

# TypeScript知识扩展

王红元 coderwhy

# 目录

## content



**1** TypeScript模块使用

**2** TypeScript命名空间

**3** 内置声明文件的使用

**4** 第三方库声明的文件

**5** 编写自定义声明文件

**6** tsconfig配置文件解析

# TypeScript模块化

- JavaScript 有一个很长的处理模块化代码的历史，TypeScript 从 2012 年开始跟进，现在已经实现支持了很多格式。但是随着时间流逝，社区和 JavaScript 规范已经使用为名为 ES Module 的格式，这也就是我们所知的 import/export 语法。
  - ES 模块在 2015 年被添加到 JavaScript 规范中，到 2020 年，大部分的 web 浏览器和 JavaScript 运行环境都已经广泛支持。
  - 所以在TypeScript中最主要使用的模块化方案就是ES Module；

```
ck_dev > src > utils > TS math.ts > ...  
export function add(num1: number, num2: number) {  
  return num1 + num2  
}  
  
export function sub(num1: number, num2: number) {  
  return num1 - num2  
}
```

- 在前面我们已经学习过各种各样模块化方案以及对应的细节，这里我们主要学习TypeScript中一些比较特别的细节。

# 非模块 (Non-modules)

- 我们需要先理解 TypeScript 认为什么是一个模块。

- JavaScript 规范声明任何没有 `export` 的 JavaScript 文件都应该被认为是一个脚本，而非一个模块。
- 在一个脚本文件中，变量和类型会被声明在共享的全局作用域，将多个输入文件合并成一个输出文件，或者在 HTML 使用多个 `<script>` 标签加载这些文件。

- 如果你有一个文件，现在没有任何 `import` 或者 `export`，但是希望你希望它被作为模块处理，添加这行代码：

```
export {}
```

- 这会把文件改成一个没有导出任何内容的模块，这个语法可以生效，无论你的模块目标是什么。

# 内置类型导入 (Inline type imports)

- TypeScript 4.5 也允许单独的导入，你需要使用 `type` 前缀，表明被导入的是一个类型：

```
import { type IFoo, type IDType } from "./foo"

const id: IDType = 100
const foo: IFoo = {
  name: "why",
  age: 18
}
```

- 这些可以让一个非 TypeScript 编译器比如 Babel、swc 或者 esbuild 知道什么样的导入可以被安全移除。

Together these allow a non-TypeScript transpiler like Babel, swc or esbuild to know what imports can be safely removed.

# 命名空间namespace（了解）

■ TypeScript 有它自己的模块格式，名为 namespaces，它在 ES 模块标准之前出现。

- 命名空间在TypeScript早期时，称之为内部模块，目的是将一个模块内部再进行作用域的划分，防止一些命名冲突的问题；
- 虽然命名空间没有被废弃，但是由于 ES 模块已经拥有了命名空间的大部分特性，因此更推荐使用 ES 模块，这样才能与 JavaScript 的（发展）方向保持一致。

```
export namespace Time {  
  export function format(time: string) {  
    return "2022-10-10"  
  }  
  
  export const name = "time"  
}
```

```
export namespace Price {  
  export function format(price: string) {  
    return "¥20.00"  
  }  
}
```

useful features for creating complex definition files, and still sees active use [in DefinitelyTyped](#). While not deprecated, the majority of the features in namespaces exist in ES Modules and we recommend you use that to align with JavaScript's direction.

# 类型的查找

- 之前我们所有的typescript中的类型，几乎都是我们自己编写的，但是我们也有用到一些其他的类型：

```
const imageEl = document.getElementById("image") as HTMLImageElement;
```

- 大家是否会奇怪，我们的HTMLImageElement类型来自哪里呢？甚至是document为什么可以有getElementById的方法呢？

- 其实这里就涉及到typescript对类型的管理和查找规则了。

- 我们这里先给大家介绍另外一种typescript文件：.d.ts文件

- 我们之前编写的typescript文件都是 .ts 文件，这些文件最终会输出 .js 文件，也是我们通常编写代码的地方；

- 还有另外一种文件 .d.ts 文件，它是用来做类型的声明(declare)，称之为类型声明 (Type Declaration) 或者类型定义 (Type Definition) 文件。

- 它仅仅用来做类型检测，告知typescript我们有哪些类型；

- 那么typescript会在哪里查找我们的类型声明呢？

- 内置类型声明；

- 外部定义类型声明；

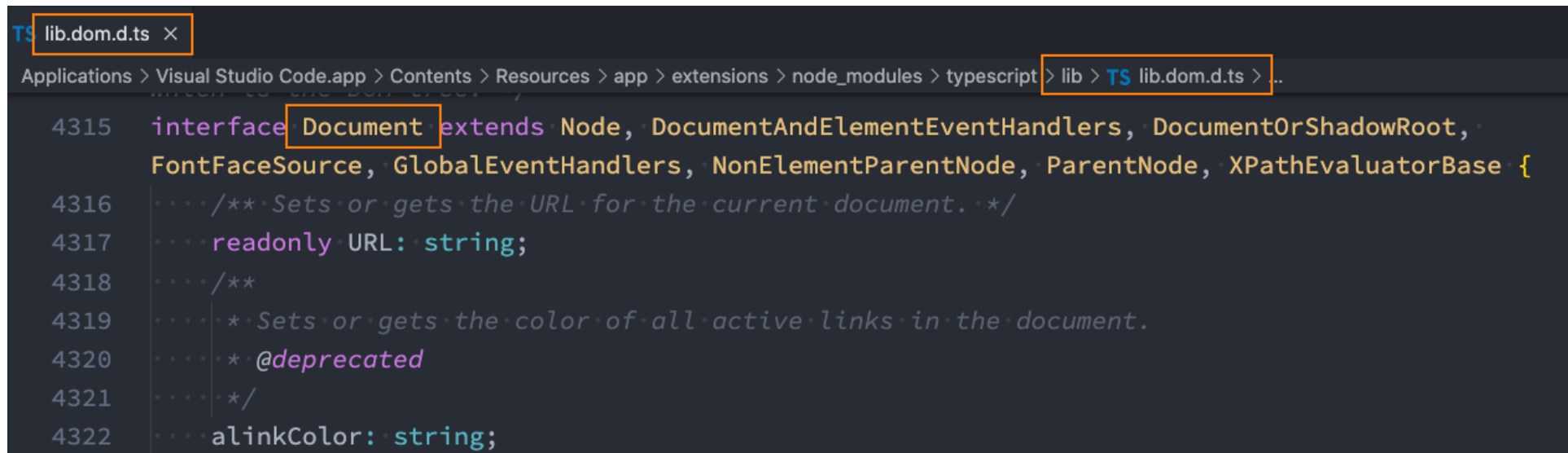
- 自己定义类型声明；

# 内置类型声明

■ 内置类型声明是typescript自带的、帮助我们内置了JavaScript运行时的一些标准化API的声明文件；

- 包括比如Function、String、Math、Date等内置类型；
- 也包括运行环境中的DOM API，比如Window、Document等；

■ TypeScript 使用模式命名这些声明文件lib.[something].d.ts。



```
TS lib.dom.d.ts x
Applications > Visual Studio Code.app > Contents > Resources > app > extensions > node_modules > typescript > lib > TS lib.dom.d.ts > ..
4315 interface Document extends Node, DocumentAndElementEventHandlers, DocumentOrShadowRoot,
FontFaceSource, GlobalEventHandlers, NonElementParentNode, ParentNode, XPathEvaluatorBase {
4316     /** Sets or gets the URL for the current document. */
4317     readonly URL: string;
4318     /**
4319      * Sets or gets the color of all active links in the document.
4320      * @deprecated
4321      */
4322     aLinkColor: string;
```

■ 内置类型声明通常在我们安装typescript的环境中会带有的；

- <https://github.com/microsoft/TypeScript/tree/main/lib>



## ■ 我们可以通过target和lib来决定哪些内置类型声明是可以使用的：

□ 例如，startsWith字符串方法只能从称为ECMAScript 6的 JavaScript 版本开始使用；

## ■ 我们可以通过target的编译选项来配置：TypeScript通过lib根据您的target设置更改默认包含的文件来帮助解决此问题。

□ <https://www.typescriptlang.org/tsconfig#lib>

Name	Contents
ES5	Core definitions for all ES3 and ES5 functionality
ES2015	Additional APIs available in ES2015 (also known as ES6) - array.find, Promise, Proxy, Symbol, Map, Set, Reflect, etc.
ES6	Alias for "ES2015"
ES2016	Additional APIs available in ES2016 - array.include, etc.
ES7	Alias for "ES2016"
ES2017	Additional APIs available in ES2017 - Object.entries, Object.values, Atomics, SharedArrayBuffer, date.formatToParts, typed arrays, etc.
ES2018	Additional APIs available in ES2018 - async iterables, promise.finally, Intl.PluralRules, regexp.groups, etc.
ES2019	Additional APIs available in ES2019 - array.flat, array.flatMap, Object.fromEntries, string.trimStart, string.trimEnd, etc.
ES2020	Additional APIs available in ES2020 - string.matchAll, etc.
ES2021	Additional APIs available in ES2021 - promise.any, string.replaceAll etc.
ESNext	Additional APIs available in ESNext - This changes as the JavaScript specification evolves

# 外部定义类型声明 – 第三方库

- 外部类型声明通常是我们使用一些库（比如第三方库）时，需要的一些类型声明。
- 这些库通常有两种类型声明方式：
  - 方式一：在**自己库中进行类型声明**（编写.d.ts文件），比如axios
  - 方式二：通过**社区的一个公有库DefinitelyTyped存放类型声明文件**
    - 该库的GitHub地址：<https://github.com/DefinitelyTyped/DefinitelyTyped/>
    - 该库查找声明安装方式的地址：<https://www.typescriptlang.org/dt/search?search=>
    - 比如我们安装react的类型声明： `npm i @types/react --save-dev`

TypeScript automatically finds type definitions under `node_modules/@types`, so there's no other step needed to get these types available in your program.

```
import React from "react"
import axios from "axios"
```

# 外部定义类型声明 – 自定义声明

## ■ 什么情况下需要自己来定义声明文件呢？

- 情况一：我们使用的第三方库是一个纯的JavaScript库，没有对应的声明文件；比如lodash
- 情况二：我们给自己的代码中声明一些类型，方便在其他地方直接进行使用；

```
let wName = "coderwhy"
let wAge = 18
let wHeight = 1.88

function wFoo() {
  console.log("wFoo")
}

function wBar() {
  console.log("wBar")
}

function Person(name, age) {
  this.name = name
  this.age = age
}
```

```
declare let wName: string;
declare let wAge: number;
declare let wHeight: number

declare function wFoo(): void
declare function wBar(): void

declare class Person {
  name: string
  age: number

  constructor(name: string, age: number)
}
```

# declare 声明模块

- 我们也可以声明模块，比如lodash模块默认不能使用的情况，可以自己来声明这个模块：

```
declare module "lodash" {  
  export function join(args: any[]): any;  
}
```

- 声明模块的语法: **declare module '模块名' {}**。
  - 在声明模块的内部，我们可以通过 **export** 导出对应库的类、函数等；

# declare 声明文件

## ■ 在某些情况下，我们也可以声明文件：

- 比如在开发vue的过程中，默认是不识别我们的.vue文件的，那么我们就需要对其进行文件的声明；
- 比如在开发中我们使用了 jpg 这类图片文件，默认typescript也是不支持的，也需要对其进行声明；

```
declare module '*.vue' {  
  import { DefineComponent } from 'vue'  
  const component: DefineComponent  
  
  export default component  
}  
  
declare module '*.jpg' {  
  const src: string  
  export default src  
}
```

# declare 命名空间

- 比如我们在index.html中直接引入了jQuery:

- CDN地址: <https://cdn.bootcdn.net/ajax/libs/jquery/3.6.0/jquery.js>

- 我们可以进行命名空间的声明:

```
declare namespace $ {  
  function ajax(settings: any): void  
}
```

- 在main.ts中就可以使用了:

```
$.ajax({  
  url: "http://123.207.32.32:8000/home/multidata",  
  success: (res: any) => {  
    console.log(res);  
  },  
});
```

# 认识tsconfig.json文件

## ■ 什么是tsconfig.json文件呢？（官方的解释）

- 当目录中出现了 tsconfig.json 文件，则说明该目录是 TypeScript 项目的根目录；
- tsconfig.json 文件指定了编译项目所需的根目录下的文件以及编译选项。

## ■ 官方的解释有点“官方”，直接看我的解释。

## ■ tsconfig.json文件有两个作用：

- **作用一（主要的作用）**：让TypeScript Compiler在编译的时候，知道如何去编译TypeScript代码和进行类型检测；
  - ✓ 比如是否允许不明确的this选项，是否允许隐式的any类型；
  - ✓ 将TypeScript代码编译成什么版本的JavaScript代码；
- **作用二**：让编辑器（比如VSCode）可以按照正确的方式识别TypeScript代码；
  - ✓ 对于哪些语法进行提示、类型错误检测等等；

## ■ JavaScript 项目可以使用 jsconfig.json 文件，它的作用与 tsconfig.json 基本相同，只是默认启用了一些 JavaScript 相关的编译选项。

- 在之前的Vue项目、React项目中我们也有使用过；



# tsconfig.json配置

## ■ tsconfig.json在编译时如何被使用呢？

- 在调用 `tsc` 命令并且没有其它输入文件参数时，编译器将由当前目录开始向父级目录寻找包含 `tsconfig` 文件的目录。
- 调用 `tsc` 命令并且没有其他输入文件参数，可以使用 `--project`（或者只是 `-p`）的命令行选项来指定包含了 `tsconfig.json` 的目录；
- 当命令行中指定了输入文件参数，`tsconfig.json` 文件会被忽略；

## ■ webpack中使用ts-loader进行打包时，也会自动读取tsconfig文件，根据配置编译TypeScript代码。

## ■ tsconfig.json文件包括哪些选项呢？

- `tsconfig.json`本身包括的选项非常非常多，我们不需要每一个都记住；
- 可以查看文档对于每个选项的解释：<https://www.typescriptlang.org/tsconfig>
- 当我们开发项目的时候，选择TypeScript模板时，`tsconfig`文件默认都会帮助我们配置好的；

## ■ 接下来我们学习一下哪些重要的、常见的选项。



# tsconfig.json 顶层选项



```
{  
  "include": ["src/**/*", "tests/**/*"]  
}
```

Which would include:

```
.  
├── scripts  
│   ├── lint.ts  
│   ├── update_deps.ts  
│   └── utils.ts  
├── src  
│   ├── client  
│   │   ├── index.ts  
│   │   └── utils.ts  
│   └── server  
│       └── index.ts  
├── tests  
│   ├── app.test.ts  
│   ├── utils.ts  
│   └── tests.d.ts  
├── package.json  
├── tsconfig.json  
└── yarn.lock
```

# tsconfig.json文件

- tsconfig.json是用于配置TypeScript编译时的配置选项：

- <https://www.typescriptlang.org/tsconfig>

- 我们这里讲解几个比较常见的：

```
{
  "compilerOptions": {
    // 目标代码
    "target": "esnext",
    // 生成代码使用的模块化
    "module": "esnext",
    // 打开所有的严格模式检查
    "strict": true,
    "allowJs": false,
    "noImplicitAny": false,
    // jsx的处理方式(保留原有的jsx格式)
    "jsx": "preserve",
    // 是否帮助导入一些需要的功能模块
    "importHelpers": true,
    // 按照node的模块解析规则
    // https://www.typescriptlang.org/docs/handbook/module-resolution.html#module-resolution-strategies
    "moduleResolution": "node",
    // 跳过对整个库进行类似检测, 而仅仅检测你用到的类型
    "skipLibCheck": true,
```

# tsconfig.json文件

```
... // 可以让es module 和 commonjs 相互调用
... "esModuleInterop": true,
... // 允许合成默认模块导出
... // import * as react from 'react': false
... // import react from 'react': true
... "allowSyntheticDefaultImports": true,
... // 是否要生成sourcemap文件
... "sourceMap": true,
... // 文件路径在解析时的基本url
... "baseUrl": ".",
... // 指定types文件需要加载哪些(默认是都会进行加载的)
... // "types": [
... // ... "webpack-env"
... // ],
... // 路径的映射设置, 类似于webpack中的 alias
... "paths": {
...   "@/*": ["src/*"]
... },
... // 指定我们需要使用到的库(也可以不配置, 直接根据target来获取)
... "lib": ["esnext", "dom", "dom.iterable", "scripthost"]
... },
... "include": [ ...
... ],
... "exclude": ["node_modules"]
}
```