# package.json属性详解

参考文档列表(<https://docs.npmjs.com/>)

本文档的内容是package.json配置里边的属性含义。package.json必须是一个严格的json文件，而不仅仅是js里边的一个对象。其中很多属性可以通过npm-config来生成。

## name

package.json中最重要的属性是name和version两个属性，这两个属性是必须要有的，否则模块就无法被安装，这两个属性一起形成了一个npm模块的唯一标识符。模块中内容变更的同时，模块版本也应该一起变化。name属性就是你的模块名称，下面是一些命名规则:name必须小于等于214个字节，包括前缀名称在内（如 xxx/xxxmodule）。

1. name不能以”\_”或”.”开头
2. 不能含有大写字母
3. name会成为url的一部分，不能含有url非法字符

下面是官网文档的一些建议： 

1. 不要使用和node核心模块一样的名称
2. name中不要含有”js”和”node”。 It’s assumed that it’s js, since you’re writing a package.json file, and you can specify the engine using the “engines” field. (See below.)
3. name属性会成为模块url、命令行中的一个参数或者一个文件夹名称，任何非url安全的字符在name中都不能使用，也不能以”\_”或”.”开头
4. name属性也许会被写在require()的参数中，所以最好取个简短而语义化的值。
5. 创建一个模块前可以先到后边的网址查查name是否已经被占用. <https://www.npmjs.com/>

name属性可以有一些前缀如 e.g. @myorg/mypackage.在npm-scope(7)的文档中可以看到详细说明

## version

version必须可以被npm依赖的一个node-semver模块解析。具体规则见下面的dependencies模块

## description

一个描述，方便别人了解你的模块作用，搜索的时候也有用。

## keywords

一个字符串数组，方便别人搜索到本模块

## homepage

项目主页url注意: 这个项目主页url和url属性不同，如果你填写了url属性，npm注册工具会认为你把项目发布到其他地方了，获取模块的时候不会从npm官方仓库获取，而是会重定向到url属性配置的地址。（原文档中用了 spit(吐)这个单词，作者表示他不是在开玩笑:） 

## bugs

填写一个bug提交地址或者一个邮箱，被你的模块坑到的人可以通过这里吐槽，例如： 

{ "url" : "https://github.com/owner/project/issues"

, "email" : "project@hostname.com"

}

url和email可以任意填或不填，如果只填一个，可以直接写成一个字符串而不是对象。如果填写了url，npm bugs命令会使用这个url。

## license

你应该为你的模块制定一个协议，让用户知道他们有何权限来使用你的模块，以及使用该模块有哪些限制。最简单的，例如你用BSD-3-Clause 或 MIT之类的协议，如下：{ “license” : “BSD-3-Clause” }你可以在https://spdx.org/licenses/ 这个地址查阅协议列表 。

## 和用户相关的属性: author, contributors

“author”是一个码农， “contributors”是一个码农数组。 “person”是一个有一些描述属性的对象，如下 like this: 

{ "name" : "Barney Rubble"

, "email" : "b@rubble.com"

, "url" : "http://barnyrubble.tumblr.com/"

}

也可以按如下格式缩写，npm会帮着转换:“Barney Rubble b@rubble.com (http://barnyrubble.tumblr.com/)”email和url属性实际上都是可以省略的。描述用户信息的还有一个”maintainers”（维护者）属性。

## files

“files”属性的值是一个数组，内容是模块下文件名或者文件夹名，如果是文件夹名，则文件夹下所有的文件也会被包含进来（除非文件被另一些配置排除了）你也可以在模块根目录下创建一个”.npmignore”文件（windows下无法直接创建以”.”开头的文件，使用linux命令行工具创建如git bash），写在这个文件里边的文件即便被写在files属性里边也会被排除在外，这个文件的写法”.gitignore”类似。

## main

main属性指定了程序的主入口文件。意思是，如果你的模块被命名为foo，用户安装了这个模块并通过require(“foo”)来使用这个模块，那么require返回的内容就是main属性指定的文件中 module.exports指向的对象。它应该指向模块根目录下的一个文件。对大对数模块而言，这个属性更多的是让模块有一个主入口文件，然而很多模块并不写这个属性。

## bin

很多模块有一个或多个需要配置到PATH路径下的可执行模块，npm让这个工作变得十分简单（实际上npm本身也是通过bin属性安装为一个可执行命令的）如果要用npm的这个功能，在package.json里边配置一个bin属性。bin属性是一个已命令名称为key，本地文件名称为value的map如下： 

{ "bin" : { "myapp" : "./cli.js" } }

模块安装的时候，若是全局安装，则npm会为bin中配置的文件在bin目录下创建一个软连接（对于windows系统，默认会在C:\Users\username\AppData\Roaming\npm目录下），若是局部安装，则会在项目内的./node\_modules/.bin/目录下创建一个软链接。因此，按上面的例子，当你安装myapp的时候，npm就会为cli.js在/usr/local/bin/myapp路径创建一个软链接。如果你的模块只有一个可执行文件，并且它的命令名称和模块名称一样，你可以只写一个字符串来代替上面那种配置，例如： 

{ "name": "my-program"

, "version": "1.2.5"

, "bin": "./path/to/program" }

作用和如下写法相同:

{ "name": "my-program"

, "version": "1.2.5"

, "bin" : { "my-program" : "./path/to/program" } }

## man

制定一个或通过数组制定一些文件来让linux下的man命令查找文档地址。如果只有一个文件被指定的话，安装后直接使用man+模块名称，而不管man指定的文件的实际名称。例如:

{ "name" : "foo"

, "version" : "1.2.3"

, "description" : "A packaged foo fooer for fooing foos"

, "main" : "foo.js"

, "man" : "./man/doc.1"

}

通过man foo命令会得到 ./man/doc.1 文件的内容。如果man文件名称不是以模块名称开头的，安装的时候会给加上模块名称前缀。因此，下面这段配置：

{ "name" : "foo"

, "version" : "1.2.3"

, "description" : "A packaged foo fooer for fooing foos"

, "main" : "foo.js"

, "man" : [ "./man/foo.1", "./man/bar.1" ]

}

会创建一些文件来作为man foo和man foo-bar命令的结果。man文件必须以数字结尾，或者如果被压缩了，以.gz结尾。数字表示文件将被安装到man的哪个部分。

{ "name" : "foo"

, "version" : "1.2.3"

, "description" : "A packaged foo fooer for fooing foos"

, "main" : "foo.js"

, "man" : [ "./man/foo.1", "./man/foo.2" ]

}

会创建 man foo 和 man 2 foo 两条命令。

## directories

CommonJs通过directories来制定一些方法来描述模块的结构，看看npm的package.json文件https://registry.npmjs.org/npm/latest ，可以发现里边有这个字段的内容。



目前这个配置没有任何作用，将来可能会整出一些花样来。

## directories.lib

告诉用户模块中lib目录在哪，这个配置目前没有任何作用，但是对使用模块的人来说是一个很有用的信息。

## directories.bin

如果你在这里指定了bin目录，这个配置下面的文件会被加入到bin路径下，如果你已经在package.json中配置了bin目录，那么这里的配置将不起任何作用。

## directories.man

指定一个目录，目录里边都是man文件，这是一种配置man文件的语法糖。

## directories.doc

在这个目录里边放一些markdown文件，可能最终有一天它们会被友好的展现出来（应该是在npm的网站上）

## directories.example

放一些示例脚本，或许某一天会有用 - -！

## repository

指定一个代码存放地址，对想要为你的项目贡献代码的人有帮助。像这样：

"repository" :

{ "type" : "git"

, "url" : "https://github.com/npm/npm.git"

}

"repository" :

{ "type" : "svn"

, "url" : "https://v8.googlecode.com/svn/trunk/"

}

若你的模块放在GitHub, GitHub gist, Bitbucket, or GitLab的仓库里，npm install的时候可以使用缩写标记来完成：

"repository": "npm/npm"

"repository": "gist:11081aaa281"

"repository": "bitbucket:example/repo"

"repository": "gitlab:another/repo"

## scripts

scripts属性是一个对象，里边指定了项目的生命周期个各个环节需要执行的命令。key是生命周期中的事件，value是要执行的命令。具体的内容有 install start stop 等，详见 https://docs.npmjs.com/misc/scripts

## config

用来设置一些项目不怎么变化的项目配置，例如port等。用户用的时候可以使用如下用法： 

http.createServer(...).listen(process.env.npm\_package\_config\_port)

可以通过npm config set foo:port 80来修改config。详见https://docs.npmjs.com/misc/config

{ "name" : "foo"

, "config" : { "port" : "8080" } }

## dependencies

dependencies属性是一个对象，配置模块依赖的模块列表，key是模块名称，value是版本范围，版本范围是一个字符，可以被一个或多个空格分割。dependencies也可以被指定为一个git地址或者一个压缩包地址。不要把测试工具或transpilers写到dependencies中。 下面是一些写法，详见https://docs.npmjs.com/misc/semver 

* version 精确匹配版本
* >version 必须大于某个版本
* >=version 大于等于
* <version 小于
* <=versionversion 小于
* ~version “约等于”，具体规则详见semver文档
* ^version “兼容版本”具体规则详见semver文档
* 1.2.x 仅一点二点几的版本
* http://… 见下面url作为denpendencies的说明
* 任何版本
* “” 空字符，和\*相同
* version1 - version2 相当于 >=version1 <=version2.
* range1 || range2 范围1和范围2满足任意一个都行
* git… 见下面git url作为denpendencies的说明
* user/repo See 见下面GitHub仓库的说明
* tag 发布的一个特殊的标签，见npm-tag的文档 https://docs.npmjs.com/getting-started/using-tags
* path/path/path 见下面本地模块的说明下面的写法都是可以的:

{ "dependencies" :

{ "foo" : "1.0.0 - 2.9999.9999"

, "bar" : ">=1.0.2 <2.1.2"

, "baz" : ">1.0.2 <=2.3.4"

, "boo" : "2.0.1"

, "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"

, "asd" : "http://asdf.com/asdf.tar.gz"

, "til" : "~1.2"

, "elf" : "~1.2.3"

, "two" : "2.x"

, "thr" : "3.3.x"

, "lat" : "latest"

, "dyl" : "file:../dyl"

}

}

## URLs as Dependencies

在版本范围的地方可以写一个url指向一个压缩包，模块安装的时候会把这个压缩包下载下来安装到模块本地。

Git URLs as Dependencies

Git url可以像下面一样:

git://github.com/user/project.git#commit-ish

git+ssh://user@hostname:project.git#commit-ish

git+ssh://user@hostname/project.git#commit-ish

git+http://user@hostname/project/blah.git#commit-ish

git+https://user@hostname/project/blah.git#commit-ish

commit-ish 可以是任意标签，哈希值，或者可以检出的分支，默认是master分支。

## GitHub URLs

支持github的 username/modulename 的写法，#后边可以加后缀写明分支hash或标签： 

{

"name": "foo",

"version": "0.0.0",

"dependencies": {

"express": "visionmedia/express",

"mocha": "visionmedia/mocha#4727d357ea"

}

}

## Local Paths

npm2.0.0版本以上可以提供一个本地路径来安装一个本地的模块，通过npm install xxx –save 来安装，格式如下： 

../foo/bar

~/foo/bar

./foo/bar

/foo/bar

package.json 生成的相对路径如下: 

{

"name": "baz",

"dependencies": {

"bar": "file:../foo/bar"

}

}

这种属性在离线开发或者测试需要用npm install的情况，又不想自己搞一个npm server的时候有用，但是发布模块到公共仓库时不应该使用这种属性。

## devDependencies

如果有人想要下载并使用你的模块，也许他们并不希望或需要下载一些你在开发过程中使用的额外的测试或者文档框架。在这种情况下，最好的方法是把这些依赖添加到devDependencies属性的对象中。这些模块会在npm link或者npm install的时候被安装，也可以像其他npm配置一样被管理，详见npm的config文档。对于一些跨平台的构建任务，例如把CoffeeScript编译成JavaScript，就可以通过在package.json的script属性里边配置prepublish脚本来完成这个任务，然后需要依赖的coffee-script模块就写在devDependencies属性种。例如: 

{ "name": "ethopia-waza",

"description": "a delightfully fruity coffee varietal",

"version": "1.2.3",

"devDependencies": {

"coffee-script": "~1.6.3"

},

"scripts": {

"prepublish": "coffee -o lib/ -c src/waza.coffee"

},

"main": "lib/waza.js"

}

prepublish脚本会在发布之前运行，因此用户在使用之前就不用再自己去完成编译的过程了。在开发模式下，运行npm install也会执行这个脚本（见npm script文档），因此可以很方便的调试。

## peerDependencies

有时候做一些插件开发，比如grunt等工具的插件，它们往往是在grunt的某个版本的基础上开发的，而在他们的代码中并不会出现require(“grunt”)这样的依赖，dependencies配置里边也不会写上grunt的依赖，为了说明此模块只能作为插件跑在宿主的某个版本范围下，可以配置peerDependencies： 

{

"name": "tea-latte",

"version": "1.3.5",

"peerDependencies": {

"tea": "2.x"

}

}

上面这个配置确保再npm install的时候tea-latte会和2.x版本的tea一起安装，而且它们两个的依赖关系是同级的：├── tea-latte@1.3.5└── bundledDependencies

上面的单词少个d，写成bundleDependencies也可以。指定发布的时候会被一起打包的模块。

## bundledDependencies

上面的单词少个d，写成bundleDependencies也可以。指定发布的时候会被一起打包的模块。

## optionalDependencies

如果一个依赖模块可以被使用， 同时你也希望在该模块找不到或无法获取时npm继续运行，你可以把这个模块依赖放到optionalDependencies配置中。这个配置的写法和dependencies的写法一样，不同的是这里边写的模块安装失败不会导致npm install失败。当然，这种模块就需要你自己在代码中处理模块确实的情况了，例如： 

try {

var foo = require('foo')

var fooVersion = require('foo/package.json').version

} catch (er) {

foo = null

}

if ( notGoodFooVersion(fooVersion) ) {

foo = null

}

// .. then later in your program ..

if (foo) {

foo.doFooThings()

}

optionalDependencies 中的配置会覆盖dependencies中的配置，最好只在一个地方写。

## engines

你可以指定项目运行的node版本范围，如下：{ “engines” : { “node” : “>=0.10.3 <0.12” } }和dependencies一样，如果你不指定版本范围或者指定为\*，任何版本的node都可以。也可以指定一些npm版本可以正确的安装你的模块，例如：{ “engines” : { “npm” : “~1.0.20” } }要注意的是，除非你设置了engine-strict属性，engines属性是仅供参考的。 

## engineStrict

注意：这个属性已经弃用，将在npm 3.0.0 版本干掉。 

## os

可以指定你的模块只能在哪个操作系统上跑：“os” : [ “darwin”, “linux” ]也可以指定黑名单而不是白名单：“os” : [ “!win32” ]服务的操作系统是由process.platform来判断的，这个属性允许黑白名单同时存在，虽然没啥必要这样搞… 

## cpu

限制模块只能在某某cpu架构下运行“cpu” : [ “x64”, “ia32” ]同样可以设置黑名单:“cpu” : [ “!arm”, “!mips” ]cpu架构通过 process.arch 判断 

## preferGlobal

如果您的软件包主要用于安装到全局的命令行应用程序，那么该值设置为true ，如果它被安装在本地，则提供一个警告。实际上该配置并没有阻止用户把模块安装到本地，只是防止该模块被错误的使用引起一些问题。 

## private

如果这个属性被设置为true，npm将拒绝发布它，这是为了防止一个私有模块被无意间发布出去。如果你只想让模块被发布到一个特定的npm仓库，如一个内部的仓库，可与在下面的publishConfig中配置仓库参数。 

## publishConfig

这个配置是会在模块发布时用到的一些值的集合。如果你不想模块被默认被标记为最新的，或者默认发布到公共仓库，可以在这里配置tag或仓库地址。 

## DEFAULT VALUES

npm设置了一些默认参数，如：“scripts”: {“start”: “node server.js”}如果模块根目录下有一个server.js文件，那么npm start会默认运行这个文件。“scripts”:{“preinstall”: “node-gyp rebuild”}如果模块根目录下有binding.gyp, npm将默认用node-gyp来编译preinstall的脚本“contributors”: […]若模块根目录下有AUTHORS 文件，则npm会按Name (url)格式解析每一行的数据添加到contributors中，可以用#添加行注释

# gulp使用篇：

# 项目目录结构

gulps/

├── src/app/ 开发目录

│ ├── sass/

│ │ └─ main.scss

│ ├── scripts/

│ │ └─ javascript.js

│ └── index.html

├── dist/ dist 目标文件目录，这里是编译、压缩过的文件。

└── release/ release 可以发布至服务器的文件。

### 使用npm生成package.json

替换npm 因为网络原因不能用 你懂得

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

使用npm init 初始化package.json文件

{

"name": "gulpbuilder",

"version": "1.0.0",

"description": "Gulp builder",

"main": "gulpfile.js",

"directories": {

"doc": "doc"

},

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [

"gulp",

"sass",

"less",

"style",

"html",

"browser-sync",

"liveReload"

],

"author": {

"name": "linxl",

"email": "350825854@qq.com"

},

"license": "ISC"

}