Package.json

在学习webpack工程化章节的时候，有不少小伙伴们对package.json描述字段感兴趣，比如对script字段的用法，如何定义脚本命令，版本依赖的符号是什么含义。

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。
- 它是一个标准的json对象，描述了这个项目的配置信息(名称，版本，许可证等元数据)以及所需要的各种模块。
- Npm install命令会根据这个配置文件，自动下载依赖的模块，配置运行和开发的环境。

创建案例代码

- 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"
}
```

package 核心字段（元数据）

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。
- 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。

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React 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

preferGlobal

preferGlobal的值是布尔值，表示当用户不将该模块安装为全局模块时（即不用–global参数），要不要显示警告，表示该模块的本意就是安装为全局模块。

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"
 ],
```

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

homepage

主页信息，一个表示项目首页的url

bug反馈信息

一个表示接收问题反馈的url地址，也可以是email地址。一般是Github项目下的issues

```
"bugs":
{"url":"http://path/to/bug","email":"bug@example.com"},
```

repository

指明你的项目源代码仓库所在位置，这有助于其他人为你的项目贡献(contribute)代码，如果你的git项目是托管在GitHub上的，那么 `npm docs` 命令是能够找到的。

对于组件库很有用，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"
},
```

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"
 ]
```

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"
 },
```

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/

依赖包管理

`npm 目前支持5种依赖包管理类型：`

dependencies：应用依赖/业务依赖

devDependencies：开发依赖

---

peerDependencies：同伴依赖

bundledDependencies / bundleDependencies：打包依赖

`optionalDependencies`：可选依赖

我们平时常用的依赖是dependencies，devdependencies，剩下三种依赖有发布包到平台的需求才会使用到。

dependencies：业务依赖

- 列出了项目使用的所有依赖项
- 使用 npm CLI 安装软件包时，它将下载到你的 *node_modules/* 文件夹中，并将一个条目添加到你的依赖项属性中
- 用于指定应用依赖的外部包，一些依赖是应用发布上线后，正常执行时所需要的。意思就是这些依赖项应该是线上代码的一部分。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。

通过以下指令安装依赖会放在dependencies

```
npm i packageName --save
npm i packageName -S
npm install packageName --save
npm install packageName -S
npm i packageName  //npm 5.x版本，不需要-S/--save指令就会把依赖添加到dependencies中去。
```

> 如果没有指定版本，直接写一个包的名字，则安装当前npm仓库中这个包的最新版本。如果要指定版本的，可以把版本号写在包名后面，比如 `npm i react@16.12.0 -S` 。

在依赖版本中看到的插入符号（`^`）和波浪号（`~`）是 SemVer 中定义的版本范围的表示法

```
{
  "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",
  }
```

主要版本匹配

- **指定版本**：比如 `1.2.2`，遵循“大版本.次要版本.小版本”的格式规定，安装时只安装指定版本。
- **波浪号（tilde）+指定版本**：比如 `~1.2.2`，表示安装1.2.x的最新版本（不低于1.2.2），但是不安装1.3.x，也就是说安装时不改变大版本号和次要版本号。
- **插入号（caret）+指定版本**：比如^1.2.2，表示安装1.x.x的最新版本（不低于1.2.2），但是不安装2.x.x，也就是说安装时不改变大版本号。需要注意的是，如果大版本号为0，则插入号的行为与波浪号相同，这是因为此时处于开发阶段，即使是次要版本号变动，也可能带来程序的不兼容。
- **latest**：安装最新版本

devDependencies:开发依赖

- 项目开发时需要的依赖，不应该是线上代码的一部分。通常是单元测试或者打包工具等

通过以下指令安装依赖会放在devDependencies

```
npm i packageName --save-dev
npm i packageName -D
npm install packageName --save-dev
npm install packageName -D
```

> dependencies和devDependencies本质上没有什么区别，只是单纯的一个规范作用。执行npm install时两个依赖都会下载到本地。

peerDependencies:同等/同伴依赖

这种依赖的作用是提示宿主环境去安装插件在 `peerDependencies` 中所指定依赖的包，最终解决插件与所依赖包不一致的问题。

举个例子来给大家说明下。 `element-ui@2.6.3` 只是提供一套基于 `vue` 的 `ui` 组件库，但它要求宿主环境需要安装指定的 `vue` 版本，所以你可以看到 `element` 项目中的 `package.json` 中具有一项配置：

```
"peerDependencies": {
    "vue": "^2.5.16"
}
```

它要求宿主环境安装 `3.0.0 > vue@ >= 2.5.16` 的版本，也就是 `element-ui` 的运行依赖宿主环境提供的该版本范围的 `vue` 依赖包。

> `npm 3.x` 只会在安装结束后检查本次安装是否正确，如果不正确会给用户打印警告提示，比如提示用户有的包必须安装或者有的包版本不对等。
>
> 大白话：如果你安装我，那么你最好也要按照我的要求安装A、B和C。

bundledDependencies / bundleDependencies 打包依赖

这种依赖跟 `npm pack` 打包命令有关。假设 `package.json` 中有如下配置：

```
{
  "name": "font-end",
  "version": "1.0.0",
  "dependencies": {
    "fe1": "^0.3.2",
    ...
  },
  "devDependencies": {
    ...
    "fe2": "^1.0.0"
  },
  "bundledDependencies": [
    "fe1",
    "fe2"
  ]
}
```

执行打包命令`npm pack`，会生成`front-end-1.0.0.tgz`压缩包，并且该压缩包中包含`fe1`和`fe2`两个安装包，这样使用者执行`npm install front-end-1.0.0.tgz`也会安装这两个依赖。

> 在`bundledDependencies`中指定的依赖包，必须先在`dependencies`和`devDependencies`声明过，否则打包会报错。

optionalDependencies:可选依赖

这种依赖中的依赖项即使安装失败了，也不影响整个安装的过程。需要注意的是，如果一个依赖同时出现在`dependencies`和`optionalDependencies`中，那么`optionalDependencies`会获得更高的优先级，可能造成一些预期之外的效果，所以尽量要避免这种情况发生。

Package-lock.json

现在执行`npm install`的时候，就会在当前目录生成一个`package-lock.json`的文件。

用以记录当前状态下实际安装的各个npm package的具体来源和版本号。

用于依赖包版本管理

- 在大版本相同的前提下，如果一个模块在 `package.json` 中的小版本要**大于** `package-lock.json` 中的小版本，则在执行 `npm install` 时，会将该模块更新到大版本下的最新的版本，并将版本号更新至 `package-lock.json` 。如果**小于**，则被 `package-lock.json` 中的版本锁定。
- 如果一个模块在 `package.json` 和 `package-lock.json` 中的大版本不相同，则在执行 `npm install` 时，都将根据 `package.json` 中大版本下的最新版本进行更新，并将版本号更新至 `package-lock.json` 。
- 如果一个模块在 `package.json` 中有记录，而在 `package-lock.json` 中无记录，执行 `npm install` 后，则会在 `package-lock.json` 生成该模块的详细记录。同理，一个模块在 `package.json` 中无记录，而在 `package-lock.json` 中有记录，执行 `npm install` 后，则会在 `package-lock.json` 删除该模块的详细记录。

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 index.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

package.json 中的 bin 字段

`package.json` 中的字段 `bin` 表示的是一个**可执行文件到指定文件源的映射**。通过 `npm bin` 指令显示当前项目的 `bin` 目录的路径。例如在 `@vue/cli` 的 `package.json` 中：

```
"bin": {
  "vue": "bin/vue.js"
}
复制代码
```

如果全局安装 `@vue/cli` 的话，**`@vue/cli` 源文件会被安装在全局源文件安装目录（`/user/local/lib/node_modules`）下，而 `npm` 会在全局可执行 `bin` 文件安装目录（`/usr/local/bin`）下创建一个指**

向 `/usr/local/lib/node_modules/@vue/cli/bin/vue.js` 文件的名为 `vue` 的软链接，这样就可以直接在终端输入 `vue` 来执行相关命令。如下图所示：

如果局部安装 `@vue/cli` 的话， `npm` 则会在本地项目 `./node_modules/.bin` 目录下创建一个指向 `./node_moudles/@vue/cli/bin/vue.js` 名为 `vue` 的软链接，这个时候需要在终端中输入 `./node_modules/.bin/vue` 来执行（也可以使用 `npx vue` 命令来执行， `npx` 的作用就是为了方便调用项目内部安装的模块）。

> 软链接（符号链接）是一类特殊的可执行文件， 其包含有一条以绝对路径或者相对路径的形式指向其它文件或者目录的引用。在 `bin` 目录下执行 `ll` 指令可以查看具体的软链接指向。在对链接文件进行读或写操作的时候，系统会自动把该操作转换为对源文件的操作，但删除链接文件时，系统仅仅删除链接文件，而不删除源文件本身。

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