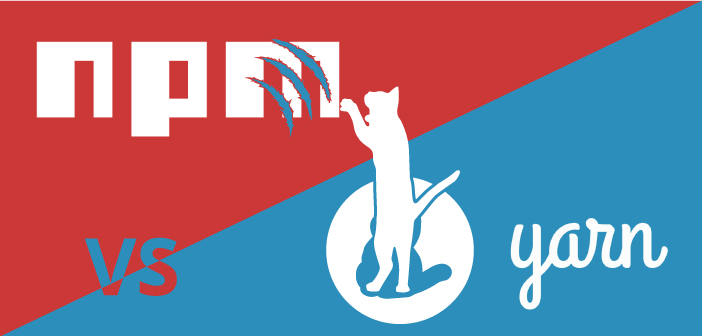
第02课：关于包管理器你应该知道的



我发现介绍 npm 以及 Yarn 等包管理器的文章相对较少，但是在我们的实际开发中却经常遇到和包管理器相关的问题。有时候我们可以通过 rm -rf node\_modules && npm i 暴力地将问题解决，有时候却不能。因此我们总结下使用包管理器的正确姿势。

### 什么是包管理器？

让我们还是从概念开始说起，包管理器是一个可以让开发者便捷地获取代码和分发代码的工具。一些具有特定功能的代码（框架、库等）按照特性形式被封装成包，开发者可以通过包管理器安装这些包，也可以把自己的代码通过包的方式分享给别人使用。

在很多语言中都有包管理器，比如 Java 中的 Maven，Ruby 中的 gem。在目前 JavaScript 应用中，最主流的包管理器是 npm 和 Yarn，本文主要介绍它们。

JavaScript 是一个缺乏标准库的语言。当你想解决 URL 处理、日期处理这类很常见的问题，在没有标准库的情况下就只能自己动手写代码或者从自己以前的工程中拷贝代码。而使用 npm 等包管理器最大的好处是里面有很多非常成熟的包来解决这类问题，可以直接安装到项目中使用，给开发工作带来了很大的便利。

### npm 特性一览

下面我会着重介绍 npm 在开发中最常用的特性。由于不同 npm 版本之间会有一些差异，我会尽量使用共通的样例，使不同版本之间表现得尽量一致。如果你是一个初学者我建议你使用和我一样或者更高的版本，这里我使用的 npm 版本是 5.5.1。

#### 初始化你的 npm 工程

不妨让我们从工程的初始化开始介绍 npm。

mkdir npm-demo && cd npm-demo # 新建并进入目录

npm init # 初始化 npm 项目

初始化时会要求你输入一些基本信息，这里我们使用默认的，一路回车下来即可。接着打开工程中的 package.json：

{

"name": "npm-demo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "",

"license": "ISC"

}

这里介绍几个比较重要的字段：

* name：当我们将包上传到 npm 平台上时，它类似于该包的 id，不能和平台上已有的包冲突。别人如果要安装你这个包的时候也是用这个名字。
* version：这里的版本号遵循 npm 特有的 semver 语法，后面我们会特别介绍。
* main：这是包执行的入口，当这个包被别的地方引入的时候，引入的其实就是这个入口文件。
* scripts：npm 提供的脚本命令功能，比如当我们执行 npm run test，npm 就会帮我们执行后面定义的脚本，利用它可以实现很多开发和构建中的自动化命令。

#### 仓库与依赖

在提到所有与依赖相关的话题之前，我想首先介绍 npm 仓库（ registry ）。仓库是一个遵循 npm 特定包规范的站点，它提供 API 来让用户上传和下载包、获取包信息、以及管理用户账号。

npm 提供了一个官方的仓库，但是在国内由于特定原因我们需要翻墙才能使用它。因此一般的做法是使用淘宝提供的仓库。设置方法：

npm config set registry https://registry.npm.taobao.org

通过 npm install 我们可以从远程或者目标路径获取 npm 包。这里有一些比较重要的参数：

* -S, --save：在 npm5 以前，你需要指定这个参数来让刚刚安装的包记录在 package.json 的 dependencies 中，否则只是安装在项目本地的 node\_modules 中。一般我们要求必须要加这个参数，否则就会造成不同环境下 npm install 的结果不一样。在 npm5 之后这变成了一个默认行为而不需要手动添加了。
* -D, --save-dev：指定当前环境是开发环境，将包的信息记录在 devDependencies 中。

在 npm 中依赖主要分为两种—— dependencies 和 devDependencies（还有其它类型的依赖但是这里不涉及）。两者的区别在于 dependencies 是生产环境下的依赖（ prod )，devDependencies 则是开发环境下的依赖（ dev ）。

比如说我现在工程中的 pakcage.json 是这样：

{

"name": "npm-demo",

"dependencies": {

"split": "^1.0.1"

},

"devDependencies": {

"q": "^1.5.0"

},

...

}

* 当不加任何参数时执行 npm install，这两个包都会被安装到 node\_modules 中。
* 当执行 npm install --only=prod，只会安装 dependencies 中的包。
* 当执行 npm install --only=dev，只会安装 devDependencies 中的包。

通过添加安装参数，可以实现环境区分。当工程规模比较大的时候，通过只安装当前环境的包可以加快总体安装的速度。另外一点需要注意的是，当我们把当前这个包发布出去以后，别人通过 npm install npm-demo 安装它时，只会安装它的 dependencies，而会忽略 devDependencies。这意味着所有与功能相关的依赖都要放在 dependencies 中，而 devDependencies 中通常会放一些如构建工具（ Rollup ）、质量检测工具（ Eslint ）等只有本地开发才使用的包。

#### semver ——一把双刃剑

上面提到版本号 ^1.0.1 前面的 ^ 是 npm 特有的 semver 版本号语法中的符号。semver 全称是 semantic versioning，也就是语义化的版本。它设计的初衷在于允许依赖的版本是动态的，因而可以快捷地获取到一定范围内的最新版本。下面列出了 semver 的各种形式来让我们直观地理解一下：

* ^version：对应所有中版本号和小版本号。比如 ^1.0.1 可以理解为所有形式为 1.x.x的最新版本。
* ~version：对应所有小版本号。比如 ~1.0.1 可以理解为所有形式为 1.0.x 的最新版本。
* version：对应特定版本。比如版本号为 1.0.1 ，则安装的模块版本只能为 1.0.1。

动态版本使得开发者无需更改 pakcage.json 中的依赖版本信息，便可以获取到他所定义的范围内最新的版本。比如当我们通过 npm install split 安装了这个包之后，package.json 中它的版本号为 ^1.0.1。当 split 这个包的作者发布了一个新的版本 1.0.2 的时候，我们下次重新执行 npm install 就会装这个 1.0.2 的版本了。

semver 带来的优势是可以使 npm 包的发布者很方便地将小版本推送到使用者。比如当有一些小的 bugfix 时候，用户不需要有任何感知，便可以将其中的问题修复。其实 npm 对于各个版本号的意义也有约定。

* 小版本发布（如 1.0.1 -> 1.0.2）：bugfix 以及微小的改动
* 中版本发布（如 1.0.1 -> 1.1.0）：添加了新的特性，同时并不会破坏原有的功能或更改接口
* 大版本发布（如 1.0.1 -> 2.0.0）：添加了破坏性的改动，有功能或接口不再兼容

只要包的发布者和使用者都遵循这样的规则，那么还是能带来很多收益的。但是问题就在于这种约定并非强制，也并不能保证百分百的开发者都能遵守。试想一下，假如某一个包的开发者更改了接口名称，而只发布了一个小版本。那所有调用了这个更改前的接口的使用方，在不知情的情况下更新到这个新版本就无法正常工作了。

之前公司内部的一个项目使用了一个 UI 库，在几乎未改任何代码的情况下重新发布之后发现页面样式出了问题，造成了线上故障。开发同学查了很久才发现是 UI 库自己升了一个小版本，把自己内部的一处样式改掉了。类似这样的问题其实很难避免，在使用 semver 动态规则的同时，我们的应用无时不刻在冒着依赖不稳定的风险。

#### 固定版本号，保障你的应用

由之前的讨论可以看出，为了保证构建的可靠性，需要将 JavaScript 模块的版本固定下来，这样才能使每次构建的结果保持一致。可能也是发现了 semver 存在这样的问题，在 npm 早期的版本中提供了 shrinkwrap 这样一个固定模块版本号的功能。在项目中执行 npm shrinkwrap，即可根据当前 node\_moduels 中的包信息生成依赖和版本描述信息—— npm-shrinkwrap.json。当该工程下一次执行 npm install 时，就会按照该文件中的信息获取特定的版本，而不再使用 package.json 中的动态版本规则。一个 shrinkwrap 文件的结构类似以下这种形式：

{

"name": "foobar",

"version": "1.0.0",

"dependencies": {

"through": {

"version": "2.3.8",

"resolved": "https://registry.npmjs.org/through/-/through-2.3.8.tgz",

"integrity": "sha1-DdTJ/6q8NXlgsbckEV1+Doai4fU="

}

}

}

利用 shrinkwrap 我们可以保证构建的一致性，但是在实际使用中仍然存在一些问题。在较低版本的 npm 中，当我们执行模块的添加、更新、删除操作时，shrinkwrap.json 并不会自动同步，仍然需要手动执行 npm shrinkwrap，这无疑使工作流程更复杂。假如某个项目成员忘记了手动同步，就会导致最终构建结果与预期不一致。另外 shrinkwrap 作为 npm 中一个非默认的构建模式，让每个人接受和使用仍需要一定的学习成本。

在 npm4 中解决了这个问题，shrinkwrap.json 不再需要手动维护，当执行包的安装、更新、删除等操作后 npm 会自动为我们更新它。而到了 npm5 的版本，固定版本号成为了一个默认行为，并且提出了新的约束版本号的文件—— package-lock.json。它和 shrinkwrap.json 本质上是同样的结构和功能，唯一的区别在于 package-lock.json 只能用在顶层包而不能存在于依赖包中，而 shrinkwrap.json 则可以。

早期版本的 shrinkwrap 命令还不够成熟，在生成过程中容易出现错误。因此建议最好使用 npm5 的固定版本号功能，无论是 package-lock.json 还是 shrinkwrap.json，只要存在一个于工程中即可保证构建过程中依赖的稳定性。

#### 使用 scripts 让工作自动化

在初始化 npm 项目时候，我们在 package.json 看到有 scripts 字段：

{

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

}

}

这是 npm 提供的脚本命令功能，脚本命令分为两种：

* npm 生命周期命令：在一个 npm 包的发布和安装等过程中允许开发者写很多钩子，比如 preinstall、postinstall、prepublish、postpublish 等等，这对于包的开发者来说非常有用。
* 自定义命令：这里是除了以上那些生命周期中的命令以外的命令，允许开发者自行定义命令名称和执行脚本。

{

"scripts": {

"dev": "webpack-dev-server",

"build": "eslint ./src && webpack"

}

}

上面两个自定义命令，分别包含了使用于不同环境下的脚本，通过这项功能我们可以免去总是输入一大串命令的痛苦。合理地配置 scripts 甚至可以使我们不必使用 Gulp、Grunt 等自动化流程工具，完全交给 npm scripts 来完成。

### Yarn 带来的变革

Yarn 是 Facebook 公司在 2016 年 10 月 11 日开源的模块管理器，它宣称比 npm 更快、更安全、更可靠。Yarn 并不重头建立一个新的 Javascript 模块仓库，而只是替代 npm 客户端来管理原有的 node\_modules 中的模块，并弥补 npm 的缺陷。鉴于某些原因，为了使用 Yarn ，我们需要把它的源指向国内，比如采用 cnpm 的源：

yarn config set registry https://registry.npm.taobao.org

在 Yarn 出现的时候，npm 还是 4 的版本。那时候锁定版本号并不是一个默认行为，Yarn 带来的一个重大改进就是会帮助开发者自动生成和维护版本号描述文件。初次执行 Yarn 时会在项目中自动生成一个名为 yarn.lock 的文件，它与 npm shrinkwrap 的内容形式很相近，并且会随着模块的更新自动同步。

Yarn 在当时另外一项比较大的改进就是性能比当时 npm 更优。Yarn 为了解决当时 npm 安装模块速度慢问题，在拉取包时采用并行操作，优化了请求队列，更高效地利用当前的网络资源。同时默认的 yarn.lock 也无形中减少了解析 semver 与获取模块最新版本的时间。

npm 的版本号锁定问题和性能问题其实已经被开发者诟病了许久，Yarn 这个竞争对手的出现可以说给 npm 带来了改进的动力。后来 npm5 出现学习了很多 Yarn 的特性，比如 npm 现在也会默认生成的 pakcage-lock.json，并且改进了缓存策略来提升安装包时候的速度。

对于开发者来说目前 npm5 和 Yarn 都是可以满足日常开发需要的，使用自己觉得顺手的工具就好。

### 使用包管理器的正确姿势

#### 使用 npm5 或 Yarn 来管理模块

对于现在的开发者来说不建议使用低版本 npm 进行开发，原因如下：

* npm3 之前有一个很大的问题，即 node\_modules 中的包层级是嵌套的，这会造成整个安装速度极慢且有很多重复和冗余的包。
* 早期的 npm 版本安装模块不会自动生成包版本信息文件，这个问题上面已经说了，会为构建过程留下隐患。
* npm5 和 Yarn 有更加良好的缓存机制，可以有效提升安装速度。
* 现在的工具有更友善的交互，报错信息也会比较清晰，方便查找模块安装错误。

#### 万能修复大法？rm -rf node\_modules && npm i

很多开发同学在使用 npm 的时候一发现 npm 模块有问题就执行 rm -rf node\_modules && npm i，一些情况下可以解决问题，一些情况下却不能，让我们尝试去看看这是为什么。

当我们执行 npm install 或者 yarn 来安装模块的时候，大概经历了几个过程：

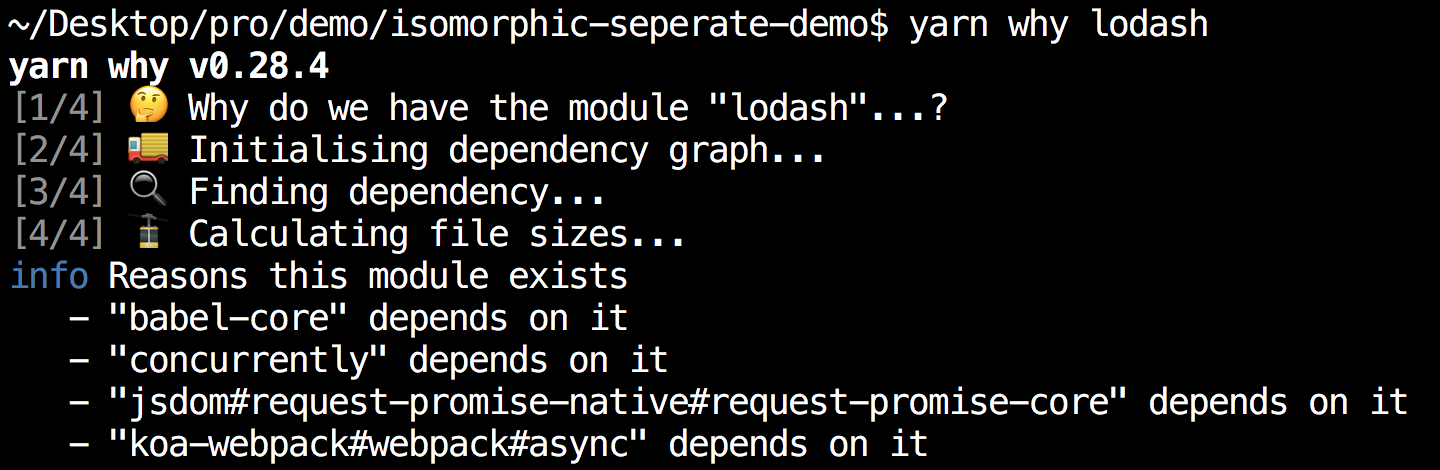
1. 首先会寻找包版本信息文件（ pakcage-lock.json，yarn.lock等），如果发现有版本信息文件，则依照它来进行模块安装。
2. 检查 pakcage.json 中的依赖，如果此时项目中不存在版本信息文件，则完全按照 pakcage.json 进行安装，并生成一个版本信息文件。如果此时存在版本信息文件，则只会安装 package.json 中有而版本信息文件中没有的包。
3. 如果确实有这种新包，则更新版本信息文件。

因此当我们发现项目中的某个包和我们预想不一致时，首先查看版本信息文件中该包的来源和版本，因为在安装过程中它的优先级最高。有的时候执行了 rm -rf node\_modules && npm i 也没有解决问题，可能是由于版本信息文件中这个包本身就有问题，无论你怎么删掉重装也还是一样。

#### 不知道包怎么来的？试试 yarn why

npm 依赖包的层级通常很深，有时候我们不知道怎么就把一个包装在项目里了，但是又不清楚它是由谁引入的，这时可以通过 yarn 来帮我们查看。

yarn why <package-name>



通过 log 可以清晰地看出依赖关系，都有哪些包把 lodash 引入到了项目中。

#### 搭建内网 npm 仓库

在日常开发中，很多公司内部的包我们不希望发布到 npm 官方仓库，因为里面可能涉及一些不希望暴露给外部的代码。这时我们可以选择在公司内网搭建 npm 私有仓库。

借助 npm 的 scope 特性我们可以实现公有和私有包的区分，在为 npm 包设定名字的时候可以加上 scope。它的格式是 @somescope/somepackagename，比如公司内部的包命名可以为 @mynpm/myui，这种包只能上传到私有仓库，官方仓库是不接受的。

现在比较主流的解决方案有 cnpm 和 verdaccio。两者的区别主要在于同步官方仓库包的机制：

* cnpm 的机制把所有官方仓库的包拷贝到私有仓库中，然后每隔几分钟从官方仓库中同步一次。这样的优点是用户下载包会比较快，缺点是包刚发布上去可能需要等待几分钟来同步或者去手动同步。另外就是对硬盘要有一定要求，毕竟全球那么多 JavaScript 开发者每天都在发布无数的包上去，都需要去同步到内网仓库中。
* verdaccio 相对来说则智能一些。它并不会主动拷贝和同步官方源的包，只有在用户去安装包的时候才从官方的仓库拉取，并在本地生成缓存。经过验证在一段时间之后它的缓存命中率是非常高的，因为用户在绝大多数情况下都在安装同样的包。

以上通过区分内外网仓库，可以防止内部的包泄露到外网中。