目录

[C# 编码约定 2](#_Toc116474621)

[命名约定 2](#_Toc116474622)

[基本约定 2](#_Toc116474623)

[标示符类型命名约定 4](#_Toc116474624)

[布局约定 5](#_Toc116474625)

[注释约定 7](#_Toc116474626)

[语言准则 7](#_Toc116474627)

[修饰语 7](#_Toc116474628)

[字符串数据类型 8](#_Toc116474629)

[类型转换 8](#_Toc116474630)

[隐式类型本地变量 9](#_Toc116474631)

[数组 10](#_Toc116474632)

[委托 10](#_Toc116474633)

[try-catch 和 using 语句正在异常处理中 11](#_Toc116474634)

[&& 和 || 运算符 12](#_Toc116474635)

[事件处理 13](#_Toc116474636)

[静态成员 14](#_Toc116474637)

[LINQ 查询 14](#_Toc116474638)

[goto语句 15](#_Toc116474639)

[其它参考 15](#_Toc116474640)

[附1 标识符编写约定表 15](#_Toc116474641)

[附2 组件名称缩写列表 16](#_Toc116474642)

# C# 编码约定

主要目标是在公司源代码中实现一致性和可读性。

该版基于.net Framework 4.8。

## 命名约定

### 基本约定

* 使用可以准确说明变量/字段/类的完整的英文描述符，如firstName。对一些作用显而易见的变量可以采用简单的命名，如在循环里的递增（减）变量就可以被命名为 “i”。
* 要尽量采用项目所涉及领域的术语。
* 有两种适合的书写方法，适应于不同类型的[标识符](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/coding-style/identifier-names)

**帕斯卡拼写法**

命名class或struct时，使用pascal大小写（“PascalCasing”）。

public class DataService

{

}

public struct ValueCoordinate

{

}

命名interface时，使用pascal大小写并在名称前面加上前缀“I”。这可以清楚地向使用者表明这是interface。

public interface IWorkerQueue

{

}

命名类型的public成员（例如字段、属性、事件、方法和本地函数）时，使用pascal大小写。

public class ExampleEvents

{

// A public field, these should be used sparingly

public bool IsValid;

// An init-only property

public IWorkerQueue WorkerQueue { get; init; }

// An event

public event Action EventProcessing;

// Method

public void StartEventProcessing()

{

// Local function

static int CountQueueItems() => WorkerQueue.Count;

// ...

}

}

**驼峰式大小写**

命名private或internal字段时，使用驼峰式大小写（“camelCasing”），并且它们以“\_”作为前缀。

public class DataService

{

private IWorkerQueue \_workerQueue;

}

为private或internal的static字段时，使用“s\_”前缀，对于线程静态，使用“t\_”。

public class DataService

{

private static IWorkerQueue s\_workerQueue;

[ThreadStatic]

private static TimeSpan t\_timeSpan;

}

编写方法参数时，使用驼峰式大小写。

public T SomeMethod<T>(int someNumber, bool isValid)

{

}

* 避免使用缩写，如果一定要使用，就谨慎使用。同时，应该保留一个标准缩写的列表，并且在使用时保持一致。
* 对常见缩略词，两个字母的缩写要采用统一大小写的方式（例如：ioStream，getIOStream）；多字母缩写采用首字母大写，其他字母小写的方式（例如：getHtmlTag）。
* 避免使用长名字（最好不超过15个字母）。
* 避免使用相似或者仅在大小写上有区别的名字。

### 标示符类型命名约定

* 程序集和命名空间

公司名称 + 项目名称 + 模块名称（可选），例如： Company.ProjectName.ModuleName。

一般情况下命名空间和目录结构相同。

* 类

类的名字要用名词。

避免使用单词的缩写，除非它的缩写已经广为人知，如HTTP。

　　泛型类型参数命名：为T或者以T开头的描述性名字，例如：

public class List<T>

{

}

public class MyClass<TSession>

{

}

对同一项目的不同命名空间中的类，命名避免重复。避免引用时的冲突和混淆。

* 方法

第一个单词一般是动词。

如果方法返回一个成员变量的值，方法名以Get + 成员变量名，如若返回的值是bool变量，一般以Is作为前缀。尽量用属性来替代方法。

如果方法修改一个成员变量的值，方法名以：Set + 成员变量名。尽量用属性来替代方法。

* 变量

　　不要用“&”作为第一个字母。

　　尽量使用短且有意义的单词。

如果变量是集合，则变量名要用复数。例如表格的行数：RowsCount。

## 布局约定

好的布局利用格式设置来强调代码的结构并使代码更便于阅读。

* 使用默认的代码编辑器设置（智能缩进、4 字符缩进、制表符保存为空格）。
* 每行只写一条语句。

// 正确

private int a = 0;

private int b = 0;

// 错误

private int a = 0; private int b = 0;

* 每行只写一个声明。
* 如果连续行未自动缩进，将它们缩进一个制表符位（四个空格）。
* 在方法定义与属性定义之间添加至少一个空白行。

public int Test { get; set; }

// 正确，此注释行的上方留了一行空白，不能贴紧

public void Run()

{

}

* 使用括号突出表达式中的子句，如下面的代码所示。

if ((val1 > val2) && (val1 > val3))

{

// Take appropriate action.

}

* 任何时候都要避免出现多个空行。（例如，类型的成员之间不能有两个空行）。
* 避免出现多余的空格。可使用Visual Studio辅助检测（“编辑->高级->查看空白”）。
* Visual Studio代码格式化快捷键：Ctrl + K + D。

## 注释约定

* 将注释放在单独的行上，而非代码行的末尾，如下面的示例所示。

// 正确

// Define i and init.

int i=0;

// 错误

int i=0; // Define i and init.

* 如果注释为英文的话，则第一个单词的首字母应大写，即应符合英文的书写规范。
* 在注释分隔符 (//) 与注释文本之间插入一个空格，如下面的示例所示。

// The following declaration creates a query. It does not run

// the query.

* 确保所有public的成员都有必要的XML注释，从而提供有关其行为的适当说明。
* 标记用法参考：[建议标记](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/xmldoc/recommended-tags)

## 语言准则

### 修饰语

* 总是指定可见性，即使它是默认的。

// 正确

private string \_foo；

// 错误

string \_foo；

* 可见性应该是第一个修饰语。

// 正确

public abstract void Student();

// 错误

abstract public void Student();

### 字符串数据类型

若要在循环中追加字符串，尤其是在使用大量文本时，使用 StringBuilder 对象。

var phrase = "lalalalalalalalalalalalalalalalalalalalalalala";

var manyPhrases = new StringBuilder();

for (var i = 0; i < 10000; i++)

{

manyPhrases.Append(phrase);

}

//Console.WriteLine("tra" + manyPhrases);

### 类型转换

* 判断对象是否为给定类型时，使用is操作符来完成。

object o = new object();

if (o is string)

{

}

* object转换成引用类型时，使用as操作符来完成。

Employee myEmployee = myObject as Employee;

if (myEmployee != null)

{

}

**备注：**as操作符适用于引用类型或可以为null的类型。

* 字符串转值类型，例如int.TryParse而不是Int32.TryParse。
* 值类型之间转换，使用系统提供的Conver类所涉及的静态方法（例如Convert.ToInt32）。

### 隐式类型本地变量

* 当变量类型明显来自赋值的右侧时，或者当精度类型不重要时，可对本地变量进行隐式类型化。

var var1 = "This is clearly a string.";

var var2 = 27;

* 当类型并非明显来自赋值的右侧时，不要使用 [var](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/statements/declarations#implicitly-typed-local-variables)。不要假设类型明显来自方法名称。如果变量类型为 new 运算符或显式强制转换，则将其视为明显来自方法名称。

int var3 = Convert.ToInt32(Console.ReadLine());

int var4 = ExampleClass.ResultSoFar();

* 不要依靠变量名称来指定变量的类型。它可能不正确。在以下示例中，变量名称 inputInt 会产生误导性。它是字符串。

var inputInt = Console.ReadLine();

Console.WriteLine(inputInt);

* 避免使用 var 来代替 dynamic。如果想要进行运行时类型推理，可使用 dynamic。
* 使用隐式类型化来确定 for 循环中循环变量的类型。下面的示例在for语句中使用隐式类型化。

for (var i = 0; i < 10000; i++)

{

}

* 不要使用隐式类型化来确定 [foreach](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/statements/iteration-statements#the-foreach-statement) 循环中循环变量的类型。在大多数情况下，集合中的元素类型并不明显。不应仅依靠集合的名称来推断其元素的类型。下面的示例在 foreach 语句中使用显式类型化。

foreach (char ch in laugh)

{

if (ch == 'h')

Console.Write("H");

else

Console.Write(ch);

}

Console.WriteLine();

### 数组

当在声明行上初始化数组时，在简洁的语法中不能使用var替代string[]。

string[] vowels1 = { "a", "e", "i", "o", "u" };

如果使用显式实例化，则可以使用 var。

var vowels2 = new string[] { "a", "e", "i", "o", "u" };

### 委托

使用 [Func<>和Action<>](https://learn.microsoft.com/zh-cn/dotnet/standard/delegates-lambdas)，而不是定义委托类型。在类中，定义委托方法。

public static Action<string> ActionExample1 = x => Console.WriteLine(x.ToString());

public static Action<string, string> ActionExample2 = (x, y) => Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x => Convert.ToInt32(x);

public static Func<int, int, int> FuncExample2 = (x, y) => x + y;

使用 [Func<>和Action<>](https://learn.microsoft.com/zh-cn/dotnet/standard/delegates-lambdas)委托定义的签名来调用方法。

ActionExample1("string for x");

ActionExample2("string for x", "string for y");

Console.WriteLine($"The value is {FuncExample1("1")}");

Console.WriteLine($"The sum is {FuncExample2(1, 2)}");

在类中，定义委托类型和具有匹配签名的方法。

public delegate void Del(string message);

public static void DelMethod(string str)

{

Console.WriteLine("DelMethod argument: {0}", str);

}

创建委托类型的实例，然后调用该实例。以下声明显示了紧缩的语法。

Del exampleDel2 = DelMethod;

exampleDel2("Hey");

以下声明使用了完整的语法。

Del exampleDel1 = new Del(DelMethod);

exampleDel1("Hey");

### try-catch 和 using 语句正在异常处理中

* 对大多数异常处理使用 try-catch 语句。

static string GetValueFromArray(string[] array, int index)

{

try

{

return array[index];

}

catch (System.IndexOutOfRangeException ex)

{

Console.WriteLine("Index is out of range: {0}", index);

throw;

}

}

* 通过使用 using语句简化你的代码。如果具有 try-finally 语句（该语句中 finally 块的唯一代码是对 Dispose 方法的调用），使用 using 语句代替。

在以下示例中，try-finally 语句仅在 finally 块中调用 Dispose。

Font font1 = new Font("Arial", 10.0f);

try

{

byte charset = font1.GdiCharSet;

}

finally

{

if (font1 != null)

{

((IDisposable)font1).Dispose();

}

}

可以使用 using 语句执行相同的操作。

using (Font font2 = new Font("Arial", 10.0f))

{

byte charset2 = font2.GdiCharSet;

}

### && 和 || 运算符

若要通过跳过不必要的比较来避免异常并提高性能，在执行比较时使用[&&](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/boolean-logical-operators#conditional-logical-and-operator-)（而不是[&](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-and-operator-)）和[||](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/boolean-logical-operators#conditional-logical-or-operator-)（而不是[|](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/boolean-logical-operators#logical-or-operator-)），如下面的示例所示。

Console.Write("Enter a dividend: ");

int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");

int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))

{

Console.WriteLine("Quotient: {0}", dividend / divisor);

}

else

{

Console.WriteLine("Attempted division by 0 ends up here.");

}

如果除数为0，则if语句中的第二个子句将导致运行时错误。但是，当第一个表达式为false时，&&运算符将发生短路。也就是说，它并不评估第二个表达式。如果divisor为 0，则&运算符将同时计算这两个表达式，这会导致运行时错误。

### 事件处理

如果定义一个稍后不需要删除的事件处理程序，请使用lambda表达式。

public Form2()

{

this.Click += (s, e) =>

{

MessageBox.Show(

((MouseEventArgs)e).Location.ToString());

};

}

Lambda表达式缩短了以下传统定义。

public Form1()

{

this.Click += new EventHandler(Form1\_Click);

}

void Form1\_Click(object? sender, EventArgs e)

{

MessageBox.Show(((MouseEventArgs)e).Location.ToString());

}

### 静态成员

* 使用类名调用static成员：ClassName.StaticMember。这种做法通过明确静态访问使代码更易于阅读。不要使用派生类的名称来限定基类中定义的静态成员。编译该代码时，代码可读性具有误导性，如果向派生类添加具有相同名称的静态成员，代码可能会被破坏。
* 在静态字段上使用时，readonly应在static之后。

// 正确

private static readonly string s\_student;

// 错误

private readonly static string s\_student;

### LINQ 查询

* 在查询变量和范围变量的声明中使用隐式类型化。对齐[from](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/from-clause)子句下的查询子句，如下面的示例所示。

var seattleCustomers = from customer in customers

where customer.City == "Seattle"

select customer.Name;

* 在其他查询子句前面使用[where](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/where-clause)子句，确保后面的查询子句作用于经过缩减和筛选的一组数据。

var seattleCustomers2 = from customer in customers

where customer.City == "Seattle"

orderby customer.Name

select customer;

* 使用多行from子句代替[join](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/join-clause)子句来访问内部集合。例如，Student对象的集合可能包含测验分数的集合。当执行以下查询时，它返回高于90的分数，并返回得到该分数的学生的姓氏。

var scoreQuery = from student in students

from score in student.Scores!

where score > 90

select new { Last = student.LastName, score };

### goto语句

不建议使用。

## 其它参考

[C#编码约定](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/coding-style/coding-conventions)

[C#语言规范](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/language-specification/readme)

## 附1 标识符编写约定表

|  |  |  |
| --- | --- | --- |
| **标识符** | **大小写** | **示例** |
| 命名空间 | Pascal | namespace Com.Techstar.ProductionCenter |
| 类型 | Pascal | public class DevsList |
| 接口 | Pascal | public interface ITableModel |
| 方法 | Pascal | public void UpdateData() |
| 属性 | Pascal | Public int Length{…} |
| 事件 | Pascal | public event EventHandler Changed |
| 私有字段 | Camel | private string fieldName |
| 非私有字段 | Pascal | public string FieldName |
| 枚举值 | Pascal | FileMode{Append} |
| 参数 | Camel | public void UpdateData(string fieldName) |
| 局部变量 | Camel | string fieldName |

## 附2 组件名称缩写列表

缩写的基本原则是取组件类名各单词的第一个字母，如果只有一个单词，则去掉其中的元音，留下辅音。缩写全部为小写。

|  |  |  |
| --- | --- | --- |
| **组件类型** | **缩写** | **例子** |
| Label | lbl | lblNote |
| TextBox | txt | txtName |
| Button | btn | btnOK |
| ImageButton | ib | ibOK |
| LinkButton | lb | lbJump |
| HyperLink | hl | hlJump |
| DropDownList | ddl | ddlList |
| CheckBox | cb | cbChoice |
| CheckBoxList | cbl | cblGroup |
| RadioButton | rb | rbChoice |
| RadioButtonList | rbl | rblGroup |
| Image | img | imgBeauty |
| Panel | pnl | pnlTree |
| TreeView | tv | tvUnit |
| ImageDateTimeInput | dti | dtiStart |
| ComboBox | cb | cbList |
| PageBar | pb | pbMaster |