第3章

页面组件

CHAPTER 3

组件(Component)是界面搭建与显示的最小单位，HarmonyOS ArkUI 声明式开发范式为开发者提供了丰富多样的 UI 组件，我们可以使用这些组件轻松地编写出更加丰富、漂亮的界面。组件主要有：基础组件、容器组件、媒体组件、绘制组件以及画布组件。接下来我们将对这些组件进行详细的讲解。

3.1　基础组件

基础组件是视图层的基本组成单元，包括 Text 组件、Image 组件、TextInput 组件、Button 组件、Divider 组件、LoadingProgress 组件、Radio 组件、Toggle 组件、Progress 组件、Slider 组件等。

3.1.1　Text 组件

Text 组件用于在界面上展示一段文本信息。它可以包含子组件 Span。像 Text、Span、Button、TextInput 等这种包含文本元素的组件，都可以使用 fontColor、fontSize、fontStyle、fontWeight、fontFamily 这些文本样式来设置文本的颜色、大小、样式、粗细以及字体。

1. 接口

Text 组件的接口如下：

```
Text(content?:string | Resource)
```

我们可以看出，Text 组件的入口参数是 string 类型或者 Resource 类型。string 类型我们已经很清楚了，比如 Text('显示文字')中的'显示文字'就是 string 类型。还可以是 string 类型的变量，或者其他类型的变量转成 string 类型。

举例：

```
Text('显示字符串')              //string 类型的字符串作为入口参数显示“显示字符串”
let str:string = '字符串变量'   //定义一个 string 类型的变量 str
Text(str)                       //string 类型的变量做入口参数，显示“字符串变量”
let num:number = 100            //定义一个 number 类型的变量 num
Text(num.toString())            //将其他类型变量转成 string 类型作入口参数，显示“100”
```

Resource 类型，就需要在 DevEco 工程目录的 entry/str/main/resources/base/element 目录下的 string.json 文件中增加如下内容：

```
,
{
  "name": "Text_Content",
  "value": "资源文字内容"
}
```

将上述代码增加到如图 3-1 所示的位置。这样，Text_Content 就变成 Resource 资源了。

然后就可以通过“$r('app.type.name')”的形式引用资源了。其中 app 代表应用内 resources 目录中定义的资源，type 代表资源类型（或资源存放位置），可取 color、float、string、plural、media 其中之一，name 代表资源命名，由开发者定义资源时确定。在上述文件中，type 就取 string，name 就是 Text_Content。具体调用形式如下：

```
Text( $r("app.string.Text_Content"))      //显示“资源文字内容”
```

由于 Resource 资源管理本质上只是将内容存储到对应的 string、float 等类型文件中，

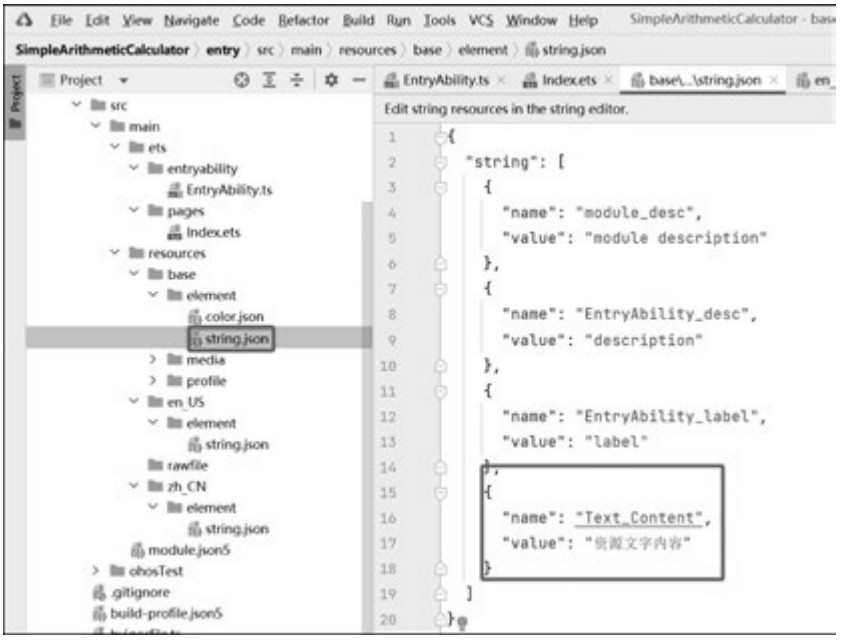


图 3-1　增加文字资源的位置

其效果与在代码中直接设置(硬编码)并无本质区别,因此,在后续内容中,我们将不再单独讲解 Resource 类型的具体使用。

2. 属性设置

1) fontColor 属性

fontColor 的参数类型可以是 Color 枚举类型、number 类型、string 类型,还可以是 Resource 类型。格式如下:

```
fontColor(value: Color | number | string | Resource)
```

其中,Color 枚举有 White、Black、Blue、Brown、Gray、Green、Grey、Orange、Pink、Red、Yellow、Transparent 等类型。number 类型是十六进制的表示方式,以 0x 开头,后面共 6 位,每 2 位是一个颜色值,对应了红、绿、蓝三个颜色的值。比如 0x1f4e79,红色、绿色、蓝色的值是"1f""4e""79",换算成十进制分别为 31、78、121。string 也是后面共 6 位,每 2 位是一个颜色值,对应了红、绿、蓝三个颜色的值,但是开头是以 # 开头而不是 0x,比如上面的 0x1f4e79 就可以写成'#1f4e79'。

举例:

```
Text('文字')
.fontColor(Colol.Red)          //以 Color 枚举类型来设置字体颜色为红色
.fontColor(0xff0000)           //以 number 类型的形式来设置字体颜色为红色
.fontColor('#ff0000')          //以 string 类型的形式来设置字体颜色为红色
```

2) fontSize 属性

fontSize 属性用于设置字体大小。格式如下:

```
fontSize(value: number | string | Resource)
```

fontSize 的参数类型可以是 number、string、Resource。是 number 时不需要写单位 fp，是 string 时 fp 可写可不写。

举例：

```
Text('文字')
.fontSize(24)                    //以 number 类型的形式来设置字体大小
.fontSize('24')                  //以 string 类型的形式来设置字体大小
.fontSize('24fp')                //以带单位的 string 类型的形式来设置字体大小
```

3）fontStyle 属性

fontStyle 属性用于设置字体样式。格式如下：

```
fontStyle(value: FontStyle)
```

设置时使用枚举类型 FontStyle 来设置，FontStyle 有 Normal（正常）和 Italic（斜体）两个属性。

举例：

```
Text('文字')
.fontStyle(FontStyle.Normal)             //设置字体样式为正常
.fontStyle(FontStyle.Italic)             //设置字体样式为斜体
```

4）fontWeight 属性

fontWeight 属性用于设置字体粗细。格式如下：

```
fontWeight(value: number | FontWeight | string)
```

number 取值范围为[100，900]，取值间隔为 100，默认为 400，取值越大，字体越粗。fontWeight 中相应的枚举值为：Lighter（较细）、Normal（正常）、Medium（中等）、Bold（较粗）、Bolder（非常粗）几种。string 可以使用"100""200""400""900""lighter""regular""medium""bold""bolder"等形式。

举例：

```
Text('文字')
.fontWeight(100)                         //使用 number 形式设置字体粗细为 100
.fontWeight(FontWeight.Lighter)          //使用 FontWeight 形式设置字体粗细为较细
.fontWeight('100')                       //使用 string 形式设置字体粗细为 100
.fontWeight('lighter')                   //使用 string 形式设置字体粗细为较细
```

5）fontFamily 属性

fontFamily 属性用于设置字体类型，比如“宋体”“黑体”、Times New Roman 等。格式如下：

```
fontFamily(value: string | Resource)
```

字体设置时，还可以将多个字体使用“，”进行分隔，按照顺序的优先级进行生效。

举例：

```
Text('文字')
.fontFamily('黑体')          //设置字体为宋体
.fontFamily('Arial')         //设置字体为 Arial
.fontFamily('宋体,Arial')    //如果系统有宋体，则用宋体，如果系统没有宋体，则使用 Arial 字体
```

6）textAlign 属性

使用 textAlign 属性可以设置文本的对齐方式。格式如下：

```
textAlign(value: TextAlign)
```

其入口参数中使用的是 TextAlign 枚举量，这个枚举量有三个枚举类型：Start、Center、End。

举例：

```
Text('鸿蒙').width(300).fontSize(30).borderWidth(2)      //宽度为 300vp,边框线宽为 2vp
```

如上所示，Start 指的是水平对齐首部，Center 指的是水平对齐居中，End 指的是水平对齐尾部。也就是我们常说的左对齐、居中和右对齐。

7）文本最大行数，超过行数剪裁设置

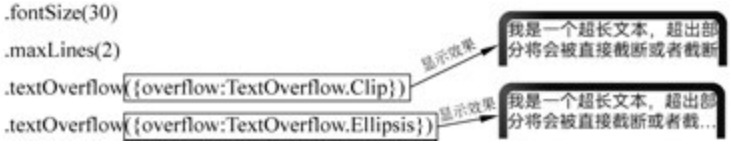
当文本内容较多超出了 Text 组件范围时，可使用 textOverflow 设置文本截取方式，需配合 maxLines 最大文本行数使用，单独设置不生效。maxLines 比较简单，比如 maxLines(1)设置行数为 1 行，maxLines(2)设置行数为 2 行。而 textOverflow 的格式如下：

```
textOverflow({overflow:TextOverflow})
```

其中，TextOverflow 枚举类型有三个：Clip、None、Ellipsis。从目前的版本来看，Clip 和 None 基本一样，都是直接截断，而 Ellipsis 则是超出的部分用省略号替代。

举例：

```
Text('我是一个超长文本,超出部分将会被直接截断或者截断后使用...代替')
```



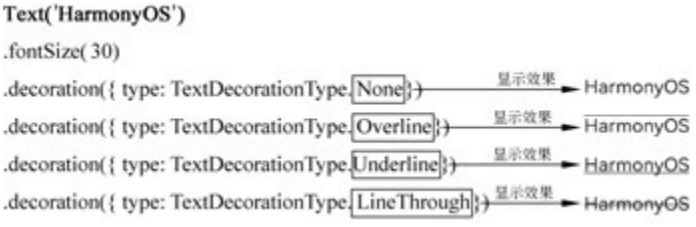
8）文本装饰线 decoration 属性

使用 decoration 设置文本装饰线样式及其颜色。格式如下：

```
decoration({type:TextDecorationType, color:Color})
```

decoration 包含 type 和 color 两个参数，其中，type 用于设置装饰线样式，参数 TextDecorationType 为枚举类型，值有 None、Overline、Underline、ineThrough 等。color 为可选参数，使用了前面讲过的 Color 枚举类型。

举例：



3.1.2　Image 组件

Image 为图片组件，常用于在应用中显示图片。Image 支持加载 string、PixelMap 和 Resource 类型的数据源，支持 PNG、JPG、BMP、SVG 和 GIF 类型的图片格式。

1. 接口

```
Image(src: string | PixelMap | Resource)
```

string 格式可用于加载网络图片。比如浏览新闻的时候，图片一般从网络加载而来，Image 组件支持加载网络图片。例如 Image('https://www.example.com/xxx.png')，而此时图片还显示不出来，因为应用访问网络时需要申请 ohos.permission.INTERNET 权限，HarmonyOS 提供了一种访问控制机制即应用权限，用来保证这些数据或功能不会被不当或恶意使用。具体操作是在 DevEco Studio 工程的 entry\src\main 目录下的 module.json5 文件中的“"module":{”后面直接加入下面虚框中的内容来申请网络权限。

```
{
  "module":{
      "requestPermissions":[
        {
          "name": "ohos.permission.INTERNET"
        }
      ],
    ...
  }
}
```

PixelMap 格式为像素图，常用于图片编辑的场景，此处不做展开讲解。

Resource 是访问本地图片的推荐方式。这种方式需要我们提前将图片放入到 DevEco Studio 工程的 entry\src\main\resources\base 目录下的 media 文件夹下面。比如我们在其他文件夹中有一个 0.jpg 图片，选中后直接按 Ctrl+C 组合键进行复制：①将工程文件左侧 entry\src\ain\resources\base 目录下的 media 文件夹选中。②右击，选择 Paste 命令。③弹出一个 Copy 对话框，这里可以修改图片文件名，此处我们选择默认，直接单击右下角的 OK 按钮。④此时就可以看到 medial 文件夹下多了一张名为 0.jpg 的图片文件，具体如图 3-2 所示。

将图片放入 media 文件夹后就可以使用 Resource 调用了。调用的时候，可以通过“$r('app.type.name')”的形式引用资源。同样，app 代表应用内 resources 目录中定义的资源。type 代表资源类型(或资源存放位置)，可取 color、float、string、plural、media 其中之一，此处就是 media。name 代表资源命名，我们在选图片的时候只取文件名，不留后缀。比如我们上面放到 media 文件下的 0.jpg，就可以用 $r('app.media.0')去调用。

```
@Entry
@Component
struct second {
```

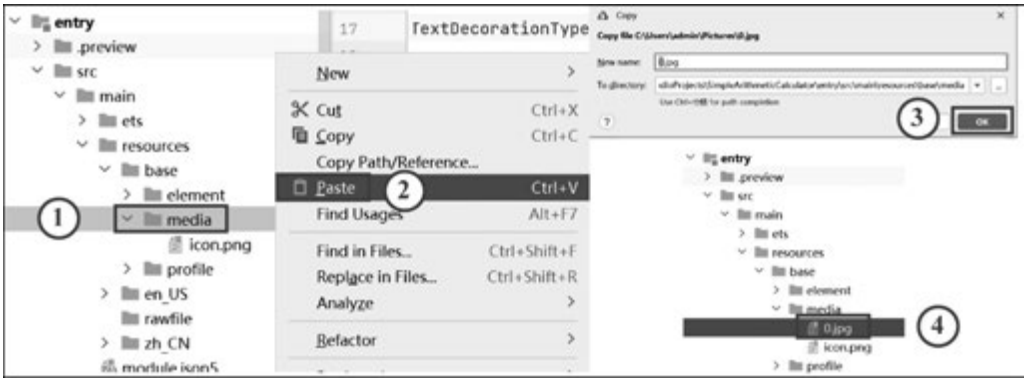


图 3-2　将图片放入 media 文件夹的过程

```
  build() {
    Column() {
    Image( $ r('app.media.0'))
    }
  }
}
```

通过上面的代码，图 3-3 所示的原图在预览器中显示出如图 3-4 所示的效果。图是显示出来了，但是显示得不全。这时就需要对图像显示效果进行属性设置。

图 3-3　0.jpg 原图

图 3-4　仿真器中的效果

2. 属性设置

1）width 属性和 height 属性

对于 Image 组件的属性设置，最简单的两个属性是 width()和 height()，它们分别是对图片宽度和高度的设置。

举例：

```
Image( $ r('app.media.0'))    //通过资源调用的方式调用了 media 文件夹下的 0.jpg 图片
.width(300)                   //设置宽度为 300vp
.height(100)                  //设置高度为 100vp
```

通过图 3-5 所示的效果和图 3-3 的原图比较，我们可以看出，图片没有完全展示。这个展示方式，就需要使用 objectFit()属性进行设置了。

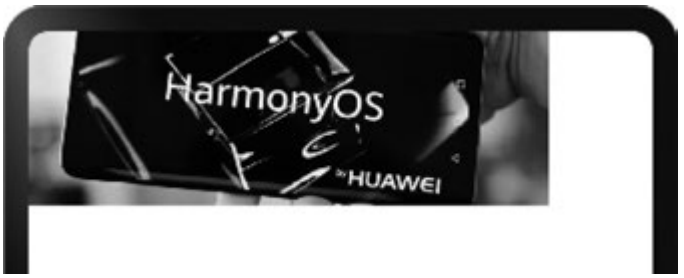


图 3-5　设置 width()和 height()属性后的显示效果

2）objectFit 属性

为了使图片在页面中有更好的显示效果，有时候需要对图片进行缩放处理。这就可以使用 objectFit 属性设置图片的缩放类型，objectFit 的参数类型为 ImageFit。objectFit 格式如下：

```
objectFit(value: ImageFit)
```

入口参数中使用了 ImageFit 枚举类型，枚举类型有 Contain、Cover、Fill、ScaleDown、Auto、None 等。其中，Contain 表示保持宽高比缩小或放大，使图片完全在显示边界内；Cover(默认值)表示保持宽高比缩放，使图两边都大于或等于边界；Fill 表示不保持宽高比缩放，使图片充满显示边界；ScaleDown 表示保持宽高比显示，图片缩小或者保持不变；Auto 表示自适应显示；None 表示保持原有尺寸。

```
Image( $ r('app.media.0'))              //通过资源调用的方式调用 media 文件夹下的 0.jpg
   .width(300)                          //设置宽度为 300vp
   .height(100)                         //设置高度为 100vp
   .backgroundColor(Color.Orange)       //为了在图片比较小时也能看到 Image 边界，设置了背景色
   .objectFit(ImageFit.Contain)         //Contain 模式
   .objectFit(ImageFit.Cover)           //Cover 模式
   .objectFit(ImageFit.Fill)            //Fill 模式
   .objectFit(ImageFit.ScaleDown)       //ScaleDown 模式
   .objectFit(ImageFit.Auto)            //Auto 模式
   .objectFit(ImageFit.None)            //None 模式
```

在上述代码中，为了能看到 Image 组件的边界，我们专门添加了橘色的背景色，设置了 backgroundColor(Color. Orange)属性，当图片比较小的时候，就可通过背景色看出 Image 组件的边界。上面 6 种模式，预览器中查看的效果如图 3-6 所示。

3.1.3　TextInput 组件

TextInput 组件用于输入单行文本，响应输入事件。TextInput 使用也非常广泛，如应用登录账号密码、发送消息等。

1. 接口

```
TextInput(value?:{placeholder?:ResourceStr,text?:ResourceStr,controller?:
TextInputController})
```

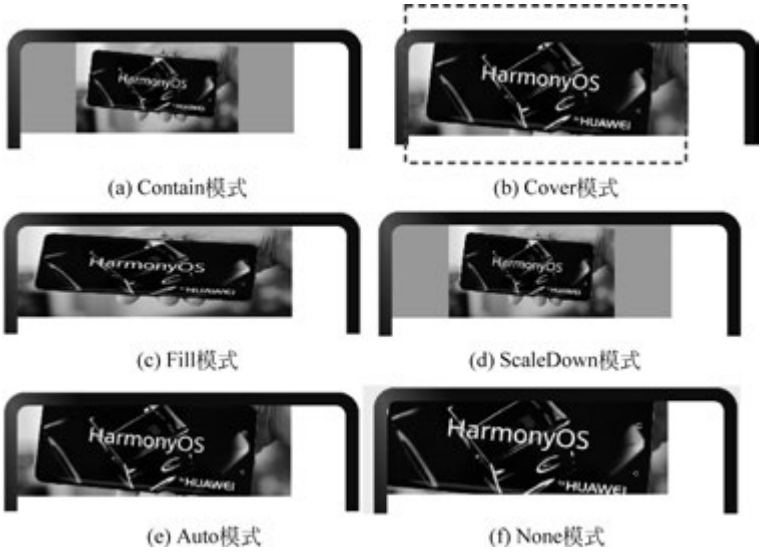
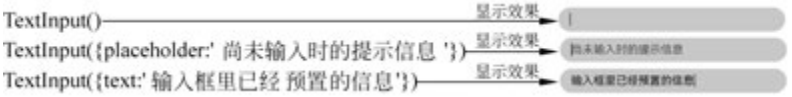


图 3-6　ImageFit 的几种模式的效果图

value 后面有问号，说明 value 可以不输入，所以 TextInput()的括号里可以为空。其中，placeholder 用来设置尚未输入时的提示文本；text 设置输入框当前的文本内容；controller 用于设置 TextInput 控制器。

举例：



下面讲解 controller 的 TextInput 控制器，我们需要给自定义组件设置一个成员变量 tiController，这个变量的类型就是 TextInputController，然后通过 new 实例化一个控制器。再使用 controller 的 caretPosition(value：number)来设置光标位置。这里的入口参数 number 是指从字符串开始到光标所在位置的字符长度。下面这个示例就通过单击一个按钮，实现了将光标的位置移到第二个字符后面的功能。

示例：

```
@Entry
@Component
struct TextInputDemo {
  tiController:TextInputController = new TextInputController()
  build(){
    Column(){
      TextInput({controller:this.tiController})
      Button('设置光标到第二个字符位置')
        .onClick(() => {
          this.tiController.caretPosition(2)
        })
  }
}
```

通过上面的例子我们可以看出，随着我们输入“鸿蒙应用开发”光标应该一直跟在输入

的最后一个字符的位置，我们单击下方按钮后，光标被移到了第二个字符后面的位置，如图 3-7 所示。

(a) 输入过程中光标的位置　　(b) 单击下方按钮后光标的位置

图 3-7　代码 controller 控制后的效果

2. 属性设置

1）输入文本属性设置

和 Text 组件一样，TextInput 组件也支持文本通用属性的 fontColor、fontSize、fontStyle、fontWeight、fontFamily 的设置。

举例：

```
TextInput({placeholder:'尚未输入时的提示信息'})      //为 TextInput 设置提示信息
  .fontColor(Color.Blue)                            //设置输入字体颜色为蓝色
  .fontSize(20)                                     //设置字体大小为 20fp
  .fontStyle(FontStyle.Italic)                      //设置字体为斜体
  .fontWeight(FontWeight.Bold)                      //设置字体加粗显示
  .fontFamily('Arial')                              //设置字体为 Arial 字体
```

如图 3-8 所示，从显示效果可以看出，这些字体的设置，并不能对尚未输入时的提示信息的字体进行设置，而是对用户输入的字体进行了设置。如果要对提示信息的字体进行设置，则需要使用 placeholderColor 和 placeholderFont 两个属性来设置。

(a) 未输入内容时提示信息的显示效果　　(b) 输入HarmonyOS后的显示效果

图 3-8　提示信息和用户输入信息的显示效果

2）提示信息文字设置

placeholderColor()设置比较简单，只需要在入口参数中增加一个颜色就可以了。比如需要将提示文字的颜色设成蓝色，就可以使用以下方式中的任意一种进行设置：placeholderColor(Color.Blue)、placeholderColor(0x0000ff)、placeholderColor('#0000ff')。

placeholderFont()可以通过在入口参数中添加{}的形式来设文本大小 size，粗细 weight，样式 style，字体 family。这些属性之间使用逗号隔开。例如：

```
placeholderFont({size:20,weight:FontWeight.Bold,family:'Arial',style:FontStyle.Italic})
```

把上面的那个示例再拿出来设置一下提示信息的文字样式：

```
TextInput({placeholder:'尚未输入时的提示信息'})
 .fontColor(Color.Blue)
 .fontSize(20)
 .fontStyle(FontStyle.Italic)
 .fontWeight(FontWeight.Bold)
 .fontFamily('Arial')
.placeholderFont({size:20,weight:FontWeight.Bold,style:FontStyle.Italic})//设置提示文字
                                                                          //字体
```

```
.placeholderColor(Color.Blue)                    //设置提示文字字体的颜色
```

(a) 未输入内容时提示信息的显示效果　(b) 输入HarmonyOS后的显示效果

图 3-9　增加提示文字的设置后的显示效果

3）输入类型设置

在使用 TextInput 组件进行输入的时候，有时需要输入密码，而输入密码的时候，我们希望用“……”来替代。这个时候就需要使用 TextInput 组件的 type 属性进行设置了。type 属性的格式如下：

```
type(value:InputType)
```

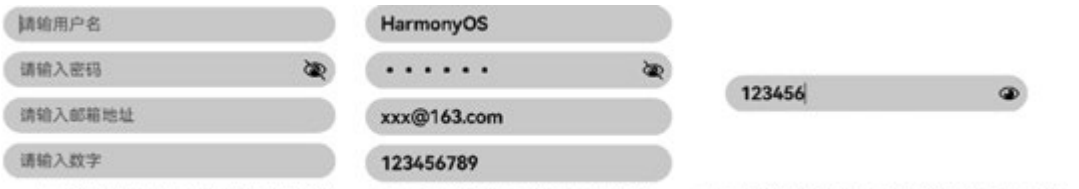
type 的参数类型为枚举 InputType 类型，InputType 枚举类型包括 Normal、Password、Email、Number 等形式。

其中，Normal 是基本输入模式，支持输入数字、字母、下画线、空格、特殊字符；Password 是密码输入模式，输入内容后会立刻变成小黑点，能防止旁人偷窥；Email 是 E-mail 地址输入模式；Number 是纯数字输入模式。

举例：

```
TextInput({placeholder:'请输用户名'})
  .fontSize(20)
  .fontWeight(FontWeight.Bold)
  .type(InputType.Normal)
TextInput({placeholder:'请输入密码'})
  .fontSize(20)
  .fontWeight(FontWeight.Bold)
  .type(InputType.Password)
TextInput({placeholder:'请输入邮箱地址'})
  .fontSize(20)
  .fontWeight(FontWeight.Bold)
  .type(InputType.Email)
TextInput({placeholder:'请输入数字'})
  .fontSize(20)
  .fontWeight(FontWeight.Bold)
  .type(InputType.Number)
```

通过上述例子，我们测试一下就可以发现，设成 Password 模式后右侧就会多一小眼睛图标，单击眼睛图标后会将输入的内容在隐藏和显示之间切换，还有 Number 模式下，只能输入数字字符，输入其他字符时无法输入进去的。具体如图 3-10 所示。



(a) 4种类型未输入时的显示效果　(a) 4种类型输入内容后的效果　(c) 密码类型单击右侧眼睛图标后的效果

图 3-10　4 种输入类型的显示效果

3. 输入文本事件

通过 TextInput 进行输入的时候，我们希望有一个事件，能实时地记录输入的内容。这个时候，就需要使用 onChange 事件。onChange 事件的格式如下：

```
onChange(callback:(value: string) => void)
```

我们可以看到 onChange 事件的入口参数就是一个回调函数。所谓回调函数，就是指输入内容一旦发生变化，就会被触发的函数。也就是说一旦输入的内容改变一次，里面的入口参数中的箭头函数就会被执行一次。这里的 value 就是输入文本里的内容。

举例：

```
@Entry
@Component
struct Index{
  @State inputValue:string = ''          //定义一个被@State 修饰的变量，来记录用户的输入内容
  build() {
    Column(){
      Text(this.inputValue)              //被@State 修饰过，所以变量改变 Text 组件显示就刷新
      TextInput()
        .onChange((value) =>{            //回调函数
          this.inputValue = value        //在回调函数中用 inputValue 变量记录输入的内容
        })
    }
  }
}
```

上述输入文本事件的运行效果如图 3-11 所示。

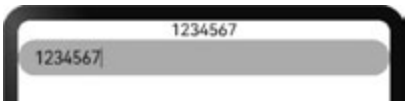


图 3-11　输入文本事件的运行效果

3.1.4　Button 组件

Button 组件是使用频率非常高的一个组件。主要用来响应单击操作，可以包含子组件。

1. 接口

接口格式如下：

```
Button(label?:ResourceStr,options?:{type?:ButtonType, stateEffect?:boolean})
```

这里有两个入口参数，一个用于设置按钮的文字 label、一个用于设置按钮的选项 options。lable 就是按钮上要显示的文字。比如 Button('登录')，这样按钮上就会显示“登录”两个字。options 有两个参数。其中一个是 type，它的类型是 ButtonType 枚举类型，用于描述按钮的显示样式。如图 3-12 所示，ButtonType 有三种样式：胶囊型按钮 Capsule(默认值，圆角默认为高度的一半)、圆形按钮 Circle、普通按钮 Normal(默认不带圆角)。

图 3-12　按钮的三种样式效果

stateEffect 是 boolean 类型的参数，它只有 true 和 false 两种情况，代表了按钮按下时是否开启按压态显示效果。当设置为 true 时按压效果开启，当设置为 false 时按压效果关闭，默认是 true。

2. 属性设置

Button 按钮的属性跟其他的都比较像，常用的高度、宽度、字体大小、字体粗细、字体样式、字体颜色、背景色等都可以进行设置，效果如图 3-13 所示。

举例 1：

```
Button('登录')
    .width(200)
    .height(40)
    .fontSize(28)
    .fontWeight(FontWeight.Medium)
    .fontStyle(FontStyle.Italic)
    .backgroundColor('#F95D25')
```

通过 3.1.3 节我们知道，当 Button 组件的 type 样式为普通按钮 Normal 的时候，按钮的四个角是直角。我们其实可以通过 borderRadius(value:BorderRadiuses)这个属性来设置四个角的圆角半径，从而设置按钮四个角为圆形。比如 borderRadius(8)就是设置四个角的圆角半径为 8。

举例 2：

```
Button({type:ButtonType.Normal})
    .height(50)
    .width(150)
    .borderRadius(10)
Button({type:ButtonType.Normal})
    .height(50)
    .width(150)
    .borderRadius(25)
```

可以看出，当设置四个角的圆角半径为高度的一半的时候，就跟胶囊样式的 ButtonType.Capsule 一样了，如图 3-14 所示。

图 3-13　按钮的显示效果

图 3-14　设置按钮为圆角

3. 按钮单击事件

Button 组件的单击事件使用得非常多，使用的时候是在 onClick()的入口函数中，传入一个箭头函数形式的回调函数。每次 Button 组件被单击这个回调函数就会被调用一次。

```
Button('登录').onClick(() =>{ /* 处理单击事件的逻辑 */ })
```

下面举一个使用 if 语句来控制显示哪一个控件的一个按钮单击实例。

举例：

```
@Entry
@Component
struct Index{
  @State count:number = 0                          //定义一个被@State 修饰的变量
  build() {
    Column({space:10}){
      Button('count + 1')                          //按钮显示"count + 1"
        .onClick(() =>{                            //定义单击事件
          this.count++                             //每次单击按钮则 count 增加 1
        })
      if(this.count % 2 == 0){                     //如 count 是偶数
        Text(this.count + '是偶数')                 //则用 Text 组件显示 count 值 + '是偶数'
      }else{                                       //如 count 是奇数
        TextInput({text:this.count + '是奇数'})    //则用 TextInput 组件显示 count 值 + '是奇数'
      }
    }.width('100 % ')
  }
}
```

如图 3-15 所示，当单击完按钮后，count 的值就会增加 1。当 count 为偶数的时候，Text 组件就会显示出来，并显示当前的 count 值和“是偶数”；当 count 为奇数的时候，TextInput 组件就会显示出来，并显示当前的 count 值和“是奇数”。

图 3-15　单击后的运行效果图

4. 包含子组件

Button 组件是可以包含子组件的。比如可以在 Button 组件后面的花括号中使用容器组件，容器组件中还可以放置其他的组件，比如下面这个例子中，就放了一个 LoadingProgress 组件和一个 Text 组件。

举例 1：

```
Button({type:ButtonType.Normal}){                                  //设置为 Normal 类型
 Row(){                                                            //子组件含容器组件 Row
   LoadingProgress().width(30).height(30).color(Color.White)       //LoadingProgress 子组件
   Text('加载中...').fontSize(24).fontColor(Color.White)           //Text 子组件
 }
}.height(40).width(135)                                            //设置 Button 组件尺寸
```

上述代码的运行效果如图 3-16 所示。

图 3-16　button 组件示例“加载中...”运行效果

举例 2：

```
Button({type:ButtonType.Circle}) {                          //设置为 Circle 类型
   Image( $ r('app.media.delete')).width(50).height(50)     //调 media 文件夹下的图片
}.width(80).height(80)                                      //设置 Button 组件尺寸
```

上述代码的运行效果如图 3-17 所示。

图 3-17　button 组件示例“delete”运行效果

3.1.5　LoadingProgress 组件

LoadingProgress 组件用于显示加载进度，比如应用的登录界面，单击“登录”按钮后，显示“正在登录”的进度条状态。LoadingProgress 的使用非常简单，没有入口参数，没有子组件，只需要设置颜色和宽高就可以了，显示效果如图 3-18 所示。

举例：

```
LoadingProgress()                   //LoadingProgress 组件无入口参数
  .color(Color.Blue)                //设置颜色为蓝色
  .height(60).width(60)             //设置高度和宽度都为 60vp
```

图 3-18　LoadingProgress 的显示效果

3.1.6　单选按钮 Radio 组件

Radio 组件是一个单选按钮，这个组件的应用也比较多。

1. 接口

可以通过调用接口来创建，接口调用形式如下：

```
Radio(options: {value: string, group: string})
```

其中，value 是单选按钮的名称，group 是单选按钮所属群组的名称。使用的时候，经常和 Text 组件配合使用。

```
Radio({value:'男', group:'radioGroup'})
Text('男')
Radio({value:'女', group:'radioGroup'})
Text('女')
```

效果如右侧所示：◯男 ◯女

上面还有一个 group 的设置，这个例子中组的名称是一样的，组名都是 radioGroup，说明它们两个是在一个组内。如果在同一个组中，它们就是互斥的。即单击选中一个，另一个

就会取消选中。

2. 属性设置

Radio 组件仅支持选中和未选中两种样式的属性设置，还支持高度和宽度的设置，不支持自定义颜色和形状的设置。

选中状态可以使用 checked(value:boolean)属性来设置，状态分别为 false 和 true，设置为 true 时表示单选按钮被选中，默认是 false。像上面这个例子，可以在创建值为“男”的 Radio 组件的时候就把它的这个属性设为 true。

```
Radio({value:'男', group:'radioGroup'})
Text('男')
Radio({value:'女', group:'radioGroup'})
Text('女')
```

效果为：◉男 ○女。因为两个 Radio 组件在一个组内，所以单击右侧的 Radio 组件后，右侧会变为选中状态，左侧选中状态会消失，单击后的效果为：◉男 ○女。

3. 触发事件

Radio 组件可以绑定 onChange 事件来响应选中后的事件处理。onChange()函数的入口参数是一个回调函数，回调函数的参数是一个 boolean 参数，传入的是当前该 Radio 组件的选中状态。使用方法如下：

```
onChange((isChecked:boolean) =>{
   if(isChecked) {
     //需要执行的操作
   }
})
```

接下来举一个例子。

```
Radio({value:'男', group:'radioGroup'})
  .onChange((isChecked) =>{
    if(isChecked){
    this.message = '现在选中的性别是"男"'}
  })
Text('男')
Radio({value:'女', group:'radioGroup'})
  .onChange((isChecked) =>{
    if(isChecked){
    this.message = '现在选中的性别是"女"'}
  })
Text('女')
```

上述代码的运行效果如图 3-19 所示。

图 3-19　值改变事件

3.1.7　切换按钮 Toggle 组件

Toggle 组件主要可以提供一个类似于开关的功能。可以提供勾选框样式、状态按钮样式及开关样式。

1. 接口

Toggle 组件接口如下：

```
Toggle(options:{type:ToggleType, isOn?:boolean})
```

Toggle 有两个参数：type 和 isOn。type 使用的是枚举类型 ToggleType，ToggleType 枚举样式有三种：开关样式 Switch(样式如)、单选按钮样式 Checkbox(样式如)和状态按钮样式 Button(样式如 status button)。其中 Switch 和 Checkbox 都不包含子组件，而 Button 可以包含子组件。isOn 是代表开关是否打开，true：打开，false：关闭，默认值为 false。

举例：

```
Toggle({type:ToggleType.Checkbox, isOn:false})
Toggle({type:ToggleType.Checkbox, isOn:true })
Toggle({type:ToggleType.Switch, isOn:false})
Toggle({type:ToggleType.Switch, isOn:true })
Toggle({type:ToggleType.Button, isOn:false}){
  Text('按钮').fontSize(20).fontColor('#ffffff')
}.backgroundColor('#6798F9').selectedColor(Color.Red).width(120)
Toggle({type:ToggleType.Button, isOn:true}){
  Text('按钮').fontSize(20).fontColor('#ffffff')
}.backgroundColor('#6798F9').selectedColor(Color.Red).width(120)
```

2. 属性设置

我们在前边含有子组件的 Button 样式中就使用了 backgroundColor()和 selectedColor()两个属性。其中，backgroundColor()用来设置未选中状态下的按钮颜色，selectedColor()用来设置选中状态下的按钮颜色。

另外还有一个属性设置：switchPointColor()。这个设置仅对 type 为 ToggleType.Switch 的时候生效。用来设置 Switch 类型的圆形滑块颜色。

举例：

```
Toggle({type:ToggleType.Switch}).switchPointColor(Color.Pink)
```

3. 触发事件

与 Radio 组件相似，Toggle 组件的触发事件也是用 onChange()来响应选中后的事件处理的。onChange()函数的入口参数是回调函数，回调函数的参数是一个 boolean 参数，传入的是当前该 Toggle 组件的选中状态。使用方法如下：

```
Toggle({ type: ToggleType.Switch, isOn: false })
  .onChange((isOn:boolean) =>{
      if(isOn) {
        //需要执行的操作
      }
  })
```

4. Toggle 实例

我们手机经常需要打开和关闭蓝牙，下面用 Toggle 做一个蓝牙开关的简单例子(这里

只做界面，并不真正去控制蓝牙）。我们希望在上面写这个页面的标题“蓝牙模式”，紧接着下面有一个蓝牙开关横条，横条左侧有“蓝牙”两个字，右侧有一个滑动开关。当滑动开关状态发生改变后，屏幕下侧有一个信息提示，但是这个提示在显示后几秒自动消失。这种出现又自动消失的提示信息，我们通常称之为气泡消息。蓝牙开关界面设计如图 3-20 所示。

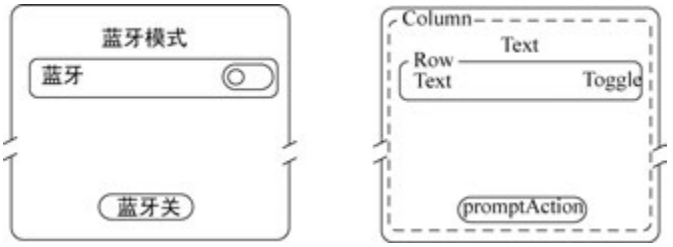


图 3-20　蓝牙开关界面设计

经分析，我们只需要在根目录下放置一个 Column 容器，容器的上面放置一个用于显示标题“蓝牙模式”的 Text 组件，下面放置一个 Row 容器。Row 容器里面放置 2 个内容，一个是左侧的用于显示“蓝牙”两个字的 Text 组件，右侧是一个设置成 Switch 模式的 Toggle 组件。分析完之后我们就可以写代码了。

在正式写代码之前，要介绍一些 promptAction。此处只讲它的 showToast()方法（气泡信息展示方法）。

首先需要在使用的页面的顶部先将 promptAction 的 API 引入进来：

```
import promptAction from '@ohos.promptAction'
```

然后，在需要弹出信息的地方调用 promptAction 的 showToast 方法即可。showToast 的入口参数中需要使用花括号输入 message 信息，使用方法如下：

```
promptAction.showToast({message:'气泡提示信息'})
```

promptAction 的其他方法我们就不介绍了，接下来根据上面的分析把页面的代码写出来：

```
@Entry
@Component
struct ToggleDemo{
  build(){
    Column(){
      Text('蓝牙模式')
      Row(){
        Text('蓝牙')
        Toggle({type:ToggleType.Switch})
      }
    }
  }
}
```

通过图 3-21 可见，还需要对字体的大小和样式进行设置。把标题“蓝牙模式”字体大小设为 28，字体加粗，添加.fontSize(28).fontWeight(FontWeight.Bold)设置。把“蓝牙”字体设为 24。然后将 Row 组件宽度设为 95%，把 Column 组件的宽度设为 100%，Row 的宽度设为 95%，再添加一个浅蓝（“#bdd7ee”）的背景色。为了美观，再将 Row 组件的四个角

图 3-21 “蓝牙模式”初始效果

都设一个半径为 15 的圆角，添加 borderRadius(15)。然后再添加一个轴向的两端对齐 justifyContent(FlexAlign.SpaceBetween)(这将在 3.2 节详细介绍)，这样“蓝牙”Text 组件和 Toggle 组件就能分布在 Row 组件的两端了。

添加完属性设置之后的代码如下：

```
@Entry
@Component
struct ToggleDemo{
  build(){
    Column(){
      Text('蓝牙模式').fontSize(28).fontWeight(FontWeight.Bold)
      Row(){
        Text('蓝牙').fontSize(24)
        Toggle({type:ToggleType.Switch})
      }.backgroundColor('#bdd7ee').width('95%').borderRadius(15)
        .justifyContent(FlexAlign.SpaceBetween)
    }.width('100%')
  }
}
```

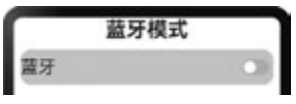


图 3-22 属性设置后的显示效果

属性设置后的显示效果如图 3-22 所示。

然后需要对 Toggle 添加开关事件。由于用到了 promptAction，所以在文件的最上部分需要添加导入 import promptAction from '@ohos.promptAction'。然后对 Toggle 组件进行事件设置。

```
.onChange((isOn) =>{
 promptAction.showToast({message:isOn?'蓝牙开':'蓝牙关'})
})
```

这里面使用了一个条件运算符(?:)，如果开关 isOn 为 true，则 message 就是“蓝牙开”，否则就是“蓝牙关”。添加完后，本实例就结束了，运行效果如图 3-23 所示。

最终的完整代码如下：

```
import promptAction from '@ohos.promptAction'
@Entry
@Component
struct ToggleDemo{
  build(){
    Column({space:10}){
      Text('蓝牙模式').fontSize(28).fontWeight(FontWeight.Bold)
      Row(){
        Text('蓝牙').fontSize(24)
        Toggle({type:ToggleType.Switch})
          .onChange((isOn) =>{
            promptAction.showToast({message:isOn?'蓝牙开':'蓝牙关'})
          })
      }
      .backgroundColor('#bdd7ee')
```

```
        .width('95 % ').borderRadius(15)
        .justifyContent(FlexAlign.SpaceBetween)
      }.width('100 % ')
    }
}
```

图 3-23　示例“蓝牙开关”最终效果

3.1.8　进度条 Progress 组件

当开发的软件有比较耗时的操作，需要实时地将进度展示在界面上时，就需要用到 Progress 组件了。

1. 接口及属性设置

```
Progress(options:{value:number, total?:number, type?:ProgressType})
```

在入口参数中，value 用于设置初始进度值，total 用于设置进度总长度，type 用于设置 Progress 的样式。Progress 的样式设置使用的是 ProgressType 枚举类型，这个枚举类型的选项有：线性进度条 Linear(默认)、环形无刻度进度条 Ring、环形有刻度进度条 ScaleRing、圆形进度条 Eclipse、胶囊进度条 Capsule。

1) 线性进度条举例

```
Progress({value:20,total:100,type: ProgressType.Linear}).width(200).height(50)
```

设置完成效果如下：

可以用 color()设置已经完成部分的进度条颜色，用 backgroundColor()设置未完成部分的颜色。使用 style(value:ProgressStyleOptions)设置进度条的宽度。ProgressStyleOptions 中有设置进度条宽度的属性 strokeWidth(默认值为 4vp，不支持百分比设置)、设置环形进度条总刻度数的属性 scaleCount、设置环形进度条刻度粗细的属性 scaleWidth(刻度粗细大于

进度条宽度时，为系统默认粗细，默认为 2vp，不支持百分比设置）、进度平滑动效的开关 enableSmoothEffect（默认 true）。

```
Progress({value:20,total:100,type:ProgressType.Linear}).width(200).height(50)
.color(Color.Red).backgroundColor(Color.Pink).style({strokeWidth:15})
```

设置完成的效果如下：

2）环形无刻度进度条举例

```
Progress({value:40,total:150,type:ProgressType.Ring})
.width(100).height(100)
```

同上面一样，默认已完成进度部分为蓝色，进度条宽度为 2.0vp。

下面这个例子将已完成进度部分的颜色换成了灰色，进度条宽度调为了 15vp：

```
Progress({value:40,total:150,type:ProgressType.Ring})
 .width(100).height(100)
 .color(Color.Grey)
 .style({strokeWidth:15})
```

3）环形有刻度进度条举例

```
Progress({value:20,total:150,type:ProgressType.ScaleRing})
 .width(100).height(100)
```

对于默认的刻度数量，可以通过前面提到的 style 方法中的 ProgressStyleOptions 参数进行调整：使用 scaleWidth 设置进度条的总刻度数，并通过 scaleWidth 调整刻度的粗细。以下代码示例展示了具体设置：将环形有刻度进度条的宽度（strokeWiddth）设为 15vp、总刻度数（scaleCount）设为 20、刻度粗细（scaleWidth）设为 5vp，同时使用 background()方法将未完成部分的颜色设置为黑色。

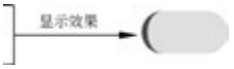
```
Progress({ value:20, total:150, type:ProgressType.ScaleRing })
.width(100).height(100).backgroundColor(Color.Black)
.style({ strokeWidth:15, scaleCount:20, scaleWidth:5 })
```

4）圆形进度条举例

```
Progress({value:10,total:150,type:ProgressType.Eclipse })
.width(100).height(100)
```

5）胶囊进度条举例

```
Progress({value:10,total:150,type:ProgressType.Capsule})
.width(100).height(50)
```



2. Progress 实例

下面我们做一个 Progress 组件的小例子。在界面上放置一个进度条和两个按钮，再放一个 Progress 组件，然后单击左侧按钮，进度条值就减少，单击右侧按钮，进度条值就增加。

从图 3-24 所示的设计图中可以看出，需要一个 Column 容器组件。Column 容器组件中有三个内容，一个是用于放置题目的 Text 组件、一个是用于展示进度的 Progress 组件，

还有一个用于存放两个 Button 组件的 Row 容器组件。Row 容器组件里存放了两个带有子组件的 Button 组件。带有子组件的两个 Button 组件里面都是含有一个 Image 组件和一个 Text 组件(两个组件里的 Image 组件和 Image 组件的顺序不同)。

图 3-24　Progress 实例的设计分析图

我们设置 Progress 控件的 type 为胶囊类型的 Capsule 类型,值为 0,总进度值为 100。然后将一个向左的箭头图标(left. png)和一个向右的箭头图标(right. png)放入工程的 entry\src\main\resources\base\media 文件夹下面,这样就可以使用资源提取符 $ r('')获取图片了,具体获取方式是: $ r('app. media. left')、$ r('app. media. right),把两张图片的宽和高都限定为 50vp。根据以上信息,我们可以写出如下代码:

```
Column(){
 Text('进度条实例')
 Progress({value:0,total:100,type:ProgressType.Capsule})
 Row(){
  Button(){
   Row(){
    Image( $ r('app.media.left'))
      .width(50).height(50)
    Text('减少 - ')
   }
  }
  Button(){
   Row(){
    Text(' + 增加')
    Image( $ r('app.media.Right'))
    .width(50).height(50)
   }
  }
 }
}
```

从上面的代码效果可以看出,我们还需要做以下属性设置。

(1) 需要加大加粗标题 Text 的字体,设置字体大小为 28fp,设置字体粗细为粗体: .fontSize(28) .fontWeight(FontWeight. Bold)。

(2) 需要设置进度条的宽度为窗体的 98%,设置进度条的高度为 50vp。所以需要对其进行如下设置: .width('98%'). style({strokeWidth:50})

(3) 需要将带子组件的 Button 组件的宽度设置为 150vp,高度设置为 70vp。具体设置方式为: .width(150). height(70)。然后将里面的 Text 子组件的字体大小设置为 30fp,颜色设置为白色。具体设置为: .fontSize(30). fontColor(Color. White)。

(4) 为了让所有的组件和手机外边之间留有一定的空白,对 Column 组件设置了一个 padding(10)。为了组件和组件之间留有间隙,我们在 Row 组件和 Column 组件的入口参

数中都使用了{space:具体值}的方式进行了设置。

设置完成后的代码如下：

```
Column({space:10}){
  Text('进度条实例') .fontSize(28).fontWeight(FontWeight.Bold)
  Progress({value:0,total:100,type:ProgressType.Capsule})
    .width('98 % ').style({strokeWidth:50})
  Row({space:40}){
   Button(){
    Row({space:10}){
      Image( $ r('app.media.left'))
        .width(50).height(50)
      Text('减少 - ').fontSize(30).fontColor(Color.White)
    }
   }.width(150).height(70)
   Button(){
    Row({space:10}){
      Text(' + 增加').fontSize(30).fontColor(Color.White)
      Image( $ r('app.media.right'))
       .width(50).height(50)
      }.width(150).height(70)
     }
   }
}.padding(10)
```

图 3-25 Progress 应用示例效果图

上述代码的运行效果如图 3-25 所示。

这样界面方面的代码就写完了，就剩下写逻辑代码了。我们需要在这个页面中定义一个被@State 修饰的 progressValue 变量，用这个变量来记录目前的进度值。

```
@State progressValue: number = 0
```

然后给两个按钮设置 onClick 事件响应。

左侧的按钮每单击 1 次，progressValue 值减 1，右侧的按钮每单击 1 次，progressValue 值加 1。

左侧单击事件：

```
.onClick(() = >{
    this.progressValue -= 1
})
```

右侧单击事件：

```
.onClick(() = >{
    this.progressValue += 1
})
```

这里要注意，progressValue 值不能高于 100，也不能低于 0。所以 progressValue 值变化之后要在左侧的事件中增加一个判断 if(this.progressValue＞100){this.progressValue＝0}，在右侧的事件里增加一个判断 if(this.progressValue＞100){this.progressValue＝0}。

最后还需要使用 progressValue 对 Progress 控件仅进行赋值，使用.value()属性来设置。具体的设置方法为：.value(this.progressValue)。

这些全部完成后，本实例的全部功能就完成了。所有代码如下：

```
@Entry
@Component
struct ProgressDemo {
  @State progressValue: number = 0
  build() {
    Column({ space: 10 }) {
      Text('进度条实例').fontSize(28).fontWeight(FontWeight.Bold)
      Progress({ value: this.progressValue, total: 100, type: ProgressType.Capsule })
        .width('98%')
        .style({ strokeWidth:50})
        .backgroundColor('#d9d9d9')
        .value(this.progressValue)
      Row({space:40}){
        Button(){
          Row({space:10}){
            Image($r('app.media.left')).width(50).height(50)
            Text('减少 - ').fontSize(30).fontColor(Color.White)
          }
        }.width(150).height(70)
        .onClick(() => {
          this.progressValue -= 1
          if (this.progressValue > 100) {
            this.progressValue = 0
          }
        })
        Button(){
          Row({ space: 10 }) {
            Text('+ 增加').fontSize(30).fontColor(Color.White)
            Image($r('app.media.right')).width(50).height(50)
          }
        }.width(150).height(70)
        .onClick(() => {
          this.progressValue += 1
          if (this.progressValue > 100) {
            this.progressValue = 0
          }
        })
      }
    }.padding(10)
  }
}
```

运行效果如图 3-26 所示，单击“增加”按钮进度条就会向右增加，单击“减少”按钮进度条就会向左减少。

图 3-26　进度条实例最终运行效果

3.1.9　滑动条 Slider 组件

滑动条 Slider 组件通常用于快速调节设置值，如音量调节、亮度调节等应用场景。

1. 接口

Slider 组件的接口参数较多，接口格式如下：

```
Slider(options?:{value?:number, min?:number, max?:number, step?:number,
style?:SliderStyle, direction?:Axis, reverse?:boolean})
```

其中，value 是当前的进度值(默认是后面参数 min 的值)。min 是设置的最小值(默认值为 0)；max 是设置的最大值(默认是 100)。如果用户在输入 min 和 max 时不小心输入 min＞＝max 异常情况，系统会自动将 min 取成 0，max 取成 100。如果前面给定的 value 值不在[min, max]范围，则 value 靠近 min 取 min，value 靠近 max 取 max。step 指的是设置 Slider 滑动步长(默认值为 1，取值范围为[0.01，max])，当设置小于或等于 0 的值时，或设置成百分比形式的值时，系统也会重新将其设为默认值 1。

style 是使用 SliderStyle 枚举类型来设置 Slider 的滑块与滑轨显示样式的。SliderStyle 的枚举类型有 2 个：滑块在滑轨上 OutSet(默认)、滑块在滑轨内 InSet。

OutSet 的样式： 

InSet 的样式： 

direction 是滑动条的方向设置。设置时使用的是 Axis 枚举类型，有纵向 Vertical 和横向 Horizontal(默认)两个属性。

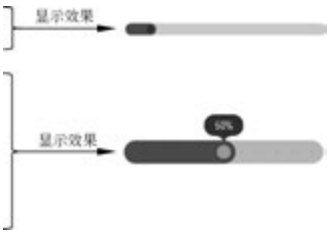
reverse 是 boolean 类型的，设置滑动条取值范围是否反向，横向 Slider 默认为从左往右滑动，竖向 Slider 默认为从上往下滑动，默认值是 false。

2. 属性设置

blockColor 属性用于设置滑块颜色，trackColor 设置滑轨的背景颜色。selectedColor 设置滑轨的已滑动部分颜色。showSteps 设置当前是否显示步长刻度值(默认值：false)。showTips 设置滑动时是否显示百分比气泡提示(默认值：false)。trackThickness 设置滑轨的粗细(默认值：当参数 style 的值设置 SliderStyle.OutSet 时为 4.0vp，设置 SliderStyle.InSet 时为 20.0vp)。

举例：

```
Slider({value:10,style:SliderStyle.InSet})
  .blockColor(Color.Blue)    //设置滑块颜色
Slider({value:10,style:SliderStyle.InSet})
  .blockColor(Color.Orange) //设置滑块颜色
  .trackColor(Color.Pink)    //设置滑轨颜色
  .showSteps(true)           //显示步长线
  .showTips(true)            //气泡形式显示进度百分比
  .trackThickness(40)        //将滑轨粗细调成 40vp
```



3. 事件

滑动条被滑动之后，会调用 onChange 事件，滑动条的 onChange 事件的格式如下：

```
onChange(callback:(value:number, mode:SliderChangeMode) => void)
```

其中 callback 函数是 Slider 滑动时触发的事件回调。value 是当前滑动进度值。若返回值有小数，可使用 number.toFixed()方法将数据处理为预期的精度。mode 表示拖动状态。拖动状态使用了 SliderChangeMode 枚举来设置，枚举值有：Begin 表示鼠标接触或者按下滑块(这个枚举类型对应的值是 0)；Moving 表示正在拖动滑块过程中(这个枚举类型对应的值是 1)；End 表示鼠标离开滑块(这个枚举类型对应的值是 2)。

举例：

现在页面结构体创建一个被@State 修饰的成员变量：

```
@State sliderMessage:string = ''
```

然后，添加一个 Text 组件用来显示信息。用 sliderMessage 来显示当前的值和模式。

```
Text(this. sliderMessage + '').height(100).fontSize(26)
```

然后给 Slider 添加 onChange 事件：

```
Slider({value:5,min:0,max:10,style:SliderStyle.InSet})
  .blockColor(Color.Orange)
  .trackColor(Color.Pink)
  .showTips(true)
  .trackThickness(40)
  .onChange((value,mode) =>{
    this.sliderMessage = '当前值:' + value.toFixed(2) + '模式:' + mode
  })
```

上述代码的运行效果如图 3-27 所示。

(a) Begin接触或者按下滑块时　(b) Moving正在拖动滑块过程中　(c) End鼠标离开滑块时

图 3-27　滑动条应用示例效果图

3.1.10　Select 组件

Select 组件提供下拉选择菜单，可以让用户在多个选项之间选择。

1. 接口

Select 组件的接口如下：

```
Select(options: Array<SelectOption>)
```

它的参数是一个 Array 数组。数组中的每一个选项都是一个 SelectOption 对象，SelectOption 对象由一个 value(下拉选项内容)键值对和一个 icon(下拉选项图片)键值对组成。

举例：

```
Select([{value:'选项 1', icon: $ r("app.media.icon")},
        {value:'选项 2', icon: $ r("app.media.icon")},
        {value:'选项 3', icon: $ r("app.media.icon")},
        {value:'选项 4', icon: $ r("app.media.icon")}])
```

2. 属性

设置属性时需要区分 3 个内容：下拉菜单本身、下拉菜单选项、下拉菜单选中项。这两个指代的位置可以参考图 3-28。后续设置属性时要注意区分。

1) 下拉菜单本身的属性设置

selected(value:number)：用于设置下拉菜单初始选项的索引，第一项的索引为 0。当

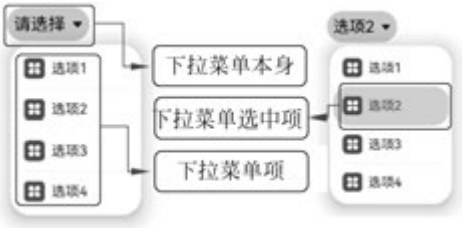


图 3-28　设置属性的 3 个内容

不设置 selected 属性时，默认选择值为－1，菜单项不选中。

value(value:string)：设置下拉菜单本身显示的内容。当菜单选中时选中的内容会将其替换。

font(value:Font)：可以设置下拉按钮本身的文本样式。Font 类型指的是使用花括号把 font 的 size、weight、family、style 等内容进行设置。比如 font({size: 20, weight: FontWeight. Nomal, family: 'Arial, style: FontStyle. Italic})，默认值是{size: '16fp', weight: FontWeight. Medium}。

fontColor(value: ResourceColor)：前面我们也用过 ResourceColor，ResourceColor 指的是可以用于设置颜色的所有类型。包括 Color 枚举类型、类似于 0xffffff 格式的 number 类型、类似于'#ffffff'格式的 string 类型和使用 $r('')引入资源的方式，引入系统资源或者应用资源中的颜色。fontColor 就是用来设置下拉按钮本身的文本颜色，默认值为'#E6000000'。

这里要单独讲一下 8 位形式的颜色值表示方式。以前碰到的颜色值表示方法都是 6 位形式，比如'#1122ff','11'(十六进制)对应的是红色值 17(十进制),'22'(十六进制)对应的是绿色值 34(十进制),'ff'(十六进制)对应的是蓝色值 255(十进制)。像'#E6000000'这种 8 位形式的颜色值表示方法，则是在原来 6 位值的前面加了 2 个数来表示透明度。比如'#E6000000',其中'e6'(十六进制)就是对应的透明度值 230(十进制),十进制的 0 值对应的是全透明，十进制的 255 对应的是全不透明，后面的 6 位跟上面的 6 位形式表示方法的含义就完全一样了。

2) 下拉菜单下拉选中项的属性设置

selectedOptionFont(value:Font)：设置下拉菜单选中项的文本样式。默认值：{ size: '16fp',weight: FontWeight. Regular }。

selectedOptionFontColor(value: ResourceColor)：设置下拉菜单选中项的字体颜色。默认值：'#FF000000'。

selectedOptionBgColor(value: ResourceColor)：设置下拉菜单选中项的背景色。默认值：'#33007DFF'。

3) 下拉菜单下拉选项的属性设置

optionFont(value:Font)：设置下拉菜单项的文本样式。默认值：{ size: '16fp', weight: FontWeight. Regular }。

optionFontColor(value: ResourceColor)：设置下拉菜单项的文本颜色。默认值：'#ff182431'。

optionBgColor(value: ResourceColor)：设置下拉菜单项的背景色。默认值：'#ffffffff'。

3. 事件

Select 组件最常用的一个事件是下拉菜单选中某一项的回调。

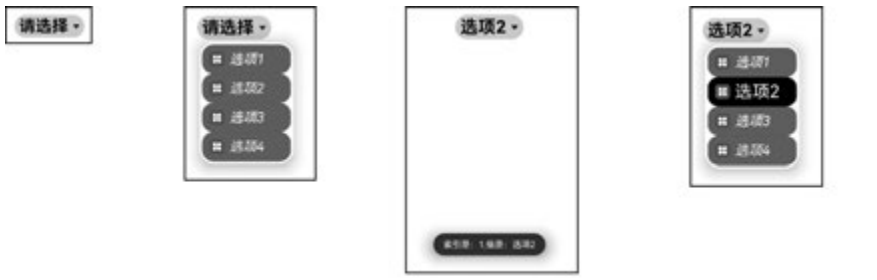
```
onSelect(callback: (index: number, value?: string) => void)
```

一旦下拉菜单进行选中操作，就会触发 onSelect 中的回调函数。其中，index 是选中项的索引(索引从 0 开始)，value 是选中项的值。

举例：

```
Select([{value:'选项 1', icon: $r("app.media.icon")},      //设置下拉菜单选项 1
        {value:'选项 2', icon: $r("app.media.icon")},      //设置下拉菜单选项 2
        {value:'选项 3', icon: $r("app.media.icon")},      //设置下拉菜单选项 3
        {value:'选项 4', icon: $r("app.media.icon")}])     //设置下拉菜单选项 4
  .value('请选择')                          //下拉菜单选项未被选中时下拉菜单本身显示的内容
  .font({size:28,weight:FontWeight.Bold})                 //设置下拉菜单本身的字体样式
  .fontColor('#000000')                                   //设置下拉菜单本身的字体颜色
  .selectedOptionFont({size:28,style:FontStyle.Normal})   //设置下拉菜单选中项的字体
  .selectedOptionFontColor(Color.White)                   //设置下拉菜单选中项的字体颜色
  .selectedOptionBgColor(Color.Black)                     //设置下拉菜单选中项的背景颜色
  .optionFont({size:22,style:FontStyle.Italic})           //设置下拉菜单项的字体样式
  .optionFontColor(Color.White)                           //设置下拉菜单项的字体颜色
  .optionBgColor(Color.Gray)                              //设置下拉菜单项的背景颜色
  .onSelect((index,value) =>{                             //菜单下拉项被选中的回调事件
   promptAction.showToast({message:`索引是:${index},值是:${value}`})
                          //使用 3.1.7 节中讲到的 promptAction 显示被选中的 index 和 value
  })
```

这段代码运行后的效果如图 3-29 所示。每个下拉菜单项的背景设置为了灰色，字体设置为了斜体。选择“选项 2”后，下侧会通过 promptAction 弹出一个气泡提示信息，提示索引号“1”(从 0 开始，选项 1 对应的索引号是 0)和该选项的内容“选项 2”。选中“选项 2”后，再次打开下拉菜单，可以看出菜单选中项“选项 2”和其他的菜单项不同，字体偏大(28fp)，字体样式不是斜体。



(a) 初始　(b) 单击下拉后　(c) 选中选项2后　(d) 再次打开下拉菜单

图 3-29　选择组件应用示例效果图

3.2　容器组件

一个丰富的页面需要很多组件组成，那么，如何才能让这些组件有条不紊地在页面上布

局呢？这就需要借助容器组件来实现。常用的容器组件有 Row、Column、List、Grid、Tab、Flex 等。

容器组件是一种比较特殊的组件，它可以包含其他组件，而且按照一定的规律布局，生成页面。容器组件除了放置基础组件外，也可以放置容器组件，通过多层布局的嵌套，布局出更丰富的页面。

3.2.1　主轴和交叉轴

线性布局容器表示按照垂直方向或者水平方向排列子组件的容器，ArkTS 提供了 Column 和 Row 容器来实现线性布局。Column 表示沿垂直方向布局的容器，Row 表示沿水平方向布局的容器。在学习容器组件之前，需要理解两个非常重要的概念——主轴和交叉轴。

1. 主轴和交叉轴的概念

在布局容器中，默认存在主轴和交叉轴两根轴，这两根轴相互垂直。不同容器中主轴的方向是不一样的。在 Column 容器中主轴为垂直方向，交叉轴为水平方向。在 Row 容器中主轴为水平方向，交叉轴为垂直方向，如图 3-30 所示。

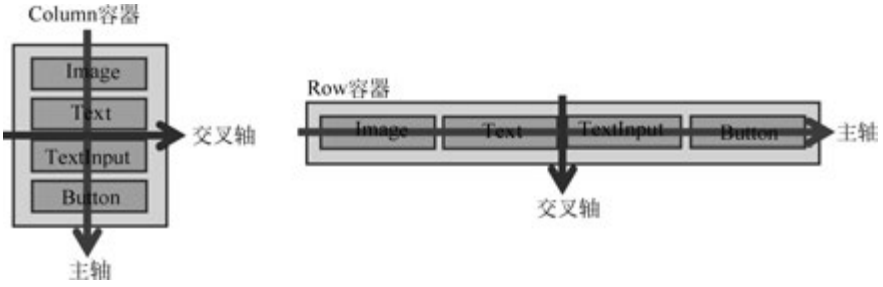


图 3-30　在 Column 和 Row 容器组件中主轴和交叉轴的方向

2. 主轴方向对齐

容器组件里面的子组件（图 3-31 中的 Image、Text、TextInput、Button）在主轴方向是用 justifyContent(value:FlexAlign)属性来进行对齐设置的。其参数是枚举类型 FlexAlign。

FlexAlign 的枚举量如下。

Start：元素在主轴方向首端对齐，第一个元素与行首对齐，同时后续的元素与前一个对齐，如图 3-31 所示。

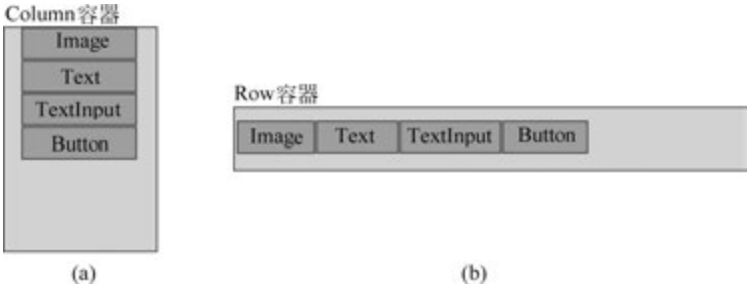


图 3-31　FlexAlign. Start 在 Column 和 Row 容器中的对齐方式

Center：元素在主轴方向中心对齐，第一个元素与行首的距离与最后一个元素与行尾距离相同，如图 3-32 所示。

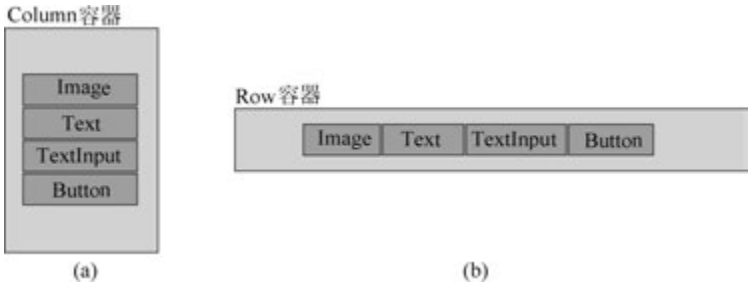


图 3-32　FlexAlign. Centert 在 Column 和 Row 容器中的对齐方式

End：元素在主轴方向尾部对齐，最后一个元素与行尾对齐，其他元素与后一个对齐，如图 3-33 所示。

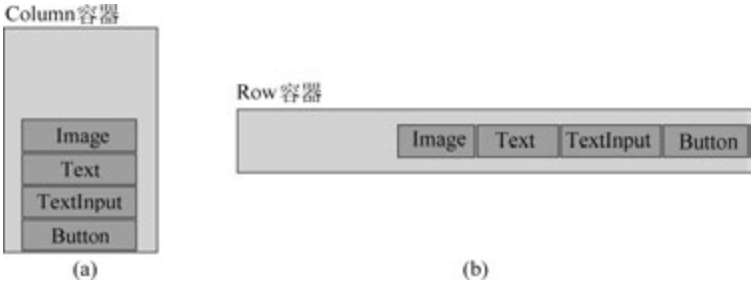


图 3-33　FlexAlign. End 在 Column 和 Row 容器中的对齐方式

SpaceBetween：主轴方向均匀分配弹性元素，相邻元素之间距离相同。第一个元素与行首对齐，最后一个元素与行尾对齐，如图 3-34 所示。

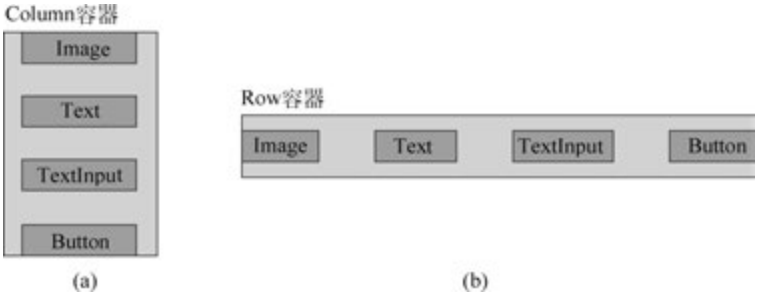


图 3-34　FlexAlign. SpaceBetween 在 Column 和 Row 容器中的对齐方式

SpaceAround：主轴方向均匀分配弹性元素，相邻元素之间距离相同。第一个元素到行首的距离和最后一个元素到行尾的距离是相邻元素之间距离的一半，如图 3-35 所示。

SpaceEvenly：主轴方向均匀分配弹性元素，相邻元素之间的距离、第一个元素与行首的间距、最后一个元素到行尾的间距完全一样，如图 3-36 所示。

3. 交叉轴方向对齐

子组件在交叉轴方向上的对齐方式使用 alignItems 属性来设置，Column 这类纵向容

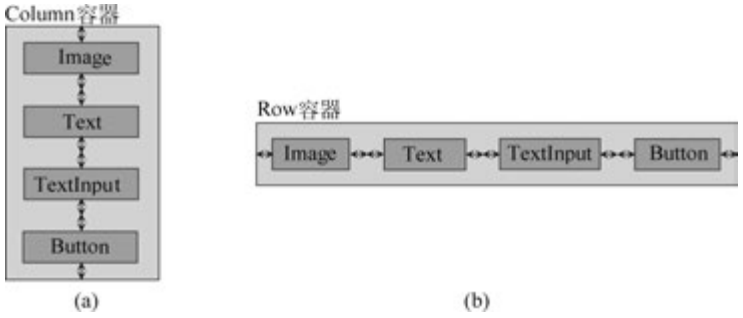


图 3-35　FlexAlign. SpaceAround 在 Column 和 Row 容器中的对齐方式

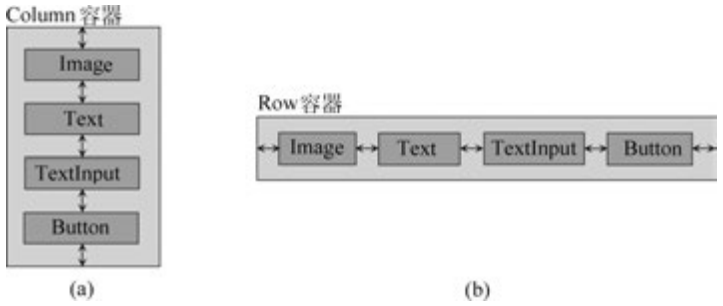


图 3-36　FlexAlign. SpaceEvenly 在 Column 和 Row 容器中的对齐方式

器的参数是枚举类型 HorizontalAlign，Row 这类横向容器的参数是枚举类型 VerticalAlign。

HorizontalAlign 的枚举量有：Start、Center、End。相应的三种对齐方式如图 3-37 所示。

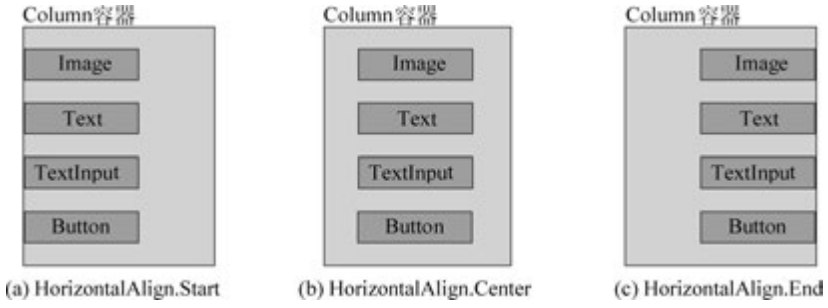


图 3-37　HorizontalAlign 的三种对齐方式

VerticalAlign 的枚举量有：Top、Center、Bottom。相应的三种对齐方式如图 3-38 所示。这里要注意，为了更为形象一些，没有使用 Start 和 End，而是使用了 Top 和 Bottom。

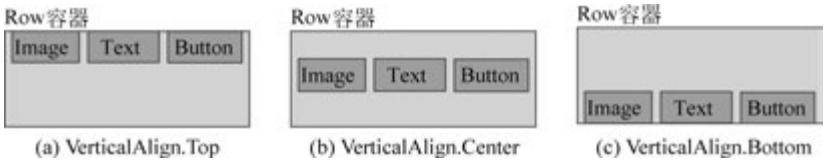


图 3-38　VerticalAlign 的三种对齐方式

3.2.2　Column 和 Row 容器组件

1. 接口

前面我们对 Column 容器组件和 Row 容器组件已经介绍了很多。下面介绍 Column 和 Row 容器的接口：

```
Column(value?:{space?: string | number})
Row(value?:{space?: string | number})
```

Column 和 Row 的接口都有一个可选参数 space，表示子组件在主轴方向上的间距。

举例：

```
Column({space:10}){     //用 number 设置
  Image()
  Text()
  Button()
}
Row({space: '10vp'}){     //用 string 设置
  Image()
  Text()
  Button()
}
```

上述代码的运行效果如图 3-39 所示。

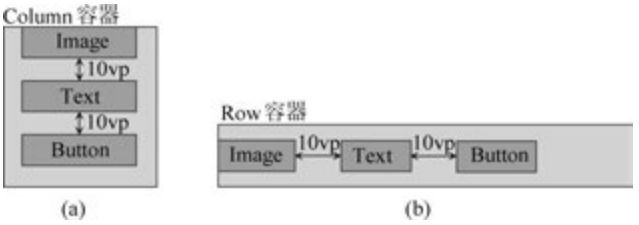


图 3-39　将 Column 和 Row 容器的入口参数中设置 space 为 10 的效果

2. Column 和 Row 容器应用实例

1）布局设计

我们学习了前面内容，就可以通过容器组件的组装，将页面变得有序规整了。比如想要做一个图 3-40(a)所示的关于“鸿蒙小课堂”的登录界面。

通过分析，可以得出图 3-40(b)所示的布局格式。根据上面的布局，可以写出如下代码：

```
Column({ space: 10 }) {
  Image( $ r('app.media.HWLogo'))
  Text('鸿蒙小课堂')
  Text('登录之后可以使用更多服务')
  TextInput({ placeholder: '请输入账号' })
  TextInput({ placeholder: '请输入密码' })
    .type(InputType.Password)
  Row() {
    Text('短信验证码登录')
    Text('忘记密码')
```

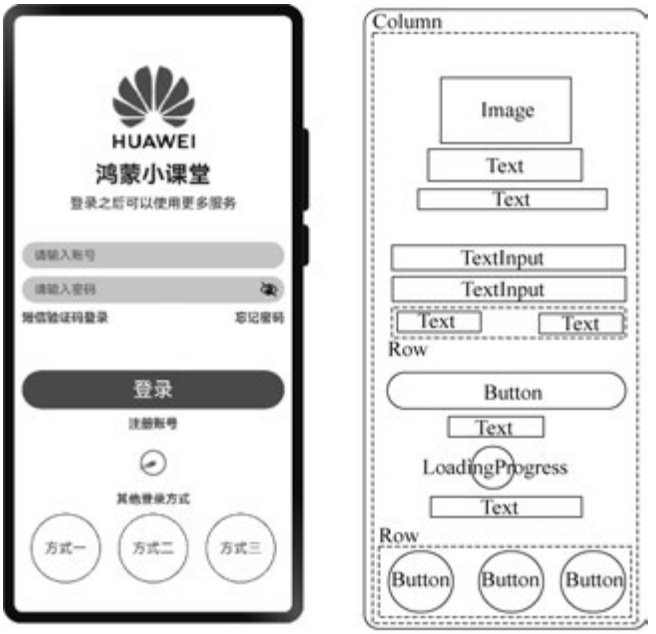


图 3-40　使用容器组件后的页面和代码的对应情况

```
  }
  Button('登录')
  Text('注册账号')
  LoadingProgress()
  Text('其他登录方式')
  Row(){
  Button('方式一',{type:ButtonType.Circle})
  Button('方式二',{type:ButtonType.Circle})
  Button('方式三',{type:ButtonType.Circle})
  }
}
```

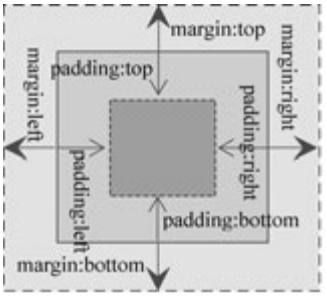


图 3-41　margin 和 padding 示意图

从布局可以看出，Column 组件里面的组件之间的距离并不是均匀的。这种不均匀的间隔就要用到 margin 和 padding 属性了。

2）margin 和 padding 属性

由图 3-41 可以看出，margin 是外边距。是自身和外部容器之间的距离，padding 是在自己的大小的基础上向内的一段距离。

可以使用. padding(10)的形式，直接将 padding 的 top、bottom、left、right 统一设成 10vp。也可通过在入口参数中使用花括号的形式对四个边分别进行设置，比如 . padding({top:10,bottom:10,left:10,right:10})

通常使用 padding 来设置容器里面的内容和容器边留有一定的余量。

```
Column(){                                    //设置一个外容器
  Row(){                                     //设置一个内容器
```

```
  }.backgroundColor(Color.Orange)          //将内容器背景色设为橘色
  .width('100%')
  .height('100%')
  .borderWidth(2)                          //为方便查看,给内容器设置一个 2vp 宽度的框
}.backgroundColor(Color.Pink)              //将外容器的背景色设为粉色
 .width('100%').height('100%')
 .padding(20)                              //设置外容器和内容器四边都保持 20vp 空白区域
```

可以通过 margin 来设置组件和组件之间的距离。比如下面的举例代码：

举例：

```
Column(){
  Button('按钮 1').width("90%")
  Button('按钮 2').width("90%")
   .margin({top:10})                       //按钮 2 和按钮 1 间空 10vp
  Button('按钮 3').width("90%")
   .margin({top:30})                       //按钮 3 和按钮 2 间空 30vp
  Button('按钮 4').width("90%")
   .margin({top:50})                       //按钮 4 和按钮 3 间空 50vp
   }.backgroundColor(Color.Pink)
   .width('100%').height('100%')
```

通过上面代码可以设置“按钮 2”和“按钮 1”之间的间隔为 10vp，设置“按钮 3”和“按钮 2”之间的间隔为 30vp，设置“按钮 4”和“按钮 3”之间的间隔为 50vp。

padding 和 margin 效果如图 3-42 所示。

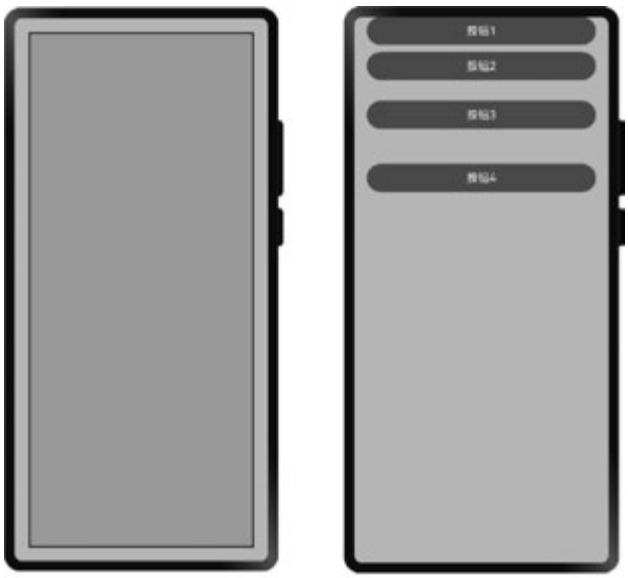


图 3-42　padding 和 margin 效果

3）属性设置

下面继续在前面“布局设计”中的布局代码的基础上增加属性设置。之前的代码灰色显示，新增的属性代码使用加黑显示。

```
Column({ space: 10 }) {
  Image( $r('app.media.HWLogo'))
    .height(120).width(120).margin({ top: 50 })
```

```
    Text('鸿蒙小课堂')
      .fontSize(30)
      .fontWeight(FontWeight.Bold)
    Text('登录之后可以使用更多服务')
      .fontSize(18)
      .margin({ bottom: 30 })
    TextInput({ placeholder: '请输入账号' }).backgroundColor('#d6d6d6')
    TextInput({ placeholder: '请输入密码' }).backgroundColor('#d6d6d6')
      .type(InputType.Password)
    Row() {
      Text('短信验证码登录').fontColor('#0974E9').fontWeight(FontWeight.Bold)
      Text('忘记密码').fontColor('#0974E9').fontWeight(FontWeight.Bold)
    }.justifyContent(FlexAlign.SpaceBetween).width('100%')
    Button('登录').width('100%').height(50).fontSize(26).margin({top:50})
    Text('注册账号').fontColor('#0974E9').fontWeight(FontWeight.Bold)
    LoadingProgress().width(60).height(60)
    Text('其他登录方式').fontColor('#666666').fontWeight(FontWeight.Bold)
    Row(){
      Button('方式一',{type:ButtonType.Circle}).fontSize(18).width(90).height(90)
        .backgroundColor(Color.Transparent).borderWidth(1).fontColor(Color.Gray)
      Button('方式二',{type:ButtonType.Circle}).fontSize(18).width(90).height(90)
        .backgroundColor(Color.Transparent).borderWidth(1).fontColor(Color.Gray)
      Button('方式三',{type:ButtonType.Circle}).fontSize(18).width(90).height(90)
        .backgroundColor(Color.Transparent).borderWidth(1).fontColor(Color.Gray)
    }.justifyContent(FlexAlign.SpaceAround).width('100%')
  }.width('100%').padding(10)
```

所有属性设置完毕后，就能达到图 3-39(a)所示的效果了。

3.2.3　List 和 Grid 容器组件

手机应用中常见到数据列表，如设置页面、通讯录、商品列表等，都使用了 List 和 Grid 容器组件。图 3-43(a)所示的两个页面中，“首页”中包含两个网格布局(Grid 列表)，“商城”中包含一个商品线性列表(List 列表)。列表中都包含相同宽度和高度的列表项，列表项可以包含文本等内容。ArkUI 提供了 List 组件和 Grid 组件(呈现形式如图 3-43(b)所示)，用这两个组件能很轻松地完成一些列表页面。

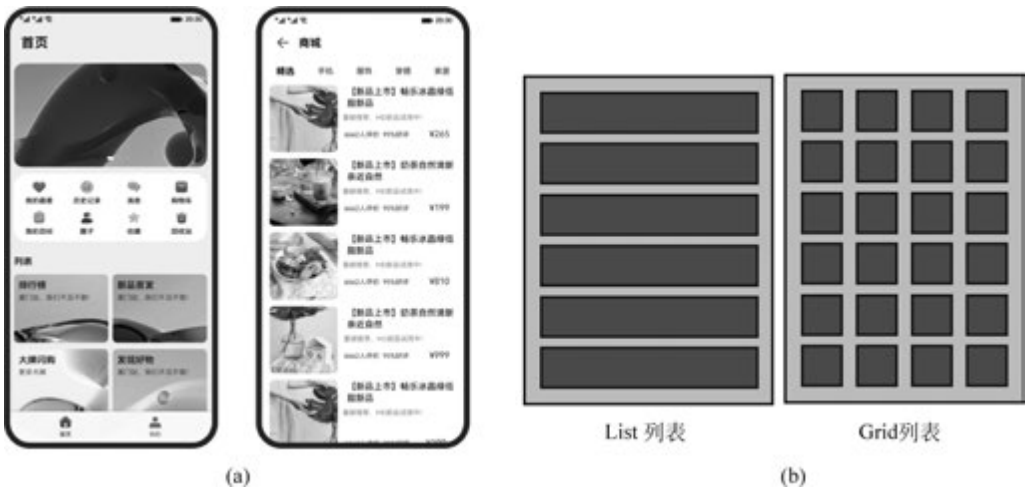


图 3-43　Grid 和 List 页面

1. List 容器组件

List 是常用的滚动类容器组件，和子组件 ListItem 一起使用。List 列表中的每一个列表项就是一个 ListItem 子组件。List 的子组件必须是 ListItem，ListItem 必须配合 List 来使用。但是，我们可以将某几个 ListItem 用 ListItemGroup 组装起来，这样就可以将 ListItem 进行分组显示了。List 和 ListItem 以及 ListItemGroup 的包含关系如图 3-44 所示

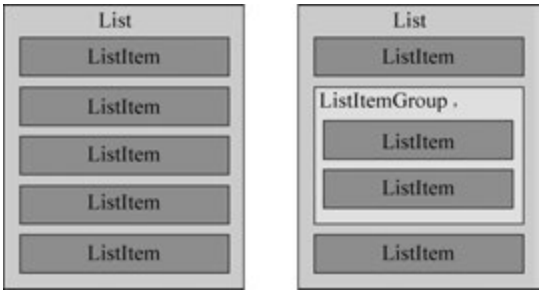


图 3-44　List 和 ListItem 以及 ListItemGroup 的包含关系

1）接口

```
List(value?{ space?:number|string, initialIndex?:number,scroller?:Scroller})
```

其中，space 用于设置列表项间距；initialIndex 用于设置 List 初次加载时起始位置显示的 ListItem；scroller 是控制器，可以控制 List 组件的滚动。

2）静态创建

List 静态地创建其列表项 ListItem 的内容：

ListItem 中含有单个组件：

```
List({space:10}){
   ListItem(){
     Text("列表项 1")
   }
   ListItem(){
     Text("列表项 2")
   }
   ListItem(){
     Text("列表项 3")
   }
   ListItem(){
     Text("列表项 4")
   }
}
```

上述代码效果如下：

ListItem 中含有多个组件的情况：

```
List({space:10}){
   ListItem(){
     Row() {
        Text("列表项 1")
        Toggle({type:ToggleType.Checkbox,isOn:true})
     }
   }
   ListItem(){
      Row() {
        Text("列表项 2")
        Toggle({type:ToggleType.Checkbox,isOn:true})
       }
   }
}
```

上述代码效果如下：

通过上面的代码，就可以把一个列表项创建出来了。但是仔细观察可以发现，ListItem内部的代码结构是一模一样的，这样写会显得代码很臃肿。ArkTS 通过 ForEach 提供了组件的循环渲染能力。我们可以使用 ForEach，在其中以嵌套 ListItem 的形式来代替多个平铺的、内容相似的 ListItem，从而减少重复代码。

3）使用 ForEach 循环渲染列表

ForEach 基于数组类型数据执行循环渲染。通过 ForEach 从数组中获取数据，并为每个数据项创建相应的组件。ForEach 必须在容器组件内使用。下面讲解 ForEach 循环的用法。

```
ForEach(
  arr: any[],
  itemGenerator: (item: any, index?: number) => void,
  keyGenerator?: (item: any, index?: number) => string
)
```

其中：

arr 是数组，是 ForEach 循环遍历用到的数组。允许设置为空数组，空数组场景下将不会创建子组件。

itemGenerator 是生成子组件的 lambda 函数，为数组中的每个数据项创建一个或多个子组件，单个子组件或子组件列表必须包括在 lamda 函数的花括号“{..}”中。子组件的类型必须是 ForEach 的父容器组件所允许的（例如，只有当 ForEach 父级为 List 组件时，才允许 Litemitem 子组件）。

keyGenerator 是生成键值的 lambda 函数，用于给数组中的每一个数据项生成唯一且固定的键值。键值生成器的功能是可以省去的，但是为了使开发框架能够更好地识别数组更改，提高性能，建议提供。将数组反向时，如果没有提供键值生成器，则 ForEach 中的所有节点都将重建。

4）ForEach 应用的例子

下面用 ForEach 举个例子：

```
@Entry
@Component
struct second {
  listName:string[] = ['列表项 1','列表项 2','列表项 3','列表项 4']
  build(){
    List({space:10}){
      ForEach(this.listName,(item:string) =>{
        ListItem(){
          Row()
            Text(item)
            Toggle({type:ToggleType.Checkbox,isOn:true})
          }
        }
      },item => item)
    }.padding(10)
  }
}
```

从上面代码可以看出，使用 ForEach 循环后，代码量要少了很多，也简洁了很多。上述代码的运行结果如图 3-45 所示。

图 3-45　ForEach 循环渲染

5）设置列表分割线

List 组件子组件 ListItem 之间默认是没有分割线的，部分场景子组件的 ListItem 间需要设置分割线，这时候可以使用 List 组件的 divider 属性。

divider 包含以下 4 个参数。

- strokeWidth：分割线的线宽。
- color：分割线的颜色。
- startMargin：分割线距离列表侧边起始端的距离。
- endMargin：分割线距离列表侧边结束端的距离。

举例：

```
@Entry
@Component
struct second {
  listName:string[] = ['列表项 1','列表项 2','列表项 3','列表项 4']
  build() {
    List({space:20}) {
      ForEach(this.listName,(item) =>{
        ListItem(){
          Row(){
            Text(item).fontSize(32)
            Toggle({type:ToggleType.Checkbox,isOn:true})
              .width(30).height(30)
          }.justifyContent(FlexAlign.SpaceBetween)
            .width('100%')
        }
      },item => item)
    }.padding(20)
    .divider({strokeWidth:2,color:Color.Blue,startMargin:10,endMargin:10})
  }
}
```

上述代码的运行效果如图 3-46 所示。

6）设置 List 排列方向

如图 3-47 所示，List 组件里面的列表项默认是按垂直方向 Axis. Vertical 排列的，如果想让列表沿水平方向排列，可以将 List 组件的 listDirection 属性设置为 Axis. Horizontal。例如：List(){…}. listDirection(Axis. Horizontal)。

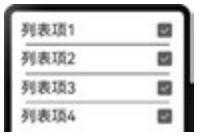


图 3-46　纵向列表应用示例效果图

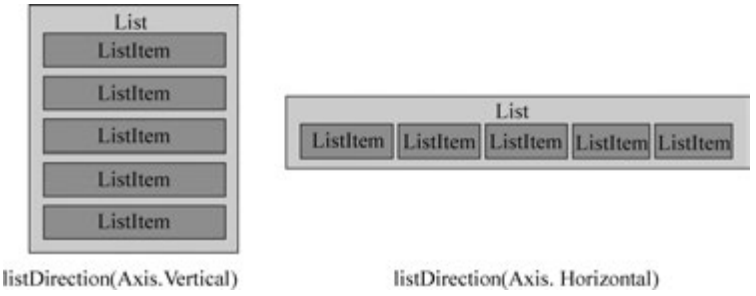


图 3-47　List 组件的排列方向

2. Grid 容器组件

Grid 组件为网格容器，是一种网格列表，由“行”和“列”分割的单元格组成。一般和子组件 GridItem 一起使用，如图 3-48 所示。

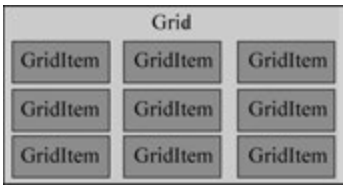


图 3-48　Grid 和 GridItem 组合使用

接口如下：只有一个 scroller 控制器的参数，可以控制 Grid 的滚动。

```
Grid(scroller?:Scroller)
```

属性如下：

columnsTemplate 用于设置当前网格布局的列的数量，单位是 fr；

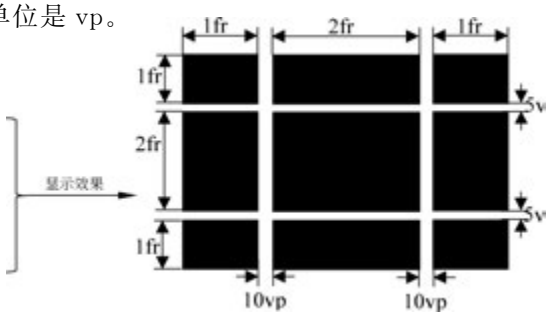
rowsTemplate 用于设置当前网格布局的行的数量，单位是 fr；

columnsGap 用于设置行与行的间距，单位是 vp；

rowsGap 用于设置列与列的间距，单位是 vp。

比如我们设置 Grid 属性如下：

```
Grid(){…}
  .columnsTemplate('1fr 2fr 1fr')
  .rowsTemplate('1fr 2fr 1fr')
  .columnsGap(10)
  .rowsGap(5)
```



下面我们再举个例子，为了方便起见，我们使用 ForEach 循环创建 GridItem：

```
@Entry
@Component
struct GridDemo{
```

```
  gridList:string[] = ['GridItem1','GridItem2','GridItem3','GridItem4','GridItem5',
                       'GridItem6','GridItem7','GridItem8','GridItem9']
  build(){
    Grid(){
      ForEach(this.gridList,(value) =>{
        GridItem(){
          Text(value).fontSize(33)
        }.backgroundColor('#CEE1F2')
      })
    }
      .columnsTemplate('1fr 2fr 1fr')
      .rowsTemplate('1fr 2fr 1fr')
      .columnsGap(10)
      .rowsGap(5)
  }
}
```

为了方便观看效果，我们可以通过单击预览器中的 按钮，将预览器的屏幕横过来，效果如图 3-49 所示。

3. 应用实例

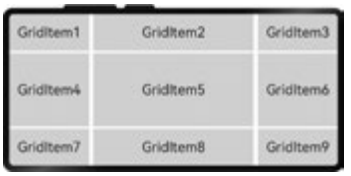


图 3-49　列表应用示例效果图

下面我们用 List 和 Grid 两个容器组件一起做一个类似于属性设置的页面。这里我们将尝试使用 2.2.3 节中讲到的在被@Entry 装饰的自定义组件(页面入口)中调用其他自定义组件的情况。本例共有三个页面文件 Index.ets、subGridItem.ets、subListItem.ets 和一个类文件 classContent.ets。

由图 3-50 可以看出，这个实例的页面中有三个容器组件，分别为 Column 容器组件、Grid 容器组件以及 List 容器组件。内置组件只有 2 个 Text 组件。而自定义组件有 2 种，一种是 Grid 中的自定义组件 1 和 List 组件中的自定义组件 2，自定义组件 1 中有 1 个 Column 容器组件中包含着 1 个 Image 组件和 1 个 Text 组件，自定义组件 2 中有 1 个 Row 容器组件中包含着 2 个 Image 组件和 1 个 Text 组件。

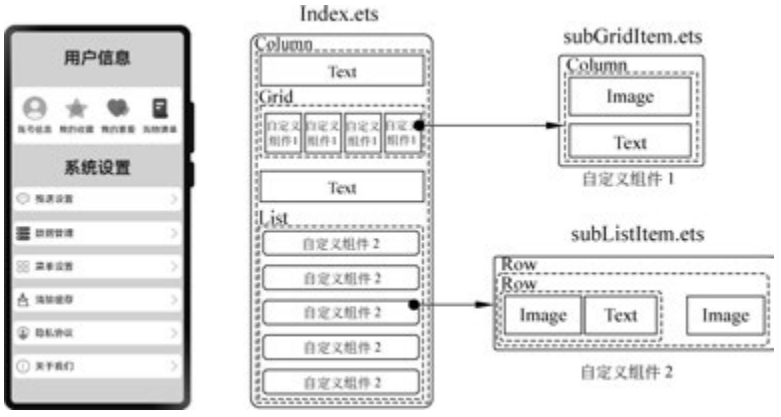


图 3-50　实例效果图及组件布局图

为了后续传递方便，我们先创建一个类用于存放图标以及图标下面的描述文字(如“账号信息”“我的收藏”等)。如图 3-51 所示，创建时需要先在左侧工程目录 entry\src\main\ets\pages 中的 pages 文件夹上右击，在弹出的快捷菜单中选择 New→ArkTS File 命令。这样就

能在 pages 文件夹下创建一个新的 ArkTS 文件了。

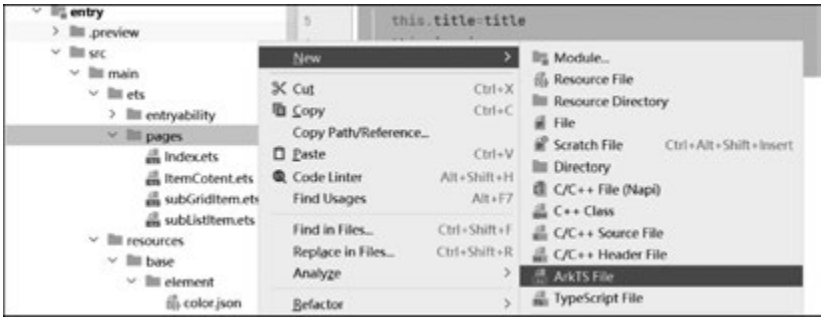


图 3-51　创建 ArkTS File

在弹出的新创建界面中输入我们要创建的类名 classContent，如图 3-52 所示。

图 3-52　输入新创建的 ArkTS 文件的名称

在 classContent.ets 文件中添加如下代码：

```
export class classContent{      //创建类，一定要用 export，这样其他文件才能导入使用
  title:Resource                //成员变量 title，注意此处类型为 Resource
  img?:Resource                 //成员变量 img，注意此处类型为 Resource
  constructor(title:Resource, img:Resource) {      //构造函数传入两个参数
    this.title = title          //传入的参数赋值给本类的 titile 成员变量，this 指代本类
    this.img = img              //传入的参数赋值给本类的 img 成员变量，this 指代本类
  }
}
```

使用同样的方法在 pages 文件夹中再创建一个自定义组件 1 文件，将文件取名为“subGridItem.ets”。根据图 3-50 所示的自定义组件 1 的组成样式，在 subGridItem.ets 文件中添加代码如下：

```
@Component        //此处用@Componet，不用@Entry，只有入口页面才用@Entry 修饰
export struct subGridItem{      //自定义组件 1(图 3-53)，要用 export 暴露出来以供其他文件调用
  img:Resource     //Resource 是资源，图片在 entry\src\main\resources\media 文件夹下
  title:Resource  //文字资源常放于 entry\src\main\resources\base\element\string.json
  build(){
    Column({space:10}){        //space:10 让 Column 容器内组件之间有 10vp 的间隔
      Image(this.img).width(50).height(50) //设置图像宽高均为 50vp
      Text(this.title).fontSize(18)        //设置文字的字体大小为 18fp
    }
  }
}
```

我们在 pages 文件夹下再创建一个自定义组件 2 文件，将文件取名为“subListItem.ets”。根据图 3-50 所示的自定义组件 2 的组成样式，在 subListItem.ets 文件中添加代码如下：

```
@Component      //此处用@Componet，不用@Entry，只有入口页面才用@Entry 修饰
export struct subListItem{      //自定义组件 2(图 3-54)，要用 export 暴露出来以供其他文件调用
  img:Resource          //Resource 是资源，图片资源在 entry\src\main\resources\media 文件夹下
  title:Resource//文字资源常放于 entry\src\main\resources\base\element\string.json 中
  build(){
    Row(){
```

```
      Row({space:10}){                                    //让 Row 内组件间留 10vp 间隔
        Image(this.img).width(30).height(30)              //宽高设为 30vp
        Text(this.title).fontSize(20)                     //文字大小设为 20fp
      }
      Image( $ r('app.media.right_grey')).width(20).height(20) //右侧图像高宽都设为 20vp
    }.justifyContent(FlexAlign.SpaceBetween)//SpaceBetween 可让 Row(2)和 Image(2)
//分布在 Row(1)组件的两端
    .borderRadius(8).width('100 % ')//Row(1)设宽为 100 % ,四周半径为 8vp 的圆角
.height(50).padding(5).backgroundColor('＃fff')//设高度为 50vp,Row(1)内的组件和
  }                  //Row(1)的外边之间留 5vp 空隙,设背景色为白色('＃fffff'可以简写成'＃fff')
}
```

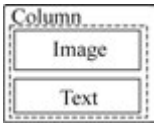


图 3-53　自定义组件 1(subGridItem)

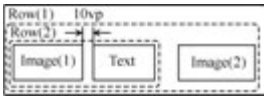


图 3-54　自定义组件 2(subGridItem)

在进行主页面代码编写之前,需要先完成文字资源和图片资源的准备工作。文字资源需要在 entry\src\main\resources\base\element\string.json 中添加以下代码:

```
{
  "string": [
    {
      "name": "module_desc",
      "value": "module description"
    },
    {
      "name": "EntryAbility_desc",
      "value": "description"
    },
    {
      "name": "EntryAbility_label",
      "value": "label"
    }
  ]
}
```

可将右侧内容加入string.json文件中的此处

```
,
    {
      "name": "user",
      "value": "账号信息"
    },
    {
      "name": "myfavorite",
      "value": "我的收藏"
    },
    {
      "name": "mylove",
      "value": "我的喜爱"
    },
    {
      "name": "shoppinglist",
      "value": "购物清单"
    },
    {
      "name": "news",
      "value": "推送设置"
    },
    {
      "name": "data",
      "value": "数据管理"
    },
    {
      "name": "memu",
      "value": "菜单设置"
    },
    {
      "name":
"erasestorage",
      "value": "清除缓存"
    },
    {
      "name": "privacy",
      "value": "隐私协议"
    },
    {
      "name": "aboutus",
      "value": "关于我们"
    }
```

图片资源(图 3-55)则需要把用到的图片放到 entry\src\main\resources\media 文件夹下就可以了。

图 3-55　放入工程 media 文件夹下的图片资源

接下来编写 Index.ets 主页面(也就是我们的入口页面),我们先从一个最简单的页面代码开始:

```
①                    //导入其他文件的位置
@Entry               //@Entry 修饰,表示 Index 页面是入口页面
@Component           //@Component 修饰,表明 Index 是自定义组件
struct Index {       //定义结构体 Index
  ②                  //添加自定义组件成员变量和成员函数的地方
  build() {          //自定义组件的描述函数
  ③                  //添加页面组件的地方
  }
}
```

代码中①②③标出了我们常用来添加代码的地方。

首先要在①处将我们的 1 个类文件和 2 个自定义组件文件导入进来。其中'./'是相对路径中的同级目录的意思。比如'./X'代表 X 文件的位置和当前的 Index 文件在同一个文件夹下。

```
import { classContent } from './classContent'
import { subGridItem } from './subGridItem'
import { subListItem } from './subListItem'
```

其次需要在②处定义两个数据类型为 classContent 的数组。其中 $r('')为资源引用的方式,$r('')中的单引号中的内容就是资源的位置。其中文字资源和图片资源在前面都准备好了,这里只需要按照对应的位置引用即可。

```
gridSubItem:classContent[] = [
    new classContent( $r('app.string.user'), $r('app.media.user')),
    new classContent( $r('app.string.myfavorite'), $r('app.media.favorite')),
    new classContent( $r('app.string.mylove'), $r('app.media.love')),
    new classContent( $r('app.string.shoppinglist'), $r('app.media.shoppinglist'))]
listSubItem:classContent[] = [
    new classContent( $r('app.string.news'), $r('app.media.news')),
    new classContent( $r('app.string.data'), $r('app.media.data')),
    new classContent( $r('app.string.memu'), $r('app.media.menu')),
    new classContent( $r('app.string.erasestorage'), $r('app.media.storage')),
    new classContent( $r('app.string.privacy'), $r('app.media.privacy')),
    new classContent( $r('app.string.aboutus'), $r('app.media.about'))]
```

gridSubItem 和 listSubItem 两个数组中存放的文字和图标如图 3-56 所示。

然后,需要在③处添加组件,可以先根据图 3-57 所示的页面设计图写出一段简单的页面代码:

```
Column(){
  Text('用户信息')
  Grid(){
  ④
  }
  Text('系统设置')
  List(){
  ⑤
 }
}
```

(a) gridSubItem数组

(b) listSubItem数组

图 3-56　gridSubItem 和 listSubItem 两个数组中存放的文字和图标

图 3-57　页面设计图

然后可以在上面的④和⑤处，分别使用 ForEach 语句来循环放置“自定义组件 1”和“自定义组件 2”。

在④处添加如下代码：

```
ForEach(this.gridSubItem,     //第 1 个参数表示要对 gridSubItem 内容遍历
  (item:classContent) =>{     //第 2 个参数是箭头函数，item 是遍历到的 gridSubItem 中的内容
    GridItem(){               //箭头函数的函数体中放与 Grid 对应的 GridItem
      subGridItem({img:item.img,title:item.title})   //GridItem 中放自定义组件 1，将 item 的
    }                 //title 和 item 的 img 分别传给 subGridItem 子组件的成员变量 title 和 img
})                            //第 3 个参数省略
```

在⑤处添加如下代码：

```
ForEach(this.listSubItem,     //第 1 个参数表示要对 listSubItem 内容遍历
  (item:classContent) =>{     //第 2 个参数是箭头函数，item 是遍历到的 listSubItem 中的内容
    ListItem(){               //箭头函数的函数体中放与 List 对应的 ListItem
      subListItem({title:item.title,img:item.img })  //ListItem 中放自定义组件 2，将 item 的
    }                 //title 和 item 的 img 分别传给 subListItem 子组件的成员变量 title 和 img
})                            //第 3 个参数省略
```

然后对各个组件的属性进行设置。最终 Index.ets 完整的代码如下：

```
import { classContent } from './classContent'
import { subGridItem } from './subGridItem'
import { subListItem } from './subListItem'
@Entry
@Component
struct Index {
  @State message: string = 'Hello World'
  gridSubItem: classContent[] = [
    new classContent( $r('app.string.user'), $r('app.media.user')),
    new classContent( $r('app.string.myfavorite'), $r('app.media.favorite')),
    new classContent( $r('app.string.mylove'), $r('app.media.love')),
```

```
      new classContent( $r('app.string.shoppinglist'), $r('app.media.shoppinglist'))]
    listSubItem: classContent[] = [
      new classContent( $r('app.string.news'), $r('app.media.news')),
      new classContent( $r('app.string.data'), $r('app.media.data')),
      new classContent( $r('app.string.memu'), $r('app.media.menu')),
      new classContent( $r('app.string.erasestorage'), $r('app.media.storage')),
      new classContent( $r('app.string.privacy'), $r('app.media.privacy')),
      new classContent( $r('app.string.aboutus'), $r('app.media.about'))]
    build() {
      Column() {
        Text('用户信息').fontSize(35).fontWeight('bold').margin({ top: 30, bottom: 40 })
        Grid() {
          ForEach(this.gridSubItem, (item: classContent) => {
            GridItem() {
              subGridItem({ img: item.img, title: item.title })
            }
          })
        }
        .rowsTemplate('1fr')
        .columnsTemplate('1fr 1fr 1fr 1fr')
        .rowsGap(12)
        .columnsGap(8)
        .height(120)
        .borderRadius(8)
        .backgroundColor('#fff')
        .padding(5)
        Text('系统设置').fontSize(35).fontWeight('bold').margin({top:30,bottom:20})
        List({ space: 20 }) {
          ForEach(this.listSubItem, (item: classContent) => {
            ListItem() {
              subListItem({ title: item.title, img: item.img })
            }
          })
        }
      }
      .width('100%').height('100%').padding(5).backgroundColor('#D9E2F3')
    }
}
```

最终显示效果如图 3-58 所示。

3.2.4 Tab 容器组件

ArkUI 开发框架提供了一种页签容器组件 Tabs，开发者通过 Tabs 组件可以很容易地实现内容视图的切换。页签容器 Tabs 的形式多种多样，不同的页面设计的页签不一样，可以把页签设置在底部、顶部或者侧边，如图 3-59 所示。

1. 组件接口

Tabs 组件仅可包含子组件 TabContent，每一页签对应一个内容视图 TabContent 组件。Tabs 组件的接口如下：

```
Tabs(value?: {barPosition?: BarPosition, index?: number, controller?: TabsController})
```

图 3-58　“属性设置页面”最终效果图

图 3-59　Tab 容器的例子

Tabs 的所有参数后面都有“?”号，都是可选参数，也就是说，所有的参数都有缺省值，可以不输入。这些参数的含义如下。

barPosition：用来设置 Tabs 的页签位置。它的参数是枚举量 BarPosition。BarPosition 有两个枚举量分别为 BarPosition. Start 和 BarPosition. End。默认值是 BarPosition. Start。

index：用来设置当前显示页签的索引。

controller：设置 Tabs 控制器。

2. Tabs 组件的组成及形式

Tabs 组件包含“导航部分”和“内容部分”，“导航部分”包含多个 tabBar，“内容部分”只显示 1 个 TabContent。单击到哪个 tabbar，tabbar 对应的 TabContent 就会显示在“内容部分”。

当导航部分在顶部时，称为“顶部导航”；在底部时，称为“底部导航”；在左侧时，称为“左侧导航”；在右侧时，称为“右侧导航”。Tabs 组件的组成及几种形式具体如图 3-60 所示。

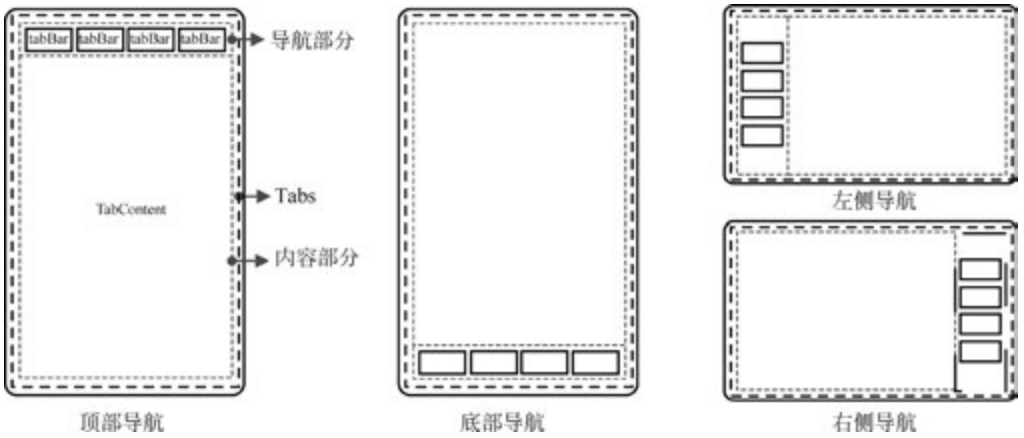


图 3-60　Tabs 组件的组成及几种形式

3. 基本用法

Tabs 组件可以在它后面的花括号中添加多个 TabContent，每个 TabContent 要设置 1 次 tabBar 的属性设置，比如下面就是它简单的使用方式：

```
Tabs(){
    TabContent(){              //放置第 1 个页面
      页面 1 的内容            //往第 1 个页面中放需要用到的组件
    }.tabBar('页面 1')         //设置第 1 个页面的标签
    TabContent(){              //放置第 2 个页面
      页面 2 的内容            //往第 2 个页面中放需要用到的组件
    }.tabBar('页面 2')         //设置第 2 个页面的标签
    …                          //放置其他页面
}
```

图 3-61　Tabs 组件应用示例

比如：

```
@Entry
@Component
struct Index {
  build() {
    Tabs() {
      TabContent() {
        Text('页面 1 中的内容')
      }.tabBar('页面 1 的标签')
      TabContent() {
        Text('页面 2 中的内容')
      }.tabBar('页面 2 的标签')
    }
  }
}
```

上述代码的运行结果如图 3-61 所示。

4. 属性设置

还可以对 Tabs 组件进行属性设置，有. barWidth 和. barHeight 属性，这两个属性设置的是导航部分的大小，如图 3-62 所示。

图 3-62　barWidth 和 barHeight 属性

设置属性时，可以给 Tabs 组件设置属性，也可以给 TabContent 设置属性。

举例：

```
Column(){
  Tabs(){
    TabContent(){                        //创建第 1 个 TabContent
    }.tabBar('红色')                     //创建第 1 个 TabContent 对应的 tabBar
      .backgroundColor(Color.Red)        //设置第 1 个 TabContent 的背景色属性为红色
    TabContent(){                        //创建第 2 个 TabContent
    }.tabBar('黄色')                     //创建第 2 个 TabContent 对应的 tabBar
      .backgroundColor(Color.Yellow)     //设置第 2 个 TabContent 的背景色属性为黄色
    TabContent(){                        //创建第 3 个 TabContent
    }.tabBar('蓝色')                     //创建第 3 个 TabContent 对应的 tabBar
      .backgroundColor(Color.Blue)       //设置第 3 个 TabContent 的背景色属性为蓝色
  }
  .barWidth('100%')                      //设置包含 tabBar 的“导航部分”的宽度
  .barHeight(60)                         //设置包含 tabBar 的“导航部分”的高度
  .width('100%')                         //设置 Tabs 整个组件的宽度
  .height('100%')                        //设置 Tabs 整个组件的高度
  .backgroundColor(0xF5F5F5)             //设置 Tabs 整个组件的背景颜色
}.width('100%').height('100%')           //设置 Column 组件的宽度和长度
```

上述代码的运行效果如图 3-63 所示。

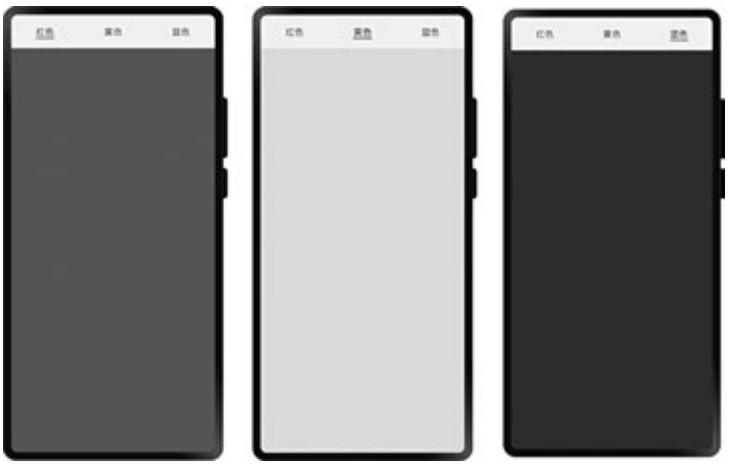


图 3-63　Tabs 应用示例总效果图

有时我们需要使用. barMode(value: BarMode)属性来设置 TabBar 的布局模式。这个属性的输入参数是一个枚举变量 BarMode。BarMode 有两个属性枚举量，一个是 BarMode. Fixed(默认)，另一个是 BarMode. Scrollable。

BarMode.Fixed：所有 TabBar 平分 barWidth 宽度(纵向时平分 barHeight 高度)，页签不可滚动。

BarMode.Scrollable：每一个 TabBar 均使用实际布局宽度，超过总长度(横向时为 barWidth，纵向时为 barHeight)后可滑动。

因为 Tabs 的布局模式默认是 Fixed 的，所以 Tabs 的页签是不可滑动的。当页签比较多的时候，可能会导致页签显示不全，将布局模式设置为 Scrollable 的话，可以实现页签的滚动。

3.2.5 Flex 容器组件

Flex 容器组件是以弹性方式布局子组件的容器组件，可以包含子组件。

```
Flex(value?: FlexOptions)
```

其中 FlexOptions 对象包含的参数如表 3-1 所示。

表 3-1 FlexOptions 对象中的参数

参数名	参数类型	必填	默认值	参数描述
direction	FlexDirection	否	FlexDirection.Row	子组件在 Flex 容器上排列的方向，即主轴的方向
wrap	FlexWrap	否	FlexWrap.NoWrap	Flex 容器是单行/列还是多行/列排列
justifyContent	FlexAlign	否	FlexAlign.Start	子组件在 Flex 容器主轴上的对齐格式
alignItems	ItemAlign	否	ItemAlign.Start	子组件在 Flex 容器交叉轴上的对齐格式
alignContent	FlexAlign	否	FlexAlign.Start	交叉轴中有额外的空间时，多行内容的对齐方式。仅在 wrap 为 Wrap 或 WrapReverse 下生效

其中，FlexDirection 枚举类型包括 FlexDirection.Row、FlexDirection.RowReverse、FlexDirection.Column、FlexDirection.ColumnReverse；FlexWrap 枚举类型包括 FlexWrap.NoWrap、FlexWrap.Wrap、FlexWrap.WrapReverse；ItemAlign 枚举类型包括 ItemAlign.Auto、ItemAlign.Start、ItemAlign.Center、ItemAlign.End、ItemAlign.Stretch、ItemAlign.Baseline；FlexAlign 枚举类型包括 FlexAlign.Start、FlexAlign.Center、FlexAlign.End、FlexAlign.SpaceBetween、FlexAlign.SpaceAround、FlexAlign.SpaceEvenly。

通过下面一段代码，我们来看一下所有的设置参数对应的效果。

```
@Styles function global_s(){.height(50).width(80).border({width:1})}
@Entry
@Component
struct xxx {
  @Styles local_s(){.height(50).width(80).border({width:1})}
  build() {
    Column() {
      Flex({ XXX }) {
        Text('1').global_s(); Text('2').local_s();Text('3').local_s(); Text('4').local_s()
      } } } }
```

上面使用了@Style 来定义组件的重复样式，@Styles 目前仅支持通用属性和通用事件的设置。如果在组件内定义不需要关键字 function，全局定义时需要添加 function 关键字，比如上面被@Styles 修饰的 global_s()就是全局定义，则 local_s()在组件内定义。目前被

@Style 修饰的函数不能传入参数。

当上面 XXX 部分为 direction 参数时，比如 direction：FlexDirection. Row。我们来看一下其中 Row、RowReverse、Column、ColumnReverse 的区别，如图 3-64 所示。

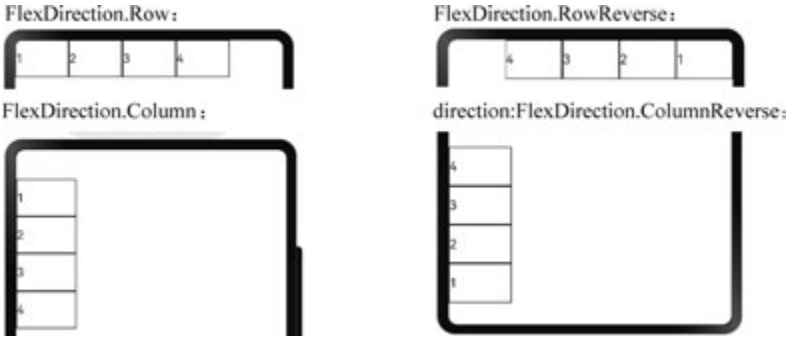


图 3-64　Row、RowReverse、Column、ColumnReverse 的区别

当上面 XXX 部分为 wrap 参数时，比如 wrap：FlexWrap. NoWrap。将代码中的 . width(80)都改为. width(120)后，下面我们看一下其中 NoWrap、Wrap、WrapReverse 的区别（图 3-65）。

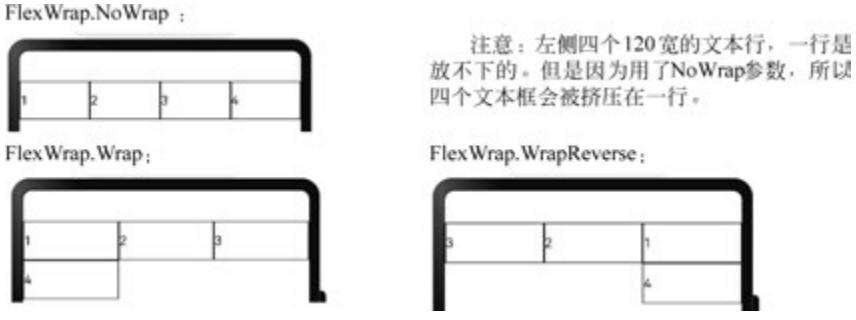


图 3-65　NoWrap、Wrap、WrapReverse 的区别

当上面 XXX 部分为 justifyContent 参数时，比如 justifyContent：FlexAlign. Start。将代码中的. width(80)仍然保持 80，下面我们看一下其中 Start、Center、End、SpaceBetween、SpaceAround、SpaceEvenly 的区别（图 3-66）。

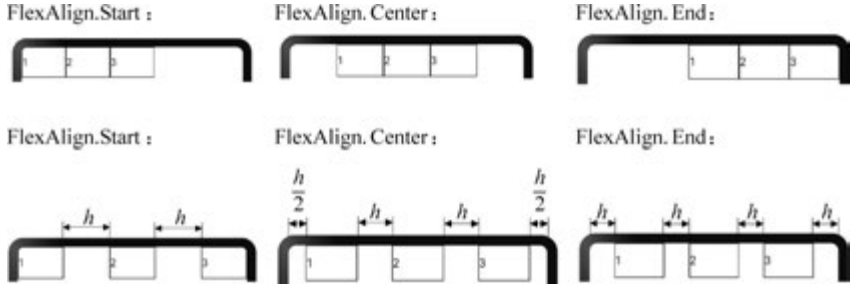


图 3-66　Start、Center、End、SpaceBetween、SpaceAround、SpaceEvenly 的区别

这里就列出这几个参数的作用，如果读者对其他的参数也感兴趣，可自行测试感受。

3.2.6 Stack 层叠容器组件

层叠布局(StackLayout)用于在屏幕上预留一块区域来显示组件中的元素,提供元素可以重叠的布局。层叠布局通过 Stack 容器组件实现位置的固定定位与层叠,容器中的子元素(子组件)依次入栈,后一个子元素覆盖前一个子元素,子元素可以叠加,也可以设置位置。层叠布局具有较强的页面层叠、位置定位能力,其使用场景有广告、卡片层叠效果等。

Stack 容器组件有如下特点。

- 按照子元素的先后顺序层叠,后面的元素覆盖在前面的元素上面。
- 默认的对齐方式是垂直水平居中。
- 目前版本(Beta 2 版本)需要设定宽度和高度属性,否则预览器无法渲染。

1. 对齐方式

我们通过下面一段代码来查看对齐方式的作用,下面代码中放了 3 个 Column 块,尺寸一个比一个小,如图 3-67 所示。

```
@Entry
@Component
struct stackDemo {
  build(){
    Stack(){
      Column(){}.width("100 % ").height(300)      //块 1
      .backgroundColor(Color.Red).borderWidth(1)
      Column(){}.width("80 % ").height(200)       //块 2
      .backgroundColor(Color.Orange).borderWidth(1)
      Column(){}.width("60 % ").height(100)       //块 3
      .backgroundColor(Color.Pink).borderWidth(1)
    }.width('100 % ').height(400) ① .
  }
}
```

图 3-67 三个 Column 块的默认显示

从图 3-67 可以看出,后面创建的 Column 块会覆盖在前面的元素上面,默认情况下,Stack 内部的元素,是上下左右居中对齐的。我们可以通过对 Stack 设置 alignContent 属性来设置 Stack 内部元素的对齐方式。在上面代码中的①处添加 alignContent 属性设置,比如将①处写上. alignContent(Alignment. TopStart)。Alignment 枚举类型有以下几种方式:TopStart 是顶部左对齐;Top 是顶部居中对齐;TopEnd 是顶部右侧对齐;Start 是中间部分左对齐;Center 是中间部分居中对齐(默认值);End 是中间部分右对齐;BottomStart 是底部左对齐;Bottom 是底部居中对齐;BottomEnd 是底部右侧对齐。效果如图 3-68 所示。

2. Z 序控制

Stack 容器组件中,默认的子元素的层叠顺序是根据元素引用的先后顺序。除了通过

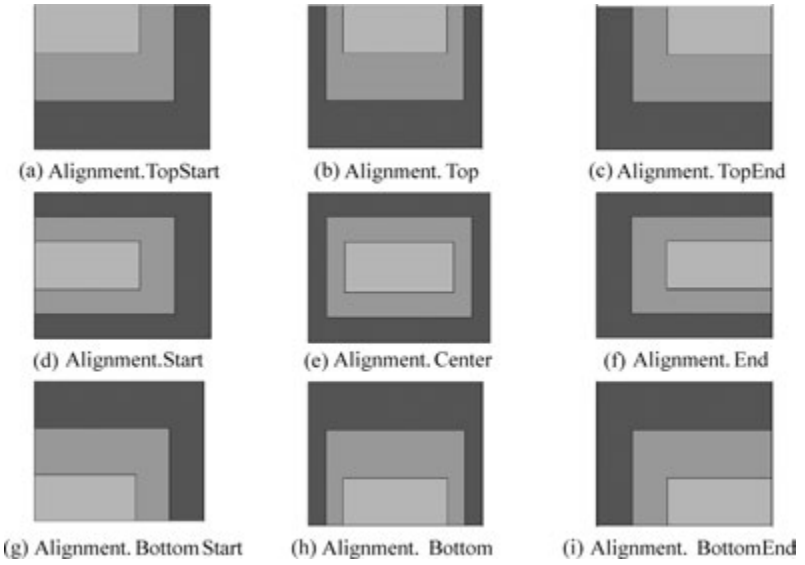


图 3-68　Stack 内部元素的对齐方式

先后顺序来控制层叠顺序之外，还可以通过 Z 序控制的 zIndex 属性来控制。zIndex 值越大，显示层级越高，越靠上。比如下面这段代码，我们通过设置 zIndex 属性，就可以让 Column 块 1 在最上层，Column 块 2 在中间，Column 块 3 在最下层。

```
@Entry
@Component
struct stackDemo {
  build(){
    Stack(){
      Column(){}.width("100 % ").height(300)       //块 1
      .backgroundColor(Color.Red).borderWidth(1).zIndex(3)
      Column(){}.width("80 % ").height(200)        //块 2
      .backgroundColor(Color.Orange).borderWidth(1) .zIndex(2)
      Column(){}.width("60 % ").height(100)        //块 3
      .backgroundColor(Color.Pink).borderWidth(1) .zIndex(1)
    }.width('100 % ').height(400)
  }
}
```

3.3　媒体组件

3.3.1　Video 组件

Video 组件用于播放视频文件并控制其播放状态，常用于短视频和应用内部视频的列表页面。可以加载本地视频和网络视频。当视频完整出现时会自动播放，用户单击视频区域则会暂停播放，同时显示播放进度条，通过拖动播放进度条指定视频播放到具体位置。Video 组件不支持预览模式，远程模拟视频不能播放，只能使用远程真机进行播放。

1. 组件接口

```
Video(value: VideoOptions)
```

VideoOptions 对象包含参数 src、currentProgressRate、previewUri、controller。

其中，src 指定视频播放源的路径。视频的数据源，支持本地视频和网络视频，视频支持的格式是 MP4、MKV、TS。访问本地视频时，需要把播放的视频文件放到 rawfile 文件下，通过 $ rawfile('xxx')的形式引用视频文件。还可以以 string 格式加载网络视频，加载网络视频时需要在 module. json5 文件中申请网络权限。申请时需要将下面的代码块放到 module. json5 文件的第 2 行“"module": { ”的下面一行。

```
"requestPermissions": [
  {
    "name":"ohos.permission.INTERNET"
  },
],
```

currentProgressRate 用于设置视频播放倍速，视频播放倍速可以使用 number 的形式，number 仅支持 0.75，1.0，1.25，1.75，2.0，默认值是 1.0。还可以使用 PlaybackSpeed 枚举类型：Speed_Forward_1_00_X、Speed_Forward_0_75_X、Speed_Forward_1_00_X、Speed_Forward_1_25_X、Speed_Forward_1_75_X、Speed_Forward_2_00_X。

previewUri 指定视频未播放时的预览图片路径，一般需要将图片放入 entry\src\resources\base\media 文件夹下面，然后使用 $ r('app. media. xxx')的资源调用的形式将 xxx. png(后缀可以是 jpg、gif 等)调用进来。

controller 设置视频控制器，用于自定义控制视频。一个 VideoController 对象可以控制一个或多个 video。使用时需要先导入对象：

```
let controller: VideoController = new VideoController()
```

然后在 Video 组件中设置 controller 属性：

```
Video({
  …
  controller: this.controller
})
```

最后，在需要控制的地方使用定义的 controller 进行控制：

```
Row() {
  Button('start').onClick(() => {
    this.controller.start()                    //开始播放
  }).margin(2)
  Button('pause').onClick(() => {
    this.controller.pause()                    //暂停播放
   }).margin(2)
  Button('stop').onClick(() => {
    this.controller.stop()                     //结束播放
  }).margin(2)
}
```

2. 属性设置

Video 组件还有 muted、autoPlay、controls、loop、objectFit 等属性设置。

muted 属性的形式是：muted(value：boolean)。用于设置是否静音。参数是布尔变量，true 表示静音，false 表示不静音，默认是 false。

autoPlay 属性的形式是：autoPlay(value：boolean)。用于设置是否自动播放。参数是布尔变量，true 表示自动播放，false 表示不自动播放，默认是 false。

controls 属性的形式是：controls(value：boolean)。用于设置控制视频播放的控制栏是否显示。参数是布尔变量，true 表示显示播放控制栏，false 表示不显示播放控制栏，默认是 true。

loop 属性的形式是：loop(value：boolean)。用于设置是否单个视频循环播放。参数是布尔变量，true 表示循环播放，false 表示不循环播放，默认是 false。

objectFit 属性的形式是：objectFit(value：ImageFit)。用于设置视频显示模式。参数是一个枚举类型 ImageFit，这个枚举类型代表的意义也和前边讲过的 Image 组件相同。其中，ImageFit.Contain 表示保持宽高比进行缩小或者放大，使得视频画面完全显示在显示边界内。ImageFit.Cover 表示保持宽高比进行缩小或者放大，使得视频画面两边都大于或等于显示边界。ImageFit.Aut 表示视频画面会根据其自身尺寸和组件的尺寸进行适当缩放，以在保持比例的同时填充视图。ImageFit.Fill 表示不保持宽高比进行放大缩小，使得视频画面充满显示边界。ImageFit.ScaleDown 表示保持宽高比显示，视频画面缩小或者保持不变。ImageFit.None 表示保持原有尺寸显示。

3. 事件

Video 组件回调事件主要为播放开始 onStart、暂停结束、播放失败、播放停止、视频准备和操作进度条等事件，除此之外，Video 组件也支持通用事件的调用，如单击、触摸等事件的调用。具体使用方式请参考如下代码：

```
@Entry
@Component
struct VideoPlayer{
  private controller:VideoController|undefined;
  private preview:Resource = $r('app.media.preview');
  private innerResource:Resource = $rawfile('videoTest.mp4');
  @State msg:string = ""
  build(){
    Column() {
      Text("视频事件信息:" + this.msg)
      Video({src:this.innerResource,previewUri: this.preview,controller: this.controller})
        .onStart(() => {this.msg = "Video start" })                    //播放开始事件
        .onPause(() =>{this.msg = "Video pause"})                      //播放暂停事件
        .onStop(() => {this.msg = "Video stoped."})                    //播放停止事件
        .onFinish(() =>{this.msg = "Video finish."})                   //播放结束事件
        .onUpdate((event) => {this.msg = "Video update."})             //更新事件回调
        .onPrepared((event) => {this.msg = "Video prepared."})         //准备事件回调
        .onError(() => { this.msg = "Video error."})                   //失败事件回调
    }
  }
}
```

上面例子中我们使用了一个本地视频 videoTest.mp4，使用了一个预览图片 preview.

png，所以需要提前将 videoTest. mp4 放到 entry\src\main\resources\rawfile 文件夹下，将 preview. png 放到 entry\src\main\resources\base\media 文件夹下。另外，由于 Video 组件不能用预览器观看，只能用本地模拟器、远程模拟器、远程真机或者真机查看效果，此处我们使用本地模拟器。

Video 事件的运行效果如图 3-69 所示：当视频准备好后，就会显示预览图片，并调用 onPrepared 事件；当开始播放后，就会调用 onStart 事件；当暂停时就会触发 onPause 事件；当播放时就会触发 onUpdate 事件；当播放结束时就会触发结束 onFinish 事件。

图 3-69　Video 事件的运行效果

3.4　绘图组件

绘制组件主要有 Line、Polyline、Circle、Ellipse、Polygon、Path、Rect、Shape 等。

3.4.1　Line

Line 组件是直线绘制组件。

1. 接口

```
Line(value?: {width?: string | number, height?: string | number})
```

接口中的 width 和 height 指的就是 Line 绘制区域的宽度和高度。

2. 属性

经常用到的属性有 stroke、startPoint、endPoint、stokeWidth、strokeOpacity、antiAlias 等。

stroke 属性非常重要，目前版本如果不设置这个属性，是看不到线的。因此必须设置 stroke 属性。从 stroke(value：ResourceColor)可以看出，stroke 的参数是一个颜色值。从

前面的学习我们知道可以是资源的形式、可以是十六进制的形式、也可以是字符串的形式。这里主要采用字符串的形式，比如红色使用"#ff0000"。

startPoint 属性设置直线起点坐标点(相对坐标)，相对坐标指的就是上面的 Line 的绘制区域。这个起点指的是在这个区域中的相对起点位置。从参数的形式 startPoint(value: Array<any>)中可以看出，参数是一个数组值，需要这样定义：.startPoint([10,10])，其中的 10 的单位是 vp。如果不设置此项属性，则默认值是[0,0]。

endPoint 属性设置直线终点坐标点(相对坐标)，相对坐标指的也是上面的 Line 的绘制区域。这个终点指的也是在这个区域中的相对终点位置。从参数的形式 endPoint(value: Array<any>)中可以看出，参数是一个数组值，需要这样定义：.endPoint([100,100])，其中的 100 的单位是 vp。如果不设置此项属性，则默认值也是[0,0]。

strokeWidth 属性用于设置线的宽度：strokeWidth(value:Length)。

strokeOpacity 属性设置线的透明度：strokeOpacity(value: number|string|Resource)，也是可以使用 number、string、Resource 的形式。该属性的取值范围是[0.0,1.0]，0.0 是全透明，1 是全不透明，若给定值小于 0.0，则取值为 0.0；若给定值大于 1.0，则取值为 1.0，其余异常值按 1.0 处理。

antiAlias 属性设置是否开启抗锯齿效果，形式是 antiAlias(value: boolean)，当参数是 true 时绘制出来的线就没有锯齿效果，就更清晰，默认就是 true。所以不设置此项属性的时候，默认抗锯齿效果是打开的。

3. 举例

```
@Entry
@Component
struct LineExample {
  build(){
    Column({space:10}){
      Line()
        .width(200).height(150)                    //设置 Line 的绘制区域宽 200 高 150
        .startPoint([0,0])                         //起止点坐标均是相对于 Line
        .endPoint([50,100])                        //组件本身绘制区域的坐标
        .stroke(Color.Black)                       //设置线的颜色
        .backgroundColor('#F5F5F5')                //设置绘制区域 1 的背景色
      Line()
        .width(200).height(150)                    //设置绘制区域宽 200 高 150
        .startPoint([50,50]).endPoint([150,150])//设置起始点坐标
        .strokeWidth(5)                            //设置线宽为 5
        .stroke(Color.Orange)                      //设置线的颜色
        .backgroundColor('#F5F5F5')                //设置绘制区域 2 的背景色
      Line()
        .width(50).height(50)                      //设置绘制区域宽 50 高 150
        .startPoint([0,0])       //坐标点超出 Line 组件的宽高范围时线条会画出组件绘制区域
        .endPoint([100,100])
        .stroke(Color.Black)                       //设置折线的颜色
        .strokeWidth(3)                            //设置线宽为 3
        .backgroundColor('#F5F5F5')                //设置绘制区域 3 的背景色
    }.alignItems(HorizontalAlign.Start)            //让三个区域左对齐
  }
}
```

上述代码的运行效果如图 3-70 所示。

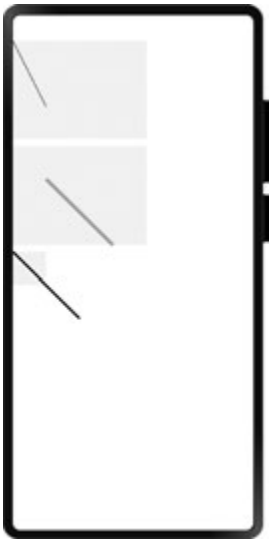


图 3-70　绘图组件应用示例

3.4.2　Polyline

Polyline 组件用于绘制折线。

1. 接口

```
Polyline(value?: {width?: string | number, height?: string | number})
```

接口中的 width 和 height 指的就是 Polyline 的绘制区域的宽度和高度。

2. 属性

经常用到的属性有 stroke、points、strokeWidth、strokefill、strokeOpacity、fill、antiAlias 等。

stroke 属性非常重要，目前版本如果不设置这个属性，则看不到边框线条。因此必须设置 stroke 属性。

points 属性用于设置折线经过的坐标点列表，形式为 points(value：Array<any>)。参数为数组，可以通过.points([[0,0],[20,60],[100,100]])的形式来设置一个经过坐标为(0,0)、(20,60)、(100,100)三个点的折线。

strokeWidth 属性用于设置折线的宽度：strokeWidth(value：Length)。

strokeOpacity 属性设置折线的透明度：strokeOpacity(value：number | string | Resource)，也可以使用 number、string、Resource 的形式。该属性的取值范围是[0.0,1.0]，0.0 是全透明，1.0 是全不透明，若给定值小于 0.0，则取值为 0.0；若给定值大于 1.0，则取值为 1.0，其余异常值按 1.0 处理。

fill 属性用于设置填充区域的颜色，形式为 fill(value：ResourceColor)。如果不设置此属性，则填充颜色默认是 Color.Black。

fillOpacity 设置填充区域的透明度，形式为 fillOpacity(value:number|string|Resource)。取值范围是[0.0,1.0]，若给定值小于 0.0，则取值为 0.0；若给定值大于 1.0，则取值为 1.0，其余异常值按 1.0 处理。

3. 举例

下面这个例子是在 100×100 的矩形框中绘制一段折线，起点为(0,0)，经过(100,150)，到达终点(300,300)。

```
@Entry
@Component
struct LineExample{
  build(){
    Column({space:10}){
      Polyline({width:400,height:400})            //设置绘制区域
        .points([[0,0],[100,150],[300,300]])      //设置折线点
        .fillOpacity(0)                           //设置填充透明度,0 为全透明
        .stroke(Color.Blue)                       //设置折线颜色为蓝色
        .strokeWidth(3)                           //折线宽度为 3
        .borderWidth(1)                           //设置绘制区域的边框线宽为 1
        .backgroundColor('#d2d2d2')               //设置绘制区域的背景色
    }
  }
}
```

上述代码的运行效果如图 3-71 所示。

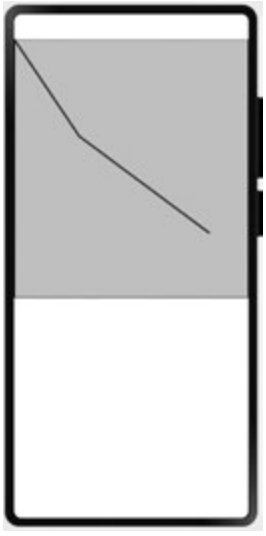


图 3-71　Polyline 组件应用示例

3.4.3　Circle

Circle 组件是一种用于绘制圆形的组件。

1. 接口

```
Circle(value?: CircleOptions)
```

参数中的 CircleOptions 用于设置圆形的尺寸，包括 width 和 height 属性。使用时，如果设置一个宽和高都为 150 的圆形(即圆的直径为 150)，可以设置参数如下：

```
Circle({ width: 150, height: 150 }).
```

圆形的尺寸也可以用通用的属性 width 和 height 来设置，一旦设置了 width 和 height 的属性，CircleOptions 就失效了，比如：

```
Circle({width:400,height:400})                    //用接口的方式设置圆的直径为 400
    .width(150).height(150)                       //用属性的方式设置圆的直径为 150
```

上面这段代码最终绘制出来的圆的半径为 150。

2. 属性

常用的属性有 fill、fillOpacity、stroke、strokeOpacity、strokeWidth、antiAlias。这些属性的意义都和前面的 Polyline 组件的属性相似。fill 用于设置填充区域的颜色，默认为黑色；fillOpacity 用于设置填充区域的透明度；stroke 用于设置圆形边线的颜色；strokeOpacity 用于设置圆形边线的透明度；strokeWidth 用于设置圆形边线的宽度。

3. 举例

下面绘制两个圆。绘制的第一个圆直径为 150，不设置任何属性；绘制的第二个圆的直径也为 150，填充透明度设为 0，边框为红色。

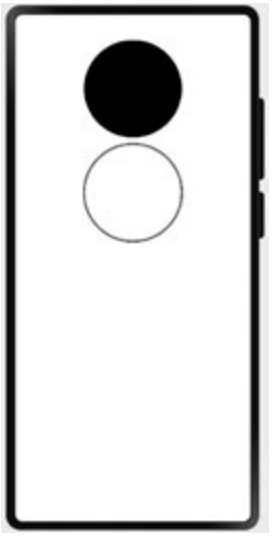


图 3-72　两个圆的显示效果

```
@Entry
@Component
struct CircleExample{
  build(){
    Column({space:10}){
      Circle({width:150,height:150})      //绘制第一个圆的直径为 150
      Circle()                            //绘制第二个圆
        .width(150)                       //圆的宽度为 150
        .height(150)                      //圆的高度为 150
        .fillOpacity(0)                   //填充设为全透明
        .strokeWidth(3)                   //设置圆的线宽为 3
        .stroke(Color.Red)                //设置线的颜色为红色
    }.width('100%')
  }
}
```

由于第一个圆没有设置任何属性，fill 属性默认是填充黑色，所以第一个圆显示的填充色是黑色。第二个圆填充的透明度设为全透明了，所以第二个圆没有填充色。两个圆的显示效果如图 3-72 所示。

3.4.4　Ellipse

Ellipse 组件是一个绘制椭圆的组件，Ellipse 组件和 Circle 组件基本相同，区别是 Circle 组件的 width 和 height 必须相同，而 Ellipse 组件的 width 和 height 不能相同。

下面这个例子绘制了两个椭圆，一个宽为 150 高为 80，另一个宽为 150 高为 100。

```
@Entry
@Component
struct EllipseExample{
  build(){
    Column({space:10}){
      Ellipse({width:150,height:80})     //绘制第一个椭圆
      Ellipse()                          //绘制第二个椭圆
        .width(150)
        .height(100)
        .fillOpacity(0)                  //填充设为全透明
        .stroke(Color.Blue)              //线的颜色设为蓝色
        .strokeWidth(3)                  //线宽设为 3
    }.width('100%')
  }
}
```

上述代码的运行效果如图 3-73 所示。

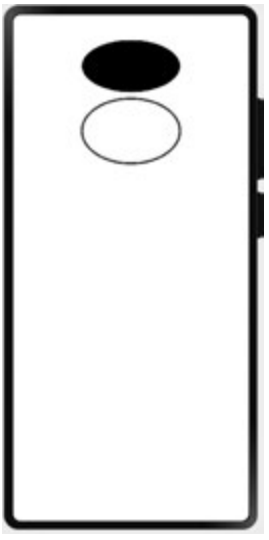


图 3-73　两个椭圆的显示效果

3.4.5　Polygon

Polygon 组件是用于绘制多边形的绘制组件。

1. 接口

```
Polygon(value?: {width?: string | number, height?: string | number})
```

接口参数也是需要设置绘制多边形的绘制区域的 width 和 height。

2. 属性

Polygon 的属性和 Polyline 的属性非常相似，也都是有 points、fill、fillOpacity、stroke、strokeOpacity、strokeWidth、antiAlias 等。此处不展开讲解了。

3. 举例

下面的例子中绘制了三个多边形。第一个是在 200×200 的矩形区域中绘制一个三角形，起点为(0,0)，经过(50,100)，终点为(100,0)；第二个是在 100×100 的绘制区域中绘制一个四边形，起点为(0,0)，经过(0,100)和(100,100)，终点为(100,0)；第三个是在 100×100 的绘制区域中绘制一个五边形，起点为(50,0)，依次经过(0,50)、(20,100)和(80,100)，终点为(100,50)。

```
@Entry
@Component
struct PolygonExample {
  build(){
    Column({space: 10}){
      Polygon({width:200, height:200})                          //绘制多边形 1
        .points([[0,0],[50,100],[100,0]])                       //多边形点坐标
        .fill(Color.Green)                                      //设置填充色
        .backgroundColor('#8be2db')                             //设置绘制区域背景色
      Polygon().width(100).height(100)                          //绘制多边形 2
        .points([[0,0],[0,100],[100,100],[100,0]])              //多边形点坐标
        .fillOpacity(0)                                         //设置填充色为透明
        .strokeWidth(5)                                         //设置多边形边的线宽
        .stroke(Color.Blue)                                     //设置多边形线的颜色
      Polygon().width(100).height(100)                          //绘制多边形 2
        .points([[50,0],[0,50],[20,100],[80,100],[100,50]]) //点坐标
        .fill(Color.Red)                                        //设置填充色
        .fillOpacity(0.6)                                       //设置填充色的透明度
    }.width('100%').margin({ top: 10 })
  }
}
```

上述代码的运行效果如图 3-74 所示。

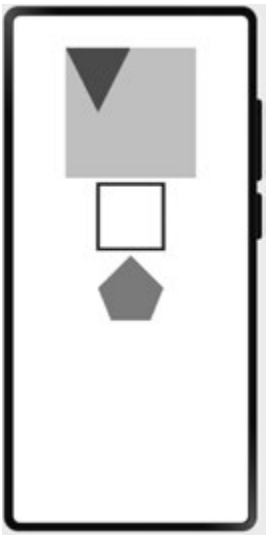


图 3-74　多边形的显示效果

3.4.6　Path

Path 组件是路径绘制组件，根据绘制路径生成封闭的自定义形状。

1. 接口

```
Path(value?:{width?:number| string;height?:number|string;commands?:string})
```

接口参数也是需要设置绘制多边形的绘制区域的 width 和 height。

2. 属性

常用的属性除了和之前相同的 fill、fillOpacity、stroke、strokeOpacity、strokeWidth、antiAlias 等之外，还有一个主要的属性是 commands。从它的形式 commands(value: string)可以看出，参数只有一个 string 字符串，这个 string 字符串是用于设置路径绘制的命令字符串，单位为 px。

表 3-2　commands 绘制命令

命令	名　　称	参数	说　　明
M	moveto	(x y)	在给定的 (x, y) 坐标处开始一个新的子路径。例如，M 0 0 表示将点(0, 0)作为新子路径的起始点
L	lineto	(x y)	从当前点到给定的 (x, y) 坐标画一条线，该坐标成为新的当前点。例如，L 50 50 表示绘制当前点到点(50, 50)的直线，并将点(50, 50)作为新子路径的起始点
H	horizontal lineto	x	从当前点绘制一条水平线，等效于将 y 坐标指定为 0 的 L 命令。例如，H 50 表示绘制当前点到点(50, 0)的直线，并将点(50, 0)作为新子路径的起始点
V	vertical lineto	y	从当前点绘制一条垂直线，等效于将 x 坐标指定为 0 的 L 命令。例如，V 50 表示绘制当前点到点(0, 50)的直线，并将点(0, 50)作为新子路径的起始点
C	curveto	(x1 y1 x2 y2 x y)	使用(x1,y1)作为曲线起点的控制点，(x2,y2)作为曲线终点的控制点，从当前点到(x,y)绘制三次贝塞尔曲线。例如，C100 100 250 100 250 200 表示绘制当前点到点(250, 200)的三次贝塞尔曲线，并将点(250,200)作为新子路径的起始点
S	smooth curveto	(x2 y2 x y)	(x2,y2)作为曲线终点的控制点，从当前点到(x,y)绘制三次贝塞尔曲线。若前一个命令是 C 或 S，则起点控制点是上一个命令的终点控制点相对于起点的映射。例如，C100 100 250 100 250 200 S400 300 400 200 第二段贝塞尔曲线的起点控制点为(250,300)。如果没有前一个命令或者前一个命令不是 C 或 S，则第一个控制点与当前点重合
Q	quadratic Belzier curve	(x1 y1 x y)	使用(x1,y1)作为控制点，从当前点到(x,y)绘制二次贝塞尔曲线。例如，Q400 50 600 300 表示绘制当前点到点(600, 300)的二次贝塞尔曲线，并将点(600,300)作为新子路径的起始点

续表

命令	名　　称	参数	说　　明
T	smooth quadratic Belzier curveto	(x y)	从当前点到(x,y)绘制二次贝塞尔曲线。若前一个命令是 Q 或 T,则控制点是上一个命令的终点控制点相对于起点的映射。例如,Q400 50 600 300 T1000 300 第二段贝塞尔曲线的控制点为(800,350)。如果没有前一个命令或者前一个命令不是 Q 或 T,则第一个控制点与当前点重合
Z	closepath	none	通过将当前路径连接回当前子路径的初始点来关闭当前子路径

例如,commands('M0 20 L50 50 L50 100 Z')定义了一个三角形,起始于位置(0,20),接着绘制点(0,20)到点(50,50)的直线,再绘制点(50,50)到点(50,100)的直线,最后绘制点(50,100)到点(0,20)的直线关闭路径,形成封闭三角形。

3. 举例

使用 Path 组件绘制 3 个图形:一个三角形、一个正方形、一个五边形。

```
@Entry
@Component
struct pathDemo {
  build() {
    Flex({justifyContent: FlexAlign.SpaceBetween}) {
      Path()                                                          //三角形
        .width('210px').height('310px')                               //设置绘制区域尺寸
        .commands('M100 0 L200 240 L0 240 Z')                         //命令绘制
        .fillOpacity(0)                                               //设置填充透明
        .stroke(Color.Black).strokeWidth(3)                           //设置线的颜色及线宽
      Path()                                                          //正方形
        .width('210px').height('310px')                               //设置绘制区域尺寸
        .commands('M0 0 H200 V200 H0 Z')                              //命令绘制
        .fillOpacity(0)                                               //设置填充透明
        .stroke(Color.Black).strokeWidth(3)                           //设置线的颜色及线宽
      Path()                                                          //五边形
        .width('210px').height('310px')                               //设置绘制区域尺寸
        .commands('M100 0 L0 100 L50 200 L150 200 L200 100 Z')        //命令绘制
        .fillOpacity(0)                                               //设置填充透明
        .stroke(Color.Black).strokeWidth(3)                           //设置线的颜色及线宽
    }.width('100%')
  }
}
```

上述代码的运行效果如图 3-75 所示。

图 3-75　自定义形状的显示效果

3.4.7　Rect

Rect 组件是一个矩形绘制组件。

1. 接口

```
Rect(value?: {width?: string | number,height?: string | number,radius?: string | number | Array
<string | number>} |
```

其中，width 是宽度，默认值是 0；height 是高度，默认值是 0；radius 是圆角半径，支持分别设置四个角的圆角度数，所以可以是数值（四个角的圆角半径相同）也可以是数值数组（四个角的圆角半径分别进行设置），该属性和 radiusWidth/radiusHeight 属性效果类似，在组合使用时优先于 radiusWidth/radiusHeight 生效，默认值是 0；radiusWidth 是圆角宽度，默认值是 0；radiusHeight 是圆角高度，默认值是 0。

2. 属性

Rect 的属性除了之前的 fill、fillOpacity、stroke、strokeOpacity、strokeWidth、antiAlias，多了 radiusWidth、radiusHeight、radius 等属性。

radiusWidth 属性用于设置圆角的宽度，形式是 radiusWidth(value:number|string)。如果只设置 radiusWidth，不设置 radiusHeight，则认为 radiusHeight 的值和 radiusWidth 一致。

radiusHeight 属性用于设置圆角的高度，形式是 radiusHeight(value:number|string)。

radius 设置圆角半径大小，形式是 radius(value:number|string|Array<string|number>)。通过它也可以设置矩形的圆角半径大小，可以通过一个值来设置所有圆角的大小，也可以通过一个数组分别对四个圆角的半径进行设置。

比如.radius(20)：表明矩形的四个圆角的 radiusWidth 和 radiusHeight 都是 20。.radius([[10,20],[20,30],[30,40],[40,50]])：表明图 3-76 中①位置的 radiusWidth=10，radiusHeight = 20；②位置的 radiusWidth = 20，radiusHeight = 30；③位置的 radiusWidth=30，radiusHeight=40；④位置的 radiusWidth=40，radiusHeight=50。

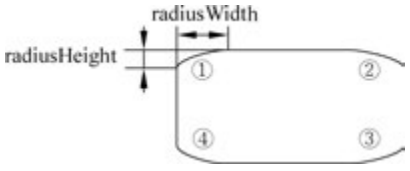


图 3-76　radiusWidth 和 radiusHeight 对应的位置

3. 举例

使用 radiusWidth 单独设置、radiusWidth 和 radiusHeight 配合设置、radius 设置等方式设置矩形的圆角。

```
@Entry
@Component
struct RectExample{
  build(){
```

```
    Column({space:10}){
      Rect({width:'90%',height:80})                    //绘制矩形 1
        .fill(Color.Red)                               //设置填充为红色
      Rect()                                           //绘制矩形 2
        .width('90%').height(80)                       //用属性方式设置宽高
        .fillOpacity(0)                                //设置填充色透明
        .stroke(Color.Red).strokeWidth(3)              //设置线的颜色和宽度
        .radiusWidth(20)                               //用 radiusWidth 设置四个圆角的宽高全为 20
      Rect({width:'90%',height:80})                    //绘制矩形 3
        .radiusHeight(20).radiusWidth(40)              //设置四个圆角的宽和高
        .fill(Color.Red)                               //设置填充色为红色
      Rect({width:'90%',height:80})                    //绘制矩形 4
        .radius(20)                                    //用 radius 方式设置四个圆角的宽高全为 20
        .fill(Color.Red)                               //设置填充色为红色
      Rect({ width:'90%',height: 80})                  //绘制矩形 5
        .radius([[10,20],[20,30],[30,40],[40,50]])  //用 radius 分别设置四个圆角的宽和高
        .fill(Color.Red)                               //设置填充色为红色
    }.width('100%')
  }
}
```

上述代码的运行效果如图 3-77 所示。

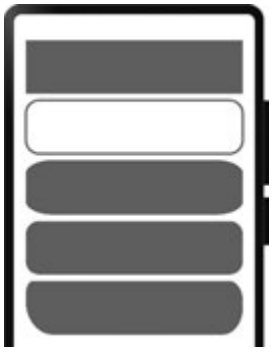


图 3-77　矩形的圆角显示效果

3.4.8　Shape

Shape 组件是绘制组件的父组件，可以在父组件中描述所有绘制组件均支持的通用属性。它可以包含 Rect、Path、Circle、Ellipse、Polyline、Polygon、Image、Text、Column、Row、Shape 子组件。

1. 接口

```
Shape(value?: PixelMap)
```

参数 PixelMap 是绘制目标，可将图形绘制在指定的 PixelMap 对象中，若未设置，则在当前绘制目标中进行绘制。

2. 属性

属性同样有 fill、fillOpacity、stroke、strokeOpacity、strokeWidth、antiAlias 等。另外还

有 viewPort 属性和 mesh 属性。

viewPort 属性用于设置形状的视口，形式如下：

```
viewPort(value:{x?:number|string;y?:number|string;width?:number|string;height?:number|string})
```

参数中有 x、y 代表窗口的位置，width 和 height 代表窗口的宽度和高度，示例如下。

例 1：

```
Shape(){Rect().width(300).height(100)}
  .width(300)
  .height(100)
  .viewPort({x:0,y:0,width:300,height:100})
  .backgroundColor('#ff72cfbc')
}
```

运行效果：

例 2：

```
Shape(){Rect().width(300).height(100)}
  .width(300)
  .height(100)
  .viewPort({x:0,y:0,width:600,height:200})
  .backgroundColor('#ff72cfbc')
}
```

运行效果：

例 3：

```
Shape(){Rect().width(300).height(100)}
  .width(300)
  .height(100)
  .viewPort({x:-100,y:-100,width:600,height:200})
  .backgroundColor('#ff72cfbc')
}
```

运行效果：

从上面的代码可以看出，上面例 1～例 3 的前三行代码都设置了一个 300vp×100vp 大小的 Shape 绘制区域，Shape 内部都绘制了一个 300vp×100vp 大小的 Rect 矩形组件。当设置 viewPort 属性后，就将 viewPort 设置的大小映射到了 Shape 的 300vp×100vp 区域内了。

比如，例 2 和例 3 中，viewPort 的宽度和高度分别为 600vp、200vp，那么原来 Shape 的大小区域 300vp×100vp 代表的就是 600vp×200vp 的大小了。

而 x 和 y 表示 viewPort 视窗的偏移，比如例 3 中，x 设置为－100vp，y 设置为－100vp，代表视窗向左偏移了 100vp，向上偏移了 100vp，所以例 3 中视觉看上去，矩形向右向下偏移了 100vp。

mesh 属性此处不介绍了，感兴趣的读者可查阅 API 手册。

举例：

```
@Entry
@Component
struct ShapeExample {
```

```
build() {
  Column({ space: 10 }) {
    Text('basic').fontSize(22)
    //在 Shape 的(-2, -2)点绘制一个 300×50 带边框的矩形,颜色为 0x317AF7,边框颜色为
    //黑色,边框宽度为 4,边框间隙为 20,向左偏移 10,线条两端样式为半圆,拐角样式为圆角,
    //抗锯齿(默认开启)
    //在 Shape 的(-2, 58)点绘制一个 300×50 带边框的椭圆,颜色为 0x317AF7,边框颜色为
    //黑色,边框宽度为 4,边框间隙为 20,向左偏移 10,线条两端样式为半圆,拐角样式为圆角,
    //抗锯齿(默认开启)
    //在 Shape 的(-2, 118)点绘制一个 300×10 的直线路径,颜色为 0x317AF7,边框颜色为
    //黑色,宽度为 4,间隙为 20,向左偏移 10,线条两端样式为半圆,拐角样式为圆角,抗锯齿
    //(默认开启)
    Shape() {
      Rect().width(300).height(50)
      Ellipse().width(300).height(50).offset({ x: 0, y: 60 })
      Path().width(300).height(10).commands('M0 0 L900 0').offset({ x: 0, y: 120 })
    }
    .width(350)
    .height(140)
    .viewPort({ x: -2, y: -2, width: 304, height: 130 })
    .fill(0x317AF7)
    .stroke(Color.Black)
    .strokeWidth(4)
    .strokeDashArray([20])
    .strokeDashOffset(10)
    .strokeLineCap(LineCapStyle.Round)
    .strokeLineJoin(LineJoinStyle.Round)
    .antiAlias(true)
    //分别在 Shape 的(0, 0)、(-5, -5)点绘制一个 300×50 带边框的矩形,可以看出之所以
    //将视口的起始位置坐标设为负值是因为绘制的起点默认为线宽的中点位置,因此要让边框
    //完全显示就需要让视口偏移半个线宽
    Shape() {
      Rect().width(300).height(50)
    }
    .width(350)
    .height(80)
    .viewPort({ x: 0, y: 0, width: 320, height: 70 })
    .fill(0x317AF7)
    .stroke(Color.Black)
    .strokeWidth(10)
    Shape() {
      Rect().width(300).height(50)
    }
    .width(350)
    .height(80)
    .viewPort({ x: -5, y: -5, width: 320, height: 70 })
    .fill(0x317AF7)
    .stroke(Color.Black)
    .strokeWidth(10)
    Text('path').fontSize(22)
    //在 Shape 的(0, -5)点绘制一条直线路径,颜色为 0xEE8443,线条宽度为 10,线条间隙为 20
    Shape() {
      Path().width(300).height(10).commands('M0 0 L900 0')
    }
```

```
      .width(350)
      .height(20)
      .viewPort({ x: 0, y: -5, width: 300, height: 20 })
      .stroke(0xEE8443)
      .strokeWidth(10)
      .strokeDashArray([20])
      //在 Shape 的(0, -5)点绘制一条直线路径,颜色为 0xEE8443,线条宽度为 10,线条间隙为
      //20,向左偏移 10
      Shape() {
        Path().width(300).height(10).commands('M0 0 L900 0')
      }
      .width(350)
      .height(20)
      .viewPort({ x: 0, y: -5, width: 300, height: 20 })
      .stroke(0xEE8443)
      .strokeWidth(10)
      .strokeDashArray([20])
      .strokeDashOffset(10)
      //在 Shape 的(0, -5)点绘制一条直线路径,颜色为 0xEE8443,线条宽度为 10,透明度为 0.5
      Shape() {
        Path().width(300).height(10).commands('M0 0 L900 0')
      }
      .width(350)
      .height(20)
      .viewPort({ x: 0, y: -5, width: 300, height: 20 })
      .stroke(0xEE8443)
      .strokeWidth(10)
      .strokeOpacity(0.5)
      //在 Shape 的(0, -5)点绘制一条直线路径,颜色为 0xEE8443,线条宽度为 10,线条间隙为
      //20,线条两端样式为半圆
      Shape() {
        Path().width(300).height(10).commands('M0 0 L900 0')
      }
      .width(350)
      .height(20)
      .viewPort({ x: 0, y: -5, width: 300, height: 20 })
      .stroke(0xEE8443)
      .strokeWidth(10)
      .strokeDashArray([20])
      .strokeLineCap(LineCapStyle.Round)
      //在 Shape 的(-20, -5)点绘制一条封闭路径,颜色为 0x317AF7,线条宽度为 10,边框颜色
      //为 0xEE8443,拐角样式为锐角(默认值)
      Shape(){
        Path().width(200).height(60).commands('M0 0 L400 0 L400 150 Z')
      }
      .width(300)
      .height(200)
      .viewPort({ x: -20, y: -5, width: 310, height: 90 })
      .fill(0x317AF7)
      .stroke(0xEE8443)
      .strokeWidth(10)
      .strokeLineJoin(LineJoinStyle.Miter)
      .strokeMiterLimit(5)
    }.width('100%').margin({ top: 15 })
  }
}
```

运行效果如图 3-78 所示。

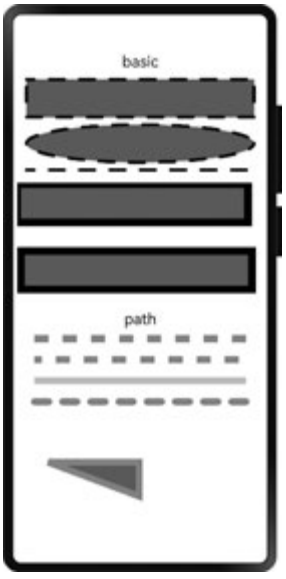


图 3-78　绘图组件总应用示例效果图

3.5　画布组件

Canvas 提供画布组件，用于自定义绘制图形，开发者使用 CanvasRenderingContext2D 对象和 OffscreenCanvasRenderingContext2D 对象在 Canvas 组件上进行绘制，绘制对象可以是基础形状、文本、图片等。

3.5.1　Canvas

Canvas 是一个可自定义 width、height 的矩形画布，画布左上角为坐标原点，以像素为单位，水平向右为 x 轴，垂直向下为 y 轴，画布内所有元素都基于原点进行定位。

Canvas 的接口如下：

```
Canvas(context?: CanvasRenderingContext2D | DrawingRenderingContext)
```

由接口可见，Canvas 的参数为 CanvasRenderingContext2D 和 DrawingRenderingContext。Canvas 只是画布，绘制需要使用 CanvasRenderingContext2D 和 DrawingRenderingContext。绘制一般在 onReady(event()=>{})事件中进行。Canvas 组件初始化完成时或者 Canvas 组件发生大小变化时调用该回调，当该事件被触发时画布被清空，该事件之后 Canvas 组件宽高确定且可获取，可使用 Canvas 相关 API 进行绘制。

3.5.2　CanvasRenderingContext2D 对象

使用 CanvasRenderingContext2D 对象时，需要使用 RenderingContextSettings 来配置 CanvasRenderingContext2D 对象的参数，包括是否开启抗锯齿。

```
@Entry
```

```
@Component
struct CanvasExample {
  settings:RenderingContextSettings = new RenderingContextSettings(true)
  //true 表示开启抗锯齿
  //创建 CanvasRenderingContext2D,在 Canvas 中调用 CanvasRenderingContext2D 来绘制
  context:CanvasRenderingContext2D = new CanvasRenderingContext2D(this.settings)
  build(){
   Column(){
   //在 Canvas 中调用 CanvasRenderingContext2D 对象
   Canvas(this.context)
    .width('100%').height('100%')
        .backgroundColor('#00ff90')
    .onReady(() =>{
      this.context.fillRect(0,30,100,100)
    })
  }
 }
}
```

上述代码的运行效果如图 3-79 所示。

图 3-79　CanvasRenderingContext2D 应用示例效果图

3.5.3　DrawingRenderingContext 对象

```
@Entry
@Component
struct CanvasExample {
//创建 CanvasRenderingContext2D,在 Canvas 中调用 CanvasRenderingContext2D 来绘制
  context:DrawingRenderingContext = new DrawingRenderingContext()
  build(){
   Column(){
   //在 Canvas 中调用 CanvasRenderingContext2D 对象
    Canvas(this.context)
      .width('100%').height('100%')
          .backgroundColor('#00ff90')
      .onReady(() =>{
        this.context.canvas.drawRect(vp2px(30),vp2px(30),vp2px(130),vp2px(130))
      })
    }
  }
}
```

上述代码的运行效果如图 3-80 所示。

图 3-80　DrawingRenderingContext 应用示例效果图

3.6 数据单位

前面的学习中，经常碰到 vp 这个单位，还经常碰到 px 这种类型的单位。这些都代表了什么意义呢？本节我们来学习一下单位相关的内容。

3.6.1 数据单位介绍

ArkUI 为开发者提供 4 种像素单位，框架采用 vp 为基准数据单位。其实从我们学习的过程中可以发现，还有 px、fp、lpx 等单位类型。它们代表的含义如下。

px：屏幕物理像素单位。

vp：屏幕密度相关像素，根据屏幕像素密度转换为屏幕物理像素，在 DevEco Studio 中，当某个数值不带单位时，其实默认的单位就是 vp。

fp：字体像素，与 vp 类似，适用于屏幕密度变化，随系统字体大小设置变化。

lpx：视窗逻辑像素单位，lpx 为实际屏幕宽度与逻辑宽度（通过 designWidth 配置）的比值，designWidth 默认值为 720。当 designWidth 为 720 时，在实际宽度为 1440 物理像素的屏幕上，1lpx 为 2px 大小。

vp 和 px 之间的关系如下：

$$\text{vp}=\frac{\text{px}}{\text{PPI}}\times 160$$

其中，PPI 指的是屏幕像素点密度（Pixels Per Inch），它是对角线上的像素点数除以屏幕尺寸（单位为英寸，1 英寸＝2.54 厘米），代表了每英寸中有多少个像素点。

$$\text{PPI}=\frac{\sqrt{w^2+h^2}}{\text{size}}$$

其中，w 是屏幕的横向分辨率，h 是屏幕的纵向分辨率，size 是屏幕的尺寸（单位是英寸）。

下面用一个屏幕大小为 5.5 英寸，屏幕分辨率为 1920×1080 的手机为例，来计算 vp 和 px 之间的关系。

我们先计算 PPI，$\text{PPI}=\frac{\sqrt{1080^2+1920^2}}{5.5}=400.53$

$$\text{vp}=\frac{\text{px}}{400.53}\times 160$$

通过上面计算可以得出：1px＝2.5vp。

3.6.2 单位之间的转化

其实，我们不需要自己去计算。因为 ArkTS 给我们提供了转化函数，如表 3-3 所示。

表 3-3 像素单位之间的转化函数

函　数	描　述
vp2px(value:number):number	将 vp 单位的数值转换为以 px 为单位的数值
px2vp(value:number):number	将 px 单位的数值转换为以 vp 为单位的数值

续表

函　　数	描　　述
fp2px(value:number):number	将 fp 单位的数值转换为以 px 为单位的数值
px2fp(value:number):number	将 px 单位的数值转换为以 fp 为单位的数值
lpx2px(value:number):number	将 lpx 单位的数值转换为以 px 为单位的数值
px2lpx(value: number):number	将 px 单位的数值转换为以 lpx 为单位的数值

使用的时候，找到对应的函数直接调用就可以了。

习题 3

一、填空题

1. HarmonyOS ArkUI 中的基础组件包括 Text、Image、TextInput、Button 等，其中用于输入单行文本的组件是_____，用于显示图片的组件是_____。

2. Text 组件的文本对齐方式通过_____属性设置，支持 Start、Center、End 三种枚举值。

3. Image 组件加载本地图片时，推荐使用_____类型的数据源，图片需放置在工程的_____目录下。

4. Button 组件的三种显示样式通过_____枚举类型设置，分别为 Capsule、Circle 和 Normal。

5. List 组件和 Grid 组件是常用的列表容器，其中_____组件用于线性列表布局，_____组件用于网格布局。

二、判断题

1. Image 组件的 objectFit 属性默认值为 Cover，保持宽高比缩放使图片完全覆盖显示边界。（　　）

2. List 组件的子组件必须是 ListItem，Grid 组件的子组件必须是 GridItem。（　　）

3. Video 组件支持预览模式，可在远程模拟器中直接播放本地视频。（　　）

4. Shape 组件可以包含 Rect、Path、Circle 等子组件，实现复杂图形的绘制。（　　）

三、单选题

1. 以下哪个属性用于设置 Text 组件的字体粗细？（　　）

A. fontStyle　　B. fontWeight　　C. fontFamily　　D. fontSize

2. Image 组件加载网络图片时，入口参数应使用哪种类型？（　　）

A. string　　B. PixelMap　　C. Resource　　D. File

3. Grid 组件的列布局通过哪个属性设置？（　　）

A. rowsTemplate　　B. columnsTemplate

C. columnsGap　　D. rowsGap

四、多选题

1. TextInput 组件支持的输入类型包括(　　)。

A. Normal　　B. Password　　C. Email　　D. Number

2. Button 组件的属性设置包括(　　)。

A. width 和 height　　B. backgroundColor 和 borderRadius

C. fontSize 和 fontWeight　　D. type 和 stateEffect

3. 下列哪些组件支持事件处理?(　　)

A. Button 的 onClick　　B. TextInput 的 onChange

C. Radio 的 onChange　　D. Video 的 onStart