

第 1 章

◀ TypeScript 基础 ▶

JavaScript 可以说是当前 Web 开发中流行的脚本语言，是作为前端开发工程师必备的一项技能。随着编程技术的发展，JavaScript 现在已经成为一门功能全面的编程语言，能够处理复杂的计算和交互。Node.js 的出现，让 JavaScript 可以编写服务端代码，Node.js 的出现使得 JavaScript 成为与 PHP、Python、Perl 和 Ruby 等服务端脚本语言平起平坐的语言。在目前各类应用程序 Web 化和移动化的背景下，JavaScript 语言可谓如日中天。

JavaScript 是弱类型的语言，设计得过于灵活，导致编写的代码可能存在预期之外的各类奇葩 Bug，因此在使用 JavaScript 构建大型可扩展的应用时，可能会出现代码后续难以升级和维护的情况。

那么有没有这样一种语言，既可以兼容标准的 JavaScript 语法，同时又具有 C#或 Java 这类高级语言的若干特征呢？如可以采用面向对象的编程方法，分模块地构建 JavaScript 库和 Web 应用；在编写 JavaScript 代码时，可以实现智能语法提示，并在编码（编译）阶段发现语法和类型错误，从而降低代码在运行时的错误率。

鉴于 JavaScript 目前在构建大规模、可扩展应用上的不足，微软公司设计了 TypeScript 语言。TypeScript 语言具有静态类型检测和面向对象的特征，在编译阶段可以及时发现语法错误，同时支持分模块开发，编译后转换成原生的 JavaScript 代码，可以直接运行在各类浏览器上，而不需要额外的配置。

通过本章的学习，可以让读者了解 TypeScript 的基本概念以及开发环境搭建。本章主要涉及的知识点有：

- TypeScript 相关概念。
- TypeScript 和 JavaScript 区别。
- TypeScript 相比 JavaScript 具有哪些优势。
- TypeScript 开发环境搭建，学会基本的 TypeScript 开发环境搭建，以及构建第一个简单的 TypeScript 应用。



本章重点介绍一下 TypeScript 背景，下一章开始介绍 TypeScript 的基本语法知识。

1.1 什么是 TypeScript

TypeScript 是微软公司开发的开源编程语言。它本质上是在 JavaScript 语言中添加了可选的静态类型和基于类的面向对象编程等新特征。TypeScript 是由大神 Anders Hejlsberg 主导设计的。他是 Turbo Pascal 编译器的主要作者、Delphi、C#和 TypeScript 之父以及.NET 的创立者，有评论说他对语言和汇编的理解全世界没几个人能超越，足见其造诣之高。

2012 年 10 月，微软发布第一个 TypeScript 版本，截止到此书写作时，最新版本为 TypeScript 3.3，经过多次版本的迭代，目前 TypeScript 语言已经日趋成熟。

维基百科（<https://en.wikipedia.org/wiki/TypeScript>）中整理的关于 TypeScript 的历史版本如表 1.1 所示。

表 1.1 TypeScript 的历史版本

版本号	发布日期	重大变化
0.8	2012 年 10 月 1 日	首次发布
0.9	2013 年 6 月 18 日	无
1.1	2014 年 10 月 6 日	性能改进，1.1 版本的编译器速度比之前发布的版本快 4 倍
1.3	2014 年 11 月 12 日	protected 修饰符，元组类型
1.4	2015 年 1 月 20 日	联合类型，let 和 const 声明，类型别名等
1.5	2015 年 7 月 20 日	ES6 模块，namespace 关键字，for..of 支持，装饰器
1.6	2015 年 9 月 16 日	JSX 支持，交集类型，本地类型声明，抽象类和方法，用户定义的类型保护功能
1.7	2015 年 11 月 30 日	async 和 await 支持
1.8	2016 年 2 月 22 日	约束泛型，控制流分析错误和 allowJs 选项
2.0	2016 年 9 月 22 日	null 和 undefined 类型，基于控制流的类型分析，区分联合类型，never 类型，readonly 关键字和 this 函数类型
2.1	2016 年 11 月 8 日	keyof 和查找类型，映射类型
2.2	2017 年 2 月 22 日	混合类，object 类型
2.3	2017 年 4 月 27 日	async 迭代，泛型参数默认值，严格选项
2.4	2017 年 6 月 27 日	动态导入表达式，字符串枚举，泛型的改进推理，回调参数的严格逆解
2.5	2017 年 8 月 31 日	可选的 catch 子句变量
2.6	2017 年 10 月 31 日	严格的功能类型
2.7	2018 年 1 月 31 日	常量命名属性，固定长度元组
2.8	2018 年 3 月 27 日	条件类型，keyof 改进

(续表)

版本号	发布日期	重大变化
2.9	2018 年 5 月 14 日	支持 <code>keyof</code> 和映射对象类型中的符号和数字文字
3.0	2018 年 7 月 30 日	项目引用, 使用元组提取和传播参数列表
3.1	2018 年 9 月 27 日	可映射的元组和数组类型
3.2	2018 年 11 月 30 日	更严格地检查绑定和调用
3.3	2019 年 1 月 31 日	关于联合类型方法的宽松规则, 复合项目的增量构建

那么到底什么是 TypeScript 呢?

TypeScript 是 JavaScript 的超集, 专门为开发大规模可扩展的应用程序而设计, 且可编译为原生 JavaScript 的一种静态类型语言。TypeScript 从命名上可以看出是由 Type 和 Script 组成的, 其中 Type 表示是一种类型语言, 可以进行静态类型检查, Script 表示是兼容 JavaScript 的脚本语言。

编程语言中类型系统是提高代码性能的一个关键因素, 类型系统对构建优化的编译器和进行语法正确性检查等非常有用。同时类型系统可以为集成开发环境 (Integrated Development Environment, 简称 IDE) 提供智能代码补全、重构和代码导航这些功能, 而这背后都离不开具有类型系统的编译器。

如果希望编程语言具有可维护性, 在灵活和规范之间寻求合适的度非常重要。动态语言对于开发小型项目非常有用, 但大型项目需要采用严格的类型检查。Python 作者 Guido 计划在 Python 语言中也添加类似 TypeScript 的类型系统技术。

TypeScript 和 JavaScript 的逻辑关系可以用图 1.1 表示。

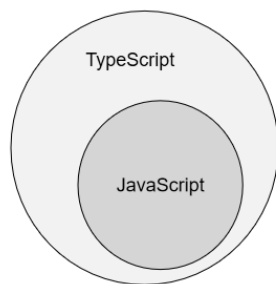


图 1.1 TypeScript 和 JavaScript 的关系图

提示

虽然图 1.1 表示的是 TypeScript 包含 JavaScript, 但还有极少数 JavaScript 语法在 TypeScript 中不支持, 如 `with`。

在很多介绍 JavaScript 和 TypeScript 的地方都会涉及一个名词 ECMAScript。这里有必要先介绍一下 ECMAScript 的基本概念。

ECMAScript 是一种由 ECMA 国际 (European Computer Manufacturers Association) 通过

ECMA-262 标准化的脚本程序设计语言。ECMAScript 可以理解为 JavaScript 的标准规范。到写本书为止有 6 个 ECMAScript 版本，具体如表 1.2 所示。

表 1.2 ECMAScript 版本

版本	时间	说明
ECMAScript 1	1997 年 06 月	首版
ECMAScript 2	1998 年 06 月	格式修正，以使得其形式与 ISO/IEC16262 国际标准一致
ECMAScript 3	1999 年 12 月	强大的正则表达式，更好的文字链处理，新的控制指令，异常处理，错误定义更加明确，数字输出的格式化及其他改变
ECMAScript 4		放弃发布
ECMAScript 5	2009 年 12 月	完善了 ECMAScript 3 版本、增加严格模式"strict mode"以及新的功能，如 getter 和 setter、JSON 库支持和更完整的对象属性
ECMAScript 5.1	2011 年 06 月	使规范更符合 ISO/IEC16262:2011 第三版
ECMAScript 6	2015 年 06 月	ECMAScript 6 (ES6)，也称为 ECMAScript 2015 (ES2015)。增加了非常重要的内容：let、const、class、modules、arrow functions、template string、destructuring、default、rest argument、binary data、promises 等
ECMAScript 7	2016 年 06 月	也被称为 ECMAScript 2016 (ES2016)。主要是完善 ES6 规范，除此之外还包括两个新的功能：求幂运算符 (**) 和 array.prototype.includes 方法
ECMAScript 8	2017 年 06 月	增加新的功能，如并发、原子操作、字符串填充、Object.values 和 Object.entries、await/async 等

虽然 ECMAScript 6 有大量的更新，但是它依旧完全向后兼容以前的版本。各主流浏览器的新版本基本都支持 ECMAScript 5 特征，具体可以访问 <https://caniuse.com> 网址输入 ES5 关键词进行查询，如图 1.2 所示。



图 1.2 ES5 各浏览器支持情况

截止到写此书时，ES6 的新特征仍然没有被目前所有主流浏览器所支持，所以热衷于使用

ES6 最新特性的开发者需要将代码转译为 ES5 代码。若想一睹各浏览器对于 ES6 特性的具体支持情况，这里推荐参考由 kangax 维护的 ECMAScript 兼容表（ECMAScript Compatibility Table），网址为 <https://kangax.github.io/compat-table/es6/>。

TypeScript 与 ECMAScript 6 规范一致。TypeScript 设计的目标是让 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。TypeScript 的语言功能除符合 ECMAScript 6 规范外，还包含泛型和类型注释等功能，这些功能是对 ECMAScript 6 规范的扩展。



一般来说，TypeScript 需要编译成 JavaScript 才能运行。

TypeScript 语言具有如下特点：

- TypeScript 以 JavaScript 为基础

TypeScript 是 JavaScript 的超集，意味着合法的 JavaScript 代码（也有少数例外）可以直接保存成扩展名为 .ts 的文件，即可用 TypeScript 编译器进行编译并运行。

- TypeScript 支持第三方 JavaScript 库

由于 TypeScript 在编译后就转成原生的 JavaScript 代码，因此第三方 JavaScript 框架或工具可以很方便地在 TypeScript 代码中进行引用。

- TypeScript 是可移植的

TypeScript 是可以跨浏览器、设备和操作系统进行移植的，即“一处编写，多处运行”。它可以在 JavaScript 的任何环境中运行，而且可以用 ES6 的语法来编写代码，通过配置生成 ES5 版本（或其他版本）JavaScript 代码。

- TypeScript 是静态类型语言

TypeScript 是静态类型语言，可以在代码编辑阶段进行类型检查，及时发现语法等错误，以提高代码的稳定性。

1.2 为什么要学习 TypeScript

任何一门语言的诞生和发展都是有缘由的，从某种程度上来说，TypeScript 语言的诞生是历史发展的必然。目前 Web 应用越来越复杂，必然导致 JavaScript 代码的快速增长。

由于目前各主流浏览器中的 JavaScript 引擎还没有完全实现 ES6 的特征，如 JavaScript 模块导入与导出和面向对象编程中的类与接口等。另外，JavaScript 是一种动态语言，很难做到静态类型检查。这将导致很多 JavaScript 语法问题在编码阶段无法暴露，而只能在运行时暴露。

在这种背景下，微软使用 Apache 授权协议推出开源语言 TypeScript，增加了可选类型、类和模块等特征，可编译成标准的 JavaScript 代码，并保证编译后的 JavaScript 代码兼容性。

另外, TypeScript 是一门静态类型语言, 本身具有静态类型检查的功能, 很好地弥补了 JavaScript 在静态类型检查上的不足。

因此, TypeScript 非常适合开发大规模可扩展的 JavaScript 应用程序。这也是我们要学习 TypeScript 的主要原因。换句话说, 学习 TypeScript 可以让开发和维护大规模可扩展的 JavaScript 应用程序以满足目前日益复杂的 Web 应用对 JavaScript 的要求。

不能说 TypeScript 有多牛, 只能说 TypeScript 顺应了时代。TypeScript 带有编译期类型检查, 在编写大规模应用程序的时候有明显优势, 更容易进行代码重构和让别人理解代码的意图。TypeScript 语言从 C# 语言中继承了很多优雅的设计, 比如枚举、泛型等语言特性, 这让 TypeScript 在语法上更加优雅。

1.2.1 TypeScript 与 JavaScript 对比有什么优势

TypeScript 和 JavaScript 的关系实际上就像 Java 和 Groovy 的关系, 一个静态, 一个动态, 前者稳健, 后者灵活。

JavaScript 是一种脚本语言, 无须编译, 基于对象和事件驱动, 只要嵌入 HTML 代码中, 就能由浏览器逐行加载解释执行, JavaScript 的语法虽然简单, 但是比较难以掌握, 使用的变量为弱类型, 比较灵活。

TypeScript 是微软开发和维护的一种具有面向对象功能的编程语言, 是 JavaScript 的超集, 可以载入 JavaScript 代码运行, 并扩展了 JavaScript 的语法。TypeScript 增加了静态类型、类、模块、接口和类型注解等特征。

概括起来, TypeScript 与 JavaScript 相比, 主要具有以下优势:

- 编译时检查

TypeScript 是静态类型的语言, 静态类型可以让开发工具(编译器)在编码阶段(编译阶段)即时检测各类语法错误。对于任何一门语言来说, 利用开发工具即时发现并提示修复错误是当今开发团队的迫切需求。有了这项功能后, 就会让开发人员编写出更加健壮的代码, 同时也提高了代码的可读性。

- 面向对象特征

面向对象编程可以更好地构建大规模应用程序, 通过对现实问题合理的抽象, 可以利用面向对象特征中的接口、类等来构建可复用、易扩展的大型应用程序。TypeScript 支持面向对象功能, 可以更好地构建大型 JavaScript 应用程序。

- 更好的协作

当开发大型项目时, 会有许多开发人员参与, 此时分模块开发尤其重要。TypeScript 支持分模块开发, 这样可以更好地进行分工协作, 最后在合并的时候解决命名冲突等问题, 这对于团队协作来说是至关重要的。

- 更强的生产力

TypeScript 遵循 ES6 规范，可以让代码编辑器（IDE）实现代码智能提示，代码自动完成和代码重构等操作，这些功能有助于提高开发人员的工作效率。



不是任何规模的 Web 应用都适合用 TypeScript，对于小规模应用 JavaScript 可能更合适。

1.2.2 TypeScript 给前端开发带来的好处

鉴于前面阐述的 TypeScript 语言的设计初衷，以及相比 JavaScript 的若干优势，可以概括出 TypeScript 给前端开发带来的好处：

- 提高编码效率和代码质量

传统的 JavaScript 在编写代码时，往往比较痛苦的是没有一个很好的编辑器（IDE），可以像 C#或者 Java 那样，IDE 对代码进行智能提示和语法错误检查，从而导致 JavaScript 代码在编码阶段很难发现潜在的错误。

TypeScript 是一种静态类型语言，可以让编辑器实现包括代码补全、接口提示、跳转到定义和代码重构等操作。借助编辑器，可以在编译阶段就发现大部分语法错误，这总比 JavaScript 在运行时发现错误要好得多。

- 增加了代码的可读性和可维护性

一般来说，理解 C#代码或者 Java 代码会比 JavaScript 代码更加容易，因为 C#或 Java 语言是强类型的，且支持面向对象特征。强类型语言本身就是一个很好的说明文档，大部分函数可以看类型定义就大致明白如何使用。JavaScript 很多库中利用了不少高级语言特征，开发人员可能无法很好地理解其意图。

- 胜任大规模应用开发

TypeScript 是具有面向对象特征的编程语言，在大规模应用开发中，可以利用模块和类等特征对代码进行合理规划，达到高内聚低耦合的目的。TypeScript 可以让复杂的代码结构更加清晰、一致和简单，降低了代码后续维护和升级的难度。因此，在面对大型应用开发时，使用 TypeScript 往往更加合适。

- 使用最先进的 JavaScript 语法

TypeScript 语法遵循 ES6 规范，由于其语法和 JavaScript 类似，因此前端从 JavaScript 转入 TypeScript 会感觉差异并不大，降低了 TypeScript 的学习难度。TypeScript 更新速度较快，不断支持最新的 ECMAScript 版本特性，以帮助构建强大的 Web 应用。

TypeScript 可以让前端开发人员利用先进的 JavaScript 功能去编写代码，然后通过编译，自动生成针对 ES5 或者 ECMAScript 3 环境的 JavaScript，让先进的技术落地。

1.3 安装 TypeScript

前面阐述了 TypeScript 的相关概念,以及 TypeScript 相比 JavaScript 来说具备的若干优势。作为程序员来说,经常挂在嘴上的是“光说好没用,关键是给我看代码”(Talk is cheap. Show me the code)。

假设把学习 TypeScript 当作一次自驾游,那么搭建 TypeScript 的开发环境就相当于确定自驾游的代步工具。

获取 TypeScript 开发工具有两种主要方法:

- 通过 npm 命令行安装。
- 通过安装 TypeScript 的 Visual Studio 插件。

下面分别说明这两种方法的安装步骤。

1.3.1 npm 安装

npm 是 JavaScript 常用的包管理工具,也是 Node.js 的默认包管理工具。通过 npm 可以安装、共享、分发代码和管理项目依赖关系。

由于 TypeScript 需要通过 npm 安装,而 npm 依赖于 Node.js,因此第一步就是先安装 Node.js 环境。

1. 安装 Node.js

(1) Node.js 发布于 2009 年 5 月,由 Ryan Dahl 开发,实质是对 Chrome V8 引擎进行封装。可以在 <https://nodejs.org> 官网下载最新的版本(例如下载 10.15.1 LTS 版),如图 1.3 所示。

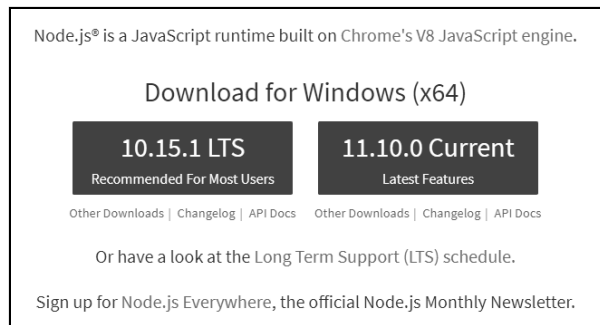


图 1.3 Node.js 下载界面

(2) 如果需要下载之前的版本,可以在 <https://nodejs.org/en/download/releases/> 中下载。这里下载 Node.js 10.13.0 进行安装。下载完成后双击文件,打开安装界面,如图 1.4 所示。

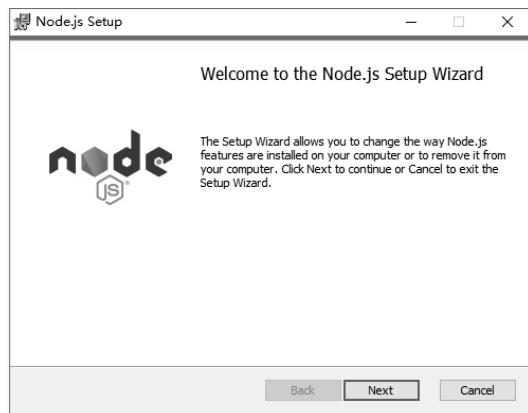


图 1.4 Node.js 安装界面

(3) 一般来说, 按照向导一步一步根据默认配置进行安装即可, 安装完成后, 需要验证一下安装是否成功。用组合键 Win+R 打开 Windows 操作系统上的运行界面, 输入 “cmd” 后按回车键打开命令行工具。



演示环境的操作系统是 Windows 10 64 位教育版。

(4) 在命令行工具中输入 “node -v” 和 “npm -v” 查看是否安装成功, 如果安装成功就会显示版本号, 如图 1.5 所示。

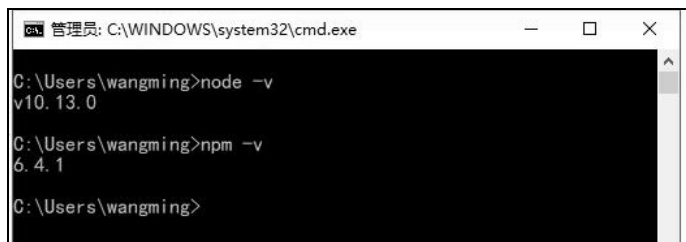


图 1.5 Node.js 查看版本界面

从图 1.5 可以看出, node -v 命令输出 Node.js 的版本为 10.13.0, 说明 Node.js 安装成功, 且 Node.js 目录已经注册到环境变量 path 中。npm 是随同 Node.js 一起安装的包管理工具, 能解决 Node.js 代码部署上的很多问题。



npm 的包安装分为本地安装 (local) 和全局安装 (global) 两种, 从命令来看, 差别只是有没有 -g。例如, npm install express -g 表示全局安装, 如果不带 -g 就表示本地安装。

本地安装将安装包放在 ./node_modules 下 (运行 npm 命令时所在的目录), 如果没有 node_modules 目录, 就会在当前执行 npm 命令的目录下生成 node_modules 目录。本地安装的模块需要通过 require() 来引入。

全局安装则将安装包放在 node 的安装目录下 (window), 全局安装的模块可以直接在命

令行里使用。

2. 使用 npm 安装 TypeScript

在命令行工具界面中输入命令“npm install -g typescript”全局安装 TypeScript，稍等片刻，等待安装完成后，用命令 tsc -v 查看其版本号来验证是否安装成功，如图 1.6 所示。

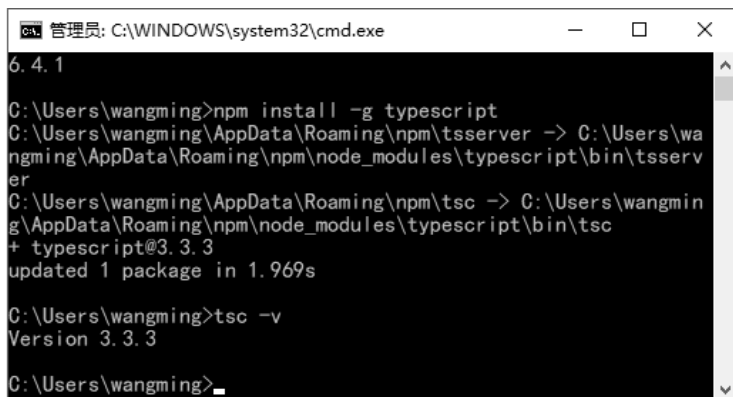


图 1.6 npm 安装 TypeScript 界面

从图 1.6 可以看出，当前安装的 TypeScript 版本为 3.3.3。至此，TypeScript 安装完成。



npm 默认镜像是国外的地址，速度可能会比较慢，或者无法下载包。建议将包仓库地址配置为国内镜像，如 npm 淘宝镜像。

修改 npm 的镜像如代码 1-1 所示。

【代码 1-1】持久修改 npm 的镜像：npm_register.txt

```
01 //持久使用
02 npm config set registry https://registry.npm.taobao.org
03 //验证是否成功
04 npm config get registry
```

1.3.2 Visual Studio 插件安装

由于 TypeScript 语言是微软公司开发的，因此势必在其 IDE Visual Studio 上进行集成。Visual Studio 2017 和 Visual Studio 2015 Update 3 默认包含 TypeScript。

Visual Studio 是一个完整的集成开发工具，提供了一站式开发工具集合，能够支持现在 IT 行业上主流的编程语言。它包括了整个软件生命周期所需要的大部分工具，如 UML 建模工具、代码管理工具、代码编辑和调试、程序测试和程序发布等。Visual Studio 所写的目标代码适用于微软支持的所有平台。

Visual Studio 版本很多，其中 Visual Studio Community 为社区版，适用于学生、开源和个人。该版本有相对完备的免费 IDE，可用于开发 Android、iOS、Windows 和 Web 的应用程序。

如果在安装 Visual Studio 的时候未安装 TypeScript 工具,后续仍可通过下载插件 TypeScript SDK for Visual Studio 进行安装。

1. 安装 TypeScript SDK for Visual Studio

(1) 打开 Visual Studio 开发工具,在菜单【工具】下单击【扩展和更新(U)...】菜单项,界面如图 1.7 所示。



图 1.7 Visual Studio 工具菜单界面

(2) 在弹出的【扩展和更新】界面,通过在右边的文本框中输入 typescript 进行联网搜索,找到对应版本的 TypeScript SDK for Visual Studio,这里选择 TypeScript 3.3.1 for Visual Studio 2017,单击【下载】按钮进行插件下载,如图 1.8 所示。

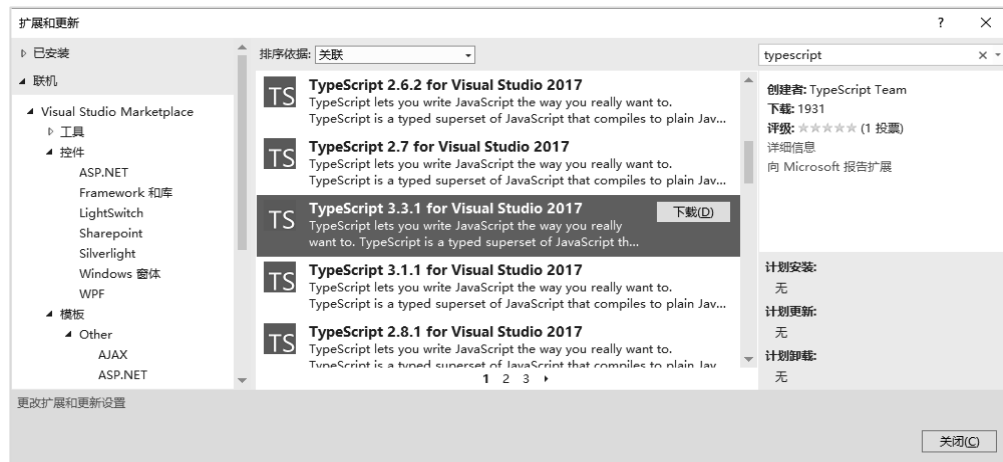


图 1.8 Visual Studio 扩展和更新界面

(3) 下载完成后,双击 TypeScript_SDK.exe 文件进行 TypeScript 环境安装,在弹出的安装界面上单击【Install】按钮完成安装,如图 1.9 所示。

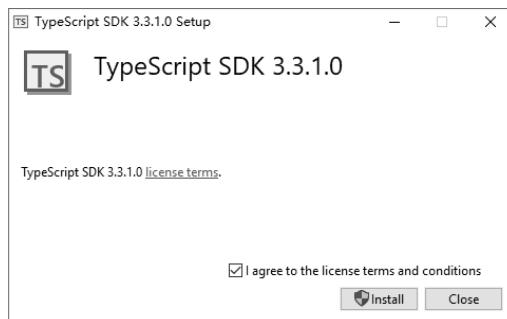


图 1.9 TypeScript SDK 安装界面

2. 安装 TypeScript HTML Application Template

TypeScript SDK 安装完成后，并没有包含创建 TypeScript 项目的模板，因此还需要通过扩展和更新界面安装 TypeScript HTML Application Template 插件，单击【下载】按钮进行联网下载并完成安装，如图 1.10 所示。

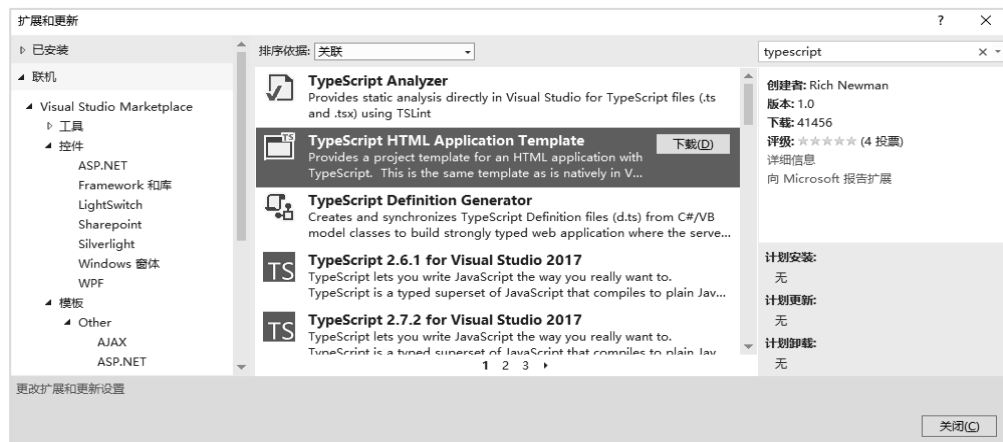


图 1.10 TypeScript HTML Application Template 下载界面

在弹出的【VSIX Installer】界面中，单击【修改】按钮进行安装，如图 1.11 所示。

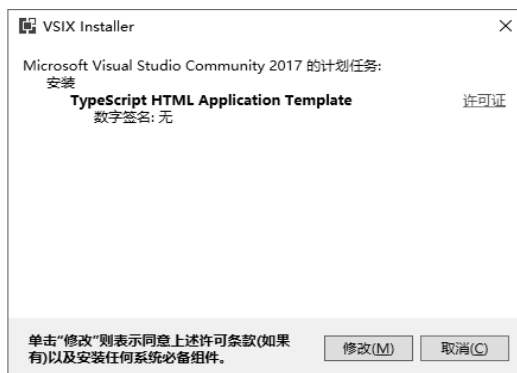


图 1.11 TypeScript HTML Application Template 插件安装界面

至此，通过 Visual Studio 安装 TypeScript 相关插件来搭建开发环境就完成了。



TypeScript HTML Application Template 插件在下载完成后，需要重启 Visual Studio 才能安装。

1.4 开始第一个 TypeScript 文件

在 TypeScript 开发环境搭建完成后，就可以正式进入 TypeScript 程序开发了。本节通过创建一个简单的 TypeScript 程序让读者首先从感性上直观地了解 TypeScript 的编码、编译和运行基本过程。

按照惯例，学习一门新的语言，第一个程序往往都是 HelloWorld。我们也遵循这样的习惯，定义一个 helloWorld 函数，让 TypeScript 打印出“Hello, My First TypeScript”。

1.4.1 选择 TypeScript 编辑器

工欲善其事，必先利其器。虽然可以用任何文本编辑器进行 TypeScript 程序的开发，但是借助强大的编辑器既可以提高开发效率，也可以通过类型检测等手段保证代码质量。从 TypeScript 官网可以看出编辑器还是比较多的，如图 1.12 所示。

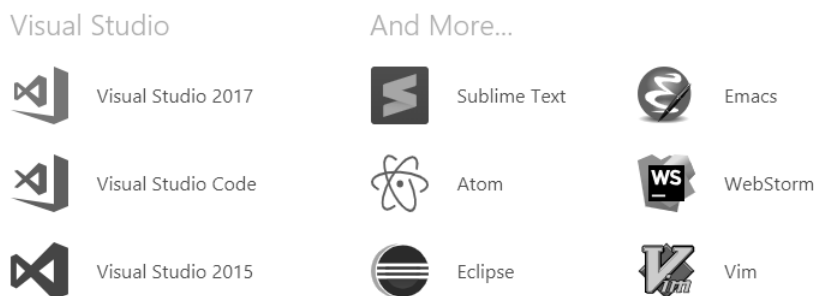


图 1.12 TypeScript 常用的编辑器

Visual Studio 2017/2015 安装所需空间比较大，比较消耗电脑资源，我们也可以选择微软的开源轻量级编辑器 Visual Studio Code 来开发，而且 Visual Studio Code 是跨操作系统的，不但可以在 Window 操作系统上进行程序开发，也可以在 Linux 和 Mac 中进行开发。

微软在 2015 年 4 月 30 日的 Build 开发者大会上正式宣布了 Visual Studio Code 项目，一个运行于 Mac OS X、Windows 和 Linux 之上的，针对编写现代 Web 和云应用的跨平台源代码编辑器。Visual Studio Code 对 Web 开发的支持尤其好，同时支持多种主流语言，例如 C#、Java、PHP、C++、JavaScript 和 TypeScript 等。

在网站 <https://code.visualstudio.com/download> 中可以下载安装文件，这里下载的是 Window

64 位 1.31.1 版本。下载完成后双击 VSCodeUserSetup-x64-1.31.1.exe 进行安装,如图 1.13 所示。

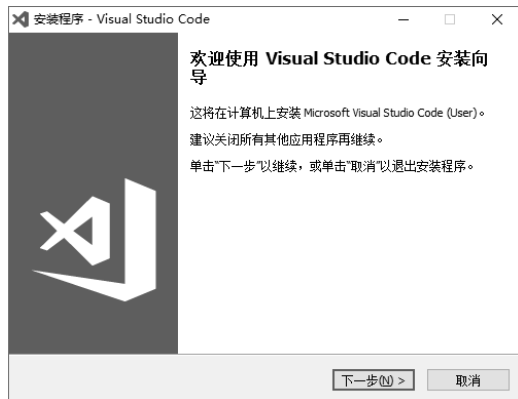


图 1.13 Visual Studio Code 安装界面

按照向导操作即可,最后完成 Visual Studio Code 的安装。

提示

Visual Studio Code 默认情况下是不包含 TypeScript 语言的,但可以通过安装插件来开发 TypeScript,同时 Visual Studio Code 通过插件还支持 C#、Java 和 PHP 等语言。

对于单个文件而言,用在线的编辑器进行编码会更加方便,在 TypeScript 官网上有一个练习(Playground)链接,即 <http://www.typescriptlang.org/play/index.html>,具体界面如图 1.14 所示。



图 1.14 在线 TypeScript Playground 界面

提示

TypeScript Playground 只能编写单个文件。使用它本地无须安装任何 TypeScript 环境。

由图 1.14 可以看出,在左边的区域是 TypeScript 脚本,其中【编译项】按钮可以配置一

些编译选项。当修改 TypeScript 脚本时，右边会自动转译成对应的 JavaScript 代码。当单击右边的【运行】按钮时，即可执行右边生成的 JavaScript 代码。

本书中前几章的演示代码只要是单文件的形式都可以在 TypeScript 官网上 Playground 提供的在线编辑器上运行。

1.4.2 编写 TypeScript 文件

当 Visual Studio Code (VSCode) 完成安装后，双击桌面快捷方式打开 VSCode 编辑器，在【File】菜单下单击【New File】创建一个新文件，并保存为 helloworld.ts。

TypeScript 编码有一些指导规则，函数命名采用 camelCase 命名规则，也就是首字母小写、其他单词首字母大写。文件 helloworld.ts 的内容如代码 1-2 所示。

【代码 1-2】 helloWorld 函数：helloworld.ts

```
01  /**
02   * 第一个 TS 程序（注释用于代码提示）
03   * @param <msg 字符串类型>
04   * @return(字符串)
05   */
06  function helloWorld(msg:string):string{
07      return "Hello, " + msg;
08  }
09  let msg = "My First TypeScript";
10  document.body.innerHTML = helloWorld(msg);
```

从上述代码中可以看出，helloWorld 的函数上用/**...*/ 写了一段注释。TypeDoc 可以根据写在/** ... */之间的注释生成 api 文档。在 Visual Studio Code 中打开 helloworld.ts，如图 1.15 所示。

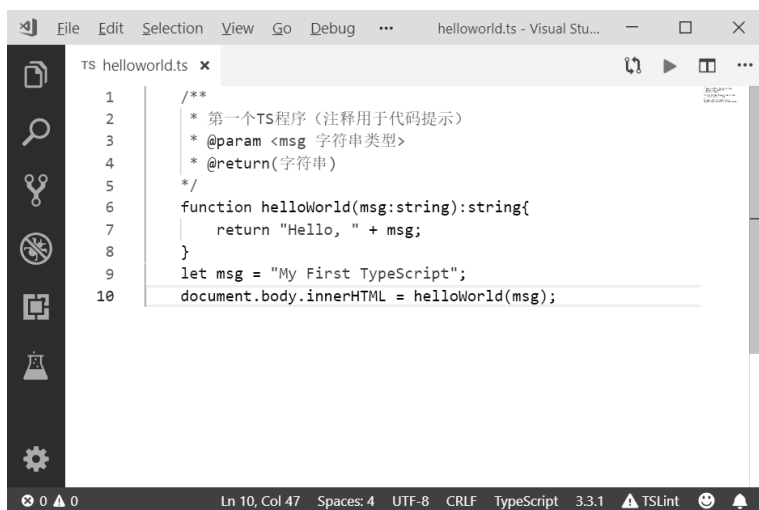


图 1.15 Visual Studio Code 编辑界面

1.4.3 编译 TypeScript 文件

前面提到，TypeScript 文件需要经过编译器编译后转成原生 JavaScript 代码才能执行。因此，为了运行 helloworld.ts 中的代码，需要对 helloworld.ts 文件进行编译。这里用 Windows 命令行工具调用 tsc 命令编译 helloworld.ts，命令如下：

```
tsc helloworld.ts
```

编译成功后，会在 helloworld.ts 同一个目录下生成一个 helloworld.js 文件，内容如代码 1-3 所示。

【代码 1-3】 helloworld.ts 编译后代码：helloworld.js

```
01  /**
02   * 第一个 TS 程序
03   * @param <msg>
04   * @return(string)
05   */
06  function helloWorld(msg) {
07      return "Hello, " + msg;
08  }
09  var msg = "My First TypeScript";
10  document.body.innerHTML = helloWorld(msg);
```

从代码 1-3 可以看出，TypeScript 代码和生成的 JavaScript 代码很相似。



在 Visual Studio Code 中需要经过配置才能自动进行 TypeScript 文件编译。

当然也可以同时编译多个.ts 文件，语法如下：

```
tsc file1.ts, file2.ts
```

tsc 常用编译参数如表 1.3 所示。

表 1.3 tsc 常用编译参数

编译参数	参数说明
--help	显示帮助信息
--module	载入扩展模块
--target	设置 ECMA 版本，如 ES5
--declaration	额外生成一个.d.ts 扩展名的文件，命令 tsc ts-hw.ts --declaration 会生成 ts-hw.d.ts、ts-hw.js 两个文件
--removeComments	删除文件的注释
--out	编译多个文件并合并到一个输出的文件
--sourcemap	生成一个 sourcemap (.map)文件。sourcemap 是一个存储源代码与编译代码对应位置映射的信息文件
--module noImplicitAny	在表达式和声明上有隐含的 any 类型时报错
--watch	在监视模式下运行编译器，会监视输出文件，在它们改变时重新编译

1.4.4 在网页中调用 TypeScript 文件

我们编写的 `helloworld.ts` 代码如何在浏览器里面运行呢？一般来说，浏览器不能直接嵌入 TypeScript 文件，而是嵌入 TypeScript 文件编译后对应的 JavaScript 文件。下面创建一个 `index.html` 来作为容器，将编译好的 `helloworld.js` 引入。`index.html` 具体内容如代码 1-4 所示。

【代码 1-4】 调用 `helloworld.ts` 生成的 `helloworld.js` 文件：`index.html`

```
01  <!DOCTYPE html>
02  <html lang="en">
03  <head>
04      <meta charset="UTF-8">
05      <meta name="viewport" content="width=device-width,
initial-scale=1.0">
06      <meta http-equiv="X-UA-Compatible" content="ie=edge">
07      <title>index</title>
08  </head>
09  <body>
10      <!-- 引入文件 -->
11      <script src="helloworld.js" ></script>
12  </body>
13  </html>
```

用浏览器打开 `index.html`，可以看到页面上打印出“Hello,My First TypeScript”的文本信息，界面如图 1.16 所示。

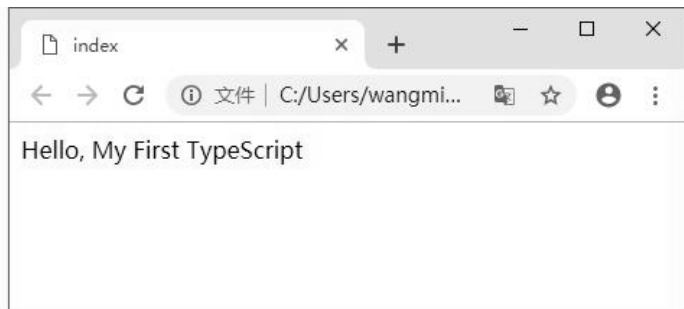


图 1.16 `index.html` 运行界面



虽然通过动态编译技术可以在浏览器中直接嵌入 TypeScript 文件，但是一般不建议这么做。

1.5 TypeScript 的组成部分（语言、编译器、语言服务）

TypeScript 整个体系组成比较复杂，从本质上讲它主要由以下 3 个部分构成：

- TypeScript 编译器核心：包括语法、关键字和类型注释等。
- 独立的 TypeScript 编译器(tsc.exe)：将 TypeScript 编写的代码转换成等效的 JavaScript 代码，可以通过参数动态生成 ES5 或者 ES3 等目标代码。
- TypeScript 语言服务：在 TypeScript 编译器核心层上公开了一个额外的层，是类似编辑器的应用程序。语言服务支持常见的代码编辑器中需要的操作，如代码智能提示、代码重构、代码格式化、代码折叠和着色等。

TypeScript 组成部分的分层示意如图 1.17 所示。



图 1.17 TypeScript 的组成分层示意图

1.6 小结

本章首先阐述了 TypeScript 语言的相关概念及其产生的缘由。TypeScript 作为 JavaScript 的超集，提供了面向对象和类型检查等新的特征，可以更好地支撑大规模应用程序的开发；然后阐述了与 JavaScript 相比 TypeScript 语言的主要优势以及我们学习 TypeScript 的目的。接着对搭建基本的 TypeScript 开发环境进行了详细说明，并在 Visual Studio Code 中构建了第一个 TypeScript 应用程序，让读者对 TypeScript 从基本编码以及用 tsc 命令进行编译到引入网页运行这个过程有了一个感性认识。

第 2 章

◀ TypeScript 基本语法 ▶

上一章主要对 TypeScript 语言相关概念和特点进行了概述，并罗列了 TypeScript 相比于 JavaScript 而言有哪些优势，以及 TypeScript 给前端开发带来的好处。本章将对 TypeScript 语言的基本语法进行详细说明。

学习任何一门编程语言，掌握其语言的基本语法是后续进行实战的基础。通过本章的学习，读者可以掌握 TypeScript 语言中的基本语法，如数据类型、变量和运算符。最后对数字（数值）和字符串这两种数据类型的基本方法进行详细说明。

本章主要涉及的知识点有：

- 类型：学会 TypeScript 的基础类型、枚举和特有的类型。
- 变量：学会 TypeScript 的变量声明以及作用域等。
- 运算符：学会 TypeScript 各类运算符的语法，如算术运算符、逻辑运算符和类型运算符等。
- 数字：学会 TypeScript 语言中数字的基本操作，如数值对象的属性和方法。
- 字符串：学会 TypeScript 语言中字符串的基本操作，如字符串的构造函数和方法。



本章内容会涉及后续章节才详细说明的函数和流程控制语句等内容，读者先不必细究。

2.1 认识一些编程语言的术语

TypeScript 是一门计算机编程语言，必然会涉及一些编程方面的术语。这些术语会经常在后面提及，因此有必要对这些术语的基本概念做一个说明，如果你之前学过面向对象或者其他语言，可以选择性地跳过阅读这些概念。

2.1.1 标识符

在计算机编程语言中，为了便于记忆和理解，使用一个符合约定的名字给变量、常量、函数和类等命名，以建立起名称与实体之间的映射关系，这个名字就是标识符（identifier）。标

标识符通常由字母和数字以及其他少数几个特有的字符构成。

通俗来讲，标识符可以理解为人姓名，往往我们用一个人的姓名来代表这个人，以区分不同的人。

2.1.2 数据类型

数据类型（Data Type）是一种数据分类，包含一组数据的共性属性和方法总称，它告诉编译器或解释器如何使用数据。数据类型限定了开发人员可以对数据执行的操作、数据的组成以及存储该类型的值的方式。

举例而言，鸟和鱼可以看成是一种动物的类型，鱼这个类型就限制了它的数据组成只能是草鱼、鲑鱼和鲨鱼等，同时鱼这个类型还有一些特有的行为，如在水中游；而鸟这个类型限制了它的数据组成只能是燕子、大雁和麻雀等，同时鸟这个类型还有一些特有行为，如在天上飞。

也就是说，只要某个事物可以归为某类型，那么它就必须具备这个类型的特有属性和方法。

2.1.3 原始数据类型

原始数据类型（Primitive Data Type）通常是内置或基本的语言实现类型。原始数据类型一般情况下与计算机内存中的对象一一对应，但由于语言及其实现的不同，也可能不一致。但是，通常情况下对原始数据类型的操作是最快的。原始类型基本上都是值类型，其赋值都是在内存中复制的副本。

2.1.4 变量和参数

变量（Variable）是一个用于保存值的占位符，可以通过变量名称来获得对值的引用。变量一般由变量名、变量类型和变量值组成。在计算机编程中，变量或标量是与关联的标识符配对的存储位置（由存储器地址标识），其包含值的一些信息。变量名称是引用存储值的常用方法。变量名和变量值的这种分离结构允许在程序运行时，变量名可以动态绑定值，换句话说，变量的值可以在程序执行过程中改变。变量名往往就是一个标识符，命名规则二者一致。

参数通俗地讲就是函数运算时需要参与运算的值。参数虽然和变量比较类似，但是二者还是不同的概念：参数一般用于函数中，变量既可以在函数中也可以在其他地方使用；参数一般用于传递值，而变量一般用于存储值。

2.1.5 函数和方法

函数是一段代码，通过函数名来进行调用，从而给外界提供服务。它能将一些数据（参数）传递进去进行处理，然后返回一些数据（返回值），也可以没有返回值。方法也是一段代码，也通过方法名来进行调用，但它必须依附于一个对象。方法和函数形式上大致是相同的，但使用上存在差异。将函数与某个对象建立联系时，函数就是方法。函数可以直接通过函数名调用，而方法必须通过对象和方法名来调用。

2.1.6 表达式和语句

表达式（Expression）是由数字、运算符、括号、变量名等按照一定顺序组成的且能求得值的式子，如 $x+(7*y)+2$ 。表达式本质上是一个值，可以当作一个具体的值使用。因此可以将它赋给变量，也可以当作参数传递。单独的一个运算对象（常量或变量）也可以叫作表达式，这是最简单的表达式。表达式一般只能出现在赋值的右边，而不能是左边。

语句（Statement）在 TypeScript 中是由分号结尾的，一条语句相当于一个完整的计算机指令，包括声明语句、赋值语句、函数表达式语句、空语句、复合语句（由花括号 {} 括起来的一条或多条语句）。二者的区别就是表达式可以求值，但是语句不可以。

2.1.7 字面量

字面量（Literal）是在编码中表示一个固定值的表示法（Notation）。几乎所有计算机编程语言都具有对基本值的字面量表示，如浮点数、字符串和布尔类型等。字面量也叫作直接量。例如，“Hello World”就是字符串字面量；99.88 就是数值字面量，true 就是布尔字面量。

2.2 认识 TypeScript 的简单语法

在介绍类型的时候会用一些代码来进行说明，就必然涉及一些 TypeScript 的编码规范和基本语法。因此在正式开始介绍 TypeScript 类型之前，本节有必要简要地介绍一下 TypeScript 的基本语法。

2.2.1 注释语法

为了提高代码的可读性，让其他人可以更好地理解某段代码的逻辑意图，必须在关键的地方对代码进行注释。在 TypeScript 语言中，注释方式主要有 3 种，分别是单行注释、多行注释以及用于生成 API 文档的注释。3 种注释方式如下所示：

```
01    // 当行注释
02    /*
03        多行注释
04        可以跨行进行注释
05    */
06    /**
07        * API 文档注释，可以供 TypeDoc 工具识别生成 API 说明文档
08    */
```

可以看出，// ... 表示单行注释，/* ... */用于多行注释，而 /** ... */ 用于自动生成 API 说明文档的注释。API 文档注释一般用于函数上，用来说明函数中参数的类型及参数的具体含义，同时说明函数的返回值。

通俗来讲，注释是给人阅读的，TypeScript 编译器并不会去解析它。一般情况下，当脚本文件发布到生产环境下后，会利用工具对代码进行混淆和压缩，这个过程中就将注释进行了删除，从而减少文件大小，且增加了被阅读的难度。

2.2.2 区分大小写

TypeScript 是区分大小写的，变量名 `someThing` 和 `something` 是不同的。因此在编码的时候一定要注意。

2.2.3 保留字

TypeScript 中有很多内置的类型和对象等，从而占用了一些标识符，这些用于系统的特殊标识符为语言的保留字，不能用于变量的命名（标识符）。例如，下面的关键词是保留字，是不能用作标识符的：

<code>break</code>	<code>case</code>	<code>catch</code>	<code>class</code>
<code>const</code>	<code>continue</code>	<code>debugger</code>	<code>default</code>
<code>delete</code>	<code>do</code>	<code>else</code>	<code>enum</code>
<code>export</code>	<code>extends</code>	<code>false</code>	<code>finally</code>
<code>for</code>	<code>function</code>	<code>if</code>	<code>import</code>
<code>in</code>	<code>instanceof</code>	<code>new</code>	<code>null</code>
<code>return</code>	<code>super</code>	<code>switch</code>	<code>this</code>
<code>throw</code>	<code>true</code>	<code>try</code>	<code>typeof</code>
<code>var</code>	<code>void</code>	<code>while</code>	<code>with</code>

下面的关键词不能用于用户定义的类型名称，这些是 TypeScript 内置的类型：

<code>any</code>	<code>boolean</code>	<code>number</code>	<code>string</code>
<code>symbol</code>			

下面的关键词在特定上下文中有特殊意义，虽然是合法的标识符，但是为了防止歧义，不建议使用：

<code>abstract</code>	<code>as</code>	<code>async</code>	<code>await</code>
<code>constructor</code>	<code>declare</code>	<code>from</code>	<code>get</code>
<code>is</code>	<code>module</code>	<code>namespace</code>	<code>of</code>
<code>require</code>	<code>set</code>	<code>type</code>	

2.2.4 语句用;分隔

两个语句之间若处于同一行，中间必须用英文分号（`;`）进行分隔。每行末尾可以省略；但是不建议这样操作，因为在压缩代码的时候会压缩到一行上，这样没有分隔的两个语句可能

会出现错误。



TypeScript 编译成 JavaScript 的时候会在没有分号的行末自动加上“;”。例如，“let b = 3”会编译为“var b = 3;”。

2.2.5 文件扩展名为.ts

TypeScript 脚本文件的扩展名为.ts。

2.2.6 变量声明

TypeScript 可以用 let 和 var 声明一个变量（变量的声明将在第3章详细介绍），声明变量的语法为：

```
let 或 var 变量名 : 数据类型 = 初始化值 ;
```

例如：

```
let varName : string = "hello world" ;
```



变量名必须遵循一定的命名规范，例如不允许用数字打头等。

2.2.7 异常处理

在 TypeScript 中，可以用 throw 关键字抛出一个异常。在 JavaScript 中，throw 可以抛出任何类型的异常。但是在 TypeScript 中，throw 抛出的必须是一个 Error 对象，如下所示。

```
throw new Error("错误信息");
```

要自定义异常，可以继承 Error 类。当需要一个特定的异常行为或者希望 catch 块可以分辨异常类型时，自定义异常就会很有用。处理异常需要使用 try ... catch 语句块。大体上和 C# 的使用方法类似。下面的代码 2-1 给出一段 try ... catch 的示例代码，此代码并不具有什么实际意义，只是为了演示而已。

【代码 2-1】 try ... catch 示例：try_catch.ts

```
01  try {  
02      let a = b / 0; // b 未定义  
03  }  
04  catch (error) {  
05      switch (error.name) {  
06          case 'errorOne': {  
07              console.log(error.message);  
08              break;
```

```
09      }
10      case 'errorTwo': {
11          console.log(error.message);
12          break;
13      }
14      default: {
15          throw new Error("异常:"+error);
16      }
17  }
18  }
19  finally {
20      console.log("执行结束");
21  }
```



TypeScript 不支持多个 `catch` 块，只能在一个 `catch` 中通过 `switch` 来区分不同的异常类型，进而进行差异化处理。

在很多语言中，任何数值除以 0 都会导致错误而终止程序执行。但是在 TypeScript（和 JavaScript 一致）中，会返回特殊的值，比 0 大的数除以 0 则会得到无穷大 `Infinity`，而 `0/0` 则返回 `NaN`，从而不会影响程序的执行。

2.3 类型

TypeScript 可以说是一门具有面向对象特征的静态类型语言，可以用来描述现实世界中的对象。不同的对象具有不同的属性和行为。一般来说，现实中的某个事物具有不同类型的属性，以人这个对象为例，有身高属性、名字属性、性别属性和是否结婚等属性，这些属性的值分别是数值型、字符型、枚举型和布尔型。



TypeScript 语言的静态类型系统 (Type System) 在程序编译阶段就可以检查类型的有效性，这可以确保代码按预期运行。

编程语言若没有类型，则无法确切描述现实对象，也就失去了编程语言的价值。因此，类型对于任何一门语言而言都是核心基础。TypeScript 中的所有类型都是 `any` 类型的子类型。`any` 类型可以表示任何值。根据官方的《TypeScript Language Specification》文档描述，TypeScript 数据类型除了 `any` 类型外，其他的可以分类为原始类型 (primitive types)、对象类型 (object types)、联合类型 (union types)、交叉类型 (intersection types) 和类型参数 (type parameters)。另外，数据类型又可以分为内置类型 (如数值类型) 和用户自定义类型 (如枚举类型和类等)。

本节介绍 TypeScript 语言的一些常用类型。

2.3.1 基础类型

TypeScript 的原始类型（primitive types）有数值型（number）、布尔型（boolean）、字符型（string）、符号型（symbol）、void 型、null 型、undefined 型和用户自定义的枚举类型 8 种。本小节重点对数值型、布尔型和字符型这些基础类型进行阐述。

1. 数值型

TypeScript 中的数值型和 JavaScript 一样，是双精度 64 位浮点值。它可以用来表示整数和分数，在 TypeScript 中并没有整数型。注意，在有些金融计算领域，如果对于精度要求较高，就需要注意计算误差的问题。

一般来说，判定一个变量是否需要设置成数值类型，可以看它是否需要进行四则运算，如果一个变量可以加减乘除，那么这个变量很可能就是数值类型，如金额。

数值类型的变量可以存储不同进制的数值，默认是十进制，也可以用 0b 表示二进制、0o 表示八进制、0x 表示十六进制。

下面给出几种 TypeScript 中数值型变量常见的声明语法，示例如代码 2-2 所示。

【代码 2-2】数值类型声明示例：numbers.ts

```
01 let num1: number = 89.2 ;           //分数，十进制
02 let int2 : number = 2 ;              //整数，十进制
03 let binaryVar: number = 0b1010;     //二进制
04 let octalVar: number = 0o744;       //八进制
05 let hexVar: number = 0xf00d;        //十六进制
```



变量声明数值类型后，不允许将字符类型或布尔类型等不兼容类型赋值给此变量。

01 行用“let num1: number”声明了一个名为 num1 的数值（number）类型变量。然后用=对变量 num1 进行初始化，初始值为 89.2，这个值并没有加前缀，因此默认是十进制的。这是最常用的一种数值进制。

另外需要注意的是，TypeScript 中有 Number 和 number 两种类型，但是它们是不一样的，Number 是 number 类型的封装对象。Number 对象的值是 number 类型。

2. 布尔型

布尔型数据表示逻辑值，值只能是 true 或 false。布尔值是最基础的数据类型，在 TypeScript 中，使用 boolean 关键词来定义布尔类型，示例如代码 2-3 所示。

【代码 2-3】布尔类型声明示例：boolean.ts

```
01 let isMan: boolean = true ;
02 let isBoy : boolean = false ;
```

01 行用“let isMan: boolean”声明了一个名为 isMan 的布尔（boolean）类型变量。然后用=对变量 isMan 进行初始化，初始值为 true，这个值只能是 true 或 false，不能是 1 或者 0 等。

另外需要注意的是，TypeScript 中有 Boolean 和 boolean 两种类型，但是它们是不一样的，Boolean 是 boolean 类型的封装对象，Boolean 对象的值是 boolean 类型。从下面给出的示例代码 2-4 可以看出，二者是存在差异的。

【代码 2-4】布尔类型示例：boolean2.ts

```
01 let a : boolean = new Boolean(1) ;    //错误
02 let b : boolean = Boolean(1) ;        //正确
```

从代码 2-4 可以看出，在 TypeScript 中，boolean 类型和 new Boolean(1) 是不兼容的。但是和直接调用 Boolean(1) 是兼容的。



在 TypeScript 中，Number、String 和 Boolean 分别是 number、string 和 boolean 的封装对象。

3. 字符型

字符型（字符串类型）表示 Unicode 字符序列，可以用单引号 ' 或者双引号 " 来表示字符。不过一般建议用双引号来表示字符。二者可以互相嵌套使用。\\ 符号可以对 " 等进行转义。从下面给出的示例代码 2-5 可以看出，单引号 ' 或者双引号 " 都可以给字符类型的变量进行赋值。

【代码 2-5】字符类型示例：string.ts

```
01 let msg: string= " hello world ";
02 let msg2: string= ' hello world ';
03 let a = "I'M ok" ;
04 let b = 'hello "world" ' ;
05 let c = "\"helo\"" ;
```

模板字符串（template string）是增强版的字符串，用反引号 ` 标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

下面的代码 2-6 给出了模板字符串示例。使用模板字符串的一个最大好处就是可以防止传统的变量和字符通过+进行拼接的时候单引号和双引号相互嵌套所导致的不容易发现拼接错误的问题。

【代码 2-6】模板字符串示例：string_template.ts

```
01 let name: string = "JackYunDi";
02 let age: number = 2;
03 let msg: string = '今年 ${name}已经${age}岁了';//今年 JackYunDi 已经 2 岁了
04 let b = '
05     hello typescript
06     hello world
```

07 ' ; //多行文本



字符串可以通过索引来获取值中对应的字符，如"hello"[0]输出 h。

另外，字符串可以通过+进行字符拼接，这个操作在日常的编程实战中也是非常常见的用法。代码 2-7 给出了字符串拼接示例。

【代码 2-7】 字符串+拼接示例：string_join.ts

```
01 let firstName: string = "Jack";
02 let lastName: string = "Wang";
03 let fullName = firstName + " " + lastName;
04 console.log(fullName);           //Jack Wang
```

如果字符串和数字用+字符连接，那么结果将成为字符串。因此可以将空字符串和数值相加用于将数值类型转化成字符类型，如代码 2-8 所示。

【代码 2-8】 字符串和数字拼接示例：string_join2.ts

```
01 let res = "" + 5;
02 console.log(res );           //"5"
```

另外，字符串和布尔型用+进行拼接，也生成字符串，如 true + "" 的值为"true"。

如果字符串和数组用+字符连接，那么结果将成为字符串。因此可以将空字符串和数组相加用于将数值的值转成用(,)分隔的一个字符串，如代码 2-9 所示。

【代码 2-9】 字符串和数组拼接示例：string_join3.ts

```
01 let res = "" + [1,2,3];
02 console.log(res );           //"1,2,3"
```

2.3.2 枚举

TypeScript 语言支持枚举(enum)类型。枚举类型是对 JavaScript 标准数据类型的一个补充。枚举用于取值被限定在一定范围内的场景，比如一周只能有 7 天，彩虹的颜色限定为赤、橙、黄、绿、青、蓝、紫，这些都适合用枚举来表示。

TypeScript 可以像 C#语言一样，可以使用枚举类型为一组数值赋予更加友好的名称，从而提升代码的可读性，枚举使用 enum 关键字来定义，代码 2-10 给出了枚举的示例。

【代码 2-10】 枚举示例：enums.ts

```
01 enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
02 let today: Days = Days.Sun;
```

在代码 2-10 中，01 行用 enum 关键词声明了一个名为 Days 的枚举类型，一般枚举类型的标识符首字母大写。02 行用自定义的枚举类型来声明一个新的变量 today，并赋值为 Days.Sun。使用枚举可以限定我们的赋值范围，防止赋值错误，例如不能为 today 变量赋值为

Days.OneDay。



一般情况下，枚举类型的变量本质上只是数值。Days.Sun 实际上是 0。“let today: Days = 2;” 也没有语法错误。

默认情况下，枚举中的元素从 0 开始编号。同时也会对枚举值到枚举名进行反向映射。代码 2-11 给出了枚举值和枚举名之间的映射关系用法。

【代码 2-11】 枚举值和枚举名映射示例：enums2.ts

```
01  enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
02  console.log(Days["Sun"] === 0);      // true
03  console.log(Days["Mon"] === 1);      // true
04  console.log(Days["Tue"] === 2);      // true
05  console.log(Days["Wed"] === 3);      // true
06  console.log(Days["Sat"] === 6);      // true
07  console.log(Days[0] === "Sun");      // true
08  console.log(Days[1] === "Mon");      // true
09  console.log(Days[2] === "Tue");      // true
10  console.log(Days[6] === "Sat");      // true
```

可以根据实际情况，手动指定成员的索引数值（一般为整数，但是也可以是小数或负数）。例如，可以将上面的例子改成从 1 开始编号，枚举支持连续编号和不连续编号，也支持部分编号和部分不编号，如代码 2-12 所示。

【代码 2-12】 枚举索引编号示例：enums3.ts

```
01  enum Days {Sun = 1, Mon = 2, Tue = 4 , Wed = 3 , Thu =5, Fri, Sat};
02  console.log(Days["Sun"] === 1);      // true
03  console.log(Days["Mon"] === 2);      // true
04  console.log(Days["Tue"] === 4);      // true
05  console.log(Days["Wed"] === 3);      // true
06  console.log(Days["Thu"] === 5);      // true
07  console.log(Days["Fri"] === 6);      // true
08  console.log(Days["Sat"] === 7);      // true
```



给枚举类型进行手动赋值时，一定要注意手动编号和自动编号不要重复，否则会相互覆盖。

枚举手动编号和自动编号如果出现重复，那么重复的枚举名会指向同一个值，而这个数值只会返回最后一个赋值的枚举名。TypeScript 编译器并不会提示错误或警告。这种情况如代码 2-13 所示。

【代码 2-13】 枚举索引编号示例：enums4.ts

```
01  enum Days {Sun = 1, Mon = 2, Tue = 4 , Wed = 3 , Thu , Fri, Sat};
```

```

02 console.log(Days["Sun"] === 1);           // true
03 console.log(Days["Mon"] === 2);           // true
04 console.log(Days["Tue"] === 4);           // true
05 console.log(Days.Tue);                     // 4
06 console.log(Days.Thu);                     // 4
07 console.log(Days.Fri);                     // 5
08 console.log(Days.Sat);                     // 6
09 console.log(Days[4]);                       // Thu

```

在代码 2-13 的例子中，Wed = 3 后，并未手动进行编号，系统自动编号为递增编号，即 Thu 是 4、Fri 是 5、Sat 是 6。但是 TypeScript 并没有报错，导致 Days[4] 的值先是 Tue 而后又被 Thu 覆盖了。因此 Days[4] 的值为 Thu。

通常情况下，枚举名的赋值一般为数值，但是手动赋值的枚举名可以不是数字，如字符串。此时需要使用类型断言（这部分内容将在后续章节进行说明）来让 tsc 无视类型检查。代码 2-14 演示了枚举索引用字符串进行编号。

【代码 2-14】 枚举索引编号示例：enums5.ts

```

01 enum Days {Sun = <any>"S", Mon = 2, Tue = 4 , Wed , Thu , Fri, Sat};
02 console.log(Days["Sun"] === <any>"S"); // true
03 console.log(Days["Mon"] === 2);         // true
04 console.log(Days["Tue"] === 4);         // true
05 console.log(Days["S"]);                  //Sun

```

另外，在声明枚举类型时，可以在关键词 enum 前加上 const 来限定此枚举是一个常数枚举。常数枚举的示例见代码 2-15 所示。

【代码 2-15】 常数枚举示例：enums6.ts

```

01 const enum Directions {
02     Up,
03     Down,
04     Left,
05     Right
06 }
07 let directions: Directions = Directions.Up;
08 console.log(directions);           // 0

```

常数枚举与普通枚举的区别是，它会在编译阶段被删除。代码 2-15 如果用 tsc 编译成 JavaScript，内容如代码 2-16 所示。

【代码 2-16】 常数枚举编译 JavaScript 示例：enums6.js

```

01 var directions = 0 /* Up */;
02 console.log(directions);

```

代码 2-15 如果去掉 const 关键词，用 tsc 编译成 JavaScript，内容如代码 2-17 所示。

【代码 2-17】 普通枚举编译 JavaScript 示例：enums6_2.js

```
01  var Directions;
02  (function (Directions) {
03      Directions[Directions["Up"] = 0] = "Up";
04      Directions[Directions["Down"] = 1] = "Down";
05      Directions[Directions["Left"] = 2] = "Left";
06      Directions[Directions["Right"] = 3] = "Right";
07  })(Directions || (Directions = {}));
08  var directions = Directions.Up;
09  console.log(directions);
```

外部枚举（Ambient Enums）是使用 `declare enum` 定义的枚举类型，`declare` 定义的类型只会用于编译时的检查，编译成 JavaScript 后会被删除。因此，外部枚举与声明语句一样，常出现在声明文件（关于声明文件将在后续章节进行详细说明）中。外部枚举示例如代码 2-18 所示。

【代码 2-18】 外部枚举示例：enums7.ts

```
01  declare enum Directions {
02      Up,
03      Down,
04      Left,
05      Right
06  }
07  let directions : Directions = Directions.Up;
08  console.log(directions);
```

将代码 2-18 中的代码编译成 JavaScript 代码，内容如代码 2-19 所示。

【代码 2-19】 外部枚举编译成 JavaScript 示例：enums7.js

```
01  var directions = Directions.Up; //Directions 未定义
02  console.log(directions);
```



从代码 2-19 可以看出，外部枚举定义的枚举在生成 JavaScript 的时候会整段进行自动删除，从而出现 `Directions` 未定义的情况。

2.3.3 任意值

TypeScript 语言是一种静态类型的 JavaScript，可以更好地进行编译检查和代码分析等，但有些时候 TypeScript 需要和 JavaScript 库进行交互，这时就需要任意值（any）类型。

在某些情况下，编程阶段还不清楚要声明的变量是什么类型，这些值可能来自于动态的内容，比如来自用户输入或第三方代码库。这种情况下，我们不希望类型检查器对这些值进行检

查，此时可以声明一个任意值类型的变量。任意值类型示例如代码 2-20 所示。

【代码 2-20】 任意值示例：any.ts

```
01 let myVar: any = 7;
02 myVar= "maybe a string instead";
03 myVar= false;
```

从代码 2-20 可以看出，任意值变量 `myVar` 初始化值为数值 7，然后对其赋值字符串 "maybe a string instead" 和布尔值 `false`，都可以编译通过。

由于任意值类型允许我们在编译时可选择地包含或移除类型检查，因此在对现有代码进行改写的时候，任意值类型是十分有用的。但是由于 `any` 类型不让编译器进行类型检查，一般尽量不使用，除非必须使用它才能解决问题。



`any` 类型上没有任何内置的属性和方法可以被调用，它只能在运行时检测该属性或方法是否存在。因此声明一个变量为任意值之后，编译器无法帮助你进行类型检测和代码提示。

任意值类型和 `Object` 看起来有相似的作用，但是 `Object` 类型的变量只是允许你给它赋不同类型的值，但是却不能够在它上面调用可能存在的方法，即便它真的有这些方法。代码 2-21 给出了对比二者的示例。

【代码 2-21】 `any` 和 `Object` 对比示例：any_Object.ts

```
01 let notSure: any = 4;
02 notSure.ifItExists(); // ifItExist 方法在运行时可能存在
03 notSure.toFixed(); // toFixed 是数值 4 的方法
04 let prettySure: Object = 4; // 此处是大写的 Object，不是小写的 object
05 prettySure.toFixed(); // 错误 Object 类型没有 toFixed 方法
```

从代码 2-21 可以看出，`any` 类型的变量可以调用任何方法和属性，但是 `Object` 类型的变量却不允许调用此类型之外的任何属性和方法，即使 `Object` 对象有这个属性或方法也不允许。



代码 2-21 中 02 行在编码阶段是没有错误的，但是当编译成 JavaScript 运行时，就会报 `ifItExists` 方法不存在的错误。

另外，当只知道一部分数据的类型时，`any` 类型也是有用的。比如，你有一个列表，它包含了不同类型的数据，那么我可以用任意值数组来进行存储，如代码 2-22 所示。

【代码 2-22】 `any` 数组示例：any_array.ts

```
01 let list: any[] = [1, true, "free"];
02 list[1] = 100;
```



变量如果在声明的时候未明确指定其类型且未赋值，那么它会被识别为任意值类型。

TypeScript 会在没有明确地指定类型的时候推测出一个类型，这就是类型推论。如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 `any` 类型而完全不被编译器进行类型检查，如代码 2-23 所示。

【代码 2-23】 `any` 类型推论示例：any_infer.ts

```
01 let some;           //any 类型
02 some = 'Seven';
03 some = 7;
04 some.getName();
```

在代码 2-23 中，01 行只是声明了一个变量 `some`，但是并未明确指定其类型，也没有赋值，因此变量 `some` 会被推断为 `any` 类型。

2.3.4 空值、Null 与 Undefined

这里将空值（`void`）、`Null` 与 `Undefined` 放在一起介绍，主要是由于它们有容易混淆的地方。下面将依次对 `void`、`Null` 与 `Undefined` 类型进行详细说明。

1. 空值

空值（`void`）表示不返回任何值，一般在函数返回类型上使用，以表示没有返回值的函数。JavaScript 没有空值类型。在 TypeScript 中，可以用 `void` 关键词表示没有任何返回值的函数，如代码 2-24 所示。

【代码 2-24】 `void` 函数示例：void_func.ts

```
01 function hello():void{
02     console.log("void 类型");
03 }
```

`void` 一般都可以省略，从而简化代码。声明一个 `void` 类型的变量没有什么实际用途，因为你只能将它赋值为 `undefined` 和 `null`。而且一个 `void` 类型的变量也不能赋值到其他类型上（除了 `any` 类型以外），如代码 2-25 所示。

【代码 2-25】 `void` 变量示例：void_var.ts

```
01 let vu: void = null;
02 let vu2: void =undefined;
03 let num: number = vu;           // 错误, 不能将 void 赋值到 number
04 let num2: any= vu;              // 正确
```

2. null

`null` 表示不存在对象值。在 TypeScript 中，可以使用 `null` 关键词来定义一个原始数据类型，但要注意这本身没有实际意义。`null` 一般当作值来用，而不是当作类型来用。

```
let n: null = null; //无意义
```


`null` 类型的变量可以被赋值为 `null` 或 `undefined` 或 `any`，其他值不能对其进行赋值。代码 2-26 给出了 `null` 类型的变量用法。

【代码 2-26】 `null` 变量示例：`null_var.ts`

```
01 let uv: null = null;
02 let uv2: null = undefined;
03 let uv3: null = 2;           //错误
04 let a: any = 2 ;
05 uv = a ;
06 console.log(a);           //2
```

从代码 2-26 可以看出，`null` 既可以是数据类型也可以是值。`null` 类型的变量不能将数值 2 赋值给它，但是可以赋值 `any` 类型的变量，而 `any` 类型的变量可以赋值为 2。因此上述的 06 行输出 2。

3. `undefined`

`undefined` 表示变量已经声明但是尚未初始化变量的值。`undefined` 和 `null` 是所有类型的子类型。也就是说 `undefined` 和 `null` 类型的变量可以赋值给所有类型的变量。和 `null` 一样，在 TypeScript 中可以使用 `undefined` 来定义一个原始数据类型，但要注意这没有实际意义，`undefined` 一般当作值来用。

```
let n: undefined = undefined ;           //无意义
```

`undefined` 类型的变量可以被赋值为 `null` 或 `undefined` 或 `any`，其他值不能对其进行赋值。代码 2-27 给出了 `undefined` 类型的变量用法。

【代码 2-27】 `undefined` 变量示例：`undefined_var.ts`

```
01 let uv: undefined = undefined ;
02 let uv2: undefined = null;
03 let uv3: undefined = 3;           //错误
04 let a: any = 6;
05 uv = a ;
06 console.log(uv);           //6;
```

从代码 2-27 可以看出，`undefined` 既可以是数据类型也可以是值。`undefined` 类型的变量不能将数值 6 赋给它，但是可以赋值 `any` 类型的变量，而 `any` 类型的变量可以赋值为 6。因此代码 2-27 中的 06 行输出 6。

综上可知，声明一个 `void` 类型，`undefined` 类型和 `null` 类型的变量其实是没有什么意义的，因为你只能为它赋予 `undefined` 和 `null`，当然也可以是 `any`。

`void` 一般只用于函数返回值上，但是经常省略。`undefined` 和 `null` 一般都是当作值来使用的，`undefined` 表示变量已经声明但是未初始化变量的值，而 `null` 表示值初始化为 `null`。默认情况下，`null` 和 `undefined` 是所有类型的子类型。换句话说，你可以把 `null` 和 `undefined` 赋值

给任何类型的变量。



在编译时，如果开启了 `--strictNullChecks` 配置，那么 `null` 和 `undefined` 只能赋值给 `void` 和它们本身。这能避免很多常见的问题。

2.3.5 Never

`Never` 类型是其他类型（包括 `null` 和 `undefined`）的子类型，代表从不会出现的值。`Never` 类型只能赋值给自身，其他任何类型不能给其赋值，包括任意值类型。`Never` 类型在 TypeScript 中的类型关键词是 `never`。

`Never` 类型一般出现在函数抛出异常 `Error` 或存在无法正常结束（死循环）的情况下。代码 2-28 给出返回 `Never` 类型的函数示例。

【代码 2-28】 `Never` 类型的函数示例：`never.ts`

```
01 // 返回 never 的函数
02 function error(message: string): never {
03     throw new Error(message);
04 }
05 // 推断返回值类型为 never
06 function fail() {
07     return error("Something failed");
08 }
09 // 返回 never 的函数必须存在无法结束
10 function infiniteLoop(): never {
11     while (true) {
12     }
13 }
```

2.3.6 Symbols

自 ES6 引入了一种新的原始数据类型 `Symbol`，表示独一无二的值。它是 JavaScript 语言的第七种原始数据类型。

`Symbol` 类型的变量一旦创建就不可变更，且不能为它设置属性。`Symbol` 一般是用作对象的一个属性。即使两个 `Symbol` 声明的时候是同名的，也不是同一个变量，这样就能避免命名冲突的问题。

`Symbol` 类型的值是通过 `Symbol` 构造函数进行创建的。代码 2-29 给出了 `Symbol` 类型的变量声明示例。

【代码 2-29】 `Symbol` 类型的示例：`symbol.ts`

```
01 let s1 = Symbol('name'); // Symbol()
02 let s2 = Symbol('age');
```

```

03 console.log(s1)           // Symbol(name)
04 console.log(s2)           // Symbol(age)
05 console.log(s1.toString()) // "Symbol(name)"
06 console.log(s2.toString()) // "Symbol(age)"

```

Symbol 函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述。这个描述主要是为了在控制台显示，或者转为字符串时比较容易区分不同的 Symbol 变量。



Symbol 函数前不能使用 new 命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象。也就是说，由于 Symbol 值不是对象，因此不能添加属性。

Symbol 是不可改变且唯一的，Symbol 函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的 Symbol 函数的返回值是不相等的，如代码 2-30 所示。

【代码 2-30】 Symbol 类型的示例：symbol2.ts

```

01 // 没有参数的情况
02 let s1 = Symbol();
03 let s2 = Symbol();
04 console.log(s1 === s2) // false
05 // 有参数的情况
06 let s3 = Symbol('age');
07 let s4 = Symbol('age');
08 console.log(s3 === s4) // false

```

像字符串一样，Symbol 也可以被用作对象属性的键等。对象的属性名现在可以有两种类型，一种是字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型的就表示这个属性是独一无二的。这个特征可以保证不会与其他同名属性产生冲突。代码 2-31 给出了 Symbol 类型作为属性的示例。

【代码 2-31】 Symbol 类型作为属性的示例：symbol3.ts

```

01 let sym = Symbol("name");
02 let sym2 = Symbol("name");
03 let obj = {
04     [sym]: "value",
05     [sym2]: "value2",
06     name: "value3"
07 }; // 作为对象属性的键
08 console.log(obj);

```

代码 2-31 编译成 JavaScript 后在浏览器中运行可以打印出如下信息：

```
▼ Object ⓘ
  name: "value3"
  Symbol(name): "value"
  Symbol(name): "value2"
  ▶ __proto__: Object
```



调用 `obj[sym]` 时报错，提示 `Type 'symbol' cannot be used as an index type`，但是生成的 JavaScript 可以正确输出。

Symbol 值不能与其他类型的值进行运算，会报错，如代码 2-32 所示。

【代码 2-32】 Symbol 与其他类型运算示例：symbol4.ts

```
01 let s3 = Symbol('age');
02 let s4 = s3+"是 symbol";    //错误
03 console.log(s4)
```

Symbol 值可以显式转为字符串和布尔值，但是不能转为数值，如代码 2-33 所示。

【代码 2-33】 Symbol 显示转换示例：symbol5.ts

```
01 let s3 = Symbol('age');
02 let s4 = String(s3);        //Symbol(age)
03 let s5 = Boolean(s3);       //true
04 console.log(s4)
05 console.log(s5)
06 let s6 = Number(s3);        //错误
07 console.log(s6)
```

2.3.7 交叉类型

交叉类型（Intersection Types）可以将多个类型合并为一个类型。合并后的交叉类型包含了其中所有类型的特性。以经典的动画片葫芦娃来举例，每个葫芦娃都有自己的特长，7 个葫芦娃可合体为葫芦小金刚，他拥有所有葫芦娃的技能，非常强大。

下面假设有两个自定义类型，一个是 Car 类型，具备 `driverOnRoad` 功能；一个是 Ship 类型，具备 `driverInWater` 功能。我们通过交叉类型 `Car & Ship` 来合并功能，得到一个 `carShip` 对象。Car & Ship 是 Car 类型和 Ship 类型的交叉类型，如代码 2-34 所示。

【代码 2-34】 交叉类型示例：intersection_types.ts

```
01 class Car {
02     public driverOnRoad() {
03         console.log("can driver on road");
04     }
05 }
```

```

06  class Ship {
07      public driverInWater() {
08          console.log("can driver in water");
09      }
10  }
11  let car = new Car();
12  let ship = new Ship();
13  let carShip: Car & Ship = <Car & Ship>{};
14  carShip["driverOnRoad"] = car["driverOnRoad"];
15  carShip["driverInWater"] = ship["driverInWater"];
16  carShip.driverInWater();//can driver in water
17  carShip.driverOnRoad();//can driver on road

```

代码 2-34 例子中涉及类的相关知识，将在后续章节进行详细说明。这里读者不必细究。13 行创建了一个 `Car & Ship` 类型的变量 `carShip`，用 `{}` 进行了初始化，并用 `<Car & Ship>` 对空对象进行类型断言（将在 2.3.9 小节中进行详细说明），否则报错。

2.3.8 Union 类型

联合类型（Union Types）表示取值可以为多种类型中的一种。联合类型与交叉类型在用法上完全不同。

假设有一个 `padLeft` 函数，让它可以在某个字符串的左边进行填充。该函数有两个参数：一个是需要被填充的字符串，是字符类型；另一个是要填充的对象，可以是 `number` 类型或 `string` 类型。

如果传入 `number` 类型的填充对象，那么在字符串左边填充 `number` 个空格；如果传入 `string` 类型的填充对象，那么在字符串左边填充该字符串即可。

为了实现第二个参数 `padding` 既可以是 `number` 类型也可以是 `string` 类型的效果，需要使用联合类型。`number | string` 表示 `number` 和 `string` 的联合类型。下面的代码 2-35 给出了 `padLeft` 函数使用联合类型的示例。

【代码 2-35】 联合类型示例：union_types.ts

```

01  function padLeft(value: string, padding: number | string) {
02      if (typeof padding === "number") {
03          return Array(padding + 1).join(" ") + value;
04      }
05      if (typeof padding === "string") {
06          return padding + value;
07      }
08      throw new Error(`参数为 string 或 number, 但传入 '${padding}'`);
09  }
10  console.log(padLeft("Hello world", 3));           //   Hello world
11  console.log(padLeft("Hello world", "  "));        //  __ Hello world

```

```

12 //Argument of type 'true' is not assignable
13 //to parameter of type 'string | number'.
14 console.log(padLeft("Hello world", true)); // error

```

提示

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候, 只能访问此联合类型的所有类型里共有的属性或方法。

还上面的 Car 和 Ship 类型为例, 我们扩展一个同名的方法 toUpper, 联合类型 Car | Ship 的实例就只能调用它们共有的方法 toUpper, 而 driverOnRoad 和 driverInWater 不能调用, 具体示例如代码 2-36 所示。

【代码 2-36】 联合类型调用方法示例：union_types2.ts

```

01 class Car {
02     public driverOnRoad() {
03         console.log("can driver on road");
04     }
05     public toUpper(str: string) {
06         return str.toUpperCase();
07     }
08 }
09 class Ship {
10     public driverInWater() {
11         console.log("can driver in water");
12     }
13     public toUpper(str2: string) {
14         return str2.toUpperCase();
15     }
16 }
17 let car = new Car();
18 let ship = new Ship();
19 let carShip: Car | Ship = <Car | Ship>{};
20 carShip["driverOnRoad"] = car["driverOnRoad"];
21 carShip["driverInWater"] = ship["driverInWater"];
22 carShip["toUpper"] = ship["toUpper"];
23 let str: string = carShip.toUpper("hello world");
24 console.log(str); //共有方法
25 //carShip.driverOnRoad(); //不存在
26 //carShip.driverInWater(); //不存在
27 (<Car>carShip).driverOnRoad(); //OK
28 (<Ship>carShip).driverInWater(); //OK

```

提示

可以用类型断言将 Car | Ship 断言成一个 Car 或者 Ship 类型的对象, 从而调用特有的方法。

联合类型往往比较长，也不容易记忆和书写，我们可以用类型别名（Type Aliases）来解决这个问题。类型别名可以用来给一个类型起新名字，特别对于联合类型而言，起一个有意义的名字会让人更加容易理解。

类型别名的语法是：

```
type 类型别名 = 类型或表达式;
```

类型别名可以用于简单类型和自定义类型，也可以用于表达式。我们可以用 `type` 给表达式 `() => string` 起一个别名 `myfunc`；也可以用 `type` 给联合类型 `string | number | myfunc` 起一个别名 `NameOrStringOrMyFunc`。具体示例如代码 2-37 所示。

【代码 2-37】 类型别名示例：type_aliases.ts

```
01  type myfunc = () => string;
02  type NameOrStringOrMyFunc = string | number | myfunc;
03  function getName(n: NameOrStringOrMyFunc): string {
04      if (typeof n === 'string') {
05          return n;
06      }
07      else if (typeof n === 'number') {
08          return n.toString();
09      }
10      else {
11          return n();
12      }
13  }
14  let a :string = "hello";
15  let b: number = 999;
16  let c = function () {
17      return "hello my func";
18  }
19  console.log(getName(a));      //hello
20  console.log(getName(b));      //999
21  console.log(getName(c));      //hello my func
```

另外，还可以给内置类型起一个别名，如 `string`，如代码 2-38 所示。

【代码 2-38】 内置类型的类型别名示例：type_aliases2.ts

```
01  let newString = string ;
02  let a : newString = "new string type" ;
```



虽然可以用 `type` 给内置类型起别名，但为了防止混淆，不建议给字符、数值或布尔等类型起别名。

最后，符号 `|` 也可以用于定义字符串字面量类型。这种类型用来约束字符的取值只能是某几个字符串中的一个，如用 `type` 定义一个表示事件的字符串字面量类型 `EventNames`，并作为函数 `handleEvent` 的参数，这样此参数只能是 `'click'` 或 `'dbclick'` 或 `'mousemove'`，如代码 2-39 所示。

【代码 2-39】字符串字面量类型示例：string_literal_type.ts

```
01 type EventNames = 'click' | 'dbclick' | 'mousemove';
02 function handleEvent(ele: Element, event: EventNames) {
03     console.log(event);
04 }
05 let ele = document.getElementById('div'); //内置对象
06 handleEvent(ele, 'click');                // 没问题
07 handleEvent(ele, 'dbclick');              // 没问题
08 handleEvent(ele, 'mousemove');            // 没问题
09 handleEvent(ele, 'scroll');               // 不存在
```



类型别名与字符串字面量类型都是使用 `type` 进行定义的，注意二者的区别。

2.3.9 类型断言

类型断言（Type Assertion）可以用来手动指定一个值的类型。类型断言语法是：

- `<类型> 值或者对象`
- `值或者对象 as 类型`



在 `tsx` 语法（`React.jsx` 语法的 `ts` 版）中必须用后一种，因此 `<>` 有特殊意义。

类型断言一般和联合类型一起使用，可以将一个联合类型的变量指定为一个更加具体的类型进行操作，从而可以使用特定类型的属性和方法。代码 2-40 给出了类型断言的示例。

【代码 2-40】类型断言示例：type_assert.ts

```
01 function getLength(a: string | number): number {
02     //if ((a as string).length) {
03     if ((<string>a).length) {
04         return (<string>a).length;
05     } else {
06         return a.toString().length;
07     }
08 }
09 console.log(getLength(6));                //1
10 console.log(getLength("hello"));         //5
```


联合类型 `string | number` 限定参数 `a` 的类型，可以用类型断言 `<string>a` 指定类型为 `string`，从而可以调用字符的 `length` 属性，如果传入的是数值，那么会返回 `a.toString().length` 的值。类型断言成一个联合类型 `string | number` 中不存在的类型（如 `boolean`）是不允许的。

提示

类型断言不是类型转换，且类型推断不能直接进行调用，需要放于条件判断中或者先将其转化成 `unknown` 再进行类型断言，如 `console.log(<string><unknown>a).length)` 当 `a` 为数值时返回 `undefined`。

2.4 let 与 var

在 JavaScript 中定义变量一般都是用 `var` 关键词来声明，在 ES6 中引入 `let` 也可以声明变量。在 TypeScript 语言中，支持用 `var` 和 `let` 进行变量声明，但二者在变量声明上有着明显的区别。

通过 `var` 定义的变量，作用域是整个封闭函数，是全域的。通过 `let` 定义的变量，作用域是在块级或者子块中。因此，采用 `let` 声明变量更加安全，也更容易规避一些不易发现的错误。

在 JavaScript 中有一个变量提升机制，浏览器中 JavaScript 引擎在运行代码之前会进行预解析，首先将函数声明和变量声明进行解析，然后对函数和变量进行调用和赋值等，这种机制就是变量提升。代码 2-41 中包含 3 段代码，注意这 3 段代码应该分别运行，不要一起运行。

【代码 2-41】 变量提升示例：var_hosit.ts

```
01 //代码段 1-----
02 var myvar :string = '变量值';
03 console.log(myvar);           // 变量值
04 //代码段 2-----
05 var myvar :string = '变量值';
06 (function() {
07     console.log(myvar);       //变量值
08 }) ();
09 //代码段 3-----
10 var myvar :string = '变量值';
11 (function() {
12     console.log(myvar);       // undefined
13     var myvar :string = '内部变量值';
14 }) ();
```

在代码 2-41 中，代码段 1 会在控制台打印出“变量值”，这很容易理解；代码段 2 也会控制台打印出“变量值”，JavaScript 编译器首先在匿名函数内部作用域（Scope）查看变量 `myvar` 是否声明，发现没有就继续向上一级的作用域（Scope）查看是否声明 `myvar`，发现存在就打印出

该作用域的 `myvar` 值。代码段 3 只是对代码段 2 做一个微调，结果却输出了 `undefined`。

可理解为内部变量 `myvar` 在匿名函数内最后一行进行变量声明并赋值，但是 JavaScript 解释器会将变量声明（不包含赋值）提升（Hositing）到匿名函数的第一行（顶部），由于只是声明 `myvar` 变量，在执行 `console.log(myvar)` 语句时并未对 `myvar` 进行赋值，因此最终在控制台输出 `undefined`。

在 Javascript 语言中，变量的声明（注意不包含变量初始化）会被提升（置顶）到声明所在的上下文，也就是说，在变量的作用域内，不管变量在何处声明，都会被提升到作用域的顶部，但是变量初始化的顺序不变。

即使 `var` 声明的变量处于当前作用域的末尾，也会提升到作用域的头部并初始化为 `undefined`，在此之前都可以进行调用，并不会出现变量未定义的错误。



提示 `let` 声明的变量会进行变量提升，但是在作用域所在的顶部和 `let` 声明变量之前 `let` 声明的变量都无法访问，从而保证安全。

`let` 是 ES6 新增的变量声明方式，是用来替代 `var` 的设计，本节要介绍的就是它与 `var` 的不同。

2.4.1 `let` 声明的变量是块级作用域

`let` 声明的变量是块级作用域，大括号 `{}` 包围的区域是一个独立的作用域，如下面的代码 2-42 所示。

【代码 2-42】 变量 `let` 声明块级作用域示例：let1.ts

```
01  if (true) {
02      let msg = "hello";
03  }
04  console.log(msg);    //错误
```

在 `if` 块级作用域中用 `let` 声明一个 `msg` 变量，但是在块级作用域外不能访问此 `msg` 变量，如果将 `let` 换成 `var` 则可以在 `if` 块级作用域外进行访问。因此建议用 `let` 替代 `var` 进行变量声明，以提升代码的可读性和防止变量冲突。

2.4.2 `let` 不允许在同域中声明同名变量

`let` 声明的变量是块级作用域。在同一个作用域中，一旦 `let` 声明完一个变量后，就不允许再次声明一个同名的变量，即使用 `var` 进行声明也不可以，如代码 2-43 所示。

【代码 2-43】 同域中声明同名变量示例：let2.ts

```
01  //块变量不允许重名
02  let myvar: string = '变量值';
03  var myvar: string = "var 值";
04  console.log(myvar);    // 变量值
```



函数的参数和函数体属于同一个作用域，因此 `let` 命名的函数也不允许和参数名同名。

下面的代码 2-44 中的函数 `func` 中有一个参数 `arg`，和 02 行 `let` 声明的 `arg` 在同一个作用域，由于二者变量名相同，因此会报错。另外，在 TypeScript 中，函数名也不允许重复。

【代码 2-44】 `let` 函数中声明同名变量示例：let3.ts

```
01  function func(arg) {
02      let arg = 2;      //和参数 arg 重名
03  }
04  func("2");
05  //函数不能重名
06  function func(arg) {
07      {
08          let arg2 = arg + "2";
09      }
10  }
11  func("3") ;
```

2.4.3 `let` 禁止声明前访问

`let` 用死区 (temporal dead zone) 规避了变量提升带来的问题，因此也就无法在声明前对变量进行调用。在下面的代码 2-45 中，04 行用 `let` 声明了一个变量 `tmp`，那么在 `if` 块作用域中，03 行访问变量 `tmp` 会报错。并且，`let` 声明的变量生命周期仅在块作用域中，不会污染外部的变量 `tmp`，因此 06 行打印的仍然是 123。

【代码 2-45】 `let` 声明禁止声明前访问示例：let4.ts

```
01  var tmp = 123;
02  if (true) {
03      tmp = 'abc';      //块作用域变量 tmp 在声明之前无法调用
04      let tmp;
05  }
06  alert(tmp);          //输出值为 123，全局 tmp 与局部 tmp 不影响
```

为什么需要块级作用域？`var` 创建的变量只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景，这种方式往往让代码难于让人理解其意图。概括起来，`var` 声明的变量往往有如下问题：

(1) 内层变量可能会覆盖外层变量

下面的代码 2-46 按照一般理解会打印出“外部变量”，但实际情况是打印出了 `undefined`。由于 `var` 不是块级作用域，再加上变量提升机制 (`var msg = undefined` 提升到 02 行和 03 行之间)，在调用 `func` 时，首先将 `msg` 赋值为 `undefined`，然后打印出值，导致内层的 `msg` 变量

覆盖了外层的 msg 变量。

【代码 2-46】 var 内层变量覆盖外层变量示例：var1.ts

```
01 var msg : string = "外部变量";
02 function func() {
03     console.log(msg);
04     if (false) {
05         var msg : string = '内部变量';
06     }
07 }
08 func(); // undefined
```

(2) 用来计数的循环变量泄露为全局变量

for 循环有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域，如代码 2-47 所示。

【代码 2-47】 var for 循环变量泄露为全局变量示例：var2.ts

```
01 var msg:string = 'hello wolrd';
02 for (var i = 0; i < msg.length; i++) {
03     console.log(msg[i]);
04 }
05 console.log(i); // 11
```

上面的代码 2-47 中，变量 i 只用来控制循环，但是循环结束后它并没有消失，泄露成了全局变量。let 声明的变量就不会出现这种问题。for 循环中用 let 可以避免循环变量泄露为全局变量的问题，如代码 2-48 所示。

【代码 2-48】 let for 循环变量不会泄露为全局变量示例：var3.ts

```
01 var msg:string = 'hello wolrd';
02 for (let i = 0; i < msg.length; i++) {
03     console.log(msg[i]);
04 }
05 console.log(i); //报错
```

let 允许块级作用域的任意嵌套，外层作用域无法读取内层作用域的变量，且内层作用域可以定义外层作用域的同名变量，如代码 2-49 所示。

【代码 2-49】 变量 var 声明示例：var4.ts

```
01 {
02     let n: number = 9;
03     {
04         let msg:string = 'Hello World';
05         let n: number = 10; //不同块级作用域可以同名
06     }
07     console.log(msg); //报错，无法找到 msg
08 };
```



let 块级作用域的出现，实际上使得获得广泛应用的立即执行函数表达式 (IIFE) 显得没有那么必要了。

2.5 变量

变量是一种占位符，用于引用计算机内存地址。变量是方便人来存取数据的，而内存地址是方便计算机来存取数据的。可以把变量看作存储数据的容器。类型分为值类型和引用类型，所以变量可分为值类型变量和引用类型变量。

JavaScript 中的数据类型主要包括两种，一种是基本类型（值类型），另一种是引用类型（引用类型）。内存分为两个部分，栈内存和堆内存。基本类型值保存在栈内存中，引用类型值在堆内存中保存着对象、在栈内存中保存着指向堆内存的指针。

JavaScript 中，基本类型值包括 `undefined`、`null`、`number`、`string` 和 `boolean`，在内存中分别占有固定大小的空间。引用类型值只有 `object`，这种值的大小不固定，可以动态添加属性和方法，而基本类型则不可以。

TypeScript 和 JavaScript 类似。对基本类型值进行复制，复制的是值本身，相当于复制了一个副本，修改一个变量的值不会影响另外一个变量的值。而对引用类型值进行复制，复制的是对象所在的内存地址。所以两者指向的都是栈内存中的同一个数据，修改一个变量会导致另外一个变量的值也进行修改。

理解变量的值类型和引用类型是非常重要的。为了让读者更加直观地了解值类型和引用类型变量的核心区别，下面用一张 C# 语言的示例图来说明值类型和引用类型的差异，如图 2.1 所示。

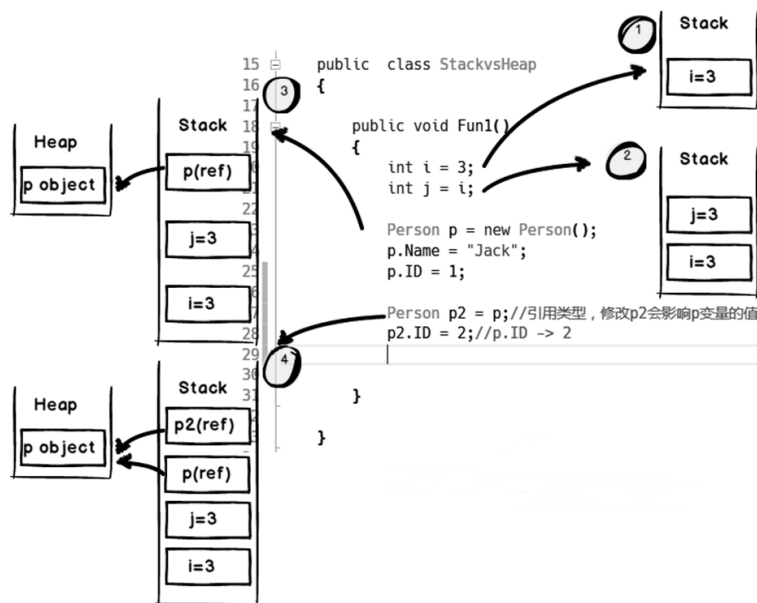


图 2.1 C#中的值类型和引用类型示意图

在图 2.1 中，由于 `int j=i` 中是 `int` 类型，为值类型变量，因此 `j` 变量的值是 `i` 变量值的副本，值都在 Stack 内存中，都是 3，但是二者不是同一个对象。因此，修改 `i` 不会修改 `j`，修改 `j` 也不会修改 `i`。二者是独立的。

在 `Person p2=p` 语句中, `Person` 为类, 是引用类型, `p2` 和 `p` 指向同一个 `Heap` 地址块, 因此修改 `p2` 的值会影响 `p` 的值。



在 JavaScript 中, 函数的参数传递是按值传递, 而且不能按引用传递。

在 TypeScript 中, 字符串、布尔型和数值型都是值类型, 而类、数组和元组等都是引用类型。从下面的代码 2-50 可以看出, 数组是引用类型, 变量 `a` 和 `b` 指向同一个内存地址, 修改了 `b` 的值, 同时也修改了 `a` 的值。

【代码 2-50】 引用类型示例: `ref_var.ts`

```
01 let a = [1,2];
02 let b = a;           //数组是引用类型
03 b[2] = 3;
04 console.log(a);      //[1,2,3]
05 console.log(b);      //[1,2,3]
```

值类型就不是这样的, 如字符串和数值类型。从下面的代码 2-51 可以看出, 字符类型是值类型变量, 变量 `a` 和 `b` 指向的是不同的内存地址, 只是值一开始一致而已, 修改了 `b` 的值, 不会修改 `a` 的值。

【代码 2-51】 值类型示例: `value_var.ts`

```
01 let a = "1,2";
02 let b = a;
03 b= b+",3";
04 console.log(a); //1,2
05 console.log(b); //1,2,3
```

2.5.1 声明变量

在 ES5 中声明变量的方法最常用的就是 `var`。在 ES6 中, 添加了 `let` 和 `const` 进行变量声明。在 ES6 环境下, 一般的变量声明都采用 `let`, 而不建议使用 `var`。

变量的命名一般都是有约定的。在 TypeScript 中, 变量命名必须满足如下规则:

- 变量名称可以包含数字和字母, 如 `stuName01`。
- 除了下划线 `_` 和美元 `$` 符号外, 不能包含其他特殊字符, 包括空格, 如 `_stuName` 和 `$tmp` 都是合法的变量名。
- 变量名不能以数字开头, 如 `9Num` 是错误的。



TypeScript 是区分大小写的, 比如 `numA` 和 `NumA` 不同。

在 TypeScript 编码规范中, 建议在使用前变量一定要先声明。变量声明可以使用以下 4 种方式:

(1) `[var 或 let 或 const][变量名]:[类型] = 值;`

此范式进行变量声明,同时指定了声明变量的类型及初始值。代码 2-52 分别给出了用 `var`、`let` 和 `const` 声明变量的方式。

【代码 2-52】 变量声明并初始化的示例：declare_var1.ts

```
01 var uname:string = "JackWang";
02 let uname2:string = "JackWang";
03 const version:string = "1.0";
```



`const` 声明的常量变量一定要初始化,否则会报错。

(2) `[var 或 let][变量名]:[类型];`

此范式进行变量声明,只指定了声明变量的类型,初始值默认为 `undefined`。代码 2-53 分别给出了用 `var` 和 `let` 声明变量的方式。

【代码 2-53】 变量声明不初始化的示例：declare_var2.ts

```
01 var uname:string ;
02 let uname2:string;
```

(3) `[var 或 let][变量名];`

此范式进行变量声明,只提供了变量名,声明变量的类型和初始值都未提供。变量类型默认为 `any`,变量值默认为 `undefined`,如代码 2-54 所示。

【代码 2-54】 变量声明未提供类型和初始值示例：declare_var3.ts

```
01 var uname;
02 let uname2;
```

(4) `[var 或 let 或 const][变量名] = 值;`

此范式进行变量声明,未指定声明变量的类型,但是给出了初始值,这时会用类型推断来确定变量的类型,这种写法更加简洁,如代码 2-55 所示。

【代码 2-55】 变量声明只给出初始值的示例：declare_var4.ts

```
01 var uname = "JackWang";    //string
02 let uname2 = 100 ;         //number
03 const version = "1.0";     //string
```

2.5.2 变量的作用域

变量的作用域就是定义的变量可以使用的代码范围。变量可以分为全局变量和局部变量。在日常的程序开发中,尽量少用全局变量,防止变量冲突。局部变量只在块作用域和函数体内有效,从而保证变量的安全访问。

程序中变量的可用性由变量作用域决定。TypeScript 有以下几种作用域:

- 全局作用域：全局变量定义在程序结构的外部，可以在代码的任何位置使用。
- 类作用域：这个变量也可以称为字段。类变量声明在一个类里，但在类的方法外面无法访问。该变量可以通过类的实例对象来访问。类变量也可以是静态的，可以通过类名直接访问。
- 局部作用域：局部变量，只能在声明它的一个代码块（如方法）中使用。

代码 2-56 说明了 3 种作用域的使用。此示例涉及类的相关知识点，但读者此时不要过多在意类的语法，这些会在后面的章节详细进行说明。此时只需理解变量作用域的相关知识点即可。

【代码 2-56】 变量作用域示例：scope_var.ts

```
01  var global_var = 12                // 全局变量
02  class MyClass {
03      clazz_val = 13;                // 类变量
04      static sval = 10;              // 静态变量
05      storeNum(): void {
06          var local_var = 14;        // 局部变量
07      }
08  }
09  console.log("全局变量为: " + global_var);
10  console.log(MyClass.sval);         // 静态变量
11  var obj = new MyClass();
12  console.log("类变量: " + obj.clazz_val);
```



var 在函数或方法中声明变量，作用域限于此函数或方法中。

2.5.3 const 声明变量

const 和 let 在声明变量的用法上基本一致，只是 const 声明的变量被赋值后不能再改变，是只读的常量。因此，对于 const 声明的变量来说，只声明不赋值就会报错。

const 声明的变量作用域和 let 一致，但是在某些情况下用 const 更加安全，可以防止在其他地方被修改从而影响程序的正常运行，如代码 2-57 所示。

【代码 2-57】 const 声明变量示例：const_var.ts

```
01  const cVar = "hello";
02  cVar = "change";                // 错误，不能赋值
03  console.log(cVar);
```

const 实际上保证的并不是变量的值不能改动，而是变量指向的那个内存地址所保存的数据不能改动。对于简单类型的数据（如数值、字符串和布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。

对于复合类型的数据（主要是对象和数组），变量指向的是内存地址，保存的只是一个指向实际数据的指针，`const` 只能保证这个指针是固定的（总是指向一个固定的地址），至于它指向的数据结构是不是可变的，则完全不能控制。因此，将一个对象声明为常量类型，其对象的值也是可以修改的，如代码 2-58 所示。

【代码 2-58】 `const` 声明复合类型变量示例：const_var.ts

```
01 interface Object{
02     prop: string;
03     func: () => string;
04 }
05 const foo: Object = {};
06 // 为 foo 添加一个属性可以成功
07 foo.prop = "123";           //说明值可以修改
08 foo.func = function (): string {
09     return "hello";
10 }
11 // 将 foo 指向另一个对象就会报错
12 foo = {}; //报错
13 const a = [];
14 a.push('Hello');           // 可执行
15 a.length = 0;              // 可执行
16 a = ['Dave'];              // 报错
```

代码 2-58 涉及接口的相关知识点，此示例为了说明 `const` 声明的常量对象本身的值是可以修改的，必须借助接口来扩展 `Object` 的属性或方法。

在代码 2-58 中，常量 `foo` 储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把 `foo` 指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。



在 TypeScript 中，`Object` 类型的变量不能动态添加属性和方法，而只能通过接口扩展属性或者方法。

在代码 2-58 中，13 行声明了一个数组常量 `a`，这个数组中的相关属性是可以修改的，但是如果将另一个数组赋值给 `a` 就会报错。如果真的想将对象冻结，应该使用 `Object.freeze` 方法，如代码 2-59 所示。

【代码 2-59】 `const` 声明数值变量冻结示例：const_freeze.ts

```
01 const foo = Object.freeze([]);
02 foo.length = 1;           //报错
```

2.6 运算符

运算符定义了将在数据上执行的某些功能。例如，在表达式 $8 + 7 = 15$ 中， $+$ 和 $=$ 就是一种运算符（算术运算符）。TypeScript 中的主要运算符可归类为：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 条件运算符
- 字符串操作符
- 类型运算符

不同的运算符可以组合使用，但是各类型的运算符的优先级不同：关系运算符的优先级低于算术运算符；关系运算符的优先级高于赋值运算符；单目运算优于双目运算；先算术运算，后移位运算，再按位运算，逻辑运算最后结合。例如， $1 << 3 + 2 \& 7$ 等价于 $(1 << (3 + 2)) \& 7$ 。



当不同运算符组合在一起的时候，用括号()将先计算的括起来，这样可以提高代码可读性。

2.6.1 算术运算符

算术运算符（arithmetic operators）就是用来处理四则运算的符号，是最简单、最常用的符号。尤其是数字的处理，几乎都会使用到算术运算符。



自增和自减运算符只能用于操作变量，不能直接用于操作数值或常量！例如， $5++$ 和 $8--$ 等写法都是错误的。

另外需要注意的是 $a++$ 和 $++a$ 的异同点，它们的相同点都是给变量 a 加 1，不同点是 $a++$ 是先参与程序的运行再加 1，而 $++a$ 则是先加 1 再参与程序的运行。因此，如果 $a=8$ ，那么 $\text{console.log}(a++)$ 会打印出 8，而 $\text{console.log}(++a)$ 会打印出 9，但最后 a 的值都为 9。

表 2.1 给出了 TypeScript 中算术运算符的具体说明和示例。

表 2.1 算术运算符说明

运算符	描述	示例
+	加法，返回操作数的总和	<pre>let a : number = 10; let b : number = 2; let c : number = a + b; console.log(c); // 12</pre>

(续表)

运算符	描述	示例
-	减法, 返回值的差	<pre>let a : number = 10; let b : number = 2; let c : number = a - b; console.log(c); // 8</pre>
*	乘法, 返回值的乘积	<pre>let a : number = 10; let b : number = 2; let c : number = a * b; console.log(c); // 20</pre>
/	除法, 执行除法运算并返回商	<pre>let a : number = 10; let b : number = 2; let c : number = a / b; console.log(c); // 5</pre>
%	取余, 执行除法并返回余数	<pre>let a : number = 9; let b : number = 2; let c : number = a % b; console.log(c); // 1</pre>
++	递增, 将变量的值增加 1	<pre>let a : number = 9; let c : number = a ++; console.log(c); // 10</pre>
--	递减, 将变量的值减少 1	<pre>let a : number = 9; let c : number = a -- ; console.log(c); // 8</pre>

2.6.2 关系运算符

关系运算符用于确定两个实体对象之间的关系类型, 有小于、小于等于、大于、等于、大于等于和不等于 6 种。关系运算符返回一个 boolean 值, 即 true 或者 false。TypeScript 中关系运算符的具体说明和示例如表 2.2 所示。

表 2.2 关系运算符说明

运算符	描述	示例
>	大于	<pre>let a : number = 10; let b : number = 2; console.log(a > b); // true</pre>
<	小于	<pre>let a : number = 10; let b : number = 2; console.log(a < b); // false</pre>
>=	大于等于	<pre>let a : number = 10; let b : number = 2; console.log(a >= b); // true</pre>

(续表)

运算符	描述	示例
<code><=</code>	小于等于	<pre>let a : number =10; let b : number =2; console.log(a<=b); //false</pre>
<code>==</code>	等于	<pre>let a : number =10; let b : number =2; console.log(a==b); //false</pre>
<code>!=</code>	不等于	<pre>let a : number =10; let b : number =2; console.log(a!=b); //true</pre>

2.6.3 逻辑运算符

逻辑运算符用于组合两个或多个条件。逻辑运算符也返回一个 `boolean` 值。逻辑运算符一般和关系运算符配合使用，多用于 `if` 条件判断和循环中断条件等场景。TypeScript 中逻辑运算符的具体说明和示例如表 2.3 所示。

表 2.3 逻辑运算符说明

运算符	描述	示例
<code>&&</code>	仅当指定的所有表达式都返回 <code>true</code> 时，运算符才返回 <code>true</code>	<pre>let a : number =10; let b : number =2; console.log(a>9 && b>7); //false</pre>
<code> </code>	如果指定的表达式至少有一个返回 <code>true</code> ，则运算符返回 <code>true</code>	<pre>let a : number =10; let b : number =2; console.log(a>9 b>7); //true</pre>
<code>!</code>	运算符返回相反的表达式结果	<pre>let a : number =10; let b : number =2; console.log(! (a>9 && b>7)); //true</pre>

2.6.4 按位运算符

按位运算符用来对二进制位进行操作。位运算符一般用于数值，在具体运算时要先将十进制转成二进制后再计算。TypeScript 中按位运算符的具体说明和示例如表 2.4 所示。

表 2.4 按位运算符说明

运算符	描述	示例
<code>&</code>	按位与，对其整数参数的每一位执行“与”运算	<pre>let a : number = 2 ; // 10 let b : number = 3 ; // 11 console.log(a & b); // 2</pre>

(续表)

运算符	描述	示例
	按位或，对其整数参数的每一位执行“或”运算	let a : number = 2 ; // 10 let b : number = 3 ; // 11 console.log(a & b); // 3
^	按位异或，对其整数参数的每一位执行“异或”运算。异或意味着操作数 1 为 true 或操作数 2 为 true 但两者不能同时为 true	let a : number = 2 ; // 10 let b : number = 3 ; // 11 console.log(a ^ b); // 1
~	按位取反，这是一个一元运算符，并通过取反操作数中的所有位进行操作	let a : number = 2 ; // 10 let b : number = 3 ; // 11 console.log(~b); // -4 console.log(~a); // -3
<<	左移，通过在第二个操作数指定的位数将第一个操作数中的所有位向左移动。新位用零填充。将一个值左移一个位置相当于将其乘以 2，移位两个位置相当于乘以 4，以此类推	let a : number = 2 ; // 10 let b : number = 3 ; // 11 console.log(1<<b); // 8 console.log(2<<a); // 8
>>	右移，二进制右移运算符。左操作数的值是由右操作数指定的位数来向右移动	let a : number = 2 ; // 10 let b : number = 3 ; // 11 console.log(a>>1); // 1 console.log(b>>2); // 0
>>>	无符号右移，这个运算符就像>>运算符一样，只不过在左边移入的位总是为零	let a : number = 2 ; // 10 let b : number = 3 ; // 11 console.log(a>>>1); // 1 console.log(b>>>2); // 0

2.6.5 赋值运算符

基本的赋值运算符是=。它的优先级低于其他的运算符，所以对该运算符往往最后读取。TypeScript 中赋值运算符的具体说明和示例如表 2.5 所示。

表 2.5 赋值运算符说明

运算符	描述	示例
=	简单的赋值，将值从右侧操作数赋给左侧操作数	let a : number = 2 ; let c: number = 3; c = a; console.log(c); // 2
+=	加法赋值，将右操作数添加到左操作数并将结果赋给左操作数	let a : number = 2 ; let c: number = 3; c += a;//c = c + a; console.log(c); // 5

(续表)

运算符	描述	示例
<code>-=</code>	减法赋值，从左操作数中减去右操作数，并将结果赋给左操作数	<pre>let a : number = 2 ; let c: number = 3; c -= a;//c = c - a; console.log(c); //-1</pre>
<code>*=</code>	乘法赋值，将右操作数与左操作数相乘，并将结果赋给左操作数	<pre>let a : number = 2 ; let c: number = 3; c *= a;//c = c * a; console.log(c); // 6</pre>
<code>/=</code>	除法赋值，将左操作数除以右操作数，并将结果赋给左操作数	<pre>let a : number = 2 ; let c: number = 3; c /= a;//c = c / a; console.log(c); // 1.5</pre>

2.6.6 等号运算符

等号运算符 `===` 和 `==` 都可以用于判断两个对象是否相等，但是具体细节上不同。`==` 在比较的时候会进行自动数据类型转换。而`===`是严格比较，不会进行自动转换，要求进行比较的操作数必须类型和值一致，不一致时返回 `false`，如代码 2-60 所示。

【代码 2-60】 等号运算符示例：equal_opt.ts

```
01 let a: number = 1;
02 let b :any = "1";
03 console.log(a == b);           //true
04 console.log(a === b);         //false
```

2.6.7 否定运算符 (-)

否定运算符 `-` 可以更改值的符号。举一个例子：5 应用否定运算符为 `-5`。否定运算符的基本用法如代码 2-61 所示。

【代码 2-61】 否定运算符示例：neg_opt.ts

```
01 let a: number = 1;
02 let b: number = - a;
03 console.log(b);           // -1
```



两个连续的否定运算符可以抵消，但不能直接用`-5`，而是用`-(-5)`，结果为 5。

2.6.8 连接运算符 (+)

连接运算符一般用于字符串拼接。用于字符串时的 + 运算符将第二个字符串追加到第一个字符串上。连接运算符还可以连接字符和数字、字符和数组以及字符和布尔等。这些不同的连接在结果上也是有差异的。代码 2-62 给出了连接运算符的相关用法。

【代码 2-62】 连接运算符示例：join_opt.ts

```
01 let a: string = "hello";
02 let b: string = "world";
03 let c: string = a + " " + b;
04 console.log(c);           // hello world
05 let arr = [1, 2, 3];
06 console.log("" + arr);    //"1,2,3"
07 console.log(""+ true);    // "true"
08 console.log("" + null);   //"null"
```



连接运算符和算术运算符中的+不同，连接运算符中必须有一个是字符串。5+"6"是"56"；而 5+6=11。

2.6.9 条件运算符 (?)

条件运算符用来表示一个条件表达式。条件运算有时也被称为三元运算符，基本语法如下：

条件表达式 ? 条件表达式为 true 时的值 : 条件表达式为 false 时的值

代码 2-63 给出了条件运算符的基本用法。

【代码 2-63】 条件运算符示例：condition_opt.ts

```
01 let a: number = 10;
02 let c: string = a>9 ? "大于 9" : "小于等于 9";
03 console.log(c);           // 大于 9
```

在代码 2-63 中，第 02 行检查变量 a 中的值是否大于 9。如果 a 设置为大于 9 的值，就返回字符串“大于 9”，否则返回字符串“小于等于 9”。由于变量 a 是 10，10>9 为 true，因此返回第一个字符串“大于 9”。



条件运算符可以替换简单的 if... else 语句，让代码更加简洁。

2.6.10 类型运算符 (typeof)

typeof 操作符返回一个字符串，用以获取一个变量或者表达式的类型。typeof 运算符一般只能返回如下几个结果：number, boolean, string, symbol, function, object 和 undefined。

表 2.6 给出常见的类型对象的 `typeof` 返回值。

表 2.6 常见类型对象的 `typeof` 返回值

变量类型	示例	typeof 返回值
number	<code>typeof 2</code>	"number"
boolean	<code>typeof true</code>	"boolean"
string	<code>typeof "hello"</code>	"string"
null	<code>typeof null</code>	"object"
undefined	<code>typeof undefined</code>	"undefined"
function	<code>typeof JSON.stringify</code>	"function"
array	<code>typeof [1,2]</code>	"object"
object	<code>typeof {}</code>	"object"
enum	<code>typeof Colors;</code>	"object"
enum	<code>typeof Colors.Red</code>	"number"
class	<code>typeof Person</code>	"function"
tuple	<code>typeof [2,"hello"]</code>	"object"

为了验证结果，用下面的代码 2-64 来查看不同类型的变量上用 `typeof` 输出的结果。

【代码 2-64】 类型运算符示例：`typeof.ts`

```

01  let a: number = 2;
02  console.log(typeof a);    //"number"
03  let b: string = "hello";
04  console.log(typeof b);    //"string"
05  let c: boolean = true;
06  console.log(typeof c);    //"boolean"
07  let d = null;
08  console.log(typeof d);    //"object"
09  console.log(typeof undefinedVar);    //未定义 undefined
10  enum Colors {
11      Red,
12      Green,
13      Yellow
14  }
15  let color: Colors = Colors.Red;
16  console.log(typeof color); //"number"
17  let f = function () {
18      console.log("hello world");
19  };
20  console.log(typeof f);    //"function"
21  let g = [];
22  console.log(typeof g);    //"object"
23  let m: number[] = [1, 2];

```



```
24 console.log(typeof m); // "object"
```



let 声明的变量在声明之前不可用 `typeof` 来输出操作数的类型。

代码 2-65 给出一个错误的示范。01 行用 `typeof a` 查看 `a` 变量的类型，但是由于 `a` 类型是 `let` 声明的，因此不能在声明之前进行调用。03 行用 `typeof b` 查看变量 `b` 的类型，由于 `b` 是用 `var` 声明的，会进行变量提升，因此输出 `undefined` 而没有报错。

【代码 2-65】 类型运算符示例 2：typeof2.ts

```
01 console.log(typeof a); //声明之前不能调用
02 let a: number = 2;
03 console.log(typeof b); //undefined
04 var b: number = 2;
```

2.6.11 instanceof 运算符

`instanceof` 运算符可用于测试对象是否为指定类型的实例，如果是，那么返回的值为 `true`，否则返回 `false`。`instanceof` 运算符的基本语法为：

类实例 `instanceof` 类

下面定义了一个 `People` 类，其中有两个私有属性 `name` 和 `age`，如代码 2-66 所示。05 行用 `new People` 创建了一个类的实例 `man`，06 行 `man instanceof People` 则返回 `true`。

【代码 2-66】 instanceof 运算符示例：instanceof.ts

```
01 class People {
02     private name: string = "";
03     private age: string = "";
04 }
05 let man: People = new People ();
06 alert(man instanceof People ); //true
```



`instanceof` 左边的只能是 `any` 类型或者对象类型或者类型参数 (type parameter)。其他不能用，如 `"hello" instanceof string` 报错。

2.6.12 展开运算符 (...)

展开运算符 (spread operator) 允许一个表达式在某处展开。展开运算符在多个参数 (用于函数调用) 或多个元素 (用于数组字面量) 或者多个变量 (用于解构赋值) 的地方可以使用。合理使用展开运算符可以使代码更加简洁。展开运算符为 `"..."`。

展开运算符 `"..."` 主要有以下应用场景：

(1) 函数动态参数

在 TypeScript 中可以定义 add 函数，其参数为...args（args 是 rest 剩余参数，后面章节再详细介绍）可以允许传入任意个数的参数。在 add 函数内部，用 for...of 循环将传入的参数求和并返回值，如代码 2-67 所示。

【代码 2-67】 展开运算符示例：spread_opt1.ts

```
01 function add(...args) {
02     let sum = 0;
03     for (let item of args) {
04         sum += item;
05     }
06     return sum;
07 }
08 let arr = [1, 2, 3, 4];
09 //let args = add(arr);    //"01,2,3,4"
10 let args = add(...arr);
11 console.log(args);        //10
```



如果调用 add(arr)，那么返回的结果不是 10。

在代码 2-67 中，在调用函数时先声明了一个数值型数组 arr，将 add(...arr) 进行传入即可获取数组元素的累加值 10。

(2) 数组合并

假设有两个数组，那么用展开运算符可以很方便地进行合并。展开运算符对数组进行合并的语法相当简洁，如代码 2-68 所示。

【代码 2-68】 展开运算符数组合并示例：spread_opt2.ts

```
01 let arr = [1, 2, 3];
02 let args = [...arr, 4, 5];
03 console.log(args);    // [1,2,3,4,5]
```

(3) 复制数组

由于数组是按照引用传递值的，因此要想复制一个数组的副本，不能直接赋值。此时用展开运算符可以很方便地进行数组备份，如代码 2-69 所示。

【代码 2-69】 展开运算符复制数组示例：spread_opt3.ts

```
01 let arr = [1, 2, 3];
02 let arr2 = [...arr];    //和 arr.slice() 一致
03 arr2.push(4);
04 console.log(arr);       //[1,2,3]
05 console.log(arr2);      //[1,2,3,4]
```

（4）解构赋值

展开运算符在解构赋值中的作用是将多个数组项组合成一个新数组。不过要注意，解构赋值中展开运算符只能用在末尾，如代码 2-70 所示。

【代码 2-70】 展开运算符解构赋值示例：spread_opt4.ts

```
01 let [a, b, ...arg3] = [1, 2, 3, 4];
02 console.log(a);           //1
03 console.log(b);           //2
04 console.log(arg3);        //[3,4]
```

（5）ES7 草案中的对象展开运算符

可以将对象当中的一部分属性取出来生成一个新对象赋值给展开运算符的参数。例如，下面代码 2-71 中的 `args` 值就为一个新的对象 `{b:4,c:3}`。

【代码 2-71】 展开运算符对象解构示例：spread_opt5.ts

```
01 let { y, a, ...args } = { a: 1, y: 2, c: 3, b: 4 };
02 console.log(a);           //1 获取对象 a 属性
03 console.log(y);           //2 获取对象 y 属性
04 console.log(args);        //{b:4,c:3}
```

2.7 数字

数字是一种用来表示数的书写符号。不同的记数系统可以使用相同的数字。数字在复数范围内可以分实数和虚数，实数又可以划分有理数和无理数或分为整数和小数，任何有理数都可以化成分数形式。

一般在现实生活中，常用的数字进制是十进制，计算机内部是基于二进制的。不同进制的数可以相互换算。数学之美，可以用数字来抽象现实的事物发展规律，从而指导现实生活，为人类服务。

在一个应用程序中，数字和字符串一样，是非常常用的一种数据类型，在一些财务或工程领域的软件中更显重要。

在 TypeScript 中，可以用 `number` 来定义一个数值类型的变量。另外，可以利用 `number` 的类型封装对象 `Number`，调用其构造函数来创建一个 `Number` 对象，它的值是 `number` 类型的，如代码 2-72 所示。

【代码 2-72】 Number 基本用法示例：Number.ts

```
01 let a: number = 2.8;
02 // 'number' is a primitive, but 'Number' is a wrapper object
03 // let b: number = new Number("2");    // 错误
04 let c: Number = new Number("2");
```

```

05  alert(c);        //2
06  let d: Number = new Number(3);
07  alert(d);        //3
08  let e: Number = new Number(true);
09  alert(e);        //1
10  let f: Number = new Number(false);
11  alert(f);        //0
12  let g: Number = new Number("true");
13  alert(g);        //NaN
14  let h: Number = new Number({});
15  alert(h);        //NaN
16  let m: Number = new Number(6);
17  alert(m);        //6

```

可以用 `let` 或者 `var` 进行数值变量的声明。在代码 2-72 中，01 行中用 `let` 声明了一个数值变量 `a`，并初始化其值为 2.8。04 行用 `new Number("2")` 构造函数来创建值为 2 的 `Number` 对象。16 行用 `new Number(6)` 构造函数也可以创建值为 6 的 `Number` 对象。`new Number(true)` 和 `Number(false)` 分别返回值为 1 和 0 的 `Number` 对象。



传入的参数将在 `Number` 构造函数中被转换到 `number`，如果转换失败，就返回 `NaN`，否则返回转换的值为 `number` 类型的 `Number` 对象。

2.7.1 Number 的属性

数值作为一个常用的对象，其中也有一些常用的属性来说明对象的相关信息。例如，数值本身是有最大值和最小值的。表 2.7 列出了一组 `Number` 对象的属性。

表 2.7 Number 基本属性

属性	属性说明
MAX_VALUE	数字的最大可能值可以是 $1.7976931348623157E + 308$
MIN_VALUE	数字的最小可能值可以是 $5E-324$
NaN	等于一个不是数字的值
NEGATIVE_INFINITY	小于 MIN_VALUE 的值
POSITIVE_INFINITY	大于 MAX_VALUE 的值
EPSILON	<code>Number.EPSILON</code> 实际上是能够表示的最小精度。误差如果小于这个值，就可以认为不存在误差，值为 $2.220446049250313e-16$
prototype	<code>Number</code> 对象的静态属性。使用 <code>prototype</code> 属性将新属性和方法分配给当前文档中的 <code>Number</code> 对象

(续表)

属性	属性说明
MAX_SAFE_INTEGER	最大可精确计算的整数, 2^{53}
MIN_SAFE_INTEGER	最小可精确计算的整数, -2^{53}

下面用代码来输出 Number 对象的属性。代码 2-73 给出了 Number 属性中的基本用法。

【代码 2-73】 Number 属性示例：Number2.ts

```
01 console.log("number 最大值:" + Number.MAX_VALUE);
02 console.log("number 最小值:" + Number.MIN_VALUE);
03 console.log("负无穷: " + Number.NEGATIVE_INFINITY);
04 console.log("正无穷: " + Number.POSITIVE_INFINITY);
05 console.log(Number.prototype);
```

2.7.2 NaN

NaN 是代表非数字值的特殊值, 用于指示某个值不是数字。可以把 Number 对象设置为该值, 来指示其不是数字值。可以使用 isNaN() 全局函数来判断一个值是否是 NaN 值。

Number.NaN 是一个特殊值, 说明某些算术运算 (如求负数的平方根) 的结果不是数字。方法 parseInt() 和 parseFloat() 在不能解析指定的字符串时就返回这个值。

TypeScript 以 NaN 的形式表示 Number.NaN。注意, NaN 与其他数值进行比较的结果总是不相等的, 包括它自身在内。因此, 不能与 Number.NaN 比较来检测一个值是不是数字, 而只能调用 isNaN() 来比较, 如代码 2-74 所示。

【代码 2-74】 NaN 示例：nan.ts

```
01 let a = Number.NaN;
02 let b = Number.NaN;
03 console.log(a == b);           //false
04 console.log(Number.isNaN(a));  //true
```

2.7.3 prototype

上面提到 prototype 属性使我们有能力向对象上动态添加属性和方法。在 TypeScript 中, 函数可以直接用 prototype 对函数属性和方法进行扩展, 如代码 2-75 所示。

【代码 2-75】 prototype 函数扩展示例：prototype_func.ts

```
01 function people(id:string, name:string) {
02     this.id = id;
03     this.name = name;
04 }
```

```

05  var emp = new people("123", "Smith");
06  var jack = new people("234", "JACK");
07  people.prototype.email = "smith@163.com";
08  people.prototype.walk = function () {
09      console.log(this.name + " walk");
10  }
11  jack.email = "jack@163.com";
12  console.log(emp.id);           // 123
13  console.log(emp.name);         //Smith
14  console.log(emp.email);         //smith@163.com
15  console.log(emp.walk());       //Smith walk
16  console.log(jack.id);          //234
17  console.log(jack.email);        // jack@163.com
18  console.log(jack.walk());       //jack walk

```

Number 对象无法通过 prototype 直接添加属性和方法，如下所示。

```
Number.prototype.prop2 = "3" ; //错误
```

那么该怎样去扩展 Number 对象呢？这里可以借助在 Number 接口 interface 上来进行扩展（接口将在后续章节详细说明），如代码 2-76 所示。

【代码 2-76】 prototype 对 Number 对象扩展示例：prototype_number.ts

```

01  interface Number {
02      padLeft(chars: string, length: number): string;
03  }
04  Number.prototype.padLeft = function (chars: string, length: number):
    string {
05      return (chars.repeat(length) + this);           //this 代码值
06  };
07  let a = 9;
08  console.log(a.padLeft("0", 3));                      //0009

```

代码 2-76 中 01~03 行通过在 interface Number 接口上定义了一个 padLeft 的方法，可以在 Number.prototype.padLeft 上定义具体实现逻辑。



TypeScript 中的 Number.prototype 不能直接添加新的属性和方法，和 JavaScript 不一样。

2.7.4 Number 的方法

Number 对象不但包含一些属性，同时也包含一些方法。这些方法可以更好地为我们提供服务。表 2.8 列出了一组 Number 对象的基本方法。

表 2.8 Number 基本方法

方法	方法说明
toExponential()	强制数字以指数表示法显示
toFixed()	格式化小数点右侧具有特定位数的数字
toLocaleString()	以浏览器的本地设置而变化的格式返回当前数字的 string 值
toPrecision()	定义显示数字的总位数（包括小数点左侧和右侧的数字）。负的精度将引发错误
toString()	返回数字的值的 string 表示形式。该函数可以传入一个基数参数，一个介于 2 和 36 之间的整数，指定用于表示数值的进制的基数
valueOf()	返回数字的原始值
isFinite()	用来检查一个数值是否为有限的（finite），即不是 Infinity
isNaN()	是否为非数值型
isInteger()	用来判断一个数值是否为整数
parseInt()	将参数解析成整数，解析成功返回数值，否则返回 NaN
parseFloat()	将参数解析成浮点数，解析成功返回数值，否则返回 NaN
isSafeInteger()	表示某值是否在整数范围-2 ⁵³ ~2 ⁵³ 之间（不含两个端点），超过这个范围就返回 false

下面针对 Number 的方法用代码来具体查看一下各方法的主要作用，这样更容易直观掌握各方法的实际含义，如代码 2-77 所示。

【代码 2-77】 Number 方法示例：method_number.ts

```

01  let a = 12345.28;
02  console.log(a.toExponential(2));           //1.23e+4
03  console.log(a.toFixed(2));                 //12345.28
04  console.log(a.toLocaleString());           //12,345.28
05  console.log(a.toString());                 //12345.28
06  console.log(a.toString(2));
07  console.log(a.toString(8));                 // 30071.2172702436561
08  console.log(a.toPrecision(1));             //1e+4
09  console.log(a.valueOf());                   //12345.28
10  console.log(Number.isFinite(a));           //true
11  console.log(Number.isNaN(a));              //false
12  console.log(Number.isInteger(a));          //false
13  console.log(Number.isSafeInteger(a));      //false
14  console.log(Number.parseFloat("2.18"));    //2.18
15  console.log(Number.parseInt("2.18"));      //2
16  console.log(Number.parseFloat("2.18"));    //2.18

```

2.8 字符串

字符串（string）在任何一门编程语言中都至关重要。字符串在存储上类似字符数组，所以它每一位的单个元素都是可以提取的。字符串是由数字、字母、下画线组成的一串字符。它是 TypeScript 语言中表示文本的数据类型。

在 TypeScript 语言中，可以用半角的双引号或单引号来表示字符串值，并且二者可以相互嵌套使用。在动态语言中，用字符串来表示代码，并且可以动态执行，这个功能异常强大，但是也比较危险，可能会执行恶意代码，如代码 2-78 所示。

【代码 2-78】 string eval 方法示例：string_eval.ts

```
01 let b = 12345.28;
02 eval("let a = 2 ; console.log(a+b);");
```

2.8.1 构造函数

在 TypeScript 中，可以用 new String()构造函数来定义一个 String 对象。它的值是 string 类型的。代码 2-79 演示了 String 构造函数的基本用法。11 行中的语句是错误的，string 是原始类型，而 String 是它的封装对象，二者不一样。

【代码 2-79】 String 构造函数示例：string_construct.ts

```
01 let a = new String("dddd");
02 console.log(a);
03 let b = new String(222);
04 console.log(b);
05 let c = new String(true);
06 console.log(c);
07 console.log(c[0]);           //t
08 let d = new String({});
09 console.log();
10 // 'string' is a primitive, but 'String' is a wrapper object
11 let str:string = new String("dddd");    //错误
```

2.8.2 prototype

String.prototype 中内置了很多方法，这样就可以在任何一个 string 类型的变量上去直接调用 String.prototype 中的方法。在 JavaScript 中，prototype 是实现面向对象的一个重要机制，可以动态地扩展属性和方法。

在 TypeScript 中，如果用代码 2-80 来扩展方法，编译器就会报错。

【代码 2-80】 String.prototype 扩展错误示例：string_prototype.ts

```

01  // 错误
02  String.prototype.padZero = function (this: string, length: number) {
03      var s = this;
04      while (s.length < length) {
05          s = '0' + s;
06      }
07      return s;
08  };

```



TypeScript 中的 String.prototype 不能直接添加新的属性和方法，这和 JavaScript 不一样。

如果需要动态地扩展 String.prototype 中的方法或属性，我们必须通过定义 interface String 来实现扩展，如代码 2-81 所示。

【代码 2-81】 String.prototype 扩展示例：string_prototype2.ts

```

01  interface String {
02      leadingChars(chars: string|number, length: number): string;
03  }
04  String.prototype.leadingChars = function (chars: string|number, length:
05      number): string {
06      return (chars.toString().repeat(length) + this).substr(-length);
07  };
08  let a = "@";
09  console.log(a.leadingChars("0", 8)); //0000000@

```

在代码 2-81 中，首先用接口 interface String 来声明一个方法 leadingChars，这样就可以用 String.prototype.leadingChars 来具体实现其方法逻辑了。

2.8.3 字符串的方法

字符串的一个常用属性是 length，可以输出字符串的长度。在字符串中更多使用的是方法。这些方法可以更好地让我们操作字符串。表 2.9 列出了一组 String 对象的方法。

表 2.9 String 基本方法

方法	说明
charAt()	返回在指定位置的字符
charCodeAt()	返回在指定位置的字符的 Unicode 编码
concat()	连接两个或更多字符串，并返回新的字符串
indexOf()	某个指定的字符串值在字符串中首次出现的位置
lastIndexOf()	从后向前搜索字符串，并从起始位置（0）开始计算返回字符串最后出现的位置

(续表)

方法	说明
localeCompare()	用本地特定的顺序来比较两个字符串
match()	查找到一个或多个正则表达式的匹配
replace()	替换与正则表达式匹配的子串
search()	检索与正则表达式相匹配的值
slice()	把字符串分割为子字符串数组
substr()	从起始索引号提取字符串中指定数目的字符
substring	提取字符串中两个指定的索引号之间的字符
toLowerCase()	把字符串转换为小写
toString()	返回字符串
toUpperCase()	把字符串转换为大写
valueOf()	返回指定字符串对象的原始值
includes	返回布尔值，表示是否找到了参数字符串
startsWith	返回布尔值，表示参数字符串是否在原字符串的头部
endsWith	返回布尔值，表示参数字符串是否在原字符串的尾部
repeat	返回一个新字符串，表示将原字符串重复 n 次
padStart	如果某个字符串不够指定长度，会在头部补全
padEnd	如果某个字符串不够指定长度，会在尾部补全
matchAll	返回一个正则表达式在当前字符串的所有匹配

2.9 小结

本章主要对 TypeScript 的基本语法进行了详细介绍，其中涉及很多核心概念，如变量、标识符、参数、函数以及数据类型。数据类型分为值类型和引用类型，它们是存在不少差异的，必须理解其本质。

本章是非常重要的知识点，是理论基础。要想学好 TypeScript 语言，必须先掌握好该语言的基本语法，才能由浅入深地继续学习后面的章节。

这一章涉及不少关于函数、类、接口和数值等的知识点，将在后续章节进行详细说明。

第 3 章

◀ 流程控制 ▶

上一章对 TypeScript 的基本语法进行了阐述，让读者对 TypeScript 的数据类型、变量的声明方式、let 和 var 在作用域上的区别等知识点有了初步的掌握。其中部分示例涉及流程控制。本章将对 TypeScript 的流程控制进行详细介绍。

通过本章的学习，可以让读者掌握如何通过流程控制语句来改变程序运行的顺序。流程控制是任何一门语言必须掌握的知识点。在声明式的编程语言中，流程控制指令是指会改变程序运行顺序的指令，可能是运行不同位置的指令，或是在多段程序中选择一個运行。TypeScript 语言所提供的流程控制指令主要有以下几种：

- 无条件分支指令，继续运行在其他位置的一段指令，例如 TypeScript 语言的 goto 指令。
- 条件分支指令，若特定条件成立时，运行一段指令，例如 TypeScript 语言的 if 指令。
- 循环指令，运行一段指令若干次，直到特定条件成立为止，例如 TypeScript 语言的 for 指令。
- 执行子程序指令，运行位于不同位置的一段指令，但完成后会继续运行原来要运行的指令。
- 无条件的终止指令，停止程序，不运行任何指令。

本章主要涉及的知识点有：

- 条件判断：学会 if...else 和 switch 等条件判断控制的语法以及用途。
- 循环流程：学会 for、while 等基本循环控制的语法以及用途。
- break 和 continue 的区别：掌握在循环体中 break 和 continue 的区别。



本章内容用到了函数的部分语法，关于函数的具体用法将在第 5 章介绍。

3.1 条件判断

我们知道，现实生活中很多事物在不同的条件下会有不同的结果，比如人生，往往选择很重要，在人生的分叉路口，一个人如何选择往往决定其后续的发展轨迹。一个程序的运行也是如此，在不同的条件下需要执行不同的操作，这也是程序的魅力所在，只要设计合理，就可以根据实际情况执行合理的逻辑。在某种程度上来说，程序的“智能”特征依赖于条件判断。

条件语句是一种根据条件执行不同代码的语句，如果条件满足就执行一段代码，否则执行其他代码。条件语句表达的意图可以理解为对某些事的决策规则或者表达某种关系，即“如果什么，则什么”。

例如，公司的员工需要请假，那么每个公司都会制定一套请假的流程来确定请假的规则。例如，首先员工发起请假申请，然后员工的主管进行审批，审批后会根据条件判断来确定下一步的审批人是谁，如果请假天数大于 3 天，由于请假天数比较多，请假单会在主管审批后流转至总经理处进行审批，总经理处审批后再流转至 HR 处审批，以便做好考勤工作；如果请假天数小于等于 3，那么请假单会在主管审批后流转至 HR 处进行审批，HR 审批通过后，流程结束。具体的流程示意如图 3.1 所示。

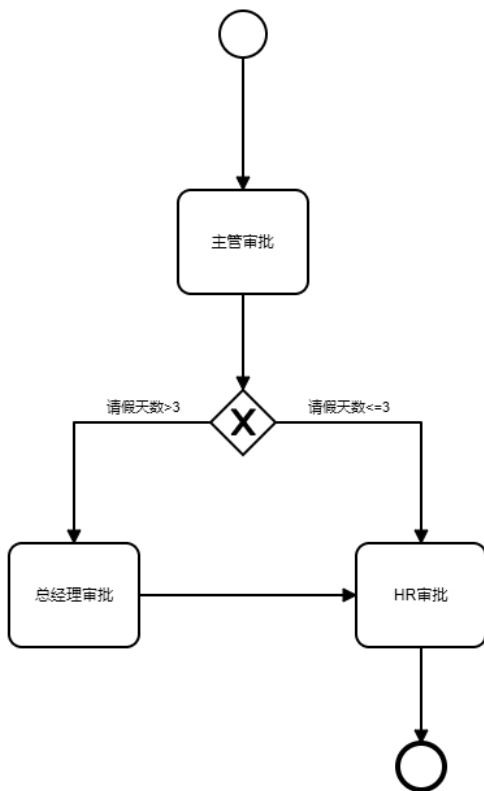


图 3.1 请假流程图

如果用 TypeScript 语言来构建一个请假流程的程序，就必须用到条件判断的相关知识。正

是有了条件判断，从而使得程序可以根据规则来灵活处理现实问题。在 TypeScript 语言中，实现条件判断的方法有多种，例如 if、if...else 和 switch。下面分别对这几种做详细说明。

3.1.1 if、if...else

程序实现条件判断最简单的就是 if，在 TypeScript 语言中，if 语法和 JavaScript 基本一致。为了直观了解 if 语法，假设有一个函数 getGrade，其中有一个数值类型的参数 money，如果参数 money 大于 8000，那么等级为 A；如果参数 money 大于 3500 且小于等于 8000，那么等级为 B；如果参数 money 大于 900 且小于等于 3500，那么等级为 C；其他情况等级为 D。具体内容如代码 3-1 所示。

【代码 3-1】 函数 getGrade 用 if 实现代码：ts001.ts

```
01  /**
02   * 根据 money 参数给出等级 A,B,C,D
03   * @param money
04   * @returns 字符串 A,B,C,D
05   */
06  function getGrade(money: number) {
07      if (money > 8000) {
08          return "A";
09      }
10      if (money <= 8000 && money>3500) {
11          return "B";
12      }
13      if (money <= 3500 && money>900) {
14          return "C";
15      }
16      return "D";
17  }
18  let a = getGrade(8002);
19  console.log(a);          //A
20  a = getGrade(5000);
21  console.log(a);          //B
22  a = getGrade(3200);
23  console.log(a);          //C
```



一般来说，if 语句所判定的条件必须是全集，否则可能会出现异常。

另一个常用的条件判断为 if ... else。还以上面的逻辑为例，这次将 if 换成 if ...else 进行编码，具体内容如代码 3-2 所示。

【代码 3-2】 函数 getGrade 用 if ... else 实现代码：ts002.ts

```

01  /**
02   * 根据 money 参数给出等级 A,B,C,D
03   * @param money
04   * @returns 字符串 A,B,C,D
05   */
06  function getGrade(money: number) {
07      if (money > 8000) {
08          return "A";
09      }
10      else if (money <= 8000 && money>3500) {
11          return "B";
12      }
13      else if (money <= 3500 && money>900) {
14          return "C";
15      }
16      else
17          return "D";    //只有一条语句，那么可以省略{}
18  }
19  let a = getGrade(8002);
20  console.log(a);        //A
21  a = getGrade(5000);
22  console.log(a);        //B
23  a = getGrade(3200);
24  console.log(a);        //C

```



if 或者 else 下如果只有一条语句，那么可以省略{}，但是建议保留，提高代码的可读性。

另外一个需要注意的是，在条件判定语句中，不要出现无法到达的语句，如下面的代码 3-3 中 08 行代码永远无法执行。默认情况下此代码不会报错，需要指定 TypeScript 中的 --allowUnreachableCode 编译选项为 false，用于检测此类错误。

【代码 3-3】 函数 ferror 实现代码：ts003.ts

```

01  function ferror(a) {
02      if (a) {
03          return true;
04      }
05      else {
06          return false;
07      }
08      return 1;    //无法到达的代码
09  }

```

第 1 章已经简单介绍了如何安装开发工具 Visual Studio Code，并学会用命令 `tsc` 对 TypeScript 文件进行编译，但并未介绍 `tsc` 命令的编译选项如何开启。下面就在 Visual Studio Code 工具中对如何配置 `tsc` 的编译选项进行说明。

在 Visual Studio Code 中，一般来说，一个 TypeScript 项目就对应一个文件夹。首先用 Visual Studio Code 打开 `chapter3` 文件夹，新建一个 `tsconfig.json` 文件。如果一个目录里存在一个 `tsconfig.json` 文件，那么编译器就认为这个目录是 TypeScript 项目的根目录。`tsconfig.json` 文件中指定了用来编译这个项目的若干配置和编译选项。`tsconfig.json` 的具体配置如图 3.2 所示。

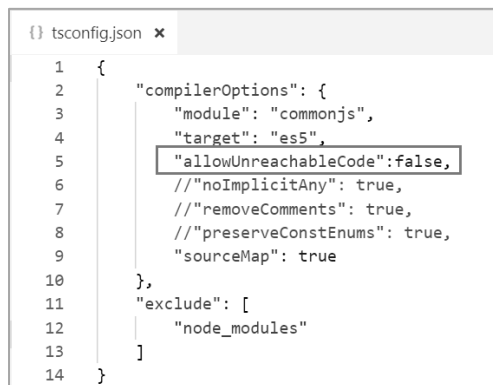


图 3.2 tsconfig.json 配置

不带任何参数的情况下调用 `tsc` 命令，编译器会从当前目录开始去查找 `tsconfig.json` 文件，如果找不到就逐级向上搜索父目录。当找到 `tsconfig.json` 时就会解析它，并按照相应配置去编译。

在 `tsconfig.json` 文件中，在 `"compilerOptions"` 节点下配置 `"allowUnreachableCode": false` 来开启 `--allowUnreachableCode` 编译选项。

然后在 Visual Studio Code 中新建一个文件并命名为 `ts003.ts`，其内容如代码 3-3 所示。这时 Visual Studio Code 编辑器会提示检测到不可达的代码（`unreachable code detected.`），具体提示如图 3.3 所示。

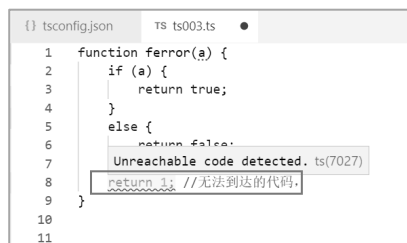
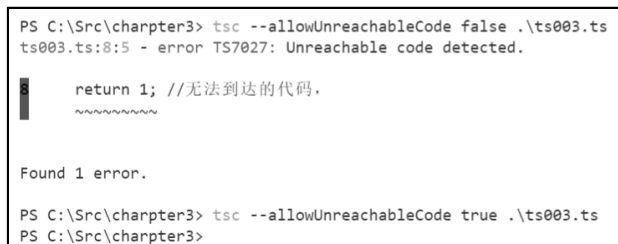


图 3.3 Visual Studio Code 检测到不可达代码提示界面

另外一种方法是，也可以用 `tsc` 命令来编译代码，查看代码是否有不可达的代码段。`tsc` 命令为：

```
tsc --allowUnreachableCode false .\ts003.ts
```

将此命令在 Visual Studio Code 中的 terminal 终端中执行时，编译器也会告知同样的错误，具体如图 3.4 所示。



```
PS C:\Src\chapter3> tsc --allowUnreachableCode false .\ts003.ts
ts003.ts:8:5 - error TS7027: Unreachable code detected.

8       return 1; //无法到达的代码。
      ~~~~~

Found 1 error.

PS C:\Src\chapter3> tsc --allowUnreachableCode true .\ts003.ts
PS C:\Src\chapter3>
```

图 3.4 Visual Studio Code 用 tsc 命令编译界面

从上面的两种方法对比来看，第一种方法通过配置 tsconfig.json 文件会更加方便，可以让编辑器自动检测错误。第二种方法只能在调用 tsc 编译的时候才能发现问题。

这个特性能捕获到的一个更不易发觉的错误是在 return 语句后添加换行语句而导致的检测到不可及的代码。因为 TypeScript 代码最后会编译成 JavaScript 执行，而 JavaScript 会自动在行末结束 return 语句。换句话说，如果在 return 行末没有其他语句，那么会自动添加分号（；），进而结束函数体。return 后面的代码相当于永不执行，变成了一个无法可达的代码区域，如代码 3-4 所示。

【代码 3-4】 函数 ferror_return 实现代码：ts004.ts

```
01  function ferror_return () {
02      return // 换行导致自动插入分号;
03      {
04          x: "string" // 检测到不可及的代码
05      }
06  }
```

3.1.2 嵌套 if

当某些条件判断需要判定的参数有多个，或者逻辑比较复杂时，可能单层 if 不能很好地解决问题，这时需要对 if 语句进行嵌套来解决。代码 3-5 给出了嵌套 if 的示例。

【代码 3-5】 函数 getGrade 嵌套 if 实现代码：ts005.ts

```
01  /**
02   * Sex 枚举
03   */
04  enum Sex {
05      Man = 1,
06      Female = 2,
07  }
08  /**
09   * 根据 money 和 sex 参数给出等级 A,B,C,D
```



```

10  * @param money
11  * @param sex
12  * @returns 字符串 A,B,C,D
13  */
14  function getGrade(money:number,sex:Sex){
15      if(sex === Sex.Man){
16          if (money > 8000) {
17              return "A";
18          }
19          else if (money <= 8000 && money>3500) {
20              return "B";
21          }
22          else if (money <= 3500 && money>900) {
23              return "C";
24          }
25          else
26              return "D";
27      }
28      else{
29          if (money > 7000) {
30              return "A";
31          }
32          else if (money <= 7000 && money>3000) {
33              return "B";
34          }
35          else if (money <= 3000 && money>800) {
36              return "C";
37          }
38          else
39              return "D";
40      }
41  }
42  let a = getGrade(7100,Sex.Man);
43  console.log(a);          //B
44  a = getGrade(7100,Sex.Female);
45  console.log(a);          //A

```



if 嵌套不宜过多，一般来说超过2层代码的可读性就会大大降低。

3.1.3 switch

除了 if 和 if ... else 以外，switch 也可以实现条件判断，但要注意 switch 语句中的每一个

case 表达式类型必须与 switch 表达式类型相匹配，否则会报错。

另外，switch 中的 case 语句块如果有逻辑语句，不是空的，那么必须用 break 结尾，否则程序会贯穿 case 区域，导致结果错误。代码 3-6 给出了 switch 的基本用法。

【代码 3-6】 函数 swithDemo1 实现代码：ts006.ts

```
01  /**
02   * 判断奇偶数
03   * @param num
04   * @returns 字符串
05   */
06  function swithDemo1(num:number){
07      let ret = "";
08      switch(num % 2){
09          case 0 : {
10              ret = "偶数";
11              break;          //在非空情况下 break 不能少，否则程序会贯穿此 case 区域
12          }
13          case 1: {
14              ret = "奇数";
15              break;
16          }
17      }
18      return ret;
19  }
20  let a = swithDemo1(201);
21  console.log(a);          //奇数
22  a = swithDemo1(202);
23  console.log(a);          //偶数
```

在代码 3-6 中，如果 11 行处将 break 去掉或者注释掉，那么 swithDemo1（202）返回"奇数"，而不是"偶数"。缺少 break 程序不会跳出 switch 的 {} 块，而是继续执行 case 1:{ }，也就是将 ret 重新赋值为 "奇数"。



TypeScript 现支持对当 switch 语句 case 中出现贯穿时报错的检测。这个检测默认是关闭的，可以使用 --noFallthroughCasesInSwitch 启用。

参考前面的 tsconfig.json 编译项配置，用 "noFallthroughCasesInSwitch":true 开启配置。那在 Visual Studio Code 中打开 ts006.ts 文件时，如果将 11 行的 break 注释掉，则编译器会提示穿透 switch 语句中的 case 错误（Fallthrough case in swith.）。检测提示界面如图 3.5 所示。

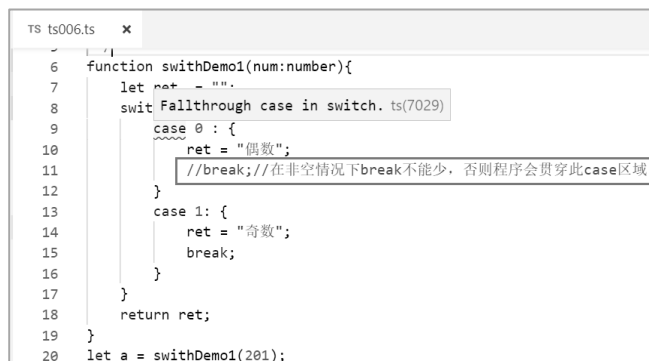


图 3.5 Visual Studio Code 穿透 switch 提示界面

另外, JavaScript 语言中没有返回值的代码分支会隐式地返回 `undefined`。这个特征往往会导导致程序不能按照预期执行。现在 TypeScript 编译器可以将这种方式标记为隐式返回, 虽然 TypeScript 编译器对于隐式返回的检查默认是被关闭的, 但是可以使用 `--noImplicitReturns` 来启用。代码 3-7 给出了检测隐式返回 `undefined` 的示例。

【代码 3-7】 函数 `freturn` 隐式返回代码: `ts007.ts`

```

01 function freturn(x) { // 错误: 不是所有分支都返回了值
02     if (x) {
03         return false;
04     }
05     // 隐式返回了 undefined
06 }

```

在 `tsconfig.json` 文件中用 `"noImplicitReturns":true` 开启编译选项, 那么在 Visual Studio Code 中打开 `ts007.ts` 文件时编译器会提示不是所有路径返回值的错误 (Not all code paths return a value.)。检测提示界面如图 3.6 所示。

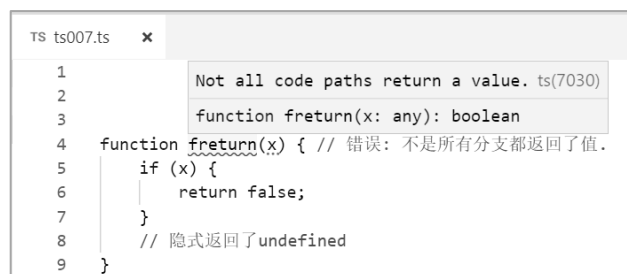


图 3.6 Visual Studio Code 提示不是所有路径返回值界面

3.2 循环

循环控制语句可以重复调用某段代码，直到满足某一条件退出或者永不退出，无限循环。一般在程序中除了条件判定以外，循环语句的使用率也是非常高的，因此我们必须掌握循环的基本用法。

在现实生活中，循环控制的例子也随处可见。例如，上学的时候，老师上课前用花名册点名，看有没有同学缺勤。学生花名册是一个包含学生学号和姓名等信息的列表。老师从第一个人开始，逐一向下进行点名，来上课的同学答“到”即可。

老师逐一循环对花名册中的每个同学进行点名的过程实际上就可以看作是一个循环的流程控制过程。老师要从花名册这个列表中查询出符合特定条件（缺勤的）的同学，必须借助循环和条件判断才能完成。

另外，我们的时钟计时也可以看作是一个循环，以分钟数和小时数为例，分钟数每隔 60 秒加 1，然后判断分钟数是否为 60，如果是，那么小时数加 1，同时置分钟数为 0，重新进行计时，等 60 秒后分钟数再加 1，如此循环；如果不是，那么再过 60 秒后分钟数加 1，如此循环，如图 3.7 所示。

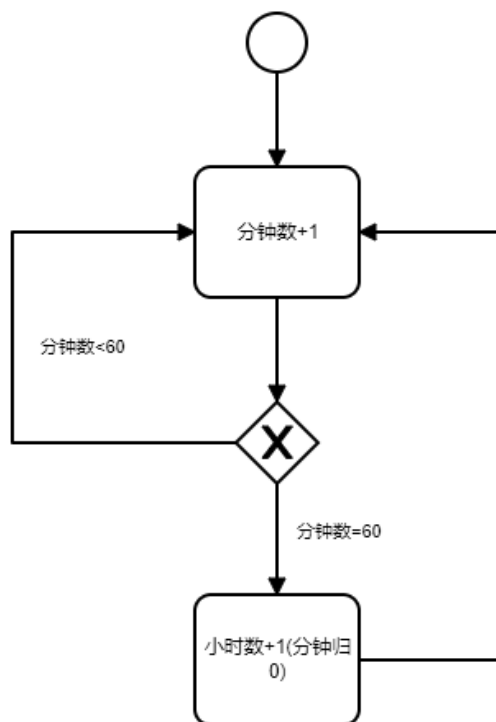


图 3.7 时钟计时循环流程图

在 TypeScript 中循环分为确定循环和不确定循环，常用的循环语句有 `for` 和 `while` 等。

3.2.1 for

在循环语句中，迭代次数是确定/固定的循环称为确定循环。for 循环一般来说是一个确定循环的实现。在 TypeScript 中，循环中的 for 语法和 JavaScript 一致，下面给出一个数组 testArray，通过循环将其各个元素打印出来。代码 3-8 给出了 for 的基本用法。

【代码 3-8】 函数 forDemo1 实现代码：ts008.ts

```
01  /**
02   * for 语法演示
03   */
04  function forDemo1(){
05      let testArray = [20, 30, 40, 50];
06      for (let i = 0; i < testArray.length; i ++) {
07          console.log(testArray[i]);
08      }
09  }
10  forDemo1();           // 20 30 40 50
```

在代码 3-8 中 05 行声明了一个名为 testArray 的数组，关于数组的具体语法，参见第 4 章。testArray 数组有 4 个元素，分别是 20、30、40 和 50。由于数组是一个可变长度的结构，要想打印出其中的每个元素，就必须利用循环才能完成。



for 循环中用 let 进行变量 i 声明，不要用 var，防止污染外部作用域中的变量。

如果 for 循环中循环的次数较多，上面代码中的 06 行可以进行优化，从而提高执行效率。由于 testArray.length 在每次循环的时候都要重新计算一下，因此可以用一个变量暂存这个 testArray.length 的值，这样循环的效率就会提高，改进的代码如代码 3-9 所示。

【代码 3-9】 函数 forDemo1 改进代码：ts009.ts

```
01  /**
02   * for 语法改进版演示
03   */
04  function forDemo1(){
05      let testArray = [20, 30, 40, 50];
06      let len = testArray.length;
07      for (let i = 0; i < len; i ++) {
08          console.log(testArray[i]);
09      }
10  }
11  forDemo1();           // 20 30 40 50
```

在 TypeScript 中，循环中也支持 for...in 语法。for...in 循环是对具体对象的键 key 和属性进行循环（不会忽略属性）。常用于打印对象或集合中键值对中的键名或是数组的下标及属性

值，代码 3-10 给出了 for...in 的用法。

【代码 3-10】 for...in 循环示例代码：ts010.ts

```

01  /**
02   * for...in 语法演示
03   */
04  interface Array<T>{
05      prop: string;
06  }
07  function forDemol(){
08      let testArray = [20, 30, 40, 50];
09      testArray.prop = "propValue";      //扩展属性
10      let len = testArray.length;
11      for (let item in testArray) {
12          console.log(item);      // 0 1 2 3 prop
13          console.log(testArray[item]); //20 30 40 50 propValue
14      }
15  }
16  forDemol();

```

在上面的代码 3-10 中，04~06 行用 interface Array<T>对数组泛型定义了一个 prop 字符型的属性，为了演示 for...in 可以打印出对象的属性。这一段涉及的知识点有数组、泛型和接口等，超出本章范围。这些知识点会陆续在后续章节进行介绍。

在 TypeScript 中还有一个 for...of 循环，是对具体对象中的值进行循环，而不是对键 key 的循环，并且忽略对象属性。代码 3-11 给出了 for...of 的用法。

【代码 3-11】 for...of 示例代码：ts011.ts

```

01  /**
02   * for...in 语法改进版演示
03   */
04  interface Array<T>{
05      prop: string;
06  }
07  function forDemol(){
08      let testArray = [20, 30, 40, 50];
09      testArray.prop = "propValue";      //扩展属性
10      let len = testArray.length;
11      for (let item of testArray) {
12          console.log(item);      //20 30 40 50
13      }
14      //console.log(item);      //var item 可以打印出 prop
15  }
16  forDemol();

```



也可以用 `for` 实现无限循环，即 `for(;;)`。

3.2.2 while

`for` 循环一般用于有限循环中，循环的迭代次数往往固定。在现实生活中，有些时候我们不知道循环何时退出，循环中的迭代次数不确定或未知时可以使用不确定循环 `while` 和 `do...while` 实现。

以现实中的例子来说，假如现在已经是晚上 11 点了，我们从网站上下载一个非常大的视频文件，由于下载的速度是时刻变化的，没必要一直等待它下载完成后人工关闭电脑。此时可以设置一个任务，在此文件下载完成后自动关闭电脑。

这个过程实际上可以用 `while` 循环来解决下载文件完成后自动关闭电脑的问题。模拟代码如下代码 3-12 所示。

【代码 3-12】 `while` 实现 `whileShutDownPC` 代码：ts012.ts

```
01  /**
02   * while 语法演示
03   */
04  function whileShutDownPC() {
05      let percent: number = 0;
06      while (percent < 100) {
07          console.log(percent + "%");    // 当前进度%
08          percent++;
09      }
10      shutdownPc();
11  }
12  function shutdownPc() {
13      console.log("执行关闭电脑操作");
14  }
15  whileShutDownPC();
```



`while` 循环体中的代码可能不执行。

3.2.3 do...while

`do...while` 循环类似于 `while` 循环，只是 `do...while` 循环不会在第一次循环执行时评估条件。将代码 3-12 中的 `while` 代码改成 `do...while` 的代码，如代码 3-13 所示。

【代码 3-13】 `do...while` 实现 `whileShutDownPC` 代码：ts013.ts

```
01  /**
```

```

02  * do...while 语法演示
03  */
04  function whileShutDownPC() {
05      let percent: number = 0;
06      do {
07          console.log(percent + "%");    //当前进度
08          percent++;
09      }
10      while (percent < 100);
11      shutdownPc();
12  }
13  function shutdownPc() {
14      console.log("执行关闭电脑操作");
15  }
16  whileShutDownPC();

```



do...while 循环体中的代码至少执行一次。

3.3 break 和 continue

在无限循环中，如果想跳出循环，就要借助 **break**。**break** 和 **continue** 经常和循环语句一起使用，用于更好地控制逻辑。

break 语句可用于跳出循环，并退出所在的循环体。**continue** 语句可以在出现了指定的条件下中断循环中的迭代，然后继续循环中的下一个迭代，循环并未退出。**break** 的基本用法如代码 3-14 所示。

【代码 3-14】 **break** 语句跳出循环示例代码：ts014.ts

```

01  /**
02   * break 语法演示
03   */
04  function beakDemol() {
05      let num:number = 100;
06      while (true) {
07          //console.log(num);
08          if(num == 108){
09              console.log("break");
10              break;
11          }
12          num++;

```



```

13     }
14     console.log(num);           //108
15 }
16 beakDemo1();                   //108

```

代码 3-14 定义了一个 `beakDemo1` 函数，函数体内用 `while(true)` 创建了一个无限循环，如果不用一个条件来中断循环，那么此函数将是一个死循环，如果直接在浏览器中运行，会导致浏览器卡死。

为了在符合某一个条件的情况下跳出循环体，这里用条件判断语句来实现。我们初始化一个数值型的变量 `num` 为 100，循环体内的末尾递增加 1，然后再次循环。当 `num` 的值为 108 的时候，用 `break` 语句退出循环 `while`，执行 14 行的 `console.log(num)` 语句，打印出结果。

当 `num` 的值小于 108 的时候，`while` 循环体中大的执行路径为①→②→①循环。当 `num` 的值等于 108 的时候，执行③处的 `break` 语句，直接退出 `while` 循环，`break` 后面循环体内的代码将不执行，即执行路径为③→④。这个基本过程可以用图 3.8 来说明。

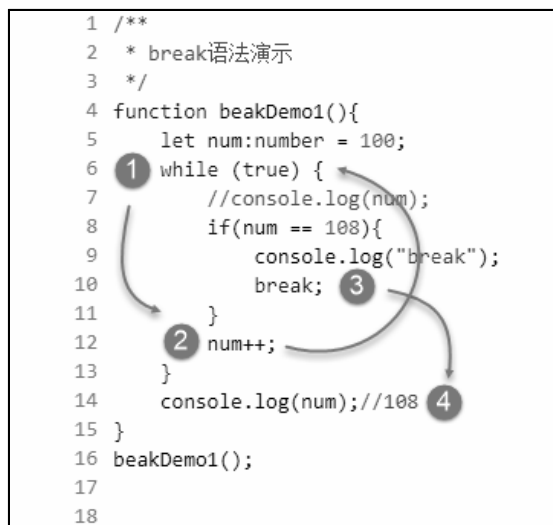


图 3.8 break 中断循环示意图

`continue` 语句跳过当前迭代中的后续语句，并将流程控制调整到循环的开头。与 `break` 语句不同，`continue` 不会退出循环。它终止当前迭代并开始后续迭代。`continue` 语句在循环体中的基本用法如代码 3-15 所示。

【代码 3-15】 `continue` 语句示例代码：ts015.ts

```

01  /**
02   * continue 语法演示
03   */
04  function continueDemo2(){
05      let num:number = 100;
06      while (num<200) {

```

```

07         num++;
08         if(num == 108){
09             console.log("continue");
10             continue;
11         }
12         console.log(num);          //100 ... 199, 缺少 108
13     }
14     console.log(num);              //200
15 }
16 continueDemo2();                  //200

```

提示

若将 07 行 `num++` 移动到 11 行之后，则此循环就是一个死循环，永不退出。

当 `num` 的值不等于 108 的时候，`while` 循环体中的执行路径为①→②→①循环。当 `num` 的值等于 108 的时候，执行③处的 `continue` 语句，但不退出 `while` 循环，只是此次不执行 `continue` 后面循环体内的代码而已，也就是不执行②处的 `console.log(num)`，即再次循环，由于 108 仍然小于 200，条件成立，继续执行④处的 `num++`，执行路径为③→④。

因此，`continueDemo2` 中循环体将依次打印出的 100 到 199 中唯独缺少 108，当 `num` 等于 108 的时候打印出了“continue”。这个基本过程可以用图 3.9 来说明。

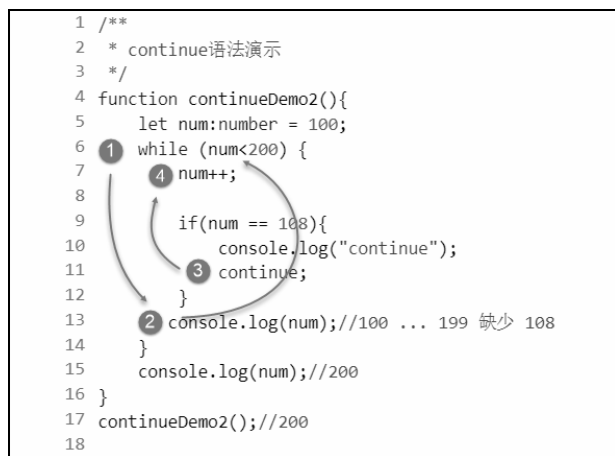


图 3.9 `continue` 中断循环示意图

另外，在 TypeScript 中 `continue` 还有一个跳转到标签的功能，类似于 `goto` 的作用，如代码 3-16 所示。

【代码 3-16】 `continue` 跳转到标签的示例代码：ts016.ts

```

01  /**
02  * continue 跳转到标签演示
03  */
04  forLabel1: for (let i = 0; i < 3; i++) {

```

```
05     forLabel2 : for (let j = 0; j < 3; j++) {  
06         if (i == 1 && j == 1) {  
07             continue forLabel1;//中断本次循环，跳转到 04 行的 forLabel1 继续执行  
08         } else {  
09             console.log("i = " + i + ", j = " + j);  
10         }  
11     }  
12 }
```



这种用法在某些特殊情况下可以解决问题，但是一般不建议使用，跳转到标签可以任意打乱执行顺序，严重降低了代码的可读性。

3.4 小结

本章主要对 TypeScript 语言中的流程控制语句进行了阐述，让读者掌握条件判断和循环的几种基本用法。其中，条件判断的实现可以用 if、if... else 以及 switch 进行实现。循环可以用 for 和 while 实现，其中在循环体中注意 break 和 continue 的区别。

另外，本章也对如何在 Visual Studio Code 中开启相关 tsc 编译选项进行了说明。