第3章

ArkTS语言快速入门

3.1 ArkUI 与 ArkTS 概述

ArkUI 开发框架是方舟开发框架的简称，它是一套构建 HarmonyOS/OpenHarmony 应用界面的声明式 UI 开发框架，它使用极简的 UI 信息语法、丰富的 UI 组件以及实时界面语言工具，帮助开发者提升应用界面开发效率 30%，开发者只需要使用官方提供的 API，就能在多个 HarmonyOS/OpenHarmony 设备上实现既丰富又流畅的用户界面体验。

鸿蒙 UI 开发框架 ArkUI 支持多种编程语言，为了简化开发以及降低学习难度，本书将使用 ArkTS 语言作为主要开发语言。在正式进入应用程序开发之前，读者不妨先对 ArkTS 语言进行简要的了解。若要说到 ArkTS 语言，与之相关联的则是 JS(JavaScript)和 TS(TypeScript)这两种已有一定应用规模的语言。本章将从 JS 到 TS 再到 ArkTS 的发展顺序入手，了解 ArkTS 语言的前世今生，再着重对 ArkTS 语言的语法展开介绍。

3.1.1 JS 语言和 TS 语言

JS 语言由 Mozilla 创造，最初主要是为了解决页面中的逻辑交互问题，它和 HTML(负责页面内容)、CSS(负责页面布局和样式)共同组成了 Web 页面/应用开发的基础。后期随着 Web 和浏览器的普及，以及 Node.js 将 JS 扩展到浏览器以外的环境，JS 语言得到了飞速的发展。与之相应，为了提升开发效率，一大批框架不断涌现出来，例如当前常用的 React.js 和 Vue.js。

伴随着 JS 的迅速发展，在 JS 开发中伴随的问题也相应产生。在大型工程中会涉及较复杂的代码以及较多的团队协作，对语言的规范性，模块的复用性、扩展性以及相关的开发工具都提出了更高的要求，为了解决这些痛点，Microsoft 在 JS 的基础上，创建了 TS 语言。

TS 在 JS 的基础上引入了类型系统，并提供了类型检查以及类型自动推导能力，可以进行编译时错误检查。在类型系统的基础上，引入了声明文件来管理接口和其他的自定义类型，便于各个模块之间的分工协作。同时，TS 也有相应的编辑器、编译器、IDE 插件等相关工具。此外，TS 可以通过编译器编译出原生 JS 应用；同时，JS 的应用也是合法的 TS 应用，所以 TS 对 JS 生态也做出了较好的补充。

3.1.2　ArkTS

如上所述，基于JS的前端框架以及TS的引入，进一步提升了应用开发效率，但依然存在一些不足。从开发者视角来看，开发一个应用需要了解3种语言(JS/TS、HTML和CSS)。对非Web开发者来说，这无疑是较为沉重的负担；在运行时维度来看，尽管TS有了类型的加持，但也只用于编译时的检查，运行时引擎还是无法利用到基于类型系统的优化。

ArkTS是鸿蒙生态的一种应用开发语言，从ArkTS的名字也可以看出，ArkTS是在TS语言的基础上，进行了相应的扩展以适应鸿蒙应用程序的开发，可以说ArkTS是TS的超集，而TS则是JS的超集。

由于JS/TS的应用较为广泛，生态较为完善，所以ArkTS选择以TS为基础进行扩展，继承了TS的所有特性。当前，ArkTS在TS的基础上主要扩展了以下能力。

(1) 基本UI描述：ArkTS定义了各种装饰器、自定义组件、UI描述机制，再配合UI开发框架中的UI内置组件、事件方法、属性方法等共同构成了UI开发的主体。

(2) 状态管理：ArkTS提供了多维度的状态管理机制，在UI开发框架中，和UI相关联的数据，不仅可以在组件内使用，还可以在不同组件层级间传递，比如父子组件之间、爷孙组件之间，也可以是全局范围内的传递，还可以是跨设备传递。另外，从数据的传递形式来看，可分为只读的单向传递和可变更的双向传递。开发者可以灵活地利用这些能力来实现数据和UI的联动。

(3) 动态构建UI元素：ArkTS提供了动态构建UI元素的能力，不仅可自定义组件内部的UI结构，还可复用组件样式，扩展原生组件。

(4) 渲染控制：ArkTS提供了渲染控制的能力。条件渲染可根据应用的不同状态，渲染对应状态下的部分内容。循环渲染可从数据源中迭代获取数据，并在每次迭代过程中创建相应的组件。

(5) 使用限制与扩展：ArkTS在使用过程中存在限制与约束，同时也扩展了双向绑定等能力。

与声明式UI相对应的是命令式UI，两者的区别在于声明式UI将页面声明出来而不需要手动更新，界面会自动完成更新。而在传统的命令式UI中，UI在xml文件中被定义，若要修改UI中的某个元素，则需要手动使用代码命令UI进行更新。在声明式UI的框架下，自动更新的不仅是数据，页面中的任何元素发生更改时均可自动被更新。声明式UI不只是一种良好的代码风格，更是一种强大的开发功能，对代码的开发和运行效率提升均有一定的帮助。以下给出简单示例，帮助读者体会声明式UI中元素的自动更新特性。

```
1.    // entry/src/main/ets/pages/Index.ets
2.    @Entry
3.    @Preview
4.    @Component
5.    struct Index1 {
6.      @State message: string = 'Hello HarmonyOS'
7.      @State text_show : boolean = true
8.
9.      build() {
10.       Row() {
```

```
11.          Column() {
12.            Text(this.message)
13.              .fontSize(50)
14.              .fontWeight(FontWeight.Bold)
15.              .onClick(() =>{
16.                if(this.text_show){
17.                  this.text_show = false
18.                  this.message = 'hello world'
19.                }else{
20.                  this.text_show = true
21.                }
22.              })
23.            if (this.text_show){
24.              Text(this.message)
25.                .fontSize(50)
26.                .fontWeight(FontWeight.Bold)
27.            }
28.          }
29.          .width('100%')
30.        }
31.        .height('100%')
32.      }
33.    }
```

在工程创建生成的初始代码的基础上做出一些修改，具体的创建过程见 1.3 节。在模板现有代码的基础上，定义一个由@State 修饰器修饰的 boolean 类型的变量 text_show 来控制第二个 Text 组件的显示。对原有的 Text 组件添加一个单击监听事件，当该组件被单击触控时该事件被触发，在该事件中通过改变 text_show 的值来控制第二个组件是否被展示，并在 if 语句中修改原有 Text 组件展示的文本内容。图 3-1 展示了在本示例中，声明式 UI 范式对数据和组件的自动更新。当读者学习到这里时，可能会对此感到困难，不用担心，本书将在后面的例子中一一讲解对应的知识点。

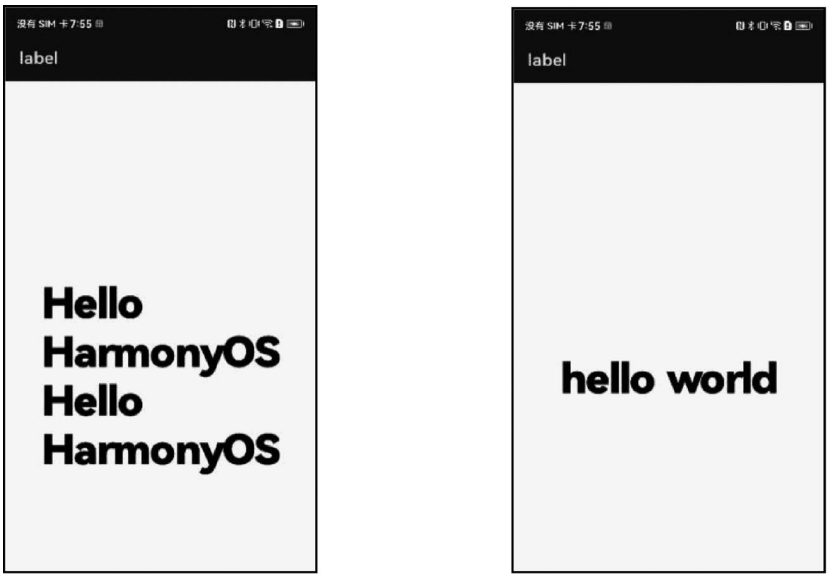

图 3-1　声明式 UI 自动更新示意图

在本章后续内容中将从 TS 语言的变量和数据类型入手，TS 作为强类型语言，对于变量的要求与 JS 略有不同，不被声明数据类型的变量在 TS 中是不被允许使用的。因此，本节从 TS 语言基本知识入手，随后着重讲解 ArkTS 相对 TS 以及 JS 的扩展知识。

3.2　TypeScript 基础知识

从本节开始，将展开对 TS 语言语法的讲解。TS 语言作为强类型语言，对于变量的使用与 JS 有很大的不同，不被声明数据类型的变量在 TS 中是不被允许使用的，这样做的好处是可将许多 JS 项目在运行阶段出现的错误在编译阶段解决。

3.2.1　数据类型

数据类型是强类型语言编程中一个非常重要的概念，相当于对变量施加的限制，表 3-1 为 TS 语言中的变量介绍。

表 3-1　变量类型介绍

数据类型	关键字	描述
任意类型	any	any 类型的变量可以被赋予任何类型的值

数字类型	number	双精度浮点值，被用来表示整数和浮点数
		`let binaryLiteral: number = 0b1010;　　// 二进制`
		`let octalLiteral: number = 0o744;　　// 八进制`
		`let decLiteral: number = 6;　　// 十进制`
		`let hexLiteral: number = 0xf00d;　　// 十六进制`
		注意：TypeScript 和 JavaScript 没有整数类型

字符串类型	string	使用单引号或双引号表示字符串类型，反引号定义多行文本和内嵌表达式
		`let name: string = "HarmonyOS";`
		`let years: number = 2;`
		`let words: string = `您好，今年是 ${ name } 发布 ${ years + 1} 周年`;`

布尔类型	boolean	表示逻辑值 true 和 false `let flag: boolean = true;`

数组类型	无	变量声明为数组
		`// 在元素类型后面加上[]`
		`let arr: number[] = [1, 2];`
		`// 或者使用数组泛型`
		`let arr: Array<number> = [1, 2];`

元组	无	元组类型标识已知元素类型和数量的数组，元组中各个元素的类型不必相同，但对应位置的元素类型应该相同
		`let x: [string, number];`
		`x = ['HarmonyOS', 1];　　// 运行正常`
		`x = [1, 'HarmonyOS '];　　// 报错`
		`console.log(x[0]);　　// 输出 Runoob`

续表

数据类型	关键字	描述

枚举	enum	枚举类型用于定义数值集合
		enum Color {Red, Green, Blue};
		let c: Color = Color.Blue;
		console.log(c); // 输出 2

void	void	用于标识方法返回值的类型
		function hello(): void {
		alert("Hello HarmonyOS ");
		}

null	null	表示对象值缺失
undefined	undefined	用于初始化变量为一个未定义的值
never	never	其他类型,代表不会出现的值

3.2.2 变量声明

TS是一门灵活的编程语言,但在TS中声明变量仍需要遵循以下规则:

(1) 变量名称可以包含数字和字母。

(2) 除了下画线_和美元$符号外,不能包含其他特殊字符,包括空格。

(3) 变量名不能以数字开头。

变量使用前必须先声明,在TS中,常用的变量声明关键词有const,let,var三种。

1. const

const声明的是一个只读的常量,一旦赋值,就不能再改变它的值了。常量通常使用大写字母命名,以便于与变量区分。

```
const num = 1
const obj = {name:'Alice',age: 18}
obj.age = 20                    // 可以修改属性
obj = {}                        //报错,不能重新赋值
```

2. let

let作用域为块级作用域。和var不同,let声明的变量只能在声明的块级作用域内访问,而不能在外部访问。

```
if (true) {
let x = 10
console.log(x)                  // 10
}
console.log(x)                  // 报错,x 未定义
```

3. var

var作用域为函数作用域或全局作用域。在函数内部声明的变量只能在函数内部访问,而在全局作用域声明的变量则可以在任何地方访问。此外,var声明的变量可以被重复声明,而重复声明时,只有最后一个声明有效。

```
function foo() {
  var x = 1
  if (true) {
    var x = 2
```

```
    }
    console.log(x)                    // 2
}
```

一般情况下，建议使用 const 或 let 来声明变量，这有助于避免一些常见的 JavaScript 的变量问题，如变量提升和作用域混乱等问题。

除了 const、let、var 外，在 TypeScript 中还有一些其他的声明变量的关键字。

(1) readonly：用于标识只读属性或只读数组，不能被重新赋值。

(2) static：用于标识类的静态属性或静态方法，不需要实例化类即可使用。

(3) abstract：用于标识抽象类或抽象方法，不能被实例化，只能被子类继承并实现。

(4) public、private、protected：用于标识类的属性或方法的访问权限。

3.2.3　控制语句

控制语句是用于控制程序的执行流程的语句。在 TypeScript 中，开发者可以使用各种控制语句来实现条件执行、循环和代码跳转等操作。以下是 TypeScript 中常见的控制语句的介绍和示例代码。

1. 条件语句

条件语句根据给定的条件决定是否执行特定的代码块。

(1) if 语句：if 语句用于根据条件执行特定的代码块。

```
let num: number = 10;

if (num > 0) {
  console.log("Number is positive.");
} else if (num < 0) {
  console.log("Number is negative.");
} else {
  console.log("Number is zero.");
}
```

(2) switch 语句：switch 语句根据表达式的值来匹配不同的情况，并执行相应的代码块。

```
let day: number = 3;
let dayName: string;

switch (day) {
  case 1:
    dayName = "Monday";
    break;
  case 2:
    dayName = "Tuesday";
    break;
case 3:
    dayName = "Wednesday";
    break;
  default:
    dayName = "Unknown";
}

console.log(`Today is ${dayName}.`);              //输出 Today is Wednesday .
```

2. 循环语句

循环语句用于重复执行特定的代码块，分为以下几种循环。

(1) for 循环：for 循环通过初始化语句、条件表达式和更新语句来控制循环的执行次数。

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

(2) while 循环：while 循环在条件为真时重复执行代码块。

```
let i = 1;

while (i <= 5) {
  console.log(i);
  i++;
}
```

(3) do-while 循环：do-while 循环首先执行一次代码块，然后在条件为真时重复执行。

```
let i = 1;

do {
  console.log(i);
  i++;
} while (i <= 5);
```

3. 跳转语句

跳转语句用于在代码中进行跳转操作。

(1) break 语句：break 语句用于终止循环或跳出 switch 语句的执行。

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    break;                                  // 终止循环
  }
  console.log(i);
}                                           //输出 1 2
```

(2) continue 语句：continue 语句用于跳过当前迭代并继续下一次迭代。

```
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue;                               // 终止循环
  }
  console.log(i);                           // 输出 1 2 4 5
}
```

(3) return 语句：结束程序返回结果的语句。

```
function addNumbers(a: number, b: number): number {
  if (a < 0 || b < 0) {
    return -1;                              // 返回负数并终止函数执行
  }
  return a + b;                             // 返回两个数字的和
}

console.log(addNumbers(2, 3));              // 输出: 5
console.log(addNumbers(-2, 3));             // 输出: -1
```

3.2.4　函数

函数是一组一起执行一个任务的语句。开发者可以把代码划分到不同的函数中。如何将代码划分到不同的函数中是由开发者决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务进行的。

函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体。

接下来通过以下代码示例介绍 TypeScrip 中函数的使用方法。

```
// TypeScript 函数的介绍和使用方法示例

// 定义一个简单的函数,接收两个数字参数并返回它们的和
function addNumbers(a: number, b: number): number {
  return a + b;
}

// 调用函数并打印结果
console.log(addNumbers(2, 3));                    // 输出: 5

// 定义一个函数,接收一个字符串参数,并返回字符串的长度
function getStringLength(str: string): number {
  return str.length;
}

// 调用函数并打印结果
console.log(getStringLength("Hello"));            // 输出: 5

// 定义一个函数,接收一个数组参数,并返回数组中所有元素的和
function sumArray(numbers: number[]): number {
  let sum = 0;
  for (let num of numbers) {
    sum += num;
  }
  return sum;
}

// 调用函数并打印结果
console.log(sumArray([1, 2, 3, 4, 5]));           // 输出: 15
```

在上述示例中首先定义了一个简单的函数 addNumbers，它接收两个数字参数 a 和 b，并指定它们的类型为 number。函数的返回类型也被指定为 number，表示函数将返回一个数字。函数体内部通过将两个参数相加并返回结果实现了求和的逻辑。

接下来定义了另一个函数 getStringLength，它接收一个字符串参数 str，并返回字符串的长度(数字类型)。函数体内部使用了字符串的 length 属性来获取字符串的长度，并将其作为返回值。

最后定义了一个函数 sumArray，它接收一个数字数组参数 numbers，并返回数组中所有元素的和。函数内部使用了循环来遍历数组中的每个元素，并将其累加到变量 sum 中，最后将 sum 返回作为结果。

通过在函数定义中指定参数类型和返回类型，TypeScript 提供了更好的类型检查和类型推断功能，可以帮助用户在编写代码时捕获潜在的类型错误，并提供更好的开发体验。用户可以根据自己的需求为函数添加参数和返回类型，并根据函数的具体功能实现函数体内

部的逻辑。

3.2.5 类

TypeScript 是面向对象的 JavaScript。类描述了所创建的对象共同的属性和方法。TypeScript 支持面向对象的所有特性，比如类、接口等。

TypeScript 类定义方式如下：

```
class class_name {
  // 类作用域
}
```

定义类的关键字为 class，后面紧跟类名，类可以包含以下几个模块(类的数据成员)。

(1) 字段：字段是类里面声明的变量。字段表示对象的有关数据。

(2) 构造函数：类实例化时调用，可以为类的对象分配内存。

(3) 方法：方法为对象要执行的操作。

本书将通过以下例子介绍 TypeScript 中类的使用方法。

```
class Mobile{
    // 字段
    MobileName:string;

    // 构造函数
    constructor(MobileName:string) {
        this. MobileName = MobileName
    }

    // 方法
    showMobileName():void {
        console.log("函数中显示设备型号 : " + this.MobileName)
    }
}

// 创建一个对象
var obj = new Mobile("HuaweiP60")

// 访问字段
console.log("读取设备型号 : " + obj.MobileName)

// 访问方法
obj.showMobileName()
```

在上述代码示例代码中定义了一个名为 Mobile 类，它有一个字段 MobileName 和一个构造函数。构造函数接收一个参数 MobileName，并将其赋值给类的字段 MobileName。类还有一个名为 showMobileName 的方法，用于打印设备型号。紧接着创建了一个 Mobile 类的对象 obj，通过传递"HuaweiP60"作为参数来实例化它。然后，通过使用 obj. MobileName 就可以访问字段，并使用 obj. showMobileName() 调用方法。

当运行这段代码时，它将输出以下内容：

```
读取设备型号: HuaweiP60
函数中显示设备型号: HuaweiP60
```

3.2.6　命名空间和模块

良好的代码组织和管理有助于提高代码可读性，便于推动团队合作及后续代码维护。TypeScript 为开发人员提供了模块(Modules)和命名空间(Namespaces)功能以便于组织和管理代码。

1. 模块 Modules

模块是一种将代码组织为可重用和可组合的单元的方式。与命名空间不同，模块将代码分割为多个文件，每个文件可以包含一个或多个模块。模块使用 export 关键字导出需要暴露的内容，并使用 import 关键字引入其他模块中的内容控制语句。

```
//moduleA.ts
// 导出一个变量
export const num = 100
export const str = 'Hello HarmonyOS '
export const reg = /^鸿蒙开发 $ /

// 导出一个函数
export function fn() {}

// 导出一个类
export class Student {}
export class Person extends People {}

// 导出一个接口
export interface Users {}

//Ixdex.ts
import { str as s } from './moduleA'
console.log(s)
```

在上述代码示例中，在 moduleA.ts 文件中定义了多个变量和方法，在 Index.ts 中引入了 str 变量并且起别名为 s，执行 Index.ts 文件，使用者将会看到输出 Hello HormonyOS。

2. 命名空间 Namespaces

命名空间是一种将相关的代码组织在一起的方式，它可以防止全局命名冲突。可以将相关的类、接口、函数和其他类型放置在命名空间中，并使用该命名空间访问其中的内容。命名空间使用 namespace 关键字定义。在模块内容的介绍中，读者已经学习了使用 export 关键词用于暴露代码供其他模块使用，但是当引入多个文件时，如果多个文件中有相同命名的变量或方法就会产生冲突，命名空间则为开发者解决了这一问题。

```
// 定义一个命名空间
namespace MyNamespace {
  // 导出一个变量
  export const num: number = 100;

  // 导出一个函数
  export function greet(name: string): void {
    console.log(`Hello, ${name}!`);
  }

  // 导出一个类
  export class Person {
```

```
        constructor(private name: string) {}

        public sayHello(): void {
          console.log(`Hello, my name is ${this.name}.`);
        }
      }
    }

    // 使用命名空间中的成员
    console.log(MyNamespace.num);
    MyNamespace.greet("John");

    const person = new MyNamespace.Person("Alice");
    person.sayHello();
```

在上述示例中定义了一个名为 MyNamespace 的命名空间。在命名空间中导出了一个变量 num，一个函数 greet，以及一个类 Person。这些成员可以通过 MyNamespace 访问到。

在代码的最后部分，通过 MyNamespace.num 访问了命名空间中的变量，调用了命名空间中的函数 MyNamespace.greet("John")，以及创建了一个命名空间中的类的实例，并调用了其中的方法。

注意事项：

(1) 使用命名空间时，可以在一个文件中定义一个命名空间，也可以将命名空间拆分到多个文件中，使用<reference path="otherFile.ts">来引用其他文件中的命名空间。

(2) 当命名空间的内容较多或需要与其他命名空间进行交互时，可以使用 import 和 export 来引入和导出命名空间的成员。

(3) 命名空间的名称应具有描述性，并遵循合理的命名规范，以保证代码的可读性和维护性。

(4) 命名空间可以嵌套，形成多层次的命名空间结构，以进一步组织和封装代码。

通过使用命名空间，开发者可以更好地组织和管理 TypeScript 代码，避免全局命名冲突，并提供更好的封装性和可维护性。

3.2.7 迭代器

TypeScript 中的迭代器(Iterator)表示一种流接口，用于顺序访问集合中的元素。它可以逐个访问集合内的元素，而不需要关心集合的内部实现。

要实现一个迭代器，需要实现以下两点。

(1) 一个 next()方法，它返回一个包含两个属性的对象。

① done：boolean 类型，表示迭代是否结束。

② value：表示当前迭代的值。

(2) 实现 Iterable 接口，该接口要求实现一个方法：Symbol.iterator，该方法需要返回一个迭代器对象。

```
class Numbers implements Iterable<number> {
  [Symbol.iterator]() {
    let n = 0;
    return {
      next() {
        n += 1;
```

```
            return { value: n, done: false };
          }
        }
      }
    }

    let numbers = new Numbers();
    for (let n of numbers) {
      console.log(n);
    }
```

for…of 和 for…in 均可迭代一个列表，但是用于迭代的值却不同：for…in 迭代的是对象的键，而 for…of 则迭代的是对象的值。

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i);                 // "0", "1", "2",
}

for (let i of list) {
    console.log(i);                 // "4", "5", "6"
}
```

3.3　使用 ArkTS

在上一节中，已经基本介绍了 TS 语言的基本知识，所以在此就可以使用 ArkTS 语言构建鸿蒙应用程序了，下面用一个简单的例子开始对 ArkTS 的基本组成。如图 3-2 所示的代码示例，UI 界面包含两段文本、一条分割线和一个按钮，当开发者单击按钮时，文本内容会从“Hello World”变为“Hello ArkUI”。

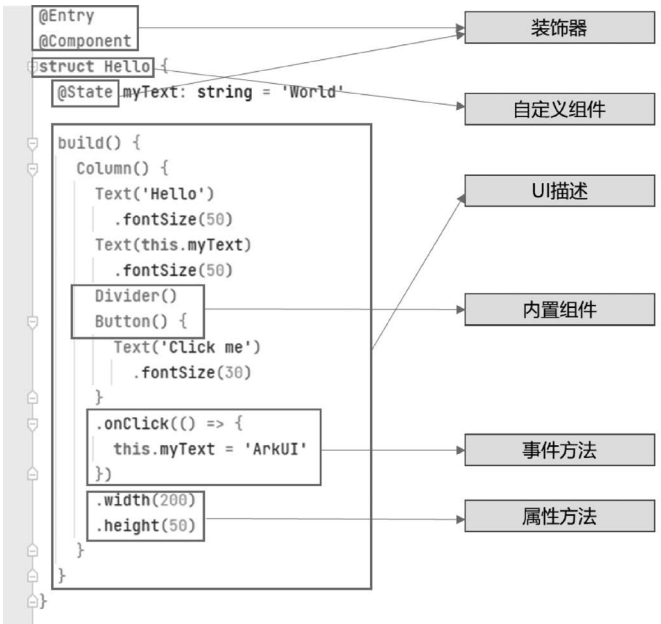

图 3-2　ArkTS 框架示意图

这个示例中包含的 ArkTS 声明式开发范式的基本组成说明如下。

(1) 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。具体而言，@Component 表示这是个自定义组件；@Entry 表示这是个入口组件；@State 表示组件中的状态变量，这个状态变化会引起 UI 变更。

(2) 自定义组件：可复用的 UI 单元，可组合其他组件，如上述被 @Component 装饰的 struct Hello。

(3) 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。

(4) 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。

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

(6) 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()。

在接下来的内容中将逐步讲解上述内容。

3.3.1 自定义组件基本结构

在 ArkUI 中，UI 显示的内容均为组件，由框架直接提供的称为系统组件，由开发者定义的称为自定义组件。在进行 UI 界面设计时，通常并不是简单地将系统组件进行组合使用，而是需要考虑代码可复用性、业务逻辑与 UI 分离、后续版本演进等因素。因此，将 UI 和部分业务逻辑封装成自定义组件是不可或缺的能力。ArkTS 通过装饰器@Component 和@Entry 装饰 struct 关键字声明的数据结构，构成一个自定义组件。自定义组件中提供了一个 build 函数，开发者需在该函数内以链式调用的方式进行基本的 UI 描述。

1. 基本概念

(1) struct：自定义组件可以基于 struct 实现，不能有继承关系，对于 struct 的实例化，可以省略 new。

(2) 装饰器：装饰器给被装饰的对象赋予某种能力，其不仅可以装饰类或结构体，还可以装饰类的属性。多个装饰器可以叠加到目标元素上，定义在同行中或者分开多行，推荐分开多行定义，@Entry、@Component、@ Preview 和@State 均为常用的装饰器。

```
1.    @Entry
2.    @Component
3.    struct MyComponent {
4.    }
```

(3) build()函数：自定义组件必须定义 build()函数，并且禁止自定义构造函数。build()函数满足 Builder 构造器接口定义，用于定义组件的声明式 UI 描述，通过声明式 UI 构建界面及功能。在@Entry 装饰的自定义组件 build 函数中，开发者必须指定一个且唯一的根节点，根节点必须为容器组件，例如 Row()和 Column()，而非页面组件同样需要一个唯一的根节点，但可以为非容器组件，例如 Image()，接着在根节点里使用声明式 UI 对页面进行构建。

(4) @Component：装饰 struct，结构体在装饰后具有基于组件的能力，需要实现 build

方法来创建 UI。

（5）@Entry：装饰 struct，组件被装饰后作为页面的入口，页面加载时将被渲染显示，在同一个文件中，可以有多个组件定义，但只能有一个组件被@Entry 装饰作为页面的入口组件。

（6）@Preview：装饰 struct，用@Preview 装饰的自定义组件可以在 DevEco Studio 的预览器上进行实时预览，加载页面时，将创建并显示@Preview 装饰的自定义组件。

（7）@State：装饰组件中的状态变量，当被@State 装饰的变量值发生变化时，组件就会重新构建对应的 UI。除了@State，ArkTS 还提供了多种状态变量装饰器用来更好地管理组件内的通信问题，将会在后面的章节介绍其用法。

（8）链式调用：以"." 链式调用的方式配置 UI 组件的属性方法、事件方法等。

以下通过一段代码示例帮助读者更好地了解声明式 UI 代码是如何编写的。

```
1.    //entry/src/main/ets/pages/index2.ets
2.    @Entry
3.    @Component
4.    @Preview
5.    struct Index2 {
6.       @State message: string = 'Hello World1'
7.      UnStateMessage:string = "Hello World2"
8.      build() {
9.        Row() {
10.         Column() {
11.           Button("change string").onClick(() =>{
12.             this.message = "New World1"
13.             this.UnStakeMessage = " New World2"
14.           })
15.           Text(this.message)
16.             .fontSize(50)
17.             .fontWeight(FontWeight.Bold)
18.           Text(this.UnStakeMessage)
19.             .fontSize(50)
20.             .fontWeight(FontWeight.Bold)
21.         }
22.         .width('100%')
23.       }
24.       .height('50%')
25.     }
26.   }
```

在上述代码示例中，定义了两个变量 message 和 UnStateMessage，其中 message 变量使用@State 装饰器进行修饰，这使得其值变化时可以通知页面进行重新渲染，而 UnStateMessage 并不具备这一属性。在 build 构造函数中，使用 Row()作为组件根节点，也就是在根节点中配置的组件将按照行的形式展开。在根节点中 Row 中，又使用了 Column 列节点，在列节点中配置的组件以列的形式排布，节点之间层层嵌套，有序地构成了 UI 界面见图 3-3。在 Column 节点中，配置了一个 Button 按钮组件和两个 Text 文本组件，赋予按钮一个单击事件用于改变两个文本组件中使用的字符串的值，单击事件触发后，使用者就可以看到使用@State 装饰的变量的文本组件被重新渲染，而与之相对应的使用普通字符串变量的组件则毫无变化。

图 3-3　UI 特性展示

2. UI 描述规范

为了完整地构建组件，通常需要考虑以下 4 个配置。

1）参数配置

组件的本质是一种由 ArkUI 定义好的类，和其他类一样，组件本身也拥有构造函数，构造函数分为有参构造函数和无参的默认构造函数，如果组件的接口定义中不包含必选构造参数，组件后面的“()”中不需要配置任何内容，也就是不需要传入参数即可使用。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“()”中必须配置相应参数，参数可以使用常量进行赋值。

```
Column() {
  Image('./test.jpg')
  Image(this.imagePath)
  Image('https://' + this.imageUrl)
  //$r形式引入应用资源，可用于多语言场景
  Text($r('app.string.title_value'))
  Text(`count: ${this.count}`)
  Text()
}
```

在上述代码示例中，Image 必须传入参数才可以使用，这里的参数可以使用变量或表达式来构造 Image 组件的参数，也可以使用字符串形式的常量进行配置。

2）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 Text 组件的字体大小。

```
Text('test')
  .fontSize(12)
```

（2）配置组件的多个属性。

```
Image('test.jpg')
  .alt('error.jpg')
  .width(100)
  .height(100)
```

(3) 除了直接传递常量参数外,还可以传递变量或表达式。

```
Text('hello')
  .fontSize(this.size)
Image('test.jpg')
  .width(this.count % 2 === 0? 100 : 200)
  .height(this.offset + 100)
```

(4) 对于系统组件,ArkUI 还为其属性预定义了一些枚举类型供开发者调用,枚举类型可以作为参数传递,但必须满足参数类型要求。例如,可以按以下方式配置 Text 组件的颜色和字体样式。

```
Text('hello')
  .fontSize(20)
  .fontColor(Color.Red)
  .fontWeight(FontWeight.Bold)
```

3) 事件配置

通过事件方法可以配置组件支持的事件,事件方法紧随组件,并用"."运算符连接。

(1) 使用 lambda 表达式配置组件的事件方法:

```
Button('Click me')
  .onClick(() => {
    this.myText = 'ArkUI';
  })
```

(2) 使用匿名函数表达式配置组件的事件方法,要求使用 bind,以确保函数体中的 this 引用包含的组件:

```
Button('add counter')
  .onClick(function(){
    this.counter += 2;
  }.bind(this))
```

(3) 使用组件的成员函数配置组件的事件方法:

```
myClickHandler(): void {
  this.counter += 2;
}
...
Button('add counter')
  .onClick(this.myClickHandler.bind(this))
```

4) 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。

以下是简单的 Column 组件配置子组件的示例。

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

容器组件均支持子组件配置，可以实现相对复杂的多级嵌套。

```
Column() {
  Row() {
    Image('test1.jpg')
      .width(100)
      .height(100)
    Button('click +1')
      .onClick(() => {
        console.info('+1 clicked!');
      })
  }
}
```

3.3.2 页面和自定义组件生命周期

当开发应用程序时，生命周期函数是与组件（或对象）相关联的特殊方法，它们在组件的生命周期中的不同阶段被调用。这些函数允许用户在适当的时候执行特定的操作，例如初始化组件、处理数据更新、销毁组件等。

在开始之前，读者需要先明确自定义组件和页面的关系。

自定义组件：@Component 装饰的 UI 单元，可以组合多个系统组件实现 UI 的复用。

页面：应用的 UI 页面。可以由一个或者多个自定义组件组成，@Entry 装饰的自定义组件为页面的入口组件，即页面的根节点，一个页面有且仅能有一个@Entry。只有被@Entry 装饰的组件才可以调用页面的生命周期。

页面生命周期，即被@Entry 装饰的组件生命周期，提供以下生命周期接口。

（1）onPageShow：页面每次显示时触发。

（2）onPageHide：页面每次隐藏时触发一次。

（3）onBackPress：当用户单击“返回”按钮时触发。

组件生命周期，即一般用@Component 装饰的自定义组件的生命周期，提供以下生命周期接口。

（1）aboutToAppear：组件即将出现时回调该接口，具体时机为在创建自定义组件的新实例后，在执行其 build()函数之前执行。

（2）aboutToDisappear：在自定义组件即将析构销毁时执行。

生命周期流程如图 3-4 所示，展示的是被@Entry 装饰的组件（首页）生命周期。

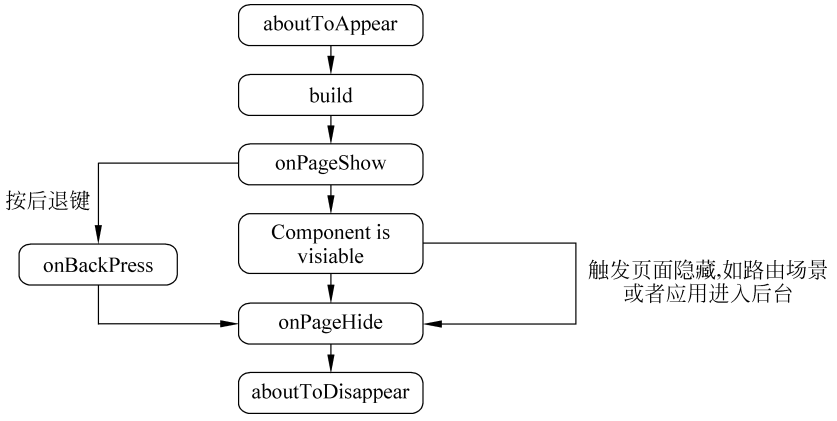

图 3-4 页面生命周期流程示意图

以下示例展示了生命周期的调用时机。

```
//entry/src/main/ets/pages/index3.ets
import router from '@ohos.router';

@Entry
@Component
struct MyComponent {
  @State showChild: boolean = true;

  // 只有被@Entry装饰的组件才可以调用页面的生命周期
  onPageShow() {
    console.info('Index onPageShow');
  }
  // 只有被@Entry装饰的组件才可以调用页面的生命周期
  onPageHide() {
    console.info('Index onPageHide');
  }

  // 只有被@Entry装饰的组件才可以调用页面的生命周期
  onBackPress() {
    console.info('Index onBackPress');
  }

  // 组件生命周期
  aboutToAppear() {
    console.info('MyComponent aboutToAppear');
  }

  // 组件生命周期
  aboutToDisappear() {
    console.info('MyComponent aboutToDisappear');
  }

  build() {
    Column() {
      // this.showChild为true,创建Child子组件,执行Child aboutToAppear
      if (this.showChild) {
        Child()
      }
      // this.showChild为false,删除Child子组件,执行Child aboutToDisappear
      Button('create or delete Child').onClick(() => {
        this.showChild = false;
      })
      // push到Page2页面,执行onPageHide
      Button('push to next page')
        .onClick(() => {
          router.pushUrl({ url: 'pages/Index2' });
        })
    }

  }
}

@Component
struct Child {
  @State title: string = 'Hello World';
  // 组件生命周期
```

```
57.      aboutToDisappear() {
58.        console.info('[lifeCycle] Child aboutToDisappear')
59.      }
60.      // 组件生命周期
61.      aboutToAppear() {
62.        console.info('[lifeCycle] Child aboutToAppear')
63.      }
64.
65.      build() {
66.        Text(this.title).fontSize(50).onClick(() => {
67.          this.title = 'Hello ArkUI';
68.        })
69.      }
70.    }
```

以上示例中，Index 页面包含两个自定义组件，一个是被@Entry 装饰的 MyComponent，也是页面的入口组件，即页面的根节点；另一个是 Child，是 MyComponent 的子组件。只有@Entry 装饰的节点才可以生效页面的生命周期方法，所以 MyComponent 中声明了当前 Index 页面的页面生命周期函数。同时 MyComponent 和其子组件 Child 也声明了组件的生命周期函数。

（1）应用冷启动的初始化流程为：MyComponent aboutToAppear→MyComponent build→Child aboutToAppear→Child build→Child build 执行完毕→MyComponent build 执行完毕→Index onPageShow。

（2）单击“delete Child”，if 绑定的 this.showChild 变成 false，删除 Child 组件，会执行 Child aboutToDisappear 方法。

（3）单击“push to next page”，调用 router.pushUrl 接口，跳转到另外一个页面，当前 Index 页面隐藏，执行页面生命周期 Index onPageHide。此处调用的是 router.pushUrl 接口，Index 页面被隐藏，并没有销毁，所以只调用 onPageHide。跳转到新页面后，执行初始化新页面的生命周期的流程。

（4）如果调用的是 router.replaceUrl，则当前 Index 页面被销毁，执行的生命周期流程将变为：Index onPageHide→MyComponent aboutToDisappear→Child aboutToDisappear。上文已经提到，组件的销毁是从组件树上直接摘下子树，所以先调用父组件的 aboutToDisappear，再调用子组件的 aboutToDisappear，然后执行初始化新页面的生命周期流程。

（5）单击“返回”按钮，触发页面生命周期 IndexonBackPress。最小化应用或者应用进入后台，触发 Index onPageHide。这两个状态下应用都没有被销毁，所以并不会执行组件的 aboutToDisappear。应用回到前台，执行 Index onPageShow。

（6）退出应用，执行 Index onPageHide→MyComponent aboutToDisappear→Child aboutToDisappear。

3.4 其他装饰器

除了前文提到的@Component、@Entry、@Preview、@State 这 4 种最常见的装饰器外，ArkTS 还为开发者提供了一系列其他强有力的装饰器配合开发者进行开发。

3.4.1　@Builder 装饰器：用于自定义构建函数

前面章节介绍了如何创建一个自定义组件。该自定义组件内部 UI 结构固定，仅与使用方进行数据传递。ArkUI 还提供了一种更轻量的 UI 元素复用机制@Builder，@Builder 装饰的函数遵循 build()函数语法规则，开发者可以将重复使用的 UI 元素抽象成一个方法，在 build 方法里调用。

为了简化语言，@Builder 装饰的函数也称为“自定义构建函数”。

自定义构建函数分为组件内自定义构建函数和全局自定义构建函数，在组件内定义的构建函数需要遵循以下规则：

（1）允许在自定义组件内定义一个或多个自定义构建函数，该函数被认为是该组件的私有、特殊类型的成员函数。

（2）自定义构建函数可以在所属组件的 build 方法和其他自定义构建函数中调用，但不允许在组件外调用。

（3）在自定义函数体中，this 指代当前所属组件，组件的状态变量可以在自定义构建函数内访问。建议通过 this 访问自定义组件的状态变量而不是参数传递。

而全局的自定义构建函数可以被整个应用获取，配合前文介绍的命名空间和模块功能可以更好地实现模块化开发。

```
//组件内自定义构建函数
@Builder myBuilderFunction({ … })
//组件内自定义构建函数使用方法
this.myBuilderFunction({ … })

//全局自定义构建函数
@Builder function MyGlobalBuilderFunction({ … })
//全局自定义构建函数使用方法,注意在全局自定义构建函数中不允许使用 this 和 bind 方法
MyGlobalBuilderFunction()
```

自定义构建函数提供了两种传参方式，即按值传递和按引用传递。使用按引用传递参数时，传递的参数可为状态变量，且状态变量的改变会引起@Builder 方法内的 UI 刷新。ArkUI 提供 $ $（单个 $ 代表解析表达式）作为按引用传递参数的范式。以下是使用按引用传递参数代码示例。

```
//entry/src/main/pages/Abuilder.ets
@Builder function ABuilder( $ $ : { paramA1: string }) {
  Row() {
    Text(`UseStateVarByReference: ${ $ $ .paramA1} `)
  }
}
@Entry
@Component
struct Parent {
  @State label: string = 'Hello';
  build() {
    Column() {
      // 在 Parent 组件中调用 ABuilder 的时候,将 this.label 引用传递给 ABuilder
      ABuilder({ paramA1: this.label })
      Button('Click me').onClick(() => {
        // 单击"Click me"后,UI 从"Hello"刷新为"ArkUI"
```

```
17.            this.label = 'ArkUI';
18.          })
19.        }
20.      }
21.    }
```

注意：当使用引用方式传递参数时，不要试图在自定义构建函数中修改父组件传递过来的参数，这并不会引起父组件刷新 UI 重新渲染，反而会导致错误产生。在后面章节中，读者将会学习到其他方法用来解决父子组件中参数通信问题。

3.4.2 @BuilderParam 装饰器

当开发者创建了自定义组件，并想对该组件添加特定功能时，例如在自定义组件中添加一个单击跳转操作。若直接在组件内嵌入事件方法，将会导致所有引入该自定义组件的地方均增加了该功能。为解决此问题，ArkUI 引入了@BuilderParam 装饰器，@BuilderParam 用来装饰指向@Builder 方法的变量，开发者可在初始化自定义组件时对此属性进行赋值，为自定义组件增加特定的功能。通俗地讲，开发者在设计自定义组件时，提前预留了位置给使用此自定义组件的开发人员进行功能扩展。该装饰器用于声明任意 UI 描述的一个元素，类似 Vue 开发框架中的 slot 占位符。

```
1.    //entry/src/main/pages/builderPadram.ets
2.    @Builder function GlobalBuilder1( $ $ : {label: string }) {
3.      Text( $ $ .label)
4.        .width(400)
5.        .height(50)
6.        .backgroundColor(Color.Blue)
7.    }
8.
9.    @Component
10.   struct Child {
11.     label: string = 'Child'
12.     // 无参数类型,指向的 componentBuilder 也是无参数类型
13.     @BuilderParam aBuilder0: () => void;
14.     // 有参数类型,指向的 GlobalBuilder1 也是有参数类型的方法
15.     @BuilderParam aBuilder1: ( $ $ : { label : string}) => void;
16.
17.     build() {
18.       Column() {
19.         this.aBuilder0()
20.         this.aBuilder1({label: 'global Builder label'} )
21.       }
22.     }
23.   }
24.
25.   @Entry
26.   @Component
27.   struct BuildParent {
28.     label: string = 'Parent'
29.
30.     @Builder componentBuilder() {
31.       Text(`${this.label}`)
32.     }
33.
34.     build() {
```

```
35.       Column() {
36.         this.componentBuilder()
37.         Child({ aBuilder0: this.componentBuilder, aBuilder1: GlobalBuilder1 })
38.       }
39.     }
40.   }
```

在上述代码示例中，在子组件 Child 中定义了两个由@BuilderParam 装饰的函数，当在父组件 Parent 中使用时，分别通过参数传入两个自定义构造函数以达到扩展 Child 组件的功能。

3.4.3　@Styles 装饰器

如果每个组件的样式都需要单独设置，在开发过程中会出现大量代码进行重复样式设置，虽然可以复制粘贴，但为了代码简洁性和后续方便维护，本节推出了可以提炼公共样式进行复用的装饰器@Styles。@Styles 装饰器可以将多条样式设置提炼成一个方法，直接在组件声明的位置调用。组件内@Styles 的优先级高于全局@Styles。框架优先找当前组件内的@Styles，如果找不到，则会全局查找。通过@Styles 装饰器可以快速定义并复用自定义样式。

```
1.    // 定义在全局的@Styles 封装的样式
2.    //entry/src/main/ets/pages/Styls.ets
3.    @Styles function globalFancy  () {
4.      .width(150)
5.      .height(100)
6.      .backgroundColor(Color.Pink)
7.    }
8.
9.    @Entry
10.   @Component
11.   struct FancyUse {
12.     @State heightVlaue: number = 100
13.     // 定义在组件内的@Styles 封装的样式
14.     @Styles fancy() {
15.       .width(200)
16.       .height(this.heightVlaue)
17.       .backgroundColor(Color.Yellow)
18.       .onClick(() => {
19.         this.heightVlaue = 200
20.       })
21.     }
22.
23.     build() {
24.       Column({ space: 10 }) {
25.         // 使用全局的@Styles 封装的样式
26.         Text('FancyA')
27.           .globalFancy ()
28.           .fontSize(30)
29.         // 使用组件内的@Styles 封装的样式
30.         Text('FancyB')
31.           .fancy()
32.           .fontSize(30)
33.       }
34.     }
35.   }
```

在上述代码示例中，分别定义了全局@Styles 装饰器和组件内@Styles 装饰器，并且在 FancyUse 中给出了使用方法。

3.4.4 stateStyles

stateStyles 不属于装饰器种类，但搭配@Styles 装饰器可以依据组件内部状态的不同，快速设置不同样式。这就是本章要介绍的内容 stateStyles（又称为“多态样式”）。stateStyles 类似于 Web 开发中的 css 伪类，但语法不同。ArkUI 为开发者提供了以下 4 种状态。

（1）focused：获焦态。

（2）normal：正常态。

（3）pressed：按压态。

（4）disabled：不可用态。

下面的示例展示了 stateStyles 最基本的使用场景。Button 处于第一个组件，默认获焦，生效 focused 指定的粉色样式。按压时显示为 pressed 态指定的黑色。如果在 Button 前再放一个组件，使其不处于获焦态，就会生效 normal 态的黄色。

```
@Entry
@Component
struct StateStylesSample {
  build() {
    Column() {
      Button('Click me')
        .stateStyles({
          focused: {
            .backgroundColor(Color.Pink)
          },
          pressed: {
            .backgroundColor(Color.Black)
          },
          normal: {
            .backgroundColor(Color.Yellow)
          }
        })
    }.margin('30%')
  }
}
```

以下示例通过@Styles 指定 stateStyles 的不同状态。

```
//entry/src/main/ets/pages/stateStyles.ets
@Entry
@Component
struct MyComponent {
  @Styles normalStyle() {
    .backgroundColor(Color.Gray)
  }

  @Styles pressedStyle() {
    .backgroundColor(Color.Red)
  }

  build() {
```

```
14.         Column() {
15.           Text('Text1')
16.             .fontSize(50)
17.             .fontColor(Color.White)
18.             .stateStyles({
19.               normal: this.normalStyle,
20.               pressed: this.pressedStyle,
21.             })
22.         }
23.       }
24.     }
```

在此示例中，当单击文本框时背景颜色变为红色，此段示例仅做简单的展示。

3.5　状态管理

在前文中，提到由@State装饰器装饰的变量发生变化时会引起UI渲染，并且当开发者用引用的方式从父组件中向子组件中传递参数时，当此变量在父组件中变化时会引起子组件相关变量一起重现渲染。但是到目前为止，还没有一种方法可以实现从子组件中改变变量使父组件重现渲染，但是借助状态管理装饰器就可以实现此功能。本节将延续3.4节中的装饰器内容继续介绍用于状态管理的装饰器。

从数据的传递形式和同步类型层面看，装饰器也可分为：

（1）只读的单向传递；

（2）可变更的双向传递。

状态图示如图3-5所示，在后面将一一介绍各个装饰器的用法。开发者可以灵活地利用这些能力来实现数据和UI的联动。

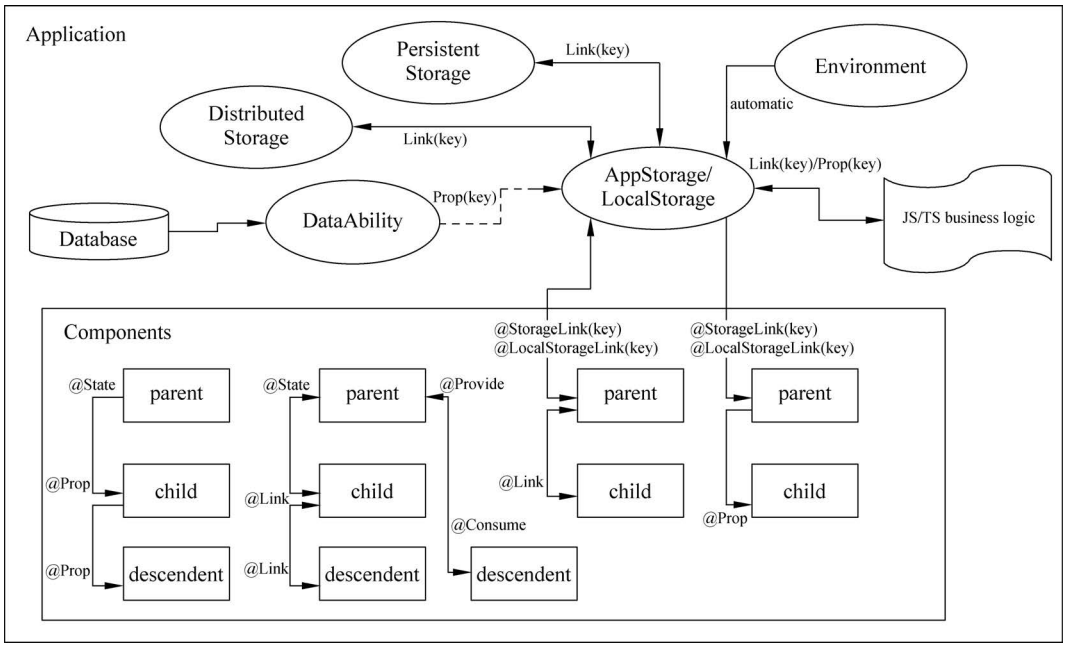

图3-5　页面级状态管理与应用关系图

在图 3-5 中，Components 部分的装饰器为组件级别的状态管理，Application 部分为应用的状态管理。开发者可以通过@StorageLink/@LocalStorageLink 和@StorageProp/@LocalStorageProp 实现应用和组件状态的双向和单向同步。图中箭头方向为数据同步方向，单箭头为单向同步，双箭头为双向同步。

3.5.1 @State 装饰器

在前文中，已经讲解了@State 装饰的变量，或称为状态变量，一旦变量拥有了状态属性，就和自定义组件的渲染绑定起来。当状态改变时，UI 会发生对应的渲染改变。

但是并不是状态变量的所有更改都会引起 UI 的刷新，只有可以被框架观察到的修改才会引起 UI 刷新。本小节会介绍什么样的修改才能被观察到，以及观察到变化后，框架是怎么引起 UI 刷新的，即框架的行为表现是什么。

当装饰的数据类型为 boolean、string、number 类型时，可以观察到数值的变化。

```
@State count: number = 0;
// 此变量变化可以被观察到
this.count = 1;
```

当装饰的数据类型为 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。示例如下。

```
//声明 ClassA 和 Model 类.
class ClassA {
  public value: string;

  constructor(value: string) {
    this.value = value;
  }
}

class Model {
  public value: string;
  public name: ClassA;
  constructor(value: string, a: ClassA) {
    this.value = value;
    this.name = a;
  }
}
// class 类型
@State title: Model = new Model('Hello', new ClassA('World'));
// class 类型赋值
this.title = new Model('Hi', new ClassA('ArkUI'));
// class 属性的赋值
this.title.value = 'Hi'
// 嵌套的属性赋值观察不到
this.title.name.value = 'ArkUI'
```

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。示例如下。

```
//@State 装饰的对象为 Model 类型数组时
@State title: Model[] = [new Model(11), new Model(1)]
//数组自身的赋值可以观察到
this.title = [new Model(2)]
```

```
    //数组项的赋值可以观察到
    this.title[0] = new Model(2)
    //删除数组项可以观察到
    this.title.pop()
    //新增数组项可以观察到
    this.title.push(new Model(12))
```

3.5.2　@Prop 装饰器

在前文中介绍到可以使用$ $的方式传递引用参数，虽然在子组件中可以接收父组件参数变化重新渲染，但是这种方式仍有一种弊端，即开发者不能在子组件中修改引用的参数，在本节中使用@Prop 装饰的变量可以和父组件建立单向的同步关系。@Prop 装饰的变量是可变的，但是变化不会同步回其父组件。

以下示例是@State 到子组件@Prop 简单数据同步，父组件 ParentComponent 的状态变量 countDownStartValue 初始化子组件 CountDownComponent 中@Prop 装饰的 count，单击“Try again”，count 的修改仅保留在 CountDownComponent，不会同步给父组件 CountDownComponent。

ParentComponent 的状态变量 countDownStartValue 的变化将重置 CountDownComponent 的 count。

```
//entry/src/main/ets/pages/Prop_example.ets
@Component
struct CountDownComponent {
  @Prop count: number;
  costOfOneAttempt: number = 1;

  build() {
    Column() {
      if (this.count > 0) {
        Text(`You have ${this.count} Nuggets left`)
      } else {
        Text('Game over!')
      }
      // @Prop 装饰的变量不会同步给父组件
      Button(`Try again`).onClick(() => {
        this.count -= this.costOfOneAttempt;
      })
    }
  }
}

@Entry
@Component
struct ParentComponent {
  @State countDownStartValue: number = 10;

  build() {
    Column() {
      Text(`Grant ${this.countDownStartValue} nuggets to play.`)
      // 父组件的数据源的修改会同步给子组件
      Button(`+1 - Nuggets in New Game`).onClick(() => {
        this.countDownStartValue += 1;
      })
      // 父组件的修改会同步给子组件
      Button(`-1  - Nuggets in New Game`).onClick(() => {
```

```
36.             this.countDownStartValue -= 1;
37.           })
38.
39.           CountDownComponent({ count: this.countDownStartValue, costOfOneAttempt: 2 })
40.         }
41.       }
42.     }
```

在上面的代码示例中：

(1) CountDownComponent 子组件首次创建时其@Prop 装饰的 count 变量将从父组件@State 装饰的 countDownStartValue 变量初始化。

(2) 按“+1”或“-1”按钮时，父组件的@State 装饰的 countDown-StartValue 值会变化，这将触发父组件重新渲染，在父组件重新渲染过程中会刷新使用 countDownStartValue 状态变量的 UI 组件并单向同步更新 CountDownComponent 子组件中的 count 值。

(3) 更新 count 状态变量值也会触发 CountDownComponent 的重新渲染，在重新渲染过程中，评估使用 count 状态变量的 if 语句条件(this. count > 0)，并执行 true 分支中的使用 count 状态变量的 UI 组件相关描述来更新 Text 组件的 UI 显示。

(4) 当按下子组件 CountDownComponent 的“Try again”按钮时，其@Prop 变量 count 将被更改，但是 count 值的更改不会影响父组件的 countDownStartValue 值。

(5) 父组件的 countDownStartValue 值会变化时，父组件的修改将覆盖子组件 CountDownComponent 中 count 本地的修改。

3.5.3 @Link 装饰器

在前文介绍的装饰器中所提供的功能中，父组件可以改变子组件中的值，但是还没有一种办法可以实现从子组件中同步变化到父组件中，@Link 装饰器为开发者提供了这一功能。

为了了解@Link 变量初始化和更新机制，有必要先了解父组件和拥有@Link 变量的子组件的关系，以及初始渲染和双向更新的流程(以父组件为@State 为例)。

(1) 初始渲染：执行父组件的 build()函数后将创建子组件的新实例。初始化过程包括以下内容。

① 必须指定父组件中的@State 变量，用于初始化子组件的@Link 变量。子组件的@Link 变量值与其父组件的数据源变量保持同步(双向数据同步)。

② 父组件的@State 状态变量包装类通过构造函数传给子组件，子组件的@Link 包装类得到父组件的@State 的状态变量后，将当前@Link 包装类 this 指针注册给父组件的@State 变量。

(2) @Link 数据源的更新：即父组件中状态变量更新，引起相关子组件的@Link 的更新。

① 通过初始渲染的步骤可知，子组件@Link 包装类把当前 this 指针注册给父组件。父组件@State 变量变更后，会遍历更新所有依赖它的系统组件(elementId)和状态变量(如@Link 包装类)。

② 通知@Link 包装类更新后，子组件中所有依赖@Link 状态变量的系统组件(elementId)都会被通知更新，以此实现父组件对子组件的状态数据同步。

③ @Link 的更新：当子组件中@Link 更新后，处理步骤如下（以父组件为@State 为例）：@Link 更新后，调用父组件的@State 包装类的 set 方法，将更新后的数值同步回父组件。

子组件@Link 和父组件@State 分别遍历依赖的系统组件，进行对应的 UI 的更新，以此实现子组件@Link 同步回父组件@State。

接下来通过一个简单的代码示例了解@Link 装饰器用法。

```
//entry/src/main/ets/pages/Link_example.ets
class GreenButtonState {
  width: number = 0;
  constructor(width: number) {
    this.width = width;
  }
}
@Component
struct GreenButton {
  @Link greenButtonState: GreenButtonState;
  build() {
    Button('Green Button')
      .width(this.greenButtonState.width)
      .height(150.0)
      .backgroundColor('#00ff00')
      .onClick(() => {
        if (this.greenButtonState.width < 700) {
          // 更新 class 的属性，变化可以被观察到同步回父组件
          this.greenButtonState.width += 125;
        } else {
          // 更新 class，变化可以被观察到同步回父组件
          this.greenButtonState = new GreenButtonState(100);
        }
      })
  }
}
@Component
struct YellowButton {
  @Link yellowButtonState: number;
  build() {
    Button('Yellow Button')
      .width(this.yellowButtonState)
      .height(150.0)
      .backgroundColor('#ffff00')
      .onClick(() => {
        // 子组件的简单类型可以同步回父组件
        this.yellowButtonState += 50.0;
      })
  }
}
@Entry
@Component
struct ShufflingContainer {
  @State greenButtonState: GreenButtonState = new GreenButtonState(300);
  @State yellowButtonProp: number = 100;
  build() {
    Column() {
      // 简单类型从父组件@State 向子组件@Link 数据同步
      Button('Parent View: Set yellowButton')
```

```
50.             .onClick(() => {
51.               this.yellowButtonProp = (this.yellowButtonProp < 700)? this.yellowButtonProp +
                  100 : 100;
52.             })
53.           // class 类型从父组件@State 向子组件@Link 数据同步
54.           Button('Parent View: Set GreenButton')
55.             .onClick(() => {
56.               this.greenButtonState.width = (this.greenButtonState.width < 700)? this.
                  greenButtonState.width + 100 : 100;
57.             })
58.           // class 类型初始化@Link
59.           GreenButton({ greenButtonState: $greenButtonState })
60.           // 简单类型初始化@Link
61.           YellowButton({ yellowButtonState: $yellowButtonProp })
62.         }
63.       }
64.     }
```

以上述代码示例中，单击父组件 ShufflingContainer 中的“Parent View：Set yellowButton”和“Parent View：Set GreenButton”，可以从父组件将变化同步给子组件，子组件 GreenButton 和 YellowButton 中@Link 装饰变量的变化也会同步给其父组件。

3.5.4 @Provide 装饰器和@Consume 装饰器

前文中介绍的装饰器只能用于父子组件内通信，本节将要介绍的@Provide 和@Consume 装饰器，应用于与后代组件的双向数据同步，以及状态数据在多个层级之间传递的场景。不同于上文提到的父子组件之间通过命名参数机制传递，@Provide 和@Consume 摆脱参数传递机制的束缚，实现跨层级传递。

其中，@Provide 装饰的变量是在祖先节点中，可以理解为被“提供”给后代的状态变量。@Consume 装饰的变量是在后代组件中，去“消费（绑定）”祖先节点提供的变量。

@Provide/@Consume 装饰的状态变量有以下特性：

（1）@Provide 装饰的状态变量自动对其所有后代组件可用，即该变量被“provide”给它的后代组件。由此可见，@Provide 的方便之处在于，开发者不需要多次在组件之间传递变量。

（2）后代组件通过使用@Consume 去获取@Provide 提供的变量，建立@Provide 和@Consume 之间的双向数据同步，与@State/@Link 不同的是，前者可以在多层级的父子组件之间传递。

（3）@Provide 和@Consume 可以通过相同的变量名或者相同的变量别名绑定，变量类型必须相同。

```
1.  // 通过相同的变量名绑定
2.  @Provide a: number = 0;
3.  @Consume a: number;
4.
5.  // 通过相同的变量别名绑定
6.  @Provide('a') b: number = 0;
7.  @Consume('a') c: number;
```

@Provide 和@Consume 通过相同的变量名或者相同的变量别名绑定时，@Provide 修饰的变量和@Consume 修饰的变量是一对多的关系。不允许在同一个自定义组件内，包括其子组件中声明多个同名或者同别名的@Provide 装饰的变量。

下面的示例是与后代组件双向同步状态@Provide 和@Consume 场景。当分别单击 CompA 和 CompD 组件内 Button 时，reviewVotes 的更改会双向同步在 CompA 和 CompD 中。

```
//entry/src/main/ets/pages/Provide_Consume.ets
@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@Provide 装饰的变量
  @Consume reviewVotes: number;

  build() {
    Column() {
      Text(`reviewVotes( ${this.reviewVotes})`)
      Button(`reviewVotes( ${this.reviewVotes}), give +1`)
       .onClick(() => this.reviewVotes += 1)
    }
    .width('50%')
  }
}

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

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

@Entry
@Component
struct CompA {
  // @Provide 装饰的变量 reviewVotes 由入口组件 CompA 提供其后代组件
  @Provide reviewVotes: number = 0;

  build() {
    Column() {
      Button(`reviewVotes( ${this.reviewVotes}), give +1`)
        .onClick(() => this.reviewVotes += 1)
      CompB()
    }
  }
}
```

3.6　应用间状态通信

3.5 节中介绍的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 根据不同特性，提供了多种应用状态管理的能力。

（1）LocalStorage：页面级 UI 状态存储，通常用于 UIAbility 内、页面间的状态共享。

（2）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。

（3）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。

（4）Environment：应用程序运行的设备的环境参数，环境参数会同步到 AppStorage 中，可以和 AppStorage 搭配使用。

3.6.1 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 也可以在 UIAbility 内、页面间共享状态。

本书仅介绍 LocalStorage 使用场景和相关的装饰器：@LocalStorageProp 和@LocalStorageLink。

LocalStorage 是 ArkTS 为构建页面级别状态变量提供存储的内存内“数据库”。

（1）应用程序可以创建多个 LocalStorage 实例，LocalStorage 实例可以在页面内共享，也可以通过 GetShared 接口，获取在 UIAbility 里创建的 GetShared，实现跨页面、UIAbility 内共享。

（2）组件树的根节点，即被@Entry 装饰的@Component，可以被分配一个 LocalStorage 实例，此组件的所有子组件实例将自动获得对该 LocalStorage 实例的访问权限。

（3）被@Component 装饰的组件最多可以访问一个 LocalStorage 实例和 AppStorage，未被@Entry 装饰的组件不可被独立分配 LocalStorage 实例，只能接收父组件通过@Entry 传递来的 LocalStorage 实例。一个 LocalStorage 实例在组件树上可以被分配给多个组件。

（4）LocalStorage 中的所有属性都是可变的。

（5）应用程序决定 LocalStorage 对象的生命周期。当应用释放最后一个指向 LocalStorage 的引用时，比如销毁最后一个自定义组件，LocalStorage 将被 JS Engine 垃圾回收。

（6）LocalStorage 根据与@Component 装饰的组件的同步类型不同，提供了两个装饰器。

① @LocalStorageProp：@LocalStorageProp 装饰的变量与 LocalStorage 中给定属性建立单行同步关系。

② @LocalStorageLink：@LocalStorageLink 装饰的变量和在@Component 中创建与 LocalStorage 中给定属性建立双向同步关系，也就是说修改@LocalStorageLink 装饰的变量会同步返回到 LocalStorage 中，并影响其他被@LocalStorageProp 装饰的同一 key 变量。

1. 应用逻辑使用 LocalStorage

```
let storage = new LocalStorage({ 'PropA': 47 });  // 创建新实例并使用给定对象初始化
let propA = storage.get('PropA')                  // propA == 47
let link1 = storage.link('PropA');                // link1.get() == 47
let link2 = storage.link('PropA');                // link2.get() == 47
```

```
let prop = storage.prop('PropA');                    // prop.get() = 47
link1.set(48);       // link1.get() == link2.get() == prop.get() == 48
prop.set(1);         //  prop.get() = 1; 但是 link1.get() == link2.get() == 48
link1.set(49);       //  link1.get() == link2.get() == prop.get() == 49
```

以上代码示例介绍了在应用逻辑内使用 LocalStorage，可以看到以 prop 方式获取的变量被修改时，修改不会同步返回在 LocalStorage 中存储的数值，但是以 link 方式获取的变量会同步返回给 LocalStorage 对应的 key 值，并且修改其他所有用到此 key 的变量。

2. 从 UI 内部使用 LocalStorage

除了应用程序逻辑使用 LocalStorage，还可以借助 LocalStorage 相关的两个装饰器@LocalStorageProp 和@LocalStorageLink，在 UI 组件内部获取 LocalStorage 实例中存储的状态变量。

接下来的代码示例以@LocalStorage 为例展示以下内容：

（1）使用构造函数创建 LocalStorage 实例 storage；

（2）使用@Entry 装饰器将 storage 添加到 CompA 顶层组件中；

（3）@LocalStorageLink 绑定 LocalStorage 给定的属性，建立双向数据同步。@LocalStorageProp 装饰器装饰的变量只能单向数据同步。

```
//ets/src/main/ets/pages/LocalStorage.ets
// 创建新实例并使用给定对象初始化
let storage = new LocalStorage({ 'PropA': 42, 'PropB':2023});
// 使 LocalStorage 可从@Component 组件访问
@Entry(storage)
@Component
struct CompLocal {
  // @LocalStorageProp 变量装饰器与 LocalStorage 中的'PropA'属性建立单向绑定
  // @LocalStorageLink 变量装饰器与 LocalStorage 中的'PropB'属性建立双向绑定
  @LocalStorageProp('PropA') storProp1: number = 1;
  @LocalStorageLink('PropB') storProp2: number = 2022
  build() {
    Column({ space: 15 }) {
      // 单击后从 42 开始加 1，只改变当前组件显示的 storProp1，不会同步到 LocalStorage 中
      Button(`父组件改变 PropA ${this.storProp1}`)
        .onClick(() => this.storProp1 += 1)
      // 单击后从 2023 开始加 1 ，数据会同步到 LocalStorage 中
      Button(`父组件改变 PropB ${this.storProp2}`)
        .onClick(() => this.storProp2 += 1)
      Child()
    }
  }
}

@Component
struct ChildLocal {
  // @LocalStorageProp 变量装饰器与 LocalStorage 中的'ProA'和'PropB'属性建立单向绑定
  @LocalStorageProp('PropA') storProp1: number = 2;
  @LocalStorageProp('PropB') storProp2: number = 2021;

  build() {
    Column({ space: 15 }) {
     // 当 CompA 改变时，当前 storProp1 不会改变，显示 42，storProp2 会随着改变
     Text(`Parent from LocalStorage ${this.storProp1}`)
     Text(`Parent from LocalStorage ${this.storProp2}`)
```

```
36.         }
37.     }
38.   }
```

在上述代码示例中，当用户单击父组件中的两个按钮时，可以在子组件中观察到对应变化。

3.6.2 AppStorage：应用全局的 UI 状态存储

AppStorage 是应用全局的 UI 状态存储，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。

和 LocalStorage 不同的是，LocalStorage 是页面级的，通常应用于页面内的数据共享。而对于 AppStorage，是应用级的全局状态共享。AppStorage 还相当于整个应用的“中枢”，持久化数据 PersistentStorage 和环境变量 Environment 都是通过和 AppStorage 中转，才可以和 UI 交互。

本书仅介绍 AppStorage 使用场景和相关的装饰器：@StorageProp 和@StorageLink。

1. @StorageProp

在上文中已经提到，如果要建立 AppStorage 和自定义组件的联系，需要使用@StorageProp 和@StorageLink 装饰器。使用@StorageProp(key)/@StorageLink(key)装饰组件内的变量，key 标识了 AppStorage 的属性。

当自定义组件初始化时，@StorageProp(key)/@StorageLink(key)装饰的变量会通过给定的 key，绑定在 AppStorage 对应属性，完成初始化。本地初始化是必要的，因为无法保证 AppStorage 一定存在给定的 key，这取决于应用逻辑，是否在组件初始化之前在 AppStorage 实例中存入对应的属性。

@StorageProp(key)是和 AppStorage 中 key 对应的属性建立单向数据同步，系统允许本地改变的发生，但是对于@StorageProp，本地的修改永远不会同步回 AppStorage 中；相反，如果 AppStorage 给定 key 的属性发生改变，改变会被同步给@StorageProp，并覆盖本地的修改。

2. @StorageLink

@StorageLink(key)是和 AppStorage 中 key 对应的属性建立双向数据同步。本地修改发生，该修改会被返回 AppStorage 中。

AppStorage 中的修改发生后，该修改会被同步到所有绑定 AppStorage 对应 key 的属性上，包括单向(@StorageProp 和通过 Prop 创建的单向绑定变量)、双向(@StorageLink 和通过 Link 创建的双向绑定变量)变量和其他实例(比如 PersistentStorage)。

3. 从应用逻辑使用 AppStorage 和 LocalStorage

AppStorage 是单例，它的所有 API 都是静态的，使用方法类似于 LocalStorage 中对应的非静态方法。

```
1.    AppStorage.SetOrCreate('PropA', 47);
2.
3.    let storage: LocalStorage = new LocalStorage({ 'PropA': 17 });
4.    let propA: number = AppStorage.Get('PropA') // propA in AppStorage == 47, propA in
      LocalStorage == 17
5.    var link1: SubscribedAbstractProperty<number> = AppStorage.Link('PropA'); // link1.
      get() == 47
6.    var link2: SubscribedAbstractProperty<number> = AppStorage.Link('PropA'); // link2.
```

```
    get() == 47
var prop: SubscribedAbstractProperty<number> = AppStorage.Prop('PropA'); // prop.get() = 47

link1.set(48); // two-way sync: link1.get() == link2.get() == prop.get() == 48
prop.set(1);   // one-way sync: prop.get() = 1; but link1.get() == link2.get() == 48
link1.set(49); // two-way sync: link1.get() == link2.get() == prop.get() == 49

storage.get('PropA')      // == 17
storage.set('PropA', 101);

storage.get('PropA')      // == 101

AppStorage.Get('PropA')   // == 49
link1.get()               // == 49
link2.get()               // == 49
prop.get()                // == 49
```

4. 从 UI 内部使用 AppStorage 和 LocalStorage

@StorageLink 变量装饰器与 AppStorage 配合使用，正如@ LocalStorageLink 与 LocalStorage 配合使用一样。此装饰器使用 AppStorage 中的属性创建双向数据同步。

```
//entry/src/main/ets/pages/AppStorage.ets
AppStorage.SetOrCreate('PropA', 47);
let storageApp = new LocalStorage({ 'PropA': 48 });

@Entry(storageApp)
@Component
struct CompAppStorage {
  @StorageLink('PropA') storLink: number = 1;
  @LocalStorageLink('PropA') localStorLink: number = 1;

  build() {
    Column({ space: 20 }) {
      Text(`From AppStorage ${this.storLink}`)
        .onClick(() => this.storLink += 1)

      Text(`From LocalStorage ${this.localStorLink}`)
        .onClick(() => this.localStorLink += 1)
    }
  }
}
```

3.6.3　PersistentStorage：持久化存储 UI 状态

3.6.1、3.6.2 小节介绍的 LocalStorage 和 AppStorage 都是运行时的内存，不能做到在应用退出再次启动后，依然保存选定的结果。为了能提供保存用户信息功能，就需要用到 PersistentStorage。

PersistentStorage 是应用程序中的可选单例对象。此对象的作用是持久化存储选定的 AppStorage 属性，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。

1. 概述

PersistentStorage 将选定的 AppStorage 属性保留在设备磁盘上。应用程序通过 API，以决定哪些 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，另外还有一些接口可以用于管理持久化属性，但是业务逻辑始终是通过 AppStorage 获取和设置属性的。

```
//entry/src/main/ets/pages/PersistentStorage.ets
//初始化 PersistentStorage
PersistentStorage.PersistProp('aProp', 47);
//在 AppStorage 获取对应属性
AppStorage.Get('aProp');
@Entry
@Component
struct Persistent {
  @State message: string = 'Hello World'
//组件内定部定义
  @StorageLink('aProp') aProp: number = 48

  build() {
    Row() {
      Column() {
        Text(this.message)
        // 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
        Text(`${this.aProp}`)
          .onClick(() => {
            this.aProp += 1;
          })
      }
    }
  }
}
```

以下过程讲述了新应用安装后首次启动时是怎样处理持久化存储过程的。

(1) 调用 PersistProp 初始化 PersistentStorage，首先查询在 PersistentStorage 本地文件中是否存在“aProp”，查询结果为不存在，因为应用是第一次安装。

(2) 接着查询属性“aProp”在 AppStorage 中是否存在，依旧不存在。

(3) 在 AppStorge 中创建名为“aProp”的 number 类型属性，属性初始值是定义的默认值 47。

(4) PersistentStorage 将属性“aProp”和值 47 写入磁盘，AppStorage 中“aProp”对应的值和其后续的更改将被持久化。

(5) 在 Index 组件中创建状态变量@StorageLink('aProp') aProp，和 AppStorage 中“aProp”双向绑定，在创建的过程中会在 AppStorage 中查找，成功找到“aProp”，所以使用其在 AppStorage 找到的值 47。

3.6.4 @Watch 装饰器：状态变量更改通知

@Watch 应用于对状态变量的监听。如果开发者需要关注某个状态变量的值是否改变，可以使用@Watch 为状态变量设置回调函数。@Watch 装饰器与@State、@Prop、@Link 等装饰器搭配使用，建议放在这些装饰器之后。以下过程简单解释@Watch 装饰器响应过程。

(1) 当观察到状态变量的变化(包括双向绑定的 AppStorage 和 LocalStorage 中对应的 key 发生的变化)时，对应的@Watch 的回调方法将被触发。

(2) @Watch 方法在自定义组件的属性变更之后同步执行。

（3）如果在@Watch的方法里改变了其他的状态变量，也会引起状态变更和@Watch的执行。

以下代码示例说明如何在子组件中观察@Link变量。

```
//entry/src/main/ets/pages/watch.ets
class PurchaseItem {
  static NextId: number = 0;
  public id: number;
  public price: number;

  constructor(price: number) {
    this.id = PurchaseItem.NextId++;
    this.price = price;
  }
}

@Component
struct BasketViewer {
  @Link @Watch('onBasketUpdated') shopBasket: PurchaseItem[];
  @State totalPurchase: number = 0;

  updateTotal(): number {
    let total = this.shopBasket.reduce((sum, i) => sum + i.price, 0);
    // 超过 100 欧元可享受折扣
    if (total >= 100) {
      total = 0.9 * total;
    }
    return total;
  }
  // @Watch 回调函数
  onBasketUpdated(propName: string): void {
    this.totalPurchase = this.updateTotal();
  }

  build() {
    Column() {
      ForEach(this.shopBasket,
        (item) => {
          Text(`Price: ${item.price.toFixed(2)} `)
        },
        item => item.id.toString()
      )
      Text(`Total: ${this.totalPurchase.toFixed(2)} `)
    }
  }
}

@Entry
@Component
struct BasketModifier {
  @State shopBasket: PurchaseItem[] = [];

  build() {
    Column() {
      Button('Add to basket')
        .onClick(() => {
          this.shopBasket.push(new PurchaseItem(Math.round(100 * Math.random())))
```

```
54.            })
55.          BasketViewer({ shopBasket: $shopBasket })
56.        }
57.      }
58.    }
```

上述代码示例处理步骤如下：

(1) BasketModifier 组件的 Button.onClick 向 BasketModifier shopBasket 中添加条目。

(2) @Link 装饰的 BasketViewer shopBasket 值发生变化。

(3) 状态管理框架调用@Watch 函数 BasketViewer onBasketUpdated 更新 BaketViewer TotalPurchase 的值。

(4) @Link shopBasket 的改变，新增了数组项，ForEach 组件会执行 item Builder，渲染构建新的 Item 项；@State totalPurchase 改变，对应的 Text 组件也重新渲染；重新渲染是异步发生的。

3.7 渲染控制

ArkUI 通过自定义组件的 build()函数和@builder 装饰器中的声明式 UI 描述语句构建相应的 UI。在声明式描述语句中开发者除了使用系统组件外，还可以使用渲染控制语句来辅助 UI 的构建，这些渲染控制语句包括控制组件是否显示的条件渲染语句、基于数组数据快速生成组件的循环渲染语句以及针对大数据量场景的数据懒加载语句。

3.7.1 if/else：条件渲染

ArkTS 为开发者提供了渲染控制的能力。条件渲染可根据应用的不同状态，使用 if、else 和 else if 渲染对应状态下的 UI 内容。

在 if 和 else if 语句后可以使用状态变量，当 if、else if 后跟随的状态判断中使用的状态变量值变化时，条件渲染语句会进行更新。

以下代码示例通过 if…else…语句与拥有@State 装饰变量的子组件配合使用展示条件渲染。

```
1.    //entry/src/main/ets/pages/if_else.ets
2.    @Component
3.    struct CounterView {
4.      @State counter: number = 0;
5.      label: string = 'unknown';
6.
7.      build() {
8.        Row() {
9.          Text(`${this.label}`)
10.         Button(`counter ${this.counter} + 1`)
11.           .onClick(() => {
12.             this.counter += 1;
13.           })
14.       }
15.     }
16.   }
17.
18.   @Entry
19.   @Component
```

```
struct MainView {
  @State toggle: boolean = true;

  build() {
    Column() {
      if (this.toggle) {
        CounterView({ label: 'CounterView # positive' })
      } else {
        CounterView({ label: 'CounterView # negative' })

      }
      Button(`toggle ${this.toggle}`)
        .onClick(() => {
          this.toggle = !this.toggle;
        })
    }
  }
}
```

在上述代码中，CounterView(label 为 'CounterView # positive')子组件在初次渲染时创建。此子组件携带名为 counter 的状态变量。当修改 CounterView.counter 状态变量时，CounterView(label 为 'CounterView # positive')子组件重新渲染时并保留状态变量值。当 MainView.toggle 状态变量的值更改为 false 时，MainView 父组件内的 if 语句将更新，随后将删除 CounterView(label 为 'CounterView # positive')子组件。与此同时，将创建新的 CounterView(label 为 'CounterView # negative')实例，而它自己的 counter 状态变量设置为初始值 0。

3.7.2　ForEach：循环渲染

在实际开发中，开发者通常会遇到需要将重复的组件展示多次的情况，例如购物软件中一件件商品、外卖软件中的食品，开发者往往不能事先知道需要渲染的组件数量，ArkTS 则为开发者提供了 ForEach 用于基于数组类型数据执行循环渲染。

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

参数 arr 必须是数组，允许设置为空数组，空数组场景下将不会创建子组件。同时允许设置返回值为数组类型的函数，例如 arr.slice(1,3)，设置的函数不得改变包括数组本身在内的任何状态变量，如 Array.splice、Array.sort 或 Array.reverse 这些改变原数组的函数。参数 itemGenerator 为生成子组件的 lambda 函数，为数组中的每一个数据项创建一个或多个子组件，单个子组件或子组件列表必须包括在大括号“{…}”中。参数 keyGenerator 不是必需的，用于给数组中的每一个数据项生成唯一且固定的键值。但是，为了使开发框架能够更好地识别数组更改，提高性能，建议开发者进行提供。如将数组反向时，如果没有提供键值生成器，则 ForEach 中的所有节点都将重建。

以下代码示例简单展示 ForEach 用法。

```
@Entry
@Component
```

```
3.    struct MyComponent {
4.      @State arr: number[] = [10, 20, 30];
5.
6.      build() {
7.        Column({ space: 5 }) {
8.          Button('Reverse Array')
9.            .onClick(() => {
10.             this.arr.reverse();
11.           })
12.         ForEach(this.arr, (item: number) => {
13.           Text(`item value: ${item}`).fontSize(18)
14.           Divider().strokeWidth(2)//分割线
15.         }, (item: number) => item.toString())
16.       }
17.     }
18.   }
```

通过以上代码示例，相信读者已经简单了解了循环渲染的简单用法，接下来的代码示例将介绍 ForEach 循环渲染更复杂的用法。

```
1.    //entry/src/main/ets/pages/ForEach.ets
2.    let NextID: number = 0;
3.
4.    @Observed
5.    class MyCounter {
6.      public id: number;
7.      public c: number;
8.
9.    constructor(c: number) {
10.       this.id = NextID++;
11.       this.c = c;
12.     }
13.   }
14.
15.   @Component
16.   struct CounterViewHere {
17.     @ObjectLink counter: MyCounter;
18.     label: string = 'CounterView';
19.
20.     build() {
21.       Button(`CounterView [ ${this.label}] this.counter.c = ${this.counter.c} + 1`)
22.         .width(200).height(50)
23.         .onClick(() => {
24.           this.counter.c += 1;
25.         })
26.     }
27.   }
28.
29.   @Entry
30.   @Component
31.   struct MainViewHere {
32.     @State firstIndex: number = 0;
33.     @State counters: Array<MyCounter> = [new MyCounter(0), new MyCounter(0), new
      MyCounter(0),
34.       new MyCounter(0), new MyCounter(0)];
35.
36.     build() {
37.       Column() {
```

```
38.          ForEach(this.counters.slice(this.firstIndex, this.firstIndex + 3),
39.            (item) => {
40.              CounterViewHere({ label: `Counter item # ${item.id}`, counter: item })
41.            },
42.            (item) => item.id.toString()
43.          )
44.          Button(`Counters: shift up`)
45.            .width(200).height(50)
46.            .onClick(() => {
47.              this.firstIndex = Math.min(this.firstIndex + 1, this.counters.length - 3);
48.            })
49.          Button(`counters: shift down`)
50.            .width(200).height(50)
51.            .onClick(() => {
52.              this.firstIndex = Math.max(0, this.firstIndex - 1);
53.            })
54.        }
55.      }
56.    }
```

当增加 firstIndex 的值时，Mainview 内的 ForEach 将更新，并删除与项 ID firstIndex-1 关联的 CounterView 子组件。对于 ID 为 firstindex＋3 的数组项，将创建新的 CounterView 子组件实例。由于 CounterView 子组件的状态变量 counter 值由父组件 Mainview 维护，故重建 CounterView 子组件实例不会重建状态变量 counter 值。

3.7.3　LazyForEach：数据懒加载

LazyForEach 从提供的数据源中按需迭代数据，并在每次迭代过程中创建相应的组件。当 LazyForEach 在滚动容器中使用时，框架会根据滚动容器可视区域按需创建组件，当组件划出可视区域外时，框架会进行组件销毁回收以降低内存占用。

LazyForEach 用法与 ForEach 相似，但提供了更强大的功能，所以也需要开发者编写更复杂的代码。

```
1.    LazyForEach(
2.        dataSource: IDataSource,                    // 需要进行数据迭代的数据源
3.        itemGenerator: (item: any) => void,         // 子组件生成函数
4.        keyGenerator? : (item: any) => string       // (可选)键值生成函数
5.    ): void
6.    interface IDataSource {                         //数据源需要实现的接口
7.        totalCount(): number;                       //数据源的总数量
8.        getData(index: number): any;                // 返回单个数据
9.        registerDataChangeListener(listener: DataChangeListener): void; // 注册监听者用于
                                                                          // 观察数据变化
10.       unregisterDataChangeListener(listener: DataChangeListener): void; // 取消注册监听者
11.   }
12.   interface DataChangeListener {
13.       onDataReloaded(): void;                     // 数据重新加载时调用
14.       onDataAdd(index: number): void;             // 数据添加时被调用
15.       onDataMove(from: number, to: number): void; // 数据被移动时调用
16.       onDataDelete(index: number): void;          // 数据被删除时调用
17.       onDataChange(index: number): void;          // 数据被改变时调用
18.   }
```

读者看到以上代码将会陷入困惑之中，空洞地介绍以上接口对读者毫无帮助，接下来将

以一段代码帮助读者学习 LazyForEach 功能的用法。

```
//entry/src/main/ets/pages/LazyForEach.ets
// Basic implementation of IDataSource to handle data listener
class BasicDataSource implements IDataSource {
  private listeners: DataChangeListener[] = [];

  public totalCount(): number {
    return 0;
  }

public getData(index: number): any {
  return undefined;
}

registerDataChangeListener(listener: DataChangeListener): void {
  if (this.listeners.indexOf(listener) < 0) {
    console.info('add listener');
    this.listeners.push(listener);
  }
}

unregisterDataChangeListener(listener: DataChangeListener): void {
  const pos = this.listeners.indexOf(listener);
  if (pos >= 0) {
    console.info('remove listener');
    this.listeners.splice(pos, 1);
  }
}

notifyDataReload(): void {
  this.listeners.forEach(listener => {
    listener.onDataReloaded();
  })
}

notifyDataAdd(index: number): void {
  this.listeners.forEach(listener => {
    listener.onDataAdd(index);
  })
}

notifyDataChange(index: number): void {
  this.listeners.forEach(listener => {
    listener.onDataChange(index);
  })
}

notifyDataDelete(index: number): void {
  this.listeners.forEach(listener => {
    listener.onDataDelete(index);
  })
}

notifyDataMove(from: number, to: number): void {
  this.listeners.forEach(listener => {
    listener.onDataMove(from, to);
  })
```

```
57.     }
58.   }
59.
60.   class MyDataSource extends BasicDataSource {
61.     private dataArray: string[] = ['/path/image0', '/path/image1', '/path/image2',
        '/path/image3'];
62.
63.     public totalCount(): number {
64.       return this.dataArray.length;
65.     }
66.
67.     public getData(index: number): any {
68.       return this.dataArray[index];
69.     }
70.
71.     public addData(index: number, data: string): void {
72.       this.dataArray.splice(index, 0, data);
73.       this.notifyDataAdd(index);
74.     }
75.
76.     public pushData(data: string): void {
77.       this.dataArray.push(data);
78.       this.notifyDataAdd(this.dataArray.length - 1);
79.     }
80.   }
81.
82.   @Entry
83.   @Component
84.   struct MyComponent {
85.     private data: MyDataSource = new MyDataSource();
86.
87.     build() {
88.       List({ space: 3 }) {
89.         LazyForEach(this.data, (item: string) => {
90.           ListItem() {
91.             Row() {
92.               Image(item).width('30%').height(50)
93.               Text(item).fontSize(20).margin({ left: 10 })
94.             }.margin({ left: 10, right: 10 })
95.           }
96.           .onClick(() => {
97.             this.data.pushData('/path/image' + this.data.totalCount());
98.           })
99.         }, item => item)
100.      }
101.    }
102.  }
```

在上述代码示例中，表示了一个基本的 IDataSource 接口及其派生类 MyDataSource 的实现。还包括一个名为 MyComponent 的组件，该组件使用 MyDataSource 来显示一个包含图像和相应文本的列表。

BasicDataSource 类实现了 IDataSource 接口，并提供了基本的注册和通知数据变化监听器的功能。它维护一个 DataChangeListener 对象的数组，并提供了添加、删除和通知监听器的方法，当数据发生变化时会触发通知。

MyDataSource 类扩展了 BasicDataSource，并添加了特定于示例的附加功能。它维护

了一个名为 dataArray 的内部数组，用于存储图像路径的列表。该类重写了 IDataSource 接口中的 totalCount()和 getData()方法，分别提供数据的总数和根据给定索引从 dataArray 中获取数据。它还添加了 addData()和 pushData()方法，用于向 dataArray 添加新数据并在数据添加时通知监听器。

MyComponent 结构表示一个使用 MyDataSource 来显示图像和相应文本列表的组件。它使用 List 组件显示列表，并使用 LazyForEach 组件对 MyDataSource 的 data 数组进行迭代。对于 data 数组中的每个项，它渲染一个包含 Image 和 Text 组件的 ListItem 组件。Image 组件显示与项对应的图像，Text 组件显示项的文本。此外，每个 ListItem 组件附加了一个 onClick 事件，当单击时调用 MyDataSource 的 pushData()方法向列表中添加新图像。

3.8 本章小结

本章花费了大量篇幅介绍 TS 和 ArkTS 的基本知识，当读者阅读到这里，可能仍然对鸿蒙开发充满了困惑，但是不要担心，到此为止本书意在使读者快速了解鸿蒙开发的构成和基本开发组件，在接下来的章节中将更详细地介绍鸿蒙开发。

3.9 课后习题

1. ArkTS 与 JavaScript 和 TypeScript 的主要区别在于哪一方面？（　　）

A. 语法　　B. 编译方式

C. 执行速度　　D. 针对鸿蒙系统的优化

2. 以下哪种数据类型不是 TypeScript 基础数据类型之一？（　　）

A. number　　B. string　　C. boolean　　D. character

3. 在 ArkTS 中，哪一个装饰器用于自定义构建函数？（　　）

A. @State　　B. @Prop　　C. @Builder　　D. @Watch

4. 哪一种存储方式用于在页面级别存储 UI 状态？（　　）

A. LocalStorage　　B. AppStorage

C. PersistentStorage　　D. SessionStorage

5. TypeScript 中的变量声明方式有三种：_____、_____和_____。

6. 在 ArkTS 中，_____装饰器用于定义一个状态变量，当状态变量更改时，UI 会自动更新。

7. 使用 ArkTS 时，自定义组件的基本结构包括组件的定义、_____、_____和事件处理。

8. 在 ArkTS 中，_____装饰器用于监听状态变量的更改并执行相应的操作。

9. 解释 TypeScript 的主要数据类型，并举例说明如何声明这些类型的变量。

10. 在 ArkTS 中，如何定义和使用自定义组件？请说明自定义组件的基本结构和生命周期。

11. 解释 ArkTS 中的状态管理装饰器，并举例说明如何使用@State 和@Prop 装饰器管理组件状态。