


MVVM 工具包简介

项目 • 2023/03/10

包 `CommunityToolkit.Mvvm` (又名 MVVM 工具包, 以前名为 `Microsoft.Toolkit.Mvvm`) 是一个现代、快速且模块化的 MVVM 库。它是 .NET 社区工具包的一部分, 围绕以下原则构建:

- **平台和运行时独立** - .NET Standard 2.0、.NET Standard 2.1 和 .NET 6  (UI Framework 不可知)
- **易于选取和使用** - 在“MVVM”之外, 对应用程序结构或编码范例 (没有严格的要求, 即灵活使用。
- **点菜** - 自由选择要使用的组件。
- **参考实现** - 精益和性能, 为基类库中包含的接口提供实现, 但缺少直接使用它们的具体类型。

MVVM 工具包由 Microsoft 维护和发布, 是 .NET Foundation 的一部分。它还由内置于 Windows 中的多个第三方应用程序 (例如 [Microsoft Store](#) ) 使用。

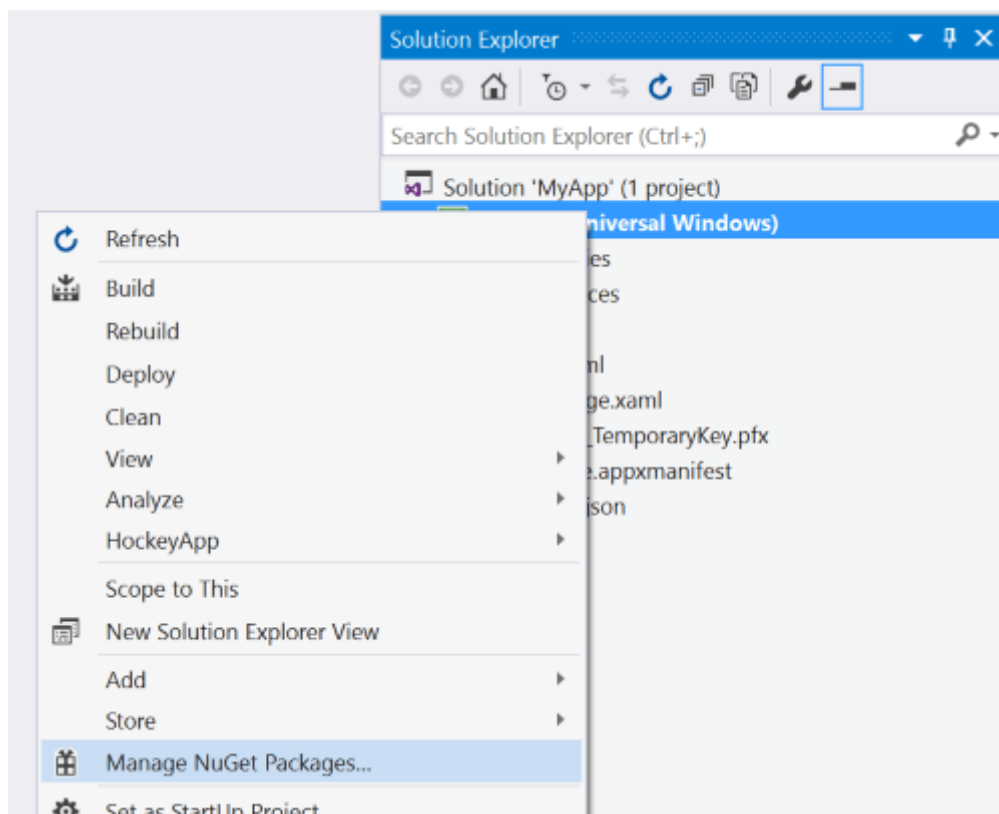
此包面向 .NET Standard, 因此可在任何应用平台上使用: UWP、WinForms、WPF、Xamarin、Uno 等;和在任何运行时上: .NET Native、.NET Core、.NET Framework 或 Mono。它在所有它们上运行。API 图面在所有情况下都是相同的, 因此非常适合生成共享库。

此外, MVVM 工具包还有一个 .NET 6 目标, 用于在 .NET 6 上运行时启用更多内部优化。在这两种情况下, 公共 API 图面完全相同, 因此 NuGet 将始终解析包的最佳版本, 而使用者无需担心哪些 API 将在其平台上可用。

入门

若要从 Visual Studio 中安装包, 请执行以下操作:

1. 在“解决方案资源管理器”中, 右键单击项目并选择“**管理 NuGet 包**”。搜索 `CommunityToolkit.Mvvm` 并安装它。



2. 添加 using 或 Imports 指令以使用新 API :

c#

```
using CommunityToolkit.Mvvm;
```

VB

```
Imports CommunityToolkit.Mvvm
```

3. 代码示例在 MVVM 工具包的其他文档页和项目的 [单元测试](#) 中提供。

何时应使用此包？

使用此包访问标准、独立、轻型类型的集合，这些类型为使用 MVVM 模式生成新式应用提供入门实现。仅这些类型通常足以让许多用户生成应用，而无需额外的外部引用。

包含的类型包括：

- **CommunityToolkit.Mvvm.ComponentModel**
 - [ObservableObject](#)
 - [ObservableRecipient](#)
 - [ObservableValidator](#)
- **CommunityToolkit.Mvvm.DependencyInjection**
 - [Ioc](#)

- **CommunityToolkit.Mvvm.Input**
 - [RelayCommand](#)
 - [RelayCommand<T>](#)
 - [AsyncRelayCommand](#)
 - [AsyncRelayCommand<T>](#)
 - [IRelayCommand](#)
 - [IRelayCommand<T>](#)
 - [IAsyncRelayCommand](#)
 - [IAsyncRelayCommand<T>](#)
- **CommunityToolkit.Mvvm.Messaging**
 - [IMessenger](#)
 - [WeakReferenceMessenger](#)
 - [StrongReferenceMessenger](#)
 - [IRecipient<TMessage>](#)
 - [MessageHandler<TRecipient, TMessage>](#)
- **CommunityToolkit.Mvvm.Messaging.Messages**
 - [PropertyChangedMessage<T>](#)
 - [RequestMessage<T>](#)
 - [AsyncRequestMessage<T>](#)
 - [CollectionRequestMessage<T>](#)
 - [AsyncCollectionRequestMessage<T>](#)
 - [ValueChangedMessage<T>](#)

此包旨在提供尽可能多的灵活性，因此开发人员可以自由选择要使用的组件。所有类型都是松散耦合的，因此只需包含你使用的内容。无需使用一系列特定的包罗万象 API 进行“全能”操作，在使用这些帮助程序生成应用时，也不需要遵循一组必需的模式。以最符合需求的方式组合这些构建基块。

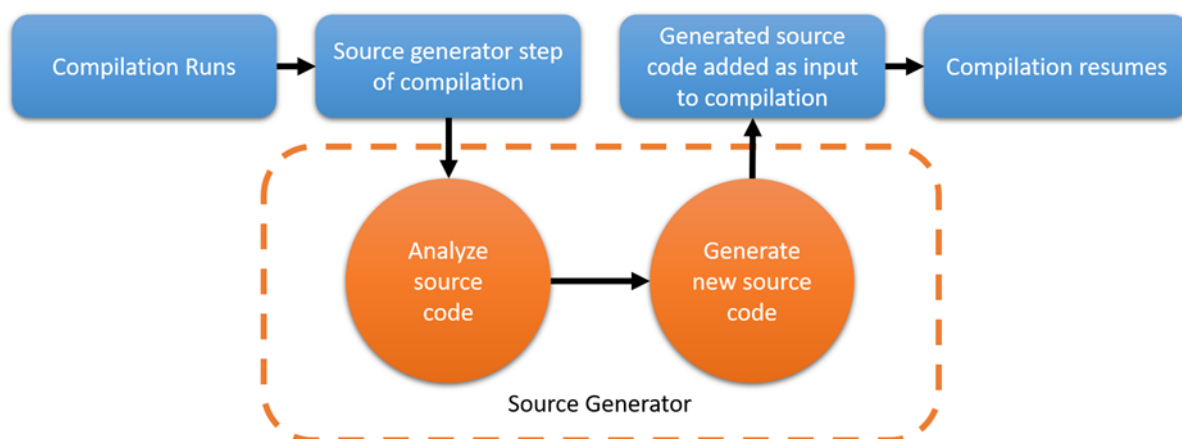
其他资源

- 查看多个 UI 框架 [的示例应用](#) [↗] () 以查看 MVVM 工具包的运行情况。
- 还可以在 [单元测试](#) [↗] 中找到更多示例。

MVVM 源生成器

项目 • 2023/03/10

从版本 8.0 开始，MVVM 工具包包含全新的 Roslyn 源生成器，有助于在使用 MVVM 体系结构编写代码时大幅减少样板。它们可简化需要设置可观察属性、命令等的方案。如果不熟悉源生成器，可 [在此处](#) 阅读有关它们的详细信息。这是一种简化的视图，介绍了它们的工作原理：



这意味着，在编写代码时，MVVM 工具包生成器现在将负责在后台为你生成其他代码，因此无需担心。然后，此代码将编译并包含在应用程序中，因此最终结果与手动编写所有额外代码完全相同，但不必执行所有这些额外工作！🔧

例如，如果不必正常设置可观察属性，则不会很好：

C#

```
private string? name;

public string? Name
{
    get => name;
    set => SetProperty(ref name, value);
}
```

你只能有一个简单的 [批注字段](#) 来表达相同的内容？

C#

```
[ObservableProperty]
private string? name;
```

如何创建命令：

C#

```
private void SayHello()
{
    Console.WriteLine("Hello");
}

private ICommand? sayHelloCommand;

public ICommand SayHelloCommand => sayHelloCommand ??= new
RelayCommand(SayHello);
```

如果我们 能有方法，别无他法呢？

C#

```
[RelayCommand]
private void SayHello()
{
    Console.WriteLine("Hello");
}
```

有了新的 MVVM 源生成器，所有这一切都是可能的，还有更多！🐙

📌 备注

源生成器可以独立于 MVVM 工具包中的其他现有功能使用，并且可以根据需要使用源生成器与以前的 API 进行混合和匹配。也就是说，你可以随意开始在新文件中使用源生成器，并最终迁移较旧的文件来减少详细程度，但不必始终在整个项目或应用程序中使用任一方法。

这些文档将准确了解 MVVM 生成器中包含的功能，以及如何使用这些功能：

- **CommunityToolkit.Mvvm.ComponentModel**
 - [ObservableProperty](#)
 - [INotifyPropertyChanged](#)
- **CommunityToolkit.Mvvm.Input**
 - [RelayCommand](#)

示例

- 查看多个 UI 框架的[示例应用](#)🔗 () 查看 MVVM 工具包的操作。
- 还可以在 [单元测试](#)🔗 中找到更多示例。

ObservableProperty 属性

文章 • 2023/03/30

类型 `ObservableProperty` 是一个属性，允许从批注字段生成可观察属性。其用途是大大减少定义可观测属性所需的样本量。

❗ 备注

若要正常工作，批注字段需要位于具有必要 `INotifyPropertyChanged` 基础结构的分部类中。如果该类型是嵌套的，则声明语法树中的所有类型也必须注释为部分。否则将导致编译错误，因为生成器将无法使用请求的可观测属性生成该类型的不同分部声明。

平台 API：`ObservableProperty`、`NotifyPropertyChangedForNotifyCanExecuteChangedFor`、`NotifyDataErrorInfo`、`NotifyPropertyChangedRecipients`、`ICommand`、`IRelayCommand`、`ObservableValidator`、`PropertyChangedMessage<T>`、`IMessenger`

工作原理

特性 `ObservableProperty` 可用于批注分部类型中的字段，如下所示：

C#

```
[ObservableProperty]
private string? name;
```

它将生成如下所示的可观测属性：

C#

```
public string? Name
{
    get => name;
    set => SetProperty(ref name, value);
}
```

它还将通过优化的实现执行此操作，因此最终结果会更快。

❗ 备注

将基于字段名称创建生成的属性的名称。生成器假定字段命名 `lowerCamel` 为、`_lowerCamel` 或 `m_lowerCamel`，并将转换为 `UpperCamel`，以遵循正确的 .NET 命名约定。生成的属性将始终具有公共访问器，但可以使用任何可见性声明字段，（`private` 建议）。

更改后运行代码

生成的代码实际上比此代码要复杂一些，原因是它还公开了一些可以实现以挂钩到通知逻辑的方法，并在即将更新属性时运行其他逻辑，并在需要更新属性后立即运行其他逻辑。也就是说，生成的代码实际上类似于以下内容：

C#

```
public string? Name
{
    get => name;
    set
    {
        if (!EqualityComparer<string?>.Default.Equals(name, value))
        {
            OnNameChanging(value);
            OnPropertyChanged();
            name = value;
            OnNameChanged(value);
            OnPropertyChanged();
        }
    }
}

partial void OnNameChanging(string? value);
partial void OnNameChanged(string? value);
```

这允许你实现这两种方法中的任何一个来注入其他代码：

C#

```
[ObservableProperty]
private string? name;

partial void OnNameChanging(string? value)
{
    Console.WriteLine($"Name is about to change to {value}");
}

partial void OnNameChanged(string? value)
{
    Console.WriteLine($"Name has changed to {value}");
}
```

你可以自由地只实现这两种方法中的一个，或者两者都不实现。如果未 (实现它们，或者只) 一个，则编译器将仅删除整个调用 ()，因此在不需要此附加功能的情况下，根本不会降低性能。

❗ 备注

生成的方法是没有实现 **的分部方法**，这意味着如果选择实现它们，则无法为它们指定显式可访问性。也就是说，这些方法的实现也应声明为仅 `partial` 方法，并且它们将始终隐式具有专用辅助功能。尝试添加显式辅助功能 (例如添加 `public` 或 `private`) 将导致错误，因为 C# 中不允许这样。

通知依赖属性

假设你有一个 `FullName` 想要在更改 `Name` 时发出通知的属性。可以使用 `NotifyPropertyChangedFor` 属性执行此操作，如下所示：

C#

```
[ObservableProperty]
[NotifyPropertyChangedFor(nameof(FullName))]
private string? name;
```

这将导致生成与以下等效的属性：

C#

```
public string? Name
{
    get => name;
    set
    {
        if (SetProperty(ref name, value))
        {
            OnPropertyChanged("FullName");
        }
    }
}
```

通知依赖命令

假设你有一个命令，其执行状态取决于此属性的值。也就是说，每当属性更改时，命令的执行状态都应失效并再次计算。换句话说， `ICommand.CanExecuteChanged` 应再次引发。可以使用 属性来实现此目的 `NotifyCanExecuteChangedFor`：

C#

```
[ObservableProperty]
[NotifyCanExecuteChangedFor(nameof(MyCommand))]
private string? name;
```

这将导致生成与以下等效的属性：

C#

```
public string? Name
{
    get => name;
    set
    {
        if (SetProperty(ref name, value))
        {
            MyCommand.NotifyCanExecuteChanged();
        }
    }
}
```

若要执行此操作，目标命令必须是某个 `IRelayCommand` 属性。

请求属性验证

如果属性是在继承自 `ObservableValidator` 的类型中声明的，则还可以使用任何验证属性对其进行批注，然后请求生成的 setter 触发该属性的验证。这可以通过 属性来实现

`NotifyDataErrorInfo`：

C#

```
[ObservableProperty]
[NotifyDataErrorInfo]
[Required]
[MinLength(2)] // Any other validation attributes too...
private string? name;
```

这将导致生成以下属性：

C#

```

public string? Name
{
    get => name;
    set
    {
        if (SetProperty(ref name, value))
        {
            ValidateProperty(value, "Value2");
        }
    }
}

```

然后，生成的 `ValidateProperty` 调用将验证 属性并更新对象的状态 `ObservableValidator`，以便 UI 组件可以对其进行响应并相应地显示任何验证错误。

❗ 备注

根据设计，只有继承自 `ValidationAttribute` 的字段属性才会转发到生成的属性。这是专门为支持数据验证方案而进行的。将忽略所有其他字段属性，因此目前无法在字段上添加其他自定义属性，并将它们也应用于生成的属性。如果需要，（例如若要控制序列化），请考虑改用传统的手动属性。

发送通知消息

如果属性是在继承自 `ObservableRecipient` 的类型中声明的，则可以使用 `NotifyPropertyChangedRecipients` 特性指示生成器也插入代码，以便为属性更改发送属性更改消息。这将允许已注册的收件人动态响应更改。也就是说，请考虑以下代码：

```

C#

[ObservableProperty]
[NotifyPropertyChangedRecipients]
private string? name;

```

这将导致生成以下属性：

```

C#

public string? Name
{
    get => name;
    set
    {
        string? oldValue = name;

```

```
        if (SetProperty(ref name, value))
        {
            Broadcast(oldValue, value);
        }
    }
}
```

然后，生成的 `Broadcast` 调用将使用当前 viewmodel 中使用的 实例向所有已注册的订阅者发送新的 `PropertyChangedMessage<T>` `IMessenger` 。

示例

- 查看多个 UI 框架 [的示例应用](#) ()，了解 MVVM 工具包的运行情况。
- 还可以在 [单元测试](#) 中找到更多示例。

RelayCommand 属性

项目 • 2023/03/10

类型 [RelayCommand](#) 是一个属性，允许为带批注的方法生成中继命令属性。其用途是完全消除在 viewmodel 中定义包装专用方法的命令所需的样本。

❗ 备注

若要正常工作，批注方法需要位于 **分部类** 中。如果类型是嵌套的，则声明语法树中的所有类型也必须注释为部分。这样做将导致编译错误，因为生成器将无法使用请求的命令生成该类型的不同部分声明。

平台 API : [RelayCommand](#)、[IRelayCommand](#)、[ICommand](#)、[IRelayCommand<T>](#)、[IAsyncRelayCommand](#)、[TaskIAsyncRelayCommand<T>](#)、[CancellationToken](#)

工作原理

该 `RelayCommand` 特性可用于批注分部类型中的方法，如下所示：

C#

```
[RelayCommand]
private void GreetUser()
{
    Console.WriteLine("Hello!");
}
```

它将生成如下所示的命令：

C#

```
private RelayCommand? greetUserCommand;

public IRelayCommand GreetUserCommand => greetUserCommand ??= new
    RelayCommand(GreetUser);
```

❗ 备注

将基于方法名称创建生成的命令的名称。生成器将使用方法名称并在末尾追加“Command”，如果存在，它将去除“On”前缀。此外，对于异步方法，在应用“Command”之前，也会删除“Async”后缀。

命令参数

该 `[RelayCommand]` 属性支持使用参数为方法创建命令。在这种情况下，它会自动将生成的命令更改为 `IRelayCommand<T>` 相反，接受相同类型的参数：

```
C#

[RelayCommand]
private void GreetUser(User user)
{
    Console.WriteLine($"Hello {user.Name}!");
}
```

这将导致以下生成的代码：

```
C#

private RelayCommand<User>? greetUserCommand;

public IRelayCommand<User> GreetUserCommand => greetUserCommand ??= new
    RelayCommand<User>(GreetUser);
```

生成的命令将自动使用参数的类型作为其类型参数。

异步命令

该 `[RelayCommand]` 命令还支持通过 `IAsyncRelayCommand` 接口 `IAsyncRelayCommand<T>` 包装异步方法。每当方法返回 `Task` 类型时，都会自动处理此情况。例如：

```
C#

[RelayCommand]
private async Task GreetUserAsync()
{
    User user = await userService.GetCurrentUserAsync();

    Console.WriteLine($"Hello {user.Name}!");
}
```

这将导致以下代码：

```
C#
```

```
private AsyncRelayCommand? greetUserCommand;

public IAsyncRelayCommand GreetUserCommand => greetUserCommand ??= new
AsyncRelayCommand(GreetUserAsync);
```

如果该方法采用参数，则生成的命令也将是泛型命令。

此方法具有 [CancellationToken](#) 一个特殊情况，因为该方法将传播到命令以启用取消。也就是说，如下所示的方法：

```
C#

[RelayCommand]
private async Task GreetUserAsync(Cancellation token)
{
    try
    {
        User user = await userService.GetCurrentUserAsync(token);

        Console.WriteLine($"Hello {user.Name}!");
    }
    catch (OperationCanceledException)
    {
    }
}
```

将导致生成的命令将令牌传递给包装方法。这使使用者只需调用 [IAsyncRelayCommand.Cancel](#) 以发出该令牌的信号，并允许挂起的操作正确停止。

启用和禁用命令

通常，能够禁用命令，然后才能使状态失效，并再次检查是否可以执行命令。为了支持这一点，该 `RelayCommand` 属性公开 `CanExecute` 属性，该属性可用于指示用于评估是否可以执行命令的目标属性或方法：

```
C#

[RelayCommand(CanExecute = nameof(CanGreetUser))]
private void GreetUser(User? user)
{
    Console.WriteLine($"Hello {user!.Name}!");
}

private bool CanGreetUser(User? user)
{
    return user is not null;
}
```

这样，`CanGreetUser` 当按钮首次绑定到 UI ((例如，按钮)) 时调用，然后在每次 `IRelayCommand.NotifyCanExecuteChanged` 调用命令时再次调用该按钮。

例如，这是命令可以绑定到属性以控制其状态的方式：

C#

```
[ObservableProperty]
[NotifyCanExecuteChangedFor(nameof(GreetUserCommand))]
private User? selectedUser;
```

XML

```
<!-- Note: this example uses traditional XAML binding syntax -->
<Button
    Content="Greet user"
    Command="{Binding GreetUserCommand}"
    CommandParameter="{Binding SelectedUser}"/>
```

在此示例中，每次生成 `SelectedUser` 属性的值更改时都会调用 `GreetUserCommand.NotifyCanExecuteChanged()` 方法。UI 具有 `Button` 控件绑定 `GreetUserCommand`，这意味着每次引发其 `CanExecuteChanged` 事件时，它都会再次调用其 `CanExecute` 方法。这将导致计算包装 `CanGreetUser` 的方法，这将基于输入实例 (`SelectedUser` UI 中绑定到属性) `null` 是否 `User` 返回按钮的新状态。这意味着每当 `SelectedUser` 发生更改时，`GreetUserCommand` 都将基于该属性是否具有值 (此方案中所需的行为) 启用。

❗ 备注

当方法或属性的 `CanExecute` 返回值发生更改时，**该命令不会自动知道**。由开发人员调用 `IRelayCommand.NotifyCanExecuteChanged` 使命令失效，并请求再次评估链接 `CanExecute` 方法，然后更新绑定到命令的控件的视觉状态。

处理并发执行

每当命令是异步的，都可以将其配置为决定是否允许并发执行。使用特性 `RelayCommand` 时，可以通过属性设置此属性 `AllowConcurrentExecutions`。默认值为 `false`，这意味着在执行挂起之前，该命令会将其状态指示为禁用状态。如果设置为 `true` 该调用，则可以将任意数量的并发调用排队。

请注意，如果命令接受取消令牌，则请求并发执行时也会取消令牌。主要区别是，如果允许并发执行，该命令将保持启用状态，并且它将启动新的请求执行，而无需等待上一个执行实际完成。

处理异步异常

异步中继命令处理异常的方式有两种不同的方法：

- **await 和重新引发 (默认)**：当命令等待调用完成时，任何异常自然都会在同一同步上下文中引发。这通常意味着引发的异常只会使应用崩溃，这与同步命令的行为一致，(引发异常也会使应用崩溃)。
- **流异常到任务计划程序**：如果命令配置为将异常流式流式传送到任务计划程序，则引发的异常不会使应用崩溃，而是通过公开 `IAsyncRelayCommand.ExecutionTask` 的异常以及浮泡 `TaskScheduler.UnobservedTaskException`而变为可用。这可实现更高级的方案(，例如，让 UI 组件绑定到任务，并根据操作结果) 显示不同的结果，但正确使用更为复杂。

默认行为是命令等待并重新引发异常。这可以通过属性进行配置

`FlowExceptionsToTaskScheduler`：

```
C#

[RelayCommand(FlowExceptionsToTaskScheduler = true)]
private async Task GreetUserAsync(Cancellation token)
{
    User user = await userService.GetCurrentUserAsync(token);

    Console.WriteLine($"Hello {user.Name}!");
}
```

在这种情况下，`try/catch` 不需要，因为异常将不再崩溃应用。请注意，这还会导致其他不相关的异常自动重新引发，因此应仔细决定如何处理每个单独的方案并相应地配置其余代码。

取消异步操作的命令

异步命令的最后一个选项是请求生成取消命令的功能。这是一个 `ICommand` 包装异步中继命令，可用于请求取消操作。此命令将自动发出信号，以反映它是否可以在任何给定时间使用。例如，如果链接命令未执行，它将报告其状态，因为它不可执行。这可以按如下所示使用：

```
C#
```



```
[RelayCommand(IncludeCancelCommand = true)]
private async Task DoWorkAsync(Cancellation_token token)
{
    // Do some long running work...
}
```

这将导致 `DoWorkCancelCommand` 也会生成属性。然后，可以绑定到其他一些 UI 组件，以便用户轻松取消挂起的异步操作。

示例

- 查看多个 UI 框架 [的示例应用](#) () 查看 MVVM 工具包的操作。
- 还可以在 [单元测试](#) 中找到更多示例。

INotifyPropertyChanged 属性

项目 • 2023/03/10

该 `INotifyPropertyChanged` 类型是一个属性，允许将 MVVM 支持代码插入现有类型。与其他相关属性 (`ObservableObject` 和 `ObservableRecipient`) 一起，其用途是支持开发人员，以防需要这些类型的相同功能，但目标类型已经从另一种类型实现。由于 C# 不允许多个继承，因此这些属性可用于让 MVVM 工具包生成器将相同的代码直接添加到这些类型中，从而避开此限制。

❗ 备注

若要正常工作，批注类型需要位于 **分部类** 中。如果类型是嵌套的，则声明语法树中的所有类型也必须注释为部分。这样做将导致编译错误，因为生成器将无法使用请求的其他代码生成该类型的不同部分声明。

❗ 备注

这些属性仅用于目标类型不能仅从等效类型继承 (等情况下使用。从 `ObservableObject`)。如果可能，则继承是建议的方法，因为它通过将重复的代码创建到最终程序集中来减小二进制大小。

平台 API：`INotifyPropertyChanged`、`ObservableObject`、`ObservableRecipient`

如何使用它们

使用这些属性中的任何一个非常简单：只需将它们添加到 **分部类**，相应的类型中的所有代码都将自动生成到该类型中。例如，请考虑以下事项：

C#

```
[INotifyPropertyChanged]
public partial class MyViewModel : SomeOtherType
{
}
```

这将在类型中 `MyViewModel` 生成完整的 `INotifyPropertyChanged` 实现，并完成其他帮助程序 (，例如 `SetProperty` 可用于减少详细程度)。下面是各种属性的简要摘要：

- `INotifyPropertyChanged`：实现接口并添加帮助程序方法来设置属性并引发事件。

- [ObservableObject](#) : 添加类型中的所有代码 `ObservableObject` 。从概念上讲，它与 `INotifyPropertyChanged` 它实现的主要区别是，它也是实现 `INotifyPropertyChanging` 的。
- [ObservableRecipient](#) : 添加类型中的所有代码 `ObservableRecipient` 。具体而言，这可以添加到从 `ObservableValidator` 中继承的类型来合并这两种类型。

示例

- 查看多个 UI 框架 [的示例应用](#) () 查看 MVVM 工具包的操作。
- 还可以在 [单元测试](#) 中找到更多示例。

ObservableObject

项目 • 2023/03/10

ObservableObject 这是通过实现 `INotifyPropertyChanged` 和 `INotifyPropertyChanging` 接口可观察的对象的基类。 它可以用作需要支持属性更改通知的各种对象的起点。

平台 API : `ObservableObject` , , `TaskNotifierTaskNotifier<T>`

工作原理

`ObservableObject` 具有以下主要功能：

- 它为和 `INotifyPropertyChanging` 公开 `PropertyChanged` 事件 `PropertyChanging` 提供了基本实现 `INotifyPropertyChanged`。
- 它提供了一系列 `SetProperty` 方法，可用于从继承 `ObservableObject` 自的类型轻松设置属性值，并自动引发相应的事件。
- 它提供了类似于 `SetPropertyAndNotifyOnCompletion` 此方法，`SetProperty` 但能够设置 `Task` 属性并在分配的任务完成后自动引发通知事件。
- 它公开了可在派生类型中重写的 `OnPropertyChanged` 和 `OnPropertyChanging` 方法，以自定义如何引发通知事件。

Simple 属性

下面是如何实现自定义属性的通知支持的示例：

```
C#  
  
public class User : ObservableObject  
{  
    private string name;  
  
    public string Name  
    {  
        get => name;  
        set => SetProperty(ref name, value);  
    }  
}
```

提供 `SetProperty<T>(ref T, T, string)` 的方法检查属性的当前值，并更新它（如果不同），然后还会自动引发相关事件。 属性名称是通过使用属性自动捕获的 `[CallerMemberName]`，因此无需手动指定要更新的属性。

包装不可观测模型

例如，使用数据库项时，常见的方案是创建一个包装的“可绑定”模型，该模型中继数据库模型的属性，并在需要时引发属性更改通知。如果想要将通知支持注入到未实现接口的 `INotifyPropertyChanged` 模型，还需要这样做。 `ObservableObject` 提供了一种专用方法，使此过程更简单。对于以下示例， `User` 是直接映射数据库表的模型，而不继承自 `ObservableObject`：

C#

```
public class ObservableUser : ObservableObject
{
    private readonly User user;

    public ObservableUser(User user) => this.user = user;

    public string Name
    {
        get => user.Name;
        set => SetProperty(user.Name, value, user, (u, n) => u.Name = n);
    }
}
```

在本例中 `SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string)`，我们使用重载。签名比上一个签名要复杂一些，这需让代码保持非常高效，即使我们无权访问上一个支持字段，就像在前面的方案中一样。我们可以详细浏览此方法签名的每个部分，以了解不同组件的角色：

- `TModel` 是一个类型参数，指示要包装的模型的类型。在本例中，这将是我们的 `User` 类。请注意，我们不需要显式指定此代码 - C# 编译器将通过调用 `SetProperty` 方法的方式自动推断这一点。
- `T` 是要设置的属性的类型。同样 `TModel`，这会自动推断。
- `T oldValue` 是第一个参数，在本例中，我们将用于 `user.Name` 传递要包装的该属性的当前值。
- `T newValue` 是要设置为属性的新值，此处我们将传递 `value`，这是属性 setter 中的输入值。
- `TModel model` 是我们正在包装的目标模型，在本例中，我们将传递存储在字段中的 `user` 实例。
- `Action<TModel, T> callback` 是一个函数，如果属性的新值不同于当前值，并且需要设置该属性，则调用该函数。此操作将由此回调函数完成，该函数接收作为目标模型的输入和要设置的新属性值。在本例中，我们只需通过执行 `u.Name = n`，将输入值 (`n Name` 分配给属性 `()`)。在这里，请务必避免从当前范围捕获值，并且只与作为回调输入的值进行交互，因为这允许 C# 编译器缓存回调函数并执行许多性能

改进。这是因为，我们不只是直接访问 `user` 此处的字段或 `value` setter 中的参数，而是只使用 lambda 表达式的输入参数。

该方法 `SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string)` 使创建这些包装属性极其简单，因为它负责检索和设置目标属性，同时提供极其紧凑 API。

① 备注

与使用 LINQ 表达式的此方法的实现相比，特别是通过类型 `Expression<Func<T>>` 参数而不是状态和回调参数实现，可以通过此方法实现的性能改进非常重要。具体而言，此版本比使用 LINQ 表达式快约 200 倍，根本不会进行任何内存分配。

处理 `Task<T>` 属性

如果属性是一个 `Task` 属性，则还必须在任务完成后引发通知事件，以便正确更新绑定。例如，若要在任务所表示的操作上显示加载指示器或其他状态信息。 `ObservableObject` 具有此方案的 API：

```
C#

public class MyModel : ObservableObject
{
    private TaskNotifier<int>? requestTask;

    public Task<int>? RequestTask
    {
        get => requestTask;
        set => SetPropertyAndNotifyOnCompletion(ref requestTask, value);
    }

    public void RequestValue()
    {
        RequestTask = WebService.LoadMyValueAsync();
    }
}
```

此处，该方法 `SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>, Task<T>, string)` 将负责更新目标字段、监视新任务（如果存在）以及在该任务完成时引发通知事件。这样，就可以只绑定到任务属性，并在其状态发生更改时收到通知。这是一种特殊类型，它 `TaskNotifier<T>` `ObservableObject` 包装目标 `Task<T>` 实例并启用此方法所需的通知逻辑。 `TaskNotifier` 如果只有常规 `Task` 类型，还可以直接使用该类型。

① 备注

该方法 `SetPropertyAndNotifyOnCompletion` 旨在替换包中 `Microsoft.Toolkit` 类型的方法 `NotifyTaskCompletion<T>`。如果使用了此类型，则只能将其替换为内部 `Task` (或 `Task<TResult>`) 属性，然后 `SetPropertyAndNotifyOnCompletion` 该方法可用于设置其值并引发通知更改。类型公开 `NotifyTaskCompletion<T>` 的所有属性都直接在实例上 `Task` 可用。

示例

- 查看多个 UI 框架的[示例应用](#) (,) 查看 MVVM Toolkit。
- 还可以在 [单元测试](#) 中找到更多示例。

ObservableRecipient

项目 • 2023/03/10

该 `ObservableRecipient` 类型是可观察对象的基类，也充当邮件的收件人。此类是一个扩展 `ObservableObject`，它还提供使用类型的 `IMessenger` 内置支持。

平台 API： `ObservableRecipient`、`ObservableObjectIMessenger`、`WeakReferenceMessenger`、`IRecipient<TMessage>PropertyChangedMessage<T>`

工作原理

该 `ObservableRecipient` 类型旨在用作也使用该 `IMessenger` 功能的 viewmodel 的基础，因为它为它提供内置支持。具体而言：

- 它有一个无参数构造函数和一个 `IMessenger` 用于依赖注入实例的构造函数。它还公开 `Messenger` 可用于在 viewmodel 中发送和接收消息的属性。如果使用无参数构造函数，则会 `WeakReferenceMessenger.Default` 将实例分配给 `Messenger` 该属性。
- 它公开用于 `IsActive` 激活/停用 viewmodel 的属性。在此上下文中，若要“激活”，则表示给定的 viewmodel 被标记为正在使用，例如。它将开始侦听已注册的消息、执行其他设置操作等。有两个相关方法，`OnActivated` 在 `OnDeactivated` 属性更改值时调用该方法。默认情况下，`OnDeactivated` 自动从所有已注册的消息中注销当前实例。为了获得最佳结果并避免内存泄漏，建议用于 `OnActivated` 向消息注册以及用于 `OnDeactivated` 执行清理操作。此模式允许多次启用/禁用 viewmodel，同时在每次停用内存泄漏风险的情况下安全地收集。默认情况下，`OnActivated` 将自动注册通过 `IRecipient<TMessage>` 接口定义的所有消息处理程序。
- 它公开一个 `Broadcast<T>(T, T, string)` 方法，该方法通过 `IMessenger` 属性中可用的 `Messenger` 实例发送 `PropertyChangedMessage<T>` 消息。这可用于轻松广播 viewmodel 属性中的更改，而无需手动检索 `Messenger` 要使用的实例。此方法由各种 `SetProperty` 方法的重载使用，该方法具有附加 `bool broadcast` 属性来指示是否也发送消息。

下面是在活动时接收 `LoggedInUserRequestMessage` 消息的 viewmodel 示例：

C#

```
public class MyViewModel : ObservableRecipient,
    IRecipient<LoggedInUserRequestMessage>
{
```



```

public void Receive(LoginUserRequestMessage message)
{
    // Handle the message here
}
}

```

在上面的示例中，`OnActivated` 使用该方法作为要调用的操作自动将实例注册为邮件收件人 `LoginUserRequestMessage`。 `IRecipient<TMessage>` 使用接口不是必需的，注册也可以手动（，即使仅使用内联 lambda 表达式）：

C#

```

public class MyViewModel : ObservableRecipient
{
    protected override void OnActivated()
    {
        // Using a method group...
        Messenger.Register<MyViewModel, LoginUserRequestMessage>(this,
(r, m) => r.Receive(m));

        // ...or a lambda expression
        Messenger.Register<MyViewModel, LoginUserRequestMessage>(this,
(r, m) =>
        {
            // Handle the message here
        });
    }

    private void Receive(LoginUserRequestMessage message)
    {
        // Handle the message here
    }
}

```

示例

- 查看多个 UI 框架的[示例应用](#) () 查看 MVVM Toolkit。
- 还可以在 [单元测试](#) 中找到更多示例。

ObservableValidator

项目 • 2023/03/10

`ObservableValidator`是实现 `INotifyDataErrorInfo`基类，支持验证向其他应用程序模块公开的属性。它还继承自 `ObservableObject`，因此它也实现 `INotifyPropertyChanged` 和 `INotifyPropertyChanging`。它可以用作需要同时支持属性更改通知和属性验证的所有对象的起点。

平台 API：`ObservableValidator`、`ObservableObject`

工作原理

`ObservableValidator` 具有以下主要功能：

- 它为 提供基本实现 `INotifyDataErrorInfo`，公开 `ErrorsChanged` 事件和其他必要的 API。
- 它提供一系列附加 `SetProperty` 重载，(基 `ObservableObject` 类) 提供的重载，这些重载提供在更新其值之前自动验证属性和引发必要事件的能力。
- 它公开了许多 `TrySetProperty` 重载，这些重载类似于 `SetProperty`，但仅在验证成功时更新目标属性，如果有任何) 进一步检查，则返回生成的错误 (。
- 它公开 `ValidateProperty` 方法，该方法可用于手动触发特定属性的验证，以防其值尚未更新，但其验证依赖于已更新的另一个属性的值。
- 它公开 `ValidateAllProperties` 方法，该方法会自动执行当前实例中所有公共实例属性的验证，前提是它们至少应用了一个 `[ValidationAttribute]` 属性。
- 它公开了一个 `ClearAllErrors` 方法，该方法在重置绑定到用户可能想要再次填充的某个表单的模型时非常有用。
- 它提供了许多构造函数，允许传递不同的参数来初始化 `ValidationContext` 将用于验证属性的实例。使用可能需要其他服务或选项才能正常工作的自定义验证属性时，这尤其有用。

简单属性

下面是如何实现支持更改通知和验证的属性的示例：

C#

```
public class RegistrationForm : ObservableValidator
{
    private string name;

    [Required]
```

```

[MinLength(2)]
[MaxLength(100)]
public string Name
{
    get => name;
    set => SetProperty(ref name, value, true);
}
}

```

此处，我们将调用 `SetProperty<T>(ref T, T, bool, string)` 由 `ObservableValidator` 公开的方法，并将附加 `bool` 参数设置为 `true` 指示我们还希望在属性值更新时验证属性。`ObservableValidator` 将使用使用应用于 属性的属性指定的所有检查，对每个新值自动运行验证。然后，其他 (组件 (如 UI 控件)) 可以与 `viewmodel` 交互，并修改其状态以反映 `viewmodel` 中当前存在的错误，方法是注册 `ErrorsChanged` 并使用 `GetErrors(string)` 方法检索已修改的每个属性的错误列表。

自定义验证方法

有时，验证属性需要 `viewmodel` 有权访问其他服务、数据或其他 API。可通过不同的方法向属性添加自定义验证，具体取决于方案和所需的灵活性级别。下面是一个示例，[\[CustomValidationAttribute\]](#) 说明如何使用 类型来指示需要调用特定方法才能对属性执行其他验证：

C#

```

public class RegistrationForm : ObservableValidator
{
    private readonly IFancyService service;

    public RegistrationForm(IFancyService service)
    {
        this.service = service;
    }

    private string name;

    [Required]
    [MinLength(2)]
    [MaxLength(100)]
    [CustomValidation(typeof(RegistrationForm), nameof(ValidateName))]
    public string Name
    {
        get => this.name;
        set => SetProperty(ref this.name, value, true);
    }

    public static ValidationResult ValidateName(string name,
        ValidationContext context)

```

```

    {
        RegistrationForm instance =
            (RegistrationForm)context.ObjectInstance;
        bool isValid = instance.service.Validate(name);

        if (isValid)
        {
            return ValidationResult.Success;
        }

        return new("The name was not validated by the fancy service");
    }
}

```

在这种情况下，我们有一个静态 `ValidateName` 方法，该方法将通过注入到 viewmodel 中的服务对 `Name` 属性执行验证。此方法接收 `name` 属性值和正在使用的 `ValidationContext` 实例，其中包含诸如 viewmodel 实例、要验证的属性的名称以及服务提供程序和—些可以使用或设置的自定义标志等内容。在本例中，我们将从验证上下文中检索 `RegistrationForm` 实例，并从那里使用注入的服务来验证属性。请注意，此验证将在其他属性中指定的验证旁边执行，因此我们可以随意组合自定义验证方法和现有验证属性，但我们喜欢。

自定义验证属性

执行自定义验证的另一种方法是实现自定义 `[ValidationAttribute]`，然后将验证逻辑插入重写 `IsValid` 的方法中。与上述方法相比，这样可以实现额外的灵活性，因为它使得在多个位置重复使用同一属性变得非常简单。

假设我们想要根据属性相对于同一 viewmodel 中另一个属性的相对值来验证该属性。第一步是定义自定义 `[GreaterThanAttribute]`，如下所示：

```

C#

public sealed class GreaterThanAttribute : ValidationAttribute
{
    public GreaterThanAttribute(string propertyName)
    {
        PropertyName = propertyName;
    }

    public string PropertyName { get; }

    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        object
            instance = validationContext.ObjectInstance,

```

```

        otherValue =
instance.GetType().GetProperty(PropertyName).GetValue(instance);

        if (((IComparable)value).CompareTo(otherValue) > 0)
        {
            return ValidationResult.Success;
        }

        return new("The current value is smaller than the other one");
    }
}

```

接下来，我们可以将此属性添加到 viewmodel 中：

```

C#

public class ComparableModel : ObservableValidator
{
    private int a;

    [Range(10, 100)]
    [GreaterThan(nameof(B))]
    public int A
    {
        get => this.a;
        set => SetProperty(ref this.a, value, true);
    }

    private int b;

    [Range(20, 80)]
    public int B
    {
        get => this.b;
        set
        {
            SetProperty(ref this.b, value, true);
            ValidateProperty(A, nameof(A));
        }
    }
}

```

在这种情况下，我们有两个数字属性，这些属性必须位于特定的范围内，并且彼此之间具有特定关系（A 需要大于 B）。我们在第一个属性上添加了新的 `[GreaterThanAttribute]`，并在它的 setter 中添加了对 `ValidateProperty` 的 B 调用，因此每当 B 更改（时，会再次验证，A 因为其验证状态取决于它）。只需在 viewmodel 中使用这两行代码即可启用此自定义验证，并且还获得了一个可重用的自定义验证属性的好处，该属性在应用程序中的其他 viewmodel 中也很有用。此方法还有助于代码模块化，因为验证逻辑现在与 viewmodel 定义本身完全分离。

示例

- 查看多个 UI 框架 [的示例应用](#) () 以查看 MVVM 工具包的运行情况。
- 还可以在 [单元测试](#) 中找到更多示例。

RelayCommand 和 RelayCommand<T>

项目 • 2023/03/10

并且 `RelayCommand` 和 `RelayCommand<T>` 是 `ICommand` 可以向视图公开方法或委托的实现。这些类型充当在 viewmodel 和 UI 元素之间绑定命令的方法。

平台 API : `RelayCommand`、`RelayCommand<T>`、`IRelayCommand` 和 `IRelayCommand<T>`

工作原理

`RelayCommand` 具有以下 `RelayCommand<T>` 主要功能：

- 它们提供接口的基本 `ICommand` 实现。
- 它们还实现 `IRelayCommand` (和 `IRelayCommand<T>`) 接口，该接口公开了 `NotifyCanExecuteChanged` 引发 `CanExecuteChanged` 事件的方法。
- 它们公开采用类似和（允许包装标准方法和 lambda 表达式）的 `Action Func<T>` 委托的构造函数。

使用 `ICommand`

下面演示如何设置简单命令：

C#

```
public class MyViewModel : ObservableObject
{
    public MyViewModel()
    {
        IncrementCounterCommand = new RelayCommand(IncrementCounter);
    }

    private int counter;

    public int Counter
    {
        get => counter;
        private set => SetProperty(ref counter, value);
    }

    public ICommand IncrementCounterCommand { get; }

    private void IncrementCounter() => Counter++;
}
```

然后，可以使用 WinUI XAML (相对 UI：

XML

```
<Page
  x:Class="MyApp.Views.MyPage"
  xmlns:viewModels="using:MyApp.ViewModels">
  <Page.DataContext>
    <viewModels:MyViewModel x:Name="ViewModel"/>
  </Page.DataContext>

  <StackPanel Spacing="8">
    <TextBlock Text="{x:Bind ViewModel.Counter, Mode=OneWay}"/>
    <Button
      Content="Click me!"
      Command="{x:Bind ViewModel.IncrementCounterCommand}"/>
  </StackPanel>
</Page>
```

绑定到 `Button` viewmodel 中包装专用 `IncrementCounter` 方法的绑定 `ICommand`。显示 `TextBlock` 属性的值 `Counter`，并在每次属性值更改时更新。

示例

- 查看多个 UI 框架的[示例应用](#) (,) 查看 MVVM Toolkit。
- 还可以在 [单元测试](#) 中找到更多示例。

AsyncRelayCommand 和 AsyncRelayCommand<T>

项目 • 2023/03/10

[AsyncRelayCommand<T>](#)以及[AsyncRelayCommand](#)扩展 [ICommand](#) 由[RelayCommand](#)异步操作提供支持的功能的实现。

平台 API : [AsyncRelayCommand](#)、[AsyncRelayCommand<T>RelayCommand](#)、[IAsyncRelayCommandIAsyncRelayCommand<T>](#)

工作原理

[AsyncRelayCommand](#) 具有以下 [AsyncRelayCommand<T>](#) 主要功能：

- 它们扩展库中包含的同步命令的功能，并支持 [Task](#) 返回委托。
- 它们可以使用附加 [CancellationTokens](#) 参数包装异步函数以支持取消，并公开属性 [CanBeCanceled](#) [IsCancellationRequested](#) 以及 [Cancel](#) 方法。
- 它们公开可用于 [ExecutionTask](#) 监视挂起操作进度的属性，以及 [IsRunning](#) 可用于检查操作完成时间的属性。这对于将命令绑定到 UI 元素（如加载指示器）特别有用。
- 它们实现和[IAsyncRelayCommandIAsyncRelayCommand<T>](#)接口，这意味着 viewmodel 可以使用它们轻松公开命令，以减少类型之间的紧密耦合。例如，这样就可以轻松地将命令替换为公开相同公共 API 图面的自定义实现（如果需要）。

使用异步命令

假设一个方案类似于示例中所述的 [RelayCommand](#) 方案，但执行异步操作的命令：

C#

```
public class MyViewModel : ObservableObject
{
    public MyViewModel()
    {
        DownloadTextCommand = new AsyncRelayCommand(DownloadText);
    }

    public IAsyncRelayCommand DownloadTextCommand { get; }

    private Task<string> DownloadText()
    {
        return WebService.LoadMyTextAsync();
    }
}
```

```
}  
}
```

使用相关的 UI 代码：

XML

```
<Page  
    x:Class="MyApp.Views.MyPage"  
    xmlns:viewModels="using:MyApp.ViewModels"  
    xmlns:converters="using:Microsoft.Toolkit.Uwp.UI.Converters">  
    <Page.DataContext>  
        <viewModels:MyViewModel x:Name="ViewModel"/>  
    </Page.DataContext>  
    <Page.Resources>  
        <converters:TaskResultConverter x:Key="TaskResultConverter"/>  
    </Page.Resources>  
  
    <StackPanel Spacing="8" xml:space="default">  
        <TextBlock>  
            <Run Text="Task status:"/>  
            <Run Text="{x:Bind  
ViewModel.DownloadTextCommand.ExecutionTask.Status, Mode=OneWay}"/>  
            <LineBreak/>  
            <Run Text="Result:"/>  
            <Run Text="{x:Bind ViewModel.DownloadTextCommand.ExecutionTask,  
Converter={StaticResource TaskResultConverter}, Mode=OneWay}"/>  
        </TextBlock>  
        <Button  
            Content="Click me!"  
            Command="{x:Bind ViewModel.DownloadTextCommand}"/>  
        <ProgressRing  
            HorizontalAlignment="Left"  
            IsActive="{x:Bind ViewModel.DownloadTextCommand.IsRunning,  
Mode=OneWay}"/>  
    </StackPanel>  
</Page>
```

单击该 `Button` 命令后，将调用命令并 `ExecutionTask` 更新。操作完成后，该属性将引发 UI 中反映的通知。在这种情况下，将显示任务状态和任务的当前结果。请注意，若要显示任务的结果，必须使用 `TaskExtensions.GetResultOrDefault` 该方法 - 这提供了对尚未完成且尚未完成的任务（未阻止线程（并可能导致死锁））的访问。

示例

- 查看多个 UI 框架的[示例应用](#) () 查看 MVVM 工具包的操作。
- 还可以在[单元测试](#) 中找到更多示例。

loc (控制) 的反转

项目 • 2023/03/10

一种常见模式，可用于使用 MVVM 模式提高应用程序的代码库中的模块化性，即使用某种形式的反转控制。最常见的解决方案之一是使用依赖项注入，它包括创建大量注入后端类的服务，(ie. 作为参数传递给 viewmodel 构造函数) - 这允许使用这些服务的代码不依赖于这些服务的实现详细信息，并且还可以轻松地交换这些服务的具体实现。此模式还使平台特定的功能更易于后端代码使用，方法是通过服务进行抽象化，然后在需要时注入这些功能。

MVVM 工具包不提供内置 API 来方便使用此模式，因为已有专用库专门用于此模式，例如 `Microsoft.Extensions.DependencyInjection` 包，该包提供功能齐全的强大 DI API 集，并充当易于设置和使用 `IServiceProvider` 的功能。以下指南将参考此库，并提供一系列示例，说明如何使用 MVVM 模式将其集成到应用程序中。

平台 API : [loc](#)

配置和解析服务

第一步是声明实例 `IServiceProvider`，并在启动时初始化所有必要的服务。例如，在 UWP (上，但其他框架也可以使用类似的设置，)：

C#

```
public sealed partial class App : Application
{
    public App()
    {
        Services = ConfigureServices();

        this.InitializeComponent();
    }

    /// <summary>
    /// Gets the current <see cref="App"/> instance in use
    /// </summary>
    public new static App Current => (App)Application.Current;

    /// <summary>
    /// Gets the <see cref="IServiceProvider"/> instance to resolve
    application services.
    /// </summary>
    public IServiceProvider Services { get; }

    /// <summary>
    /// Configures the services for the application.
```

```

/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<IFilesService, FilesService>();
    services.AddSingleton<ISettingsService, SettingsService>();
    services.AddSingleton<IClipboardService, ClipboardService>();
    services.AddSingleton<IShareService, ShareService>();
    services.AddSingleton<IEmailService, EmailService>();

    return services.BuildServiceProvider();
}
}

```

在此，该 `Services` 属性在启动时初始化，所有应用程序服务和 viewmodel 都注册。还有一个新 `Current` 属性可用于从应用程序中的其他视图轻松访问 `Services` 该属性。例如：

```

C#

IFilesService filesService = App.Current.Services.GetService<IFilesService>
();

// Use the files service here...

```

此处的关键方面是，每个服务很可能都使用特定于平台的 API，但由于这些 API 都通过代码正在使用的接口抽象出来，因此每当我们只是解析实例并使用它来执行操作时，我们都不需要担心它们。

构造函数注入

可用的一项强大功能是“构造函数注入”，这意味着 DI 服务提供商能够在创建所请求类型的实例时自动解析已注册服务之间的间接依赖关系。请考虑以下服务：

```

C#

public class FileLogger : IFileLogger
{
    private readonly IFilesService FilesService;
    private readonly IConsoleService ConsoleService;

    public FileLogger(
        IFilesService fileService,
        IConsoleService consoleService)
    {
        FilesService = fileService;
        ConsoleService = consoleService;
    }
}

```

```

    }

    // Methods for the ILogger interface here...
}

```

在这里，我们有一个 `FileLogger` 实现 `ILogger` 接口的类型，需要 `IFileService` 和 `IConsoleService` 实例。构造函数注入意味着 DI 服务提供商会自动收集所有必要的服务，如下所示：

```

C#

/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<IFileService, FileService>();
    services.AddSingleton<IConsoleService, ConsoleService>();
    services.AddSingleton<ILogger, FileLogger>();

    return services.BuildServiceProvider();
}

// Retrieve a logger service with constructor injection
ILogger logger = App.Current.Services.GetService<ILogger>();

```

DI 服务提供商将自动检查是否注册了所有必要的服务，然后它将检索它们并调用已注册 `ILogger` 的具体类型的构造函数，以获取要返回的实例。

viewmodels 怎么样？

服务提供商的名称中包含“服务”，但它实际上可用于解析任何类的实例，包括 viewmodels！上述相同概念仍适用，包括构造函数注入。假设我们有一个 `ContactsViewModel` 类型，它通过它的构造函数使用一个 `IContactsService` 和一个 `IPhoneService` 实例。我们可以采用 `ConfigureServices` 如下所示的方法：

```

C#

/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

```

```
// Services
services.AddSingleton<IContactsService, ContactsService>();
services.AddSingleton<IPhoneService, PhoneService>();

// Viewmodels
services.AddTransient<ContactsViewModel>();

return services.BuildServiceProvider();
}
```

然后在我们的 `ContactsView` 中，我们将按如下所示分配数据上下文：

```
C#

public ContactsView()
{
    this.InitializeComponent();
    this.DataContext = App.Current.Services.GetService<ContactsViewModel>();
}
```

更多文档

有关详细信息 `Microsoft.Extensions.DependencyInjection`，请参阅 [此处](#)。

示例

- 查看多个 UI 框架 [的示例应用](#) () 查看 MVVM 工具包的操作。
- 还可以在 [单元测试](#) 中找到更多示例。

Messenger

项目 • 2023/03/10

接口 `IMessenger` 是可用于在不同对象之间交换消息的类型协定。这可用于将应用程序的不同模块分离，而无需保留对所引用类型的强引用。还可以将消息发送到特定通道，由令牌唯一标识，并在应用程序的不同部分中具有不同的信使。MVVM Toolkit提供两种现装实现：`WeakReferenceMessenger`和`StrongReferenceMessenger`者在内部使用弱引用，为收件人提供自动内存管理，而后者使用强引用，并且要求开发人员在不再需要接收者时手动取消订阅收件人，（有关如何取消注册消息处理程序的更多详细信息，请参阅以下），但换言之，这样可以提高性能和少得多的内存使用。

平台 API：`IMessenger`、`WeakReferenceMessenger`、`IRecipient<TMessage>`、`StrongReferenceMessenger`、`ObservableRecipient`、`MessageHandler<TRecipient, TMessage>`、`RequestMessage<T>`、`AsyncRequestMessage<T>`、`CollectionRequestMessage<T>`、`AsyncCollectionRequestMessage<T>`

工作原理

实现 `IMessenger` 的类型负责维护接收方与相对消息处理程序（接收方之间的链接）。任何对象都可以使用消息处理程序将给定邮件类型注册为收件人，每当 `IMessenger` 该实例用于发送该类型的消息时，都会调用该对象。还可以通过特定通信通道发送消息，（唯一令牌）标识的每个通道，以便多个模块可以交换同一类型的消息，而不会造成冲突。在没有令牌的情况下发送的消息使用默认共享通道。

可通过两种方式执行消息注册：通过 `IRecipient<TMessage>` 接口或使用充当消息处理程序的 `MessageHandler<TRecipient, TMessage>` 委托。第一个允许使用对扩展的单个调用注册所有处理程序，该扩展 `RegisterAll` 会自动注册所有声明的消息处理程序的收件人，而后者在需要更多灵活性或想要将简单的 lambda 表达式用作消息处理程序时非常有用。

`StrongReferenceMessenger` 同时 `WeakReferenceMessenger` 公开一个 `Default` 属性，该属性提供内置于包中的线程安全实现。如果需要，还可以创建多个信使实例，例如，如果将另一个信使实例注入到应用的不同模块中，（应用的不同模块，则在同一进程中运行的多个窗口）。

❗ 备注

由于类型 `WeakReferenceMessenger` 更易于使用，并且与库中信使类型 `MvvmLight` 的行为匹配，因此它是 MVVM Toolkit中类型使用 `ObservableRecipient` 的默认类型。通过将实例传递给该类的构造函数，仍可使用该 `StrongReferenceType` 实例。

发送和接收消息

考虑以下情况：

C#

```
// Create a message
public class LoggedInUserChangedMessage : ValueChangedMessage<User>
{
    public LoggedInUserChangedMessage(User user) : base(user)
    {
    }
}

// Register a message in some module
WeakReferenceMessenger.Default.Register<LoggedInUserChangedMessage>(this,
(r, m) =>
{
    // Handle the message here, with r being the recipient and m being the
    // input message. Using the recipient passed as input makes it so that
    // the lambda expression doesn't capture "this", improving performance.
}));

// Send a message from some other module
WeakReferenceMessenger.Default.Send(new LoggedInUserChangedMessage(user));
```

假设此消息类型在简单的消息应用程序中使用，该应用程序显示一个标头，其中包含当前记录的用户的用户名和配置文件图像、一个包含对话列表的面板，以及另一个包含当前对话消息的面板（如果已选中）。假设这三个部分分别受这些

`HeaderViewModel` `ConversationsListViewModel` 部分和 `ConversationViewModel` 类型支持。

在此方案中，登录 `LoggedInUserChangedMessage` 操作完成后可能会发送 `HeaderViewModel` 消息，并且这两个其他 `viewmodel` 都可以为其注册处理程序。例如，

`ConversationsListViewModel` 将加载新用户的会话列表，如果存在会话，将 `ConversationViewModel` 仅关闭当前对话。

该 `IMessenger` 实例负责向所有已注册的收件人传送邮件。请注意，收件人可以订阅特定类型的消息。请注意，继承的消息类型未在 MVVM Toolkit 提供的默认 `IMessenger` 实现中注册。

不再需要收件人时，应取消注册，以便停止接收邮件。可以通过邮件类型、注册令牌或收件人取消注册：

C#


```
// Unregisters the recipient from a message type
WeakReferenceMessenger.Default.Unregister<LoggedInUserChangedMessage>(this);

// Unregisters the recipient from a message type in a specified channel
WeakReferenceMessenger.Default.Unregister<LoggedInUserChangedMessage, int>
(this, 42);

// Unregister the recipient from all messages, across all channels
WeakReferenceMessenger.Default.UnregisterAll(this);
```

⚠ 警告

如前所述，在使用类型时 `WeakReferenceMessenger`，这不是绝对必要的，因为它使用弱引用来跟踪收件人，这意味着未使用的收件人仍将有资格进行垃圾回收，即使它们仍然具有活动邮件处理程序。不过，取消订阅它们还是不错的做法，以提高性能。另一方面，`StrongReferenceMessenger` 实现使用强引用来跟踪已注册的收件人。这是出于性能原因完成的，这意味着应手动取消注册每个已注册收件人以避免内存泄漏。也就是说，只要注册收件人，正在使用的 `StrongReferenceMessenger` 实例就会保留对其的活动引用，这将阻止垃圾回收器能够收集该实例。可以手动处理此内容，也可以从中 `ObservableRecipient` 继承，默认情况下，在停用收件人的所有邮件注册时会自动将其删除，(查看 `ObservableRecipient` 有关此) 的详细信息文档。

也可以使用 `IRecipient<TMessage>` 接口注册消息处理程序。在这种情况下，每个收件人都需要为给定邮件类型实现接口，并提供 `Receive(TMessage)` 在接收消息时将调用的方法，如下所示：

C#

```
// Create a message
public class MyRecipient : IRecipient<LoggedInUserChangedMessage>
{
    public void Receive(LoggedInUserChangedMessage message)
    {
        // Handle the message here...
    }
}

// Register that specific message...
WeakReferenceMessenger.Default.Register<LoggedInUserChangedMessage>(this);

// ...or alternatively, register all declared handlers
WeakReferenceMessenger.Default.RegisterAll(this);

// Send a message from some other module
WeakReferenceMessenger.Default.Send(new LoggedInUserChangedMessage(user));
```

使用请求消息

信使实例的另一个有用功能是，它们还可用于将值从模块请求到另一个模块。为此，包含一个基 `RequestMessage<T>` 类，可以使用该基类，如下所示：

C#

```
// Create a message
public class LoggedInUserRequestMessage : RequestMessage<User>
{
}

// Register the receiver in a module
WeakReferenceMessenger.Default.Register<MyViewModel,
LoggedInUserRequestMessage>(this, (r, m) =>
{
    // Assume that "CurrentUser" is a private member in our viewmodel.
    // As before, we're accessing it through the recipient passed as
    // input to the handler, to avoid capturing "this" in the delegate.
    m.Reply(r.CurrentUser);
});

// Request the value from another module
User user = WeakReferenceMessenger.Default.Send<LoggedInUserRequestMessage>
();
```

该 `RequestMessage<T>` 类包含一个隐式转换器，使从 `LoggedInUserRequestMessage` 其包含 `User` 的对象进行转换成为可能。这还会检查是否已收到消息的响应，如果不是这种情况，则引发异常。还可以发送请求消息而不提供此强制响应保证：只需将返回的消息存储在本地变量中，然后手动检查响应值是否可用。如果方法返回时 `Send` 未收到响应，则这样做不会触发自动异常。

同一命名空间还包括其他方案的基请求消息：

`AsyncRequestMessage<T>` `CollectionRequestMessage<T>` 和

`AsyncCollectionRequestMessage<T>`。下面介绍如何使用异步请求消息：

C#

```
// Create a message
public class LoggedInUserRequestMessage : AsyncRequestMessage<User>
{
}

// Register the receiver in a module
WeakReferenceMessenger.Default.Register<MyViewModel,
LoggedInUserRequestMessage>(this, (r, m) =>
```

```
{  
    m.Reply(r.GetCurrentUserAsync()); // We're replying with a Task<User>  
});  
  
// Request the value from another module (we can directly await on the  
// request)  
User user = await  
WeakReferenceMessenger.Default.Send<LoggedInUserRequestMessage>();
```

示例

- 查看多个 UI 框架的[示例应用](#) () 查看 MVVM Toolkit。
- 还可以在 [单元测试](#) 中找到更多示例。

将内容放在一起

项目 • 2023/03/10

现在，我们已经概述了通过 `CommunityToolkit.Mvvm` 包提供的所有不同组件，接下来我们来看看它们共同构建单个更大的示例的实际示例。在本例中，我们希望为所选数量的子 reddit 生成一个非常简单且极简的 Reddit 浏览器。

我们想要生成什么

首先，我们确切地概述要生成的内容：

- 一个最少的 Reddit 浏览器由两个“小组件”组成：一个显示来自 subreddit 的文章，另一个显示当前所选帖子。这两个小组件需要是自包含的，并且彼此没有强引用。
- 我们希望用户能够从可用选项列表选择一个 subreddit，并且我们希望将所选的 subreddit 保存为设置，并在下次加载示例时加载它。
- 我们希望 subreddit 小组件还提供刷新按钮来重新加载当前 subreddit。
- 就此示例而言，我们不需要能够处理所有可能的帖子类型。我们将只为所有加载的文章分配一个示例文本，并直接显示该文本，使事情变得更简单。

设置 viewmodels

让我们从为 subreddit 小组件提供支持的 viewmodel 开始，让我们回顾一下我们需要的工具：

- **命令**：我们需要视图才能请求 viewmodel 从所选子reddit重新加载当前帖子列表。我们可以使用 `AsyncRelayCommand` 类型包装将从 Reddit 中提取帖子的私有方法。此处，我们通过接口公开命令 `IAsyncRelayCommand`，以避免对所使用的确切命令类型的强引用。这也使我们能够在将来更改命令类型，而无需担心任何 UI 组件依赖于正在使用的特定类型。
- **属性**：我们需要向 UI 公开一些值，如果可观测属性是我们打算完全替换的值，可以使用这些值，或者使用自身可观测 (的属性 (例如))。 `ObservableCollection<T>`)。在本例中，我们有：
 - `ObservableCollection<object> Posts`，这是已加载帖子的可观察列表。在这里，我们只是使用 `object` 作为占位符，因为我们尚未创建一个模型来表示帖子。我们稍后可以替换它。
 - `IReadOnlyList<string> Subreddits`，它是一个只读列表，其中包含我们允许用户从中选择的子reddit的名称。此属性永远不会更新，因此也不需要观察它。
 - `string SelectedSubreddit`，这是当前选定的子 reddit。此属性需要绑定到 UI，因为它既用于指示加载示例时最后一个选定的子，又在用户更改选择时直接从 UI

进行操作。此处，我们将使用 SetProperty 类中的 ObservableObject 方法。

- object SelectedPost，这是当前所选的帖子。在本例中 SetProperty，我们使用类中的 ObservableRecipient 方法来指示我们还希望在此属性更改时广播通知。这样做是为了能够通知帖子小组件当前帖子选择已更改。
- **方法：**我们只需要一个私有 LoadPostsAsync 方法，该方法将由异步命令包装，其中包含从所选子reddit加载帖子的逻辑。

下面是到目前为止的 viewmodel：

C#

```
public sealed class SubredditWidgetViewModel : ObservableRecipient
{
    /// <summary>
    /// Creates a new <see cref="SubredditWidgetViewModel"/> instance.
    /// </summary>
    public SubredditWidgetViewModel()
    {
        LoadPostsCommand = new AsyncRelayCommand(LoadPostsAsync);
    }

    /// <summary>
    /// Gets the <see cref="IAsyncRelayCommand"/> instance responsible for
    loading posts.
    /// </summary>
    public IAsyncRelayCommand LoadPostsCommand { get; }

    /// <summary>
    /// Gets the collection of loaded posts.
    /// </summary>
    public ObservableCollection<object> Posts { get; } = new
    ObservableCollection<object>();

    /// <summary>
    /// Gets the collection of available subreddits to pick from.
    /// </summary>
    public IReadOnlyList<string> Subreddits { get; } = new[]
    {
        "microsoft",
        "windows",
        "surface",
        "windowsphone",
        "dotnet",
        "csharp"
    };

    private string selectedSubreddit;

    /// <summary>
    /// Gets or sets the currently selected subreddit.
    /// </summary>
    public string SelectedSubreddit
```

```

{
    get => selectedSubreddit;
    set => SetProperty(ref selectedSubreddit, value);
}

private object selectedPost;

/// <summary>
/// Gets or sets the currently selected subreddit.
/// </summary>
public object SelectedPost
{
    get => selectedPost;
    set => SetProperty(ref selectedPost, value, true);
}

/// <summary>
/// Loads the posts from a specified subreddit.
/// </summary>
private async Task LoadPostsAsync()
{
    // TODO...
}
}

```

现在，让我们看一下发布小组件的 viewmodel 所需的内容。这将是一个更简单的 viewmodel，因为它实际上只需要公开具有 `Post` 当前所选帖子的属性，并从 subreddit 小组件接收广播消息即可更新属性 `Post`。它可以如下所示：

```

C#

public sealed class PostWidgetViewModel : ObservableRecipient,
    IRecipient<PropertyChangedMessage<object>>
{
    private object post;

    /// <summary>
    /// Gets the currently selected post, if any.
    /// </summary>
    public object Post
    {
        get => post;
        private set => SetProperty(ref post, value);
    }

    /// <inheritdoc/>
    public void Receive(PropertyChangedMessage<object> message)
    {
        if (message.Sender.GetType() == typeof(SubredditWidgetViewModel) &&
            message.PropertyName ==
            nameof(SubredditWidgetViewModel.SelectedPost))
        {

```

```

        Post = message.NewValue;
    }
}
}

```

在本例中，我们将使用 `IRecipient<TMessage>` 接口来声明希望 viewmodel 接收的消息。当属性设置为 `true` 时 `IsActive`，`ObservableRecipient` 类将自动添加声明的消息的处理程序。请注意，不强制使用此方法，也可以手动注册每个消息处理程序，如下所示：

C#

```

public sealed class PostWidgetViewModel : ObservableRecipient
{
    protected override void OnActivated()
    {
        // We use a method group here, but a lambda expression is also valid
        Messenger.Register<PostWidgetViewModel,
        PropertyChangedMessage<object>>(this, (r, m) => r.Receive(m));
    }

    /// <inheritdoc/>
    public void Receive(PropertyChangedMessage<object> message)
    {
        if (message.Sender.GetType() == typeof(SubredditWidgetViewModel) &&
            message.PropertyName ==
            nameof(SubredditWidgetViewModel.SelectedPost))
        {
            Post = message.NewValue;
        }
    }
}

```

现在，我们已准备好 viewmodels 的草稿，我们可以开始研究所需的服务。

生成设置服务

① 备注

此示例是使用依赖项注入模式生成的，这是处理 viewmodels 中服务的推荐方法。也可以使用其他模式，例如服务定位器模式，但 MVVM 工具包不提供内置 API 来启用此模式。

由于我们希望保存并持久保存某些属性，因此需要一种方法让 viewmodel 能够与应用程序设置交互。不过，我们不应直接在 viewmodel 中使用特定于平台的 API，因为这样会阻止我们在可移植的 .NET Standard 项目中拥有所有 viewmodel。可以通过使用服务和

库中的 API `Microsoft.Extensions.DependencyInjection` 为应用程序设置

`IServiceProvider` 实例来解决此问题。其思路是编写表示我们需要的所有 API 图面的接口，然后在所有应用程序目标上实现此接口的特定于平台的类型。viewmodel 将仅与接口交互，因此它们根本不具有对任何特定于平台的类型的任何强引用。

下面是设置服务的简单界面：

C#

```
public interface ISettingsService
{
    /// <summary>
    /// Assigns a value to a settings key.
    /// </summary>
    /// <typeparam name="T">The type of the object bound to the key.
</typeparam>
    /// <param name="key">The key to check.</param>
    /// <param name="value">The value to assign to the setting key.</param>
    void SetValue<T>(string key, T value);

    /// <summary>
    /// Reads a value from the current <see cref="IServiceProvider"/>
    instance and returns its casting in the right type.
    /// </summary>
    /// <typeparam name="T">The type of the object to retrieve.</typeparam>
    /// <param name="key">The key associated to the requested object.
</param>
    [Pure]
    T GetValue<T>(string key);
}
```

我们可以假设实现此接口的平台特定类型将负责处理实际序列化设置、将其存储到磁盘，然后读回设置所需的所有逻辑。现在，我们可以在 `SubredditWidgetViewModel` 中使用此服务，以便使 `SelectedSubreddit` 属性持久化：

C#

```
/// <summary>
/// Gets the <see cref="ISettingsService"/> instance to use.
/// </summary>
private readonly ISettingsService SettingsService;

/// <summary>
/// Creates a new <see cref="SubredditWidgetViewModel"/> instance.
/// </summary>
public SubredditWidgetViewModel(ISettingsService settingsService)
{
    SettingsService = settingsService;

    selectedSubreddit = settingsService.GetValue<string>
```



```

(nameof(SelectedSubreddit)) ?? Subreddits[0];
}

private string selectedSubreddit;

/// <summary>
/// Gets or sets the currently selected subreddit.
/// </summary>
public string SelectedSubreddit
{
    get => selectedSubreddit;
    set
    {
        SetProperty(ref selectedSubreddit, value);

        SettingsService.SetValue(nameof(SelectedSubreddit), value);
    }
}

```

此处，我们将使用依赖项注入和构造函数注入，如上所述。我们声明了一个 `ISettingsService SettingsService` 字段，该字段仅存储设置服务（我们将作为参数在 `viewmodel` 构造函数）接收，然后通过使用上一个值或仅使用第一个可用的 `subreddit` 在构造函数中初始化 `SelectedSubreddit` 属性。然后，我们还修改了 `SelectedSubreddit` 资源库，以便它还使用设置服务将新值保存到磁盘。

很好！现在，我们只需编写此服务的平台特定版本，这次直接在某个应用项目中编写。下面是该服务在 UWP 上的外观：

```

C#

public sealed class SettingsService : ISettingsService
{
    /// <summary>
    /// The <see cref="IPropertySet"/> with the settings targeted by the
    /// current instance.
    /// </summary>
    private readonly IPropertySet SettingsStorage =
        ApplicationData.Current.LocalSettings.Values;

    /// <inheritdoc/>
    public void SetValue<T>(string key, T value)
    {
        if (!SettingsStorage.ContainsKey(key)) SettingsStorage.Add(key,
value);
        else SettingsStorage[key] = value;
    }

    /// <inheritdoc/>
    public T GetValue<T>(string key)
    {
        if (SettingsStorage.TryGetValue(key, out object value))

```

```

    {
        return (T)value;
    }

    return default;
}
}

```

最后一个难题是将此特定于平台的服务注入到服务提供商实例中。我们可以在启动时执行此操作，如下所示：

```

C#

/// <summary>
/// Gets the <see cref="IServiceProvider"/> instance to resolve application
/// services.
/// </summary>
public IServiceProvider Services { get; }

/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<ISettingsService, SettingsService>();
    services.AddTransient<PostWidgetViewModel>();

    return services.BuildServiceProvider();
}

```

这会将的单 `SettingsService` 一实例实例注册为实现的类型 `ISettingsService`。我们还将 `PostWidgetViewModel` 注册为暂时性服务，这意味着每次检索实例时，它都是一个新的实例，(如果想要) 多个独立的发布小组件，这非常有用。这意味着，每次解析实例 `ISettingsService` 时，正在使用的应用是 UWP 实例时，它都会收到一个 `SettingsService` 实例，该实例将在后台使用 UWP API 来操作设置。太完美了！

生成 Reddit 服务

我们缺少的后端的最后一个组件是一个能够使用 Reddit REST API 从我们感兴趣的子 reddit 中提取帖子的服务。若要生成它，我们将使用 [refit](#)，这是一个库，用于轻松生成类型安全服务以与 REST API 交互。与以前一样，我们需要使用服务将实现的所有 API 定义接口，如下所示：

```

C#

```

```

public interface IRedditService
{
    /// <summary>
    /// Get a list of posts from a given subreddit
    /// </summary>
    /// <param name="subreddit">The subreddit name.</param>
    [Get("/r/{subreddit}/.json")]
    Task<PostsQueryResponse> GetSubredditPostsAsync(string subreddit);
}

```

这是 `PostsQueryResponse` 我们编写的模型，用于映射该 API 的 JSON 响应。该类的确切结构并不重要 - 足以说明它包含项的集合，这些项是表示帖子的 `Post` 简单模型，如下所示：

```

C#

public class Post
{
    /// <summary>
    /// Gets or sets the title of the post.
    /// </summary>
    public string Title { get; set; }

    /// <summary>
    /// Gets or sets the URL to the post thumbnail, if present.
    /// </summary>
    public string Thumbnail { get; set; }

    /// <summary>
    /// Gets the text of the post.
    /// </summary>
    public string SelfText { get; }
}

```

获得服务和模型后，即可将它们插入 viewmodel 以完成后端。执行此操作时，我们还可以将这些 `object` 占位符 `Post` 替换为我们定义的类型：

```

C#

public sealed class SubredditWidgetViewModel : ObservableRecipient
{
    /// <summary>
    /// Gets the <see cref="IRedditService"/> instance to use.
    /// </summary>
    private readonly IRedditService RedditService =
        Ioc.Default.GetRequiredService<IRedditService>();

    /// <summary>
    /// Loads the posts from a specified subreddit.

```

```

/// </summary>
private async Task LoadPostsAsync()
{
    var response = await
RedditService.GetSubredditPostsAsync(SelectedSubreddit);

    Posts.Clear();

    foreach (var item in response.Data.Items)
    {
        Posts.Add(item.Data);
    }
}
}

```

我们添加了一个新 `IRedditService` 字段来存储服务，就像对设置服务所做的一样，我们实现了 `LoadPostsAsync` 方法，该方法以前为空。

现在缺少的最后一部分只是将实际服务注入服务提供商。在这种情况下，最大的区别在于，通过使用 `refit`，我们实际上根本不需要实现服务！库将在后台自动创建一个类型，为我们实现服务。因此，我们只需要获取实例 `IRedditService` 并直接注入它，如下所示：

```

C#

/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<ISettingsService, SettingsService>();
    services.AddSingleton(RestService.For<IRedditService>
("https://www.reddit.com/"));
    services.AddTransient<PostWidgetViewModel>();

    return services.BuildServiceProvider();
}

```

这就是我们需要做的！现在，我们已准备好使用所有后端，包括我们专门为此应用创建的两个自定义服务！🎉

生成 UI

完成所有后端后，我们可以为小组件编写 UI。请注意，使用 MVVM 模式如何让我们首先专注于业务逻辑，而无需编写任何 UI 相关代码，直到现在为止。为了简单起见，我们

将删除所有不与 viewmodel 交互的 UI 代码，并逐一浏览每个不同的控件。可以在示例应用中找到完整的源代码。

在浏览各种控件之前，下面介绍如何解析应用程序中所有不同视图的 viewmodel (例如。PostWidgetView)：

C#

```
public PostWidgetView()
{
    this.InitializeComponent();
    this.DataContext = App.Current.Services.GetService<PostWidgetViewModel>
();
}

public PostWidgetViewModel ViewModel => (PostWidgetViewModel)DataContext;
```

我们将使用 `IServiceProvider` 实例来解析 `PostWidgetViewModel` 所需的对象，然后该对象将分配给数据上下文属性。我们还将创建一个强类型 `ViewModel` 属性，该属性只是将数据上下文转换为正确的 viewmodel 类型 - 这是在 XAML 代码中启用 `x:Bind` 所必需的。

让我们从 subreddit 小组件开始，该小组件具有 `ComboBox` 用于选择 subreddit、`Button` 用于刷新源、`ListView` 用于显示帖子和 `ProgressBar` 用于指示源加载时间的。我们假设 `ViewModel` 属性表示我们之前描述的 viewmodel 的实例 - 这可以在 XAML 中声明，也可以直接在代码隐藏中声明。

Subreddit 选择器：

XML

```
<ComboBox
    ItemsSource="{x:Bind ViewModel.Subreddits}"
    SelectedItem="{x:Bind ViewModel.SelectedSubreddit, Mode=TwoWay}">
    <interactivity:Interaction.Behaviors>
        <core:EventTriggerBehavior EventName="SelectionChanged">
            <core:InvokeCommandAction Command="{x:Bind
ViewModel.LoadPostsCommand}"/>
        </core:EventTriggerBehavior>
    </interactivity:Interaction.Behaviors>
</ComboBox>
```

此处，我们将源绑定到 `Subreddits` 属性，并将所选项绑定到 `SelectedSubreddit` 属性。`Subreddits` 请注意，属性只绑定一次，因为集合本身会发送更改通知，而 `SelectedSubreddit` 属性与 `TwoWay` 模式绑定，因为我们既需要它才能加载从设置检索的

值，又能在用户更改所选内容时更新 viewmodel 中的属性。此外，每当选择发生更改时，我们都会使用 XAML 行为来调用命令。

“刷新”按钮：

XML

```
<Button Command="{x:Bind ViewModel.LoadPostsCommand}"/>
```

此组件非常简单，只需将自定义命令绑定到 `Command` 按钮的属性，以便在用户单击该命令时调用该命令。

帖子列表：

XML

```
<ListView
    ItemsSource="{x:Bind ViewModel.Posts}"
    SelectedItem="{x:Bind ViewModel.SelectedPost, Mode=TwoWay}">
    <ListView.ItemTemplate>
        <DataTemplate x:DataType="models:Post">
            <Grid>
                <TextBlock Text="{x:Bind Title}"/>
                <controls:ImageEx Source="{x:Bind Thumbnail}"/>
            </Grid>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

此处，我们将 `ListView` 源和所选内容绑定到 viewmodel 属性，以及用于显示每个可用文章的模板。我们使用 `x:DataType` 在模板中启用 `x:Bind`，并且有两个控件直接 `Title` 绑定到帖子的 `Title` 和 `Thumbnail` 属性。

加载栏：

XML

```
<ProgressBar Visibility="{x:Bind ViewModel.LoadPostsCommand.IsRunning,
    Mode=OneWay}"/>
```

此处，我们将绑定到 `IsRunning` 属性，该属性是接口的 `IAsyncRelayCommand` 一部分。每当该命令的异步操作开始或完成时，类型 `AsyncRelayCommand` 将负责引发该属性的通知。

最后一个缺失部分是帖子小组件的 UI。与以前一样，为了简单起见，我们删除了与 viewmodel 交互时不需要的所有与 UI 相关的代码。示例应用中提供了完整的源代码。

XML

```
<Grid>

    <!--Header-->
    <Grid>
        <TextBlock Text="{x:Bind ViewModel.Post.Title, Mode=OneWay}"/>
        <controls:ImageEx Source="{x:Bind ViewModel.Post.Thumbnail,
Mode=OneWay}"/>
    </Grid>

    <!--Content-->
    <ScrollView>
        <TextBlock Text="{x:Bind ViewModel.Post.SelfText, Mode=OneWay}"/>
    </ScrollView>
</Grid>
```

在这里，我们只有一个标头，其中的 `TextBlock` 和 控件将其 `Text` 和 `Source` 属性绑定到模型中 `Post` 的相应属性，以及一个用于显示所选文章 (示例) 内容的简单

`TextBlock` `ScrollView` 。 `ImageEx`

示例应用程序

[此处](#) 提供了示例应用程序。

好走！

现在，我们已生成所有 viewmodel、必要的服务和小组件的 UI - 简单的 Reddit 浏览器已完成！这只是一个示例，说明如何按照 MVVM 模式并使用 MVVM 工具包中的 API 生成应用。

如上所述，这只是一个引用，你可以随意修改此结构以满足你的需求和/或仅从库中选取和选择一部分组件。无论你采用哪种方法，MVVM 工具包都应为启动新应用程序时提供坚实的基础，让你专注于业务逻辑，而无需担心手动执行所有必要的管道来为 MVVM 模式提供适当的支持。

从 MvvmLight 迁移

项目 • 2023/03/10

本文概述了 [MvvmLight 工具包](#) 与 [MVVM 工具包](#) 之间的一些主要差异，以便简化迁移。

虽然本文专门重点介绍从 MvvmLight 到 MVVM 工具包的迁移，但请注意，MVVM 工具包中进行了其他改进，因此强烈建议查看各个新 API 的文档。

平台 API： [ObservableObject](#)、[ObservableRecipientRelayCommand](#)、[RelayCommand<T>](#)、[AsyncRelayCommand](#)、[AsyncRelayCommand<T>IMessenger](#)、[WeakReferenceMessenger](#)、[StrongReferenceMessenger](#)、[IRecipient<TMessage>](#)、[MessageHandler<TRecipient, TMessage>](#)、[IMessengerExtensions](#)

安装 MVVM 工具包

若要利用 MVVM 工具包，首先需要将最新的 NuGet 包安装到现有的 .NET 应用程序。

通过 .NET CLI 安装

```
dotnet add package CommunityToolkit.Mvvm --version 8.1.0
```

通过 PackageReference 安装

XML

```
<PackageReference Include="CommunityToolkit.Mvvm" Version="8.1.0" />
```

迁移 ObservableObject

以下步骤侧重于迁移利用 `ObservableObject` MvvmLight 工具包的现有组件。MVVM 工具包提供 [ObservableObject](#) 类似的类型。

此处的第一个更改是交换组件中的 using 指令。

C#


```
// MvvmLight
using GalaSoft.MvvmLight;

// MVVM Toolkit
using CommunityToolkit.Mvvm.ComponentModel;
```

下面是在当前解决方案中使用时需要执行的迁移列表。

ObservableObject 方法

Set<T>(Expression, ref T, T)

Set(Expression, ref T, T) 没有 like-for-like 方法签名替换。

但是，SetProperty(ref T, T, string) 提供相同的功能，并具有额外的性能优势。

C#

```
// MvvmLight
Set(() => MyProperty, ref this.myProperty, value);

// MVVM Toolkit
SetProperty(ref this.myProperty, value);
```

请注意，string 如果从属性的 setter 调用方法，则不需要参数，因为它是从调用方成员名称推断出来的，如下所示。如果要为与调用方法的属性不同的属性调用 SetProperty，可以使用运算符来执行此操作 nameof，这对于使没有硬编码名称的代码更容易出错非常有用。例如：

C#

```
SetProperty(ref this.someProperty, value, nameof(SomeProperty));
```

Set<T>(string, ref T, T)

Set<T>(string, ref T, T) 没有 like-for-like 方法签名替换。

但是，SetProperty<T>(ref T, T, string) 通过重新排序的参数提供相同的功能。

C#

```
// MvvmLight
Set(nameof(MyProperty), ref this.myProperty, value);
```

```
// MVVM Toolkit
SetProperty(ref this.myProperty, value);
```

Set<T>(ref T, T, string)

Set<T>(ref T, T, string) 具有重命名的直接替换。 SetProperty<T>(ref T, T, string)

C#

```
// MvvmLight
Set(ref this.myProperty, value, nameof(MyProperty));

// MVVM Toolkit
SetProperty(ref this.myProperty, value);
```

RaisePropertyChanged(string)

RaisePropertyChanged(string) 具有重命名的直接替换。 OnPropertyChanged(string)

C#

```
// MvvmLight
RaisePropertyChanged(nameof(MyProperty));

// MVVM Toolkit
OnPropertyChanged();
```

与一 SetProperty 样，方法会自动推断 OnPropertyChanged 当前属性的名称。如果要使用此方法为另一个属性手动引发 PropertyChanged 事件，还可以再次使用 nameof 运算符手动指定该属性的名称。例如：

C#

```
OnPropertyChanged(nameof(SomeProperty));
```

RaisePropertyChanged<T>(Expression)

RaisePropertyChanged<T>(Expression) 没有直接替换项。

为了提高性能，建议将 替换为 RaisePropertyChanged<T>(Expression) 使用 nameof 关键字的 OnPropertyChanged(string)，而不是 (或不使用参数 (如果目标属性与调用方法的属性相同)，以便可以如上文所述自动推断名称)。

C#

```
// MvvmLight
RaisePropertyChanged(() => MyProperty);

// MVVM Toolkit
OnPropertyChanged(nameof(MyProperty));
```

VerifyPropertyName(string)

无法直接替换 `VerifyPropertyName(string)` 方法，应更改或删除使用此方法的任何代码。

MVVM 工具包中省略的原因是，对属性使用 `nameof` 关键字可验证它是否存在。生成 `MvvmLight` 时，`nameof` 关键字不可用，并且此方法用于确保属性存在于对象上。

C#

```
// MvvmLight
VerifyPropertyName(nameof(MyProperty));

// MVVM Toolkit
// No direct replacement, remove
```

ObservableObject 属性

PropertyChangedHandler

`PropertyChangedHandler` 没有直接替换项。

若要通过 `PropertyChanged` 事件处理程序引发属性更改事件，需要改为调用 `OnPropertyChanged` 方法。

C#

```
// MvvmLight
PropertyChangedEventHandler handler = PropertyChangedHandler;

// MVVM Toolkit
OnPropertyChanged();
```

迁移 ViewModelBase

以下步骤侧重于迁移利用 `ViewModelBase` `MvvmLight` 工具包的现有组件。

MVVM 工具包提供了一种 `ObservableRecipient` 提供类似功能的类型。

下面是在当前解决方案中使用时需要执行的迁移列表。

ViewModelBase 方法

`Set<T>(string, ref T, T, bool)`

`Set<T>(string, ref T, T, bool)` 没有 like-for-like 方法签名替换。

但是，`SetProperty<T>(ref T, T, bool, string)` 通过重新排序的参数提供相同的功能。

C#

```
// MvvmLight
Set(nameof(MyProperty), ref this.myProperty, value, true);

// MVVM Toolkit
SetProperty(ref this.myProperty, value, true);
```

请注意，值和广播布尔参数在 MVVM 工具包的实现中不是可选的，必须提供才能使用此方法。此更改的原因是，通过在调用此方法时省略广播参数，它将默认调用 `ObservableObject` 的 `SetProperty` 方法。

此外，`string` 如果从属性的 setter 调用方法，则不需要参数，因为它是从调用方成员名称推断出来的，就像基类中 `ObservableObject` 的方法一样。

`Set<T>(ref T, T, bool, string)`

`Set<T>(ref T, T, bool, string)` 具有重命名的直接替换。`SetProperty<T>(ref T, T, bool, string)`

C#

```
// MvvmLight
Set(ref this.myProperty, value, true, nameof(MyProperty));

// MVVM Toolkit
SetProperty(ref this.myProperty, value, true);
```

`Set<T>(Expression, ref T, T, bool)`

`Set<T>(Expression, ref T, T, bool)` 没有直接替换项。

为了提高性能，建议改为使用 `nameof` 关键字将其替换为 MVVM 工具包 `SetProperty<T>(ref T, T, bool, string)`。

C#

```
// MvvmLight
Set<MyObject>(() => MyProperty, ref this.myProperty, value, true);

// MVVM Toolkit
SetProperty(ref this.myProperty, value, true);
```

Broadcast<T>(T, T, string)

`Broadcast<T>(T, T, string)` 具有不需要重命名的直接替换项。

C#

```
// MvvmLight
Broadcast<MyObject>(oldValue, newValue, nameof(MyProperty));

// MVVM Toolkit
Broadcast(oldValue, newValue, nameof(MyProperty));
```

请注意，调用 `Broadcast` 方法时通过 `Messenger` 属性发送的消息在 MVVM 工具包库中直接替换 `PropertyChangedMessage`。

RaisePropertyChanged<T>(string, T, T, bool)

无法直接替换 `RaisePropertyChanged<T>(string, T, T, bool)` 方法。

最简单的替代方法是调用 `OnPropertyChanged` 并随后调用 `Broadcast` 来实现此功能。

C#

```
// MvvmLight
RaisePropertyChanged<MyObject>(nameof(MyProperty), oldValue, newValue, true);

// MVVM Toolkit
OnPropertyChanged();
Broadcast(oldValue, newValue, nameof(MyProperty));
```

RaisePropertyChanged<T>(Expression, T, T, bool)

无法直接替换 `RaisePropertyChanged<T>(Expression, T, T, bool)` 方法。

最简单的替代方法是调用 `OnPropertyChanged` 并随后调用 `Broadcast` 来实现此功能。

C#

```
// MvvmLight
RaisePropertyChanged<MyObject>(() => MyProperty, oldValue, newValue, true);

// MVVM Toolkit
OnPropertyChanged(nameof(MyProperty));
Broadcast(oldValue, newValue, nameof(MyProperty));
```

ICleanup.Cleanup()

接口没有直接替代项 `ICleanup`。

但是，`ObservableRecipient` 提供了一个 `OnDeactivated` 方法，该方法应用于提供与相同的功能 `Cleanup`。

`OnDeactivated` 在调用 MVVM 工具包时，还会取消注册所有已注册的 messenger 事件。

C#

```
// MvvmLight
Cleanup();

// MVVM Toolkit
OnDeactivated();
```

请注意，`OnActivated` 可以从现有解决方案调用和 `OnDeactivated` 方法，就像使用 `Cleanup` 一样。

但是，`ObservableRecipient` 公开属性 `IsActive`，该属性在设置时还控制对这些方法的调用。

ViewModelBase 属性

MessengerInstance

`MessengerInstance` 具有重命名的直接替换。`Messenger`

C#

```
// MvvmLight
IMessenger messenger = MessengerInstance;

// MVVM Toolkit
IMessenger messenger = Messenger;
```

① 备注

属性的 `Messenger` 默认值为 `WeakReferenceMessenger.Default` 实例，它是 MVVM 工具包中的标准弱引用信使实现。只需将另 `IMessenger` 一个实例注入 `ObservableRecipient` 构造函数即可对此进行自定义。

IsInDesignMode

没有直接替换 属性，`IsInDesignMode` 应更改或删除使用此属性的任何代码。

MVVM 工具包中省略的原因是 属性 `IsInDesignMode` 公开了特定于平台的实现。MVVM 工具包设计为与平台无关。

C#

```
// MvvmLight
var isInDesignMode = IsInDesignMode;

// MVVM Toolkit
// No direct replacement, remove
```

ViewModelBase 静态属性

IsInDesignModeStatic

没有直接替换 属性，`IsInDesignModeStatic` 应更改或删除使用此属性的任何代码。

MVVM 工具包中省略的原因是 属性 `IsInDesignMode` 公开了特定于平台的实现。MVVM 工具包设计为与平台无关。

C#

```
// MvvmLight
var isInDesignMode = ViewModelBase.IsInDesignModeStatic;
```

```
// MVVM Toolkit
// No direct replacement, remove
```

迁移 RelayCommand

以下步骤侧重于迁移利用 RelayCommand MvvmLight 工具包的现有组件。

MVVM 工具包提供一种 RelayCommand 类型，该类型利用 ICommand 系统接口提供类似功能。

下面是在当前解决方案中使用时需要执行的迁移列表。如果方法或属性未列出，则 MVVM 工具包中直接替换相同名称，无需更改。

此处的第一个更改是交换组件中的 using 指令。

```
C#

// MvvmLight
using GalaSoft.MvvmLight.Command;
using Galasoft.MvvmLight.CommandWpf;

// MVVM Toolkit
using CommunityToolkit.Mvvm.Input;
```

❗ 备注

MvvmLight 使用弱引用在命令与从关联类调用的操作之间建立链接。这并非 MVVM Toolkit 实现所必需的，如果此可选参数已在任何构造函数中设置为 true，则会将其删除。

将 RelayCommand 与异步操作配合使用

如果当前将 MvvmLight RelayCommand 实现与异步操作一起使用，则 MVVM 工具包会针对这些方案公开改进的实现。

只需将现有 RelayCommand AsyncRelayCommand 替换为已为异步目的而生成的。

```
C#

// MvvmLight
var command = new RelayCommand(() => OnCommandAsync());
var command = new RelayCommand(async () => await OnCommandAsync());
```



```
// MVVM Toolkit
var asyncCommand = new AsyncRelayCommand(OnCommandAsync);
```

RelayCommand 方法

RaiseCanExecuteChanged()

可以使用 MVVM 工具包的 `NotifyCanExecuteChanged()` 方法实现的功能 `RaiseCanExecuteChanged()`。

```
C#

// MvvmLight
var command = new RelayCommand(OnCommand);
command.RaiseCanExecuteChanged();

// MVVM Toolkit
var command = new RelayCommand(OnCommand);
command.NotifyCanExecuteChanged();
```

迁移 RelayCommand<T>

以下步骤侧重于迁移利用 `RelayCommand<T>` MvvmLight 工具包的现有组件。

MVVM 工具包提供一种 `RelayCommand<T>` 类型，该类型利用 `ICommand` 系统接口提供类似功能。

下面是在当前解决方案中使用时需要执行的迁移列表。如果方法或属性未列出，则 MVVM 工具包中直接替换相同名称，无需更改。

此处的第一个更改是交换组件中的 using 指令。

```
C#

// MvvmLight
using GalaSoft.MvvmLight.Command;
using Galasoft.MvvmLight.CommandWpf;

// MVVM Toolkit
using CommunityToolkit.Mvvm.Input;
```

将 RelayCommand 与异步操作配合使用

如果当前将 `MvvmLight RelayCommand<T>` 实现与异步操作一起使用，则 MVVM 工具包会针对这些方案公开改进的实现。

只需将现有 `RelayCommand<T>` `AsyncRelayCommand<T>` 替换为已为异步目的而生成的。

C#

```
// MvvmLight
var command = new RelayCommand<string>(async () => await OnCommandAsync());

// MVVM Toolkit
var asyncCommand = new AsyncRelayCommand<string>(OnCommandAsync);
```

RelayCommand<T> 方法

RaiseCanExecuteChanged()

可以使用 MVVM 工具包的 `NotifyCanExecuteChanged()` 方法实现的功能

`RaiseCanExecuteChanged()`。

C#

```
// MvvmLight
var command = new RelayCommand<string>(OnCommand);
command.RaiseCanExecuteChanged();

// MVVM Toolkit
var command = new RelayCommand<string>(OnCommand);
command.NotifyCanExecuteChanged();
```

迁移 SimpleIoc

MVVM 工具包中的 `IoC` 实现不包含任何用于自行处理依赖项注入的内置逻辑，因此你可以随意使用任何第三方库来检索 `IServiceProvider` 实例，然后可以将其传递给

`Ioc.ConfigureServices` 方法。在下面的示例中 `ServiceCollection`，将使用库中的类型 `Microsoft.Extensions.DependencyInjection`。

这是 `MvvmLight` 和 `MVVM 工具包` 之间的最大更改。

如果已使用 `ASP.NET Core` 应用程序实现依赖项注入，则此实现会让人感到熟悉。

注册依赖项

使用 MvvmLight 时，你可能已使用 `SimpleIoc` 注册了类似于这些方案的依赖项。

C#

```
public void RegisterServices()
{
    SimpleIoc.Default.Register<INavigationService, NavigationService>();

    SimpleIoc.Default.Register<IDialogService>(() => new DialogService());
}
```

使用 MVVM 工具包，可以实现如下相同的操作。

C#

```
public void RegisterServices()
{
    Ioc.Default.ConfigureServices(
        new ServiceCollection()
            .AddSingleton<INavigationService, NavigationService>()
            .AddSingleton<IDialogService>(new DialogService())
            .BuildServiceProvider());
}
```

解析依赖项

初始化后，可以从类检索服务，`Ioc` 就像使用 `SimpleIoc` 一样：

C#

```
IDialogService dialogService = SimpleIoc.Default.GetInstance<IDialogService>
();
```

迁移到 MVVM 工具包后，你将通过以下方法实现相同的目标：

C#

```
IDialogService dialogService = Ioc.Default.GetService<IDialogService>();
```

删除依赖项

使用 `SimpleIoc`，可以使用以下方法调用取消注册依赖项。

C#

```
SimpleIoc.Default.Unregister<INavigationService>();
```

使用 MVVM 工具包 `Ioc` 实现无法直接替代删除依赖项。

首选构造函数

向 MvvmLight 的 `SimpleIoc` 注册依赖项时，可以在类中选择为具有多个构造函数的依赖项提供 `PreferredConstructor` 属性。

此属性将需要删除使用的位置，并且你将需要使用正在使用的第三方依赖项注入库中的任何属性（如果受支持）。

迁移 Messenger

以下步骤侧重于迁移利用 `Messenger` MvvmLight 工具包的现有组件。

MVVM 工具包提供两个 messenger 实现 (`WeakReferenceMessenger` 和 `StrongReferenceMessenger`，请参阅 [此处](#) 提供类似功能的文档，下面详细介绍了一些关键差异。

下面是在当前解决方案中使用时需要执行的迁移列表。

此处的第一个更改是交换组件中的 `using` 指令。

C#

```
// MvvmLight
using GalaSoft.MvvmLight.Messaging;

// MVVM Toolkit
using CommunityToolkit.Mvvm.Messaging;
```

Messenger 方法

`Register<TMessage>(object, Action<TMessage>)`

`Register<TMessage>(object, Action<TMessage>)` 的功能可以通过 MVVM 工具包的 `IMessenger` 扩展方法 `Register<TRecipient, TMessage>(object, MessageHandler<TRecipient, TMessage>)` 实现。

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage>(this, static (r, m) =>
    r.OnMyMessageReceived(m));
```

此签名的原因是它允许信使使用弱引用来正确跟踪收件人，并避免创建闭包来捕获收件人本身。也就是说，输入接收方作为输入传递给 lambda 表达式，因此不需要由 lambda 表达式本身捕获。这也会导致代码更高效，因为同一处理程序可以重复使用多次，无需分配。请注意，这只是注册处理程序的受支持方法之一，还可以改用

`IRecipient<TMessage>` 接口，([信使文档](#)) 中详述，这使得注册自动且不那么详细。

❗ 备注

`static` lambda 表达式的修饰符需要 C# 9，它是可选的。在此处使用它可以确保不会意外捕获收件人或某个其他成员，从而导致分配关闭，但这不是必需的。如果无法使用 C# 9，可以在此处删除 `static`，并小心确保代码未捕获任何内容。

此外，此示例和下面的示例将仅使用 `Messenger` 中的 `ObservableRecipient` 属性。如果只想从代码中的其他任何位置静态访问 `Messenger` 实例，则同样适用相同的示例，唯一的区别是 `Messenger` 需要替换为 例如。 `WeakReferenceMessenger.Default` 相反。

`Register<TMessage>(object, bool, Action<TMessage>)`

无法直接替换此注册机制，它允许你也支持接收派生消息类型的消息。此更改是有意，`Messenger` 因为实现旨在不使用反射来实现其性能优势。

或者，可以通过几个选项来实现此功能。

- 创建自定义 `IMessenger` 实现。
- 使用共享处理程序注册其他消息类型，然后检查类型并调用正确的方法。

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, true, this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage>(this, static (r, m) =>
    r.OnMyMessageReceived(m));
Messenger.Register<MyViewModel, MyOtherMessage>(this, static (r, m) =>
    r.OnMyMessageReceived(m));
```

Register<TMessage>(object, object, Action<TMessage>)

可以使用 MVVM 工具包的 `Register<TRecipient, TMessage, TToken>(object, TToken, MessageHandler<TRecipient, TMessage>)` 方法实现的功能 `Register<TMessage>(object, object, Action<TMessage>)`。

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, nameof(MyViewModel),
this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage, string>(this,
nameof(MyViewModel), static (r, m) => r.OnMyMessageReceived(m));
```

Register<TMessage>(object, object, bool, Action<TMessage>)

无法直接替换此注册机制，它允许你也支持接收派生消息类型的消息。此更改是有意，`Messenger` 因为实现旨在不使用反射来实现其性能优势。

或者，可以通过几个选项来实现此功能。

- 创建自定义 `IMessenger` 实现。
- 使用共享处理程序注册其他消息类型，然后检查类型并调用正确的方法。

C#

```
// MvvmLight
Messenger.Default.Register<MyMessage>(this, nameof(MyViewModel), true,
this.OnMyMessageReceived);

// MVVM Toolkit
Messenger.Register<MyViewModel, MyMessage, string>(this,
nameof(MyViewModel), static (r, m) => r.OnMyMessageReceived(m));
Messenger.Register<MyViewModel, MyOtherMessage, string>(this,
nameof(MyViewModel), static (r, m) => r.OnMyMessageReceived(m));
```

Send<TMessage>(TMessage)

`Send<TMessage>(TMessage)` 的功能可以通过 MVVM 工具包的 `IMessenger` 扩展方法 `Send<TMessage>(TMessage)` 实现。

C#

```
// MvvmLight
Messenger.Default.Send<MyMessage>(new MyMessage());
Messenger.Default.Send(new MyMessage());

// MVVM Toolkit
Messenger.Send(new MyMessage());
```

在上述情况下，所发送的消息具有无参数构造函数，MVVM 工具包具有简化的扩展，用于以此格式发送消息。

C#

```
// MVVM Toolkit
Messenger.Send<MyMessage>();
```

Send<TMessage>(TMessage, object)

可以使用 MVVM 工具包的 `Send<TMessage, TToken>(TMessage, TToken)` 方法实现的功能 `Send<TMessage>(TMessage, object)`。

C#

```
// MvvmLight
Messenger.Default.Send<MyMessage>(new MyMessage(), nameof(MyViewModel));
Messenger.Default.Send(new MyMessage(), nameof(MyViewModel));

// MVVM Toolkit
Messenger.Send(new MyMessage(), nameof(MyViewModel));
```

Unregister(object)

可以使用 MVVM 工具包的 `UnregisterAll(object)` 方法实现的功能 `Unregister(object)`。

C#

```
// MvvmLight
Messenger.Default.Unregister(this);

// MVVM Toolkit
Messenger.UnregisterAll(this);
```

Unregister<TMessage>(object)

`Unregister<TMessage>(object)` 的功能可以通过 MVVM 工具包的 `IMessenger` 扩展方法 `Unregister<TMessage>(object)` 实现。

C#

```
// MvvmLight
Messenger.Default.Unregister<MyMessage>(this);

// MVVM Toolkit
Messenger.Unregister<MyMessage>(this);
```

`Unregister<TMessage>(object, Action<TMessage>)`

MVVM 工具包中没有方法的直接替代 `Unregister<TMessage>(object, Action<TMessage>)` 项。

省略的原因是，对于任何给定消息类型，邮件收件人只能有一个已注册的处理程序。

建议使用 MVVM 工具包的 `IMessenger` 扩展方法 `Unregister<TMessage>(object)` 实现此功能。

C#

```
// MvvmLight
Messenger.Default.Unregister<MyMessage>(this, OnMyMessageReceived);

// MVVM Toolkit
Messenger.Unregister<MyMessage>(this);
```

`Unregister<TMessage>(object, object)`

可以使用 MVVM 工具包的 `Unregister<TMessage, TToken>(object, TToken)` 方法实现的功能 `Unregister<TMessage>(object, object)`。

C#

```
// MvvmLight
Messenger.Default.Unregister<MyMessage>(this, nameof(MyViewModel));

// MVVM Toolkit
Messenger.Unregister<MyMessage, string>(this, nameof(MyViewModel));
```

`Unregister<TMessage>(object, object, Action<TMessage>)`

MVVM 工具包中没有方法的直接替代 `Unregister<TMessage>(object, object, Action<TMessage>)` 项。

省略的原因是，对于任何给定消息类型，邮件收件人只能有一个已注册的处理程序。

建议使用 MVVM 工具包的 `Unregister<TMessage, TToken>(object, TToken)` 方法实现此功能。

C#

```
// MvvmLight
Messenger.Default.Unregister<MyMessage>(this, nameof(MyViewModel),
OnMyMessageReceived);

// MVVM Toolkit
Messenger.Unregister<MyMessage, string>(this, nameof(MyViewModel));
```

Cleanup()

方法 `Cleanup` 在 MVVM 工具包中直接替换了同名。请注意，仅当使用弱引用的信使时，此方法才有用，而在调用此方法时，`StrongReferenceMessenger` 类型将不执行任何操作，因为内部状态已在使用信使时自动剪裁。

C#

```
// MvvmLight
Messenger.Default.Cleanup();

// MVVM Toolkit
Messenger.Cleanup();
```

RequestCleanup()

MVVM 工具包中没有方法的直接替代 `RequestCleanup` 项。在 `MvvmLight` 的上下文中，`RequestCleanup` 用于启动请求，以删除由于实现利用弱引用而不再处于活动状态的注册。

对方法的任何调用 `RequestCleanup` 都可以删除或替换为 `Cleanup`。

C#

```
// MvvmLight
Messenger.Default.RequestCleanup();
```

```
// MVVM Toolkit
// No direct replacement, remove
```

ResetAll()

可以使用 MVVM 工具包的 `Reset()` 方法实现的功能 `ResetAll()`。

与将实例为 `null` 的 `MvvmLight` 实现不同，MVVM 工具包清除已注册的映射。

C#

```
// MvvmLight
Messenger.Default.ResetAll();

// MVVM Toolkit
Messenger.Reset();
```

Messenger 静态方法

OverrideDefault(IMessenger)

MVVM 工具包中没有方法的直接替代 `OverrideDefault(IMessenger)` 项。

若要使用的 `IMessenger` 自定义实现，请在服务注册中注册自定义实现，以便进行依赖关系注入，或者手动构造静态实例，并在需要时传递此实例。

C#

```
// MvvmLight
Messenger.OverrideDefault(new Messenger());

// MVVM Toolkit
// No direct replacement
```

Reset()

MVVM 工具包中的静态 `Reset` 方法无法直接替换。

可以通过调用 `Reset` 信使类型之一的静态 `Default` 实例的方法来实现相同的功能。

C#

```
// MvvmLight
Messenger.Reset();
```

```
// MVVM Toolkit
WeakReferenceMessenger.Default.Reset();
```

Messenger 静态属性

Default

Default 具有直接替换，Default 无需更改现有实现。

C#

```
// MvvmLight
IMessenger messenger = Messenger.Default;

// MVVM Toolkit
IMessenger messenger = WeakReferenceMessenger.Default;
```

迁移消息类型

MvvmLight 工具包中提供的消息类型设计为基础，供开发人员在需要时使用。

虽然 MVVM 工具包提供了一些替代方法，但这些消息类型无法直接替代。建议查看 [可用的消息类型](#)。

或者，如果解决方案利用 MvvmLight 消息类型，则可以轻松地将这些消息类型移植到你自己的代码库中。

迁移特定于平台的组件

在当前的 MVVM 工具包实现中，MvvmLight 工具包中存在的特定于平台的组件没有替代项。

以下组件及其关联的帮助程序/扩展方法没有替代项，在迁移到 MVVM 工具包时需要考虑。

Android/iOS/Windows 特定

- DialogService
- DispatcherHelper
- NavigationService

Android/iOS 特定

- `ActivityBase`
- `Binding`
- `BindingMode`
- `PropertyChangedEventManager`
- `UpdateTriggerMode`

特定于 Android

- `CachingViewHolder`
- `ObservableAdapter`
- `ObservableRecyclerViewAdapter`

特定于 iOS

- `ObservableCollectionViewSource`
- `ObservableTableViewController`
- `ObservableTableViewSource`

帮助程序

- `Empty`
- `WeakAction`
- `WeakFunc`

从 MVVM 基本版迁移

项目 • 2023/03/10

本文介绍如何迁移在 [Windows Template Studio](#) 中使用 [MVVM](#) 基本选项生成的应用，以改用 MVVM 工具包库。它适用于使用 Windows Template Studio 创建的 UWP 和 WPF 应用。

平台 API：[ObservableObject](#)，[RelayCommand](#)

本文仅重点介绍迁移，不介绍如何使用库提供的其他功能。

安装 MVVM 工具包

若要使用 MVVM 工具包，必须将 NuGet 包安装到现有应用程序中。

通过 .NET CLI 安装

```
dotnet add package CommunityToolkit.Mvvm --version 8.1.0
```

通过 PackageReference 安装

XML

```
<PackageReference Include="CommunityToolkit.Mvvm" Version="8.1.0" />
```

更新项目

迁移 Windows Template Studio 生成的代码有四个步骤。

1. 删除旧文件。
2. 替换使用 `Observable`。
3. 添加新的命名空间引用。
4. 使用不同的名称更新方法。

1. 删除旧文件

MVVM Basic 由两个文件组成

C#

```
\Helpers\Observable.cs  
\Helpers\RelayCommand.cs
```

删除这两个文件。

如果此时尝试生成项目，将看到大量错误。它们可用于识别需要更改的文件。

2. 替换使用 `Observable`

类 `Observable` 用作 ViewModels 的基类。MVVM 工具包包含一个类似的类，该类具有称为的 `ObservableObject` 附加功能。

将以前从 继承 `Observable` 的所有类更改为继承自 `ObservableObject`。

例如

C#

```
public class MainViewModel : Observable
```

将变为

C#

```
public class MainViewModel : ObservableObject
```

3. 添加新的命名空间引用

`CommunityToolkit.Mvvm.ComponentModel` 在引用 的所有文件中添加对 命名空间的 `ObservableObject` 引用。

可以手动添加相应的指令，将光标移动到 `ObservableObject` 然后按 `Ctrl+.` 以访问“快速操作”菜单，以便添加此指令。

C#

```
using CommunityToolkit.Mvvm.ComponentModel;
```

CommunityToolkit.Mvvm.Input 在引用 的所有文件中添加对 命名空间的RelayCommand引用。

可以手动添加相应的 指令，将光标移动到 ， RelayCommand 然后按 Ctrl+. 以访问“快速操作”菜单，以便添加此指令。

C#

```
using CommunityToolkit.Mvvm.Input;
```

4. 更新具有不同名称的方法

必须更新两种方法，以便为同一功能提供不同的名称。

对 的所有调用 Observable.Set 都必须替换为对 的 ObservableObject.SetProperty 调用。

因此，

C#

```
set { Set(ref _elementTheme, value); }
```

将变为

C#

```
set { SetProperty(ref _elementTheme, value); }
```

对 的所有调用 RelayCommand.OnCanExecuteChanged 都必须替换为对 的 RelayCommand.NotifyCanExecuteChanged 调用。

因此，

C#

```
(UndoCommand as RelayCommand)?.OnCanExecuteChanged();
```

将变为

C#

```
(UndoCommand as RelayCommand)?.NotifyCanExecuteChanged();
```

应用现在应使用与以前相同的功能。