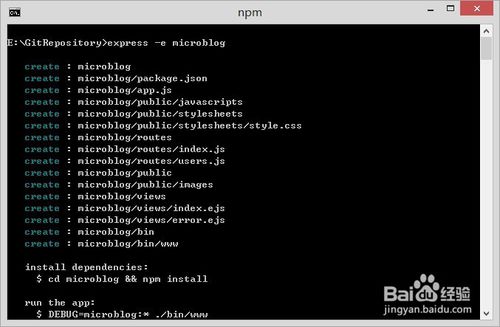
cmd命令行中输入以下命令安装express，目前安装的是express4，项目目录结构和以前会有些差别。

npm install -g express

2

输入命令创建一个名为"microblog"的空项目，-e参数代表使用的ejs模板，如果想用jade的话请使用-t

express -e microblog

[](http://jingyan.baidu.com/album/92255446aafc06851748f451.html?picindex=1)

3

cd到microblog目录下，执行命令，安装项目依赖。

npm install

[](http://jingyan.baidu.com/album/92255446aafc06851748f451.html?picindex=2)

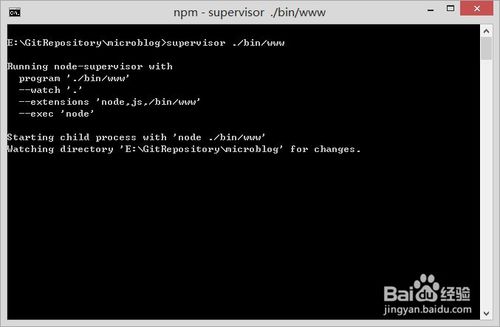
4

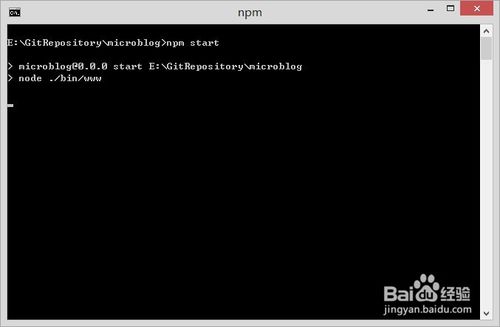
下面来启动这个项目，如果安装了supervisor的话可以执行以下命令，supervisor的作用就是热部署项目，每次修改之后可以立即生效。

supervisor ./bin/www

没有安装supervisor的话也没有关系，直接用npm自带的方式启动即可

npm start

[](http://jingyan.baidu.com/album/92255446aafc06851748f451.html?picindex=3)

[](http://jingyan.baidu.com/album/92255446aafc06851748f451.html?picindex=4)

5

打开浏览器，输入localhost:3000

看到以下页面代表部署成功！

[](http://jingyan.baidu.com/album/92255446aafc06851748f451.html?picindex=5)

# **通过 nodeclub 项目源码来讲解如何做一个 nodejs + express + mongodb 项目**

## **##1. About**

* 1.1 ****what****: nodeclub 是 [cnodejs.org](http://cnodejs.org/" \t "http://cnodejs.org/topic/_blank) 的源码，CNode 算是一个基本的博客系统，包含文章发布， 关注，评论等功能。这些功能可以说是任何一个网站的基础。从 nodeclub 里可以学到什么？
  + 1.基本的架构
  + 2.开发测试过程
  + 3.MVC 的设计
  + 4.middleware 的正确用法
  + 5.如何设计 Mongodb schema
  + 6.如何正确的使用 Mongoose
  + 7.如何实现一个标签系统
  + 8.plugins? services ?
  + 9.如何正确的使用 EJS helper
  + 10.到底该怎样写路由， restful？
  + 11.如何做基本的控制验证
  + 12.如何发邮件
  + 13.session
  + 14.GitHub 用户登录
  + 15.图片上传
  + 16.消息发送

除了 nodeclub 源码的学习笔记以外， 还会有一点最近捣鼓这一块的经验分享

* 1.一个完整的消息订阅设计
* 2.消息推送, socket + express 如何合作?
* 3.包装 action
* 4.蛋疼的异步回调如何处理

[nodeclub源码](https://github.com/cnodejs/nodeclub" \t "http://cnodejs.org/topic/_blank)

* 1.2 ****why****:

对于想用 nodejs + express + mongodb 来做网站技术基础的项目， nodeclub 可以说是很好的源码级指南，当然也是我的指南，这篇文章权当做个人学习 nodeclub 的学习笔记。

* 1.3 ****who****

who = 一名本应该在写前端的但不知怎的一直在写后端的马脓 ->

[@echo](/user/echo) 'github: https://github.com/6174'

[@echo](/user/echo) 'weibo: http://weibo.com/u/2254313183'

[@echo](/user/echo) 'email: 57017125[@qq](/user/qq).com'

[@echo](/user/echo) 'ps: 一直在求后端partner中，有意者联系我'

[@send](/user/send)()

## **2. nodeclub 中用到了哪些开源技术**

2.1 Node.js 项目一大优点就是有一个 package.json, 里边的 dependencies & devDependencies 可以看到这个项目所有的依赖。 对于有经验的开发者来说， 看完 package.json 基本就能知道项目的架构是怎样。

2.2 ****dependencies****

* express: 基础框架：
* mongodb: 数据存储
* mongoose: orm
* connect-mongo: session （对于redis， 可以使用connect-redis）
* nodemailer：邮件
* validator：验证
* passport，passport-github： passport，
* loader: ejs-view-helper, 静态资源加载处理
* 其他： event-proxy, node-markdown, ndir
* 2.3 ****devDependencies****
* 测试框架：mocha, should
* 运行： forever
* 请求模拟: supertest
* 2.4 nodeclub 以 express + mongodb + mongoose 作为基本框架, 典型的 MVC 应用
* ****Model****: 对应mongoose orm， models目录
* ****view****： ejs模板， views目录
* ****controler****：express middleware , contollers目录
* 2.5 目录结构：

- common/- controllers/- libs/# express中间件， 基本的auth， session 验证- middlewares/- models/#消息， 邮件服务- services/- plugins/#可以看做是对model处理的加工库- proxy/- test/- views/- app.js- route.js- config.js

## **3. 应用入口 app.js**

神圣的入口文件，几乎每个项目都会有一个 entry，对于了解一个应用熟悉入口逻辑很重要。 下面将分步来看看，nodeclub 的 app.js 做了什么：

### **3.1 require(./config)**

* 3.1.1 应用相关的配置的设置， 主要分为
* 1.应用全局数据配置
* 2.数据库连接配置
* 3.session，auth 相关配置
* 4.rss配置
* 5.mail配置
* 6.第三方连接相关配置， github， weibo

配置文件也是了解应用的一个好地方， 在 config.default.js 中可以看到以下信息, 这些很可能是我们平时做应用开发的时候没有留意到的地方

//--应用数据统计

google\_tracker\_id: 'UA-41753901-5',

//--静态文件很可能使用cdn来做

site\_static\_host: '', // 静态文件存储域名

//--求解释

site\_enable\_search\_preview: false, // 开启google search preview

site\_google\_search\_domain: 'cnodejs.org', // google search preview中要搜索的域名

//--运营数据

list\_topic\_count: 20,

post\_interval: 10000,

admins: { admin: true },

side\_ads:[]

allow\_sign\_up: true,

//--插件模式

plugins: []

* 3.1.2 当然这里的配置文件是 default 的，配置文件可以放在一个 config 的文件夹下面，多个文件的方式来整理。比如运营数据配置和其他数据配置分开，因为很有可能需要做一个小的工具来让非技术人员配置相关参数。这时候可以用一个 index.js 作为 facade，相当于一个大的 node module。

### **3.2 require('./models')**

* 3.2.1 之前已经讲了 models/ 目录对应 MVC 的 M 部分。
* 3.2.2 models/ 目录下面有 index.js, require('./models') 相当于 require('./models/index')

index 相当于一个模型的 facade, index.js 做得事情分别是

* 1.connect mongodb
* 2.require 各个 model 模块
* 3.exports 所有的 model

简单而言就是初始化了应用 model 层。

* 3.2.3 模型使用 orm 框架 mogoose 来写，了解 mogoose 过后， models 部分的代码也就是秒懂了

， 我说的只是代码，literaly, 一个项目的核心就是 model 的设计，以前做过的任何项目都是一样， 数据库 table 的设计好坏直接影响应用的开发以及性能。 下面来看看各个 model 的 schema 设计(几乎直接 ctr+c, ctr+v 加上了一点点注释) :

* 3.2.4 user

var UserSchema = new Schema({

//--基本用户信息， index表示在mongodb中会建立索引

//--unique: true 唯一性设置

name: { type: String, index: true },

loginname: { type: String, unique: true },

pass: { type: String },

email: { type: String, unique: true },

url: { type: String },

profile\_image\_url: {type: String},

location: { type: String },

signature: { type: String },

profile: { type: String },

weibo: { type: String },

avatar: { type: String },

githubId: { type: String, index: true },

githubUsername: {type: String},

is\_block: {type: Boolean, default: false},

//--用户产生数据meta

score: { type: Number, default: 0 },

topic\_count: { type: Number, default: 0 },

reply\_count: { type: Number, default: 0 },

follower\_count: { type: Number, default: 0 },

following\_count: { type: Number, default: 0 },

collect\_tag\_count: { type: Number, default: 0 },

collect\_topic\_count: { type: Number, default: 0 },

create\_at: { type: Date, default: Date.now },

update\_at: { type: Date, default: Date.now },

is\_star: { type: Boolean },

level: { type: String },

active: { type: Boolean, default: true },

//-mail

receive\_reply\_mail: {type: Boolean, default: false },

receive\_at\_mail: { type: Boolean, default: false },

from\_wp: { type: Boolean },

retrieve\_time : {type: Number},

retrieve\_key : {type: String}});

* 3.2.5 topic 话题

//1 <- 多//tag <- topic <- collect var TopicSchema = new Schema({

title: { type: String },

content: { type: String },

author\_id: { type: ObjectId },

top: { type: Boolean, default: false },

reply\_count: { type: Number, default: 0 },

visit\_count: { type: Number, default: 0 },

collect\_count: { type: Number, default: 0 },

create\_at: { type: Date, default: Date.now },

update\_at: { type: Date, default: Date.now },

//--这里reply的设计方式不知道是否合适， 因为mongdb不同于关系型数据库，这里每次读取文章都需要重reply集合里边查找遍历一边，文章是读繁忙的。

//-- 一个document的大小为5Mb， 一本牛津词典的内容， 我觉得将reply放在这里应该不会有太大问题。 即便不存放reply 内容， 存放一个id数组也会好很多。

//-- 客官们怎么看?

last\_reply: { type: ObjectId },

last\_reply\_at: { type: Date, default: Date.now },

content\_is\_html: { type: Boolean }});

var ReplySchema = new Schema({

content: { type: String },

topic\_id: { type: ObjectId, index: true },

author\_id: { type: ObjectId },

reply\_id : { type: ObjectId },

create\_at: { type: Date, default: Date.now },

update\_at: { type: Date, default: Date.now },

content\_is\_html: { type: Boolean }});

//--话题集合var TopicCollectSchema = new Schema({

user\_id: { type: ObjectId },

topic\_id: { type: ObjectId },

create\_at: { type: Date, default: Date.now }});

//--话题标签var TopicTagSchema = new Schema({

topic\_id: { type: ObjectId },

tag\_id: { type: ObjectId },

create\_at: { type: Date, default: Date.now }});

* 3.2.6 tag 标签系统

//tag <- collectvar TagSchema = new Schema({

name: { type: String },

order: { type: Number, default: 1 },

description: { type: String },

background: { type: String },

topic\_count: { type: Number, default: 0 },

collect\_count: { type: Number, default: 0 },

create\_at: { type: Date, default: Date.now }});

var TagCollectSchema = new Schema({

user\_id: { type: ObjectId, index: true },

tag\_id: { type: ObjectId },

create\_at: { type: Date, default: Date.now }});

* 3.2.7 关系

var RelationSchema = new Schema({

user\_id: { type: ObjectId },

follow\_id: { type: ObjectId },

create\_at: { type: Date, default: Date.now }});

* 3.2.8 消息 消息 model 设计， 对于一个 blog 来说， 基本的只有回复消息， 这里加了关注和@消息。

/\*

\* type:

\* reply: xx 回复了你的话题

\* reply2: xx 在话题中回复了你

\* follow: xx 关注了你

\* at: xx ＠了你

\*/var MessageSchema = new Schema({

type: { type: String },

master\_id: { type: ObjectId, index: true },

author\_id: { type: ObjectId },

topic\_id: { type: ObjectId },

reply\_id: { type: ObjectId },

has\_read: { type: Boolean, default: false },

create\_at: { type: Date, default: Date.now }});

###3.3 require middlewares

3.3.1 express 的基础是 middleware，或者说 express 的基础是 connect，connect 的基础是 middleware。middleware 模式在 professional nodejs 中有一个专门的章节来讲解。何为 middleware 呢？ middleware 模式 相当于一个加工流水线（大家叫 middleware stack），每一个 middleware 相当于一个加工步骤，当出现一个 http 请求的时候，http 请求会挨着每个 middleware 执行下去。 express 里处理一个请求的过程基本上就是请求通过 middleware stack 的过程： \* -> middlewares -> 路由 -> controllers -> errorhandlering。

3.3.2 middleware 怎样做到的， 异步的方法呢？ middleware 使用 promise 的方式来处理异步，所有每个 middleware 都有三个参数 req, res, next, 对于异步的情况， 必须要调用 next() 方法。不然后续的 middleware 就无法执行。 ps: debug 的时候没调用 next() 还不会报错，一定注意

3.3.3 auth.js

auth.js exports 出来的函数全部都是中间件，从变量名就完全清楚的知道到底在做什么了

//-- 需要admin权限

exports.adminRequired = function (req, res, next) {}

//-- 需要有用户

exports.userRequired = function (req, res, next) {}

//-- 需要有用户并登录

exports.signinRequired = function (req, res, next) {

if (!req.session.user) {

res.render('notify/notify', {error: '未登入用户不能发布话题。'});

return;

}

next(); }

//-- 屏蔽用户 -\_-

exports.blockUser = function (req, res, next) {}

这里其实就可以看到中间件的作用了，我们以前写 php 的时候每次都需要判断用户是否登录， 没登陆 redirect 到 index.php ，只不过这里的方式是通过中间件来处理。 明白这里什么意思，其他的中间件模块也就秒懂了。

###3.4 require(’./routes’)

3.4.1 express 的世界里另外一个很重要的就是route， Node.js 启动的是服务， 监听了某一端口， 接受 http or https or socket 请求, 那 url 中像 /index.php?blabla 这一串的存在怎么处理呢， express 的 route 功能就可以帮我们解析。

3.4.2 MVC 中如何将一个请求和 controller 联系起来呢， route 就是这样的纽带

//--get, post 请求

app.get('/signin', sign.showLogin);

app.post('/signin', sign.login);//--使用中间件

app.get('/signup', configMiddleware.github, passport.authenticate('github'));

app.post('/:topic\_id/reply', auth.userRequired, limit.postInterval, reply.add);

* 3.4.3 route 是了解一个应用最佳的地方，一个请求如何处理， 到相应的 controller 去看就知道了。 相比起在PHP环境下配置更加灵活。当然你说你通过nginx来配置也很灵活，好吧，我们说的不是一回事。

### **3.5 initialization**

* 3.5.1 experess initialize: app.js 中其他大多部分就是express的初始化了， 初始化流程如下：
* 1.配置上传 upload\_dir
* 2.模板引擎设置
* 3.express 通用中间件设置
* 4.pasport 中间件
* 5.自定义中间件
  + 1.auth\_user
  + 2.block\_user
  + 3.staticfile: upload
  + 4.staticfile: user\_data
* 6.csrf
* 7.errorhandler
* 8.set view cache

[@Note](/user/Note)：配置的顺序很重要， 中间件的执行顺序是按照定义顺序来执行的， 如果一个中间件依赖另外的中间件， 而自己先执行了， 这种情况就会错误。 常见的问题就是session配置， 一定要记得配置 session 中间件的时候， 要先配置 cookieParser。

* 3.5.2 session 设置

这个步骤在 initialize 里边已经有了， 不过再单独讲一下， nodeclub 使用的是 connect-mongo 来作为 session 的存储

//--cookieParser一定要在前面， 因为session的设置依赖cookie

app.use(express.cookieParser());

app.use(express.session({

secret: config.session\_secret,

store: new MongoStore({

db: config.db\_name,

}),

}));

* 3.5.3 view helpers

使用过 ejs 的肯定知道， ejs 里边 view helper 设置很简单， 就像赋值变量一样。 当对于一些通用的 helper 可以这样设置：

app.helpers({

config: config,

Loader: Loader,

assets: assets

});

app.dynamicHelpers(require('./common/render\_helpers'));

* 3.5.4 github pasport initialize

// github oauth

passport.serializeUser(function (user, done) {

done(null, user);

});

passport.deserializeUser(function (user, done) {

done(null, user);

});

passport.use(new GitHubStrategy(config.GITHUB\_OAUTH, githubStrategyMiddleware));

* 3.5.5 start app

## **##4. 用户注册**

* 4.1 user 是每个应用都会处理的基本， 注册登录登出， 看看 nodeclub 做了哪些事情：
* 4.2 路由：

//--设置能否直接注册， 不能的话通过github注册if (config.allow\_sign\_up) {

app.get('/signup', sign.showSignup);

app.post('/signup', sign.signup);} else {

app.get('/signup', configMiddleware.github, passport.authenticate('github'));}

app.post('/signout', sign.signout);

app.get('/signin', sign.showLogin);

app.post('/signin', sign.login);

* 4.3 controller & model：sign.signup

sanitize = validator.sanitize;

check = validator.check;

exports.signup = function (req, res, next) {

//--xss 消毒

var name = sanitize(req.body.name).trim();

name = sanitize(name).xss();

...

//--validations

try {

check(name, '用户名只能使用0-9，a-z，A-Z。').isAlphanumeric();

} catch (e) {

res.render('sign/signup', {error: e.message, name: name, email: email});

return;

}

...

//--用用户名登录或者email登录

query = {'$or': [{'loginname': loginname}, {'email': email}]}

User.getUserByQuery(query, {}, function(){

...

pass = md5(pass);

...

User.newAndSave(name, loginname, pass, email, avatar\_url, false, function (err) {

...

// 发送激活邮件

mail.sendActiveMail(email, md5(email + config.session\_secret), name);

res.render('sign/signup', {

success: '欢迎加入 ' + config.name + '！我们已给您的注册邮箱发送了一封邮件，请点击里面的链接来激活您的帐号。'

});

})

})}

## **##5. mongoose 的使用**

* 5.1 使用User.newAndSave，
* 5.2 异步 callback pyramid

一个应用通常会遇到这样的情景， 一个页面需要的数据包括， 文章列表， 评论列表，用户数据，广告数据， other stuff… 问题是每个都是异步的， 怎么办。 user 数据获取过后的 callback 调用文章列表获取， 文章列表获取的 callback 调用评论列表的获取… 这样就太蛋疼了。 nodeclub 使用了 eventproxy 模块优雅的解决这样的问题：

render = function(){}

var proxy = EventProxy.create('tags', 'topics', 'hot\_topics', 'stars', 'tops', 'no\_reply\_topics', 'pages', render);

proxy.fail(next);

Tag.getAllTags(proxy.done('tags'));

Topic.getTopicsByQuery(query, options, proxy.done('topics'));

User.getUsersByQuery({ is\_star: true }, { limit: 5 }, proxy.done('stars'));

看完代码不言而喻。。。 当然异步处理的方法有很多:

* 1.基于事件的：eventProxy
* 2.基于promise的：Async.js Q.js, when.js
* 3.基于编译的：continuation, wind
* 4.基于语言语法的：yield， livescript

文章最后会讲一下我我的异步选择方案

## **##6. 消息**

* 6.1 原先以为有动态的消息推送， 有队列处理， 错了， 木有
* 6.2 在 Sublime text 里边全局搜索 sendReply2Message 会发现是在 controller/reply.js 里边调用的， 也就是说，消息是直接触发的。
* 6.3 好吧， 这部分大概大家都能秒懂。。

## **##7. 开发**

###7.1 测试

* 7.1.1 一个项目必定离不开测试， nodeclub基于mocha BDD测试框架， 一切的前提假设至少能看懂jasmine或者mocha或者任何一个BDD风格的测试代码。
* 7.1.2 打开即看到app.js

var app = require('../app');

describe('app.js', function () {

//--before， 执行it的前面会执行

before(function (done) {

//--done, 异步方法

app.listen(3001, done);

});

after(function () {

app.close();

});

it('should / status 200', function (done) {

//--使用 app.request()就可以模拟请求了？ 这个api哪里来的， 求解释？

app.request().get('/').end(function (res) {

res.should.status(200);

done();

});

});});//--按理说应该是可以正常运行了但是我一直出现这个错误://--connect ADDRNOTAVAIL 知道的求解释//--我尝试用supertest直接测试， 但是也是一直timeout， mocha//--里边加大timeout时间， 结果就是一直没反应。

//--分析原因， express版本问题， nodeclub中express的版本还是2.x, 所以才会有//--app.request(), app.close()这些api//--第二个原因， 到supertest官网， 发现人家都已经转战到superagent项目了， 于是我写了下面这个测试脚本， 可以通过了var express = require('express');var should = require('should');var path = require('path');var superagent = require('superagent');var app = express()

app.get('/user', function(req, res, next) {

res.send(200, {

name: 'tobi'

})})

describe('myapp.js', function() {

this.timeout(5000)

before(function(done) {

app.listen(21, done);

})

after(function() {

// app.close()

})

it('should /status 200', function(done) {

agent = superagent.agent()

agent.get('http://localhost:21/user').end(function(err, res) {

console.log(err, res)

res.should.have.status(200);

res.text.should.include('tobi');

return done();

});

})})

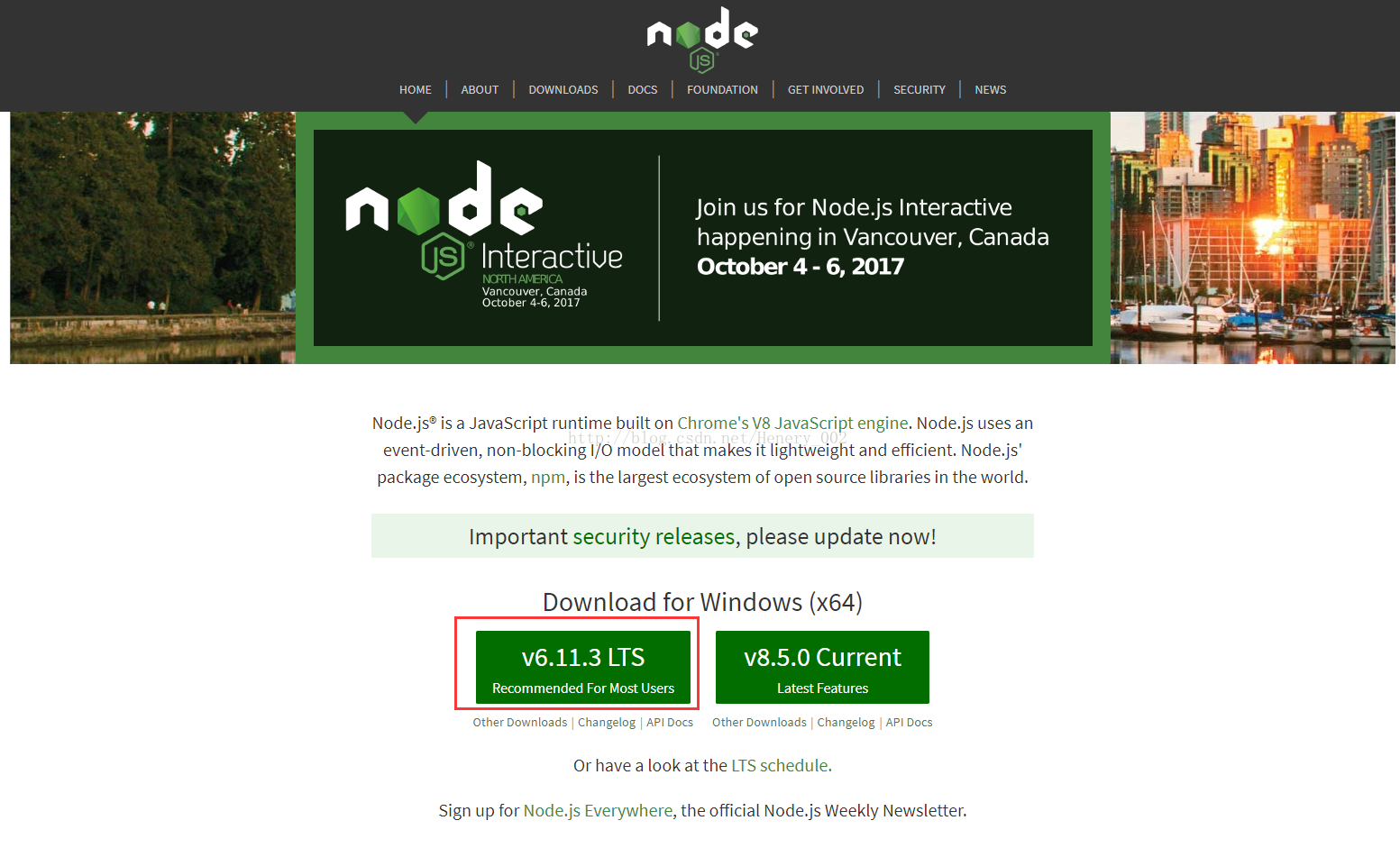
###7.2 运行

* nodejs是单线程应用， 如果我们用node命令来运行我们的应用， 当出现一个小错误， 它就挂了。 然后没有然后了。 避免这种问题的方法有如下工具：
* 1.forever
* 2.nodemon
* 3.supervisor nodeclub 使用 forever 来运行项目， 使用这类工具的好处就是， 当有代码改动过后， 会自动的重启应用。 不必每次自己去运行 node \*.js

## **##8. 说说自己的经验**

待续… ###8.1 消息订阅设计 ###8.2 express + socket ###8.3 异步 ###8.4 Action

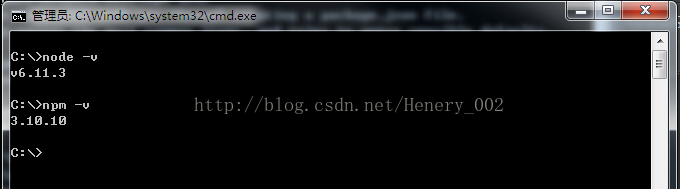
****1.下载Node.js****

[https://nodejs.org/en/](https://nodejs.org/en/" \t "https://blog.csdn.net/henery_002/article/details/_blank)，左侧为稳定版，右侧为最新版，推荐稳定版。

****2.安装Node.js****

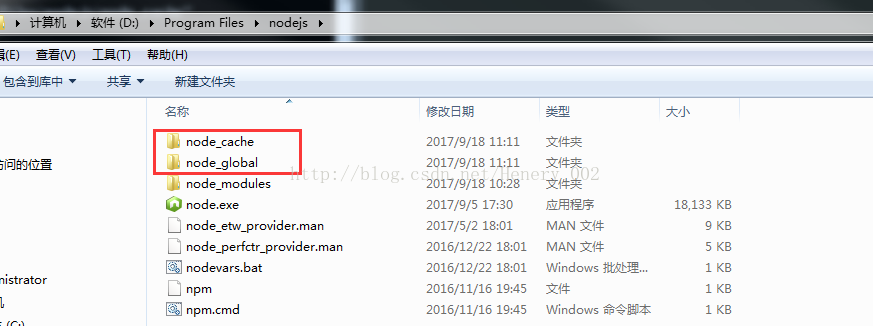
一路下一步，记住安装路径（此处安装在 D:\Program Files\nodejs 下）。

安装完成后（需要配置环境变量：新建NODE\_PATH，取值为node的安装路径，如：E:\Program Files\nodejs\node\_modules）运行cmd，执行node -v和npm -v命令查看安装情况。如执行结果如下图，说明安装成功。



（执行完该命令后，nodejs安装目录下会自动生成文件夹node\_global，若未生成则手动创建）

****3.配置npm的全局模块的存放路径以及cache的路径****

可使用Node.js的安装路径，在此路径下建两个文件夹node\_global和node\_cache：

然后执行如下命令：

 npm config set prefix "D:\Program Files\nodejs\node\_global" 回车

 npm config set cache "D:\Program Files\nodejs\node\_cache" 回车



然后可以输入指令查看：

 npm config get prefix 回车

 npm config get cache 回车

****4.配置环境变量【\*\*\*\*】****

4.1 控制面板-系统和安全-系统-高级系统设置-高级-环境变量，

新建【系统变量】

变量名：NODE\_PATH

变量值：D:\Program Files\nodejs\node\_global\node\_modules

（即步骤2全局模块的存放路径，亲测实际应该是D:\Program Files\nodejs\node\_global\）

4.2 再将【用户变量】下的path值后追加：

D:\Program Files\nodejs\node\_global

（亲测实际在【系统变量】中的path中也追加了NODE\_PATH的相同路径，即D:\Program Files\nodejs\node\_global）

至此，便可使用 npm 安装相关模块，如安装最常用的 express 模块进行测试：

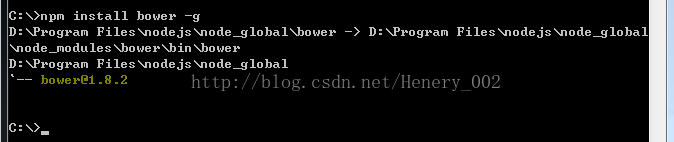
 npm install express -g 回车



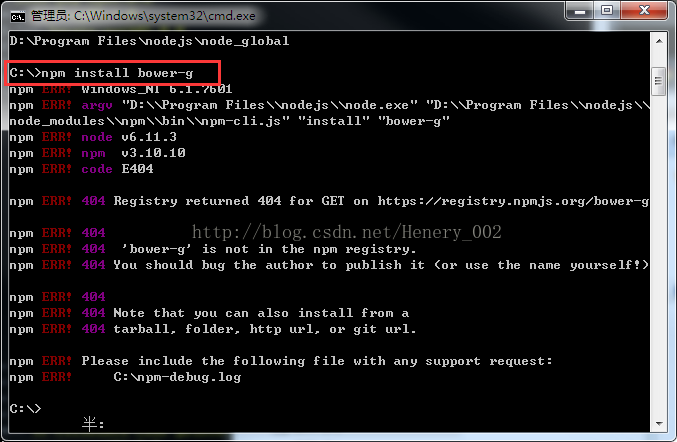


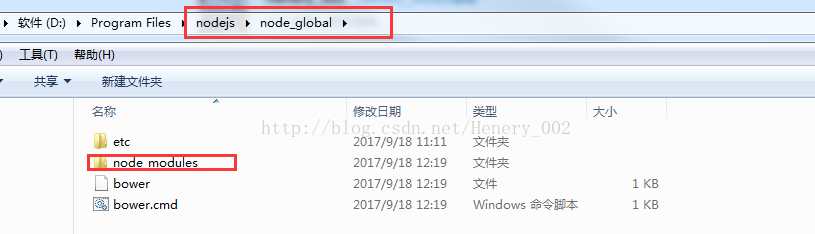
****5.尝试安装模块bower****

执行命令：

  C:\>npm install bower -g （-g指安装到node\_global文件夹中，-g前有一个空格）

若-g前没有空格，错误安装结果为：



安装成功后，将看到node\_global文件夹下新增文件夹 node\_modules>bower

然后执行 bower -v。（若报错bower不是内部或外部命令，则是因为环境变量未配置好）

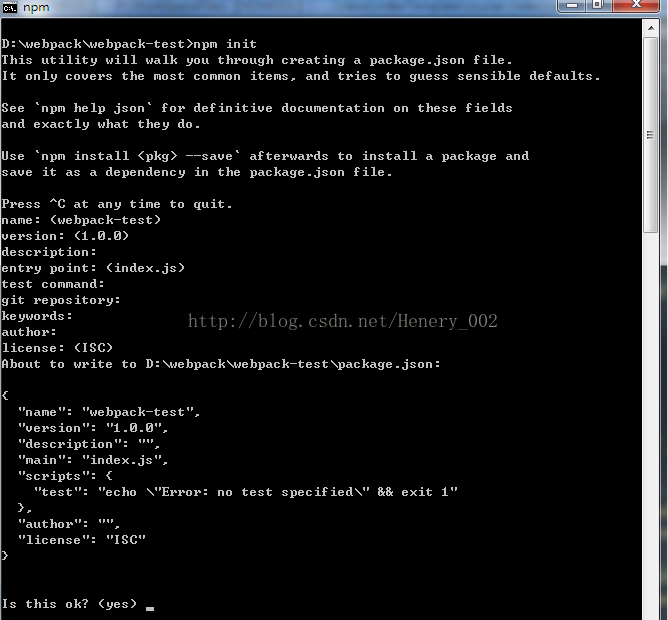
IMG_265

****6.初始化npm****

重新运行cmd（命令行创建子文件夹方法：mkdir webpack-test），初始化npm：

 npm init 回车。

一路回车（则将会以默认参数创建初始化json文件package.json）。





****7.安装webpack****

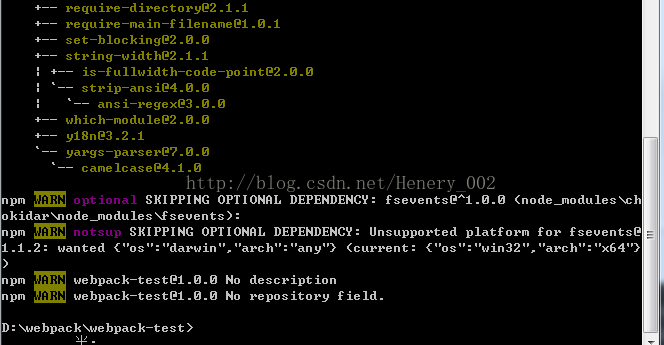
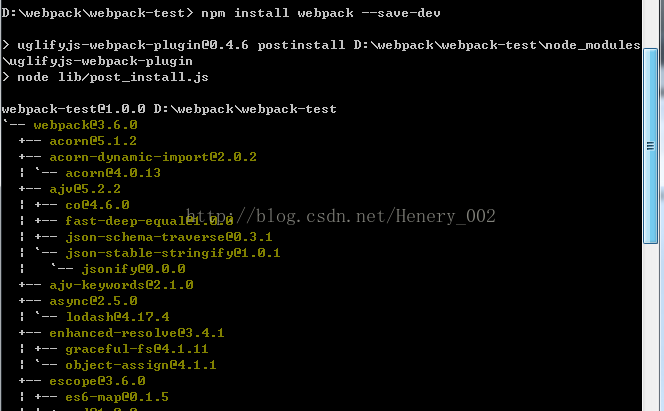
首先需要安装一个全局webpack：

 npm install webpack -g 回车

这样才可以正确使用webpack的命令；然后还可以在当前项目里也安装一个webpack，这是官方推荐的做法：

 npm install webpack --save-dev 回车

（此处我只安装了全局webpack，即只执行了第一步，并未影响）



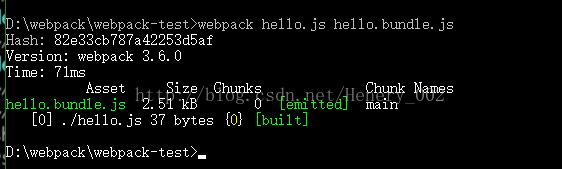
安装完成后在执行 webpack 测试时极易出错提示 webpack不是内部或外部文件。解决方法：建议在检查环境变量（NODE\_PATH、Path）是否有问题之后，删除已安装的webpack重新进行 -g 全局安装。

  
  
webpack测试成功后，可进行步骤8：

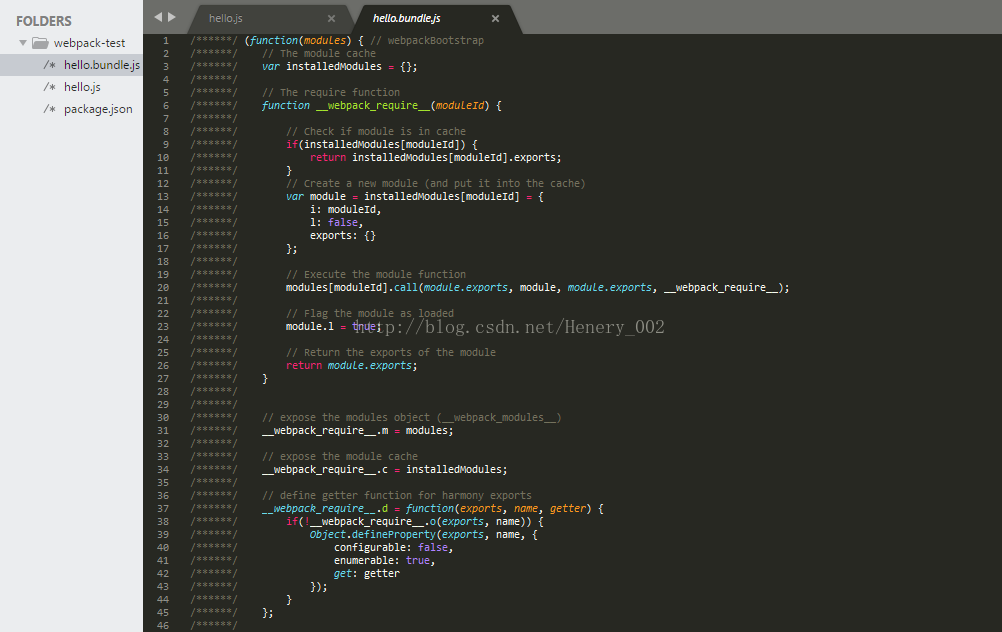
****8.使用webpack打包hello.js测试文件****

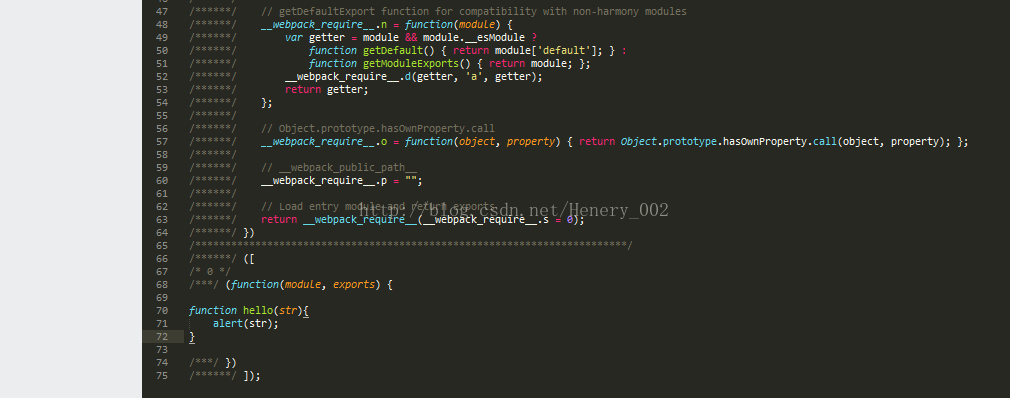
对示例文件hello.js进行打包：

  D:webpack\webpack-test>webpack hello.js hello.bundle.js 回车



成功后在webpack-test文件夹下会生成打包后的js文件hello.bundle.js：





至此，nodejs/npm的安装、webpack的安装、打包过程简易介绍到此，如有偏颇请指出，敬请期待后续更新。