**NPM 使用介绍**

NPM是随同NodeJS一起安装的包管理工具，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。

由于新版的nodejs已经集成了npm，所以之前npm也一并安装好了。同样可以通过输入 **"npm -v"**来测试是否成功安装。命令如下，出现版本提示表示安装成功:

$ npm -v

2.3.0

如果你安装的是旧版本的 npm，可以很容易得通过 npm 命令来升级，命令如下：

$ sudo npm install npm -g

/usr/local/bin/npm -> /usr/local/lib/node\_modules/npm/bin/npm-cli.js

npm@2.14.2 /usr/local/lib/node\_modules/npm

如果是 Window 系统使用以下命令即可：

npm install npm -g

*使用淘宝镜像的命令：*

cnpm install npm -g

**使用 npm 命令安装模块**

npm 安装 Node.js 模块语法格式如下：

$ npm install <Module Name>

以下实例，我们使用 npm 命令安装常用的 Node.js web框架模块 **express**:

$ npm install express

安装好之后，express 包就放在了工程目录下的 node\_modules 目录中，因此在代码中只需要通过 **require('express')** 的方式就好，无需指定第三方包路径。

var express = require('express');

**全局安装与本地安装**

npm 的包安装分为本地安装（local）、全局安装（global）两种，从敲的命令行来看，差别只是有没有-g而已，比如

npm install express # 本地安装

npm install express -g # 全局安装

如果出现以下错误：

npm err! Error: connect ECONNREFUSED 127.0.0.1:8087

解决办法为：

$ npm config set proxy null

**本地安装**

* 1. 将安装包放在 ./node\_modules 下（运行 npm 命令时所在的目录），如果没有 node\_modules 目录，会在当前执行 npm 命令的目录下生成 node\_modules 目录。
* 2. 可以通过 require() 来引入本地安装的包。

**全局安装**

* 1. 将安装包放在 /usr/local 下或者你 node 的安装目录。
* 2. 可以直接在命令行里使用。

如果你希望具备两者功能，则需要在两个地方安装它或使用 **npm link**。

接下来我们使用全局方式安装 express

$ npm install express -g

安装过程输出如下内容，第一行输出了模块的版本号及安装位置。

express@4.13.3 node\_modules/express

├── escape-html@1.0.2

├── range-parser@1.0.2

├── merge-descriptors@1.0.0

├── array-flatten@1.1.1

├── cookie@0.1.3

├── utils-merge@1.0.0

├── parseurl@1.3.0

├── cookie-signature@1.0.6

├── methods@1.1.1

├── fresh@0.3.0

├── vary@1.0.1

├── path-to-regexp@0.1.7

├── content-type@1.0.1

├── etag@1.7.0

├── serve-static@1.10.0

├── content-disposition@0.5.0

├── depd@1.0.1

├── qs@4.0.0

├── finalhandler@0.4.0 (unpipe@1.0.0)

├── on-finished@2.3.0 (ee-first@1.1.1)

├── proxy-addr@1.0.8 (forwarded@0.1.0, ipaddr.js@1.0.1)

├── debug@2.2.0 (ms@0.7.1)

├── type-is@1.6.8 (media-typer@0.3.0, mime-types@2.1.6)

├── accepts@1.2.12 (negotiator@0.5.3, mime-types@2.1.6)

└── send@0.13.0 (destroy@1.0.3, statuses@1.2.1, ms@0.7.1, mime@1.3.4, http-errors@1.3.1)

**查看安装信息**

你可以使用以下命令来查看所有全局安装的模块：

$ npm list -g

├─┬ cnpm@4.3.2

│ ├── auto-correct@1.0.0

│ ├── bagpipe@0.3.5

│ ├── colors@1.1.2

│ ├─┬ commander@2.9.0

│ │ └── graceful-readlink@1.0.1

│ ├─┬ cross-spawn@0.2.9

│ │ └── lru-cache@2.7.3

……

如果要查看某个模块的版本号，可以使用命令如下：

$ npm list grunt

projectName@projectVersion /path/to/project/folder

└── grunt@0.4.1

[**使用 package.json**](日记.docx)

package.json 位于模块的目录下，用于定义包的属性。接下来让我们来看下 express 包的 package.json 文件，位于 node\_modules/express/package.json 内容：

{

"name": "express",

"description": "Fast, unopinionated, minimalist web framework",

"version": "4.13.3",

"author": {

"name": "TJ Holowaychuk",

"email": "tj@vision-media.ca"

},

"contributors": [

{

"name": "Aaron Heckmann",

"email": "aaron.heckmann+github@gmail.com"

},

{

"name": "Ciaran Jessup",

"email": "ciaranj@gmail.com"

},

{

"name": "Douglas Christopher Wilson",

"email": "doug@somethingdoug.com"

},

{

"name": "Guillermo Rauch",

"email": "rauchg@gmail.com"

},

{

"name": "Jonathan Ong",

"email": "me@jongleberry.com"

},

{

"name": "Roman Shtylman",

"email": "shtylman+expressjs@gmail.com"

},

{

"name": "Young Jae Sim",

"email": "hanul@hanul.me"

}

],

"license": "MIT",

"repository": {

"type": "git",

"url": "git+https://github.com/strongloop/express.git"

},

"homepage": "http://expressjs.com/",

"keywords": [

"express",

"framework",

"sinatra",

"web",

"rest",

"restful",

"router",

"app",

"api"

],

"dependencies": {

"accepts": "~1.2.12",

"array-flatten": "1.1.1",

"content-disposition": "0.5.0",

"content-type": "~1.0.1",

"cookie": "0.1.3",

"cookie-signature": "1.0.6",

"debug": "~2.2.0",

"depd": "~1.0.1",

"escape-html": "1.0.2",

"etag": "~1.7.0",

"finalhandler": "0.4.0",

"fresh": "0.3.0",

"merge-descriptors": "1.0.0",

"methods": "~1.1.1",

"on-finished": "~2.3.0",

"parseurl": "~1.3.0",

"path-to-regexp": "0.1.7",

"proxy-addr": "~1.0.8",

"qs": "4.0.0",

"range-parser": "~1.0.2",

"send": "0.13.0",

"serve-static": "~1.10.0",

"type-is": "~1.6.6",

"utils-merge": "1.0.0",

"vary": "~1.0.1"

},

"devDependencies": {

"after": "0.8.1",

"ejs": "2.3.3",

"istanbul": "0.3.17",

"marked": "0.3.5",

"mocha": "2.2.5",

"should": "7.0.2",

"supertest": "1.0.1",

"body-parser": "~1.13.3",

"connect-redis": "~2.4.1",

"cookie-parser": "~1.3.5",

"cookie-session": "~1.2.0",

"express-session": "~1.11.3",

"jade": "~1.11.0",

"method-override": "~2.3.5",

"morgan": "~1.6.1",

"multiparty": "~4.1.2",

"vhost": "~3.0.1"

},

"engines": {

"node": ">= 0.10.0"

},

"files": [

"LICENSE",

"History.md",

"Readme.md",

"index.js",

"lib/"

],

"scripts": {

"test": "mocha --require test/support/env --reporter spec --bail --check-leaks test/ test/acceptance/",

"test-ci": "istanbul cover node\_modules/mocha/bin/\_mocha --report lcovonly -- --require test/support/env --reporter spec --check-leaks test/ test/acceptance/",

"test-cov": "istanbul cover node\_modules/mocha/bin/\_mocha -- --require test/support/env --reporter dot --check-leaks test/ test/acceptance/",

"test-tap": "mocha --require test/support/env --reporter tap --check-leaks test/ test/acceptance/"

},

"gitHead": "ef7ad681b245fba023843ce94f6bcb8e275bbb8e",

"bugs": {

"url": "https://github.com/strongloop/express/issues"

},

"\_id": "express@4.13.3",

"\_shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",

"\_from": "express@\*",

"\_npmVersion": "1.4.28",

"\_npmUser": {

"name": "dougwilson",

"email": "doug@somethingdoug.com"

},

"maintainers": [

{

"name": "tjholowaychuk",

"email": "tj@vision-media.ca"

},

{

"name": "jongleberry",

"email": "jonathanrichardong@gmail.com"

},

{

"name": "dougwilson",

"email": "doug@somethingdoug.com"

},

{

"name": "rfeng",

"email": "enjoyjava@gmail.com"

},

{

"name": "aredridel",

"email": "aredridel@dinhe.net"

},

{

"name": "strongloop",

"email": "callback@strongloop.com"

},

{

"name": "defunctzombie",

"email": "shtylman@gmail.com"

}

],

"dist": {

"shasum": "ddb2f1fb4502bf33598d2b032b037960ca6c80a3",

"tarball": "http://registry.npmjs.org/express/-/express-4.13.3.tgz"

},

"directories": {},

"\_resolved": "https://registry.npmjs.org/express/-/express-4.13.3.tgz",

"readme": "ERROR: No README data found!"

}

**Package.json 属性说明**

* **name** - 包名。
* **version** - 包的版本号。
* **description** - 包的描述。
* **homepage** - 包的官网 url 。
* **author** - 包的作者姓名。
* **contributors** - 包的其他贡献者姓名。
* **dependencies** - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node\_module 目录下。
* **repository** - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。
* **main** - main 字段指定了程序的主入口文件，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。
* **keywords** - 关键字

**卸载模块**

我们可以使用以下命令来卸载 Node.js 模块。

$ npm uninstall express

卸载后，你可以到 /node\_modules/ 目录下查看包是否还存在，或者使用以下命令查看：

$ npm ls

**更新模块**

我们可以使用以下命令更新模块：

$ npm update express

**搜索模块**

使用以下来搜索模块：

$ npm search express

**创建模块**

创建模块，package.json 文件是必不可少的。我们可以使用 NPM 生成 package.json 文件，生成的文件包含了基本的结果。

$ npm init

This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields

and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and

save it as a dependency in the package.json file.

Press ^C at any time to quit.

name: (node\_modules) runoob # 模块名

version: (1.0.0)

description: Node.js 测试模块(www.runoob.com) # 描述

entry point: (index.js)

test command: make test

git repository: https://github.com/runoob/runoob.git # Github 地址

keywords:

author:

license: (ISC)

About to write to ……/node\_modules/package.json: # 生成地址

{

"name": "runoob",

"version": "1.0.0",

"description": "Node.js 测试模块(www.runoob.com)",

……

}

Is this ok? (yes) yes

以上的信息，你需要根据你自己的情况输入。在最后输入 "yes" 后会生成 package.json 文件。

接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

$ npm adduser

Username: mcmohd

Password:

Email: (this IS public) mcmohd@gmail.com

接下来我们就用以下命令来发布模块：

$ npm publish

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

**版本号**

使用NPM下载和发布代码时都会接触到版本号。NPM使用语义版本号来管理代码，这里简单介绍一下。

语义版本号分为X.Y.Z三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新。

* 如果只是修复bug，需要更新Z位。
* 如果是新增了功能，但是向下兼容，需要更新Y位。
* 如果有大变动，向下不兼容，需要更新X位。

版本号有了这个保证后，在申明第三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如"argv": "0.0.x"表示依赖于0.0.x系列的最新版argv。

NPM支持的所有版本号范围指定方式可以查看[官方文档](https://npmjs.org/doc/files/package.json.html#dependencies)。

**NPM 常用命令**

除了本章介绍的部分外，NPM还提供了很多功能，package.json里也有很多其它有用的字段。

除了可以在[npmjs.org/doc/](https://npmjs.org/doc/)查看官方文档外，这里再介绍一些NPM常用命令。

NPM提供了很多命令，例如install和publish，使用npm help可查看所有命令。

* NPM提供了很多命令，例如install和publish，使用npm help可查看所有命令。
* 使用npm help <command>可查看某条命令的详细帮助，例如npm help install。
* 在package.json所在目录下使用npm install . -g可先在本地安装当前命令行程序，可用于发布前的本地测试。
* 使用npm update <package>可以把当前目录下node\_modules子目录里边的对应模块更新至最新版本。
* 使用npm update <package> -g可以把全局安装的对应命令行程序更新至最新版。
* 使用npm cache clear可以清空NPM本地缓存，用于对付使用相同版本号发布新版本代码的人。
* 使用npm unpublish <package>@<version>可以撤销发布自己发布过的某个版本代码。

**使用淘宝 NPM 镜像**

大家都知道国内直接使用 npm 的官方镜像是非常慢的，这里推荐使用淘宝 NPM 镜像。

淘宝 NPM 镜像是一个完整 npmjs.org 镜像，你可以用此代替官方版本(只读)，同步频率目前为 10分钟 一次以保证尽量与官方服务同步。

你可以使用淘宝定制的 cnpm (gzip 压缩支持) 命令行工具代替默认的 npm:

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

这样就可以使用 cnpm 命令来安装模块了：

$ cnpm install [name]

Yarn

**亮点**

1. 极致的快速—会缓存已经下载过的包，避免重复下载~~~这特性比npm好多了，不用每次都请求下载
   * 离线模式 — 之前下载过的包，可以离线再次安装
2. 安全 — 下载前会检查签名及包的完整性
3. 可靠可确定性
   * 保证各平台依赖的一致性
4. 网络优化 — 力求网络资源最大利用化，让资源下载完美队列执行，避免大量的无用请求，下载失败会自动重新请求，避免整个安装过程失败
5. 扁平化模式 — 对于不匹配的依赖版本的包创立一个独立的包，避免创建重复的。

**结构**

在 Node 生态系统中，依赖通常安装在项目的 node\_modules 文件夹中。然而，这个文件的结构和实际依赖树可能有所区别，因为重复的依赖可以合并到一起。npm 客户端把依赖安装到 node\_modules 目录的过程具有不确定性。这意味着当依赖的安装顺序不同时，node\_modules 目录的结构可能会发生变化。这种差异可能会导致类似“我的机子上可以运行，别的机子不行”的情况，并且通常要花费大量时间定位与解决。

Yarn 通过 lockfiles 文件以及一个确定性的、可靠的安装算法，解决了版本问题和 npm 的**不确定性问题**。Lockfile 文件把安装的软件包版本锁定在某个特定版本，并保证 node\_modules 目录在所有机器上的安装结果都是相同的。Lockfile 还使用简洁的有序键名的格式，保证了每次的文件变化最小化，进行代码审查也更为简单。

安装过程分为以下三个步骤：

1. **处理：** Yarn 通过向代码仓库发送请求，并递归查找每个依赖项，从而解决依赖关系。
2. **抓取：** 接下来，Yarn 会查找全局的缓存目录，检查所需的软件包是否已被下载。如果没有，Yarn 会抓取对应的压缩包，并放置在全局的缓存目录中，因此 Yarn 支持离线安装，同一个安装包不需要下载多次。依赖也可以通过 tarball 的压缩形式放置在源码控制系统中，以支持完整的离线安装。
3. **生成：** 最后，Yarn 从全局缓存中把需要用到的所有文件复制到本地的 node\_modules 目录中。

通过清晰地细分这些步骤，以及确定性的算法支持，使得 Yarn 支持并行操作，从而最大化地利用资源，并加速安装进程。在一些 Facebook 的项目上，Yarn 甚至可以把安装过程降低一个数量级，从几分钟到只需几秒钟。Yarn 还使用了互斥锁，以确保多个 CLI 实例同时运行时不会互相冲突与影响。

纵观整个过程，Yarn 对于软件包安装加上了严格的限制。你可以对哪个生命周期脚本作用于哪个软件包进行控制。软件包的 checksum 也会存储在 lockfile 中，以确保每一次安装都可以得到同一个包。