第 1 章

认识TypeScript

本章概述

TypeScript 是 Microsoft 公司开发的一款开源的脚本语言，同时实现 TypeScript 的编译器也是开源的。TypeScript 是在 JavaScript 的基础上添加静态定义类型构建而成的，最初是为了解决 JavaScript 代码规模大时出现的类型问题，同时 TypeScript 中添加了一些高级特性，如装饰器、函数等，这些特性在 JavaScript 中是不存在的。TypeScript 更像 Java、C++、PHP 等面向对象开发的语言，可以在开发大型企业级应用时使用。本章将讲解什么是 TypeScript，以及 TypeScript 的简单使用。

知识导读

本章要点（已掌握的在方框中打钩）

- ☐ 什么是 TypeScript。
- ☐ 为什么要学习 TypeScript。
- ☐ 安装 TypeScript。
- ☐ 第一个 TypeScript 程序。

1.1 什么是 TypeScript

TypeScript 是一种开源的编程语言，是专为 JavaScript 开发大规模应用而设计的，它包含 JavaScript 现有的所有功能。TypeScript 可以嵌入 HTML 页面中，在浏览器端执行。TypeScript 代码需要编译成 JavaScript 代码才能运行。TypeScript 语言是跨平台的，经过编译后可以运行在任意浏览器。

1.2 为什么要学习 TypeScript

TypeScript 是目前较为流行的一门编程语言，同时也是强类型语言，有一套类型机制，可以在编译时进行强类型判断，从而大大提高代码的编写效率。TypeScript 是在 JavaScript 的基础上开发的语法，重点解决了 JavaScript 语言自有类型系统的不足。TypeScript 拥有面向对象编程语言的所有特性，相比 JavaScript，更能提高项目的开发效率，且 TypeScript 具有多功能性，几乎可以编写任何代码，包括移动端和计算机端的应用程序、Web 服务的前端和后端等。使用 TypeScript 开发可以极大地提高代码的可靠程度。

1.2.1 TypeScript 与 JavaScript 对比有什么优势

TypeScript 和 JavaScript 是目前开发中常用的两种脚本语言。TypeScript 可以使用 JavaScript 中的所有代码，它是为了方便 JavaScript 的开发而创建的。相比 JavaScript，TypeScript 有以下 5 个优势：

（1）TypeScript 引入了“类”和“模块”的概念，可以把数据、函数和类封装到模块中。

（2）TypeScript 可以在编码时检查错误，大大提高了开发效率。

（3）TypeScript 的面向对象编程语言的特性，增加了代码的可读性。

（4）TypeScript 使代码的重构变得更加容易。

（5）TypeScript 可用于开发大型应用。

TypeScript 与 JavaScript 的对比如表 1-1 所示。

表 1-1 TypeScript 与 JavaScript 的对比

对比项目	JavaScript	TypeScript
类型	弱类型	强类型
模块化	不支持	支持
泛型	不支持	支持
接口	不支持	支持
浏览器中使用	支持	不支持

说明：

（1）弱类型：无法选择静态类型。

（2）强类型：支持静态和动态类型。

（3）JavaScript 支持在浏览器中直接使用。TypeScript 不支持在浏览器中直接使用，需要将代码转换为 JavaScript 以实现浏览器的兼容性。

1.2.2　TypeScript 给前端开发带来的好处

TypeScript 语言是 JavaScript 的超集，包含 JavaScript 的所有元素。相比 JavaScript，TypeScript 增加了代码的可维护性和可读性，且 TypeScript 的编译工具可以运行在任何系统中。对于前端的开发者来说，TypeScript 带来了以下 3 个好处：

（1）TypeScript 是一门强类型语言，相比弱类型语言，编码中的错误会更早暴露，使代码更加智能和准确，并且减少了不必要的类型判断。

（2）TypeScript 增、删、改了一些公共接口，与接口相关的页面和函数会及时提示报错，可以更方便地找到错误并修改。

（3）TypeScript 支持分模块开发，这样在开发大型项目时可以更好地进行分工协作。

1.3　安装 TypeScript

在学习 TypeScript 之前首先要进行 TypeScript 环境的搭建和代码编辑器的选择，从官网可以看到支持 TypeScript 的编辑器有 CATS、Eclipse、Visual Studio、Visual Studio Code 等。TypeScript 环境搭建的流程具体如下：

（1）安装 TypeScript 之前需要确认计算机中是否安装了 Node.js（可以在命令行工具中输入 node -v 查看安装的 Node.js 版本）。

（2）按 Win+R 组合键，输入 cmd，打开命令行界面，如图 1-1 所示。

（3）在命令行工具中输入 npm install -g typescript，按 Enter 键运行，将 TypeScript 安装到全局，如图 1-2 所示。

图 1-1　打开命令行界面

图 1-2　TypeScript 的安装

（4）在命令行工具中输入 tsc –v，按 Enter 键运行，查看 TypeScript 是否安装成功，如果输出 TypeScript 的版本号，则表示安装成功，如图 1-3 所示；如果无法识别，则表示安装失败。

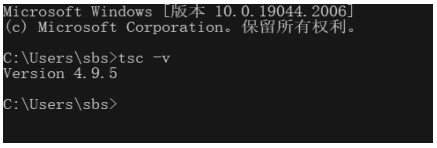

图 1-3　TypeScript 安装成功

1.3.1 Node.js 的安装

Node.js 于 2009 年发布，是一套 JavaScript 的运行环境，使 JavaScript 可以脱离浏览器运行。Node.js 主要用于编写一些服务器端的网络应用。Node.js 与 PHP、Python 等语言最大的不同在于 Node.js 是非阻塞的，可以多条命令同时运行。Node.js 适用于 Windows、Linux、macOS 等操作系统。Node.js 相当于运行在服务器端的 JavaScript，它的出现使 JavaScript 成为与 Python、PHP、Ruby 等服务端语言平起平坐的脚本语言。Node.js 的具体安装步骤如下：

（1）打开浏览器，在浏览器中输入网址 *https://nodejs.cn/download/*，打开 Node.js 中文网，如图 1-4 所示。

图 1-4　Node.js 的搜索

（2）根据操作系统下载对应的 Node.js 的安装包，这里下载的是 Windows 64 位的安装包，如图 1-5 所示。

图 1-5　Node.js 的下载

（3）开始安装。

①双击下载好的安装包，开始安装，如图 1-6 所示。

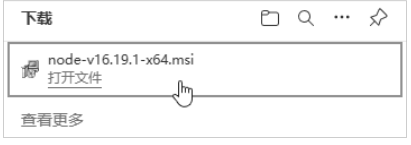

图 1-6　Node.js 的安装包

②打开安装界面，如图 1-7 所示，单击 Next 按钮，进入下一步。

③如图 1-8 所示，勾选接受协议，单击 Next 按钮，进入下一步。

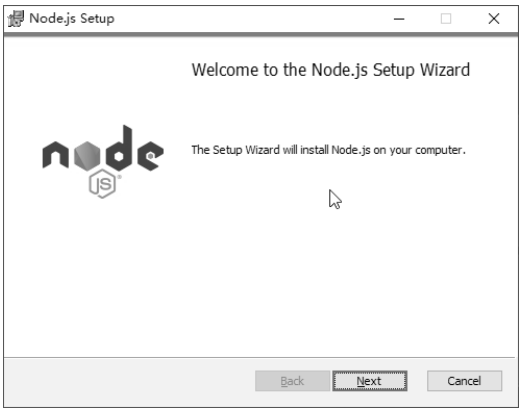

图 1-7　Node.js 的安装（1）

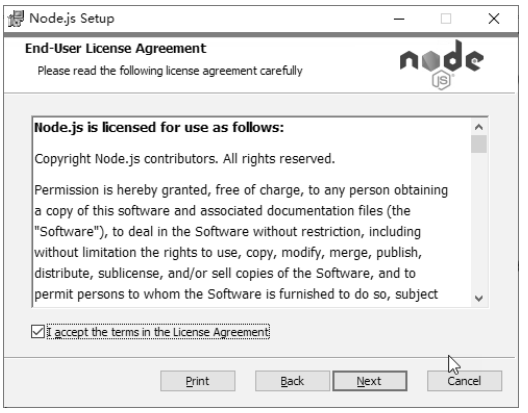

图 1-8　Node.js 的安装（2）

④选择安装目录，默认为 C:/Program Files\node.js\，如图 1-9 所示，选择完成后单击 Next 按钮，进入下一步。

⑤选择需要的安装模式（默认即可，也可根据需求选择），如图 1-10 所示，选择完成后单击 Next 按钮，进入下一步。

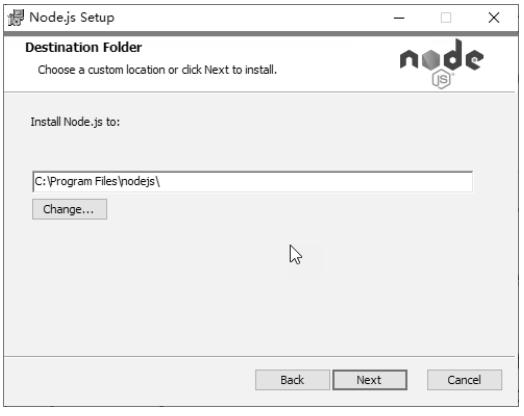

图 1-9　Node.js 的安装（3）

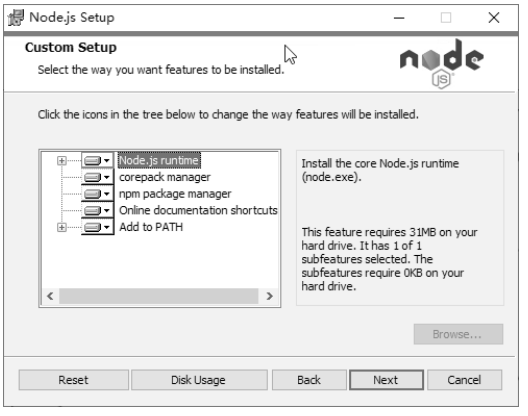

图 1-10　Node.js 的安装（4）

⑥选择安装工具（可根据个人需求选择），如图 1-11 所示，选择完成后单击 Next 按钮，进入下一步。

⑦单击 Install 按钮，开始安装，如图 1-12 所示。

⑧检查 Node.js 是否安装成功，在命令行工具中输入 node -v，按 Enter 键运行，如果返回 Node.js 的版本号，即表示安装成功，如图 1-13 所示；如果无法识别，则表示安装失败。

⑨检查 npm 是否安装成功，在命令行工具中输入 npm -v，按 Enter 键运行，如果返回 npm 的版本号即表示安装成功，如图 1-14 所示；如果无法识别，则表示安装失败（安装 Node.js 时会自动安装 npm）。

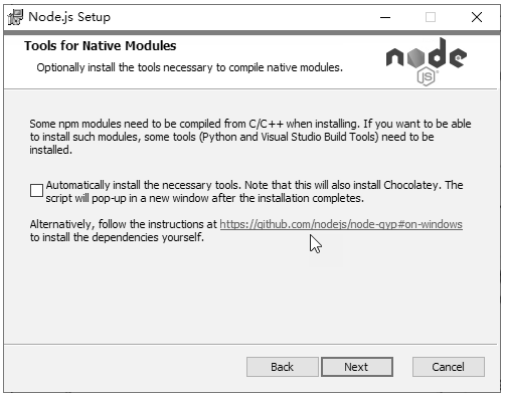

图 1-11　Node.js 的安装（5）

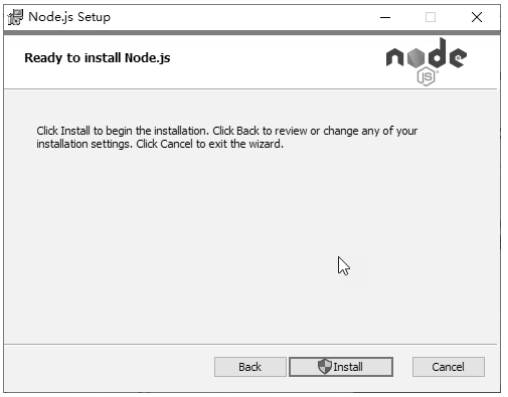

图 1-12　Node.js 的安装（6）

图 1-13　检查 Node.js 的安装

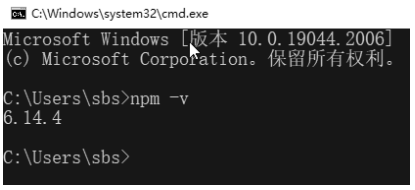

图 1-14　检查 npm 的安装

1.3.2　Visual Studio Code 的安装

Visual Studio Code（简称 VS Code）是 Microsoft 公司于 2015 年发布的一款免费的跨平台集成开发环境，可以运行于 Windows、Linux、Mac OS 等操作系统上。该款编辑器具有热键绑定、语法高亮、括号匹配等特性，同时匹配了丰富的组合键如 Ctrl+K+S 为打开快捷键窗口，Ctrl+K+F 为放大视图等。想要使用这款便捷的编程软件，首先需要将其安装到操作系统中，具体安装步骤如下：

（1）在浏览器中输入网址 *https:code.visualstudio.com/*，打开 Visual Studio Code 的官网，如图 1-15 所示。

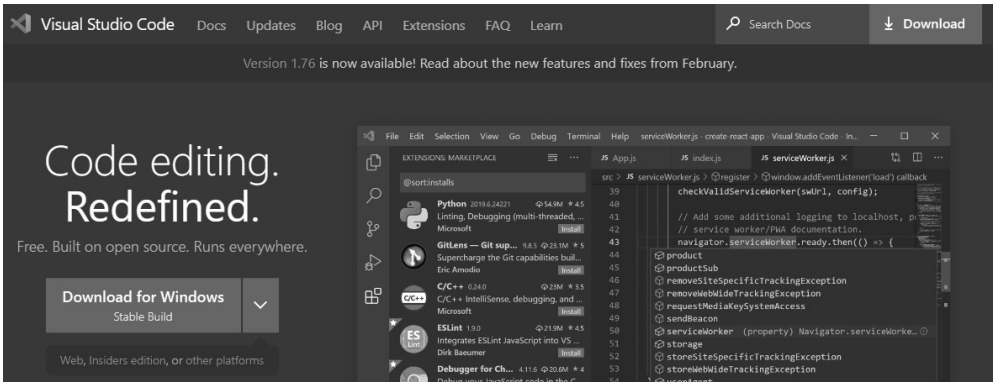

图 1-15　Visual Studio Code 的搜索

（2）根据操作系统下载对应的 Visual Studio Code 安装包，这里下载的是 Windows 系统的安装包，如图 1-16 所示。

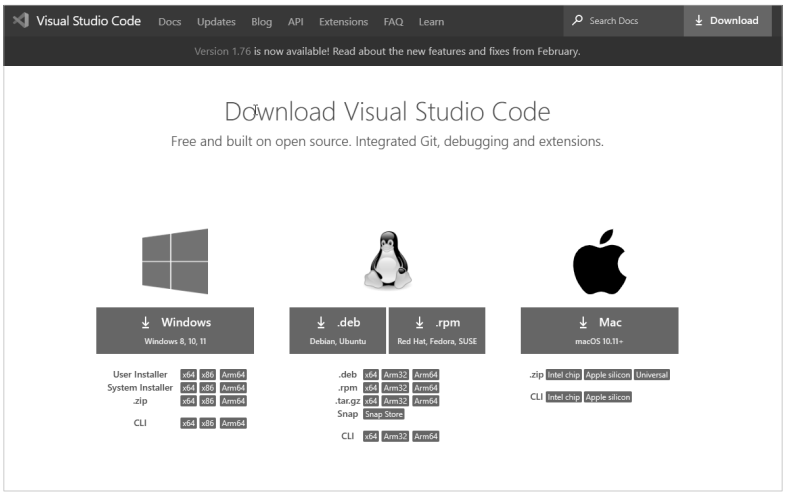

图 1-16　Visual Studio Code 的下载

（3）开始安装。

①双击下载好的安装包，进行安装，如图 1-17 所示。

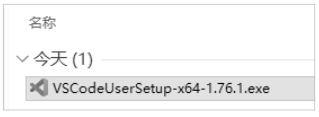

图 1-17　Visual Studio Code 的安装包

②打开安装界面，选择［我同意此协议］单选按钮，单击“下一步”按钮，如图 1-18 所示。

③选择安装路径（建议安装到 D 盘），选择完成后单击“下一步”按钮，如图 1-19 所示。

图 1-18　Visual Studio Code 的安装流程（1）

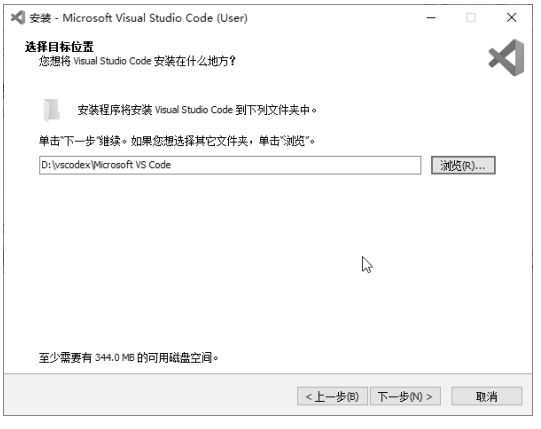

图 1-19　Visual Studio Code 的安装流程（2）

④选择开始菜单文件夹（默认即可，也可以选择不创建开始文件夹），选择完成后单击“下一步”按钮，如图 1-20 所示。

⑤选择附加任务（可根据个人需求选择是否勾选），选择完成后单击“下一步”按钮，如图 1-21 所示。

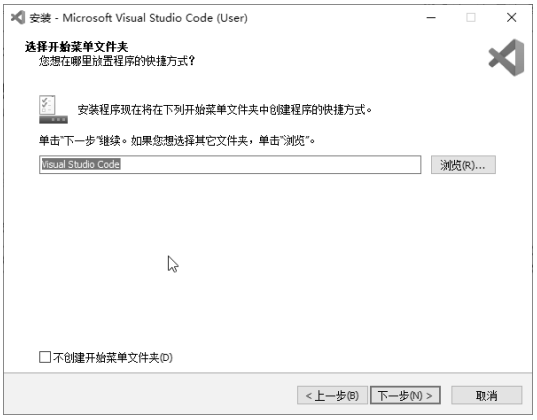

图 1-20　Visual Studio Code 的安装流程（3）

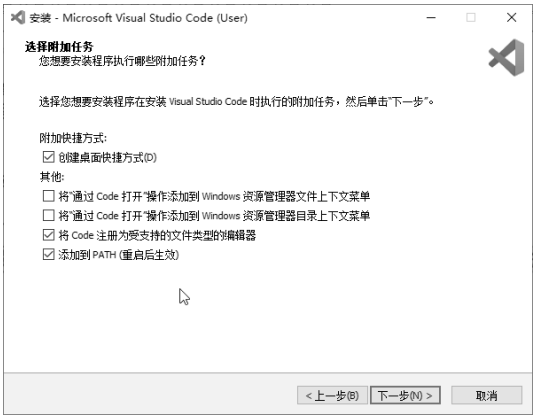

图 1-21　Visual Studio Code 的安装流程（4）

⑥配置完成后单击“安装”按钮，如需修改之前的配置，可以单击“上一步”按钮重新配置，如图 1-22 所示。

⑦单击“安装”按钮，如图 1-23 所示。

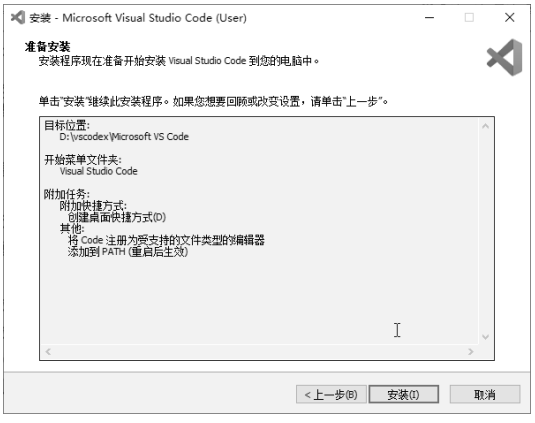

图 1-22　Visual Studio Code 的安装流程（5）

图 1-23　Visual Studio Code 的安装流程（6）

⑧启动 Visual Studio Code，如图 1-24 所示。

⑨将 Visual Studio Code 切换为中文模式（根据个人需求选择是否切换），打开 Visual Studio Code 在扩展中搜索 chinese 中文插件，单击 Install 按钮开始安装，安装成功后重新启动即可切换为中文模式，如图 1-25 所示。

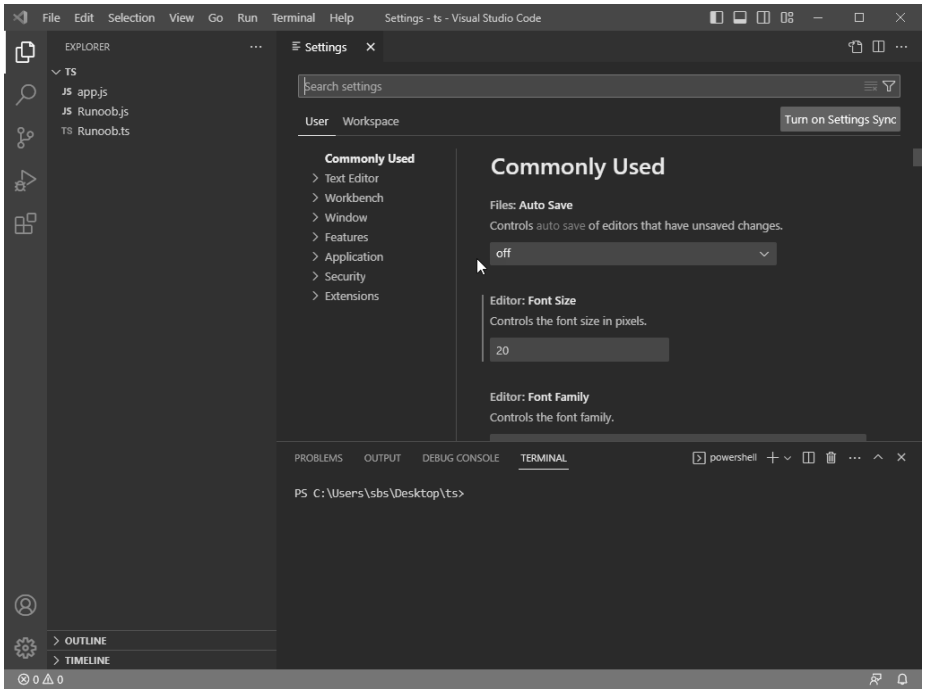

图 1-24　Visual Studio Code 的安装流程（7）

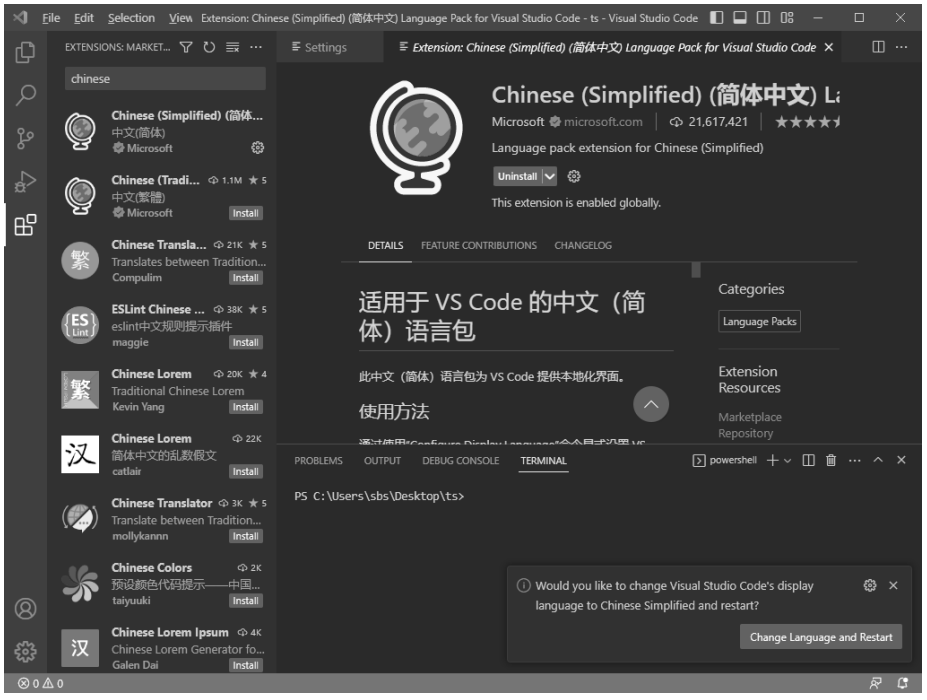

图 1-25　Visual Studio Code 的安装流程（8）

1.4 第一个 TypeScript 程序

想要快速掌握一门编程语言的最快方式就是使用它来编写代码，接下来将使用 Visual Studio Code 编写一个简单的 TypeScript 实例。

【例 1-1】编写一个 TypeScript，在控制台输出“Hello TypeScript！”。

步骤 1：新建一个文件夹并命名为 TypeScript，然后在 Visual Studio Code 中选择“文件”→“打开文件夹”命令，打开新建的 TypeScript 文件夹，如图 1-26 所示。

步骤 2：在 TypeScript 文件夹下新建文件 HelloTypeScript.ts，如图 1-27 所示。

图 1-26　打开 TypeScript 文件夹

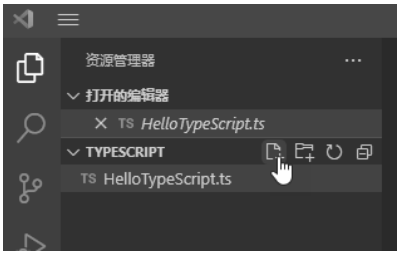

图 1-27　新建 HelloTypeScript.ts 文件

步骤 3：在 HelloTypeScript.ts 中输入代码。

例 1-1　第一个 TypeScript 程序

```
// 声明一个变量 hello 并赋值为 "Hello TypeScript!"
const hello : string = "Hello TypeScript!"
// 在控制台中打印
console.log(hello)
```

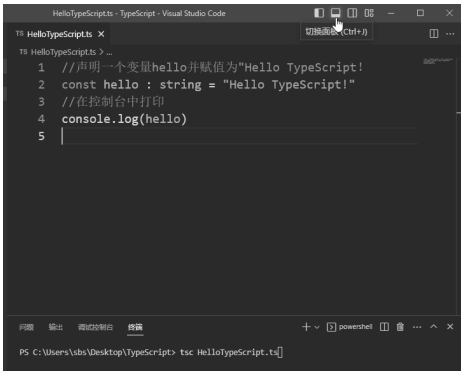

图 1-28　打开 Visual Studio Code 的终端

步骤 4：单击切换面板，也可以使用快捷键 Ctrl+J，打开 Visual Studio Code 的终端，如图 1-28 所示。

步骤 5：在终端中输入命令 tsc HelloTypeScript.ts，按 Enter 键运行，如图 1-29 所示。

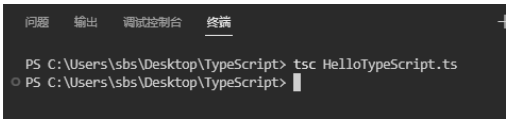

图 1-29　输入命令 tsc HelloTypeScript.ts

步骤 6：此时会在 TypeScript 文件夹下生成一个名称为 HelloTypeScript.js 的文件，如图 1-30 所示。

图 1-30　生成 HelloTypeScript.js 文件

步骤 7：在终端中输入命令 node HelloTypeScript.js，按 Enter 键运行，返回“Hello TypeScript！”，说明程序执行成功，如图 1-31 所示。

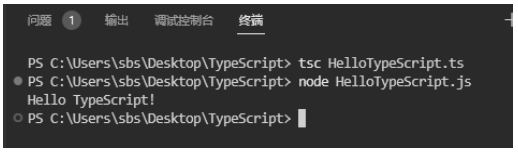

图 1-31　执行结果

1.5　就业面试技巧与解析

本章主要讲解了 TypeScript 与 JavaScript 的区别、TypeScript 相对于 JavaScript 的优势，node.js 和 Visual Studio Code 的安装，以及第一个 TypeScript 程序的开发。通过上面的讲解，相信大家都已熟练掌握。这些知识在面试中常以下面的形式体现。

1.5.1　面试技巧与解析（一）

面试官：TypeScript 与 JavaScript 的关系是怎样的？

应聘者：TypeScript 是 JavaScript 的超集。TypeScript 是在 JavaScript 的基础上添加类型系统而形成的一门强类型语言，且 TypeScript 是完全兼容 JavaScript 的。

1.5.2　面试技巧与解析（二）

面试官：相比 JavaScript，TypeScript 为前端开发带来了哪些好处？

应聘者：TypeScript 是一门强类型语言，可以使代码中的类型错误更早暴露，相比 JavaScript 提高了代码的可读性和可维护性，且 TypeScript 支持分模块开发，在开发大型项目时可以更好地进行分工协作。

第 2 章

TypeScript基本语法

本章概述

每种语言都定义了自己的语法规范，TypeScript 语言也是一样的。一个 TypeScript 程序主要由变量、函数、语法和表达式、模块、注释组成。TypeScript 是一门面向对象的编程语言，因此它也遵循面向对象的原则。本章将讲解 TypeScript 的基本语法。

知识导读

本章要点（已掌握的在方框中打钩）

- ☐ TypeScript 编程术语。
- ☐ TypeScript 基本语法。
- ☐ TypeScript 数据类型。
- ☐ TypeScript 运算符。
- ☐ TypeScript 控制语句。

2.1 TypeScript 编程术语

作为一门编程语言，需要满足一套语言描述规则，同大多数语言一样，TypeScript 也有在编程过程中制定的编程规范。在 TypeScript 中常用的编程术语有标识符、数据类型、变量和参数、函数和方法、表达式和语句等。

1. 标识符

标识符是对变量、方法、数组、类赋予名称，在声明标识符时要遵循如下规则：

（1）标识符不能以数字开头。

（2）标识符不能使用关键字。

（3）标识符中不能包含空格。

（4）标识符中除 _ 和 $ 外，不能使用其他符号。

TypeScript 标识符示例如表 2-1 所示。

表 2-1　TypeScript 标识符示例

有　效	无　效
identifier	0identifier
oneAny	any
$identifier	#identifier
_identifier	Ide ntifier

2. 数据类型

TypeScript 是一门强类型的编程语言，可以声明具有数据类型的变量。TypeScript 的数据类型如表 2-2 所示。

表 2-2　TypeScript 的数据类型

关　键　字	数据类型	是否为原始数据类型
boolean	布尔类型	是
number	数字类型	是
string	字符串类型	是
array	数组类型	否
无	元组类型	否
enum	枚举类型	否
any	任意类型	否
void	void	否
undefined	undefined	是
null	null	是
never	never	否

提示：字符串类型的 string 首字母是小写的。

3. 变量和参数

参数是针对函数调用而言的，是函数的补充成分，分为形参和实参，一般用于方法内的传递参数。变量是在程序运行中允许改变的量。每个变量都有一个唯一的名字，又称标识符。变量的值可以在程序的执行过程中改变，因此，在程序中大部分使用变量来存储操作数据。在 TypeScript 中声明一个变量时需要给这个变量指定类型。

4. 函数和方法

函数是以 function 开头的一段代码，可以通过函数的名称调用。函数可以接收一些参数

并返回，也可以没有返回值。方法是一种特殊的函数，通过对象来调用 TypeScript 函数。TypeScript 支持 JavaScript 所有的函数语法。相比 JavaScript，TypeScript 新增了数据类型和函数重载等新特性。函数是程序中必不可少的一部分，它使我们可以重用一些代码。使用函数可以大大提高代码的重用性。

例 2-1　TypeScript 的函数示例

```
//声明一个函数
function example(a:number,b:number):number{
return a + b;//返回 a 和 b 的和
}
//调用函数并传参
let c:number = example(3,4);
//打印结果
console.log("运行结果为："+ c);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-2　TypeScript 的函数示例编译的 JavaScript 代码

```
//声明一个函数
function example(a, b) {
    return a + b; //返回 a 和 b 的和
}
//调用函数并传参
var c = example(3, 4);
//打印结果
console.log("运行结果为："+ c);
```

函数运行结果如图 2-1 所示。

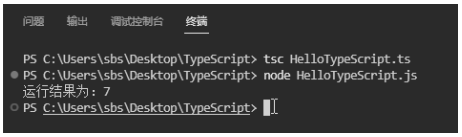

图 2-1　函数运行结果

5. 表达式和语句

表达式和语句的区分对于编码来说是非常重要的，每个 TypeScript 程序都是由一个或多个语句组成的。每条语句都是一条指令，而表达式从本质上来讲是生成的一段代码，通常放在语句的插槽内，成为语句的一部分。

（1）表达式：TypeScript 的常用表达式有箭头表达式、数组析构表达式等。

（2）语句：TypeScript 的常用语句有判断语句（if、else）、循环语句（for）等。

2.2　TypeScript 基本语法

TypeScript 基本语法如下。

1. 注释语法

TypeScript 的注释语法类似于 Java、JavaScript 等语言，也支持单行注释和多行注释，

注释中的字符会被 TypeScript 的编译器忽略。

例 2-3　TypeScript 的注释示例

```
 /**
 * 第一个 TypeScript 程序
 * 这是一个多行注释的实例
 */
var msg:string = "Hello TypeScript";
// 在控制台打印，这是一个单行注释的实例
/* 这也是一个单行注释的实例 */
console.log(msg);
```

2. 区分大小写

TypeScript 对字母大小写敏感，因此，在声明和使用一些关键字、函数、变量时要严格区分字母的大小写。例如，name 和 Name 就是两个不同的变量。在 TypeScript 中一般使用小写字母定义参数的类型。

例 2-4　TypeScript 区分大小写的示例

```
// 声明变量（正确）
var msg:number = 123;
// 声明变量（错误）
var msg1:Number = 123;
// 在控制台打印
console.log(msg);
```

3. 保留字

TypeScript 的保留字是为以后版本升级预留的关键字，不可以作为标识符使用。TypeScript 的保留字如表 2-3 所示。

表 2-3　TypeScript 的保留字

break	as	catch	switch	package
case	if	throw	else	implements
var	number	string	get	interface
module	type	instanceof	typeof	function
public	private	enum	export	do
finally	for	while	void	try
null	super	this	new	yield
in	return	true	false	const
any	extends	static	let	continue

4. 语句用“;”分隔

在 TypeScript 程序中，每条指令都是一个语句，和 JavaScript 代码相同，分号在 TypeScript 代码中是可选的，可以使用或不使用。但是考虑到代码的可读性和浏览器的兼容性，建议在

每条语句结束时都添加分号。

例 2-5　TypeScript 语句中分号使用的示例

```
//这条语句是合法的
var msg:number = 123;
//这条语句也是合法的
var msg1:number = 123
//在控制台打印
console.log(msg);
console.log(msg1);
```

5. 文件扩展名为 .ts

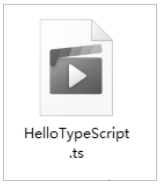

图 2-2　TypeScript 的扩展名

文件名可以分为两部分，一部分为文件名，另一部分为定义文件类型的扩展名。在系统中可以通过文件的扩展名来区分文件的种类和用途，如 JavaScript 文件的扩展名为 .js，TypeScript 的扩展名为 .ts，如图 2-2 所示。

6. 变量声明

变量是在程序运行中允许改变的量，在使用变量之前需要先声明变量。变量的名称可以包含字母和数字，但是不能以数字开头，除了可以包含下画线和美元符号，不能包含其他特殊字符。

（1）变量的声明方式。可以通过 var、let、const 来声明变量，其中 let 和 const 能够声明具有块级作用域的变量，变量的声明方式有 4 种，如表 2-4 所示。

表 2-4　变量的声明方式

声明方式	例　子	说　明
var [变量名] : [类型] = 值;	var msg: msg = "Hello";	类型为string，值为Hello
var [变量名] : [类型];	var msg:string;	类型为string，值为undefined
var [变量名] = 值;	var msg = "Hello";	类型为任意类型，值为Hello
var [变量名];	var msg;	类型为任意类型，值为undefined

例 2-6　声明变量的示例代码

```
//声明一个变量 userName，类型为 string，值为小明
var userName: string = '小明';
//声明一个变量 score1，类型为 number，值为 80
var score1: number = 80;
var score2: number = 90;
var score3: number = 100;
//计算 score1、score2、score3 的和并赋值给 score
var score: number = score1 + score2 + score3;
console.log("姓名："+ userName);
console.log("总成绩："+ score);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-7　声明变量示例代码编译的 JavaScript 代码

```
// 声明一个变量 userName，类型为 string，值为小明
var userName = "小明";
// 声明一个变量 score1，类型为 number，值为 80
var score1 = 80;
var score2 = 90;
var score3 = 100;
// 计算 score1、score2、score3 的和并赋值给 score
var score = score1 + score2 + score3;
console.log("姓名：" + userName);
console.log("总成绩：" + score);
```

声明变量示例代码的运行结果如图 2-3 所示。

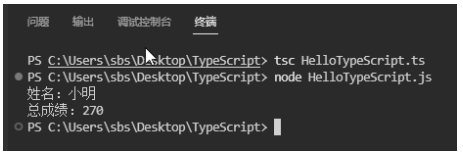

图 2-3　声明变量示例代码的运行结果

（2）变量的作用域。变量的作用域由定义变量的位置决定，而在程序中一个变量的可使用范围由它的作用域决定。TypeScript 的作用域可分为全局作用域（定义在程序结构的外部，可以在任意位置使用）、类作用域（在类中声明，但是在类方法的外面，通过类的对象访问，当类变量为静态变量时，可以通过类名直接访问）和局部作用域（只能在声明它的代码块中使用，例如，在一个方法中声明，那么这个变量只能在这个方法中使用）。

例 2-8　变量作用域的示例代码

```
// 全局变量
var global_variable: number = 12;
class classVariable {
    // 实例变量
    example_variable: number = 13;
    // 静态变量
    static static_variable: number = 14;
    storeNum():void {
        // 局部变量
       var local_variable: number = 15;
       console.log("我是局部变量：" + local_variable);
    }
 }
 console.log("我是全局变量：" + global_variable);
 var obj = new classVariable();
 console.log("我是实例变量：" + obj.example_variable);
 console.log("我是静态变量：" + classVariable.static_variable);
 // 调用 storeNum 方法
 obj.storeNum();
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-9　变量作用域示例代码编译的 JavaScript 代码

```
// 全局变量
var global_variable = 12;
var classVariable = /** @class */ (function () {
    function classVariable() {
        // 实例变量
        this.example_variable = 13;
    }
    classVariable.prototype.storeNum = function () {
        // 局部变量
        var local_variable = 15;
        console.log("我是局部变量：" + local_variable);
    };
    // 静态变量
    classVariable.static_variable = 14;
    return classVariable;
}());
console.log("我是全局变量：" + global_variable);
var obj = new classVariable();
console.log("我是实例变量：" + obj.example_variable);
console.log("我是静态变量：" + classVariable.static_variable);
// 调用 storeNum 方法
obj.storeNum();
```

变量作用域示例代码的运行结果如图 2-4 所示。

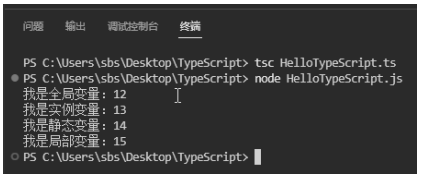

图 2-4　变量作用域示例代码的运行结果

7. 异常处理

TypeScript 通过关键字 throw 来抛出异常，然后通过 try/catch 块来捕获异常。TypeScript 中常出现的异常类型如表 2-5 所示。

表 2-5　异常类型

异常类型	说　明
RangeError	字符类型的数据超出最大范围
ReferenceError	引用的数据无效
SyntaxError	解析无效
TypeError	变量或参数不是有效类型
URLError	传入无效参数至encodeURI()和decodeURI()

2.3　TypeScript 数据类型

为了使代码更加规范，提高代码的可读性，TypeScript 主要提供了以下几种数据类型：数字类型、字符串类型、布尔类型、未定义类型和空类型、枚举类型、任意值类型、数组类型、元组类型、never 类型、Symbol 类型，以及字面量类型、联合类型、类型断言。

2.3.1　数字类型

在 TypeScript 中，数字类型表示一个数字，类型注解为 number，这个数字可以是正数、负数、整数、小数等。在 TypeScript 中只有一种数字类型，用双精度 64 位的浮点数表示，其中符号占 1 位，指数占 11 位，小数占 52 位，且支持二进制、八进制、十进制、十六进制，如 5、−2、3.2 等。

例 2-10　数字类型的示例代码

```
// 数字类型
var num:number = 123;
num = 222;                // 正确
num = 12.3;               // 正确
num = -123;               // 正确
//num = '123'             // 错误
```

2.3.2　字符串类型

在 TypeScript 中字符串类型用于表示文本信息。一个字符串是由零个或多个字符拼接而成的，字符可以是文字、数字、字母、标点等，如 ' 小明 '、'name'、'name1' 等。字符串需要放在单引号、双引号，或者反引号内（定义内嵌表达式或多行文本时使用），定义常用文本时建议使用单引号。

例 2-11　字符串类型的示例代码

```
// 字符串类型
var str:string = 'Hello';
str = 'Hello1';           // 正确
str = "Hello2";           // 正确
//str = 1;                // 错误
//str = true;             // 错误
```

1. 字符串拼接

如果将一个加号用于数字类型，则是将数字类型的值相加。如果将加号用于字符串类型，则是将字符串进行拼接，将加号后面的字符串拼接到加号之前的字符串之后，形成一个新的字符串。

例 2-12　字符串拼接的示例代码

```
//字符串拼接
var str:string = 'Hello';
str = str + 'TypeScript';
console.log("运行结果为:" + str);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-13　字符串拼接示例代码编译的 JavaScript 代码

```
//字符串拼接
var str = 'Hello';
str = str + 'TypeScript';
console.log("运行结果为:" + str);
```

运行此代码得到如图 2-5 所示的结果。

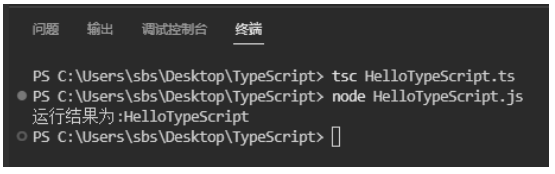

图 2-5　字符串拼接示例代码的运行结果

2. 获取字符串的长度

每个字符串都有长度，也就是它包含的字符的数量，可以通过 length 属性获取字符串的长度。

例 2-14　获取字符串长度的示例代码

```
//获取字符串的长度
var str:string = 'Hello';
console.log("字符串的长度为: " + str.length);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-15　获取字符串长度示例代码编译的 JavaScript 代码

```
//获取字符串的长度
var str = 'Hello';
console.log("字符串的长度为: " + str.length);
```

运行此代码得到如图 2-6 所示的结果。

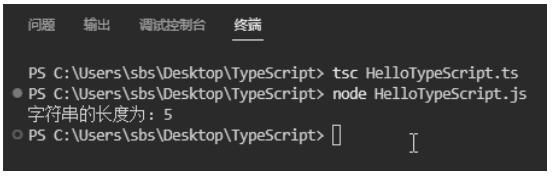

图 2-6　获取字符串长度示例代码的运行结果

3. 大小写转换

在 TypeScript 中可以通过 toLowerCase()、str.toUpperCase()、str.toLocaleLowerCase()、str.toLocaleUpperCase() 等方法转换字符串中字母的大小写。

例 2-16　大小写转换的示例代码

```
// 转换字符串中字母的大小写
var str:string = 'Hello';
console.log("正常显示为：" + str);
console.log("以小写方式展示：" + str.toLowerCase());
console.log("以大写方式展示：" + str.toUpperCase());
console.log("以本地方式将字符串转化为小写：" + str.toLocaleLowerCase());
console.log("以本地方式将字符串转化为大写：" + str.toLocaleUpperCase());
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-17　大小写转换示例代码编译的 JavaScript 代码

```
// 修改字符串中字母的大小写
var str = 'Hello';
console.log("正常显示为：" + str);
console.log("以小写方式展示：" + str.toLowerCase());
console.log("以大写方式展示：" + str.toUpperCase());
console.log("以本地方式将字符串转化为小写：" + str.toLocaleLowerCase());
console.log("以本地方式将字符串转化为大写：" + str.toLocaleUpperCase());
```

运行此代码得到如图 2-7 所示的结果。

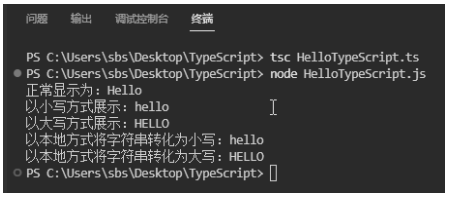

图 2-7　大小写转换示例代码的运行结果

2.3.3　布尔类型

在 TypeScript 中，布尔类型用来表示真假，类型注解为 boolean。在数据类型中数值类型和字符转类型的值可能会有无穷个，但是布尔类型的值只有两个，这两个值分别为 true 和 false，其中 true 表示真，false 表示假。

例 2-18　布尔类型的示例代码

```
// 布尔类型
var bool:boolean = true;
bool = false;              // 正确
//bool = 1;                // 错误
//num = '123';             // 错误
```

2.3.4 未定义类型和空类型

在 TypeScript 中，未定义类型用来表示还没有赋值的变量或赋予了一个不存在的属性值的变量，它没有值。空用来表示一个变量赋予的值为空，它是有值的，只是值为空。undefined 和 null 都只有一个值，那就是它本身。

例 2-19　未定义类型和空类型的示例代码

```
// 未定义变量
var u: undefined;
// 空变量
var n: null = null;
```

2.3.5 枚举类型

在 TypeScript 中，枚举类型用于数值集合，类型注解为 enum，它是对 JavaScript 的标准数据类型的补充，可以为一组数值赋予一个新的名字。TypeScript 中的枚举可以分为数字枚举、字符串枚举和异构枚举。使用枚举可以大大减少代码的编译时间和运行时间，同时可以使代码变得更加整洁清晰。

1. 数字枚举

使用关键字 enum 定义一个枚举类型，其中的字段以逗号分隔。

例 2-20　枚举类型的示例代码

```
// 数字枚举
enum num { One, Two, Three }
// 打印结果
console.log(num)
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-21　枚举类型示例代码编译的 JavaScript 代码

```
// 数字枚举
var num;
(function (num) {
    num[num["One"] = 0] = "One";
    num[num["Two"] = 1] = "Two";
    num[num["Three"] = 2] = "Three";
})(num || (num = {}));
// 打印结果
console.log(num);
```

运行此代码得到如图 2-8 所示的结果。

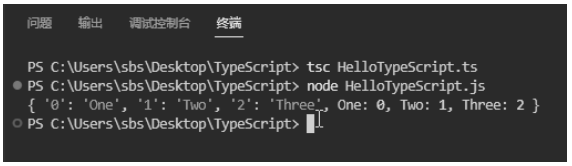

图 2-8　枚举类型示例代码的运行结果

说明：第一个枚举成员的值默认为 0，之后枚举成员的值依次加 1。

2. 自定义枚举

添加枚举中的成员时可以为成员赋值，当对成员赋值后，其余成员会在它的基础上依次加 1。数字枚举中的值和名称是双向绑定的，因此，可以根据成员变量的值获取成员变量的名称。

例 2-22　自定义枚举类型的示例代码

```
// 自定义枚举
enum num { One = 3, Two, Three }
// 打印结果
console.log(num)
 // 获取值为 3 的成员名称
let onerName: string = num[3];
console.log(onerName)
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-23　自定义枚举类型示例代码编译的 JavaScript 代码

```
// 自定义枚举
var num;
(function (num) {
    num[num["One"] = 3] = "One";
    num[num["Two"] = 4] = "Two";
    num[num["Three"] = 5] = "Three";
})(num || (num = {}));
// 打印结果
console.log(num);
// 获取值为 3 的成员名称
var onerName = num[3];
console.log(onerName);
```

运行此代码得到如图 2-9 所示的结果。

图 2-9　自定义枚举类型示例代码的运行结果

3. 常量枚举

通过在关键字 enum 之前添加修饰符 const，const 修饰符会在编译阶段被移除，使用常量枚举可以大大提高数值枚举的性能。

例 2-24　常量枚举类型的示例代码

```
//常量枚举
const enum num { One, Two, Three }
```

4. 计算枚举

数值枚举中成员的值可以是常量，也可以是表达式，当成员的值为表达式时，通过计算表达式所得到的结果，对枚举中的成员赋初始值。

例 2-25　计算枚举类型的示例代码

```
//计算枚举
enum num { One = getNum(1), Two = One * 2}
function getNum(n: number): number {
     return 3 * n
  }
//打印结果
console.log("运行结果为:" + num.One);
console.log("运行结果为:" + num.Two);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-26　计算枚举类型示例代码编译的 JavaScript 代码

```
//计算枚举
var num;
(function (num) {
    num[num["One"] = getNum(1)] = "One";
    num[num["Two"] = num.One * 2] = "Two";
})(num || (num = {}));
function getNum(n) {
    return 3 * n;
}
//打印结果
console.log("运行结果为:" + num.One);
console.log("运行结果为:" + num.Two);
```

运行此代码得到如图 2-10 所示的结果。

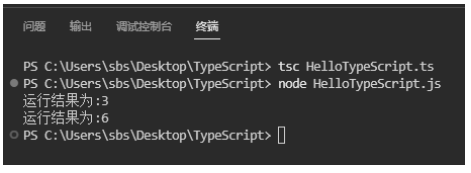

图 2-10　计算枚举类型示例代码的运行结果

5. 字符串枚举

字符串枚举和数字枚举的最大不同在于，字符串枚举无法双向绑定，不可以通过枚举成员的值获取枚举成员的名称。

例 2-27　字符串枚举类型的示例代码

```
//字符串枚举
enum num { One = '小明', Two = '小红' }
//打印结果
console.log(num);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-28　字符串枚举类型示例代码编译的 JavaScript 代码

```
//字符串枚举
var num;
(function (num) {
    num["One"] = "\u5C0F\u660E";
    num["Two"] = "\u5C0F\u7EA2";
})(num || (num = {}));
//打印结果
console.log(num);
```

运行此代码得到如图 2-11 所示的结果。

图 2-11　字符串枚举类型示例代码的运行结果

6. 异构枚举

当一个枚举中的成员类别既有数字类型又有字符串类型时，称为异构枚举。

例 2-29　异构枚举类型的示例代码

```
//异构枚举
enum num { One = '小明', Two = 123 }
//打印结果
console.log(num)
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-30　异构枚举类型示例代码编译的 JavaScript 代码

```
//异构枚举
var num;
(function (num) {
    num["One"] = "\u5C0F\u660E";
    num[num["Two"] = 123] = "Two";
```

```
})(num || (num = {}));
// 打印结果
console.log(num);
```

运行此代码得到如图 2-12 所示的结果。

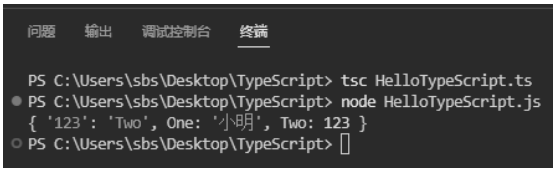

图 2-12　异构枚举类型示例代码的运行结果

2.3.6　任意值类型

在 TypeScript 中，任意值类型是在一些不明确类型的变量中使用的，其类型注解为 any。any 类型通常用于变量的值是动态改变时、改写已有的代码时和定义需要存储各种类型的数组时。

例 2-31　任意值类型的示例代码

```
// 任意值类型
let a: any = 1;                                          // 数字类型
a = 'Hello TypeScript';                                  // 字符串类型
a = true;                                                // 布尔类型
let arrayList: any[] = [1, true, 'hello'];               // 数组
```

2.3.7　数组类型

在 TypeScript 中，数组类型用来存储多个数据的集合，可以将一些无序的数据有序地排列起来。数组中的元素可以是任意类型的值，想要获取数组中的值时，可以通过索引访问，索引值从 0 开始，当访问的索引值不存在时，返回的值为 undefined。数组长度用来表示一个数组中可以存放多少个元素，可以通过 undefined 属性获取数组的长度。

在 TypeScript 中，声明数组类型的方式有两种：一种是以字面量的方式声明，另一种是以泛型的方式声明。

例 2-32　数组类型的示例代码

```
// 字面量方式声明数组
let myArr1: string[] = ['hello', 'TypeScript'];
let myArr2: number[] = [1,2];
// 泛型方式声明数组
let myArr3: Array<string | number> = ['hello', 1];
console.log("运行结果为：" + myArr1);
```

```
console.log("运行结果为：" + myArr2);
console.log("运行结果为：" + myArr3);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-33　数组类型示例代码编译的 JavaScript 代码

```
// 字面量方式声明数组
var myArr1 = ['hello', 'TypeScript'];
var myArr2 = [1, 2];
//泛型方式声明数组
var myArr3 = ['hello', 1];
console.log("运行结果为：" + myArr1);
console.log("运行结果为：" + myArr2);
console.log("运行结果为：" + myArr3);
```

运行此代码得到如图 2-13 所示的结果。

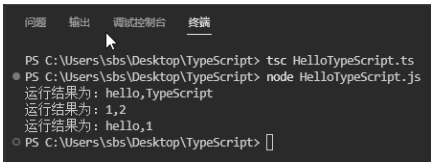

图 2-13　数组类型示例代码的运行结果

泛型数组：声明泛型数组的方式有两种，一种是使用 Array<> 声明，它修饰的数组内的数据是可以修改的。另一种是使用 ReadonlyArray<> 声明，它修饰的数组内的数据是不可以修改的，但可以将 Array<> 类型的值赋给 ReadonlyArray<> 类型，其中 ReadonlyArray<> 还提供了一种简写的声明方式，通过 readonly Type[] 声明。

例 2-34　泛型数组类型的示例代码

```
//声明泛型数组
let myArr1: Array<string> = ['小明', '小红'];
myArr1[0] = '小刚';
let myArr2: ReadonlyArray<string> = ['白色', '红色'];
//myArr2[0] = '白色';  //编译报错
console.log(myArr1);
console.log(myArr2);
//将 myArr1 的值赋给 myArr2
myArr2 = myArr1;
console.log(myArr2);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-35　泛型数组类型示例代码编译的 JavaScript 代码

```
//声明泛型数组
var myArr1 = ['小明', '小红'];
myArr1[0] = '小刚';
var myArr2 = ['白色', '红色'];
```

```
//myArr2[0] = '白色';  //编译报错
console.log(myArr1);
console.log(myArr2);
//将myArr1的值赋给myArr2
myArr2 = myArr1;
console.log(myArr2);
```

运行此代码得到如图 2-14 所示的结果。

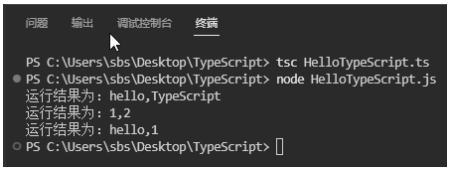

图 2-14　泛型数组类型示例代码的运行结果

2.3.8　元组类型

在 TypeScript 中，元组类型也可以理解为一种 Array 类型，其类型注解为 tuple。元组类型和 Array 类型最大的区别在于，元组类型明确指出了数组中包含多少种类型，包含多少个元素，以及每个元素所在的位置。

例 2-36　元组类型的示例代码

```
//元组类型
let x: [string, number];
x = ['hello', 10];
// x = [10, 'hello']; //编译错误
console.log(x);
console.log("获取第一个元素："+x[0]);
//当访问的索引超过元组长度时，编译错误
//console.log("获取第一个元素："+x[3]);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-37　元组类型示例代码编译的 JavaScript 代码

```
//元组类型
var x;
x = ['hello', 10];
// x = [10, 'hello']; //编译错误
console.log(x);
console.log("获取第一个元素：" + x[0]);
//当访问的索引超过元组长度时，编译错误
//console.log("获取第一个元素："+x[3]);
```

运行此代码得到如图 2-15 所示的结果。

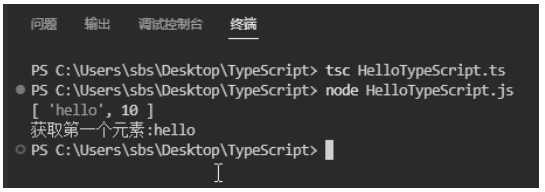

图 2-15　元组类型示例代码的运行结果

可选元素类型：在一个元素类型后面添加问号（?）用来表示当前元素是可选的，可选元素必须在必填元素之后。

例 2-38　元组可选元素类型的示例代码

```
// 元组的可选元素
let x: [string, number?];
x = ['a'];
console.log(x);
x = ['a', 1];
console.log(x);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-39　元组可选元素类型示例代码编译的 JavaScript 代码

```
// 元组的可选元素
var x;
x = ['a'];
console.log(x);
x = ['a', 1];
console.log(x);
```

运行此代码得到如图 2-16 所示的结果。

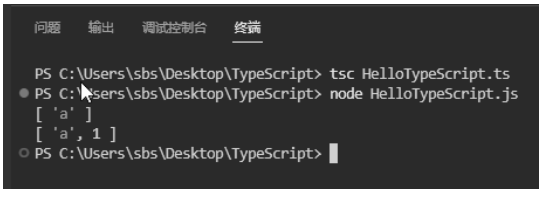

图 2-16　元组可选元素类型示例代码的运行结果

2.3.9　never 类型

在 TypeScript 中，never 类型用来表示一个永远不会有值的类型，通常用于抛出异常和一些没有返回值的函数。

例 2-40　never 类型的示例代码

```
//never 类型示例
```

```
let x: never;
// x = 123; // 编译错误
//编译正确，赋值为异常
x = (()=>{ throw new Error('异常')})();
//编译正确，返回结果为异常
// function error(message: string): never {
//     throw new Error(message);
// }
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-41　never 类型示例代码编译的 JavaScript 代码

```
//never 类型示例
var x;
// x = 123; // 编译错误
//编译正确，赋值为异常
x = (function () { throw new Error('异常'); })();
//编译正确，返回结果为异常
// function error(message: string): never {
//     throw new Error(message);
// }
```

运行此代码得到如图 2-17 所示的结果。

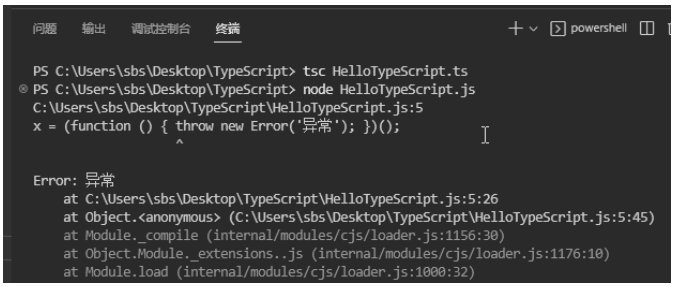

图 2-17　never 类型示例代码的运行结果

2.3.10　Symbol 类型

Symbol 是一种新的原生类型，它的值是通过构造函数来创建的，因为内存地址的指针不同，所以 Symbol 类型的值具有唯一的特性，即使是两个完全相等的参数，Symbol 类型的值也是不相等的，并且只支持 string 和 number 两种类型的参数。

例 2-42　Symbol 类型的示例代码

```
// 创建一个 Symbol 类型的变量
let sym = Symbol()
// 用作对象属性的键
let obj = {
```

```
    [sym]: "hello world"
}
// 获取对象的属性
console.log(obj[sym])
```

提示：Symbol 是在 2015 年之后推出的新类型，之前的 TypeScript 版本无法使用。如果编译成 JavaScript 代码时报错（报错信息如图 2-18 所示），可以使用 tsc xxx.ts --target es6 指令编译。

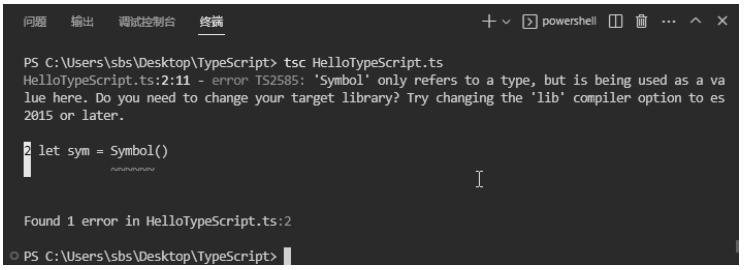

图 2-18　报错信息

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-43　Symbol 类型示例代码编译的 JavaScript 代码

```
// 创建一个 Symbol 类型的变量
let sym = Symbol();
// 用作对象属性的键
let obj = {
    [sym]: "hello world"
};
// 获取对象的属性
console.log(obj[sym]);
```

运行此代码得到如图 2-19 所示的结果。

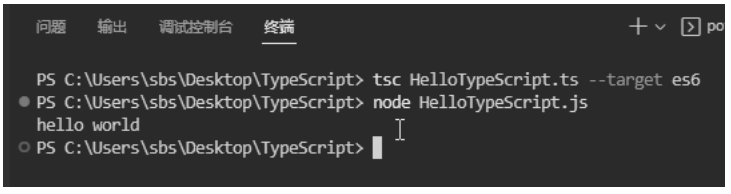

图 2-19　Symbol 类型示例代码的运行结果

当一个对象中有多个 Symbol 定义的属性时，就需要通过遍历来获取了，但是无法通过 for in、Object.keys、getOwnPropertyNames、JSON.stringfy 等方式来获取。

例 2-44　遍历 Symbol 的示例代码

```
let sym1 = Symbol('a')
let sym2 = Symbol('b')
const obj = {
```

```
  [sym1]: 'one',
  [sym2]: 'two',
  a: 1,
}
// 获取 Symbol 属性的数据
console.log(Object.getOwnPropertySymbols(obj));
// 获取全部数据
console.log(Reflect.ownKeys(obj));
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-45　遍历 Symbol 示例代码编译的 JavaScript 代码

```
let sym1 = Symbol('a');
let sym2 = Symbol('b');
const obj = {
    [sym1]: 'one',
    [sym2]: 'two',
    a: 1,
};
// 获取 Symbol 属性的数据
console.log(Object.getOwnPropertySymbols(obj));
// 获取全部数据
console.log(Reflect.ownKeys(obj));
```

运行此代码得到如图 2-20 所示的结果。

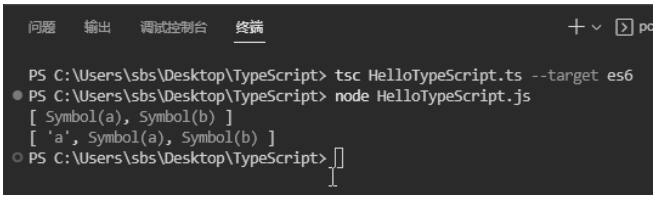

图 2-20　遍历 Symbol 示例代码的运行结果

2.3.11　字面量类型、联合类型、类型断言

在 TypeScript 中，字面量类型就是直接写出一个值，将它赋值给变量，主要分为数字类型的字面量、字符串类型的字面量、布尔类型的字面量、对象类型的字面量和枚举类型的字面量。联合类型主要实现一个变量的值有多个类型的情况，多个类型之间用竖线（|）分隔。类型断言也称类型适配，主要功能是将一个变量的类型断言为另一个类型，通过 as 关键字来声明断言的类型。

1. 字面量类型

通过等号（=）将值赋给变量。

例 2-46　字面量类型的示例代码

```
//数字字面量类型
const num = 1;
//字符串字面量类型
const str = "小明";
//布尔字面量类型
const flag = true;
//对象字面量类型
const obj = { name: "小明", age: 3 };
```

2. 联合类型

当一个值可能有多个类型时，可以通过联合类型实现。

例 2-47　联合类型的示例代码

```
//联合类型示例
let unite: number|string;
unite = 1;
console.log(unite);
unite = 'hello';
console.log(unite);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-48　联合类型示例代码编译的 JavaScript 代码

```
//联合类型示例
var unite;
unite = 1;
console.log(unite);
unite = 'hello';
console.log(unite);
```

运行此代码得到如图 2-21 所示的结果。

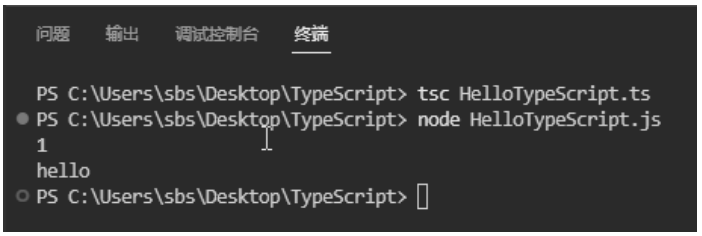

图 2-21　联合类型示例代码的运行结果

3. 类型断言

通过 as 关键字或者 <> 来改变一个变量的类型。

例 2-49　类型断言的示例代码

```
// 类型断言
// 声明一个任意类型的变量
```

```
let x: any = 'hello';
// 将 x 断言为 string 类型，并将字符串转换为大写
let a = (x as string).toUpperCase();
console.log(a);
// 将 x 断言为 string 类型，并将字符串转换为小写
let b = (<string>x).toLowerCase();
console.log(b);
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-50　类型断言示例代码编译的 JavaScript 代码

```
// 类型断言
// 声明一个任意类型的变量
var x = 'hello';
// 将 x 断言为 string 类型，并将字符串转换为大写
var a = x.toUpperCase();
console.log(a);
// 将 x 断言为 string 类型，并将字符串转换为小写
var b = x.toLowerCase();
console.log(b);
```

运行此代码得到如图 2-22 所示的结果。

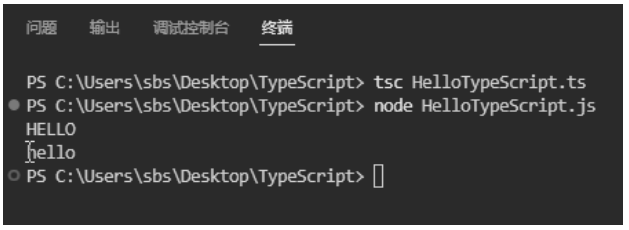

图 2-22　类型断言示例代码的运行结果

2.4　TypeScript 运算符

运算符是程序中必不可少的一部分，它是表达式中用于运算的符号。TypeScript 中的运算符可以分为算术运算符、逻辑运算符、关系运算符、按位运算符、赋值运算符、类型运算符等。

2.4.1　算术运算符

算数运算是比较简单的运算符，也是程序中比较常用的运算符，常应用于数字表达式中。TypeScript 的算数运算符包括加（+）、减（-）、乘（*）、除（/）、求余（%）、自增（++）、自减（--），如表 2-6 所示。

表 2-6　算术运算符应用实例

运　算　符	说　　明	示例（a=10，b=20）
+	加法	a+b等于30
-	减法	b-a等于10
*	乘法	a*b等于200
/	除法	b/a等于2
%	求余	b%a等于0
++	自增	a++等于21
– –	自减	a– –等于19

说明：

（1）自增（++）在前时先递增再赋值，++ 在后时先赋值再递增。

（2）自减（––）在前时先递减再赋值，–– 在后时先赋值再递减。

2.4.2　逻辑运算符

逻辑运算符通常用于将运算的变量连接起来，组成一个逻辑表达式，然后判断这个逻辑表达式是否成立，成立则为 true，不成立则为 false。在 TypeScript 中，逻辑运算符有与（&&）、或（||）、非（！）3 个，如表 2-7 所示。

表 2-7　逻辑运算符应用实例

运　算　符	说　　明	示例(a为true，b为false)
&&	与，当两个操作数都为真时,结果为真	a&&b结果为false
||	或，当两个操作数有一个为真时,结果为真	a||b结果为true
!	非，如果操作数为真,结果为假	!a结果为false

2.4.3　关系运算符

关系运算符又称比较运算符，常用于两个操作数的关系运算，以确定两个操作数之间的关系。在 TypeScript 中，关系运算符有大于（>）、小于（<）、大于或等于（>=）、小于或等于（<=）、等于（==）、不等于（!=）6 个，如表 2-8 所示。

表 2-8　关系运算符应用实例

运　算　符	说　　明	示例（a=10，b=20）
>	大于	a>b结果为false
<	小于	a<b结果为true
>=	大于或等于	a>=b结果为false
<=	小于或等于	a<=b结果为true
==	等于	a==b结果为false
!=	不等于	a!=b结果为true

2.4.4　按位运算符

位运算通常是先将操作数转换为二进制，然后进行运算，最后以十进制输出。位运算的操作数和结果都必须是整数。在 TypeScript 中，位运算符有按位与（&）、按位或（|）、按位异或（^）、按位取反（～）、左移（<<）、右移（>>）、无符号右移（>>>）7 个，如表 2-9 所示。

表 2-9　按位运算符应用实例

运　算　符	说　　明	示例（a=10，b=20）
&	按位与	a&b结果为0
|	按位或	a|b结果为30
^	按位异或	a^b结果为30
～	按位取反	～a结果为–11
<<	左移	a<<b结果为10485760
>>	右移	a>>b结果为0
>>>	无符号右移	a>>>b结果为0

说明：

（1）按位与：两个操作数对应位的值均为 1，则结果为 1，否则结果为 0。

（2）按位或：两个操作数对应位的值中有一个或多个为 1，则结果为 1，否则结果为 0。

（3）按位异或：两个操作数对应位的值互不相同时，则结果为 1，否则结果为 0。

（4）按位非：将操作数转换为二进制后，对每一位二进制数取反。

2.4.5　赋值运算符、类型运算符

赋值运算符通常用于给变量赋值。在 TypeScript 中，赋值运算符有等于（=）、加等于（+=）、减等于（-=）、乘等于（*=）、除等于（/=）5 个。

类型运算符也就是 typeof 运算符，用于判断数据的类型。

例 2-51　类型运算符的示例代码

```
//类型运算符示例
let a: string = "小明";
console.log(typeof(a));
```

在终端中输入 tsc 指令后可以得到 JavaScript 代码。

例 2-52　类型运算符示例代码编译的 JavaScript 代码

```
//类型运算符示例
var a = "小明";
console.log(typeof (a));
```

运行此代码得到如图 2-23 所示的结果。

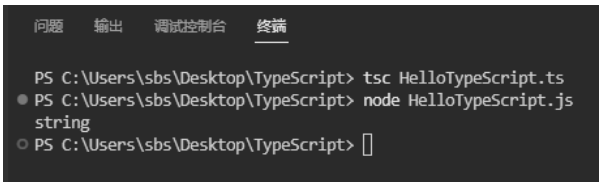

图 2-23　类型运算符示例代码的运行结果

2.5　TypeScript 控制语句

TypeScript 中有多种类型的控制语句，使用这些语句可以对程序的流程进行控制和判断。下面将详细介绍 TypeScript 控制语句中的条件语句、循环语句和跳转语句。

2.5.1　条件语句

在 TypeScript 中，条件语句是一种比较简单的结构语句，包括 if 语句、if…else 语句、if…else…if 语句及 switch…case 语句，下面将详细讲解这些语句的用法，以及各自的特点。

1. if 语句

if 语句是编码中常用的一种判断语句，通过判断条件表达式的值来判断程序的执行流程和顺序。

例 2-53　if 语句的示例代码

```
//if 语句
var num:number = 2
if (num > 0) {
   console.log("num 的值大于 0")
}
```

编译为 JavaScript 代码后运行结果如图 2-24 所示。

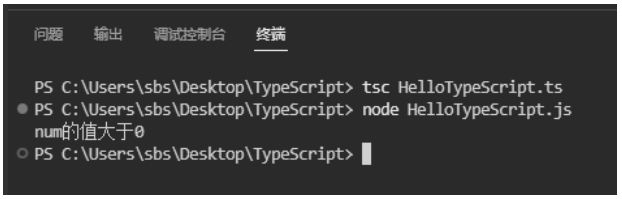

图 2-24　if 语句示例代码的运行结果

2. if…else 语句

当 if 子句中的条件不成立时，执行 else 子句中的语句。

例 2-54　if…else 语句的示例代码

```
//if…else 语句
var num:number = -1
if (num > 0) {
   console.log("num 的值大于 0")
}else{
  console.log("num 的值小于 0")
}
```

编译为 JavaScript 代码后运行结果如图 2-25 所示。

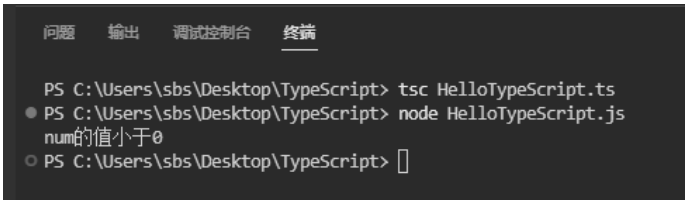

图 2-25　if…else 语句示例代码的运行结果

3. if…else…if 语句

if…else…if 语句称为多条件判断语句，该语句选择多个代码块之一来执行。

例 2-55　if…else…if 语句的示例代码

```
//if...else…if 语句
var num:number = -1
if (num > 1) {
   console.log("num 的值大于 1")
```

```
}else if(num < 1){
  console.log("num 的值小于 1")
}else{
  console.log("num 的值等于 1")
}
```

编译为 JavaScript 代码后运行结果如图 2-26 所示。

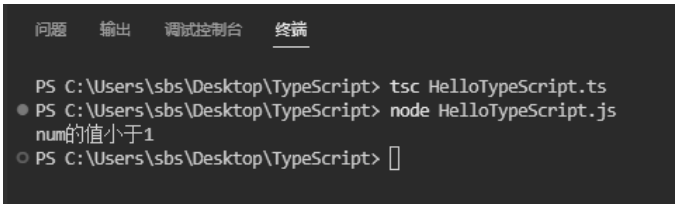

图 2-26　if…else…if 语句示例代码的运行结果

4. switch…case 语句

该语句用于判断一个变量与一系列值中的某个值是否相等。

例 2-56　switch…case 语句的示例代码

```
// switch…case 语句
var str:string = "B";
switch(str) {
    case "A": {
        console.log(" 优 ");
        break;
    }
    case "B": {
        console.log(" 良 ");
        break;
    }
    case "C": {
        console.log(" 及格 ");
        break;
    }
    case "D": {
        console.log(" 不及格 ");
        break;
    }
    default: {
        console.log(" 非法输入 ");
        break;
    }
}
```

编译为 JavaScript 代码后运行结果如图 2-27 所示。

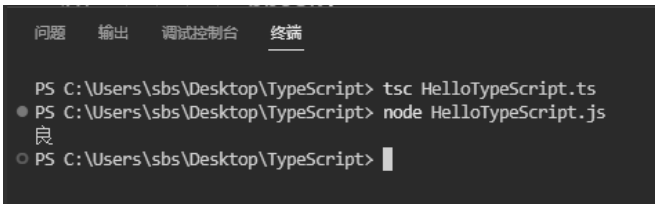

图 2-27　switch…case 语句示例代码的运行结果

说明：当执行到 break; 时，switch 语句会被终止。default 语句修饰的代码块在所有 case 语句都不满足时执行。

2.5.2　循环语句

循环语句可以重复调用某一段代码，直到满足条件时退出，在开发过程中循环语句也是比较常用的语句，包括 for 循环语句、for…in 循环语句、for…of 循环语句、forEach 循环语句，以及 while 循环语句、do…while 循环语句。下面将详细介绍这些循环语句的用法。

1. for 循环语句

例 2-57　for 循环语句的示例代码

```
let array = [1, "小明", false];
var i:number = 0;
// 通过 for 循环获取数组中的元素
for (i; i <array.length; i++) {
    console.log(array[i])
}
```

编译为 JavaScript 代码后运行结果如图 2-28 所示。

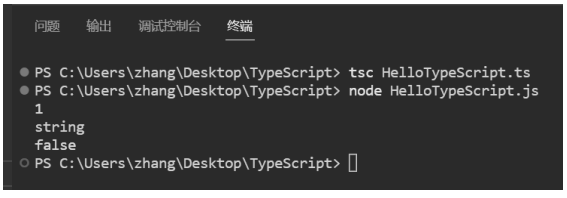

图 2-28　for 循环语句示例代码的运行结果

2. for…in 循环语句

例 2-58　for…in 循环语句的示例代码

```
let array = [1, "小明", false];
var i:string;
// 通过 for…in 循环获取数组中的元素
for (i in array) {
```

```
    console.log(array[i])
}
```

编译为 JavaScript 代码后运行结果如图 2-29 所示。

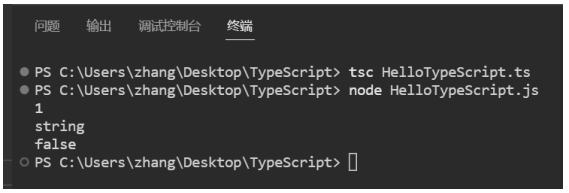

图 2-29　for…in 循环语句示例代码的运行结果

3. for…of 循环语句

例 2-59　for…of 循环语句的示例代码

```
let array = [1, "小明", false];
// 通过 for...of 循环获取数组中的元素
for (let i of array) {
    console.log(i)
}
```

编译为 JavaScript 代码后运行结果如图 2-30 所示。

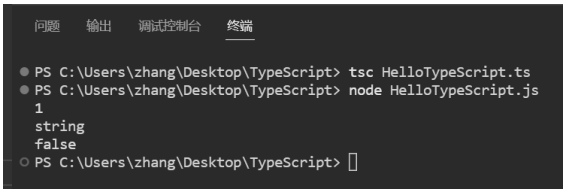

图 2-30　for…of 循环语句示例代码的运行结果

4. forEach 循环语句

例 2-60　forEach 循环语句的示例代码

```
let array = [1, "小明", false];
// 通过 forEach 循环获取数组中的元素
array.forEach((value, index) => {
    console.log(value);
});
```

编译为 JavaScript 代码后运行结果如图 2-31 所示。

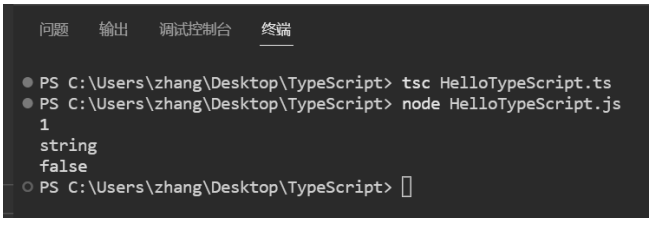

图 2-31　forEach 循环语句示例代码的运行结果

5. while 循环语句

例 2-61　while 循环语句的示例代码

```
let num = 3
// 使用 while 循环打印出从 3 开始大于或等于 1 的数
while (num >= 1) {
    console.log(num);
    num--;
}
```

编译为 JavaScript 代码后运行结果如图 2-32 所示。

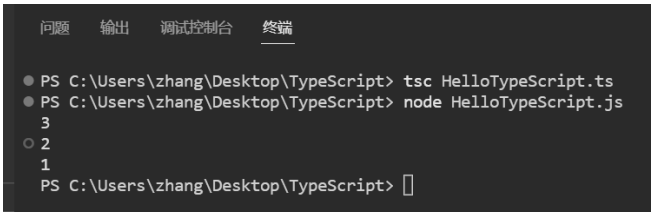

图 2-32　while 循环语句示例代码的运行结果

6. do…while 循环语句

例 2-62　do…while 循环语句的示例代码

```
let num = 3
// 使用 do…while 循环打印出从 3 开始大于或等于 1 的数
do {
    console.log(num);
    num--;
} while(num>=1);
```

编译为 JavaScript 代码后运行结果如图 2-33 所示。

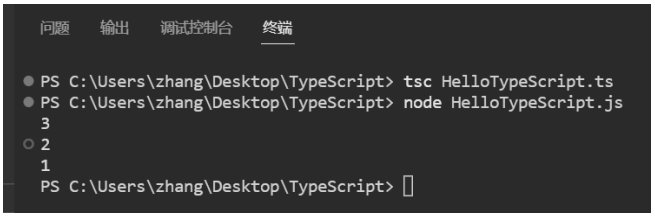

图 2-33　do…while 循环语句示例代码的运行结果

2.5.3　跳转语句

如果想要在一个循环过程中跳出循环，可以使用 break 语句和 continue 语句。break 语句和 continue 语句都可以实现跳出循环的操作，区别在于 break 语句跳出循环时会直接退出所有的循环体，而 continue 语句只会终止当前迭代，不会直接跳出所有循环。

例 2-63　使用 break 跳出循环的示例代码

```
let num: number = 1
// 当 num 的值为 3 时跳出循环
while (num < 5) {
    if (num == 3) {
        break;
    }
    console.log(num);
    num++;
}
```

编译为 JavaScript 代码后运行结果如图 2-34 所示。

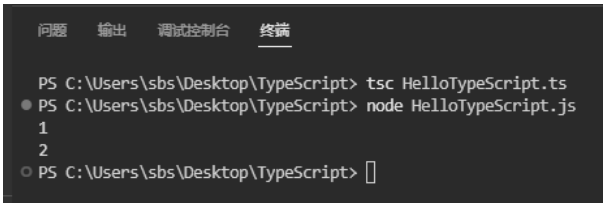

图 2-34　使用 break 跳出循环示例代码的运行结果

例 2-64　使用 continue 跳出循环的示例代码

```
var num: number = 0
//  输出 0 ～ 10 的奇数
for (num = 0; num <= 10; num++) {
    if (num % 2 == 0) {
        continue;
    }
    console.log(num)
}
```

编译为 JavaScript 代码后运行结果如图 2-35 所示。

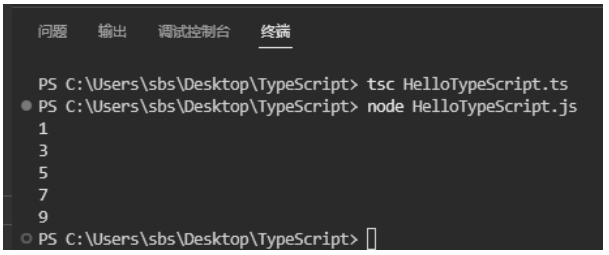

图 2-35　使用 continue 跳出循环示例代码的运行结果

2.6 就业面试技巧与解析

本章主要讲解了 TypeScript 的编程术语、基本语法、数据类型、运算符和控制语句等，通过上面的讲解，相信大家都已熟练掌握。这些知识在面试中常以下面的形式体现。

2.6.1 面试技巧与解析（一）

面试官：说一说你在开发中常用的数据类型都有哪些。

应聘者：在开发中我常用的数据类型有以下几种。

（1）数字类型：用于定义整数、小数和负数等。

（2）字符串类型：用于定义一些字符。

（3）布尔类型：用于表示 true 和 false。

（4）数组类型：用于定义和存储相同类型的对象。

（5）元组类型：用于定义和存储不同类型的对象。

（6）枚举类型：常用于定义数值集合。

2.6.2 面试技巧与解析（二）

面试官：描述一下 break 和 continue 的区别。

应聘者：在 TypeScript 中，break 语句和 continue 语句都是用来跳出循环的，区别在于 break 语句的作用是立即跳出当前循环，而 continue 语句的作用是停止正在进行的循环，直接进入下一次循环。