

深入浅出Node.js (myblog项目);

使用 Express + MongoDB 搭建一个博客;

(Express 4.x 相关API可以参考: <http://www.expressjs.com.cn/4x/api.html>)

(MongoDB 相关API可以参考: <http://www.runoob.com/mongodb/mongodb-tutorial.html>)

1.开发环境以及初始化工作;

MAC OS

**Node.js: 8.9.1**

**MongoDB: 3.4.10**

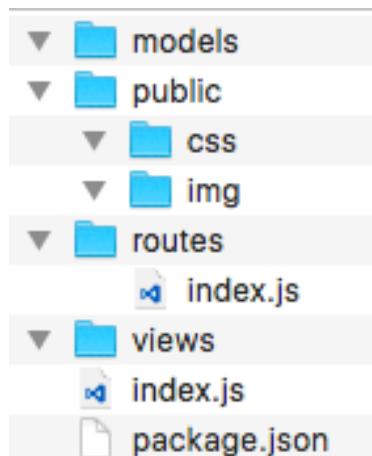
**Express: 4.16.2**

删除之前测试用的myblog目录, 重新初始化express项目, 仍旧使用myblog做为项目根目录;

<1>终端(terminal)上输入:

```
cd ~  
mkdir myblog  
cd myblog  
npm init
```

<2>在myblog目录下创建以下目录和文件(package.json 已经自动创建):



上面创建的文件及文件夹的用处:

- 1)models: 存放操作数据库的文件;
- 2)public: 存放静态文件, 如样式、图片等;
- 3)routes: 存放路由文件;
- 4)views: 存放视图模板文件;
- 5)index.js: 项目主文件(web应用入口);
- 6)package.json: 存储项目名、描述、作者、依赖等等信息;

需要注意的是, 这里的结构遵循了 MVC (模型(model) – 视图(view) – 控制器(controller/route)) 的开发模式;

<3>安装依赖模块;

在终端输入:

```
npm i config-lite connect-flash connect-mongo ejs express express-formidable  
express-session marked moment mongolass objectid-to-timestamp sha1 winston  
express-winston --save
```

上面所安装的所有模块用处:

- 1)express: web 框架;
- 2)express-session: session 中间件;
- 3)connect-mongo: 将 session 存储于 mongodb, 结合 express-session 使用;
- 4)connect-flash: 页面通知的中间件, 基于 session 实现;
- 5)ejs: 模板;
- 6)express-formidable: 接收表单及文件上传的中间件;
- 7)config-lite: 读取配置文件;
- 8)marked: markdown 解析;
- 9)moment: 时间格式化;
- 10)mongolass: mongodb 驱动;
- 11)objectid-to-timestamp: 根据 ObjectId 生成时间戳;
- 12)sha1: sha1 加密, 用于密码加密;
- 13>winston: 日志;
- 14)express-winston: express 的 winston 日志中间件;

之后会详细讲解这些模块的具体用法;

<4>ESLint;

ESLint 是一个代码规范和语法错误检查工具; 使用 ESLint 可以规范我们的代码书写, 可以在编写代码期间就能发现一些错误;

ESLint 需要结合编辑器或 IDE 使用:

- Sublime Text 需要装两个插件: SublimeLinter + SublimeLinter-contrib-eslint;
- VS Code 需要装一个插件: ESLint;

补充:

Sublime Text 安装插件通过 `ctrl+shift+p` 调出 Package Control, 输入 `install` 选择 `Install Package` 回车; 输入对应插件名搜索, 回车安装;  
VS Code 安装插件需要点击左侧'扩展'页;

在终端输入:

```
npm i eslint -g  
eslint --init
```

初始化 eslint 配置, 依次选择:

```
How would you like to configure ESLint?  
-> Use a popular style guide  
Which style guide do you want to follow?  
-> Standard  
What format do you want your config file to be in?  
-> JSON
```

注意:

1.上面的选择需要配合上下箭头按键, 如果 Windows 用户使用其他命令行工具无法上下切换选项, 切换回 cmd;  
2.eslint 会创建一个 `.eslintrc.json` 的配置文件(以.开头的文件在MAC中属于隐藏文件类型, 可以在终端中进入其所在路径, 使用`ls -a`, 然后`cat filename` 来查看此类文件的内容), 同时自动安装并添加相关的模块到 `devDependencies`; 这里选择的 Standard 规范的主要特点是不加分号;

#### <5> EditorConfig;

EditorConfig 是一个保持缩进风格的一致的工具, 当多人共同开发一个项目的时候, 往往会出现每个人用不同编辑器的情况, 而且有的人用 tab 缩进, 有的人用 2 个空格缩进, 有的人用 4 个空格缩进, EditorConfig 就是为了解决这类问题而诞生;

EditorConfig 需要结合编辑器或 IDE 使用, 如:

- Sublime Text 需要装一个插件: EditorConfig;
- VS Code 需要装一个插件: EditorConfig for VS Code;

在 myblog 目录下新建 .editorconfig 的文件, 添加如下内容:

```
# editorconfig.org
root = true

[*]
indent_style = space
indent_size = 2
end_of_line = lf
charset = utf-8
trim_trailing whitespace = true
insert_final_newline = true
tab_width = 2

[*.md]
trim_trailing whitespace = false

[Makefile]
indent_style = tab
```

上面配置的含义是: 使用 2 个空格缩进, tab 长度也是 2 个空格;  
trim\_trailing whitespace 用来删除每一行最后多余的空格, insert\_final\_newline 用来在代码最后插入一个空的换行;

在以上一系列依赖安装配置之后, myblog 项目下的 package.json:

```
{
  "name": "myblog",
  "version": "1.0.0",
  "description": "my blog project",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "config-lite": "^2.1.0",
    "connect-flash": "^0.1.1",
    "connect-mongo": "^2.0.0",
    "ejs": "^2.5.7",
```

```
"express": "^4.16.2",
"express-formidable": "^1.0.0",
"express-session": "^1.15.6",
"express-winston": "^2.4.0",
"marked": "^0.3.6",
"moment": "^2.19.2",
"mongolass": "^3.1.5",
"objectid-to-timestamp": "^1.3.0",
"sha1": "^1.1.1",
"winston": "^2.4.0"
},
"devDependencies": {
  "eslint": "^4.11.0",
  "eslint-config-standard": "^10.2.1",
  "eslint-plugin-import": "^2.8.0",
  "eslint-plugin-node": "^5.2.1",
  "eslint-plugin-promise": "^3.6.0",
  "eslint-plugin-standard": "^3.0.1"
}
}
```

需要注意的是, ESLint在全局安装并且执行init后会自动在当前项目创建config文件并且在package.json中的devDependencies属性里记录依赖信息, 同时需要在开发者使用的编辑器/IDE中安装相关插件, 这样当开发者使用这个编辑器编辑此项目中的文件时插件就会找到项目中的ESLint配置, 执行全局安装的ESLint分析配置文件, 根据配置内容检查开发者正在编辑的文件, 在编辑器中标识出不符合规范的语法或者语法错误;  
但是全局安装**EditorConfig**模块后, 并没有初始化的步骤, 所以需要手动在项目中创建相关配置文件, 并且最好在**package.json**的devDependencies中记录依赖(这样其他开发者就能快速了解这个项目开发中需要使用的模块), 当然这个模块也需要在开发者使用的编辑器/IDE中安装相关插件, 插件与模块配合作用的原理与ESLint类似;

补充:

1.如何在windows中创建文件名以"."开头的文件;

参考: <http://blog.csdn.net/zhu1500527791/article/details/69681374>

2.vi/vim, cat, touch, echo命令;

vi/vim命令打开文件后可以修改文件内容, cat只能用来查看但是不能修改文件内容; touch可以新建一个空文件, 也可以修改文件的创建时间;

echo可以创建或者写入文件内容, 如: echo hi > .npmignore的命令意思就是新建.npmignore文件并写入内容hi ;

所以如果在windows或者MAC中手动创建一个“开头的文件但是提示不允许创建此类文件, 那就可以在终端使用echo或者touch命令来创建, 如:

```
touch .editorconfig
```

```
ls -a
```

```
vim .editorconfig
```

.....编辑内容;

```
cat .editorconfig
```

3.关于vim相关命令;

参考:

<https://jingyan.baidu.com/article/495ba8410ff14d38b30ede01.html>

4.在终端中处理带空格的文件(夹)名;

如果在终端中需要处理类似: Program Files这样的文件名, 不能直接使用cd Program Files 这样的命令, 会报错, 而是需要将带空格的文件名使用双引号括起后再处理, 如: cd "Program Files" ;

2.配置文件;

通常情况下我们会将配置与代码分离, 将配置写到一个配置文件里, 如 config.js 或 config.json , 并放到项目的根目录下;

但实际开发中我们会有许多环境, 如: 本地开发环境、测试环境和线上环境等, 不同环境的配置不同 (如: MongoDB 的地址), 不可能每次部署时都要去修改对配置文件的引用, 如: config.test.js 或者 config.production.js; config-lite 模块就能解决这个问题;

**config-lite** 是一个轻量的读取配置文件的模块; config-lite 会根据环境变量 (NODE\_ENV) 的不同加载 config 目录下不同的配置文件; 如果不设置 NODE\_ENV, 则读取默认的 default 配置文件, 如果设置了 NODE\_ENV 则会合并指定的配置文件和 default 配置文件作为配置, config-lite 支持 .js、.json、.node、.yml、.yaml 后缀的文件;

例子:

如果程序以 NODE\_ENV=test node app 启动, 则 config-lite 会依次降级查找 config/test.js、config/test.json、config/test.node、config/test.yml、config/test.yaml 并合并 default 配置; 如果程序以 NODE\_ENV=production node app 启动, 则 config-lite 会依次降级查找 config/production.js、config/production.json、config/

production.node、config/production.yml、config/production.yaml 并合并 default 配置;  
config-lite 还支持冒泡查找配置, 即从传入的路径开始从该目录不断往上一级目录查找 config 目录, 直到找到或者到达根目录为止;

在myblog 下新建 config 目录, 在该目录下新建 default.js ;

### **config/default.js;**

```
module.exports = {
  port: 3000,
  session: {
    secret: 'myblog',
    key: 'myblog',
    maxAge: 2592000000
  },
  mongodb: 'mongodb://localhost:27017/myblog'
}
```

上面配置内容的含义:

- 1)port: 程序启动要监听的端口号;
- 2)session: express-session 的配置信息, 之后会介绍;
- 3)mongodb: mongodb 的地址, 以 mongodb:// 协议开头, myblog 为 db 名;

补充:

1.YML文件格式是YAML (YAML Ain't Markup Language)编写的文件格式, 为了强调这种语言以数据做为中心, 而不是以标记语言为重点;

YAML是一种直观的能够被电脑识别的数据序列化格式, 容易被人类阅读, 并且容易和脚本语言交互; YAML类似于XML, 但是语法比XML简单得多, 对于转化成数组或hash的数据处理起来非常简单有效;

例子:

```
name: Tom Smith
age: 37
spouse:
  name: Jane Smith
  age: 25
children:
- name: Jimmy Smith
  age: 15
- name1: Jenny Smith
```

age1: 12

参考:

[http://blog.sina.com.cn/s/blog\\_53ab41fd0102whll.html](http://blog.sina.com.cn/s/blog_53ab41fd0102whll.html)

3.myblog的功能与路由设计;

这里设计的myblog项目只关注与基本功能, 其余的功能 (如: 归档、标签、分页等等) 可另外实现;

并且由于页面是后端渲染的, 所以只通过简单的 <a>(GET) 和 <form>(POST) 与后端进行交互, 如果使用 jQuery 或者其他前端框架 (如 Angular、Vue、React 等等) 可通过 Ajax 与后端交互, 这种情况下 api 的设计应尽量遵循 Restful 风格;

功能及路由设计如下:

(1)注册

- <1>注册页: GET /signup
- <2>注册 (包含上传头像) : POST /signup

(2)登录

- <1>登录页: GET /signin
- <2>登录: POST /signin

(3)登出: GET /signout

(4)查看文章

- <1>主页: GET /posts
- <2>个人主页: GET /posts?author=xxx
- <3>查看一篇文章 (包含留言) : GET /posts/:postId

(5)发表文章

- <1>发表文章页: GET /posts/create
- <2>发表文章: POST /posts/create

(6)修改文章

- <1>修改文章页: GET /posts/:postId/edit
- <2>修改文章: POST /posts/:postId/edit

(7)删除文章: GET /posts/:postId/remove

(8)留言

- <1>创建留言: POST /posts/:postId/comment
- <2>删除留言: GET /posts/:postId/comment/:commentId/remove

4.Session;

由于 HTTP 协议是无状态的协议, 所以服务端需要记录用户的状态时, 就需要用某种机制来识别具体的用户, 这个机制就是会话 (Session) ;

这里通过引入 express-session 中间件实现对会话的支持:

```
app.use(session(options))
```

session中间件会在 req 上添加 session 对象, 即 req.session 初始值为 {}, 当登录后设置 req.session.user = 用户信息, 返回浏览器的头信息中会带上 set-cookie 将 session id 写到浏览器 cookie 中, 那么该用户下次请求时, 通过带上来的 cookie 中的 session id 就可以查找到该用户, 并将用户信息保存到 req.session.user;

## 5. 页面通知;

这里需要这样的功能: 当用户操作成功时需要显示一个成功的通知, 如: 登录成功跳转到主页时需要显示一个登陆成功的通知; 当用户操作失败时需要显示一个失败的通知, 如: 注册时用户名被占用了, 需要显示一个用户名已占用的通知; 通知只显示一次, 刷新后消失, 可以通过 connect-flash 中间件实现这个功能;

**connect-flash** 是基于 session 实现的, 它的原理很简单: 设置初始值 req.session.flash={}, 通过 req.flash(name, value) 设置这个对象下的字段和值, 通过 req.flash(name) 获取这个对象下的值, 同时删除这个字段, 实现了只显示一次刷新后消失的功能;

补充:

1.**express-session**、**connect-mongo** 和 **connect-flash** 的区别与联系;

1)express-session: 会话 (session) 支持中间件;

2)connect-mongo: 将 session 存储于 mongodb, 需结合 express-session 使用, 我们也可以将 session 存储于 redis, 如 **connect-redis**;

3)connect-flash: 基于 session 实现的用于通知功能的中间件, 需结合 express-session 使用;

## 6. 权限控制;

这里需要给博客添加权限控制, 只有登录用户可以留言或者编辑自己的文章, 未登录用户只能浏览;

可以把用户状态的检查封装成一个中间件, 在每个需要权限控制的路由加载该中间件, 即

可实现页面的权限控制; 在 myblog 下新建 middlewares 目录, 在该目录下新建 check.js:

### **middlewares/check.js;**

```
module.exports = {
  checkLogin: function checkLogin (req, res, next) {
    if (!req.session.user) {
      req.flash('error', '未登录')
      return res.redirect('/signin')
    }
    next()
  },

  checkNotLogin: function checkNotLogin (req, res, next) {
    if (req.session.user) {
      req.flash('error', '已登录')
      return res.redirect('back') // 返回到header中referer指定的链接;
    }
    next()
  }
}
```

上例的中间件模块的输出接口提供了两个函数:

- 1)checkLogin: 当用户信息 (req.session.user) 不存在, 即认为用户没有登录, 则跳转到登录页, 同时显示未登录的通知, 用于需要用户登录才能操作的页面;
- 2)checkNotLogin: 当用户信息 (req.session.user) 存在, 即认为用户已经登录, 则跳转到之前的页面, 同时显示已登录的通知, 如: 已登录用户尝试访问登录、注册页面;

补充:

1.Express中res.redirect('back')和res.location('back')区别;

**back**重定向, 重定向到请求的[referer](#), 当没有[referer](#)请求头的情况下, 默认为'/';

location()方法只会设置Location头, 而redirect()方法除了会设置Location头外还可自动或手动设置HTTP状态码;

location()方法实现过程大致如下:

```
res.location = function(url){
  var req = this.req;
```

```
// "back" 相当于 referrer的别名;
if ('back' == url) url = req.get('Referrer') || '/';

// 设置Location;
this.setHeader('Location', url);
return this;
};

res.location('back');
res.statusCode = 301;
res.end('响应的内容');
// 或:
res.location('back');
res.sent(302);
```

上例中可以发现, location()方法本质上是调用了ServerResponse对象的setHeader()方法, 但并没有设置状态码, 通过location()设置头信息后, 其后的代码还会执行;

redirect()方法是对location()方法的扩展; 通过location()设置Loction头后, 设置HTTP状态码(默认为302, 可以通过第一个参数指定为301), 最后通过ServerResponse对象的end()方法返回响应信息; 调用redirect()方法后, 其后的代码都不会被执行;

如:

```
res.redirect(301, 'back');
```

参考:

<https://www.cnblogs.com/duhuo/p/5609127.html>

7.创建路由文件;

**routes/index.js;**

```
module.exports = function (app) {
  app.get('/', function (req, res) {
    res.redirect('/posts')
  })
  app.use('/signup', require('./signup'))
  app.use('/signin', require('./signin'))
  app.use('/signout', require('./signout'))
  app.use('/posts', require('./posts'))
}
```

```
routes/posts.js;

const express = require('express')
const router = express.Router()

const checkLogin = require('../middlewares/check').checkLogin

// GET /posts 所有用户或者特定用户的文章页(如: GET /posts?author=xxx)
router.get('/', function (req, res, next) {
  res.send('主页')
})

// POST /posts/create 发表一篇文章
router.post('/create', checkLogin, function (req, res, next) {
  res.send('发表文章')
})

// GET /posts/create 发表文章页
router.get('/create', checkLogin, function (req, res, next) {
  res.send('发表文章页')
})

// GET /posts/:postId 指定的文章页
router.get('/:postId', function (req, res, next) {
  res.send('文章详情页')
})

// GET /posts/:postId/edit 更新文章页
router.get('/:postId/edit', checkLogin, function (req, res, next) {
  res.send('更新文章页')
})

// POST /posts/:postId/edit 更新一篇文章
router.post('/:postId/edit', checkLogin, function (req, res, next) {
  res.send('更新文章')
})

// GET /posts/:postId/remove 删除一篇文章
router.get('/:postId/remove', checkLogin, function (req, res, next) {
  res.send('删除文章')
})
```

```
// POST /posts/:postId/comment 创建一条留言
// router.post('/:postId/comment', checkLogin, function (req, res, next) {
//   res.send('创建留言')
// })

/// GET /posts/:postId/comment/:commentId/remove 删除一条留言
// router.get('/:postId/comment/:commentId/remove', checkLogin, function (req,
res, next) {
//   res.send('删除留言')
// })

module.exports = router
```

上例中的'/:postId/comment'和'/:postId/comment/:commentId/remove'这两个留言相关的路由之所以被注释了，是因为对创建/删除留言的路由操作不会通过这两个路径来实现，会在routes/comments.js中单独指定，之后会介绍；

### **routes/signin.js;**

```
const express = require('express')
const router = express.Router()

const checkNotLogin = require('../middlewares/check').checkNotLogin

// GET /signin 登录页
router.get('/', checkNotLogin, function (req, res, next) {
  res.send('登录页')

})

// POST /signin 用户登录
router.post('/', checkNotLogin, function (req, res, next) {
  res.send('登录')
})

module.exports = router
```

### **routes/signup.js;**

```
const express = require('express')
```

```
const router = express.Router()

const checkNotLogin = require('../middlewares/check').checkNotLogin

// GET /signup 注册页
router.get('/', checkNotLogin, function (req, res, next) {
  res.send('注册页')
})

// POST /signup 用户注册
router.post('/', checkNotLogin, function (req, res, next) {
  res.send('注册')
})

module.exports = router
```

### **routes/signout.js;**

```
const express = require('express')
const router = express.Router()

const checkLogin = require('../middlewares/check').checkLogin

// GET /signout 登出
router.get('/', checkLogin, function (req, res, next) {
  res.send('登出')
})

module.exports = router
```

### **根目录index.js;**

```
const path = require('path')
const express = require('express')
const session = require('express-session')
const MongoStore = require('connect-mongo')(session)
const flash = require('connect-flash')
const config = require('config-lite')(__dirname)
const routes = require('./routes') //如果指定的是一个文件夹(指定路径中不存在名为routes的文件)会自动去匹配routes文件夹下的index.js文件;
const pkg = require('./package')
```

```

const app = express()

// 设置模板目录
app.set('views', path.join(__dirname, 'views'))
// 设置模板引擎为 ejs
app.set('view engine', 'ejs')

// 设置静态文件目录
app.use(express.static(path.join(__dirname, 'public')))

// session 中间件
app.use(session({
  name: config.session.key, // 设置 cookie 中保存 session id 的字段名称
  secret: config.session.secret, // 通过设置 secret 来计算 hash 值并放在 cookie 中, 使产生的 signedCookie 防篡改
  resave: true, // 强制更新 session
  saveUninitialized: false, // 设置为 false, 强制创建一个 session, 即使用户未登录
  cookie: {
    maxAge: config.session.maxAge // 过期时间, 过期后 cookie 中的 session id 自动删除
  },
  store: new MongoStore({// 将 session 存储到 mongodb
    url: config.mongodb // mongodb 地址
  })
}))
// flash 中间件, 用来显示通知
app.use(flash())

// 路由
routes(app)

// 监听端口, 启动程序
app.listen(config.port, function () {
  console.log(` ${pkg.name} listening on port ${config.port}`)
})

```

需要注意的是, 中间件的加载顺序很重要; 如: 上面设置静态文件目录的中间件 express.static 需要在 routes(app) 之前加载, 这样对静态文件的请求就不会进入业务逻辑的路由里; flash 中间件在 session 中间件之后加载, 因为 flash 是基于 session 实现的;

需要注意的是, 设置了 express.static 中间件后对静态文件的访问不需要在 url 中添加 express.static 中指定的存放静态文件的目录(如果在 url 中加入这个目录名将访问不到静

态资源), 上例中是: public, 所以如果请求一个保存在myblog/public/img 下的.png文件, 那么请求路径应该类似:

[http://localhost:3000/img/upload\\_6695a0dadaaa3f8c23a39d6a466ba4c.png](http://localhost:3000/img/upload_6695a0dadaaa3f8c23a39d6a466ba4c.png)

关于express.static中间件的用法以及作用, 参考:

<http://www.expressjs.com.cn/starter/static-files.html>

运行 supervisor index 启动博客, 访问以下地址查看效果:

- 1) <http://localhost:3000/posts>
- 2) <http://localhost:3000/signout>
- 3) <http://localhost:3000/signup>

补充:

1. 安装/配置MongoDB, 之前在项目依赖中安装的**connect-mongo/mongolass**只是用来在**MongoDB**中存储**session**/操作**MongoDB**数据库的中间件, 而数据库本身需要事先安装和配置, 然后在启动**web**应用前先启动数据库;

(1) 安装MongoDB:

<https://www.mongodb.com/download-center#community>

(2) 在**mongodb**目录下新建二个目录: **data/db**, **etc**(存放**mongod.conf**);

**mongod.conf**内容:

```
#mongodb config file
dbpath=/Users/jiusong/mongodb/data/db/
logpath=/Users/jiusong/mongodb/mongod.log
logappend = true
port = 27017
fork = true
auth = true
```

(3) 配置环境变量;

`echo 'export PATH=/Users/jiusong/mongodb/bin:$PATH'>>~/.bash_profile`

添加完成后为使环境变量生效, 可重启**shell**终端或在用户根目录(`cd ~`)输入命令:

`source .bash_profile`

查看环境变量是否添加成功: `echo $PATH`

(4) 添加操作权限;

`sudo chown -R jiusong /data/db`

(5)进入mongodb的"bin"目录(此操作必须在第(3)步操作相同的终端session中进行),使用命令"./mongod"或"mongod"启动MongoDB server,启动成功后最后一行应该是端口号,如:

```
waiting for connections on port 27017
```

如果mongod命令报错: exception in initAndListen: 29 Data directory /data/db not found., terminating

那么使用下面的命令来显示指定/data/db路径:

```
mongod --dbpath /Users/jiusong/mongodb/data/db
```

(6)启动之后在浏览器中访问: localhost:27017, 显示:

```
It looks like you are trying to access MongoDB over HTTP on the native driver port.
```

注意:

1.将上面的命令行直接复制到终端执行可能会出现command not found之类的报错,需要复制到终端后手动删除并重新添加这些命令行中的空格来解决;

关于mongodb的安装与配置,参考:

- (1)<http://www.cnblogs.com/wx1993/p/5187530.html>
- (2)<https://www.cnblogs.com/baiyunchen/p/5111674.html>

2.运行node项目时可能会报错: Error: listen EADDRINUSE :::3000, 这是端口已被占用的错误,解决方法:

(1)查看占用端口的进程;

```
sudo lsof -i:3000
```

(2)显示如下:

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
Google	662	jiusong	101u	IPv6	0x6e776494d64f64d7	0t0	TCP	localhost:59094->localhost:hbc (CLOSE_WAIT)
node	3241	jiusong	13u	IPv6	0x6e776494da1b8a17	0t0	TCP	*:hbc (LISTEN)

(3)关闭某个占用端口的进程;

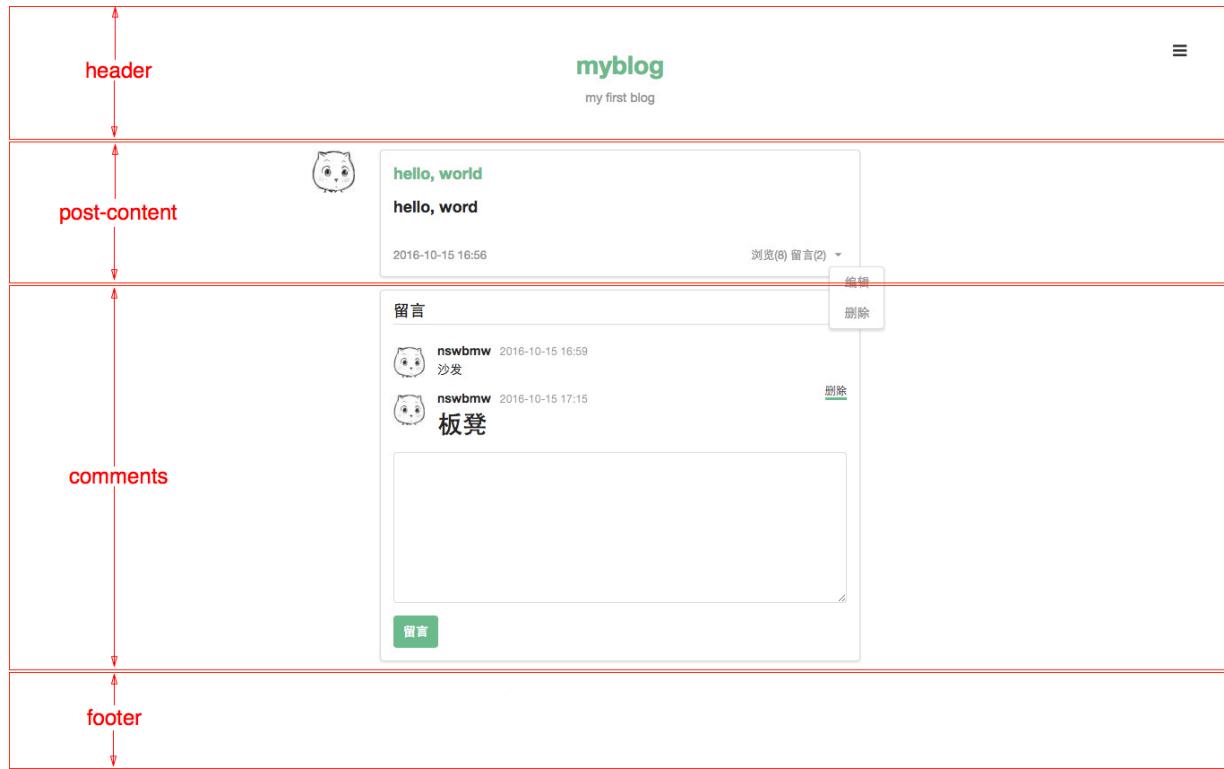
```
sudo kill 3241
```

8.页面设计;

(1)模板与组件;

这里的组件指的是模板片段, 这里需要将模板拆分成一些组件, 然后使用 ejs 的 include 方法将组件组合起来进行渲染;

组件的拆分参考:



### views/header.ejs;

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title><%= blog.title %></title>
    <link rel="stylesheet" href="//cdn.bootcss.com/semantic-ui/2.1.8/
semantic.min.css">
    <link rel="stylesheet" href="/css/style.css">
    <script src="//cdn.bootcss.com/jquery/1.11.3/jquery.min.js"></script>
    <script src="//cdn.bootcss.com/semantic-ui/2.1.8/semantic.min.js"></script>
  </head>
  <body>
    <%- include('components/nav') %>
    <%- include('components/nav-setting') %>
```

```
<%- include('components/notification') %>
```

### **views/components/nav.ejs;**

```
<div class="nav">
  <div class="ui grid">
    <div class="four wide column"></div>

    <div class="eight wide column">
      <a href="/posts"><h1><%= blog.title %></h1></a>
      <p><%= blog.description %></p>
    </div>
  </div>
</div>
```

### **views/components/nav-setting.ejs;**

```
<div class="nav-setting">
  <div class="ui buttons">
    <div class="ui floating dropdown button">
      <i class="icon bars"></i>
      <div class="menu">
        <% if (user) { %>
          <a class="item" href="/posts?author=<%= user._id %>">个人主页</a>
          <div class="divider"></div>
          <a class="item" href="/posts/create">发表文章</a>
          <a class="item" href="/signout">登出</a>
        <% } else { %>
          <a class="item" href="/signin">登录</a>
          <a class="item" href="/signup">注册</a>
        <% } %>
      </div>
    </div>
  </div>
</div>
```

### **views/components/notification.ejs;**

```
<div class="ui grid">
  <div class="four wide column"></div>
```

```
<div class="eight wide column">

<% if (success) { %>
  <div class="ui success message">
    <p><%= success %></p>
  </div>
<% } %>

<% if (error) { %>
  <div class="ui error message">
    <p><%= error %></p>
  </div>
<% } %>

</div>
</div>
```

### **views/footer.ejs;**

```
<script type="text/javascript">
// 使通知框延时自动从页面删除
$(document).ready( function () {
  // 延时清除掉成功、失败提示信息
  if($('.ui.success.message').length > 0) {
    $('.ui.success.message').fadeOut(3000)
  } else if($('.ui.error.message').length > 0) {
    $('.ui.error.message').fadeOut(3000)
  }

  // 初始化下拉框;
  $('.ui.dropdown').dropdown();

  // 鼠标悬浮在头像上， 弹出气泡提示框
  $('.post-content .avatar-link').popup({
    inline: true,
    position: 'bottom right',
    lastResort: 'bottom right'
  });
})
</script>
</body>
</html>
```

上面 <script></script> 中是 semantic-ui 操控页面控件的代码, 一定要放到 footer.ejs 的 </body> 的前面, 因为只有页面加载完后才能通过 JQuery 获取 DOM 元素;

补充:

### 1. Semantic-UI;

Semantic UI 是完全语义化的前端UI框架, 与其它UI框架如: Bootstrap 和 Foundation 比起来, 它使用起来更加简洁;

由于 semantic 的 js 动作依赖与 jQuery 文件, 所以 jQuery 文件一定要先与 semantic 的 js 引用;

这里引用的是外链的 semantic js 和 css, 更好的方式是将其作为本地依赖安装(这样就能本地定制一些框架的配置):

终端执行:

```
npm install semantic-ui --save  
cd semantic/  
gulp build
```

HTML 中引用它:

```
<link rel="stylesheet" type="text/css" href="semantic/dist/semantic.min.css">  
<script src="semantic/dist/semantic.min.js"></script>
```

Semantic-UI 的具体用法, 参考:

<http://www.semantic-ui.cn/introduction/getting-started.html>

### (2) app.locals 和 res.locals;

修改根目录 index.js, 在 routes(app) 这条代码之前添加:

```
// 设置模板全局常量  
app.locals.blog = {  
  title: pkg.name,  
  description: pkg.description  
  
}  
  
// 添加模板必需的三个变量  
app.use(function (req, res, next) {  
  res.locals.user = req.session.user
```

```
res.locals.success = req.flash('success').toString()
res.locals.error = req.flash('error').toString()
next()
})
```

上面的 ejs 模板中使用了 blog、user、success、error 变量, blog变量挂载到了 app.locals 下, user、success、error 挂载到了res.locals下;  
那么 app.locals 和 res.locals 的作用和区别是什么?

express 源码:

#### **express/lib/application.js;**

```
app.render = function render(name, options, callback) {
  ...
  var opts = options;
  var renderOptions = {};
  ...
  // merge app.locals
  merge(renderOptions, this.locals);

  // merge options._locals
  if (opts._locals) {
    merge(renderOptions, opts._locals);
  }

  // merge options
  merge(renderOptions, opts);
  ...
  tryRender(view, renderOptions, done);
};
```

#### **express/lib/response.js;**

```
res.render = function render(view, options, callback) {
  var app = this.req.app;
  var opts = options || {};
  ...
  // merge res.locals
  opts._locals = self.locals;
  ...
  // render
```

```
    app.render(view, opts, done);
};
```

从上面的express源码中可以发现, res.render函数内容最终调用了app.render函数, 将res.locals挂载到options对象的\_locals属性中后将这个options对象做为options对象参数传入app.render函数, 根据app.render函数中三处合并(merge)操作, 可以总结出最终传入将要渲染模板的options参数对象优先级为:

res.render 传入的模板参数对象 > res.locals 对象 > app.locals 对象(如果出现同名属性, 高权限对象的属性会覆盖低权限的对象同名属性);

app.locals 和 res.locals 都用来渲染模板, 区别在于: app.locals 上通常挂载常量信息  
(如: 站点的名字、描述、作者这种一般不会变更的信息), res.locals 上通常挂载变量信息, 即每次请求可能的值都不一样 (如: res.locals.user = req.session.user)

所以, 之前在根目录index.js中添加的那段内容可以让 express 为我们自动 merge 并在调用render方法时传入这四个变量(blog、user、success、error)到模板中, 所以我们之后可以在模板中直接使用它们;

(3)样式文件;

**public/css/style.css;**

```
/* ----- 全局样式 ----- */

body {
  width: 1100px;
  height: 100%;
  margin: 0 auto;
  padding-top: 40px;
}

a:hover {
  border-bottom: 3px solid #4fc08d;
}

.button {
  background-color: #4fc08d !important;
  color: #fff !important;
}

.avatar {
  border-radius: 3px;
```

```
width: 48px;
height: 48px;
float: right;
}

/* ----- nav ----- */

.nav {
margin-bottom: 20px;
color: #999;
text-align: center;
}

.nav h1 {
color: #4fc08d;
display: inline-block;
margin: 10px 0;
}

/* ----- nav-setting ----- */

.nav-setting {
position: fixed;
right: 30px;
top: 35px;
z-index: 999;
}

.nav-setting .ui.dropdown.button {
padding: 10px 10px 0 10px;
background-color: #fff !important;
}

.nav-setting .icon.bars {
color: #000;
font-size: 18px;
}

/* ----- post-content ----- */

.post-content h3 a {
color: #4fc08d !important;
}
```

```
.post-content .tag {  
    font-size: 13px;  
    margin-right: 5px;  
    color: #999;  
}  
  
.post-content .tag.right {  
    float: right;  
    margin-right: 0;  
}  
  
.post-content .tag.right a {  
    color: #999;  
}
```

## 9.数据库;

之前安装的 **Mongolass** 模块是mondgo驱动, 用来操作 mongodb 进行增删改查;

(1)在 myblog 下新建 lib 目录, 在该目录下新建 mongo.js;

**lib/mongo.js;**

```
const config = require('config-lite')(__dirname)  
const Mongolass = require('mongolass')  
const mongolass = new Mongolass()  
mongolass.connect(config.mongodb)
```

Mongolass模块的优势是:

Mongolass 保持了与 mongodb官方 一样的 api, 又借鉴了许多 Mongoose(功能强大但是过于复杂) 的优点(支持 Promise, 文档校验等), 同时又保持了精简;

**node-mongodb-native, Mongoose和Mongolass 的优缺点比较, 参考:**

<https://github.com/nswbmw/N-blog/blob/master/book/4.6%20%E8%BF%9E%E6%8E%A5%E6%95%B0%E6%8D%AE%E5%BA%93.md>

例子:

```
const Mongolass = require('mongolass')
```

```

const mongolass = new Mongolass('mongodb://localhost:27017/test')

const User = mongolass.model('User', {
  name: { type: 'string' },
  age: { type: 'number' }
})

User
  .insertOne({ name: 'nswbmw', age: 'wrong age' })
  .exec()
  .then(console.log)
  .catch(function (e) {
    console.error(e)
    console.error(e.stack)
  })

```

上例会报错: Error: (\$.\_age: "wrong age") ✘ (type: number), 错误的原因是在 insertOne 一条用户数据到用户表的时候, age 期望是一个 number 类型的值, 而这里传入的字符串 wrong age;

## (2) 用户模型设计;

这里只存储用户的名称、密码（加密后的）、头像、性别和个人简介这几个字段;

### **lib/mongo.js;**

```

exports.User = mongolass.model('User', {
  name: { type: 'string' },
  password: { type: 'string' },
  avatar: { type: 'string' },
  gender: { type: 'string', enum: ['m', 'f', 'x'] },
  bio: { type: 'string' }
})
exports.User.index({ name: 1 }, { unique: true }).exec() //根据用户名找到用户, 数字1 表示name键的索引按升序存储(-1表示键的索引按照降序存储), 用户名全局唯一;

```

上面定义并导出了 User 这个 model, 同时设置了 name 的唯一索引, 保证用户名是不重复的;

注意:

1. Mongolass 中的 model 相当于 mongodb 中的 collection, 只不过添加了插件的功能;

补充:

1.MongoDB的索引;

参考:

<https://www.cnblogs.com/stephen-liu74/archive/2012/08/01/2561557.html>

2.数据库中的Schema为**数据库对象**的集合,一个用户一般对应一个schema,即schema的个数同user的个数相同,而且schema名字同user名字一一对应并且相同,所以我们可以说schema为user的别名;

A schema is a collection of database objects (used by a user).  
schema objects are the logical structures that directly refer to the database's data. (eg: tables views sequences stored procedures synonyms indexes clusters and database links)

A user is a name defined in the database that can connect to and access objects. schemas and users help database administrators manage database security.

参考:

[https://www.biaodianfu.com/database-schema.html?utm\\_source=tuicool&utm\\_medium=referral](https://www.biaodianfu.com/database-schema.html?utm_source=tuicool&utm_medium=referral)

3.MongoDB终端命令大全;

进入mongodb/bin路径后在命令行输出mongo后就可以进行数据库操作了;

参考:

<https://www.cnblogs.com/xusir/archive/2012/12/24/2830957.html>

10.注册页;

**views/signup.ejs;**

```
<%- include('header') %>

<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">
    <form class="ui form segment" method="post" enctype="multipart/form-data">
      <div class="field required">
        <label>用户名</label>
        <input placeholder="用户名" type="text" name="name">
```

```

</div>
<div class="field required">
    <label>密码</label>
    <input placeholder="密码" type="password" name="password">
</div>
<div class="field required">
    <label>重复密码</label>
    <input placeholder="重复密码" type="password" name="repassword">
</div>
<div class="field required">
    <label>性别</label>
    <select class="ui compact dropdown" name="gender">
        <option value="">性别</option>
        <option value="m">男</option>
        <option value="f">女</option>
        <option value="x">保密</option>
    </select>
</div>
<div class="field required">
    <label>头像</label>
    <input type="file" name="avatar">
</div>
<div class="field required">
    <label>个人简介</label>
    <textarea name="bio" rows="5"></textarea>
</div>
<input type="submit" class="ui button fluid" value="注册">
</form>
</div>
</div>

<%- include('footer') %>

```

注意:

1.form 表单要添加 enctype="multipart/form-data" 属性才能上传文件;

修改 routes/signup.js 中获取注册页的路由;

```

// GET /signup 注册页
router.get('/', checkNotLogin, function (req, res, next) {
    res.render('signup')
})

```

访问 localhost:3000/signup, 可以在页面上看到如下效果:

The screenshot shows a registration form titled "myblog" with the subtitle "my first blog". The form fields include:

- 用户名 \*: An input field labeled "用户名".
- 密码 \*: An input field labeled "密码".
- 重复密码 \*: An input field labeled "重复密码".
- 性别 \*: A dropdown menu showing "男" (Male).
- 头像 \*: A file upload input field with the placeholder "选择文件" (Select file) and "未选择任何文件" (No file selected).
- 个人简介 \*: A large text area labeled "个人简介".

A green "注册" (Register) button is at the bottom.

11.关于semantic-ui 的相关内容可以参考: 'semantic-ui 的使用'笔记;

.....

12.注册与文件上传;

(1)使用 **express-formidable** 处理 form 表单 (包括文件上传) ;

在index.js中'app.use(flash())'之后添加:

```
// 处理表单及文件上传的中间件
app.use(require('express-formidable')({
  uploadDir: path.join(__dirname, 'public/img'), // 上传文件目录
  keepExtensions: true// 保留后缀
}))
```

(2)新建 models/users.js;

**models/users.js;**

```
const User = require('../lib/mongo').User

module.exports = {
  // 注册一个用户
  create: function create (user) {
    return User.create(user).exec()
  }
}
```

(3)修改routes/signup.js;

**routes/signup.js;**

```
const fs = require('fs')
const path = require('path')
const sha1 = require('sha1')
const express = require('express')
const router = express.Router()

const UserModel = require('../models/users')
const checkNotLogin = require('../middlewares/check').checkNotLogin

// GET /signup 注册页
router.get('/', checkNotLogin, function (req, res, next) {
  res.render('signup')
})

// POST /signup 用户注册
router.post('/', checkNotLogin, function (req, res, next) {
  const name = req.fields.name
  const gender = req.fields.gender
  const bio = req.fields.bio
  const avatar = req.files.avatar.path.split(path.sep).pop()
  let password = req.fields.password
  const repassword = req.fields.repassword

  // 校验参数
  try {
    if (!(name.length >= 1 && name.length <= 10)) {
```

```
        throw new Error('名字请限制在 1-10 个字符')
    }
    if (['m', 'f', 'x'].indexOf(gender) === -1) {
        throw new Error('性别只能是 m、f 或 x')
    }
    if (!(bio.length >= 1 && bio.length <= 30)) {
        throw new Error('个人简介请限制在 1-30 个字符')
    }
    if (!req.files.avatar.name) {
        throw new Error('缺少头像')
    }
    if (password.length < 6) {
        throw new Error('密码至少 6 个字符')
    }
    if (password !== repassword) {
        throw new Error('两次输入密码不一致')
    }
} catch (e) {
    // 注册失败，异步删除上传的头像
    fs.unlink(req.files.avatar.path)
    req.flash('error', e.message)
    return res.redirect('/signup')
}

// 明文密码加密
password = sha1(password)

// 待写入数据库的用户信息
let user = {
    name: name,
    password: password,
    gender: gender,
    bio: bio,
    avatar: avatar
}
// 用户信息写入数据库
UserModel.create(user)
.then(function (result) {
    // 此 user 是插入 mongodb 后的值，包含 _id
    user = result.ops[0]
    // 删除密码这种敏感信息，将用户信息存入 session
    delete user.password
    req.session.user = user
})
```

```

// 写入 flash
req.flash('success', '注册成功')
// 跳转到首页
res.redirect('/posts')
})
.catch(function (e) {
// 注册失败，异步删除上传的头像
fs.unlink(req.files.avatar.path)
// 用户名被占用则跳回注册页，而不是错误页
if (e.message.match('duplicate key')) {
req.flash('error', '用户名已被占用')
return res.redirect('/signup')
}
next(e)
})
})

```

module.exports = router

上例中, 使用 express-formidable 处理表单的上传, 表单普通字段挂载到 req.fields 上, 表单上传后的文件挂载到 req.files 上, 文件存储在 public/img 目录下;  
然后校验收到的表单字段, 校验通过后将用户信息插入到 MongoDB 中, 成功则跳转到主页并显示‘注册成功’的通知, 失败 (如用户名被占用) 则跳转回注册页面并显示‘用户名已被占用’的通知;  
注册失败时 (参数校验失败或者存数据库时出错), 需要删除已经上传到 public/img 目录下的头像;

注意:

1. 使用 sha1 加密用户的密码, sha1 并不是一种十分安全的加密方式, 实际开发中可以使用更安全的 **bcrypt** 或 **scrypt** 加密;
2. 启动 node 项目后可能会报错: Block-scoped declarations (let, const, function, class) not yet supported outside strict mode, 这是由于在块级作用域中使用了 ES6 的语法, 此时需要在这个模块文件的开头添加: 'use strict' ;
3. 启动 node 项目后还可能会报错:

/Users/jiusong/myblog/node\_modules/mongolass/lib/query.js:52

```

this[plugin.name] = (...args) => {
    ^
    ^
    ^
```

SyntaxError: Unexpected token ...

这是由于 node 版本过低造成的问题, 升级 node 版本即可:

n 模块是专门用来管理 node.js 的版本的:

npm install -g n

升级node.js到最新稳定版:

n stable

也可以指定版本号:

n v9.2.0

补充:

1.path.sep 用来返回各平台的分隔符;

如:

'foo/bar/baz'.split(path.sep); // \*nix 返回['foo', 'bar', 'baz']

'foo\\bar\\baz'.split(path.sep) //windows 返回 ['foo', 'bar', 'baz']

2.启动node项目时可能会报错: MaxListenersExceededWarning: Possible EventEmitter memory leak detected;

参考:

<https://stackoverflow.com/questions/9768444/possible-eventemitter-memory-leak-detected>

(4)修改 routes/posts.js;

**routes/posts.js;**

```
router.get('/', function (req, res, next) {  
  res.render('posts')  
})
```

(5)新建 views/posts.ejs;

**views/posts.ejs;**

<%- include('header') %>

这是主页

<%- include('footer') %>

13.登出;

修改 routes/signout.js;

```
routes/signout.js;

const express = require('express')
const router = express.Router()

const checkLogin = require('../middlewares/check').checkLogin

// GET /signout 登出
router.get('/', checkLogin, function (req, res, next) {
  // 清空 session 中用户信息
  req.session.user = null
  req.flash('success', '登出成功')
  // 登出成功后跳转到主页
  res.redirect('/posts')
})

module.exports = router
```

然后点击右上角下拉栏中的登出, 页面显示如下:



14. 登录页;

(1) 修改 routes/signin.js;

**routes/signin.js;**

```
router.get('/', checkNotLogin, function (req, res, next) {
  res.render('signin')
})
```

(2) 新建 views/signin.ejs;

```
views/signin.ejs;

<%- include('header') %>

<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">
    <form class="ui form segment" method="post">
      <div class="field required">
        <label>用户名</label>
        <input placeholder="用户名" type="text" name="name">
      </div>
      <div class="field required">
        <label>密码</label>
        <input placeholder="密码" type="password" name="password">
      </div>
      <input type="submit" class="ui button fluid" value="登录">
    </form>
  </div>
</div>

<%- include('footer') %>
```

15. 登录；

(1) 修改 models/users.js；

**models/users.js;**

```
const User = require('../lib/mongo').User

module.exports = {
  // 注册一个用户
  create: function create (user) {
    return User.create(user).exec()
  },

  // 通过用户名获取用户信息
  getUserByName: function getUserByName (name) {
    return User
      .findOne({ name: name })
```

```
.addCreatedAt()  
.exec()  
}  
}
```

上例中使用了 addCreatedAt 自定义插件（通过 \_id 生成时间戳）；

(2)修改 lib/mongo.js, 添加下面的代码;

### **lib/mongo.js;**

```
const moment = require('moment')  
const objectIdToTimestamp = require('objectid-to-timestamp')  
  
// 根据 id 生成创建时间 created_at  
mongolass.plugin('addCreatedAt', {  
  afterFind: function (results) {  
    results.forEach(function (item) {  
      item.created_at = moment(objectIdToTimestamp(item._id)).format('YYYY-MM-  
DD HH:mm')  
    })  
    return results  
  },  
  afterFindOne: function (result) {  
    if (result) {  
      result.created_at = moment(objectIdToTimestamp(result._id)).format('YYYY-  
MM-DD HH:mm')  
    }  
    return result  
  }  
})
```

上例中, 自定义的addCreatedAt插件是通过objectIdToTimestamp和moment实现的, 作用是将查找到的项的\_id解析为指定格式的时间, 这个时间就是这个项在数据库中的创建时间;

需要注意的是, 使用mongolass的find()方法返回的是一个数组(如果只能查询到一个结果也会放在数组中), 如果使用findOne()方法返回的一定是一个单一的对象;

(3)修改 routes/signin.js;

```
routes/signin.js;
```

```
const sha1 = require('sha1')
const express = require('express')
const router = express.Router()

const UserModel = require('../models/users')
const checkNotLogin = require('../middlewares/check').checkNotLogin

// GET /signin 登录页
router.get('/', checkNotLogin, function (req, res, next) {
  res.render('signin')
})

// POST /signin 用户登录
router.post('/', checkNotLogin, function (req, res, next) {
  const name = req.fields.name
  const password = req.fields.password

  // 校验参数
  try {
    if (!name.length) {
      throw new Error('请填写用户名')
    }
    if (!password.length) {
      throw new Error('请填写密码')
    }
  } catch (e) {
    req.flash('error', e.message)
    return res.redirect('back')
  }

  UserModel.getUserByName(name)
    .then(function (user) {
      if (!user) {
        req.flash('error', '用户不存在')
        return res.redirect('back')
      }
      // 检查密码是否匹配
      if (sha1(password) !== user.password) {
        req.flash('error', '用户名或密码错误')
        return res.redirect('back')
      }
    })
})
```

```

    req.flash('success', '登录成功')
    // 用户信息写入 session
    delete user.password
    req.session.user = user
    // 跳转到主页
    res.redirect('/posts')
  })
  .catch(next)
}

module.exports = router

```

上例中，在POST /signin 的路由处理函数中，通过传上来的 name 去数据库中找到对应用户，校验传上来的密码是否跟数据库中的一致；不一致则返回上一页（即登录页）并显示‘用户名或密码错误’的通知，一致则将用户信息写入 session，跳转到主页并显示‘登录成功’的通知；

可以注意到的是，无论是对数据的创建、插入还是查询操作，mongolass的api基本都是以 promise的形式返回的；

在登录页面成功登陆后，页面效果如下：



16. 文章模型；

这里只存储文章的作者 id、标题、正文和点击量这几个字段，对应修改 lib/mongo.js；

**lib/mongo.js；**

```

exports.Post = mongolass.model('Post', {
  author: { type: Mongolass.Types.ObjectId },
  title: { type: 'string' },

```

```
content: { type: 'string' },
pv: { type: 'number' }
})
exports.Post.index({ author: 1, _id: -1 }).exec() // 按创建时间降序查看用户的文章列表;
```

17.发表文章;

(1)创建发表文章页，新建 views/create.ejs;

### **views/create.ejs;**

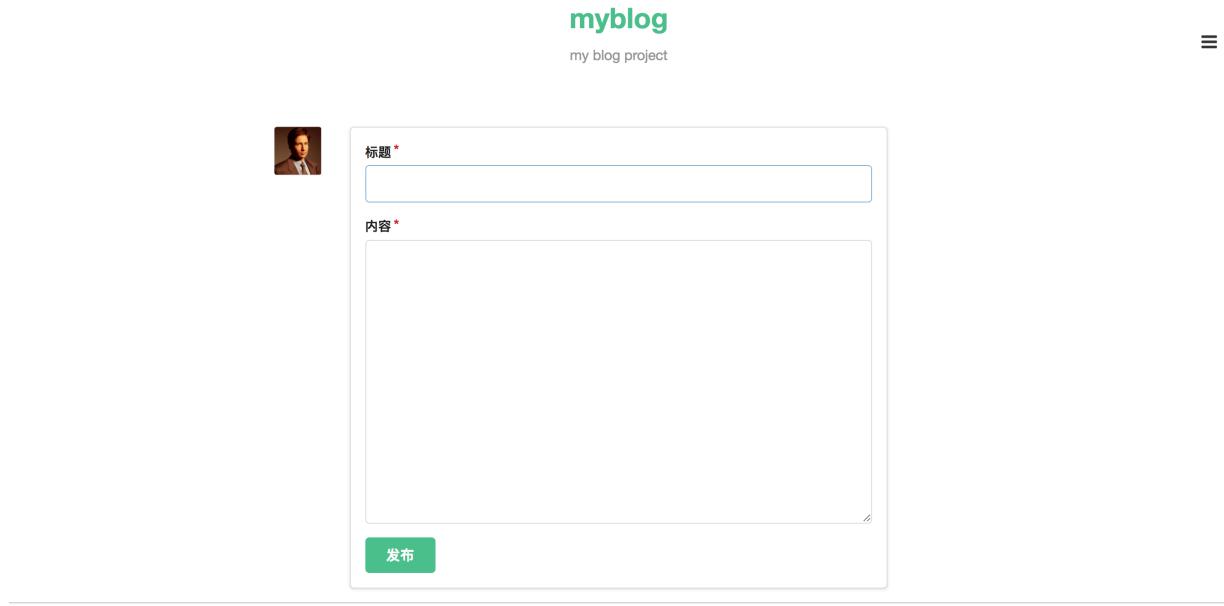
```
<%- include('header') %>

<div class="ui grid">
  <div class="four wide column">
    <a class="avatar avatar-link"
      href="/posts?author=<%= user._id %>"
      data-title="<%= user.name %> | <%= ({m: '男', f: '女', x: '保密'})[user.gender]
%>" 
      data-content="<%= user.bio %>">
      
    </a>
  </div>

  <div class="eight wide column">
    <form class="ui form segment" method="post">
      <div class="field required">
        <label>标题</label>
        <input type="text" name="title">
      </div>
      <div class="field required">
        <label>内容</label>
        <textarea name="content" rows="15"></textarea>
      </div>
      <input type="submit" class="ui button" value="发布">
    </form>
  </div>
</div>

<%- include('footer') %>
```

登录成功状态下，点击右上角‘发表文章’后页面如下：



注意：

1.avatar类在style.css中定义如下：

```
.avatar {  
    border-radius: 3px;  
    width: 48px;  
    height: 48px;  
    float: right;  
}
```

2.上例中的avator图片链接的popup行为定义在了footer.ejs中：

```
$('.post-content .avatar-link').popup({  
    inline: true,  
    position: 'bottom right',  
    lastResort: 'bottom right'  
});
```

上例中，.post-content这个类名目前还未指定，之后应该会设置；

inline, lastResort属性的作用可以参考：

<https://semantic-ui.com/modules/popup.html#/settings> (关键字: inline, lastResort)

popup效果如下:



(2)修改 routes/posts.js;

**routes/posts.js;**

```
// GET /posts/create 发表文章页;
router.get('/create', checkLogin, function (req, res, next) {
  res.render('create')
})
```

(3)新建 models/posts.js 用来存放与文章模型操作相关的代码;

**models/posts.js;**

```
const Post = require('../lib/mongo').Post

module.exports = {
  // 创建一篇文章
  create: function create (post) {
    return Post.create(post).exec()
  }
}
```

(4)修改 routes/posts.js;

**routes/posts.js;**

```
const PostModel = require('../models/posts')
.....
// POST /posts/create 发表一篇文章
router.post('/create', checkLogin, function (req, res, next) {
```

```

const author = req.session.user._id
const title = req.fields.title
const content = req.fields.content

// 校验参数
try {
  if (!title.length) {
    throw new Error('请填写标题')
  }
  if (!content.length) {
    throw new Error('请填写内容')
  }
} catch (e) {
  req.flash('error', e.message)
  return res.redirect('back')
}

let post = {
  author: author,
  title: title,
  content: content,
  pv: 0
}

PostModel.create(post)
  .then(function (result) {
    // 此 post 是插入 mongodb 后的值，包含 _id
    post = result.ops[0]
    req.flash('success', '发表成功')
    // 发表成功后跳转到该文章页
    res.redirect(`/posts/${post._id}`)
  })
  .catch(next)
})

```

上例中校验了上传的表单字段，并将文章信息插入数据库，成功后跳转到该文章页并显示‘发表成功’的通知，失败后请求会进入错误处理中间件；

18.主页与文章页；

(1)修改 models/posts.js;

## **models/posts.js;**

```
const marked = require('marked')
const Post = require('../lib/mongo').Post

// 将 post 的 content 从 markdown 转换成 html
Post.plugin('contentToHtml', {
  afterFind: function (posts) {
    return posts.map(function (post) {
      post.content = marked(post.content)
      return post
    })
  },
  afterFindOne: function (post) {
    if (post) {
      post.content = marked(post.content)
    }
    return post
  }
})

module.exports = {
  // 创建一篇文章
  create: function create (post) {
    return Post.create(post).exec()
  },

  // 通过文章 id 获取一篇文章
  getPostById: function getPostById (postId) {
    return Post
      .findOne({ _id: postId })
      .populate({ path: 'author', model: 'User' })
      .addCreatedAt()
      .contentToHtml()
      .exec()
  },

  // 按创建时间降序(后创建的排在前面)获取所有用户文章或者某个特定用户的所有文章
  getPosts: function getPosts (author) {
    const query = {}
    if (author) {
```

```

        query.author = author
    }
    return Post
        .find(query)
        .populate({ path: 'author', model: 'User' })
        .sort({ _id: -1 })
        .addCreatedAt()
        .contentToHtml()
        .exec()
    },
}

// 通过文章 id 给 pv 加 1
incPv: function incPv (postId) {
    return Post
        .update({ _id: postId }, { $inc: { pv: 1 } })
        .exec()
}
}

```

注意:

- 1.上例中将文章的内容以markdown 格式来解析, 所以在发表文章的时候可使用 markdown 语法 (如插入链接、图片等等), 关于 markdown 的使用可以参考:  
[Markdown 语法说明](#);
- 2.上例中在 PostModel 上注册了 contentToHtml, 而 addCreatedAt 是在 lib/mongo.js 中 mongolass 上注册的; 也就是说 contentToHtml 只针对 PostModel 有效, 而 addCreatedAt 对所有 Model 都有效;

补充:

1.markdown(md)格式的文件;

md = markdown;

Markdown 是一种轻量级标记语言, 它允许人们“使用易读易写的纯文本格式编写文档, 然后转换成有效的XHTML(或者HTML)文档;

Markdown 的生成器有足够的智能, 不需要额外标注这是 HTML 或是 Markdown; 只要直接加标签就可以了;

(1)一些HTML区块元素(块级元素)——比如 <div>、<table>、<pre>、<p> 等标签, 必须在前后加上空行与其它非HTML内容区隔开, 还要求它们的开始标签与结尾标签不能用制表符或空格来缩进; Markdown 的生成器足够智能, 不会在 HTML 区块标签外加上

不必要的 <p> 标签;

例子(在 Markdown 文件里加上一段 HTML 表格):

这是一个普通段落。

```
<table>
  <tr>
    <td>Foo</td>
  </tr>
</table>
```

这是另一个普通段落。

需要注意的是, 在 HTML 区块标签间的 Markdown 格式语法将不会被处理; 比如: 在 HTML 区块内使用 Markdown 样式的\*强调\*不会有效果;

(2)HTML的区段(行内)标签如 <span>、<cite>、<del> 可以在 Markdown 的段落、列表或是标题里随意使用;

需要注意的是, Markdown 语法在 HTML 区段标签间是有效的;

特殊字符自动转换:

在 HTML 文件中, 有些字符比较特殊, 如: '<' 和 '&' ; '<' 符号用于起始标签, '&' 符号则用于标记 HTML 实体; 所以如果只是想要显示这些字符的原型, 必须要使用字符实体的形式, 像是 &lt; 和 &amp;;

而由于 markdown 最终会被转换为 html, 所以其中哪些字符需要以其原型显示, 哪些属于有特殊功能的字符就需要被明确区分, 好在 Markdown 的设计让开发者可以自然地书写字符, 需要转换的由它来处理好了;

例子1:

使用的 & 字符是 HTML 字符实体的一部分时, 它会保留原状, 否则它会被转换成 &amp; ; &copy; 会被保留, 使其能够在页面上显示版权符号©, 而 AT&T 会被 markdown 转换为 AT&amp;T, 使其能够在页面上显示字符的原型'AT&T';

例子2:

如果把 '<' 符号作为 HTML 标签的定界符使用, 那 Markdown 也不会对它做任何转换, 但是如果只是如:

4 < 5

Markdown 将会把它转换为:

4 &lt; 5

npm上, 下载比较多的markdown工具有两款: marked和markdown, 它们都能够把md文件转换为html;

参考:

<http://wowubuntu.com/markdown/>

<https://www.cnblogs.com/liugang-vip/p/6337580.html>

2. populate方法;

MongoDB是非关联数据库, 但是有时候需要关联查询, 就会用到populate方法;

参考:

<http://www.jb51.net/article/119138.htm>

(2)修改 views/posts.ejs;

**views/posts.ejs;**

```
<%- include('header') %>

<% posts.forEach(function (post) { %>
  <%- include('components/post-content', { post: post }) %>
<% }) %>

<%- include('footer') %>
```

(3)新建 views/components/post-content.ejs 用来存放单篇文章的模板片段;

**views/components/post-content.ejs;**

```
<div class="post-content">
  <div class="ui grid">
    <div class="four wide column">
      <a class="avatar avatar-link"
        href="/posts?author=<%= post.author._id %>"
        data-title="<%= post.author.name %> | <%= ({m: '男', f: '女', x: '保密'})[post.author.gender] %>"
        data-content="<%= post.author.bio %>">
        
```

```

        </a>
    </div>

    <div class="eight wide column">
        <div class="ui segment">
            <h3><a href="/posts/<%= post._id %>"><%= post.title %></a></h3>
            <pre><%- post.content %></pre>
            <div>
                <span class="tag"><%= post.created_at %></span>
                <span class="tag right">
                    <span>浏览(<%= post.pv || 0 %>)</span>
                    <span>留言(<%= post.commentsCount || 0 %>)</span>
                </span>
            </div>
            <% if (user && post.author._id && user._id.toString() ===
post.author._id.toString()) { %>
                <div class="ui inline dropdown">
                    <div class="text"></div>
                    <i class="dropdown icon"></i>
                    <div class="menu">
                        <div class="item"><a href="/posts/<%= post._id %>/edit">编辑</a></
div>
                        <div class="item"><a href="/posts/<%= post._id %>/remove">删除</
a></div>
                    </div>
                </div>
            <% } %>
        </div>
        </div>
    </div>
</div>

```

上例中, `<pre><%- post.content %></pre>`是为了显示html格式的内容;

需要注意的是, tag类名在style.css中有定义:

```

.post-content .tag {
    font-size: 13px;
    margin-right: 5px;
    color: #999;
}

```

```
.post-content .tag.right {  
    float: right;  
    margin-right: 0;  
}
```

补充:

1.<pre>标签的主要作用是: 在pre元素中的文本会保留空格和换行符;

参考:

<https://www.cnblogs.com/wengxuesong/p/5541924.html>

(4)修改 routes/posts.js;

**routes/posts.js;**

```
router.get('/', function (req, res, next) {  
    const author = req.query.author  
  
    PostModel.getPosts(author)  
        .then(function (posts) {  
            res.render('posts', {  
                posts: posts  
            })  
        })  
        .catch(next)  
})
```

上例说明, 访问路径中是否存在author查询字段决定了页面显示所有用户的文章还是显示某一个指定用户的所有文章(在非登录情况下也可以显示用户的文, 但是不存在对文章编辑和删除功能的下拉菜单);

在登录状态下访问: <http://localhost:3000/posts>, 页面显示如下:

**myblog**  
my blog project

- song9**  
六六六六六六六  
2017-11-28 19:00 浏览(0) 留言(0)
- song3**  
123123123  
2017-11-28 18:59 浏览(0) 留言(0)
- first blog1**  
song first blog.  
2017-11-27 18:53 浏览(0) 留言(0)

点击头像后, 页面显示如下:

① localhost:3000/posts?author=5a1d41aec7f6caf9be199d0b

**myblog**  
my blog project

**song9 | 男**  
123456一二三五六  
song9  
六六六六六六六  
2017-11-28 19:00 浏览(0) 留言(0)

编辑  
删除

(5)为文章详情页新建 views/post.ejs;

**views/post.ejs;**

```
<%- include('header') %>
<%- include('components/post-content') %>
<%- include('footer') %>
```

(6)修改routes/posts.js中文章详情页部分内容;

```
// GET /posts/:postId 单独一篇的文章页
router.get('/:postId', function (req, res, next) {
  const postId = req.params.postId

  Promise.all([
    PostModel.getPostById(postId), // 获取文章信息
    PostModel.incPv(postId)// pv 加 1
  ])
  .then(function (result) {
    const post = result[0]
    if (!post) {
      throw new Error('该文章不存在')
    }

    res.render('post', {
      post: post
    })
  })
  .catch(next)
})
```

上例中，在then方法回调函数中抛出的error会被catch方法中的next函数作为参数接收；

点击文章标题后，页面显示如下(注意浏览器的地址栏):

The screenshot shows a browser window with the following details:

- Address Bar:** Shows the URL `① localhost:3000/posts/5a1bee971e9a7d1066f974d0`.
- Toolbar:** Standard browser toolbar with icons for search, refresh, and navigation.
- Bookmark Bar:** Shows several bookmarks including "Managed Bookmarks", "百度一下，你就知道", "System Dashboard", "Buy sports, concert...", "Your Profile", "Foobunny API docu...", "Backbone.js API中文...", "Underscore.js 中文...", and "Other Bookmark".
- Header:** The page title is "myblog" and the subtitle is "my blog project".
- Content Area:** Displays a blog post with the following details:
  - Thumbnail:** A small profile picture of a person.
  - Title:** **first blog1**
  - Content:** **song first blog.**
  - Date:** 2017-11-27 18:53
  - Actions:** "浏览(0)" and "留言(0)"
- Right Sidebar:** Contains buttons for "登录" (Login) and "注册" (Register).



19. 编辑与删除文章;

(1) 修改 models/posts.js(添加3个方法);

### **models/posts.js;**

```
// 通过文章 id 获取一篇原生文章 (编辑文章)
getRawPostById: function getRawPostById (postId) {
  return Post
    .findOne({ _id: postId })
    .populate({ path: 'author', model: 'User' })
    .exec()
},
// 通过文章 id 更新一篇文章
updatePostById: function updatePostById (postId, data) {
  return Post.update({ _id: postId }, { $set: data }).exec()
},
// 通过文章 id 删除一篇文章
delPostById: function delPostById (postId) {
  return Post.remove({ _id: postId }).exec()
}
```

(2) 新建编辑文章页 views/edit.ejs;

### **views/edit.ejs;**

```
<%- include('header') %>
```

```

<div class="ui grid">
  <div class="four wide column">
    <a class="avatar"
      href="/posts?author=<%= user._id %>"
      data-title="<%= user.name %> | <%= ({m: '男', f: '女', x: '保密'})[user.gender]
%>">
      data-content="<%= user.bio %>">
      
    </a>
  </div>

  <div class="eight wide column">
    <form class="ui form segment" method="post" action="/posts/<%= post._id
%>/edit">
      <div class="field required">
        <label>标题</label>
        <input type="text" name="title" value="<%= post.title %>">
      </div>
      <div class="field required">
        <label>内容</label>
        <textarea name="content" rows="15"><%= post.content %></textarea>
      </div>
      <input type="submit" class="ui button" value="发布">
    </form>
  </div>
</div>

<%- include('footer') %>

```

(3)修改 routes/posts.js;

### **routes/posts.js;**

```

// GET /posts/:postId/edit 更新文章页
router.get('/:postId/edit', checkLogin, function (req, res, next) {
  const postId = req.params.postId
  const author = req.session.user._id

  PostModel.getRawPostById(postId)
    .then(function (post) {
      if (!post) {

```

```
        throw new Error('该文章不存在')
    }
    if (author.toString() !== post.author._id.toString()) {
        throw new Error('权限不足')
    }
    res.render('edit', {
        post: post
    })
})
.catch(next)
})

// POST /posts/:postId/edit 更新一篇文章
router.post('/:postId/edit', checkLogin, function (req, res, next) {
    const postId = req.params.postId
    const author = req.session.user._id
    const title = req.fields.title
    const content = req.fields.content

    // 校验参数
    try {
        if (!title.length) {
            throw new Error('请填写标题')
        }
        if (!content.length) {
            throw new Error('请填写内容')
        }
    } catch (e) {
        req.flash('error', e.message)
        return res.redirect('back')
    }

    PostModel.getRawPostById(postId)
        .then(function (post) {
            if (!post) {
                throw new Error('文章不存在')
            }
            if (post.author._id.toString() !== author.toString()) {
                throw new Error('没有权限')
            }
            PostModel.updatePostById(postId, { title: title, content: content })
                .then(function () {
                    req.flash('success', '编辑文章成功')
                })
        })
})
```

```
// 编辑成功后跳转到上一页
res.redirect(`/posts/${postId}`)
})
.catch(next)
})
.catch(next)
})

// GET /posts/:postId/remove 删除一篇文章
router.get('/:postId/remove', checkLogin, function (req, res, next) {
const postId = req.params.postId
const author = req.session.user._id

PostModel.getRawPostById(postId)
.then(function (post) {
if (!post) {
throw new Error('文章不存在')
}
if (post.author._id.toString() !== author.toString()) {
throw new Error('没有权限')
}
PostModel.delPostById(postId)
.then(function () {
req.flash('success', '删除成功')
// 删除成功后跳转到主页
res.redirect('/posts')
})
.catch(next)
})
.catch(next)
})
```

① localhost:3000/posts/5a1e62ff660fd31af70c76a5/edit

Managed Bookmarks 百度一下，你就知道 System Dashboard -... Buy sports, concert... Your Profile Foobunny API docu... Backbone.js API中文... Underscore.js 中文... » Other Bookmarks

my blog project

**标题\***

**内容\***

```
<span style='color:red;'>edited&&saved</span>
<div>done</div>
```

发布

① localhost:3000/posts/5a1e62ff660fd31af70c76a5

Managed Bookmarks 百度一下，你就知道 System Dashboard -... Buy sports, concert... Your Profile Foobunny API docu... Backbone.js API中文... Underscore.js 中文... » Other Bookmarks

my blog project

## myblog

my blog project

**123test (edited)**

edited&saved

done

2017-11-29 15:34

浏览(4) 留言(0) ▾

① localhost:3000/posts?author=5a1d4179c7f6caf9be199d09

JS Bin - Collaborative JavaScript Debugging

Managed Bookmarks 百度一下，你就知道 System Dashboard -... Buy sports, concert... Your Profile Foobunny API docu... Backbone.js API中文... Underscore.js 中文... » Other Bookmarks

my blog project

## myblog

my blog project

**123test (edited)**

edited&saved

done

2017-11-29 15:34

浏览(5) 留言(0) ▾

**song3**

123123123

2017-11-28 18:59

浏览(1) 留言(0) ▾

20.留言;

(1)添加浏览模型, 修改 lib/mongo.js(需要留言的作者 id、留言内容和关联的文章 id 这几个字段);

**lib/mongo.js;**

```
exports.Comment = mongolass.model('Comment', {
  author: { type: Mongolass.Types.ObjectId },
  content: { type: 'string' },
  postId: { type: Mongolass.Types.ObjectId }
})
exports.Comment.index({ postId: 1, _id: 1 }).exec() // 通过文章 id 获取该文章下所有
留言, 按留言创建时间升序
```

(2)显示留言, 新建 views/components/comments.ejs;

**views/components/comments.ejs;**

```
<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">
    <div class="ui segment">
      <div class="ui minimal comments">
        <h3 class="ui dividing header">留言</h3>

        <% comments.forEach(function (comment) { %>
          <div class="comment">
            <span class="avatar">
              
            </span>
            <div class="content">
              <a class="author" href="/posts?author=<%= comment.author._id
%>"><%= comment.author.name %></a>
              <div class="metadata">
                <span class="date"><%= comment.created_at %></span>
              </div>
              <div class="text"><%- comment.content %></div>

              <% if (user && comment.author._id && user._id.toString() ===
comment.author._id.toString()) { %>
```

```

<div class="actions">
  <a class="reply" href="/comments/<%= comment._id %>/remove">删除
</a>
</div>
<% } %>
</div>
</div>
<% }) %>

<% if (user) { %>
<form class="ui reply form" method="post" action="/comments">
  <input name="postId" value="<%= post._id %>" hidden>
  <div class="field">
    <textarea name="content"></textarea>
  </div>
  <input type="submit" class="ui icon button" value="留言" />
</form>
<% } %>

</div>
</div>
</div>
</div>

```

上例中在提交留言表单时带上了文章 id (postId) , 通过 hidden 隐藏;  
很显然, 目前只有留言者本人可以删除自己的留言, 其实可以改进为文章作者可以删除自己文章下的任何一条留言;

(3)在文章页引入留言的模板片段, 修改 views/post.ejs;

#### **views/post.ejs:**

```

<%- include('header') %>

<%- include('components/post-content') %>
<%- include('components/comments') %>

<%- include('footer') %>

```

(4)新建 models/comments.js, 存放留言相关的数据库操作;

## **models/comments.js;**

```
const marked = require('marked')
const Comment = require('../lib/mongo').Comment

// 将 comment 的 content 从 markdown 转换成 html
Comment.plugin('contentToHtml', {
  afterFind: function (comments) {
    return comments.map(function (comment) {
      comment.content = marked(comment.content)
      return comment
    })
  }
})

module.exports = {
  // 创建一个留言
  create: function create (comment) {
    return Comment.create(comment).exec()
  },

  // 通过留言 id 获取一个留言
  getCommentById: function getCommentById (commentId) {
    return Comment.findOne({ _id: commentId }).exec()
  },

  // 通过留言 id 删除一个留言
  delCommentById: function delCommentById (commentId) {
    return Comment.remove({ _id: commentId }).exec()
  },

  // 通过文章 id 删除该文章下所有留言
  delCommentsByPostId: function delCommentsByPostId (postId) {
    return Comment.remove({ postId: postId }).exec()
  },

  // 通过文章 id 获取该文章下所有留言，按留言创建时间升序
  getComments: function getComments (postId) {
    return Comment
      .find({ postId: postId })
      .populate({ path: 'author', model: 'User' })
      .sort({ _id: 1 })
      .addCreatedAt()
  }
}
```

```
.contentToHtml()
.exec()
},

// 通过文章 id 获取该文章下留言数
getCommentsCount: function getCommentsCount (postId) {
  return Comment.count({ postId: postId }).exec()
}

}
```

需要:

- 1.留言也支持 markdown;
- 2.删除一篇文章成功后也要删除该文章下所有的评论, 上例子中的 delCommentsByPostId 就是用来做这件事的;

(5)修改 models/posts.js, 添加以下代码;

### **models/posts.js;**

```
const CommentModel = require('./comments')

// 给 post 添加留言数 commentsCount
Post.plugin('addCommentsCount', {
  afterFind: function (posts) {
    return Promise.all(posts.map(function (post) {
      return CommentModel.getCommentsCount(post._id).then(function (commentsCount) {
        post.commentsCount = commentsCount
        return post
      })
    }))
  },
  afterFindOne: function (post) {
    if (post) {
      return CommentModel.getCommentsCount(post._id).then(function (count) {
        post.commentsCount = count
        return post
      })
    }
    return post
  }
})
```

上例中，在 PostModel 上注册了 addCommentsCount 用来给每篇文章添加留言数 commentsCount;  
需要注意的是，addCommentsCount插件的afterFind方法返回的不是一个数组，而是一个Promise对象，所以很显然Mongolass接受插件最终返回Promise对象；

(6)修改getPostById, getPosts, 添加addCommentsCount, 修改delPostById方法;

### **models/posts.js;**

```
.....  
.addCreatedAt()  
.addCommentsCount()  
.contentToHtml()  
.....  
  
// 通过文章 id 删除一篇文章  
delPostById: function delPostById (postId) {  
    return Post.remove({_id: postId })  
    .exec()  
    .then(function (res) {  
        // 文章删除后，再删除该文章下的所有留言  
        if (res.result.ok && res.result.n > 0) {  
            return CommentModel.delCommentsByPostId(postId)  
        }  
    })  
}  
}
```

(7)修改 routes/posts.js;

### **routes/posts.js;**

```
const CommentModel = require('../models/comments')  
.....  
// GET /posts/:postId 单独一篇的文章页  
router.get('/:postId', function (req, res, next) {  
    const postId = req.params.postId  
  
    Promise.all([  
        PostModel.getPostById(postId), // 获取文章信息  
        CommentModel.getComments(postId), // 获取该文章所有留言
```

```

PostModel.incPv(postId)// pv 加 1
])
.then(function (result) {
  const post = result[0]
  const comments = result[1]
  if (!post) {
    throw new Error('该文章不存在')
  }

  res.render('post', {
    post: post,
    comments: comments
  })
})
.catch(next)
})

```

现在访问文章页，可以看到留言的部分：

The screenshot shows a user profile page on the left and a post detail page on the right. The post detail page displays a post by '123test (edited)' with the status 'edited&saved'. It includes a timestamp '2017-11-29 15:34' and a link to '浏览(8) 留言(0)'. Below the post, there is a large input field for comments and a green '留言' (Comment) button.

个人主页

发表文章

登出

留言

(8)发表与删除留言, 新建routes/comments.js;

routes/comments.js;

```
const express = require('express')
```

```
const router = express.Router()

const checkLogin = require('../middlewares/check').checkLogin
const CommentModel = require('../models/comments')

// POST /comments 创建一条留言
router.post('/', checkLogin, function (req, res, next) {
  const author = req.session.user._id
  const postId = req.fields.postId
  const content = req.fields.content

  // 校验参数
  try {
    if (!content.length) {
      throw new Error('请填写留言内容')
    }
  } catch (e) {
    req.flash('error', e.message)
    return res.redirect('back')
  }

  const comment = {
    author: author,
    postId: postId,
    content: content
  }

  CommentModel.create(comment)
    .then(function () {
      req.flash('success', '留言成功')
      // 留言成功后跳转到上一页
      res.redirect('back')
    })
    .catch(next)
  })

// GET /comments/:commentId/remove 删除一条留言
router.get('/:commentId/remove', checkLogin, function (req, res, next) {
  const commentId = req.params.commentId
  const author = req.session.user._id

  CommentModel.getCommentById(commentId)
    .then(function (comment) {
```

```
if (!comment) {
  throw new Error('留言不存在')
}
if (comment.author.toString() !== author.toString()) {
  throw new Error('没有权限删除留言')
}
CommentModel.delCommentById(commentId)
  .then(function () {
    req.flash('success', '删除成功')
    // 删除成功后跳转到上一页
    res.redirect('back')
  })
  .catch(next)
}).catch(next)
})

module.exports = router
```

(9)修改routes/index.js, 注册comments路由;

```
routes/index.js;;
.....
app.use('/comments', require('./comments'))
.....
```

对留言部分操作的截图:



## 123test (edited)

edited&saved

done

2017-11-29 15:34

浏览(10) 留言(1) ▾



个人主页

发表文章

登出

### 留言



song3 2017-12-01 17:39  
blue?

删除

留言



## 123test (edited)

edited&saved

done

2017-11-29 15:34

浏览(11) 留言(2) ▾



### 留言



song3 2017-12-01 17:39  
blue?

删除



song3 2017-12-01 17:41  
ok, just for test

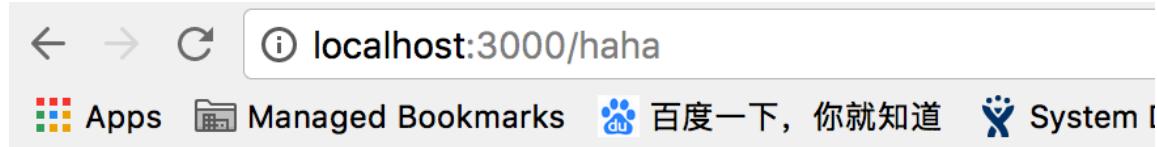
留言

The screenshot shows a blog application interface. At the top, there is a header with the title "myblog" and a subtitle "my blog project". On the right side of the header is a user menu with three vertical dots, "登录" (Login), and "注册" (Register). Below the header, there is a post card with a thumbnail image of a person, the title "123test (edited)", and the status "edited&saved". The post was "done" on "2017-11-29 15:34" and has "浏览(18) 留言(3)". Below the post card is a comment section titled "留言" (Comments) with three entries:

- song3 2017-12-01 17:39 blue?
- song9 2017-12-01 17:53 i am here
- song 2017-12-01 17:54 ok, i am here too. dude

21.404页面;

当访问一个不存在的地址, 如: <http://localhost:3000/haha> 页面会显示如下:



Cannot GET /haha

(1)修改 routes/index.js, 在所有其它路由之后添加:

```
// 404 page
app.use(function (req, res) {
  if (!res.headersSent) {
    res.status(404).render('404')
  }
})
```

(2)新建 views/404.ejs;

### **views/404.ejs;**

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title><%= blog.title %></title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <div class="errorPage">This is a 404 myblog page!</div>
  </body>
</html>
```

css/style.css;

```
.....
/* ----- 404 page ----- */
```

```
div.errorPage {
  position: fixed;
  top: 41%;
  left: 33%;
  font-size: 45px;
}
```

上例中的res.headersSent作用如下:

#### **res.headersSent**

Boolean property that indicates if the app sent HTTP headers for the response.

```
app.get('/', function (req, res) {
  console.log(res.headersSent); // false
  res.send('OK');
  console.log(res.headersSent); // true
})
```

404页面显示如下:



## 22. 错误页面;

之前提到过, express 有一个内置的错误处理逻辑, 如果程序中出错会直接将错误栈返回并显示到页面上;

如访问: localhost:3000/posts/xxx/edit 将会直接在页面中显示错误栈, 如下:

Error: 该文章不存在

```
at /Users/jiusong/myblog/routes/posts.js:95:15
at <anonymous>
at process._tickCallback (internal/process/next_tick.js:188:7)
```

(1) 在根目录index.js中添加错误处理中间件;

```
//错误处理中间件;
app.use(function (err, req, res, next) {
  console.error(err)
  req.flash('error', err.message)
  res.redirect('/posts')
})
```

之前在routes/...下的许多路由处理函数中使用了在promise的then方法中直接抛出错误, 然后用.catch(next)来接收的方式来处理错误, 其实也就是相当于将then方法中抛出的错误对象做为第一个参数传入到next函数中, 之后的错误处理中间件将会处理这个错误对象;

上例中的中间件就是用来处理这些错误对象的错误处理中间件;

重新访问: localhost:3000/posts/xxx/edit, 页面显示如下(此时url已经跳转改变了):



node.js终端显示如下:

```
Error: 该文章不存在
    at /Users/jiusong/myblog/routes/posts.js:95:15
    at <anonymous>
    at process._tickCallback (internal/process/next_tick.js:188:7)
```

23.日志;

日志分为正常请求的日志和错误请求的日志, 这里会实现将这两种日志都打印到终端并写入文件;

这里使用 `winston` 和 `express-winston` 记录日志;

(1)新建 logs 目录存放日志文件, 修改根目录 index.js;

```
const winston = require('winston')
const expressWinston = require('express-winston')
.....
// 正常请求的日志
app.use(expressWinston.logger({
  transports: [
    new (winston.transports.Console)({
      json: true,
      colorize: true
    }),
    new winston.transports.File({
      filename: 'logs/success.log'
    })
})
```

```
]
}))  
  
// 路由  
routes(app)  
  
// 错误请求的日志  
app.use(expressWinston.errorLogger({  
    transports: [  
        new winston.transports.Console({  
            json: true,  
            colorize: true  
        }),  
        new winston.transports.File({  
            filename: 'logs/error.log'  
        })  
    ]  
}))  
.....
```

刷新页面观察终端输出及 logs 下的文件; 可以发现: winston 会生成两个.log文件, 将正常请求的日志打印到终端并写入了 logs/success.log, 将错误请求的日志打印到终端并写入了 logs/error.log;

当访问<http://localhost:3000/haha>时;

在node.js终端输出为:

```
{
    "res": {
        "statusCode": 404
    },
    "req": {
        "url": "/haha",
        "headers": {
            "host": "localhost:3000",
            "connection": "keep-alive",
            "cache-control": "max-age=0",
            "upgrade-insecure-requests": "1",
            "user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36",
            "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
            "upgrade-insecure-requests": "1"
        }
    }
}
```

```

    "accept-encoding": "gzip, deflate, br",
    "accept-language": "en-US,en;q=0.9,es;q=0.8",
    "cookie":
"xdVisitorId=1321NNA1zvCZ-0EbmNjE_v5ZDJYOOZgJhvEpZwEXBMm2w2C6D;
_mibhv=anon-1489036515072-2740807910_5167;
JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u;
AMCV_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7
C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA
AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW
p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE;
_ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014;
myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW.
2dcxviy9DVFTnzxs8huvcdYRfO%2BVFWrkl1cfi80glzo"
},
"method": "GET",
"httpVersion": "1.1",
"originalUrl": "/haha",
"query": {}
},
"responseTime": 7,
"level": "info",
"message": "HTTP GET /haha"
}

```

success.log中记录的内容:

```
{"res":{"statusCode":404,"req":{"url":"/haha","headers":{"host":"localhost:
3000","connection":"keep-alive","cache-control":"max-age=0","upgrade-insecure-
requests":"1","user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/
537.36","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8","accept-encoding":"gzip, deflate, br","accept-
language":"en-
US,en;q=0.9,es;q=0.8","cookie":"xdVisitorId=1321NNA1zvCZ-0EbmNjE_v5ZDJYOO
SZgJhvEpZwEXBMm2w2C6D; _mibhv=anon-1489036515072-2740807910_5167;
JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u;
AMCV_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7
C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA
AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW
p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE;
_ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014;
myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW.
2dcxviy9DVFTnzxs8huvcdYRfO%2BVFWrkl1cfi80glzo"
}}
```

```
2dcxvi9DVFtnzxs8huvcDYRfO%2BVFWrkl1cfi80glzo"},"method":"GET","httpVersion":1.1,"originalUrl":"/haha","query":{},"responseTime":7,"level":"info","message":"HTTP GET /haha","timestamp":"2017-12-04T08:38:30.678Z"}
```

当访问localhost:3000/posts/xxx/edit时；

在node.js终端输出为：

```
{
  "date": "Mon Dec 04 2017 16:41:44 GMT+0800 (CST)",
  "process": {
    "pid": 71854,
    "uid": 110273178,
    "gid": 110273178,
    "cwd": "/Users/jiusong/myblog",
    "execPath": "/usr/local/bin/node",
    "version": "v9.2.0",
    "argv": [
      "/usr/local/bin/node",
      "/Users/jiusong/myblog/index"
    ],
    "memoryUsage": {
      "rss": 51228672,
      "heapTotal": 27684864,
      "heapUsed": 24560696,
      "external": 35702291
    }
  },
  "os": {
    "loadavg": [
      2.966796875,
      2.83056640625,
      2.62255859375
    ],
    "uptime": 1202208
  },
  "trace": [
    {
      "column": 15,
      "file": "/Users/jiusong/myblog/routes/posts.js",
      "function": null,
      "line": 95,
    }
  ]
}
```

```
"method": null,
"native": false
},
{
  "column": null,
  "file": null,
  "function": null,
  "line": null,
  "method": null,
  "native": false
},
{
  "column": 7,
  "file": "internal/process/next_tick.js",
  "function": "process._tickCallback",
  "line": 188,
  "method": "_tickCallback",
  "native": false
}
],
"stack": [
  "Error: 该文章不存在",
  "  at /Users/jiusong/myblog/routes/posts.js:95:15",
  "  at <anonymous>",
  "  at process._tickCallback (internal/process/next_tick.js:188:7)"
],
"req": {
  "url": "/posts/xxx/edit",
  "headers": {
    "host": "localhost:3000",
    "connection": "keep-alive",
    "user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36",
    "upgrade-insecure-requests": "1",
    "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
    "accept-encoding": "gzip, deflate, br",
    "accept-language": "en-US,en;q=0.9,es;q=0.8",
    "cookie": "xdVisitorId=1321NNA1zvCZ-0EbmNjE_v5ZDJYOOSZgJhjvEpZwEXBMm2w2C6D;_mibhv=anon-1489036515072-2740807910_5167;JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u;AMCV_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7"
}
```

C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA  
AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW  
p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE;  
\_ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014;  
myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW.  
2dcxviy9DVFtnzxs8huvcdYRfO%2BVFWrkI1cfi80glzo"  
},  
"method": "GET",  
"httpVersion": "1.1",  
"originalUrl": "/posts/xxx/edit",  
"query": {}  
,  
"level": "error",  
"message": "middlewareError"  
}  
Error: 该文章不存在  
at /Users/jiusong/myblog/routes/posts.js:95:15  
at <anonymous>  
at process.\_tickCallback (internal/process/next\_tick.js:188:7)  
{  
"res": {  
"statusCode": 302  
,  
"req": {  
"url": "/posts/xxx/edit",  
"headers": {  
"host": "localhost:3000",  
"connection": "keep-alive",  
"user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36",  
"upgrade-insecure-requests": "1",  
"accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8",  
"accept-encoding": "gzip, deflate, br",  
"accept-language": "en-US,en;q=0.9,es;q=0.8",  
"cookie":  
"xdVisitorId=1321NNA1zvCZ-0EbmNjE\_v5ZDJYOOSZgJhjvEpZwEXBMm2w2C6D;  
\_mibhv=anon-1489036515072-2740807910\_5167;  
JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u;  
AMCV\_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA  
AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW  
p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE;

```
_ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014;
myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW.
2dcxviy9DVFnzxs8huvCDYRfO%2BVFWrkI1cfi80glzo"
},
"method": "GET",
"httpVersion": "1.1",
"originalUrl": "/posts/xxx/edit",
"query": {}
},
"responseTime": 19,
"level": "info",
"message": "HTTP GET /posts/xxx/edit"
}
{
"res": {
"statusCode": 200
},
"req": {
"url": "/posts",
"headers": {
"host": "localhost:3000",
"connection": "keep-alive",
"user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36",
"upgrade-insecure-requests": "1",
"accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
"accept-encoding": "gzip, deflate, br",
"accept-language": "en-US,en;q=0.9,es;q=0.8",
"cookie": "xdVisitorId=1321NNA1zvCZ-0EbmNjE_v5ZDJYOOSZgJhvEpZwEXBMm2w2C6D;
_mibhv=anon-1489036515072-2740807910_5167;
JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u;
AMCV_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA
AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW
p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE;
_ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014;
myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW.
2dcxviy9DVFnzxs8huvCDYRfO%2BVFWrkI1cfi80glzo",
"if-none-match": "W/\\"210c-llNeYcFCNkRI5m7T8RaLfhjxqXU\\\""
},
"method": "GET",
```

```
        "httpVersion": "1.1",
        "originalUrl": "/posts",
        "query": {}
    },
    "responseTime": 21,
    "level": "info",
    "message": "HTTP GET /posts"
}
```

success.log中记录的内容:

```
{"res":{"statusCode":302,"req":{"url":"/posts/xxx/edit","headers":
{"host":"localhost:3000","connection":"keep-alive","user-agent":"Mozilla/5.0
(Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/62.0.3202.94 Safari/537.36","upgrade-insecure-
requests":"1","accept":"text/html,application/xhtml+xml,application/
xml;q=0.9,image/webp,image/apng,*/*;q=0.8","accept-encoding":"gzip, deflate,
br","accept-language":"en-
US,en;q=0.9,es;q=0.8","cookie":"xdVisitorId=1321NNA1zvCZ-0EbmNjE_v5ZDJYOO
SZgJhvEpZwEXBMm2w2C6D; _mibhv=anon-1489036515072-2740807910_5167;
JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u;
AMCV_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7
C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA
AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW
p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE;
_ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014;
myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW.
2dcxviy9DVFtnzxs8huvcdYRfO%2BVFWrkI1cfi80glzo"},"method":"GET","httpVersi
on":"1.1","originalUrl":"/posts/xxx/edit","query":{},"responseTime":
19,"level":"info","message":"HTTP GET /posts/xxx/
edit","timestamp":"2017-12-04T08:41:44.163Z"}
{"res":{"statusCode":200,"req":{"url":"/posts","headers":{"host":"localhost:
3000","connection":"keep-alive","user-agent":"Mozilla/5.0 (Macintosh; Intel Mac
OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94
Safari/537.36","upgrade-insecure-requests":"1","accept":"text/html,application/
xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8","accept-
encoding":"gzip, deflate, br","accept-language":"en-
US,en;q=0.9,es;q=0.8","cookie":"xdVisitorId=1321NNA1zvCZ-0EbmNjE_v5ZDJYOO
SZgJhvEpZwEXBMm2w2C6D; _mibhv=anon-1489036515072-2740807910_5167;
JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u;
AMCV_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7
C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA
```

AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW  
p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE;  
\_ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014;  
myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW.  
2dcxviy9DVFtnzxs8huvcdYRfO%2BVFWrkI1cfi80glzo","if-none-match":"W/\"210c-  
IINeYcFCNkRI5m7T8RaLfhjqXU\""}, "method": "GET", "httpVersion": "1.1", "originalUrl":  
"/posts", "query": {}, "responseTime": 21, "level": "info", "message": "HTTP GET /  
posts", "timestamp": "2017-12-04T08:41:44.208Z"}

error.log中记录的内容:

```
{"date": "Mon Dec 04 2017 16:41:44 GMT+0800 (CST)", "process": {"pid": 71854, "uid": 110273178, "gid": 110273178, "cwd": "/Users/jiusong/myblog", "execPath": "/usr/local/bin/node", "version": "v9.2.0", "argv": ["/usr/local/bin/node", "/Users/jiusong/myblog/index"], "memoryUsage": {"rss": 51228672, "heapTotal": 27684864, "heapUsed": 24560696, "external": 35702291}}, "os": {"loadavg": [2.966796875, 2.83056640625, 2.62255859375], "uptime": 1202208}, "trace": [{"column": 15, "file": "/Users/jiusong/myblog/routes/posts.js", "function": null, "line": 95, "method": null, "native": false}, {"column": null, "file": null, "function": null, "line": null, "method": null, "native": false}, {"column": 7, "file": "internal/process/next_tick.js", "function": "process._tickCallback", "line": 188, "method": "_tickCallback", "native": false}], "stack": ["Error: 该文章不存在", " at /Users/jiusong/myblog/routes/posts.js:95:15", " at <anonymous>", " at process._tickCallback (internal/process/next_tick.js:188:7)"], "req": {"url": "/posts/xxx/edit", "headers": {"host": "localhost:3000", "connection": "keep-alive", "user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36", "upgrade-insecure-requests": "1", "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8", "accept-encoding": "gzip, deflate, br", "accept-language": "en-US,en;q=0.9,es;q=0.8"}, "cookie": "xdVisitorId=1321NNA1zvCZ-0EbmNjE_v5ZDJYOO SZgJhjvEpZwEXBMm2w2C6D; _mihv=anon-1489036515072-2740807910_5167; JSESSIONID=18ssnrck1r9yr1tfg8plf28m7u; AMCV_1AEC46735278551A0A490D45%40AdobeOrg=1304406280%7CMCIDTS%7C17500%7CMCMID%7C42414153539315392372556410827646960196%7CMCA AMLH-1512356266%7C9%7CMCAAMB-1512527877%7CRKhpRz8krg2tLO6pguXW p5olkAcUniQYPHaMWWgdJ3xzPWQmdj0y%7CMCAID%7CNONE; _ga=GA1.1.1390294497.1512009754; fsr.a=1512040577014; myblog=s%3AdMac5ZSvcSyRy-O6O7Fx0okAHYOmptqW. 2dcxviy9DVFtnzxs8huvcdYRfO%2BVFWrkI1cfi80glzo"}, "method": "GET", "httpVersion": "1.1"}]
```

```
on": "1.1", "originalUrl": "/posts/xxx/edit", "query": {}}, "level": "error", "message": "middlewareError", "timestamp": "2017-12-04T08:41:44.158Z"}
```

通过上面的log记录可以发现，只要有请求进入了根目录的index.js，也就是说无论是否会返回404页面或者在应用内部发生报错，都将被视为一次正常的访问，只要这次访问最终以：res.redirect或者res.render这样的形式响应给客户端，那么就会将这次访问的信息显示在node.js终端并记录在success.log中，需要注意的是，上例中之所以终端在302响应记录之前显示错误信息的记录是因为应用的流程是先使用next(error)让错误处理中间件处理错误对象，然后在错误处理中间件中再执行了302响应；当应用中存在next(error)这样需要错误处理中间件来处理的错误时，会在node.js终端以及error.log文件中记录错误信息；所以，记录正常请求日志的中间件要放到 routes(app) 之前，记录错误请求日志的中间件要放到 routes(app) 之后；

## 24. gitignore文件的指定；

如果想把项目托管到 git 服务器上（如：[GitHub](#)），不想把线上配置、本地调试的 logs 以及 node\_modules 添加到 git 的版本控制中，这个时候就需要 .gitignore 文件了，git 会读取 .gitignore 并忽略这些文件(.gitignore是隐藏文件);

补充：

显示所有隐藏文件/文件夹，在控制台输入：

```
显示: defaults write com.apple.finder AppleShowAllFiles -bool true
```

```
隐藏: defaults write com.apple.finder AppleShowAllFiles -bool false
```

(1)在 myblog 下新建 .gitignore 文件，添加如下配置：

```
.gitignore;
```

```
config/*  
!config/default.*  
npm-debug.log  
node_modules  
coverage
```

上例中，只有 config/default.js 会加入 git 的版本控制，而 config 目录下的其他配置文件则会被忽略，因为把线上配置加入到 git 是一个不安全的行为，通常需要在本地或者线上环境手动创建 config/production.js，然后添加一些线上的配置（如：mongodb 配置）即可覆盖相应的 default 配置；

注意：后面讲到部署到 Heroku 时，因为无法登录到 Heroku 主机，所以可以把以下两行删掉，将 config/production.js 也加入 git 中；

```
config/*  
!config/default.*
```

(2)在 public/img 目录下创建 .gitignore：

```
# Ignore everything in this directory  
*  
# Except this file  
.gitignore
```

上例的设置会让 git 忽略 public/img 目录下所有上传的头像图片文件，而不忽略 public/img 这个目录结构；

(3)在 logs 目录下创建 .gitignore 忽略日志文件，而不忽略 logs 这个目录结构；

```
# Ignore everything in this directory  
*  
# Except this file  
.gitignore
```

25. 功能测试；

mocha 和 supertest 是常用的测试组合，通常用来测试 restful 的 api 接口，这里用来测试博客应用；

(1) 在 myblog 下新建 test 文件夹存放测试文件；

(2) 安装所需模块；

```
npm i mocha supertest --save-dev
```

如果安装模块时发生报错： rollbackFailedOptional .....

```
npm ERR! code ECONNRESET  
npm ERR! errno ECONNRESET
```

**npm ERR! network request to https://registry-npmjs.stubcorp.com/nexus/repository/npm/mocha failed, reason: socket hang up**

.....

解决办法是执行:

npm config set registry http://registry.npmjs.org/

参考:

<https://stackoverflow.com/questions/36421657/npm-express-module-not-installing>

(3)修改 package.json;

将:

```
"scripts": {  
  "test": "echo \\\"Error: no test specified\\\" && exit 1"  
}
```

修改为:

```
"scripts": {  
  "test": "mocha test"  
}
```

(4)修改根目录index.js;

将:

```
// 监听端口，启动程序  
app.listen(config.port, function () {  
  console.log(` ${pkg.name} listening on port ${config.port}`)  
})
```

修改为:

```
if (module.parent) {  
  // 被 require, 则导出 app  
  module.exports = app  
} else {  
  // 监听端口，启动程序
```

```
app.listen(config.port, function () {
  console.log(`\$ {pkg.name} listening on port \$ {config.port}`)
})
}
```

上例可以实现：直接启动 index.js 则会监听端口启动程序，如果 index.js 被 require 了，则导出 app，通常用于测试；

(5)以注册为例进行测试；

<1>将一张图片放到 test 目录下，用于测试上传头像；

<2>新建 test/signup.js，添加测试代码：

#### **test/signup.js:**

```
'use strict'
```

```
const path = require('path')
const assert = require('assert')
const request = require('supertest')
const app = require('../index')
const User = require('../lib/mongo').User

const testName1 = 'songtest1'
const testName2 = 'songtest2'
const testName3 = 'songtest3'
describe('signup', function () {
  describe('POST /signup', function () {
    const agent = request.agent(app)// persist cookie when redirect
    beforeEach(function (done) {
      // 创建一个用户
      User.create({
        name: testName1,
        password: '123456',
        avatar: '',
        gender: 'x',
        bio: ''
      })
      .exec()
      .then(function () {
        done()
      })
    })
  })
})
```

```
        })
        .catch(done)
    })

afterEach(function (done) {
    // 删除测试用户
    User.remove({ name: { $in: [testName1, testName2, testName3] } })
        .exec()
        .then(function () {
            done()
        })
        .catch(done)
})

// 用户名错误的情况
it('wrong name', function (done) {
    agent
        .post('/signup')
        .type('form')
        .attach('avatar', path.join(__dirname, 'avatar.png'))
        .field({ name: "" })
        .redirects()
        .end(function (err, res) {
            if (err) return done(err)
            assert(res.text.match(/名字请限制在 1-10 个字符/))
            done()
        })
})

// 性别错误的情况
it('wrong gender', function (done) {
    agent
        .post('/signup')
        .type('form')
        .attach('avatar', path.join(__dirname, 'avatar.png'))
        .field({ name: testName2, gender: 'a' })
        .redirects()
        .end(function (err, res) {
            if (err) return done(err)
            assert(res.text.match(/性别只能是 m、f 或 x/))
            done()
        })
})
```

```

// 其余的参数测试自行补充
// 用户名被占用的情况
it('duplicate name', function (done) {
  agent
    .post('/signup')
    .type('form')
    .attach('avatar', path.join(__dirname, 'avatar.png'))
    .field({ name: testName1, gender: 'm', bio: 'noder', password: '123456',
    repassword: '123456' })
    .redirects()
    .end(function (err, res) {
      if (err) return done(err)
      assert(res.text.match(/用户名已被占用/))
      done()
    })
  })
}

// 注册成功的情况
it('success', function (done) {
  agent
    .post('/signup')
    .type('form')
    .attach('avatar', path.join(__dirname, 'avatar.png'))
    .field({ name: testName2, gender: 'm', bio: 'noder', password: '123456',
    repassword: '123456' })
    .redirects()
    .end(function (err, res) {
      if (err) return done(err)
      assert(res.text.match(/注册成功/))
      done()
    })
  })
})
})
})

```

上例中, res.text返回的内容就是最终生成的将要在页面上显示的完整页面内容, 比如在测试'用户名已被占用'的测试实例中, res.text返回的内容为:

```

<html>
  <head>
    <meta charset="utf-8">
    <title>myblog</title>

```

```
<link rel="stylesheet" href="//cdn.bootcss.com/semantic-ui/2.1.8/
semantic.min.css">
<link rel="stylesheet" href="/css/style.css">
<script src="//cdn.bootcss.com/jquery/1.11.3/jquery.min.js"></script>
<script src="//cdn.bootcss.com/semantic-ui/2.1.8/semantic.min.js"></script>
</head>
<body>
<div class="nav">
<div class="ui grid">
<div class="four wide column"></div>

<div class="eight wide column">
<a href="/posts"><h1>myblog</h1></a>
<p>my blog project</p>
</div>
</div>
</div>

<div class="nav-setting">
<div class="ui buttons">
<div class="ui floating dropdown button">
<i class="icon bars"></i>
<div class="menu">

<a class="item" href="/signin">登录</a>
<a class="item" href="/signup">注册</a>

</div>
</div>
</div>
</div>

<div class="ui grid">
<div class="four wide column"></div>
<div class="eight wide column">

<div class="ui error message">
<p>用户名已被占用</p>
</div>

</div>
</div>
```

```
<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">
    <form class="ui form segment" method="post" enctype="multipart/form-data">
      <div class="field required">
        <label>用户名</label>
        <input placeholder="用户名" type="text" name="name">
      </div>
      <div class="field required">
        <label>密码</label>
        <input placeholder="密码" type="password" name="password">
      </div>
      <div class="field required">
        <label>重复密码</label>
        <input placeholder="重复密码" type="password" name="repassword">
      </div>
      <div class="field required">
        <label>性别</label>
        <select class="ui compact dropdown" name="gender">
          <option value="">性别</option>
          <option value="m">男</option>
          <option value="f">女</option>
          <option value="x">保密</option>
        </select>
      </div>
      <div class="field required">
        <label>头像</label>
        <input type="file" name="avatar">
      </div>
      <div class="field required">
        <label>个人简介</label>
        <textarea name="bio" rows="5"></textarea>
      </div>
      <input type="submit" class="ui button fluid" value="注册">
    </form>
  </div>
</div>

<script type="text/javascript">
  // 使通知框延时自动从页面删除
  $(document).ready( function () {
    // 延时清除掉成功、失败提示信息
    if($('.ui.success.message').length > 0) {
```

```

        $('.ui.success.message').fadeOut(3000)
    } else if($('.ui.error.message').length > 0) {
        $('.ui.error.message').fadeOut(3000)
    }

    // 点击按钮弹出下拉框
    $('.ui.dropdown').dropdown();

    // 鼠标悬浮在头像上，弹出气泡提示框
    $('.post-content .avatar-link').popup({
        inline: true,
        position: 'bottom right',
        lastResort: 'bottom right'
    });
}
</script>
</body>
</html>

```

补充:

1.Mocha和Supertest;

Mocha;

是非常流行JavaScript测试框架之一，在浏览器和Node环境都可以使用；

should;

should是一个很简单的、贴近自然语言的断言库；当然，Mocha是适配所有的断言库的（比如assert, expect之类的）；

Supertest;

只使用Mocha和should就几乎可以满足所有JavaScript函数的单元测试；但是对于Node应用而言，不仅仅是函数的集合，比如一个web应用的测试；这时候就需要配合一个http代理，完成Http请求和路由的测试；

Supertest是一个HTTP代理服务引擎，可以模拟一切HTTP请求行为；Supertest可以搭配任意的应用框架，从而进行应用的单元测试；

Mocha API;

- describe (moduleName, testDetails)

由上述代码可看出，describe是可以嵌套的，比如上述代码嵌套的两个describe就可以理解成测试人员希望测试'signup'模块下的'POST /signup'子模块；  
moduleName 是可以随便取的，关键是要让人读明白就好；

- `it (info, function)`  
具体的测试语句会放在it的回调函数里，一般来说info字符串会写期望的正确输出的简要一句话文字说明。当该it block内的test failed的时候控制台就会把详细信息打印出来。一般是从最外层的describe的module\_name开始输出，如果最后输出info表示该期望的info内容没有被满足；一个it对应一个实际的test case, it中的Mocha可以接受任何形式的断言的返回结果将做为test case是否通过的判断条件；
- `assert()`  
上例中使用的是nodejs里的assert.js的一种断言形式，比较常用的还有chai模块；

例子：

```
// 这是一个简单的加法函数
//add.js
function add(a, b){
    return a+b;
}
module.exports = add;

//add.test.js
var add = require('./add');
var should = require('should');
describe('test add', function() {
    it('1 + 1 should be equal to 2', function(done){
        (add(1,1) === 2).should.be.true;
    });
});
```

上例是同步函数的测试，但是在Node环境中，绝大部分的业务逻辑都是异步的，所以测试结果的回调是JavaScript测试框架需要解决的首要问题；Mocha提供了一个回调函数done，在业务代码执行完毕之后调用done()结束测试用例，不然的话测试用例就会出现timeout的情况导致测试用例失败（由于异步调用会直接返回，所以会读取不到任何断言的结果）；Mocha默认的超时时间是2000毫秒，如果由于测试用例的执行时间比较长需要延长超时时间，可以在命令行添加 -t 参数，比如：mocha -t 3000 add.test.js

cookie持久化；

在很多业务测试中，需要用户先登录才有权限执行操作；这个时候作为HTTP请求模拟，必须要可以保存一些Cookie数据，也就是Cookie的持久化；

在 supertest 中，可以通过 `var request = supertest.agent(app)` 获取一个 agent 对象，这个对象的 API 跟直接在 supertest 上调用各种方法是一样的；这个request在被多次调用 get 和 post 之后，可以一路把 cookie 都保存下来；

还有一种方法是在发起请求时，调用 `.set('Cookie', 'a cookie string')` 这样的方式，如：

```

var supertest = require('supertest'),
express = require('express');
var app = express();
var request = supertest(app);

describe('GET /users', function(){
  it('respond with json', function(done){
    request
      .get('/user')
      .set('Accept', 'application/json')
      .expect(200)
      .end(function(err, res){
        should.not.exist(err);
        res.text.should.containEql('success');
        done();
      });
  });
});

```

参考:

- (1)<http://www.cnblogs.com/wade-xu/p/4665250.html>
- (2)<http://www.imooc.com/article/2631>

补充:

- 1.上例中.expect() 方法是一个断言, 表示在执行之后期望的状态码是200(OK); 如果期望接收到的数据为html页面, 那么可以设置.expect('Content-Type', 'text/html;charset=utf-8');
- 2..end() 的作用是执行一个request请求, 然后就可以在回调函数里根据业务逻辑的返回数据做断言分析了;

<3>如果编辑器报语法错误(如: describe 未定义等等), 修改 .eslintrc.json 如下:

```
{
  "extends": "standard",
  "globals": {
    "describe": true,
    "beforeEach": true,
    "afterEach": true,
    "it": true
  }
}
```

```
}
```

上例这样的设置后, eslint 会忽略 globals 中变量未定义的警告;

## 26. 测试覆盖率;

一般情况下测试的理想状态是覆盖所有的情况（包括各种出错的情况及正确时的情况），但是如果只是通过人工来计算需要写哪些测试是不够的，总会有疏漏之处，最简单的办法就是可以直观的看出测试是否覆盖了所有的代码，这就是测试覆盖率，即被测试覆盖到的代码行数占总代码行数的比例；

需要注意的是，即使测试覆盖率达到 100% 也不能说明测试覆盖了所有的情况，只能说明基本覆盖了所有的情况；

**istanbul** 是一个常用的生成测试覆盖率的库，它会将测试的结果报告生成 html 页面，并放到项目根目录的 coverage 目录下；

关于istanbul, 可以参考:

<http://www.ruanyifeng.com/blog/2015/06/istanbul.html>

(1)首先安装 istanbul;

```
npm i istanbul --save-dev
```

(2)配置 istanbul, 修改package.json;

**package.json:**

```
"scripts": {  
  "test": "mocha test"  
}
```

修改为：

```
"scripts": {  
  "test": "istanbul cover _mocha"  
}
```

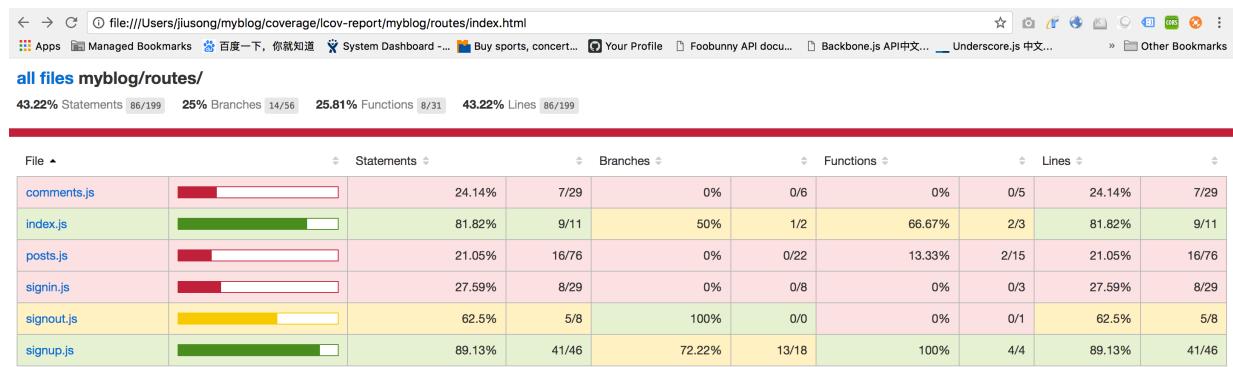
上例中的命令就是将istanbul 和 mocha 结合使用；

如果运行正常的话，终端会输出，如：

```
=====
Writing coverage object [/Users/jiusong/myblog/coverage/coverage.json]
Writing coverage reports at [/Users/jiusong/myblog/coverage]
=====

===== Coverage summary =====
Statements : 50.81% ( 157/309 )
Branches   : 22.97% ( 17/74 )
Functions   : 30.77% ( 20/65 )
Lines      : 50.81% ( 157/309 )
=====
```

打开 myblog/coverage/lcov-report/index.html，显示如下：



点击查看某个代码文件具体的覆盖率，显示如下：

## all files / myblog/routes/ signup.js

88.37% Statements 38/43 72.22% Branches 13/18 100% Functions 4/4 88.37% Lines 38/43

```
1 1x var path = require('path');
2 1x var sha1 = require('sha1');
3 1x var express = require('express');
4 1x var router = express.Router();
5
6 1x var UserModel = require('../models/users');
7 1x var checkNotLogin = require('../middlewares/check').checkNotLogin;
8
9 // GET /signup 注册页
10 1x router.get('/', checkNotLogin, function(req, res, next) {
11 3x   res.render('signup');
12 });
13
14 // POST /signup 用户注册
15 1x router.post('/', checkNotLogin, function(req, res, next) {
16 4x   var name = req.fields.name;
17 4x   var gender = req.fields.gender;
18 4x   var bio = req.fields.bio;
19 4x   var avatar = req.files.avatar.path.split(path.sep).pop();
20 4x   var password = req.fields.password;
21 4x   var repassword = req.fields.repassword;
22
23 // 校验参数
24 4x try {
25 4x   if (!(name.length >= 1 && name.length <= 10)) {
26 1x     throw new Error('名字请限制在 1-10 个字符');
27   }
28 3x   if ([ 'm', 'f', 'x' ].indexOf(gender) === -1) {
29 1x     throw new Error('性别只能是 m、f 或 x');
30   }
31 2x   I if (!(bio.length >= 1 && bio.length <= 30)) {
32     throw new Error('个人简介请限制在 1-30 个字符');
33   }
34 2x   I if (!req.files.avatar.name) {
35     throw new Error('缺少头像');
36   }
37 2x   I if (password < 6) {
38     throw new Error('密码至少 6 个字符');
39   }
40 2x   I if (password !== repassword) {
41     throw new Error('两次输入密码不一致');
42   }
43 } catch (e) {
44 2x   req.flash('error', e.message);
45 2x   return res.redirect('/signup');
46 }
47
48 // 明文密码加密
49 2x password = sha1(password);
50
51 // 待写入数据库的用户信息
52 2x var user = {
53   name: name,
54   password: password,
55   gender: gender,
```

上例中显示红色的行表示测试没有覆盖到, 因为之前只写了 name 和 gender 的测试;

需要注意的是, 使用上例的方式可能会发生在终端没有istanbul coverage相关输出只有mocha测试信息的情况下(coverage文件夹下也没有生成任何内容), 解决办法:

(1)使用babel-istanbul 代替 istanbul;

```
npm i babel-core --save-dev
npm i babel-cli --save-dev
npm i babel-istanbul --save-dev
```

(2)修改package.json;

```
"scripts": {
  "test:coverage": "babel-node ./node_modules/.bin/babel-istanbul cover ./node_modules/.bin/_mocha ./test/signup.js"
},
.....
```

(3)在终端输入:

```
npm run test:coverage
```

如果上面的方法还是不能输出istanbul coverage相关的输出, 那么可以使用第三种方法:

修改package.json;

```
"scripts": {
  "test": "babel-node ./node_modules/istanbul/lib/cli cover node_modules/mocha/bin/_mocha -- --bail --recursive './test/signup.js'"
},
.....
```

补充:

1.直接在终端使用mocha或者Istanbul等命令会报类似错误: -sh: istanbul: command not found

这是由于:

./node\_modules/.bin is not in \$PATH. There are some solutions in [How to use](#)

## package installed locally in node\_modules?

解决办法是：

it might be easier to just wrap your babel-cli commands in npm scripts. This works because npm run adds the output of npm bin (node\_modules/.bin) to the PATH provided to scripts;

这就是为什么使用npm script来执行其他命令行时不会出现\$PATH环境变量未设置错误的原因；

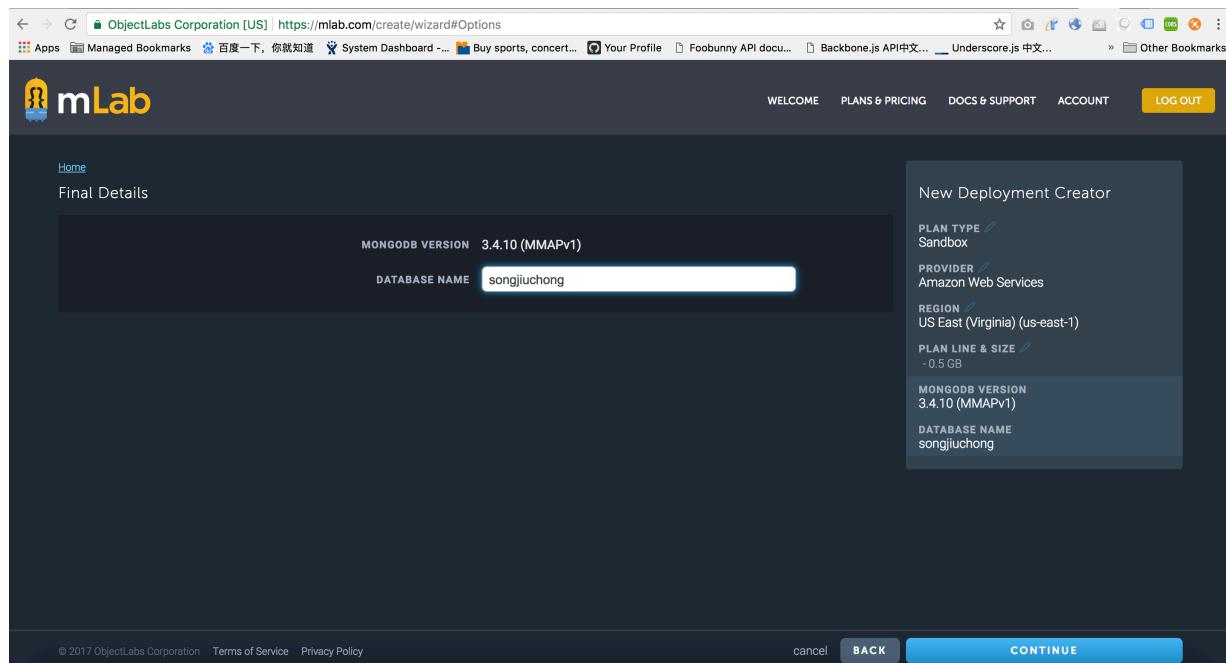
27.部署；

(1)申请MLab；

MLab (前身是 MongoLab) 是一个 mongodb 云数据库提供商, 我们可以选择 500MB 空间的免费套餐用来测试；

<1>注册；

<2>点击右上角的 Create New 创建一个数据库 (如: songjiuchong) ；



The screenshot shows the mLab Order Confirmation page. At the top, it displays the mLab logo and navigation links for Welcome, Plans & Pricing, Docs & Support, Account, and Log Out. Below the header, the page title is "Order Confirmation". The main content area shows a summary of the selected plan: PLAN TYPE is Sandbox, CLOUD PROVIDER is Amazon Web Services, REGION is US East (Virginia) (us-east-1), and PLAN LINE & SIZE is Sandbox. It also indicates that the plan is FREE. Other details shown include STORAGE (0.5 GB), MONGODB VERSION (3.4.10 (MMAPv1)), and DATABASE NAME (songjiuchong). At the bottom of the summary box, it says "Total Price" and "FREE". Below the summary, there are links for Terms of Service and Privacy Policy, and buttons for cancel, BACK, and SUBMIT ORDER.

<3>每个数据库至少需要一个 user, 点击 Users 下的 Add database user 创建一个用户;

注意: 不要选中 Make read-only, 因为会有写数据库的操作;

The screenshot shows the mLab Database Users creation dialog. The dialog is titled "Add new database user". It contains fields for "Database username\*" (set to "jiusong"), "Database password\*", "Confirm password\*", and a checkbox for "Make read-only" which is unchecked. To the right of the dialog, the mLab interface shows the "Database: songjiuchong" page with a message indicating the user "jiusong" has been added. There are also instructions for connecting via mongo shell or driver, and a note about sandbox redundancy. Navigation tabs at the bottom include Collections and Users (which is active).

从上面的截图可以看出, MLab分配的 mongodb url 是:

`mongodb://<dbuser>:<dbpassword>@ds019786.mlab.com:19786/songjiuchong`

在终端使用mongo shell:

`mongo ds019786.mlab.com:19786/songjiuchong -u jiusong -p Therapists22391425!`

成功连接到MLab数据库后就可以在终端操作数据库了;

<4>新建 config/production.js, 添加如下代码:

**config/production.js;**

```
module.exports = {
  mongodb:'mongodb://jiusong:Therapists22391425!@ds019786.mlab.com:19786/
  songjiuchong'
}
```

上例中使用了此数据库user的真实id:password来替换<dbuser>:<dbpassword>;

<5>停止程序, 以 production 配置重新启动程序:

`NODE_ENV=production supervisor index`

需要注意的是, Windows 用户需要全局安装 `cross-env`, 终端输入:

`npm i cross-env -g`

```
cross-env NODE_ENV=production supervisor index
```

(2)pm2;

当博客要部署到线上服务器时, 不能单纯的靠 node index 或者 supervisor index 来启动了, 因为当断掉了 SSH 连接后(关闭对本地或远程服务器执行命令的终端连接会话后)服务进程就终止了, 这时我们就需要像 pm2 或者 forever 这样的进程管理工具了; pm2 是 Node.js 下的生产环境进程管理工具, 就是我们常说的进程守护工具, 可以用来在生产环境中进行自动重启、日志记录、错误预警等等;

pm2和forever是启动Nodejs服务常用到的两个工具; 使用这两个指令可以使node服务在后台运行 (类似于linux的nohup), 另外它们可以在服务因异常或其他原因被杀掉后进行自动重启; 由于Node的单线程特征, 自动重启能很大程度上的提高它的健壮性;

pm2和forever的使用可以参考:

<http://www.jianshu.com/p/225b9284cfb8>

补充:

1.关于SSH连接;

SSH 英文全称是secure shell, 它建立在应用层和传输层基础上, 专为远程登录会话和其他网络服务提供安全性的协议;

ssh会对远程登陆时的认证信息和远程执行的命令进行加密传输;

参考:

[http://www.ruanyifeng.com/blog/2011/12/ssh\\_remote\\_login.html](http://www.ruanyifeng.com/blog/2011/12/ssh_remote_login.html)

<1>全局安装 pm2;

```
npm i pm2 -g
```

补充:

1.pm2 常用命令:

pm2 start/stop: 启动/停止程序

pm2 reload/restart [id|name]: 重启程序

pm2 logs [id|name]: 查看日志

pm2 l/list: 列出程序列表

更多命令可以使用 pm2 -h 查看;

<2>修改 package.json, 添加 start 的命令;

```
"scripts": {  
  "test": "nyc --reporter=html --reporter=text mocha",  
  "start": "NODE_ENV=production pm2 start index.js --name 'myblog'"  
}
```

注意:

1.由于对mLab远程云数据库的连接问题, 这里暂时在package.json中使用:

```
"scripts": {  
  "test": "istanbul cover _mocha",  
  "start": "pm2 start index.js --name 'myblog'"  
},  
"_comment": {  
  "scripts": {  
    "test": "nyc --reporter=html --reporter=text mocha",  
    "start": "NODE_ENV=production pm2 start index.js --name 'myblog'"  
  }  
},
```

关于json文件中的注释, 可以参考Page Dev helper中: '170.在JSON文件中添加注释;'

<3>运行 npm start 通过 pm2 启动程序;

```
LM-SHC-16501358:myblog jiusong$ npm start  
> myblog@1.0.0 start /Users/jiusong/myblog  
> pm2 start index.js --name 'myblog'  
  
[PM2] Applying action restartProcessId on app [myblog](ids: 0)  
[PM2] [myblog](0) ✓  
[PM2] Process successfully started  
  
          App name   id   mode   pid      status  restart  uptime   cpu     mem        user      watching  
          myblog     0   fork    80185  online     0       0s      0%    12.6 MB    jiusong  
  
Use `pm2 show <id|name>` to get more details about an app  
LM-SHC-16501358:myblog jiusong$  
LM-SHC-16501358:myblog jiusong$  
LM-SHC-16501358:myblog jiusong$  
LM-SHC-16501358:myblog jiusong$ █
```

可以发现, 通过pm2启动应用之后, 终端已经可以做其它操作了, 也就是说应用已经在后

台运行了, 此时关闭终端也不会影响应用;

如果需要关闭应用, 可以通过:

```
pm2 stop index.js
```

```
LM-SHC-16501358:myblog jiusong$ pm2 stop index.js
[PM2] Applying action stopProcessId on app [index.js](ids: 0)
[PM2] [myblog](0) ✓

  App name   id   mode   pid   status   restart   uptime   cpu     mem     user     watching
  myblog      0   fork    0   stopped    0        0       0%     0 B     jiusong

  Use `pm2 show <id|name>` to get more details about an app
LM-SHC-16501358:myblog jiusong$
```

(3)将论坛应用部署到 Heroku 提供的云服务器;

Heroku 是一个支持多种编程语言的云服务平台, 并且提供免费的基础套餐供开发者测试使用;

在Heroku上部署web项目的流程, 可以参考:

<https://devcenter.heroku.com/articles/getting-started-with-nodejs>

<1>安装heroku CLI (<https://devcenter.heroku.com/articles/heroku-cli>);

```
brew install heroku/brew/heroku
```

此时可能会报错:

cannot be installed as a binary package and must be built from source.

To continue, you must install Xcode from the App Store,

or the CLT by running:

```
xcode-select --install
```

<2>安装Xcode command line developer tools;

```
$ xcode-select --install
```

在弹出框中选Install继续安装;

<3>检查是否成功安装;

```
$ xcode-select -p
```

终端显示:

/Library/Developer/CommandLineTools

<4>检查heroku是否安装成功;

```
$ heroku --version
```

终端显示:

heroku-cli/6.14.40-86ef14c (darwin-x64) node-v9.2.0

<5>注册后登陆, 然后下载安装 Heroku 的命令行工具包 toolbelt (如果之前在第一步已经安装了heroku CLI, 就可以跳过此步继续);

## Set up

In this step you will install the Heroku Command Line Interface (CLI), formerly known as the Heroku Toolbelt. You will use the CLI to manage and scale your applications, to provision add-ons, to view the logs of your application as it runs on Heroku, as well as to help run your application locally.



<6>来到Heroku 的 Dashboard(<https://dashboard.heroku.com/apps>), 点击右上角 New -> Create New App 创建一个应用, 创建成功后在本地终端运行下面的代码来验证登陆;

```
heroku login
```

Log in using the email address and password you used when creating your Heroku account:

```
$ heroku login  
Enter your Heroku credentials.  
Email: zeke@example.com  
Password:  
...
```

填写正确的 email 和 password 验证通过后, 本地会产生一个 SSH public key, 之后本地向heroku发送的请求(SSH通信)就会通过这个服务器的公钥加密发送;

```
Enter your Heroku credentials:  
Email: jiусong@ebay.com  
Password: *****  
Logged in as jiусong@ebay.com  
LM-SHC-16501358:myblog jiусong$ █
```

<7>删除.gitignore 中:

```
config/*  
!config/default/*
```

由于无法在 Heroku 主机中创建 production 配置文件, 所以这里需要将 production 配置也上传到 Heroku 提供的远程git库中;

<8>修改根目录index.js;

将 app.listen 修改为:

```
const port = process.env.PORT || config.port  
app.listen(port, function () {  
  console.log(`#${pkg.name} listening on port ${port}`)  
})
```

由于Heroku 会动态分配端口(通过shell环境变量 PORT=xxx 指定, 如: NODE\_ENV=xxx), 所以不能用配置文件指定端口;

<9>修改 package.json, 在 scripts 中添加'heroku'命令;

```
"heroku": "NODE_ENV=production node index"
```

<10>在根目录下新建 Procfile 文件, 添加如下内容:

```
web: npm run heroku
```

Procfile 文件会告诉 Heroku 使用什么命令启动一个 web 服务;

注意:

1.这个Procfile文件不能有类型后缀, 只需要Procfile这个文件名即可;

<11>Deploy the app to Heroku remote git repository;

(1. 在本地创建一个git库(这里创建一个文件夹mygit, 然后将本地的myblog项目文件夹复制到mygit中, myblog就是这个项目的本地git库);

(2. 将mygit/myblog中的内容同步到heroku为开发者指定的app创建的远程git库中(一旦用户将本地git库内容同步到heroku指定的远端库, heroku就会立刻将远端库中的代码部署到云服务器中并启动web应用);

在终端输入:

```
cd mygit/myblog  
git init  
heroku create
```

```
LM-SHC-16501358:myblog jiusong$ git init  
Initialized empty Git repository in /Users/jiusong/mygit/myblog/.git/
```

Create an app on Heroku, which prepares Heroku to receive your source code.

```
$ heroku create  
Creating sharp-rain-871... done, stack is cedar-14  
http://sharp-rain-871.herokuapp.com/ | https://git.heroku.com/sharp-rain-871.git  
Git remote heroku added
```

When you create an app, a git remote (called `heroku`) is also created and associated with your local git repository.

Heroku generates a random name (in this case `sharp-rain-871`) for your app, or you can pass a parameter to specify your own app name.

输入`heroku create`后, 终端显示如下:

```
LM-SHC-16501358:myblog jiusong$ heroku create  
Creating app... done, ⚡ frozen-crag-82138  
https://frozen-crag-82138.herokuapp.com/ | https://git.heroku.com/frozen-crag-82138.git
```

(3. 如果想要在heroku上创建一个自定义名称的app, 可以使用下面的方法(app名可能已经被占用):

参考:

<https://agilewarrior.wordpress.com/2014/05/16/how-to-create-a-named-app-in-heroku/>(重要)

`heroku apps:create songjiuchongblog`

```
LM-SHC-16501358:myblog jiusong$ heroku apps:create songjiuchongblog  
Creating ⚡ songjiuchongblog... done  
https://songjiuchongblog.herokuapp.com/ | https://git.heroku.com/songjiuchongblog.git
```

查看目前拥有的app列表可以使用命令:

`heroku apps`

```
LM-SHC-16501358:myblog jiusong$ heroku apps  
==== jiusong@ebay.com Apps  
frozen-crag-82138
```

查看目前绑定的远端库;

```
LM-SHC-16501358:myblog jiusong$ git remote -v  
heroku  https://git.heroku.com/frozen-crag-82138.git (fetch)  
heroku  https://git.heroku.com/frozen-crag-82138.git (push)
```

解绑之前随机命名的app对应远端库, 绑定后来自定义重命名的app对应远端库;

```
> git remote rm heroku  
> git remote add heroku https://git.heroku.com/songjiuchongblog.git  
> git remote -v
```

```
heroku  https://git.heroku.com/songjiuchongblog.git (fetch)  
heroku  https://git.heroku.com/songjiuchongblog.git (push)
```

(4. 将本地git库mygit/myblog中的内容同步到heroku指定的远端库:

```
LM-SHC-16501358:myblog jiusong$ git add -A  
git  
LM-SHC-16501358:myblog jiusong$ git commit -m 'init songjiuchongblog'
```

```
LM-SHC-16501358:myblog jiusong$ git push heroku master  
Counting objects: 605, done.
```

```
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Node.js app detected
remote:
remote: -----> Creating runtime environment
remote:
remote:     NPM_CONFIG_LOGLEVEL=error
remote:     NPM_CONFIG_PRODUCTION=true
remote:     NODE_VERBOSE=false
remote:     NODE_ENV=production
remote:     NODE_MODULES_CACHE=true
remote:
remote: -----> Installing binaries
remote:       engines.node (package.json): unspecified
remote:       engines.npm (package.json):   unspecified (use default)
remote:
remote:       Resolving node version 8.x...
remote:       Downloading and installing node 8.9.3...
remote:       Using default npm version: 5.5.1
remote:
remote: -----> Restoring cache
remote:       Skipping cache restore (not-found)
remote:
remote: -----> Building dependencies
remote:       Installing node modules (package.json + package-lock)
remote:
remote:         > semantic-ui@2.2.13 install /tmp/build_9393bb24d5a4e4a5a6fd808314a870d7/node_modules/semantic-ui
remote:           > gulp install
remote:
remote:             [02:20:01] Using gulpfile /tmp/build_9393bb24d5a4e4a5a6fd808314a870d7/node_modules/semantic-ui/gulpfile.js
remote:             [02:20:01] Starting 'install'...
remote:
Current version of Semantic UI already installed
remote:       added 2340 packages in 40.375s
remote:
remote: -----> Caching build
remote:       Clearing previous node cache
remote:       Saving 2 cacheDirectories (default):
remote:         - node_modules
remote:         - bower_components (nothing to cache)
```

```
remote: -----> Build succeeded!
remote: -----> Discovering process types
remote:       Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:       Done: 51.2M
remote: -----> Launching...
remote:       Released v3
remote:       https://songjiuchongblog.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/songjiuchongblog.git
 * [new branch]      master -> master
LM-SHC-16501358:myblog jiusong$
```

根据上面的截图可以发现, 在将本地的git库同步到heroku远程的git库时会将应用同时部署到heroku云服务器中;

<12> 访问部署在heroku云服务器上的项目;

The application is now deployed. Ensure that at least one instance of the app is running:

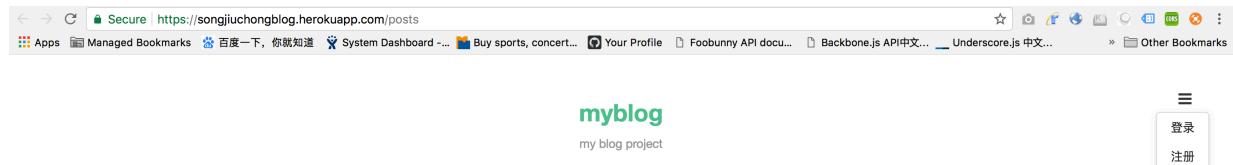
```
$ heroku ps:scale web=1
```

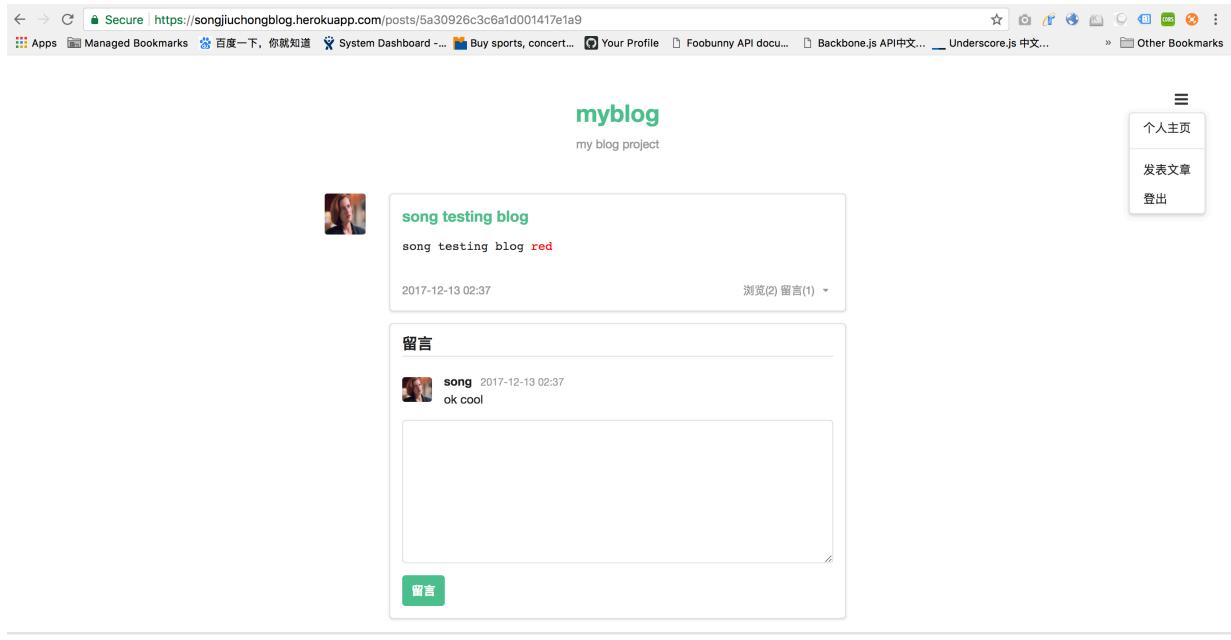
Now visit the app at the URL generated by its app name. As a handy shortcut, you can open the website as follows:

```
$ heroku open
```

```
LM-SHC-16501358:myblog jiusong$ heroku ps:scale web=1
Scaling dynos... done, now running web at 1:Free
```

执行heroku open之后自动打开了页面(当然也可以直接在浏览器中访问: <https://songjiuchongblog.herokuapp.com/posts>), 显示如下:





## <13>关于dyno;

Right now, your app is running on a single web **dyno**. Think of a dyno as a lightweight container that runs the command specified in the `Procfile`.

You can check how many dynos are running using the `ps` command:

```
$ heroku ps
==== web (Free): `node index.js`
web.1: up 2014/04/25 16:26:38 (~ 1s ago)
```

By default, your app is deployed on a free dyno. Free dynos will sleep after a half hour of inactivity (if they don't receive any traffic). This causes a delay of a few seconds for the first request upon waking. Subsequent requests will perform normally. Free dynos also consume from a monthly, account-level quota of **free dyno hours** - as long as the quota is not exhausted, all free apps can continue to run.

Scaling an application on Heroku is equivalent to changing the number of dynos that are running. Scale the number of web dynos to zero:

```
$ heroku ps:scale web=0
```

Access the app again by hitting refresh on the web tab, or [heroku open](#) to open it in a web tab. You will get an error message because you no longer have any web dynos available to serve requests.

Scale it up again:

```
$ heroku ps:scale web=1
```

补充:

A dyno is a Heroku metric defined as being a "single process of any type running on the Heroku platform".

dyno是Heroku的一个度量标准，定义为“在Heroku平台上运行的、任何类型的单一进程”；

It is offered in two versions, a free standard version, and a professional subscription version that costs \$0.06 per dyno hour.

该产品提供两个版本，一个是免费的标准版，一个是专业版订阅服务，目前价格为每dyno 小时0.06美分；

<14>查看heroku服务器日志;

View information about your running app using one of the [logging commands](#), `heroku logs --tail`:

```
$ heroku logs --tail
2011-03-10T10:22:30-08:00 heroku[web.1]: State changed from created to starting
2011-03-10T10:22:32-08:00 heroku[web.1]: Running process with command: `node index.js`
2011-03-10T10:22:33-08:00 heroku[web.1]: Listening on 18320
2011-03-10T10:22:34-08:00 heroku[web.1]: State changed from starting to up
```

Visit your application in the browser again, and you'll see another log message generated.

Press `Control+C` to stop streaming the logs.

如果不使用`--tail`参数，那么会接收并显示到执行命令为止服务器中记录的日志，而不是实时连续地接收日志；

这里所说的日志，其中大部分都是在heroku dyno终端上的输出内容；

```

^CLM-SHC-16501358:myblog jiusong$ heroku logs
2017-12-13T02:37:39.054440+00:00 app[web.1]:   "httpVersion": "1.1",
2017-12-13T02:37:39.054435+00:00 app[web.1]:   "x-forwarded-for": "216.113.160.67",
2017-12-13T02:37:39.054441+00:00 app[web.1]:   "originalUrl": "/posts/5a30926c3c6a1d001417e1a9/edit",
2017-12-13T02:37:39.054436+00:00 app[web.1]:   "x-forwarded-port": "443",
2017-12-13T02:37:39.054442+00:00 app[web.1]:   "query": {},
2017-12-13T02:37:39.054437+00:00 app[web.1]:   "via": "1.1 vegur",
2017-12-13T02:37:39.054436+00:00 app[web.1]:   "connect-time": "0",
2017-12-13T02:37:39.054438+00:00 app[web.1]:   "x-forwarded-proto": "https",
2017-12-13T02:37:39.054439+00:00 app[web.1]:   "x-request-start": "1513132659011",
2017-12-13T02:37:39.054442+00:00 app[web.1]:   "total-route-time": "0"
2017-12-13T02:37:39.054442+00:00 app[web.1]: },
2017-12-13T02:37:39.054443+00:00 app[web.1]:   "responseTime": 12,
2017-12-13T02:37:39.054443+00:00 app[web.1]:   "level": "info",
2017-12-13T02:37:39.054444+00:00 app[web.1]:   "message": "HTTP GET /posts/5a30926c3c6a1d001417e1a9/edit"
2017-12-13T02:37:39.054444+00:00 app[web.1]: }
2017-12-13T02:37:39.059015+00:00 heroku[router]: at=info method=GET path="/posts/5a30926c3c6a1d001417e1a9/edit" host=songjiuchongblog.herokuapp.com request_id=e4e4c31d2-4097-493b-8497-3251a74b9205 fwd="216.113.160.67" dyno=web.1 connect=0ms service=46ms status=200 bytes=2973 protocol=https
2017-12-13T02:37:43.371738+00:00 heroku[router]: at=info method=GET path="/css/style.css" host=songjiuchongblog.herokuapp.com request_id=09d9a055-2d41-420f-addf-67e72c5c98f0 fwd="216.113.160.67" dyno=web.1 connect=1ms service=6ms status=304 bytes=237 protocol=https
2017-12-13T02:37:43.629482+00:00 heroku[router]: at=info method=GET path="/img/upload_7bc5fd246b79984a9b7a9be34bb283f.png" host=songjiuchongblog.herokuapp.com request_id=8693c27b-4ec2-4718-ba8b-e4c45e81eb9e fwd="216.113.160.67" dyno=web.1 connect=0ms service=3ms status=304 bytes=238 protocol=https
2017-12-13T02:37:39.733515+00:00 heroku[router]: at=info method=GET path="/css/style.css" host=songjiuchongblog.herokuapp.com request_id=4a1861da-276f-44f7-a554-3e029cc0038a fwd="216.113.160.67" dyno=web.1 connect=0ms service=3ms status=304 bytes=237 protocol=https
2017-12-13T02:37:55.128263+00:00 app[web.1]: {
2017-12-13T02:37:55.128272+00:00 app[web.1]:   "res": {
2017-12-13T02:37:55.128273+00:00 app[web.1]:     "statusCode": 302
2017-12-13T02:37:55.128274+00:00 app[web.1]:   },
2017-12-13T02:37:55.128275+00:00 app[web.1]:   "url": "/comments",
2017-12-13T02:37:55.128275+00:00 app[web.1]:   "headers": {
2017-12-13T02:37:55.128274+00:00 app[web.1]:     "req": {
2017-12-13T02:37:55.128276+00:00 app[web.1]:       "host": "songjiuchongblog.herokuapp.com",
2017-12-13T02:37:55.128277+00:00 app[web.1]:       "connection": "close",
2017-12-13T02:37:55.128277+00:00 app[web.1]:       "cache-control": "max-age=0",
2017-12-13T02:37:55.128278+00:00 app[web.1]:       "origin": "https://songjiuchongblog.herokuapp.com",
2017-12-13T02:37:55.128279+00:00 app[web.1]:       "upgrade-insecure-requests": "1",
2017-12-13T02:37:55.128280+00:00 app[web.1]:       "content-type": "application/x-www-form-urlencoded",
2017-12-13T02:37:55.128281+00:00 app[web.1]:       "user-agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36",
2017-12-13T02:37:55.128282+00:00 app[web.1]:     "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",

```

<15>使用heroku 在本地启动app;

## Run the app locally

Now start your application locally using the `heroku local` command, which was installed as part of the Heroku CLI:

```
$ heroku local web
[OKAY] Loaded ENV .env File as KEY=VALUE Format
1:23:15 PM web.1 | Node app is running on port 5000
```

Just like Heroku, `heroku local` examines the `Procfile` to determine what to run.

Open <http://localhost:5000> with your web browser. You should see your app running locally.

To stop the app from running locally, in the CLI, press `Ctrl+C` to exit.

```

LM-SHC-16501358:myblog jiusong$ heroku local web
[WARN] No ENV file found
14:32:10 web.1    | > myblog@1.0.0 heroku /Users/jiusong/mygit/myblog
14:32:10 web.1    | > NODE_ENV=production node index
14:32:10 web.1    | myblog listening on port 5000

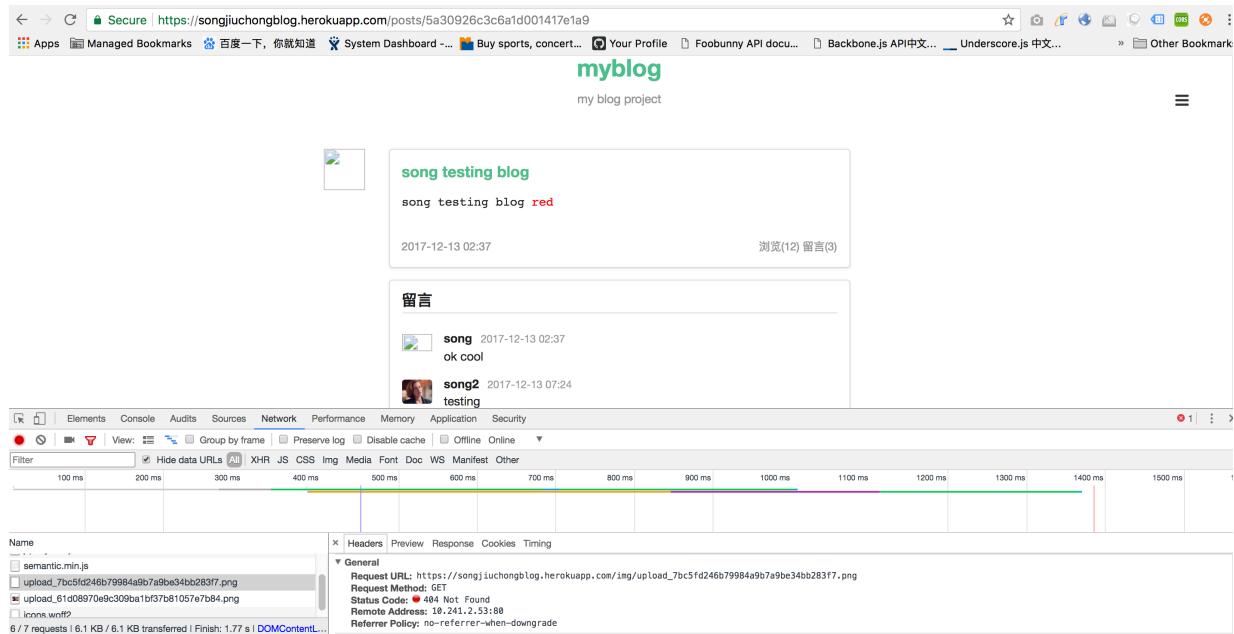
```

需要注意的是, 如果是从heroku远程git库pull的app, 那么由于.gitignore中设置了node\_modules, 所以需要在本地先使用npm install来安装所有依赖, 然后再启动应用;

<16>在heroku 云服务器中访问myblog应用出现的问题;

之前也提到过, 由于heroku云服务器会在应用闲置半小时的情况下(没有任何的traffic)自动关闭应用, 虽然dyno进程不会自动关闭, 但是由于当应用再次被访问时就会重启应用, 此时启动的应用是根据对应的heroku远程git库中的代码来重新部署的, 也就是说之前保存在应用目录下的内容已经丢失了(比如用户上传到img文件夹中的头像文件);不过由于其他信息都保存在mLAB云数据库中, 所以目前受到这个机制影响的只有img文件夹下的用户上传头像文件;

所以当用户在heroku云服务器上的myblog应用中注册并上传了头像文件, 一旦heroku关闭一次应用, 那么下次再访问应用页面时就会出现找不到img中相应图片的问题(接下来会介绍的两个收费云服务器由于机制不同, 所以不存在这个问题), 参考下面的截图:



补充:

1.使用heroku -h 命令可以查看heroku相关所有命令的用法;

例子:

heroku -h

heroku apps -h

```
LM-SHC-16501358:myblog jiusong$ heroku apps -h
Usage: heroku apps [flags]

list your apps

Flags:
  -A, --all           include apps in all teams
  -p, --personal      list apps in personal account when a default team is set
  -s, --space SPACE   filter by space
  -t, --team TEAM     team to use
  --json              output in json format

Example:

$ heroku apps
==== My Apps
example
example2

==== Collaborated Apps
theirapp other@owner.name

heroku apps commands: (get help with heroku help apps:COMMAND)
apps                      list your apps
apps:create [APP]          creates a new app
apps:destroy [APP]          permanently destroy an app
apps:errors                 view app errors
apps:favorites              list favorited apps
apps:favorites:add          favorites an app
apps:favorites:remove        unfavorites an app
apps:info [APP]              show detailed app information
apps:join                   add yourself to an organization app
apps:leave                  remove yourself from an organization app
apps:lock                   prevent organization members from joining an app
apps:open [PATH]             open the app in a web browser
apps:rename NEWNAME          rename an app
apps:stacks                 show the list of available stacks
apps:stacks:set STACK        set the stack of an app
apps:transfer RECIPIENT      transfer applications to another user, organization or team
apps:unlock                  unlock an app so any organization member can join
```

28.部署到其它收费云服务器:

- (1)部署到 UCloud;
- (2)部署到阿里云;

上面的这两种云服务都是收费的,但是与免费套餐下的heroku不同的是,它们提供一个可以让用户远程登录的主机,用户登陆后可以在本地操作这台主机,而heroku只会自动部署它指定的git库中的内容到某一个或多个dyno中(也就是运行Procfile中指定命令的进程);

所以要使用上面的这两种云服务必须:

- <1>注册并购买云服务主机;
- <2>本地登录远程主机;
- <3>按照类似本地创建项目时的步骤在远程主机中下载安装: git, nodejs, npm, pm2, MongoDB 等程序;
- <4>将项目源码从自己的git库中同步到远程主机中新建的git库,然后启动应用;
- <5>访问云服务分配的公网ip,测试应用是否已经部署成功;

具体参考:

<https://github.com/nswbmw/N-blog/blob/master/book/>

4.15%20%E9%83%A8%E7%BD%B2.md (4.15.3 部署到 UCloud, 4.15.4 部署到阿里云)