第 5 章 组　　件

【学习目标】

- 理解 HarmonyOS 应用中的 UI 组件
- 掌握组件的通用属性，会自定义属性
- 掌握组件事件的配置方式，会使用通用的事件
- 理解状态管理模型，掌握组件状态，会用常用的状态装饰器

5.1　概述

在 HarmonyOS 应用开发中，组件是为应用开发提供的界面元素，例如文本框、图像、按钮、进度条等。组件是应用界面的基本构成单元，借助组件，开发者可以高效地构建自己的应用图形界面。

组件是绘制在设备屏幕上的一个对象，组件也是组成应用用户界面的基本单位。组件需要放置到组件容器中。组件容器可以容纳组件，也可以容纳组件容器，组件容器可以理解成特殊的组件，它是可以容纳其他内容的组件，组件容器可以称为容器组件或布局组件，也称为布局。可以说应用中绝大多数的用户界面效果是由组件容器和普通组件对象构成的，二者通过包含和被包含、相互配合形成丰富的用户界面，应用界面中的组件容器和普通组件在组织上是一棵树形结构。

在基于 ArkTS 的 HarmonyOS 应用开发中，系统提供了丰富的组件，如 Text、TextArea、Button、Image、Slider 等。系统也提供了很多容器组件，如 Row、Column、Flex、Navigator、List、Tabs 等。尽管这里把组件分为容器组件和普通组件，但实际上有的普通组件也可以容纳其他的组件，因此容器组件和普通组件实际上没有明确的界限，它们都是组件。

在应用界面中，实例化组件的一般语法如下：

```
组件名([参数]){                                    //参数视组件而定，是可选的
    //子组件，如果没有子组件，则此部分和花括号都可以省略
}.链式调用组件属性方法()                           //组件属性方法一般也有参数
```

实例化组件也可以称为创建组件,例如,创建一个基本的按钮组件的代码如下:

```
Button('Ok')
    .fontSize(20)
    .borderRadius( $r('sys.float.ohos_id_corner_radius_button') )
    .backgroundColor( 0x666666 )
    .width(90).height(90)
```

在创建组件时,可以包含子组件,子组件需要用花括号括起来,而且组件可以嵌套包含。例如下面创建一个按钮组件,其内容包含 Row 容器组件,Row 中又包括 Image 和 Text 组件,代码如下:

```
Button() {
    Row() {
        Image( $r('app.media.loading'))   //要求存在 loading 资源图片
            .width(20).height(20).margin({ left: 12 })
        Text('加载中...')
            .fontSize(12).fontColor(0xffffff)
    }.alignItems( VerticalAlign.Center )
}.borderRadius(8).backgroundColor(0x317aff).width(90)
```

应用构建界面必须自定义组件,自定义组件是由装饰器@Component 装饰的 struct 结构体,自定义组件内容由其生成器函数进行构造,其内部可以实例化系统的内置组件和别的自定义组件。下面是构建的一个简单的登录界面例子,代码如下:

```
//ch05/LoginUI 项目中 Index.ets 文件
@Entry                                                    //入口组件装饰器
@Component                                                //组件装饰器
struct Index {                                            //定义组件
  build() {                                               //生成器函数
    Column() {                                            //列容器组件
      Image( $r('app.media.icon'))                        //图片组件
        .height(100).width(100)                           //设置高、宽属性
        .margin({ top: 150 })                             //设置边距
      TextInput().width("80%").height(50).margin(10)      //输入框组件
      TextInput().width("80%").height(50).margin(10)
        .type(InputType.Password)
      Button("登录")                                      //按钮组件
        .width("60%")
        .height(50)
        .margin(20)
    }.height("100%")                                      //设置 Column 的高度
    .width("100%")                                        //设置 Column 的宽度
  }
}
```

以上代码构建的界面效果如图 5-1 所示。

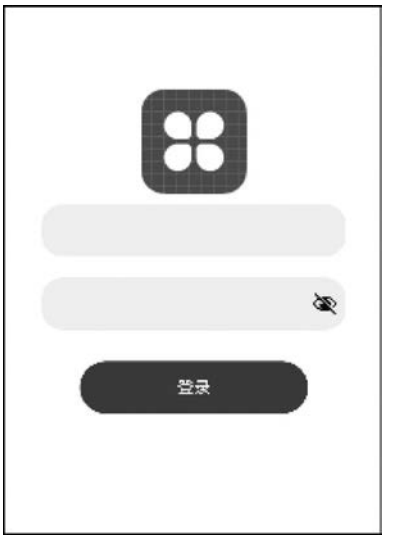


图 5-1 简单的登录界面

5.2 组件属性

组件的显示效果一般是由组件的属性决定的，组件属性有通用属性和自定义属性，通用属性是所有组件都有的属性，如大小、位置、边框、背景等，自定义属性是一些组件特有的属性，以实现自己特有的样式。下面介绍一些通用属性和自定义属性，希望读者能够举一反三，设计出符合需求的组件。

5.2.1 通用属性

1. 尺寸

尺寸是用来设置组件大小的，如宽、高等。一般组件具有的尺寸属性及说明见表 5-1。

表 5-1 组件的尺寸属性及说明

名 称	说 明	取值举例
width	宽度，缺省时使用元素自身内容需要的宽度，值为 Length 类型	Button("您好") .width(100) .height(50)
height	高度，缺省时使用元素自身内容需要的高度，值为 Length 类型	Button("您好") .width('100%') .height('30px')

size	大小，可以同时设置宽和高，值为 JSON 对象，内部包含宽度和高度，宽和高类型为 Length 类型	{
		width?:Length,
		height?:Length
		}

续表

名　称	说　明	取值举例
padding	内边距，可以同时设置4个方向内边距，也可设置指定方向的内边距，设置4个方向时，采用JSON对象参数，内部包含4个方向的值都是Length类型，当设置一个值时表示同时设置4个方向内边距一致	{ top?:Length, right?:Length, bottom?:Length, left?:Length }或 Length
margin	外边距，方式同上	同上

constraintSize	设置约束尺寸，对组件布局进行尺寸范围限制，包括宽度和高度的最小值和最大值	{
		minWidth?:Length,
		maxWidth?:Length,
		minHeight?:Length,
		maxHeight?:Length
		}

layoutWeight	组件在布局中的大小权重，在容器尺寸确定时，元素与兄弟节点主轴布局尺寸按照权重进行分配，默认自适应占满剩余空间。该属性仅在Row/Column/Flex有效	值为number类型数值

在描述尺寸时一般会用到Length类型值，Length类型是系统定义的类型，它可以是字符串(string)、数值(number)和资源(resource)。

在使用字符串表示尺寸大小时，可以显式指定像素单位，如'30px'、'30vp'等，也可设置百分比字符串，如'80%'。

在使用数值表示尺寸大小时，可以直接使用数值，其默认单位是vp，如30。

这里资源是使用引入资源的方式，使用系统资源或者应用资源中的尺寸。

2. 位置

位置属性顾名思义是设置组件的位置关系的，如居中、坐标、偏移量等。一般组件具有的位置属性及说明见表5-2。

表5-2　组件的位置属性及说明

名　称	说　明	取值举例
align	组件内容的对齐方式，只有当设置的宽、高超过元素内容大小时才有效，值为Alignment类型	Text('您好') .size({width:100,height:100 }) .align(Alignment.End) 另外，Alignment还有Top、TopStart、Start、Center、Buttom等枚举值
direction	设置元素水平方向的布局，可选值为Direction枚举类型	Direction有3个枚举值 Ltr表示元素从左到右布局 Rtl表示元素从右到左布局 Auto表示系统默认布局方向

续表

名　称	说　明	取值举例

position	位置，表示组件在父容器中的位置。默认以组件的左上角为基准	{
		x:Length,
		y:Length
		}

markAnchor	组件位置定位时的锚点，以元素顶部起点作为基准点进行偏移	{
		x:Length,
		y:Length
		}
		默认值为(0,0)

offset	相对布局完成位置坐标偏移量，设置该属性，不影响父容器布局，仅在绘制时进行位置调整	{
		x:Length,
		y:Length
		}
		默认值为(0,0)

3. 背景

组件可以设置背景，既可以设置颜色背景，也可以设置图片背景。设置背景的主要属性见表 5-3。

表 5-3　组件的背景属性及说明

名　称	说　明	取值举例
backgroundColor	背景颜色，值为 Color 类型	Text('您好') .size({width: 100, height: 100 }) .backgroundColor(0xCCCCCC)
backgroundImage	背景图片，可以设置一张图片的地址，可以是网络图片资源或本地图片资源，可以重复设置	如 Row(){}.backgroundImage('/comment/bg.jpg',ImageRepeat.X) 表示使用 bg.jpg 作为背景，沿 x 轴方向重复平铺
backgroundImageSize	背景图片大小，值为 JSON 格式数据或 ImageSize 类型	{ width?:Length, height?:Length }或 ImageSize
backgroundImagePosition	背景图在组件中的显示位置，值为 JSON 格式数据或 Alignment 类型	{ width?:Length, height?:Length }或 Alignment

4. 文本样式

文本样式主要用于设置组件内显示的文本的颜色、大小、字体等，关于文本设置的主要

属性见表 5-4。

表 5-4　文本样式属性及说明

名　　称	说　　明	取 值 举 例
fontColor	文本颜色,值为 Color 类型,可以直接用颜色值,也可以采用系统中的颜色	Text('您好') . size({width: 100, height: 100 }) . fontColor(0xFF0000)
fontSize	字体大小,值为 Length 类型,当为数值时默认单位为 fp	Text('您好') . size({width: 100, height: 100 }) . fontSize(30)
fontStyle	字体样式,值为 FontStyle 类型	Text('您好') . fontStyle(FontStyle. Normal)
fontWeight	字体粗细,值为 number 或 FontWeight 枚举类型,数值可以取 100～900 中的整百数值,默认为 400,值越大字越粗,FontWeight 提供了枚举类型	Text(this. message) . fontSize(50) . fontWeight(FontWeight. Bold)
fontFamily	字体,值为字符串类型。可以设置一种字体,也可以设置多种候选字体,以','分隔,按顺序选择显示的字体	Text(this. message) . fontSize(50) . fontFamily('Arial,sans-serif')

5. 其他属性

关于组件的属性还有很多,不过所有的属性都可以通过链式调用的方式设置,由于属性众多,且每个属性一般可以设定多种值,这里没有必要进行一一介绍。下面分类说明一下组件属性的主要用途,以便读者能够快速地了解组件属性,并能在使用时可以有的放矢地进行查询。关于组件属性的分类说明见表 5-5。

表 5-5　组件属性的分类说明

名　　称	主要用途说明
显示样式方面	组件的大小、位置、背景、透明度、边框、颜色渐变等
布局约束方面	宽高比、显示优先级、Flex 约束、栅格间距等
显示控制方面	显示隐藏控制、禁用控制、增加浮层、Z 序控制(层控制)等
图形图像处理方面	图形变换(旋转、平移、缩放、矩阵变换),图像效果(模糊、阴影、灰度、高光、饱和度、对比度、反转、颜色叠加、色相旋转、裁剪、遮盖等)

5.2.2　自定义属性

尽管系统中的组件属性非常丰富,但是在实际开发中难免还会定义一些特有的属性。开发者自定义组件属性可以扩展组件的属性,也可以在自定义的组件中定义属性。扩展已有组件的属性可以通过装饰器@Extend 实现,这一点在前面已经介绍过,下面介绍自定义组件属性。

自定义组件是通过@Component 修饰的,组件内可以自定义属性,即定义组件的成员

变量，自定义属性可以由多种不同的装饰器修饰，自定义组件及属性的一般形式如下：

```
@Component                                    //组件装饰器
struct MyComponent {                          //组件名
    count:number = 0                          //无装饰器的常规成员变量,初始化为 0
    @State mystate: 类型 = 初始值
    @Prop myprop: 类型
    @StorageProp mystorageprop: 类型 = 初始值
    @Link mylink: 类型
    @StorageLink mystoragelink: 类型  = 初始值
    @Provide myprovide: 类型 = 初始值
    @Consume myconsume: 类型
    @ObjectLink myobjectlink: 类型
    build() {                                 //生成器函数
        //省略了构造内部组件
    }
}
```

组件中的成员变量可以通过两种方式初始化，一种是在定义组件时直接进行本地初始化，另外一种方式是在实例化组件时通过传递构造参数进行初始化。

本地初始化直接在定义组件内实现，如上述定义的组件中的 count 成员变量（也称为属性）本地初始化，代码如下：

```
count:number = 0                              //本地初始化
```

本地初始化的成员在组件进行实例化时会自动初始化实例成员，实例化组件的代码如下：

```
MyComponent()                                 //实例化组件,其成员变量 count 的值为 0
```

通过传递构造参数进行初始化是在构建组件时为其传递构造参数，参数一般以 JSON 对象的方式传递，JSON 对象中的键值要求和属性名称相同，示例代码如下：

```
MyComponent( { count : 值 } )                 //构建组件时传递 JSON 对象格式数据
```

无论是自定义组件还是系统内置组件，相当一部分属性是可以通过构造参数进行初始化的。下面是一个完整的使用自定义组件并初始化的示例，代码如下：

```
//ch05/MyComponent 项目中 Index.ets 文件
@Entry                                                  //入口组件装饰器
@Component                                              //组件装饰器
struct Index {                                          //定义组件
  build() {                                             //生成器函数
    Column() {
      MyComponent()                                     //自定义无参数实例化
      MyComponent({ count: 100 })                       //自定义组件带参数实例化
    }.size({ width: '100%', height: '100%' })           //系统组件带参数构建
```

```
  }
}

@Component                                          //组件装饰器
struct MyComponent {                                //定义组件
  count: number = 0                                 //无装饰器的常规成员变量,初始化为 0
  build() {                                         //生成器函数
    if (this.count == 0) {
      Text('无可显示内容').fontSize(26)
    } else {
      Text('当前数量为' + this.count).fontSize(26)
    }
  }
}
```

图 5-2　自定义组件

以上代码实例化了两个自定义组件 MyComponent 对象,一个没有参数,另一个带有构造参数,它们在初始化 count 属性时有所不同,运行效果如图 5-2 所示。

对于没有装饰器装饰的组件属性,既可以本地初始化,也可以通过构造参数初始化,但是对于由装饰器修饰的组件,在进行初始化时有一定的限制,有的属性只能进行本地初始化,有的只能通过构造参数初始化。具体的初始化限制见表 5-6。

表 5-6　组件属性初始化限制说明

修饰属性的装饰器	本地初始化限制说明	构造参数初始化限制说明
@State	必须进行本地初始化	可选
@Prop	禁止进行本地初始化	必须进行构造参数初始化
@StorageProp	必须进行本地初始化	禁止进行构造参数初始化
@Link	禁止进行本地初始化	必须进行构造参数初始化
@StorageLink	必须进行本地初始化	禁止进行构造参数初始化
@Provide	必须进行本地初始化	可选
@Consume	禁止进行本地初始化	禁止进行构造参数初始化
@ObjectLink	禁止进行本地初始化	必须进行构造参数初始化
常规成员变量	可选	可选

由表 5-6 可知,有的装饰器修饰的属性只能通过本地初始化的方式初始化,如@StorageProp、@StorageLink。有的装饰器修饰的属性只能通过构造参数初始化,如@Prop、@Link、@ObjectLink。由@State、@Provide 修饰的属性必须进行本地初始化,同时也可以通过构造参数初始化,通过构造参数初始化会覆盖本地初始化的值。由@Consume 修饰的属性禁止初始化,常规成员变量初始化比较自由。

除了组件自身属性在初始化时有一定限制外,在组件存在父子关系时,组件的属性赋值也存在一定的限制,这里所讲的父子关系是一种包含关系,如果一个组件内包含了另外一个

组件，则前者为父组件，后者为子组件，示例代码如下：

```
//ch05/ParentChild项目中 Index.ets 文件
@Entry
@Component
struct Parent {                              //父组件定义
  b: boolean = true

  build() {
    Column() {                               //列容器
      Child()                                //子组件实例
      Child({ childState: this.b })          //子组件实例,传递参数
    }.width("100%")
  }
}

@Component
struct Child {                               //子组件定义
  @State childState: boolean = false         //初始化

  build() {
    Row() {
      Text('子组件状态' + this.childState)
        .fontSize(30)
    }
  }
}
```

以上代码定了父组件 Parent，其中实例化了两个子组件 Child 对象，在初始化第 2 个子组件时传递了构造参数，并采用了父组件成员变量。以上代码的运行效果如图 5-3 所示。

图 5-3 父子组传递参数

由于组件属性的初始化时机和不同装饰器装饰的属性更新机制不同，所以父组件属性数据在初始化子组件时有一定的限制。在使用父组件属性作为构造参数初始化子组件时，具体的限制说明见表 5-7。

表 5-7 父组件属性初始化子组件的限制说明

名 称	主要用途说明
@State	不允许使用父组件中@State、@Link、@Prop 装饰的属性作为参数初始化 允许使用常规变量作为参数初始化
@Link	不允许使用父组件中@Prop、@StorageProp 装饰的属性和常规变量作为参数初始化 允许使用@State、@Link、@StorageLink 装饰的属性作为参数初始化

续表

名　　称	主要用途说明
@Prop	不允许使用父组件中@StorageLink、@StorageProp装饰的属性和常规变量作为参数初始化 允许使用@State、@Link、@Prop装饰的属性作为参数初始化
常规成员变量	允许使用任意装饰的属性和常规变量作为参数初始化

由表5-7可知,子组件中由@State修饰的属性不允许由父组件的@State装饰的属性进行初始化,这是因为父组件的状态变化一般不需要重新构建其中的所有子组件,因此有必要限制这种参数传递。

实际上,不同的装饰器规定了属性的更新方式,要进一步了解深层原因,需要进一步理解组件的状态。

5.3　组件事件

组件事件的设置和组件属性比较类似,不同的是用于设置属性的方法是以数据为参数的,而设置组件事件的方法是以函数为参数的。

5.3.1　组件事件配置方式

通过组件提供的事件方法可以配置组件支持的事件,并通过实现响应的代码达到事件处理的目的。配置组件事件有3种方式,下面以按钮(Button)的单击事件为例说明这3种方式。

1. 使用Lambda表达式配置组件的事件

使用Lambda表达式作为按钮单击事件的参数,便可以为按钮配置单击响应,示例代码如下:

```
//ch05/EventSet项目中 Index.ets文件部分代码
@Entry
@Component
struct ComponentA {
  @State count: number = 0

  build() {
    Column() {

      Button('单击1')
        .onClick(() => {                    //单击响应,Lambda表达式方式
          this.count++
        })

      Text(`count: ${this.count}`).fontSize(25)
    }.width('100%')
  }
}
```

通过 Button 组件的 onClick()方法为按钮设置单击响应，当单击按钮时，会调用 Lambda 表达式对应的代码，实现按钮单击响应。

2. 使用组件的成员函数配置组件的事件

该方式需要定义组件成员函数，然后以成员函数作为参数传递给组件事件方法，示例代码如下：

```
//ch05/EventSet 项目中 Index.ets 文件部分代码
@Entry
@Component
struct ComponentA {
  @State count: number = 0

  fun(): void {                              //定义成员函数
    console.log("do something in fun")
    this.count++                             //绑定后可以改变成员变量
  }

  build() {
    Column() {

      Button('单击 2')
        //成员函数作为参数，如果不绑定，则不能访问 this.count
        //.onClick( this.fun )
        .onClick(this.fun.bind(this))        //绑定

      Text(`count: ${this.count}`).fontSize(25)
    }.width('100%')
  }
}
```

以上代码通过在按钮的 onClick 方法参数中传入组件成员函数(this.fun)实现事件配置，当单击按钮时，会调用 fun 函数。

需要注意的是，这里的 fun 函数不能操作组件中的成员，因为，在 fun 函数中不能确保函数体中 this 引用了包含的组件。如果想操作组件的成员变量，则需要调用绑定(bind(this))操作。

3. 使用匿名函数表达式配置组件的事件

使用匿名函数不需要为函数命名，函数的定义和使用是一体的，例如对于前面按钮单击事件，使用匿名函数响应单击事件的代码如下：

```
//ch05/EventSet 项目中 Index.ets 文件部分代码
Button('单击 3')
  .onClick(function(){                       //匿名函数
    this.count++
    console.log("do something")
  }.bind(this))
```

5.3.2 通用事件方法

前面用到了单击事件的设置(onClick()),组件有一些通用的事件设置方法,如单击事件、触摸事件、按键事件等,这些方法适用所有组件,包括普通组件和容器(布局)组件。下面简要介绍这些事件方法。

1. 单击事件

单击事件指组件被单击时触发的事件,该事件对应的方法声明如下:

```
onClick(callback: ( event?:ClickEvent ) => void )
```

单击事件方法参数是一个回调函数,该回调函数有一个可选的单击事件参数,类型名为ClickEvent,ClickEvent 对象的属性说明见表 5-8。

表 5-8 ClickEvent 对象说明属性

属性名称及类型	说 明
x:number	x 坐标,值为单击点相对于被单击组件左边沿的距离
y:number	y 坐标,值为单击点相对于被单击组件上边沿的距离
screenX:number	单击点相对于设备屏幕左边沿的 x 坐标
screenY:number	单击点相对于设备屏幕上边沿的 y 坐标
target:EventTarget	被单击的组件对象,该对象内含有组件的区域信息
timestamp	单击事件发送的时间戳

下面是使用单击事件参数的示例,代码如下:

```
//ch05/WithClickEvent 项目中 Index.ets 文件部分代码
Button('单击按钮')
  .onClick((event) => {                    //单击响应,Lambda 表达式方式
    console.log("screenX:" + event.screenX)
    console.log("screenY:" + event.screenY)
    console.log("source:" + event.source.toString())
    console.log("x:" + event.x)
    console.log("y:" + event.y)
    console.log("timestamp: " + event.timestamp.toString())
  })
```

除了单击事件,组件还有一些通用的事件设置方法,如触摸事件、按键事件等。这些方法适用所有组件,包括普通组件和容器(布局)组件,下面简要介绍这些事件方法。

2. 触摸事件

触摸事件指组件被触摸时触发的事件,当手指放在组件上、滑动或从组件上移开时,该事件会被触发。该事件对应的方法声明如下:

```
onTouch( callback : ( event?:TouchEvent ) => void)
```

触摸事件方法参数是一个回调函数,该回调函数有一个可选的触摸事件参数,类型名为

TouchEvent，TouchEvent 对象的属性说明见表 5-9。

表 5-9 TouchEvent 对象说明属性

属性名称及类型	说 明
type：TouchType	触摸事件的类型。TouchType 包含 4 种枚举类型，即 Down：手指按下；Up：手指抬起；Move：手指按压在屏幕上移动；Cancel：触摸事件取消
touches：Array	触摸的全部手指信息
changedTouches：Array	当前发生变化的手指信息
target：EventTarget	被触摸的组件对象，该对象内含有组件的区域信息
timestamp	触摸事件发送的时间戳

3. 按键事件

按键事件指组件与键盘、遥控器等按键设备交互时触发的事件。绑定该方法的组件获焦后，按键动作触发该方法，该事件对应的方法声明如下：

```
onKeyEvent( event : ( event?:KeyEvent ) => void )
```

按键事件方法参数是一个回调函数，该回调函数有一个可选的按键事件参数，类型名为 KeyEvent，KeyEvent 对象的属性及接口说明见表 5-10。

表 5-10 KeyEvent 对象说明属性及接口说明

属性名称及接口	说 明
type：KeyType	按键的类型，KeyType 有两个枚举量，即 Down：按键按下；Up：按键松开
keyCode：number	按键的键码
keyText：string	按键的键值
keySource：KeySource	触发当前按键的输入设备类型
deviceId：number	触发当前按键的输入设备 ID
timestamp：number	按键发生的时间戳
stopPropagation()：void	阻塞事件冒泡传递

4. 焦点事件

焦点事件指组件在获得或失去焦点时响应的事件。该类事件有两个事件配置方法，分别对应获得焦点事件和失去焦点事件，对应的方法声明如下：

```
onFocus( callback: () => void )          //获得焦点，当前组件获取焦点时触发回调
onBlur( callback: () => void )           //失去焦点，当前组件失去焦点时触发回调
```

焦点事件从 API Version 8 开始支持，目前支撑的组件有 Text、Button、Image、List 和 Grid。

5. 其他事件

除了前面介绍的常用事件外，组件还支持其他的组件事件，这些组件事件在使用上尽管都有差别，但是它们都和常用事件类似，都是通过设置事件处理函数进行事件处理，这里不再详细说明。这里仅给出一些其他事件配置方法的简要说明，以便读者可以快速地认识这

些事件。一些其他事件的配置函数及说明见表 5-11。

表 5-11　其他事件配置方法及说明

事 件 方 法	简 要 说 明
onDragStart()	第 1 次拖曳组件时,触发回调
onDragEnter()	拖曳进入组件范围内时,触发回调
onDragMove()	拖曳在组件范围内移动时,触发回调
onDragLeave()	拖曳离开组件范围内时,触发回调
onDrop()	当在本组件范围内停止拖曳行为时,触发回调
onHover()	鼠标进入或退出组件时触发该回调
onMouse()	当前组件被鼠标按键单击时或者鼠标在组件上移动时,触发该回调
onAreaChange()	组件区域变化时触发该回调,如变大或变小
onAppear()	组件挂载显示时触发此回调
onDisappear()	组件卸载消失时触发此回调

注意,表 5-11 中省略了这些事件方法的回调函数参数。

5.4　状态管理

5.4.1　状态模型

在基于 ArkTS 的声明式 UI 编程范式中,UI 是应用程序状态的函数,当应用程序状态发生改变时,系统会自动更新界面。

对于没有数据存储的 UI 界面来讲,界面中的组件可以通过状态装饰器装饰成员变量来关联组件。关联的组件之间,在一个组件的状态数据发生变化时,系统会自动更新相关组件。组件对应的状态装饰器及相互更新状态的关系如图 5-4 所示。

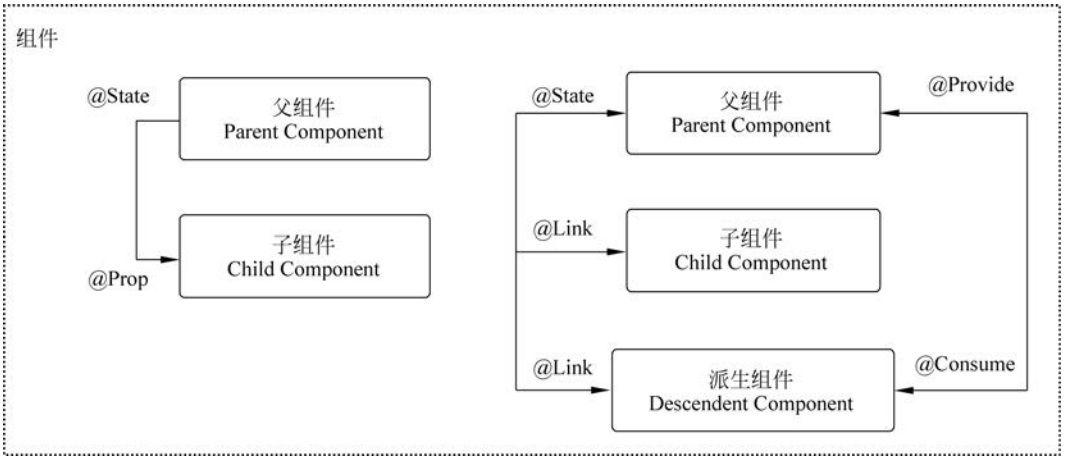


图 5-4　组件间的数据关联

如图 5-4 所示，当父组件中由@State 修饰的成员变量发生变化时，系统框架会自动通知子组件中由@Prop 修饰的成员变量，并更新 UI，这种传递是单向的，而对于子组件中由@Link 修饰的成员变量，父子组件之间状态变化传递是双向的。

对于具有数据存储的应用来讲，应用中不仅包含组件界面，而且包含其他元素，如应用存储（AppStorage）、能力（Ability）、持久存储（Persistent）等。组件和应用存储之间可以建立双向或单向的数据更新机制，具体如图 5-5 所示。

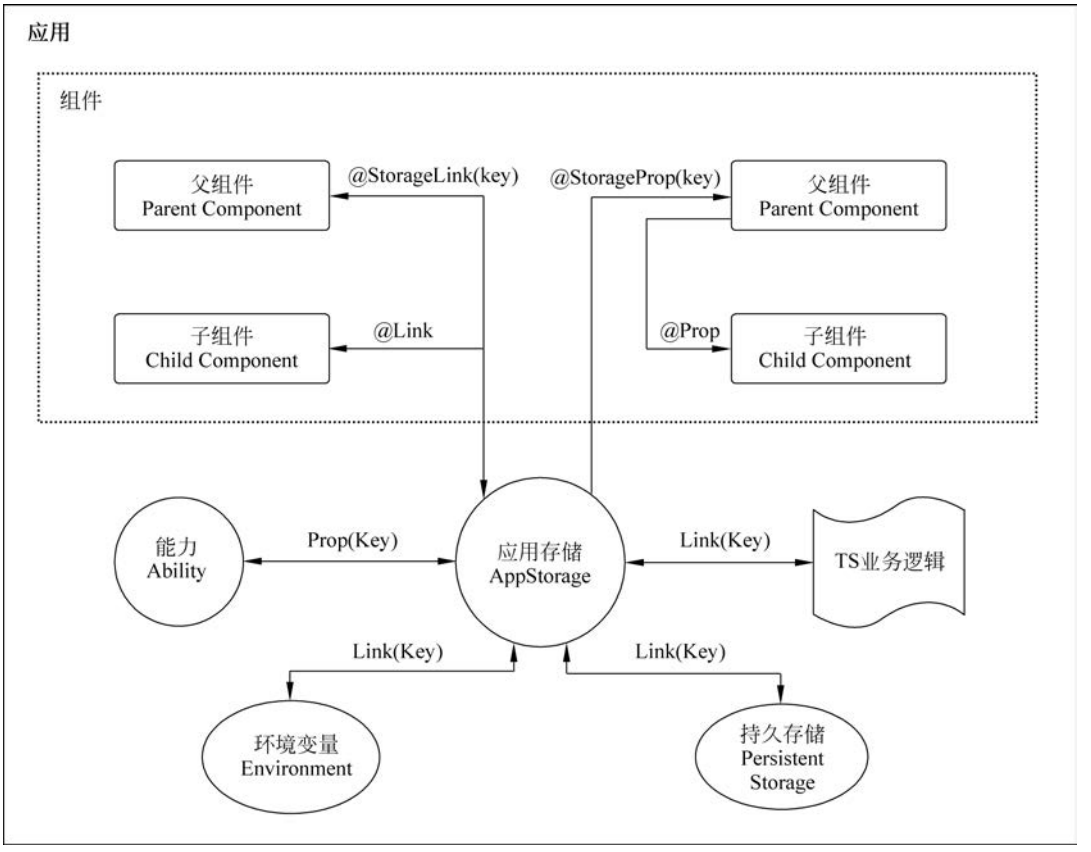


图 5-5 应用中数据关联

应用存储是整个 UI 应用程序状态的数据中心，UI 框架会针对应用程序创建单例 AppStorage 对象，并提供相应的装饰器和接口供应用程序使用。AppStorage 提供了用于业务逻辑实现的 API，用于添加、读取、修改和删除应用程序的状态属性，修改后的状态数据会同步到 UI 组件并更新。

5.4.2 组件状态

在组件状态管理方面，系统框架主要提供了 3 个装饰器，分别是@State、@Prop、@Link。

1. 装饰器@State

组件中,由@State装饰的变量是组件内部状态数据,当内部状态数据变化时,会调用所在组件的生成器方法进行界面刷新,因此组件会随着内部状态数据的变化而实时更新。

关于@State装饰器需要注意以下几点:

(1) 组件只能监听到第1层状态数据的改变,内层数据的改变无法触发build生命周期。

(2) @State可以装饰class、number、boolean、string类型成员,也可以装饰它们构成的数组,如Array<class>、Array<string>、Array<boolean>、Array<number>,但是,不允许装饰object和any类型成员。

(3) 在组件具有多个实例的情况下,各个实例的内部状态数据是独立的。

(4) 使用@State装饰的变量必须进行本地初始化,否则可能导致框架行为未定义。

(5) 对于自定义组件,内部状态变量名可以通过构造参数进行初始化。

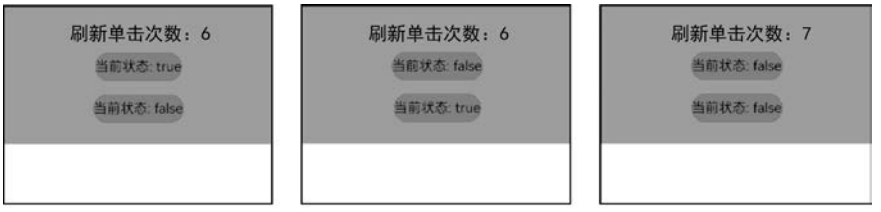
下面是一个关于@State的实例,代码如下:

```
//ch05/StateDemo项目中Index.ets文件
@Entry
@Component
struct Index {
    @State count: number = 0                          //由@State装饰
    build() {
        Column() {
            Text(`刷新单击次数: ${this.count}`)
                .fontSize(30)
            MyButton()                                //无参数实例化
            MyButton({ isclicked: false })            //带参数初始化
        }.width("100%").backgroundColor(0xff0000).padding(20)
        .onClick(() => {                              //背景区域,Column单击响应
            this.count++                              //更新内部状态数据,会重新build
        }
        )
    }
}

@Component
struct MyButton {
    @State isclicked: boolean = true                  //由@State装饰
    build() {
        Column() {
            Button() {
                Text(`当前状态: ${this.isclicked}`)
                    .fontSize(20)
            }.margin(10)
            .onClick(() => {                          //按钮,Button单击响应
                this.isclicked = !this.isclicked
```

```
            }
            )
        }
    }
}
```

以上代码运行的效果如图 5-6 所示，当每次单击背景区域时都会更新组件 Index 中的 count 内部状态变量，进而重新构建整个界面。在界面中有两个 MyButton 的实例，一个构建没有参数，另一个带有构造参数。组件 MyButton 内也有内部状态变量，单击时会重刷新 MyButton 实例状态，两个 MyButton 实例相互独立。由于第 1 个 MyButton 实例构建没有参数传递，在单击背景区刷新时，第 1 个 MyButton 的状态不变化。



(a) Column单击6次　　(b) 按钮响应　　(c) Column单击7次

图 5-6　内部状态数据实例

2. 装饰器@Prop

装饰器@Prop 与@State 有相同的语义，但初始化方式不同。由@Prop 装饰的变量在使用其父组件提供的由@State 装饰的变量进行初始化时，便建立了父子组件之间的状态绑定。绑定后，父组件中@State 变量的变化会通知子组件中的@Prop 变量，但在子组件内部修改@Prop 变量时不会通知父组件，即@Prop 属于单向数据绑定。

关于@Prop 装饰器需要注意以下几点：

(1) 装饰器@Prop 只能装饰简单类型，包括 number、string、boolean 等简单类型。

(2) 在组件中可以定义多个由@Prop 装饰的属性。

(3) 在自定义组件时，不能在组件内初始化@Prop 变量。

(4) 在创建组件实例时，必须通过构造参数初始化所有@Prop 变量。

下面是一个关于@Prop 的实例，代码如下：

```
//ch05/PropDemo 项目中 Index.ets 文件
@Entry
@Component
struct ParentComponent {
    @State goldCount: number = 10                          //由@State 装饰的变量
    build() {
        Column() {
            Row() {
                Text(`初始化金币数量: ${this.goldCount} `)
```

```
                    .margin(10).fontSize(15)

                Button() {
                    Text('- 1 ')
                }.margin(5).size({ height: 30, width: 30 })
                .backgroundColor(0xccccff)
                .onClick(() => {                                    //单击响应
                    this.goldCount -= 1
                })

                Button() {
                    Text(' + 1 ')
                }.margin(5).size({ height: 30, width: 30 })
                .backgroundColor(0xccccff)
                .onClick(() => {                                    //单击响应
                    this.goldCount += 1
                })
            }.margin(10)

            //下面创建 3 个子组件,必须初始化@Prop 变量 count
            //普通变量可以不通过参数初始化
            ChildComponent({ count: this.goldCount })               //有绑定
            ChildComponent({ count: this.goldCount, cost: 2 })      //有绑定
            ChildComponent({ count: 100, cost: 5 })                 //没有绑定
        }.backgroundColor(0xeeeeee)
        .width('100 % ')
    }
}

@Component
struct ChildComponent {
    @Prop count: number                                             //由@Prop 装饰的变量
    private cost: number = 1                                        //普通成员变量

    build() {
        Row() {
            if (this.count > 0) {
                Text(` 您剩余金币: ${this.count} 个  `)
            } else {
                Text('已用完!')
            }

            Button() {
                Text(`单击消费 ${this.cost} 金币`)
            }.padding(10)
            .onClick(() => {                                        //单击响应
                this.count -= this.cost
            })
        }.backgroundColor(0xbbbbbb)
```

```
            .margin(5).padding(10)
        }
    }
```

以上代码运行的效果如图 5-7 所示，当每次单击父组件(ParentComponent)中的＋1 或－1 按钮时，程序会初始化金币数量(goldCount 值)，由于前两个子组件和父组件进行了绑定，所以子组件也会刷新数量(count 值)，而第 3 个子组件不刷新。在子组件中单击消费金币时，各个子组件独立减少自己的金币数量，相互之间互不干扰，也不会更新父组件。

(a) 初始　　(b) 增加金币　　(c) 消费金币

图 5-7　关于@Prop 的数据状态实例

3. 装饰器@Link

由@Link 装饰的变量可以和父组件的@State 变量建立双向数据绑定连接，这一点和@prop 装饰的单向绑定不同，由@Link 建立的双向绑定可以实现父组件和子组件的双向联动。

关于@Link 装饰器需要注意以下几点：

(1) 装饰器@Link 可以装饰的变量与@State 的类型相同，即 class、number、string、boolean 及它们的数组。

(2) 由@Link 装饰的变量不能在组件内进行本地初始化。

(3) 自定义组件在实例化时，必须初始化所有@Link 变量。

(4) 由@Link 装饰的变量可以使用父组件的@State 变量或@Link 变量的引用来进行初始化。

(5) 初始化所用的@State 变量，可以通过'$'操作符创建引用。

(6) 绑定后，子组件对@Link 变量的更改将同步修改父组件的@State 变量，反之亦然。

下面是一个关于@Link 的实例，代码如下：

```
//ch05/LinkDemo 项目中 Index.ets 文件
@Entry
@Component
struct ParentComponent {
    @State message: string = ''                    //由@State 装饰的变量
    @State curValue: string = ''                   //由@State 装饰的变量
```

```
    build() {
        Column() {
            Text(` 欢迎您 `).fontSize(25)
            Text(` ${this.message} `)                    //提示信息
                .margin(5).fontSize(18).fontColor(0xff0000)
            //下面创建子组件,必须初始化@Link 变量,普通变量通过参数初始化可选
            //这里 msg 和 message 绑定,value 和 curValue 绑定
            ChildComponent({ msg: $message, value: $curValue, //绑定
                hint: '输入用户名'})
            Text(`当前值: ${this.curValue} `)            //提示信息
                .margin(5).fontSize(18).fontColor(0x00ff00)
            Button() {
                Text(` 重 置 `).fontSize(22)
            }.onClick(() => {                            //响应单击
                this.message = '您单击了重置'
                this.curValue = ''                       //这里会更新绑定的子组件变量 value
            })
        }.backgroundColor(0xeeeeee)
        .width('100%').padding(30)
    }
}

@Component
struct ChildComponent {
    @Link msg: string                                    //由@Link 装饰的变量
    @Link value: string                                  //由@Link 装饰的变量
    minLength: number = 8                                //普通变量
    hint: string = ''                                    //普通变量

    build() {
        Row() {
            //输入框组件
            TextInput({ text: this.value, placeholder: this.hint })
                .fontSize(30)
                .onChange((v: string) => {               //当输入内容时响应
                    this.value = v
                    if (v.length < this.minLength) {
                        //msg 更新,会更新父组件的 message 变量
                        this.msg = '当前输入的长度为 ' + v.length
                         + '小于' + this.minLength;
                    } else {
                        this.msg = '符合长度要求'
                    }
                })
        }.backgroundColor(0xbbbbbb)
        .margin(5).padding(10)
    }
}
```

以上代码运行的效果如图 5-8 所示，当在子组件的输入框中输入数据时，由于子组件中 msg、value 分别和父组件中的 message、curValue 进行了数据绑定，因此父组件中会实时更新提示。当在父组件中单击“重置”按钮时，由于父组件中 curValue 被赋值成了空字符串，这样子组件中 value 值也被更新成空字符串，进而会把子组件的输入框内容置空。通过@Link 装饰可以实现父子组件之间数据的双向绑定，动态更新。

(a) 初始　(b) 输入不合规数据　(c) 输入合规数据　(d) 重置

图 5-8　关于@Link 的数据状态实例

5.4.3　应用程序状态

通过组件的状态数据可以实现组件之间的数据绑定，实现父组件和子组件之间的互动，比较适合用于单个页面内多个组件的情景。对于多个页面之间的数据共享采用应用存储更为合适。

AppStorage 是应用程序中的单例对象，由 UI 框架在应用程序启动时创建，在应用程序退出时销毁，为应用程序范围内的可变状态属性提供中央存储。AppStorage 可以保存属性及属性值，属性值可以通过唯一的键值进行访问。

UI 组件可以通过装饰器与 AppStorage 进行同步，应用业务逻辑的实现也可以通过接口访问 AppStorage 存储的数据。

组件成员和 AppStorage 进行同步的装饰器有@StorageLink 和@StorageProp。

组件通过@StorageLink(key)装饰的状态变量与 AppStorage 建立双向数据绑定。当创建包含@StorageLink 的状态变量的组件时，该状态变量的值将使用 AppStorage 中对应的值进行初始化。在 UI 组件中对@StorageLink 的状态变量所做的更改将同步到 AppStorage 中，并从 AppStorage 同步到任何其他绑定实例中，如 PersistentStorage 或其他绑定的 UI 组件。

组件通过@StorageProp(key)装饰的状态变量与 AppStorage 建立单向数据绑定。当创建包含@StoageProp 的状态变量的组件时，该状态变量的值将使用 AppStorage 中的值进行初始化。AppStorage 中的属性值的更改会导致绑定的 UI 组件进行状态更新。

下面是一个使用 AppStorage 的实例，该实例中包含 index.ets 和 next.ets 两个 ETS 文件，index.ets 文件的代码如下：

```
//ch05/AppStorageDemo 项目中 index.ets 文件
import router from '@ohos.router';
```

```
@Entry
@Component
struct Index {
  //注意下面两个装饰符都是@StorageLink
  @StorageLink('count1') indexCount1: number = 0          //双向绑定
  @StorageLink('count2') indexCount2: number = 0          //双向绑定
  private label: string = '单击'

  build() {
    Column({ space: 20 }) {
      Text('当前页面是 index.ets').fontSize(30)
      Button(` ${this.label} `)
        .onClick(() => {                                  //响应单击
          //下面通过 API 修改应用存储变量
          var temp = AppStorage.Get<number>('count1')
          AppStorage.Set('count1', temp + 1)
          //下面通过组件成员变量修改
          this.indexCount2++
        })

      //使用组件的成员变量
      Text(`indexCount1: ${this.indexCount1}`).fontSize(25)
      Text(`indexCount2: ${this.indexCount2}`).fontSize(25)

      //使用存储变量
      Text(`count1: ${AppStorage.Get<number>('count1').toString()}`)
        .fontSize(25)
      Text(`count2: ${AppStorage.Get<number>('count2').toString()}`)
        .fontSize(25)

      Button('跳转到 next 页面')
        .onClick(() => {                                  //单击响应
          router.push({ url: 'pages/next' })              //跳转到 next.ets
        })
    }.width('100%').padding(20)
    .backgroundColor(0xccFFcc)
  }
}
```

next.ets 文件的代码如下：

```
//ch05/AppStorageDemo 项目中 next.ets 文件
import router from '@ohos.router';
@Entry
@Component
struct Next {
  //注意下面两个装饰符都是@StorageProp
  @StorageProp('count1') nextCount1: number = 0          //单向绑定
```

```
  @StorageProp('count2') nextCount2: number = 0          //单向绑定

  private label: string = '单击'

  build() {
    Column({ space: 20 }) {
      Text(`当前页面是 next.ets`).fontSize(30)
      Button(` ${this.label} `)
        .onClick(() => {
          //可以通过 API 修改应用存储变量
          var t = AppStorage.Get<number>('count1')
          AppStorage.Set<number>('count1', t + 1)
          //下面的修改会提示错误,TypeError: no setter for property
          this.nextCount2++})

      //使用组件的成员变量
      Text(`nextCount1: ${this.nextCount1}`).fontSize(25)
      Text(`nextCount2: ${this.nextCount2}`).fontSize(25)

      //通过 API 使用存储变量
      Text(`count1: ${AppStorage.Get<number>('count1').toString()} `)
        .fontSize(25)
      Text(`count2: ${AppStorage.Get<number>('count2').toString()} `)
        .fontSize(25)

      Button('返回')
        .onClick(() => {
          router.back()                                  //返回上一个页面
        })
    }.width('100%').padding(20)
    .backgroundColor(0xFFcccc)
  }
}
```

以上代码的运行效果如图 5-9 所示。当在 index 页面(图 5-9(a))上单击按钮时,4 个变量都会改变,因为在 index 中采用的是@StorageLink 装饰器,indexCount1 和 count1,

(a) index页上单击　　(b) next页　　(c) next页上单击

图 5-9　AppStorage 和组件数据绑定

indexCount2 和 count2 建立的都是双向绑定，因此无论通过 AppStorage 提供的 API 修改 count1，还是直接修改 indexCount2 都会更新绑定的另一个变量。

当跳转到 next 页面(图 5-9(b))时，4 个变量也会同步显示，但是在 next 页面上单击按钮时(图 5-9(c))，只有 indexCount1 和 count1 会同步更新，这里采用的是@StorageLink 装饰器，建立的是单向绑定。

AppStorage 中存储的数据采用的是键-值对的形式，AppStorage 提供了操作存储数据的接口 API，常用的 API 的说明见表 5-12。

表 5-12　AppStorage 提供的部分接口

方 法 声 明	说　　明
Set(key:string,newValue:T):void	将已保存的 key 值设置为新的 newValue 值
SetOrCreate(key: string, newValue: T): boolean	创建一个键为 key，值为 newValue 的属性，如果已存在，并且可以改写，则替换为新值 newValue，返回值为 true，否则返回值为 false。属性值不支持 null 和 undefined
Get(key:string):T	获取 key 对应的值，如果不存在，则返回 undefined
Has(propName:string):boolean	判断对应键值的属性是否存在
Link(key:string):@Link	如果存在具有给定键的数据，则返回此属性的双向数据绑定，该双向绑定意味着变量或者组件对数据的更改将同步到 AppStorage，通过 AppStorage 对数据的修改将同步到变量或者组件。如果具有此键的属性不存在或属性为只读，则返回 undefined
SetAndLink(key:string,defaultValue:T):@Link	与 Link 接口类似，不同的是，在 key 不存在时创建并赋默认值
Prop(key:string):@Link	与 Link 不同的是，该接口建立的是单向绑定。Prop 方法对应的属性值类型只能是简单类型
SetAndProp(propName: string, defaultValue: S):@Prop	与 Prop 接口类似，不同的是，在 key 不存在时创建并赋默认值
Keys():array<string>	返回包含所有键的字符串数组
Delete(key:string):boolean	删除 key 对应的键-值对
IsMutable(key:string):boolean	判断 key 属性是否存在且可以改变
Clear():boolean	删除所有的属性，如果当前有状态变量依旧引用属性，则不能清除，返回值为 false

AppStorage 的选择状态属性可以与不同的数据源或数据接收器同步，这些数据源和接收器可以是设备上的本地或远程，并具有不同的功能，如数据持久性。这样的数据源和接收器可以独立于 UI 在业务逻辑中实现。

另外，系统框架提供的环境对象(Environment)也是在应用程序启动时创建的单例对象，它可以为 AppStorage 提供一系列应用程序需要的环境状态属性。通过 Environment 提供 API 可以和 AppStorage 建立联系，代码如下：

```
Environment.EnvProp("accessibilityEnabled", "default")
```

以上代码可以使无障碍屏幕朗读环境变量(accessibilityEnabled)和 AppStorage 建立联系,通过 AppStorage 可以访问该变量,代码如下:

```
var read = AppStorage.Get("accessibilityEnabled")        //获得环境变量值
```

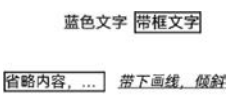
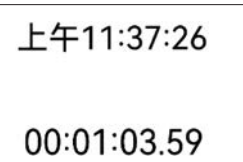
总之,AppStorage 可以为多个页面中的组件提供底层的统一数据支持,实现在应用运行期间组件之间的数据共享,同时数据可以实现单向或双向的绑定。在应用开发中,AppStorage 可以和环境变量、持久化存储、TS 业务逻辑等建立数据绑定关系,为上层组件提供数据支撑。

5.5 系统内置组件简介

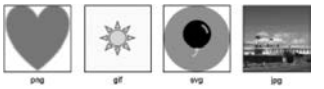
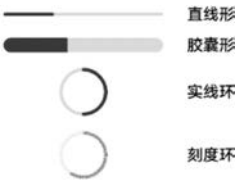
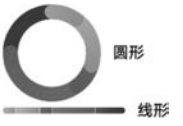
在基于 TS 扩展的声明式开发范式的方舟开发框架中,系统为 HarmonyOS 应用开发提供了很多内置组件,通过这些组件可以使开发者更轻松地构建出丰富的界面。

内置组件在使用的方式上基本是相同的,开发者可以类比前面介绍的组件进行学习并掌握。由于组件众多,组件的属性和方法也非常丰富,限于篇幅原因,这里不再一一介绍它们的具体使用细节。为了能够使读者快速地了解及认识这些组件,这里简要地给出这些组件的名称、主要用途和一些实例效果供读者参考,具体见表 5-13。

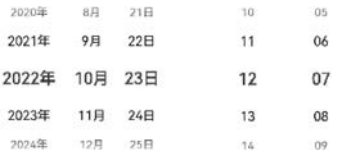
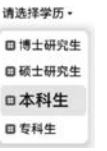
表 5-13 系统内置组件简要说明

组件名称	主要用途	实例效果
Text	文本,用于显示文字,可以修饰文字,也可以在其他组件内部使用	
TextInput TextArea	TextInput 单行文本输入框,可以输入一行文本,如用于输入账号、密码等。 TextArea 为多行输入框	
TextClock TextTimer	TextClock 通过文本显示当前系统时间,支持不同时区的时间显示,精度到秒。 TextTimer 为文本计时器组件,支持自定义时间格式	
TextPicker	文本滑动选择器,可以关联一个字符串数组,通过上下滑动选择需要的文本选项	

续表

组件名称	主要用途	实例效果
Image	图片，用于显示图片，图片源可以是资源图片或网络图片等，可以在其他组件的内部使用。另外，还有 ImageAnimator 图片帧动画组件，通过提供多张图片，实现逐帧播放图片，可以配置播放的图片列表，每张图片可以配置时长	 图片还可以渲染、缩放、裁剪、重复平铺等
Button	按钮，一般用于单击，支持多种样式按钮，结合 Image 可以做出各种个性化按钮	
Checkbox	复选框，通常用于某选项的复选	
CheckboxGroup	复选框群组，用于控制复选框全选或全不选	
Radio	单选框，用于单选，同组的单选框可以互斥	
Progress	进度条，用于显示内容加载或操作处理进度可视化展示	
Slider	滑动条组件，用来通过滑动调节设置值，如设置进度、音量、亮度等	
Rating	评价条，一般用于服务、商品等评价	
DataPanel	数据面板组件，用于对多个数据占比情况进行展示，可以采用圆形或线性图形的方式展示	
QRCode	二维码，可以根据数据生成并显示二维码	
Search	搜索框组件，提供用于搜索内容的输入区域	
Navigation	导航组件，一般用作页面的根容器，通过属性设置来展示页面的标题、工具栏、菜单	

续表

组件名称	主要用途	实例效果
Stepper StepperItem	Stepper为步骤导航器，每步对应一个StepperItem步骤导航器元素	
DatePicker TimePicker	DatePicker是日期选择器，可以通过上下滑动选择年、月、日 TimePicker是时间选择器，可以通过上下滑动选择时间	
Select	下拉选择菜单，可以通过下拉在多个选项之间选择一个选项	
Badge	新事件标记，在组件上提供事件信息展示	
Counter	计数器，用于购物数量增加或者减少计数等	

除了表5-13中列出的组件外，系统框架还提供了一些特殊功能的组件，如空白填充组件(Blank)、分隔器组件(Divider)等，还有一些功能高级的组件，如Web组件(Web)、用于EGL/OpenGLES和媒体数据写入的组件(XComponent)等。开发者也可根据已有的组件，组装出更多更丰富的自定义组件。

小结

本章介绍了ArkUI开发框架中的组件。组件是界面的基本组成单元，组件也是应用中的一个对象。组件可以设置属性和事件。通用属性是所有组件都具有的属性，如尺寸、位置、背景、样式等，组件还可以自定义属性以满足个性化需求。组件事件可以为组件设置动态特性，通过回调函数响应组件的行为，使组件具有可以操作的功能。

在HarmonyOS基于ArkTS的声明式UI编程范式中，UI是应用程序状态的函数。状态管理为应用中的组件更新提供了动态机制，组件是通过装饰器实现状态管理的。父子组件之间、组件和应用存储之间可以通过状态管理实现数据绑定，使应用界面可以动态刷新。系统提供了丰富的组件，开发者可以根据需求选择合适的组件构建应用界面。