- C# 入门及进阶的首选之作
- 内容全面翔实，讲解精彩权威
- 全新升级版，涵盖 C# 5.0

PEARSON

Essential C# 5.0

C#本质论（第4版）



[美] Mark Michaelis　Eric Lippert　著
周靖　译

人民邮电出版社
POSTS & TELECOM PRESS

C#本质论（第4版）

作为历年来深受读者欢迎的 C# 权威指南，本书针对 C# 5.0 和 .NET 4.5 引入的新功能和新编程模式进行了全面升级，适合各种水平的 C# 程序员阅读。它将向读者展示如何编写简洁、强大、健壮、安全和易于维护的 C# 代码。微软“最有价值专家”Mark Michaelis 和 C# 首席开发师 Eric Lippert 对这一语言进行了全面而深入的探讨，为读者进行高效软件开发打好基础。

作者用简洁、可下载的代码示例阐释关键构造。每章开头的思维导图更能帮读者理清头绪。这一版还新增了大量 C# 编码规范，编写 C# 代码的最佳实践一目了然。

书中对初学者主题和高级主题进行了明显区分。对于 C# 的新手，可通过本书快速上手；对于有经验的 C# 开发者，可通过学习 C# 5.0 的创新技术（如 async/await 模式）来了解如何应对当今最复杂的编程挑战。

- 掌握 C# 数据类型、操作符、控制流、方法和参数
- 最大化利用 C# 面向对象构造，包括类、继承、接口等
- 构建可靠和高效的异常处理机制
- 利用泛型、委托、Lambda 表达式和事件降低代码复杂度
- 利用反射和特性进行动态编程
- 使用查询表达式查询几乎任何类型的数据
- 创建自定义集合来处理业务对象
- 理解 .NET 4.5 开发平台上的 CLI 和 C#
- 理解宣告式编程、嵌入元数据、反射和特性
- 全面掌握多线程处理和同步，包括新的 async/await 模式
- 讨论面向 Windows 8 的 WinRT 和 C# 编程
- 通过 P/Invoke、指针和直接内存操作来实现与其他语言代码的互操作
- 理解 C# 程序和底层“运行时”的关联

PEARSON
www.pearson.com

人民邮电出版社－信息技术分社
http://weibo.com/ptpitbooks

美术编辑：王建国

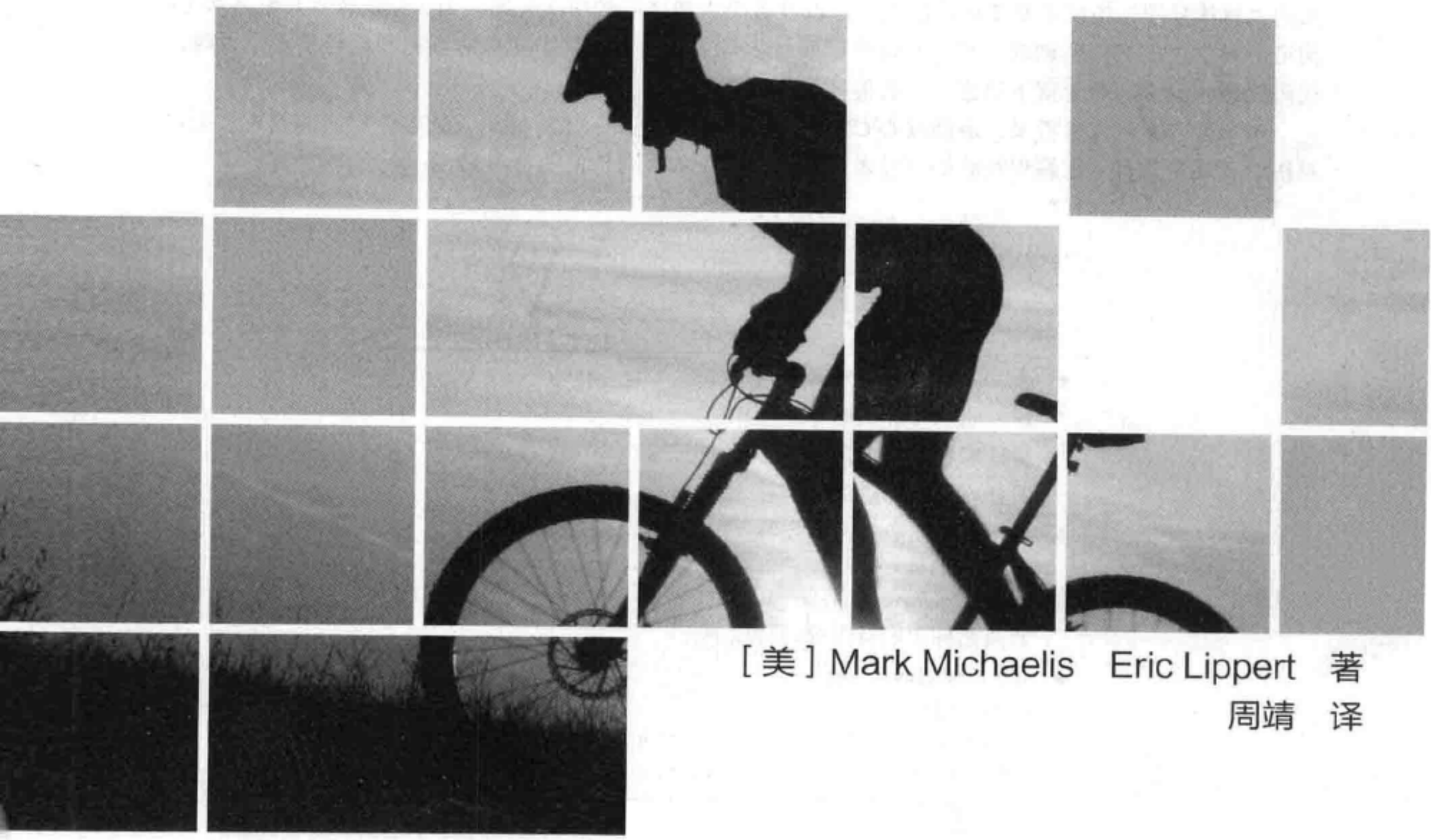
分类建议：计算机／程序设计／.NET 技术
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33675-0

ISBN 978-7-115-33675-0
定价：108.00 元

Essential C# 5.0

C#本质论（第4版）



[美] Mark Michaelis　Eric Lippert　著
周靖　译

人民邮电出版社
北京

图书在版编目（CIP）数据

C#本质论 : 第4版 / (美) 米凯利斯(Michaelis,M.), (美) 利珀特 (Lippert,E.) 著 ; 周靖译. -- 北京 : 人民邮电出版社, 2014.3
书名原文: Essential C# 5.0
ISBN 978-7-115-33675-0

Ⅰ. ①C… Ⅱ. ①米… ②利… ③周… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第273860号

内 容 提 要

这是 C# 领域中一部广受好评的名作，作者用一种易于理解的方式详细介绍了 C# 语言的各个方面。全书共有 21 章和 4 个附录，介绍了 C# 语言的数据类型、操作符、方法、类、接口、异常处理等基本概念，深入讨论了泛型、迭代器、反射、线程和互操作性等高级主题，还介绍了 LINQ 技术，以及与其相关的扩展方法、分部方法、Lambda 表达式、标准查询操作符和查询表达式等内容。每章开头的“思维导图”指明本章要讨论的主题，以及各个主题之间的层次关系。为了帮助读者理解各种 C# 构造，书中用丰富的示例演示每一种特性，而且为每个概念都提供了相应的规范和最佳实践，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。

本书是一本语言参考书，遵循核心 C# 5.0 语言规范，适合对 C# 感兴趣的各个层次的读者。无论对初学者还是具有一定编程经验的开发者，本书都是一本很有价值的参考书。

◆ 著　　[美] Mark Michaelis　Eric Lippert
译　　周　靖
责任编辑　杨海玲
责任印制　程彦红　杨林杰
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷
◆ 开本：787×1092　1/16
印张：42.25
字数：901 千字　　2014 年 3 月第 4 版
印数：13 501 – 17 000 册　　2014 年 3 月北京第 1 次印刷
著作权合同登记号　图字：01-2012-9275 号

定价：108.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

Authorized translation from the English language edition, entitled *Essential C# 5.0*, 9780321877581 by MICHAELIS, MARK; LIPPERT, ERIC, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright © 2013 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD. and POSTS & TELECOM PRESS Copyright © 2014.

本书中文简体字版由Pearson Education Asia Ltd.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

献给我的家人Elisabeth、Benjamin、Hanna和Abigail。

感谢你们容忍我花费漫长的时间来写作，

而且经常是在你们最需要我的时候。谢谢你们！

序

强强联手，造就了这本伟大的C#书籍！Mark Michaelis的《C#本质论》系列早已成为经典，如今更与著名的C#博主Eric Lippert携手，使之升华为无与伦比的大师级著作。

一般人的印象是Eric写博客，而Mark写书，但我刚认识他们的时候可不是这样的。

2005年LINQ（语言集成查询）公布时，我才刚加入Microsoft，正好见证了PDC会议上的令人激动的公开发布。虽然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。那个时候人人都在谈论它，宣传小册子满天飞。那是C#和.NET的大日子，至今仍然令人难忘。

但会场的实验室区域却相当安静，在那儿，人们可以自行按照精心编写的实验指南进行技术预览。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。他在做自己的实验，梳理文档，和别人沟通，忙着收集自己的照片。

作为C#社区的新人，我在那次会议上见到了许多名人。但老实说，后来都记不大清楚了。唯一记得住的就是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，我还没有完全了解它。"他希望完整地理解并消化这一技术，而在这之前，他不希望别人的先入之见影响他的思考。

所以并没有像我预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而且颇有裨益。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了。对我们这些语言设计者来说，Mark是最有价值的社区成员，因为他非常聪明，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。如果我们开发的某种技术能够通过Mark的测试，就没有什么好担心的了！

这些特质也使Mark成为一名出色的作家。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。

Eric是C#团队中和我共事7年的同事。他在这里的资格比我老得多。回想第一次见到他时，他正在向团队解释如何理清乱局。确切地说，当时C#编译器代码库正需要在架构上进行一些重大调整，而新功能却很难加入进来——我们迫切需要LINQ来解决这些问题。Eric已经调查好了我们需要有什么样的架构（阶段啊阶段！当时甚至根本就没有这个概

念！）。更重要的是，如何逐步跟进。令人惊讶的是，虽然这个主题是如此复杂，作为团队和代码库新人的我，居然立即就明白了他所讲的东西！

从博客就能看出，他讲东西超清楚，而且结构合理，读者能快速地把一个问题搞明白，除了能收获解决方案之外，阅读过程同样令人愉悦。还不止如此！每次Eric钻研一个复杂的问题并与团队分享见解时，他的电子邮件都是那么一丝不苟又充满乐趣。基本上不可能忽视Eric提出的任何问题，因为你会迫不及待地想要看完他的文章。除此之外，他的文字还非常大气！所以，我超喜欢看他的东西，包括经常更新的博客文章。作为C#编译器团队和语言设计团队的一员，他为我们带来的愉悦和帮助真是太多了。

总之，能和这两位出色的伙伴共事，我深感荣幸。Eric帮我理清思路，Mark帮我说出真心话。他们帮助读者认清楚问题并掌握解决方案。通过分别主攻C#的“内”和“外”，本书的这个版本在完整性方面达到了前所未有的高度。没有人能像这两位大师一样帮助你正确地理解C# 5.0。

请好好享用本书！

——Mads Torgersen

微软公司C#项目经理

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范型的重大转变。每一种范型都是以前一种为基础的，其宗旨都是增强代码的组织，并降低复杂性。本书将带领你体验这样的范型转变过程。

本书开始的几章指导你学习**顺序编程结构**。在这种编程结构中，语句是按照编写的顺序来执行的。这种结构的问题在于，随着需求的增加，复杂性会按指数级增长。为了降低复杂性，将代码块转变成方法，产生了**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，而不必在程序中重复这些代码。然而，即使有这种结构，程序还是会很快变得臃肿不堪，需要进行进一步抽象。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5章进行讨论。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进行初级的声明性编程[①]（第17章）。

本书有以下3种主要职能。

- 全面讲述C#语言，其内容已经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的基础。
- 对于已经熟悉了C#的读者，本书探讨了一些较为复杂的编程范型，并深入讨论了语言最新版本（C# 5.0和.NET 4.5）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方面的“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]思想的笃信者，我希望即使是一名刚开始学习编程的新手，在学到本书第2章末尾的时候，也能动手开始写基本的C#代码。

有许多主题都没有在本书中进行讨论。你在本书找不到ASP.NET、ADO.NET、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 Framework有关，但它们都值得用专门的书分专题进行讲述。幸运的是对于这些主题，都已经有丰富的图书供读者选择了。

① 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。——译者注

② 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。——译者注

本书的重点在于C#以及基类库中的类型。读完本书之后，你在上述任何领域继续深入学习都会有游刃有余的感觉。

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，不因使用assembly、link、chain、thread和fusion等字眼而打击初学者的信心，否则许多人会误以为这是一本讲冶金而不是程序设计的书[①]。本书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”的开发者。但我还是认真编排了本书的内容，以便使各种层次的开发者都能够从中获益。

- *初学者*：假如你是编程新手，本书将帮助你从入门级程序员过渡成为C#开发者，消除以后在面临任何C#编程任务时的害怕心理。本书不仅要教会你语法，还要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在动手中学习，而不是等熟知了它的所有“理论”之后再动手。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者很容易上手的。到第4章结束时，这些开发者应该可以开始写基本的控制流程序。然而，要成为真正的C#开发者，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#开发者必须熟练地从对象及其关系的角度来思考问题。为此，第5章的“初学者主题”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，软件工程师们应该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”理念的开发者*：C++和Java程序员以及许多有经验的Visual Basic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规范及模式。你教别人学C#时，本书也颇有助益。随着C# 3.0、C# 4.0和C# 5.0的出现，一些最重要的增强包括：
  - 隐式类型的变量（参见第2章）；
  - 扩展方法（参见第5章）；
  - 分部方法（参见第5章）；
  - 匿名类型（参见第11章）；
  - 泛型（参见第11章）；

---

① 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”。——译者注

② 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。——译者注

- Lambda语句和表达式（参见第12章）；
- 表达式树（参见第12章）；
- 标准查询操作符（参见第14章）；
- 查询表达式（参见第15章）；
- 动态编程（参见第17章）；
- 用任务编程库和async进行多线程编程（参见第18章）；
- 用PLINQ进行并行查询处理（参见第18章）；
- 并发集合（第19章）。

考虑到许多人还不熟悉这些主题，因此本书围绕它们展开了详细的讨论。涉及高级C#开发的还有“指针”这一主题，该主题将在第21章讨论。就算是有经验的C#开发者，也未必能很透彻地理解这一主题。

本书特色

本书是一本语言参考书，它遵循核心C#5.0 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规范和最佳实践，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。

为了增强可读性，所有代码均进行了特殊格式处理，而且每一章的内容都使用思维导图来概括。

C#编码规范

本书新版本最重大的改进之一就是增加了大量编码规范，例如第16章中的一个规范如下所示：

> **规范**
>
> 要确保相等的对象有相同的散列码。
>
> 要确保对象的散列码在一个散列表中永不变化。
>
> 要确保散列算法快速产生良好分布的散列码。
>
> 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时宜地写出最高效代码的专家的区别，关键就是这些编码规范。专家不仅让代码通过编译，还遵循最佳实践，降低产生bug的概率，并使代码的维护变得更容易。编码规范强调了一些关键原则，开发时务必注意。

代码示例

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，其中包括Mono、Rotor和Microsoft .NET平台。除了在解释只和某些平台有关的重要概念的地方（例如，解释如何正确处理Windows单线程用户界面），我很少使用平台或厂商特有的库。

下面是一个示例代码清单。

代码清单1-9　变量的声明和赋值

```
class MiracleMax
{
  static void Main()
  {
        数据类型
        string max;
               变量
      max = "Have fun storming the castle!";

      System.Console.WriteLine(max);
  }
}
```

下面简单介绍一下代码格式。

- 注释以斜体表示。

```
/*Display a greeting  to the console
   using composite formatting.*/
```

- 关键字加粗。

```
static void  Main()
```

- 有的代码被突出显示，是为了指明这些代码与之前列出的有所区别，或是为了演示正文中介绍的概念。

```
    System.Console.Write /* No new line */ (
```

突出显示的内容可能是一整行，也可能仅仅是一行中的几个字符。

```
System.Console.WriteLine(
      "Your full name is {0} {1}.",
```

- 不完整的程序清单包含一个省略号，表示无关的代码已省略。

```
//…
```

- 在代码清单之后，列出了对应的控制台输出，如下例所示：

输出1-4

```
>HeyYou.exe
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
```

- 执行程序时要由用户输入的内容加粗显示。

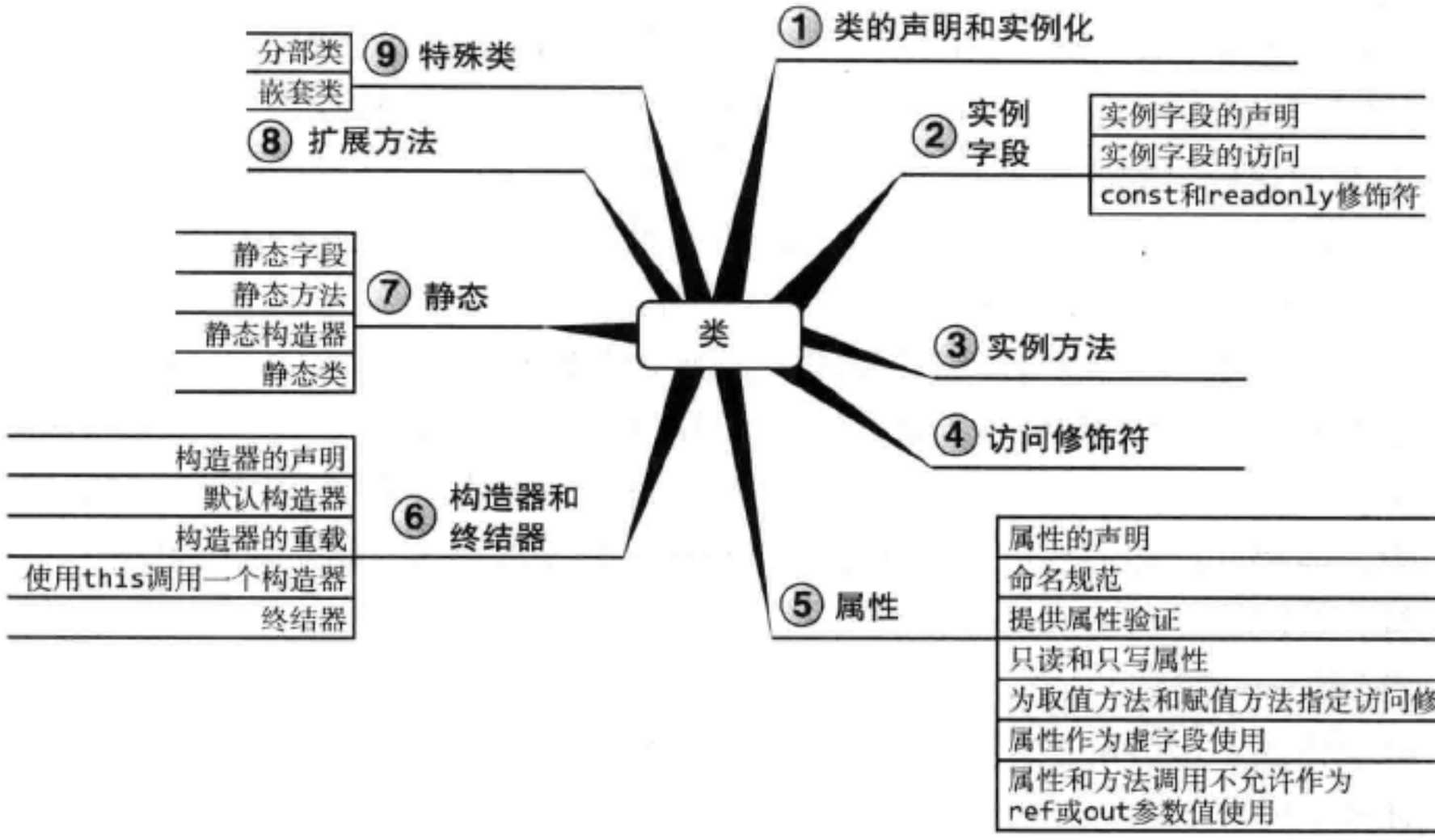
虽然提供完整的代码示例可以方便地复制到自己的程序中，但这样做会降低学习特定主题的意义。因此，需要自行修改代码示例，然后再把它们集成到自己的程序中。代码示例中最大的问题是省略了像异常处理这样的错误检查。此外，书中的代码示例没有显式地包含using System语句；在所有的例子中，这个语句都是必需的。

请访问intellitect.com/essentialcsharp和informit.com/mswinseries下载示例代码。[①]

① 本书中文版的资源下载和勘误也通过译者博客（transbot.blog.163.com）提供。——译者注

思维导图

每章开头都有一幅**思维导图**[①]。作为提纲，它的作用是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5章）。



每一章的主题显示在思维导图的中心，高级主题围绕中心展开。利用思维导图，读者可以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

根据自己的经验水平，书中特殊的代码块和页面边缘的灰色竖线条可以帮你轻松地找到适合自己的内容。

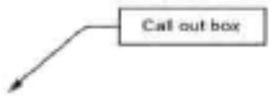
- 初学者主题：特别针对入门级程序员提供的定义或解释。
- 高级主题：可以让有经验的开发者将注意力放在他们最关心的内容上。
- 标注：用标注框[②]来强调关键原则，使读者对其重要性一目了然。
- 语言对比：分散在正文中的补充内容，描述了C#和其他语言的关键差异，为熟悉其他语言的读者提供指引。

本书内容组织

总地说来，软件工程的宗旨就是对复杂性进行管理。本书正是基于这个宗旨来组织内容的。第1章～第4章介绍的是结构化编程，学习了这些内容后，你可以立即开始写一些功能简单的代码。第5章～第9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11章～第13章介绍了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。在理解了它们之后，你可以更加轻松地理解如何通过反射和特性来实现动态编程。在后续的章节中，将广泛运用

---

① 思维导图，即mind map，又称脑图、心智图，其作用是帮助你学习、组织和存储你想要的所有信息。它以自然的方式对信息进行分类，使你能够立即得到你想要的一切。你可以将其想象成一幅帮助自己记忆和思考的思维路线图。——译者注

② 标注框，即callout box，是一种需要准确排版的书稿元素。比如： 

。——译者注

反射和特性来实现线程处理和互操作性。

本书最后专门用一章（第21章）来讲述CLI。这一章针对开发平台对C#语言进行描述。之所以要放到最后，是因为它并不是C#特有的，而且不会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（**使用黑体**的章号表明那一章的内容在C# 3.0~C#5.0版本中都有）。

- 第1章——C#概述：这一章在展示了用C#编写的HelloWorld程序之后，进一步剖析了C#。这应当能使读者熟悉C#程序的“外观和感觉”。这一章提供了让读者编译和调试自己的程序所需的足够多的细节。此外，还简单描述了执行C#程序的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，这一章介绍了C#的两种基本数据类型，即值类型和引用类型，另外还讨论了类型间的转换以及C#对数组的支持。
- 第3章——操作符和控制流：计算机擅长重复性操作，为了利用计算机的这个能力，需要知道如何在自己的程序中添加循环和条件逻辑。这一章还讨论了C#操作符、数据转换以及预处理指令。
- **第4章**——方法和参数：这一章研究了有关方法及其参数的细节，其中包括通过参数来传值、传引用和返回数据。C# 4.0添加了对默认参数的支持，本章解释了如何使用它们。
- **第5章**——类：前面已经学过了类的基本构成元素，这一章将这些构造合并到一起，从而获得具有完整功能的类型。类是面向对象技术的核心，它为一“类”对象定义了模板。
- 第6章——继承：虽然继承是许多开发者的基本编程手段，但C#提供了独特的构造，比如new修饰符。这一章讨论了继承语法的细节，其中包括重写（overriding）。
- **第7章**——接口：这一章展示了如何利用接口来定义类之间的“可以进行版本控制的交互契约”（versionable interaction contract）。C#同时包含显式和隐式的接口成员实现，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。
- 第8章——值类型：尽管不如引用类型那么流行，但有些情况下仍然有必要定义行为类似于C#内置基本类型的值类型。这一章要介绍如何定义结构（structure），同时揭示它们的特性。
- 第9章——合式类型：这一章讨论了一些更高级的类型定义，解释了如何实现操作符，比如“+”和转型操作符，并描述了如何将多个类封装到一个库中。除此之外，这一章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10章——异常处理：这一章是对第4章引入的异常处理机制的一个延伸讨论，描述了如何利用异常层次结构来创建自定义异常。此外，它还强调了异常处理的一些最佳实践。
- **第11章**——泛型：从某种意义上说，泛型或许是C# 1.0缺少的一个最重要的特性。这一章全面讨论了自2.0引入的这个特性。除此之外，C# 4.0增加了对协变和逆变的支持。本章将在泛型的背景中探讨它们。
- **第12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）有了显著的不同，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。通过这一章的学习，你将知道Lambda表达式是在委托的基础上构建起来的，它提供了比委托更加优雅和简洁的语法。本章的内容是第14章讨论的新的集合API的基础。

- 第13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。本章还探讨了匿名方法，这也是C# 2.0新增的。
- **第14章**——支持标准查询操作符的集合接口：我们通过讨论新的Enumerable类的扩展方法，向你介绍C# 3.0引入的一些简单但又非常强大的改变。Enumerable类使一个全新的集合API成为可能，这个API称为“标准查询操作符”，本章对它进行了详细讨论。
- **第15章**——使用查询表达式的LINQ：如果只使用标准查询操作符，就会形成让人难以辨认的长语句。然而，查询表达式提供了一种类似SQL风格的语法，能够有效地解决这个问题。这一章会详细讨论这种表达式。
- **第16章**——构建自定义集合：在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的集合。本章讨论了具体如何做；同时，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17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，特性使说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。这一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。这一章还讨论了如何通过基类库（Base Class Library，BCL）中的序列化框架来实现文件的输入和输出。C# 4.0增加了一个新的关键字，即dynamic。该关键字将所有类型检查都移至运行时进行，因而极大扩展了C# 能做的事情。
- **第18章**——多线程处理：大多数现代的程序都要求使用线程来执行长时间运行的任务，还要确保对并发的事件进行快速响应。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是一项复杂的任务。这一章讨论了如何操纵线程，并讲述了如何采取一些必要的措施来防止将多线程应用程序弄得一团糟。
- **第19章**——线程同步：这一章以第18章为基础，演示了如何利用一些内建的线程处理模式来简化对多线程代码的显式控制。
- **第20章**——平台互操作性和不安全的代码：必须认识到的是，C#是相对年轻的一种语言，许多现有的代码是用其他语言写成的。为了用好这些现有的代码，C#通过P/Invoke提供了对互操作性（非托管代码的调用）的支持。除此之外，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的权限才能运行，但它具有与C风格的API完全兼容的能力。
- 第21章——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。这一章讨论了C#程序与底层“运行时”及其规范的关系。
- 附录A——下载和安装C#编译器与CLI平台：这个附录介绍了如何安装微软.NET和

Mono，它们是编译和运行C#代码的基础平台。

- 附录B——Tic-Tac-Toc源代码清单：第3章和第4章用到的程序在这里提供了完整源代码。
- 附录C——使用TPL和C# 5.0之前的多线程处理模式：这个附录详细说明了在使用C# 5.0和/或任务并行库（TPL）之前的版本进行开发时的多线程处理模式。
- 附录D——C# 5.0的Async/Await模式之前的计时器：这个附录描述了在.NET 4.5/C# 5.0之前使用的三种不同类型的计时器。

希望本书成为你学习和掌握C#技能的一个好帮手。另外，以后需要了解C#的一些特殊主题及其内部工作原理的时候，本书也是一本出色的参考书。

——Mark Michaelis

IntelliTect.com/mark

Twitter: @Intellitect, @MarkMichaelis

致谢

世界上没有任何一本书是作者单枪匹马就能出版的，在此，我要向此过程中帮助过我的所有人致以衷心的感谢。

表达感激之情的顺序并不重要，我是想到谁就感谢谁。到现在为止，为了让我顺利完成此书，我的家人做出了巨大的牺牲。在Banjamin、Hanna和Abigail眼中，他们的爸爸经常因为此书而无暇顾及他们，但Elisabeth承受的更多。家里的大事小情全靠她一个人，她独自承担家庭的重任。我原本希望本书每一次出版都变得更容易一些，但遗憾的是，实情并非如此。随着孩子们越来越大，生活越来越紧张和忙碌，没有我，Elisabeth几乎时时刻刻都在承受高强度的压力。我感到万分抱歉，谢谢你！

为保证本书技术上的准确性，许多技术编辑对本书中的各章都进行了仔细审阅。我常常惊讶于他们的认真程度，任何不易察觉的小错误都逃不过他们的火眼金睛，他们是Paul Bramsman、Kody Brown、Ian Davis、Doug Dechow、Gerard Frantz、Thomas Heavey、Anson Horton、Brian Jones、Shane Kercheval、 Angelika Langer、Eric Lippert、John Michaelis、Jason Morse、Nicholas Paldino、Jon Skeet、Michael Strokesbary、Robert Stokesbary、John Timney和Stephen Toub。还要感谢Mandy Frei辛苦记录大量再版时需要的改动。

Eric给了我太多的惊奇。他对C#术语的掌握程度令人“望而生畏”，我非常欣赏他的修改，尤其是他在术语方面表现出的力求完美。他对C# 3.0相关章节做了不小的改进，在本书的第2版中，我唯一感到遗憾的就是未能让他审阅所有章节。然而，这个遗憾终于得到了弥补。Eric兢兢业业地审阅了这一版的每一章，他审得非常仔细，也非常严谨。正是因为他的辛勤付出，才使本书变得比前两版还要好，我对此致以由衷的感谢。谢谢你，Eric！我想象不出还有谁能比你做得更出色。正因为你，本书才真正实现了从“很好”到“极好”的飞越。

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专门审阅了重写的（嗯，这是第三次了）关于多线程的两章，并重点检查了C# 5.0的异步支持。谢谢你，Stephen！

感谢Addison-Wesley的所有员工，感谢他们在与我合作期间表现出来的耐心，容忍我将注意力频频转移到书稿之外的其他事情。感谢Elizabeth Ryan、Audrey Doyle、Vicki Rowland、Curt Johnson和Joan Murray。尤其要感谢Joan，虽然很多次我不仅交稿延误，而且回邮件还特别慢，但她却总是富有耐心。

目录

第1章 C#概述 …… 1

1.1 Hello, World …… 1
1.2 C#语法基础 …… 3
1.2.1 C#关键字 …… 3
1.2.2 标识符 …… 4
1.2.3 类型定义 …… 5
1.2.4 Main …… 6
1.2.5 语句和语句分隔符 …… 7
1.2.6 空白 …… 8
1.2.7 使用变量 …… 8
1.2.8 数据类型 …… 9
1.2.9 变量的声明 …… 9
1.2.10 变量的赋值 …… 10
1.2.11 变量的使用 …… 11
1.3 控制台输入和输出 …… 11
1.3.1 从控制台获取输入 …… 11
1.3.2 将输出写入控制台 …… 12
1.3.3 注释 …… 14
1.3.4 托管执行和公共语言基础结构 …… 16
1.3.5 C#和.NET版本 …… 17
1.3.6 CIL和ILDASM …… 18
1.4 小结 …… 20

第2章 数据类型 …… 21

2.1 基本数值类型 …… 21
2.1.1 整数类型 …… 22
2.1.2 浮点类型 …… 23
2.1.3 decimal类型 …… 23
2.1.4 字面值 …… 24
2.2 更多基本类型 …… 27
2.2.1 布尔类型 …… 27
2.2.2 字符类型 …… 27
2.2.3 字符串 …… 29
2.3 null和void …… 34
2.3.1 null …… 34
2.3.2 void …… 35
2.4 类型的分类 …… 37
2.4.1 值类型 …… 37
2.4.2 引用类型 …… 37
2.5 可空修饰符 …… 38
2.6 数据类型之间的转换 …… 39
2.6.1 显式转型 …… 39
2.6.2 隐式转型 …… 41
2.6.3 不使用转型操作符的类型转换 …… 42
2.7 数组 …… 43
2.7.1 数组的声明 …… 44
2.7.2 数组的实例化和赋值 …… 45
2.7.3 数组的使用 …… 48
2.7.4 字符串作为数组使用 …… 52
2.7.5 常见数组错误 …… 53

2.8 小结 …… 55
第3章 操作符和控制流 …… 57
3.1 操作符 …… 57
3.1.1 一元操作符正和负 …… 58
3.1.2 二元算术操作符 …… 58
3.1.3 复合赋值操作符 …… 64
3.1.4 递增和递减操作符 …… 65
3.1.5 常量表达式和常量符号 … 68
3.2 控制流程概述 …… 69
3.2.1 if语句 …… 70
3.2.2 嵌套if …… 71
3.3 代码块 …… 73
3.4 代码块、作用域和声明空间 …… 74
3.5 布尔表达式 …… 76
3.5.1 关系操作符和相等性操作符 …… 77
3.5.2 逻辑布尔操作符 …… 77
3.5.3 逻辑求反操作符 …… 78
3.5.4 条件操作符 …… 79
3.5.5 空接合操作符 …… 80
3.6 按位操作符 …… 80
3.6.1 移位操作符 …… 81
3.6.2 按位操作符 …… 82
3.6.3 按位赋值操作符 …… 83
3.6.4 按位取反操作符 …… 84
3.7 控制流语句（续） …… 84
3.7.1 while和do/while循环 … 84
3.7.2 for循环 …… 86
3.7.3 foreach循环 …… 88
3.7.4 switch语句 …… 90
3.8 跳转语句 …… 92
3.8.1 break语句 …… 92
3.8.2 continue语句 …… 94
3.8.3 goto语句 …… 95
3.9 C#预处理指令 …… 97
3.9.1 排除和包含代码 …… 98
3.9.2 定义预处理符号 …… 98
3.9.3 生成错误和警告 …… 99
3.9.4 关闭警告消息 …… 99
3.9.5 nowarn:<warn list>选项 … 99
3.9.6 指定行号 …… 100
3.9.7 可视编辑器提示 …… 100
3.10 小结 …… 101
第4章 方法和参数 …… 103
4.1 方法的调用 …… 104
4.1.1 命名空间 …… 105
4.1.2 类型名称 …… 106
4.1.3 作用域 …… 107
4.1.4 方法名称 …… 107
4.1.5 形参和实参 …… 107
4.1.6 方法返回值 …… 107
4.1.7 语句与方法调用的比较 … 108
4.2 方法的声明 …… 108
4.2.1 形式参数声明 …… 109
4.2.2 方法返回类型声明 …… 110
4.3 using指令 …… 111
4.4 Main()的返回值和参数 …… 114
4.5 方法的参数 …… 116
4.5.1 值参数 …… 116
4.5.2 引用参数（ref） …… 117
4.5.3 输出参数（out） …… 118
4.5.4 参数数组 …… 120
4.6 递归 …… 122
4.7 方法重载 …… 124
4.8 可选参数 …… 126
4.9 用异常实现基本错误处理 …… 129
4.9.1 捕捉错误 …… 130
4.9.2 使用throw语句报告错误 … 136
4.10 小结 …… 138
第5章 类 …… 139
5.1 类的定义和实例化 …… 141
5.2 实例字段 …… 144
5.2.1 实例字段的声明 …… 144
5.2.2 实例字段的访问 …… 144
5.3 实例方法 …… 145
5.4 使用this关键字 …… 146
5.5 访问修饰符 …… 151
5.6 属性 …… 153
5.6.1 属性的声明 …… 154
5.6.2 自动实现的属性 …… 155

5.6.3 属性和字段的设计编码规范 …… 157
5.6.4 提供属性验证 …… 157
5.6.5 只读和只写属性 …… 159
5.6.6 为取值方法和赋值方法指定访问修饰符 …… 160
5.6.7 属性作为虚字段使用 …… 161
5.6.8 属性和方法调用不允许作为ref或out参数值使用 …… 163
5.7 构造器 …… 164
5.7.1 构造器的声明 …… 164
5.7.2 默认构造器 …… 165
5.7.3 对象初始化器 …… 166
5.7.4 构造器的重载 …… 167
5.7.5 构造器链：使用this调用另一个构造器 …… 168
5.8 静态成员 …… 172
5.8.1 静态字段 …… 172
5.8.2 静态方法 …… 174
5.8.3 静态构造器 …… 175
5.8.4 静态属性 …… 176
5.8.5 静态类 …… 177
5.9 扩展方法 …… 178
5.10 封装数据 …… 179
5.10.1 const …… 179
5.10.2 readonly …… 180
5.11 嵌套类 …… 181
5.12 分部类 …… 183
5.12.1 定义分部类 …… 183
5.12.2 分部方法 …… 184
5.13 小结 …… 186

第6章 继承 …… 187

6.1 派生 …… 188
6.1.1 基类型和派生类型之间的转型 …… 190
6.1.2 private访问修饰符 …… 191
6.1.3 protected访问修饰符 …… 192
6.1.4 扩展方法 …… 193
6.1.5 单继承 …… 194
6.1.6 密封类 …… 195
6.2 基类的重写 …… 196
6.2.1 virtual修饰符 …… 196
6.2.2 new修饰符 …… 199
6.2.3 sealed修饰符 …… 202
6.2.4 base成员 …… 203
6.2.5 构造器 …… 204
6.3 抽象类 …… 204
6.4 所有类都从System.Object派生 … 208
6.5 使用is操作符验证基础类型 …… 209
6.6 使用as操作符进行转换 …… 210
6.7 小结 …… 211

第7章 接 口 …… 213

7.1 接口概述 …… 213
7.2 通过接口实现多态性 …… 215
7.3 接口实现 …… 218
7.3.1 显式成员实现 …… 220
7.3.2 隐式成员实现 …… 221
7.3.3 显式接口实现与隐式接口实现的比较 …… 221
7.4 在实现类和接口之间转换 …… 222
7.5 接口继承 …… 222
7.6 多接口继承 …… 224
7.7 接口上的扩展方法 …… 225
7.8 通过接口实现多继承 …… 226
7.9 版本控制 …… 228
7.10 接口与类的比较 …… 229
7.11 接口与特性的比较 …… 230
7.12 小结 …… 230

第8章 值类型 …… 231

8.1 结构 …… 231
8.1.1 结构的初始化 …… 235
8.1.2 default操作符的使用 … 237
8.1.3 值类型的继承和接口 …… 237
8.2 装箱 …… 238
8.3 枚举 …… 243
8.3.1 枚举之间的类型兼容性 … 246
8.3.2 枚举和字符串之间的转换 …… 246
8.3.3 枚举作为标志使用 …… 248
8.4 小结 …… 251

第9章 良构类型 …… 253

9.1 重写object的成员 …… 253
9.1.1 重写ToString() …… 253
9.1.2 重写GetHashCode() …… 254
9.1.3 重写Equals() …… 256
9.2 操作符重载 …… 263
9.2.1 比较操作符 …… 264
9.2.2 二元操作符 …… 265
9.2.3 赋值与二元操作符的结合 …… 266
9.2.4 条件逻辑操作符 …… 266
9.2.5 一元操作符 …… 267
9.2.6 转换操作符 …… 268
9.2.7 转换操作符的规范 …… 269
9.3 引用其他程序集 …… 269
9.3.1 更改程序集目标 …… 270
9.3.2 引用程序集 …… 270
9.3.3 类型封装 …… 271
9.4 定义命名空间 …… 272
9.5 XML注释 …… 275
9.5.1 将XML注释与代码构造关联到一起 …… 275
9.5.2 生成XML文档文件 …… 277
9.6 垃圾回收 …… 278
9.7 资源清理 …… 280
9.7.1 终结器 …… 280
9.7.2 使用using语句进行确定性终结 …… 282
9.7.3 垃圾回收、终结和IDisposable …… 284
9.8 推迟初始化 …… 286
9.9 小结 …… 288

第10章 异常处理 …… 289

10.1 多异常类型 …… 289
10.2 捕捉异常 …… 291
10.3 常规catch块 …… 293
10.4 异常处理的规范 …… 295
10.5 定义自定义异常 …… 297
10.6 封装异常并重新引发 …… 299
10.7 小结 …… 302

第11章 泛 型 …… 303

11.1 如果C# 没有泛型 …… 303
11.2 泛型类型概述 …… 307
11.2.1 泛型类的使用 …… 308
11.2.2 简单泛型类的定义 …… 309
11.2.3 泛型的优点 …… 310
11.2.4 类型参数命名规范 …… 310
11.2.5 泛型接口和结构 …… 311
11.2.6 构造器和终结器的定义 313
11.2.7 默认值的指定 …… 313
11.2.8 多个类型参数 …… 314
11.2.9 元数 …… 315
11.2.10 嵌套泛型类型 …… 316
11.3 约束 …… 317
11.3.1 接口约束 …… 319
11.3.2 类类型约束 …… 320
11.3.3 struct/class约束 …… 321
11.3.4 多个约束 …… 321
11.3.5 构造器约束 …… 322
11.3.6 约束继承 …… 323
11.4 泛型方法 …… 327
11.4.1 泛型方法类型推断 …… 328
11.4.2 约束的指定 …… 328
11.5 协变性和逆变性 …… 330
11.5.1 在C# 4.0中使用out类型参数修饰符允许协变性 …… 331
11.5.2 在C# 4.0中使用in类型参数修饰符允许逆变性 …… 332
11.5.3 数组对不安全协变性的支持 …… 335
11.6 泛型的内部机制 …… 335
11.6.1 基于值类型的泛型的实例化 …… 336
11.6.2 基于引用类型的泛型的实例化 …… 337
11.7 小结 …… 338

第12章 委托和Lambda表达式 … 339

12.1 委托概述 …… 340
12.1.1 背景 …… 340
12.1.2 委托数据类型 …… 341

12.1.3 委托类型的声明 …… 342
12.1.4 委托的实例化 …… 343
12.2 Lambda表达式 …… 347
12.2.1 语句Lambda …… 347
12.2.2 表达式Lambda …… 349
12.3 匿名方法 …… 351
12.4 通用的委托：System.Func和System.Action …… 352
12.4.1 委托没有结构相等性 … 353
12.4.2 外部变量 …… 355
12.4.3 表达式树 …… 359
12.5 小结 …… 364

第13章 事 件 …… 365

13.1 使用多播委托来编码Observer模式 …… 366
13.1.1 定义订阅者方法 …… 366
13.1.2 定义发布者 …… 367
13.1.3 连接发布者和订阅者 … 368
13.1.4 调用委托 …… 368
13.1.5 检查null值 …… 369
13.1.6 委托操作符 …… 370
13.1.7 顺序调用 …… 371
13.1.8 错误处理 …… 374
13.1.9 方法返回值和传引用 … 376
13.2 事件 …… 376
13.2.1 事件的作用 …… 376
13.2.2 事件的声明 …… 378
13.2.3 编码规范 …… 379
13.2.4 泛型和委托 …… 380
13.2.5 自定义事件的实现 …… 383
13.3 小结 …… 384

第14章 支持标准查询操作符的集合接口 …… 385

14.1 匿名类型和隐式类型的局部变量声明 …… 386
14.1.1 匿名类型 …… 386
14.1.2 隐式类型的局部变量 … 387
14.1.3 匿名类型和隐式局部变量的更多注意事项 …… 388
14.2 集合初始化器 …… 390
14.3 是什么使类成为集合：IEnumerable<T> …… 392
14.3.1 foreach和数组 …… 392
14.3.2 foreach和IEnumerable<T> …… 393
14.3.3 foreach循环内不要修改集合 …… 396
14.4 标准查询操作符 …… 397
14.4.1 使用Where()来筛选 … 400
14.4.2 使用Select()来投射 … 401
14.4.3 使用Count()对元素进行计数 …… 403
14.4.4 推迟执行 …… 404
14.4.5 使用OrderBy()和ThenBy()来排序 …… 408
14.4.6 使用Join()来执行内部联接 …… 412
14.4.7 使用GroupBy分组结果 … 415
14.4.8 使用GroupJoin()实现一对多关系 …… 416
14.4.9 调用Selectmany() …… 418
14.4.10 更多标准查询操作符 … 420
14.5 小结 …… 423

第15章 使用查询表达式的LINQ …… 425

15.1 查询表达式概述 …… 426
15.1.1 投射 …… 427
15.1.2 筛选 …… 432
15.1.3 排序 …… 433
15.1.4 let子句 …… 434
15.1.5 分组 …… 435
15.1.6 使用into进行查询延续 …… 437
15.1.7 用多个from子句“平整”序列的序列 …… 438
15.2 查询表达式作为方法调用 …… 439
15.3 小结 …… 440

第16章 构建自定义集合 …… 441

16.1 更多集合接口 …… 442
16.1.1 IList<T>与IDictionary<TKey, TValue> …… 442

16.1.2 ICollection<T> …… 443
16.2 主要集合类 …… 443
16.2.1 列表集合：List<T> …… 443
16.2.2 全序 …… 446
16.2.3 搜索List<T> …… 447
16.2.4 字典集合：Dictionary<TKey, TValue> …… 449
16.2.5 已排序集合：SortedDictionary<TKey,TValue>和SortedList<T> …… 453
16.2.6 栈集合：Stack<T> …… 454
16.2.7 队列集合：Queue<T> … 455
16.2.8 链表：LinkedList<T> … 456
16.3 提供索引器 …… 456
16.4 返回null或者空集合 …… 459
16.5 迭代器 …… 459
16.5.1 迭代器的定义 …… 460
16.5.2 迭代器语法 …… 460
16.5.3 从迭代器生成值 …… 461
16.5.4 迭代器和状态 …… 463
16.5.5 更多的迭代器例子 …… 464
16.5.6 将yield return语句放到循环中 …… 465
16.5.7 取消更多的迭代：yield break …… 467
16.5.8 在单个类中创建多个迭代器 …… 469
16.5.9 yield语句的要求 …… 470
16.6 小结 …… 470

第17章 反射、特性和动态编程 … 471

17.1 反射 …… 471
17.1.1 使用System.Type访问元数据 …… 472
17.1.2 成员调用 …… 473
17.1.3 泛型类型上的反射 …… 478
17.2 特性 …… 479
17.2.1 自定义特性 …… 482
17.2.2 查找特性 …… 483
17.2.3 使用构造器来初始化特性 …… 484
17.2.4 System.AttributeUsage-Attribute …… 488
17.2.5 命名参数 …… 488
17.3 使用动态对象进行编程 …… 498
17.3.1 使用dynamic调用反射 … 499
17.3.2 dynamic的原则和行为 … 500
17.3.3 为什么需要动态绑定 … 501
17.3.4 静态编译与动态编程的比较 …… 502
17.3.5 实现自定义动态对象 … 503
17.4 小结 …… 505

第18章 多线程处理 …… 507

18.1 多线程基础 …… 509
18.2 使用System.Threading …… 512
18.2.1 使用System.Threading.Thread进行异步操作 … 512
18.2.2 线程管理 …… 514
18.2.3 在生产代码中不要让线程进入睡眠 …… 514
18.2.4 在生产代码中不要中止线程 …… 515
18.2.5 线程池处理 …… 516
18.3 异步任务 …… 517
18.3.1 从Thread到Task …… 518
18.3.2 理解异步任务 …… 518
18.3.3 任务延续 …… 521
18.3.4 用AggregateException处理Task上的未处理异常 … 525
18.4 取消任务 …… 530
18.4.1 Task.Run()是Task.Factory.StartNew()的简化形式 532
18.4.2 长时间运行的任务 …… 532
18.4.3 对任务进行资源清理 … 533
18.5 C# 5.0基于任务的异步模式 …… 533
18.5.1 以同步方式调用高延迟操作 …… 534
18.5.2 使用TPL异步调用高延迟操作 …… 535
18.5.3 通过async和await实现基于任务的异步模式 …… 538
18.5.4 异步Lambda …… 541
18.5.5 任务调度器和同步上下文

…… 546
18.5.6 async/await和Windows UI …… 547
18.5.7 await操作符 …… 548
18.6 并行迭代 …… 549
18.7 并行执行LINQ查询 …… 556
18.8 小结 …… 560

第19章 线程同步 …… 561

19.1 为什么要同步 …… 562
19.1.1 使用Monitor来同步 …… 565
19.1.2 使用lock关键字 …… 567
19.1.3 lock对象的选择 …… 568
19.1.4 为什么要避免锁定this、typeof(type)和string …… 569
19.1.5 将字段声明为volatile 570
19.1.6 使用System.Threading.Interlocked类 …… 570
19.1.7 多个线程时的事件通知 571
19.1.8 同步设计最佳实践 …… 572
19.1.9 更多的同步类型 …… 573
19.1.10 线程本地存储 …… 580
19.2 计时器 …… 583
19.3 小结 …… 584

第20章 平台互操作性和不安全的代码 …… 585

20.1 在C#中使用WinRT库 …… 586
20.1.1 具有自定义Add/Remove处理程序的WinRT事件 586
20.1.2 自动变换的接口 …… 587
20.1.3 基于任务的异步 …… 587
20.2 平台调用 …… 588
20.2.1 外部函数的声明 …… 588
20.2.2 参数的数据类型 …… 588
20.2.3 使用ref而不是指针 …… 590
20.2.4 为顺序布局使用StructLayoutAttribute… 590
20.2.5 错误处理 …… 591
20.2.6 使用SafeHandle …… 593
20.2.7 外部函数的调用 …… 595
20.2.8 用包装器简化API调用 … 597
20.2.9 函数指针映射到委托 … 598
20.2.10 编码规范 …… 598
20.3 指针和地址 …… 598
20.3.1 不安全的代码 …… 598
20.3.2 指针的声明 …… 599
20.3.3 指针的赋值 …… 600
20.3.4 指针的解引用 …… 602
20.3.5 访问被引用物类型的成员 …… 604
20.3.6 通过委托执行不安全的代码 …… 604
20.4 小结 …… 606

第21章 CLI …… 607

21.1 CLI的定义 …… 607
21.2 CLI实现 …… 608
21.3 C#编译成机器码 …… 609
21.4 运行时 …… 611
21.4.1 垃圾回收 …… 611
21.4.2 .NET的垃圾回收 …… 611
21.4.3 类型安全 …… 612
21.4.4 代码访问安全性 …… 612
21.4.5 平台可移植性 …… 612
21.4.6 性能 …… 613
21.5 应用程序域 …… 614
21.6 程序集、清单和模块 …… 614
21.7 公共中间语言 …… 616
21.8 公共类型系统 …… 616
21.9 公共语言规范 …… 617
21.10 基类库 …… 617
21.11 元数据 …… 617
21.12 小结 …… 618

附录A 下载和安装C#编译器与CLI平台 …… 619

附录B 井字棋源代码清单 …… 621

附录C TPL和C# 5.0之前的多线程模式 …… 627

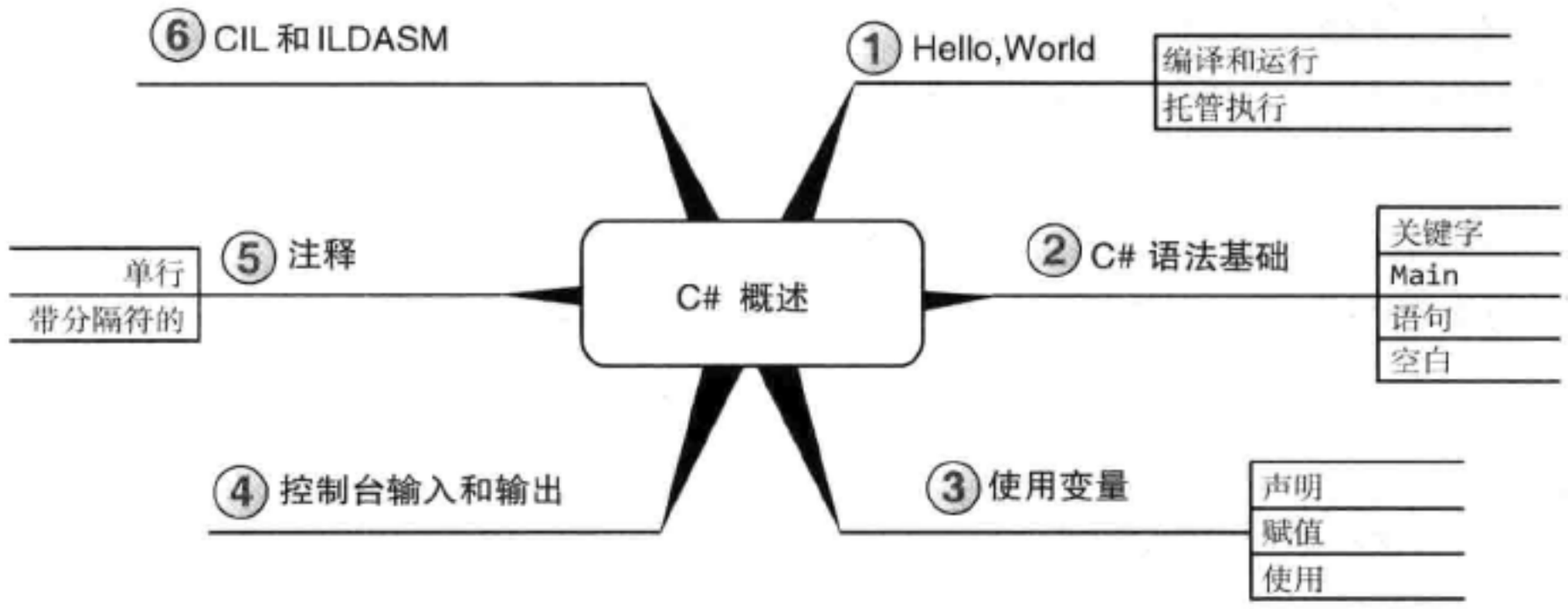
附录D C# 5.0 async/await模式之前的计时器 …… 649

第1章

C#概述

C#是一种成熟的语言，它基于C风格语言（C、C++和Java）的特性而设计，有经验的程序员很快就能熟悉它。[①]作为构建软件组件和应用程序的编程语言，C#是更大、更复杂的执行平台——公共语言基础结构（Common Language Infrastructure，CLI）——的一部分。

本章使用传统HelloWorld程序介绍C#，重点是C#语法基础，包括在C#程序的可执行文件中定义入口。通过本章的学习，将熟悉C#的语法风格和结构，并能写最简单的C#程序。讨论C#语法基础之前，将简单介绍托管执行环境，并解释C#程序在运行时如何执行。本章最后会讨论如何声明变量、如何在控制台上写入和获取数据以及如何为C#代码添加注释。



1.1 Hello, World

学习新语言最好的办法就是动手写代码。第一个例子是经典HelloWorld程序。这个程序在屏幕上显示一些文本。

① 第一次C#设计会议在1998年举行。

代码清单1-1展示了完整的HelloWorld程序，在之后的小节中还将介绍代码的编译方法。

代码清单1-1 用C#编写的HelloWorld[①]

```
class HelloWorld
{
 static void Main()
 {
   System.Console.WriteLine("Hello. My name is Inigo Montoya.");
 }
}
```

> **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。

有Java、C或者C++编程经验的读者很快就能看出相似的地方。类似于Java，C#也从C和C++继承了基本的语法[②]。语法标点（如分号和大括号）、特性（如区分大小写）和关键字（如class、public和void）对于这些程序员来说并不陌生。初学者和有其他语言背景的程序员通过这个程序能很快体会到这些构造的直观性。

编译并运行应用程序

C#编译器允许为C#源代码文件使用任何文件扩展名，但一般使用.cs。将源代码保存到文件后，开发者必须编译它（附录A会指导你安装编译器）。由于C#标准没有规定命令机制，所以实际的编译命令取决于C#编译器的实现。

假定将代码清单1-1的代码放到一个名为HelloWorld.cs的文件中，而且使用的是Microsoft.NET编译器，编译器的路径也已设置好，就可以执行如输出1-1所示的编译命令。[③]

输出1-1

```
>csc.exe HelloWorld.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17626
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

具体输出将取决于所用的编译器的版本。

运行生成的程序HelloWorld.exe，将显示如输出1-2所示的消息。

输出1-2

```
>HelloWorld.exe
Hello. My name is Inigo Montoya.
```

C#编译器创建的程序HelloWorld.exe是一个程序集（assembly）。虽然在这个例子

---

① 如果不知道Inigo Montoya是谁，请找《公主新娘》（The Princess Bride）这部电影来看一看。

② 设计C#时，语言的创建者从C/C++规范中删除了他们不喜欢的一些特性，同时创建了他们喜欢的一些特性。开发组还有其他语言的资深专家。

③ 使用Mono编译器（开源编译器，网址是www.mono-project.com）进行编译，命令几乎一样，只是编译器名称是mcs.exe，而不是csc.exe。虽然我个人很乐意给出在每种平台下使用的命令，但这样会使读者分心。请参见附录A了解Mono的详情。

中创建的是一个能独立运行的程序，但也可以不这么做，而是创建能由另一个较大的程序引用的代码库。库（或称为类库）的文件扩展名是.dll，其中dll代表“动态链接库”（Dynamic Link Library，DLL）。库也是一个程序集。换言之，一次成功的C#编译生成的肯定是程序集，无论它是程序还是库。

语言对比：Java——文件名必须和类名匹配

在Java中，文件名必须和类名一致。C#虽然也常常遵守这一约定，但却不是必需的。在C#中，一个文件可以包含多个类；而且从C# 2.0开始，一个类的代码可以拆分到多个文件中。

1.2 C#语法基础

成功编译并运行HelloWorld程序之后，我们来分析代码，了解它的各个组成部分。首先熟悉一下C#关键字以及开发者选用的标识符。

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，我们将其称为**关键字**。关键字提供了具体的语法，编译器根据这些语法来解释程序员书写的表达式。在HelloWorld程序中，class、static和void均是关键字。

编译器利用关键字来识别代码的结构与组织方式。因为编译器对这些单词有着严格的解释，所以开发人员只能按照C#的语言规则将关键字放在特定的位置。一旦程序员违反规则，编译器就会报错。

1.2.1 C#关键字

表1-1总结了C#关键字。

表1-1 C#关键字

abstract	else	let* (3)	sizeof
add*	enum	lock	stackalloc
alias* (2)	equals* (3)	long	static
as	event	namespace	string
ascending* (3)	explicit	new	struct
async* (5)	extern	null	switch
await* (5)	false	object	this
base	finally	on* (3)	throw
bool	fixed	operator	true
break	float	orderby* (3)	try
by* (3)	for	out	typeof
byte	foreach	override	uint
case	from* (3)	params	ulong

续表

catch	get* (1)	partial* (2)	unchecked
char	global* (2)	private	unsafe
checked	goto	protected	ushort
class	group* (3)	public	using
const	if	readonly	value* (1)
continue	implicit	ref	var* (3)
decimal	in	remove*	virtual
default	int	return	void
delegate	interface	sbyte	volatile
descending* (3)	internal	sealed	where* (3)
do	into* (3)	select* (3)	while
double	is	set* (1)	yield* (2)
dynamic* (4)	join* (3)	short	

*上下文关键字，括号中的数字代表加入该上下文关键字的版本。

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**（contextual keyword），它们在特定位置才有意义。除了那些位置，上下文关键字没有任何特殊意义。[①]这样所有C# 1.0代码都完全兼容于后续的版本。[②]

1.2.2 标识符

和其他语言一样，C#用**标识符**标识程序员编写代码的构造。在代码清单1-1中，HelloWorld和Main都是标识符。分配标识符之后，以后就能用它引用所标识的构造。因此，开发人员应分配有意义的名称，不要随意分配。

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.NET Framework Guidelines建议不要在标识符中使用单词缩写[③]，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，或者开发别人要使用的库的时候。

标识符有两种基本的大小写风格。第一种风格是CLI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），即除了第一个字母小写，其他约定与Pascal大小写风格一样，例如quotient、firstName、httpFileCollection、ioStream和theDreadPirateRoberts。

① 例如，在C# 2.0设计之初，语言的设计者们将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。然而，语言的设计者最终选择使用yield return而不是yield，从而避免将yield作为新关键字。除非与return连用，否则yield没有任何特殊意义。

② 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。还有一些极少见的泛型表达式，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。

③ 有两种单词缩写，一种是“Abbreviation”，比如Professor缩写为Prof.；另一种是“Contraction”，如Doctor缩写为Dr。——译者注

规范

要更注重标识符的清晰而不是简短。

不要在标识符名称中使用单词缩写。

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，但标识符中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。

在极少数情况下，有的标识符（如Main）可能在C#语言中具有特殊含义。

规范

要把只包含两个字母的首字母缩写词全部大写，除非它是camel大小写风格标识符的第一个单词。

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camel大小写风格标识符的第一个单词。

在camel大小写风格标识符开头的首字母缩写词中，所有字母都**不要**大写。

不要使用匈牙利命名法（也就是，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“@”前缀可作为标识符使用，例如可命名局部变量@return。类似地（虽不符合C#大小写规范），可命名方法@throw()。

在微软的实现中，还有4个未文档化的保留关键字，即__arglist、__makeref、__reftype和__refvalue。它们仅在罕见的互操作情形下才会用到，平时完全可以忽略。注意这4个特殊关键字以双下划线开头。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，开发人员自己不要创建这样的标识符。

1.2.3 类型定义

C#中所有代码都出现在一个类型定义的内部，最常见的类型定义是以关键字class开头的。如代码清单1-2所示，**类定义**（class definition）是class <标识符>{ ... }形式的一个代码块。

代码清单1-2 基本的类声明

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

类型的名称（本例是HelloWorld）可以随便取，但根据约定，它应当使用Pascal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。（对于包含Main()方法的类，Program是个很好的名称。Main()方法将在稍后详述。）

规范

要用名词或名词短语命名类。

要为所有类名使用Pascal大小写风格。

一个程序通常包含多个类型，每个类型都包含多个方法。

1.2.4 Main

初学者主题：什么是方法？

从语法上说，C#程序中的**方法**是一个已命名的代码块，该代码块由一个方法声明（如static void Main()）引入，后跟一对大括号（{}），其中包含零或多条语句。方法可以执行计算和/或操作。与书写语言中的段落相似，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，可以在多个地方调用，所以避免了代码的重复。方法声明除了负责引入方法之外，还要定义方法名以及要传入和传出方法的数据。在代码清单1-3中，Main()后跟{ ... }便是一个C#方法的例子。

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HelloWorld.exe执行程序，程序会启动并解析Main的位置，然后执行其中第一条语句，如代码清单1-3所示。

代码清单1-3 HelloWorld分解示意图

```
class HelloWorld
{
  static void Main()      } 方法声明
  {
    System.Console.WriteLine("Hello, My name is Inigo Montoya");
  }                                                              } Main } 类定义
                                  语句
}
```

虽然Main方法声明可以进行某种程度的改变，但关键字static和方法名Main是始终都是程序必需的。

高级主题：Main方法的声明

C#要求Main方法的返回类型为void或int，而且要么不带参数，要么接收一个字符串数组作为参数。代码清单1-4展示了Main方法的完整声明。

代码清单1-4 带有参数和返回类型的Main方法

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

args参数是一个字符串数组，用于接收命令行参数。但数组第一个元素不是程序名

称，而是可执行文件名称[①]之后的第一个命令行参数，这一点与C和C++不同。要获取执行程序所用的完整命令，可以使用System.Environment.CommandLine。

Main返回的int值是状态码，标识程序执行是否成功。返回非零值通常意味着错误。

语言对比：C++/Java—— main()是全部小写的

与C风格的“前辈们”不同，C#的Main方法名使用大写M，以便与C#的Pascal大小写风格命名约定保持一致。

将Main方法指定为static意味着这是“静态”方法，可用“*类名.方法名*”的形式调用它。如果不指定static，用于启动程序的命令控制台还要先对类进行**实例化**（instantiation），然后才能调用方法。第5章会用一节的篇幅专门讲述静态成员。

Main()之前的void表明该方法不返回任何数据（第2章会进一步解释）。

C#和C/C++一样使用大括号封闭构造（如类或者方法）的主体。例如，Main方法的主体就是用大括号封闭起来的。在本例中，方法的主体只有一条语句。

1.2.5 语句和语句分隔符

Main方法只包含一条**语句**，即System.Console.WriteLine();，它在控制台上输出一行文本。C#通常用分号标识**语句**结束，每条语句都由代码要执行的一个或多个行动构成。声明变量、控制程序流程或者调用方法，所有这些都是语句的例子。

语言对比：Visual Basic——基于行的语句

有的语言以行为基本单位，这意味着不加上特殊标记，语句便不能跨行。在Visual Basic 2010以前，Visual Basic一直是典型的基于行的语言。它要求在行末添加下划线表示语句跨越多行。从Visual Basic 2010开始，行连续符在许多时候都变成可选的。

高级主题：没有分号的语句

C#的许多编程元素都以分号结尾。不要求使用分号的例子是switch语句。由于大括号总是包含在switch语句中，所以C#不要求语句后加上分号。事实上，代码块本身就被视为语句（它们也由语句构成），不要求以分号结尾。类似地，有的编程元素（如using指令）虽然末尾有分号但不被视为语句。

由于换行与否不影响语句的分隔，所以可以将多条语句放到同一行，C#编译器会认为这一行包含多条指令。例如，代码清单1-5在同一行包含了两条语句。执行时，它们会在控制台窗口中分两行显示Up和Down。

代码清单1-5 一行中包含多条语句

```
System.Console.WriteLine("Up");System.Console.WriteLine("Down");
```

① 也就是程序名称，比如HelloWorld.exe。——译者注

C#还允许一条语句跨越多行。同样地，C#编译器会根据分号判断语句的结束位置。代码清单1-6展示了一个例子。

代码清单1-6 一条语句跨越多行

```
System.Console.WriteLine(
  "Hello. My name is Inigo Montoya.");
```

代码清单1-6的WriteLine()语句的原始版本来自HelloWorld程序，它在这里跨越了多行。

1.2.6 空白

分号使C#编译器能忽略代码中的空白。除了少许例外情况，C#允许在代码中随意插入空白而不改变其语义。在代码清单1-5和代码清单1-6中，在语句中或语句间换行都可以，对编译器最终创建的可执行文件没有任何影响。

初学者主题：什么是空白？

空白是一个或多个连续的格式字符（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。

程序员经常利用空白对代码进行缩进来增强可读性。来看看代码清单1-7和代码清单1-8展示的两个版本的HelloWorld程序。

代码清单1-7 不缩进

```
class HelloWorld
{
static void Main()
{
System.Console.WriteLine("Hello Inigo Montoya");
}
}
```

代码清单1-8 删除一切可以删除的空白

```
class HelloWorld{static void Main()
{System.Console.WriteLine("Hello Inigo Montoya");}}
```

虽然这两个版本看起来和原始版本颇有不同，但C#编译器认为这几个版本的代码没有任何区别。

初学者主题：用空白来格式化代码

为了增强可读性，利用空白对代码进行缩进是非常重要的。写代码时要遵循已经建立的编码标准和约定，以增强代码的可读性。

本书约定每个大括号都单独占一行，并缩进大括号之间的代码。假如一对大括号之间有第二对大括号，那么第二对大括号中的所有代码也要缩进。

这不是统一的C#标准，只是一种风格偏好。

1.2.7 使用变量

前面已接触了最基本的C#程序，下面让我们来声明一个局部变量。变量声明后就可以

被赋值，将值替换成新值，并可在计算和输出等操作中使用。然而，变量一经声明，数据类型就无法改变。在代码清单1-9中，string max就是一个变量声明。

代码清单1-9 变量的声明和赋值

```
class MiracleMax
{
  static void Main()
  {
        数据类型
      string max;
             变量
    max = "Have fun storming the castle!";

    System.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，程序以后可以对这个存储位置进行赋值和修改操作。局部意味着是在方法内部声明变量。

声明变量就是定义它，需要：

a. 指定变量要包含的数据的类型；

b. 为它分配标识符（变量名）。

1.2.8 数据类型

代码清单1-9声明了string类型的变量。本章还使用了int和char。

- int是指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数据类型。

初学者主题：什么是数据类型？

一个变量声明所指定的数据的类型称为**数据类型**。数据类型，或者简称为**类型**，是具有相似特征和行为的个体的分类。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性的一些个体的定义。

1.2.9 变量的声明

在代码清单1-9中，string max是一个变量声明，它声明了一个名为max的string类型的变量。还可以在同一条语句中声明多个变量，办法是指定数据类型一次，然后用逗号分隔每个标识符，如代码清单1-10所示。

---

① 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），第二个代码值称为“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。——译者注

代码清单1-10 在一条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开发者提供一次数据类型，因此所有变量都具有相同类型。

在C#中，变量名可以用任何字母或者下划线（_）开头，后跟任意数量的字母、数字和/或下划线。但根据约定，局部变量名采用的是camel大小写风格命名（即除了第一个单词，其他每个单词的首字母大写），而且不包含下划线。

> **规范**
>
> 要为局部变量使用camel大小写风格的命名。

1.2.10 变量的赋值

局部变量声明后必须在引用之前为其赋值。一个办法是使用=操作符，或者称为简单赋值操作符。操作符是一种特殊符号，标识了代码要执行的操作。代码清单1-11演示了如何利用赋值操作符指定变量max[①]和valerie要指向的字符串值。

代码清单1-11 更改变量的值

```
class MiracleMax
{
  static void Main()
  {
      string valerie;
      string max = "Have fun storming the castle!";

      valerie = "Think it will work?";

      System.Console.WriteLine(max);
      System.Console.WriteLine(valerie);

      max = "It would take a miracle.";
      System.Console.WriteLine(max);
  }
}
```

从这个代码清单可以看出，既可以在声明变量的同时对它赋值（如变量max），也可以在声明了变量之后用另一条语句赋值（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MiracleMax.exe程序，会生成如输出1-3所示的结果。

输出1-3

```
>MiracleMax.exe
Have fun storming the castle!
Think it will work?
It would take a miracle.
```

① 这里的max不是数学函数，而是变量名。

C#要求局部变量在读取之前“明确赋值”。此外，一次赋值会返回一个值。所以，C#允许在同一条语句中进行多个赋值操作，如代码清单1-12所示。

代码清单1-12 赋值会返回一个值，该值可用于再次赋值

```
class MiracleMax
{
  static void Main()
  {
    // ...
    string requirements, max;
    requirements = max = "It would take a miracle.";
    // ...
  }
}
```

1.2.11 变量的使用

赋值后就能用变量标识符引用值。因此，在System.Console.WriteLine(max)语句中使用变量max时，程序在控制台上显示Have fun storming the castle!，也就是max的值。更改max的值并执行相同的System.Console.WriteLine(max)语句，会显示max的新值，即It would take a miracle.。

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量（literal）[①]，都是不可变的（或者说是不可修改的）。例如，不可能将字符串"Come As You Are"更改为"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。

1.3 控制台输入和输出

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

1.3.1 从控制台获取输入

获取用户在控制台输入文本的一种方法是使用System.Console.ReadLine()。此方法将暂停程序执行以便用户输入字符。一旦用户按回车键创建新的一行，程序就会继续执行。System.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。下面来看一下代码清单1-13以及输出1-4。

代码清单1-13 使用System.Console.ReadLine()

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

① 即literal，是指以文本形式嵌入的数据。literal有多种译法，没有一种占绝对优势。最典型的译法是“字面量”、“文字常量”和“直接量”。本书采用前者。——译者注

```
        System.Console.WriteLine("Hey you!");

        System.Console.Write("Enter your first name: ");
        firstName = System.Console.ReadLine();

        System.Console.Write("Enter your last name: ");
        lastName = System.Console.ReadLine();

        ...
    }
}
```

输出1-4

```
>HeyYou.exe
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
```

在每条提示信息之后，程序都用System.Console.ReadLine()方法获取用户输入的文本，并将其赋给恰当的变量。在第二个System.Console.ReadLine()赋值操作完成之后，firstName引用值Inigo，而lastName引用值Montoya。

高级主题：System.Console.Read()

除了System.Console.ReadLine()还有System.Console.Read()方法。但System.Console.Read()方法返回的是与读取的字符值对应的整数，如果没有更多的字符可用，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1-14所示。

代码清单1-14 使用System.Console.Read()

```
int readValue;
char character;
readValue = System.Console.Read();
character = (char) readValue;
System.Console.Write(character);
```

注意，除非用户按回车键，否则System.Console.Read()方法不会返回输入。按回车键之前不会对字符进行处理，即使用户已经输入了多个字符。

C# 2.0新增了System.Console.ReadKey()方法。它和System.Console.Read()方法不同，返回的是用户所按键输入。可用它拦截用户按键操作，并执行相应行动，比如校验按键，限制只能按数字键。

1.3.2 将输出写入控制台

代码清单1-13中是用System.Console.Write()而不是System.Console. WriteLine()方法提示用户输入他的名和姓。System.Console.Write()方法不在显示文本之后添加一个换行符，而是将当前位置保持在同一行上。这样用户输入的内容就会与提示内容处于同一行。代码清单1-13的输出清楚地演示了System.Console.Write()的效果。

下一步是将使用System.Console.ReadLine()获取的值写回控制台。在代码清单1-15所示的例子中，程序将在控制台上输出用户的全名。但这段代码使用的是System.Console.WriteLine()的一个变体。输出1-5展示了对应的输出。

代码清单1-15 使用System.Console.WriteLine()进行格式化

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

      System.Console.WriteLine("Hey you!");

      System.Console.Write("Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write("Enter your last name: ");
      lastName = System.Console.ReadLine();

      System.Console.WriteLine(
        "Your full name is {0} {1}.", firstName, lastName);
  }
}
```

输出1-5

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Inigo Montoya.
```

在代码清单1-15所示的例子中，不是先用Write语句输出your full name is，再用Write语句输出firstName，然后用第三条Write语句输出空格，最后用WriteLine语句输出lastName。相反，这个例子利用**复合格式化**（composite formatting）来进行一次性输出。在复合格式化中，代码首先提供**格式字符串**（format string）来定义输出格式。本例的格式字符串是"Your full name is {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。

注意索引值是从零开始的。每个要插入的参数，或者称为**格式项**（format item），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参数，所以它对应于索引值0。类似地，lastName对应于索引值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1-16中的代码交换了两个索引占位符的位置并添加了一个逗号，从而改变了姓名的显示方式（参见输出1-6）。

代码清单1-16 交换索引占位符和对应的变量

```
System.Console.WriteLine("Your full name is {1}, {0}",
  firstName, lastName);
```

输出1-6

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Montoya, Inigo
```

占位符除了能在格式字符串中按任意顺序出现之外，同一个占位符还能在一个格式字符串中多次使用。另外，也可以忽略占位符。然而，每个占位符都必须对应一个参数，不能使用没有对应参数的占位符。

1.3.3 注释

本节将修改代码清单1-15中的程序，在其中添加注释。提供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1-17中展示了新的代码，输出1-7展示了对应的输出。

代码清单1-17 在代码中添加注释

```
class CommentSamples
{
  static void Main()
  {
                                                    单行注释
      string firstName; // Variable for storing the first name
      string lastName;  // Variable for storing the last name

      System.Console.WriteLine("Hey you!");

                                语句内部带分隔符的注释
      System.Console.Write /* No new line */ (
          "Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write /* No new line */ (
          "Enter your last name: ");
      lastName = System.Console.ReadLine();

      /* Display a greeting to the console
         using composite formatting. */          } 带分隔符的注释
      System.Console.WriteLine("Your full name is {0} {1}.",
          firstName, lastName);
      // This is the end
      // of the program listing
  }
}
```

输出1-7

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Inigo Montoya.
```

虽然插入了注释，但编译和执行新程序之后生成的输出和以前是一样的。

程序员使用注释来描述和解释他们写的代码，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，因而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1-17使用了其中的两种。

表1-2 C#注释类型

注释类型	说 明	例 子
带分隔符的注释	一个正斜杠，后跟一个星号，即/*，用于开始一条带分隔符的注释。要结束注释，需要在星号之后跟上一个正斜杠，即*/。这种形式的注释可在代码文件中跨越多行，也可以在一行代码中嵌入使用。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，仅用来进行格式处理	/*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。但是，可以连续使用多条单行注释，就像代码清单1-17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，而且可以把它们放到一个单独的文本文件中。XML带分隔符的注释是C# 2.0中新增的，但它的语法完全与C# 1.0兼容	/**注释**/
XML单行注释	XML单行注释以///开头，并延续到行末。除此之外，编译器可以将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9章将更全面地讨论XML注释。届时将讨论XML标准的各种XML标记。

编程史上，确实有一段时期认为，如果一个程序员编写的代码没有详尽的注释，都不好意思说自己是专业和有经验的程序员。但是时代变了。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具价值。如果开发人员发现需要写注释才能说清楚代码块的功用，应该考虑更清楚地重写代码，而不是洋洋洒洒写一堆注释。写注释其实就是重复代码思路的过程。不仅会造成混乱，降低可读性，还很容易过时，因为将来可能更改代码却没有及时更新注释。

规范

不要使用注释，除非代码本身“一言难尽”。

要尽量编写清晰的代码，而不是通过注释澄清复杂的算法。

初学者主题：XML

XML（Extensible Markup Language，可扩展标记语言）是一种简单而又灵活的文本格式，常用于Web应用程序以及应用程序间的数据交换。XML之所以是“可扩展”的，是因为XML文档中包含的是对数据进行描述的信息，也就是所谓的**元数据**（metadata）。下面是一个示例XML文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  <book title="Essential C# 5.0">
      <chapters>
          <chapter title="Introducing C#"/>
          <chapter title="Operators and Control Flow"/>
          ...
      </chapters>
  </book>
</body>
```

文件以Header元素开始，它描述了XML文件的版本和字符编码方式。之后是“book”元素。元素以尖括号中的单词开头，比如<body>。为了结束元素，要将同一个单词放在尖括号中，同时为单词添加一个正斜杠前缀，比如</body>。除了元素，XML还支持属性。title="Essential C# 5.0"就是XML属性的例子。注意，XML文件包含了对数据（如“Essential C# 5.0”、“Operators and Control Flow”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，但优点是提供了描述来帮助解释数据。

1.3.4 托管执行和公共语言基础结构

处理器不能直接解释程序集。程序集采用的是另一种语言，即**公共中间语言**（Common Intermediate Language，CIL），或简称为**中间语言**（IL）[①]。C#编译器将C#源代码文件转换成这种中间语言。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（通常在执行时进行）。该步骤涉及C#程序执行中的一个重要元素：VES（Virtual Execution System，**虚拟执行系统**）。VES偶尔也被人称为**运行时**，它根据需要编译CIL代码，这个过程称为**即时编译**或者**JIT编译**（just-in-time compilation）。假如代码在像“运行时”这样的一个“代理”的上下文中执行，就将这些代码称为**托管代码**（managed code），而在“运行时”的控制下执行的过程称为**托管执行**（managed execution）。之所以称为托管代码，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。执行过程中不需要“运行时”的代码称为**本机代码**（native code）或者**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”。为了明确起见，本书使用“执行时”来表示“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。

VES规范被包含在一个包容面更广的规范——CLI（Common Language Infrastructure，**公共语言基础结构**）规范[②]中，作为国际标准，CLI包含了以下几方面的规范。

- VES或“运行时”。
- CIL。
- 为语言互操作性提供支持的类型系统，称为CTS（Common Type System，**公共类型系统**）。
- 如何编写能通过CLI兼容语言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）。
- 一个公共编程框架，称为**基类库**（Base Class Library，BCL），所有语言的开发者都能利用它。

如果是在一个CLI实现的上下文中运行，程序员不需要直接写代码就能使用大量服务和功能，包括如下几个。

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

② 参见Miller,J.和S. Ragsdale编著的*The Common Language Infrastructure Annotated Standard*（Addison-Wesley，2004）。

- 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，确保只在兼容类型之间转换。这有助于防止发生缓冲区溢出——安全问题的一个主要诱因。
- 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明。
- 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：允许程序集在多种操作系统中运行。要做到这一点，一个显而易见的限制就是不能使用平台相关的库。所以，同Java一样，不可避免地有一些“个性”问题需要解决。
- BCL：提供开发者能够（在所有CLI实现中）依赖的大型代码库，使他们不必亲自编写这些代码。

> **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将会用到的一些术语。第21章将专门探讨CLI及其与C#开发者的关系。虽然那一章在本书的最后，但它的内容实际上并不依赖于之前的任何一章。所以，假如想多了解一下CLI，随时都能直接翻到那一章。

1.3.5 C#和.NET版本

你大概注意到了输出1-1中的“.NET Framework 4.5”字样。截至本书写作时为止，微软对.NET Framework进行了6次大的升级，但C#编译器只进行了5次大的升级。.NET Framework 3.0是在C#编译器的两次大的升级之间（以及Visual Studio 2005和2008之间）发布的，它补充了许多新的API。结果，和C# 3.0对应的.NET Framework版本是3.5。C# 4.0和.NET Framework 4.0发布时，两者的版本号再次同步。但和C# 5.0对应的是.NET Framework 4.5，所以现在的版本号又不同步了。表1-3简单总结了C#和.NET的版本。

表1-3 C#和.NET版本

版 本	描 述
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	新增了一套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)	添加了对LINQ的支持，对集合编程API进行了大幅改进。.NET Framework 3.5对原有的API进行了扩展以支持LINQ
C# 4.0和.NET Framework 4 (Visual Studio 2010)	添加了对动态类型的支持，对多线程编程API进行了大幅改进，强调了多处理器和核心支持。
C# 5.0和.NET Framework 4.5 (Visual Studio 2012)和WinRT集成	添加了对异步方法调用的支持，同时不需要显式注册委托回调。在框架中新增了对Windows Runtime(WinRT)互操作性的支持

只要编译器版本和代码要求的版本匹配，本书大多数代码都能在非微软平台上运行。虽然提供每种C#平台的完整细节能帮到一些人，但也会使人分心，所以本书最终只包含和Microsoft .NET平台有关的细节。原因很简单，微软公司的实现到目前为止仍占统治地位。而且即使需要转移到其他平台，转移的过程也相当容易。

.NET 4.5新增的“重头戏”是支持调用Windows Runtime（WinRT）中的组件。WinRT内部都是一些本机代码（native code），提供了.NET的一个全新执行环境。但无论API风格，还是“运行时”的行为，都像极了.NET。出于现实考虑，WinRT中的组件是一套全新API——大多数功能都和.NET Framework重叠。区别是新API只能在Windows 8上使用，而且都是从无到有设计出来的，旨在提供与.NET相似的编程体验，提供以前只能通过.NET包装器由“Win32”API提供的功能。虽然是“重头戏”，但作为平台的WinRT和学习.NET/C#基本无关。所以本书不打算深入讨论它。

1.3.6 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。处理器只能理解机器码，因此CIL代码必须先转换成机器码才能由处理器执行。给定一个程序集（DLL文件或可执行文件），可以使用CIL反汇编程序将其析构成对应的CIL表示，从而查看其CIL代码。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），它能对程序或者它的类库执行反汇编，显示由C#编译器生成的CIL代码。

在不同的CLI实现中，使用CIL反汇编程序的命令也有所区别。可以像输出1-8展示的那样，在命令行中执行.NET CIL反汇编程序：

输出1-8

```
>ildasm /text HelloWorld.exe
```

使用/text选项的目的是让输出在命令控制台上显示，而不是在新窗口中显示。执行上述命令得到的输出流是HelloWorld.exe程序所含CIL代码的一个“转储”（dump）。CIL代码比机器码更容易理解。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。

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，而不把它分发到用户机器上）。但假如目的只是减小别人获得源代码的可能性，可以考虑使用一些混淆器（obfuscator）产品。这些混淆器会打开IL代码，将代码加密成一种功能不变但更难于理解的形式。这可以防止普通开发者访问代码，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高级安全防护，否则混淆器足矣。

高级主题：HelloWorld.exe的CIL输出

代码清单1-18展示了ILDASM创建的CIL代码。

代码清单1-18 示例CIL输出

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.0.30319.17369
//  Copyright (c) Microsoft Corporation.  All rights reserved.
```

```
  // Metadata version: v4.0.30319
  .assembly extern mscorlib
  {
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                          //
➥.z\V.4..
    .ver 4:0:0:0
  }
  .assembly HelloWorld
  {
    .custom instance void [mscorlib]System.Runtime.CompilerServices.Compilat
➥ionRelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00 00 00 )
    .custom instance void [mscorlib]System.Runtime.CompilerServices.Runtime
➥CompatibilityAttribute::.ctor() = ( 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.
    .hash algorithm 0x00008004
    .ver 0:0:0:0
  }
  .module HelloWorld.exe
  // MVID: {D229AC10-1DEC-47A1-AA62-3BA19389E37E}
  .imagebase 0x00400000
  .file alignment 0x00000200
  .stackreserve 0x00100000
  .subsystem 0x0003       // WINDOWS_CUI
  .corflags 0x00000001    //  ILONLY
  // Image base: 0x00490000


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

.class private auto ansi beforefieldinit HelloWorld
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main() cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello. My name is Inigo Montoya."
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method HelloWorld::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method HelloWorld::.ctor

} // end of class HelloWorld
```

```
// ===========================================================

// *********** DISASSEMBLY COMPLETE ***********************
```

程序开头是清单（manifest）信息。其中不仅包括被反编译的模块的全名（HelloWorld.exe），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，这可比阅读并理解机器码（汇编程序）容易多了。上述代码中出现了对System.Console.WriteLine()的显式引用。CIL代码清单包含许多暂时没有什么用处的外围信息，但如果开发者想要理解C#模块（或者任何基于CLI的程序）的内部工作原理，但又拿不到源代码，那么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）可以将CIL自动反编译成C#。

1.4 小结

本章对C#进行了初步介绍。通过本章的学习，你熟悉了基本的C#语法。由于C#与C++风格语言的相似性，本章许多内容可能都是你所熟悉的。然而，C#和托管代码确实有一些独特性，比如会编译成CIL等。C#的另一个关键特征在于它是完全面向对象的。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。

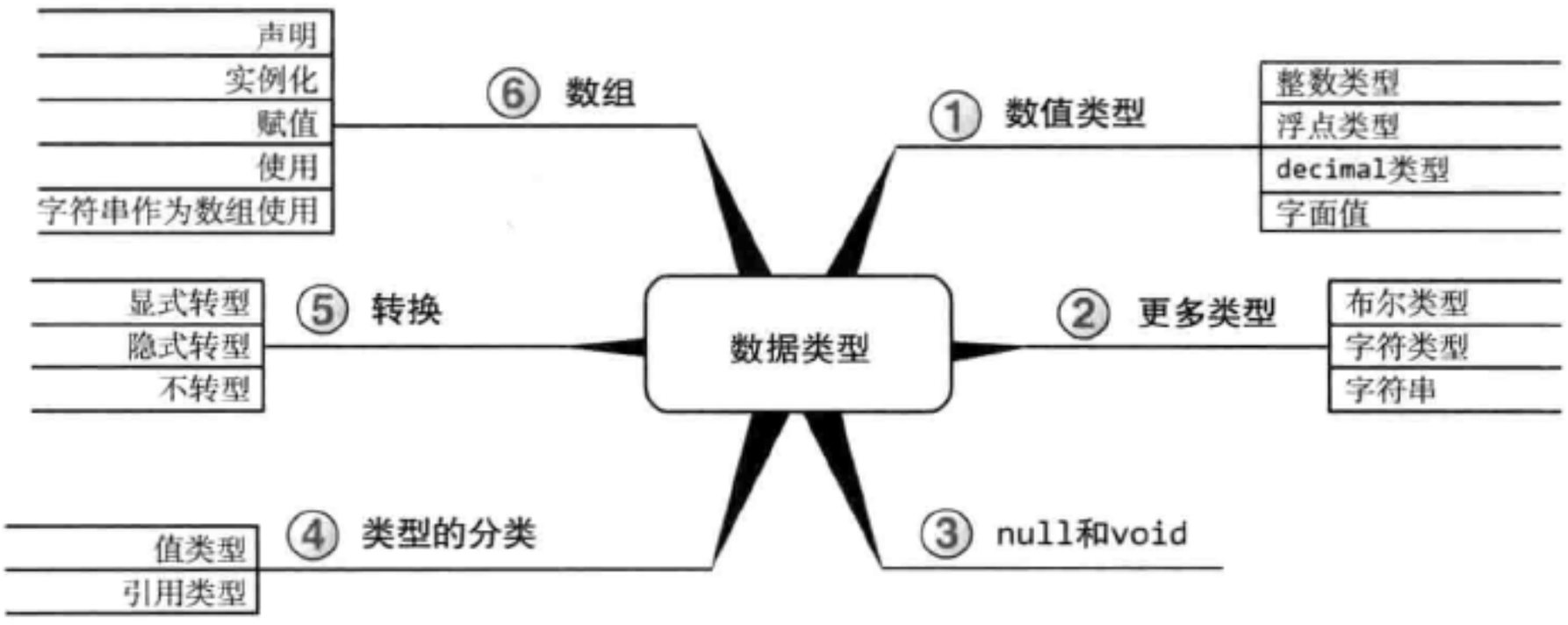
下一章将探讨C#语言中的基本数据类型，并讨论如何将这些数据类型应用于操作数来构成表达式。

第2章

数据类型

以第1章的HelloWorld程序为基础，你已经对C#语言、它的结构、基本语法特征以及如何编写最简单的程序有了初步理解。本章将探讨基本C#类型，继续学习C#的基础知识。

到目前为止，只用过少量基本数据类型，而且只是一笔带过，没有详细解释。C#有大量类型，而且可以合并不同类型来创建新类型。C#有几种类型非常简单，被视为其他所有类型的基础。这些类型称为**预定义类型**（predefined type）或者**基元类型**（primitive type）[①]。C#语言的基元类型包括8种整数类型、2种用于科学计算的二进制浮点类型、1种用于金融计算的十进制浮点类型、1种布尔类型以及1种字符类型。本章将探讨这些基元数据类型，更深入地研究string类型，并引入数组的概念。



2.1 基本数值类型

C#的基本数值类型都有关键字和它们关联。这些类型包括整数类型、浮点类型以及decimal类型。decimal是一种特殊的浮点类型，能够存储大数字而无表示错误。

① 也翻译为“基本类型”、“基础类型”或者“原始类型”。——译者注

2.1.1 整数类型

C#共有8种整数类型，可以选择最恰当的一种来存放数据，以避免浪费资源。表2-1对每种整数类型进行了总结。

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	后缀
sbyte	8位	-128 ~ 127	System.SByte	是	
byte	8位	0 ~ 255	System.Byte	否	
short	16位	-32 768 ~ 32 767	System.Int16	是	
ushort	16位	0 ~ 65 535	System.UInt16	否	
int	32位	-2 147 483 648 ~ 2 147 483 647	System.Int32	是	
uint	32位	0 ~ 4 294 967 295	System.UInt32	否	U或u
long	64位	-9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807	System.Int64	是	L或l
ulong	64位	0 ~ 18 446 744 073 709 551 615	System.UInt64	否	UL或ul

表2-1（以及表2-2和表2-3）中有一列专门给出了每种类型的完整名称。本章稍后会讲述后缀问题。C#的所有基元类型都有短名称和完整名称。完整名称对应于BCL中的类型命名。该名称在所有语言中都是相同的，而且对程序集里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缩写。其实从编译器的角度看，两种名称是完全一样的，最终都生成相同的代码。事实上，检查最终生成的CIL代码，根本看不出源代码中具体使用的名称。

C#支持完整BCL名称和关键字，造成开发人员为在什么时候用什么而犯难。不要时而用这个，时而用那个，最好坚持用一种。C#开发人员一般选择使用C#关键字。例如，使用int而不是System.Int32，使用string而不是System.String（甚至不要使用String这种简化形式）。

规范

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

要保持一致而不要变来变去。

坚持一致性可能会和其他编码规范冲突。例如，虽然应该用C#关键字取代BCL名称，但有时需要维护风格相反的文件（或文件库）。这时最好维持原有风格，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，如果原有“风格”实际上是不好的编码实践，有可能引入bug，严重妨碍维护，还是应该尽一切努力彻底纠正问题。

语言对比：C++ —— short数据类型

在C/C++中，short数据类型是short int的缩写。在C#中，short本身就是一种实际的数据类型。

2.1.2 浮点类型

浮点数的精度是可变的。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地表示该数[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非常大的数字（6.02×1023），即使误差为108，结果仍然非常接近6.02×1023，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，浮点数的精度由有效数字的个数决定，而不是由一个固定值（比如± 0.01）决定。

C#支持表2-2所示的两种浮点数。

表2-2 浮点类型

类型	大小	范围	BCL名称	有效数字	后缀
float	32位	$\pm 1.5 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$	System.Single	7	F或f
double	64位	$\pm 5.0 \times 10^{-324} \sim \pm 1.7 \times 10^{308}$	System.Double	15～16	与

二进制数被转换成十进制数，以便于人们理解。二进制数位被转换成15位十进制数位，余数构成第16位十进制数位，如表2-2所示。具体地说，$1.7 \times 10^{307} \sim 1 \times 10^{308}$的数只有15位有效数字。但$1 \times 10^{308} \sim 1.7 \times 10^{308}$的数有16位有效数字。decimal类型的有效数字范围与此相似。

2.1.3 decimal类型

C#还提供了128位精度的十进制浮点类型（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	BCL名称	有效数字	后缀
decimal	128位	$\pm 1.0 \times 10^{-28} \sim$ 大约 $\pm 7.9 \times 10^{28}$	System.Decimal	28～29	M或m

与浮点数不同，decimal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，而不是一个近似值。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（虽然这个差别可以忽略不计）。

高级主题：解析浮点类型和decimal类型

除非超过范围，否则decimal数字表示的十进制数都是完全准确的。与此相反，用二进制浮点数表示十进制数，则可能造成舍入错误。这类似于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。两种情况下都会产生某种形式的舍入错误。

decimal被表示成$\pm N \times 10^k$；其中，N是96位的正整数，而k是指数，-28 <= k <= 0。

而二进制浮点数是$\pm N \times 2^k$的任意数字。其中，N是用24位（float）或53位（double）的正整数，k是-149～+104（float）或者-1075～+970（double）范围内的任何整数。

① 例如0.1，表示成分数是1/10，分母10不是2的整数次幂，因此1/10不能用有限二进制小数表示。——译者注

2.1.4 字面值

字面值或**字面量**（literal value）表示源代码中的固定值。例如，假定希望用System.Console. WriteLine()输出整数值42和double值1.618 034（黄金分割比例），可以使用如代码清单2-1所示的代码。

代码清单2-1 指定字面值

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

输出2-1展示了代码清单2-1的结果。

输出2-1

```
42
1.618034
```

初学者主题：在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，所以开发者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获取值，比如从一个配置文件中。这样，以后需要修改值的时候，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（没有小数点）通常默认为int，前提是该值不是太大，以至于无法用int来存储。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数值类型赋值，前提是字面值对于目标数据类型来说是合法的。例如，short s = 42和byte b = 77都是允许的。但这一点仅对字面值成立。如果不使用额外的语法，b = s就是非法的，具体请参见2.6节。

2.1节讲过，C#有许多数值类型。在代码清单2-2中，一个字面值被直接放到C#代码中。由于带小数点的值默认为double类型，所以如输出2-2所示，结果是1.61803398874989（最后一个数字5丢失了），这符合我们预期的double值的精度。

代码清单2-2 指定一个double字面值

```
System.Console.WriteLine(1.618033988749895);
```

输出2-2

```
1.61803398874989
```

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，这是通过追加一个M（或者m）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 指定一个decimal字面值

```
System.Console.WriteLine(1.618033988749895M);
```

输出2-3

```
1.6180339887498895
```

现在，代码清单2-3的输出与预期的结果相同：1.618033988749895。注意，d表示double，之所以用m表示decimal，是因为这种数据类型经常用于货币（monetary）计算。

还可以使用F和D作为后缀，将字面量显式声明为float或者double。对于整数数据类型，相应的后缀是U、L、LU和UL。整数字面量的类型是像下面这样确定的：

- 没有后缀的数值字面量按照以下顺序，解析成能够存储该值的第一种数据类型：int、uint、long、ulong。
- 具有后缀U的数值字面量按照以下顺序，解析成能够存储该值的第一种数据类型：uint、ulong。
- 具有后缀L的数值字面量按照以下顺序，解析成能够存储该值的第一个数据类型：long、ulong。
- 如果数值字面量的后缀是UL或LU，就解析成ulong类型。

注意，字面量的后缀不区分大小写。但对于long，一般推荐使用大写字母L，因为小写字母l和数字1不好区分。

某些时候应考虑使用指数记数法，避免在小数点前后写许多个0。为了使用指数记数法，需要使用e或E中缀，在中缀字母后面添加正整数或者负整数，并在字面量的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2-4和输出2-4所示。

代码清单2-4 指数记数法

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

输出2-4

```
6.023E+23
```

规范

要使用大写的字面量后缀（如1.6180339887498895M）。

初学者主题：十六进制记数法

我们一般使用的都是十进制记数法。换言之，每位数字都可以使用10个符号（0~9）表示。除此之外，还可以使用十六进制记数法。换言之，每位数字都可以使用16个符号表示：0~9，A~F（也可以用小写）。所以，0x000A对应于十进制值10，而0x002A对应于十进制值42。不过，实际的数是相同的。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。

每位十六进制数字都是4个位，所以一个字节可以表示两位十六进制数字。

前面讨论数值字面量的时候只使用了十进制值。事实上，C#还允许指定十六进制值。为了指定十六进制值，要为值附加0x前缀，再添加希望使用的任何十六进制数字，如代码清单2-5所示。

代码清单2-5 十六进制字面值

```
// Display the value 42 using a hexadecimal literal.
System.Console.WriteLine(0x002A);
```

输出2-5展示了结果。

输出2-5

```
42
```

注意，代码输出的仍然是42，而不是0x002A。

高级主题：将数格式化成十六进制

要以十六进制形式输出一个数值，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2-6展示了一个例子。

代码清单2-6 十六进制格式说明符的例子

```
// Displays "0x2A"
System.Console.WriteLine("0x{0:X}", 42);
```

输出2-6展示了结果。

输出2-6

```
0x2A
```

注意，数值字面量（42）可以随意采用十进制或十六进制形式，结果是一样的。

高级主题：round-trip格式化

默认情况下，执行System.Console.WriteLine(1.618033988749895);语句会显示1.61803398874989，最后一个数位会被丢弃。为了更准确地支持与double值对应的字符串形式，我们可以使用格式字符串和round-trip格式说明符R（或者r）进行转换。例如，string.Format ("{0:R}", 1.618033988749895)会返回结果1.618033988749895。

将round-trip格式说明符返回的字符串转换回数值肯定能获得原始值。所以在代码清单2-7中，如果没有使用round-trip格式，两个数字就不相等了。

代码清单2-7 使用R格式说明符进行格式化

```
// ...
const double number = 1.618033988749895;
double result;
string text;

text = string.Format("{0}", number);
result = double.Parse(text);
System.Console.WriteLine("{0}: result != number",
  result != number);

text = string.Format("{0:R}", number);
result = double.Parse(text);
```

```
System.Console.WriteLine("{0}: result == number",
  result == number);
// ...
```

输出2-7显示了结果。

输出2-7

```
True: 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。

如果你还不熟悉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操作符。

2.2 更多基本类型

我们迄今为止只讨论了基本数值类型。C#还包括其他一些类型：bool、char和string。

2.2.1 布尔类型

另一个C#基元类型是布尔（Boolean）类型bool，也称为条件类型。它在条件语句和表达式中表示真或假。允许的值包括关键字true和false。bool的BCL名称是System.Boolean。例如，为了在不区分大小写的前提下比较两个字符串，可以调用string.Compare()方法并传递bool字面量true，如代码清单2-8所示。

代码清单2-8 以不区分大小写的方式比较两个字符串

```
string option;
...
int comparison = string.Compare(option, "/Help", true);
```

在这个例子中，我们以不区分大小写的方式比较变量option的内容和字面量/Help，并将结果赋给comparison。

虽然从理论上说，一个二进制位就足以容纳一个布尔类型的值，但bool数据类型的实际大小是一个字节。

2.2.2 字符类型

字符类型char表示16位字符，其取值范围对应于Unicode字符集。从技术上说，char的大小和16位无符号整数（ushort）相同，后者的取值范围是0～65 535。但char是C#的一个独特类型，在代码中应该体现这一点。

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，用来表示大多数语言中的字符。它便于计算机系统构建本地化的应用程序，为不同的语言文化显示具有本地特色的字符。

高级主题：16位不足以表示所有Unicode字符

令人遗憾的是，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”[①]的char构成，总共32位。

为了输入char类型的字面量，需要将字符放到一对单引号中，比如'A'。所有键盘字符都可以像这样输入，包括字母、数字以及特殊符号。

有的字符不能直接插入源代码，需要进行特殊处理。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。由于反斜杠用于标志转义序列的开始，所以不能再用来直接表示一个仅斜杠字符，而要用\\表示反斜杠字符。

代码清单2-9输出一个单引号，因为\'代表一个单引号。

代码清单2-9 使用转义序列显示单引号

```
class SingleQuote
{
  static void Main()
  {
      System.Console.WriteLine('\'');
  }
}
```

表2-4总结了转义序列以及字符的Unicode编码。

表2-4 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	Alert (system beep)	\u0007
\b	退格	\u0008
\f	换页（Form feed）	\u000C
\n	换行（Line feed或者newline）	\u000A
\r	回车	\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（前三个占位符是可选的），\uxxxx的长度可变版本	\u3A
\Uxxxxxxxx	Unicode转义序列，用于创建代理项对	\UD840DC01（乁）

① 代理项的详情参见1.3.1节的译者注。——译者注

可以使用Unicode代码表示任何字符。为此，请为Unicode值附加\u前缀。可以使用十六进制记数法表示Unicode字符。例如，字母A的十六进制值是0x41，代码清单2-10使用Unicode字符显示笑脸符号（:)），输出2-8展示了结果。

代码清单2-10 使用Unicode编码显示笑脸符号

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

输出2-8

```
:)
```

2.2.3 字符串

零或多个字符组成的有限序列称为**字符串**。C#的基本字符串类型是string，BCL名称是System.String。对于已经熟悉了其他语言的开发者，string的一些特点或许是他们意想不到的。比如字符串逐字前缀字符@，以及string属于不可变类型的事实。

1. 字面量

为了将字面量字符串输入代码，要将文本放入双引号（"）内，就像HelloWorld程序中那样。字符串由字符构成，所以转义序列可以嵌入字符串内。

例如，代码清单2-11显示了两行文本。不过，这里没有使用System.Console.WriteLine()，而是使用System.Console.Write()来输出换行符\n。输出2-9展示了结果。

代码清单2-11 使用字符\n插入换行符

```
class DuelOfWits
{
  static void Main()
  {
      System.Console.Write(
          "\"Truly, you have a dizzying intellect.\"");
      System.Console.Write("\n\"Wait 'til I get going!\"\n");
  }
}
```

输出2-9

```
"Truly, you have a dizzying intellect."
"Wait 'til I get going!"
```

双引号要用转义序列输出，否则会被用于定义字符串开始与结束。

在C#中，可以在字符串前面使用@符号，指明转义序列不被处理。结果是一个**逐字字符串字面量**（verbatim string literal），它不仅将反斜杠当作普通字符处理，还会逐字解释所有空白字符。例如，代码清单2-12的三角形会在控制台上原样输出，其中包括反斜杠、换行符以及缩进。输出2-10展示了结果。

如果不使用@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。

代码清单2-12 使用逐字字符串字面量来显示一个三角形

```
class Triangle
{
  static void Main()
  {
      System.Console.Write(@"begin
              /\
             /  \
            /    \
           /      \
          /________\
end");
  }
}
```

输出2-10

```
begin
        /\
       /  \
      /    \
     /      \
    /________\
end
```

在以@开头的字符串中，唯一支持的转义序列是""，它代表一个双引号，这个双引号不会终止字符串。

语言对比：C++——在编译时拼接字符串

和C++不同，C#不会自动拼接字符串字面量。例如，不能像下面这样指定一个字符串字面量：

```
"Major Strasser has been shot. "
"Round up the usual suspects."
```

必须使用+操作符进行拼接（然而，如果编译器能在编译时计算结果，最终的CIL代码包含的还是单独一个字符串）。

假如同一个字符串字面量在程序集中多次出现，编译器在程序集中只定义字符串一次，且所有变量都将指向同一个字符串。这样一来，假如在代码中多次插入包含大量字符的同一个字符串字面量，最终的程序集只反映其中一个的大小。

2. 字符串方法

与System.Console类型相似，string类型也包含几个方法。可以用这些方法格式化、拼接和比较字符串。

在表2-5中，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，而是返回结果。

表2-5 string的静态方法

语句	例子
static string string.Format(string format, ...)	string text, firstName, lastName; //... text = string.Format("Your full name is {0} {1}.", firstName, lastName); // Display // "Your full name is <firstName> <lastName>." System.Console.WriteLine(text);
static string string.Concat(string str0, string str1)	string text, firstName, lastName; //... text = string.Concat(firstName, lastName); // Display "<firstName><lastName>", notice // that there is no space between names. System.Console.WriteLine(text);
static int string. Compare(string str0, string str1)	string option; //... // String comparison in which case matters. int result = string.Compare(option, "/help"); // Display: // 0 if equal // negative if option < /help // positive if option > /help System.Console.WriteLine(result);
	string option; //... // Case-insensitive string comparison int result = string.Compare(option, "/Help", true); // Display: // 0 if equal // < 0 if option < /help // > 0 if option > /help System.Console.WriteLine(result);

表2-5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（如concate）之前附加包含了该方法（如string）的那个类型的名称。然而，string类还包含一些实例方法。实例方法不是以类型名作为前缀，而是以变量名（或者对实例的其他引用）作为前缀。表2-6列出了部分实例方法及其示例。

表2-6 string方法

语 句	例 子
bool StartsWith(string value) bool EndsWith(string value)	string lastName //... bool isPhd = lastName. EndsWith("Ph.D."); bool isDr = lastName. StartsWith("Dr.");

续表

语 句	例 子
string ToLower() string ToUpper()	string severity = "warning"; // Display the severity in uppercase System.Console.WriteLine(severity. ToUpper());
string Trim() string Trim(...) string TrimEnd() string TrimStart()	// Remove any whitespace at the start or end. username = username.Trim();
string Replace(string oldValue, string newValue)	string filename; //... // Remove ?'s from the string filename = filename.Replace("?", "");;

3. 换行符

输出新行所需的字符取决于执行代码的操作系统。在Microsoft Windows平台上，换行符是\r和\n这两个字符的组合，在UNIX上使用的则是单个\n。为了消除平台之间的不一致，一个办法是使用System.Console.WriteLine()输出空行。如果不准备在控制台上输出，同时仍然需要在多个平台上执行，那么可以使用System.Environment.NewLine。换言之，System.Console. WriteLine ("Hello World")和System.Console.Write("Hello World" + System.Environment.NewLine)是等价的。

高级主题：C#属性

从技术上讲，下一节提到的Length成员实际不是方法，因为调用它时没有使用圆括号。Length实际是string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，属性具有称为赋值方法（setter）和取值方法（getter）的特殊方法，但要使用字段语法访问那些方法。

研究属性的底层CIL实现，发现它编译成两个方法：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5章更详细地讨论了属性。

4. 字符串长度

判断字符串的长度，可以使用string的Length成员。该成员是一个只读属性。不能设置它，调用它也不需要任何参数。代码清单2-13演示了如何使用Length属性，输出2-11展示了结果。

代码清单2-13 使用string的Length成员

```
class PalindromeLength
{
  static void Main()
  {
      string palindrome;

      System.Console.Write("Enter a palindrome: ");
      palindrome = System.Console.ReadLine();
```

```
        System.Console.WriteLine(
            "The palindrome, \"{0}\" is {1} characters.",
            palindrome, palindrome.Length);
    }
}
```

输出2-11

```
Enter a palindrome: Never odd or even
The palindrome, "Never odd or even" is 17 characters.
```

字符串的长度不能直接设置，它是根据字符串中的字符数计算得到的。此外，字符串的长度不能更改，因为字符串是**不可变的**。

5. 字符串是不可变的

string类型的关键特征在于**它不可变的**（immutable）。可以为string变量赋一个全新的值，但出于性能方面的考虑，没有提供修改字符串内容的机制。所以，不可能将字符串中的字母全部转换为大写。只能新建字符串，让它成为旧字符串的大写字母版本，但旧字符串在这个过程中不会被修改。代码清单2-14展示了一个例子。

代码清单2-14 错误，string是不可变的

```
class Uppercase
{
  static void Main()
  {
      string text;

      System.Console.Write("Enter text: ");
      text = System.Console.ReadLine();

      // UNEXPECTED:  Does not convert text to uppercase
      text.ToUpper();

      System.Console.WriteLine(text);
  }
}
```

输出2-12展示了结果。

输出2-12

```
Enter text: 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中的字符转换成大写。但由于string类型是不可变的，所以text.ToUpper()不会进行这样的修改。相反，text.ToUpper()会返回一个新字符串，它需要保存到变量中，或者直接传给System.Console.WriteLine()。代码清单2-15给出了纠正后的代码，输出2-13展示了结果。

代码清单2-15 字符串处理

```
class Uppercase
{
```

```
    static void Main()
    {
        string text, uppercase;

        System.Console.Write("Enter text: ");
        text = System.Console.ReadLine();

        // Return a new string in uppercase
        uppercase = text.ToUpper();

        System.Console.WriteLine(uppercase);
    }
}
```

输出2-13

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

假如忘记字符串不可变的特点，很容易就会在使用其他字符串方法时犯下和代码清单2-14相似的错误。

为了真正更改text中的值，要将ToUpper()的返回值赋回给text，如下例所示：

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

6. System.Text.StringBuilder

如果有大量字符串需要修改，比如要经历多个步骤来构造一个长字符串，那么应当使用数据类型System.Text.StringBuilder，而不是string。StringBuilder包含Append()、AppendFormat()、Insert()、Remove()和Replace()等方法。虽然string也提供了其中一些方法，但两者关键的区别在于，在StringBuilder上，这些方法会修改StringBuilder本身中的数据，而不是简单地返回一个新字符串。

2.3 null和void

与类型有关的两个额外的关键字是null和void。null值表明变量不引用任何有效的对象。void表示没有类型，或者没有任何值。

2.3.1 null

null也可以作为字符串字面量的类型使用。null表示将变量设为“无”。null值只能赋给引用类型、指针类型和可空值类型。本书到目前为止只讲过了string这一种引用类型，第5章将详细讨论创建类（它们是引用类型）。现在只需知道引用类型包含的是对内存中的一个位置的引用，与实际数据的存储位置不同。将变量设为null，会显式地设置引用，使它不指向任何位置。事实上，甚至可以检查引用类型的变量是否不指向任何位置。代码清单2-16演示了如何将null赋给string变量。

代码清单2-16 将null赋给字符串变量

```
static void Main()
{
    string faxNumber;
```

```
        // ...

        // Clear the value of faxNumber.
        faxNumber = null;

        // ...
    }
```

必须注意，和根本不赋值相比，将null赋给引用类型的变量是完全不同的概念。换言之，赋值为null的变量已被设置，而未赋值的变量未被设置，所以假如在赋值前使用变量会造成编译时错误。

将null值赋给一个string，并不等同于将空字符串""赋给它。null意味着变量无任何值，而""意味着变量有一个称为“空字符串”的值。这种区分相当有用。例如，编程逻辑可以将为null的faxNumber解释成“传真号码未知”，而将为""的faxNumber解释成“无传真号码”。

2.3.2 void

有时候，C#语法要求指定数据类型但不传递任何数据。例如，假定一个方法没有返回值，C#就允许在数据类型的位置放一个void关键字。HelloWorld程序中的Main方法声明就是一个例子。在返回类型的位置使用void意味着方法不返回任何数据，同时告诉编译器不要指望会有一个值。void本质上并不是一个数据类型，它只是用于指出没有数据类型这一事实。

语言对比：C++

无论在C++中还是在C#中，void都有两个含义：一是用于标记方法不返回任何数据，二是用于表示指针指向未知类型的存储位置。C++程序中经常使用void**这样的指针类型。虽然C#中也可用相同的语法表示指向未知类型的存储位置的指针，但这种用法在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。

语言对比：Visual Basic——返回void就像定义子程序

在Visual Basic中，与C#的“返回void”等价的是定义子程序（Sub/End Sub）而不是返回值的函数。

高级主题：隐式类型的局部变量

C# 3.0新增了上下文关键字var来声明**隐式类型的局部变量**。只要在声明变量的同时用确定类型的表达式初始化它，C# 3.0和以后的版本就允许变量的数据类型是“隐式”的，而不必显式地指定数据类型，如代码清单2-17所示。

代码清单2-17 操纵字符串

```
class Uppercase
{
  static void Main()
```

```
    {
        System.Console.Write("Enter text: ");
        var text = System.Console.ReadLine();

        // Return a new string in uppercase
        var uppercase = text.ToUpper();

        System.Console.WriteLine(uppercase);
    }
}
```

这个代码清单和代码清单2-15相比只有两处不同。首先，声明时不是使用显式的数据类型string而是var。结果生成的CIL代码和显式使用string完全一样。但var告诉编译器：应该由编译器根据声明时所赋的值（System.Console. ReadLine()）来推断数据类型。

其次，text和uppercase这两个变量都是在声明的同时赋值。不赋值会造成编译时错误。如前所述，编译器根据所赋的值来推断等号右边的表达式的数据类型，并在最终的CIL代码中正确地声明变量，就像程序员显式指定了类型那样。

虽然允许使用var取代显式的数据类型，但在数据类型已知的情况下最好不要使用var。例如，最好将text和uppercase声明为string。这不仅使代码更容易理解，还能验证右侧表达式返回的是所期望的数据类型。用var声明变量，右侧的数据类型应该是非常明显的；否则应该避免使用var声明。

C# 3.0添加var的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的（参见第14章），代码清单2-18进行了演示。

代码清单2-18 隐式局部变量和匿名类型

```
class Program
{
  static void Main()
  {
      var patent1 =
          new { Title = "Bifocals",
          YearOfPublication = "1784" };
      var patent2 =
          new { Title = "Phonograph",
          YearOfPublication = "1877" };

      System.Console.WriteLine("{0} ({1})",
          patent1.Title, patent1.YearOfPublication);
      System.Console.WriteLine("{0} ({1})",
          patent2.Title, patent1.YearOfPublication);
  }
}
```

输出2-14展示了上述代码的输出。

输出2-14

```
Bifocals (1784)
Phonograph (1784)
```

代码清单2-18演示了如何将匿名类型的实例赋给隐式类型的局部变量（var变量）。这种形式的操作在C# 3.0中具有重要的作用，是连接（joining或associating）数据类型的关键。另外，它还能通过减少数据元素的数量来有效地减小类型的大小。

2.4 类型的分类

所有类型都可以划分为**值类型**和**引用类型**两类。它们的区别在于复制方式：值类型的数据总是进行值复制，而引用类型的数据总是进行引用复制。

2.4.1 值类型

除了string，本书迄今为止讲到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值在内存中实际存储的位置。因此，将第一个变量的值赋给第二个变量会在新变量的位置创建原始变量的值的一个内存副本。相同值类型的第二个变量不能引用和第一个变量相同的内存位置。所以，更改第一个变量的值不会影响第二个变量的值。图2-1对此进行了演示。number1引用内存中的特定的位置，该位置包含值42。将number1的值赋给number2之后，两个变量都包含值42。但是，修改其中任何一个值都不会影响另一个值。

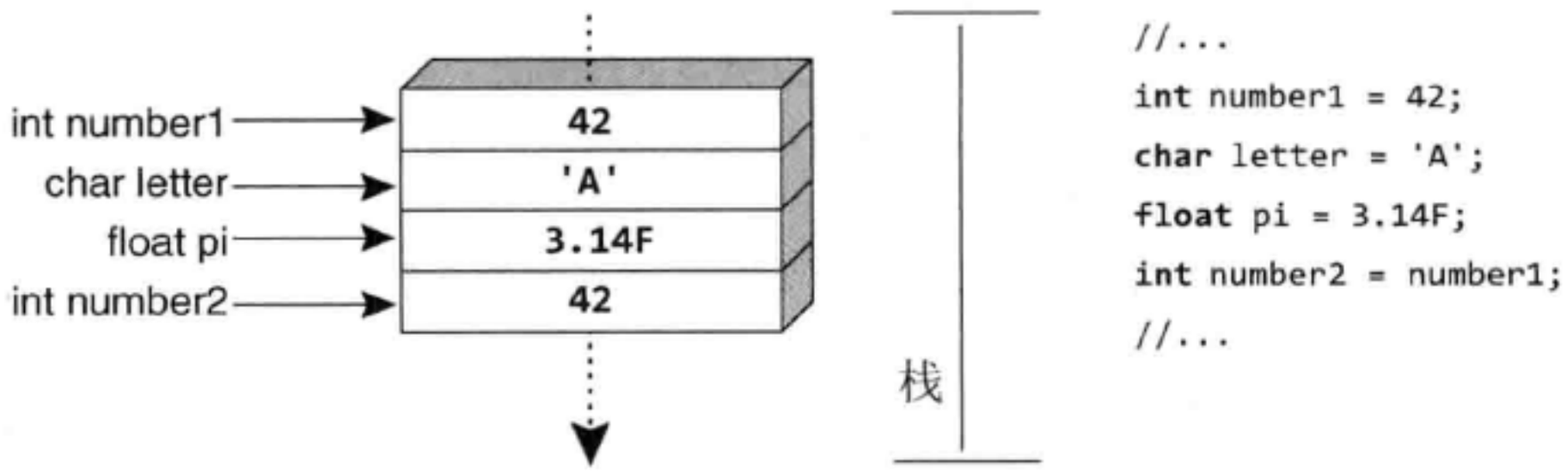


图2-1 值类型的实例直接包含数据

类似地，将值类型的实例传给方法，如Console.WriteLine()，也会生成一个内存副本。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中的原始值。由于值类型需要创建内存副本，因此定义时不要让它们占用太多内存（通常应该小于16字节）。

2.4.2 引用类型

相反，引用类型的变量存储的是对数据存储位置的引用，而不是直接存储数据。要去那个位置才能找到真正的数据。因此，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。引用类型指向的内存区域称为**堆**（heap），如图2-2所示。

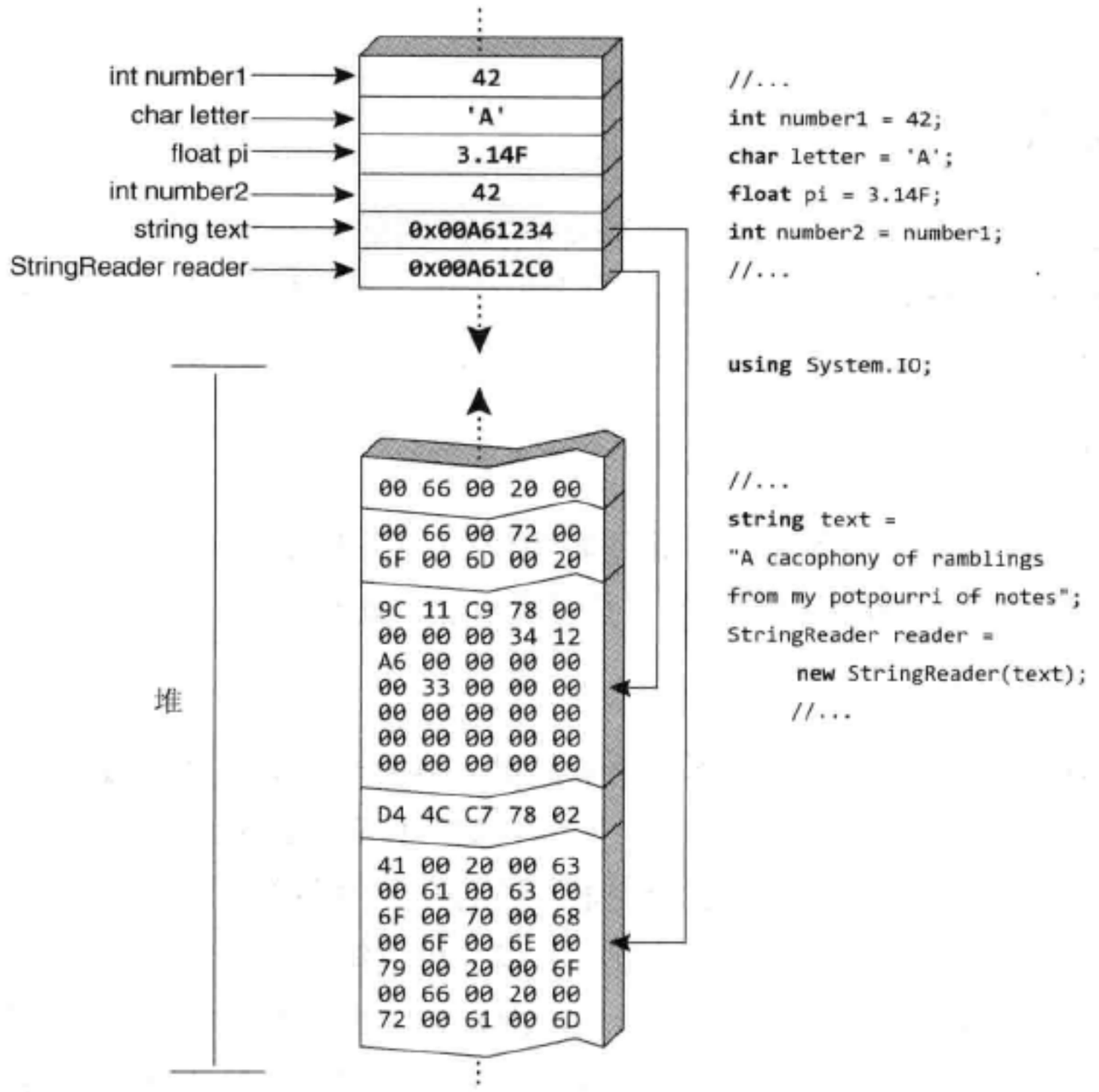


图2-2 引用类型指向堆

引用类型不像值类型那样要求创建数据的内存副本，所以复制引用类型的实例比复制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，只会复制引用而不需要复制所引用的数据。事实上，每个引用总是处理器的“原生大小”。也就是，32位处理器只需复制32位引用，64位处理器只需复制64位引用，以此类推。显然，复制对一个大数据块的引用，比复制整个数据块快得多。

由于引用类型只复制对数据的引用，所以两个不同的变量可引用相同的数据。如果两个变量引用同一个对象，利用一个变量更改对象的字段，用另一个对象访问字段时将看到更改结果。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，控制返回调用者之后，将看到更改后的结果。有鉴于此，在决定定义引用类型还是值类型时，一个决定性的因素就是：如果对象在逻辑上是固定大小的不可变的值，就考虑定义成值类型；如果逻辑上是可引用的可变的对象，就考虑定义成引用类型。

除了string以及自定义类（如Program），本书迄今为止讲到的所有类型都是值类型。但大多数类型都是引用类型。另外，虽然偶尔需要自定义的值类型，但更常用的还是自定义的引用类型。

2.5 可空修饰符

一般不能将null值赋给值类型。这是因为根据定义，值类型不能包含引用，即使是对“无（nothing）”的引用。但在值本来就缺失的时候，这也会带来问题。例如，在指定计数的时候，假如计数未知，应该如何输入?一个可能的解决方案是指定特殊值，比如-1或者

int.MaxValue，但这些都是有效的整数。我们倒更希望直接将null赋给值类型，因为这不是有效的整数。

为了声明可以存储null的变量，要使用可空修饰符?。代码清单2-19演示了从C# 2.0开始引入的这个特性。

代码清单2-19 使用可空修饰符

```
static void Main()
{
  int? count = null;
  do
  {
      // ...
  }
  while(count == null);
}
```

将null赋给值类型，这在数据库编程中尤其有用。在数据表中，经常出现值类型的列允许为空的情况。除非允许包含null值，否则在C#代码中检索这些列并将它们的值赋给对应的字段会出问题。幸好，可空修饰符能够妥善地解决这个问题。

2.6 数据类型之间的转换

考虑到各种CLI实现会预定义大量类型，加上代码能够定义无限数量的类型，所以类型之间的相互转换至关重要。会造成转换的最常见的操作就是**转型**或者**强制类型转换**（casting）。

考虑将long变量转换成int变量的情形。long类型能容纳的最大值是9 223 372 036 854 775 808，而int类型能容纳的最大值是2 147 483 647。所以，转换时可能丢失数据——long变量包含的值可能大于int能容纳的最大值。有可能造成大小变小或者引发异常（因为转换失败）的任何转换都需要执行**显式转型**（explicit cast）。相反，不会变小，而且不会引发异常（无论操作数的类型是什么）的任何转换都属于**隐式转型**（implicit cast）。

2.6.1 显式转型

在C#中，可以使用**转型操作符**执行转型。通过在圆括号中指定希望变量转换成的类型，表明你已确认了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2-20将一个long类型的变量转换成int类型的变量，而且显式地告诉系统尝试这个操作。

代码清单2-20 显式转型的例子

```
long longNumber = 50918309109;
int intNumber = (int) longNumber;
                ^^^^^
                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。我知道这个值能适应目标类型。”只有程序员像这样做出明确选择，编译器才允许转换。但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，“运行时”可能引发异常。所以，完全要由程序员负责确保数据成功转换，或者提供错误处理代码来处理不能成功转换的情况。

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，以至于容不下赋值的数据，那么会发生什么情况。假如容不下，数据在赋值时默认会溢出。代码清单2-21展示了一个例子。

代码清单2-21 溢出整数值

```
public class Program
{
  public static void Main()
  {
      // int.MaxValue equals 2147483647
      int n = int.MaxValue;
      n = n + 1 ;
      System.Console.WriteLine(n);
  }
}
```

输出2-15展示了结果。

输出2-15

```
-2147483648
```

代码清单2-21会在控制台上显示值-2147483648。然而，将上述代码放到一个checked块中，或者在编译时使用checked选项，就会使“运行时”引发System.OverflowException异常。代码清单2-22给出了checked块的语法。

代码清单2-22 checked块示例

```
public class Program
{
  public static void Main()
  {
      checked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
          n = n + 1 ;
          System.Console.WriteLine(n);
      }
  }
}
```

输出2-16展示了结果。

输出2-16

```
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow at Program.Main() in ...Program.cs:line 12
```

在checked块内，如果在运行时发生一次溢出的赋值，就会引发异常。

C#编译器提供了一个命令行选项将默认行为从unchecked改成checked。除此之外，C#还支持unchecked块，它强制不进行溢出检查，不会为块中溢出的赋值引发异常，如代码清单2-23所示。

代码清单2-23 unchecked块示例

```
using System;

public class Program
{
  public static void Main()
  {
      unchecked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
          n = n + 1 ;
          System.Console.WriteLine(n);
      }
  }
}
```

输出2-17展示了结果。

输出2-17

```
-2147483648
```

即使编译时打开了checked选项，在执行期间，上述代码中的unchecked关键字也会阻止“运行时”引发异常。

并不能因为你使用了转型操作符，就可以将一种类型转换为其他任何类型。编译器仍然会检查转型操作的有效性。例如，不能将一个long转换成bool。因为没有定义这种转换，所以编译器不允许这样的转型。

语言对比：数值转换成布尔值

一些人可能觉得奇怪，C#中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都是很普遍的。C#不支持这样的转换，是为了避免可能发生的歧义，比如—1到底对应true还是false？更重要的是，如同下一章要讲到的那样，这还有助于避免用户在本来应该使用相等操作符的时候使用了赋值操作符。例如，可以避免在本来应该写成if(x==42){...}的时候写成了if(x=42){...}。

2.6.2 隐式转型

在有些情况下，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，编译器能断定这样的转换可以正常完成。代码清单2-24只使用一个赋值操作符，就实现了从int到long的转换。

代码清单2-24 隐式转型无需使用转型操作符

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

即使不要求显式转型操作符（因为允许隐式转型），仍然可以强制添加转型操作符，如代码清单2-25所示。

代码清单2-25 隐式转型使用转型操作符

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

2.6.3 不使用转型操作符的类型转换

由于没有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。每个数值数据类型都包含一个Parse()方法，它允许将字符串转换成对应的数值类型。代码清单2-26演示了这个调用。

代码清单2-26 使用int.Parse()将一个string转换为数值数据类型

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

还可利用特殊类型System.Convert将一种类型转换成另一种类型。代码清单2-27展示了它的用法。

代码清单2-27 使用System.Convert进行类型转换

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

不过，System.Convert只支持预定义类型，而且是不可扩展的。它允许从任何基元类型（bool、char、sbyte、short、int、long、ushort、uint、ulong、float、double、decimal、DateTime和string）转换到其他基元类型。

除此之外，所有类型都支持ToString()方法，可以用它提供一个类型的字符串表示。代码清单2-28演示了如何使用该方法，输出2-18展示了结果。

代码清单2-28 使用ToString()转换成一个string

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

输出2-18

```
True
```

对于大多数类型，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。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）开始，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，只是转换失败的情况下，它不引发异常，而是返回false，如代

码清单2-29所示。

代码清单2-29 用TryParse()代替无效转型异常

```
double number;
string input;

System.Console.Write("Enter a number: ");
input = System.Console.ReadLine();
if (double.TryParse(input, out number))
{
    // Converted correctly, now use number
    // ...
}
else
{
    System.Console.WriteLine(
        "The text entered was not a valid number.");
}
```

输出2-19展示了结果。

输出2-19

```
Enter a number: forty-two
The text entered was not a valid number.
```

在上述代码中，从输入字符串解析到的值是通过out参数（本例是number）返回的。

Parse()和TryParse()的关键区别在于，假如转换失败，TryParse()不会引发异常。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。用户完全有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（由于预见到用户会输入无效数据，所以要想办法避免引发异常）。

2.7 数组

第1章没有提到的一种特殊的变量声明就是数组声明。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#的数组索引是从零开始的。所以，我们说C#中的数组是**基于零**的。

初学者主题：数组

利用数组，我们可以使用变量来声明相同类型的多个数据项的集合。数组中每个数据项都使用名为**索引**的整数值进行唯一性标识。C#数组中的第一个数据项使用索引0访问。程序员应确保指定的索引值小于数组的大小（数组中的元素总数）。因为C#数组是基于零的，所以数组最后一个元素的索引值要比数组元素的总数小1。

初学者可将索引想象成偏移量。第一项距离数组开头的偏移量是0，第二项的偏移量是1，依此类推。

数组是几乎每一种编程语言的基本组成部分，因此所有开发人员都要学会它。虽然，C#编程经常用到数组，而且初学者也确实应该掌握它，但目前，在存储数据集合时，大多数程序使用的都是泛型集合类型而不是数组。如果只是为了熟悉一下数组的实例化和赋值，可以浏览下一节。表2-7列出了要注意的重点。泛型集合将在第11章中详细讲述。

表2-7 数组的重点

说明	例子
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。	`string[] languages; // one dimensional` `int[,] cells;  // two dimensional`
赋值 new关键字和对应的数据类型在声明时可选。 声明之后，要用new关键字实例化数组。 数组赋值可以不用字面值。这样，每一项都被初始化为默认值。 不提供字面值就必须指定数组大小。（数组的大小不一定是一个常量，也可以是一个在运行时能够计算出的变量）。 从C# 3.0开始可以不指定数据类型。	`string[] languages = { "C#", "COBOL", "Java",` `  "C++", "Visual Basic", "Pascal",` `  "Fortran", "Lisp", "J#"};` `languages = new string[9];` `languages = new string[]{"C#", "COBOL", "Java",` `  "C++", "Visual Basic", "Pascal",` `  "Fortran", "Lisp", "J#" };` `// Multidimensional array assignment` `// and initialization.` `int[,] cells = int[3,3];` `cells = {` `  {1, 0, 2},` `  {1, 2, 0},` `  {1, 2, 1}` `};`
default关键字 使用default操作符显式地获取任何数据类型的默认值。	`int count = default(int);`
访问数组 数组是基于零的，因此第一个元素的索引是0。用方括号存储和获取数组数据。	`string[] languages = new string[9]{` `  "C#", "COBOL", "Java",` `  "C++", "Visual Basic", "Pascal",` `  "Fortran", "Lisp", "J#"};` `// Save "C++" to variable called language.` `string language = languages[3];` `// Assign "Java" to the C++ position.` `languages[3] = languages[2];` `// Assign language to location of "Java".` `languages[2] = language;`

除此之外，在本章的最后一节“常见数组错误”中，还会讲到数组的一些特性。

2.7.1 数组的声明

在C#中，使用方括号声明数组变量。首先要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2-30将名为languages的变量声明为字符串数组。

代码清单2-30 数组的声明

```
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。在本例中，声明的是一个一维数组。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是紧跟在数据类型之后的，而不是出现在变量声明之后。这样一来，可以确保所有类型信息都放到一起，而不是像C++和Java那样，分散于标识符前后。

代码清单2-30定义的是一维数组。方括号中的逗号用于定义附加的维。例如，代码清单2-31为井字棋棋盘定义了一个二维数组。

代码清单2-31 声明二维数组

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

在代码清单2-31中，我们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。使用更多的逗号，可以定义更多的维，数组总维数等于逗号数加1。注意，在一个特定的维上，数据项的数量不是变量声明的一部分。这是在创建（实例化）数组，并为每个元素分配空间时指定的。

2.7.2 数组的实例化和赋值

声明数组之后，可以立即为其填充值。方法是在一对大括号中使用以逗号分隔的数据项列表。代码清单2-32声明了一个字符串数组，然后在一对大括号中指定了9种不同的编程语言的名称。

代码清单2-32 声明数组的同时进行赋值

```
string[] languages = { "C#", "COBOL", "Java",
    "C++", "Visual Basic", "Pascal",
    "Fortran", "Lisp", "J#"};
```

在这个以逗号分隔的列表中，第一项成为数组的第一个元素，第二项成为第二个元素，以此类推。我们使用大括号来定义数组字面量。

只有在同一个语句中声明并赋值，才能使用代码清单2-32的赋值语法。如果在声明后赋值，则需要使用new关键字，如代码清单2-33所示。

代码清单2-33 声明后再进行数组赋值

```
string[] languages;
languages = new string[]{"C#", "COBOL", "Java",
    "C++", "Visual Basic", "Pascal",
    "Fortran", "Lisp", "J#" };
```

自C# 3.0起，不必在new后面指定数组的数据类型（string），只要编译器能根据初始化列表中的数据类型推断出数组元素的类型。但是，方括号仍然不可缺少。

C#还支持将new关键字作为声明语句的一部分，所以允许像代码清单2-34那样的赋值和声明。

代码清单2-34 声明的同时使用new进行数组赋值

```
string[] languages = new string[]{
    "C#", "COBOL", "Java",
    "C++", "Visual Basic", "Pascal",
    "Fortran", "Lisp", "J#"};
```

使用new关键字会让“运行时”为数据类型分配内存。它指示“运行时”实例化数据类型——本例是一个数组。

只要将new关键字作为数组赋值的一部分，就可以同时在方括号内指定数组的大小。代码清单2-35演示了这个语法。

代码清单2-35 使用new关键字时可指定数组的大小

```
string[] languages = new string[9]{
    "C#", "COBOL", "Java",
    "C++", "Visual Basic", "Pascal",
    "Fortran", "Lisp", "J#"};
```

在初始化语句中指定的数组的大小必须和大括号中包含的元素数量相匹配。除此之外，也可以分配数组但不指定初始值，如代码清单2-36所示。

代码清单2-36 分配数组但不指定初始值

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

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

- 引用类型（如string）初始化为null；
- 数值类型初始化为零；
- 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，每个字段都被初始化为默认值。

结果是，在使用前不必单独对数组的每个元素进行赋值。

从C# 2.0开始可以使用default()表达式判断数据类型的默认值。default()获取数据类型作为参数。例如，default(int)返回0，而default(bool)返回false。

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2-37根据在Console.ReadLine()调用中由用户指定的大小创建数组。

代码清单2-37 在运行时定义数组大小

```
string[] groceryList;
System.Console.Write("How many items on the list? ");
int size = int.Parse(System.Console.ReadLine());
groceryList = new string[size];
// ...
```

C#采取类似的方式处理多维数组。要以逗号分隔每一维上的大小。代码清单2-38初始化了一个没有开始走棋的井字棋棋盘。

代码清单2-38 声明二维数组

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

还可以像代码清单2-39那样，将井字棋棋盘初始化成特定的棋子布局。

代码清单2-39 初始化二维整数数组

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

数组包含3个int[]类型的元素，每个元素都具有相同的大小。在本例中，这个大小是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2-40那样的声明是无效的。

代码清单2-40 大小不一致的多维数组会造成错误

```
// ERROR:  Each dimension must be consistently sized.
int[,] cells = {
        {1, 0, 2, 0},
        {1, 2, 0},
        {1, 2}
        {1}
    };
```

要表示井字棋棋盘，并不需要在每个位置都使用整数。另一个办法是为每个玩家都提供单独的虚拟棋盘，每个棋盘都包含一个bool来指出玩家选择的位置。代码清单2-41对应于一个三维棋盘。

代码清单2-41 初始化一个三维数组

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

    // Player 2 moves                   //    |   | O
    {   {false, false, true},           // ---+---+---
        {false, true, false},           //    | O |
        {false, true, true} }           // ---+---+---
                                        //    | O |
};
```

在这个例子中，棋盘被初始化，每一维的大小被显式指定。new表达式中除了指定大小，还提供了数组的字面值。bool[,,]类型的字面值被分解成两个bool[,]类型的数组，每

个数组的大小都是3×3。每个二维数组都由3个bool数组（大小为3）构成。

如前所述，多维数组的每一维的大小都必须一致。我们还可以定义**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2-42那样初始化交错数组。

代码清单2-42 初始化交错数组

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

交错数组不使用逗号标识新的维。相反，交错数组定义由数组构成的数组。在代码清单2-42中，int[]是数据类型，所以它之后的[]声明了一个int[]类型的数组。

注意，交错数组要求为内部的每个数组都创建数组实例。在这个例子中，我们使用new实例化交错数组的内部元素。如果遗失这个实例化部分，会造成编译时错误。

2.7.3 数组的使用

可以使用方括号表示法，也就是所谓的**数组访问符**（array accessor），来访问数组中一个特定的数据项。为了获取数组的第一个数据项，要指定0作为索引。在代码清单2-43中，我们将languages变量中的第5项（其索引为4，因为第一项的索引是0）的值存储到变量language中。

代码清单2-43 声明并访问数组

```
string[] languages = new string[9]{
    "C#", "COBOL", "Java",
    "C++", "Visual Basic", "Pascal",
    "Fortran", "Lisp", "J#"};
// Retrieve 3rd item in languages array (Java)
string language = languages[4];
```

方括号表示法还用于将数据存储到数组中。代码清单2-44交换了"C++"和"Java"的顺序。

代码清单2-44 交换数组中不同位置的数据

```
string[] languages = new string[9]{
    "C#", "COBOL", "Java",
    "C++", "Visual Basic", "Pascal",
    "Fortran", "Lisp", "J#"};
// Save "C++" to variable called language.
string language = languages[3];
// Assign "Java" to the C++ position.
languages[3] = languages[2];
// Assign language to location of "Java".
languages[2] = language;
```

在多维数组中，元素是用每个维上的一个索引来标识的，如代码清单2-45所示。

代码清单2-45 初始化二维整数数组

```
int[,] cells = {
        {1, 0, 2},
        {0, 2, 0},
        {1, 2, 1}
    };
// Set the winning tic-tac-toe move to be player 1.
cells[1,0] = 1;
```

交错数组元素的赋值稍微有所不同，这是因为它必须与交错数组的声明保持一致。第一个索引指定了“由数组构成的数组”中的一个数组。第二个索引则指定了是该数组中的哪一项（参见代码清单2-46）。

代码清单2-46 声明一个交错数组

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

cells[1][0] = 1;
// ...
```

1. 长度

像代码清单2-47那样获取数组的长度。

代码清单2-47 获取数组的长度

```
Console.WriteLine("There are {0} languages in the array.",
    languages.Length);
```

数组的长度是固定的，不能随便更改，除非重新创建数组。除此之外，越过数组的边界（或长度）会造成“运行时”报错。用无效索引（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2-48中，如果用数组长度作为索引来访问数组就会出错。

代码清单2-48 访问超过数组的边界会引发异常

```
string languages = new string[9];
...
// RUNTIME ERROR: index out of bounds - should
// be 8 for the last element
languages[4] = languages[9];
```

> **注意**
>
> Length成员返回数组中数据项的个数，而不是返回最高的索引值。languages变量的Length成员是9，而languages变量的最高索引是8，那是从起点能到达的最远的一个位置。

语言对比：C++——缓冲区溢出错误

非托管的C++并非总是检查是否越过数组的边界。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**（buffer overrun）。相反，CLR能防止所有C#（和托管C++）代码越过数组边界，消除了在托管代码中发生缓冲区溢出的可能。

比较好的做法是使用Length代替硬编码的数组大小。例如，为了将Length作为索引来使用，有必要在它上面减1，以避免越界错误，如代码清单2-49所示。

代码清单2-49 在数组索引中使用Length-1

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

为了避免越过数组边界，应使用长度检查来验证数组的长度大于0。在访问数组中的最后一项时，要像代码清单2-49那样，使用Length-1而不是硬编码的值。

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返回的是外部数组的元素数——交错数组是“数组构成的数组”，所以Length只作用于外部数组，只统计它的元素数（也就是具体由多少个数组构成），而不管各内部数组共包含了多少个元素。

2. 更多的数组方法

数组提供了更多的方法来操纵数组中的元素，其中包括Sort()、BinarySearch()、Reverse()和Clear()等，如代码清单2-50所示。

代码清单2-50 更多的数组方法

```
class ProgrammingLanguages
{
  static void Main()
  {
      string[] languages = new string[]{
          "C#", "COBOL", "Java",
          "C++", "Visual Basic", "Pascal",
          "Fortran", "Lisp", "J#"};

      System.Array.Sort(languages);

      string searchString = "COBOL";
      int index = System.Array.BinarySearch(
          languages, searchString);
      System.Console.WriteLine(
          "The wave of the future, {0}, is at index {1}.",
          searchString, index);

      System.Console.WriteLine();
      System.Console.WriteLine("{0,-20}\t{1,-20}",
          "First Element", "Last Element");
```

```
        System.Console.WriteLine("{0,-20}\t{1,-20}",
            "-------------", "-------------");
        System.Console.WriteLine("{0,-20}\t{1,-20}",
            languages[0], languages[languages.Length-1]);

        System.Array.Reverse(languages);
        System.Console.WriteLine("{0,-20}\t{1,-20}",
            languages[0], languages[languages.Length-1]);

        // Note this does not remove all items from the array.
        // Rather it sets each item to the type's default value.
        System.Array.Clear(languages, 0, languages.Length);
        System.Console.WriteLine("{0,-20}\t{1,-20}",
            languages[0], languages[languages.Length-1]);
        System.Console.WriteLine(
            "After clearing, the array size is: {0}",
            languages.Length);
    }
}
```

输出2-20展示了结果。

输出2-20

```
The wave of the future, COBOL, is at index 1.

First Element           Last Element
-------------           -------------
C#                      Visual Basic
Visual Basic            C#

After clearing, the array size is: 9
```

这些方法由System.Array类提供。大多数方法都是一目了然的。只需注意以下两点。

- 使用BinarySearch()方法前要对数组进行排序。如果值不按升序排序，会返回不正确的索引。搜索的元素不存在，会返回负值。使用取反操作符~index，会返回大于搜索值的第一个索引（如果有的话）。
- Clear()方法不删除数组元素，而且不将长度设为零。数组大小是固定的，不能修改。所以，Clear()方法将数组中的每个元素都设为其默认值（false、0或者null）。这解释了在调用Clear()之后输出数组时，Console.WriteLine()为什么会创建一个空行。

语言对比：Visual Basic——允许改变数组的维

Visual Basic提供了Redim语句来更改数组中的元素个数。虽然没有等价的C#关键字，但.NET 2.0提供了Resize()方法来重新创建数组，并将所有元素复制到新数组。这个方法称为System.Array.Resize。

3. 数组实例方法

类似于字符串，数组也有一些实例成员，它们不是从数据类型访问，而是直接从变量中访问。Length就是实例成员，因为是通过数组变量来访问Length，而不是通过类。其他常用实例成员还有GetLength()、Rank和Clone()。

要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。调用该方法时要指定返回哪一维的长度，如代码清单2-51所示。

代码清单2-51 获取特定维的大小

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

结果如输出2-21所示。

输出2-21

```
2
```

代码清单2-51输出2，这是第一维上的元素个数。

还可以访问数组的Rank成员来获取整个数组的维数。例如，cells.Rank返回3。

默认情况下，将一个数组变量赋值给另一个数组变量只会复制数组引用，而不是数组中单独的元素。要创建数组的全新副本，需使用数组的Clone()方法。该方法返回数组的一个副本，更改这个新数组中的任何成员都不会影响原始数组的成员。

2.7.4 字符串作为数组使用

访问string类型的变量类似于访问字符数组。例如，可调用palindrome[3]获取名为palindrome的string的第四个字符。但要注意，由于字符串是不可变的，所以不可能为字符串中的特定字符赋值。所以，将palindrome声明为string，palindrome[3]='a'这样的写法在C#中是不允许的。代码清单2-52使用数组访问符来判断命令行上的参数是不是选项（选项的第一个字符是-）。

代码清单2-52 查找命令行选项

```
string[] args;
...
if(args[0][0]=='-')
{
    //This parameter is an option
}
```

上述代码使用了if语句，该语句将在第3章讲述。除此之外，还使用第一个数组访问符获取字符串数组args的第一个元素，并用第二个数组访问符获取字符串的第一个字符。因此代码清单2-53所示代码与清单2-52所示代码是等价的。

代码清单2-53 查找命令行选项（简化版）

```
string[] args;
...
string arg = args[0];
if(arg[0] == '-')
{
    //This parameter is an option
}
```

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。获得字符数组后，可以使用System.Array.Reverse()方法反转数组中的元素，如代码清单2-54所示，该程序可以判断字符串是不是回文[①]。

代码清单2-54 反转字符串

```
class Palindrome
{
  static void Main()
  {
    string reverse, palindrome;
    char[] temp;

    System.Console.Write("Enter a palindrome: ");
    palindrome = System.Console.ReadLine();

    // Remove spaces and convert to lowercase
    reverse = palindrome.Replace(" ", "");
    reverse = reverse.ToLower();

    // Convert to an array
    temp = reverse.ToCharArray();

    // Reverse the array
    System.Array.Reverse(temp);

    // Convert the array back to a string and
    // check if reverse string is the same.
    if(reverse == new string(temp))
    {
        System.Console.WriteLine("\"{0}\" is a palindrome.",
             palindrome);
    }
    else
    {
        System.Console.WriteLine(
            "\"{0}\" is NOT a palindrome.",
            palindrome);
    }
  }
}
```

输出2-22展示了结果。

输出2-22

```
Enter a palindrome: NeverOddOrEven
"NeverOddOrEven" is a palindrome.
```

这个例子使用new关键字根据反转好的字符数组创建新字符串。

2.7.5 常见数组错误

前面描述了3种不同类型的数组：一维、多维和交错数组。一些规则和特性约束着数组

① palindrome，正反读都一样的词或句子。——译者注

的声明和使用。表2-8总结了一些最常见的错误，有助于巩固对这些规则的了解。阅读时最好先研究一下“常见错误”栏中的代码（先不要看错误描述和改正后的代码），看看自己是否能发现其中的错误，从而检查你对数组及其语法的理解。

表2-8 常见数组编码错误

常见错误	错误描述	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[];`	除非提供数组字面量，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，但数组字面量中没有任何元素。数组的大小必须与数组字面量中的元素个数相符	`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数组进行实例化，所以它暂时是不可访问的	`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} };`
`int[][] numbers =` `  { {42, 84},` `  {84, 42} };`	交错数组要求对数组中的数组进行实例化	`int[][] numbers =` `  { new int[]{42, 84},` `  new int[]{84, 42} };`

2.8 小结

即使是有经验的程序员，也要注意C#引入的几个新的编程构造。例如，本章探讨了decimal类型，它可以用于精确的金融计算。此外，本章还提到，布尔类型bool不会隐式转换成整数，从而防止了在条件表达式中错误地使用赋值操作符。C#与众不同的其他地方是使用@作为字符串逐字前缀字符，它强迫字符串忽略转义字符。除此之外，C#中的string数据类型是不可变的。

为了实现数据类型之间的相互转换，C#引入了显式和隐式的转型操作符。在后续章节，将学习如何为自定义类型定义这两种类型的操作符。

本章最后讨论了数组的C#语法，并探讨了各种不同的数组处理方式。对于许多开发者来说，掌握这些语法并不是一件容易的事情。所以我们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。本章最后出现的if语句将在下一章全面讨论。