1. ts模块简介

如果一个文件包含import或export语句就是一个模块，相应地如果文件不包含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，必须使用exprot命令声明；如果一个文件要使用模块的接口，需要用import命令导入。

如果一个文件不包含export语句，但是希望把它当做一个模块，可以在脚本头部添加一行export语句，不会起到任何作用，就是将文件当做模块处理，代码变成了内部代码。

```
export {};
```

ts支持ES6模块的语法，加载模块时可以省略模块文件的后缀名。

2. import type语句

import语句可以同时输入类型和正常的接口，但是这样不利于区分，ts提供了type关键字

方法一：在输入的类型前面加上type，表示导出的是一个类型

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

方法二：使用import type，表示这个语句只能输入类型，不能输入接口

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

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

import type也可以输入默认类型

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

可以输入所有的类型

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

同样的，export也有两种方法导出类型

方法一：表示输出的是个类型

方法二：表示输出的都是类型

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

// 方法一
export {type A, type B};

// 方法二
export type {A, B};
```

3. importsNotUsedAsValues编译设置

ts 特有的输入类型（type）的 import 语句，编译成 JavaScript 时怎么处理呢？

ts 提供了`importsNotUsedAsValues`编译设置项，有三个可能的值。

（1）`remove`：这是默认值，自动删除输入类型的 import 语句。

（2）`preserve`：保留输入类型的 import 语句。

（3）`error`：保留输入类型的 import 语句（与`preserve`相同），但是必须写成`import type`的形式，否则报错。

请看示例，下面是一个输入类型的 import 语句。

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

```

上面示例中，`TypeA`是一个类型。

`remove`的编译结果会将该语句删掉。

`preserve`的编译结果会保留该语句，但会删掉其中涉及类型的部分。

```ts
import './a';

```

上面就是`preserve`的编译结果，可以看到编译后的`import`语句不从`a.js`输入任何接口（包括类型），但是会引发`a.js`的执行，因此会保留`a.js`里面的副作用。

`error`的编译结果与`preserve`相同，但在编译过程中会报错，因为它要求输入类型的`import`语句必须写成`import type`的形式。原始语句改成下面的形式，就不会报错。

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

4. CommonJS模块

1. import = 语句

ts 使用`import =`语句输入 CommonJS 模块。

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

```

上面示例中，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';
// 等同于
import fs = require('fs');
```

2. export = 语句

ts 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); // 12
```

5. 模块定位

是一种算法，用来确定import语句和export语句里面的模块文件位置。

编译参数`moduleResolution`，用来指定具体使用哪一种定位算法。常用的算法有两种：`Classic`和`Node`。

如果没有指定`moduleResolution`，它的默认值与编译参数`module`有关。`module`设为`commonjs`时（项目脚本采用 CommonJS 模块格式），`moduleResolution`的默认值为`Node`，即采用 Node.js 的模块定位算法。其他情况下（`module`设为 es2015、 esnext、amd, system, umd 等等），就采用`Classic`定位算法。

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和非相对模块两种。

相对模块指定是路径以`/`、`./`开头的模块，是根据当前脚本的位置进行计算的，一般用于保存在当前项目目录结构中的模块脚本

非相对模块指的是不带路径信息的模块，是由baseUrl属性或者模块映射确定的，通常用于加载外部模块

2. Classic方法

以当前脚本的路径作为**基准路径**来计算相对模块的位置，只在一个目录下查找。

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。

3.Node方法

就是模拟Node.js的模块加载方法，也就是require()的实现方法。

相对模块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");`，ts 按照以下顺序查找。

1. 当前目录是否包含`b.ts`、`b.tsx`、`b.d.ts`。如果不存在就执行下一步。
2. 当前目录是否存在子目录`b`，该子目录里面的`package.json`文件是否有`types`字段指定了模块入口文件。如果不存在就执行下一步。
3. 当前目录的子目录`b`是否包含`index.ts`、`index.tsx`、`index.d.ts`。如果不存在就报错。

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。

1. 当前目录的子目录`node_modules`是否包含`b.ts`、`b.tsx`、`b.d.ts`。
2. 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。
3. 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`。
4. 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`。
5. 进入上一层目录，重复上面4步，直到找到为止。

4. 路径映射

可以在tsconfig.json文件里手动指定脚本模块的路径

（1）baseUrl

`baseUrl`字段可以手动指定脚本模块的基准目录。

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

```

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

（2）paths

`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。

注意，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

`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}"]
  }
}

```

上面示例中，`rootDirs`指定了模块定位时，需要查找的不同的国际化目录。

5. 编译选项`traceResolution`

由于模块定位的过程很复杂，tsc 命令有一个`--traceResolution`参数，能够在编译时在命令行显示模块定位的每一步。

```
$ tsc --traceResolution

```

上面示例中，`traceResolution`会输出模块定位的判断过程。

6. 编译选项`noResolve`

tsc 命令的`--noResolve`参数，表示模块定位时，只考虑在命令行传入的模块。

举例来说，`app.ts`包含如下两行代码。

```
import * as A from "moduleA";
import * as B from "moduleB";

```

使用下面的命令进行编译。

```
$ tsc app.ts moduleA.ts --noResolve

```

上面命令使用`--noResolve`参数，因此可以定位到`moduleA.ts`，因为它从命令行传入了；无法定位到`moduleB`，因为它没有传入，因此会报错。