

作者：@小蘑菇小哥 百度上海研发中心资深前端工程师，pwa开源解决方案iavas作者，百度mp核心开发者

原文：https://zhuanlan.zhhu.com/p/56002037

在我看来，nodejs 的成功原因除了它采用了前端 js 相同的语法，直接吸引了一大波前端开发者作为初始用户之外，它内置的包管理器 npm 也居功至伟。npm 能够很好的管理 nodejs 项目的依赖，也使得开发者发布自己的包的异常容易。这样一来，不论你使用别人的包，还是自己发布包给别人使用，成本都不大。这和我大学学习的 Java 1 x 相比就轻松愉快的多（现在 Java 已今非昔比，我不敢乱评论），开发者热情高涨的话，整个生态就会更加活跃，进步速度也就更加快了。看一看 GitHub 上 JS 项目的占比，再看看 npm 官网包的数量，就能略知一二。

前阵子公司的一名新人问了我一个问题：如何区分项目的依赖中，哪些应该放在 dependencies，而哪些应该放在 devDependencies 呢？

其实这个问题我在早先也有过，所以非常能够体会他的心情。为了防止误人子弟，我查阅了一些资料，发现其实 nodejs 中总共有 5 种依赖：

- dependencies (常用)
- devDependencies （常用）
- peerDependencies （不太常用）
- bundledDependencies (我之前没用过)
- optionalDependencies (我之前没用过)

所以我趁此机会，整理了这篇文章，分享给更多仍有此迷茫的人们。

dependencies

这是 npm 最基本的依赖，通过命令 npm i xxx -S 或者 npm i xxx —save 来安装一个包，并且添加到 package.json 的 dependencies 里面（这里 i 是 install 的简写，两者均可）。

如果直接只写一个包的名字，则安装当前 npm registry 中这个包的最新版；如果要指定版本的，可以把版本号写在包名后面，例如 npm i webpack@3.0.0 —save。

npm install 也支持 tag，tar 包地址等等，不过那些不太常用，可以查看官方文档。

dependencies 比较简单，我就不再多做解释了。注意一点：npm 5 x 开始可以省略 —save，即如果执行 npm install xxx，npm 一样会把包的依赖添加到 package.json 中去。要关闭这个功能，可以使用 npm config set save false。

devDependencies

很多 nodejs 新手都不满 dependencies 和 devDependencies，导致依赖随便分配，或者把依赖统统都写在 dependencies。这也是我编写本文的初衷。

先说定义。顾名思义，devDependencies 就是开发中使用的依赖，它区别于实际的依赖，也就是说，在线上状态不需要使用的依赖，就是开发依赖。

再说意义。为什么 npm 要把它单独拆出来呢？最终目的是为了减少 node_modules 目录的大小以及 npm install 花费的时间，因为 npm 的依赖是被嵌套的，所以可能看上去 package.json 中只有几个依赖，但实际上它又扩散到 N 个，而 N 个又扩散到 N 平方个，一层层扩散出去，可谓子子孙孙无穷尽也。如果能够尽量减少不使用的依赖，那么就能够节省线上机器的硬盘资源，也可以节省部署上线的时间。

在实际开发中，大概有这么几类可以归为开发依赖：

构建工具

现在比较热门的是 webpack 和 rollup，以往还有 grunt, gulp 等等。这些构建工具会生成生产环境的代码，之后在线上使用时就直接使用这些压缩过的代码。所以这类构建工具是属于开发依赖的。

像 webpack 还分为代码方式使用（webpack）和命令行方式使用（webpack-cli），这些都是开发依赖。另外它们可能还会提供一些内置的常用插件，如 xxx-webpack-plugin，这些都算开发依赖。

预处理器

这里指的是对源代码进行一定的处理，生成最终代码的工具。比较典型的有 CSS 中的 less, stylus, sass, scss 等等，以及 JS 中的 coffee-script, babel 等等。它们做的事情虽然各有不同，但原理是一致的。

以 babel 为例，常用的有两种使用方式。其一一是内嵌在 webpack 或者 rollup 等构建工具中，一般以 loader 或者 plugin 的形式出现，例如 babel-loader，其二二是单独使用（小项目较多），例如 babel-cli，babel 还额外有自己的插件体系，例如 xxx-babel-plugin，类似地，less 也有与之对应的 less-loader 和 lessc，这些都算作开发依赖。

在 babel 中还有一个注意点，那就是 babel-runtime 是 dependencies 而不是 devDependencies。具体分析我在之前的 babel 文章中提过，就不再重复了。

测试工具

严格来说，测试和开发并不是一个过程。但它们同属于“线上状态不需要使用的依赖”，因此也就归入开发依赖了。常用的如 chai, e2e, karma, coveralls 等等都在此列。

真的是开发才用的依赖包

最后一类比较杂，很难用一个大概类括起来。总之就是开发时需要使用的，而实际上上线时要么是已经打包成最终代码了，要么就是不需要使用了。比如 webpack-dev-server 支持开发热加载，线上是不用的；babel-register 因为性能原因也不能用在线上，其他还可能和具体业务相关，就看各位开发者自己识别了。

把依赖安装成开发依赖，则可以使用 npm i -D 或者 npm i —save-dev 命令。

如果想达成刚才说的缩减安装包的目的，可以使用命令 npm i —production 忽略开发依赖，只安装依赖，这通常在线上机器（或者 QA 环境）上使用。因此还有一个最根本的识别依赖的方式，那就是用这条命令安装，如果项目跑不起来，那就是识别有误了。

peerDependencies

如果仅作为 npm 包的使用者，了解前两项就足够我们日常的使用了。接下来的三种依赖都是作为包的发布者才会使用到的手段，所以我们转换角色，以发布者的身份来讨论接下来的问题。

如果我们开发一个常规的包，例如命名为 my-lib，其中需要使用 request 这个包来发送请求，因此代码里一定会有 const request = require('request')，如上面的讨论，这种情况下 request 是作为 dependencies 出现在 package.json 里面的，那么在使用者通过命令 npm i my-lib 安装我们依赖的时候，这个 request 也会作为依赖的一部分被安装到使用者的项目中。

那我们还为什么需要这个 peerDependencies 呢？

根据 npm 官网的文档，这个属性主要用于插件类（Plugin）项目。常规来说，为了插件生态的繁荣，插件项目一般会被设计地尽量简单，通过数据结构和固定的方法接口进行耦合，而不会要求插件项目去依赖本体。例如我们比较熟悉的 express 中间件，只要你返回一个方法 return function someMiddleware(req, res, next)，它就成为了 express 中间件，受本体调用，并通过三个参数把本体的信息传递过来，在插件内部使用。因此 express middleware 是不需要依赖 express 的。类似的情况还包括 Grunt 插件，Chai 插件和 Winston transports 等。

但很明显，这类插件脱离本体是不能单独运行的。因此虽然插件不依赖本体，但想要自己能够实际运行起来，还得要求使用者把本体也纳入依赖。这就是介于“不依赖”和“依赖”之间的中间状态，就是 peerDependencies 的主要使用场景。

例如我们提供一个包，其中的 package.json 如下：

```
{
  "name": "my-greate-express-middleware",
  "version": "1.0.0",
  "peerDependencies": {
    "express": "≥3.0.0"
  }
}
```

在 npm 3.x 及以后版本，如果使用者安装了我们的插件，并且在自己的项目中没有依赖 express 时，会在最后弹出一句提示，表示有一个包需要您依赖 express 3.x，因此您必须自己额外安装。另外如果使用者依赖了不同版本的 express，npm 也会弹出提示，让开发者自己决定是否继续使用这个包。

bundledDependencies

这是一种比较 peerDependencies 更加少见的依赖项，也可以写作 bundleDependencies（bundle 后面的 d 省略），和上述的依赖不同，这个属性并不是一个键值的对象，而是一个数组，元素为表示包的名字的字符串。例如

```
{
  "name": "awesome-web-framework",
  "version": "1.0.0",
  "bundledDependencies": [
    "renderized", "super-streams"
  ]
}
```

当我们希望以压缩包的方式发布项目时（比如你不想放到 npm registry 里面去），我们会使用 npm pack 来生成（如上述例子，就会生成 awesome-web-framework-1.0.0.tgz）。编写了 bundleDependencies 之后，npm 会把这两个包（renderized, super-streams）也一起加入到压缩包中。这样之后其他使用者执行 npm install awesome-web-framework-1.0.0.tgz 时也会安装这两个依赖了。

如果我们使用常规的 npm publish 的方式来发布的话，这个属性不会生效；而作为使用方的话，大部分项目也都是从 npm registry 中搜索并引用依赖的，所以使用到的场景也相当少。

optionalDependencies

这也是一种很少见的依赖项，从名字可以得知，它描述一种“可选”的依赖。和 dependencies 相比，它的不同点有：

- 即使这个依赖安装失败，也不影响整个安装过程
- 程序应该自己处理安装失败时的情况

关于第二点，我想表达的意思是：

```
let foo
let fooVersion
try {
  foo = require('foo')
  fooVersion = require('foo/package.json').version
} catch (e) {
  // 安装依赖失败时找不到包，需要自己处理
}

// 如果安装的依赖版本不符合实际需求，我们也需要自己处理，当发生安装失败
if (!isSupportVersion(fooVersion)) {
  foo = null
}

// 如果安装成功，执行某些操作
if (foo) {
  foo.doSomething()
}
```

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，可能造成预期之外的效果，因此最好不要出现这种情况。

在实际项目中，如果某个包已经失效，我们通常会寻找它的替代者，或者干脆换一个实施方案。使用这种“不确定”的依赖，一方面会增加代码中的判断，增加逻辑的复杂度；另一方面也会大大降低测试覆盖率，增加构造测试用例的难度。所以我不建议使用这个依赖项，如果你原先就不知道有这个，那就继续当做不知道吧。

版本号写法

如上的 5 种依赖，除了 bundledDependencies，其他四种都是需要写版本号的。如果作为使用者，使用 npm i —save 或者 npm i —save-dev 会自动生成依赖的版本号，不过我建议大家还是稍微了解下常用的版本号的写法。

首先我们得搞清三位版本号的定义，以‘a.b.c’举例，它们的含义是：

a - 主要版本（也叫大版本，major version）

大版本的升级很可能意味着与低版本不兼容的 API 或者用法，是一次颠覆性的升级（想想 webpack 3 > 4）。

b - 次要版本（也叫小版本，minor version）

小版本的升级应当兼容同一个大版本内的 API 和用法，因此应该对开发者透明。所以我们通常只说大版本号，很少会精确到小版本号。

特殊情况是如果大版本号是 0 的话，意味着整个包处于内测状态，所以每个小版本之间也可能会不兼容，所以在选择依赖时，尽量避开大版本号是 0 的包。

c - 补丁（patch）

一般用于修复 bug 或者很细微的变更，也需要保持向前兼容。

之后我们看一下常规的版本号写法：

“1.2.3” - 无更新前的精确版本号

表示只依赖这个版本，任何其他版本号都不匹配。在一些比较重要的线上项目中，我比较建议使用这种方式锁定版本。前阵子的 npm 挖坑以及 anti-design 彩蛋，其实都可以通过锁定版本来规避问题（彩蛋颇难一些，挖坑是肯定可以规避）。

“^1.2.3” - 兼具更新和安全的折中考虑

这是 npm i xxx —save 之后系统生成的默认版本号（^ 加上当前最新版号），官方的定义是“能够兼容除了最左侧的非 0 版本号之外的其他变化”（Allows changes that do not modify the left-most non-zero digit in the [major, minor, patch] tuple）。这句话很晦口，举几个例子大家就明白了：

- “^1.2.3”等价于“>= 1.2.3 < 2.0.0”。即只要最左侧的“1”不变，其他都可以改变。所以“1.2.4”，“1.3.0”都可以兼容。
- “^0.2.3”等价于“>= 0.2.3 < 0.3.0”。因为最左侧的是“0”，所以这个不算，顺延到第二位“2”。那么只要这个“2”不变，其他的都兼容，比如“0.2.4”和“0.2.99”。
- “^0.0.3”等价于“>= 0.0.3 < 0.0.4”。这里最左侧的非 0 只有“3”，且没有其他版本号了，所以这个也等价于精确的“0.0.3”。

从这几个例子可以看出，^ 是一个更新和安全兼容的写法。一般大版本号升级到 1 就表示项目正式发布了，而 0 开头就表示还在测试版，这也是 ^ 区别对待两者的原因。

“~1.2.3” - 比 ^ 更加安全的小版本更新

关于 ~ 的定义分为两部分：如果版本号（第二位），则只兼容 patch（第三位）的修改；如果没有列出小版本号，则兼容第二和第三位的修改。因此它两种情况都理解一下这个定义：

“~1.2.3”列出了小版本号（2），因此只兼容第三位的修改，等价于“>= 1.2.3 < 1.3.0”。

“~1.2”也列出了小版本号，因此和上面一样兼容第三位的修改，等价于“>= 1.2.0 < 1.3.0”。

“~1.2”没有列出小版本号，可以兼容第二第三位的修改，因此等价于“>= 1.0.0 < 2.0.0”

和 ^ 不同的是，~ 并不对 0 或者 1 区别对待，所以“~0”等价于“>= 0.0.0 < 1.0.0”，和“~1”是相同的算法。比较而言，~ 更加谨慎。当首位是 0 并且列出了第二位的时候，两者是等价的，例如 ~0.2.3 和 ^0.2.3。

在 nodejs 的上古版本(v0.10.26, 2014年2月发布的)，npm i —save 默认使用的是 ~，现在已经改成 ^ 了。这个改动也是为了让使用者能最大限度的更新依赖包。

“1.x”或者“1.*” - 使用通配符

这个比起上面那些符号就好理解的多，x（大小写皆可）和的含义相同，都表示可以匹配任何内容。具体来说：

“*”或者“*”（空字符串）表示可以匹配任何版本。

“1.x”，“1.*”和“1”都表示要求大版本是 1，因此等价于“>=1.0.0 < 2.0.0”。

“1.2.x”，“1.2.*”和“1.2”都表示锁定前两位，因此等价于“>= 1.2.0 < 1.3.0”。

因为位于结尾的通配符一般可以省略，而常规也不太可能修正那样把匹配符写在中间，所以大多数情况通配符都可以省略。使用最多的还是匹配所有版本的“这个”。

“1.2.3-beta.2” - 带预发布关键词的，如 alpha, beta, rc, pr 等

先说预发布的定义，我们需要以开发者的角度来考虑这个问题。假设当前线上版本是“1.2.3”，如果我作了一些改动需要发布版本“1.2.4”，但我不想直接上线（因为使用“~1.2.3”或者“^1.2.3”的用户都会直接静默更新），这就需要使用预发布功能。因此我可能会发布“1.2.4-alpha.1”或者“1.2.4-beta.1”等等。

理解了它诞生的初衷，之后的使用就很自然了。

“>1.2.4-alpha.1”，表示我接受“1.2.4”版本所有大于 1.2.4 的 alpha 预发布版本，因此如“1.2.4-alpha.7”是符合要求的，但“1.2.4-beta.1”和“1.2.5-alpha.2”都不符合。此外如果是正式版本（不带预发布关键词），只要版本号符合要求即可，不检查预发布版本号，例如“1.2.5”，“1.3.0”都是认可的。

“~1.2.4-alpha.1”表示“>=1.2.4-alpha.1 < 1.3.0”。这样“1.2.5”，“1.2.4-alpha.2”都符合条件，而“1.2.5-alpha.1”，“1.3.0”不符合。

“^1.2.4-alpha.1”表示“>=1.2.4-alpha.1 < 2.0.0”。这样“1.2.5”，“1.2.4-alpha.2”，“1.3.0”都符合条件，而“1.2.5-alpha.1”，“2.0.0”不符合。

版本号还有更多的写法，例如范围(a - b)，大于小于号(>= a < b)，或表达式[1 || 表达式2]等等，因为用的不多，这里不再展开。详细的文档可以参见 semver，它同时也是一个 npm 包，可以用来比较两个版本号的大小，以及是否符合要求等。

其他写法

除了版本号，依赖包还可以通过如下几种方式进行依赖（使用的也不算太多，可以粗略了解一下）：

Tag

除了版本号之外，通常某个包还可能会有 Tag 来标识一些里程碑意义的版本。例如 express@next 表示即将到来的下一个大版本（可提前体验），而 some-lib@latest 等价于 some-lib，因为 latest 是默认存在并指向最新版本的。其他的自定义 Tag 都可以由开发者通过 npm tag 来指定。

因为 npm i package@version 和 npm i package@tag 的语法是相同的，因此 Tag 和版本号必须不能重复。所以一般建议 Tag 不要以数字或者字母 v 开头。

URL

可以指定 URL 指明依赖包的源地址，通常是一个 tar 包，例如“https://some.site.com/lib.tar.gz”。这个 tar 包通常是随 npm pack 来发布的。

顺带提一句：本质上，npm 的所有包都是以 tar 包发布的。使用 npm 常规发布的包也是被 npm 冠上版本号等后缀，由 npm registry 托管供大家下载的。

Git URL

可以指定一个 Git 地址（不单纯指 GitHub，任何 git 协议的均可），npm 自动从该地址下载并安装。这里就需要指明协议，用户名，密码，路径，分支名和版本号等，比较复杂。详情可以查看官方文档，举例如下：

```
git+ssh://git@github.com:npm/cli.git#v1.0.2
git+ssh://git@github.com:npm/cli#commit=5.0
git+https://isaacs@github.com:npm/cli.git
git+https://github.com/npm/cli.git#v1.0.2
```

作为最大的 Git 代码库，如果使用的是 GitHub 存放代码，还可以直接使用 use!repo 的简写方式，例如：

```
{
  "dependencies": {
    "express": "expressjs/express",
    "mocha": "mochajs/mocha#4723d357ee",
    "module": "mochajs/repo#feature/vbranch"
  }
}
```

本地路径

npm 支持使用本地路径来指向一个依赖包，这时候需要在路径之前添加 file:，例如：

```
{
  "dependencies": {
    "bar1": "file:../foo/bar1",
    "bar2": "file:~/foo/bar2",
    "bar3": "file:/foo/bar3"
  }
}
```

package-lock.json

从 npm 5.x 开始，在执行 npm i 之后，会在根目录额外生成一个 package-lock.json，既然讲到了依赖，我就额外扩展一下这个 package-lock.json 的结构和作用。

package-lock.json 内部记录的是每一个依赖的实际安装信息，例如名字，安装版本号，安装的地址（npm registry 上的 tar 包地址）等等，额外的，它会把依赖的依赖也记录起来，因此整个文件是一个树形结构，保存依赖嵌套关系（类似以前版本的 node_modules 目录）。一个简单的例子如下：

```
{
  "name": "my-lib",
  "version": "1.0.0",
  "lockfileVersion": 1,
  "requires": true,
  "dependencies": {
    "array-union": {
      "version": "1.0.2",
      "resolved": "http://registry.npm.taobao.org/array-union/download/array-union-1.0.2.tgz",
      "integrity": "sha1-wj80v9wKnDz5Sew8k47D0kc=",
      "dev": true,
      "requires": {
        "array-uniq": "1.0.1"
      }
    }
  }
}
```

在执行 npm i 的时候，如果发现根目录下只有 package.json 存在（这通常发生在刚创建项目时），就按照它的记录逐层递归安装依赖，并生成一个 package-lock.json 文件。如果发现根目录下两者皆有（这通常发生在开发同事把项目 checkout 到本地之后），则 npm 会比较两者。如果两者所示含义不同，则以 package.json 为准，并更新 package-lock.json；否则就直接按 package-lock 所示的版本号安装。

它存在的意义主要有 4 点：

- 在团队开发中，确保每个团队成员安装的依赖版本是一致的。否则因为依赖版本不一致导致的效果差异，一般很难查出来源。
- 通常 node_modules 目录都是会交代的，因此要回溯到某一天的状态是不可能的。但现在在 node_modules 目录和 package-lock.json 是一一对应的。所以如果开发者想回到之前某一天的目录状态，只要把这两个文件回溯到那一天 package 的，再 npm i 就行了。
- 因为 package-lock.json 已经足以描述 node_modules 的大概信息（尤其是深层嵌套依赖），所以通过这个文件就可以查阅某个依赖包是被谁依赖进来的，而不用去翻 node_modules 目录（事实上现在目录结构打平而非嵌套，翻也翻不出来了）
- 在安装过程中，npm 内部会检查 node_modules 目录中已有的依赖包，并和 package-lock.json 进行比较。如果重复，则跳过安装，在能大大优化安装时间。

npm 官网建议：把 package-lock.json 一起提交到代码库中，不要 ignore。但是在执行 npm publish 的时候，它会被忽略而不会发布出去。

yarn

从 nodejs 诞生之初，npm 就是其内置的包管理器，并且以其易于使用，易于发布的特点极大地助推了 nodejs 在开发者中的流行和使用，但事物总有其两面性，易于发布的确实大推动生态的繁荣，但同时也降低了发布的门槛。包的数量在突飞猛进，一个项目的依赖项从几十个上升到几十个，再加上内部的嵌套循环依赖，给使用者带来了极大的麻烦，node_modules 目录越来越大，npm install 的时间也越来越长。

在这种情况下，Facebook 率先站出来，发布了由他们开发的另一个包管理器 yarn（1.0版本于2017年9月）。一旦有了挑战者出现，势必会引发双方对于功能、稳定性，易于发布等诸多方面的竞争，对于开发者来说也是极其有利的。从结果来说，npm 也吸取了来自 yarn 借鉴来的优点，例如上面谈论的 package-lock.json，最早就出自 yarn.lock，所以我们来粗略比较一下两者的区别，以及我们应当如何选择。

版本锁定

这个在 package-lock.json 已经讨论过了，不再赘述。在这个功能点上，两者都已具备。

多个包的管理（monorepositories）

一个包在 npm 中可以被称为 repositories。通常我们发布某个功能，其实就是发布一个包，由它提供各种 API 来提供功能。但随着功能越来越复杂以及按需加载，把所有东西全部放到一个包中发布已经不够优秀，因此出现了多个包管理的需求。

通常一个包会把自己的功能拆分为核心功能和其他部分，然后每个部分是一个 repositories。而使用者通常在使用核心之后，可以自己选择要使用哪些依赖的该种方式比较常见，如 babel 和它的插件，express 和它的中间件等。

作为一个多个包的项目的开发者/维护者，安装依赖和发布都会是一件很麻烦的事情，因为 npm 只认根目录的 package.json，那么就必须要进入每个包进行 npm install。而发布时，也必须逐个修改每个包的版本号，并到每个目录中进行 npm publish。

为了解决这个问题，社区一个叫做 lerna 的库通过增加 lerna.js 来帮助管理所有的包，而在 yarn 这边，引入了一个叫做工作区(workspace)的概念。因此这点上来说，应该是 yarn 胜出了，不过 npm 配合 lerna 也能够实现这个需求。

安装速度

npm 被诟病最多问题之一就是其安装速度。有些依赖很多的项目，安装 npm 需要耗费 5-10 分钟甚至更长。造成这个问题的本质是 npm 采用串行的安装方式，一个包安装完下一个。针对这一点，npm 改为并行安装，因此本质上提升了安装速度。

离线可用

yarn 默认支持离线安装，即安装过一次的包，会在电脑中保留一份（缓存位置可以通过 yarn config set yarn-offline-mirror 进行指定）。之后再次安装，直接复制过来就可以。

npm 早先完全通过网络请求的（为了保持其时效性），但后期也借鉴了 yarn 创建了缓存。从 npm 5.x 开始我们可以使用 npm install xxx —prefer-offline 来优先使用缓存（意思是缓存没有再发送网络请求），或者 npm install xxx —offline 来完全使用缓存（意思是缓存没有就安装失败）。

控制台信息

常年使用 npm 的同学知道，安装完依赖后，npm 会列出一颗依赖树。这颗树通常会很长很复杂，我们不会过多关注，因此 yarn 精简了这部分信息，直接输出安装结果。这样万一安装过程中有报错日志也不至于被刷屏。

不过 npm 5.x 也把这颗树给去掉了。这又是一个互相借鉴提高的例子。

总结来说，yarn 的推出主要是针对 npm 早期版本的很多问题。但 npm 也意识到了来自竞争对手的强大压力，因此在 5.x 开始逐步优化着。从 5.x 开始就已经和 yarn 不分伯仲了，因此如何选择多看是否有历史包袱。如果是新项目的话，就看程序员个人的喜好了。

后记

本文从一个很小的问题开始，本意是想分享如何鉴别一个应用应该归类在 dependencies 还是 devDependencies，后来层层深入，通过查阅资料发现了好多依赖相关的知识，例如其他几种依赖，版本锁定的机制以及和 yarn 的比较等等，最终变成一篇长文。希望通过本文能让大家了解到依赖管理的一些大概，在之后的艰难道路上能够更加顺利，也能反过来为整个生态的繁荣贡献自己的力量。

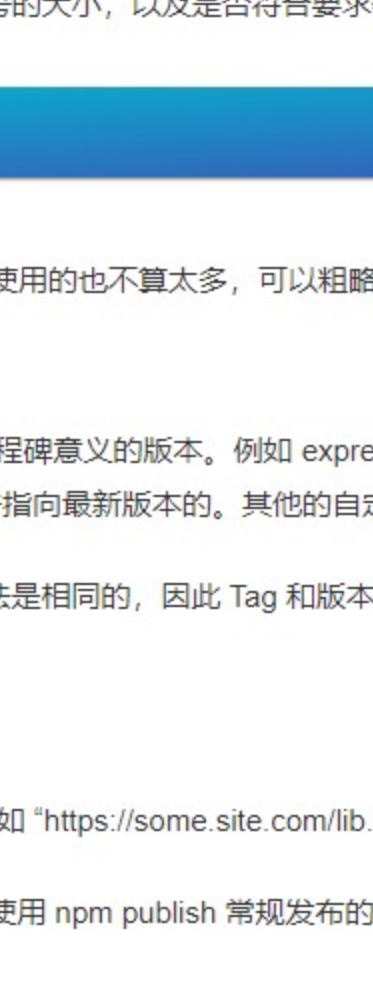
参考文章

- npm 官网的 dependencies 文档
- npm 官方微博的 peerDependencies 介绍 - 这篇有点老了，npm 依赖还是嵌套关系
- Why use peerDependencies in npm for plugins - 比较简略，不过说的在点上
- Types of dependencies - 虽然是 yarn 的介绍，但概念和 npm 一致，且很精准
- semver - npm 官方用来比较版本号的包
- npm install —save" No Longer Using Tildes - 早期的一篇博客，npm 对依赖版本号默认处理的变更
- npm 官网的 package-lock.json 文档
- Workspaces in Yarn - yarn 官网介绍的 workspace 功能
- Here's what you need to know about npm 5 - 介绍 npm 5.x 的重要改进点

●编号919，输入编号直达本文

●输入m获取文章目录

推荐↓↓↓



Web开发

更多推荐 **《25个技术类微信公众号》**

涵盖：程序人生、算法与数据结构、黑客技术与网络安全、大数据技术、前端开发、Java、Python、Web开发、安卓开发、iOS开发、C/C++、.NET、Linux、数据库、运维等。

阅读原文