45丨配置基础概念：模块解析，目标和映射

周爱民（Aimingoo）

目录

与类型系统无关的模块等“语句”

```
type name = ...
class name { ... }
enum name { ... }
interface name { ... }
function name(...): ...
```

类型系统（名字声明）

```
namespace/module name { ... }
import / export [type] ...
/// <reference path = "./lib/X.d.ts" />
declare ...
@xxx
```

注4 注1 注2 注3

模块系统或扩展

注1：TypeScript支持三种风格的包/库。一种是全局包，使用引用指令（///）装载；一种是js包（NODE/NPM），使用import 来装载；最后一种是UMD包，它可以在网页中使用<script>标签来装载。
注2：declare实际与类型系统无关，它用来支持.d.ts和环境。
注3：装饰器是ES规范的特性，并不是TS特有的。
注4：注意模块/名字空间名不属于类型系统，它与类型名是有不同语义的。注意“关键概念”：import/export，用于在不同的模块之间交换名字。

模块（module） vs. 包（package）

【内部模块 / 外部模块】

模块

```
import …
export …
module X { … }
namespace X { … }
Foo.x = …
```

包

```
/// …
<script src=…
import for Node.js/NPM package.json
  – devDependencies / dependencies
  – type / typing
  – imports / exports
  – …
```

1）包总是.js的，因为它的含义是“已交付的js代码库”
2）导入包时，会按默认规则查找对应的.d.ts
3）包没有扩展名（TypeScript中不允许指定导入包/模块的扩展名）

TypeScript支持三种风格的包/库。一种是全局包，使用引用指令（///）装载；一种是js包（NODE/NPM），使用import 来装载；最后一种是UMD包，它可以在网页中使用<script>标签来装载。

无数种模块

TS中的模块

```
export …
```

Node.js中的模块

```
module.exports =
            …
```

ECMAScript中的模块

```
import …
export …
```

浏览器中的模块

```
<script src=…
```

无数种模块

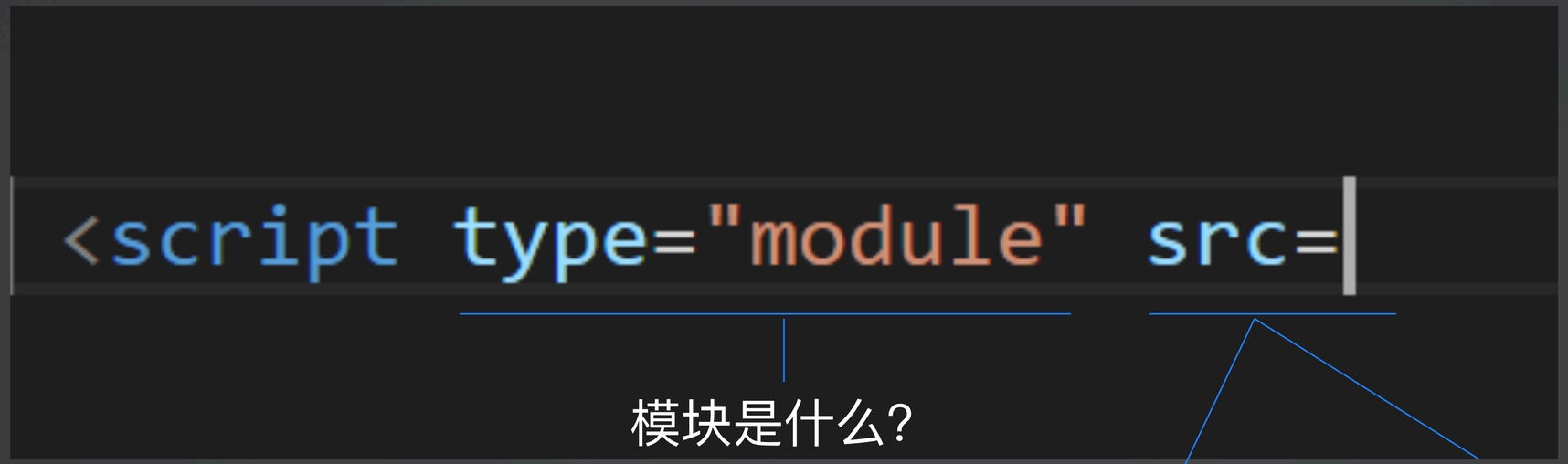
无数种模块

无数种版本

无数种加载器

……

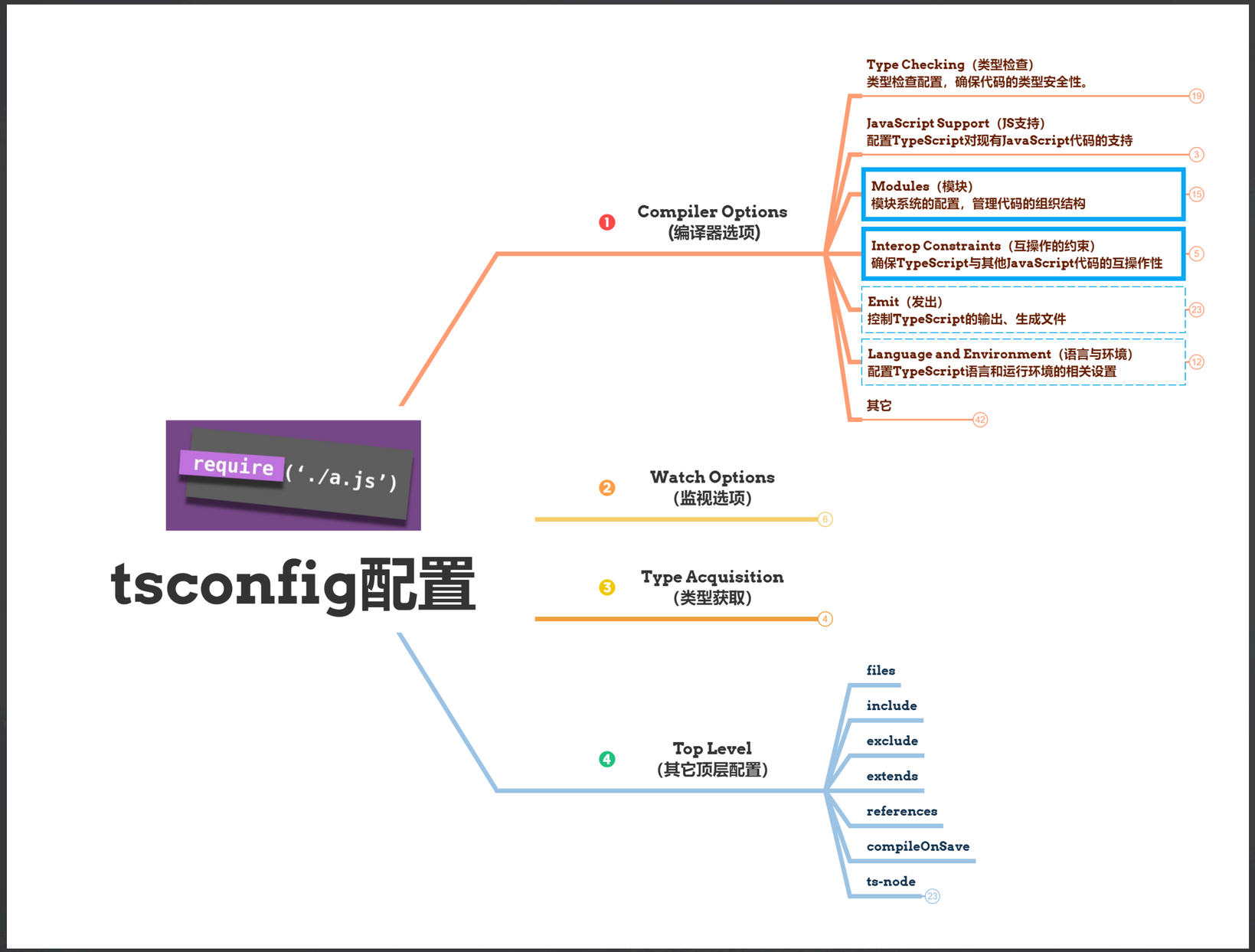
问题的核心



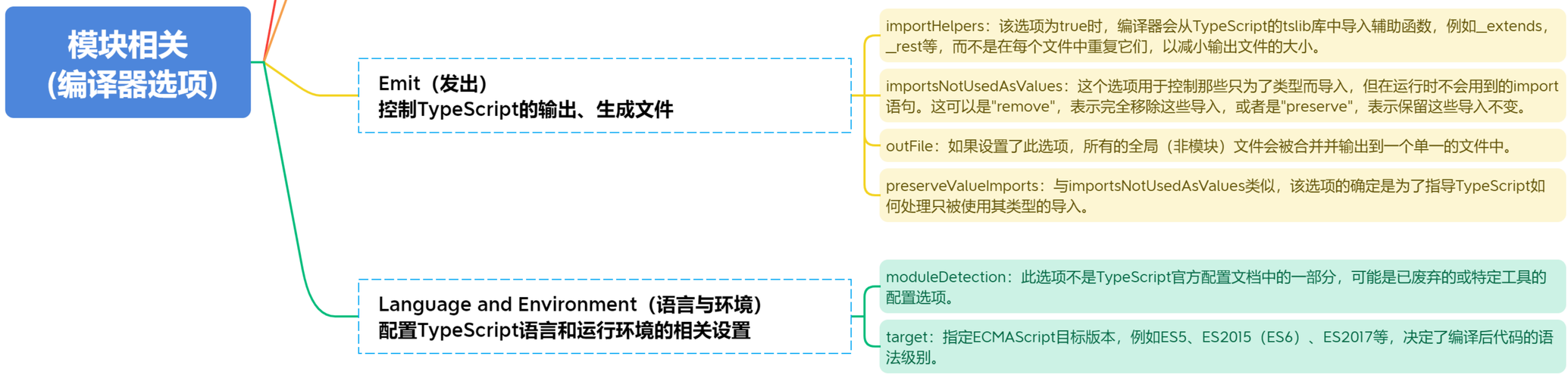
<script type="module" defer async src=...

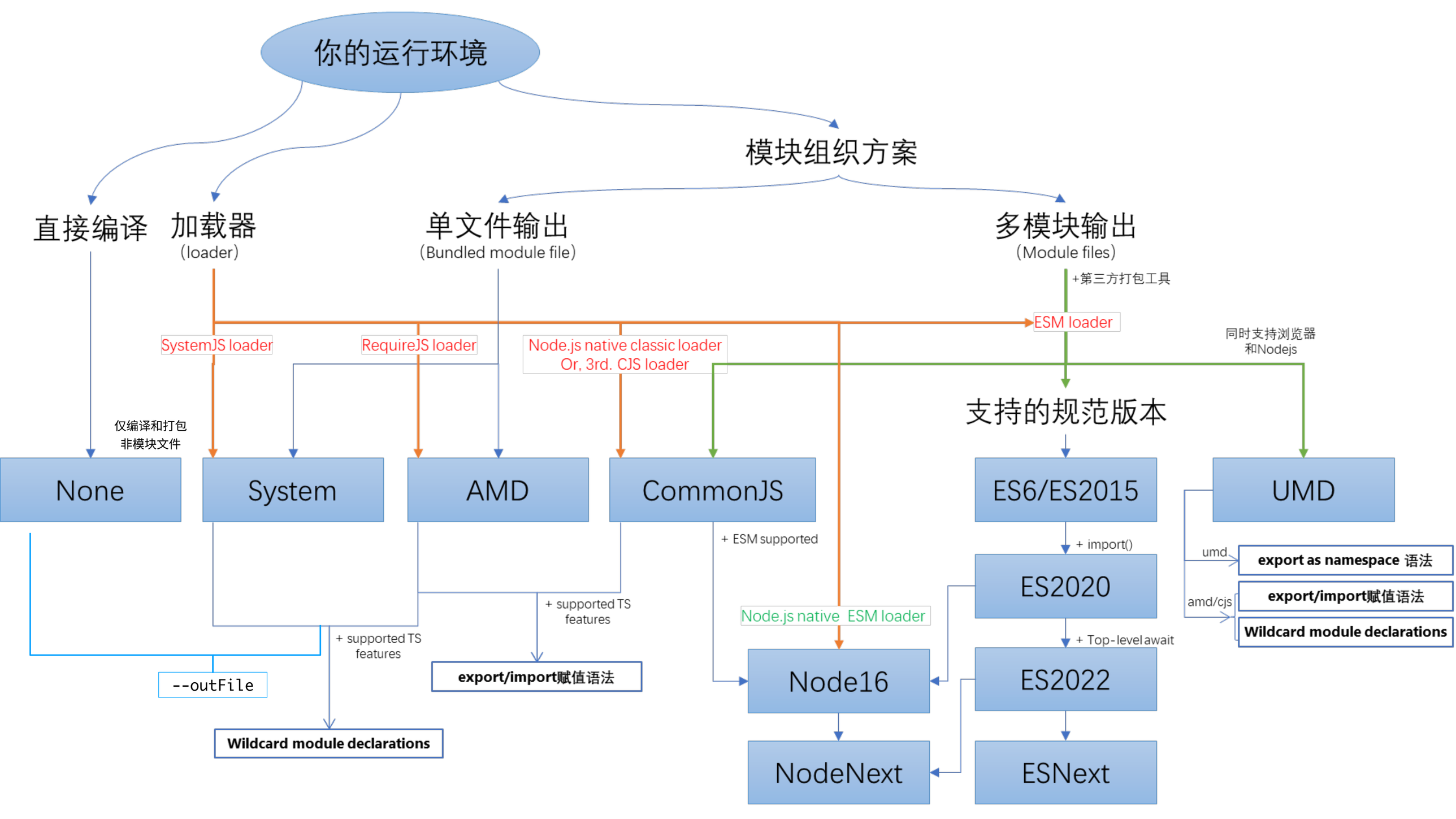
配　置

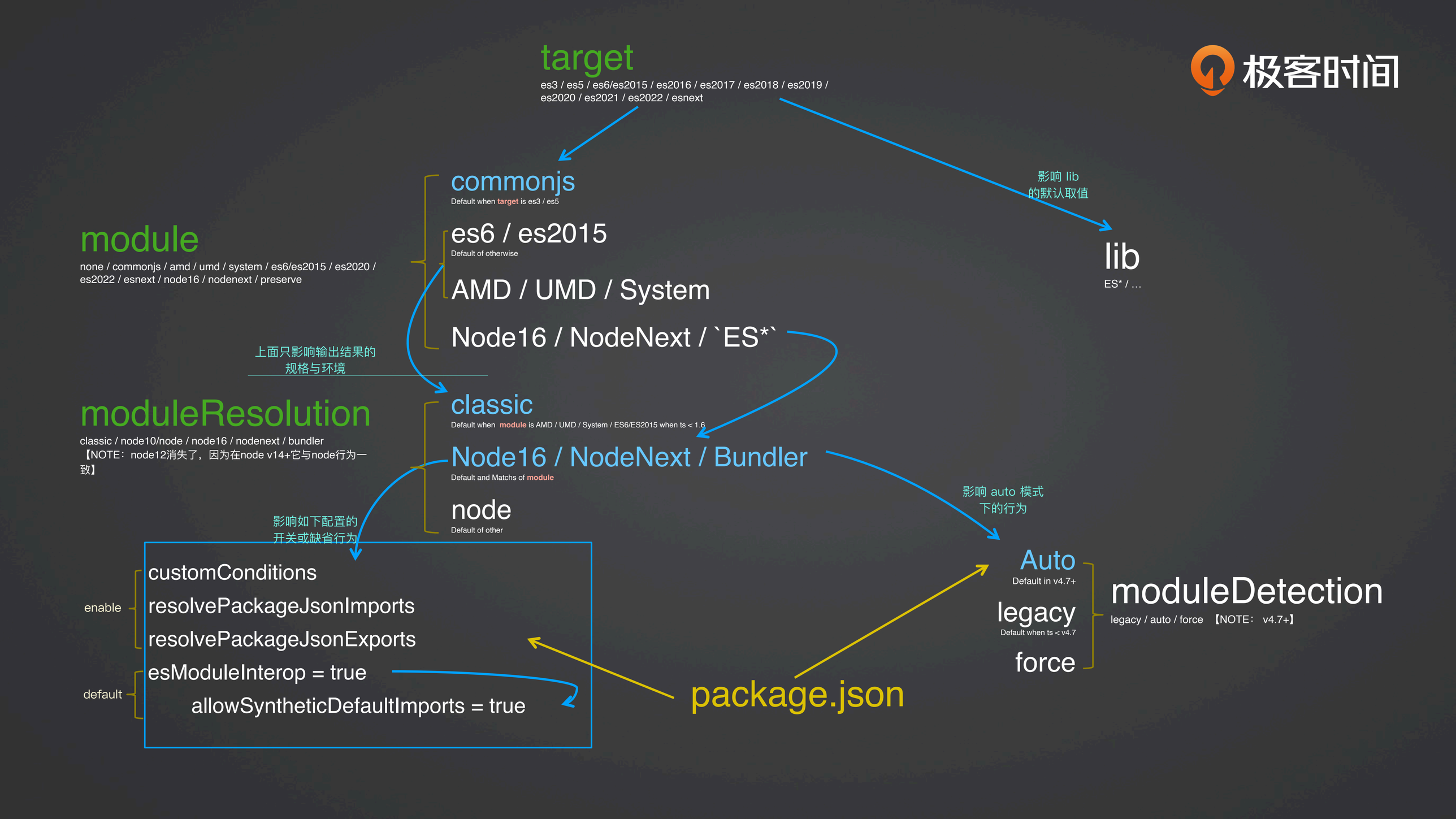
（概要）


require('./a.js')
tsconfig配置
1 Compiler Options
(编译器选项)
Type Checking（类型检查）
类型检查配置，确保代码的类型安全性。
19
JavaScript Support（JS支持）
配置TypeScript对现有JavaScript代码的支持
3
Modules（模块）
模块系统的配置，管理代码的组织结构
15
Interop Constraints（互操作的约束）
确保TypeScript与其他JavaScript代码的互操作性
5
Emit（发出）
控制TypeScript的输出、生成文件
23
Language and Environment（语言与环境）
配置TypeScript语言和运行环境的相关设置
12
其它
42
2 Watch Options
(监视选项)
6
3 Type Acquisition
(类型获取)
4
4 Top Level
(其它顶层配置)
files
include
exclude
extends
references
compileOnSave
ts-node
23

配置（模块相关）


模块相关
(编译器选项)
Modules（模块）
模块系统的配置，管理代码的组织结构
15
Interop Constraints（互操作的约束）
确保TypeScript与其他JavaScript代码的互操作性
5
Emit（发出）
控制TypeScript的输出、生成文件
importHelpers：该选项为true时，编译器会从TypeScript的tslib库中导入辅助函数，例如_extends，_rest等，而不是在每个文件中重复它们，以减小输出文件的大小。
importsNotUsedAsValues：这个选项用于控制那些只为了类型而导入，但在运行时不会用到的import语句。这可以是"remove"，表示完全移除这些导入，或者是"preserve"，表示保留这些导入不变。
outFile：如果设置了此选项，所有的全局（非模块）文件会被合并并输出到一个单一的文件中。
preserveValueImports：与importsNotUsedAsValues类似，该选项的确定是为了指导TypeScript如何处理只被使用其类型的导入。
Language and Environment（语言与环境）
配置TypeScript语言和运行环境的相关设置
moduleDetection：此选项不是TypeScript官方配置文档中的一部分，可能是已废弃的或特定工具的配置选项。
target：指定ECMAScript目标版本，例如ES5、ES2015（ES6）、ES2017等，决定了编译后代码的语法级别。


你的运行环境
直接编译
加载器
(loader)
模块组织方案
单文件输出
(Bundled module file)
多模块输出
(Module files)
+第三方打包工具
SystemJS loader
RequireJS loader
Node.js native classic loader
Or, 3rd. CJS loader
ESM loader
同时支持浏览器
和Nodejs
仅编译和打包
非模块文件
支持的规范版本
None
System
AMD
CommonJS
ES6/ES2015
UMD
+ ESM supported
+ import()
umd
export as namespace 语法
ES2020
amd/cjs
export/import赋值语法
Wildcard module declarations
+ supported TS
features
+ supported TS
features
Node.js native ESM loader
+ Top-level await
--outFile
export/import赋值语法
Node16
ES2022
Wildcard module declarations
NodeNext
ESNext


target
es3 / es5 / es6/es2015 / es2016 / es2017 / es2018 / es2019 /
es2020 / es2021 / es2022 / esnext
commonjs
Default when target is es3 / es5
es6 / es2015
Default of otherwise
AMD / UMD / System
Node16 / NodeNext / `ES*`
module
none / commonjs / amd / umd / system / es6/es2015 / es2020 /
es2022 / esnext / node16 / nodenext / preserve
影响 lib
的默认取值
lib
ES* / …
上面只影响输出结果的
规格与环境
moduleResolution
classic / node10/node / node16 / nodenext / bundler
【NOTE：node12消失了，因为在node v14+它与node行为一
致】
classic
Default when module is AMD / UMD / System / ES6/ES2015 when ts < 1.6
Node16 / NodeNext / Bundler
Default and Matchs of module
node
Default of other
影响如下配置的
开关或缺省行为
影响 auto 模式
下的行为
enable
customConditions
resolvePackageJsonImports
resolvePackageJsonExports
default
esModuleInterop = true
allowSyntheticDefaultImports = true
package.json
Auto
Default in v4.7+
legacy
Default when ts < v4.7
force
moduleDetection
legacy / auto / force 【NOTE： v4.7+】

解析路径与映射（映射）

上面只对 outDir 中的计算有意义

rootDir

影响对files中的公共父目录（路径）的计算， 表现为outDir中目录的结构变化。

rootDirs

影响import/require等语句中对“使用相对路径的模块”的位置识别。

用于查找 js 包

baseUrl

定义所有使用“非相对路径（即模块名）”存放的路径，在扫描node_modules之前查找。

paths

定义一组【模块名与查找路径】的映射表，在扫描node_modules之前查找。如果定义了baseUrl，则以baseUrl为相对位置，否则以tsconfig.json文件所在位置为相对位置。

用于查找类型库

types

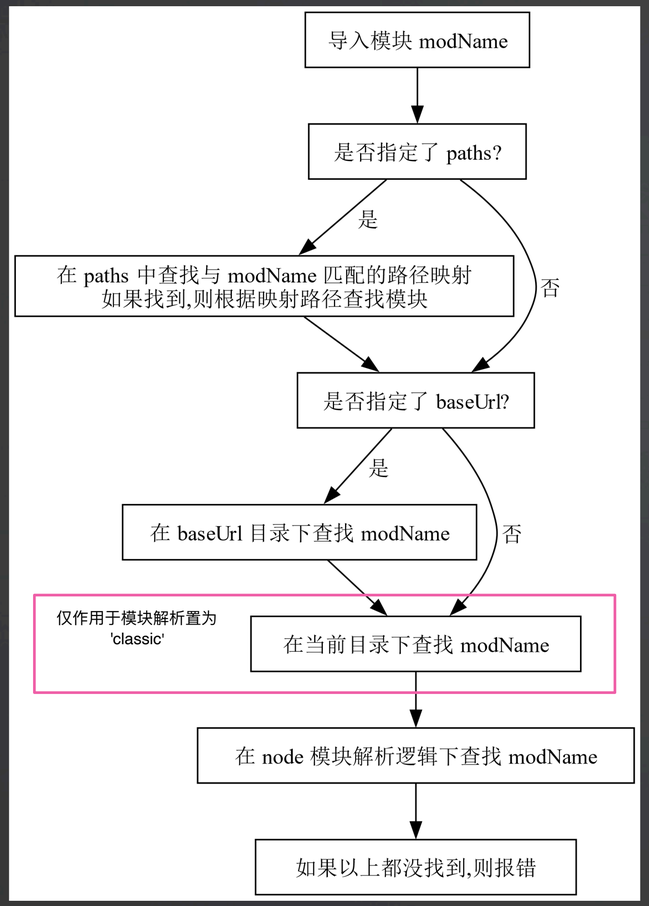
定义需要主动加载类型信息（.d.ts）的包名。

typeRoots

定义主动加载时需要扫描的类型库根目录。

“使用包名的模块”

解析路径与映射（解析）



查找 js 包：如果要加载的包名在paths列表中，那么就优先以 paths 所指示的位置列表查找该包。每个目录中的检查次序如后（见下例）。

查找类型库：如果要加载的包名在types列表中，那么就以tsconfig.json所在目录为当前工作目录起始，遍历由typesRoot决定的目录，检查该名字。以每一个目录中的检查次序如下：

① *modName*.ts、.tsx、.d.ts；
② `types` fields in *pkgName*/package.json；
③ *pkgName*/index.ts、.tsx、.d.ts

总结

1. 包是一组模块的封装和界面

2. 模块中最重要的配置项
   - 模块（module）用于指定“当前项目”编译输出的模块格式
     - 它实际上也是用于预设外部装载（当前项目）时如何理解“什么是模块”
   - 模块解析策略（moduleResolution）用于决定在哪些位置以及如何查找模块
     - @see https://sebacode.medium.com/how-to-docment-a-node-js-api-with-swagger-554101246a4d
     - 它会影响包括全局的tsc以及当前项目在内的许多行为，例如当前目录中没有安装tsc，那么classic模式不会向父目录查找文件，因此会导致tsc/deno等出现“import(”undici-types“)”时找不到模块。
   - 模块识别（moduleDetection）用于“当前项目”理解“什么是模块”

3. 使用 baseUrl / paths 映射包名与包的查找路径；使用 types / typeRoots 映射类型库XXX（XXX.d.ts）的查找路径。
   - 在types中声明类型库名（包名，例如XXX），这同时适用于使用import type / import 从包XXX导入类型。

作业

1：全面阅读 tsconfig.json 的参考手册

@see: https://www.typescriptlang.org/zh/tsconfig/

@see: https://juejin.cn/post/7293417195438178315　　// NOTE：（推荐）配置项解析，更新到ts@5.2

2：了解 TypeScript 的模块解析策略

@see: https://www.typescriptlang.org/docs/handbook/modules/theory.html#module-resolution

3：了解 Node.js 的模块装载与解析策略

@see: https://nodejs.org/api/esm.html#loaders

THANKS