d.ts 类型声明文件

单独使用的模块，一般会同时提供一个单独的类型声明文件（declaration file），把本模块的外部接口的所有类型都写在这个文件里面，便于模块使用者了解接口，也便于编译器检查使用者的用法是否正确。

类型声明文件里面只有类型代码，没有具体的代码实现。它的文件名一般为 `[模块名].d.ts` 的形式，其中的 `d` 表示 `declaration`（声明）。

```
// module a
const maxInterval = 12;
function getArrayLength(arr) {
  return arr.length;
}
module.exports = {
  getArrayLength,
  maxInterval,
};
```

它的类型声明文件可以写成：

```
export function getArrayLength(arr: any[]): number;
export const maxInterval: 12;
```

类型声明文件也可以使用 `export =` 命令，输出对外接口。`moment` 模块的类型声明文件的例子：

```
declare module 'moment' {
  function moment(): any;
  export = moment;
}
```

上例中，模块 `moment` 内部有一个函数 `moment()`，而 `export =` 表示 `module.exports` 输出的就是这个函数。

除了使用 `export =`，模块输出在类型声明文件中，也可以使用 `export default` 表示。

```
// 模块输出
module.exports = 3.142;

// 类型输出文件
// 写法一
declare const pi: number;
export default pi;

// 写法二
```

```
declare const pi: number;
export = pi;
```

上例中，模块输出的是一个整数，那么可以用 `export default` 或 `export =` 表示输出这个值。

```
// types.d.ts
export interface Character {
  catchphrase?: string;
  name: string;
}
```

然后，就可以在 TypeScript 脚本里面导入该文件声明的类型。

```
// index.ts
import { Character } from "./types";
export const character:Character = {
  catchphrase: "Yee-haw!",
  name: "Sandy Cheeks",
};
```

类型声明文件也可以包括在项目的 `tsconfig.json` 文件里面，这样的话，编译器打包项目时，会自动将类型声明文件加入编译，而不必在每个脚本里面加载类型声明文件。比如，`moment` 模块的类型声明文件是 `moment.d.ts`，使用 `moment` 模块的项目可以将其加入项目的 `tsconfig.json` 文件。

```
{
  "compilerOptions": {},
  "files": [
    "src/index.ts",
    "typings/moment.d.ts"
  ]
}
```

1. 类型声明文件的来源

类型声明文件主要有以下三种来源。

- TypeScript 编译器自动生成。
- TypeScript 内置类型文件。
- 外部模块的类型声明文件，需要自己安装。

1.1. 自动生成

只要使用编译选项 `declaration`，编译器就会在编译时自动生成单独的类型声明文件。

`tsconfig.json` 文件里面，打开这个选项。

```
{
  "compilerOptions": {
    "declaration": true
  }
}
```

你也可以在命令行打开这个选项。`tsc --declaration`

1.2. 内置声明文件

安装 TypeScript 语言时，会同时安装一些内置的类型声明文件，主要是内置的全局对象（JavaScript 语言接口和运行环境 API）的类型声明。

这些内置声明文件位于 TypeScript 语言安装目录的 `lib` 文件夹内，数量大概有几十个，下面是其中一些主要文件。

> - lib.d.ts
> - lib.dom.d.ts
> - lib.es2015.d.ts
> - lib.es2016.d.ts
> - lib.es2017.d.ts
> - lib.es2018.d.ts
> - lib.es2019.d.ts
> - lib.es2020.d.ts
> - lib.es5.d.ts
> - lib.es6.d.ts

这些内置声明文件的文件名统一为“`lib.[description].d.ts`”的形式，其中 `description` 部分描述了文件内容。比如，`lib.dom.d.ts` 这个文件就描述了 `DOM` 结构的类型。

TypeScript 编译器会自动根据编译目标 `target` 的值，加载对应的内置声明文件，所以不需要特别的配置。但是，可以使用编译选项 `lib`，指定加载哪些内置声明文件。

```
{
  "compilerOptions": {
    "lib": ["dom", "es2021"]
  }
}
```

上例中，`lib` 选项指定加载 `dom` 和 `es2021` 这两个内置类型声明文件。

编译选项 `noLib` 会禁止加载任何内置声明文件。

1.3. 外部类型声明文件

如果项目中使用了外部的某个第三方代码库，那么就需要这个库的类型声明文件。

这时又分成三种情况。

- （1）这个库自带了类型声明文件。

一般来说，如果这个库的源码包含了 `[vendor].d.ts` 文件，那么就自带了类型声明文件。其中的 `vendor` 表示这个库的名字，比如 `moment` 这个库就自带 `moment.d.ts`。使用这个库可能需要单独加载它的类型声明文件。

- （2）这个库没有自带，但是可以找到社区制作的类型声明文件。

第三方库如果没有提供类型声明文件，社区往往会提供。TypeScript 社区主要使用 DefinitelyTyped 仓库，各种类型声明文件都会提交到那里，已经包含了几千个第三方库。

这些声明文件都会作为一个单独的库，发布到 `npm` 的 `@types` 名称空间之下。比如，`jQuery` 的类型声明文件就发布成 `@types/jquery` 这个库，使用时安装这个库就可以了。

```
npm install @types/jquery --save-dev
```

执行上面的命令，`@types/jquery` 这个库就安装到项目的 `node_modules/@types/jquery` 目录，里面的 `index.d.ts` 文件就是 jQuery 的类型声明文件。如果类型声明文件不是 `index.d.ts`，那么就需要在 `package.json` 的 `types` 或 `typings` 字段，指定类型声明文件的文件名。

TypeScript 会自动加载node_modules/@types目录下的模块，但可以使用编译选项 `typeRoots` 改变这种行为。

```
{
  "compilerOptions": {
    "typeRoots": ["./typings", "./vendor/types"]
  }
}
```

上例表示，TypeScript 不再去 `node_modules/@types` 目录，而是去跟当前 `tsconfig.json` 同级的 `typings` 和 `vendor/types` 子目录，加载类型模块了。

默认情况下，TypeScript 会自动加载typeRoots目录里的所有模块，编译选项types可以指定加载哪些模块。

```
{
  "compilerOptions": {
    "types" : ["jquery"]
  }
}
```

上面设置中，`types` 属性是一个数组，成员是所要加载的类型模块，要加载几个模块，这个数组就有几个成员，每个类型模块在 `typeRoots` 目录下都有一个自己的子目录。这样的话，TypeScript 就会自动去 `jquery` 子目录，加载 `jQuery` 的类型声明文件。

- （3）找不到类型声明文件，需要自己写。

有时实在没有第三方库的类型声明文件，又很难完整给出该库的类型描述，这时你可以告诉 TypeScript 相关对象的类型是 `any`。比如，使用 `jQuery` 的脚本 `declare var $:any`。

```
// 或者
declare type JQuery = any;
declare var $:JQuery;
```

上面代码表示，`jQuery` 的 `$` 对象是外部引入的，类型是 `any`，也就是 TypeScript 不用对它进行类型检查。

也可以将整个外部模块的类型设为 any。`declare module '模块名';`

有了上面的命令，指定模块的所有接口都将视为 any 类型。

2. declare 关键字

类型声明文件只包含类型描述，不包含具体实现，所以非常适合使用 `declare` 语句来描述类型。

类型声明文件里面，变量的类型描述必须使用 `declare` 命令，否则会报错，因为变量声明语句是值相关代码。

```
declare let foo:string;
```

interface 类型有没有 `declare` 都可以，因为 interface 是完全的类型代码。

```
interface Foo {} // 正确
declare interface Foo {} // 正确
```

类型声明文件里面，顶层可以使用 `export` 命令，也可以不用，除非使用者脚本会显式使用 `export` 命令输入类型。

```
export interface Data {
  version: string;
}
```

一些类型声明文件：

```
// moment.d.ts
declare module 'moment' {
  export interface Moment {
    format(format:string): string;

    add(
      amount: number,
      unit: 'days' | 'months' | 'years'
    ): Moment;

    subtract(
      amount:number,
```

```
      unit:'days' | 'months' | 'years'
    ): Moment;
  }

  function moment(input?: string | Date): Moment;

  export default moment;
}
```

```
// D3.d.ts
declare namespace D3 {
  export interface Selectors {
    select: {
      (selector: string): Selection;
      (element: EventTarget): Selection;
    };
  }

  export interface Event {
    x: number;
    y: number;
  }

  export interface Base extends Selectors {
    event: Event;
  }
}

declare var d3: D3.Base;
```

3. 模块发布

当前模块如果包含自己的类型声明文件，可以在 `package.json` 文件里面添加一个 `types` 字段或 `typings` 字段，指明类型声明文件的位置。

```
{
  "name": "awesome",
  "author": "Vandelay Industries",
  "version": "1.0.0",
  "main": "./lib/main.js",
  "types": "./lib/main.d.ts"
}
```

上例中，`types` 字段给出了类型声明文件的位置。

如果类型声明文件名为index.d.ts，且在项目的根目录中，那就不需要在package.json里面注明了。

有时，类型声明文件会单独发布成一个 `npm` 模块，这时用户就必须同时加载该模块。

```
{
  "name": "browserify-typescript-extension",
  "author": "Vandelay Industries",
  "version": "1.0.0",
  "main": "./lib/main.js",
  "types": "./lib/main.d.ts",
  "dependencies": {
    "browserify": "latest",
    "@types/browserify": "latest",
    "typescript": "next"
  }
}
```

上例是一个模块的 `package.json` 文件，该模块需要 `browserify` 模块。由于后者的类型声明文件是一个单独的模块 `@types/browserify`，所以还需要加载那个模块。

4. 三斜杠命令

如果类型声明文件的内容非常多，可以拆分成多个文件，然后入口文件使用三斜杠命令，加载其他拆分后的文件。

举例来说，入口文件是 `main.d.ts`，里面的接口定义在 `interfaces.d.ts`，函数定义在 `functions.d.ts`。那么，`main.d.ts` 里面可以用三斜杠命令，加载后面两个文件。

```
/// <reference path="./interfaces.d.ts" />
/// <reference path="./functions.d.ts" />
```

三斜杠命令（`///`）是一个 TypeScript 编译器命令，用来指定编译器行为。它只能用在文件的头部，如果用在其他地方，会被当作普通的注释。**若一个文件中使用了三斜线命令，那么在三斜线命令之前只允许使用单行注释、多行注释和其他三斜线命令，否则三斜杠命令也会被当作普通的注释。**

除了拆分类型声明文件，三斜杠命令也可以用于普通脚本加载类型声明文件。

三斜杠命令主要包含三个参数，代表三种不同的命令。

> - path
> - types
> - lib

`/// <reference path="" />` 是最常见的三斜杠命令，告诉编译器在编译时需要包括的文件，常用来声明当前脚本依赖的类型文件。

```
/// <reference path="./lib.ts" />
let count = add(1, 2);
```

上例中，当前脚本依赖于 `./lib.ts`，里面是 `add()` 的定义。编译当前脚本时，还会同时编译 `./lib.ts`。编译产物会有两个 JS 文件，一个当前脚本，另一个就是 `./lib.js`。

```
// 当前脚本依赖于 Node.js 类型声明文件:
/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("https://www.typescriptlang.org");
```

编译器会在预处理阶段，找出所有三斜杠引用的文件，将其添加到编译列表中，然后一起编译。

`path` 参数指定了所引入文件的路径。如果该路径是一个相对路径，则基于当前脚本的路径进行计算。

使用该命令时，`path` 参数必须指向一个存在的文件，若文件不存在会报错。`path` 参数不允许指向当前文件。

默认情况下，每个三斜杠命令引入的脚本，都会编译成单独的 JS 文件。如果希望编译后只产出一个合并文件，可以使用编译选项 `outFile`。但是，`outFile` 编译选项不支持合并 `CommonJS` 模块和 `ES` 模块，只有当编译参数 `module` 的值设为 `None`、`System` 或 `AMD` 时，才能编译成一个文件。

如果打开了编译参数noResolve，则忽略三斜杠指令。将其当作一般的注释，原样保留在编译产物中。

`/// <reference types="" />`，`types` 参数用来告诉编译器当前脚本依赖某个 `DefinitelyTyped` 类型库，通常安装在 `node_modules/@types` 目录。`types` 参数的值是类型库的名称，也就是安装到 `node_modules/@types` 目录中的子目录的名字。

```
/// <reference types="node" />
```

上例中，这个三斜杠命令表示编译时添加 `Node.js` 的类型库，实际添加的脚本是 `node_modules` 目录里面的 `@types/node/index.d.ts`。

这个命令的作用类似于 `import` 命令。

注意，这个命令只在你自己手写类型声明文件（`.d.ts` 文件）时，才有必要用到，也就是说，只应该用在 `.d.ts` 文件中，普通的.ts脚本文件不需要写这个命令。如果是普通的 `.ts` 脚本，可以使用 `tsconfig.json` 文件的 `types` 属性指定依赖的类型库。

`/// <reference lib="" />` 命令允许脚本文件显式包含内置 `lib` 库，等同于在 `tsconfig.json` 文件里面使用 `lib` 属性指定 `lib` 库。

安装 TypeScript 软件包时，会同时安装一些内置的类型声明文件，即内置的 `lib` 库。这些库文件位于 TypeScript 安装目录的 `lib` 文件夹中，它们描述了 JavaScript 语言和引擎的标准 API。

库文件并不是固定的，会随着 TypeScript 版本的升级而更新。库文件统一使用 `“lib.[description].d.ts”` 的命名方式，而 `/// <reference lib="" />` 里面的 `lib` 属性的值就是库文件名的 `description` 部分，比如 `lib="es2015"` 就表示加载库文件 `lib.es2015.d.ts`。

```
/// <reference lib="es2017.string" />
```

上例中，`es2017.string` 对应的库文件就是 `lib.es2017.string.d.ts`。