模块解析

> 这节假设你已经了解了模块的一些基本知识 请阅读 模块 (./modules.html)文档了解更多信息。

*模块解析*就是指编译器所要依据的一个流程，用它来找出某个导入操作所引用的具体值。 假设有一个导入语句 `import { a } from "moduleA"`；为了去检查任何对 `a` 的使用，编译器需要准确的知道它表示什么，并且会需要检查它的定义 `moduleA` 。

这时候，编译器会想知道“ `moduleA` 的shape是怎样的？” 这听上去很简单， `moduleA` 可能在你写的某个 `.ts` / `.tsx` 文件里或者在你的代码所依赖的 `.d.ts` 里。

首先，编译器会尝试定位表示导入模块的文件。 编译会遵循下列二种策略之一： Classic或Node。 这些策略会告诉编译器到 *哪里*去查找 `moduleA` 。

如果它们失败了并且如果模块名是非相对的（且是在 `"moduleA"` 的情况下），编译器会尝试定位一个外部模块声明 (./modules.html#ambient-modules)。 我们接下来会讲到非相对导入。

最后，如果编译器还是不能解析这个模块，它会记录一个错误。 在这种情况下，错误可能为 `error TS2307: Cannot find module 'moduleA'.`

相对 vs. 非相对模块导入

根据模块引用是相对的还是非相对的，模块导入会以不同的方式解析。

*相对导入*是以 `/` ， `./` 或 `../` 开头的。 下面是一些例子：

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

所有其它形式的导入被当作*非相对*的。 下面是一些例子：

- `import * as $ from "jQuery";`
- `import { Component } from "angular2/core";`

相对导入解析时是相对于导入它的文件来的，并且*不能*解析为一个外部模块声明。 你应该为你自己写的模块使用相对导入，这样能确保它们在运行时的相对位置。

非相对模块的导入可以相对于 `baseUrl` 或通过下文会讲到的路径映射来进行解析。 它们还可以被解析能 外部模块声明 (./modules.html#ambient-modules)。 使用非相对路径来导入你的外部依赖。

模块解析策略

共有两种可用的模块解析策略：Node和Classic。 你可以使用 `--moduleResolution` 标记指定使用哪种模块解析策略。 若未指定，那么在使用了 `--module AMD | System | ES2015` 时的默认值为Classic，其它情况时则为Node。

Classic

这种策略以前是TypeScript默认的解析策略。 现在，它存在的理由主要是为了向后兼容。

相对导入的模块是相对于导入它的文件进行解析的。 因此 `/root/src/folder/A.ts` 文件里的 `import { b } from "./moduleB"` 会使用下面的查找流程：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`

对于非相对模块的导入，编译器则会从包含导入文件的目录开始依次向上级目录遍历，尝试定位匹配的声明文件。

比如：

有一个对 `moduleB` 的非相对导入 `import { b } from "moduleB"` ，它是在 `/root/src/folder/A.ts` 文件里，会以如下的方式来定位 `"moduleB"` ：

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`
3. `/root/src/moduleB.ts`
4. `/root/src/moduleB.d.ts`
5. `/root/moduleB.ts`
6. `/root/moduleB.d.ts`
7. `/moduleB.ts`
8. `/moduleB.d.ts`

Node

这个解析策略试图在运行时模仿Node.js (https://nodejs.org/)模块解析机制。 完整的Node.js解析算法可以在 Node.js module documentation (https://nodejs.org/api/modules.html#modules_all_together)找到。

Node.js如何解析模块

为了理解TypeScript编译依照的解析步骤，先弄明白Node.js模块是非常重要的。 通常，在Node.js里导入是通过 `require` 函数调用进行的。 Node.js会根据 `require` 的是相对路径还是非相对路径做出不同的行为。

相对路径很简单。 例如，假设有一个文件路径为 `/root/src/moduleA.js` ，包含了一个导入 `var x = require("./moduleB");` Node.js以下面的顺序解析这个导入：

1. 将 `/root/src/moduleB.js` 视为文件，检查是否存在。

2. 将 `/root/src/moduleB` 视为目录，检查是否它包含 `package.json` 文件并且其指定了一个 `"main"` 模块。 在我们的例子里，如果Node.js发现文件 `/root/src/moduleB/package.json` 包含了 `{ "main": "lib/mainModule.js" }` ，那么Node.js会引用 `/root/src/moduleB/lib/mainModule.js` 。

3. 将 `/root/src/moduleB` 视为目录，检查它是否包含 `index.js` 文件。 这个文件会被隐式地当作那个文件夹下的"main"模块。

你可以阅读Node.js文档了解更多详细信息：file modules (https://nodejs.org/api/modules.html#modules_file_modules) 和 folder modules (https://nodejs.org/api/modules.html#modules_folders_as_modules)。

但是，非相对模块名的解析是个完全不同的过程。 Node会在一个特殊的文件夹 `node_modules` 里查找你的模块。 `node_modules` 可能与当前文件在同一级目录下，或者在上层目录里。 Node会向上级目录遍历，查找每个 `node_modules` 直到它找到要加载的模块。

还是用上面例子，但假设 `/root/src/moduleA.js` 里使用的是非相对路径导入 `var x = require("moduleB");` 。 Node则会以下面的顺序去解析 `moduleB` ，直到有一个匹配上。

1. `/root/src/node_modules/moduleB.js`
2. `/root/src/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
3. `/root/src/node_modules/moduleB/index.js`

4. `/root/node_modules/moduleB.js`
5. `/root/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
6. `/root/node_modules/moduleB/index.js`

7. `/node_modules/moduleB.js`
8. `/node_modules/moduleB/package.json` (如果指定了 `"main"` 属性)
9. `/node_modules/moduleB/index.js`

注意Node.js在步骤（4）和（7）会向上跳一级目录。

你可以阅读Node.js文档了解更多详细信息：loading modules from `node_modules` (https://nodejs.org/api/modules.html#modules_loading_from_node_modules_folders)。

TypeScript如何解析模块

TypeScript是模仿Node.js运行时的解析策略来在编译阶段定位模块定义文件。 因此，TypeScript在Node解析逻辑基础上增加了TypeScript源文件的扩展名（ `.ts` ， `.tsx` 和 `.d.ts` ）。 同时，TypeScript在 `package.json` 里使用字段 `"typings"` 来表示类似 `"main"` 的意义 - 编译器会使用它来找到要使用的"main"定义文件。

比如，有一个导入语句 `import { b } from "./moduleB"` 在 `/root/src/moduleA.ts` 里，会以下面的流程来定位 `"./moduleB"` ：

1. `/root/src/moduleB.ts`
2. `/root/src/moduleB.tsx`
3. `/root/src/moduleB.d.ts`
4. `/root/src/moduleB/package.json` (如果指定了 `"typings"` 属性)
5. `/root/src/moduleB/index.ts`
6. `/root/src/moduleB/index.tsx`
7. `/root/src/moduleB/index.d.ts`

回想一下Node.js先查找 `moduleB.js` 文件，然后是合适的 `package.json` ，再之后是 `index.js` 。

类似地，非相对的导入会遵循Node.js的解析逻辑，首先查找文件，然后是合适的文件夹。 因此 `/src/moduleA.ts` 文件里的 `import { b } from "moduleB"` 会以下面的查找顺序解析：

1. `/root/src/node_modules/moduleB.ts`
2. `/root/src/node_modules/moduleB.tsx`
3. `/root/src/node_modules/moduleB.d.ts`
4. `/root/src/node_modules/moduleB/package.json` (如果指定了 `"typings"` 属性)
5. `/root/src/node_modules/moduleB/index.ts`
6. `/root/src/node_modules/moduleB/index.tsx`
7. `/root/src/node_modules/moduleB/index.d.ts`

8. `/root/node_modules/moduleB.ts`
9. `/root/node_modules/moduleB.tsx`
10. `/root/node_modules/moduleB.d.ts`
11. `/root/node_modules/moduleB/package.json` (如果指定了 `"typings"` 属性)
12. `/root/node_modules/moduleB/index.ts`
13. `/root/node_modules/moduleB/index.tsx`
14. `/root/node_modules/moduleB/index.d.ts`

15. `/node_modules/moduleB.ts`
16. `/node_modules/moduleB.tsx`
17. `/node_modules/moduleB.d.ts`
18. `/node_modules/moduleB/package.json` (如果指定了 `"typings"` 属性)
19. `/node_modules/moduleB/index.ts`
20. `/node_modules/moduleB/index.tsx`
21. `/node_modules/moduleB/index.d.ts`

不要被这里步骤的数量吓到 - TypeScript只是在步骤（8）和（15）向上跳了两次目录。 这并不比Node.js里的流程复杂。

附加的模块解析标记

有时工程源码结构与输出结构不同。 通常是要经过一系统的构建步骤最后生成输出。 它们包括将 `.ts` 编译成 `.js` ，将不同位置的依赖拷贝至一个输出位置。 最终结果就是运行时的模块名与包含它们声明的源文件里的模块名不同。 或者最终输出文件里的模块路径与编译时的源文件路径不同了。

TypeScript编译器有一些额外的标记用来*通知*编译器在源码编译成最终输出的过程中都发生了哪个转换。

有一点要特别注意的是编译器*不会*进行这些转换操作； 它只是利用这些信息来指导模块的导入。

Base URL

在利用AMD模块加载器的应用里使用 `baseUrl` 是常见做法，它要求在运行时模块都被放到了一个文件夹里。 这些模块的源码可以在不同的目录下，但是构建脚本会将它们集中到一起。

设置 `baseUrl` 来告诉编译器到哪里去查找模块。 所有非相对模块导入都会被当做相对于 `baseUrl` 。

*baseUrl*的值由以下两者之一决定：

- 命令行中*baseUrl*的值（如果给定的路径是相对的，那么将相对于当前路径进行计算）
- ‘tsconfig.json’里的*baseUrl*属性（如果给定的路径是相对的，那么将相对于‘tsconfig.json’路径进行计算）

注意相对模块的导入不会被设置的 `baseUrl` 所影响，因为它们总是相对于导入它们的文件。

阅读更多关于 `baseUrl` 的信息RequireJS (http://requirejs.org/docs/api.html#config-baseUrl)和 SystemJS (https://github.com/systemjs/systemjs/blob/master/docs/overview.md#baseurl)。

路径映射

有时模块不是直接放在*baseUrl*下面。 比如，充分 `"jquery"` 模块地导入，在运行时可能被解释为 `"node_modules\jquery\dist\jquery.slim.min.js"` 。 加载器使用映射配置来将模块名映射到运行时的文件，查看RequireJs documentation (http://requirejs.org/docs/api.html#config-paths)和SystemJS documentation (https://github.com/systemjs/systemjs/blob/master/docs/overview.md#map-config)。

TypeScript编译器通过使用 `tsconfig.json` 文件里的 `"paths"` 来支持这样的声明映射。 下面是一个如何指定 `jquery` 的 `"paths"` 的例子。

```
{
  "compilerOptions": {
    "baseUrl": ".", // This must be specified if "paths" is.
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"]
    }
  }
}
```

通过 `"paths"` 我们还可以指定复杂的映射，包括指定多个回退位置。 假设在一个工程配置里，有一些模块位于一处，而其它的则在另个的位置。 构建过程会将它们集中至一处。 工程结构可能如下：

```
projectRoot
├── folder1
│   ├── file1.ts (imports 'folder1/file2' and 'folder2/file3')
│   └── file2.ts
├── generated
│   ├── folder1
│   └── folder2
│       └── file3.ts
└── tsconfig.json
```

相应的 `tsconfig.json` 文件如下：

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": [
        "*",
        "generated/*"
      ]
    }
  }
}
```

它告诉编译器所有匹配 `"*"` （所有的值）模式的模块导入会在以下两个位置查找：

1. `"*"` ： 表示名字不发生改变，所以映射为 `<moduleName> => <baseUrl>\<moduleName>`
2. `"generated\*"` 表示模块名添加了“generated”前缀，所以映射为 `<moduleName> => <baseUrl>\generated\<moduleName>`

按照这个逻辑，编译器将会如下尝试解析这两个导入：

- 导入'folder1/file2'
  1. 匹配'*'模式且通配符捕获到整个名字。
  2. 尝试列表里的第一个替换：'*' -> `folder1/file2` 。
  3. 替换结果为相对名 - 与*baseUrl*合并 -> `projectRoot/folder1/file2.ts` 。
  4. 文件存在。完成。
- 导入'folder2/file3'
  1. 匹配'*'模式且通配符捕获到整个名字。
  2. 尝试列表里的第一个替换：'*' -> `folder2/file3` 。
  3. 替换结果为相对名 - 与*baseUrl*合并 -> `projectRoot/folder2/file3.ts` 。
  4. 文件不存在，跳到第二个替换。
  5. 第二个替换：'generated/*' -> `generated/folder2/file3` 。
  6. 替换结果为相对名 - 与*baseUrl*合并 -> `projectRoot/generated/folder2/file3.ts` 。
  7. 文件存在。完成。

利用 `rootDirs` 指定虚拟目录

有时多个目录下的工程源文件在编译时会进行合并放在某个输出目录下。 这可以看做一些源目录创建了一个“虚拟”目录。

利用 `rootDirs` ，可以告诉编译器生成这个虚拟目录的*roots*； 因此编译器可以在“虚拟”目录下解析相对模块导入，就 *好像*它们被合并在了一起一样。

比如，有下面的工程结构：

```
src
└── views
    └── view1.ts (imports './template1')
    └── view2.ts

generated
└── templates
        └── views
            └── template1.ts (imports './view2')
```

`src/views` 里的文件是用于控制UI的用户代码。 `generated/templates` 是UI模版，在构建时通过模版生成器自动生成。 构建中的一步会将 `/src/views` 和 `/generated/templates/views` 的输出拷贝到同一个目录下。 在运行时，视图可以假设它的模版与它同在一个目录下，因此可以使用相对导入 `"./template"` 。

可以使用 `"rootDirs"` 来告诉编译器。 `"rootDirs"` 指定了一个*roots*列表，列表里的内容会在运行时被合并。 因此，针对这个例子， `tsconfig.json` 如下：

```
{
  "compilerOptions": {
    "rootDirs": [
      "src/views",
      "generated/templates/views"
    ]
  }
}
```

每当编译器在 `rootDirs` 的子目录下找到一个相对模块导入，它会尝试从 `rootDirs` 里导入。

跟踪模块解析

如之前讨论，编译器在解析模块时可能访问当前文件夹外的文件。 这会导致很难诊断模块为什么没有被解析，或解析到了错误的位置。 通过 `--traceResolution` 启用编译器的模块解析跟踪，它会告诉我们在模块解析过程中发生了什么。

假设我们有一个使用了 `typescript` 模块的简单应用。 `app.ts` 里有一个这样的导入 `import * as ts from "typescript"` 。

```
│   tsconfig.json
├───node_modules
│   └───typescript
│       └───lib
│               typescript.d.ts
└───src
        app.ts
```

使用 `--traceResolution` 调用编译器。

```
tsc --traceResolution
```

输出结果如下：

```
======== Resolving module 'typescript' from 'src/app.ts'. ========
Module resolution kind is not specified, using 'NodeJs'.
Loading module 'typescript' from 'node_modules' folder.
File 'src/node_modules/typescript.ts' does not exist.
File 'src/node_modules/typescript.tsx' does not exist.
File 'src/node_modules/typescript.d.ts' does not exist.
File 'src/node_modules/typescript/package.json' does not exist.
File 'node_modules/typescript.ts' does not exist.
File 'node_modules/typescript.tsx' does not exist.
File 'node_modules/typescript.d.ts' does not exist.
Found 'package.json' at 'node_modules/typescript/package.json'.
'package.json' has 'typings' field './lib/typescript.d.ts' that references 'node_modules/typescript/lib/typescript.d.ts'.
File 'node_modules/typescript/lib/typescript.d.ts' exist - use it as a module resolution result.
======== Module name 'typescript' was successfully resolved to 'node_modules/typescript/lib/typescript.d.ts'. ========
```

需要留意的地方

- 导入的名字及位置

> ======== Resolving module **'typescript'** from **'src/app.ts'**. ========

- 编译器使用的策略

> Module resolution kind is not specified, using **'NodeJs'**.

- 从npm加载typings

> 'package.json' has **'typings'** field './lib/typescript.d.ts' that references 'node_modules/typescript/lib/typescript.d.ts'.

- 最终结果

> ======== Module name 'typescript' was **successfully resolved** to 'node_modules/typescript/lib/typescript.d.ts'. ========

使用 `--noResolve`

正常来讲编译器会在开始编译之前解析模块导入。 每当它成功地解析了对一个文件 `import` ，这个文件被会加到一个文件列表里，以供编译器稍后处理。

`--noResolve` 编译选项告诉编译器不要添加任何不是在命令行上传入的文件到编译列表。 编译器仍然会尝试解析模块，但是只要没有指定这个文件，那么它就不会被包含在内。

比如

app.ts

```
import * as A from "moduleA" // OK, moduleA passed on the command-line
import * as B from "moduleB" // Error TS2307: Cannot find module 'moduleB'.
```

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

使用 `--noResolve` 编译 `app.ts` ：

- 可能正确找到 `moduleA` ，因为它在命令行上指定了。
- 找不到 `moduleB` ，因为没有在命令行上传递。

常见问题

为什么在 `exclude` 列表里的模块还会被编译器使用

`tsconfig.json` 将文件夹转变一个“工程” 如果不指定任何 `“exclude”` 或 `“files”` ，文件夹里的所有文件包括 `tsconfig.json` 和所有的子目录都会在编译列表里。 如果你想利用 `“exclude”` 排除某些文件，甚至你想指定所有要编译的文件列表，请使用 `“files”` 。

有些是被 `tsconfig.json` 自动加入的。 它不会涉及到上面讨论的模块解析。 如果编译器识别出一个文件是模块导入目标，它就会加到编译列表里，不管它是否被排除了。

因此，要从编译列表中排除一个文件，你需要在排除它的同时，还要排除所有对它进行 `import` 或使用了 `/// <reference path="..." />` 指令的文件。