使用 npm shrinkwrap 来管理项目依赖

敬威 · 2015-10-23 17:30

管理依赖是一个复杂软件开发过程中必定会遇到的问题。

在Node.js项目开发的时候，我们也经常需要安装和升级对应的依赖。虽然 npm 以及语意化的版本号 (semantic versioning, semver) 让开发过程中依赖的获取和升级变得非常容易， 但不严格的版本号限制，也带来了版本号的不确定性。主要的问题可能有三个：

1. npm 建议使用 semver 的应用程序版本，但这也完全依赖第三方包遵守这一规则。如果你依赖于的包不遵循 semver ，或者依赖的包的新版本有重大更改（而你使用了 `^` 的宽泛版本安装），这潜在可能是会导致问题的。
2. 另一个问题的出现是由于 npm 安装依赖的机制。npm 的安装包是有层次结构的，手动控制要安装的软件包的版本号可以实现，但是你只能在 package.json 使用精确的版本号控制你的直接依赖包，但那些多层以上的依赖就没办法控制了；一个第三方包不严谨的版本依赖生命可能破坏你的依赖管理。
3. 在开发阶段执行得到的版本，和后续部署时得到的可能是不一致的，更不可控的是，你依赖的第三方包也有这样的情况会导致潜在的上线风险。

如果要控制上线的风险，我们就必需要解决这个问题，这时候，就需要使用 `npm shrinkwrap` 这个命令来解决问题。

介绍 shrinkwrap

`npm shrinkwrap` 可以按照当前项目 node_modules 目录内的安装包情况生成稳定的版本号描述。

比方说，有一个包 A

```
{
  "name": "A",
  "version": "0.1.0",
  "dependencies": {
    "B": "<0.1.0"
  }
}
```

还有一个包 B

```
    {
      "name": "B",
      "version": "0.0.1",
      "dependencies": {
        "C": "<0.1.0"
      }
    }
```

以及包 C

```
    {
      "name": "C",
      "version": "0.0.1"
    }
```

你的项目只依赖于 A，于是 `npm install` 会得到这样的目录结构

```
  A@0.1.0
    `-- B@0.0.1
        `-- C@0.0.1
```

这时候，B@0.0.2 发布了，这时候在一个新的环境下执行 `npm install` 将得到

```
A@0.1.0
    `-- B@0.0.2
        `-- C@0.0.1
```

这时候两次安装得到的版本号就不一致了。而通过 shrinkwrap 命令，我们可以保证在所有环境下安装得到稳定的结果。

在项目引入新包的时候，或者 A 的开发者执行一下 `npm shrinkwrap`，可以在项目根目录得到一个 npm-shrinkwrap.json 文件。

这个文件内容如下

```
{
  "name": "A",
  "version": "0.1.0",
  "dependencies": {
    "B": {
      "version": "0.0.1",
      "dependencies": {
        "C": {
          "version": "0.0.1"
        }
      }
    }
  }
}
```

shrinkwrap 命令根据目前安装在node_modules的文件情况锁定依赖版本。在项目中执行 `npm install` 的时候，npm 会检查在根目录下有没有 npm-shrinkwrap.json 文件，如果 shrinkwrap 文件存在的话，npm 会使用它（而不是 package.json）来确定安装的各个包的版本号信息。

这样一来，在安装时候确定的所有版本信息会稳定的固化在 shrinkwrap 里。无论是A，B 和 C 中的版本如何变化，或者它们的 package.json 文件如何修改，你始终能保证，在你项目中执行 `npm install` 的到的版本号时稳定的。

在开发中使用 shrinkwrap

在开发过程中，引入一个新包的流程如下

1. `npm install PACKAGE_NAME@VERSION --save` 获取特定版本的包
2. 测试功能
3. 测试功能正常后，执行 `npm shrinkwrap` 把依赖写入 shrinkwrap 文件
4. 在代码仓库中提交 shrinkwrap / package.json 描述

升级一个包的流程应该是这样

1. `npm outdated` 获取项目所有依赖的更新信息
2. `npm install PACKAGE_NAME@VERSION --save` 获取特定版本的包
3. 测试功能
4. 测试功能正常后，执行 `npm shrinkwrap` 把依赖写入 shrinkwrap 文件
5. 在代码仓库中提交 shrinkwrap / package.json 描述

删除一个包的流程如下

1. `npm uninstall PACKAGE_NAME --save` 删除这个包

2. 测试功能
3. 测试功能正常后，执行 `npm shrinkwrap` 把更新的依赖写入 shrinkwrap 文件
4. 在代码仓库中提交 shrinkwrap / package.json 描述

比一般的安装多了一步手工生成 shrinkwrap 文件。在实际工作中，有时候我们会忘记这一步，导致上线时候没有获取到依赖包的特定版本。

介绍 npm-shrinkwrap-install

去年我引入 shrinkwrap 工作流的时候，npm 官方的 shrinkwrap 命令还有很多问题，比如

- 在生成版本描述的时候不会忽略 devDependencies 和 optionalDependencies
- 不会检查 package.json 和 shrinkwrap 文件的差异
- 不会删除 from 字段（这个字段没有用），导致在 diff 时候会出现多余的信息

所以我写了一个 bin/shrinkwrap 脚本 (http://git.sankuai.com/projects/FE/repos/fe-paidui/browse/bin/shrinkwrap)。这个脚本会自动对比 package.json 和 npm-shrinkwrap.json 文件的区别，获取需要更新的版本，然后对相关信息进行更新。(更新：现在的 npm shrinkwrap 已经修复了很多的问题，但 from 字段有时候仍然有些小问题。）

当时为了忽略 devDependencies 和 optionalDependencies，我会执行 `npm prune` 删除额外的包之后才生成版本描述，然后再把其它的包装回来，导致脚本执行时间有点长。

后面 uber 发布了 npm-shrinkwrap 工具 (https://github.com/uber/npm-shrinkwrap)，可以更高效的生成版本描述。可惜这个包不支持 scoped package。我花了一点时间 patch 了这个工具，因为它们的发版太慢，所以我发布了一份 @th507/npm-shrinkwrap (https://www.npmjs.com/package/@th507/npm-shrinkwrap)，可以支持 scoped package。

在上面这个包的基础上，我还写了另外一个小工具，叫做 npm-shrinkwrap-install (https://github.com/th507/npm-shrinkwrap-install)，它可以无缝替换 `npm install` 的执行过程，让 shrinkwrap 文件的生成变得更自动。

安装

```
$ npm install npm-shrinkwrap-install
```

安装完成之后，有如下命令可以使用

安装依赖的命令
npm-install
npm-i

删除依赖的命令

npm-unisntall

npm-un

npm-remove

npm-rm

npm-r

手工生成 shrinkwrap 描述

npm-shrinkwrap

在开发中使用 npm-install

引入新依赖包

1. `npm-install PACKAGE_NAME@VERSION --save` 获取特定版本的包
2. 测试功能
3. 在代码仓库中提交 shrinkwrap / package.json 描述

npm-install 在运行时会对 package.json 中的依赖做校验，如果你直接修改 package.json 文件，或者是指定了一个非严格的版本号，在运行的时候都会做更新检查，防止遗漏。

值得注意的是，因为 npm-install 会进行依赖对比和校验，在安装新包的时候需要带上 --save 参数。否则，在自动更新 shrinkwrap 描述之前，脚本会自动移除多余的依赖包，导致你新安装的包被删除。

升级依赖包

1. `npm outdated` 获取项目所有依赖的更新信息
2. `npm-install PACKAGE_NAME@VERSION --save` 获取特定版本的包
3. 测试功能
4. 在代码仓库中提交 shrinkwrap / package.json 描述

package.json 文件中指定了一个非严格的版本号的依赖在运行 npm-install 的时候会做自动更新检查，无需指定版本号，如果你不希望进行自动更新，请在 package.json 中使用严格版本号。

删除依赖包

1. `npm-uninstall PACKAGE_NAME --save` 删除这个包
2. 测试功能
3. 在代码仓库中提交 shrinkwrap / package.json 描述

可以看到，在实际使用中没有引入额外的流程，对开发者基本没有学习的负担。但仍然注意，不建议开发者执行 `npm update` 命令更新所有的依赖。

小结

通过引入 shrinkwrap 文件，我们可以较好的管理项目的依赖关系，让上线变得更轻松。需要注意的是，尽管相关工具可以帮助你减化工作流程、可靠的分发依赖描述，但工具不能取代功能测试；每次升级依赖版本之后，我们仍然应该进行相关测试来确保项目能可靠的运行在该环境中。

如果使用 @th507/npm-shrinkwrap (https://www.npmjs.com/package/@th507/npm-shrinkwrap) 或者 npm-shrinkwrap-install (https://github.com/th507/npm-shrinkwrap-install) 有任何问题欢迎给我提 issue。

本文没有涉及如何优化上线前的安装过程，缩短依赖构建时间，这个问题留给专门的文介绍。

node iojs dependencies dependency shrinkwrap install