C1Toolbar指令教程（WPF版）

WPF版版Toolbar 支持指令框架，在WPF中使用指令有几个目的，其最主要的目的在于分离对象来调用逻辑指令，指令可以降低原始界面与行为处理之间的耦合度。
以下将演示如何在WPF应用程序中使用C1Toolbar指令。

第一部分：使用指令库

WPF 提供了预定义指令库如下：
应用指令(剪切，复制，粘贴) 导航指令(前进，后退) 媒体指令(播放，停止，暂停) 编辑指令控件指令
许多WPF控件都内置这些支持的指令，因此您可以不用编写任何代码实现它们。
以下步骤介绍了如何使用C1Toolbar实现常用的剪切板功能。

1. 打开或创建一个新的WPF应用程序。
2. 添加两个RowDefinitions 到默认的Grid元素，为第一行设置为自动高度，如下：

```
XAML
<Grid.RowDefinitions>
<RowDefinition Height="Auto" />

<RowDefinition />
</Grid.RowDefinitions>
```

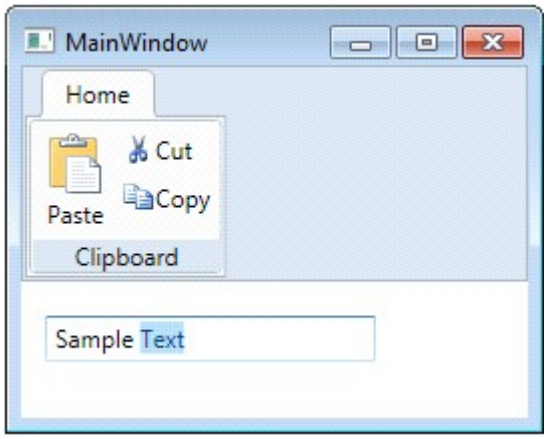
3. 添加一个C1Toolbar 以填充第一行。
4. 粘贴以下XAML并使用标签，分组和按钮来填充C1Toolbar：

```
XAML
<c1:C1Toolbar Name="c1Toolbar1" FocusManager.IsFocusScope="True">
<c1:C1ToolbarTabControl>
<c1:C1ToolbarTabItem Header="Home">
<c1:C1ToolbarGroup Header="Clipboard">
<c1:C1ToolbarButton LabelTitle="Paste"
Command="ApplicationCommands.Paste" LargeImageSource="/Resources/paste.png" />

<c1:C1ToolbarButton LabelTitle="Cut"
Command="ApplicationCommands.Cut" SmallImageSource="/Resources/cut.png" />
<c1:C1ToolbarButton LabelTitle="Copy"
Command="ApplicationCommands.Copy" SmallImageSource="/Resources/copy.png" />
</c1:C1ToolbarGroup>
</c1:C1ToolbarTabItem> </c1:C1ToolbarTabControl>
</c1:C1Toolbar>
```

该标记在C1Toolbar中创建一个包含单个分组的标签，Clipboard 分组包含三个C1ToolbarButtons 并为每一个从WPF应用指令库中分配一个常用的剪切板指令，注意C1Toolbar的IsFocusScope 属性被设置为True ，这可以使C1Toolbar 跟踪其范围内所有聚焦的元素，小图和大图都可以指定给C1ToolbarButtons，尽管这些并不是必要的。

5. 在C1Toolbar下添加一个TextBox 控件。
6. 运行应用程序并观察通过点击C1ToolbarButtons以执行常用的剪切板指令(粘贴，拷贝和剪切) 。



第二部分：创建自定义指令

如果在指令库中的指令不能满足您的需求，此时您可以创建您自己的指令，您仅需通过实现ICommand接口实现自定义指令，WPF提供了一个名为RoutedCommand(及 RoutedUICommand)的具体实现，通过元素树来定义一个指令路由。
以下步骤将介绍如何使用RoutedCommands来添加一个自定义指令到C1Toolbar。

1. 使用第一部分的示例，打开隐藏代码文件。
2. 定义一个名为ClearCommand的RoutedCommand，该指令将用于清除页面中TextBox 的文本。

```
public static RoutedCommand ClearCommand = new RoutedCommand();
```

1. 创建一个事件处理以定义指令逻辑。

```
C#
//执行ClearCommand逻辑 private void ExecutedClearCommand(object sender, ExecutedRoutedEventArgs e)

{ textBox1.Clear();
}
```

2. 创建另一个事件处理以决定指令是否能被执行，当一个指令不能被执行时，与指令相关联的Toolbar按钮将变成灰色并禁用。

```
C#
// 若文本框中有文本则返回true。
private void CanExecuteClearCommand(object sender, CanExecuteRoutedEventArgs e)

{ if (textBox1.Text.Length > 0)
{
e.CanExecute = true; } else

{
e.CanExecute = false;
}
}
```

3. 下一步，创建一个CommandBinding以关联事件处理的指令，当指令调用时，元素树将被遍历以查找

CommandBinding对象，将以下代码放到InitializeComponent 后来调用您的页面：

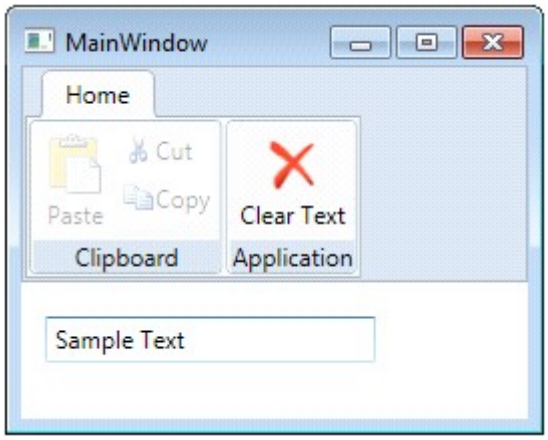
```
C#
CommandBinding customCommandBinding = new CommandBinding(ClearCommand,

ExecutedClearCommand, CanExecuteClearCommand);
// 添加CommandBinding到根元素
this.CommandBindings.Add(customCommandBinding);
```

1. 在包含C1Toolbar的XAML中，在分组中添加一个新的C1ToolbarButton并名为Application。
2. 在代码中设置Command 属性为静态指令。

```
XAML
<c1:C1ToolbarGroup Header="Application">
<c1:C1ToolbarButton LabelTitle="Clear Text" Command="{x:Static local:MainWindow.ClearCommand}" LargeImageSource="/Resources/clear.png"/> </c1:C1ToolbarGroup>
```

3. 运行应用程序并观察当工具栏按钮被点击时，ClearCommand 将通过遍历元素树查找项关联的CommandBinding，如果找到则相关联的事件处理将调用您的指令逻辑并执行。



第三部分：MVVM中使用指令

指令操作是MVVM（Model-View-ViewModel）设计模式的重要部分，它将UI从业务逻辑中分离开来，WPF版版 Toolbar支持指令框架并应用于广泛使用的MVVM模式中，在MVVM设计的应用程序中，指令目标常常在ViewModel中完成，其并不是UI元素树的部分，因此，在MVVM中使用RoutedCommand并不是一个好的ICommand实现，您可以使用一个特别的实现如DelegateCommand 或 RelayCommand，可以使您的View绑定到非UI元素树的对象。以下步骤介绍如何使用RelayCommand 类和C1Toolbar 创建第二部分相同的自定义指令。

1. 创建一个名为MainViewModel的新类以实现INotifyPropertyChanged 接口，该类将作为我们包含C1Toolbar View的ViewModel。

C#

```
class MainViewModel : System.ComponentModel.INotifyPropertyChanged
{ private string textValue = ""; public string TextValue
{ get { return textValue;
} set {
textValue = value;
OnPropertyChanged("TextValue");
} }
private RelayCommand clearCommand; public ICommand ClearCommand
{ get { if (clearCommand == null)
{ clearCommand = new RelayCommand(param => this.Clear(), param => this.CanClear()); } return clearCommand;

} }
private bool CanClear()
{ return textValue.Length > 0;
}
private void Clear()
{
TextValue = "";
} public event PropertyChangedEventHandler PropertyChanged;
protected void OnPropertyChanged(string propertyName)
{ PropertyChangedEventHandler handler = PropertyChanged;
if (handler != null)
{ handler(this, new PropertyChangedEventArgs(propertyName));
```

}
}
}
该类使用RelayCommand，它是DelegateCommand轻量级的变体，两个ICommand的实现允许您将指令逻辑作为参数委托给方法进行传递，指令目标不是UI元素树的部分，RelayCommand 和 DelegateCommand 类不是WPF框架的组成部分，且不能通过MVVM工具箱在线查找。

1. 添加RelayCommand 类到您的项目中。

C#

```
/// <summary>
/// 该类允许将指令逻辑作为参数委托给方法传递, /// 并使一个View绑定指令到非元素树的对象。
/// </summary> public class RelayCommand : ICommand
{
#region Fields
readonly Action<object> _execute; readonly Predicate<object> _canExecute;
#endregion // Fields #region Constructors
public RelayCommand(Action<object> execute)
: this(execute, null)
{ }
public RelayCommand(Action<object> execute, Predicate<object> canExecute)
{ if (execute == null) throw new ArgumentNullException("execute");
_execute = execute;
_canExecute = canExecute;
}
#endregion // 构造函数
#region ICommand Members
<ac:structured-macro ac:name="unmigrated-wiki-markup" ac:schema-version="1"
ac:macro-id="1781b40d-c8ad-4371-8626-204fc33f1574"><ac:plain-text-body><![CDATA[ [DebuggerStepThrough] public bool
CanExecute(object parameter)
]]></ac:plain-text-body></ac:structured-macro>
{ return _canExecute == null ? true : _canExecute(parameter);
} public event EventHandler CanExecuteChanged
```

```
{ add { CommandManager.RequerySuggested += value; } remove { CommandManager.RequerySuggested -= value; } }
public void Execute(object parameter)
{
_execute(parameter);
}
```

```
#endregion // ICommand成员
}
```

2. 为了绑定到ViewModel，需要添加以下XAML 到您包含C1Toolbar 的View上方:

```
XAML
<Window.Resources>
<local:MainViewModel x:Key="viewModel" />
</Window.Resources>
<Window.DataContext>
<Binding Source="{StaticResource viewModel}"/>

</Window.DataContext>
```

添加到您ViewModel 的XAML将作为资源绑定到Window或UserControl的DataContext中。

3. 此时添加以下工具栏分组到您在第一部分创建的C1Toolbar 。

```
XAML
<c1:C1ToolbarGroup Header="Application">
<c1:C1ToolbarButton LabelTitle="Clear Text" Command="{Binding ClearCommand}"

LargeImageSource="/Resources/delete.png"/>
</c1:C1ToolbarGroup>
```

该工具栏分组包含一个带有Command属性的C1ToolbarButton，Command绑定到ViewModel定义的ClearCommand。

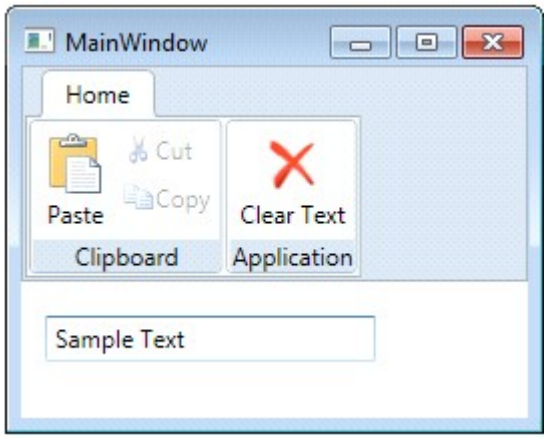
4. 由于指令遵循MVVM优秀的特性，文本框的 Text 属性也应该绑定到ViewModel的值，如果我们希望在上面应用业务逻辑，则需要绑定Text 属性到ViewModel中定义的TextValue。

```
XAML
<TextBox Grid.Row="1" Text="{Binding TextValue,
UpdateSourceTrigger=PropertyChanged}" Height="23" HorizontalAlignment="Left"

Margin="12,17,0,0" Name="textBox1" VerticalAlignment="Top" Width="165" />
```

通过将UpdateSourceTrigger 设置为 PropertyChanged，在ViewModel中的TextValue属性将可以在任何时刻更新Text值，而不是只有当文本框失去焦点时。

5. 运行应用程序并观察行为中的自定义RelayCommand 。



通过MVVM设计模式来降低UI与业务逻辑之间的耦合度，对于您的应用开发是非常实用的。