

npm 你了解多少

对于 npm，同学们一定没少接触，但是对于他的问题又有很多，无法摸清头脑。我们先简单了解一下 npm 的背景。nodejs 社区乃至 Web 前端工程化领域发展到今天，作为 node 自带的包管理工具的 npm 已经成为每个前端开发者必备的工具。但是现实状况是，我们很多人对这个 nodejs 基础设施的使用和了解还停留在：会用 npm install 这里（一言不合就删除整个 node_modules 目录然后重新 install 这种事你没做过吗？害羞脸）。

npm 介绍

npm 是 node 的包管理工具，定义明确就是用来管理 node 的包，包括安装，卸载，更新，发布等

当然 npm 能成为现在世界上最大规模的包管理系统，很大程度上确实归功于它足够用户友好，你看即使我只会执行 install 也不必太担心出什么大岔子。但是 npm 的功能远不止于 install 一下那么简单，这篇文章帮你扒一扒那些你可能不知道的 npm 原理、特性、技巧，以及最佳实践。

1. npm init

init 大家都知道是什么意思，“**初始化**”，这个初始化的含义和目的是什么呢，初始化 npm？显然不是这么简单的。在使用 npm 的项目中都会看到一个文件，**package.json**。这个文件和我们的初始化有关。大家对于 npm 的第一印象是‘包管理工具’，这个管理不单单是安装，卸载。既然是管理就要起到管理的作用，那么它的作用是：**初始化 package.json 文件，这个文件用于执行脚本，记录当前项目的包（webpack, loader 等等）等操作。**

当我们在终端(Terminal)，也就是常说的 cmd 或者 git bash 类似的终端窗口上。执行 npm init。会出现类似下面的窗口。

```
C:\Users\DYZ96\Desktop\ding>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (ding) demo
version: (1.0.0)
description: a npm demo
entry point: (a.js) index.js
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\DYZ96\Desktop\ding\package.json:

{
  "name": "demo",
  "version": "1.0.0",
  "description": "a npm demo",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this OK? (yes) yes
C:\Users\DYZ96\Desktop\ding>z
```

在这里我们能看到当我们执行 `npm init` 后终端会依次询问 `name`, `version`, `description` 等字段。可以自行填写，也可以不填。采用默认值或者空值。如果采用默认值一路回车就可以了。而如果想要偷懒步免去一直按 `enter`，在命令后追加 `-yes` 或者 `-y` 参数即可，其作用与一路下一步相同。这样下来我们的一个带有配置性质的 `JSON` 对象被生成，用于管理项目。

对于 `package.json` 文件的默认行为能不能修改呢，就是里面默认的配置字段能不能修改呢，是可以修改的。但是意义不大在这里就不和同学们过多阐述了，如果有想尝试的同学，查询官网相关操作，即可完成配置。

2. npm install

依赖管理是 `npm` 的核心功能，原理就是执行 `npm install` 从 `package.json` 中的 `dependencies`, `devDependencies` 将依赖包安装到当前目录的 `./node_modules` 文件夹中。或者安装依赖包时，记录到 `dependencies` 或者 `devDependencies`。至于这两个字段什么意思我们稍后给大家解释。

我们先了解一下 `package` 的概念。

我们都知道要手动安装一个包时，执行 `npm install <package>` 命令即可。这里的第三个参数 `package` 通常就是我们所要安装的包名，默认配置下 `npm` 会从默认的源 (`Registry`) 中查找该包名对应的包地址，并下载安装。但在 `npm` 的世界里，除了简单的指定包名，`package` 还可以是一个指向有效包名的 `http url/git url/文件夹路径`。后者了解一下即可。

通常来说安装 依赖包有以下几种方式。安装都是通过 `npm install` 进行安装。就是后面的修饰符有些差异。

`Npm install webpack` 直接安装想要的依赖包。

`Npm install webpack@4.41` 安装依赖包的同时确定版本

`Npm install webpack@latest` 安装最新的依赖包（和 `npm install webpack` 性质相同）

npm install 是如何工作的？

`npm install` 执行完毕后，我们可以在 `node_modules` 中看到所有依赖的包。虽然使用者无需关注这个目录里的文件夹结构细节，只管在业务代码中引用依赖包即可，但了解 `node_modules` 的内容可以帮我们更好理解 `npm` 如何工作。接下来以最新版的 `npm` 举例。

执行 `npm install`:

1. 执行工程自身 `preinstall`

当前 `npm` 工程如果定义了 `preinstall` 钩子此时会被执行。

2. 确定首层依赖模块

首先需要做的是确定工程中的首层依赖，也就是 `dependencies` 和 `devDependencies` 属性中直接指定的模块（假设此时没有添加 `npm install` 参数）。工程本身是整棵依赖树的根节点，每个首层依赖模块都是根节点下面的一棵子树，`npm` 会开启多进程从每个首层依赖模块开始逐步寻找更深层级的节点。

3. 获取模块

获取模块是一个递归的过程，分为以下几步：获取模块信息。在下载一个模块

之前，首先要确定其版本，这是因为 `package.json` 中往往是 semantic version（semver，语义化版本）。此时如果版本描述文件（`npm-shrinkwrap.json` 或 `package-lock.json`）中有该模块信息直接拿即可，如果没有则从仓库获取。如 `package.json` 中某个包的版本是 `^1.1.0`，npm 就会去仓库中获取符合 `1.x.x` 形式的最新版本。获取模块实体。上一步会获取到模块的压缩包地址（`resolved` 字段），npm 会用此地址检查本地缓存，缓存中有就直接拿，如果没有则从仓库下载。查找该模块依赖，如果有依赖则回到第 1 步，如果没有则停止。

4. 模块扁平化

上一步获取到的是一棵完整的依赖树，其中可能包含大量重复模块。比如 A 模块依赖于 `lodash`，B 模块同样依赖于 `lodash`。在 npm3 以前会严格按照依赖树的结构进行安装，因此会造成模块冗余。

从 npm3 开始默认加入了一个 dedupe 的过程。它会遍历所有节点，逐个将模块放在根节点下面，也就是 `node_modules` 的第一层。当发现有重复模块时，则将其丢弃。

这里需要对重复模块进行一个定义，它指的是模块名相同且 semver 兼容。每个 semver 都对应一段版本允许范围，如果两个模块的版本允许范围存在交集，那么就可以得到一个兼容版本，而不必版本号完全一致，这可以使更多冗余模块在 dedupe 过程中被去掉。

比如 `node_modules` 下 `foo` 模块依赖 `lodash@^1.0.0`，`bar` 模块依赖 `lodash@^1.1.0`，则 `^1.1.0` 为兼容版本。

而当 `foo` 依赖 `lodash@^2.0.0`，`bar` 依赖 `lodash@^1.1.0`，则依据 semver 的规则，二者不存在兼容版本。会将一个版本放在 `node_modules` 中，另一个仍保留在依赖树里。

5. 安装模块

这一步将会更新工程中的 `node_modules`，并执行模块中的生命周期函数（按照 `preinstall`、`install`、`postinstall` 的顺序）

最后一步是生成或更新版本描述文件，`npm install` 过程完成。

之前提到两个概念一个是 `dependencies` 另一个是 `devDependencies`。这两个字段用来干什么的。如何动态向里面添加我们安装的依赖。

这两个字段的的目的用于记录生产环境和开发环境的使用依赖。但只是起到记录的作用。

开发环境和生产环境的区别在于，开发环境需要我们本地的环境进行打包解析。比如 Webpack 的配置，以及对应的各种 plugin 或者 loader。我们只是在开发的时候利用这些东西给我们转化语法，最终打包出来的结果一定没有他们。

生产环境指的是把所有东西都转换好了，把最终的结果打包上线。所以不需要本地环境相关的一来播。

当我们执行 `npm install xxx --save` 或者 `npm install xxx -S` 时，我们的依赖包的包名会放到 `dependencies` 里面，

当我们执行 `npm install xxx --save-dev` 或者 `npm install xxx -D`，我们的依赖包的包名会放到 `devDependencies` 中。

npm scripts

在 package.json 当中存在一个字段 scripts

```
"scripts": {  
  "test": "echo dui"  
},
```

我们就可以通过 `npm run test` 命令来执行这段脚本，像在 shell 中执行该命令 `echo dui` 一样，看到终端输出 `dui`。

这里面的出现了一个新的命令 `npm run`。他可以查找我们在 scripts 里面定义好的脚本去执行。比如说：一个 webpack 的打包过程 `webpack --config ./webpack.config.js --mode development` 对于这种命令每次执行我们都需要打这么长的内容，我们可以通过添加到 scripts 中进行简化 像这样：

```
"scripts": {  
  "test": "echo dui",  
  "dev": "webpack --mode development --config ./webpack.development.js"  
}
```

在这里面我们直接运行 `npm run dev` 即可，是不是很方便。

但是这里要注意，我们看似：`npm run dev` 和上面的那一长串的功效是一样的，但是存在实质上的不同。

当我们执行 `webpack` 时，这个 `webpack` 指令是在系统层面的 `webpack` 指令，当我们没有全局安装 `webpack` 时，我们执行 `webpack` 指令，会报错，不是一个有效的命令等等。

这里说一下，想安装全局的依赖包 加上修饰符 `-g` 或者 `-global`。

当我们通过 `npm run xxx` 时，后面的指令会执行，先在当前目录下 `./node_modules/.bin` 目录下查找指令，没有找到再去全局找。

我们可以看一下 `.bin` 目录下面的可执行文件：

这台电脑 > 桌面 > ding > node_modules > .bin					搜索 ".bin"
名称	修改日期	类型	大小		
sshpk-verify	2019/5/29 16:48	文件	1 KB		
sshpk-verify.cmd	2019/5/29 16:48	Windows 命令脚本	1 KB		
terser	2019/5/29 15:49	文件	1 KB		
terser.cmd	2019/5/29 15:49	Windows 命令脚本	1 KB		
uuid	2019/5/29 16:48	文件	1 KB		
uuid.cmd	2019/5/29 16:48	Windows 命令脚本	1 KB		
webpack	2019/5/29 15:49	文件	1 KB		
webpack.cmd	2019/5/29 15:49	Windows 命令脚本	1 KB		

这些 `.cmd` 文件就是我们可执行的指令。

接下来总结下：

`npm run` 命令执行时，会把 `./node_modules/.bin/` 目录添加到执行环境的 `PATH` 变量中，因此如果某个命令行包未全局安装，而只安装在了当前项目的 `node_modules` 中，通过 `npm run` 一样可以调用该命令。

执行 `npm` 脚本时要传入参数，需要在命令后加 `--` 标明，如 `npm run test -- --grep="pattern"` 可以将 `--grep="pattern"` 参数传给 `test` 命令

`npm` 提供了 `pre` 和 `post` 两种钩子机制，可以定义某个脚本前后的执行脚本

运行时变量：在 `npm run` 的脚本执行环境内，可以通过环境变量的方式获取许多运行时相关信息，以下都可以通过 `process.env` 对象访问获得：

`npm_lifecycle_event` - 正在运行的脚本名称

`npm_package_<key>` - 获取当前包 `package.json` 中某个字段的配置值：如 `npm_package_name` 获取包名

`npm_package_<key>_<sub-key>` - `package.json` 中嵌套字段属性：如 `npm_package_dependencies_webpack` 可以获取到 `package.json` 中的 `dependencies.webpack` 字段的值，即 `webpack` 的版本号

目前来说，大家了解到这里就可以，`npm` 的指令还是很多的，如果了解 `npm` 其他指令的同学，官网自行了解一下。