# 1.他们的类型不同

## 类是一种引用类型,把一个类对象变量赋值给另外一个类对象变量,这两个变量指向的是同一个内存地址,修改任意一个变量的属性,在另外一个变量上也会体现出来.类可以被继承,类也可以继承自接口.

## 结构体是一种值类型,把一个结构体对象的值赋值给另外一个结构体变量,他们的值是一样的,但是他们是内存是不一样的.修改一个变量的属性并不会影响另外一个变量的属性值.结构体不能被继承,但是结构体可以继承自接口.

# 2.类的3大特性:封装,继承,多态

## 1>封装, 可将无意在类或程序集外部使用的方法和变量隐藏起来，以减小编码错误或遭恶意利用的可能性。

### 成员

所有方法、字段、常量、属性和事件都必须在类型内部进行声明；这些称为类型的“成员”。与其他一些语言不同的是，C# 中没有全局变量或方法。即使是作为程序入口点的 Main 方法也必须在类或结构内部进行声明。下表列出了可在类或结构中声明的所有不同种类的成员。

* [字段](https://msdn.microsoft.com/zh-CN/library/ms173118.aspx)
* [常量](https://msdn.microsoft.com/zh-CN/library/ms173119.aspx)
* [属性](https://msdn.microsoft.com/zh-CN/library/x9fsa0sw.aspx)
* [方法](https://msdn.microsoft.com/zh-CN/library/ms173114.aspx)
* [构造函数](https://msdn.microsoft.com/zh-CN/library/ace5hbzh.aspx)
* [析构函数](https://msdn.microsoft.com/zh-CN/library/66x5fx1b.aspx)
* [事件](https://msdn.microsoft.com/zh-CN/library/awbftdfh.aspx)
* [索引器](https://msdn.microsoft.com/zh-CN/library/6x16t2tx.aspx)
* [运算符](https://msdn.microsoft.com/zh-CN/library/ms173145.aspx)
* [嵌套类型](https://msdn.microsoft.com/zh-CN/library/ms173120.aspx)

### 访问修饰符

使用访问修饰符 [public](https://msdn.microsoft.com/zh-CN/library/yzh058ae.aspx)、[protected](https://msdn.microsoft.com/zh-CN/library/bcd5672a.aspx)、[internal](https://msdn.microsoft.com/zh-CN/library/7c5ka91b.aspx)、**protected internal** 和 [private](https://msdn.microsoft.com/zh-CN/library/st6sy9xe.aspx) 可以指定类型及其成员对于客户端代码的可访问性。默认可访问性为 **private**。

**作用范围**

| **访问修饰符** | **说明** |
| --- | --- |
| public | 公有访问，不受任何限制。 |
| private | 私有访问，只限于本类成员访问，子类、实例都不能访问。 |
| protected | 保护访问，只限于本类和子类访问，实例不能访问。 |
| internal | 内部访问，只限于本项目内访问，其他不能访问。 |
| protected internal | 内部保护访问，只限于本项目或者子类访问，其他不能访问。 |

**补充**

1. protected internal 在现版本的编程使用较少，它是将保护访问与内部访问结合起来形成的一种内部保护访问。
2. 直接在命名空间中声明的类和结构（即：没有嵌套在其他类或结构中的类和结构）可以是公共类和结构，也可以是内部类和结构。如果不指定访问修饰符，则默认为 internal。
3. 派生类的可访问性不能高于其基类型。换句话说，不能有从内部类A派生的公共类B。 如果允许这种情况，将会使A成为公共类，因为A的所有受保护的成员或内部成员都可以从派生类访问。
4. 结构成员，包括嵌套的类和结构，可以声明为公共的、 内部的，或私有的。 类成员（包括嵌套的类和结构）可以为公共的、受保护的内部、受保护的、内部的或私有的。 类成员和结构成员的访问级别，包括嵌套类和结构，默认为私有。 不可以从包含类型之外访问私有嵌套类型。

**C#中方法、类等的默认访问修饰符**

1. 接口(interface)  
   接口成员访问修饰符默认为public,且不能显示使用访问修饰符。
2. 类(class)  
   构造函数默认为public访问修饰符。析构函数不能显示使用访问修饰符且默认为private访问修饰符。类的成员默认访问修饰符为private。
3. 枚举(enum)  
   枚举类型成员默认为public访问修饰符，且不能显示使用修饰符。
4. 结构(struct)  
   结构成员默认为private修饰符。结构成员无法声明为protected成员，因为结构不支持继承。
5. 嵌套类型  
   嵌套类型的默认访问修饰符为private。 和类，结构的成员默认访问类型一致。

**成员类型的可修饰，及其默认（即如果不指定）修饰符**

| **成员类型** | **默认修饰符** | **可被修饰符** |
| --- | --- | --- |
| enum | public | none |
| class | private | public、protected、internal、private、protected internal |
| interface | public | none |
| struct | private | public、internal、private |

**接口成员、枚举成员、 委托行为**

1. 接口成员始终是公共成员，因为接口的用途是让其他类型能够访问某个类或结构。访问修饰符不能应用于接口成员。
2. 枚举成员始终是公共的，不能应用任何访问修饰符。

委托行为类似于类和结构。默认情况下，它们在命名空间中直接声明时具有内部访问权，在嵌套时具有私有访问权

## 2>继承,和c++不一样,c#只支持单继承

### 类（而非结构）支持继承的概念。派生自另一个类（“基类”）的类将自动包含基类除构造函数和析构函数之外的所有公共、受保护和内部成员。

### 可以将类声明为[抽象](https://msdn.microsoft.com/zh-CN/library/sf985hc5.aspx)类，表示该类的一个或多个方法不具有实现。抽象类虽然无法直接实例化，但可以用作其他类的基类，由其他类提供缺少的实现。还可以将类声明为[密封](https://msdn.microsoft.com/zh-CN/library/88c54tsw.aspx)类，以禁止其他类从该类继承。

### 抽象类可以包含一个或多个本身声明为抽象的方法签名。 这些签名指定参数和返回值，但没有任何实现（方法体）。 抽象类不必包含抽象成员；但是，如果类包含抽象成员，则类本身必须声明为抽象。 本身不抽象的派生类必须为来自抽象基类的任何抽象方法提供实现。

### 抽象方法和虚方法

基类将方法声明为 [virtual](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/virtual) 时，派生类可以使用其自己的实现[override](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/override)该方法。 如果基类将成员声明为 [abstract](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/abstract)，则必须在直接继承自该类的任何非抽象类中重写该方法。 如果派生类本身是抽象的，则它会继承抽象成员而不会实现它们。 抽象和虚拟成员是多形性（面向对象的编程的第二个主要特征）的基础。

#### 类继承实例,

##### 注意c#中有一种构造函数叫做静态构造函数,用static修饰,它用来初始化类的静态成员变量

|  |
| --- |
| public class WorkItem  {  // Static field currentID stores the job ID of the last WorkItem that  // has been created.  private static int currentID;  //Properties.  protected int ID { get; set; }  protected string Title { get; set; }  protected string Description { get; set; }  protected TimeSpan jobLength { get; set; }  // Default constructor. If a derived class does not invoke a base-  // class constructor explicitly, the default constructor is called  // implicitly.  public WorkItem()  {  ID = 0;  Title = "Default title";  Description = "Default description.";  jobLength = new TimeSpan();  }  // Instance constructor that has three parameters.  public WorkItem(string title, string desc, TimeSpan joblen)  {  this.ID = GetNextID();  this.Title = title;  this.Description = desc;  this.jobLength = joblen;  }  // Static constructor to initialize the static member, currentID. This  // constructor is called one time, automatically, before any instance  // of WorkItem or ChangeRequest is created, or currentID is referenced.  static WorkItem() => currentID = 0; //静态构造函数,用来初始化静态成员变量.这种写法是lambda表达式,也叫做箭头函数  // currentID is a static field. It is incremented each time a new  // instance of WorkItem is created.  protected int GetNextID() => ++currentID;  // Method Update enables you to update the title and job length of an  // existing WorkItem object.  public void Update(string title, TimeSpan joblen)  {  this.Title = title;  this.jobLength = joblen;  }  // Virtual method override of the ToString method that is inherited  // from System.Object.  public override string ToString() =>  $"{this.ID} - {this.Title}";  }  // ChangeRequest derives from WorkItem and adds a property (originalItemID)  // and two constructors.  public class ChangeRequest : WorkItem  {  protected int originalItemID { get; set; }  // Constructors. Because neither constructor calls a base-class  // constructor explicitly, the default constructor in the base class  // is called implicitly. The base class must contain a default  // constructor.  // Default constructor for the derived class.  public ChangeRequest() { }  // Instance constructor that has four parameters.  public ChangeRequest(string title, string desc, TimeSpan jobLen,  int originalID)  {  // The following properties and the GetNexID method are inherited  // from WorkItem.  this.ID = GetNextID();  this.Title = title;  this.Description = desc;  this.jobLength = jobLen;  // Property originalItemID is a member of ChangeRequest, but not  // of WorkItem.  this.originalItemID = originalID;  }  } |

##### 类的使用

|  |
| --- |
| // Create an instance of WorkItem by using the constructor in the  // base class that takes three arguments.  WorkItem item = new WorkItem("Fix Bugs",  "Fix all bugs in my code branch",  new TimeSpan(3, 4, 0, 0));  // Create an instance of ChangeRequest by using the constructor in  // the derived class that takes four arguments.  ChangeRequest change = new ChangeRequest("Change Base Class Design",  "Add members to the class",  new TimeSpan(4, 0, 0),  1);  // Use the ToString method defined in WorkItem.  Console.WriteLine(item.ToString());  // Use the inherited Update method to change the title of the  // ChangeRequest object.  change.Update("Change the Design of the Base Class",  new TimeSpan(4, 0, 0));  // ChangeRequest inherits WorkItem's override of ToString.  Console.WriteLine(change.ToString());  /\* Output:  1 - Fix Bugs  2 - Change the Design of the Base Class  \*/ |

### 接口

接口是定义一组成员的引用类型。 实现该接口的所有类和结构都必须实现这组成员。 接口可以为其中任何成员或全部成员定义默认实现。 类可以实现多个接口，即使它只能派生自单个直接基类。

接口用于为类定义特定功能，这些功能不一定具有“is a (是)”关系。 例如，[System.IEquatable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.iequatable-1) 接口可由任何类或结构实现，以确定该类型的两个对象是否等效（但是由该类型定义等效性）。 [IEquatable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.iequatable-1) 不表示基类和派生类之间存在的同一种“是”关系（例如，Mammal 是 Animal）

### 泛型类型

可以使用一个或多个类型参数来定义类和结构。客户端代码在创建类型的实例时提供类型。例如，[System.Collections.Generic](https://msdn.microsoft.com/zh-CN/library/system.collections.generic.aspx) 命名空间中的 [List<T>](https://msdn.microsoft.com/zh-CN/library/6sh2ey19.aspx) 类使用一个类型参数进行定义。客户端代码创建 List<string> 或 List<int> 的实例来指定列表中将包含的类型。

### 静态类型

可以将类（不是结构）声明为[静态](https://msdn.microsoft.com/zh-CN/library/98f28cdx.aspx)。静态类只能包含静态成员，不能使用 new 关键字进行实例化。在程序加载时，静态类的一个副本将加载到内存中，可通过类名称访问该类的成员。类和结构都可以包含静态成员。

### 嵌套类型,也就是内部类

在[类](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/class)、[构造](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct)或[接口](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/interface)中定义的类型称为嵌套类型。 例如

C#

public class Container

{

class Nested

{

Nested() { }

}

}

不论外部类型是类、接口还是构造，嵌套类型均默认为 [private](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private)；仅可从其包含类型中进行访问。 在上一个示例中，Nested 类无法访问外部类型。

还可指定[访问修饰符](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/access-modifiers)来定义嵌套类型的可访问性，如下所示：

* “类”的嵌套类型可以是 [public](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/public)、[protected](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/protected)、[internal](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/internal)、[protected internal](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/protected-internal)、[private](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private) 或 [private protected](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private-protected)。

但是，在[密封类](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/sealed)中定义 protected、protected internal 或 private protected 嵌套类将产生编译器警告 [CS0628](https://learn.microsoft.com/zh-cn/dotnet/csharp/misc/cs0628)“封闭类汇中声明了新的受保护成员”。

另请注意，使嵌套类型在外部可见违反了代码质量规则 [CA1034](https://learn.microsoft.com/zh-cn/dotnet/fundamentals/code-analysis/quality-rules/ca1034)“嵌套类型不应是可见的”。

* 构造的嵌套类型可以是 [public](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/public)、[internal](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/internal) 或 [private](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private)。

以下示例使 Nested 类为 public：

C#

public class Container

{

public class Nested

{

Nested() { }

}

}

嵌套类型（或内部类型）可访问包含类型（或外部类型）。 若要访问包含类型，请将其作为参数传递给嵌套类型的构造函数。 例如：

C#

public class Container

{

public class Nested

{

private Container? parent;

public Nested()

{

}

public Nested(Container parent)

{

this.parent = parent;

}

}

}

嵌套类型可以访问其包含类型可以访问的所有成员。 它可以访问包含类型的私有成员和受保护成员（包括所有继承的受保护成员）。

在前面的声明中，类 Nested 的完整名称为 Container.Nested。 这是用来创建嵌套类新实例的名称，如下所示：

C#

Container.Nested nest = new Container.Nested();

### 分部类型

可以在一个代码文件中定义类、结构或方法的一部分，而在另一个代码文件中定义另一部分.

#### 在以下几种情况下需要拆分类定义：

* 通过单独的文件声明某个类可以让多位程序员同时对该类进行处理。
* 你可以向该类中添加代码，而不必重新创建包括自动生成的源代码的源文件。 Visual Studio 在创建Windows 窗体、Web 服务包装器代码等时会使用这种方法。 你可以创建使用这些类的代码，这样就不需要修改由Visual Studio生成的文件。
* [源代码生成器](https://learn.microsoft.com/zh-cn/dotnet/csharp/roslyn-sdk/source-generators-overview)可以在类中生成额外的功能。

若要拆分类定义，请使用 [partial](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/partial-type) 关键字修饰符，如下所示：

C#

public partial class Employee

{

public void DoWork()

{

}

}

public partial class Employee

{

public void GoToLunch()

{

}

}

#### partial 关键字指示可在命名空间中定义该类、结构或接口的其他部分。 所有部分都必须使用 partial 关键字。

在编译时，各个部分都必须可用来形成最终的类型。 各个部分必须具有相同的可访问性，如 public、private 等。

如果将任意部分声明为抽象的，则整个类型都被视为抽象的。 如果将任意部分声明为密封的，则整个类型都被视为密封的。 如果任意部分声明基类型，则整个类型都将继承该类。

指定基类的所有部分必须一致，但忽略基类的部分仍继承该基类型。 各个部分可以指定不同的基接口，最终类型将实现所有分部声明所列出的全部接口。 在某一分部定义中声明的任何类、结构或接口成员可供所有其他部分使用。 最终类型是所有部分在编译时的组合。

#### 注意:

##### partial 修饰符不可用于委托或枚举声明中。

下面的示例演示嵌套类型可以是分部的，即使它们所嵌套于的类型本身并不是分部的也如此。

C#复制

class Container

{

partial class Nested

{

void Test() { }

}

partial class Nested

{

void Test2() { }

}

}

编译时会对分部类型定义的属性进行合并。 以下面的声明为例：

C#复制

[SerializableAttribute]

partial class Moon { }

[ObsoleteAttribute]

partial class Moon { }

它们等效于以下声明：

C#复制

[SerializableAttribute]

[ObsoleteAttribute]

class Moon { }

将从所有分部类型定义中对以下内容进行合并：

* XML 注释。 但是，如果分部成员的两个声明都包含注释，则仅包括实现成员的注释。
* interfaces
* 泛型类型参数属性
* class 特性
* 成员

以下面的声明为例：

C#复制

partial class Earth : Planet, IRotate { }

partial class Earth : IRevolve { }

它们等效于以下声明：

C#复制

class Earth : Planet, IRotate, IRevolve { }

#### 限制

处理分部类定义时需遵循下面的几个规则：

* 要作为同一类型的各个部分的所有分部类型定义都必须使用 partial 进行修饰。 例如，下面的类声明会生成错误：

C#复制

public partial class A { }

//public class A { } // Error, must also be marked partial

* partial 修饰符只能出现在紧靠关键字 class、struct 或 interface 前面的位置。
* 分部类型定义中允许使用嵌套的分部类型，如下面的示例中所示：

C#复制

partial class ClassWithNestedClass

{

partial class NestedClass { }

}

partial class ClassWithNestedClass

{

partial class NestedClass { }

}

* 要成为同一类型的各个部分的所有分部类型定义都必须在同一程序集和同一模块（.exe 或 .dll 文件）中进行定义。 分部定义不能跨越多个模块。
* 类名和泛型类型参数在所有的分部类型定义中都必须匹配。 泛型类型可以是分部的。 每个分部声明都必须以相同的顺序使用相同的参数名。
* 下面用于分部类型定义中的关键字是可选的，但是如果某关键字出现在一个分部类型定义中，则必须在相同类型的其他分部定义中指定相同的关键字：
  + [公共](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/public)
  + [private](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private)
  + [受保护](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/protected)
  + [internal](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/internal)
  + [abstract](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/abstract)
  + [sealed](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/sealed)
  + 基类
  + [new](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/new-modifier) 修饰符（嵌套部分）
  + 泛型约束

#### 示例

下面的示例在一个分部类定义中声明 Coords 类的字段和构造函数，在另一个分部类定义中声明成员 PrintCoords。

C#复制

public partial class Coords

{

private int x;

private int y;

public Coords(int x, int y)

{

this.x = x;

this.y = y;

}

}

public partial class Coords

{

public void PrintCoords()

{

Console.WriteLine("Coords: {0},{1}", x, y);

}

}

class TestCoords

{

static void Main()

{

Coords myCoords = new Coords(10, 15);

myCoords.PrintCoords();

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

// Output: Coords: 10,15

从下面的示例可以看出，你也可以开发分部结构和接口。

C#复制

partial interface ITest

{

void Interface\_Test();

}

partial interface ITest

{

void Interface\_Test2();

}

partial struct S1

{

void Struct\_Test() { }

}

partial struct S1

{

void Struct\_Test2() { }

}

#### 分部成员

分部类或结构可以包含分部成员。 类的一个部分包含成员的签名。 可以在同一部分或另一部分中定义实现。

当签名遵循以下规则时，分部方法不需要实现：

* 声明未包含任何访问修饰符。 默认情况下，该方法具有 [private](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private) 访问权限。
* 返回类型为 [void](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/void)。
* 没有任何参数具有 [out](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/method-parameters#out-parameter-modifier) 修饰符。
* 方法声明不能包括以下任何修饰符：
  + [virtual](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/virtual)
  + [override](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/override)
  + [sealed](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/sealed)
  + [new](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/new-modifier)
  + [extern](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/extern)

当未提供实现时，在编译时会移除该方法以及对该方法的所有调用。

任何不符合所有这些限制的方法（包括属性和索引器）都必须提供实现。 此实现可以由源生成器提供。 [分部属性](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/partial-member)不能使用自动实现的属性来实现。 编译器无法区分自动实现的属性和分部属性的声明声明。

分部方法允许类的某个部分的实现者声明成员。 类的另一部分的实现者可以定义该成员。 在以下两个情形中，此分离很有用：生成样板代码的模板和源生成器。

* **模板代码**：模板保留方法名称和签名，以使生成的代码可以调用方法。 这些方法遵循允许开发人员决定是否实现方法的限制。 如果未实现该方法，编译器会移除方法签名以及对该方法的所有调用。 调用该方法（包括调用中的任何参数计算结果）在运行时没有任何影响。 因此，分部类中的任何代码都可以随意地使用分部方法，即使未提供实现也是如此。 未实现该方法时调用该方法不会导致编译时错误或运行时错误。
* **源生成器**：源生成器提供成员的实现。 开发人员可以添加成员声明（通常由源生成器读取属性）。 开发人员可以编写调用这些成员的代码。 源生成器在编译过程中运行并提供实现。 在这种情况下，不会遵循不经常实现的分部成员的限制。

C#复制

// Definition in file1.cs

partial void OnNameChanged();

// Implementation in file2.cs

partial void OnNameChanged()

{

// method body

}

* 分部成员声明必须以上下文关键字 [partial](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/partial-type) 开头。
* 分部类型的两个部分中的分部成员签名必须匹配。
* 分部成员可以有 [static](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/static) 和 [unsafe](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/unsafe) 修饰符。
* 分部成员可能是泛型成员。 约束在定义和实现方法声明时必须相同。 参数和类型参数名称在定义和实现方法声明时不必相同。
* 你可以为已定义并实现的分部方法生成[委托](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/reference-types)，但不能为没有实现的分部方法生成委托。

### 对象初始值设定项

#### 可以实例化和初始化类或结构对象以及对象的集合，无需显式调用其构造函数

使用对象初始值设定项，你可以在创建对象时向对象的任何可访问字段或属性分配值，而无需调用后跟赋值语句行的构造函数。 利用对象初始值设定项语法，你可为构造函数指定参数或忽略参数（以及括号语法）。 以下示例演示如何使用具有命名类型 Cat 的对象初始值设定项以及如何调用无参数构造函数。 注意自动实现的属性在 Cat 类中的用法。

C#

public class Cat

{

// Auto-implemented properties.

public int Age { get; set; }

public string? Name { get; set; }

public Cat()

{

}

public Cat(string name)

{

this.Name = name;

}

}

C#

Cat cat = new Cat { Age = 10, Name = "Fluffy" };

Cat sameCat = new Cat("Fluffy"){ Age = 10 };

#### 对象初始值设定项语法允许你创建一个实例，然后将具有其分配属性的新建对象指定给赋值中的变量。

除了分配字段和属性外，对象初始值设定项还可以设置索引器。 请思考这个基本的 Matrix 类：

C#

public class Matrix

{

private double[,] storage = new double[3, 3];

public double this[int row, int column]

{

// The embedded array will throw out of range exceptions as appropriate.

get { return storage[row, column]; }

set { storage[row, column] = value; }

}

}

可以使用以下代码初始化标识矩阵：

C#

var identity = new Matrix

{

[0, 0] = 1.0,

[0, 1] = 0.0,

[0, 2] = 0.0,

[1, 0] = 0.0,

[1, 1] = 1.0,

[1, 2] = 0.0,

[2, 0] = 0.0,

[2, 1] = 0.0,

[2, 2] = 1.0,

};

包含可访问资源库的任何可访问索引器都可以用作对象初始值设定项中的表达式之一，这与参数的数量或类型无关。 索引参数构成左侧赋值，而表达式右侧是值。 例如，如果 IndexersExample 具有适当的索引器，则以下初始值设定项都是有效的：

C#

var thing = new IndexersExample

{

name = "object one",

[1] = '1',

[2] = '4',

[3] = '9',

Size = Math.PI,

['C',4] = "Middle C"

}

对于要进行编译的前面的代码，IndexersExample 类型必须具有以下成员：

C#

public string name;

public double Size { set { ... }; }

public char this[int i] { set { ... }; }

public string this[char c, int i] { set { ... }; }

#### 具有匿名类型的对象初始值设定项

尽管对象初始值设定项可用于任何上下文中，但它们在 LINQ 查询表达式中特别有用。 查询表达式常使用只能通过使用对象初始值设定项进行初始化的[匿名类型](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/types/anonymous-types)，如下面的声明所示。

C#

var pet = new { Age = 10, Name = "Fluffy" };

利用匿名类型，LINQ 查询表达式中的 select 子句可以将原始序列的对象转换为其值和形状可能不同于原始序列的对象。 建议只存储某个序列中每个对象的部分信息。 在下面的示例中，假定产品对象 (p) 包含很多字段和方法，而你只想创建包含产品名和单价的对象序列。

C#

var productInfos =

from p in products

select new { p.ProductName, p.UnitPrice };

执行此查询时，productInfos 变量包含一系列对象，这些对象可以在 foreach 语句中进行访问，如下面的示例所示：

C#

foreach(var p in productInfos){...}

新的匿名类型中的每个对象都具有两个公共属性，这两个属性接收与原始对象中的属性或字段相同的名称。 你还可在创建匿名类型时重命名字段；下面的示例将 UnitPrice 字段重命名为 Price。

C#

select new {p.ProductName, Price = p.UnitPrice};

#### 带 required 修饰符的对象初始值设定项

可以使用 required 关键字强制调用方使用对象初始值设定项设置属性或字段的值。 不需要将所需属性设置为构造函数参数。 编译器可确保所有调用方初始化这些值。

C#

public class Pet

{

public required int Age;

public string Name;

}

// `Age` field is necessary to be initialized.

// You don't need to initialize `Name` property

var pet = new Pet() { Age = 10};

// Compiler error:

// Error CS9035 Required member 'Pet.Age' must be set in the object initializer or attribute constructor.

// var pet = new Pet();

通常的做法是保证对象正确初始化，尤其是在要管理多个字段或属性，并且不希望将它们全部包含在构造函数中时。

#### 带 init 访问器的对象初始值设定项

确保无人更改设计，并且可以使用访问器来限制 init 对象。 它有助于限制属性值的设置。

C#

public class Person

{

public string FirstName { get; set; }

public string LastName { get; init; }

}

// The `LastName` property can be set only during initialization. It CAN'T be modified afterwards.

// The `FirstName` property can be modified after initialization.

var pet = new Person() { FirstName = "Joe", LastName = "Doe"};

// You can assign the FirstName property to a different value.

pet.FirstName = "Jane";

// Compiler error:

// Error CS8852 Init - only property or indexer 'Person.LastName' can only be assigned in an object initializer,

// or on 'this' or 'base' in an instance constructor or an 'init' accessor.

// pet.LastName = "Kowalski";

必需的仅限 init 的属性支持不可变结构，同时允许该类型用户使用自然语法。

#### 具有类类型属性的对象初始值设定项

初始化对象时，考虑类类型属性的含义至关重要：

C#

public class HowToClassTypedInitializer

{

public class EmbeddedClassTypeA

{

public int I { get; set; }

public bool B { get; set; }

public string S { get; set; }

public EmbeddedClassTypeB ClassB { get; set; }

public override string ToString() => $"{I}|{B}|{S}|||{ClassB}";

public EmbeddedClassTypeA()

{

Console.WriteLine($"Entering EmbeddedClassTypeA constructor. Values are: {this}");

I = 3;

B = true;

S = "abc";

ClassB = new() { BB = true, BI = 43 };

Console.WriteLine($"Exiting EmbeddedClassTypeA constructor. Values are: {this})");

}

}

public class EmbeddedClassTypeB

{

public int BI { get; set; }

public bool BB { get; set; }

public string BS { get; set; }

public override string ToString() => $"{BI}|{BB}|{BS}";

public EmbeddedClassTypeB()

{

Console.WriteLine($"Entering EmbeddedClassTypeB constructor. Values are: {this}");

BI = 23;

BB = false;

BS = "BBBabc";

Console.WriteLine($"Exiting EmbeddedClassTypeB constructor. Values are: {this})");

}

}

public static void Main()

{

var a = new EmbeddedClassTypeA

{

I = 103,

B = false,

ClassB = { BI = 100003 }

};

Console.WriteLine($"After initializing EmbeddedClassTypeA: {a}");

var a2 = new EmbeddedClassTypeA

{

I = 103,

B = false,

ClassB = new() { BI = 100003 } //New instance

};

Console.WriteLine($"After initializing EmbeddedClassTypeA a2: {a2}");

}

// Output:

//Entering EmbeddedClassTypeA constructor Values are: 0|False||||

//Entering EmbeddedClassTypeB constructor Values are: 0|False|

//Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)

//Exiting EmbeddedClassTypeA constructor Values are: 3|True|abc|||43|True|BBBabc)

//After initializing EmbeddedClassTypeA: 103|False|abc|||100003|True|BBBabc

//Entering EmbeddedClassTypeA constructor Values are: 0|False||||

//Entering EmbeddedClassTypeB constructor Values are: 0|False|

//Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)

//Exiting EmbeddedClassTypeA constructor Values are: 3|True|abc|||43|True|BBBabc)

//Entering EmbeddedClassTypeB constructor Values are: 0|False|

//Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)

//After initializing EmbeddedClassTypeA a2: 103|False|abc|||100003|False|BBBabc

}

以下示例演示了对于 ClassB，初始化过程如何涉及到更新特定的值，同时保留原始实例中的其他值。 初始值设定项重用当前实例：ClassB 的值为：100003（此处分配的新值）、true（在 EmbeddedClassTypeA 的初始化中保留的值）、BBBabc（EmbeddedClassTypeB 中未更改的默认值）。

#### 集合初始值设定项

在初始化实现 [IEnumerable](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.ienumerable) 的集合类型和初始化使用适当的签名作为实例方法或扩展方法的 Add 时，集合初始值设定项允许指定一个或多个元素初始值设定项。 元素初始值设定项可以是值、表达式或对象初始值设定项。 通过使用集合初始值设定项，无需指定多个调用；编译器将自动添加这些调用。

下面的示例演示了两个简单的集合初始值设定项：

C#

List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };

下面的集合初始值设定项使用对象初始值设定项来初始化上一个示例中定义的 Cat 类的对象。 各个对象初始值设定项括在大括号中且用逗号隔开。

C#

List<Cat> cats = new List<Cat>

{

new Cat{ Name = "Sylvester", Age=8 },

new Cat{ Name = "Whiskers", Age=2 },

new Cat{ Name = "Sasha", Age=14 }

};

如果集合的 Add 方法允许，则可以将 [null](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/null) 指定为集合初始值设定项中的一个元素。

C#

List<Cat?> moreCats = new List<Cat?>

{

new Cat{ Name = "Furrytail", Age=5 },

new Cat{ Name = "Peaches", Age=4 },

null

};

如果集合支持读取/写入索引，可以指定索引元素。

C#

var numbers = new Dictionary<int, string>

{

[7] = "seven",

[9] = "nine",

[13] = "thirteen"

};

前面的示例生成调用 [Item[TKey]](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.dictionary-2.item#system-collections-generic-dictionary-2-item(-0)) 以设置值的代码。 还可使用以下语法初始化字典和其他关联容器。 请注意，它使用具有多个值的对象，而不是带括号和赋值的索引器语法：

C#复制

var moreNumbers = new Dictionary<int, string>

{

{19, "nineteen" },

{23, "twenty-three" },

{42, "forty-two" }

};

此初始值设定项示例调用 [Add(TKey, TValue)](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.dictionary-2.add#system-collections-generic-dictionary-2-add(-0-1))，将这三个项添加到字典中。 由于编译器生成的方法调用不同，这两种初始化关联集合的不同方法的行为略有不同。 这两种变量都适用于 Dictionary 类。 其他类型根据它们的公共 API 可能只支持两者中的一种。

#### 具有集合只读属性初始化的对象初始值设定项

某些类可能具有属性为只读的集合属性，如以下示例中 CatOwner 的 Cats 属性：

C#

public class CatOwner

{

public IList<Cat> Cats { get; } = new List<Cat>();

}

由于无法为属性分配新列表，因此你无法使用迄今为止讨论的集合初始值设定项语法：

C#

CatOwner owner = new CatOwner

{

Cats = new List<Cat>

{

new Cat{ Name = "Sylvester", Age=8 },

new Cat{ Name = "Whiskers", Age=2 },

new Cat{ Name = "Sasha", Age=14 }

}

};

但是，可以通过省略列表创建 (new List<Cat>)，使用初始化语法将新条目添加到 Cats，如下所示：

C#

CatOwner owner = new CatOwner

{

Cats =

{

new Cat{ Name = "Sylvester", Age=8 },

new Cat{ Name = "Whiskers", Age=2 },

new Cat{ Name = "Sasha", Age=14 }

}

};

要添加的条目集显示在大括号中。 上述代码与编写代码相同：

C#

CatOwner owner = new ();

owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });

owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });

owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });

##### 示例

下例结合了对象和集合初始值设定项的概念。

C#

public class InitializationSample

{

public class Cat

{

// Auto-implemented properties.

public int Age { get; set; }

public string? Name { get; set; }

public Cat() { }

public Cat(string name)

{

Name = name;

}

}

public static void Main()

{

Cat cat = new Cat { Age = 10, Name = "Fluffy" };

Cat sameCat = new Cat("Fluffy"){ Age = 10 };

List<Cat> cats = new List<Cat>

{

new Cat { Name = "Sylvester", Age = 8 },

new Cat { Name = "Whiskers", Age = 2 },

new Cat { Name = "Sasha", Age = 14 }

};

List<Cat?> moreCats = new List<Cat?>

{

new Cat { Name = "Furrytail", Age = 5 },

new Cat { Name = "Peaches", Age = 4 },

null

};

// Display results.

System.Console.WriteLine(cat.Name);

foreach (Cat c in cats)

{

System.Console.WriteLine(c.Name);

}

foreach (Cat? c in moreCats)

{

if (c != null)

{

System.Console.WriteLine(c.Name);

}

else

{

System.Console.WriteLine("List element has null value.");

}

}

}

// Output:

//Fluffy

//Sylvester

//Whiskers

//Sasha

//Furrytail

//Peaches

//List element has null value.

}

以下示例演示了一个对象，该对象实现 [IEnumerable](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.ienumerable) 并包含具有多个参数的 Add 方法。 它使用一个集合初始值设定项，其中列表中的每个项都有多个元素，这些元素对应于 Add 方法的签名。

C#

public class FullExample

{

class FormattedAddresses : IEnumerable<string>

{

private List<string> internalList = new List<string>();

public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() => internalList.GetEnumerator();

public void Add(string firstname, string lastname,

string street, string city,

string state, string zipcode) => internalList.Add($"""

{firstname} {lastname}

{street}

{city}, {state} {zipcode}

"""

);

}

public static void Main()

{

FormattedAddresses addresses = new FormattedAddresses()

{

{"John", "Doe", "123 Street", "Topeka", "KS", "00000" },

{"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }

};

Console.WriteLine("Address Entries:");

foreach (string addressEntry in addresses)

{

Console.WriteLine("\r\n" + addressEntry);

}

}

/\*

\* Prints:

Address Entries:

John Doe

123 Street

Topeka, KS 00000

Jane Smith

456 Street

Topeka, KS 00000

\*/

}

Add 方法可使用 params 关键字来获取可变数量的自变量，如下例中所示。 此示例还演示了索引器的自定义实现，以使用索引初始化集合。 从 C# 13 开始，params 参数不再局限于数组。 它可以是集合类型或接口。

C#

public class DictionaryExample

{

class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull

{

private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>();

public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() => internalDictionary.GetEnumerator();

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() => internalDictionary.GetEnumerator();

public List<TValue> this[TKey key]

{

get => internalDictionary[key];

set => Add(key, value);

}

public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable<TValue>)values);

public void Add(TKey key, IEnumerable<TValue> values)

{

if (!internalDictionary.TryGetValue(key, out List<TValue>? storedValues))

{

internalDictionary.Add(key, storedValues = new List<TValue>());

}

storedValues.AddRange(values);

}

}

public static void Main()

{

RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary1

= new RudimentaryMultiValuedDictionary<string, string>()

{

{"Group1", "Bob", "John", "Mary" },

{"Group2", "Eric", "Emily", "Debbie", "Jesse" }

};

RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary2

= new RudimentaryMultiValuedDictionary<string, string>()

{

["Group1"] = new List<string>() { "Bob", "John", "Mary" },

["Group2"] = new List<string>() { "Eric", "Emily", "Debbie", "Jesse" }

};

RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary3

= new RudimentaryMultiValuedDictionary<string, string>()

{

{"Group1", new string []{ "Bob", "John", "Mary" } },

{ "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }

};

Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary1)

{

Console.WriteLine($"\r\nMembers of group {group.Key}: ");

foreach (string member in group.Value)

{

Console.WriteLine(member);

}

}

Console.WriteLine("\r\nUsing second multi-valued dictionary created with a collection initializer using indexing:");

foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary2)

{

Console.WriteLine($"\r\nMembers of group {group.Key}: ");

foreach (string member in group.Value)

{

Console.WriteLine(member);

}

}

Console.WriteLine("\r\nUsing third multi-valued dictionary created with a collection initializer using indexing:");

foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary3)

{

Console.WriteLine($"\r\nMembers of group {group.Key}: ");

foreach (string member in group.Value)

{

Console.WriteLine(member);

}

}

}

/\*

\* Prints:

Using first multi-valued dictionary created with a collection initializer:

Members of group Group1:

Bob

John

Mary

Members of group Group2:

Eric

Emily

Debbie

Jesse

Using second multi-valued dictionary created with a collection initializer using indexing:

Members of group Group1:

Bob

John

Mary

Members of group Group2:

Eric

Emily

Debbie

Jesse

Using third multi-valued dictionary created with a collection initializer using indexing:

Members of group Group1:

Bob

John

Mary

Members of group Group2:

Eric

Emily

Debbie

Jesse

\*/

}

### 匿名类型

在不方便或没必要创建命名类的情况下，例如当使用无需保留或传递给其他方法的数据结构填充列表时，可以使用匿名类型. 匿名类型提供了一种方便的方法，可用来将一组只读属性封装到单个对象中，而无需首先显式定义一个类型。 类型名由编译器生成，并且不能在源代码级使用。 每个属性的类型由编译器推断。

可结合使用 [new](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/new-operator) 运算符和对象初始值设定项创建匿名类型。

以下示例显示了用两个名为 Amount 和 Message 的属性进行初始化的匿名类型。

C#复制

var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following

// statement to verify that their inferred types are int and string.

Console.WriteLine(v.Amount + v.Message);

匿名类型通常用在查询表达式的 [select](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/select-clause) 子句中，以便返回源序列中每个对象的属性子集。 有关查询的详细信息

匿名类型包含一个或多个公共只读属性。 包含其他种类的类成员（如方法或事件）为无效。 用来初始化属性的表达式不能为 null、匿名函数或指针类型。

最常见的方案是用其他类型的属性初始化匿名类型。 在下面的示例中，假定名为 Product 的类存在。 类 Product 包括 Color 和 Price 属性，以及你不感兴趣的其他属性。 变量 Productproducts 是 对象的集合。 匿名类型声明以 new 关键字开始。 声明初始化了一个只使用 Product 的两个属性的新类型。 使用匿名类型会导致在查询中返回的数据量变少。

如果你没有在匿名类型中指定成员名称，编译器会为匿名类型成员指定与用于初始化这些成员的属性相同的名称。 需要为使用表达式初始化的属性提供名称，如下面的示例所示。 在下面示例中，匿名类型的属性名称都为 PriceColor 和 。

C#复制

var productQuery =

from prod in products

select new { prod.Color, prod.Price };

foreach (var v in productQuery)

{

Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);

}

**提示**

可以使用 .NET 样式规则 [**IDE0037**](https://learn.microsoft.com/zh-cn/dotnet/fundamentals/code-analysis/style-rules/ide0037) 强制执行是首选推断成员名称还是显式成员名称。

还可以按另一种类型（类、结构或另一个匿名类型）的对象定义字段。 它通过使用保存此对象的变量来完成，如以下示例中所示，其中两个匿名类型是使用已实例化的用户定义类型创建的。 在这两种情况下，匿名类型 shipment 和 shipmentWithBonus 中的 product 字段的类型均为 Product，其中包含每个字段的默认值。 bonus 字段将是编译器创建的匿名类型。

C#复制

var product = new Product();

var bonus = new { note = "You won!" };

var shipment = new { address = "Nowhere St.", product };

var shipmentWithBonus = new { address = "Somewhere St.", product, bonus };

通常，当使用匿名类型来初始化变量时，可以通过使用 [var](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/statements/declarations#implicitly-typed-local-variables) 将变量作为隐式键入的本地变量来进行声明。 类型名称无法在变量声明中给出，因为只有编译器能访问匿名类型的基础名称。 有关 var 的详细信息，请参阅[隐式类型本地变量](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables)。

可通过将隐式键入的本地变量与隐式键入的数组相结合创建匿名键入的元素的数组，如下面的示例所示。

C#复制

var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};

匿名类型是 [class](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/class) 类型，它们直接派生自 [object](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/reference-types)，并且无法强制转换为除 [object](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/reference-types) 外的任何类型。 虽然你的应用程序不能访问它，编译器还是提供了每一个匿名类型的名称。 从公共语言运行时的角度来看，匿名类型与任何其他引用类型没有什么不同。

如果程序集中的两个或多个匿名对象初始值指定了属性序列，这些属性采用相同顺序且具有相同的名称和类型，则编译器将对象视为相同类型的实例。 它们共享同一编译器生成的类型信息。

匿名类型支持采用 [with 表达式](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/with-expression)形式的非破坏性修改。 这使你能够创建匿名类型的新实例，其中一个或多个属性具有新值：

C#复制

var apple = new { Item = "apples", Price = 1.35 };

var onSale = apple with { Price = 0.79 };

Console.WriteLine(apple);

Console.WriteLine(onSale);

无法将字段、属性、时间或方法的返回类型声明为具有匿名类型。 同样，你不能将方法、属性、构造函数或索引器的形参声明为具有匿名类型。 要将匿名类型或包含匿名类型的集合作为参数传递给某一方法，可将参数作为类型 object 进行声明。 但是，对匿名类型使用 object 违背了强类型的目的。 如果必须存储查询结果或者必须将查询结果传递到方法边界外部，请考虑使用普通的命名结构或类而不是匿名类型。

由于匿名类型上的 [Equals](https://learn.microsoft.com/zh-cn/dotnet/api/system.object.equals) 和 [GetHashCode](https://learn.microsoft.com/zh-cn/dotnet/api/system.object.gethashcode) 方法是根据方法属性的 Equals 和 GetHashCode 定义的，因此仅当同一匿名类型的两个实例的所有属性都相等时，这两个实例才相等。

**备注**

匿名类型的[**辅助功能级别**](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers)为 internal，因此在不同程序集中定义的两种匿名类型并非同一类型。 因此，当在不同的程序集中进行定义时，匿名类型的实例不能彼此相等，即使其所有属性都相等。

匿名类型确实会重写 [ToString](https://learn.microsoft.com/zh-cn/dotnet/api/system.object.tostring) 方法，将用大括号括起来的每个属性的名称和 ToString 输出连接起来。

复制

var v = new { Title = "Hello", Age = 24 };

Console.WriteLine(v.ToString()); // "{ Title = Hello, Age = 24 }"

### 扩展方法

通过创建一个单独的类型，然后将该类型的方法当作原始类型的方法来调用，可以在不必创建派生类的情况下对类进行“扩展”。

扩展方法使你能够向现有类型“添加”方法，而无需创建新的派生类型、重新编译或以其他方式修改原始类型。 扩展方法是一种静态方法，但可以像扩展类型上的实例方法一样进行调用。 对于用 C#、F# 和 Visual Basic 编写的客户端代码，调用扩展方法与调用在类型中定义的方法没有明显区别。

最常见的扩展方法是 LINQ 标准查询运算符，它将查询功能添加到现有的 [System.Collections.IEnumerable](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.ienumerable) 和 [System.Collections.Generic.IEnumerable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.ienumerable-1) 类型。 若要使用标准查询运算符，请先使用 using System.Linq 指令将它们置于范围中。 然后，任何实现了 [IEnumerable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.ienumerable-1) 的类型看起来都具有 [GroupBy](https://learn.microsoft.com/zh-cn/dotnet/api/system.linq.enumerable.groupby)、[OrderBy](https://learn.microsoft.com/zh-cn/dotnet/api/system.linq.enumerable.orderby)、[Average](https://learn.microsoft.com/zh-cn/dotnet/api/system.linq.enumerable.average) 等实例方法。 在 [IEnumerable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.ienumerable-1) 类型的实例（如 [List<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.list-1) 或 [Array](https://learn.microsoft.com/zh-cn/dotnet/api/system.array)）后键入“dot”时，可以在 IntelliSense 语句完成中看到这些附加方法。

#### OrderBy 示例

下面的示例演示如何对一个整数数组调用标准查询运算符 OrderBy 方法。 括号里面的表达式是一个 lambda 表达式。 很多标准查询运算符采用 Lambda 表达式作为参数，但这不是扩展方法的必要条件。 有关详细信息，请参阅 [Lambda 表达式](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/lambda-expressions)。

C#复制

class ExtensionMethods2

{

static void Main()

{

int[] ints = [10, 45, 15, 39, 21, 26];

var result = ints.OrderBy(g => g);

foreach (var i in result)

{

System.Console.Write(i + " ");

}

}

}

//Output: 10 15 21 26 39 45

扩展方法被定义为静态方法，但它们是通过实例方法语法进行调用的。 它们的第一个参数指定方法操作的类型。 该参数位于 [this](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/this) 修饰符之后。 仅当你使用 using 指令将命名空间显式导入到源代码中之后，扩展方法才位于范围中。

下面的示例演示为 [System.String](https://learn.microsoft.com/zh-cn/dotnet/api/system.string) 类定义的一个扩展方法。 它是在非嵌套的、非泛型静态类内部定义的：

C#复制

namespace ExtensionMethods

{

public static class MyExtensions

{

public static int WordCount(this string str)

{

return str.Split(new char[] { ' ', '.', '?' },

StringSplitOptions.RemoveEmptyEntries).Length;

}

}

}

可使用此 WordCount 指令将 using 扩展方法置于范围中：

C#复制

using ExtensionMethods;

而且，可以使用以下语法从应用程序中调用该扩展方法：

C#复制

string s = "Hello Extension Methods";

int i = s.WordCount();

在代码中，可以使用实例方法语法调用该扩展方法。 编译器生成的中间语言 (IL) 会将代码转换为对静态方法的调用。 并未真正违反封装原则。 扩展方法无法访问它们所扩展的类型中的专用变量。

MyExtensions 类和 WordCount 方法都是 static，可以像所有其他 static 成员那样对其进行访问。 WordCount 方法可以像其他 static 方法一样调用，如下所示：

C#复制

string s = "Hello Extension Methods";

int i = MyExtensions.WordCount(s);

上述 C# 代码：

* 声明并分配一个名为 s 和值为 "Hello Extension Methods" 的新 string。
* 调用 MyExtensions.WordCount 给定自变量 s。

有关详细信息，请参阅[如何实现和调用自定义扩展方法](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/how-to-implement-and-call-a-custom-extension-method)。

通常，你更多时候是调用扩展方法而不是实现你自己的扩展方法。 由于扩展方法是使用实例方法语法调用的，因此不需要任何特殊知识即可从客户端代码中使用它们。 若要为特定类型启用扩展方法，只需为在其中定义这些方法的命名空间添加 using 指令。 例如，若要使用标准查询运算符，请将此 using 指令添加到代码中：

C#复制

using System.Linq;

（你可能还必须添加对 System.Core.dll 的引用。）你将注意到，标准查询运算符现在作为可供大多数 [IEnumerable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.ienumerable-1) 类型使用的附加方法显示在 IntelliSense 中。

#### 在编译时绑定扩展方法

可以使用扩展方法来扩展类或接口，但不能重写扩展方法。 与接口或类方法具有相同名称和签名的扩展方法永远不会被调用。 编译时，扩展方法的优先级总是比类型本身中定义的实例方法低。 换句话说，如果某个类型具有一个名为 Process(int i) 的方法，而你有一个具有相同签名的扩展方法，则编译器总是绑定到该实例方法。 当编译器遇到方法调用时，它首先在该类型的实例方法中寻找匹配的方法。 如果未找到任何匹配方法，编译器将搜索为该类型定义的任何扩展方法，并且绑定到它找到的第一个扩展方法。

##### 示例

下面的示例演示 C# 编译器在确定是将方法调用绑定到类型上的实例方法还是绑定到扩展方法时所遵循的规则。 静态类 Extensions 包含为任何实现了 IMyInterface 的类型定义的扩展方法。 类 A、B 和 C 都实现了该接口。

MethodB 扩展方法永远不会被调用，因为它的名称和签名与这些类已经实现的方法完全匹配。

如果编译器找不到具有匹配签名的实例方法，它会绑定到匹配的扩展方法（如果存在这样的方法）。

C#复制

// Define an interface named IMyInterface.

namespace DefineIMyInterface

{

public interface IMyInterface

{

// Any class that implements IMyInterface must define a method

// that matches the following signature.

void MethodB();

}

}

// Define extension methods for IMyInterface.

namespace Extensions

{

using System;

using DefineIMyInterface;

// The following extension methods can be accessed by instances of any

// class that implements IMyInterface.

public static class Extension

{

public static void MethodA(this IMyInterface myInterface, int i)

{

Console.WriteLine

("Extension.MethodA(this IMyInterface myInterface, int i)");

}

public static void MethodA(this IMyInterface myInterface, string s)

{

Console.WriteLine

("Extension.MethodA(this IMyInterface myInterface, string s)");

}

// This method is never called in ExtensionMethodsDemo1, because each

// of the three classes A, B, and C implements a method named MethodB

// that has a matching signature.

public static void MethodB(this IMyInterface myInterface)

{

Console.WriteLine

("Extension.MethodB(this IMyInterface myInterface)");

}

}

}

// Define three classes that implement IMyInterface, and then use them to test

// the extension methods.

namespace ExtensionMethodsDemo1

{

using System;

using Extensions;

using DefineIMyInterface;

class A : IMyInterface

{

public void MethodB() { Console.WriteLine("A.MethodB()"); }

}

class B : IMyInterface

{

public void MethodB() { Console.WriteLine("B.MethodB()"); }

public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }

}

class C : IMyInterface

{

public void MethodB() { Console.WriteLine("C.MethodB()"); }

public void MethodA(object obj)

{

Console.WriteLine("C.MethodA(object obj)");

}

}

class ExtMethodDemo

{

static void Main(string[] args)

{

// Declare an instance of class A, class B, and class C.

A a = new A();

B b = new B();

C c = new C();

// For a, b, and c, call the following methods:

// -- MethodA with an int argument

// -- MethodA with a string argument

// -- MethodB with no argument.

// A contains no MethodA, so each call to MethodA resolves to

// the extension method that has a matching signature.

a.MethodA(1); // Extension.MethodA(IMyInterface, int)

a.MethodA("hello"); // Extension.MethodA(IMyInterface, string)

// A has a method that matches the signature of the following call

// to MethodB.

a.MethodB(); // A.MethodB()

// B has methods that match the signatures of the following

// method calls.

b.MethodA(1); // B.MethodA(int)

b.MethodB(); // B.MethodB()

// B has no matching method for the following call, but

// class Extension does.

b.MethodA("hello"); // Extension.MethodA(IMyInterface, string)

// C contains an instance method that matches each of the following

// method calls.

c.MethodA(1); // C.MethodA(object)

c.MethodA("hello"); // C.MethodA(object)

c.MethodB(); // C.MethodB()

}

}

}

/\* Output:

Extension.MethodA(this IMyInterface myInterface, int i)

Extension.MethodA(this IMyInterface myInterface, string s)

A.MethodB()

B.MethodA(int i)

B.MethodB()

Extension.MethodA(this IMyInterface myInterface, string s)

C.MethodA(object obj)

C.MethodA(object obj)

C.MethodB()

\*/

#### 常见使用模式

#### 集合功能

过去，创建”集合类”通常是为了使给定类型实现 [System.Collections.Generic.IEnumerable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.ienumerable-1) 接口，并实现对该类型集合的功能。 创建这种类型的集合对象没有任何问题，但也可以通过对 [System.Collections.Generic.IEnumerable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.ienumerable-1) 使用扩展来实现相同的功能。 扩展的优势是允许从任何集合（如 [System.Array](https://learn.microsoft.com/zh-cn/dotnet/api/system.array) 或实现该类型 [System.Collections.Generic.IEnumerable<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.ienumerable-1) 的 [System.Collections.Generic.List<T>](https://learn.microsoft.com/zh-cn/dotnet/api/system.collections.generic.list-1)）调用功能。 可以在[本文前面的内容](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/extension-methods?redirectedfrom=MSDN#orderby-example)中找到使用 Int32 的数组的示例。

#### 特定于层的功能

使用洋葱架构或其他分层应用程序设计时，通常具有一组域实体或数据传输对象，可用于跨应用程序边界进行通信。 这些对象通常不包含任何功能，或者只包含适用于应用程序的所有层的最少功能。 使用扩展方法可以添加特定于每个应用程序层的功能，而无需使用其他层中不需要的方法来向下加载对象。

C#复制

public class DomainEntity

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

}

static class DomainEntityExtensions

{

static string FullName(this DomainEntity value)

=> $"{value.FirstName} {value.LastName}";

}

#### 扩展预定义类型

当需要创建可重用功能时，我们无需创建新对象，而是可以扩展现有类型，例如 .NET 或 CLR 类型。 例如，如果不使用扩展方法，我们可能会创建 Engine 或 Query 类，对可从代码中的多个位置调用的 SQL Server 执行查询。 但是，如果换做使用扩展方法扩展 [System.Data.SqlClient.SqlConnection](https://learn.microsoft.com/zh-cn/dotnet/api/system.data.sqlclient.sqlconnection) 类，就可以从与 SQL Server 连接的任何位置执行该查询。 其他示例可能是向 [System.String](https://learn.microsoft.com/zh-cn/dotnet/api/system.string) 类添加常见功能、扩展 [System.IO.Stream](https://learn.microsoft.com/zh-cn/dotnet/api/system.io.stream) 和 [System.Exception](https://learn.microsoft.com/zh-cn/dotnet/api/system.exception) 对象的数据处理功能以实现特定的错误处理功能。 这些用例的类型仅受想象力和判断力的限制。

使用 struct 类型扩展预定义类型可能很困难，因为它们已通过值传递给方法。 这意味着将对结构的副本进行任何结构更改。 扩展方法退出后，将不显示这些更改。 可以将 ref 修饰符添加到第一个参数，使其成为 ref 扩展方法。 ref 关键字可以在 this 关键字之前或之后显示，不会有任何语义差异。 添加 ref 修饰符表示第一个参数是按引用传递的。 在这种情况下，可以编写扩展方法来更改要扩展的结构的状态（请注意，私有成员不可访问）。 仅允许值类型或受结构约束的泛型类型（有关详细信息，请参阅 [struct 约束](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct#struct-constraint)）作为 ref 扩展方法的第一个参数。 以下示例演示如何使用 ref 扩展方法直接修改内置类型，而无需重新分配结果或使用 ref 关键字传递函数：

C#复制

public static class IntExtensions

{

public static void Increment(this int number)

=> number++;

// Take note of the extra ref keyword here

public static void RefIncrement(this ref int number)

=> number++;

}

public static class IntProgram

{

public static void Test()

{

int x = 1;

// Takes x by value leading to the extension method

// Increment modifying its own copy, leaving x unchanged

x.Increment();

Console.WriteLine($"x is now {x}"); // x is now 1

// Takes x by reference leading to the extension method

// RefIncrement changing the value of x directly

x.RefIncrement();

Console.WriteLine($"x is now {x}"); // x is now 2

}

}

下一个示例演示用户定义的结构类型的 ref 扩展方法：

C#复制

public struct Account

{

public uint id;

public float balance;

private int secret;

}

public static class AccountExtensions

{

// ref keyword can also appear before the this keyword

public static void Deposit(ref this Account account, float amount)

{

account.balance += amount;

// The following line results in an error as an extension

// method is not allowed to access private members

// account.secret = 1; // CS0122

}

}

public static class AccountProgram

{

public static void Test()

{

Account account = new()

{

id = 1,

balance = 100f

};

Console.WriteLine($"I have ${account.balance}"); // I have $100

account.Deposit(50f);

Console.WriteLine($"I have ${account.balance}"); // I have $150

}

}

#### 通用准则

尽管通过修改对象的代码来添加功能，或者在合理和可行的情况下派生新类型等方式仍是可取的，但扩展方法已成为在整个 .NET 生态系统中创建可重用功能的关键选项。 对于原始源不受控制、派生对象不合适或不可用，或者不应在功能适用范围之外公开功能的情况，扩展方法是一个不错的选择。

有关派生类型的详细信息，请参阅[继承](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/object-oriented/inheritance)。

在使用扩展方法来扩展你无法控制其源代码的类型时，你需要承受该类型实现中的更改会导致扩展方法失效的风险。

如果确实为给定类型实现了扩展方法，请记住以下几点：

* 如果扩展方法与该类型中定义的方法具有相同的签名，则扩展方法永远不会被调用。
* 在命名空间级别将扩展方法置于范围中。 例如，如果你在一个名为 Extensions 的命名空间中具有多个包含扩展方法的静态类，则这些扩展方法将全部由 using Extensions; 指令置于范围中。

针对已实现的类库，不应为了避免程序集的版本号递增而使用扩展方法。 如果要向你拥有源代码的库中添加重要功能，请遵循适用于程序集版本控制的 .NET 准则。

### 隐式类型化局部变量

在类或结构方法中，可以使用隐式类型来指示编译器在编译时确定正确的类型.

可声明局部变量而无需提供显式类型。 var 关键字指示编译器通过初始化语句右侧的表达式推断变量的类型。 推断类型可以是内置类型、匿名类型、用户定义类型或 .NET 类库中定义的类型。 有关如何使用 var 初始化数组的详细信息，请参阅[隐式类型化数组](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/arrays#implicitly-typed-arrays)。

##### 以下示例演示使用 var 声明局部变量的各种方式：

C#

// i is compiled as an int

var i = 5;

// s is compiled as a string

var s = "Hello";

// a is compiled as int[]

var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>

// or perhaps IQueryable<Customer>

var expr =

from c in customers

where c.City == "London"

select c;

// anon is compiled as an anonymous type

var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>

var list = new List<int>();

##### 重要的是了解 var 关键字并不意味着“变体”，并且并不指示变量是松散类型或是后期绑定。 它只表示由编译器确定并分配最适合的类型。

##### 在以下上下文中，可使用 var 关键字：

* 在局部变量（在方法范围内声明的变量）上，如前面的示例所示。
* 在 [for](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/statements/iteration-statements#the-for-statement) 初始化语句中。

C#复制

for (var x = 1; x < 10; x++)

* 在 [foreach](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/statements/iteration-statements#the-foreach-statement) 初始化语句中。

C#复制

foreach (var item in list) {...}

* 在 [using](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/statements/using) 域间中。

C#复制

using (var file = new StreamReader("C:\\myfile.txt")) {...}

有关详细信息，请参阅[如何在查询表达式中使用隐式类型化局部变量和数组](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/how-to-use-implicitly-typed-local-variables-and-arrays-in-a-query-expression)。

#### var 和匿名类型

在许多情况下，使用 var 是可选的，只是一种语法便利。 但是，在使用匿名类型初始化变量时，如果需要在以后访问对象的属性，则必须将变量声明为 var。 这是 LINQ 查询表达式中的常见方案。 有关详细信息，请参阅[匿名类型](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/types/anonymous-types)。

从源代码角度来看，匿名类型没有名称。 因此，如果使用 var 初始化了查询变量，则访问返回对象序列中的属性的唯一方法是在 foreach 语句中将 var 用作迭代变量的类型。

C#

class ImplicitlyTypedLocals2

{

static void Main()

{

string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

// If a query produces a sequence of anonymous types,

// then use var in the foreach statement to access the properties.

var upperLowerWords =

from w in words

select new { Upper = w.ToUpper(), Lower = w.ToLower() };

// Execute the query

foreach (var ul in upperLowerWords)

{

Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);

}

}

}

/\* Outputs:

Uppercase: APPLE, Lowercase: apple

Uppercase: BLUEBERRY, Lowercase: blueberry

Uppercase: CHERRY, Lowercase: cherry

\*/

#### 备注

##### 以下限制适用于隐式类型化变量声明：

* 仅当局部变量在相同语句中进行声明和初始化时，才能使用 var；使用var定义的变量不能初始化为 null，也不能初始化为方法组或匿名函数。
* var 不能在类范围内对字段使用。
* 使用 var 声明的变量不能在初始化表达式中使用。 换句话说，此表达式是合法的：int i = (i = 20);，但是此表达式会生成编译时错误：var i = (i = 20);
* 不能在相同语句中初始化多个隐式类型化变量。
* 如果一种名为 var 的类型处于范围内，则 var 关键字会解析为该类型名称，不会被视为隐式类型化局部变量声明的一部分。

带 var 关键字的隐式类型只能应用于本地方法范围内的变量。 隐式类型不可用于类字段，因为 C# 编译器在处理代码时会遇到逻辑悖论：编译器需要知道字段的类型，但它在分析赋值表达式前无法确定类型，而表达式在不知道类型的情况下无法进行计算。 考虑下列代码：

C#

private var bookTitles;

bookTitles 是类型为 var 的类字段。 由于该字段没有要计算的表达式，编译器无法推断出 bookTitles 应该是哪种类型。 此外，向该字段添加表达式（就像对本地变量执行的操作一样）也是不够的：

C#

private var bookTitles = new List<string>();

当编译器在代码编译期间遇到字段时，它会在处理与其关联的任何表达式之前记录每个字段的类型。 编译器在尝试分析 bookTitles 时遇到相同的悖论：它需要知道字段的类型，但编译器通常会通过分析表达式来确定 var 的类型，这在事先不知道类型的情况下无法实现。

你可能会发现，对于在其中难以确定查询变量的确切构造类型的查询表达式，var 也可能会十分有用。 这可能会针对分组和排序操作发生。

当变量的特定类型在键盘上键入时很繁琐、或是显而易见、或是不会提高代码的可读性时，var 关键字也可能非常有用。 var 采用此方法提供帮助的一个示例是针对嵌套泛型类型（如用于分组操作的类型）。 在下面的查询中，查询变量的类型是 IEnumerable<IGrouping<string, Student>>。 只要你和必须维护你的代码的其他人了解这一点，使用隐式类型化实现便利性和简便性时便不会出现问题。

C#

// Same as previous example except we use the entire last name as a key.

// Query variable is an IEnumerable<IGrouping<string, Student>>

var studentQuery3 =

from student in students

group student by student.Last;

使用 var 有助于简化代码，但是它的使用应该限制在需要使用它的情况下，或在它可使代码更易于读取的情况下。 有关何时正确使用 var 的详细信息，请参阅 C# 编码指南一文中的[隐式类型本地变量](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/coding-style/coding-conventions#implicitly-typed-local-variables)节。