第5章

ArkTS 语言概述

ArkTS 是当前 HarmonyOS 应用开发的主要语言。目前，HarmonyOS 应用开发的主推模型 Stage 的新版本开发不再支持 Java 和 JavaScript，因此，开发 HarmonyOS 应用的应用开发人员学习 ArkTS 开发语言是必需的。

5.1 初识 ArkTS 语言

Mozilla 创建了 JavaScript，Microsoft 创建了 TypeScript，而华为进一步推出了 ArkTS。ArkTS 是 HarmonyOS 优选的主力应用开发语言，使用.ets 作为 ArkTS 语言源码文件扩展名。ArkTS 是围绕应用开发并匹配 ArkUI 框架，在 TypeScript 的基础上做了进一步扩展，保持了 TypeScript 的基本风格，扩展了声明式 UI、状态管理等相应的能力，同时通过规范定义强化开发期静态检查和分析，提升程序执行稳定性和性能，代码量相较于传统的 JavaScript/TypeScript 语言下降了 30%，让开发人员以更简洁、更自然的方式开发跨端的高性能应用。

为了更好地了解 ArkTS，需要首先明白 ArkTS、TypeScript 和 JavaScript 之间的关系。图 5-1 展示了三者之间的关系。可以看出，TypeScript 是 JavaScript 的超集，ArkTS 则是 TypeScript 的超集。具体表现如下。

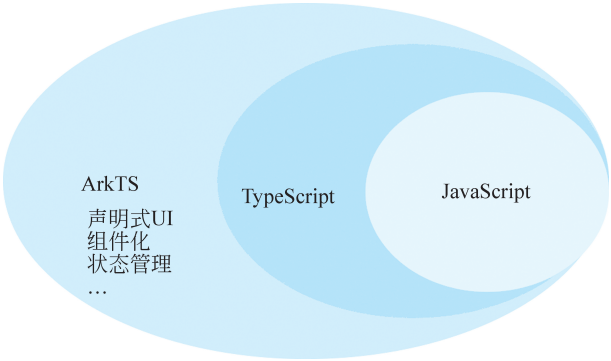

图 5-1 ArkTS、TypeScript 和 JavaScript 关系

(1) JavaScript 是一种属于网络的高级脚本语言，已经被广泛用于 Web 应用开发，常用来为网页添加各式各样的动态功能，为用户提供更流畅美观的浏览效果。

（2）TypeScript 是 JavaScript 的一个超集，它扩展了 JavaScript 的语法，通过在 JavaScript 的基础上添加静态类型定义构建而成，是一个开源的编程语言。

（3）ArkTS 兼容 TypeScript 语言，进一步规范强化静态检查和分析，将编译时所确定的类型应用到运行性能优化中，同时拓展了声明式 UI、状态管理、并发任务等能力。

ArkTS 对比标准 TypeScript，差异具体体现在以下几方面。

（1）强制使用静态类型。静态类型是 ArkTS 最重要的特性之一。如果使用静态类型，那么程序中变量的类型就是确定的。同时，由于所有类型在程序实际运行前都是已知的，编译器可以验证代码的正确性，从而减少运行时的类型检查，有助于性能提升。

（2）禁止在运行时改变对象布局。为实现最大性能，ArkTS 要求在程序执行期间不能更改对象布局。

（3）限制运算符语义。为了获得更好的性能并鼓励开发者编写更清晰的代码，ArkTS 限制了一些运算符的语义。例如，一元加法运算符只能作用于数字，不能用于其他类型的变量。

（4）不支持 Structural typing。对 Structural typing 的支持需要在语言、编译器和运行时进行大量的考虑和仔细的实现，当前 ArkTS 不支持该特性。

当前，在 UI 开发框架中，ArkTS 主要扩展了如下能力。

（1）基本语法：ArkTS 定义了声明式 UI 描述、自定义组件和动态扩展 UI 元素的能力，再配合 ArkUI 开发框架中的系统组件及其相关的事件方法、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，与 UI 相关联的数据可以在组件内使用，也可以在不同组件层级间传递，如父子组件之间、爷孙组件之间，还可以在应用全局范围内传递或跨设备传递。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。开发人员可以灵活地利用这些能力来实现数据和 UI 的联动。

（3）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 UI 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。数据懒加载从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。

当然，ArkTS 作为一种比较新的程序设计语言，还处在版本持续演进阶段，伴随着其设计定位的不断完善，结合应用开发/运行的现实需求，ArkTS 的语言体系将更加成熟，逐步提供并行和并发能力增强、系统类型增强、分布式开发范式等更多特性，ArkTS 应用的开发和运行体验将会进一步提升。

5.2 ArkTS 基础语法

5.2.1 基本知识

ArkTS 通过声明引入变量、常量、类型和函数。其中，变量以关键字 let 开头的声明引入，该变量在程序执行期间可以具有不同的值。只读常量以关键字 const 开头的声明引入，

该常量只能被赋值一次。若对常量重新赋值，会造成编译时错误。由于 ArkTS 是一种静态类型语言，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包含初始值，开发人员就可以不显式指定变量和常量的类型。以下两行定义 String 类型变量的示例代码是等效的。

```
let l1: string = 'Welcome';
let l2 = 'Welcome';
```

ArkTS 中的变量类型有 Number 类型、Boolean 类型、String 类型、Void 类型、Object 类型、Array 类型、Enum 类型、Union 类型、Aliases 类型。其中，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 类型由 true 和 false 两个逻辑值组成，通常在条件语句中使用 boolean 类型的变量。String 代表字符序列，可以使用转义字符来表示字符，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，也可以使用反向单引号(`)括起来的模板字面量。Void 类型用于指定函数没有返回值，此类型只有一个值，同样是 void。由于 Void 是引用类型，因此它可以用于泛型类型参数。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。Array，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。数组可由数组复合字面量(即用方括号括起来的零个或多个表达式的列表，其中每个表达式为数组中的一个元素)来赋值。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。

Enum 类型，又称枚举类型，是预先定义的一组命名值的值类型，其中，命名值又称为枚举常量。使用枚举常量时必须以枚举类型名称为前缀，常量表达式可以用于显式设置枚举常量的值。

```
//定义枚举变量 ColorSet1,有 Red、Green、Blue 三个值
enum ColorSet1 { Red, Green, Blue }
//定义枚举变量 ColorSet2,有 Red、Green、Blue 三个值,并显式设置常量的十六进制数值
enum ColorSet2 { White = 0xFF, Grey = 0x7F, Black = 0x00 }

//使用枚举变量
let c: ColorSet = ColorSet1.Red;
```

Union 类型，即联合类型，是由多个类型组合成的引用类型，联合类型包含变量可能的所有类型。可以用不同的机制获取联合类型中特定类型的值。

```
//定义 Cat 类,有 sleep 和 meow 两个方法
class Cat { sleep () {}; meow () {} }
//定义 Dog 类,有 sleep 和 bark 两个方法
class Dog { sleep () {}; bark () {} }
//定义 Frog 类,有 sleep 和 leap 两个方法
class Frog { sleep () {}; leap () {} }

//Cat、Dog、Frog 是一些类型(类或接口)
type Animal = Cat | Dog | Frog | number
```

```
//可以将类型为联合类型的变量赋值为任何组成类型的有效值
let animal: Animal = new Cat();
animal = 42;
animal = new Frog();

if (animal instanceof Frog) {
  let frog: Frog = animal as Frog;                    //animal 在这里是 Frog 类型
  animal.leap();                                      //frog 跳了第一次
  frog.leap();                                        //frog 跳了第二次
  //结果:frog 跳了两次
}

animal.sleep ();                                      //任何动物都可以睡觉
```

Aliases 类型为匿名类型(数组、函数、对象字面量或联合类型)提供名称,或为已有类型提供替代名称。

```
type Matrix = number[][];
type Handler = (s: string, no: number) => string;
type Predicate <T> = (x: T) => Boolean;
type NullableObject = Object | null;
```

ArkTS 中的运算符有赋值运算符、比较运算符、算术运算符、位运算符和逻辑运算符。赋值运算符=的使用方法为 x=y,意思是将 y 的值赋给 x。比较运算符有==、!=、>、>=、<和<=,其中,==表示如果两个操作数相等,则返回 true;!=表示如果两个操作数不相等,则返回 true;>表示如果左操作数大于右操作数,则返回 true;>=表示如果左操作数大于或等于右操作数,则返回 true;<表示如果左操作数小于右操作数,则返回 true;<=表示如果左操作数小于或等于右操作数,则返回 true。算术运算符包括一元运算符和二元运算符。一元运算符包括-(负号)、+(正号)、--(自减 1)、++(自增 1);二元运算符包括+(加号)、-(减号)、*(乘号)、/(除号)和%(取余号,除法后的余数)。位运算符包括 &、|、^、~、<<、>>和>>>。& 表示按位与,如果两个操作数的对应位都为 1,则将这个位设置为 1,否则设置为 0;| 表示按位或,如果两个操作数的相应位中至少有一个为 1,则将这个位设置为 1,否则设置为 0;^表示按位异或,如果两个操作数的对应位不同,则将这个位设置为 1,否则设置为 0;~表示按位非,反转操作数的位;<<表示左移,将运算符左侧的二进制表示向左移运算符右侧的数值位;>>表示算术右移,将运算符左侧的二进制表示向右移运算符右侧的数值位,带符号扩展;>>>表示逻辑右移,将运算符左侧的二进制表示向右移运算符右侧的数值位,左边补 0。逻辑运算符包括 &&(逻辑与)、||(逻辑或)和!(逻辑非)。

ArkTS 中的语句有 if 语句、switch 语句、for 语句、for…of 语句、while 语句、do…while 语句、break 语句、continue 语句、throw 语句和 try 语句。

1. if 语句

if 语句用于需要根据逻辑条件执行不同代码块的场景。当逻辑条件为真时,执行对应的一组语句,否则执行另一组语句(如果有的话)。else 部分也可能包含 if 语句。

2. switch 语句

switch 语句用于根据 switch 表达式值匹配的结果执行不同代码块的场景。当匹配成功时，执行对应的代码块，否则继续向下进行表达式值的匹配。

switch 语句的示例如下。

```
switch (expression) {
  case label1:                                  //如果 label1 匹配，则执行
    //...
    //语句 1
    //...
    break;                                      //可省略
  case label2:
  case label3:                                  //如果 label2 或 label3 匹配，则执行
    //...
    //语句 2、3
    //...
    break;                                      //可省略
  default:
    //默认语句
}
```

需要注意的是，switch 语句中 expression（即 switch 表达式值）的类型必须是 number、enum 或 string，而 label（即紧跟 case 关键字后的值）必须是常量表达式或枚举常量值。如果 switch 语句中 expression 的值等于某个 label 的值，则执行相应的代码块。如果没有任何一个 label 值与 expression 值相匹配，并且 switch 具有 default 子句，那么程序会执行 default 子句对应的代码块。break 语句（可选的）允许跳出 switch 语句并继续执行 switch 语句之后的语句。如果没有 break 语句，则在执行完成当前 label 对应的代码块后，程序继续执行 switch 中的下一个 label 对应的代码块。

3. for 语句

for 语句会被重复执行，直到循环退出语句值为 false。

for 语句的示例如下。

```
for ([init]; [condition]; [update]) {
  statements
}
```

for 语句的执行流程为：首先执行 init 表达式（如有）。init 表达式通常初始化一个或多个循环计数器。然后计算 condition。如果 condition 的计算结果值为 true（或者没有 condition 语句），则程序执行 statements（即循环主体的语句）。然后执行 update 表达式（如有）。最后再重新计算 condition，并循环执行。如果 condition 的计算结果值为 false，则 for 循环终止。

4. for…of 语句

使用 for…of 语句可以遍历数组或字符串。

for…of 语句的示例如下。

```
for (forVar of expression) {
  statements
}
```

在以下代码示例中，通过 for…of 语句实现了对字符串“a for…of example”中所有字符的遍历，每次从字符串中取出一个字符后将其赋值给 ca 变量，然后执行 statements(即循环主体的语句)。

```
for (let ca of 'a for-of example') {
  /* process ca */
statements
}
```

5. while 语句

while 语句应用于当满足某一条件时进行循环的情况，多用于解决循环次数事先不确定的问题。

while 语句的示例如下。

```
while (condition) {
  statements
}
```

只要条件表达式 condition 的值为 true，while 语句就会执行 statements 语句。此外，condition 必须是逻辑表达式。

6. do…while 语句

do…while 语句与 while 语句的格式不同。

do…while 语句的示例如下。

```
do {
  statements
} while (condition)
```

do…while 语句在条件表达式 condition 的值为 false 之前，statements 语句会重复执行。同时，与 while 语句不同的是，do…while 语句在判断条件表达式 condition 的值之前先执行一次 statements。此外，condition 必须是逻辑表达式。

7. break 语句

使用 break 语句可以终止循环语句，跳出循环体；break 语句也可以终止 switch 语句，继续执行 switch 语句后的代码。但是，如果 break 语句后带有标识符，则将控制流转移到该标识符所包含的语句块之外。

break 语句的示例如下。

```
let a = 0
ouer:
while (true) {
```

```
  switch (a) {
    case 0:
      //statements
      break ouer                          //中断 while 语句,接下来跳转出 ouer 标识的语句块
  }
}
```

8. continue 语句

continue 语句会停止当前循环迭代的执行,并将控制传递给下一个迭代。

9. throw 语句和 try 语句

throw 语句用于抛出异常或错误,而 try 语句用于捕获和处理异常或错误。

throw 语句和 try 语句联合使用的示例如下。

```
function processData(s: string) {
  let error: Error | null = null;

  try {
    console.log('Data processed: ' + s);
    //...
    //可能发生异常的语句
    //...
  } catch (e) {
    error = e as Error;
    //...
    //异常处理
    //...
    throw new Error('this error')      //抛出错误提示
  } finally {
    if (error != null) {
      console.log(`Error caught: input='${s}', message='${error.message}'`)
    }
  }
}
```

在以上代码示例中,try 出现在方法体中,它自身是一个代码块,表示尝试执行代码块的语句。如果在执行过程中 try 代码块中有某条语句抛出异常,那么代码块后面的语句将不被执行。catch 出现在 try 代码块的后面,自身也是一个代码块,用于捕获异常 try 代码块中可能抛出的异常。catch 关键字后面紧接着它能捕获的异常类型,所有异常类型的子类异常也能被捕获。而 throw 出现在方法体(上例中 throw 位于 catch 方法体)中,用于抛出异常。以上代码示例中,try 代码块在执行过程中遇到异常情况时,将异常信息封装为异常对象,然后通过 throw 语句抛出异常信息(上例中抛出了“this error”)。如果 finally 存在的话(上例中存在),任何执行 try 或者 catch 中的 return 语句之前,都会先执行 finally 语句。如果 finally 中有 return 语句,那么程序就返回并结束了,所以 finally 中的 return 语句是一定会被执行的。

5.2.2 函数

函数声明引入一个函数，包含其名称、参数列表、返回类型和函数体。一个包含两个 string 类型的参数且返回类型为 string 的函数示例如下。

```
function comb(a: string, b: string): string {
  let c: string = `${a} ${b}`;
  return c;
}
```

在函数声明中，需要为每个参数标记类型。如果参数为可选参数，那么允许在调用函数时省略该参数。可选参数的格式有以下两种。

第一种为 name?: Type。一个使用可选参数的函数示例如下。

```
function welcome(name?: string) {
  if (name == undefined) {
    console.log('Welcome!');
  } else {
    console.log(`Welcome ${name}!`);
  }
}
```

第二种为设置的参数默认值。如果在函数调用中这个参数被省略了，则会使用此参数的默认值作为实参。一个使用参数默认值的函数示例如下。

```
function mul(num: number, coeff: number = 2): number {
  return num * coeff;
}

mul(2);                                    //第 2 个参数使用默认值 2，返回 2 * 2
mul(2, 3);                                 //第 2 个参数使用传参 3，返回 2 * 3
```

函数的最后一个参数可以是 rest 参数。使用 rest 参数时，允许函数或方法接收任意数量的实参。一个使用 rest 参数的函数示例如下。

```
function mul(…nums: number[]): number {
  let res = 1;
  for (let num of nums)
    res *= num;
  return res;
}

mul()                                      //传递 0 个参数，返回 0
mul(1, 2, 3)                               //传递 3 个参数，返回 6
```

如果可以从函数体内推断出函数返回类型，那么可以在函数声明中将返回类型省略标注。此外，如果函数不需要返回值，那么其返回类型可以显式指定为 void 或直接省略标注。

这类函数在函数体内也不需要返回语句。函数中定义的变量和其他实例仅可以在函数内部访问，不能从外部访问。如果函数中定义的变量在外部作用域中已有实例同名，则函数内的局部变量定义将覆盖外部定义。

函数类型通常用于定义回调。一个将函数类型应用于回调的示例如下。

```
type trigFunc = (x: number) => number;              //定义一个函数类型

function do_action(f: trigFunc) {
   f(3.141592653589);                               //调用函数
}

do_action(Math.sin);                                //将函数作为参数传入
```

函数可以定义为箭头函数(Lambda 函数)，箭头函数的返回类型可以省略，此时的返回类型通过函数体推断。表达式可以指定为箭头函数，使表达更简短。以下三种表达方式是等价的。

```
//表达方式 1
let sum1 = (x: number, y: number): number => {
  return x + y;
}
//表达方式 2
let sum1 = (x: number, y: number) => { return x + y; }
//表达方式 3
let sum1 = (x: number, y: number) => x + y;
```

箭头函数通常在另一个函数中定义。作为内部函数，它可以访问外部函数中定义的所有变量和函数。为了捕获上下文，内部函数将其环境组合成闭包，以允许内部函数在自身环境之外的访问。一个将箭头函数作为另一个函数 f 的内部函数的示例如下。

```
function f(): () => number {
  let cnt = 0;
  return (): number => { cnt++; return cnt; }      //内部函数,形成闭包,捕获 cnt 变量
}

let z = f();                                        //定义变量,关联函数 f
z();                                                //第 1 次调用,返回:1
z();                                                //第 2 次调用,返回:2
```

ArkTS 也允许函数重载。通过编写重载函数，指定函数的不同调用方式。具体的实现方法为：为同一个函数写入多个同名但签名不同的函数头，函数实现紧随其后。但是，不允许重载函数同时拥有相同的名字以及参数列表，否则将会编译报错。以下示例给出了名为 welcome 的两个不同的重载函数。

```
function welcome(): void;                           /* 第一个函数定义 */
function welcome (a: string): void;                 /* 第二个函数定义 */
function welcome (a?: string): void {               /* 两个重载函数的实现 */
```

```
  console.log(a);
}

welcome ();                                          //使用第一个定义
welcome ('Welcome');                                 //使用第二个定义
```

5.2.3 类

类声明引入一个新类型，并定义其字段、方法和构造函数。

【例 5-1】 Person 类。

```
class Person {
//第一个字段
  firstName: string = ''
//第二个字段
  lastName: string = ''
//第三个字段
static numberOfPersons = 0

//构造函数
  constructor (f: string, l: string) {
    this.firstName = f;
    this.lastName = l;
    Person.numberOfPersons++;
  }

//名称为 fullName 的方法
  fullName(): string {
    return this.firstName + ' ' + this.lastName;
  }
}
```

在以上示例中，给出了一个 Person 类，该类拥有三个字段，分别为 firstName、lastName 和 numberOfPersons，拥有一个构造函数和一个名称为 fullName 的方法。

完成类的定义后，有以下两种创建实例的方法。

方法一：使用关键字 new 创建实例。示例如下。

```
let p = new Person('Michael', 'Jordan');
console.log(p.fullName());
```

方法二：使用对象字面量创建实例。示例如下。

```
let p: Person = {firstName: 'Michael', lastName: 'Jordan'};
console.log(p.fullName());
```

在类中，字段是直接声明的某种类型的变量。类可以具有实例字段或者静态字段。其中，实例字段存在于类的每个实例上。每个实例都有自己的实例字段集合。要访问实例字