Package.json

录播

文档

官方文档：https://docs.npmjs.com/files/package.json.html

什么是package.json

在node.js中，有模块的概念，这个模块可以是一个库、框架、项目等。这个模块的描述文件就是package.json。

它是一个json文件，描述了这个项目的重要信息。

项目的 *package.json* 是配置和描述如何与程序交互和运行的中心。 npm CLI（和 yarn）用它来识别你的项目并了解如何处理项目的依赖关系。 *package.json* 文件使 npm 可以启动你的项目、运行脚本、安装依赖项、发布到 NPM 注册表以及许多其他有用的任务。

package核心字段

创建项目目录

```
mkdir package-learn && cd package-learn
```

初始化

```
npm init -y
```

package.json生成

```
{
  "name": "package-learn",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

name

包的名称，发布到npm平台上，显示的名称，业务引入是require(name)的名称。

规范

- 唯一
- 名称必须小于或等于214个字符。这包括作用域包的作用域。
- 名称不能以点或下划线开头。
- 新包的名称中不能有大写字母。
- 名称最后成为URL、命令行上的参数和文件夹名称的一部分。因此，名称不能包含任何非url安全的字符。

version

包版本，对于业务项目来说，这个往往不太重要，但是如果你要发布自己的项目，这个就显得十分重要了。name和version共同决定了唯一一份代码。npm是用 **npm-semver** 来解析版本号的

npm模块的完整的版本号一般是**【主版本 . 次要版本 . 小版本】**，比如16.10.1

- 大版本：大的变动，可能影响了向后的兼容性
- 次要版本：增加了新的特性不改变已有特性
- 小版本：修改bug或其他小的改动

Create-react-app包的package.json

```
"devDependencies": {
    "@testing-library/jest-dom": "^4.2.0",
    "@testing-library/react": "^9.3.0",
    "@testing-library/user-event": "^7.1.2",
    "alex": "^8.0.0",
    "eslint": "^6.1.0",
    "execa": "1.0.0",
    "fs-extra": "^7.0.1",
    "get-port": "^4.2.0",
    "globby": "^9.1.0",
    "husky": "^1.3.1",
    "jest": "24.9.0",
    "lerna": "3.19.0",
    "lerna-changelog": "~0.8.2",
    "lint-staged": "^8.0.4",
    "meow": "^5.0.0",
    "multimatch": "^3.0.0",
    "prettier": "1.19.1",
    "puppeteer": "^2.0.0",
    "strip-ansi": "^5.1.0",
    "svg-term-cli": "^2.1.1",
    "tempy": "^0.2.1",
    "wait-for-localhost": "^3.1.0"
  },
```

版本匹配

- 指定版本： 比如"prettier": "1.19.1",
- 次要版本不变：比如"lerna-changelog": "~0.8.2",大版本和次要版本不变，小版本可以变，不能低于0.8.2
- 大版本不变：比如"meow": "^5.0.0"：大版本不能变，如果大版本为0，行为会像~波浪号，因为大版本为0，可能处于开发阶段，次要版本的改变可能也会带来兼容问题。

在程序中做版本匹配，推荐使用 npm-semver

description

包的描述，字符串格式，发布包之后，用户在 npmjs.com 使用搜索，在结果列表里发现你的包，和对应的描述。

```
"description": "Create React apps with no build
configuration.",
```

keywords

字段是一个字符串数组，其作用与描述相似。 NPM 注册表会为该字段建立索引，能够在有人搜索软件包时帮助找到它们。数组中的每个值都是与你的程序包关联的一个关键字

如果你不准备发布包，这个字段就没啥用了。

```
"keywords": [
    "react"
 ],
```

main

项目的主要入口，启动项目的文件

```
{
  "main": "index.js"
}

可以指定项目的主要入口，用户安装使用，require()就能返回主要入口文件的
export module.exports暴露的对象，
```

license

包的许可证，根据许可证的类型，用户知道如何使用它，有哪些限制。

```
"license": "ISC"  ISC 许可证
"license": "MIT"  MIT 许可证

##不想开源
{"license":"UNLICENSED"}
```

如果你不想提供许可证，或者明确不想授予使用私有或未发布的软件包的权限，则可以将 `UNLICENSED` 作为许可证，或者设置{"private":true}

如何选择开源许可证

参考文档:

阮一峰老师：http://www.ruanyifeng.com/blog/2011/05/how_to_choose_free_software_licenses.html

许可证帮助：https://choosealicense.com/

People字段

author

项目的作者。可以为字符串，对象

contributors

项目的贡献者

`author` 和 `contributors` 字段的功能类似。它们都是 `people` 字段，可以是 `"Name"` 格式的字符串，也可以是具有 `name, email, url` 字段的对象。email 和 url 都是可选的。

`author` 只供一个人使用， `contributors` 则可以由多个人组成。

```
"author": "kkb@example.com https://www.kaikeba.com/",
"contributors": [{
  "name": "lao han",
  "email": "example@example.com",
  "url": "https://www.kaikeba.com/"
}]
```

这些信息，可以在npm平台搜索列表上展示

依赖包管理

dependencies && devDependencies

这两个主要就是存放我们项目依赖的库的地方了， devDependencies主要是存放用于本地开发的， dependencies会在我们开发的时候带到线上。

通过 `npm i xxx -S` 会放在dependencies， `npm i xxx -D` 会放在devDependencies。所以我们在装包的时候一定要考虑这个包在线上是否用的到， 不要全都放到dependencies中，增加我们打包的体积和效率。

dependencies 应用依赖，业务依赖

```
{
  "dependencies" :{
    "foo" : "1.0.0 - 2.9999.9999", // 指定版本范围
    "bar" : ">=1.0.2 <2.1.2",
    "baz" : ">1.0.2 <=2.3.4",
    "boo" : "2.0.1", // 指定版本
    "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0",
    "asd" : "http://asdf.com/asdf.tar.gz", // 指定包地址
    "til" : "~1.2",  // 最近可用版本
    "elf" : "~1.2.3",
    "elf" : "^1.2.3", // 兼容版本
    "two" : "2.x", // 2.1、2.2、...、2.9皆可用
    "thr" : "*",  // 任意版本
    "thr2": "", // 任意版本
    "lat" : "latest", // 当前最新
    "dyl" : "file:../dyl", // 本地地址
    "xyz" : "git+ssh://git@github.com:npm/npm.git#v1.0.27", //
git 地址
    "fir" : "git+ssh://git@github.com:npm/npm#semver:^5.0",
    "wdy" : "git+https://isaacs@github.com/npm/npm.git",
    "xxy" : "git://github.com/npm/npm.git#v1.0.27",
  }
```

devDependencies 开发环境依赖,通常是单元测试或者打包工具等

peerDependencies

同等依赖，同伴依赖，指定当前包（也就是你写的包）兼容的宿主版本。如何理解呢？ 试想一下，我们编写一个webpack的插件，而webpack却有多个主版本，我们只想兼容最新的版本，此时就可以用同等依赖（**peerDependencies**）来指定：

```
{
  "name": "webpack-my-plugin",
  "version": "0.0.1",
  "peerDependencies": {
    "webpack": "4.x"
  }
```

当别人使用我们的插件时，**peerDependencies**就会告诉明确告诉使用方，你需要安装该插件哪个宿主版本。

browserslist

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

private

如果设为true，无法通过`npm publish`发布代码。

```
{
  "private":"true"
}
```

engines

指定项目所依赖的node环境、npm版本等

```
"engines": {
    "node": ">=8",
    "npm": ">= 4.0.0"
 },
```

files

数组。表示代码包下载安装完成时包括的所有文件

作用和**gitignore**类似，只不过是反过来的，如果要包含所有文件可以使用[*]表示。

```
"files": [
    "index.js",
    "createReactApp.js",
    "yarn.lock.cached"
 ]
```

repository

对于组件库很有用。让组件库使用者找到你的代码库地址。这个配置项会直接在组件库的npm首页生效

```
"repository": {
    "type": "git",
    "url": "git+https://github.com/xxx.git"
},
```

难点

scripts

是 npm CLI 用来运行项目任务的强大工具。他们可以完成开发过程中的大多数任务,指定运行脚本命令的npm命令行缩写。

```
"scripts": {
    "build": "cd packages/react-scripts && node bin/react-scripts.js build",
    "changelog": "lerna-changelog",
    "create-react-app": "node tasks/cra.js",
    "e2e": "tasks/e2e-simple.sh",
    "e2e:docker": "tasks/local-test.sh",
    "postinstall": "cd packages/react-error-overlay/ && yarn build:prod",
    "publish": "tasks/publish.sh",
    "start": "cd packages/react-scripts && node bin/react-scripts.js start",
    "test": "cd packages/react-scripts && node bin/react-scripts.js test",
    "format": "prettier --trailing-comma es5 --single-quote --write 'packages/*/*.js' 'packages/*/!(node_modules)/**/*.js'",
    "dev": "rimraf \"config/.conf.json\" && rimraf \"src/next.config.js\",
```

```
    "clean": "rimraf ./dist && mkdir dist",
    "prebuild": "npm run clean",
    "build:test": "cross-env NODE_ENV=production webpack"
  }
```

npm 脚本的原理非常简单。每当执行npm run，就会自动新建一个 Shell，在这个 Shell 里面执行指定的脚本命令。因此，只要是 Shell（一般是 Bash）可以运行的命令，就可以写在 npm 脚本里面。

自定义脚本

我们最常用的npm start，npm run dev ……，这些脚本都是可以用户自定义的，只用在scripts中写相应的shell脚本，可以快速的帮助我们编写打包，启动脚本

```
"scripts": {
  "build": "webpack --config build.js",
  "start": "node index.js",
  "test": "tap test/*.js"
}
```

然后我们就可以使用npm run ${name}来执行对应脚本

比较特别的是，npm run新建的这个 Shell，会将当前目录的node_modules/.bin子目录加入PATH变量，执行结束后，再将PATH变量恢复原样。

这意味着，当前目录的node_modules/.bin子目录里面的所有脚本，都可以直接用脚本名调用，而不必加上路径。

例如执行tap命令，你可以直接写

```
"scripts": {"test": "tap test/\*.js"}
```

而不是

```
"scripts": {"test": "node_modules/.bin/tap test/\*.js"}
```

npm run

执行npm run 可以列出所有可以执行的脚本命令

cross-env

运行跨平台设置和使用环境变量的脚本，

原因：当您使用NODE_ENV =production, 来设置环境变量时，大多数Windows命令提示将会阻塞(报错)。 （异常是Windows上的Bash，它使用本机Bash。）同样，Windows和POSIX命令如何使用环境变量也有区别。使用POSIX，您可以使用：$ ENV_VAR和使用%ENV_VAR%的Windows。 **说人话：windows不支持NODE_ENV=development的设置方式。**

cross-env能够提供一个设置环境变量的scripts，让你能够以unix方式设置环境变量，然后在windows上也能兼容运行。

```
#安装
npm install --save-dev cross-env

#使用
{
  "scripts": {
    "build": "cross-env NODE_ENV=production webpack --config
build/webpack.config.js"
  }
}
```

*通配符

`*` 表示任意文件名， `**` 表示任意一层子目录。

```
"lint": "jshint *.js"
"lint": "jshint **/*.js"
```

如果要将通配符传入原始命令，防止被 Shell 转义，要将星号转义。

```
"test": "tap test/\*.js"
```

脚本传参符号： --

```
"server": "webpack-dev-server --mode=development --open --
iframe=true "
```

脚本执行顺序

并行执行（即同时的平行执行），可以使用 `&` 符号

```
$ npm run script1.js & npm run script2.js
```

继发执行（即只有前一个任务成功，才执行下一个任务），可以使用 `&&` 符号

```
$ npm run script1.js && npm run script2.js
```

脚本钩子

npm 脚本有pre和post两个钩子，前者是在脚本运行前，后者是在脚本运行后执行, 所有的命令脚本都可以使用钩子（包括自定义的脚本）。

例如：运行npm run build，会按以下顺序执行：

npm run prebuild --> npm run build --> npm run postbuild

```
"clean": "rimraf ./dist && mkdir dist",
"prebuild": "npm run clean",
"build": "cross-env NODE_ENV=production webpack"
"clean": "rimraf ./dist && mkdir dist","prebuild": "npm run
clean","build": "cross-env NODE_ENV=production webpack"
```

npm 默认提供下面这些钩子:

```
prepublish, postpublish
preinstall, postinstall
preuninstall, postuninstall
preversion, postversion
pretest, posttest
prestop, poststop
prestart, poststart
prerestart, postrestart
```

常用脚本命令

```
// 删除目录
"clean": "rimraf dist/*",
// 本地搭建一个 HTTP 服务
"serve": "http-server -p 9090 dist/",
// 打开浏览器
"open:dev": "opener http://localhost:9090",
// 实时刷新
"livereload": "live-reload --port 9091 dist/",
// 构建 HTML 文件
"build:html": "jade index.jade > dist/index.html",
// 只要 CSS 文件有变动，就重新执行构建
"watch:css": "watch 'npm run build:css' assets/styles/",
// 只要 HTML 文件有变动，就重新执行构建
"watch:html": "watch 'npm run build:html' assets/html",
// 构建 favicon
```

```
"build:favicon": "node scripts/favicon.js",
```

拿到package.json的变量

npm 脚本有一个非常强大的功能，就是可以使用 npm 的内部变量。

首先，通过npm_package_前缀，npm 脚本可以拿到package.json里面的字段。比如，下面是一个package.json。

```
// package.json
{
  "name": "foo",
  "version": "1.2.5",
  "scripts": {
    "view": "node view.js"
  }
}
```

我们可以在自己的js中这样：

```
console.log(process.env.npm_package_name); // foo
console.log(process.env.npm_package_version); // 1.2.5
```

bin

用来指定各个内部命令对应的可执行文件的路径

它是一个命令名和本地文件名的映射。在安装时，如果是全局安装，npm将会使用符号链接把这些文件链接到prefix/bin，如果是本地安装，会链接到./node_modules/.bin/。

通俗点理解就是我们全局安装， 我们就可以在命令行中执行这个文件， 本地安装我们可以在当前工程目录的命令行中执行该文件。

```
"bin": {
    "create-react-app": "./index.js"
}


##index.js
#!/usr/bin/env node
...
```

要注意： 这个index.js文件的头部必须有这个 `#!/usr/bin/env node` 节点， 否则脚本将在没有节点可执行文件的情况下启动。

Demo

通过npm init -y创建一个package.json文件。

```
{
    "name": "test-bin",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "bin": {
        "kkb": "./index.js"
    },
    "scripts": {},
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {}
}
```

在package.json的同级目录新建index.js文件

```
#!/usr/bin/env node
console.log('开课吧')
```

然后在项目目录下执行： mac下： `sudo npm i -g` , window下： `npm i -g`

接下来你在任意目录新开一个命令行， 输入 `kkb` ,你会看到 `开课吧` 字段

config

用于添加命令行的环境变量

git commit

规范

git

管理你的 package.json

npm CLI 和 lerna

lerna 用于管理多 package，且各 package 可能会互相引用的项目。 lerna 通过两种方式管理子项目的版本号

Fixed/Locked mode (default)：每次执行 lerna publish 都会将所涉及到的包升级到最新一个版本，开发者只需要确定发布下一个 version。
Independent mode：由开发者自行管理子项目的 version，每次执行 lerna publish 都需要确定每个包的下个版本号

lerna init

初始化一个 lerna 工程

- packages(目录)
- lerna.json(配置文件)
- package.json(工程描述文件)

lerna bootstrap

安装各 packages 依赖

lerna publish

发布 packges

lerna 有两种工作模式,Independent mode 和 Fixed/Locked mode