7.9 常量与静态量

然而，成员常量比局部常量更有趣，因为它们表现得像静态值。它们对类的每个实例都是“可见的”，而且即使没有类的实例也可以使用。与真正的静态量不同，常量没有自己的存储位置，而是在编译时被编译器替换。这种方式类似于 C 和 C++中的#define 值。

例如，下面的代码声明了类 X，带有常量字段 PI。Main 没有创建 X 的任何实例，但仍然可以使用字段 PI 并打印它的值。图 7-6 阐明了这段代码。

```
class X
{
   public const double PI = 3.1416;
}

class Program
{
   static void Main()
   {
      Console.WriteLine($"pi = { X.PI }");     //使用常量字段 PI
   }
}
```

这段代码产生以下输出：

```
pi = 3.1416
```

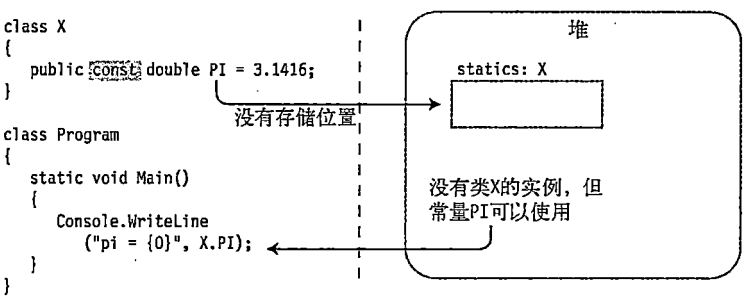

图 7-6 常量字段表现得像静态字段，但是在内存中没有存储位置

虽然常量成员表现得像静态值，但不能将常量声明为 static，如下面的代码所示：

```
static const double PI = 3.14;      //错误：不能将常量声明为 static
```

7.10 属性

属性是代表类实例或类中的数据项的成员。使用属性就像写入或读取一个字段，语法相同。例如，下面的代码展示了名为 MyClass 的类的使用，它有一个公有字段和一个公有属性。从

用法上无法区分它们。

```
MyClass mc = new MyClass();

mc.MyField    = 5;                                    //给字段赋值
mc.MyProperty = 10;                                   //给属性赋值

Console.WriteLine($"{ mc.MyField } { mc.MyProperty }"); //读取字段和属性
```

与字段类似，属性有如下特征。

- 它是命名的类成员。
- 它有类型。
- 它可以被赋值和读取。

然而和字段不同，属性是一个函数成员。

- 它不一定为数据存储分配内存！
- 它执行代码。

属性是一组（两个）匹配的、命名的、称为访问器的方法。

- set 访问器为属性赋值。
- get 访问器从属性获取值。

图 7-7 展示了属性的表示法。左边的代码展示了声明一个名为 MyValue 的 int 类型属性的语法，右边的图像展示了属性在文本中可视化的方式。请注意，访问器从后面伸出，因为它们不能被直接调用。这一点你很快会看到。

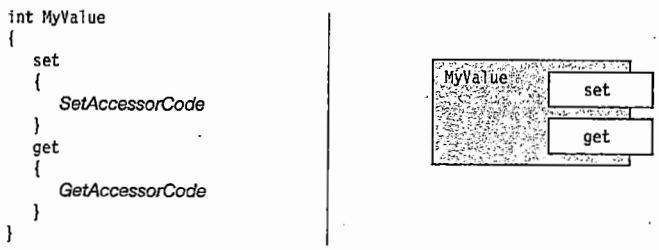

图 7-7　int 类型、名称为 MyValuer 的属性示例

7.10.1　属性声明和访问器

set 和 get 访问器有预定义的语法和语义。可以把 set 访问器想象成一个方法，带有单一的参数，它“设置”属性的值。get 访问器没有参数并从属性返回一个值。

- set 访问器总是：
  - 拥有一个单独的、隐式的值参，名称为 value，与属性的类型相同；
  - 拥有一个返回类型 void。
- get 访问器总是：
  - 没有参数；

■ 拥有一个与属性类型相同的返回类型。

属性声明的结构如图 7-8 所示。注意，图中的访问器声明既没有显式的参数，也没有返回类型声明。不需要它们，因为它们已经隐含在属性的类型中了。

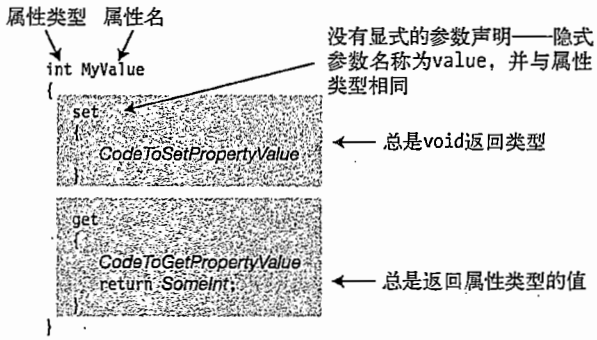

图 7-8 属性声明的语法和结构

set 访问器中的隐式参数 value 是一个普通的值参。和其他值参一样，可以用它发送数据到方法体或访问器块。在块的内部，可以像普通变量那样使用 value，包括对它赋值。

访问器的其他要点如下。

- get 访问器的所有执行路径必须包含一条 return 语句，它返回一个属性类型的值。
- 访问器 set 和 get 可以以任何顺序声明，并且，除了这两个访问器外，属性上不允许有其他方法。

7.10.2 属性示例

下面的代码展示了一个名为 C1 的类的声明示例，它含有一个名为 MyValue 的属性。

- 请注意，属性本身没有任何存储。取而代之，访问器决定如何处理发送进来的数据，以及应将什么数据发送出去。在这种情况下，属性使用一个名为 TheRealValue 的字段作为存储。
- set 访问器接受它的输入参数 value，并把它的值赋给字段 TheRealValue。
- get 访问器只是返回字段 TheRealValue 的值。

图 7-9 说明了这段代码。

```
class C1
{
   private int theRealValue;                    //字段：分配内存

   public int MyValue                           //属性：未分配内存
   {
      set { theRealValue = value; }
      get { return theRealValue; }
   }
}
```

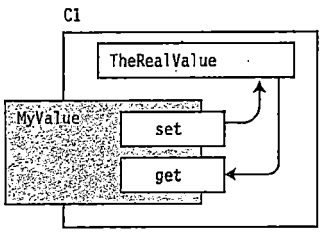

图 7-9　属性访问器常常使用字段作为存储

7.10.3　使用属性

就像之前看到的，写入和读取属性的方法与访问字段一样。访问器被隐式调用。

- 要写入一个属性，在赋值语句的左边使用属性的名称。
- 要读取一个属性，把属性的名称用在表达式中。

例如，下面的代码包含一个名为 MyValue 的属性的声明。只需使用属性名就可以写入和读取属性，就好像它是一个字段名。

```
int MyValue              //属性声明
{
   set{ ... }
   get{ ... }
}
...
属性名称
   ↓
MyValue = 5;             //赋值：隐式调用 set 方法
z = MyValue;             //表达式：隐式调用 get 方法
      ↑
   属性名称
```

属性会根据是写入还是读取来隐式地调用适当的访问器。不能显式地调用访问器，因为这样做会产生编译错误。

```
y = MyValue.get();       //错误！不能显式调用 get 访问器
MyValue.set(5);          //错误！不能显式调用 set 访问器
```

7.10.4　属性和关联字段

属性常和字段关联，这一点我们在前两节已经看到了。一种常见的方式是在类中将字段声明为 private 以封装该字段，并声明一个 public 属性来控制从类的外部对该字段的访问。和属性关联的字段常被称为后备字段或后备存储。

例如，下面的代码使用公有属性 MyValue 来控制对私有字段 TheRealValue 的访问。

```
class C1
{
   private int theRealValue = 10;   //后备字段：分配内存
```

```
    public  int MyValue                 //属性：不分配内存
    {
        set{ theRealValue = value; }  //设置 TheRealValue 字段的值
        get{ return theRealValue; }   //获取字段的值
    }
}

class Program
{
    static void Main()
    {
                                    把属性看作一个字段，从中读取它的值
        C1 c = new C1();                          ↓
        Console.WriteLine("MyValue:  {0}", c.MyValue);

        c.MyValue = 20;          ← 使用赋值语句设置属性的值
        Console.WriteLine("MyValue:  {0}", c.MyValue);
    }
}
```

属性和它们的后备字段有几种命名约定。一种约定是两个名称使用相同的内容，但字段使用 Camel 大小写，属性使用 Pascal 大小写。（在 Camel 大小写风格中，复合词标识符中每个单词的首字母大写——除了第一个单词，其余字母都是小写。在 Pascal 大小写风格中，复合词中每个单词的首字母都是大写。）虽然这违反了“仅使用大小写区分不同标识符是个坏习惯”这条一般规则，但它有个好处，即可以把两个标识符以一种有意义的方式联系在一起。

另一种约定是属性使用 Pascal 大小写，字段使用相同标识符的 Camel 大小写版本，并以下划线开始。

下面的代码展示了两种约定：

```
private int firstField;                 //Camel 大小写
public  int FirstField                  //Pascal 大小写
{
    get { return firstField; }
    set { firstField = value; }
}

private int _secondField;               //下划线及 Camel 大小写
public  int SecondField
{
    get { return _secondField; }
    set { _secondField = value; }
}
```

7.10.5 执行其他计算

属性访问器并不局限于对关联的后备字段传进传出数据。访问器 get 和 set 能执行任何计算，也可以不执行任何计算。唯一必需的行为是 get 访问器要返回一个属性类型的值。

例如，下面的示例展示了一个有效的（但可能没有用处的）属性，它仅在 get 访问器被调用

时返回值5。当set访问器被调用时，它什么也不做。隐式参数value的值被忽略了。

```
public int Useless
{
   set{ }              //什么也不设置
   get{ return 5; }    //只是返回值5
}
```

下面的代码展示了一个更现实和有用的属性，其中set访问器在设置关联字段之前实现过滤。set访问器把字段TheRealValue的值设置成输入值，如果输入值大于100，就将TheRealValue设置为100。

```
int theRealValue = 10;                                    //字段
int MyValue                                               //属性
{
   set { theRealValue = value > 100 ? 100 : value; }   //条件运算符
   get { return theRealValue; }
}
```

C# 7.0为属性的getter/setter引入了另一种语法，这种语法使用表达函数体。虽然第14章会详细讨论表达函数体（或者叫lambda表达式），但是为了完整性，这里演示了这种新的语法。这种语法只有在访问函数体由一个表达式组成的时候才能使用。

```
int MyValue
{
   set => value > 100 ? 100 : value;
   get => theRealValue;
}
```

说明　在上面的代码示例中，从等号到语句结尾部分的语法叫作条件运算符，第9章会详细阐述。条件运算符是一种三元运算符，计算问号之前的表达式，如果表达式计算结果为true，那么返回问号后的第一个表达式，否则返回冒号之后的表达式。有些人可能会使用if...then语句，不过条件运算符更合适，我们将在第9章介绍这两种构造的细节。

7.10.6　只读和只写属性

要想不定义属性的某个访问器，可以忽略该访问器的声明。

- 只有get访问器的属性称为只读属性。只读属性能够安全地将一个数据项从类或类的实例中传出，而不必让调用者修改属性值。
- 只有set访问器的属性称为只写属性。只写属性很少见，因为它们几乎没有实际用途。如果想在赋值时触发一个副作用，应该使用方法而不是属性。
- 两个访问器中至少有一个必须定义，否则编译器会产生一条错误消息。

图7-10阐述了只读和只写属性。

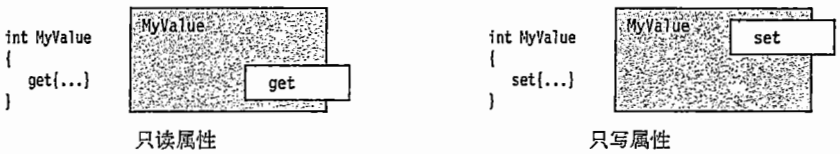

图 7-10 属性可以只定义一个访问器

7.10.7 属性与公有字段

按照推荐的编码实践，属性比公有字段更好，理由如下。

- 属性是函数成员而不是数据成员，允许你处理输入和输出，而公有字段不行。
- 属性可以只读或只写，而字段不行。
- 编译后的变量和编译后的属性语义不同。

如果要发布一个由其他代码引用的程序集，那么第三点将会带来一些影响。例如，有的时候开发人员可能想用公有字段代替属性，因为如果以后需要为字段的数据增加处理逻辑的话，可以再把字段改为属性。这没错，但是如果那样修改的话，所有访问这个字段的其他程序集都需要重新编译，因为字段和属性在编译后的语义不一样。另外，如果实现的是属性，那么只需要修改属性的实现，而无须重新编译访问它的其他程序集。

7.10.8 计算只读属性示例

迄今为止，在大多示例中，属性都和一个后备字段关联，并且 get 和 set 访问器引用该字段。然而，属性并非必须和字段关联。在下面的示例中，get 访问器计算出返回值。

在下面的示例代码中，类 RightTriangle 表示一个直角三角形。图 7-11 阐释了只读属性 Hypotenuse。

- 它有两个公有字段，分别表示直角三角形的两条直角边的长度。这两个字段可以被写入和读取。
- 第三条边由属性 Hypotenuse 表示，它是一个只读属性，其返回值基于另外两条边的长度。它没有存储在字段中。相反，它在需要时根据当前 A 和 B 的值计算正确的值。

```
class RightTriangle
{
   public double A = 3;
   public double B = 4;
   public double Hypotenuse                    //只读属性
   {
      get{ return Math.Sqrt((A*A)+(B*B)); }    //计算返回值
   }
}

class Program
{
```

```
    static void Main()
    {
        RightTriangle c = new RightTriangle();
        Console.WriteLine($"Hypotenuse: { c.Hypotenuse }");
    }
}
```

这段代码产生以下输出：

```
Hypotenuse:  5
```

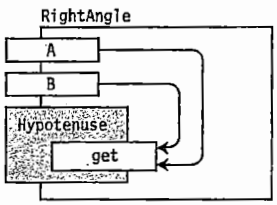

图 7-11　只读属性 Hypotenuse

7.10.9　自动实现属性

因为属性经常被关联到后备字段，所以 C# 提供了自动实现属性（automatically implemented property 或 auto-implemented property，常简称为“自动属性”，auto-property），允许只声明属性而不声明后备字段。编译器会为你创建隐藏的后备字段，并且自动挂接到 get 和 set 访问器上。

自动实现属性的要点如下。

- 不声明后备字段——编译器根据属性的类型分配存储。
- 不能提供访问器的方法体——它们必须被简单地声明为分号。get 担当简单的内存读，set 担当简单的写。但是，因为无法访问自动属性的方法体，所以在使用自动属性时调试代码通常会更加困难。

  从 C# 6.0 开始，可以使用只读自动属性了。此外，还可以将自动属性初始化作为其声明的一部分。

下面的代码展示了一个自动实现属性的示例。

```
class C1
{                    ←没有声明后备字段
    public int MyValue                          //分配内存
    {
        set; get;
    }     ↑    ↑
}  访问器的方法体被声明为分号

class Program
{
    static void Main()
```

```
    {                                    像使用规则属性那样使用自动属性
        C1 c = new C1();                          ↓
        Console.WriteLine("MyValue:  {0}", c.MyValue);

        c.MyValue = 20;
        Console.WriteLine("MyValue:  {0}", c.MyValue);
    }

}
```

这段代码产生以下输出：

```
MyValue:  0
MyValue:  20
```

除了方便以外，利用自动实现属性还能在想声明一个公有字段的地方轻松地插入一个属性。

7.10.10 静态属性

属性也可以声明为 static。静态属性的访问器和所有静态成员一样，具有以下特点。

- 不能访问类的实例成员，但能被实例成员访问。
- 不管类是否有实例，它们都是存在的。
- 在类的内部，可以仅使用名称来引用静态属性。
- 在类的外部，正如本章前面描述的，可以通过类名或者使用 using static 结构来引用静态属性。

例如，下面的代码展示了一个类，它带有名为 MyValue 的自动实现的静态属性。在 Main 的头三行，即使没有类的实例，也能访问属性。Main 的最后一行调用一个实例方法，它从类的内部访问属性。

```
using System;
using static ConsoleTestApp.Trivial;
namespace ConsoleTestApp
{
    class Trivial {
        public static int MyValue { get; set; }     从类的内部访问
        public void PrintValue()                          ↓
        { Console.WriteLine("Value from inside: {0}", MyValue); }
    }

    class Program {                                 从类的外部访问
        static void Main() {                              ↓
            Console.WriteLine("Init Value: {0}", Trivial.MyValue);
            Trivial.MyValue = 10;                ← 从类的外部访问
            Console.WriteLine("New Value : {0}", Trivial.MyValue);

            MyValue = 20; ← 从类的外部访问，但由于使用了 using static，所以没有使用类名
            Console.WriteLine($"New Value : { MyValue }");
```

```
            Trivial tr = new Trivial();
            tr.PrintValue();
        }
    }

}
```

```
Init Value: 0
New Value : 10
New Value : 20
Value from inside: 20
```

7.11　实例构造函数

实例构造函数[①]是一个特殊的方法，它在创建类的每个新实例时执行。

- 构造函数用于初始化类实例的状态。
- 如果希望能从类的外部创建类的实例，需要将构造函数声明为 public。

图 7-12 阐述了构造函数的语法。除了下面这几点，构造函数看起来很像类声明中的其他方法。

- 构造函数的名称和类名相同。
- 构造函数不能有返回值。

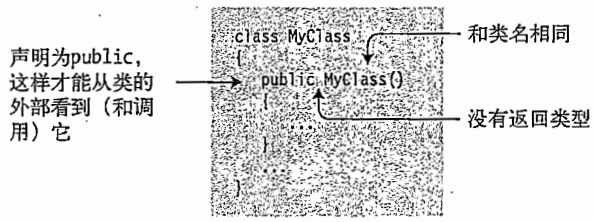

图 7-12　构造函数声明

例如，下面的类使用构造函数初始化其字段。本例中，它有一个名为 TimeOfInstantiation 的字段被初始化为当前的日期和时间。

```
class MyClass
{
   DateTime TimeOfInstantiation;                    //字段
   ...
   public MyClass()                                 //构造函数
   {
      TimeOfInstantiation = DateTime.Now;           //初始化字段
   }
   ...
}
```

① “constructor”一词在微软官方文档中也常译为“构造器”。——编者注