npm的package.json中文文档 #6

New issue

Open ericdum opened this issue on 2 Dec 2013 · 37 comments

ericdum commented on 2 Dec 2013 Owner

> 以前看nodejs只是看看API没有具体看这些细节，把别人的package.json拿来抄一抄，看字段也知道留下name、version、private、dependencies就好了，能方便地npm install部署就可以了，也没去深究。现在想在用起来在架构上面就遇到了很多问题。包括文件结构、管理、部署、重启……
> 昨天想先了解一下YO、GRUNT、Bower这一套东西正好看到了package.json的devDependencies字段。找到这个文档看，看着看着就看成了中文，也给大家送送福利。这样大家看起来就快多了。
> 有什么问题希望大家快快拍砖。

package.json(5)网页版

简介

本文档有所有package.json中必要的配置。它必须是真正的json，而不是js对象。

本文档中描述的很多行为都受`npm-config(7)`的影响。

默认值

npm会根据包内容设置一些默认值。

- `"scripts": {"start": "node server.js"}`

  如果包的根目录有`server.js`文件，npm会默认将`start`命令设置为`node server.js`。

- `"scripts":{"preinstall": "node-waf clean || true; node-waf configure build"}`

  如果包的根目录有`wscript`文件，npm会默认将`preinstall`命令用node-waf进行编译。

- `"scripts":{"preinstall": "node-gyp rebuild"}`

  如果包的根目录有`binding.gyp`文件，npm会默认将`preinstall`命令用node-gyp进行编译。

- `"contributors": [...]`

  如果包的根目录有`AUTHORS`文件，npm会默认逐行按`Name <email> (url)`格式处理，邮箱和url是可选的。#号和空格开头的行会被忽略。

name

在package.json中_最_重要的就是name和version字段。他们都是必须的，如果没有就无法install。name和version一起组成的标识在假设中是唯一的。改变包应该同时改变version。

name是这个东西的名字。注意：

- 不要把node或者js放在名字中。因为你写了package.json它就被假定成为了js，不过你可以用"engine"字段指定一个引擎（见后文）。
- 这个名字会作为在URL的一部分、命令行的参数或者文件夹的名字。任何non-url-safe的字符都是不能用的。
- 这个名字可能会作为参数被传入require()，所以它应该比较短，但也要意义清晰。
- 在你爱上你的名字之前，你可能要去npm registry查看一下这个名字是否已经被使用了。
  http://registry.npmjs.org/

version

在package.json中_最_重要的就是name和version字段。他们都是必须的，如果没有就无法install。name和version一起组成的标识在假设中是唯一的。改变包应该同时改变version。

version必须能被node-semver解析，它被包在npm的依赖中。（要自己用可以执行`npm install semver`）

Assignees

No one assigned

Labels

nodejs
翻译

Projects

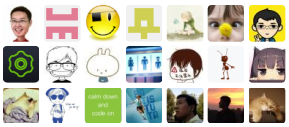
None yet

Milestone

No milestone

Notifications

35 participants



and others

可用的“数字”或者“范围”见semver(7).

description

放简介，字符串。方便屌丝们在`npm search`中搜索。

keywords

关键字，数组、字符串。还是方便屌丝们在`npm search`中搜索。

homepage

项目官网的url。

注意：这和“url”_不_一样。如果你放一个“url”字段，registry会以为是一个跳转到你发布在其他地方的地址，然后喊你滚粗。

嗯，滚粗，没开玩笑。

bugs

你项目的提交问题的url和（或）邮件地址。这对遇到问题的屌丝很有帮助。

差不多长这样：

```
{ "url" : "http://github.com/owner/project/issues"
, "email" : "project@hostname.com"
}
```

你可以指定一个或者指定两个。如果你只想提供一个url，那就不用对象了，字符串就行。

如果提供了url，它会被`npm bugs`命令使用。

license

你应该要指定一个许可证，让人知道使用的权利和限制的。

最简单的方法是，假如你用一个像BSD或者MIT这样通用的许可证，就只需要指定一个许可证的名字，像这样：

```
{ "license" : "BSD" }
```

如果你又更复杂的许可条件，或者想要提供给更多地细节，可以这样:

```
"licenses" : [
  { "type" : "MyLicense"
  , "url" : "http://github.com/owner/project/path/to/license"
  }
]
```

在根目录中提供一个许可证文件也蛮好的。

people fields: author, contributors

author是一个人。contributors是一堆人的数组。person是一个有name字段，可选的有url、email字段的对象，像这样：

```
{ "name" : "Barney Rubble"
, "email" : "b@rubble.com"
, "url" : "http://barnyrubble.tumblr.com/"
}
```

或者可以把所有的东西都放到一个字符串里，npm会给你解析：

```
"Barney Rubble <b@rubble.com> (http://barnyrubble.tumblr.com/)
```

email和url在两种形式中都是可选的。

也可以在你的npm用户信息中设置一个顶级的maintainers字段。

files

files是一个包含项目中的文件的数组。如果命名了一个文件夹，那也会包含文件夹中的文件。（除非被其他条件忽略了）

你也可以提供一个.npmignore文件，让即使被包含在files字段中得文件被留下。其实就像.gitignore一样。

main

main字段配置一个文件名指向模块的入口程序。如果你包的名字叫`foo`，然后用户`require("foo")`，main配置的模块的exports对象会被返回。

这应该是一个相对于根目录的文件路径。

对于大多数模块，它是非常有意义的，其他的都没啥。

bin

很多包都有一个或多个可执行的文件希望被放到PATH中。npm让妈妈再也不用担心了（实际上，就是这个功能让npm可执行的）。

要用这个功能，给package.json中的`bin`字段一个命令名到文件位置的map。初始化的时候npm会将他链接到`prefix/bin`（全局初始化）或者`./node_modules/.bin/`（本地初始化）。

比如，npm有：

```
{ "bin" : { "npm" : "./cli.js" } }
```

所以，当你初始化npm，它会创建一个符号链接到`cli.js`脚本到`/usr/local/bin/npm`。

如果你只有一个可执行文件，并且名字和包名一样。那么你可以只用一个字符串，比如：

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin": "./path/to/program" }
```

结果和这个一样：

```
{ "name": "my-program"
, "version": "1.2.5"
, "bin" : { "my-program" : "./path/to/program" } }
```

man

指定一个单一的文件或者一个文件数组供`man`程序使用。

如果只提供一个单一的文件，那么它初始化后就是`man <pkgname>`的结果，而不管实际的文件名是神马，比如：

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : "./man/doc.1"
}
```

这样`man foo`就可以用到`./man/doc.1`文件了。

如果文件名不是以包名开头，那么它会被冠以前缀，下面的：

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
```

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

会为`man foo`和`man foo-bar`创建文件。

man文件需要以数字结束，然后可选地压缩后以`.gz`为后缀。The number dictates which man section the file is installed into.

```
{ "name" : "foo"
, "version" : "1.2.3"
, "description" : "A packaged foo fooer for fooing foos"
, "main" : "foo.js"
, "man" : [ "./man/foo.1", "./man/foo.2" ]
}
```

会为`man foo`和`man 2 foo`创建。

directories

CommonJS Packages规范说明了几种方式让你可以用`directories`hash标示出包得结构。如果看一下npm's package.json，你会看到有directories标示出doc, lib, and man。

在未来，这个信息可能会被用到。

directories.lib

告诉屌丝们你的库文件夹在哪里。目前没有什么特别的东西需要用到lib文件夹，但确实是重要的元信息。

directories.bin

如果你指定一个“bin”目录，然后在那个文件夹中得所有文件都会被当做"bin"字段使用。

如果你已经指定了“bin”字段，那这个就无效。

directories.man

一个放满man页面的文件夹。贴心地创建一个“man”字段。
A folder that is full of man pages. Sugar to generate a "man" array by walking the folder.

directories.doc

将markdown文件放在这里。最后，这些会被很好地展示出来，也许，某一天。
Put markdown files in here. Eventually, these will be displayed nicely, maybe, someday.

directories.example

将事例脚本放在这里。某一天，它可能会以聪明的方式展示出来。

repository

指定你的代码存放的地方。这个对希望贡献的人有帮助。如果git仓库在github上，那么`npm docs`命令能找到你。

这样做：

```
"repository" :
  { "type" : "git"
  , "url" : "http://github.com/isaacs/npm.git"
  }

"repository" :
  { "type" : "svn"
  , "url" : "http://v8.googlecode.com/svn/trunk/"
  }
```

URL应该是公开的（即便是只读的）能直接被未经过修改的版本控制程序处理的url。不应该是一个html的项目页面。因为它是给计算机看的。

scripts

“scripts”是一个由脚本命令组成的hash对象，他们在包不同的生命周期中被执行。key是生命周期事件，value是要运行的命令。

参见 npm-scripts(7)

config

"config" hash可以用来配置用于包脚本中的跨版本参数。在实例中，如果一个包有下面的配置：

```
{ "name" : "foo"
, "config" : { "port" : "8080" } }
```

然后有一个“start”命令引用了`npm_package_config_port`环境变量，用户可以通过`npm config set foo:port 8001`来重写他。

参见 npm-config(7) 和 npm-scripts(7)。

dependencies

依赖是给一组包名指定版本范围的一个hash。这个版本范围是一个由一个或多个空格分隔的字符串。依赖还可以用tarball或者git URL。

请不要将测试或过渡性的依赖放在`dependencies` hash中。见下文的`devDependencies`。

详见semver(7).

- `version` 必须完全和`version`一致
- `>version` 必须比`version`大
- `>=version` 同上
- `<version` 同上
- `<=version` 同上
- `~version` 大约一样，见semver(7)
- `1.2.x` 1.2.0, 1.2.1, 等，但不包括1.3.0
- `http://...` 见下文'依赖URL'
- `*` 所有
- `""` 空，同`*`
- `version1 - version2` 同 `>=version1 <=version2`.
- `range1 || range2` 二选一。
- `git...` 见下文'依赖Git URL'
- `user/repo` 见下文'GitHub URLs'

比如下面都是合法的：

```
{ "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"
  }
}
```

依赖URL

可以指定一个tarball URL，这个tarball将在包被初始化的时候下载并初始化。

依赖Git URL

Git urls 可以是下面几种形式：

```
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` 是可以被 `git checkout` 的任何tag、sha或者branch。默认为 `master` 。

GitHub URLs

1.1.65版后，你可以仅仅用“user/foo-project”引用GitHub urls，比如：

```
{
  "name": "foo",
  "version": "0.0.0",
  "dependencies": {
    "express": "visionmedia/express"
  }
}
```

devDependencies

如果有人要使用你的模块，那么他们可能不需要你开发使用的外部测试或者文档框架。

在这种情况下，最好将这些附属的项目列在 `devDependencies` 中。

这些东西会在执行 `npm link` 或者 `npm install` 的时候初始化，并可以像其他npm配置参数一样管理。详见 npm-config(7)。

对于非特定平台的构建步骤，比如需要编译CoffeeScript，可以用 `prepublish` 脚本去实现，并把它依赖的包放在devDependency中。（译者注：prepublish定义了在执行 `npm publish` 的时候先行执行的脚本）

比如：

```
{ "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publishing前运行，这样用户就不用自己去require来编译就能使用。并且在开发模式中（比如本地运行 `npm install` ）会运行这个脚本以便更好地测试。

peerDependencies

在一些场景中，如在一个host中不必须进行 `require` 时候，你想表现你的package与一个host工具或者库的兼容关键。这一般用来引用_插件_。尤其是你的模块可能要暴露一个特定的接口，并由host文档来预期和指定。

比如：

```
{
  "name": "tea-latte",
  "version": "1.3.5"
  "peerDependencies": {
    "tea": "2.x"
  }
}
```

这能保证你的package可以只和tea的2.x版本一起初始化。 `npm install tea-latte` 可能会产生下面的依赖关系

```
â”œâ”€€â”€ tea-latte@1.3.5
â””â”€â”€ tea@2.2.0
```

试图初始化另一个有会冲突的依赖的插件将导致一个错误。因此，确保你的插件的需求约束越弱越好，而不要去把它锁定到一个特定的版本。

假设这个host遵守semver规范，只改变这个package的主版本会打破你的插件。因此，如果你在package中用过每个1.x版本，就用"^1.0"或者"1.x"来表示。如果你依赖于功能介绍1.5.2，用">= 1.5.2 < 2"。

bundledDependencies

一组包名，他们会在发布的时候被打包进去。

拼成`"bundleDependencies"`（缺d）也可以。

optionalDependencies

如果一个依赖可用，但你希望在它安装错误的时候npm也能继续初始化，那么你可以把它放在`optionalDependencies` hash中。这是一个包名到版本或者url的map，就像`dependencies` hash一样。只是它运行错误。

处理缺乏依赖也是你的程序的责任。比如像这样：

```
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`会覆盖`dependencies`中同名的项，所以通常比只放在一个地方好。

engines

你可以指定工作的node的版本：

```
{ "engines" : { "node" : ">=0.10.3 <0.12" } }
```

并且，像dependensies一样，如果你不指定版本或者指定“*”作为版本，那么所有版本的node都可以。

如果指定一个“engines”字段，那么npm会需要node在里面，如果“engines”被省略，npm会假定它在node上工作。

你也可以用“engines”字段来指定哪一个npm版本能更好地初始化你的程序，如：

```
{ "engines" : { "npm" : "~1.0.20" } }
```

记住，除非用户设置`engine-strict`标记，这个字段只是建议值。

engineStrict

如果你确定你的模块_一定不_会运行在你指定版本之外的node或者npm上，你可以在package.json文件中设置`"engineStrict":true`。它会重写用户的`engine-strict`设置。

除非你非常非常确定，否则不要这样做。如果你的engines hash过度地限制，很可能轻易让自己陷入窘境。慎重地考虑这个选择。如果大家滥用它，它会再以后的npm版本中被删除。

os

你可以指定你的模块要运行在哪些操作系统中：

```
"os" : [ "darwin", "linux" ]
```

你也可以用黑名单代替白名单，在名字前面加上“!”就可以了：

```
"os" : [ "!win32" ]
```

操作系统用`process.platform`来探测。

虽然没有很好地理由，但它是同时支持黑名单和白名单的。

cpu

如果你的代码只能运行在特定的cpu架构下，你可以指定一个：

```
"cpu" : [ "x64", "ia32" ]
```

就像`os`选项，你也可以黑一个架构：

```
"cpu" : [ "!arm", "!mips" ]
```

cpu架构用`process.arch`探测。

preferGlobal

如果包主要是需要全局安装的命令行程序，就设置它为`true`来提供一个warning给只在局部安装的人。

它不会真正的防止用户在局部安装，但如果它没有按预期工作它会帮助防止产生误会。

private

如果你设置`"private": true`，npm就不会发布它。

这是一个防止意外发布私有库的方式。如果你要确定给定的包是只发布在特定registry（如内部registry）的，用`publishConfig`hash的描述来重写`registry`的publish-time配置参数。

publishConfig

这是一个在publish-time使用的配置集合。当你想设置tag或者registry的时候它非常有用，所以你可以确定一个给定的包没有打上“lastest”的tag或者被默认发布到全局的公开registry。

任何配置都可以被重写，但当然可能只有“tag”和“registry”与发布的意图有关。

参见npm-config(7)有可以被重写的列表。

SEE ALSO

- semver(7)
- npm-init(1)
- npm-version(1)
- npm-config(1)
- npm-config(7)
- npm-help(1)
- npm-faq(7)
- npm-install(1)
- npm-publish(1)
- npm-rm(1)

👍 20 ❤️ 5

mingyun commented on 27 Feb 2014

很详细，屌丝明白了

ameizi referenced this issue in **ameizi/DevArticles** on 27 Sep 2014

npm的package.json中文文档 #8 Open