第3章

宝剑锋从磨砺出　梅花香自苦寒来
——MAUI开发理论

3.1　XAML可扩展的应用程序标记语言

3.1.1　XAML概述

XAML(eXtensible Application Markup Language)是基于XML的可扩展应用程序标记语言。XAML具有跨语言、跨平台、层次化、可读性的特点,是一种声明式的、具有层次结构的标记性语言,主要用于界面设计。XAML组件包括*.xaml界面文件和*.xaml.cs代码文件。这样做的好处是将界面设计和业务逻辑相分离,实现界面与逻辑的解耦,提升可维护性、可理解性、可测试性。XAML能够方便地实现数据绑定机制,实现MVVM设计模式。Visual Studio能够支持XAML热重载,实时修改界面代码后,调试过程无须重启,能够在UI上形成快速迭代,提升工作效率。Visual Studio还能够支持XAML可视化,提供XAML可视化树,使用属性视图能够方便地对属性进行编辑操作。目前,.NET MAUI应用程序中没有针对XAML的视觉设计器。

XAML由以下元素组成。

声明。XAML文件根节点属性。包括XAML命名空间引用声明、XAML动态库引用声明、XAML类引用声明、XAML标记声明。

对象。基于XAML描述,可创建、可实例化。包括容器、控件、资源。

资源。与对象关联的存储或定义,具有统一定义数据、样式或行为的功效。包括数据资源、样式资源、模板资源。

属性。定义控件的数据、外观或行为。包括普通属性、内容属性、附加属性。

模板。定义控件的整体效果,具有复用性。

触发器。能够根据事件或属性变化更改控件外观。

动画。提供渐变性、交互性、动态性效果。

标记扩展。自定义组件、控件，扩充标记。

3.1.2 XAML 基本语法

XAML 是标准的 XML，但是还包括自身的一些特性，如普通属性、内容属性、附加属性和标记扩展。

Visual Studio 新建. NET MAUI XAML 文件的方法是，在项目中选择文件夹后，右击，从弹出的快捷菜单中选择“菜单项添加”→“类”命令，然后选择. NET MAUI ContentPage(XAML)模板，如图 3-1 所示，并输入文件名。这样在选中的文件夹下面自动生成 XAML 文件和 XAML 文件相对应的类文件。XAML 文件相对应的类文件默认是隐藏的，需要在 Visual Studio 中展开才能看到。

图 3-1 新建. NET MAUI ContentPage(XAML)

【例 3-1】 XAML 基本语法。

NewPage1. xaml 代码如下：

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
< ContentPage xmlns = "http://schemas.microsoft.com/dotnet/2021/maui"
              xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class = "MAUIDemo.Pages.NewPage1"
              Title = "NewPage1">
    < VerticalStackLayout >
        < Label
            Text = "Welcome to .NET MAUI!"
            VerticalOptions = "Center"
```

```
            HorizontalOptions = "Center" />
    </VerticalStackLayout >
</ContentPage >
```

首行和普通 XML 一样，声明了 XML 文件的版本号和编码方式。ContentPage 是内容页面，对应的属性含义详见 3.1.4 节内容。内容部分定义了 VerticalStackLayout（垂直布局）。内部包括 Label（标签）控件，具有 Text（文本）属性，属性值显示 Welcome to. NET MAUI!，VerticalOptions 垂直对齐方式属性是 Center（居中）显示和 HorizontalOptions 水平对齐方式属性也是 Center 显示。属性统一使用键值对的方式进行定义，中间使用赋值运算符，属性值使用双引号进行包裹。XAML 中的标签可以采用单标签或双标签的形式。

NewPage1. xaml. cs 代码如下：

```
namespace MAUIDemo. Pages;

public partial class NewPage1 : ContentPage
{
    public NewPage1()
    {
        InitializeComponent();
    }
}
```

NewPage1. xaml. cs 是 NewPage1. xaml 对应的代码文件。NewPage1 类继承 ContentPage 类，属于分部类。构造方法调用 InitializeComponent()完成界面控件初始化操作。命名空间与 XAML 文件中 x:Class 相对应。

3.1.3　XAML 标记扩展

XAML 标记扩展的目的是提升扩展性和灵活性。下面直接引入示例展示 XAML 标记扩展的各种使用方式。

【例 3-2】 XAML 标记扩展。

```
<VerticalStackLayout x:Name = "vStackLayout">
    <Label
        BackgroundColor = "Blue"
        HorizontalOptions = "Center"
        Text = "属性示例"
        VerticalOptions = "Center"
        WidthRequest = "300" />
    <Label BackgroundColor = "Orange" Text = "x:StaticExtension 成员">
        <Label. WidthRequest >
            <x:StaticExtension Member = "core:XAMLConstants. WidthRequest" />
        </Label. WidthRequest >
    </Label >
    <Label BackgroundColor = "Red" Text = "x:Static 成员">
        <Label. WidthRequest >
            <x:Static Member = "core:XAMLConstants. WidthRequest" />
        </Label. WidthRequest >
    </Label >
    <Label
```

```
19          BackgroundColor = "Green"
20          Text = "x:StaticExtension 成员"
21          WidthRequest = "{x:StaticExtension Member = core:XAMLConstants.WidthRequest}" />
22      < Label
23          BackgroundColor = "Gray"
24          Text = "x:Static Member = core"
25          WidthRequest = "{x:Static Member = core:XAMLConstants.WidthRequest}" />
26      < Label
27          BackgroundColor = "Yellow"
28          Text = "x:Static core"
29          WidthRequest = "{x:Static core:XAMLConstants.WidthRequest}" />
30      < Label
31          BackgroundColor = "Violet"
32          HeightRequest = "{x:Static sys:Math.E}"
33          Scale = "5"
34          Text = "x:Static sys"
35          WidthRequest = "{x:Static sys:Math.PI}" />
36      < Slider
37          x:Name = "slider1"
38          HorizontalOptions = "Center"
39          Maximum = "100"
40          WidthRequest = "300" />
41      < Label BindingContext = "{x:Reference slider1}" Text = "{Binding Value}" />
42      < ListView
43          HorizontalOptions = "Center"
44          VerticalOptions = "Center"
45          WidthRequest = "300">
46          < ListView.ItemsSource HorizontalOptions = "Center">
47              < x:Array Type = "{x:Type sys:String}">
48                  < sys:String > x:Array AAA </sys:String >
49                  < sys:String > x:Array BBB </sys:String >
50                  < sys:String > x:Array CCC </sys:String >
51              </x:Array >
52          </ListView.ItemsSource >
53      </ListView >
54      < Label
55          BackgroundColor = "DarkOliveGreen"
56          FontFamily = "{x:Null}"
57          Text = "OnPlatform" />
58      < Label
59          BackgroundColor = "Aqua"
60          Text = "OnPlatform"
61          WidthRequest = "{OnPlatform 300,
62                                      iOS = 350,
63                                      Android = 250,
64                                      Tizen = 260,
65                                      MacCatalyst = 320}" />
66       < Label BackgroundColor = "{OnIdiom LightGreen, Phone = Yellow, Desktop = Green,
    Tablet = Orange}" Text = "OnIdiom" />
67      < Grid BackgroundColor = "{AppThemeBinding Light = {StaticResource DayModePrimaryColor},
    Dark = {StaticResource NightModePrimaryColor}}" WidthRequest = "300">
68          < Button Style = "{StaticResource ButtonTheme}" Text = "AppThemeBinding" />
69      </Grid >
70      < Label BackgroundColor = "{core:RGB R = 150, G = 80, B = 120}" Text = "IMarkupExtension" />
71  </VerticalStackLayout >
```

上述示例从上往下依次对应标记扩展的使用方式。

属性设置。类似 HTML/XML 属性语法，对控件设置相应的属性。

静态扩展。使用 x:StaticExtension 标记扩展并配置 Member 属性设置相应的属性。

静态引用。使用 x:Static 标记扩展并配置 Member 属性设置相应的属性。

插值静态扩展。使用 x:StaticExtension 标记扩展结合花括号插值语法并配置 Member 属性设置相应的属性。

插值静态引用。使用 x:Static 标记扩展结合花括号插值语法并配置 Member 属性设置相应的属性。

插值静态引用。使用 x:Static 标记扩展结合花括号插值语法并省略 Member 属性直接设置相应的属性。

命名空间引用。通过命名空间机制引入变量作为属性值。

控件引用。BindingContext(绑定上下文)属性中使用 x:Reference 标记扩展引用之前通过 x:Name 定义的控件，Binding(绑定)属性配置绑定的变量。

类型标记扩展。使用 x:Type 标记扩展，后面指定类型参数。

数组标记扩展。使用 x:Array 标记扩展，通过 Type 属性指定数组元素类型。

空值标记扩展。使用 x:Null 标记扩展，表示空。

平台标记扩展。使用 OnPlatform 标记扩展，分别设置 iOS、Android、Tizen、MacCatalyst 不同平台显示的参数值。

习语标记扩展。使用 OnIdiom 标记扩展，分别设置 Phone 手机、Desktop 桌面、Tablet 平板不同终端显示的参数值。

主题标记扩展。使用 AppThemeBinding 标记扩展，结合 StaticResource 引用相应的资源展示不同的主题。

自定义标记扩展。继承 IMarkupExtension 接口自定义标记扩展。

RGBExtension.cs 代码如下：

```
1   namespace MAUIDemo.Core
2   {
3       public class RGBExtension : IMarkupExtension<Color>
4       {
5           public int R
6           {
7               get;
8               set;
9           }
10          public int G
11          {
12              get;
13              set;
14          }
15          public int B
16          {
17              get;
18              set;
19          }
20          public Color ProvideValue(IServiceProvider serviceProvider)
```

```
21          {
22              return Color.FromRgb(R,G,B);
23          }
24       object IMarkupExtension.ProvideValue(IServiceProvider serviceProvider)
25          {
26      return (this as IMarkupExtension<Color>).ProvideValue(serviceProvider);
27          }
28      }
29  }
```

上述代码展示了自定义标记扩展的使用，RGBExtension 继承 IMarkupExtension，泛型参数是 Color(颜色)类。重写 ProvideValue()方法完成对象转换。

XAML 标记扩展涉及的语法现象示例程序的运行结果如图 3-2 所示。

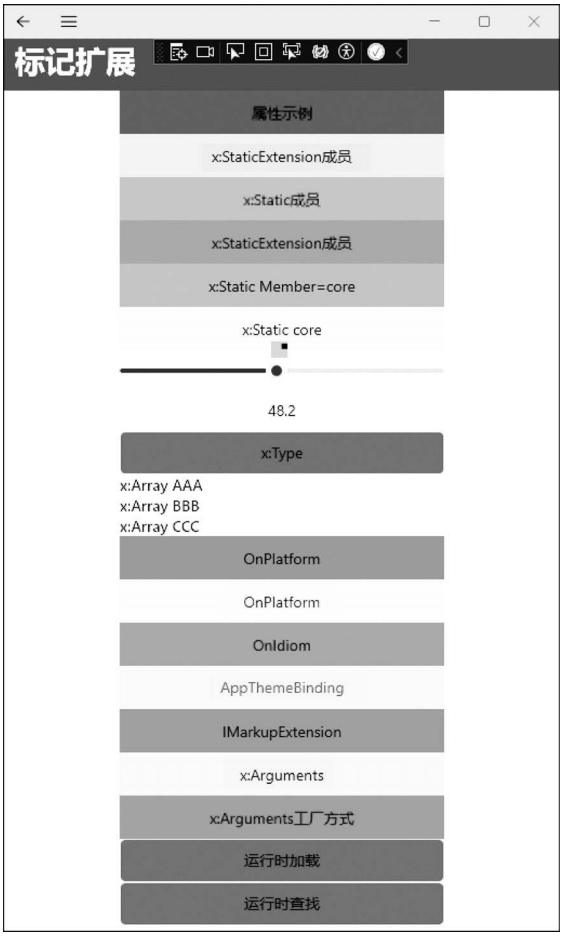

图 3-2　XAML 标记扩展涉及的语法现象示例程序的运行结果

3.1.4　XAML 命名空间

命名空间是用于组织类文件的层次化结构。例 3-3 包含了常见的命名空间引用情况。

【例 3-3】　XAMLPage.xaml 命名空间。

```
<ContentPage
    x:Class = "MAUIDemo.Pages.XAMLPage"
    xmlns = "http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:core = "clr-namespace:MAUIDemo.Core"
    xmlns:sys = "clr-namespace:System;assembly = mscorlib"
    Title = "标记扩展">
```

ContentPage 根节点包含了 6 个属性。

x:Class。声明当前组织 XAML 命名空间的类。

xmlns。XML Namespaces 声明了.NET MAUI 命名空间。

xmlns:x。声明了.NET MAUI 关于 xaml 的命名空间。

xmlns:core。引入自定义的命名空间。clr-namespace 指定命名空间前缀。

xmlns:sys。引入系统定义的命名空间。clr-namespace 指定命名空间前缀，assembly 属性指定命名空间对应的程序集。

Title。声明当前页面标题。

3.1.5　XAML 参数传递

XAML 参数传递主要通过 x:Arguments 标记扩展。

【例 3-4】　XAML 参数传递。

```
<Label Text = "x:Arguments">
    <Label.BackgroundColor>
        <Color>
            <x:Arguments>
                <x:Int32>110</x:Int32>
                <x:Int32>233</x:Int32>
                <x:Int32>89</x:Int32>
            </x:Arguments>
        </Color>
    </Label.BackgroundColor>
</Label>
<Label Text = "x:Arguments 工厂方式">
    <Label.BackgroundColor>
        <Color x:FactoryMethod = "FromRgb">
            <x:Arguments>
                <x:Int32>199</x:Int32>
                <x:Int32>22</x:Int32>
                <x:Int32>98</x:Int32>
            </x:Arguments>
        </Color>
    </Label.BackgroundColor>
</Label>
```

XAML 参数传递构造对象有如下方式。

参数构造。使用 x:Arguments 标记扩展，传入参数构造相应对象。

工厂构造。使用 x:Arguments 标记扩展，结合 x:FactoryMethod 标记扩展利用工厂方法传入参数构造相应对象。

3.1.6 XAML 动态加载

【例 3-5】 XAML 动态加载。

XAML 动态加载界面代码如下：

```
<Button
    Command="{Binding OnClick}"
    CommandParameter="{x:Type Stepper}"
    Text="x:Type" />
<Button Command="{Binding OnRuntimeClick}" Text="运行时加载" />
<Button Command="{Binding OnFindClick}" Text="运行时查找" />
```

XAML 动态加载逻辑代码如下：

```
public ICommand OnClick
{
  get;
  set;
}
public ICommand OnRuntimeClick
{
    get;
    set;
}
public ICommand OnFindClick
{
    get;
    set;
}
OnClick = new Command<Type>((Type type) =>
{
  View view = (View)Activator.CreateInstance(type);
  view.HorizontalOptions = LayoutOptions.Center;
  view.VerticalOptions = LayoutOptions.Center;
  vStackLayout.Insert(9, view);
});
OnRuntimeClick = new Command(() =>
{
    string xaml = "<Label Text=\"OnRuntimeClick\" HorizontalOptions=\"Center\"
VerticalOptions=\"Center\" BackgroundColor=\"Olive\" WidthRequest=\"300\"/>";
    Label label = new Label().LoadFromXaml(xaml);
    vStackLayout.Insert(18, label);
});
OnFindClick = new Command(async () =>
{
    string xaml = """
    <?xml version="1.0" encoding="utf-8"?>
        <ContentPage
            xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">
        <Button Text="OnFindClick"   x:Name="btn" HorizontalOptions="Center"
VerticalOptions="Center" BackgroundColor="Olive" WidthRequest="300"/>
        </ContentPage>
    """;
    ContentPage page = new ContentPage().LoadFromXaml(xaml);
```

```
40        Button btn = page.FindByName<Button>("btn");
41        await DisplayAlert("信息", btn.Text, "确认");
42    });
```

上述代码涉及的3个方法如下。

OnClick()。通过CommandParameter属性传入Stepper对象调用Activator静态方法CreateInstance()实例化，设置参数后插入垂直布局中索引值为9的位置。

OnRuntimeClick()。运行时加载。运行时通过调用LoadFromXaml()方法动态加载XAML构建的Label控件。

OnFindClick()。运行时查找。运行时通过调用FindByName()方法动态查找指定名称的按钮控件，查找后对按钮控件的属性进行弹窗展示。

3.1.7　XAML编译选项

XAML文件格式是*.xaml，一方面特定于不同平台的处理器将XAML文件*.xaml解释翻译成描述界面的中间语言；另一方面把处理后的中间语言与C#代码*.xaml.cs进行链接，二者被.NET编译器进行编译，最终形成可执行程序。.NET MAUI应用程序默认开启XAML编译功能。XAML编译器直接将XAML文件编译为中间语言，通过编译过程一方面检查语法正确性，另一方面优化代码并减少最终生成程序集文件的大小。

关闭编译功能。类文件上方添加注解[XamlCompilation (XamlCompilationOptions.Skip)]关闭XAML编译器编译功能。

开启编译功能。类文件上方添加注解[XamlCompilation(XamlCompilationOptions.Compile)]开启XAML编译器编译功能。

3.2　MAUI生命周期

MAUI包含4种状态：Not Running(未运行)、Running(运行)、Stopped(停止)和Deactivated(去激活)。其中Not Running状态尚未加载至内存。Window类的跨平台生命周期事件会导致上述状态之间的切换。MAUI生命周期状态转换如图3-3所示。

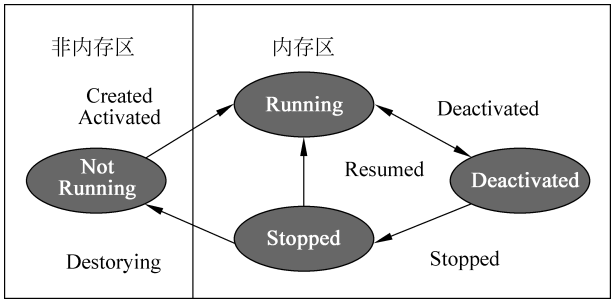

图3-3　MAUI生命周期状态转换

MAUI事件针对不同平台映射到不同的方法。

Activated。激活窗口后引发。

Created。创建窗口后引发。

Deactivated。窗口失去焦点时引发。

Destroying。窗口销毁时引发。

Resumed。应用恢复时引发。

Stopped。窗口不可见时引发。

MAUI 不同平台生命周期的对应关系见表 3-1。

表 3-1　MAUI 不同平台生命周期的对应关系

事　　件	Android 平台	iOS 平台	Windows 平台
Created	OnPostCreate	FinishedLaunching	Created
Activated	OnResume	OnActivated	Activated(CodeActivated and PointerActivated)
Deactivated	OnPause	OnResignActivation	Activated(Deactivated)
Stopped	OnStop	DidEnterBackground	VisibilityChanged
Resumed	OnRestart	WillEnterForeground	Resumed
Destroying	OnDestroy	WillTerminate	Closed

例 3-6 中的代码重写了 CreateWindow()方法，并向 Window 对象添加 Created()事件委托，内部完成个性化创建逻辑即可。读者实验时可在相应位置增加断点进行调试。

【例 3-6】 MAUI 生命周期。

App.xaml.cs 代码如下：

```
1   public partial class App : Application
2   {
3       protected override Window CreateWindow(IActivationState activationState)
4       {
5           Window window = base.CreateWindow(activationState);
6           window.Created += (sender, e) =>
7           {
8               // 创建逻辑
9           };
10          return window;
11      }
12      // 此处省略其他代码
13  }
```

3.3　MAUI 行为特性

除用子类继承控件的方式外，为现有控件增加新功能和特性还可以通过行为操作，这样做能够增强扩展性，如图 3-4 所示。

图 3-4 中的行为特性演示包括增加和删除行为。

【例 3-7】 MAUI 行为特性。

BehaviorPage.xaml 代码如下：

```
1   <?xml version = "1.0" encoding = "UTF - 8" ?>
2   < ContentPage
```

图 3-4　通过行为操作为现有控件增加新功能

```
3       x:Class = "MAUIDemo. Pages. BehaviorPage"
4       xmlns = "http://schemas. microsoft. com/dotnet/2021/maui"
5       xmlns:x = "http://schemas. microsoft. com/winfx/2009/xaml"
6       xmlns:local = "clr - namespace:MAUIDemo. Behaviors"
7       Title = "行为操作">
8       < ContentPage. Resources >
9           <!-- 此处省略其他代码 -->
10          < Style x:Key = "PositiveValidationStyle" TargetType = "Entry">
11              < Style. Setters >
12                  < Setter Property = "HorizontalOptions" Value = "Center" />
13                  < Setter Property = "WidthRequest" Value = "300" />
14                  < Setter Property = "local:PositiveValidationBehavior. AttachBehavior"
    Value = "true" />
15              </Style. Setters >
16          </Style >
17      </ContentPage. Resources >
18      < VerticalStackLayout >
19          < Entry Placeholder = "行为示例(输入正数)">
20              < Entry. Behaviors >
21                  < local:PositiveValidationBehavior />
22              </Entry. Behaviors >
23          </Entry >
24          < Entry
25              x:Name = "entry2"
26              Placeholder = "样式行为(输入正数)"
27              Style = "{StaticResource PositiveValidationStyle}" />
28          < Button Command = "{Binding OnAddClick}" Text = "增加行为" />
29          < Button Command = "{Binding OnDeleteClick}" Text = "删除行为" />
30      </VerticalStackLayout >
31  </ContentPage >
```

行为演示页面中，页面资源中定义了 Entry 控件的 HorizontalOptions 水平对齐属性、WidthRequest 宽度属性，PositiveValidationBehavior 为自定义行为属性。页面中包含两种行为使用方式。上面的 Entry 控件通过 Entry. Behaviors 标记扩展调用 PositiveValidationBehavior 实现的相关操作，下面的 Entry 控件引用了资源部分定义的样式资源，通过定义的样式资源控制行为。

BehaviorPage. xaml. cs 代码如下：

```
1   using MAUIDemo. Behaviors;
2   using System. Windows. Input;
```

```
3
4   namespace MAUIDemo.Pages;
5
6   public partial class BehaviorPage : ContentPage
7   {
8       public ICommand OnDeleteClick
9       {
10          get;
11          set;
12      }
13      public ICommand OnAddClick
14      {
15          get;
16          set;
17      }
18      public BehaviorPage()
19      {
20          InitializeComponent();
21          OnDeleteClick = new Command<Type>((Type type) =>
22          {
23              Behavior del = entry2.Behaviors.FirstOrDefault(behavior => behavior is
    PositiveValidationBehavior);
24              if (del != null)
25              {
26                  entry2.Behaviors.Remove(del);
27              }
28          });
29          OnAddClick = new Command<Type>((Type type) =>
30          {
31              entry2.Behaviors.Add(new PositiveValidationBehavior());
32          });
33          BindingContext = this;
34      }
35  }
```

控件的 Behaviors 属性包含了涉及的全部行为列表，增加行为逻辑中向该列表中追加了自定义的 PositiveValidationBehavior 行为。删除行为逻辑中遍历全部行为列表，如果找到 PositiveValidationBehavior 行为则删除。

PositiveValidationBehavior.cs 代码如下：

```
1   using Microsoft.Maui.Controls;
2
3   namespace MAUIDemo.Behaviors
4   {
5       public class PositiveValidationBehavior : Behavior<Entry>
6       {
7           public static readonly BindableProperty AttachBehaviorProperty = BindableProperty.
    CreateAttached("AttachBehavior", typeof(bool), typeof(PositiveValidationBehavior), false,
    propertyChanged: OnAttachBehaviorChanged);
8           protected override void OnAttachedTo(Entry entry)
9           {
10              entry.TextChanged += OnEntryTextChanged;
11              base.OnAttachedTo(entry);
12          }
13          protected override void OnDetachingFrom(Entry entry)
```

```
14          {
15              entry.TextChanged -= OnEntryTextChanged;
16              base.OnDetachingFrom(entry);
17          }
18          void OnEntryTextChanged(object sender, TextChangedEventArgs args)
19          {
20              _ = Int32.TryParse(args.NewTextValue, out int result);
21              bool isValid = result > 0;
22              ((Entry)sender).TextColor = isValid ? Colors.LightGreen : Colors.Red;
23          }
24          public static bool GetAttachBehavior(BindableObject view)
25          {
26              return (bool)view.GetValue(AttachBehaviorProperty);
27          }
28          public static void SetAttachBehavior(BindableObject view, bool value)
29          {
30              view.SetValue(AttachBehaviorProperty, value);
31          }
32          static void OnAttachBehaviorChanged(BindableObject view, object oldValue,
    object newValue)
33          {
34              if (view is not Entry entry)
35              {
36                  return;
37              }
38              if (Convert.ToBoolean(newValue))
39              {
40                  entry.Behaviors.Add(new PositiveValidationBehavior());
41              }
42              else
43              {
44                  Behavior del = entry.Behaviors.FirstOrDefault(behavior => behavior
    is PositiveValidationBehavior);
45                  if (del != null)
46                  {
47                      entry.Behaviors.Remove(del);
48                  }
49              }
50          }
51      }
52  }
```

具体自定义行为实现时需要继承 Behavior<T>类，控件类型作为泛型参数。自定义的 PositiveValidationBehavior 正数验证行为定义了静态只读的 BindableProperty 可绑定的属性。通过 GetAttachBehavior()和 SetAttachBehavior()实现该属性的存取和访问。可绑定属性属性值发生变化时调用 OnAttachBehaviorChanged()方法，该方法首先判断是否为 Entry 控件导致，否则直接返回，如果为 Entry 控件导致，则根据输入值情况进行行为的增减。如果输入值转换为布尔型变量为 true，追加 PositiveValidationBehavior 行为，否则删除 PositiveValidationBehavior 行为。OnAttachedTo()和 OnAttachedFrom()方法分别用于增减 OnEntryTextChanged()事件委托。OnEntryTextChanged()方法用于判断输入数据正负，如果为正则设置为浅绿色。

3.4 MAUI 手势特性

MAUI 手势特性是基于硬件特性抽象出的手势识别器。用户发起的连续手势动作导致屏幕识别位置的连续变更，对应数据参数信息状态发生变化。包括如下手势特性。

DragGestureRecognizer。拖动手势。DragStarting 是开始拖放事件。

DropGestureRecognizer。拖放手势。DragOver 是拖放过程事件，Drop 是拖放完毕事件。

PanGestureRecognizer。移动手势。PanUpdated 是移动更新事件。

TapGestureRecognizer。触摸手势。NumberOfTapsRequired 属性代表触发事件的触摸次数，Tapped 是触摸事件。

SwipeGestureRecognizer。滑动手势。Direction 属性代表滑动方向，Swiped 是滑动事件。

图 3-5 所示为手势操作的运行效果。

图 3-5 手势操作的运行效果

【例 3-8】 MAUI 手势特性。

GesturePage.xaml 代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    x:Class="MAUIDemo.Pages.GesturePage"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:viewmodels="clr-namespace:MAUIDemo.ViewModels"
    Title="手势操作">
    <VerticalStackLayout>
        <Image HeightRequest="50" Source="dotnet_bot.png">
            <Image.GestureRecognizers>
                <DragGestureRecognizer DragStarting="OnDragStarting" />
```

```
                    <DropGestureRecognizer DragOver="OnDragOver" />
                    <DropGestureRecognizer Drop="OnDrop" />
                    <PanGestureRecognizer PanUpdated="OnPanUpdated" />
        <TapGestureRecognizer NumberOfTapsRequired="2" Tapped="OnTapped" />
                </Image.GestureRecognizers>
            </Image>
            <Grid>
                <viewmodels:PinchViewModel>
                    <Image HeightRequest="50" Source="dotnet_bot.png" />
                </viewmodels:PinchViewModel>
            </Grid>
            <BoxView
                HeightRequest="50"
                WidthRequest="50"
                Color="Chartreuse">
                <BoxView.GestureRecognizers>
<SwipeGestureRecognizer Direction="Left" Swiped="OnSwipedAsync" />
SwipeGestureRecognizer Direction="Right" Swiped="OnSwipedAsync" />
<SwipeGestureRecognizer Direction="Up" Swiped="OnSwipedAsync" />
<SwipeGestureRecognizer Direction="Down" Swiped="OnSwipedAsync" />
                </BoxView.GestureRecognizers>
            </BoxView>
        </VerticalStackLayout>
</ContentPage>
```

上述代码针对控件的 GestureRecognizers 标记扩展进行手势相关的配置。Image 是图片控件，BoxView 是盒视图控件。无论是何种手势，ContentView 内容视图内部维护着 GestureRecognizers 手势识别器这个数据结构，包含多个手势识别器的链表。

GesturePage.xaml.cs 代码如下：

```
namespace MAUIDemo.Pages;

public partial class GesturePage : ContentPage
{
    double X, Y;
    public GesturePage()
    {
        InitializeComponent();
    }
    void OnDragStarting(object sender, DragStartingEventArgs e)
    {
    }
    void OnDragOver(object sender, DragEventArgs e)
    {
        e.AcceptedOperation = DataPackageOperation.None;
    }
    async void OnDrop(object sender, DropEventArgs e)
    {
    }
    void OnPanUpdated(object sender, PanUpdatedEventArgs e)
    {
        switch (e.StatusType)
        {
            case GestureStatus.Running:
```

```
                    Content.TranslationX = Math.Clamp(X + e.TotalX, -Content.Width,
    Content.Width);
                    Content.TranslationY = Math.Clamp(Y + e.TotalY, -Content.Height,
    Content.Height);
                    break;
                case GestureStatus.Completed:
                    X = Content.TranslationX;
                    Y = Content.TranslationY;
                    break;
            }
        }
        async void OnSwipedAsync(object sender, SwipedEventArgs e)
        {
            switch (e.Direction)
            {
                case SwipeDirection.Left:
                    await DisplayAlert("信息", "Left", "确认");
                    break;
                case SwipeDirection.Right:
                    await DisplayAlert("信息", "Right", "确认");
                    break;
                case SwipeDirection.Up:
                    await DisplayAlert("信息", "Up", "确认");
                    break;
                case SwipeDirection.Down:
                    await DisplayAlert("信息", "Down", "确认");
                    break;
            }
        }
        async void OnTapped(object sender, EventArgs e)
        {
            await DisplayAlert("信息", "OnTapped", "确认");
        }
    }
```

对应逻辑部分的代码展示了手势触发的各种事件执行逻辑。特别说明一下PanUpdatedEventArgs是移动手势更新事件参数,根据StatusType枚举确定移动手势更新是Running(运行)态,还是Completed(完成)态。SwipedEventArgs是滑动手势更新事件参数,根据Direction枚举确定滑动手势的滑动方向。

PinchViewModel.cs代码如下:

```
namespace MAUIDemo.ViewModels
{
    public class PinchViewModel : ContentView
    {
        double currentScale = 1;
        double startScale = 1;
        double xOffset = 0;
        double yOffset = 0;
        public PinchViewModel()
        {
            PinchGestureRecognizer pinchGesture = new();
            pinchGesture.PinchUpdated += OnPinchUpdated;
            GestureRecognizers.Add(pinchGesture);
```

```
14          }
15          private void OnPinchUpdated(object sender, PinchGestureUpdatedEventArgs e)
16          {
17              if (e.Status == GestureStatus.Started)
18              {
19                  startScale = Content.Scale;
20                  Content.AnchorX = 0;
21                  Content.AnchorY = 0;
22              }
23              if (e.Status == GestureStatus.Running)
24              {
25                  currentScale += (e.Scale - 1) * startScale;
26                  currentScale = Math.Max(1, currentScale);
27                  double renderedX = Content.X + xOffset;
28                  double deltaX = renderedX / Width;
29                  double deltaWidth = Width / (Content.Width * startScale);
30                  double originX = (e.ScaleOrigin.X - deltaX) * deltaWidth;
31                  double renderedY = Content.Y + yOffset;
32                  double deltaY = renderedY / Height;
33                  double deltaHeight = Height / (Content.Height * startScale);
34                  double originY = (e.ScaleOrigin.Y - deltaY) * deltaHeight;
35                  double targetX = xOffset - (originX * Content.Width) * (currentScale
      - startScale);
36                  double targetY = yOffset - (originY * Content.Height) * (currentScale
      - startScale);
37                  Content.TranslationX = Math.Clamp(targetX, -Content.Width * (currentScale
      - 1), 0);
38                  Content.TranslationY = Math.Clamp(targetY, -Content.Height * (currentScale
      - 1), 0);
39                  Content.Scale = currentScale;
40              }
41              if (e.Status == GestureStatus.Completed)
42              {
43                  xOffset = Content.TranslationX;
44                  yOffset = Content.TranslationY;
45              }
46          }
47      }
48  }
```

上述代码是微软公司官方提供的移动手势视图模型，用于完成手势操作的正交变换（仅涉及平移和缩放，不涉及旋转）。构造方法通过构造 PinchGestureRecognizer（移动手势识别器）并增加事件委托，将移动手势识别器追加至内部的 GestureRecognizers（手势识别器）列表数据结构中。OnPinchUpdated 事件根据手势状态 Started（开始）、Running（运行）、Completed（完成）更新相关数据，实现手势操作的正交变换。

3.5　MAUI 数据绑定

3.5.1　数据绑定概述

数据绑定是指将两个对象的属性进行关联，其中一个对象的属性发生变化引起另一个对象相关联的属性变化。涉及的两个对象称为目标对象和源对象。目标对象是继承

BindableObject 的可绑定对象。目标对象的 BindingContext 属性设置为源对象，调用 SetBinding()方法后即可完成数据绑定。数据绑定的方法分为基本绑定、高级绑定、路径绑定、条件绑定、模型绑定。

3.5.2 基本绑定

基本绑定是最简单的一种数据绑定。

【例 3-9】 基本绑定。

```
<VerticalStackLayout>
    <Label x:Name="label1" Text="文本 1" />
    <Label x:Name="label2" Text="文本 2" />
    <Label
        x:Name="label3"
        BindingContext="{x:Reference Name=slider}"
        Rotation="{Binding Path=Value}"
        Text="文本 3" />
    <Label
        x:Name="label4"
        BindingContext="{x:Reference slider}"
        Rotation="{Binding Value}"
        Text="文本 4" />
    <Label x:Name="label5" Text="文本 5">
        <Label.Rotation>
            <Binding Path="Value" Source="{x:Reference slider}" />
        </Label.Rotation>
    </Label>
    <VerticalStackLayout BindingContext="{x:Reference slider}">
        <Label
            x:Name="label6"
            Rotation="{Binding Value}"
            Text="文本 6" />
    </VerticalStackLayout>
    <Slider
        x:Name="slider"
        Maximum="360"
        VerticalOptions="Center"
        WidthRequest="300" />
    <Label Text="{Binding Source={x:Reference slider}, Path=Value, StringFormat=
'The slider value is {0:F3}'}" />
</VerticalStackLayout>
```

BindingPage.xaml.cs 代码如下：

```
namespace MAUIDemo.Pages;

public partial class BindingPage : ContentPage
{
    public BindingPage()
    {
        InitializeComponent();
        label1.BindingContext = slider;
        label1.SetBinding(Label.RotationProperty, "Value");
        label2.SetBinding(Label.RotationProperty, new Binding("Value", source: slider));
    }
}
```

上述代码使用 6 种方式进行绑定，VerticalStackLayout 垂直布局最后一个 Label 控件通过 Binding Source＝{x:Reference slider}绑定了上面定义的 Slider 滑块控件，Path＝Value 是监控滑块控件的 Value 属性，StringFormat 使用字符串插值方法完成信息的格式化展示。实际使用时，绑定方式非常灵活，对应上述 6 个标签的基本绑定方式如下。

第 1 个标签采用 C＃代码方式进行绑定，设置 BindingContext 并调用 SetBinding()方法完成数据绑定，参数代表绑定的属性是标签的 RotationProperty 旋转属性。

第 2 个标签采用 C＃代码方式进行绑定，直接调用 SetBinding()的重载方法完成数据绑定，第一个参数代表绑定的属性是标签的 RotationProperty 旋转属性，第二个参数使用 Binding 对象，采用绑定源控件属性和绑定源控件名称进行构造。

第 3 个标签采用 XAML 方式进行绑定，BindingContext 使用{x:Reference Name＝slider}指定源对象为 Slider 控件，Rotation(旋转)属性使用{Binding Path＝Value}指定绑定的是 Slider 控件的 Value 属性。

第 4 个标签采用 XAML 方式进行绑定，与第 3 个标签不同的是，BindingContext 和 Rotation 属性的表达采用简写方式。

第 5 个标签采用 XAML 方式进行绑定，采用 XAML 标记扩展< Label. Rotation >层级化方式进行绑定的定义。

第 6 个标签采用 XAML 方式进行绑定，Label 控件嵌套在 VerticalStackLayout 垂直布局内部，垂直布局为目标对象设置其 BindingContext 属性，再对嵌套在内部的 Label 控件进行 Rotation 属性设置绑定。

基本绑定示例的运行效果如图 3-6 所示。

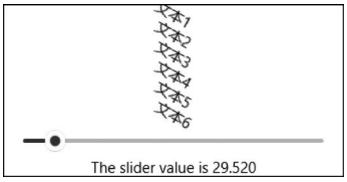

图 3-6　基本绑定示例的运行效果

3.5.3　高级绑定

高级绑定分为自身绑定、绑定回退、相对绑定和模板绑定。

【例 3-10】　自身绑定。

```
<Button
    BackgroundColor = "Yellow"
    Text = "自身绑定"
    WidthRequest = "100"
    HeightRequest = "{Binding Source = {RelativeSource Self}, Path = WidthRequest}" />
```

自身绑定是特殊的相对绑定。通过 RelativeSource 相对绑定关键字 Self 引用自身，绑定通过 Path 参数指定 WidthRequest。这样 HeightRequest 高度值和 WidthRequest 宽度值相等。

【例 3-11】　绑定回退。

```
<ContentPage.Resources>
    <sys:String x:Key = "FallBack">绑定回退</sys:String>
</ContentPage.Resources>
<Button
    BackgroundColor = "LawnGreen"
    Text = "{Binding Parameter, FallbackValue = {StaticResource FallBack}}"
```

```
    WidthRequest = "100"
    HeightRequest = "100"/>
```

绑定回退是指因找不到相应的数据导致绑定失败时，为提升用户感知而显示的临时信息，通过 FallbackValue 属性指定。上述代码引用了资源中的 FallBack 字符串关键字，对应绑定回退字样。

将相对绑定和模板绑定合并在一个示例中，下面首先定义了 GoodViewTemplate 模板资源。该模板资源通过 Frame 框架实现，Frame 框架通过 RelativeSource 相对绑定绑定到了 TemplatedParent 父模板中，意味着该框架嵌入了后面的 GoodView 自定义控件中。Frame 框架内部通过 VerticalStackLayout 垂直布局排放 3 个标签控件，第一个标签控件对 Name 商品名称进行了绑定，第二个标签控件用于分隔符显示，第三个标签控件对 Description 商品描述进行了绑定。GoodView 自定义商品视图控件的样式资源中设置了 GoodViewTemplate 控件模板属性，该控件模板属性引用了之前定义的 GoodViewTemplate 模板资源，通过这种方式实现模板绑定。

【例 3-12】 相对绑定和模板绑定。

相对绑定和模板绑定资源代码如下：

```
<ContentPage.Resources>
    <ControlTemplate x:Key = "GoodViewTemplate">
            <Frame BackgroundColor = "{TemplateBinding ControlBackgroundColor}"
 BindingContext = "{Binding Source = {RelativeSource TemplatedParent}}">
            <VerticalStackLayout>
                <Label Text = "{TemplateBinding Name}" />
                <Label BackgroundColor = "Red" HeightRequest = "20"/>
                <Label Text = "{TemplateBinding Description}"/>
            </VerticalStackLayout>
        </Frame>
    </ControlTemplate>
    <Style TargetType = "controls:GoodView">
            <Setter Property = "ControlTemplate" Value = "{StaticResource
 GoodViewTemplate}" />
    </Style>
</ContentPage.Resources>
```

GoodView 自定义商品控件继承 ContentView 类，内部定义了 BindableProperty 可绑定属性 NameProperty 商品名称属性和 DescriptionProperty 商品描述属性，这两个可绑定属性作为 Name 商品名称和 Description 商品描述访问器的参数。同时，Name 商品名称和 Description 商品描述使用 nameof 运算符作为对应可绑定属性的参数。

GoodView.cs 代码如下：

```
namespace MAUIDemo.Controls
{
    public class GoodView : ContentView
    {
        public static readonly BindableProperty NameProperty = BindableProperty.
Create(nameof(Name), typeof(string), typeof(GoodView), String.Empty);
        public static readonly BindableProperty DescriptionProperty = BindableProperty.
Create(nameof(Description), typeof(string), typeof(GoodView), String.Empty);
        public string Name
```

```
        {
            get => (string)GetValue(NameProperty);
            set => SetValue(NameProperty, value);
        }
        public string Description
        {
            get => (string)GetValue(DescriptionProperty);
            set => SetValue(DescriptionProperty, value);
        }
        public new Color BackgroundColor
        {
            get;
            set;
        }
    }
}
```

高级绑定(包括自身绑定、绑定回退、相对绑定、模板绑定)示例的运行效果如图 3-7 所示。

图 3-7　高级绑定示例的运行效果

3.5.4　路径绑定

路径绑定是特殊的数据绑定,通过设置绑定的 Path 属性指定绑定的数据对象。路径绑定通过时间选择器控件进行演示。

例 3-13 的代码中,两个 Label 控件通过设置绑定的 Path 属性分别绑定了 TimePicker(时间选择器)控件的 TotalSeconds 秒数和 DateTimeFormat.DayNames[0] 每周的第一天。

【例 3-13】 路径绑定。

```
<VerticalStackLayout>
    <TimePicker
        x:Name = "timePicker"
        HorizontalOptions = "Center"
        VerticalOptions = "Center" />
    <Label Text = "{Binding Source = {x:Reference timePicker}, Path = Time.TotalSeconds,
StringFormat = '{0}总秒数'}" />
    <Label Text = "{Binding Source = {x:Static globe:CultureInfo.CurrentCulture}, Path =
DateTimeFormat.DayNames[0], StringFormat = '每周的第一天是{0}'}" />
</VerticalStackLayout>
```

路径绑定示例的运行效果如图 3-8 所示。

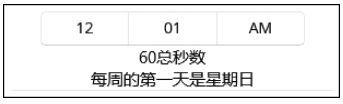

图 3-8　路径绑定示例的运行效果

3.5.5　条件绑定

条件绑定是自定义满足某些条件的绑定逻辑,自定义的类需要继承 IMultiValueConverter 接口。

【例 3-14】 条件绑定。

条件绑定资源代码如下:

```
<ContentPage.Resources>
  <converters:AllSatisfyMultiConverter x:Key="allSatisfyMultiConverter" />
<ContentPage.Resources>
```

条件绑定界面代码如下：

```
<HorizontalStackLayout HorizontalOptions="Center">
    <CheckBox>
        <CheckBox.IsChecked>
  <MultiBinding Converter="{StaticResource allSatisfyMultiConverter}">
                <Binding Path="Good.Fresh" />
                <Binding Path="Good.Inventory" />
            </MultiBinding>
        </CheckBox.IsChecked>
    </CheckBox>
</HorizontalStackLayout>
```

条件绑定界面引用了前面定义的条件绑定资源。

AllSatisfyMultiConverter.cs 代码如下：

```
using System.Globalization;

namespace MAUIDemo.Converters
{
    public class AllSatisfyMultiConverter : IMultiValueConverter
    {
        public object Convert(object[] values, Type targetType, object parameter,
CultureInfo culture)
        {
            if (values == null || !targetType.IsAssignableFrom(typeof(bool)))
            {
                return false;
            }
            foreach (var value in values)
            {
                if (value is not bool flag)
                {
                    return false;
                }
                else if (!flag)
                {
                    return false;
                }
            }
            return true;
        }
        public object[] ConvertBack(object value, Type[] targetTypes, object parameter,
CultureInfo culture)
        {
            if (value is not bool flag || targetTypes.Any(t => !t.IsAssignableFrom
(typeof(bool))))
            {
                return null;
            }
            if (flag)
            {
```

```
                return targetTypes.Select(t => (object)true).ToArray();
            }
            else
            {
                return null;
            }
        }
    }
}
```

条件绑定转换器需要实现 Convert()方法和 ConvertBack()方法用于实现类型之间的相互转换。上述示例中，如果商品满足是 Fresh(新鲜)的且有 Inventory(库存)，则 CheckBox(复选框)控件默认效果是选中。

3.5.6　模型绑定

实际应用程序中，数据绑定与 MVVM 模型紧密结合，相辅相成、相得益彰。界面中应用视图模型，将视图模型中的数据在视图中动态呈现，实现模型绑定。

首先定义 RGBViewModel 类，继承 INotifyPropertyChanged 接口的目的是保证属性数据变化时及时通知视图层，从而实现数据属性的一致性变化。定义 Color 成员变量，由 Hue(色调)、Saturation(饱和度)、Luminosity(亮度)3 个分量进行描述。分别对这 4 个属性设置访问器。

【例 3-15】 模型绑定。

RGBViewModel.cs 代码如下：

```
using System.ComponentModel;

namespace MAUIDemo.ViewModels
{
    public class RGBViewModel : INotifyPropertyChanged
    {
        private Color color;
        float hue;
        float saturation;
        float luminosity;
        public event PropertyChangedEventHandler PropertyChanged;
        public float Hue
        {
            get
            {
                return hue;
            }
            set
            {
                if (hue != value)
                {
                    Color = Color.FromHsla(value, saturation, luminosity);
                }
            }
        }
        public float Saturation
        {
```

```
            get
            {
                return saturation;
            }
            set
            {
                if (saturation != value)
                {
                    Color = Color.FromHsla(hue, value, luminosity);
                }
            }
        }
        public float Luminosity
        {
            get
            {
                return luminosity;
            }
            set
            {
                if (luminosity != value)
                {
                    Color = Color.FromHsla(hue, saturation, value);
                }
            }
        }
        public Color Color
        {
            get
            {
                return color;
            }
            set
            {
                if (color != value)
                {
                    color = value;
                    hue = color.GetHue();
                    saturation = color.GetSaturation();
                    luminosity = color.GetLuminosity();
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Hue"));
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Saturation"));
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Luminosity"));
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));
                }
            }
        }
    }
}
```

其次，在视图中定义资源，引用 RGBViewModel 视图模型。

模型绑定资源代码如下：

```
<ContentPage.BindingContext>
    <viewmodels:RGBViewModel Color="Teal" />
</ContentPage.BindingContext>
```

```
<ContentPage.Resources>
    <Style x:Key="SliderStyle" TargetType="Slider">
        <Setter Property="WidthRequest" Value="300" />
        <Setter Property="Minimum" Value="0" />
        <Setter Property="Maximum" Value="1" />
    </Style>
</ContentPage.Resources>
```

最后，在界面中定义 Grid(网格)布局，包含 BoxView(盒视图)和 3 个 Slider(滑块)控件。3 个 Slider 控件引用资源定义的样式，Slider 控件的 Value 属性绑定了 RGBViewModel 模型视图中的属性。

模型绑定界面代码如下：

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="100" />
    </Grid.RowDefinitions>
    <BoxView
        Grid.Row="0"
        WidthRequest="300"
        Color="{Binding Color}" />
    <StackLayout Grid.Row="1">
        <Slider Style="{StaticResource SliderStyle}" Value="{Binding Hue}" />
<Slider Style="{StaticResource SliderStyle}"Value="{Binding Saturation}" />
<Slider Style="{StaticResource SliderStyle}"Value="{Binding Luminosity}" />
    </StackLayout>
</Grid>
```

模型绑定示例的运行效果如图 3-9 所示。

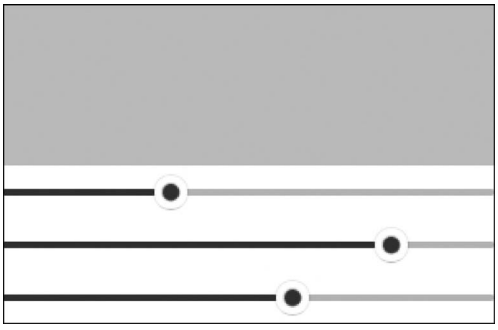

图 3-9　模型绑定示例的运行效果

3.5.7　绑定转换器

通过定义绑定转换器进行数值转换，将转换后的对象赋值为相应的属性，从而改变控件的外观或行为。定义的转换器需要继承 IValueConverter 接口，重写 Convert()和 ConvertBack()两个方法。这两个方法完成类型的相互转换。

【例 3-16】　绑定转换器。

DoubleToBoolConverter.cs 代码如下：

```
using System.Globalization;

namespace MAUIDemo.Converters
{
    public class DoubleToBoolConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
        {
            if (value == null || value.Equals(""))
            {
                return false;
            }
            _ = Double.TryParse(value.ToString(), out double res);
            return res * GetParameter(parameter) >= 0;
        }
        public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture)
        {
            int flag = GetParameter(parameter);
            return (bool)value ? flag : -flag;
        }
        private static int GetParameter(object parameter)
        {
            if (parameter is int || parameter is float || parameter is double)
                return (int)parameter;
            else if(parameter is string)
                return Int32.Parse(parameter.ToString());
            return 0;
        }
    }
}
```

下面定义了绑定转换器资源。

绑定转换器资源代码如下：

```
<ContentPage.Resources>
    <converters:DoubleToBoolConverter x:Key="doubleToBool" />
</ContentPage.Resources>
```

绑定转换器界面中的 Button 控件引用了 Entry 输入条目控件，指定采用 DoubleToBoolConverter 转换器。界面将 ConverterParameter 作为 Convert()和 ConvertBack()两个方法的第三个参数进行传递。

绑定转换器界面代码如下：

```
<VerticalStackLayout>
    <Entry
        x:Name="entry1"
        Placeholder="符号转换器"
        Text="" />
    <Button IsEnabled="{Binding Source={x:Reference entry1}, Path=Text, Converter=
{StaticResource doubleToBool}, ConverterParameter=-1}" Text="符号转换器" />
</VerticalStackLayout>
```

绑定转换器示例的运行效果如图 3-10 所示。

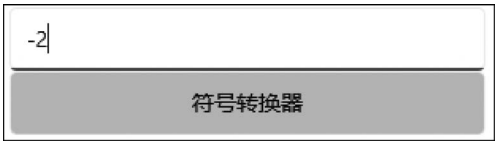

图 3-10　绑定转换器示例的运行效果

3.6　MAUI 模板介绍

3.6.1　控件模板

控件模板是为控件量身定做的个性化界面展示的组织方式。控件模板的实现方式有使用 XAML 定义或完全使用 C# 代码两种方式。本书中大量使用 XAML 定义模板，这种方式随处可见。完全使用 C# 代码的方式请读者参阅第 4 章的相关内容。

3.6.2　数据模板

数据模板是为控件量身定做的个性化数据展示的组织方式。数据模板选择器是根据不同的条件适配不同的数据模板。下面自定义的 ItemTemplateSelector(条目数据模板选择器)继承了 DataTemplateSelector(数据模板选择器)。通过 OnSelectTemplate()方法实现模板选择，该方法包含两个参数，第一个参数是模板名称前缀参数，用于根据名称前缀进行匹配，第二个参数是绑定容器对象参数。

【例 3-17】　数据模板。

ItemTemplateSelector.cs 代码如下：

```
namespace MAUIDemo.Templates
{
    public class ItemTemplateSelector : DataTemplateSelector
    {
        public DataTemplate DefaultTemplate
        {
            get;
            set;
        }
        public DataTemplate FavorTemplate
        {
            get;
            set;
        }
        protected override DataTemplate OnSelectTemplate(object item, BindableObject container)
        {
return item.ToString().Equals("Favor") ? FavorTemplate : DefaultTemplate;
        }
    }
}
```

BindableLayoutPage.xaml 在资源部分定义了两个不同的模板以及一个模板选择器。模板选择器分别对上述两个定义的模板进行了静态资源绑定。嵌套在内部的

VerticalStackLayout 垂直布局涉及如下 3 个关键属性。

BindableLayout. EmptyView。无模板绑定时的默认显示。

BindableLayout. ItemTemplateSelector。绑定模板选择器。这里引用了上面定义的数据模板选择器。

BindableLayout. ItemsSource。绑定数据源。

BindableLayoutPage. xaml 代码如下：

```
<?xml version = "1. 0" encoding = "UTF - 8" ?>
< ContentPage
    x:Class = "MAUIDemo. Pages. Layouts. BindableLayoutPage"
    xmlns = "http://schemas. microsoft. com/dotnet/2021/maui"
    xmlns:x = "http://schemas. microsoft. com/winfx/2009/xaml"
    xmlns:templates = "clr - namespace:MAUIDemo. Templates"
    xmlns:viewmodels = "clr - namespace:MAUIDemo. ViewModels"
    Title = "绑定布局">
    < ContentPage. BindingContext >
        < viewmodels:CarViewModel />
    </ContentPage. BindingContext >
    < ContentPage. Resources >
        < DataTemplate x:Key = "defaultTemplate">
            < Label BackgroundColor = "Beige" Text = "{Binding CarName}" />
        </DataTemplate >
        < DataTemplate x:Key = "favorTemplate">
            < Label BackgroundColor = "Chartreuse" Text = "{Binding CarName}" />
        </DataTemplate >
        < templates:ItemTemplateSelector
            x:Key = "itemTemplateSelector"
            DefaultTemplate = "{StaticResource defaultTemplate}"
            FavorTemplate = "{StaticResource favorTemplate}" />
    </ContentPage. Resources >
    < VerticalStackLayout >
        < Button
            Command = "{Binding OnClear}"
            HeightRequest = "40"
            HorizontalOptions = "Center"
            Text = "清空库存"
            WidthRequest = "300" />
        < Button
            Command = "{Binding OnLoad}"
            HeightRequest = "40"
            HorizontalOptions = "Center"
            Text = "加载汽车"
            WidthRequest = "300" />
        < VerticalStackLayout
            BindableLayout. EmptyView = "No Car"
            BindableLayout. ItemTemplateSelector = "{StaticResource itemTemplateSelector}"
            BindableLayout. ItemsSource = "{Binding CarList}"
            HorizontalOptions = "Center" />
    </VerticalStackLayout >
</ContentPage >
```

ContentPage. BindingContext 配置了绑定上下文，此处引用了视图模型命名空间中的 CarViewModel(汽车视图模型)。CarViewModel 继承自下面的 BaseViewModel(基本

视图模型)类。

BaseViewModel.cs 代码如下：

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace MAUIDemo.ViewModels
{
    public class BaseViewModel : IQueryAttributable, INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
     public virtual void ApplyQueryAttributes(IDictionary<string, object> query)
        {
        }
        protected virtual void OnPropertyChanged([CallerMemberName] string propertyName =
null)
        {
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

BaseViewModel 类实现了 IQueryAttributable(查询属性)接口和 INotifyPropertyChanged(属性变更)接口。OnPropertyChanged()触发属性变更事件。

Car.cs 代码如下：

```
namespace MAUIDemo.Models
{
    public class Car
    {
        public Car(string CarName)
        {
            this.CarName = CarName;
        }
        public string CarName
        {
            get;
            set;
        }
    }
}
```

本示例定义的 Car(汽车)数据结构如上。

CarViewModel.cs 代码如下：

```
using MAUIDemo.Data;
using MAUIDemo.Models;
using System.Collections.ObjectModel;
using System.Windows.Input;

namespace MAUIDemo.ViewModels
{
    public class CarViewModel: BaseViewModel
    {
        public ICommand OnClear
```

```
        {
            set;
            get;
        }
        public ICommand OnLoad
        {
            set;
            get;
        }
        public ObservableCollection<Car> CarList
        {
            get;
            set;
        }
        public CarViewModel()
        {
            CarList = Mocker.LoadCars();
            OnClear = new Command(() =>
            {
                CarList.Clear();
                OnPropertyChanged("CarList");
            });
            OnLoad = new Command(() =>
            {
                CarList = Mocker.LoadCars();
                OnPropertyChanged("CarList");
            });
        }
    }
}
```

CarViewModel 内部维护着 CarList(汽车列表),作为 ObservableCollection(可观察)属性列表,调用父类的 OnPropertyChanged()方法可实现属性变更。OnClear()方法完成清空汽车列表数据,OnLoad()方法完成加载汽车列表数据。Mocker 对象是自定义的数据模拟器,因篇幅所限,不在此展示,读者可自行查阅相关代码。

3.7 MAUI 触发器

3.7.1 触发器概述

触发器是基于 XAML 声明式语法完成,通过数据、状态、事件等的改变,进而更改控件外观的一种机制。控件通过 Triggers XAML 标记扩展来定义,视觉状态的变更通过 VisualState 视觉状态来定义。.NET MAUI 中的触发器分为普通触发器、样式触发器、数据触发器、事件触发器、条件触发器、动画触发器、状态触发器、比较触发器、设备触发器、方向触发器和自适应触发器。其中普通触发器、样式触发器、数据触发器、事件触发器、条件触发器、动画触发器是通过 Triggers XAML 标记扩展来定义的,而状态触发器、比较触发器、设备触发器、方向触发器、自适应触发器是通过视觉状态来定义的。

3.7.2　普通触发器

大部分控件可使用普通触发器，通过 Triggers XAML 标记扩展来定义。

【例 3-18】 普通触发器。

```
<Entry
    HorizontalOptions = "Center"
    Placeholder = "普通触发器"
    WidthRequest = "300">
    <Entry.Triggers>
        <Trigger TargetType = "Entry" Property = "IsFocused" Value = "True">
            <Setter Property = "BackgroundColor" Value = "BlueViolet" />
        </Trigger>
    </Entry.Triggers>
</Entry>
```

上述代码中 Trigger 标签的 TargetType 属性指向 Entry 输入条目控件自身，当 IsFocused 属性值为 True 时，此时激活触发器，导致 BackgroundColor(背景色)属性设置为 BlueViolet。

3.7.3　样式触发器

通过 Style.Triggers XAML 标记扩展定义样式触发器。

【例 3-19】 样式触发器。

样式触发器资源代码如下：

```
<ContentPage.Resources>
    <Style x:Key = "EntryTrigger" TargetType = "Entry">
        <Setter Property = "WidthRequest" Value = "300" />
        <Setter Property = "HeightRequest" Value = "40" />
        <Setter Property = "HorizontalOptions" Value = "Center" />
        <Style.Triggers>
            <Trigger TargetType = "Entry" Property = "IsFocused" Value = "True">
                <Setter Property = "BackgroundColor" Value = "DarkRed" />
            </Trigger>
        </Style.Triggers>
    </Style>
</ContentPage.Resources>
```

上述资源定义中定义了样式触发器。与普通触发器类似，当 IsFocused 属性值为 True 时，此时激活触发器，导致 BackgroundColor 属性的更改。

样式触发器界面代码如下：

```
<Entry Placeholder = "样式触发器" Style = "{StaticResource EntryTrigger}" />
```

引用定义的样式触发器资源时，只需将 Style 属性设置为静态引用即可。

3.7.4　数据触发器

数据触发器使用 DataTrigger 进行定义，当满足某些数据条件时，激活此类触发器。

【例 3-20】 数据触发器。

```
<Entry
    x:Name = "entry"
    HorizontalOptions = "Center"
    Placeholder = "数据触发器"
    Text = ""
    WidthRequest = "300" />
<Button
    HorizontalOptions = "Center"
    Text = "按钮状态"
    WidthRequest = "300">
    <Button.Triggers>
        <DataTrigger
            Binding = "{Binding Source = {x:Reference entry}, Path = Text.Length}"
            TargetType = "Button"
            Value = "0">
            <Setter Property = "BackgroundColor" Value = "YellowGreen" />
            <Setter Property = "IsEnabled" Value = "False" />
        </DataTrigger>
    </Button.Triggers>
</Button>
```

上述代码 Button 控件中的数据触发器绑定了 Entry 输入条目控件，输入条目控件的文本长度为 0 时激活数据触发器。该数据触发器的效果是将 TargetType(目标类型)控件，即 Button 控件的 BackgroundColor 变为 YellowGreen(黄绿色)，IsEnabled(是否启用)属性变为 False 不可用。

3.7.5 事件触发器

事件触发器基于事件驱动机制进行激活。

【例 3-21】 事件触发器。

事件触发器界面代码如下：

```
<Entry
    HorizontalOptions = "Center"
    Placeholder = "事件触发器"
    WidthRequest = "300">
    <Entry.Triggers>
        <EventTrigger Event = "TextChanged">
            <triggers:EntryTriggerAction />
        </EventTrigger>
    </Entry.Triggers>
</Entry>
```

上述代码通过 EventTrigger XAML 标记扩展来定义事件触发器。触发器激活后的事件由 Event 属性配置。这里的事件回调函数是 TextChanged。

EntryTriggerAction.cs 代码如下：

```
using System.Text.RegularExpressions;

namespace MAUIDemo.Triggers
{
```

```
    public class EntryTriggerAction : TriggerAction<Entry>
    {
        protected override void Invoke(Entry entry)
        {
    entry.TextColor = IsNumeric(entry.Text) ? Colors.LightGreen : Colors.Red;
        }
        /// <summary>
        /// 判断是否是数字
        /// </summary>
        /// <param name="value">字符串</param>
        /// <returns>是否是数字</returns>
        public static bool IsNumeric(string value)
        {
            return Regex.IsMatch(value, @"^SymbolYCp[ +- ]?\d*[.]?\d* $");
        }
    }
}
```

事件触发器的动作类 EntryTriggerAction 需要继承 TriggerAction<T>，其中泛型参数需要指明控件类型。重写 Invoke()方法执行激活后的逻辑。IsNumeric()方法通过正则表达式判断是否是数字。根据判断结果，修改条目控件的文字颜色。

3.7.6　条件触发器

条件触发器是满足指定的条件后才激活相应的逻辑。例 3-22 中定义了两个 Slider 控件，MultiTrigger 是多个条件 XAML 标记扩展，MultiTrigger.Conditions 指定了需要满足的条件集合，BindingCondition 是单个条件。通过 Binding 属性绑定滑块控件的值，当满足设置的 Value(即 Maximum 最大值)时，条件触发器生效，将 BackgroundColor 属性设置为 LightGreen(浅绿色)。

【例 3-22】 条件触发器。

```
<Slider
    x:Name="slider1"
    Maximum="100"
    WidthRequest="300" />
<Slider
    x:Name="slider2"
    Maximum="100"
    WidthRequest="300" />
<Button Text="条件触发器" WidthRequest="300">
    <Button.Triggers>
        <MultiTrigger TargetType="Button">
            <MultiTrigger.Conditions>
                <BindingCondition Binding="{Binding Source={x:Reference slider1},
Path=Value}" Value="100" />
                <BindingCondition Binding="{Binding Source={x:Reference slider2},
Path=Value}" Value="100" />
            </MultiTrigger.Conditions>
            <Setter Property="BackgroundColor" Value="LightGreen" />
        </MultiTrigger>
    </Button.Triggers>
</Button>
```

同时将两个滑块控件的值手动设置到最大即可激活条件触发器，执行前后的效果分别如图 3-11 和图 3-12 所示。

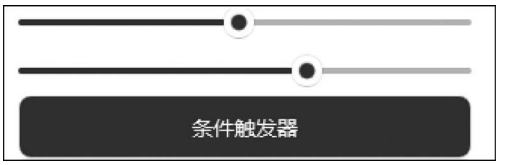

图 3-11　条件触发器触发前状态

图 3-12　条件触发器触发后状态

3.7.7　动画触发器

动画触发器通过定义两个状态，这里的状态类似动画制作中的关键帧。动画触发器被激活后，不同状态之间的切换过程产生动画过渡效果。

【例 3-23】　动画触发器。

动画触发器界面代码如下：

```
<Entry Placeholder = "动画触发器" WidthRequest = "300">
    <Entry.Triggers>
        <Trigger TargetType = "Entry" Property = "Entry.IsFocused" Value = "True">
            <Trigger.EnterActions>
                <triggers:AnimationTriggerAction Begin = "0" />
            </Trigger.EnterActions>
            <Trigger.ExitActions>
                <triggers:AnimationTriggerAction Begin = "1" />
            </Trigger.ExitActions>
        </Trigger>
    </Entry.Triggers>
</Entry>
```

上述代码通过 Trigger.EnterActions 定义两个动画，第一个动画的状态是 Begin 属性为 0，第二个动画的状态是 Begin 属性为 1。两个状态之间的过渡逻辑需要自定义。

AnimationTriggerAction.cs 代码如下：

```
namespace MAUIDemo.Triggers
{
    public class AnimationTriggerAction : TriggerAction<VisualElement>
    {
        public int Begin
        {
            get;
            set;
        }
        protected override void Invoke(VisualElement sender)
        {
            sender.Animate("AnimationTriggerAction", new Animation((d) =>
            {
                double val = Begin == 1 ? d : 1 - d;
                sender.BackgroundColor = Color.FromRgb(1, val, 1);
            }),
            length: 1500,
            easing: Easing.Linear);
```

```
        }
    }
}
```

自定义的 AnimationTriggerAction 同样需要继承 TriggerAction < T >类，这里的泛型参数指定为 VisualElement(可视化元素)。内部包含自定义的 Begin 属性用于区分状态。重写的 Invoke()方法中，设置了 Animation 对象。其中第三个参数指明了需要动画的缓动函数。关于缓动动画的内容，读者可参阅本书第 4 章中的相关内容。

3.7.8　状态触发器

状态触发器和动画触发器类似，只不过是将 VisualElement 对象换成了 VisualState 对象。以下是关于状态触发器的 XAML 标记扩展。

VisualStateGroupList。可视化状态组列表。用于描述一系列可视化状态组。

VisualStateGroup。可视化状态组。用于描述一系列可视化状态。

VisualState。可视化状态。用于描述单个可视化状态。

VisualState. StateTriggers。可视化状态的状态触发器。定义状态触发条件。

VisualState. Setters。可视化状态的状态触发器激活后的设置。定义状态触发器激活后设置的属性。

【例 3-24】 状态触发器。

状态触发器资源代码如下：

```
<ContentPage.Resources>
    <Style x:Key="GridTrigger" TargetType="Grid">
        <Setter Property="VisualStateManager.VisualStateGroups">
            <VisualStateGroupList>
                <VisualStateGroup>
                    <VisualState x:Name="Checked">
                        <VisualState.StateTriggers>
<StateTrigger IsActive="{Binding IsChecked}" IsActiveChanged="OnChecked" />
                        </VisualState.StateTriggers>
                        <VisualState.Setters>
<Setter Property="BackgroundColor" Value="LightBlue" />
                        </VisualState.Setters>
                    </VisualState>
                    <VisualState x:Name="Unchecked">
                        <VisualState.StateTriggers>
                            <StateTrigger IsActive="{Binding IsChecked}" IsActive
Changed="OnUnchecked" />
                        </VisualState.StateTriggers>
                        <VisualState.Setters>
                     <Setter Property="BackgroundColor" Value="LightSalmon" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateGroupList>
        </Setter>
    </Style>
</ContentPage.Resources>
```

上述代码在资源中定义了状态触发器。IsChecked 属性选中时触发 OnChecked 事件，不选中时触发 OnUnchecked 事件。触发执行逻辑就是将 BackgroundColor 属性设置为相应的颜色。

状态触发器界面代码如下：

```
<HorizontalStackLayout HorizontalOptions = "Center">
    <CheckBox
        x:Name = "checkBox1"
        HorizontalOptions = "Center"
        IsChecked = "False" />
    <Label Text = "状态触发器" VerticalOptions = "Center" />
</HorizontalStackLayout>
<Grid
    BindingContext = "{x:Reference checkBox1}"
    HorizontalOptions = "Center"
    Style = "{StaticResource GridTrigger}">
    <Grid.RowDefinitions>
        <RowDefinition Height = "50" />
        <RowDefinition Height = "50" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width = "150" />
        <ColumnDefinition Width = "150" />
    </Grid.ColumnDefinitions>
    <Label
        Grid.Row = "0"
        Grid.Column = "0"
        Text = "0 行 0 列" />
    <Label
        Grid.Row = "0"
        Grid.Column = "1"
        Text = "0 行 1 列" />
    <Label
        Grid.Row = "1"
        Grid.Column = "0"
        Text = "1 行 0 列" />
    <Label
        Grid.Row = "1"
        Grid.Column = "1"
        Text = "1 行 1 列" />
</Grid>
```

状态触发器界面中定义了两行两列的表格，当 CheckBox 控件状态改变时触发相应的事件。

TriggerPage.xaml.cs 代码如下：

```
void OnChecked(object sender, EventArgs e)
{
    StateTriggerBase stateTrigger = sender as StateTriggerBase;
    Console.WriteLine( $ "Checked state active: {stateTrigger.IsActive}");
}
void OnUnchecked(object sender, EventArgs e)
{
    StateTriggerBase stateTrigger = sender as StateTriggerBase;
```

```
9        Console.WriteLine( $ "Unchecked state active: {stateTrigger.IsActive}");
10   }
```

事件函数中将参数 sender 转换为 StateTriggerBase 对象。通过 IsActive 属性来检测触发器是否被激活。

3.7.9　比较触发器

比较触发器资源定义的语法与状态触发器类似，只不过是将 StateTrigger 换成 CompareStateTrigger。

【例 3-25】 比较触发器。

比较触发器资源代码如下：

```
1    < ContentPage.Resources >
2       < Style x:Key = "CompareTrigger" TargetType = "Grid">
3           < Setter Property = "VisualStateManager.VisualStateGroups">
4               < VisualStateGroupList >
5                   < VisualStateGroup >
6                       < VisualState x:Name = "Checked">
7                           < VisualState.StateTriggers >
8                               < CompareStateTrigger Property = "{Binding Source = {x:
    Reference checkBox2}, Path = IsChecked}" Value = "True" />
9                           </VisualState.StateTriggers >
10                          < VisualState.Setters >
11                              < Setter Property = "BackgroundColor" Value = "Blue" />
12                          </VisualState.Setters >
13                      </VisualState >
14                      < VisualState x:Name = "Unchecked">
15                          < VisualState.StateTriggers >
16                              < CompareStateTrigger Property = "{Binding Source = {x:
    Reference checkBox2}, Path = IsChecked}" Value = "False" />
17                          </VisualState.StateTriggers >
18                          < VisualState.Setters >
19                              < Setter Property = "BackgroundColor" Value = "Green" />
20                          </VisualState.Setters >
21                      </VisualState >
22                  </VisualStateGroup >
23              </VisualStateGroupList >
24          </Setter >
25      </Style >
26   </ContentPage.Resources >
```

比较触发器界面中定义了两行两列的表格，当 CheckBox 控件状态改变时触发相应的事件。

比较触发器界面代码如下：

```
1    < HorizontalStackLayout HorizontalOptions = "Center">
2        < CheckBox
3            x:Name = "checkBox2"
4            HorizontalOptions = "Center"
5            IsChecked = "False" />
6        < Label Text = "比较触发器" VerticalOptions = "Center" />
7    </HorizontalStackLayout >
8    < Grid
```

```
        BindingContext = "{x:Reference checkBox2}"
        HorizontalOptions = "Center"
        Style = "{StaticResource CompareTrigger}">
        < Grid.RowDefinitions >
            < RowDefinition Height = "50" />
            < RowDefinition Height = "50" />
        </Grid.RowDefinitions >
        < Grid.ColumnDefinitions >
            < ColumnDefinition Width = "150" />
            < ColumnDefinition Width = "150" />
        </Grid.ColumnDefinitions >
        < Label
            Grid.Row = "0"
            Grid.Column = "0"
            Text = "0 行 0 列" />
        < Label
            Grid.Row = "0"
            Grid.Column = "1"
            Text = "0 行 1 列" />
        < Label
            Grid.Row = "1"
            Grid.Column = "0"
            Text = "1 行 0 列" />
        < Label
            Grid.Row = "1"
            Grid.Column = "1"
            Text = "1 行 1 列" />
    </Grid >
```

手动勾选或取消复选框控件的选项，看到表格的颜色会随之改变。

3.7.10 设备触发器

设备触发器是针对不同的设备触发不同的效果，也是通过视觉状态进行定义的，这里使用 DeviceStateTrigger（设备触发器）类进行定义。参数 Device 指明设备类型。UWP 指通用 Windows 操作系统设备、Android 指安卓设备等。

【例 3-26】 设备触发器。

设备触发器资源代码如下：

```
    < ContentPage.Resources >
        < Style x:Key = "DeviceTrigger" TargetType = "Button">
            < Setter Property = "HorizontalOptions" Value = "Center" />
            < Setter Property = "WidthRequest" Value = "300" />
            < Setter Property = "VisualStateManager.VisualStateGroups">
                < VisualStateGroupList >
                    < VisualStateGroup >
                        < VisualState x:Name = "UWP">
                            < VisualState.StateTriggers >
                                < DeviceStateTrigger Device = "UWP" />
                            </VisualState.StateTriggers >
                            < VisualState.Setters >
                        < Setter Property = "BackgroundColor" Value = "LightGreen" />
                            </VisualState.Setters >
                        </VisualState >
```

```
                        <VisualState x:Name = "Android">
                            <VisualState.StateTriggers>
                                <DeviceStateTrigger Device = "Android" />
                            </VisualState.StateTriggers>
                            <VisualState.Setters>
                          <Setter Property = "BackgroundColor" Value = "LightBlue" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateGroupList>
            </Setter>
        </Style>
    </ContentPage.Resources>
```

设备触发器资源的定义完全类似,引用时通过 Style 属性指定对应的资源即可。

设备触发器界面代码如下:

```
<Button Style = "{StaticResource DeviceTrigger}" Text = "设备触发器" />
```

3.7.11　方向触发器

方向触发器基于判断设备方向从而决定触发器是否被激活。也是通过视觉状态进行定义的,这里使用 OrientationStateTrigger(方向触发器)类进行定义。Orientation 属性指明方向。

【例 3-27】 方向触发器。

方向触发器资源代码如下:

```
<ContentPage.Resources>
    <Style x:Key = "OrientationTrigger" TargetType = "Button">
        <Setter Property = "HorizontalOptions" Value = "Center" />
        <Setter Property = "WidthRequest" Value = "300" />
        <Setter Property = "VisualStateManager.VisualStateGroups">
            <VisualStateGroupList>
                <VisualStateGroup>
                    <VisualState x:Name = "Portrait">
                        <VisualState.StateTriggers>
                            <OrientationStateTrigger Orientation = "Portrait" />
                        </VisualState.StateTriggers>
                        <VisualState.Setters>
                      <Setter Property = "BackgroundColor" Value = "LightGreen" />
                        </VisualState.Setters>
                    </VisualState>
                    <VisualState x:Name = "Landscape">
                        <VisualState.StateTriggers>
                           <OrientationStateTrigger Orientation = "Landscape" />
                        </VisualState.StateTriggers>
                        <VisualState.Setters>
                        <Setter Property = "BackgroundColor" Value = "SeaGreen" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateGroupList>
        </Setter>
```

```
    </Style>
</ContentPage.Resources>
```

方向触发器资源的定义完全类似,引用时通过 Style 属性指定对应的资源即可。

方向触发器界面代码如下:

```
<Button Style="{StaticResource OrientationTrigger}" Text="方向触发器" />
```

3.7.12 自适应触发器

自适应触发器是根据容器中文字的多少来触发相应的显示状态。观察自适应触发器示例前后状态的运行效果分别如图 3-13 和图 3-14 所示。

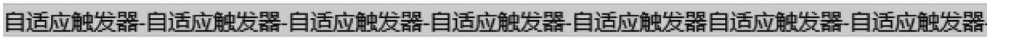

图 3-13 自适应触发器触发前状态

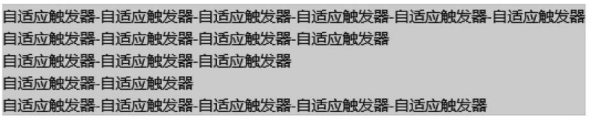

图 3-14 自适应触发器触发后状态

自适应触发器也是通过视觉状态进行定义的,这里使用 AdaptiveTrigger(自适应触发器)类进行定义。MinWindowWidth 属性指明最小视窗宽度。

【例 3-28】 自适应触发器。

自适应触发器资源代码如下:

```
<ContentPage.Resources>
    <Style x:Key="AdaptiveTrigger" TargetType="StackLayout">
        <Setter Property="VisualStateManager.VisualStateGroups">
            <VisualStateGroupList>
                <VisualStateGroup>
                    <VisualState x:Name="Vertical">
                        <VisualState.StateTriggers>
                            <AdaptiveTrigger MinWindowWidth="0" />
                        </VisualState.StateTriggers>
                        <VisualState.Setters>
                            <Setter Property="Orientation" Value="Vertical" />
                        </VisualState.Setters>
                    </VisualState>
                    <VisualState x:Name="Horizontal">
                        <VisualState.StateTriggers>
                            <AdaptiveTrigger MinWindowWidth="800" />
                        </VisualState.StateTriggers>
                        <VisualState.Setters>
                           <Setter Property="Orientation" Value="Horizontal" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateGroupList>
        </Setter>
```

```
    </Style>
</ContentPage.Resources>
```

自适应触发器资源的定义完全类似，引用时通过 Style 属性指定对应的资源即可。

自适应触发器界面代码如下：

```
<StackLayout
    BackgroundColor="Gold"
    HorizontalOptions="Center"
    Style="{StaticResource AdaptiveTrigger}">
    <Label Text="自适应触发器-自适应触发器-自适应触发器-自适应触发器-自适应触发器-自适应触发器" />
    <Label Text="自适应触发器-自适应触发器-自适应触发器-自适应触发器" />
    <Label Text="自适应触发器-自适应触发器-自适应触发器" />
    <Label Text="自适应触发器-自适应触发器" />
    <Label Text="自适应触发器-自适应触发器-自适应触发器-自适应触发器-自适应触发器" />
</StackLayout>
```

自适应触发器界面代码中专门设置了 5 个 Label 控件，内容文字长短不一。读者可通过变更视窗长度和宽度自行查看运行效果。

3.8　MAUI 消息通信

3.8.1　消息概述

消息是一种进程间通信机制，基于生产者-消费者模型实现。生产者-消费者是操作系统中经典的多线程并发协作机制，尤其在分布式场合使用广泛。生产者和消费者之间存在缓冲区，生产者生产数据后，暂时存放在缓冲区中，待消费者空闲时取走。缓冲区为空时，消费者阻塞；缓冲区为满时，生产者阻塞。缓冲区的引入目的是解决生产者和消费者之间处理速度不匹配问题。解决思路是定义信号量，并利用锁机制实现。缓冲区使用队列这种数据结构，可以根据实际需求采用循环队列。先入先出的这种方式接收或取走相应的数据。将生产者-消费者模型具体化，衍生出了发布-订阅模型。实现发布-订阅模型比较成熟的中间件有 ActiveMQ、RabbitMQ、RocketMQ、Kafka 等。

.NET MAUI 消息通信使用 MessagingCenter 对象封装了底层实现细节。使用基于事件机制的发布-订阅模型，降低事件发布方和订阅方之间的耦合关系，实现多发布方和多订阅方。内部使用弱引用机制无须使数据保持实时活跃状态，必要时可进行垃圾回收(Garbage Collection，GC)。用户通过调用相应的方法即可快速实现消息的订阅、发布和取消。本节涉及的示例集中在 MessagingCenterPage 页面中，如图 3-15 所示。

【例 3-29】　消息中心。

MessagingCenterPage.xaml.cs 代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    x:Class="MAUIDemo.Pages.MessagingCenterPage"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
```

图 3-15　消息概述

```
5       xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
6       Title = "消息中心">
7       < VerticalStackLayout >
8           < Entry
9               x:Name = "entry"
10              Placeholder = "消息内容"
11              Text = ""
12              WidthRequest = "300" />
13          < Button
14              Command = "{Binding OnSend}"
15              Style = "{StaticResource NormButton}"
16              Text = "消息发布" />
17          < Button
18              Command = "{Binding OnSubscribe}"
19              Style = "{StaticResource NormButton}"
20              Text = "消息订阅" />
21          < Button
22              Command = "{Binding OnUnsubscribe}"
23              Style = "{StaticResource NormButton}"
24              Text = "取消订阅" />
25      </VerticalStackLayout >
26  </ContentPage >
```

MessagingCenterPage 使用 VerticalStackLayout 垂直布局。内部包含 Entry 控件用于输入待测试的消息内容，3 个 Button 控件分别用于测试消息发布、消息订阅和取消订阅。

3.8.2　消息发布

消息发布是消息发布者将用户输入的消息进行发布。

【例 3-30】　消息发布。

```
1  OnSend = new Command(() =>
2  {
3      MessagingCenter.Send < MessagingCenterPage, string >(this, "message", entry.Text);
4  });
```

Send()方法完成消息发布，涉及的 5 个参数如下。

(1) 泛型参数一是发布消息的类型。上述代码对应当前页面。

（2）泛型参数二是发布消息的有效负载类型。上述代码对应字符串类型。

（3）形参一是发布消息的类型。上述代码对应当前页面。

（4）形参二是发布消息的主题。主题可以理解为感兴趣的频道。上述代码对应的主题是 message。

（5）形参三是发布消息的有效负载类型。上述代码对应字符串类型。

3.8.3　消息订阅

消息订阅是消息订阅者对感兴趣的消息进行订阅。

【例 3-31】 消息订阅。

```
1  OnSubscribe = new Command(() =>
2  {
3      MessagingCenter.Subscribe<MessagingCenterPage, string>(this, "message", async
   (sender, arg) =>
4      {
5          await DisplayAlert("订阅消息", "消息参数>" + arg, "确认");
6      });
7  });
```

Subscribe()方法完成消息订阅，涉及的 5 个参数如下。

（1）泛型参数一是订阅消息的类型。上述代码对应当前页面。

（2）泛型参数二是订阅消息的有效负载类型。上述代码对应字符串类型。

（3）形参一是订阅消息的类型。上述代码对应当前页面。

（4）形参二是订阅消息的主题。主题可以理解为感兴趣的频道。上述代码对应的主题是 message。

（5）形参三是订阅消息的回调函数。回调函数包含两个参数，参数一是发布者，参数二是消息参数内容。

3.8.4　取消订阅

取消订阅是用户取消对指定主题的订阅，即不再接收此主题或频道的消息信息。

【例 3-32】 取消订阅。

```
1  OnUnsubscribe = new Command(() =>
2  {
3  MessagingCenter.Unsubscribe<MessagingCenterPage, string>(this, "message");
4  });
```

Unsubscribe()方法完成取消订阅，涉及的 4 个参数如下。

（1）泛型参数一是取消订阅消息的类型。上述代码对应当前页面。

（2）泛型参数二是取消订阅消息的有效负载类型。上述代码对应字符串类型。

（3）形参一是取消订阅消息的类型。上述代码对应当前页面。

（4）形参二是取消订阅消息的主题。主题可以理解为感兴趣的频道。上述代码对应的主题是 message。