使用 npm shrinkwrap 来管理项目依赖

http://tech.meituan.com/npm-shrinkwrap.html
https://nodejs.org/en/blog/npm/managing-node-js-dependencies-with-shrinkwrap/

```
NAME
       npm-shrinkwrap -- Lock down dependency versions

SYNOPSIS
       npm shrinkwrap

DESCRIPTION
       This  command  locks down the versions of a package's dependencies so
       that you can control exactly which versions of each  dependency  will
       be used when your package is installed.
```

https://docs.npmjs.com/cli/shrinkwrap

Since **`npm shrinkwrap`** is intended to lock down your dependencies for production use, **`devDependencies`** will not be included unless you explicitly set the **`--dev`** flag when you run **`npm shrinkwrap`**. If installed **`devDependencies`** are excluded, then npm will print a warning. If you want them to be installed with your module by default, please consider adding them to **`dependencies`** instead.

npm install npm-shrinkwrap —save-dev
npm shrinkwrap
这样就在package.json的目录下生成一个：npm-shrinkwrap.json
在生成它之前，可能会提示，

```
npm WARN shrinkwrap   winston: ^2.2.0" }
npm ERR! Linux 2.6.32-573.el6.x86_64
npm ERR! argv "/usr/local/bin/node" "/usr/local/bin/npm" "shrink"
npm ERR! node v4.4.4
npm ERR! npm  v3.9.2

npm ERR! Problems were encountered
npm ERR! Please correct and try again.
npm ERR! extraneous: pend@1.2.0 /home/lizhishan/gitlab/cms-be/node_modules/pend
npm ERR!
npm ERR! If you need help, you may report this error at:
npm ERR!     <https://github.com/npm/npm/issues>

npm ERR! Please include the following file with any support request:
```

这是因为，该包虽然被安装了，但其实并没有被project使用. 这时，可以将其删掉。

比如，

更新一个包：
生产环境下生成的npm-shrinkwrap中
mysql 是2.10.2

但使用
npm install mysql@latest --save

```
lizhishans-MacBook-Pro:cms-be lizhishan$ npm install mysql@latest --save
npm WARN shrinkwrap Excluding devDependency: npm-shrinkwrap@200.5.1 { async: '^2.0.0-rc.3',
npm WARN shrinkwrap   'aws-sdk': '^2.3.16',
npm WARN shrinkwrap   'body-parser': '~1.13.2',
npm WARN shrinkwrap   compression: '^1.6.2',
npm WARN shrinkwrap   'cookie-parser': '~1.3.5',
npm WARN shrinkwrap   cors: '^2.7.1',
npm WARN shrinkwrap   dateformat: '^1.0.12',
npm WARN shrinkwrap   debug: '~2.2.0',
npm WARN shrinkwrap   ejs: '~2.3.3',
npm WARN shrinkwrap   'excel-export': '^0.5.1',
npm WARN shrinkwrap   express: '~4.13.1',
npm WARN shrinkwrap   extend: '^3.0.0',
npm WARN shrinkwrap   formidable: '^1.0.17',
npm WARN shrinkwrap   guid: '0.0.12',
npm WARN shrinkwrap   jsonschema: '^1.1.0',
npm WARN shrinkwrap   jsonwebtoken: '^5.7.0',
npm WARN shrinkwrap   ldapjs: '^1.0.0',
npm WARN shrinkwrap   lwip: '0.0.9',
npm WARN shrinkwrap   'mime-types': '^2.1.11',
npm WARN shrinkwrap   morgan: '~1.6.1',
npm WARN shrinkwrap   ms: '^0.7.1',
npm WARN shrinkwrap   mysql: '^2.11.1',
npm WARN shrinkwrap   'node-uuid': '^1.4.7',
npm WARN shrinkwrap   path: '^0.12.7',
npm WARN shrinkwrap   querystring: '^0.2.0',
npm WARN shrinkwrap   request: '^2.72.0',
npm WARN shrinkwrap   'request-ip': '^1.2.2',
npm WARN shrinkwrap   'serve-favicon': '~2.3.0',
npm WARN shrinkwrap   'socks5-http-client': '^1.0.2',
npm WARN shrinkwrap   'sprintf-js': '^1.0.3',
npm WARN shrinkwrap   uuidv5: '0.0.0',
npm WARN shrinkwrap   validator: '^5.2.0',
npm WARN shrinkwrap   winston: '^2.2.0' }
cms-be@ /private/tmp/cms-be
└─┬ mysql@2.11.1
  ├── bignumber.js@2.3.0
  └── sqlstring@2.0.1
```

这样mysql就被更新了。

```
lizhishans-MacBook-Pro:cms-be lizhishan$ git diff package.json
diff --git a/package.json b/package.json
index dfcb096..65ddef5 100644
--- a/package.json
+++ b/package.json
@@ -27,7 +27,7 @@
     "mime-types": "^2.1.11",
     "morgan": "~1.6.1",
     "ms": "^0.7.1",
-    "mysql": "^2.10.2",
+    "mysql": "^2.11.1",
     "node-uuid": "^1.4.7",
     "path": "^0.12.7",
     "querystring": "^0.2.0",
```

git diff npm-shrinkwrap.json

```
    "mysql": {
-     "version": "2.10.2",
-     "from": "mysql@>=2.10.2 <3.0.0",
-     "resolved": "https://registry.npmjs.org/mysql/-/mysql-2.10.2.tgz"
+     "version": "2.11.1",
+     "from": "mysql@latest",
+     "resolved": "https://registry.npmjs.org/mysql/-/mysql-2.11.1.tgz"
    },
    "nan": {
```

所以，版本固定为问题解决了，要更新package的问题也解决了。

新增一个包：
npm install eslint --save-dev
它会将其同时保存到package.json,nom-shrink.json两个文件中。

删除一个包：
npm uninstall aws-sdk --save
它同样会更新这两个描述性文件中。

为甚bower_component可以check－in，而node_module缺不能?
比如native的package：lwip

Why not just check `node_modules` into git?

One previously proposed solution is to "npm install" your dependencies during development and commit the results into source control. Then you deploy your app from a specific git SHA knowing you've got exactly the same bits that you tested in development. This does address the problem, but it has its own issues: for one, binaries are tricky because you need to "npm install" them to get their sources, but this builds the [system-dependent] binary too. You can avoid checking in the binaries and use "npm rebuild" at build time, but we've had a lot of difficulty trying to do this.[2] At best, this is second-class treatment for binary modules, which are critical for many important types of Node applications.[3]

Besides the issues with binary modules, this approach just felt wrong to many of us. There's a reason we don't check binaries into source control, and it's not just because they're platform-dependent. (After all, we could build and check in binaries for all supported platforms and operating systems.) It's because that approach is error-prone and redundant: error-prone because it introduces a new human failure mode where someone checks in a source change but doesn't regenerate all the binaries, and redundant because the binaries can always be built from the sources alone. An important principle of software version control is that you don't check in files derived directly from other files by a simple transformation.[4] Instead, you check in the original sources and automate the transformations via the build process.

Dependencies are just like binaries in this regard: they're files derived from a simple transformation of something else that is (or could easily be) already available: the name and version of the dependency. Checking them in has all the same problems as checking in binaries: people could update package.json without updating the checked-in module (or vice versa). Besides that, adding new dependencies has to be done by hand, introducing more opportunities for error (checking in the wrong files, not checking in certain files, inadvertently changing files, and so on). Our feeling was: why check in this whole dependency tree (and create a mess for binary add-ons) when we could just check in the package name and version and have the build process do the rest?

Finally, the approach of checking in node_modules doesn't really scale for us. We've got at least a dozen repos that will use restify, and it doesn't make sense to check that in everywhere when we could instead just specify which version each one is using. There's another principle at work here, which is **separation of concerns**: each repo specifies *what* it needs, while the build process figures out *where to get it*.

Checking in front-end dependencies

https://addyosmani.com/blog/checking-in-front-end-dependencies/

http://javascript.tutorialhorizon.com/2015/03/21/what-is-npm-shrinkwrap-and-when-is-it-needed/

There are 2 main problems with the way npm install works

1. Although npm recommends using semver for application versioning, it is completely upto the package author to honor this rule. This can be problematic if the package you depend on does not follow semver and a newer version of the package has breaking changes.
   Even if the package author follows semver, there is still a probability that a bug might get introduced in a compatible version.
2. The other issue arises due to the way npm install works. Since running an npm install install a hierarchy of packages to be installed, if you wished to manually control the version numbers of the packages that you want to be installed, you could do that by using the exact version numbers in your package.json. However that only solves the problem for the direct dependents of your package. It does not give you control over the installed versions of the deeply nested packages that are the dependencies of your dependencies and beyond.

This can be crucial to you in a production environment because you need to ensure that each production re-deployment always always installs the same versions of the package as the other deployments.

This is where `npm shrinkwrap` comes into play. When you run npm shrinkwrap in a project after running `npm install`, it creates a file called `npm-shrinkwrap.json` which lists the exact package versions of all the installed packages in the entire hierarchy. If you check this into your version control and your collegue clones and does an `npm install`, then this time they will get the exact package version for the full hierarchy as specified in the `npm-shrinkwrap.json` file.

In order to update your `npm-shrinkwrap.json` file, you would need to run `npm update <package_name>`, thereby specifying the exact package that needs to be updated and then re-run `npm shrinkwrap` to updated your npm-shrinkwrap.json file.

If you need to find out which packages have become outdated, simply run

```
npm outdated
```

The above command will simply read the repository and inform you of any outdated packages. You can then examine them and decide whether or not you want to include them in production after thoroughly testing them.

Also note that by default npm shrinkwrap does not include your devDependencies unless you pass run it with the –dev flag.

```
npm shrinkwrap --dev
```

http://dtrace.org/blogs/dap/2012/01/05/where-does-your-node-program-spend-its-time/