

实验十二 自定义组件基础

一、实验目的

1. 了解 DevEco Studio 的使用
2. 学习 ArkTS 语言，掌握自定义组件
3. 编写代码
4. 编译运行
5. 在模拟器上运行

二、实验原理

1. 鸿蒙开发原理
2. ArkTS, ArkUI 开发原理
3. 鸿蒙应用运行原理

三、实验仪器材料

1. 计算机实训室电脑一台
2. DevEco Studio 开发环境及鸿蒙手机模拟器

四、实验步骤

1. 打开 DevEco Studio，点击 Create Project 创建工程。

配置好项目名称（如 MyComponents），存放位置（上图是放在 D 盘某个目录下），设备类型等，然后点击 Finish 按钮，进入到开发界面。

项目创建成功。然后，清理代码，找到 `entry > src > main > ets > pages` 里面的 `Index.ets` 文件，将 `build() {}` 的 `{}` 里面的代码清空。

2. 自定义组件基础

我们在前面用到的组件基本上都是由系统框架提供的，我们称之为系统组件，比如 Column, Row 等容器组件，Text, Button 等基础组件；这一节开始我们来自定义组件，也就是由开发者自己定义的组件。

在鸿蒙系统中，可以通过 @Component 注解来创建自定义组件

@Component 官方文档：

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/arkts-create-custom-components>

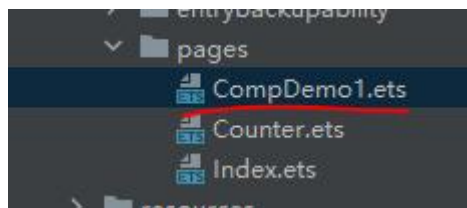
@Component 装饰器仅装饰 struct 关键字声明的数据结构。被装饰的 struct 具备组件化的能力，需要实现 build 方法描述 UI，一个 struct 只能被一个 @Component 装饰。@Component 可以接受一个可选的 boolean 类型参数。

语法：

```
@Component  
  
struct MyComponent {  
  
}
```

自定义组件的定义很简单，和页面组件的唯一区别就是没有 @Entry。我们通过实例来理解。

创建一个新的文件 CompDemo.ets：



加入代码：

```
@Entry  
  
@Component  
  
struct CompDemo1 {  
  
  build() {  
  
    Column() {
```

```
    }  
  }  
}
```

在@Entry 的上面我们来定义一些自定义组件，首先加入代码：

```
@Component  
  
struct Header {  
  
  build() {  
  
    Row() {  
  
      Text('头部区域').fontColor(Color.White)  
  
    }.width('100%')  
  
    .height(60)  
  
    .backgroundColor(Color.Green)  
  
  }  
  
}
```

然后，我们在页面的 build 中加入红色字体那一行：

```
@Entry  
  
@Component  
  
struct CompDemo1 {  
  
  build() {  
  
    Column() {  
  
      Header()  
  
    }  
  
  }  
  
}
```

```
}  
  
}
```

我们自定义了一个 Header 组件，然后在页面中使用了这个子组件。

效果：



再加入代码：

```
@Component  
struct ContentPiece {  
  build() {  
    Row({space: 20}) {  
      Text('我是子组件内容')  
      Button('Click Me')  
    }.width('100%')  
    .height(30)  
    .justifyContent(FlexAlign.Center)  
  }  
}
```

```

@Component

struct ContentInfo {

  build() {

    Column({space: 10}) {

      ContentPiece()

      ContentPiece()

      ContentPiece()

    }.width('100%')

    .height(400)

    .justifyContent(FlexAlign.Center)

    .backgroundColor(Color.Pink)

  }

}

```

注意，这里我们用到了两个子组件，其中一个引用了另外一个。

然后，在页面的 build 中加入红色字体那一行：

```

@Entry

@Component

struct CompDemo1 {

  build() {

    Column() {

      Header()

```

```

ContentInfo()

}

}

}

```

效果：



类似地，我们再添加一个底部区域子组件：

```

@Component

struct Footer {

  build() {

    Row() {

```

```

        Text('底部区域').fontColor(Color.White)

      ).width('100%')

      .height(60)

      .backgroundColor(Color.Blue)

    }

  }
}

```

然后请自己添加调用代码。效果：



如果我们把 ContentInfo 的高度：

```

      .height(400)

```

改为：

```

      .layoutWeight(1)

```

请尝试查看效果。

这样，我们通过几个子组件，让页面布局的主程序变得很简洁清晰。

子组件里面可以定义自己的状态变量，而这个状态变量只对当前的子组件起作用。比如，我们针对 `ContentPiece`，加一个 `count` 变量以及按钮的 `onClick` 事件，代码变为：

```
@Component
struct ContentPiece {
  @State count: number = 0

  build() {
    Row({space: 20}) {
      Text(this.count.toString())

      Button('Click Me').onClick() => {
        this.count++
      })

    }.width('100%')
    .height(30)
    .justifyContent(FlexAlign.Center)
  }
}
```

效果：

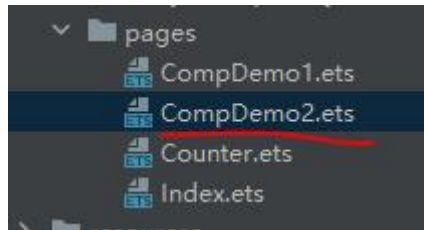


可以看到，点击按钮，只会影响当前的子组件，因为只是当前的子组件的状态变量发生了变化。

3. 自定义组件的通用样式和通用事件

同样，自定义组件也可以通过点语法，设置通用样式，触发通用事件。

我们通过实例来尝试。先创建一个新的文件 CompDemo2.ets:



先创建一个自定义子组件并使用，代码：

```
@Component
struct SubComp {
    build() {
        Row({space: 10}) {
            Text('子组件')
            Button('点击')
        }.width(200)
        .height(50)
        .backgroundColor(Color.Pink)
        .margin(10)
    }
}
```

```
@Entry

@Component

struct CompDemo2 {

    build() {

        Column() {

            SubComp()

        }

    }

}
```

此时我们要注意，我们在子组件内部给 Row() 设定了一些通用的属性，但是在调用这个子组件的时候，使用默认的 SubComp() 设置，没有设定属性。

效果：



我们再给 SubComp()添加一些通用属性，红色字体部分：

```
@Entry

@Component

struct CompDemo2 {

    build() {

        Column() {

            SubComp().width(300)

        }

    }

}
```

```

        .height(80)

        .backgroundColor(Color.Gray)

        .margin(5)
    }
}
}

```

此时效果：



注意，SubComp() 的属性并不会影响到子组件内部设定的属性。我们再给 SubComp() 添加一个通用点击事件，红色字体部分：

```

SubComp().width(300)

        .height(80)

        .backgroundColor(Color.Gray)

        .margin(5)

        .onClick()=> {

            AlertDialog.show({

                message: 'Hello, I am Clicked.'

            })

        })
}

```

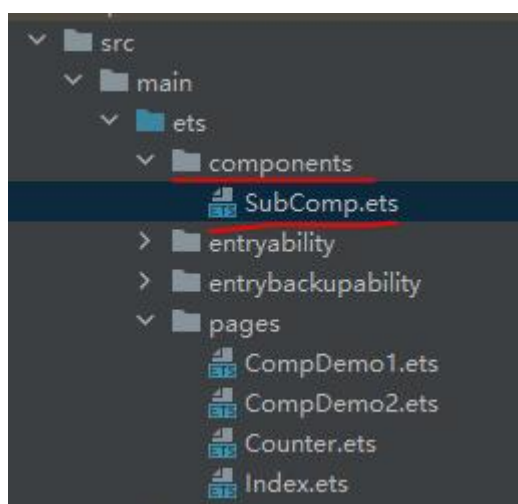
可以看到，只要点击整个灰色背景部分区域内部的任何地方（包括粉红色区域），都会有弹窗出现：



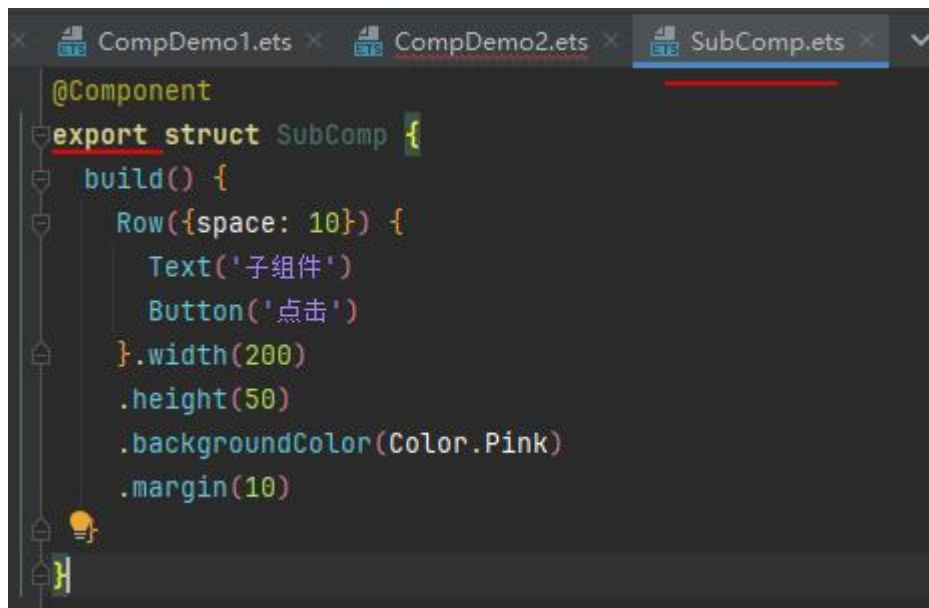
也就是说，点击事件是针对整个子组件的。相当于是在内部定义的外面还可以再包一层。

我们在前面的例子中，子组件的定义和页面的定义都在一个文件里面。实际项目中，通常的做法是将子组件定义在单独的文件中，导出子组件，然后在需要用到的地方导入即可。

比如，我们在 `main > ets` 下创建一个目录 `components`，在其中创建一个文件 `SubComp.ets`：

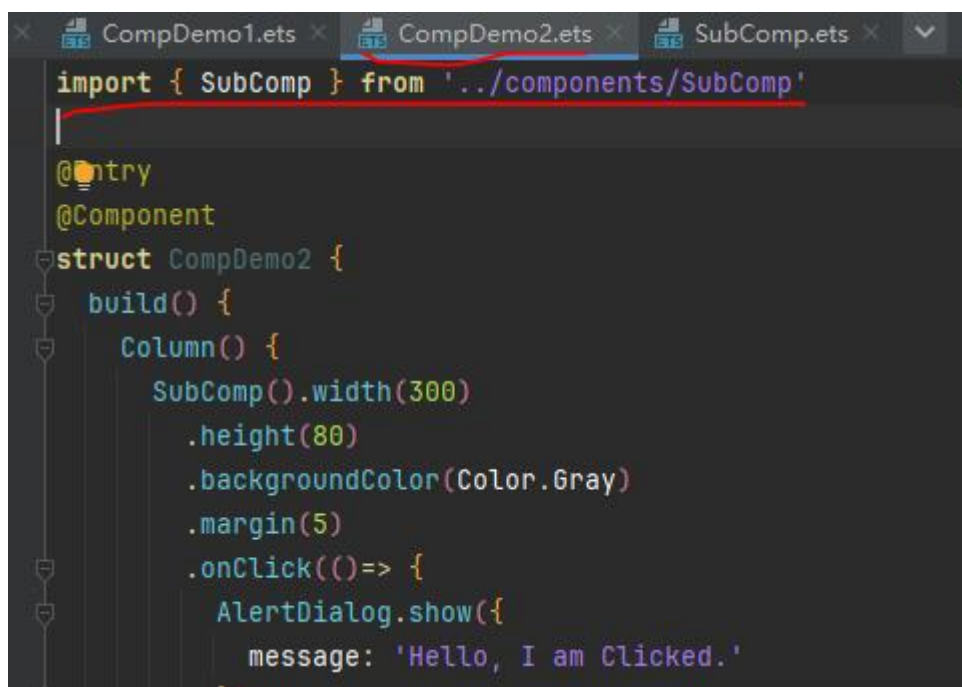


将上面例子中的 `SubComp` 子组件代码移到 `SubComp.ets` 中，然后，在 `struct` 前面加上 `export`：

A screenshot of an IDE window showing the code for SubComp.ets. The file is selected in the tab bar. The code defines a component structure SubComp with a build method that contains a Row of Text and Button components, with various styling properties like width, height, backgroundColor, and margin.

```
@Component
export struct SubComp {
  build() {
    Row({space: 10}) {
      Text('子组件')
      Button('点击')
    }.width(200)
    .height(50)
    .backgroundColor(Color.Pink)
    .margin(10)
  }
}
```

然后，在 CompDemo2.ets 中，导入 SubComp：

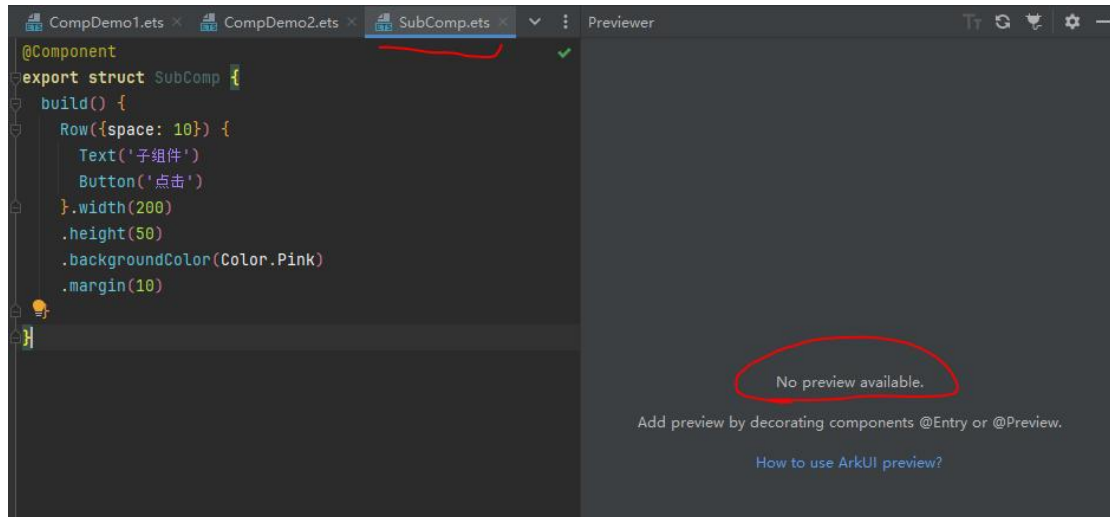
A screenshot of an IDE window showing the code for CompDemo2.ets. The file is selected in the tab bar. The code imports SubComp from a relative path and defines a component structure CompDemo2 with a build method that contains a Column of SubComp and an AlertDialog.

```
import { SubComp } from '../components/SubComp'

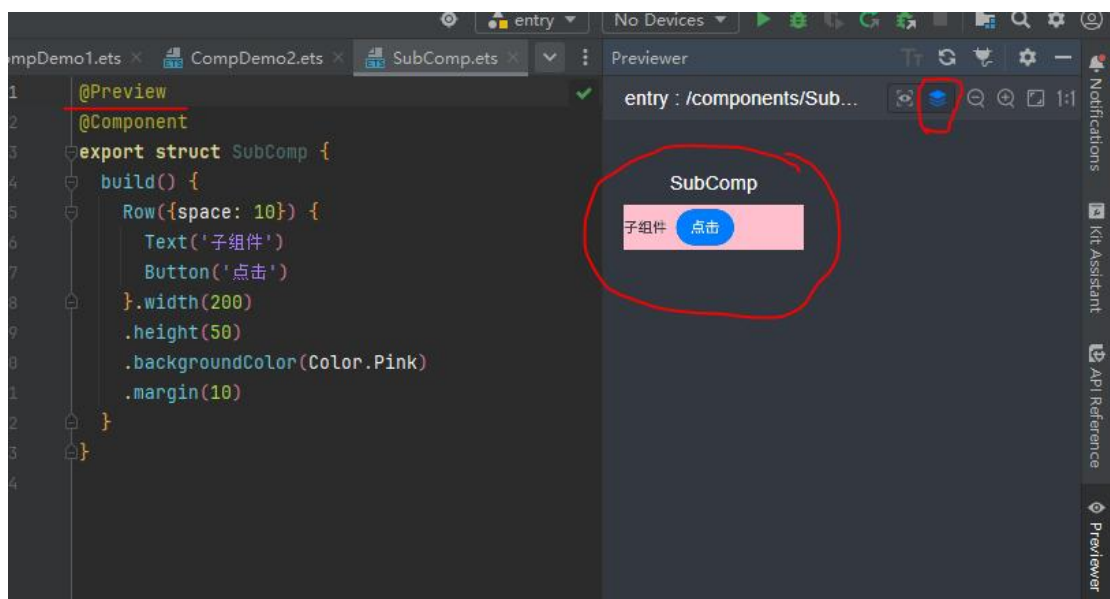
@Entry
@Component
struct CompDemo2 {
  build() {
    Column() {
      SubComp().width(300)
        .height(80)
        .backgroundColor(Color.Gray)
        .margin(5)
      .onClick(()=> {
        AlertDialog.show({
          message: 'Hello, I am Clicked.'
        })
      })
    }
  }
}
```

刷新 Previewer，效果是一样的。

注意，如果此时我们让编辑器处在 SubComp.ets 文件编辑状态，刷新 Previewer，会得到：



对于子组件，我们在设计的时候，其实也是很需要有预览的能力的。那么，对于组件，如何才能预览呢？答案是只需要在`@Component` 前面加上`@Preview` 即可：



此时，在 Previewer 里面可以看到子组件的预览效果，同时，在预览类型那里是“组件”而不是页面（右上角红色圆圈）。

`@Preview` 装饰器

官方文档：

<https://developer.huawei.com/consumer/cn/doc/harmonyos-references/ts-universal-component-previewer#preview%E8%A3%85%E9%A5%B0%E5%99%A8>

`@Preview` 装饰器用来装饰自定义组件，从而实现组件预览。

华为鸿蒙开发中 @Builder 和 @Component 有什么区别和联系？什么时候该用哪个？

特点	@Component（自定义组件）	@Builder（自定义构造函数）
主要用途	封装完整的、具有业务逻辑和状态管理的组件。	封装可复用的、轻量级的 UI 结构（代码片段）。
状态变量	可以定义自己的状态变量（@State, @Prop 等）。	不能定义自己的状态变量。数据交互通常通过参数传递或访问所属组件的状态变量。
生命周期	支持生命周期函数（onAppear, onDisappear 等）。	不支持生命周期函数。
层级关系	通常用于构建较高层级的组件，可以包含复杂的 UI 结构。	通常用于组件内部或全局，复用较小的 UI 结构片段。
调用方式	像基础组件一样，直接在 build() 方法中作为标签调用，例如 <MyComponent/>。	作为函数或方法调用，直接写在 build() 方法的 UI 语句块内，例如 MyBuilderFunction() 或 this.myBuilderMethod()。

4. 何时该用哪个？

使用 @Component 的场景：

- 构建一个独立、可复用的 UI 单元，例如一个自定义的按钮、列表项、对话框、或整个页面。
- 你需要管理组件内部的状态，当状态变化时，组件的 UI 需要自动刷新。
- 你需要管理组件的生命周期，例如在组件出现或消失时执行特定的逻辑（如数据加载、资源清理）。
- 组件结构复杂，包含较多的业务逻辑。

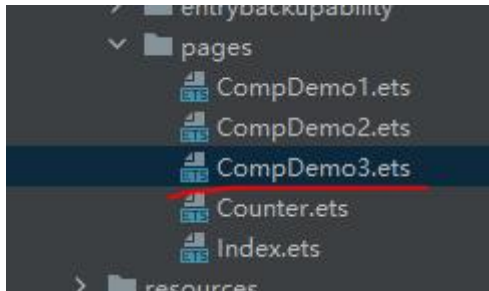
使用 @Builder 的场景:

- 你想要复用一个没有自身状态的 UI 代码片段, 例如:
- 一个列表项中重复的“图标+文本”结构。
- 多个组件中样式和布局相同的一个页眉或页脚部分。
- 根据不同条件渲染的 UI 块。
- 你只是想将 build() 方法中的重复布局代码抽象出来, 使 build() 方法更简洁、可读。
- 轻量级的 UI 封装, 不涉及复杂的状态管理和业务逻辑。

5. 自定义组件的成员函数和成员变量

对于自定义组件, 除了必须要实现的 build() 函数之外, 还可以定义其他的成员函数以及成员变量。成员变量 (包括数据和函数, 即变量可以是函数变量) 的值可以通过外部传参实现覆盖。

我们通过实例来尝试。先创建一个新的文件 CompDemo3.ets:



添加基础代码, 为了方便, 没有将子组件放到另外的文件中, 当然如果要预览, 可以在

@Component 前面加上@Preview:

```
@Component
struct MyCard {
    build() {
        Column() {
            Row() {
                Text('我的订单').fontSize(18)
            }
        }
    }
}
```

```
        Text('查看更多>').fontSize(18)
    }

    .width('100%')

    .justifyContent(FlexAlign.SpaceBetween)

    Row({space: 20}) {

        Text('内容部分').fontSize(18)

        Button('按钮')

    }

    .padding(20)

}

.padding(10)

.width('100%')

.height(200)

.margin({ bottom: 20 })

.borderRadius(10)

.backgroundColor(Color.White)

}

}
```

```
@Entry
```

```
@Component
```

```
struct CompDemo3 {  
    build() {  
        Column() {  
            MyCard()  
            MyCard()  
        }  
        .width('100%')  
        .height('100%')  
        .backgroundColor('#ccc')  
        .padding(20)  
    }  
}
```

上面这些代码，相信同学们已经掌握了，效果：



在实际应用中，每个 Card 里面的标题，事件，内容可能是不同的，所以我们需要给子组件添加成员变量，然后在调用子组件的时候传入。

在 MyCard 子组件中添加成员变量，代码放在 build() 前面：

```
// 成员变量 - 数据
```

```
title: string = '默认的大标题'
```

```
extra: string = '查看更多 >'
```

```
// 成员变量 - 函数 - 可以外部传入覆盖的
```

```
getMore = (): void => {}
```

```
// 成员函数 - 不可以外部传入覆盖
```

```
sayHi() {
```

```
  AlertDialog.show({
```

```
    message: '打招呼, 你好'
```

```
  })
```

```
}
```

可以看到，我们可以定义一些数据类的成员变量并赋初值，也可以定义函数类的，通常函数类的往往定义为一个空的函数（也可以不为空）。也可以定义成员函数，有具体的实现，这种往往是无需外部传入的。区别在于，可传入的成员函数是写成箭头函数的形式。

在子组件的 build() 中使用这些成员变量和成员函数，注意红色字体部分的变化：

```
build() {
```

```
  Column() {
```

```
    Row() {
```

```
Text(this.title).fontSize(18)
```

```
Text(this.extra).fontSize(18)
```

```
.onClick() => {
```

```
  this.getMore()
```

```
})
```

```
}
```

```
.width('100%')
```

```
.justifyContent(FlexAlign.SpaceBetween)
```

```
Row() {
```

```
  Text('内容部分').fontSize(18)
```

```
  Button('按钮')
```

```
.onClick() => {
```

```
  this.sayHi()
```

```
})
```

```
}
```

```
.padding(20)
```

```
}
```

```
.padding(10)
```

```
.width('100%')
```

```
.height(200)
```

```
.margin({ bottom: 20 })
```

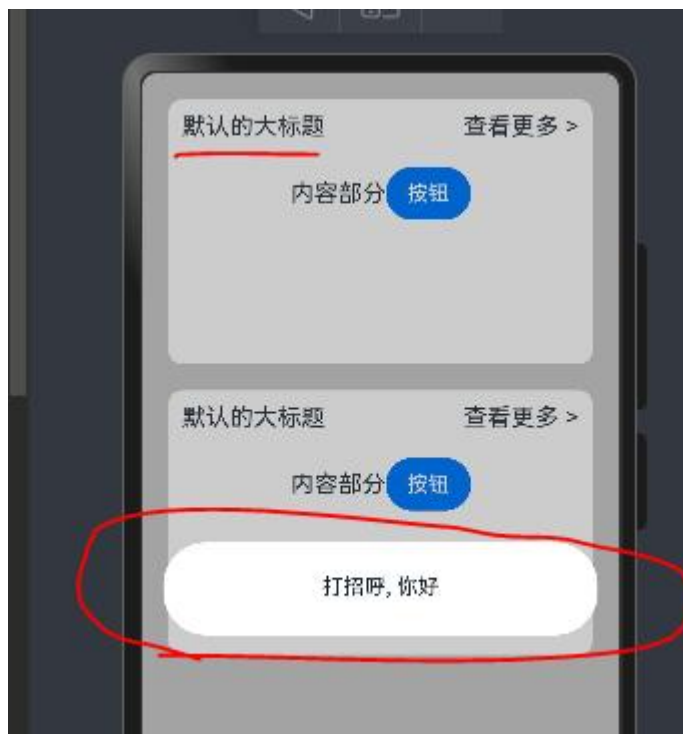
```
.borderRadius(10)
```

```
.backgroundColor(Color.White)
```

```
}
```

由于 title 的默认值是“默认的大标题”，所以 Card 上标题那里会有变化；而按钮现在的点击事件就是弹窗提示“打招呼，你好”，因此，点击任何一个子组件的按钮都一样。

此时效果：



现在，在页面组件中调用子组件的时候，我们尝试传入不同的值。如果我们在 MyCard 里面尝试去设定传入参数的时候，可以看到提示：



也就是说，系统已经帮我们在编辑器里找到了子组件可以设定的成员变量和函数。

代码：

```
build() {  
  Column() {  
    MyCard({  
      title: '广东培正学院的食堂',  
      extra: '看看吃什么?',  
      getMore: () => {  
        AlertDialog.show({  
          message: '中午吃什么总是很难选择'  
        })  
      })  
    }  
  })  
}
```

```
MyCard({
```

```
  title: '哪些运动适合我',
```

```
  extra: '除了睡懒觉>',
```

```
  getMore: () => {
```

```
    AlertDialog.show({
```

```
      message: '足球，篮球，排球， ...'
```

```
    })
```

```
  }
```

```
})
```

```
}
```

```
  .width('100%')
```

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

```
  .backgroundColor('#ccc')
```

```
  .padding(20)
```

```
}
```

效果：



可以看到，通过传入不同的成员变量和成员函数，可以对子组件进行灵活的配置。

当然，成员函数也是可以设一个“初值”，即缺省的执行内容的。比如，我们把 `getMore` 改为：

```
// 成员变量 - 函数 - 可以外部传入覆盖的  
  
getMore = (): void => {  
  
    AlertDialog.show({  
  
        message: '缺省的消息内容'  
  
    })  
  
}
```

内容不是空的。然后在页面调用的时候，我们把第二个 Card 改为：

```

53  @Entry
54  @Component
55  struct CompDemo3 {
56  build() {
57    Column() {
58      MyCard({
59        title: '广东培正学院的食堂',
60        extra: '看看吃什么?',
61        getMore: () => {
62          AlertDialog.show({
63            message: '中午吃什么总是很难选择'
64          })
65        }
66      })
67      MyCard({
68        title: '哪些运动适合我',
69        extra: '除了睡懒觉>',
70      })
71    }
72    .width('100%')

```

也就是去掉了 `getMore` 的传入。此时，点击“除了睡懒觉>”，得到的效果是：



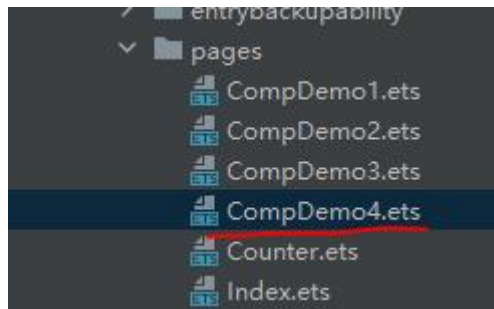
也就是调用了缺省的 `getMore` 函数执行的内容。

请自行尝试将上面的“内容部分”和按钮上的文字也改为成员变量。

6. @BuilderParam 传递 UI

对于自定义组件，可以利用@BuilderParam 构建函数，让外部 UI 传递进来。我们通过实例来尝试。

创建一个新的文件 CompDemo4.ets:



加入基础代码:

```
@Component
```

```
struct SubComp {
```

```
    build() {
```

```
    }
```

```
}
```

```
@Entry
```

```
@Component
```

```
struct CompDemo4 {
```

```
    build() {
```

```
        Column() {
```

```
            SubComp()
```

```
}
```

```
}
```

```
}
```

定义了一个自定义子组件 SubComp，在页面中使用了。此时 Previewer 是空白的。

在 SubComp 子组件中，利用 @Builder 定义一个构造函数，然后在 build() 中调用这个构造函数，子组件完整代码：

```
@Component
```

```
struct SubComp {
```

```
    @Builder
```

```
    defaultBuilder() {
```

```
        Column() {
```

```
            Text('默认的内容')
```

```
            Text('默认的内容')
```

```
            Text('默认的内容')
```

```
        }
```

```
    }
```

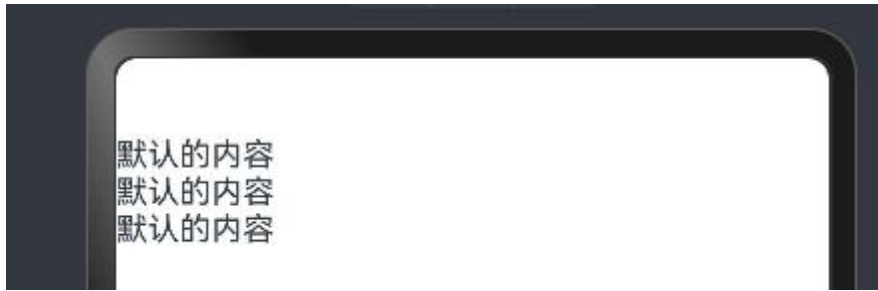
```
    build() {
```

```
        this.defaultBuilder()
```

```
    }
```

```
}
```

刷新 Previewer，效果：



对于这个@Builder 中的 UI 内容，能不能在调用子组件的时候进行改变呢？

答案是可以，此时就用到@BuilderParam，也就是相当于将这个@Builder 函数作为外部可传入的。

@BuilderParam 官网资料：

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/arkts-builderparam>

修改子组件代码，注意红色字体部分为新增或修改的：

```
@Component
```

```
struct SubComp {
```

```
    @Builder
```

```
    defaultBuilder() {
```

```
        Column() {
```

```
            Text('默认的内容')
```

```
            Text('默认的内容')
```

```
            Text('默认的内容')
```

```
        }
```

```
    }
```

```
    @BuilderParam ContentBuilder : () => void = this.defaultBuilder
```

```
    build() {
```

```

        this.ContentBuilder()

    }

}

```

定义一个@BuilderParam 函数变量,初始值为这个 defaultBuilder 函数,然后在外部可以传入。

现在,到页面的 build()那里去修改,注意红色字体部分:

```

@Entry

@Component

struct CompDemo4 {

    build() {

        Column() {

            SubComp() {

                Row() {

                    Button('New Button')

                    Text('New Content')

                }

            }

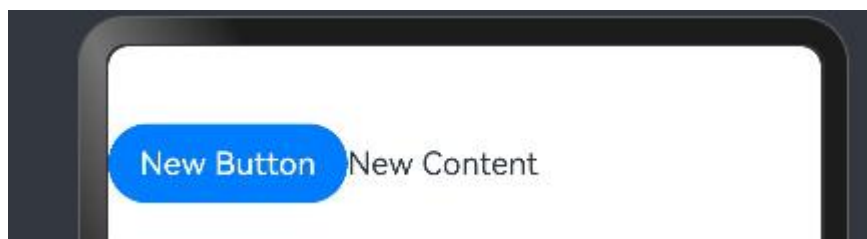
        }

    }

}

```

效果:



也就是说，在调用自定义子组件的时候，通过在后面用{}包起来的 UI 界面可以取代掉组件中的 defaultBuilder 里面的界面内容。

我们可以回到 **CompDemo3.ets**，把 Card 的内容部分用 @BuilderParam 来优化一下。

在子组件 MyCard 中加入定义：

```
@Builder
```

```
defaultBuilder() {
```

```
    Column() {
```

```
        Text('默认的内容')
```

```
    }
```

```
}
```

```
@BuilderParam ContentBuilder : () => void = this.defaultBuilder
```

在对应的位置调用这个 ContentBuilder():

```
build() {  
    Column() {  
        Row() {  
            Text(this.title).fontSize(18)  
            Text(this.extra).fontSize(18)  
                .onClick(() => {  
                    this.getMore()  
                })  
        }  
        .width('100%')  
        .justifyContent(FlexAlign.SpaceBetween)  
        Row() {  
            this.ContentBuilder()  
        }  
        .padding(20)  
    }  
    .padding(10)  
    .width('100%')
```

请自己找到位置进行代码修改。

此时刷新 Previewer 的效果：



在页面 `build()` 里面调用，比如食堂的 `Card`，加入：

```
{  
  Column({space: 20}) {  
    Text('味道都很不错啊').fontSize(24)  
    Row(){  
      Button('Agree')  
      Blank(10)  
      Button('No!')  
    }  
  }  
}
```

```
}
```

```
}
```

对于运动，添加：

```
{
```

```
  Row({space: 10}){
```

```
    Button('篮球')
```

```
    Button('足球')
```

```
    Button('乒乓球')
```

```
  }
```

```
}
```

效果：

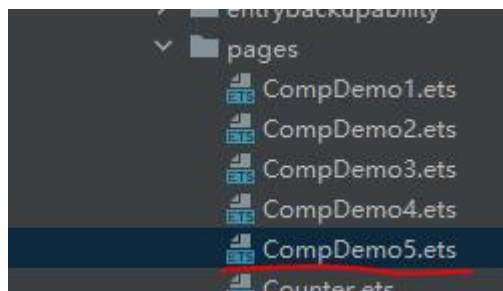


可以看到，通过这种方式，可以给子组件传入不同的 UI 界面。

7. 多个@BuilderParam 的情况

在上一部分，对于自定义组件，我们用一个@BuilderParam 构建了一个函数变量，让外部 UI 传递进来。但是，既然是“Param”，就可以有多个。对于多个的情况，应该如何处理？我们通过实例来尝试。

创建一个新的文件 CompDemo5.ets:



加入基础代码：

```
@Component

struct MyCard {

    build() {

        Column() {

            // 标题部分

            Row() {

                Text('标题部分')

            }

            .height(30)

            .width('100%')

            .border({ color: '#ccc', width: { bottom: 1 } })
```

```
.padding({ left: 10 })
```

```
// 内容部分
```

```
Row() {
```

```
Text('内容部分')
```

```
}
```

```
.width('100%')
```

```
.padding(10)
```

```
}
```

```
.width('100%')
```

```
.height(100)
```

```
.backgroundColor(Color.White)
```

```
.borderRadius(10)
```

```
.justifyContent(FlexAlign.Start)
```

```
}
```

```
}
```

```
@Entry
```

```
@Component
```

```
struct CompDemo5 {
```

```
build() {
```

```
Column({ space: 10 }) {
```

```
MyCard()
```

```

MyCard()

}

.width('100%')

.height('100%')

.padding(20)

.backgroundColor('#ccc')

}

}

```

效果：



如果我们希望标题部分和内容部分都通过`@BuilderParam` 来构建函数，然后外部调用的时候传入，可以定义两个`@BuilderParam`。在子组件中加入代码（位置自己找到）：

```

@Builder titleDefaultBuilder () {

    Text('我是默认的大标题')

}

@Builder contentDefaultBuilder () {

    Text('我是默认的内容')

```

```
}
```

```
@BuilderParam tBuilder: () => void = this.titleDefaultBuilder
```

```
@BuilderParam cBuilder: () => void = this.contentDefaultBuilder
```

然后在相应的地方改动，改为调用这两个 Builder()：

```
Row() {
```

```
    this.tBuilder()
```

```
}
```

以及：

```
Row() {
```

```
    this.cBuilder()
```

```
}
```

刷新 Previewer，效果：



在页面的代码中，先定义两个 Builder()：

```
@Builder ftBuilder () {
```

```
    Text('我是传入的大标题结构')
```

```
}
```

```
@Builder fcBuilder () {
```

```
Text('我是内容部分')
```

```
Text('我是内容部分')
```

```
Text('我是内容部分')
```

```
}
```

然后我们在第二个 MyCard 中调用：

```
MyCard()
```

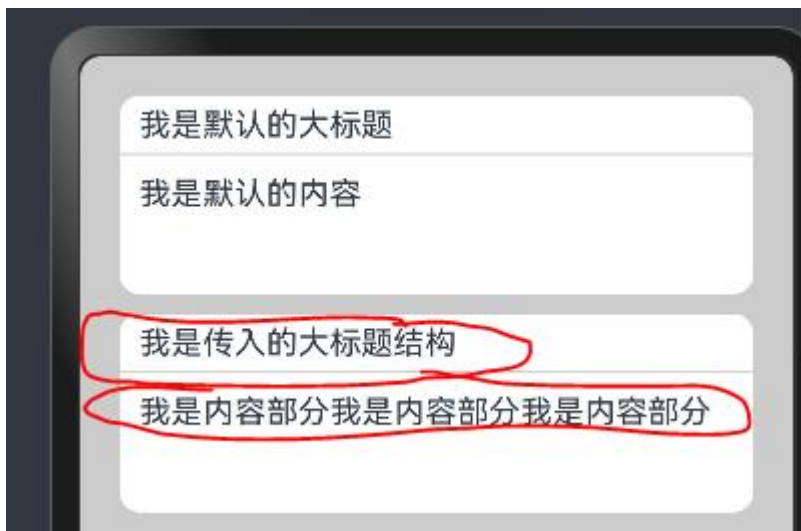
```
MyCard({
```

```
  tBuilder: this.ftBuilder,
```

```
  cBuilder: this.fcBuilder
```

```
})
```

这样可以和第一个 MyCard 做一个比较。效果：



也就是说，如果只是单个@BuilderParam，可以写在 MyCard(){} 的花括号里面，直接替换掉，所以通常命名的时候称之为 defaultBuilder。对于多个的情况，还是写在()圆括号里面，以参数的方式传入。

到此，我们对自定义组件有了一定的了解，也做了一些尝试。

五、实验注意事项

1. 注意教师的操作演示。
2. 学生机与教师机内网连通，能接收和提交实验结果。
3. 按实验要求输入测试数据，并查看输出结果是否符合实验结果。

六、思考题

1. 通过这个实验，你学到了什么？