HUAWEI

ArkTS语言

华为云 一切皆服务

前言

- ArkTS是HarmonyOS优选的主力应用开发语言。当前，ArkTS在TS基础上主要扩展了声明式UI能力，让开发者以更简洁、更自然的方式开发高性能应用。
- 本章将会带领开发者们初识ArkTS语言，学习ArkTS语言的基础语法、UI描述规范、常用装饰器、状态管理装饰器和渲染控制语法。

课程目标

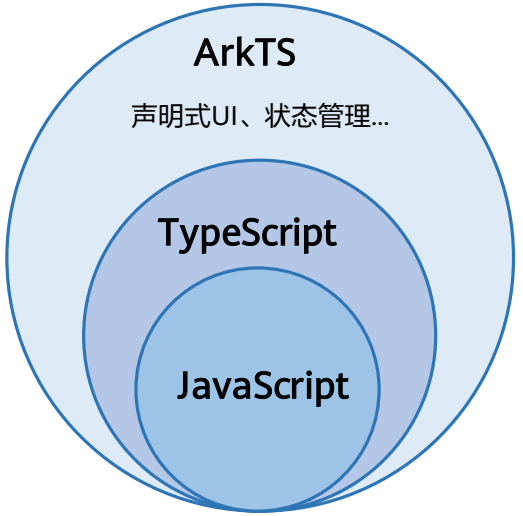
- 学完本课程后，您将能够：
  - 描述ArkTS语言；
  - 掌握ArkTS语言基础语法；
  - 运用ArkTS中的常用装饰器；
  - 理解状态管理的基本概念；
  - 掌握页面级变量的状态管理；
  - 掌握渲染控制语法。

目录

ArkTS的前世今生

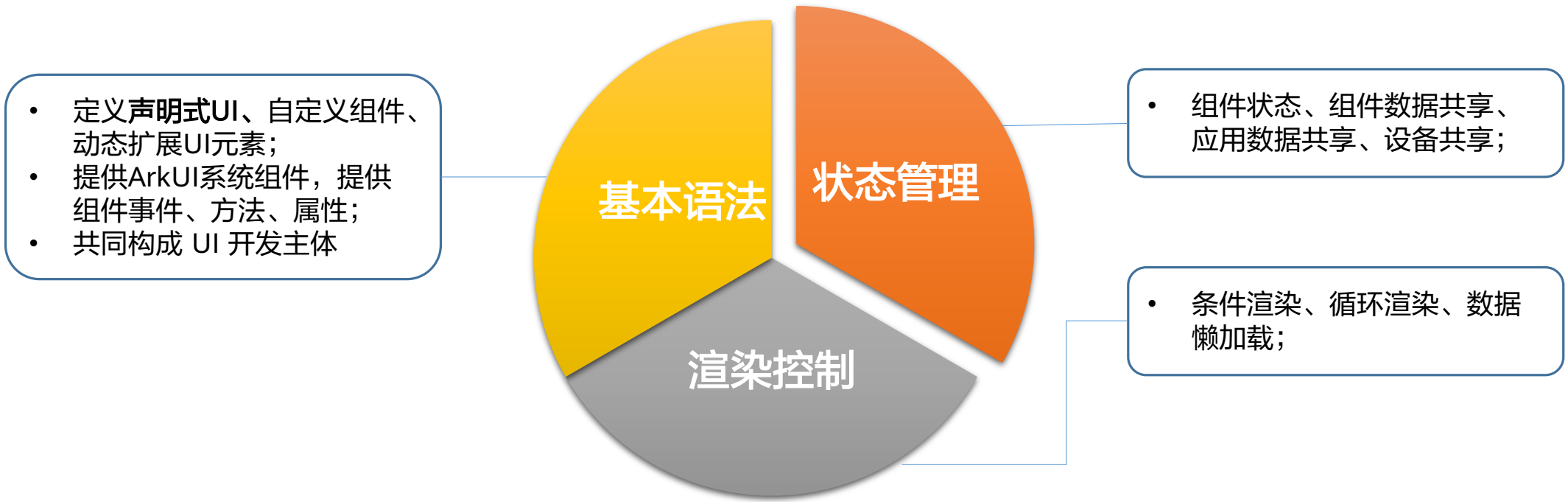
- ArkTS是鸿蒙生态应用的开发语言，使用.ets作为ArkTS语言源码文件后缀。它在保持TypeScript（简称TS）基本语法风格的基础上，对TS的动态类型特性施加更严格的约束，引入静态类型。同时提供了声明式UI、状态管理等能力，让开发者以更简洁、更自然的方式开发高性能应用。

JavaScript	• 一种属于网络的高级脚本语言，广泛应用于Web应用开发，常用来为网页添加各式各样的动态功能。
TypeScript	• 它是JavaScript的一个超集，扩展了JavaScript的语法，通过在JavaScript的基础上添加静态类型定义构建而成，是开源编程语言。
ArkTS	• 它兼容TypeScript语言，并进一步拓展了声明式UI、状态管理、并发任务等能力。



ArkTS的扩展特性

- ArkTS是一种为构建高性能应用而设计的编程语言。ArkTS在继承TypeScript绝大多数语法的基础上进行了优化，以提供更高的性能和开发效率。
- 以下是ArkTS主要扩展的能力：



什么是“声明式”

- ArkTS提供了简洁自然的声明式语法、组件化机制、数据-UI自动关联等能力，实现了贴近自然语言，书写效率更高的编程方式。

那么，什么是声明式？ → 一般来说我们对于声明式的理解都是相对于命令式而言的。

师傅，去XX图书馆。下个路口左转，再前进1000米，前面路口掉头，前进150米，右拐就到了。

命令式

详细的命令机器怎么（How）去处理一件事情以达到你想要的结果（What）。结果取决于水平。

师傅，去XX图书馆。

声明式

只告诉你想要的结果（What），机器自己摸索过程（How）。机器需具备一定智能性。

自然简洁语法

- 最常见的“声明式”语言，可以说是SQL，这是一门数据库语言。

```
var dogsWithOwners = [];
var dog,owner;
for(var di=0;di<dogs.length;di++){
  dog = dogs[di];
  for(var oi=0;oi<dogs.length;oi++){
    owner = owners[di];
    if(owner&&dog.owner_id==owner.id){
      dogsWithOwners.push({
        dog:dog,
        owner:owner
      })
    }
  }
}
```

命令式风格代码

```
SELECT * from dogs INNER JOIN owners
WHERE
dog.owner_id = owner.id
```

声明式风格代码

在这个例子里我们可以看到，使用其他通用语言“命令式”的完成查询，要比使用SQL语言“声明式”的完成查询复杂的多。
在数据量很大的时候两种方法的复杂度差距会更加明显，数据库可以帮助你收集数据分布的统计信息、维护索引和选择最佳执行路径，以保证查询性能，如果你自行编码完成这些工作，那代码量会有成千上万行。

“声明式UI”的好处

- 在很多特定的领域，我们往往更喜欢“声明式”的语言。比如数据处理领域，我们上一张胶片中所提到的SQL语言就是其中的代表之一。
- 在前端界面开发领域，也很少有人愿意使用“命令式”语言。因为在这些特定的领域，已经有开发人员帮我们归纳和提取了完备的what，消除了实践“声明式”的最大障碍。例如ArkUI已经提供了丰富的组件库，供开发者使用 。

代码风格：声明式 UI 使用更抽象、更高级的语法，通常使用配置、声明或描述来构建用户界面；命令式 UI 更侧重于编写详细的指令和操作来控制用户界面的构建和交互。

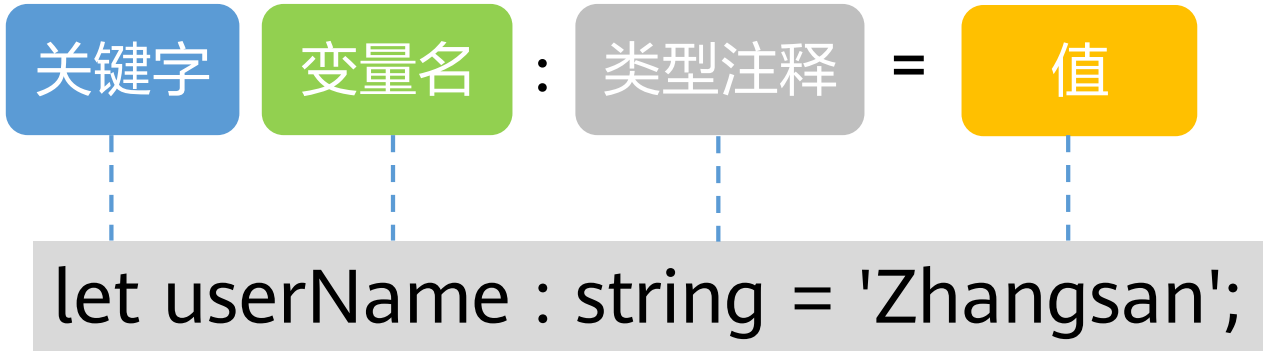
可维护性：声明式 UI 更易于维护，因为它的代码更清晰、简洁，易于理解；命令式 UI 则可能更复杂、更冗长，因为需要明确指定每个步骤和操作的细节。

目录

HUAWEI

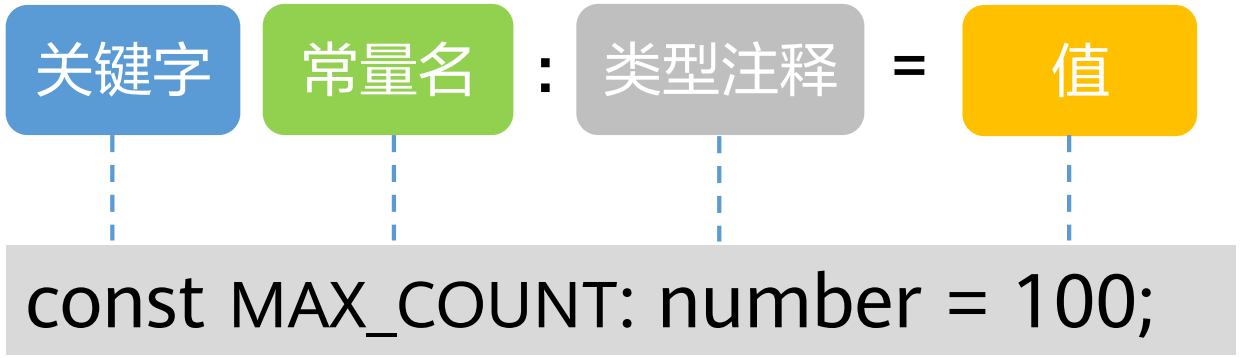
变量声明

- 在编程的世界里，数据是构建一切的基础。在ArkTS中，变量就是存储数据的容器。它们就像是我们生活中的盒子，可以装下各种物品，让我们在需要的时候随时取用。
- 在ArkTS中可以通过关键字let开头的声明引入变量，并通过类型注释指定类型。该变量在程序执行期间可以具有不同的值。



常量声明

- 变量在运行过程中是可以变化的。但在现实生活中，有一些数据是不会变化的，比如：数学上的PI、自然常数e等等，这些用变量来表示就不太恰当。所以如果我们希望一个值声明之后，后续运行过程中不能变化，我们就可使用常量来表示。
- 在ArkTS中使用const声明一个常量，以关键字const开头的声明引入只读常量，该常量只能被赋值一次。对常量重新赋值会造成编译时错误。



自动类型推断

- 由于ArkTS是一种静态类型语言，其特点是所有数据的类型都必须在编译时确定。
- 但是，如果一个变量或常量的声明中已赋予了初始值，那么开发者就不需要明确指定类型，而是可以让编译器根据上下文进行推导数据类型。
- 以下示例中，两条声明语句都是有效的，两个变量都是string类型：

```
let hi1: string = ‘hello’;
//初始化时定义了变量类型为String。

let hi2 = ‘hello, world’;
//虽然没有定义变量类型，但初始化时编译器自动推断了变量类型为String类型。
```

变量声明限制

- 任何一种代码的书写中，命名的规范都是我们基础的体现，都是我们需要去遵守的。ArkTS变量声明也存在如下限制：

只能包含数字、字母、下划线以及$符号，不能包含其他特殊字符，包括空格。

首个符号，不能以数字开头，只能字母、下划线以及$符号开头。

不能使用关键字和保留字（已经定义了特殊含义的单词）命名，例如：this、let。

次数有限制：同一个名字的变量在相同作用域中只能声明一次。

目录

数据类型

- 无论是Android还是iOS开发，都提供了多种数据类型用于常见的业务开发，但在ArkTs中，数据类型就大有不同，例如int，long，float，double统一就是number类型。
- ArkTs是TypeScript的超集，其数据类型也是基于TypeScript而来。可以大致分为以下4种类型：

基础类型： Number、String、Boolean、Enum

引用类型： 基类Object、Array

联合类型： Union

类型别名： Aliases

基础类型

- Number类型：ArkTS提供number和Number类型，任何整数和浮点数都可以被赋给此类型的变量。
- Boolean类型：Boolean类型由true和false两个逻辑值组成。
- String类型：字符串字面量由单引号（‘）或双引号（”）之间括起来的零个或多个字符组成。

```
let age: number = 20;
let isTrue: boolean = true;
let name: string = '小明';

console.log(age.toString()); // 输出：20
console.log(name); // 输出小明
console.log('我的名字叫${name},今年${age}岁');
```

Enum类型

- Enum类型，又称枚举类型，通常用于为程序中的一组相关的常量，或约定变量只能在该组数据范围内选择值，以便于程序的可读性和维护性。

语法

```
enum 枚举名 {
  常量1 = 值,
  常量2 = 值,
  ..... }
```



```
//枚举颜色
enum ColorSet {
  White = 0xFF,
  Grey = 0x7F,
  Black = 0x00 }
```

- 使用枚举类型，约束变量：

```
let color: ColorSet = ColorSet.Grey
// 取值会从枚举常量列表ColorSet中获取
```

引用类型

- Object类型：Object类型是所有引用类型的基类型。任何值，包括基本类型的值（它们会被自动装箱），都可以直接被赋给Object类型的变量。

```
var obj = {name:"zyj"};   // 创建一个对象
obj.name = "percy";       // 改变 name 属性的值
obj.age = 21;             // 添加 age 属性
```

- Array类型：即数组，是由可赋值给数组声明中指定的元素类型的数据组成的对象。数组的长度由数组中元素的个数来确定。数组中第一个元素的索引为0。

```
// 数组声明了类型
let array = Array<number>(1, 2, 3, 4, 5)
// 数组没有声明类型
let arr = [1, "字符串", true, new Test()]
let array = Array<any>(1, "字符串", true, new Test())
```

Union类型

- union类型，即联合类型，是由多个类型组合成的引用类型。联合类型包含了变量可能的所有类型。

语法

```
let 变量: 类型1 | 类型2 | 类型3 = 值
```

```
let luckyNum: number | string = 7;
luckyNum= 'seven';
```

- 联合类型还可以将变量值，约定在一组数据范围内进行选择。

```
let sex: 'man' | 'woman' = 'woman'
sex='secret';  //给性别赋值时只能选择'man'，'woman'，这里附了其他值就会出现报错。
```

Aliases类型

- Aliases为类型别名，它允许为已有的类型定义一个别名，以提高代码的可读性和可维护性。通过类型别名，可以给复杂或重复出现的类型定义一个简洁的名称。

1、基础类型别名：

```
type ID = number;
let userId: ID = 123;
// 使用type关键字为number类型定义了一个别名ID，然后将其用于声明变量userId。
```

2、联合类型别名：

```
type Status = "success" | "failure";
let result: Status = "success";
// 使用type关键字为字符串字面量类型定义了一个联合类型别名Status，它只允许取值为"success"或"failure"。然后将其用于声明变量result。
```

组件内部声明变量

- 组件内部声明变量不需要使用let关键字，而且在调用组件内变量时，需要使用this去引用。也可以在组件内部直接声明变量。

```
 //使用let关键字可以在组件外定义变量
let message2:string = 'Hi world'
@Entry
@Component
struct Index {
  message: string = 'Hello World' //组件内部声明变量不需要使用let关键字
  build() {
    Row() {
      Column() {
        //使用this表示找当前组件内的变量 不用this表示在全局中查找
        Text(this.message)
        Text(message2)
      }
    }
  }}
```

目录

函数声明

- 函数是一组一起执行一个任务的语句，函数声明需要告诉编译器函数的名称、参数和返回类型。
- 以下示例是一个简单的函数：

```
// 有名函数：给变量设置为number类型
function add(x: number, y: number): number {
  let z: number = x + y;
  return z;
}
```

```
//匿名函数：给变量设置为number类型
let myAdd = function (x: number, y: number): number {
   return x + y;
}
```

箭头函数

- 箭头函数表达式的语法比传统的函数表达式更简洁，例如：箭头函数的返回类型可以省略；省略时，返回类型通过函数体推断。

```
let myAdd = function (x: number, y: number): number {
    return x + y;
}
```

转换成箭头函数

```
let myAdd = (x: number, y: number): number => {
    return x + y;
}
```

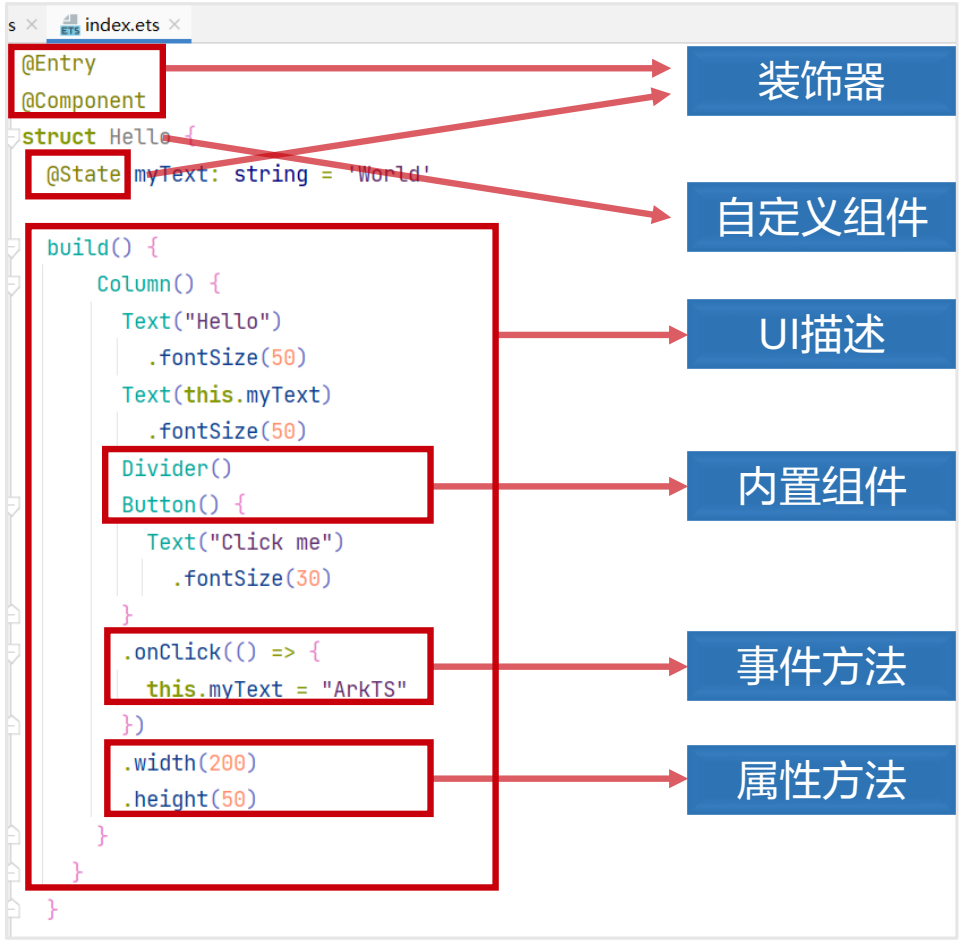
表达式可以指定为箭头函数，使表达更简短，因此以下两种表达方式是等价的：

```
let sum1 = (x: number, y: number) => { return x + y; }
let sum2 = (x: number, y: number) => x + y
```

目录

ArkTS的基本组成

- 在初步了解了ArkTS语言之后，我们以一个具体的示例来说明ArkTS的基本组成。



装饰器：用来装饰类、结构体、方法以及变量，赋予其特殊的含义，如上述示例中 @Entry 、 @Component 、 @State 都是装饰器。

自定义组件：可复用的 UI 单元，可组合其它组件，如上述被 @Component 装饰的 struct Hello。

UI描述：声明式的方法来描述UI的结构，例如build()方法中的代码块。

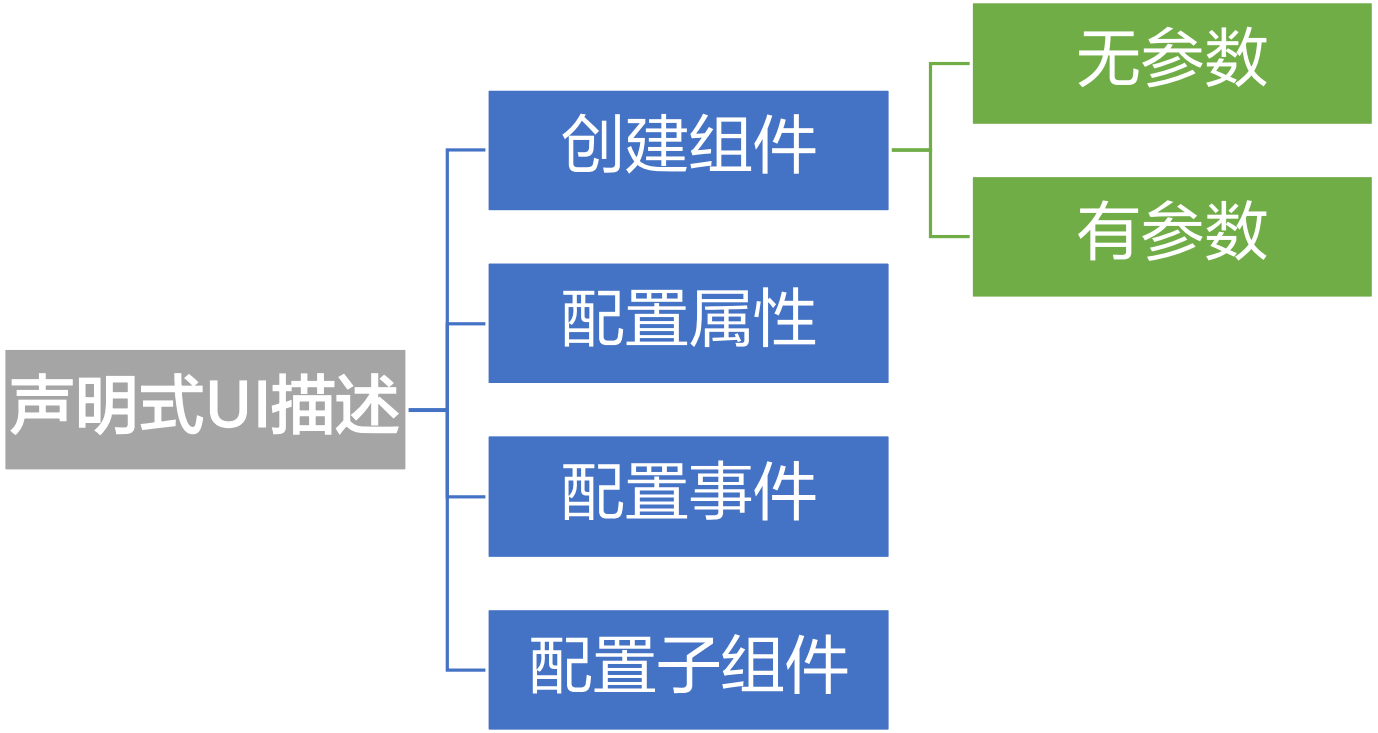
内置组件：ArkTS中默认内置的基本组件和布局组件，开发者可以直接调用，如Column、Text、Divider、Button等。

事件方法：用于添加组件对事件的响应逻辑，统一通过事件方法进行设置，如跟随在Button后面的onClick()。

属性方法：用于组件属性的配置，统一通过属性方法设置，如fontSize()、width()、height()、color() 等，可通过链式调用的方式设置多项属性。

声明式UI描述

- 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。接下来，我们将着重学习UI描述规范。



组件无参数构造配置

- 根据组件构造方法的不同，创建组件包含有参数和无参数两种方式。
- 组件的接口定义不包含必选构造参数，组件后面的“()”中不需要配置任何内容。例如，Divider组件不包含构造参数：

```
Column() {
    Text('item 1')
    Divider()
    Text('item 2')
}
```

必选参数构造配置

- 如果组件的接口定义中包含必选构造参数，则在组件后面的“()”中必须配置参数，参数可以使用常量进行赋值。例如：
  - Image组件的必选参数src：

```
Image('https://xyz/a.jpg')
```

  - Text组件的可选参数content：

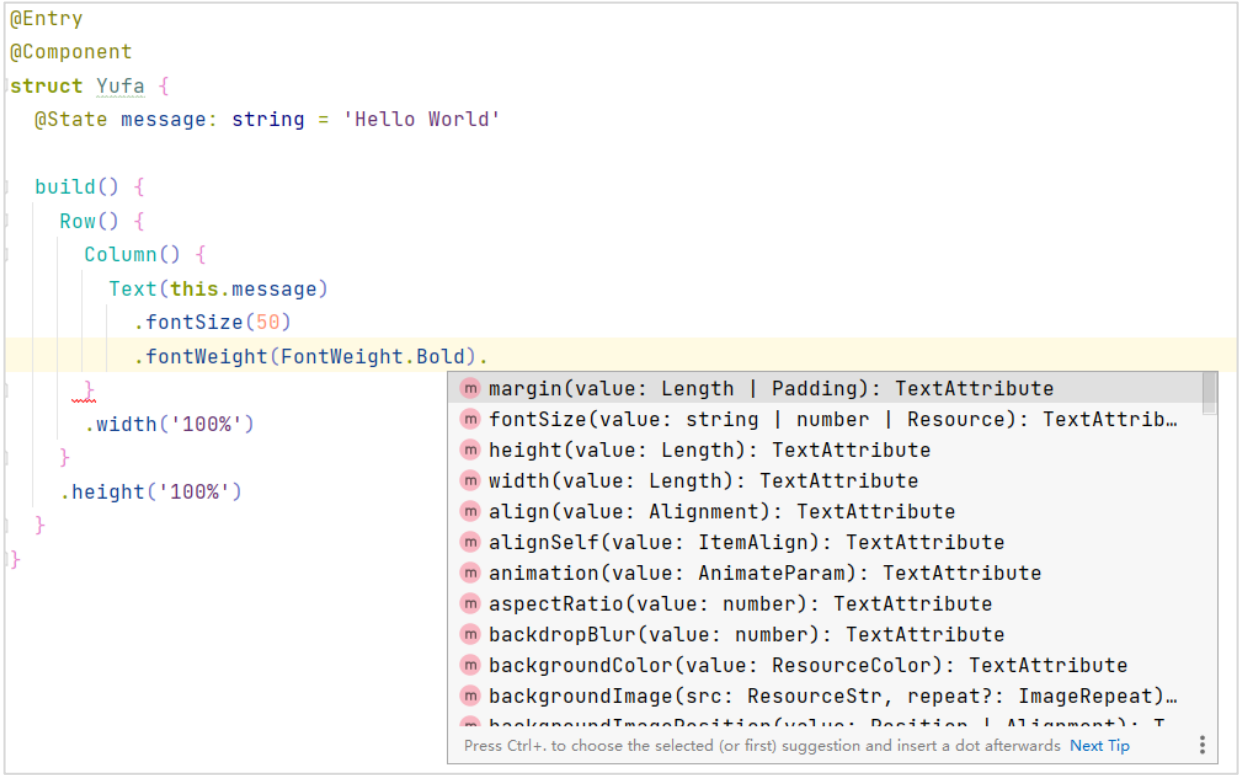
```
Text('123')
```

- 变量或表达式也可以用于参数赋值，其中表达式返回的结果类型必须满足参数类型要求。例如，传递变量或表达式来构造Image和Text组件的参数：

```
Image(this.imagePath)
Image('https://' + this.imageUrl)
Text(‘count: ${this.count}’)
```

属性方法

- 属性方法：用于配置组件属性，如fontSize()、width()、height()、color()等。在DevEco Studio可以在组件后面通过“.”链式调用的方式配置UI的属性，建议每个属性方法单独写一行。



事件方法

- 事件方法：在事件方法的回调中添加组件响应逻辑。例如，为Button组件添加onClick方法，在onClick方法的回调中添加点击响应逻辑。

使用**箭头函数**配置组件的事件方法。

```
Button('add counter')
    .onClick(() => {
        this.counter += 2
    })
```

子组件配置

- 对于支持子组件配置的组件，例如容器组件，在“{ ... }”里为组件添加子组件的UI描述。Column、Row、Stack、Grid和List组件都是容器组件。
- 以下是简单的Column示例：

```
Column() {
    Text('Hello')
        .fontSize(100)
    Divider()
    Text(this.myText)
        .fontSize(100)
        .fontColor(Color.Red)
}
```

```
Column() {
    Column() {
        Button() {
            Text('+ 1')
        }.type(ButtonType.Capsule)
        .onClick(() => console.log ('+1 clicked!'))
        Image('1.jpg')
    }
    Divider()
    Column() {
        Button() {
            Text('+ 2')
        }.type(ButtonType.Capsule)
        .onClick(() => console.log ('+2 clicked!'))
        Image('2.jpg')
    }
    Divider()
    Column() {
        Button() {
            Text('+ 3')
        }.type(ButtonType.Capsule)
        .onClick(() => console.log('+3 clicked!'))
        Image('3.jpg')
    }
}.alignItems(HorizontalAlign.Center)
```

目录

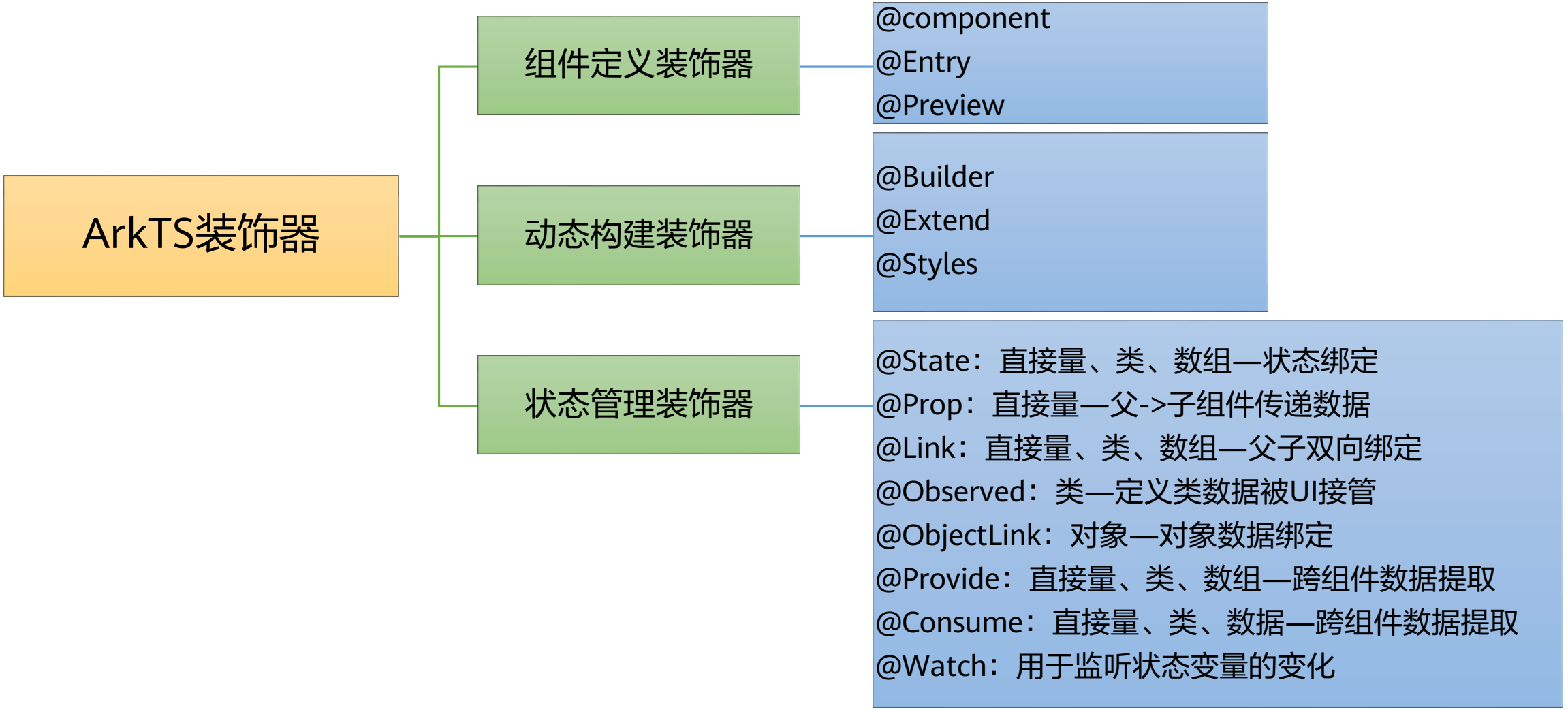
HUAWEI

装饰器

- 装饰器是一种在不改变原类和使用继承的情况下，动态地扩展对象功能的特殊类型的声明。它能用于装饰类、结构、方法和变量。共性开头存在@符号。例如，@Entry、@Component和@State都是装饰器。

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

ArkTS装饰器



HUAWEI

@Entry

- 用@Entry装饰的自定义组件用作页面的默认入口组件，加载页面时，将首先创建并呈现@Entry装饰的自定义组件，在单个源文件中，有且仅有一个@Entry。

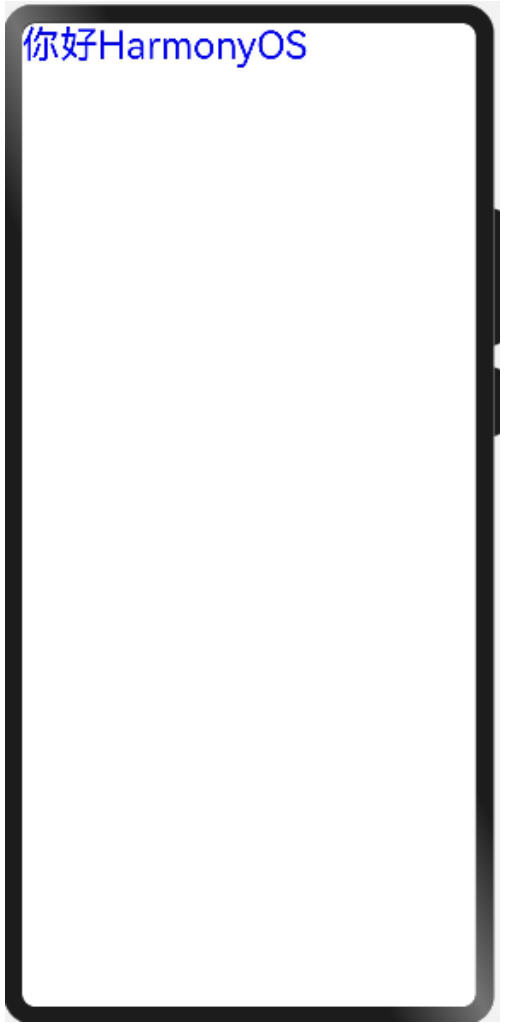
表示@Entry是下方的组件整个页面渲染载入的入口，作为一个入口组件，先载入@Entry，再载入其它组件。

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

练一练：更改@Entry的位置

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
@Component
struct test1{
  build(){
    Text("你好HarmonyOS")
      .fontColor(Color.Blue)
      .fontSize(30)
  }
}
```


```
@Component
struct Index {
  @State message: string = 'Hello World'

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
@Entry
@Component
struct test1{
  build(){
    Text("你好HarmonyOS")
      .fontColor(Color.Blue)
      .fontSize(30)
  }
}
```



HUAWEI

UI描述

- 由struct这个关键字修饰的结构名为Index，在Index结构体{...}书写代码，在这结构体还会有变量message和组件，变量可以被装饰器修饰，build(){...}用来写想要用到的组件和代码。

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

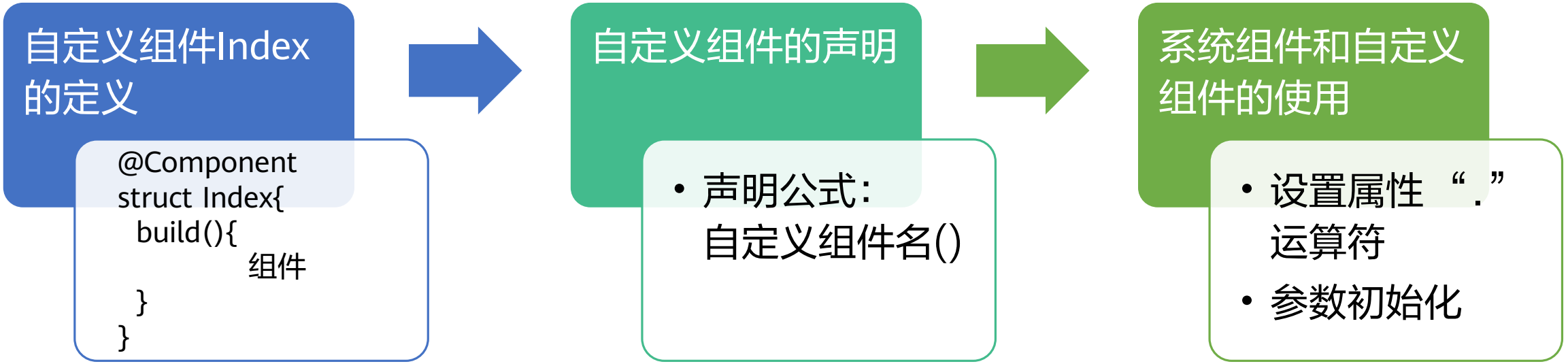
组件的声明和使用

- **组件分为系统组件和自定义组件。**
- 二者区别：系统组件在华为开发者文档中有详细说明且包含在开发工具中，可以直接调用；而自定义组件需要开发者自行写代码实现，先定义后调用。
- 系统组件的声明
  - 非容器组件声明公式1：系统组件名()
  - 容器组件声明公式2：系统组件名(){}

```
@Entry
@Component
struct Index{
  build(){
    Row(){
      Column(){
        Text()
      }
    }
  }
}
```

@Component

- @Component装饰的struct表示该结构体具有组件化能力，能够成为一个独立的组件，这种类型的组件也称为自定义组件，在build方法里描述UI结构。



@Component：可组合

- 允许开发人员组合使用内置组件、其他组件、公共属性和方法。

```
@Component
struct test1{
  build(){
    Text("你好HarmonyOS")
      .fontColor(Color.Blue)
      .fontSize(30)
  }
}
```

```
@Entry
@Component
struct Index {
  @State message: string = 'Hello World'
  build() {
    Row() {
      Column() {
        test1()
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

HUAWEI

@Component：可重用

- 自定义组件可以被其他组件重用，并作为不同的实例在不同的父组件或容器中使用。

```
@Component
struct test1{
  build(){
    Text("你好HarmonyOS")
      .fontColor(Color.Blue)
      .fontSize(30)
  }
}
@Component
struct test2{
  build(){
    Text("开始学习吧")
      .fontColor(Color.Red)
      .fontSize(30)
  }
}
```

```
@Entry
@Component
struct Index {
  @State message: string = 'Hello World'
  build() {
    Row() {
      Column() {
        test1()
        test1()
        test2()
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

HUAWEI

@Component：数据驱动更新

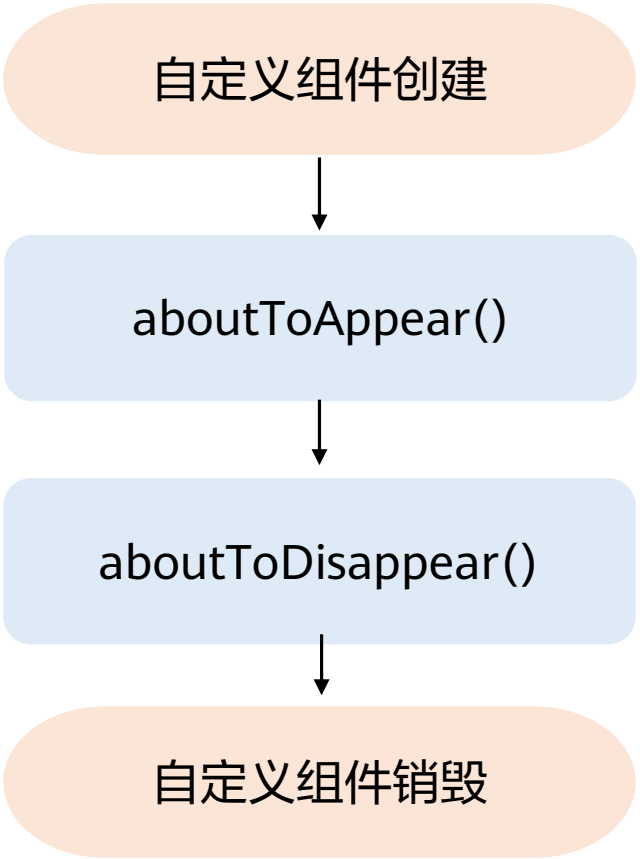
- 由状态变量的数据驱动，实现UI自动更新。

```
@Component
struct test1{
  build(){
    Text("你好HarmonyOS")
      .fontColor(Color.Blue)
      .fontSize(30)
  }
}
```

若自定义组件中的变量、属性发生变化，当前的组件就会重新渲染绘制。

自定义组件生命周期回调函数

- 自定义组件的生命周期回调函数用于通知用户该自定义组件的生命周期，这些回调函数是私有的，在运行时由开发框架在特定的时间进行调用，不能从应用程序中手动调用这些回调函数。

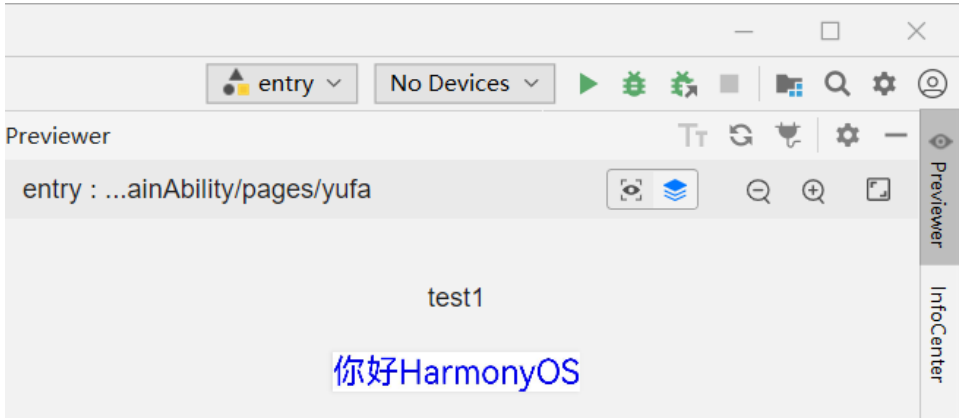


@Preview

- 用@Preview装饰的自定义组件可以在DevEco Studio的预览器上进行预览，加载页面时，将创建并呈现@Preview装饰的自定义组件。

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

  build() {
    Row() {
      Column() {
        test1()
        test2()
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

```
@Preview
@Component
struct test1{
  build(){
    Text("你好HarmonyOS")
      .fontColor(Color.Blue)
      .fontSize(30)
  }
}
@Component
struct test2{
  build(){
    Text("开始学习吧")
      .fontColor(Color.Red)
      .fontSize(30)
  }
}
```



目录

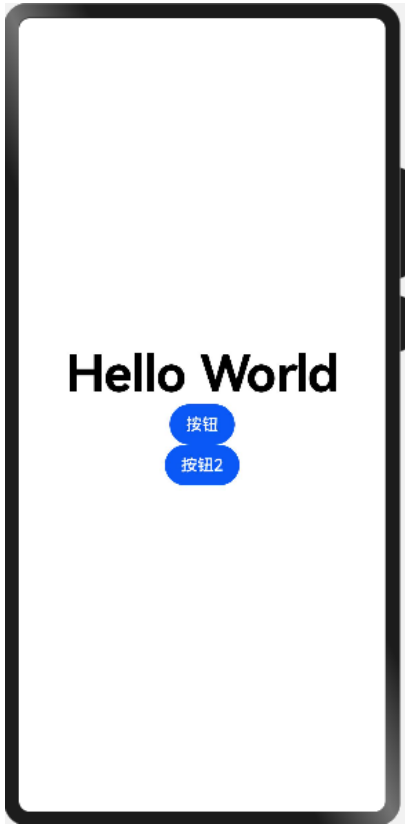
HUAWEI

@Builder

- @Builder装饰的方法用于定义组件的声明式UI描述，在一个自定义组件内快速生成多个布局内容。

```
@Component
struct myTest{
  build(){
    Button("按钮")
  }
}
```

```
@Entry
@Component
struct Index {
 @State message: string = 'Hello World'
  @Builder myTest2(){
    Button("按钮2")
  }

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
          myTest()
          this.myTest2()
      }
      .width('100%')
    }
    .height('100%')
  }
}
```



HUAWEI

@Extend

- @Extend装饰器将新的属性函数添加到内置组件上，如Text、Column、Button等。通过@Extend装饰器可以快速定义并复用组件的自定义样式。

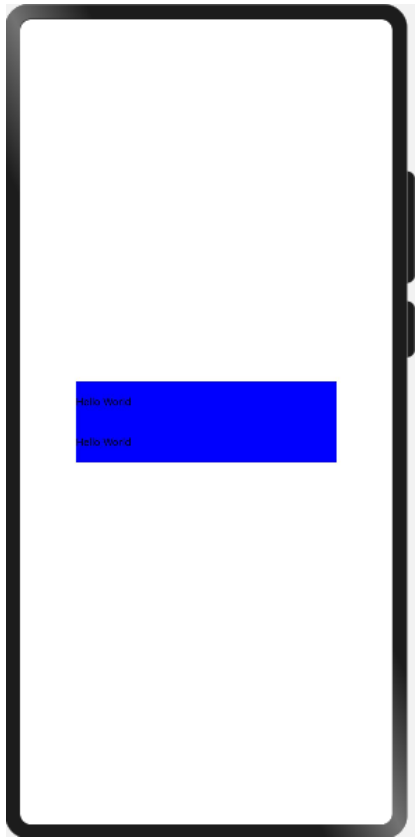
```
@Entry
@Component
struct Index {
  @State message: string = 'Hello World'
  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontColor(Color.Red)
        Text(this.message)
          .fontSize(50)
          .fontColor(Color.Red)
        Text(this.message)
          .fontSize(50)
          .fontColor(Color.Red)
}
      .width("100%")
    }
    .height("100%")
  }
}
```


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

  build() {
    Row() {
      Column() {
        Text(this.message).font(30)
        Text(this.message).font(30)
        Text(this.message).font(30)
      }
      .width("100%")
    }
    .height("100%")
  }
}
@Extend(Text) function font(fontSize:number){
  .fontColor(Color.Green)
  .fontSize(fontSize)
  .fontStyle(FontStyle.Italic)
}
```

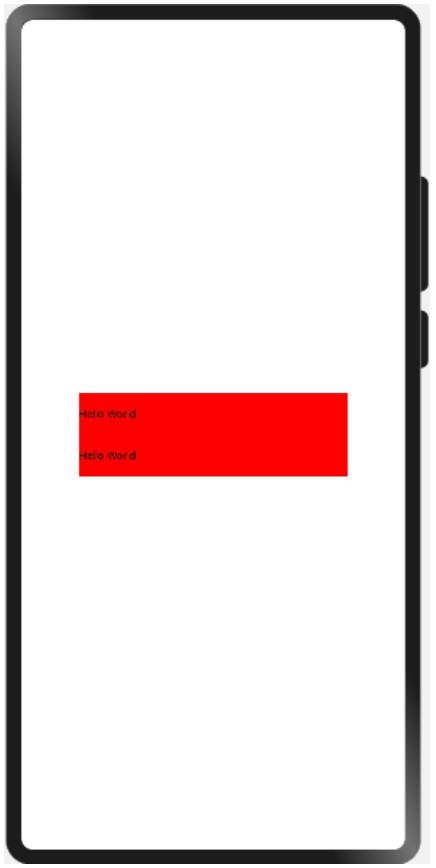
@Styles

- @Styles装饰器将新的属性函数添加到组件上，如Text、Column、Button等。当前@Styles仅支持通用属性。通过@Styles装饰器可以快速定义并复用组件的自定义样式。

```
@Entry
@Component
struct Index {
  @State message: string = 'Hello World'
  build() {
    Row() {
      Column() {
        Text(this.message)
          .width("70")
          .height("5%")
          .backgroundColor(Color.Blue)
        Text(this.message)
          .width("70")
          .height("5%")
          .backgroundColor(Color.Blue)
}
      .width("100%")
    }
    .height("100%")
  }
}
```

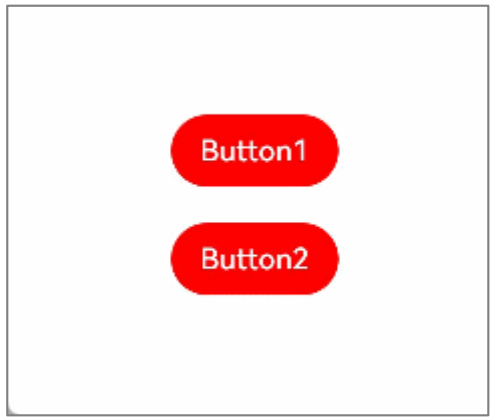



```
@Entry
@Component
struct Yufa {
  @State message: string = 'Hello World'
  build() {
    Row() {
      Column() {
        Text(this.message)
          .globalStyle()
        Text(this.message)
          .globalStyle()
      }
      .width("100%")
    }
    .height("100%")
  }
}
@Styles function globalStyle(){
  .width("70%")
  .height("5%")
  .backgroundColor(Color.Red)
}
```



stateStyles：多态样式

- stateStyles是属性方法，可以根据UI内部状态来设置样式，类似于css伪类，但语法不同。ArkUI提供以下五种状态：
  - focused：获焦态。
  - normal：正常态。
  - pressed：按压态。
  - disabled：不可用态。
  - selected10+：选中态。




```
Column() {
      Button('Button1')
        .stateStyles({
          focused: {
            .backgroundColor(Color.Pink)
          }, pressed: {
            .backgroundColor(Color.Black)
          },normal: {
            .backgroundColor(Color.Red)
          }
        }) .margin(20)
      Button('Button2')
        .stateStyles({
          focused: {
            .backgroundColor(Color.Pink)
          },
          pressed: {
            .backgroundColor(Color.Black)
          },
          normal: {
            .backgroundColor(Color.Red)
          }
        })
    }.margin('30%')
```

目录

状态管理概述

- 当开发者希望构建一个动态的、有交互的界面，就需要引入“状态”的概念。
- 在声明式UI编程框架中，UI是程序状态的运行结果，用户构建了一个UI模型，其中被装饰器修饰的变量叫做状态变量。当状态变量改变时，UI作为返回结果，也将进行对应的改变。
- 这些状态变量变化所带来的UI的重新渲染，在ArkUI中统称为状态管理机制。

基本概念

- **状态变量**：被状态装饰器装饰的变量，状态变量值的改变会引起UI的渲染更新。
  - 示例：@State num: number = 1,其中，@State是状态装饰器，num是状态变量。
- **常规变量**：没有被状态装饰器装饰的变量，通常应用于辅助计算。它的改变永远不会引起UI的刷新。
  - 以下示例中increaseBy变量为常规变量。
- **数据源/同步源**：状态变量的原始来源，可以同步给不同的状态数据。通常意义为父组件传给子组件的数据。
  - 以下示例中数据源为count: 1。
- **从父组件初始化**：父组件使用命名参数机制，将指定参数传递给子组件。子组件初始化的默认值在有父组件传值的情况下，会被覆盖。

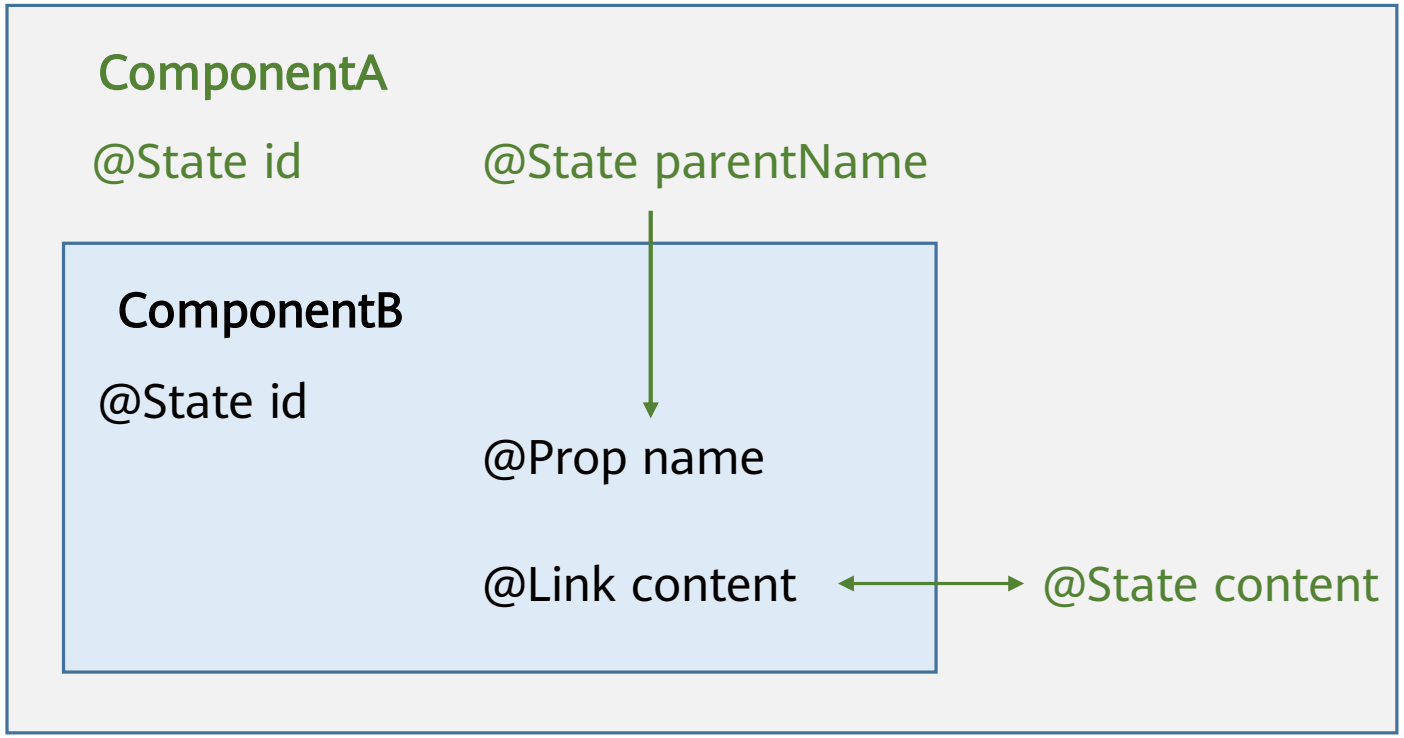
基本概念示例

- **初始化子节点**：父组件中状态变量可以传递给子组件，初始化子组件对应的状态变量。
- **本地初始化**：在变量声明的时候赋值，作为变量的默认值。示例：@State count: number = 0。

```
@Component
struct MyComponent {
  @State count: number = 0; // 状态变量
  private increaseBy: number = 1; // 常规变量

  build() {
  }
}

@Component
struct Parent {
  build() {
    Column() {
     // 从父组件初始化，覆盖本地定义的默认值
      MyComponent({ count: 1, increaseBy: 2 })
    } //数据源/同步源
  }
}
```

组件状态管理装饰器

- 组件状态管理装饰器用来管理组件中的状态，它们分别是：@State、@Prop、@Link。
- @State、@Prop、@Link三者关系如图所示：



@State定义

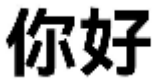
- @State装饰的变量是组件内部的状态数据，当这些状态数据被修改时，将会调用所在组件的build方法进行UI刷新。

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

修改message，Text文本组件中绑定this.message

“Hello World”改为“你好”

```
@State message: string = '你好'
```

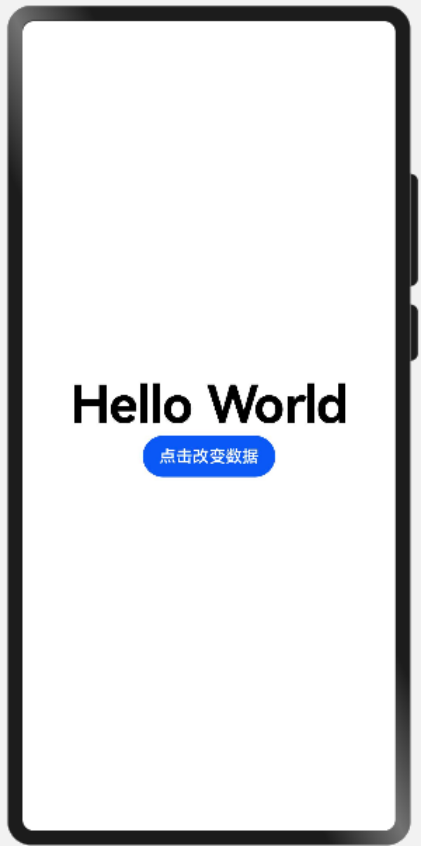


HUAWEI

思考：@State作用

- 若删除@State装饰器，修改其原修饰的变量，观察预览器中的数据是否改变?

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
        Button("点击改变数据")
          .onClick(()=>{this.message="欢迎您学习
HarmonyOS"})
      }
      .width('100%')
    }
    .height('100%')
  }
}
```



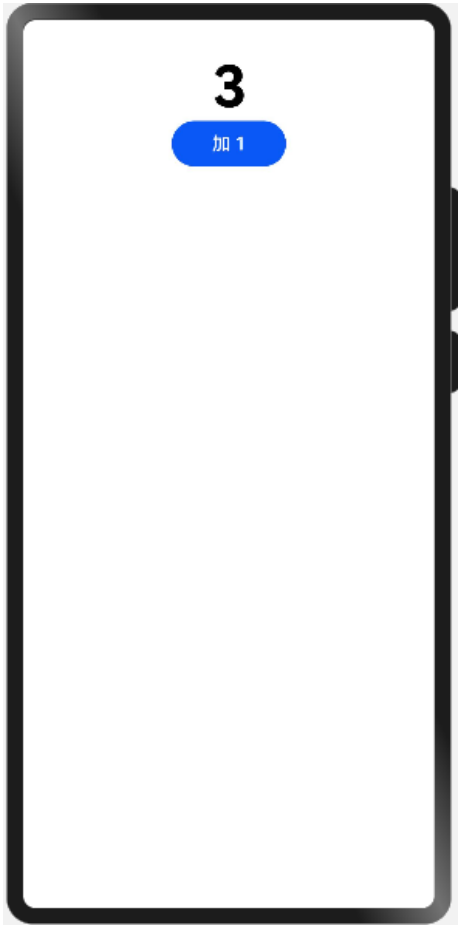
HUAWEI

@State状态数据特征

- @State状态数据具有以下特征：
  - 支持多种类型：允许class、number、boolean、string强类型的按值和按引用类型。允许这些强类型构成的数组，即Array<class>、Array<string>、Array<boolean>、Array<number>。不允许object和any。
  - 支持多实例：组件不同实例的内部状态数据独立。
  - 内部私有：标记为@State的属性是私有变量，只能在组件内访问。
  - 需要本地初始化：必须为所有@State变量分配初始值，将变量保持未初始化可能导致框架行为未定义。
  - 创建自定义组件时支持通过状态变量名设置初始值：在创建组件实例时，可以通过变量名显式指定@State状态属性的初始值。

@State示例

```
@Entry
@Component
struct Index {
  @State myVal: number = 0

  build() {
    Column(){
      Text(`${this.myVal}`)
        .fontSize(50)
        .fontWeight(FontWeight.Bold)
        .margin({top: 30})
      Button('加 1')
        .width('100')
        .onClick(()=> {
          this.myVal++
        })
    }
    .width('100%')
    .height('100%')
  }
}
```



@Prop

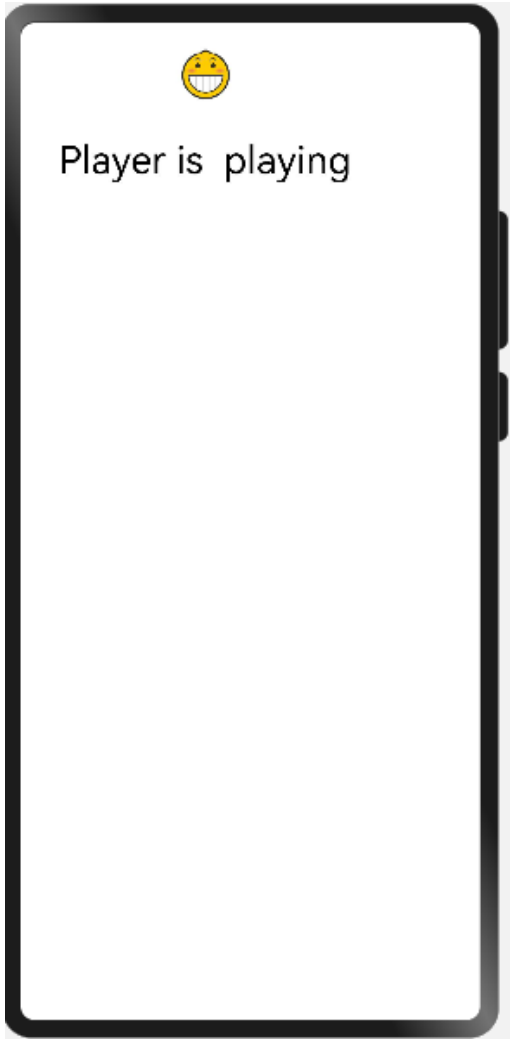
- @Prop与@State有相同的语义，但初始化方式不同。@Prop装饰的变量必须使用其父组件提供的@State变量进行初始化，允许组件内部修改@Prop变量，但更改不会通知给父组件，即@Prop属于单向数据绑定。
- @Prop状态数据具有以下特征：
  - 支持简单类型：仅支持number、string、boolean简单类型；
  - 私有：仅在组件内访问；
  - 支持多个实例：一个组件中可以定义多个标有@Prop的属性；
- 创建自定义组件时将值传递给@Prop变量进行初始化：在创建组件的新实例时，必须初始化所有@Prop变量，不支持在组件内部进行初始化。

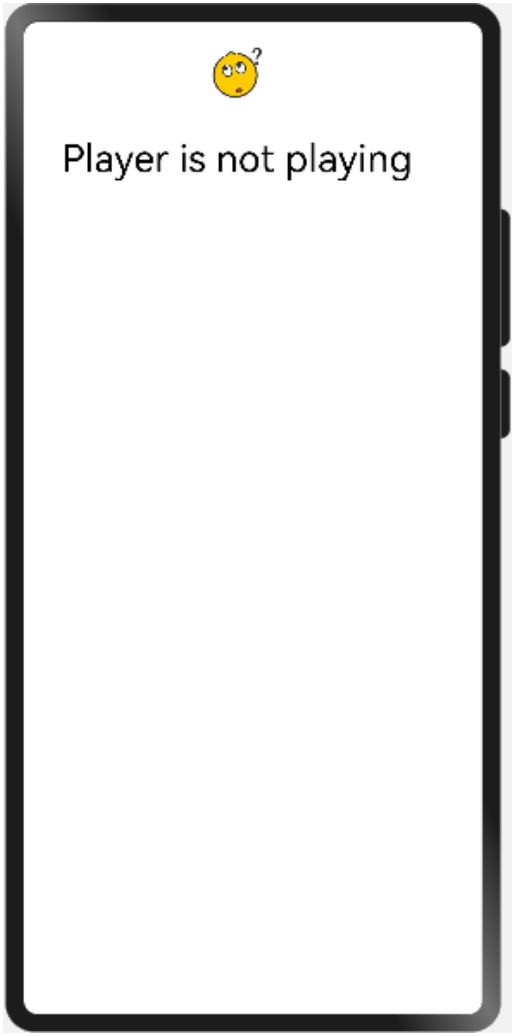
@Link

- @Link装饰的变量可以和父组件的@State变量建立双向数据绑定；
- 支持多种类型：@Link变量的值与@State变量的类型相同，即class、number、string、boolean或这些类型的数组；
- 私有：仅在组件内访问；
- 单个数据源：初始化@Link变量的父组件的变量必须是@State变量；
- 双向通信：子组件对@Link变量的更改将同步修改父组件的@State变量；
- 创建自定义组件时需要将变量的引用传递给@Link变量，在创建组件的新实例时，必须使用命名参数初始化所有@Link变量。@Link变量可以使用@State变量或@Link变量的引用进行初始化，@State变量可以通过“$”操作符创建引用。

@Link示例

```
@Entry
@Component
struct index {
  @State isPlaying: boolean = true
  build() {
    Column() {
      PlayButton({buttonPlaying: $isPlaying})
      Text(`Player is ${this.isPlaying? '':'not'} playing`)
        .fontSize(30) .margin(30)
    }
  }
}
@Component
struct PlayButton {
  @Link buttonPlaying: boolean
  build() {
    Column() {
      Button({ type: ButtonType.Circle, stateEffect: true }) {
        Image(this.buttonPlaying?
$r('app.media.play') :$r('app.media.pause'))
      }.onClick(() => {
        this.buttonPlaying = !this.buttonPlaying
      })
      .margin({top:20})
    }
  }
}
```





HUAWEI

@Consume和@Provide数据管理

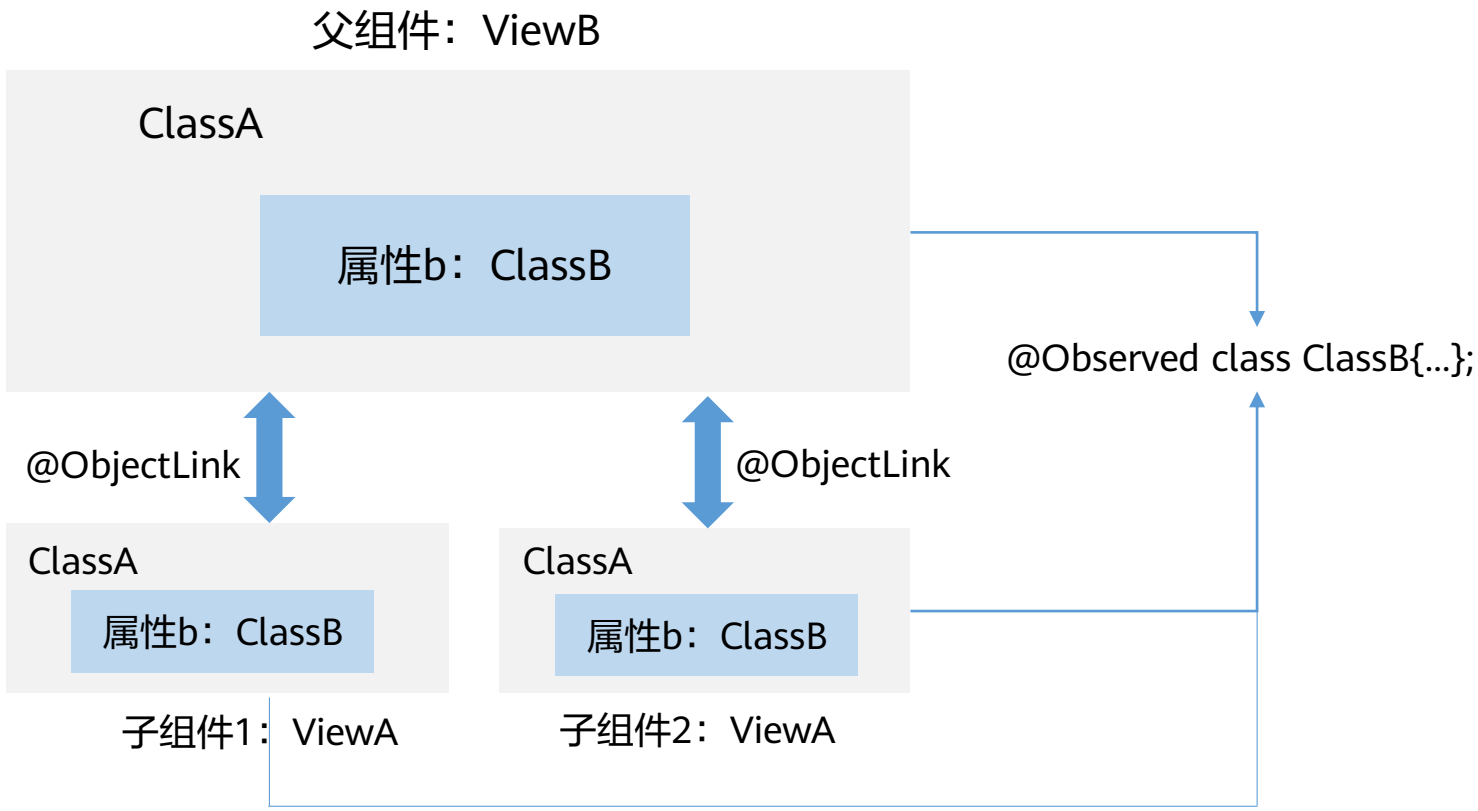
- @Provide作为数据的提供方，可以更新其子孙节点的数据，并触发页面渲染。
- @Consume在感知到@Provide数据的更新后，会触发当前view的重新渲染。

```
@Entry
@Component
struct ProvidePage {
  @Provide count: number = 1
  build() {
    Column() {
      //蓝色数字
      Text(`${this.count}`).fontColor(Color.Blue).fontSize(30)
      //使用ConsumeComponent的时候并没有将
ProvidePage.count传给ConsumeComponent
      ConsumeComponent()
    }.height('100%')
    .width('100%')
    .alignItems(HorizontalAlign.Center)
    .justifyContent(FlexAlign.Center)

  }
}
```

```
@Component
struct ConsumeComponent {
  //变量名要跟@Provide一样
  @Consume count: number
  build() {
    Column() {
      //红色数字
      Text(this.count.toString()).fontSize(30)
        .fontColor(Color.Red).margin({ top: 30 })
      //点击+1
      Text("点击+1").fontSize(30).onClick(() => {
        this.count++
      }).margin({ top: 30 })
    }

  }
}
```

@Observed和@ObjectLink数据管理

- @Observed应用于类，表示该类中的数据变更被UI页面管理，例如：@Observed class ClassA {}。@ObjectLink应用于被@Observed所装饰类的对象，例如：@ObjectLink a: ClassA。



@Watch

- @Watch用于监听状态变量的变化，语法结构为：

```
@State @Watch("onChanged") count : number = 0
```

- 如上给状态变量增加一个@Watch装饰器，通过@Watch注册一个回调方法onChanged，当状态变量count被改变时， 触发onChanged回调。
- 装饰器@State、@Prop、@Link、@ObjectLink、@Provide、@Consume、@StorageProp以及@StorageLink装饰的变量可以监听其变化。

目录

条件渲染

- 通过if/else条件渲染在页面上显示自定义条件的内容。
- if条件语句可以使用状态变量。
- 使用if可以使子组件的渲染依赖条件语句。
- 必须在容器组件内使用。
- 某些容器组件限制子组件的类型或数量。将if放置在这些组件内时，这些限制将应用于if和else语句内创建的组件。例如，Grid组件的子组件仅支持GridItem组件，在Grid组件内使用if时，则if条件语句内仅允许使用GridItem组件。

```
Column() {
    if (this.count > 0) {
        Text('count is positive')
    } else {
        Text('count is Negative')
    }
}
```

循环渲染

- 通过循环渲染（ForEach组件）来迭代数组，并为每个数组项创建相应的组件，可减少代码复杂度。

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

本章小结

- 描述本章主要讲述了使用ArkTS语言开发时装饰器的使用方法，通过了解各类装饰器的作用，进行代码讲解及效果图显示，能够熟练运用装饰器进行自定义组件的开发。

思考题

1. （多选题）右侧代码中，哪几项属于装饰器?

   A. @Entry

   B. @Component

   C. Struct

   D. @State

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

  build() {
    Row() {
      Column() {
        Text(this.message)
          .fontSize(50)
          .fontWeight(FontWeight.Bold)
      }
      .width('100%')
    }
    .height('100%')
  }
}
```

思考题

2. （判断题）用@Entry装饰的自定义组件用作页面的默认入口组件，加载页面时，将首先创建并呈现@Entry装饰的自定义组件，在单个源文件中，有且仅有一个@Entry。（ ）

A. 正确

B. 错误

学习推荐

- 官方学习网站
  - HarmonyOS官网：https://developer.harmonyos.com/
  - HarmonyOS应用开发文档： https://developer.huawei.com/consumer/cn/
  - OpenHarmony官网：https://edu.huaweicloud.com/
  - 华为开发者论坛： https://developer.huawei.com/consumer/cn/forum/

华为云开发者学堂

感谢