

Node.js 项目实战（零基础入门到精通）

NodeJs 零基础快速搭建一个 microblog 网站项目

作者：侯烽泰

日期：2017-12-05

笔名：画风

前言

初学 NodeJs 时曾在网上看过不少 NodeJs 的书籍以及项目实战，无奈 NodeJs 更新太快变化太大了，网上很多书籍以及项目教程所涉及的代码已不适用，要么就是结构混乱，让初学者苦不堪言（ps：自己动手过，遇到了坑，才会让自己成长）；本文旨在让初学者有个大概了解，能够快速入门上手，从零开始到搭建项目架构（MVC 模式）和数据库，并测试通过 CRUD 等数据库操作。此项目功能比较简单：只有登录、注册、退出、发表文章、修改文章、删除文章、上传图片这些功能。

注：此项目为入门级别，大神可以飘过，有不足之处请多多指教
请尊重别人的劳动成果，转载请注明出处，谢谢！

项目源码地址：<https://github.com/houfengtai/microblog.git>

参考于[使用 Express + MongoDB 搭建多人博客](#)

技术栈

NodeJs8.9.1 + Express4.15.5 + MongoDB3.4

项目运行

注意：由于使用到数据库，请先预装好 MongoDB,以及 Node 环境,项目链接数据库配置文件为 setting.js,修改为安装数据库时的电脑 IP

```
git clone https://github.com/houfengtai/microblog.git
```

```
cd microblog
npm install
node ./bin/www
```

另外

本项目非使用到前端框架，后期会把该项目写成纯 API 框架形式，前端会使用 VUE2.x 或 angular.js 等编写前端项目用来前后台对接，以达到前后端分离效果。

学前准备（安装环境）本教程所使用系统为 win10

1、安装 NodeJs

到官网 <https://nodejs.org/en/download> 下载最新版本，已安装的不是最新版的请按自己系统自行百度教程升级最新版本
查看是否安装成功：打开控制台输入

```
node -v
```

出现版本号即表示安装成功，最新版本是不需要配置环境变量的，安装时会自动配置好的

2、安装 Express（这是目前 nodeJs 唯一官方认证框架，类似于 Struts、spring 等，通常用于 web 后端开发）

```
npm install -g express-generator
npm install -g express
express -V
```

note: 其中 -g 代表全局安装，这样的话以后可以在任意目录下使用 express 命令创建项目 -V 是大写的，用来查看当前安装的版本号

3、创建项目（体验效果）

(1) 去到指定目录下

```
cd d:\\workspace
```

(2) 使用 express 命令创建 express 项目

```
express microblog
```

(3) 进入项目根目录，安装项目依赖包

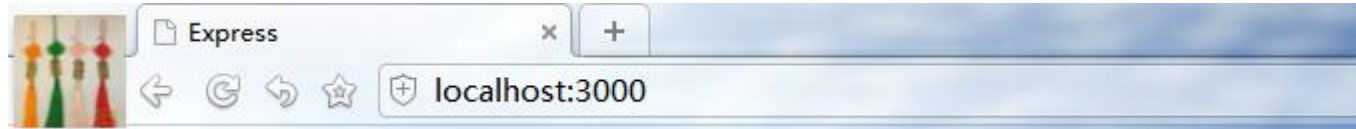
```
cd ./microblog
```

```
npm install
```

(4) 启动项目:启动方法有多种, 项目后期有额外拓展

```
npm start
```

(5) 浏览器查看项目效果: <http://127.0.0.1:3000> 然后你会看到:express 的欢迎信息。然后开始我们的 express 之旅吧!











Express

Welcome to Express

开始开发

刚刚已经创建了一个最基础化的 express 项目了, 接下来就开始着手开发了, 首先进入到项目所在目录查看整个项目结构。

 bin	2017/12/1 14:51	文件夹	
 node_modules	2017/12/1 14:53	文件夹	
 public	2017/12/1 14:51	文件夹	
 routes	2017/12/1 14:51	文件夹	
 views	2017/12/1 14:51	文件夹	
 app.js	2017/12/1 14:51	JetBrains WebSt...	2 KB
 package.json	2017/12/1 14:51	JSON 文件	1 KB
 package-lock.json	2017/12/1 14:53	JSON 文件	26 KB

开发前准备

开发工具

开发工具有：IntelliJ IDEA 2017.2.5 、JetBrains WebStorm 2017.2.5、eclipse 等等

本文使用 IntelliJ IDEA 2017.2.5,至于为什么选用这个，主要是因为 IDEA 很好兼容 NodeJs 插件,js 各种提示都很到位。WebStorm 看名字就知道跟 IDEA 是同一个公司的，兼容性一样好，eclipse 开发 NodeJs 稍微差一些,很多 js 提示不到位,使用 ES6 写的代码提示报错。

IDEA 下载官网：<https://www.jetbrains.com/idea/download>

安装：按照步骤自行安装。安装完成之后先别打开，IDEA 是需要依赖 JDK 的，所以先安装 JDK1.8。

JDK 下载地址：<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

选择对应系统的版本安装

配置环境变量:自行百度对应系统 JDK 环境变量配置

IDEA 是需要钱的，这个对于我们来说是没有一点技术含量的，百度一个秘钥就可以破解了。

数据库

数据库有：

关系型数据库：Oracle、DB2、SQL server、Mysql 等等

非关系型数据库：MongoDB、HBase、Redis 等等

本文使用 MongoDB, mongo 是面向文档的数据库，存储数据格式为 BSON, 与 NodeJs 操作的 JSON 最为衔接，效率也是最快的。

安装方法（有多种环境安装，自行选择一个吧）：

1、Docker 上安装（docker 是什么？这个不详说了，适用于高级人群，不适合初学者）：

查找 Docker Hub 上的 mongo 镜像

```
docker search mongo
```

拉取官方镜像（本文使用 mongo3.4 版本）

```
docker pull mongo:3.4
```

使用 mongo 镜像

```
docker run -p 27017:27017 -e $PWD/db:/data/db -d mongo:3.4
```

命令说明：

-p 27017:27017 :将容器的 27017 端口映射到主机的 27017 端口

-e \$PWD/db:/data/db :将主机中当前目录下的 db 挂载到容器的/data/db，作为 mongo 数据存储目录

查看容器启动情况

```
docker ps
```

使用 mongo 镜像执行 mongo 命令连接到刚启动的容器, 主机 IP 为 172.17.0.1（IP 设为本机 IP）

```
winpty docker exec -it mongoddb bash  
mongo --host 172.17.0.1
```

查看 mongoddb 是否能连接上

浏览器访问:<http://172.17.0.1:27017> 提示：

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

表示安装成功

2、Vmware 上 Linux CentOS 6.7 环境安装（推荐使用）：

Vm 虚拟机自行安装 Linux CentOS 6.7，本文就不详说了。（注：Docker 与 Vm 是有冲突的，只能启动其一）

Linux 超级管理员（root）登陆情况下,打开控制台（初学者使用桌面，高级点的使用 Xshell1 链接）

下载 mongodb

```
wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-amazon-3.4.6.tgz
```

解压文件

```
tar zxvf mongodb-linux-x86_64-amazon-3.4.6.tgz
```

把解压后文件移动到 /usr/local/mongodb 目录下

```
mv mongodb-linux-x86_64-amazon-3.4.6 /usr/local/mongodb
```

如果 mongodb 目录不存在则先创建

```
cd /usr/local/  
mkdir mongodb  
cd  
mv mongodb-linux-x86_64-amazon-3.4.6 /usr/local/mongodb
```

进入/usr/local/mongodb 目录

```
cd /usr/local/mongodb
```

创建 data 目录

```
mkdir data  
cd ./data
```

在 data 目录下创建 db 与 logs 文件夹

```
mkdir db  
mkdir logs
```

在 data 目录下创建 mongodb.conf 文件

```
vi mongodb.conf
```

添加以下内容

```
#数据目录
dbpath = /usr/local/mongodb/data/db

#日志目录
logpath = /usr/local/mongodb/data/logs/mongodb.log

#设置后台运行
fork = true

#日志输出方式
logappend = true

#开启认证
#auth = true
```

按左上角 Esc 键

再按 Shift + : 冒号键后输入 wq 保存退出（更多 Linux 命令自行百度学习）

更新配置文件

```
cd /usr/local/mongodb
./bin/mongod --config /usr/local/mongodb/data/mongodb.conf
```

启动 mongodb

```
./bin/mongo
```

修改防火墙开放端口

```
cd /etc/sysconfig
vi iptables
```

```
i
```

22 端口下添加一行 27017 端口 按左上角 Esc 键

再按 Shift + : 冒号键后输入 wq 保存退出（更多 Linux 命令自行百度学习）

查看 Linux 本机 IP

```
ifconfig
```

浏览器访问:<http://本机 IP:27017> 提示:

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

表示安装成功

拓展:

关闭 mongodb

```
cd
pkill mongod
```

3、当然还有在本机安装（不建议）：按不同系统自行百度吧！

正式开发

经过上面的努力，所有的环境都已安装就绪，下面我们就开始来正式写代码了，想想都有点小激动吧！

NodeJs API:<https://nodejs.org/dist/latest-v8.x/docs/api/>

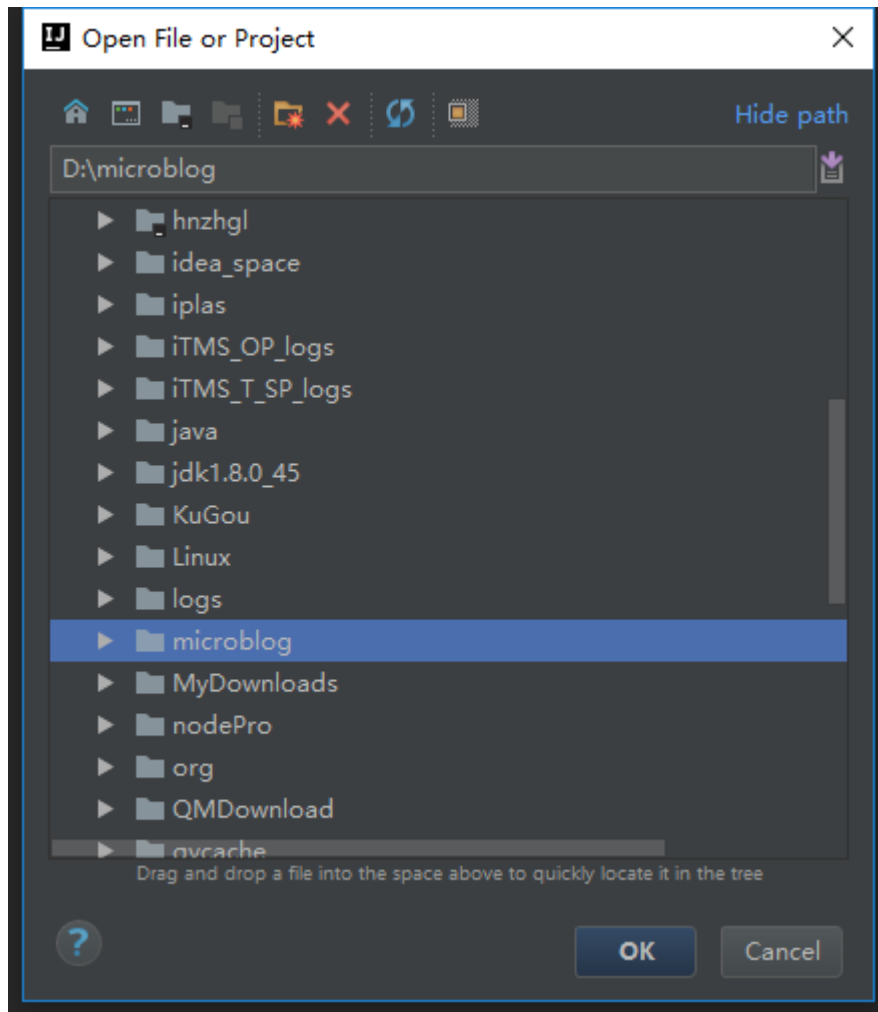
NodeJs API（中文）:<http://nodejs.cn/api/>

Express API:<http://www.nodeclass.com/api/express4.html>

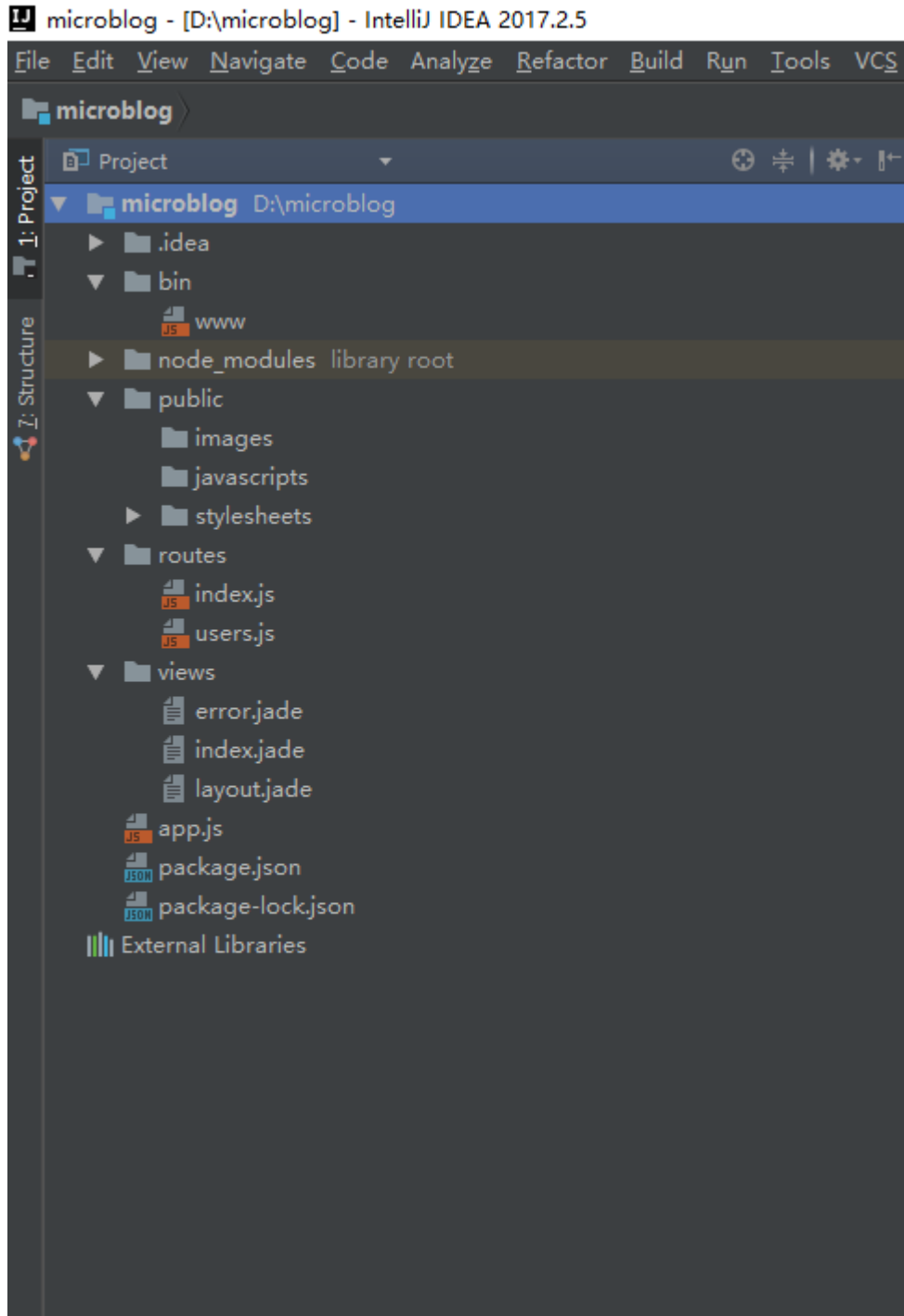
Express API（中文）:<http://www.expressjs.com.cn/4x/api.html>

打开我们的 IDEA 工具（需要了解更多的 IDEA 工具使用方法请自行百度）

点击左上角 File-->Open... -->选择刚才创建的 microblog 项目



看一下我们刚才创建的项目总体结构



app.js: 启动文件，或者说入口文件

package.json: 存储着工程的信息及模块依赖

当在 dependencies 中添加依赖的模块时，运行 npm install, npm 会检查当前目录下的 package.json, 并自动安装所有指定的模块 node_modules: 存放 package.json 中安装的模块，当你在 package.json 添加依赖的模块并安装后，存放在这个文件夹下

public: 存放 image、css、js 等文件

routes: 存放路由文件

views: 存放视图文件或者说模版文件

bin: 存放可执行文件

打开 app.js, 让我们看看里面究竟有什么:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var index = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', index);
app.use('/users', users);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
```

```
res.render('error');
});
```

```
module.exports = app;
```

这里我们通过 `require()` 加载了 `express`、`path` 等模块, 以及 `routes` 文件夹下的 `index.js` 和 `users.js` 路由文件。 下面来讲解每行代码的含义。

```
var app = express()
```

生成一个 `express` 实例 `app`。

```
app.set('views', path.join(__dirname, 'views'))
```

设置 `views` 文件夹为存放视图文件的目录, 即存放模板文件的地方, `dirname` 为全局变量, 存储当前正在执行的脚本所在的目录

```
app.set('view engine', 'jade')
```

设置视图模板引擎为 `jade`

```
app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')))
```

设置 `public/favicon.ico` 为 `favicon` 图标

```
app.use(logger('dev'))
```

加载日志中间件

```
app.use(bodyParser.json())
```

加载解析 `json` 的中间件

```
app.use(bodyParser.urlencoded({ extended: false }))
```

加载解析 `urlencoded` 请求体的中间件

```
app.use(cookieParser())
```

加载解析 `cookie` 的中间件

```
app.use(express.static(path.join(__dirname, 'public')))
```

设置 `public` 文件夹为存放静态文件的目录

```
app.use('/', index);
```

```
app.use('/users', users);
```

路由控制器

```
// catch 404 and forward to error handler
```

```
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});
```

捕获 404 错误, 并转发到错误处理器

```
// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  // render the error page
  res.status(err.status || 500);
  res.render('error');
});
```

开发环境下的错误处理器，将错误信息渲染 error 模版并显示到浏览器中

```
module.exports = app
```

导出 app 实例供其他模块调用

再看看 ./bin/www 文件（启动文件几乎是不需要改动的）：

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var app = require('../app');
var debug = require('debug')('microblog:server');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);

/**
 * Listen on provided port, on all network interfaces.
 */
```

```
server.listen(port);
server.on('error', onError);
server.on('listening', onListening);

/**
 * Normalize a port into a number, string, or false.
 */

function normalizePort(val) {
  var port = parseInt(val, 10);

  if (isNaN(port)) {
    // named pipe
    return val;
  }

  if (port >= 0) {
    // port number
    return port;
  }

  return false;
}

/**
 * Event listener for HTTP server "error" event.
 */

function onError(error) {
  if (error.syscall !== 'listen') {
    throw error;
  }

  var bind = typeof port === 'string'
    ? 'Pipe ' + port
    : 'Port ' + port;

  // handle specific listen errors with friendly messages
  switch (error.code) {
    case 'EACCES':
      console.error(bind + ' requires elevated privileges');
      process.exit(1);
      break;
```

```

    case 'EADDRINUSE':
      console.error(bind + ' is already in use');
      process.exit(1);
      break;
    default:
      throw error;
  }
}

/**
 * Event listener for HTTP server "listening" event.
 */

function onListening() {
  var addr = server.address();
  var bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  debug('Listening on ' + bind);
}

```

根据 Express 版本的迭代更新，很多由 express 集成的模块功能又独立出去了。旧版本的创建服务器功能已移植到 http 模块上了。

```
#!/usr/bin/env node
```

表明是 node 可执行文件

```

var app = require('../app');
var debug = require('debug')('microblog:server');
var http = require('http');

```

引入模块依赖

```

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

```

设置端口号

```
var server = http.createServer(app);
```

创建 HTTP 服务器

```

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);

```

在所有网络接口上监听所提供的端口

再看一下 ./routes/index.js 文件：

```
var express = require('express');
```

```
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

生成一个路由实例用来捕获访问主页的 GET 请求，导出这个路由并在 app.js 中通过 app.use('/', routes); 加载。这样，当访问主页时，就会调用 res.render('index', { title: 'Express' }); 渲染 views/index.jade 模版并显示到浏览器中。

再看一下 ./views/index.jade 文件：

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

可以看出该模板继承了 layout 模板，在渲染模板时我们传入了一个变量 title 值为 express 字符串，模板引擎会将所有 {title} 替换为 express，然后将渲染后生成的 html 显示到浏览器中。

jade 文档：<http://jade-lang.com/>

这个不习惯使用 jade 的话是可以改成 html 传统模式的，具体方法如下：

打开 app.js 这个文件，找到以下行：

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

替换成

```
app.set('views', path.join(__dirname, 'views'));
//app.set('view engine', 'ejs');
app.engine('html', require('ejs').__express); //把 ejs 修改成 html 模块
app.set('view engine', 'html');
```

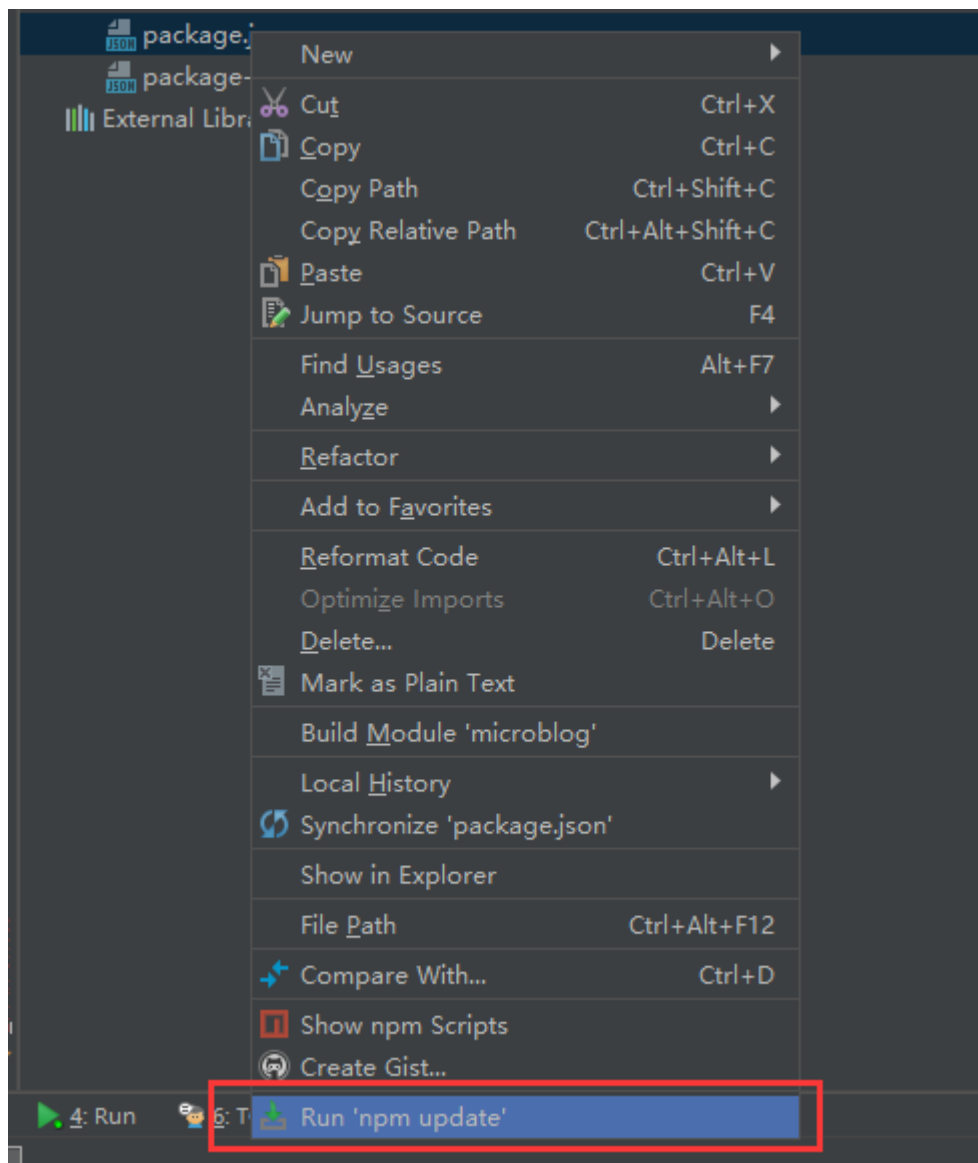
打开 package.json 文件，找到以下行：

```
"jade": "^1.11.0",
```

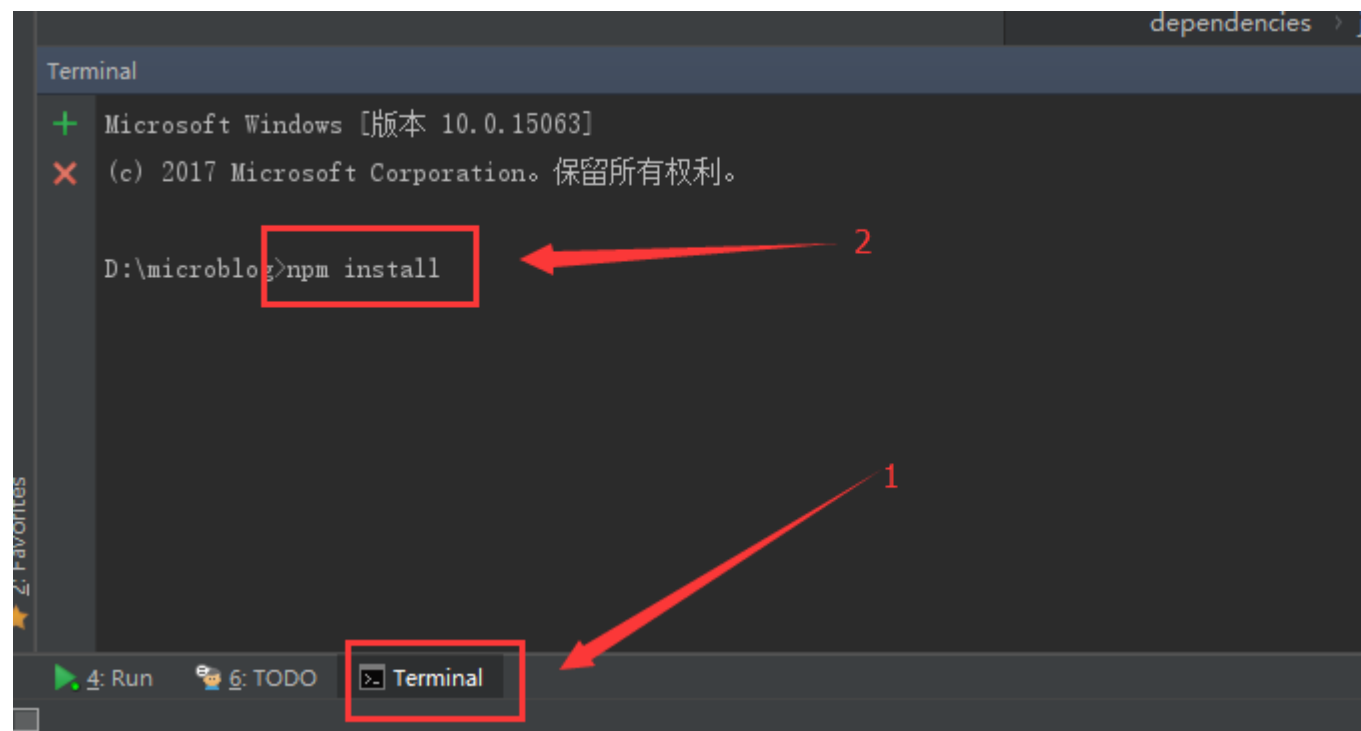
替换成

```
"ejs": "~2.5.6",
```

我们保存一下，然后选中 package.json 文件右键选择 run 'npm update' 如图：



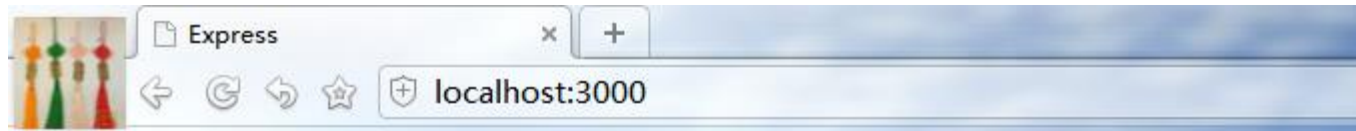
或者打开控制台输入“npm install” 或者输入“npm update” 回车，如图：



把 ./views/ 目录下所有 jade 文件改成 html 文件
把 index.html 内容改成

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title><%= title %></title>
  <link rel='stylesheet', href='/stylesheets/style.css'>
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
</body>
</html>
```

IDEA 重启一下项目，浏览器访问 <http://127.0.0.1:3000> 效果如下：



Express

Welcome to Express

发现可以使用 html 样板了。

路由控制（核心）

上面我们学习了如何创建和启动一个项目，并且对整个项目结构进行了讲解和运行流程，接下来开始讲解 Express 框架的基本使用与路由控制。

Express 文档：<http://www.nodeclass.com/api/express4.html>

Express 文档（中文）：<http://www.expressjs.com.cn/4x/api.html>

工作原理

在 ./routes/index.js 文件中有以下代码：

```
router.get('/', function(req, res, next) {  
  res.render('index', { title: 'Express' });  
});
```

这段代码的意思是当访问主页时，调用 jade 模板引擎，来渲染/views/index.jade 模板文件（即将 title 变量全部替换为字符串 Express），生成静态页面并显示在浏览器中。可以看出请求访问浏览器地址到返回数据整个运行流程是：

```
HTTP-->app.js-->routes-->response
```

也就说每一个访问都需要创建一个 routes，这样，随着时间的推移，app.js 文件不断地 require，不断地 app.use，主入口文件 app.js 就会变得非常臃肿难以维护了，也不符合代码模块化的思想。作为一个有梦想的程序员，需要向着架构师方向前进。代码不仅要写的好，层次要分明，复用率要高。

所以我们要对代码进行改进，Express 官方给出的写法是在 app.js 中实现了简单的路由分配，然后再去 index.js 中找到对应的路由函数，最终实现路由功能。我们要把所有请求的 routes 配置到/routes/index.js 文件上，这样只需要维护一个文件即可，app.js 文件只有一个总路由。打开 app.js 文件，把

```
var index = require('./routes/index');  
var users = require('./routes/users');
```

改成：

```
var routes = require('./routes/index');
```

把

```
app.use('/', index);  
app.use('/users', users);
```

改成：

```
routes(app);
```

修改/routes/index.js 文件内容改为如下：

```
module.exports = function(app) {  
  
  app.get('/', function (req, res) {  
    res.render('index', { title: 'Express' });  
  });  
  
};
```

再次运行项目，你会发现主页没有发生任何改变，说明我们的修改是可行的没有影响的。

路由规则

Express 封装了多种 http 请求，我们主要用的是 get 和 post 以及 all（以后跨域请求会用到），即 `app.get()` 和 `app.post()` 以及 `app.all()`。

`app.get()` 和 `app.post()` 的第一个参数都为请求的路径，第二个参数为处理请求的回调函数，回调函数有两个参数分别是 `req` 和 `res`，代表请求信息和响应信息。路径请求及对应的获取路径有以下几种形式：

`req.query`

```
// GET /find?f=one+two
req.query.f
// => "one two"

// GET /find?name=huafeng&sex=001&article[state]=001
req.query.name
// => "huafeng"

req.query.article.state
// => "001"
```

`req.body`

```
// POST user[name]=huafeng&user[email]=848989747@qq.com
req.body.user.name
// => "tobi"

req.body.user.email
// => "848989747@qq.com"

// POST { "name": "huafeng" }
req.body.name
// => "huafeng"
```

`req.params`

```
// GET /user/huafeng
req.params.name
// => "huafeng"

// GET /js/javascripts/article.js
req.params[0]
// => "javascripts/article.js"
```

`req.param(name)`

```
// ?name=huafeng
req.param('name')
// => "huafeng"

// POST name=huafeng
req.param('name')
// => "huafeng"

// /user/huafeng for /user/:name
req.param('name')
// => "huafeng"
```

可以看出：

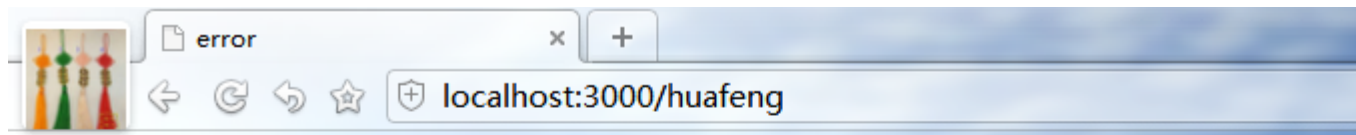
- `req.query`： 处理 `get` 请求，获取 `get` 请求参数
- `req.params`： 处理 `/:xxx` 形式的 `get` 或 `post` 请求，获取请求参数
- `req.body`： 处理 `post` 请求，获取 `post` 请求体
- `req.param()`： 处理 `get` 和 `post` 请求，但查找优先级由高到低为
`req.params`→`req.body`→`req.query`

路由规则还支持正则表达式，更多请查看 Express 官方文档：

<http://www.nodeclass.com/api/express4.html>

添加路由规则

当我们访问不存在的请求时, 会显示如下：



Not Found

404

```
Error: Not Found
    at D:\microblog\app.js:32:13
    at Layer.handle [as handle_request] (D:\microblog\node_modules\express\lib\router\la
    at trim_prefix (D:\microblog\node_modules\express\lib\router\index.js:317:13)
    at D:\microblog\node_modules\express\lib\router\index.js:284:7
    at Function.process_params (D:\microblog\node_modules\express\lib\router\index.js:33
    at next (D:\microblog\node_modules\express\lib\router\index.js:275:10)
    at D:\microblog\node_modules\express\lib\router\index.js:635:15
    at next (D:\microblog\node_modules\express\lib\router\index.js:260:14)
    at Function.handle (D:\microblog\node_modules\express\lib\router\index.js:174:3)
    at router (D:\microblog\node_modules\express\lib\router\index.js:47:12)
```

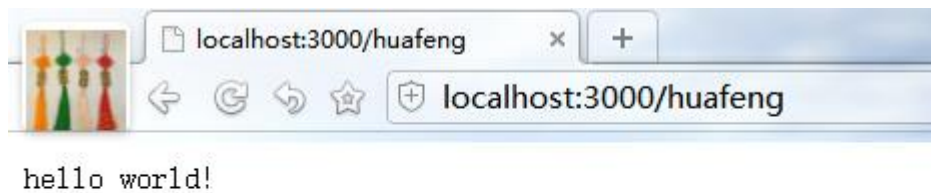
这是因为不存在 `/huafeng` 的路由规则，而且它也不是一个 `public` 目录下的文件，所以 `express` 返回了 `404 Not Found` 的错误。下面我们来添加这条路由规则，使得当访问 `localhost:3000/huafeng` 时，页面显示 `hello world!`

注意：以下修改仅用于测试，看到效果后再把代码还原回来。

修改 `/routes/index.js`，在 `app.get('/')` 函数后添加一条路由规则：

```
app.get('/huafeng', function (req, res) {
  res.send("hello world!");
});
```

重启项目之后，刷新浏览器如下：



创建路由就是如此简单！

模版引擎

接下来开始学习模板引擎。

什么是模板引擎

模板引擎（Template Engine）是一个将页面模板和要显示的数据结合起来生成 HTML 页面的工具。如果说上面讲到的 express 中的路由控制方法相当于 MVC 中的控制器的话，那模板引擎就相当于 MVC 中的视图。

模板引擎的功能是将页面模板和要显示的数据结合起来生成 HTML 页面。它既可以运行在服务器端又可以运行在客户端，大多数时候它都在服务器端直接被解析为 HTML，解析完成后再传输给客户端，因此客户端甚至无法判断页面是否是模板引擎生成的。有时候模板引擎也可以运行在客户端，即浏览器中，典型的代表就是 XSLT，它以 XML 为输入，在客户端生成 HTML 页面。但是由于浏览器兼容性问题，XSLT 并不是很流行。目前的主流还是由服务器运行模板引擎。在 MVC 架构中，模板引擎包含在服务器端。控制器得到用户请求后，从模型获取数据，调用模板引擎。模板引擎以数据和页面模板为输入，生成 HTML 页面，然后返回给控制器，由控制器交回客户端。——《Node.js 开发指南》

什么是 jade ？

jade 是模板引擎的一种，也是我们这个教程中使用的模板引擎，因为它使用起来十分简单，而且与 express 集成良好。

详情见：[jade 文档](#)

使用模板引擎

前面我们通过以下两行代码设置了模板文件的存储位置 and 使用的模板引擎：

```
app.set('views', path.join(__dirname, 'views'));
```



```
app.set('view engine', 'jade');
```

注意： 我们通过 `express -e mrioblog` 只是初始化了一个使用 `jade` 模板引擎的工程而已，比如 `node_modules` 下添加了 `jade` 模块，`views` 文件夹下有 `index.jade`。并不是说强制该工程只能使用 `jade` 不能使用其他的模板引擎比如 `ejs`，真正指定使用哪个模板引擎的是 `app.set('view engine', 'jade')`；刚才我们就已经改成我们比较熟悉的 `html` 了。

在 `/routes/index.js` 中通过调用 `res.render()` 渲染模版，并将其产生的页面直接返回给客户端。它接受两个参数，第一个是模板的名称，即 `views` 目录下的模板文件名，扩展名 `.jade`（注：刚才已改成 `html`）可选。第二个参数是传递给模板的数据对象，用于模板翻译。

打开 `views/index.html`，内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title><%= title %></title>
  <link rel='stylesheet', href='/stylesheets/style.css'>
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
</body>
</html>
```

当我们 `res.render('index', { title: 'Express' })`；时，模板引擎会把 `<%= title %>` 替换成 `Express`，然后把替换后的页面显示给用户。

渲染后生成的页面代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Express</title>
  <link rel='stylesheet', href='/stylesheets/style.css'>
</head>
<body>
  <h1>Express</h1>
  <p>Welcome to Express</p>
</body>
```

```
</html>
```

注意： 我们通过 `app.use(express.static(path.join(__dirname, 'public')))` 设置了静态文件目录为 `public` 文件夹，所以上面代码中的 `href='/stylesheets/style.css'` 就相当于 `href='public/stylesheets/style.css'`。

我们刚才改了模板引擎为 `html`，实质上是使用 `ejs` 的标签系统，要是学过 `jsp` 的估计对这个就非常熟悉了，它只有以下 3 个标签：

- `<% code %>`: JavaScript 代码。
- `<%= code %>`: 显示替换过 HTML 特殊字符的内容。
- `<%- code %>`: 显示原始 HTML 内容。

注意： `<%= code %>` 和 `<%- code %>` 的区别，当变量 `code` 为普通字符串时，两者没有区别。当 `code` 比如为

Hello

这种字符串时，`<%= code %>` 会原样输出

hello

而 `<%- code %>` 则会显示 H1 大的 `hello` 字符串。

我们可以在 `<% %>` 内使用 JavaScript 代码。下面是 `ejs` 的官方示例：

The Data

```
supplies: ['mop', 'broom', 'duster']
```

The Template

```
<ul>
<% for(var i=0; i<supplies.length; i++) {%>
  <li><%= supplies[i] %></li>
<% } %>
</ul>
```

The Result

```
<ul>
  <li>mop</li>
  <li>broom</li>
  <li>duster</li>
</ul>
```

开始搭建微博（microblog）

功能分析

搭建一个简单的具有登陆、注册、发表文章、修改文章、删除文章、退出功能的microblog。

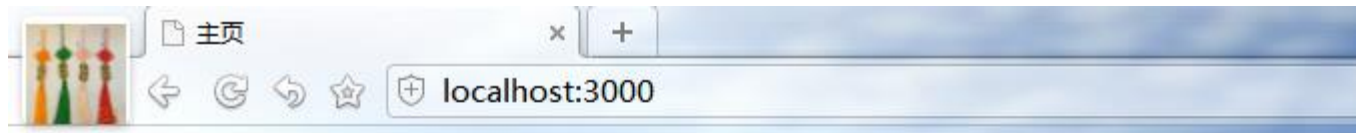
设计方向

未登录：主页顶部显示标题、欢迎语、登陆、注册按钮，下边部分则显示已发表的文章以及作者和发表日期。

登录后：主页顶部显示标题、欢迎语、发表、退出按钮，下边部分则显示已发表的文章以及作者和发表日期。

用户登陆和注册和退出和发表文章成功后都跳转返回到首页。

未登录时，主页（index.html）如下：



主页

Welcome to 主页

登录

注册

登陆页 (login.html) :



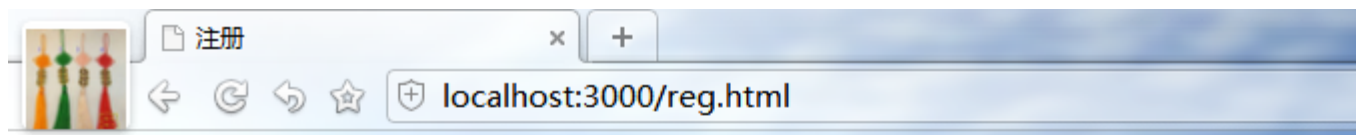
登录

用户名

密码

登录

注册页 (register.html) :



注册

用户名

请输入用户名

密码

请输入密码

确认密码

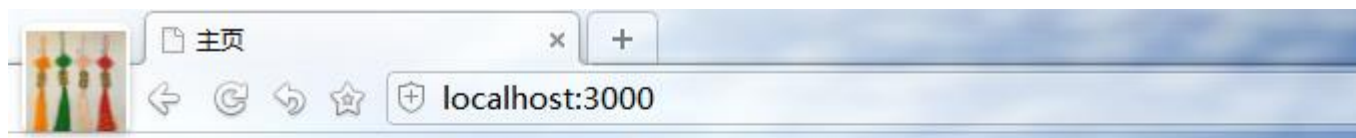
请再次输入密码

邮箱

请输入电子邮箱

注册

登录后，主页（index.html）如下：



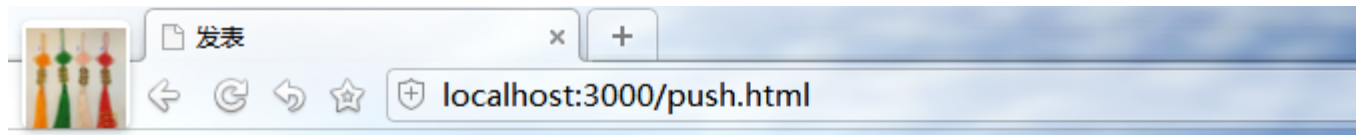
主页

张三, Welcome to 主页

发表

退出

文章发表页 (push_article.html) :



发表

标题

请输入文章标题

正文

请输入文章正文

发布

注意： 没有登出页，当点击“退出”后，退出登陆并返回到主页，上面的页面比较丑，这不是本文重点，以后会专门写一些文章介绍大前端的，手把手教会编写漂亮页面。
html 学习手册：<http://www.w3school.com.cn/>

路由规划

我们上面已经构思好了设计图。接下来就是要规划路由了，路由是整个项目的骨架部分，处在于核心位置，是页面与后台数据交互的转接点。所以我们需要优先考虑。

根据构思好的设计图，我们把路由规划成如下：

```
/: 首页
/login.html: 登陆
/reg.html: 注册
/push.html: 文章发表
/logout: 退出
```

对于/login.html 与/reg.html 只能是未登录的用户访问，而/push.html 与/logout 只能是已登录的用户访问， 根据登录用户与未登录用户主页会展示不同的内容。

修改/routes/index.js 文件，如下：

```
module.exports = function(app) {
  /**
   * 主页
   */
  app.get('/', function (req, res) {
    res.render('index', { title: '主页' });
  });

  /**
   * 登录跳转页面
   */
  app.get('/login.html', function (req, res) {
    res.render('login', { title: '登录' });
  });

  /**
   * 注册跳转页面
   */
  app.get('/reg.html', function (req, res) {
    res.render('register', { title: '注册' });
  });

  /**
   * 文章发表跳转页面
   */
  app.get('/push.html', function (req, res) {
    res.render('push_article', { title: '发表' });
  });

  /**
   * 退出登录方法
   */
}
```

```
app.get('/logout', function (req, res) {  
    res.redirect('/');//退出成功跳转到主页  
});  
};
```

完成这一步之后，如何针对登录与非登录用户展示不同内容呢？或者说如何判断是否已登录，实际上就是判断用户的状态。这里就涉及到了会话（session）机制了，使用 session 记录用户登录状态，并且访问数据库保存和读取用户信息。

使用数据库

之前我们已经选好了 MongoDB 作为本项目数据库，也已经安装好了。

想了解更多 MongoDB 的知识详情请查阅：<http://www.mongodb.org.cn/>

连接 MongoDB

之前我们虽然已经安装好并且启动成功了，但是我们需要连接上数据库后才能使用数据库。那么我们如何在 NodeJs 上使用 MongoDB 呢？这里我们就需要用到官方的 node-mongodb-native 驱动模块（其实还有更好的第三方驱动模块，以后再说），打开 package.json，在 dependencies 中添加一行：

```
"mongodb": "~2.2.31"
```

然后运行 npm install 或 npm update 更新依赖的模块，稍等片刻后 mongodb 模块就下载并安装完成了。

接下来我们在项目的根目录下创建 setting.js 文件，用项目的一些配置信息，比如数据库连接信息。我们将数据库命名为 microblog 与项目名一致。setting.js 文件内容如下：

```
module.exports = {  
    db: "microblog",  
    host: "127.0.0.1",  
    port: 27017,  
    url: "mongodb://127.0.0.1:27017/microblog"  
};
```

其中 db 是数据库的名称，host 是数据库的地址，port 是数据库的端口号，url 是数据库完整地址。

接下来在根目录下新建 models 文件夹，并在 models 文件夹下新建 db.js，添加如下代码：

```
var settings = require("../settings"),  
    Db = require('mongodb').Db,
```

```

Server = require('mongodb').Server;

module.exports = new Db(settings.db, new Server(settings.host,
settings.port,{
  socketOptions: { connectTimeoutMS: 500 },
  poolSize: 10,
  auto_reconnect: true
}), {
  numberOfRetries: 3,
  retryMilliseconds: 500
}),{safe: true});

```

其中通过 `new Db(settings.db, new Server(settings.host, settings.port, { socketOptions: { connectTimeoutMS: 500 }, poolSize: 10, auto_reconnect: true }, { numberOfRetries: 3, retryMilliseconds: 500 })), {safe: true});` 设置数据库名、数据库地址和数据库端口以及数据库连接池的一些参数配置创建了一个数据库连接实例，并通过 `module.exports` 导出该实例。这样，我们就可以通过 `require` 这个文件来对数据库进行读写了。

打开 `app.js`，在 `var routes = require('./routes/index');` 下添加：

```

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

```

会话支持

会话是一种持久的网络协议，用于完成服务器和客户端之间的一些交互行为。会话是一个比连接粒度更大的概念，一次会话可能包含多次连接，每次连接都被认为是会话的一次操作。在网络应用开发中，有必要实现会话以帮助用户交互。例如网上购物的场景，用户浏览了多个页面，购买了一些物品，这些请求在多次连接中完成。许多应用层网络协议都是由会话支持的，如 FTP、Telnet 等，而 HTTP 协议是无状态的，本身不支持会话，因此在没有额外手段的帮助下，前面场景中服务器不知道用户购买了什么。为了在无状态的 HTTP 协议之上实现会话，Cookie 诞生了。Cookie 是一些存储在客户端的信息，每次连接的时候由浏览器向服务器递交，服务器也向浏览器发起存储 Cookie 的请求，依靠这样的手段服务器可以识别客户端。我们通常意义上的 HTTP 会话功能就是这样实现的。具体来说，浏览器首次向服务器发起请求时，服务器生成一个唯一标识符并发送给客户端浏览器，浏览器将这个唯一标识符存储在 Cookie 中，以后每次再发起请求，客户端浏览器都会向服务器传送这个唯一标识符，服务器通过这个唯一标识符来识别用户。对于开发者来说，我们无须关心浏览器端的存储，需要关注的仅仅是如何通过这个唯一标识符来识别用户。很多服务端脚本语言都有会话功能，如 PHP，把每个唯一标识符存储到文件中。——《Node.js 开发指南》

`express` 也提供了会话中间件，默认情况下是把用户信息存储在内存中，但我们既然已经有了 MongoDB，不妨把会话信息存储在数据库中，便于持久维护。为了使用这一功能，我们需要借助 `express-session` 和 `connect-mongo` 这两个第三方中间件，在 `package.json` 中添加：

```
"express-session": "1.15.4",
"connect-mongo": "1.3.2"
```

运行 `npm install` 安装模块, 打开 `app.js`, 在 `var settings = require('./settings');` 下添加以下代码:

```
var session = require('express-session');
var MongoStore = require('connect-mongo')(session);
```

在 `app.use(express.static(path.join(__dirname, 'public')));` 下添加以下代码:

```
app.use(session({
  secret: settings.cookieSecret,
  key: settings.db, // cookie name
  cookie: {maxAge: 1000 * 60 * 60 * 24 * 30}, // 30 days
  store: new MongoStore({
    db: settings.db,
    host: settings.host,
    port: settings.port,
    url: settings.url
  })
}));
```

使用 `express-session` 和 `connect-mongo` 模块实现了将会话信息存储到 `mongodb` 中。`secret` 用来防止篡改 `cookie`, `key` 的值为 `cookie` 的名字, 通过设置 `cookie` 的 `maxAge` 值设定 `cookie` 的生存期, 这里我们设置 `cookie` 的生存期为 30 天, 设置它的 `store` 参数为 `MongoStore` 实例, 把会话信息存储到数据库中, 以避免丢失。在后面的小节中, 我们可以通过 `req.session` 获取当前用户的会话对象, 获取用户的相关信息。

注册和登录

页面设计

主页 `/views/index.html` 代码修改如下:

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
  <style type="text/css">
```

```

    a{text-decoration: none;display: inline-block;margin-
right:15px;color:#666;font-size:14px;text-align:center;border:1px solid
#e5e5e5;height:35px;width:100px;line-height:35px;border-radius:3px;}
    a.login{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    a.reg:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    a.upload:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    .but-edit{display: inline-block; height:30px;width:60px;border:1px
solid #00B7FF;border-radius: 3px;line-height:30px;text-align:
center;cursor: pointer;background: #19a4e1;color:#fff;}
  </style>
</head>
<body>
<h1><%= title %></h1>

<% if(user){ %>
<p> <%= user.userName %>,Welcome to <%= title %></p>
<%} else {%>
<p>Welcome to <%= title %></p>
<%}%>
<div>
  <% if(user){ %>
  <a class="login" href="/push.html">发表</a>
  <a class="reg" href="/loginout">退出</a>
  <%} else {%>
  <a class="login" href="/login.html">登录</a>
  <a class="reg" href="/reg.html">注册</a>
  <%}%>
  <!--<a class="upload" href="/upload.html">上传头像</a>-->

</div>
</body>
</html>

```

把/routes/index.js 中的

```

app.get('/', function (req, res) {
  res.render('index', { title: '主页' });
});

```

替换成

```

app.get('/', function (req, res) {
  res.render('index', { title: '主页',user:null });
});

```

```
});
```

在/views 目录下新建 login.html（登录页）代码如下：

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <style type="text/css">
      h1{border-bottom:1px dotted #e5e5e5;padding-bottom:10px;}
      .row{width:100%;height:35px;margin:15px 0px;}
      .row span{display: inline-block;width:80px;text-align:right;height:35px;line-height:35px;color:#666;font-size:14px;margin-right:15px;}
      .row input{border:1px solid #e5e5e5;border-radius:3px;color:#666;text-indent: 1em;width:200px;height:35px;line-height:35px;}
      .row input:hover{border:1px solid #19a4e1;}
      .but-sub{width:80px;height:35px;line-height:35px;text-align:center;border:1px solid #19a4e1;border-radius:3px;background:#19a4e1;cursor: pointer;color:#fff;font-size:14px;}
    </style>
  </head>
  <body>
    <h1><%= title %></h1>
    <div>
      <form action="/login.do" method="post">
        <div class="row">
          <span>用户名</span><input type="text"
name="userName" placeholder="请输入用户名">
        </div>
        <div class="row">
          <span>密码</span><input class="pwd"
name="password" type="password" placeholder="请输入密码">
        </div>
        <div class="row">
          <button class="but-sub" type="submit">登录
</button>
        </div>
      </form>
    </div>
  </body>
</html>
```

在/views 目录下新建 register.html（注册页）代码如下：

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <style type="text/css">
      h1{border-bottom:1px dotted #e5e5e5;padding-bottom:10px;}
      .row{width:100%;height:35px;margin:15px 0px;}
      .row span{display: inline-block;width:80px;text-align:right;height:35px;line-height:35px;color:#666;font-size:14px;margin-right:15px;}
      .row input{border:1px solid #e5e5e5;border-radius:3px;color:#666;text-indent: 1em;width:200px;height:35px;line-height:35px;}
      .row input:hover{border:1px solid #19a4e1;}
      .but-sub{width:80px;height:35px;line-height:35px;text-align:center;border:1px solid #19a4e1;border-radius:3px;background:#19a4e1;cursor: pointer;color:#fff;font-size:14px;}
    </style>
  </head>
  <body>
    <h1><%= title %></h1>
    <div>
      <form action="/reg.do" method="post">
        <div class="row">
          <span>用户名</span><input type="text"
name="userName" placeholder="请输入用户名">
        </div>
        <div class="row">
          <span>密码</span><input id="pwd1" class="pwd"
name="password" type="password" placeholder="请输入密码">
        </div>
        <div class="row">
          <span>确认密码</span><input id="pwd2" class="pwd"
name="password-repeat" type="password" placeholder="请再次输入密码">
        </div>
        <div class="row">
          <span>邮箱</span><input class="email" name="email"
type="text" placeholder="请输入电子邮箱">
        </div>
        <div class="row">
          <button class="but-sub" type="submit">注册
</button>
        </div>
      </form>
    </div>
  </body>
</html>
```

```

        </form>
    </div>
</body>
</html>

```

在/views 目录下新建 push_article.html（文章发表页）代码如下：

```

<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <style type="text/css">
      h1{border-bottom:1px dotted #e5e5e5;padding-bottom:10px;}
      .row{width:100%;height:35px;margin:15px 0px;}
      .row span{display: inline-block;width:80px;text-align:left;
height:35px;line-height:35px;color:#666;font-size:14px;margin-right:15px;}
      input{border:1px solid #e5e5e5;border-radius:3px;color:#666;text-indent: 1em;
width:620px;height:35px;line-height:35px;}
      input:hover{border:1px solid #19a4e1;}
      textarea{border:1px solid #e5e5e5;border-radius:3px;color:#666;font-size:14px;
width:600px;height:250px;line-height:22px;resize:none;padding:10px;}
      textarea:hover{border:1px solid #19a4e1;}
      .but-sub{width:80px;height:35px;line-height:35px;text-align:center;
border:1px solid #19a4e1;border-radius:3px;background:#19a4e1;cursor: pointer;
color:#fff;font-size:14px;}
    </style>
  </head>
  <body>
    <h1><%= title %></h1>
    <div>
      <form action="/push.do" method="post">
        <div class="row">
          <span>标题</span>
        </div>
        <input type="text" name="title" placeholder="请输入文章标题">
      </div>
      <div class="row">
        <span>正文</span>
      </div>
      <textarea rows="" cols="" name="content" placeholder="请输入文章正文"></textarea>
    </div>
  </body>
</html>

```



```

        <!-- <div class="row">
        </div> -->
        <div class="row">
            <button class="but-sub" type="submit">发布
</button>

        </div>
    </form>
</div>
</body>
</html>

```

到此为止已经创建好了登录页面、注册页面以及发表文章页面，接下来重启一下项目看一下效果。

拓展

每次我们更新代码后，都需要手动停止并重启应用，使用 supervisor 模块可以解决这个问题，每当我们保存修改的文件时，supervisor 都会自动帮我们重启应用。通过控制台：

```
npm install -g supervisor
```

全局安装 supervisor 。使用 supervisor 命令启动 app.js：

```
supervisor app
```

页面通知

接下来我们实现用户的注册和登陆，在这之前我们需要引入 flash 模块来实现页面通知（即成功与错误信息的显示）的功能。

什么是 flash?

我们所说的 flash 即 connect-flash 模块 <https://github.com/jaredhanson/connect-flash> flash 是一个在 session 中用于存储信息的特定区域。信息写入 flash，下一次显示完毕后即被清除。典型的应用是结合重定向的功能，确保信息是提供给下一个被渲染的页面。

在 package.json 添加一行代码：

```
"connect-flash": "0.1.1"
```

然后 npm install 安装 connect-flash 模块。修改 app.js，在 var settings = require('./settings'); 后添加：

```
var flash = require('connect-flash');
```

在 `app.use(logger('dev'))` 之前添加：

```
app.use(flash());
```

这样就可以使用 flash 功能了。

注册响应

前面我们已经把注册页面创建好了，当然点击注册是没有任何反应的，那是因为还没对 POST 请求做出处理功能，下面我们来实现它。

在 `/models` 目录下新建 `user.js` 文件，添加代码如下：

```
var mongodb = require('./db');

function User(user) {
  this.userName = user.userName;
  this.password = user.password;
  this.email = user.email;
};
module.exports = User;

//存储用户信息
User.prototype.save = function(callback) {
  //要存入数据库的用户文档
  var user = {
    userName: this.userName,
    password: this.password,
    email: this.email
  };

  //打开数据库
  mongodb.open(function (err, db) {
    if (err) {
      return callback(err); //错误，返回 err 信息
    }
    //读取 users 集合
    db.collection('users', function (err, collection) {
      if (err) {
        mongodb.close();
        return callback(err); //错误，返回 err 信息
      }
      //将用户数据插入 users 集合
      collection.insert(user, {
```

```

        safe: true
    }, function (err, user) {
        mongodb.close();
        if (err) {
            return callback(err); // 错误, 返回 err 信息
        }
        callback(null, user.ops[0]); // 成功! err 为 null, 并返回存储后的用户文档
    });
});
});
};

// 读取用户信息
User.get = function(name, callback) {
    // 打开数据库
    mongodb.open(function (err, db) {
        if (err) {
            return callback(err); // 错误, 返回 err 信息
        }
        // 读取 users 集合
        db.collection('users', function (err, collection) {
            if (err) {
                mongodb.close();
                return callback(err); // 错误, 返回 err 信息
            }
            // 查找用户名 (name 键) 值为 name 一个文档
            collection.findOne({
                userName: name
            }, function (err, user) {
                mongodb.close();
                if (err) {
                    return callback(err); // 失败! 返回 err 信息
                }
                callback(null, user); // 成功! 返回查询的用户信息
            });
        });
    });
};
};

```

这里通过 `User.prototype.save` 实现了用户信息的存储, 通过 `User.get` 实现了用户信息的读取。

打开 `/routes/index.js`, 在最前面添加如下代码:

```
var crypto = require('crypto'), User = require('../models/user.js');
```

通过 `require()` 引入 `crypto` 模块和 `user.js` 用户模型文件, `crypto` 是 NodeJs 的一个核心模块, 我们用它生成散列值来加密密码。

在 `app.get('/reg.html', function (req, res) { res.render('register', { title: '注册' }); });` 后面添加:

```
/**
 * 注册方法
 */
app.post('/reg.do', function (req, res) {
  var name = req.body.userName,
      password = req.body.password,
      password_re = req.body['password-repeat'];
  //检验用户两次输入的密码是否一致
  if (password_re != password) {
    req.flash('error', '两次输入的密码不一致!');
    return res.redirect('/reg.html');//返回注册页
  }
  //生成密码的 md5 值
  var md5 = crypto.createHash('md5'),
      password = md5.update(req.body.password).digest('hex');
  var newUser = new User({
    userName: name,
    password: password,
    email: req.body.email
  });
  //检查用户名是否已经存在
  User.get(newUser.userName, function (err, user) {
    if (err) {
      req.flash('error', err);
      return res.redirect('/');
    }
    if (user) {
      req.flash('error', '用户已存在!');
      return res.redirect('/reg.html');//返回注册页
    }
    //如果不存在则新增用户
    newUser.save(function (err, user) {
      if (err) {
        req.flash('error', err);
        return res.redirect('/reg.html');//注册失败返回注册页
      }
      req.session.user = user;//用户信息存入 session
      req.flash('success', '注册成功!');
    });
  });
});
```

```
        res.redirect('/');//注册成功后返回主页
    });
});
});
```

注意： 我们把用户信息存储在了 session 里，以后就可以通过 req.session.user 读取用户信息。

- req.body: 就是 POST 请求信息解析过后的对象，例如我们要访问 POST 来的表单内的 name="password" 域的值，只需访问 req.body['password'] 或 req.body.password 即可,当然也可以用 req.param() 获取数据。
- res.redirect: 重定向功能，实现了页面的跳转，更多关于 res.redirect 的信息请查阅：<http://expressjs.com/api.html#res.redirect> 。
- User: 在前面的代码中，我们直接使用了 User 对象。User 是一个描述数据的对象，即 MVC 架构中的模型。前面我们使用了许多视图和控制器，这是第一次接触到模型。与视图和控制器不同，模型是真正与数据打交道的工具，没有模型，网站就只是一个外壳，不能发挥真实的作用，因此它是框架中最根本的部分。

现在，启动应用，在浏览器输入 localhost:3000 注册试试吧！注册成功后显示如下：



发现我们并不知道是否注册成功，我们查看数据库中是否存入了用户的信息，打开一个命令行切换到 `mongodb/bin/`（注：保证数据库已打开的前提下，`mongodb/bin/`指 MongoDB 安装时的目录），输入：

```
mongo
```

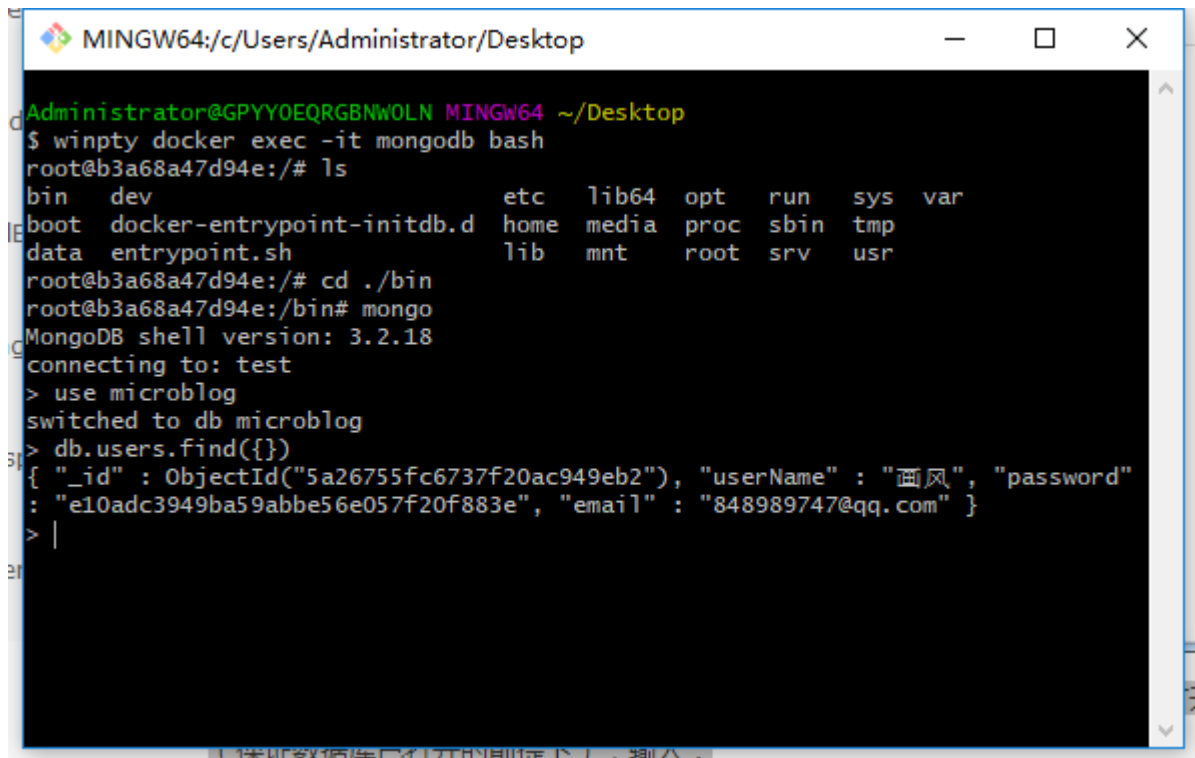
回车 然后输入：

```
use microblog
```

回车 然后输入：

```
db.users.find({})
```

发现可以看到用户信息已经成功存入数据库。
由于本文 MongoDB 是安装在 Docker 上的, 命令有所不同, 但效果是一样的, 如下图:



```
MINGW64:/c/Users/Administrator/Desktop
Administrator@GPYYOERGBNWOLN MINGW64 ~/Desktop
$ winpty docker exec -it mongodb bash
root@b3a68a47d94e:/# ls
bin    dev          etc    lib64  opt    run    sys    var
boot  docker-entrypoint-initdb.d  home  media  proc  sbin  tmp
data  entrypoint.sh  lib    mnt    root  srv    usr
root@b3a68a47d94e:/# cd ./bin
root@b3a68a47d94e:/bin# mongo
MongoDB shell version: 3.2.18
connecting to: test
> use microblog
switched to db microblog
> db.users.find({})
{ "_id" : ObjectId("5a26755fc6737f20ac949eb2"), "userName" : "画风", "password" : "e10adc3949ba59abbe56e057f20f883e", "email" : "848989747@qq.com" }
> |
```

接下来我们实现当注册成功返回主页时，顶部会显示“XXX, Welcome to 主页”字样，即添加 flash 的页面通知功能。

打开/routes/index.js 文件，把

```
app.get('/', function (req, res) {
  res.render('index', { title: '主页', user: null });
});
```

修改成

```
app.get('/', function (req, res) {
  res.render('index', { title: '主页',
    user: req.session.user,
    success: req.flash('success').toString(),
    error: req.flash('error').toString()
  });
});
```

将 app.get('/reg.html') 修改如下：

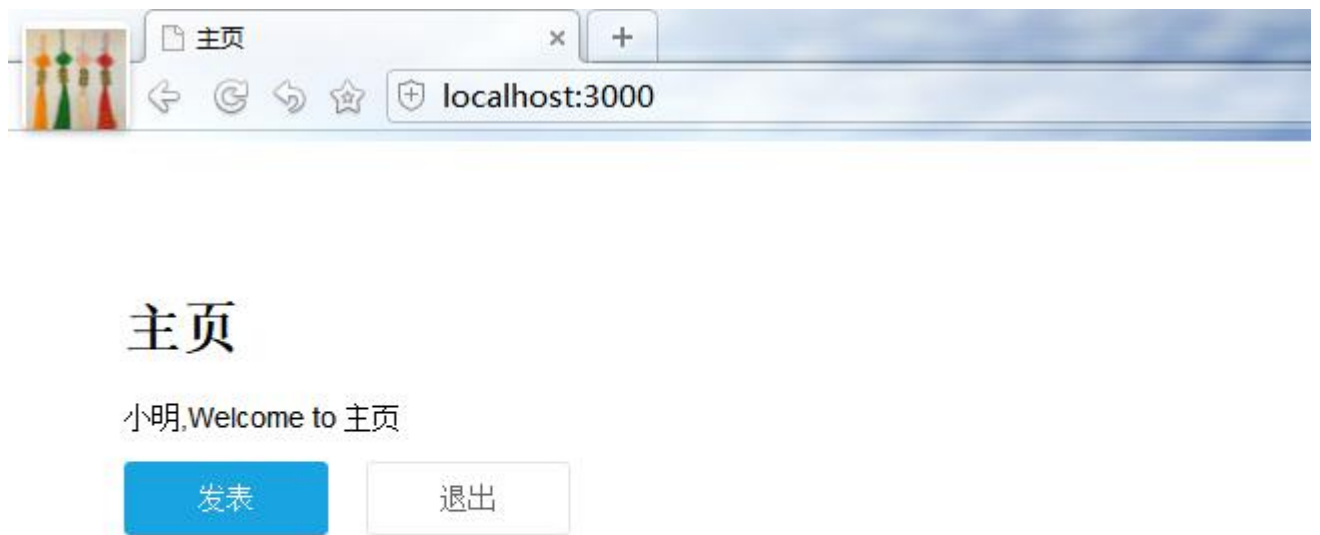
```
app.get('/reg.html', function (req, res) {
  res.render('register', { title: '注册',
    user: req.session.user,
```

```
    success: req.flash('success').toString(),  
    error: req.flash('error').toString()  
  });  
});
```

打开/routes/register.html 文件，在<h1><%= title %></h1>下添加：

```
<% if(error){ %>  
    <div style="color:red;margin-top:15px;"><%=error%></div>  
    <%}%>
```

然后我们再运行项目，注册成功后显示如下：



我们通过 session 的使用实现了对用户状态的检查，再根据不同的用户状态显示不同的按钮信息。说一下大体流程： 用户注册成功后，把用户信息保存到 session，页面跳转到主页，显示 XXX, Welcom to 主页 的字样。同时把 session 的信息 赋值给了 user，在渲染 index.html 文件时通过检测 user 判断用户是否在线，根据用户状态的不同显示不同的按钮信息。

success: req.flash('success').toString() 的意思是把成功的信息赋值给变量 success
error: req.flash('error').toString() 的意思是将错误的信息赋值给变量 error
然后我们在渲染 ejs 模版文件时传递这两个变量来进行检测并显示通知。

登录与退出响应

现在实现用户登录功能。

打开/routes/index.js 在 app.get('/login.html') 下添加一下代码：

```
/**
 * 登录方法
 */
app.post('/login', function (req, res) {
  //生成密码的 md5 值
  var md5 = crypto.createHash('md5'),
      password = md5.update(req.body.password).digest('hex');
  //检查用户是否存在
  User.get(req.body.userName, function (err, user) {
    if (!user) {
      req.flash('error', '用户不存在!');
      return res.redirect('/login.html');//用户不存在则跳转到登录页
    }
    //检查密码是否一致
    if (user.password != password) {
      req.flash('error', '密码错误!');
      return res.redirect('/login.html');//密码错误则跳转到登录页
    }
    //用户名密码都匹配后，将用户信息存入 session
    req.session.user = user;
    req.flash('success', '登陆成功!');
    res.redirect('/');//登陆成功后跳转到主页
  });
});
```

将 app.get('/login.html') 修改成如下：

```
app.get('/login.html', function (req, res) {
  res.render('login', { title: '登录',
    user: req.session.user,
    success: req.flash('success').toString(),
    error: req.flash('error').toString()
  });
});
```

这样就不会出现 'user is not defined' 的错误了（ejs 的标签相对于 jsp 来说稍微弱了）。

接下来是实现退出功能：

把 `app.set("/logout")` 修改成如下：

```
app.get('/logout', function (req, res) {
  req.session.user = null;
  req.flash('success', '退出成功!');
  res.redirect('/');//退出成功后跳转到主页
});
```

实现退出功能非常简单，只有一行代码：`req.session.user = null`；把 session 中的 user 赋值为 null，丢弃掉用户信息即可。

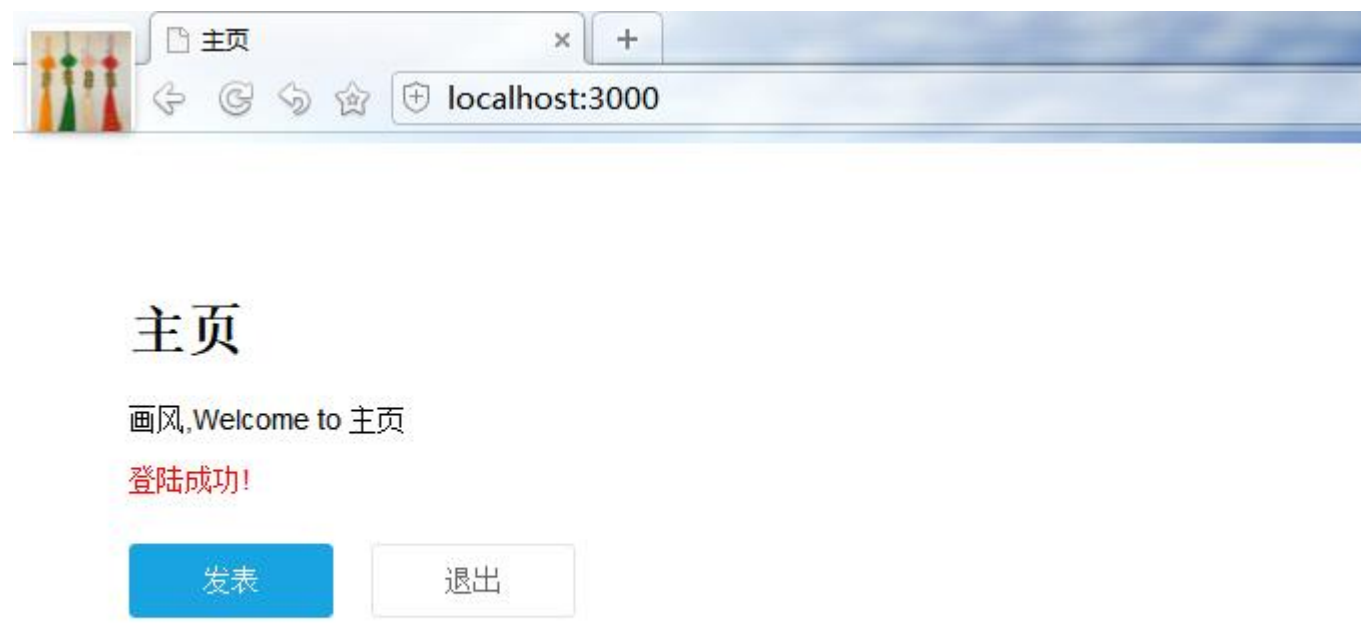
把 `/views/index.html` 修改成如下（接收一下 success 消息）：

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
  <style type="text/css">
    a{text-decoration: none;display: inline-block;margin-
right:15px;color:#666;font-size:14px;text-align:center;border:1px solid
#e5e5e5;height:35px;width:100px;line-height:35px;border-radius:3px;}
    a.login{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    a.reg:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    a.upload:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    .but-edit{display: inline-block; height:30px;width:60px;border:1px
solid #00B7FF;border-radius: 3px;line-height:30px;text-align:
center;cursor: pointer;background: #19a4e1;color:#fff;}
    .tips{color: red;height: 35px;margin: 5px 0px;}
  </style>
</head>
<body>
<h1><%= title %></h1>

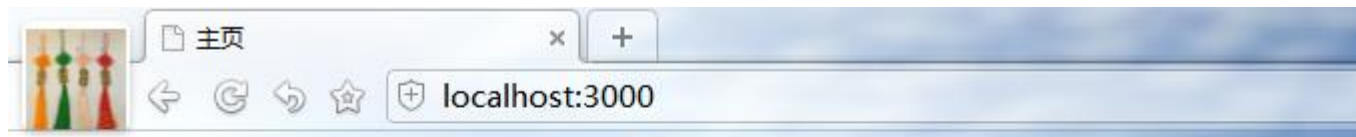
<% if(user){ %>
<p> <%= user.userName %>,Welcome to <%= title %></p>
<%} else {%>
<p>Welcome to <%= title %></p>
<%}%>
<% if(success){ %><div class="tips"><%= success%></div><%}%>
<div>
  <% if(user){ %>
  <a class="login" href="/push.html">发表</a>
```

```
<a class="reg" href="/loginout">退出</a>
<%} else {%>
<a class="login" href="/login.html">登录</a>
<a class="reg" href="/reg.html">注册</a>
<%}%>
<!--<a class="upload" href="/upload.html">上传头像</a-->
</div>
</body>
</html>
```

登录成功后页面如下：



退出成功后页面如下：



主页

Welcome to 主页

退出成功!

登录

注册

目前，我们已经实现了注册、登录功能，同时根据不同的登录状态显示不同的按钮。

页面控制

虽然我们上面已经实现了登录注册功能,但是并不能阻止比如已登录的用户再次访问注册页面 <http://localhost:3000/reg.html> ,这样有可能就会造成重复注册或登录。为此,我们是要控制访问页面的权限。即注册和登录页面应该阻止已登录的用户访问,退出后以及之后我们需要实现的发表、修改和删除文章只对已登录的用户访问。这需要如何控制呢?我们只需要把用户登录状态的检查放到路由中间件中,在每个路径前增加路由中间件,即可实现页面权限控制。我们增加 `checkNotLogin` (判断没有登录) 和 `checkLogin` (判断已登录) 两个函数来实现这个功能。

```
function checkLogin(req, res, next) {  
  if (!req.session.user) {  
    req.flash('error', '未登录!');  
    res.redirect('/login.html');  
  } else {  
    next();  
  }  
}  
  
function checkNotLogin(req, res, next) {  
  if (req.session.user) {
```

```

    req.flash('error', '已登录!');
    res.redirect('back');//返回之前的页面
  } else{
    next();
  }
}
}

```

checkNotLogin 和 checkLogin 用来检测是否登陆，并通过 next() 转移控制权，检测到未登录则跳转到登录页，检测到已登录则跳转到前一个页面。

最终/routes/index.js 文件代码如下：

```

var crypto = require('crypto'), User = require('../models/user.js');

module.exports = function(app) {
  /**
   * 主页
   */
  app.get('/', function (req, res) {
    res.render('index', { title: '主页',
      user: req.session.user,
      success: req.flash('success').toString(),
      error: req.flash('error').toString()
    });
  });

  /**
   * 登录跳转页面
   */
  app.get('/login.html', checkNotLogin);
  app.get('/login.html', function (req, res) {
    res.render('login', { title: '登录',
      user: req.session.user,
      success: req.flash('success').toString(),
      error: req.flash('error').toString()
    });
  });

  /**
   * 登录方法
   */
  app.post('/login.do', checkNotLogin);
  app.post('/login.do', function (req, res) {
    //生成密码的 md5 值

```

```

    var md5 = crypto.createHash('md5'),
        password = md5.update(req.body.password).digest('hex');
    //检查用户是否存在
    User.get(req.body.userName, function (err, user) {
        if (!user) {
            req.flash('error', '用户不存在!');
            return res.redirect('/login.html');//用户不存在则跳转到登录页
        }
        //检查密码是否一致
        if (user.password !== password) {
            req.flash('error', '密码错误!');
            return res.redirect('/login.html');//密码错误则跳转到登录页
        }
        //用户名密码都匹配后, 将用户信息存入 session
        req.session.user = user;
        req.flash('success', '登陆成功!');
        res.redirect('/');//登陆成功后跳转到主页
    });
});

/**
 * 注册跳转页面
 */
app.get('/reg.html', checkNotLogin);
app.get('/reg.html', function (req, res) {
    res.render('register', { title: '注册',
        user: req.session.user,
        success: req.flash('success').toString(),
        error: req.flash('error').toString()
    });
});

/**
 * 注册方法
 */
app.post('/reg.do', checkNotLogin);
app.post('/reg.do', function (req, res) {
    var name = req.body.userName,
        password = req.body.password,
        password_re = req.body['password-repeat'];
    //检验用户两次输入的密码是否一致
    if (password_re !== password) {
        req.flash('error', '两次输入的密码不一致!');
        return res.redirect('/reg.html');//返回注册页
    }
});

```

```

    }
    //生成密码的 md5 值
    var md5 = crypto.createHash('md5'),
        password = md5.update(req.body.password).digest('hex');
    var newUser = new User({
        userName: name,
        password: password,
        email: req.body.email
    });
    //检查用户名是否已经存在
    User.get(newUser.userName, function (err, user) {
        if (err) {
            req.flash('error', err);
            return res.redirect('/');
        }
        if (user) {
            req.flash('error', '用户已存在!');
            return res.redirect('/reg.html');//返回注册页
        }
        //如果不存在则新增用户
        newUser.save(function (err, user) {
            if (err) {
                req.flash('error', err);
                return res.redirect('/reg.html');//注册失败返回主册页
            }
            req.session.user = user;//用户信息存入 session
            req.flash('success', '注册成功!');
            res.redirect('/');//注册成功后返回主页
        });
    });
});

/**
 * 文章发表跳转页面
 */
app.get('/push.html', checkLogin);
app.get('/push.html', function (req, res) {
    res.render('push_article', { title: '发表' });
});

/**
 * 退出登录方法
 */
app.get('/logout', checkLogin);

```

```

app.get('/logout', function (req, res) {
  req.session.user = null;
  req.flash('success', '退出成功!');
  res.redirect('/');//退出成功跳转到主页
});

function checkLogin(req, res, next) {
  if (!req.session.user) {
    req.flash('error', '未登录!');
    res.redirect('/login.html');
  } else{
    next();
  }
}

function checkNotLogin(req, res, next) {
  if (req.session.user) {
    req.flash('error', '已登录!');
    res.redirect('back');//返回之前的页面
  } else{
    next();
  }
}
};

```

注意：为了维护用户状态和 flash 的通知功能，我们给每个 ejs 模版文件传入了以下三个值：

```

user: req.session.user,
success: req.flash('success').toString(),
error: req.flash('error').toString()

```

发表文章

现在我们的微博已经具备了用户注册、登陆、页面权限控制的功能，接下来我们完成微博最核心的部分——发表文章。在这一节，我们将会实现发表文章的功能，完成整个微博的设计。

文章模型

前面我们已经写好了文章发表的页面（push_article.html），接下来仿照用户模型，把文章模型命名为 Article，它拥有与 User 相识的接口，分别是 Article.get 和 Article.prototype.save。

Article.get 的功能是从数据库中获取文章，可以按指定用户获取，也可以获取全部的内容。

Article.prototype.save 是 Article 对象原型的方法，用来将文章保存到数据库。

在 models 文件夹下新建 article.js，添加如下代码：

```
/**
 * http://usejsdoc.org/
 */
var mongodb = require('./db');

function Article(userName, title, content) {
  this.userName = userName;
  this.title = title;
  this.content = content;
}

module.exports = Article;

// 存储一篇文章及其相关信息
Article.prototype.save = function(callback) {
  var date = new Date();
  // 存储各种时间格式，方便以后扩展
  var time = {
    date : date,
    year : date.getFullYear(),
    month : date.getFullYear() + "-" + (date.getMonth() + 1),
    day : date.getFullYear() + "-" + (date.getMonth() + 1) +
date.getDate(),
    minute : date.getFullYear() + "-" + (date.getMonth() + 1)
+ "-" + date.getDate() + " " +
date.getHours() + ":" + (date.getMinutes() < 10 ? '0' +
date.getMinutes() : date.getMinutes())
  };

  // 要存入数据库的文档
  var article = {
    userName : this.userName,
    title : this.title,
    content : this.content,
    time : time
  };

  // 打开 mongodb
  mongodb.open(function(err, db) {
    if(err) {
```



```

    });
    callback(null, docs); //成功！以数组形式返回
    查询的结果
  });
});
};

```

发表响应

接下来我们实现文章发表功能，打开/routes/index.js 在顶部把 var crypto = require('crypto'), User = require('../models/user.js'); 改为：

```

var crypto = require('crypto'), User = require('../models/user.js'), Article
= require('../models/article.js');

```

在 app.get('/push.html') 下面添加：

```

/**
 * 文章发表方法
 */
app.post('/push.do', checkLogin);
app.post('/push.do', function (req, res) {
  var currentUser = req.session.user,
      post = new Article(currentUser.userName, req.body.title,
req.body.content);
  post.save(function (err) {
    if (err) {
      req.flash('error', err);
      return res.redirect('/');
    }
    req.flash('success', '发布成功!');
    res.redirect('/');//发表成功跳转到主页
  });
});

```

最后，我们修改 /views/index.html ，让主页底部显示发表过的文章及其相关信息。

打开 /views/index.html ，修改如下：

```

<!DOCTYPE html>
<html>
<head>

```

```

<title><%= title %></title>
<link rel='stylesheet' href='/stylesheets/style.css' />
<style type="text/css">
  a{text-decoration: none;display: inline-block;margin-
right:15px;color:#666;font-size:14px;text-align:center;border:1px solid
#e5e5e5;height:35px;width:100px;line-height:35px;border-radius:3px;}
  a.login{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
  a.reg:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
  a.upload:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
  .but-edit{display: inline-block; height:30px;width:60px;border:1px
solid #00B7FF;border-radius: 3px;line-height:30px;text-align:
center;cursor: pointer;background: #19a4e1;color:#fff;}
  .tips{color: red;height: 35px;margin: 5px 0px;}
</style>
</head>
<body>
<h1><%= title %></h1>

<% if(user){ %>
<p> <%= user.userName %>,Welcome to <%= title %></p>
<%} else {%>
<p>Welcome to <%= title %></p>
<%}%>
<% if(success){ %><div class="tips"><%= success%></div><%}%>
<div>
  <% if(user){ %>
  <a class="login" href="/push.html">发表</a>
  <a class="reg" href="/loginout">退出</a>
  <%} else {%>
  <a class="login" href="/login.html">登录</a>
  <a class="reg" href="/reg.html">注册</a>
  <%}%>
  <!--<a class="upload" href="/upload.html">上传头像</a-->
</div>
<% articles.forEach(function (article, index) { %>
<p><h2><span><%= article.title %></span></h2></p>
<p class="info">
  作者: <span><%= article.userName %></span> |
  日期: <%= article.time.minute %>
</p>
<p><%= article.content %></p>
<% }) %>
</body>
</html>

```

打开/routes/index.js 文件，把：

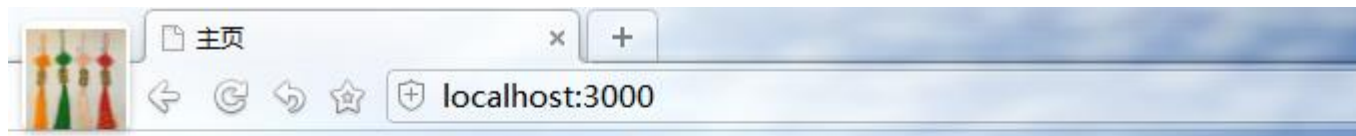
```
app.get('/', function (req, res) {  
  res.render('index', { title: '主页',  
    user:req.session.user,  
    success: req.flash('success').toString(),  
    error: req.flash('error').toString()  
  });  
});
```

修改成：

```
app.get('/', function (req, res) {  
  Article.get(null,function(err,articles){  
    if(err){  
      articles = [];  
    }  
    res.render('index', { title: '主页',  
      user:req.session.user,  
      articles:articles,  
      success: req.flash('success').toString(),  
      error: req.flash('error').toString()  
    });  
  })  
});
```

至此，我们的项目就建成了。

接下来我们重启一下项目，发表一篇文章，效果如下：



主页

画风, Welcome to 主页

发表

退出

这是第一篇文章！

作者：画风 | 日期：2017-12-6 16:01

This is the first article.

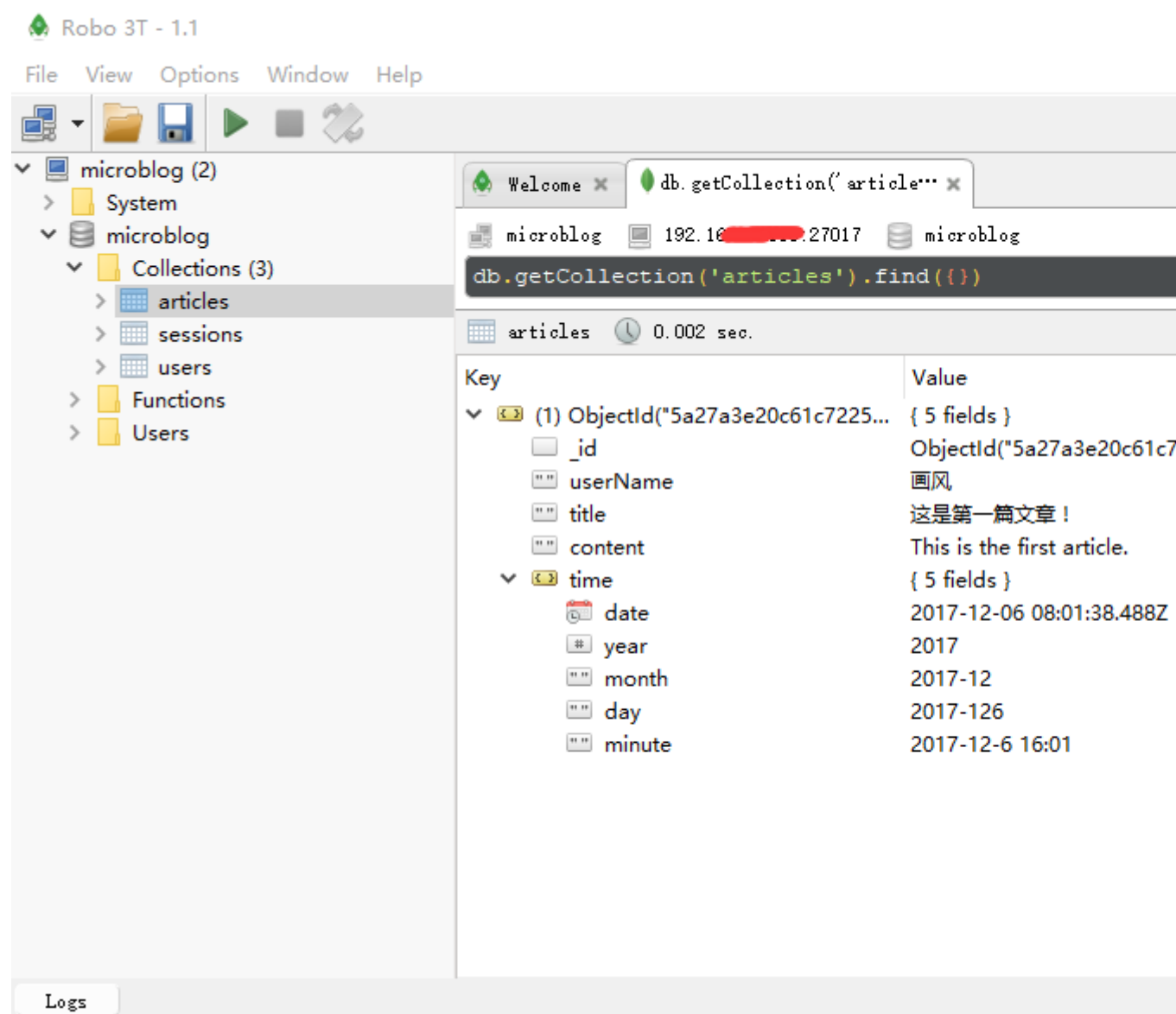
接下来我们查看一下数据库：

```
Administrator@GPYY0EQRBNWOLN MINGW64 ~/Desktop
$ winpty docker exec -it mongodb bash
root@b3a68a47d94e:/# cd ./bin
root@b3a68a47d94e:/bin# mongo
MongoDB shell version: 3.2.18
connecting to: test
> use microblog
switched to db microblog
> db.articles.find()
{ "_id" : ObjectId("5a27a3e20c61c722548b2482"), "userName" : "画风", "title" : "这是第一篇文章！", "content" : "This is the first article.", "time" : { "date" : ISODate("2017-12-06T08:01:38.488Z"), "year" : 2017, "month" : "2017-12", "day" : "2017-126", "minute" : "2017-12-6 16:01" } }
> |
```

拓展

使用图形化工具管理 MongoDB。对于数据量大的或者不熟悉习惯使用命令的童鞋来说，有一款图形化工具是多么好的选择。在这里给大家推荐一款叫做的 Robo 3T 的软件。这是一个基于 Shell 的跨平台开源 MongoDB 管理工具。嵌入了 JavaScript 引擎和 MongoDB mongo。只要你会使用 mongo shell，你就会使用 Robo 3T，它提供语法高亮、自动完成、差别视图等。

下载安装 Robo 3T 后，运行我们的项目，注册一个用户并发表几篇文章，初次打开 Robo 3T，点击 Create 创建一个名为 microblog（名字自定）的数据库链接（默认监听 localhost:27017），点击 Connect 就连接到数据库了。如图所示：



编辑与删除文章

上面我们已经完成了文章的发表功能，接下来我们添加文章编辑与删除功能。

设计：当用户登录成功后能够删除编辑和删除已发表的文章（暂时不考虑文章权限问题，每个登录用户都能操作），编辑文章时我们设定暂时只修改文章的内容，标题不做改变。

现在我们开始做文章编辑功能，打开/views/index.html 文件，修改成如下：

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
  <style type="text/css">
    a{text-decoration: none;display: inline-block;margin-
right:15px;color:#666;font-size:14px;text-align:center;border:1px solid
#e5e5e5;height:35px;width:100px;line-height:35px;border-radius:3px;}
    a.login{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    a.reg:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    a.upload:hover{border:1px solid #19a4e1;color:#fff;background:#19a4e1;}
    .but-edit{display: inline-block; height:30px;width:60px;border:1px
solid #00B7FF;border-radius: 3px;line-height:30px;text-align:
center;cursor: pointer;background: #19a4e1;color:#fff;}
    .tips{color: red;height: 35px;margin: 5px 0px;}
  </style>
</head>
<body>
<h1><%= title %></h1>

<% if(user){ %>
<p> <%= user.userName %>,Welcome to <%= title %></p>
<%} else {%>
<p>Welcome to <%= title %></p>
<%}%>
<% if(success){ %><div class="tips"><%= success%></div><%}%>
<div>
  <% if(user){ %>
    <a class="login" href="/push.html">发表</a>
    <a class="reg" href="/loginout">退出</a>
  <%} else {%>
    <a class="login" href="/login.html">登录</a>
    <a class="reg" href="/reg.html">注册</a>
  <%}%>
  <!--<a class="upload" href="/upload.html">上传头像</a>-->
</div>
```



```

<% articles.forEach(function (article, index) { %>
<p><h2><span><%= article.title %></span></h2></p>
<p class="info">
  作者: <span><%= article.userName %></span> |
  日期: <%= article.time.minute %>
</p>
<p><%= article.content %></p>
<span class="but-edit" onclick="editArticle('<%= article._id %>')">修改
</span>
<span class="but-edit" onclick="delArticle('<%= article._id %>')">删除
</span>
<% }) %>
</body>
</html>
<script>
  function editArticle(id){
    window.open("/edit/article.html?objId="+id);
  }
  function delArticle(id){
    window.location.href = "/del/article.html?objId="+id;
  }
</script>

```

至此，我们已经在每篇文章下面添加了“修改”和“删除”两个按钮，并且注册了两个响应跳转方法。

接下来，打开/models/article.js，在 var mongodb = require('./db'); 前面添加：

```
const ObjectID = require('mongodb').ObjectID;
```

在最后添加代码如下：

```

Article.findById = function(id, callback) {
  //打开数据库
  mongodb.open(function (err, db) {
    if (err) {
      return callback(err);
    }
    //读取 articles 集合
    db.collection('articles', function (err, collection) {
      if (err) {
        mongodb.close();
        return callback(err);
      }
    }
  }

```

```

        //根据 ID 进行查询
        collection.findOne({_id: new ObjectId(id)}, function (err, doc)
{
            mongodb.close();
            if (err) {
                return callback(err);
            }
            callback(null, doc);//返回查询的一篇文章
        });
    });
});
});
};

```

打开/routes/index.js ，在 app.post('/push.do') 后面添加如下：

```

/**
 * 文章编辑跳转页面
 */
app.get('/edit/article.html',checkLogin);
app.get('/edit/article.html',function(req, res){
    var _id = req.param("objId");
    Article.findById(_id,function(err,doc){
        if(err){
            doc = null;
        };
        console.log(JSON.stringify(doc));
        res.render("edit_article",{article:doc});
    })
});

```

在/views 目录下新建 edit_article.html ，添加如下代码：

```

<!DOCTYPE html>
<html>
  <head>
    <title>文章修改</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
    <style type="text/css">
      h1{border-bottom:1px dotted #e5e5e5;padding-bottom:10px;}
      .row{width:100%;height:35px;margin:15px 0px;}
      .row span{display: inline-block;width:80px;text-align:left;
height:35px;line-height:35px;color:#666;font-size:14px;margin-right:15px;}
    
```

```

    input{border:1px solid #e5e5e5;border-radius:3px;color:#666;text-
indent: 1em;width:620px;height:35px;line-height:35px;}
    input:hover{border:1px solid #19a4e1;}
    textarea{border:1px solid #e5e5e5;border-radius:3px;color:#666;font-
size:14px;width:600px;height:250px;line-height:22px;resize:none;padding:
10px;}
    textarea:hover{border:1px solid #19a4e1;}
    .but-sub{width:80px;height:35px;line-height:35px;text-
align:center;border:1px solid #19a4e1;border-
radius:3px;background:#19a4e1;cursor: pointer;color:#fff;font-size:14px;}
  </style>
</head>
<body>
  <h1>文章修改</h1>
  <div>
    <form action="/edit/article.do" method="post">
      <div class="row">
        <span>标题</span>
      </div>
      <input type="hidden" name="_id" value="<%=
article._id%>">
      <input type="text" name="title" value="<%=
article.title%>" placeholder="请输入文章标题">
      <div class="row">
        <span>正文</span>
      </div>
      <textarea rows="" cols="" name="content" placeholder="请输
入文章正文"><%= article.content%></textarea>
      <!-- <div class="row">
      </div> -->
      <div class="row">
        <button class="but-sub" type="submit">保存
      </button>
      </div>
    </form>
  </div>
</body>
</html>

```

现在，运行我们的项目看看吧。在文章下面，当我们点击 `修改` 后就会跳转到该文章对应的编辑页面了。接下来我们实现将修改后的文章提交到数据库。

打开 `/models/article.js`，在最后面添加：

```

Article.update = function (req, res, callback) {
    var _id = req.body._id;
    console.log(_id);
    var date = new Date();
    //存储各种时间格式，方便以后扩展
    var time = {
        date : date,
        year : date.getFullYear(),
        month : date.getFullYear() + "-" + (date.getMonth() + 1),
        day : date.getFullYear() + "-" + (date.getMonth() + 1) +
date.getDate(),
        minute : date.getFullYear() + "-" + (date.getMonth() + 1) + "-" +
date.getDate() + " " +
        date.getHours() + ":" + (date.getMinutes() < 10 ? '0' +
date.getMinutes() : date.getMinutes())
    };
    var data = {$set:{content : req.body.content, time : time}};

    //打开数据库
    mongodb.open(function (err, db) {
        if (err) {
            return callback(err);
        }
        //读取 articles 集合
        db.collection('articles', function (err, collection) {
            if (err) {
                mongodb.close();
                return callback(err);
            }
            //根据 ID 进行查询
            collection.update({_id: new ObjectId(_id)},data, function (err)
{
                mongodb.close();
                if (err) {
                    return callback(err);
                }
                callback(null); //成功！ 返回查询的信息
            });
        });
    });
}

```

打开 /routes/index.js ，在 app.get('/edit/article.html') 后面添加如下代码：

```

/**
 * 文章编辑功能
 */
app.post('/edit/article.do', checkLogin);
app.post('/edit/article.do', function(req, res){
  Article.update(req, res, function(err){
    if(err){
      return res.render("error");
    };
    req.flash('success', '修改成功!');
    res.redirect("/");
  })
});

```

现在，我们就可以编辑并保存文章了。赶紧试试吧！



接下来，我们实现删除文章的功能。打开 `/models/article.js`，在最后添加如下代码：

```
//删除一篇文章
```

```

Article.removeById = function(id, callback) {
  //打开数据库
  mongodb.open(function (err, db) {
    if (err) {
      return callback(err);
    }
    //读取 articles 集合
    db.collection('articles', function (err, collection) {
      if (err) {
        mongodb.close();
        return callback(err);
      }
      //根据 ID 查找并删除一篇文章
      collection.remove({_id: new ObjectId(id)}, {
        w: 1
      }, function (err) {
        mongodb.close();
        if (err) {
          return callback(err);
        }
        callback(null);
      });
    });
  });
};

```

打开/routes/index.js ，在 app.post('/edit/article.do') 后面添加以下代码：

```

/**
 * 文章删除功能
 */
app.get('/del/article.html',checkLogin);
app.get('/del/article.html',function(req,res){
  var _id = req.param("objId");
  Article.removeById(_id,function(err){
    if(err){
      return res.render("error",{message :err});
    }
    req.flash('success', '删除成功!');
    res.redirect("/");
  })
});

```

重启一下我们的项目，登录后选择点击文章下面的 删除 按钮，效果如下：



至此，我们整个项目已经开发完成，如果你能根据本文的所有代码都自己编写一次，相信你对 NodeJs 已经有大体的了解了，自己也能构建一个简单的网站了。当然本文是以最基本的 CRUD 进行讲解的，在现实项目中是比这个复杂非常非常多的。

高手进阶

在上面我们开发了一个简单的 microblog，对于真正的项目来说是非常简陋的，使用的技术也不多，项目结构也是混乱不堪。接下来，开始讲解一下如何对项目进行架构封装，使开发变得更加有条理，结构更加明了，代码简洁。这个就留到下一章《程序员的架构之路》。