第 3 章

TypeScript 基础

【学习目标】

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

本章介绍 TypeScript 基础，由于华为公司推出的 ArkTS 是基于 TypeScript 语言的，因此要求读者具有一定的语言基础，已经熟悉 TypeScript 的读者可以跳过本章。另外，本章对 TypeScript 的介绍是简明扼要的，是以在 HarmonyOS 应用开发中可以使用 TypeScript 语言为目的，如果读者需要进一步掌握 TypeScript，则需要学习其他专门针对该编程语言的资料。

3.1 TypeScript 语言简介

TypeScript 是由微软开发的自由和开源的编程语言，它是 JavaScript 的一个超集。JavaScript 从 1995 年问世，现在已成为最广泛使用的跨平台语言之一。虽然采用 JavaScript 编写的程序的大小、范围和复杂性呈指数级增长，但 JavaScript 语言表达严重不足，同时编码过程中的常见拼写、类型等错误都不能预先检查。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 的目标更多是开发大型应用，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 语言为 JavaScript 增加了新的特性，主要包括以下几种新特性。

- 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

TypeScript 包含了 JavaScript 的全部特性，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联合会(European Computer Manufactures Association，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。

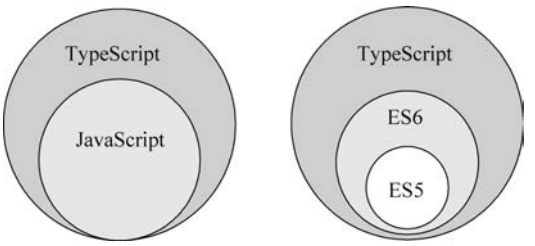

图 3-1　TypeScript、JavaScript、ES5、ES6 关系

3.2　TypeScript 简单使用

使用 TypeScript 编写程序，需要安装其编译环境，安装编译环境首先需要安装 NPM 工具，NPM 的全称是 Node Package Manager，是一个 Node.js 包管理和分发工具，已经成为 Node 模块的标准。NPM 是可以随同 Node.js 一起安装的包管理工具，因此可以直接安装 Node.js 以安装 NPM。关于 Node.js 的安装可以访问其官网下载并安装。

假设本地环境已经安装了 NPM 工具，可以使用以下命令来安装 TypeScript。首先通过命令设置镜像网址，命令如下：

```
npm config set registry https://registry.npmmirror.com
```

通过命令安装 TypeScript，命令如下：

```
npm install -g typescript
```

待安装完成后，可以使用 tsc -v 命令查看版本信息，如果显示出版本信息，则说明 TypeScript 编译器已经安装成功，显示版本信息的命令如下：

```
tsc -v
Version 4.7.4
```

在当前目录下，建立一个文本文件，并输入下面的内容，将文件名保存为 app.ts。TypeScript 源程序的扩展名为 ts，代码如下：

```
var msg:string = "Hello World"
console.log(msg)
```

这样，一个最简单的 TypeScript 程序就写好了，然后可以执行以下命令将 TypeScript 代码转化成 JavaScript 代码如下：

```
tsc app.ts
```

此时，会在当前目录下生成一个 app.js 文件，其内容如下：

```
var msg = "Hello World";
console.log(msg);
```

接下来，可以使用 node 命令来执行 JavaScript 代码，执行 app.js 文件的命令如下：

```
node app.js
```

上述由 app.ts 转换为 app.js 的过程表明 TypeScript 代码是通过 TypeScript 编译器(tsc)把代码转化成了 JavaScript 代码，如图 3-2 所示，最终真正运行的还是 JavaScript 程序。

图 3-2 TypeScript 编译成 JavaScript

TypeScript 会忽略程序中出现的空格、制表符和换行符。空格、制表符通常用来缩进代码，使代码易于阅读和理解。

TypeScript 区分大写和小写字符，即仅大小写不同的标识符也被认为是不同的标识符。

TypeScript 语句后面的分号(;)是可选的，当一行只有一个语句时，其后可以使用分号，也可以不使用，如果一行有多个语句，则需要使用分号分隔，代码如下：

```
console.log("Hello"); console.log("World");
```

TypeScript 支持两种类型的注释，单行注释采用“//”，一行中在//后面的文字是注释内容，多行注释采用“/*”“*/”，位于二者之间的多行内容为注释。这一点和很多编程语言相同。

3.3 基本类型和运算符

3.3.1 数据类型

TypeScript 是强类型语言,要求所有的量都要有明确的类型,TypeScript 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 类型和任意类型。

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
```

字符串类型:若干个字符组成的串,可以使用单引号('),也可以使用双引号(")引起来,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>`
)
```

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
```

数组类型：若干个相同的类型组成的一组数据，在一种类型后面加上中括号([])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]
```

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必相同。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[1] )                       //输出 10.6
t = [ 10.6,"price" ]                      //错误
```

枚举类型：枚举类型用于定义若干个数值集合，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 }
let c: Color = Color.Blue
console.log(c);                             //输出 2
```

函数空类型：即 void 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 {
    alert("This is something");
}
```

空类型：也称为 null 类型，表示对象值缺失，在 JavaScript 中 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 null 时，其返回的是 object。

未定义类型：即 undefined 类型，表示未定义，在 JavaScript 中，当一个变量没有设置值时就为 undefined。通过 typeof 判断一个没有值的变量会返回 undefined。

null 和 undefined 是其他任何类型(包括 void)的子类型,可以赋值给其他类型,赋值后的类型会变成 null 或 undefined,但是在 TypeScript 中启用严格的空校验(--strictNullChecks)特性,就可以使 null 和 undefined 只能被赋值给 void 或本身对应的类型,示例代码如下:

```
//启用 --strictNullChecks
let x: string;
x = "good";                                          //编译正确
x = undefined;                                       //编译错误
x = null;                                            //编译错误
```

在上面的例子中变量 x 只能是数字类型。如果一种类型可能出现 null 或 undefined,则可以用 | 来支持多种类型,示例代码如下:

```
//启用 --strictNullChecks
let y: number | null | undefined;
y = 1;                                               //编译正确
y = undefined;                                       //编译正确
y = null;                                            //编译正确
```

无果类型:即 never 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止点,如无限循环。never 类型是其他类型的子类型,包括 null 和 undefined,示例代码如下:

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

任意类型:也称为 any 类型,是针对类型不明确的变量使用的一种数据类型,常用于类型会动态改变的情况,示例代码如下:

```
let x: any = 1                  //数字类型
x = 'I am zhangsan'             //字符串类型
x = false                       //布尔类型
x.show()                        //当 x 是一个对象时,show 方法在运行时可能存在,编译时不检查
let arr: any[] = [ 0, true, 'good' ]                 //any 类型数组
arr[1] = 100
```

总体来讲，TypeScript 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，TypeScript 还支持自定义类型，如类等。

数据类型在使用的过程中一般遵循一致性原则，即什么类型的变量就赋予对应类型的值。

通过类型定义变量的一般格式如下：

```
修饰符  变量名:类型名 [ = 值]
```

TypeScript 变量的使用规则如下：

(1) 变量名可以包含数字、字母、下画线(_)和美元($)符号，不能包含其他特殊字符和空格。

(2) 变量名不能以数字开头。

(3) 变量使用前必须先声明。

(4) 变量类型可以自动推定和隐式转换。

变量的使用，示例代码如下：

```
var s1:string = "good"                    //定义变量并初始化
var s2:string                             //未初始化,变量值会被设置为 undefined
var s3 = "anything"                       //自动将 s3 推定为 any 类型
var s4                                    //s4 为 any 类型,值为 undefined
console.log(typeof(s4))                   //输出 undefined,因为 s4 未赋值
s4 = "anything"                           //var 修饰表示变量,可以多次被赋值
console.log(typeof(s4))                   //输出 string
let x: number = 100
console.log(typeof(x))                    //输出 number
x = s4                                    //any 类型可以赋值给 number,自动转换
console.log(typeof(x))                    //输出 string

x = s1                                    //错误,字符串不能赋值给数值实例
```

在 TypeScript 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：

```
var 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
```

3.3.2 运算符

TypeScript 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。

算术运算符包括加(+)、减(-)、乘(*)、除(/)、求余(%)、自增(++)和自减(--)。

关系运算符包括等于(==)、不等于(!=)、大于(>)、小于(<)、大于或等于(>=)、小

于或等于(<=)。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(!==)会要求值和类型都不相同。

逻辑运算符包括逻辑与(&&)、逻辑或(||)、逻辑非(!)。

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

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(/=)。

三元条件运算符只有问号冒号运算符(?:)。

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。

除了上述的运算符外,还有一些其他运算符。如负号运算符(-),其写法和减号运算符一样,作为一元运算符时表示负号;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。

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

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执行不同的分支,TypeScript 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个或多个语句组成,语法格式如下:

```
if( 布尔表达式 ){
    //布尔表达式 true 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 语句相对于在 if…else 语句中嵌套了一个 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 语句是一个多路分支语句，一个 switch 语句允许测试一个变量等于多个值的情况，每个值称为一个 case，语法格式如下：

```
switch( 表达式 ){
    case 常量表达式 1 :
        //执行语句
        break;                          //可选  执行 break,跳出 switch 语句
    case 常量表达式 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 语句必须遵循以下规则：

(1) 在一个 switch 中可以有任意数量的 case 语句，每个 case 后跟一个要比较的值和一个冒号。

(2) case 后的常量表达式必须和 switch 后括号内的表达式具有相同的数据类型，并且必须是一个常量或字面量。

(3) 当被判断的表达式值等于 case 中的常量时，case 后跟的语句将被执行，直到遇到 break 语句为止。

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

(5) 不是每个 case 都需要包含 break。如果 case 语句不包含 break，则控制流将会继续判断后续的 case 是否为真，直到遇到 break 为止。

(6) 一个 switch 语句可有一个可选的 default，出现在 switch 结尾。default 可用于在上面所有 case 都不为真时执行一个任务，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，TypeScript 语言中循环语句有 for 语句、for…in 语句、while 语句、do…while 语句，另外还支持 for…of、for…in、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 的结果为 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,以替代 for…in 和 forEach()。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 );                    //依次输出 6、some 和 false
}
```

for…in 和 for…of 都可以遍历可迭代的数据结构,不同的是,for…in 返回的是被迭代对象的键,而 for…of 返回的是被迭代对象的属性的值,示例代码如下：

```
let arr = [ 6, "some", false ]
for(let e in arr) {
     console.log( e );                                    //依次输出 0,1 和 2
}
```

forEach、every 和 some 是 JavaScript 的循环语法，TypeScript 作为 JavaScript 的语法超集，当然默认也是支持的。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'];
arr.forEach( element => console.log(element) );     //对每个元素执行操作

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
                  //在循环过程中一旦执行返回值为 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; //在循环过程中一旦执行返回值为 true 循环停止
    }
)
```

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()的最终返回值为 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()的返回值为 true。如果没有一个满足条件的元素,则 some()的最终返回值为 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,而是会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。TypeScript 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 )
```

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。

函数定义时,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,调用时也应输入相关个数的实际参数,实际参数会对应传递给函数的参数。

函数定义时,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(?)。定义可选参数的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。

函数定义时,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,则使用默认的参数值,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
}
```

默认值参数,在函数调用时也可以传递参数,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。默认值参数也必须跟在必须参数的后面。

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,剩余参数针对函数参数个数不确定的情况,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输入,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
    var i;
    var sum:number = 0;
    for( i = 0;i < nums.length;i++) {
        sum = sum + nums[i];
    }
    console.log("sum = ",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. 匿名函数

匿名函数是一个没有函数名的函数。一般在程序运行时动态声明,除了没有函数名外,其他的特性与一般函数一样。为了使用一般会将匿名函数赋值给一个变量,定义和调用匿名函数的基本用法如下:

```
var f = function() {                         //定义匿名函数,赋值给 f
        //函数体代码
    }
f()                                          //调用匿名函数
```

在不赋值给变量的情况下,匿名函数可以自调用,只需在函数后使用( ),示例代码如下:

```
function() {                                    //定义匿名函数
    //函数体代码
}()                                             //调用匿名函数
```

匿名函数可以带参数,这一点和一般的函数相同。

3. 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:

```
var 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:

```
var 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
```

当只有一个参数时,如果类型可以推定,则可以省略(),示例代码如下:

```
var f = x => {                                  //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:

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

Lambda 函数也可以有返回值类型,示例代码如下:

```
var 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。

3.5 类和接口

3.5.1 类和对象

TypeScript 语言可以说是面向对象的 JavaScript。TypeScript 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

类是描述了所创建的对象的共同的属性和方法的抽象，在 TypeScript 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
    constructor(name:string) {            //构造函数，有特定的名字，不和类同名
        this.name = name
    }
    show():void {                         //方法，注意前面没有 function 关键字
        console.log("姓名:" + this.name )
    }
}
```

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：

```
var 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：

```
var obj = new Person("张三")
```

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
```

对象其实就是包含一组键-值对的实例，这里的值可以是标量、函数、数组、对象等。在 TypeScript 中，除了可以通过类创建对象外，还可以直接使用类似 JSON 格式的方式创建对象，示例代码如下：

```
var obj = {
    name: "王五",                         //标量
    say: function() {                     //函数
        console.log( this.name )
    },
    scores:[100,99]                       //集合
```

```
}
obj.say()                                    //调用 say 函数
```

3. 成员权限

类中的成员有公有(public)、私有(private)与保护(protected)3 种保护权限,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 {
    private    name:string;                  //私有属性
    protected  id:string;                    //保护属性
    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. 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,可以直接通过类名调用,示例代码如下:

```
class Car {
    static num:number;                       //静态成员
    public static disp():void {              //public 可以省略
        console.log("num 值为 " + Car.num)
    }
}
Car.num = 12                                 //初始化静态变量
Car.disp()                                   //调用静态方法
```

5. 类的继承

TypeScript 中类的继承使用关键字 extends,在 TypeScript 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,m:string ) {       //构造函数
        super(n)                             //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)
    }
```

```
}
var obj = new Student("张三","计算机")
obj.show()
```

3.5.2 接口

接口是一系列抽象的声明，TypeScript 定义接口的基本语法如下：

```
interface 接口名 {     }
```

接口在 TypeScript 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 的接口规则
    //参数必须包含一个 tag 属性
    console.log( label.tag )
}
let obj = { api:9 , tag: "HarmonyOS" }
printLabel( obj )                                    //输出 HarmonyOS
```

接口可以继承，也就是说接口可以扩展于其他接口，TypeScript 中允许接口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 { }
var 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，其使用的示例代码如下：

```
interface IRun {                                     //定义接口
    n:number
    run():void
}
class Run implements IRun {                          //实现接口
    n:number
    run():void{
        var i
        for( i = 0; i < this.n; i++){
```

```
            console.log( i )
        }
    }
    constructor( n:number ) {              //构造函数
        this.n = n
    }
}
var obj = new Run( 6 )                     //创建对象
obj.run()
```

3.6 模块

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。ES 模块是在 2015 年被添加到 JavaScript 规范中的,到 2020 年得到了广泛的 Web 浏览器和 JavaScript 运行时支持。

TypeScript 模块的设计理念是可以替换的代码块。模块是在其自身的作用域里执行,这样定义在模块里面的变量、函数和类等在模块外部是不可见的。如果需要在模块外可见,则需要明确地使用 export 导出。同时,使用时需要通过 import 导入所导出的模块中的变量、函数、类等。两个模块之间的关系是通过在文件级别上使用 import 和 export 建立的。

模块使用模块加载器去导入其他的模块。在运行时,模块加载器的作用是在执行此模块代码前去查找并执行这个模块的所有依赖。常用的加载器是服务于 Node.js 的 CommonJS 和 Web 应用的 Require.js。

3.6.1 模块导出与导入

对于 ES 模块语法,文件可以通过 export default 方式声明默认导出,示例代码如下:

```
//ch03/hello.ts
export default function helloWorld() {
    console.log("您好,世界!")
}
```

在另外一个文件中,可以通过 import 导入模块,示例代码如下:

```
//ch03/app1.ts
import helloWorld from "./hello"           //从文件 hello.js 导入模块
helloWorld();                              //调用函数
```

除了默认导出之外,可以通过 export 导出多个变量和函数等,示例代码如下:

```
//ch03/maths.ts
export var pi = 3.14
```

```
export let e = 2.718
export class RandomNumber {
    //这里省略了类实现代码
}
export function abs(num: number) {
    if (num < 0){
        return num * -1
    }
    return num
}
```

对于文件 maths.ts 的导出,可以在别的文件中导入,示例代码如下:

```
//ch03/app2.ts
import { pi, e, abs } from "./maths"
console.log(pi)
const a = abs( -100 * e )
console.log(e)
```

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

```
//ch03/app3.ts
import { pi as π } from "./maths";          //以 π 作为 pi 的新名
console.log( π );
```

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,示例代码如下:

```
//ch03/app4.ts
import * as math from "./maths"             //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.e )                //通过命名空间访问 e
```

在 ES6 语法里,可以使用 export 和 export default 两种方式导出,二者的主要区别如下:

(1) export 为导出,export default 为默认导出。

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

(3) 通过 export 方式导出,在导入时需要加花括号{ },如果使用 export default 导出,则在导入时不需要加花括号{ }。

例如,对于 export 导出的基本用法如下:

```
//ch03/testa.ts
export const s = "something"
export function log( s ) {
   return s;
}
```

对应的导入方式的示例代码如下：

```
//ch03/testb.ts
import { s, log } from './testa'                    //注意带花括号,也可以分开导入(两次)
```

例如,对于 export default 导出的基本用法如下：

```
//ch03/testc.ts
const s = "something"
export default s                                    //注意 export default 导出只能有一个
```

默认导出所对应的导入方式,示例代码如下：

```
//文件 testd.ts
import s from './testc';                            //导入时不带花括号
export default s                                    //export default 导出只能有一个,这里再次导出 s
```

当使用 export default 为模块指定默认输出时,可以不需要知道所要加载模块的标识符名称,示例代码如下：

```
//ch03/teste.ts
let s = "something"
export default 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
//default 只能有一个值,所以一个文件内不能有多个 export default
```

对于默认导出,在另一个文件中导入时,示例代码如下：

```
//ch03/testf.ts
import any1 from "./teste"
import any2 from "./teste"
//本质上,teste.ts 文件的 export default 会输出一个叫作 default 的变量
//系统允许为它取任意名字且不需要用花括号包含
console.log( any1 )                                 //输出 something
console.log( any2 )                                 //输出 something
```

3.6.2 CommonJS 模块用法

CommonJS 是 NPM 上大多数模块的交付格式,其导出通过设置 exports 全局属性来导出的 module,使用 require 语句导入文件,示例代码如下：

```
//文件:testg.ts
function abs(num: number) {
    if (num < 0) return num * -1;
    return num;
}
module.exports = {
```

```
    pi: 3.14,
    abs,
}
```

对于以上文件代码，在其他文件中引入的示例代码如下：

```
//ch03/testh.ts
const maths = require("./testg");
console.log( maths.pi )
maths.abs( -3.6 )
```

总之，模块可以很好地组织多文件应用。另外，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。

3.7 装饰器

装饰器(Decorator)在 TypeScript 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，如 Angular、Nestjs 等。这里简要地介绍装饰器以便读者能够更好地进行基于 TypeScript 的 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。装饰器很像组合一系列函数，通过它可以轻松实现代理模式来使代码更简洁。

启用装饰器，可以通过命令行参数或在 tsconfig.json 文件中启用实验性装饰器编译器选项实现。通过命令行启用装饰器的命令如下：

```
tsc -- target ES6 -- experimentalDecorators
```

对于配置文件 tsconfig.json，启用装饰器的配置如下：

```
{
    "compilerOptions": {
        "target": "ES6",
        "experimentalDecorators": true
    }
}
```

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() {                          //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，当这个函数作为装饰器来使用时，这个函数就是装饰器工厂，示例代码如下：

```
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，在绑定时由于在函数后面写上了()，所以会先执行装饰器工厂以便获得真正的装饰器，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会执行一次，下面是一个说明示例，代码如下：

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

@f
class A {}
```

以上代码输出 a decorator，即使没有使用类定义对象，这里也会执行装饰器。

装饰器可以组合，即装饰器组合起来一起使用，组合使用的装饰器会从上至下地依次执行所有的装饰器工厂，获得所有真正的装饰器后，再从下至上地执行所有的装饰器，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 () { 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 的开发中使用了很多定义的装饰器。

小结

本章简要地介绍了 TypeScript 的基础知识，包括基本类型、运算符、控制语句、函数、类、接口、模块、装饰器等。TypeScript 是 JavaScript 的一个超集，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 更适合开发大型应用。

HarmonyOS 基于 TypeScript 进行了扩展，提出了 ArkTS 语言，并实现了声明式开发范式——方舟开发框架(ArkUI)，是目前 HarmonyOS 应用开发的主推框架，因此开发者有必要掌握 TypeScript 语言基础。