

模块

任何包含 `import` 或 `export` 语句的文件，就是一个模块（module）。相应地，如果文件不包含 `import` 和 `export` 语句，就是一个全局的脚本文件。

模块本身就是一个作用域，不属于全局作用域。模块内部的变量、函数、类只在内部可见，对于模块外部是不可见的。暴露给外部的接口，必须用 `export` 命令声明；如果其他文件要使用模块的接口，必须用 `import` 命令来输入。

如果一个文件不包含 `export` 语句，但是希望把它当作一个模块（即内部变量对外不可见），可以在脚本头部添加一行语句 `export {}`；

添加 `export {}`；后不产生任何实际作用，但会让当前文件被当作模块处理，所有它的代码都变成了内部代码。

TypeScript 模块除了支持所有 ES 模块的语法，特别之处在于允许输出和输入类型。

```
export type Bool = true | false;
// 等价于
type Bool = true | false;
export { Bool };
```

假定上面的模块文件为 `a.ts`，另一个文件 `b.ts` 就可以使用 `import` 语句，输入这个类型。

```
import { Bool } from './a';
let foo:Bool = true;
```

上例中，`import` 语句加载的是一个类型。注意，加载文件写成 `./a`，没有写脚本文件的后缀名。TypeScript 允许加载模块时，省略模块文件的后缀名，它会自动定位，将 `./a` 定位到 `./a.ts`。

编译时，可以两个脚本同时编译。

```
tsc a.ts b.ts
```

上面命令会将 `a.ts` 和 `b.ts` 分别编译成 `a.js` 和 `b.js`。也可以只编译 `b.ts`，因为它是入口脚本，`tsc` 会自动编译它依赖的所有脚本。

```
tsc b.ts
```

上面命令发现 `b.ts` 依赖 `a.ts`，就会自动寻找 `a.ts`，也将其同时编译，因此编译产物还是 `a.js` 和 `b.js` 两个文件。

如果想将 `a.ts` 和 `b.ts` 编译成一个文件，可以使用 `--outFile` 参数。

```
tsc --outFile result.js b.ts
```

上面示例将 `a.ts` 和 `b.ts` 合并编译为 `result.js`。

1. import type 语句

`import` 在一条语句中，可以同时输入类型和正常接口。

```
// a.ts
export interface A {
  foo: string;
}
export let a = 123;

// b.ts
import { A, a } from './a';
```

上例中，文件 `a.ts` 的 `export` 语句输出了一个类型 `A` 和一个正常接口 `a`，另一个文件 `b.ts` 则在同一条语句中输入了类型和正常接口。

这样很不利于区分类型和正常接口，容易造成混淆。为了解决这个问题，TypeScript 引入了两个解决方法。

- 第一个方法是在 `import` 语句输入的类型前面加上 `type` 关键字。

```
import { type A, a } from './a';
```

上面示例中，`import` 语句输入的类型 `A` 前面有 `type` 关键字，表示这是一个类型。

- 第二个方法是使用 `import type` 语句，这个语句只能输入类型，不能输入正常接口。

```
import type { A } from './a'; // 正确
import type { a } from './a'; // 报错
```

上例中，`import type` 输入类型 `A` 是正确的，但是输入正常接口 `a` 就会报错。

`import type` 语句也可以输入默认类型。

```
import type DefaultType from 'moduleA';
```

`import type` 在一个名称空间下，输入所有类型的写法如下。

```
import type * as TypeNS from 'moduleA';
```

同样的，`export` 语句也有两种方法，表示输出的是类型。

```
type A = 'a';
type B = 'b';

export {type A, type B}; // 方法一

export type {A, B};      // 方法二
```

上例中，方法一是使用 `type` 关键字作为前缀，表示输出的是类型；方法二是使用 `export type` 语句，表示整行输出的都是类型。

`export type` 将一个类作为类型输出的例子。

```
class Point {
  x: number;
  y: number;
}
export type { Point };
```

上例中，由于使用了 `export type` 语句，输出的并不是 `Point` 这个类，而是 `Point` 代表的实例类型。输入时，只能作为类型输入。

```
import type { Point } from './module';
const p:Point = { x: 0, y: 0 };
```

上例中，`Point` 只能作为类型输入，不能当作正常接口使用。

2. CommonJS 模块

CommonJS 是 Node.js 的专用模块格式，与 ES 模块格式不兼容。

2.1. import = 语句

TypeScript 使用 `import =` 语句输入 CommonJS 模块。

```
import fs = require('fs');
const code = fs.readFileSync('hello.ts', 'utf8');
```

上例中，使用 `import = 语句` 和 `require()` 命令输入了一个 `CommonJS` 模块。模块本身的用法跟 `Node.js` 是一样的。

除了使用 `import = 语句`，TypeScript 还允许使用 `import * as [接口名] from "模块文件"` 输入 `CommonJS` 模块。

```
import * as fs from 'fs';  
// 等价于  
import fs = require('fs');
```

2.2. export = 语句

TypeScript 使用 `export = 语句`，输出 `CommonJS` 模块的对象，等同于 `CommonJS` 的 `module.exports` 对象。

```
let obj = { foo: 123 };  
export = obj;
```

`export = 语句` 输出的对象，只能使用 `import = 语句` 加载。

```
import obj = require('./a');  
console.log(obj.foo); // 123
```

3. 模块定位

模块定位 (`module resolution`) 指的是确定 `import` 语句和 `export` 语句里面的模块文件位置。

```
import { TypeA } from './a';
```

上例中，TypeScript 怎么确定 `./a` 到底是指哪一个模块，这就叫做“模块定位”。

模块定位有两种方法，一种称为 `Classic` 方法，另一种称为 `Node` 方法。可以使用编译参数 `moduleResolution`，指定使用哪一种方法。

没有指定定位方法时，就看原始脚本采用什么模块格式。如果模块格式是 `CommonJS`（即编译时指定 `--module commonjs`），那么模块定位采用 `Node` 方法，否则采用 `Classic` 方法（模块格式为 `es2015`、`esnext`、`amd`、`system`、`umd` 等等）。

3.1. 相对模块，非相对模块

加载模块时，目标模块分为相对模块（`relative import`）和非相对模块两种（`non-relative import`）。

相对模块指的是路径以 `/`、`./`、`../` 开头的模块。下面 `import` 语句加载的模块，都是相对模块。

```
import Entry from "../components/Entry";
import { DefaultHeaders } from "../constants/http";
import "/mod";
```

非相对模块指的是不带有路径信息的模块。 下面 `import` 语句加载的模块，都是非相对模块。

```
import * as $ from "jquery";
import { Component } from "@angular/core";
```

3.2. Classic 方法

Classic 方法以当前脚本的路径作为“基准路径”，计算相对模块的位置。比如，脚本 `a.ts` 里面有一行代码 `import { b } from "./b"`，那么 TypeScript 就会在 `a.ts` 所在的目录，查找 `b.ts` 和 `b.d.ts`。

至于非相对模块，也是以当前脚本的路径作为起点，一层层查找上级目录。比如，脚本 `a.ts` 里面有一行代码 `import { b } from "b"`，那么就会查找 `b.ts` 和 `b.d.ts`。

3.3. Node 方法

Node 方法就是模拟 `Node.js` 的模块加载方法。

相对模块依然是以当前脚本的路径作为“基准路径”。比如，脚本文件 `a.ts` 里面有一行代码 `let x = require("./b");`，TypeScript 按照以下顺序查找。

- 当前目录是否包含 `b.ts`、`b.tsx`、`b.d.ts`。
- 当前目录是否有子目录 `b`，该子目录是否存在文件 `package.json`，该文件的 `types` 字段是否指定了入口文件，如果是的就加载该文件。
- 当前目录的子目录 `b` 是否包含 `index.ts`、`index.tsx`、`index.d.ts`。

非相对模块则是以当前脚本的路径作为起点，逐级向上层目录查找是否存在子目录 `node_modules`。比如，脚本文件 `a.js` 有一行 `let x = require("b");`，TypeScript 按照以下顺序进行查找。

- 当前目录的子目录 `node_modules` 是否包含 `b.ts`、`b.tsx`、`b.d.ts`。
- 当前目录的子目录 `node_modules`，是否存在文件 `package.json`，该文件的 `types` 字段是否指定了入口文件，如果是的就加载该文件。
- 当前目录的子目录 `node_modules` 里面，是否包含子目录 `@types`，在该目录中查找文件 `b.d.ts`。
- 当前目录的子目录 `node_modules` 里面，是否包含子目录 `b`，在该目录中查找 `index.ts`、`index.tsx`、`index.d.ts`。
- 进入上一层目录，重复上面4步，直到找到为止。

3.4. 路径映射

TypeScript 允许开发者在 `tsconfig.json` 文件里面，手动指定脚本模块的路径。

- (1) `baseUrl` 字段可以手动指定脚本模块的基准目录。

```
{
  "compilerOptions": {
    "baseUrl": "."
  }
}
```

上例中，`baseUrl` 是一个点，表示基准目录就是 `tsconfig.json` 所在的目录。

- (2) `paths` 字段指定非相对路径的模块与实际脚本的映射。

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"]
    }
  }
}
```

上面示例中，加载模块 `jquery` 时，实际加载的脚本是 `node_modules/jquery/dist/jquery`，它的位置要根据 `baseUrl` 字段计算得到。

上例的 `jquery` 属性的值是一个数组，可以指定多个路径。如果第一个脚本路径不存在，那么就加载第二个路径，以此类推。

- (3) `rootDirs` 字段指定模块定位时必须查找的其他目录。

```
{
  "compilerOptions": {
    "rootDirs": ["src/zh", "src/de", "src/#{locale}"]
  }
}
```