--

[WWW 1](#_Toc5537)

[常用的node模块 2](#_Toc6060)

[第一个demo 2](#_Toc20936)

[hollow world 2](#_Toc32559)

[nodejs里的module.exports和exports的关系 2](#_Toc20089)

[npm Scripts 2](#_Toc23053)

[nodemon 2](#_Toc2859)

[模块加载方式及机制 4](#_Toc31668)

[www 4](#_Toc24602)

[载入内置模块 4](#_Toc3026)

[载入文件模块 4](#_Toc19742)

[载入文件目录模块 4](#_Toc3669)

[载入node\_modules里的模块 4](#_Toc1040)

[自动缓存已载入模块 4](#_Toc23723)

[如何使用npm来管理你的Node.js依赖 5](#_Toc8668)

[npm install 5](#_Toc32136)

[npm install --sa#ve 与 npm install --save-dev 的区别 5](#_Toc14433)

[package.json属性说明 6](#_Toc7019)

[npm link指令 6](#_Toc32760)

[其他 6](#_Toc4099)

[一行命令搞定node.js 版本升级 7](#_Toc12986)

[制作命令行工具 7](#_Toc28218)

[源文件结构 7](#_Toc15424)

[基础步骤 7](#_Toc20050)

[引用及更新 7](#_Toc27294)

[nodejs colors 模块设置控制台文本颜色 7](#_Toc2407)

[其他 8](#_Toc21950)

[node-inspector 8](#_Toc10359)

[所历前端“姿势”更替记(其一)](https://www.jeffjade.com/2016/05/14/106-front-end-learning-record/)

<https://www.jeffjade.com/2016/05/14/106-front-end-learning-record/>

[Vue ES6 Jade Scss Webpack Gulp](https://www.jeffjade.com/2016/05/08/106-vue-es6-jade-scss-webpack-gulp/)

<https://www.jeffjade.com/2016/05/14/106-front-end-learning-record/>

史上最全的前端资源大汇总

<http://www.jianshu.com/p/6cb49271cd2a>

前端开发者github的star整理

<https://github.com/folger-fan/widget/tree/master/github_stars_explain>

阿里云的node文档

<http://alinode.aliyun.com/blog>

《深入理解Node.js：核心思想与源码分析》

<https://yjhjstz.gitbooks.io/deep-into-node/content/>

天猫双11前端

<https://github.com/tmallfe/tmallfe.github.io/issues>

WWW

|  |  |
| --- | --- |
|  | spm:  <https://github.com/spmjs/spm/wiki>  [从几张图来看看越来越难做的前端](http://www.cnblogs.com/chenyablog/p/6390500.html)  <http://www.cnblogs.com/chenyablog/p/6390500.html>  [MDN](https://developer.mozilla.org/zh-CN) [Web 技术文档](https://developer.mozilla.org/zh-CN/docs/Web) [Web API 接口](https://developer.mozilla.org/zh-CN/docs/Web/API) [Canvas](https://developer.mozilla.org/zh-CN/docs/Web/API/Canvas_API) Canvas教程https://developer.mozilla.org/zh-CN/docs/Web/API/Canvas\_API/Tutorial  醉牛前端---前端工具 框架 库 地址汇总  <http://f2er.club/>  十个Node.js开发易犯错误之一:不使用开发工具(自动重启，自动刷新)  <http://www.jdon.com/idea/nodejs/top-10-mistakes-node-developers-make-1.html>  javascript设计模式  <http://www.cnblogs.com/TomXu/tag/JavaScript/default.html?page=2> |
| node学习路线 <http://www.open-open.com/lib/view/open1403574545233.html>  [主页](https://cnodejs.org/)/ Node.js 新手入门  <https://cnodejs.org/getstart>  node学习路线  <http://www.open-open.com/lib/view/open1403574545233.html>  从零开始nodejs系列文章  <http://blog.fens.me/series-nodejs/>  Node.js 包教不包会 <https://github.com/alsotang/node-lessons>  阿里前端node7天 <http://nqdeng.github.io/7-days-nodejs/>  <https://github.com/nqdeng/7-days-nodejs/>  node  <https://toutiao.io/k/96hsla>  [node干货](https://github.com/CGerAJ/node123)  <https://github.com/CGerAJ/node123>  node应用小试  <http://blog.csdn.net/a1104258464/article/category/6149761>  nodejs方向指引：  <https://juejin.im/post/58cf4a3144d90400690b7be7#comment>  <https://zhuanlan.zhihu.com/p/25933892>  使用 Express + MongoDB 搭建多人博客  <https://github.com/nswbmw/N-blog> |
| 作者：韦捷 链接：https://www.zhihu.com/question/55886635/answer/147275692 来源：知乎 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。  如果你已经学会了 HTML CSS JavaScript jQuery，还想要往前一步的话，我推荐：  首先你需要安装 NodeJS 然后看这里（[《一起学 Node.js》彻底重写完毕 - CNode技术社区](https://link.zhihu.com/?target=https://cnodejs.org/topic/581b0c4ebb9452c9052e7acb" \t "_blank)）免费的 NodeJS 教程哎，时间多，可以看完顺便学习后端知识，时间不多，看前三章，看完后有收获，记得给作者打赏，请他喝杯咖啡。接下来马马虎虎就知道 NodeJS 是什么鬼了。  然后学习 Vue 技术栈，现在有许多公司在使用，岗位数量有保证。同时它简单易学，不像 AngularJS 那么重，也没有 React 那么多文绉绉的概念，文档齐全教程丰富，随便在 GitHub 上搜 Vue 都能找到许多实例项目可以参考学习。 先跟着 Vue 官方教程([https://cn.vuejs.org/v2/guide/](https://link.zhihu.com/?target=https://cn.vuejs.org/v2/guide/state-management.html" \t "_blank))走一遍，基本上 Vue 技术栈中常用的几个模块就能一一碰到了，它们分别是 vue-router vuex。  然后当你看完教程想要新建目录准备大干一场的时候，发觉自己什么都不会，别担心，这是人之常情。学过 NodeJS 的你，可以在 5 秒钟内敲出指令，通过 NPM 安装一个 vue-cli （文档地址：[vuejs/vue-cli: Simple CLI for scaffolding Vue.js projects](https://link.zhihu.com/?target=https://github.com/vuejs/vue-cli" \t "_blank)）然后使用它的 webpack 模版（文档地址：[vuejs-templates/webpack: A full-featured Webpack + vue-loader setup with hot reload, linting, testing &amp;amp; css extraction.](https://link.zhihu.com/?target=https://github.com/vuejs-templates/webpack" \t "_blank)）不出三分钟，你就得到了一个配置完善的 Vue 项目，有一整套的基于 Webpack 的构建流程，连目录都给你建好了，你只需在此基础上不断的增加自己的代码。  接着可以在 GitHub 上找一个你喜欢的项目模仿学习，比如在 GitHub 上搜索「vue 知乎日报」就能出现许多相关项目。尽量找新的，并且使用了全套 Vue 技术（Vue、Vue-Router、Vuex、Vue-Resource）的项目来参考。  接着，对着各种文档，边查看边练习吧，当你磕磕碰碰完成一个完整的项目时，你会发觉自己更上一层楼了，找工作也没原来那么难了。再看原来那些 jQuery 代码，你就会发现那是上个时代的产物了。  另外，当你想要让 Webpack 帮你处理更多的东西时你可以看看 Webpack([webpack](https://link.zhihu.com/?target=https://webpack.js.org/" \t "_blank)) 的文档，然后修改 vue-cli 给你生成的 Webpack 的配置代码，你就能尝试配置自己的项目了，顺便学习 Webpack 的使用。 想写 SCSS ？Webpack 有个 scss-loader。 想写 ES6 ？ Webpack 有个 babel-loader。 等等等等……  这题，是越答越歪了....  === 2017.4 更新===  前文提到的 Vue-Resource 作者已经停止维护，替代品是 axios 地址在这 ：[mzabriskie/axios](https://link.zhihu.com/?target=https://github.com/mzabriskie/axios" \t "_blank)  如果 Webpack 一下子不好弄懂，这里有个非常不错的 webpack 中文教程：[Introduction · Webpack for fools](https://link.zhihu.com/?target=https://vikingmute.gitbooks.io/webpack-for-fools/content/" \t "_blank) |
|  |
|  | 学习 Node.js 的书籍：  IMG_256  推荐国内比较专业的 cnode 社区 <https://cnodejs.org，有任何问题都可以发帖提问，会有很多人回答。>  其他就数https://github.com/trending，每日必看。 |
|  | 他给的几个学习建议是： 1、可以去深入学习3D加速、rem、flex和svg。 2、css3动画。 3、canvas，把API整体看一遍，多练习。 4、学习自动化工具，gulp、grunt。 5、多记一下平时碰到的问题，并把解决方案记录下来（这是个好习惯） |
|  | 腾讯SNG雏鹰计划Mini项目杂谈  <http://natumsol.github.io/2016/08/02/mini/> |
|  | [browserify使用手册](http://www.cnblogs.com/liulangmao/p/4920534.html)  <http://www.cnblogs.com/liulangmao/p/4920534.html?utm_source=tuicool&utm_medium=referral> |
|  | 知识点教程：  <http://www.gimoo.net/t/1412/54a24d71ae74f.html>  nodeEN：  <https://nodejs.org/api/path.html#path_path_extname_path>  nodeCH:  <http://nodeapi.ucdok.com/api/>  菜鸟教程：  <http://www.runoob.com/nodejs/nodejs-restful-api.html> |
|  | 初探nodejs学习笔记- 如何使用nodejs接收get/post 数据（参数）  <http://my.oschina.net/freddon/blog/513853> |
|  | [web界面上的字体兼容方案](https://www.baidufe.com/item/60cd11d3bfdee5c51369.html" \t "https://www.baidufe.com/_blank)  <https://www.baidufe.com/item/60cd11d3bfdee5c51369.html> |
|  | 公共组件库  <https://www.baidufe.com/component> |
|  | Node.js使用cheerio抓取网页数据DEMO  <http://www.sufeinet.com/thread-8413-1-1.html> |
|  | uglifyjs压缩  <http://jiangxiao-2000.iteye.com/blog/1941144>  <http://www.zhangxinxu.com/wordpress/2013/01/uglifyjs-compress-js/>  <http://www.cnblogs.com/sorex/archive/2011/12/14/2287635.html> |
|  | 使用nrm控制npm源  <https://chenjiangsong.github.io/blog/2016/03/13/%E4%BD%BF%E7%94%A8nrm%E6%8E%A7%E5%88%B6npm%E6%BA%90/>  [前端自动化构建和发布系统的设计](http://stylechen.com/static-build.html" \o "前端自动化构建和发布系统的设计（一）)  <http://stylechen.com/category/node-js>  10个Node.js开发者最易犯的错误  <https://zhuanlan.zhihu.com/p/19944110?columnSlug=FrontendMagazine> |
|  | Gulp应该和Grunt比较，他们的区别我就不说了，说说用处吧。Gulp / Grunt 是一种工具，能够优化前端工作流程。比如自动刷新页面、combo、压缩css、js、编译less等等。简单来说，就是使用Gulp/Grunt，然后配置你需要的插件，就可以把以前需要手工做的事情让它帮你做了。  说到 browserify / webpack ，那还要说到 seajs / requirejs 。这四个都是JS模块化的方案。其中seajs / require 是一种类型，browserify / webpack 是另一种类型。  seajs / require : 是一种在线"编译" 模块的方案，相当于在页面上加载一个 CMD/AMD 解释器。这样浏览器就认识了 define、exports、module 这些东西。也就实现了模块化。  browserify / webpack : 是一个预编译模块的方案，相比于上面 ，这个方案更加智能。没用过browserify，这里以webpack为例。首先，它是预编译的，不需要在浏览器中加载解释器。另外，你在本地直接写JS，不管是 AMD / CMD / ES6 风格的模块化，它都能认识，并且编译成浏览器认识的JS。  这样就知道，Gulp是一个工具，而webpack等等是模块化方案。Gulp也可以配置seajs、requirejs甚至webpack的插件。 |
| 常用的node模块 | socket.io ：作用为客户端和服务器端的双向实时通信；  uglify-js ：作用为js文件的压缩；  clean-css ：作用为css文件的压缩；  node-smushit ：作用为图片文件的压缩；  walk ：作用为分析磁盘的目录结构；  node.js的前100个流行模块  <http://blog.csdn.net/kavensu/article/details/17890503> |
|  |  |

第一个demo

|  |  |
| --- | --- |
| hollow world | ---新建server.js  var http = require("http");  http.createServer(function(request, response) {  response.writeHead(200, {"Content-Type": "text/plain"});//返回的状态，返回的文本类型  response.write("Hello World");  response.end();//response.end("hellow world")  }).listen(8888);//.listen(1337,'127.0.0.1')  ---http://localhost:8888/ |
|  | =node  进入node环境变量  =F12  进入调试  区别：window process |

[nodejs里的module.exports和exports的关系](http://www.cnblogs.com/liulangmao/p/4901665.html)

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  | | Node.js模块里exports与module.exports的区别?  https://www.zhihu.com/question/26621212/answer/33432974  Module.exports和exports的区别---详细  https://www.ycjcl.cc/2017/02/10/module-exportshe-exportsde-qu-bie/ | |
|  | //下面这种情况下是一致的:  exports.foo=123 //可以是字符串,数字,数组或函数,都无所谓  module.exports.foo=123 //可以是字符串,数字,数组或函数,都无所谓  //下面这种情况下是不一致的:  module.exports = 123 //可以是字符串,数字,数组或函数,都无所谓  exports = 123 //可以是字符串,数字,数组或函数,都无所谓  至于为什么会这样,我一直都不太清楚,直到看到下面这段代码:  var module = {  exports: {  }  };  (function(module,exports){  module.exports = 123; //可以修改module对象下的exports属性  　　　　 exports = 123;　　　　　　//修改的是当前函数环境里的exports变量,而非module.exports  　　　　 exports.foo = 123 //可以修改module.exports下的foo属性  })(module,module.exports);  这个函数里的参数module和exports就对应了nodejs里的module和exports,module和module.exports都作为参数被传递给了函数,所以,当我们修改参数的属性的时候,它对于参数指向的这个对象是有效的,而直接修改这个参数,只是在函数环境里修改这个变量,对象并没有被修改.  这就解释了为什么可以直接给module.exports赋值,却不能给exports直接赋值.  也解释了为什么module.exports.foo和exports.foo是完全一样的. | |
|  | 改变不了 exports 的引用。准确来说，是改变后实际导出的还是 exports 原引用指向的对象。module.exports 就是用来修复此问题的。 如果我们把你的 JS 文件整个放在一个闭包内执行： define(**function**(require, exports) {  exports = **function**() {}; }); exports 的引用改变外部根本观察不到。但如果换成 module.exports： define(**function**(require, exports, module) {  module.exports = **function**() {}; }); 这时候 module 对内外的观察者来说都是同一个东西，在内部改变了 module.exports 在外部能获取到。 | |
|  | 用一句话来说明就是，require方能看到的只有module.exports这个对象，它是看不到exports对象的，而我们在编写模块时用到的exports对象实际上只是对module.exports的引用。  首先说一个概念： ECMAScript的变量值类型共有两种： 基本类型 (primitive values) ： 包括Undefined, Null, Boolean, Number和String五种基本数据类型 引用类型 (reference values) ： 保存在内存中的对象们，不能直接操作，只能通过保存在变量中的地址引用对其进行操作。 我们今天要讨论的exports和module.exports属于Object类型，属于引用类型。 看下面的例子： **var module** = {  exports:{  name:"我是module的exports属性"  } }; **var exports** = **module**.exports; //exports是对module.exports的引用，也就是exports现在指向的内存地址和module.exports指向的内存地址是一样的  console.log(**module**.exports); // { name: '我是module的exports属性' } console.log(**exports**); // { name: '我是module的exports属性' }  **exports**.name = "我想改一下名字";  console.log(**module**.exports); // { name: '我想改一下名字' } console.log(**exports**); // { name: '我想改一下名字' } //看到没，引用的结果就是a和b都操作同一内存地址下的数据   **/\*修改方式1：这个时候我在某个文件定义了一个想导出的模块\*/ var Circle** = {  name:"我是一个圆",  func:**function**(x){  **return** x\*x\*3.14;  } };  **exports** = **Circle**; // 看清楚了，Circle这个Object在内存中指向了新的地址，所以exports也指向了这个新的地址，和原来的地址没有半毛钱关系了  console.log(**module**.exports); // { name: '我想改一下名字' } console.log(**exports**); // { name: '我是一个圆', func: [Function] }   回到nodejs中，**module**.exports初始的时候置为{},exports也指向这个空对象。  **/\*修改方式2\*/ exports**.name = **function**(x){  console.log(x); };  **/\*修改方式3：下面这个一毛一样，因为都是修改的同一内存地址里的东西\*/ module**.exports.name = **function**(x){  console.log(x); }; **/\*但是这样写就有了区别了\*/ exports** = **function**(x){  console.log(x); };  //上面的 function是一块新的内存地址，导致exports与module.exports不存在任何关系，而require方能看到的只有module.exports这个对象，看不到exports对象，所以这样写是导不出去的。  //下面的写法是可以导出去的。说句题外话，module.exports除了导出对象，函数，还可以导出所有的类型，比如字符串、数值等。 **module**.exports = **function**(x){  console.log(x); }; | |
| 示例 | 从require导入方式去理解，关键有两个变量(全局变量module.exports，局部变量exports)、一个返回值(module.exports)  function require(...) {  var module = {  exports: {}  };  ((module, exports) => {  // --- 你的被引入代码 Start  // var exports = module.exports = {}; (默认都有的)  exports = function () {};  // 此时，exports不再挂载到module.exports；require() 将导出{}默认对象  module.exports = function () {};  // 此时，这个模块将导出some\_func对象，覆盖exports上的some\_func  // --- 你的被引入代码 End  })(module, module.exports);  // 不管是exports还是module.exports，最后返回的还是module.exports  return module.exports;  } | |
| require 结果 | exports = function () {}  // {} | |
| module = function () {}  // {} | |
| exports.a = function () {  console.log(1);  }  //{ a: [Function] } | |
| module.exports.a = function () {  console.log(1);  }  // { a: [Function] } | |
| module.exports = {  b: function () {  console.log(2)  }  }  //{ b: [Function: b] } | |

save save-dev

|  |  |
| --- | --- |
|  | --save 和 --save-dev的区别  --save是对生产环境所需依赖的声明(开发应用中使用的框架，库，比如jquery，bootstrap等)  --save-dev是对开发环境所需依赖的声明(构建工具，测试工具，比如babel，gulp等)  正常使用npm install时，会下载dependencies和devDependencies中的模块  当使用npm install --production或者注明NODE\_ENV变量值为production时，只会下载dependencies中的模块 |

npm Scripts

|  |  |
| --- | --- |
|  | 为什么使用 npm Scripts 构建项目  <http://www.css88.com/archives/7308>  [前端小丑](http://www.cnblogs.com/webxiaochou/)-[npm命令及npm scripts的简单总结](http://www.cnblogs.com/webxiaochou/p/6664906.html)  <http://www.cnblogs.com/webxiaochou/p/6664906.html>  npm中文文档  <https://www.kancloud.cn/shellway/npm-doc/199981> |
|  |  |
|  |  |
|  |  |
|  |  |

nodemon

|  |  |
| --- | --- |
| 简单使用 | <https://github.com/remy/nodemon>  npm install -g nodemon  在doc下输入上面的，就安装成功了，以后我们启动我们的项目的时候，就输入nodemon server.js  以前是node server.js  这样，每次我们的js文件修改后，node就会自动重启，就省的我们手动去重启了，这样方便了很多。  还有注意一下，这里有一个坑：我一直使用下面的方法启动node: node  这样启动是没有问题的，因为环境变量中配置了node了嘛，但是如果nodemon d:/js/node/app.js  这样nodemon监视的是启动目录，而不是d:/js/node这个目录，比如 c:\program>nodemon d:/js/node/app.js 他实际上监视的是c:\program这个目录，你需要修改启动目录里才可以。 |
| 3）如果想通过 npm start 命令来启动应用同时又想用 nodemon 来监控文件改动，可以修改 npm 的 package.js 文件中的 scripts.start：  1 "scripts": {2 "start": "nodemon ./bin/www"3 } |
| nodemon({  script: ...,  stdout: false // important: this tells nodemon not to output to console  }).on('readable', function() { // the `readable` event indicates that data is ready to pick up  this.stdout.pipe(fs.createWriteStream('output.txt'));  this.stderr.pipe(fs.createWriteStream('err.txt'));  }); |
| <https://github.com/JacksonGariety/gulp-nodemon>  $ npm install --save-dev gulp-nodemon  Example below will start server.js in development mode and watch for changes, as well as watch all .html and .js files in the directory.  gulp.task('start', function () {  nodemon({  script: 'server.js',  ext: 'js html',  env: { 'NODE\_ENV': 'development' }  })  }) |
|  | [nodemon 基本配置与使用](http://www.cnblogs.com/JuFoFu/p/5140302.html)  <http://www.cnblogs.com/JuFoFu/p/5140302.html> |
|  | 在开发环境下，往往需要一个工具来自动重启项目工程，之前接触过 python 的 supervisor，现在写 node 的时候发现 supervisior 在很多地方都有他的身影，node 也有一个 npm 模块 [supervisior](https://github.com/petruisfan/node-supervisor) 也是用来监控进程的，不过除了 supervisior 外，还有很多其他的工具，从 github 的评分上看，比较热门的有 forever，nodemon，node-dev，具体这些工具的区别可以参考这篇文章 [Comparison: Tools to Automate Restarting Node.js Server After Code Changes](https://strongloop.com/strongblog/comparison-tools-to-automate-restarting-node-js-server-after-code-changes-forever-nodemon-nodesupervisor-nodedev/)，个人觉得在开发环境还是用 nodemon，因为配置比较方便，文档也很清晰。所以这里先主要讲 nodemon。  nodemon 的安装：  npm install -g nodemon  安装完 nodemon 后，就可以用 nodemon 来代替 node 来启动应用：  nodemon [your node app]（相当于 node [your node app]）  如果没有在应用中指定端口，可以在命令中指定：  nodemon ./server.js localhost 8080  可以运行 debug 模式：  nodemon --debug ./server.js 80  查看帮助，帮助里面有很多选项都是一目了然：  nodemon -h 或者 nodemon -help  nodemon 比较流行的原因之一就是它的可配置性比较高，下面是官网给出的配置文件 [nodemon.json](https://github.com/remy/nodemon/blob/master/doc/sample-nodemon.md) 的例子，加上我自己了解到的有用的一些配置，开发环境建议可以把每个参数都写上备用，生产环境就把没有必要的参数去掉，有些字段是可以在命令行模式以参数形式给出的，可以通过 -h 查看，下面逐个解释： |
|  | {  "restartable": "rs",  "ignore": [  ".git",  "node\_modules/\*\*/node\_modules"  ],  "verbose": true,  "execMap": {  "": "node"  "js": "node --harmony"  },  "events": {  "restart": "osascript -e 'display notification \"App restarted due to:\n'$FILENAME'\" with title \"nodemon\"'"  },  "watch": [  "test/fixtures/",  "test/samples/"  ],  "env": {  "NODE\_ENV": "development",  "PORT": "3000"  },  "ext": "js json",  "legacy-watch": false  } |
|  | restartable：重启的命令，默认是 rs ，可以改成你自己喜欢的字符串。当用 nodemon 启动应用时，可以直接键入 rs 直接重启服务。除了字符串值外，还可以设置 false 值，这个值的意思是当 nodemon 影响了你自己的终端命令时，设置为 false 则不会在 nodemon 运行期间监听 rs 的重启命令。  ignore：忽略的文件后缀名或者文件夹，文件路径的书写用相对于 nodemon.json 所在位置的相对路径，下同。nodemon 会默认忽略一些文件，默认忽略的文件有：.git, node\_modules, bower\_components, .sass-cache，如果这些文件想要加入监控，需要重写默认忽略参数字段 ignoreRoot，比如加入："ignoreRoot": [".git", "bower\_components", ".sass-cache"]，然后在 watch 中将 node\_modules 文件路径加入监控，那么 node\_modules 内的文件也加入了监控了。  verbose：true 表示输出详细启动与重启信息，如下图：  IMG_256  false 表示不输出这些运行信息，如下图：  IMG_257  execMap：运行服务的后缀名和对应的运行命令，"js": "node --harmony" 表示用 nodemon 代替 node  --harmony 运行 js 后缀文件；"" 指 www 这些没有后缀名的文件；默认的 defaults.js 配置文件会识别一些文件：py: 'python',rb: 'ruby'。  events：这个字段表示 nodemon 运行到某些状态时的一些触发事件，总共有五个状态：  start - 子进程（即监控的应用）启动  crash - 子进程崩溃，不会触发 exit  exit - 子进程完全退出，不是非正常的崩溃  restart - 子进程重启  config:update - nodemon 的 config 文件改变  状态后面可以带标准输入输出语句，比如 mac 系统下设置： "start": "echo 'app start'"，那么启动应用时会输出 app start 信息，其他类似命令如 ls，ps 等等标准命令都可以在这里定义。除此之外，也可以写js来监控，github 上有介绍： [events.md](https://github.com/remy/nodemon/blob/master/doc/events.md)，不过我试过之后没有成功，如果有懂的朋友，记得在评论不吝赐教。^\_^  watch：监控的文件夹路径或者文件路径。  env：运行环境 development 是开发环境，production 是生产环境。port 是端口号。  ext：监控指定后缀名的文件，用空格间隔。默认监控的后缀文件：.js, .coffee, .litcoffee, .json。但是对于没有文件后缀的文件，比如 www 文件，我暂时找不到怎么用 nodemon 去监控，就算在 watch 中包含了，nodemon 也会忽略掉。  注：关于监控以及忽略文件修改有个顺序的问题，或者说优先级，首先 nodemon 会先读取 watch 里面需要监控的文件或文件路径，再从文件中选择监控 ext 中指定的后缀名，最后去掉从 ignore 中指定的忽略文件或文件路径。  legacy-watch：nodemon 使用 [Chokidar](https://www.npmjs.com/package/chokidar) 作为底层监控系统，但是如果监控失效，或者提示没有需要监控的文件时，就需要使用轮询模式（polling mode），即设置 legacy-watch 为 true，也可以在命令行中指定：  $ nodemon --legacy-watch  $ nodemon -L # 简写 |
|  | nodemon 的默认配置文件 [default.js](https://github.com/remy/nodemon/blob/master/lib/config/defaults.js)： |
|  | // default options for config.options  module.exports = {  restartable: 'rs',  colours: true,  execMap: {  py: 'python',  rb: 'ruby',  // more can be added here such as ls: lsc - but please ensure it's cross  // compatible with linux, mac and windows, or make the default.js  // dynamically append the `.cmd` for node based utilities  },  ignoreRoot: ['.git', 'node\_modules', 'bower\_components', '.sass-cache'],  watch: ['\*.\*'],  stdin: true,  runOnChangeOnly: false,  verbose: false,  // 'stdout' refers to the default behaviour of a required nodemon's child,  // but also includes stderr. If this is false, data is still dispatched via  // nodemon.on('stdout/stderr')  stdout: true,  }; |
|  | 有几个比较少用到的配置字段：  1）colous：输出信息颜色标示。  2）runOnChangeOnly：true 时运行 nodemon www 项目不会启动，只保持对文件的监控，当监控的文件有修改并保存时才会启动应用，其他没有影响。默认是 false 即一开始就启动应用并监控文件改动。  3）stdin，stdout：这个是关于标准输入输出的设置，上文提到 nodemon.json 文件中的 events 字段可以为状态设置标准输入输出语句，如果这里设置了 false，标准输入输入语句就会实效。 |
|  | github 上给出了一个[faq.js](https://github.com/remy/nodemon/blob/master/faq.md) 解答了一些常见的问题，有的上文已经提到，还有一些比较常见的列举如下：  1）当自己的应用启动服务带的参数和 nodemon 冲突时，可以利用下面的方法来解决冲突：  $ nodemon app.js -- -L -opt2 -opt3  以 -- 为分隔，nodemon 不会去读取 -- 后面的参数，而是传给 app.js。  2）当应用因为某些原因奔溃时，nodemon 不会自动重启，会输出以下信息：  [nodemon] app crashed - waiting for file changes before starting...  这个时需要修改文件并保存后 nodemon 才会重启应用，这在开发环境没什么关系，但是如果想把 nodemon 放在线上时，我们往往希望 nodemon 能够自动重启崩溃的应用，这个时候就需要 forever 来辅助了，有一个[issue](https://github.com/remy/nodemon/issues/30) 专门讲这个问题。使用 forever 来重启 nodemon 时，在 nodemon 启动时需要加个参数 --exitcrash：  nodemon www --exitcrash  这样当应用崩溃后，nodemon 会自动中断退出，forever 检测到 nodemon 退出后就会重启 nodemon，nodemon 又会重启应用。其他就是 forever 的配置了，因为这里只讲 nodemon，所以就不涉及 forever，到时候总结 forever 的时候再讲，感兴趣的看[这里](https://github.com/remy/nodemon/blob/master/faq.md" \l "using-nodemon-with-forever)。  3）如果想通过 npm start 命令来启动应用同时又想用 nodemon 来监控文件改动，可以修改 npm 的 package.js 文件中的 scripts.start：  "scripts": {  "start": "nodemon ./bin/www"  }  那么用 npm start 启动后就是执行 nodemon ./bin/www。 |
|  | 还有注意一下，这里有一个坑：我一直使用下面的方法启动node: node  这样启动是没有问题的，因为环境变量中配置了node了嘛，但是如果nodemon d:/js/node/app.js  这样nodemon监视的是启动目录，而不是d:/js/node这个目录，比如 c:\program>nodemon d:/js/node/app.js  他实际上监视的是c:\program这个目录，你需要修改启动目录里才可以。 |
|  | supervisor |

### 模块加载方式及机制

|  |  |
| --- | --- |
| www | <http://www.cnblogs.com/snandy/p/3445550.html> |
|  | Node.js中模块可以通过文件路径或名字获取模块的引用。模块的引用会映射到一个js文件路径，除非它是一个Node内置模块。Node的内置模块公开了一些常用的API给开发者，并且它们在Node进程开始的时候就预加载了。    其它的如通过NPM安装的第三方模块（third-party modules）或本地模块（local modules），每个模块都会暴露一个公开的API。以便开发者可以导入。如  var mod = require('module\_name')  此句执行后，Node内部会载入内置模块或通过NPM安装的模块。require函数会返回一个对象，该对象公开的API可能是函数，对象，或者属性如函数，数组，甚至任意类型的JS对象。    这里列下node模块的载入及缓存机制  1.载入内置模块（A Core Module）  2.载入文件模块（A File Module）  3.载入文件目录模块（A Folder Module）  4.载入node\_modules里的模块  5.自动缓存已载入模块 |
| 载入内置模块 | Node的内置模块被编译为二进制形式，引用时直接使用名字而非文件路径。当第三方的模块和内置模块同名时，内置模块将覆盖第三方同名模块。因此命名时需要注意不要和内置模块同名。如获取一个http模块:  var http = require('http')  返回的http即是实现了HTTP功能Node的内置模块。 |
| 载入文件模块 | 绝对路径:  var myMod = require('/home/base/my\_mod')  相对路径:  var myMod = require('./my\_mod')  注意，这里忽略了扩展名“.js”，以下是对等的  var myMod = require('./my\_mod')  var myMod = require('./my\_mod.js') |
| 载入文件目录模块 | 可以直接require一个目录，假设有一个目录名为folder，如  var myMod = require('./folder')  此时，Node将搜索整个folder目录，Node会假设folder为一个包并试图找到包定义文件package.json。如果folder目录里没有包含package.json文件，Node会假设默认主文件为index.js，即会加载index.js。如果index.js也不存在，那么加载将失败。   |  |  | | --- | --- | | 目录结构如下 | package.json定义如下    {  "name": "pack",  "main": "modA.js"  } |   此时 require('./folder') 将返回模块modA.js。  如果package.json不存在，那么将返回模块index.js。  果index.js也不存在，那么将发生载入异常。 |
| 载入node\_modules里的模块 | 如果模块名不是路径，也不是内置模块，Node将试图去当前目录的node\_modules文件夹里搜索。如果当前目录的node\_modules里没有找到，Node会从父目录的node\_modules里搜索，这样递归下去直到根目录。  不必担心，npm命令可让我们很方便的去安装，卸载，更新node\_modules目录。 |
| 自动缓存已载入模块 | 对于已加载的模块Node会缓存下来，而不必每次都重新搜索。下面是一个示例  modA.js  console.log('模块modA开始加载...')  exports = function() {      console.log('Hi')  }  console.log('模块modA加载完毕')  init.js  var mod1 = require('./modA')  var mod2 = require('./modA')  console.log(mod1 === mod2)  命令行执行：  node init.js  IMG_256  可以看到虽然require了两次，但modA.js仍然只执行了一次。mod1和mod2是相同的，即两个引用都指向了同一个模块对象。 |

如何使用npm来管理你的Node.js依赖

|  |  |
| --- | --- |
|  | 如何使用NPM来管理你的Node.js依赖  <http://www.infoq.com/cn/articles/msh-using-npm-manage-node.js-dependence>    npm config  <https://docs.npmjs.com/misc/config> |
| npm install | 安装模块只需要 npm install express connect 命令给我们带来了很大的方便。安装模块的路径分两种：  全局路径，也就是带上参数 -g 的安装模式。这个命令会把模块安装在 $PREFIX/lib/node\_modules 下，可通过命令 npm root -g 查看全局模块的安装目录。 package.json 里定义的bin会安装到 $PREFIX/bin 目录下，如果模块带有 man page 会安装到 $PREFIX/share/man 目录下。  本地路径，不带 -g 参数的。从当前目录一直查找到根目录/下有没有 node\_modules 目录，有模块安装到这个目录下的 node\_modules 目录里，如果没有找到则把模块安装到当前目录 node\_modules 目录下。package.josn 定义的 bin 会安装到 node\_modules/.bin 目录下，man page 则不会安装。  我们需要选择什么样的安装方式呢？全局模式可以让你不用担心找不到模块，如果不需要还是尽量避免全局模式。  如果我们只是 require('pkg') 一个模块，我们不需要使用全局模式。  如果我们需要在命令行中调用，我们需要使用全局模式。因为这个安装把 package.josn里 bin 下的定义安装到 $PATH 目录下。  有些模块我们既需要在命令行中调用又想 require('pkg') ，那么我们可以使用全局模式安装，然后使用下一段要讲的命令 npm link 把它链接到本地的 node\_modules 目录下。  不要担心 package.josn 里 script 中定义的命令会不会因为不是全局安装而不能运行。比如在例子里定义的 devDependencies 的 vows 。在调用 npm test 时 npm 会把 node\_modules/.bin 目录放到环境变量 $PATH 的最前面。 |
| [npm install --sa#ve 与 npm install --save-dev 的区别](http://www.cnblogs.com/hollen/p/5956012.html) | 以npm安装msbuild为例：  npm install msbuild:  会把msbuild包安装到node\_modules目录中  不会修改package.json  之后运行npm install命令时，不会自动安装msbuild  npm install --save:  会把msbuild包安装到node\_modules目录中  会在package.json的dependencies属性下添加msbuild  之后运行npm install命令时，会自动安装msbuild到node\_modules目录中  之后运行npm install --production或者注明NODE\_ENV变量值为production时，会自动安装msbuild到node\_modules目录中  npm install --save-dev:  会把msbuild包安装到node\_modules目录中  会在package.json的devDependencies属性下添加msbuild  之后运行npm install命令时，会自动安装msbuild到node\_modules目录中  之后运行npm install --production或者注明NODE\_ENV变量值为production时，不会自动安装msbuild到node\_modules目录中  使用原则:  运行时需要用到的包使用--save，否则使用--save-dev。  --save-dev 是你开发时候依赖的东西，--save 是你发布之后还依赖的东西。  比如，你写 ES6 代码，如果你想编译成 ES5 发布那么 babel 就是devDependencies。 如果你用了 jQuery，由于发布之后还是依赖jQuery，所以是dependencies。  但是在 npm 里面除了二进制的依赖，似乎也不用区分是不是dev。  因为使用npm就是自己编译的意思，而不使用npm直接拿编译后的版本的，这些依赖项也看不到。 |
|  |  |
|  | npm install 在安装 npm 包时，有两种命令参数可以把它们的信息写入 package.json 文件，一个是npm install --save另一个是 npm install --save-dev，他们表面上的区别是--save 会把依赖包名称添加到 package.json 文件 dependencies 键下，--save-dev 则添加到 package.json 文件 devDependencies 键下，譬如：  {  "name": "yo",  "version": "0.0.0",  "dependencies": {},  "devDependencies": {  "grunt": "~0.4.1",  "grunt-contrib-copy": "~0.4.1",  "grunt-contrib-concat": "~0.3.0",  "grunt-contrib-uglify": "~0.2.0",  "grunt-contrib-compass": "~0.7.0",  "grunt-contrib-jshint": "~0.7.0",  "grunt-contrib-cssmin": "~0.7.0",  } }  不过这只是它们的表面区别。它们真正的区别是，devDependencies 下列出的模块，是我们开发时用的，比如 我们安装 js的压缩包gulp-uglify 时，  我们采用的是 npm install --save-dev gulp-uglify （见上图）命令安装，因为我们在发布后用不到它，而只是在我们开发才用到它。dependencies 下的模块，则是我们发布后还需要依赖的模块，譬如像jQuery库或者Angular框架类似的，我们在开发完后后肯定还要依赖它们，否则就运行不了。  另外需要补充的是：  正常使用npm install时，会下载dependencies和devDependencies中的模块，当使用npm install --production或者注明NODE\_ENV变量值为production时，只会下载dependencies中的模块。 |
|  | 区别：  在生产环境，使用npm install --production安装 dependencies 部分的模块  在开发环境，npm i ，安装所有devDependencies 和 dependencies里面的模块  所以，在生产环境，就不需要安装 gulp ,webpack 这样的模块，可以通过这种方法区分  几种简写指令：  -P 相当于 --save-prod， 添加dependencies 里面所有的包。在 -D -O 不存在时，-P 就是默认值  -S 相当于 --save; 添加dependencies 里面所有的包。  -D 相当于 --save-dev; 添加devDependencies 里面所有的包，  -O 相当于 --save-optional, 添加在 optionalDependencies 里面的包，  --no-save： 阻止保存记录在dependencies 中，  合理使用这两个依赖记录，可以保证减少在生产环境下，打包后体积大小。  <https://segmentfault.com/a/1190000010675594> |
| 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 字段是一个模块ID，它是一个指向你程序的主要项目。就是说，如果你包的名字叫 express，然后用户安装它，然后require("express")。  keywords - 关键字 |
|  |  |
| npm link指令 | 对开发者而言，这算是最有价值的命令。假设我们开发了一个模块叫 test，然后我们在 test-example 里引用这个模块 ，每次 test 模块的变动我们都需要反映到 test-example 模块里。不要担心，有了 npm link 命令一切变的非常容易。  首先我们需要把 test 链接到全局模式下：  cd ~/work/node/test # 进入test模块目录  npm link # 创建链接到$PREFIX/lib/node\_modules  那么 test 的模块将被链接到 $PREFIX/lib/node\_modules 下，就像我的机器上 $PREFIX 指到 /usr/local ，那么 /usr/local/lib/node\_modules/test 将会链接到 ~/work/node/test 下。  执行脚本 bin/test.js 被链接到 /usr/local/bin/test 上。  接下来我们需要把 test 引用到 test-example 项目中来：  cd ~/work/node/test-example # 进入test-example模块目录 |
|  | npm link test # 把全局模式的模块链接到本地  npm link test 命令会去 $PREFIX/lib/node\_modules 目录下查找名叫 test 的模块，找到这个模块后把 $PREFIX/lib/node\_modules/test 的目录链接到 ~/work/node/test-example/node\_modules/test 这个目录上来。  现在任何 test 模块上的改动都会直接映射到 test-example 上来。再比如假设我们开发很多应用，每个应用都用到 Coffee-script ：  npm install coffee-script -g # 全局模式下安装coffee-script  cd ~/work/node/test # 进入开发目录  npm link coffee-script # 把全局模式的coffee-script模块链接到本地的node\_modules下  cd ../test-example # 进入另外的一个开发目录  npm link coffee-script # 把全局模式的coffee-script模块链接到本地  npm update coffee-script -g # 更新全局模式的coffee-script，所有link过去的项目同时更新了。  就像你看到，npm link 对于开发时一个模块被多个模块引用时非常有用。 |
| windows 的用户会想，我这儿没有 UNIX 下的 link 工具怎么办？别担心只要你的 Node.js 支持 fs.symlink 就可用到这个特性。  原理  linux的系统下会自动执行ln -s 命令来创建一个软连接指向你的全局包路径 |
| npm link创建全局链接(不支持windows)  举个例子，我们已经通过 npm install -g express 安装了 express，这时在工程的目录下运行命令：  npm link express ./node\_modules/express -> /usr/local/lib/node\_modules/express  我们可以在 node\_modules 子目录中发现一个指向安装到全局的包的符号链接。通过这种方法，我们就可以把全局包当本地包来使用了。 |
| 其他 | 官方文档  <https://docs.npmjs.com/>  菜鸟教程NPM 使用介绍  <http://www.runoob.com/nodejs/nodejs-npm.html>  卸载 Node.js 模块  $ npm uninstall express  查看  $ npm ls  更新模块：  $ npm update express  搜索模块：  $ npm search express  查看npm配置  npm config list //查看基本配置  npm config list -l //查看所有配置  npm本地安装与全局安装  npm install grunt // 本地安装，则是将模块下载到当前命令行所在目录 npm install -g grunt//全局安装，模块将被下载安装到【全局目录】中  npm如何获取全局安装的默认目录  npm config get prefix  npm如何设置全局安装的默认目录  npm config set prefix “directory”  查看自己全局安装过的包，用命令  npm list -g --depth 0    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。     1. 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 initThis 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 fieldsand 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 adduserUsername: mcmohdPassword: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。 |
|  | npm config list/ls 显示配置信息 npm config list/ls -l 更详细  npm config set prefix path 修改npm全局安装目录 npm config set cache path 修改npm cache目录；把新的npm路径修改到系统环境变量中，才可生效 npm config set registry='https://registry.npm.taobao.org/' 设置npm资源镜像，加快下载安装速度，还有其他镜像  npm -h 显示帮助信息，建议多查看 npm -l display full usage info ;-l is --long npm <cmd> -h 显示某个命令的帮助信息 npm help npm npm help <term>  npm ls 显示工程目录下本地安装的包，--depth=0,显示初级依赖层次 npm -g ls 显示全局安装的包  npm i/install --save xxx 安装包信息将加入到dependencies（生产阶段的依赖） npm i/install --save-dev xxx 安装包信息将加入到devDependencies（开发阶段的依赖），所以开发阶段一般使用它 npm i --save-exact xxx 精确安装包版本，package.json里的依赖包的value是具体的版本号，前边没有符号  npm root 查看包的安装路径,及node\_modules的路径  npm view modulename 查看模块的package.json信息 npm view moudleName dependencies 查看包的依赖关系 npm view moduleName repository.url 查看包的源文件地址 npm view moduleName engines 查看包所依赖的node版本  npm outdated 检测显示过时的包  npm update 更新包  npm uninstall 卸载包  npm init 在项目中引导创建一个package.json文件 |

一行命令搞定node.js 版本升级

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | <http://www.jb51.net/article/52409.htm>  今天，又发现一个超级简单的升级node.js的方法。一行命令搞定，省去了重新编译安装的过程。  node有一个模块叫n（这名字可够短的。。。），是专门用来管理node.js的版本的。  首先安装n模块：   |  |  | | --- | --- | | 1 | npm install -g n |   第二步：  升级node.js到最新稳定版   |  |  | | --- | --- | | 1 | n stable |   是不是很简单？！  n后面也可以跟随版本号比如：   |  |  | | --- | --- | | 1 | n v0.10.26 |   或   |  |  | | --- | --- | | 1 | n 0.10.26 |   另外分享几个npm的常用命令   |  |  | | --- | --- | |  | npm -v          #显示版本，检查npm 是否正确安装。    npm install express   #安装express模块    npm install -g express  #全局安装express模块    npm list         #列出已安装模块    npm show express     #显示模块详情    npm update        #升级当前目录下的项目的所有模块    npm update express    #升级当前目录下的项目的指定模块    npm update -g express  #升级全局安装的express模块    npm uninstall express  #删除指定的模块 | |
|  |  |

制作命令行工具

|  |  |
| --- | --- |
|  | [Nodejs 制作命令行工具](https://segmentfault.com/a/1190000002918295)-Commander-yargs  <https://segmentfault.com/a/1190000002918295>  用node.js开发命令行工具-简单-显示文件目录  <http://binbinliao.com/programming/commandline-nodejs.html>  极客学院：[首页](http://www.jikexueyuan.com/) > [职业课程库](http://www.jikexueyuan.com/course/) > [Node.js 基础](http://www.jikexueyuan.com/course/nodejs/) > [使用 Node.js 开发命令行工具](http://www.jikexueyuan.com/course/1332.html) >命令行工具  <http://www.jikexueyuan.com/course/1332_1.html>  读取node.js命令行输入内容的小工具-simple  <https://www.baidufe.com/item/6e1ffe79fa84ffb8e03b.html>  从零开始打造个人专属命令行工具集——yargs完全指南  <http://www.jianshu.com/p/fef668d61085> |
|  |  |
| 源文件结构 | 开头一行指定脚本的解释程序; |
| 基础步骤 | ming目录执行 npm init 得到如下json： |
| {  **"name"**: **"ming"**,  **"version"**: **"1.0.0"**,  **"description"**: **""**,  **"main"**: **"ming.js"**,  **"scripts"**: {  **"test"**: **"echo \"Error: no test specified\" && exit 1"** },  */\*bin中的值不能是index.js:如果写为index.js会导致> npm link > ming 执行的结果是直接打开index.js文件\*/*  **"bin"**: {  **"ming"**: **"ming.js"** },  **"author"**: **""**,  **"license"**: **"ISC"** } |
| 对bin做修改：bin字段中，ming是命令行的名字，ming.js是要执行的文件，然后需要使用npm来使命令ming可以在整个系统下执行。在当前目录下执行：npm link |
| 引用及更新 | 在一个新的模块中，通过npm link获取ming的引用，将全局的ming变为局部：  F:\HTML\practice\testnode\命令行\ming-test>**npm link ming**  F:\HTML\practice\testnode\命令行\ming-test\node\_modules\ming -> C:\Users\Administrator\AppData\Roaming\npm\node\_modules\ming -> F:\HTML\practice\testnode\命令行\ming    当ming文件中内容被修改后，在ming-test中重复执行npm link ming即可将ming中的修改同步至ming-test中； |
|  | package.json 文件中 bin 里面的内容表示这个字段将 wcj 命令映射到了你的 bin/wcj.js 脚本。[bin参考](https://docs.npmjs.com/files/package.json" \l "bin" \t "https://segmentfault.com/a/_blank)  此工具采用 npm版本号采用的 [semver](http://semver.org/lang/zh-CN/" \t "https://segmentfault.com/a/_blank) 规则  "bin": { "wcj": "bin/wcj.js" }  [列子源码](https://github.com/jaywcjlove/wcj/blob/master/examples/ls.js" \t "https://segmentfault.com/a/_blank)  全局运行命令调试  确保你在 package.json 文件中添加了 bin 节点。然后打开命令了工具进入 wcj 目录  install  如果在项目目录下运行没有问题，可以将当前目录模块安装到全局，也可以采用此方法来更新你的命令行工具  sudo npm install . -g  link  或者目录输入 npm link 会自动添加全局的 symbolic link ，然后就可以使用自己的命令了。  $ wcj#README.md#bin#package.json  $ cmd -v# version is 1.0.0  $ cmd -h #Useage:# -v --version [show version]  错误处理  在运行 sudo npm install . -g 会有一堆警告可以忽视  如果你已经 npm link 搞了一遍你再 link 一遍，会报如下错误。即使你 npm unlink 也会报如下错误：  npm linknpm ERR! Darwin 14.3.0npm ERR! argv "node" "/usr/local/bin/npm" "link"npm ERR! node v0.10.36npm ERR! npm v2.7.1npm ERR! path /usr/local/bin/wcjnpm ERR! code EEXIST  npm ERR! Refusing to delete: /usr/local/bin/wcj not in /Applications/XAMPP/xamppfiles/htdocs/git/github.com/myJS/wcj  File exists: /usr/local/bin/wcj  Move it away, and try again.  让你删除 /usr/local/bin/wcj 再 npm link ， 删除此目录运行 rm -rf /usr/local/bin/wcj |
|  |  |

[nodejs colors 模块设置控制台文本颜色](http://blog.csdn.net/dai_jing/article/details/47295067)

|  |  |
| --- | --- |
|  | <http://blog.csdn.net/somnus3421/article/details/51200145> |
| cnpm install colors  node colorModuleTest.js |

|  |  |
| --- | --- |
|  | Node.js 中的循环依赖  <http://div.io/topic/1512> |
|  | 我们在写node的时候有可能会遇到循环依赖的情况，什么是循环依赖，怎么避免或解决循环依赖问题？  先看一段官网给出的循环依赖的代码:  a.js:  console.log('a starting');  exports.done = false;var b = require('./b.js'); // ---> 1  console.log('in a, b.done = %j', b.done);  exports.done = true;  console.log('a done') // ---> 4  b.js:  console.log('b starting');  exports.done = false;var a = require('./a.js'); // ---> 2// console.log(a); ---> {done:false}  console.log('in b, a.done = %j', a.done); // ---> 3  exports.done = true;  console.log('b done');  main.js:  console.log('main starting'); var a = require('./a.js'); // --> 0var b = require('./b.js');  console.log('in main, a.done=%j, b.done=%j', a.done, b.done);  如果我们启动 main.js 会出现什么情况？ 在 a.js 中加载 b.js，然后在b.js中加载 a.js，然后再在 a.js中加载 b.js 吗？这样就会造成循环依赖死循环。  让我们执行看看：  $ node main.js  main starting  a starting  b startingin b, a.done = false  b donein a, b.done = true  a donein main, a.done=true, b.done=true  可以看到程序并没有陷入死循环，从上面的执行结果可以看到 main.js 中先require了 a.js ，a.js 中执行完了console和export.done=fasle之后，转而去加载b.js，待b.js被load完之后，再返回a.js中执行完剩下的代码。  我在官网的代码基础上增加了一些注释，基本 load 顺序就是按照这个0-->1-->2-->3-->4的顺序去执行的，然后在第二步下面我打印出了require('./a')的结果，可以看到是{done:false}，可以猜测在b.js中require('./a')的结果是a.js中已经执行到的exports出的值。  上面所说的还只是基于结果基础上的猜测，没有什么说服力，为了验证我的猜测是正确的，我把 Node 的源码稍微翻看了一些，C++ 的代码看不懂没关系，能看懂 JS 的部分就可以了，下面就是 Node 源码的分析（主要是 module 的分析，[Node 源码在此](https://github.com/nodejs/node" \t "http://div.io/topic/_blank)）：  将会分析的主要源码：  node/src/node.js  node/lib/module.js |
|  |  |
|  |  |
|  |  |

其他

|  |  |
| --- | --- |
|  | 孢子社区前端 js 部署是本地缓存加载的，所有的 js 和 css 被打包成几个独立的文件，缓存在 localStorage 内。  每次访问主页时，会请求部署包的 hash，发现与本地缓存不匹配的，单独更新变化的代码包。  更新完毕后，再由引导程序按顺序加载 css 和各个 js 包，进入框架初始化部分。  因此，除非用户遇到升级前端代码，刷新浏览器只会看到一次获取代码包 hash 的请求。  楼主的疑问是对的，对技术很敏锐。没有愚弄大家。  其他还有一些技术，比如不同浏览器缓存机制不同(是否支持 localStorage)，不同浏览器对应的 css 包不同(icon 支持格式，是否支持 dada:，dada: 最大尺寸限制等)。就不一一展开了。 |

node-inspector

|  |  |
| --- | --- |
|  | <https://cnodejs.org/topic/4f77c41f8a04d82a3d5dfcb9>  <http://jingyan.baidu.com/article/dca1fa6fbd580ff1a44052de.html> |
|  |  |

JavaScript的运行机制Event Loop（事件循环）

|  |  |
| --- | --- |
|  | <http://blog.csdn.net/qq_35534823/article/details/69388045#t1>  我们都知道JavaScript是一个单线程的语言。单线程也就意味着同一时间只能做一件事。  JavaScript之所以设计为单线程，是因为它的用途主要还是用来操作DOM，为了避免复杂性，所以JavaScript在诞生之初就是单线程的，这也是这么语言的核心特征，未来也不会改变。  虽然HTML5为了尽可能地利用CPU的计算能力而推出了Web Worker标准，允许JavaScript脚本创建多个线程，但是需要注意的是子线程必须受主线程的控制，并且还不能操作DOM。所以，这个标准并没有改变JavaScript的单线程本质。  任务队列  单线程就意味着所有任务都需要排队，前一个任务结束，才会执行后一个任务。如果前一个任务耗时很长，后一个任务就不得不一直等着。  但是这种效率很低，所以JavaScript的任务分成两种，一种是同步任务（synchronous），另一种是异步任务（asynchronous）。同步任务指的是，在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务；异步任务指的是，不进入主线程、而进入’任务队列’(task queue)的任务，只有’任务队列’通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。  执行过程如下：  1 所有同步任务都在主线程执行，形成一个执行栈.  2 所有异步任务都在任务队列中(task queue) 。只要异步任务有了运行结果，就在‘任务队列’之中放置一个事件.（这个事件是一个flag）  3 等线程执行完执行栈中的同步任务之后，系统会读取‘任务队列’，看看里面有哪些事件，然后把等待执行的任务（有flag的任务）推入到执行栈中，开始执行。  主线程不断重复第三步。  IMG_256  只要主线程空了，就会读取‘任务队列’。这就是js的运行机制。  事件和回调函数  任务队列其实是一个事件队列，因为里面的异步任务在有了运行结果（IO设备，用户产生的事件包括click等，ajax，定时器）之后，就会在‘任务队列’中添加一个事件，表示相关的异步任务可以进入‘执行栈’了。主线程读取异步队列，其实就是读取里面的事件。  异步任务必须指定回调函数，ES7的async，await 其实也是指定了回调函数的，只不过是用同步写法写异步。  ‘任务队列’是一个先进先出的数据结构，排在前面的事件，优先被主线程读取。  再次强调的是，是排在前面的事件，而不是排在前面的异步任务，因为主线程读取的是事件。  看个例子：  console.log(1)  setTimeout(function(){console.log(2)}, 5)  setTimeout(function(){console.log(3)}, 4);  console.log(4);  如果读取的是异步任务的话，那么应该是1，4，2，3  而结果是1，4，3，2  因为异步队列中的两个任务，第二个任务先有运行结果，先获得flag事件。所以排在task queue的前面，先被读取到。  另外需要注意的是，对于setTimeout这类定时器只是将事件插入到了‘任务队列’里，必须要等到执行栈中的代码执行完之后，主线程才会执行它指定的回调函数。所以说如果执行栈中的代码耗时很长，就有可能等很久，所以没办法保证回调函数一定会在setTimeout()指定的时间执行。  Event Loop  主线程中‘任务队列’中读取事件，这个过程是循环不断的。所以这种运行机制叫做Event Loop（时间循环）。  IMG_257 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |