





This repository ▾

Search or type a command 🔍

ExploreGistBlogHelp

Macro-Bin

nswbmw / N-blog

Watch ▾92

★ Star387

Fork245

HomePagesHistory

New Page

第1章 Express MongoDB 搭建多人博客

Edit PagePage HistoryClone URL

学习环境

node.js: v0.10.16+

快速开始

安装 Express

<>①🔗📖⚡📊🔗

express 是 Node.js 上最流行的 Web 开发框架。更多关于 express 的知识请查阅本书目录。我们用 express 来搭建我们的博客，打开命令行，输入：

```
npm install -g express
```

我们需要用全局模式安装 express，因为只有这样才能在命令行中使用它。目前 express 最新版本为 express 3.3.8。

新建一个工程

我们约定今后的学习把 D:\blog 文件夹作为我们的工程目录。

windows 下打开 cmd 切换到 D 盘，输入 `express -e ejs blog`（注意 express 3.* 中安装 ejs 不再是 -t 而是 -e，可以输入 `express -h` 查看），然后输入 `cd blog&npm install` 安装所需模块，如下图所示：

```
D:\>express -e ejs blog

create : blog
create : blog/package.json
create : blog/app.js
create : blog/public
create : blog/public/javascripts
create : blog/routes
create : blog/routes/index.js
create : blog/routes/user.js
create : blog/public/stylesheets
create : blog/public/stylesheets/style.css
create : blog/public/images
create : blog/views
create : blog/views/index.ejs

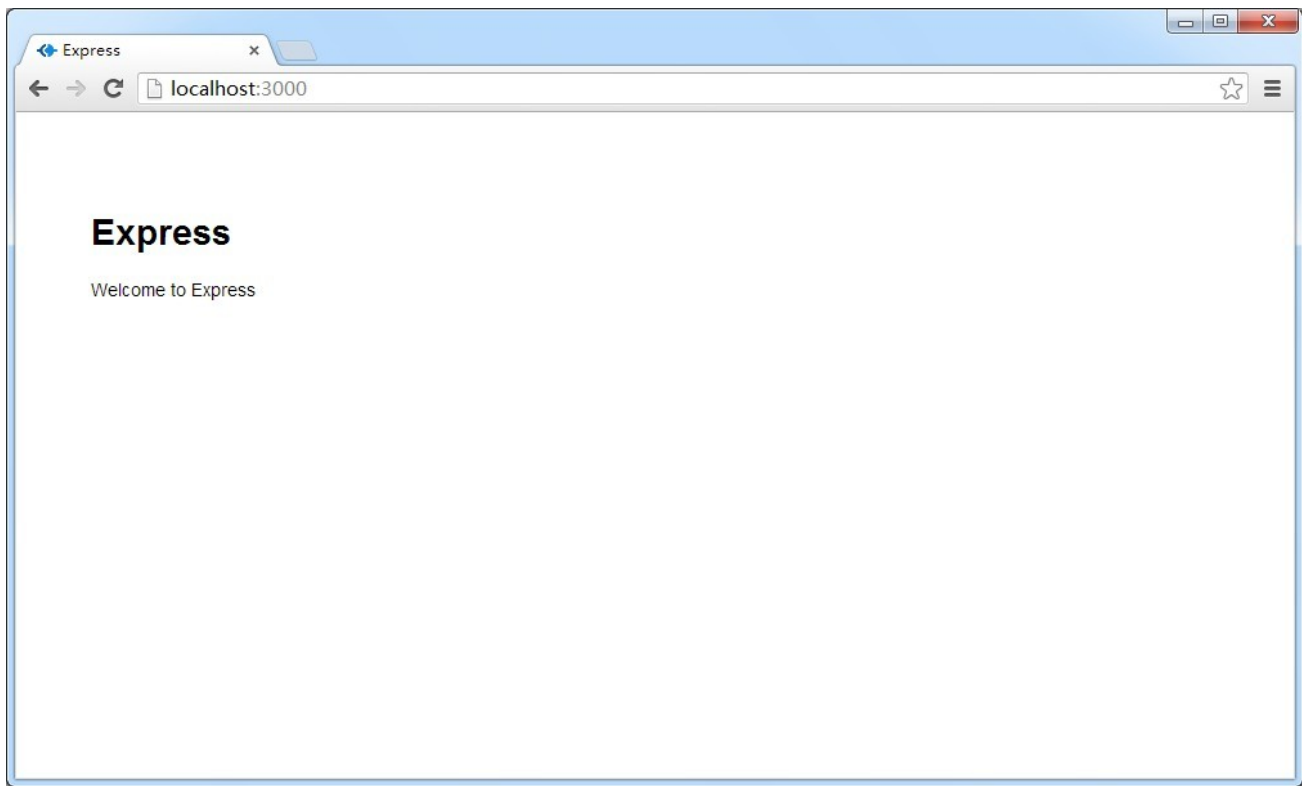
install dependencies:
$ cd blog && npm install

run the app:
$ node app

D:\>cd blog

D:\blog>npm install
npm http GET https://registry.npmjs.org/express/3.3.8
npm http GET https://registry.npmjs.org/ejs
npm http 304 https://registry.npmjs.org/ejs
npm http 200 https://registry.npmjs.org/express/3.3.8
npm http GET https://registry.npmjs.org/express/-/express-3.3.8.tgz
npm http 200 https://registry.npmjs.org/express/-/express-3.3.8.tgz
npm http GET https://registry.npmjs.org/mkdirp/0.3.5
```

安装完成后输入 `node app`，此时命令行中会显示 Express server listening on port 3000，在浏览器里输入 `localhost:3000`，如下所示：



我们用 `express` 初始化了一个工程，并指定使用 `ejs` 模板引擎，下一节我们讲解工程的内部结构。

工程结构

我们回头看看生成的工程目录里面有什么，打开 `D:\blog`，里面如图所示：

node_modules	2013/9/4/星期三 ...	文件夹	
public	2013/9/4/星期三 ...	文件夹	
routes	2013/9/4/星期三 ...	文件夹	
views	2013/9/4/星期三 ...	文件夹	
app.js	2013/9/4/星期三 ...	JS 文件	1 KB
package.json	2013/9/4/星期三 ...	JSON 文件	1 KB

app.js: 启动文件

package.json: 存储着工程的信息及所需的依赖模块，当在 `dependencies` 中添加依赖时，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依赖模块

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: 存放模版文件

打开 `app.js`，让我们看看里面究竟有什么东西：

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.favicon());
app.use(express.logger('dev'));
```

```
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

这里我们通过 `require()` 加载了 `express`、`http`、`path` 模块，以及 `routes` 文件夹下的 `index.js` 和 `user.js` 文件。更多关于模块及模块加载顺序的信息请查阅 `Modules` 章节。

因为 `express` 框架是依赖 `connect` 框架（Node 的一个中间件框架）创建而成的，以下内容可查阅 `connect` 文档：

<http://www.senchalabs.org/connect/> 和 `express` 官方文档：<http://expressjs.com/api.html> 了解更多内容。

`app.set('port', process.env.PORT || 3000)`：设置端口为 `process.env.PORT` 或 `3000`

`app.set('views', __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，`__dirname` 为全局变量，存储着当前正在执行脚本所在的目录名。

`app.set('view engine', 'ejs')`：设置视图模版引擎为 `ejs`

`app.use(express.favicon())`：`connect` 内建的中间件，使用默认的 `favicon` 图标，如果想使用自己的图标，需改为

`app.use(express.favicon(__dirname + '/public/images/favicon.ico'))`；这里我们把自定义的 `favicon.ico` 放到了 `public/images` 文件夹下。

`app.use(express.logger('dev'))`：`connect` 内建的中间件，在开发环境下使用，在终端显示简单的日志，比如在启动 `app.js` 后访问 `localhost:3000`，终端会输出：

```
Express server listening on port 3000
GET / 200 21ms - 206b
GET /stylesheets/style.css 304 4ms
```

假如你去掉这一行代码，不管你怎么刷新网页，终端都只有一行 `Express server listening on port 3000`。

`app.use(express.bodyParser())`：`connect` 内建的中间件，用来解析请求体，支持 `application/json`，`application/x-www-form-urlencoded`，和 `multipart/form-data`。

`app.use(express.methodOverride())`：`connect` 内建的中间件，可以协助处理 `POST` 请求，伪装 `PUT`、`DELETE` 和其他 `HTTP` 方法。

`app.use(app.router)`：应用解析路由的规则，后面会有介绍。

`app.use(express.static(path.join(__dirname, 'public')))`：`connect` 内建的中间件，设置根目录下的 `public` 文件夹为存放 `image`、`css`、`js` 等静态文件的目录。

`if ('development' == app.get('env')) {app.use(express.errorHandler());}`：开发环境下的错误处理，输出错误信息。

`app.get('/', routes.index)`：路由控制器，如果用户访问 `'/'`（主页），则由 `routes.index` 来处理，`routes/index.js` 内容如下：

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

通过 `exports.index` 导出 `index` 函数接口，`app.get('/', routes.index)` 相当于：

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

`res.render('index', { title: 'Express' })`：调用 `ejs` 模板引擎解析 `views/index.ejs`（因为我们之前通过

`app.set('views', __dirname + '/views')` 设置了模板文件默认存储在 `views` 下），并传入一个对象作为参数，这个对象只有一个属性 `title`，它的值为字符串 `Express`，即用字符串 `Express` 替换 `views/index.ejs` 中所有 `title` 变量，后面我们将会了解更多关于模板引擎的内容。

```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

这段代码的意思是创建 `http` 服务器并监听 `3000` 端口，成功后在命令行中显示 `Express server listening on port 3000`，然后我们就可以通过在

浏览器输入 `localhost:3000` 来访问了。

这一小节我们学习了如何创建一个工程并启动它，了解了工程的大体结构，下一节我们将学习 `express` 的基本使用及路由控制。

路由控制

工作原理

上面提到过 `app.js` 中 `app.get('/', routes.index)` 可以用以下代码取代：

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

这段代码的意思是当访问主页时，调用 `ejs` 模板引擎，来渲染 `index.ejs` 模版文件（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里。

我们来作一些修改，以上代码实现了路由的功能，我们当然可以不要 `routes/index.js` 文件，把实现路由功能的代码都放在 `app.js` 里，但随着时间的推移 `app.js` 会变得臃肿难以维护，这也违背了代码模块化的思想，所以我们把实现路由功能的代码都放在 `routes/index.js` 里。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。我们不妨把路由控制器和实现路由功能的函数都放到 `index.js` 里，`app.js` 中只有一个总的路由接口。

打开 `app.js`，删除 `var user = require('./routes/user');`（我们这里用不到 `routes/user.js`，同时也删除这个文件）和删除

```
app.get('/', routes.index);
app.get('/users', user.list);
```

在 `app.js` 最后添加一行代码：

```
routes(app);
```

修改 `index.js` 如下：

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

现在，再运行你的 `app`，你会发现主页毫无二致。这其实是 `exports` 和 `module.exports` 的不同使用方法。详见 `核心模块` 章节。

路由规则

`express` 封装了多种 `http` 请求方式，我们主要只使用 `get` 和 `post` 两种。

`get` 和 `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 /search?q=tobi+ferret
req.query.q
// => "tobi ferret"

// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse
req.query.order
// => "desc"

req.query.shoe.color
// => "blue"

req.query.shoe.type
// => "converse"
```

`req.body`

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

req.body.user.email
// => "tobi@learnboost.com"

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

req.params

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

// GET /file/javascripts/jquery.js
req.params[0]
// => "javascripts/jquery.js"

**req.param(name)**

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

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

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

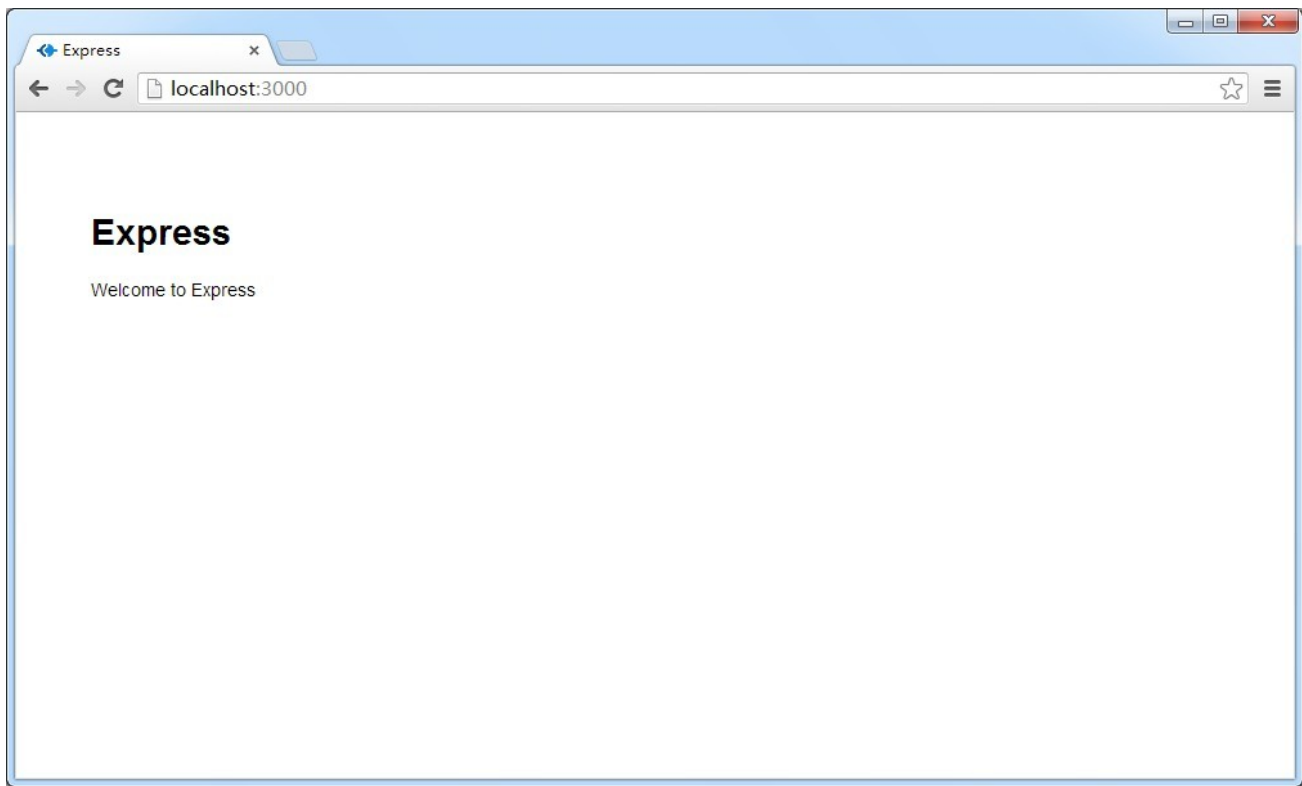
不难看出：

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

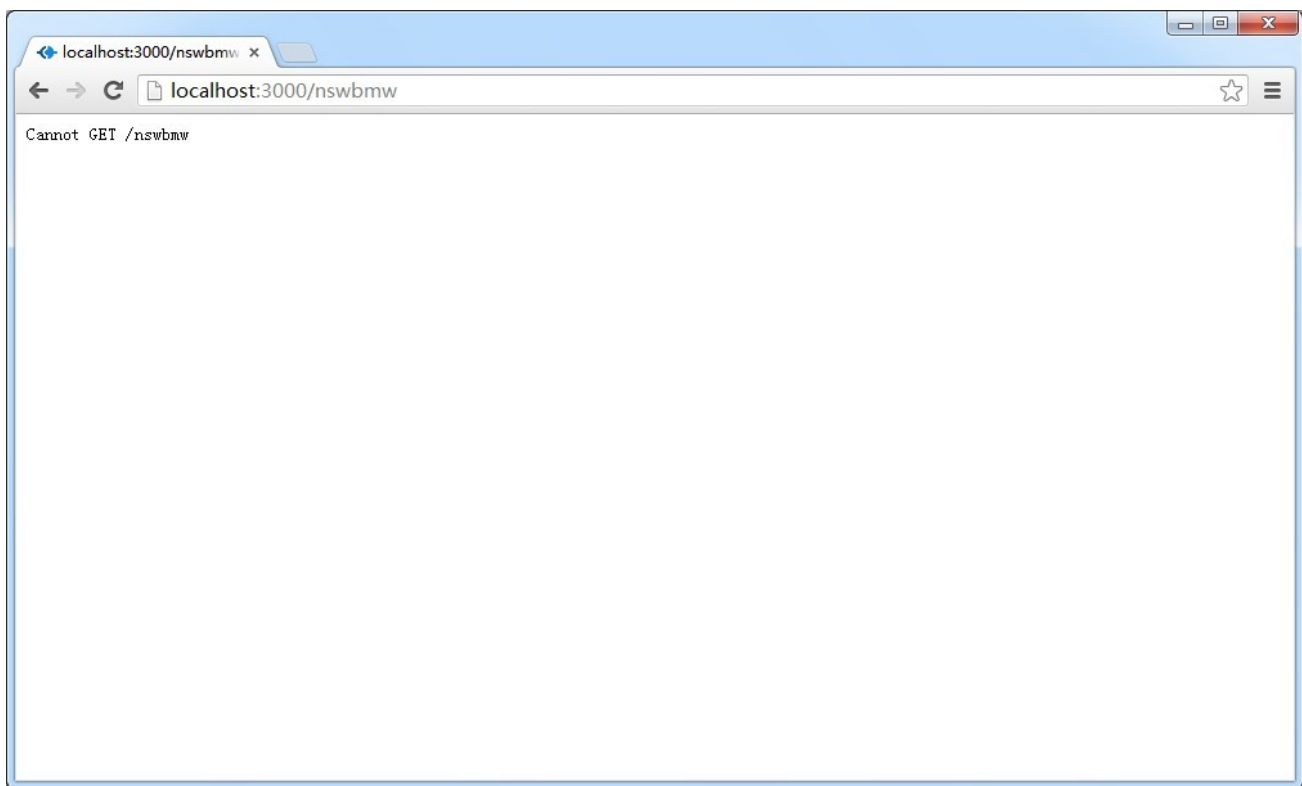
路径规则还支持正则表达式，更多请查阅：<http://expressjs.com/api.html>

添加路由规则

当我们访问 `localhost:3000` 时，会显示：



当我们访问 `localhost:3000/nswbmw` 这种不存在的页面时就会显示：



这是因为不存在 `/nswbmw` 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。

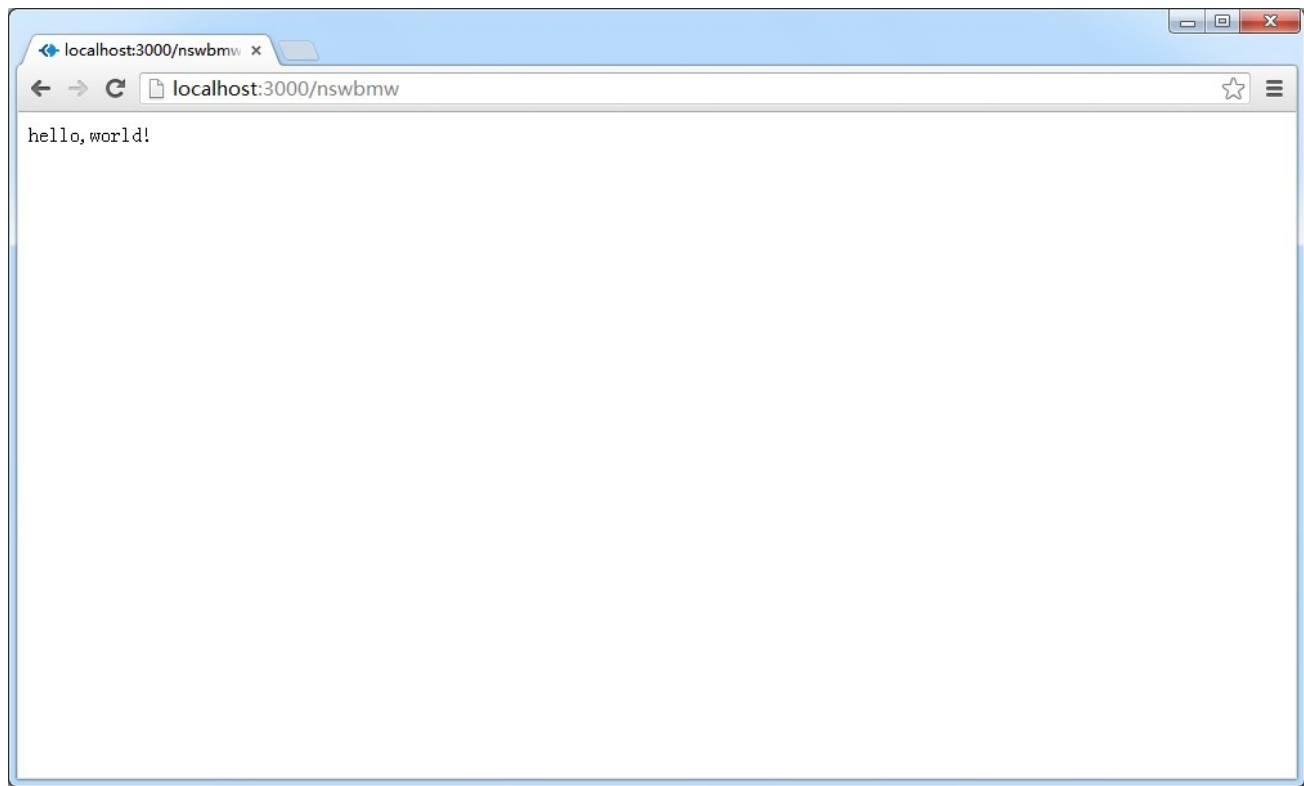
下面我们来添加这条路由规则，使得当访问 `localhost:3000/nswbmw` 时，页面显示 `hello,world!`

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

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

访问 `localhost:3000/nswbmw` 页面显示如下：



很简单吧？这一节我们学习了基本的路由规则及如何添加一条路由规则，下一节我们将学习模板引擎的知识。

模版引擎

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 **ejs**？

ejs 是模板引擎的一种，也是我们这个教程中使用的模板引擎，因为它十分简单，而且与 **express** 集成良好。

使用模板引擎

前面我们通过以下两行代码设置了模板引擎和页面模板的存储位置：

```
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
```

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 **.ejs** 可选。第二个参数是传递给模板的数据，用于模板翻译。

index.ejs

```
<!DOCTYPE html>
<html>
  <head>
```

```
<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为：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Express</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>Express</h1>
    <p>Welcome to Express</p>
  </body>
</html>
```

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

`ejs` 的标签系统非常简单，它只有以下3种标签。

- `<% 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` 为字符串时，两者没有区别。当 `code` 比如为 `<h1>hello</h1>` 时，`<%= code %>` 会原样输出 `<h1>hello</h1>`，而 `<%- code %>` 则会输出 H1 大的 `hello`。

`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>
```

我们可以用上述三种方式实现页面模板系统能实现的任何内容。

页面布局

Express 3.* 中我们不再使用 `layout.ejs` 进行页面布局，转而使用 `include` 来替代。`include` 的简单使用如下：

a.ejs

```
<%- include b %>
hello,world!
```



```
<%- include c %>
```

b.ejs

```
this is b
```

c.ejs

```
this is c
```

最终 a.ejs 会显示:

```
this is b
hello,world!
this is c
```

这一节我们学习了模版引擎的相关知识，下一节我们正式开始学习如何从头开始搭建一个多人博客。

搭建多人博客

功能分析

作为入门教程，我们要搭建的博客具有简单的允许多人注册、登录、发表文章、登出的功能。

设计目标

未登录：主页左侧导航显示 home、login、register，右侧显示已发表的文章、发表日期及作者。

登录后：主页左侧导航显示 home、post、logout，右侧显示已发表的文章、发表日期及作者。

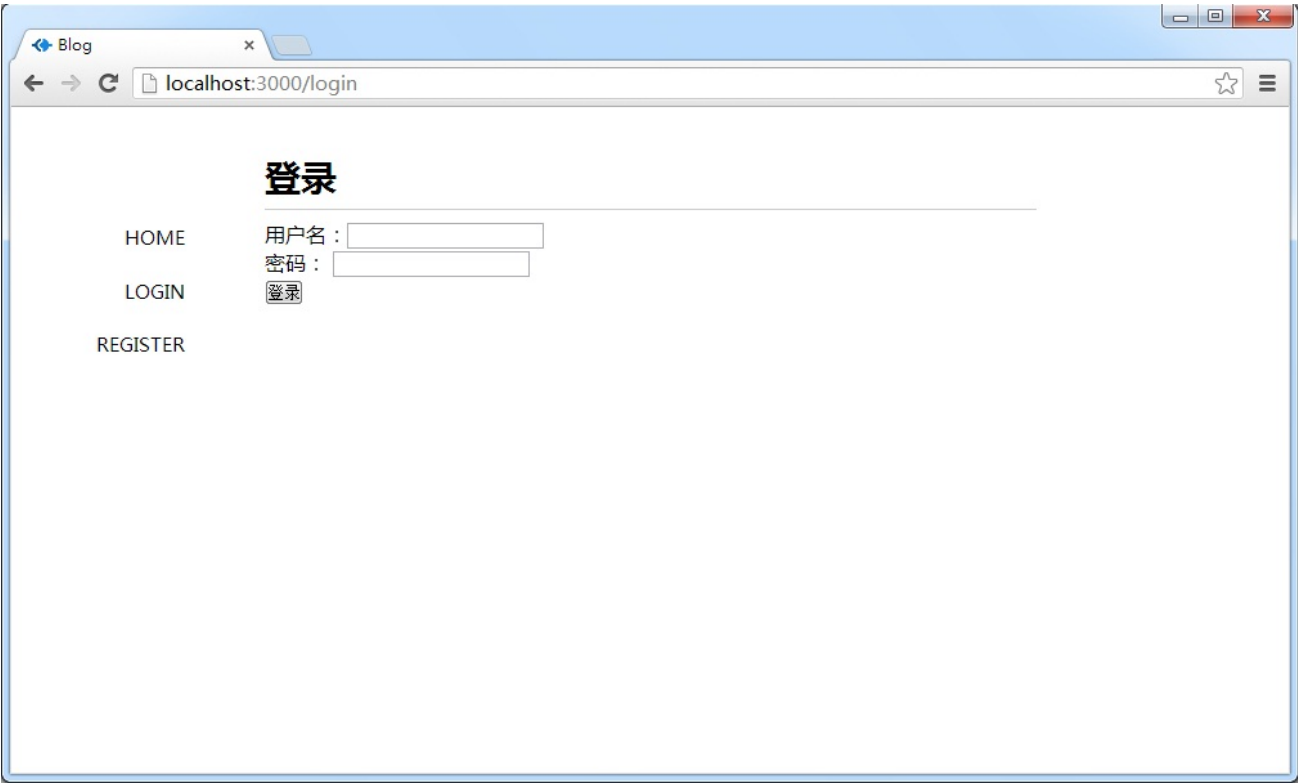
用户登录、注册、发表成功以及登出后都返回到主页。

用户登陆前：

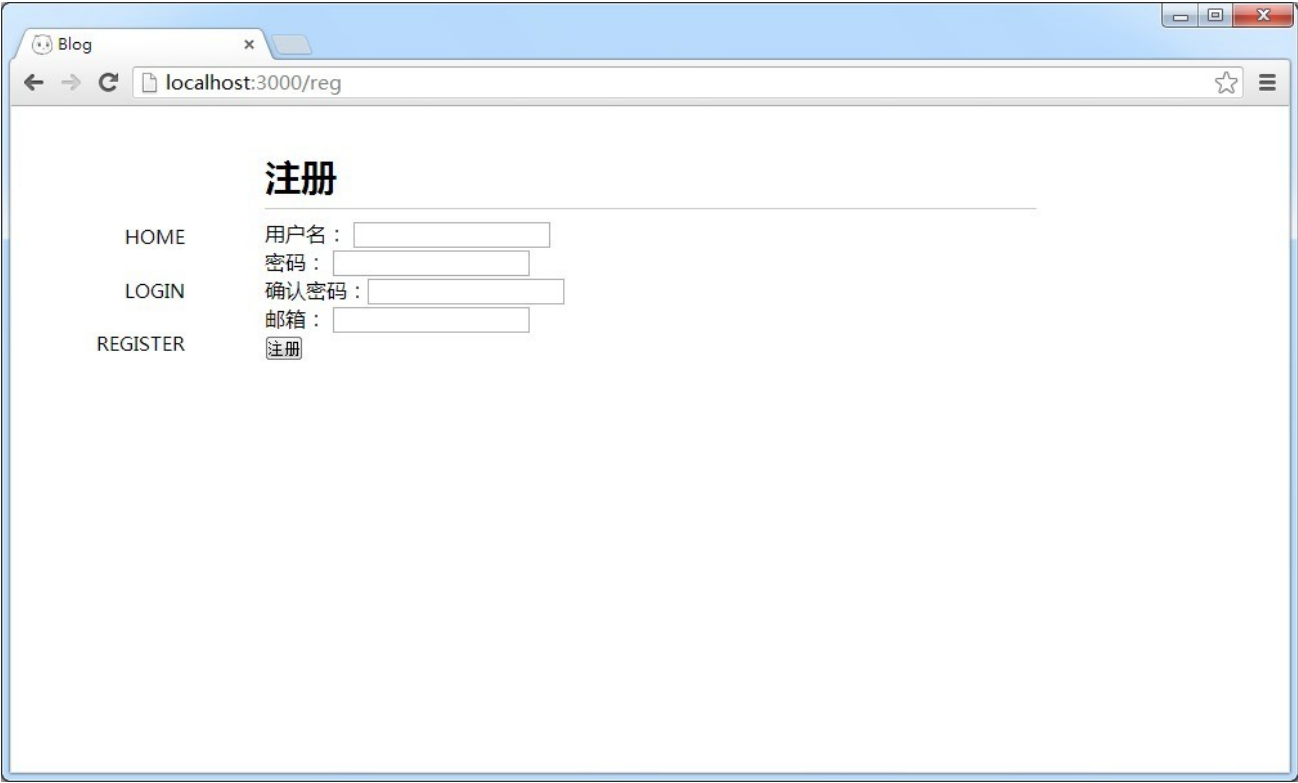
主页：



登录页：



注册页:

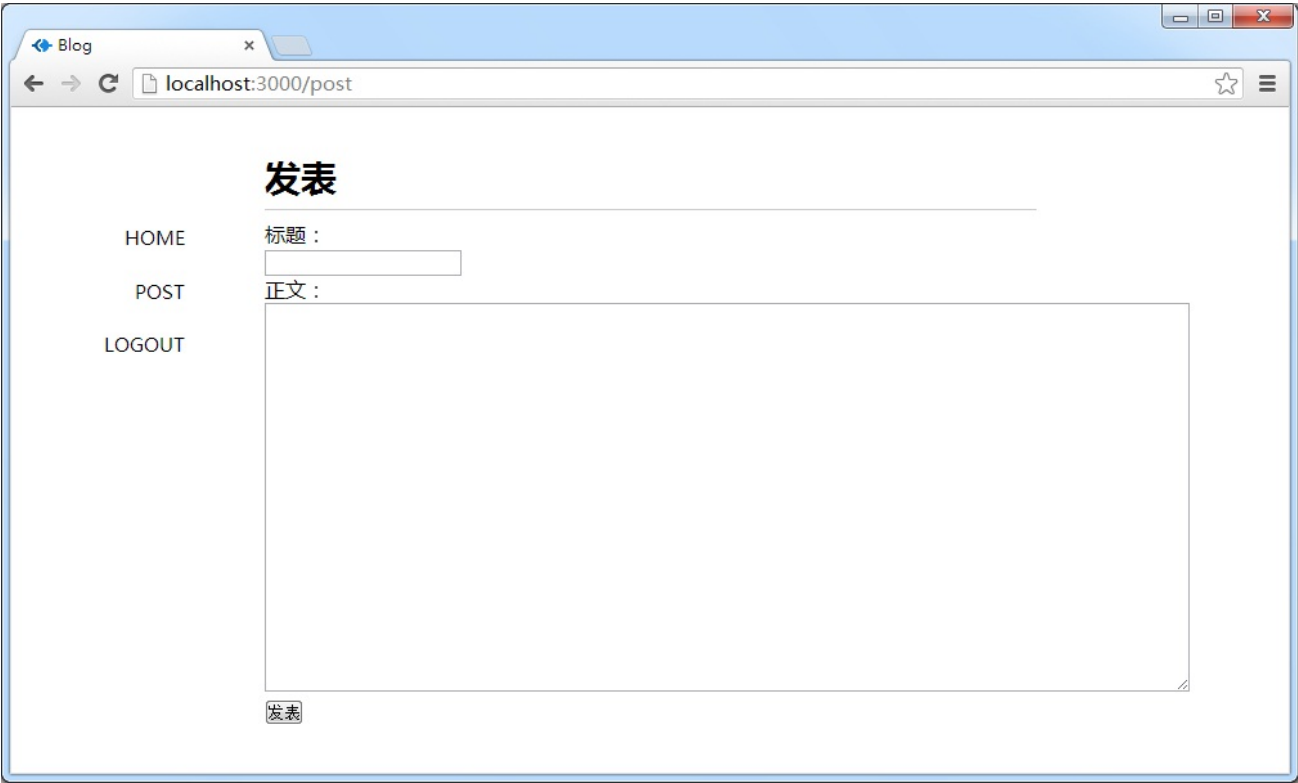


用户登录后:

主页:



发表页：



注意：没有登出页，当点击 LOGOUT 后，退出登陆并返回到主页。

路由规划

我们已经把设计的构想图贴出来了，接下来的任务就是写路由规划了。路由规划，或者说控制器规划是整个网站的骨架部分，因为它处于整个架构的枢纽位置，相当于各个接口之间的粘合剂，所以应该优先考虑。

根据构思的设计图，我们作以下路由规划：

```
/ : 首页
/login : 用户登录
/reg : 用户注册
/post : 发表文章
```

```
/logout : 登出
```

我们要求 `login` 和 `reg` 页只能是未登录的用户访问，而 `post` 页和 `logout` 只能是已登录的用户访问。左侧导航列表则针对已登录和未登录的用户显示不同的内容。

修改 `index.js` 如下：

```
module.exports = function(app) {
  app.get('/', function (req, res) {
    res.render('index', { title: '主页' });
  });
  app.get('/reg', function (req, res) {
    res.render('reg', { title: '注册' });
  });
  app.post('/reg', function (req, res) {
  });
  app.get('/login', function (req, res) {
    res.render('login', { title: '登录' });
  });
  app.post('/login', function (req, res) {
  });
  app.get('/post', function (req, res) {
    res.render('post', { title: '发表' });
  });
  app.post('/post', function (req, res) {
  });
  app.get('/logout', function (req, res) {
  });
};
```

如何针对已登录和未登录的用户显示不同的内容呢？或者说如何判断用户是否已经登陆了呢？进一步说如何记住用户的登录状态呢？我们通过引入会话机制，来记录用户登录状态，还要访问数据库来保存和读取用户信息。下一节我们将学习如何使用数据库。

使用数据库

MongoDB简介

MongoDB 是一个对象数据库，它没有表、行等概念，也没有固定的模式和结构，所有的数据以文档的形式存储。所谓文档就是一个关联数组式的对象，它的内部由属性组成，一个属性对应的值可能是一个数、字符串、日期、数组，甚至是一个嵌套的文档。

——《Node.js开发指南》

下面是一个 MongoDB 文档的示例：

```
{
  "_id" : ObjectId( "4f7fe8432b4a1077a7c551e8" ),
  "name" : "nswbmw",
  "age" : 22,
  "email" : [ "xxx@126.com", "xxx@gmail.com" ],
  "family" : {
    "mother" : { ... },
    "father" : { ... },
    "sister" : { ... },
    "address" : "earth"
  }
}
```

更多有关 MongoDB 的知识请参考《mongodb权威指南》或查阅：<http://www.mongodb.org/>

安装MongoDB

安装 `mongodb` 很简单，去官网（<http://www.mongodb.org/downloads>）下载最新版的 `mongodb`（目前为 `v2.4.6`），解压到 `D` 盘并把文件夹重命名为 `mongodb`，并在 `mongodb` 文件夹里新建 `blog` 文件夹作为我们博客内容的存储目录。打开 `cmd`，切换到 `d:\mongodb\bin` 目录下，然后输入 `mongod -dbpath d:\mongodb\blog` 设置 `blog` 文件夹作为我们工程的存储目录并启动。为了方便以后使用数据库，我们在桌面上新建 `启动mongodb.bat`，并写入 `d:\mongodb\bin\mongod.exe -dbpath d:\mongodb\blog`，这样我们以后只需运行桌面上的 `启动mongodb.bat` 就可启动数据库了。

连接MongoDB

数据库虽然安装并启动成功了，但我们需要连接数据库后才能使用数据库。怎么才能在 `Node.js` 中使用 `MongoDb` 呢？我们使用官方提供的 `node-mongodb-native` 模块，打开 `package.json`，在 `dependencies` 中添加一行代码：

```
{
  "name": "blog",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.3.8",
    "ejs": "*",
    "mongodb": "*"
  }
}
```

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

接下来在工程的根目录中创建 `settings.js` 文件，用于保存该项目的配置信息，比如数据库的连接信息。我们将数据库命名为 `blog`，数据库服务器在本地，因此 `settings.js` 文件的内容如下：

```
module.exports = {
  cookieSecret: 'myblog',
  db: 'blog',
  host: 'localhost'
};
```

其中 `db` 是数据库的名称，`host` 是数据库的地址。`cookieSecret` 用于 `Cookie` 加密与数据库无关，我们留作后用。

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,
    Connection = require('mongodb').Connection,
    Server = require('mongodb').Server;
module.exports = new Db(settings.db, new Server(settings.host, Connection.DEFAULT_PORT, {}));
```

其中 `new Db(settings.db, new Server(settings.host, Connection.DEFAULT_PORT, {}))`；通过设置数据库名、数据库地址和数据库端口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()` 本模块来对数据库进行读写了。

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，我们首先要获取一个叫做 `connect-mongo`（参见 <https://github.com/kcbanner/connect-mongo>）的模块，在 `package.json` 中添加一行代码：

```
{
  "name": "blog",
  "version": "0.0.1",
  "private": true,
```

```
"scripts": {
  "start": "node app.js"
},
"dependencies": {
  "express": "3.3.8",
  "ejs": "*",
  "mongodb": "*",
  "connect-mongo": "*"
}
}
```

运行 `npm install` 安装模块。然后打开 `app.js`，在 `var path = require('path');` 后添加以下代码：

```
var MongoStore = require('connect-mongo')(express);
var settings = require('./settings');
```

在 `app.use(express.methodOverride());` 后添加：

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

其中 `express.cookieParser()` 是 Cookie 解析的中间件。`express.session()` 则提供会话支持，`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，以维护用户相关的信息。

注册和登陆

我们已经准备好了数据库访问和会话存储的相关信息，接下来我们完成用户注册和登录功能。

页面设计

首先我们来完成主页、登录页和注册页的页面设计。

修改 `views/index.ejs` 如下：

```
<%- include header %>
这是主页
<%- include footer %>
```

在 `views` 下新建 `header.ejs`，添加如下代码：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Blog</title>
<link rel="stylesheet" href="/stylesheets/style.css">
</head>
<body>

<header>
<h1><%= title %></h1>
</header>

<nav>
<span><a title="主页" href="/">home</a></span>
<span><a title="登录" href="/login">login</a></span>
<span><a title="注册" href="/reg">register</a></span>
</nav>

<article>
```

在 `views` 下新建 `footer.ejs`，添加如下代码：

```
</article>
</body>
</html>
```

修改 `public/stylesheets/style.css` 如下：

```
/* inspired by http://yihui.name/cn/ */
*{padding:0;margin:0;}
body{width:600px;margin:2em auto;padding:0 2em;font-size:14px;font-family:"Microsoft YaHei";}
p{line-height:24px;margin:1em 0;}
header{padding:.5em 0;border-bottom:1px solid #cccccc;}
nav{position:fixed;left:12em;font-family:"Microsoft YaHei";font-size:1.1em;text-transform:uppercase;width:9em;text-align:right;}
nav a{display:block;text-decoration:none;padding:.7em 1em;color:#000000;}
nav a:hover{background-color:#ff0000;color:#f9f9f9;-webkit-transition:color .2s linear;}
article{font-size:16px;padding-top:.5em;}
article a{color:#dd0000;text-decoration:none;}
article a:hover{color:#333333;text-decoration:underline;}
.info{font-size:14px;}
```

主页显示如下：



接下来在 `views` 下新建 `login.ejs`，内容如下：

```
<%- include header %>
<form method="post">
  用户名: <input type="text" name="name"/><br />
  密码:   <input type="password" name="password"/><br />
         <input type="submit" value="登录"/>
</form>
<%- include footer %>
```

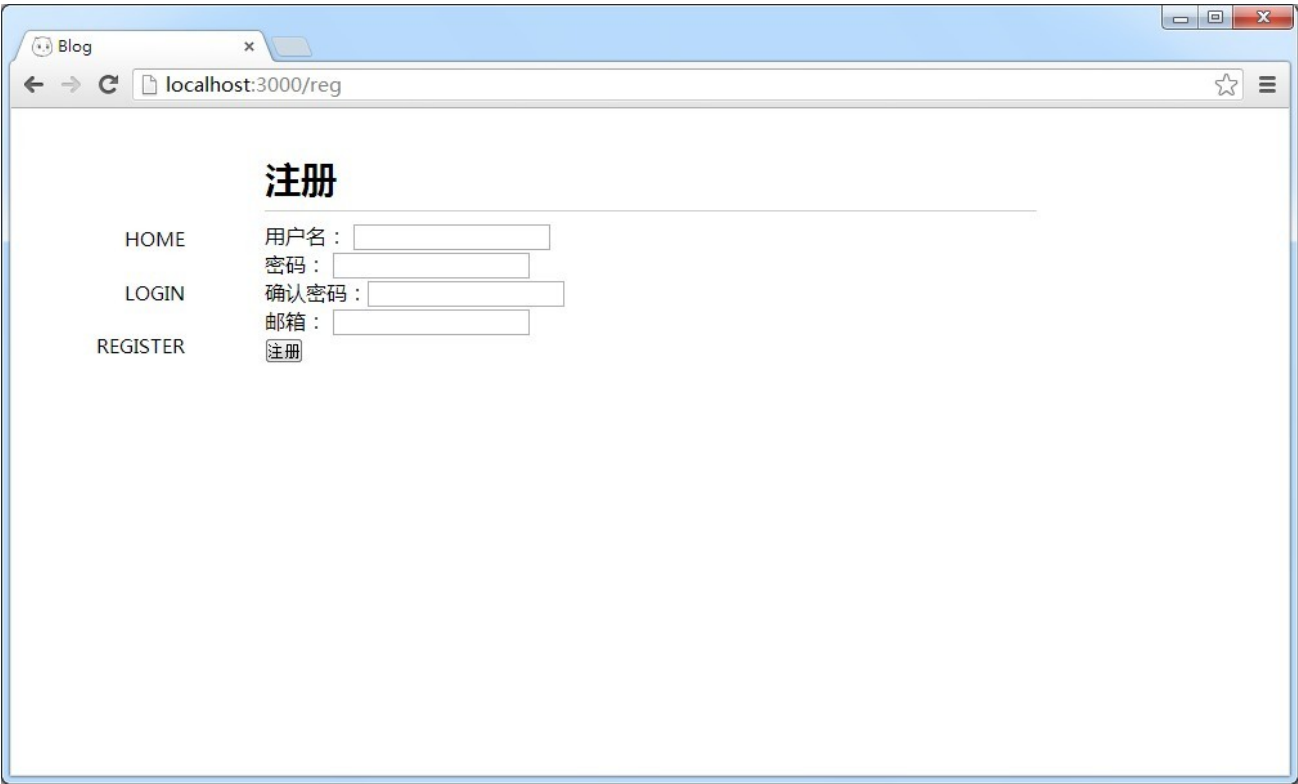
登录页面显示如下：



在 views 下新建 reg.ejs，内容如下：

```
<%- include header %>
<form method="post">
  用户名: <input type="text" name="name"/><br />
  密码: <input type="password" name="password"/><br />
  确认密码: <input type="password" name="password-repeat"/><br />
  邮箱: <input type="email" name="email"/><br />
        <input type="submit" value="注册"/>
</form>
<%- include footer %>
```

注册页面显示如下：



至此，未登录时的主页、注册页、登录页都已经完成。

在桌面新建 `启动app.bat` 并写入：

```
node d:\blog\app
```

以后我们就可以通过依次打开 `启动mongodb.bat` 和 `启动app.bat` 来启动我们的博客了。

此时运行 `启动mongodb.bat` 和 `启动app.bat`，查看一下效果吧。

页面通知

接下来我们实现用户注册和登陆，在这之前我们需要引入 **flash** 模块来实现页面的通知和错误信息显示的功能。

什么是 **flash**？

我们所说的 **flash** 即 **connect-flash** 模块（详见 <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。这个中间件是从 **Express 2.x** 提取出来的，**Express 3.x** 不再支持，但是通过 **connect-flash** 外部模块可以实现这个功能。

在 **package.json** 添加一行代码：

```
{
  "name": "blog",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.3.8",
    "ejs": "*",
    "mongodb": "*",
    "connect-mongo": "*",
    "connect-flash": "*"
  }
}
```

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

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

在 `app.set('view engine', 'ejs');` 后添加：

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

现在我们就可以使用 **flash** 功能了。

注册响应

前面我们已经完成了注册页，当然现在点击注册是没有效果的，因为我们还没有实现处理 **POST** 请求的功能，下面就来实现它。

在 **models** 文件夹下新建 **user.js**，添加如下代码：

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

function User(user) {
  this.name = user.name;
  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 = {
    name: this.name,
    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(); //关闭数据库
      callback(null, user[0]); //成功! err 为 null, 并返回存储后的文档
    });
  });
});
});

//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(); //关闭数据库
        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({
        name: name
      }, function(err, user){
        mongodb.close(); //关闭数据库
        if (user) {
          return callback(null, user); //成功! 返回查询的用户信息
        }
        callback(err); //失败! 返回 err 信息
      });
    });
  });
});
};

```

我们通过 `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 `Node.js` 的一个核心模块, 我们后面用它生成散列值来加密密码。

修改 `app.post('/reg')` 如下:

```

app.post('/reg', function (req, res) {
  var name = req.body.name,
      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');
  }
  //生成密码的 md5 值
  var md5 = crypto.createHash('md5'),
      password = md5.update(req.body.password).digest('hex');
  var newUser = new User({
    name: req.body.name,

```

```
password: password,
email: req.body.email
});
//检查用户名是否已经存在
User.get(newUser.name, function (err, user) {
  if (user) {
    req.flash('error', '用户已存在!');
    return res.redirect('/reg');//用户名存在则返回注册页
  }
  //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');
    }
    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，例如我们要访问用户传递的 `name="password"` 域的值，只需访问 `req.body['password']` 或 `req.body.password` 即可。
- **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，启动数据库和 `app.js`，在浏览器输入 `localhost:3000` 注册试试吧！注册成功后显示如下：



这样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，切换到 `d:\mongodb\bin\mongo`（注意要保持数据库连接），输入：

```
D:\mongodb\bin>mongo
MongoDB shell version: 2.4.6
connecting to: test
Server has startup warnings:
Wed Sep 04 20:13:19.130 [initandlisten]
Wed Sep 04 20:13:19.131 [initandlisten] ** NOTE: This is a 32 bit MongoDB binary
.
Wed Sep 04 20:13:19.131 [initandlisten] **      32 bit builds are limited to less
ss than 2GB of data (or less with --journal).
Wed Sep 04 20:13:19.131 [initandlisten] **      Note that journaling defaults to
o off for 32 bit and is currently off.
Wed Sep 04 20:13:19.131 [initandlisten] **      See http://dochub.mongodb.org/core/32bit
Wed Sep 04 20:13:19.132 [initandlisten]
> use blog
switched to db blog
> db.users.find(<
< "name" : "nswbmw", "password" : "d41d8cd98f00b204e9800998ecf8427e", "email" :
"qxqzk@126.com", "_id" : ObjectId<"522722f8dd0bae101d000001"> }
>
```

可以看到，用户信息已经成功存入数据库。但这还不是我们想要的效果，我们想要的效果是当注册成功返回主页时，左侧导航显示 HOME、POST、LOGOUT，右侧显示 注册成功！ 字样。下面我们来实现它，添加 flash 功能。

修改 header.ejs，将 `<nav></nav>` 修改如下：

```
<nav>
<span><a title="主页" href="/">home</a></span>
<% if (user) { %>
  <span><a title="发表" href="/post">post</a></span>
  <span><a title="登出" href="/logout">logout</a></span>
<% } else { %>
  <span><a title="登录" href="/login">login</a></span>
  <span><a title="注册" href="/reg">register</a></span>
<% } %>
</nav>
```

在 `<article>` 后添加如下代码：

```
<% if (success) { %>
  <div><%= success %></div>
<% } %>
<% if (error) { %>
  <div><%= error %> </div>
<% } %>
```

修改 index.js，将 `app.get('/')` 修改如下：

```
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 index.js，`app.get('reg')` 修改如下：

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

现在运行我们的博客，注册成功后显示如下：



我们通过对 `session` 的使用实现了对用户状态的检测，再根据不同的用户状态显示不同的导航信息。

简单解释一下流程：用户在注册成功后，把用户信息存入 `session`，同时页面跳转到主页显示 注册成功！。然后把 `session` 中的用户信息赋给变量 `user`，在渲染 `ejs` 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导航信息。

`success: req.flash('success').toString()` 用来获取 `req.flash('success', '注册成功!');` 的信息并赋值给变量 `success`，
`error: req.flash('error').toString()` 用来获取 `req.flash('error', '用户不存在!');` 的信息并赋值给变量 `error`，然后我们在渲染 `ejs` 模版文件时传递这两个变量来检测并显示通知。

登录与登出响应

现在我们来实现登陆的功能。

打开 `index.js` 将 `app.post('/login')` 修改如下：

```
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.name, function (err, user) {
    if (!user) {
      req.flash('error', '用户不存在!');
      return res.redirect('/login');//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');//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('/');
  });
});
```

接下来我们实现登出响应。修改 `app.get('/logout')` 如下：

```
app.get('/logout', function (req, res) {
  req.session.user = null;
  req.flash('success', '登出成功!');
  res.redirect('/');
});
```

注意：通过把 req.session.user 赋值 null 丢掉 session 中用户的信息，实现用户的退出。

登录后页面显示如下：



登出后页面显示如下：



至此，我们实现了用户注册与登陆的功能，并且根据用户登录状态显示不同的导航。

页面权限控制

我们虽然已经完成了用户注册与登陆的功能，但并不能阻止比如已经登陆的用户访问 localhost:3000/reg 页面（读者可亲自尝试下）。为此，我们需要为页面设置访问权限。即注册和登陆页面应该阻止已登陆的用户访问，登出及后面我们将要实现的发表页只对已登录的用户开放。如何实现页面权限的控制呢？我们可以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。我们添加 `checkNotLogin` 和 `checkLogin` 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');
  }
  next();
}

function checkNotLogin(req, res, next) {
  if (req.session.user) {
    req.flash('error', '已登录!');
    res.redirect('back');
  }
  next();
}
```

注意：`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上一个页面。

我们修改 `index.js` 后最终代码如下：

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

module.exports = function(app) {
  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()
    });
  });

  app.get('/reg', checkNotLogin);
  app.get('/reg', function (req, res) {
    res.render('reg', {
      title: '注册',
      user: req.session.user,
      success: req.flash('success').toString(),
      error: req.flash('error').toString()
    });
  });

  app.post('/reg', checkNotLogin);
  app.post('/reg', function (req, res) {
    var name = req.body.name,
        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');
    }
    //生成密码的 md5 值
    var md5 = crypto.createHash('md5'),
        password = md5.update(req.body.password).digest('hex');
    var newUser = new User({
      name: req.body.name,
      password: password,
      email: req.body.email
    });
    //检查用户名是否已经存在
    User.get(newUser.name, function (err, user) {
      if (user) {
        req.flash('error', '用户已存在!');
        return res.redirect('/reg');//用户名存在则返回注册页
      }
      //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');
        }
      });
    });
  });
}
```

```
    }
    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
  });
});

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

app.post('/login', checkNotLogin);
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.name, function (err, user) {
    if (!user) {
      req.flash('error', '用户不存在!');
      return res.redirect('/login');//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');//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('/');
  });
});

app.get('/post', checkLogin);
app.get('/post', function (req, res) {
  res.render('post', {
    title: '发表',
    user: req.session.user,
    success: req.flash('success').toString(),
    error: req.flash('error').toString()
  });
});

app.post('/post', checkLogin);
app.post('/post', function (req, res) {
});

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

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

function checkNotLogin(req, res, next) {
  if (req.session.user) {
    req.flash('error', '已登录!');
    res.redirect('back');
  }
}
```



```
    }
    next();
  }
};
```

注意：为了维护用户状态和 **flash** 通知功能的使用，我们在 `app.get('/')` 和 `app.get('/reg')` 和 `app.get('/login')` 和 `app.get('/post')` 里添加了以下代码：

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

发表文章

现在我们的博客已经具备了用户注册、登陆、页面权限控制的功能，接下来我们完成博客最核心的部分——发表文章。在这一节，我们将会实现发表文章的功能，完成整个博客的设计。

页面设计

我们先来完成发表页的页面设计。在 **views** 文件夹下新建 **post.ejs**，添加如下代码：

```
<%- include header %>
<form method="post">
  标题: <br />
  <input type="text" name="title" /><br />
  正文: <br />
  <textarea name="post" rows="20" cols="100"></textarea><br />
  <input type="submit" value="发表" />
</form>
<%- include footer %>
```

文章模型

仿照用户模型，我们将文章模型命名为 **Post** 对象，它拥有与 **User** 相似的接口，分别是 **Post.get** 和 **Post.prototype.save**。**Post.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。**Post.prototype.save** 是 **Post** 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 **post.js**，添加如下代码：

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

function Post(name, title, post) {
  this.name = name;
  this.title= title;
  this.post = post;
}

module.exports = Post;

//存储一篇文章及其相关信息
Post.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.getMinute()
  }
  //要存入数据库的文档
  var post = {
    name: this.name,
    time: time,
    title: this.title,
    post: this.post
  };
  //打开数据库
```

```
mongodb.open(function (err, db) {
  if (err) {
    return callback(err);
  }
  //读取 posts 集合
  db.collection('posts', function (err, collection) {
    if (err) {
      mongodb.close();
      return callback(err);
    }
    //将文档插入 posts 集合
    collection.insert(post, {
      safe: true
    }, function (err, post) {
      mongodb.close();
      callback(null);
    });
  });
});

//读取文章及其相关信息
Post.get = function(name, callback) {
  //打开数据库
  mongodb.open(function (err, db) {
    if (err) {
      return callback(err);
    }
    //读取 posts 集合
    db.collection('posts', function(err, collection) {
      if (err) {
        mongodb.close();
        return callback(err);
      }
      var query = {};
      if (name) {
        query.name = name;
      }
      //根据 query 对象查询文章
      collection.find(query).sort({
        time: -1
      }).toArray(function (err, docs) {
        mongodb.close();
        if (err) {
          return callback(err);//失败! 返回 err
        }
        callback(null, docs);//成功! 以数组形式返回查询的结果
      });
    });
  });
};
```

发表响应

接下来我们给发表文章注册响应，打开 `index.js`，在 `User = require('../models/user.js')` 后添加一行代码：

```
Post = require('../models/post.js');
```

修改 `app.post('/post')` 如下：

```
app.post('/post', checkLogin);
app.post('/post', function (req, res) {
  var currentUser = req.session.user,
    post = new Post(currentUser.name, req.body.title, req.body.post);
  post.save(function (err) {
    if (err) {
      req.flash('error', err);
      return res.redirect('/');
    }
    req.flash('success', '发布成功!');
    res.redirect('/');
  });
});
```

```
});  
});
```

最后，我们修改 `index.ejs`，让主页右侧显示发表过的文章及其相关信息。

打开 `index.ejs`，修改如下：

```
<%- include header %>  
<% posts.forEach(function (post, index) { %>  
  <p><h2><a href="#"><%= post.title %></a></h2></p>  
  <p class="info">  
    作者: <a href="#"><%= post.name %></a> |  
    日期: <%= post.time.minute %>  
  </p>  
  <p><%= post.post %></p>  
<% }) %>  
<%- include footer %>
```

打开 `index.js`，修改 `app.get('/')` 如下：

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

至此，我们的博客就建成了。

运行 `mongodb.bat` 和 `启动app.bat`，发表一篇博文如下：



此时，查看一下数据库，如图所示：

```
D:\mongodb\bin>mongo
MongoDB shell version: 2.4.6
connecting to: test
Server has startup warnings:
Thu Sep 05 10:08:28.526 [initandlisten]
Thu Sep 05 10:08:28.526 [initandlisten] ** NOTE: This is a 32 bit MongoDB binary
.
Thu Sep 05 10:08:28.527 [initandlisten] **      32 bit builds are limited to le
ss than 2GB of data (or less with --journal).
Thu Sep 05 10:08:28.527 [initandlisten] **      Note that journaling defaults t
o off for 32 bit and is currently off.
Thu Sep 05 10:08:28.527 [initandlisten] **      See http://dochub.mongodb.org/c
ore/32bit
Thu Sep 05 10:08:28.527 [initandlisten]
> use blog
switched to db blog
> db.posts.find(<
< { "name" : "nsubmw", "time" : < "date" : ISODate("2013-09-05T02:09:19.027Z"), "y
ear" : 2013, "month" : "2013-9", "day" : "2013-9-5", "minute" : "2013-9-5 10:9"
>, "title" : "海贼王", "post" : "《ONE PIECE》（中文译名：海贼王、航海王），是一
部连载中的日本少年漫画作品，作者为尾田荣一郎。该作于1997年起在日本漫画杂志《周刊
少年Jump》定期连载，另外有同名的电视动画、海贼王剧场版和电子游戏等周边媒体产品。
《ONE PIECE》漫画单行本在日本以外的亦已有30多个翻译版本发行，发行量在日本本土突
破3亿部，是日本图书出版史上发行量最高的作品。", "_id" : ObjectId("5227e7cfffca516
5815000002") >
>
```

Tips: Robomongo 是一款开源的跨平台的 MongoDB 管理工具，支持 shell 查询，使用起来非常简单，使用它我们就不要在命令行中敲命令查询数据库了。

Last edited by nswbmw, 11 days ago