35　提供初始化方法**
>
> ```
> public class Employee
> {
>     // FirstName 和 LastName 在 Initialize()方法内部设置
>     #pragma warning disable CS8618
>     public Employee(string firstName, string lastName)
>     {
>         int id;
> ```

```
            // 生成 employee ID...
            // ...
            Initialize(id, firstName, lastName);
        }

        public Employee(int id, string firstName, string lastName)
        {
            Initialize(id, firstName, lastName);
        }

        public Employee(int id)
        {
            string firstName;
            string lastName;
            Id = id;

            // 查找员工数据
            // ...

            Initialize(id, firstName, lastName);
        }
        #pragma warning restore CS8618

        private void Initialize(
            int id, string firstName, string lastName)
        {
            Id = id;
            FirstName = firstName;
            LastName = lastName;
        }
        // ...
    }
```

在本例中，负责集中初始化的方法是 `Initialize()`，它同时获取员工的名字、姓氏和 ID。注意，像之前的代码清单 6.34 展示的那样，仍然可以从一个构造函数中调用另一个构造函数。

6.8 在构造函数中初始化非空引用类型的属性

在本章的所有示例程序中，我们禁用了以下 C#可空警告。

> CS8618：不可为 null 的字段/属性未初始化（必须包含非 null 值），请考虑将属性声明为可以为 null。[1]

声明引用类型的非空字段或非空自动实现的属性时，这些字段和属性显然应该在在包容对

[1] 译注：大白话就是“字段或属性不可为空（非空），但又没有初始化，所以考虑声明为可空。”

象完全实例化之前完成初始化。不这么做，这些字段和属性就会保持默认的 null 值。而既然可以为 null，为什么要声明为非空？

但问题在于，非空字段和属性经常是间接初始化的。这个初始化位置在当前构造函数的作用域外，因此也超出了编译器代码分析的范围——即便它们仍然是通过构造函数调用的方法或属性来初始化的[①]。下面展示了该实践的一些例子。

- 在一个简单属性中，在将值赋给编译器报告为未初始化的支持字段之前，验证要赋给字段的值非空（参见代码清单 6.20）。
- 用计算属性（例如代码清单 6.22 的 Name 属性）来设置类的其他非空属性或字段。
- 像清单 6.34 和清单 6.35 那样用一个方法来集中初始化。
- 公共属性由触发实例化然后初始化属性的外部代理完成初始化。[②]

大多数情况下，引用类型的非空字段或非空自动实现的属性（本节称为非空字段/属性，暗示是“引用类型”）是通过构造函数调用的属性或方法来间接赋值的。遗憾的是，C#编译器不识别对非空字段/属性的间接赋值。

此外，所有非空字段/属性都需要确保它们不被赋一个 null 值。在字段的情况下，它们需要封装到属性中，用赋值方法（setter）的逻辑进行验证，确保不会被赋 null 值（记住，字段验证要依赖于以下设计规范：不从字段的包装属性外部访问字段）。结果是，对于非空的、可读/可写的、完全实现的引用类型属性，应该设计验证机制来防止被赋 null 值。

对于非空的自动实现属性，需要将封装性限制为只读，在实例化期间赋值，并在赋值前验证所赋的值不为 null。至于可读可写的、非空引用类型的自动实现属性，则应当尽量避免，特别是那些具有公共 setter 的，这是因为不好防止将 null 值赋给它。虽然可以从构造函数中向属性赋值，从而避免产生“不可为 null 的属性未初始化”编译器警告，但这还不够：该属性是可读可写的，因此它完全可能在实例化后被赋 null 值，使你把它设为“非空”意图落不到实处。

6.8.1 可读/可写非空引用类型的属性

代码清单 6.36 演示了如何告诉编译器不要错误地显示警告“不可为 null 的字段/属性尚未初始化”。最终目标是让程序员告知编译器这些属性/字段是非空（不可为 null）的，使编译器可以通知调用者这些属性/字段不可为 null。

[①] 也可能通过一个代理（例如第 18 章讲述的反射）来初始化。

[②] 例如 MSTest 中的 TestContext 属性，或者通过依赖注入（Dependency Injection，DI）来初始化的对象。

代码清单 6.36　在非空属性上提供验证

```
public class Employee
{
    public Employee(string name)
    {
        Name = name;
    }
    public string Name
    {
        get => _Name!;
        set => _Name = value ?? throw new ArgumentNullException(
            nameof(value));
    }
    private string? _Name;
    // ...
}
```

上述代码处理未直接由构造函数初始化的非空属性/字段，它具有几个重要特点（排名不分先后）。

- 属性的赋值方法在设置非可空字段的值之前有一个检查 `null` 的操作。代码清单 6.36 使用了空合并操作符，如果新值为 `null`，就抛出一个 `ArgumentNullException`。
- 构造函数调用一个对非可空字段进行间接赋值的方法或属性，但未识别该字段被初始化为非 `null` 值。
- 支持字段 `_Name` 声明为可空，以避免编译器警告字段未初始化。
- 取值函数使用空包容操作符返回字段。因为已通过了赋值函数的验证，所以可以放心地宣称它不为 `null`。

对于非空属性，将支持字段声明为可空似乎是没有意义的。但这是必要的，因为编译器无法识别在构造函数外部对非空字段/属性的赋值。幸好，由于赋值方法中的非空检查确保字段永远不为 `null`，所以作为程序员，你有权在返回字段时使用空包容操作符（`!`）。

6.8.2 只读自动实现的引用类型属性

本节之前说过，非空自动实现的引用类型属性应该是只读的，以避免无效的 `null` 赋值。然而，在实例化期间，还是需要对参数进行验证，如代码清单 6.37 所示。

代码清单 6.37　验证非空引用类型的自动实现属性

```
public class Employee
{
    public Employee(string name)
    {
        Name = name ?? throw new ArgumentNullException(nameof(name));
    }

    public string Name { get; }
}
```

有些人觉得，对于非空引用类型的自动实现属性，为它提供一个私有赋值函数就可以了。[①]虽然可以这样做，但一个更恰当的问题是，类是否可能错误地将 null 赋给属性？如果不在赋值方法中对字段赋值进行验证，那么如何保证不会错误地将 null 值赋给它呢？尽管编译器会在实例化期间验证你的意图，但开发人员是否总是能像代码清单 6.37 展示的构造函数那样，记得对传给你的类的数据进行 null 值检查呢？

设计规范

DO implement non-nullable read/write reference fully implemented properties with a nullable backing field, a null-forgiveness operator when returning the field from the getter, and non-null validation in the property setter.

实现非空的、可读/可写的、引用类型的、完全实现的属性时，**要**用一个可空的支持字段，在取值函数中使用空包容操作符返回字段，并在属性的赋值方法中进行非空验证。

DO assign non-nullable reference type properties before instantiation completes.

要在实例化完成之前完成向非空引用类型属性的赋值。

DO implement non-nullable reference type automatically implemented properties as read-only.

要将非空引用类型的自动实现属性实现为只读。

[①] 译注：声明属性时若只提供一个 get;，会自动生成一个私有 setter 和一个私有支持字段。

DO use a nullable check for all reference type properties and fields that are not initialized before instantiation completes.

对于在实例化完成之前未初始化的所有引用类型的属性和字段，**要**执行可空检查。

6.8.3 required 修饰符

从 C# 11 开始，可以将字段或属性标记为必需（required）。这种字段或属性必须在对象构造期间通过对象初始化器来赋值（参见代码清单 6.38）。

代码清单 6.38　使用 required 修饰符

```
public class Book
{
    string? _Title;
    public required string Title
    {
        get
        {
            return _Title!;
        }
        set
        {
            _Title = value ?? throw new ArgumentNullException(nameof(value));
        }
    }

    string? _Isbn;
    public required string Isbn
    {
        get
        {
            return _Isbn!;
        }
        set
        {
            _Isbn = value ?? throw new ArgumentNullException(nameof(value));
        }
    }
    public string? Subtitle { get; set; }
    // ...
}

public class Program
```

```
{
    public static void Main()
    {
        // ...
        Book book = new()
        {
            Isbn = "978-0135972267",
            Title = "阿罗有支彩色笔"
        };
        // ...
    }
}
```

添加 required 修饰符后，在实例化一本书的时候，就必须在一个对象初始化器中同时为书号（Isbn）和书名（Title）提供值了，如代码清单 6.39 所示。

代码清单 6.39　必须在对象初始化器中为必需的成员赋值

```
// Error CS9035:
// 必须在对象初始化值设定项或属性构造函数中
// 设置所需的成员'Book.Isbn'。①
Book book = new() { Title = "C#本质论" };

// ...
```

注意，由于 Book 没有显式地包含一个构造函数，所以我们依赖自动生成的默认构造函数来实例化书籍。这正是我们想要的效果，因为各个 required 成员定义了如何构造对象，可以取代任何构造函数的作用。相反，如果为 required 成员提供带有参数的构造函数，那么在构造函数参数和对象初始化器中都要指定这些值，这多少显得有点多余。例如，假定提供了一个接收 Title （书名）参数的构造函数，那么必须像下面这样实例化。

```
Book book = new("机器学习与人工智能实战：基于业务场景的工程应用")
{
    Title = "机器学习与人工智能实战",
    Isbn = " 9787302635239"
};
```

① 译注：如前所述，文档中将 object initializer 翻译为“对象初始值设定项”。我们采用“对象初始化器”或者“对象初始化列表”。

为了避免这种冗余，可以使用 SetsRequiredMembers 特性来修饰构造函数，指示编译器在调用相关构造函数时，禁用所有对“对象初始化器”的要求。（从本质上说，SetsRequiredMembers 属性会告诉编译器，现在由开发人员负责设置所有 required 成员，所以编译器可以忽略对初始化赋值的检查。但令人遗憾的是，至于是不是真的发生了这样的初始化，编译器是不会跑去核实的。代码清单 6.40 展示了如何使用 SetsRequiredMembers 特性。

代码清单 6.40　禁用 required 对象初始化

```
[SetsRequiredMembers]
public Book(int id)
{
    Id = id;

    // 查找书籍数据
    // ...
    // ...
}
```

有了这样的构造函数，就可以在实例化一本书的时候不设置任何 required 成员。

```
Book book = new(42) {
    Subtitle = "基于业务场景的工程应用"};
```

当然，这样做的缺点在于，它假设构造函数会将有效的值赋给 required 成员。如果证实不了这个假设，那么告诉编译器忽略 required 成员是没有意义的。这样做反面允许构造函数不设置 required 成员，并留下无效的值。另外，对于非 required 成员，相较于只允许通过对象初始化器来设置，是不是使用构造函数来设置更有优势呢？另一方面，如果某个数据类型的默认值本身就是有效的，那么显然没必要将值标记为 required。

你可能已经猜到了，不能设置一个带有私有赋值方法的 required 公共类型。相反，required 成员的赋值方法必须匹配成员所在的那个类型的可见性。例如，由于 Book 是公共的，所以它的 Isbn 属性的赋值方法也必须公共。

最后需要注意的是，在将类发布到生产环境后，如果新增 required 成员，那么会导致实例化该类型的现有代码编译失败，因为现在必须通过对象初始化器向这些 required 成员赋值。这就使新版本变得与现有代码不兼容；因此，应该避免这种情况。

设计规范

DO NOT use constructor parameters to initialize required properties; instead, rely on object initializer–specified values.

不要使用构造函数参数来初始化 `required` 属性；相反，依赖于对象初始化器指定的值。

DO NOT use the `SetsRequiredMembers` attribute unless all `required` parameters are assigned valid values during construction.

除非在构造过程中向所有 `required` 参数赋了有效的值，否则**不要**使用 `SetsRequiredMembers` 特性。

CONSIDER having a default constructor only on types with `required` parameters, relying on the object initializer to set both required and non-required members

考虑仅为带有 `required` 参数的类型使用默认构造函数，依赖对象初始化器来设置必需和非必需成员。

AVOID adding `required` members to released types to avoid breaking the compile on existing code.

避免为已发布的类型添加 `required` 成员，避免破坏现有代码的编译。

AVOID `required` members where the default value of the type is valid.

避免在类型的默认值有效时使用 `required` 成员。

6.9 可空特性

有的时候，我们可以不禁用可空引用类型或可空警告。相反，更合适的做法是告诉编译器你的可空意图。为此，可以使用一种称为**特性**（attribute）的构造（参见第 18 章）直接将元数据嵌入代码。`System.Diagnostics.CodeAnalysis` 命名空间定义了 7 种不同的可空特性，并分别标识为前置条件或后置条件，如表 6.1 所示。

表 6.1 可空特性

特性	类别	描述
AllowNull	前置条件	非可空的输入实参可能为 null。
DisallowNull	前置条件	可空的输入实参永远不应为 null。
MaybeNull	后置条件	非空返回值可能为 null。
NotNull	后置条件	可空返回值永远不应为 null。
MaybeNullWhen	后置条件	当方法返回指定的 bool 值时，非空的输入实参可能为 null。
NotNullWhen	后置条件	当方法返回指定的 bool 值时，可空的输入实参不为 null。
NotNullIfNotNull	后置条件	如果为指定形参提供的实参不为 null，那么返回值也不为 null。

这些特性在某些时候非常有用，因为数据类型的可空性偶尔是不足的。为此，可以用特性来修饰方法的传入数据（某个前置条件的可空特性）或传出数据（某个后置条件的可空特性），从而克服这种不足。前置条件告诉调用者指定的值是否可以为 null，而后置条件告诉调用者传出的数据应具有什么可空性（可空还是不可空）。代码清单 6.41 用遵循 try-get 模式的方法对此进行了演示。

代码清单 6.41　使用 NotNullWhen 和 NotNullIfNotNull 特性

```
using System.Diagnostics.CodeAnalysis;
// ...
public class NullabilityAttributesExamined
{
    // ...
    public static bool TryGetDigitAsText(
        char number, [NotNullWhen(true)] out string? text) =>
```

```
            (text = number switch
            {
                '1' => "一",
                '2' => "二",
                '3' => "三",
                '4' => "四",
                // ...
                '9' => "九",
                _ => null
            }) is not null;

    // 在 C# 11/.NET 7.0 之前，不支持为参数使用 nameof()
    [return: NotNullIfNotNull(nameof(text))]
    public static string? TryGetDigitsAsText(string? text)
    {
        if (text == null) return null;

        string result = "";
        foreach (char character in text)
        {
            if (TryGetDigitAsText(character, out string? digitText))
            {
                if (result != "") result += '-';
                result += digitText.ToLower();
            }
        }
        return result;
    }
}
```

注意，TryGetDigitAsText()对 digitText.ToLower()的调用没有使用空合并操作符，而且即使 text 声明为可空，也不会发出警告。这是因为 TryGetDigitAsText()中的 text 参数用 NotNullWhen(true)特性进行了修饰。该特性告诉编译器，如果该方法返回 true（与 NotNullWhen 特性指定的值相符），那么你的意图是 digitText 不会为 null。NotNullWhen 特性是一个“后置条件”声明，它告诉调用者，如果该方法返回 true，那么输出（text）不为 null。

类似地，对于 TryGetDigitsAsText()方法[1]，如果为 text 参数指定的值不为 null，那么返回值也不会为 null。这是因为作为前置条件的可空特性 NotNullIfNotNull 根据 text 参数的输入值是否为 null 来判断返回值是否可能为 null。

[1] 译注：注意方法名多了一个 s。

高级主题：用可空特性修饰泛型类型的参数

声明泛型成员或类型时，我们偶尔希望使用可空修饰符来修饰类型参数。但问题在于，可空值类型（Nullable<T>）与可空引用类型是不同的数据类型。因此，修饰了可空性的类型参数需要一个约束，将类型参数限制为值类型或引用类型。如果没有这个约束，那么会报告以下错误：

```
Error CS8627：可为 null 的类型参数必须已知为值类型或不可为 null 的引用类型。考虑添加一个'class'，'struct'或类型约束。
```

但是，如果无论值类型还是引用类型，它们的逻辑都一样，那么被迫实现两个不同的方法显得太不合常理——特别是考虑到即便使用了不同的约束，也不会导致不同的签名，所以还是无法重载。来看看代码清单 6.42 的例子。

代码清单 6.42　为可能为 null 的返回值使用 MaybeNull 特性

```
// ...
[return: MaybeNull]
public static T GetObject<T>(
  IEnumerable<T> sequence, Func<T, bool> match)
=>
// ...
```

假设我们的意图是返回集合中与谓词匹配的项。如果不存在这样的项，就返回 default(T)——这对于引用类型来说就是 null。遗憾的是，编译器不允许没有约束的 T?。为了避免警告，同时仍然向调用者声明返回值可能为空，我们可以使用作为后置条件的 MaybeNull 特性，同时仍然将返回类型写成 T（没有可空修饰符?）。

6.10 解构函数

使用构造函数，我们可以获取多个参数，并把它们全部封装到一个对象中。但到目前为止，我们一直没有介绍任何语言构造来做相反的事情，即把封装好的数据拆分为它的各个组成部分。当然，完全可以将每个属性手动赋给变量。但是，如果有太多这样的变量，就需要大量单独的语句。自 C# 7.0 推出元组语法后，该操作得到了极大简化。如代码清单 6.43 所示，可以声明一个像 Deconstruct()这样的方法来做这件事情。

代码清单 6.43　定义和使用解构函数

```
public class Employee
{
    // ...
    public void Deconstruct(
```

```
            out int id, out string firstName,
            out string lastName, out string? salary)
        {
            (id, firstName, lastName, salary) =
                (Id, FirstName, LastName, Salary);
        }
        // ...
    }

    public class Program
    {
        public static void Main()
        {
            Employee employee;
            employee = new("Inigo", "Montoya")
            {
                // 利用对象初始化器语法
                Salary = "太少了"
            };
            // ...

            employee.Deconstruct(out _, out string firstName,
                out string lastName, out string? salary);

            System.Console.WriteLine(
                "{0} {1}: {2}",
                firstName, lastName, salary);
        }
    }
```

如第 5 章所述，可以直接调用这个方法，调用前以内联形式声明 out 参数即可。但是，从 C# 7.0 开始，还可以直接将对象实例赋给一个元组，从而隐式调用 Deconstruct()方法（称为**解构函数**）[①]，如代码清单 6.44 所示。

代码清单 6.44　隐式调用解构函数

```
    public class Program
    {
        public static void Main()
```

[①] 译注：注意区分 deconstructor 和 destructor，前者是 C#新增的解构函数，后者是 C++等传统语言的析构器（析构函数）。

```
    {
        Employee employee;
        employee = new("Inigo", "Montoya")
        {
            // 利用对象初始化器语法
            Salary = "太少了"
        };

        // ...

        (_, string firstName, string lastName, string? salary) = employee;

        System.Console.WriteLine(
            "{0} {1}: {2}",
            firstName, lastName, salary);
    }
}
```

该语法生成的 CIL 代码和图 6.43 完全一样，只是更简单（而且更让人注意不到调用了 Deconstruct()方法）。注意，只允许用元组语法向那些和 out 参数匹配的变量赋值。不允许向元组类型的变量赋值，例如：

```
(int, string, string, string) tuple = employee;
```

也不允许向元组中的具名项赋值：

```
(int id, string firstName, string lastName, string salary) tuple = employee
```

为了声明解构函数，方法名必须是 Deconstruct，其签名是返回 void 并接收两个或更多 out 参数。基于该签名，可以将对象实例直接赋给一个元组而无需显式方法调用。

要注意的是，在 C# 10 之前，元组中的变量必须是新声明或者事先声明好的，新声明的变量和现有的变量不能混用。从 C# 10 开始，这一限制已经取消了。

6.11 静态成员

static 关键字在第 1 章的 HelloWorld 例子中简单接触过。本节将完整定义 static。

先考虑一个例子。假定每个员工 Id 值都不能重复，一个解决方案是通过计数器来跟踪每个员工 ID。但是，如果值作为实例字段存储，每次实例化对象都要创建一个新的 NextId 字段，造成每个 Employee 对象实例都要为那个字段分配内存。而且最大的问题在于，每次实例化 Employee 对象，以前实例化的所有 Employee 对象的 NextId 值都需要更新为下一个（可分配的）ID 值。在这种情况下，我们真正需要的是可由所有 Employee 对象实例共享的一个字段。

> 语言对比：C++——全局变量和函数
>
> 和以前的许多语言不同，C#没有全局变量或全局函数。C#的所有字段和方法都在类的上下文中。在 C#中，与全局字段或函数等价的是静态字段或方法。“全局变量/函数”和“C#静态字段/方法”没有功能性上的差异，只是静态字段/方法可以包含访问修饰符（比如 `private`），从而限制访问并提供更好的封装。

6.11.1 静态字段

我们使用 `static` 关键字来定义能由多个实例共享的数据，如代码清单 6.45 所示。

代码清单 6.45　声明静态字段

```
public class Employee
{
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
        Id = NextId;
        NextId++;
    }

    // ...

    public static int NextId;
    public int Id { get; private set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? Salary { get; set; } = "不够";

    // ...
}
```

本例用 `static` 修饰符将 `NextId` 字段声明为**静态字段**。和 `Id` 不同，所有 `Employee` 实例都共享同一个 `NextId` 存储位置。`Employee` 构造函数先将 `NextId` 的值赋给新 `Employee` 对象的 `Id`，然后立即递增 `NextId`。创建另一个 `Employee` 对象时，由于 `NextId` 的值已递增，所以新 `Employee` 对象的 `Id` 字段将获得不同的值。

和**实例字段**（非静态字段）一样，静态字段也可以在声明时初始化，如代码清单 6.46 所示。

代码清单 6.46　声明时向静态字段赋值

```
public class Employee
{
    // ...
    public static int NextId = 42;
    // ...
}
```

和实例字段不同，未初始化的静态字段将获得默认值（0、null、false 等，即 default(T) 的结果，其中 T 是类型名）。所以，没有显式赋值的静态字段也是可以访问的。

每新建一个对象实例，非静态字段（实例字段）都要占用一个新的存储位置。静态字段从属于类而非实例。因此，是使用类名从类外部访问静态字段。代码清单 6.47 展示了一个新的 Program 类（使用代码清单 6.45 的 Employee 类），结果如输出 6.8 所示。

代码清单 6.47　访问静态字段

```
using System;

public class Program
{
    public static void Main()
    {
        Employee.NextId = 1000000;

        Employee employee1 = new(
            "Inigo", "Montoya");
        Employee employee2 = new(
            "Princess", "Buttercup");

        Console.WriteLine(
            "{0} {1} ({2})",
            employee1.FirstName,
            employee1.LastName,
            employee1.Id);
        Console.WriteLine(
            "{0} {1} ({2})",
            employee2.FirstName,
            employee2.LastName,
            employee2.Id);

        Console.WriteLine(
            $"NextId = {Employee.NextId}");
    }
    // ...
}
```

输出 6.8

```
Inigo Montoya (1000000)
Princess Buttercup (1000001)
NextId = 1000002
```

注意，我们是通过类名 Employee 来设置和获取静态字段 NextId 的初始值，而不是通过对类的实例的引用。只有在类（或派生类）内部的代码中才能省略类名。换言之，Employee(...)构造函数不需要使用 Employee.NextId。这些代码已经在 Employee 类的上下文中，所以不需要专门指出上下文。变量的作用域是可以不加限定来引用它的程序代码区域，而静态字段的作用域是类（及其任何派生类）。

虽然引用静态字段的方式与引用实例字段的方式稍有区别，但不能在同一个类中定义同名的静态字段和实例字段。引用错误字段的几率会很高，C#的设计者决定禁止这样的代码。所以，重复的名称在声明空间中会造成编译错误。

> **初学者主题：类和对象都能关联数据**
>
> 类和对象都能关联数据。将类想象成模具，将对象想象成根据该模具浇铸的零件，可以更好地理解这一点。
>
> 例如，一个模具拥有的数据可能包括：到目前为止已用模具浇铸的零件数、下个零件的序列号、当前注入模具的液态塑料的颜色以及模具每小时生产的零件数量。类似地，零件也拥有它自己的数据：序列号、颜色以及生产日期/时间。虽然零件颜色就是生产零件时在模具中注入的塑料的颜色，但零件显然不包含模具中当前注入的塑料颜色数据，也不包含要生产的下个零件的序列号数据。
>
> 设计对象时，程序员要考虑字段和方法应声明为静态还是基于实例。一般应将不需要访问任何实例数据的方法声明为静态方法。静态字段主要存储对应于类的数据，比如新实例的默认值或者已创建实例个数。而实例字段主要存储和对象关联的数据。

6.11.2 静态方法

和静态字段一样，直接在类名后访问静态方法（比如 Console.ReadLine()）。访问这种方法不需要有实例。代码清单 6.48 展示了一个声明和调用静态方法的例子。

代码清单 6.48　为 DirectoryInfoExtension 类定义静态方法

```
public static class DirectoryInfoExtension
{
    public static void CopyTo(
```

```
        DirectoryInfo sourceDirectory, string target,
        SearchOption option, string searchPattern)
    {
        if (target[^1] != Path.DirectorySeparatorChar)
        {
            target += Path.DirectorySeparatorChar;
        }
        Directory.CreateDirectory(target);
        for (int i = 0; i < searchPattern.Length; i++)
        {
            foreach (string file in
                Directory.EnumerateFiles(
                    sourceDirectory.FullName, searchPattern))
            {
                File.Copy(file,
                    target + Path.GetFileName(file), true);
            }
        }

        // 复制子目录(以递归方式)
        if (option == SearchOption.AllDirectories)
        {
            foreach (string element in
                Directory.EnumerateDirectories(
                    sourceDirectory.FullName))
            {
                Copy(element,
                    target + Path.GetFileName(element),
                    searchPattern);
            }
        }
    }
    // ...
}

public class Program
{
    public static void Main(params string[] args)
    {
        DirectoryInfo source = new(args[0]);
        string target = args[1];

        DirectoryInfoExtension.CopyTo(
            source, target,
            SearchOption.AllDirectories, "*");
    }
}
```

在代码清单 6.48 中，DirectoryInfoExtension.CopyTo()方法获取一个 DirectoryInfo 对象，将基础目录结构复制到新位置。

由于静态方法不通过实例引用，所以 this 关键字在静态方法中无效。此外，要在静态方法内部直接访问实例字段或实例方法，必须先获得对字段或方法所属的那个实例的引用。（注意，Main()是静态方法的另一个例子。）

该方法本应由 System.IO.Directory 类提供，或作为 System.IO.DirectoryInfo 类的实例方法提供。但两个类都没提供，所以代码清单 6.48 在一个全新的类中定义该方法。本章后面讲述扩展方法的小节会解释如何使它表现为 DirectoryInfo 类的实例方法。

6.11.3 静态构造函数

除了静态字段和方法，C#还支持**静态构造函数**，用于对类（而不是类的实例）进行初始化。静态构造函数不显式调用；相反，“运行时”在首次访问类时自动调用静态构造函数。“首次访问类”可能发生在调用普通构造函数时，也可能发生在访问类的静态方法或字段时。由于静态构造函数不能显式调用，所以不允许任何参数。

静态构造函数的作用是将类中的静态数据初始化成特定值，尤其是在无法通过声明时的一次简单赋值来获得初始值的时候。代码清单 6.49 展示了一个例子。

代码清单 6.49　声明静态构造函数

```
public class Employee
{
    static Employee()
    {
        Random randomGenerator = new();
        NextId = randomGenerator.Next(101, 999);
    }

    // ...
    public static int NextId = 42;
    // ...
}
```

本例将 NextId 的初始值设为 101～998 的随机整数。由于初始值涉及方法调用，所以 NextId 的初始化代码被放到一个静态构造函数中，而没有作为声明的一部分。

如本例所示，假如对 NextId 的赋值既在静态构造函数中进行，又在声明时进行，那么当初始化结束时，最终获得什么值？观察 C#编译器生成的 CIL 代码，会发现声明时的赋值

被移动了位置，成为静态构造函数中的第一个语句。因此，NextId 最终包含由 randomGenerator.Next(101, 999)生成的随机数，而不是声明 NextId 时所赋的值。结论是静态构造函数中的赋值优先于声明时的赋值，这和实例字段的情况一样。

注意没有“静态终结器”的说法。还要注意不要在静态构造函数中抛出异常，否则会造成类型在应用程序[①]剩下的生存期内无法使用。

> **高级主题：最好在声明时进行静态初始化（而不要使用静态构造函数）**
>
> 静态构造函数在首次访问类的任何成员之前执行，无论该成员是静态字段，是其他静态成员，还是实例构造函数。为支持这个设计，编译器添加代码来检查类型的所有静态成员和构造函数，确保首先运行静态构造函数。
>
> 如果没有静态构造函数，编译器会将所有静态成员初始化为它们的默认值，而且不会添加对静态构造函数的检查。结果是静态字段会在访问前得到初始化，但不一定在调用静态方法或任何实例构造函数之前。有的时候，对静态成员进行初始化的代价比较高，而且访问前确实没必要初始化，所以这个设计能带来一定的性能提升。有鉴于此，请考虑要么以内联方式初始化静态字段（而不要使用静态构造函数），要么在声明时初始化。[②]

设计规范

CONSIDER either initializing static fields inline rather than using a static constructor or initializing them at declaration time.

考虑要么以内联方式初始化静态字段（而不要使用静态构造函数），要么在声明时初始化。

6.11.4 静态属性

属性也可以声明为 static。代码清单 6.50 将 NextId 数据包装成属性。

[①] 更准确的说法是“应用程序域”（AppDomain），即“操作系统进程”在 CLR 中的虚拟等价物。

[②] 译注：*http://tinyurl.com/3ec3z599* 展示了使用内联初始化来代替静态构造函数的一个例子。

代码清单 6.50　声明静态属性

```
public class Employee
{
    // ...
    public static int NextId
    {
        get
        {
            return _NextId;
        }
        private set
        {
            _NextId = value;
        }
    }
    public static int _NextId = 42;
    // ...
}
```

相较于使用公共静态字段，使用静态属性几乎肯定会更好，因为公共静态字段在任何地方都能调用，而静态属性至少提供了一定程度的封装。

从 C# 6.0 开始，整个 NextId 实现（含不可访问的支持字段）都可以简化为带一个初始化器的自动实现属性。

```
public static int NextId { get; private set; } = 42;
```

6.11.5 静态类

有的类不含任何实例字段。例如，假定 SimpleMath 类包含与数学运算 Max()和 Min()对应的函数，如代码清单 6.51 所示。

代码清单 6.51　声明静态类

```
using static SimpleMath;

public static class SimpleMath
{
    // params 支持可变数量的参数
    public static int Max(params int[] numbers)
    {
        // 检查 numbers 数组中是否至少有一项
        if(numbers.Length == 0)
        {
            throw new ArgumentException(
```

```
                "numbers 不能空白", nameof(numbers));
        }

        int result;
        result = numbers[0];
        foreach(int number in numbers)
        {
            if(number > result)
            {
                result = number;
            }
        }
        return result;
    }

    // params 支持可变数量的参数
    public static int Min(params int[] numbers)
    {
        // 检查 numbers 数组中是否至少有一项
        if (numbers.Length == 0)
        {
            throw new ArgumentException(
                "numbers 不能空白", nameof(numbers));
        }

        int result;
        result = numbers[0];
        foreach(int number in numbers)
        {
            if(number < result)
            {
                result = number;
            }
        }
        return result;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        int[] numbers = new int[args.Length];
        for (int index = 0; index < args.Length; index++)
        {
            numbers[index] = args[index].Length;
        }

        Console.WriteLine(
            $@"最长的实参长度 = {
```

```
                Max(numbers) }");

            Console.WriteLine(
                $@"最短的实参长度 = {
                    Min(numbers) }");
        }
    }
```

该类不包含任何实例字段（或方法），创建能实例化的类没有意义。所以用 static 关键字修饰该类。声明类时使用 static 关键字有两方面的意义。首先，它防止程序员写代码来实例化 SimpleMath 类。其次，防止在类的内部声明任何实例字段或方法。既然类无法实例化，实例成员自然也就没了意义。在代码清单 6.51 中，Program 类其实也应设计成静态类，因为它只包含静态成员。

静态类的另一个特点是，C#编译器会自动在 CIL 代码中把它标记为 abstract 和 sealed。这会将类指定为**不可扩展**；换言之，不能从它派生出其他类，甚至不能实例化它。（sealed 和 abstract 关键字的详情将在第 7 章介绍。）

第 5 章说过，可以为 SimpleMath 这样的静态类使用 using static 指令。例如，由于在代码清单 6.51 顶部添加了 using static SimpleMath;指令，所以可以在不添加 SimpleMath 前缀的前提下调用 Max。

```
Console.WriteLine(
    $@"最长的实参长度 = { Max(numbers) }");
```

6.12 扩展方法

来考虑用于处理文件系统目录的 System.IO.DirectoryInfo 类。类支持的功能包括列出文件和子目录（DirectoryInfo.GetFiles()）以及移动目录（DirectoryInfo.Move()）。但它不直接支持复制功能。需要这样的方法得自己实现，如本章前面的代码清单 6.48 所示。

当初声明的 DirectoryInfoExtension.CopyTo()是标准静态方法。但 是，CopyTo()方法在调用方式上有别于 DirectoryInfo.Move()。这是令人遗憾的一个设计。理想情况是为 DirectoryInfo 添加一个方法，能在获得一个实例的情况下将 CopyTo()作为实例方法来调用，例如 directory.CopyTo()。

C# 3.0 引入了**扩展方法**的概念，能在一个类中模拟为另一个不同的类创建实例方法，从而实现对后者的扩展。只需更改静态方法的签名，使第一个参数成为要扩展的类型，并在类型名称前附加 this 关键字即可，如代码清单 6.52 所示。

代码清单 6.52　在 DirectoryInfoExtension 类中定义静态 CopyTo()方法来扩展

DirectoryInfo 类

```
public static class DirectoryInfoExtension
{
    public static void CopyTo(
        this DirectoryInfo sourceDirectory, string target,
        SearchOption option, string searchPattern)
    {
        // ...
    }
}

// ...
        DirectoryInfo directory = new(".\\Source");
        directory.CopyTo(".\\Target",
            SearchOption.TopDirectoryOnly, "*");
        // ...
    }
}
```

该设计允许为任何类添加“实例方法”，即使那些不在同一个程序集中的类。但是，如果查看最终生成的 CIL 代码，会发现扩展方法是作为普通静态方法调用的。下面列出了扩展方法的要求。

- 第一个参数是要扩展或者要操作的类型，称为“被扩展类型”。
- 为了指定扩展方法，要在被扩展的类型名称前附加 this 修饰符。
- 为了将方法作为扩展方法来访问，要用 using 指令导入扩展类型[①]的命名空间，或者将扩展类型和调用代码放在同一命名空间。

如果扩展方法的签名和被扩展类型中现有的签名匹配（换言之，假如 DirectoryInfo 已经有一个 CopyTo()方法了），那么扩展方法永远得不到调用，除非是作为普通静态方法。

注意，通过继承（将于第 7 章讲述）来特化类型要优于使用扩展方法。扩展方法无益于建立清楚的版本控制机制，因为一旦在被扩展类型中添加匹配的签名，就会覆盖现有扩展方法，而且不会发出任何警告。如果对被扩展的类的源代码没有控制权，该问题将变得更加突出。另一个问题是，虽然 Visual Studio 的“智能感知”支持扩展方法，但假如只是查看调用代码（也就是调用了扩展方法的代码），是不易看出一个方法是不是扩展方法的。

总之，扩展方法要慎用。例如，不要为 object 类型定义扩展方法。第 8 章将讨论扩展方法如何与接口配合使用。很少在没有这种配合的前提下定义扩展方法。

[①] 译注：即对“被扩展的类型”进行扩展的那个类型，或者说声明扩展方法的那个类型。

设计规范

AVOID frivolously defining extension methods, especially on types you don't own.

避免随便定义扩展方法，尤其是不要为自己没有所有权的类型定义。

6.13 封装数据

除了本章前面讨论的属性和访问修饰符，还有其他几种特殊方式可将数据封装到类中。例如，还有另外两个字段修饰符：const（声明局部变量时见过）和 readonly。

6.13.1 const

和 const 值一样，const 字段（称为常量字段）包含在编译时确定的值，运行时不可修改。像 π 这样的值就很适合声明为常量字段。代码清单 6.5 3 展示了如何声明常量字段。

代码清单 6. 53　声明常量字段

```
// 单位换算类
public class ConvertUnits
{
    public const float CentimetersPerInch = 2.54F; // 1 英寸多少厘米
    public const int CupsPerGallon = 16;  // 一加仑多少杯
    // ...
}
```

常量字段自动成为静态字段，因为不需要为每个对象实例都生成新的字段实例。但是，主动将常量字段声明为 static 会造成编译错误。另外，常量字段通常只声明为有字面值的类型（string，int 和 double 等）。Program 或 System.Guid 等类型则不能用于常量字段。

在 public 常量表达式中，必须使用随着时间的推移不会发生变化的值。圆周率、阿伏加德罗常数和赤道长度都是很好的例子。以后可能发生变化的值就不合适。例如，版本号、人口数量和汇率都不适合作为常量。

设计规范

DO use constant fields for values that will never change.

要为恒定不变的值使用常量字段。

AVOID constant fields for values that will change over time.

避免为将来会发生变化的值使用常量字段。

> **高级主题：public 常量应该是恒定值**
>
> public 常量应恒定不变，因为如果修改它，在使用它的程序集中不一定能反映出最新改变。如果一个程序集引用了另一个程序集中的常量，常量值将直接编译到引用程序集中。所以，如果被引用程序集中的常量值发生改变，而引用程序集没有重新编译，那么引用程序集将继续使用原始值而非新值。将来可能改变的值应指定为 readonly，不要指定为常量。

6.13.2 readonly

和 const 不同，readonly 修饰符只能用于字段（不能用于局部变量），它指出字段值只能从构造函数中更改，或在声明时通过初始化器（初始值设定项）指定。代码清单 6.54 演示如何声明 readonly 字段。

代码清单 6.54　声明 readonly 字段

```
public class Employee
{
    public Employee(int id)
    {
        _Id = id;
    }

    // ...

    private readonly int _Id;
    public int Id
    {
        get { return _Id; }

    }
    // 错误：不能向只读字段赋值(除非通过构造函数或者变量初始化器)
    public void SetId(int id) => _Id = id;
    // ...
}
```

和 const 字段不同，每个实例的 readonly 字段值都可以不同。事实上，readonly 字段的值可以在构造函数中更改。此外，readonly 字段可以是实例或静态字段。另一个关键区别是，可以在执行时为 readonly 字段赋值，而非只能在编译时。编译器一般不要求从包装了字段的属性外部访问该字段，但 readonly 字段是一个例外，因为它必须通过构造函数或初始化器来设置。但除了这个例外情况，一般不应从属性外部访问属性的支持字段。

和 const 字段相比，readonly 字段的另一个重要特点是不限于有字面值的类型。例如，可以声明 readonly System.Guid 实例字段：

```
public static readonly Guid ComIUnknownGuid =
    new Guid("00000000-0000-0000-C000-000000000046");
```

声明为常量则不行，因为没有 GUID 的 C#字面值形式。

由于设计规范要求字段不要从其包容属性外部访问，所以从 C# 6.0 开始，readonly 修饰符几乎完全没了用武之地。相反，无脑选择本章前面讨论的只读自动实现属性就可以了，如代码清单 6.55 所示。

代码清单 6.55　声明只读自动实现的属性

```
class TicTacToeBoard
{
    // 将两个玩家的初始棋盘设为全 false (空白)
    //     |   |           |   |
    // ---+---+---     ---+---+---
    //     |   |           |   |
    // ---+---+---     ---+---+---
    //     |   |           |   |
    // 玩家 1 - X       玩家 2 - O
    public bool[,,] Cells { get; } = new bool[2, 3, 3];
    // 错误：不能向 Cells 属性赋值，因为它是只读的
    // public void SetCells(bool[,,] value) =>
    //         _Cells = new bool[2, 3, 3];

    // ...
}
```

无论使用 C# 6.0 只读自动实现属性，还是使用 readonly 字段，确保数组引用的“不可变”性质都是一项有用的防御性编程技术。它确保数组实例保持不变，同时允许修改数组中的元素。如果不施加只读限制，那么很容易就会误将新数组赋给成员，这样会丢弃现有数组而不是更新其中的数组元素。换言之，向数组施加只读限制，不会冻结数组的内容。相反，它只是冻结数组实例（以及数组中的元素数量），因为不可能重新赋值来指向一个新的数组实例。数组中的元素仍然可写。

设计规范

DO favor read-only automatically implemented properties over read-only fields.

要优先选择只读自动实现的属性而不是只读字段。

6.14 嵌套类

在类中除了定义方法和字段，还可以定义另一个类。这称为**嵌套类**。假如一个类在它的包容类外部没有多大意义，就适合把它设计成嵌套类。

假设有一个类用于处理程序的命令行选项。通常，像这样的类在每个程序中的处理方式都是不同的，没有理由使 CommandLine 类能够从包含 Main()的那个类的外部访问。代码清单 6.56 演示了这样的一个嵌套类。

代码清单 6.56　定义嵌套类

```
// CommandLine 类嵌套在 Program 类中

public class Program
{
    // 定义一个嵌套类来专门处理命令行
    private class CommandLine
    {
        public CommandLine(string[] arguments)
        {
            for (int argumentCounter = 0;
                argumentCounter < arguments.Length;
                argumentCounter++)
            {
                _ = argumentCounter switch
                {
                    0 => Action = arguments[0].ToLower(),
                    1 => Id = arguments[1],
                    2 => FirstName = arguments[2],
                    3 => LastName = arguments[3],
                    _ => throw new ArgumentException(
                        $"非预期的参数" +
                        $"'{arguments[argumentCounter]}'")
                };
            }
        }
```

```
        public string? Action { get; }
        public string? Id { get; }
        public string? FirstName { get; }
        public string? LastName { get; }
    }

    public static void Main(string[] args)
    {
        CommandLine commandLine = new(args);

        // 为避免分心，这里故意省略了错误处理

        switch (commandLine.Action)
        {
            case "new":
                // 新建一个员工
                // ...
                break;
            case "update":
                // 更新现有员工的数据
                // ...
                break;
            case "delete":
                // 删除现有员工的文件
                // ...
                break;
            default:
                Console.WriteLine(
                    "Employee.exe " +
                    "new|update|delete " +
                    "<id> [名字] [姓氏]");
                break;
        }
    }
}
```

本例的嵌套类是 Program.CommandLine。和其他所有类成员一样，包容类内部没有必要使用包容类名称前缀，直接把它引用为 CommandLine 就好。

嵌套类的独特之处是可以为类自身指定 private 访问修饰符。由于类的作用是解析命令行，并将每个实参放到单独字段中，所以它在该应用程序中只和 Program 类有关系。使用 private 访问修饰符可以限定类的作用域，防止从类的外部访问。只有嵌套类才能这样做。

嵌套类中的 this 成员引用嵌套类而非包容类的实例。嵌套类要访问包容类的实例，一个方案是显式传递包容类的实例，比如通过构造函数或方法参数。

嵌套类另一个有趣的地方在于它能访问包容类的任何成员，其中包括私有成员。反之则不然，包容类不能访问嵌套类的私有成员。

嵌套类用得很少。要从包容类型外部引用，就不能定义成嵌套类。另外要警惕 `public` 嵌套类；它们意味着不良的编码风格，可能造成混淆和难以阅读。

设计规范

AVOID publicly exposed nested types. The only exception is if the declaration of such a type is unlikely or pertains to an advanced customization scenario.

避免声明公共嵌套类型。少数高级自定义场景才需考虑。

> **语言对比：Java——内部类**
>
> Java 不仅支持嵌套类，还支持内部类（inner class）。内部类对应和包容类实例关联的对象，而非仅仅和包容类有语法上的包容关系。C#允许在外部类中包含嵌套类型的一个实例字段，从而获得相同的结构。一个工厂方法或构造函数可以确保在内部类的实例中设置对外部类的相应实例的引用。

6.15 分部类

从 C# 2.0 起支持**分部类**（partial classes）。分部类是一个类的多个部分，编译器可把它们合并成一个完整的类。虽然可以在同一个文件中定义两个或更多分部类，但分部类的目的就是将一个类的定义划分到多个文件中。这对生成或修改代码的工具尤其有用。通过分部类，由工具处理的文件可独立于开发者手动编码的文件。

6.15.1 定义分部类

C# 允许在 `class` 前添加上下文关键字 `partial` 来声明分部类[①]，如代码清单 6.57 所示。

[①] 接口（第 8 章）和结构（第 9 章）也可以分部。

代码清单 6.57　定义分部类

```
// 文件: Program1.cs
partial class Program
{
}
// 文件: Program2.cs
partial class Program
{
}
```

本例将 Program 的每个部分都放到单独文件中（参见注释）。除了用于代码生成器，分部类另一个常见的应用是将每个嵌套类都放到它们自己的文件中。这符合编码规范“将每个类定义都放到它自己的文件中”。例如，代码清单 6.5 8 将 Program.CommandLine 类放到和核心 Program 成员分开的一个文件中。

代码清单 6.58　用分部类定义嵌套类

```
// 文件: Program.cs
partial class Program
{
    static void Main(string[] args)
    {
        CommandLine commandLine = new(args);
        switch (commandLine.Action)
        {
            // ...
        }
    }
}
// 文件: Program+CommandLine.cs
partial class Program
{
    // 定义一个嵌套类来处理命令行
    private class CommandLine
    {
        // ...
    }
}
```

不允许用分部类扩展编译好的类或其他程序集中的类。只能利用分部类在同一个程序集中将一个类的实现拆分成多个文件。

6.15.2 分部方法

C# 3.0 引入了分部方法的概念，对分部类进行了扩展。分部方法只能存在于分部类中，而

且和分部类相似，主要作用方便代码的自动生成。

假定代码生成工具能根据数据库中的 Person 表为 Person 类生成对应的 Person.Designer.cs 文件。该工具检查表并为表中每一列创建属性。但问题在于，工具经常都不能生成必要的验证逻辑，因为这些逻辑依赖于未在表定义中嵌入的业务规则。所以，Person 类的开发人员需要自行添加验证逻辑。Personal.Designer.cs 文件是不好直接修改的，因为假如文件被重新生成（例如，为了适应数据库中新增的一个列），所做的更改就会丢失。相反，Person 类的代码结构应独立出来，使生成的代码在一个文件中，自定义代码（含业务规则）在另一个文件中，后者不受任何重新生成动作的影响。如上一节所示，分部类很适合将一个类拆分为多个文件。但这样可能还不够。经常还需要**分部方法**。

分部方法允许声明方法而不需要实现。但是，如果包含了可选的实现，该实现就可放到某个姊妹分部类定义中——可能在单独的文件中。代码清单 6.5 9 展示了如何为 Person 类声明和实现分部方法。

代码清单 6. 59　为 Person 类声明和实现分部方法

```
// 文件: Person.Designer.cs
public partial class Person
{
    static partial void OnLastNameChanging(string value);
    static partial void OnFirstNameChanging(string value);

    // ...
    public string LastName
    {
        get
        {
            return _LastName;
        }
        set
        {
            if (_LastName != value)
            {
                OnLastNameChanging(value);
                _LastName = value;
            }
        }
    }
    private string _LastName;

    // ...
    public string FirstName
    {
        get
```

```
        {
            return _FirstName;
        }
        set
        {
            if (_FirstName != value)
            {
                OnFirstNameChanging(value);
                _FirstName = value;
            }
        }
    }
    private string _FirstName;

    public partial string GetName();
}

// 文件: Person.cs
partial class Person
{
    static partial void OnLastNameChanging(string value)
    {
        value = value ??
            throw new ArgumentNullException(nameof(value));

        if (value.Trim().Length == 0)
        {
            throw new ArgumentException(
                $"{nameof(LastName)}不能空白。",
                nameof(value));
        }
    }

    // ...

    public partial string GetName() => $"{FirstName} {LastName}";
}
```

Person.Designer.cs 文件包含 OnLastNameChanging()和 OnFirstNameChanging()方法的声明。此外，LastName 和 FirstName 属性调用了它们对应的 *Changing* 方法。虽然这两个方法只有声明而没有实现，但却能成功通过编译。关键在于方法声明附加了上下文关键字 partial，其所在的类也是一个 partial 类。

代码清单 6.59 只展示了 OnLastNameChanging()方法的实现。该实现会检查传递的新 LastName 值，无效就抛出异常。注意两个地方的 OnLastNameChanging()签名是匹配的。

在 C# 9.0 之前，分部方法必须返回 void。如果方法不返回 void 又没有提供实现，那么调用一个未实现的方法返回什么才算合理？为避免对返回值进行任何无端的猜测，C#的设计者决定只允许方法返回 void。类似地，out 参数在分部方法中不允许。需要返回值可以使用 ref 参数。最后，不能为分部方法附加一个可访问性修饰符（例如，private 或 public）。相反，分部方法隐式为 private。

但从 C# 9.0 开始，所有这些限制都被移除了。分部方法可以返回值，可以有 out 参数，甚至可以有访问修饰符。事实上，为了区分这个无限制版本的分部方法，C#要求明确使用访问修饰符，即使本来就打算私有的方法也不例外。为了移除限制，C# 9.0 要求所有带有访问修饰符的分部方法还必须有一个配对的实现。如代码清单 6.59 所示，如果声明了一个没有实现的分部方法，即 public string GetName()，那么必须同时用相同的签名来提供一个实现，例如 public partial string GetName() => $"{FirstName} {LastName}"。通过这种方式，C#编译器确保任何返回值（包括通过 out 参数返回的）都能成功，因为该方法真的有一个实现。

总之，分部方法使生成的代码能调用并非一定要实现的方法。此外，如果没有为分部方法提供实现，CIL 中不会出现分部方法的任何踪迹。这样在保持代码尽量小的同时，还保证了高的灵活性。

6.16 小结

本章讲解了 C#类以及面向对象程序设计。讨论了字段，并讨论了如何在类的实例上访问它们。

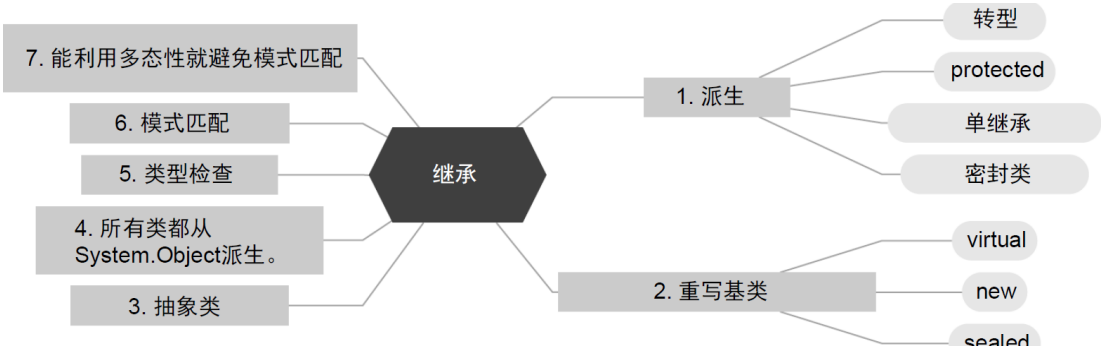
是在每个实例上都存储一份数据，还是为一个类型的所有实例统一存储一份？本章详细讲解了应如何取舍。静态数据与类关联，而实例数据在每个对象上存储。

本章还以方法和数据的访问修饰符为背景探讨了封装问题。介绍了 C#属性，并解释了如何用它封装私有字段。

下一章学习如何通过“继承”关联不同的类，并体会继承对于面向对象编程思维的影响。

第 7 章 继承

第 6 章讨论了一个类如何通过字段和属性来引用其他类。本章讨论如何利用类的继承关系建立类层次结构。



> **初学者主题：继承的定义**
>
> 第 6 章已简单介绍了继承。下面是对已定义的术语的简单回顾。
>
> - *派生/继承*：对基类进行特化，以添加额外成员或自定义基类成员。
> - *派生类型/子类型*：一种特化的类型，继承了较常规类型的成员。
> - 基/超/父类型：其成员由派生类型继承的常规类型。
>
> 继承建立了“属于”（is-a 或 is a kind of）关系。派生类型总是隐式属于基类型。如同硬盘属于存储设备，从存储设备类型派生的其他任何类型都属于存储设备。但是，反之则不成立。存储设备不一定是硬盘。
>
> > **注意**：在编码的时候，继承用于定义两个类的“属于”关系 ，派生类属于基类的一种特化。

7.1 派生

经常需要扩展现有类型来添加功能（行为和数据）。继承正是为了该目的而设计的。例如，假定已经有一个 `Person` 类，我们可以创建 `Employee` 类，并在其中添加 `EmployeeId` 和 `Department` 等员工特有的属性。也可以采取相反的操作。例如，假定已经有一个在 PDA（个人数字助理）中使用的 `Contact` 类，现在想为 PDA 添加行事历支持。为此，我们可以创建 `Appointment` 类。但不是重新定义这两个类都适用的方法和属性，而是对 `Contact` 类进行**重构**。具体地说，将两者都适用的方法和属性从 `Contact` 移至名为 `PdaItem` 的基类

中，并让 Contact 和 Appointment 都从该基类派生。换言之，Contact 和 Appointment 现在都“属于”（is-a）一种 PdaItem，如图 7.1 所示。

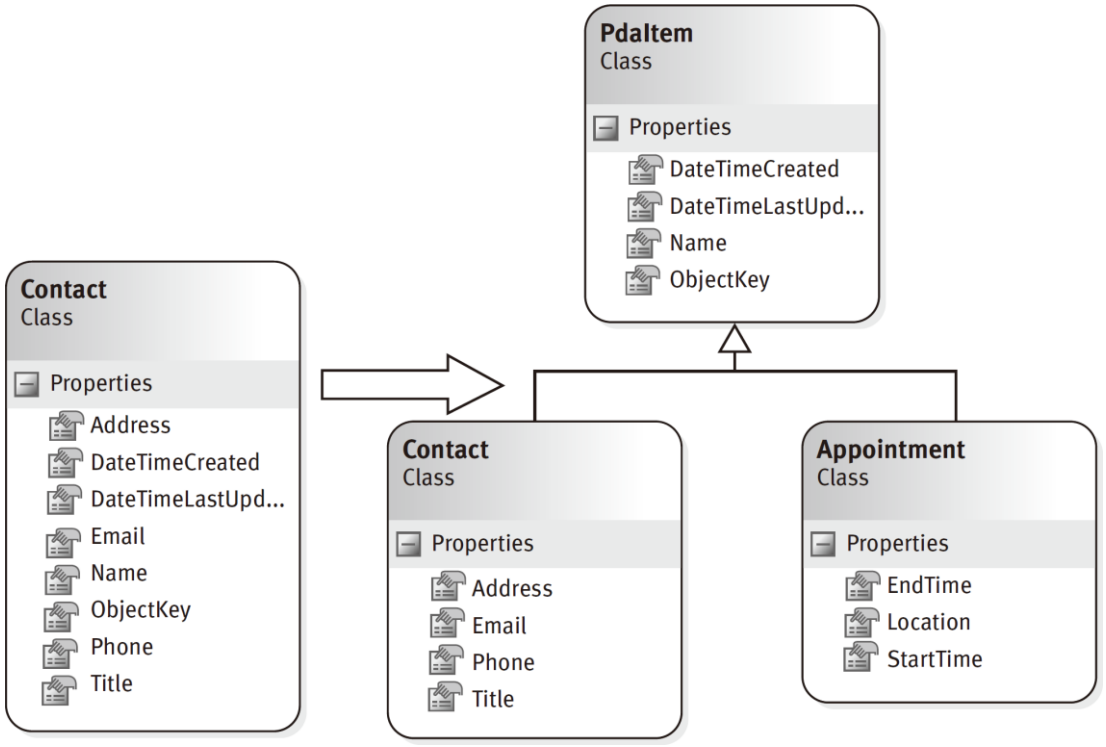


图 7.1 重构为基类

在本例中，两者共用的项是 Created，LastUpdated，Name 和 ObjectKey 等。通过派生，基类 PdaItem 定义的方法可以从 PdaItem 的所有子类中访问。

声明派生类时，要在类标识符后添加一个冒号，接着添加基类名称，如代码清单 7.1 所示。

代码清单 7.1 从一个类派生出另一个类

```
public class PdaItem
{
    [DisallowNull]
    public string? Name { get; set; }
    public DateTime LastUpdated { get; set; }
}
// Contact 类从 PdaItem 类继承
public class Contact : PdaItem
{
```

```
    public string? Address { get; set; }
    public string? Phone { get; set; }

    // ...
}
```

代码清单 7.2 展示了如何访问 Contact 类继承的属性。

代码清单 7.2　使用继承的属性

```
public class Program
{
    public static void Main()
    {
        Contact contact = new();
        contact.Name = "Inigo Montoya";

        // ...
    }
}
```

虽然 Contact 没有直接定义 Name 属性，但 Contact 的所有实例都可以访问来自 PdaItem 的 Name 属性，并把它作为 Contact 的一部分使用。此外，从 Contact 派生的其他任何类也会继承 PdaItem 类（或者 PdaItem 的父类）的成员。该继承链没有限制，每个派生类都拥有由其所有基类公开的所有成员（参见代码清单 7.3）。换言之，虽然 Customer 不直接从 PdaItem 派生，但它依然继承了 PdaItem 的成员。

> **注意：**通过继承，基类的每个成员都出现在派生类构成的链条中。

代码清单 7.3　逐级派生构成了一个继承链

```
public class PdaItem : object
{
    // ...
}
public class Appointment : PdaItem
{
    // ...
```

```
}
public class Contact : PdaItem
{
    // ...
}
public class Customer : Contact
{
    // ...
}
```

在代码清单 7.3 中，我们让 PdaItem 显式地从 object 派生。虽然允许这样写，但完全没必要，因为所有类都隐式派生自 object。

> **注意**：除非明确指定了其他基类，否则所有类都默认从 object 派生。

7.1.1 基类型和派生类型之间的转型

如代码清单 7.4 所示，由于派生建立了“属于”（is-a）关系，所以总是可以将派生类型的值直接赋给基类型的变量。

代码清单 7.4　隐式基类型转换

```
public class Program
{
    public static void Main()
    {
        // 派生类型可以隐式转换为基类型
        Contact contact = new();
        PdaItem item = contact;
        // ...

        // 基类型则必须显式转型为派生类型
        contact = (Contact)item;
        // ...
    }
}
```

派生类型 Contact“属于”一种 PdaItem，可以直接赋给 PdaItem 类型的变量。这称为**隐式转型**，不需要添加转型（强制类型转换）操作符。而且根据规则，转换总会成功，不会抛出异常。

反之则不然。PdaItem 并非一定“属于”一种 Contact。它可能是一个 Appointment 或者其

他派生类型。因此，从基类型转换为派生类型，要求执行**显式转型**，而显式转型在运行时可能失败。如代码清单 7.4 所示，为了执行显式转型，必须在原始引用名称前将要转换成的目标类型放到一对圆括号中。

执行显式转型，程序员相当于要求编译器信任他，或者说程序员告诉编译器他知道这样做的后果。只要圆括号中的目标类型确实从基类型派生，C#编译器就允许这个转换。但是，虽然在编译的时候，C#编译器允许在可能兼容的类型之间执行显式转型，但 CLR 仍会在运行时验证该显式转型。如果对象实例实际不是目标类型，那么会抛出异常。

即使类型层次结构允许隐式转型，C#编译器也允许添加转型操作符（虽然多余）。例如，将 `contact` 赋给 `item` 可以像下面这样添加转型操作符：

```
item = (PdaItem)contact;
```

甚至在无需转型时也能添加转型操作符：

```
contact = (Contact)contact;
```

> **注意：**派生类型能隐式转型为它的基类。相反，基类向派生类的转换要求显式的转型操作符，因为转换可能失败。虽然编译器允许可能有效的显式转型，但“运行时”会坚持检查，无效转型必然抛出异常。

> **初学者主题：在继承链中进行类型转换**
>
> 隐式转型为基类不会实例化一个新实例。相反，是将同一个实例作为基类型来引用，它现在提供的功能（可访问的成员）是基类型的。这类似于将 CD-ROM 驱动器说成是一种存储设备。由于并非所有存储设备都支持弹出操作，所以 `CDROM` 转型为存储设备后不再支持弹出。如果调用 `storageDevice.Eject()`，那么即使被转型的对象原本是支持 `Eject()`方法的 `CDROM` 对象，也无法通过编译。
>
> 类似地，将基类向下转型为派生类会引用更具体的类型，类型可用的操作也会得到扩展。但这种转换有限制，被转换的必须确实是目标类型（或者它的派生类型）的实例。

> **高级主题：自定义转换**

类型间的转换并不限于单一继承链中的类型。完全不相关的类型也能相互转换，比如在 Address 和 string 之间的转换。关键在于，要在两个类型之间提供转型操作符。C# 允许类型包含显式或隐式转型操作符。如果转型可能失败，比如从 long 转型为 int，那么开发人员应定义显式转型操作符。这样可以提醒开发人员：只有在他们确定转型会成功的时候，才应执行转换；否则就准备好在失败时捕捉异常。执行有损转换的时候，也应优先执行显式转型而不是隐式转型。例如，将 float 转型为 int，小数部分会被丢弃。即使接着执行一次反向转换（int 转型回 float），丢失的部分也找不回来。

代码清单 7.5 展示了隐式转型操作符的例子（GPS 坐标转换成 UTM 坐标）。

代码清单 7.5　定义转型操作符

```
class GpsCoordinates
{
    // ...

    public static implicit operator UtmCoordinates(
        GpsCoordinates coordinates)
    {
        // ...
    }
}
```

本例实现从 GpsCoordinates 向 UtmCoordinates 的隐式转换。可以写类似的转换来反转上述过程。将 implicit 替换成 explicit 就是显式转换。

7.1.2 private 访问修饰符

派生类继承除构造函数和析构器[①]之外的所有基类成员。但继承并不意味着一定能访问。例如在代码清单 7.6 中，private 字段_Name 就不能在 Contact 类中使用，因为私有成员只能在声明它们的那个类型中访问。

代码清单 7.6　私有成员能继承但不能访问

---

① 译注：如第 6 章的“高级主题：终结器”所述，C#实际并没有 C++的“析构器”或“析构函数”（destructor）的概念，用 ILDasm.exe 检查 C#的“~类名(){}”方法，会发现 C#编译器实际是在模块的元数据中生成了名为 Finalize 的 protected override 方法，这在 C#中称为“终结器”。和 C++的析构器不同的是，Finalize 方法执行的不是“确定性析构”。——摘自《CLR via C#》一书的 21.3 节。

```
public class PdaItem
{
    // ...
    private string _Name;

    public string Name
    {
        get { return _Name; }
        set { _Name = value; }
    }
    // ...
}

public class Contact : PdaItem
{
    // ...
}

public class Program
{
    public static void Main()
    {
        Contact contact = new();
        // 错误:  'PdaItem._Name'不可访问，因为它
        // 具有一定的保护级别
        contact._Name = "Inigo Montoya";
    }
}
```

根据封装原则，派生类不能访问基类的 private 成员[①]。这就强迫基类开发者决定一个成员是否能由派生类访问。本例的基类定义了一个 API，其中_Name 只能通过 Name 属性更改。假如以后在 Name 属性中添加了验证机制，那么所有派生类不需要任何修改就能马上享受到验证带来的好处，因为它们从一开始就不能直接访问_Name 字段。

> **注意：**派生类不能访问基类的私有成员。

7.1.3 protected 访问修饰符

pubic 或 private 代表两种极端情况，中间还可进行更细致的封装。可以在基类中定义只有派生类才能访问的成员（当然，任何成员始终都可以访问同一类型中的其他成员）。以

[①] 一个极少见的例外情况是派生类同时也是基类的嵌套类。

代码清单 7.7 的 ObjectKey 属性为例。

代码清单 7.7　protected 成员只能从派生类访问

```
using System.IO;

public class PdaItem
{
    public PdaItem(Guid objectKey) => ObjectKey = objectKey;
    protected Guid ObjectKey { get; }
}

public class Contact : PdaItem
{
    public Contact(Guid objectKey)
        : base(objectKey) { }

    public void Save()
    {
        // 使用<ObjectKey>.dat 作为文件名来实例化一个 FileStream
        using FileStream stream = File.OpenWrite(
            ObjectKey + ".dat");
        // ...
        stream.Dispose();
    }

    public static Contact Copy(Contact contact)
        => new(contact.ObjectKey);

    public static Contact Copy(PdaItem pdaItem) =>
    // 错误：不能访问受保护成员 PdaItem.ObjectKey。
    // 改为使用((Contact)pdaItem).ObjectKey
        new(pdaItem.ObjectKey);
}

public class Program
{
    public static void Main()
    {
        Contact contact = new(Guid.NewGuid());
        // 错误：  'PdaItem.ObjectKey'不可访问
        Console.WriteLine(contact.ObjectKey);
    }
}
```

ObjectKey 用 protected 访问修饰符定义。结果是在 PdaItem 的外部，它只能从 PdaItem

的派生类中访问。由于 Contact 从 PdaItem 派生，所以 Contact 的所有成员都能访问 ObjectKey。相反，由于 Program 不是从 PdaItem 派生，所以在 Program 内使用 ObjectKey 属性会造成编译错误。

> **注意：**基类的受保护成员只能从基类及其派生链的其他类中访问。

静态 Contact.Copy(PdaItem pdaItem)方法有一个容易被忽视的细节。开发者经常都会惊讶地发现，即使 Contact 从 PdaItem 派生，但也无法从 Contact 类内部访问一个 PdaItem 实例的受保护 ObjectKey 属性。这是由于万一该 PdaItem 是一个 Address 呢？Contact 不应访问 Address 的受保护成员。所以，封装成功阻止了 Contact 修改 Address 的 ObjectKey。不过，成功转型为 Contact 可以绕过该限制，即(Contact)pdaItem).ObjectKey。另外，访问 contact.ObjectKey 也可以。这里的基本规则是，要从派生类中访问受保护成员，必须能在编译时确定该成员是派生类（或者它的某个子类）中的实例。

7.1.4 扩展方法

扩展方法从技术上说不是类型的成员，所以不可继承。但是，由于每个派生类都可以作为它的任何基类的实例使用，所以对一个类型进行扩展的方法也可扩展它的任何派生类型。换言之，如果扩展了像 PdaItem 这样的基类，那么所有扩展方法在派生类中也能使用。但是，和所有扩展方法一样，实例方法有更高的优先级。因此，如果继承链中出现一个兼容的签名，它将优先于扩展方法。

很少需要为基类写扩展方法。扩展方法的一个基本原则是，如果手上有基类的代码，那么直接修改基类会更好。即使基类代码不可用，程序员也应考虑在基类和派生类实现的接口上添加扩展方法。下一章将具体讨论接口，并讨论它们如何与扩展方法配合使用。

7.1.5 单继承

继承树中的类理论上数量无限。例如，Customer 派生自 Contact，Contact 派生自 PdaItem，PdaItem 派生自 object。但 C#是单继承语言，C#编译成的 CIL 语言也是一样。这意味着一个类不能直接从两个类派生。例如，Contact 不能既直接派生自 PdaItem，又直接派生自 Person。

> **语言对比：C++——多继承**
>
> C#只支持单继承，这是在面向对象编程方面，它与 C++最主要的区别之一。

极少数需要多继承类结构的时候，一个解决方案是使用**聚合**（aggregation）；换言之，不是一个类从另一个类继承，而是一个类包含另一个类的实例。C# 8.0 提供了额外的语言构造来做到这一点，所以我们准备在第 8 章具体讲解如何实现聚合。

7.1.6 密封类

为了正确设计类，使其他人能通过派生来扩展功能，需要对它进行全面测试，验证派生能成功进行。代码清单 7.8 将类标记为 `sealed` 来避免非预期的派生，并避免出现因此而出现的问题。

代码清单 7.8　用密封类禁止派生

```
public sealed class CommandLineParser
{
    // ...
}

// 错误：无法从密封类型派生
public sealed class DerivedCommandLineParser
: CommandLineParser
{
    // ...
}
```

密封类用 `sealed` 修饰符禁止从其派生。`string` 类型就是用 `sealed` 修饰符禁止派生的例子。

7.2 重写基类

基类除构造函数和析构器（析构函数）之外的所有成员都会在派生类中继承。但某些情况下，一个成员可能在基类中没有得到最佳的实现。下面以 `PdaItem` 的 `Name` 属性为例。在由 `Appointment` 类继承的时候，它的实现或许是可以接受的。然而，对于 `Contact` 类，`Name` 属性应该返回 `FirstName` 和 `LastName` 属性合并起来的结果。类似地，对 `Name` 进行赋值时，应拆分为 `FirstName` 和 `LastName`。换言之，对于派生类，基类属性的声明是合适的，但实现并非总是合适的。因此，需要一种机制在派生类中使用自定义的实现来**重写**（override，覆盖或覆写）基类中的实现。

7.2.1 virtual 修饰符

C#支持实例方法和属性的重写，但不支持字段和任何静态成员的重写。为了进行重写，在基类和派生类中都要显式地执行一个操作。在基类中，必须将允许重写的每个成员都标

记为 virtual。如果一个 public 或 protected 成员没有包含 virtual 修饰符，就不允许子类重写该成员。在派生类中，则要用 override 来修饰成员，表明这是重写的成员。

代码清单 7.9 展示了属性重写的一个例子。

> **语言对比：Java——默认虚方法**
>
> Java 的方法默认为虚，希望非虚的方法必须显式密封。相反，C#默认非虚。

代码清单 7.9　重写属性

```
public class PdaItem
{
    public virtual string Name { get; set; }
    // ...
}

public class Contact : PdaItem
{
    public override string Name
    {
        get
        {
            return $"{FirstName} {LastName}";
        }

        set
        {
            string[] names = value.Split(' ');
            // 未显示错误处理
            FirstName = names[0];
            LastName = names[1];
        }
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }

    // ...
}
```

PdaItem 的 Name 属性使用了 virtual 修饰符，Contact 的 Name 属性则用关键字 override 修饰。本例拿掉 virtual 会报错，拿掉 override 会生成警告（稍后会详细讨论）。C#要求

显式使用 override 关键字重写方法。换言之，virtual 标志着方法或属性可以在派生类中被替换（重写）。

> **语言对比：Java 和 C++——隐式重写**
>
> 和 Java 和 C++不同，C#在派生类中为成员使用的 override 关键字是必须的。C#不允许隐式重写。为了重写方法，基类和派生类成员必须匹配，而且要有对应的 virtual 和 override 关键字。此外，override 关键字意味着派生类的实现会替换基类的实现。

重写成员会造成“运行时”调用最深的或者说派生得最远的（most derived）实现，如代码清单 7.10 和输出 7.1 所示。

代码清单 7.10　“运行时”调用虚方法派生得最远的实现

```
public class Program
{
    public static void Main()
    {
        Contact contact;
        PdaItem item;

        contact = new Contact();
        item = contact;

        // 通过 PdaItem 变量来设置姓名
        item.Name = "Inigo Montoya";

        // 证明已设置 FirstName 和 LastName
        Console.WriteLine(
            $"{ contact.FirstName } { contact.LastName}");
    }
}
```

输出 7.1

```
Inigo Montoya
```

代码清单 7.10 调用 item.Name 来设置姓名，该属性是在 PdaItem 中声明的。不过，最终设置的仍然是 contact 的 FirstName 和 LastName。这里的规则是：“运行时”在遇到虚方法的时候，会调用虚成员派生得最远的重写。本例实例化一个 Contact 并调用 Contact.Name，因为 Contact 包含 Name 属性派生得最远的实现。

虚方法只提供默认实现，也就是可由派生类完全重写的实现。但是，由于继承在设计上的复杂性，所以请事先想好是否需要虚方法，而不要默认就将成员声明为 `virtual`。

这一决定对程序未来的发展非常重要，因为如果定义了一个虚方法，而将来又希望将其改为非虚，那么很可能破坏那些已经重写了该方法的派生类。因此，一旦虚成员已经发布，那么就应一直保持这种状态。总之，在决定引入虚成员时一定要仔细斟酌；例如，考虑是否用 `private protected`（私有受保护）来限制它的可访问性。

> **语言对比：C++——构造期间对方法调用的调度**
>
> C++在构造期间不调度虚方法。相反，在构造期间，类型与基类型关联，而不是与派生类型关联，虚方法调用的是基类的实现。C#则相反，会将虚方法调用调度给派生得最远的类型。这是为了与以下设计规范保持一致：“总是调用派生得最远的虚成员，即使派生的构造函数尚未完全执行完毕”。但无论如何，在 C#中应尽量避免出现这种情况。[①]

最后要说的是，虚暗示着实例，只有实例成员才可以是 `virtual` 的。CLR 根据实例化期间指定的具体类型判断将虚方法调用调度给哪个。所以 `static virtual` 方法毫无意义，编译器也不允许。

7.2.2 协变返回类型

一般来说，重写方法的签名必须匹配被重写的基方法的签名。但从 C# 9.0 开始引入了一项改进，允许重写方法指定与基方法不同的返回类型，前提是返回类型与基方法的返回类型兼容，这称为**协变返回类型**（covariant return type）。代码清单 7.11 展示了一个例子。

代码清单 7.11　协变返回类型

```
public class Base
{
```

[①] 译注：意思是说，尽量不要在构造函数中调用会影响所构造对象的任何虚方法。原因是假如这个虚方法在当前要实例化的类型的派生类型中进行了重写，就会调用重写的实现。但在继承层次结构中，字段尚未完全初始化。这时调用虚方法将导致无法预测的行为。

```
    public virtual Base Create() => new();
}
public class Derived : Base
{
    public override Derived Create() => new();
}
```

注意，在这个例子中，基类的 Create()方法返回 Base，而派生类的同名方法返回 Derived。尽管后者的签名 override 了前者，但如果后者的返回类型是前者的返回类型的派生类型，那么返回类型是可以不同的。协变将在第 12 章进一步讨论。[①]

7.2.3 new 修饰符

如果重写方法没有使用 override 关键字，编译器会生成警告消息，如输出 7.2 和 7.3 所示。

输出 7. 2

```
warning CS0114:“<派生类中的成员名>”隐藏继承的成员“<基类中的成员名>”。若要使当前成员重写该实现，请添加关键字 override。否则，添加关键字 new。
```

输出 7. 3

```
warning CS0108: “<派生类中的成员名>隐藏了继承的成员“<基类中的成员名>。如果是有意隐藏，请使用关键字 new”。
```

一个明显的解决方案是添加 override 修饰符（假定基类成员是 virtual 的）。但正如警告消息所指出的那样，还可以选择使用 new 修饰符。请思考表 7.1 总结的情形——这种情形称为**脆弱的基类**（brittle base class 或 fragile base class）。

表 7.1 在什么时候使用 new 修饰符

活动	代码
程序员 A 定义了 Person	public class Person {

[①] 译注：可以这样记忆协变和逆变：在需要使用基类的地方，可以使用它的派生类，这称为“协变”；在需要使用派生类的地方，可以使用它的基类，则称为“逆变”。

类，其中包含属性 FirstName 和 LastName	`    public string FirstName { get; set; }` `    public string LastName { get; set; }` `}`
程序员 B 从 Person 派生出 Contact，并添加了额外的属性 Name。另外还定义了 Program 类，其 Main()方法会实例化一个 Contact，对 Name 赋值，然后打印姓名	`public class Contact : Person` `{` `    public string Name` `    {` `        get` `        {` `            return FirstName + " " + LastName;` `        }` `        set` `        {` `            string[] names = value.Split(' ');` `            // 未显示错误处理` `            FirstName = names[0];` `            LastName = names[1];` `        }` `    }` `}`
不久，程序员 A 自己也添加了 Name 属性，但不是将取值方法实现成 FirstName + " " + LastName，而是实现成 LastName + ", " + FirstName。另外，也没有将属性定义为 virtual，而且在 Display()方法中使用了该属性	`public class Person` `{` `    // ...` `    public string Name` `    {` `        get` `        {` `            return LastName + ", " + FirstName;` `        }` `        set` `        {` `            string[] names = value.Split(", ");` `            // 未显示错误处理` `            LastName = names[0];` `            FirstName = names[1];` `        }` `    }` `    public void Display(Person person)` `    {` `        // 显示<LastName>, <FirstName>` `        Console.WriteLine(person.Name);` `    }` `}`

Person.Name 非虚表明程序员 A 希望 Display()方法总是使用 Person 的实现，即便传给它的数据类型是 Person 的派生类型 Contact。但是，程序员 B 希望在变量数据类型为 Contact 的任何情况下都使用 Contact.Name（程序员 B 其实并不知道 Person.Name 属性，因为它最开始并不存在）。为了允许添加 Person.Name，同时不破坏两个程序员预期的行为，你不能假定该属性是 virtual 的。此外，由于 C#要求重写成员显式使用 override 修饰符，所以必须采用其他某种形式的语法，确保基类新增的成员不会造成派生类编译失败。

这种语义要用 new 修饰符实现，它在基类面前隐藏了派生类重新声明的成员。这时不是调用派生得最远的成员。相反，是搜索继承链，找到使用 new 修饰符的那个成员之前的、派生得最远的成员，然后调用该成员。如果继承链仅包含两个类，那么就使用基类的成员，感觉就是派生类并没有声明那个成员（如果派生的实现重写了基类成员）。虽然编译器会生成如输出 7.2 或 7.3 所示的警告，但假如既没有指定 override，也没有指定 new，就默认为 new，从而维持了版本的安全性。[①]

来看看代码清单 7.12 的例子，输出 7.4 展示了结果。

代码清单 7.12　对比 override 与 new 修饰符

```
public class Program
{
    public class BaseClass
    {
        public static void DisplayName()
        {
            Console.WriteLine("BaseClass");
        }
    }

    public class DerivedClass : BaseClass
    {
        // 编译器警告：DisplayName()隐藏继承的成员。
        // 如果是有意隐藏，请使用关键字 new。
        public virtual void DisplayName()
        {
            Console.WriteLine("DerivedClass");
        }
    }
```

[①] 译注：《CLR via C#》中文版的 6.6.3 节更清楚地说明了如何用 override 和 new 来进行版本控制。

```
    public class SubDerivedClass : DerivedClass
    {
        public override void DisplayName()
        {
            Console.WriteLine("SubDerivedClass");
        }
    }

    public class SuperSubDerivedClass : SubDerivedClass
    {
        public static new void DisplayName()
        {
            Console.WriteLine("SuperSubDerivedClass");
        }
    }

    public static void Main()
    {
        SuperSubDerivedClass superSubDerivedClass = new();

        SubDerivedClass subDerivedClass = superSubDerivedClass;
        DerivedClass derivedClass = superSubDerivedClass;
        BaseClass baseClass = superSubDerivedClass;

        SuperSubDerivedClass.DisplayName();
        subDerivedClass.DisplayName();
        derivedClass.DisplayName();
        BaseClass.DisplayName();
    }
}
```

输出 7.4

```
SuperSubDerivedClass
SubDerivedClass
SubDerivedClass
BaseClass
```

之所以会得到输出 7.4 的结果，是由于以下几个原因。

- SuperSubDerivedClass.DisplayName()显示 SuperSubDerivedClass，它下面没有派生类了，所以不可能还有重写版本。
- SubDerivedClass.DisplayName()是重写了基类虚成员的、派生得最远的成员。使用了 new 修饰符的 SuperSubDerivedClass.DisplayName()被隐藏。
- DerivedClass.DisplayName()是虚方法，而 SubDerivedClass.DisplayName()是重写了它的派生得最远的成员。和前面一样，使用了 new 修饰符的 SuperSubDerivedClass.DisplayName()被隐藏。

- BaseClass.DisplayName()没有重新声明任何基类成员，而且非虚。所以，它会被直接调用。

就 CIL 来说，new 修饰符对编译器生成的代码没有任何影响，但它会为方法生成 newslot 元数据特性。从 C#的角度看，它唯一的作用就是移除编译器警告。

7.2.4 sealed 修饰符

为类使用 sealed 修饰符，将禁止从该类派生（将类“密封”）。类似地，虚成员也可以密封，如代码清单 7.13 所示。这会禁止子类重写基类的虚成员。例如，假定子类 B 重写了基类 A 的一个成员，并希望禁止子类 B 的派生类继续重写该成员，就可以考虑使用 sealed 修饰符。

代码清单 7.13　密封成员

```
class A
{
    public virtual void Method()
    {
    }
}
class B : A
{
    public override sealed void Method()
    {
    }
}
class C : B
{
    // 错误：无法重写密封成员
    public override void Method()
    //{
    //}
}
```

本例为类 B 的 Method()声明使用了 sealed 修饰符，这会禁止 C 重写 Method()。

除非有很好的理由，一般很少将整个类标记为密封。事实上，人们越来越倾向于将类设置成非密封类，因为单元测试需要创建仿制对象（mock object、也称为测试替身或者 test double）来代替真正的实现。有的时候，对单独虚成员进行密封的代价过高，还不如将整个类密封。不过，一般都倾向于对单独成员进行有针对性的密封（例如，可能需要依赖基类的实现来获得正确的行为）。

7.2.5 base 成员

重写成员时，经常需要调用它的基类版本，如代码清单 7.14 所示。

代码清单 7. 14　访问基类成员

```
using static System.Environment;

public class Address
{
    public string StreetAddress;
    public string City;
    public string State;
    public string Zip;

    public override string ToString()
    {
        return $"{StreetAddress + NewLine}"
            + $"{City}, {State}  {Zip}";
    }
}

public class InternationalAddress : Address
{
    public string Country;

    public override string ToString()
    {
        return base.ToString() +
            NewLine + Country;
    }
}
```

在代码清单 7.14 中，InternationalAddress 从 Address 继承并实现了 ToString()。调用基类的实现需使用 base 关键字。base 的语法和 this 几乎完全一样，也允许作为构造函数的一部分使用（稍后详述）。

另外，即使在 Address.ToString()实现中也要使用 override 修饰符，因为 ToString()也是 object 的成员。用 override 修饰的任何成员都自动成为虚成员，子类可以进一步“特化”它的实现。

> **注意：**用 override 修饰的任何方法自动为虚。只有基类的虚方法才能重写，所以重写

后的方法还是虚方法。

7.2.6 调用基类构造函数

实例化派生类时，“运行时”首先调用基类构造函数，防止绕过基类的初始化机制。但是，如果基类没有可访问的（非私有的）默认构造函数，就不知道如何构造基类，C#编译器会报错。

为了避免因为缺少可访问的默认构造函数而造成错误，程序员需要在派生类构造函数的头部显式指定要运行哪一个基类构造函数，如代码清单 7.15 所示。

代码清单 7.15　指定要调用的基类构造函数

```
using System.Diagnostics.CodeAnalysis;

public class PdaItem
{
    public PdaItem(string name)
    {
        Name = name;
    }
    public virtual string Name { get; set; }
    // ...
}
public class Contact : PdaItem
{
    // 禁止警告，因为 FirstName 和 LastName 是通过 Name 属性来设置的
    // 不可为 null 的字段未初始化
#pragma warning disable CS8618
    public Contact(string name) :
        base(name)
    {
    }
#pragma warning restore CS8618

    public override string Name
    {
        get
        {
            return $"{FirstName} {LastName}";
        }

        set
        {
            string[] names = value.Split(' ');
            // 未显示错误处理
```

```
                FirstName = names[0];
                LastName = names[1];
            }
        }

        [NotNull]
        [DisallowNull]
        public string FirstName { get; set; }
        [NotNull]
        [DisallowNull]
        public string LastName { get; set; }

        // ...
    }

    public class Appointment : PdaItem
    {
        public Appointment(string name, string location,
            DateTime startDateTime, DateTime endDateTime) :
            base(name)
        {
            Location = location;
            StartDateTime = startDateTime;
            EndDateTime = endDateTime;
        }

        public DateTime StartDateTime { get; set; }
        public DateTime EndDateTime { get; set; }
        public string Location { get; set; }

        // ...
    }
```

通过在代码中明确指定一个基类构造函数，“运行时”就知道在调用派生类构造函数之前，要先调用哪一个基类构造函数。

7.3 抽象类

前面许多继承的例子都定义了一个名为 PdaItem 的类，它定义了在 Contact 和 Appointment 等派生类中通用的方法和属性。但是，PdaItem 本身不适合实例化。PdaItem 的实例没有意义。只有作为基类，在从其派生的一系列数据类型之间共享默认的方法实现，才是 PdaItem 类真正的意义。这意味着 PdaItem 应被设计成**抽象类**。抽象类是仅供派生的类。抽象类不能实例化，只能实例化从它派生的类。不抽象、可以直接实例化的类称为具体类。

抽象类是非常基础的一个面向对象程序设计概念，所以本节会相应地进行说明。但是，从 C# 8.0 开始，接口支持之前仅限于抽象类的*几乎*所有功能（具体而言，不能声明实例字段）

的一个超集。虽然新的接口功能会在第 8 章详述，但你事先应该理解抽象成员的概念。因此，本节会先带你认识抽象类。

> 初学者主题：抽象类
>
> **抽象类**代表抽象实体。其**抽象成员**定义了从抽象实体派生的对象*应该*包含什么。但是，抽象成员不包含具体的实现。通常，抽象类中的大多数功能性都是未实现的。一个类要从抽象类成功地派生，必须为抽象基类中的抽象方法提供具体的实现。

为了定义抽象类，要求为类定义添加 `abstract` 修饰符，如代码清单 7.16 所示。

代码清单 7.16　定义抽象类

```
// 定义抽象类
public abstract class PdaItem
{
    public PdaItem(string name)
    {
        Name = name;
    }
    public virtual string Name { get; set; }
}

public class Program
{
    public static void Main()
    {
        // 错误：无法创建抽象类的实例
        PdaItem item = new("Inigo Montoya");
    }
}
```

不能实例化还在其次，抽象类的主要特点在于它包含**抽象成员**。抽象成员是没有实现的方法或属性，作用是强制所有派生类都提供具体的实现。来看看代码清单 7.17 的例子。

代码清单 7.17　定义抽象成员

```
using static System.Environment;
```

```
// 定义抽象类
public abstract class PdaItem
{
    public PdaItem(string name)
    {
        Name = name;
    }

    public virtual string Name { get; set; }
    public abstract string GetSummary();
}

public class Contact : PdaItem
{
    // ...
    public override string Name
    {
        get
        {
            return $"{FirstName} {LastName}";
        }
        set
        {
            string[] names = value.Split(' ');
            // 未显示错误处理
            FirstName = names[0];
            LastName = names[1];
        }
    }

    public string FirstName
    {
        get
        {
            return _FirstName!;
        }
        set
        {
            _FirstName = value ??
                throw new ArgumentNullException(nameof(value)); ;
        }
    }
    private string? _FirstName;

    public string LastName
    {
        get
        {
            return _LastName!;
```

```
        }
        set
        {
            _LastName = value ?? throw new ArgumentNullException(nameof(value));
        }
    }
    private string? _LastName;
    public string? Address { get; set; }

    public override string GetSummary()
    {
        return $"名字: {FirstName + NewLine}"
        + $"姓氏: {LastName + NewLine}"
        + $"地址: {Address + NewLine}";
    }

    // ...
}

public class Appointment : PdaItem
{
    public Appointment(string name, string location,
        DateTime startDateTime, DateTime endDateTime) :
        base(name)
    {
        Location = location;
        StartDateTime = startDateTime;
        EndDateTime = endDateTime;
    }

    public DateTime StartDateTime { get; set; }
    public DateTime EndDateTime { get; set; }
    public string Location { get; set; }

    // ...
    public override string GetSummary()
    {
        return $"主题: {Name + NewLine}"
            + $"开始时间: {StartDateTime + NewLine}"
            + $"结束时间: {EndDateTime + NewLine}"
            + $"地点: {Location}";
    }
}
```

代码清单 7.17 将 GetSummary()定义为抽象成员，所以它不包含任何实现。随即，代码在 Contact 中重写它并提供具体实现。由于抽象成员设计为被重写，所以自动为虚（但不能

显式这样声明[1])。此外，抽象成员不能声明为私有，否则派生类看不见它们。

开发具有良好设计的对象层次结构殊为不易。所以在编程抽象类型时，一定要自己实现至少一个（最好多个）从抽象类型派生的具体类型，以检验自己的设计。

如果不在 Contact 中提供 GetSummary()的实现，那么编译器会报错。

> **注意**：抽象成员必须被重写，所以自动为虚，但不能用 virtual 关键字显式声明。

> **语言对比：C++——纯虚函数**
>
> C++使用神秘的“=0”表示法来定义抽象函数。这些函数在 C++中称为**纯虚函数**。与 C#相反，C++不要求类本身进行任何特殊的声明。和 C#的抽象类修饰符 abstract 不同，当 C++类包含纯虚函数时，不需要对该类的声明进行任何特殊的修饰。

> **注意**：通过声明抽象成员，抽象类的编程者清楚地指出：为了建立具体类与抽象基类（本例是 PdaItem）之间的“属于”（is-a）关系，派生类必须实现抽象成员——抽象类无法为这种成员提供恰当的默认实现。

> **初学者主题：多态性**
>
> 相同成员签名在不同类中有不同实现，这称为多态性（polymorphism），是面向对象编程的一个重要概念。英语“poly”代表“多”，“morph”代表“形态”，多态性是指同一个签名可以有多个实现。同一个签名不能在一个类中多次使用，所以该签名的每个实现必然包含在不同类中。
>
> 多态性的基本设计思想是：只有对象自己才知道如何最好地执行特定操作，通过规定

[1] 译注：作者的意思是说不能写成 abstract virtual。记住，用 abstract 来修饰的不包含实现，而用 virtual 修饰的必然包含实现。

> 调用这些操作的通用方式，多态性还促进了代码重用，因为通用的东西不必重复编码。例如，假定有多种类型的文档，每种文档都知道具体如何执行自己这种文档的Print()操作，那么不是定义单个 Print()方法，在其中包含 switch 语句来处理每种文档类型的特殊打印逻辑，而是利用多态性调用与想要打印的文档类型对应的Print()方法。例如，为字处理文档类调用 Print()，会根据字处理文档的特点进行打印。相反，为图形文档类调用同一个方法，会根据图形文档的特点进行打印。无论如何，对于任意文档类型，打印它唯一要做的就是调用 Print()，其他不用考虑。
>
> 将具体打印逻辑从 switch 语句中移除有利于维护。首先，具体的实现位于不同文档类的上下文中，而不是位于另一个较远的地方，这符合封装原则。其次，以后添加新的文档类型时，我们不需要更改 switch 语句。相反，唯一要做的就是在新的文档类型中实现 Print()签名。

抽象成员是实现多态性的一个手段。基类指定方法签名，派生类提供具体实现，如代码清单 7.18 和输出 7.5 所示。

代码清单 7.18　利用多态性列出 PdaItem

```
public class Program
{
    public static void Main()
    {
        PdaItem[] pda = new PdaItem[3];

        Contact contact = new("Sherlock Holmes")
        {
            Address = "221B Baker Street, London, England"
        };
        pda[0] = contact;

        Appointment appointment = new(
               "Soccer tournament", "Estádio da Machava",
               new DateTime(2008, 7, 18), new DateTime(2008, 7, 19));
        pda[1] = appointment;

        contact = new Contact("Anne Frank")
        {
            Address = "Apt 56B, Whitehaven Mansions, Sandhurst Sq, London"
        };
        pda[2] = contact;

        List(pda);
    }
```

```
        public static void List(PdaItem[] items)
        {
            // 利用多态性来实现。派生类知道 GetSummary()的实现细节。
            foreach (PdaItem item in items)
            {
                Console.WriteLine("________");
                Console.WriteLine(item.GetSummary());
            }
        }
    }
```

输出 7.5

```
________
名字: Sherlock
姓氏: Holmes
地址: 221B Baker Street, London, England


________
主题: Soccer tournament
开始时间: 2008/7/18 0:00:00
结束时间: 2008/7/19 0:00:00
地点: Estádio da Machava
________
名字: Anne
姓氏: Frank
地址: Apt 56B, Whitehaven Mansions, Sandhurst Sq, London
```

这样就可以调用基类的方法，但方法具体由派生类来实现。输出 7.5 证明，List()方法能成功显示 Contact 和 Appointment 对象，而且每种对象都以自定义方式显示。调用抽象 GetSummary()方法，实际调用的是每个实例特有的重写方法。

7.4 所有类都从 System.Object 派生

任何类，不管是自定义类，还是系统内建的类，都定义好了如表 7.2 所示的方法。

表 7.2 System.Object 的成员

方法名	说明
public virtual bool Equals(object o)	如果作为参数提供的对象和当前对象实例包含相同的值，就返回 true
public virtual int GetHashCode()	返回对象值的哈希码。它对于 HashTable 这样的集合非常有用

`public Type GetType()`	返回与对象实例的类型对应的 `System.Type` 类型的一个对象
`public static bool ReferenceEquals (object a, object b)`	两个参数引用同一个对象就返回 `true`
`public virtual string ToString()`	返回对象实例的字符串表示
`public virtual void Finalize()`	析构器的一个别名，通知对象准备终结。C# 禁止直接调用该方法
`protected object MemberwiseClone()`	执行浅拷贝来克隆对象。会复制引用，但被引用类型中的数据不会复制

表 7.2 的所有方法都通过继承为所有对象所用，所有类都直接或间接从 `object` 派生。即使字面值也支持这些方法。所以，下面这种看起来颇为奇怪的代码实际是合法的。

```
Console.WriteLine( 42.ToString() );
```

即使类定义没有显式指明自己从 `object` 派生，也肯定是从 `object` 派生的。所以，在代码清单 7.19 中，`PdaItem` 的两个声明会产生完全一致的 CIL。

代码清单 7. 19　即使不显式指定从哪里派生，也隐式派生自 `System.Object`

```
public class PdaItem
{
    // ...
}
public class PdaItem : object
{
    // ...
}
```

如果 `object` 的默认实现不好使，程序员可以自己重写其中的三个虚方法（`Finalize()`自然排除在外），第 10 章将介绍具体如何做。

7.5 类型检查

自 C# 1.0 以来，出现了多种检查对象类型的方法，包括 `is` 操作符和 `switch` 语句。C# 8.0

更是引入了 switch 表达式，允许从这种表达式返回值。

7.5.1 使用 is 操作符进行类型检查

由于 C#允许在继承链中进行向下转型，所以在尝试进行类型转换之前，可能需要先确定基础类型是什么。此外，在没有实现多态性的情况下，一些要依赖特定类型的行为也可能要事先确定类型。为了确定类型，C# 自 C# 1.0 就引入了 is 操作符（参见代码清单 7.20）。

代码清单 7. 20　使用 is 操作符确定基础类型

```
public class Person
{
    // ...
}

public class Employee : Person
{
    // ...
}

public class Program
{
    private static object? GetObjectById(string id)
    {
        // ...
    }

    public static void Main(params string[] args)
    {
        string id = args[0];
        object? entity = GetObjectById(id);

        if (entity is Person)
        {
            Person person = (Person) entity;
            Console.WriteLine(
                $"Id 对应于一个{nameof(Person)}对象: {
                    person.FirstName} {person.LastName}。");

            if (entity is Employee employee)
            {
                Console.WriteLine(
                    $"Id({employee.Id})也是一个{
                        nameof(Employee)}对象。");
            }
        }
```

```
        else if (entity is null)
        {
            Console.WriteLine(
                $"Id 未知，所以返回 null。");
        }
        else
        {
            Console.WriteLine(
                $"Id'{id}'不是一个{
                    nameof(Employee)}或{nameof(Person)}对象。");
        }
    }
}
```

调用 GetObjectById()方法将返回基于指定 id 的可空对象。代码清单 7.20 在返回的实例上调用 is 操作符。首先，它检查值是否为 null。接着，它判断返回的对象是不是 Person，然后判断是不是 Employee。如果 GetObjectById()返回一个 Person，那么输出将是：“ID 对应于一个 Person 对象……。”如果 GetObjectById()返回一个 Employee ，那么会输出消息来说明该 ID 既是 Person，也是 Employee。由于 Employee 是 Person 的派生类，因此所有 Employee 同时也是 Person。

7.5.1 用声明进行类型检查

通过显式转型，程序员宣布自己负责创建清晰的代码逻辑来避免无效的强制类型转换。如可能发生无效转型，那么应首选进行类型检查并完全避免异常。代码清单 7.20 对此进行了演示。它首先检查 entity 是不是一个 Person，是的话就把它转型为 Person。

```
// ...
if (entity is Person)
{
    Person person = (Person) entity;
    // ...
}
```

is 操作符的好处是能创建一个显式转型可能失败、但又不会产生异常处理开销的代码路径。从 C# 7.0 开始，is 操作符（以及稍后讲述的 switch 语法）允许在同一个步骤中执行类型检查和赋值。代码清单 7.20 对此进行了演示。

```
if (entity is Employee employee)
{
    // ...
}
```

在本例中，在检查输入操作数是否为员工时，还声明了一个类型为 Employee 的新变量。因此，GetObjectById(id)的结果与 Employee 类型进行匹配（类型模式匹配），并在同一表

达式中赋给变量 employee。如果 GetObjectById(id)的结果为 null 或者不是 Employee，则生成 false，不进行赋值，开始执行 else 子句。

注意，employee 变量在 if 语句内部和之后都可用。但是，在 else 子句内部或之后访问它之前，需要向它赋值。此外，可以使用 var 隐式指定数据类型，这符合预期。但是，和平时的局部变量声明一样，当数据类型不明显时需谨慎使用 var。[1]

7.5.3 使用 switch 语句进行类型检查

还可以用 switch 语句进行类型检查，但除了 null 检查，它还能检查多种类型——所有检查都在同一个语句中进行。代码清单 7.21 对此进行了演示，它使用了一个从多种类型返回时间的方法（使用.NET 6.0 新增的 TimeOnly）。

代码清单 7.21　用 switch 语句进行类型检查

```
public static TimeOnly GetTime(object input)
{
    switch (input)
    {
        case DateTime datetime:
            return TimeOnly.FromDateTime(datetime);
        case DateTimeOffset datetimeOffset:
            return TimeOnly.FromDateTime(datetimeOffset.DateTime);
        case string dateText:
            return TimeOnly.Parse(dateText);
        case null:
           throw new ArgumentNullException(nameof(input));
        default:
            throw new ArgumentException(
                $"无效类型 - {input.GetType().FullName}");
    };
}
```

代码清单 7.21 列出了一个处理 null 的情况。但是，如果所有这些 case 都不匹配，那么会在 default 中抛出异常。与 is 操作符一样，可以同时声明一个变量，以便访问与 case 匹配的对象实例。

[1] 译注：出于对代码可维护性的考虑，许多企业要求平时避免使用 var（或 C++的 auto），除非是没有它们就特别不好写的一些逻辑。

从 C# 8.0 开始，switch 语句包括一种更简洁的表达式形式，可以从中返回一个值。

7.5.4 使用 switch 表达式进行类型检查

与 switch 语句相似，从 C# 8.0 开始引入的 switch 表达式针对不同的 case 提供了不同的路径。这种 case 通常是一个匹配表达式（兼具 switch 表达式和语句的作用）。与 switch 语句不同的是，switch 表达式没有标签，没有 break 语句，甚至根本没有封闭的语句。最重要的是，除非抛出异常，否则 switch 表达式会返回一个值。这使得它成为代码清单 7.21 的 switch 语句的首选迭代方案，因为后者之所以允许返回值，仅仅是因为嵌入的 return 语句的作用。例如，不能将 switch 语句的结果赋值给一个变量。代码清单 7.22 展示了修改后的例子。

代码清单 7. 22　使用 switch 表达式来返回一个值

```
public static TimeOnly GetTime(object input) =>
    input switch
    {
        DateTime datetime
            => TimeOnly.FromDateTime(datetime),
        DateTimeOffset datetimeOffset
                => TimeOnly.FromDateTime(datetimeOffset.DateTime),
        string dateText => TimeOnly.Parse(
            dateText),
        null => throw new ArgumentNullException(nameof(input)),
        _ => throw new ArgumentException(
            $"无效类型 - {input.GetType().FullName}"),
    };
```

在代码清单 7.22 中，注意 switch 表达式与 switch 语句的区别。现在，switch 语句的输入操作数出现在 switch 关键字之前。每个匹配表达式单纯用类型来标识，可以选择后跟一个变量声明。另外，与用冒号来指定标签不同，switch 表达式使用的是 Lambda 操作符=>。最后，现在不是使用 default 关键字来捕捉其他所有情况，而是使用弃元（即_）。

从功能上说，代码清单 7.21 和代码清单 7.22 是完全等价的。它们都检查类型，允许变量声明，允许 null 检查，并且都有一个默认情况。但要注意的是，switch 表达式要求_（即默认 case）必须放在最后，否则它会屏蔽后续所有匹配表达式。相反，switch 语句的 default 提供了相同的功能，但位置随意。

7.6 模式匹配

is 操作符和 switch 语句一直处于不温不火的状态，变动很少，直到 C# 7.0 增加了声明和经过了大幅改进的模式匹配功能。模式匹配提供了一种特殊的语法对条件进行求值，并相

应地选择代码分支。C# 团队一直到 C# 11 都在持续改进模式匹配。

本节将回顾所有的模式匹配功能：类型模式和常量模式（C# 7.0），关系模式（C# 9.0），逻辑模式（C# 9.0）、元组模式（C# 8.0）、位置模式（C# 9.0）、属性模式（C# 7.0）、扩展属性模式（C# 10）以及列表模式（C# 11.0）。所有这些功能都利用了递归模式的能力，允许对类型进行多次检查。

7.6.1 常量模式(C# 7.0)

第 4 章讲解了如何使用 is 操作符来检查 null（即 data is null）和 not null，并解释了基本 switch 语句，这些都是常量模式的例子。它的基本原理是，可以将输入操作数与任意常量进行比较。例如，可以将 data.Day 与值 21 进行比较，看看是不是一个月的 21 号。代码清单 7.23 演示了 is 和 switch 表达式。

代码清单 7. 23　使用 is 操作符进行常量模式匹配

```
using System.Diagnostics.CodeAnalysis;

public static class SolsticeHelper
{
    public static bool IsSolstice(DateTime date)
    {
        if (date.Day is 21)
        {
            if (date.Month is 12)
            {
                return true;
            }
            if (date.Month is 6)
            {
                return true;
            }
        }
        return false;
    }

    public static bool TryGetSolstice(DateTime date,
        [NotNullWhen(true)] out string? solstice)
    {
        if (date.Day is 21)
        {
            if ((solstice = date.Month switch
            {
                12 => "冬至",
                6 => "夏至",
```

```
                    _ => null
                }) is not null) return true;
            }
            solstice = null;
            return false;
        }
    }
```

常量可以是任何字面值，包括字符串、定义的常量，另外当然还有 null。但要注意，虽然可以将字符串与空串""进行比较，但将其与 string.Empty 进行比较会失败，因为 string.Empty 没有被声明为常量。

7.6.2 关系模式(C# 9.0)

在关系模式中，可以使用<，>，<= 和>=等操作符与一个常量进行比较。代码清单 7.24 演示了关系模式的 is 操作符和 switch 表达式版本。（注意，==操作符不算在关系模式内，因为这个功能在常量模式匹配中就已经可以使用了。）

代码清单 7.24　关系模式匹配示例

```
public static bool IsDeveloperWorkHours(int hourOfTheDay) =>
    hourOfTheDay is > 10 && hourOfTheDay is < 24;

public static string GetPeriodOfDay(int hourOfTheDay) =>
    hourOfTheDay switch
    {
        < 6 => "黎明",
        < 12 => "上午",
        < 18 => "下午",
        < 24 => "晚上",
        int hour => throw new ArgumentOutOfRangeException(nameof(hourOfTheDay),
            $"指定了一天中无效的小时数。")
    };
```

需要注意的是，从 C# 7.0 开始，匹配表达式就不再要求必须互斥了。例如，可以同时有匹配表达式>0 和<24。无论小时数是多少，这两个都可以为真。顺序将决定代码执行的最终路径。因此，在 switch 语句和 switch 表达式中排列匹配表达式时必须谨慎（和排列一系列 if 语句是一样的道理）。

7.6.3 逻辑模式(C# 9.0)

使用逻辑模式，可以将带有单个变量的关系匹配表达式与操作符 not，and 和 or 结合，从而分别提供否定（negated）、合取（conjunctive）和析取（disjunctive）模式。因此，现在

不需要使用像&&或||这样的逻辑操作符（例如，代码清单 7.24 中的IsDeveloperWorkHours()方法）。相反，可以指定输入操作数一次，然后用多个匹配表达式对它进行判断，如代码清单 7.25 所示。

代码清单 7. 25　逻辑模式匹配示例

```
public static bool IsStandardWorkHours(
    TimeOnly time) =>
        time.Hour is > 8
            and < 17
            and not 12; // 午餐时间不是标准工作时间

public static bool TryGetPhoneButton(
    char character,
    [NotNullWhen(true)] out char? button)
{
    return (button = char.ToLower(character) switch
    {
        '1' => '1',
        '2' or >= 'a' and <= 'c' => '2',
        // not 操作符和圆括号示例(C# 10)
        '3' or not (< 'd' or > 'f') => '3',
        '4' or >= 'g' and <= 'i' => '4',
        '5' or >= 'j' and <= 'l' => '5',
        '6' or >= 'm' and <= 'o' => '6',
        '7' or >= 'p' and <= 's' => '7',
        '8' or >= 't' and <= 'v' => '8',
        '9' or >= 'w' and <= 'z' => '9',
        '0' or '+' => '0',
        _ => null,// 设置 button 来指示值是无效的
    }) is not null;
}
```

操作符默认的优先级顺序是 not，and，or。因此，前两个例子不需要使用圆括号来明确指定优先级顺序。但是，如代码清单 7.25 所示，可以在 switch 表达式中使用圆括号来改变这个默认顺序。

需要注意的是，在使用 or 和 not 操作符的时候，不能同时声明一个变量。例如：

```
if (input is "data" or string text) { }
```

上述代码会导致编译时错误：“CS8780: 在 not 或 or 模式中不能声明变量”。这是因为它会造成关于 text 是否已初始化的歧义。

7.6.4 圆括号模式(C# 9.0)

为了改变逻辑模式操作符的优先级顺序，可以使用圆括号对它们进行分组，如代码清单 7.26 所示。

代码清单 7. 26　圆括号模式

```
public bool IsOutsideOfStandardWorkHours(
    TimeOnly time) =>
        time.Hour is not
            (> 8 and < 17 and not 12); // 圆括号模式 - C# 10
```

圆括号模式在 is 操作符和 switch 语法中都可以使用。

7.6.5 元组模式(C# 8.0)

使用元组模式匹配，可以检查元组中的常量值，或者将元组项赋给一个变量，如代码清单 7.27 所示。

代码清单 7. 27　使用 is 操作符进行元组模式匹配

```
public static void Main(params string[] args)
{
    const int command = 0;
    const int fileName = 1;
    const string dataFile = "data.dat";

    // ...
    if ((args.Length, args[command].ToLower()) is (1, "cat"))
    {
        Console.WriteLine(File.ReadAllText(dataFile));
    }
    else if ((args.Length, args[command].ToLower()) is (2, "encrypt"))
    {
        string data = File.ReadAllText(dataFile);
        File.WriteAllText(
            args[fileName], Encrypt(data).ToString());
    }
}
```

和你预期的一样，switch 表达式和 switch 语句也支持这个模式，如代码清单 7.28 所示。

代码清单 7. 28　使用 switch 语句进行元组模式匹配

```
public static void Main(params string[] args)
{
    const int action = 0;
    const int fileName = 1;
    const string dataFile = "data.dat";

    // ...
    switch ((args.Length, args[action].ToLower()))
    {
        case (1, "cat"):
            Console.WriteLine(File.ReadAllText(dataFile));
            break;
        case (2, "encrypt"):
            {
                string data = File.ReadAllText(dataFile);
                File.WriteAllText(
                    args[fileName], Encrypt(data).ToString());
            }
            break;
        default:
            Console.WriteLine("参数无效。");
            break;
    }
}
```

在代码清单 7.27 和代码清单 7.28 中，我们针对一个元组进行模式匹配。元组中包含了 args 数组的长度及其元素。在第一个匹配表达式中，我们检查是否提供了一个参数，而且该参数是动作"cat"。在第二个匹配表达式中，我们检查数组的第一项是不是动作"encrypt"。此外，如果初始匹配表达式求值为 true，那么代码清单 7.27 将元组中的第三个元素赋给变量 fileName。代码清单 7.28 的 switch 语句则没有进行变量赋值，因为 switch 语句的输入操作数对所有匹配表达式来说都是相同的。

每个元素匹配都可以是常量或变量。由于元组在 is 操作符求值之前就已经实例化，所以不能先使用"encrypt"场景，因为如果请求的是"cat"动作，args[fileName]就不是一个有效的索引了。

7.6.6 位置模式(C# 8.0)

基于 C# 7.0 引入的解构函数构造（参见第 6 章），C# 8.0 新增了位置模式匹配，其语法与元组模式匹配非常相似，如代码清单 7.29 所示。

代码清单 7.29　使用 is 操作符进行位置模式匹配

```
using System.Drawing;

public static class PointHelper
{
    public static void Deconstruct(
        this Point point, out int x, out int y) =>
            (x, y) = (point.X, point.Y);

    public static bool IsVisibleOnVGAScreen(Point point) =>
        point is (>=0 and <=1920, >=0 and <=1080);

    public static string GetQuadrant(Point point) => point switch
    {
        (>=0, >=0) => "象限 I",    //  II | I
        (<=0, >=0) => "象限 II",   // ____|____
        (<=0, <=0) => "象限 III", //     |
        (>=0, <=0) => "象限 IV"    // III | IV
    };
}
```

System.Drawing.Point 类型默认没有解构函数（deconstructor）。但是，我们可以通过扩展方法来为它添加一个，以便将 Point 转换为包含 X 和 Y 坐标的一个元组。然后，使用解构函数来匹配模式中每个以逗号分隔的匹配表达式的顺序。在 IsVisibleOnVGAScreen()的例子中，X 与>=0 and <=1920 匹配，而 Y 与>=0 and <=1080 匹配。在 GetQuadrant()的 switch 表达式中，也使用了类似的范围表达式。

7.6.7 属性模式(C# 8.0 和 C# 10.0)

使用属性模式，匹配表达式将以 switch 表达式中标识的数据类型的属性名称和值为基础。代码清单 7.30 展示了如何进行属性模式匹配。

代码清单 7.30　属性模式匹配

```
public record class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Role { get; set; }

    public Employee(int id, string name, string role) =>
```

```
        (Id, Name, Role) = (id, name, role);
}


public class ExpenseItem
{
    public int Id { get; set; }
    public string ItemName { get; set; }
    public decimal CostAmount { get; set; }
    public DateTime ExpenseDate { get; set; }
    public Employee Employee { get; set; }

    public ExpenseItem(
        int id, string name, decimal amount, DateTime date,
            Employee employee) =>
                (Id, Employee, ItemName, CostAmount, ExpenseDate) =
                (id, employee, name, amount, date);

    public static bool ValidateExpenseItem(ExpenseItem expenseItem) =>
        expenseItem switch
        {
            // 注意：属性模式会核实输入值不为 null。
            // 以下是扩展属性模式
            { ItemName.Length: > 0, Employee.Role: "Admin" } => true,

            // 以下是普通属性模式
            {
                ItemName: { Length: > 0 }, Employee: { Role: "Manager" },
                ExpenseDate: DateTime date
            }
                when date >= DateTime.Now.AddDays(-30) => true,
            {
                ItemName.Length: > 0, Employee.Name.Length: > 0,
                CostAmount: <= 1000, ExpenseDate: DateTime date
            }
                when date >= DateTime.Now.AddDays(-30) => true,
            { } => false, // 可以删除这个 not null 检查
            _ => false
        };
}
```

不管什么属性匹配表达式，首先要注意的是，它们只有在输入操作数不为 null 时才会匹配。这就是为什么一个空属性匹配表达式{ }，只要输入操作数不为 null 就会匹配。在代码清单 7.30 中，{ }匹配表达式其实是多余的，因为 switch 语句最后的默认匹配表达式会捕获所有 null 以及之前未捕获的、直通下来的所有非 null 输入操作数。虽然{ }可以作为一个非 null 检查，但直接指定 not null（C# 9.0）更佳，因为它的语法更清晰。

在代码清单 7.30 中，switch 表达式中的第一个匹配表达式以 ItemName.Length: > 0 开始，要求 ItemName.Length 大于 0。该表达式暗示 ItemName 同时不为 null。事实上，由于初始的 null 检查是隐含的，所以编译器不允许在属性表达式中使用空条件操作符，即 ItemName?.Length。

在代码清单 7.30 中，第一个匹配表达式还要求 Employee.Role 设为"Admin"。如果这两个匹配表达式都匹配，那么 switch 表达式将返回 true，表示 ExpenseItem 有效。完整的匹配表达式使用了在 C# 10.0 中引入的**扩展属性模式匹配**语法，可以用点符号访问子属性（Length 和 Role）。

代码清单 7.30 的第二个匹配表达式是{ ItemName: { Length: > 0 }, Employee: {Role: "Manager" }, ExpenseDate: DateTime date }，这是传统的属性匹配表达式（非 C# 10.0 的扩展版本），语法略有不同。属性匹配表达式最早是在 C# 7.0 中引入的，并仍然支持，只是可能会显示警告，因为扩展语法更简单，在所有情况下都更推荐。

第二个匹配表达式包括对 ExpenseDate 的属性匹配，并声明了一个变量 date。虽然 date 仍然用作 switch 筛选器的一部分，但它出现在 **when 子句**中，而不是出现在匹配表达式中。

7.6.8 when 子句

不管什么形式的常量和关系匹配表达式，都必须使用常量。但是，这可能会造成一些限制，因为比较操作数在编译时经常是未知的。为了克服这一限制，C# 7.0 引入了 when 子句，可以在其中添加任何条件表达式（返回布尔值就行）。

代码清单 7.30 声明了一个 DateTime date 变量来检查 ExpenseItem.ExpenseDate 是否不超过 30 天。然而，由于这个值依赖于当前日期和时间，所以不可能是常量，我们无法使用匹配表达式。为此，代码在匹配表达式后面使用了 when 子句，将 date 值与 DateTime.Now.AddDays(-30)比较。when 子句为（可选的）更复杂的条件提供了一个全能位置，克服了只能使用常量的限制。

前面说过，从 C# 7.0 开始，匹配表达式不一定非要互斥，而 when 子句的加入功不可没。有了 when 子句，就不再需要在编译时确定完整的 case 逻辑和检查互斥性。但要提醒你注意的是，switch 表达式中每个 case 出现的顺序非常关键，要避免较早的 case 捕获本该由后面的 case 处理的条件。

7.6.9 使用无关类型进行模式匹配

模式匹配的一个有趣能力是，它可以从无关的类型中提取数据，并将数据转换为通用格式。代码清单 7.31 展示了一个例子。

代码清单 7.31　switch 表达式中的模式匹配

```
public static string? CompositeFormatDate(
        object input, string compositeFormatString) =>
    input switch
    {
        DateTime
            { Year: int year, Month: int month, Day: int day }
            => (year, month, day),
        DateTimeOffset
            { Year: int year, Month: int month, Day: int day }
                => (year, month, day),
        DateOnly
            { Year: int year, Month: int month, Day: int day }
                => (year, month, day),
        string dateText => DateTime.TryParse(
            dateText, out DateTime dateTime) ?
                (dateTime.Year, dateTime.Month, dateTime.Day) :
                // default ((int Year, int Month, int Day)?)更佳，
                // 但要到第 12 章才会讲到。
                ((int Year, int Month, int Day)?) null,
        _ => null
    } is (int, int, int) date ? string.Format(
        compositeFormatString, date.Year, date.Month, date.Day) : null;
```

在代码清单 7.32 的 switch 表达式中，第一个匹配表达式使用类型模式匹配（C# 7.0）来检查输入是否为 DateTime 类型。如果条件成立，它会将结果传递给属性模式匹配来声明并赋值变量 year，month 和 day。然后，使用这些变量在一个元组表达式中返回元组(year, month, day)。DateTimeOffset 和 DateOnly 的匹配表达式也是以同样的方式工作的。

对于 string 匹配表达式，如果 TryParse()不成功，那么我们返回一个 default ((int Year, int Month, int Day)?)[1]，它求值为 null。注意，这里不能直接返回 null，因为(int Year, int Month, int Day)（由其他匹配表达式返回的 ValueTuple 类型）和 null 之间没有隐式转换。相反，需要指定一个可空的元组数据类型，以准确地判断 switch 表达式的类型。如果不使用 default 操作符，那么替代方案是执行强制类型转换，即本例使用的((int Year, int Month, int Day)?) null。

代码清单 7.31 总共使用了 4 种数据类型，虽然在概念上相似，但除了都从 object 派生外，它们之间并无继承关系。因此，输入操作数可用的唯一数据类型就是 object。然而，我们可以从每一个类型中提取代表年、月、日的属性（即使这些名称不存在，或者不同的数据

[1] 详情参见第 12 章。

类型有不同的叫法）。在本例中，我们提取(int Year, int Month, int Day)?，即一个可空的元组。另外，只要元组不为null，我们就能用提取的年、月和日值来构建复合字符串。

7.6.10 递归模式匹配(C# 7.0)

如前所述，模式匹配的大部分能力要在 switch 语句或 switch 表达式中才能真正发挥出来。然而，一个例外可能是当模式匹配被递归使用时。代码清单 7.32 提供了一个例子（不可否认，人为迹象很严重），展示了当递归应用模式时潜在的复杂性。

代码清单 7. 32　用 is 操作符进行递归模式匹配

```
Person inigo = new("Inigo", "Montoya");
var buttercup =
    (FirstName: "Princess", LastName: "Buttercup");

(Person inigo, (string FirstName, string LastName) buttercup) couple =
    (inigo, buttercup);

if (couple is
    ( // 元组：从 Person 的解构函数获取
        ( // 位置：选择左侧或元组
            { // firstName 的属性
                Length: int inigoFirstNameLength
            },
         _ // 丢弃元组的姓氏部分
        ),
    { // Princess Buttercup 元组的属性
        FirstName: string buttercupFirstName }))
{
    Console.WriteLine(
        $"({ inigoFirstNameLength }, { buttercupFirstName })");
}
else
{
    // ...
}
// ...
```

在这个例子中，couple 的类型是：[①]

[①] 译注：在《公主新娘》中，Inigo Montoya 和 Princess Buttercup 是一对儿。

```
(Person, (string FirstName, string LastName))
```

因此，第一个匹配发生在外层元组(inigo, buttercup)上。接下来，利用 Person 类的解构函数对 inigo 使用位置模式匹配。这会选择一个(FirstName, LastName)元组，从中使用属性模式匹配来提取 inigo.FirstName 值的 Length（"Inigo"字符串的长度为 5）。

位置模式匹配中的 LastName 部分用下划线标记为弃元。最后，使用属性模式匹配选择 buttercup.FirstName（即"Princess"）

尽管模式匹配是选择数据的一种强大手段，但使用需谨慎，特别要注意可读性。即便像代码清单 7.32 那样提供了注释，但理解代码仍然可能具有挑战性。没有注释，就更难了，如下所示。

```
if (couple is ( ( { Length: int inigoFirstNameLength }, _ ),
    { FirstName: string buttercupFirstName })) { ...}
```

即便如此，在 switch 语句和 switch 表达式中，模式匹配的强大是毋庸置疑的。

7.6.11 列表模式(C# 11.0)

C# 11 的一个重要的新特性是引入了列表模式。这使得可以对数组之类的数据项列表进行模式匹配。在列表中的元素上，可以使用任何类型的模式匹配。代码清单 7.33 演示了如何使用这一特性来解析 Main 方法的 args 参数。

代码清单 7.33　在 switch 表达式中进行模式匹配

```
public static void Main(string[] args)
{
    // 为了简化，所有选项假定全小写

    // 第一个参数是用'/', '-'或'--'等前缀标注的选项

    switch (args)
    {
        case ["--help" or ['/' or '-', 'h' or '?']]:
            // 例: --help, /h, -h, /?, -?
            DisplayHelp();
            break;
        case [ ['/' or '-', char option], ..]:
            // 选项以'/', '-'开头，有 0 个或更多实参
            if(!EvaluateOption($"{option}", args[1..]))
            {
                DisplayHelp();
            }
            break;
```

```
            case [ ['-', '-', ..] option, ..]:
                // 选项以"--"开头，有 0 个或更多实参
                if(!EvaluateOption(option[2..], args[1..]))
                {
                    DisplayHelp();
                }
                break;

            // 用以下 case 来提供默认行动是多余的，因为它和 default 重复了，
            // 只是出于演示目的而提供。
            case []:
                // 未提供命令行参数

            default:
                DisplayHelp();
                break;
        }
    }

    private static bool EvaluateOption(string option, string[] args) =>
        (option, args) switch
        {
            ("cat" or "c", [string fileName]) =>
                CatalogFile(fileName),
            ("copy", [string sourceFile, string targetFile]) =>
                CopyFile(sourceFile, targetFile),
            _ => false
        };

    private static bool CopyFile(object sourceFile, string targetFile)
    {
        Console.WriteLine($"复制 '{sourceFile}' '{targetFile}'...");
        return true;
    }
```

在代码清单 7.33 中，第一个 case 寻找" --help"作为 args 数组的第一个且唯一一个元素。如果没有匹配，那么列表匹配表达式的剩余部分将匹配第一个以/或-开头，并后跟?或 h 的元素。注意，列表模式的后半部分寻找恰好两个字符的情况（因为字符串本质上是字符数组），表达式中的逗号（,）将选项前缀与单字符表达式分开。

```
['/' or '-', 'h' or '?']
```

接下来的两个 case 也对 args 的第一个元素进行求值，判断第一个字符是/还是-，或者在第三个 case 中，args 的第一个元素是否以--开头。在这两种情况下，如果匹配，args[0]都被赋给 option 变量。遗憾的是，没有办法将列表剩余的元素捕获到一个变量中。列表模式匹配总是寻找确切数量的元素，或者允许使用..来表示不需要对剩余元素做进一步的

求值。因此，在调用 EvaluateOption()时，我们使用范围 args[1..]来传递除第一个之外的所有 args 数组元素（有关范围的示例，请参见第 3 章）。

第四个 case 使用[ ]来标识一个 0 元素的列表。这在本例中是多余的，因为 default 情况也会捕获 args 数组包含零个元素的情况。

在代码清单 7.33 的 EvaluateOption()方法中，我们使用一个元组来先求值命令部分。元组的第二个元素则求值剩余的参数。对于"cat"匹配表达式，它寻找只包含单个数据项的列表，而对于"copy"匹配表达式，则寻找恰好有两个元素的列表。在这两种情况下，都会将数组元素捕获到变量中，并用它们来调用相应的命令方法（编录只需提供一个文件名，而复制需要提供两个）。

列表模式匹配的功能很强大，而且由于它操作的是字符串中的字符，所以无需使用正则表达式，就可以进行简单的字符串解析。

7.7 能利用多态性就避免模式匹配

虽然模式匹配是很重要的一个能力，但在使用它之前，应考虑涉及多态性的问题。多态性允许将一个行为扩展到其他数据类型，同时不必对定义行为的实现进行任何修改。例如，将姓名属性 Name 添加到基类 PdaItem 中，然后就可以从 PdaItem 的不同子类对象中获取不同的 Name 值。虽然也可以编写一个长长的 switch 语句或表达式，用类型模式匹配来判断一个对象属于 PdaItem 的哪一个子类，并据此返回不同的 Name 值。但这二者相比，显然多态设计更好。因为前者允许从 PdaItem 任意派生出新的子类（甚至可以在不同的程序集中），同时无须重新编译现有代码。相反，后者需要修改模式匹配代码来加入对新类型的支持。不过，多态性也不是万能的。在不适合使用多态性的时候，模式匹配（特别是属性模式匹配）就是一个很好的替代方案。

那么，什么时候不适合使用多态性？第一种情况是在对象层次结构无法满足程序要求的时候。例如，可能需要处理来自多个不相关系统的类。另外，在这种情况下，需要多态性的代码往往不在你的控制范围之内，而且不能修改。在代码清单 7.23 中，对日期的处理便是这样的一个例子。

第二种情况是你要添加的功能并不属于这些类的核心抽象的一部分。例如，向驾驶员收取的高速费会因车辆类型而变，但高速费并不是车辆的核心功能。

> **高级主题：使用 as 操作符进行转换**
>
> 除了 is 操作符，C#还支持一个 as 操作符。相较于 is 操作符，as 操作符的一个优势在于它不仅检查操作数是否为特定类型，还会尝试将其转换为该特定数据类型，并在

源类型本质上（在继承链中）不属于目标类型时赋值 null。除此之外，它相较于强制类型转换还有一个优势，即不会引发异常。代码清单 7.34 展示了如何使用 as 操作符。

代码清单 7. 34　使用 as 操作符进行数据类型转换

```
 public class PdaItem
{
    protected Guid ObjectKey { get; }
    // ...
}

public class Contact : PdaItem
{
    // ...
    public Contact(string name) => Name = name;

    public static Contact Load(PdaItem pdaItem)
    {
        Contact? contact = pdaItem as Contact;
        if (contact is not null)
        {
            Console.WriteLine(
                $"ObjectKey: {contact.ObjectKey}");
            return (Contact)pdaItem;
        }
        else
        {
            throw new ArgumentException(
                $"{nameof(pdaItem)}不属于{nameof(Contact)}类型");
        }
    }
    // ...
}
```

使用 as 操作符可以避免用额外的 try-catch 代码来处理转换无效的情况，因为 as 操作符提供了尝试执行转型但转型失败后不引发异常的一种方式。

is 操作符相较于 as 操作符的一个优点是后者不能成功地判断基础类型。as 能在继承链中向上或向下隐式转型。和 as 操作符不同的是，is 操作符可以判断基础类型。另外，as 操作符主要支持引用类型，而 is 操作符支持所有类型。

更重要的是，as 操作符一般要求采取额外的步骤对被赋值的变量执行 null 检查。由于模式匹配 is 操作符已自动包含了该项检查，所以现在 as 操作符几乎没了用武之地——只要使用的是 C# 7.0 或更高版本。

7.8 小结

本章讨论了如何从一个类派生，并添加额外的方法和属性来“特化”那个类。讨论了如何使用 private 和 protected 访问修饰符控制封装级别。

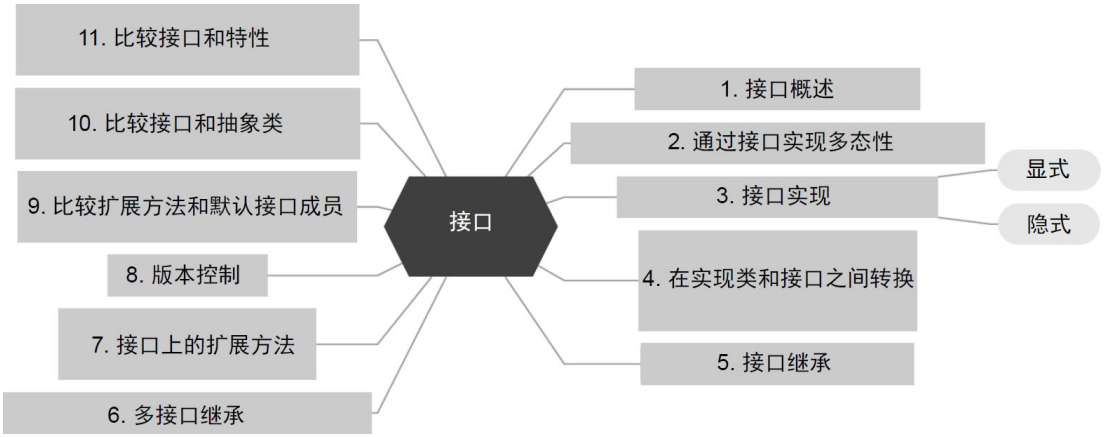
还详细讨论了如何重写基类实现，以及如何使用 new 修饰符隐藏基类实现。C#提供 virtual 修饰符来控制重写，它告诉派生类的程序员需要重写哪些成员。要完全禁止派生，需要为类使用 sealed 修饰符。类似地，为成员使用 sealed 修饰符，会禁止子类继续重写该成员。

本章简单地提到所有类型都从 object 派生。第 10 章将进一步讨论这个问题。届时会讲解 object 的三个虚方法为重写提出的具体规则和原则。但在此之前，首先要掌握在面向对象编程的基础上发展起来的另一种编程模式：接口。这是第 8 章的主题。

本章最后探讨了如何使用 is 操作符以及 switch 语句和 C# 8.0 的 switch 表达式进行模式匹配。虽然 C# 7.0 提供了最初的模式匹配支持，但直到 C# 11，才实现了对这些功能的充分扩展。

第 8 章 接口

在 C#中，并非只能通过继承实现多态性（像第 7 章讨论的那样），还能通过接口实现。和抽象类不同，在 C# 8.0 之前，接口不能包含任何实现。（但即便是在 C# 8.0 中，除非是为了对接口进行“版本控制”，否则这个功能该不该使用都是两说。）但和抽象类相似，接口也定义了一组成员，调用者可以认为这些成员都已实现。



类型通过实现接口来定义其功能。**接口实现关系**是一种“能做”（can do）关系：类型“能做”接口所规定的事情。在“实现接口的类型”和“使用接口的代码”之间，接口订立了一个“契约”[①]。实现接口的类型必须使用接口要求的签名来定义方法。本章首先讨论了接口的实现和使用。最后，我们讨论了接口的默认实现成员，以及这一新特性所引入的多种思维模式（和额外的复杂性）。

8.1 接口概述

> **初学者主题：为什么需要接口**
>
> 接口之所以有用，是因为和抽象类不同，它能将实现细节和所提供的服务完全隔开。接口好比电源插座。电如何输送到插座是实现细节：可能是煤电、核电或太阳能发电；发电机可能在隔壁，也可能在很远的地方。插座订立了“契约”。它以特定频率提供特定电压，要求使用该接口的电器提供兼容的插头。电器不必关心电如何输送到插座，只需提供兼容的插头。

[①] 译注：文档称为“协定”。

> 来看看下面这个例子。目前有许多文件压缩格式，包括.zip，.7-zip，.cab，.lha，.tar，.tar.gz，.tar，.bz2，.bh，.rar，.arj，.arc，.ace，.zoo，.gz，.bzip2，.xxe，.mime，.uue 以及.yenc 等。如果为每种压缩格式都单独创建一个类，那么每个压缩实现都可能有不同的方法签名，无法在它们之间提供标准调用规范。虽然方法可以在基类中声明为抽象成员，但假如都从一个通用基类派生，那么会用掉唯一的基类机会（C#只允许单继承）。不同的压缩实现没什么通用的代码可以放到基类中，这使基类实现变得毫无意义。重点在于，基类除了允许共享成员签名，还允许共享实现。但是，接口只允许共享成员签名，不允许共享实现。
>
> 所以，此时不是共享一个通用基类，而是每个压缩类都实现一个通用的接口。接口订立了契约，类必须履行该契约才能与实现了该接口的其他类交互。虽然存在着多种压缩算法，但假如它们都实现了 `IFileCompression` 接口以及该接口的压缩方法 `Compress()`和解压方法 `Uncompress()`，那么在需要压缩和解压时，只需执行到 `IFileCompression` 接口的一次转型，然后调用其成员方法即可，根本不用关心具体是哪个类在实现那些方法。这就实现了多态性：每个压缩类都有相同的方法签名，但签名的具体实现不同。

代码清单 8.1 展示了示例接口 `IFileCompression`。根据约定（该约定是如此根深蒂固，以至于不好改动），接口名称要采用 PascalCase 大小写规范，并附加一个“`I`”前缀。

代码清单 8.1　定义接口

```
interface IFileCompression
{
    void Compress(string targetFileName, string[] fileList);
    void Uncompress(
        string compressedFileName, string expandDirectoryName);
}
```

`IFileCompression` 定义了一个类为了与其他压缩类协作而必须实现的方法。接口的强大之处在于，调用者可以在不修改调用代码的情况下随便切换不同的实现。

在 C# 8.0 之前，接口的一个关键特征是不包含任何实现和数据（字段）。接口中的方法声明始终用一个分号来取代大括号。至于接口中的属性，虽然看起来像是自动实现的属性，但它没有支持字段。事实上，字段（数据）也不能在接口声明中出现。

从 C# 8.0 开始，关于接口的许多限制被放宽了。其主要目的是让接口在发布之后，仍然可以在一定限度之内做一些改变。本章要到“C# 8.0 和更高版本的接口版本控制”一节才开

始介绍这些新特性。在此之前，我们的重点是基于接口来实现多态性。因为这才是接口最能发挥作用的地方。了解了这些传统知识之后，才能更容易地理解新的语言特性会在什么时候发挥作用。所以让我们暂时不提 C# 8.0，姑且认为接口中不能包含任何数据和实现。等到进入 C# 8.0 的领域之后再学习新规则。

在接口中声明的成员描述了在实现该接口的类型中必须能访问的成员。而所有非公共成员的目的都是阻止其他代码访问成员。因此，C#不允许为接口成员使用访问修饰符；相反，所有成员都自动公共。[①]

设计规范

DO use PascalCasing and an "I" prefix for interface names.

接口名称**要**使用 Pascal 大小写风格，并附加"I"前缀。

8.2 通过接口实现多态性

来看看代码清单 8.2 展示的另一个例子，它的结果如输出 8.1 所示。任何类要通过 ConsoleListControl 类显示，就必须实现 IListable 接口所定义的成员。换言之，实现了 IListable 接口的任何类都可以用 ConsoleListControl 显示它自身。IListable 接口目前只要求只读属性 CellValues。

代码清单 8.2　实现和使用接口

```
public interface IListable
{
    // 返回一行中每个单元格的值
    string?[] CellValues { get; }
}

public abstract class PdaItem
{
    public PdaItem(string name)
    {
        Name = name;
    }

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

[①] 在 C# 8.0 之前。

```
}

public class Contact : PdaItem, IListable
{
    public Contact(string firstName, string lastName,
        string address, string phone)
        : base(GetName(firstName, lastName))
    {
        FirstName = firstName;
        LastName = lastName;
        Address = address;
        Phone = phone;
    }

    public string FirstName { get; }
    public string LastName { get; }
    public string Address { get; }
    public string Phone { get; }
    public static string GetName(string firstName, string lastName)
        => $"{firstName} {lastName}";

    public string[] CellValues
    {
        get
        {
            return new string[]
            {
                FirstName,
                LastName,
                Phone,
                Address
            };
        }
    }

    public static string[] Headers
    {
        get
        {
            return new string[] {
                "First Name", "Last Name    ",
                "Phone       ",
                "Address                        " };
        }
    }
    // ...
}

public class Publication : IListable
{
```

```
    public Publication(string title, string author, int year)
    {
        Title = title;
        Author = author;
        Year = year;
    }

    public string Title { get; }
    public string Author { get; }
    public int Year { get; }

    public string?[] CellValues
    {
        get
        {
            return new string?[]
            {
                Title,
                Author,
                Year.ToString()
            };
        }
    }

    public static string[] Headers
    {
        get
        {
            return new string[] {
                "Title                                                ",
                "Author             ",
                "Year" };
        }
    }

    // ...
}

public class Program
{
    public static void Main()
    {
        Contact[] contacts = new Contact[]
        {
          new(
              "Dick", "Traci",
              "123 Main St., Spokane, WA  99037",
              "123-123-1234"),
          new(
              "Andrew", "Littman",
```

```
                "1417 Palmary St., Dallas, TX 55555",
                "555-123-4567"),
            new(
                "Mary", "Hartfelt",
                "1520 Thunder Way, Elizabethton, PA 44444",
                "444-123-4567"),
            new(
                "John", "Lindherst",
                "1 Aerial Way Dr., Monteray, NH 88888",
                "222-987-6543"),
            new(
                "Pat", "Wilson",
                "565 Irving Dr., Parksdale, FL 22222",
                "123-456-7890"),
            new(
                "Jane", "Doe",
                "123 Main St., Aurora, IL 66666",
                "333-345-6789")
        };

        // 类可以隐式转型为其支持的接口

        ConsoleListControl.List(Contact.Headers, contacts);

        Console.WriteLine();

        Publication[] publications = new Publication[3] {
            new(
                "The End of Poverty: Economic Possibilities for Our Time",
                "Jeffrey Sachs", 2006),
            new("Orthodoxy",
                "G.K. Chesterton", 1908),
            new(
                "The Hitchhiker's Guide to the Galaxy",
                "Douglas Adams", 1979)
            };
        ConsoleListControl.List(
            Publication.Headers, publications);
    }
}

public class ConsoleListControl
{
    public static void List(string[] headers, IListable[] items)
    {
        int[] columnWidths = DisplayHeaders(headers);

        for (int count = 0; count < items.Length; count++)
        {
            string?[] values = items[count].CellValues;
```

```
                DisplayItemRow(columnWidths, values);
            }
        }

        /// <summary>显示列标题</summary>
        /// <returns>返回由列宽构成的一个数组</returns>
        private static int[] DisplayHeaders(string[] headers)
        {
            // ...
        }

        private static void DisplayItemRow(
            int[] columnWidths, string?[] values)
        {
            // ...
        }
    }
```

输出 8.1

```
First Name Last Name    Phone            Address
Dick       Traci        123-123-1234123  Main St., Spokane, WA  99037
Andrew     Littman      555-123-45671417 Palmary St., Dallas, TX 55555
Mary       Hartfelt     444-123-45671520 Thunder Way, Elizabethton, PA 44444
John       Lindherst    222-987-65431    Aerial Way Dr., Monteray, NH 88888
Pat        Wilson       123-456-7890565  Irving Dr., Parksdale, FL 22222
Jane       Doe          333-345-6789123  Main St., Aurora, IL 66666

Title                                                 Author             Year
The End of Poverty: Economic Possibilities for Our Time  Jeffrey Sachs      2006
Orthodoxy                                             G.K. Chesterton    1908
The Hitchhiker's Guide to the Galaxy                  Douglas Adams      1979
```

在代码清单 8.2 中，ConsoleListControl 可以显示看似无关的类（即分别代一联系人和出版物的 Contact 和 Publication 类）。一个类是否能显示，只取决于它是否实现了必要的接口。ConsoleListControl.List()方法依赖多态性正确显示传给它的对象集合。每个类都有自己的 CellValues 实现。将类转型为 IListable 后，就可以调用特定的实现。

8.3 接口实现

声明类来实现接口，这类似于从基类派生——要实现的接口和基类名称以逗号分隔（基类在前，接口顺序任意）。类可以实现多个接口，但只能从一个基类直接派生，如代码清单 8.3 所示。

代码清单 8.3　实现接口

```
public class Contact : PdaItem, IListable, IComparable
{
    // ...

    #region IComparable 成员
    /// <summary>
    ///
    /// </summary>
    /// <param name="obj"></param>
    /// <returns>
    /// 小于零     该实例小于 obj
    /// 零         该实例等于 obj
    /// 大于零     该实例大于 obj
    /// </returns>
    public int CompareTo(object? obj) => obj switch
    {
        null => 1,
        Contact contact when ReferenceEquals(this, obj) => 0,
        Contact { LastName: string lastName }
            when LastName.CompareTo(lastName) != 0 =>
                LastName.CompareTo(lastName),
        Contact { FirstName: string firstName }
            when FirstName.CompareTo(firstName) != 0 =>
                FirstName.CompareTo(firstName),
        Contact _ => 0,
        _ => throw new ArgumentException(
            $"参数不是{ nameof(Contact) }类型的一个值",
            nameof(obj))
    };
    #endregion

    #region IListable 成员
    string?[] IListable.CellValues
    {
        get
        {
            return new string?[]
            {
                FirstName,
                LastName,
                Phone,
                Address
            };
        }
    }
    #endregion

    // ...
}
```

实现接口时，接口的所有（抽象[①]）成员都必须实现。抽象类可以提供接口成员的一个抽象实现。非抽象实现则可以在方法主体中抛出一个 NotImplementedException 异常。总之，无论如何都要为接口成员提供一个“实现”。

接口的一个重要特征是永远不能实例化。换言之，不能用 new 创建接口。因此，接口没有构造函数或终结器。只有实例化实现了接口的类型，才能使用接口实例。另外，接口不能包含静态成员[②]。接口为多态性而生，而假如没有实现接口的那个类型的实例，多态性的意义何在？

每个（未实现的[③]）接口成员都是抽象的，所有派生类都必须实现它。因此，不需要也不可能为接口成员显式添加 abstract 修饰符[④]。

在类型中实现接口成员时有两种方式：**显式**和**隐式**。之前在实现 IComparable 的 CompareTo 接口时，采用的是隐式实现，是用当前类型的 public 成员来实现接口成员。

8.3.1 显式成员实现

显式实现的方法只能通过接口本身调用，最典型的做法是将对象转型为接口。例如，在代码清单 8.4 中，是先将 Contact 对象转型为 IListable，然后调用 Contact 类显式实现的 CellValues 成员。

代码清单 8.4　调用显式接口成员实现

```
string?[] values;
Contact contact = new("Inigo Montoya");
// ...

// 错误：不能在 contact 上直接调用 CellValues
// values = contact.CellValues;

// 应首先转型为 IListable
values = ((IListable)contact).CellValues;
```

[①] 向接口添加非抽象成员的能力是从 C# 8.0 开始才有的。本章最后才会解释，目前暂时忽略。

[②] 在 C# 8.0 之前。

[③] 已实现的成员只有在 C# 8.0 或更高版本中才可以有。

[④] 在 C# 8.0 之前。

```
// ...
```

本例是在同一个语句中执行强制类型转换和调用 CellValues。但是，完全可以在调用 CellValues 之前先将 contact 赋给一个 IListable 变量。

为了显式实现接口成员，需要在接口成员名称前附加接口名称前缀，如代码清单 8.5 所示。

代码清单 8.5　显式接口实现

```
public class Contact : PdaItem, IListable, IComparable
{
    // ...
    #region IListable 成员
    string?[] IListable.CellValues
    {
        get
        {
            return new string?[]
            {
                FirstName,
                LastName,
                Phone,
                Address
            };
        }
    }
    #endregion
    // ...
}
```

代码清单 8.5 通过为属性名附加 IListable 前缀来显式实现 CellValues。此外，由于显式接口实现直接与接口关联，所以没必要使用 virtual、override 或者 public 来修饰它们。事实上，这些修饰符是不被允许的。这些成员不被视为类的公共成员，标注 public 有误导之嫌。

注意，虽然不允许用 override（重写）关键字来修饰接口，但在实现由接口定义的签名时，仍然可以把这个过程称为“重写”。

8.3.2 隐式成员实现

在代码清单 8.3 中，注意 CompareTo()没有附加 IComparable 前缀，所以该成员是隐式实现

的。要隐式实现成员，只要求成员是公共的，且签名与接口成员签名相符。接口成员实现不要求使用 override 关键字或者其他任何表明该成员与接口关联的指示符。此外，由于成员像其他类成员那样声明，所以可以像调用其他类成员那样直接调用隐式实现的成员：

```
result = contact1.CompareTo(contact2);
```

换言之，隐式成员实现不要求执行强制类型转换，因为成员能直接调用，没有在实现它的类型中被隐藏起来。[①]

显式实现不允许的许多修饰符对于隐式实现都是必须或可选的。例如，隐式成员实现必须是 public 的。还可以选择添加一个 virtual，但具体要取决于是否允许派生类重写实现。如果去掉 virtual，将导致成员被密封。

8.3.3 比较显式与隐式接口实现

隐式和显式接口成员实现之间的主要区别并不在于方法声明的语法，而在于是否能通过类型的实例直接访问该方法。对于显式实现，只能通过接口访问该方法。

建立类层次结构时，需要建模真实世界的“属于”（is a）关系。例如，长颈鹿“属于”哺乳动物。这些是“语义”（semantic）关系。而接口用于建模“机制”（mechanism）关系。PdaItem “不属于”一种“可比较”（comparable）的东西，但它仍然可以实现 IComparable 接口。该接口与语义模型无关，只是实现机制的细节。显式接口实现的目的就是将“机制问题”和“模型问题”分开。要求调用者先将对象转换为接口（比如 IComparable），然后才能认为对象“可比较”，从而显式区分你想在什么时候与模型沟通，以及想在什么时候处理实现机制。

一般来说，最好的做法是将一个类的公共层面限制成“全模型”，尽量少地涉及无关的机制。遗憾的是，有的机制在.NET 中是不可避免的。例如，在现实世界中不能获得长颈鹿的哈希码，或者将长颈鹿转换成字符串。但在.NET 中，可以获得 Giraffe（长颈鹿）类的哈希码（GetHashCode()），并把它转换成字符串（ToString()）。将 object 作为通用基类，.NET 混合了模型代码和机制代码——即使混合程度仍然比较有限。

可通过回答以下问题来决定显式还是隐式实现。

- **成员是不是核心的类功能？**
  以 Contact 类的 CellValues 属性实现为例。该成员并非 Contact 类型的一个密不可分的部分，仅仅是辅助成员，可能只有 ConsoleListControl 类才会访问它。所以没必要把它设计成 Contact 对象的一个直接可见的成员，使本来就很庞大的成员列表变

[①] 译注：相反，显式实现的接口成员在当前类中会被隐藏，即这个实现在该类的外部不可见，不能直接访问。只有当对象被视为其接口类型时，这种成员才可以访问。

得更拥挤。

作为一个反例，再来看看 IFileCompression.Compress()成员。在 ZipCompression 类中包含隐式的 Compress()实现是一个非常合理的选择，因为 Compress()是 ZipCompression 类的核心功能，所以应当能从 ZipCompression 类直接访问。

- **接口成员名称作为类成员名称是否恰当？**

  假定 ITrace 接口的 Dump()成员将类的数据写入跟踪日志。在 Person 或者 Truck（卡车）类中隐式实现 Dump()会混淆该方法的作用[①]。所以，更好的选择是显式实现，确保只能通过 ITrace 数据类型调用 Dump()，使该方法不会产生歧义。总之，假如成员的用途在实现类中不明确，就考虑显式实现。

- **是否已经有相同签名的类成员？**

  显式接口成员实现不会在类型的声明空间添加具名元素。所以，如果类型已存在可能冲突的成员，那么显式接口成员可与之同名、同签名。

大多数时候，我们都是凭直觉选择隐式还是显式接口成员实现。但在选择时参考上述问题可做出更稳妥的选择。由于从隐式变成显式会造成版本中断，因此较稳妥的做法是全部显式实现接口成员，使它们以后都能安全地变成隐式。另外，隐式还是显式不需要在所有接口成员之间保持一致，所以完全可以将部分成员定义成显式，将其他定义成隐式。

8.4 在实现类和接口之间转换

类似于派生类和基类的关系，实现类也可以隐式转换为接口，无需转型操作符。实现类的实例总是包含接口的全部成员，所以总是能成功转换为接口类型。

虽然从实现类型向接口的转换总是成功，但可能有多个类型实现了同一个接口。所以，无法保证从接口向实现类型的向下转型能成功。接口必须显式转型为它的某个实现类型。

8.5 接口继承

一个接口可以从另一个接口派生，派生的接口将继承“基接口”的所有成员[②]。如代码清单 8.6 所示，直接从 IReadableSettingsProvider 派生的接口是显式基接口。

[①] 译注：Dump 本来的意思是“转储”类的数据，但用于 Truck 类会把它同“卸货”联系起来，从而造成混淆。

[②] 从 C# 8.0 开始引入的非 private 成员除外。

代码清单 8.6　从一个接口派生出另一个接口

```
interface IReadableSettingsProvider
{
    string GetSetting(string name, string defaultValue);
}

interface ISettingsProvider : IReadableSettingsProvider
{
    void SetSetting(string name, string value);
}

public class FileSettingsProvider : ISettingsProvider
{
    #region ISettingsProvider 成员
    public void SetSetting(string name, string value)
    {
        // ...
    }
    #endregion

    #region IReadableSettingsProvider 成员
    public string GetSetting(string name, string defaultValue)
    {
        // ...
    }
    #endregion
}
```

本例的 ISettingsProvider 从 IReadableSettingsProvider 派生，所以会继承其成员。如后者还有一个显式基接口，那么 ISettingsProvider 也会继承其成员。总之，派生层次结构中的完整接口集合直接就是所有基接口的一个累积。

有趣的是，假如显式实现 GetSetting()，那么必须通过 IReadableSettingsProvider 进行。在代码清单 8.7 中，通过 ISettingsProvider 进行将无法编译，如输出 8.2 所示。

代码清单 8.7　显式实现接口成员，但未提供正确的包容接口

```
// 错误:  GetSetting()在 ISettingsProvider 上不可用
string ISettingsProvider.GetSetting(
    string name, string defaultValue)
{
    // ...
```

```
}
```

编译代码清单 8.7，会获得如输出 8.2 所示的错误信息。

输出 8.2

```
显式接口声明中的“ISettingsProvider.GetSetting”不是接口的成员。
```

伴随这个输出，还有一条错误信息指出 IReadableSettingsProvider.GetSetting()尚有实现。显式实现接口成员时，必须在完全限定的接口成员名称中引用最初声明它的那个接口的名称。

即使类实现的是从基接口（IReadableSettingsProvider）派生的接口（ISettingsProvider），仍然可明确声明自己要实现这两个接口，如代码清单 8.8 所示。

代码清单 8.8　在类声明中使用基接口

```
public class FileSettingsProvider : ISettingsProvider,
    IReadableSettingsProvider
{
    #region ISettingsProvider 成员
    public void SetSetting(string name, string value)
    {
        // ...
    }
    #endregion

    #region IReadableSettingsProvider 成员
    public string GetSetting(string name, string defaultValue)
    {
        // ...
    }
    #endregion
}
```

在这个代码清单中，类的接口实现并无变化。虽然在类的 header 中提供额外的接口实现声明，这多少显得多余，但它带来了更好的可读性。

提供多个接口，而不是单独提供一个复合接口，这个决策在很大程度上依赖于接口设计者对实现类（implementing class）有什么要求。提供一个 IReadableSettingsProvider 接口，设计者告诉实现者只需实现一个用于检索（读取）设置的 setting provider，不需要实现向设置的写入，从而避免了复杂性，减轻了实现者的负担。

相反，但凡要实现 ISettingsProvider，任何类都不可能只支持写入而不支持读取。所以，ISettingsProvider 和 IReadableSettingsProvider 之间的*继承*关系强迫 FileSettingsProvider 类同时实现这两个接口。

最后要强调的是，虽然“继承”这个词用得没错，但更准确的说法是接口代表契约，而在一份契约中，可以指定另一份契约也必须遵守的条款。所以，ISettingsProvider : IReadableSettingsProvider 从概念上是说 ISettingsProvider 契约还要求遵守 IReadableSettingsProvider 契约，而不是说 ISettingsProvider “属于一种” IReadableSettingsProvider。话虽如此，但为了和标准的 C#术语保持一致，本章剩余部分仍会使用继承关系术语。

8.6 多接口继承

就像类能实现多个接口那样，接口也能从多个接口继承，而且语法和类的继承/实现语法一致，如代码清单 8.9 所示。

代码清单 8.9　多接口继承

```
interface IReadableSettingsProvider
{
    string GetSetting(string name, string defaultValue);
}

interface IWriteableSettingsProvider
{
    void SetSetting(string name, string value);
}

interface ISettingsProvider : IReadableSettingsProvider,
    IWriteableSettingsProvider
{
}
```

很少出现无任何成员的接口。但是，假如要求同时实现两个接口，这就是一种合理的选择。代码清单 8.9 和代码清单 8.6 的区别在于，现在可以在不提供任何读取功能的前提下实现 IWriteableSettingsProvider。代码清单 8.6 的 FileSettingsProvider 不受影响，但假如它使用的是显式成员实现，就要以稍微不同的方式指定成员从属于哪个接口。

8.7 接口上的扩展方法

扩展方法的一个重要特点是除了能作用于类，还能作用于接口。语法和作用于类时一样。方法第一个参数是要扩展的接口，该参数必须附加 this 修饰符。代码清单 8.10 展示了在 Listable 类上声明、作用于 IListable 接口的一个扩展方法。

代码清单 8.10　接口扩展方法

```
public class Program
{
    public static void Main()
    {
        Contact[] contacts = new Contact[] {
            new(
                "Dick", "Traci",
                "123 Main St., Spokane, WA  99037",
                "123-123-1234")
            // ...
        };

        // 类隐式转换为它们支持的接口
        contacts.List(Contact.Headers);

        Console.WriteLine();

        Publication[] publications = new Publication[3] {
            new("The End of Poverty: Economic Possibilities for Our Time",
                "Jeffrey Sachs", 2006),
            new("Orthodoxy",
                "G.K. Chesterton", 1908),
            new(
                "The Hitchhiker's Guide to the Galaxy",
                "Douglas Adams", 1979)
            };
        publications.List(Publication.Headers);
    }
}

public static class Listable
{
    public static void List(
        this IListable[] items, string[] headers)
    {
        int[] columnWidths = DisplayHeaders(headers);

        for (int itemCount = 0; itemCount < items.Length; itemCount++)
```

```
        {
            if (items[itemCount] is not null)
            {
                string?[] values = items[itemCount].CellValues;

                DisplayItemRow(columnWidths, values);
            }
        }
    }
    // ...
}
```

注意本例被扩展的不是 IListable（虽然也可以），而是 IListable[]。这证明 C#不仅能为特定类型的实例添加扩展方法，还允许为该类型的对象集合添加。对扩展方法的支持是实现语言集成查询（Language Integrated Query，LINQ）的基础。IEnumerable 是所有集合都要实现的基本接口。通过为 IEnumerable 定义扩展方法，所有集合都能享受 LINQ 支持。这显著改变了对象集合的编程方式，该主题将在第 15 章详细讨论。

> **初学者主题：接口图示**
>
> 在 UML 风格的图中[①]，接口可能有两种形式。第一种，可以将接口显示成与类继承相似的继承关系。在图 8.1 中，IPerson 和 IContact 之间的关系就是这样的。第二种，可以使用小圆圈显示接口，一般将这种小圆圈称为“棒棒糖”（lollipop），如图 8.1 的 IPerson 和 IContact 所示。
>
> 在图 8.1 中，Contact 从 PdaItem 派生并实现 IContact。此外，它还聚合了 Person 类，后者实现了 IPerson。虽然 Visual Studio 类设计器不支持，但接口有时也可以使用与派生关系相似的箭头来显示。例如，在图 8.1 中，Person 可以连接一条箭头线到 IPerson，而不是画一个“棒棒糖”来表示。

[①] UML 是 Unified Modeling Language（统一建模语言）的简称，是用图来建模对象设计的一个标准规范。

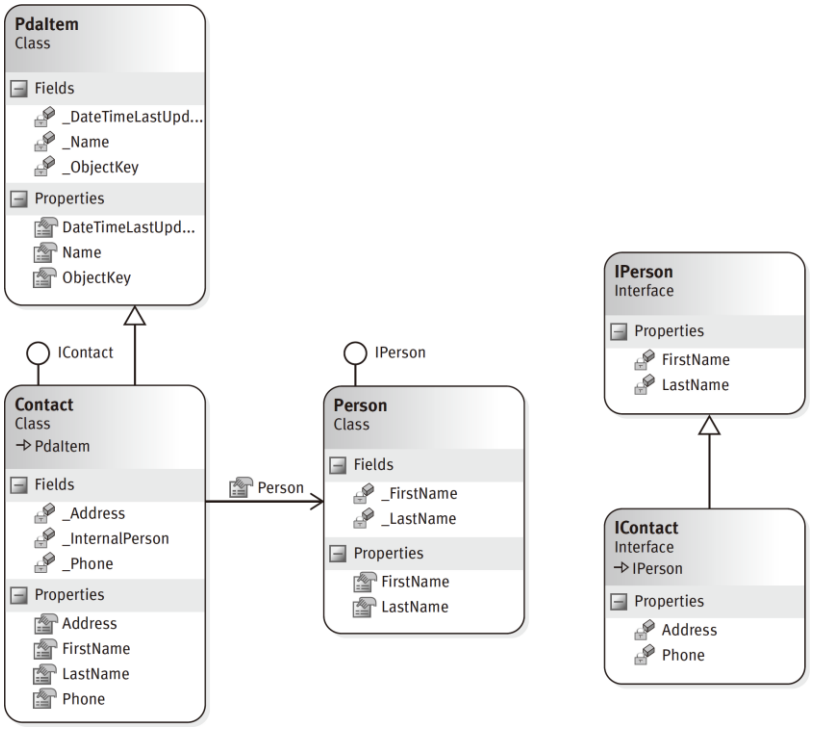


图 8.1 通过聚合和接口解决单继承限制

8.8 版本控制

在 C# 8.0 之前，如果创建的组件或应用程序是供其他开发人员使用的，那么在创建新版本的时候不应更改接口。接口在实现接口的类和使用接口的类之间订立了契约，修改接口相当于修改契约，会使基于接口写的代码失效。

更改或删除特定接口成员的签名显然会造成现有代码的中断，因为除非现有的代码同步修改，否则对该成员的任何调用都不再能够编译。更改类的 `public` 或 `protected` 成员签名也会这样。但和类不同，在接口中添加成员也可能造成代码无法编译——除非进行额外的修改。问题在于，实现接口的任何类都必须完整地实现，必须提供针对所有成员的实现。添加新接口成员后，编译器会要求开发人员在实现接口的类中添加新的接口成员。

从 C# 8.0 开始，“不应更改接口”规则略有变化。C# 8.0 允许在接口中为成员提供默认实现，这样就可以在接口中添加成员，同时不会在现有实现上触发编译器错误（虽然仍然不能以版本兼容的方式删除或修改现有成员）。在 C# 8.0 之前，要达到类似于更改接口的结果，只能通过添加额外的接口来实现。本节将同时讨论这两种方法。

设计规范

DO NOT add members without a default implementation to a published interface.

不要为已发布的接口添加没有默认实现的成员。

8.8.1 C# 8.0 之前的接口版本控制

来看看代码清单 8.11 的 IDistributedSettingsProvider 接口，它很好地演示了如何以一种版本兼容的方式扩展现有接口。假定最开始只定义了 ISettingsProvider 接口（如代码清单 8.6 所示）。但新版本要求设置能分布到多个资源（URI[1]），或许每台机器都单独保存一套。为此，我们创建了 IDistributedSettingsProvider，它从 ISettingsProvider 派生。

代码清单 8.11　一个接口从另一个派生

```
interface IDistributedSettingsProvider : ISettingsProvider
{
    /// <summary>
    /// 获取特定 URI 的设置
    /// </summary>
    /// <param name="uri">
    /// 和设置关联的 URI 名称</param>
    /// <param name="name">设置名称</param>
    /// <param name="defaultValue">
    /// 在设置未找到的前提下返回的值</param>
    /// <returns>指定的设置</returns>
    string GetSetting(
        string uri, string name, string defaultValue);

    /// <summary>
    /// 为特定 URI 进行设置
    /// </summary>
    /// <param name="uri">
    /// 和设置关联的 URI 名称</param>
    /// <param name="name">设置名称</param>
    /// <param name="value">要持久化存储的值</param>
    /// <returns>指定的设置</returns>
    void SetSetting(
        string uri, string name, string value);
}
```

[1] URI 是 Universal Resource Identifier 的简称。

该设计的重点在于，其他实现了 ISettingsProvider 的程序员可选择升级自己的实现来包含 IDistributedSettingsProvider，也可选择忽略它。

但是，如果不是新建接口，而是在现有的 ISettingsProvider 接口中添加与 URI 相关的方法，那么在新的接口定义下，实现了该接口的类就不再能成功编译。这个改变造成了版本中断——不管是在二进制级别上，还是在源代码级别上。

开发阶段当然能随便更改接口，虽然开发人员可能要为此付出不少劳动（为了面面俱到）。但发布了就不要更改。相反，应创建第二个接口（可从原始接口派生）。注意，代码清单 8.11 使用了对接口成员进行描述的 XML 注释，第 10 章会详细解释这种注释。

8.8.2 C# 8.0 之后的接口版本控制

到目前为止，我们只提到了 C# 8.0 对接口的功能扩展，但尚未做具体介绍。本节将介绍一个新概念：**默认接口成员**。如前所述，在 C# 8.0 之前，对已发布的接口做任何更改都会导致基于该接口的程序中断。也就是说，接口一旦发布就不可更改。但是，从 C# 8.0 开始，微软引入了一个新的语言特性，允许在接口中加入成员的实现。换言之，是具体的成员，而非只是声明。代码清单 8.12 的 CellColors 属性展示了一个例子。

代码清单 8.12　用默认接口成员对接口进行版本控制

```
public interface IListable
{
    // 返回一行中每个单元格的值
    string?[] CellValues
    {
        get;
    }

    ConsoleColor[] CellColors
    {
        get
        {
            var result = new ConsoleColor[CellValues.Length];
            // 使用泛型 Array 方法来填充数组
            // (参见第 12 章)
            Array.Fill(result, DefaultColumnColor);
            return result;
        }
    }
    static public ConsoleColor DefaultColumnColor { get; set; }
}
```

```
// ...

public class Contact : PdaItem, IListable
{
    // ...

    #region IListable
    string[] IListable.CellValues
    {
        get
        {
            return new string[]
            {
                FirstName,
                LastName,
                Phone,
                Address
            };
        }
    }
    // *** 未提供 CellColors 的实现 *** //
    #endregion IListable

    // ...
}

public class Publication : IListable
{
    // ...
    #region IListable
    string?[] IListable.CellValues
    {
        get
        {
            return new string[]
            {
                Title,
                Author,
                Year.ToString()
            };
        }
    }

    ConsoleColor[] IListable.CellColors
    {
        get
        {
            string?[] columns = ((IListable)this).CellValues;
            ConsoleColor[] result = ((IListable)this).CellColors;
```

```
            if (columns[YearIndex]?.Length != 4)
            {
                result[YearIndex] = ConsoleColor.Red;
            }
            return result;
        }
    }
    #endregion IListable
    // ...
}
```

注意，本例添加了 `CellColors` 属性的取值方法（getter）。如你所见，尽管它是接口的成员，但还是包括了一个具体的实现。这个语言特性称为**默认接口成员**，因为它为该方法提供了一个默认实现。这样一来，任何实现该接口的类都会拥有一个默认实现——以后即使接口有了额外的成员，只要这些成员提供了默认实现，代码就会继续编译而无需任何更改。例如，`Contact` 类就没有为 `CellColors` 属性的 getter 提供实现，所以它依赖于 `IListable` 接口提供的默认实现。

当然，完全可以在实现类中重写方法的默认实现，以提供更适合该类的不同行为。这种行为与实现本章开头概述的多态性的目的是一致的。

然而，默认接口成员还具有额外的特性。其主要目的是支持默认接口成员的重构（尽管有些人会对这种解释有所争议）。将它们用于其他任何目的，都表明代码结构可能存在缺陷，因为这意味着接口被用于除了“实现多态性”以外的其他目的。表 8.1 列出了额外的语言构造及其存在的一些重要限制。[①]

表 8.1 默认接口重构特性

C# 8.0 引入的接口构造	示例代码
静态成员 可以在接口上定义静态成员，包括字段、构造函数和方法（甚至可以定义一个静态 `Main` 方法，即程序的入口点）。接口上的静态成员的默认访问可访问性是 `public`。	`public interface ISampleInterface` `{` `  private static string? _Field;` `  public static string? Field` `  {` `  get => _Field;` `  private set => _Field = value;` `  }` `  static ISampleInterface() =>`

[①] 译注：完整表格已在本书配套资源中提供，详情访问 *https://bookzhou.com* 或本书 GitHub 项目。

	`` Field = "Nelson Mandela"; public static string? GetField() => Field; }```
已实现的实例属性和方法 可以在接口上定义已实现的属性和成员。由于不支持实例字段，所以属性不能依赖于支持字段。另外，也不支持自动实现的属性。 注意，为了访问默认实现的属性，必须把它转型为包含该成员的接口。在右边的例子中，除非 Person 类实现 IPerson 类规定的属性（名字、姓氏等），否则不可以使用默认接口成员 Name。	```public interface IPerson { // 标准的抽象属性定义 string FirstName { get; set; } string LastName { get; set; } // 已实现的实例属性和方法 public string Name => GetName(); public string GetName() => $"{FirstName} {LastName}"; } public class Person : IPerson { // ... } public class Program { public static void Main() { Person inigo = new Person("Inigo", "Montoya"); Console.Write(((IPerson)inigo).Name); } }```
public 访问修饰符 这是所有实例接口成员的默认可访问性。人为添加此关键字，有助于澄清代码的可访问性。但要注意，无论是否有 public 访问修饰符，编译器生成的 CIL 代码都是一样的。	```public interface IPerson { // 所有成员默认 public string FirstName { get; set; } public string LastName { get; set; } string Initials => $"{FirstName[0]}{LastName[0]}"; public string Name => GetName(); public string GetName() => $"{FirstName} {LastName}"; }```
protected 访问修饰符 只能从当前接口或派生接口中访问。	```public interface IPerson { // ... protected void Initialize() => { /* ... */ }; }```

private 访问修饰符

private 访问修饰符限制成员仅在声明它的接口中可用。它设计用于支持默认接口成员的重构。所有私有成员必须包括一个实现。

```
public interface IPerson
{
    string FirstName { get; set; }
    string LastName { get; set; }
    string Name => GetName();
    private string GetName() =>
        $"{FirstName} {LastName}";
}
```

internal 访问修饰符

internal 成员只在声明它们的同一程序集内可见。

```
public interface IPerson
{
    string FirstName { get; set; }
    string LastName { get; set; }
    string Name => GetName();
    internal string GetName() =>
        $"{FirstName} {LastName}";
}
```

protected internal 访问修饰符

protected internal 成员是 protected 和 internal 的超集，其可见性限于同一程序集，以及从包容接口派生的其他接口。类似于 protected 成员，程序集外部的类看不到 protected internal 成员。

```
public interface IPerson
{
    string FirstName { get; set; }
    string LastName { get; set; }
    string Name => GetName();
    protected internal string GetName() =>
        $"{FirstName} {LastName}";
}
```

private protected 访问修饰符

只有在当前接口或派生接口中才能访问 private protected 成员。即使实现了接口的类也无法访问 private protected 成员。在右侧的 Person 类中，PersonTitle 属性就是一个错误示范。

```
class Program
{
    static void Main()
    {
        IPerson? person = null;
        // 实现接口的类不能调用
        // private protected 接口成员
        _ = person?.GetName();
        Console.WriteLine(person);
    }
}
public interface IPerson
{
    string FirstName { get; }
    string LastName { get; }
    string Name => GetName();
    private protected string GetName() =>
        $"{FirstName} {LastName}";
}

public interface IEmployee : IPerson
```

	{ int EmployeeId => GetName().GetHashCode(); } public class Person : IPerson { public Person(string firstName, string lastName) { FirstName = firstName ?? throw new ArgumentNullException(nameof(firstName)); LastName = lastName ?? throw new ArgumentNullException(nameof(lastName)); } public string FirstName { get; } public string LastName { get; } // private protected 接口成员不能在实现 // 该接口的类中访问。 public string PersonTitle => GetName().ToUpper(); }
virtual 修饰符 接口中已实现的成员默认 virtual，这意味着当调用接口成员时，会调用该方法具有相同签名的派生实现（如果有的话）。但是，和 public 访问修饰符一样，可以显式地将一个成员标记为 virtual，从而增强可读性。未实现的接口成员不允许使用 virtual 修饰符。和其他时候一样，virtual 修饰符与 private，static 和 sealed 修饰符不兼容。	public interface IPerson { // 未提供实现的接口成员不允许 virtual string FirstName { get; set; } string LastName { get; set; } virtual string Name => GetName(); private string GetName() => $"{FirstName} {LastName}"; }
sealed 修饰符 为了防止实现接口的类重写方法，可以将其标记为 sealed，确保实现类无法修改方法实现。	public interface IWorkflowActivity { // 私有，所以非虚 private static void Start() => Console.WriteLine("IWorkflowActivity.Start()..."); // 密封以防止重写 sealed void Run()

	`` { try { Start(); InternalRun(); } finally { Stop(); } } protected void InternalRun(); // 私有，所以非虚 private static void Stop() => Console.WriteLine("IWorkflowActivity.Stop().."); } ```
abstract 修饰符 只有未提供实现的成员才可以使用 abstract 修饰符，但写等于没写，因为这种成员默认抽象。所有抽象成员都自动 virtual；将抽象成员显式声明为 virtual 会引发编译错误。	``` public interface IPerson { // 没有实现的成员可以 abstract abstract string FirstName { get; set; } string LastName { get; set; } // 有实现的成员不允许 abstract string Name => GetName(); private string GetName() => $"{FirstName} {LastName}"; } ```
分部接口和分部方法 现在可以提供方法的分部（partial）实现，这些方法不能有输出数据（return 或 ref/out），并可选择在同一接口的第二个声明中提供完全实现的方法。分部方法总是 private——它们不支持访问修饰符。	``` public partial interface IThing { string Value { get; protected set; } void SetValue(string value) { AssertValueIsValid(value); Value = value; } static partial void AssertValueIsValid(string value); } public partial interface IThing { static partial void AssertValueIsValid(string value) { ```

	`` // 值无效就抛出异常 switch (value) { case null: throw new ArgumentNullException(nameof(value)); case "": throw new ArgumentException("空串无效", nameof(value)); case string _ when string.IsNullOrWhiteSpace(value): throw new ArgumentException("不能为空白字符", nameof(value)); }; } } ```

表 8.1 要注意几点。首先，接口不支持自动实现的属性，因为不支持实例字段（自动实现属性必需的支持字段）。这与抽象类形成了显著区别，后者支持实例字段和自动实现的属性。

其次，接口上的静态成员也默认公共。虽然静态成员必须有一个实现，而且在这方面完全相当于类的静态成员，但与之相反的是，接口实例成员的目的是支持多态性，因此它们默认 public。

8.8.3 受保护接口成员提供了额外的封装和多态性

创建类的时候，程序员应谨慎选择是否允许方法被重写，因为他们无法控制派生类的实现。虚方法不应包含关键代码，因为一旦派生类重写了这些方法，这些代码可能永远得不到调用。

代码清单 8.13 展示了一个名为 Run()的虚方法。如果 WorkflowActivity 的程序员希望在调用 Run()的时候，关键的 Start()和 Stop()方法也得到调用，那么希望可能落空，Run()方法可能失败。

代码清单 8.13　鲁莽地依赖虚方法实现

```
public class WorkflowActivity
{
    private static void Start()
```

```
    {
        // 关键代码
    }
    public virtual void Run()
    {
        Start();
        // 做某事...
        Stop();
    }
    private static void Stop()
    {
        // 关键代码
    }
}
```

在重写 Run()方法时，开发者完全可能不会调用关键的 Start()和 Stop()方法。

现在，让我们考虑一个完全实现了此场景的版本，它具有以下封装要求。

- 不允许重写 Run()。
- 不允许调用 Start()或 Stop()，因其执行顺序完全受其包容类型（我们将其命名为 IWorkflowActivity）的控制。
- 允许替换“做某事…”代码块中执行的内容。
- 如果重写 Start()和 Stop()有合理的理由，那么实现它们的类就不应调用它们——它们成了基实现的一部分。
- 允许派生类型提供一个 Run()方法，但在 IWorkflowActivity 上调用 Run()时，不应调用该派生类型的 Run()。

为了满足所有这些要求，C# 8.0 提供了对 protected 接口成员的支持，它和类的 protected 成员存在一些显著的差异。代码清单 8.14 演示了这些差异，输出 8.3 展示了结果。

代码清单 8.14　强制按要求封装 Run()

```
public interface IWorkflowActivity
{
    // 私有，因此非虚
    private static void Start() =>
        Console.WriteLine(
            "IWorkflowActivity.Start()...");

    // 密封以防止重写
    sealed void Run()
    {
```

```
        try
        {
            Start();
            InternalRun();
        }
        finally
        {
            Stop();
        }
    }

    protected void InternalRun();

    // 私有，因此非虚
    private static void Stop() =>
        Console.WriteLine(
            "IWorkflowActivity.Stop()...");
}

public interface IExecuteProcessActivity : IWorkflowActivity
{
    protected void RedirectStandardInOut() =>
        Console.WriteLine(
            "IExecuteProcessActivity.RedirectStandardInOut()...");

    // 密封的不允许重写
    void IWorkflowActivity.InternalRun()
    {
        RedirectStandardInOut();
        ExecuteProcess();
        RestoreStandardInOut();
    }
    protected void ExecuteProcess();
    protected void RestoreStandardInOut() =>
        Console.WriteLine(
            "IExecuteProcessActivity.RestoreStandardInOut()...");
}

public class ExecuteProcessActivity : IExecuteProcessActivity
{
    public ExecuteProcessActivity(string executablePath) =>
        ExecutableName = executablePath
            ?? throw new ArgumentNullException(nameof(executablePath));

    public string ExecutableName { get; }

    void IExecuteProcessActivity.RedirectStandardInOut() =>
        Console.WriteLine(
            "ExecuteProcessActivity.RedirectStandardInOut()...");
```

```
        void IExecuteProcessActivity.ExecuteProcess() =>
            Console.WriteLine(
                $"ExecuteProcessActivity.IExecuteProcessActivity.ExecuteProcess()...");

        public static void Run()
        {
            ExecuteProcessActivity activity = new("dotnet");
            // 受保护成员不可由实现类调用，
            // 即使该成员是在类中实现的。
            // ((IWorkflowActivity)this).InternalRun();
            //  activity.RedirectStandardInOut();
            //  activity.ExecuteProcess();
            Console.WriteLine(
                @$"正在用进程'{activity.ExecutableName}'执行非多态性的 Run()。");
        }
    }

    public class Program
    {
        public static void Main()
        {
            ExecuteProcessActivity activity = new("dotnet");

            Console.WriteLine(
                "正在调用((IExecuteProcessActivity)activity).Run()...");
            // 输出：
            // 正在调用((IExecuteProcessActivity)activity).Run()...
            // IWorkflowActivity.Start()...
            // ExecuteProcessActivity.RedirectStandardInOut()...
            // ExecuteProcessActivity.IExecuteProcessActivity.
            // ExecuteProcess()...
            // IExecuteProcessActivity.RestoreStandardInOut()...
            // IWorkflowActivity.Stop()..
            ((IExecuteProcessActivity)activity).Run();

            // 输出：
            // 正在调用 activity.Run()...
            // 正在用进程'dotnet'执行非多态性的 Run()。
            Console.WriteLine();
            Console.WriteLine(
                "正在调用 activity.Run()...");
            ExecuteProcessActivity.Run();
        }
    }
```

输出 8.3

```
正在调用((IExecuteProcessActivity)activity).Run()...
IWorkflowActivity.Start()...
ExecuteProcessActivity.RedirectStandardInOut()...
ExecuteProcessActivity.IExecuteProcessActivity.ExecuteProcess()...
IExecuteProcessActivity.RestoreStandardInOut()...
```

```
IWorkflowActivity.Stop()...

正在调用 activity.Run()...
正在用进程'dotnet'执行非多态性的 Run()。
```

注意，IWorkflowActivity.Run() 被标记为 sealed，因此非虚。这阻止了任何派生类型更改其实现。给定一个 IWorkflowActivity 类型，在它上面的任何 Run()调用将始终执行 IWorkflowActivity 的实现。

IWorkflowActivity 的 Start()和 Stop()方法是私有的，所以对所有其他类型都不可见。尽管 IExecuteProcessActivity 看似有启动/停止类型的活动，但 IWorkflowActivity 并不允许替换其实现。

IWorkflowActivity 定义了一个受保护的 InternalRun()方法，允许 IExecuteProcessActivity（以及 ExecuteProcessActivity，如果需要）重载它。但要注意的是，ExecuteProcessActivity 的任何成员都不能调用 InternalRun()。可能这个方法永远不应该脱离 Start()和 Stop()的顺序运行，所以只有层次结构中的接口（IWorkflowActivity 或 IExecuteProcessActivity）被允许调用受保护成员。

所有受保护接口成员都可以重写任何默认接口成员，前提是要显式地进行。例如，ExecuteProcessActivity 所实现的 RedirectStandardInOut()和 ExecuteProcess()都显式添加了 IExecuteProcessActivity 前缀。另外，与受保护的 InternalRun()方法一样，实现接口的类型不能调用受保护成员；例如，不能在 ExecuteProcessActivity 类中调用 RedirectStandardInOut()和 ExecuteProcess()，尽管它们是在该类中实现的。

RedirectStandardInOut()和 RestoreStandardInOut()虽然没有显式声明为 virtual，但它们其实都是虚方法（成员除非密封，否则均默认为虚）。因此，当 IExecuteProcessActivity.InternalRun()调用 RedirectStandardInOut()时，调用的是 ExecuteProcessActivity 的实现，而非 IExecuteProcessActivity 的实现。

实现接口的类型可能提供与接口中的 sealed 签名匹配的方法。例如，假定 ExecuteProcessActivity 提供了与 IWorkflowActivity 中的 sealed Run()签名匹配的同名 Run()，那么会执行与该类型关联的实现。换言之，Program.Main()在调用((IExecuteProcessActivity)activity).Run()的时候，调用的是 IExecuteProcessActivity.Run()，而 activity.Run()调用的是 ExecuteProessActivity.Run()——其中 activity 是 ExecuteProcessActivity 类型。

总的来说，使用受保护接口成员以及其他成员修饰符提供的封装，我们可以获得一个全面的封装机制——尽管这肯定有点复杂就是了。

8.9 比较扩展方法和默认接口成员

需要向一个已经发布了的接口添加功能时，相比扩展方法和派生出一个新接口，什么情况下选择添加默认接口方法更好？要对此做出明智的选择，需要考虑以下因素。

- 通过在接口的实例上实现具有相同签名的方法，都可以做到的概念上“重写”。
- 扩展方法可以在任何程序集中实现，并不局限于接口所在的程序集。
- 虽然允许默认接口属性，但由于接口不允许实例字段，因此在接口中实现计算属性是比较困难的。
- 虽然不支持“扩展属性”，但可以用所谓的“getter”扩展方法（例如，`GetData()`）来实现计算。这样就不局限于 C# 8.0 或者.NET Core 3.0 之后的版本了。
- 从旧接口派生出新接口，可以在其中自由添加新功能，而且不会引起版本不兼容的问题，也不受框架的限制。
- 如果选择派生出新接口，那么需要修改实现类，让它实现新接口并利用新的功能。
- 默认接口成员只能通过接口类型来调用。即使是实现了该接口的类型，除非转型为接口，否则也不能访问默认接口成员。换言之，除非在基类中提供了一个实现，否则默认接口成员的行为类似于显式实现的接口成员。
- 接口可以定义 `protected virtual` 成员，但这种成员只能由当前接口和派生接口使用，不能在实现了接口的类中使用。
- 实现类可以重写默认接口成员，从而允许每个类根据需要定义行为。使用扩展方法时，具体怎么绑定是基于编译时可以访问的扩展方法来解析的。结果就是，具体使用哪个实现，是在编译时而不是在运行时确定的。因此，使用扩展方法时，实现类的作者无法为从库中调用的方法提供不同的实现。例如，`System.Linq.Enumerable.Count()`的作用是获取集合中的元素数量，它通过转型为基于索引的 `List` 来提供了一个特殊的实现，能直接计数，效率很高。结果就是，为了利用这种效率的提升，唯一的办法就是实现一个基于 `List` 的接口。相比之下，使用默认接口实现，任何实现类都可以重写此方法以提供更好的版本。

总之，只有通过第二个接口或者默认接口成员，才能添加属性的多态行为。当更新的接口仅包含方法而不包含属性时，优先选择扩展方法。

设计规范

CONSIDER using extension methods or an additional interface in place of default interface members when adding methods to a published interface.

如果需要为已发布的接口添加新方法，**考虑**使用扩展方法或派生出一个新接口，而不是使用默认接口成员。

DO use extension methods when the interface providing the polymorphic behavior is not under your control.

如果提供多态行为的接口不受你的控制，那么**要**使用扩展方法。

8.10 比较接口和抽象类

接口引入了另一类的数据类型（是少数不扩展 System.Object 的类型之一[①]）。但和类不同，接口永远不能实例化。只能通过对实现了接口的一个对象的引用来访问接口实例。不能用 new 操作符创建接口实例。所以，接口不能包含任何构造函数或终结器。在 C# 8.0 之前，，接口不允许静态成员。

接口近似于抽象类，有一些共同点，比如都缺少实例化能力。表 8.2 对它们进行了比较。鉴于抽象类和接口各有优势和劣势，所以必须根据表 8.2 进行的比较和后续的设计规范，做出最符合成本效益的决策。

表 8.1 比较抽象类和接口

抽象类	**接口**
不能直接实例化，只能实例化一个非抽象的派生类	不能直接实例化，只能实例化一个实现类型
派生类要么自己也是抽象的，要么必须实现所有抽象成员	实现类型必须实例化所有接口成员
可以添加额外的非抽象成员，由所有派生类继承，不会破坏跨版本兼容性	从 C# 8.0/.NET Core 3.0 开始，可以添加额外的默认接口成员，由所有派生类继承，而不会中断跨版本兼容性
可以声明方法、属性和字段（以及其他所有成员类型，包括构造函数和终结器）	实例成员限于方法和属性，不能声明字段、构造函数或终结器。静态成员则没有限制，可以声明静态构造函数、静态事件和静态字段

[①] 不扩展 System.Object 的还有指针类型和“类型参数”类型。不过，每个接口类型都可以转换为 System.Object，并允许在接口的任何实例上调用 System.Object 的方法。所以，这个区别或许有点儿吹毛求疵。

成员可以为实例或静态，甚至可以为抽象，非抽象成员可以提供默认实现供派生类使用	从 C# 8.0/.NET Core 3.0 开始，成员可以为实例、抽象或静态，甚至可以为非抽象成员提供实现供派生类使用
可以声明虚成员。不应重写的成员不要声明为 `virtual`（参见代码清单 8.13）	所有（非密封）成员都是虚成员，不管是否显式声明为 `virtual`。因此，接口没有办法阻止重写行为。
派生类只能从一个基类派生（单继承）	实现类型可实现任意多的接口
不支持静态抽象成员	C# 11 引入了静态抽象成员

设计规范

CONSIDER defining an interface if you need to support its functionality on types that already inherit from some other type.

要使已从其他类型派生的类型支持某些功能，**考虑**定义接口。[1]

总的来说，如果能在.NET Core 3.0 或更高版本的框架上开发，那么 C# 8.0（或更高版本）定义的接口具有抽象类的几乎所有能力。虽然不能在接口中声明实例字段，但由于实现类型可以重写接口定义的属性以提供存储，而接口可以利用这种存储，所以接口实际上提供了抽象类所提供的所有功能的一个超集。此外，`protected` 接口成员提供了更好的封装性，并能模拟多继承。但要注意的是，默认接口成员的目的是进行版本控制而不是实现多态性，因此在需要多态性的时候，使用这种成员需谨慎。

C# 11 增加了对静态抽象（`static abstract`）成员的支持。由于这些功能在没有泛型的情况下无法完全发挥作用，因此我们将在第 12 章重拾这一话题。

[1] 译注：例如，你可能已经有一个 `Animal` 基类和几个派生类如 `Dog` 和 `Cat`。如果想为某些动物类（例如 `Bird`）添加特定的行为（例如，`IFlyable` 表示它们可以飞），那么定义一个接口而不是使所有动物都继承自一个共同的“能飞的动物”基类更为合适，

8.11 比较接口和特性

有的时候，我们使用无任何成员的接口（无论直接定义还是继承的成员都没有）来描述关于类型的信息。例如，可以创建名为 `IObsolete` 的一个标记接口（marker interface）指出某类型已被另一类型取代。[①]一般认为这是对接口机制的“滥用”；接口应表示类型能执行的功能，而非陈述关于类型的事实。所以这时不是使用标记接口，而是改为使用特性（attributes）。详情参见第 18 章。

设计规范

AVOID using “marker” interfaces with no members; use attributes instead.

避免使用无成员的标记接口，改为使用特性。

8.12 小结

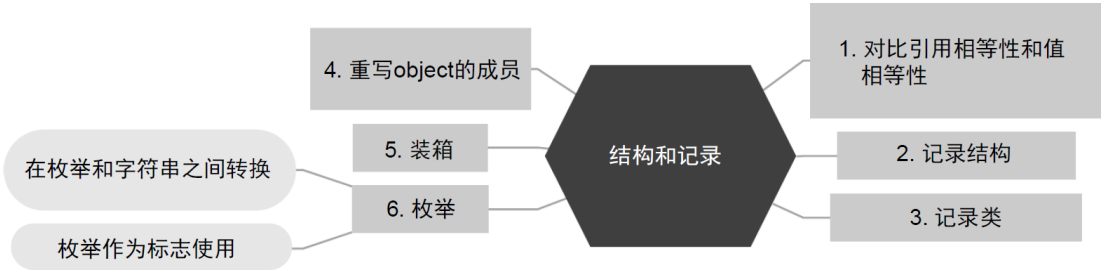
接口是 C#面向对象编程的关键元素，提供了和抽象类相似的多态能力。同时，由于类能同时实现多个接口，所以还不占用单继承的机会。从 C# 8.0/.NET Core 3.0 开始，接口可以通过“默认接口成员”来包含实现，这使其几乎成为抽象类所提供的所有功能的一个超集。当然，这同时也放弃了向后兼容。

在 C#中，接口可以显式或隐式实现，具体取决于实现类是直接公开接口成员，还是通过到接口的强制转换来公开。此外，对显式和隐式的决定是在接口成员的级别上做出的。可以隐式实现接口的一个成员，显式实现接口的另一个成员。

[①] 译注：例如，假定 `OldClass` 已经废弃，不建议继续使用。如果用标记接口的方式，就是让该类从空无一物的标记接口继承。例如，`class OldClass : IObsolete{}`。但是，不建议采用这种方法，而是改为使用特性。例如，在类的声明前加一个`[Obsolete("OldClass 已启用，改为使用 NewClass")]`特性。

第 9 章 结构和记录

到目前为止，本书已经使用了大量值类型，例如 `int`。本章不仅要讨论值类型的使用，还要讨论如何自定义值类型。值类型的一个关键概念是能够比较实例是否具有相同的值，这一概念也适用于引用类型。然而，现在不必从头编码这一功能。C# 9.0 和 C# 10.0 分别引入了记录类（record class）和记录结构（record struct），从而提供了所谓的**记录构造**（record construct），可以极大地简化编码。本章将探讨结构、记录和一种称为“枚举”的特殊值类型。



自定义结构要想正确实现，存在一些明显的复杂性，而且相对来说也很少需要。结构在C#开发中自然扮演着一个重要的角色，但与自定义类相比，开发者平时声明的自定义结构是相当少的。在需要与非托管代码互操作的代码中，才有可能需要频繁使用自定义结构。此外，除非是单个值，占用内存为 16 字节或更少，不可变，而且很少装箱，否则不应定义结构。本章将更详细地讨论这些概念。

设计规范

DO NOT define a struct unless it logically represents a single value, consumes 16 bytes or less of storage, is immutable, and is infrequently boxed.

除非结构在逻辑上代表单个值，占用内存为 16 字节或更少，不可变，而且很少发生装箱，否则**不要**定义结构。

> **初学者主题：类型的分类**
>
> 迄今为止讨论的所有类型都属于两个类别之一：引用类型和值类型。两者区别在于拷贝策略。不同策略造成每种类型在内存中以不同方式存储。为巩固之前所学，这里重新总结一下值类型和引用类型，以便温故而知新。

值类型

值类型的变量直接包含数据，如图 9.1 所示。换言之，变量名直接与值的存储位置关联。因此，将原始变量的值赋给另一个变量，会在新变量的位置创建原始变量值的内存拷贝。两个值类型的变量不可能引用同一个内存位置（除非其中一个或两个是 `out` 或 `ref` 参数；根据定义，这种参数是另一个变量的别名），所以更改一个变量的值不会影响另一个变量。

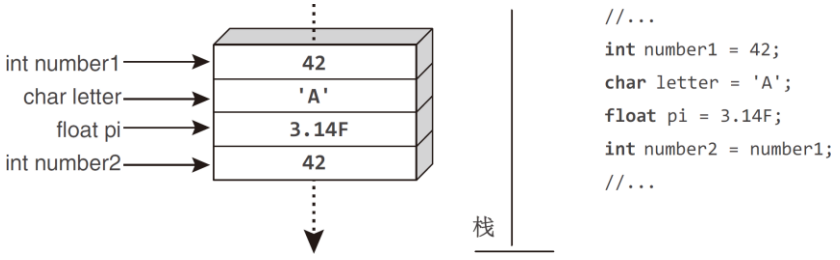


图 9.1 值类型的变量直接包含数据

值类型的变量就像上面写了数字的纸。要更改数字，可以擦除并写上不同的数字。也可以将数字从这张纸拷贝到另一张纸。但两张纸就独立了。在一张纸上面擦除和替换不会影响另一张纸。

类似地，将值类型的实例传给 `Console.WriteLine()`这样的方法也会创建一个内存拷贝，具体就是从实参的存储位置拷贝到形参的内存位置。在方法内部对形参变量进行任何修改都不会影响调用者中的原始值。由于值类型要求创建内存拷贝，所以定义时不要让它们消耗太多内存（一般应为 16 字节或更少）。

设计规范

AVOID creating value types that consume more than 16 bytes of memory.

避免创建占用超过 16 字节内存的值类型。

值类型的值一般只是短时间存在；通常作为表达式的一部分，或者用于激活方法。在这些情况下，值类型的变量和临时值经常存储在称为**栈**（stack）的临时存储池中。（用词不太恰当，临时池并非一定要从栈中分配存储。事实上，它经常选择从可用的寄存器中分配存储。不过这属于实现细节。）

临时池清理起来的代价低于需要垃圾回收的堆。不过，值类型要比引用类型更频繁地拷贝，会对性能造成一定影响。总之，不要觉得“值类型更快是因为能在栈上分

配”。

引用类型

相反，引用类型变量的值是对一个对象实例的引用（参见图 9.2）。引用类型的变量存储的是引用（通常作为内存地址实现），要去那个位置才能找到对象实例的数据。因此，为了访问数据，“运行时”要从变量读取引用，进行“解引用”[①]才能到达实际包含了实例数据的内存位置。

所以，引用类型的变量关联了两个存储位置：直接与变量关联的存储位置，以及由变量中存储的值引用的存储位置。

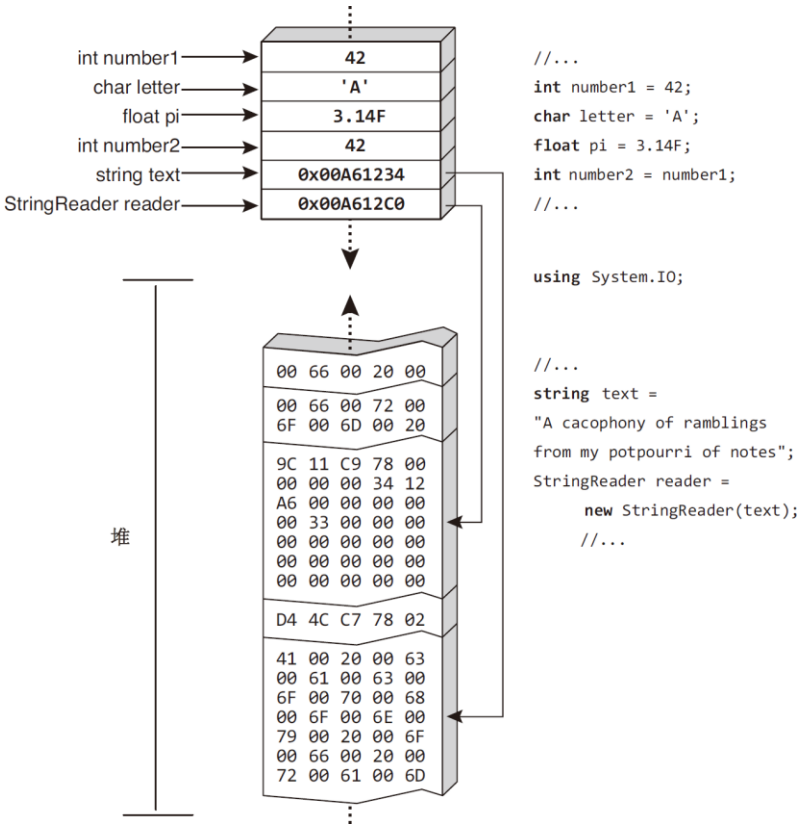


图 9.2 引用类型指向堆

[①] 译注：引用（reference）是地址；解引用（dereference）从地址获取资源。后者还可以说成“提领”和“用引”。

引用类型的变量也像是一张上面总是写了东西的纸。例如，假定一张纸上写了家庭地址“123 Sesame Street, New York City”。纸是变量，地址是对一幢建筑物的引用。纸和上面写的地址都不是建筑物本身，而且纸在哪里跟建筑物在哪里没有任何关系。在另一张纸上拷贝该引用，两张纸的内容都引用同一幢建筑物。以后将建筑物漆成绿色，可以观察到两张纸引用的建筑物都变成了绿色，因为它们引用的是同样的东西。

处理直接与变量（或临时值）关联的存储位置时，方式和处理与值类型变量关联的存储位置没有区别。如果已知变量仅短时间存在，那么就在临时存储池中分配。对于引用类型的变量，它的值要么是 `null`，要么是对需要进行垃圾回收的堆上的一个存储位置的引用。

值类型的变量直接存储实例的数据；相反，要进行一次额外的“跳转”才能访问到与引用关联的数据。首先要对引用进行“解引用”来找到实际数据的存储位置，然后才能读取或写入数据。对引用类型的值进行拷贝的时候，拷贝的只是引用，这个引用非常小。（引用的大小保证不超过处理器的“bit size”；32 位机器是 4 字节的引用， 64 位机器是 8 字节的引用，以此类推。）相反，拷贝值类型的值会拷贝所有数据，这些数据可能很大。所以，有时拷贝引用类型的效率更高。这正是设计规范要求值类型不得大于 16 字节的原因。如果拷贝值类型的代价比拷贝引用高出四倍，就应该考虑把它设计成引用类型了。

由于引用类型只拷贝对数据的引用，所以两个变量可以引用同一份数据。此外，通过一个变量对数据做出的更改可以通过另一个变量观察到。赋值和方法调用均会如此。

还是前面的例子，如果将建筑物的地址传给一个方法，那么会生成包含地址（引用）的那张纸的拷贝，并将该拷贝传给方法。方法无法更改原始纸张的内容来引用不同的建筑物。但是，如果该方法将引用的建筑物漆成别的颜色，那么当方法返回时，调用者会观察到这一变化。

9.1 对比引用相等性和值相等性

但是，在我们考虑实际的例子之前，先来考虑一下为什么需要实现**基于值的相等性**（或简称为**值相等性**）。如果两个引用都指向同一个对象实例，那么它们是完全相同的（**引用相等**）。另一方面，具有相同值的对象是**值相等**的。为了支持值相等，类型需要实现基于值的相等性。两个引用相等的对象显然也是值相等的——现在比较的是对同一个对象的两个引用。然而，如果两个对象之间的（相关）数据也相等，那么它们可以具有相等的值。只有引用类型才可以引用相等，它支持的是**同一性**（identity）的概念——两个引用指向同一个实例。

相比之下，值类型永远不可能引用相等。如果比较 42 是否与 42 引用相等，那么答案总是为 false。每个值类型都存储在内存中的不同位置（引用），甚至将值类型传递给方法进行比较这一简单的行为，也会创建值类型的额外拷贝。

> **注意：**在值类型上调用 ReferenceEquals()总是返回 false。

虽然值类型永远不可能引用相等，但引用类型可以为值相等的判断提供支持。object 包含一个名为 ReferenceEquals()的静态方法，它判断两个实参是否完全相同——引用相等。另外，object 还定义了一个名为 Equals()的虚方法，它默认依赖 ReferenceEquals()来判断引用类型的相等性。但是，完全可以在引用类型和值类型上自定义 Equals()方法，使其实现值相等性判断，而不是只能判断引用相等。

事实上，应该总是在值类型上重写 Equals()方法，因为否则只能使用默认实现，该实现仅比较第一个字段来判断值相等性。但是，这通常还不够。假定一个值类型有三个属性，那么仅比较第一个属性很可能不足以检查值相等性。例如，给定一个角度对象 Angle，仅比较第一个度属性（Degrees）是不够的。两个角度只有在度和分秒数（以及 Angle 包含的其他任何属性）都相等的前提下才相等[①]。因此，几乎总是需要为值类型提供一个自定义的值相等实现。

然而，值相等性不仅限于值类型。值相等性也适用于引用类型。string 就是一个很好的例子。尽管 string 是引用类型，但假如两个字符串中的所有字母都一样，那么即使两个字符串是不同的实例，它们的比较结果也应该是相等。

除此之外，为了正确重写 Equals()，要求存在并可能被重写的一系列额外成员，具体将在后面的“实现值相等性”一节描述。这些成员包括 GetHashCode()以及==和!=操作符。然而，在此之前，让我们先深入了解如何定义一个值类型。

> **注意：**object.Equals()的实现，即在重写之前所有对象的默认实现，仅依赖于 ReferenceEquals()。所有值类型都要重写默认实现，以比较值类型的一个或多个值。引用类型也可以选择重写 Equals()来支持值相等性判断。自定义值类型的时候，始终都要实现相等性判断。

[①] 译注：例如，一个角度可能是 30 度 15 分 45 秒。

9.2 结构

除了 string 和 object 是引用类型，其他所有 C#内建类型（比如 bool 和 decimal）都是值类型。框架还提供了其他大量值类型。开发人员甚至能定义自己的值类型。

为自定义值类型，要使用和定义类/接口相似的语法。区别在于值类型使用关键字 struct，如代码清单 9.1 所示。遗憾的是，方法主体中注释掉的省略号代表相当多的复杂性，本章稍后会详述。

代码清单 9.1　示例结构

```
public struct Angle
{
    // ...
}
```

9.2.1 记录结构

幸好，从 C# 9.0 开始，创建结构时所涉及的许多复杂性都通过一个新的上下文关键字 record 被消除了。该关键字触发编译器生成与值类型行为相关的大部分重要且复杂的代码。代码清单 9.2 展示了一个例子。通过这个简单的语法，我们有了一个描述高精度角度的值类型，其中包括其度、分和秒（一分是一度的六十分之一，一秒是一分的六十分之一）。这个系统主要用于导航，因为它有一个好处，即在赤道上海平面上的一分弧线正好是一海里。[①]

代码清单 9.2　声明记录结构

```
// 使用 record struct 构造来声明一个值类型
public readonly record struct Angle(
    int Degrees, int Minutes, int Seconds, string? Name = null);
```

这种语法仅用一个主构造函数便完成了整个类型的定义。但和第 6 章描述的常规主构造函

[①] 译注：具体来说，地球被想象成一个完美球体，赤道是地球的最大圆周。赤道被分成 360 度，每度又被细分为 60 分。在这种度量体系中，赤道上的“一分弧线”——即赤道周长的 1/360 的 1/60——被定义为一海里。

数不同，记录的主构造函数还会依据（可选的[1]）位置参数生成一组初始属性（直到 C# 12.0 才可用）。因此，这个简单的代码结构，即所谓的**记录结构**（record struct），满足了构建值类型所需的全部必要条件，包括值类型声明、通过属性来实现的数据存储、不变性、构造函数初始化、解构、相等性行为，甚至还提供了 ToString()诊断功能。代码清单 9.3 演示了如何使用 Angle 类型。

代码清单 9.3　使用记录结构

```
using System.Diagnostics;

public class Program
{
    public static void Main()
    {
        (int degrees, int minutes, int seconds) = (90, 0, 0);

        // 构造函数根据位置参数来生成
        Angle angle = new(degrees, minutes, seconds);

        // 记录包含一个 ToString()实现，它返回：
        // "Angle { Degrees = 90, Minutes = 0, Seconds = 0, Name =  }"
        Console.WriteLine(angle.ToString());

        // 记录有一个使用了位置参数的解构函数
        if (angle is (int, int, int, string) angleData)
        {
            Trace.Assert(angle.Degrees == angleData.Degrees);
            Trace.Assert(angle.Minutes == angleData.Minutes);
            Trace.Assert(angle.Seconds == angleData.Seconds);
        }

        Angle copy = new(degrees, minutes, seconds);
        // 记录提供了一个自定义的相等性操作符
        Trace.Assert(angle == copy);

        // with 操作符等价于：
        // Angle copy = new(degrees, minutes, seconds);
        copy = angle with { };
        Trace.Assert(angle == copy);
```

[1] 无论位置参数还是围绕它们的圆括号都是可选的（没有位置参数时，圆括号可以省略）。虽然仅用圆括号来声明会很奇怪，但这样做其实类似于声明一个空白结构或类。那么，什么是“位置参数”？在你列出一系列参数的时候，其实就已经定义了它们的“位置”。

```
        // with 操作符支持“对象初始化器”语法，
        // 用于实例化一个修改过的拷贝。
        Angle modifiedCopy = angle with { Degrees = 180 };
        Trace.Assert(angle != modifiedCopy);
    }
}
```

从代码清单 9.2 可以看出，单纯依据位置参数，就自动生成了构造函数和解构函数。另外，还依据这些位置参数生成了属性（未显示）。如果所有属性相等，那么两个 Angle 的实例会被求值为相等，这种相等性判断是遵照相等性操作符（==）的重写规范来实现的。

9.2.2 记录结构的 CIL 代码

记录结构的所有功能都是在编译时由 C#编译器生成的。代码清单 9.4 展示了与编译器生成的 CIL 代码等价的 C#代码。

代码清单 9.4　与记录结构 CIL 代码等价的 C#代码

```
using System.Runtime.CompilerServices;
using System.Text;

[CompilerGenerated]
public readonly struct Angle : IEquatable<Angle>
{
    public int Degrees { get; init; }
    public int Minutes { get; init; }
    public int Seconds { get; init; }
    public string? Name { get; init; }
    public Angle(int Degrees, int Minutes, int Seconds, string? Name)
    {
        this.Degrees = Degrees;
        this.Minutes = Minutes;
        this.Seconds = Seconds;
        this.Name = Name;
    }

    public override string ToString()
    {
        StringBuilder stringBuilder = new();
        stringBuilder.Append("Angle");
        stringBuilder.Append(" { ");
        if (PrintMembers(stringBuilder))
        {
            stringBuilder.Append(' ');
```

```
        }
        stringBuilder.Append('}');
        return stringBuilder.ToString();
    }

    private bool PrintMembers(StringBuilder builder)
    {
        builder.Append("Degrees = ");
        builder.Append(Degrees);
        builder.Append(", Minutes = ");
        builder.Append(Minutes);
        builder.Append(", Seconds = ");
        builder.Append(Seconds);
        builder.Append(", Name = ");
        builder.Append(Name);
        return true;
    }

    public static bool operator !=(Angle left, Angle right) =>
        !(left == right);

    public static bool operator ==(Angle left, Angle right) =>
        left.Equals(right);

    public override int GetHashCode()
    {
        static int GetHashCode(int integer) =>
            EqualityComparer<int>.Default.GetHashCode(integer);

        return GetHashCode(Degrees) * -1521134295
            + GetHashCode(Minutes) * -1521134295
            + GetHashCode(Seconds) * -1521134295
            + EqualityComparer<string>.Default.GetHashCode(Name!);
    }

    public override bool Equals(object? obj) =>
        obj is Angle angle && Equals(angle);

    public bool Equals(Angle other) =>
        EqualityComparer<int>.Default.Equals(Degrees, other.Degrees)
        && EqualityComparer<int>.Default.Equals(Minutes, other.Minutes)
        && EqualityComparer<int>.Default.Equals(Seconds, other.Seconds)
        && EqualityComparer<string>.Default.Equals(Name, other.Name);

    public void Deconstruct(out int Degrees, out int Minutes,
        out int Seconds, out string? Name)
            => (Degrees, Minutes, Seconds, Name) =
                (this.Degrees, this.Minutes, this.Seconds, this.Name);
}
```

如代码清单 9.4 所示，虽然代码清单 9.2 只有区区一行代码，但在编译时会生成大量代码。首先要注意的是，`Angle` 不是一个 `class`，而是一个 `struct`，而且用于实现“语法糖”的 `record` 上下文关键字被拿掉了。在 C#中，自定义的值类型必须是一个 `struct`。而且，虽然 C#允许从头开始写整个结构，但就像代码清单 9.4 演示的那样，C# 9.0 的 `record` 关键字会自动生成大量样板代码，使得再也没有理由不使用 `record` 关键字开始。更棒的是，你以后可以自定义，根据需要重写默认生成的实现。

注意，C# 9.0 只允许创建记录类，编译器只允许写一个 `record` 关键字，后面不能有 `struct`。但从 C# 10.0 开始，由于新增了用 `record struct` 创建记录结构的功能，所以允许显式声明 `record class` 来创建记录类。为了清晰起见，我们建议在创建记录类时总是使用 `record class`，而不要简写为 `record`。

设计规范

DO use record struct when declaring a struct (C# 10.0).

在声明结构时**要**使用 `record struct`（C# 10.0）。

DO use record class (C# 10.0) for clarity, rather than the abbreviated record-only syntax.

为了清晰起见，**要**写 `record class`（C# 10.0），而不要简写为 `record`。

所有值类型都隐式密封（不能从它们派生）。此外，除了枚举之外的所有值类型（本章后面会讨论枚举）都派生自 `System.ValueType`。因此，结构的继承链总是从 `object` 到 `System.ValueType` 再到自定义结构。

`System.ValueType` 通过重写 `object` 的所有虚方法来实现了值类型的行为。这些虚方法有三个，即 `Equals()`，`ToString()`和 `GetHashCode()`。然而，这些默认实现往往还不够，因为它把进一步特化这些方法的责任留给了开发人员——当然，这是截止 C# 10.0 的情况。现在，由于引入了 `record` 修饰符，所以 C#编译器生成的自定义重写更适合生产代码。记录结构提供的许多功能也适用于记录类。

9.3 记录类

`record` 修饰符一开始就是为类类型准备的（C# 9.0），对值类型的支持是 C# 10.0 才引入的。代码清单 9.5 展示了一个例子。

代码清单 9.5　声明记录类

```
//使用 record class 构造来声明一个引用类型
public record class Coordinate(
    Angle Longitude, Angle Latitude)
```

在这个记录类中，我们使用 Angle 类型作为位置参数传入 Coordinate 类，以表示经度和纬度。

为了区分记录结构和记录类及其非记录版本，本章会把它们的非记录版本分别称为**标准结构**和**标准类**。在 C#引入了记录类之后，我们自然会产生一个疑问，何时应该使用记录类而不是标准类，反过来呢？这里的关键在于，也或许是定义记录类唯一的原因，我们希望一个类型能够判断**值相等性**。事实上，每当一个自定义类型需要判断值相等性时（值类型始终需要），就应该尽可能使用记录。当然，如果使用的是 C# 9.0 之前的一个版本（对于引用类型）或者 C# 10.0 之前的一个版本（对于值类型），那么自然是无法享受到记录的好处的。

当基类不是记录时，也不能为一个类使用记录构造。记录类只能从另一个记录类继承，而标准类只能从另一个标准类继承。不能混合标准类和记录来搭建继承结构。

> **注意：**默认情况下，引用类型应该实现为标准类，而且仅在需要判断值相等性时，才实现为记录类。

记录结构的许多特色在记录类中也有，包括简洁的声明、通过属性来提供的数据存储、构造函数初始化、解构、ToString()诊断功能以及最重要的值相等性判断（而非仅仅是引用相等性）——这个功能要自己实现，就可能太麻烦了。代码清单 9.6 展示了与编译器为记录类生成的 CIL 代码等价的 C#代码。

代码清单 9.6　与记录类 CIL 代码等价的 C#代码

```
using System.Runtime.CompilerServices;
using System.Text;

[CompilerGenerated]
public class Coordinate : IEquatable<Coordinate>
{
```

```
public Angle Longitude { get; init; }
public Angle Latitude { get; init; }

public Coordinate(Angle Longitude, Angle Latitude) : base()
{
    this.Longitude = Longitude;
    this.Latitude = Latitude;
}

public override string ToString()
{
    StringBuilder stringBuilder = new ();
    stringBuilder.Append("坐标");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(' ');
    }
    stringBuilder.Append('}');
    return stringBuilder.ToString();
}

protected virtual bool PrintMembers(StringBuilder builder)
{
    RuntimeHelpers.EnsureSufficientExecutionStack();
    builder.Append("经度 = ");
    builder.Append(Longitude.ToString());
    builder.Append(", 纬度 = ");
    builder.Append(Latitude.ToString());
    return true;
}

public static bool operator !=(
    Coordinate? left, Coordinate? right) =>
        !(left == right);

public static bool operator ==(
    Coordinate? left, Coordinate? right) =>
        ReferenceEquals(left, right) ||
            (left?.Equals(right) ?? false);

public override int GetHashCode()
{
    static int GetHashCode(Angle angle) =>
        EqualityComparer<Angle>.Default.GetHashCode(angle);

    return (EqualityComparer<Type>.Default.GetHashCode(
        EqualityContract()) * -1521134295
        + GetHashCode(Longitude)) * -1521134295
```

```
            + GetHashCode(Latitude);
    }

    public override bool Equals(object? obj) =>
        Equals(obj as Coordinate);

    public virtual bool Equals(Coordinate? other) =>
        (object)this == other || (other is not null
            && EqualityContract() == other!.EqualityContract()
            && EqualityComparer<Angle>.Default.Equals(
                Longitude, other!.Longitude)
            && EqualityComparer<Angle>.Default.Equals(
                Latitude, other!.Latitude));

    public void Deconstruct(
        out Angle Longitude, out Angle Latitude)
    {
        Longitude = this.Longitude;
        Latitude = this.Latitude;
    }

    protected virtual Type EqualityContract() => typeof(Coordinate);

    public Type ExternalEqualityContract => EqualityContract();

    // IL 中的实际名称是"<Clone>$"。但是，不能
    // 自行在记录中添加一个名为 Clone 的方法。
    public Coordinate Clone() => new(this);

    protected Coordinate(Coordinate original)
    {
        Longitude = original.Longitude;
        Latitude = original.Latitude;
    }
}
```

为记录结构和记录类生成的代码存在几处预料之中的差异。

首先，一些成员被标记为 virtual，包括 PrintMembers(StringBuilder builder)，Equals(Coordinate? Other)和 EqualityContract()。相反，记录结构中绝对不会出现 virtual 修饰符，因为所有结构（值类型）都隐式密封，在记录结构中使用 virtual 没有意义。

其次，记录类需要考虑 null 值的可能性。因此，==操作符和两个相等性判断方法都需要检查参数值为 null 的情况。

设计规范

DO use records, where possible, if you want equality based on data rather than identity.

如果希望判断数据相等性而不是同一性（identity），那么**要**尽可能使用记录。

记录结构隐式密封，所以不可能继承。与之相反，记录类支持继承。主要限制在于，记录类只能从其他记录类继承，甚至不能从标准类继承。代码清单 9.7 展示了一个例子。

代码清单 9.7　记录类继承

```
public record class NamedCoordinate(
    Angle Longitude, Angle Latitude, string Name) :
        Coordinate(Longitude, Latitude);
```

注意，在定义继承关系时，还可以使用跟在基类名称后面的参数来确定如何调用基类构造函数。在本例中，我们将 `Longitude` 和 `Latitude` 传给由 `Coordinate` 的位置参数生成的构造函数。

在本节剩余的部分，将深入探讨记录背后的编码细节，强调记录类和记录结构之间的差异，并且最重要的是，深入探讨如何实现值相等性判断。我们首先讨论记录的数据存储。

9.3.1 用属性提供数据存储

注意，位置参数 `Degree`，`Minutes` 和 `Seconds` 映射到 `Angle` 结构的自动生成属性。`Coordinate` 类的 `Longitude` 和 `Latitude` 参数与此相似。一旦初始化完成，属性就是只读的（用 init-only setter 实现，参见 6.7.5 节）。“只读”这一事实是记录类的默认行为。声明记录结构的时候，也可以用 `readonly` 修饰符来指定这一行为——该修饰符导致值类型在初始化完成后变得**不可变**（immutable）。

记录有一个容易被忽视的特点，即根据约定，位置参数通常采用 PascalCase 大小写风格。因此，构造函数的参数也是如此。虽然编译器不要求这样做，但使用 PascalCase 大小写风格来命名位置参数，可以确保相应的属性也使用 PascalCase 大小写风格——后者在类型 API 对外公开时，就非常引人注目了。

为了自定义默认实现，可以使用相同的名称定义每个属性——也许使用一个支持字段，再加上自定义验证功能。甚至可以使用同名字段来替换属性。此外，可以编码额外的属性和

字段，即使它们没有作为位置参数包含在内。唯一的限制是，由于结构的 `readonly` 修饰符，所有属性和字段也必须只读。

注意，自动生成的属性没有关联的验证机制。因此，非可空引用类型的位置参数不会包括一个 `null` 检查。出于这个原因，请考虑始终将引用类型的位置参数定义为可空，除非你提供了进行 `null` 检查的自定义实现。

设计规范

DO use PascalCase for the positional parameters of the record (C# 9.0).

要为记录的位置参数使用 PascalCase 大小写风格（C# 9.0）。

DO define all reference type positional parameters as nullable if not providing a custom property implementation that checks for `null`.

如果没有提供执行了 `null` 检查的自定义属性实现，那么**要**将所有引用类型的位置参数定义为可空。

DO implement custom non-nullable properties for all non-nullable positional parameters.

要为所有非可空位置参数实现自定义非可空属性。

9.3.2 不可变值类型

所有值类型和记录类的一个重要准则是不可变（immutable）。换言之，一旦实例化了其中任何一个，实例就不能修改。该准则是出于三方面很好的理由。

首先，对于结构来说，值类型应该代表值。两个整数相加，其中任何一个都不应改变。换言之，两个加数应该不可变，而是生成第三个值作为结果。其次，由于值类型拷贝的是值而非引用，所以很容易混淆并错误地以为对一个值类型变量的更改会造成另一个值类型的更改，就像引用类型那样。

第三，无论结构还是记录类，哈希码是根据类型中存储的数据计算出来的，哈希码永远不应发生变化。（参见本章后面的“高级主题：重写 `GetHashCode()`”）。如果这些类型中的数据改变了，并且计算出新的哈希码，那么在旧哈希码存在的集合中搜索新的哈希码，可能导致值永远找不到，从而导致不正确的行为。

注意，代码清单 9.2 的 `Angle` 结构是不可变的，因为所有属性都是使用 `init`-only setter 而

不是 set 来声明的自动实现只读属性[①]。为了验证自己确实遵循了“不可变”准则，可以在结构定义中（无论记录结构还是标准结构）使用 readonly 修饰符[②]，如代码清单 9.8 所示。

代码清单 9.8　在结构上使用 readonly 修饰符

```
readonly struct Angle { }
```

和记录一样，编译器现在会验证整个结构不可变。但凡发现一个非只读的字段或者有赋值方法（setter）的属性，编译器就会报错。但是，不可以在记录类或标准类上使用 readonly 修饰符。

在比较罕见的情况下，你可能需要更细粒度的控制，而不是将整个结构声明为只读。C# 8.0 允许将单独的结构（不是类）成员声明为只读（其中包括方法，甚至包括 getter——它们可能会修改对象的状态，尽管不应该这样）。例如，Move()方法可以包含一个 readonly 修饰符，如代码清单 9.9 所示：

代码清单 9.9　将 struct 成员定义为只读

```
public readonly Angle Move(int degrees, int minutes, int seconds)
{
    return new Angle(
        Degrees + degrees,
        Minutes + minutes,
        Seconds + seconds);
}
```

如本例所示，如果希望在成员中修改结构，那么应该像 Move()方法那样，新建一个 Angle 实例并返回它。

任何 readonly 成员只要修改了结构数据（无论属性还是字段），或者调用了一个非只读的成员，那么都会报告编译时错误。通过支持对成员的只读访问，开发人员表明了成员是否

[①] 自动实现的只读属性（即用 init 关键字替代了 set）是自 C# 6.0 引入的。

[②] 自 C# 7.2 开始。

可以修改对象实例的行为意图。注意，非自动实现的属性可以在 getter 或 setter 上使用 `readonly` 修饰符（尽管后者可能会很奇怪）。如果想两者同时修饰，那么 `readonly` 修饰符应该放在属性本身上，而不是分别放在 getter 和 setter 上。

虽然允许，但在结构本身只读的时候，在结构成员上使用 `readonly` 修饰符是多余的。不过，最好还是在结构中使用只读（`{get;}`）或者仅初始化 setter（`{get; init;}`）自动实现属性，而不是使用字段。

设计规范

DO use the `readonly` modifier on a struct definition, making value types immutable

要在结构定义上使用 `readonly` 修饰符，使值类型不可变。

DO use read-only or init-only setter automatically implemented properties rather than fields within structs.

在结构中**要**使用只读或仅初始化 setter 自动实现属性，而不是使用字段。

值得注意的是，元组（`System.ValueTuple`）是打破不可变准则的一个例子。要了解为什么它是一个例外，请访问 *http://tinyurl.com/3mcuhepd*。

9.3.3 使用 `with` 操作符克隆记录

由于大多数记录（尤其是记录结构）都是不可变的，所以不是修改它们，而是创建一个新实例来包含修改后的数据。为此，最简单的方法是使用 C# 9.0 引入的 `with` 操作符。如代码清单 9.3 末尾所示，`with` 操作符克隆现有实例来创建一个新副本。然而，并非只能创建一个精确的副本。相反，可以使用对象初始化器形式的语法，在新实例中包含修改后数据（参见代码清单 9.10）。

代码清单 9.10　用 `with` 操作符克隆记录

```
Angle angle = new(90, 0, 0, null);

// with 操作符等价于：
// Angle copy = new(degrees, minutes, seconds);
Angle copy = angle with { };
Trace.Assert(angle == copy);
```

```
// with 操作符支持用对象初始化器形式的
// 语法来实例化一个修改后的拷贝。
Angle modifiedCopy = angle with { Degrees = 180 };
Trace.Assert(angle != modifiedCopy);
```

with 操作符的源（左侧）操作数是源实例。虽然可以创建一个精确的克隆，但使用对象初始化器语法，可以在实例化期间修改任何可访问的成员，并将不同的值赋给它。

记录结构的克隆是一个内存复制过程，类似于以传值方式向方法传递参数。因此，不能更改被克隆的记录结构的实现。

对于记录类，过程则略有不同。C#编译器生成了一个隐藏的 Clone()方法（如代码清单 9.6 所示），该方法进而调用拷贝构造函数[①]，并将源实例作为参数传递。代码清单 9.11 重现了这部分代码。

代码清单 9.11　编译器生成 Clone()方法来进行对象类的克隆

```
// IL 中的实际名称是"<Clone>$"。但是，不能
// 自行在记录中添加一个名为 Clone 的方法。
public Coordinate Clone() => new(this);

protected Coordinate(Coordinate original)
{
    Longitude = original.Longitude;
    Latitude = original.Latitude;
}
```

注意，在使用对象初始化器语法的情况下，被赋值的属性不能是只读的（需要一个 setter 或者 init-only setter）。此外，编译器生成的 Clone()方法使用了一个特殊的、在 C#语言中非法的名称，使其只能通过 with 操作符访问。然而，如果想自定义克隆行为，可以提供自己的拷贝构造函数实现。

9.3.4 记录构造函数

注意，代码清单 9.2 的记录声明与代码清单 9.4 中由编译器生成的构造函数签名看起来几乎完全一样。类似地，代码清单 9.5 中的记录类声明也差不多等价于代码清单 9.6 中由编

[①] 只获取一个参数的构造函数，该参数必须具有包容类型。参见第 6 章。

译器生成的构造函数。总之，记录声明及其位置参数为 C#编译器生成的、具有等价签名的构造函数提供了一个基本结构。

和属性一样，记录的一个特别之处是位置参数根据约定采用 PascalCase 大小写风格，所以构造函数的参数也是如此。（因此，在初始化属性时，生成的构造函数代码使用 `this` 限定符来区分参数和属性本身。）

可以在记录的定义中添加新的构造函数。例如，可以提供一个获取字符串而不是整数作为参数的构造函数，如代码清单 9.12 所示。

代码清单 9.12　新增记录构造函数

```
public Angle(
    string degrees, string minutes, string seconds)
    : this(int.Parse(degrees),
           int.Parse(minutes),
           int.Parse(seconds))
{ }
```

实现附加的构造函数时，采用的是与其他任何构造函数相同的语法。唯一额外的限制是它必须调用 `this` 构造函数初始化器。换言之，必须调用记录生成的构造函数，或者借助另一个构造函数来调用记录生成的构造函数，以确保完成由位置参数生成的那些属性的初始化。

9.3.5 记录结构初始化

在 C# 10.0 之前，是不允许为值类型的“结构”自定义一个默认构造函数的。无论如何，只要不是通过 `new` 操作符来调用构造函数，从而显式实例化结构，那么结构内包含的所有数据都会隐式初始化为相应数据类型的默认值。具体地说，引用类型数据的字段初始化为 `null`，数字类型字段初始化为 `0`，布尔类型字段初始化为 `false`，其中包括为自动实现属性提供支持的字段。类似地，由于编译器在构造函数中注入了**声明时的字段和属性初始化**[①]（例如 `string Description { get; init; } = ""`），所以如果不是调用 `new` 操作符，那么它（声明时的初始化）是不会执行的。

意识到这一行为很重要，因为和引用类型不同，值类型通常在没有 `new` 操作符的情况下实例化。例如，如果将值类型赋值为 `default`，或者不初始化类上的一个值类型字段，那么

[①] 从 C# 10.0 开始支持。

不会执行任何构造函数，未初始化的数组项（如代码清单 9.13 所示）也是如此。

代码清单 9.13　未初始化的数组项

```
Angle[] angles = new[42];
```

在这两种情况下，无论默认构造函数将数据成员初始化为什么，这些成员最终获得的都是其数据类型的默认值。有鉴于此，不要依赖默认构造函数或声明时的成员初始化（再次强调，这只是针对值类型在没有用 new 的前提下的实例化）。

在 C# 11.0 之前，结构中的每个构造函数都必须初始化结构的所有字段（以及只读的自动实现属性[①]）。在 C# 10.0 和更早的版本中，如果未初始化结构内的所有数据，那么会导致以下编译时错误。

> CS0171：在控制返回给调用者之前，字段必须被完全赋值。考虑更新到语言版本'11.0'以自动设置字段的默认值。

考虑到结构的字段初始化要求，简洁的只读自动实现属性语法，以及“避免从包容属性外部访问字段”这一设计规范，我们应该优先在结构中使用只读自动实现属性而不是字段。

设计规范

DO ensure that the default value of a struct is valid; encapsulation cannot prevent obtaining the default “all zero” value of a struct

要确保结构的默认值是有效的；封装不能防止获取结构的默认“全零”值。

DO NOT rely on either default constructors or member initialization at declaration to run on a value type.

对于值类型，**不要**依赖默认构造函数或声明时的成员初始化会得到调用。

[①] 从 C# 6.0 开始，通过只读的自动实现属性进行初始化就足够了，因为支持字段是未知的，除此之外没有其他手段可以初始化它。

> **高级主题：将 new 用于值类型**
>
> 为引用类型使用 new 操作符，“运行时”会在托管堆上创建对象的新实例，将所有字段初始化为默认值，再调用构造函数，将对实例的引用以 this 的形式传递。new 操作符最后返回对实例的引用，该引用被拷贝到和变量关联的内存位置。与之相反，为值类型使用 new 操作符，“运行时”会在临时存储池中创建对象的新实例，将所有字段初始化为默认值，调用构造函数，将临时存储位置作为 ref 变量以 this 的形式传递。结果是值被存储到临时存储位置，然后可将该值拷贝到和变量关联的内存位置。
>
> 和类不同的是，结构不支持终结器。结构以传值的形式拷贝，不像引用类型那样具有“引用同一性”（referential identity）。所以，难以知道什么时候能安全执行终结器并释放结构所占用的非托管资源。垃圾回收器知道在什么时候没有了对引用类型实例的“活动”引用，可在此之后的任何时间运行终结器。但是，“运行时”没有任何机制能跟踪值类型在特定时刻有多少个拷贝。

> **语言对比：C++——struct 定义了带公共成员的类型**
>
> 在 C++中，对于用 struct 和 class 声明的类型，区别在于默认的可访问性是公共还是私有。两者在 C#中的区别则大得多，在于类型的实例是以值还是引用的形式拷贝的。

9.3.6 记录的解构函数

除了属性和构造函数，C#编译器还会自动生成一个和位置参数对应的解析函数（deconstructor），如代码清单 9.14 所示。

代码清单 9.14　记录的解构函数

```
public void Deconstruct(
    out int Degrees, out int Minutes, out int Seconds)
{
    Degrees = this.Degrees;
    Minutes = this.Minutes;
    Seconds = this.Seconds;
}
```

这样就可以实现如代码清单 9.3 所示的模式匹配（代码清单 9.15 重现了这段代码）。

代码清单 9.15　模式匹配

```
if (angle is (int, int, int, string) angleData)
{
    // ...
}
```

9.4 重写 object 的成员

第 6 章说过，object 是终极基类，所有类和结构最终都从它派生。还讨论了 object 提供的方法，并提到其中一些是虚方法。本节讨论对虚方法进行重写的细节。注意，重写这个过程对于记录来说是自动完成的。但是，通过研究它生成的代码，在自定义它生成的代码，或者重写一个非记录的类型时，我们可以理解有哪些工作是必须完成的。

9.4.1 重写 ToString()

如果在对象上调用 ToString()，那么会默认返回类的完全限定名称。例如，在一个 System.IO.FileStream 对象上调用 ToString()，会返回字符串"System.IO.FileStream"。但是，对于某些类来说，ToString()应该返回更有意义的结果。以 string 类为例，ToString()应该返回字符串值本身。类似地，在一个 Angle 上调用 ToString()时，会返回：

```
Angle { Degrees = 90, Minutes = 0, Seconds = 0, Name = }
```

Angle 的声明请参见代码清单 9.2，调用结果请参见代码清单 9.3。

调用 System.Console.WriteLine()和 System.Diagnostics.Trace.Write()等方法时，会调用一个对象的 ToString()方法[①]。因此，应该重写该方法，输出比默认实现更有意义的信息。

为了重写 ToString()，只需用 override 修饰 ToString()方法，并返回一个 string。代码清单 9.16 展示了一个例子。

代码清单 9.16　重写 ToString()方法

```
public override string ToString()
```

[①] 除非定义了一个隐式转型操作符，如稍后的“高级主题：转型操作符（()）”所述。

```
    {
        string prefix =
            string.IsNullOrWhiteSpace(Name) ? string.Empty : Name + ": ";
        return $"{prefix}{Degrees}°  {Minutes}' {Seconds}\"";
    }
```

无论如何，你重写的版本不要返回空串或 null，因为缺乏输出会令人非常困惑。开发人员在 IDE 中调试或者向日志文件写入时，ToString()非常有用，因为它能提供更详尽的信息。实现本地化重载方法和其他高级格式化功能时要小心，要确保提供适合一般终端用户的文本显示（换言之，要友好）。同时，显示的字符串不要过长（不要超出屏幕宽度，否则会被截断）——尤其是在 IDE 中调试时。

如果能输出更能说明问题的诊断信息（特别是当目标用户是开发人员时），请考虑重写 ToString()方法。object.ToString()默认输出类型名称，这对用户来说并不友好。另外，从 C# 10.0 开始，可以密封 record 的 ToString()实现，防止由派生类型重写——包括编译器生成的实现。

设计规范

DO override ToString() whenever useful developer-oriented diagnostic strings can be returned.

如果需要返回有用的、给开发人员看的诊断字符串，那么**要**重写 ToString()。

CONSIDER trying to keep the string returned from ToString() short.

考虑使 ToString()返回的字符串简短。

DO NOT return an empty string or null from ToString().

不要从 ToString()返回空串（""）或 null。

DO NOT throwing exceptions or making observable side effects (changing the object state) from ToString().

不要从 ToString()抛出异常或造成可观察到的副作用（改变对象状态）。

DO provide an overloaded ToString(string format) or implement IFormattable if the return value requires formatting or is culture-sensitive (e.g., DateTime).

如果返回值要求格式化或者对语言文化敏感（例如，DateTime），那么**要**重载 ToString(string format)或实现 IFormattable。

CONSIDER returning a unique string from ToString() so as to identify the object instance.

考虑从 ToString()返回独一无二的字符串以标识对象实例。

9.4.2 实现值相等性

开发者经常想当然地认为实现值相等性非常简单。毕竟，比较两个对象有多难呢？然而，其中包含的陷阱不少，许多地方都需要仔细思考和测试。幸好，现在有两个办法可以显著简化代码。首先，如果使用记录（无论记录结构还是记录类），那么 C#编译器会自动生成 Equals()方法及其所有相关成员，支持对记录中所有自动实现的属性和字段的比较（无论它们是不是作为位置参数包含进来的）。其次，可以将标识数据元素[①]（identifying data elements，即自动实现的属性和字段）组合成一个元组，然后比较这些元组。在引用类型的情况下，还需要注意任何潜在的 null 值。

代码清单 9.3 展示了为记录生成的 Equals()实现，它实现了值相等性。C#记录的一大优势在于，它们能自动生成值相等性实现，免去你手动实现之苦，因为要实现的方法可能出乎预料地复杂，其中涉及继承、null、不同数据类型、值类型/引用类型的区别以及对实例中包含的值本身进行比较。话虽如此，如果记录自动生成的实现不符合你的需求，那么也可以自己编写方法。一种可能的情况是，你可能不希望将某些属性值纳入相等性判断。例如，在对 Angle 值进行相等性判断时，你可能不希望考虑 Angle 的 Name 属性，而是仅考虑 Degrees，Minutes 和 Seconds 等属性。代码清单 9.24 会展示一个完整的自定义记录结构。但在此之前，先来看看代码清单 9.17 的节选。

[①] 译注：对于 identifying data，个人更喜欢“关键数据”这一说法。

代码清单 9.17　自定义相等性实现

```
public bool Equals(Angle other) =>
    (Degrees, Minutes, Seconds).Equals(
        (other.Degrees, other.Minutes, other.Seconds));
```

注意，值标识属性被组合成元组，并与其他对象的相同值进行比较。另外要注意的是，这里提供的是 Equals 的一个重载。本例的实现传递了一个 Angle 参数。而 C#编译器自动生成的实现传递的是一个 obj 参数，已经实现了其他所有值相等性功能。但是，只有记录才会这样。其他类型的情况并非如此——你必须自己提供所有这些自定义。另外，每次更新 Equals()方法，还应考虑同时更新 GetHashCode()，以确保使用相同的数据元素。如果为 record 类型重写 Equals()而不重写 GetHashCode()，那么会收到以下警告（反之亦然）。

```
CS8851 '<类名>'定义'Equals'，而不定义'GetHashCode'
```

虽然本例看起来与代码清单 9.4 为记录自动生成的 Equals()版本不同，但在内部，元组（System.ValueTuple<...>）也使用 EqualityComparer<T>。而 EqualityComparer<T>依赖于 IEquatable<T>的类型参数实现（其中只包含一个 Equals<T>(T other)成员）。因此，为了正确重写 Equals，你需要实现 IEquatable<T>。

一旦重写了 Equals()，就有可能出现不一致的问题。也就是说，两个对象可能对于 Equals()返回 true，但对于==操作符返回 false，这是由于==默认执行的是引用相等性检查。为了纠正这个问题，还有必要重写相等（==）和不相等（!=）操作符。当然，这些代码也会作为记录实现的一部分为你自动生成，如代码清单 9.18 所示。

代码清单 9.18　重写==和!=操作符

```
public static bool operator ==(Coordinate? left, Coordinate? right) =>
    ReferenceEquals(left, right) || (left?.Equals(right) ?? false);
// ...
public static bool operator !=(Angle left, Angle right) =>
    !(left == right);
```

大多数时候，我们可以将这些操作符的实现逻辑委托给 Equals()，反之亦然。然而，注意不要让==操作符调用自身[①]而造成无限递归。在本例中，由于我们最初使用了

[①] 记住，像 operator ==(){}这样的称为“操作符方法”。

ReferenceEquals()，所以可以避免这种递归。如果两个值都是 null，那么操作符方法会返回 true。值类型自然不需要检查 null。有关操作符重载的更多信息，请参见第 10 章。

最后，如果自定义类型需要支持排序，那么请考虑实现 IComparable<T>。如果需要本地化，则请考虑实现 IFormattable 接口。在不使用记录结构的前提下，实现值相等性更完整的实现请参见“高级主题：重写 GetHashCode()”。关于值类型，还有另外两个重要主题需要涉及。第一个是装箱，第二个是枚举，这两个都将在后续小节讨论。

> **高级主题：值相等性原则**
>
> 注意，代码清单 9.24 中 Equals()的签名与代码清单 9.19 展示的签名是不匹配的，后者使用了 object 参数。
>
> **代码清单 9.19　不匹配的签名**
>
> ```
> public override bool Equals(object? obj)
> ```
>
> 在类或结构（非记录类型）上实现 Equals()时，还有其他几个因素需要考虑。所有这些都可以从代码清单 9.4（记录结构）和代码清单 9.6（记录类）中一探究竟（它们都是因为 record 修饰符而自动生成的）。
>
> 为了在自定义类型上实现值相等性，需要采取以下步骤。
>
> 1. 实现泛型 IEquatable<T>接口。[①]
>
> 2. 实现 Equals(object? obj)方法，检查 obj 是否具有与自定义类型相同的类型，并调用强类型的 Equals()辅助方法，这样就可以将操作数视为自定义类型而不是 object（参见代码清单 9.20）。
>
> **代码清单 9.20　在自定义类型上实现值相等性**
>
> ```
> public override bool Equals(object? obj)
> {
>     return Equals(obj as Foo);
> }
> ```

[①] 我们将在第 12 章讨论泛型。

3. 实现强类型的 Equals()辅助方法，如代码清单 9.21 所示，并检查 obj 是否为 null（仅限引用类型）。

代码清单 9.21　实现强类型的 Equals

```
public bool Equals(Foo? other)
{
    if (other is null) return false;
    // 步骤 5
    // 返回(this.Number, this.Name) == (other.Number, other.Name);
    return true;
}
```

4. 对于不从 System.Object 派生的引用类型，要检查 base.Equals()，这有利于重构。[①]

5. 比较每个标识数据成员的相等性，最好是使用元组。

6. 重写 GetHashCode()。

7. 重写==和!=操作符（参见第 10 章）。

其中，步骤 3～步骤 4 发生在 Equals()方法的重载版本中，它专门接收自定义数据类型（例如 Coordinate）。这样一来，两个 Coordinate 的比较将完全避开 Equals(object? obj)及其类型检查。

Equals()方法绝不应抛出异常，因为从常理来讲，任何对象都可以和其他任何对象比较，永远都不应导致异常。

设计规范

DO implement GetHashCode(), Equals(), the == operator, and the != operator together—

[①] 译注：这主要是考虑到继承层次结构的问题。基类可能也重写了 Equals()，所以在派生类中重写该方法时，有必要调用 base.Equals()来包含基类的相等性逻辑。这样就不需要在每个派生类中重复基类的相等性逻辑，有利于重构。

not one of these without the other three.

GetHashCode()、Equals()、==操作符和!=操作符**要**一起实现——不要单独实现其中一个而忽略其他三个。

DO use the same algorithm when implementing Equals(), ==, and !=.

要用相同的算法来实现 Equals()、==和!=。

DO NOT throw exceptions from implementations of GetHashCode(), Equals(), ==, and !=.

不要从 GetHashCode()、Equals()、==和!=的实现中抛出异常。

AVOID including mutable data when overriding the equality-related members on mutable reference types or if the implementation would be significantly slower with such overriding.

在可变（mutable）引用类型上重写与相等性相关的成员时，**避免**包含可变数据，或者如果这样的重写会使实现显著变慢。

DO implement all the equality-related methods when implementing IEquatable.

在实现 IEquatable 时，**要**实现所有与相等性相关的方法。

高级主题：重写 GetHashCode()

如果创建的是记录（record）类型，那么 GetHashCode()会作为值相等性判断的一部分自动实现（参见代码清单 9.4 和代码清单 9.6）。如果创建的不是记录类型，那么只要自己定义了相等性实现，那么基本上还必须要同时实现 GetHashCode()。即使是记录，如果自定义了 Equals()实现，那么也可以考虑重写 GetHashCode()，以使用与新的 Equals()实现相似的值集合。如果只重写 Equals()而不重写 GetHashCode()，那么会收到以下警告（对于非 record 类型）：

```
CS0659: '<类名>'重写 Object.Equals(object o)但不重写 Object.GetHashCode()
```

总之，在使用非记录的类型时，重写 Equals()的同时也基本上必须重写

GetHashCode()。[①]

哈希码（hash code）的目的是通过生成与对象的值相对应的数字来*高效平衡哈希表*[②]。虽然目前存在许多指导原则[③]，但下面列出了最简单的做法。

1. 直接依赖编译器为记录自动生成的实现（参见代码清单 9.4）。如果需要重写 Equals()，那么也大概率需要重写 GetHashCode()。所以，为什么不干脆使用记录呢？

2. 调用 System.HashCode 的 Combine()方法，指定每个标识字段（参见代码清单 9.22）。

代码清单 9.22　使用 Combine 方法重写 GetHashCode

```
public override int GetHashCode() =>
    HashCode.Combine(Degrees, Minutes, Seconds);
```

3. 将构成对象唯一性的字段作为元组元素，在元组上调用 ValueTuple 的 GetHashCode()方法，如代码清单 9.23 所示。（如果标识字段是数字，要注意不要错误地使用字段本身。相反，要使用它们的哈希码值。）

代码清单 9.23　在元组上调用 GetHashCode()方法

```
public override int GetHashCode() =>
    (Degrees, Minutes, Seconds).GetHashCode();
```

4. ValueTuple 会自己调用 HashCode.Combine()。所以，不费脑子的做法是用同一批标识字段创建一个 ValueTuple 元组，然后调用结果元组的 GetHashCode()成员即可。

[①] 译注：简单地说，无论如何都要确保相等性算法和对象哈希码算法一致（虽然这些都是“警告”，但请遵循“警告即 bug”的原则）

[②] 译注：也就是要提供良好的随机分布，使哈希表获得最佳性能。

[③] 在 StackOverflow 上面有一个很好的讨论，请参见 *http://tinyurl.com/56m2jut2*。

总之，除非有非常正当的理由无法这样做（例如，在 C#较老的版本中无法使用记录），否则在重写 GetHashCode()和 Equals()的时候应该使用记录。

幸好，一旦你决定重写 GetHashCode()，就可以按照以下良好的 GetHashCode()实现原则行事。（“*必须*”是指必须满足的要求，“*性能*”是指为了增强性能而需要采取的措施，“*安全性*”是指为了保障安全性而需要采取的措施。）

- *必须*：相等的对象必然具有相等的哈希码（例如，如果 a.Equals(b)，那么 a.GetHashCode() == b.GetHashCode()）。
- *必须*：即使对象的数据发生了改变，GetHashCode()也应该始终返回相同的值。许多时候，应该缓存方法的返回值来确保这一点。但是，一旦缓存了这个值，那么在在检查相等性的时候就千万别再用哈希码了。否则，两个完全一样的对象（其中一个具有发生了更改的标识属性的缓存哈希码）会返回错误的结果。
- *必须*：GetHashCode()不应抛出任何异常；GetHashCode()要始终返回一个值。
- *性能*：哈希码应该尽可能唯一。但是，由于哈希码只是返回一个 int，所以只要一种对象允许包含的值的数量超过一个 int 的容纳能力（这就几乎涵盖所有类型了），那么哈希码必然存在重复。一个很容易想到的例子是 long，因为 long 的取值范围大于 int，所以假如规定每个 int 值都只能标识一个不同的 long 值，那么肯定剩下大量 long 值没法标识。
- *性能*：可能的哈希码应当在 int 的范围内平均分布。例如，创建哈希码时如果没有考虑到字符串在拉丁语言中的分布主要集中在初始的 128 个 ASCII 字符上，就会造成字符串值的分布非常不平均，所以不能算是好的 GetHashCode()算法。
- *性能*：GetHashCode()的性能应该优化。GetHashCode()通常在 Equals()实现中用于“短路”一次完整的相等性比较（哈希码都不同，自然没必要进行完整的相等性比较了）。所以，当类型作为字典集合中的“键”类型使用时，会频繁调用该方法。
- *性能*：两个对象的细微差异应造成哈希值的极大差异。理想情况下，1 bit 的差异应造成哈希码平均 16 bits 的差异。 这有助于确保不管哈希表如何对哈希值进行“装桶”（bucketing）[1]，也能保持良好的平衡性。
- *安全性*：攻击者应难以伪造具有特定哈希码的对象。攻击手法是向哈希表中填写大量哈希为同一个值的数据。如果哈希表的实现不高效，就易于受到 DOS（拒绝服务）攻击。

你可能已经注意到了，许多原则是相互冲突的。事实上，很难有一种哈希算法既快又满足所有这些原则。和任何设计问题一样，好的解决方案必然是综合考虑的结果。

[1] 译注：StackOverflow 上面有关于哈希桶的一个很好的讨论，请参见 *http://tinyurl.com/28j9furn*。

9.4.3 自定义记录的行为

C#编译器明显为记录结构和记录类生成了大量代码。但是，其实所有行为都是可以自定义的。例如，本章前面说过，可以添加任何额外的成员，包括属性、字段、构造函数和方法等。更重要的是，可以为自动生成的成员提供自己的版本。只需自行编码具有匹配签名的记录成员，就可以用自己的行为替换系统生成的默认行为。例如，如果更喜欢只读属性而不是 init-only setter 属性，那么只需声明属性（或字段）来匹配位置属性的名称即可。或者，通过提供自己的拷贝构造函数，可以将自定义行为注入到 record 的克隆（通过 with 操作符）过程中。类似地，如果想改变相等性的语义（例如，只依据记录的部分属性/字段来判断相等性），那么可以定义自己的 Equals()方法。代码清单 9.24 展示了对记录进行自定义的几个例子。

代码清单 9. 24　自定义记录

```
public readonly record struct Angle(
    int Degrees, int Minutes, int Seconds, string? Name = null)
{

    public int Degrees { get; } = Degrees;

    public Angle(
        string degrees, string minutes, string seconds)
        : this(int.Parse(degrees),
              int.Parse(minutes), int.Parse(seconds))
    {
    }

    public override readonly string ToString()
    {
        string prefix =
            string.IsNullOrWhiteSpace(Name)?string.Empty : Name+": ";
        return $"{prefix}{Degrees}°  {Minutes}' {Seconds}\"";
    }

    // 修改 Equals()以忽略 Name 属性
    public bool Equals(Angle other) =>
        (Degrees, Minutes, Seconds).Equals(
            (other.Degrees, other.Minutes, other.Seconds));

    public override int GetHashCode() =>
        HashCode.Combine(Degrees.GetHashCode(),
            Minutes.GetHashCode(), Seconds.GetHashCode);

    #if UsingTupleToGenerateHashCode
    public override int GetHashCode() =>
```

```
            (Degrees, Minutes, Seconds).GetHashCode();
        #endif // UsingTupleToGenerateHashCode
    }
```

9.5 装箱

我们知道，值类型的变量直接包含它们的数据，而引用类型的变量包含对另一个存储位置的引用。但是，将值类型转换成它实现的某个接口或终极基类 object 时会发生什么。结果必然是对一个存储位置的引用。该位置表面上包含引用类型的实例，但实际包含值类型的值。这种转换称为**装箱**（boxing），它具有一些特殊行为。从值类型的变量（直接引用其数据）转换为引用类型（引用堆上的一个位置）会涉及以下几个步骤。

1. 首先在堆上分配内存。它将用于存放值类型的数据以及少许额外开销（SyncBlockIndex 和方法表指针）。需要这些开销，对象才能看起来像引用类型的托管对象实例。
2. 接着发生一次内存拷贝动作，将当前存储位置的值类型数据拷贝到堆上分配好的位置。
3. 最后，整个转换过程的结果是对堆上新存储位置的一个引用。

相反的过程称为**拆箱**（unboxing）。具体就是核实已装箱值的类型兼容于要拆箱成的值的类型，再拷贝堆中存储的值。这个过程的结果是堆上存储的值的一个拷贝。

装箱和拆箱之所以重要，是因为装箱会影响性能和行为。除了学习如何在 C#代码中识别它们之外，开发者还应通过查看 CIL，在一个特定的代码片段中统计 box/unbox 指令的数量。如表 9.1 所示，每个操作都有对应的指令。

表 9.1 CIL 中的装箱代码

C#代码	CIL 代码
`static void Main()` `{` `  int number;` `  object thing;` `  number = 42;` `  // 装箱` `  thing = number;` `  // 拆箱` `  number = (int)thing;`	`.method private hidebysig` `  static void Main() cil managed` `{` `  .entrypoint` `  // Code size  21 (0x15)` `  .maxstack  1` `  .locals init ([0] int32 number,` `  [1] object thing)` `  IL_0000: nop` `  IL_0001: ldc.i4.s  42` `  IL_0003: stloc.0` `  IL_0004: ldloc.0` `  IL_0005: box  [mscorlib]System.Int32` `  IL_000a: stloc.1`

```
return;                                 IL_000b: ldloc.1
}                                       IL_000c: unbox.any    [mscorlib]System.Int32
                                        IL_0011: stloc.0
                                        IL_0012: br.s         IL_0014
                                        IL_0014: ret
                                      } // end of method Program::Main
```

如果装箱和拆箱不是很频繁，那么性能问题不大。但有时装箱很容易被忽视，而且会非常频繁地发生，这就可能大幅影响性能。代码清单 9.25 和输出 9.1 展示了一个例子。ArrayList 类型维护着一个对象引用列表。所以，在列表中添加整数或浮点数会造成对值进行装箱以获取引用。

代码清单 9.25　容易忽视的 box 和 unbox 指令

```
public class DisplayFibonacci
{
    public static void Main()
    {
        // 故意使用 ArrayList 来演示装箱
        System.Collections.ArrayList list = new();

        Console.Write("输入 2～1000 的整数，我将生成这么多个斐波那契数：");
        string? inputText = Console.ReadLine();
        if (!int.TryParse(inputText, out int totalCount))
        {
            Console.WriteLine($"'{inputText}'不是一个有效的整数。");
            return;
        }

        if (totalCount == 7)  // 用一个魔法数字来执行测试
        {
            // 如果获取的值是 double，那么会触发异常
            list.Add(0);  // 要求转型为 double，或者附加'D'后缀。
                          // 无论转型还是使用'D'后缀，生成的 CIL 都是一样的。

        }
        else
        {
            list.Add((double)0);
        }

        list.Add((double)1);

        for (int count = 2; count < totalCount; count++)
```

```
        {
            list.Add(
                (double)list[count - 1]! +
                (double)list[count - 2]!);
        }

        // 用 foreach 来澄清装箱操作，而不是使用：
        // Console.WriteLine(string.Join(", ", list.ToArray()));
        foreach (double? count in list)
        {
            Console.Write($"{count}, ");
        }
    }
}
```

输出 9.1

```
输入 2～1000 的整数，我将生成这么多个斐波那契数: 42
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,
10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309,
3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155, 165580141,
```

代码编译后，会在 CIL 中生成 5 个 box 指令和 3 个 unbox 指令。

1. 前两个 box 指令是在对 list.Add()的初始调用中发生的。ArrayList.Add 方法的签名是 int Add(object value)[1]。所以，传给该方法的任何值类型都会被装箱。
2. 接着是在 for 循环内部 Add()调用中的两个 unbox 指令。ArrayList 的索引操作符（即[]）返回的总是 object，因为那正是 ArrayList 所包含的。但为了将两个值加到一起，需要将它们转换回 double。从 object 向值类型的转换作为 unbox 调用来实现。
3. 现在要获取加法运算的结果并将其放到 ArrayList 实例中，这再次造成了一次 box 操作。注意，前两个 unbox 指令和这个 box 指令是在一次循环迭代中发生的。
4. 在后续的 foreach 循环中，我们遍历 ArrayList 中的每一项，并把它们的值赋给 count。但是，ArrayList 包含的是对象引用，所以每次赋值都要执行 unbox 操作。
5. 对于 foreach 循环中调用的 Console.WriteLine()方法，它的签名是 void Console.Write(string format, object arg)。因此，每次调用都要执行从 double 到 object 的装箱。

每次装箱都涉及内存分配和拷贝；每次拆箱都涉及类型检查和拷贝。如果所有操作都用已拆箱的类型完成，就可以避免内存分配和类型检查。显然，可以通过避免许多装箱操作来

[1] 这里故意使用了一个 object 类型的集合，而不是使用强类型集合，例如泛型集合（参见第 12 章）。

提升代码性能。例如，上例的 foreach 循环完全可以使用 object 而不是 double 来改进性能。另一个改进是将 ArrayList 数据类型更改为泛型集合（参见第 12 章）。但无论如何，这里的重点在于，有的装箱操作很容易被人忽视，开发人员要留意那些可能反复装箱并大幅影响性能的情况。

还有一个问题是运行时的装箱。假如，假定修改最开始的两个 Add()调用，不是使用强制类型转换（或 double 字面值），而是直接在数组列表中插入整数。由于 int 会隐式转型为 double，所以这个修改似乎没什么大不了的。但是，在 foreach 循环中获取值以后，向 double 的转型就会出问题。这相当于是在 unbox 操作后，立即执行从已装箱 int 向 double 的内存拷贝。该操作要想成功，必须先转型为 int，再转型为 double。否则，代码会在执行时引发 InvalidCastException 异常。代码清单 9.26 展示了注释掉的一个类似的错误，在它后面才是正确的转型方式。

代码清单 9.6　必须先拆箱为基础类型

```
// ...

int number;
object thing;
double bigNumber;

number = 42;
thing = number;
// 错误: InvalidCastException
// bigNumber = (double)thing;
bigNumber = (double)(int)thing;
// ...
```

> **高级主题：lock 语句中的值类型**
>
> C#支持用于同步代码的 lock 语句。该语句实际编译成 System.Threading.Monitor 的 Enter()和 Exit()方法。两个方法必须成对调用。Enter()记录由其唯一的引用类型参数传递的一个 lock，这样使用同一个引用调用 Exit()时就可释放该 lock。值类型的问题在于装箱，所以每次调用 Enter()或 Exit()都会在堆上创建新值。而将一个拷贝的引用同另一个拷贝的引用比较总是返回 false。所以，无法将 Enter()与对应的 Exit()钩到一起。因此，不允许在 lock()语句中使用值类型。
>
> 代码清单 9.27 揭示了更多的运行时装箱问题，输出 9.2 展示了结果。

代码清单 9.27　容易忽视的装箱问题

```
interface IAngle
{
    void MoveTo(int degrees, int minutes, int seconds);
}

struct Angle : IAngle
{
    // ...

    // 注意:  这会使 Angle “可变”，有违设计规范
    public void MoveTo(int degrees, int minutes, int seconds)
    {
        _Degrees = degrees;
        _Minutes = minutes;
        _Seconds = seconds;
    }
    // ...
}
public class Program
{
    public static void Main()
    {
        // ...

        Angle angle = new(25, 58, 23);
        // 例 1: 简单装箱操作
        object objectAngle = angle;  // 装箱
        Console.Write(((Angle)objectAngle).Degrees);

        // 例 2: 拆箱，修改已拆箱的值，然后丢弃值
        ((Angle)objectAngle).MoveTo
            (26, 58, 23);
        Console.Write(", " + ((Angle)objectAngle).Degrees);

        // 例 3: 装箱，修改已装箱的值，然后丢弃对箱子的引用
        ((IAngle)angle).MoveTo(26, 58, 23);
        Console.Write(", " + ((Angle)angle).Degrees);

        // 例 4: 直接修改已装箱的值
        ((IAngle)objectAngle).MoveTo(26, 58, 23);
        Console.WriteLine(", " + ((Angle)objectAngle).Degrees);

        // ...
    }
}
```

输出 9.2

```
25, 25, 25, 26
```

代码清单 9.27 使用了 Angle 结构和 IAngle 接口。还要注意的是，实现 IAngle.MoveTo()使 Angle 成为可变（mutable）值类型。这带来了一些问题。通过演示这些问题，你将理解使值类型不可变（immutable）的重要性。

在代码清单 9.27 的“例 1”中，我们在初始化 angle 后把它装箱到 objectAngle 变量中。接着，“例 2”调用 MoveTo()将_Degrees 更改为 26。但正如输出演示的那样，这次不会发生实际的改变。这里的问题在于，为了调用 MoveTo()，编译器要对 objectAngle 进行拆箱，并且（根据定义）创建值的拷贝。值类型拷贝的是值，这正是它们叫*值类型*的原因。虽然结果值在执行时被成功修改，但值的这个拷贝会被丢弃。objectAngle 引用的堆位置未发生任何改变。

本章之前讲过，值类型的变量就像上面了写了值的纸。对值进行装箱相当于对纸进行复印，将复印件放到箱子中。对值进行拆箱相当于对箱子中的纸进行复印。编辑第二个复印件并不会影响箱子中的纸。

在“例 3”中，类似的问题在反方向上发生。这次不是直接调用 MoveTo()，而是将值强制转换为 IAngle。转换为接口类型会对值进行装箱，所以“运行时”将 angle 中的数据拷贝到堆，并返回对该箱子的引用。接着，方法修改被引用的箱子中的值。angle 中的数据保持未修改状态。

最后一种情况（例 4），向 IAngle 的强制类型转换是引用转换而不是装箱转换。值已通过向 object 的转换而装箱了。所以，这次转换不会发生值的拷贝。对 MoveTo()的调用将更新箱子中存储的_Degrees 值，代码的行为符合预期。

从这个例子可以看出，可变值类型很容易使人困惑，因为往往修改的是值的拷贝，而不是真正想要修改的存储位置。如果第一时间就避免使用可变值类型，就不会有这样的困惑了。

设计规范

AVOID mutable value types.

避免可变值类型。

高级主题：如何在方法调用期间避免装箱

任何时候在值类型上调用方法，接收调用的值类型（在方法主体中用 this 表示）必须

是变量而不是值，因为方法可能尝试修改接收者。显然，它必须修改接收者的存储位置，而不是修改接收者的值的拷贝再丢弃该拷贝。代码清单9.27的“例2”和“例4”演示了在已装箱值类型上调用方法时，这一事实对性能造成的影响。

在“例 2”中，拆箱逻辑上生成已装箱的值，而不是生成对包含已装箱拷贝的“堆上存储位置”（箱子）的引用。那么，哪个存储位置作为 this 传给方法调用呢？不可能是堆上箱子的位置，因为拆箱生成的是那个值的拷贝，而不是对存储位置的引用。

在需要值类型的变量但只有一个值可用的情况下，会发生两件事之一。要么是C#编译器生成代码来创建一个新的临时存储位置，将值从箱子拷贝到新位置，使临时存储位置成为所需的变量；要么不允许该操作并报错。本例采用第一个策略。新的临时存储位置成为该调用的接收者。MoveTo()方法完成修改后，该临时存储位置被丢弃。

每次“拆箱后调用”，无论方法是否真的修改变量都会重复该过程：对已装箱值进行类型检查，拆箱以生成已装箱值的存储位置，分配临时变量，将值从箱子拷贝到临时变量，再调用方法并传递临时存储位置。显然，如果不修改变量，好多工作都可避免。但C#编译器不知道一个方法会不会修改接收者，所以只好“宁错杀，不放过”。

在已装箱值类型上调用接口方法，所有这些开销都可避免。这时预期的接收者是箱子中的存储位置；如果接口方法要修改存储位置，那么修改的是已装箱的位置。执行类型检查、分配新的临时存储和生成拷贝的开销不再需要。相反，“运行时”直接将箱子中的存储位置作为结构的方法调用的接收者。

代码清单 9.28 调用了 int 值类型实现的 IFormattable 接口中的 ToString()的双参数版本。在本例中，方法调用的接收者是已装箱值类型，但在调用接口方法时不会拆箱。

代码清单 9. 28　避免拆箱和拷贝

```
int number;
object thing;
number = 42;
// 装箱
thing = number;
// 这里不会发生拆箱
string text = ((IFormattable)thing).ToString(
    "X", null);
Console.WriteLine(text);
```

你或许会问，如果将值类型的实例作为接收者来调用object声明的虚方法ToString()会发生什么？实例会装箱、拆箱还是什么？这要视情况而定：

- 如果接收者已拆箱，而且结构重写了 ToString()，那么将直接调用重写的方法。没必要以虚方式调用，因为方法不能被更深的派生类重写了。记住，所有值类型都隐式密封。
- 如果接收者已拆箱，而且结构没有重写 ToString()，那么必须调用基类的实现，该实现预期的接收者是一个对象引用。所以，接收者被装箱。
- 如果接收者已装箱，而且结构重写了 ToString()，那么将箱子中的存储位置传给重写方法而不拆箱。
- 如果接收者已装箱，而且结构没有重写 ToString()，那么将对箱子的引用传给基类的实现，该实现预期的正是一个引用。

9.6 枚举

对比如代码清单 9.29 所示的两个代码片段。

代码清单 9.29　比较整数 switch 和枚举 switch

```
int connectionState;
// ...
switch (connectionState)
{
    case 0:
        // ...
        break;
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    case 3:
        // ...
        break;
}

// ...
ConnectionState connectionState;
// ...
switch (connectionState)
{
    case ConnectionState.Connected:
        // 已连接
        break;
```

```
        case ConnectionState.Connecting:
            // 正在连接
            break;
        case ConnectionState.Disconnected:
            // 已断开
            break;
        case ConnectionState.Disconnecting:
            // 正在断开
            break;
    }
```

两者在可读性上有显著区别。在第二个代码片段中，各个 case 令人一目了然。不过，两者在运行时的性能完全一样，因为第二个代码片段在每个 case 中使用了**枚举值**。

枚举是可由开发者声明的值类型。枚举的关键特征是在编译时声明了一组具名常量值，这使代码更易读。代码清单 9.30 展示了一个典型的枚举声明。

代码清单 9. 30　定义枚举

```
enum ConnectionState
{
    Disconnected,
    Connecting,
    Connected,
    Disconnecting
}
```

> **注意：**用枚举替代布尔值能改善可读性。例如，SetState(DeviceState.On)的可读性明显优于 SetState(true)。

使用枚举值需要为其附加枚举名称前缀。例如，使用 Connected 值的语法是 ConnectionState.Connected。不要在枚举值名称中包含枚举名称，避免出现像 ConnectionState.ConnectionStateConnected 这样啰嗦的写法。根据约定，除了位标志（稍后讨论）之外的其他所有枚举名称应该是单数形式。例如，应该是 ConnectionState 而不是 ConnectionStates。

设计规范

DO NOT use the enum type name as part of the values name.

不要使用枚举类型名称作为值名称的一部分。

DO use an enum type name that is singular unless the enum is a flag.

要使用单数形式的枚举类型名称，除非该枚举作为标志（flag）使用。

枚举值实际是作为整数常量实现的。默认第一个枚举值是 0，后续每一项都递增 1。但可以显式地为枚举中的成员赋值，如代码清单 9.31 所示。

代码清单 9.31　定义枚举类型

```
enum ConnectionState : short
{
    Disconnected,
    Connecting = 10,
    Connected,
    Joined = Connected,
    Disconnecting
}
```

Disconnected 仍然具有默认值 0，Connecting 则被显式赋值为 10，所以它后面的 Connected 会被赋值为 11。随后，Joined 也被赋值为 11，也就是赋给 Connected 的值（此时不需要为 Connected 附加枚举名称前缀，因为目前正处在枚举的作用域内）。最后，Disconnecting 自动递增 1，所以值是 12。

枚举总是具有一个基础类型，可以是除 char 之外的任意整型。事实上，枚举类型的性能完全取决于基础类型的性能。默认基础类型是 int，但可以使用继承语法指定其他类型。例如在代码清单 9.31 中，使用的不是 int 而是 short。为了保持一致性，这里的语法模拟了继承的语法，但其实并没有真正建立继承关系。所有枚举的基类都是 System.Enum，后者又从 System.ValueType 派生。另外，这些类都是密封的（值类型的特点），不能从现有枚举类型派生以添加额外成员。

设计规范

CONSIDER using the default 32-bit integer type as the underlying type of an enum. Use a smaller type only if you must do so for interoperability; use a larger type only if you are creating a flags enum with more than 32 flags.

考虑使用默认 32 位整型作为枚举基础类型。只有出于互操作性或性能方面的考虑才使用较小的类型，只有创建标志（flag）数超过 32 个的标志枚举[①]才使用较大的类型。

枚举不过是具有基础类型的一组名称，对于枚举类型的变量，它的值并不限于声明中命名的值。例如，由于整数 `42` 能转型为 `short`，所以也能转型为 `ConnectionState`，即使它没有对应的 `ConnectionState` 枚举值。值能转换成基础类型，就能转换成枚举类型。

该设计的优点在于未来可以在新的 API 版本中为枚举添加新值，同时不会破坏早期版本。另外，枚举值为已知值提供了名称，同时允许在运行时分配未知的值。该设计的缺点在于编码时须谨慎，要主动考虑到未命名值的可能性。例如，如果将 `case ConnectionState.Disconnecting` 替换成 `default`，并认定 `default` case 唯一可能的值就是 `ConnectionState.Disconnecting`，那么就是不明智的。相反，应显式处理 `Disconnecting` 这一 `case`，而让 `default` case 报告一个错误，或者执行其他无害的行为。然而，正如前面讲到的，从枚举转换为基础类型以及从基础类型转换为枚举类型都涉及显式转型，而不是隐式转型。例如，假定方法签名是 `void ReportState(ConnectionState state)`，就不能调用 `ReportState(10)`。唯一的例外是传递 `0`，因为 `0` 能隐式转换为任何枚举。

虽然允许在代码未来的版本中为枚举添加额外的值，但这样做时要小心。在枚举中部插入枚举值，会使其后的所有枚举值发生顺移。例如，在 `Connected` 之前添加 `Flooded` 或 `Locked` 值，会造成 `Connected` 值改变。这会影响根据新枚举来重新编译的所有版本。不过，任何基于旧枚举编译的代码都会继续使用旧值。为了避免旧的枚举值发生改变，除了在列表末尾插入新的枚举值，另一个办法就是显式赋值。

设计规范

CONSIDER adding new members to existing enums, but keep in mind the compatibility risk.

考虑在现有枚举中添加新成员，但要注意兼容性风险。

AVOID creating enums that represent an “incomplete” set of values, such as product version numbers.

[①] 参见本章稍后的“枚举作为标志使用”一节。

避免创建代表“不完整”值（如产品版本号）集合的枚举。

AVOID creating “reserved for future use” values in an enum.

避免在枚举中创建“保留给将来使用”的值。

AVOID enums that contain a single value.

避免包含单个值的枚举。

DO provide a value of `0` (none) on simple enums, knowing that `0` will be the default value when no explicit initialization is provided.

要为简单枚举提供值 `0` 来代表无，强调若不显式初始化，`0` 就是默认值。[①]

> **高级主题：枚举之间的类型兼容性**
>
> C#不支持不同枚举数组之间的直接转型。但 CLR 允许，前提是两个枚举具有相同的基础类型。为了避开 C#的限制，技巧是先转型为 `System.Array`，如代码清单 9.32 末尾所示。
>
> **代码清单 9.32　枚举数组之间的转型**
>
> ```
> enum ConnectionState1
> {
>     Disconnected,
>     Connecting,
> ```

[①] 译注：意思是说，显式地在枚举定义中包含一个值为 `0` 的成员（通常命名为 `None` 或类似的名称），以清晰地表示“没有值”或“默认状态”。好处是，即使在变量未显式初始化的情况下，枚举变量仍有一个有效且明确的值（即 `0`），防止潜在的错误或未定义行为。

```
        Connected,
        Disconnecting
    }

    enum ConnectionState2
    {
        Disconnected,
        Connecting,
        Connected,
        Disconnecting
    }
    public class Program
    {
        public static void Main()
        {
            ConnectionState1[] states =
                (ConnectionState1[])(Array)new ConnectionState2[42];
        }
    }
```

这个技巧利用了 CLR 的赋值兼容性比 C#宽松这一事实。（还可以利用相同的技巧进行非法转换，例如从 int[]转换成 uint[]。）但是，使用该技巧务必慎重，因为 C#规范没有说这个技巧在不同 CLR 实现中都能发挥作用。

9.6.1 在枚举和字符串之间转换

枚举一个方便的地方是 ToString()方法（通过 System.Console.WriteLine()这样的方法来调用）会输出枚举值标识符，如代码清单 9.33 所示。

代码清单 9.33　向 Trace Buffer 写入枚举值

```
System.Diagnostics.Trace.WriteLine(
    $"连接目前{ConnectionState.Disconnecting}。");
```

上述代码将输出 9.3 的文本写入 Trace Buffer（跟踪缓冲区）。

输出 9.3

```
连接目前 Disconnecting。
```

从字符串向枚举的转换刚开始较难掌握，因为它涉及由 System.Enum 基类提供的一个静态方法。代码清单 9.34 展示了在不利用泛型（参见第 12 章）的前提下如何做，这个例子会输出“Idle”。

代码清单 9.34　使用 Enum.TryParse()将字符串转换为枚举

```
System.Diagnostics.ThreadPriorityLevel priority;
if(Enum.TryParse("Idle", out priority))
{
    Console.WriteLine(priority);
}
```

这个例子展示了如何在编译时确定枚举值，这类似于直接使用字面值（但实际上这里的值是通过字符串动态解析得到的）。TryParse()方法[①]（从技术上说是 TryParse<T>()）使用了泛型，但类型参数可以推断出来。所以，可以使用代码清单 9.34 的语法转换为枚举。除此之外，还有一个等价的 Parse()方法，会在失败时抛出异常。正是因为这个原因，所以除非肯定能成功，否则最好不要使用 Parse()。

不管是用“Parse”还是“TryParse”模式来编码，从字符串向枚举的转换都是不可本地化的。所以，如果本地化是一项硬性要求，只有对那些不公开给用户的消息，开发人员才可进行这种形式的转换。

设计规范

AVOID direct enum/string conversions where the string must be localized into the user’s language.

如果字符串必须本地化成用户的语言，那么**避免**直接在枚举和字符串之间转换。

9.6.2 枚举作为标志使用

开发人员许多时候不仅希望枚举值独一无二，还希望能对其进行组合以表示复合值。以 System.IO.FileAttributes 为例。如代码清单 9.35 所示，该枚举用于表示各种文件属性：只读、隐藏、存档等。在前面定义的 ConnectionState 枚举中，每个值都是互斥的。而 FileAttributes 枚举值允许而且本来就设计为自由组合。例如，一个文件可能同时只读和

[①] 从.NET Framework 4.0 开始提供。

隐藏。为了支持该行为，每个枚举值都是位置唯一的二进制位。

代码清单 9.35　枚举作为位标志

```
[Flags]
public enum FileAttributes
{
    None = 0,                       // 000000000000000
    ReadOnly = 1 << 0,              // 000000000000001
    Hidden = 1 << 1,                // 000000000000010
    System = 1 << 2,                // 000000000000100
    Directory = 1 << 4,             // 000000000010000
    Archive = 1 << 5,               // 000000000100000
    Device = 1 << 6,                // 000000001000000
    Normal = 1 << 7,                // 000000010000000
    Temporary = 1 << 8,             // 000000100000000
    SparseFile = 1 << 9,            // 000001000000000
    ReparsePoint = 1 << 10,         // 000010000000000
    Compressed = 1 << 11,           // 000100000000000
    Offline = 1 << 12,              // 001000000000000
    NotContentIndexed = 1 << 13,    // 010000000000000
    Encrypted = 1 << 14,            // 100000000000000
}
```

> **注意：**位标志枚举名称通常使用复数，因为它的值代表标志（flags）的一个集合。

我们使用按位 OR 操作符来连接（join）枚举值，使用 `Enum.HasFlags()`方法或按位 AND 操作符来测试特定位是否存在。代码清单 9.36 和输出 9.4 展示了这两种情况。[①]

代码清单 9.36　为标志枚举值使用按位 OR 和 AND

```
using System;
using System.IO;

public class Program
{
    public static void Main()
```

[①] 注意，`FileAttributes.Hidden` 这个枚举值在 Linux 上不起作用。

```
    {
        string fileName = @".enumtest.txt";

        System.IO.FileInfo enumFile = new(fileName);
        if (!enumFile.Exists)
        {
            enumFile.Create().Dispose();
        }

        try
        {
            System.IO.FileInfo file = new(fileName)
            {
                Attributes = FileAttributes.Hidden |
                    FileAttributes.ReadOnly
            };

            Console.WriteLine($"{file.Attributes} = {(int)file.Attributes}");

            // Linux 上仅支持 ReadOnly 属性
            // (Hidden 属性在 Linux 上不起作用)
            if (!file.Attributes.HasFlag(FileAttributes.Hidden))
            {
                throw new Exception("文件不是隐藏。");
            }

            if ((file.Attributes & FileAttributes.ReadOnly) !=
                FileAttributes.ReadOnly)
            {
                throw new Exception("文件不是只读。");
            }
        }
        finally
        {
            if (enumFile.Exists)
            {
                enumFile.Attributes = FileAttributes.Normal;
                enumFile.Delete();
            }
        }
    }
}
```

输出 9.4

```
ReadOnly, Hidden = 3
```

本例用按位 OR 操作符将示例文件同时设为只读和隐藏，测试完成后将文件删除。

注意，枚举中的每个值不一定只对应一个标志。完全可以主动为常用的标志组合定义额外的枚举值，如代码清单 9.3 7 所示。

代码清单 9. 37　为常用组合定义枚举值

```
[Flags]
enum DistributedChannel
{
    None = 0,
    Transacted = 1,
    Queued = 2,
    Encrypted = 4,
    Persisted = 16,
    FaultTolerant =
        Transacted | Queued | Persisted
}
```

一个好习惯是在标志枚举中包含值为 `0` 的 `None` 成员，因为无论枚举类型的字段，还是枚举类型的一个数组中的元素，其初始默认值都是 `0`。避免最后一个枚举值对应像 `Maximum`（最大）这样的东西，因为 `Maximum` 可能被解释成有效枚举值。要检查枚举是否包含某个值，请使用 `System.Enum.IsDefined()`方法。

设计规范

DO use the `FlagsAttribute` to mark enums that contain flags.

要用 `FlagsAttribute` 标记包含标志（flag）的枚举。

DO provide a `None` value equal to `0` for all flag enums.

要为所有标志枚举提供等于 `0` 的 `None` 值。

AVOID creating flag enums where the zero value has a meaning other than “no flags are set”.

避免标志枚举中的零值是除了“所有标志都未设置”之外的其他意思。

CONSIDER providing special values for commonly used combinations of flags.

考虑为常用标志组合提供特殊值。

DO NOT include "sentinel" values (such as a value called `Maximum`); such values can be confusing to the user.

不要包含"哨兵"值（如 `Maximum`），这种值会使用户困惑。

DO use powers of two to ensure that all flag combinations are represented uniquely.

要用 2 的乘方确保所有标志组合都不重复。[1]

> **高级主题：`FlagsAttribute`**
>
> 如果决定使用位标志枚举，那么枚举的声明应该用 `FlagsAttribute` 来标记。该特性应包含在一对方括号中（关于特性的更多信息，请参见第 18 章），并放在枚举声明之前，如代码清单 9.38 和输出 9.5 所示。
>
> **代码清单 9.38　使用 `FlagsAttribute`**
>
> ```
> // FileAttributes 在 System.IO 中定义
> /*
> [Flags]
> public enum FileAttributes
> {
>     ReadOnly = 0x0001,
>     Hidden   = 0x0002,
>     // ...
> ```

[1] 译注：创建用于表示位标志（bit flags）的枚举时，建议使用 2 的幂（即 1，2，4，8，16 等）作为枚举值。这样做确保了每个枚举值都是唯一的，并且可以组合使用不同的枚举值来表示多个标志。具体就是像本例这样，每个单独的文件属性仅一个二进制位为 1 即可。

```
    }
    */
    using System;
    using System.IO;

    public class Program
    {
        public static void Main()
        {
            string fileName = @"enumtest.txt";
            // ...
            FileInfo file = new(fileName);
            file.Open(FileMode.OpenOrCreate).Dispose();

            FileAttributes startingAttributes =
                file.Attributes;

            file.Attributes = FileAttributes.Hidden |
                FileAttributes.ReadOnly;

            Console.WriteLine("原本输出\"{1}\"，替换为\"{0}\"。",
                file.Attributes.ToString().Replace(",", " |"),
                file.Attributes);

            FileAttributes attributes =
                (FileAttributes)Enum.Parse(typeof(FileAttributes),
                file.Attributes.ToString());

            Console.WriteLine(attributes);

            File.SetAttributes(fileName,
                startingAttributes);
            file.Delete();
        }
    }
```

输出 9.5

```
原本输出"ReadOnly, Hidden"，替换为"ReadOnly | Hidden"。
ReadOnly, Hidden
```

该特性指出枚举值可以组合。它还改变了 ToString()和 Parse()方法的行为。例如，为用 FlagsAttribute 修饰的枚举调用 ToString()方法，会为已设置的每个枚举标志输出对应的字符串。在代码清单 9.38 中，file.Attributes.ToString()返回 ReadOnly, Hidden。而如果没有用 FlagsAttribute 修饰，返回的就是 3。如果两个枚举值相同，那么 ToString()返回第一个。但如前所述，使用需谨慎，因为无法本地

> 化。[①]
>
> 将值从字符串解析成枚举也是可行的。每个枚举值标识符都以逗号分隔。
>
> 注意，FlagsAttribute 不会自动分配唯一的标志值，也不会核实它们有唯一的值。那样做也没有意义，因为经常都需要重复值和组合值。相反，应该显式分配每个枚举项的值。

9.7 小结

本章首先对比了引用相等性和值相等性，以及在默认情况下，引用类型会在检查相等性时使用引用相等性，而值类型会检查值本身（或者对于自定义值类型，检查自定义值类型内的数据）。但要记住的是，与自定义类的数量相比，自定义值类型（即 struct）相对要少得多，通常仅限于要与非托管代码互操作的代码。

接着，我们介绍了记录结构（record struct）和记录类（record class），指出虽然编译器为两者生成的 CIL 代码大致相同，但也存在一些重要差异。最重要的是，记录结构和记录类都重写了相等性以使用值类型相等性。这个设计非常有用，因为要为某个类型实现值相等性时，自己实现既复杂又容易出错。两者 CIL 代码另一个相似之处是，它们都使用“位置参数”来生成属性、构造函数、ToString()实现和解构函数 Deconstruct()。但是，两者的不同之处在于，默认情况下记录结构是可读写的，而记录类默认为所有位置参数（对应的属性）使用 init-only setter。话虽如此，由于“可变”值类型很容易引入混乱或者有 bug 的代码，而且我们通常用值类型来建模“不可变”的值，所以最佳实践是用 readonly 修饰符来声明记录结构。此外，只有记录类支持继承——虽然只能从其他记录类继承。实际上，所有结构（无论 record struct 还是传统 struct）都不支持继承，因为所有值类型都隐式密封。

值类型必须作为引用类型以多态的方式处理时，会发生装箱。装箱容易使人犯迷糊，容易在执行时（而非编译时）出问题。虽然需要清楚认识它以避免问题，但也不必过于紧张。值类型很有用，具有性能上的优势，不要惧怕用它。值类型渗透到本书几乎每一章，容易出问题的时候并不多。虽然罗列了装箱可能发生的问题，并用代码进行了演示，但现实中其实很少遇到相同情形。只要遵守“不要创建可变值类型”这一设计规范，就可以避免其中的多数问题。而这也正是你在语言内建的值类型中没有遇到这些问题的原因。

或许最容易出的问题就是循环内的反复装箱操作。不过，使用泛型（第 12 章）可以显著

[①] 译注：记住，将特性应用于源代码中的目标元素时， C#编译器允许省略 Attribute 后缀，这样可以少打一点字，并提升源代码的可读性。正是因为这个原因，所以在应用 FlagsAttribute 的时候，简单地写[Flags]就可以了。

减少装箱。而且即使不用泛型，该问题对性能造成的影响也微乎其微，除非你确定某个算法因为装箱而引入了性能瓶颈。

本章还介绍了枚举。枚举类型是许多编程语言中的标准结构。它们有助于提高 API 的可用性和代码的可读性。

下一章介绍更多设计规范来帮助你创建“良好形式”（well-formed）的值类型和引用类型。首先讨论的是如何定义操作符重载方法。该主题同时适用于结构和类，但它更重要的意义还是在于完善结构定义，使其具有“良好形式”。