第 1 章

初识鸿蒙HarmonyOS NEXT

1

本章是本书的首章，将首先介绍如何构建一个HarmonyOS NEXT开发环境，以便顺利演示本书中的开发范例；然后详细解析ArkTS语言在HarmonyOS开发中的应用，最后介绍一个简单的开发案例，使读者能够快速上手HarmonyOS开发。

1.1 DevEco Studio开发工具

本节首先介绍HarmonyOS NEXT开发工具DevEco Studio的下载与安装，然后介绍其基本功能和使用方法，最后，介绍ArkTS Stage模型的工程目录结构。

1.1.1 下载和安装DevEco Studio

HarmonyOS NEXT开发专属的IDE（集成开发系统）是HUAWEI DevEco Studio（简称DevEco Studio）。作为一款专为HarmonyOS应用及服务开发者设计的集成开发环境，DevEco Studio提供了全面的开发、调试和部署支持。

读者可以在Harmonyos官方网站免费下载和使用DevEco Studio。

DevEco Studio支持Windows系统和macOS系统，在开发应用/服务前，需要配置应用/服务的开发环境。环境配置可参考如图1-1所示的流程。

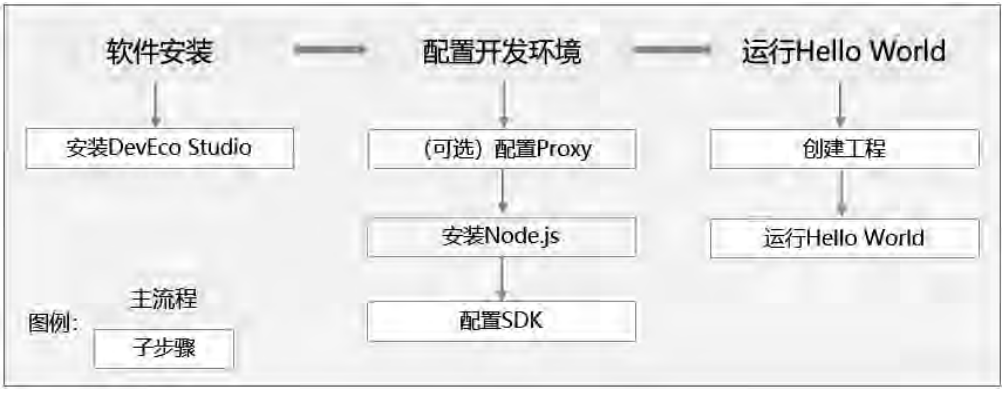


图1-1 环境配置流程

下面将分别介绍在Windows和macOS系统中安装DevEco Studio的操作方法。

1. Windows环境下的安装

在Windows环境下，用户可以通过华为开发者联盟官网下载DevEco Studio的安装包，并根据向导完成安装过程。为保证DevEco Studio正常运行，建议计算机配置满足如表1-1所示的条件。

表1-1　Windows环境下安装DevEco Studio的计算机配置

操作系统	Windows 10 64 位、Windows 11 64 位
内存	8GB及以上
硬盘	100GB及以上
分辨率	1280×800像素及以上

安装DevEco Studio的具体步骤如下：

01 下载完成后，双击下载的deveco-studio-xxxx.exe，进入DevEco Studio安装向导。在打开的对话框中选择安装路径，默认安装于C:\Program Files路径下，也可以单击Browse...按钮指定其他安装路径；然后单击Next按钮，如图1-2所示。

02 在安装选项界面勾选DevEco Studio复选框后，单击Next按钮，直至安装完成，如图1-3所示。

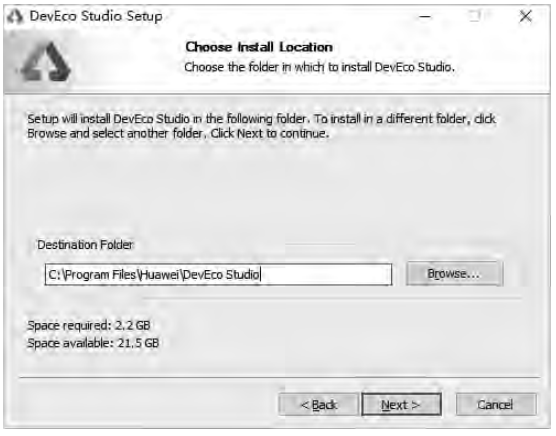


图1-2　安装DevEco Studio 1

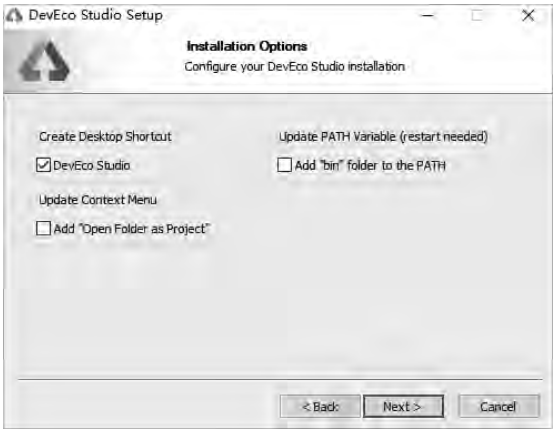


图1-3　安装DevEco Studio 2

03 最后，单击Finish按钮完成安装，如图1-4所示。

注意　Windows环境下同样需要配置Node.js环境，可以参考“2. macOS环境下的安装”中的“开发环境配置”中的相关步骤进行操作。

2. macOS环境下的安装

macOS用户可以按照Windows环境下的安装步骤安装DevEco Studio。为保证DevEco Studio能够在macOS系统中正常运行，建议计算机配置满足如表1-2所示的要求。

图1-4　安装DevEco Studio 3

表1-2　macOS环境下安装DevEco Studio的计算机配置

操作系统	macOS(X86) 10.15/11/12/13/14、macOS(ARM) 11/12/13/14
内存	8GB及以上
硬盘	100GB及以上
分辨率	1280×800像素及以上

1）安装 DevEco Studio

在安装界面中，将DevEco-Studio.app拖曳到Applications中，等待安装完成，如图1-5所示。

图1-5　安装DevEco Studio

2）开发环境配置

开发软件安装完成后，还需要进行环境配置才可以使用，具体步骤如下：

01 运行已安装的DevEco Studio，首次使用时选择Do not import settings，单击OK按钮。

02 安装Node.js。单击Local选项，可以指定本地已安装的Node.js（IDE级别）路径位置，如果本地没有合适的版本，可以单击Install选项，选择下载源和存储路径后进行在线下载；然后单击Next按钮进入下一步，如图1-6所示。

图1-6　开发环境配置1

03 在SDK Setup界面，单击按钮，设置HarmonyOS SDK存储路径，然后单击Next按钮进入下一步，如图1-7所示。

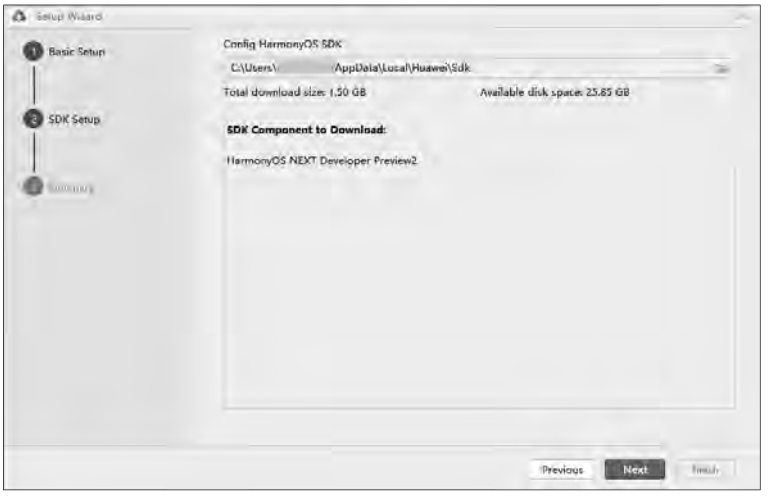


图1-7　开发环境配置2

04 确认设置项的信息，单击Next按钮开始安装，如图1-8所示。

图1-8　开发环境配置3

05 配置完成后，单击Finish按钮，界面会进入DevEco Studio欢迎页。

接下来，我们就可以使用DevEco Studio来构建工程项目了。

1.1.2　DevEco Studio的基本使用

安装完成后，用户可以开始探索DevEco Studio的基本功能，如创建Hello World项目，这是每个开发者入门编程的第一步。

让我们一起来创建第一个Hello World吧！

01 打开DevEco Studio，在欢迎页单击Create Project选项，创建一个新工程。

02 根据工程创建向导，选择创建Application或Atomic Service。选择Empty Ability模板，然后单击Next按钮，如图1-9所示。

图1-9　DevEco Studio的基本使用1

03 填写工程相关信息，然后单击Finish按钮，如图1-10所示。

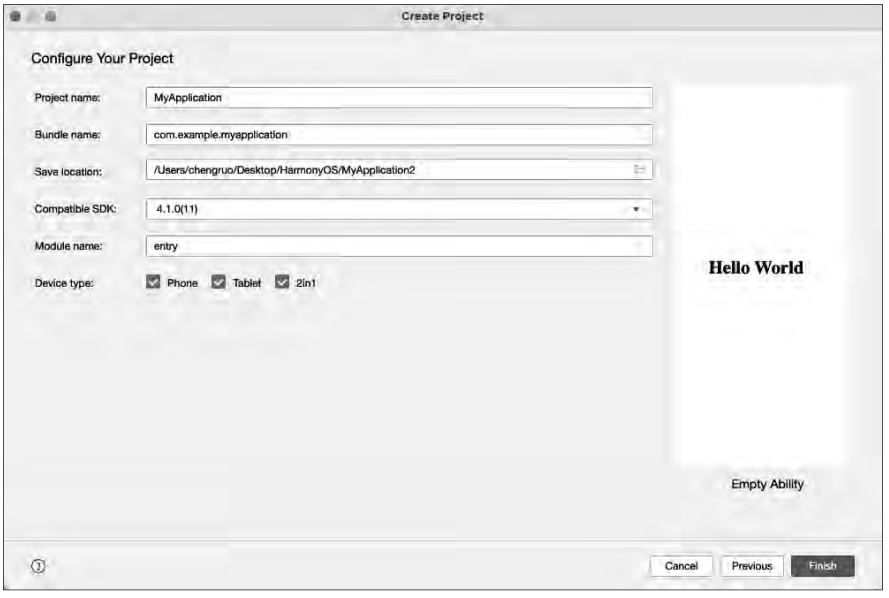


图1-10　DevEco Studio的基本使用2

注意 单击Finish按钮之后，DevEco Studio会自动进行工程的同步。等待工程同步完成之后就可以在界面右侧看到预览效果图了，如图1-10所示。创建工程页面的各项目参数解释如表1-3所示。

表1-3　创建工程页面的参数解释

项　　目	解　　释
Project name	工程的名称，可以自定义，由字母、数字和下画线组成
Bundle name	标识应用的包名，确保标识应用的唯一性
Save location	工程文件本地存储路径，由字母、数字和下画线组成，不能包含中文字符
Compile SDK	应用/服务的目标API版本，在编译构建时，DevEco Studio会根据指定的Compile API版本进行编译打包。如需开发API 11的应用/服务，则选择4.1.0(API 11)
Compatible SDK	兼容的最低API版本
Module name	模块的名称
Device type	该工程模板支持的设备类型
Node	配置当前工程运行的Node.js版本，可选择已有的Node.js或下载新的Node.js版本

1.1.3　手机运行Hello World应用

在掌握了DevEco Studio的基本操作后，我们将进入实践阶段，通过手机运行一个简单的Hello World应用，来直观体验HarmonyOS应用的开发流程和运行效果。

01 将搭载HarmonyOS系统的手机与计算机连接起来。

02 单击File→Project Structure...→Project→Signing Configs，勾选Support HarmonyOS和Automatically generate signature复选框，单击界面提示中的Sign In按钮，使用华为账号登录。等待自动签名完成后，单击OK按钮即可，如图1-11所示。

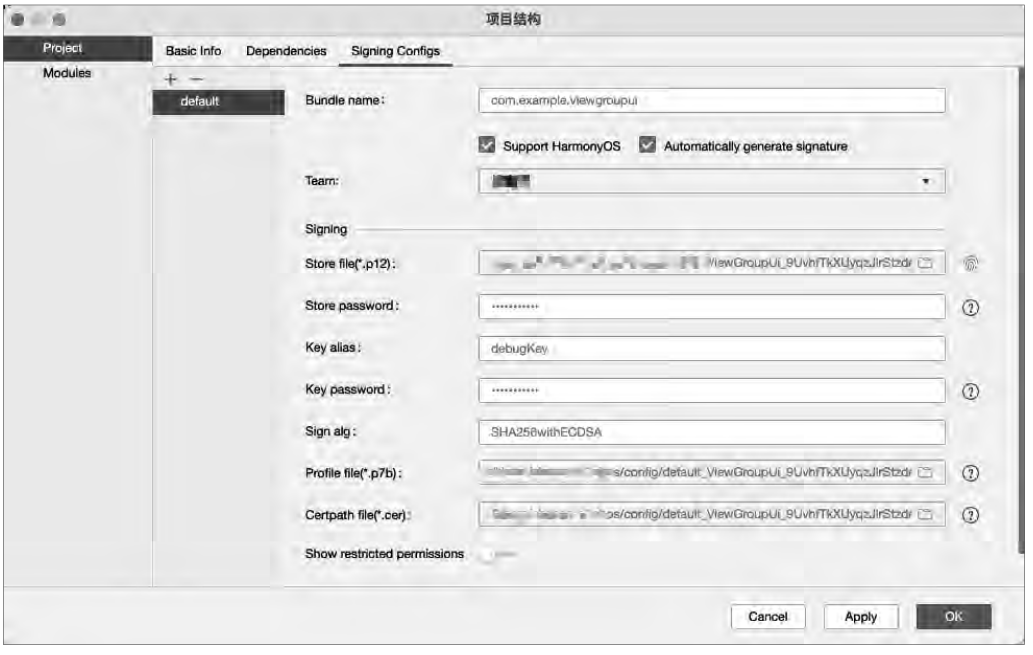


图1-11　配置项目选项

03 系统会自动生成工程代码，然后在编辑窗口右上角的工具中单击▶按钮运行，如图1-12所示。手机上就会出现“Hello World”的运行效果，如图1-13所示。

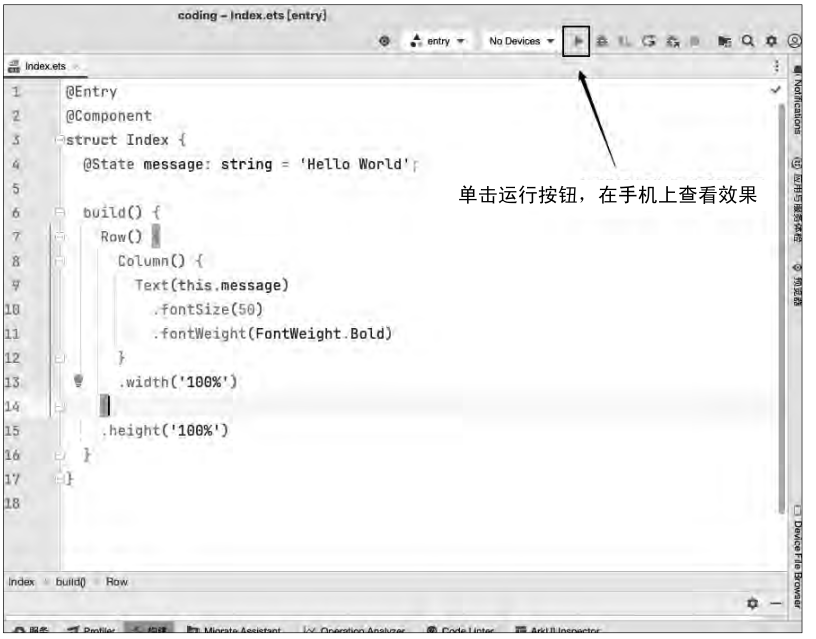


图1-12　自动生成的工程代码

图1-13　运行结果

至此，我们已成功运行了第一个应用。这虽然是一个入门案例，但却让我们迈出了成为HarmonyOS开发者的第一步。接下来，我们将循序渐进地介绍原生鸿蒙开发的一系列知识点。

1.1.4　了解基本工程目录

本小节将详细介绍ArkTS Stage模型的工程目录结构，重点要掌握该工程目录的构成及各部分的作用。

首先，看一下我们创建的第一个项目的目录结构，如图1-14所示。

目录结构中各部分的功能和作用说明如下：

- AppScope > app.json5：应用的全局配置信息。
- entry：应用/服务模块，编译构建生成一个HAP。

HAP（HarmonyOS Ability Package）是鸿蒙操作系统中的一种应用包格式，用于打包和分发应用程序。它与Android中的APK文件类似，包含应用程序的所有必要组件，如代码、资源、配置文件等。在鸿蒙系统中，HAP文件用于安装和运行应用程序。

- src > main > ets：用于存放ArkTS源码。
- src > main > ets > entryability：应用/服务的入口。
- src > main > ets > pages：应用/服务包含的页面。
- src > main > resources：用于存放应用/服务模块所用到的资源文件，如图形、多媒体、字符串、布局文件等。其相关目录及说明如表1-4所示。

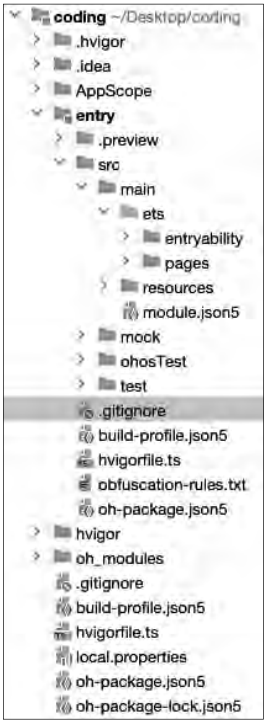


图1-14　目录结构

在实际的开发过程中，我们通常会在media目录下存放一些静态的媒体资源；在element目录下存放开发过程中的一些公有变量，比如全局的字体颜色、背景颜色以及一些文本信息等；当使用Web组件加载页面时，会在rawfile目录下存放一些HTML页面，用于本地的加载。

表1-4　resources（资源）目录文件说明

资源目录	资源文件说明
base>element	包括字符串、整型数、颜色、样式等资源的JSON文件。每个资源均由JSON格式进行定义，例如： ● boolean.json：布尔型 ● color.json：颜色 ● float.json：浮点型 ● intarray.json：整型数组 ● integer.json：整型 ● pattern.json：样式 ● plural.json：复数形式 ● strarray.json：字符串数组 ● string.json：字符串值
base>media	多媒体文件，如图形、视频、音频等文件，支持的文件格式包括.png、.gif、.mp3、.mp4等
rawfile	用于存储任意格式的原始资源文件。rawfile不会根据设备的状态去匹配不同的资源，需要指定文件路径和文件名进行引用

- src > main > module.json5：Stage模型模块配置文件，主要包含HAP的配置信息、应用在具体设备上的配置信息以及应用的全局配置信息。
- entry > build-profile.json5：当前的模块信息、编译信息配置项，包括buildOption、targets配置等。
- entry > hvigorfile.ts：模块级编译构建任务脚本。
- entry >oh-package.json5：配置第三方包声明文件的入口及包名。
- oh_modules：用于存放第三方库依赖信息，包含应用/服务所依赖的第三方库文件。
- build-profile.json5：应用级配置信息，包括签名、产品配置等。
- hvigorfile.ts：应用级编译构建任务脚本。

DevEco Studio采用了ArkTS Stage模型，以帮助开发者更好地组织和管理项目文件。了解该工程目录结构对于高效开发至关重要。

1.2　ArkTS语言之基本UI描述

ArkTS（Ark TypeScript）是一种基于TypeScript的声明式语言，专为HarmonyOS应用开发而设计。它允许开发者以简捷、高效的方式描述用户界面（UI）和交互逻辑。

ArkTS以声明方式组合和扩展组件来描述应用程序的UI，同时还提供了基本的属性、事件和子组件配置方法，帮助开发者实现应用交互逻辑。本节将会对创建组件、配置属性、配置事件以及配置子组件进行讲解，帮助初学者快速掌握单框架鸿蒙的开发规范。

1.2.1　基本概念

与TypeScript语言类似，在ArkTS语言中开发者也会经常和组件打交道。组件是构成HarmonyOS应用用户界面的核心，从文本框到按钮，再到图像，每一个视觉元素都是由组件构成的。这些组件不仅定义了应用的外观，还赋予了它们行为和灵魂。更令人兴奋的是，组件之间可以嵌套，形成复杂的父子关系，从而创造出具有无限可能的界面布局。

ArkTS中常用的基本概念如下：

- 组件：在ArkTS中一切皆为组件。组件是构成用户界面的基本单位，包括文本框、按钮、图像等。
- 属性：用于配置组件的外观和行为，如颜色、尺寸、边距等。
- 事件：定义组件如何响应用户操作，如单击、滑动等。
- 子组件：组件可以包含其他组件，形成嵌套结构，实现复杂的布局。

1.2.2　创建组件

根据组件构造方法的不同，创建组件包含无参数和有参数两种方式。

1. 无参数方式创建组件

如果组件的接口定义没有包含必选构造参数，则组件后面的“()”不需要配置任何内容。

例如，Divider组件不包含构造参数，展示的是一条横线；Text组件用于展示文本。代码如下：

```
// 定义一个名为Index2的结构体，用于构建UI组件
@Entry
@Component
struct Index2 {
 // build方法用于构建UI组件
 build() {
    // 创建一个Column组件，用于垂直排列子组件
    Column() {
      // 添加一个Text组件，显示文本"Item 1"
      Text('Item 1').fontSize(50)

      // 添加一个Divider组件，用于分隔内容
      Divider()
      // 添加一个Text组件，显示文本"Item 2"
      Text('Item 2').fontSize(50)

    }
   // 设置Column组件的宽度为100%
   .width('100%')
   // 设置Column组件的高度为100%
    .height('100%')
 }
}
```

界面效果如图1-15所示。

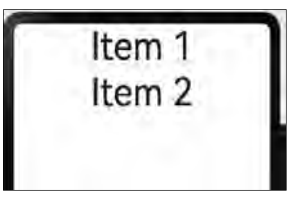


图1-15　无参数方式创建组件

2. 有参数方式创建组件

如果组件的接口定义包含构造参数，则组件后面的“()”需要配置相应参数。

例如Image组件的必选参数src，如图1-16所示。

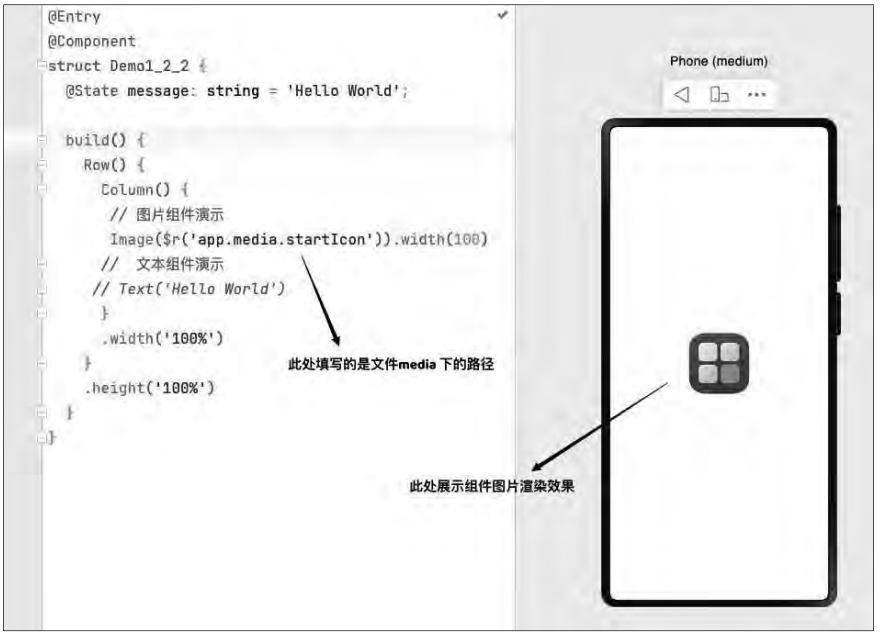


图1-16　有参数方式创建组件1

又如Text组件的非必选参数content，如图1-17所示。

图1-17　有参数方式创建组件2

通过创建组件可以知道，在ArkTS中的组件大致分为两类：一类是无参数组件，另一类是有参数组件。我们可以通过组合利用这些组件来完善应用程序。需要注意的是，在开发过程中更多的关注点会在组件的大的分类上，比如基础组件、容器组件、媒体组件等，通过对每一种组件的认知更加得心应手地进行项目开发。

1.2.3　属性配置

属性方法以“.”链式调用的方式配置系统组件的样式和其他属性，建议每个属性方法单独写一行。

1. 配置文本字体大小

配置文本字体大小的示例代码如下：

```
// 定义一个名为Demo1_2_3的结构体，用于构建UI组件
@Entry
@Component
struct Demo1_2_3 {
  // 定义一个状态变量message，初始值为'Hello World'
  @State message: string = 'Hello World';

  // build方法用于构建UI组件
  build() {
    // 创建一个Row组件，用于水平排列子组件
    Row() {
      // 创建一个Column组件，用于垂直排列子组件
      Column() {
        // 添加一个Text组件，用于显示message的值，并设置字体大小为50
        Text(this.message)
          .fontSize(50)
      }
      // 设置Column组件的宽度为100%
      .width('100%')
    }
    // 设置Row组件的高度为100%
    .height('100%')
  }
}
```

界面效果如图1-18所示。

2. 给文本配置多个属性

给文本配置多个属性的示例代码如下：

```
// 定义一个名为Demo1_2_3的结构体，用于构建UI组件
@Entry
@Component
struct Demo1_2_3 {
  // 定义一个状态变量message，初始值为'Hello World'
  @State message: string = 'Hello World';

  // build方法用于构建UI组件
  build() {
    // 创建一个Row组件，用于水平排列子组件
    Row() {
      // 创建一个Column组件，用于垂直排列子组件
      Column() {
        // 添加一个Text组件，用于显示message的值，并设置字体大小为50，字体粗细为800
        Text(this.message)
          .fontSize(50)
          .fontWeight(800)
      }
      // 设置Column组件的宽度为100%
```

```
        .width('100%')
      }
      // 设置Row组件的高度为100%
      .height('100%')
    }
  }
```

当添加了属性.fontWeight(800)时，字体会变粗，界面效果如图1-19所示。

图1-18　属性配置1

图1-19　属性配置2

3. 给文本配置变量或者表达式

给文本配置变量或者表达式的示例代码如下：

```
// 定义一个名为Demo1_2_3的结构体，用于构建UI组件
@Entry
@Component
struct Demo1_2_3 {
  // 定义一个状态变量message，初始值为'Hello World'
  @State message: string = 'Hello World';
  // 定义一个布尔类型的状态变量flag，初始值为false
  @State flag:boolean = false;
  // 定义一个数字类型的状态变量fontWeight，初始值为300
  @State fontWeight:number =300;

  // build方法用于构建UI组件
  build() {
    // 创建一个Row组件，用于水平排列子组件
    Row() {
      // 创建一个Column组件，用于垂直排列子组件
      Column() {
        // 添加一个Text组件，用于显示message的值
        Text(this.message)
          // 根据flag的值设置字体大小，如果flag为true，那么字体大小为30，否则为50
          .fontSize(this.flag ? 30 : 50)
          // 将fontWeight的值加上400后作为字体粗细值
          .fontWeight(this.fontWeight +400);
      }
      // 设置Column组件的宽度为100%
      .width('100%');
    }
    // 设置Row组件的高度为100%
```

```
      .height('100%');
    }
  }
```

在上述代码示例中，首先定义了两个变量flag和fontWeight，并在Text组件中采用三元运算符来判断文字的大小，字体的粗细采用常量传参计算的方式展示。界面效果如图1-20和图1-21所示。

当flag为false时

图1-20　属性配置3

当flag为true时

图1-21　属性配置4

4. 使用枚举的方式配置文本属性

在开发过程中，我们通常会将大部分相同的属性或者多个组件可以复用的属性单独定义出来，然后在使用过程中再进行属性的调用，这个时候大部分会采用枚举的方式。枚举类型是一种特殊的数据类型，约定变量只能在一组数据范围内选择值。

使用枚举的方式配置文本属性的示例代码如下：

```
// 1. 定义枚举（定义常量列表）
enum ThemeColor {
  Red = '#ff0f29',                          // 红色
  Orange = '#ff7100',                       // 橙色
  Green = '#30b30e'                         // 绿色
}

// 2. 给变量设定枚举类型
let color: ThemeColor = ThemeColor.Red      // 将颜色设置为红色
console.log('color', color)                 // 输出颜色值

@Entry
@Component
struct Demo1_2_3 {
  @State message: string = 'Hello World';  // 定义状态变量message，初始值为'Hello World'
  build() {
    Row() {                                 // 创建一个行组件
      Column() {                            // 创建一个列组件
        Text(this.message)                  // 创建一个文本组件，用于显示message的值
          .fontSize(50)                     // 设置字体大小为50
          .fontColor(color)                 // 设置字体颜色为之前定义的枚举值
      }
      .width('100%')                        // 设置列组件宽度为100%
    }
    .height('100%')                         // 设置行组件高度为100%
  }
}
```

在上述示例代码中通过enum来定义常量列表：

```
enum ThemeColor {
  Red = '#ff0f29',
  Orange = '#ff7100',
  Green = '#30b30e'
}
```

通过点链式调用的方式获取枚举的颜色值并赋给变量后，在text组件中就可以直接获取这个颜色值了，界面效果如图1-22所示。

图1-22　属性配置5

1.2.4　事件配置

事件方法和属性配置类似，也以“.”链式调用的方式配置系统组件支持的事件。常用的事件配置有以下4种方式：

（1）使用箭头函数配置组件的事件方法。

（2）使用匿名函数表达式配置组件的事件方法，要求使用“() => {...}”，以确保函数与组件绑定，同时符合ArkTS语法规范。

箭头函数和匿名函数表达式的事件方法是一样的，具体操作示例如下：

```
@Entry
@Component
struct Demo1_2_4 {
  @State message: string = 'Hello World';

  build() {
    Row() {
      Column() {
        // 使用箭头函数配置组件的事件方法
        Button('btn')
          .onClick(()=>{
            this.message = 'ArkUI'
          })

        Text(this.message).fontSize(50)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

在上述示例代码中，为Button组件绑定了箭头函数，当单击Button按钮时，Text组件的文本发生改变。界面效果如图1-23和图1-24所示。

单击按钮前

图1-23　事件配置1

单击按钮后

图1-24　事件配置2

（3）使用组件的成员函数配置组件的事件方法，需要bind this。

在实际的开发过程中，通常会遇见比较复杂的场景，这个时候如果将所有的处理逻辑都放在UI组件中，会使得项目臃肿，这种情况应该怎么处理呢？通常情况下我们会单独定义一个函数，在这个函数中编写逻辑，然后将这个函数绑定到UI组件中进行使用，这样项目就不会显得臃肿，项目结构也更加清晰明了。

示例代码如下：

```
@Entry
@Component
struct Demo1_2_4 {
  @State message: string = 'Hello World';

  myClickHandle():void{
    this.message = 'ArkUI'
  }
  build() {

    Row() {
      Column() {
        Button('btn')
          .onClick(this.myClickHandle.bind(this))

        Text(this.message).fontSize(50)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

在上述代码中，定义了一个函数myClickHandle，声明它的返回值是void，并且在Button按钮被单击时触发该函数。需要注意的是，这种方法需要使用bind(this)的方式进行绑定才可以触发。界面效果见图1-23和图1-24。

（4）在使用声明的箭头函数时，可以直接调用，不需要bind this。

这种情况是相对于方式（3）而言的，在开发过程中要省略bind this，只需要修改方式（3）的示例代码中的以下函数即可：

```
myClickHandle():void{
    this.message = 'ArkUI'
  }
```

将其改成箭头函数，代码如下：

```
myClickHandle=()=>{
  this.message = 'ArkUI'
}
```

注意，无论是方式（3）还是方式（4），对于函数的使用均不是调用，也就是函数名后面不需要加上()。

1.2.5　子组件配置

如果组件支持子组件配置，则需在尾随闭包“{...}”中为组件添加子组件的UI描述。Column、Row、Stack、Grid、List等组件都是容器组件，可以在其中添加子组件。

在1.2.4节中会发现，所有的Text组件或者Button组件都在容器组件Column中，当然它们的属性也在对应的组件后面进行添加。

在进行布局时，通常情况下我们会对想要修改的组件进行属性配置或者事件配置，而容器组件则会对子组件进行控制。

1.3　ArkTS语言之状态管理

在ArkTS中，状态管理是构建响应式用户界面的关键。通过一系列装饰器和指令，ArkTS提供了一种简捷而有效的方式来处理组件的状态和数据流。本节将介绍ArkTS语言中的状态管理相关特性，包括@State、@Prop、@Link、@Consume、@Provide、@Consume、@Provide以及@Watch。

1.3.1　@State

@State装饰器用于定义状态变量，它赋予变量状态属性，使其与自定义组件的渲染紧密绑定。当状态发生改变时，UI会自动同步更新。这类变量是私有的，仅能在组件内部访问。在声明状态变量时，需要指定其类型和初始值，也可以通过命名参数由父组件进行初始化。

其特性包括：

- 装饰器参数：无须提供。
- 同步类型：不与父组件中的任何变量同步。
- 允许装饰的变量类型：包括Object、class、string、number、boolean、enum，以及这些类型的数组。支持Date类型。API 11及以上版本支持Map、Set类型。同时也支持undefined和null类型。
- 类型支持场景：请参考相关文档观察变化。
- API 11及以上版本支持联合类型，例如string | number，string | undefined或ClassA | null。
- 使用undefined和null时，建议显式指定类型，以遵循TypeScript的类型校验。推荐写法：@State a: string | undefined = undefined。不推荐写法：@State a: string = undefined。
- 支持ArkUI框架定义的联合类型Length、ResourceStr、ResourceColor。
- 类型必须指定，不支持使用any类型。
- 被装饰变量的初始值：必须本地初始化。

@State变量传递/访问规则如下：

- 初始化：可为空，从父组件或本地初始化。若从父组件初始化，将覆盖本地初始化。
- 支持类型：可以从父组件传递常规变量（仅数值初始化，不触发UI刷新），以及由@State、@Link、@Prop、@Provide、@Consume、@ObjectLink、@StorageLink、@StorageProp、@LocalStorageLink和@LocalStorageProp装饰的变量来初始化子组件的@State。
- 初始化子组件：由@State装饰的变量可以用来初始化子组件的常规变量、@State、@Link、@Prop、@Provide。
- 访问限制：不支持组件外访问，变量只能在组件内部访问。

初始化规则如图1-25所示。

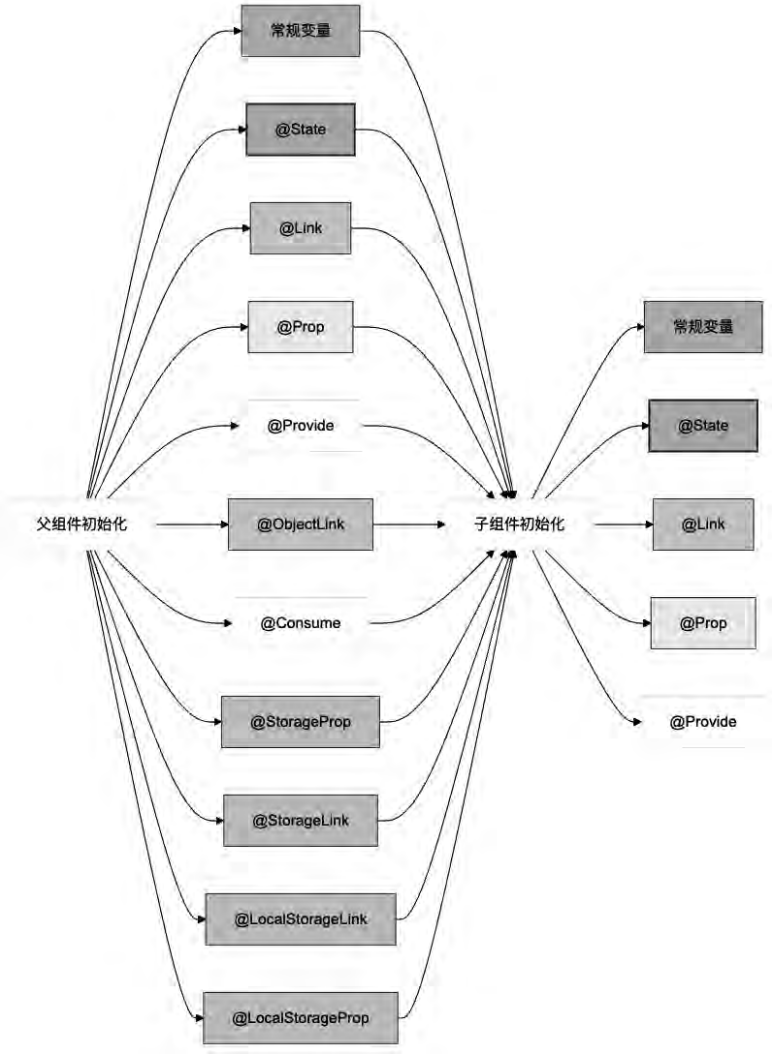


图1-25　初始化规则

注意，并非所有状态变量的变更都会导致UI更新，只有框架能观察到的修改才会触发UI更新。接下来将阐述哪些修改会被框架观察到，以及框架在观察到变化后的行为。

（1）当@State装饰的数据类型为boolean、string、number时，可以观察到数值的变化。也就是说，当数据类型为boolean、string、number时，如果数据发生改变，那么页面也会同步更新。示例如下：

```
@Entry
@Component
struct State_demo {
  @State bool: boolean = false
  @State message: string = '你好'
  @State num: number = 1

  build() {
    Row() {
      Column() {
        Button('修改数据').onClick(() => {
          this.bool = !this.bool
```

```
            this.message = '数据被修改了'
            this.num++
          })

          // 修改string类型
          Text(this.message).fontSize(20)
          // 修改number 类型
          Text(this.num.toString()).fontSize(20)
          // 修改boolean 类型
          Text(this.bool.toString()).fontSize(20)
        }
        .width('100%')
      }
      .height('100%')
    }
  }
```

在上述示例代码中，定义了3种数据类型，通过单击Button按钮来进行数据修改。需要注意的是，在ArkTS中所有需要渲染出来的内容必须是string格式，因此将number和boolean类型的数据通过toString()方法转换成字符串。效果如图1-26和图1-27所示。

（2）当@State装饰的数据类型为class或者Object时，可以观察到自身赋值的变化和其属性赋值的变化，即Object.keys(observedObject)返回的所有属性。

单击按钮前

图1-26　@State装饰器1

单击按钮后

图1-27　@State装饰器2

示例如下：

```
// 声明一个TestClass类，包含一个字符串类型的value属性
class TestClass {
  value: string;

  // 构造函数，接收一个字符串参数并将其赋值给value属性
  constructor(value: string) {
    this.value = value;
  }
}

// 声明一个Model类，包含两个公共属性：一个字符串类型的value和一个TestClass类型的name
class Model {
  public value: string;
  public name: TestClass;

  // 构造函数，接收一个字符串参数和一个TestClass实例，并分别赋值给value和name属性
  constructor(value: string, a: TestClass) {
```

```
        this.value = value;
        this.name = a;
      }
    }
```

接下来使用定义的类型来观察数据的变化。首先使用@State装饰Model，然后进行数据修改，以观察是否可以修改数据。示例代码如下：

```
    // 定义一个名为message的@State变量，类型为Model，初始值为一个新的Model实例，包含字符串'Hello'和一个TestClass实例，其参数为'World'
    @State message: Model = new Model('Hello', new TestClass('World'));

    build() {
      // 创建一个Row组件，宽度为100%
      Row() {
        // 创建一个Column组件
        Column() {
          // 创建一个按钮，单击该按钮时修改message的值
          Button('修改数据').> {
            // 将message的值更新为一个新的Model实例，包含字符串'HI'和一个TestClass实例，其参数为'数据被修改了'
            this.message = new Model('HI', new TestClass('数据被修改了'))
          })

          // 创建一个文本组件，用于显示message的value属性值，字体大小为20
          Text(this.message.value).fontSize(20)
          // 创建一个文本组件，用于显示message的name属性的value属性值，字体大小为20
          Text(this.message.name.value).fontSize(20)
        }
        .width('100%') // 设置Column组件的宽度为100%
      }
      .height('100%') // 设置Row组件的高度为100%
    }
```

界面效果如图1-28和图1-29所示。说明当@State装饰的是class类型时，数据的变化是可以被观察到的。

单击按钮前

图1-28　@State装饰器3

单击按钮后

图1-29　@State装饰器4

如果只修改@State装饰的某一个属性呢？

示例代码如下：

```
    // 创建一个名为message的Model实例，包含两个属性：value和name
    @State message: Model = new Model('Hello', new TestClass('World'))

    build() {
      // 创建一个Row布局
```

```
      Row() {
        // 创建一个Column布局
        Column() {
          // 创建一个按钮，单击该按钮后修改message的值和name的值
          Button('修改数据').> {
            this.message.value = 'HI'
            this.message.name.value = '数据被修改了'
          })

          // 显示message的value属性值，字体大小为20
          Text(this.message.value).fontSize(20)
          // 显示message的name属性值，字体大小为20
          Text(this.message.name.value).fontSize(20)
        }
        // 设置Column布局的宽度为100%
        .width('100%')
      }
      // 设置Row布局的高度为100%
      .height('100%')
    }
```

在示例代码中，我们通过以下代码来修改@State装饰的数据：

```
    this.message.value = 'HI'
    this.message.name.value = '数据被修改了'
```

效果图见图1-28和图1-29，发现数据被修改后依旧可以被观察到。

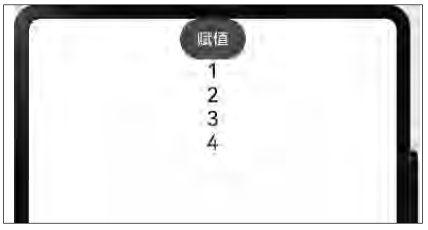
（3）当装饰的对象是array时，可以观察到数组本身的赋值和添加、删除、更新数组的变化。

数组的操作在开发的过程中是比较常见的，通过下面的代码示例将会更好地理解关于数组的操作方案。

```
    // 第一步：声明一个ArrayClass类，包含一个数值类型的属性value
    class ArrayClass {
      value: number;

      // 构造函数，用于初始化ArrayClass实例的value属性
      constructor(value: number) {
        this.value = value;
      }
    }

    // 第二步：使用@State装饰器创建一个ArrayClass类型的数组cousomArray，并初始化为包含4个元素的数组
    @State cousomArray: ArrayClass[] = [new ArrayClass(1), new ArrayClass(2), new
ArrayClass(3), new ArrayClass(4)];

    // 第三步：对数组进行操作
    // 赋值操作：单击按钮后，将cousomArray数组重新赋值为只包含一个元素（值为11）的新数组
    build() {
      Column() {
        Button('赋值').> {
          this.cousomArray = [new ArrayClass(11)];
        })
        // 遍历cousomArray数组，显示每个元素的value值
        ForEach(this.cousomArray, (item: ArrayClass, index) => {
          Text(item.value.toString()).fontSize(20);
        });
      }
      .width('100%');
    }
```

注意　代码中用了一个新的属性 ForEach ，这个属性的作用是遍历数组，将数组的每一项内容进行展示，后面会有详细讲解。

案例代码主要是实现当单击按钮时，将自定义数组对象的内容进行替换，效果如图1-30和图1-31所示。



单击按钮前

图1-30　数组的赋值操作1

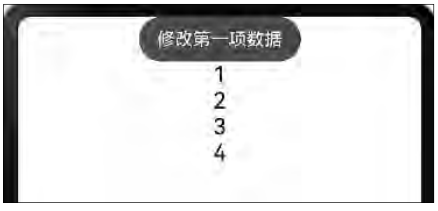
单击按钮后

图1-31　数组的赋值操作2

修改数组中的某一项数据，代码如下：

```
Button('修改第一项数据').onClick(() => {
   this.cousomArray[0] = new ArrayClass(11)
 })
```

单击按钮直接对数组中的第一项数据进行修改，效果如图1-32和图1-33所示。



单击按钮前

图1-32　数组的替换操作1

单击按钮后

图1-33　数组的替换操作2

删除数组中的数据，代码如下：

```
Button('删除数据').onClick(()=>{
   this.cousomArray.pop()
})
```

有JavaScript基础的读者应该很清楚，案例中我们使用了pop()方法对数组进行数据的删除操作，效果如图1-34和图1-35所示。

单击按钮前

图1-34　数组的删除操作1

单击按钮后

图1-35　数组的删除操作2

为数组新增数据，代码如下：

```
Button('新增数据').onClick(()=>{
  this.cousomArray.push(new ArrayClass(20))
})
```

采用数组的操作方法push为cousomArray这个数组项新增了一个数据，效果如图1-36和图1-37所示。

单击按钮前

图1-36　数组的新增操作1

单击按钮后

图1-37　数组的新增操作2

我们通过.value来获取数组中对应的数据，然后直接进行赋值操作，是否可以呢？

直接对数组中的属性进行赋值，代码如下：

```
Button('直接赋值').onClick(()=>{
  this.cousomArray[0].value = 10
})
```

效果如图1-38所示。可以发现这种赋值方式是不可以的。

图1-38　对数组中的属性进行赋值

1.3.2　@Prop

@Prop装饰器用于将父组件的数据传递给子组件，使得子组件能够接收和使用这些数据。

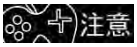
@Prop装饰器具有以下特点：

- 单向下行绑定：父组件的状态变化会同步到子组件的@Prop变量，但子组件@Prop变量的修改不会回传给父组件。
- 支持类型：Object、class、string、number、boolean、enum、数组，以及这些类型的联合体，如string | number。从API 11开始支持Map、Set类型。
- 建议对于undefined和null使用显式类型定义，以利用TypeScript的类型校验。
- 支持ArkUI框架的特定联合类型Length、ResourceStr、ResourceColor，必须指定类型。

@Prop和数据源类型需一致，包括简单数据类型同步、数组项同步以及对象属性同步。

嵌套传递层数建议不超过5层，以避免性能问题。

允许本地初始化@Prop变量，与@Require结合时父组件需构造传参。



由@Prop装饰的变量在进行深拷贝时，除了基本类型、Map、Set、Date、Array之外，其他类型会丢失。@Prop装饰器不适用于使用@Entry装饰的自定义组件。

接下来从@Prop的不同使用场景来进一步了解其特性。

场景一：父组件@State到子组件@Prop简单数据类型同步

案例实现思路如下：

（1）在入口文件中用@State装饰器装饰一个count变量，声明两个按钮用于操作count的数据变化，添加一个Text文本组件来展示当前组件下的count值。

（2）引入子组件，用于展示，并将count作为值传递过去。

（3）定义一个子组件ChildComponent，子组件中用@Prop装饰器接收count的值，依旧定义两个按钮用于操作@Prop装饰器装饰的count数据，同样添加一个Text文本组件用于展示当前组件下的count值。

具体代码实现如图1-39所示。

初始效果如图1-40所示。

```
@Component
struct ChildComponent {
  @Prop count: number = 0

  build() {
    Column() {
      Button('子组件减一').onClick(() => {
        this.count -= 1
      })
      Button('子组件加一').onClick(() => {
        this.count += 1
      })
      Text(`此时子组件的值是:${this.count}`)
    }
  }
}

@Entry
@Component
struct Demo1_3_2Prop {
  @State count: number = 0;

  build() {
    Column() {
      Button('父组件减一').onClick(() => {
        this.count -= 1
      })
      Button('父组件加一').onClick(() => {
        this.count += 1
      })

      Text(`此时父组件的值是:${this.count}`)
      ChildComponent({ count: this.count })

    }
    .justifyContent(FlexAlign.Start)
    .height('100%')
    .width('100%')
  }
}
```

子组件布局：
检测@Prop装饰的数据在当前组件下被修改之后，父组件的数据是否会发生变化。

父组件布局：
1. 检测用@State装饰的数据被修改后，是否会向下流转到子组件
2. 检测子组件数据被修改后，父组件的数据是否会改变

图1-39　父组件状态至子组件属性的简单数据同步

图1-40　初始效果

单击“父组件加一”按钮，此时父、子组件的count值均为1，效果如图1-41所示。

接着单击“子组件减一”按钮，此时发生了变化，父组件的count值依旧是1，但是子组件的count值却变成了0，效果如图1-42所示。

因此得出结论，@Prop装饰器是单向同步的，子组件@Prop变量的修改不会同步到父组件的状态变量上。

图1-41　单击父组件按钮

图1-42　单击子组件按钮

场景二：@Prop嵌套场景

在嵌套场景下，每一层都要用@Observed装饰，且每一层都要被@Prop接收，这样数据才能形成响应式。

接下来我们逐步地进行解析。

（1）首先创建一个嵌套类的对象结构，代码如下：

```
// 创建一个嵌套类对象的数据结构
@Observed
class ClassA {
  // 定义一个公共属性 title，类型为字符串
  public title: string;

  // 构造函数，接收一个字符串参数 title，并将其赋值给实例的 title 属性
  constructor(title: string) {
    this.title = title;
  }
}

@Observed
class ClassB {
  // 定义一个公共属性 name，类型为字符串
  public name: string;
  // 定义一个公共属性 a，类型为 ClassA 类的实例
  public a: ClassA;

  // 构造函数，接收一个字符串参数name和一个ClassA类的实例a，并分别赋值给实例的name和a属性
  constructor(name: string, a: ClassA) {
    this.name = name;
    this.a = a;
  }
}
```

（2）项目文件的结构布局如图1-43所示。

在父组件中，创建了两个按钮用于修改testObj这个ClassB类，同时为了验证在嵌套场景下，每一层都要用@Observed装饰，且每一层都要被@Prop接收，这样才能实现数据的动态渲染效果，我们在text组件上也添加了单击事件用于修改数据，但这不是必需的，主要为了方便演示。因为在ClassB类中，name属性并不在嵌套属性内，也就是说当嵌套的属性title发生了变化，但是页面没有发生改变时，如果name的数据发生了改变，那么页面中嵌套属性的数据也会跟着更新，但这并不是我们想要的效果。如果想要实现嵌套属性的动态渲染，应该怎么做呢？这里引入了Child1这个子组件，并用@Prop来接收嵌套的类ClassA，从而实现动态渲染。

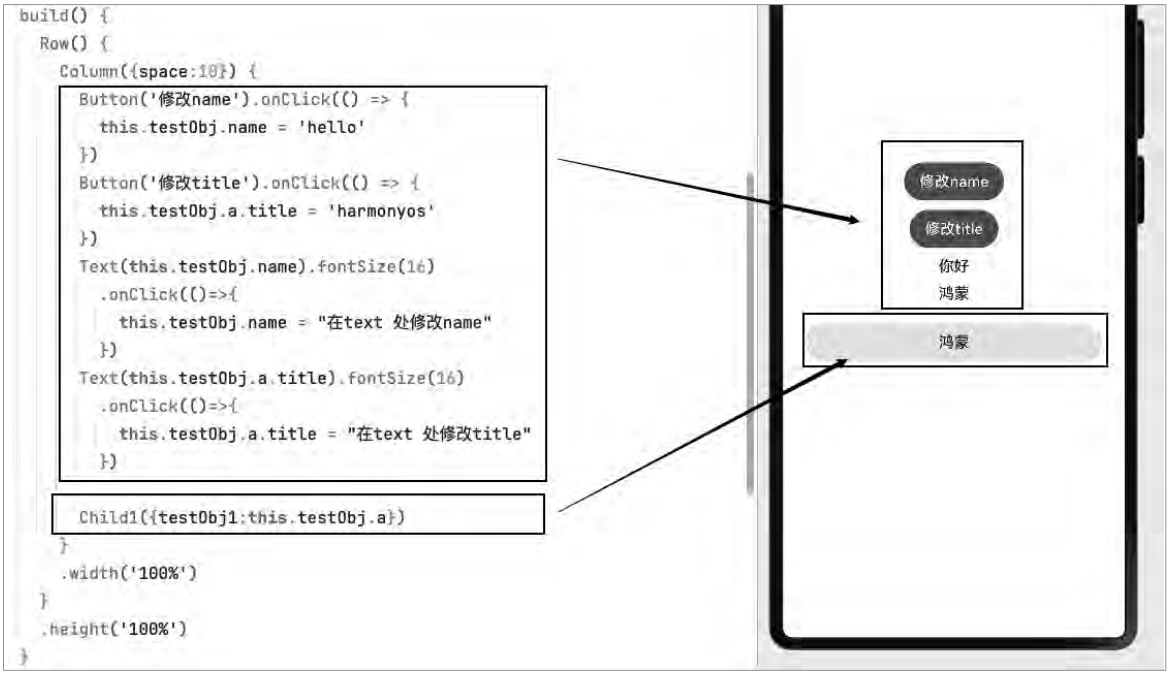


图1-43　项目文件的结构布局

（3）编写子组件并使用@Prop来装饰ClassA，详细代码及解释如图1-44所示。

图1-44　Child子组件

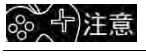
完整代码如下：

```
// 创建一个嵌套类对象的数据结构
@Observed
class ClassA {
  public title: string;   // 定义一个公共属性 title，类型为字符串

  constructor(title: string) {
    this.title = title    // 构造函数，接收一个字符串参数 title，并将其赋值给实例的 title 属性
  }
}

@Observed
class ClassB {
  public name: string;         // 定义一个公共属性 name，类型为字符串
  public a: ClassA;            // 定义一个公共属性 a，类型为 ClassA 类的实例

  constructor(name: string, a: ClassA) {
    this.name = name; // 构造函数，接收一个字符串参数name和一个ClassA类的实例a，并分别赋值给实例的name和a属性
    this.a = a;
  }
}

@Entry
@Component
struct Demo1_2_3prop_Observed {
  @State testObj: ClassB = new ClassB('你好', new ClassA('鸿蒙')) // 定义一个状态变量 testObj，类型为 ClassB，并初始化为一个新的 ClassB 实例

  build() {
    Row() {                          // 创建一个行布局
      Column({space:10}) {           // 创建一个列布局，设置间距为10
        Button('修改name').> {       // 创建一个按钮，单击该按钮时修改testObj的name属性
          this.testObj.name = 'hello'
        })
        Button('修改title').> {      // 创建一个按钮，单击该按钮时修改testObj的a属性的title属性
          this.testObj.a.title = 'harmonyos'
        })
        Text(this.testObj.name).fontSize(16)          // 创建一个文本组件，用于显示testObj的name属性，字体大小为16
          .>{
            this.testObj.name = "在text 处修改name"    // 单击文本组件时修改testObj的name属性
          })
        Text(this.testObj.a.title).fontSize(16)       // 创建一个文本组件，用于显示testObj的a属性的title属性，字体大小为16
          .>{
            this.testObj.a.title = "在text 处修改title" // 单击文本组件时修改testObj的a属性的title属性
          })
```

1.3.3　@Link

在项目开发的过程中如果只有@Prop父子单向同步，那将有许多场景在开发的过程中产生过多不必要的代码。因此，HarmonyOS不仅提供了单向数据同步，还提供了父子双向同步的方法——@Link。

@Link装饰器用于创建双向绑定，使得父组件和子组件之间的数据能够实时同步更新。



@Link装饰器不能在@Entry装饰的自定义组件中使用。

接下来通过一个案例来讲解@Link的使用方法。

父组件代码结构如图1-45所示。

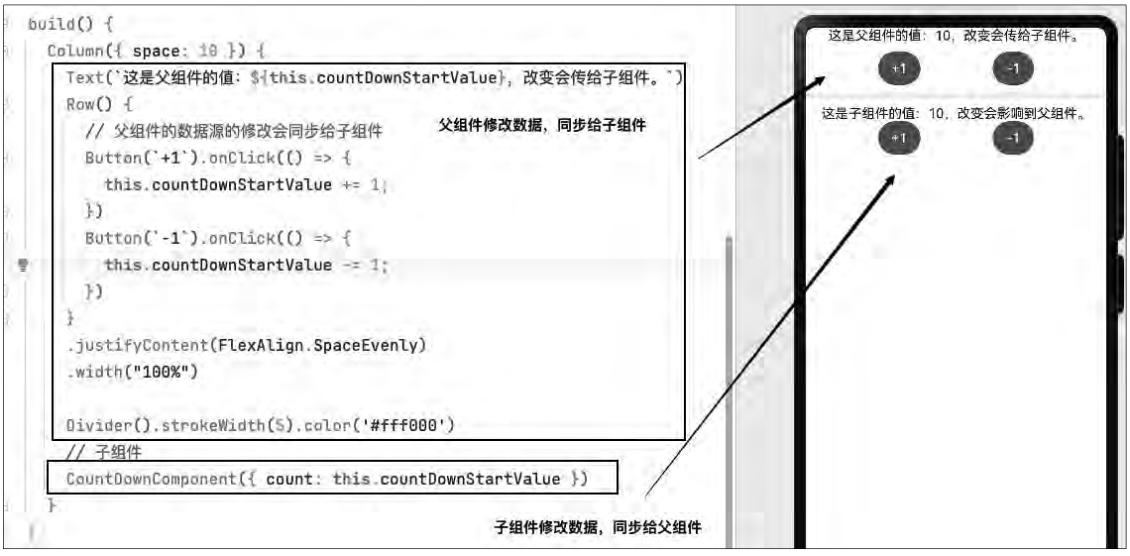


图1-45　父组件代码结构

子组件代码结构如图1-46所示。

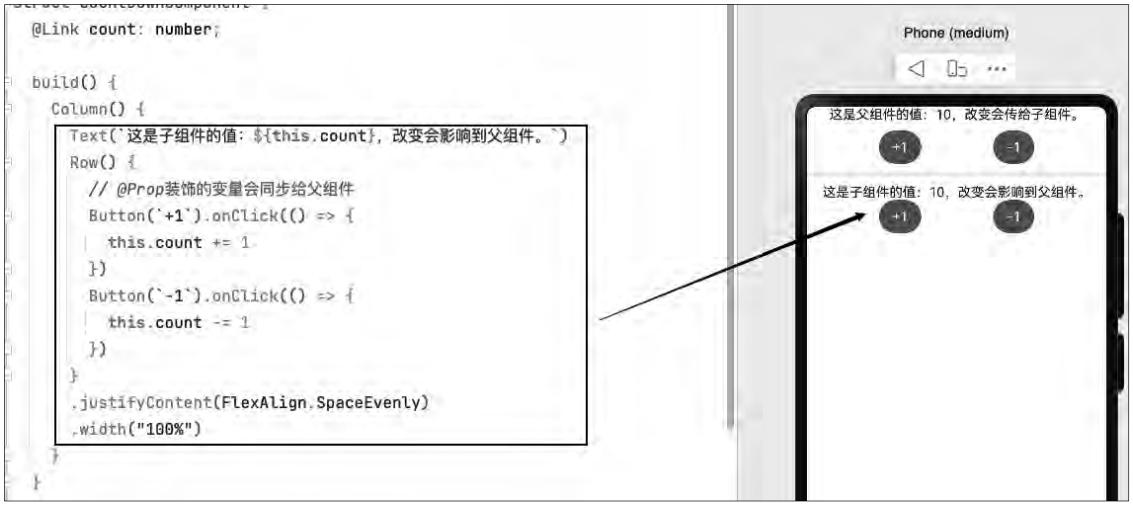


图1-46　子组件代码结构

在父组件中通过@State定义变量countDownStartValue，子组件通过@Link使用组件传值的方式来接收变量，此时便可以实现父、子组件数据的双向同步。具体效果：当单击父组件的“+1”或者“−1”按钮时父、子组件的值会同时改变；同样单击子组件的“+1”或者“−1”按钮时父、子组件的值也会同时改变，从而实现了父、子组件的双向同步。这里需要注意的是子组件的传值方式：

```
CountDownComponent({ count: $countDownStartValue })
```

这里$countDownStartValue可以写成this.countDownStartValue，主要区别于是否添加$符号。Count既是接收的变量，也是@Link装饰的变量。

1.3.4　@Observed和@ObjectLink

在开发过程中，不仅会用到简单的数据类型，一些复杂的数据类型也是经常需要处理的。对于复杂数据类型，比如二维数组或者数组项是class，或者class的属性是class等这类数据，它们的第二层属性的变化是无法动态渲染的。基于此，HarmonyOS提供了两个装饰器@Observed和@ObjectLink，用来操作此类数据。

@Observed装饰器用于监听对象属性的变化，而@ObjectLink则用于链接两个对象，确保它们的状态同步。

@Observed类装饰器用于装饰class，不需要参数，用于观察属性变化，配合@ObjectLink使用。@ObjectLink变量装饰器用于指定必须为@Observed装饰的class实例的变量，支持复杂类型，如Date、Array、Map、Set以及它们的联合类型。@ObjectLink装饰的变量是只读的，不能改变分配的值。

@Observed和@ObjectLink装饰器用于在涉及嵌套对象或数组元素为对象的场景中进行双向数据同步。

在1.3.2节的@Prop嵌套场景中有对上述装饰器的一些介绍，接下来从@Observed和@ObjectLink的使用场景开始，详细介绍两个装饰器的使用。

观察如图1-47所示的基本案例。

```
@Observed
class Person {                    定义一个复杂的数据结构
  name: string
  age: number
  girlfriend: Person | undefined

  constructor(name: string, age: number, girlfriend?: Person) {
    this.name = name
    this.age = age
    this.girlfriend = girlfriend
  }
}

//子组件
@Component
struct Childs {                   用于接收传值，同时赋予动态属性
  @ObjectLink p: Person

  build() {
    Column() {
      Text(`姓名: ${this.p.name} , 年龄 ${this.p.age} `)
    }
  }
}
```

```
@Entry
@Component
struct Demo1_3_4 {                初始化数据
  @State p: Person =
    new Person('小明', 20, new Person('李华', 19))
  @State gfs: Person[] = [
    new Person('韩梅梅', 20),
    new Person('刘夏', 21),
  ]
  build() {
    Column() {
      // 传值
      Childs({ p: this.p.girlfriend })    给子组件传值
      Text('女友的列表')
      ForEach(this.gfs, (item: Person) => {
        Childs({ p: item }).onClick(() => {
          item.age++
        })
      })
      Text('父组件中嵌套数据的变化')
      ForEach(this.gfs, (item: Person) => {
        Text(`姓名: ${item.name} , 年龄 ${item.age} `).onClick(() => {
          item.age++
        })
      })
    }
    .width('100%')
    .height('100%')
  }
}
```

图1-47　基本案例

在这个案例中可以发现以下几种现象：

- 单击子组件的元素时，对应的数据会同步更新。
- 单击父组件元素时，对应的数据不会同步更新，相反子组件的数据会进行更新。

子组件的单击事件可以改成如下方法，效果一致：

```
Child({ p: item }).onClick(() => {
  item.age++
})
```

完整代码如下：

```
@Observed
class Person {
  name: string
  age: number
  girlfriend: Person | undefined

  constructor(name: string, age: number, girlfriend?: Person) {
    this.name = name
    this.age = age
    this.girlfriend = girlfriend
  }
}

    @Entry
    @Component
    struct Demo1_3_4 {
      @State p: Person =
        new Person('小明', 20, new Person('李华', 19))
      @State gfs: Person[] = [
        new Person('韩梅梅', 20),
        new Person('刘夏', 21),
      ]
      build() {
        Column() {
          // 传值
          Child({ p: this.p.girlfriend })
          Text('女友的列表')
          ForEach(this.gfs, (item: Person) => {
            Child({ p: item }).onClick(() => {
              item.age++
            })
          })
          Text('父组件中嵌套数据的变化')
          ForEach(this.gfs, (item: Person) => {
            Text(`姓名: ${item.name} , 年龄 ${item.age} `).onClick(() => {
              item.age++
            })
          })
        }
        .width('100%')
        .height('100%')
      }
    }

// 子组件
@Component
struct Child {
  @ObjectLink p: Person

  build() {
    Column() {
```

```
          Text(`姓名: ${this.p.name} , 年龄 ${this.p.age} `)
        }
      }
    }
```

1.3.5　@Consume和@Provide

设想这样一种场景，有3个及以上的组件层级，类似于如图1-48所示的组件层级。如果子组件3要访问根组件的某个变量，我们应该如何去做呢？此时的想法可能是将根组件的变量传递给子组件1，然后从子组件1传递给子组件2，最后从子组件2传递给子组件3。虽然这样可以实现我们的目的，但是设想一下，如果组件的嵌套层次达到100层呢。对此HarmonyOS提供了两个装饰器@Consume和@Provide，主要用于在多个层级之间传递数据的场景。

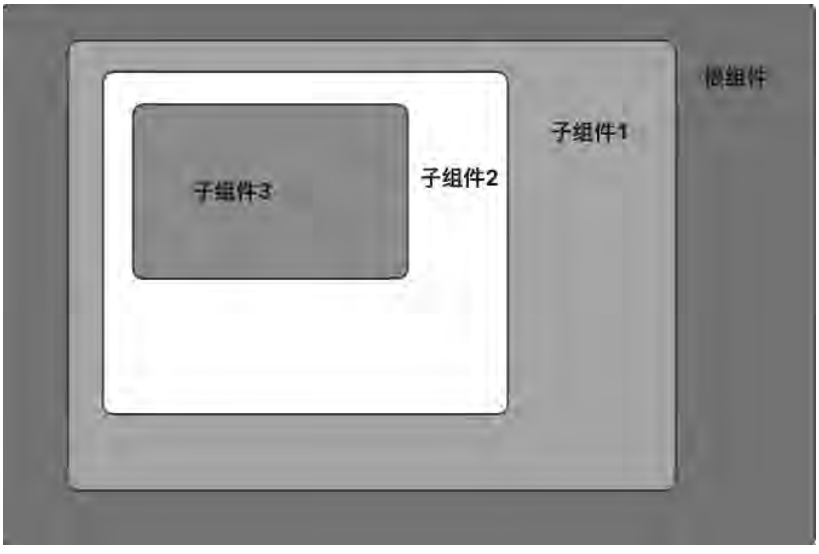


图1-48　组件层级

这种双向数据同步的机制为开发者提供了一种优雅的方式来管理状态数据的流动，尤其适用于多层级的父子组件之间的数据传递。

@Consume和@Provide装饰器特性如下：

- @Provide装饰的变量位于祖先节点中，被提供给后代组件，可以看作“提供”给后代的状态变量。
- @Consume装饰的变量位于后代组件中，用于“消费（绑定）”祖先节点提供的变量。

接下来通过案例来掌握这两个装饰器的具体使用场景。在案例中，单击根组件的按钮可以修改变量的数据，同时单击@Consume 装饰的后代组件中的按钮依旧可以修改此变量，具体代码示例以及代码解释如图1-49所示。

图1-49　代码示例

完整代码如下：

```
@Entry
@Component
struct CompA {
  // @Provide装饰的变量count由入口组件CompA提供给后代组件
  @Provide count: number = 0;

  build() {
    Column() {
      Button(`CompA组件数据: (${this.count})`)
        .onClick(() => this.count += 1)
      CompB()
    }
  }
}

@Component
struct CompB {
  build() {
    CompC()
  }
}

@Component
```

```
struct CompC {
  build() {
    Row({ space: 5 }) {
      CompD()
      CompD()
    }
  }
}

@Component
struct CompD {
  // @Consume装饰的变量通过相同的属性名绑定其祖先组件CompA内的用@Provide装饰的变量
  @Consume count: number;

  build() {
    Column() {
      Text(`CompD组件数据(${this.count})`)
      Button('CompD组件按钮')
        .onClick(() => this.count += 1)
    }
    .width('50%')
  }
}
```

1.3.6　@Watch

有这样一个场景，当一个变量发生变化时，页面中的某个元素会动态地显示与隐藏。在这种情况下，就需要对这个变量进行监听，也就是需要使用本小节将要介绍的另一个装饰器@Watch。

@Watch装饰器用于监视某个表达式的变化，当表达式的值发生变化时，可以执行特定的逻辑操作。@Watch装饰器的特性如下：

- @Watch装饰器用于监听状态变量变化并执行回调。
- 装饰器参数必须是对(string) => void类型函数的引用。
- @Watch适用于所有被装饰器标记的状态变量，不包括常规变量。
- 推荐先使用@State、@Prop、@Link等装饰器，再使用@Watch。

在下面的案例中，用@Watch装饰器装饰父组件传过来的变量hasShow，并定义函数handleChange，当监听的数据发生变化时，handleChange函数就会被触发。因此，我们可以在这里进行一些相关的操作，如图1-50所示。

图1-50　@watch的使用

完整代码如下：

```
// 定义一个名为Demo1_3_6Watch的组件，用于展示状态监听和子组件更新
@Entry
@Component
struct Demo1_3_6Watch {
  // 定义一个名为hasShow的状态变量，初始值为true
  @State hasShow: boolean = true;

  // 构建组件的方法
  build() {
      Column() {
         // 创建一个按钮，单击该按钮时修改hasShow的值
         Button('修改hasShow 状态').>{
            this.hasShow = ! this.hasShow
         })
        // 创建一个ChildWatch组件，并将hasShow作为props传递给它
        ChildWatch({hasShow:this.hasShow})
      }
      // 设置组件的宽度和高度为100%
      .width('100%')
      .height('100%')
  }
}

// 定义一个名为ChildWatch的组件，用于监听父组件传递过来的props的变化
@Component
struct ChildWatch{
  // 定义一个名为hasShow的props，并监听它的值的变化
  @Prop @Watch('handleChange') hasShow:boolean = true
  // 定义一个名为childCom的状态变量，初始值为true
  @State childCom:boolean = true

  // 当hasShow发生变化时调用的方法
  handleChange():void{
     this.childCom = this.hasShow
  }

  // 构建组件的方法
  build() {
    Column(){
      // 显示当前监听的值
      Text(`当前监听的值是：${this.hasShow}`)
      if (this.childCom){
        // 如果childCom为true，则显示子组件的内容
        Text('这是子组件的内容')
      }
    }
  }
}
```

1.4　ArkTS语言之状态管理进阶

在1.3节中介绍道，装饰器仅能在页面内即一个组件树上，共享状态变量。如果开发者要实现应用级的或者多个页面的状态数据共享，就需要用到应用级别的状态管理的概念。

ArkTS拥有多种应用状态的管理能力，包括：

- LocalStorage：页面级UI状态存储，通常用于UIAbility内、页面间的状态共享。
- AppStorage：特殊的单例LocalStorage对象，由UI框架在应用程序启动时创建，为应用程序UI状态属性提供中央存储。
- PersistentStorage：持久化存储UI状态，通常和AppStorage配合使用，选择AppStorage存储的数据写入磁盘，以确保这些属性在应用程序重新启动时的值与应用程序关闭时的值相同。
- Environment：应用程序运行的设备的环境参数，同步到AppStorage中，可以和AppStorage搭配使用。

1.4.1 LocalStorage：页面级UI状态存储

LocalStorage是页面级的UI状态存储，通过@Entry装饰器接收的参数可以在页面内共享同一个LocalStorage实例。LocalStorage支持UIAbility实例内多个页面间的状态共享。其具有如下特点：

- LocalStorage是ArkTS提供的内存“数据库”，用于存储页面级别的状态变量。
- 应用程序可以创建多个LocalStorage实例，并支持在页面内和跨页面、UIAbility实例间共享。
- 根组件（@Entry装饰的@Component）可分配LocalStorage实例，其子组件自动获得访问权限。
- 每个组件最多访问一个LocalStorage实例和AppStorage，未标记@Entry的组件只能接收来自父组件的实例。
- LocalStorage属性是可变的，其生命周期由应用程序管理，垃圾回收由JS Engine负责。
- 提供了@LocalStorageProp和@LocalStorageLink两个装饰器，分别用于建立单向和双向同步关系。

1. @LocalStorageProp装饰器（单向同步）

我们可以通过以下几个步骤来创将一个简单的单向同步实例。

01 使用构造函数创建LocalStorage实例storage，通过setOrCreate('key',value)来设置存储的value（值），同时通过key来读取存储的值，代码如下：

```
let storage = new LocalStorage();
storage.setOrCreate('PropA',47);
```

02 使用@Entry装饰器将storage添加到顶层组件中，以“@LocalStorageProp(key) xxx:类型=值”的形式读取存储的值，其中xxx是自定义的变量名称，代码如下：

```
@Entry(storage)
@Component
struct Demo1 {
  @LocalStorageProp('PropA') localStorLink: number = 1
  build() {
    Column({ space: 15 }) {
      // 单击后从47开始加1，只改变当前组件显示的localStorLink，不会同步到LocalStorage中
      Button(`Parent from LocalStorage ${this.localStorLink}`)
        .onClick(() => this.localStorLink += 1)
      Demo2()
    }
  }
}
```

03 创建一个组件Demo2()用作参考，代码如下：

```
@Component
export struct Demo2 {
  // @LocalStorageProp变量装饰器与LocalStorage中的'PropA'属性建立单向绑定
  @LocalStorageProp('PropA') storProp2: number = 2;
  build() {
    Column({ space: 15 }) {
      // 当Demo1改变时，当前storProp2不会改变，显示47
      Text(`Parent from LocalStorage ${this.storProp2}`)
    }
  }
}
```

@LocalStorageProp装饰器（单向同步）效果如图1-51所示。

通过效果图可以看到，父子组件均可以读取到存储的数据，但是当父组件的数据被修改时，子组件的数据并没有发生改变，或者说通过storage.setOrCreate存储的数据并不会发生改变。注意，需要将storage 添加到顶层组件@Entry(storage)中，否则无法读取到存储的数据。

图1-51　@LocalStorageProp装饰器（单向同步）

2. @LocalStorageLink装饰器（双向同步）

双向同步的案例可直接使用@LocalStorageProp装饰器中的案例，只需将对应的@LocalStorageProp装饰器改为@LocalStorageLink即可。同时为了展示双向同步的效果，需要对子组件进行简单的修改。子组件内容如下：

```
@Component
struct demo2{
  // @LocalStorageLink 与 LocalStorage中的'PropA'属性建立双向绑定
  @LocalStorageLink('PropA') storageLinkTwo : number = 1;

  build(){
    Button('Child from LocalStorage ${this.storageLinkTwo}')
      // 更改将同步至LocalStorage中的'PropA'以及Parent.storageLinkOne
      .onClick(()=> this.storageLinkTwo += 1)
  }
}
```

完整代码如下：

```
let storage = new LocalStorage();
storage.setOrCreate( 'PropA', 47 )
@Component
struct demo2{
  // @LocalStorageLink 与 LocalStorage中的'PropA'属性建立双向绑定
  @LocalStorageLink('PropA') storageLinkTwo : number = 1;

  build(){
    Button(`Child from LocalStorage ${this.storageLinkTwo}`)
      // 更改将同步至LocalStorage中的'PropA'以及Parent.storageLinkOne
      .onClick(()=> this.storageLinkTwo += 1)
  }
}

// 使LocalStorage可从@Component组件访问
@Entry(storage)
@Component
```

```
struct demo1 {
  // @LocalStorageLink 与 LocalStorage中的'PropA'属性建立双向绑定
  @LocalStorageLink('PropA') storageLinkOne : number = 1

  build() {
    Column({ space: 15 }) {
      Button(`Parent from LocalStorage ${this.storageLinkOne}`)
        .onClick(() => this.storageLinkOne += 1)
      // @Component子组件自动获得对LocalStoragePager LocalStorage实例的访问权限
      demo2()
    }
  }
}
```

此时运行代码，会发现无论是子组件的按钮，还是父组件的按钮，在触发时均会同步修改数据，也就是说@LocalStorageLink装饰器是可以和存储的数据进行双向同步的。

1.4.2　AppStorage：应用全局的UI状态存储

通过1.4.1节的学习，我们知道LocalStorage是一个页面级别的UI状态存储，同时通过@LocalStorageLink和@LocalStorageProp分别实现数据的双向同步和单向同步，本小节将讲解另一个状态存储AppStorage（全局的UI状态存储）。AppStorage也拥有两个装饰器，是@StorageProp和@StorageLink，分别对应LocalStorage中的@LocalStorageProp 和@LocalStorageLink，可以与存储的数据建立单向数据传递和双向数据传递。

AppStorage是和应用的进程绑定的，由UI框架在应用程序启动时创建，为应用程序UI状态属性提供中央存储。

1. @StorageProp装饰器（单向同步）

我们可以通过以下几个步骤来创建一个简单的单向同步实例。

01 创建一个UI内部的状态存储，代码如下：

```
AppStorage.SetOrCreate('Prop',60)
```

02 在入口组件中通过“@StorageProp('Prop')　xxx：类型 = 值”的方式来接收存储的数据，在UI界面上编写按钮来展示数据，当单击按钮时修改数据，代码如下：

```
@Entry
@Component
struct StoragePropPage {
  @StorageProp('Prop') storagePropOne: number = 1;
  build() {
    Column({ space: 15 }) {
      // 单击后从60开始加1，只改变当前组件显示的storagePropOne，不会同步到AppStorage中
      Button(`Parent from AppStorage ${this.storagePropOne}`).onClick(() =>
this.storagePropOne += 1)
      ChildStorageProp()
    }
  }
}
```

03 添加一个子组件ChildStorageProp，用于观察父组件在修改数据时是否为单向同步，代码如下：

```
@Component
struct ChildStorageProp {
```

```
    // @LocalStorageProp变量装饰器与LocalStorage中的'Prop'属性建立单向绑定
    @StorageProp('Prop') storagePropTwo: number = 2;

    build(){
      Column({ space: 15 }) {
        // 当StoragePropPage改变时，当前storagePropTwo不会改变，显示60
        Text(`Parent from AppStorage ${this.storagePropTwo}`)
      }
    }
  }
```

代码运行效果如图1-52所示。

图1-52　@StorageProp装饰器

通过效果图可以看到，父子组件均可以读取到存储的数据，但是当父组件的数据被修改时，子组件的数据并没有发生改变，或者说通过StorageProp装饰器读取的数据是单向的。

完整代码如下：

```
  AppStorage.SetOrCreate('Prop',60)

  @Entry
  @Component
  struct StoragePropPage {
    @StorageProp('Prop') storagePropOne: number = 1;

    build() {
      Column({ space: 15 }) {
        // 单击后从60开始加1，只改变当前组件显示的storagePropOne，不会同步到AppStorage中
        Button(`Parent from AppStorage ${this.storagePropOne}`).onClick(() =>
this.storagePropOne += 1)
        ChildStorageProp()
      }
    }
  }

  @Component
  struct ChildStorageProp {
    @StorageProp('Prop') storagePropTwo: number = 2;

    build(){
      Column({ space: 15 }) {
        // 当StoragePropPage改变时，当前storagePropTwo不会改变，显示60
        Text(`Parent from AppStorage ${this.storagePropTwo}`)
      }
    }
  }
```

2. @StorageLink装饰器（双向同步）

双向同步的案例可以直接使用@StorageProp装饰器中的案例，只需将对应的@StorageProp装饰器改为@StorageLink即可。同时为了展示双向同步的效果，需要对子组件进行简单的修改。

子组件代码如下：

```
  @Component
  struct ChildStorageProp2 {
    @StorageLink('Prop') storagePropTwo: number = 2;

    build(){
      Column({ space: 15 }) {
```

```
          Button(`Parent from AppStorage ${this.storagePropTwo}`).onClick(() =>
this.storagePropTwo += 1)

        }
      }
    }
```

完整代码如下：

```
    AppStorage.SetOrCreate('Prop',60)

    @Entry
    @Component
    struct StoragePropPage1 {
      @StorageLink('Prop') storagePropOne: number = 1;
      build() {
        Column({ space: 15 }) {
          Button(`Parent from AppStorage ${this.storagePropOne}`).onClick(() =>
this.storagePropOne += 1)
          ChildStorageProp2()
        }
      }
    }
    @Component
    struct ChildStorageProp2 {
      @StorageLink('Prop') storagePropTwo: number = 2;
      build(){
        Column({ space: 15 }) {
          Button(`Parent from AppStorage ${this.storagePropTwo}`).onClick(() =>
this.storagePropTwo += 1)
        }
      }
    }
```

此时运行代码，会发现无论是子组件的按钮还是父组件的按钮，在触发时均会同步修改数据，也就是说@StorageLink装饰器是可以和存储的数据进行双向同步的

无论是1.4.1节中的LocalStorage页面级UI状态存储，还是1.4.2节中的AppStorage应用全局的UI状态存储，均在UI内部使用，而非应用逻辑内，这一知识点会在后面的章节中进行讲解。

1.4.3　PersistentStorage：持久化存储UI状态

前两节介绍的LocalStorage和AppStorage都是运行时的内存，但是在应用退出并再次启动后，依然能保存选定的结果，是应用开发中十分常见的现象，这就需要用到PersistentStorage。PersistentStorage的特性主要有以下几点：

- PersistentStorage将选定的AppStorage属性保留在设备磁盘上。
- 应用程序通过API来决定哪些AppStorage属性应借助PersistentStorage持久化。
- UI和业务逻辑不直接访问PersistentStorage中的属性，所有属性访问都是对AppStorage的访问，AppStorage中的更改会自动同步到PersistentStorage。
- PersistentStorage和AppStorage中的属性建立双向同步。应用开发通常通过AppStorage访问PersistentStorage。

需要注意的是，PersistentStorage支持number、string、boolean、enum等简单类型和可JSON化的对

象，但不包括Date、Map、Set等复杂内置类型和对象的属性方法；不支持嵌套对象、undefined和null。应避免持久化大型数据集和经常变化的变量，且数据量应控制在2KB以内，以避免影响UI性能。大量数据存储建议使用数据库。PersistentStorage仅限UI页面内使用。

我们通过以下几个步骤来实现从AppStorage中访问PersistentStorage初始化的属性。

01 初始化PersistentStorage：

```
PersistentStorage.PersistProp('Aprop', 30);
```

02 在组件内部通过StorageLink来获取对应的属性：

```
@StorageLink('Aprop') aProp: number = 1
```

完整代码如下：

```
PersistentStorage.PersistProp('Aprop', 30);

@Entry
@Component
struct PersistentStoragePage {
  @StorageLink('Aprop') aProp: number = 1

  build() {
    Row() {
      Column() {
        // 应用退出时会保存当前结果。重新启动后，会显示上一次的保存结果
        Button(`当前值${this.aProp}`).onClick(() => this.aProp += 1)

      }
    }
  }
}
```

该案例与其他的存储相比，区别在于应用退出时会保存当前结果，重新启动后会显示上一次的保存结果。

1.5　ArkTS语言之动态构建UI元素

本节将会介绍一些新的装饰器组件，这些装饰器将会使我们的开发更加方便、快捷。

1.5.1　@Builder

1. @Builder是什么

在开发的过程中，当页面中有多个相同的UI结构时，如果每个都单独声明，会造成大量的重复代码。为了避免产生重复的代码，我们可以将相同的UI结构提炼为一个自定义组件，用于完成UI结构的复用。

除了上述方法之外，ArkTS 还提供了一种更轻量的UI结构复用机制——@Builder装饰器。开发者可以将重复使用的UI元素抽象成一个@Builder方法，该方法可在build()方法中多次调用，以完成UI结构的复用。

2. @Builder的使用

@Builder的使用分为组件内的使用和全局的使用，下面通过两个例子来进行讲解。

1）@Builder 在组件内的使用

@Builder在组件内使用的语法是：

- 定义结构时要放在struct的里面进行定义，也就是在组件内定义。
- 定义的语法是：@Builder xxxx(){ ... }，其中xxxx是自定义的名称。
- 在组件内使用“this.自定义名称”的方式进行使用，如this.xxxx()。

示例代码如下：

```
@Entry
@Component
struct Demo1_5_1Builder_in {
  @State message: string = 'Hello World';

  build() {
    Column() {
      Row({ space: 50 }) {
        //复用UI结构
        this.compButtonBuilder('编辑', () => console.log('编辑'))
        this.compButtonBuilder('发送', () => console.log('发送'))
      }
    }.width('100%')
    .height('100%')
    .justifyContent(FlexAlign.Center)
  }

  //定义UI结构
  @Builder compButtonBuilder(text: string, callback: () => void) {
    Button() {
      Row({ space: 10 }) {
        Text(text)
          .fontColor(Color.White)
          .fontSize(25)
      }
    }.width(120)
    .height(50)
    .onClick(callback)
  }
}
```

在示例代码中，在build的同级中使用@Builder来装饰一个自定义的函数名称，用来定义UI结构，自定义函数接收两个参数，text和事件回调；在build组件中使用“this.自定义函数名”的方式来复用我们定义的UI结构。

效果如图1-53所示。

图1-53　@Builder在组件内的使用

2）@Builder 在全局的使用

@Builder在全局的使用与在UI组件内部的使用是有一些区别的，主要区别如下：

- 定义结构时要放在struct的外面进行定义，也就是在全局定义。
- 定义的语法是：@Builder function xxxx(){}，其中xxxx是自定义的名称。
- 直接在组件内使用自定义名称即可，如xxxx()。

示例代码如下：

```
@Entry
@Component
struct Demo1_5_1Builder_global {
  @State message: string = 'Hello World';

  build() {
    Column() {
      Row({ space: 50 }) {
        //复用UI结构
        globalButtonBuilder( '编辑', () => console.log('编辑'))
        globalButtonBuilder('发送', () => console.log('发送'))
      }
    }.width('100%')
    .height('100%')
    .justifyContent(FlexAlign.Center)
    }
}

//定义UI结构
@Builder function globalButtonBuilder( text: string, callback: () => void) {
  Button() {
    Row({ space: 10 }) {
      Text(text)
        .fontColor(Color.White)
        .fontSize(25)
    }
  }.width(120)
  .height(50)
  .onClick(callback)
}
```

在上述示例代码中，在struct的外面使用@Builder function xxxx(){}来定义UI结构，自定义函数接收两个参数，text和事件回调；在build的组件中使用自定义函数名的方式来复用我们定义的UI结构。

效果如图1-53所示。

1.5.2　@BuilderParam

@BuilderParam用于装饰自定义组件（struct）中的属性，装饰的属性可作为一个UI结构的占位符，待创建该组件时，可通过参数为其传入具体的内容。其作用类似于Vue框架中的槽（slot）。

下面通过几个步骤来进一步了解@BuilderParam装饰器的用法。

01　定义一个组件，使用@BuilderParam占位，代码如下：

```
@Component
struct Container {
  //@BuilderParam属性
  @BuilderParam content: () => void
  build() {
    Column() {
      Text('其他内容')                //其他内容
      this.content();                 //占位符
      Button('其他内容')              //其他内容
    }
  }
}
```

02 使用@Builder装饰器来定义UI结构，代码如下：

```
@Builder function autoUI1() {
  Text('这是自定义插槽1')
}

@Builder function autoUI2() {
  Text('这是自定义插槽2')
}
```

03 在组件中通过传入不同的UI来填充结构，代码如下：

```
Container({content:autoUI1})
Divider().strokeWidth(20)
Container({content:autoUI2})
```

完整代码讲解如图1-54所示。

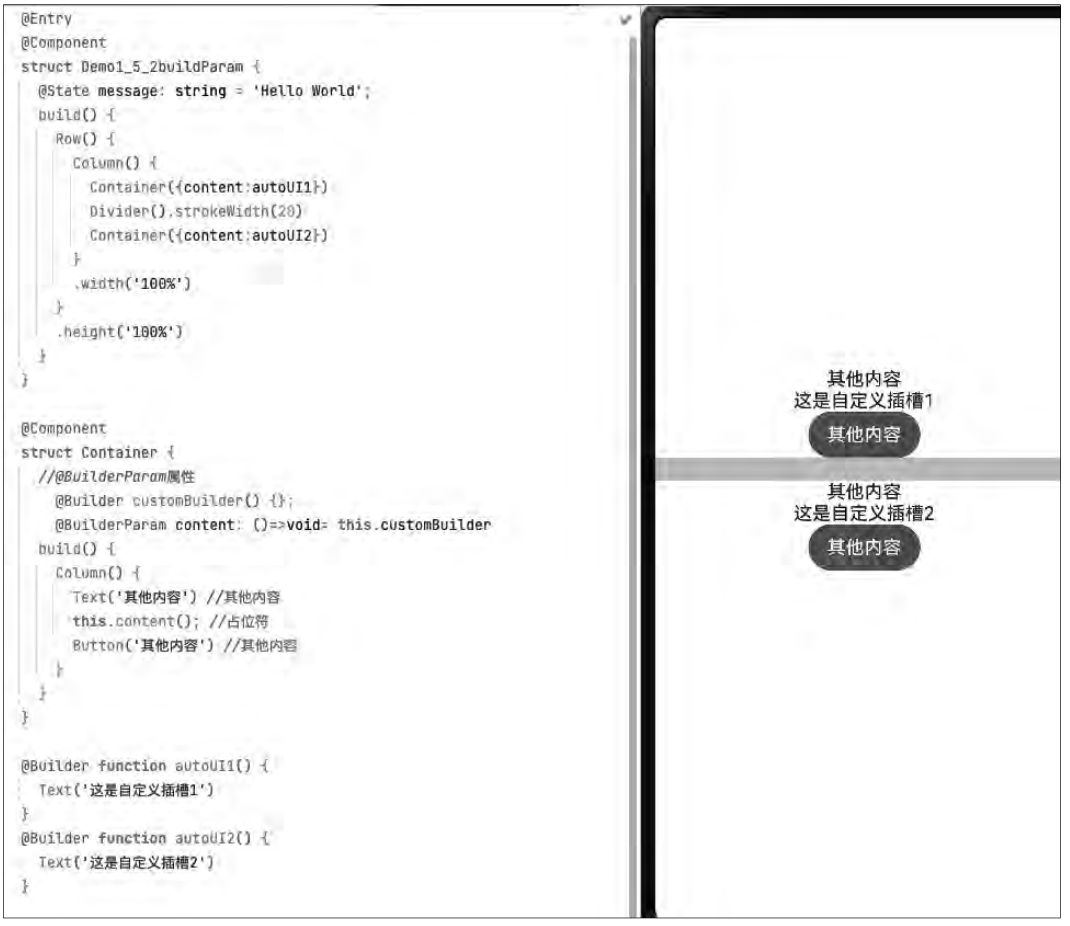


图1-54　@BuilderParam装饰器

1.5.3　@Styles

在开发过程中，不可避免地要写一些样式来设置元素的展示形态，比如字体的粗细、图片的大小等，但是会有很多重复的样式。面对这些重复的样式，我们没有必要去粘贴复制。为了代码的简洁性和后续维护的便利性，可以提炼出一些公共样式进行复用，这时就会用到@Styles装饰器了。

在使用@Styles装饰器之前，需要明确以下几点：

- @Styles装饰器仅支持通用属性和通用事件。
- @Styles方法不支持参数。
- @Styles可以定义为全局，也可以定义在组件内，定义成全局时需要在方法名前面添加function关键字，组件内则不需要添加function关键字。
- @Styles关键字不支持export导出，只能在当前文件内使用。

@Styles装饰器的使用方法如下：

```
// 全局
@Styles function functionName() { ... }

//在组件内
@Component
struct FancyUse {
  @Styles fancy() {
    .height(100)
  }
}
@Styles装饰器使用案例
```

@Styles装饰器的示例代码如下：

```
// 定义一个全局的通用样式
@Styles function globalStyles(){
    .width(300)
    .height(200)
    .backgroundColor('#fff000')
    .margin(10)
    .padding(10)
}

@Entry
@Component
struct Demo1_5_3styles {
  // 定义组件内的通用样式
  @Styles myStyles(){
    .width(300)
    .height(200)
    .backgroundColor('#000fff')
    .margin(10)
    .padding(10)
  }
  build() {
      Column() {
      // 使用全局样式
        Text('这是全局样式')
          .globalStyles()
          .fontSize(20)
      // 组件内的样式
        Text('组件内的样式')
          .myStyles()
          .fontColor('#ffffff')
      }
      .width('100%')
```

```
      .height('100%')
    }
  }
```

在上述示例代码中，分别定义了全局的Styles函数封装和组件内的函数封装。无论是全局的还是组件内的，只需在函数中使用“.属性名：属性”的格式即可。在使用过程中，直接将其当作普通的属性使用即可，同时也可以为当前的UI组件添加其他的一些样式。

效果如图1-55所示。

1.5.4　@Extend

在1.5.3节中讲解了@Styles装饰器，我们可以使用@Styles进行样式的封装，对相同的样式进行重用。本小节介绍的@Extend装饰器的作用主要是扩展原生组件的样式。

@Extend装饰器的使用语法如下：

```
@Extend(ui组件名称) function functionName { ... }
```

在使用@Extend装饰器时，需要注意以下几点：

- @Extend装饰器仅支持在全局定义，不支持在组件内定义。
- @Export只能在当前文件内使用，不支持export的导出。
- @Extend支持封装指定组件的私有属性、私有事件和自身定义的全局方法。
- @Extend装饰的方法支持参数，可以在调用时传递参数，调用遵循TypeScript方法传值调用。

图1-55　@Styles装饰器

@Extend装饰器的示例代码如下：

```
@Entry
@Component
struct Demo1_5_4Extend {
  build() {
    Row({ space: 10 }) {
      Text('商品')
        .TextExtendStyle(100, Color.White, 20)
      Text('harmonyos')
        .TextExtendStyle(200, Color.Pink,30)
      Text('单框架鸿蒙')
        .TextExtendStyle(300, Color.Orange,40)
    }
  }
}

@Extend(Text) function TextExtendStyle(weight:number, color:Color, fontSize:number) {
   .fontWeight(weight)
  .fontColor(color)
  .fontSize(fontSize)
  .backgroundColor('#000000')
}
```

在上述示例代码中，3个Text UI组件拥有共同的样式属性，只是每个属性的值不同。此时我们可以将公共的样式属性放到@Extend中，并通过传参的方式来设置每个属性的值。这样可使代码更加整洁，同时也减少了代码量