第一章：TypeScript

1. TypeScript 简介

1.1. TypeScript 是什么?

1. TypeScript 是以 JavaScript 为基础构建的语言。
2. TypeScript 是 JavaScript 的超集。
3. TypeScript 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。
4. TypeScript 可以在任何支持 JavaScript 的平台中执行（只要支持 JavaScript 的平台都可以用 TS）。
5. TS 代码不能被 JS 解析器直接执行，需要通过编译器编译为 JS，然后再交由 JS 解析器执行。
6. TS 完全兼容 JS，换言之，任何的 JS 代码都可以直接当 TS 代码使用。

1.2. TypeScript 增加了什么?

- 类型
- 支持 ES 的新特性
- 添加了 ES 不具备的新特性（抽象类、接口、装饰器等）
- 丰富的配置选项（根据需求对进行配置的修改）
- 强大的开发工具

相比较而言，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，减小了运行时异常出现的机率；
- TS 代码可以编译为任意版本的 JS 代码，可以有效解决不同 JS 运行环境的兼容问题；
- 同样的功能，TS 代码量要大于 JS，但由于 TS 代码结构更加清晰，变量类型更加明确，在后期维护中 TS 却远远胜于 JS。

2. 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`

3. 基本类型

3.1. 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{
  ...
}
```

3.2. 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是同时进行的，可以省略类型声明（不过建议都写上，方便后期维护）

3.3. 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	值本身	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, null)	没有值(或undefined, null)
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]	任意的JS数组
tuple	[4, 5]	元组，TS新增类型，固定长度的数组
enum	enum{A, B}	枚举，TS中新增类型

3.4. 类型实例

- number

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

// a 的类型设置为了number，在以后使用中a的值只能是数字
let a: number
a = 10
// a = 'hello' // 此行代码会报错，因为变量a的类型是number，不能赋值字符串
```

- boolean

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

- string

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

let fullName: string = `Bob Bobbington`
let age: number = 19
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类型，通过字面量可以确定的变量的取值范围

```ts
// 可以使用 | 来连接多个类型（类和类型）
let color: 'red' | 'blue' | 'black'
let num: 1 | 2 | 3 | 4 | 5

let b: 'male' | 'female'
b = 'male'
b = 'famale'

let c: boolean | string
c = true
c = 'hello'
```

- any
  - any 表示的是任意类型，一个变量设置类型为 any 后，相当于对该变量关闭了 TS 的类型检测
  - 使用 TS 时，不建议使用 any 类型
  - any 分为显式的和隐式的，变量声明时没有指定类型会自动判断变量的类型为 any
    - 显式 any：`let d: any`
    - 隐式 any：`let d`
  - any 类型的变量可以赋值给任意变量（不会报错）=> 不仅霍霍自己，还霍霍别人

```ts
let s: string

let d: any // 显式 any
d = 10
d = 'hello'
d = true

let d // 隐式 any

// d 的类型是 any，它可以赋值给任意变量（不会报错）=> 不仅霍霍自己，还霍霍别人
s = d
```

- unknown
  - 表示未知的值
  - 实际上就是一个类型安全的 any
  - unknown 类型的变量不能直接赋值给其他变量 => 只霍霍自己，不会霍霍别人

```ts
let s: string

let e: unknown
e = 10
e = 'hello'
e = true

// unknown 类型的值赋值给别的类型的值，会报错 => 只霍霍自己，不会霍霍别人
// s = e
// 解决上述问题
if (typeof e === 'string') {
  s = e
```

```
}
s = e as string // 类型断言，也可以解决 unknown 类型不能赋值的问题
```

- void
  - 当函数没有返回值的时候，类型时void

```
// void 用来表示空，以函数为例，就表示没有返回值的函数
function fn(): void {
  // return null
  // return undefined
}

let unusable: void = undefined
let unuse: void = null
```

- never
  - never 表示永远不会返回结果

```
function error(message: string): never {
  throw new Error(message)
}
```

- object（没啥用）

```
let a: object
a = {}
a = function () {}
// 以上用法没啥用，不够具体

// 一般用以下用法
// {} 用来指定对象中可以包含哪些属性
// 语法：{属性名: 属性值类型， 属性名: 属性值类型}
// 在属性名后边加上 ? 表示属性是可选的
let b: {name: string, age?: number}  // age?: number 表示 age 属性有没有都可以
b = {name: 'jack', age: 18}; // 变量包含的属性必须跟上边定义的结构一模一样

// [propName: string]: any 表示任意类型的属性
let c: {name: string, [propName: string]: any}
c = {name: 'rose', age: 18, gender: '男'}

/*
  函数结构类型声明：
    语法：(形参: 类型， 形参: 类型 ...) => 返回值类型
*/
//: 表示有两个 number 类型的参数，返回值是 number
let d: (a: number, b: number) => number
d = function (n1, n2): number{
  return n1 + n2
}
```

- array

```
/*
  数组的类型声明：
    类型[]
    Array<类型>
*/
// string[] 表示字符串类型数组
let e: string[]
e = ['q', 'w', 'e', 'r']

// number[] 表示数字数组（只存数字）
let list: number[] = [1, 3, 4]
let list: Array<number> = [1, 3, 4]
// 以上两种方法是等价的
```

- tuple
  - 元组
  - 元组就是固定长度的数组

```
/*
  语法：[类型， 类型， 类型， ...]
*/
let x: [string, number]
x = ['hello', 10]
```

- enum
  - 变量值在多个值之间进行选择的时候，我们可以用枚举

```
/*
  enum 枚举
*/
// 定义枚举类型
enum Gender {
  Male,
  Female
}

let i: {name: string, gender: Gender}
i = {
  name: '孙悟空',
  gender: Gender.Male
}
console.log(i.gender === Gender.Male)

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

enum Color {
  Red = 1,
  Green = 2,
  Blue = 4,
}
```

```
let c: Color = Color.Green

```

- & 的使用

```
// & 表示同时
let j: { name: string } & { age: number } // 表示 j 既要满足前边一个对象，又要满足后边一个对象
j = { name: 'jack', age: 18}
```

- 类型的别名

```
// let k: 1 | 2 | 3 | 4 | 5
// let l: 1 | 2 | 3 | 4 | 5
// type myType = string // 表示string类型的别名是myType，string 与 myType 是等价的
type myType = 1 | 2 | 3 | 4 | 5
let k: myType
let l: myType
k = 2
k = 1
l = 3
l = 5
```

3.5. 类型断言

- 它可以用来告诉解析器变量的实际类型
- 语法：

```
变量 as 类型
或者
<类型>变量
```

- 有些情况下，变量的类型对于我们来说很明确，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

```
// s = e as string
let someValue: unknwon = 'this is a string'
let strLength: number = (someValue as string).length
```

  - 第二种

```
// s = <string>e
let someValue: unknown = 'this is a string'
let strLength: number = (<string>someValue).length
```

4. 编译选项

4.1. 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，并在文件发送变化时对文件进行重新编译。
- 示例：

```
tsc xxx.ts -w
```

4.2. 自动编译整个项目

- 如果使用 tsc 指令，则可以自动将当前目录下的所有 ts 文件编译为 js 文件。
- 但是能直接使用 tsc 命令的前提是，要现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 tsc 命令即可完成对整个项目的编译、tsc -w 即可监视整个项目下的 ts 文件
- tsconfig.json 是 ts 编译器的配置文件，ts 编译器可以根据它的信息来对代码进行编译
- 配置选项：
  - include
    - 定义希望被编译的文件所在的目录
    - 默认值：["**/*"]
    - 路径：** 表示任意目录，* 表示任意文件
    - 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文件目录
    - 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：

```
"extend": "./configs/base"
```

      - 上述示例中，当前配置文件会自动包含 configs 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
  - 编译器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
    1. 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"
}
```

- 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成一个 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解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
    - 默认值是 false
    - 示例：

```
{
  "compilerOptions": {
    "allowJs": true
  }
}
```

- checkJs
  - 是否对 js 文件进行检查
  - 默认值是 false
  - 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
  - 只对代码进行检查而不需要编译的时候用
  - 默认值：false
- sourceMap
  - 是否生成 sourceMap
  - 默认值：false

2. 严格检查

- strict
  - 所有严格检查的总开关
  - 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
- stictFunctionTypes
  - 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

3. 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

4. 高级
    - allowUnreachableCode
        - 检查不可迭代码
        - 可选值：
            - true，忽略不可迭代码
            - false，不可迭代码引起错误
    - noEmitOnError
        - 有错误额情况下不进行编译
        - 默认值：false

---

5. typescript 结合 webpack 使用

- 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
          ```
        - 一共安装了 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 wenjian
            - 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，每次构建都会线清除目录

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

```
const { resolve } = require('path')
const HmtlWebpackPlugin = require('html-webpack-plugin')
const { CleanWebpackPlugin } = require('clean-webpack-plugin')

module.exports = {
  // 指定入口文件
  entry: './src/index.ts',
  // 指定打包文件输出到哪里
  output: {
    filename: 'bundle.js',
    path: resolve(__dirname, 'dist'),
    environment: {
      arrowFunction: false //关闭 webpack 的箭头函数，可选
      const: false // 禁止使用 const
    }
  },
  // 指定 webpack 打包时要使用的模块
  module: {
    // 指定要加载的规则
    rules: [
      {
        // 指定的是规则生效的文件
        test: /\.ts$/,
        exclude: /node_modules/,
        loader: 'ts-loader'
      }
    ]
  },
  plugins: [
    new CleanWebpackPlugin(), // 现在已经不用了，直接在output中添加：clean: tr
    new HtmlWebpackPlugin({
      title: 'TS测试',
    }),
  ],
  // 用来设置引用模块，表示 ts 文件和 js 文件都可以作为模块化使用
  resolve: {
    extensions: ['.ts', '.js']
  },
  optimizetion: {
    minimize: false // 关闭代码压缩，可选
  },
  devServer: {
    contentBase: './dist',
    port: 5000,
  },
  devtool: 'inline-source-map',
}
```

4. 根目录下创建 tsconfig.json，配置可以根据自己需要配置

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

5. 修改 package.json 添加如下配置

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

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

6. 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
         - 是 JavaScript 的运行环境（或者说模拟 JavaScript 运行环境的代码）
         - 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

  ```
  const { resolve } = require('path')
  const HmtlWebpackPlugin = require('html-webpack-plugin')
  const { CleanWebpackPlugin } = require('clean-webpack-plugin')

  module.exports = {
  ```

```
  // 指定入口文件
  entry: './src/index.ts',
  // 指定打包文件输出到哪里
  output: {
    filename: 'bundle.js',
    path: resolve(__dirname, 'dist'),
    // 打包之前先清除上次编译的文件
    clean: true,
    // 配置打包环境
    environment: {
      // 告诉 webpack 不使用箭头函数
      arrowFunction: false //关闭 webpack 的箭头函数，可选
    }
  },
  // 指定 webpack 打包时要使用的模块
  module: {
    // 指定要加载的规则
    rules: [
      {
        // 指定的是规则生效的文件
        test: /\.ts$/,
        exclude: /node_modules/,
        use: [
          // 配置 babel
          {
            // 指定加载器
            loader: 'babel-loader',
            options: {
              presets: [
                [
                  // 指定环境的插件
                  '@babel/preset-env',
                  {
                    // 要兼容的目标浏览器
                    targets: {
                      'chrome': '58',
                      'ie': '11'
                    },
                    //指定corejs额版本
                    'corejs': '3',
                    // 使用corejs的方式 'usage' 表示按需加载
                    'useBuiltIns': 'usage'
                  }
                ]
              ]
            },
            loader: 'ts-loader'
          }
        ]
      }
    ]
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'TS测试',
    }),
  ],
```

```
  // 用来设置引用模块，表示 ts 文件和 js 文件都可以作为模块化使用
  resolve: {
    extensions: ['.ts', '.js']
  },
  optimizetion: {
    minimize: false // 关闭代码压缩，可选
  },
  devServer: {
    contentBase: './dist',
    port: 5000,
  },
  devtool: 'inline-source-map',
}
```

- 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。

第二章：面向对象

面向对象是程序中一个非常重要的思想，它被很多同学理解成了一个比较难，比较深奥的问题，其实不然。面向对象很简单，简而言之就是程序之中所有的操作都需要通过对象来完成。

- 举例来说：
  - 操作浏览器要使用 window 对象
  - 操作网页要使用 document 对象
  - 操作控制台要使用 console 对象

一切操作都要通过对象，也就是所谓的面向对象，那么对象到底是什么呢？这就要先说到程序是什么，计算机程序的本质就是对现实事物的抽象，抽象的反义词是具体，比如：照片是对一个具体的人的抽象，汽车模型是对具体汽车的抽象等等。程序也是对事物的抽象，在程序中我们可以表示一个人、一条狗、一把枪、一颗子弹等等所有的事物。一个事物到了程序中就变成了一个对象。

在程序中所有的对象都被分成了两个部分：数据和功能，以人为例，人的姓名、性别、年龄、身高、体重等属于数据，人可以说话、走路、吃饭、睡觉等这些行为属于人的功能。数据在对象中被称为属性，而功能就被称为方法。

所以简而言之，在程序中一切皆是对象。

1. 类（class）

要想面向对象，操作对象，首先要拥有对象，那么下一个问题就是如何创建对象。要创建对象，必须要先定义类，所谓的类可以理解为对象的模型，程序中可以根据类创建指定类型的对象。举例来说：可以通过 Person 类来创建人的对象，通过 Dog 类来创建狗的对象，通过 Car 类来创建汽车的对象，不同的类可以用来创建不同的对象。

- 定义类

```
class 类名 {
```

```
  属性名: 类型

  // 构造函数
  // 构造函数会在对象创建时调用
  constructor(参数: 类型) {
    // 在实例方法中，this就表示当前的实例
    // 在构造函数中当前对象就是当前新建的那个对象
    // 可以通过this向新建的对象中添加属性
    this.属性名 = 参数
  }

  方法名() {
    // 在方法中可以通过this来表示当前调用的对象
    ...
  }
}
```

- 示例：

```
class Person {
  name: string
  age: number

  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }

  sayHello() {
    console.log(`大家好，我是${this.name}`)
  }
}
```

- 使用类：

```
const p = new Person('孙悟空',  18')
p.sayHello()
```

2. 面向对象的特点

2.1. 封装

- 对象实质上就是属性和方法的容器，它的主要作用就是存储属性和方法，这就是所谓的封装
- 默认情况下，对象的属性是可以任意的修改的，为了确保数据的安全性，在 TS 中可以对属性的权限进行设置
- 只读属性（readonly）：
  - 如果在声明属性时添加一个 readonly，则属性便成了只读属性无法修改
- TS 中属性具有三种修饰符：
  - public（默认值），可以在类、子类和对象中修改
  - protected，可以在类、子类中修改，但不能在对象中修改
  - private，只能在类中修改

- 示例：
  - public
    - 可以在类、子类和对象中修改

```
class Person{
  public name: string // 写或者不写都是 public
  public age: number

  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }

  sayHello() {
    console.log(`大家好，我是${this.name}`)
  }
}

class Employee extends Person{
  constructor(name: string, age: number) {
    super(name, age)
    this.name = name // 子类中可以修改
  }
}

const p = new Person('孙悟空', 18)
p.name = '猪八戒' // 可以通过对象修改
```

- protected
  - 受保护的属性，只能在当前类和当前类的子类中访问（修改）
  - 不能在实例对象中访问（修改）

```
class Person{
  protected name: string
  protected age: number

  constructor(name: string, age: number) {
    this.name = name // 可以修改
    this.age = age
  }

  sayHello() {
    console.log(`大家好，我是${this.name}`)
  }
}

class Employee extends Person{
  constructor(name: string, age: number) {
    super(name, age)
    this.name = name // 子类中可以修改
  }
}

const p = new Person('孙悟空', 18)
p.name = '猪八戒' // 不能通过对象修改
```

- private
  - 私有属性，只能在类的内部进行访问（修改）
  - 通过在类中添加方法使得私有属性可以被外部访问

```
class Person{
  private name: string
  private age: number

  constructor(name: string, age: number) {
    this.name = name // 可以修改
    this.age = age
  }

  sayHello() {
    console.log(`大家好，我是${this.name}`)
  }
}

class Employee extends Person{
  constructor(name: string, age: number) {
    super(name, age)
    this.name = name
  }
}

const p = new Person('孙悟空', 18)
p.name = '猪八戒' // 不能通过对象和子类修改
```

- 语法糖

```
class C{

  // 可以直接将属性定义在构造函数中
  constructor(public name: string, public age: number) {
  }
}

// 以上代码等价于
/*
class C{
  name: string
  age: number
  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }
}
*/

const c = new C('Jack', 17)
```

- 属性存取器
  - 对于一些不希望被任意修改的属性，可以将其设置为 private

- 直接将其设置为 private 将导致无法通过对象修改其中的属性
- 我们可以在类中定义一组读取、设置属性的方法，这种对属性读取或设置的方法被称为属性的存取器
- 读取属性的方法叫做 getter 方法，设置属性的方法叫做 setter 方法
- 示例：

```
class Person{
  private _name: string

  constructor(name: string) {
    this._name = name
  }

  // 定义方法，用来获取 name 属性
  geName() {
    return this._name
  }

  // 定义方法，用来设置name 属性
  setName(value: string) {
    this._name = value
  }
}

const p1 = new Person('孙悟空')
console.log(p1.getName()) // 通过getter读取name 属性
p1.setName('猪八戒') // 通过setter修改name属性
```

- TS 中设置属性存取器

```
class Person{
  private _name: string

  constructor(name: string) {
    this._name = name
  }

  // TS 中设置getter方法
  get name(){
    console.log('get name()执行了')
    return this._name
  }

  set name(value: string){
    this._name = value
  }
}

const per = new Person('Jack', 18)
// 使用
console.log(per.name)
per.name = 'Rose'
```

- 静态属性

- 静态属性（方法），也成为类属性。使用静态属性无需创建实例，通过类即可直接使用
- 静态属性（方法）使用 static 开头
- 示例：

```
class Tools{
  /*
    直接定义的属性是实例属性，需要通过对象的实例去访问：
      const per = new Person()
      per.name
    使用 static 开头的属性是静态属性（类属性），可以直接通过类去访问
  */
  static PI = 3.1415926

  static sum(num1: number, num2: number) {
    return num1 + num2
  }
}

console.log(Tools.PI)
console.log(Tools.sum(1, 2))
```

- this
  - 在类中，使用 `this` 表示当前的实例对象

2.2. 继承

- 继承是面向对象的又一特性
- 通过继承可以将其他类中的属性和方法引入到当前类中
  - 示例：

```
class Animal{
  name: string
  age: number

  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }
}

/*
  此时，Animal被称为父类，Dog被称为子类
  使用继承后，子类将会拥有父类中所有的方法和属性
  通过继承可以将多个类中共有的代码写在一个父类中，
    这样只需要写一次即可让所有的子类都同时拥有父类中的属性和方法
    如果希望在子类中添加一些父类没有的属性或方法直接加就可以了
  如果在子类中添加了和父类相同名称的方法，则子类方法会覆盖父类的方法（并不会改变父类原本的方法）

*/
// 使Dog类继承Animal类
class Dog extends Animal{
```

```
  bark() {
    console.log(`${this.name}在汪汪叫! `)
  }
}

const dog = new Dog('旺财', 4)
dog.bark()
```

- 通过继承可以在不修改类的情况下完成对类的扩展
- 重写
  - 发生继承时，如果子类中的方法会替换父类中的同名方法，这就称为方法的重写
  - 如果在子类中添加了和父类相同名称的方法，则子类方法会覆盖父类的方法（并不会改变父类原本的方法）
  - 示例：

```
class Animal{
  name: string
  age: number

  constructor(name: string, age: number) {
    this.name = name
    this.age = age
  }

  run() {
    console.log(`父类中的run方法! `)
  }
}

class Dog extends Animal{
  bark() {
    console.log(`${this.name}在汪汪叫! `)
  }

  run() {
    console.log(`子类中的run方法，会重写父类中的run方法! `)
  }
}

const dog = new Dog('旺财', 2)
dog.bark()
dog.run()
```

- super关键字
  - 在子类中可以使用 super 来完成对父类的引用
  - 父类也叫超类（super）
  - super 在子类中表示当前类的父类

```
class Animal{
  name: string
  age: number

  constructor(name: string, age: number) {
    this.name = name
```

```
    this.age = age
  }

  sayHello() {
    console.log('动物在叫')
  }
}

class Dog extends Animal{
  color: string

  constructor(name: string, age: number, color: string) {
    // 如果在子类中写了构造函数，在子类构造函数中必须对父类的构造函数进行调用
    super(name: string, age: number) // 调用父类的构造函数
    this.color = color
  }


  sayHello() {
    // 在类方法中 super 就表示当前类的父类
    super.sayHello()
  }
}

const dog = new Dog('旺财', 2, 'yellow')
console.log(dog.color)
```

- 抽象类（abstract class）
  - 抽象类是专门用来被其他类所继承的类，他只能被其他类继承不能用来创建实例

```
abstract class Animal{
  name: string
  constructor(name: string) {
    this.name = name
  }

  // 定义一个抽象方法
  // 抽象方法使用 abstract 开头，没有方法体
  // 抽象方法只能定义在抽象类中，子类必须对抽象方法进行重写
  abstract run(): void
}

class Dog extends Animal{
  run() {
    console.log('狗在跑~')
  }
}
```

  - 使用 abstract 开头的方法叫做抽象方法，抽象方法没有方法体只能定义在抽象类中，继承抽象类时抽象方法必须要实现。

3. 接口（Interface）

接口的作用类似于抽象类，不同点在于接口中的所有方法和属性都是没有实值的，换句话说接口中的所有方法都是抽象方法。接口主要定义一个类的结构，接口可以去限制一个对象的接口，对象只有包含接口中定义的所有属性和方法是才能匹配接口。同时，可以让 一个类去实现接口，实现接口时类中要保护接口中的所有属性。

- 示例（检查对象类型）：

```
/*
  接口用来定义一个类的结构，用来定义一个类中包含哪些属性和方法
    同时接口也可以当成（对象）类型声明去使用，区别是接口是可以重复声明的
*/
interface myInterface {
  name: string
  age: number
}

const obj: my Interface = {
  name: 'aaa',
  age: 11
}


interface Person{
  name: string
  sayHello(): void
}

function fn(per: Person) {
  per.sayHello()
}

fn({name: '孙悟空', sayHello() {console.log(`Hello，我是${this.name}`)}})
```

- 示例（实现）

```
/*
  接口可以在定义类的时候去限制类的结构
    接口中的所有属性都不能有实际的值
    接口只定义对象的结构，而不考虑实际值（类似抽象类，不同的是抽象类中可以有抽象方法，也可以有实质方法）
      在接口中所有的方法都是抽线方法
*/
interface Person{
  name: string
  sayHello(): void
}

/*
  定义类时，可以使类去实现一个接口
    实现接口就是使类满足接口的要求
*/
class Student implements Person{
  constructor(public name: string) {
    this.name = name
```

```
  }

  sayHello() {
    console.log('大家好，我是' + this.name)
  }
}
```

- 接口和抽象类的区别
  - 在抽象类中可以有抽象方法，也可以有普通方法， 但是接口中只有抽象方法
  - 抽象类使用用 extends，接口使用 implements
  - 注意：接口和抽象类都是 TS 新增的，在 JavaScript 中没有

---

4. 泛型（Generic）

定义一个函数或类是，有些情况下无法确定其中要使用的具体类型（返回值、参数、属性的类型不能确定），此时使用泛型能够发挥重要作用。

- 举个例子：

```
function test(arg: any): any {
  return arg
}
```

  - 上例中，test 函数有一个参数类型不确定，但是能确定的是其返回值的类型和参数的类型是相同的，由于类型不确定所以参数和返回值均使用了 any，但是很明显这样做是不合适的，首相使用 any 会关闭 TS 的类型检查，其次这样设置也不能体现参数和返回值是相同的类型
  - 使用泛型：

```
function test<T>(arg: T): T{
  return arg
}
```

  - 这里 `<T>` 就是泛型，T 是我们给这个类型起的名字（不一定非叫 T），设置反省后即可在函数中使用 T 来表示该类型。所以泛型其实很好理解，就表示某个类型。
  - 那么如何使用上边的函数呢?
    - 方式一（直接使用）：

```
test(10)
```

      - 使用时可以直接传递参数使用，类型会由 TS 自动推断出来，但有时编译器无法自动推断是还需要使用下面的方式
    - 方式二（指定类型）：

```
test<number>(10)
```

      - 也可以在函数后手动指定泛型
  - 可以同时指定多个泛型，泛型间使用都好隔开：

```
function test<T, K>(a: T, b: K): K{
  return b
}

test<number, string>(10, 'hello')
```

    - 使用泛型时，完全可以将泛型当成一个普通的类去使用
- 类中同样可以使用泛型：

```
class MyCladd<T>{
  prop: T

  constructor(prop: T) {
    this.prop = prop
  }
}
```

- 除此之外，也可以对泛型的范围进行约束

```
interface MyInter{
  length: number
}

// 表示传入的参数必须有length属性
function test<T extends MyInter>(arg: T): number{
  return arg.length
}
```

    - 使用 T extends MyInter 表示泛型 T 必须是 MyInter 的实现类（子类），不一定非要使用接口类，对抽象类同样使用。