

Windows 窗体文档

了解如何在 .NET 上使用适用于 Windows 的开源图形用户界面 Windows 窗体。

了解 Windows 窗体

新变化

[新增功能](#)

概述

[Windows 窗体概述](#)

教程

[创建新应用](#)

[从 .NET Framework 迁移](#)

下载

[Visual Studio 2022](#)

控制

概述

[关于控件](#)

概念

[布局](#)

操作指南

[添加控件](#)

[添加访问权限 \(快捷键\)](#)

键盘输入

① 概述

[关于键盘](#)

② 概念

[键盘事件](#)

③ 操作指南

[处理键盘输入](#)

[修改密钥事件](#)

鼠标输入

① 概述

[关于鼠标](#)

② 概念

[鼠标事件](#)

[拖放](#)

③ 操作指南

[管理游标](#)

[模拟鼠标事件](#)

.NET 9 Windows 窗体中的新增功能

项目 • 2024/12/03

本文介绍 .NET 9 Windows 窗体中的新增功能。

异步表单

① 重要

此功能集是实验性的。

新式应用需要异步通信模型。 随着 .NET 上的 Windows 窗体的发展，更多的组件需要封装到在 `async` UI 线程上运行的方法。 例如，[WebView2](#)、[本机 Windows 10 和 Windows 11 API 等](#) 控件，或语义内核[等](#) 新式异步库。 另一种方案是共享基于其他 UI 堆栈（如 WPF、WinUI 或 .NET MAUI）Windows 窗体构建 `async` 的 MVVM ViewModel。

下面列出了为支持异步方案而添加的新方法：

- [System.Windows.Forms.Form.ShowAsync](#)
- [Form.ShowDialogAsync](#)
- [TaskDialog.ShowDialogAsync](#)
- [Control.InvokeAsync](#) （此 API 不是实验性的。）

此 API 在编译器错误后面受到保护，因为它是实验性的。 若要禁止显示错误并启用对 API 的访问，请将以下内容 `PropertyGroup` 添加到项目文件：

XML

```
<PropertyGroup>
    <NoWarn>$(NoWarn);WF05002</NoWarn>
</PropertyGroup>
```

💡 提示

有关如何禁止显示此规则的详细信息，请参阅 [编译器错误WF05002](#)。

不再支持 BinaryFormatter

`BinaryFormatter` 被视为不安全，因为它容易受到反序列化攻击，这可能导致拒绝服务 (DoS)、信息泄露或远程代码执行。它在反序列化漏洞被充分理解之前实现，其设计不遵循现代安全最佳做法。

从 .NET 9 开始，其实现已被删除，以防止这些安全风险。使用时 `BinaryFormatter`，`PlatformNotSupportedException` 将引发异常。

Windows 窗体在许多方案中使用 `BinaryFormatter`，例如，在序列化剪贴板和拖放操作的数据时，最重要的是 Windows 窗体设计器。在内部，Windows 窗体继续使用更安全的 `BinaryFormatter` 子集来处理具有已知类型集的特定用例。

适用于 .NET 9 的 Windows 窗体随分析器一起交付，可帮助识别不知不觉参与二进制序列化的时间。

有关详细信息 `BinaryFormatter`，请参阅 [BinaryFormatter Windows 窗体 迁移指南](#)。

深色模式

① 重要

此功能集是实验性的。

已将对深色模式的初步支持添加到 Windows 窗体，目标是在 .NET 10 中完成支持。当颜色模式更改时，将 `SystemColors` 更改颜色模式以匹配。应用的颜色模式可以设置为以下值之一：

- `SystemColorMode.Classic` - (默认值) 浅色模式，与以前版本的 Windows 窗体相同。
- `SystemColorMode.System` - 尊重 Windows 设置的浅色或深色模式。
- `SystemColorMode.Dark` - 使用深色模式。

若要应用颜色模式，请调用 `Application.SetColorMode(SystemColorMode)` 程序启动代码：

C#

```
namespace MyExampleProject;

static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
```

```
static void Main()
{
    // To customize application configuration such as set high DPI
    settings or default font,
    // see https://aka.ms/applicationconfiguration.
    ApplicationConfiguration.Initialize();
    Application.SetColorMode(SystemColorMode.Dark);
    Application.Run(new Form1());
}
}
```

VB

Friend Module Program

```
<STAThread()
Friend Sub Main(args As String())
    Application.SetHighDpiMode(HighDpiMode.SystemAware)
    Application.EnableVisualStyles()
    Application.SetCompatibleTextRenderingDefault(False)
    Application.SetColorMode(SystemColorMode.Dark)
    Application.Run(New Form1)
End Sub

End Module
```

此 API 在编译器错误后面受到保护，因为它是实验性的。 若要禁止显示错误并启用对 API 的访问，请将以下内容 `PropertyGroup` 添加到项目文件：

XML

```
<PropertyGroup>
    <NoWarn>$(NoWarn);WF05001</NoWarn>
</PropertyGroup>
```

💡 提示

有关如何禁止显示此规则的详细信息，请参阅 [编译器错误WF05001](#)。

FolderBrowserDialog 增强功能

`FolderBrowserDialog` 现在支持选择存储在数组中的 `SelectedPaths` 多个文件夹。 若要启用多个文件夹，请设置为 `Multiselect true`。

System.Drawing 新功能和增强功能

System.Drawing 库进行了许多改进，包括包装 GDI+ 效果、支持 `ReadOnlySpan` 和更好的互操作代码生成。

System.Drawing 支持 GDI+ 效果

System.Drawing 库现在支持 GDI+ 位图效果，例如模糊和淡色。效果是 GDI+ 的一部分，但直到现在才通过 System.Drawing 公开效果。

通过调用 `Bitmap.ApplyEffect(Effect, Rectangle)` 该方法将效果应用于该 `Bitmap` 效果。为要应用效果的区域提供效果和可选 `Rectangle` 效果。用于 `Rectangle.Empty` 处理整个映像。

命名空间 `System.Drawing.Imaging.Effects` 包含可应用的效果：

- `BlackSaturationCurveEffect`
- `BlurEffect`
- `BrightnessContrastEffect`
- `ColorBalanceEffect`
- `ColorCurveEffect`
- `ColorLookupTableEffect`
- `ColorMatrixEffect`
- `ContrastCurveEffect`
- `CurveChannel`
- `DensityCurveEffect`
- `ExposureCurveEffect`
- `GrayScaleEffect`
- `HighlightCurveEffect`
- `InvertEffect`
- `LevelsEffect`
- `MidtoneCurveEffect`
- `ShadowCurveEffect`
- `SharpenEffect`
- `TintEffect`
- `VividEffect`
- `WhiteSaturationCurveEffect`

System.Drawing 支持 Span

已增强接受数组的许多方法也接受 `ReadOnlySpan`。例如，方法（例如 `GraphicsPath.AddLines(ReadOnlySpan<Point>)`, `Graphics.DrawLines(Pen, ReadOnlySpan<Point>)` 和 `DrawPolygon(Pen, ReadOnlySpan<Point>)`）接受数组或 `ReadOnlySpan`。

使用 CsWin32 进行互操作

所有互操作代码已替换为 [C# P/Invoke 源生成器 CsWin32](#)。

ToolStrip

以下改进已添加到 `ToolStrip` 控件中 `ToolStripItem`。

- 已向 `ToolStrip` 添加了一个新属性。

设置为 “`true`” 时，控件可以在窗体未对焦时与控件交互。

在发布 .NET Core 3.1 时，`Menu` 删除了所有相关控件，例如 `MainMenu` 和 `MenuItem` / `ToolStrip` 应 `ToolStripMenuItem` 改为使用。但是，`ToolStripItem` 基类没有替代事件的 `MenuItem.Select` 基类 `ToolStripMenuItem`。当鼠标或键盘用于突出显示项时，将引发此事件。

.NET 9 已添加 `ToolStripItem.SelectedChanged`，可用于检测何时突出显示菜单项。

.NET 8 Windows 窗体中的新增功能

项目 • 2024/11/11

本文介绍 .NET 8 中一些新的 Windows 窗体功能和增强功能。

从 .NET Framework 迁移到 .NET 8 时，应注意一些中断性变更。有关详细信息，请参阅 [Windows 窗体中的重大更改](#)。

数据绑定改进

新的数据绑定引擎在 .NET 7 中处于预览状态，现在已在 .NET 8 中完全启用。尽管与现有的 Windows 窗体数据绑定引擎不一样广泛，但此新引擎在 WPF 之后建模，因此可以更轻松地实现 MVVM 设计原则。

增强的数据绑定功能使充分利用 MVVM 模式和在 Windows 窗体中使用 ViewModels 中的对象关系映射器变得更加简单。这减少了代码隐藏文件中的代码量。更重要的是，它支持在 Windows 窗体和其他 .NET GUI 框架（如 WPF、UWP/WinUI 和 .NET MAUI）之间共享代码。请务必注意，虽然前面提到的 GUI 框架使用 XAML 作为 UI 技术，但 XAML 不会进入 Windows 窗体。

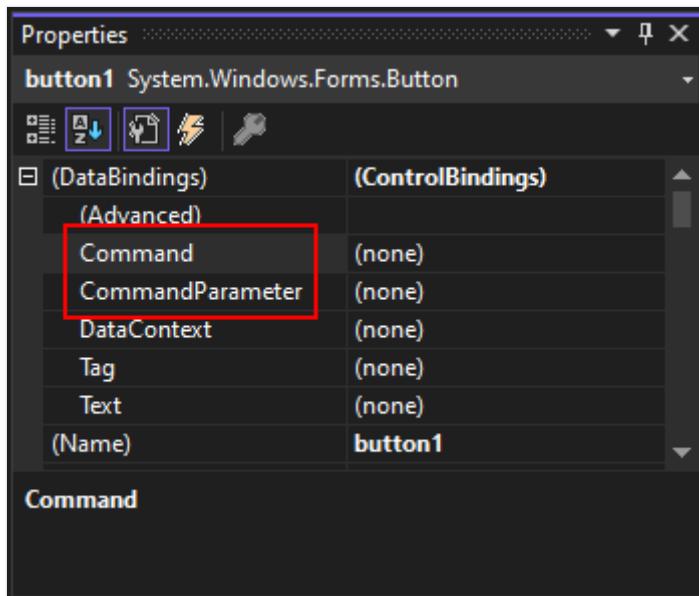
接口 [IBindableComponent](#) 和 [BindableComponent](#) 类驱动新的绑定系统。[Control](#) 实现接口并向 Windows 窗体提供新的数据绑定功能。

按钮命令

按钮命令在 .NET 7 中处于预览状态，现在已在 .NET 8 中完全启用。与 WPF 类似，可以将实现 [ICommand](#) 接口的对象实例分配给按钮的 [Command](#) 属性。单击按钮时，将调用该命令。

通过为按钮的 [CommandParameter](#) 属性指定值，可以在调用命令时提供可选参数。

[Command](#) 和 [CommandParameter](#) 属性可通过设计器中的“属性”窗口进行设置（“(DataBindings)”下），如下图所示。



按钮还侦听 `ICommand.CanExecuteChanged` 事件，这会导致控件查询 `ICommand.CanExecute` 方法。当该方法返回 `true` 时，将启用控件；返回 `false` 时则禁用该控件。

Visual Studio DPI 改进

Visual Studio 2022 17.8 推出了不感知 DPI 的设计器选项卡。以前，Visual Studio 中的“Windows 设计器”选项卡以 Visual Studio 的 DPI 运行。这会导致设计不感知 DPI 的 Windows 窗体应用时出现问题。现在，无论感知 DPI 与否，都可以确保设计器以与你希望应用运行的规模相同的规模运行。在引入此功能之前，你不得不在不感知 DPI 的模式下运行 Visual Studio，这使得在 Windows 中应用缩放时 Visual Studio 本身变得模糊。现在，你可以单独离开 Visual Studio，让设计器在不感知 DPI 的情况下运行。

可以通过向项目文件添加 `<ForceDesignerDPIUnaware>` 并将值设置为 `true`，来为 Windows 窗体项目启用不感知 DPI 的设计器。

```
XML
<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net8.0-windows</TargetFramework>
  <Nullable>enable</Nullable>
  <UseWindowsForms>true</UseWindowsForms>
  <ImplicitUsings>enable</ImplicitUsings>
  <ForceDesignerDPIUnaware>true</ForceDesignerDPIUnaware>
  <ApplicationHighDpiMode>DpiUnawareGdiScaled</ApplicationHighDpiMode>
</PropertyGroup>
```

① 重要

Visual Studio 在加载项目时读取此设置，而不是在项目更改时读取此设置。更改此设置后，请卸载并重新加载项目以使 Visual Studio 尊重它。

高 DPI 改进

改进了 [PerMonitorV2](#) 的高 DPI 呈现：

- 正确缩放嵌套控件。例如，一个在面板上的按钮，放置在一个选项卡页上。
- 请根据当前监视器 DPI 设置缩放 `Form.MaximumSize` 和 `Form.MinimumSize` 属性。

从 .NET 8 开始，此功能默认启用，你需要选择退出此功能，才能还原以前的行为。

若要禁用该功能，请将 `System.Windows.Forms.ScaleTopLevelFormMinMaxSizeForDpi` 添加到 `configProperties` 中的设置，并将值设置为 `false`：

JSON

```
{  
    "runtimeOptions": {  
        "tfm": "net8.0",  
        "frameworks": [  
            ...  
        ],  
        "configProperties": {  
            "System.Windows.Forms.ScaleTopLevelFormMinMaxSizeForDpi": false,  
        }  
    }  
}
```

其他改进

下面是一些其他值得注意的变更：

- 改进了处理 `FolderBrowserDialog` 的代码，修复了一些内存泄漏。
- Windows 窗体的代码库一直在缓慢地启用 C# 为 Null 性，从而消除了任何潜在的 null 引用错误。
- `System.Drawing` 源代码已迁移到 [Windows 窗体 GitHub 存储库](#)。
- 新式 Windows 图标可以通过新 API `System.Drawing.SystemIcons.GetStockIcon` 访问。`System.Drawing.StockIconId` 枚举列出了所有可用的系统图标。
- 现在，运行时可以使用更多设计器。有关详细信息，请参阅 [GitHub 问题 #4908](#)。

.NET 7 Windows 窗体中的新增功能

项目 • 2024/11/11

本文介绍 .NET 7 中一些新的 Windows 窗体功能和增强功能。

从 .NET Framework 迁移到 .NET 7 时，应注意一些中断性变更。有关详细信息，请参阅 [Windows 窗体中的重大更改](#)。

高 DPI 改进

改进了 [PerMonitorV2](#) 的高 DPI 呈现：

- 正确缩放嵌套控件。例如，一个在面板上的按钮，放置在一个选项卡页上。
- 根据运行 `ApplicationHighDpiMode` 设置为 `PerMonitorV2` 的应用程序的当前监视器 DPI 设置，缩放 `Form.MaximumSize` 和 `Form.MinimumSize` 属性。

在 .NET 7 中，此功能默认处于禁用状态，必须选择加入才能接收此更改。从 .NET 8 开始，此功能默认启用，你需要选择退出此功能，才能还原以前的行为。

若要启用功能，请在 `runtimeconfig.json` 中设置 `configProperties` 设置：

```
JSON

{
  "runtimeOptions": {
    "tfm": "net7.0",
    "frameworks": [
      ...
    ],
    "configProperties": {
      "System.Windows.Forms.ScaleTopLevelFormMinMaxSizeForDpi": true,
    }
  }
}
```

辅助功能改进和修复

此版本进一步改进了辅助功能，包括但不限于以下项：

- 屏幕阅读器中观察到的许多与公告相关的问题都已得到解决，可以确保有关控件的信息都是正确的。例如，[ListView](#) 现在可以正确地指示是展开还是折叠组。
- 现在，更多控件提供了 UI 自动化支持：

- [TreeView](#)
 - [DateTimePicker](#)
 - [ToolStripContainer](#)
 - [ToolStripPanel](#)
 - [FlowLayoutPanel](#)
 - [TableLayoutPanel](#)
 - [SplitContainer](#)
 - [PrintPreviewControl](#)
 - [WebBrowser](#)
- 修复了与在辅助工具（如讲述人）下运行 Windows 窗体应用程序相关的内存泄漏问题。
 - 辅助工具现在可以准确地绘制焦点指示器，并报告嵌套窗体和复合控件的某些元素（如 [DataGridView](#)、[ListView](#) 和 [TabControl](#)）的正确边框。
 - 自动化 UI [ExpandCollapse 控件模式](#)已在 [ListView](#)、[TreeView](#) 和 [PropertyGrid](#) 控件中正确实现，并且仅针对可展开项激活。
 - 更正了控件中的各种颜色对比度比率。
 - 改进了高对比度主题中的 [ToolStripTextBox](#) 和 [ToolStripButton](#) 的可见性。

数据绑定改进（预览版）

虽然 Windows 窗体已经有一个功能强大的绑定引擎，但正在引入一种更现代化的数据绑定形式，类似于 WPF 提供的数据绑定。

借助新的数据绑定功能，可以完全采用 MVVM 模式，并在 Windows 窗体中比以往更轻松地使用 ViewModels 中的对象关系映射器。这反过来又可以减少代码隐藏文件中的代码，开辟了新的测试可能性。更重要的是，它支持 Windows 窗体和其他 .NET GUI 框架（如 WPF、UWP/WinUI 和 .NET MAUI）之间的代码共享。澄清一个常见问题，目前还没有在 Windows 窗体中引入 XAML 的任何计划。

这些新的数据绑定功能在 .NET 7 中处于预览状态，而有关此功能的更多工作将在 .NET 8 中开展。

若要启用新绑定，请将 `EnablePreviewFeatures` 设置添加到项目文件。C# 和 Visual Basic 都支持此操作。

XML

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
<!-- other settings -->

<PropertyGroup>
  <EnablePreviewFeatures>true</EnablePreviewFeatures>
</PropertyGroup>

</Project>
```

以下代码片段演示了添加到 Windows 窗体中的各种类的新属性、事件和方法。即使以下代码示例采用 C#，它也适用于 Visual Basic。

C#

```
public class Control {
  [BindableAttribute(true)]
  public virtual object DataContext { get; set; }
  [BrowsableAttribute(true)]
  public event EventHandler DataContextChanged;
  protected virtual void OnDataContextChanged(EventArgs e);
  protected virtual void OnParentDataContextChanged(EventArgs e);
}

[RequiresPreviewFeaturesAttribute]
public abstract class BindableComponent : Component, IBindableComponent,
IComponent, IDisposable {
  protected BindableComponent();
  public BindingContext? BindingContext { get; set; }
  public ControlBindingsCollection DataBindings { get; }
  public event EventHandler BindingContextChanged;
  protected virtual void OnBindingContextChanged(EventArgs e);
}

public abstract class ButtonBase : Control {
  [BindableAttribute(true)]
  [RequiresPreviewFeaturesAttribute]
  public ICommand? Command { get; set; }
  [BindableAttribute(true)]
  public object? CommandParameter { [RequiresPreviewFeaturesAttribute]
get; [RequiresPreviewFeaturesAttribute] set; }
  [RequiresPreviewFeaturesAttribute]
  public event EventHandler? CommandCanExecuteChanged;
  [RequiresPreviewFeaturesAttribute]
  public event EventHandler? CommandChanged;
  [RequiresPreviewFeaturesAttribute]
  public event EventHandler? CommandParameterChanged;
  [RequiresPreviewFeaturesAttribute]
  protected virtual void OnCommandCanExecuteChanged(EventArgs e);
  [RequiresPreviewFeaturesAttribute]
  protected virtual void OnCommandChanged(EventArgs e);
  [RequiresPreviewFeaturesAttribute]
  protected virtual void OnCommandParameterChanged(EventArgs e);
  [RequiresPreviewFeaturesAttribute]
  protected virtual void OnRequestCommandExecute(EventArgs e);
```

```
}

public abstract class ToolStripItem : BindableComponent, IComponent,
IDisposable, IDropTarget {
    [BindableAttribute(true)]
    [RequiresPreviewFeaturesAttribute]
    public ICommand Command { get; set; }
    [BindableAttribute(true)]
    public object CommandParameter { [RequiresPreviewFeaturesAttribute] get;
[RequiresPreviewFeaturesAttribute] set; }
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler CommandCanExecuteChanged;
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler CommandChanged;
    [RequiresPreviewFeaturesAttribute]
    public event EventHandler CommandParameterChanged;
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandCanExecuteChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnCommandParameterChanged(EventArgs e);
    [RequiresPreviewFeaturesAttribute]
    protected virtual void OnRequestCommandExecute(EventArgs e);
}

}
```

其他改进

下面是一些其他值得注意的变更：

- 拖放处理与 Windows 拖放功能一致，具有更丰富的显示效果，如图标和文本标签。
- 文件夹和文件对话框允许使用更多选项：
 - 添加到最近
 - 检查写入访问权限
 - 展开模式
 - “确定”需要交互
 - 选择“只读”
 - 显示隐藏文件
 - 显示固定位置
 - 显示预览
- **ErrorProvider** 现在有一个 **HasErrors** 属性。
- 窗体的贴靠布局已针对 Windows 11 进行了修复。

另请参阅

- [.NET 博客 - .NET 7 中 Windows 窗体的新增功能 ↴](#)

- Windows 窗体中的中断性变更
- 教程：创建新的 WinForms 应用
- 如何将 Windows 窗体桌面应用迁移到 .NET 5

.NET 6 Windows 窗体中的新增功能

项目 · 2024/11/11

本文介绍 .NET 6 中一些新的 Windows 窗体功能和增强功能。

从 .NET Framework 迁移到 .NET 6 时，应注意一些重大更改。有关详细信息，请参阅 [Windows 窗体中的重大更改](#)。

更新了用于 C# 的模板

.NET 6 引入了许多[针对标准控制台应用程序模板的更改](#)。根据这些更改，用于 C# 的 Windows 窗体模板已更新为默认支持 [global using 指令](#)、[文件范围的命名空间](#)和[可以为 null 的引用类型](#)。

Windows 窗体未沿用的新 C# 模板的一项功能是[顶级语句](#)。典型的 Windows 窗体应用程序需要 `[STAThread]` 属性，并且由拆分成多个文件（例如设计器代码文件）的多种类型组成，因此，使用顶级语句没有意义。

新应用程序启动

生成新 Windows 窗体应用程序的模板会创建一个 `Main` 方法，该方法在你的应用程序运行时用作应用程序的入口点。此方法包含配置 Windows 窗体并显示第一个窗体的代码，称为启动代码：

```
C#  
  
class Program  
{  
    [STAThread]  
    static void Main()  
    {  
        Application.EnableVisualStyles();  
        Application.SetCompatibleTextRenderingDefault(false);  
        Application.Run(new Form1());  
    }  
}
```

在 .NET 6 中，这些模板已修改为使用 `ApplicationConfiguration.Initialize` 方法调用的新启动代码。

```
C#
```

```

class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.Run(new Form1());
    }
}

```

此方法在编译时自动生成，并包含用于配置 Windows 窗体的代码。项目文件现在也可以控制这些设置，使你可以避免在代码中对它进行配置。例如，生成的方法类似于以下代码：

C#

```

public static void Initialize()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.SetHighDpiMode(HighDpiMode.SystemAware);
}

```

Visual Studio 使用新的启动代码来配置 Windows 窗体可视化设计器。如果通过恢复旧代码以及通过绕过 `ApplicationConfiguration.Initialize` 方法来选择不使用新的启动代码，Windows 窗体可视化设计器将不会遵循你所设置的启动设置。

`Initialize` 方法中生成的设置由项目文件控制。

项目级应用程序设置

为补充 Windows 窗体的[新应用程序启动](#)功能，应在项目文件中设置以前在应用程序的启动代码中设置的一些 `Application` 设置。项目文件可以配置以下应用程序设置：

[+] 展开表

项目设置	默认值	相应的 API
<code>ApplicationVisualStyles</code>	<code>true</code>	<code>Application.EnableVisualStyles</code>
<code>ApplicationUseCompatibleTextRendering</code>	<code>false</code>	<code>Application.SetCompatibleTextRenderingDefault</code>
<code>ApplicationHighDpiMode</code>	<code>SystemAware</code>	<code>Application.SetHighDpiMode</code>
<code>ApplicationDefaultFont</code>	<code>Segoe UI, 9pt</code>	<code>Application.SetDefaultFont</code>

以下示例展示了设置这些应用程序相关属性的项目文件：

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net6.0-windows</TargetFramework>
  <Nullable>enable</Nullable>
  <UseWindowsForms>true</UseWindowsForms>
  <ImplicitUsings>enable</ImplicitUsings>

  <ApplicationVisualStyles>true</ApplicationVisualStyles>

  <ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRen
dering>
    <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>
    <ApplicationDefaultFont>Microsoft Sans Serif,
  8.25pt</ApplicationDefaultFont>

</PropertyGroup>

</Project>
```

Windows 窗体可视化设计器会使用这些设置。有关详细信息，请参阅 [Visual Studio 设计器改进部分](#)。

更改默认字体

.NET Core 3.0 上的 Windows 窗体引入了新的 Windows 窗体默认字体：“Segoe UI, 9pt”。此字体更符合 [Windows 用户体验 \(UX\) 指南](#)。但是，.NET Framework 使用“Microsoft Sans Serif, 8.25pt”作为默认字体。这一变化使一些客户更难以将他们采用像素完美的布局的大型应用程序从 .NET Framework 迁移到 .NET。之前更改整个应用程序的字体的唯一方法是编辑项目中的每个窗体，通过将 [Font](#) 属性设置为替代字体来实现。

现在，可以通过两种方式来设置默认字体：

- 设置[应用程序启动](#)代码要使用的项目文件中的默认字体：

① 重要

这是首选方法。通过使用项目来配置新应用程序启动系统，使 Visual Studio 可以在设计器中使用这些设置。

在以下示例中，项目文件将 Windows 窗体配置为使用 .NET Framework 所用的相同字体。

XML

```
<Project Sdk="Microsoft.NET.Sdk">

    <!-- other settings -->

    <PropertyGroup>
        <ApplicationDefaultFont>Microsoft Sans Serif,
        8.25pt</ApplicationDefaultFont>
    </PropertyGroup>

</Project>
```

- 或 -

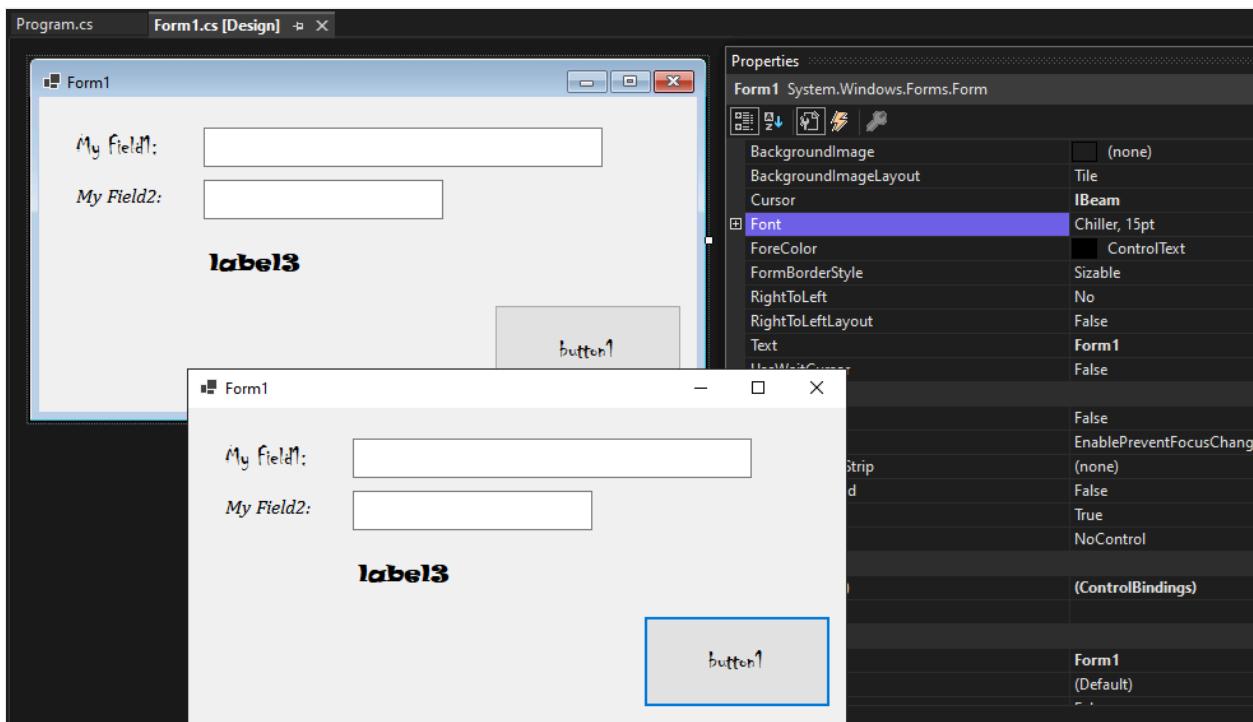
- 以旧方式调用 `Application.SetDefaultFont` API (但没有设计器支持)：

C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.SetHighDpiMode(HighDpiMode.SystemAware);
        Application.SetDefaultFont(new Font(new FontFamily("Microsoft
Sans Serif"), 8.25f));
        Application.Run(new Form1());
    }
}
```

Visual Studio 设计器改进

Windows 窗体可视化设计器现在可以准确地反映默认字体。适用于 .NET 的 Windows 窗体的早期版本没有在可视化设计器中正确显示 Segoe UI 字体，实际上使用的是 .NET Framework 的默认字体来设计窗体。由于新的[新应用程序启动](#)功能，可视化设计器可准确地反映默认字体。此外，可视化设计器还遵循在项目文件中设置的默认字体。



更多的运行时设计器

存在于 .NET Framework 中并且能够生成通用设计器（例如生成报表设计器）的设计器已添加到 .NET 6：

- [System.ComponentModel.Design.ComponentDesigner](#)
- [System.Windows.Forms.Design.ButtonBaseDesigner](#)
- [System.Windows.Forms.Design.ComboBoxDesigner](#)
- [System.Windows.Forms.Design.ControlDesigner](#)
- [System.Windows.Forms.Design.DocumentDesigner](#)
- [System.Windows.Forms.Design.DocumentDesigner](#)
- [System.Windows.Forms.Design.FormDocumentDesigner](#)
- [System.Windows.Forms.Design.GroupBoxDesigner](#)
- [System.Windows.Forms.Design.LabelDesigner](#)
- [System.Windows.Forms.Design.ListBoxDesigner](#)
- [System.Windows.Forms.Design.ListViewDesigner](#)
- [System.Windows.Forms.Design.MaskedTextBoxDesigner](#)
- [System.Windows.Forms.Design.PanelDesigner](#)
- [System.Windows.Forms.Design.ParentControlDesigner](#)
- [System.Windows.Forms.Design.ParentControlDesigner](#)
- [System.Windows.Forms.Design.PictureBoxDesigner](#)
- [System.Windows.Forms.Design.RadioButtonDesigner](#)
- [System.Windows.Forms.Design.RichTextBoxDesigner](#)
- [System.Windows.Forms.Design.ScrollableControlDesigner](#)
- [System.Windows.Forms.Design.ScrollableControlDesigner](#)
- [System.Windows.Forms.Design.TextBoxBaseDesigner](#)

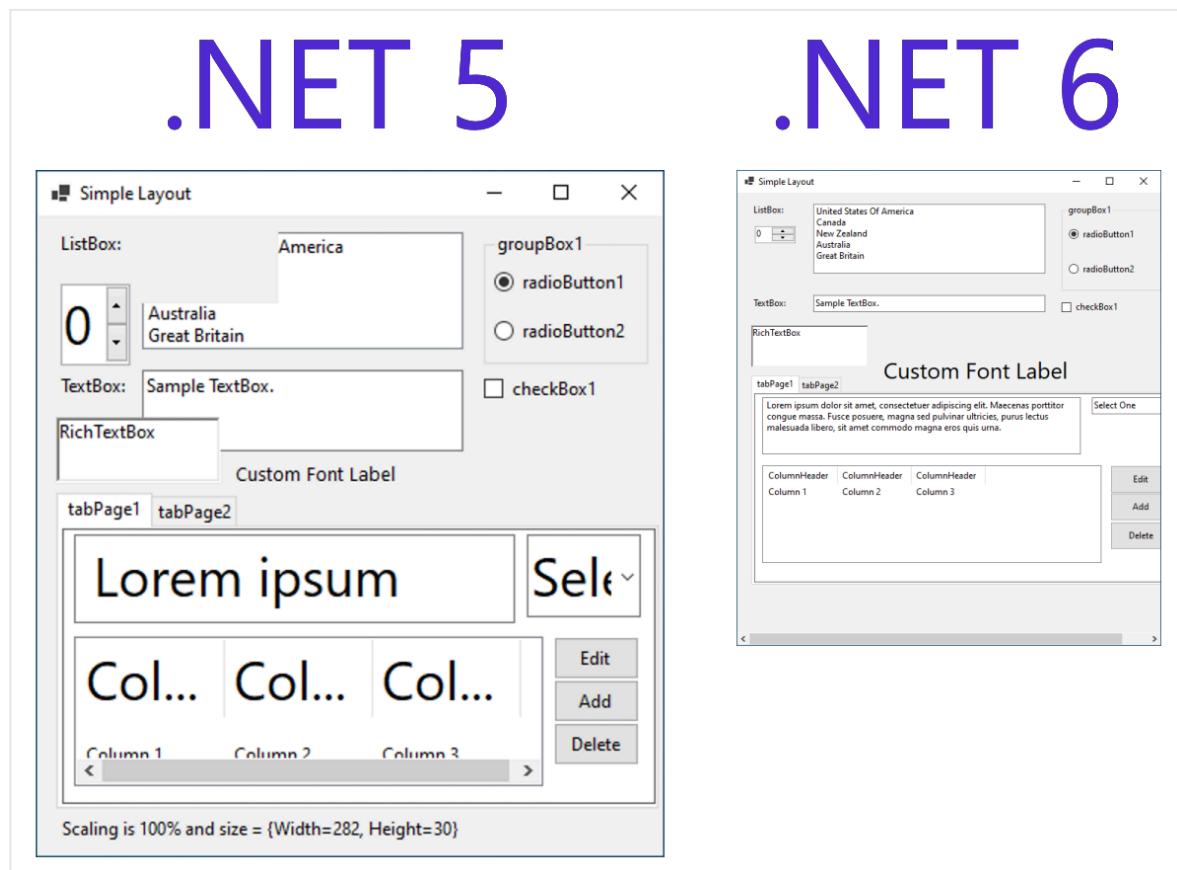
- System.Windows.Forms.Design.TextBoxDesigner
- System.Windows.Forms.Design.ToolStripDesigner
- System.Windows.Forms.Design.ToolStripDropDownDesigner
- System.Windows.Forms.Design.ToolStripItemDesigner
- System.Windows.Forms.Design.ToolStripItemDesigner
- System.Windows.Forms.Design.TreeViewDesigner
- System.Windows.Forms.Design.UpDownBaseDesigner
- System.Windows.Forms.Design.UserControlDocumentDesigner

针对 PerMonitorV2 的高 DPI 改进

改进了 PerMonitorV2 的高 DPI 呈现：

- 控件是使用与应用程序相同的 DPI 感知创建的。
- 容器控件和 MDI 子窗口改进了缩放行为。

例如，在 .NET 5 中，将 Windows 窗体应用从缩放比例为 200% 的监视器移动到缩放比例为 100% 的监视器会导致控件错位。.NET 6 中已大幅改进了这一点：



新 API

- System.Windows.Forms.Application.SetDefaultFont

- System.Windows.Forms.Control.IsAncestorSiteInDesignMode
- System.Windows.Forms.ProfessionalColors.StatusStripBorder
- System.Windows.Forms.ProfessionalColorTable.StatusStripBorder

新 Visual Basic API

- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventHandler
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.MinimumSplashScreenDisplayTime
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.MinimumSplashScreenDisplayTime
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.Font
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.Font
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.HighDpiMode
- Microsoft.VisualBasic.ApplicationServices.ApplyApplicationDefaultsEventArgs.HighDpiMode
- Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase.ApplyApplicationDefaults
- Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase.HighDpiMode
- Microsoft.VisualBasic.ApplicationServices.WindowsFormsApplicationBase.HighDpiMode

已更新的 API

- System.Windows.Forms.Control.Invoke 现在接受 System.Action 和 System.Func<TResult> 作为输入参数。
- System.Windows.Forms.Control.BeginInvoke 现在接受 System.Action 作为输入参数。
- 使用以下成员扩展了 System.Windows.Forms.DialogResult：
 - TryAgain
 - Continue
- System.Windows.Forms.Form 新增了一个属性：MdiChildrenMinimizedAnchorBottom
- 使用以下成员扩展了 System.Windows.Forms.MessageBoxButtons：
 - CancelTryContinue

- 使用以下成员扩展了 [System.Windows.Forms.MessageBoxDefaultButton](#)：
 - [Button4](#)
- [System.Windows.Forms.LinkClickedEventArgs](#) 现在新增了一个构造函数，并使用以下属性进行了扩展：
 - [System.Windows.Forms.LinkClickedEventArgs.LinkLength](#)
 - [System.Windows.Forms.LinkClickedEventArgs.LinkStart](#)
- [System.Windows.Forms.NotifyIcon.Text](#) 现在限制为 127 个字符（原来为 63 个字符）。

改进了 辅助功能

Microsoft UI 自动化模式可更好地与讲述人和 Jaws 等辅助工具协同工作。

另请参阅

- Windows 窗体中的中断性变更
- 教程：创建新的 WinForms 应用
- 如何将Windows 窗体桌面应用升级到 .NET

.NET 5 Windows 窗体中的新增功能

项目 • 2024/11/11

适用于 .NET 5 的 Windows 窗体增添了以下优于 .NET Framework 的功能和增强功能。

从 .NET Framework 迁移到 .NET 5 时，应注意一些重大更改。有关详细信息，请参阅 [Windows 窗体中的重大更改](#)。

增强功能

- Microsoft UI 自动化模式可更好地与讲述人和 Jaws 等辅助工具协同工作。
- 性能提升。
- 对于高 DPI 分辨率（如 4k 监视器），VB.NET 项目模板默认使用 DPI SystemAware 设置。
- 默认字体与当前 Windows 设计建议相符。

⊗ 注意

这可能会影响从 .NET Framework 迁移的应用的布局。

新控件

将 Windows 窗体移植到 .NET Framework 后，添加了以下控件：

- [System.Windows.Forms.TaskDialog](#)

任务对话框是可用于显示信息并接收用户的简单输入的文本框。与消息框类似，其格式由操作系统根据你设置的参数进行设置。任务对话框具有比消息框更多的功能。有关详细信息，请参阅[任务对话框示例](#)。

- [Microsoft.Web.WebView2.WinForms.WebView2](#)

具有新式 Web 支持的新 Web 浏览器控件。基于 Edge (Chromium)。有关详细信息，请参阅 [Windows 窗体中的 WebView2 入门](#)。

增强的控件

- [System.Windows.Forms.ListView](#)

- 支持可折叠的组
- 页脚
- 组副标题、任务和标题图像
- [System.Windows.Forms.FolderBrowserDialog](#)

此对话框已升级为使用新式 Windows 体验，而不是旧版 Windows 7 体验。

- [System.Windows.Forms.FileDialog](#)

- 增加了对 [ClientGuid](#) 的支持。

`ClientGuid` 支持调用应用程序，以将 GUID 与对话框的持久状态关联。对话框的状态可能包括最后访问的文件夹以及对话框的位置和大小等因素。通常，此状态根据可执行文件的名称持久保留。凭借 `ClientGuid`，应用程序可以在同一应用程序中保持对话框的不同状态。

- [System.Windows.Forms.TextRenderer](#)

添加了对 [ReadOnlySpan<T>](#) 的支持，以增强呈现文本的性能。

另请参阅

- [Windows 窗体中的中断性变更](#)
- [教程：创建新的 WinForms 应用 \(Windows 窗体 .NET\)](#)
- [如何将 Windows 窗体桌面应用迁移到 .NET 5](#)

桌面指南 (Windows 窗体 .NET)

项目 • 2024/11/14

欢迎使用 Windows 窗体的桌面指南，Windows 窗体是一个可创建适用于 Windows 的丰富桌面客户端应用的 UI 框架。Windows 窗体开发平台支持广泛的应用开发功能，包括控件、图形、数据绑定和用户输入。Windows 窗体采用 Visual Studio 中的拖放式可视化设计器，可轻松创建 Windows 窗体应用。

Windows 窗体有两种实现：

1. 托管于 [GitHub](#) 上的开放源代码实现。

此版本在 .NET 6 及更高版本上运行。

使用 Visual Studio 2022 版本 17.12 [为 .NET 9](#) Windows 窗体最新版本。

2. 受 Visual Studio 2022、Visual Studio 2019 和 Visual Studio 2017 支持的 .NET Framework 4 实现。

.NET Framework 4 是仅限 Windows 的 .NET 版本，被视为一个 Windows 操作系统组件。此版本的 Windows 窗体随 .NET Framework 一起分发。

有关 Windows 窗体的 .NET Framework 版本的详细信息，请参阅 [.NET Framework 的 Windows 窗体](#)。

介绍

Windows 窗体是用于生成 Windows 桌面应用的 UI 框架。它提供了一种基于 Visual Studio 中提供的可视化设计器创建桌面应用的高效方法。利用视觉对象控件的拖放放置等功能，可以轻松生成桌面应用。

使用 Windows 窗体，可以开发包含丰富图形的应用，这些应用易于部署和更新，并且在脱机状态下或连接到 Internet 时都可正常工作。Windows 窗体应用可以访问运行应用的计算机的本地硬件和文件系统。

要了解如何创建 Windows 窗体应用，请参阅[教程：创建新的 WinForms 应用](#)。

为什么要从 .NET Framework 迁移

.NET 的 Windows 窗体提供优于 .NET Framework 的新功能和增强功能。有关详细信息，请参阅 [.NET 9 Windows 窗体中的新增功能](#)。若要了解如何升级应用，请参阅[如何将 Windows 窗体桌面应用升级到 .NET](#)

生成丰富的交互式用户界面

Windows 窗体是用于 .NET 的 UI 技术，是一组简化读取和写入文件系统等常见应用任务的托管库。 使用类似于 Visual Studio 的开发环境时，可以创建 Windows 窗体智能客户端应用，该应用可显示信息、请求来自用户的输入以及通过网络与远程计算机通信。

在 Windows 窗体中，窗体是一种可视图面，可在其上对用户显示信息。 通常情况下，通过向窗体添加控件和开发对用户操作（如点击鼠标或按键）的响应来生成 Windows 窗体应用。 控件是离散的 UI 元素，用于显示数据或接受数据输入。

当用户对你的窗体或一个窗体控件执行了某个操作，该操作将生成一个事件。 应用通过代码对这些事件做出反应，并在事件发生时对其进行处理。

Windows 窗体包含各种可以向窗体添加的控件：显示文本框、按钮、下拉框、单选按钮甚至网页的控件。 如果某个现有控件不满足你的需要，Windows 窗体还支持使用 [UserControl](#) 类创建自己的自定义控件。

Windows 窗体具有丰富的 UI 控件，这些控件可模拟 Microsoft Office 等高端应用中的功能。 使用 [ToolStrip](#) 和 [MenuStrip](#) 控件时，可以创建包含文本和图像的工具栏和菜单、显示子菜单和托管其他控件（如文本框和组合框）。

借助 Visual Studio 中的拖放式 Windows 窗体设计器，可以轻松创建 Windows 窗体应用。 只需用光标选中控件，然后将它们放置到窗体中所需的位置即可。 设计器提供诸如网格线和对齐线的工具，以便简化对齐控件的操作。 可以使用 [FlowLayoutPanel](#)、[TableLayoutPanel](#) 和 [SplitContainer](#) 控件在更短的时间内创建高级窗体布局。

最后，如果必须创建自己的自定义用户界面元素，[System.Drawing](#) 命名空间包含各种类，可用以直接在窗体上呈现线条、圆形和其他形状。

创建窗体和控件

如需了解如何使用这些功能的步骤信息，请参阅以下“帮助”主题。

- [如何向项目添加窗体](#)
- [如何向窗体添加控件](#)

显示和操纵数据

许多应用必须显示数据库、XML 文件、JSON 文件、Web 服务或其他数据源中的数据。 Windows 窗体提供了一个灵活的控件（名为 [DataGridView](#) 控件），用于以传统的行和列格式显示此类表格数据，以便使每段数据块均占据其自己的单元格。 使用 [DataGridView](#) 时，可以自定义各个单元格的外观，将任意行和列锁定在所需位置，在单元格内部显示复杂控件，此外还具有其他功能。

通过网络连接到数据源对于 Windows 窗体而言是一个简单的任务。 [BindingSource](#) 组件表示与数据源的连接，并公开完成以下操作的方法：将数据绑定到控件、导航到上一个和下一个记录、编辑记录，以及将更改保存回原始源。[BindingNavigator](#) 控件通过 [BindingSource](#) 组件提供一个简单界面，可供用户在记录间导航。

可以使用 Visual Studio 中的“数据源”窗口轻松创建数据绑定控件。 窗口显示数据源，如数据库、Web 服务和项目中的对象。 可以通过将此窗口中的项拖动到项目中的窗体上来创建数据绑定控件。 还可以通过将对象从“数据源”窗口拖动到现有控件上来将现有控件与数据进行数据绑定。

可在 Windows 窗体中管理的另一类数据绑定是“设置”。 大多数应用均必须保留有关其运行时状态的某些信息（例如窗体的最后已知大小）以及用户首选项数据（例如保存的文件的默认位置）。“应用程序设置”功能通过提供一种在客户端计算机上存储两种类型的设置的简单方法来满足这些要求。 通过使用 Visual Studio 或代码编辑器定义这些设置后，这些设置便作为 XML 保留并自动在运行时读取回内存中。

将应用部署到客户端计算机

编写了应用后，必须将应用发送给用户，以便他们可以在自己的客户端计算机上安装和运行此应用。 使用 ClickOnce 技术时，只需进行几次点击便可以从 Visual Studio 内部部署应用，然后向用户提供指向 Web 上的应用的 URL。 ClickOnce 可管理应用中的所有元素和依赖项，并确保应用正确安装到客户端计算机上。

ClickOnce 应用可以配置为仅在用户连接到网络时运行，或配置为联机和脱机时均可运行。 如果指定应用支持脱机操作，则 ClickOnce 会在用户的“开始”菜单中添加一个指向应用的链接。 然后用户便可以打开应用，而无需使用此 URL。

如果你更新应用，则会同时向你的 Web 服务器发布一个新的部署清单和应用的一个新副本。 ClickOnce 将检测到有可用更新并将升级用户的安装。 更新旧应用无需进行自定义编程。

另请参阅

- [教程：创建新的 WinForms 应用](#)
- [如何向项目添加窗体](#)
- [添加控件](#)

教程：使用 .NET 创建 Windows 窗体应用

项目 • 2024/11/16

本教程介绍如何使用 Visual Studio 创建新的Windows 窗体应用。 使用 Visual Studio，可以向窗体添加控件并处理事件。 在本教程结束时，你有一个简单的应用，用于向列表框添加名称。

在本教程中，你将了解：

- ✓ 创建新的 Windows 窗体应用
- ✓ 将控件添加到窗体
- ✓ 处理控制事件以提供应用功能
- ✓ 运行应用

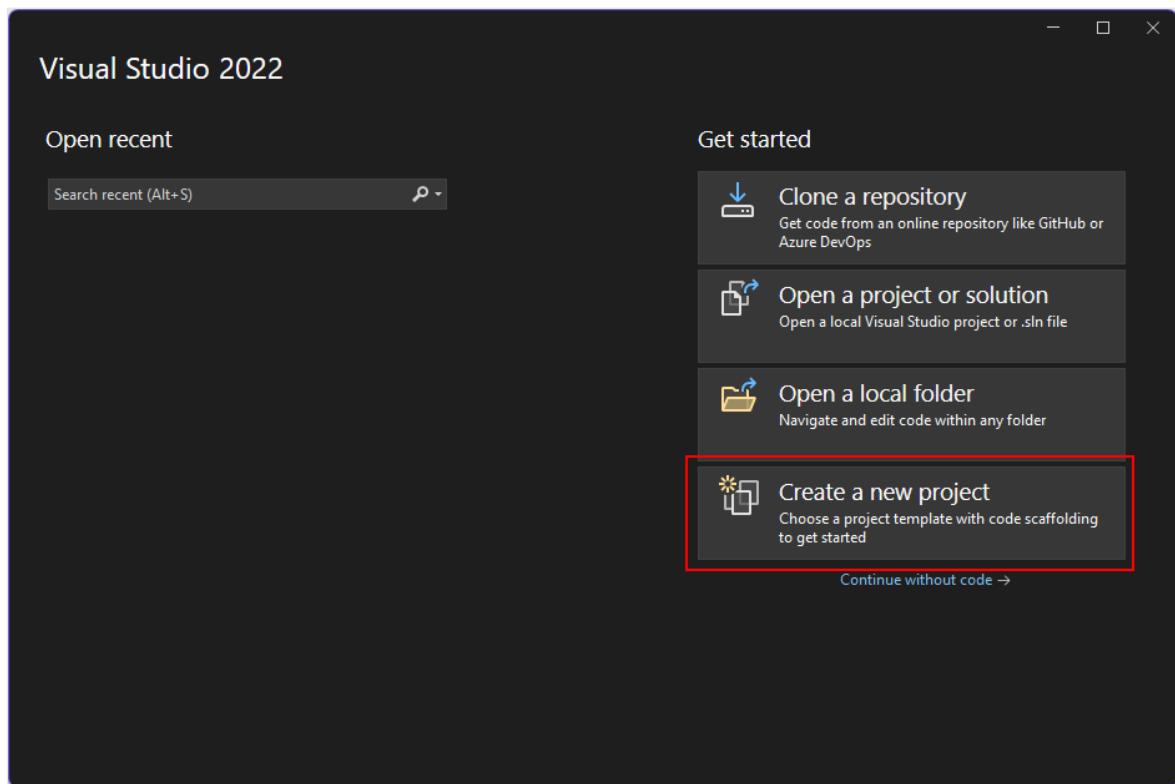
先决条件

- [Visual Studio 2022 版本 17.12 或更高版本](#)
 - 选择 [.NET 桌面开发工作负载](#)
 - 选择 [.NET 9 单个组件](#)

创建 Windows 窗体应用程序

创建新应用的第一步是打开 Visual Studio 并通过模板生成应用。

1. 打开 Visual Studio。
2. 选择“创建新项目”。

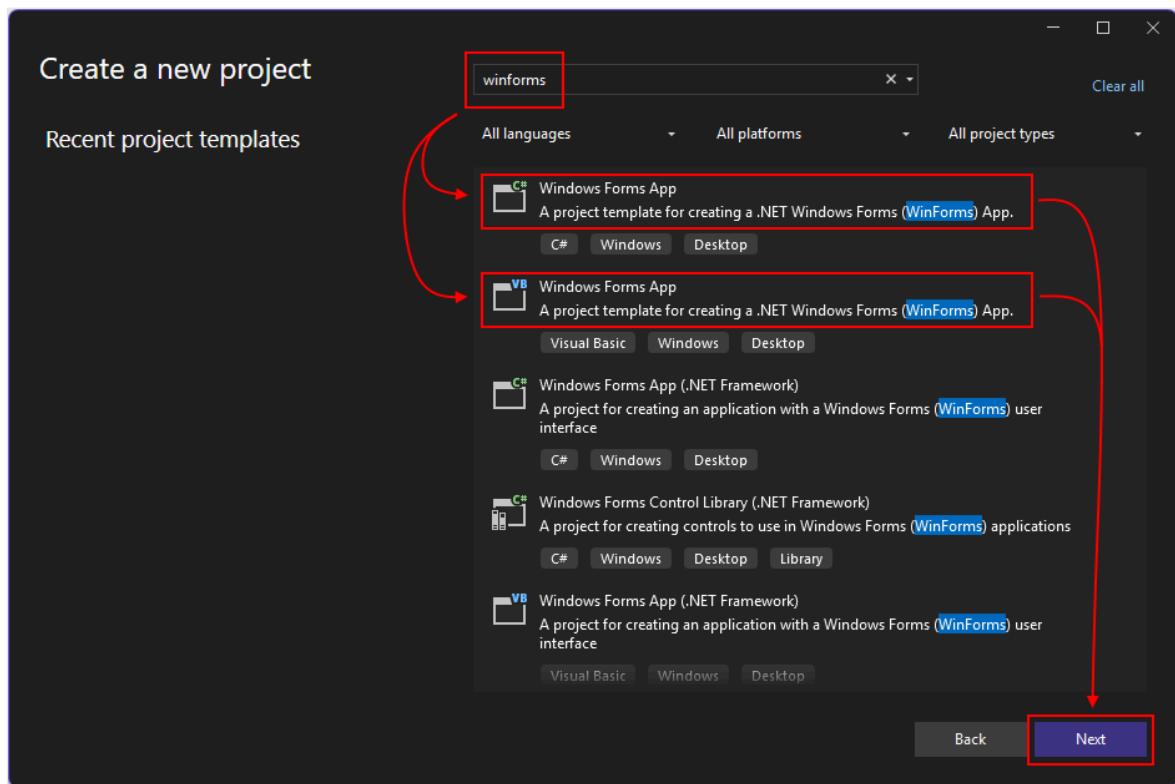


3. 在“搜索模板”框中，键入“winforms”，然后等待搜索结果显示。
4. 在“代码语言”下拉列表中，选择“C#”或“Visual Basic”。
5. 在模板列表中，选择**Windows 窗体应用**，然后选择“**下一步**”。

① 重要

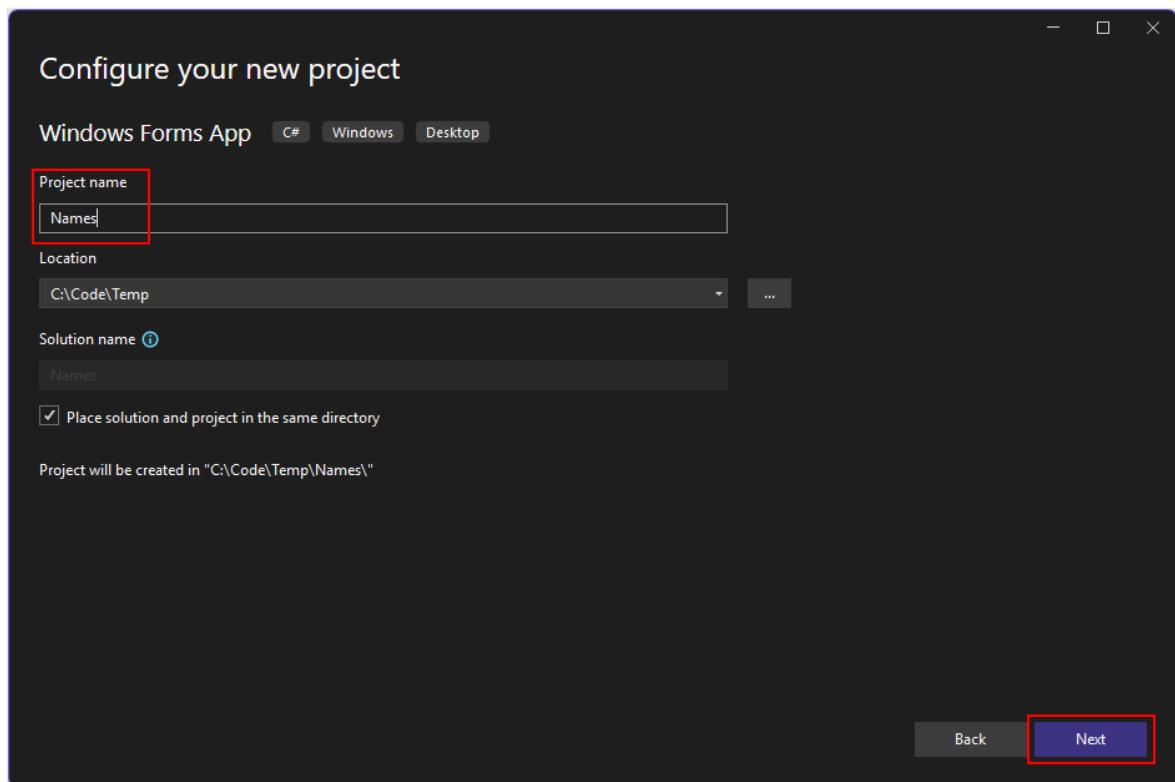
请勿选择“Windows 窗体应用 (.NET Framework)”模板。

下图显示了 C# 和 Visual Basic .NET 项目模板。如果应用 **了代码语言筛选器**，则会列出相应的模板。

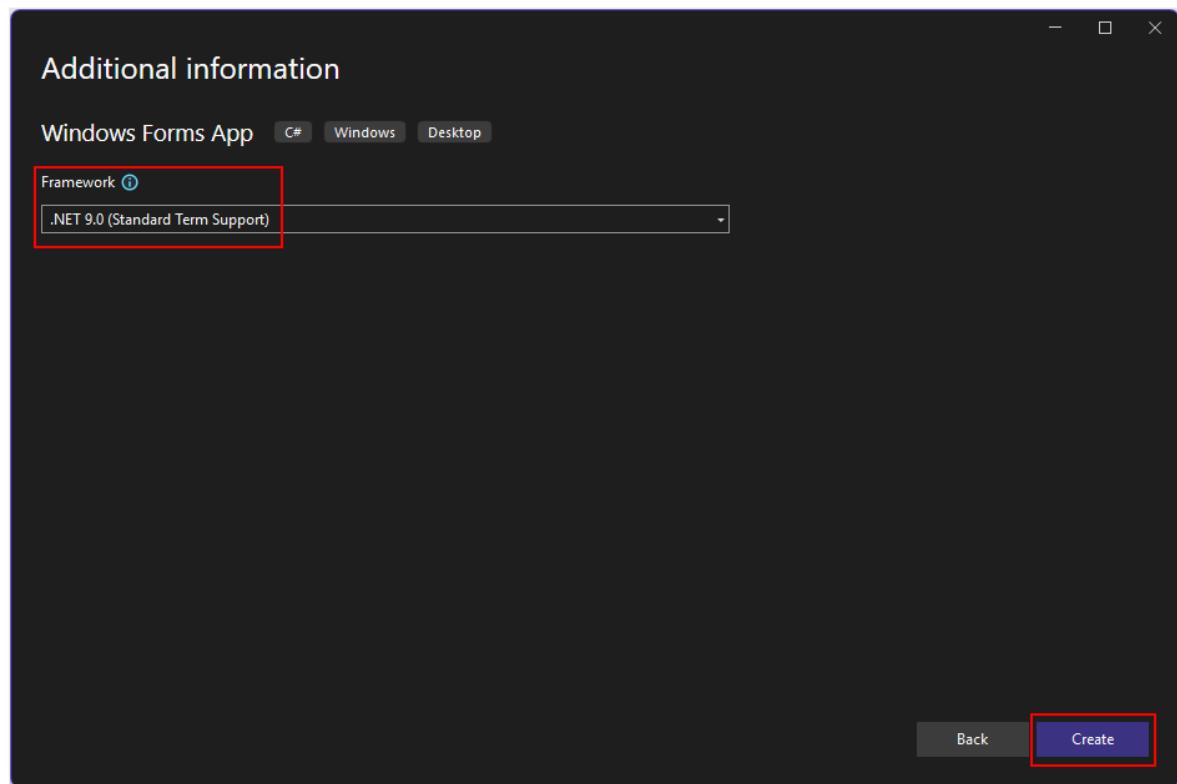


6. 在“配置新项目”窗口中，将项目名称设置为“名称”，然后选择“下一步”。

还可以通过调整“位置”路径将项目保存到其他文件夹。



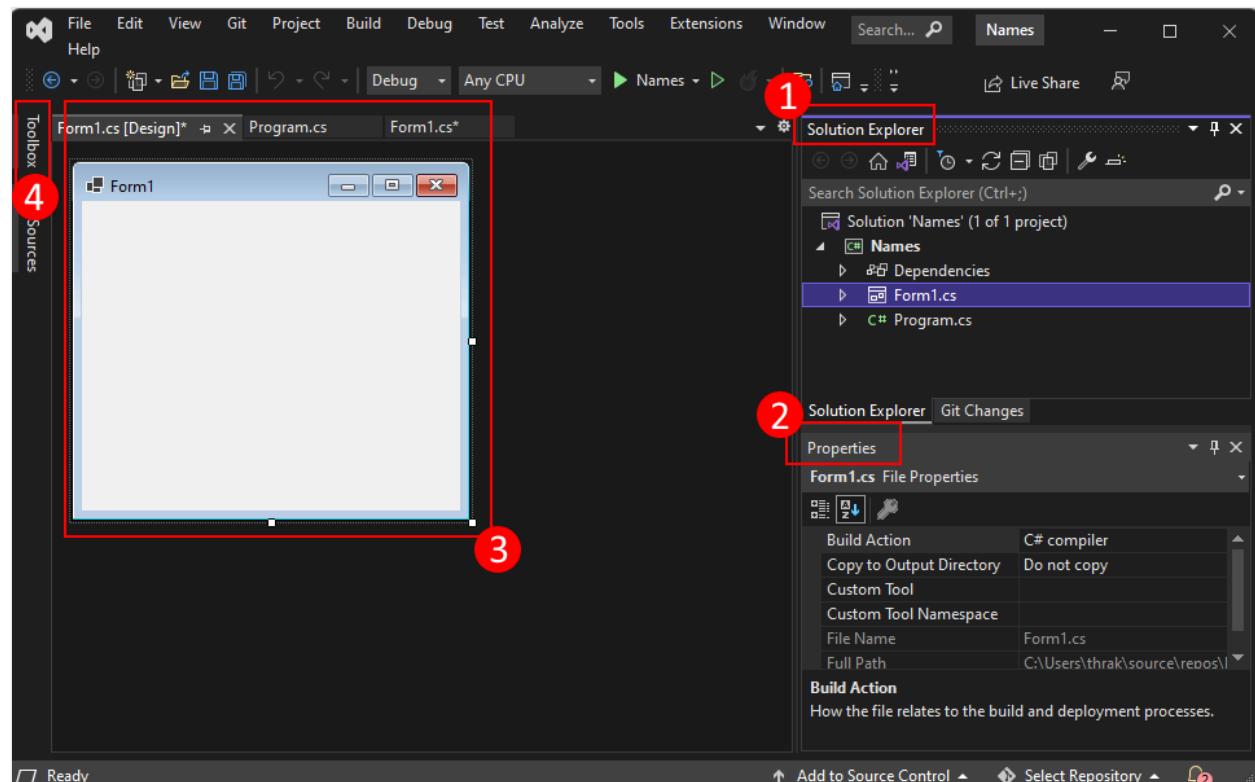
7. 最后，在“其他信息”窗口中，选择框架设置的 .NET 9.0（标准术语支持），然后选择“创建”。



生成应用后，Visual Studio 应打开默认窗体 *Form1* 的设计器窗口。如果窗体设计器不可见，请双击解决方案资源管理器窗口中的窗体以打开设计器窗口。

Visual Studio 的重要部分

在 Visual Studio 中对 Windows 窗体的支持在创建应用时具有四个重要组件：



1. “解决方案资源管理器”

所有项目文件、代码、窗体、资源都在此窗口中显示。

2. 属性

此窗口显示可以根据所选项的上下文配置的属性设置。例如，如果从**解决方案资源管理器**中选择项目，将显示与文件相关的设置。如果选择设计器中的对象，将显示控件或窗体的属性。

3. 窗体设计器

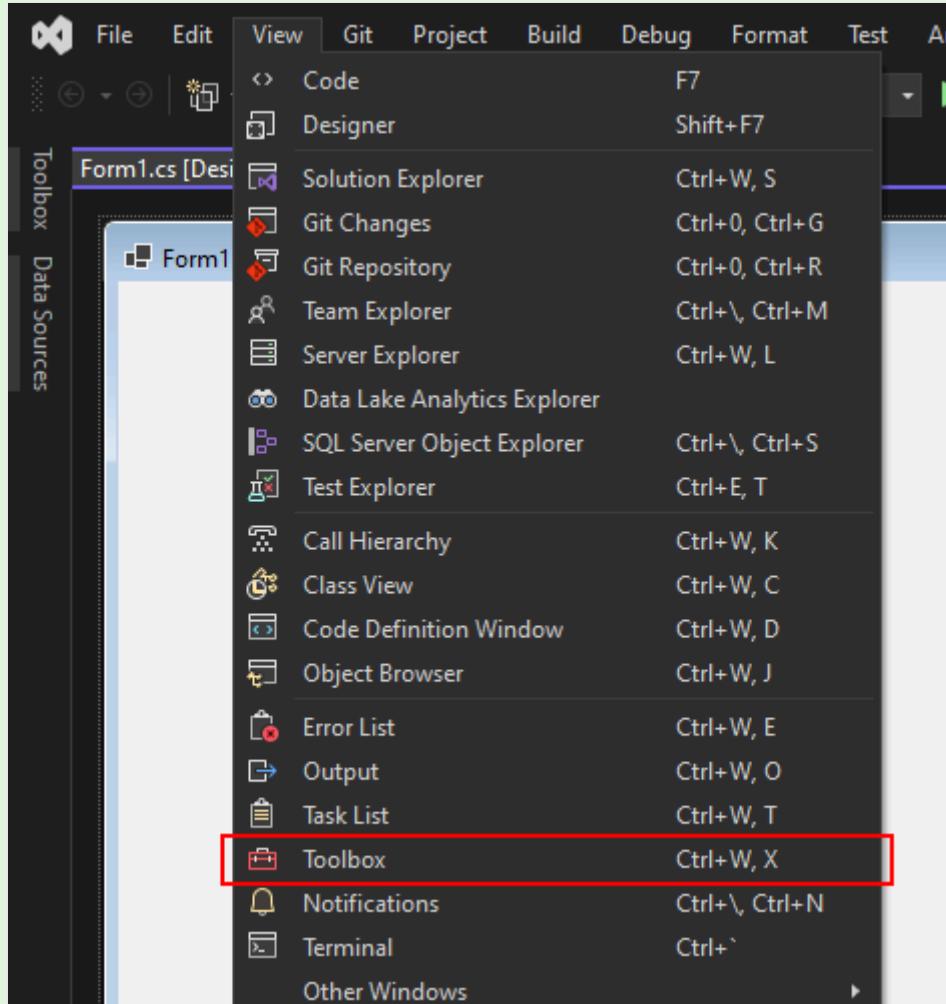
这是窗体的设计器。它是交互式的，可以从“工具箱”拖放对象。通过在设计器中选择和移动项，可以直观地为应用构建用户界面 (UI)。

4. 工具箱

工具箱包含可添加到窗体的所有控件。若要将控件添加到当前窗体，请双击控件或拖放控件。

💡 提示

如果工具箱不可见，则可以通过“查看”>“工具箱”菜单项来显示它。

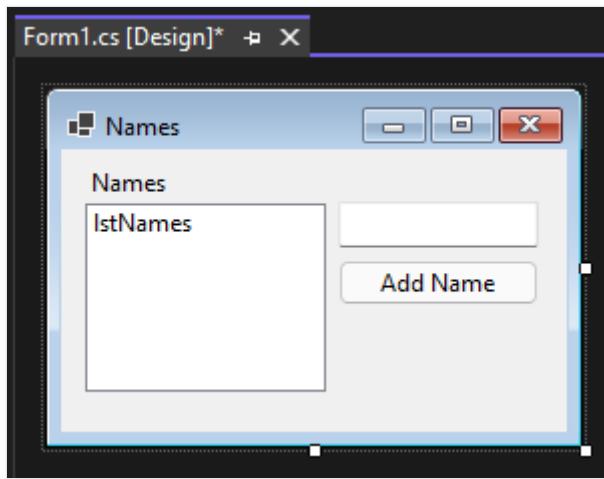


向窗体添加控件

打开 Form1 窗体设计器后，使用“**工具箱**”窗口将以下控件从工具箱中拖动并将它们拖放到窗体中，从而将控件添加到窗体中：

- 按钮
- 标签
- Listbox
- 文本框

根据下图定位和调整控件的大小：



可以使用鼠标移动控件并调整控件大小以匹配上一个图像，也可以使用下表配置每个控件。 若要配置控件，请在设计器中选择它，然后在“属性”窗口中设置相应的设置。 配置窗体时，请选择窗体的标题栏。

展开表

Object	设置	“值”
标签	位置	12, 9
	文本	Names
Listbox	名称	lstNames
	位置	12, 27
文本框	大小	120, 94
	名称	txtName
	位置	138, 26
	大小	100, 23

Object	设置	“值”
Button	名称	btnAdd
	位置	138, 55
	大小	100, 23
	文本	Add Name
窗体	文本	Names
	大小	268, 180

处理事件

现在，窗体已设置其所有控件，下一步是添加事件处理程序以响应用户输入。转到表单设计器并执行以下步骤：

1. 选择 窗体上的“添加名称”按钮控件。
2. 在“属性”窗口中，选择事件图标  以列出按钮的事件。
3. 找到“单击”事件，然后双击它以生成事件处理程序。

此操作将以下代码添加到窗体中：

```
C#
private void btnAdd_Click(object sender, EventArgs e)
{}
```

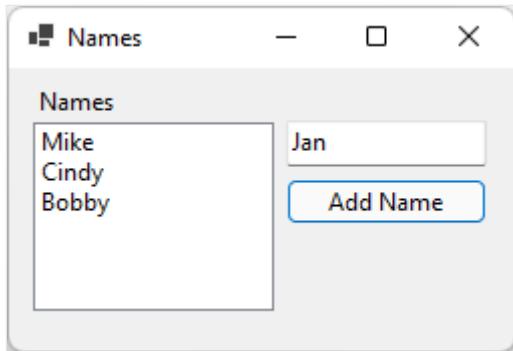
此处理程序的代码将向列表框 `lstNames` 添加文本框指定 `txtName` 的名称。但是，若要添加名称，需要满足两个条件：提供的名称不能为空，且该名称不能已存在。

4. 以下代码演示如何将名称添加到 `lstNames` 控件：

```
C#
private void btnAdd_Click(object sender, EventArgs e)
{
    if (!string.IsNullOrWhiteSpace(txtName.Text) &&
        !lstNames.Items.Contains(txtName.Text))
        lstNames.Items.Add(txtName.Text);
}
```

运行应用

处理事件后，按 **F5** 键或 **从菜单中选择“调试>开始调试”** 来运行应用。 应用启动时，将显示窗体，你可以在文本框中输入名称并选择按钮。



相关内容

- [详细了解 Windows 窗体](#)
- [使用控件概述](#)
- [事件概述](#)

从 Windows 窗体 .NET Framework 迁移到 .NET

项目 · 2024/12/03

本文介绍如何使用 .NET 升级助手将 Windows 窗体桌面应用升级到 .NET。Windows 窗体仍然是仅限 Windows 的框架，尽管 .NET 是一种跨平台技术。

先决条件

- Windows 操作系统。
- [下载并提取本文中使用的演示应用。](#)
- Visual Studio 2022 版本 17.12 或更高版本面向 .NET 9。
- [适用于 Visual Studio 的 .NET 升级助手扩展。](#)

评估

在执行升级之前，应分析项目。使用 .NET 升级助手对项目执行代码分析会生成可以引用的报表，以识别潜在的迁移阻止程序。

若要分析项目并生成报表，请右键单击解决方案资源管理器中的解决方案文件，然后选择“[升级](#)”。有关执行分析的详细信息，请参阅 [使用 .NET 升级助手分析项目](#)。

迁移依赖项

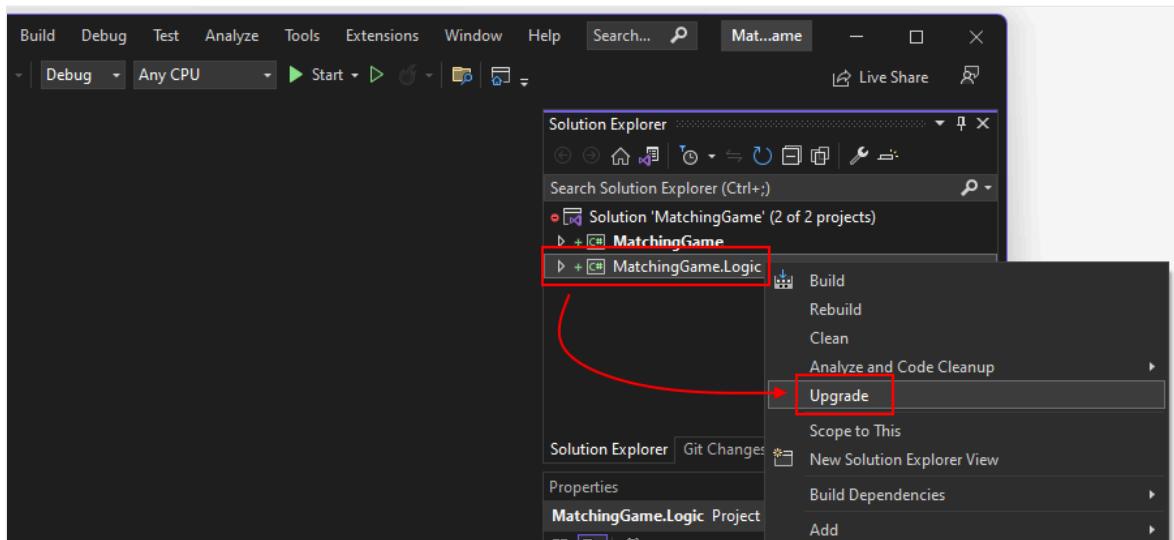
如果要升级多个项目，请从没有依赖项的项目开始。在匹配游戏示例中，MatchingGame 项目依赖于 MatchingGame.Logic 库，因此应首先升级 MatchingGame.Logic。

提示

请务必对代码进行备份，例如在源代码管理中或副本中。

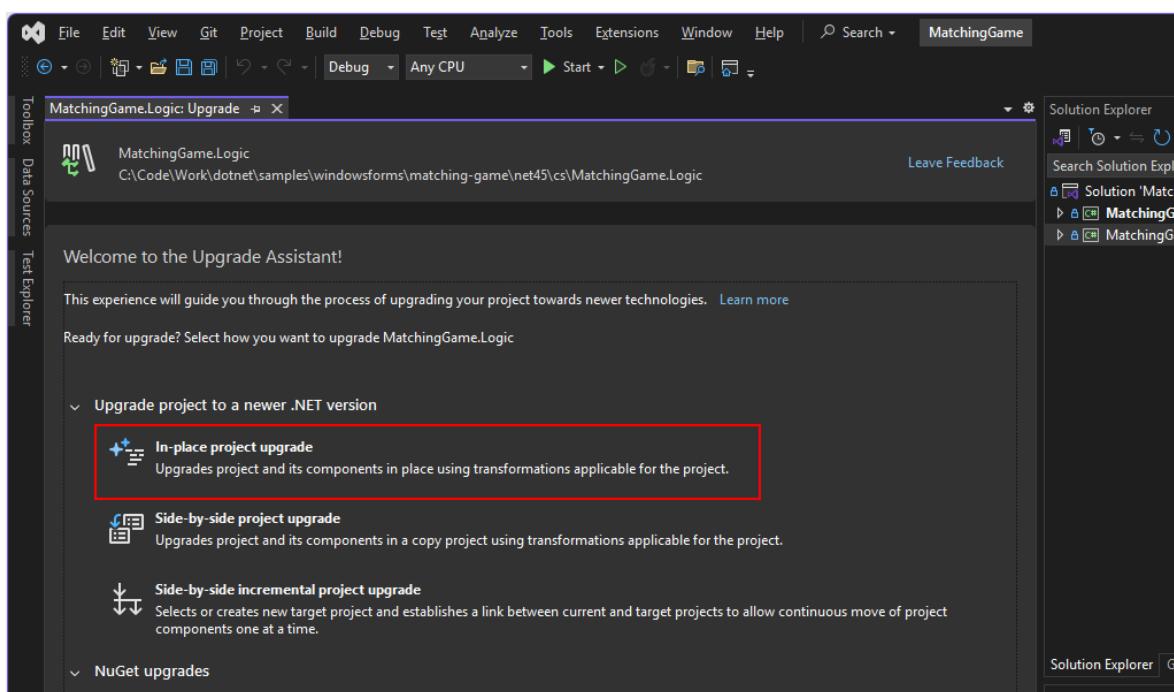
使用以下步骤在 Visual Studio 中升级项目：

1. 右键单击“[解决方案资源管理器](#)”窗口中的 MatchingGame.Logic 项目，然后选择“[升级](#)”：



将打开一个新选项卡，提示你选择要执行的升级。

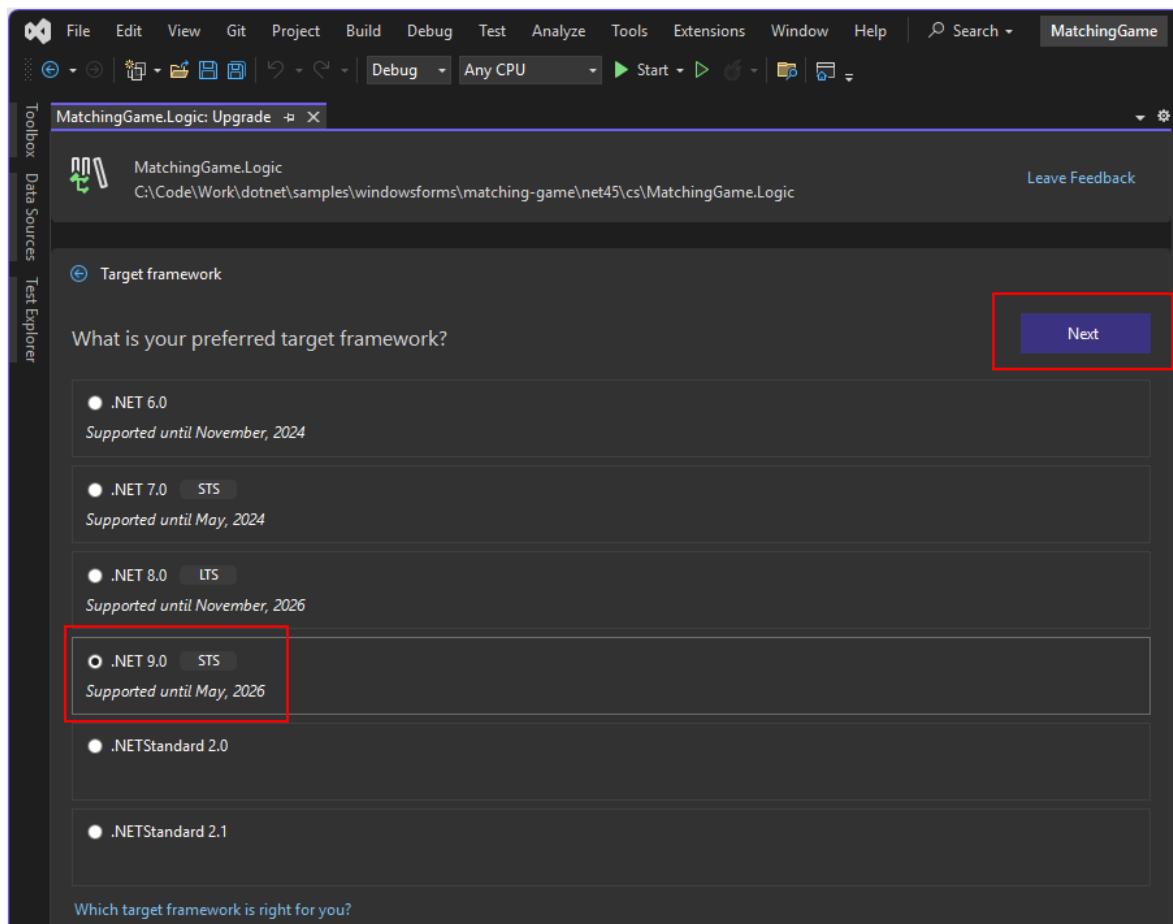
2. 选择“就地项目升级”。



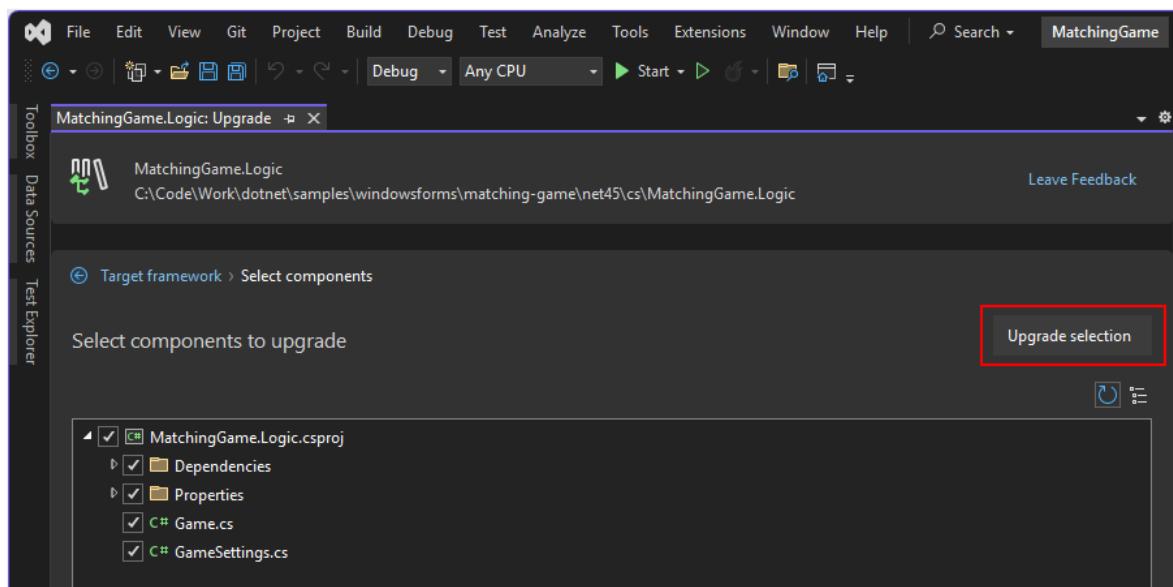
3. 接下来，选择目标框架。

根据要升级的项目类型，会显示不同的选项。.NET Framework 和 .NET 都可以使用 .NET Standard 2.0。如果库不依赖于桌面技术（如Windows 窗体），这是一个不错的选择，此项目确实如此。

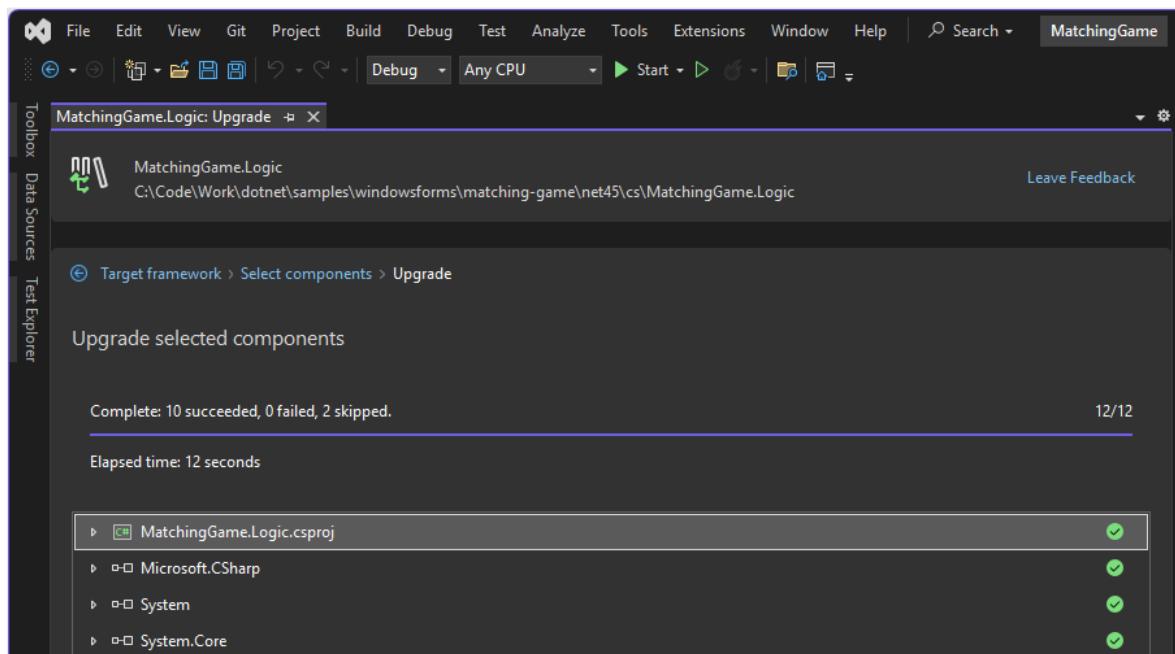
选择 .NET 9.0，然后选择“下一步”。



4. 此时将显示一个树状结构，其中包含与项目相关的所有工件，例如代码文件和库。可以升级单个生成生成工件或整个项目（这是默认设置）。选择“**升级选择**”以开始升级。



5. 升级完成后，将显示结果：



具有实心绿色圆圈的生成工件已升级，而具有空绿色圆圈的生成工件则被跳过。跳过的生成工件意味着升级助手没有找到任何可以升级的内容。

现在应用程序的支持库已升级，请升级主应用程序。

Visual Basic 项目说明

目前，.NET 升级助手无法识别在.NET Framework 上的 Visual Basic 模板创建的设置文件中的用法 `System.Configuration`。它也不尊重使用 `My` .NET Framework 项目中使用的扩展，例如 `My.Computer` 和 `My.User`。这些扩展已在.NET 中删除。由于这两个问题，使用 .NET 升级助手迁移后，Visual Basic 库不会编译。

若要解决此问题，项目必须面向 Windows 并引用 Windows 窗体。

1. 迁移完成后，双击解决方案资源管理器窗口中的 MatchingGame.Logic 项目。
2. 找到 `<Project>/<PropertyGroup>` 元素。
3. 在 XML 编辑器中，将值 `<TargetFramework>` 从中 `net9.0` 更改为 `net9.0-windows`。
4. 添加到 `<UseWindowsForms>true</UseWindowsForms>` 后面的 `<TargetFramework>` 行。

项目设置应类似于以下代码片段：

```
XML

<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net9.0-windows</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
    <OutputType>Library</OutputType>
    <MyType>Windows</MyType>
  </PropertyGroup>
</Project>
```

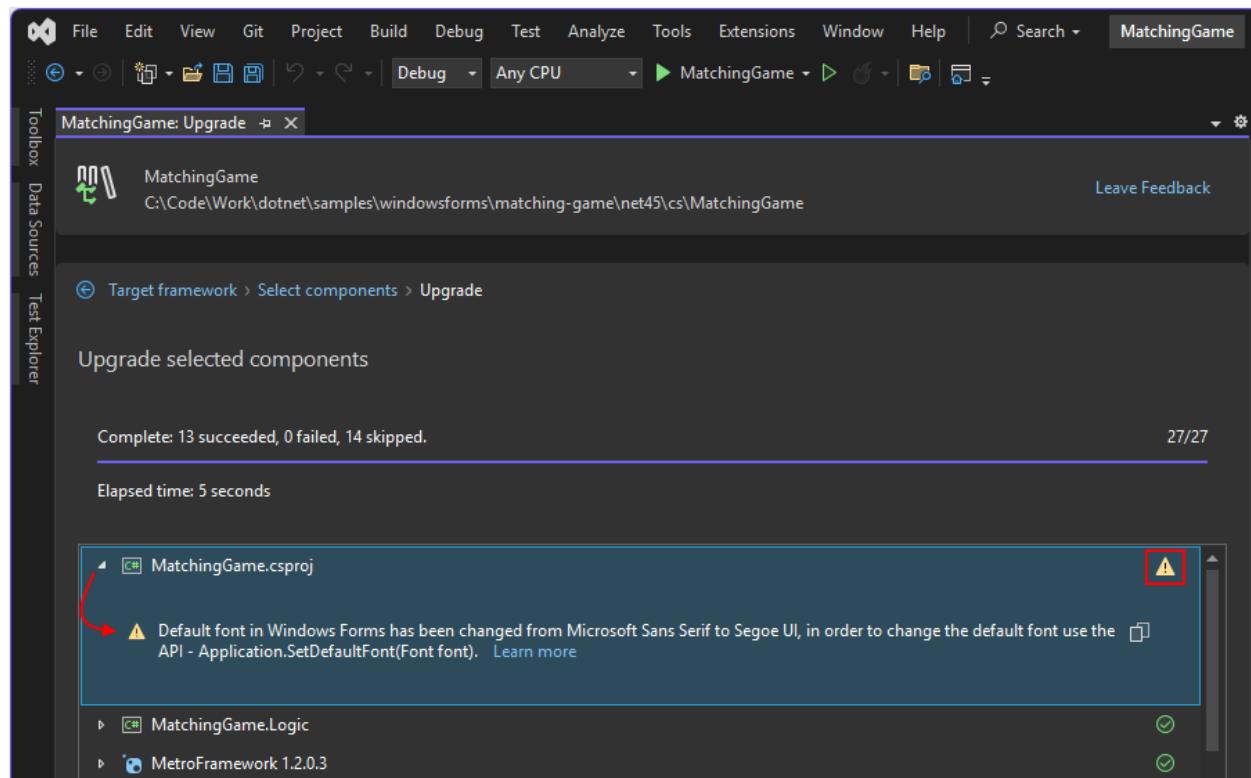
... other settings removed for brevity ...

迁移主项目

升级所有支持库后，即可升级主应用项目。对于示例应用程序，只有一个库项目需要升级，该项目已在上一部分中升级。

1. 右键单击“解决方案资源管理器”窗口中的 MatchingGame 项目，然后选择“**升级**”：
2. 选择“**就地项目升级**”。
3. 选择目标框架的 .NET 9.0，然后选择“**下一步**”。
4. 保留选中所有项目，然后选择“**升级选择**”。

升级完成后，将显示结果。请注意 Windows 窗体项目为何具有警告符号。展开该项并显示有关该步骤的详细信息：



请注意，项目升级组件提到默认字体已更改。由于字体可能会影响控件布局，因此你需要检查项目中的每一个窗体和自定义控件，以确保 UI 正确排列。

生成干净的内部版本

升级主项目后，清理并编译它。

1. 右键单击“解决方案资源管理器”窗口中的 MatchingGame 项目，然后选择“**清理**”。
2. 右键单击“解决方案资源管理器”窗口中的 MatchingGame 项目，然后选择“**生成**”。

如果应用程序遇到任何错误，可以在“[错误列表](#)”窗口中找到这些错误，并提供修复这些错误的建议。

Windows 窗体匹配游戏示例项目现已升级到 .NET 9。

升级后体验

如果要将应用从 .NET Framework 移植到 .NET，请查看 [从 .NET Framework 升级到 .NET 后的现代化文章](#)。

相关内容

- [从 .NET Framework 移植到 .NET。](#)

移植指南概述了将代码从 .NET Framework 移植到 .NET 时应考虑的事项。项目的复杂性决定了在项目文件的初始迁移之后要做多少工作。

- [从 .NET Framework 升级到 .NET 后实现现代化。](#)

自 .NET Framework 以来，.NET 的世界发生了很大的变化。此链接提供有关在升级后如何实现应用现代化的某些信息。

事件概述 (Windows 窗体 .NET)

项目 · 2024/11/05

事件是可以通过代码响应或“处理”的操作。事件可由用户操作（例如单击鼠标或按某个键）、程序代码或系统生成。

事件驱动的应用程序执行代码以响应事件。每个窗体和控件都公开一组预定义事件，你可根据这些事件进行编程。如果发生其中一个事件并且在相关联的事件处理程序中有代码，则调用该代码。

对象引发的事件类型会发生变化，但对于大多数控件，很多类型都是通用的。例如，大多数对象都会处理 [Click](#) 事件。如果用户单击窗体，就会执行窗体的 [Click](#) 事件处理程序内的代码。

① 备注

许多事件会与其他事件同时发生。例如，在发生 [DoubleClick](#) 事件期间，还会发生 [MouseDown](#)、[MouseUp](#) 以及 [Click](#) 事件。

有关如何引发和使用事件的信息，请参阅[处理和引发事件](#)。

委托及其角色

委托是 .NET 中通常用于建立事件处理机制的类。委托大体上相当于 Visual C++ 和其他面向对象语言中常用的函数指针。但与函数指针不同的是，委托是面向对象的、类型安全的和保险的。另外，函数指针只包含对特定函数的引用，而委托由对对象的引用以及对该对象内一个或多个方法的引用组成。

此事件模型使用“委托”将事件绑定到用来处理它们的方法。委托允许其他类通过指定处理程序方法来注册事件通知。当发生事件时，委托会调用绑定的方法。有关如何定义委托的详细信息，请参阅[处理和引发事件](#)。

委托可绑定到单个方法或多个方法，后者又称为多路广播。创建事件的委托时，通常会创建多播事件。极少的例外情况是，事件可能会生成一个特定过程（例如显示对话框），而该过程在逻辑上不在每个事件中重复多次。有关如何创建多播委托的信息，请参阅[如何合并委托 \(多播委托\)](#)。

多播委托维护它所绑定到的方法的调用列表。多路广播委托支持将方法添加到调用列表的 [Combine](#) 方法以及将其移除的 [Remove](#) 方法。

当应用程序记录某个事件时，该控件将通过调用该事件的委托引发事件。 委托依次调用绑定的方法。 最常见的情况（多播委托）是，委托依次调用调用列表中的每个绑定方法，这样可提供一对多通知。 此策略意味着控件不需要维护事件通知的目标对象列表 - 委托可处理所有的注册和通知。

委托还允许将多个事件绑定到同一个方法上，从而允许多对一通知。 例如，单击按钮事件和单击菜单命令事件都能调用同一委托，然后该委托调用单个方法以相同方式处理各个事件。

与委托一起使用的绑定机制是动态的：委托可在运行时绑定到其签名与事件处理程序的签名相匹配的任何方法上。 借助此功能，你可以根据条件设置或更改绑定方法，并动态地将事件处理程序附加到控件上。

另请参阅

- [处理和引发事件](#)

自动缩放 (Windows 窗体 .NET)

项目 • 2024/12/18

自动缩放功能使窗体及其控件可以在一台设有特定显示分辨率或字体的计算机上设计后，能够在另一台具有不同显示分辨率或字体的计算机上正确显示。它确保窗体及其控件可智能调整大小，使其与用户和其他开发人员计算机上的本机窗口和其他应用程序保持一致。自动缩放和视觉样式使 Windows 窗体应用程序能够与每个用户计算机上的本机 Windows 应用程序保持一致的外观和体验。

在 Windows 窗体中，自动缩放在大多数情况下按预期效果工作。但是，字体方案更改可能会有问题。

需要自动缩放

如果没有自动缩放，设计用于一个显示分辨率或字体的应用程序会在更改分辨率或字体时显示过小或过大。例如，如果应用程序以 Tahoma 9 点作为基准设计，那么在系统字体为 Tahoma 12 点的计算机上运行时，会显得太小。文本元素（如标题、菜单、文本框内容等）在显示时会比其他应用程序的小。此外，包含文本的用户界面（UI）元素的大小，如标题栏、菜单和许多控件都依赖于使用的字体。在此示例中，这些元素也会显得相对较小。

当应用程序设计为特定显示分辨率时，会出现类似的情况。最常见的显示分辨率是每英寸 96 个点（DPI），相当于 100% 显示缩放，但支持 125%、150%、200%（分别相当于 120、144 和 192 DPI）及更高的分辨率的显示设备变得越来越常见。如果不进行调整，应用程序（尤其是基于图形的应用程序）在以另一分辨率运行时，针对一个分辨率设计的分辨率将显得太大或太小。

自动缩放通过根据相对字号或显示分辨率自动调整窗体及其子控件的大小来解决这些问题。Windows 操作系统支持使用称为对话框单位的相对度量单位自动缩放对话框。对话单元基于系统字体，其与像素的关系可以通过 Win32 SDK 函数 `GetDialogBaseUnits` 确定。当用户更改 Windows 使用的主题时，将自动相应地调整所有对话框。此外，Windows 窗体支持根据默认系统字体或显示分辨率自动缩放。（可选）可以在应用程序中禁用自动缩放。

⊗ 注意

不支持 DPI 和字体缩放模式的任意混合。尽管可以使用一种模式（例如 DPI）对用户控件进行缩放，并使用另一种模式（字体）将其放置在窗体上而没有任何问题，但如果基准窗体使用一种模式，而派生窗体使用另一种模式，可能会导致意外的结果。

自动缩放操作

Windows 窗体使用以下逻辑自动缩放窗体及其内容：

1. 在设计时，每个 `ContainerControl` 分别在 `AutoScaleMode` 和 `AutoScaleDimensions` 中记录缩放模式和当前分辨率。
2. 在运行时，实际分辨率存储在 `CurrentAutoScaleDimensions` 属性中。
`AutoScaleFactor` 属性动态计算运行时和设计时缩放分辨率之间的比率。
3. 当窗体加载时，如果 `CurrentAutoScaleDimensions` 和 `AutoScaleDimensions` 的值不同，则调用 `PerformAutoScale` 方法以调整控件及其子控件。此方法挂起布局并调用 `Scale` 方法来执行实际缩放。之后，将更新 `AutoScaleDimensions` 的值以防止渐进式缩放。
4. 在以下情况下，也会自动调用 `PerformAutoScale`：
 - 当缩放模式为 `Font` 时，响应 `OnFontChanged` 事件。
 - 当容器控件的布局恢复并且 `AutoScaleDimensions` 或 `AutoScaleMode` 属性中检测到更改时。
 - 如上所述，当父 `ContainerControl` 进行缩放时。每个容器控件都负责使用自己的缩放因子（而不是其父容器的缩放因子）缩放其子控件。
5. 子控件可以通过多种方式修改其缩放行为：
 - 可以重写此 `ScaleChildren` 属性，以决定其子控件是否应进行缩放。
 - 可以重写 `GetScaledBounds` 方法以调整控件缩放到的边界，但不能调整缩放逻辑。
 - 可以重写 `ScaleControl` 方法以更改当前控件的缩放逻辑。

另请参阅

- [AutoScaleMode](#)
- [Scale](#)
- [PerformAutoScale](#)
- [AutoScaleDimensions](#)

如何向项目添加窗体（Windows 窗体 .NET）

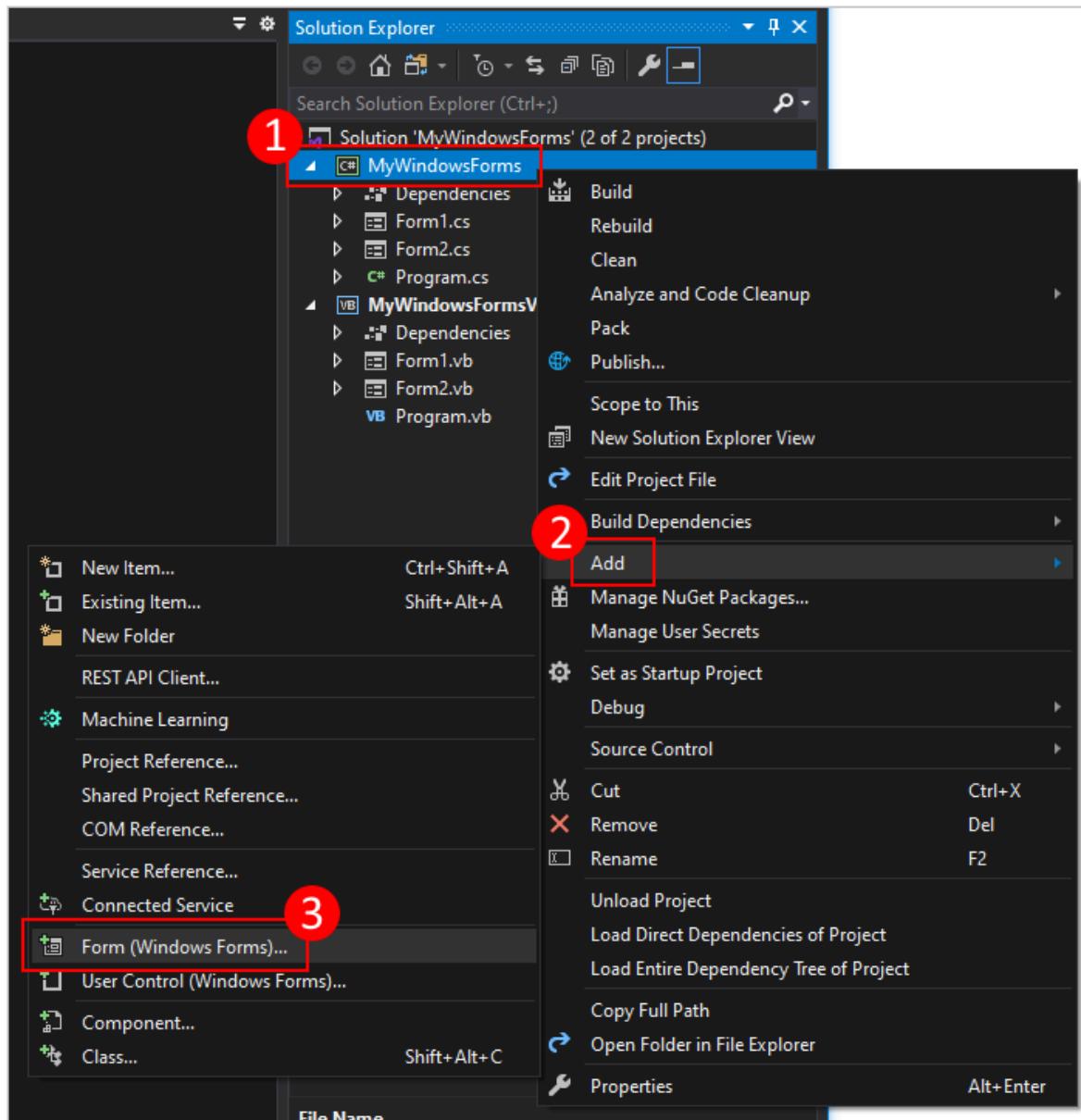
项目 • 2024/12/20

在 Visual Studio 中向项目添加窗体。如果应用具有多个窗体，你可以选择哪个作为应用的启动窗体，并且可以同时显示多个窗体。

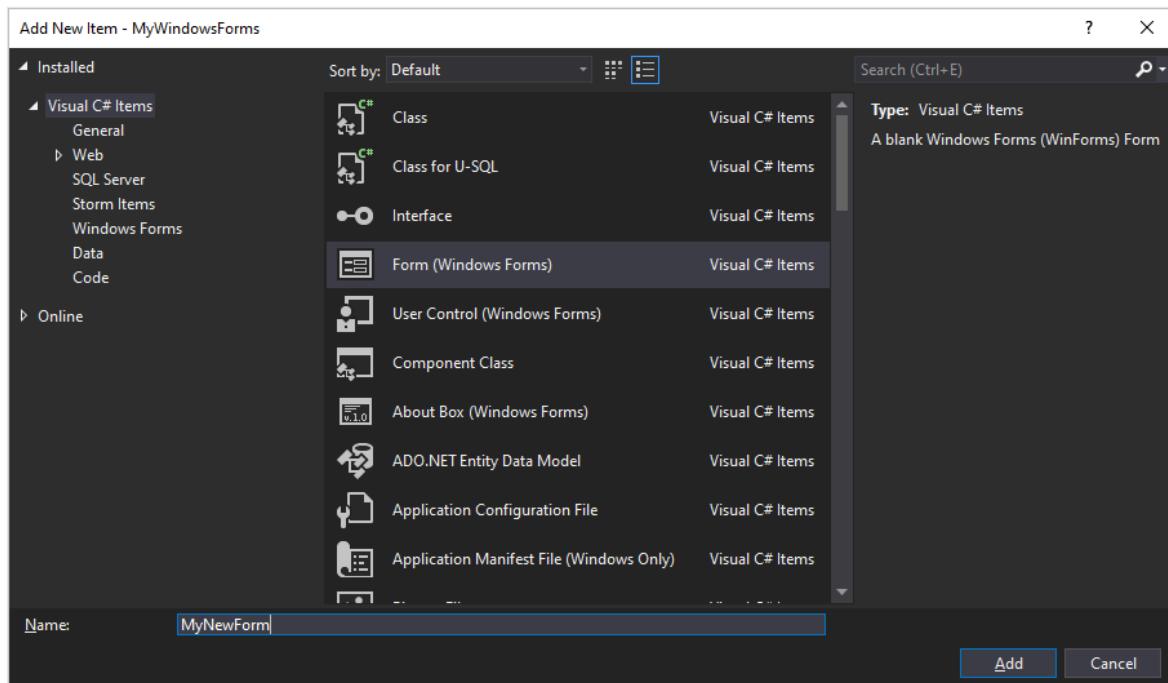
添加新窗体

在 Visual Studio 中添加新窗体。

1. 在 Visual Studio 中，找到“项目资源管理器”窗格。右键单击项目，然后选择“添加”“窗体(Windows 窗体)”。



- 在“名称”框中键入窗体的名称，例如 MyNewForm。Visual Studio 将提供一个默认名称，这也是你可以使用的唯一名称。



添加窗体后，Visual Studio 将打开窗体的窗体设计器。

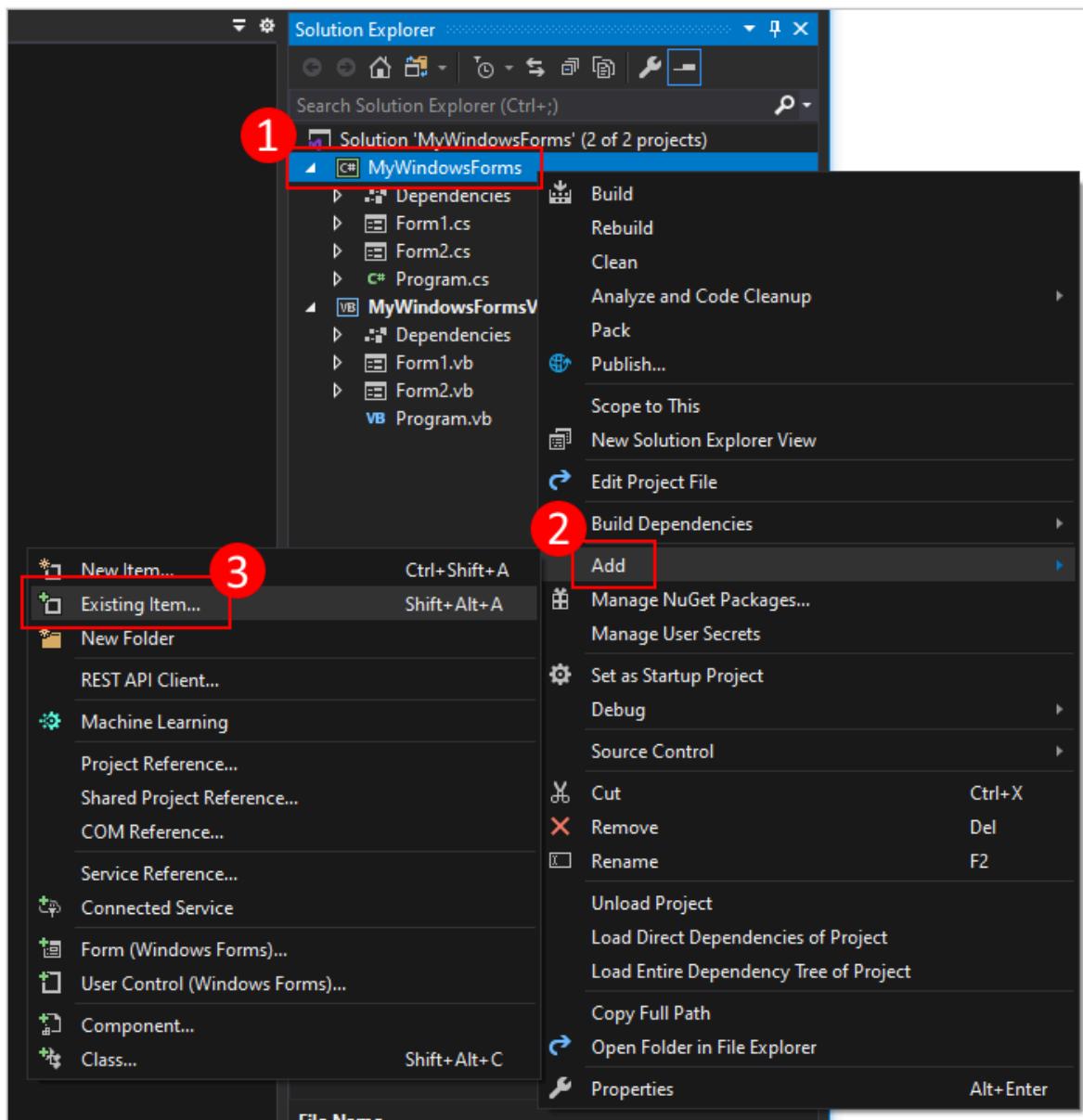
向窗体添加项目引用

如果源文件包含在窗体中，则可以通过将文件复制到项目所在文件夹来将窗体添加到项目。项目会自动引用项目的同一文件夹或子文件夹中的所有代码文件。

窗体由两个同名文件构成：form2.cs（form2 是文件名的示例）和 form2.Designer.cs。有时会存在同名资源文件，即 form2.resx。如上一示例中所示，form2 表示基本文件名。建议将所有相关文件复制到项目文件夹。

也可使用 Visual Studio 将文件导入项目。将现有文件添加到项目时，该文件将复制到项目所在文件夹。

- 在 Visual Studio 中，找到“项目资源管理器”窗格。右键单击项目，然后选择“添加”“现有项”>。



2. 导航到包含窗体文件的文件夹。

3. 选择 form2.cs 文件，其中 form2 是相关窗体文件的基本文件名。请勿选择其他文件，例如 form2.Designer.cs。

另请参阅

- 如何定位窗体并重设其大小 (Windows 窗体 .NET)
- 事件概述 (Windows 窗体 .NET)
- 控件的位置和布局 (Windows 窗体 .NET)

如何定位窗体并重设其大小 (Windows 窗体 .NET)

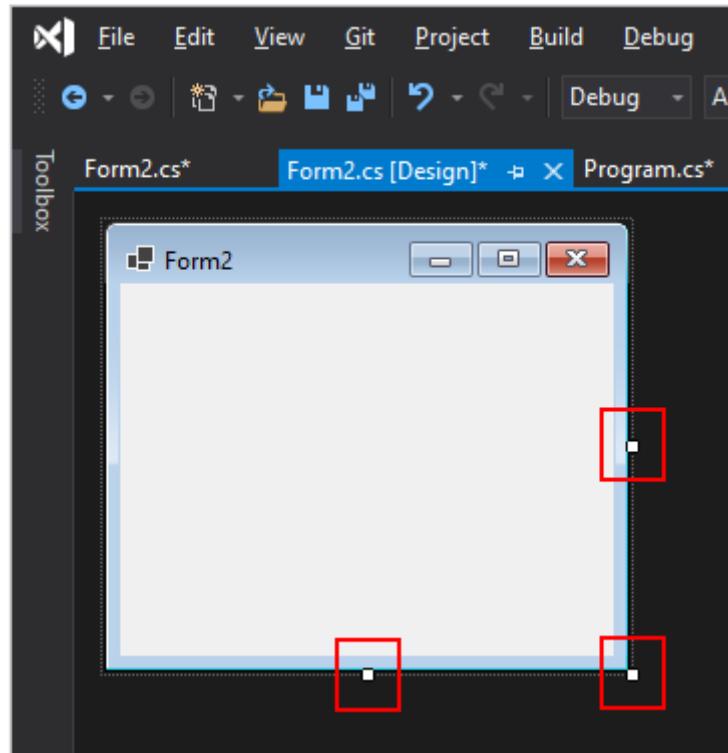
项目 • 2024/11/05

创建窗体时，大小和位置最初设置为默认值。窗体的默认大小通常为 800x500 像素的宽度和高度。显示窗体时的初始位置取决于一些不同的设置。

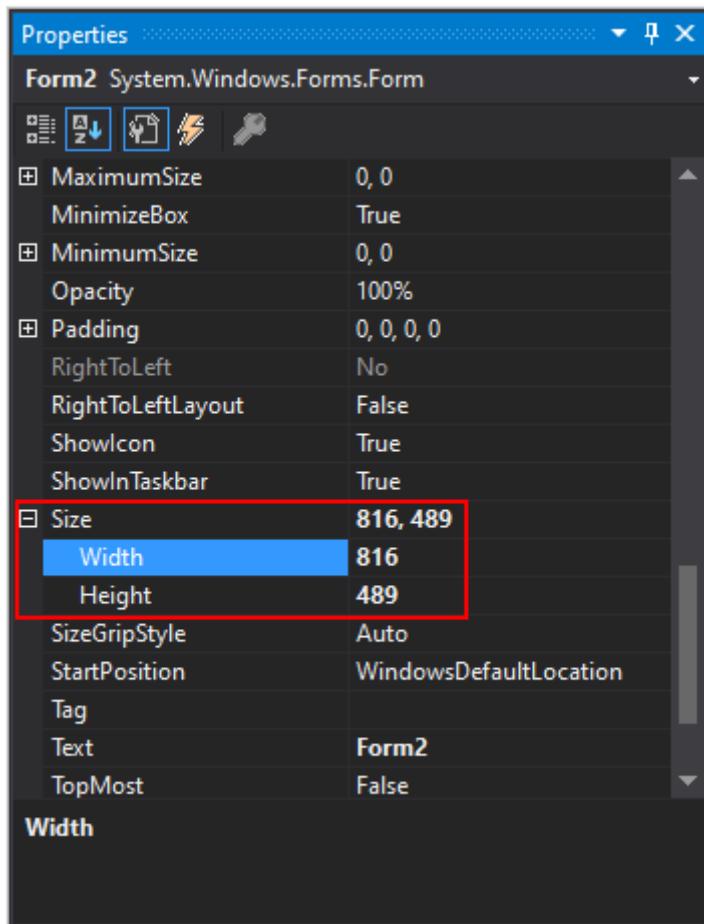
可以在设计时使用 Visual Studio 更改窗体的大小，也可以在运行时使用代码更改窗体的大小。

使用设计器重设大小

在将新窗体添加到项目后，可以通过两种不同的方式来设置窗体的大小。首先，可使用设计器中的大小调整手柄来设置它。通过拖动右边缘、底边缘或角，可以调整窗体的大小。



在设计器处于打开状态时，第二种重设窗体大小的方法是通过“属性”窗格。选择窗体，然后在 Visual Studio 中找到“属性”窗格。向下滚动至“大小”并将其展开。可以手动设置“宽度”和“高度”。



在代码中重设大小

尽管设计器会设置窗体的起始大小，也可以通过代码来重设窗体大小。当应用程序的某些相关内容确定窗体的默认大小不够时，使用代码来调整窗体的大小很有用。

若要重设窗体的大小，请更改 `Size`，它表示窗体的宽度和高度。

重设当前窗体的大小

只要代码在窗体的上下文中运行，就可以更改当前窗体的大小。例如，如果 `Form1` 上有一个按钮，则单击该按钮时，将调用 `Click` 事件处理程序来重设窗体的大小：

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Size = new Size(250, 200);
```

重设其他窗体的大小

创建其他窗体后，可以使用引用该窗体的变量来更改其大小。例如，假设有两个窗体：`Form1`（在本示例中为启动窗体）和 `Form2`。`Form1` 有一个按钮，单击该按钮时，将调用

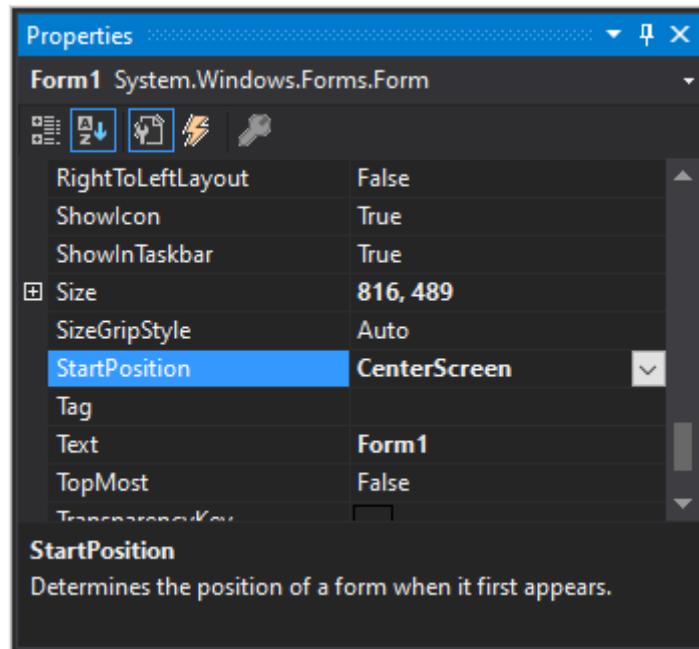
`Click` 事件。此事件的处理程序创建 `Form2` 窗体的新实例、设置大小，然后显示该实例：

```
C#  
  
private void button1_Click(object sender, EventArgs e)  
{  
    Form2 form = new Form2();  
    form.Size = new Size(250, 200);  
    form.Show();  
}
```

如果未手动设置 `Size`，则窗体的默认大小即在设计时设置的大小。

使用设计器定位

创建并显示窗体实例时，窗体的初始位置由 `StartPosition` 属性决定。`Location` 属性保存窗体的当前位置。可以通过设计器设置这两个属性。



展开表

FormStartPosition	说明
Enum	
CenterParent	窗体在父窗体的边界内居中。
CenterScreen	窗体在当前显示屏中居中。
手动	窗体的位置由 <code>Location</code> 属性决定。

FormStartPosition	说明
Enum	
WindowsDefaultBounds	窗体位于 Windows 默认位置，且大小重设为 Windows 决定的默认大小。
WindowsDefaultLocation	窗体位于 Windows 默认位置，没有重设大小。

`CenterParent` 值仅适用于多文档界面 (MDI) 子窗体或使用 `ShowDialog` 方法显示的常规窗体。`CenterParent` 不影响使用 `Show` 方法显示的常规窗体。 若要让某个窗体 (`form` 变量) 相对于另一个窗体 (`parentForm` 变量) 居中，请使用以下代码：

C#

```
form.StartPosition = FormStartPosition.Manual;
form.Location = new Point(parentForm.Width / 2 - form.Width / 2 +
parentForm.Location.X,
                           parentForm.Height / 2 - form.Height / 2 +
parentForm.Location.Y);
form.Show();
```

使用代码定位

尽管可以使用设计器来设置窗体的起始位置，也可以使用代码来更改起始位置模式或手动设置位置。如果需要相对于屏幕或其他窗体手动定位窗体并重设其大小，则使用代码来定位窗体非常有用。

移动当前窗体

只要代码在窗体的上下文中运行，就可以移动当前窗体。例如，如果 `Form1` 上有一个按钮，则单击该按钮时，将调用 `Click` 事件处理程序。此示例中的处理程序通过设置 `Location` 属性将窗体的位置更改为屏幕的左上角：

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Location = new Point(0, 0);
```

定位其他窗体

创建其他窗体后，可以使用引用该窗体的变量来更改其位置。例如，假设有两个窗体：`Form1`（在本示例中为启动窗体）和 `Form2`。`Form1` 有一个按钮，单击该按钮时，将调用

`Click` 事件。此事件的处理程序创建 `Form2` 窗体的新实例并设置位置：

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 form = new Form2();
    form.Location = new Point(0, 0);
    form.Show();
}
```

如果未设置 `Location`，则窗体的默认位置基于 `StartPosition` 属性在设计时的设置。

另请参阅

- [如何向项目添加窗体 \(Windows 窗体 .NET\)](#)
- [事件概述 \(Windows 窗体 .NET\)](#)
- [控件的位置和布局 \(Windows 窗体 .NET\)](#)

如何定位窗体并重设其大小 (Windows 窗体 .NET)

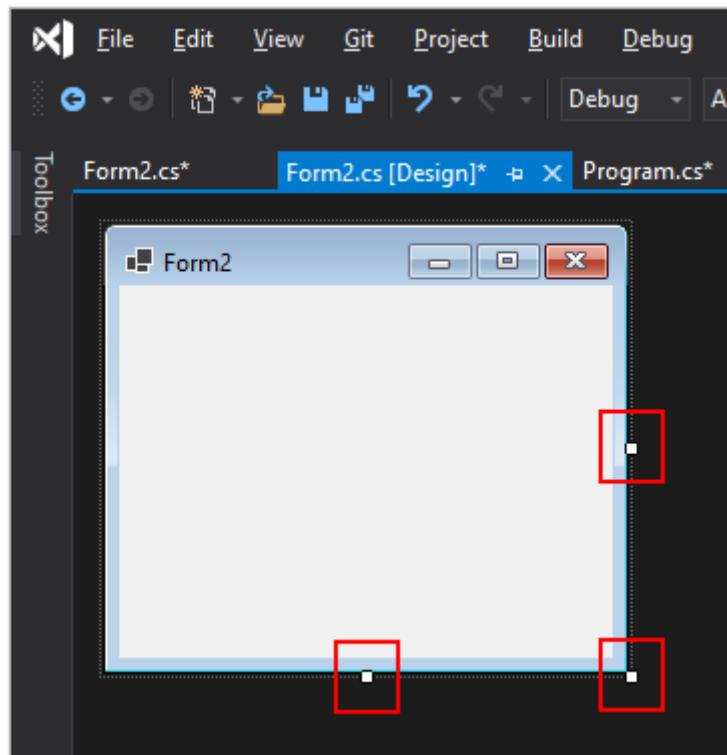
项目 • 2024/11/05

创建窗体时，大小和位置最初设置为默认值。窗体的默认大小通常为 800x500 像素的宽度和高度。显示窗体时的初始位置取决于一些不同的设置。

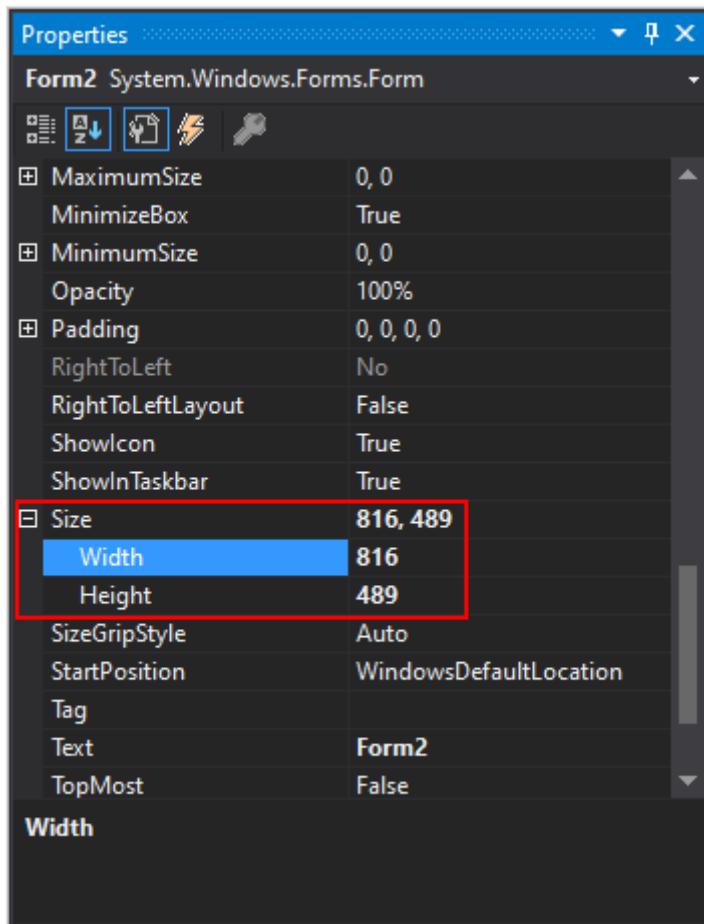
可以在设计时使用 Visual Studio 更改窗体的大小，也可以在运行时使用代码更改窗体的大小。

使用设计器重设大小

在将[新窗体添加到项目](#)后，可以通过两种不同的方式来设置窗体的大小。首先，可使用设计器中的大小调整手柄来设置它。通过拖动右边缘、底边缘或角，可以调整窗体的大小。



在设计器处于打开状态时，第二种重设窗体大小的方法是通过“属性”窗格。选择窗体，然后在 Visual Studio 中找到“属性”窗格。向下滚动至“大小”并将其展开。可以手动设置“宽度”和“高度”。



在代码中重设大小

尽管设计器会设置窗体的起始大小，也可以通过代码来重设窗体大小。当应用程序的某些相关内容确定窗体的默认大小不够时，使用代码来调整窗体的大小很有用。

若要重设窗体的大小，请更改 `Size`，它表示窗体的宽度和高度。

重设当前窗体的大小

只要代码在窗体的上下文中运行，就可以更改当前窗体的大小。例如，如果 `Form1` 上有一个按钮，则单击该按钮时，将调用 `Click` 事件处理程序来重设窗体的大小：

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Size = new Size(250, 200);
```

重设其他窗体的大小

创建其他窗体后，可以使用引用该窗体的变量来更改其大小。例如，假设有两个窗体：`Form1`（在本示例中为启动窗体）和 `Form2`。`Form1` 有一个按钮，单击该按钮时，将调用

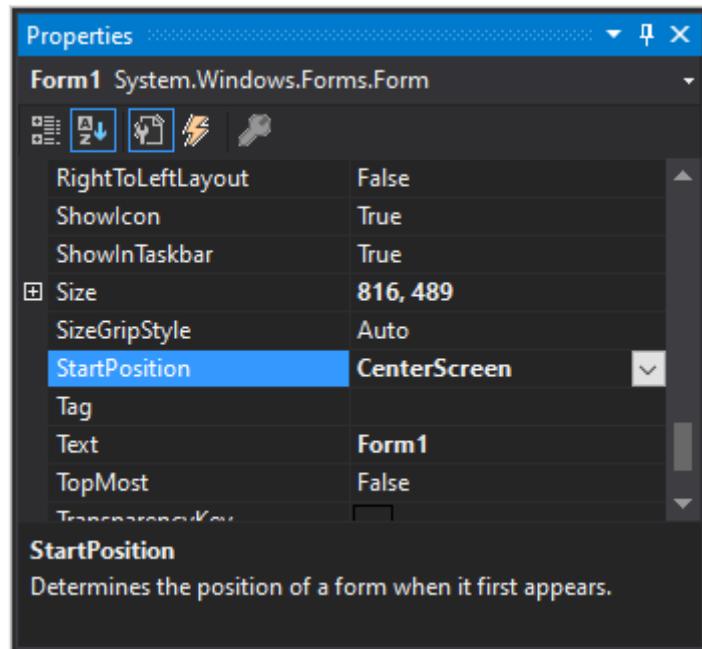
`Click` 事件。此事件的处理程序创建 `Form2` 窗体的新实例、设置大小，然后显示该实例：

```
C#  
  
private void button1_Click(object sender, EventArgs e)  
{  
    Form2 form = new Form2();  
    form.Size = new Size(250, 200);  
    form.Show();  
}
```

如果未手动设置 `Size`，则窗体的默认大小即在设计时设置的大小。

使用设计器定位

创建并显示窗体实例时，窗体的初始位置由 `StartPosition` 属性决定。`Location` 属性保存窗体的当前位置。可以通过设计器设置这两个属性。



展开表

FormStartPosition	说明
Enum	
CenterParent	窗体在父窗体的边界内居中。
CenterScreen	窗体在当前显示屏中居中。
手动	窗体的位置由 <code>Location</code> 属性决定。

FormStartPosition	说明
Enum	
WindowsDefaultBounds	窗体位于 Windows 默认位置，且大小重设为 Windows 决定的默认大小。
WindowsDefaultLocation	窗体位于 Windows 默认位置，没有重设大小。

`CenterParent` 值仅适用于多文档界面 (MDI) 子窗体或使用 `ShowDialog` 方法显示的常规窗体。`CenterParent` 不影响使用 `Show` 方法显示的常规窗体。 若要让某个窗体 (`form` 变量) 相对于另一个窗体 (`parentForm` 变量) 居中，请使用以下代码：

C#

```
form.StartPosition = FormStartPosition.Manual;
form.Location = new Point(parentForm.Width / 2 - form.Width / 2 +
parentForm.Location.X,
                           parentForm.Height / 2 - form.Height / 2 +
parentForm.Location.Y);
form.Show();
```

使用代码定位

尽管可以使用设计器来设置窗体的起始位置，也可以使用代码来更改起始位置模式或手动设置位置。如果需要相对于屏幕或其他窗体手动定位窗体并重设其大小，则使用代码来定位窗体非常有用。

移动当前窗体

只要代码在窗体的上下文中运行，就可以移动当前窗体。例如，如果 `Form1` 上有一个按钮，则单击该按钮时，将调用 `Click` 事件处理程序。此示例中的处理程序通过设置 `Location` 属性将窗体的位置更改为屏幕的左上角：

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Location = new Point(0, 0);
```

定位其他窗体

创建其他窗体后，可以使用引用该窗体的变量来更改其位置。例如，假设有两个窗体：`Form1`（在本示例中为启动窗体）和 `Form2`。`Form1` 有一个按钮，单击该按钮时，将调用

`Click` 事件。此事件的处理程序创建 `Form2` 窗体的新实例并设置位置：

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 form = new Form2();
    form.Location = new Point(0, 0);
    form.Show();
}
```

如果未设置 `Location`，则窗体的默认位置基于 `StartPosition` 属性在设计时的设置。

另请参阅

- [如何向项目添加窗体 \(Windows 窗体 .NET\)](#)
- [事件概述 \(Windows 窗体 .NET\)](#)
- [控件的位置和布局 \(Windows 窗体 .NET\)](#)

控件使用概述 (Windows 窗体 .NET)

项目 · 2024/12/18

Windows 窗体控件是可重用组件，用于封装用户界面功能，并在基于 Windows 的客户端应用程序中使用。Windows 窗体不仅提供许多现成的控件，还提供用于开发自己控件的架构。可以合并现有控件、扩展现有控件或创作自己的自定义控件。有关详细信息，请参阅 [自定义控件的类型](#)。

添加控件

控件通过 Visual Studio 设计器添加。使用设计器，可以放置、调整控件大小、对齐和移动控件。或者，可以通过代码添加控件。有关详细信息，请参阅 [添加控件 \(Windows 窗体\)](#)。

布局选项

控件在父控件中显示的位置是由相对于父控件表面左上角的 [Location](#) 属性值确定的。在父级中的左上角位置坐标为 (x_0, y_0) 。控件的大小由 [Size](#) 属性确定，表示控件的宽度和高度。

除了手动定位和调整大小外，还提供多种容器控件以协助控件的自动布局。

有关详细信息，请参阅 [控件的位置和布局](#) 和 [如何停靠和定位控件](#)。

控制事件

控件通过基类 [Control](#) 提供 60 多个事件。其中包括 [Paint](#) 事件，导致绘制控件、与显示窗口相关的事件，例如 [Resize](#) 和 [Layout](#) 事件，以及低级别鼠标和键盘事件。某些低级别事件通过 [Control](#) 合成为语义事件，例如 [Click](#) 和 [DoubleClick](#)。大多数共享事件属于以下类别：

- 鼠标事件
- 键盘事件
- 属性更改事件
- 其他事件

并非每个控件都响应每个事件。例如，[Label](#) 控件不响应键盘输入，并且不会引发 [Control.PreviewKeyDown](#) 事件。

通常，控件是一个底层 Win32 控件的封装器，并且使用 [Paint](#) 事件在控件顶部绘制可能会受到限制，甚至根本不执行任何操作，因为控件最终由 Windows 绘制。

有关详细信息，请参阅 [控制事件](#) 和 [如何处理控件事件](#)。

控制辅助功能

Windows 窗体支持屏幕阅读器和用于语言命令的语言输入工具的辅助功能。但是，您必须重视可访问性来设计 UI。Windows 窗体控件公开各种属性来支持辅助功能。有关这些属性的详细信息，请参阅[“为控件提供辅助功能信息”](#)。

另请参阅

- [控件的位置和布局](#)
- [标签控件概述](#)
- [控制事件](#)
- [自定义控件的类型](#)
- [在控件上进行绘画和图画](#)
- [为控件提供辅助功能信息](#)

控件的位置和布局 (Windows 窗体 .NET)

项目 • 2025/01/31

Windows 窗体中的控件位置不仅由控件确定，还由控件的父级确定。本文介绍控件提供的不同设置以及影响布局的不同类型的父容器。

固定位置和大小

控件在父级上的位置由父级表面左上角的 [Location](#) 属性的值确定。父级中左上角的位置坐标为 (x_0, y_0) 。控件的大小由 [Size](#) 属性确定，表示控件的宽度和高度。

控件相对于容器 的 位置

当控件被添加到支持自动定位的父控件时，该控件的位置和大小会发生改变。在这种情况下，可能无法手动调整控件的位置和大小，具体取决于父级的类型。

[MaximumSize](#) 和 [MinimumSize](#) 属性有助于设置控件可以使用的最小和最大空间。

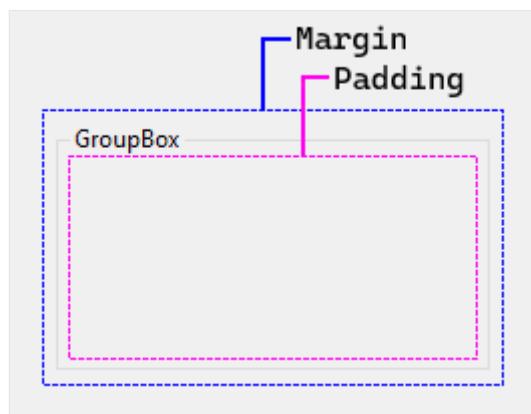
边距和填充

有两个控件属性可帮助精确放置控件：[Margin](#) 和 [Padding](#)。

[Margin](#) 属性定义控件周围的空间，该空间使其他控件与控件的边框保持指定距离。

[Padding](#) 属性定义控件内部的空间，该空间用于保持控件的内容（例如，其 [Text](#) 属性的值）与控件边框保持一个指定的距离。

下图显示了控件上的 [Margin](#) 和 [Padding](#) 属性。



在定位和调整控件大小时，Visual Studio 设计器将遵循这些属性。对齐线显示为参考线，有助于你保持在控件的指定边距之外。例如，在将控件拖到另一个控件旁边时，

Visual Studio 会显示对齐线：



自动放置和大小

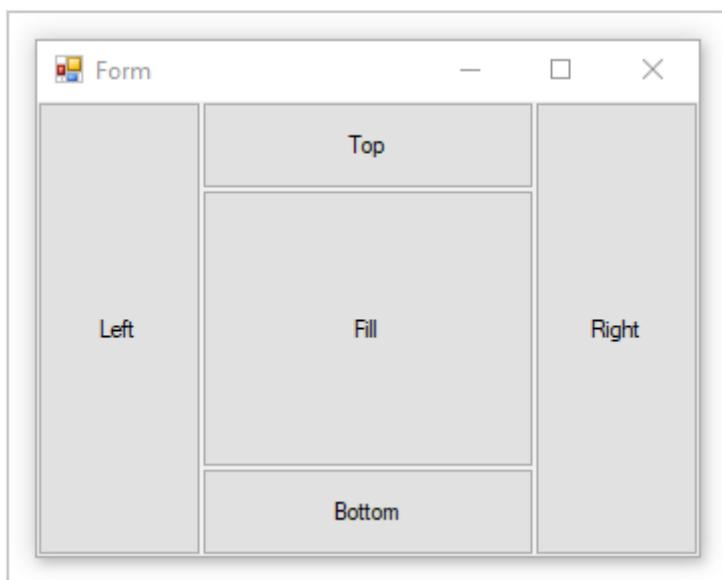
控件可以自动放置在其父级中。某些父级容器会强制放置，而另一些容器会遵循指导放置的控件设置。控件上有两个属性可帮助在父级中实现自动放置和自动调整大小，它们是 [Dock](#) 和 [Anchor](#)。

拖动顺序可影响自动放置。控件的拖动顺序由父级 [Controls](#) 集合中控件的索引确定。此索引称为 [Z-order](#)。每个控件都按其在集合中的相反顺序进行拖动。这意味着，该集合是先入后出、后入先出的集合。

[MinimumSize](#) 和 [MaximumSize](#) 属性有助于设置控件可以使用的最小和最大空间。

码头

[Dock](#) 属性可设置控件的哪个边与父级的对应边对齐，以及如何在父级中调整控件的大小。

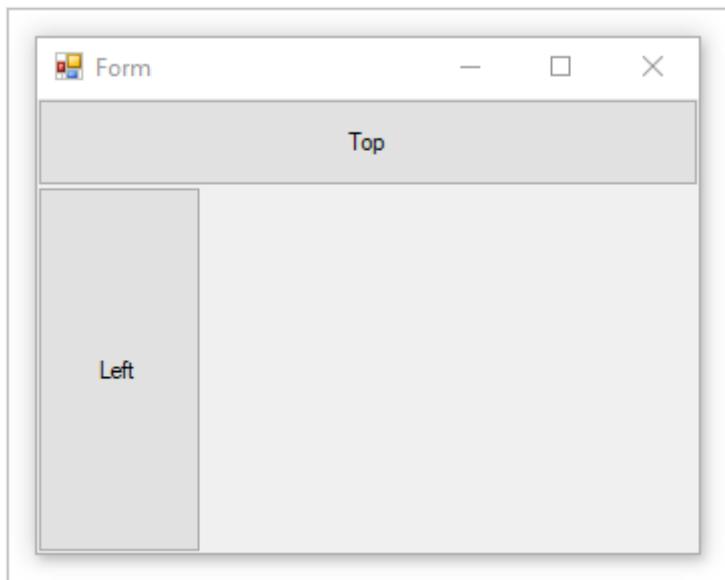


停靠控件时，容器将确定控件应占据的空间，然后调整其大小并放置。基于停靠样式，仍采用控件的宽度和高度。例如，如果将控件停靠在顶部，则采用控件的 [Height](#)，但自

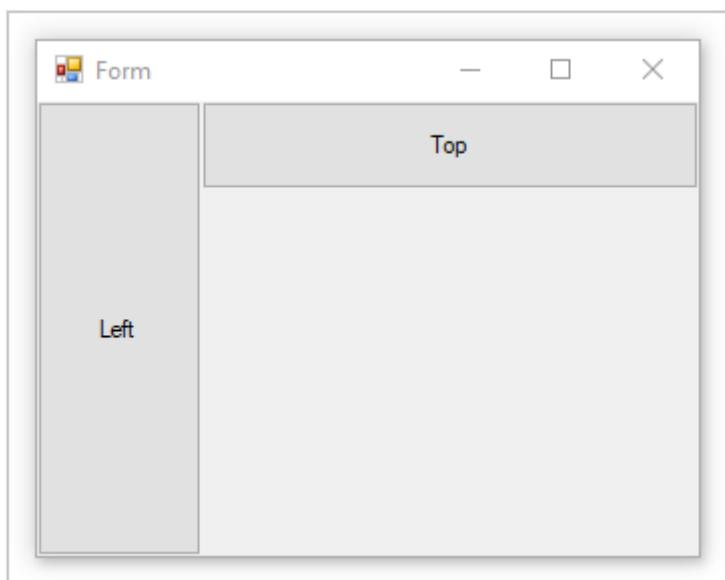
动调整 **Width**。如果将控件停靠在左侧，则采用控件的 **Width**，但自动调整 **Height**。

无法手动设置控件的 **Location**，因为停靠控件将自动控制其位置。

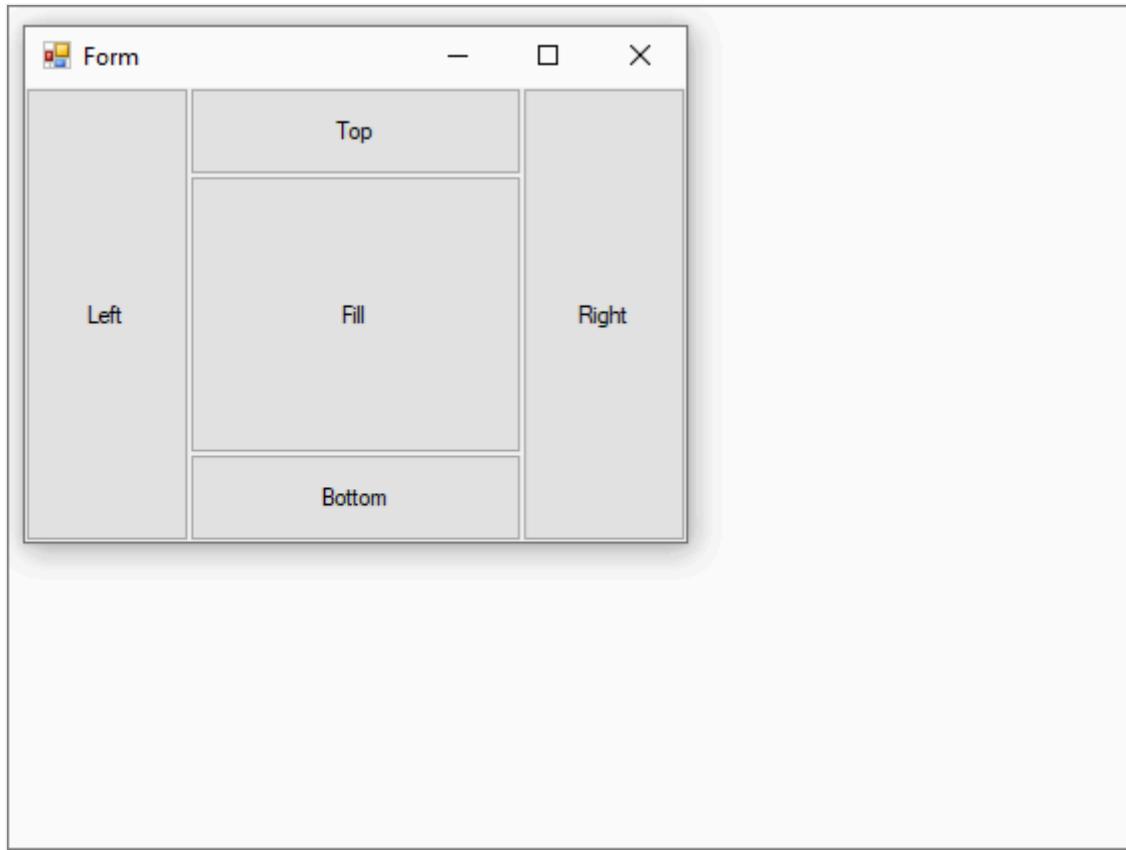
控件的 **Z-order** 会影响停靠。对停靠后的控件进行布局时将占用可用空间。例如，如果控件先出并停靠在顶部，则它将占用容器的整个宽度。如果下一个控件被停靠在左侧，则该控件的可用垂直空间较少。



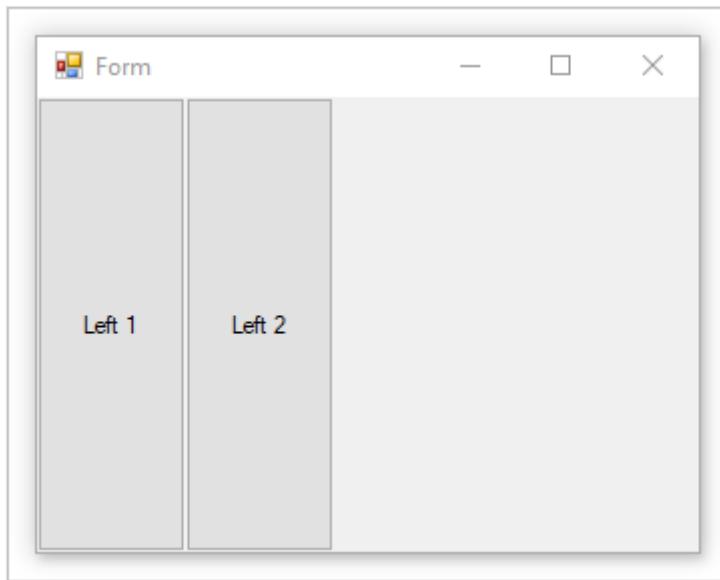
如果控件的 **Z-order** 倒置，停靠在左侧的控件将拥有更多的初始可用空间。该控件使用容器的整个高度。停靠在顶部的控件具有较少的水平空间。



随着容器增大和缩小，停靠在容器上的控件将重新定位并调整大小，以保持其位置和大小处于适当状态。



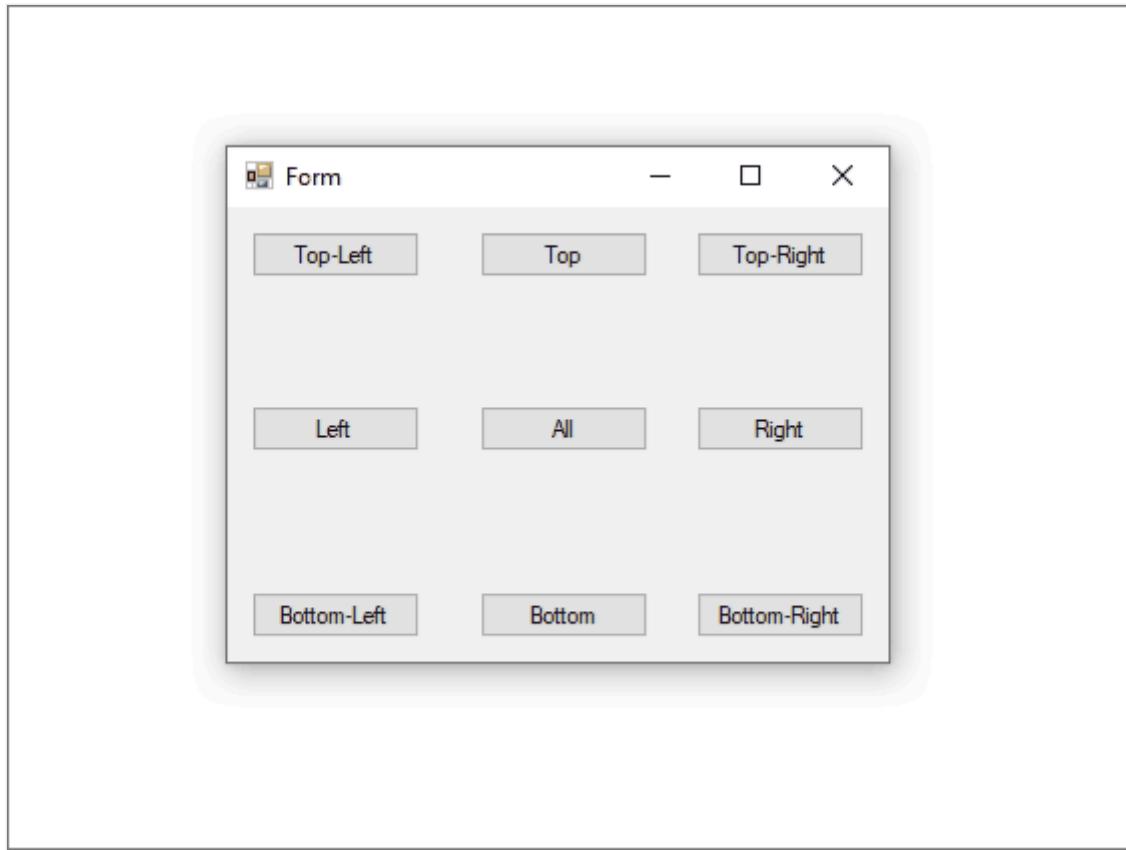
如果多个控件停靠在容器的同一侧，则根据其 Z-order 将其堆叠。



锚

通过定位控件，可将控件绑定到父级容器的一侧或多侧。容器尺寸发生变化时，任何子级控件都将保持其与定位侧的距离。

控件可以定位到一侧或多侧，不存在限制。定位点由 [Anchor](#) 属性设置。



自动调整大小

[AutoSize](#) 属性使控件能够根据需要更改其大小，以适应由 [PreferredSize](#) 属性指定的大小。通过设置 [AutoSizeMode](#) 属性来调整特定控件的大小行为。

只有某些控件支持 [AutoSize](#) 属性。此外，支持 [AutoSize](#) 属性的某些控件还支持 [AutoSizeMode](#) 属性。

[+] 展开表

Always true 行为	描述
自动调整大小是一种运行时功能。	这意味着它永远不会增大或缩小控件，因此不会产生进一步的影响。
如果控件更改大小，则其 Location 属性的值始终保持不变。	控件的内容导致控件增大时，控件将向右和向下扩展。控件不会向左侧扩展。
AutoSize 为 <code>true</code> 时，采用 Dock 和 Anchor 属性。	控件的 Location 属性的值调整为正确的值。 Label 控件是此规则的例外。将停靠的 Label 控件的 AutoSize 属性的值设置为 <code>true</code> 时， Label 控件将不会拉伸。
无论控件 AutoSize 属性的值如何，控件的 MaximumSize 和 MinimumSize 属性始终	MaximumSize 和 MinimumSize 属性不受 AutoSize 属性的影响。

Always true 行为	描述
受到尊重。	
默认情况下没有设置最小大小。	这意味着，如果控件设置为在 <code>AutoSize</code> 下收缩并且没有内容，则其 <code>Size</code> 属性的值为 <code>(0x,0y)</code> 。在这种情况下，控件将缩小为点，且不明显可见。
如果控件未实现 <code>GetPreferredSize</code> 方法， <code>GetPreferredSize</code> 方法将返回分配给 <code>Size</code> 属性的最后一个值。	这意味着将 <code>AutoSize</code> 设置为 <code>true</code> 将不起作用。
<code>TableLayoutPanel</code> 单元格中的控件始终会缩小以适应单元格，直到达到其 <code>MinimumSize</code> 。	此大小被规定为最大值。当单元格是 <code>AutoSize</code> 行或列的一部分时，情况并非如此。

AutoSizeMode 属性

`AutoSizeMode` 属性提供对默认 `AutoSize` 行为的更精细的控制。`AutoSizeMode` 属性指定控件如何根据其内容调整自身大小。例如，内容可以是 `Button` 控件的文本或容器的子控件。

以下列表显示了 `AutoSizeMode` 值及其行为。

- `AutoSizeMode.GrowAndShrink`

控件会增大或收缩以包含其内容。

`MinimumSize` 和 `MaximumSize` 的值会被接受，但 `Size` 属性的当前值会被忽略。

这与具有 `AutoSize` 属性的控件的行为相同，没有 `AutoSizeMode` 属性。

- `AutoSizeMode.GrowOnly`

控件会根据需要增大以包含其内容，但不会缩小到小于其 `Size` 属性指定的值。

这是 `AutoSizeMode` 的默认值。

支持 `AutoSize` 属性的控件

下表描述了各控件对自动调整大小的支持级别：

[+] 展开表

控件	支持 <code>AutoSize</code>	支持 <code>AutoSizeMode</code>
Button	✓	✓
CheckedListBox	✓	✓
FlowLayoutPanel	✓	✓
Form	✓	✓
GroupBox	✓	✓
Panel	✓	✓
TableLayoutPanel	✓	✓
CheckBox	✓	✗
DomainUpDown	✓	✗
Label	✓	✗
LinkLabel	✓	✗
MaskedTextBox	✓	✗
NumericUpDown	✓	✗
RadioButton	✓	✗
TextBox	✓	✗
TrackBar	✓	✗
CheckedListBox	✗	✗
ComboBox	✗	✗
DataGridView	✗	✗
DateTimePicker	✗	✗
ListBox	✗	✗
ListView	✗	✗
MaskedTextBox	✗	✗
MonthCalendar	✗	✗
ProgressBar	✗	✗
PropertyGrid	✗	✗

控件	支持 <code>AutoSize</code>	支持 <code>AutoSizeMode</code>
RichTextBox	✗	✗
SplitContainer	✗	✗
TabControl	✗	✗
TabPage	✗	✗
TreeView	✗	✗
WebBrowser	✗	✗
ScrollBar	✗	✗

设计环境中的 `AutoSize`

下表根据控件 `AutoSize` 和 `AutoSizeMode` 属性的值，在设计时描述控件的大小调整行为。

替代 `SelectionRules` 属性以确定给定控件是否处于用户可调整大小的状态。在下表中，“不能调整大小”表示仅 `Moveable`，“可调整大小”表示 `AllSizeable` 和 `Moveable`。

[+] 展开表

<code>AutoSize</code> 设置	<code>AutoSizeMode</code> 设 置	行为
<code>true</code>	属性不可用。	用户无法在设计时调整控件的大小，以下控件除外： - TextBox - MaskedTextBox - RichTextBox - TrackBar
<code>true</code>	<code>GrowAndShrink</code>	用户无法在设计时调整控件的大小。
<code>true</code>	<code>GrowOnly</code>	用户可以在设计时调整控件的大小。设置 <code>Size</code> 属性时， 用户只能增加控件的大小。
已隐藏 <code>false</code> 或 <code>AutoSize</code>	不适用。	用户可以在设计时调整控件的大小。

① 备注

为了最大限度地提高工作效率，Visual Studio 中的 Windows 窗体设计器会隐藏 `Form` 类的 `AutoSize` 属性。在设计时，窗体的行为如同 `AutoSize` 属性设置为

`false`，而不考虑其实际设置如何。在运行时，不会进行特殊调整，并且会按照属性设置指定的方式应用 `AutoSize` 属性。

容器：窗体

`Form` 是 Windows Forms 的主要对象。Windows 窗体应用程序通常会始终显示窗体。窗体包含控件，并采用控件的 `Location` 和 `Size` 属性来实现手动放置。窗体还响应 `Dock` 属性，以实现自动放置。

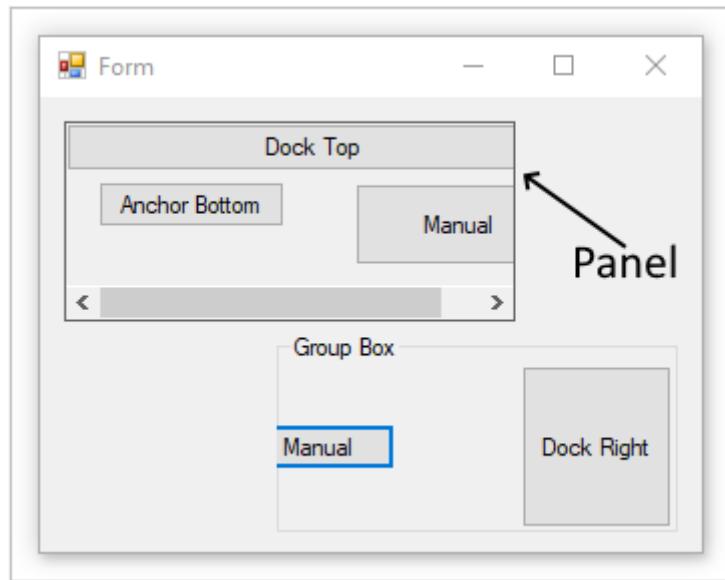
大多数时候，窗体的边缘会有抓手，允许用户调整窗体的大小。控件的 `Anchor` 属性允许控件在调整窗体大小时增大和缩小。

容器：面板

`Panel` 控件类似于表单，因为它仅仅是把控件组合在一起。它支持窗体所支持的手动和自动放置样式。有关详细信息，请参阅[容器：窗体](#)部分。

面板与父级无缝相融，并切断超出面板边界的控件的任何区域。如果控件位于面板边界之外，并且 `AutoScroll` 设置为 `true`，将显示滚动条，用户可以滚动面板。

与[组框](#)控件不同，面板没有标题和边框。



上图具有一个面板，其中设置了 `BorderStyle` 属性，用于演示面板的边界。

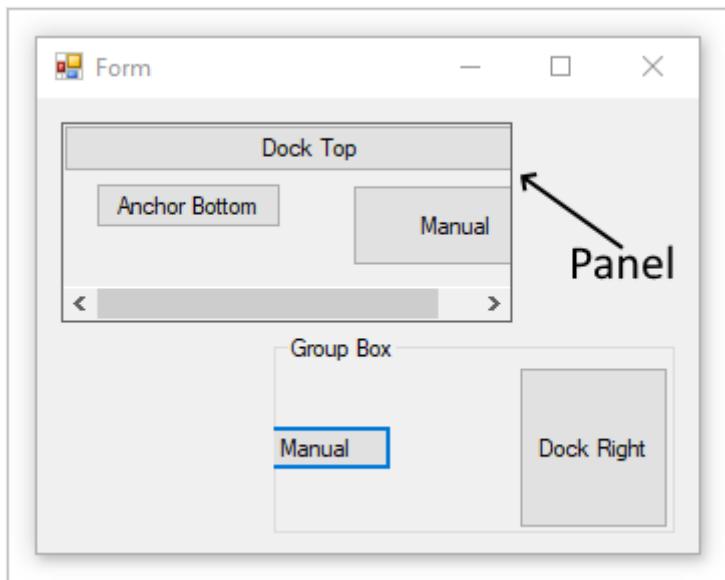
容器：分组框

`GroupBox` 控件为其他控件提供可识别的分组。通常，可使用分组框按功能细分窗体。例如，你可能有一个表示个人信息的信息表单，各个与地址相关的字段将归类在一起。

设计时，可轻松移动分组框及其包含的控件。

分组框支持窗体所支持的手动和自动放置样式。更多信息，请参阅 [容器：表单](#) 节。分组框还会切断控件超出面板边界的任何部分。

与 [面板](#) 控件不同，组框无法滚动内容和显示滚动条。

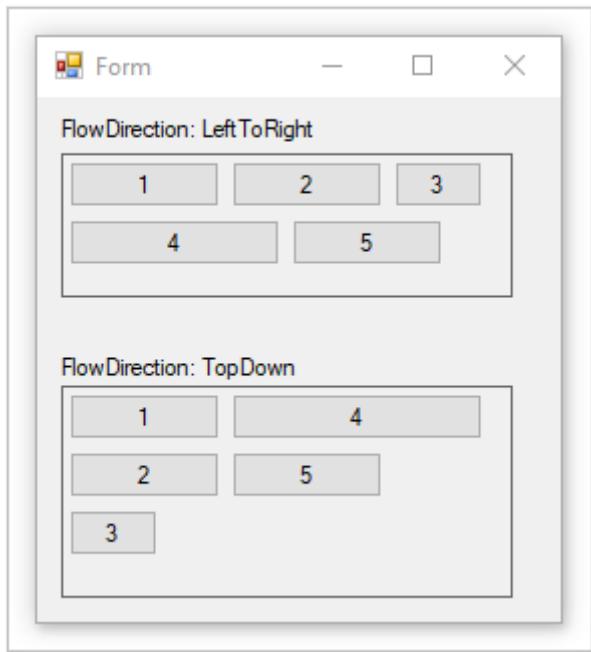


容器：流布局

[FlowLayoutPanel](#) 控件按水平或垂直流方向排列其内容。可以将控件的内容从一行换行到下一行，也可以从一列换行到下一列。或者，可以剪切其内容，而非进行换行。

可以通过设置 [FlowDirection](#) 属性的值来指定流方向。[FlowLayoutPanel](#) 控件在从右到左(RTL)布局中正确地反转其排列方向。也可通过设置 [WrapContents](#) 属性的值来指定是对 [FlowLayoutPanel](#) 控件的内容进行换行还是剪切。

将 [AutoSize](#) 属性设置为 `true` 时，[FlowLayoutPanel](#) 控件会自动调整其内容的大小。它还为其子控件提供 [FlowBreak](#) 属性。将 [FlowBreak](#) 属性的值设置为 `true` 会导致 [FlowLayoutPanel](#) 控件停止在当前流方向中布局控件，并换行到下一行或列。



上图包含两个 `FlowLayoutPanel` 控件，其中 `BorderStyle` 属性设置为演示控件边界。

容器：表格布局

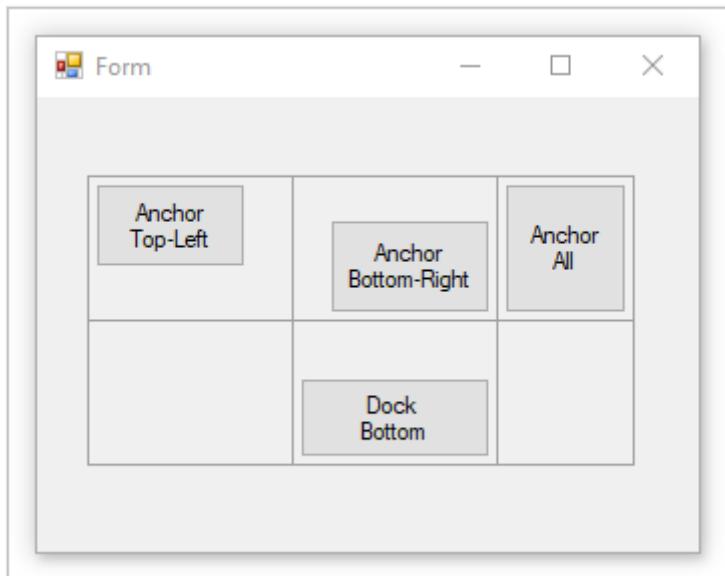
`TableLayoutPanel` 控件在网格中排列其内容。由于布局是在设计时和运行时完成的，因此它可以在应用程序环境发生更改时动态更改。这使面板中的控件能够按比例调整大小，因此它们可以响应变化，比如父控件尺寸的调整或由于本地化导致的文本长度变化。

任何 Windows 窗体控件均可以是 `TableLayoutPanel` 控制的子控件，包括 `TableLayoutPanel` 的其他实例。这样，就可以构建适应运行时更改的复杂布局。

在 `TableLayoutPanel` 控件充满子控件后，还可控制扩展的方向（水平或垂直）。默认情况下，`TableLayoutPanel` 控件通过添加行向下展开。

可以使用 `RowStyles` 和 `ColumnStyles` 属性来控制行和列的大小和样式。可以单独设置行或列的属性。

`TableLayoutPanel` 控件将以下属性添加到其子控件：`Cell`、`Column`、`Row`、`ColumnSpan` 和 `RowSpan`。

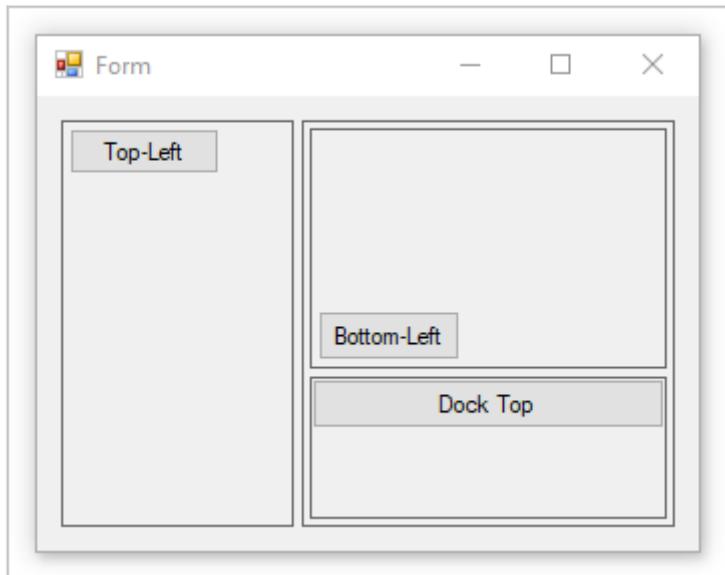


上图包含一个表，其中设置了 [CellBorderStyle](#) 属性来演示每个单元格的边界。

容器：拆分容器

可以将 Windows 窗体 [SplitContainer](#) 控件视为复合控件;它是两个面板，由可移动条分隔。当鼠标指针位于条上时，指针的形状会改变，以指示该条是可移动的。

使用 [SplitContainer](#) 控件，可以创建复杂的用户界面;通常，一个面板中的选择决定了另一个面板中显示的对象。这种排列方式对于显示和浏览信息非常有效。使用两个面板可以聚合区域中的信息，条形图或“拆分器”使用户能够轻松调整面板的大小。

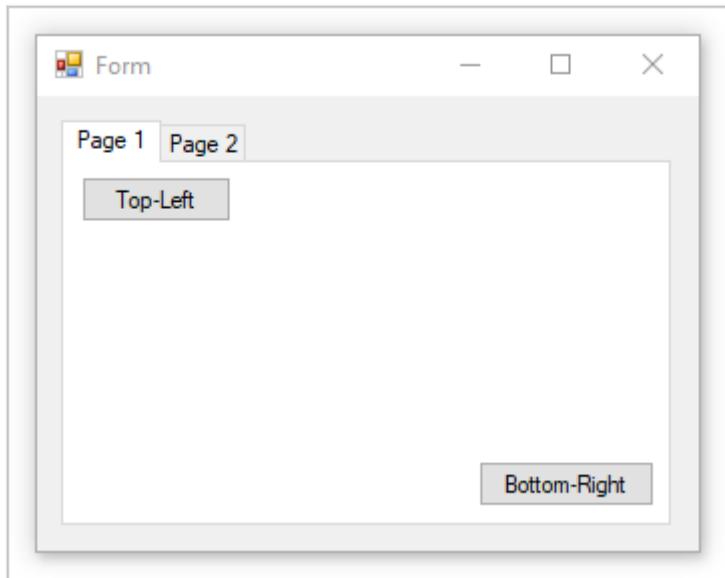


上图的拆分容器用于创建左右窗格。右窗格包含第二个拆分容器，其中 [Orientation](#) 设置为 [Vertical](#)。[BorderStyle](#) 属性设置为演示每个面板的边界。

容器：选项卡控件

[TabControl](#) 显示多个选项卡，类似于笔记本中的分隔符或文件柜中文件夹上的标签。选项卡可以包含图片和其他控件。使用选项卡控件生成在 Windows 操作系统中显示许多位置的多页对话框类型，例如控制面板和显示属性。此外，可以使用 [TabControl](#) 创建属性页，这些页用于设置一组相关属性。

[TabControl](#) 最重要的属性是 [TabPages](#)，其中包含各个选项卡。每个单独的选项卡都是一个 [TabPage](#) 对象。



标签控件概述 (Windows 窗体 .NET)

项目 • 2024/12/18

Windows 窗体 [Label](#) 控件用于显示用户无法编辑的文本。它们用于标识窗体上的对象，并提供特定控件所代表或执行的操作的说明。例如，可以使用标签向文本框、列表框、组合框等添加描述性标题。还可以编写代码来更改标签在运行时响应事件时显示的文本。

使用标签控件

由于 [Label](#) 控件无法接收焦点，因此可用于为其他控件创建访问键。访问键允许用户按 Tab 键顺序聚焦下一个控件，方法是使用所选访问键按 Alt 键。有关详细信息，请参阅[使用标签来聚焦控件](#)。

标签中显示的标题包含在 [Text](#) 属性中。使用 [TextAlign](#) 属性可以设置标签中的文本的对齐方式。有关详细信息，请参阅[如何设置 Windows 窗体控件所显示的文本](#)。

另请参阅

- [使用标签聚焦控件 \(Windows 窗体 .NET\)](#)
- [如何设置控件的显示文本 \(Windows 窗体.NET\)](#)
- [AutoSizeMode](#)
- [Scale](#)
- [PerformAutoScale](#)
- [AutoScaleDimensions](#)

控制事件 (Windows 窗体 .NET)

项目 · 2024/12/18

控件提供当用户与控件交互或控件状态更改时引发的事件。本文介绍大多数控件共享的常见事件、由用户交互引发的事件，以及特定于特定控件的事件。有关 Windows 窗体中事件的详细信息，请参阅[事件概述](#) 和[处理和引发事件](#)。

有关如何添加或删除控件事件处理程序的详细信息，请参阅[如何处理事件](#)。

常见事件

控件通过基类 [Control](#) 提供 60 多个事件。其中包括 [Paint](#) 事件，导致绘制控件、与显示窗口相关的事件，例如 [Resize](#) 和 [Layout](#) 事件，以及低级别鼠标和键盘事件。某些低级别事件通过 [Control](#) 合成为语义事件，例如 [Click](#) 和 [DoubleClick](#)。大多数共享事件属于以下类别：

- 鼠标事件
- 键盘事件
- 属性更改事件
- 其他事件

鼠标事件

考虑到 Windows 窗体是用户界面 (UI) 技术，鼠标输入是用户与 Windows 窗体应用程序交互的主要方式。所有控件都提供与鼠标相关的基本事件：

- [MouseClick](#)
- [MouseDoubleClick](#)
- [MouseDown](#)
- [MouseEnter](#)
- [MouseHover](#)
- [MouseLeave](#)
- [MouseMove](#)
- [MouseUp](#)
- [MouseWheel](#)
- [Click](#)

如需更多信息，请参阅[使用鼠标事件](#)。

键盘事件

如果控件响应用户输入（如 `TextBox` 或 `Button` 控件），则会为控件引发相应的输入事件。为了接收键盘事件，控件必须处于聚焦状态。某些控件（如 `Label` 控件）无法聚焦且无法接收键盘事件。下面是键盘事件的列表：

- `KeyDown`
- `KeyPress`
- `KeyUp`

有关详细信息，请参阅 [使用键盘事件](#)。

属性更改事件

Windows 窗体遵循具有变更事件的属性的 `PropertyNameChanged` 模式。Windows 窗体提供的数据绑定引擎可识别此模式并与之很好地集成。创建自己的控件时，实现此模式。

此模式使用属性 `FirstName` 实现以下规则，例如：

- 将属性命名为：`FirstName`。
- 使用模式 `PropertyNameChanged : FirstNameChanged` 为属性创建事件。
- 使用模式 `OnPropertyNameChanged : OnFirstNameChanged` 来创建私有或受保护的方法。

如果 `FirstName` 属性集修改了支持值，则会调用 `OnFirstNameChanged` 方法。

`OnFirstNameChanged` 方法引发 `FirstNameChanged` 事件。

下面是控件的一些常见属性更改事件：

 展开表

事件	描述
<code>BackColorChanged</code>	当 <code>BackColor</code> 属性的值更改时发生。
<code>BackgroundImageChanged</code>	当 <code>BackgroundImage</code> 属性的值更改时发生。
<code>BindingContextChanged</code>	当 <code>BindingContext</code> 属性的值更改时发生。
<code>DockChanged</code>	当 <code>Dock</code> 属性的值更改时发生。
<code>EnabledChanged</code>	当 <code>Enabled</code> 属性值发生更改时发生。
<code>FontChanged</code>	当 <code>Font</code> 属性值更改时发生。
<code>ForeColorChanged</code>	当 <code>ForeColor</code> 属性值更改时发生。

事件	描述
LocationChanged	当 Location 属性值发生更改时发生。
SizeChanged	当 Size 属性值更改时发生。
VisibleChanged	当 Visible 属性值更改时发生。

有关事件的完整列表，请参阅 [控件类的 事件](#) 部分。

其他事件

控件还会根据控件的状态或其他与控件的交互引发事件。例如，如果控件具有焦点，并且用户按下 [F1](#) 键，则会引发 [HelpRequested](#) 事件。如果用户按下窗体上的上下文敏感帮助按钮，然后按控件上的帮助光标，也会引发此事件。

另一个示例是，当控件被更改、移动或调整大小时，将触发 [Paint](#) 事件。此事件为开发人员提供了绘制控件和更改其外观的机会。

有关事件的完整列表，请参阅 [控件类的 事件](#) 部分。

另请参阅

- [如何处理事件](#)
- [事件概述](#)
- [使用鼠标事件](#)
- [使用键盘事件](#)
- [System.Windows.Forms.Control](#)
- [System.Windows.Forms.Control.Click](#)
- [System.Windows.Forms.Button](#)

自定义控件 (Windows Forms .NET)

项目 • 2024/12/18

使用 Windows 窗体，可以通过继承创建新控件或修改现有控件。本文重点介绍了创建新控件的方式之间的差异，并提供有关如何为项目选择特定类型的控件的信息。

基础控件类

[Control](#) 类是 Windows 窗体控件的基类。它提供 Windows 窗体应用程序中视觉显示所需的基础结构，并提供以下功能：

- 暴露窗口句柄。
- 管理消息路由。
- 提供鼠标和键盘事件，以及许多其他用户界面事件。
- 提供高级布局功能。
- 包含特定于视觉显示的许多属性，例如 [ForeColor](#)、[BackColor](#)、[Height](#) 和 [Width](#)。

由于大部分基础结构由基类提供，因此开发自己的 Windows 窗体控件相对容易。

创建自己的控件

可以创建三种类型的自定义控件：用户控件、扩展控件和自定义控件。下表可帮助你确定应创建的控件类型：

[+] 展开表

如果...	创建 ...
<ul style="list-style-type: none">你想要将多个 Windows 窗体控件的功能合并到单个可重用单元中。	通过继承 System.Windows.Forms.UserControl ，设计 用户控件 。
<ul style="list-style-type: none">你所需的大多数功能已经与现有的 Windows 窗体控件相同。您不需要自定义图形用户界面，或者您想要为现有控件设计新的图形用户界面。	通过从特定的 Windows 窗体控件继承来扩展控件。
<ul style="list-style-type: none">你希望为控件提供自定义的图形表示形式。你需要实现无法通过标准控件提供的自定义功能。	通过继承 System.Windows.Forms.Control ，创建 自定义控件 。

用户控件

用户控件是作为单个控件提供给使用者的 Windows 窗体控件的集合。此类控件称为 **复合控件**。包含的控件称为 **组件控件**。

用户控件包含与每个包含的 Windows 窗体控件关联的所有固有功能，使你可以有选择地公开和绑定其属性。用户控件还无需任何额外开发努力即可提供大量默认键盘处理功能。

例如，可以生成用户控件以显示数据库中的客户地址数据。此控件将包括用于显示数据库字段的 [DataGridView](#) 控件、用于处理绑定到数据源的 [BindingSource](#)，以及用于在记录中移动的 [BindingNavigator](#) 控件。你可以有选择地公开数据绑定属性，并且可以打包并重复使用整个控件，从应用程序到应用程序。

有关详细信息，请参阅 [用户控件概述](#)。

扩展控件

您可以从任何现有的 Windows 窗体控件派生一个继承控件。使用此方法，你可以保留 Windows 窗体控件的所有固有功能，然后通过添加自定义属性、方法或其他功能来扩展该功能。使用此选项，您可以覆盖基本控件的绘图逻辑，然后通过更改其外观来扩展用户界面。

例如，可以创建派生自 [Button](#) 控件的控件，该控件跟踪用户单击的次数。

在某些控件中，还可以通过重写基类的 [OnPaint](#) 方法，向控件的图形用户界面添加自定义外观。对于跟踪单击的扩展按钮，可以重写 [OnPaint](#) 方法以调用 [OnPaint](#) 的基本实现，然后在 [Button](#) 控件工作区的一角绘制单击计数。

自定义控件

创建控件的另一种方法是通过从 [Control](#) 继承，从头开始创建一个控件。[Control](#) 类提供控件所需的所有基本功能，包括鼠标和键盘处理事件，但没有特定于控件的功能或图形界面。

通过继承自 [Control](#) 类来创建控件，比继承自 [UserControl](#) 或现有的 Windows 窗体控件需要更多的思考和努力。由于为你留下了大量实现，因此你的控件可以比复合控件或扩展控件具有更大的灵活性，并且你可以定制控件以满足你的确切需求。

若要实现自定义控件，必须为控件的 [OnPaint](#) 事件编写代码，该事件控制控件的直观绘制方式。您还必须为控件编写功能特定的行为。还可以替代 [WndProc](#) 方法并直接处理

Windows 消息。这是创建控件的最强大方法，但要有效地使用此技术，需要熟悉 Microsoft Win32® API。

自定义控件的一个示例是一个时钟控件，它复制模拟时钟的外观和行为。为了响应来自内部 [Timer](#) 组件的 [Tick](#) 事件，调用自定义绘制以使时钟的指针移动。

自定义设计体验

如果需要实现自定义设计时体验，可以编写自己的设计器。对于复合控件，请从 [ParentControlDesigner](#) 或 [DocumentDesigner](#) 类派生自定义设计器类。对于扩展控件和自定义控件，请从 [ControlDesigner](#) 类派生自定义设计器类。

使用 [DesignerAttribute](#) 将您的控件与设计器相关联。

以下信息已过期，但可能有助于你。

- [\(Visual Studio 2013\) 扩展 Design-Time 支持。](#)
- [\(Visual Studio 2013\) 如何创建一个利用 Design-Time 功能的 Windows 窗体控件。](#)

在控件上绘图 (Windows 窗体 .NET)

项目 • 2024/11/05

控件的自定义绘制是 Windows 窗体可以轻松完成的众多复杂任务之一。创作自定义控件时，有许多选项可用于处理控件的图形外观。如果要创作[自定义控件](#)（即从 [Control](#) 继承的控件），则必须提供代码以呈现其图形表示形式。

如果要创建[复合控件](#)（即从 [UserControl](#) 继承的控件或某个[现有的 Windows 窗体控件](#)），则可以替代标准图形表示形式，并提供你自己的图形代码。

如果要在不创建新控件的情况下为现有控件提供自定义呈现，选项会变得更为有限。但是，对于控件和应用程序，仍有各种各样的图形。

控件呈现涉及以下元素：

- 基类 [System.Windows.Forms.Control](#) 提供的绘图功能。
- GDI 图形库的基本元素。
- 绘图区域的几何图形。
- 释放图形资源的过程。

控件提供的绘图

基类 [Control](#) 通过其 [Paint](#) 事件提供绘图功能。每当控件需要更新其显示时，它都会引发 [Paint](#) 事件。有关 .NET 中的事件的详细信息，请参阅[处理和引发事件](#)。

[Paint](#) 事件的事件数据类 [PaintEventArgs](#) 包含绘制控件所需的数据 - 图形对象的句柄和表示绘制区域的矩形。

C#

```
public class PaintEventArgs : EventArgs, IDisposable
{
    public System.Drawing.Rectangle ClipRectangle { get; }
    public System.Drawing.Graphics Graphics { get; }

    // Other properties and methods.
}
```

[Graphics](#) 是一个可封装绘图功能的托管类，如本文后面的 GDI 讨论中所述。

[ClipRectangle](#) 是 [Rectangle](#) 结构的实例，它定义了可在其中绘制控件的可用区域。控件开发人员可以使用控件的 [ClipRectangle](#) 属性来计算 [ClipRectangle](#)，如本文后面的几何图形讨论中所述。

OnPaint

控件必须通过重写它从 [Control](#) 继承的 [OnPaint](#) 方法来提供呈现逻辑。[OnPaint](#) 可访问图形对象和矩形，以通过传递给它的 [PaintEventArgs](#) 实例的 [Graphics](#) 和 [ClipRectangle](#) 属性进行绘制。

下面的代码使用 [System.Drawing](#) 命名空间：

C#

```
protected override void OnPaint(PaintEventArgs e)
{
    // Call the OnPaint method of the base class.
    base.OnPaint(e);

    // Declare and instantiate a new pen that will be disposed of at the end
    // of the method.
    using var myPen = new Pen(Color.Aqua);

    // Create a rectangle that represents the size of the control, minus 1
    // pixel.
    var area = new Rectangle(new Point(0, 0), new Size(this.Size.Width - 1,
    this.Size.Height - 1));

    // Draw an aqua rectangle in the rectangle represented by the control.
    e.Graphics.DrawRectangle(myPen, area);
}
```

[Control](#) 基类的 [OnPaint](#) 方法不实现任何绘图功能，只是调用注册到 [Paint](#) 事件的事件委托。重写 [OnPaint](#) 时，应确保调用基类的 [OnPaint](#) 方法，以便注册的委托可接收 [Paint](#) 事件。但是，绘制整个表面的控件不应调用基类的 [OnPaint](#)，因为这会引起闪烁。

① 备注

请勿直接从控件调用 [OnPaint](#)；请改为调用 [Invalidate](#) 方法（从 [Control](#) 继承）或调用 [Invalidate](#) 的其他方法。[Invalidate](#) 方法又会调用 [OnPaint](#)。重载 [Invalidate](#) 方法，并根据提供给 [Invalidate](#) `e` 的参数重绘其部分屏幕区域或整个屏幕区域。

控件的 [OnPaint](#) 方法中的代码将在第一次绘制控件以及每次刷新该控件时执行。若要确保在每次调整控件大小时都重新进行绘制，请将下面的行添加到控件的构造函数中：

C#

```
SetStyle(ControlStyles.ResizeRedraw, true);
```

OnPaintBackground

[Control](#) 基类定义了另一种可用于绘图的方法，即 [OnPaintBackground](#) 方法。

C#

```
protected virtual void OnPaintBackground(PaintEventArgs e);
```

[OnPaintBackground](#) 绘制窗口的背景（并以相同方式绘制形状），并保证速度较快，而 [OnPaint](#) 绘制详细信息，速度可能较慢，因为单个绘制请求会合并为一个 [Paint](#) 事件，其中涵盖了所有需要重新绘制的区域。例如，如果想要为控件绘制颜色渐变的背景，则可能需要调用 [OnPaintBackground](#)。

虽然 [OnPaintBackground](#) 具有类似事件的命名法并采用与 [OnPaint](#) 方法相同的参数，但 [OnPaintBackground](#) 不是真正的事件方法。不存在 [PaintBackground](#) 事件，并且 [OnPaintBackground](#) 不调用事件委托。重写 [OnPaintBackground](#) 方法时，无需使用派生类即可调用其基类的 [OnPaintBackground](#) 方法。

GDI+ 基础知识

[Graphics](#) 类提供用于绘制各种形状（如圆形、三角形、弧形和椭圆形）的方法，以及用于显示文本的方法。[System.Drawing](#) 命名空间包含命名空间和类，它们可用于封装图形元素，如形状（圆形、矩形、弧形等）、颜色、字体、画笔等。

绘图区域的几何图形

控件的 [ClientRectangle](#) 属性指定可用于用户屏幕上的控件的矩形区域，而 [PaintEventArgs](#) 的 [ClipRectangle](#) 属性指定所绘制的区域。当控件的一小部分显示发生变化时，控件可能只需要绘制部分可用区域。在这些情况下，控件开发人员必须计算要在其中进行绘制的实际矩形，并将其传递到 [Invalidate](#)。采用 [Rectangle](#) 或 [Region](#) 作为参数的 [Invalidate](#) 的重载版本使用该参数生成 [PaintEventArgs](#) 的 [ClipRectangle](#) 属性。

释放图形资源

图形对象成本昂贵，因为它们使用系统资源。此类对象包括 [System.Drawing.Graphics](#) 类的实例以及 [System.Drawing.Brush](#)、[System.Drawing.Pen](#) 和其他图形类的实例。请务必仅在需要时才创建图形资源，并在使用完之后立即释放。如果创建实现 [IDisposable](#) 接口的类型的实例，请在使用完该实例之后，调用其 [Dispose](#) 方法来释放资源。

另请参阅

- [自定义控件的类型](#)

为控件提供辅助功能信息（Windows 窗体 .NET）

项目 · 2024/12/18

无障碍辅助工具是专门的程序和设备，可帮助残障人士更加有效地使用计算机。例如，盲人使用的屏幕阅读器，以及提供口头命令而不是使用鼠标或键盘的人的语音输入实用工具。这些无障碍辅助工具与 Windows 窗体控件所公开的辅助功能属性进行交互。这些属性包括：

- System.Windows.Forms.AccessibleObject
- System.Windows.Forms.Control.AccessibleDefaultActionDescription
- System.Windows.Forms.Control.AccessibleDescription
- System.Windows.Forms.Control.AccessibleName
- System.Windows.Forms.AccessibleRole

AccessibilityObject 属性

此只读属性包含 `AccessibleObject` 实例。`AccessibleObject` 实现 `IAccessible` 接口，该接口提供有关控件的说明、屏幕位置、导航功能和值的信息。当控件添加到窗体时，设计器将设置此值。

AccessibleDefaultActionDescription 属性

此字符串描述控件的操作。它不会出现在“属性”窗口中，只能在代码中设置。以下示例设置按钮控件的 `AccessibleDefaultActionDescription` 属性：

C#

```
button1.AccessibleDefaultActionDescription = "Closes the application.;"
```

AccessibleDescription 属性

此字符串描述控件。`AccessibleDescription` 属性可以在“属性”窗口中设置，也可以在代码中设置，如下所示：

C#

```
button1.AccessibleDescription = "A button with text 'Exit'" ;
```

AccessibleName 属性

向辅助功能工具报告的控件名称。 [AccessibleName](#) 属性可以在“属性”窗口中设置，也可以在代码中设置，如下所示：

C#

```
button1.AccessibleName = "Order";
```

AccessibleRole 属性

此属性包含 [AccessibleRole](#) 枚举，描述控件的用户界面角色。 新控件的值设置为 `Default`。 这意味着，默认情况下，`Button` 控件充当 `Button`。 如果控件具有另一个角色，则可能需要重置此属性。 例如，你可能使用 `PictureBox` 控件作为 `Chart`，并且可能希望辅助功能将角色报告为 `Chart`，而不是 `PictureBox`。 你可能还希望为已开发的自定义控件指定此属性。 可以在“属性”窗口中或代码中设置此属性，如下所示：

C#

```
pictureBox1.AccessibleRole = AccessibleRole.Chart;
```

另请参阅

- [标签控件概述 \(Windows 窗体 .NET\)](#)
- [AccessibleObject](#)
- [Control.AccessibleObject](#)
- [Control.AccessibleDefaultActionDescription](#)
- [Control.AccessibleDescription](#)
- [Control.AccessibleName](#)
- [Control.AccessibleRole](#)
- [AccessibleRole](#)

向窗体添加控件（Windows 窗体 .NET）

项目 · 2025/01/30

大多数窗体都是通过将控件添加到窗体图面来定义用户界面（UI）设计的。 控件是窗体上的组件，用于显示信息或接受用户输入。

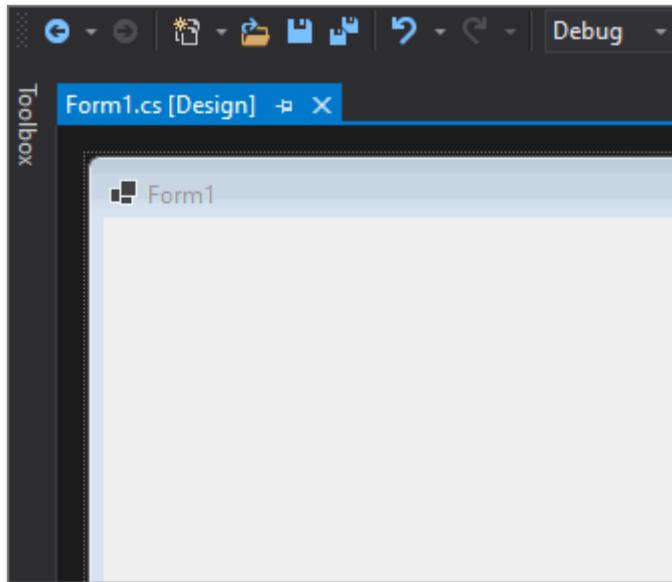
控件添加到窗体的主要方式是通过 Visual Studio Designer，但你也可以通过代码在运行时管理窗体上的控件。

使用设计器添加

Visual Studio 使用窗体设计器设计窗体。有一个“控件”窗格，其中列出了应用可用的所有控件。可以通过两种方式从窗格中添加控件：

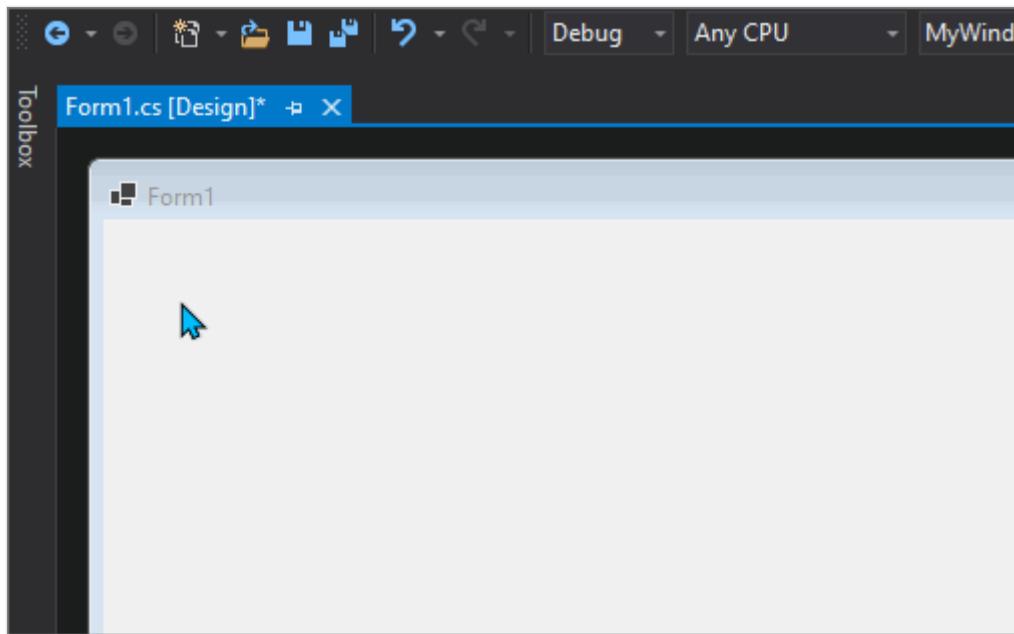
双击添加控件

双击控件时，会自动将其以默认设置添加到当前打开的窗体中。



通过绘图添加控件

通过单击选择控件。 在表单中拖动选择一个区域。 控件将被放置以适应您所选区域的大小。



的工具箱中拖动

选择并绘制控件

通过代码添加

可以创建控件，然后在运行时使用窗体的 [Controls](#) 集合将其添加到窗体。此集合还可用于从窗体中删除控件。

以下代码添加并放置两个控件：[Label](#) 和 [TextBox](#)：

```
C#  
  
Label label1 = new Label()  
{  
    Text = "&First Name",  
    Location = new Point(10, 10),  
    TabIndex = 10  
};  
  
TextBox field1 = new TextBox()  
{  
    Location = new Point(label1.Location.X, label1.Bounds.Bottom +  
    Padding.Top),  
    TabIndex = 11  
};  
  
Controls.Add(label1);  
Controls.Add(field1);
```

另请参阅

- [设置 Windows 窗体控件显示的文本](#)
- [向控件添加访问键快捷方式](#)

- System.Windows.Forms.Label
- System.Windows.Forms.TextBox
- System.Windows.Forms.Button

向控件添加访问键快捷方式 (Windows 窗体 .NET)

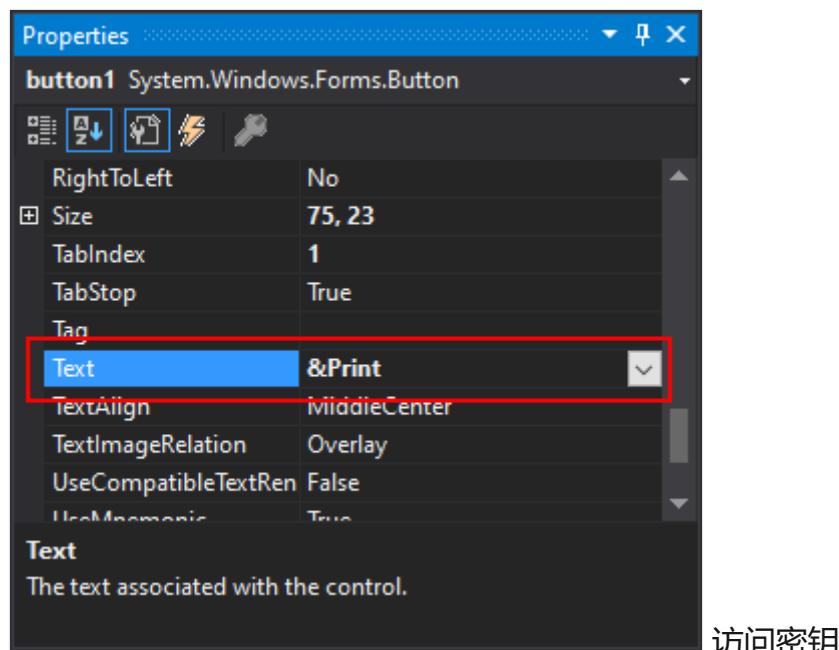
项目 • 2025/01/30

访问键是菜单、菜单项或控件标签（如按钮）文本中的带下划线字符。使用访问键，用户可以“单击”按钮，方法是将 `Alt` 键与预定义的访问键结合使用。例如，如果某个按钮运行打印窗体的过程，因此其 `Text` 属性设置为“Print”，则在字母“P”之前添加 & 号将导致在运行时为按钮文本中的字母“P”添加下划线。用户可以通过按 `Alt` 来运行与按钮关联的命令。

无法接收焦点的控件不能具有访问键，但标签控件除外。

设计器

在 Visual Studio 的 **属性** 窗口中，将 `Text` 属性设置为一个包含与号 (&) 的字符串，并将与号放在将作为访问键的字母之前。例如，若要将字母“P”设置为访问键，请输入 **&打印**。



编程

将 `Text` 属性设置为一个字符串，该字符串包含将作为快捷方式的字母前的与号 (&)。

C#

```
// Set the letter "P" as an access key.
```

```
button1.Text = "&Print";
```

使用标签来聚焦控件

尽管标签无法聚焦，但它能够在窗体的 Tab 键顺序中聚焦下一个控件。每个控件都向 `TabIndex` 属性分配一个值，通常按升序排列。为 `Label.Text` 属性分配访问键时，将聚焦于按 Tab 键顺序排列的下一个控件。

使用 [编程](#) 部分中的示例，如果按钮没有设置任何文本，而是显示打印机的图像，则可以使用标签来聚焦按钮。

C#

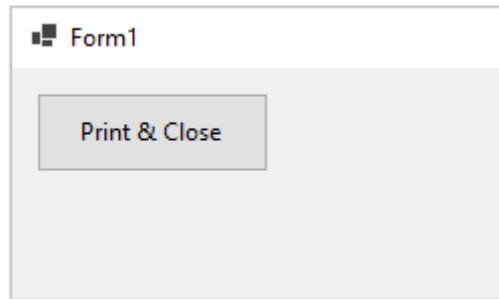
```
// Set the letter "P" as an access key.  
label1.Text = "&Print";  
label1.TabIndex = 9  
button1.TabIndex = 10
```

显示 & 符号

设置将 & 符号解释为访问键的控件的文本或描述文字时，请使用两个连续的 & 符号 (&&) 来显示单个 & 符号。例如，"Print && Close" 按钮上的文本显示在 Print & Close 的标题中：

C#

```
// Set the letter "P" as an access key.  
button1.Text = "Print && Close";
```



中显示和符号

另请参阅

- [设置 Windows 窗体控件显示的文本](#)
- [System.Windows.Forms.Button](#)
- [System.Windows.Forms.Label](#)

如何：设置控件显示的文本（Windows 窗体 .NET）

项目 • 2025/01/30

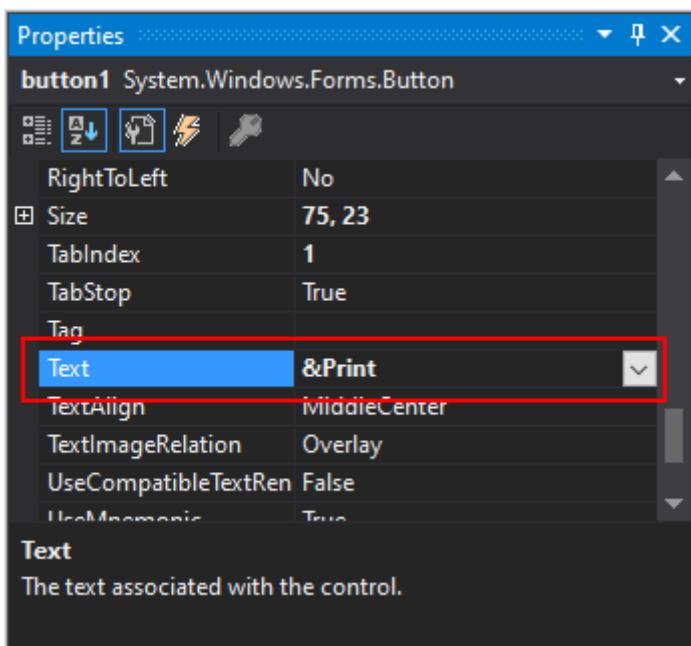
Windows 窗体控件通常显示一些与控件的主要功能相关的文本。例如，[Button](#) 控件通常显示一个标题，指示单击按钮时将执行什么操作。对于所有控件，可以使用 [Text](#) 属性设置或返回文本。可以使用 [Font](#) 属性更改字体。

还可使用[设计器](#)来设置文本。

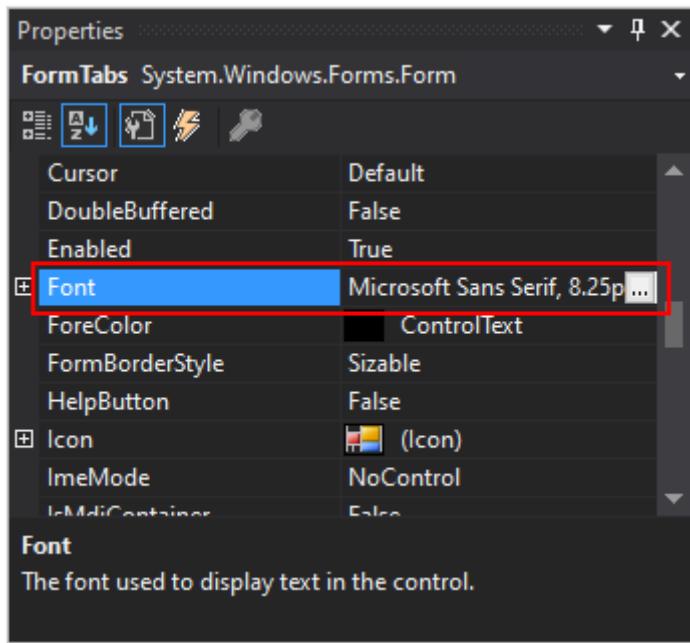
设计器

- 在 Visual Studio 的 [属性](#) 窗口中，将控件 [Text](#) 属性设置为适当的字符串。

若要创建带下划线的快捷键，请在将作为快捷键的字母之前包括一个与号 (&)。

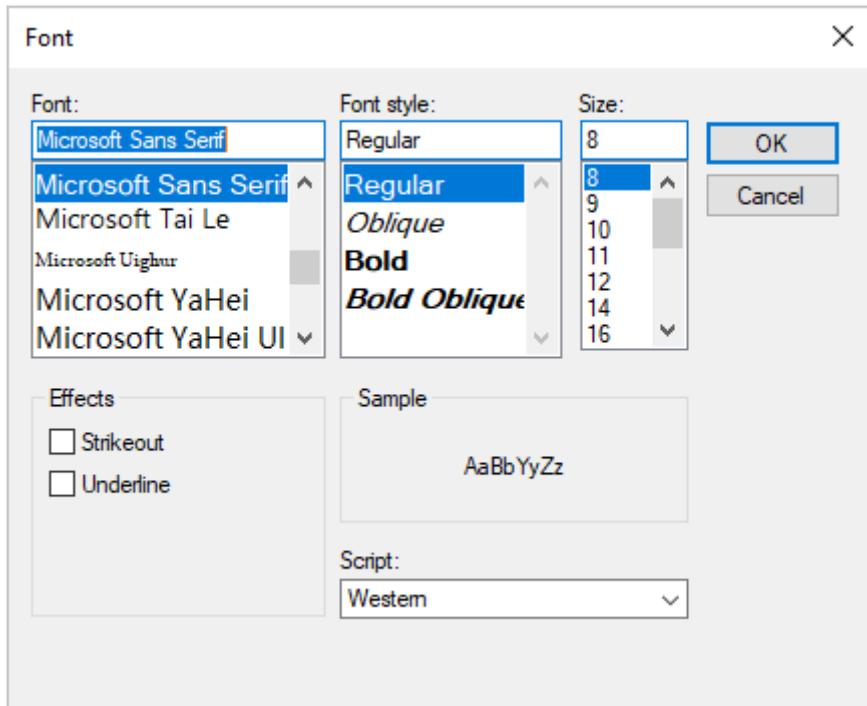


- 在 [属性](#) 窗口中，选择 [字体](#) 属性旁边的省略号按钮 (…)。



在标准字体对话框中，使用类型、大小和样式等设置调整字体。

为 .NET Windows 窗体的 Visual Studio 属性窗格提供字体设置窗口。



编程

- 将 [Text](#) 属性设置为字符串。

若要创建带下划线的访问密钥，请在将作为访问密钥的字母之前包括一个和号 (&)。

- 将 [Font](#) 属性设置为 [Font](#)类型的对象。

C#

```
button1.Text = "Click here to save changes";
button1.Font = new Font("Arial", 10, FontStyle.Bold,
GraphicsUnit.Point);
```

① 备注

可使用转义符来显示用户界面元素中的特殊字符，通常对这些元素（如菜单项）有着不同的解释。例如，下面的代码行将菜单项的文本设置为“& 现读取完全不同的内容”：

C#

```
mpMenuItem.Text = "&& Now For Something Completely Different";
```

另请参阅

- [为 Windows 窗体控件创建访问密钥](#)
- [System.Windows.Forms.Control.Text](#)

如何在 Windows 窗体上设置 Tab 顺序 (Windows 窗体 .NET)

项目 • 2025/01/30

Tab 顺序是指用户通过按 `Tab` 键，将焦点从一个控件移动到另一个控件的顺序。每个表单都有其独立的标签导航顺序。默认情况下，Tab 顺序与创建控件的顺序相同。`Tab` 键顺序编号从零开始，按值递增，并使用 `TabIndex` 属性进行设置。

还可使用[设计器](#)来设置 Tab 键顺序。

可使用 `TabIndex` 属性在设计器的属性窗口中设置 Tab 键顺序。控件的 `TabIndex` 属性确定它在 Tab 顺序中的位置。默认情况下，添加到设计器的第一个控件的 `TabIndex` 值为 0，第二个控件的 `TabIndex` 为 1，依此而已。聚焦最高 `TabIndex` 值之后，按 `Tab` 将循环并聚焦具有最低 `TabIndex` 值的控件。

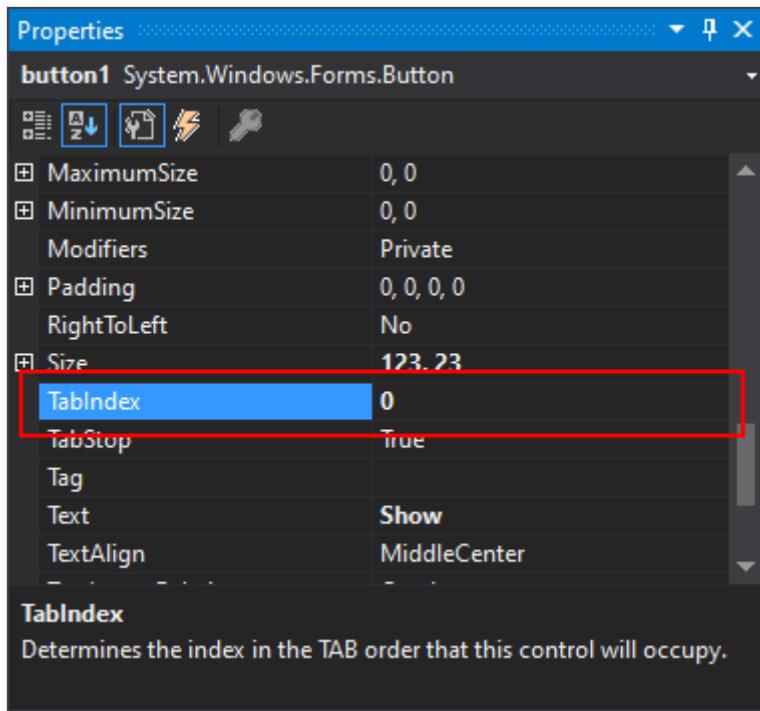
容器控件（如 `GroupBox` 控件）将子控件视为独立于窗体的其余部分。容器中的每个子级都有自己的 `TabIndex` 值。由于容器控件无法聚焦，当 Tab 键顺序到达容器控件时，将聚焦具有最低 `TabIndex` 值的容器的子控件。按下 `Tab` 时，每个子控件都将按其 `TabIndex` 值聚焦，直到最后一个控件。在最后一个控件上按下 `Tab` 时，焦点会根据下一个 `TabIndex` 值恢复到容器父级中的下一个控件。

可以按 Tab 键顺序跳过窗体上任一控件。通常，在运行时连续按 `Tab` 会按 Tab 键顺序选择每个控件。通过关闭 `TabStop` 属性，控件将按窗体的 Tab 键顺序传递。

设计师

使用 Visual Studio 设计器 **属性** 窗口设置控件的 Tab 键顺序。

1. 在设计器中选择控件。
2. 在 Visual Studio 的 **属性** 窗口中，将控件的 `TabIndex` 属性设置为适当的数字。



编程

- 将 `TabIndex` 属性设置为数值。

C#

```
Button1.TabIndex = 1;
```

从 Tab 键顺序中删除控件

通过将 `TabStop` 属性设置为 `false`，可以阻止控件在按下 `Tab` 键时接收焦点。使用 `Tab` 键循环浏览控件时，将跳过该控件。当此属性设置为 `false` 时，控件不会丢失其 Tab 键顺序。

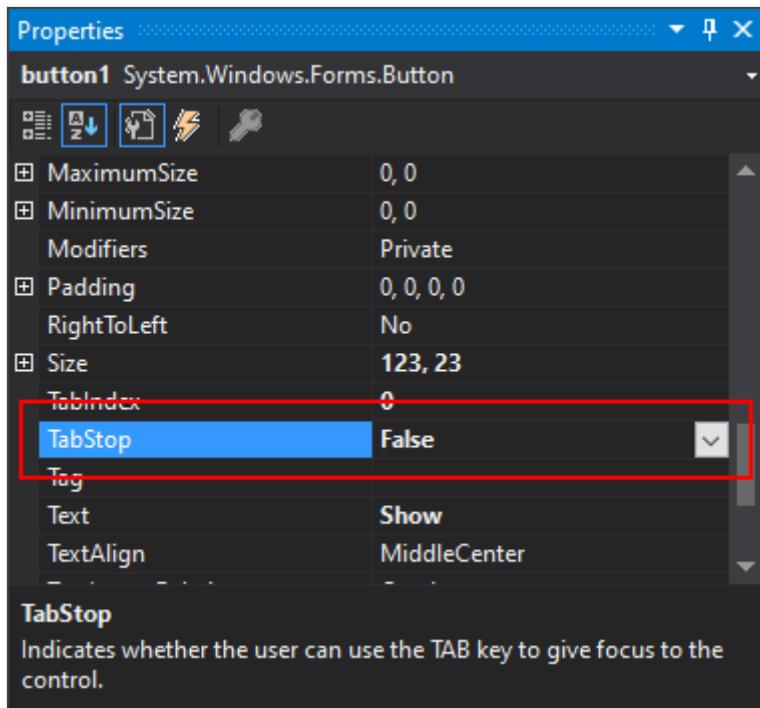
① 备注

单选按钮组在运行时具有单个制表位。所选按钮（其 `Checked` 属性设置为 `true` 的按钮）会自动将其 `TabStop` 属性设置为 `true`。单选按钮组中的其他按钮会将其 `TabStop` 属性设置为 `false`。

使用设计器设置 TabStop

- 在设计器中选择控件。

2. 在 Visual Studio 的 属性 窗口中，将 TabStop 属性设置为 False。



以编程方式设置 TabStop

1. 将 TabStop 属性设置为 false。

```
C#  
Button1.TabStop = false;
```

另请参阅

- 向窗体添加控件
- System.Windows.Forms.Control.TabIndex
- System.Windows.Forms.Control.TabStop

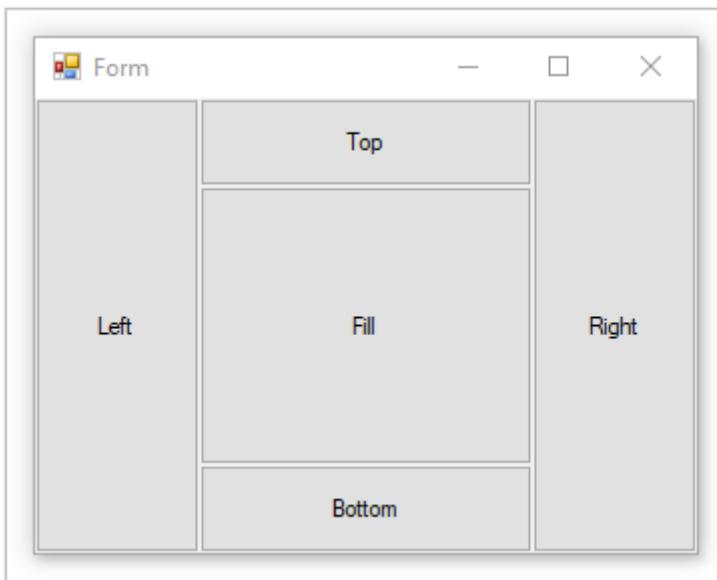
如何停靠和定位控件（Windows 窗体 .NET）

项目 • 2025/01/30

如果要设计用户可以在运行时调整大小的窗体，窗体上的控件应调整大小并正确重新定位。当窗体大小发生变化时，控件具有两个特性，可以帮助自动放置和调整大小。

- [Control.Dock](#)

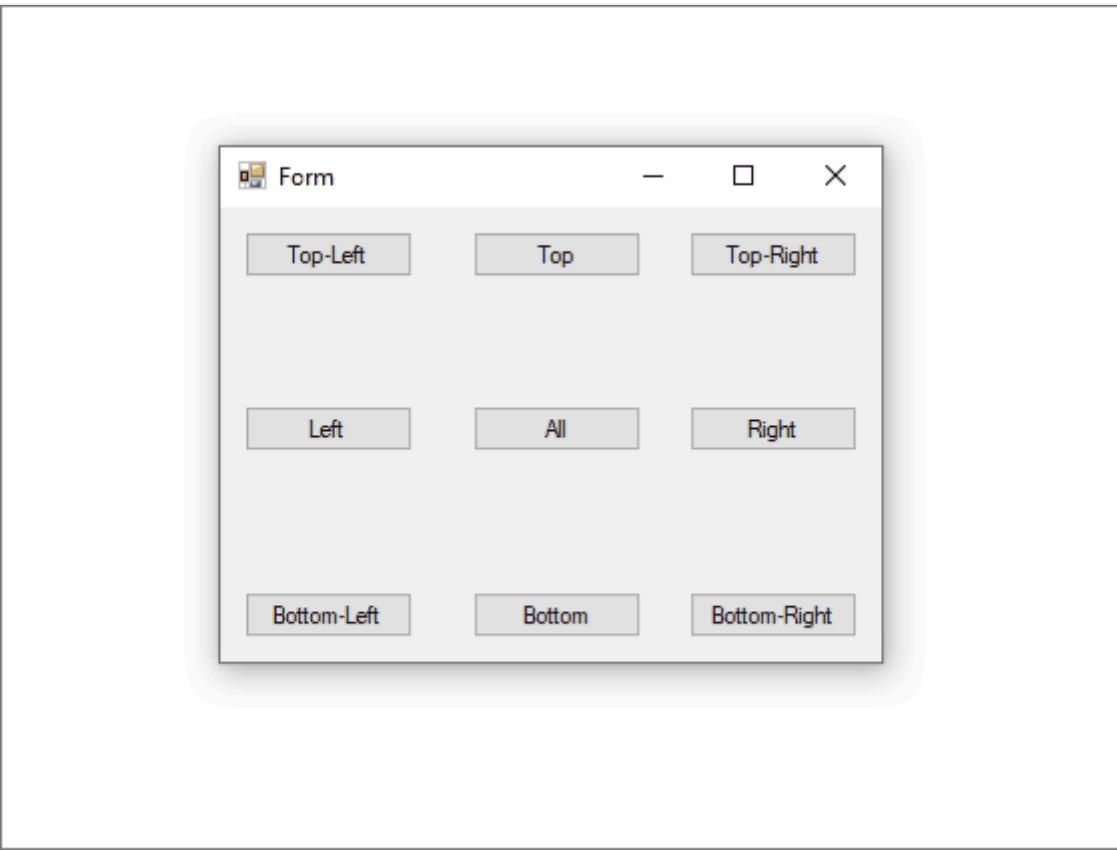
停靠的控件填充控件容器（窗体或容器控件）的边缘。例如，Windows 资源管理器将其 [TreeView](#) 控件停靠在窗口的左侧，将其 [ListView](#) 控件停靠在窗口的右侧。停靠模式可以是控件容器的任何一侧，也可以设置为填充容器的剩余空间。



控件以反向 z 轴顺序停靠，[Dock](#) 属性与 [AutoSize](#) 属性互相作用。有关详细信息，请参阅[自动调整大小](#)。

- [Control.Anchor](#)

重设已定位的控件窗体的大小后，控件将保持它与定位点位置之间的距离。例如，如果你有一个定位在窗体左边缘、右边缘和下边缘的 [TextBox](#) 控件，那么当窗体的大小重设后，[TextBox](#) 控件会在水平方向重设大小，使其与窗体的右侧和左侧保持相同距离。控件还在垂直方向定位自身，以便其位置始终与窗体下边缘的距离相同。如果控件未被锚定，当窗体大小调整时，控件相对于窗体边缘的位置将发生变化。



有关详细信息，请参阅 [控件的位置和布局](#)。

停靠控件

控件通过设置其 [Dock](#) 属性来停靠。

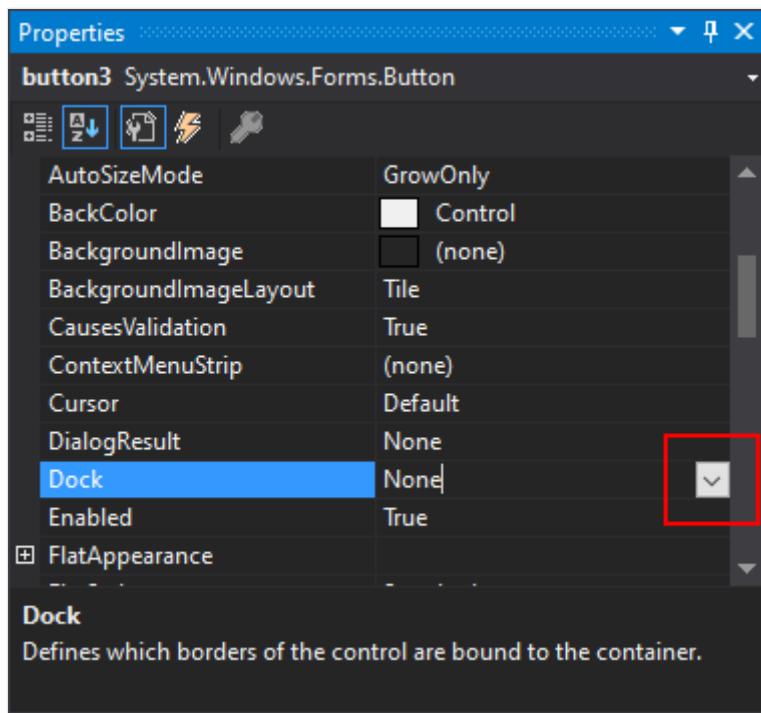
① 备注

继承的控件必须为 [Protected](#) 才能停靠。若要更改控件的访问级别，在 **属性** 窗口中设置其 **修饰符** 属性。

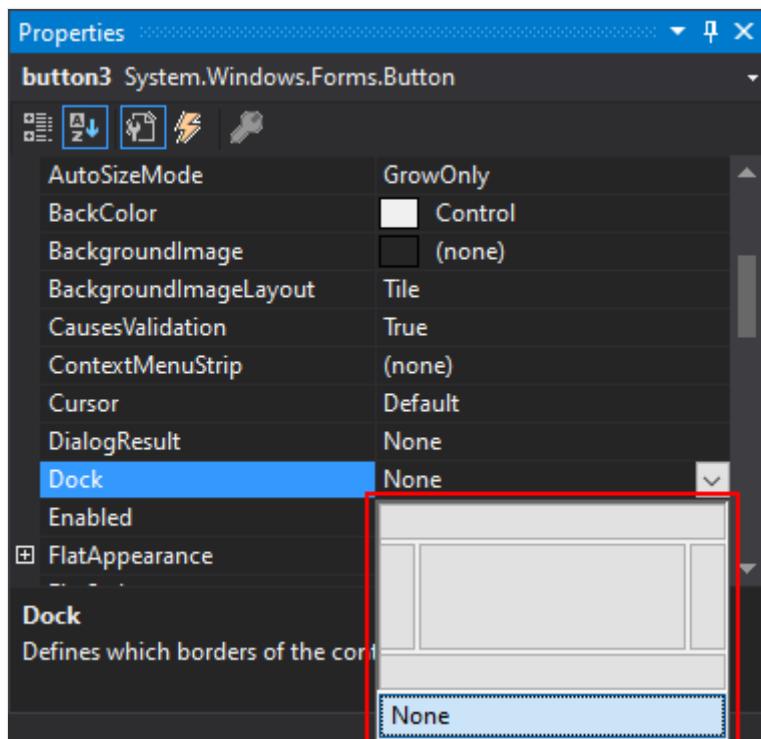
使用设计器

使用 Visual Studio 设计器 **属性** 窗口来设置控件的对接模式。

1. 在设计器中选择控件。
2. 在属性窗口中，选择“停靠”属性右侧的箭头。



3. 选择表示要停靠控件的容器边缘的按钮。 若要填充控件窗体或容器控件的内容，请按中心框。 单击“(无)”以禁用停靠。



控件将自动重设大小以适应停靠边缘的边界。

以编程方式设置停靠

1. 设置控件上的 Dock 属性。 在此示例中，按钮停靠在其容器右侧：

C#

```
button1.Dock = DockStyle.Right;
```

定位控件

通过将控件的 [Anchor](#) 属性设置为一个或多个值来将其定位到边缘。

① 备注

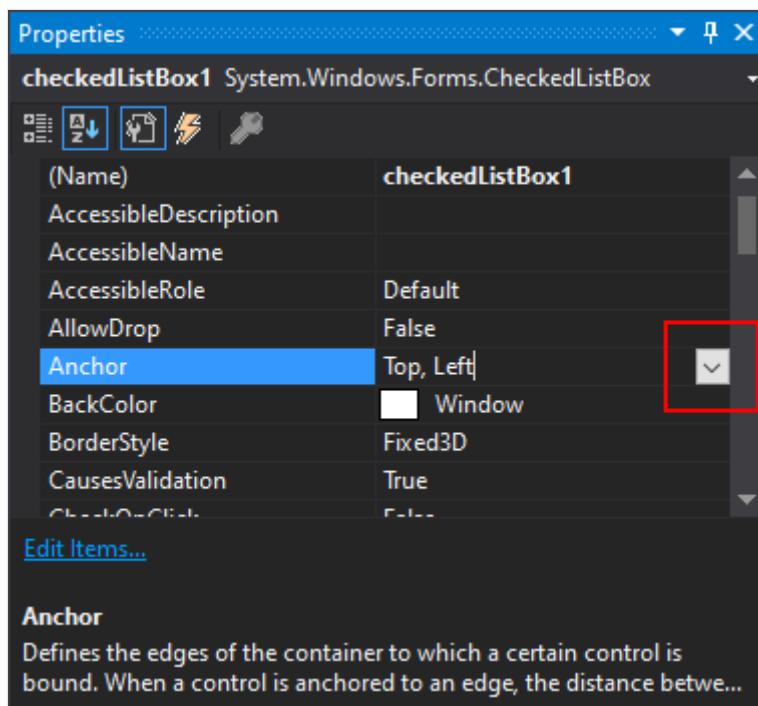
某些控件（如 [ComboBox](#) 控件）对其高度有限制。将控件定位到其窗体或容器底部不能强制控件超出其高度限制。

继承的控件必须为 [Protected](#) 才能定位。若要更改控件的访问级别，请在其 [属性](#) 窗口中设置其 [Modifiers](#) 属性。

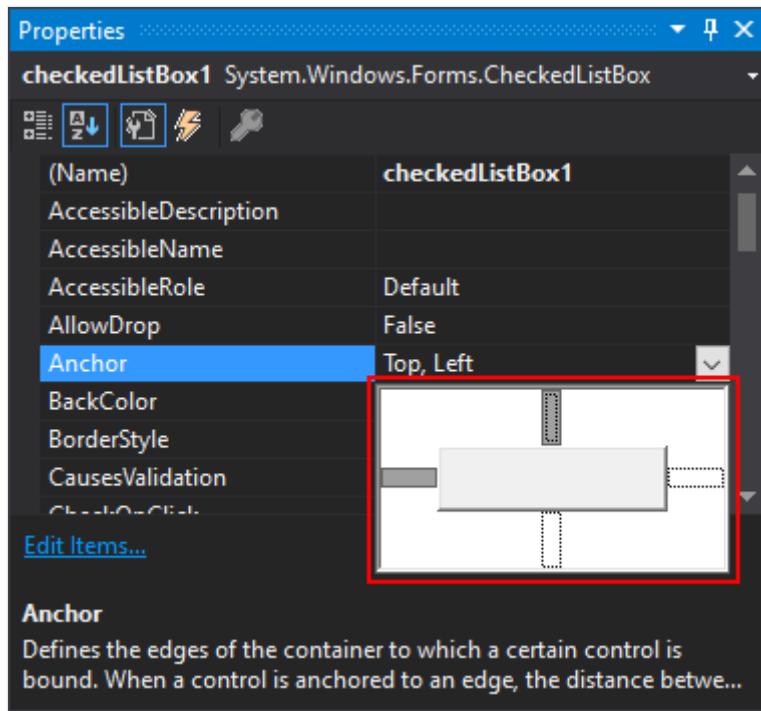
使用设计器

使用 Visual Studio 设计器的属性窗口来设置控件的定位边缘。

1. 在设计器中选择控件。
2. 在属性窗口中，选择“定位点”属性右侧的箭头。



3. 若要设置或取消锚点，请选择十字形的顶部、左边、右边或底部的臂。



以编程方式设置定位点

1. 设置控件上的 `Anchor` 属性。在此示例中，按钮固定在其容器的右侧和底部两侧：

C#

```
button1.Anchor = AnchorStyles.Bottom | AnchorStyles.Right;
```

另请参阅

- [控件的位置和布局。](#)
- [System.Windows.Forms.Control.Anchor](#)
- [System.Windows.Forms.Control.Dock](#)

如何在控件上显示图像（Windows 窗体 .NET）

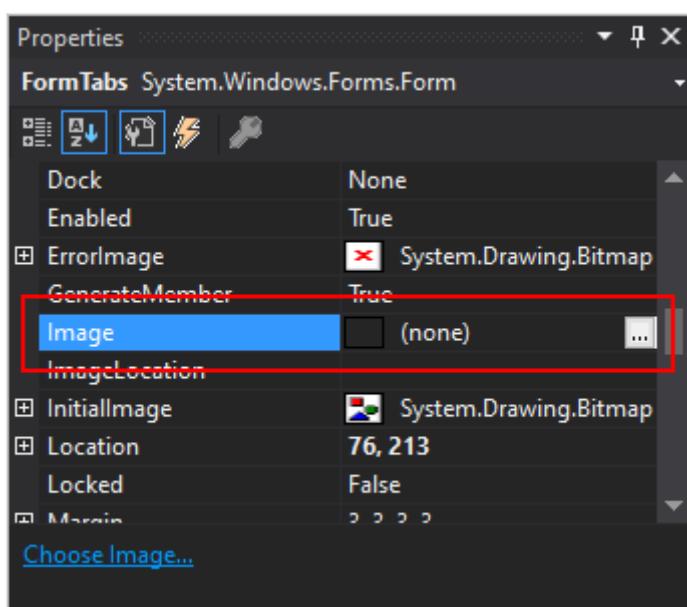
项目 • 2025/01/30

多个 Windows 窗体控件可以显示图像。这些图像可以是阐明控件用途的图标，例如表示“保存”命令的按钮上的磁盘图标。或者，图标可以是背景图像，以便为控件提供所需的外观和行为。

显示图像 - 设计器

在 Visual Studio 中，使用视觉设计器显示图像。

1. 打开包含待更改控件的表单的视觉设计器。
2. 选择控件。
3. 在“属性”窗格中，选择控件的 **Image** 或 **BackgroundImage** 属性。
4. 选择省略号 (**...**) 以显示“选择资源”对话框，然后选择要显示的图像。



显示图像 - 代码

将控件的 `Image` 或 `BackgroundImage` 属性设置为 `Image`类型的对象。通常，你将使用 `FromFile` 方法从文件加载映像。

在以下代码示例中，为图像位置设置的路径是“我的图片”文件夹。运行 Windows 操作系统的大多数计算机都包含此目录。这也使系统访问级别最少的用户能够安全地运行应

用程序。下面的代码示例要求你已具有一个添加了 **PictureBox** 控件的窗体。

C#

```
// Replace the image named below with your own icon.  
// Note the escape character used (@) when specifying the path.  
pictureBox1.Image = Image.FromFile  
    (System.Environment.GetFolderPath  
    (System.Environment.SpecialFolder.MyPictures)  
    + @"\Image.gif");
```

另请参阅

- [System.Drawing.Image.FromFile](#)
- [System.Drawing.Image](#)
- [System.Windows.Forms.Control.BackgroundImage](#)

如何处理控制事件 (Windows Forms .NET)

项目 • 2024/12/18

控件 (和窗体) 的事件通常通过 Visual Studio 的 Windows 窗体视觉设计器进行设置。通过可视化设计工具设置事件被称为在设计时处理事件。还可以在代码中动态处理事件，称为在运行时处理事件。通过运行时创建的事件，可以根据应用当前正在执行的操作动态连接事件处理程序。

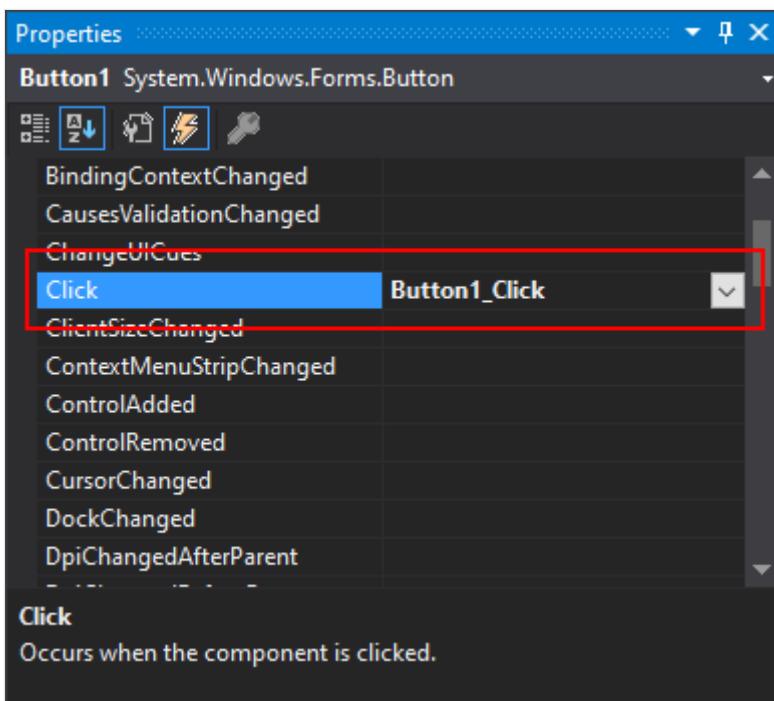
处理事件 - 设计器

在 Visual Studio 中，使用 Visual Designer 管理控件事件的处理程序。视觉设计器将生成处理程序代码，并将其添加到事件中。

设置处理程序

使用 **属性面板** 来添加或设置事件的处理程序：

1. 打开包含要更改的控件的窗体的可视化设计器。
2. 选择控件。
3. 通过按事件按钮 (⚡) 将 **属性** 窗格模式更改为 **事件**。
4. 查找要向其添加处理程序的事件，例如，**Click** 事件：



5. 执行下列操作之一：

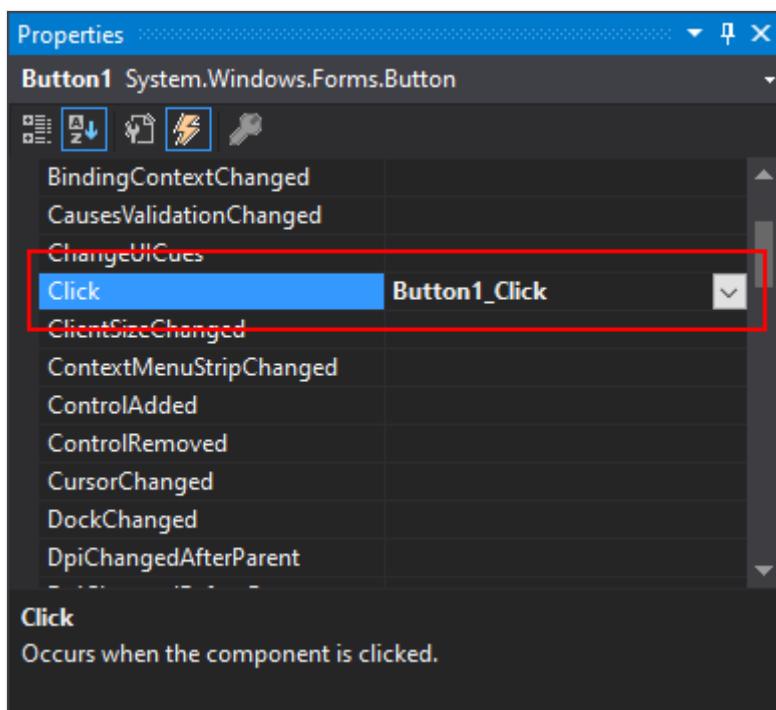
- 双击事件以生成新处理程序，如果未分配任何处理程序，则为空。如果不是空白，此操作将打开窗体代码，并导航到现有的事件处理程序。
- 使用选择框 () 选择现有处理程序。

选择框将列出具有事件处理程序兼容方法签名的所有方法。

清除处理程序

若要删除事件处理程序，不能只删除表单代码隐藏文件中的处理程序代码，该代码仍由事件引用。使用 **属性** 窗格来移除事件的处理程序。

1. 打开包含控件的窗体的可视化设计器以进行更改。
2. 选择控件。
3. 通过按事件按钮 () 将 **属性** 窗格模式更改为 **事件**。
4. 查找包含要删除的处理程序的事件，例如，**Click** 事件：



5. 右键单击事件并选择 **重置**。

处理事件 - 代码

通常通过可视化设计器在设计时向控件添加事件处理程序。不过，可以在运行时创建控件，这要求在代码中添加事件处理程序。在代码中添加处理程序也使你有机会向同一事

件添加多个处理程序。

添加处理程序

以下示例演示如何创建控件并添加事件处理程序。此控件是在 `Button.Click` 事件处理程序中创建另一个按钮。当按下 **按钮1** 时，代码移动并调整新按钮的大小。新按钮的 `Click` 事件由 `MyNewButton_Click` 方法处理。为使新按钮显示，需要将其添加到窗体的 `Controls` 集合中。还有用于删除 `Button1.Click` 事件处理程序的代码，[删除处理程序](#) 部分对此进行了讨论。

```
C#  
  
private void button1_Click(object sender, EventArgs e)  
{  
    // Create and add the button  
    Button myNewButton = new()  
    {  
        Location = new Point(10, 10),  
        Size = new Size(120, 25),  
        Text = "Do work"  
    };  
  
    // Handle the Click event for the new button  
    myNewButton.Click += MyNewButton_Click;  
    this.Controls.Add(myNewButton);  
  
    // Remove this button handler so the user cannot do this twice  
    button1.Click -= button1_Click;  
}  
  
private void MyNewButton_Click(object sender, EventArgs e)  
{  
}
```

若要运行此代码，请使用 Visual Studio Visual Designer 对窗体执行以下操作：

1. 向窗体添加新按钮并将其命名 **Button1**。
2. 按事件按钮 ()，将 **属性** 窗格模式更改为 **事件**。
3. 双击 `Click` 事件以生成处理程序。此操作将打开代码窗口，并生成空白 `Button1_Click` 方法。
4. 将方法代码替换为上面的代码。

有关 C# 事件的详细信息，请参阅 [事件 \(C#\)](#) 有关 Visual Basic 事件的详细信息，请参阅 [事件 \(Visual Basic\)](#)

删除处理程序

[添加处理程序](#) 节使用了一些代码来演示如何添加处理程序。该代码还包含删除处理程序的调用：

C#

```
button1.Click -= button1_Click;
```

此语法可用于从任何事件中删除任何事件处理程序。

有关 C# 事件的详细信息，请参阅 [事件 \(C#\)](#) 有关 Visual Basic 事件的详细信息，请参阅 [事件 \(Visual Basic\)](#)

如何将多个事件与同一处理程序一起使用

使用 Visual Studio Visual Designer 的“属性”窗格，可以选择其他事件已使用的同一处理程序。按照 [设置处理程序](#) 部分的说明选择现有处理程序，而不是创建新处理程序。

在 C# 中，处理程序被附加到窗体设计器代码中的控件事件，而窗体设计器代码是通过可视化设计器进行更改的。有关 C# 事件的详细信息，请参阅 [事件 \(C#\)](#)

Visual Basic

在 Visual Basic 中，事件处理程序被附加到控件的事件中，并在窗体的代码隐藏文件中声明了处理程序代码。可以将多个 `Handles` 关键字添加到事件处理程序代码中，以将其与多个事件一起使用。视觉设计器将为你生成 `Handles` 关键字，并将其添加到事件处理程序。但是，只要处理程序方法的签名与事件匹配，就可以自行对任何控件的事件和事件处理程序执行此操作。有关 Visual Basic 事件的详细信息，请参阅 [事件 \(Visual Basic\)](#)

此代码演示如何将同一方法用作两个不同的 `Button.Click` 事件的处理程序：

VB

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles
    Button1.Click, Button2.Click
    'Do some work to handle the events
End Sub
```

另请参阅

- [控制事件](#)

- 事件概述
- 使用鼠标事件
- 使用键盘操作事件
- System.Windows.Forms.Button

如何对控件进行线程安全的调用 (Windows 窗体 .NET)

项目 • 2024/11/05

多线程处理可以改进 Windows 窗体应用的性能，但对 Windows 窗体控件的访问本质上不是线程安全的。多线程处理可将代码公开到严重和复杂的 bug。有两个或两个以上线程操作控件可能会迫使该控件处于不一致状态并导致争用条件、死锁和冻结或挂起。如果要在应用中实现多线程处理，请务必以线程安全的方式调用跨线程控件。有关详细信息，请参阅[托管线程处理的最佳做法](#)。

可通过两种方法从未创建 Windows 窗体控件的线程安全地调用该控件。使用 `System.Windows.Forms.Control.Invoke` 方法调用在主线程中创建的委托，进而调用控件。或者，实现一个 `System.ComponentModel.BackgroundWorker`，它使用事件驱动模型将后台线程中完成的工作与结果报告分开。

不安全的跨线程调用

直接从未创建控件的线程调用该控件是不安全的。以下代码片段演示了对 `System.Windows.Forms.TextBox` 控件的不安全调用。`Button1_Click` 事件处理程序创建一个新的 `WriteTextUnsafe` 线程，该线程直接设置主线程的 `TextBox.Text` 属性。

```
C#  
  
private void button1_Click(object sender, EventArgs e)  
{  
    var thread2 = new System.Threading.Thread(WriteTextUnsafe);  
    thread2.Start();  
}  
  
private void WriteTextUnsafe() =>  
    textBox1.Text = "This text was set unsafely.";
```

Visual Studio 调试器通过引发 `InvalidOperationException` 检测这些不安全线程调用，并显示消息“跨线程操作无效。控件从创建它的线程以外的线程访问。”在 Visual Studio 调试期间，对于不安全的跨线程调用总是会发生 `InvalidOperationException`，并且可能在应用运行时发生。应解决此问题，但也可以通过将 `Control.CheckForIllegalCrossThreadCalls` 属性设置为 `false` 来禁用该异常。

安全的跨线程调用

以下代码示例演示了两种从未创建 Windows 窗体控件的线程安全调用该窗体的方法：

1. [System.Windows.Forms.Control.Invoke](#) 方法，它从主线程调用委托以调用控件。
2. [System.ComponentModel.BackgroundWorker](#) 组件，它提供事件驱动模型。

在这两个示例中，后台线程都会休眠一秒钟以模拟该线程中正在完成的工作。

示例：使用 Invoke 方法

下面的示例演示了一种用于确保对 Windows 窗体控件进行线程安全调用的模式。它查询 [System.Windows.Forms.Control.InvokeRequired](#) 属性，该属性将控件的创建线程 ID 与调用线程 ID 进行比较。如果它们不同，应调用 [Control.Invoke](#) 方法。

`WriteTextSafe` 允许将 `TextBox` 控件的 `Text` 属性设置为一个新值。该方法查询 `InvokeRequired`。如果 `InvokeRequired` 返回 `true`，则 `WriteTextSafe` 以递归方式调用自身，并将该方法作为委托传递给 `Invoke` 方法。如果 `InvokeRequired` 返回 `false`，则 `WriteTextSafe` 直接设置 `TextBox.Text`。`Button1_Click` 事件处理程序创建新线程并运行 `WriteTextSafe` 方法。

```
C#  
  
private void button1_Click(object sender, EventArgs e)  
{  
    var threadParameters = new System.Threading.ThreadStart(delegate {  
        WriteTextSafe("This text was set safely.");});  
    var thread2 = new System.Threading.Thread(threadParameters);  
    thread2.Start();  
}  
  
public void WriteTextSafe(string text)  
{  
    if (textBox1.InvokeRequired)  
    {  
        // Call this same method but append THREAD2 to the text  
        Action safeWrite = delegate { WriteTextSafe($"{text} (THREAD2)"); };  
        textBox1.Invoke(safeWrite);  
    }  
    else  
        textBox1.Text = text;  
}
```

示例：使用 BackgroundWorker

实现多线程处理的一种简单方法是使用 [System.ComponentModel.BackgroundWorker](#) 组件，该组件使用事件驱动模型。后台线程引发不与主线程交互的

`BackgroundWorker.DoWork` 事件。主线程运行 `BackgroundWorker.ProgressChanged` 和 `BackgroundWorker.RunWorkerCompleted` 事件处理程序，它们可以调用主线程的控件。

要使用 `BackgroundWorker` 进行线程安全的调用，请处理 `DoWork` 事件。后台辅助角色使用两个事件来报告状态：`ProgressChanged` 和 `RunWorkerCompleted`。

`ProgressChanged` 事件用于将状态更新传达给主线程，而 `RunWorkerCompleted` 事件用于指示后台辅助角色已完成其工作。若要启动后台线程，请调用 `BackgroundWorker.RunWorkerAsync`。

该示例在 `DoWork` 事件中从 0 到 10 进行计数，计数之间暂停一秒钟。它使用 `ProgressChanged` 事件处理程序将数字报告回主线程并设置 `TextBox` 控件的 `Text` 属性。要使 `ProgressChanged` 事件有效，必须将 `BackgroundWorker.WorkerReportsProgress` 属性设置为 `true`。

C#

```
private void button1_Click(object sender, EventArgs e)
{
    if (!backgroundWorker1.IsBusy)
        backgroundWorker1.RunWorkerAsync();
}

private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    int counter = 0;
    int max = 10;

    while (counter <= max)
    {
        backgroundWorker1.ReportProgress(0, counter.ToString());
        System.Threading.Thread.Sleep(1000);
        counter++;
    }
}

private void backgroundWorker1_ProgressChanged(object sender,
ProgressChangedEventArgs e) =>
    textBox1.Text = (string)e.UserState;
```

自定义控件 (Windows Forms .NET)

项目 • 2024/12/18

使用 Windows 窗体，可以通过继承创建新控件或修改现有控件。本文重点介绍了创建新控件的方式之间的差异，并提供有关如何为项目选择特定类型的控件的信息。

基础控件类

[Control](#) 类是 Windows 窗体控件的基类。它提供 Windows 窗体应用程序中视觉显示所需的基础结构，并提供以下功能：

- 暴露窗口句柄。
- 管理消息路由。
- 提供鼠标和键盘事件，以及许多其他用户界面事件。
- 提供高级布局功能。
- 包含特定于视觉显示的许多属性，例如 [ForeColor](#)、[BackColor](#)、[Height](#) 和 [Width](#)。

由于大部分基础结构由基类提供，因此开发自己的 Windows 窗体控件相对容易。

创建自己的控件

可以创建三种类型的自定义控件：用户控件、扩展控件和自定义控件。下表可帮助你确定应创建的控件类型：

[+] 展开表

如果...	创建 ...
<ul style="list-style-type: none">你想要将多个 Windows 窗体控件的功能合并到单个可重用单元中。	通过继承 System.Windows.Forms.UserControl ，设计 用户控件 。
<ul style="list-style-type: none">你所需的大多数功能已经与现有的 Windows 窗体控件相同。您不需要自定义图形用户界面，或者您想要为现有控件设计新的图形用户界面。	通过从特定的 Windows 窗体控件继承来扩展控件。
<ul style="list-style-type: none">你希望为控件提供自定义的图形表示形式。你需要实现无法通过标准控件提供的自定义功能。	通过继承 System.Windows.Forms.Control ，创建 自定义控件 。

用户控件

用户控件是作为单个控件提供给使用者的 Windows 窗体控件的集合。此类控件称为 **复合控件**。包含的控件称为 **组件控件**。

用户控件包含与每个包含的 Windows 窗体控件关联的所有固有功能，使你可以有选择地公开和绑定其属性。用户控件还无需任何额外开发努力即可提供大量默认键盘处理功能。

例如，可以生成用户控件以显示数据库中的客户地址数据。此控件将包括用于显示数据库字段的 [DataGridView](#) 控件、用于处理绑定到数据源的 [BindingSource](#)，以及用于在记录中移动的 [BindingNavigator](#) 控件。你可以有选择地公开数据绑定属性，并且可以打包并重复使用整个控件，从应用程序到应用程序。

有关详细信息，请参阅 [用户控件概述](#)。

扩展控件

您可以从任何现有的 Windows 窗体控件派生一个继承控件。使用此方法，你可以保留 Windows 窗体控件的所有固有功能，然后通过添加自定义属性、方法或其他功能来扩展该功能。使用此选项，您可以覆盖基本控件的绘图逻辑，然后通过更改其外观来扩展用户界面。

例如，可以创建派生自 [Button](#) 控件的控件，该控件跟踪用户单击的次数。

在某些控件中，还可以通过重写基类的 [OnPaint](#) 方法，向控件的图形用户界面添加自定义外观。对于跟踪单击的扩展按钮，可以重写 [OnPaint](#) 方法以调用 [OnPaint](#) 的基本实现，然后在 [Button](#) 控件工作区的一角绘制单击计数。

自定义控件

创建控件的另一种方法是通过从 [Control](#) 继承，从头开始创建一个控件。[Control](#) 类提供控件所需的所有基本功能，包括鼠标和键盘处理事件，但没有特定于控件的功能或图形界面。

通过继承自 [Control](#) 类来创建控件，比继承自 [UserControl](#) 或现有的 Windows 窗体控件需要更多的思考和努力。由于为你留下了大量实现，因此你的控件可以比复合控件或扩展控件具有更大的灵活性，并且你可以定制控件以满足你的确切需求。

若要实现自定义控件，必须为控件的 [OnPaint](#) 事件编写代码，该事件控制控件的直观绘制方式。您还必须为控件编写功能特定的行为。还可以替代 [WndProc](#) 方法并直接处理

Windows 消息。这是创建控件的最强大方法，但要有效地使用此技术，需要熟悉 Microsoft Win32® API。

自定义控件的一个示例是一个时钟控件，它复制模拟时钟的外观和行为。为了响应来自内部 [Timer](#) 组件的 [Tick](#) 事件，调用自定义绘制以使时钟的指针移动。

自定义设计体验

如果需要实现自定义设计时体验，可以编写自己的设计器。对于复合控件，请从 [ParentControlDesigner](#) 或 [DocumentDesigner](#) 类派生自定义设计器类。对于扩展控件和自定义控件，请从 [ControlDesigner](#) 类派生自定义设计器类。

使用 [DesignerAttribute](#) 将您的控件与设计器相关联。

以下信息已过期，但可能有助于你。

- [\(Visual Studio 2013\) 扩展 Design-Time 支持。](#)
- [\(Visual Studio 2013\) 如何创建一个利用 Design-Time 功能的 Windows 窗体控件。](#)

用户控件设计指南 (Windows Forms .NET)

项目 • 2025/01/30

本文提供了创建用户控件的指南。建议遵循以下准则，确保设计与其他 Windows 窗体控件一致的用户控件。

定义事件

事件通常传达状态更改并提醒用户如何与 Windows 窗体控件交互。有关事件和委托的详细信息，请参阅[处理和引发事件](#)。

定义自己的事件时，请遵循以下建议：

- 定义没有任何关联数据的事件时，请使用 `EventHandler` 事件委托。在你拥有关联数据时，请使用 `EventHandler<EventArgs>` 事件委托。
- 在引发具有关联数据的事件时，从 `EventArgs` 派生并用你的数据扩展它。
- 将控件实例作为 `sender` 参数传递。
- 创建一个名为 `On{EventName}` 的方法，该方法引发事件，该事件标记为 `protected` 和 `virtual`（在 C# 中）或 `Protected` 和 `Overridable`（在 Visual Basic 中）。

C#

```
// The event
public event EventHandler AllowInteractionChanged;

// The backing field for the property
private bool _allowInteraction;

// The property
public bool AllowInteraction
{
    get => _allowInteraction;
    set
    {
        // When the value has changed, call the method to raise the event
        if (_allowInteraction != value)
        {
            _allowInteraction = value;
            OnAllowInteractionChanged();
        }
    }
}

// Raises the event
private void OnAllowInteractionChanged()
```

```
public virtual void OnAllowInteractionChanged() =>
    AllowInteractionChanged?.Invoke(this, EventArgs.Empty);
```

属性更改事件

如果希望控件在属性发生更改时发送通知，请定义名为 `{PropertyName}Changed` 的事件。这是 Windows 窗体中使用的命名约定。属性更改事件的关联事件委托类型是 `EventHandler`，事件数据类型则是 `EventArgs`。基类 `Control` 定义许多属性更改的事件，例如 `BackColorChanged`、`BackgroundImageChanged`、`FontChanged`、`LocationChanged`。当属性遵循此命名约定时，它会收到双向数据绑定支持。

[定义事件](#) 部分中的相同建议也适用于此处。

C#

```
public class ProgressReportEventArgs : EventArgs
{
    public readonly int Value;
    public readonly int Maximum;

    public ProgressReportEventArgs(int value, int maximum) =>
        (Value, Maximum) = (value, maximum);
}

public event EventHandler<ProgressReportEventArgs> ProgressChanged;

public virtual void OnProgressChanged(int value, int maximum) =>
    ProgressChanged?.Invoke(this, new ProgressReportEventArgs(value,
maximum));
```

性能

控件属性应支持 Windows 窗体视觉设计器。**属性** 窗口与控件属性交互，用户希望使用它来更改控件的属性。将 `DefaultValueAttribute` 添加到属性，或创建相应的 `Reset<PropertyName>` 和 `ShouldSerialize<PropertyName>` 方法。有关详细信息，请参阅 [DefaultValueAttribute](#) 和 [Reset](#) 和 [ShouldSerialize](#)。

不希望向 Windows 窗体可视化设计器公开的属性应向该属性添加 `BrowsableAttribute`，为特性的参数传递 `false`。这会将该属性从 **属性** 窗口中隐藏起来。有关详细信息，请参阅 [定义属性](#) 和 [属性的特性](#)。

How to extend an existing control (Windows Forms .NET)

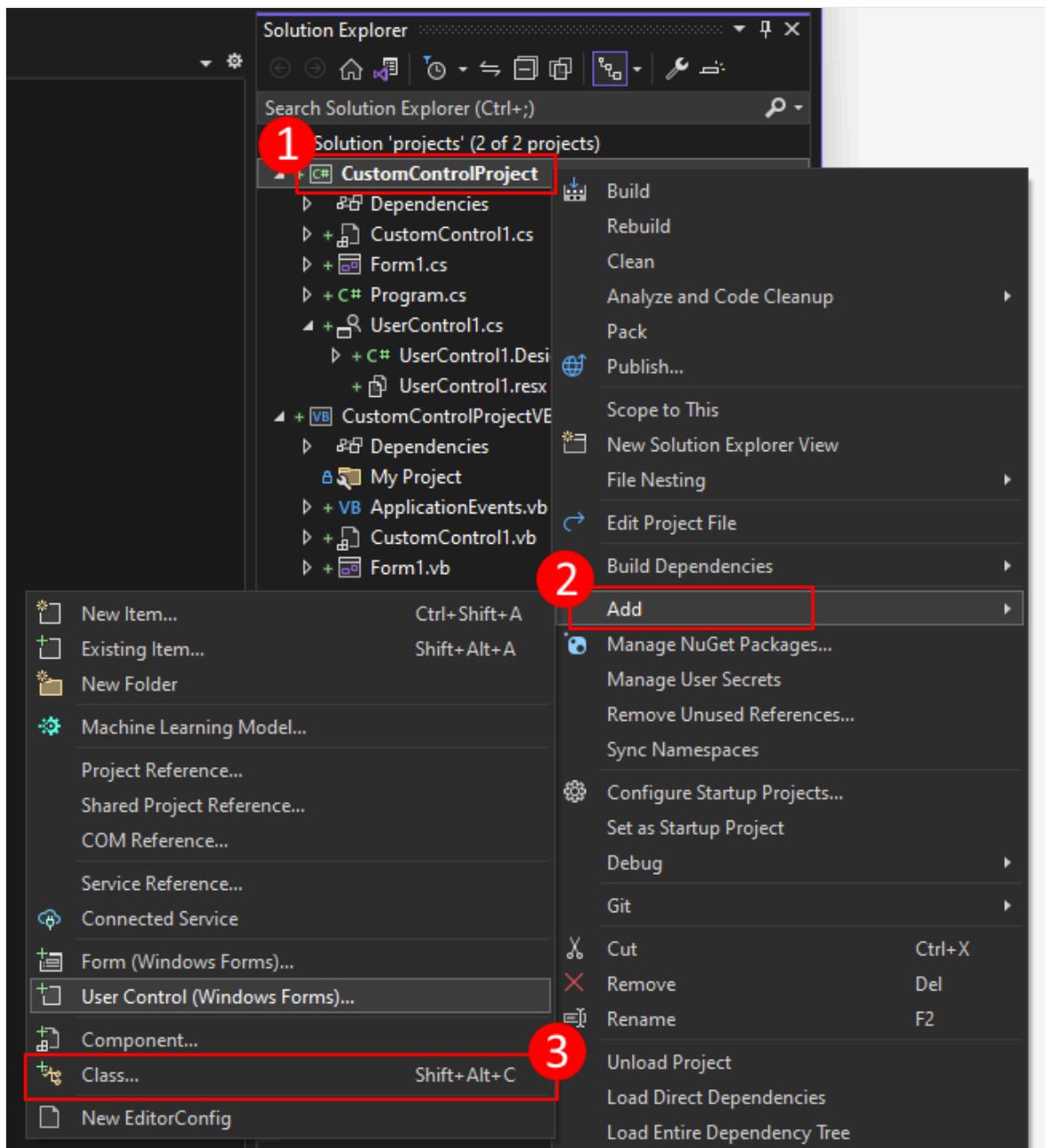
Article • 01/24/2025

If you want to add more features to an existing control, you can create a control that inherits from an existing control. The new control contains all of the capabilities and visual aspect of the base control, but gives you opportunity to extend it. For example, if you created a control that inherits [Button](#), your new control would look and act exactly like a button. You could create new methods and properties to customize the behavior of the control. Some controls allow you to override the [OnPaint](#) method to change the way the control looks.

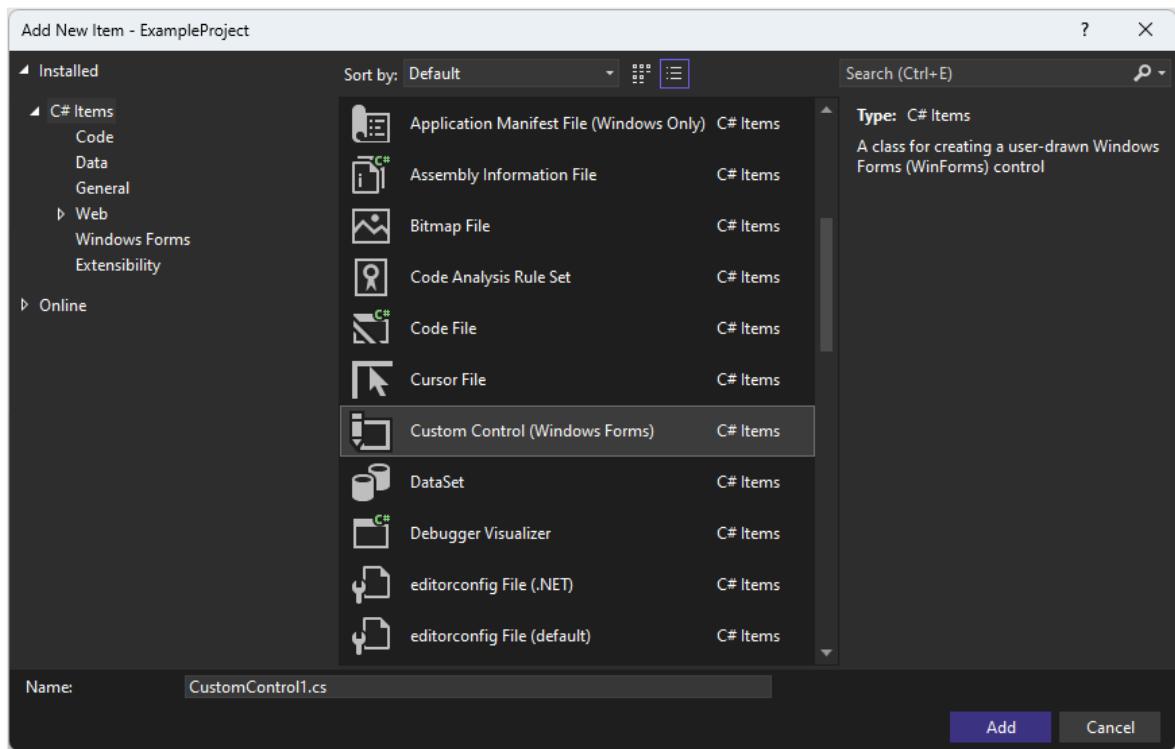
Add a custom control to a project

After creating a new project, use the Visual Studio templates to create a user control. The following steps demonstrate how to add a user control to your project:

1. In Visual Studio, find the **Project Explorer** pane. Right-click on the project and choose **Add > Class**.



2. In the **Name** box, type a name for your user control. Visual Studio provides a default and unique name that you may use. Next, press **Add**.



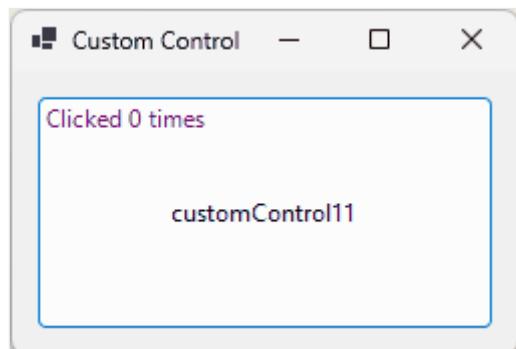
3. In **Design** mode of the control, press **F7** or click the **switch to code view** link.

💡 Tip

You can also right-click the file in the **Solution Explorer** window and select **View Code**.

Change the custom control to a button

In this section, you learn how to change a custom control into a button that counts and displays the number of times it's clicked.



After you add a custom control to your project named `CustomControl1`, the control designer should be opened. If it's not, double-click on the control in the **Solution Explorer**. Follow these steps to convert the custom control into a control that inherits from `Button` and extends it:

1. With the control designer opened, press **F7** or right-click on the designer window and select **View Code**.

2. In the code-editor, you should see a class definition:

```
C#  
  
namespace CustomControlProject  
{  
    public partial class CustomControl2 : Control  
    {  
        public CustomControl2()  
        {  
            InitializeComponent();  
        }  
  
        protected override void OnPaint(PaintEventArgs pe)  
        {  
            base.OnPaint(pe);  
        }  
    }  
}
```

3. Change the base class from `Control` to `Button`.

Important

If you're using **Visual Basic**, the base class is defined in the `*.designer.vb` file of your control. The base class to use in Visual Basic is

`System.Windows.Forms.Button`.

4. Add a class-scoped variable named `_counter`.

```
C#  
  
private int _counter = 0;
```

5. Override the `OnPaint` method. This method draws the control. The control should draw a string on top of the button, so you must call the base class' `OnPaint` method first, then draw a string.

```
C#  
  
protected override void OnPaint(PaintEventArgs pe)  
{  
    // Draw the control
```

```
base.OnPaint(pe);

// Paint our string on top of it
pe.Graphics.DrawString($"Clicked {_counter} times", Font,
Brushes.Purple, new PointF(3, 3));
}
```

6. Lastly, override the `onClick` method. This method is called every time the control is pressed. The code is going to increase the counter, and then call the `Invalidate` method, which forces the control to redraw itself.

C#

```
protected override void OnClick(EventArgs e)
{
    // Increase the counter and redraw the control
    _counter++;
    Invalidate();

    // Call the base method to invoke the Click event
    base.OnClick(e);
}
```

The final code should look like the following snippet:

C#

```
public partial class CustomControl1 : Button
{
    private int _counter = 0;

    public CustomControl1()
    {
        InitializeComponent();
    }

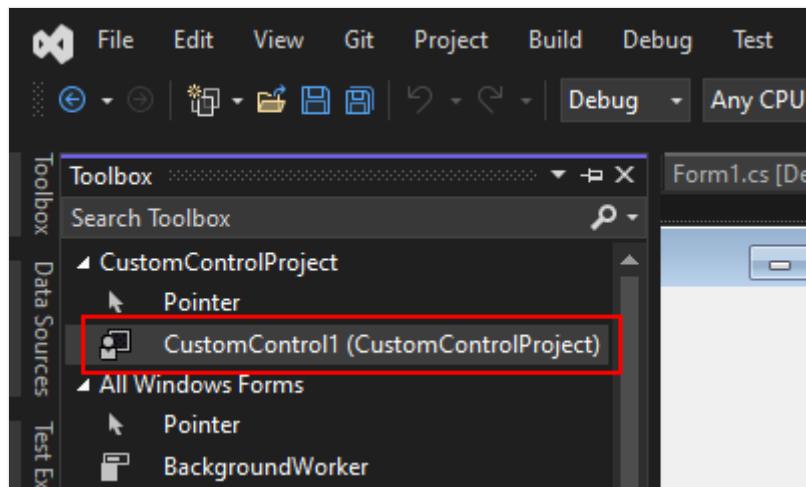
    protected override void OnPaint(PaintEventArgs pe)
    {
        // Draw the control
        base.OnPaint(pe);

        // Paint our string on top of it
        pe.Graphics.DrawString($"Clicked {_counter} times", Font,
Brushes.Purple, new PointF(3, 3));
    }

    protected override void OnClick(EventArgs e)
    {
        // Increase the counter and redraw the control
        _counter++;
        Invalidate();
    }
}
```

```
// Call the base method to invoke the Click event  
base.OnClick(e);  
}  
}
```

Now that the control is created, compile the project to populate the **Toolbox** window with the new control. Open a form designer and drag the control to the form. Run the project and press the button. Each press increases the number of clicks by one. The total clicks are printed as text on top of the button.



用户控件概述 (Windows 窗体 .NET)

项目 • 2025/01/31

用户控件是封装在通用容器中的 Windows 窗体控件的集合。此类控件称为 **复合控件**。其包含的控件称为构成控件。用户控件派生自 [UserControl](#) 类。

用户控件的设计方式与窗体类似，具有可视化设计器。通过视觉设计器创建、排列和修改构成控件。控件事件和逻辑的编写方式与设计窗体时完全相同。用户控件与其他任何控件一样放置在表单上。

用户控件可以在创建它们的项目中使用，也可以在引用这些用户控件库的其他项目中使用。

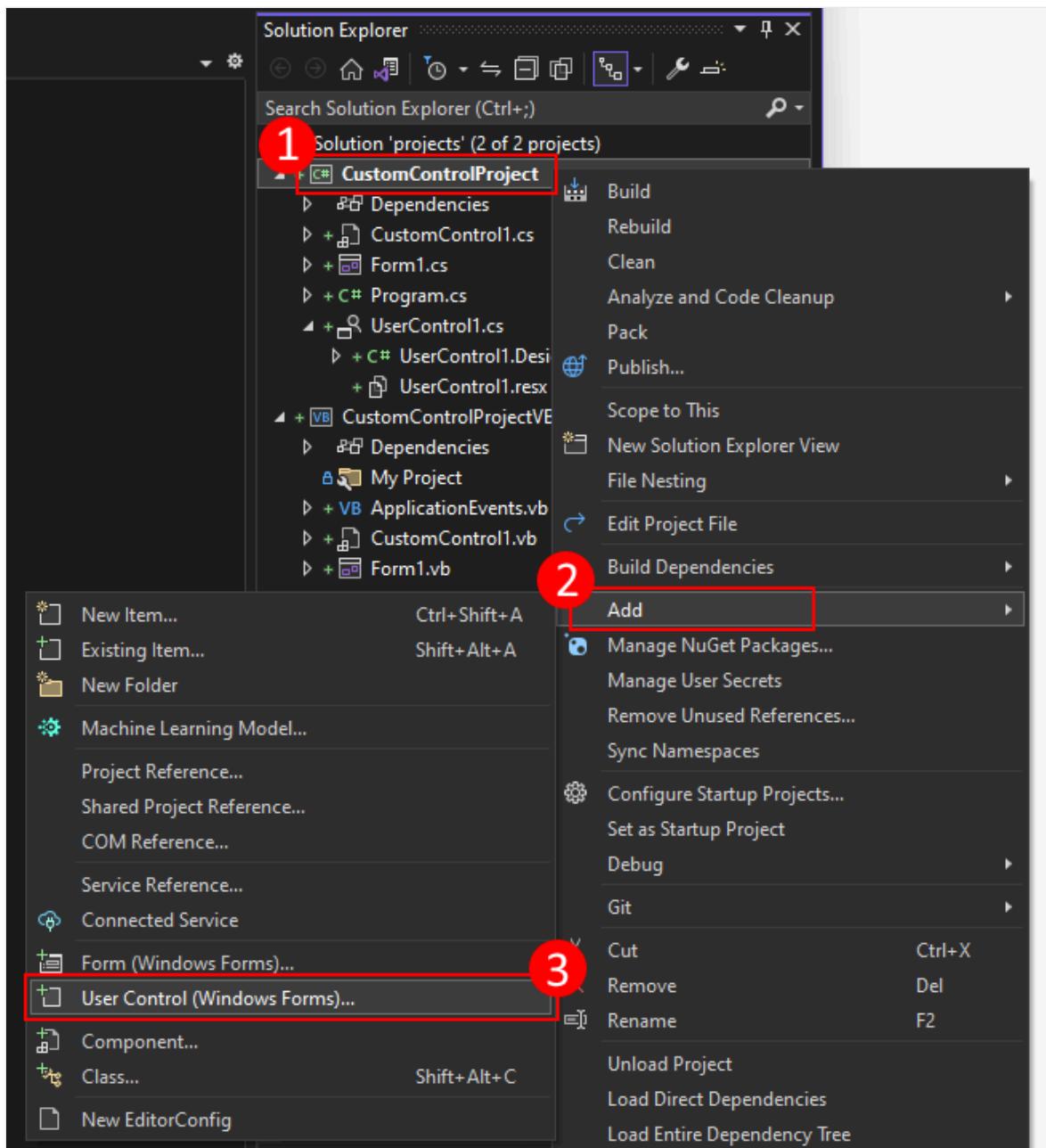
构成控件

构成控件可供用户控件使用，应用用户可以在运行时单独与其交互，但构成控件声明的属性和方法不会向使用者公开。例如，如果将 `TextBox` 和 `Button` 控件放在用户控件上，则按钮的 `click` 事件由用户控件在内部处理，而不是由放置用户控件的窗体处理。

将用户控件添加到项目

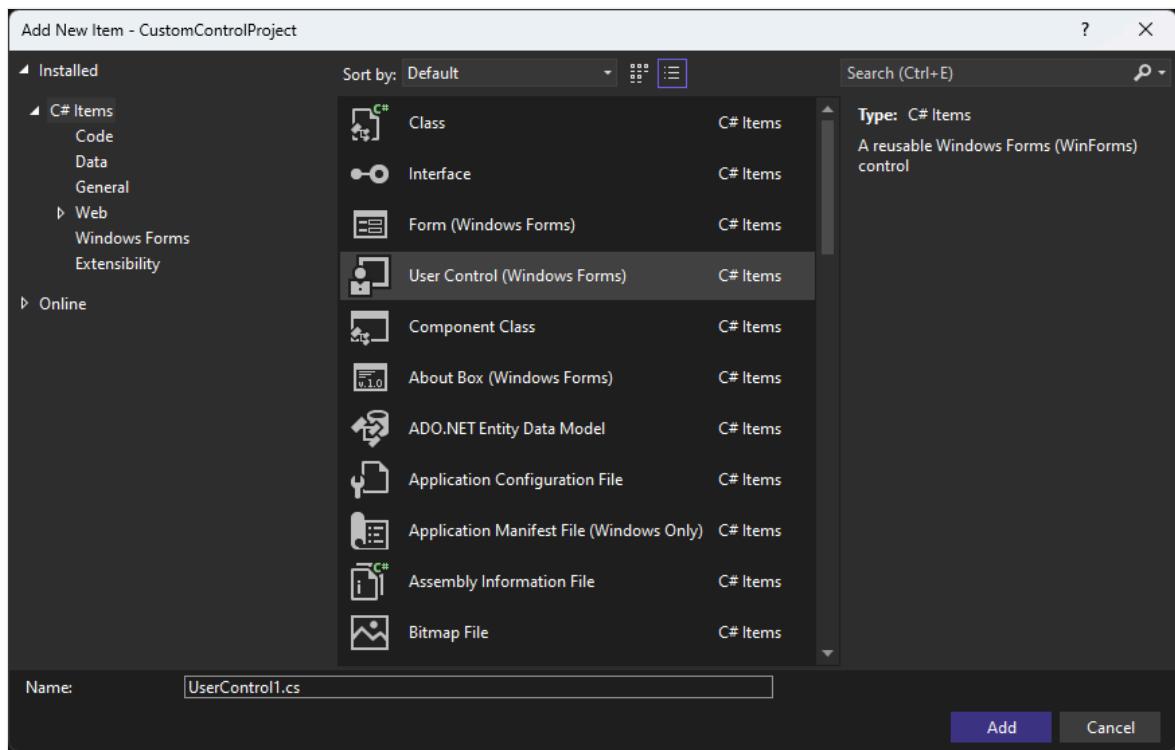
创建新项目后，使用 Visual Studio 模板创建用户控件。以下步骤演示如何将用户控件添加到项目：

1. 在 Visual Studio 中，找到 **项目资源管理器** 窗格。右键单击项目并选择“添加”>“用户控件(Windows 窗体)”。

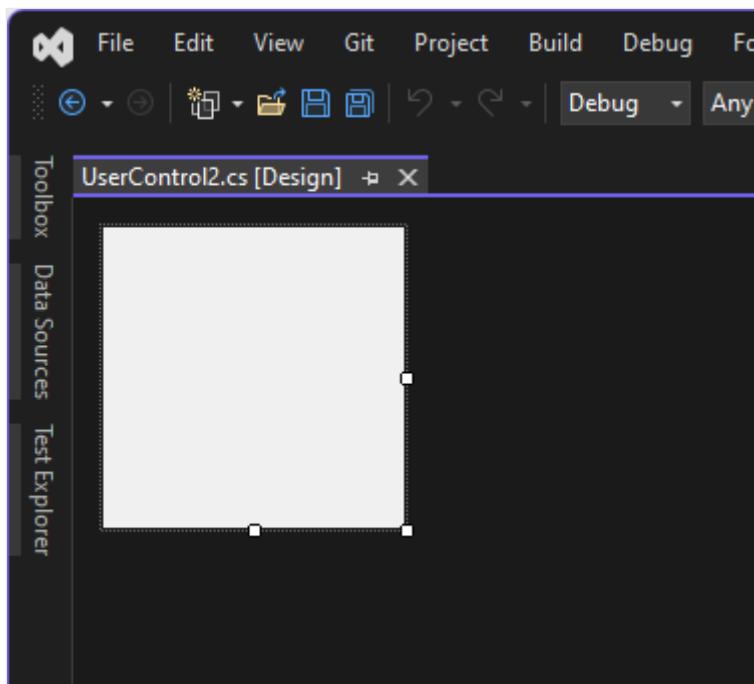


2. 在“名称”框中，键入用户控件的名称。Visual Studio 提供一个可以使用的默认和唯一名称。接下来，按 **添加**。

在 Visual Studio for Windows 窗体中添加项对话框



创建用户控件后，Visual Studio 将打开设计器：



有关工作用户控件的示例，请参阅 [如何创建用户控件](#)。

如何创建用户控件（Windows 窗体 .NET）

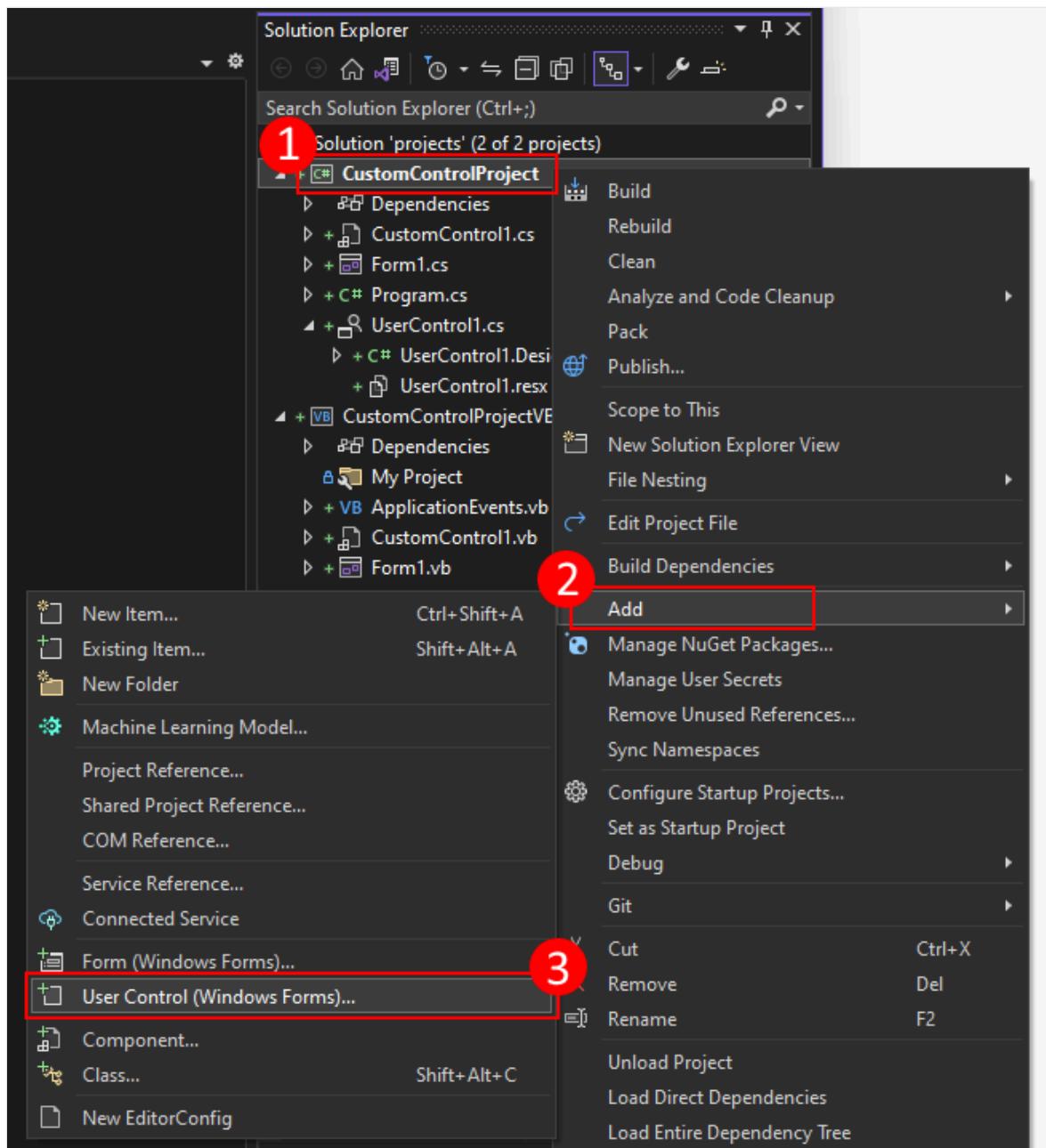
项目 • 2024/12/18

本文介绍如何向项目添加用户控件，然后将该用户控件添加到项目 form。 创建一个可重用的用户控件，并使其兼具视觉吸引力和功能性。 新控件对 [TextBox](#) 控件和 [Button](#) 控件进行分组。 当用户选择 button 该文本时，将清除文本框中的文本。 有关用户控件的详细信息，请参阅[用户控件概述](#)。

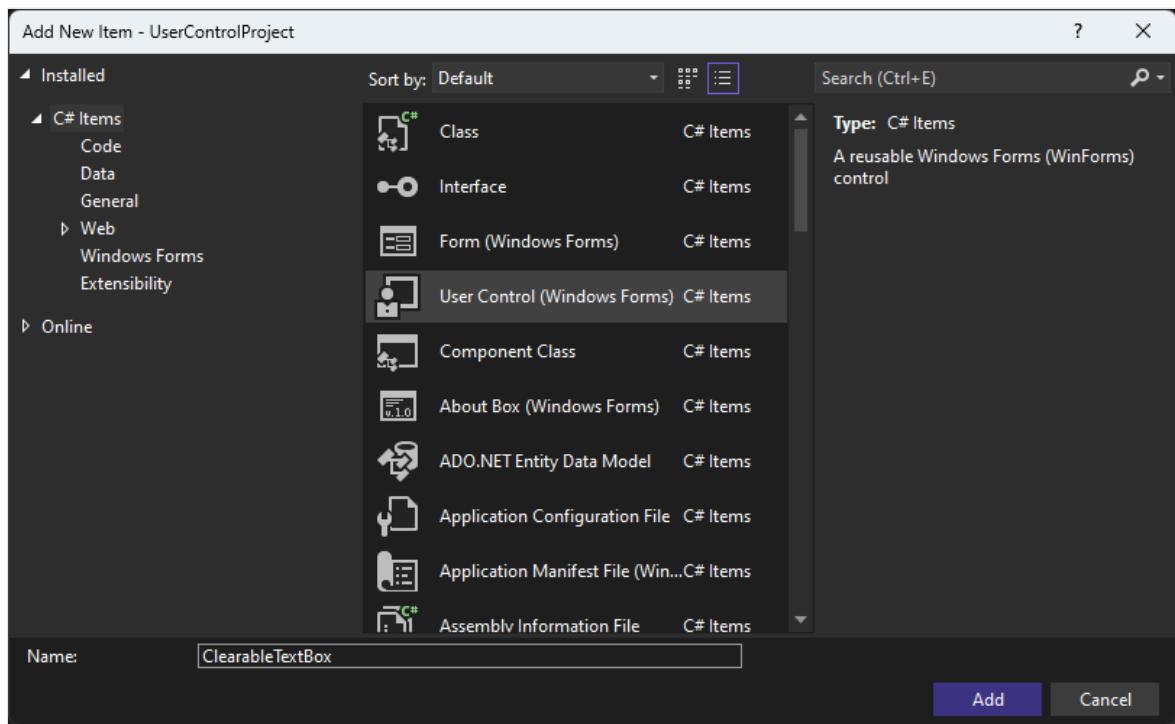
将用户控件添加到项目

在 Visual Studio 中打开 Windows 窗体项目后，使用 Visual Studio 模板创建用户控件：

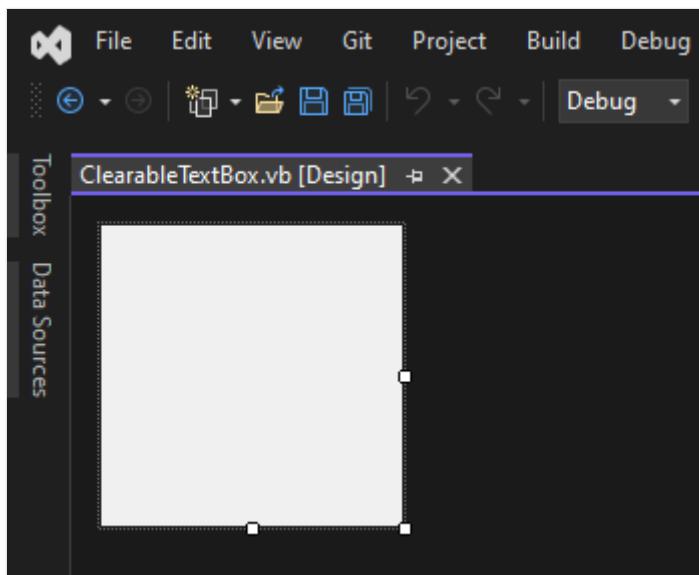
1. 在 Visual Studio 中，找到“项目资源管理器”窗口。 右键单击项目并选择“添加”>“用 户控件（Windows 窗体）”。



2. 将控件的名称设置为 ClearableTextBox，然后按 Add。



创建用户控件后，Visual Studio 将打开设计器：



设计可清除文本框

用户控件由构成控件组成，这些控件是在设计图面上创建的控件，就像设计方法一样。按照以下步骤添加和配置用户控件及其构成控件：

1. 打开设计器后，用户控件设计图面应为选定对象。如果不是，请单击设计图面以将其选中。在“属性”窗口中，设置下列属性：

展开表

属性	Value
MinimumSize	84, 53
大小	191, 53

2. 添加 Label 控件。 设置以下属性：

 展开表

属性	Value
名称	lblTitle
位置	3, 5

3. 添加 TextBox 控件。 设置以下属性：

 展开表

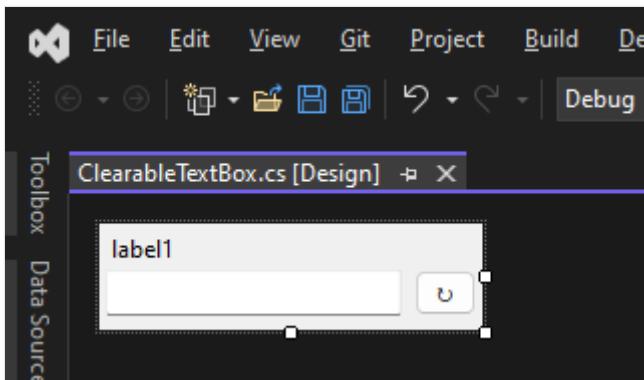
属性	Value
名称	txtValue
定位点	Top, Left, Right
位置	3, 23
大小	148, 23

4. 添加 Button 控件。 设置以下属性：

 展开表

属性	Value
名称	btnClear
定位点	Top, Right
位置	157, 23
大小	31, 23
文本	清除

此控件应如下所示：



5. 按 F7 打开 ClearableTextBox 类的代码编辑器。

6. 执行以下代码更改：

- 在代码文件顶部导入 System.ComponentModel 命名空间。
- 将 DefaultEvent 特性添加到该类中。此属性设置在设计器中双击控件时使用者生成的事件。使用者是声明和使用此控件的对象。有关属性的详细信息，请参阅[特性 \(C#\)](#) 或[特性概述 \(Visual Basic\)](#)。

```
C#  
  
using System.ComponentModel;  
  
namespace UserControlProject  
{  
    [DefaultEvent(nameof(TextChanged))]  
    public partial class ClearableTextBox : UserControl
```

c. 添加将 TextBox.TextChanged 事件转发给使用者的事件处理程序：

```
C#  
  
[Browsable(true)]  
public new event EventHandler? TextChanged  
{  
    add => txtValue.TextChanged += value;  
    remove => txtValue.TextChanged -= value;  
}
```

请注意，此事件在其上声明了 Browsable 属性。将 Browsable 应用于事件或属性后，当在设计器中选择该控件时，它可以控制该项在“属性”窗口中是否可见。在本例中，true 作为参数传递给指示事件应可见的属性。

d. 添加名为 Text 的字符串属性，以将 TextBox.Text 属性转发给使用者：

```
C#
```

```
[Browsable(true)]
public new string Text
{
    get => txtValue.Text;
    set => txtValue.Text = value;
}
```

- e. 添加名为 `Title` 的字符串属性，以将 `Label.Text` 属性转发给使用者：

```
C#
[Browsable(true)]
public string Title
{
    get => lblTitle.Text;
    set => lblTitle.Text = value;
}
```

7. 切换回 `ClearableTextBox` 设计器，然后双击 `btnClear` 控件以生成 `Click` 事件的处理程序。为处理程序添加以下代码，以清除 `txtValue` 文本框：

```
C#
private void btnClear_Click(object sender, EventArgs e) =>
    Text = "";
```

8. 最后，在解决方案资源管理器窗口中，右键单击项目并选择“生成”以生成项目。应不会有任何错误，生成完成后，控件 `ClearableTextBox` 会显示在工具箱中以供使用。

下一步是在一个 `form.` 中使用该控件。

示例应用程序

如果在最后一节中创建了一个新项目，则有一个名为 `Form` 的空白，否则请创建新的 `form` 项目。

1. 在解决方案资源管理器窗口中，双击 `form` 打开设计器。`form` 应选择“设计图面”。
2. 将 `form's Size` 属性设置为 `432, 315`。
3. 打开工具箱窗口，然后双击 `ClearableTextBox` 控件。此控件应列在以项目命名的部分下。
4. 再次双击 `ClearableTextBox` 控件以生成第二个控件。

5. 返回到设计器并将控件分开，以便完全显示它们。

6. 选择一个控件并设置以下属性：

 展开表

属性	Value
名称	ctlFirstName
位置	12, 12
大小	191, 53
游戏	First Name

7. 选择另一个控件并设置以下属性：

 展开表

属性	Value
名称	ctlLastName
位置	12, 71
大小	191, 53
游戏	Last Name

8. 返回**工具箱**窗口，向控件label添加form控件，并设置以下属性：

 展开表

属性	Value
名称	lblFullName
位置	12, 252

9. 接下来，需要为这两个用户控件生成事件处理程序。在设计器中，双击 `ctlFirstName` 控件。此操作将生成 `TextChanged` 事件的事件处理程序，并打开代码编辑器。

10. 切换回设计器并双击 `ctlLastName` 控件以生成第二个事件处理程序。

11. 交換回设计器，然后双击 form“标题栏”。此操作会生成 Load 事件的事件处理程序。
12. 在代码编辑器中，添加 UpdateNameLabel 方法。此方法将合并两个名称以创建一条消息，并将该消息分配给 lblFullName 控件。

```
C#  
  
private void UpdateNameLabel()  
{  
    if (string.IsNullOrWhiteSpace(ctlFirstName.Text) ||  
        string.IsNullOrWhiteSpace(ctlLastName.Text))  
        lblFullName.Text = "Please fill out both the first name and the  
        last name.";  
    else  
        lblFullName.Text = $"Hello {ctlFirstName.Text}  
        {ctlLastName.Text}, I hope you're having a good day.";  
}
```

13. 对于这两个 TextChanged 事件处理程序，请调用 UpdateNameLabel 方法：

```
C#  
  
private void ctlFirstName_TextChanged(object sender, EventArgs e) =>  
    UpdateNameLabel();  
  
private void ctlLastName_TextChanged(object sender, EventArgs e) =>  
    UpdateNameLabel();
```

14. 最后，从 UpdateNameLabel's form 事件调用 Load 该方法：

```
C#  
  
private void Form1_Load(object sender, EventArgs e) =>  
    UpdateNameLabel();
```

运行项目并输入名字和姓氏：

Form1

First Name

John

Last Name

Smith

Hello John Smith, I hope you're having a good day.

请尝试按 ↻ button 其中一个文本框重置。

Visual Studio 自定义控件的设计时支持 (Windows 窗体 .NET)

项目 • 2024/11/16

正如你在与 Windows 窗体设计器交互时注意到的，Windows 窗体控件提供了许多不同的设计时功能。Visual Studio Designer 提供的一些功能包括对齐线、操作项和属性网格。所有这些功能都提供了一种在设计时交互和自定义控件的简单方法。本文概述了可以添加到自定义控件的支持类型，以便更好地为控件使用者提供设计时体验。

与 .NET Framework 有何不同

自定义控件的许多基本设计元素在 .NET Framework 中保持不变。但是，如果使用更高级的设计器自定义功能，例如操作列表、类型转换器、自定义对话框，则可以处理一些独特的方案。

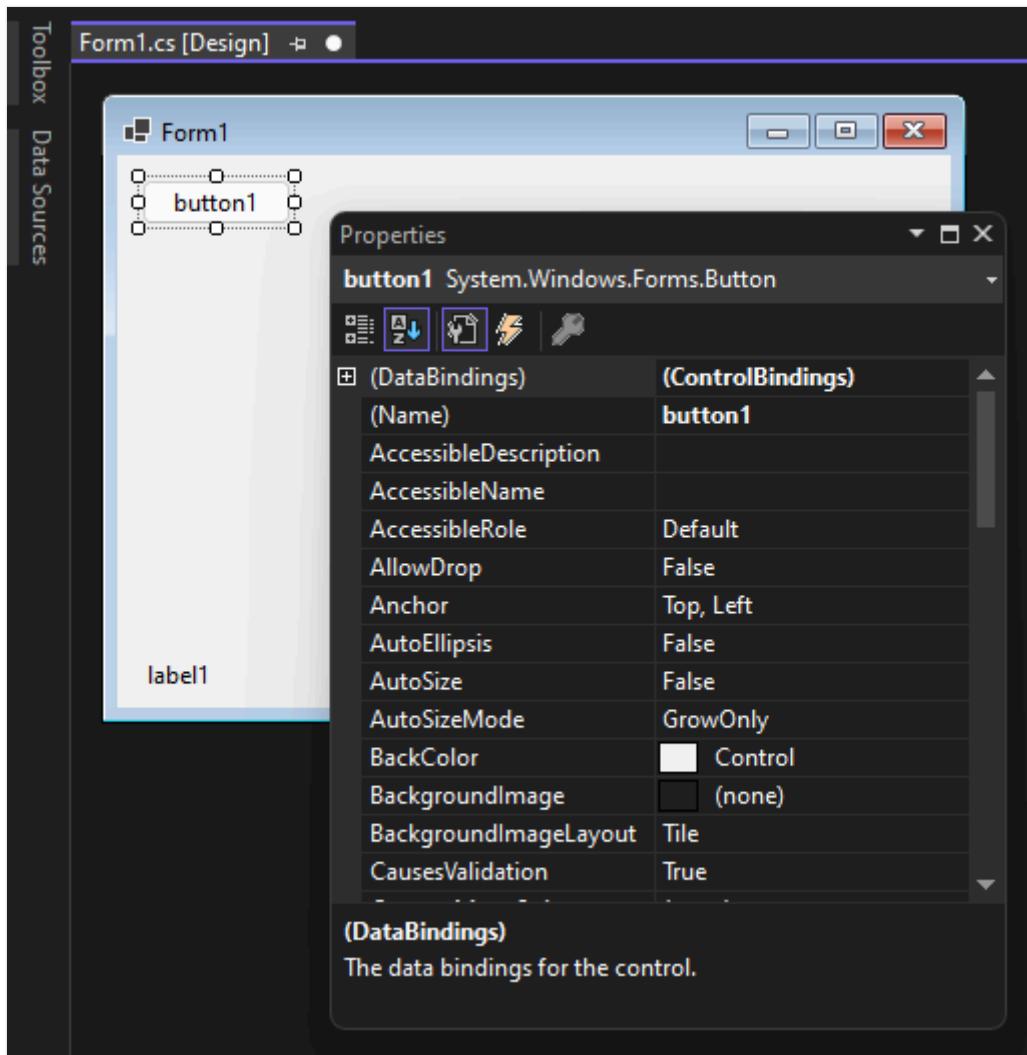
Visual Studio 是基于 .NET Framework 的应用程序，因此，针对 Windows 窗体看到的可视化设计器也基于 .NET Framework。使用 .NET Framework 项目时，Visual Studio 环境和设计 Windows 窗体应用在同一进程中运行，`devenv.exe`。使用 Windows 窗体 .NET（而不是 .NET Framework）应用时，这会带来问题。.NET 和 .NET Framework 不能在同一进程中运行。因此，Windows 窗体 .NET 使用不同的设计器“进程外”设计器。

进程外设计器是一个名为 `DesignToolsServer.exe` 的进程，在 Visual Studio 的 `devenv.exe` 进程中运行。`DesignToolsServer.exe` 进程在应用面向的 .NET 的同一版本和平台（如 .NET 9 和 x64）中运行。当自定义控件需要在 `devenv.exe` 中显示 UI 时，自定义控件必须实现客户端-服务器体系结构，以便与 `devenv.exe` 进行通信。有关详细信息，请参阅[.NET Framework \(Windows 窗体 .NET\) 以来的设计器更改](#)。

“属性”窗口

Visual Studio 属性窗口显示所选控件的属性和事件或 form。这通常是对自定义控件或组件执行的第一个自定义点。

下图显示了在视觉设计器中选择的 `Button` 控件，以及显示 button“属性”的属性网格：



可以控制有关自定义控件的信息在属性网格中的显示方式的一些方面。 特性将应用于自定义控件类或类属性。

类的属性

下表显示了可用于在设计时指定自定义控件和组件的行为的特性。

展开表

属性	说明
DefaultEventAttribute	指定组件的默认事件。
DefaultPropertyAttribute	指定组件的默认属性。
DesignerAttribute	指定用于为组件实现设计时服务的类。
DesignerCategoryAttribute	指定类设计器属于某一类别。
ToolboxItemAttribute	表示工具箱项的特性。
ToolboxItemFilterAttribute	指定要用于工具箱项的筛选器字符串和筛选器类型。

属性的特性

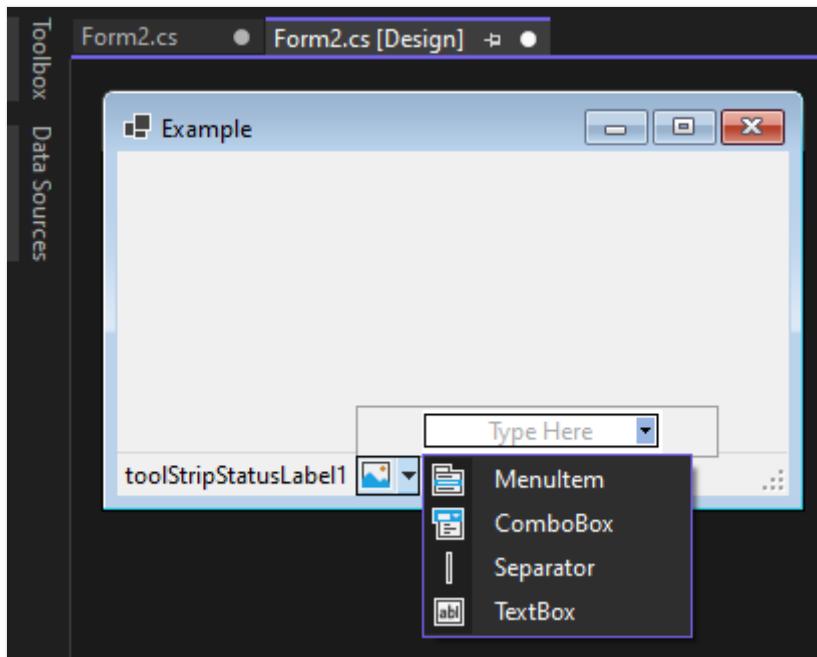
下表显示了可应用于自定义控件和组件的属性或其他成员的特性。

 展开表

属性	说明
AmbientValueAttribute	指定要传递给属性的值，以使该属性从另一个源中获取其值。 这称为“环境”。
BrowsableAttribute	指定属性或事件是否应在“属性”窗口中显示。
CategoryAttribute	指定当属性或事件显示在一个设置为 Categorized 模式的 PropertyGrid 控件中时，用于对属性或事件分组的类别的名称。
DefaultValueAttribute	指定属性的默认值。
DescriptionAttribute	指定属性或事件的说明。
DisplayNameAttribute	指定不返回值且不采用任何自变量的属性、事件或公共方法的显示名称。
EditorAttribute	指定用于更改属性的编辑器。
EditorBrowsableAttribute	指定可在编辑器中查看的属性或方法。
HelpKeywordAttribute	为类或成员指定上下文关键字。
LocalizableAttribute	指定是否应本地化某一属性。
PasswordPropertyTextAttribute	指示对象的文本表示形式由星号等字符隐藏。
ReadOnlyAttribute	指定此特性绑定到的属性在设计时是只读还是可读/写。
RefreshPropertiesAttribute	指示关联的属性值更改时应刷新属性网格。
TypeConverterAttribute	指定对于此属性绑定到的对象要使用哪种类型作为转换器。

自定义控件设计器

可以通过创作关联的自定义设计器来增强自定义控件的设计时体验。 默认情况下，自定义控件显示在主机的设计图面上，看起来与运行时一样。 使用自定义设计器，可以增强控件的设计时视图、添加操作项、对齐线和其他项，从而帮助用户确定如何布局和配置控件。 例如，在设计时，[ToolStrip](#) 设计器会为用户添加额外控件以添加、删除和配置各个项目，如下图所示：



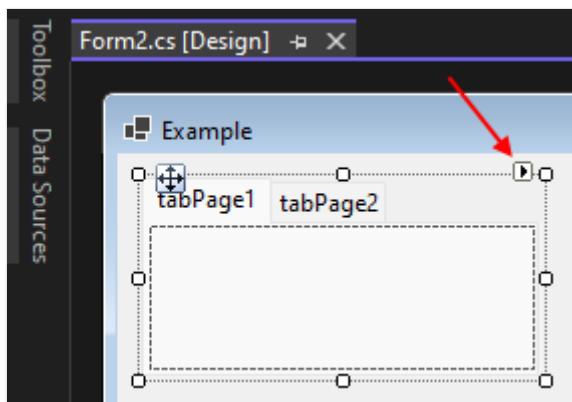
可以通过执行以下步骤来创建自己的自定义设计器：

1. 添加对 [Microsoft.WinForms.Designer.SDK NuGet 包](#) 的引用。
2. 创建从 `Microsoft.DotNet.DesignTools.Designers.ControlDesigner` 类继承的类型。
3. 在用户控件类中，使用 `System.ComponentModel.DesignerAttribute` 类特性标记该类，并传递在上一步中创建的类型。

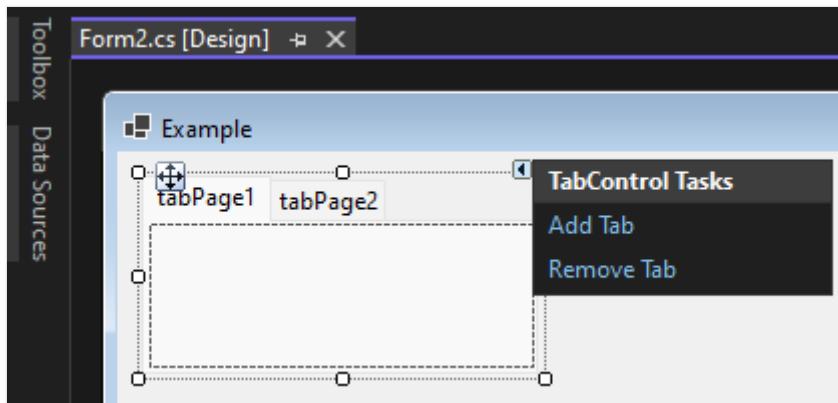
有关详细信息，请参阅[与 .NET Framework 有何不同部分](#)。

操作项

设计器操作是上下文相关的菜单，允许用户快速执行常见任务。例如，如果将 a 添加到 a `TabControl` form，你将在控件中添加和删除选项卡。选项卡在“属性”窗口中通过显示选项卡集合编辑器的 `TabPage` 属性进行管理。除了强制用户始终筛选属性列表来查找 `TabPage` 属性，而是 `TabControl` 提供仅在选择控件时可见的智能标记button，如下图所示：



选择智能标记后，将显示操作列表：



通过添加“添加选项卡”和“删除选项卡”操作，控件的设计器可使其快速添加或删除选项卡。

创建操作项列表

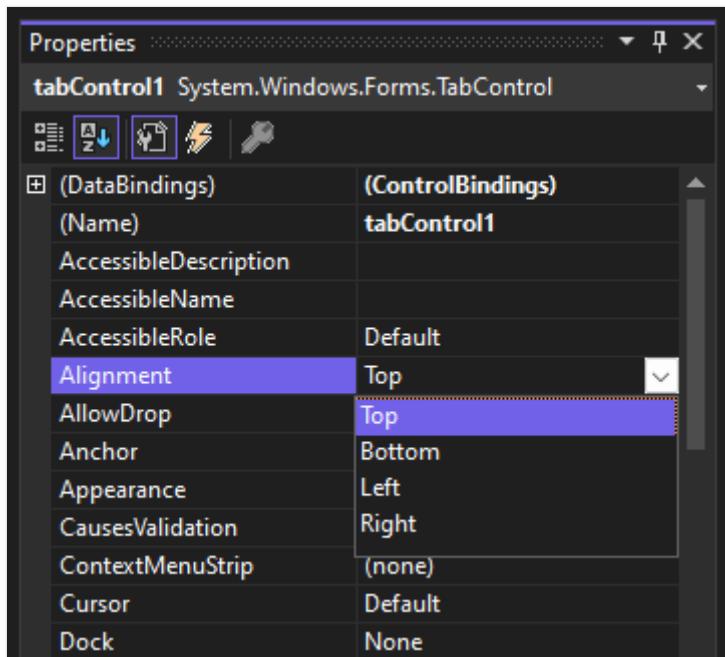
操作项列表由你创建的 `ControlDesigner` 类型提供。以下步骤是创建自己的操作列表的基本指南：

1. 添加对 [Microsoft.WinForms.Designer.SDK NuGet 包](#) 的引用。
2. 创建继承自 `Microsoft.DotNet.DesignTools.Designers.Actions.DesignerActionList` 的新操作列表类。
3. 将属性添加到希望用户访问的操作列表。例如，将 `bool` 或 `Boolean` (在 Visual Basic 中) 属性添加到类会在操作列表中创建 `CheckBox` 控件。
4. 按照[自定义控件设计器](#)部分中的步骤创建新设计器。
5. 在设计器类中，重写 `ActionLists` 属性，其将返回`Microsoft.DotNet.DesignTools.Designers.Actions.DesignerActionListCollection`类型。
6. 将操作列表添加到 `DesignerActionListCollection` 实例，并返回该列表。

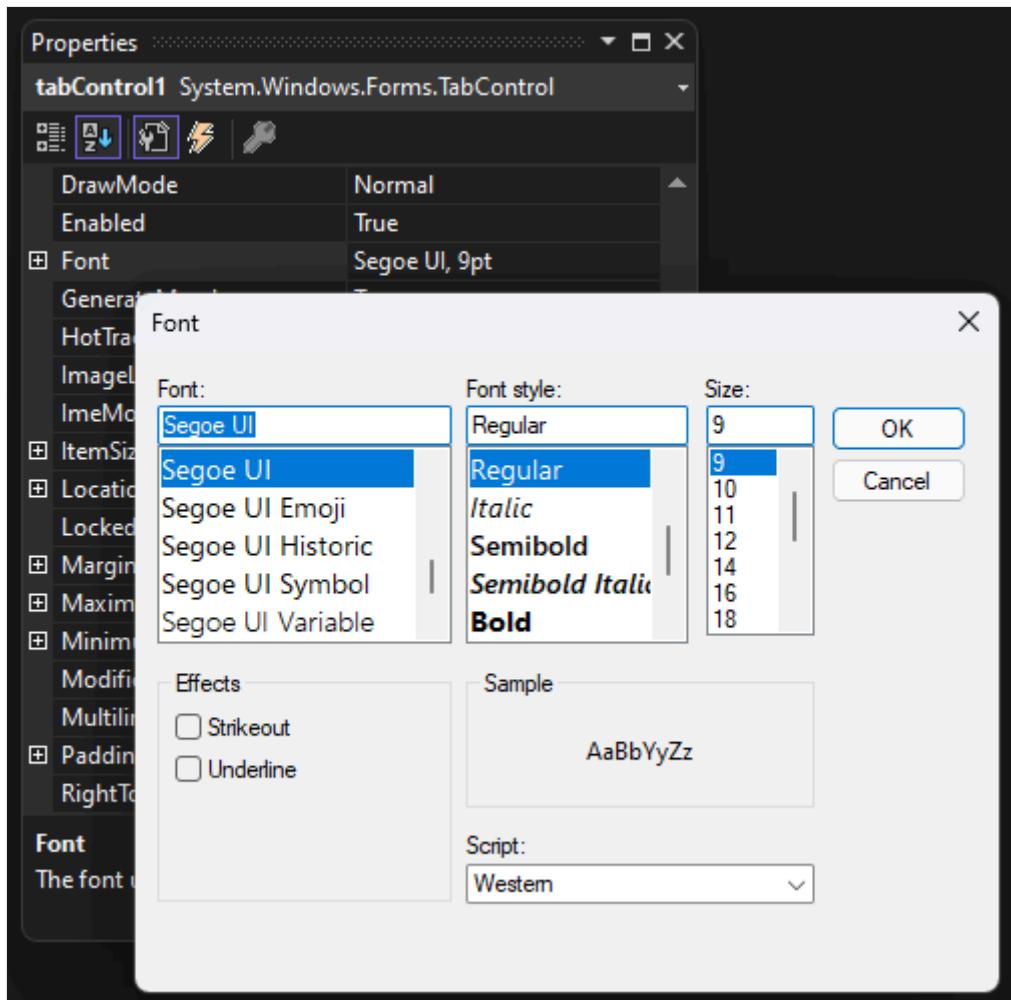
有关操作列表的示例，请参阅[Windows 窗体设计器扩展性文档](#)和[示例 GitHub 存储库](#)，特别是[TileRepeater.Designer.Server/ControlDesigner](#) 文件夹。

模式对话框类型编辑器

在“属性”窗口中，大多数属性都可以轻松地在网格中编辑，例如当属性的后备类型是枚举、布尔或数字时。



有时，属性更为复杂，需要一个自定义对话框，用户可以使用该对话框来更改属性。例如，[Font](#) 属性是 [System.Drawing.Font](#) 类型，其中包含许多更改字体外观的属性。这在“属性”窗口中不容易显示，因此此属性使用自定义对话框编辑字体：



如果自定义控件属性使用的是由 Windows 窗体 提供的内置类型编辑器，则可以使用 [EditorAttribute](#) Visual Studio 要使用的相应 .NET Framework 编辑器标记属性。通过使用内置编辑器，可以避免复制进程外设计器提供的代理-对象客户端-服务器通信的要求。

引用内置类型编辑器时，请使用 .NET Framework 类型，而不是 .NET 类型：

C#

```
[Editor("System.Windows.Forms.Design.FileNameEditor, System.Design,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a",
"System.Drawing.Design.UITypeEditor, System.Drawing,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")]
public string? Filename { get; set; }
```

设计器自 .NET Framework (Windows 窗体 .NET) 以来发生更改

项目 · 2024/11/16

适用于 .NET 的 Windows 窗体的可视化设计器自 .NET Framework 以来进行了一些改进和更改。这些更改主要影响自定义控件设计器。本文介绍与 .NET Framework 的主要区别。

Visual Studio 是基于 .NET Framework 的应用程序，因此，针对 Windows 窗体看到的可视化设计器也基于 .NET Framework。使用 .NET Framework 项目，Visual Studio 环境和正在设计的 Windows 窗体应用在同一进程中运行：devenv.exe。使用 Windows 窗体 .NET（而不是 .NET Framework）应用时，这会带来问题。.NET 和 .NET Framework 代码不能在同一进程中运行。因此，Windows 窗体 .NET 使用不同的设计器“进程外”设计器。

进程外设计器

进程外设计器是一个名为 DesignToolsServer.exe 的进程，在 Visual Studio 的 devenv.exe 进程中运行。DesignToolsServer.exe 进程在应用已设置为目标的 .NET 版本和平台上运行，例如 .NET 9 和 x64。

在 Visual Studio 设计器中，为设计器上的每个组件和控件创建 .NET Framework 代理对象，并使这些对象与 DesignToolsServer.exe 设计器中项目的真实 .NET 对象进行通信。

控件设计器

对于 .NET，控件设计器需要使用 [NuGet](#) 上提供的 `Microsoft.WinForms.Designer.SDK` API 进行编码。此库是对 .NET 的 .NET Framework 设计器的重构。所有设计器类型都已移动到不同的命名空间，但类型名称大多相同。要更新 .NET 的 .NET Framework 设计器，必须稍微调整命名空间。

- 设计器类和其他相关类型（例如 `ControlDesigner` 和 `ComponentTray`）已从 `System.Windows.Forms.Design` 命名空间移动到 `Microsoft.DotNet.DesignTools.Designers` 命名空间。
- `System.ComponentModel.Design` 命名空间中的操作列表相关类型已移动到 `Microsoft.DotNet.DesignTools.Designers.Actions` 命名空间。
- `System.Windows.Forms.Design.Behavior` 命名空间中与行为相关的类型（如装饰器和对齐线）已移动到 `Microsoft.DotNet.DesignTools.Designers.Behaviors` 命名空间。

自定义类型编辑器

自定义类型编辑器比控件设计器复杂得多。由于 Visual Studio 进程基于 .NET Framework，因此 Visual Studio 上下文中显示的任何 UI 也必须基于 .NET Framework。例如，在创建显示通过单击 `button` 属性网格中调用的自定义类型编辑器的 .NET 控件时，此设计会产生问题。此对话框无法在 Visual Studio 的上下文中显示。

进程外设计器可处理大多数控件设计器功能，例如装饰器、内置类型编辑器和自定义绘图。每当需要显示自定义模式对话框（例如显示新类型编辑器）时，你都需要复制进程外设计器提供的代理与对象之间以及客户端与服务器之间的通信。这会比旧的 .NET Framework 系统产生更多的开销。

如果自定义控件属性使用由 Windows 窗体 提供的类型编辑器，则可以使用 [EditorAttribute](#) 希望 Visual Studio 使用的相应 .NET Framework 编辑器来标记属性。通过使用内置编辑器，可以避免复制进程外设计器提供的代理与对象之间以及客户端与服务器之间的通信。

```
C#  
  
[Editor("System.Windows.Forms.Design.FileNameEditor, System.Design,  
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a",  
"System.Drawing.Design.UITypeEditor, System.Drawing,  
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a")]  
public string? Filename { get; set; }
```

创建类型编辑器

要创建自定义设计器来提供类型编辑器，需要各种项目，如以下列表所述：

- `Control`：此项目是包含控件代码的自定义控件库。这是用户在想要使用你的时将引用的库。
- `Control.Client`：包含自定义设计器 UI 对话框的 .NET Framework 项目的 Windows 窗体。
- `Control.Server`：包含控件的自定义设计器代码的 .NET 项目的 Windows 窗体。
- `Control.Protocol`：一个 .NET Standard 项目，其中包含 `Control.Client` 和 `Control.Server` 项目使用的通信类。
- `Control.Package`：包含所有其他项目的 NuGet 包项目。此包采用一种允许 Visual Studio Windows 窗体用于 .NET 工具主机和使用控件库和设计器的方式进行格式化。

即使类型编辑器派生自现有编辑器（例如 [ColorEditor](#) 或 [FileNameEditor](#)），也仍然必须创建代理与对象以及客户端与服务器之间的通信，因为你提供了一种新的 UI 类型以在

Visual Studio 上下文中显示。但是，在 Visual Studio 中实现该类型编辑器的代码要简单得多。

① 重要

详细描述此方案的文档正在编写中。在该文档发布之前，请使用以下博客文章和示例来指导你创建、发布和使用此项目结构：

- [博客：WinForm 进程外设计器的自定义控件 ↗](#)
- [TileRepeater 控件示例 ↗](#)

自定义控件的设计时属性 (Windows 窗体 .NET)

项目 · 2024/12/19

本文介绍如何在 Visual Studio 的 Windows 窗体可视化设计器中为控件处理属性。

每个控件从基类继承多个属性 [System.Windows.Forms.Control](#)，例如：

- [Enabled](#)
- [Font](#)
- [ForeColor](#)
- [Focused](#)
- [Visible](#)
- [Width](#)

创建控件时，可以定义新属性并控制它们在设计器中的显示方式。

定义属性

具有由控件定义的 **获取** 访问器的任何公共属性都会自动显示在 Visual Studio **属性** 窗口中。如果该属性还定义了 **集** 访问器，则可以在 **属性** 窗口中修改该属性。但是，可以通过应用 [BrowsableAttribute](#)，在 **属性** 窗口中显式显示或隐藏属性。此属性使用一个布尔参数来指示是否进行显示。有关属性的详细信息，请参阅 [属性 \(C#\)](#) 或 [属性概述 \(Visual Basic\)](#)。

```
C#  
[Browsable(false)]  
public bool IsSelected { get; set; }
```

[注意]不能隐式转换为或从字符串转换的复杂属性需要类型转换器。

序列化属性

控件上设置的属性将被序列化到设计器的后台代码文件中。当属性的值设置为默认值以外的其他值时，将发生这种情况。

当设计器检测到对属性的更改时，它将计算控件的所有属性，并序列化其值与该属性的默认值不匹配的任何属性。属性的值序列化到设计器的后台代码文件中。默认值可帮助设计器确定应序列化哪些属性值。

默认值

当属性应用 `DefaultValueAttribute` 属性或属性的类包含特定于属性的 `Reset` 和 `ShouldSerialize` 方法时，该属性被视为具有默认值。有关属性的详细信息，请参阅 [属性 \(C#\)](#) 或 [属性概述 \(Visual Basic\)](#)。

通过设置默认值，可以启用以下内容：

- 如果此属性已从其默认值修改，则在 **属性** 窗口中会提供视觉提示。
- 用户可以右键单击该属性并选择 **重置** 将属性还原为其默认值。
- 设计器生成更高效的代码。

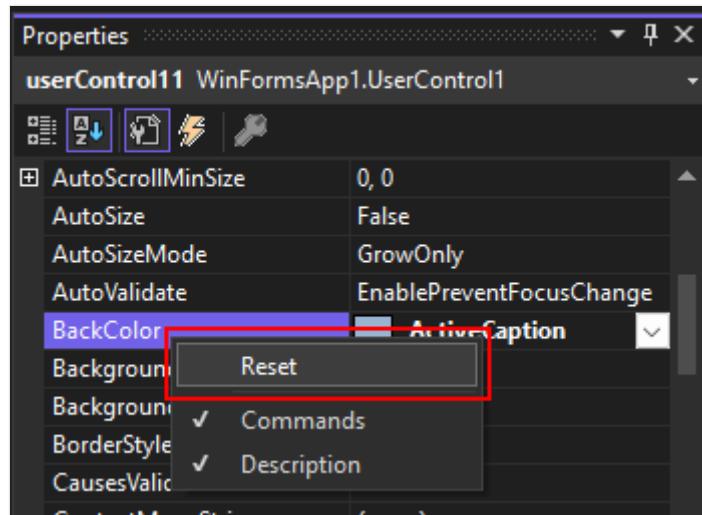
如果属性使用简单类型（如基元类型），则可以通过将 `DefaultValueAttribute` 应用于属性来设置默认值。但是，具有此属性的属性不会自动以该赋值开头。必须将属性的后盾字段设置为相同的默认值。可以在声明或类的构造函数中设置属性。

如果属性是复杂类型，或者想要控制设计器的重置和序列化行为，请在类上定义 `Reset<PropertyName>` 和 `ShouldSerialize<PropertyName>` 方法。例如，如果控件定义了 `Age` 属性，则方法命名为 `ResetAge` 和 `ShouldSerializeAge`。

① 重要

将 `DefaultValueAttribute` 应用于属性，或同时提供 `Reset<PropertyName>` 和 `ShouldSerialize<PropertyName>` 方法。不要同时使用这两者。

可以通过 **属性** 窗口右键单击属性名称，然后选择 **重置**，将属性重置为默认值。



在以下情况下启用 **属性**>**右键单击菜单选项**>**重置** 的可用性：

- 该属性应用了 `DefaultValueAttribute` 属性，并且该属性的值与属性的值不匹配。

- 该属性的类定义了一个没有 `ShouldSerialize<PropertyName>` 的 `Reset<PropertyName>` 方法。
- 该属性的类定义 `Reset<PropertyName>` 方法，`ShouldSerialize<PropertyName>` 返回 `true`。

默认值属性

如果属性的值与 `DefaultValueAttribute` 提供的值不匹配，则此属性被视为已更改，可通过 **属性** 窗口重置。

① 重要

此属性不应用于具有相应 `Reset<PropertyName>` 和 `ShouldSerialize<PropertyName>` 方法的属性。

以下代码声明两个属性：默认值为 `North` 的枚举，以及默认值为 10 的整数。

C#

```
[DefaultValue(typeof(Directions), "North")]
public Directions PointerDirection { get; set; } = Directions.North;

[DefaultValue(10)]
public int DistanceInFeet { get; set; } = 10;
```

重置和 `ShouldSerialize`

如前所述，`Reset<PropertyName>` 方法和 `ShouldSerialize<PropertyName>` 方法不仅提供了指导属性重置行为的机会，还可用于判断值是否已更改且是否应序列化为设计器的后台代码文件。这两种方法需要协同工作，不应该单独定义其中一种。

① 重要

不应为具有 `DefaultValueAttribute` 的属性创建 `Reset<PropertyName>` 和 `ShouldSerialize<PropertyName>` 方法。

当定义 `Reset<PropertyName>` 时，**属性** 窗口会显示该属性的 **重置** 上下文菜单选项。选择 **重置** 时，将调用 `Reset<PropertyName>` 方法。**重置** 上下文菜单选项的启用或禁用取决于 `ShouldSerialize<PropertyName>` 方法返回的内容。当 `ShouldSerialize<PropertyName>` 返回 `true` 时，这表明属性已不再是默认值，应序列化到代码隐藏文件中，并启用 **重置** 上

下文菜单选项。当返回 `false` 时，重置的上下文菜单选项将被禁用，并且背后代码的属性设置代码已被移除。

💡 提示

这两种方法都可以且应该使用专用作用域进行定义，以便它们不会构成控件的公共 API。

以下代码片段声明名为 `Direction` 的属性。此属性的设计器行为由 `ResetDirection` 和 `ShouldSerializeDirection` 方法控制。

C#

```
public Directions Direction { get; set; } = Directions.None;

private void ResetDirection() =>
    Direction = Directions.None;

private bool ShouldSerializeDirection() =>
    Direction != Directions.None;
```

类型转换器

虽然类型转换器通常将一种类型转换为另一种类型，但它们也为属性网格和其他设计时控件提供字符串到值转换。字符串到值转换允许在这些设计时控件中表示复杂属性。

大多数内置数据类型（数字、枚举和其他类型）都有提供字符串到值转换和执行验证检查的默认类型转换器。默认类型转换器位于 `System.ComponentModel` 命名空间中，以转换的类型命名。转换器类型名称使用以下格式：`{type name}Converter`。例如，`StringConverter`、`TimeSpanConverter` 和 `Int32Converter`。

类型转换器在设计阶段与**属性窗口** 广泛使用。可以使用 `TypeConverterAttribute` 将类型转换器应用于属性或类型。

属性 窗口使用转换器在属性上声明 `TypeConverterAttribute` 时将属性显示为字符串值。在类型上声明 `TypeConverterAttribute` 时，**属性** 窗口将使用转换器来处理该类型的每个属性。类型转换器还有助于将属性值序列化到设计器代码隐藏文件中。

类型编辑器

当属性的类型为内置类型或已知类型时，**属性** 窗口会自动使用属性的类型编辑器。例如，布尔值以带有 `True` 和 `False` 值的组合框进行编辑，`DateTime` 类型使用日历下拉菜

单。

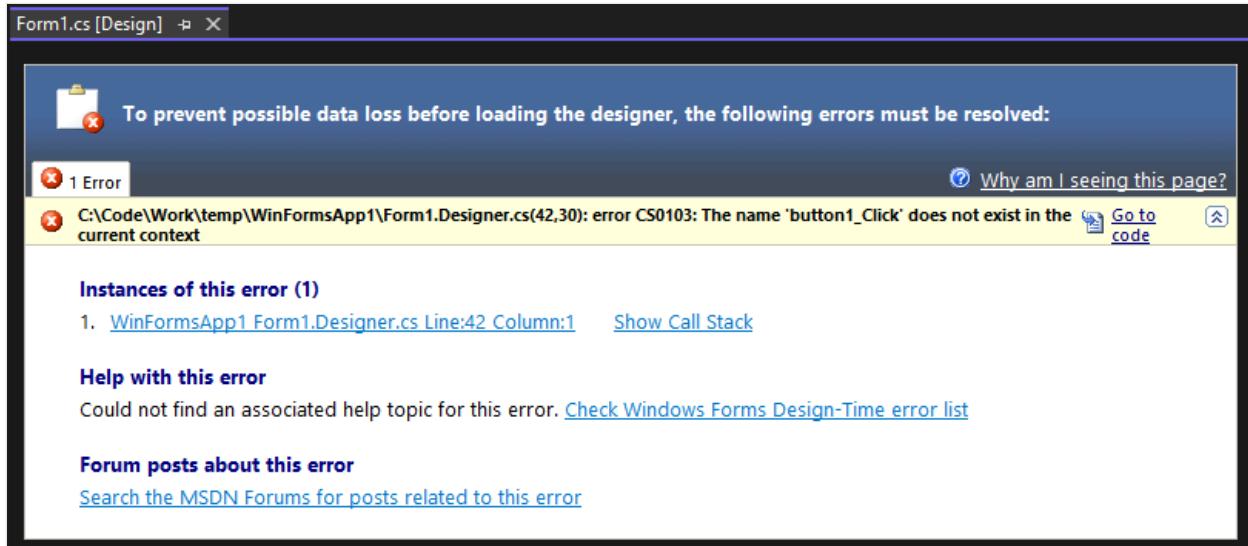
① **重要**

自 .NET Framework 以来，自定义类型编辑器已更改。有关详细信息，请参阅 [.NET Framework \(Windows 窗体 .NET\) 以来的设计器更改](#)。

Windows 窗体设计器错误页 (Windows 窗体 .NET)

项目 • 2024/11/05

如果 Windows 窗体设计器由于代码、第三方组件或其他位置的错误而未能加载，将显示错误页而不是设计器。此错误页不一定表示设计器中的 bug。bug 可能位于代码隐藏文件中的某个位置。错误显示在可折叠的黄色栏中，其中带有用于跳转到代码页上错误位置的链接。



错误窗口

错误窗口由不同的部分组成。

- 黄色栏

将为每个错误创建黄色可折叠栏，并按说明进行分组。该栏描述阻止设计器加载属性的编译器错误。其中包含以下详细信息：

- 错误所在的文件。
- 文件中出错的列和行。
- 错误代码。
- 错误的描述。
- 直接导航到错误的链接。

- 此错误的实例

黄色错误栏展开后，会列出错误的每个实例。许多错误类型采用以下格式包含确切位置：<项目名称> <窗体名称> 行:<行号> 列:<列号>。如果调用堆栈与错误关

联，则可选择“**显示调用堆栈**”链接以进行查看。检查调用堆栈可以进一步帮助你解决这个错误。

① 重要

错误的元素可能因所使用的代码语言而异。

- 有关此错误的帮助

如果提供了针对错误的帮助文章，请选择“**MSDN 帮助**”链接，直接导航到帮助页。

- 有关此错误的论坛帖子

选择“**在 MSDN 论坛中搜索与此错误相关的帖子**”链接，以导航到旧的 Microsoft 开发人员网络论坛。可以在 [Microsoft Q&A](#) 或 [StackOverflow](#) 论坛上搜索或提问。

首先要采取的措施

通常可以通过清理和重新生成项目或解决方案来清除错误。

1. 找到“解决方案资源管理器”窗口。
2. 右键单击解决方案或项目，然后选择“清理”。
3. 右键单击解决方案或项目，然后选择“重新生成”。

还可以尝试从项目文件夹中删除“bin”和“obj”文件夹。这可能会清除临时文件或导致发生“还原”操作，从而修复错误依赖项。

使用以下部分对常见的设计时错误进行会审。

常见设计时错误

此部分列出了可能会遇到的一些错误。

- 当前上下文中不存在名称“<名称>”
- “<标识符名称>”不是有效标识符
- “<名称>”已存在于“<项目名称>”中
- “<工具箱选项卡名称>”不是工具箱类别
- 请求的语言分析程序未安装
- 缺少生成和分析源代码所需的服务
- 尝试创建“<对象名称>”的实例时发生异常
- 另一个编辑器以不兼容的模式打开了“<文档名称>”
- 另一个编辑器对“<文档名称>”进行了更改

- 另一个编辑器以不兼容的模式打开了此文件
- 数组设置级别“<数组中的设置级别>”过高
- 无法打开程序集“<程序集名称>”
- 错误的元素类型。此序列化程序要求元素类型为“<类型名称>”
- 此时无法访问 Visual Studio 工具箱
- “<事件名称>”事件是只读的，因此无法将事件处理程序绑定到该事件
- 请求的组件不是设计容器的成员，因此无法创建该组件的方法名称
- 对象“<名称>”已命名为“<名称>”，因此无法命名该对象
- 无法移除或损坏继承的组件“<组件名称>”
- 类别“<工具箱选项卡名称>”没有类“<类名>”的工具
- 类“<类名>”没有匹配的构造函数
- 属性“<属性名称>”的代码生成失败
- 组件“<组件名称>”未在其构造函数中调用 Container.Add
- 组件名称不能为空
- 未能访问变量“<变量名称>”，因为它尚未初始化
- 找不到类型“<类型名称>”
- 无法加载类型“<类型名称>”
- 未能找到继承的组件的项目项模板
- 委托类“<类名>”没有 invoke 方法。此类是否为委托
- 成员“<成员名称>”的重复声明
- 从区域性“<区域性名称>”的资源文件中读取资源时出错
- 从默认区域性“<区域性名称>”的资源文件中读取资源时出错
- 未能分析方法“<方法名称>”
- 无效的组件名称：“<组件名称>”
- 类型“<类名>”由同一文件中的几个分部类构成
- 未能找到程序集“<程序集名称>”
- 程序集名称“<程序集名称>”无效
- 无法设计基类“<类名>”
- 不能在此版本的 Visual Studio 中设计类“<类名>”
- 不能在此版本的 Visual Studio 中设计类“<类名>”
- 该类名不是此语言的有效标识符
- 组件包含对“<引用名称>”的循环引用，因此无法添加该组件
- 此时无法修改设计器
- 文件中的类都不能进行设计，因此未能为该文件显示设计器
- 未安装基类“<类名>”的设计器
- 设计器必须创建类型“<类型名称>”的实例，但该类型已声明为抽象，因此设计器无法创建该类型的实例
- 未能在设计器中加载该文件
- 该文件的语言不支持必需的代码分析和生成服务
- 语言分析程序类“<类名>”没有正确实现
- 名称“<名称>”已由另一个对象使用

- 对象“<对象名称>”没有实现 IComponent 接口
- 对象“<对象名称>”为属性“<属性名称>”返回了 null，而这是不允许的
- 序列化数据对象的类型不正确
- 需要服务“<服务名称>”，但未能找到它
- 服务实例必须从“<接口名称>”派生或实现它
- 未能修改代码窗口中的文本
- 工具箱枚举数对象仅支持一次检索一个项
- 未能从工具箱中检索到“<组件名称>”的工具箱项
- 未能从工具箱中检索到“<工具箱项名称>”的工具箱项
- 未能找到类型“<类型名称>”
- 只能从主应用程序线程调用类型解析服务
- 变量“<变量名称>”未声明或从未赋值
- 菜单命令“<菜单命令名称>”已经有一个命令处理程序
- 已有一个名为“<组件名称>”的组件
- 已有一个工具箱项创建者注册了格式“<格式名称>”
- 此语言引擎不支持用于加载设计器的 CodeModel
- 类型“<类型名称>”不具有带有“<参数类型名称>”类型参数的构造函数
- 无法添加对当前应用程序的引用“<引用名称>”
- 无法签出当前文件
- 无法找到名为“<选项对话框选项卡名称>”的页
- 无法在页“<选项对话框选项卡名称>”上找到属性“<属性名称>”
- 该文件内的类不是从可进行可视化设计的类继承，因此 Visual Studio 无法为该文件打开设计器
- Visual Studio 无法保存或加载类型“<类型名称>”的实例
- Visual Studio 无法在设计视图中打开“<文档名称>”
- Visual Studio 未能找到用于“<类型名称>”类型的类的设计器

当前上下文中不存在名称“<名称>”

最常见的情况是，当删除或重命名设计器文件引用的代码隐藏文件中的事件处理程序时，你会看到此错误。打开 `<form>.designer.<language>` 代码文件并从窗体或控件中删除事件处理程序。

<标识符名称> 不是有效标识符

此错误指示字段、方法、事件或对象未正确命名。

“<名称>”已存在于“<项目名称>”中

你为项目中已存在的继承窗体指定了名称。 若要更正此错误，请为继承窗体提供唯一名称。

“<工具箱选项卡名称>”不是工具箱类别

第三方设计器尝试访问工具箱上不存在的选项卡。 请联系组件供应商。

请求的语言分析程序未安装

Visual Studio 尝试加载为文件类型注册的设计器，但无法加载。 这很可能是由于安装过程中发生了错误。 请联系你用于修补程序的语言的供应商。

缺少生成和分析源代码所需的服务

此错误是第三方组件的问题。 请联系组件供应商。

尝试创建“<对象名称>”的实例时发生异常

第三方设计器请求 Visual Studio 创建对象，但对象引发错误。 请联系组件供应商。

另一个编辑器以不兼容的模式打开了“<文档名称>”

如果尝试打开已在另一个编辑器中打开的文件，则会出现此错误。 会显示已打开文件的编辑器。 若要更正此错误，请关闭打开文件的编辑器，然后重试。

另一个编辑器对“<文档名称>”进行了更改

关闭设计器，然后重新打开才能使更改生效。 通常，Visual Studio 会在进行更改后自动重新加载设计器。 但是，其他设计器（如第三方组件设计器）可能不支持重新加载行为。 在这种情况下，Visual Studio 会提示你手动关闭并重新打开设计器。

另一个编辑器以不兼容的模式打开了此文件

此消息类似于“另一个编辑器以不兼容的模式打开了‘<文档名称>’”，但 Visual Studio 无法确定文件名。 若要更正此错误，请关闭打开文件的编辑器，然后重试。

数组秩“<数组中的秩>”过高

在设计器分析的代码块中，Visual Studio 仅支持单维数组。 多维数组在此区域之外有效。

无法打开程序集“<程序集名称>”

尝试打开无法打开的文件时，会出现此错误消息。请验证该文件存在并且是有效程序集。

错误的元素类型。此序列化程序要求元素类型为“<类型名称>”

此错误是第三方组件的问题。请联系组件供应商。

此时无法访问 Visual Studio 工具箱

Visual Studio 在工具箱不可用时对其进行了调用。如果看到此错误，请使用[报告问题](#)来记录问题。

“<事件名称>”事件是只读的，因此无法将事件处理程序绑定到该事件

尝试将事件连接到从基类继承的控件时，通常会出现此错误。如果控件的成员变量是私有变量，则 Visual Studio 无法将事件连接到方法。私有继承控件不能绑定额外的事件。

请求的组件不是设计容器的成员，因此无法创建该组件的方法名称

Visual Studio 尝试将事件处理程序添加到设计器中没有成员变量的组件。请联系组件供应商。

对象“<名称>”已命名为“<名称>”，因此无法命名该对象

此错误是 Visual Studio 序列化程序中的内部错误。它指示序列化程序尝试对一个对象命名两次，此操作不受支持。如果看到此错误，请使用[报告问题](#)来记录问题。

无法移除或损坏继承的组件“<组件名称>”

继承的控件由其继承类所拥有。必须在控件的起源类中更改继承的控件。因而无法重命名或销毁它。

类别“<工具箱选项卡名称>”没有类“<>类名”的工具

设计器尝试引用特定工具箱选项卡上的类，但该类不存在。请联系组件供应商。

类“<类名>”没有匹配的构造函数

第三方设计器要求 Visual Studio 在不存在的构造函数中创建具有特定参数的对象。请联系组件供应商。

属性“<属性名称>”的代码生成失败

此错误是错误的泛型包装器。此消息附带的错误字符串会提供有关错误消息的更多详细信息，并提供指向更具体帮助文章的链接。若要更正此错误，请解决追加到此错误的错误消息中指定的错误。

组件“<组件名称>”未在其构造函数中调用 `container.Add()`

此消息与窗体上加载或放置的组件中的错误相关。它指示组件未将自己添加到其容器控件（无论是其他控件还是窗体）。设计器会继续工作，但组件在运行时可能会出现问题。

若要更正错误，请联系组件供应商。或者，如果是创建的组件，请在组件构造函数中调用 `IContainer.Add` 方法。

组件名称不能为空

尝试将组件重命名为空值时，会出现此错误。

未能访问变量“<变量名称>”，因为它尚未初始化

此错误可能是由于两种情形导致的。第三方组件供应商分发的控件或组件有问题，或者你编写的代码在组件之间具有递归依赖项。

若要更正此错误，请确保代码没有递归依赖项。如果不存在此类问题，请记下错误消息的确切文本并联系组件供应商。

找不到类型“<类型名称>”

错误消息：“找不到类型‘<类型名称>’。请确保已引用包含此类型的程序集。如果此类型为开发项目的一部分，请确保已成功生成该项目。”

发生此错误是因为找不到引用。请确保引用错误消息中指示的类型，并且还引用了该类型所需的所有程序集。通常，问题是解决方案中的控件未生成。若要生成，请从“生成”菜单中选择“生成解决方案”。否则，如果控件已生成，请从解决方案资源管理器中“引用”或“依赖项”文件夹的右键单击菜单中手动添加引用。

无法加载类型“<类型名称>”

Visual Studio 尝试关联事件处理方法，但找不到该方法的一个或多个参数类型。此错误通常是由于缺少引用导致的。若要更正此错误，请将包含类型的引用添加到项目，然后重试。

找不到继承的组件的项目项模板

Visual Studio 中继承表单的模板不可用。如果看到此错误，请使用[报告问题](#)来记录问题。

委托类“<类名>”没有 invoke 方法。此类是否为委托

Visual Studio 尝试创建事件处理程序，但事件类型有问题。如果事件是通过不符合 CLS 的语言所创建，则可能会发生此错误。请联系组件供应商。

成员“<成员名称>”的重复声明

出现此错误的原因是成员变量已声明两次（例如，在代码中声明了两个名为 `Button1` 的控件）。名称必须在继承表单间唯一。此外，名称不能仅通过大小写来区分。

从区域性“<区域性名称>”的资源文件中读取资源时出错

如果项目中存在错误的 .resx 文件，则可能会发生此错误。

若要更正该错误，请执行以下操作：

1. 选择解决方案资源管理器中的“**显示所有文件**”按钮以查看与解决方案关联的 .resx 文件。
2. 通过右键单击 .resx 文件并选择“打开”，在 XML 编辑器中加载 .resx 文件。
3. 手动编辑 .resx 文件以解决错误。

从默认区域性“<区域性名称>”的资源文件中读取资源时出错

如果对于默认区域性，项目中存在错误的 .resx 文件，则可能会发生此错误。

若要更正该错误，请执行以下操作：

1. 选择解决方案资源管理器中的“**显示所有文件**”按钮以查看与解决方案关联的 .resx 文件。
2. 通过右键单击 .resx 文件并选择“打开”，在 XML 编辑器中加载 .resx 文件。
3. 手动编辑 .resx 文件以解决错误。

未能分析方法“<方法名称>”

错误消息：“未能分析方法‘<方法名称>’。 分析器报告以下错误：‘<错误字符串>’。 请查看任务列表以了解潜在的错误。”

这是针对分析过程中出现的问题的常规错误消息。 这些错误通常是由于语法错误导致的。 有关与错误相关的特定消息，请参阅任务列表。

无效的组件名称：“<组件名称>”

你尝试将组件重命名为对该语言无效的值。 若要更正此错误，请命名组件，使其符合该语言的命名规则。

类型“<类名>”由同一文件中的几个分部类构成

使用 [partial](#) 关键字在多个文件中定义类时，在每个文件中只能有一个分部定义。

若要更正此错误，请从文件中移除类的所有分部定义（只保留一个）。

未能找到程序集“<程序集名称>”

此错误类似于“找不到类型‘<类型名称>’”，但发生此错误通常是由于元数据属性。 若要更正此错误，请检查是否引用了属性使用的所有程序集。

程序集名称“<程序集名称>”无效

组件请求了特定程序集，但组件提供的名称不是有效程序集名称。 请联系组件供应商。

无法设计基类“<类名>”

Visual Studio 加载了类，但无法设计类，因为类的实现者未提供设计器。 如果类支持设计器，请确保不存在会导致设计器中的显示问题的问题（例如编译器错误）。 此外，请

确保对类的所有引用都正确且所有类名都拼写正确。否则，如果类不可设计，请在代码视图中进行编辑。

未能加载基类“<类名>”

项目中未引用类，因此 Visual Studio 无法加载它。若要更正此错误，请在项目中添加对类的引用，然后关闭并重新打开 Windows 窗体设计器窗口。

不能在此版本的 Visual Studio 中设计“<类名>”类

此控件或组件的设计器不支持与 Visual Studio 相同的类型。请联系组件供应商。

该类名不是此语言的有效标识符

由用户创建的源代码具有对所用语言无效的类名。若要更正此错误，请命名类，使其符合语言要求。

组件包含对“<引用名称>”的循环引用，因此无法添加该组件

无法将控件或组件添加到自身。可能出现此问题的另一种情况是，如果窗体（例如 `Form1`）的 `InitializeComponent` 方法中有代码创建 `Form1` 的另一个实例。

此时无法修改设计器

当编辑器中的文件标记为只读时，会发生此错误。确保文件未标记为只读且应用程序未在运行。

由于文件中的类都不能进行设计，因此无法为该文件显示设计器

当 Visual Studio 找不到满足设计器要求的基类时，会发生此错误。窗体和控件必须派生自支持设计器的基类。如果要从继承窗体或控件派生，请确保已生成项目。

未安装基类“<类名>”的设计器

Visual Studio 无法加载类的设计器。如果看到此错误，请使用[报告问题](#)来记录问题。

设计器必须创建类型“<类型名称>”的实例，但该类型已声明为抽象，因此设计器无法创建该类型的实例

发生此错误是因为传递给设计器的对象基类是不允许的[抽象类](#)。

无法在设计器中加载该文件

此文件的基类不支持任何设计器。 解决方法是使用代码视图处理文件。 在解决方案资源管理器中右键单击文件，然后选择“查看代码”。

该文件的语言不支持必需的代码分析和生成服务

错误消息：“该文件的语言不支持必需的代码分析和生成服务。 确保你正在打开的文件是项目的成员，然后尝试重新打开该文件。”

此错误很可能是由于打开不支持设计器的项目中的文件所导致的。

语言分析器类“<类名>”没有正确实现

错误消息：“语言分析器类‘<类名>’没有正确实现。 请和供应商联系以获得更新的分析器模块。”

所使用的语言注册了不是从正确基类派生的设计器类。 请联系你使用的语言的供应商。

名称“<名称>”已由另一个对象使用

这是 Visual Studio 序列化程序中的内部错误。 如果看到此错误，请使用[报告问题](#)来记录问题。

对象“<对象名称>”没有实现 [IComponent](#) 接口

Visual Studio 尝试创建组件，但创建的对象未实现 [IComponent](#) 接口。 请联系组件供应商以获取修补程序。

对象“<对象名称>”为属性“<属性名称>”返回了 null，而这是不允许的

某些 .NET 属性应始终返回对象。 例如，窗体的控件集合应始终返回对象，即使其中没有控件也是如此。

若要更正此错误，请确保错误中指定的属性不为 null。

序列化数据对象的类型不正确

序列化程序提供的数据对象不是与所使用的当前序列化程序匹配的类型实例。请联系组件供应商。

需要服务“<服务名称>”，但未能找到它

Visual Studio 所需的服务不可用。如果尝试加载不支持该设计器的项目，请改为使用代码编辑器进行更改。否则，如果看到此错误，请使用[报告问题](#)来记录问题。

服务实例必须从“<接口名称>”派生或实现它

此错误指示组件或组件设计器调用了 AddService 方法，该方法需要接口和对象，但指定的对象未实现指定接口。请联系组件供应商。

未能修改代码窗口中的文本

如果 Visual Studio 由于磁盘空间或内存问题而无法编辑文件，或者文件标记为只读，则会发生此错误。

工具箱枚举数对象仅支持一次检索一个项

如果看到此错误，请使用[报告问题](#)来记录问题。

未能从工具箱中检索到“<组件名称>”的工具箱项

相关组件在 Visual Studio 访问它时引发了异常。请联系组件供应商。

未能从工具箱中检索到“<工具箱项名称>”的工具箱项

如果工具箱项中的数据损坏或组件版本已更改，则会发生此错误。请尝试从工具箱中移除该项，然后再将其添加回工具箱。

未能找到类型“<类型名称>”

当加载设计器时，Visual Studio 未能找到类型。请确保已引用包含该类型的程序集。如果该程序集是当前开发项目的一部分，请确保已生成了该项目。

只能从主应用程序线程调用类型解析服务

Visual Studio 尝试从错误的线程访问所需资源。当用于创建设计器的代码从主应用程序线程以外的线程调用类型解析服务时，会显示此错误。若要更正此错误，请从正确的线

程调用服务或联系组件供应商。

变量“<变量名称>”未声明或从未赋值

源代码引用了未声明或赋值的变量（如 Button1）。如果变量未赋值，则此消息显示为警告，而不是错误。

菜单命令“<菜单命令名称>”已经有一个命令处理程序

如果第三方设计器将已有处理程序的命令添加到命令表中，则会出现此错误。请联系组件供应商。

已有一个名为“<组件名称>”的组件

错误消息：“已有一个名为’<组件名称>’的组件。组件的名称必须是唯一的，而且名称必须不区分大小写。名称也不能与继承的类中的任何组件名称冲突。”

当在属性窗口中更改了组件的名称时，会出现此错误消息。若要更正此错误，请确保所有组件名称都是唯一的，不区分大小写，并且不会与继承类中任何组件的名称冲突。

已有一个工具箱项创建者注册了格式“<格式名称>”

第三方组件对工具箱选项卡上的项进行了回调，但该项已包含回调。请联系组件供应商。

此语言引擎不支持用于加载设计器的 CodeModel

此消息类似于“该文件的语言不支持必需的代码分析和生成服务”，但此消息涉及内部注册问题。如果看到此错误，请使用[报告问题](#)来记录问题。

类型“<类型名称>”不具有带有“<参数类型名称>”类型参数的构造函数

Visual Studio 找不到具有匹配参数的[构造函数](#)。此错误可能是由于为构造函数提供的类型不是所需类型所导致的。例如，Point 构造函数可能采用两个整数。如果提供了 float 类型，则会引发此错误。

若要更正此错误，请使用其他构造函数，或是显式强制转换参数类型，使其与构造函数提供的类型匹配。

无法添加对当前应用程序的引用”<引用名称>”

Visual Studio 无法添加引用。 若要更正此错误，请检查是否尚未引用该引用的不同版本。

无法签出当前文件

将当前签入的文件更改为源代码管理时，会出现此错误。通常，Visual Studio 会提供文件签出对话框，以便用户可以签出文件。这次文件未签出，可能是由于签出期间发生合并冲突。若要更正此错误，请确保文件未锁定，然后尝试手动签出文件。

无法找到名为”<选项对话框选项卡名称>”的页

当组件设计器使用不存在的名称请求访问“选项”对话框中的页时，会出现此错误。请联系组件供应商。

无法在页”<选项对话框选项卡名称>”上找到属性”<属性名称>”

当组件设计器请求访问“选项”对话框中某页上的特定值，但该值不存在时，会出现此错误。请联系组件供应商。

由于该文件内的类不是从可进行可视化设计的类继承，因此 Visual Studio 无法为该文件打开设计器

Visual Studio 加载了类，但未能加载该类的设计器。Visual Studio 要求设计器使用文件中的第一个类。若要更正此错误，请移动类代码以使其成为文件中的第一个类，然后重新加载该设计器。

Visual Studio 无法保存或加载类型”<类型名称>”的实例

这是第三方组件的问题。请联系组件供应商。

Visual Studio 无法在设计视图中打开”<文档名称>”

此错误指示项目的语言不支持设计器，会在你尝试在“打开文件”对话框中或从解决方案资源管理器打开文件时出现。请改为在代码视图中编辑文件。

Visual Studio 未能找到用于“<类型名称>”类型的类的设计器

Visual Studio 加载了类，但无法设计该类。 请改为通过右键单击类并选择“查看代码”，在代码视图中编辑类。

32 位问题排查 (Windows Forms .NET)

项目 · 2024/12/18

升级到 Visual Studio 2022 后，您可能会遇到应用程序设计时体验无法运行的问题。这可能与引用 32 位组件相关。Visual Studio 2022 是一个 64 位进程，无论基础技术（如 .NET Framework、.NET 或 COM\ActiveX）如何，都无法加载 32 位组件。在尝试升级 Visual Studio 之前，你可能没有意识到你在使用 32 位组件。编译为 64 位或目标 AnyCPU 的引用将继续工作。如果您引用的组件编译成 AnyCPU，但碰巧引用了 32 位的内容，您同样会碰到问题。

怎么了

Windows 窗体代码以两种模式运行：设计时和运行时。在运行时，你以编译的任何模式运行：32 位或 64 位，.NET Framework 或 .NET。在设计时，代码在 Visual Studio 中运行，这是一个 64 位 .NET Framework 进程。如果项目代码与该环境不匹配，则无法在设计器中运行。例如，如果项目面向 32 位 .NET Framework 或 64 位 .NET，则它与 Visual Studio 的 64 位 .NET Framework 进程不匹配。问题在于：Visual Studio 中的 Windows 窗体设计器无法直接实例化 32 位组件或 .NET 组件，它只能实例化 64 位 .NET Framework 组件。为了解决这些集成问题，Windows 窗体团队为 Visual Studio 创建了 **外部进程设计器**，作为 Windows 窗体设计器的翻译层。进程外设计器代表代码与 Visual Studio 64 位 .NET Framework 设计器通信，以便可以将设计器与 .NET 项目配合使用。

早期版本的 Visual Studio 面向 32 位，你的项目可能编译为 AnyCPU，以便在设计模式下选择 32 位，以匹配 Visual Studio。32 位特定引用能够正常工作，但如果是 64 位特定引用，可能会在使用设计器时遇到问题。使用 Visual Studio 2022 时，问题已逆转。Visual Studio 2022 仅在 64 位中可用。编译为 AnyCPU 的组件和库在 32 位和 64 位中都正常工作，并且 Visual Studio 2022 64 位中没有问题。但是，升级到 Visual Studio 2022 后，如果项目依赖于 32 位特定组件，则项目在设计时可能无法运行。即使在为 AnyCPU 编译引用的组件时也是如此，但碰巧直接引用 32 位组件或 32 位 COM\ActiveX 库。

总之，Visual Studio 2022 中的 Windows 窗体设计器无法使用 32 位组件，这是一个 64 位应用。**进程外设计器** 是为 .NET 应用的 Windows 窗体在设计时提供帮助而创建的，适用于 32 位和 64 位系统。此设计器现在有助于加载 32 位和 64 位 .NET Framework 组件。

你能做什么

你应该考虑一些设计更改，这有助于你的项目。

- 从 .NET Framework 升级到 .NET 8+。

.NET 使用进程外设计器，这有助于解决 32 位设计器问题。

- 使用 .NET Framework，将应用设置为目标 AnyCPU。

如果您将目标设为 AnyCPU 并启用 Prefer 32-bit，那么在 Visual Studio 设计时，您的应用程序会以 64 位模式运行，但在运行时编译为 32 位。

- 为 AnyCPU 或 64 位重新编译 32 位组件。

如果有权访问 32 位组件的源代码，请尝试将其编译为 AnyCPU 或 64 位，并引用该新版本。

- 查找 64 位备用组件。

如果使用的是其他人拥有的组件，请检查它们是否提供 64 位版本，并引用该版本。

- 试用进程外设计器。

最后一个选项是为 .NET Framework 启用进程外设计器。

进程外设计器

如果项目面向 .NET，则已使用进程外设计器。但是，如果仍在使用 .NET Framework，则需要启用进程外设计器。

⚠ 警告

由于相同的体系结构差异，更新后的进程外 32 位 .NET Framework 设计器无法与旧的进程内 .NET Framework 设计器完全一致。高度自定义的控件设计器不兼容。如果从第三方使用自定义控件库，请检查它们是否提供支持进程外 .NET Framework 设计器的版本。

进程外设计器，存在一些限制，可处理 Visual Studio 2022 的 32 位问题：

- .NET Framework 受益于改进的类型解析。
- .NET Framework 和 .NET 都支持 ActiveX 和 COM 引用。
- Visual Studio 中的进程内设计器检测到 32 位程序集加载失败，并可以建议启用进程外设计器。

使用进程外设计器

支持 32 位引用需要 Visual Studio 17.9 或更高版本。通过将以下 `<PropertyGroup>` 设置添加到项目文件来启用它：

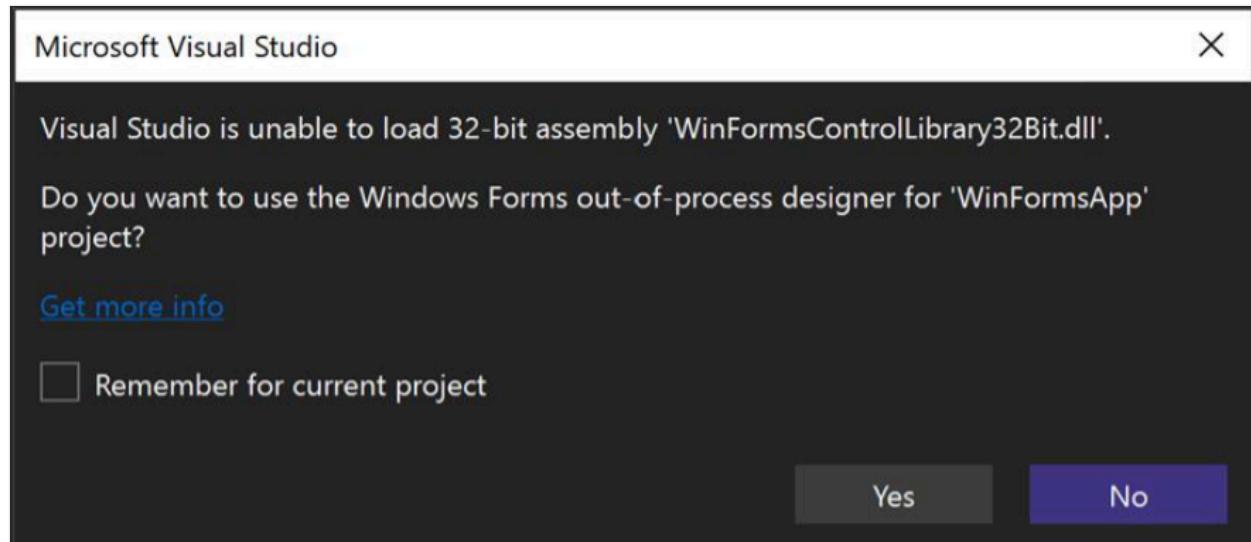
XML

```
<PropertyGroup>
    <UseWinFormsOutOfProcDesigner>True</UseWinFormsOutOfProcDesigner>
</PropertyGroup>
```

修改项目文件后，重新加载项目。

32 位问题检测

目前，当 Visual Studio 检测到 32 位引用无法加载时，它会提示启用 Windows 窗体进程外设计器。如果同意启用它，则会为你更新项目，然后重新加载。



此检测功能控制在 Visual Studio 菜单中的“**工具>选项**”>**预览功能**下。

另请参阅

- .NET 博客 - 32 位 .NET Framework 项目的 WinForms 设计器选择 ↴

使用 Reset 和 ShouldSerialize 进行属性控制 (Windows Forms .NET)

项目 • 2024/12/18

本文介绍如何在 Visual Studio 中创建 `Reset<PropertyName>` 和 `ShouldSerialize<PropertyName>` 方法来管理 **属性** 窗口的属性。如果属性没有简单的默认值，`Reset` 和 `ShouldSerialize` 是可以为属性提供的可选方法。如果该属性具有简单的默认值，则应应用 `DefaultValueAttribute`，并改为向属性类构造函数提供默认值。这两种机制都支持设计器中的以下功能：

- 如果属性已从其默认值修改，则此属性在属性浏览器中提供视觉指示。
- 用户可以右键单击该属性并选择 **重置** 将属性还原为其默认值。
- 设计器生成更高效的代码。

有关属性的详细信息，请参阅 [Reset 和 ShouldSerialize](#)。

辅助代码

本文通过创建指南针玫瑰控件来演示 `Reset` 和 `ShouldSerialize` 方法。如果使用的是自己的用户控件，则可以跳过本部分。

- 将以下枚举添加到代码：

```
C#  
  
public enum Directions  
{  
    None,  
    North,  
    NorthEast,  
    East,  
    SouthEast,  
    South,  
    SouthWest,  
    West,  
    NorthWest,  
}
```

- 添加名为 `CompassRose` 的用户控件。
- 向用户控件添加一个名为 `Direction` 且类型为 `Directions` 的属性。

```
public Directions Direction { get; set; } = Directions.None;
```

重置

`Reset<PropertyName>` 方法将相应的 `<PropertyName>` 属性重置为其默认值。

以下代码将 `Direction` 属性重置为 `None`，该属性被视为指南针玫瑰控件的默认值：

C#

```
private void ResetDirection() =>
    Direction = Directions.None;
```

ShouldSerialize

`ShouldSerialize<PropertyName>` 方法返回一个布尔值，该值指示后备属性是否已从其默认值更改，并且应该序列化到设计器代码中。

当 `Direction` 属性不等于 `None` 时，以下代码返回 `true`，指示已选择方向：

C#

```
private bool ShouldSerializeDirection() =>
    Direction != Directions.None;
```

例

以下代码显示了 `Direction` 属性的 `Reset` 和 `ShouldSerialize` 方法：

C#

```
public partial class CompassRose : UserControl
{
    public Directions Direction { get; set; } = Directions.None;

    public CompassRose() =>
        InitializeComponent();

    private void ResetDirection() =>
        Direction = Directions.None;

    private bool ShouldSerializeDirection() =>
```

```
        Direction != Directions.None;  
    }
```

在工具箱中设置控件的图标 (Windows 窗体 .NET)

项目 · 2024/11/05

你创建的控件始终会收到 Visual Studio 中工具箱窗口的通用图标。但是，当你更改图标时，它会为控件增添专业感，并使其在工具箱中脱颖而出。本文介绍如何设置控件的图标。

位图图标

Visual Studio 中工具箱窗口的图标必须符合某些标准，否则它们将被忽略或错误显示。

- 大小：控件的图标必须是 16x16 位图图像。
- 文件类型：图标可以是位图 (.bmp) 或 Windows 图标 (.ico) 文件。
- 透明度：洋红色 (RGB: 255, 0, 255, Hex: 0xFF00FF) 呈现透明。
- 主题：Visual Studio 有多个主题，但每个主题要么是深色，要么是浅色。你的图标应设计为浅色主题。当 Visual Studio 使用深色主题时，图标中的深色和浅色将自动反转。

如何分配图标

图标分配给具有 `ToolboxBitmapAttribute` 特性的控件。有关属性的详细信息，请参阅[特性 \(C#\)](#) 或[特性概述 \(Visual Basic\)](#)。

💡 提示

可以从 [GitHub](#) 下载示例图标。

该特性设置在控件的类上，它具有三个不同的构造函数：

- `ToolboxBitmapAttribute(Type)` - 此构造函数接收单个类型引用，并会从该类型尝试查找用作图标的嵌入资源。

该类型的 `FullName` 用于在该类型的程序集中查找嵌入资源，格式如下：`{project-name}.{namespace-path}.{type-name}{.bmp|.ico}`。例如，如果引用了 `MyProject.MyNamespace.CompassRose` 类型，则特性将查找名为 `MyProject.MyNamespace.CompassRose.bmp` 或 `MyProject.MyNamespace.CompassRose.ico` 的嵌入资源。

C#

```
// Looks for a CompassRose.bmp or CompassRose.ico embedded resource in
// the
// same namespace as the CompassRose type.
[ToolboxBitmap(typeof(CompassRose))]
public partial class CompassRose : UserControl
{
    // Code for the control
}
```

- [ToolboxBitmapAttribute\(Type, String\)](#) - 此构造函数接收两个参数。第一个参数是一种类型，第二个参数是该类型的程序集中[嵌入资源](#)的命名空间和名称。

C#

```
// Loads the icon from the WinFormsApp1.Resources.CompassRose.bmp
resource
// in the assembly containing the type CompassRose
[ToolboxBitmap(typeof(CompassRose),
"WinFormsApp1.Resources.CompassRose.bmp")]
public partial class CompassRose : UserControl
{
    // Code for the control
}
```

- [ToolboxBitmapAttribute\(String\)](#) - 此构造函数接收单个字符串参数，即图标文件的绝对路径。

C#

```
// Loads the icon from a file on disk
[ToolboxBitmap(@"C:\Files\Resources\MyIcon.bmp")]
public partial class CompassRose : UserControl
{
    // Code for the control
}
```

使用键盘的概述 (Windows 窗体 .NET)

项目 • 2024/11/05

在 Windows 窗体中，用户输入以 [Windows 消息](#)的形式发送到应用程序。一系列可重写的方法在应用程序、窗体和控件级别处理这些消息。当这些方法收到键盘消息时，它们会引发可处理的事件以获取有关键输入的信息。在许多情况下，Windows 窗体应用程序只需通过处理这些事件即可处理所有用户输入。在其他情况下，应用程序可能需要重写处理消息的方法之一，以便在应用程序、窗体或控件接收特定消息之前截获该消息。

键盘事件

所有 Windows 窗体控件都将继承与鼠标和键盘输入相关的一组事件。例如，控件可以处理 [KeyPress](#) 事件以确定所按下的键的字符代码。有关详细信息，请参阅[使用键盘事件](#)。

处理用户输入消息的方法

窗体和控件可以访问 [IMessageFilter](#) 接口和一组可重写的方法，这些方法可在消息队列中的不同位置处理 Windows 消息。这些方法都有 [Message](#) 参数，该参数用于封装 Windows 消息的低级别详细信息。可以实现或重写这些方法来检查消息，然后使用此消息或将其传递给消息队列中的下一个使用者。下表显示用于处理 Windows 窗体中所有 Windows 消息的方法。

[展开表](#)

方法	说明
PreFilterMessage	此方法在应用程序级别截获排队的（也称为已发布的）Windows 消息。
PreProcessMessage	此方法在 Windows 消息经过处理之前，在窗体和控件级别截获这些消息。
WndProc	此方法在窗体和控件级别处理 Windows 消息。
DefWndProc	此方法在窗体和控件级别执行 Windows 消息的默认处理。这提供了窗口的最小功能。
OnNotifyMessage	此方法在消息经过处理之后，在窗体和控件级别截获这些消息。必须设置 EnableNotifyMessage 样式位才能调用此方法。

键盘和鼠标消息也由其他一组特定于这些类型消息的可重写方法进行处理。有关详细信息，请参阅[键的预处理](#)部分。

密钥类型

Windows 窗体将键盘输入标识为由按位 [Keys 枚举](#) 表示的虚拟键代码。 使用 [Keys 枚举](#)，可以组合一系列按键以生成单个值。 这些值与 WM_KEYDOWN 和 WM_SYSKEYDOWN Windows 消息所附带的值相对应。 可以通过处理 [KeyDown](#) 或 [KeyUp](#) 事件来检测大多数物理按键。 字符键是 [Keys 枚举](#) 的子集，它们与 WM_CHAR 和 WM_SYSCHAR Windows 消息所附带的值相对应。 如果通过组合按键得到一个字符，则可以通过处理 [KeyPress](#) 事件来检测该字符。

键盘事件的顺序

正如上面列出的那样，在一个控件上可能出现三个与键盘相关的事件。 以下顺序是发生这些事件的常规顺序：

1. 用户按“a”键，该键将被预处理和调度，并且会发生 [KeyDown](#) 事件。
2. 用户按住“a”键，该键将被预处理和调度，并且会发生 [KeyPress](#) 事件。 当用户按住某个键时，此事件会发生多次。
3. 用户松开“a”键，该键将被预处理和调度，并且会发生 [KeyUp](#) 事件。

键的预处理

与其他消息一样，键盘消息也是在窗体或控件的 [WndProc](#) 方法中处理的。 但是，在处理键盘消息之前，[PreProcessMessage](#) 方法会调用一个或多个方法，这些方法可被重写以处理特殊的字符键和物理按键。 可以重写这些方法，以便在控件处理消息之前检测并筛选某些按键。 下表按照方法出现的顺序列出了正在执行的操作以及所出现的相关方法。

KeyDown 事件的预处理

 展开表

操作	相关方法	说明
检查命令键（如快捷键或菜单快捷键）。	ProcessCmdKey	此方法处理命令键，命令键的优先级高于常规键。 如果此方法返回 <code>true</code> ，则不调度键消息，而且不发生键事件。 如果它返回 <code>false</code> ，则调用 IsInputKey 。
检查该键是否为需要预处理的特殊键，或者是否为应引发 KeyDown 事件并且被调度到某个	IsInputKey	如果此方法返回 <code>true</code> ，则表示该控件为常规字符，并且会引发 KeyDown 事件。 如果返回 <code>false</code> ，则调用 ProcessDialogKey 。 注意：若要确保控件获取某个键或组合键，可以处理 PreviewKeyDown 事件，并针对所需的键或组合键将 PreviewKeyDownEventArgs 的 IsInputKey 设置为 <code>true</code> 。

操作	相关方法	说明
控件的普通字符键。		
检查该键是否为导航键（Esc、Tab、回车键或箭头键）。	ProcessDialogKey	此方法处理在控件内实现特殊功能（如在控件与其父级之间切换焦点）的物理按键。如果中间控件不处理该键，则会调用父控件的 ProcessDialogKey，直至层次结构中的最顶端控件。如果此方法返回 true，则完成预处理，而且不生成按键事件。如果它返回 false，则会发生 KeyDown 事件。

KeyPress 事件的预处理

[+] 展开表

操作	相关方法	说明
检查该键是否为控件应当处理的普通字符	IsInputChar	如果该字符是普通字符，则此方法返回 true，并且引发 KeyPress 事件，而且不再进行预处理。否则，将调用 ProcessDialogChar。
检查该字符是否为助记键（如按钮上的“确定(&O)”）	ProcessDialogChar	与 ProcessDialogKey 类似，将调用此方法，直至控件层次结构的顶端。如果控件是容器控件，此方法会通过调用控件及其子控件的 ProcessMnemonic 来检查助记键。如果 ProcessDialogChar 返回 true，则不会发生 KeyPress 事件。

处理键盘消息

键盘消息在到达窗体或控件的 WndProc 方法之后，它们会由一组可重写的方法来处理。其中的每种方法都返回一个 Boolean 值，该值指定控件是否已处理和使用了键盘消息。如果其中的某种方法返回 true，则键盘消息被视为已处理，而且它不传递到控件的基控件或父控件进行进一步处理。否则，消息停留在消息队列中，而且可能会在控件的基控件或父控件的其他方法中进行处理。下表显示用来处理键盘消息的方法。

[+] 展开表

方法	说明
ProcessKeyMessage	此方法处理由控件的 WndProc 方法接收的所有键盘消息。
ProcessKeyPreview	此方法将键盘消息发送到控件的父控件。如果 ProcessKeyPreview 返回 true，则不生成键事件，否则将调用 ProcessKeyEventArgs。

方法	说明
ProcessKeyEventArg	此方法会根据需要引发 KeyDown 、 KeyPress 和 KeyUp 事件。

重写键盘方法

在预处理和处理键盘消息时，可以使用许多方法进行重写；但是，这些方法有好有坏。下表显示可能需要完成的任务以及重写键盘方法的最佳方式。有关重写方法的详细信息，请参阅[继承 \(C# 编程指南\)](#) 或[继承 \(Visual Basic\)](#)

[+] 展开表

任务	方法
截获导航键并引发 KeyDown 事件。例如，希望在文本框中处理 Tab 键和回车键。	重写 IsInputKey 。注意：还可以处理 PreviewKeyDown 事件，并针对所需的键或组合键将 PreviewKeyDownEventArgs 的 IsInputKey 设置为 <code>true</code> 。
在控件上执行特殊的输入或导航处理。例如，你可能希望在列表控件中使用箭头键更改选定项。	重写 ProcessDialogKey
截获导航键并引发 KeyPress 事件。例如，你希望在数字调整框控件中，多次按箭头键来加快项的调整进度。	重写 IsInputChar 。
在 KeyPress 事件期间执行特殊的输入或导航处理。例如，在列表控件中，按住“r”键将跳到以字母 r 开头的项并在这些项间切换。	重写 ProcessDialogChar
执行自定义的助记键处理；例如，你希望处理所有者描述的、包含在工具栏中的按钮上的助记键。	重写 ProcessMnemonic 。

另请参阅

- [Keys](#)
- [WndProc](#)
- [PreProcessMessage](#)
- [使用键盘事件 \(Windows 窗体 .NET\)](#)
- [如何修改键盘键事件 \(Windows 窗体 .NET\)](#)
- [如何检查按下的修改键 \(Windows 窗体 .NET\)](#)
- [如何模拟键盘事件 \(Windows 窗体 .NET\)](#)

- 如何处理窗体中的键盘输入消息 (Windows 窗体 .NET)
- 添加控件 (Windows 窗体 .NET)

使用键盘事件 (Windows 窗体 .NET)

项目 · 2025/01/30

大多数 Windows 窗体程序通过处理键盘事件处理键盘输入。本文概述了键盘事件，包括有关何时使用每个事件以及为每个事件提供的数据的详细信息。有关一般事件的详细信息，请参阅 [事件概述 \(Windows 窗体 .NET\)](#)。

键盘事件

Windows 窗体提供三个事件：在用户按下键盘键时发生两个事件，在用户释放键盘键时发生一个事件。

- [KeyDown](#) 事件发生一次。
- [KeyPress](#) 事件，当用户按住同一个密钥时，可能会多次发生该事件。
- 当用户释放密钥时，[KeyUp](#) 事件发生一次。

当用户按下某个键时，Windows 窗体会根据键盘消息是指定字符键还是物理键来确定要引发的事件。有关字符和物理键的详细信息，请参阅 [键盘概述](#)、[键盘事件](#)。

下表描述了三个键盘事件。

展开表

键盘事件	描述	结果
KeyDown	当用户按下物理键时，将引发此事件。	<p>KeyDown 的处理程序接收：</p> <ul style="list-style-type: none">• KeyEventArgs 参数，该参数提供 KeyCode 属性（指定物理键盘按钮）。• Modifiers 属性（SHIFT、CTRL 或 ALT）。• KeyData 属性（合并键代码和修饰符）。 <p>KeyEventArgs 参数还提供：</p> <ul style="list-style-type: none">◦ Handled 属性，该属性可以设置为阻止基础控件接收密钥。◦ SuppressKeyPress 属性，可用于取消该击键的 KeyPress 和 KeyUp 事件。
KeyPress	当所按的某个键或多个键生成一个字符时，则引发此事件。例如，用户按 Shift 和小写“a”键，这会导致大写字母“A”字符。	<p>KeyPress 在 KeyDown 后引发。</p> <ul style="list-style-type: none">• KeyPress 的处理程序接收：• KeyPressEventArgs 参数，其中包含按下的键的字符代码。此字符代码对于字符键和修

键盘事件	描述	结果
		<p>饰键的每个组合都是唯一的。</p> <p>例如，“A”键将生成：</p> <ul style="list-style-type: none"> ◦ 字符代码 65 (如果与 SHIFT 键一起按下) ◦ 或 CAPS LOCK 键 97 (如果单独按下) , ◦ 以及 1 (如果与 CTRL 键一起按下) 。
KeyUp	当用户释放物理密钥时，将引发 KeyUp 的处理程序接收：此事件。	<p>• 一个 KeyEventArgs 参数：</p> <ul style="list-style-type: none"> ◦ 它提供 KeyCode 属性 (用于指定物理键 盘按钮) 。 ◦ Modifiers 属性 (SHIFT、CTRL 或 ALT) 。 ◦ KeyData 属性 (合并键代码和修饰符) 。

另请参阅

- [使用键盘的概述 \(Windows 窗体 .NET\)](#)
- [如何修改键盘键事件 \(Windows 窗体 .NET\)](#)
- [如何检查按下的修改键 \(Windows 窗体 .NET\)](#)
- [如何模拟键盘事件 \(Windows 窗体 .NET\)](#)
- [如何处理窗体中的键盘输入消息 \(Windows 窗体 .NET\)](#)

如何验证用户输入（Windows 窗体 .NET）概述

项目 • 2024/12/18

当用户将数据输入到应用程序中时，可能需要在应用程序使用它之前验证数据是否有效。您可能要求某些文本字段长度不为零、格式为电话号码的字段或字符串不包含无效字符。Windows Forms 提供了多种方式来验证应用程序中的输入。

MaskedTextBox 控件

如果需要要求用户以定义完善的格式（如电话号码或部件号）输入数据，则可以使用 [MaskedTextBox](#) 控件快速且最少的代码来实现此目的。掩码是由掩码语言中的字符组成的字符串，指定可在文本框中的任何给定位置输入哪些字符。该控件向用户显示一组提示。例如，如果用户键入不正确的条目（例如，在需要数字时键入字母），控件将自动拒绝输入。

[MaskedTextBox](#) 使用的掩码语言很灵活。它允许指定所需的字符、可选字符、文本字符，如连字符和括号、货币字符和日期分隔符。绑定到数据源时，该控件也有效。数据绑定上的 [Format](#) 事件可用于重新格式化传入数据以符合掩码，[Parse](#) 事件可用于重新格式化传出数据，以符合数据字段的规范。

事件驱动的验证

如果希望完全以编程方式控制验证，或者需要复杂的验证检查，则应使用大多数 Windows 窗体控件中内置的验证事件。接受自由格式用户输入的每个控件都有一个 [Validating](#) 事件，只要控件需要数据验证，就会发生该事件。在 [Validating](#) 事件处理方法中，可以通过多种方式验证用户输入。例如，如果你有一个必须包含邮政编码的文本框，则可以通过以下方式进行验证：

- 如果邮政编码必须属于特定邮政编码组，则可以对输入执行字符串比较，以验证用户输入的数据。例如，如果邮政编码必须位于集 {10001, 10002, 10003} 中，则可以使用字符串比较来验证数据。
- 如果邮政编码必须采用特定格式，则可以使用正则表达式来验证用户输入的数据。例如，若要验证表单 ##### 或 #####-####，可以使用正则表达式 ^(\d{5})(-\d{4})?\$. 若要验证表单 A#A #A#，可以使用正则表达式 [A-Z]\d[A-Z] \d[A-Z]\d。有关正则表达式的详细信息，请参阅 [.NET 正则表达式](#) 和 [正则表达式示例](#)。

- 如果邮政编码必须是有效的美国邮政编码，则可以调用邮政编码 Web 服务来验证用户输入的数据。

[Validating](#) 事件提供 [CancelEventArgs](#)类型的对象。 如果确定控件的数据无效，请通过将此对象的 [Cancel](#) 属性设置为 `true` 来取消 [Validating](#) 事件。 如果未设置 [Cancel](#) 属性，Windows 窗体将假定该控件的验证成功并引发 [Validated](#) 事件。

有关验证 [TextBox](#)中电子邮件地址的代码示例，请参阅 [Validating](#) 事件参考。

事件驱动的验证数据绑定控件

将控件绑定到数据源（例如数据库表）时，验证非常有用。 通过使用验证，可以确保控件的数据满足数据源所需的格式，并且它不包含任何特殊字符，例如引号和可能不安全的反斜杠。

使用数据绑定时，控件中的数据在执行 [Validating](#) 事件期间会与数据源同步。 如果取消 [Validating](#) 事件，则数据不会与数据源同步。

① 重要

如果自定义验证发生在 [Validating](#) 事件之后，则不会影响数据绑定。 例如，如果在尝试取消数据绑定的 [Validated](#) 事件中有代码，则数据绑定仍将发生。 在这种情况下，若要在 [Validated](#) 事件中执行验证，请将控件的 [Binding.DataSourceUpdateMode](#) 属性从 [DataSourceUpdateMode.OnValidation](#) 更改为 [DataSourceUpdateMode.Never](#)，并将 `your-control.DataBindings["field-name"].WriteValue()` 添加到验证代码。

隐式验证和显式验证

那么，控件的数据何时得到验证？ 这取决于你，开发人员。 可以根据应用程序的需求使用隐式验证或显式验证。

隐式验证

隐式验证方法会在用户输入数据时验证数据。 通过读取按下的键来验证数据，或者更常见的是在用户将输入焦点移开控件时进行验证。 当你希望在用户处理数据时提供即时反馈时，此方法非常有用。

如果要对控件使用隐式验证，必须将该控件的 [AutoValidate](#) 属性设置为 [EnablePreventFocusChange](#) 或 [EnableAllowFocusChange](#)。 如果取消 [Validating](#) 事件，控件的行为将取决于分配给 [AutoValidate](#)的值。 如果分配了

`EnablePreventFocusChange`, 取消事件将导致 `Validated` 事件不发生。输入焦点将保留在当前控件上，直到用户将数据更改为有效格式。如果分配了 `EnableAllowFocusChange`，则取消事件时不会发生 `Validated` 事件，但焦点仍将更改为下一个控件。

将 `Disable` 分配给 `AutoValidate` 属性会完全阻止隐式验证。若要验证控件，必须使用显式验证。

显式验证

显式验证方法一次验证数据。可以验证数据以响应用户操作，例如单击“**保存**”按钮或“**下一步**”链接。发生用户操作时，可以通过以下方式之一触发显式验证：

- 调用 `Validate` 以验证失去焦点的最后一个控件。
- 调用 `ValidateChildren` 以验证窗体或容器控件中的所有子控件。
- 调用自定义方法以手动验证控件中的数据。

控件的默认隐式验证行为

不同的 Windows 窗体控件对其 `AutoValidate` 属性具有不同的默认值。下表显示了最常见的控件及其默认值。

[+] 展开表

控制	默认验证行为
<code>ContainerControl</code>	<code>Inherit</code>
<code>Form</code>	<code>EnableAllowFocusChange</code>
<code>PropertyGrid</code>	Visual Studio 中未公开的属性
<code>ToolStripContainer</code>	Visual Studio 中未公开的属性
<code>SplitContainer</code>	<code>Inherit</code>
<code>UserControl</code>	<code>EnableAllowFocusChange</code>

关闭窗体并重写验证

当控件由于包含的数据无效而保持焦点时，不可能以一种常用方式关闭父窗体：

- 单击“**关闭**”按钮。
- 选择 **系统>关闭** 菜单。

- 通过编程方式调用 [Close](#) 方法。

但是，在某些情况下，你可能希望让用户关闭窗体，而不考虑控件中的值是否有效。可以通过为表单的 [FormClosing](#) 事件创建处理程序来替代验证并关闭仍包含无效数据的窗体。在事件中，将 [Cancel](#) 属性设置为 `false`。这会强制窗体关闭。有关详细信息和示例，请参阅 [Form.FormClosing](#)。

① 备注

如果强制窗体以这种方式关闭，则表单控件中尚未保存的任何数据都将丢失。此外，模式窗体在关闭控件时不会验证控件的内容。你仍然可以使用控件验证将焦点锁定到控件，但不必担心与关闭窗体相关的行为。

另请参阅

- [使用键盘事件 \(Windows 窗体 .NET\)](#)
- [Control.Validating](#)
- [Form.FormClosing](#)
- [System.Windows.Forms.FormClosingEventArgs](#)

如何修改键盘键事件 (Windows 窗体 .NET)

项目 • 2025/01/30

Windows 窗体提供使用和修改键盘输入的功能。 使用键是指处理方法或事件处理程序内的键，以便消息队列更低处的其他方法和事件不会收到键值。 修改键指的是修改键的值，以便消息队列中随后的方法和事件处理程序接收到不同的键值。 本文介绍如何完成这些任务。

使用键

在 `KeyPress` 事件处理程序中，将 `KeyPressEventArgs` 类的 `Handled` 属性设置为 `true`。

- 或 -

在 `KeyDown` 事件处理程序中，将 `KeyEventArgs` 类的 `Handled` 属性设置为 `true`。

① 备注

在 `KeyDown` 事件处理程序中设置 `Handled` 属性不会阻止针对当前按键引发 `KeyPress` 和 `KeyUp` 事件。为此，请使用 `SuppressKeyPress` 属性。

以下示例处理 `KeyPress` 事件以消耗 `A` 和 `a` 字符键。这些键无法键入到文本框中：

C#

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == 'a' || e.KeyChar == 'A')
        e.Handled = true;
}
```

修改标准字符键

在 `KeyPress` 事件处理程序中，将 `KeyPressEventArgs` 类的 `KeyChar` 属性设置为新字符键的值。

以下示例处理 `KeyPress` 事件以将任何 `A` 和 `a` 字符键更改为 `!`：

C#

```
private void textBox2_KeyPress(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar == 'a' || e.KeyChar == 'A')
    {
        e.KeyChar = '!';
        e.Handled = false;
    }
}
```

修改非字符键

仅可通过从控件继承并重写 [PreProcessMessage](#) 方法来修改非字符按键。当输入 [Message](#) 发送到控件时，将在控件引发事件之前对其进行处理。可以截获这些消息以修改或阻止它们。

以下代码示例演示了如何使用 [Message](#) 参数的 [WParam](#) 属性来更改按下的键。此代码检测从 [F1](#) 到 [F10](#) 的键，并将其转换为范围从 [0](#) 到 [9](#) 的数字键（其中 [F10](#) 映射到 [0](#)）。

C#

```
public override bool PreProcessMessage(ref Message m)
{
    const int WM_KEYDOWN = 0x100;

    if (m.Msg == WM_KEYDOWN)
    {
        Keys keyCode = (Keys)m.WParam & Keys.KeyCode;

        // Detect F1 through F9.
        m.WParam = keyCode switch
        {
            Keys.F1 => (IntPtr)Keys.D1,
            Keys.F2 => (IntPtr)Keys.D2,
            Keys.F3 => (IntPtr)Keys.D3,
            Keys.F4 => (IntPtr)Keys.D4,
            Keys.F5 => (IntPtr)Keys.D5,
            Keys.F6 => (IntPtr)Keys.D6,
            Keys.F7 => (IntPtr)Keys.D7,
            Keys.F8 => (IntPtr)Keys.D8,
            Keys.F9 => (IntPtr)Keys.D9,
            Keys.F10 => (IntPtr)Keys.D0,
            _ => m.WParam
        };
    }

    // Send all other messages to the base method.
}
```

```
        return base.PreProcessMessage(ref m);  
    }  
}
```

另请参阅

- [使用键盘的概述 \(Windows 窗体 .NET\)](#)
- [使用键盘事件 \(Windows 窗体 .NET\)](#)
- [Keys](#)
- [KeyDown](#)
- [KeyPress](#)

如何检查是否按下修饰键（Windows 窗体的.NET）

项目 · 2024/12/18

当用户在应用程序中键入键时，可以监视按下的修饰键，例如 `SHIFT`、`ALT` 和 `CTRL`。当修饰键与其他键或鼠标单击结合使用时，应用程序可以相应地做出响应。例如，按 `S` 键可能会导致屏幕上显示“s”。如果按下 `Ctrl+S` 键，当前文档可能会被保存。

如果您处理 `KeyDown` 事件，那么事件处理程序收到的 `KeyEventArgs.Modifiers` 属性将指定被按下的修饰键。此外，`KeyEventArgs.KeyData` 属性指定与按位 OR 组合的任何修饰键一起按下的字符。

如果要处理 `KeyPress` 事件或鼠标事件，事件处理程序不会收到此信息。使用 `Control` 类的 `ModifierKeys` 属性检测键修饰符。在任一情况下，都必须对适当的 `Keys` 值和要测试的值执行按位 AND。`Keys` 枚举提供了每个修饰键的变体，因此请务必按位和检查正确的值。

例如，`SHIFT` 键由以下键值表示：

- `Keys.Shift`
- `Keys.ShiftKey`
- `Keys.RShiftKey`
- `Keys.LShiftKey`

为了作为修饰键测试 `SHIFT` 键的正确值是 `Keys.Shift`。同样，若要测试 `CTRL` 和 `ALT` 作为修饰符时，应分别使用 `Keys.Control` 和 `Keys.Alt` 值。

检测修饰键

通过将 `ModifierKeys` 属性和 `Keys` 枚举值与按位 AND 运算符进行比较，检测是否按下了修饰键。

下面的代码示例演示如何确定 `KeyPress` 和 `KeyDown` 事件处理程序中是否按下了 `SHIFT` 键。

C#

```
// Event only raised when non-modifier key is pressed
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((Control.ModifierKeys & Keys.Shift) == Keys.Shift)
        MessageBox.Show("KeyPress " + Keys.Shift);
}
```

```
// Event raised as soon as shift is pressed
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
    if ((Control.ModifierKeys & Keys.Shift) == Keys.Shift)
        MessageBox.Show("KeyDown " + Keys.Shift);
}
```

另请参阅

- [使用键盘 \(Windows 窗体 .NET\) 概述](#)
- [使用键盘事件 \(Windows 窗体 .NET\)](#)
- [Keys](#)
- [ModifierKeys](#)
- [KeyDown](#)
- [KeyPress](#)

如何处理窗体中的键盘输入消息 (Windows 窗体 .NET)

项目 • 2024/11/05

Windows 窗体能够在消息到达控件之前处理窗体级别的键盘消息。本文演示如何完成此任务。

处理键盘消息

处理有效窗体的 `KeyPress` 或 `KeyDown` 事件，并将该窗体的 `KeyPreview` 属性设置为 `true`。此属性会使键盘消息在到达窗体上的任何控件之前就被窗体接收。以下代码示例通过检测所有数字键并使用 `1`、`4` 和 `7` 来处理 `KeyPress` 事件。

C#

```
// Detect all numeric characters at the form level and consume 1,4, and 7.  
// Form.KeyPreview must be set to true for this event handler to be called.  
void Form1_KeyPress(object sender, KeyPressEventArgs e)  
{  
    if (e.KeyChar >= 48 && e.KeyChar <= 57)  
    {  
        MessageBox.Show($"Form.KeyPress: '{e.KeyChar}' pressed.");  
  
        switch (e.KeyChar)  
        {  
            case (char)49:  
            case (char)52:  
            case (char)55:  
                MessageBox.Show($"Form.KeyPress: '{e.KeyChar}' consumed.");  
                e.Handled = true;  
                break;  
        }  
    }  
}
```

另请参阅

- [使用键盘的概述 \(Windows 窗体 .NET\)](#)
- [使用键盘事件 \(Windows 窗体 .NET\)](#)
- [Keys](#)
- [ModifierKeys](#)
- [KeyDown](#)
- [KeyPress](#)

如何模拟键盘事件（Windows 窗体 .NET）

项目 • 2024/11/05

Windows 窗体提供几个选项，用于以编程方式模拟键盘输入。本文将简要阐述这些选项。

使用 SendKeys

Windows 窗体提供 `System.Windows.Forms.SendKeys` 类，用于向活动应用程序发送击键。可通过两种方法将击键发送到应用程序：`SendKeys.Send` 和 `SendKeys.SendWait`。这两种方法的区别在于，在发送击键时，`SendWait` 会阻止当前线程，等待响应，而 `Send` 不会。有关 `SendWait` 的详细信息，请参阅[将击键发送到其他应用程序](#)。

⊗ 注意

如果你的应用程序旨在用于全球各种键盘，使用 `SendKeys.Send` 可能会产生不可预知的结果，应当避免。

在后台，`SendKeys` 使用较旧的 Windows 实现来发送输入，这可能会在新式 Windows 上失败，因为在新式 Windows 上应用程序不使用管理权限运行。如果较旧的实现失败，代码会自动尝试较新的 Windows 实现来发送输入。此外，如果 `SendKeys` 类使用新的实现，在将击键发送到其他应用程序时，`SendWait` 方法不再阻止当前线程。

ⓘ 重要

如果无论使用何种操作系统，你的应用程序均依赖于一致的行为，则可通过将以下应用程序设置添加至 `app.config` 文件强制执行 `SendKeys` 类以使用新的实现。

XML

```
<appSettings>
  <add key="SendKeys" value="SendInput"/>
</appSettings>
```

若要强制执行 `SendKeys` 类以仅使用以前的实现，请改用 "JournalHook" 值。

若要将击键发送到相同的应用程序

调用 [SendKeys.Send](#) 类或 [SendKeys.SendWait](#) 类的 [SendKeys](#) 方法。 指定的击键将由应用程序的活动控件接收。

下面的代码示例使用 [Send](#) 来模拟同时按 [ALT](#) 和 [DOWN](#) 键。 这些击键会导致 [ComboBox](#) 控件显示其下拉列表。 此示例假定 [Form](#) 具有 [Button](#) 和 [ComboBox](#)。

C#

```
private void button1_Click(object sender, EventArgs e)
{
    comboBox1.Focus();
    SendKeys.Send("%+{DOWN}");
}
```

若要将击键发送到不同的应用程序

[SendKeys.Send](#) 和 [SendKeys.SendWait](#) 方法将击键发送到活动应用程序，该应用程序通常是你从其发送击键的应用程序。 若要将击键发送到其他应用程序，首先需要将其激活。 由于没有可激活其他应用程序的托管方法，所以必须使用本机 Windows 方法聚焦其他应用程序。 下面的代码示例使用平台调用来调用 [FindWindow](#) 和 [SetForegroundWindow](#) 方法以激活计算器应用程序窗口，然后调用 [Send](#) 向计算器应用程序发出一系列计算。

下面的代码示例使用 [Send](#) 模拟按下键进入 Windows 10 计算器应用程序。 它首先搜索标题为 [Calculator](#) 的应用程序窗口，然后将其激活。 激活后，将发送击键以计算 10 加 10。

C#

```
[DllImport("USER32.DLL", CharSet = CharSet.Unicode)]
public static extern IntPtr FindWindow(string lpClassName, string
lpWindowName);

[DllImport("USER32.DLL")]
public static extern bool SetForegroundWindow(IntPtr hWnd);

private void button1_Click(object sender, EventArgs e)
{
    IntPtr calcWindow = FindWindow(null, "Calculator");

    if (SetForegroundWindow(calcWindow))
        SendKeys.Send("10{+}10=");
}
```

使用 [OnEventName](#) 方法

模拟键盘事件的最简单方法是在引发事件的对象上调用方法。大多数事件都具有调用这些事件的相应方法，以 `On` 后跟 `EventName` 的模式命名，如 `OnKeyPress`。这种方式只适用于自定义控件或窗体内，因为这些方法受到保护且不能从控件或窗体的上下文外访问。

这些受保护的方法可用于模拟键盘事件。

- `OnKeyDown`
- `OnKeyPress`
- `OnKeyUp`

有关这些事件的详细信息，请参阅[使用键盘事件 \(Windows 窗体 .NET\)](#)。

另请参阅

- [使用键盘的概述 \(Windows 窗体 .NET\)](#)
- [使用键盘事件 \(Windows 窗体 .NET\)](#)
- [使用鼠标事件 \(Windows 窗体 .NET\)](#)
- `SendKeys`
- `Keys`
- `KeyDown`
- `KeyPress`

使用鼠标的概述 (Windows 窗体 .NET)

项目 · 2025/01/30

接收和处理鼠标输入是每个 Windows 应用程序的重要组成部分。可以处理鼠标事件以在应用程序中执行操作，或使用鼠标位置信息执行命中测试或其他操作。此外，还可以更改应用程序中的控件处理鼠标输入的方式。本文详细介绍了这些鼠标事件，以及如何获取和更改鼠标的系统设置。

在 Windows 窗体中，用户输入以 [Windows 消息](#) 的形式发送到应用程序。一系列可重写的方法在应用程序、窗体和控件级别处理这些消息。当这些方法接收鼠标消息时，它们会引发可以处理的事件以获取有关鼠标输入的信息。在许多情况下，Windows 窗体应用程序只需处理这些事件即可处理所有用户输入。在其他情况下，应用程序可能会重写处理消息的方法之一，以便在应用程序、窗体或控件接收特定消息之前截获该消息。

鼠标事件

所有 Windows 窗体控件都继承一组与鼠标和键盘输入相关的事件。例如，控件可以处理 [MouseClick](#) 事件，以确定鼠标单击的位置。有关鼠标事件的详细信息，请参阅[使用鼠标事件](#)。

鼠标位置和命中测试

当用户移动鼠标时，操作系统将移动鼠标指针。鼠标指针有一个称为热点的单个像素，操作系统通过跟踪这个像素来识别指针的位置。当用户移动鼠标或按下鼠标按钮时，[Control](#)（包含 [HotSpot](#)）将触发相应的鼠标事件。

可以在处理鼠标事件时，通过[MouseEventArgs](#)的[Location](#)属性获取当前鼠标位置，或者使用[Cursor](#)类的[Position](#)属性来获取。然后，可以使用鼠标位置信息执行命中测试，然后根据鼠标的位置执行操作。命中测试功能内置于 Windows 窗体中的若干控件，如 [ListView](#)、[TreeView](#)、[MonthCalendar](#) 和 [DataGridView](#) 控件。

在适当的鼠标事件（例如[MouseHover](#)）中使用，命中测试在判断应用程序何时执行特定操作方面非常有用。

更改鼠标输入设置

可以通过从控件派生和使用 [GetStyle](#) 和 [SetStyle](#) 方法来检测和更改控件处理鼠标输入的方式。[SetStyle](#) 方法使用 [ControlStyles](#) 值的按位组合来确定控件是否将具有标准单击、双击行为，或控件是否将处理自己的鼠标处理。此外，[SystemInformation](#) 类包括描述鼠标功能的属性，并指定鼠标与操作系统的交互方式。下表汇总了这些属性。

[\[+\] 展开表](#)

财产	描述
DoubleClickSize	获取如下区域的尺寸 (以像素为单位) : 用户必须在此区域内单击两次, 操作系统才将这两次单击视为一次双击。
DoubleClickTime	获取要将鼠标操作视为双击的第一次单击与第二次单击之间可以经过的最大毫秒数。
MouseButtons	获取鼠标上的按钮数。
MouseButtonsSwapped	获取一个值, 该值指示是否交换了鼠标左键和右键的功能。
MouseHoverSize	获取特定矩形的尺寸 (以像素为单位), 鼠标指针必须在该矩形范围内停留达到鼠标悬停时间后, 才会生成鼠标悬停消息。
MouseHoverTime	获取一个以毫秒为单位的时间, 鼠标指针必须在悬停矩形中停留该时间后, 才会生成鼠标悬停消息。
MousePresent	获取一个值, 该值指示是否安装了鼠标。
MouseSpeed	获取一个值, 该值指示当前鼠标速度, 从 1 到 20。
MouseWheelPresent	获取一个值, 该值指示是否安装了带有鼠标滚轮的鼠标。
MouseWheelScrollDelta	获取单次鼠标轮旋转增量的增量值。
MouseWheelScrollLines	获取滚动鼠标轮时所滚动过的行数。

处理用户输入消息的方法

窗体和控件可以访问 [IMessageFilter](#) 接口和一组可重写的方法, 这些方法可在消息队列中的不同位置处理 Windows 消息。这些方法都有一个 [Message](#) 参数, 该参数封装了 Windows 消息的低级别详细信息。你可以实现或重写这些方法来检查消息, 然后处理消息或将其传递给消息队列中的下一个使用者。下表展示了用于处理 Windows 窗体中所有 Windows 消息的方法。

[\[+\] 展开表](#)

方法	说明
PreFilterMessage	此方法在应用程序级别截获排队的 (也称为已发布的) Windows 消息。
PreProcessMessage	此方法在 Windows 消息被处理之前, 会在窗体和控件级别拦截它们。
WndProc	此方法在窗体和控件级别处理 Windows 消息。

方法	说明
DefWndProc	此方法在窗体和控件级别执行 Windows 消息的默认处理。这提供了窗口的最小功能。
OnNotifyMessage	此方法在消息经过处理之后，在窗体和控件级别截获这些消息。必须设置 EnableNotifyMessage 样式位才能调用此方法。

另请参阅

- [使用鼠标事件 \(Windows 窗体 .NET\)](#)
- [拖放鼠标行为概述 \(Windows 窗体 .NET\)](#)
- [管理鼠标指针 \(Windows Forms .NET\)](#)

使用鼠标事件 (Windows 窗体 .NET)

项目 · 2024/12/18

大多数 Windows 窗体程序通过处理鼠标事件来处理鼠标输入。本文概述了鼠标事件，包括有关何时使用每个事件以及为每个事件提供的数据的详细信息。有关一般事件的详细信息，请参阅 [事件概述 \(Windows 窗体 .NET\)](#)。

鼠标事件

响应鼠标输入的主要方法是处理鼠标事件。下表显示了鼠标事件，并描述了它们何时触发。

 展开表

鼠标事件	描述
Click	在释放鼠标按钮时（通常在 <code>MouseUp</code> 事件之前）发生此事件。此事件的处理程序接收类型为 <code>EventArgs</code> 的参数。当你只需要确定单击何时发生时，处理此事件。
MouseClick	当用户使用鼠标单击控件时发生此事件。此事件的处理程序接收类型为 <code>MouseEventArgs</code> 的参数。当需要在鼠标点击发生时获取有关鼠标的信息时，处理此事件。
DoubleClick	双击控件时发生此事件。此事件的处理程序接收类型为 <code>EventArgs</code> 的参数。仅当需要确定何时发生双击时处理此事件。
MouseDoubleClick	当用户使用鼠标双击控件时发生此事件。此事件的处理程序接收类型为 <code>MouseEventArgs</code> 的参数。如果需要在双击发生时获取有关鼠标的信息，请处理此事件。
MouseDown	当鼠标指针位于控件上并且用户按下鼠标按钮时，将发生此事件。此事件的处理程序接收类型为 <code>MouseEventArgs</code> 的参数。
MouseEnter	当鼠标指针进入控件的边框或工作区时，将发生此事件，具体取决于控件的类型。此事件的处理程序接收类型为 <code>EventArgs</code> 的参数。
MouseHover	当鼠标指针停止并停留在控件上时，将发生此事件。此事件的处理程序接收类型为 <code>EventArgs</code> 的参数。
MouseLeave	当鼠标指针离开控件的边框或工作区时，将发生此事件，具体取决于控件的类型。此事件的处理程序接收类型为 <code>EventArgs</code> 的参数。
MouseMove	当鼠标指针在控件上移动时发生此事件。此事件的处理程序接收类型为 <code>MouseEventArgs</code> 的参数。

鼠标事件	描述
MouseUp	当鼠标指针位于控件上并且用户释放鼠标按钮时，会发生此事件。此事件的处理程序接收类型为 MouseEventArgs 的参数。
MouseWheel	当用户在控件具有焦点时旋转鼠标滚轮时发生此事件。此事件的处理程序接收类型为 MouseEventArgs 的参数。可以使用 MouseEventArgs 的 Delta 属性来测量鼠标滚动的距离。

鼠标信息

将发送 [MouseEventArgs](#) 到与鼠标按钮单击和鼠标移动跟踪相关的鼠标事件处理程序。[MouseEventArgs](#) 提供有关鼠标的当前状态的信息，包括鼠标指针在客户端坐标中的位置、按下鼠标按钮以及鼠标滚轮是否滚动。一些鼠标事件（例如，当鼠标指针进入或离开控件边界时引发的事件）将 [EventArgs](#) 发送到事件处理程序，但没有进一步的信息。

如果想要了解鼠标按钮的当前状态或鼠标指针的位置，并且想要避免处理鼠标事件，还可以使用 [Control](#) 类的 [MouseButtons](#) 和 [MousePosition](#) 属性。[MouseButtons](#) 返回当前按下哪些鼠标按钮的信息。[MousePosition](#) 返回鼠标指针的屏幕坐标，等效于 [Position](#) 返回的值。

在屏幕和客户端坐标之间转换

由于某些鼠标位置信息位于客户端坐标中，有些位于屏幕坐标中，因此可能需要将点从一个坐标系转换为另一个坐标系统。可以使用 [Control](#) 类上提供的 [PointToClient](#) 和 [PointToScreen](#) 方法轻松执行此操作。

标准单击事件行为

如果要按正确的顺序处理鼠标单击事件，则需要知道在 Windows 窗体控件中引发单击事件的顺序。当按下并释放任何受支持的鼠标按钮时，所有 Windows 窗体控件都按相同的顺序引发单击事件，但以下列表中指出的各个控件除外。以下列表显示了鼠标单击一个按钮时所引发的事件的顺序：

1. [MouseDown](#) 事件。
2. [Click](#) 事件。
3. [MouseClick](#) 事件。
4. [MouseUp](#) 事件。

以下是双击鼠标引发的事件顺序：

1. [MouseDown](#) 事件。

2. [Click](#) 事件。
3. [MouseClick](#) 事件。
4. [MouseUp](#) 事件。
5. [MouseDown](#) 事件。
6. [DoubleClick](#) 事件。

这可能会有所不同，具体取决于控件是否将 [StandardDoubleClick](#) 样式位设置为 `true`。有关如何设置 [ControlStyles](#) 位的详细信息，请参阅 [SetStyle](#) 方法。

7. [MouseDoubleClick](#) 事件。
8. [MouseUp](#) 事件。

独立控件

以下控件不符合标准鼠标单击事件行为：

- [Button](#)
- [CheckBox](#)
- [ComboBox](#)
- [RadioButton](#)

① 备注

对于 [ComboBox](#) 控件，如果用户单击编辑字段、按钮或列表中的项，则稍后将发生详细事件行为。

- **左键单击：** [Click](#)、[MouseClick](#)
 - **右键单击：** 未引发单击事件
 - **左双击：** [Click](#), [MouseClick](#); [Click](#), [MouseClick](#)
 - **右键双击：** 未引发单击事件
- [TextBox](#)、[RichTextBox](#)、[ListBox](#)、[MaskedTextBox](#)和 [CheckedListBox](#) 控件

① 备注

当用户单击这些控件中的任何位置时，稍后将发生详细事件行为。

- **左键单击：** [Click](#)、[MouseClick](#)

- **右键单击**: 未引发单击事件
 - **左双击**: Click、MouseClick、DoubleClick、MouseDoubleClick
 - **右键单击**: 未引发单击事件
- ListView 控件

① 备注

仅当用户单击 ListView 控件中的项时，才会发生稍后详述的事件行为。在控件的其他任意位置单击不会引发任何事件。除了稍后介绍的事件之外，还有 BeforeLabelEdit 和 AfterLabelEdit 事件，如果您希望在 ListView 控件中使用验证功能，可能会对此感兴趣。

- **左键单击**: Click、MouseClick
 - **右键单击**: Click、MouseClick
 - **左双击**: Click, MouseClick;DoubleClick, MouseDoubleClick
 - **右键单击**: Click, MouseClick;DoubleClick, MouseDoubleClick
- TreeView 控件

① 备注

仅当用户单击项本身或 TreeView 控件中项右侧时，才会发生稍后详述的事件行为。在控件的其他任何位置单击时，不会触发任何事件。除了稍后所述的事件之外，还有 BeforeCheck、BeforeSelect、BeforeLabelEdit、AfterSelect、AfterCheck 和 AfterLabelEdit 事件，如果想要将验证与 TreeView 控件一起使用，则可能对你感兴趣。

- **左键单击**: Click、MouseClick
- **右键单击**: Click、MouseClick
- **左双击**: Click, MouseClick;DoubleClick, MouseDoubleClick
- **右键单击**: Click, MouseClick;DoubleClick, MouseDoubleClick

切换控件的绘制行为

切换控件（如派生自 ButtonBase 类的控件）具有以下独特的绘画行为与鼠标单击事件的组合：

1. 用户按下鼠标按钮。
2. 控件在按下状态下进行绘制。
3. 触发 MouseDown 事件。

4. 用户释放鼠标按钮。
5. 控件呈现为凸起状态。
6. 引发 [Click](#) 事件。
7. [MouseClick](#) 事件被引发。
8. 引发 [MouseUp](#) 事件。

① 备注

如果用户在按住鼠标按钮时将指针移出切换控件（例如在按住时将鼠标从 [Button](#) 控件上移开），则切换控件将显示为凸起状态，并且仅会触发 [MouseUp](#) 事件。在这种情况下，不会发生 [Click](#) 或 [MouseClick](#) 事件。

另请参阅

- [概述使用鼠标 \(Windows 窗体 .NET\)](#)
- [管理鼠标指针 \(Windows Forms .NET\)](#)
- [如何模拟鼠标事件 \(Windows 窗体 .NET\)](#)
- [System.Windows.Forms.Control](#)

拖放鼠标行为概述 (Windows 窗体 .NET)

项目 • 2024/11/05

Windows 窗体包含一组实现拖放行为的方法、事件和类。本主题概述了 Windows 窗体对拖放功能的支持。

拖放事件

拖放操作中有两类事件：一类是拖放操作的当前目标上发生的事件，一类是拖放操作的源上发生的事件。若要执行拖放操作，必须处理这些事件。通过使用这些事件的事件参数中的可用信息，可以轻松地实现拖放操作。

当前拖放目标上的事件

下表显示在拖放操作的当前目标上发生的事件。

展开表

鼠标事件	说明
DragEnter	将对象拖入控件的边界时此事件发生。此事件的处理程序接收类型为 <code>DragEventArgs</code> 的参数。
DragOver	在鼠标指针位于控件的边界内时如果拖动对象则此事件发生。此事件的处理程序接收类型为 <code>DragEventArgs</code> 的参数。
DragDrop	拖放操作完成时此事件发生。此事件的处理程序接收类型为 <code>DragEventArgs</code> 的参数。
DragLeave	将对象拖出控件的边界时此事件发生。此事件的处理程序接收类型为 <code>EventArgs</code> 的参数。

`DragEventArgs` 类提供鼠标指针的位置、鼠标按钮和键盘修改键的当前状态、正在拖动的数据以及 `DragDropEffects` 值（指定拖动事件的源所允许的操作以及操作的目标放置效果）。

放置源上的事件

下表显示在拖放操作的源上发生的事件。

鼠标事件	说明
GiveFeedback	此事件在执行拖动操作期间发生。 借助此事件，可向用户提供可视提示（例如更改鼠标指针），通知拖放操作正在发生。 此事件的处理程序接收类型为 GiveFeedbackEventArgs 的参数。
QueryContinueDrag	此事件在拖放操作期间引发，并使拖动源可以确定是否应取消拖放操作。 此事件的处理程序接收类型为 QueryContinueDragEventArgs 的参数。

[QueryContinueDragEventArgs](#) 类提供鼠标按钮和键盘修改键的当前状态、指定是否按 ESC 键的值以及 DragAction 值（可设置为指定是否应继续拖放操作）。

执行拖放操作

拖放操作始终涉及两个组件 - 放置源和拖放目标。 若要启动拖放操作，请指定一个控件作为源，并处理 [MouseDown](#) 事件。 在事件处理程序中，调用 [DoDragDrop](#) 方法，该方法提供与放置关联的数据和 [DragDropEffects](#) 值。

将目标控件的 [AllowDrop](#) 属性设置为 `true`，以允许该控件接受拖放操作。 目标处理两个事件，第一个是响应控件上的拖动的事件，如 [DragOver](#)。 第二个事件是放置操作本身 - [DragDrop](#)。

下面的示例演示一个从 [Label](#) 控件到 [TextBox](#) 的拖动操作。 拖动操作完成后，[TextBox](#) 通过将标签的文本分配给其自身来做出响应。

C#

```
// Initiate the drag
private void label1_MouseDown(object sender, MouseEventArgs e) =>
    DoDragDrop(((Label)sender).Text, DragDropEffects.All);

// Set the effect filter and allow the drop on this control
private void textBox1_DragOver(object sender, DragEventArgs e) =>
    e.Effect = DragDropEffects.All;

// React to the drop on this control
private void textBox1_DragDrop(object sender, DragEventArgs e) =>
    textBox1.Text = (string)e.Data.GetData(typeof(string));
```

有关拖动效果的详细信息，请参阅[Data](#) 和 [AllowedEffect](#)。

另请参阅

- 使用鼠标的概述 (Windows 窗体 .NET)
- Control.DragDrop
- Control.DragEnter
- Control.DragLeave
- Control.DragOver

如何区分单击和双击（Windows 窗体 .NET）

项目 • 2024/11/05

通常情况下，一次单击会启动一个用户界面操作，而一次双击则会扩展该操作。例如，一次单击通常可选择一个项，而双击则可编辑所选的项。但是，Windows 窗体 Click 事件无法轻松应用于单击和双击执行多个不兼容操作的方案，因为绑定到 Click 或 MouseClick 事件的操作会在操作绑定到 DoubleClick 或 MouseDoubleClick 事件之前执行。本主题演示此问题的两种解决方案。

一种解决方案是处理双击事件，并回滚单击事件处理过程中的操作。在极少数情况下，可能需要通过处理 MouseDown 事件并使用 SystemInformation 类的 DoubleClickTime 和 DoubleClickSize 属性来模拟单击和双击行为。度量点击之间的时间，如果在达到 DoubleClickTime 值之前发生第二次单击，并且单击发生在由 DoubleClickSize 定义的矩形范围内，请执行双击操作；否则，请执行单击操作。

回滚单击操作

请确保你正在使用的控件具有标准双击行为。如果不具有标准双击行为，请启用具有 SetStyle 方法的控件。处理双击事件，并回滚单击操作以及双击操作。下面的代码示例演示了如何在启用了双击的情况下创建自定义按钮，以及如何回滚双击事件处理代码中的单击操作。

此代码示例使用可启用双击的新按钮控件：

```
C#  
  
public partial class DoubleClickButton : Button  
{  
    public DoubleClickButton()  
    {  
        // Set the style so a double click event occurs.  
        SetStyle(ControlStyles.StandardClick |  
ControlStyles.StandardDoubleClick, true);  
    }  
}
```

下面的代码演示了窗体如何通过单击或双击新按钮控件来更改边框样式：

```
C#  
  
public partial class Form1 : Form  
{
```

```

private FormBorderStyle _initialStyle;
private bool _isDoubleClicking;

public Form1()
{
    InitializeComponent();
}

private void Form1_Load(object sender, EventArgs e)
{
    _initialStyle = this.FormBorderStyle;

    var button1 = new DoubleClickButton();
    button1.Location = new Point(50, 50);
    button1.Size = new Size(200, 23);
    button1.Text = "Click or Double Click";
    button1.Click += Button1_Click;
    button1.DoubleClick += Button1_DoubleClick;

    Controls.Add(button1);
}

private void Button1_DoubleClick(object sender, EventArgs e)
{
    // This flag prevents the click handler logic from running
    // A double click raises the click event twice.
    _isDoubleClicking = true;
    FormBorderStyle = _initialStyle;
}

private void Button1_Click(object sender, EventArgs e)
{
    if (_isDoubleClicking)
        _isDoubleClicking = false;
    else
        FormBorderStyle = FormBorderStyle.FixedSingle;
}
}

```

区分点击

请使用 `SystemInformation` 属性和 `Timer` 组件来处理 `MouseDown` 事件和确定点击之间的位置和时间跨度。根据是否发生单击或双击执行相应的操作。下面的代码示例演示如何实现这一点。

C#

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;

```

```
using System.Windows.Forms;

namespace project
{
    public partial class Form2 : Form
    {
        private DateTime _lastClick;
        private bool _inDoubleClick;
        private Rectangle _doubleClickArea;
        private TimeSpan _doubleClickMaxTime;
        private Action _doubleClickAction;
        private Action _singleClickAction;
        private Timer _clickTimer;

        public Form2()
        {
            InitializeComponent();
            _doubleClickMaxTime =
TimeSpan.FromMilliseconds(SystemInformation.DoubleClickTime);

            _clickTimer = new Timer();
            _clickTimer.Interval = SystemInformation.DoubleClickTime;
            _clickTimer.Tick += ClickTimer_Tick;

            _singleClickAction = () => MessageBox.Show("Single clicked");
            _doubleClickAction = () => MessageBox.Show("Double clicked");
        }

        private void Form2_MouseDown(object sender, MouseEventArgs e)
        {
            if (_inDoubleClick)
            {
                _inDoubleClick = false;

                TimeSpan length = DateTime.Now - _lastClick;

                // If double click is valid, respond
                if (_doubleClickArea.Contains(e.Location) && length <
_doubleClickMaxTime)
                {
                    _clickTimer.Stop();
                    _doubleClickAction();
                }
            }

            return;
        }

        // Double click was invalid, restart
        _clickTimer.Stop();
        _clickTimer.Start();
        _lastClick = DateTime.Now;
        _inDoubleClick = true;
        _doubleClickArea = new Rectangle(e.Location -
(SystemInformation.DoubleClickSize / 2),

```

```
SystemInformation.DoubleClickSize);  
}  
  
private void ClickTimer_Tick(object sender, EventArgs e)  
{  
    // Clear double click watcher and timer  
    _inDoubleClick = false;  
    _clickTimer.Stop();  
  
    _singleClickAction();  
}  
}  
}
```

另请参阅

- [使用鼠标的概述 \(Windows 窗体 .NET\)](#)
- [使用鼠标事件 \(Windows 窗体 .NET\)](#)
- [管理鼠标指针 \(Windows 窗体 .NET\)](#)
- [如何模拟鼠标事件 \(Windows 窗体 .NET\)](#)
- [Control.Click](#)
- [Control.MouseDown](#)
- [Control.SetStyle](#)

管理鼠标指针 (Windows 窗体 .NET)

项目 · 2025/01/30

鼠标 指针 (有时称为光标) 是一个位图，用于指定屏幕上的焦点，以便使用鼠标进行用户输入。本主题概述了 Windows 窗体中的鼠标指针，并介绍了修改和控制鼠标指针的一些方法。

访问鼠标指针

鼠标指针由 [Cursor](#) 类表示，每个 [Control](#) 都有一个 [Control.Cursor](#) 属性，该属性指定该控件的指针。[Cursor](#) 类包含描述指针的属性，例如 [Position](#) 和 [HotSpot](#) 属性，以及可以修改指针外观的方法，例如 [Show](#)、[Hide](#) 和 [DrawStretched](#) 方法。

以下示例在光标位于按钮上时隐藏游标：

C#

```
private void button1_MouseEnter(object sender, EventArgs e) =>
    Cursor.Hide();

private void button1_MouseLeave(object sender, EventArgs e) =>
    Cursor.Show();
```

控制鼠标指针

有时，你可能希望限制可以使用鼠标指针的区域或更改鼠标的位置。可以使用 [Cursor](#) 的 [Position](#) 属性获取或设置鼠标的当前位置。此外，还可以限制鼠标指针可用于设置 [Clip](#) 属性的区域。默认情况下，剪辑区域是整个屏幕。

以下示例在单击鼠标指针时将鼠标指针放在两个按钮之间：

C#

```
private void button1_Click(object sender, EventArgs e) =>
    Cursor.Position = PointToScreen(button2.Location);

private void button2_Click(object sender, EventArgs e) =>
    Cursor.Position = PointToScreen(button1.Location);
```

更改鼠标指针

更改鼠标指针是向用户提供反馈的重要方式。例如，可以在 [MouseEnter](#) 和 [MouseLeave](#) 事件的处理程序中修改鼠标指针，以告知用户计算发生并限制控件中的用户交互。有时，鼠标指针将因系统事件而更改，例如当应用程序参与拖放操作时。

更改鼠标指针的主要方法是将控件的 [Control.Cursor](#) 或 [DefaultCursor](#) 属性设置为新的 [Cursor](#)。有关更改鼠标指针的示例，请参阅 [Cursor](#) 类中的代码示例。此外，[Cursors](#) 类为许多不同类型的指针（例如类似于手的指针）公开一组 [Cursor](#) 对象。

下面的示例将按钮的鼠标指针的光标更改为手形形状：

C#

```
button2.Cursor = System.Windows.Forms.Cursors.Hand;
```

若要在鼠标指针位于控件上方时显示等待指针（沙漏状），请使用 [Control](#) 类的 [UseWaitCursor](#) 属性。

另请参阅

- [使用鼠标的概述 \(Windows 窗体 .NET\)](#)
- [使用鼠标事件 \(Windows 窗体 .NET\)](#)
- [如何区分单击和双击 \(Windows 窗体 .NET\)](#)
- [如何模拟鼠标事件 \(Windows 窗体 .NET\)](#)
- [System.Windows.Forms.Cursor](#)
- [Cursor.Position](#)

如何模拟鼠标事件 (Windows 窗体 .NET)

项目 • 2025/01/30

在 Windows 窗体中模拟鼠标事件并不像模拟键盘事件那样简单。Windows 窗体不提供帮助程序类来移动鼠标和调用鼠标单击操作。控制鼠标的唯一选项是使用本机 Windows 方法。如果使用的是自定义控件或窗体，则可以模拟鼠标事件，但不能直接控制鼠标。

事件

大多数事件都有一个相应的触发方法，命名模式为 `On`，后接 `EventName`，例如 `OnMouseMove`。此选项只能在自定义控件或窗体中使用，因为这些方法受到保护，并且不能从控件或窗体的上下文外部访问。使用 `OnMouseMove` 等方法的缺点是它实际上无法控制鼠标或与控件交互，因此只会引发关联的事件。例如，如果你想模拟将鼠标悬停在 `ListBox` 中的某一项上，则 `OnMouseMove` 和 `ListBox` 不会以光标下的突出显示的项的方式直观地作出反应。

这些受保护的方法可用于模拟鼠标事件。

- `OnMouseDown`
- `OnMouseEnter`
- `OnMouseHover`
- `OnMouseLeave`
- `OnMouseMove`
- `OnMouseUp`
- `OnMouseWheel`
- `OnMouseClick`
- `OnMouseDoubleClick`

有关这些事件的详细信息，请参阅[使用鼠标事件 \(Windows 窗体 .NET\)](#)

调用单击

考虑到大多数控件在单击时会执行某些操作，例如按钮调用用户代码或多选框更改选中状态，Windows 窗体提供了一种简单的方法来触发此类单击操作。某些控件（如组合框）在单击时没有任何特殊反应，模拟单击对控件没有任何效果。

PerformClick

[System.Windows.Forms.IButtonControl](#) 接口提供模拟单击控件的 [PerformClick](#) 方法。[System.Windows.Forms.Button](#) 和 [System.Windows.Forms.LinkLabel](#) 控件都实现此接口。

C#

```
button1.PerformClick();
```

InvokeClick

对于窗体的自定义控件，使用 [InvokeOnClick](#) 方法模拟鼠标单击。这是一种受保护的方法，只能从窗体或派生的自定义控件中调用。

例如，以下代码从 `button1` 单击复选框。

C#

```
private void button1_Click(object sender, EventArgs e)
{
    InvokeOnClick(checkBox1, EventArgs.Empty);
}
```

使用原生 Windows 方法

Windows 提供了可用于模拟鼠标移动和单击（如 [User32.dll SendInput](#) 和 [User32.dll SetCursorPos](#)）的方法。以下示例将鼠标光标移动到控件的中心：

C#

```
[DllImport("user32.dll", EntryPoint = "SetCursorPos")]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern bool SetCursorPos(int x, int y);

private void button1_Click(object sender, EventArgs e)
{
    Point position = PointToScreen(checkBox1.Location) + new
Size(checkBox1.Width / 2, checkBox1.Height / 2);
    SetCursorPos(position.X, position.Y);
}
```

另请参阅

- [使用鼠标的概述（Windows 窗体 .NET）](#)

- 使用鼠标事件 (Windows 窗体 .NET)
- 如何区分单击和双击 (Windows 窗体 .NET)
- 管理鼠标指针 (Windows 窗体 .NET)

PrintDialog 组件概述 (Windows 窗体 .NET)

项目 • 2024/12/18

Windows 窗体中的打印主要包括使用 [PrintDocument](#) 组件来让用户打印。

[PrintPreviewDialog](#) 控件、[PrintDialog](#) 和 [PageSetupDialog](#) 组件为 Windows 操作系统用户提供熟悉的图形界面。

[PrintDialog](#) 组件是一个预配置的对话框，用于选择打印机、选择要打印的页面，并确定基于 Windows 的应用程序中的其他打印相关设置。它是打印机和打印相关设置的简单解决方案，无需配置自己的对话框。你可以让用户打印其文档的许多部分：全部打印、打印所选页面范围或打印所选内容。通过依赖标准 Windows 对话框，可以创建用户立即熟悉其基本功能的应用程序。[PrintDialog](#) 组件继承自 [CommonDialog](#) 类。

通常，创建 [PrintDocument](#) 组件的新实例，并设置描述如何使用 [PrinterSettings](#) 和 [PageSettings](#) 类打印的属性。调用 [Print](#) 方法实际上打印文档。

使用组件

使用 [PrintDialog.ShowDialog](#) 方法在运行时显示对话框。此组件具有与单个打印作业 ([PrintDocument](#) 类) 或单个打印机 ([PrinterSettings](#) 类) 的设置相关的属性。其中一个可能由多个打印机共享。

显示对话框方法可帮助你向窗体添加打印对话框。[PrintDialog](#) 组件显示在 Visual Studio 中 Windows 窗体设计器底部的托盘中。

如何在运行时从 PrintDialog 捕获用户输入

可以在设计时设置与打印相关的选项。有时，你可能希望在运行时更改这些选项，这很可能是因为用户所做的选择。可以使用 [PrintDialog](#) 和 [PrintDocument](#) 组件捕获用于打印文档的用户输入。以下步骤演示如何显示文档的打印对话框：

1. 向窗体添加 [PrintDialog](#) 和 [PrintDocument](#) 组件。
2. 将 [PrintDialog](#) 的 [Document](#) 属性设置为窗体中添加的 [PrintDocument](#)。

C#

```
printDialog1.Document = printDocument1;
```

3. 使用 [ShowDialog](#) 方法显示 [PrintDialog](#) 组件。

```
C#  
  
// display show dialog and if user selects "Ok" document is printed  
if (printDialog1.ShowDialog() == DialogResult.OK)  
    printDocument1.Print();
```

4. 用户在对话框中的打印选项将被复制到 [PrintDocument](#) 组件的 [PrinterSettings](#) 属性。

如何创建打印作业

在 Windows 窗体中打印功能的基础是 [PrintDocument](#) 组件，更具体地说是 [PrintPage](#) 事件。通过编写代码来处理 [PrintPage](#) 事件，可以指定要打印的内容以及如何打印它。以下步骤演示如何创建打印作业：

1. 将 [PrintDocument](#) 组件添加到窗体。

2. 编写代码来处理 [PrintPage](#) 事件。

必须编写自己的打印逻辑。此外，必须指定要打印的材料。

作为要打印的材料，在下面的代码示例中，将在 [PrintPage](#) 事件处理程序中创建红色矩形形状中的示例图形。

```
C#  
  
private void PrintDocument1_PrintPage(object sender,  
System.Drawing.Printing.PrintPageEventArgs e) =>  
    e.Graphics.FillRectangle(Brushes.Red, new Rectangle(100, 100, 100,  
100));
```

可能还需要为 [BeginPrint](#) 和 [EndPrint](#) 事件编写代码。它将有助于引入一个整数，代表要打印的总页数，并且每打印一页时该整数会递减。

① 备注

您可以将 [PrintDialog](#) 组件添加到窗体，为用户提供简洁高效的用户界面（UI）。通过设置 [PrintDialog](#) 组件的 [Document](#) 属性，可以设置与在窗体上使用的打印文档相关的属性。

有关 Windows 窗体打印作业的详细信息，包括如何以编程方式创建打印作业，请参阅 [PrintPageEventArgs](#)。

如何完成打印作业

通常，涉及打印的字处理器和其他应用程序将提供向用户显示打印作业已完成的消息的选项。在 Windows 窗体中，可以通过处理 `PrintDocument` 组件的 `EndPrint` 事件来提供此功能。

以下过程要求你已创建一个基于 Windows 的应用程序，其中包含一个 `PrintDocument` 组件。以下过程是通过基于 Windows 的应用程序实现打印功能的标准方法。有关使用 `PrintDocument` 组件从 Windows 窗体打印的详细信息，请参阅 [如何创建打印作业](#)。

1. 设置 `PrintDocument` 组件的 `DocumentName` 属性。

```
C#
```

```
printDocument1.DocumentName = "SamplePrintApp";
```

2. 编写代码来处理 `EndPrint` 事件。

在下面的代码示例中，将显示一个消息框，指示文档已完成打印。

```
C#
```

```
private void PrintDocument1_EndPrint(object sender,  
System.Drawing.Printing.PrintEventArgs e) =>  
    MessageBox.Show(printDocument1.DocumentName + " has finished  
printing.");
```

打印多页文本文件 (Windows 窗体 .NET)

项目 • 2025/01/30

基于 Windows 的应用程序通常打印文本。 [Graphics](#) 类为设备（如屏幕或打印机）提供绘制对象（图形或文本）的方法。以下部分详细介绍了打印文本文件的过程。此方法不支持打印非纯文本文件，如 Office Word 文档或 PDF 文件。

① 备注

[TextRenderer](#) 的 [DrawText](#) 方法不支持打印。在绘制用于打印的文本时，应始终使用 [Graphics](#) 的 [DrawString](#) 方法，如以下代码示例所示。

打印文本

1. 在 Visual Studio 中，在“解决方案资源管理器”窗格中双击要从中打印的窗体。这将打开视觉设计器。
2. 在 **工具箱** 中，双击 [PrintDocument](#) 组件以将其添加到窗体。这应创建一个名称为 `printDocument1` 的 [PrintDocument](#) 组件。
3. 将 [Button](#) 添加到窗体，或使用窗体上已有的按钮。
4. 在窗体的可视化设计器中，选择该按钮。在 **属性** 窗格中，选择 **事件** 筛选器按钮，然后双击 `Click` 事件以生成事件处理程序。
5. `Click` 事件代码应可见。在事件处理程序的作用域之外，将私有字符串变量添加到名为 `stringToPrint` 的类。

C#

```
private string stringToPrint = "";
```

6. 返回 `click` 事件处理程序代码，将 [DocumentName](#) 属性设置为文档的名称。此信息将发送到打印机。接下来，读取文档文本内容并将其存储在 `stringToPrint` 字符串中。最后，调用 [Print](#) 方法以引发 [PrintPage](#) 事件。下面突出显示了 [Print](#) 方法。

C#

```

private void button1_Click(object sender, EventArgs e)
{
    string docName = "testPage.txt";
    string docPath = @"C:\";
    string fullPath = System.IO.Path.Combine(docPath, docName);

    printDocument1.DocumentName = docName;

    stringToPrint = System.IO.File.ReadAllText(fullPath);

    printDocument1.Print();
}

```

7. 返回到窗体的可视化设计器，然后选择 `PrintDocument` 组件。在 **属性** 窗格中，选择 **事件** 筛选器，然后双击 `PrintPage` 事件以生成事件处理程序。
8. 在 `PrintPage` 事件处理程序中，使用 `PrintPageEventArgs` 类的 `Graphics` 属性和文档内容来计算每页的行长度和行数。绘制完每一页后，检查它是否是最后一页，并相应地设置 `PrintPageEventArgs` 的 `HasMorePages` 属性。引发 `PrintPage` 事件，直到 `HasMorePages` 为 `false`。

在下面的代码示例中，事件处理程序用于以与窗体上使用的字体相同的字体打印“testPage.txt”文件的内容。

```

C#

private void PrintDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
    int charactersOnPage = 0;
    int linesPerPage = 0;

    // Sets the value of charactersOnPage to the number of characters
    // of stringToPrint that will fit within the bounds of the page.
    e.Graphics.MeasureString(stringToPrint, this.Font,
        e.MarginBounds.Size, StringFormat.GenericTypographic,
        out charactersOnPage, out linesPerPage);

    // Draws the string within the bounds of the page
    e.Graphics.DrawString(stringToPrint, this.Font, Brushes.Black,
        e.MarginBounds, StringFormat.GenericTypographic);

    // Remove the portion of the string that has been printed.
    stringToPrint = stringToPrint.Substring(charactersOnPage);

    // Check to see if more pages are to be printed.
    e.HasMorePages = (stringToPrint.Length > 0);
}

```

另请参阅

- [Graphics](#)
- [Brush](#)
- [PrintDialog 组件概述](#)

如何打印窗体 (Windows 窗体 .NET)

项目 · 2024/12/19

在开发过程中，通常需要打印 Windows 窗体的副本。下面的代码示例演示如何使用 `CopyFromScreen` 方法打印当前窗体的副本。

示例

要运行示例代码，请使用以下设置将两个组件添加到窗体中：

 展开表

Object	Property\Event	值
按钮	Name	Button1
	Click	Button1_Click
PrintDocument	Name	PrintDocument1
	PrintPage	PrintDocument1_PrintPage

单击 `Button1` 时，将运行以下代码。该代码从窗体创建一个 `Graphics` 对象，并将其内容保存到名为 `Bitmap` 的 `memoryImage` 变量中。会调用 `PrintDocument.Print` 方法，该方法调用 `PrintPage` 事件。打印事件处理程序在打印机页的 `memoryImage` 对象上绘制 `Graphics` 位图。当打印事件处理程序代码返回时，将打印页面。

C#

```
namespace Sample_print_win_form1
{
    public partial class Form1 : Form
    {
        Bitmap memoryImage;
        public Form1()
        {
            InitializeComponent();
        }

        private void Button1_Click(object sender, EventArgs e)
        {
            Graphics myGraphics = this.CreateGraphics();
            Size s = this.Size;
            memoryImage = new Bitmap(s.Width, s.Height, myGraphics);
            Graphics memoryGraphics = Graphics.FromImage(memoryImage);
            memoryGraphics.CopyFromScreen(this.Location.X, this.Location.Y,
```

```
0, 0, s);

        printDocument1.Print();
    }

    private void PrintDocument1_PrintPage(
        System.Object sender,
        System.Drawing.Printing.PrintPageEventArgs e)
    {
        e.Graphics.DrawImage(memoryImage, 0, 0);
    }
}
```

可靠编程

以下情况可能会导致异常：

- 你没有访问打印机的权限。
- 未安装打印机。

.NET 安全性

若要运行此代码示例，必须有权访问与计算机一起使用的打印机。

另请参阅

- [PrintDocument](#)
- [如何：使用 GDI+ 呈现图像](#)
- [如何：在 Windows 窗体中打印图形](#)

使用打印预览进行打印 (Windows 窗体 .NET)

项目 • 2024/11/05

除了打印服务之外，Windows 窗体编程中通常还提供打印预览。要将打印预览服务添加到你的应用程序有一个简单的方法，就是将 [PrintPreviewDialog](#) 控件与用于打印文件的 [PrintPage](#) 事件处理逻辑结合使用。

若要预览带有 PrintPreviewDialog 控件的文本文档

1. 在 Visual Studio 中，使用“解决方案资源管理器”窗格，并双击要从中打印的窗体。此操作后将打开可视化设计器。
2. 在“工具箱”窗格中，双击 [PrintDocument](#) 组件和 [PrintPreviewDialog](#) 组件，将其添加到窗体中。
3. 将 [Button](#) 添加到窗体，或使用窗体上已存在的按钮。
4. 在窗体的可视化设计器中，选择该按钮。在“属性”窗格中，选择“事件”筛选器按钮，然后双击 [Click](#) 事件以生成事件处理程序。
5. [Click](#) 事件代码应可见。在事件处理程序的范围之外，将两个私有字符串变量添加到名为 [documentContents](#) 和 [stringToPrint](#) 的类：

```
C#  
  
// Declare a string to hold the entire document contents.  
private string documentContents = "";  
  
// Declare a variable to hold the portion of the document that  
// is not printed.  
private string stringToPrint = "";
```

6. 返回到 [Click](#) 事件处理程序代码，为你想要打印的文档设置 [DocumentName](#) 属性，然后打开并读取文档内容到之前添加的字符串。

```
C#  
  
string docName = "testPage.txt";  
string docPath = @"C:\";  
string fullPath = System.IO.Path.Combine(docPath, docName);
```

```
printDocument1.DocumentName = docName;
stringToPrint = System.IO.File.ReadAllText(fullPath);
```

7. 就像你为了打印文件所执行的操作那样，在 [PrintPage](#) 事件处理程序中使用 [Graphics](#) 类的 [PrintPageEventArgs](#) 属性和文件内容来计算每页行数并呈现文档的内容。绘制完每一页后，检查它是否是最后一页，并相应地设置 [PrintPageEventArgs](#) 的 [HasMorePages](#) 属性。引发 [PrintPage](#) 事件，直到 [HasMorePages](#) 为 `false`。当文档已完成呈现时，将字符串重置为已呈现。此外，确保 [PrintPage](#) 事件与其事件处理方法关联。

① 备注

如果已在应用程序中实现打印，你可能已完成了步骤 5 和 6。

在下列代码示例中，事件处理程序用于打印“testPage.txt”文件的内容，所用字体与窗体上使用的字体相同。

C#

```
void PrintDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
    int charactersOnPage = 0;
    int linesPerPage = 0;

    // Sets the value of charactersOnPage to the number of characters
    // of stringToPrint that will fit within the bounds of the page.
    e.Graphics.MeasureString(stringToPrint, this.Font,
        e.MarginBounds.Size, StringFormat.GenericTypographic,
        out charactersOnPage, out linesPerPage);

    // Draws the string within the bounds of the page.
    e.Graphics.DrawString(stringToPrint, this.Font, Brushes.Black,
        e.MarginBounds, StringFormat.GenericTypographic);

    // Remove the portion of the string that has been printed.
    stringToPrint = stringToPrint.Substring(charactersOnPage);

    // Check to see if more pages are to be printed.
    e.HasMorePages = (stringToPrint.Length > 0);

    // If there are no more pages, reset the string to be printed.
    if (!e.HasMorePages)
        stringToPrint = documentContents;
}
```

8. 在窗体上，将 [Document](#) 控件的 [PrintPreviewDialog](#) 属性设置为 [PrintDocument](#) 组件。

C#

```
printPreviewDialog1.Document = printDocument1;
```

9. 调用 [ShowDialog](#) 控件上的 [PrintPreviewDialog](#) 方法。请注意下面提供的突出显示的代码，通常从按钮的 [Click](#) 事件处理方法调用 [ShowDialog](#)。调用 [ShowDialog](#) 会引发 [PrintPage](#) 事件，并将输出呈现到 [PrintPreviewDialog](#) 控件。当用户选择对话框上的打印按钮时，会再次引发 [PrintPage](#) 事件，将输出发送至打印机而不是预览对话框。因此，在步骤 4 中，该字符串会在呈现过程结尾重置。

下列代码示例显示了窗体上一个按钮的 [Click](#) 事件处理方法。此事件处理方法调用方法来读取文档，并显示打印预览对话框。

C#

```
private void Button1_Click(object sender, EventArgs e)
{
    string docName = "testPage.txt";
    string docPath = @"C:\";
    string fullPath = System.IO.Path.Combine(docPath, docName);
    printDocument1.DocumentName = docName;
    stringToPrint = System.IO.File.ReadAllText(fullPath);

    printPreviewDialog1.Document = printDocument1;

    printPreviewDialog1.ShowDialog();
}
```

另请参阅

- [PrintDialog 组件概述](#)
- [打印多页文本文件](#)
- [Windows 窗体中更加安全的打印](#)

数据绑定概述 (Windows 窗体 .NET)

项目 · 2024/12/18

在 Windows 窗体中，不仅可以绑定到传统数据源，还可以绑定到几乎包含数据的任何结构。可以绑定到一个值数组，这些值可以在运行时计算、从文件中读取或从其他控件的值中派生。

此外，可以将任何控件的任何属性绑定到数据源。在传统数据绑定中，通常将显示属性（例如 `TextBox` 控件的 `Text` 属性）绑定到数据源。使用 .NET，还可以选择通过绑定设置其他属性。您可能会使用绑定来执行以下任务：

- 设置图像控件的图形。
- 设置一个或多个控件的背景色。
- 设置控件的大小。

从本质上讲，数据绑定是设置窗体上任何控件的运行时可访问属性的自动方式。

与数据绑定相关的接口

ADO.NET 允许你创建许多不同的数据结构，以满足应用程序的绑定需求和正在使用的数据。你可能想要创建自己的类，用于在 Windows 窗体中提供或使用数据。这些对象可以提供不同级别的功能和复杂性。从基本数据绑定到提供设计时支持、错误检查、更改通知，甚至支持对数据本身所做的更改进行结构化回滚。

数据绑定接口的使用者

以下部分介绍两组接口对象。第一组接口由数据源作者在数据源上实现。数据源使用者（如 Windows 窗体控件或组件）实现这些接口。第二组接口旨在由组件作者使用。组件作者在创建支持 Windows 窗体数据绑定引擎使用的数据绑定的组件时使用这些接口。可以在与表单关联的类中实现这些接口，以启用数据绑定。每个事例都提供一个类，该类实现实现与数据的交互的接口。Visual Studio 快速应用程序开发 (RAD) 数据设计体验工具已利用此功能。

数据源作者实现的接口

Windows Forms 窗体控件实现了以下接口：

- `IList` 接口

实现 [IList](#) 接口的类可以是 [Array](#)、[ArrayList](#) 或 [CollectionBase](#)。这些是类型 [Object](#) 项的索引列表，列表必须包含同质类型，因为索引的第一项确定类型。[IList](#) 仅在运行时才可用于绑定。

① 备注

如果要创建用于与 Windows 窗体绑定的业务对象列表，应考虑使用 [BindingList<T>](#)。[BindingList](#) 是一个可扩展类，用于实现双向 Windows 窗体数据绑定所需的主接口。

- [IBindingList](#) 接口

实现 [IBindingList](#) 接口的类提供更高级别的数据绑定功能。此实现提供基本的排序功能和更改通知。当列表项更改以及列表本身发生更改时，这两者都很有用。如果计划将多个控件绑定到同一数据，则更改通知非常重要。它有助于对其中一个控件进行数据更改，以传播到其他绑定控件。

① 备注

通过 [SupportsChangeNotification](#) 属性为 [IBindingList](#) 接口启用更改通知，该属性 `true` 时引发 [ListChanged](#) 事件，指示列表已更改或列表中的项已更改。

更改的类型由 [ListChangedEventArgs](#) 参数的 [ListChangedType](#) 属性描述。因此，每当数据模型更新时，任何依赖视图（如绑定到同一数据源的其他控件）也将更新。但是，列表中包含的对象在更改时必须通知列表，以便列表可以引发 [ListChanged](#) 事件。

① 备注

[BindingList<T>](#) 提供 [IBindingList](#) 接口的泛型实现。

- [IBindingListView](#) 接口

实现 [IBindingListView](#) 接口的类提供 [IBindingList](#) 实现的所有功能，以及筛选和高级排序功能。此实现提供字符串筛选功能，并通过属性描述符与方向配对来实现多列排序。

- [IEditableObject](#) 接口

实现 [IEditableObject](#) 接口的类允许对象控制何时永久更改该对象。此实现支持 [BeginEdit](#)、[EndEdit](#) 和 [CancelEdit](#) 方法，使你能够回滚对对象所做的更改。下面是

对 `BeginEdit`、`EndEdit` 和 `CancelEdit` 方法工作原理的简要说明，以及它们如何协同操作以实现对数据更改的回滚可能性。

- `BeginEdit` 方法指示对对象进行编辑的开始。 实现此接口的对象需要在 `BeginEdit` 方法调用后存储任何更新，这样一来，如果调用 `CancelEdit` 方法，则可以放弃更新。 在数据绑定 Windows 窗体中，可以在单个编辑事务的范围内多次调用 `BeginEdit`（例如，`BeginEdit`、`BeginEdit`、`EndEdit`）。 `IEditableObject` 的实现应跟踪是否已调用 `BeginEdit` 并忽略对 `BeginEdit` 的后续调用。 由于此方法可以多次调用，因此确保后续调用是无破坏性的非常重要。 后续 `BeginEdit` 调用无法销毁已进行的更新或更改在第一次 `BeginEdit` 调用中保存的数据。
- 如果对象当前处于编辑模式，`EndEdit` 方法会将自从调用 `BeginEdit` 以来的所有更改推送到基础对象中。
- `CancelEdit` 方法放弃对对象所做的任何更改。

有关 `BeginEdit`、`EndEdit` 和 `CancelEdit` 方法的工作原理的详细信息，请参阅 [将数据保存回数据库](#)。

`DataGridView` 控制使用数据功能的这一事务概念。

- [ICancelAddNew](#) 接口

实现 `ICancelAddNew` 接口的类通常实现 `IBindingList` 接口，并允许使用 `AddNew` 方法回滚对数据源所做的添加。 如果数据源实现 `IBindingList` 接口，则还应让数据源实现 `ICancelAddNew` 接口。

- [IDataErrorInfo](#) 接口

实现 `IDataErrorInfo` 接口的类允许对象向绑定控件提供自定义错误信息：

- `Error` 属性返回常规错误消息文本（例如，“发生错误”）。
 - `Item[]` 属性返回一个字符串，其中包含列中的特定错误消息（例如，“`State` 列中的值无效”）。
- [IEnumerable](#) 接口

实现 `IEnumerable` 接口的类通常由 ASP.NET 使用。 此接口的 Windows 窗体支持只能通过 `BindingSource` 组件获得。

① 备注

[BindingSource](#) 组件将所有 [IEnumerable](#) 项复制到单独的列表中，以便进行绑定。

- [ITypedList](#) 接口

实现 [ITypedList](#) 接口的集合类提供用于控制顺序和向绑定控件公开的属性集的功能。

① 备注

实施 [GetItemProperties](#) 方法且 [PropertyDescriptor](#) 数组不为空时，数组的最后一个条目将是一个属性描述符，它描述了一个作为项列表的列表属性。

- [ICustomTypeDescriptor](#) 接口

实现 [ICustomTypeDescriptor](#) 接口的类提供有关自身动态信息。此接口类似于 [ITypedList](#)，但用于对象而不是列表。此接口由 [DataRowView](#) 用于投射底层行的架构。[CustomTypeDescriptor](#) 类提供了 [ICustomTypeDescriptor](#) 的简单实现。

① 备注

若要支持对实现 [ICustomTypeDescriptor](#) 的类型进行设计时绑定，该类型还必须实现 [IComponent](#) 并作为窗体上的实例存在。

- [IListSource](#) 接口

实现 [IListSource](#) 接口的类对非列表对象启用基于列表的绑定。 [IListSource](#) 的 [GetList](#) 方法用于从未从 [IList](#) 继承的对象返回可绑定列表。 [IListSource](#) 被 [DataSet](#) 类使用。

- [IRaiseItemChangedEvents](#) 接口

实现 [IRaiseItemChangedEvents](#) 接口的类是同时实现 [IBindingList](#) 接口的可绑定列表。该接口用于指示您的类型是否通过其 [RaisesItemChangedEvents](#) 属性引发 [ItemChanged](#) 类型的 [ListChanged](#) 事件。

① 备注

如果数据源提供属性来列出前面描述的事件转换，并且正在与 [BindingSource](#) 组件交互，则应实现 [IRaiseItemChangedEvents](#)。否则， [BindingSource](#) 还会执行属性来列出事件转换，从而导致性能降低。

- [ISupportInitialize](#) 接口

实现 [ISupportInitialize](#) 接口的组件利用批处理优化来设置属性和初始化依赖属性。

[ISupportInitialize](#) 包含两种方法：

- [BeginInit](#) 指示对象初始化正在启动。
- [EndInit](#) 表示对象初始化正在完成。

- [ISupportInitializeNotification](#) 接口

实现 [ISupportInitializeNotification](#) 接口的组件还实现 [ISupportInitialize](#) 接口。此接口允许你通知其他 [ISupportInitialize](#) 组件初始化已完成。

[ISupportInitializeNotification](#) 接口包含两个成员：

- [IsInitialized](#) 返回一个 `boolean` 值，指示组件是否已初始化。
- 当调用 [EndInit](#) 时，会发生 [Initialized](#)。

- [INotifyPropertyChanged](#) 接口

实现此接口的类是在其任何属性值发生更改时引发事件的类型。此接口旨在替换为控件的每个属性设置更改事件的模式。在 [BindingList<T>](#) 中使用时，业务对象应实现 [INotifyPropertyChanged](#) 接口，[BindingList`1](#) 会将 [PropertyChanged](#) 事件转换为 [ItemChanged](#)类型的 [ListChanged](#) 事件。

① 备注

若要在绑定客户端和数据源之间的绑定中发生更改通知，绑定数据源类型应实现 [INotifyPropertyChanged](#) 接口（首选），或者为绑定类型提供 `propertyName Changed` 事件，但不可以同时使用两种方式。

组件作者实现的接口

以下接口旨在供 Windows 窗体数据绑定引擎使用：

- [IBindableComponent](#) 接口

实现此接口的类是支持数据绑定的非控制组件。此类通过接口的 [DataBindings](#) 和 [BindingContext](#) 属性返回组件的数据绑定和绑定上下文。

① 备注

如果组件继承自 [Control](#)，则无需实现 [IBindableComponent](#) 接口。

- [ICurrencyManagerProvider](#) 接口

实现 [ICurrencyManagerProvider](#) 接口的类是一个组件，它提供自己的 [CurrencyManager](#) 来管理与此特定组件关联的绑定。[CurrencyManager](#) 属性提供对自定义 [CurrencyManager](#) 的访问权限。

① 备注

继承自 [Control](#) 的类通过其 [BindingContext](#) 属性自动管理绑定，因此需要实现 [ICurrencyManagerProvider](#) 的情况相当罕见。

Windows 窗体支持的数据源

传统上，数据绑定已在应用程序中使用，以利用存储在数据库中的数据。 使用 Windows 窗体数据绑定，只要满足某些最低要求，就可以访问来自数据库以及其他结构（如数组和集合）的数据。

要绑定到的结构

在 Windows 窗体中，你可以绑定到各种结构，从简单对象（简单绑定）到复杂列表，例如 ADO.NET 数据表（复杂绑定）。对于简单绑定，Windows 窗体支持绑定到简单对象的公开属性。Windows 窗体基于列表的绑定一般要求对象支持 [IList](#) 接口或 [IListSource](#) 接口。此外，如果要通过 [BindingSource](#) 组件进行绑定，则可以绑定到支持 [IEnumerable](#) 接口的对象。

以下列表显示了可以在 Windows 窗体中绑定的结构。

- [BindingSource](#)

[BindingSource](#) 是最常见的 Windows 窗体数据源，在数据源和 Windows 窗体控件之间执行代理。一般 [BindingSource](#) 使用模式是将控件绑定到 [BindingSource](#) 并将 [BindingSource](#) 绑定到数据源（例如 ADO.NET 数据表或业务对象）。

[BindingSource](#) 提供了启用和改进数据绑定支持级别的服务。例如，基于 Windows 窗体列表的控件（如 [DataGridView](#) 和 [ComboBox](#)）不支持直接绑定到 [IEnumerable](#) 数据源，不过，通过 [BindingSource](#) 绑定可以实现这一方案。在这种情况下，

[BindingSource](#) 会将数据源转换为 [IList](#)。

- 简单对象

Windows 窗体支持使用 [Binding](#) 类型将控件属性的数据绑定到对象实例的公共属性上。 Windows 窗体还支持绑定列表为基础的控件，例如在使用 [BindingSource](#) 时将 [ListControl](#) 绑定到对象实例。

- 数组或集合

若要充当数据源，列表必须实现 [IList](#) 接口;一个示例是 [Array](#) 类的实例的数组。 有关数组的详细信息，请参阅 [如何：创建对象数组 \(Visual Basic\)](#) 。

通常，在为数据绑定创建对象列表时，应使用 [BindingList<T>](#)。[BindingList](#) 是 [IBindingList](#) 接口的通用版本。[IBindingList](#) 接口通过添加双向数据绑定所需的属性、方法和事件来扩展 [IList](#) 接口。

- [IEnumerable](#)

Windows 窗体控件可以绑定到仅支持 [IEnumerable](#) 接口的数据源（如果它们通过 [BindingSource](#) 组件进行绑定）。

- ADO.NET 数据对象

ADO.NET 提供了许多适合绑定到的数据结构。 每一种的复杂度和精致度各不相同。

- [DataColumn](#)

[DataColumn](#) 是构成 [DataTable](#) 的基本单元，因为多个列组成一个表。 每个 [DataColumn](#) 都有一个 [DataType](#) 属性，决定列中保存的数据种类（例如，在描述汽车的表中，体现汽车品牌）。 可以将控件（如 [TextBox](#) 控件的 [Text](#) 属性）简单绑定到数据表中的列。

- [DataTable](#)

[DataTable](#) 是表（行和列）在 ADO.NET 中的表示形式。 数据表包含两个集合： [DataColumn](#)，表示给定表中数据的列（最终确定可输入到该表中的数据种类），以及表示给定表中数据的行 [DataRow](#)。 可以将控件复杂绑定到数据表中包含的信息（例如将 [DataGridView](#) 控件绑定到数据表）。 当你绑定到 [DataTable](#) 时，你实际上是绑定到表的默认视图。

- [DataView](#)

[DataView](#) 是一种针对单个数据表的特殊自定义视图，可以进行筛选或排序。 数据视图是复杂绑定控件使用的数据“快照”。 可以简单绑定或复杂绑定到数据视图中的数据，但请注意，要绑定到数据的固定“图片”，而不是干净的更新数据源。

- [DataSet](#)

`DataSet` 是数据库中数据的表、关系和约束的集合。您可以将数据集中的数据进行简单绑定或复杂绑定，但请注意，您绑定的是 `DataSet` 的默认值 `DataViewManager`（请参阅下一个项目符号）。

- [DataViewManager](#)

`DataViewManager` 是整个 `DataSet` 的自定义视图，类似于 `DataView`，但包含关系。使用 `DataViewSettings` 集合，可以为给定表 `DataViewManager` 具有的任何视图设置默认筛选器和排序选项。

数据绑定的类型

Windows 窗体可以利用两种类型的数据绑定：简单绑定和复杂绑定。每个都提供不同的优势。

[+] 展开表

数据绑定的类型	描述
简单数据绑定	控件绑定到单一数据元素的能力，例如数据集表格中某列的值。简单数据绑定是控件的典型绑定类型，例如 <code>TextBox</code> 控件或 <code>Label</code> 控件，这些控件通常只显示单个值。事实上，控件上的任何属性都可以绑定到数据库中的字段。Visual Studio 中对此功能有广泛的支持。 有关详细信息，请参阅 导航数据 和 创建简单绑定控件 (Windows 窗体 .NET) 。
复杂数据绑定	控件绑定到多个数据元素（通常数据库中有多个记录）的能力。复杂绑定也称为基于列表的绑定。支持复杂绑定的控件示例包括 <code>DataGridView</code> 、 <code>ListBox</code> 和 <code>ComboBox</code> 控件。有关复杂数据绑定的示例，请参阅 如何：将 Windows 窗体 ComboBox 或 ListBox 控件绑定到数据 。

绑定源组件

为了简化数据绑定，Windows 窗体使你可以将数据源绑定到 `BindingSource` 组件，然后将控件绑定到 `BindingSource`。可以在简单或复杂的绑定方案中使用 `BindingSource`。在一情况下，`BindingSource` 充当数据源与绑定控件之间的中介，提供更改通知货币管理和其他服务。

使用数据绑定的常见方案

几乎每个商业应用程序都使用从一种或另一种类型的数据源读取的信息，通常通过数据绑定。以下列表显示了一些最常见的方案，这些方案利用数据绑定作为数据呈现和操作的方法。

[+] 展开表

场景	描述
报告	报表提供了一种灵活的方法来显示和汇总打印文档中的数据。通常创建一个报表，该报表将数据源的选定内容打印到屏幕或打印机。常见报表包括列表、发票和摘要。项目被格式化为列表列，子项组织在每个列表项下，但你应该选择最适合数据的布局。
数据输入	输入大量相关数据或提示用户输入信息的常见方法是通过数据输入表单。用户可以使用文本框、选项按钮、下拉列表和复选框输入信息或选择选项。然后，将信息提交并存储在数据库中，其结构基于输入的信息。
母版/细节关系	主/细节应用程序是用于查看相关数据的一种格式。具体而言，在经典业务示例中，有两个数据表，即“客户”表和“订单”表，它们之间存在一种连接关系，将客户与其各自的订单相联系。有关使用两个 Windows 窗体 DataGridView 控件创建主/详细信息应用程序的详细信息，请参阅 如何：使用两个 Windows 窗体 DataGridView 控件创建主窗体/详细信息窗体
查找表	另一个常见的数据呈现/操作方案是表格查找。通常，作为较大数据显示的一部分，ComboBox 控件用于显示和操作数据。键是 ComboBox 控件中显示的数据与写入数据库的数据不同。例如，如果你有一个 ComboBox 控件显示杂货店提供的物品，你可能希望看到产品的名称（面包、牛奶、鸡蛋）。但是，为了简化数据库中的信息检索和数据库规范化，你可能会将给定订单的特定项的信息存储为项编号（#501、#603 等）。因此，在您的窗体中，ComboBox 控件中杂货商品的“友好名称”与订单中存在的相关项目编号之间存在隐式连接。这是表格查找的本质。有关详细信息，请参阅 如何：使用 Windows 窗体 BindingSource 组件创建查找表 。

另请参阅

- [Binding](#)
- [数据绑定概述](#)
- [设计出色的数据源并集成更改通知](#)
- [创建简单绑定控件 \(Windows 窗体 .NET\)](#)
- [BindingSource 组件](#)
- [如何：将 Windows 窗体 DataGridView 控件绑定到数据源](#)

设计具有更改通知的出色数据源 (Windows 窗体 .NET)

项目 • 2024/11/05

Windows 窗体数据绑定最重要的概念之一是更改通知。为确保数据源和绑定控件始终具有最新数据，必须为数据绑定添加更改通知。具体来说，你希望确保绑定控件在对其数据源进行更改时得到通知。数据源在对控件的绑定属性进行更改时得到通知。

根据数据绑定的类型，有不同类型的更改通知：

- 简单绑定，其中单个控件属性绑定到对象的单个实例。
- 基于列表的绑定，它可以包括绑定到列表中项属性的单个控件属性或绑定到对象列表的控件属性。

此外，如果正在创建要用于数据绑定的 Windows 窗体控件，必须将 `PropertyNameChanged` 模式应用于控件。将模式应用于控件后，会将对控件的绑定属性的更改传播到数据源。

简单绑定的更改通知

对于简单绑定，业务对象必须在绑定属性的值更改时提供更改通知。可以通过为业务对象的每个属性公开一个 `PropertyNameChanged` 事件来提供更改通知。同时需要使用 `BindingSource` 或首选方法将业务对象绑定到控件，在该方法中业务对象实现 `INotifyPropertyChanged` 接口并在属性值更改时引发 `PropertyChanged` 事件。使用实现 `INotifyPropertyChanged` 接口的对象时，不必使用 `BindingSource` 将对象绑定到控件。但建议使用 `BindingSource`。

基于列表的绑定的更改通知

Windows 窗体依靠绑定列表来向绑定控件提供属性更改和列表更改信息。属性更改是更改列表项属性值，列表更改是从列表中删除项或向列表添加项。因此，用于数据绑定的列表必须实现 `IBindingList`，它提供两种类型的更改通知。`BindingList<T>` 是 `IBindingList` 的通用实现，旨在与 Windows 窗体数据绑定一起使用。可以创建一个 `BindingList`，其中包含实现 `INotifyPropertyChanged` 的业务对象类型，并且该列表将自动将 `PropertyChanged` 事件转换为 `ListChanged` 事件。如果绑定列表不是 `IBindingList`，必须使用 `BindingSource` 组件将对象列表绑定到 Windows 窗体控件。`BindingSource` 组件将提供属性到列表的转换，该转换类似于 `BindingList` 的属性到列表

的转换。有关详细信息，请参阅[如何：使用 BindingSource 和 INotifyPropertyChanged 接口引发更改通知](#)。

自定义控件的更改通知

最后，在控件端，必须为每个旨在绑定到数据的属性公开一个 `PropertyNameChanged` 事件。然后将对控件属性的更改传播到绑定的数据源。有关详细信息，请参阅[应用 `PropertyNameChanged` 模式](#)。

应用 `PropertyNameChanged` 模式

下面的代码示例演示如何将 `PropertyNameChanged` 模式应用于自定义控件。在实现与 Windows 窗体数据绑定引擎一起使用的自定义控件时，请应用该模式。

C#

```
// This class implements a simple user control
// that demonstrates how to apply the propertyNameChanged pattern.
[ComplexBindingProperties("DataSource", "DataMember")]
public class CustomerControl : UserControl
{
    private DataGridView dataGridView1;
    private Label label1;
    private DateTime lastUpdate = DateTime.Now;

    public EventHandler DataSourceChanged;

    public object DataSource
    {
        get
        {
            return this.dataGridView1.DataSource;
        }
        set
        {
            if (DataSource != value)
            {
                this.dataGridView1.DataSource = value;
                OnDataSourceChanged();
            }
        }
    }

    public string DataMember
    {
        get { return this.dataGridView1.DataMember; }

        set { this.dataGridView1.DataMember = value; }
    }
}
```

```

private void OnDataSourceChanged()
{
    if (DataSourceChanged != null)
    {
        DataSourceChanged(this, new EventArgs());
    }
}

public CustomerControl()
{
    this.dataGridView1 = new System.Windows.Forms.DataGridView();
    this.label1 = new System.Windows.Forms.Label();
    this.dataGridView1.ColumnHeadersHeightSizeMode =
System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;
    this.dataGridView1.ImeMode = System.Windows.Forms.ImeMode.Disable;
    this.dataGridView1.Location = new System.Drawing.Point(100, 100);
    this.dataGridView1.Size = new System.Drawing.Size(500,500);

    this.dataGridView1.TabIndex = 1;
    this.label1.AutoSize = true;
    this.label1.Location = new System.Drawing.Point(50, 50);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(76, 13);
    this.label1.TabIndex = 2;
    this.label1.Text = "Customer List:";
    this.Controls.Add(this.label1);
    this.Controls.Add(this.dataGridView1);
    this.Size = new System.Drawing.Size(450, 250);
}
}

```

实现 INotifyPropertyChanged 接口

下面的代码示例演示如何实现 [INotifyPropertyChanged](#) 接口。在 Windows 窗体数据绑定中使用的业务对象上实现该接口。实现时，该接口将业务对象上的属性更改与绑定控件进行通信。

C#

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Runtime.CompilerServices;
using System.Windows.Forms;

// Change the namespace to the project name.
namespace binding_control_example
{

```

```
// This form demonstrates using a BindingSource to bind
// a list to a DataGridView control. The list does not
// raise change notifications. However the DemoCustomer1 type
// in the list does.
public partial class Form3 : Form
{
    // This button causes the value of a list element to be changed.
    private Button changeItemBtn = new Button();

    // This DataGridView control displays the contents of the list.
    private DataGridView customersDataGridView = new DataGridView();

    // This BindingSource binds the list to the DataGridView control.
    private BindingSource customersBindingSource = new BindingSource();

    public Form3()
    {
        InitializeComponent();

        // Set up the "Change Item" button.
        this.changeItemBtn.Text = "Change Item";
        this.changeItemBtn.Dock = DockStyle.Bottom;
        this.changeItemBtn.Height = 100;
        //this.changeItemBtn.Click +=
        //    new EventHandler(changeItemBtn_Click);
        this.Controls.Add(this.changeItemBtn);

        // Set up the DataGridView.
        customersDataGridView.Dock = DockStyle.Top;
        this.Controls.Add(customersDataGridView);

        this.Size = new Size(400, 200);
    }

    private void Form3_Load(object sender, EventArgs e)
    {
        this.Top = 100;
        this.Left = 100;
        this.Height = 600;
        this.Width = 1000;

        // Create and populate the list of DemoCustomer objects
        // which will supply data to the DataGridView.
        BindingList<DemoCustomer1> customerList = new ();
        customerList.Add(DemoCustomer1.CreateNewCustomer());
        customerList.Add(DemoCustomer1.CreateNewCustomer());
        customerList.Add(DemoCustomer1.CreateNewCustomer());

        // Bind the list to the BindingSource.
        this.customersBindingSource.DataSource = customerList;

        // Attach the BindingSource to the DataGridView.
        this.customersDataGridView.DataSource =
            this.customersBindingSource;
    }
}
```

```

// Change the value of the CompanyName property for the first
// item in the list when the "Change Item" button is clicked.
void changeItemBtn_Click(object sender, EventArgs e)
{
    // Get a reference to the list from the BindingSource.
    BindingList<DemoCustomer1>? customerList =
        this.customersBindingSource.DataSource as
    BindingList<DemoCustomer1>;

    // Change the value of the CompanyName property for the
    // first item in the list.
    customerList[0].CustomerName = "Tailspin Toys";
    customerList[0].PhoneNumber = "(708)555-0150";
}

}

// This is a simple customer class that
// implements the IPropertyChanged interface.
public class DemoCustomer1 : INotifyPropertyChanged
{
    // These fields hold the values for the public properties.
    private Guid idValue = Guid.NewGuid();
    private string customerNameValue = String.Empty;
    private string phoneNumberValue = String.Empty;

    public event PropertyChangedEventHandler PropertyChanged;

    // This method is called by the Set accessor of each property.
    // The CallerMemberName attribute that is applied to the optional
    propertyName
        // parameter causes the property name of the caller to be
        substituted as an argument.
    private void NotifyPropertyChanged([CallerMemberName] String
propertyName = "")
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
        }
    }

    // The constructor is private to enforce the factory pattern.
    private DemoCustomer1()
    {
        customerNameValue = "Customer";
        phoneNumberValue = "(312)555-0100";
    }

    // This is the public factory method.
    public static DemoCustomer1 CreateNewCustomer()
    {
        return new DemoCustomer1();
    }
}

```

```
}

// This property represents an ID, suitable
// for use as a primary key in a database.
public Guid ID
{
    get
    {
        return this.idValue;
    }
}

public string CustomerName
{
    get
    {
        return this.customerNameValue;
    }

    set
    {
        if (value != this.customerNameValue)
        {
            this.customerNameValue = value;
            NotifyPropertyChanged();
        }
    }
}

public string PhoneNumber
{
    get
    {
        return this.phoneNumberValue;
    }

    set
    {
        if (value != this.phoneNumberValue)
        {
            this.phoneNumberValue = value;
            NotifyPropertyChanged();
        }
    }
}
}
```

同步绑定

在 Windows 窗体中实现数据绑定期间，多个控件会绑定到同一数据源。在某些情况下，可能需要采取额外的步骤来确保控件的绑定属性之间，以及它们和数据源之间保持同步。

在两种情况下，需要执行以下步骤：

- 数据源未实现 [IBindingList](#)，因此生成类型为 [ItemChanged](#) 的 [ListChanged](#) 事件。
- 数据源实现了 [IEditableObject](#)。

在前一种情况下，请使用 [BindingSource](#) 将数据源绑定到控件。在后一种情况下，请使用 [BindingSource](#) 并处理 [BindingComplete](#) 事件，然后在相关的 [BindingManagerBase](#) 上调用 [EndCurrentEdit](#)。

有关实现此概念的详细信息，请参阅 [BindingComplete API 参考页](#)。

另请参阅

- [数据绑定](#)
- [BindingSource](#)
- [INotifyPropertyChanged](#)
- [BindingList<T>](#)

导航数据 (Windows 窗体 .NET)

项目 · 2025/01/30

在数据源中导航记录的最简单方法是将 `BindingSource` 组件绑定到数据源，然后将控件绑定到 `BindingSource`。然后，可以使用 `BindingSource` 的内置导航方法，例如 `MoveNext`、`MoveLast`、`MovePrevious` 和 `MoveFirst`。使用这些方法可以适当地调整 `BindingSource` 的 `Position` 和 `Current` 属性。还可以通过设置 `Position` 属性来查找记录并将其设置为当前记录。

递增数据源中的记录位置

将绑定数据的 `BindingSource` 的 `Position` 属性设置为可转到必需记录位置的记录位置。以下示例演示如何在选择 `nextButton` 时，使用 `BindingSource` 的 `MoveNext` 方法，递增 `Position` 属性。`BindingSource` 与数据集 `Northwind` 的 `Customers` 表相关联。

C#

```
private void nextButton_Click(object sender, System.EventArgs e)
{
    this.customersBindingSource.MoveNext();
}
```

① 备注

将 `Position` 属性设置为超出第一条或最后一条记录的值不会导致错误，因为 Windows 窗体不会将位置设置为列表边界之外的值。如果必须知道是否已超过第一条或最后一条记录，请包含用于测试是否会超过数据元素计数的逻辑。

检查是否已超过第一条或最后一条记录

为 `PositionChanged` 事件创建事件处理程序。在处理程序中，可以测试建议的位置值是否已超过实际数据元素计数。

以下示例演示如何测试是否已到达最后一个数据元素。在该示例中，如果位于最后一个元素，则窗体上的“下一个”按钮处于禁用状态。

C#

```
void customersBindingSource_PositionChanged(object sender, EventArgs e)
{
```

```
    if (customersBindingSource.Position == customersBindingSource.Count - 1)
        nextButton.Enabled = false;
    else
        nextButton.Enabled = true;
}
```

① 备注

请注意，如果在代码中更改导航的列表，则应重新启用“下一个”按钮，以便用户可以浏览整个新列表。此外，请注意，你正在使用的特定 [BindingSource](#) 的上述 [PositionChanged](#) 事件需要与其事件处理方法关联。

查找记录并将其设置为当前项

找到要设置为当前项的记录。如果数据源实现 [IBindingList](#)，请使用 [BindingSource](#) 的 [Find](#) 方法，如示例中所示。实现 [IBindingList](#) 的数据源的一些示例是 [BindingList<T>](#) 和 [DataView](#)。

C#

```
void findButton_Click(object sender, EventArgs e)
{
    int foundIndex = customersBindingSource.Find("CustomerID", "ANTON");
    customersBindingSource.Position = foundIndex;
}
```

确保子表中的选定行保持在正确的位置

当在 Windows 窗体中使用数据绑定时，将显示父/子视图或母版/详细视图中的数据。它是一种数据绑定情境，其中来自同一源的数据被显示于两个控件中。更改一个控件中的选定内容会导致第二个控件中显示的数据发生更改。例如，第一个控件可能包含客户列表，第二个控件包含与第一个控件中所选客户相关的订单列表。

在父/子视图中显示数据时，可能需要执行额外的步骤，以确保子表中当前选定的行不会重置为表的第一行。为此，你必须缓存子表位置，并在父表更改后重置它。通常，子表重置发生在父表行中的字段首次更改时。

缓存当前子表位置

1. 声明一个整数变量来存储子表位置，以及用于存储是否缓存子表位置的布尔变量。

C#

```
private int cachedPosition = -1;
private bool cacheChildPosition = true;
```

2. 为绑定的 [CurrencyManager](#) 处理 [ListChanged](#) 事件并检查 [Reset](#) 的 [ListChangedType](#)。
3. 检查 [CurrencyManager](#) 的当前位置。如果它大于列表中的第一个条目（通常为 0），请将其保存到变量。

C#

```
void relatedCM_ListChanged(object sender, ListChangedEventArgs e)
{
    // Check to see if this is a caching situation.
    if (cacheChildPosition && cachePositionCheckBox.Checked)
    {
        // If so, check to see if it is a reset situation, and the
        // current
        // position is greater than zero.
        CurrencyManager relatedCM = sender as CurrencyManager;
        if (e.ListChangedType == ListChangedType.Reset &&
relatedCM.Position > 0)

            // If so, cache the position of the child table.
            cachedPosition = relatedCM.Position;
    }
}
```

4. 为父货币管理器处理父列表的 [CurrentChanged](#) 事件。在处理程序中，设置布尔值以指示它不是缓存方案。如果发生 [CurrentChanged](#)，则对父项的更改是列表位置的更改，而不是项值更改。

C#

```
void bindingSource1_CurrentChanged(object sender, EventArgs e)
{
    // If the CurrentChanged event occurs, this is not a caching
    // situation.
    cacheChildPosition = false;
}
```

重置子表位置

1. 为子表绑定的 [CurrencyManager](#) 处理 [PositionChanged](#) 事件。

2. 将子表位置重置为在上一过程中保存的缓存位置。

C#

```
void relatedCM_PositionChanged(object sender, EventArgs e)
{
    // Check to see if this is a caching situation.
    if (cacheChildPosition && cachePositionCheckBox.Checked)
    {
        CurrencyManager relatedCM = sender as CurrencyManager;

        // If so, check to see if the current position is
        // not equal to the cached position and the cached
        // position is not out of bounds.
        if (relatedCM.Position != cachedPosition && cachedPosition
            > 0 && cachedPosition < relatedCM.Count)
        {
            relatedCM.Position = cachedPosition;
            cachedPosition = -1;
        }
    }
}
```

若要测试代码示例，请执行以下步骤：

1. 运行示例。
2. 请确保“缓存并重置位置”复选框处于选中状态。
3. 选择“**清除父字段**”按钮以更改父表中的一个字段。请注意，子表中的选定行不会更改。
4. 关闭并重新运行示例。需要再次运行它，因为重置行为仅在父行的第一次更改上发生。
5. 清除**缓存并重置位置**的复选框。
6. 选择“**清除父字段**”按钮。请注意，子表中的选定行将更改为第一行。

另请参阅

- [数据绑定概述](#)
- [Windows 窗体支持的数据源](#)
- [Windows 窗体数据绑定中的更改通知](#)
- [如何将多个控件同步到同一数据源](#)
- [BindingSource 组件](#)

创建简单绑定控件 (Windows 窗体 .NET)

项目 • 2024/11/05

使用简单数据绑定，可以在窗体上的控件中显示单个数据元素，例如数据集表中的列值。可以将控件的任何属性简单绑定到数据值。

简单绑定控件

1. [连接到数据源](#)。
2. 在 Visual Studio 中，选择窗体上的控件，然后显示“属性”窗口。
3. 展开 DataBindings 属性。

绑定的属性显示在 DataBindings 属性下。例如，在大多数控件中，经常绑定 Text 属性。

4. 如果你想绑定的属性不是经常绑定的属性，可以选择“高级”框中的“省略号”按钮 () 以显示“**格式设置和高级绑定**”对话框和该控件的属性的完整列表。
5. 选择要绑定的属性，然后选择“绑定”下的下拉箭头。此时将显示可用数据源的列表。
6. 展开要绑定到的数据源，直到找到所需的单个数据元素。例如，如果你正在绑定到数据集表中的列值，请展开该数据集的名称，然后展开表名以显示列名。
7. 选择要绑定到的元素的名称。
8. 如果你在“格式设置和高级绑定”对话框中，选择“确定”可返回“属性”窗口。
9. 如果要绑定控件的更多属性，请重复步骤 3 到 7。

① 备注

由于简单绑定控件仅显示单个数据元素，因此在具有简单绑定控件的 Windows 窗体中包含导航逻辑是一项典型操作。

创建绑定控件并设置显示数据的格式

使用 Windows 窗体数据绑定，可以通过使用“格式设置和高级绑定”对话框来设置数据绑定控件中显示的数据的格式。

1. [连接到数据源](#)。
 2. 在 Visual Studio 中，选择窗体上的控件，然后打开“属性”窗口。
 3. 展开 DataBindings 属性，然后在“高级”框中，单击省略号按钮 (...) 以显示“**格式设置和高级绑定**”对话框，该对话框具有该控件的属性的完整列表。
 4. 选择要绑定的属性，然后选择“绑定”箭头。
此时将显示可用数据源的列表。
 5. 展开要将属性绑定到的数据源，直到找到所需的单个数据元素。
- 例如，如果你正在绑定到数据集表中的列值，请展开该数据集的名称，然后展开表名以显示列名。
6. 选择要绑定到的元素的名称。
 7. 在“格式类型”框中，选择要应用于控件中显式的数据的格式。

在任何情况下，如果数据源包含 `DBNull`，都可以指定控件中显示的值。否则，选项会略有不同，具体取决于你选择的格式类型。下表显示了格式类型和选项。

[\[+\] 展开表](#)

格式	格式选项
类型	
无格 式	无选项。
数字	使用“小数位数”上下控件指定小数位数。
货币	使用“小数位数”上下控件指定小数位数。
日期 时间	通过选择“类型”选择框中的一项来选择日期和时间的显示方式。
科学	使用“小数位数”上下控件指定小数位数。
自定 义	指定一个自定义格式字符串。 有关详细信息，请参阅 类型格式设置 。注意：不保证自定义格式字符串在数据源和绑定控件之间成功往返。请改为处理该绑定的 <code>Parse</code> 或 <code>Format</code> 事件，并在事件处理代码中应用自定义格式设置。

8. 选择“确认”以关闭“格式设置和高级绑定”对话框并返回“属性”窗口。

另请参阅

- [Binding](#)
- [数据绑定](#)
- [Windows 窗体中的用户输入验证](#)

将多个控件同步到同一数据源 (Windows 窗体 .NET)

项目 · 2024/11/05

在 Windows 窗体中实现数据绑定期间，多个控件会绑定到同一数据源。在以下情况下，需要确保控件的绑定属性彼此保持同步，并与数据源保持同步：

- 数据源未实现 [IBindingList](#)，因此生成类型为 [ItemChanged](#) 的 [ListChanged](#) 事件。
- 数据源实现了 [IEditableObject](#)。

在前一种情况下，请使用 [BindingSource](#) 将数据源绑定到控件。在后一种情况下，请使用 [BindingSource](#) 并处理 [BindingComplete](#) 事件，然后在相关的 [BindingManagerBase](#) 上调用 [EndCurrentEdit](#)。

使用 BindingSource 绑定控件的示例

下面的代码示例演示如何使用 [BindingSource](#) 组件将三个控件（两个文本框控件和一个 [DataGridView](#) 控件）绑定到 [DataSet](#) 中的同一列。该示例演示如何处理 [BindingComplete](#) 事件。它确保更改一个文本框的文本值时，使用正确的值更新另一个文本框和 [DataGridView](#) 控件。

该示例使用 [BindingSource](#) 绑定数据源和控件。或者，可以将控件直接绑定到数据源，并从窗体的 [BindingContext](#) 检索绑定的 [BindingManagerBase](#)，然后为 [BindingManagerBase](#) 处理 [BindingComplete](#) 事件。有关绑定数据源和控件的详细信息，请参阅有关 [BindingManagerBase](#) 的 [BindingComplete](#) 事件的帮助页。

C#

```
public Form1()
{
    InitializeComponent();
    set1.Tables.Add("Menu");
    set1.Tables[0].Columns.Add("Beverages");

    // Add some rows to the table.
    set1.Tables[0].Rows.Add("coffee");
    set1.Tables[0].Rows.Add("tea");
    set1.Tables[0].Rows.Add("hot chocolate");
    set1.Tables[0].Rows.Add("milk");
    set1.Tables[0].Rows.Add("orange juice");

    // Set the data source to the DataSet.
    bindingSource1.DataSource = set1;
```

```
//Set the DataMember to the Menu table.  
bindingSource1.DataMember = "Menu";  
  
// Add the control data bindings.  
dataGridView1.DataSource = bindingSource1;  
textBox1.DataBindings.Add("Text", bindingSource1,  
    "Beverages", true, DataSourceUpdateMode.OnPropertyChanged);  
textBox2.DataBindings.Add("Text", bindingSource1,  
    "Beverages", true, DataSourceUpdateMode.OnPropertyChanged);  
bindingSource1.BindingComplete +=  
    new BindingCompleteEventHandler(bindingSource1_BindingComplete);  
}  
  
void bindingSource1_BindingComplete(object sender, BindingCompleteEventArgs  
e)  
{  
    // Check if the data source has been updated, and that no error has  
    occurred.  
    if (e.BindingCompleteContext ==  
        BindingCompleteContext.DataSourceUpdate && e.Exception == null)  
  
        // If not, end the current edit.  
        e.Binding.BindingManagerBase.EndCurrentEdit();  
}
```

另请参阅

- [设计具有更改通知的出色数据源](#)
- [数据绑定](#)
- [如何：使用 BindingSource 组件跨窗体共享绑定数据](#)

编译器警告WFO5003

项目 • 2024/12/19

从 app.manifest 中删除高 DPI 设置，并通过 Application.SetHighDpiMode API 或“ApplicationHighDpiMode”项目属性进行配置。

示例

项目包含 `<dpiAware>` 文件：

XML

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
    <!-- other settings omitted for brevity -->
    <application xmlns="urn:schemas-microsoft-com:asm.v3">
        <windowsSettings>
            <dpiAware
                xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">true</dpiAware>
        </windowsSettings>
    </application>
</assembly>
```

更正此错误

Windows 窗体应用程序应通过[应用程序配置](#)或 `Application.SetHighDpiMode` API 指定应用程序 DPI 感知。

或者，可以取消警告。有关详细信息，请参阅[如何禁止显示代码分析警告](#)。

使用 C#

先 `ApplicationConfiguration.Initialize` 调用方法 `Application.Run()`。

C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.Run(new Form1());
```

```
    }  
}
```

应用编译时会根据应用项目文件中的设置生成 `ApplicationConfiguration.Initialize` 方法。例如，参见以下 `<Application*>` 设置：

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  
  <PropertyGroup>  
    <OutputType>WinExe</OutputType>  
    <TargetFramework>net9.0-windows</TargetFramework>  
    <Nullable>enable</Nullable>  
    <UseWindowsForms>true</UseWindowsForms>  
    <ImplicitUsings>enable</ImplicitUsings>  
  
    <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>  
    <ApplicationVisualStyles>true</ApplicationVisualStyles>  
  
    <ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe  
ndering>  
      <ApplicationDefaultFont>Microsoft Sans Serif,  
8.25pt</ApplicationDefaultFont>  
  
  </PropertyGroup>  
  
</Project>
```

这些设置生成以下方法：

C#

```
[CompilerGenerated]  
internal static partial class ApplicationConfiguration  
{  
  public static void Initialize()  
  {  
    global::System.Windows.Forms.Application.EnableVisualStyles();  
  
    global::System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(f  
alse);  
  
    global::System.Windows.Forms.Application.SetHighDpiMode(HighDpiMode.SystemAw  
are);  
      global::System.Windows.Forms.Application.SetDefaultFont(new Font(new  
FontFamily("Microsoft Sans Serif"), 8.25f, (FontStyle)0, (GraphicsUnit)3));  
  }  
}
```

使用 Visual Basic

Visual Basic 在两个位置设置高 DPI 模式。 第一个属性位于项目属性中，这将影响 Visual Studio 设计器，但不会影响运行时的应用。

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net9.0-windows</TargetFramework>
  <StartupObject>Sub Main</StartupObject>
  <UseWindowsForms>true</UseWindowsForms>
  <MyType>WindowsForms</MyType>

  <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>
  <ApplicationVisualStyles>false</ApplicationVisualStyles>

  <ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe
  ndering>
    <ApplicationDefaultFont>Microsoft Sans Serif,
  8.25pt</ApplicationDefaultFont>

</PropertyGroup>

</Project>
```

第二个位置位于 Visual Basic 应用程序框架设置中。 这些位于应用程序>**下的“项目属性”**页中。 这些设置将保存到 *My Project\Application.myapp* 文件或应用程序启动事件处理程序中。

在 Visual Basic 中设置字体

但是，运行时字体在 Visual Basic 中无法通过应用设置或项目文件进行配置。 必须在事件处理程序中 `ApplyApplicationDefaults` 设置它。

VB

```
Imports Microsoft.VisualBasic.ApplicationServices

Namespace My
  Partial Friend Class MyApplication
    Private Sub MyApplication_ApplyApplicationDefaults(sender As Object,
e As ApplyApplicationDefaultsEventArgs) Handles Me.ApplyApplicationDefaults
      e.HighDpiMode = HighDpiMode.SystemAware
      e.Font = New Font("Microsoft Sans Serif", 8.25)
    End Sub
  End Class
End Namespace
```

End Class

End Namespace

编译器错误WFO5001

项目 · 2024/12/03

"System.Windows.Forms.Application.SetColorMode

(System.Windows.Forms.SystemColorMode) "仅用于评估目的，在将来的更新中可能会更改或删除。取消此诊断以继续。

-或-

"System.Windows.Forms.SystemColorMode"仅用于评估目的，在将来的更新中可能会更改或删除。取消此诊断以继续。

颜色模式功能目前是实验性的，可能会更改。生成此错误，以便你了解编写设置 Windows 窗体项目颜色模式的代码的含义。必须禁止显示此错误才能继续。有关此 API 的详细信息，请参阅 [深色模式](#)。

示例

以下示例生成WFO5001：

C#

```
namespace MyExampleProject;

static class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.SetColorMode(SystemColorMode.Dark);
        Application.Run(new Form1());
    }
}
```

更正此错误

禁止显示错误，并使用以下任一方法启用对 API 的访问：

- 在 .editorConfig 文件中设置规则的严重性。

ini

```
[*.{cs,vb}]
dotnet_diagnostic.WF05001.severity = none
```

有关编辑器配置文件的详细信息，请参阅 [用于代码分析规则的配置文件](#)。

- 将以下内容 `PropertyGroup` 添加到项目文件以禁止显示错误：

XML

```
<PropertyGroup>
    <NoWarn>$(NoWarn);WF05001</NoWarn>
</PropertyGroup>
```

- 使用 `#pragma warning disable WF05001` 指令取消代码中的错误：

C#

```
namespace MyExampleProject;

static class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        #pragma warning disable WF05001
        Application.SetColorMode(SystemColorMode.Dark);
        #pragma warning restore WF05001
        Application.Run(new Form1());
    }
}
```

编译器错误WFO5002

项目 · 2024/12/03

“System.Windows.Forms.Form.ShowAsync”仅用于评估目的，在将来的更新中可能会更改或删除。取消此诊断以继续。

使用以下任一方法时，将生成此编译器错误：

- `Form.ShowAsync`
- `Form.ShowDialogAsync`
- `TaskDialog.ShowDialogAsync`

Windows 窗体异步 API 目前是实验性的，可能会更改。生成此错误，以便你了解编写使用这些 API 的代码的含义。必须禁止显示此错误才能继续。有关此 API 的详细信息，请参阅 [异步表单](#)。

示例

以下示例生成WFO5002：

```
C#  
  
private async void button1_Click(object sender, EventArgs e)  
{  
    Form1 newDialog = new();  
    await newDialog.ShowAsync();  
}
```

更正此错误

禁止显示错误，并使用以下任一方法启用对 API 的访问：

- 在 `.editorConfig` 文件中设置规则的严重性。

```
ini  
  
[*.{cs,vb}]  
dotnet_diagnostic.WF05002.severity = none
```

有关编辑器配置文件的详细信息，请参阅 [用于代码分析规则的配置文件](#)。

- 将以下内容 `PropertyGroup` 添加到项目文件以禁止显示错误：

XML

```
<PropertyGroup>
    <NoWarn>$(NoWarn);WF05002</NoWarn>
</PropertyGroup>
```

- 使用 `#pragma warning disable WF05002` 指令取消代码中的错误：

C#

```
private async void button1_Click(object sender, EventArgs e)
{
    Form1 newDialog = new();
    #pragma warning disable WF05002
        await newDialog.ShowAsync();
    #pragma warning restore WF05002
}
```

编译器错误WFO5001

项目 · 2024/12/03

"System.Windows.Forms.Application.SetColorMode

(System.Windows.Forms.SystemColorMode) "仅用于评估目的，在将来的更新中可能会更改或删除。取消此诊断以继续。

-或-

"System.Windows.Forms.SystemColorMode"仅用于评估目的，在将来的更新中可能会更改或删除。取消此诊断以继续。

颜色模式功能目前是实验性的，可能会更改。生成此错误，以便你了解编写设置 Windows 窗体项目颜色模式的代码的含义。必须禁止显示此错误才能继续。有关此 API 的详细信息，请参阅 [深色模式](#)。

示例

以下示例生成WFO5001：

C#

```
namespace MyExampleProject;

static class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.SetColorMode(SystemColorMode.Dark);
        Application.Run(new Form1());
    }
}
```

更正此错误

禁止显示错误，并使用以下任一方法启用对 API 的访问：

- 在 .editorConfig 文件中设置规则的严重性。

ini

```
[*.{cs,vb}]
dotnet_diagnostic.WF05001.severity = none
```

有关编辑器配置文件的详细信息，请参阅 [用于代码分析规则的配置文件](#)。

- 将以下内容 `PropertyGroup` 添加到项目文件以禁止显示错误：

XML

```
<PropertyGroup>
    <NoWarn>$(NoWarn);WF05001</NoWarn>
</PropertyGroup>
```

- 使用 `#pragma warning disable WF05001` 指令取消代码中的错误：

C#

```
namespace MyExampleProject;

static class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        #pragma warning disable WF05001
        Application.SetColorMode(SystemColorMode.Dark);
        #pragma warning restore WF05001
        Application.Run(new Form1());
    }
}
```

编译器错误WFO5002

项目 · 2024/12/03

“System.Windows.Forms.Form.ShowAsync”仅用于评估目的，在将来的更新中可能会更改或删除。取消此诊断以继续。

使用以下任一方法时，将生成此编译器错误：

- `Form.ShowAsync`
- `Form.ShowDialogAsync`
- `TaskDialog.ShowDialogAsync`

Windows 窗体异步 API 目前是实验性的，可能会更改。生成此错误，以便你了解编写使用这些 API 的代码的含义。必须禁止显示此错误才能继续。有关此 API 的详细信息，请参阅 [异步表单](#)。

示例

以下示例生成WFO5002：

```
C#  
  
private async void button1_Click(object sender, EventArgs e)  
{  
    Form1 newDialog = new();  
    await newDialog.ShowAsync();  
}
```

更正此错误

禁止显示错误，并使用以下任一方法启用对 API 的访问：

- 在 `.editorConfig` 文件中设置规则的严重性。

```
ini  
  
[*.{cs,vb}]  
dotnet_diagnostic.WF05002.severity = none
```

有关编辑器配置文件的详细信息，请参阅 [用于代码分析规则的配置文件](#)。

- 将以下内容 `PropertyGroup` 添加到项目文件以禁止显示错误：

XML

```
<PropertyGroup>
    <NoWarn>$(NoWarn);WF05002</NoWarn>
</PropertyGroup>
```

- 使用 `#pragma warning disable WF05002` 指令取消代码中的错误：

C#

```
private async void button1_Click(object sender, EventArgs e)
{
    Form1 newDialog = new();
    #pragma warning disable WF05002
        await newDialog.ShowAsync();
    #pragma warning restore WF05002
}
```

编译器警告WFAC001

项目 • 2025/03/07

引入的版本：.NET 6

仅支持 `OutputType=WindowsApplication` 的项目。

库项目无法调用 Windows 窗体启动代码。仅支持将 `OutputType` 设置为 `Exe` 或 `WinExe` 的项目，这是因为只有应用程序项目规定了应用程序入口点，而应用程序启动代码必须位于其中。

① 重要

从 .NET 9 开始，此警告已更改为 [WFO0001](#)。

如何修复

删除对 `ApplicationConfiguration.Initialize` 的调用，或将项目类型更改为可执行文件。

WFAC002：不支持的属性值

项目 • 2024/12/18

项目文件中定义的相关应用程序配置值无效。以下代码片段演示了有效值：

XML

```
<PropertyGroup>
  <ApplicationVisualStyles>true</ApplicationVisualStyles>
  <ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe
  ndering>
  <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>
  <ApplicationDefaultFont>Microsoft Sans Serif,
  8.25pt</ApplicationDefaultFont>
</PropertyGroup>
```

如何修复

将无效设置更改为有效值。

编译器警告WFO5003

项目 • 2024/12/19

从 app.manifest 中删除高 DPI 设置，并通过 Application.SetHighDpiMode API 或“ApplicationHighDpiMode”项目属性进行配置。

示例

项目包含 `<dpiAware>` 文件：

XML

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
    <!-- other settings omitted for brevity -->
    <application xmlns="urn:schemas-microsoft-com:asm.v3">
        <windowsSettings>
            <dpiAware
                xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">true</dpiAware>
        </windowsSettings>
    </application>
</assembly>
```

更正此错误

Windows 窗体应用程序应通过[应用程序配置](#)或 `Application.SetHighDpiMode` API 指定应用程序 DPI 感知。

或者，可以取消警告。有关详细信息，请参阅[如何禁止显示代码分析警告](#)。

使用 C#

先 `ApplicationConfiguration.Initialize` 调用方法 `Application.Run()`。

C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.Run(new Form1());
```

```
    }  
}
```

应用编译时会根据应用项目文件中的设置生成 `ApplicationConfiguration.Initialize` 方法。例如，参见以下 `<Application*>` 设置：

XML

```
<Project Sdk="Microsoft.NET.Sdk">  
  
  <PropertyGroup>  
    <OutputType>WinExe</OutputType>  
    <TargetFramework>net9.0-windows</TargetFramework>  
    <Nullable>enable</Nullable>  
    <UseWindowsForms>true</UseWindowsForms>  
    <ImplicitUsings>enable</ImplicitUsings>  
  
    <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>  
    <ApplicationVisualStyles>true</ApplicationVisualStyles>  
  
    <ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe  
ndering>  
      <ApplicationDefaultFont>Microsoft Sans Serif,  
8.25pt</ApplicationDefaultFont>  
  
  </PropertyGroup>  
  
</Project>
```

这些设置生成以下方法：

C#

```
[CompilerGenerated]  
internal static partial class ApplicationConfiguration  
{  
  public static void Initialize()  
  {  
    global::System.Windows.Forms.Application.EnableVisualStyles();  
  
    global::System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(f  
alse);  
  
    global::System.Windows.Forms.Application.SetHighDpiMode(HighDpiMode.SystemAw  
are);  
      global::System.Windows.Forms.Application.SetDefaultFont(new Font(new  
FontFamily("Microsoft Sans Serif"), 8.25f, (FontStyle)0, (GraphicsUnit)3));  
  }  
}
```

使用 Visual Basic

Visual Basic 在两个位置设置高 DPI 模式。 第一个属性位于项目属性中，这将影响 Visual Studio 设计器，但不会影响运行时的应用。

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net9.0-windows</TargetFramework>
  <StartupObject>Sub Main</StartupObject>
  <UseWindowsForms>true</UseWindowsForms>
  <MyType>WindowsForms</MyType>

  <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>
  <ApplicationVisualStyles>false</ApplicationVisualStyles>

  <ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe
  ndering>
    <ApplicationDefaultFont>Microsoft Sans Serif,
  8.25pt</ApplicationDefaultFont>

</PropertyGroup>

</Project>
```

第二个位置位于 Visual Basic 应用程序框架设置中。 这些位于应用程序>**下的“项目属性”**页中。 这些设置将保存到 *My Project\Application.myapp* 文件或应用程序启动事件处理程序中。

在 Visual Basic 中设置字体

但是，运行时字体在 Visual Basic 中无法通过应用设置或项目文件进行配置。 必须在事件处理程序中 `ApplyApplicationDefaults` 设置它。

VB

```
Imports Microsoft.VisualBasic.ApplicationServices

Namespace My
  Partial Friend Class MyApplication
    Private Sub MyApplication_ApplyApplicationDefaults(sender As Object,
e As ApplyApplicationDefaultsEventArgs) Handles Me.ApplyApplicationDefaults
      e.HighDpiMode = HighDpiMode.SystemAware
      e.Font = New Font("Microsoft Sans Serif", 8.25)
    End Sub
  End Class
End Namespace
```

End Class

End Namespace

WFAC010：不支持的高 DPI 配置。

项目 • 2024/12/19

Windows 窗体应用程序应通过[应用程序配置](#)或 `Application.SetHighDpiMode` API 指定应用程序 DPI 感知。

ⓘ 重要

从 .NET 9 开始，此警告已更改为 [WFO0003](#)。

解决方法

使用 C#

通过在 `Program` 之前调用 `ApplicationConfiguration.Initialize` 方法，使用 `Application.Run()`。

C#

```
class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();
        Application.Run(new Form1());
    }
}
```

应用编译时会根据应用项目文件中的设置生成 `ApplicationConfiguration.Initialize` 方法。例如，参见以下 `<Application*>` 设置：

XML

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net8.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWindowsForms>true</UseWindowsForms>
    <ImplicitUsings>enable</ImplicitUsings>

    <ApplicationVisualStyles>true</ApplicationVisualStyles>

```

```
<ApplicationUseCompatibleTextRendering>false</ApplicationUseCompatibleTextRe  
ndering>  
    <ApplicationHighDpiMode>SystemAware</ApplicationHighDpiMode>  
    <ApplicationDefaultFont>Microsoft Sans Serif,  
8.25pt</ApplicationDefaultFont>  
  
</PropertyGroup>  
  
</Project>
```

这些设置生成以下方法：

C#

```
[CompilerGenerated]  
internal static partial class ApplicationConfiguration  
{  
    public static void Initialize()  
    {  
        global::System.Windows.Forms.Application.EnableVisualStyles();  
  
        global::System.Windows.Forms.Application.SetCompatibleTextRenderingDefault(f  
alse);  
  
        global::System.Windows.Forms.Application.SetHighDpiMode(HighDpiMode.SystemAw  
are);  
        global::System.Windows.Forms.Application.SetDefaultFont(new Font(new  
FontFamily("Microsoft Sans Serif"), 8.25f, (FontStyle)0, (GraphicsUnit)3));  
    }  
}
```

使用 Visual Basic

Visual Basic 此时的操作方式与 C# 略有不同。 Visual Studio 需要项目文件设置来检测应用程序设置，但你还需要在项目的属性页“应用程序”>“应用程序框架”（这会影响 *My Project\Application.myapp* 文件）或在应用程序启动事件中配置该设置。

① 重要

项目属性中无法设置字体。

以下代码示例演示如何处理 `ApplyApplicationDefaults` 事件以配置默认字体和 HighDPI 模式：

VB

```
Imports Microsoft.VisualBasic.ApplicationServices

Namespace My
    Partial Friend Class MyApplication
        Private Sub MyApplication_ApplyApplicationDefaults(sender As Object,
e As ApplyApplicationDefaultsEventArgs) Handles Me.ApplyApplicationDefaults
            e.Font = New Font("Microsoft Sans Serif", 8.25)
            e.HighDpiMode = HighDpiMode.SystemAware
        End Sub
    End Class
End Namespace
```

抑制警告

如果必须使用已过时的 API，可在代码或项目文件中禁止显示警告。

若只想抑制单个冲突，请将预处理器指令添加到源文件以禁用该规则，然后重新启用警告。

C#

```
// Disable the warning.
#pragma warning disable WFAC010

// Code that uses the API.
// ...

// Re-enable the warning.
#pragma warning restore WFAC010
```

若要禁止显示项目中的所有 WFAC010 警告，请将属性 <NoWarn> 添加到项目文件。

XML

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        ...
        <NoWarn>$(NoWarn);WFAC010</NoWarn>
    </PropertyGroup>
</Project>
```

有关详细信息，请参阅[取消警告](#)。

WFDEV001: WParam、 LParam 和 Message.Result 已过时

项目 • 2025/03/07

引入的版本: .NET 7

ⓘ 重要

本文适用于维护 Windows 窗体的人员。此警告不适用于 Windows Forms。

为了降低与不同平台上的 `IntPtr` 关联的强制转换和溢出异常的风险, Windows 窗体 SDK 不允许直接使用 `Message.WParam`、`Message.LParam` 和 `Message.Result`。使用 Windows 窗体 SDK 的 `DEBUG` 版本且引用 `WParam`、`LParam` 或 `Result` 的项目由于警告 `WFDEV001` 而无法编译。

解决方法

更新代码以根据情况使用新的内部属性, `WParamInternal`、`LParamInternal` 或 `ResultInternal`。

禁止显示警告

如果必须使用过时的 API, 可以在代码或项目文件中禁止显示警告。

使用以下任一方法禁止显示警告:

- 在 `.editorConfig` 文件中设置规则的严重性。

```
ini  
[*.{cs,vb}]  
dotnet_diagnostic.WFDEV001.severity = none
```

有关编辑器配置文件的详细信息, 请参阅 [配置文件, 了解代码分析规则](#)。

- 将以下 `PropertyGroup` 添加到项目文件:

```
XML
```

```
<PropertyGroup>
  <NoWarn>$(NoWarn);WFDEV001</NoWarn>
</PropertyGroup>
```

- 在代码中使用 `#pragma warning disable WFDEV001` 指令来取消。

有关详细信息，请参阅 [如何取消代码分析警告](#)。

WFDEV002：不应使用 DomainUpDownAccessibleObject

项目 • 2024/11/05

对 `System.Windows.Forms.DomainUpDown.DomainUpDownAccessibleObject` 的任何引用都将导致警告 WFDEV002。此警告指出，

`DomainUpDown.DomainUpDownAccessibleObject` 不再用于为 `DomainUpDown` 控件提供辅助支持。`DomainUpDown.DomainUpDownAccessibleObject` 类型从未用于公共用途。

① 备注

从 .NET 8 开始，此警告已提升为错误，无法再禁止显示该错误。有关详细信息，请参阅 [WFDEV002 过时现在显示为错误](#)。

解决方法

- 更新代码以使用 `AccessibleObject` 而不是 `DomainUpDown.DomainUpDownAccessibleObject`。
- 如果使用的是 .NET 7，则可以禁止显示警告，代码将继续编译和运行。

禁止显示警告（仅限 .NET 7）

如果必须使用已过时的 API，可在代码或项目文件中禁止显示警告。

若只想抑制单个冲突，请将预处理器指令添加到源文件以禁用该规则，然后重新启用警告。

C#

```
// Disable the warning.  
#pragma warning disable WFDEV002  
  
// Code that uses obsolete API.  
// ...  
  
// Re-enable the warning.  
#pragma warning restore WFDEV002
```

若要禁止显示项目中的所有 WFDEV002 警告，请将属性 `<NoWarn>` 添加到项目文件。

XML

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    ...
    <NoWarn>$(NoWarn);WFDEV002</NoWarn>
  </PropertyGroup>
</Project>
```

有关详细信息，请参阅[取消警告](#)。

编译器警告WFDEV003

项目 · 2025/03/06

引入的版本: .NET 7

`DomainUpDown.DomainItemAccessibleObject` 已过时。请改用 `AccessibleObject`。

对 `DomainUpDown.DomainItemAccessibleObject` 的引用 在编译时生成警告 WFDEV003。

此警告指出, `DomainItemAccessibleObject` 不再用于为 `DomainUpDown` 控件中的项提供可访问的支持。此类型从未用于公共用途。

以前, 此类型的对象提供给导航 `DomainUpDown` 控件层次结构的辅助功能工具。在 .NET 7 及更高版本中, `AccessibleObject` 类型的实例用于表示辅助功能工具 `DomainUpDown` 控件中的项。

解决方法

将 `DomainUpDown.DomainItemAccessibleObject` 的引用替换为 `AccessibleObject`。

禁止显示警告

使用以下任一方法禁止显示警告:

- 在 `.editorConfig` 文件中设置规则的严重性。

```
ini

[*.{cs,vb}]
dotnet_diagnostic.WFDEV003.severity = none
```

有关编辑器配置文件的详细信息, 请参阅 [用于代码分析规则的配置文件](#)。

- 将以下 `PropertyGroup` 添加到项目文件:

```
XML

<PropertyGroup>
  <NoWarn>$(NoWarn);WFDEV003</NoWarn>
</PropertyGroup>
```

- 在代码中使用 `#pragma warning disable WFDEV003` 指令进行抑制。

有关详细信息，请参阅[如何禁止显示代码分析警告](#)。