# 扩展方法（C# 编程指南）

****Visual Studio 2015****

若要了解有关 Visual Studio 2017 RC 的最新文档，请参阅 [Visual Studio 2017 RC 文档](http://docs.microsoft.com/visualstudio)。

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。 扩展方法是一种特殊的静态方法，但可以像扩展类型上的实例方法一样进行调用。 对于用 C# 和 Visual Basic 编写的客户端代码，调用扩展方法与调用在类型中实际定义的方法之间没有明显的差异。

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://msdn.microsoft.com/zh-cn/library/system.collections.ienumerable.aspx) 和[System.Collections.Generic.IEnumerable<T>](https://msdn.microsoft.com/zh-cn/library/9eekhta0.aspx) 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://msdn.microsoft.com/zh-cn/library/9eekhta0.aspx) 的类型看起来都具有 [GroupBy<TSource, TKey>](https://msdn.microsoft.com/zh-cn/library/bb534501.aspx)、[OrderBy<TSource, TKey>](https://msdn.microsoft.com/zh-cn/library/bb534966.aspx)、[Average](https://msdn.microsoft.com/zh-cn/library/bb358946.aspx) 等实例方法。 在 [IEnumerable<T>](https://msdn.microsoft.com/zh-cn/library/9eekhta0.aspx) 类型的实例（如 [List<T>](https://msdn.microsoft.com/zh-cn/library/6sh2ey19.aspx) 或 [Array](https://msdn.microsoft.com/zh-cn/library/system.array.aspx)）后键入“dot”时，可以在 IntelliSense 语句完成中看到这些附加方法。

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://msdn.microsoft.com/zh-cn/library/bb397687.aspx)。

**C#**

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该方法作用于哪个类型，并且该参数以 [this](https://msdn.microsoft.com/zh-cn/library/dk1507sz.aspx)修饰符为前缀。 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://msdn.microsoft.com/zh-cn/library/system.string.aspx) 类定义的一个扩展方法。 请注意，它是在非嵌套的、非泛型静态类内部定义的：

**C#**

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 using 指令将 WordCount 扩展方法置于范围中：

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：

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。 但是，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。 因此，并未真正违反封装原则。 实际上，扩展方法无法访问它们所扩展的类型中的私有变量。

有关详细信息，请参阅[如何：实现和调用自定义扩展方法](https://msdn.microsoft.com/zh-cn/library/bb311042.aspx)。

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：

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://msdn.microsoft.com/zh-cn/library/9eekhta0.aspx) 类型使用的附加方法显示在 IntelliSense 中。

|  |
| --- |
| **System_CAPS_ICON_note.jpg 说明** |
| 尽管标准查询运算符没有显示在 [String](https://msdn.microsoft.com/zh-cn/library/system.string.aspx) 的 IntelliSense 中，但它们仍然可用。 |

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。 下面的示例演示编译器如何确定要绑定到哪个扩展方法或实例方法。

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#**

// Define an interface named IMyInterface.namespace DefineIMyInterface

{

using System;

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(object, int)

a.MethodA("hello"); // Extension.MethodA(object, 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(object, 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()

\*/

通用准则

通常，建议你只在不得已的情况下才实现扩展方法，并谨慎地实现。 只要有可能，必须扩展现有类型的客户端代码都应该通过创建从现有类型派生的新类型来达到这一目的。 有关详细信息，请参阅[继承](https://msdn.microsoft.com/zh-cn/library/ms173149.aspx)。

在使用扩展方法来扩展你无法更改其源代码的类型时，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。

如果你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，应遵循适用于程序集版本控制的标准 .NET Framework 准则。 有关详细信息，请参阅[程序集版本控制](https://msdn.microsoft.com/zh-cn/library/51ket42z.aspx)。