前言

前端在入门的时候，第一个了解的东西，就是`package.json`文件，这个文件伴随着所有的前端开发，生生不息。

那么问题来了，我们是否有仔细的深入了解过这个文件，看透外表，深入里层呢?

1. 字段详解

1.1 name

定义: `name` 在一个非分发(上传至npm库)项目中，仅仅作为项目名称使用。而如果作为一个模块、插件、类库分发的项目， `name` 不仅是一个项目名称，还定义了你在包管理网站上的安装名称。

```
{
  "name":"projectname"
}
```

另外, `name` 命名是有规则的，规则如下 `^(?:@[a-z0-9-*~][a-z0-9-*._~]*/)?[a-z0-9-~][a-z0-9-._~]*$` ，这个规则考点在于仅允许一次 `/` 、 `@` ,并且全部为小写。

1.2 version

定义: `version` 定义了当前项目的版本迭代进度。

现在很多项目基本不是太在乎版本号的管理更多的是采用产品版本号，还有些项目采用了产品版本号+git hash code的方式。

一般定义版本号为 `[x,y,z]` ,x、y、z的含义分别为[主版本,次版本,补丁版本]。版本通常为常规数字，每次更新递进 `1` 。

主版本(x):你的项目大变动，改的妈都不认识了，以前用你的项目基本BOOM。

次版本(y):调整了很多东西，增加了很多功能，但是以前用你的项目表示基本稳定。

补丁(z):修复自己的补丁，增加一些细节，完全不影响用你的项目。

更多的可以查看语义化版本号，这是一个大家默认推荐的版本号说明文档。

1.3 scripts

定义: `scripts` 指定了运行脚本命令的 `npm` 命令行缩写，比如 `serve` 指定了运行 `npm run serve` 时，所要执行的命令。

我们初始化的项目中 `scripts` 中的内容类似如下结构：

```
{
  "scripts":{
    "serve":"vue-cli-service serve",
    "test":"jest"
  }
}
```

可以通过配置 `scripts` 来指定运行在 `npm` 后面的命令。我们在初始化完成 `vue` , `react` 之类的项目时，工具已经给我们配置完成了命令，我们仅需要开箱即用即可。

如何运行命令呢？在我们安装完 `node` 之后， `node` 会为我们配置一个全局命令 `npm` 。

```
npm run serve
```

运行这行命令，等同于在当前目录下运行:

```
vue-cli-service serve
```

你可以视`scripts`为命令缩写。

1.4 dependencies

定义:指定了项目**运行所依赖**的模块

一般通过这种方式安装的模块，会将模块包添加到`dependencies`，而你的项目是无法脱离这个模块单独运行的。

```
npm install packageName --save
```

安装完成之后，会在你的`package.json`增加如下内容:

```
{
  "devDependencies":{
    "packageName":"version"
  }
}
```

packageName:就是第三方包名

version:安装的版本号，这个版本号支持`^`,'~','laster'。比如`^1.0.0`就说可以安装从1.0.0开始之后的版本。也有些时候为了项目稳定，会将版本号锁定，写成`1.0.0`。那么在安装的时候仅仅只会安装`1.0.0`版本。

格式	定义
1.0.0	锁定1.0.0版本，必须这个版本。
^1.0.0:插入号	比如^1.0.0，表示安装1.x.x的最新版本（不低于1.0.0），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。
~1.2.2:波浪号	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。
laster	安装最新的版本

1.5 devDependencies

定义:指定项目**`开发所需要`**的模块

在你的开发过程中，可能会引入一些第三方类库来提高开发效率，比较知名的第三方库`webpack`、`rollUp`、`less`、`babel`这些。

虽然我们开发过程中需要他们参与，但是在打包之后线上运行的时候是完全不需要他们的。这些类库就会归置到`devDependencies`中。

以下类库都建议安装到`devDependencies`：

- 单元测试支撑（mocha、chai）；
- 语法兼容（babel）；
- 语法转换（jsx to js、coffeescript to js、typescript to js）
- 程序构建与优化（webpack、gulp、grunt、uglifyJS）；
- css 处理器（postCSS、SCSS、Stylus）；
- 代码规范（eslint）；

1.6 peerDependencies

定义:提示宿主环境去安装满足插件peerDependencies所指定依赖的包，然后在插件import或者require所依赖的包的时候，永远都是引用宿主环境统一安装的npm包，最终解决插件与所依赖包不一致的问题。

由于npm@>3 之后的版本会自主管理包，使用peerDependencies的话，会避免打包不同版本的相同类库到包内。下面的介绍一些不一致情况。

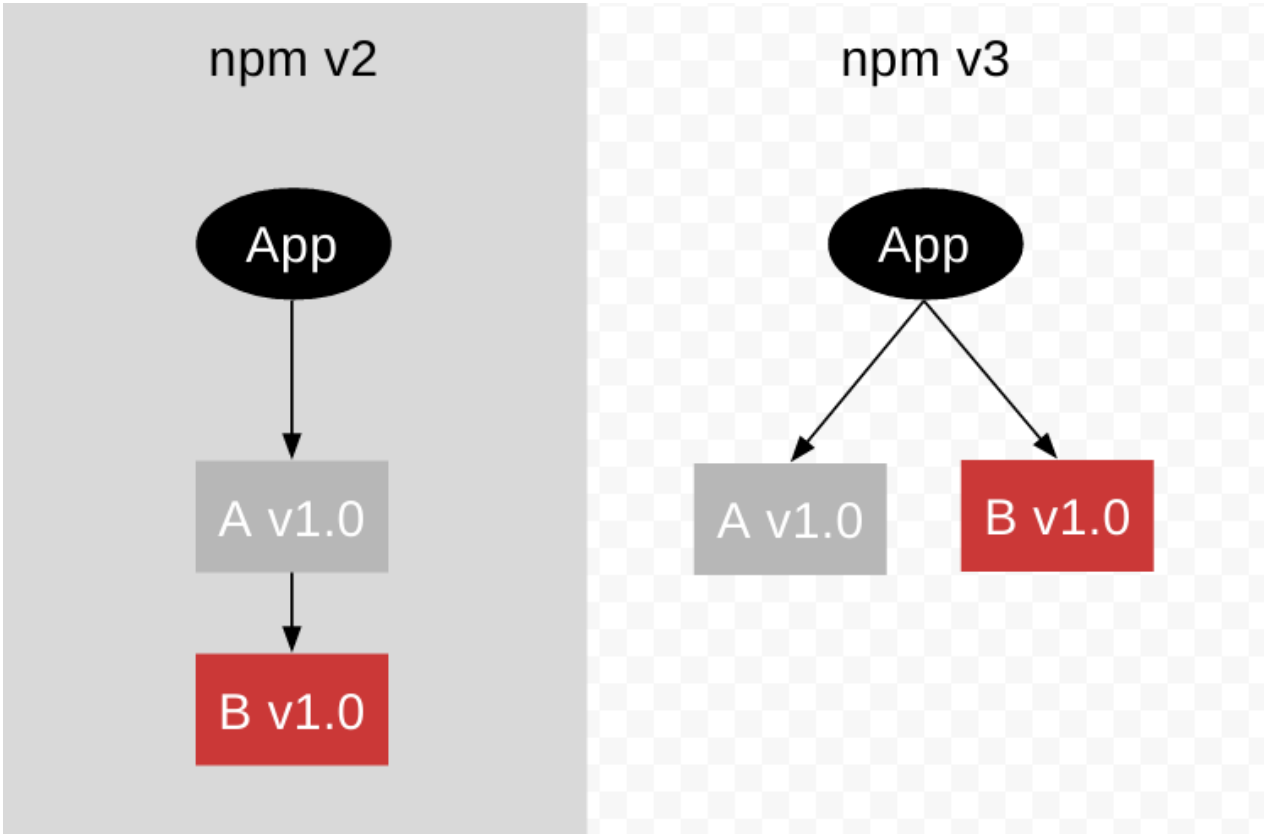
1.6.1 版本一致的情况下

如果某个package定义了PackageB的`peerDependencies`,那么在项目使用中把我列为依赖的话，那么这个项目也必需应该有对PackageB的依赖。

也就是说`package`在编译发布的时候，不会将`PackageB`打包入编译代码里面，他认为使用他的项目会安装这些依赖。

还为了在安装多个插件时，避免多次重复安装第三方库与打包出不同版本的第三方库。

在使用的版本一致的情况下，`node_modules`目录如下显示。



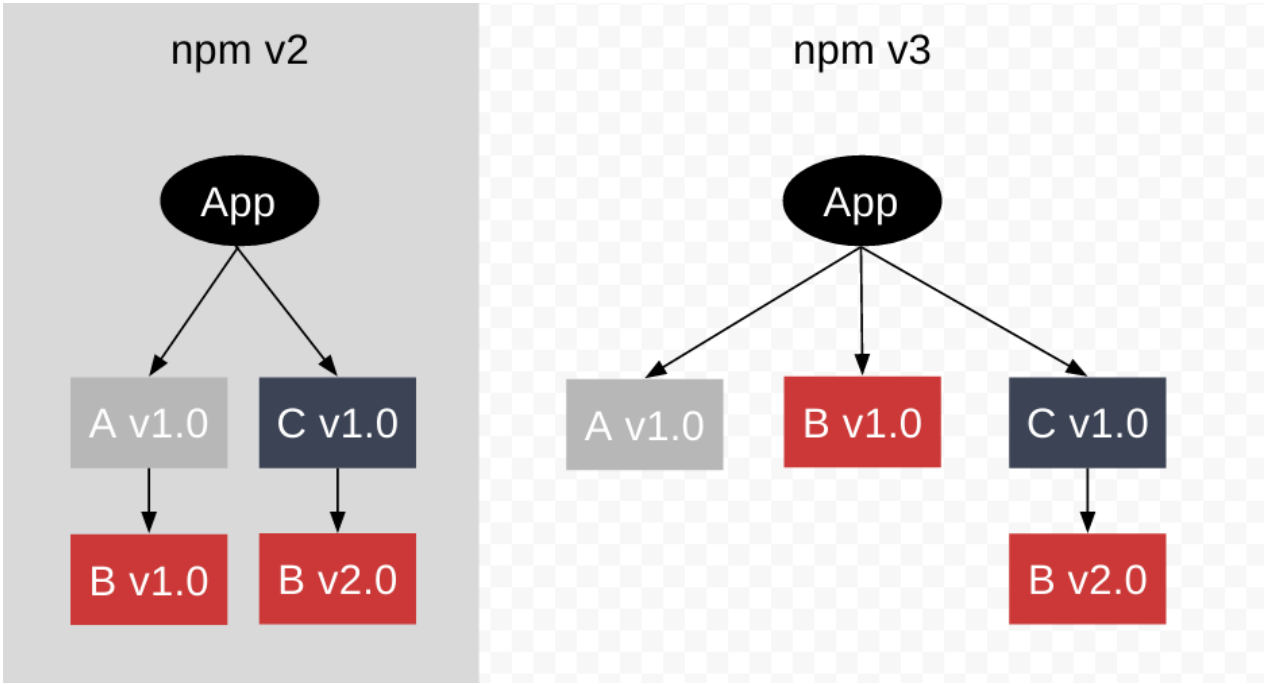
1.6.2 版本不一致的情况下

如果在项目中

A使用B 1.0版本

C使用B 2.0版本

那么`npm`在安装的时候会根据安装顺序，不同的版本放置在自己的目录下。



1.6.3 npm处理情况

- 如果用户显式依赖了核心库，则可以忽略各插件的`peerDependency`声明；
- 如果用户没有显式依赖核心库，则按照插件`peerDependencies`中声明的版本将库安装到项目根目录中；
- 当用户依赖的版本、各插件依赖的版本之间不相互兼容，貌似会报错让用户自行修复；（这里待确认）

> npm2 会主动强制安装依赖包，但是在npm3 中，依赖树的生成会尽量的扁平，相应`peerDependency`的行为有所变化。`peerDependencies`中声明的依赖，如果项目没有显式依赖并安装，则不会被npm 自动安装，转而输出`warning`日志，告诉项目开发者，你需要显式依赖了，不要再依靠我了。

1.6.4 模块依赖说明

来源:https://github.com/minhuang0/notes/issues/9

> 仅适用于 npm3
>
> 模块 A 安装 模块B 时，如何处理 package.json 里面的依赖；下面均用 父组件代替模块A，子组件代替模块 B
>
> node_modules: 模块 A 的 node_modules 目录
>
> child_node_modules: 模块 B 的 node_modules 目录，当模块 A 安装了模块 B 时，目录变成了 node_modules/child_modules_name/node_modules

Child dependencies

子组件仅在 dependencies 定义模块时，父组件的 dependencies 或 devDependencies 未定义模块时，会在 node_modules 安装子组件 dependencies 定义的模块版本；

Child devDependencies

子组件仅在 devDependencies 定义模块时，父组件不会安装子组件 devDependencies 里面模块；

Child peerDependencies

子组件仅在 peerDependencies 定义模块时，只会产生两类 warring 提示（模块不满足：unmet peer dependency 、模块错误：incorrect peer dependency），父组件 node_modules 里面不主动安装模块；

Child devDependencies、peerDependencies

子组件模块在 devDependencies 定义，且在 peerDependencies 里面规定了版本范围时，父组件不会自动安装子组件 devDependencies 定义的模块，但会提示两种 waring

当父组件 dependencies 和 devDependencies 均没有安装该模块时，子组件 peer dependency 定义存在的模块不满足：

```
warning " > test1@1.0.5" has unmet peer dependency "qs@6.5.x".
```

当父组件 dependencies 或 devDependencies 安装了一个错误的版本时会提示, 子组件 peer dependency 定义的模块存在错误

```
warning " > test1@1.0.5" has incorrect peer dependency "qs@6.5.x".
```

Child dependencies、peerDependencies

子组件在 dependencies 定义模块，且在 peerDependencies 里面规定了版本范围时，父组件会自动安装子组件 dependencies 定义的模块，但会有三种情况

- 当父组件 dependencies 和 devDependencies 均没有定义该模块时，父组件会在 node_modules 里面自动安装上子组件在 dependencies 定义模块， 并提示：

```
warning " > test1@1.0.5" has unmet peer dependency "date-fns@1.28.x".
```

- 当父组件 dependencies 或 devDependencies 有定义该模块时，父组件的模块不满足子组件 peerDependencies 的模块定义时，会在 node_modules 里面安装父组件 dependencies 或 devDependencies 定义的模块，并在 node_modules/children_modules/node_modules 里面安装子组件在 dependencies 定义的模块，并提示：

```
warning " > test1@1.0.5" has incorrect peer dependency "date-fns@1.28.x".
```

- 当父组件 dependencies 或 devDependencies 有定义该模块时，父组件的模块满足子组件 peerDependencies 的模块定义时，会在 node_modules 里面安装父组件 dependencies 或 devDependencies 定义的模块; 此时和子组件要求的模块一致；

总结

- 子组件同时在 peerDependencies 模块定义时，当父组件不满足条件时，会产生两类 warring 提示（不满足同辈依赖：unmet peer dependency 、同辈依赖错误：incorrect peer dependency）；
- 子组件仅在 devDependencies 模块定义时，不会让 node_modules 和 child_node_modules 安装该模块；在同时在 peerDependencies 模块定义时，和上面结果一样；
- 子组件在 dependencies 的模块定义会让父组件安装该配置：
    - 当父组件 dependencies 或 devDependencies 的模块定义与子组件 dependencies 的模块定义**一致时**，仅在 node_modules 下面安装一份该模块；
    - 当父组件 dependencies 或 devDependencies 的模块定义与子组件 dependencies 的模块定义**不一致时**，会在 node_modules 下面安装父组件配置的模块，在 child_node_modules(看上面定义) 安装子组件配置的版本；
- 子组件在 dependencies 的模块定义，且在 peerDependencies 模块定义时：
    - 当父组件在 dependencies 或 devDependencies 的模块定义**满足**子组件在 peerDependencies 的模块定义时，仅在 node_modules 下面安装一份该模块；
    - 当父组件 dependencies 或 devDependencies 模块定义**不满足**子组件在 peerDependencies 的模块定义条件时，会产生两类 warring 提示（不满足同辈依赖：

unmet peer dependency 、同辈依赖错误：incorrect peer dependency）；并在 node_modules 下面安装父组件配置的模块，在child_node_modules (看上面定义) 安装子组件配置的版本；

1.7 bin

定义: 为`scripts`提供命令简写

```
"bin": {
  "node": "./bin/node10"
}
```

这样可以定义当前在`scripts`中运行的`node`引用`10`版本。

1.8 main

定义:`main`字段指定了加载的入口文件，`require('moduleName')`就会加载这个文件。这个字段的默认值是模块根目录下面的`index.js`。

1.9 config

```
{
  "config" : { "port" : "8080" }
}
```

可以在`node`项目中这么使用

```
http
  .createServer(...)
  .listen(process.env.npm_package_config_port)
```

1.10 brower

定义: browser指定该模板供浏览器使用的版本。Browserify这样的浏览器打包工具，通过它就知道该打包那个文件。

1.11 engines

定义:`engines`字段指明了该模块运行的平台，比如 Node 的某个版本或者浏览器

```
{ "engines" : { "node" : ">=0.10.3 <0.12" } }
```

或指定`npm`版本

```
{ "engines" : { "npm" : "~1.0.20" } }
```

2. 打包处理

2.1 webpack

2.2 Rollup

2.3 TSDX

3. 版本修订

- 2020-02-18 第一版

4. 用处

\1. 希望区分自己私有的库和宿主环境公共库 \2. 希望所有的库全部读取宿主环境的node_modules \3. 如果出现库的版本不对，需要对库升级 \4. 不希望不同版本的类库打包入内

5. 参考

package.json参考系列:

https://javascript.ruanyifeng.com/nodejs/packagejson.html#toc2

https://www.cnblogs.com/wonyun/p/9692476.html

https://nodejs.org/es/blog/npm/peer-dependencies/

https://github.com/SamHwang1990/blog/issues/7

https://github.com/minhuang0/notes/issues/9

语义化版本:

https://semver.org/lang/zh-CN/