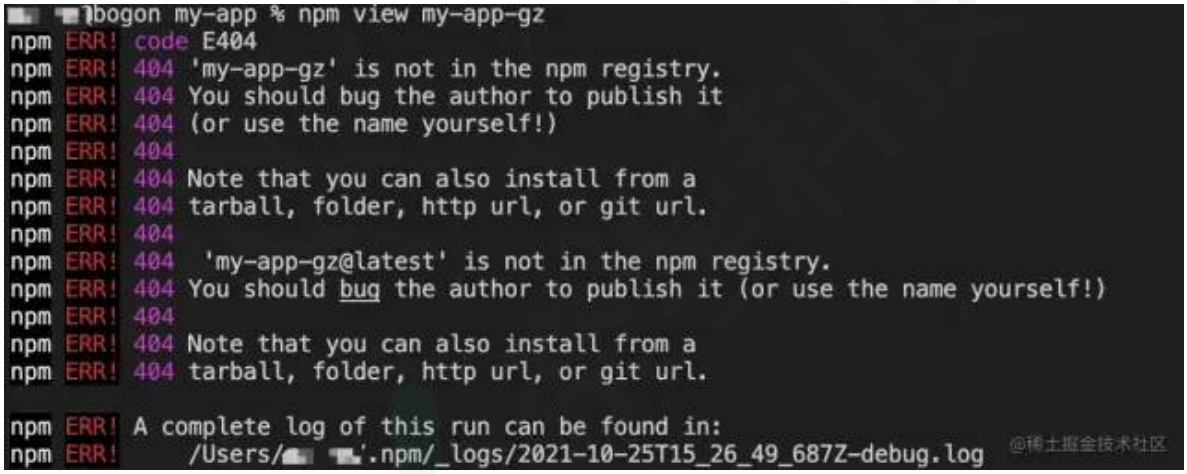
一、必须属性

package.json 中最重要的两个字段就是 name 和 version，它们都是必须的，如果没有，就无法正常执行 npm install 命令。npm 规定 package.json 文件是由名称和版本号作为唯一标识符的。

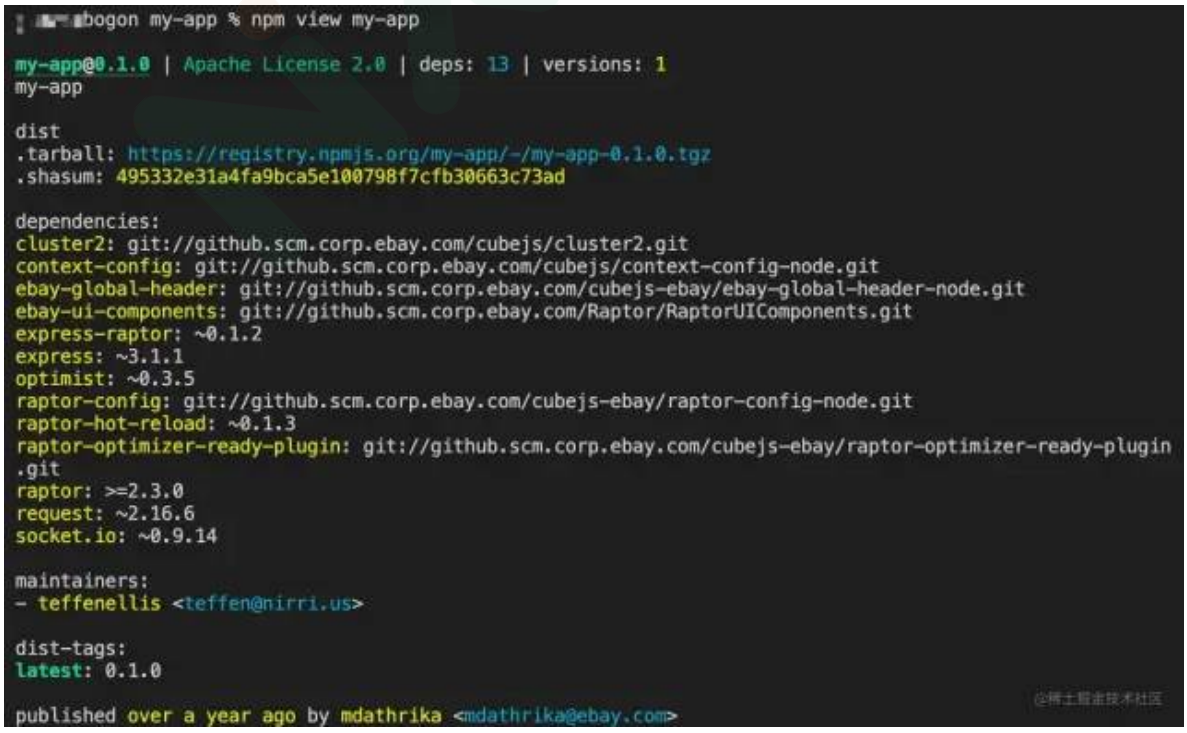
1. name

name 很容易理解，就是项目的名称，它是一个字符串。在给 name 字段命名时，需要注意以下几点：

- 名称的长度必须小于或等于 214 个字符，不能以 “.” 和“_”开头，不能包含大写字母（这是因为当软件包在 npm 上发布时，会基于此属性获得自己的 URL，所以不能包含非 URL 安全字符（non-url-safe））；
- 名称可以作为参数被传入 require("")，用来导入模块，所以应当尽可能的简短、语义化；
- 名称不能和其他模块的名称重复，可以使用 npm view 命令查询模块名是否重复，如果不重复就会提示 404：



如果 npm 包上有对应的包，则会显示包的详细信息：



实际上，我们平时开发的很多项目并不会发布在 npm 上，所以这个名称是否标准可能就不是那么重要，它不会影响项目的正常运行。如果需要发布在 npm 上，name 字段一定要符合要求。

2. version

version 字段表示该项目包的版本号，它是一个字符串。在每次项目改动后，即将发布时，都要同步的去更改项目的版本号。版本号的使用规范如下：

- 版本号的命名遵循语义化版本 2.0.0 规范，格式为：**「主版本号. 次版本号. 修订号」**，通常情况下，修改主版本号是做了大的功能性的改动，修改次版本号是新增了新功能，修改修订号就是修复了一些 bug；
- 如果某个版本的改动较大，并且不稳定，可能如法满足预期的兼容性需求，就需要发布先行版本，先行版本通过会加在版本号的后面，通过 “-” 号连接以点分隔的标识符和版本编译信息：内部版本（alpha）、公测版本（beta）和候选版本（rc，即 release candiate）。

可以通过以下命令来查看 npm 包的版本信息，以 react 为例：

```
// 查看最新版本
npm view react version
// 查看所有版本
npm view react versions
```

当执行第二条命令时，结果如下：

```
'16.8.5',
'16.8.6',
'16.9.0-alpha.0',
'16.9.0-rc.0',
'16.9.0',
'16.10.0',
'16.10.1',
'16.10.2',
'16.11.0',
'16.12.0',
'16.13.0',
'16.13.1',
'16.14.0',
'17.0.0-rc.0',
'17.0.0-rc.1',
'17.0.0-rc.2',
'17.0.0-rc.3',
'17.0.0',
'17.0.1',
'17.0.2',
'18.0.0-alpha-01be61c12',
'18.0.0-alpha-02f411578-20211019',
'18.0.0-alpha-031abd24b-20210907',
'18.0.0-alpha-05726d72c-20210927',
'18.0.0-alpha-0883c4cd3-20210929',
'18.0.0-alpha-1314299c7-20210901',
'18.0.0-alpha-19092ac8c-20210803',
'18.0.0-alpha-1a106bdc2',
'18.0.0-alpha-1a3f1afbd',
'18.0.0-alpha-1e247ff89-20211012',
'18.0.0-alpha-241485a2c-20210708',
'18.0.0-alpha-2bf2e76d3-20210826',
'18.0.0-alpha-310187264-20210716',
```


二、描述信息

package.jaon 中有五个和项目包描述信息相关的配置字段，下面就分别来看看这些字段的含义。

1. description

description 字段用来描述这个项目包，它是一个字符串，可以让其他开发者在 npm 的搜索中发现我们的项目包。

2. keywords

keywords 字段是一个字符串数组，表示这个项目包的关键词。和 description 一样，都是用来增加项目包的曝光率的。下面是 eslint 包的描述和关键词：

3. author

author 顾名思义就是作者，表示该项目包的作者。它有两种形式，一种是字符串格式：

```
"author": "CUGGZ <xxxxx@xx.com> (https://juejin.cn/user/3544481220801815)"
```

另一种是对象形式：

```
"author": {
  "name" : "CUGGZ",
  "email" : "xxxxx@xx.com",
  "url" : "https://juejin.cn/user/3544481220801815"
}
```

4. contributors

contributors 表示该项目包的贡献者，和 author 不同的是，该字段是一个数组，包含所有的贡献者，它同样有两种写法：

```
"contributors": [
    "CUGGZ0 <xxxxx@xx.com> (https://juejin.cn/user/3544481220801815)",
  "CUGGZ1 <xxxxx@xx.com> (https://juejin.cn/user/3544481220801815)"
]
"contributors": [
 {
   "name" : "CUGGZ0",
   "email" : "xxxxx@xx.com",
   "url" : "https://juejin.cn/user/3544481220801815"
 },
 {
   "name" : "CUGGZ1",
   "email" : "xxxxx@xx.com",
   "url" : "https://juejin.cn/user/3544481220801815"
 }
```

```
]
```

5. homepage

homepage 就是项目的主页地址了，它是一个字符串。

6. repository

repository 表示代码的存放仓库地址，通常有两种书写形式。第一种是字符串形式：

```
"repository": "https://github.com/facebook/react.git"
```

除此之外，还可以显式地设置版本控制系统，这时就是对象的形式：

```
"repository": {
  "type": "git",
  "url": "https://github.com/facebook/react.git"
}
```

7. bugs

bugs 表示项目提交问题的地址，该字段是一个对象，可以添加一个提交问题的地址和反馈的邮箱：

```
"bugs": {
  "url" : "https://github.com/facebook/react/issues",
  "email" : "xxxxx@xx.com"
}
```

最常见的 bugs 就是 Github 中的 issues 页面，如上就是 react 的 issues 页面地址。

三、依赖配置

通常情况下，我们的项目会依赖一个或者多个外部的依赖包，根据依赖包的不同用途，可以将他们配置在下面的五个属性下：dependencies、devDependencies、peerDependencies、bundledDependencies、optionalDependencies 。下面就来看看每个属性的含义。

1. dependencies

dependencies 字段中声明的是项目的生产环境中所必须的依赖包。当使用 npm 或 yarn 安装 npm 包时，该 npm 包会被自动插入到此配置项中：

```
npm install <PACKAGENAME>
yarn add <PACKAGENAME>
```

当在安装依赖时使用 --save 参数，也会将新安装的 npm 包写入 dependencies 属性。

```
npm install --save <PACKAGENAME>
```

该字段的值是一个对象，该对象的各个成员，分别由模块名和对应的版本要求组成，表示依赖的模块及其版本范围。

```
"dependencies": {
   "react": "^17.0.2",
   "react-dom": "^17.0.2",
   "react-scripts": "4.0.3",
},
```

这里每一项配置都是一个键值对（key-value），key 表示模块名称，value 表示模块的版本号。版本号遵循**「主版本号. 次版本号. 修订号」**的格式规定：

- **「固定版本：」** 上面的 react-scripts 的版本 4.0.3 就是固定版本，安装时只安装这个指定的版本；
- **「波浪号：」** 比如~ 4.0.3，表示安装 4.0.x 的最新版本（不低于 4.0.3），也就是说安装时不会改变主版本号和次版本号；
- **「插入号：」** 比如上面 react 的版本 ^17.0.2，表示安装 17.x.x 的最新版本（不低于 17.0.2），也就是说安装时不会改变主版本号。如果主版本号为 0，那么插入号和波浪号的行为是一致的；
- latest：安装最新的版本。

需要注意，不要把测试或者过渡性的依赖放在 dependencies，避免生产环境出现意外的问题。

2. devDependencies

devDependencies 中声明的是开发阶段需要的依赖包，如 Webpack、Eslint、Babel 等，用于辅助开发。它们不同于 dependencies，因为它们只需安装在开发设备上，而无需在生产环境中运行代码。当打包上线时并不需要这些包，所以可以把这些依赖添加到 devDependencies 中，这些依赖依然会在本地指定 npm install 时被安装和管理，但是不会被安装到生产环境中。

当使用 npm 或 yarn 安装软件包时，指定以下参数后，新安装的 npm 包会被自动插入到此列表中：

```
npm install --save-dev <PACKAGENAME>
yarn add --dev <PACKAGENAME>
"devDependencies": {
  "autoprefixer": "^7.1.2",
  "babel-core": "^6.22.1"
}
```

3. peerDependencies

有些情况下，我们的项目和所依赖的模块，都会同时依赖另一个模块，但是所依赖的版本不一样。比如，我们的项目依赖 A 模块和 B 模块的 1.0 版，而 A 模块本身又依赖 B 模块的 2.0 版。大多数情况下，这不是问题，B 模块的两个版本可以并存，同时运行。但是，有一种情况，会出现问题，就是这种依赖关系将暴露给用户。

最典型的场景就是插件，比如 A 模块是 B 模块的插件。用户安装的 B 模块是 1.0 版本，但是 A 插件只能和 2.0 版本的 B 模块一起使用。这时，用户要是将 1.0 版本的 B 的实例传给 A，就会出现问题。因此，需要一种机制，在模板安装的时候提醒用户，如果 A 和 B 一起安装，那么 B 必须是 2.0 模块。

peerDependencies 字段就是用来供插件指定其所需要的主工具的版本。

```
"name": "chai-as-promised",
"peerDependencies": {
   "chai": "1.x"
}
```

上面代码指定在安装 chai-as-promised 模块时，主程序 chai 必须一起安装，而且 chai 的版本必须是 1.x。如果项目指定的依赖是 chai 的 2.0 版本，就会报错。

需要注意，从 npm 3.0 版开始，peerDependencies 不再会默认安装了。

4. optionalDependencies

如果需要在找不到包或者安装包失败时，npm 仍然能够继续运行，则可以将该包放在 optionalDependencies 对象中，optionalDependencies 对象中的包会覆盖 dependencies 中同名的包，所以只需在一个地方进行设置即可。

需要注意，由于 optionalDependencies 中的依赖可能并未安装成功，所以一定要做异常处理，否则当获取这个依赖时，如果获取不到就会报错。

5. bundledDependencies

上面的几个依赖相关的配置项都是一个对象，而 bundledDependencies 配置项是一个数组，数组里可以指定一些模块，这些模块将在这个包发布时被一起打包。

需要注意，这个字段数组中的值必须是在 dependencies, devDependencies 两个里面声明过的包才行。

6. engines

当我们维护一些旧项目时，可能对 npm 包的版本或者 Node 版本有特殊要求，如果不满足条件就可能无法将项目跑起来。为了让项目开箱即用，可以在 engines 字段中说明具体的版本号：

```
"engines": {
 "node": ">=8.10.3 <12.13.0",
  "npm": ">=6.9.0"
}
```

需要注意，engines 只是起一个说明的作用，即使用户安装的版本不符合要求，也不影响依赖包的安装。

四、脚本配置

1. scripts

scripts 是 package.json 中内置的脚本入口，是 key-value 键值对配置，key 为可运行的命令，可以通过 npm run 来执行命令。除了运行基本的 scripts 命令，还可以结合 pre 和 post 完成前置和后续操作。先来看一组 scripts：

```
"scripts": {
  "dev": "node index.js",
  "predev": "node beforeIndex.js",
  "postdev": "node afterIndex.js"
}
```

这三个 js 文件中都有一句 console：

```
// index.js
console.log("scripts: index.js")
// beforeIndex.js
console.log("scripts: before index.js")
// afterIndex.js
console.log("scripts: after index.js")
```

当我们执行 npm run dev 命令时，输出结果如下：

```
scripts: before index.js
scripts: index.js
scripts: after index.js
```

可以看到，三个命令都执行了，执行顺序是 predev→dev→postdev。如果 scripts 命令存在一定的先后关系，则可以使用这三个配置项，分别配置执行命令。

通过配置 scripts 属性，可以定义一些常见的操作命令：

```
"scripts": {
  "dev": "webpack-dev-server --inline --progress --config
build/webpack.dev.conf.js",
  "start": "npm run dev",
  "unit": "jest --config test/unit/jest.conf.js --coverage",
  "test": "npm run unit",
  "lint": "eslint --ext .js,.vue src test/unit",
  "build": "node build/build.js"
}
```

这些脚本是命令行应用程序。可以通过调用 npm run XXX 或 yarn XXX 来运行它们，其中 XXX 是命令的名称。例如：npm run dev。我们可以为命令使用任何的名称，脚本也可以是任何操作。

使用好该字段可以大大的提升开发效率。

2. config

config 字段用来配置 scripts 运行时的配置参数，如下所示：

```
"config": {
 "port": 3000
}
```

如果运行 npm run start，则 port 字段会映射到 `npm_package_config_port` 环境变量中：

```
console.log(process.env.npm_package_config_port) // 3000
```

用户可以通过 `npm config set foo:port 3001` 命令来重写 port 的值。

五、文件 & 目录

下面来看看 package.json 中和文件以及目录相关的属性。

1. main

main 字段用来指定加载的入口文件，在 browser 和 Node 环境中都可以使用。如果我们将项目发布为 npm 包，那么当使用 require 导入 npm 包时，返回的就是 main 字段所列出的文件的 module.exports 属性。如果不指定该字段，默认是项目根目录下的 index.js。如果没找到，就会报错。

该字段的值是一个字符串：

```
"main": "./src/index.js",
```

2. browser

browser 字段可以定义 npm 包在 browser 环境下的入口文件。如果 npm 包只在 web 端使用，并且严禁在 server 端使用，使用 browser 来定义入口文件。

```
"browser": "./src/index.js"
```

3. module

module 字段可以定义 npm 包的 ESM 规范的入口文件，browser 环境和 node 环境均可使用。如果 npm 包导出的是 ESM 规范的包，使用 module 来定义入口文件。

```
"module": "./src/index.mjs",
```

需要注意，.js 文件是使用 commonJS 规范的语法 (require('xxx'))，.mjs 是用 ESM 规范的语法 (import 'xxx')。

上面三个的入口入口文件相关的配置是有差别的，特别是在不同的使用场景下。在 Web 环境中，如果使用 loader 加载 ESM（ES module），那么这三个配置的加载顺序是 browser→module→main，如果使用 require 加载 CommonJS 模块，则加载的顺序为 main→module→browser。

Webpack 在进行项目构建时，有一个 target 选项，默认为 Web，即构建 Web 应用。如果需要编译一些同构项目，如 node 项目，则只需将 webpack.config.js 的 target 选项设置为 node 进行构建即可。如果在 Node 环境中加载 CommonJS 模块，或者 ESM，则只有 main 字段有效。

4. bin

bin 字段用来指定各个内部命令对应的可执行文件的位置：

```
"bin": {
  "someTool": "./bin/someTool.js"
}
```

这里，someTool 命令对应的可执行文件为 bin 目录下的 someTool.js，someTool.js 会建立符号链接 node_modules/.bin/someTool。由于 node_modules/.bin / 目录会在运行时加入系统的 PATH 变量，因此在运行 npm 时，就可以不带路径，直接通过命令来调用这些脚本。因此，下面的写法可以简写：

```
scripts: {
  start: './node_modules/bin/someTool.js build'
}
scripts: {
    start: 'someTool build'
}
```

所有 node_modules/.bin / 目录下的命令，都可以用 npm run [命令] 的格式运行。

上面的配置在 package.json 包中提供了一个映射到本地文件名的 bin 字段，之后 npm 包将链接这个文件到 prefix/fix 里面，以便全局引入。或者链接到本地的 node_modules/.bin / 文件中，以便在本项目中使用。

5. files

files 配置是一个数组，用来描述当把 npm 包作为依赖包安装时需要说明的文件列表。当 npm 包发布时，files 指定的文件会被推送到 npm 服务器中，如果指定的是文件夹，那么该文件夹下面所有的文件都会被提交。

```
"files": [
    "LICENSE",
    "Readme.md",
    "index.js",
    "lib/"
 ]
```

如果有不想提交的文件，可以在项目根目录中新建一个. npmignore 文件，并在其中说明不需要提交的文件，防止垃圾文件推送到 npm 上。这个文件的形式和. gitignore 类似。写在这个文件中的文件即便被写在 files 属性里也会被排除在外。比如可以在该文件中这样写：

```
node_modules
.vscode
build
.DS_Store
```

6. man

man 命令是 Linux 中的帮助指令，通过该指令可以查看 Linux 中的指令帮助、配置文件帮助和编程帮助等信息。如果 node.js 模块是一个全局的命令行工具，在 package.json 通过 man 属性可以指定 man 命令查找的文档地址：

```
"man": [
 "./man/npm-access.1",
 "./man/npm-audit.1"
]
```

man 字段可以指定一个或多个文件, 当执行 man {包名} 时, 会展现给用户文档内容。

需要注意：

- man 文件必须以数字结尾，如果经过压缩，还可以使用. gz 后缀。这个数字表示文件安装到哪个 man 节中；
- 如果 man 文件名称不是以模块名称开头的，安装的时候会加上模块名称前缀。

对于上面的配置，可以使用以下命令来执行查看文档：

```
man npm-access
man npm-audit
```

7. directories

directories 字段用来规范项目的目录。node.js 模块是基于 CommonJS 模块化规范实现的，需要严格遵循 CommonJS 规范。模块目录下除了必须包含包项目描述文件 package.json 以外，还需要包含以下目录：

- bin： 存放可执行二进制文件的目录
- lib： 存放 js 代码的目录
- doc： 存放文档的目录
- test： 存放单元测试用例代码的目录

- ...

在实际的项目目录中，我们可能没有按照这个规范进行命名，那么就可以在 directories 字段指定每个目录对应的文件路径：

```
"directories": {
    "bin": "./bin",
    "lib": "./lib",
    "doc": "./doc",
    "test" "./test",
    "man": "./man"
}
```

这个属性实际上没有什么实际的作用，当然不排除未来会有什么比较有意义的用处。

六、发布配置

下面来看看和 npm 项目包发布相关的配置。

1. private

private 字段可以防止我们意外地将私有库发布到 npm 服务器。只需要将该字段设置为 true：

```
"private": true
```

2. preferGlobal

preferGlobal 字段表示当用户不把该模块安装为全局模块时，如果设置为 true 就会显示警告。它并不会真正的防止用户进行局部的安装，只是对用户进行提示，防止产生误解：

```
"preferGlobal": true
```

3. publishConfig

publishConfig 配置会在模块发布时生效，用于设置发布时一些配置项的集合。如果不想模块被默认标记为最新，或者不想发布到公共仓库，可以在这里配置 tag 或仓库地址。更详细的配置可以参考 npm-config[1]。

通常情况下，publishConfig 会配合 private 来使用，如果只想让模块发布到特定 npm 仓库，就可以这样来配置：

```
"private": true,
"publishConfig": {
  "tag": "1.1.0",
  "registry": "https://registry.npmjs.org/",
  "access": "public"
}
```

4. os

os 字段可以让我们设置该 npm 包可以在什么操作系统使用，不能再什么操作系统使用。如果我们希望开发的 npm 包只运行在 linux，为了避免出现不必要的异常，建议使用 Windows 系统的用户不要安装它，这时就可以使用 os 配置：

```
"os" ["linux"]   // 适用的操作系统
"os" ["!win32"]  // 禁用的操作系统
```

5. cpu

该配置和 OS 配置类似，用 CPU 可以更准确的限制用户的安装环境：

```
"cpu" ["x64", "AMD64"]   // 适用的cpu
"cpu" ["!arm", "!mips"]  // 禁用的cpu
```

可以看到，黑名单和白名单的区别就是，黑名单在前面加了一个 “!”。

6. license

license 字段用于指定软件的开源协议，开源协议表述了其他人获得代码后拥有的权利，可以对代码进行何种操作，何种操作又是被禁止的。常见的协议如下：

- MIT ： 只要用户在项目副本中包含了版权声明和许可声明，他们就可以拿你的代码做任何想做的事情，你也无需承担任何责任。
- Apache ： 类似于 MIT ，同时还包含了贡献者向用户提供专利授权相关的条款。
- GPL ： 修改项目代码的用户再次分发源码或二进制代码时，必须公布他的相关修改。

可以这样来声明该字段：

```
"license": "MIT"
```

七、第三方配置

package.json 文件还可以承载命令特有的配置，例如 Babel、ESLint 等。它们每个都有特有的属性，例如 eslintConfig、babel 等。它们是命令特有的，可以在相应的命令 / 项目文档中找到如何使用它们。下面来看几个常用的第三方配置项。

1. typings

typings 字段用来指定 TypeScript 的入口文件：

```
"typings": "types/index.d.ts",


复制代码
```

该字段的作用和 main 配置相同。

2. eslintConfig

eslint 的配置可以写在单独的配置文件. eslintrc.json 中，也可以写在 package.json 文件的 eslintConfig 配置项中。

```
```

```
"eslintConfig": {
      "root": true,
      "env": {
        "node": true
      },
      "extends": [
        "plugin:vue/essential",
        "eslint:recommended"
      ],
      "rules": {}
      "parserOptions": {
        "parser": "babel-eslint“
     },
}
```

3. babel

babel 用来指定 Babel 的编译配置，代码如下：

```
"babel": {
    "presets": ["@babel/preset-env"],
    "plugins": [...]
}
```

4. unpkg

使用该字段可以让 npm 上所有的文件都开启 cdn 服务，该 CND 服务由 unpkg 提供：

```
"unpkg": "dist/vue.js"
```

5. lint-staged

lint-staged 是一个在 Git 暂存文件上运行 linters 的工具，配置后每次修改一个文件即可给所有文件执行一次 lint 检查，通常配合 gitHooks 一起使用。

```
"lint-staged": {
    "*.js": [
   "eslint --fix",
   "git add"
  ]
}
```

使用 lint-staged 时，每次提交代码只会检查当前改动的文件。

6. gitHooks

gitHooks 用来定义一个钩子，在提交（commit）之前执行 ESlint 检查。在执行 lint 命令后，会自动修复暂存区的文件。修复之后的文件并不会存储在暂存区，所以需要用 git add 命令将修复后的文件重新加入暂存区。在执行 pre-commit 命令之后，如果没有错误，就会执行 git commit 命令：

```
"gitHooks": {
 "pre-commit": "lint-staged"
}
```

这里就是配合上面的 lint-staged 来进行代码的检查操作。

7. browserslist

browserslist 字段用来告知支持哪些浏览器及版本。Babel、Autoprefixer 和其他工具会用到它，以将所需的 polyfill 和 fallback 添加到目标浏览器。比如最上面的例子中的该字段值：

```
"browserslist": {
  "production": [
    ">0.2%",
    "not dead",
    "not op_mini all"
  ],
  "development": [
    "last 1 chrome version",
    "last 1 firefox version",
        "last 1 safari version"
  ]
}
```

这里指定了一个对象，里面定义了生产环境和开发环境的浏览器要求。上面的 development 就是指定开发环境中支持最后一个版本的 chrome、Firefox、safari 浏览器。这个属性是不同的前端工具之间共用目标浏览器和 node 版本的配置工具，被很多前端工具使用，比如 Babel、Autoprefixer 等。