NPM

简介

module（模块）：每一个js文件，就是一个模块。

package（包）：由多个模块组成，是实现某个功能模块的集合。

对使用者来说，模块和包的区别是透明的，因此经常不作区分。

node根据common JS规范实现了包机制，开发了npm来解决包的发布和获取需求。

NPM（Node Package Manager）是NodeJS包管理和分发工具，是随同NodeJS一起安装的。

Nodejs自身提供了基本的模块，但是开发实际应用过程中仅仅依靠这些基本模块是远远不够的，那么如何获得其他模块文件？我开发的模块又如何分享给其他人使用呢？另外，即便拥有了这些模块，又该如何管理呢？

幸运的是，有了NPM，可以很快的找到要使用的包，进行下载、安装以及管理已经安装的包，让我们减少工作量。

NPM能解决NodeJS代码部署上的很多问题，常见的使用场景有以下几种：

- 允许用户从NPM服务器下载别人编写的第三方包到本地使用。
- 允许用户从NPM服务器下载并安装别人编写的命令行程序到本地使用。
- 允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

```
//查看npm版本
$ npm -v
2.3.0
```

npm虽然是随同node一起安装的，但却是一个独立软件，也会有版本更新，可以使用如下命令升级：

```
npm install npm -g
```

由于npm官网的服务器在境外，国内访问速度很慢，所以，淘宝在国内制作了一个npm的镜像服务器，访问速度很快，

设置国内npm镜像，cmd命令执行

```
npm config set registry https://registry.npm.taobao.org --global
npm config set disturl https://npm.taobao.org/dist --global
```

使用镜像地址安装模块时，如果出现安装失败，可以尝试取消SSL验证：

```
npm config set strict-ssl false
```

另外，还可以安装淘宝镜像cnpm，使用 ” cnpm install xxx “ 的方式安装模块（不推荐）：

```
npm install -g cnpm -registry=https://registry.npm.taobao.org
```

NPM常用命令

命令	说明
npm init	初始化 package.json
npm install	安装模块
npm start	执行package.json文件中start字段的shell脚本
npm stop	执行package.json文件中stop字段的shell脚本
npm update	升级模块（包）到最新版本
npm run	运行package.json中script字段的任意指令， 如果script字段缺省，那么将会列出所有可以run的指令。
npm uninstall	卸载包

npm大部分命令，都是用于操作package.json文件，在package.json章节将详细介绍以上命令。

package.json

每个项目的根目录下面，一般都有一个package.json文件，定义了这个项目所需要的各种模块，以及项目的配置信息（比如名称、版本、许可证等）。

1. 生成

package.json文件可以手工编写，也可以使用**npm init**命令自动生成。

```
$ npm init
```

这个命令采用互动方式，要求用户回答一些问题，然后在当前目录生成一个基本的package.json文件。所有问题之中，只有项目名称（name）和项目版本（version）是必填的，其他都是选填的。并将你的配置写成一个package.json文件。

可以使用了-f|--force 或者 -y|--yes 这些参数，生成一个默认的package.json文件。

```
//-f|--force  强制生成（不会出现任何交互提问）
npm init -f

//-y|--yes 所有交互提问都回答“yes”
npm init -y
```

2. 安装模块

全局安装

在任何项目中都是可以直接调用的，很多模块是所有项目都会用到的，这类包安装到全局

安装目录：C:\Users\用户名\AppData\Roaming\npm\node_modules\

局部安装

只有本项目中才能调用，因为我们需要的包可能有多个，他们之间相互依赖的，如果我们使用全局包，那么每次包的升级、更新等就会影响你的多个项目，那么依赖关系就会被破坏，所以使用本地安装有利于不同项目之间的独立性。

```
//安装到本地 node_modules（package.json中无记录）
npm install <packages-name>

//全局安装
npm install <packages-name>  -g

//安装这个包的最新版本，并写入package.json文件的dependencies字段 （--save简写：-S）。
npm install <packages-name> --save

//安装这个包的最新版本并写入package.json文件的devDependencies字段。（--save-dev 缩写 -D）
npm install <packages-name> --save-dev
```

全局与本地的区别

使用全局模式安装的包并不能直接在 JavaScript 文件中用 require 获 得，因为 require 不会搜索 /usr/local/lib/node_modules/。

当我们要把某个包作为工程运行时的一部分时，通过本地模式获取，如果要在命令行下使用，则使用全局模式安装。

安装指定版本

```
npm install --save-dev xxxx@<版本号>
```

版本标记	说明
1.2.2	遵循“大版本.次要版本.小版本”的格式规定，安装时只安装指定版本
~1.2.2	表示安装1.2.x的最新版本（不低于1.2.2），但是不安装1.3.x 也就是说安装时不改变大版本号和次要版本号。
^1.2.2	表示安装1.x.x的最新版本（不低于1.2.2），但是不安装2.x.x
>=1.2.0 <1.3.0	大于1.2.2版 小于1.3.0

今后，复制这个package.json文件到其他地方，直接使用**npm install**命令，就会按照package.json里面的配置内容，在当前目录中安装所需模块，方便重建相同项目。

卸载包

```
//卸载全局安装包
npm uninstall -g <package>
```

3. package-lock.json

普通的package.json的库前面的版本写法默认是^开头。假如一个库的版本是^2.0，npm安装的时候并不是一定是安装的2.0,而是大于等于这个版本。

因为npm是开源世界，各库包的版本语义可能并不相同，在完全相同的一个nodejs的代码库，在不同时间或者不同npm下载源（yarn 源）之下，下到的各依赖库包版本可能有所不同，因此其依赖库包行为特征也不同，甚至完全不兼容。

因此npm最新的版本就开始提供自动生成package-lock.json功能，为的是让开发者知道只要你保存了源文件，到一个新的机器上、或者新的下载源，只要按照这个package-lock.json所标示的具体版本下载依赖库包，就能确保所有库包与你上次安装的完全一样。

package.json主要字段

scripts

scripts里，以 key / value 的方式定义**shell脚本**

> shell脚本：操作系统可以分成kernel（核心）和shell（外壳）两部分，shell就是用户界面，用于用户操作，shell都提供输入命令的窗口，在命令窗口输入的命令被称为shell脚本）

```
//定义script脚本
"scripts": {
    "mysql_login":"mysql -u root -p",
    "start":"node test.js"
}
//运行mysql_login命令
npm run mysql_login
```

```
//npm会默认设置一些script的值，比如start、test等，可以覆盖这些值，还可以如下简写：
npm start === npm run start
npm stop === npm run stop
npm test === npm run test
npm restart === npm run rerestart
```

注意：如果没有在scripts中定义start，项目根目录又有一个server.js，那么npm start，就会执行server.js

hook（钩子）用法

每个命令**运行前**都会执行对应命令的 **pre**+scriptname 脚本

每个命令**结束后**都会执行对应命令的 **post**+scriptname脚本

```
scripts: {
  "prebuild" : "echo \" this is pre_build \"",
  "build" : "echo \" this is build \"",

  "postbuild" : "echo \" this is post_build \""
```

```
}

npm run build
//运行结果
> node_test@1.0.0 prebuild E:\node_test
> echo " this is pre_build "

" this is pre_build "

> node_test@1.0.0 build E:\node_test
> echo " this is build "

" this is build "

> node_test@1.0.0 postbuild E:\node_test
> echo " this is post_build "

" this is post_build "
```

dependencies

dependencies字段指定了项目运行所依赖的模块，是生产环境中需要的依赖，即正常运行时所需要的依赖模块

devDependencies

endevDependencies指定项目开发所需要的模块。比如：clean-css，我们用它压缩css文件，它们不会被部署到生产环境。

peerDependencies

有时，你的项目和所依赖的模块，都会同时依赖另一个模块，但是所依赖的版本不一样。比如，你的项目依赖A模块和B模块的1.0版，而A模块本身又依赖B模块的2.0版。

peerDependencies字段，就是用来供插件指定其所需要的主工具的版本。

main

指定加载的入口文件，`require('moduleName')`就会加载这个文件。这个字段的默认值是模块根目录下面的`index.js`。

config

用于添加命令行的环境变量。

其他字段(较少用)

- name: 项目名称
- version: 项目版本号
- description: 项目描述
- keywords: {Array}关键词，便于用户搜索到我们的项目
- homepage: 项目url主页
- author,contributors: 作者和贡献者
- bin：用来指定各个内部命令对应的可执行文件的位置。(较少用)
- bugs: 项目问题反馈的Url或email配置

- license: 项目许可证，让使用者知道是如何被允许使用此项目
- files: 包含在项目中的文件数组
- man：用来指定当前模块的man文档的位置。
- repository: 代码被托管在何处，这对那些想要参与到这个项目中的人来说很有帮助。
- engines：指明了该模块运行的平台，比如 Node 的某个版本或者浏览器。
- browser字段：指定该模板供浏览器使用的版本。
- preferGlobal：表示当用户不将该模块安装为全局模块时（即不用–global参数），要不要显示警告。
- style：指定供浏览器使用时，样式文件所在的位置。

node_modules目录

用NPM安装的模块（包），都会保存在这个目录中。

使用**npm install**命令将自动生成node_modules目录，并且，会依据package.json配置文件安装所有的模块（包），保存在该目录中。

.bin子目录

npm run 只能执行shell脚本，如果想运行xxx.js，就需要输入“node xxx.js”这样的命令，为了简化这一操作，npm提供了一个简便方法：

1. node_modules目录中默认有一个**.bin**目录(这个目录是npm创建的，可以在CMD中用 "md .bin"命令创建）
2. 在这个目录中可以编辑运行的shell脚本，配合定义package.json文件的scripts字段，通过**npm run** 命令，可以直接运行node_modules目录中的js文件

实现步骤：

在node_modules目录下新建hxsd目录里，在hxsd目录下，新建一个测试用的mytest.js文件

```
console.log("mytest");
```

在node_modules.bin目录下创建mytest.cmd文件(cmd文件是批处理文件)

```
@IF EXIST "%~dp0\node.exe" (
  "%~dp0\node.exe"  "%~dp0\..\hxsd\mytest.js" %*
) ELSE (
  @SETLOCAL
  @SET PATHEXT=%PATHEXT:;.JS;=;%
  node  "%~dp0\..\hxsd\mytest.js" %*
)
```

在package.json文件的scripts中增加

```
"scripts": {
    "hxsd_test":"mytest"
}
```

npm运行

```
npm run hxsd_test
```