第一章 快速入门

0、TypeScript简介

1. TypeScript是JavaScript的超集。
2. 它对JS进行了扩展，向JS中引入了类型的概念，并添加了许多新的特性。
3. TS代码需要通过编译器编译为JS，然后再交由JS解析器执行。
4. TS完全兼容JS，换言之，任何的JS代码都可以直接当成JS使用。
5. 相较于JS而言，TS拥有了静态类型，更加严格的语法，更强大的功能；TS可以在代码执行前就完成代码的检查，减小了运行时异常的出现的几率；TS代码可以编译为任意版本的JS代码，可有效解决不同JS运行环境的兼容问题；同样的功能，TS的代码量要大于JS，但由于TS的代码结构更加清晰，变量类型更加明确，在后期代码的维护中TS却远远胜于JS。

1、TypeScript 开发环境搭建

1. 下载Node.js
    - 64位：https://nodejs.org/dist/v14.15.1/node-v14.15.1-x64.msi
    - 32位：https://nodejs.org/dist/v14.15.1/node-v14.15.1-x86.msi
2. 安装Node.js
3. 使用npm全局安装typescript
    - 进入命令行
    - 输入：npm i -g typescript
4. 创建一个ts文件
5. 使用tsc对ts文件进行编译
    - 进入命令行
    - 进入ts文件所在目录
    - 执行命令：tsc xxx.ts

2、基本类型

- 类型声明
    - 类型声明是TS非常重要的一个特点
    - 通过类型声明可以指定TS中变量（参数、形参）的类型
    - 指定类型后，当为变量赋值时，TS编译器会自动检查值是否符合类型声明，符合则赋值，否则报错
    - 简而言之，类型声明给变量设置了类型，使得变量只能存储某种类型的值
    - 语法：

```
let 变量: 类型;

let 变量: 类型 = 值;

function fn(参数: 类型, 参数: 类型): 类型{
      ...
}
```

- 自动类型判断
  - TS拥有自动的类型判断机制
  - 当对变量的声明和赋值是同时进行的，TS编译器会自动判断变量的类型
  - 所以如果你的变量的声明和赋值时同时进行的，可以省略掉类型声明
- 类型：

类型	例子	描述
number	1, -33, 2.5	任意数字
string	'hi', "hi", `hi`	任意字符串
boolean	true、false	布尔值true或false
字面量	其本身	限制变量的值就是该字面量的值
any	*	任意类型
unknown	*	类型安全的any
void	空值（undefined）	没有值（或undefined）
never	没有值	不能是任何值
object	{name:'孙悟空'}	任意的JS对象
array	[1,2,3]	任意JS数组
tuple	[4,5]	元素，TS新增类型，固定长度数组
enum	enum{A, B}	枚举，TS中新增类型

- number

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
```

- boolean

```
let isDone: boolean = false;
```

- string

```
let color: string = "blue";
color = 'red';

let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${fullName}.

I'll be ${age + 1} years old next month.`;
```

- 字面量
  - 也可以使用字面量去指定变量的类型，通过字面量可以确定变量的取值范围
  - ```
    let color: 'red' | 'blue' | 'black';
    let num: 1 | 2 | 3 | 4 | 5;
    ```
- any
  - ```
    let d: any = 4;
    d = 'hello';
    d = true;
    ```
- unknown
  - ```
    let notSure: unknown = 4;
    notSure = 'hello';
    ```
- void
  - ```
    let unusable: void = undefined;
    ```
- never
  - ```
    function error(message: string): never {
        throw new Error(message);
    }
    ```
- object（没啥用）
  - ```
    let obj: object = {};
    ```
- array
  - ```
    let list: number[] = [1, 2, 3];
    let list: Array<number> = [1, 2, 3];
    ```
- tuple
  - ```
    let x: [string, number];
    x = ["hello", 10];
    ```
- enum
  - ```
     enum Color {
        Red,
        Green,
        Blue,
    }
    let c: Color = Color.Green;

    ```
    ```

```
enum Color {
    Red = 1,
    Green,
    Blue,
}
let c: Color = Color.Green;

enum Color {
    Red = 1,
    Green = 2,
    Blue = 4,
}
let c: Color = Color.Green;
```

- 类型断言
  - 有些情况下，变量的类型对于我们来说是很明确，但是TS编译器却并不清楚，此时，可以通过类型断言来告诉编译器变量的类型，断言有两种形式：
    - 第一种
      - ```
        let someValue: unknown = "this is a string";
        let strLength: number = (someValue as string).length;
        ```
    - 第二种
      - ```
        let someValue: unknown = "this is a string";
        let strLength: number = (<string>someValue).length;
        ```

3、编译选项

- 自动编译文件
  - 编译文件时，使用 -w 指令后，TS编译器会自动监视文件的变化，并在文件发生变化时对文件进行重新编译。
  - 示例：
    - ```
      tsc xxx.ts -w
      ```
- 自动编译整个项目
  - 如果直接使用tsc指令，则可以自动将当前项目下的所有ts文件编译为js文件。
  - 但是能直接使用tsc命令的前提时，要先在项目根目录下创建一个ts的配置文件 tsconfig.json
  - tsconfig.json是一个JSON文件，添加配置文件后，只需只需 tsc 命令即可完成对整个项目的编译
  - 配置选项：
    - include
      - 定义希望被编译文件所在的目录
      - 默认值：["**/*"]
      - 示例：
        - ```
          "include":["src/**/*", "tests/**/*"]
          ```

        - 上述示例中，所有src目录和tests目录下的文件都会被编译
- exclude
    - 定义需要排除在外的目录
    - 默认值：["node_modules", "bower_components", "jspm_packages"]
    - 示例：
        - ```
          "exclude": ["./src/hello/**/*"]
          ```
        - 上述示例中，src下hello目录下的文件都不会被编译
- extends
    - 定义被继承的配置文件
    - 示例：
        - ```
          "extends": "./configs/base"
          ```
        - 上述示例中，当前配置文件中会自动包含config目录下base.json中的所有配置信息
- files
    - 指定被编译文件的列表，只有需要编译的文件少时才会用到
    - 示例：
        - ```
          "files": [
                "core.ts",
                "sys.ts",
                "types.ts",
                "scanner.ts",
                "parser.ts",
                "utilities.ts",
                "binder.ts",
                "checker.ts",
                "tsc.ts"
            ]
          ```
        - 列表中的文件都会被TS编译器所编译
    - compilerOptions
        - 编译选项是配置文件中非常重要也比较复杂的配置选项
        - 在compilerOptions中包含多个子选项，用来完成对编译的配置
            - 项目选项
                - target
                    - 设置ts代码编译的目标版本
                    - 可选值：
                        - ES3（默认）、ES5、ES6/ES2015、ES7/ES2016、ES2017、ES2018、ES2019、ES2020、ESNext
                    - 示例：
                        - ```
                          "compilerOptions": {
                                "target": "ES6"
                          }
                          ```

        - 如上设置，我们所编写的ts代码将会被编译为ES6版本的js代码
- lib
    - 指定代码运行时所包含的库（宿主环境）
    - 可选值：
        - ES5、ES6/ES2015、ES7/ES2016、ES2017、ES2018、ES2019、ES2020、ESNext、DOM、WebWorker、ScriptHost ……
    - 示例：
        - ```
          "compilerOptions": {
                "target": "ES6",
                "lib": ["ES6", "DOM"],
                "outDir": "dist",
                "outFile": "dist/aa.js"
          }
          ```
- module
    - 设置编译后代码使用的模块化系统
    - 可选值：
        - CommonJS、UMD、AMD、System、ES2020、ESNext、None
    - 示例：
        - ```
          "compilerOptions": {
                "module": "CommonJS"
          }
          ```
- outDir
    - 编译后文件的所在目录
    - 默认情况下，编译后的js文件会和ts文件位于相同的目录，设置outDir后可以改变编译后文件的位置
    - 示例：
        - ```
          "compilerOptions": {
                "outDir": "dist"
          }
          ```
        - 设置后编译后的js文件将会生成到dist目录
- outFile
    - 将所有的文件编译为一个js文件
    - 默认会将所有的编写在全局作用域中的代码合并为一个js文件，如果module制定了None、System或AMD则会将模块一起合并到文件之中
    - 示例：
        - ```
          "compilerOptions": {
                "outFile": "dist/app.js"
          }
          ```
- rootDir

        - 指定代码的根目录，默认情况下编译后文件的目录结构会以最长的公共目录为根目录，通过rootDir可以手动指定根目录
        - 示例：
            - ```
              "compilerOptions": {
                    "rootDir": "./src"
              }
              ```
    - allowJs
        - 是否对js文件编译
    - checkJs
        - 是否对js文件进行检查
        - 示例：
            - ```
              "compilerOptions": {
                    "allowJs": true,
                    "checkJs": true
              }
              ```
    - removeComments
        - 是否删除注释
        - 默认值：false
    - noEmit
        - 不对代码进行编译
        - 默认值：false
    - sourceMap
        - 是否生成sourceMap
        - 默认值：false

- 严格检查
    - strict
        - 启用所有的严格检查，默认值为true，设置后相当于开启了所有的严格检查
    - alwaysStrict
        - 总是以严格模式对代码进行编译
    - noImplicitAny
        - 禁止隐式的any类型
    - noImplicitThis
        - 禁止类型不明确的this
    - strictBindCallApply
        - 严格检查bind、call和apply的参数列表
    - strictFunctionTypes
        - 严格检查函数的类型
    - strictNullChecks
        - 严格的空值检查
    - strictPropertyInitialization
        - 严格检查属性是否初始化
- 额外检查

        - noFallthroughCasesInSwitch
            - 检查switch语句包含正确的break
        - noImplicitReturns
            - 检查函数没有隐式的返回值
        - noUnusedLocals
            - 检查未使用的局部变量
        - noUnusedParameters
            - 检查未使用的参数
    - 高级
        - allowUnreachableCode
            - 检查不可达代码
            - 可选值：
                - true，忽略不可达代码
                - false，不可达代码将引起错误
        - noEmitOnError
            - 有错误的情况下不进行编译
            - 默认值：false

4、webpack

- 通常情况下，实际开发中我们都需要使用构建工具对代码进行打包，TS同样也可以结合构建工具一起使用，下边以webpack为例介绍一下如何结合构建工具使用TS。
- 步骤：
    1. 初始化项目
        - 进入项目根目录，执行命令 `npm init -y`
            - 主要作用：创建package.json文件
    2. 下载构建工具
        - `npm i -D webpack webpack-cli webpack-dev-server typescript ts-loader clean-webpack-plugin`
            - 共安装了7个包
                - webpack
                    - 构建工具webpack
                - webpack-cli
                    - webpack的命令行工具
                - webpack-dev-server
                    - webpack的开发服务器
                - typescript
                    - ts编译器
                - ts-loader
                    - ts加载器，用于在webpack中编译ts文件
                - html-webpack-plugin
                    - webpack中html插件，用来自动创建html文件
                - clean-webpack-plugin

- webpack中的清除插件，每次构建都会先清除目录

3. 根目录下创建webpack的配置文件webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");

module.exports = {
      optimization:{
            minimize: false // 关闭代码压缩，可选
      },

      entry: "./src/index.ts",

      devtool: "inline-source-map",

      devServer: {
            contentBase: './dist'
      },

      output: {
            path: path.resolve(__dirname, "dist"),
            filename: "bundle.js",
            environment: {
                  arrowFunction: false // 关闭webpack的箭头函数，可选
            }
      },

      resolve: {
            extensions: [".ts", ".js"]
      },

      module: {
            rules: [
                  {
                        test: /\.ts$/,
                        use: {
                            loader: "ts-loader"
                        },
                        exclude: /node_modules/
                  }
            ]
      },

      plugins: [
            new CleanWebpackPlugin(),
            new HtmlWebpackPlugin({
                  title:'TS测试'
            }),
      ]

}
```

4. 根目录下创建tsconfig.json，配置可以根据自己需要

```
{
    "compilerOptions": {
        "target": "ES2015",
        "module": "ES2015",
        "strict": true
    }
}
```

5. 修改package.json添加如下配置

```
{
  ...略...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack",
    "start": "webpack serve --open chrome.exe"
  },
  ...略...
}
```

6. 在src下创建ts文件，并在并命令行执行 `npm run build` 对代码进行编译，或者执行 `npm start` 来启动开发服务器

5、Babel

- 经过一系列的配置，使得TS和webpack已经结合到了一起，除了webpack，开发中还经常需要结合babel来对代码进行转换以使其可以兼容到更多的浏览器，在上述步骤的基础上，通过以下步骤再将babel引入到项目中。
  1. 安装依赖包：
     - `npm i -D @babel/core @babel/preset-env babel-loader core-js`
     - 共安装了4个包，分别是：
       - @babel/core
         - babel的核心工具
       - @babel/preset-env
         - babel的预定义环境
       - @babel-loader
         - babel在webpack中的加载器
       - core-js
         - core-js用来使老版本的浏览器支持新版ES语法
  2. 修改webpack.config.js配置文件

```
...略...
module: {
    rules: [
        {
            test: /\.ts$/,
            use: [
                {
                    loader: "babel-loader",
                    options:{
```

```
                                presets: [
                                    [
                                        "@babel/preset-env",
                                        {
                                            "targets":{
                                                "chrome": "58",
                                                "ie": "11"
                                            },
                                            "corejs":"3",
                                            "useBuiltIns": "usage"
                                        }
                                    ]
                                ]
                            }
                        },
                        {
                            loader: "ts-loader",

                        }
                    ],
                    exclude: /node_modules/
                }
            ]
    }
    ...略...
```

- 如此一来，使用ts编译后的文件将会再次被babel处理，使得代码可以在大部分浏览器中直接使用，可以在配置选项的targets中指定要兼容的浏览器版本。