**构造函数（C# 编程指南）**

任何时候，只要创建[类](https://msdn.microsoft.com/zh-CN/library/0b0thckt.aspx)或[结构](https://msdn.microsoft.com/zh-CN/library/ah19swz4.aspx)，就会调用它的构造函数。类或结构可能有多个接受不同参数的构造函数。构造函数使得程序员可设置默认值、限制实例化以及编写灵活且便于阅读的代码。有关更多信息和示例，请参见[使用构造函数（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/ms173115.aspx)和[实例构造函数（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/k6sa6h87.aspx)。

如果您没有为对象提供构造函数，则默认情况下 C# 将创建一个构造函数，该构造函数实例化对象，并将成员变量设置为[默认值表（C# 参考）](https://msdn.microsoft.com/zh-CN/library/83fhsxwc.aspx)中列出的默认值。有关更多信息和示例，请参见[实例构造函数（C# 编程指南）](https://msdn.microsoft.com/zh-CN/library/k6sa6h87.aspx)。

静态类和结构也可以有构造函数。

## [使用构造函数](https://msdn.microsoft.com/zh-CN/library/ms173115.aspx)

实例化[类](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/class)或[结构](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct)时，将会调用其构造函数。 构造函数与该类或结构具有相同名称，并且通常初始化新对象的数据成员。

在下面的示例中，通过使用简单构造函数定义了一个名为 Taxi 的类。 然后使用 [new](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/new-operator) 运算符对该类进行实例化。 在为新对象分配内存之后，new 运算符立即调用 Taxi 构造函数。

C#复制

public class Taxi

{

public bool IsInitialized;

public Taxi()

{

IsInitialized = true;

}

}

class TestTaxi

{

static void Main()

{

Taxi t = new Taxi();

Console.WriteLine(t.IsInitialized);

}

}

不带任何参数的构造函数称为“无参数构造函数”。 每当使用 new 运算符实例化对象且不为 new 提供任何参数时，会调用无参数构造函数。 C# 12 引入了主构造函数。 主构造函数指定为初始化新对象而必须提供的参数。 有关详细信息，请参阅[实例构造函数](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors)。

除非类是[静态](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/static)的，否则 C# 编译器将为无构造函数的类提供一个公共的无参数构造函数，以便该类可以实例化。 有关详细信息，请参阅[静态类和静态类成员](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members)。

通过将构造函数设置为私有构造函数，可以阻止类被实例化，如下所示：

C#复制

class NLog

{

// Private Constructor:

private NLog() { }

public static double e = Math.E; //2.71828...

}

有关详细信息，请参阅[私有构造函数](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/private-constructors)。

[结构](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct)类型的构造函数类似于类构造函数。 使用 new 实例化结构类型时，将调用构造函数。 将 struct 设置为其 default 值时，运行时会将结构中的所有内存初始化为 0。 在 C# 10 之前，structs 不能包含显式无参数构造函数，因为编译器会自动提供一个。 有关详细信息，请参阅[结构类型](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct)一文的[结构初始化和默认值](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct#struct-initialization-and-default-values)部分。

以下代码使用 [Int32](https://learn.microsoft.com/zh-cn/dotnet/api/system.int32) 的无参数构造函数，因此可确保整数已初始化：

C#复制

int i = new int();

Console.WriteLine(i);

但是，下面的代码会导致编译器错误，因为它不使用 new，而且尝试使用尚未初始化的对象：

C#复制

int i;

Console.WriteLine(i);

或者，可将基于 structs 的对象（包括所有内置数值类型）初始化或赋值后使用，如下面的示例所示：

C#复制

int a = 44; // Initialize the value type...

int b;

b = 33; // Or assign it before using it.

Console.WriteLine("{0}, {1}", a, b);

类和结构都可以定义采用参数的构造函数，包括[主构造函数](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors#primary-constructors)。 必须通过 new 语句或 [base](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/base) 语句调用带参数的构造函数。 类和结构还可以定义多个构造函数，并且二者均无需定义无参数构造函数。 例如：

C#复制

public class Employee

{

public int Salary;

public Employee() { }

public Employee(int annualSalary)

{

Salary = annualSalary;

}

public Employee(int weeklySalary, int numberOfWeeks)

{

Salary = weeklySalary \* numberOfWeeks;

}

}

可使用下面任一语句创建此类：

C#复制

Employee e1 = new Employee(30000);

Employee e2 = new Employee(500, 52);

构造函数可以使用 base 关键字调用基类的构造函数。 例如：

C#复制

public class Manager : Employee

{

public Manager(int annualSalary)

: base(annualSalary)

{

//Add further instructions here.

}

}

在此示例中，在执行构造函数块之前调用基类的构造函数。 base 关键字可带参数使用，也可不带参数使用。 构造函数的任何参数都可用作 base 的参数，或用作表达式的一部分。 有关详细信息，请参阅 [base](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/base)。

在派生类中，如果不使用 base 关键字来显式调用基类构造函数，则将隐式调用无参数构造函数（若有）。 下面的构造函数声明等效：

C#复制

public Manager(int initialData)

{

//Add further instructions here.

}

C#复制

public Manager(int initialData)

: base()

{

//Add further instructions here.

}

如果基类没有提供无参数构造函数，派生类必须使用 base 显式调用基类构造函数。

构造函数可以使用 [this](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/this) 关键字调用同一对象中的另一构造函数。 和 base 一样，this 可带参数使用也可不带参数使用，构造函数中的任何参数都可用作 this 的参数，或者用作表达式的一部分。 例如，可以使用 this 重写前一示例中的第二个构造函数：

C#复制

public Employee(int weeklySalary, int numberOfWeeks)

: this(weeklySalary \* numberOfWeeks)

{

}

上一示例中使用 this 关键字会导致此构造函数被调用：

C#复制

public Employee(int annualSalary)

{

Salary = annualSalary;

}

可以将构造函数标记为[public](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)、[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) 或 [private protected](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private-protected)。 这些访问修饰符定义类的用户构造该类的方式。 有关详细信息，请参阅[访问修饰符](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers)。

可使用 [static](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/static) 关键字将构造函数声明为静态构造函数。 在访问任何静态字段之前，都将自动调用静态构造函数，它们用于初始化静态类成员。 有关详细信息，请参阅[静态构造函数](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/static-constructors)。

## [实例构造函数](https://msdn.microsoft.com/zh-CN/library/k6sa6h87.aspx)

声明一个实例构造函数，以指定在使用 [new 表达式](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/new-operator)创建某个类型的新实例时所执行的代码。 要初始化[静态](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/static)类或非静态类中的静态变量，可以定义[静态构造函数](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/static-constructors)。

如以下示例所示，可以在一种类型中声明多个实例构造函数：

C#复制

class Coords

{

public Coords()

: this(0, 0)

{ }

public Coords(int x, int y)

{

X = x;

Y = y;

}

public int X { get; set; }

public int Y { get; set; }

public override string ToString() => $"({X},{Y})";

}

class Example

{

static void Main()

{

var p1 = new Coords();

Console.WriteLine($"Coords #1 at {p1}");

// Output: Coords #1 at (0,0)

var p2 = new Coords(5, 3);

Console.WriteLine($"Coords #2 at {p2}");

// Output: Coords #2 at (5,3)

}

}

在上个示例中，第一个无参数构造函数调用两个参数都等于 0 的第二个构造函数。 要执行此操作，请使用 this 关键字。

在派生类中声明实例构造函数时，可以调用基类的构造函数。 为此，请使用 base 关键字，如以下示例所示：

C#复制

abstract class Shape

{

public const double pi = Math.PI;

protected double x, y;

public Shape(double x, double y)

{

this.x = x;

this.y = y;

}

public abstract double Area();

}

class Circle : Shape

{

public Circle(double radius)

: base(radius, 0)

{ }

public override double Area() => pi \* x \* x;

}

class Cylinder : Circle

{

public Cylinder(double radius, double height)

: base(radius)

{

y = height;

}

public override double Area() => (2 \* base.Area()) + (2 \* pi \* x \* y);

}

class Example

{

static void Main()

{

double radius = 2.5;

double height = 3.0;

var ring = new Circle(radius);

Console.WriteLine($"Area of the circle = {ring.Area():F2}");

// Output: Area of the circle = 19.63

var tube = new Cylinder(radius, height);

Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");

// Output: Area of the cylinder = 86.39

}

}

### 无参数构造函数

如果某个类没有显式实例构造函数，C# 将提供可用于实例化该类实例的无参数构造函数，如以下示例所示：

C#复制

public class Person

{

public int age;

public string name = "unknown";

}

class Example

{

static void Main()

{

var person = new Person();

Console.WriteLine($"Name: {person.name}, Age: {person.age}");

// Output: Name: unknown, Age: 0

}

}

该构造函数根据相应的初始值设定项初始化实例字段和属性。 如果字段或属性没有初始值设定项，其值将设置为字段或属性类型的[默认值](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/default-values)。 如果在某个类中声明至少一个实例构造函数，则 C# 不提供无参数构造函数。

structure 类型始终提供无参数构造函数： 无参数构造函数是可生成某种类型的默认值的隐式无参数构造函数或显式声明的无参数构造函数。 有关详细信息，请参阅[结构类型](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct)一文的[结构初始化和默认值](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/struct#struct-initialization-and-default-values)部分。

### 主构造函数

从 C# 12 开始，可以在类和结构中声明主构造函数。 将任何参数放在类型名称后面的括号中：

C#复制

public class NamedItem(string name)

{

public string Name => name;

}

主构造函数的参数位于声明类型的整个主体中。 它们可以初始化属性或字段。 它们可用作方法或局部函数中的变量。 它们可以传递给基本构造函数。

主构造函数指示这些参数对于类型的任何实例是必需的。 任何显式编写的构造函数都必须使用 this(...) 初始化表达式语法来调用主构造函数。 这可确保主构造函数参数绝对由所有构造函数分配。 对于任何 class 类型（包括 record class 类型），当主构造函数存在时，不会发出隐式无参数构造函数。 对于任何 struct 类型（包括 record struct 类型），始终发出隐式无参数构造函数，并始终将所有字段（包括主构造函数参数）初始化为 0 位模式。 如果编写显式无参数构造函数，则必须调用主构造函数。 在这种情况下，可以为主构造函数参数指定不同的值。 以下代码演示主构造函数的示例。

C#复制

// name isn't captured in Widget.

// width, height, and depth are captured as private fields

public class Widget(string name, int width, int height, int depth) : NamedItem(name)

{

public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

public int WidthInCM => width;

public int HeightInCM => height;

public int DepthInCM => depth;

public int Volume => width \* height \* depth;

}

可以通过在属性上指定 method: 目标，可以将属性添加到合成的主要构造函数方法：

C#复制

[method: MyAttribute]

public class TaggedWidget(string name)

{

// details elided

}

如果未指定 method 目标，则属性将放置在类上而不是方法上。

在 class 和 struct 类型中，主构造函数参数在类型主体中的任意位置可用。 可将参数实现为捕获的专用字段。 如果对参数的唯一引用是初始化表达式和构造函数调用，则不会在专用字段中捕获该参数。 在该类型的其他成员中使用会导致编译器在专用字段中捕获该参数。

如果该类型包含 record 修饰符，则编译器将合成一个与主构造函数参数同名的公共属性。 对于 record class 类型，如果主构造函数参数使用与基主构造函数相同的名称，则该属性是基 record class 类型的公共属性。 它在派生的 record class 类型中不会重复。 不会为非 record 类型生成这些属性。

## [私有构造函数](https://msdn.microsoft.com/zh-CN/library/kcfb85a6.aspx)

私有构造函数是一种特殊的实例构造函数。 它通常用于只包含静态成员的类中。 如果类具有一个或多个私有构造函数而没有公共构造函数，则其他类（除嵌套类外）无法创建该类的实例。 例如：

C#复制

class NLog

{

// Private Constructor:

private NLog() { }

public static double e = Math.E; //2.71828...

}

声明空构造函数可阻止自动生成无参数构造函数。 请注意，如果不对构造函数使用访问修饰符，则在默认情况下它仍为私有构造函数。 但是，通常会显式地使用 [private](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/private) 修饰符来清楚地表明该类不能被实例化。

当没有实例字段或实例方法（例如 [Math](https://learn.microsoft.com/zh-cn/dotnet/api/system.math) 类）时或者当调用方法以获得类的实例时，私有构造函数可用于阻止创建类的实例。 如果类中的所有方法都是静态的，可考虑使整个类成为静态的。 有关详细信息，请参阅[静态类和静态类成员](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members)。

## 示例

下面是使用私有构造函数的类的示例。

C#复制

public class Counter

{

private Counter() { }

public static int currentCount;

public static int IncrementCount()

{

return ++currentCount;

}

}

class TestCounter

{

static void Main()

{

// If you uncomment the following statement, it will generate

// an error because the constructor is inaccessible:

// Counter aCounter = new Counter(); // Error

Counter.currentCount = 100;

Counter.IncrementCount();

Console.WriteLine("New count: {0}", Counter.currentCount);

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

// Output: New count: 101

请注意，如果取消注释该示例中的以下语句，它将生成一个错误，因为该构造函数受其保护级别的限制而不可访问：

C#

// Counter aCounter = new Counter(); // Error

## [静态构造函数](https://msdn.microsoft.com/zh-CN/library/k9x6w0hc.aspx)

静态构造函数用于初始化任何[静态](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/static)数据，或执行仅需执行一次的特定操作。 它会在创建第一个实例或引用任何静态成员之前自动调用。 静态构造函数最多调用一次。

C#复制

class SimpleClass

{

// Static variable that must be initialized at run time.

static readonly long baseline;

// Static constructor is called at most one time, before any

// instance constructor is invoked or member is accessed.

static SimpleClass()

{

baseline = DateTime.Now.Ticks;

}

}

有多个操作在静态初始化时执行。 这些操作按以下顺序执行：

1. 静态字段设置为 0。 通常由运行时执行此初始化。
2. 静态字段初始值设定项运行。 派生程度最高类型的静态字段初始值设定项运行。
3. 基类型静态字段初始值设定项运行。 以直接基开头从每个基类型到 [System.Object](https://learn.microsoft.com/zh-cn/dotnet/api/system.object) 的静态字段初始值设定项。
4. 所有静态构造函数运行。 任何静态构造函数都会运行，从 [Object.Object](https://learn.microsoft.com/zh-cn/dotnet/api/system.object.-ctor) 的最终基类到每一个基类，再到类型。 静态构造函数执行的顺序没有指定。 但是，在创建任何实例之前，层次结构中的所有静态构造函数都会运行。

**重要**

静态构造函数在创建任何实例之前运行的规则有一个重要例外。 如果静态字段初始值设定项创建了该类型的实例，那么该初始值设定项（包括对实例构造函数的任何调用）将在静态构造函数运行之前运行。 这在单一实例模式中最为常见，如以下示例所示：

C#复制

public class Singleton

{

// Static field initializer calls instance constructor.

private static Singleton instance = new Singleton();

private Singleton()

{

Console.WriteLine("Executes before static constructor.");

}

static Singleton()

{

Console.WriteLine("Executes after instance constructor.");

}

public static Singleton Instance => instance;

}

[模块初始化表达式](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/attributes/general#moduleinitializer-attribute)可以替代静态构造函数。 有关详细信息，请参阅[模块初始化表达式的规范](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/proposals/csharp-9.0/module-initializers)。

### 备注

静态构造函数具有以下属性：

* 静态构造函数不使用访问修饰符或不具有参数。
* 类或结构只能有一个静态构造函数。
* 静态构造函数不能继承或重载。
* 静态构造函数不能直接调用，只能由公共语言运行时 (CLR) 调用。 它会自动调用。
* 用户无法控制在程序中执行静态构造函数的时间。
* 自动调用静态构造函数。 它在创建第一个实例或引用该类（不是其基类）中声明的任何静态成员之前初始化[类](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/class)。 静态构造函数在实例构造函数之前运行。 如果静态构造函数类中存在静态字段变量初始值设定项，它们将以在类声明中显示的文本顺序运行。 初始值设定项紧接着静态构造函数之前运行。
* 如果未提供静态构造函数来初始化静态字段，会将所有静态字段初始化为其默认值，如 [C# 类型的默认值](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/builtin-types/default-values)中所列。
* 如果静态构造函数引发异常，运行时不会再次调用该函数，并且类型在应用程序域的生存期内会保持未初始化状态。 大多数情况下，当静态构造函数无法实例化一个类型时，或者当静态构造函数中发生未经处理的异常时，将引发 [TypeInitializationException](https://learn.microsoft.com/zh-cn/dotnet/api/system.typeinitializationexception) 异常。 对于未在源代码中显式定义的静态构造函数，故障排除可能需要检查中间语言 (IL) 代码。
* 静态构造函数的存在将防止添加 [BeforeFieldInit](https://learn.microsoft.com/zh-cn/dotnet/api/system.reflection.typeattributes#system-reflection-typeattributes-beforefieldinit) 类型属性。 这将限制运行时优化。
* 声明为 static readonly 的字段只能在其声明时或在静态构造函数中分配。 如果不需要显式静态构造函数，请在声明时初始化静态字段，而不是通过静态构造函数，以实现更好的运行时优化。
* 运行时在单个应用程序域中多次调用静态构造函数。 该调用是基于特定类型的类在锁定区域中进行的。 静态构造函数的主体中不需要额外的锁定机制。 若要避免死锁的风险，请勿阻止静态构造函数和初始值设定项中的当前线程。 例如，不要等待任务、线程、等待句柄或事件，不要获取锁定，也不要执行阻止并行操作，如并行循环、Parallel.Invoke 和并行 LINQ 查询。

**备注**

尽管不可直接访问，但应记录显式静态构造函数的存在，以帮助故障排除初始化异常。

#### 用法

* 静态构造函数的一种典型用法是在类使用日志文件且将构造函数用于将条目写入到此文件中时使用。
* 静态构造函数对于创建非托管代码的包装类也非常有用，这种情况下构造函数可调用 LoadLibrary 方法。
* 也可在静态构造函数中轻松地对无法在编译时通过类型参数约束检查的类型参数强制执行运行时检查。

### 示例

在此示例中，类 Bus 具有静态构造函数。 创建 Bus 的第一个实例 (bus1) 时，将调用该静态构造函数，以便初始化类。 示例输出验证即使创建了两个 Bus 的实例，静态构造函数也仅运行一次，并且在实例构造函数运行前运行。

C#复制

public class Bus

{

// Static variable used by all Bus instances.

// Represents the time the first bus of the day starts its route.

protected static readonly DateTime globalStartTime;

// Property for the number of each bus.

protected int RouteNumber { get; set; }

// Static constructor to initialize the static variable.

// It is invoked before the first instance constructor is run.

static Bus()

{

globalStartTime = DateTime.Now;

// The following statement produces the first line of output,

// and the line occurs only once.

Console.WriteLine("Static constructor sets global start time to {0}",

globalStartTime.ToLongTimeString());

}

// Instance constructor.

public Bus(int routeNum)

{

RouteNumber = routeNum;

Console.WriteLine("Bus #{0} is created.", RouteNumber);

}

// Instance method.

public void Drive()

{

TimeSpan elapsedTime = DateTime.Now - globalStartTime;

// For demonstration purposes we treat milliseconds as minutes to simulate

// actual bus times. Do not do this in your actual bus schedule program!

Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",

this.RouteNumber,

elapsedTime.Milliseconds,

globalStartTime.ToShortTimeString());

}

}

class TestBus

{

static void Main()

{

// The creation of this instance activates the static constructor.

Bus bus1 = new Bus(71);

// Create a second bus.

Bus bus2 = new Bus(72);

// Send bus1 on its way.

bus1.Drive();

// Wait for bus2 to warm up.

System.Threading.Thread.Sleep(25);

// Send bus2 on its way.

bus2.Drive();

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

/\* Sample output:

Static constructor sets global start time to 3:57:08 PM.

Bus #71 is created.

Bus #72 is created.

71 is starting its route 6.00 minutes after global start time 3:57 PM.

72 is starting its route 31.00 minutes after global start time 3:57 PM.

\*/

### 另请参阅

* [C# 类型系统](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/types/)
* [构造函数](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/constructors)
* [静态类和静态类成员](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members)
* [终结器](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/finalizers)
* [构造函数设计准则](https://learn.microsoft.com/zh-cn/dotnet/standard/design-guidelines/constructor#type-constructor-guidelines)
* [安全警告 - CA2121：静态构造函数应为私有](https://learn.microsoft.com/zh-cn/visualstudio/code-quality/ca2121-static-constructors-should-be-private)
* [模块初始值设定项](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/attributes/general#moduleinitializer-attribute)

## [如何：编写复制构造函数](https://msdn.microsoft.com/zh-CN/library/ms173116.aspx)

C # [记录](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/types/records)为对象提供复制构造函数，但对于类，你必须自行编写。

**重要**

编写适用于类层次结构中所有派生类型的复制构造函数可能很困难。 如果类不是 sealed，则强烈建议考虑创建 record class 类型的层次结构，以使用编译器合成的复制构造函数。

### 示例

在下面的示例中，Person[类](https://learn.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/class)定义一个复制构造函数，该函数使用 Person 的实例作为其参数。 该参数的属性值分配给 Person 的新实例的属性。 该代码包含一个备用复制构造函数，该函数发送要复制到该类的实例构造函数的实例的 Name 和 Age 属性。 Person 类为 sealed，因此无法通过仅复制基类来声明可能会引发错误的派生类型。

C#复制

public sealed class Person

{

// Copy constructor.

public Person(Person previousPerson)

{

Name = previousPerson.Name;

Age = previousPerson.Age;

}

//// Alternate copy constructor calls the instance constructor.

//public Person(Person previousPerson)

// : this(previousPerson.Name, previousPerson.Age)

//{

//}

// Instance constructor.

public Person(string name, int age)

{

Name = name;

Age = age;

}

public int Age { get; set; }

public string Name { get; set; }

public string Details()

{

return Name + " is " + Age.ToString();

}

}

class TestPerson

{

static void Main()

{

// Create a Person object by using the instance constructor.

Person person1 = new Person("George", 40);

// Create another Person object, copying person1.

Person person2 = new Person(person1);

// Change each person's age.

person1.Age = 39;

person2.Age = 41;

// Change person2's name.

person2.Name = "Charles";

// Show details to verify that the name and age fields are distinct.

Console.WriteLine(person1.Details());

Console.WriteLine(person2.Details());

// Keep the console window open in debug mode.

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

Console.ReadKey();

}

}

// Output:

// George is 39

// Charles is 41

### 请参阅

* [ICloneable](https://learn.microsoft.com/zh-cn/dotnet/api/system.icloneable)
* [记录](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/types/records)
* [C# 类型系统](https://learn.microsoft.com/zh-cn/dotnet/csharp/fundamentals/types/)
* [构造函数](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/constructors)
* [终结器](https://learn.microsoft.com/zh-cn/dotnet/csharp/programming-guide/classes-and-structs/finalizers)

# 析构函数（C# 编程指南）

析构函数用于析构类的实例。

## 备注

* 不能在结构中定义析构函数。只能对类使用析构函数。
* 一个类只能有一个析构函数。
* 无法继承或重载析构函数。
* 无法调用析构函数。它们是被自动调用的。
* 析构函数既没有修饰符，也没有参数。

例如，下面是类 Car 的析构函数的声明：

class Car

{

~Car() // destructor

{

// cleanup statements...

}

}

该析构函数隐式地对对象的基类调用 [Finalize](https://msdn.microsoft.com/zh-cn/library/system.object.finalize.aspx)。这样，前面的析构函数代码被隐式地转换为以下代码：

protected override void Finalize()

{

try

{

// Cleanup statements...

}

finally

{

base.Finalize();

}

}

这意味着对继承链中的所有实例递归地调用 Finalize 方法（从派生程度最大的到派生程度最小的）。

| **https://wizardforcel.gitbooks.io/msdn-csharp/content/img/note.jpg 注意** |
| --- |
| 不应使用空析构函数。如果类包含析构函数，Finalize 队列中则会创建一个项。调用析构函数时，将调用垃圾回收器来处理该队列。如果析构函数为空，只会导致不必要的性能损失。 |

程序员无法控制何时调用析构函数，因为这是由垃圾回收器决定的。垃圾回收器检查是否存在应用程序不再使用的对象。如果垃圾回收器认为某个对象符合析构，则调用析构函数（如果有）并回收用来存储此对象的内存。程序退出时也会调用析构函数。

可以通过调用 [Collect](https://msdn.microsoft.com/zh-cn/library/system.gc.collect.aspx) 强制进行垃圾回收，但大多数情况下应避免这样做，因为这样会导致性能问题。

## 使用析构函数释放资源

通常，与运行时不进行垃圾回收的开发语言相比，C# 无需太多的内存管理。这是因为 .NET Framework 垃圾回收器会隐式地管理对象的内存分配和释放。但是，当应用程序封装窗口、文件和网络连接这类非托管资源时，应当使用析构函数释放这些资源。当对象符合析构时，垃圾回收器将运行对象的 Finalize 方法。

## 资源的显式释放

如果您的应用程序在使用昂贵的外部资源，我们还建议您提供一种在垃圾回收器释放对象前显式地释放资源的方式。可通过实现来自 [IDisposable](https://msdn.microsoft.com/zh-cn/library/system.idisposable.aspx) 接口的 Dispose 方法来完成这一点，该方法为对象执行必要的清理。这样可大大提高应用程序的性能。即使有这种对资源的显式控制，析构函数也是一种保护措施，可用来在对 Dispose 方法的调用失败时清理资源。

有关清理资源的更多详细信息，请参见下列主题：

* [Cleaning Up Unmanaged Resources](https://msdn.microsoft.com/zh-cn/library/498928w2.aspx)
* [Implementing a Dispose Method](https://msdn.microsoft.com/zh-cn/library/fs2xkftw.aspx)
* [using 语句（C# 参考）](https://msdn.microsoft.com/zh-cn/library/yh598w02.aspx)

下面的示例创建三个类，这三个类构成了一个继承链。类 First 是基类，Second 是从 First 派生的，而 Third 是从 Second 派生的。这三个类都有析构函数。在 Main() 中，创建了派生程度最大的类的实例。注意：程序运行时，这三个类的析构函数将自动被调用，并且是按照从派生程度最大的到派生程度最小的次序调用。

class First

{

~First()

{

System.Diagnostics.Trace.WriteLine("First's destructor is called.");

}

}

class Second : First

{

~Second()

{

System.Diagnostics.Trace.WriteLine("Second's destructor is called.");

}

}

class Third : Second

{

~Third()

{

System.Diagnostics.Trace.WriteLine("Third's destructor is called.");

}

}

class TestDestructors

{

static void Main()

{

Third t = new Third();

}

}

/\* Output (to VS Output Window):

Third's destructor is called.

Second's destructor is called.

First's destructor is called.

\*/