tsconfig.json

简介

`tsconfig.json` 是 TypeScript 项目的配置文件，放在项目的根目录。反过来说，如果一个目录里面有 `tsconfig.json` ，TypeScript 就认为这是项目的根目录。

如果项目源码是 JavaScript，但是想用 TypeScript 处理，那么配置文件的名字是 `jsconfig.json` ，它跟 `tsconfig` 的写法是一样的。

`tsconfig.json` 文件主要供 `tsc` 编译器使用，它的命令行参数 `--project` 或 `-p` 可以指定 `tsconfig.json` 的位置（目录或文件皆可）。

bash

```
$ tsc -p ./dir
```

如果不指定配置文件的位置， `tsc` 就会在当前目录下搜索 `tsconfig.json` 文件，如果不存在，就到上一级目录搜索，直到找到为止。

`tsconfig.json` 文件的格式，是一个 JSON 对象，最简单的情况可以只放置一个空对象 `{}` 。下面是一个示例。

json

```
{
  "compilerOptions": {
    "outDir": "./built",
    "allowJs": true,
    "target": "es5"
  },
  "include": ["./src/**/*"]
}
```

本章后面会详细介绍 `tsconfig.json` 的各个属性，这里简单说一下，上面示例的四个属性的含义。

- include：指定哪些文件需要编译。
- allowJs：指定源目录的 JavaScript 文件是否原样拷贝到编译后的目录。
- outDir：指定编译产物存放的目录。
- target：指定编译产物的 JS 版本。

`tsconfig.json` 文件可以不必手写，使用 tsc 命令的 `--init` 参数自动生成。

bash

```
$ tsc --init
```

上面命令生成的 `tsconfig.json` 文件，里面会有一些默认配置。

你也可以使用别人预先写好的 tsconfig.json 文件，npm 的 `@tsconfig` 名称空间下面有很多模块，都是写好的 `tsconfig.json` 样本，比如 `@tsconfig/recommended` 和 `@tsconfig/node16` 。

这些模块需要安装，以 `@tsconfig/deno` 为例。

bash

```
$ npm install --save-dev @tsconfig/deno
# 或者
$ yarn add --dev @tsconfig/deno
```

安装以后，就可以在 `tsconfig.json` 里面引用这个模块，相当于继承它的设置，然后进行扩展。

json

```
{
  "extends": "@tsconfig/deno/tsconfig.json"
}
```

`@tsconfig` 空间下包含的完整 tsconfig 文件目录，可以查看 GitHub。

`tsconfig.json` 的一级属性并不多，只有很少几个，但是 `compilerOptions` 属性有很多二级属性。下面先逐一介绍一级属性，然后再介绍 `compilerOptions` 的二级属性，按照首字母排序。

---

exclude

`exclude` 属性是一个数组，必须与 `include` 属性一起使用，用来从编译列表中去除指定的文件。它也支持使用与 `include` 属性相同的通配符。

typescript

```
{
  "include": ["**/*"],
  "exclude": ["**/*.spec.ts"]
}
```

---

extends

`tsconfig.json` 可以继承另一个 `tsconfig.json` 文件的配置。如果一个项目有多个配置，可以把共同的配置写成 `tsconfig.base.json` ，其他的配置文件继承该文件，这样便于维护和修改。

`extends` 属性用来指定所要继承的配置文件。它可以是本地文件。

json

```
{
  "extends": "../tsconfig.base.json"
}
```

如果 `extends` 属性指定的路径不是以 `./` 或 `../` 开头，那么编译器将在 `node_modules` 目录下查找指定的配置文件。

`extends` 属性也可以继承已发布的 npm 模块里面的 tsconfig 文件。

json

```
{
  "extends": "@tsconfig/node12/tsconfig.json"
}
```

`extends` 指定的 `tsconfig.json` 会先加载，然后加载当前的 `tsconfig.json` 。如果两者有重名的属性，后者会覆盖前者。

---

files

`files` 属性指定编译的文件列表，如果其中有一个文件不存在，就会报错。

它是一个数组，排在前面的文件先编译。

```javascript
{
  "files": ["a.ts", "b.ts"]
}
```

该属性必须逐一列出文件，不支持文件匹配。如果文件较多，建议使用 `include` 和 `exclude` 属性。

---

include

`include` 属性指定所要编译的文件列表，既支持逐一列出文件，也支持通配符。文件位置相对于当前配置文件而定。

```typescript
{
  "include": ["src/**/*", "tests/**/*"]
}
```

`include` 属性支持三种通配符。

- `?`：指代单个字符
- `*`：指代任意字符，不含路径分隔符
- `**`：指定任意目录层级。

如果不指定文件后缀名，默认包括 `.ts`、`.tsx` 和 `.d.ts` 文件。如果打开了 `allowJs`，那么还包括 `.js` 和 `.jsx`。

---

references

`references` 属性是一个数组，数组成员为对象，适合一个大项目由许多小项目构成的情况，用来设置需要引用的底层项目。

```javascript
{
  "references": [
```

```
    { "path": "../pkg1" },
    { "path": "../pkg2/tsconfig.json" }
  ]
}
```

`references` 数组成员对象的 `path` 属性，既可以是含有文件 `tsconfig.json` 的目录，也可以直接是该文件。

与此同时，引用的底层项目的 `tsconfig.json` 必须启用 `composite` 属性。

javascript

```
{
  "compilerOptions": {
    "composite": true
  }
}
```

---

compileOptions

`compilerOptions` 属性用来定制编译行为。这个属性可以省略，这时编译器将使用默认设置。

allowJs

`allowJs` 允许 TypeScript 项目加载 JS 脚本。编译时，也会将 JS 文件，一起拷贝到输出目录。

json

```
{
  "compilerOptions": {
    "allowJs": true
  }
}
```

alwaysStrict

`alwaysStrict` 确保脚本以 ECMAScript 严格模式进行解析，因此脚本头部不用写 `"use strict"` 。它的值是一个布尔值，默认为 `true` 。

allowSyntheticDefaultImports

`allowSyntheticDefaultImports` 允许 `import` 命令默认加载没有 `default` 输出的模块。

比如，打开这个设置，就可以写 `import React from "react";` ，而不是 `import * as React from "react";` 。

allowUnreachableCode

`allowUnreachableCode` 设置是否允许存在不可能执行到的代码。它的值有三种可能。

- `undefined` ： 默认值，编辑器显示警告。
- `true` ： 忽略不可能执行到的代码。
- `false` ： 编译器报错。

allowUnusedLabels

`allowUnusedLabels` 设置是否允许存在没有用到的代码标签（label）。它的值有三种可能。

- `undefined` ： 默认值，编辑器显示警告。
- `true` ： 忽略没有用到的代码标签。
- `false` ： 编译器报错。

baseUrl

`baseUrl` 的值为字符串，指定 TypeScript 项目的基准目录。

由于默认是以 tsconfig.json 的位置作为基准目录，所以一般情况不需要使用该属性。

typescript

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

上面示例中， `baseUrl` 为当前目录 `./` 。那么，当遇到下面的语句，TypeScript 将以 `./` 为起点，寻找 `hello/world.ts` 。

typescript

```typescript
import { helloWorld } from "hello/world";
```

checkJs

`checkJS` 设置对 JS 文件同样进行类型检查。打开这个属性，也会自动打开 `allowJs` 。它等同于在 JS 脚本的头部添加 `// @ts-check` 命令。

json

```json
{
  "compilerOptions": {
    "checkJs": true
  }
}
```

composite

`composite` 打开某些设置，使得 TypeScript 项目可以进行增量构建，往往跟 `incremental` 属性配合使用。

declaration

`declaration` 设置编译时是否为每个脚本生成类型声明文件 `.d.ts` 。

javascript

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

declarationDir

`declarationDir` 设置生成的 `.d.ts` 文件所在的目录。

typescript

```typescript
{
  "compilerOptions": {
    "declaration": true,
    "declarationDir": "./types"
```

```
  }
}
```

declarationMap

`declarationMap` 设置生成 `.d.ts` 类型声明文件的同时，还会生成对应的 Source Map 文件。

javascript

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

emitBOM

`emitBOM` 设置是否在编译结果的文件头添加字节顺序标志 BOM，默认值是 `false` 。

emitDeclarationOnly

`emitDeclarationOnly` 设置编译后只生成 `.d.ts` 文件，不生成 `.js` 文件。

esModuleInterop

`esModuleInterop` 修复了一些 CommonJS 和 ES6 模块之间的兼容性问题。

如果 `module` 属性为 `node16` 或 `nodenext` ，则 `esModuleInterop` 默认为 `true` ，其他情况默认为 `false` 。

打开这个属性，使用 `import` 命令加载 CommonJS 模块时，TypeScript 会严格检查兼容性问题是否存在。

typescript

```
import * as moment from "moment";
moment(); // 报错
```

上面示例中，根据 ES6 规范， `import * as moment` 里面的 `moment` 是一个对象，不能当作函数调用，所以第二行报错了。

解决方法就是改写上面的语句，改成加载默认接口。

```typescript
import moment from "moment";
moment(); // 不报错
```

打开 `esModuleInterop` 以后，如果将上面的代码编译成 CommonJS 模块格式，就会加入一些辅助函数，保证编译后的代码行为正确。

注意，打开 `esModuleInterop` ，将自动打开 `allowSyntheticDefaultImports` 。

exactOptionalPropertyTypes

`exactOptionalPropertyTypes` 设置可选属性不能赋值为 `undefined` 。

```typescript
// 打开 exactOptionalPropertyTypes
interface MyObj {
  foo?: "A" | "B";
}

let obj: MyObj = { foo: "A" };

obj.foo = undefined; // 报错
```

上面示例中， `foo` 是可选属性，打开 `exactOptionalPropertyTypes` 以后，该属性就不能显式赋值为 `undefined` 。

forceConsistentCasingInFileNames

`forceConsistentCasingInFileNames` 设置文件名是否为大小写敏感，默认为 `true` 。

incremental

`incremental` 让 TypeScript 项目构建时产生文件 `tsbuildinfo` ，从而完成增量构建。

inlineSourceMap

`inlineSourceMap` 设置将 SourceMap 文件写入编译后的 JS 文件中，否则会单独生成一个 `.js.map` 文件。

inlineSources

`inlineSources` 设置将原始的 `.ts` 代码嵌入编译后的 JS 中。

它要求 `sourceMap` 或 `inlineSourceMap` 至少打开一个。

isolatedModules

`isolatedModules` 设置如果当前 TypeScript 脚本作为单个模块编译，是否会因为缺少其他脚本的类型信息而报错，主要便于非官方的编译工具（比如 Babel）正确编译单个脚本。

jsx

`jsx` 设置如何处理 `.tsx` 文件。它一般以下三个值。

- `preserve` ：保持 jsx 语法不变，输出的文件名为 jsx。
- `react` ：将 `<div />` 编译成 `React.createElement("div")` ，输出的文件名为 `.js` 。
- `react-native` ：保持 jsx 语法不变，输出的文件后缀名为 `.js` 。

javascript

```
{
  "compilerOptions": {
    "jsx": "preserve"
  }
}
```

lib

`lib` 值是一个数组，描述项目需要加载的 TypeScript 内置类型描述文件，跟三斜线指令 `/// <reference lib="" />` 作用相同。

javascript

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

TypeScript 内置的类型描述文件，主要有以下一些，完整的清单可以参考 TypeScript 源码。

- ES5

- ES2015
- ES6
- ES2016
- ES7
- ES2017
- ES2018
- ES2019
- ES2020
- ES2021
- ES2022
- ESNex
- DOM
- WebWorker
- ScriptHost

listEmittedFiles

`listEmittedFiles` 设置编译时在终端显示，生成了哪些文件。

typescript

```
{
  "compilerOptions": {
    "listEmittedFiles": true
  }
}
```

listFiles

`listFiles` 设置编译时在终端显示，参与本次编译的文件列表。

javascript

```
{
  "compilerOptions": {
    "listFiles": true
  }
}
```

mapRoot

`mapRoot` 指定 SourceMap 文件的位置，而不是默认的生成位置。

typescript

```
{
  "compilerOptions": {
    "sourceMap": true,
    "mapRoot": "https://my-website.com/debug/sourcemaps/"
  }
}
```

module

`module` 指定编译产物的模块格式。它的默认值与 `target` 属性有关，如果 `target` 是 `ES3` 或 `ES5` ，它的默认值是 `commonjs` ，否则就是 `ES6/ES2015` 。

json

```
{
  "compilerOptions": {
    "module": "commonjs"
  }
}
```

它可以取以下值：none、commonjs、amd、umd、system、es6/es2015、es2020、es2022、esnext、node16、nodenext。

moduleResolution

`moduleResolution` 确定模块路径的算法，即如何查找模块。它可以取以下四种值。

- `node` ： 采用 Node.js 的 CommonJS 模块算法。
- `node16` 或 `nodenext` ： 采用 Node.js 的 ECMAScript 模块算法，从 TypeScript 4.7 开始支持。
- `classic` ： TypeScript 1.6 之前的算法，新项目不建议使用。

它的默认值与 `module` 属性有关，如果 `module` 为 `AMD` 、 `UMD` 、 `System` 或 `ES6/ES2015` ，默认值为 `classic` ；如果 `module` 为 `node16` 或 `nodenext` ，默认值为这两个值；其他情况下,默认值为 `Node` 。

moduleSuffixes

`moduleSuffixes` 指定模块的后缀名。

typescript

```
{
  "compilerOptions": {
    "moduleSuffixes": [".ios", ".native", ""]
  }
}
```

上面的设置使得 TypeScript 对于语句 `import * as foo from "./foo";` ，会搜索以下脚本 `./foo.ios.ts` 、 `./foo.native.ts` 和 `./foo.ts` 。

newLine

`newLine` 设置换行符为 `CRLF` （Windows）还是 `LF` （Linux）。

noEmit

`noEmit` 设置是否产生编译结果。如果不生成，TypeScript 编译就纯粹作为类型检查了。

noEmitHelpers

`noEmitHelpers` 设置在编译结果文件不插入 TypeScript 辅助函数，而是通过外部引入辅助函数来解决，比如 NPM 模块 `tslib` 。

noEmitOnError

`noEmitOnError` 指定一旦编译报错，就不生成编译产物，默认为 `false` 。

noFallthroughCasesInSwitch

`noFallthroughCasesInSwitch` 设置是否对没有 `break` 语句（或者 `return` 和 `throw` 语句）的 switch 分支报错，即 `case` 代码里面必须有终结语句（比如 `break` ）。

noImplicitAny

`noImplicitAny` 设置当一个表达式没有明确的类型描述、且编译器无法推断出具体类型时，是否允许将它推断为 `any` 类型。

它是一个布尔值，默认为 `true` ，即只要推断出 `any` 类型就报错。

noImplicitReturns

`noImplicitReturns` 设置是否要求函数任何情况下都必须返回一个值，即函数必须有 `return` 语句。

noImplicitThis

`noImplicitThis` 设置如果 `this` 被推断为 `any` 类型是否报错。

noUnusedLocals

`noUnusedLocals` 设置是否允许未使用的局部变量。

noUnusedParameters

`noUnusedParameters` 设置是否允许未使用的函数参数。

outDir

`outDir` 指定编译产物的存放目录。如果不指定，编译出来的 `.js` 文件存放在对应的 `.ts` 文件的相同位置。

outFile

`outFile` 设置将所有非模块的全局文件，编译在同一个文件里面。它只有在 `module` 属性为 `None` 、 `System` 、 `AMD` 时才生效，并且不能用来打包 CommonJS 或 ES6 模块。

paths

`paths` 设置模块名和模块路径的映射，也就是 TypeScript 如何导入 `require` 或 `imports` 语句加载的模块。

`paths` 基于 `baseUrl` 进行加载，所以必须同时设置后者。

typescript

```
{
  "compilerOptions": {
    "baseUrl": "./",
    "paths": {
      "b": ["bar/b"]
    }
  }
}
```

它还可以使用通配符“*”。

typescript

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

preserveConstEnums

`preserveConstEnums` 将 `const enum` 结构保留下来，不替换成常量值。

javascript

```
{
  "compilerOptions": {
    "preserveConstEnums": true
  }
}
```

pretty

`pretty` 设置美化输出终端的编译信息，默认为 `true` 。

removeComments

`removeComments` 移除 TypeScript 脚本里面的注释，默认为 `false` 。

resolveJsonModule

`resolveJsonModule` 允许 import 命令导入 JSON 文件。

rootDir

`rootDir` 设置源码脚本所在的目录，主要跟编译后的脚本结构有关。 `rootDir` 对应目录下的所有脚本，会成为输出目录里面的顶层脚本。

rootDirs

`rootDirs` 把多个不同目录，合并成一个目虚拟目录，便于模块定位。

typescript

```
{
  "compilerOptions": {
    "rootDirs": ["bar", "foo"]
  }
}
```

上面示例中， `rootDirs` 将 `bar` 和 `foo` 组成一个虚拟目录。

sourceMap

`sourceMap` 设置编译时是否生成 SourceMap 文件。

sourceRoot

`sourceRoot` 在 SourceMap 里面设置 TypeScript 源文件的位置。

typescript

```
{
  "compilerOptions": {
    "sourceMap": true,
    "sourceRoot": "https://my-website.com/debug/source/"
  }
}
```

strict

`strict` 用来打开 TypeScript 的严格检查。它的值是一个布尔值，默认是关闭的。

typescript

```
{
  "compilerOptions": {
    "strict": true
  }
}
```

这个设置相当于同时打开以下的一系列设置。

- alwaysStrict
- strictNullChecks
- strictBindCallApply
- strictFunctionTypes
- strictPropertyInitialization
- noImplicitAny
- noImplicitThis
- useUnknownInCatchVaria

打开 `strict` 的时候，允许单独关闭其中一项。

json

```
{
  "compilerOptions": {
    "strict": true,
    "alwaysStrict": false
  }
}
```

strictBindCallApply

`strictBindCallApply` 设置是否对函数的 `call()` 、 `bind()` 、 `apply()` 这三个方法进行类型检查。

如果不打开 `strictBindCallApply` 编译选项，编译器不会对以三个方法进行类型检查，参数类型都是 `any` ，传入任何参数都不会产生编译错误。

typescript

```typescript
function fn(x: string) {
  return parseInt(x);
}

// strictBindCallApply:false
const n = fn.call(undefined, false);
// 以上不报错
```

strictFunctionTypes

`strictFunctionTypes` 允许对函数更严格的参数检查。具体来说，如果函数 B 的参数是函数 A 参数的子类型，那么函数 B 不能替代函数 A。

typescript

```typescript
function fn(x: string) {
  console.log("Hello, " + x.toLowerCase());
}

type StringOrNumberFunc = (ns: string | number) => void;

// 打开 strictFunctionTypes，下面代码会报错
let func: StringOrNumberFunc = fn;
```

上面示例中，函数 `fn()` 的参数是 `StringOrNumberFunc` 参数的子集，因此 `fn` 不能替代 `StringOrNumberFunc` 。

strictNullChecks

`strictNullChecks` 设置对 `null` 和 `undefined` 进行严格类型检查。如果打开 `strict` 属性，这一项就会自动设为 `true` ，否则为 `false` 。

bash

```bash
let value:string;

// strictNullChecks:false
// 下面语句不报错
value = null;
```

它可以理解成只要打开，就需要显式检查 `null` 或 `undefined` 。

typescript

```typescript
function doSomething(x: string | null) {
  if (x === null) {
    // do nothing
  } else {
    console.log("Hello, " + x.toUpperCase());
  }
}
```

strictPropertyInitialization

`strictPropertyInitialization` 设置类的实例属性都必须初始化，包括以下几种情况。

- 设为 `undefined` 类型
- 显式初始化
- 构造函数中赋值

注意，使用该属性的同时，必须打开 `strictNullChecks` 。

typescript

```typescript
// strictPropertyInitialization: true
class User {
  // 报错，属性 username 没有初始化
  username: string;
}

// 解决方法一
class User {
  username = "张三";
}

// 解决方法二
class User {
  username: string | undefined;
}

// 解决方法三
class User {
  username: string;

  constructor(username: string) {
    this.username = username;
```

```
  }
}
// 或者
class User {
  constructor(public username: string) {}
}

// 解决方法四：赋值断言
class User {
  username!: string;

  constructor(username: string) {
    this.initialize(username);
  }

  private initialize(username: string) {
    this.username = username;
  }
}
```

suppressExcessPropertyErrors

`suppressExcessPropertyErrors` 关闭对象字面量的多余参数的报错。

target

`target` 指定编译出来的 JavaScript 代码的 ECMAScript 版本，比如 `es2021` ，默认是 `es3` 。

它可以取以下值。

- es3
- es5
- es6/es2015
- es2016
- es2017
- es2018
- es2019
- es2020
- es2021

- es2022
- esnext

注意，如果编译的目标版本过老，比如 `"target": "es3"` ，有些语法可能无法编译， `tsc` 命令会报错。

traceResolution

`traceResolution` 设置编译时，在终端输出模块解析的具体步骤。

typescript

```
{
  "compilerOptions": {
    "traceResolution": true
  }
}
```

typeRoots

`typeRoots` 设置类型模块所在的目录，默认是 `node_modules/@types` 。

typescript

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

types

`types` 设置 `typeRoots` 目录下需要包括在编译之中的类型模块。默认情况下，该目录下的所有类型模块，都会自动包括在编译之中。

typescript

```
{
  "compilerOptions": {
    "types": ["node", "jest", "express"]
  }
}
```

useUnknownInCatchVariables

`useUnknownInCatchVariables` 设置 `catch` 语句捕获的 `try` 抛出的返回值类型，从 `any` 变成 `unknown` 。

typescript

```typescript
try {
  someExternalFunction();
} catch (err) {
  err; // 类型 any
}
```

上面示例中，默认情况下， `catch` 语句的参数 `err` 类型是 `any` ，即可以是任何值。

打开 `useUnknownInCatchVariables` 以后， `err` 的类型抛出的错误将是 `unknown` 类型。这带来的变化就是使用 `err` 之前，必须缩小它的类型，否则会报错。

typescript

```typescript
try {
  someExternalFunction();
} catch (err) {
  if (err instanceof Error) {
    console.log(err.message);
  }
}
```

参考链接

- Strict Property Initialization in TypeScript, Marius Schulz

最后更新: 2023/8/13 15:25

注释指令

tsc 命令