第3章 ArkTS基础

CHAPTER 3

【学习目标】

- 了解 TypeScript 和 ArkTS 语言的关系和特点
- 认识 ArkTS 语言支持的数据类型
- 会用 ArkTS 语言的控制结构、函数
- 理解 ArkTS 语言的面向对象特征并能够正确使用

本章介绍 ArkTS 基础。ArkTS 是由华为公司推出的基于 TypeScript 的一种编程语言。已经熟悉 TypeScript 的读者可以快速阅读或跳过本章内容。另外，本章对 ArkTS 的介绍是简明扼要的，是以在 HarmonyOS 应用开发中可以使用为目的，如果读者需要进一步掌握 ArkTS，则需要学习其他专门针对该编程语言的资料。

8min

3.1 TypeScript 和 ArkTS 简介

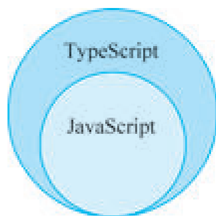
TypeScript 简称 TS，是由微软公司开发的自由和开源的编程语言，它是 JavaScript 的一个超集，包含了 JavaScript 的全部特性。JavaScript 简称 JS，1995 年问世，是目前最广泛使用的跨平台语言之一。虽然 JS 编写的程序的大小、范围和复杂性呈指数级增长，但 JS 语言表达存在严重不足，同时编码过程中的常见拼写、类型等错误等都不能预先检查。TypeScript 的最初目标是成为 JavaScript 程序的静态类型检查器，但随着其发展，TypeScript 目标更多是为了开发大型应用，同时其代码可以编译生成纯 JavaScript 代码，并可运行在任何浏览器上。

TypeScript 支持 ES6 标准，ES6 的全称是 ECMAScript 6.0。由于 JavaScript 是被 Oracle 公司注册的商标，因此 JavaScript 的另一个正式名称是 ECMAScript。1996 年 11 月，JavaScript 的创造者网景公司将其提交给国际化标准组织欧洲计算机制造商协会（简称 ECMA），希望其能够成为国际标准，后来便有了 ECMAScript。2009 年 12 月，ECMAScript 5.0 版正式发布，简称 ES5。2015 年 6 月，ES6 正式成为国际标准。图 3-1 表示了 TypeScript、JavaScript、ES5、ES6 之间的关系。

TypeScript 语言在 JavaScript 的基础上增加了很多新的特性，包括类型批注和编译时

类型检查、类型推断和类型擦除、枚举、元组、接口、泛型编程、名字空间、类、模块、Lambda函数、可选参数及默认参数等。

ArkTS 是目前鸿蒙应用开发的主要语言，它是在 TypeScript 语言的基础上发展起来的，其扩展了 TypeScript，同时也增加了一些禁止和限制规则。JavaScript、TypeScript 和 ArkTS 三者之间的关系可以用图 3-2 表示。ArkTS 和 JavaScript、TypeScript 都有交集，同时 ArkTS 也有自己独有的内容。



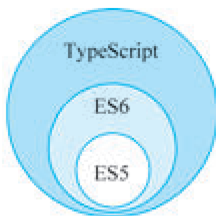


图 3-1 TypeScript、JavaScript、ES5、ES6 之间的关系

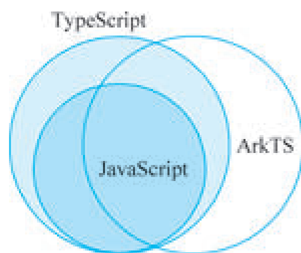


图 3-2 TypeScript、JavaScript、ArkTS 之间的关系

ArkTS 保持了 TypeScript 的基本风格，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。ArkTS 语言主要有以下特征。

（1）强制使用静态类型：静态类型是 ArkTS 最重要的特性之一。静态类型要求程序中变量的类型在编译期就是确定的，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：ArkTS 要求在程序执行期间不能更改对象布局，例如不能增加属性等，这样有助于提高性能。

（3）限制操作符的语义：为使代码编写更加清晰，同时获得更高的性能，ArkTS 限制了一些运算符的语义，例如一元＋运算符只能作用于数字，不能用于其他类型的变量。

（4）扩展了 UI 开发框架能力：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。ArkTS 提供了多维度的状态管理机制，提供了条件和循环渲染控制的能力，可以根据需要动态地构建组件。开发者可以灵活地利用这些能力来实现数据和 UI 的联动。

3.2 变量和常量

6min

3.2.1 变量

在程序中，变量表示各种可变的量，例如序号、姓名、商品价格等。ArkTS 语言声明变量的一般格式如下：

```
let 变量名 : 变量类型 = 变量初始值 ;
```

这里，let 是声明变量的关键字，是固定用法。变量名是程序员为变量起的名字，在后续

的程序中可以通过变量名访问变量,变量名是一种自定义的标识符。

在程序代码中,程序员定义的变量名、函数名、类名等都属于自定义标识符,在 ArkTS 中,标识符的命名必须遵守以下规则:

(1) 标识符由大写字母、小写字母、数字、下画线和 $ 符号组成。

(2) 标识符的首字符必须是大写字母或小写字母或下画线或 $ 符号。

(3) 自定义标识符不能是保留字,保留字可以理解成是由语言本身已经定义好的具有固定定义的系统标识符。

(4) 标识符是区分大小写的,例如 name、Name 是不同的标识符。

变量类型是在变量定义时确定的,其决定了变量可以存储什么样的值及其范围,变量类型定义后将不能再改变,使用时也必须注意其类型,示例代码如下:

```
let  isLogin : boolean = false ;
isLogin = true ;                  //正确
isLogin = 1 ;                     //错误,不能将数值类型赋值给 boolean 类型变量
```

声明变量时,既可以赋初始值,也可以不赋初始值,示例代码如下:

```
let  message : string = "Hello World";
let  count = 100 ;               //可以推定 count 为 number 类型
let  userName : string ;         //未赋初始值,后续使用前要确保其被赋值
```

声明变量时,如果不提供初始值,则必须明确给出变量的类型。在提供初始值时,变量声明中的类型及类型前的冒号(:)可以省略,ArkTS 可以根据所赋值的类型自动推定变量的类型。

注意,ArkTS 中不再支持使用 var 声明变量,这一点和 TS 不同。

3.2.2 常量

常量是程序运行期间其值不变的量,常量可以分为字面常量和名称常量。

字面常量(如 3.14、"Hello World"、true 等)在程序中表示固定内容和含义,字面常量一般简称字面量。

名称常量可以理解成声明的不可变其值的变量,一般简称常量,声明名称常量使用 const 关键字,基本格式如下:

```
const  常量名 = 值 ;
```

声明常量时,必须进行初始化赋值,常量赋值后不能再对其进行写操作,示例代码如下:

```
const  E = 2.718 ;
const  TITLE = "主页面" ;
const  MAX : number = 32765 ;
const  TRUE = true ;
```

```
const  NUM ;                    //错误,常量必须初始化赋值
count let MSG = "ok"            //错误,常量声明不能使用 let 关键字
```

注意,声明常量时,常量名前没有 let 关键字,常量名也是标识符,其命名应该符合标识符命名规则,一般建议常量名用大写字母组合。

3.3 基本类型和运算符

18min

3.3.1 数据类型

ArkTS 是强类型语言,要求所有的量都要有明确的类型,ArkTS 语言支持的类型包括数值类型、字符串类型、布尔类型、数组类型、元组类型、枚举类型、void 类型、null 类型、undefined 类型、never 类型。

(1) 数值类型:数值类型用于表示数值,其值为 64 位浮点值,可以表示整数和浮点数,可以采用二进制、八进制、十进制和十六进制。数值类型对应的关键字是 number。下面是数值类型变量的示例,代码如下:

```
let x: number = 100               //本质不是整数,是浮点数
let y: number = 3.14              //浮点数
let a: number = 0b111             //二进制
let b: number = 0o76              //八进制
let c: number = 60                //十进制
let d: number = 0xff0000          //十六进制
```

(2) 字符串类型:若干字符组成的串,既可以使用单引号('),也可以使用双引号(")引起来的若干字符,单引号和双引号要成对出现,如果字符串中间需要有引号,则可以使用转义字符。字符串类型对应的关键字是 string。下面是字符串类型变量的示例,代码如下:

```
let s1: string = '张三'              //单引号
let s2: string = "李四"              //双引号
let s3: string = "I'm Wang Wu"       //双引号内含有单引号可以不转义
let s4: string = "I\'m Zhao Liu"     //双引号内含有单引号也可以转义
let s5: string = 'I\'m Sun Qian'     //单引号内含有单引号必须转义
```

当字符串中需要出现特殊字符时,需要用转义字符,常用的转义字符见表 3-1。

表 3-1 常用的转义字符

代码	输出	代码	输出
\'	单引号	\r	回车符
\"	双引号	\t	制表符
\\	反斜杠	\b	退格
\n	换行符	\f	换页符

另外,字符串还可以使用反引号(`)来定义多行文本和内嵌表达式,内嵌表达式采用${变量}形式表示。下面是反引号的示例,代码如下:

```
let name: string = "ZhangSan"
let age: number = 18
let msg: string = `基本信息,姓名: ${ name } 年龄: ${ age }`
console.log(
      `<div>
          <span>${msg}</span>
      </div>`
)
```

(3) 布尔类型:表示逻辑值,对应的关键字是 boolean。布尔类型只有两个值,分别为 true 和 false,例如定义布尔变量的代码如下:

```
let r: boolean = true
```

(4) 数组类型:若干相同的类型组成的一组数据,在一种类型后面加上方括号([ ])即表示该类型对应的数组类型。另外,数组还可以使用泛型,例如定义数组的代码如下:

```
let arr1: number[] = [1,2,3,4,5]
let arr2: Array<number> = [1,2,3,4,5,6]
```

(5) 元组类型:元组可以理解成已知元素数量和类型的特殊数组,和数组不同的是,元组中各元素的类型不要求必须相同。元组类型使用的示例代码如下:

```
let t: [string,number]                      //t 为二元元组
t = [ "price",10.6 ]                        //赋值
console.log( t[0] );                        //输出 price
console.log( t[1].toString() );             //输出 10.6
t = [ 10.6,"price" ]                        //错误,类型不对应
```

(6) 枚举类型:枚举类型用于定义若干数值集合,使用的关键字是 enum。枚举类型使用的示例代码如下:

```
enum Color { Red, Green, Blue }             //默认值分别为 0,1,2
let c: Color = Color.Blue
console.log( c.toString() );                //输出 2
```

(7) void 类型:即函数空类型,用于表示函数返回值的类型,当函数不需要有返回值时,可以将其返回类型表示为该类型。函数空类型使用的示例代码如下:

```
function printinfo(): void {
    console.log("This is something");
}
```

(8) null 类型：也称为空类型，表示对象值缺失，null 表示空，即什么都没有，null 也是一个特殊值，表示一个空对象引用。当使用 typeof 判断检测 null 时，其返回的是 object。

(9) undefined 类型：即未定义类型，表示未定义，当一个变量没有设置值时就为 undefined。

(10) never 类型：即无果类型，表示从不会出现的值，声明为 never 类型的变量只能被 never 类型所赋值，never 在函数中通常表现为抛出异常或无法执行到终点，例如无限循环，示例代码如下：

```
let a: never
a = 100                                      //编译错误，数值不能赋值给 never 类型
a = (() =>{ throw new Error('error')})()     //语法正确，never 类型赋值
a = loop()
function loop(): never {                     //函数返回为 never 类型，可以理解成永远不返回
    while (true) {}
}
```

变量在使用的过程中一般遵循“类型一致性”和“先赋值后使用”的基本原则，示例代码如下：

```
let s1:string = "good"                       //定义变量并初始化
let s2:string                                //未初始化，使用前需要赋值
console.log( s2 )                            //错误，因为 s2 未赋值
let x: number = 100
console.log( typeof(x) )                     //输出 number
x = s1                                       //错误，类型不一致
```

在 ArkTS 中，可以通过管道符号(|)将变量定义成多种类型，示例代码如下：

```
let val:string | number                      //val 可以是 string 或 number 类型
val = 6
val = "some"
console.log("val = " + val)                  //最后一次赋值的结果
```

总体来讲，ArkTS 语言支持的数据类型还是比较丰富的。除了支持基本的类型外，还支持自定义类型，例如类等。

注意，ArkTS 不支持 Any 类型，这一点和 TypeScript 不同。

3.3.2 运算符

ArkTS 运算符包括算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、三元条件运算符、类型运算符、其他运算符。

(1) 算术运算符包括正(+)、负号(-)、自增(++)、自减(--)、加(+)、减(-)、乘(*)、除(/)、求余(%)、乘方(**)。

(2) 关系运算符包括大于(>)、小于(<)、大于或等于(>=)、小于或等于(<=)、等于

(==)、不等于(!=)。另外还有强等于(===),用于判断值和类型是否同时相同;强不等于(!==),表示在值和类型都不相同时运算结果为真。

(3) 逻辑运算符包括逻辑与(&&)、逻辑或(||)、逻辑非(!)。

(4) 位运算符包括位与(&)、位或(|)、取反(~)、异或(^)、左移(<<)、右移(>>)、无符号右移(>>>)。

(5) 赋值运算符包括赋值(=)、复合加赋值(+=)、复合减赋值(-=)、复合乘赋值(*=)、复合除赋值(/=)、复合求余赋值(%=)、复合乘方赋值(**=)。

(6) 三元条件运算符只有问号冒号运算符(?:)。

(7) 类型运算符包括 typeof 和 instanceof。typeof 是一元运算符,返回的是操作数的数据类型。instanceof 运算符用于判断对象是否为指定的类型的实例。

(8) 除了上述的运算符外,还有一些其他运算符。例如字符串连接运算符(+),其写法和加法运算符一样,当有字符串参与时表示连接;分量运算符(.),用于类或对象引用其分量;下标运算符([]),用于数组或元组引用其分量。

可选链运算符(?.),该运算符在使用一个对象的分量时首先判断该对象是否为空(null)或非定义(undefined),如果为 null 或 undefined 分量运算,则会得到 undefined,如果不为 null 或 undefined,则相当于分量运算符(.)。示例代码如下:

```
class User{
    id : number | null = null                //id 初始值设置为 null
    name ?: string                           //有"?"在这里表示 name 为可选属性
}
let user = new User()
console.log( typeof user.id )                //输出 object,null 对应类型
console.log( typeof user.name )              //输出 undefined,name 没有被赋值
let s = user.id?.toString()                  //id 可能为空,使用 user.id.toString()会报错
console.log( s )                             //输出 undefined
console.log( user.name?.toString() )         //输出 undefined
```

非空断言运算符(!.),该运算符可以告诉编译器"当前值确定不会是 null 或 undefined",进而绕过编译器的检查。不过在运行阶段,如果出现判断是 null 或 undefined,则程序会抛出错误。

空值合并运算符(??),该运算符是一个二元运算符,当左侧操作数为 null 或 undefined 时,其结果取右侧操作数的值,否则取左操作数的值。示例代码如下:

```
let name : string|null|undefined;
console.log( "欢迎您 " + (name??"游客") )        //输出 欢迎您 游客
name = "张三"
console.log( "欢迎您 " + (name??"游客") )        //输出 欢迎您 张三
```

ArkTS 运算符的使用和很多其他高级编程语言中的运算符类似,这里限于篇幅不再详细说明。

3.4 控制语句和函数

3.4.1 控制语句

程序的基本控制结构有顺序结构、分支结构和循环结构,这一点在所有的高级编程语言中都是适用的。

顺序结构比较简单,程序按照语句的先后次序执行,不需要控制语句进行控制。

1. 分支语句

分支结构也称为选择结构,分支语句根据不同的条件来选择不同的分支进行运行,ArkTS 分支语句有 if 语句、if-else 语句、if-else if-else 语句、switch-case 语句。

if 语句由一个布尔表达式后跟一条或多条语句组成,语法格式如下:

```
if( 布尔表达式 ){
   //当布尔表达式为 true 时执行语句块
}
```

if-else 语句有两个分支,if 后跟一个,else 后跟一个,语法格式如下:

```
if( 布尔表达式 ){
   //在布尔表达式为 true 时执行
}else{
   //在布尔表达式为 false 时执行
}
```

if-else if-else 语句相当于在 if-else 语句中嵌套了一条 if-else 语句,在执行多个判断条件时比较有用,语法格式如下:

```
if( 布尔表达式 1 ) {
   //在布尔表达式 1 为 true 时执行
} else if( 布尔表达式 2 ) {
   //在布尔表达式 2 为 true 时执行
} else if( 布尔表达式 3 ) {
   //在布尔表达式 3 为 true 时执行
} else {
   //在前面布尔表达式都为 false 时执行
}
```

switch-case 语句是一条多路分支语句,一条 switch 语句允许测试一个条件表达式等于多个分支表达式值的情况,每个分支称为一个 case,语法格式如下:

```
switch( 条件表达式 ){
    case 分支表达式 1 :
        //执行语句
        break;                        //可选 执行 break 跳出 switch 语句
```

```
    case 分支表达式 2 :
        //执行语句
        break;                      //可选 如果没有 break,则继续向下执行
    //可以有任意多个 case
    default : /* 可选的 */
        //默认执行语句
}
```

使用 switch 语句应遵循以下规则：

(1) 在一条 switch 语句中可以有若干 case 分支，每个 case 分支后跟一个分支表达式和一个冒号。

(2) 当被判断的条件表达式的值等于 case 后面的分支表达式的值时，case 后跟的语句将被执行，直到遇到 break 语句为止。

(3) 当遇到 break 语句时，switch 语句终止，控制流将跳转到 switch 语句后。

(4) 不是每个 case 都需要包含 break 语句。如果 case 不包含 break 语句，则控制流将会继续判断及执行后续的 case，直到遇到 break 语句为止。

(5) 一条 switch 语句可有一个可选的 default，出现在 switch 语句结尾。default 可用于在上面所有 case 都成功时执行一项任务，default 不是必需的。

2. 循环语句

循环结构是编程过程中常用的结构，ArkTS 语言中循环语句有 for 语句、while 语句、do-while 语句，另外还支持 for-of、forEach、every 和 some 循环。

for 语句的语法格式如下：

```
for( 表达式 1 ; 表达式 2 ; 表达式 3 ){
    //循环体代码
}
```

for 循环语句的控制流程如下：

(1) 首先执行表达式 1，并且只执行一次。表达式 1 一般为初始化表达式，表达式 1 可以为空。

(2) 接着会判断表达式 2，如果表达式 2 结果为 true，则执行循环体。如果值为 false，则循环终止。表达式 2 是循环条件表达式。

(3) 每执行一遍循环体后，控制流都会跳到表达式 3。表达式 3 一般为增量表达式，表达式 3 也可以为空。

(4) 再次执行表达式 2，如果表达式 2 为 true，则继续循环，这个过程会不断重复，直到表达式 2 为 false 时，循环终止。

while 循环也称为当型循环，while 语句的语法格式如下：

```
while( 循环条件表达式 ){
    //循环体
}
```

在 while 语句中，当循环条件表达式为 true 时，执行循环体，否则终止循环。

do-while 循环也称为直到型循环，do-while 语句首先执行一遍循环体，在循环的尾部检查循环条件表达式，其语法格式如下：

```
do{
    //循环体
}while( 循环条件表达式 );
```

在 do-while 语句中，同样是当循环条件表达式为 true 时，继续执行循环体，否则终止循环。

for-of 循环语句是在 ES6 中引入的，在 ArkTS 中也支持。for-of 语句允许遍历 Arrays（数组）、Strings（字符串）、Maps（映射）、Sets（集合）等可迭代的数据结构，示例代码如下：

```
let arr = [ 6, "some", false ]
for(let e of arr) {
     console.log( e.toString() );                  //依次输出 6、some 和 false
}
```

注意，ArkTS 不支持 for-in 循环语句。

forEach、every 和 some 是 JavaScript 的循环语法，ArkTS 也同样支持。这 3 个循环更像是可以迭代对象的方法。

forEach 的基本用法，示例代码如下：

```
const arr = ['a', 'b', 'c'];
//对每个元素执行操作
arr.forEach( element => console.log(element.toString()) );

arr.forEach( ( value, index, array ) => {            //又一种用法
    //value 代表当前值
    //index 代表当前下标
    //array 代表数组本身
    }
);
```

因为 forEach 在迭代器中是无法返回的，所以可以使用 every 和 some 来取代 forEach。every 循环的代码如下：

```
let list = [100, 200, 300];
list.every((value, index, array) => {
    //value 代表当前值
    //index 代表当前下标
    //array 代表数组本身
    return r;            //当返回值为 true 时继续,当返回值为 false 时退出循环
                         //在循环过程中一旦返回值为 false,循环就停止
    }
);
```

some 循环的代码如下：

```
let list = [100, 200, 300];
list.some((value, index, array) => {
    //value 代表当前值
    //index 代表当前下标
    //array 代表数组本身
    return r;                    //在循环过程中一旦返回值为 true,循环就停止
    }
)
```

every()方法使用指定函数检测数组中的所有元素，如果数组中检测到有一个元素不满足(以返回值为 false 判定)，则剩余的元素不会再进行检测，every()的返回值也为 false。如果所有元素都满足条件，则 every()最终返回值也为 true。

some()方法依次执行数组的每个元素，如果有一个元素满足条件(以返回值为 true 判断)，则剩余的元素不会再执行检测，some()的返回值也为 true。如果没有一个满足条件的元素，则 some()的最终返回值也为 false。

3. 跳转语句

在使用循环的过程中，可以使用 break 和 continue 语句进行跳转。

当在循环体内执行到 break 语句时，循环会立即终止，并且程序流将继续执行紧接着循环的下一条语句。如果循环有嵌套关系，则 break 语句停止的时候是其所在的当前层循环，语法格式如下：

```
break;                       //分号可以省略
```

当在循环体内执行到 continue 语句时，它不是终止循环，而是跳过当前循环中的剩余代码，执行当前循环的下一轮循环。对于 for 循环，continue 语句执行后会跳转到其第 3 个表达式。对于 while 和 do-while 语句，continue 语句执行后会跳转到执行循环判断条件，其语法格式如下：

```
continue;                    //分号可以省略
```

3.4.2 函数

1. 一般函数

函数是由若干语句组成的功能块，函数是组织程序的有效方法。函数声明需要让编译器能够辨析函数名、参数和返回类型。函数体是函数的执行功能代码块。ArkTS 语言中定义函数的基本语法格式如下：

```
function 函数名( 参数:类型 ):返回类型 { //function 为关键字
    //函数体代码
}
```

调用函数的基本语法格式如下：

```
函数名( 实际参数 )
```

在定义函数时，函数名必须符合标识符规则。

在函数内返回，可以执行 return 语句，return 返回的类型应该和函数的返回类型一致。

在定义函数时，可以包含多个参数，多个参数之间以逗号分隔。当函数有多个参数时，调用时也应传入相应个数的实际参数，实际参数会对应传递给函数的参数。

在定义函数时，可以定义可选参数，可选参数调用时可以不传递实际参数，可选参数比必须参数在函数定义形式上多一个问号(?)。定义可选参数函数的基本语法格式如下：

```
function 函数名( 参数 1: 类型 1, 可选参数 ?: 类型 ) {
    //函数体代码
}
```

在有可选参数的情况下，可选参数必须跟在必须参数后面。

在定义函数时，可以设置参数的默认值，这样在调用函数时，如果不传入该参数的值，则使用默认参数值，定义带默认参数的函数的基本语法格式如下：

```
function 函数名( 参数 1:类型 , 参数 2:类型 = 默认值) {
    //函数体代码
}
```

在上面的代码中，参数 2 是具有默认值的参数。对于默认值参数，在函数调用时既可以传递参数(此时会使用传递的参数值)，也可以不传递参数(此时会使用默认值)。默认值参数不能放在必须参数前面。

在定义函数时，参数不能同时设置为可选参数和默认值参数。

在定义函数时，还可以定义剩余参数，剩余参数是针对函数参数个数不确定的情况的，剩余参数语法允许将一个不确定数量的参数作为一个数组传入，剩余参数的声明名称前有 3 个点(...)，示例代码如下：

```
function getSum( ...nums:number[] ) {                  //带有剩余参数
    let i:number;
    let sum:number = 0;
    for( i = 0;i < nums.length;i++) {
        sum = sum + nums[i];
    }
    console.log("sum = %d",sum)
}
getSum(1,2,3)                                         //调用,可以传入任意多个 number
getSum(1,2,3,4,5)                                     //调用,可以传入任意多个 number
```

在定义函数时，如果有剩余参数，则必须是最后一个参数，并且只能有一个剩余参数。

2. Lambda 函数

Lambda 函数也可以称为箭头函数，是一种基于 Lambda 表达式的函数形式，也可以理解成是一种特殊的匿名函数，其箭头表达式形式比较简洁，示例代码如下：

```
let f = (x:number) => 10 + x                //表示给参数 x,执行箭头(=>)后面的语句
f(5)                                        //得到 15
```

当执行的表达式比较多时，可以采用花括号括起来，示例代码如下：

```
let f = (x:number) => {                     //表示给参数 x,执行箭头(=>)后面语句块
   x = x + 10
   return x + 20
}
f(5)                                        //得到 35
```

在无须参数的情况下，可以用空括号，示例代码如下：

```
let show = () => { console.log("something") }
show()
```

Lambda 函数也可以有返回值类型，示例代码如下：

```
let f = ( x:number ):string => {            //返回类型为 string
    return "x = " + x
}
console.log( f(6) )
```

在使用 Lambda 表达形式时，其箭头(=>)前相当于函数头说明，箭头(=>)后面相当于函数体。

3. 异步函数

异步函数是指函数被调用时不阻塞后续代码执行，异步编程允许并发执行多项任务，以提高执行效率，可以通过 async 关键字定义异步函数，示例代码如下：

```
//定义异步函数
async function asyncFun() {
      await setTimeout( () =>{}, 1000 )   //耗时 1s
      console.log('异步函数中输出内容')
      return '异步函数调用的返回值'
}
//调用异步函数,下面一行调用在后面的输出之前
asyncFun()
//异步函数调用后的代码,上一行调用了异步函数
console.log('后面执行的代码输出语句,先输出');
```

以上代码调用了异步函数 asyncFun()，该函数执行了耗时操作(这里用 setTimeout()代替)，因为其为异步函数，调用后代码不会阻塞而是继续执行后面的代码，因此输出的结果

如下：

```
后面执行的代码输出语句,先输出
异步函数中输出内容
```

如果一个函数被声名为异步函数，则它什么时候会返回是不确定的，当需要等到被调用的异步函数执行完毕后再继续调用后的代码时，可以使用 await 关键字，即等待。在上面的代码中 setTimeout() 也是一个异步函数，因此其后面的输出会延时 1s。需要注意的是 await 关键字只能用在 async 声明的异步函数内部。

异步函数返回的是 Promise 对象，使用异步函数返回的结果需要在 then 中进行，示例代码如下：

```
//调用异步函数
let s = asyncFun()
s.then(value => {             //得到异步返回结果后执行
        console.log(value);
    })
//异步函数调用后的代码
console.log('后面调用的代码,先输出');
```

以上调用输出的结果如下：

```
后面调用的代码,先输出
异步函数中输出内容
异步函数调用的返回值
```

总之，异步函数调用不会阻塞后续代码执行，异步函数的调用结果一般在消耗一定时间后才能返回，但是异步函数调用会立即返回一个 Promise 对象，可以通过其 then 操作等待函数结果返回后进行进一步处理。在 HarmonyOS 应用开发的 API 中有大量的异步接口，开发者需要深入理解异步机制。

3.5 类、接口和泛型

10min

3.5.1 类和对象

ArkTS 是面向对象的语言，支持面向对象的基本特性，具有类、对象、接口、继承等语言表达。

1. 类的定义

类是描述所创建的对象共同的属性和方法的抽象，在 ArkTS 中，类定义的基本格式如下：

```
class 类名 {
    //类体
}
```

定义类的关键字为 class,后面紧跟类名,然后是花括号括起来的类体。类体中可以包含属性和方法,示例代码如下:

```
class Person {
    name:string;                                    //属性
    age:number;                                     //属性
    constructor(name:string,age:number) {           //构造函数
        this.name = name
        this.age = age
    }
    show():void {                                   //方法，注意前面没有 function 关键字
        console.log("姓名:" + this.name + " 年龄:" + this.age)
    }
}
```

2. 创建使用对象

对象是类的实例,使用 new 关键字创建对象,语法格式如下:

```
let 对象名 = new 类名( 参数 )                        //参数
```

通过类创建对象时会调用相应的构造函数,示例代码如下:

```
let obj = new Person("张三",18)
```

类中的属性和方法可以使用点运算符(.)访问,示例代码如下:

```
obj.name = "李四"                                   //访问属性
obj.show()                                          //访问方法
```

3. 成员权限

类中的成员有公有(public)、私有(private)与保护(protected)3 种访问权限,在默认情况下为 public 权限,成员权限说明的示例代码如下:

```
class OtherPerson {
    public     name:string = '';                    //公有属性
    protected  id:string = '';                      //保护属性
    private    age:number = 18;                     //私有属性
    nickname:string = '';                           //省略权限,公有属性
    public show():void {                            //公有方法,public 可以省略
        console.log("姓名:" + this.name )
    }
}
```

具有公有权限的成员在类外和类内都能访问,具有私有权限的成员只能在类内访问,具有保护权限的成员可以在类内和子类中访问。

4. readonly 关键字

关键字 readonly 表示只读,当一个类中属性由 readonly 修饰时,类实例化的对象属性

将只能读不能修改,示例代码如下:

```
class Book{
    public readonly price:number;               //公有只读属性
    constructor(price:number) {                 //构造函数
        this.price = price
    }
}
let obj = new Book(66)                          //实例化对象
obj.price = 88                                  //错误,只读属性不能改写
```

5. static 关键字

关键字 static 用于将类的成员(属性和方法)定义为静态的,静态成员属于类本身,直接通过类名调用,示例代码如下:

```
class Car {
    static num:number;                          //静态成员,默认为公有访问权限
    public static display():void {              //public 可以省略
        console.log("num 值为 " + Car.num)
    }
}
Car.num = 12                                    //初始化静态变量
Car.display()                                   //调用静态方法
```

6. 类的继承

ArkTS 中类的继承使用关键字 extends,在 ArkTS 中只支持单继承,即一个类最多只能有一个父类。继承可以重写方法,可以通过 super 访问父类中的成员,示例代码如下:

```
class Student extends Person {
    major:string                                //增加了新属性
    constructor(n:string,a:number,m:string) {   //构造函数
        super(n,a)                              //调用父类构造函数
        this.major = m
    }
    show():void {                               //重写方法
        super.show()                            //调用父类中的函数
        console.log("专业:" + this.major)
    }
}
let stu = new Student("张三",19,"计算机")
stu.show()
```

7. 抽象类

抽象类是不能进行实例化对象的类,抽象类天生就是为了继承,抽象类中可以定义抽象属性和抽象方法,为子类规定属性和方法的签名。抽象类由 abstract 关键字修饰,定义抽象类的基本语法如下:

```
abstract class 类名{
    //抽象属性
    abstract 属性名;
    //抽象方法
    abstract 方法名;
    //也可以包含非抽象属性和方法
    属性名或方法
}
```

3.5.2 接口

接口是一系列抽象的声明,ArkTS 中定义接口的基本语法如下:

```
interface 接口名 {     }
```

接口在 ArkTS 中使用非常灵活,它可以作为类型限制数据,示例代码如下:

```
interface LabelType {
    tag: string
}
function printLabel( label: LabelType) {
    //这里参数 label 要符合 LabelType 接口规则
    //参数必须包含一个 tag 属性
    console.log(label.tag)
}
```

接口可以继承,也就是说接口可以扩展于其他接口,接口可以继承一个或多个接口,继承关键字是 extends,多继承时父接口以逗号分隔,接口继承的语法格式如下:

```
interface sub_name extends super_name1 [, super_name2 ]
```

关于继承的基本使用,示例代码如下:

```
interface I1 {a:number}
interface I2 {b:number}
interface I extends I1, I2 { }
let Iobj:I =  {a:100, b:200}
console.log("a:" + Iobj.a + " b:" + Iobj.b)            //输出 a:100 b:200
```

类可以实现接口,实现接口的关键字是 implements,示例代码如下:

```
interface IRun {                                       //定义接口
    n:number
    run():void
}
class Run implements IRun {                            //实现接口
    n:number
    run():void{
```

```
        let i = 0
        for( i = 0; i < this.n; i++ ){
            console.log( i.toString() )
        }
    }
    constructor( n:number ) {                    //构造函数
        this.n = n
    }
}
let obj = new Run( 6 )                           //创建对象
obj.run()
```

3.5.3 泛型

泛型是参数化类型，一些参数的数据类型在定义函数或类时可以暂时不指定具体类型，而采用类型参数。泛型是对类型的泛化或抽象，使一些定义更具有广泛性。

1. 泛型函数

如果一个函数在定义时使用类型参数，则该函数称为泛型函数。在定义泛型函数时，首先需要在函数名后面使用尖括号说明类型参数，然后可以在函数定义中使用泛型参数。定义一个泛型函数的示例代码如下：

```
function find<T>( array:T[],value:T ) {
    //T为类型参数,可以有多种类型参数
    for( let v of array ) {
        if( v == value )
            return true
    }
    return false
}
```

在调用泛型函数时，需要确定类型参数的具体类型，具体类型既可以显式指明，也可以由编译器自动推定，例如对于上面的泛型函数调用的示例代码如下：

```
let r:boolean
r = find<number>( [1,2,3,4,5] , 3 )              //类型T为number
console.log( r.toString() )                      //输出true,找到了3
r = find ( ["赵","钱","孙","李"] , "王" )         //自动推定类型,T为string
console.log( r.toString() )                      //输出false,未找到王
```

2. 泛型类

在定义一个类时，如果使用一个或多个类型参数，则该类称为泛型类。在定义泛型类时，首先需要在类名后面使用尖括号说明类型参数，然后可以在类定义中使用泛型参数。定义一个泛型类的示例代码如下：

```
class KVPair < K, V > {                           //K 和 V 都是类型参数
    key : K;
    value : V;
    constructor( key:K, value:V ){                //构造函数
        this.key = key;
        this.value = value;
    }
}
```

在使用泛型类实例化对象时,需要确定类型参数的具体类型,具体类型既可以显式指明,也可以由编译器自动推定,例如对于上面的泛型类实例化对象的示例代码如下:

```
//显式指定类型实例化对象
let kv1 = new KVPair < string , number >( "张三" , 99 )
//自动推定类型实例化对象
let kv2 = new KVPair( "李四" , 98 )
//注意类型的一致性
let kv3:KVPair < string,number > = new KVPair( "王五" , 98 )
//类型可以根据需要确定
let kv4 = new KVPair( "赵六" , "男" )

console.log( kv1.value + "" )                 //输出 99
console.log( kv2.key )                        //输出李四
console.log( typeof(kv3.key) )                //输出 string
console.log( kv4.value )                      //输出男
```

9min

3.6 导出与导入

在处理模块化代码方面,JavaScript 的不同方法有一定的历史,自 TypeScript 诞生以来,其实现了对多种格式的支持,但随着时间的推移,逐渐融合为 ES6 模块的格式上,其基本就是 import/export 语法。

在 ArkTS 中,默认情况下一个 ets 文件中的顶层声明包括常量、变量、函数、类、接口等,在其他文件中是不可见的。如果需要在模块外可见,则需要明确使用 export 语句导出。同时,使用时需要通过 import 语句导入所导出的模块中的常量、变量、函数、类、接口等。两个模块之间的关系是通过在文件级别上使用 export 语句和 import 语句建立的。

如果希望在另一个文件中访问本文中的声明,则可以通过 export 语句导出常量、变量、函数、类、接口等,示例代码如下:

```
//file1.ets
export const PI = 3.14159                     //导出常量
export let v = 2.718                          //导出变量
export function abs(num: number) {            //导出函数
    if (num < 0){
       return num * -1
```

```
    }
    return num
}
export class User {                           //导出类
    //这里省略了类的具体实现代码
}
export interface USB{                         //导出接口
    //这里省略了代码
}
```

对于上述文件中的导出声明，可以在另外文件中通过 import 语句导入使用，import 语句必须位于 ets 文件的最前面，即 import 语句前不能有非 import 语句，代码如下：

```
//file2.ets
import { PI, v, abs } from "./file1"          //导入
console.log( PI + "" )
const a = abs( -100 * v )
console.log( a + "" )
import { User } from "./file1"                //错误，导入语句需放到文件前面
```

导入时，使用 import {old as new}可以为已有的标识符重命名，代码如下：

```
//file3.ets
import { PI as π } from "./file1";            //以 π 作为 pi 的新名
console.log( π + ""); //由于数字不能直接作为 log 的参数，所以这里连接了内容为空的串
```

导入时，使用 * as name 可以把所有导出的对象放入一个命名空间下，代码如下：

```
//file4.ets
import * as math from "./file1"               //放到命名空间 math 下
console.log( math.PI + "" )                   //通过命名空间访问 PI
const a = math.abs( math.v )                  //通过命名空间访问 v
```

除了可以使用 export 语句进行导出外，还可以使用 export default 语句进行默认导出，代码如下：

```
//file5.ets
export default function helloWorld() {
    console.log("您好，世界！")
}
```

在另外一个文件中，在导入默认导出时，可以通过 import 语句导入模块，此时导入时不加花括号，代码如下：

```
//file6.ets
import helloWorld from "./file5"              //从文件 file5.ets 导入函数
helloWorld();                                 //调用函数
```

使用 export 语句和 export default 语句两种方式导出，二者的主要区别如下：

(1) export 语句为导出，export default 语句为默认导出。

(2) 在一个文件或模块中，export 语句、import 语句可以有多条，但 export default 语句只能有一条，即默认导出只能有一条。

(3) 通过 export 语句方式导出，在导入时需要加花括号{}；而通过 export default 方式导出，在导入时则不需要加花括号{}。

当导入使用 export default 语句导出的声明时，可以不需要知道所要加载模块的标识符名称，代码如下：

```
//file7.ets
let s = "something"
export default s                         //默认导出 s
//这里相当于为 s 变量值"something"起了一个系统默认的变量名 default
//由于 default 只能有一个值，所以一个文件内不能有多条 export default 语句
```

对于默认导出，在另一个文件中导入使用时，可以直接重新命名，代码如下：

```
//file8.ets
import any1 from "./file7"
import any2 from "./file7"
//本质上，file7.ets 文件的 export default 语句输出一个名为 default 的变量
//系统允许为它取任意名字且不需要用花括号
console.log( any1 )                       //输出 something
console.log( any2 )                       //输出 something
```

导出和导入可以很好地组织多文件应用，关于模块的导入和导出还有其他的一些用法，这里不再阐述。在 HarmonyOS 应用开发过程中，往往需要使用鸿蒙 SDK 提供的各种能力，这时就需要导入相应的接口模块，以便使用，例如如下导入：

```
import { AbilityConstant, UIAbility, Want } from '@kit.AbilityKit';
```

9min

3.7 装饰器

装饰器(Decorator)在 TS 中是非常常用的，它可以使程序变得更加美好。许多库是基于装饰器构建的，例如 Angular、Nestjs 等，在 ArkTS 中也有很多装饰器。这里简要地介绍装饰器，以便读者能够更好地进行 HarmonyOS 应用开发。

装饰器在本质上是一种特殊的函数，可以被应用在类、类属性、类方法、类访问器、类方法的参数上，从而改变原有的功能。装饰器很像是组合一系列函数，通过它可以轻松地实现代理模式来使代码更简洁。

装饰器在使用上语法十分简单，只需在想使用的装饰器前加上@符号，这样装饰器就会被应用到对应的目标上。下面是一个在类上使用装饰器的示例，代码如下：

```
function test( target:Object ) {                    //一个函数
    console.log("This is test")
}
@test                                               //在类 A 上使用装饰器
class A {}
```

如果一个函数返回一个回调函数，则在这个函数作为装饰器来使用时，这个函数就是装饰器工厂，代码如下：

```
function test() {
    console.log('test out');
    return (target) => { console.log('test in') }
}

@test()
class A{ }
```

上方代码的含义为给 A 这个类绑定了一个装饰器工厂，在绑定的时候由于在函数后面写上了()，所以会先执行装饰器工厂以获得真正的装饰器，真正的装饰器会在定义类之前执行，所以会接着执行里面的函数，上方代码输出的结果如下：

```
test out
test in
```

装饰器只在解释执行时执行一次，下面是一个说明示例，代码如下：

```
function f(target) {
    console.log('a decorator')
    return target
}

@f
class A {}
```

以上代码会输出 a decorator，即使没有使用类进行定义对象，这里也会执行装饰器。

装饰器可以组合，即将装饰器组合起来一起使用。组合使用的装饰器会按从上至下的顺序依次执行所有的装饰器工厂，获得所有真正的装饰器后，再按从下至上的顺序执行所有的装饰器，示例代码如下：

```
function t1(target) {
    console.log('t1')
}
function t2(target) {
    console.log('t2')
}
function f1() {
```

```
    console.log('f1 out')
    return (target) => { console.log('f1 in') }
}
function f2() {
    console.log('f2 out')
    return function (target) { console.log('f2 in') }
}
@t1
@f1()
@f2()
@t2
class A { }
```

以上代码的执行结果如下：

```
f1 out
f2 out
t2
f2 in
f1 in
t1
```

装饰器按照其应用的目标可以分为5种，分别为类装饰器、属性装饰器、方法装饰器、访问器装饰器、参数装饰器。5种不同的装饰器应用在不同的地方，使用5种装饰器的示例代码如下：

```
@classDecorator                          //类装饰器
class Car {
    @propertyDecorator                   //属性装饰器
    name: string;
    @methodDecorator                     //方法装饰器
    accelerate(
        @parameterDecorator              //参数装饰器
        num: number
    ) {}
    @accessorDecorator                   //访问器装饰器
    get speed() {}
}
```

关于装饰器的更多阐述可以参阅官方网络链接 https://www.typescriptlang.org/docs/handbook/decorators.html。通过装饰器可以改变现有定义的行为，装饰器的使用场景包括项前或项后回调、监听属性变化、监听方法调用、对方法参数进行转换、添加新方法、添加新属性、运行时类型检查、自动编解码、依赖注入等。通过装饰器可以简化编码，在HarmonyOS中基于ArkTS的开发中定义了很多装饰器，ArkTS中的装饰器将在后续章节中进一步地进行介绍。

小结

本章简要地介绍了 ArkTS 的基础知识，包括基本类型、运算符、控制语句、函数、类、接口、导出与导入、装饰器等。ArkTS 是基于 TS 和 JS 的编程语言，其包含了 TS 和 JS 的众多特性，同时 ArkTS 提出了一些新的规范。在 ArkTS 的基础上，HarmonyOS 实现了声明式开发范式方舟开发框架(ArkUI)等，该框架是目前 HarmonyOS 应用开发的主推框架，第3章将重点介绍该框架。

习题

第3章习题

1. 判断题

(1) TypeScript 是 JavaScript 的超集。(　　)

(2) ArkTS 不可以使用 for-in 循环语句。(　　)

(3) ArkTS 中定义函数需要使用 function 关键字。(　　)

(4) ArkTS 支持 void 类型、null 类型和 undefined 类型。(　　)

(5) ArkTS 中类的成员有公有(public)、私有(private)与保护(protected)3 种访问权限，在默认情况下为 private 权限。(　　)

2. 选择题

(1) ArkTS 中关于变量名说法正确的是(　　)。

A. 变量名不能以数字开头

B. 变量名可以包含字母和数字

C. 不能包含空格及除 _ 和 $ 外的其他特殊字符

D. 以上都正确

(2) 下面不是 ArkTS 支持的数据类型的是(　　)。

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

(3) ArkTS 源文件的后缀是(　　)。

A. .ats　　B. .arkts　　C. .ts　　D. .ets

(4) 可选链运算符是(　　)。

A. ?=　　B. ?.　　C. ?!　　D. ??

(5) ArkTS 中匿名函数的正确写法是(　　)。

A. "."语法　　B. "=>"语法　　C. noname　　D. anonymous

3. 填空题

(1) 在 ArkTS 语言中类型运算符包括 typeof 和________。

(2) 在使用循环的过程中，可以使用________和 continue 语句进行跳转。

(3) 在 ArkTS 语言中，定义类的关键字为________。

（4）在 ArkTS 语言中定义异步函数需要使用关键字________。

（5）在使用 Lambda 表达形式时，其箭头（=>）前相当于函数头说明，箭头（=>）后面相当于________。

（6）在 ArkTS 语言中，定义接口的关键字为________。

（7）在定义一个类时，如果使用一个或多个类型参数，则该类称为________。

4. 简答题

（1）试说明使用 export 语句和 export default 语句两种导出方式的区别。

（2）试说明调用异步和非异步函数的区别。