# 禁止npm自动升级依赖包以及所有下级依赖包版本的方法

码西西 (/u/324439d9a332) + 关注
2017.09.12 10:58* 字数 1950 阅读 125 评论 0 喜欢 2
(/u/324439d9a332)

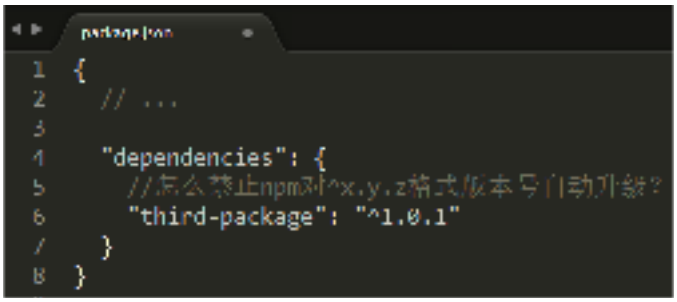
> 虽然多数npm依赖包开发者能严格遵守SemVer版本规范，但有时在实际项目中为了可靠，我们不希望未经审核评估就轻易升级一个项目依赖包，以免带来意外的风险，尤其是依赖包下更多第三方依赖包往往也指定是“^x.y.z”格式的版本。我们可以通过修改npm源码来禁止其在安装依赖包过程中匹配最新版本。

标签： npm SemVer 不要自动升级 ^版本符号

-2017.9.12-



源文链接：http://www.jianshu.com/p/f481cf9b0817
(https://www.jianshu.com/p/f481cf9b0817)




package.json

## 一、背景

基于npm管理的项目根目录下通常会有一个package.json文件，简单如下：

```
{
  "name": "ism-client",
  "version": "0.2.18",
  "private": true,
  "author": "ism",
  "license": "ISC",
  "dependencies": {
    "global-modules": "^0.2.1"
  }
}
```

里面包含的dependencies、devDependencies等节点用于定义项目需要的依赖包及版本。而这些下级依赖包也会有自己的package.json，定义了自己需要的依赖包及版本，如此可以一级级地依赖下去。依赖包可以是自己开发的，也可以是第三方开发的，如npm的官方仓库 (https://link.jianshu.com?t=https://www.npmjs.com/search?q=global-modules&page=1&ranking=optimal)中就提供了大量的第三方依赖包。官方仓库中的每个依赖包一般会有一个**依赖包信息文件**，里面包含了该依赖包的**最新版本**、**可用版本**以

及包下载地址。 通过类似https://registry.npmjs.org/global-modules (https://link.jianshu.com?t=https://registry.npmjs.org/global-modules)的URL可以访问到：


```
{
    ...
    "name": "global-modules",
    ...
    "dist-tags":
    {
        "latest": "1.0.0"
    },
    ...
    "versions":
    {
        "0.1.0":
        {
            ...
            "dist":
            {
                "shasum": "9e8011fdede4f811047f76b155495e7786d27e02",
                "tarball": "https://registry.npmjs.org/global-modules/-/global-modul
            },
            ...
        },
        "0.2.0":
        {
            ...
            "dist":
            {
                "shasum": "efb4f2d0c9a6856c106f1bba3bc0568e59c197b1",
                "tarball": "https://registry.npmjs.org/global-modules/-/global-modul
            },
            ...
        },
        "0.2.1":
        {
            ...
            "dist":
            {
                "shasum": "c2720c3b36a4565d21b08a5322f8edd46a9e4d38",
                "tarball": "https://registry.npmjs.org/global-modules/-/global-modul
            },
            ...
        },
        "0.2.2":
        {
            ...
            "dist":
            {
                "shasum": "c7e589646bf8bec457d71049553adc72e36c6346",
                "tarball": "https://registry.npmjs.org/global-modules/-/global-modul
            },
            ...
        },
        "0.2.3":
        {
            ...
            "dist":
            {
                "shasum": "ea5a3bed42c6d6ce995a4f8a1269b5dae223828d",
                "tarball": "https://registry.npmjs.org/global-modules/-/global-modul
            },
            ...
        },
        "1.0.0":
        {
            ...
            "dist":
            {
                "integrity": "sha512-sKzpEkf11GpOFuw0Zzjzmt4B4UZwjOcG757PPvrfhxcLFbc
                "shasum": "6d770f0eb523ac78164d72b5e71a8877265cc3ea",
                "tarball": "https://registry.npmjs.org/global-modules/-/global-modul
            },
            ...
        }
    },
    ...
}
```

在项目根目录下执行npm install命令即可安装项目依赖包及其更多下级依赖包。安装过程中，npm会根据SemVer (https://link.jianshu.com?t=http://semver.org/lang/zh-CN/)版本规范，自动从各依赖库的版本清单地址中匹配尽量新的版本，然后下载并安装到node_modules子目录。如上面package.json中的“global-modules”依赖包的版本为“**^0.2.1**”，根据SemVer版本规范最终会在依赖信息文件的中匹配到最新的“**0.2.3**”版本。

虽然多数依赖包开发者能严格遵守SemVer版本规范，但有时在实际项目中为了可靠，我们不希望未经审核评估就轻易升级一个项目依赖包，以免带来意外的风险，尤其是依赖包下更多第三方依赖包往往也指定是“**^x.y.z**”格式的版本。著名的left-pad事件 (https://link.jianshu.com?t=https://zhuanlan.zhihu.com/p/20707235)就因为一个第三方依赖包的撤回，导致依赖它的React、Babel、Ember依赖包无法安装，继而影响大量使用这些包的项目。这两天我也因为项目中随意指定angular版本为**"angular":"^1.5.11"**，npm install命令实际安装后的版本是**"angular":"1.6.6"**，但这个版本中将导航书签地址的默认前缀从空字符串""变成了"!" (https://link.jianshu.com?t=https://github.com/angular/angular.js/commit/aa077e81129c740041438688dff2e8d20c3d7b52)，导致项目中原有的导航路径无法解析页面无法显示，调试控制台也没有任何错误信息，耗费了大量精力才定位出问题。

虽然可以通过npm的package-lock.js、shrinkwrap.json，或者通过yarn.lock等锁定文件实现版本号锁定，但它们只是在一个包被首次安装时把**当时实际匹配到的版本号**写入到锁定文件中。也就是说，如果一个子依赖包的版本号是“**^0.2.1**”，往往它被写入到锁定文件中的不是“**0.2.1**”，而是当时匹配的最新版本如“**0.2.3**”。想想实际发布包的过程中，这个“**^0.2.1**”是怎么来的呢？它实际时作者在开发时通过npm install xxx命令安装时自动生成的，然后集成调用这个版本的子依赖包进行大理的开发、测试，然后发布了自己的包。所以这个子依赖包的“**0.2.1**”版本是作者在发布自己的包过程中实际使用的版本，对作者这次发布的包来说，这个子依赖包的版本是稳定可靠的，我们应该优先使用，我们可以修改npm，禁止其根据SemVer规则自动升级该版本。

## 二、解决方法

修改npm源码，禁止npm根据SemVer规则自动升级依赖包及其下级依赖包的版本。

### 1. 升级npm到5.0以上

不升级也可以，但后面第4、第5步对应修改点的位置可能不一样，需要自行分析对应的修改点。另外，npm 5.4.1版本容易出现权限不足无法创建文件链接、无法删除目录之类的错误 (https://link.jianshu.com?t=https://github.com/npm/npm/issues/18287)，最好不要使用。

```
D:\project\mynpm>npm install npm@5.3.0 -g
```

### 2. 测试1（npm会自动升级依赖包版本号）

(1) 新建目录 `d:\project\mynpm`
(2) 添加文件 `d:\project\mynpm\package.json`

```
{
  "name": "ism-client",
  "version": "0.2.18",
  "private": true,
  "scripts": {
    "start": "pm2 startOrGracefulReload ./pm2.json --no-daemon",
    "production": "pm2 startOrGracefulReload ./pm2.json --no-daemon",
    "debug": "npm build && node dist/app.js",
    "build": "gulp build",
    "test": "gulp build --test"
  },
  "author": "ism",
  "license": "ISC",
  "dependencies": {
    "global-modules": "^0.2.1"
  }
}
```

(3) 添加 `d:\project\mynpm\npm-test.bat` ，用于清除安装结果并再次安装。也可以不使用批处理文件，直接使用npm install命令，加--verbose是为了看到更详细的安装过程。

```
rd /s/q node_modules
del /s/f/q package-lock.json
cls
call npm install --verbose
```

(4) cmd中执行批处理文件安装依赖包

```
D:\project\mynpm>npm-test.bat
安装过程略...
```

(5) 通过package-lock.json或者node_modules目录下各依赖包的package.json，观察定义的版本和实际安装的版本。此时（2017-9-11）实际安装了8个依赖包，以下例举部分。可以发现所有 `^x.y.z` 格式定义的依赖包和子依赖包版本都被npm自动升级了：

```
项目本身:
    package.json定义的依赖:
    "global-modules": "^0.2.1" //实际安装版本0.2.3（被升级了）

node_modules/global-modules (0.2.3):
    package.json定义的依赖:
    "global-prefix": "^0.1.1", //实际安装版本0.1.5（被升级了）
    "is-windows": "^0.1.1" //实际安装版本0.2.0（被升级了）

node_modules/global-prefix (0.1.5):
    package.json定义的依赖:
    "homedir-polyfill": "1.0.1", //实际安装版本1.0.1
    "ini": "1.3.4", //实际安装版本1.3.4
    "is-windows": "0.2.0", //实际安装版本0.2.0
    "which": "1.3.0" //实际安装版本1.3.0

node_modules/is-windows (0.2.0):
    package.json定义的依赖:
    无依赖

其它5个依赖包...
```

## 3. 找到npm的安装目录

其中 `prefix` 后面的 `C:\Users\administrator\AppData\Roaming\npm` 就是你npm的安装位置，打开该文件夹后就可以看到npm安装后的源码了。

```
D:\project\mynpm>npm config list
; cli configs
metrics-registry = "https://registry.npmjs.org/"
scope = ""
user-agent = "npm/5.3.0 node/v7.8.0 win32 x64"

; builtin config undefined
prefix = "C:\\Users\\administrator\\AppData\\Roaming\\npm"

; node bin location = C:\Program Files\nodejs\node.exe
; cwd = D:\project\mynpm
; HOME = C:\Users\administrator
; "npm config ls -l" to show all defaults.
```

## 4. 修改npm安装目录中的源代码

说是修改npm的源码，其实升级版本的逻辑在npm下下级依赖包npm-pick-manifest中也有。npm整个代码量非常大，依赖非常多，要定位到这些逻辑真是困难！你果你必须用较早前的npm版本，可以把这里的修改作为参考，自行定位修改点。

(1) 修改npm-pick-manifest包的index.js文件

文件路径：

`C:\Users\administrator\AppData\Roaming\npm\node_modules\npm\node_modules\pacote\node_modules\npm-pick-manifest\index.js`

```
function pickManifest (packument, wanted, opts) {
  ...

  /* --禁止自动升级版本
  if (
    !target &&
    tagVersion &&
    packument.versions[tagVersion] &&
    semver.satisfies(tagVersion, wanted, true)
  ) {
    target = tagVersion
  }

  if (!target) {
    target = semver.maxSatisfying(versions, wanted, true)
  }
  */
  // ++禁止自动升级版本
  if (!target) {
    target = semver.minSatisfying(versions, wanted, true)
  }

  ...
}
```

(2) 修改npm包的pick-manifest-from-registry-metadata.js文件

文件路径：

`C:\Users\administrator\AppData\Roaming\npm\node_modules\npm\lib\utils\pick-manifest-from-registry-metadata.js`

```
function pickManifestFromRegistryMetadata (spec, tag, versions, metadata) {
  ...

  /* --禁止自动升级版本
  // if the tagged version satisfies, then use that.
  var tagged = metadata['dist-tags'][tag]
  if (tagged &&
      metadata.versions[tagged] &&
      semver.satisfies(tagged, spec, true)) {
    return {resolvedTo: tag, manifest: metadata.versions[tagged]}
  }
  // find the max satisfying version.
  var ms = semver.maxSatisfying(versions, spec, true)
  */
  // ++禁止自动升级版本
  var ms = semver.minSatisfying(versions, spec, true)

  if (ms) {
    return {resolvedTo: ms, manifest: metadata.versions[ms]}
  } else if (spec === '*' && versions.length && tagged && metadata.versions[tagged]) {
    return {resolvedTo: tag, manifest: metadata.versions[tagged]}
  } else {
    return
  }
}
```

## 6. 测试2（禁止npm自动升级依赖包版本号后）

(1) 复制 `d:\project\mynpm` 中的 `package.json` 、 `npm-test.bat` 两个文件到 `d:\project\mynpm2` 目录

(1) cmd中跳转到mynpm2执行批处理文件安装依赖包

```
D:\project\mynpm>cd ../mynpm2
D:\project\mynpm2>npm-test.bat
安装过程略...
```

(2) 通过package-lock.json或者node_modules目录下各依赖包的package.json，观察定义的版本和实际安装的版本。现在，所有依赖包以及依赖包下的依赖的版本都没有被自动升级了，相当于忽略了SemVer版本规范的 ^ 标识符号，直接安装的是各个包作者发布版本时使用的依赖包版本。总共安装了3个依赖包（早期版本依赖较少）：

```
项目本身:
    package.json定义的依赖:
    "global-modules": "^0.2.1" //实际安装版本0.2.1

node_modules/global-modules (0.2.1):
    package.json定义的依赖:
    "global-prefix": "^0.1.1", //实际安装版本0.1.1
    "is-windows": "^0.1.1" //实际安装版本0.1.1

node_modules/global-prefix (0.1.1):
    package.json定义的依赖:
    "is-windows": "0.1.1", //实际安装版本0.1.1

node_modules/is-windows (0.1.1):
    package.json定义的依赖:
    无依赖
```



码西西 (/u/324439d9a332)

写了 2180 字，被 3 人关注，获得了 2 个喜欢