**（1）如何封装一个模块；**

首先，我们建立一个js文件，例如命名为test.js；

然后在里面写一个函数，函数名任意；

然后通过exports.变量名，将函数赋值给这个变量；

如代码：

function test(){ //请注意这个函数名  
 console.log("1");  
}  
exports.testBegin = test; //等号后面的test，指的是上面的函数名。等号前面的testBegin，是调用时的函数名（注意区别）

这个test.js的文件就写完了，这是一个模块，他的效果是调用该函数后，输出1；

另外，不要问我这个exports是什么，我暂时也不知道。

**（2）如何调用一个模块**

在封装模块的前提下，我们新建一个文件，例如a.js，来调用之前封装的test.js模块；

方法是：

var test = require("./test"); //调用该模块  
test.testBegin(); //调用该模块的方法（注意方法名是test.js中exports后面的变量名）；

注意调用时的方法名，并非是test.js中的函数名test；

**（3）如何执行一个node.js的文件**

我们现在需要执行a.js这个调用了封装模块的文件了，执行他的方法，和执行普通的node.js的文件的方法并没有什么不同；

在a.js这个文件的目录下，打开命令行。windows是shift＋鼠标右键；

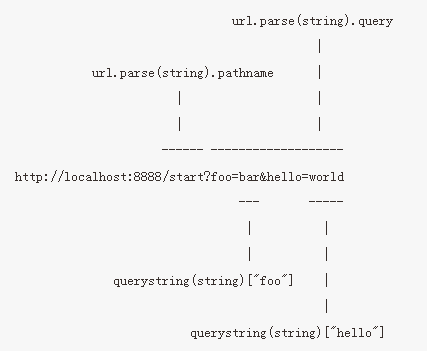
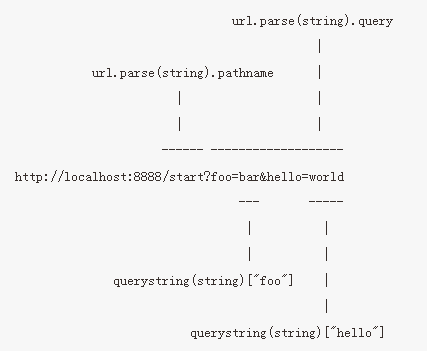
像下面这样输入即可：



然后将输出1，感觉执行执行python文件那样



**（4）获得访问者想要访问的路径**



首先上图，其中pathname指的是路径；而query指的是请求之类的东西（暂时不关心）；

假如我们想要获得pathname（即字符串“start”），我们该怎么办呢？

**方法：**

①首先，需要获取访问者需要访问的路径，方法是通过http的方法createServer

具体来说：

我们先require一个http模块：

var http = require("http");

然后调用其方法，创建一个服务器：

**http**.createServer(onRequest).listen(8888);

注意，这个方法监听的是8888端口，其参数是一个函数；

然后我们书写这个函数的内容：

function onRequest(request, response) {  
 console.log(request.url);  
}

这个函数的第一个参数request是用户访问的一些东西，我们需要的是其url变量；假设我们访问地址是这样的：

http://127.0.0.1:8888/index/loading?start

请注意红色部分，那么服务器在console.log输出的是

/index/loading?**start**

②我们下来要对这个url做点什么了；

虽然直接对上面那个参数进行操纵也可以，但这太笨了，让我们来require一个新的模块url

var url = require("url");

③调用这个新模块的方法parse，将第一步获得的url地址，作为其参数，再调用其方法pathname，就可以获得我们想要的东西了；

具体而言，需要这么一段代码：

var pathname = url.parse(request.url).pathname;

这个pathname就是我们需要的东西了；

把所有代码综合起来，是这样的：

var http = require("http");  
var url = require("url");  
function onRequest(request, response) {  
 var pathname = url.parse(request.url).pathname;  
 console.log("Request for " + pathname + " recived.");  
 response.writeHead(200, {"Content-type": "text/plain"});  
 response.write("Hello word!");  
 response.end();  
}  
http.createServer(onRequest).listen(80);  
console.log("Server has started!");

然后运行之：



我们随便访问一个本地的地址：

http://127.0.0.1:8888/index/loading?start



发现，/index/loding这部分被正常的显示出来了。

另外，favicon.ico是该网站的图标，据说很多服务器是会默认读取的；

假如我们直接访问：http://127.0.0.1:8888，显示是这样的



**（5）关于http**

我们之前有了这么一段代码（已delete掉无关部分）：

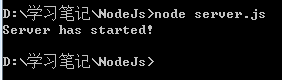
var http = require("http");

function onRequest(request, response) {  
 response.writeHead(200, {"Content-type": "text/plain"});  
 response.write("Hello word!");  
 response.end();  
}  
http.createServer(onRequest).listen(8888);

其中，http调用了NodeJs自带的一个模块“http”，而这个模块是一个服务器模块；

而createServer是这个模块的方法，效果是返回一个对象，而这个对象有一个listen的方法；

当然，由于你和我，也许都是新手，因此我们尚没有搞清楚这个到底是怎么运作的；但如果有一定经验的话，可以猜到，这个listen是监听，监听的是8888端口；如果去掉这部分会怎么样？经过测试，服务器并没有运行，如图：



而onRequest函数呢，从名字可以猜到，这个函数的效果是“当请求的时候做些什么”，我们来看其函数：

function onRequest(request, response) {  
 response.writeHead(200);  
 response.write("Hello word!");  
 response.end();  
**}**

这个函数有两个参数：request和response，顾名思义，请求和响应。

request请求指用户访问的时候，相关的信息；

response指服务器的响应，会做些什么；这个函数的三个方法，都是response相关的；

第一个：response.writeHead方法，具体来说，就是HTTP响应报文的头行；

这个指HTTP通信的响应报文中的头部分，如果没有基础的话，暂时不需要关心，写200即可，这里的200，是约定俗称的请求成功时的返回；

具体搜：

http://blog.csdn.net/fh13760184/article/details/6840696

**response.writeHead(statusCode, [reasonPhrase], [headers])**

第二个：response.write方法，具体来说，是HTTP响应报文；

简单理解，负责发送正文中的一部分，可以同时发送多个不一样的；

在上面搜：

### response.write(chunk, encoding='utf8')

第三个：response.end方法，具体来说每一个响应的结束，必须要调用这个方法，然后服务器会认为这条信息已经发送完毕了。

综合效果来说，当我们访问：http://127.0.0.1:8888/时，浏览器会出现：

Hello word!

这样的文字。

**（6）函数式编程**

所谓函数式编程，简单的理解，就是将函数作为参数传递；

目前有四个文件：

index.js

var server = require("./server"); //调用该模块  
var router = require("./router");  
var requestHandlers = require("./requestHandlers");  
  
var handle = {};  
handle["/"] = requestHandlers.start;  
handle["/start"] = requestHandlers.start;  
handle["/upload"] = requestHandlers.upload;  
  
server.start(router.route, handle); //调用该模块的方法（注意方法名是test.js中exports后面的变量名）

server.js

var http = require("http");  
var url = require("url");  
function start(route, handle) {  
 var count = 0;  
  
 function onRequest(request, response) {  
 **console**.log(count++ + "#:");  
 var pathname = url.parse(request.**url**).**pathname**;  
 **console**.log("Request for " + pathname + " recived.");  
  
 route(handle, pathname); //来源于上面的start的参数  
  
 response.writeHead(200);  
 response.write("Hello word!");  
 response.end();  
 }  
  
 http.createServer(onRequest).listen(8888);  
 **console**.log("Server has started!");  
}  
exports.start = start;

router.js

function route(handle, pathname) {  
 console.log("About to route a request for " + pathname);  
 if (typeof handle[pathname] === 'function') {  
 handle[pathname]();  
 } else {  
 console.log("No request handler found for " + pathname);  
 }  
}  
  
exports.route = route;

requestHandler.js

function start() {  
 console.log("Request handler 'start' was called");  
}  
  
function upload() {  
 console.log("Request handler 'upload' was called");  
}  
  
exports.start = start;  
exports.upload = upload;

**分析：**

①首先看index.js，他调用了3个模块，然后声明了一个空的对象，然后给这个对象声明了几个不同的变量（准确的说是函数变量），他们分别对应requestHandlers这个模块的不同函数；

调用server这个模块的start方法，传两个参数，分别是router模块的route方法和对象handle（包含3个函数变量）；

②由于调用了server模块的start方法，因此我们来看server模块；

在server模块中，调用了两个模块，分别是http和url，这两个都是NodeJs自带的，具体里面的内容就不一一分析了（前面已经说明过了）。

在servre模块的start函数（也是index.js调用的方法），有两个参数，第一个参数是route函数（来源于router模块），第二个参数是一个对象（来源于index.js）。

而在start函数中，调用了其第一个参数（route函数），由于这个参数是函数，因此运行它，给其两个参数，分别是handle（来源于index.js的对象）和pathname（来源于当前模块，是用户请求的路径）；

因此接下来我们看router模块的route函数，记得，这个函数传递了一个对象和一个路径；

③在router模块中，传递了两个参数，并进行了一次判断；

判断的逻辑是这样的，第一个参数（是一个对象），其加上下标后的值，如果是类型是函数，那么执行这个函数；如果不是函数，提示没找到该句柄；

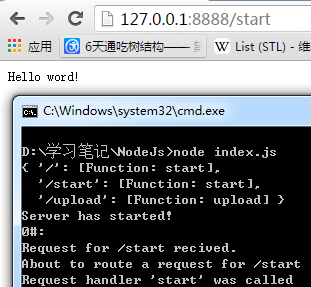
请注意，为什么说这个对象加上下标是函数呢，原因在于index中声明的这个对象，而这个对象的三个带下标的变量都是函数；

而这三个被执行函数来源于哪呢？来源于index里加载的模块requestHandlers，那么最后让我们来看看这个模块；

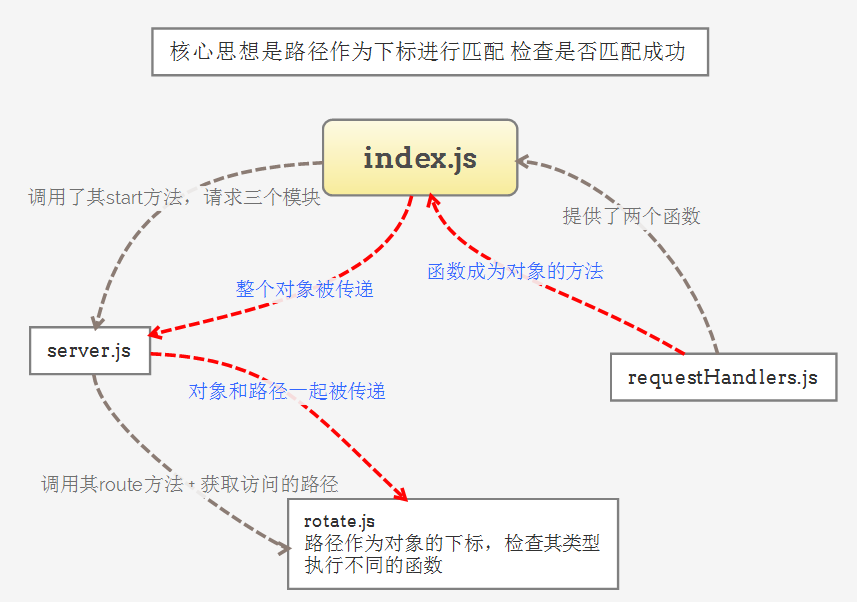
④在这个模块里，只有两个函数，分别是start和upload；

这两个函数输出不同的内容（一个关键字是start，另一个关键字是upload）；

请回想之前的下标，下标分为三个，分别是空（/），start下标（/start），和upload下标（/upload），因此当用户访问根目录、start目录、和upload目录时，调用对应的方法，而若访问其他目录（包括start目录的子目录等时，提示错误——来源于router模块）；



现在回顾这种模式是怎么实现的：



**（7）等待函数**

function sleep(milliSeconds) {  
 var startTime = new Date().getTime();  
 while (new Date() < startTime + milliSeconds); //等待参数的时间  
}

传的参数是毫秒，在这个时间前，无法执行下一句代码，类似C++的Sleep()函数

**（8）返回的两种格式**

response.writeHead(200, {**"Content-Type"**: "text/html"});  
response.write(content);  
response.end();

其中，writeHead里面的第二个参数，有多种格式，分别是：

application/xml 、 text/xml、text/html、text/plain

经查：

text/html是html格式的正文   
text/plain是无格式正文

text/xml忽略xml头所指定编码格式而默认采用us-ascii编码

application/xml会根据xml头指定的编码格式来编码：

简单来说，html就是返回一个html，plain就是无格式的，写什么就是什么，xml就是一个xml文件；

**（9）关于事件监听**

之前说到有

function onRequest(request, response)

这样一个回调函数，其中request是请求，response是回应；

根据推测，每次request的，都会触发至少一个data，一个end事件，因此应该监听这两个事件。（并且只有接受到data之后，才会接收到end事件）；

我们之前是不监听的，只要有请求，就直接执行某个函数，而监听表示只有这些事件触发后，我们才会执行某一段代码。

因此我们这么写：

var postData = "";  
request.setEncoding("utf8");  
request.addListener("data", function (postDataChunk) { //data事件  
 postData += postDataChunk;  
 **console**.log("Received POST data chunk " + postDataChunk + ".");  
 **console**.log("————————");  
})  
request.addListener("end", function () {  
 route(handle, pathname, response, postData);  
})

表示监听的编码类型为UTF8，监听data事件，把内容加起来，监听end事件，把加起来的内容发送出去。

直到end事件触发时，才会执行之前写的route函数（也就是那个写回复的函数）。

**PS：**

①根据我的测试，如果data事件注释掉，end事件会无法执行，准确的说，是无法访问任何url的。

但是，如果不注释掉，只是普通的访问的话，可是又不会触发data监听事件里面的console.log代码，很奇怪；

另外，注释掉end事件，留着data事件，也会卡住；

**（10）用nodejs控制mysql（windows）**

①安装nodejs的mysql模块

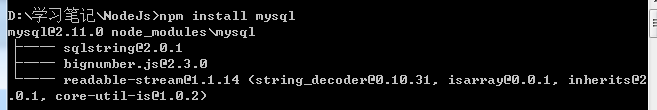
如果你的nodejs的版本比较新的话，应该不存在安装路径问题；

在你想要运行nodejs文件的根路径，启动命令行（shift+鼠标右键）；

输入：

**npm install mysql**

正常的话，应该出的是一个带树形结构的图，比如这样：



如果差别过大，可能是出问题了，那么新建一个js文件，把以下内容复制粘贴进去

注意修改mysql的账号名、密码，而端口号3306是默认的；

var mysql = require('mysql'); //调用MySQL模块  
  
//创建一个connection  
var connection = mysql.createConnection({  
 **host** : '127.0.0.1', //主机  
 **user** : 'root', //MySQL认证用户名  
 **password** : '', //MySQL认证用户密码  
 **port**: '3306', //端口号  
});  
//创建一个connection  
connection.connect(function(err){  
 if(err){  
 **console**.log('[query] - :'+err);  
 return;  
 }  
 **console**.log('[connection connect] succeed!');  
});  
//执行SQL语句  
connection.query('SELECT 1 + 1 AS solution', function(err, rows, fields) {  
 if (err) {  
 **console**.log('[query] - :'+err);  
 return;  
 }  
 **console**.log('The solution is: ', rows[0].solution);  
});  
//关闭connection  
connection.end(function(err){  
 if(err){  
 return;  
 }  
 **console**.log('[connection end] succeed!');  
});

如果正常运行的话，可以从控制台console.log出来的消息确认（具体请看代码）；

**②mysql的增add：**

假如我想在mysql里，数据库（database）名为test，表名为a\_test\_table插入数据。

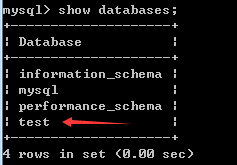
假设你只装了mysql，却不懂mysql，那么这里我推荐你安装mysql-5.6.17-winx64这个版本，这样可以保证代码正常执行。

然后进入mysql，默认账号名为root，默认密码为空；

【以下代码注意结尾的分号，分号表示一条命令的结束】

输入**show databases;**

可以查看库名。应该有一个test库，如下图：



如果没有的话，请自行搜索如何创建一个库；

然后输入**use test;**

进入test表；然后输入**show tables;**

显示test库下面的所有表，如下图：



当然，正常来说，你这里应该是空。于是我们需要建表，为了方便，建一个最简单的表，输入以下命令：

**create table a\_test\_table(**

**id int,**

**name varchar(20));**

创建一个表，他只有两个字段，分别是id（int类型表示数字），name（字符类型）。此时再打show tables; 便可以看到a\_test\_table这个表了；

再输入**elect \* from a\_test\_table;**

查询表内容，会显示空：



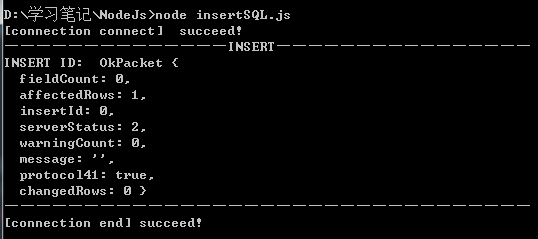
下来开始正文，如何在这个表里插入内容：

首先，新建一个js文件，记得使用utf-8格式。

下面是代码，具体请看注释：

var mysql = require('mysql'); //调用MySQL模块  
  
//创建一个connection  
var connection = mysql.createConnection({  
 **host**: '127.0.0.1', //主机地址  
 **user**: 'root', //MySQL认证用户名  
 **password**: '', //MySQL认证用户密码  
 **port**: '3306', //端口号  
 **database**: 'test', //库名  
});  
//创建一个connection，即连接到mysql的对象，如果连接错误，会提示错误。  
connection.connect(function (err) {  
 if (err) {  
 **console**.log('[query] - :' + err);  
 return;  
 }  
 **console**.log('[connection connect] succeed!');  
});  
  
var userAddSql = 'INSERT INTO a\_test\_table(id, name) VALUES(?,?)';  
/\*  
这里实际上是一个mysql语句，  
关键是第三个a\_table\_test表示表名；  
id和name表示字段名；  
两个问号表示插入的值，如果不是问号，而是一个值，那么表示这个字段固定插入这个值  
\*/  
var userAddSql\_value = [1, 'wddddd']; //这里实际依次对应的是上面2个问号，因为id是int，所以第一个是数字，第二个是字符串  
  
connection.query(userAddSql, userAddSql\_value, function (err, result) {  
 if (err) {  
 **console**.log('[INSERT ERROR] - ', err.**message**);  
 return;  
 }  
  
 **console**.log('——————————————INSERT————————————————');  
 **console**.log('INSERT ID: ', result);  
 **console**.log('—————————————————————————————————');  
})  
  
//关闭connection  
connection.end(function (err) {  
 if (err) {  
 return;  
 }  
 **console**.log('[connection end] succeed!');  
});

然后执行js文件，正常的话，会看到如下提示：



如果显示错误，请检查自己的nodejs代码，一般问题常发生在账号名、密码、库名、表名等。

**③查select：**

有没有发现代码的特点，他的关键是mysql的命令，只要我们更改命令，完全可以像直接在mysql里输入命令行一样操控mysql，例如，我们只需要改这么一段代码：

var userAddSql = 'SELECT \* FROM a\_test\_table';

即使完全不动其他代码，那么回调函数也能正常运行，其表现应该是这样：



当然，完全不改是不好的，但无论如何，我们需要的功能是实现了。

另外，显而易见，我们需要对返回内容进行操纵，那么首先我们需要明确返回内容是什么？结论是，他是一个数组（从 [] 可以猜出），然后我们这个数组只有一项，因此加上下标[0]，假如我们需要获取name属性，那么就是result[0].name；

总得来说，代码在之前的基础上这么修改：

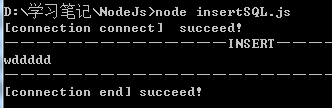
把

**console**.log('INSERT ID: ', result);

修改为：

**console**.log(result[0].name);

然后再次执行js文件，显示结果应该是这样：



**④删delete**

是不是感觉很简单，让我们go on

这次我们要删除id=1的那一项（因为id是唯一的，不会删错）——虽然我们其实就一项

在之前代码上继续修改：

var userAddSql = 'DELETE FROM a\_test\_table WHERE id = 1'; //之所以大写mysql的指令，是为了和数据区分，事实上不大写也是可以的

然后执行之；

毫无疑问，代码会报错，但是并非我们删除命令错误了，而是因为之前的代码，会输出结果的name属性，显然咯，都被删除了，哪有name属性；

为了证明我们删除命令并没错，在mysql里进行查询：



我们可以发现，我们需要删除的那一项确被删除了。

**⑤改update**

增删查改，我们只剩下改了。但是目前mysql表里没内容，我们需要重新加上一些mysql信息。

一次插入多条数据：

var userAddSql\_value = [[1, 'a'], [2, 'd']];

for (var i = 0; i < userAddSql\_value.**length**; i++) {  
 connection.query(userAddSql, userAddSql\_value[i], function (err, result) {  
 if (err) {  
 **console**.log('[INSERT ERROR] - ', err.**message**);  
 return;  
 }  
  
 **console**.log('——————————————INSERT————————————————');  
 **console**.log(result[0].name);  
 **console**.log('—————————————————————————————————');  
 })  
}

然后接下来，我们要修改数据了。

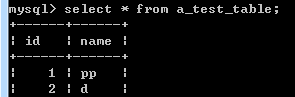
我们这次的目的，是把id=1的项，name修改为pp

需要注意的是，所有id=1的项，name属性都将被修改为pp（而不是仅仅只有一个）；

var userAddSql = 'UPDATE a\_test\_table SET name = ? WHERE id = ?';   
var userAddSql\_value = ["pp", 1];

connection.query(userAddSql, userAddSql\_value, function (err, result) {  
 if (err) {  
 **console**.log('[INSERT ERROR] - ', err.**message**);  
 return;  
 }  
  
 **console**.log('——————————————INSERT————————————————');  
 **console**.log('INSERT ID: ', result);  
 **console**.log('—————————————————————————————————');  
})

运行之，然后再查表，正常的话，你会发现，id=1的项，其值被更为为了pp



**⑥断线重连**

按照说明，使用连接池的方法更好，不过既然看到了断线重连，虽然挺迷茫，但正常运行了（虽然并不明白，而且感觉有点出入），那就先用着

代码这么写：

var mysql = require('mysql'); //调用MySQL模块  
  
//创建一个connection  
var db\_config = mysql.createConnection({  
 **host**: '127.0.0.1', //主机地址  
 **user**: 'root', //MySQL认证用户名  
 **password**: '', //MySQL认证用户密码  
 **port**: '3306', //端口号  
 **database**: 'test', //库名  
});  
var connect;  
function handleDisconnect() {  
 connect = mysql.createConnection(db\_config); //创建连接（连接的设置是上面）  
 connect.connect(function (err) { //大概是指连接，  
 if (err) {  
 **console**.log("正在连接中：" + new Date());  
 setTimeout(handleDisconnect, 3000);  
 return;  
 }  
 **console**.log("连接成功");  
 run();  
 })  
 connect.on('error', function (err) {  
 **console**.log('db error', err);  
 if (err.**code** === 'PROTOCOL\_CONNECTION\_SET') {  
 handleDisconnect();  
 } else {  
 throw err;  
 }  
 })  
}  
handleDisconnect();  
function run() {  
 db\_config.connect(function (err) {  
 if (err) {  
 **console**.log('[query] - :' + err);  
 return;  
 }  
 **console**.log('[connection connect] succeed!');  
 });  
  
 var userAddSql = 'UPDATE a\_test\_table SET name = ? WHERE id = ?';  
 var userAddSql\_value = ["www", 1];  
  
 db\_config.query(userAddSql, userAddSql\_value, function (err, result) {  
 if (err) {  
 **console**.log('[INSERT ERROR] - ', err.**message**);  
 return;  
 }  
  
 **console**.log('——————————————INSERT————————————————');  
 **console**.log('INSERT ID: ', result);  
 **console**.log('—————————————————————————————————');  
 })  
  
  
//关闭connection  
 db\_config.end(function (err) {  
 if (err) {  
 return;  
 }  
 **console**.log('[connection end] succeed!');  
 });  
}

**⑦连接池：**

然后我试了试连接池的方法：

var mysql = require('mysql');  
var pool = mysql.createPool({  
 **host**: 'localhost',  
 **user**: 'root',  
 **password**: '',  
 **database**: 'test',  
 **debug**: false,  
});  
  
var insert = function (connection, data) {  
 connection.query('INSERT INTO a\_test\_table SET ?', data, function (err, result) {  
 **console**.log('ID : ' + result.**insertId**);  
 });  
};  
  
var update = function (connection, data) {  
 connection.query('UPDATE a\_test\_table SET name = ? WHERE id = ?', data, function (err, result) {  
 });  
};  
  
var select = function (connection) {  
 connection.query('SELECT \* FROM a\_test\_table', function (err, result) {  
 result.forEach(function (user) {  
 **console**.log(user.**id** + ':' + user.name + ':');  
 });  
 });  
};  
  
pool.getConnection(function (err, connection) {  
 var data = ['www', 1];  
 select(connection);  
 update(connection, data);  
 select(connection);  
});  
  
**console**.log('mysql is start!');

**然后我尝试停用了mysql，运行这段代码，发现，停用后无法连接，并报错；如果是正在连接，那么会自动断掉连接。**

**不是很明白如何断线重连。**

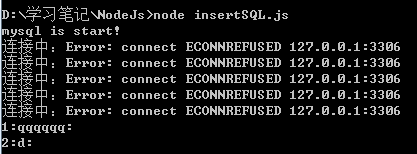
**也许是假如断线了，就运行这个pool.getConnection这句代码？**

**但是如何判断呢？不清楚**

像下面这么写么？假如没连上，的确可以尝试重复连接，但是如果连接上的时候，断掉了，却不会继续重连。

function toConnet() {  
 pool.getConnection(function (err, connection) {  
 if (err) {  
 **console**.log("连接中：" + err);  
 setTimeout(toConnet, 2000);  
 } else {  
 var data = ['bbbb', 1];  
 select(connection);  
 update(connection, data);  
 select(connection);  
 }  
 })  
}  
toConnet();

下面的表示链接成功了



**另外一个问题是：**

假如我需要多次操纵。我该如何写这段代码？

准确的说，我第一次的代码，已经成功修改了mysql的数据了，那么我下一次该修改mysql的数据，代码如何组织？

是继续写pool.getConnection这段代码么？

**⑧防止SQL注入：**

使用方法：escape()

具体用法是：

参照：

http://www.cnblogs.com/zhongweiv/p/nodejs\_mysql.html#mysql\_curd

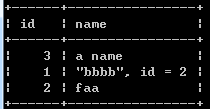
不过貌似nodejs本身就带有一定的防注入功能。例如，我把data改为：

var data = ["'bbbb', id = 2", 1];

这样拼接出来的sql代码应该是：

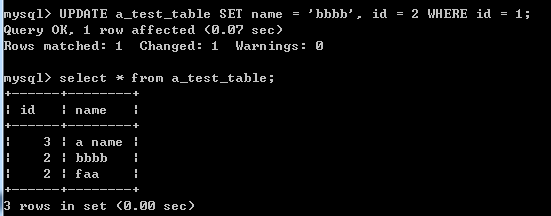
UPDATE a\_test\_table SET name = 'bbbb', id = 2 WHERE id = 1

运行这个js文件后，会发现结果变成这样：



但若在mysql里面运行这段代码：

结果变成这样：



说明我们拼接的代码是没错的，只是mysql防止了这种简单的sql注入攻击。

**（11）使用-e来执行nodejs代码**

打开cmd，然后这么输入：

node -e "console.log('Hello World');"

输出：



有点类似eval()

不过不太明白这种方法有什么优势

另外需要注意，-e后面的js代码，需要用双引号包含，而不能用单引号，就像这样：



这种是没有运行结果的

**（12）调试用插件**

打开控制台，输入：

npm install -g supervisor

将安装supervisor模块，

当代码被改动时，运行脚本会被终止，然后重新启动。

这样避免了每次修改代码后要重新启动js文件的问题。

但需要通过supervisor来启动才可以

**（13）文件读取**

模块需求：fs

方法：

①首先我们创建一个test.txt文件，里面输入内容

It's a test txt.

注意，这个txt文件应该是utf-8格式的

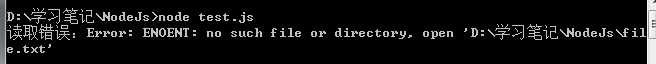
②然后我们创建一个js文件，调用fs模块，并输入一段代码：

注意，此时文件名是错误的

然后执行这个js文件

var fs = require('fs');  
  
fs.readFile('file.txt', 'utf-8', function (err, data) {  
 if (err) {  
 console.log("读取错误：" + err);  
 return;  
 }  
 console.log(data);  
})

输出：



③这次我们修改文件名，改为test.txt，再次运行js文件



④假如不指定读取的格式：（即去掉第二个参数）



输出的是十六进制字节

⑤假如读取的文件编码是ANSI，然后用utf-8读取，那么中文字符基本上是无法读取出来的。

除了utf-8格式之外，还有utf8, ucs2, ascii, binary, base64, hex等格式

注意，没有GBK之类，具体解决办法参照：

http://blog.csdn.net/youbl/article/details/29812669

另外，注意这个是异步请求。

⑥同步读取文件：

var fs = require('fs');  
var data = fs.readFileSync('test.txt', 'utf-8');  
console.log(data);

**（14）事件**

①Nodejs的异步I/O，按照说明，事件是归属于events模块的EventEmitter对象，是他的一个方法；

我个人的感觉是，这个类似于消息的发布和订阅。如果用过dojo的topic模块的人，应该能理解。

var EventEmitter = require('events').EventEmitter;  
var event = new EventEmitter(); //声明一个他的实例；  
  
event.on('aEvent', function (str) { //当aEvent这个事件  
 console.log("a event happed!");  
})  
event.on('aEvent', function (str) {  
 console.log(str);  
})  
setTimeout(function () {  
 event.emit('aEvent', 'data') //这里是发布事件（参数一是事件名，应该是唯一标识）（参数2是这个事件触发时，作为回调函数的参数。根据推论，之后的依次作为参数传递给回调函数）  
}, 1000);

**（15）模块**

当我们制作一个模块时（具体制作方法见前面），我们可能会写一个构造函数（因为在浏览器写构造函数时我们经常会这么做）。

如代码：

function sayHello() {  
 this.say = function () {  
 console.log('hello');  
 }  
}  
exports.Hello = sayHello;

在这里，我们有一个构造函数sayHello，函数里有一个方法say，输出hello。

然后导出的是sayHello这个构造函数。我们在另外一个文件中调用它。

代码这么写：

var hello = require('./test').Hello; //调用刚才写的那个模块  
var m = new hello();  
m.say();

第一行表示我们调用刚才写的那个模块。

注意，这个时候和之前不同（之前是直接require模块即可），这里需要添加一个Hello。原因在于require表示导入的是一个整体（即有多个exports导出的方法），这里我们只需要其中的Hello方法（即对应原模块中的sayHello这个构造函数）；

当然，也可以写为require('./test')，在下面的代码，把hello改为hello.Hello即可。

第二行代码表示生成一个这个构造函数的实例（还记得js里如何调用构造函数么？构造函数是不能直接调用其方法的）；

第三行代码表示调用这个实例的say方法（即模块中的this.say=的那个函数）；

**这个流程是：**

请求模块——》生成函数的实例——》调用函数的方法

我们也可以在导出的时候，用另一种方法：

module.exports = **sayHello**;

然后调用这个模块

var hello = require('./test'); //调用刚才导入的那个文件  
m = new hello();  
m.say();

这个时候hello就是这个函数，然后生成一个实例，调用其方法。

但我觉得有个潜在问题，这个模块里只能有这一个函数（不是很确定）。我验证结果如下：

修改之前的模块：

function sayHello() {  
 this.say = function () {  
 console.log('hello');  
 }  
}  
function say() {  
 console.log("say");  
}  
module.exports = sayHello;  
exports.say = say;

调用其的模块：

var hello = require('./test'); //调用刚才导入的那个文件  
console.log(hello.say)

显示是undefined，而正常情况下（删除module.exports = sayHello;这一行）应该显示：



经过查询，这种用法的作用在于，**假如我们只想将一个对象封装到模块中，就这么用。**

另外，不可以直接对exports进行赋值以替代（module.exports这种用法）。

按照说明，这个变量会在模块执行结束后被释放，但module.exports不会。

**（16）包**

①Nodejs的包类似C/C++的函数库或者Java/.Net的类库等。

将某个独立的功能封装起来，用于发布、更新、依赖管理和版本控制。

标准情况下（符合CommonJS规范）的包应该具有以下特征：

【1】package.json必须在包的顶层目录下；

【2】二进制文件放在bin下面；

【3】JavaScript代码放在lib下面；

【4】文档放在doc下面；

【5】单元测试放在test下面

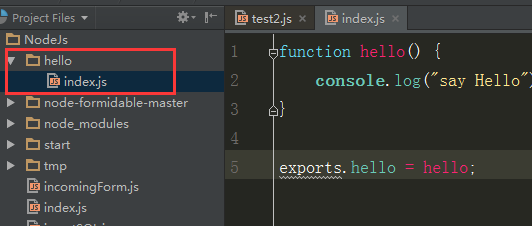
一般情况下，顶层目录下有package.json，并且符合一些标准规范就行。

②关于nodejs在使用包时的流程：

【1】首先检查文件夹下的package.json文件中的main字段（看他指向哪个js文件）；

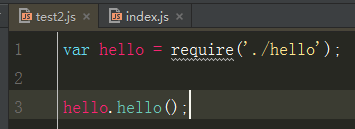
如果有，作为包的接口模块；

【2】如果没有的话，寻找index.js或index.node作为包的接口。



如图，我创建了一个hello文件夹，里面有一个index.js的文件，其中有一个hello方法。

然后调用它：



然后再运行：



正常。

但是我突然有个问题，假如有一个hello文件夹（和之前一样），又有一个hello.js的文件，那么他会调用哪个呢？have a try！

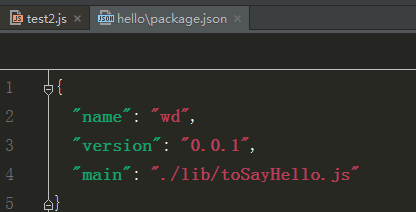


尝试之后发现，**相对于文件夹，他会优先调用同名的js文件**。

即：

**js文件——》文件夹（package.js里的main→index.js或index.node）**

然后，我们按照之前要求建立一个json文件：



这里的toSayHello.js就是之前的hello.js文件

然后把js文件放在main里描述的地方。

重新调用test2.js（记得删除之前根目录下的hello.js文件）



一切正常~~

③package.json的一些规范：

* name：包的名称，必须是唯一的，由小写英文字母、数字和下划线组成，不能包含空格。
* description：包的简要说明。
* version：符合语义化版本识别①规范的版本字符串。
* keywords：关键字数组，通常用于搜索。
* maintainers：维护者数组，每个元素要包含 name、email （可选）、web （可选）字段。
* contributors：贡献者数组，格式与maintainers相同。包的作者应该是贡献者数组的第一个元素。
* bugs：提交bug的地址，可以是网址或者电子邮件地址。
* licenses：许可证数组，每个元素要包含 type （许可证的名称）和 url （链接到许可证文本的地址）字段。
* repositories：仓库托管地址数组，每个元素要包含 type （仓库的类型，如 git ）、url （仓库的地址）和 path （相对于仓库的路径，可选）字段。
* dependencies：包的依赖，一个关联数组，由包名称和版本号组成。

④安装一个包：

【1】直接把包的文件夹复制在根目录下的node\_modules文件夹下；

【2】或者是使用npm install/i 包名

使用install或者i都可以

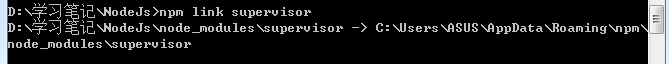
【3】以上是本地模式（仅对当前目录起效），另外还有全局模式的安装包（对所有文件起效）

npm install -g 包名

另外，使用全局模式的包的安装，不能简单的使用require来调用，

【4】全局包通过require在本地调用（当前文件夹）：

比如我之前安装了调试插件supervisor，我们想在本地调用它（虽然好像并不能这么做，但可以作为示例），我们就在根目录下面打开cmd，输入npm link supervisor即可，他会提示：



（ps：虽然书上说windows不支持这个命令，但我依然可以使用）

【5】把本地的包link到全局：

在包目录下（package.json所在的目录），运行**npm link**命令即可。

【6】也可以将这个包发布出去，具体略。

⑤nodejs的调试：

使用**node debug文件名**

来调试，具体略

**（17）全局对象**

①nodejs的全局对象是**global**

他和它的属性、方法，都可以在程序的任何地方进行访问

②以下是全局变量：

【1】在最外层定义的对象；

【2】全局对象（比如global）的属性；

【3】隐式定义的变量（未定义直接赋值的变量）；

③process

是一个全局变量，global对象的属性。

用于**描述当前Node.js进程状态**的对象，提供了一个与操作系统的简单接口。

通常**写本地命令行程序的时候使用**。

【1】process.argv

返回当前进程的命令行参数数组

就是说，假如我运行一个js文件：

node test.js a b c

其中前2个是我正常运行的，a、b、c是我在cmd启用时调用的命令行参数。于是这个就返回这个命令行参数。

其中process.argv[0]是nodejs的文件目录；

process.argv[1]是运行文件的路径

process.argv[2]是命令行参数的第一个参数，[3]、[4]等依次类推

可以通过console.log输出

【2】process.stdout

标准输出流，比console.log更底层，就简单用法来说：

process.stdout.write("1111");

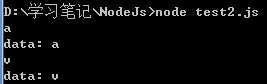
相当于：console.log("1111")

当然，实际上她能做更多，不过我目前还没搞懂，所以暂且搁置

【3】process.stdin

标准输入流，不过不太明白怎么用。有点类似c++里的cin，但是我不明白如何中止这个输入状态（回调函数会无限执行）；

process.stdin.resume();  
process.stdin.on('data', function (**data**) {  
 process.stdout.write("data: " + **data**.toString());  
})



【4】process.nextTick(callback)

为事件循环设置一项任务，nodejs会在下次事件循环调用响应时调用回调函数；

简单来说，就是分拆事件。把一个耗时间长的事件，分拆成多个事件短的（分拆多少个要看nextTick如何设置），然后执行，提高响应速度。

**具体我也不是很懂，略吧。**

④console

【1】console.log

除常规用法外，还有类似c语言的printf的用法。

例如：

**console**.log("a%db", 1);

输出：

a1b

**console**.log("a%sb", " + ");

输出：

a + b

即：

%d输出数字；

%s输出字符串；

然后我从c那边截取printf里的用法，不保证都能用（未验证）



【2】console.error输出错误

【3】console.trace()向标准错误流输出当前的调用栈

**（18）常用工具util**

util是Node.js的核心模块，提供常用函数的集合。

**①util.inherits**

实现对象间原型继承的函数，具体用法是：

var util = require("util");

util.inherits(继承, 被继承的原型)

如代码：

var util = require('util');  
function sayHello() {  
 this.sayHello = function () { //通过这个方法定义的是不能被继承的  
 **console**.log("say Hello");  
 }  
}  
function sayHi() {  
 this.sayHi = function () {  
 **console**.log("say Hi");  
 }  
}  
sayHello.**prototype**.name = function () { //通过这个方式定义的方法可以被继承  
 **console**.log("A name");  
}  
util.inherits(sayHi, sayHello);  
  
var say = new sayHi();  
say.name(); //能正常输出  
//say.sayHello(); 会出错

需要注意的是：

【1】构造函数内部的方法、属性是不能被继承的；

【2】构造函数通过prototype声明的属性、方法，是可以被继承的

**②until.inspect**

将任意对象转为字符串

原型：

util.inspect( object, [showHidden], [depth], [colors]);

object是被转换对象；

showHidden可选，默认false，如果为true，将输出更多隐藏信息；（主要是对函数方法）

depth可选，最大递归层数，默认为2，值为null表示完整递归遍历；

colors如果为true，输出格式将以ANSI编码，用于在终端显示更漂亮的效果；

如代码

var util = require('util');  
function sayHello() {  
 this.**sayHello** = "say Hello";  
}  
var say = new sayHello();  
**console**.log(util.inspect(say, false, null, true));

输出：



③类型检查：

util.isArray(object) 是否是数组

util.isRegExp(object) 是否是正则表达式

util.isDate(object) 是否是日期

util.isError(object) 是否是错误类型

**（19）events事件模块**

①events模块只提供了一个对象，**events.EventEmitter**

他的核心是 事件发射 和 事件监听 功能

每个事件由一个事件名（用于标识事件），和多个参数组成。

事件名：字符串，通常表达一定的语义；

事件被发射时，监听该事件的函数被依次调用。

②调用模块：

var events = require("events");  
var emitter = new events.EventEmitter();

事件监听：

emitter.on("/click", function () {  
 **console**.log("first event");  
})  
emitter.on("/click", function () {  
 **console**.log("second event");  
})

事件发射：

emitter.emit("/click");

注意：

【1】/click是事件名（用于标识事件）

【2】可以多个监听，用于监听同一个事件，然后依次执行；

【3】需要先监听，后发射；

【4】监听是on，发射是emit

【5】把emit发射的事件赋值给变量。如果有监听该事件的，则变量值为true，如果无监听该事件，则返回值为false。注意，该变量赋值后不会改变。

即：

var nn = emitter.emit("/click1");  
emitter.on("/click1", function () {  
 **console**.log("first event");  
})  
**console**.log(nn);

其值nn依然为false。

③只监听一次：

EventEmitter.once(事件名, 回调函数)

即把上面的on替换为once即可，然后这个只监听一次就失效；

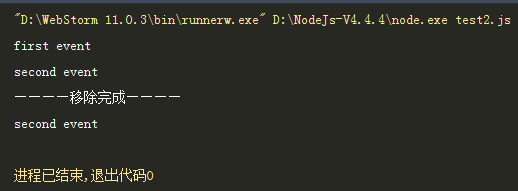
④移除监听：

EventEmitter.removeListener(事件名, 回调函数名)

注意，这里的关键是：**回调函数名**，如代码：

var events = require("events");  
var emitter = new events.EventEmitter();  
var first = function () {  
 **console**.log("first event");  
}  
var second = function () {  
 **console**.log("second event");  
}  
  
emitter.on("/click", first)  
emitter.on("/click", second)  
emitter.emit("/click");  
emitter.removeListener("/click", first);  
**console**.log("————移除完成————");  
emitter.emit("/click");

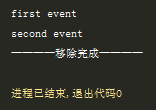
输出：



⑤全部监听移除：

var events = require("events");  
var emitter = new events.EventEmitter();  
var first = function () {  
 **console**.log("first event");  
}  
var second = function () {  
 **console**.log("second event");  
}  
  
emitter.on("/click", first)  
emitter.on("/click", second)  
emitter.emit("/click");  
emitter.removeAllListeners("/click");  
**console**.log("————移除完成————");  
emitter.emit("/click");

输出：



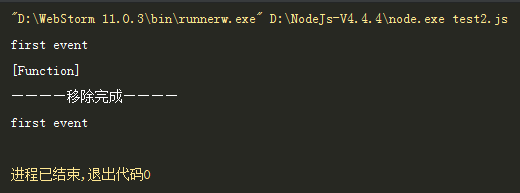
⑥error事件

当遇见异常时会发射error事件，EventEmitter规定，如果没有监听其的监听器，Node.js会把其当成异常，退出程序并打印调用栈。

因此需要设置监听其的监听器，避免遇见错误后整个程序崩溃。

var events = require("events");  
var emitter = new events.EventEmitter();  
var first = function () {  
 **console**.log("first event");  
}  
var error = function (error) {  
 **console**.log(error);  
}  
  
emitter.on("/click", first)  
emitter.on("error", error) //如果没有这一行代码，下面在发射error时会出错然后退出程序  
emitter.emit("/click");  
emitter.emit("error", error)  
**console**.log("————移除完成————");  
emitter.emit("/click");

输出：



但注意：

【1】这个似乎只能对事件的error有效，假如你把发射error事件那行替换为随便几个字母，他依然是会出错然后退出的

【2】注意，error事件不能将console.log替换为error()，会导致直接出错退出

【3】另外，我尝试了用console.log，还有util.inspect方法来显示这个函数内容，可是没成功。

**（20）文件系统fs**

①提供了文件的读取、写入、更名、删除、遍历目录、链接等功能。

所有操作都有异步和同步两种方法。

**②文件读取：readFile**

fs.readFile(**filename**, [encoding], [callback(err, **data**)])

参数一：是文件名；

参数二：编码方式，默认二进制（具体搜之前的）；

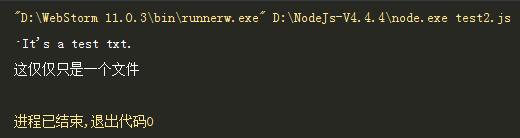
参数三：回调函数（参数一是错误，参数二是读取的文件内容）；

标准方法：

var fs = require("fs");  
fs.readFile("./test.txt", "**UTF8**", function (err, data) {  
 if (err) {  
 **console**.log(err);  
 } else {  
 **console**.log(data);  
 }  
})

参数二可以是UTF8，也可以是utf8，也可以是utf-8。效果是一样的

输出：



**同步版：**

**readFileSync(文件名, 文件编码)**

返回值是data（读取的文件内容）；如果有错误发生，会抛出异常，需要使用try和catch来处理；

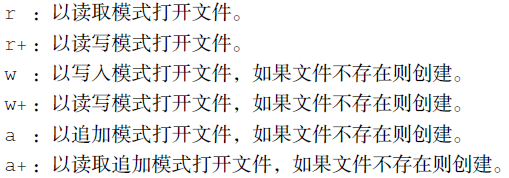
③fs.open文件的打开；

原型：

fs.open(path, flags, [mode], [callback(err, fd)])

path是路径，略；

flags表示打开的方式，具体如图：



mode表示权限，默认是0666。

6表示4+2；根据推测，1执行，2写，4读，他们的和就是权限；

回调函数的fd表示文件描述符：非负整数，表示操作系统内核为当前进程所维护的打开文件的记录表索引。

另外，第一个6表示文件所有者的权限；第二个表示同组的用户；第三个表示其他用户；

这个命令需要配合write、read，或者其他命令进行操作，例如：

var fs = require("fs");  
  
fs.open("test.txt", "a", 0666, function (err, fd) {  
 if (err) throw err;  
 fs.write(fd, "pppp", function (e) {  
 if (e) throw e;  
 fs.closeSync(fd);  
 })  
})

在test.txt文件里，追加写入pppp四个字符

**④fs.read文件的读取**

**fs**.read(fd, buffer, **offset**, **length**, position, callback(err, bytesRead, buffer));

他相对于fs.readFile来说，是更为底层的接口。

其作用是从fs.open的回调函数中的文件描述符fd中读取数据，并写入buffer指向的缓冲区对象。

offset是buffer写入的偏移量；

length是要从文件中读取的字节数；

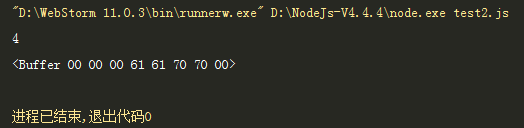
position是文件读取的起始位置（当为null时，则从当前文件指针的位置读取）；

回调函数传递的bytesRead表示读取的字节数；buffer表示缓冲区对象；

如代码：

var fs = require("fs");  
fs.open("test.txt", 'r', function (e, fd) {  
 if (e) throw e;  
 var buf = new Buffer(8); //创建一个二进制对象  
 fs.read(fd, buf, 3, 4, 2, function (err, bytesRead, buffer) { //读取fd，以二进制形式  
 if (err)throw err;  
 **console**.log(bytesRead);  
 **console**.log(buffer);  
 })  
})

输出结果：



第五行的buff表示是上面new出来的Buffer对象；

3表示写入到buf这个对象的第4个位置；

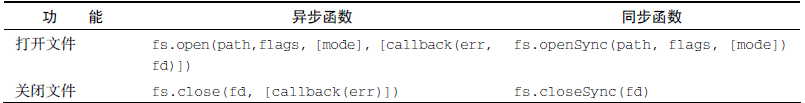
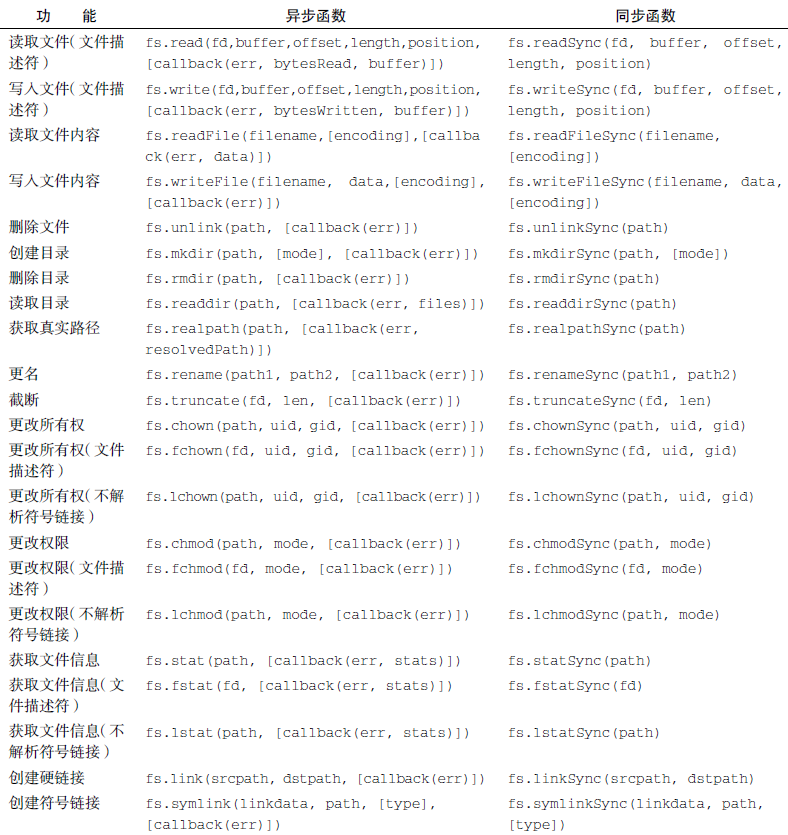
4表示读取fd 4个字节；

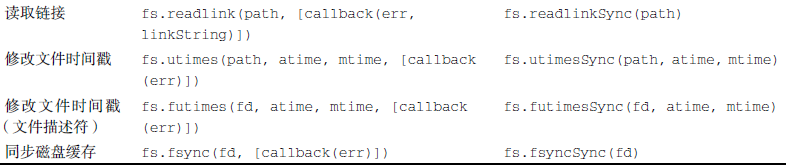
2表示从fd的第二个字节开始读取；

bytesRead表示读取到了4个字节；（假如读取到结尾了，那么这个数字将不包含读取到结尾的那些，比如position位置的数字很大的时候，这里可能就是0）

buffer表示前面那个buf对象

⑤fs模块的函数列表：



**（21）HTTP服务器与客户端**

①http模块里封装了一个高效的HTTP服务器和一个简易的http客户端。

②http.Server是一个基于事件的HTTP服务器，它的核心由Node.js下层的C++部分实现，而接口由JavaScript封装，兼顾高效和简易。

③http.request则是一个HTTP客户端工具，用于向HTTP服务器发起请求Pingback或者内容抓取。

④http.Server是一个基于事件的HTTP服务器，所有的请求都被封装为独立的事件；

开发者只需要对它的事件编写响应函数即可实现HTTP服务器的所有功能；

他继承自EventEmitter，提供了以下几个事件：

**request：**当客户端请求到来时，该事件被触发，提供两个参数**req**和**res**，分别是http.ServerRequest和http.ServerResponse的实例，表示请求和响应信息。

**connection：**当TCP连接建立时，该事件被触发，提供一个参数**socket**，为net.Socket的实例。本事件的范畴要大于request，因为客户端在Keep-Alive模式下，可能会在同一个连接内发送多次请求。

**close：**当服务器被关闭时，本事件触发（非客户端断开连接时）；

⑤**http.ServerRequest**

是HTTP请求的信息，是后端开发者最关注的内容。一般是由http.Server的request事件发送，作为第一个参数传递（即回调函数中的request或者是req）；

ServerRequest提供了一些属性：



**（22）服务器返回文字设置为utf8**

如代码：

var http = require("http");  
var server = new http.Server();  
server.on("request", function (req, res) {  
 res.writeHead(200, {**'Content-type'**: 'text/html;charset = utf8'});  
 res.write("<h1>这里是一个头</h1>");  
 res.end();  
})  
server.listen(8888);  
**console**.log("The server begin");

**（23）当用户不输入端口的时候依然能监听**

默认端口是80，所以listen(80)即可

**（24）request的结构**

这里指的是发送的结构：

先上代码：

var http = require("http");  
var util = require("util");  
var server = new http.Server();  
server.on("request", function (req, res) {  
 res.writeHead(200, {**'Content-type'**: 'text/html;charset = utf8'});  
 res.write("<h1>这里是一个头</h1>");  
 res.write("是否完成：" + req.**complete**);  
 res.write("</br>Http版本协议：" + req.httpVersion);  
 res.write("</br>请求方法：" + req.**method**);  
 res.write("</br>原始请求路径：" + req.**url**);  
 res.write("</br>请求头：" + util.inspect(req.**headers**)); //将其转为字符串  
 res.write("</br>请求尾：" + util.inspect(req.trailers));  
 res.write("</br>当前连接套接字：" + req.connetion);  
 res.write("</br>上面的别名：" + req.socket);  
 res.write("</br>client属性的别名：" + req.client);  
 res.end();  
})  
server.listen(80);  
**console**.log("The server begin");

结果为：

# 这里是一个头

是否完成：false  
Http版本协议：1.1  
请求方法：GET  
原始请求路径：/  
请求头：{ host: '127.0.0.1', connection: 'keep-alive', 'cache-control': 'max-age=0', 'upgrade-insecure-requests': '1', 'user-agent': 'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.63 Safari/537.36', accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8', 'accept-encoding': 'gzip, deflate, sdch', 'accept-language': 'zh-CN,zh;q=0.8' }  
请求尾：{}  
当前连接套接字：undefined  
上面的别名：[object Object]  
client属性的别名：[object Object]

另外，请求头是一个对象，因此可以列出他的属性，使用for in遍历之，如代码：

var http = require("http");  
var util = require("util");  
var server = new http.Server();  
server.on("request", function (req, res) {  
 res.writeHead(200, {**'Content-type'**: 'text/html;charset = utf8'});  
 for (var i in req.**headers**) {  
 res.write("req.headers[" + i + "]: " + req.**headers**[i] + "</br>")  
 }  
 res.end();  
})  
server.listen(80);  
**console**.log("The server begin");

结果：

req.headers[host]: 127.0.0.1  
req.headers[connection]: keep-alive  
req.headers[cache-control]: max-age=0  
req.headers[upgrade-insecure-requests]: 1  
req.headers[user-agent]: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.63 Safari/537.36  
req.headers[accept]: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8  
req.headers[accept-encoding]: gzip, deflate, sdch  
req.headers[accept-language]: zh-CN,zh;q=0.8

另外，很奇怪的是，我试着使用req.headers[cache-control]

或者是req.headers.cache-control ，都无法出来正常的结果，不明白。

注：这里应该这么写：req.headers['cache-control'] 即加上引号

关于cache-control的取值

http://www.phpernote.com/seo/465.html

**（25）http.ServerRequest的过程**

①http.ServerRequest提供了3个事件用于控制请求体传输：

【1】data：当请求体数据到来时，该事件被触发，提供一个参数给回调函数，是接受到的数据，该事件可能被多次调用（所有data按顺序的集合，是请求体数据）。如果该事件没有被监听，请求体将被抛弃；

【2】end：当请求体数据完成时该事件触发。此后不再触发data事件；

【3】close：用户当前请求结束时，该事件被触发。不同于end，如果用户强制终止了传输，也还是调用close

②获取GET请求内容

【1】GET请求被直接内嵌在路径中。URL是完整的请求路径（包括？后面的部分），因此手动解析后面的内容作为GET请求的参数。

Node.js的url模块中的parse函数提供了这个功能。

以url：http://127.0.0.1/user?name=byvoid&email=byvoid@byvoid.com为例：

先上代码：

var http = require("http");  
var url = require("url");  
var server = new http.Server();  
server.on("request", function (req, res) {  
 if (req.**url** == "/favicon.ico") {  
 return;  
 }  
 var m = url.parse(req.**url**, true);  
 **console**.log(m)  
 res.writeHead(200, {**'Content-type'**: 'text/html;charset = utf8'});  
 res.end();  
})  
server.listen(80);  
**console**.log("The server begin");

通过console.log返回的内容是：

Url {

protocol: null,

slashes: null,

auth: null,

host: null,

port: null,

hostname: null,

hash: null,

search: '?name=byvoid&email=byvoid@byvoid.com',

query: { name: 'byvoid', email: 'byvoid@byvoid.com' },

pathname: '/user',

path: '/user?name=byvoid&email=byvoid@byvoid.com',

href: '/user?name=byvoid&email=byvoid@byvoid.com' }

其中几个关键是：

pathname是我们关心的url地址请求；而path是完整的部分

query是get方法的部分；在这里包含了name和email两个属性，他们中间用&相连接；

于是，想要获取GET请求的方法，只需要通过url的query部分即可；

③获取POST请求内容：

POST方法的请求内容，被放置在请求体之中，即之前的data、end、close相关事件；

post方法的请求体内容的解析方法是利用querystring模块的parse方法；

var http = require("http");  
var querystring = require("querystring");  
var server = new http.Server();  
server.on("request", function (req, res) {  
 if (req.**url** == "/favicon.ico") {  
 return;  
 }  
 if (req.**url** !== "/post") { //默认页面是这个  
 var body = '<html>' +  
 '<head>' +  
 '<meta http-equiv="Content-Type" content="text/html; ' +  
 'charset=UTF-8" />' +  
 '</head>' +  
 '<body>' +  
 '<form action="/post" method="post">' +  
 '<textarea name="text" rows="20" cols="60"></textarea>' +  
 '<input type="submit" value="Submit text" />' +  
 '</form>' +  
 '</body>' +  
 '</html>';  
 res.writeHead(200, {**'Content-type'**: 'text/html;charset = utf8'});  
 res.write(body);  
 res.end();  
 } else { //当提交表单后，跳转到这个页面  
 var PostData = null; //这个变量用于存储全部数据  
 req.on("data", function (data) { //因为可能多次data事件，因此需要通过+=来完成  
 PostData += data;  
 })  
 req.on("end", function () { //当data事件触发完毕后会触发这个事件，返回事件必须写在这里，不然PostData是空  
 var message = querystring.parse(PostData); //进行解析，变成一个对象  
 **console**.log(message); //显示这个对象  
 res.writeHead(200, {**'Content-type'**: 'text/html;charset = utf8'}); //返回  
 if (PostData)  
 res.write(message.nulltext);  
 res.end();  
 })  
 }  
})  
server.listen(80);  
**console**.log("The server begin");

PS：按照说明，这种方法有严重效率问题和安全隐患。因此不能在真正的生产环境中使用这种方法。

**（26）http.ServerResponse**

①http.ServerResponse是返回给客户端的信息；

由http.Server的request事件发送的，作为第二个参数来传递，就是上面的res参数；

②由三个重要的成员函数，用户返回响应头、响应内容和结束请求；

【1】res.writeHead(statusCode, [headers]);

第一个参数statusCode表示HTTP状态码，常见的是200（请求成功），404（未找到）等；

第二个参数headers是一个数组，用于表示响应头的每个属性；例如上面的

{**'Content-type'**: 'text/html;charset = utf8'}

表示类型是text/html，编码格式是utf8；

【2】res.write(data, [encoding]);

像请求的客户端发送响应内容，data是一个Buffer（二进制形式）或者字符串，表示要发送的内容；

如果data是字符串，需要指定编码方式，默认是utf-8；

可以多次调用（在end事件前）；

【3】res.end([data], [encoding]);

结束响应，告知客户端已发送完；

本函数必须被调用，否则客户端将一直处在等待状态；

参数同write;

**（27）Express框架**

①安装：

先使用npm install -g express

安装完成后，再次使用：

npm i -g express-generator

来安装另一部分；

否则会出现：“'express' 不是内部或外部命令，也不是可运行的程序或批处理文件。”这样的提示。

原因在于，在4.X版本，把generator分离出来了，需要单独安装

②建立网站的基本结构：

命令行输入express -t ejs microblog

他会创建一个文件夹microblog，

然后在里面创建一些文件，并且告诉我们：



虽然不知道是什么意思，但是照着做吧。

先输入cd microblog，回车，然后再输入npm install

他会安装一大堆东西。

至于为什么这么输入，据说是替换什么的，但我实际操作时，发现和书上写的东西不一样啊~~~好吧，可能是书是好几年前的。

③虽然不太懂，但是没关系，我们看代码。

首先是关键是app.js

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 routes = 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('/', routes);  
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 handlers  
  
// development error handler  
// will print stacktrace  
if (app.get('env') === 'development') {  
 app.use(function (err, req, res, next) {  
 res.status(err.status || 500);  
 res.render('error', {  
 message: err.message,  
 error: err  
 });  
 });  
}  
  
// production error handler  
// no stacktraces leaked to user  
app.use(function (err, req, res, next) {  
 res.status(err.status || 500);  
 res.render('error', {  
 message: err.message,  
 error: {}  
 });  
});  
app.listen(80); //没有这行代码的话，无法正常运行，这行代码是后续添加的  
  
module.exports = app;

请注意，基本的是没有app.listen(80)这一行代码的（倒数第二行），因此执行node app.js不会进行监听。

④代码解释：

**第一部分：引用模块**

var express = require('express'); //这个模块在node\_modules文件夹中  
var path = require('path'); //nodejs自带的  
var favicon = require('serve-favicon'); //node\_modules文件夹中  
var logger = require('morgan'); //node\_modules文件夹中  
var cookieParser = require('cookie-parser'); //node\_modules文件夹中  
var bodyParser = require('body-parser'); //node\_modules文件夹中  
  
var routes = require('./routes/index'); //routes文件夹中的js文件  
var users = require('./routes/users'); //routes文件夹中的js文件  
  
var app = express(); //生成一个导入的express模块的实例

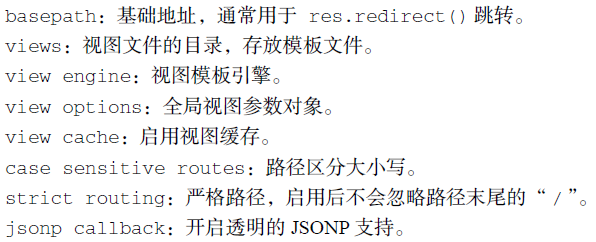
引用模块的最后，生成一个express的实例，之后的都将对其进行操作；

**第二部分，设置：**

// view engine setup  
app.set('views', path.join(\_\_dirname, 'views')); //视图文件的目录  
app.set('view engine', 'jade'); //视图模板引擎是jade

app.set是Express的设置工具，其有两个参数，一个是key，一个是值。

key有以下几种：



在以上，使用了views和view engine这两个功能。

另外：

path.join(\_\_dirname, "view", "ff")

这行代码的作用是，查找当前目录（**应该是当前文件目录？而不是根目录？**），然后是当前目录里的view文件夹，view文件夹里的ff文件夹。

假如\_\_dirname是d:\学习笔记\NodeJs\Project1\

那么这一段代码的结果是：d:\学习笔记\NodeJs\Project1\microblog\**view\ff**

可以直接使用path.join(\_\_dirname)来获取当前路径，不过这样的话显然很不漂亮

可以直接使用**\_\_dirname**来获取路径，

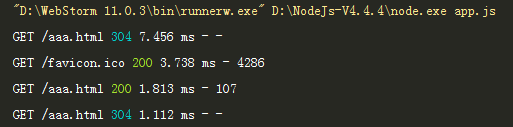
比如，文件位置是D:\学习笔记\NodeJs\test.js，那么通过这种方式获取的路径则为：D:\学习笔记\NodeJs

**第三部分，功能的加载：**

app.use(favicon(path.join(\_\_dirname, 'public', 'favicon.ico'))); //当需要加载favicon.ico时调用（应该，需要验证）  
app.use(logger('dev')); //log显示，这里指显示在命令行页面，就像console.log那样  
app.use(bodyParser.json()); //解析客户端的请求，通常是通过post发送的内容  
app.use(bodyParser.urlencoded({**extended**: false})); //另一种解析方式，具体不太明白  
app.use(cookieParser()); //cookies的解析，具体暂不清楚  
app.use(express.static(path.join(\_\_dirname, 'public'))); //对静态文件的支持，

第一个是favicon图标的加载，放在public目录之下；

第二个是日志的显示，比如



第三个、第四个是请求的解析，按照我查到的内容，由于request只能被解析一次（解析后就被消耗掉），因此先按json来解析，如果json解析不了，就按第四个的方式解析（但我不清楚第四个的解析模式是什么）；

第五个根据名字来看，是cookie的解析，具体不清楚；

第六个是对静态文件的解析，可以理解为，把public当做根目录，如果要访问public目录下的文件，可以直接通过url，例如：http://127.0.0.1/aaa.html 来访问public/aaa.html这个文件；

**第四部分，不同目录的处理：**

**app**.use('/', routes); //假如是根目录，那么交给routes.js来处理；  
**app**.use('/users', users); //假如是/users目录，交给users.js来处理

这里的routes是在上面来请求的，其文件为routes文件夹下的index.js

而users是routes文件夹下的users.js

也就是说，如果直接访问根目录（之后没有path），如

http://127.0.0.1/

或者

http://127.0.0.1/?id=aaa

index.js会处理他（具体怎么处理见下面）；

如果访问http://127.0.0.1/users 这样的，那么交给user.js来处理他；

如果有其他url，他会在public目录下找符合的文件名，如果无法找到，那么会显示404提示



**第五部分，错误处理：**

// 以下两个是对错误信息进行处理  
app.use(function (req, res, next) {  
 var err = new Error('Not Found');  
 err.**status** = 404;  
 next(err);  
});  
if (app.get('env') === 'development') {  
 app.use(function (err, req, res, next) {  
 res.**status**(err.**status** || 500);  
 res.render('error', { //将被返回给用户  
 **message**: err.**message**, //这里是错误消息，比如Not Found  
 **error**: err //这里是错误内容  
 });  
 });  
}

④也许还很晕，但无论如何，我们可以知道，当访问根目录时，这段路由代码

**app**.use('/', routes);

将我们指向了index.js文件。

下来让我们看index.js文件：

var express = require('express'); //调用express模块  
var router = express.Router(); //调用模块的Router方法  
  
/\* GET home page. \*/  
router.**get**('/', function(req, res, next) { //req是请求，res是回应  
 res.render('index', { **title**: 'Express' }); //回应调用render方法  
});  
  
module.exports = router;

【1】前两行代码是调用expres模块的Router方法。但我们并不知道他做了什么，不过没关系，我们只需要知道他会做什么就好了。

【2】router.get这个方法，是获取根目录的访问，然后给予回应；

回应内容，虽然我们不明白，不过没关系，我们只需要知道他调用的是views/**index**.jade就行了。

假如我们要调用该目录下的其他文件，那么只需要修改index就可以了。

例如，我们要访问views/test/index1.jade文件，那么这么写：

res.render('test/index1', { **title**: 'Express' });

当然，实际运用中并不是这样写就一切ok的，具体来说，还要修改jade文件中的路径（当然，理论上来说，你很可能不懂jade文件的语法，不过没关系）。

**简单做法是：**把index文件复制到test文件夹下，然后改名为index1.jade，打开它，修改第一行代码：

extends layout

为：

extends ../layout

即可，表示访问上一级目录中的layout.jade文件

此时我们重新运行app.js文件，就能访问到index1.jade这个模板文件了。

**（28）jade的简单说明**

正好我也是刚学，我会尽量结合实例和效果进行说明，如果有多种方式的话，我会尽量都写到。（另外，或许因为版本问题，有些结果和之前是有一定差异的，我写的都是经过我测试，目前最新版本Express所使用的jade）

①对照的两端代码：

HTML版：

<!DOCTYPE html>  
<**html**>  
<**head**><**title**>Express</**title**>  
 <**link** rel="stylesheet" href="/stylesheets/style.css">  
</**head**>  
<**body**></**body**>  
</**html**>

jade版：

**doctype html  
html  
 head  
 title**= **title  
 link**(rel='stylesheet', href='/stylesheets/style.css')  
 **body**

②说明：

【1】第一行不解释，直接这么写就行了，知道两行等价就行（等后面就明白意思了）；

【2】<html></html>变为了html；

<head></head>变成了head

head相对于html有缩进，head是html的下级标签；

**最简单直接的说明是：**html是树形结构，写在jade里，下级标签比上级标签多缩进一格；

**如果不懂的继续看下面：**

我们知道，html是树形结构的，树的根是html，

然后是head和body两个一级子；

而像title之类，又依赖于head或者body这两个一级子，成为二级子；

然后还有三级子，四级子等，无论如何，下级子肯定会有一个上级的父。

在写HTML的时候，我们为了方便识别，必然是下级子相对于上一级父要多缩进一格。而在jade里，我们也是这么写的；

根节点html不缩进，一级子head和body相对于html缩进一格（2个或者4个空格），二级的title和link标签，是head的子，因此相对于head再缩进一格；

然后依次类推，比如某个div是五级子，因此他缩进5格，而input是这个div的子，因此他缩进6格；

这样就解释了如何缩进的问题了。

【3】我们的标签往往会有属性，比如class，或者id，或者name，或者link那样的rel、href等属性。

在写成jade格式时，这些属性需要紧跟标签，比如link或者是div等；

例如：

<**link** rel="stylesheet" href="/stylesheets/style.css">

写成jade格式就是：

**link**(rel='stylesheet', href='/stylesheets/style.css')

用括号将属性包含起来，用逗号分隔不同的属性。

假如我们要写class、id、name属性，那么就这么写：

**div**(class="aaa",name="bb",id="cc") **ppp**

其表示：

<div name="bb" id="cc" class="aaa">ppp</div>

当然，我们也可以这么写：

**div**#cc.*aaa*(name="bb") **ppp**

和上面是等价的

于是，我们就会简单的jade模板写法了。

③其他说明：

**以下两个和标签写在一起（中间无空格）**

# 表示id

. 表示class，

**写在括号里的：**

class=变量名 可以是一个数组（数组成员需要是字符串）

**注释：**

// 必须写在单独一行，不能写在块内，输出在html时会以注释形式输出

如：

**h1**!= **title***//aaa***div First line.**

<!--注释内容--> 正常输出，但由于是注释，因此浏览器不会显示

**h1**!= **title  
<!--这是一个注释-->  
div First line.**

条件注释，如代码

**<!--[if IE]>  
div 注释内容  
<![endif]-->**

按照说明，条件注释只有在IE10，和以下版本的浏览器才会支持。

比如if IE 8就是指IE8。具体等实际使用的时候再行查看。

**输出非转义字符**

假如我们需要输出<script><script>这样的字符，方法有

【1】注意，不能省略引号，不然会出错，等号后可以空格也可以不空格；

**div**= "<script></script>"

【2】同样，空格可有可无（如果是!=的话，<script>将被解释为一个标签，即不显示）

- var html = "<script></script>"  
**div**= html

【3】但不能用<script></script>来替代html

- var html = "<script></script>"  
**div This is #{**html**}**

【4】使用管道“|”符号

-var m ="<script></script>"  
**p**.  
|  **#{**m**}**

应该除此之外还有其他的，不过我还没搞懂#{}，!{}，=，!=之间的区别

**（29）添加路由**

①之前我们有这么一段代码：

**app**.use('/', routes); //假如是根目录，那么交给routes.js来处理；  
**app**.use('/users', users); //假如是/users目录，交给users.js来处理

当访问根目录的时候，调用routes；当访问的是users路径是，由users来处理；

然后又知道，当访问其他路径时，会这么处理：

**app**.use(express.static(path.join(\_\_dirname, 'public')));

即去查看public文件夹下有没有对应的静态页面，如果有，则显示。

那么假如以上都不符合呢？那么会报错。

显然，我们会有这样一种需求，假如当访问/test这样一个路径时，我们需要对她进行一次特殊的处理，例如发送访问时间给用户。

教程上是这么写的：

app.use('/test',routes.test); //假如访问的路径是test，那么行为方式是调用routes（即index.js）的test方法

但实际上，最近版本情况下，我们的index.js文件是这样的。

**router**.get('/', function (req, res, next) { //req是请求，res是回应  
 res.render('index', {title: 'Express'}); //回应调用render方法  
});

显然，并没有这样一个方法。（教程上老版本的是这样的：index = router.get(略)）

因此我们可以这么做，直接不对app.js做任何操作，但是在index.js添加这样一段代码

router.get('/test', function (req, res, next) {  
 res.send('Hello, your visit time is ' + new Date() + '.');  
})

然后重启路由，我们访问/test路径时，就会显示：

Hello, your visit time is Sun Jun 26 2016 15:38:49 GMT+0800 (中国标准时间).

但必须明白这个机制是什么。

**②路由的机制：**

**错误的认识：**

当我们按上面的写法来写的时候，就可能会产生一个错觉：

当访问/test时，由于index.js里面有

router.get('/test',

这样一段代码，无论把这段代码放在哪里，都会交给这段代码来处理访问/test这个路径。例如把这段代码放在users.js里，然后访问http://127.0.0.1/test，也可以获得这样的回应。

当然，事实上并非这样。

**正确的认识：**

事实上是这么做的，当访问根目录时，即http://127.0.0.1，然后他会去查看routes下（即index.js文件）有有没有这样一段代码：

router.get('/'

如果有，则调用其回调函数来进行处理。

如果没有，那么他会返回错误提示。（可能是404页面，也可能是其他错误提示）。

这里的/，指的是app.js中，基础路径：

app.use('/', routes);

的根路径。

**具体举例的话：**

假如我们有一个页面是A页面，我们计划其路径为/base/pageA

【方法一】我们可以这么做：

在app.js中，加入代码：

app.use('/', routes);

注意，这里的routes指向的是index.js

然后在index.js里加入代码：

**router**.get('/base/pageA', function (req, res, next) {  
 res.send('Hello, your visit time is ' + new Date() + '.');  
})

我们便可以通过http://127.0.0.1/base/pageA来访问到我们需要访问的网页了

【方法二】

我们也可以这么做：

app.js里添加代码

var base = require('./routes/base'); //routes文件夹下的base文件

和

app.use('/base', base);

然后在routes文件夹下新建base.js文件，输入以下代码：

var express = require('express'); //调用express模块  
var router = express.Router(); //调用模块的Router方法  
  
/\* GET home page. \*/  
router.get('/pageA', function (req, res, next) {  
 res.send('Hello, in another page, your visit time is ' + new Date() + '.');  
})  
  
module.exports = router;

此时访问网页，将会显示：

Hello, in another page, your visit time is Sun Jun 26 2016 17:08:52 GMT+0800 (中国标准时间).

**解释：**

我是这么解释的，在app.js里，use函数的第一个参数，其路径表示的是基础路径，而在第二个参数里，js文件的路径是相对路径（相对于这个基础路径的路径）。

假如基础路径是“/app”，

那么相对路径是“/”时，其路径指/app/；

而相对路径是“/test”时，其路径指/app/test；

相对路径是“/index/pageA”时，其路径指/app/index/pageA

注意，根据我测试，似乎不能使用“../test”来访问/test这样的路径

**假如路径冲突会如何？**

即上面的方法一和方法二同时使用，那么访问http://127.0.0.1/base/pageA的结果是什么？

实践证明，根据app.js来定

由于在app.js里，路由是有先后顺序的，因此哪个在前，就会先交给谁处理。

例如：

app.use('/', routes); //假如是根目录，那么交给routes.js来处理；  
app.use('/base', base);

是index.js相应的代码进行处理；

而

app.use('/base', base);  
app.use('/', routes);

这样的顺序，是由base.js相应的代码进行处理。

**③路径匹配：**

假如用户访问一个这样的路径：

/base/myname

其中，myname是用户名，他可能是'abc'，也可能是'defg'等（不含引号）。

那么，我们可能就需要获取这个myname。

之前我们遇见的一般是这样的情况：

/base?name=abc

我们可以通过url模块的pathname方法来确定其name属性值。

但当前需求显然和之前不同。因此我们使用另外一个方法，在base.js添加以下代码。

router.get('/:username', function (req, res, next) {  
 res.send('username: ' + req.params.username);  
})

通过这样的方式，来将myname位置的值，赋值给req的username属性

例如，我们访问：http://127.0.0.1/base/page（注意，避免有直接对该链接进行处理的代码）

会返回我们这样的内容：

username: page

根据解释，:username这样的形式，会被自动编译为正则表达式，类似：\/user\/([^\/]+)\/? 这样的形式，而其参数可以在响应函数中通过req.params属性来访问。

另外，**路径同样支持javascript的【正则表达式】**。因此可以匹配更加复杂的路径，但由于是匿名参数，因此需要通过req.params[0]、req.params[1]来访问了。

**（30）REST风格的路由规则**

①REST指Representational State Transfer（表征状态转移），是一种基于HTTP协议的网络应用的接口风格，充分利用HTTP的方法实现统一风格接口的服务。

HTTP协议定义了以下八种标准的方法：

【1】GET

【2】HEAD（请求指定资源的响应头）

【3】POST

【4】PUT

【5】DELETE

【6】TRACE（回显服务器收到的请求，主要用于测试或诊断）

【7】CONNECT（HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器）

【8】OPTIONS（返回服务器支持的HTTP请求方法）

根据REST设计模式，分别是POST增，DELETE删，GET查，PUT改

其中，GET是安全的，即不会对资源产生变动，连续访问多次结果相同；

GET、DELETE、PUT是幂等的，即重复多次操作和一次操作，效果是一样的。

而Express支持的HTTP请求的绑定函数如下：



**（31）控制权转移next**

①之前有提到，假如有同一路径有两个方法来对其进行处理，那么只有匹配到的第一个方法会被执行，剩余的将略过；

那么有什么办法可以让两个一起执行呢？那就是next

具体而言，如代码：

在index.js中添加代码：

router.**get**('/base/pageA', function (req, res, next) {  
 res.send('index.js.');  
 next();  
});

在base.js中添加代码

router.**get**('/pageA', function (req, res) {  
 //res.send('base.js!');  
 **console**.log("base.js")  
})

注意，之所以注释掉中间那一行，是因为这样写会出错，因此第一个负责返回给用户，第二个负责在控制台显示。

启动app.js，页面显示：index.js.

而控制台显示：base.js

②这种方法的优点：

【1】轻易的实现中间件。（即将A——》B的流程修改为A——》C——》B这样）；

【2】提高代码的复用性。例如，我们首先要对一个请求进行检查，如果成立，然后对其进行处理。而这个请求可能是GET，可能是DELETE，也可能是POST或者PUT。

假如每个请求的处理函数，都需要添加一段检查代码，那么显然是不美观的。因此，我们使用all来进行处理，再在all的最后调用next，路由会根据方法的不同，自动跳转到不同的处理函数上。

**（32）jade进阶**

**①较长的jade文本：**

假如有一段文本，有两行（或者实际换行了），那么在jade里我们显然是不能写在一行的；

但是又不能简单的另起一行缩进写，因为这样的话，第一个单词会被认为是标签名。如：

**div  
 Welcome to #{title},  
 this is another row in div**

显示结果为：

<div>

<welcome>to Express,</welcome>

<this>is another row in div</this>

</div>

显然不是我们想要的。因此这么做：

【方法一】

**div Welcome to #{title},** | **this is another row in div**

**每个新起的一行，都以“|”开头，缩进。**

显示结果：

<div>Welcome to Express,

this is another row in div</div>

正是我们想要的。

【方法二】

**div**.  
 **Welcome to #{title},  
 this is another row in div**

注意，标签名后以英文句号为结尾，但是后面不跟任何单词（因此不是类名）；

内容另起一行（不然同行的无法显示）。

**②jade的变量调用**

【方法一】#{表达式} ——如#{title}

【方法二】=表达式 ——返回表达式的值

【方法三】!=表达式 ——这个的具体效果，参照最后

**③if else判断语句**

【1】首先，标签的父子兄弟关系，根据if和else的缩进情况决定；

【2】if else的内部语句，是不能由“{}”包住的。

【3】标签相对于if和else要缩进一格；

【4】if和else，可以以“-”开头，外加一个额外的空格（或者直接以if、else为开头）

【5】if和else的缩进程度要相同（同样的空格数量）

【6】unless关键字表示if的判断语句结果取反。假设原判断语句是unless(a==1)，那么相当于if(a!=1)

如代码：（title为“Express”，不包含引号）

【1】例如：

extends layout  
  
**block content  
 h1**!= **title  
 div First line.** if ( **title** == "express" )  
 **div**.  
 **Welcome to #{title},  
 this is another row in div** else  
 **div another line.  
 div Last line**

输出

# Express

First line.

another line.

Last line

【2】例如：

extends layout

**block content  
 h1**!= **title  
 div First line.** - if ( **title** == "express" )  
 **div**.  
 **Welcome to #{title},  
 this is another row in div** - else  
 **div another line.  
 div Last line**

输出同上。

例如：

extends layout

**block content  
 h1**!= **title  
 div First line.** unless ( **title** == "express" )  
 **div**.  
 **Welcome to #{title},  
 this is another row in div** else  
 **div another line.  
 div Last line**

输出：

# Express

First line.

Welcome to Express, this is another row in div

Last line

需要注意，unless前面不能有“-”，这点和if else不同。

**④循环语句**

**【1】for in语句 ：**

在JavaScript中，for in语句是遍历对象的key，然后根据key值来取得value。

**在jade不同！**他在for后，有两个参数，第一个参数是遍历获取对象的value，第二个参数才是key。简单来说，如代码：

- var obj = {**a**:"111",**b**:"222",**c**:"333",**d**:"444"};  
for value, key in obj  
 **div The key is #{**key**}, the value is #{**value**}**

这段代码如果在js中，输出的是a、b、c、d这样。但是在jade里，输出的是：

The key is a, the value is 111

The key is b, the value is 222

The key is c, the value is 333

The key is d, the value is 444

说明这里的i是value，而非key。

另外，标签的兄弟关系，根据for的缩进情况而定

**【2】for循环**

需要在for之前加“-”；

for循环内部的需要缩进，其兄弟关系根据for而定；

可以改变外部变量的值；

如代码：

- var num = 0;  
- for(; num < 5; num++)  
 **div num = #{**num**}  
div At last , the num = #{**num**}**

输出内容为：

num = 0

num = 1

num = 2

num = 3

num = 4

At last , the num = 5

**【3】each in**

each in和for in几乎是相同的，（事实上我没发现有什么不同）

所以略过

**【4】case ，when**

类似switch case，但具体还是有所不同。如代码：

- var str = "test"  
**div The word is #{**str**},now is case begin.**case str  
 when "a": **div first when is a** when "test": **div second row is test** when "b"  
 **div third when is b** default: **The words is #{**str**}  
div It's end.**

结果为：

The word is test,now is case begin.

second row is test

It's end.

特点：

《1》只执行符合条件的一行（c++的switch中，是从匹配那行开始执行到结尾）

《2》两种写法，标签写在when后面的，用冒号加空格分隔；或者另起一行然后缩进写。

《3》匹配失败的情况下，执行default那行的代码

**⑤过滤器  
没看懂，待补充**

**⑥Mixin：**

简单来说，就是jade里，某一个模块可以复用。比如有一行<div>A line</div>，

如果我想让他连续出现100次，我可以直接使用for循环；

但假如我需要出现30次，然后插入一行代码，再出现10次，再插入一行代码，最后出现60次呢？当然，你可以说可以写3个for循环。

那么假如这块div有100行长呢？复制这100行代码去三个地方么？虽然也可以，但显然不美观。

因此解决办法是，使用mixin，具体看代码：

- var num = 0;  
mixin node  
 **div The number in mixin node is #{**num++**}**+**node**()  
+**node**()  
+**node**()  
**div At last, the number in mixin node is #{**num++**}**

解释：

《1》node是一个html块，可以理解为声明一个html块，在声明时是不会被展现在页面上的。

《2》+node() 是调用这个html块，可以理解为将这个html块插入到某一个位置。他会实质的对变量、页面产生影响。

《3》记得缩进！

《4》显然是可以用for循环的，比如这样

-for (var i = 0 ; i < 10;i++)  
 +**node**()

**⑦模板继承和包含**

假如，我们有A模板，然后B、C模板都要复用A模板，我们有两种做法：

【1】在A模板里进行判断，然后决定调用B模板还是C模板；

【2】在B模板和C模板里调用A模板。

前者的缺陷在于，假如有不定数个模板要调用A模板，那么A模板会很复杂，所以弃用。

因此，我们通常采用方法【2】。

具体做法有：

**【方法1】我们需要把B模板嵌套在A模板中：**

**使用extends**

在layout.jade里有如下代码：

**block content**

注意：block是关键字，不能被修改，content可以被修改

在index.jade里这么写：

extends layout  
**block content  
 div It's index.jade  
block content1  
 div It's another line**

第一行表示导入layout.jade

第二行表示将第三行（直到遇到第四行这样的代码之前的所有代码），替换到layout中的block content处。

如果在layout中是content1，那么将会调用第四行，将第五行导入其中。

**【方法2】我们需要把A模板放在B模板的某个位置**

**我们需要调用include来做到**

假如我们有这么一段代码：（当然，为了方便，这里写的很简单）

**div The block in test**

我们将其放在test.jade之中

当需要在其他地方调用时，我们这么写：

**div Will include test.jade after this**include **test**

通过这样的显示，成功调用了test.jade中的模板

显示：

Will include test.jade after this

The block in test

注：当在不同路径时，例如在views文件夹（这是当前模板所在文件夹）中的test文件夹下的test1.jade文件，我们这么写就ok了：

include **test**/test1

**⑧缩进的标签：**

需求：

<div>

<p>a line with two indentation</p>

</div>

【方法一】

**div**: **p a line with two indentation**

注意，第一个div后面有一个英文冒号，然后空格再是p标签

【方法二】

**div  
 p a line with two indentation**

用缩进来控制

**⑨输出和非输出：**

【1】假如我们需要输出一段，那么就使用普通的方式写代码；

【2】假如我们不需要输出一段代码，例如我们对变量的操作，那么就用“-”开头（不含引号）。写在这里的代码将会被正常执行，但是不会被输出到浏览器上。

**⑩转义和非转义**

以下每段代码都是这段代码开头：

- var html = "<script></script>"

【1】

**div**= html

输出

<script></script>

注意，以上是**文字形式**

【2】

**div**!= html

输出

<script></script>

注意，以上是**标签形式**

【3】

**div html**

输出：

html

总结：

**无任何**：直接将变量名解释为文字；

**有等号：**将变量名解释为其值

**!=的形式：**将变量名解释为其值，然后对其值进行二次解释（解释为html语言）

**（33）用cookies、session来记录是否首次访问**

转自

https://github.com/alsotang/node-lessons/tree/master/lesson16

这里是时间，单位应该是毫秒

**①cookie版**

var express = require('express');

// 首先引入 cookie-parser 这个模块

var cookieParser = require('cookie-parser');

var app = express();

app.listen(3000);

// 使用 cookieParser 中间件，cookieParser(secret, options)

// 其中 secret 用来加密 cookie 字符串（下面会提到 signedCookies）

// options 传入上面介绍的 cookie 可选参数

app.use(cookieParser());

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

// 如果请求中的 cookie 存在 isVisit, 则输出 cookie

// 否则，设置 cookie 字段 isVisit, 并设置过期时间为1分钟

if (req.cookies.isVisit) {

console.log(req.cookies);

res.send("再次欢迎访问");

} else {

res.cookie('isVisit', 1, {maxAge: 60 \* 1000});

res.send("欢迎第一次访问");

}

});

②session版

【1】内存版（express默认将session存储在内存中）

var express = require('express');

// 首先引入 express-session 这个模块

var session = require('express-session');

var app = express();

app.listen(5000);

// 按照上面的解释，设置 session 的可选参数

app.use(session({

secret: 'recommand 128 bytes random string', // 建议使用 128 个字符的随机字符串

cookie: { maxAge: 60 \* 1000 }

}));

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

// 检查 session 中的 isVisit 字段

// 如果存在则增加一次，否则为 session 设置 isVisit 字段，并初始化为 1。

if(req.session.isVisit) {

req.session.isVisit++;

res.send('<p>第 ' + req.session.isVisit + '次来此页面</p>');

} else {

req.session.isVisit = 1;

res.send("欢迎第一次来这里");

console.log(req.session);

}

});

其中，console.log的session结果如下：

Session {

cookie:

{ path: '/',

\_expires: Tue Jun 28 2016 15:55:37 GMT+0800 (中国标准时间),

originalMaxAge: 6000,

httpOnly: true },

isVisit: 1 }

**（34）用连接池来控制mysql（入门版）**

①第一步，创建一个连接池：（和之前普通创建mysql的连接对象很像）

var mysql = require("mysql");  
var pool = mysql.createPool({  
 **host**: '127.0.0.1',  
 **user**: 'root',  
 **password**: '',  
 **port**: '3306',  
 **database**: 'test'  
})

我们是将其单独放在一个文件，这样的话，如果以后修改mysql的连接属性（比如地址、用户名、密码、端口或者表），就不用满世界找这些东西了，只需要修改这个文件即可。

②第二步，用连接池来连接：

在连接之前，需要说明的是，之前的方法，是三步：连接（connect）、请求（query）、结束（end）。

在使用连接池的时候，也是三步：连接（getConnection），请求（query）、断开连接（release）。但区别在于，query和release可以写在getConnection这个函数里面。

具体如代码：

var db = {};  
db.con = function (callback) { //callback是回调函数，连接建立后的connection作为其参数  
 pool.getConnection(function (err, connection) {  
 **console**.log("connect start...")  
 if (err) { //对异常进行处理  
 throw err; //抛出异常  
 } else {  
 callback(connection); //如果正常的话，执行回调函数（即请求）  
 }  
 connection.release(); //释放连接  
 **console**.log("connect end...")  
 })  
}

第一步是创建一个空对象db，为了封装，我们把代码都放在db这个对象之中，然后导出的时候只导出db，这样可以避免对连接池属性进行误操作。

第二步是创建一个函数con作为db这个对象的一个方法，然后回调函数作为参数传递给这个con方法，也就是说，当我调用函数时，实际上是这么写的：

db.con(function (connect) {  
 函数内容暂略  
})

这个作为参数的函数，将在con这个函数的内部被执行。

如果不太明白的话，稍后再说；

在con函数里，执行了pool.getConnection这行代码，其作用是连接mysql数据库。

他有一个回调函数，这个回调函数有两个参数，

第一个参数是err，如果出错的话，err将有值，应该对其进行处理；

第二个参数是connection，他实际上就是指连接成功后的对象，用其进行query处理，就像这样：

connection.query(mysql的语句, 可能的参数, 回调函数);

然后也用其进行释放连接：

connection.release();

**console**.log("connect start...")

回调函数的第一行代码是consoloe.log，通报连接开始；

**console**.log("connect end...")

最后一行代码通报连接结束（事实上连接可能还没有结束，因为他只是getConnection的回调函数，在其本身内部可能还有一些处理，但我们进行的处理已经结束了）；

if (err) { //对异常进行处理  
 throw err; //抛出异常  
} else {  
 callback(connection); //如果正常的话，执行回调函数（即请求）  
}  
connection.release(); //释放连接

这里，分别是：

【1】对异常进行处理；

【2】正常情况下的处理；

【3】释放连接；

正常的处理，将连接作为参数，传递给上面mysql.con函数的参数，作为mysql.con的参数的参数进行处理。因此实际上其流程是这样的：

函数被调用（这里指con这个自定义的函数）

——》回调函数作为参数传入（**con(callback)**）

——》回调函数获得connection作为参数，并执行自己**callback(connection)**

——》回调函数内部调用自己的第一个参数（事实上就是connection）进行请求（connection.query）；

③导出封装好的db对象（其只有一个方法，那就是con）；

module.exports = db;

④假设我们把以上代码放在一个叫做db.js的文件之中。然后我们需要引用他：

var db = require('../db')

⑤然后比如我们需要请求一个数据：

**db**.con(function (connect) {  
 connect.**query**('SELECT \* FROM bloguser WHERE username = ?', [username], function (err, result) {  
 if (err) {  
 **console**.log("select username:" + username + " error, the err information is " + err);  
 return  
 }  
 **console**.log(result);  
 })  
})

注意，我们所有请求数据的内容，都写在了这个con的回调函数中，由于封装好了，因此我们不关心他如何去连接数据库，如何断开。只需要关心请求这一步即可。

query是其自带的方法，因此我们按要求调用即可；

在请求后，我们依然是分别对err和结果进行处理；

假如err，提示错误，然后返回（这里的返回指的是返回在调用这个db.con的方法的地方）；

假如正确，那么console.log结果。

⑥当这个时候，我们利用连接池来请求mysql数据库的行为就完毕了。

下面附上完全的代码：

db.js

var mysql = require("mysql");  
var pool = mysql.createPool({  
 **host**: '127.0.0.1',  
 **user**: 'root',  
 **password**: '',  
 **port**: '3306',  
 **database**: 'test'  
})  
var db = {};  
db.con = function (callback) { //callback是回调函数，连接建立后的connection作为其参数  
 pool.getConnection(function (err, connection) {  
 **console**.log("connect start...")  
 if (err) { //对异常进行处理  
 throw err; //抛出异常  
 } else {  
 callback(connection); //如果正常的话，执行回调函数（即请求）  
 }  
 connection.release(); //释放连接  
 **console**.log("connect end...")  
 })  
}  
module.exports = db;

查询的js文件（需要修改表名，字段名，建议练手时把查询条件写死，能正常运行后再修改）：

var db = require('../db')

db.con(function (connect) {  
 connect.**query**('SELECT \* FROM bloguser WHERE username = ?', [username], function (err, result) {  
 if (err) {  
 **console**.log("select username:" + username + " error, the err information is " + err);  
 return callback(err);  
 }

**console**.log(result);  
 })  
})

**（35）express框架的send方法**

①send方法用的还挺多的，因此需要明确其作用；

②原型是：

**res.send([body|status], [body])**

即既可以直接发送内容，也可以第一个参数状态，第二个参数内容。

如果直接发送内容的话，状态会被自动补全；

③发送的内容：

示例：

res.send(new Buffer('whoop'));

res.send({ some: 'json' });

res.send('some html');

res.send(404, 'Sorry, we cannot find that!');

res.send(500, { error: 'something blew up' });

res.send(200);

【1】第一种是发送二进制（binary）内容，当其参数为Buffer（缓冲）时，Content-Type 会被设置为 "application/octet-stream" ，而这个表示其文件后缀（文件类型）是某些类型，具体可以查看：

http://www.w3school.com.cn/media/media\_mimeref.asp

而wiki上是这么说的

* application/octet-stream（任意的二进制数据）

也就是说这是一个任意的二进制数据，具体如何解释要看实际情况（比如后缀名），比如他可能是一个img，也可能是一个video。

【2】假如发送字符串，那么将被解释为html文件；

也就是说，Content-Type 默认设置为"text/html":

例如，发送了一个post，然后我res.send("aaa")，那么网页将跳转到一个只有文本aaa的页面；

【3】假如参数为Array（数组），或者Ojbect（对象），那么将返回一个JSON；

**（36）express的render方法**

①当我们需要返回一个模板时（无论是jade或者是ejs），都需要调用render方法；

②其原型为：

**res.render(view, [locals], callback)**

第一个参数view是模板（比如某个jade文件）；

第二个参数是一个对象，比如你要传递某个值给模板，就靠这个；

第三个参数是回调函数，如果不填写的话，那么就直接把页面发给客户端；

如果发生错误，那么回调函数的next(err)将被执行（注意，这里是执行的是默认的响应），如果写了回调函数，那么错误和渲染的字符串将作为参数提供给回调函数执行，而不是由默认的响应执行。

**（37）使用MD5加密**

①当我们存储密码时，最好不要存储明文的密码（因为假如数据库泄露，或者发生其他问题，会导致密码丢失）；

②因此，我们可以存储md5加密后的密码。匹配密码是否正确时，使用加密后获取的字符串和存储的密码（也是被加密后的）进行对比，如果未加密前值是一样的，那么加密后值也是一样的。

③方法是：

var md5 = crypto.createHash('md5');  
var password = md5.update(req.**body**.**password**, 'utf-8').digest('base64');

第一步获取md5这个对象；（表示调用这个对象的方法，加密的形式是MD5）；

第二步是调用md5的update方法，update的第一个参数是被处理的内容，第二个是可选的，但如果要对中文进行处理，那么就需要加上'utf-8'这个参数；

而digest指的是以什么形式进行编码，这里选择了base64，按照说明，还有hex（十六进制）、binary（二进制）这两种方法；

而password这个返回值，就是我们进行md5加密后的内容了，他是一串挺长的字符串。

**（38）动态视图助手**

①作用：假如我们需要一个变量，在不同地方的模板（jade文件）都需要调用。

显然，我们不应该使用全局变量（因为可能会带来污染）；

事实上，我们需要的是仅仅在模板中起作用的变量，因此视图助手的作用就在这里了。

②流程：

【1】首先，调用express模块，就像我们之前做的那样（事实上，不需要额外声明，当我们使用express框架的时候自然会调用它）；

【2】按照正常情况，我们需要进行路由处理，当使用视图助手时，有一件很重要的事情，就是在进行任何路由处理之前使用视图助手。

原因在于，视图助手的处理和路由处理是类似的，他会拦截req和res请求，如果先进行路由的话，那么视图助手是不会被执行的。（即使被执行了，那么由于进行路由处理的时候，没有获取到视图助手的变量，也是不会达成预期目的的）；

【3】当明确我们视图助手声明的位置后，我们开始写我们的视图助手：

注意，这里是**动态**的视图助手；

var app = express(); //生成一个导入的express模块的实例  
//定义一个动态视图助手，这段代码要放在路由之前  
app.use(function (req, res, next) {  
 //这个是示例，定义一个动态视图助手变量  
 res.**locals**.**testStr** = "这是一个动态视图助手变量";  
 //这是示例，定义一个动态视图助手方法  
 res.**locals**.testFun = function () {  
 return "这是一动态视图助手——函数的返回值"  
 };  
 //这是另一种定义的方法这种方法似乎不行  
/\* res.locals({  
 testStr2: "动态变量2",  
 testFun2: function () {  
 return "动态函数2"  
 }  
 })\*/  
 //必须有next，不然后续的路由没法执行了（因为只被执行一次）  
 next();  
})

写法很简单，app.use方法，参数是回调函数；

回调函数的参数分别是req（请求），res（回应），next（下个函数）；

然后把变量放在res.locals之中，声明一个变量testStr，给他赋一个值。

当我们在模板里调用这个变量时，直接使用testStr就可以。

同理，也可以声明一个函数testFun，调用时使用testFun()来获取其返回值（当然，可能在这个函数里对testStr进行修改）。

【4】按照我找的教程，下面被注释掉的方法应该也可以，但事实上，我自己测试时，他会提示res.locals is not a function，也就是说，不能正常使用。因此我注释掉了。

【5】最后调用next()，正常执行路由（否则后面路由的代码不会被执行）。

【6】当执行render方法时，正常执行，但是在模板里，可以调用对应的变量。如果想在render里使用变量的话，那么需要这么写：

router.get('/', function (req, res, next) { //req是请求，res是回应  
 res.render('index', {  
 **title**: res.**locals**.**testStr** });  
});

【7】如果在jade模板里调用的话，只需要这么写就可以了：（就像之前引用变量那样写就行）

**h1 #{title}**

**（39）mysql的参数化查询的报错解决办法**

假如有这样一段查询代码：

connect.**query**("SELECT \* FROM blogtext limit 0,9", [count], function (**err**, result) {

如果count是string类型，那么可以正常运行；

假设count是数字类型number，那么会报错；

解决办法：

count更改为count.toString();

如下列代码：

connect.**query**("SELECT \* FROM blogtext limit 0,9", [count.toString()], function (**err**, result) {

**（40）用Nodejs 4.X版本，制作一个微博网站**

【0】涉及到的框架、引擎、数据库：

**① express 4.X**

**② jade**

**③ mysql 5.7.x**

**注：**

①内容较长，我会尽力把整个框架、结构、顺序、思考方式说明白。

②基础是《node.js开发指南》这本书，作者：BYVoid。但他书的版本较老，很多东西现在已经无法应用，故进行更新，使用目前普遍的**express 4.x**版本（原书似乎是2.X版本），**mysql**（原书是mongodb），**jade**（原书是ejs）

【1】基本需求：

①有首页；

②支持注册；

③支持登录、登出；

④可以发表博客，发表的博客可以保存到数据库；

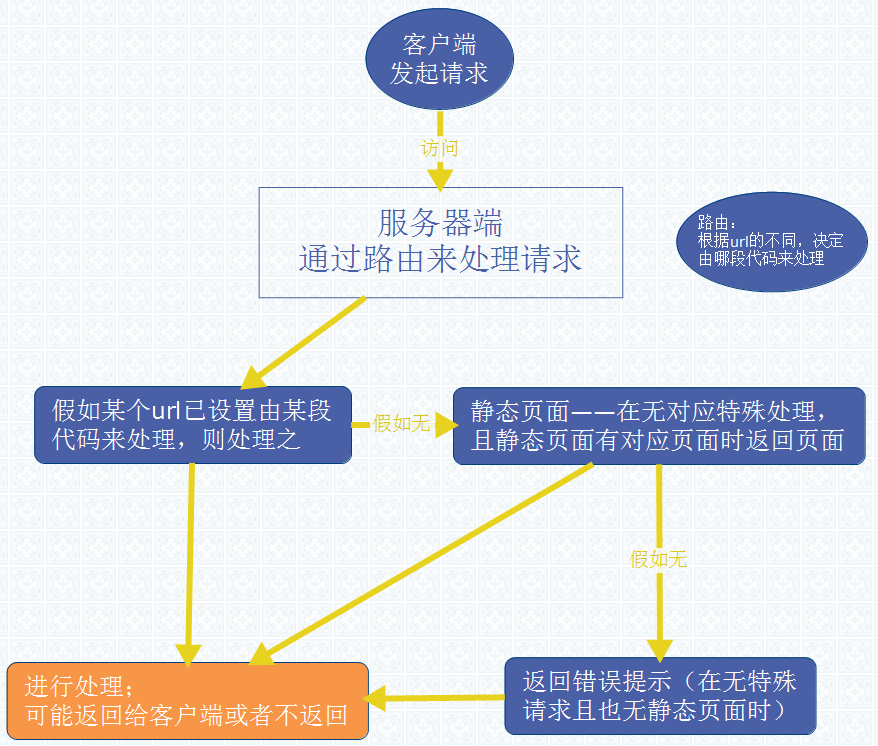
⑤可以查看博客，博客从数据库中读取，为了简化，这里设置为查看所有人的博客；

⑥查看博客时，初始查看的数量有限，但可以无限加载新的博客，直到查看完全部博客；

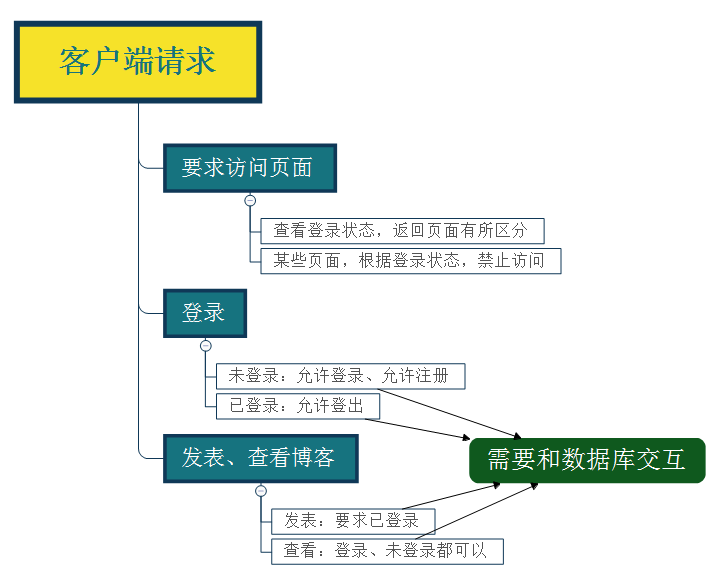
⑦一定程度上实现多语言（即可以切换显示的语言版本），但由于复杂度所限，因此只部分实现（但框架已建立好，可以通过继续完善代码实现整体的国际化）；

⑧根据登录状态，对可以访问的页面进行控制（某些允许某些禁止）

【2】前后端交互的简单过程：



【3】关于客户端请求的几种情况（涉及到数据库的）



【4】npm

npm是包管理器，在新版本里是默认安装好的，可以输入：

npm -v

来查看npm的版本

【5】express框架：

首先安装基础的express框架，他是封装好的web开发框架，里面包含了：

①路由控制、

②log显示、

③解析客户端请求、

④cookies解析、

⑤静态文件的控制

等多种功能。

**安装前注：**

**①有的人只需要简单的**

npm install -g express

npm install -g express-generator

就可以愉快的跑起express了，有的人就像向我一样苦逼，尝试各种办法，最后勉强可以用。

如果在这两行命令后，输入（V是大写的）

express -V

会返回版本号，那么直接跳到最后来看，如果不是这样，可以参考第②条。

或者直接看后面的终极解决方案

**②express设置全局方法：**

ln -s /usr/nodejs4.4.7/node-v4.4.7-linux-x64/bin/express /usr/local/bin/express

**其他需要全局的方法，理论上同理，即将nodejs安装目录下的bin文件夹下的模块，映射到/usr/local/bin/同名文件即可**

**③express命令可以使用的人：**

输入：

express -t jade myblog

效果是建立一个文件夹名为myblog的文件夹，里面会有一些文件。

正常如下图：



cd myblog

npm install

这时，npm会根据package.json来安装一些东西。

按提示输入：

SET DEBUG=myblog:\*

npm start

可以启动项目。

本机的话，通过访问http://127.0.0.1:3000/ 来查看效果

服务器的话，访问其公网ip，端口是3000

查看package.json

**"scripts"**: {  
 **"start"**: **"node ./bin/www"**},

这条属性告诉我们，需要通过bin文件夹下的www来启动，这也就是上面npm start命令的作用。

www文件应该是设置自启动的，然而我这里并不需要。但若直接启动app.js是启动不了的，因为是在www文件里面，设置的监听端口（3000）。

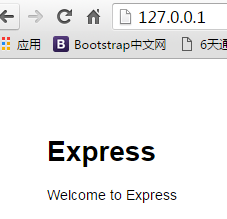
解决办法：

在app.js里面，在最后一行代码之前，添加：

app.listen(80);

于是，便可以通过app.js来启动服务器了。

访问效果如图：



假如如果像我一样倒霉，无法用express命令，打开npm也特别慢，可以先找个系统，将这些文件下载好，然后将这些文件复制到不可以用express命令的linux系统下面。

或者看最后的终极解决方案

ps：

如果npm很慢的话，可以考虑装cnpm

npm install -g cnpm --registry=https://registry.npm.taobao.org

然后在npm install这一步，使用cnpm install来替代

————————分割线————————

**④超级终极解决方案：**

我传一个装好express的压缩包，直接解压缩后就可以用。

链接：

<http://download.csdn.net/detail/qq20004604/9587054>

**⑤另一个启动方法**

即不通过app.js来启动；

在之前④的基础上，打开app.js，删除

app.listen(80);

打开package.json，将

**"start"**: **"node ./app.js"**

改回

**start"**: **"node ./bin/www"**

然后

cd bin

vi www

将

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

修改为

var port = normalizePort(process.env.PORT || '80');

然后

cd ..

npm start

启动成功，可以直接访问地址来访问页面

**【6】把启动的app.js放在后台运行：**

原本使用npm start的地方，输入

nohup npm start&

即可

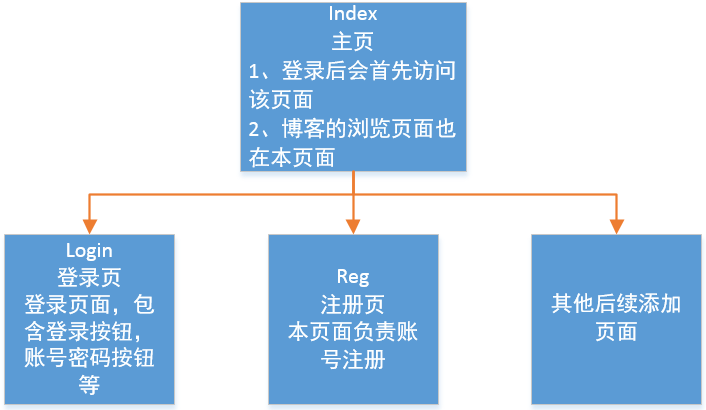
**【7】app.js之解释**

app.js可以说是一切的根本，因此请看注释，解释了每行代码的作用：

var express = require('express'); //这个模块在node\_modules文件夹中  
var path = require('path'); //nodejs自带的，路径相关  
var favicon = require('serve-favicon'); //node\_modules文件夹中  
var logger = require('morgan'); //日志相关，node\_modules文件夹中，具体看第19行代码  
var cookieParser = require('cookie-parser');//node\_modules文件夹中，用来解析cookie的，被express所调用  
var bodyParser = require('body-parser'); //用于解析请求体的（应该），被express所调用  
  
var routes = require('./routes/index'); //这里是默认的路由，路径是routes/index.js  
var users = require('./routes/users'); //这里是默认的路由，路径是routes/users.js  
//关于路由的更多内容看下面的app.use('/', routes);和app.use('/users', users);的注释  
  
var app = express(); //生成一个express的实例  
  
//设置模板引擎  
app.set('views', path.join(\_\_dirname, 'views')); //\_\_dirname表示绝对路径  
//console.log(\_\_dirname) //可以取消这一行注释，然后看看\_\_dirname的效果  
app.set('view engine', 'jade'); //表示express是使用jade格式的模板的  
  
  
//下面这行代码是设置左上角的小图标的，将其命名为favicon.ico并放在public文件夹下，如果有的话，取消这行注释  
//app.use(favicon(path.join(\_\_dirname, 'public', 'favicon.ico')));  
  
app.use(logger('dev')); //设置日志显示，如果不需要的话可以注释掉这行代码，建议练手的时候不要注释掉  
//可替换参数有default，combined，common，short，tiny，dev(相对tiny有染色）。可以自己试试效果，默认是dev  
  
app.use(bodyParser.**json**()); //解析客户端的请求，通常是通过post发送的内容  
app.use(bodyParser.**urlencoded**({**extended**: false})); //另一种解析方式，具体不太明白（如果上面解析失败的话会通过这个来）  
app.use(cookieParser()); //cookies的解析  
app.use(express.**static**(path.join(\_\_dirname, 'public'))); //普通静态html文件、js文件、css文件，都放在public文件夹下，可以直接通过url来访问  
  
//路由的处理  
app.use('/', routes); //假如访问的网址是根目录，例如http://121.41.66.68/，交给routes这个js文件来处理，具体请查看routes  
app.use('/users', users); //假如访问的是/users这样的路径，那么交给users这个js文件来处理，具体略  
//我们最后要对这个路由进行处理，让他按照我们想要的方式来做  
  
//这里对非法路径进行处理的，next表示这是一个中间件（即执行完他之后，还会执行下一个，而不是直接在这里结束了）  
//如果上面没有静态文件（29行）、没有找到被路由处理的文件（32，33行），就会交给这个来处理。  
app.use(function (req, res, next) {  
 var err = new Error('Not Found');  
 err.**status** = 404;  
 next(err); //由下面的2个app.use中的一个来处理（一般是第一个，除非第一个被注释）  
});  
  
  
//原注释说是会对错误进行加亮处理  
//这部分和下面的区别在于，这部分会将错误信息暴露给用户，而下面的不会，因此注释掉这部分  
//if (app.get('env') === 'development') {  
// app.use(function (err, req, res, next) {  
// res.status(err.status || 500);  
// res.render('error', {  
// message: err.message,  
// error: err  
// });  
// });  
//}  
  
// production error handler  
// no stacktraces leaked to user  
app.use(function (err, req, res, next) {  
 res.status(err.**status** || 500);  
 res.render('error', { //这里的error指调用views下的error模板  
 **message**: err.**message**,  
 **error**: {}  
 });  
});  
  
//这是我自行添加的，用于显示服务器启动的时间  
**console**.log("Server start at :" + new Date());  
  
//导出app，在./bin/www里会被调用  
module.exports = app;

**【8】页面关系结构**

请忽略文件名的大小写问题（手动微笑）



因此，首先有三个需要我们自定义的路由：

①当访问根/时，输出index；

②当访问/login时，输出login

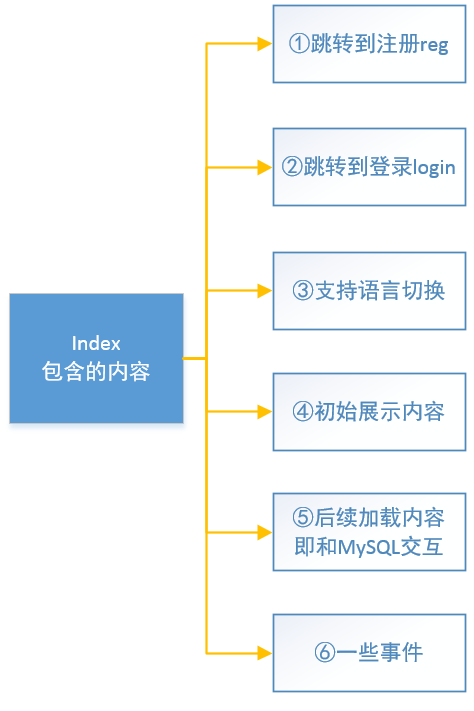
③当访问/reg时，输出reg

其他可能的路由：

④后续可能添加的页面，暂缺；

⑤不符合要求的页面，输出404（express已经完成）

【9】index页面需要包含的功能：



**【10】index的路由：**

查看app.js中的代码：

app.use('/', routes);

这部分代码已经说明了，当访问根的时候，交给routes来处理；

再次查看导入routes的地方：

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

说明负责这部分的文件是routes文件夹的index文件

这时打开index.js，

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

这部分代码，表示当访问根的 ’/’ 页面时，由index这个引擎模板来处理，模板中的title变量，其值为Express。

————————————————————————

**代码说明：**

router.**get**('/'

以上这段代码是在路由基础上进行二段捕获url的，举例：

①假如在app.js里，

app.use('/',

然后在路由的处理里，是

outer.**get**('/'

那么最终针对的url是/

②假如在app.js里，

app.use('/test',

然后在路由的处理里，是

outer.**get**('/'

那么最终针对的url是/test

③假如在app.js里，

app.use('/,

然后在路由的处理里，是

outer.**get**('/test'

那么最终针对的url依然是/test

④假如在app.js里，

app.use('/testA,

然后在路由的处理里，是

outer.**get**('/testB'

那么最终针对的url是/testA/testB

⑤捕获优先度说明：

首先根据app.js中的app.use出现顺序决定，假如②和③同时出现，那么看②和③在app.js中的代码谁先出现，就交给谁来处理；

再不调用next()方法的情况下，只会被最先出现的那个处理；

如果第一个调用了next()方法，那么第一个处理完后会执行第二个的

————————————————————————

index.js代码说明：

var express = require('express'); //调用express  
var router = express.**Router**(); //生成express的Router方法的一个实例  
  
//处理函数  
router.**get**('/', function (req, res, next) { //捕获根url  
 res.render('index', {**title**: 'Express'});  
 //res.render方法渲染一个引擎模板，  
 //第二个参数是一个对象，对象里的变量可以在引擎中使用，  
 //第三个参数是回调函数，提供两个参数，分别是err和html，err是错误，html是渲染后的页面。如果使用这个回调函数，那么将不会自动响应，即要用户自己写返回html的命令  
});  
  
module.exports = router;

注意，被引擎渲染的文件，默认是在myblog/views文件夹下

**【11】使用Bootstrap界面**

让我们自己设计界面不是不可以，不过太麻烦，因此我和原书保持一致，使用Twitter Bootstrap风格的页面。

下载他的压缩包文件，并解压缩他，

将css文件放在项目myblog/public/stylesheets下；

将img文件夹直接复制粘贴在myblog/public下；

将js文件放在myblog/public/javascripts下，

我的Bootstrap的版本是v2.3.2

另外，下载jquery，命名为jq.js，放在myblog/public/javascripts下

**【12】index.jade说明**

下来我们就要修改jade文件了，如果不知道怎么使用的话，可以看我的博客：

<http://blog.csdn.net/qq20004604/article/details/51773574>

extends layout  
  
**block content  
 h1**= **title  
 p Welcome to #{title}**

第一行的extends layout表示他导入的layout模板（即layout.jade）

block content表示以下这部分将套入layout的content位置

内容略。

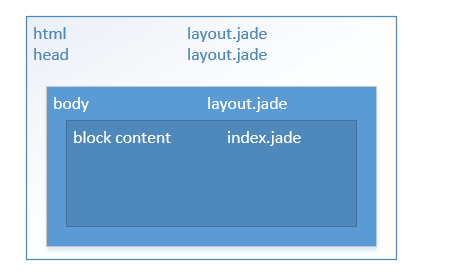
这时候再过去看layout.jade

**doctype html  
html  
 head  
 title**= **title  
 link**(rel='stylesheet', href='/stylesheets/style.css')  
 **body  
 block content**

他导入的是style.css这个样式表，页面的title是变量title

上面的index.jade中的内容将导入这里的block content区域（因为变量名一样，会对应替换）。

其结构大概如下：



然后我们根据自己实际需求来改造：

直接给出代码：

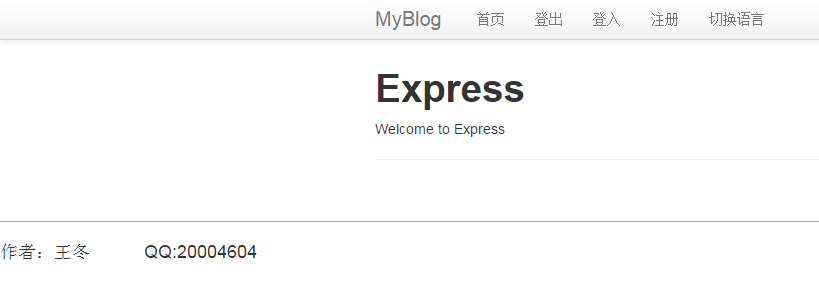
首先在stylesheets文件夹下创建一个blog.css文件，内里样式为：

body {  
 padding-top: 60px;  
 padding-botom: 40px;  
}  
  
#textarea {  
 resize: none;  
 width: 300px;  
 height: 100px;  
 cursor: text;  
}  
  
#postBlog {  
 position: relative;  
 left: 20px;  
 vertical-align: top;  
}  
  
#clearBlog {  
 position: relative;  
 left: -90px;  
 top: 27px;  
 width: 110px;  
 height: 44px;  
}  
  
.myalert {  
 position: absolute;  
}  
  
.displayNONE {  
 display: none;  
}  
  
#scrollToFoot {  
 border: 1px solid #ccc;  
 text-align: center;  
 font-size: 18px;  
 padding: 20px 0;  
}  
  
fotter p {  
 margin-top: 40px;  
 padding-top: 20px;  
 border-top: 1px solid #aaa;  
 font-size: 18px;  
}  
  
.row {  
 color: #555;  
}

其次是layout.jade

**doctype html  
html  
 head  
 title MyBlog By qq20004604  
 link**(rel='stylesheet', href='./stylesheets/bootstrap.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/bootstrap-responsive.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/blog.css')  
 **script**(type="text/javascript",src='javascripts/jq.js')  
 **script**(type="text/javascript",src='javascripts/bootstrap.min.js')  
 **script**(type="text/javascript",src='javascripts/blog.js')  
 **body  
 div**.*navbar*.*navbar-fixed-top* **div**.*navbar-inner* **div**.*container* **a**.*btn*.*btn-navbar*(data-toggle="collapse",data-target=".nav-collapse")  
 **span**.*icon-bar* **span**.*icon-bar* **span**.*icon-bar* **a**.*brand*(href="/") **MyBlog  
 div**.*nav-collapse* **ul**.*nav* **li  
 a**(href="/") **首页  
 li  
 a**(href="/logout") **登出  
 li  
 a**(href="/login") **登入  
 li  
 a**(href="/reg") **注册  
 li  
 a**(href="/language") **切换语言  
 div**#contrainer.*container* **block content  
 hr  
 fotter  
 p 作者：王冬　　　QQ:20004604**

效果如图：



然后是index.jade：

extends layout  
**block content  
 div**.*hero-unit* **h1 我的博客  
 p 这个是基于Nodejs作为后台，jade作为模板来，使用了Express作为框架  
 br  
 br** *//这部分暂时用1替代，后续会被更新* if(1)  
 **br  
 br  
 a**.*btn*.*btn-primary*.*btn-large*(href="/login") **登录  
 a**.*btn*.*btn-large*(href="/reg") **立即注册** else  
 **textarea**#textarea.*uneditable-input* **button**#postBlog.*btn*.*btn-large* **提交微博  
 button**#clearBlog.*btn*.*btn-large* **清空  
 div**#submitError.*alert*.*alert-error*.*displayNONE*.*myalert* **div**#submitSuccess.*alert*.*alert-success*.*displayNONE* **div**.*row*.*content* **div**.*span4* **h2 烟雨江南说  
 p 当欲望没有了枷锁，就没有了向前的路  
 p 只有转左，或者向右  
 p 左边是地狱，右边也是地狱  
 div**.*span4* **h2 烟雨江南说  
 p 那些都是极好极好的  
 p 可是我偏偏不喜欢  
 div**.*span4* **h2 烟雨江南说  
 p 我不怕傻  
 p 只怕  
 p 遇不到  
 p 可以让我变傻的人  
 div**.*span4* **h2 烟雨江南说  
 p 人在年轻的时候总会有些莫名的坚持，  
 p 并且以此感动着自己，  
 p 却时常会在不经意间让真正重要的东西从指间流走。  
 div**.*span4* **h2 烟雨江南说  
 p 记忆真是一种奇怪的东西，  
 p 有时候会涤荡所有的苦难，只留下温情，  
 p 有时候却磨灭掉曾有的欢乐，唯剩下苍白和丑陋。  
 div**.*span4* **h2 烟雨江南说  
 p 那存在的，都是幻影。  
 p 那永恒的，终将毁灭。  
 p 世界万物，缤纷色彩，都是被蒙蔽的人心罢了。  
 div**.*span4* **h2 烟雨江南说  
 p 诸神以真相示人，而世人却视而不见  
 div**.*span4* **h2 烟雨江南说  
 p 只有绵羊会向狮子要求平等，  
 p 而狮子们从来不会这样想。  
 div**.*span4* **h2 烟雨江南说  
 p 愿迷途的旅人，从此得享安息。  
 p 因理想而不朽，因归返而救赎。  
 div**#scrollToFoot **滚动到底部然后加载内容**

效果如图：



但此时，上面的页面切换目前还都是无效状态；

滚动然后加载内容，也正处于无效状态；

注册和登录按钮，点击后也无法正常跳转；

我们需要依次解决这些问题。

**【13】登录页面和注册页面**

接下来我们添加登录页面的路由和模板。

路由，打开app.js，在

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

之前添加

var reg = require('./routes/reg');  
var login = require('./routes/login');

之后添加：

app.use('/reg', reg); //注册的，reg.js来处理  
app.use('/login', login); //登录的，login来处理

这样的话，就添加了注册和登录页面的路由了，但目前，我们还缺其具体文件和代码。

进入routes文件夹，创建reg.js和login.js

reg.js

var express = require('express'); //调用express模块  
var router = express.**Router**(); //调用模块的Router方法  
router.**get**('/', function (req, res, next) {  
 res.render('reg')  
});  
module.exports = router;

login.js

var express = require('express'); //调用express模块  
var router = express.Router(); //调用模块的Router方法  
router.**get**('/', function (req, res, next) {  
 res.render('**login**')  
});  
module.exports = router;

现在又缺模板文件了。

进入views文件夹，创建reg.jade和login.jade

reg.jade

extends layout  
**block content  
 script**.  
 **form**.*form-horizontal* **fieldset  
 legend 注册  
 div**.*control-group* **label**.*control-label*(for="username") **用户名  
 div**.*controls* **input**.*input-xlarge*#username(type="text",name="username")  
 **p**.*help-block* **你的账户名称，用于登录和提示  
  
 div**.*control-group* **label**.*control-label*(for="password") **口令  
 div**.*controls* **input**.*input-xlarge*#password(type="password",name="password")  
  
 **div**.*control-group* **label**.*control-label*(for="password-repeat") **重复输入口令  
 div**.*controls* **input**.*input-xlarge*#password-repeat(type="password",name="password-repeat")  
  
 **div**.*form-actions* **button**.*btn*.*btn-priamry*#submit **注册**

login.jade

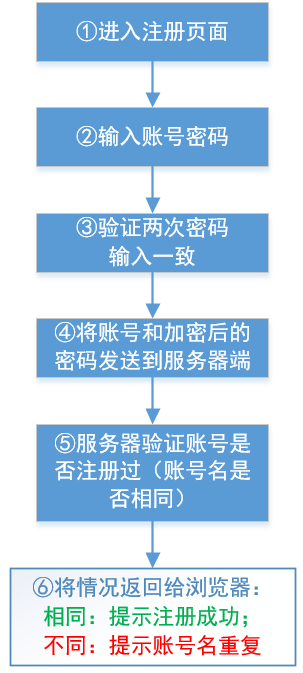
extends layout  
**block content  
 form**.*form-horizontal*(method='post')  
 **fieldset  
 legend 登录  
 div**.*control-group* **label**.*control-label*(for='username') **用户名  
 div**.*controls* **input**.*input-xlarge*#username(name='username',type='text')  
 **div**.*control-group* **label**.*control-label*(for='password') **密码  
 div**.*controls* **input**.*input-xlarge*#password(name='password',type='password')  
 **div**.*form-actions* **button**.*btn*.*btn-primary*(type="submit") **登录**

这个时候，注册和登录页面已经可以访问了（虽然还没有添加逻辑）

**【14】在和MySQL交互之前的准备工作**

首页、注册、登录页面的页面布局已经做完，下来我们需要完成注册和登录功能。

注册功能的流程：

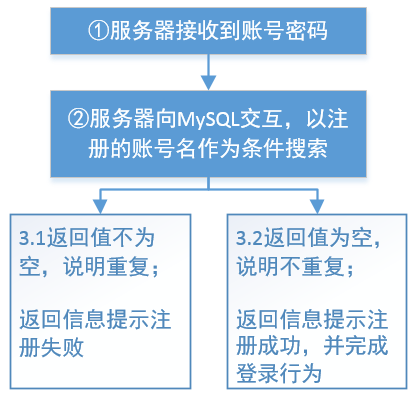


第①和②略过；

第③步由js完成，因此要对reg.jade进行修改，同时也需要添加处理登录逻辑的js代码；

第④步在第③步验证一致后发起ajax请求，和第③步一并完成；

第⑤步由后端完成，具体流程为：



第⑥步是ajax请求的回调函数，在屏幕上提示不同的内容。

也可以通过服务器重定位，推送一个提示注册成功的页面。

第③步修改：

**reg.jade**

extends layout  
**block content  
 script**(type="text/javascript",src='javascripts/reg.js')  
 **div**.*form-horizontal* **fieldset  
 legend 注册  
 div**.*control-group* **label**.*control-label*(for="username") **用户名  
 div**.*controls* **input**.*input-xlarge*#username(type="text",name="username")  
 **p**.*help-block*#tips **你的账户名称，用于登录和提示  
 p**.*help-block*.*displayNONE*#usrname-error **用户名不能为空  
 p**.*help-block*.*displayNONE*#error-alert  
  
 **div**.*control-group* **label**.*control-label*(for="password") **口令  
 div**.*controls* **input**.*input-xlarge*#password(type="password",name="password")  
 **p**.*help-block*.*displayNONE*#pw-error **密码不能为空  
  
 div**.*control-group* **label**.*control-label*(for="password-repeat") **重复输入口令  
 div**.*controls* **input**.*input-xlarge*#password-repeat(type="password",name="password-repeat")  
 **p**.*help-block*.*displayNONE*#pw-rp-error **两次密码需要一致  
  
 div**.*form-actions* **button**.*btn*.*btn-priamry*#submit-button **注册**

在这里，取消掉原生的表单提交功能，更改为用ajax发起请求。

注册页面的逻辑单独写在一个js里面，因此在public/javascripts下创建文件：

**reg.js**

**$**(**document**).ready(function () {  
 var time = 0; //用于计时  
 **$**("#submit-button").click(function () {  
 //获取账号密码  
 var name = **$**("#username").**val**();  
 var pw = **$**("#password").**val**();  
 var pwRepeat = **$**("#password-repeat").**val**();  
  
 //未输入账号名  
 if (name.**length** === 0) {  
 **$**("#username").parent().parent().addClass("error");  
 **$**("#usrname-error").removeClass("displayNONE");  
 **$**("#tips").addClass("displayNONE");  
 setTimeout(function () {  
 **$**("#username").parent().parent().removeClass("error");  
 **$**("#usrname-error").addClass("displayNONE");  
 **$**("#tips").removeClass("displayNONE");  
 }, 2000);  
 return;  
 }  
  
 //密码为空  
 if (pw.**length** === 0) {  
 **$**("#password").parent().parent().addClass("error");  
 **$**("#pw-error").removeClass("displayNONE");  
 setTimeout(function () {  
 **$**("#password").parent().parent().removeClass("error");  
 **$**("#pw-error").addClass("displayNONE");  
 }, 2000);  
 return;  
 }  
  
 //密码不相同  
 if (pw !== pwRepeat) {  
 **$**("#pw-rp-error").parent().parent().addClass("error");  
 **$**("#pw-rp-error").removeClass("displayNONE");  
 setTimeout(function () {  
 **$**("#pw-rp-error").parent().parent().removeClass("error");  
 **$**("#pw-rp-error").addClass("displayNONE");  
 }, 2000);  
 return;  
 }  
  
 var obj = {  
 **name**: name,  
 **password**: pw  
 };  
  
 //防止连续点击，先禁用提交按钮，防止重复提交  
 if (new Date() - time < 3000) {  
 return;  
 }  
 //发起请求前，更新提交时间，  
 time = new Date();  
 //发起请求，回调内容是一个对象，注册成功会有success，注册失败会有error属性  
 **$**.post("/reg", obj, function (data) {  
 if ("error" in data) {  
 **$**("#error-alert").removeClass("displayNONE")  
 **$**("#error-alert").text(data.error);  
 **$**("#error-alert").parent().parent().addClass("error");  
 setTimeout(function () {  
 **$**("#error-alert").addClass("displayNONE")  
 **$**("#error-alert").parent().parent().removeClass("error");  
 }, 2000)  
 time = 0; //注册失败，清空time计时，允许再次提交  
 } else if ("success" in data) {  
 **location**.**href** = data.**success**; //注册成功则重定向  
 }  
 })  
 })  
})

这里的验证逻辑很简单：

①账号不能为空；

②密码不能为空；

③两次密码应该相同；

为了简化，目前只设置这么多，不考虑例如最短密码长度、最长密码长度、密码需要由字母、数字、特殊符号混合的情况

打开blog.css，添加这样一个样式，用于隐藏某些暂时不需要显示的内容：

.displayNONE {  
 display: none;  
}

**【15】和MySQL交互**

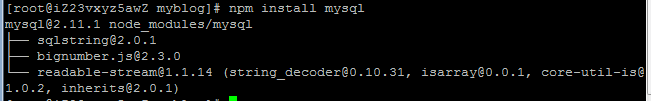
这里采用连接池的模式和MySQL交互；

①毫无疑问，我们首先应该安装MySQL模块；

进入项目的根目录，输入

**npm install mysql**

等待出现以下画面就表示装好了：

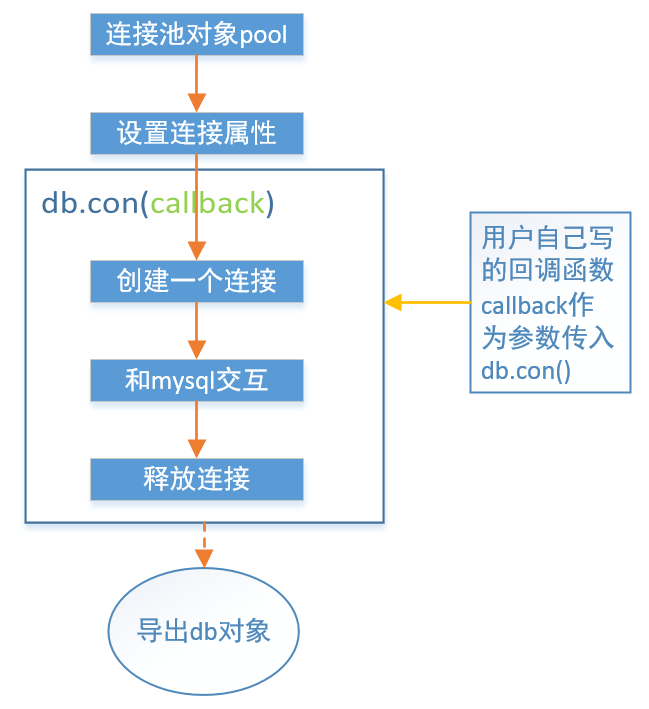


②在项目的根目录下面创建文件夹**models**，这个文件夹用于保存各种类模型

并在其中新建**db.js**

var mysql = require("mysql"); //调用nodejs和mysql交互的模块；  
var pool = mysql.createPool({ //创建连接池  
 **host**: '127.0.0.1', //表示本地的数据库  
 **user**: 'root', //账号  
 **password**: '123456', //密码  
 **port**: '3306', //默认端口  
 **database**: 'blog' //库名  
})  
var db = {};  
db.con = function (callback) { //callback是回调函数，连接建立后的connection作为其参数  
 pool.getConnection(function (err, connection) { //创建一个链接  
 if (err) { //对异常进行处理  
 throw err; //抛出异常  
 } else {  
 callback(connection); //如果正常的话，执行回调函数（即请求），具体请求在回调函数中写  
 }  
 connection.release(); //释放连接  
 })  
}  
module.exports = db;

这段代码做了这些事



我们该如何使用呢，很简单。

①生成一个db的实例；

②调用db的con方法，将请求写在回调函数之中，请求对象是回调函数的参数；

③例如：

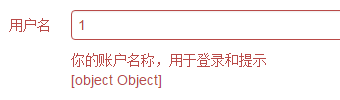
我们打开routes/reg.js，将代码修改为如下样子：

var express = require('express'); //调用express模块  
var router = express.**Router**(); //调用模块的Router方法  
var db = require('../models/db');  
  
  
router.**get**('/', function (req, res, next) {  
 res.render('reg')  
});  
  
router.post('/', function (req, res, next) { //当路由捕捉到url为/reg的post请求时，会执行以下函数  
 db.con(function (connect) { //db是我们导入的对象，connect是数据库的连接，此时数据库连接无错误，是一个正常的连接  
 //这是我们自定义的回调函数  
 connect.**query**("SELECT 1 + 1 AS solution", function (err, result) { //这里是一个对数据库的请求  
 if (err) { //假如请求出错，抛出异常  
 throw err;  
 return;  
 }  
 **console**.log(result); //控制台输出结果  
 res.send({ //将结果作为对象的error属性发送给客户端  
 **error**: result  
 });  
 }) //执行完回调函数后，会跳回db的con方法，执行接下来的属性（即释放连接）  
 })  
})  
module.exports = router;

这些代码做了一件事，在用户发起reg请求时，将查询结果（返回）；

还记不记得我们在注册页面的js回调函数，假如返回值有error属性，那么他会将error属性显示出来；

当然，由于并不能显示对象（error属性是一个对象），因此显示出来的是这样的



这只是一个测试用的代码，但通过这样的方法，说明我们的前后端交互是走通了的。

**【16】封装涉及到user的代码**

初步考虑，我们现在要注册，因此应该写一个向MySQL插入项的方法；

那是否我们就应该直接将这部分代码直接写在reg.js里面呢？

显然是不好的，这种做法违背了OOP的思想——封装；

简单来说，我们应该将用户视为一个类：

①至少包含两个方法：注册（reg）、查询（login）；

②至少包含两个属性：用户名（username）、密码（password）。

因此，新建user.js，放置于models文件夹内。

内容如下：

var db = require('./db')  
function User(user) { // 这是一个User类，传递的参数是一个对象，这个对象可以具有两个属性，分别是name和password  
 this.**name** = user.**name**; // 如果传递的user不是空，那么将其赋值给User类的实例的name属性  
 this.**password** = user.**password**; // 同上，赋给password属性  
}  
  
// 这个是插入方法  
User.**prototype**.save = function (callback) {  
 var self = this  
 if (this.**name**.**length** == 0 || this.**password**.**length** == 0) { //如果在没账号/密码的情况下就调用插入方法，则提示错误并返回  
 **console**.log("You can't save user information without NAME or PASSWORD!");  
 return callback("You can't save user information without NAME or PASSWORD!");  
 }  
 db.con(function (connect) {  
 // 数据库的表名为user，字段名为name和password  
 connect.**query**("INSERT INTO user(name,password) VALUES (?,?)", [self.**name**, self.**password**], function (err, result) {  
 // 上面的两个问号，表示第二个参数的self.name和self.password依次被替换到问号的位置；  
 // 需要注意的是：  
 // ①必须以数组形式依次排列；  
 // ②必须是字符串形式（不能是number）  
 if (err) { //如果出错，那么错误信息作为回调函数的参数返回  
 **console**.log("INSERT name:" + self.**name** + ", password:" + self.**password** + " error, the err information is " + err);  
 return callback(err);  
 }  
 callback(null, result); //如果正常执行，那么第一个参数为null（无错误），第二个参数为返回的结果  
 })  
 })  
}  
// 这个是查询方法  
User.**prototype**.get = function (callback) {  
 var self = this;  
 if (this.**name**.**length** == 0) { //如果在没账号/密码的情况下就调用插入方法，则提示错误并返回  
 **console**.log("You can't select user information without NAME!");  
 return callback("You can't select user information without NAME!");  
 }  
 var selectResult;  
 db.con(function (connect) {  
 connect.**query**('SELECT \* FROM user WHERE name = ?', [self.**name**], function (err, result) {  
 if (err) { //报错  
 **console**.log("select name:" + self.**name** + " error, the err information is " + err);  
 return callback(err);  
 }  
 //注意，这里返回的是带账号和密码的，另外，理论上是有可能有多个元素的，但由于在注册时，用户名限制了重复，因此只会返回一个  
 selectResult = result; //这里的result是一个数组，只包含一个元素（或者是空）  
 if (selectResult.**length**) { //查询到的话，数组是有元素的（即length > 0）  
 return callback(null, selectResult[0]) //这里的selectResult就是user对象，包含name和password属性  
 } else {  
 return callback(null, null); //如果查询不到，两个参数都为空  
 }  
 })  
 })  
}  
  
module.exports = User;

这个封装好的User类满足了我们的需求，下来需要在reg里面调用这个类。

下面是修改后的routes/reg.js的代码

var express = require('express'); // 调用express模块  
var router = express.Router(); // 调用模块的Router方法  
var User = require('../models/user'); // 调用刚才封装好的user类  
var crypto = require("crypto"); // 这个是加密用，nodejs本身提供  
  
  
router.**get**('/', function (req, res, next) {  
 res.render('reg')  
});  
  
router.post('/', function (req, res, next) { //当路由捕捉到url为/reg的post请求时，会执行以下函数  
 // 获取md5这个对象；（表示调用这个对象的方法，加密的形式是MD5）；  
 var md5 = crypto.createHash('md5');  
 // 调用md5的update方法，update的第一个参数是被处理的内容，第二个是可选的，但如果要对中文进行处理，那么就需要加上'utf-8'这个参数；  
 // 因此，我们的密码其实可以支持中文（理论上）  
 // digest指的是以什么形式进行编码，这里选择了base64，除此之外，还有hex（十六进制）、binary（二进制）这两种方法；  
 var password = md5.**update**(req.**body**.**password**, 'utf-8').digest('base64');  
  
 var newUser = new User({ //生成一个User的实例，并赋给他name和passowrd属性  
 **name**: req.**body**.**name**,  
 **password**: password //这里的password是加密过的（存储在数据库里也是加密过后的形式）  
 })  
 newUser.save(function (err, result) {  
 //do something  
 })  
  
})  
module.exports = router;

这部分并没有完成，之所以这样，是因为还欠缺一个东西，那就是登录状态的设置。

**【17】登录状态和session**

如何确定一个人是否在登录中？

有两个办法：①cookie；②session；

简单来说，他们之间的区别在于：

cookie保存在客户端，session保存在服务器端；

顺便补一个Cookie、LocalStorage和SessionStorage之间区别的链接：

<http://jerryzou.com/posts/cookie-and-web-storage/>

注意，这里的SessionStorage和session并不是同一个东西，前者保存在浏览器端，后者保存在服务器端。

再补一个express使用cookie和session的文章：

<http://wiki.jikexueyuan.com/project/node-lessons/cookie-session.html>

由于cookie可以在本地被修改，因此用cookie来保存用户状态是不安全的。

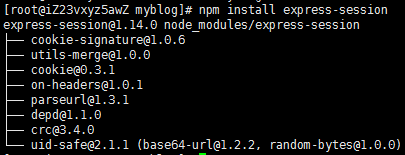
这里采用session来保存用户状态。

在高版本的express里（4.x及以上），cookie和session等模块并不直接包含在express里了，因此我们要安装express-session。

在项目的根目录输入：

npm install express-session

安装好后如图：



打开app.js，在

var app = express();

后添加如下代码：

//session  
var session = require('express-session');  
app.use(session({  
 **secret**: 'recommend 128 bytes random string',  
 **cookie**: {**maxAge**: 3600 \* 1000}  
}))

这种设置表示：

①默认session存储在内存中（默认设置，效果是重启后session失效）；

②最大有效时间（过期时间）是3600秒；

然后重新打开reg.js，在原先的空白处，添加以下代码：

if (user) { //如果第二个参数存在，说明用户名重复了，返回提示  
 return res.send({  
 **error**: "Username already exists."  
 });  
}  
if (err) { //如果报错，返回报错信息  
 **console**.log(err);  
 return res.send({  
 **error**: err  
 });  
}  
//此时说明无重复无报错，可以将用户信息存入到数据库之中  
newUser.save(function (err, result) {  
 if (err) { //如果存入时报错  
 return res.send({  
 **error**: err  
 });  
 }  
 //以上，done  
 //存储成功后返回还有点问题，需要搞明白  
 req.session.**user** = {  
 **name**: this.**name**,  
 **password**: this.**password** };  
 req.session.**success** = "注册成功！"; //添加session的注册成功信息  
 res.send({ //发送一个对象，属性为success，值为/  
 **success**: "/"  
 })  
 return;  
})

但此时还没结束，我们需要对数据库进行操作：（建议用数据库软件完成）

先建立一个库，库名为blog

在库中建立一个表，表名为user；

表中有三个字段：

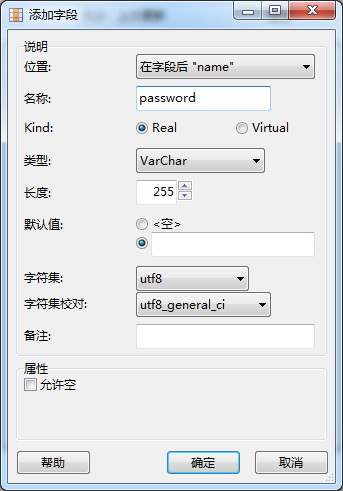
①第一个为id，类型为int，自动增量，不允许为空；

②第二个为name，类型为varchar，长度为255，字符集为utf8，不允许为空；

③第三个为password，类型为varchar，长度为255，字符集为utf8，不允许为空；

于是，注册流程已经走通，用户可以顺利注册了

账号和密码用软件创建时设置如图：



**【18】设置登录状态**

在之前，已经可以通过session保存用户了，下来，需要根据session的记录，区分用户登录和未登录的状态。

简单来说，有两种情况：

①session中的user属性为空，用户未登录；

②session中的user属性不为空，用户已登录，登录人士为user的name属性；

为了简化，这里并没有对同时登录的检测（避免一个账号在两个地方登录）。

而在访问页面时，应该检测登录状态：

①假如已经登录，那么就不应该显示登录窗口、注册窗口，以及发表微博相关的内容；

②假如还没有登录，那么就不应该显示登出的按钮；

为了完成这个目的，那么应该在引擎渲染时进行检查，最笨的做法是，将变量设置于res.render的第二个参数的属性之中：

打开**layout.jade**，将代码如下修改：

**doctype html  
html  
 head  
 title MyBlog By qq20004604  
 link**(rel='stylesheet', href='./stylesheets/bootstrap.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/bootstrap-responsive.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/blog.css')  
 **script**(type="text/javascript",src='javascripts/jq.js')  
 **script**(type="text/javascript",src='javascripts/bootstrap.min.js')  
 **script**(type="text/javascript",src='javascripts/blog.js')  
 **body  
 div**.*navbar*.*navbar-fixed-top* **div**.*navbar-inner* **div**.*container* **a**.*btn*.*btn-navbar*(data-toggle="collapse",data-target=".nav-collapse")  
 **span**.*icon-bar* **span**.*icon-bar* **span**.*icon-bar* **a**.*brand*(href="/") **MyBlog  
 div**.*nav-collapse* **ul**.*nav* **li  
 a**(href="/") **首页** *//以下是修改后的部分* if(**user**())  
 **li  
 a**(href="/logout") **登出** else  
 **li  
 a**(href="/login") **登入  
 li  
 a**(href="/reg") **注册  
 li  
 a**(href="/language") **切换语言  
 div**#contrainer.*container* **block content  
 hr  
 fotter  
 p 作者：王冬　　　QQ:20004604**

打开**index.jade**

extends layout  
**block content  
 div**.*hero-unit* **h1 我的博客  
 p 这个是基于Nodejs作为后台，jade作为模板来，使用了Express作为框架  
 br  
 br** *//修改以下部分* if(!**user**())  
 **br  
 br  
 a**.*btn*.*btn-primary*.*btn-large*(href="/login") **登录  
 a**.*btn*.*btn-large*(href="/reg") **立即注册** else  
 **textarea**#textarea.*uneditable-input* **button**#postBlog.*btn*.*btn-large* **提交微博  
 button**#clearBlog.*btn*.*btn-large* **清空  
 div**#submitError.*alert*.*alert-error*.*displayNONE*.*myalert* **div**#submitSuccess.*alert*.*alert-success*.*displayNONE* **div**.*row*.*content* **div**.*span4* **h2 烟雨江南说  
 p 当欲望没有了枷锁，就没有了向前的路  
 p 只有转左，或者向右  
 p 左边是地狱，右边也是地狱  
 div**.*span4* **h2 烟雨江南说  
 p 那些都是极好极好的  
 p 可是我偏偏不喜欢  
 div**.*span4* **h2 烟雨江南说  
 p 我不怕傻  
 p 只怕  
 p 遇不到  
 p 可以让我变傻的人  
 div**.*span4* **h2 烟雨江南说  
 p 人在年轻的时候总会有些莫名的坚持，  
 p 并且以此感动着自己，  
 p 却时常会在不经意间让真正重要的东西从指间流走。  
 div**.*span4* **h2 烟雨江南说  
 p 记忆真是一种奇怪的东西，  
 p 有时候会涤荡所有的苦难，只留下温情，  
 p 有时候却磨灭掉曾有的欢乐，唯剩下苍白和丑陋。  
 div**.*span4* **h2 烟雨江南说  
 p 那存在的，都是幻影。  
 p 那永恒的，终将毁灭。  
 p 世界万物，缤纷色彩，都是被蒙蔽的人心罢了。  
 div**.*span4* **h2 烟雨江南说  
 p 诸神以真相示人，而世人却视而不见  
 div**.*span4* **h2 烟雨江南说  
 p 只有绵羊会向狮子要求平等，  
 p 而狮子们从来不会这样想。  
 div**.*span4* **h2 烟雨江南说  
 p 愿迷途的旅人，从此得享安息。  
 p 因理想而不朽，因归返而救赎。  
 div**#scrollToFoot **滚动到底部然后加载内容**

打开**routes/index.js**，将代码如下修改：

var express = require('express'); //调用express  
var router = express.**Router**(); //生成express的Router方法的一个实例  
  
//处理函数  
router.**get**('/', function (req, res, next) { //捕获根url  
 res.render('index', {  
 user: function () { //user将读取session的属性，然后给予不同的返回值  
 if (req.session.**user**)  
 return req.session.**user**;  
 else  
 return null;  
 }  
 });  
 //res.render方法渲染一个引擎模板，  
 //第二个参数是一个对象，对象里的变量可以在引擎中使用，  
 //第三个参数是回调函数，提供两个参数，分别是err和html，err是错误，html是渲染后的页面。如果使用这个回调函数，那么将不会自动响应，即要用户自己写返回html的命令  
});  
  
module.exports = router;

**代码解释：**

①在之前的代码中，我们如果注册成功，那么会将user属性设置在session之中；

②而在index.js里，会传递一个变量，那就是user；

③这个user会读取session的user属性，由于session默认的属性是没有user的，因此返回值为null（空），而注册成功后，返回值则为user（不为空）；

④在layout.jade中，会执行这个user函数。假如未登录，user为空，那么执行else语句（显示登入和注册标签）；否则显示登出标签；

⑤在index.jade中，同样执行，假如未登录，显示登录、注册按钮；假如已经登录了，显示输入微博的窗口和提交按钮；

于是，登录后的页面是这样的：



但是这种行为个人觉得比较傻，并且也不利于做多语言扩展，因此换一种方式。

**【19】动态视图助手和多语言支持**

关于动态视图助手的说明请看我之前的博客：

<http://blog.csdn.net/qq20004604/article/details/51934991>

简单来说，使用动态视图助手时，我们可以在引擎渲染的时候，直接调用视图助手中的变量；

首先在models下面建立language.js文件，内容如下

function internation() {  
  
}  
  
internation.**prototype**.**languagePacks** = {  
 **zh\_cn**: {  
 **index**: {  
 **title**: "我的博客",  
 **description**: "这个是基于Nodejs作为后台，jade作为模板来，使用了Express作为框架",  
 **login**: "登录",  
 **regNow**: "立即注册",  
 **submit**: "提交微博",  
 **emptyInput**: "清空"  
 },  
 **layout**: {  
 **title**: "MyBlog By qq20004604",  
 **topTitle**: "MyBlog",  
 **indexButton**: "首页",  
 **logout**: "登出",  
 **login**: "登入",  
 **reg**: "注册",  
 **language**: "切换语言",  
 **foot**: "作者：王冬　　　QQ:20004604"  
 }  
 },  
 **eng**: {  
 **index**: {  
 **title**: "My blog",  
 **description**: "I use NodeJS, Express, and MySQL to design this site.",  
 **login**: "Login",  
 **regNow**: "Reg",  
 **submit**: "Submit",  
 **emptyInput**: "Clear"  
 },  
 **layout**: {  
 **title**: "MyBlog By qq20004604",  
 **topTitle**: "MyBlog",  
 **indexButton**: "Index",  
 **logout**: "Logout",  
 **login**: "Login",  
 **reg**: "Reg",  
 **language**: "Language",  
 **foot**: "I'm WangDong, you can contact me with QQ:20004604"  
 }  
 }  
}  
  
//如果有语言设置，则设置为对应的语言，否则，设置为中文；  
internation.**prototype**.getPacks = function (setting) {  
 if (!setting) {  
 return this.**languagePacks**["zh\_cn"]  
 }  
 if (setting in this.**languagePacks**) {  
 return this.**languagePacks**[setting];  
 } else {  
 return this.**languagePacks**["zh\_cn"]  
 }  
}  
  
internation.**prototype**.set = function (app) {  
 var self = this;  
 app.use(function (req, res, next) {  
 //这个是示例，定义一个动态视图助手变量  
  
 //如果有语言设置，则设置为对应的语言，否则，设置为中文；  
 res.**locals** = self.getPacks(req.session.**language**)  
  
 //下面两个因为要调用req和res，所以特殊设置  
 res.**locals**.user = function () {  
 if ('session' in req && 'user' in req.session)  
 return req.session.**user**;  
 else  
 return null;  
 };  
  
 //这里不能用error这个变量名  
 res.**locals**.err = function () {  
 //如果session不存在，或者session中没有err属性  
 if (!'session' in req || !'err' in req.session)  
 return null; //返回空  
 //否则返回err，并将session中的err重置为空（因此只返回一次）  
 var err = req.session.**err**;  
 req.session.**err** = null;  
 return err;  
 };  
  
 //注册成功的提示  
 res.**locals**.success = function () {  
 //原理同err  
 if (!'session' in req || !'success' in req.session)  
 return null;  
 var success = req.session.**success**;  
 req.session.**success** = null;  
 return success;  
 }  
 next();  
 })  
}  
  
  
module.exports = internation;

在以上代码中，程序做了以下事情：

①创建一个语言包，他包含中文和英文两种；

②设置了一个方法，他可以根据session的language属性来决定获取哪个语言包；

但也有一定缺陷：

①假如在zh\_cn的index中有一条test属性，他有一个对应的字符串；但在eng中相应的位置不存在，那么切换为eng时，这里就会显示错误；

②调用的时候，需要通过例如index.title这样的形式来调用

然后我们将这个模块加入到我们的代码之中；

打开app.js，在之前添加的关于**session**的代码后面，添加如下代码：

var language = require('./models/language'); //多语言  
var internation = new language();  
internation.set(app);

修改layout.jade

**doctype html  
html  
 head  
 title #{**layout.**title}  
 link**(rel='stylesheet', href='./stylesheets/bootstrap.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/bootstrap-responsive.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/blog.css')  
 **script**(type="text/javascript",src='javascripts/jq.js')  
 **script**(type="text/javascript",src='javascripts/bootstrap.min.js')  
 **script**(type="text/javascript",src='javascripts/blog.js')  
 **body  
 div**.*navbar*.*navbar-fixed-top* **div**.*navbar-inner* **div**.*container* **a**.*btn*.*btn-navbar*(data-toggle="collapse",data-target=".nav-collapse")  
 **span**.*icon-bar* **span**.*icon-bar* **span**.*icon-bar* **a**.*brand*(href="/") **#{**layout.**topTitle}  
 div**.*nav-collapse* **ul**.*nav* **li  
 a**(href="/") **#{**layout.**indexButton}** *//以下是修改后的部分* if(user())  
 **li  
 a**(href="/logout") **#{**layout.**logout}** else  
 **li  
 a**(href="/login") **#{**layout.**login}  
 li  
 a**(href="/reg") **#{**layout.**reg}  
 li  
 a**(href="/language") **#{**layout.**language}  
 div**#contrainer.*container* **block content  
 hr  
 fotter  
 p #{**layout.**foot}**

重新访问页面，我们会发现页面跟以前并没有什么区别，说明我们的语言包已经正常运行了。

**【20】添加语言切换功能；**

根据我们之前的代码，语言切换很简单，只需要在session添加属性即可；

首先，设置路由，routes文件夹下新建language.js

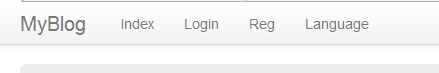
var express = require('express'); //调用express  
var router = express.**Router**(); //生成express的Router方法的一个实例  
  
//处理函数  
router.**get**('/', function (req, res, next) { //捕获根url  
 if (!('language' in req.session) || req.session.**language** === 'zh\_cn')  
 req.session.**language** = 'eng'; //更改属性  
 else  
 req.session.**language** = 'zh\_cn';  
 res.redirect('/'); //重定向  
});  
  
module.exports = router;

在app.js的路由代码附近添加如下代码：

var lan = require('./routes/language');

app.use('/language', lan); //切换语言的

重启程序，然后进入首页点击切换语言，页面顶端变为如下状态：



此时如果再次点击‘Language’，那么页面又会切回中文；

我们顺便修改layout.jade文件为：

**doctype html  
html  
 head  
 title #{**layout.**title}  
 link**(rel='stylesheet', href='./stylesheets/bootstrap.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/bootstrap-responsive.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/blog.css')  
 **script**(type="text/javascript",src='javascripts/jq.js')  
 **script**(type="text/javascript",src='javascripts/bootstrap.min.js')  
 **script**(type="text/javascript",src='javascripts/blog.js')  
 **body  
 div**.*navbar*.*navbar-fixed-top* **div**.*navbar-inner* **div**.*container* **a**.*btn*.*btn-navbar*(data-toggle="collapse",data-target=".nav-collapse")  
 **span**.*icon-bar* **span**.*icon-bar* **span**.*icon-bar* **a**.*brand*(href="/") **#{**layout.**topTitle}  
 div**.*nav-collapse* **ul**.*nav* **li  
 a**(href="/") **#{**layout.**indexButton}** *//以下是修改后的部分* if(user())  
 **li  
 a**(href="/logout") **#{**layout.**logout}** else  
 **li  
 a**(href="/login") **#{**layout.**login}  
 li  
 a**(href="/reg") **#{**layout.**reg}  
 li  
 a**(href="/language") **#{**layout.**language}  
 div**#contrainer.*container* **block content  
 hr  
 fotter  
 p #{**layout.**foot}**

再修改routes/index.js为：

var express = require('express'); //调用express  
var router = express.**Router**(); //生成express的Router方法的一个实例  
  
//处理函数  
router.**get**('/', function (req, res, next) { //捕获根url  
 res.render('index');  
});  
  
module.exports = router;

**【21】登录功能的完成**

与注册（reg.js）功能的代码类似，

首先在models的语言包中分别添加如下属性：

**message**: {  
 **reg\_success**: "注册成功！",  
 **reg\_fail**: "注册失败！",  
 **login\_success**: "登录成功！",  
 **login\_fail**: "登录失败",  
 **errorName**: "用户名不存在",  
 **errorPW**: "密码错误！"  
}

和

**message**: {  
 **reg\_success**: "Reg Success！",  
 **reg\_fail**: "Reg Failed！",  
 **login\_success**: "Login Success！",  
 **login\_fail**: "Login Failed!",  
 **errorName**: "The username is not exists.",  
 **errorPW**: "Error Password!"  
}

打开routes/login.js文件

将代码修改如下：

var express = require('express'); //调用express模块  
var router = express.**Router**(); //调用模块的Router方法  
var User = require('../models/user'); // 调用刚才封装好的user类  
var crypto = require("crypto"); // 这个是加密用，nodejs本身提供  
  
  
router.**get**('/', function (req, res, next) {  
 res.render('login')  
});  
  
router.post('/', function (req, res, next) {  
 var md5 = crypto.createHash('md5');  
 var password = md5.**update**(req.**body**.**password**, 'utf-8').digest('base64');  
  
 var newUser = new User({ //生成一个User的实例，并赋给他name和passowrd属性  
 **name**: req.**body**.**name** })  
 newUser.get(function (err, user) {  
 if (!user) { //用户名不存在  
 return res.send({**errorName**: res.**locals**.**message**.**errorName**});  
 }  
 else if (user) { //如果第二个参数存在，说明用户名重复了，那么监测密码是否相同  
 if (user.**password** === password) { //密码正确，登录成功  
 req.session.**user** = user;  
 req.session.**success** = res.**locals**.**message**.**login\_success**;  
 return res.send({**success**: '/'});  
 } else {  
 return res.send({**errorPW**: res.**locals**.**message**.**errorPW**});  
 }  
 }  
 else if (err) { //如果报错，返回报错信息  
 **console**.log(err);  
 return res.send({  
 **error**: err  
 });  
 }  
  
 })  
})  
module.exports = router;

修改login.jade

extends layout  
**block content  
 script**(type="text/javascript",src='javascripts/login.js')  
 **div**.*form-horizontal* **fieldset  
 legend 登录  
 div**.*control-group* **label**.*control-label*(for='username') **用户名  
 div**.*controls* **input**.*input-xlarge*#username(name='username',type='text')  
 **p**.*help-block*.*displayNONE*#name-alert  
 **div**.*control-group* **label**.*control-label*(for='password') **密码  
 div**.*controls* **input**.*input-xlarge*#password(name='password',type='password')  
 **p**.*help-block*.*displayNONE*#pw-alert  
 **div**.*form-actions* **button**.*btn*.*btn-primary*#submit **登录**

在public/javascripts下新建**login.js**，内容如下：

**$**(**document**).ready(function () {  
 var time = 0;  
 **$**("#submit").click(function () {  
 var user = {  
 **name**: **$**("#username").**val**(),  
 **password**: **$**("#password").**val**()  
 }  
  
 //禁止用户名、密码为空  
 if (user.**name**.**length** === 0) {  
 **$**("#name-alert").text("用户名不能为空");  
 return;  
 } else if (user.**password**.**length** === 0) {  
 **$**("#pw-alert").text("密码不能为空");  
 return;  
 }  
  
  
 if (new Date() - time < 3000)  
 return  
 time = new Date();  
 //发起ajax请求  
 **$**.post("login", user, function (data) {  
 if ("errorName" in data) {  
 **$**("#name-alert").removeClass("displayNONE")  
 **$**("#name-alert").text(data.**errorName**);  
 **$**("#name-alert").parent().parent().addClass("error");  
 setTimeout(function () {  
 **$**("#name-alert").addClass("displayNONE")  
 **$**("#name-alert").parent().parent().removeClass("error");  
 }, 2000)  
 time = 0;  
 } else if ("errorPW" in data) { //  
 **$**("#pw-alert").removeClass("displayNONE")  
 **$**("#pw-alert").text(data.**errorPW**);  
 **$**("#pw-alert").parent().parent().addClass("error");  
 setTimeout(function () {  
 **$**("#pw-alert").addClass("displayNONE")  
 **$**("#pw-alert").parent().parent().removeClass("error");  
 }, 2000)  
 time = 0;  
 } else if ("success" in data) {  
 **location**.**href** = data.success; //注册成功则重定向  
 }  
 })  
 })  
})

于是，登录相关的内容算是做完了（但没有做多语言）；

**【22】登出功能的完成**

相比较登录，登出功能就要简单的多，只需要将session的user属性重置为null即可。

在app.js的路由代码附近添加以下代码：

var logout = require('./routes/logout');

app.use('/logout', logout); //登出

在routes文件夹下新建**logout.js**，代码非常简单：

var express = require('express'); //调用express模块  
var router = express.**Router**(); //调用模块的Router方法  
  
//登出只需要清空session的user属性即可  
router.**get**('/', function (req, res, next) {  
 req.session.**user** = null;  
 res.redirect('/');  
})  
module.exports = router;

**【23】清点需求**

重新打开我们之前的需求表：

①有首页；

②支持注册；

③支持登录、登出；

④可以发表博客，发表的博客可以保存到数据库；

⑤可以查看博客，博客从数据库中读取，为了简化，这里设置为查看所有人的博客；

⑥查看博客时，初始查看的数量有限，但可以无限加载新的博客，直到查看完全部博客；

⑦一定程度上实现多语言（即可以切换显示的语言版本），但由于复杂度所限，因此只部分实现（但框架已建立好，可以通过继续完善代码实现整体的国际化）；

⑧根据登录状态，对可以访问的页面进行控制（某些允许某些禁止）

红色部分是已经完成的，蓝色部分是尚未完成的。

下来，我们的任务是完成④⑤⑥这三项。

**【24】博客的发表**

在登录之后，我们会在首页看到一个文本输入框，还有提交和清空按钮；

但目前尚没有逻辑，需要进行添加，打开public\javascripts\blog.js，添加如下代码：

**$**(**document**).ready(function () {  
 //根据url，设置高亮  
 if (**window**.**location**.**pathname** === '/') {  
 **$**(".nav-collapse .nav li:first-child").addClass("active");  
 }  
 else if (**window**.**location**.**pathname** === '/login') {  
 **$**(".nav-collapse .nav li:nth-child(2)").addClass("active");  
 }  
 else if (**window**.**location**.**pathname** === '/reg') {  
 **$**(".nav-collapse .nav li:nth-child(3)").addClass("active");  
 }  
  
 //清空输入框  
 **$**("#clearBlog").click(function () {  
 **$**("#textarea").**val**("");  
 })  
 var lastSubmit = null;  
 var timeer = null;  
 var successSubmit = null;  
 //防止连续点击  
 function ErrorAlert() {  
 if (new Date() - lastSubmit < 3000) {  
 clearTimeout(timeer);  
 }  
 lastSubmit = new Date();  
 timeer = setTimeout(function () {  
 **$**("#submitError").addClass("displayNONE");  
 }, 3000);  
 }  
  
 //提交输入框  
 **$**("#postBlog").click(function () {  
 //防止重复发表，因此需要间隔10秒  
 if (successSubmit && new Date() - successSubmit < 10000) {  
 //这里是警告提示，防止连续发送消息  
 **$**("#submitError").**text**("你发的太快了，喝喝茶吧！距离下次可以发送消息的时间还有：" + (new Date() - successSubmit).toFixed(0) + " 秒。");  
 **$**("#submitError").removeClass("displayNONE");  
 //防止连续点击  
 ErrorAlert()  
 return;  
 }  
  
 var text = **$**("#textarea").**val**();  
 if (text.**length** === 0) { //禁止发送空内容  
 **$**("#submitError").**text**("请填写输入内容");  
 **$**("#submitError").removeClass("displayNONE");  
 //防止连续点击  
 ErrorAlert()  
 return;  
 }  
 var length = 0;  
 //获取输入长度，英文字母为1，中文汉字为2  
 for (var i = 0; i < text.**length**; i++) {  
 if (text[i].match(/[^\x00-\xff]/ig) != null)  
 length += 2;  
 else  
 length += 1;  
 }  
 if (length > 255) {  
 **$**("#submitError").**text**("字符长度过长，限制字符长度为255个字节，你的文本长度为" + length + "个字节");  
 **$**("#submitError").removeClass("displayNONE");  
 ErrorAlert()  
 return;  
 }  
  
 //先清除输入框再提交  
 **$**("#textarea").**val**("");  
 successSubmit = new Date();  
 **$**.post('/post', {**text**: text}, function (item) {  
 if (item.code == 500) {  
 successSubmit = 0;  
 **$**("#submitError").**text**(item.data);  
 **$**("#submitError").removeClass("displayNONE");  
 setTimeout(function () {  
 **$**("#submitError").addClass("displayNONE");  
 }, 3000);

} else if (item.code == 200) {  
 **$**("#submitSuccess").**text**(item.data);  
 **$**("#submitSuccess").removeClass("displayNONE");  
 setTimeout(function () {  
 **$**("#submitSuccess").addClass("displayNONE");  
 }, 3000);  
 }  
 })  
 })  
  
})

这段代码做了以下事：

①根据当前的url，给对应的标签高亮；

②点击清空按钮，会清除输入框的内容；

③点击提交按钮，会验证输入框内字符串的长度，禁止发送空内容和超过255字节长度的内容；

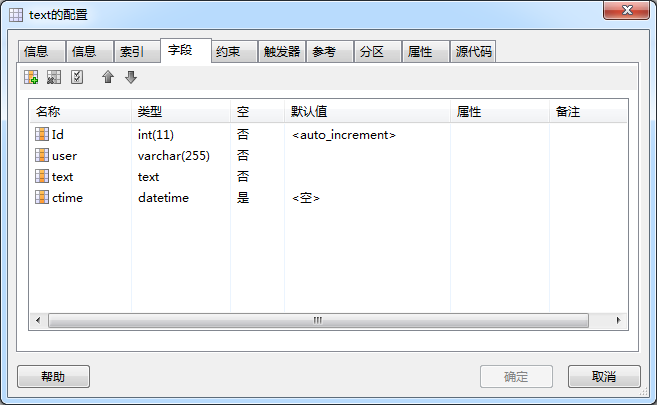
④当成功发出post请求后，禁止在短时间内连续发送post请求；当发送失败后，取消禁止状态，可以立刻再次发送

⑤验证通过后，将消息内容以post请求发送到服务器，并清空输入框；

⑥返回值有两种，一种是code属性为200，一种为500；前者为发送成功，发出提示消息；后者发送失败，提示失败原因。

接下来首先要在MySQL里创建一张表，用于保存发送的文本内容；

表名为text，字符类型为utf8，表项如下：



id为自动增量；

user为发表这个博客的用户；

text博客的内容；

ctime为博客的创建时间；

其次在models文件夹下新建**blog.js**

var db = require('./db');  
  
function BlogSubmit(user) {  
 this.**user** = user.**name**;  
 this.**text** = user.**text**;  
 this.**time** = new Date().Format("yyyy-MM-dd HH:mm:ss");  
}  
  
//存储文本内容  
BlogSubmit.**prototype**.save = function (callback) {  
 var self = this;  
 **console**.log(self.**user**);  
 db.con(function (connect) {  
 connect.**query**("INSERT INTO text(user,text,ctime) VALUES (?,?,?)", [self.**user**, self.**text**, self.**time**], function (err, result) {  
 if (err) {  
 **console**.log("INSERT text user:" + self.**user** + ", text:" + self.**text** + ", ctime: " + self.**time** + " error, the err information is " + err);  
 return callback(err);  
 }  
 callback(null, result);  
 })  
 })  
}

//给原生的Date添加一个方法，传递的参数是格式  
Date.**prototype**.Format = function (fmt) { //author: meizz  
 var obj = {  
 **"M+"**: this.getMonth() + 1, //月份  
 **"d+"**: this.getDate(), //日  
 **"H+"**: this.getHours(), //小时  
 **"m+"**: this.getMinutes(), //分  
 **"s+"**: this.getSeconds(), //秒  
 **"q+"**: **Math**.floor((this.getMonth() + 3) / 3), //季度  
 **"S"**: this.getMilliseconds() //毫秒  
 };  
 if (/(y+)/.test(fmt))  
 fmt = fmt.replace(RegExp.**$1**, (this.getFullYear() + "").substr(4 - RegExp.**$1**.**length**));  
 for (var key in obj)  
 if (new RegExp("(" + key + ")").test(fmt)) fmt = fmt.replace(RegExp.**$1**, (RegExp.**$1**.**length** == 1) ? (obj[key]) : (("00" + obj[key]).substr(("" + obj[key]).**length**)));  
  
 **console**.log(fmt);  
 return fmt; //返回值是格式化好之后的时间  
}  
  
module.exports = BlogSubmit;

然后在routes文件夹下新建post.js

var express = require('express'); //调用express模块  
var router = express.Router(); //调用模块的Router方法  
var BlogSubmit = require("../models/blog");  
  
  
router.post('/', function (req, res, next) {  
 //如果不登录是不能发表的（这个是为了防止登录过期，但是页面还保持在登录时的情况）  
 if (!req.session.**user**) {  
 req.session.**err** = "请登录"; //因为这里，需要修改layout.jade  
 return res.send({  
 **code**: 403,  
 **data**: '/login'  
 });  
 }  
 //console.log(req.session.user);  
 //登录后值为（示例）  
 //{ Id: 23,  
 // username: '1',  
 // userpassword: 'xMpCOKC5I4INzFCab3WEmw==' }  
 var blog = new BlogSubmit({  
 **name**: req.session.**user**.**name**,  
 **text**: req.**body**.**text** })  
 blog.save(function (err, result) {  
 if (err) {  
 return res.send({  
 **code**: 500,  
 **data**: "错误描述" + err  
 })  
 }  
 return res.send({  
 **code**: 200,  
 **data**: "成功发表！"  
 })  
 })  
})  
module.exports = router;

这两部分代码干了以下几件事：

①给Date()添加了一个方法，用于格式化日期；

②创建了一个BlogSubmit的类，传递的参数是一个对象，他有name属性和text属性，会将其值传递给这个类的实例的user属性和text属性中。同时，会在创建这个类的时候获取当前时间。于是，text表的三项内容有了；

③给BlogSubmit类添加了一个方法，这个方法的作用是将这个类的实例的属性，存储在MySQL之中；

④由于表限制text中必须保存内容，因此假如用户通过其他方法直接发起一个post请求，也是不可能成功的（会报错）；

⑤保存的前提是用户已登录，未登录的时候发表（比如登录过期）会提示错误。

⑥无论成功或者失败，都会给用户返回一个对象，成功的code属性为200，失败的为500；

最后不要忘记在app.js里添加路由：

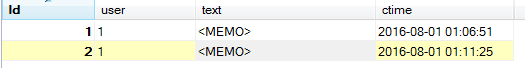
var postBlog = require('./routes/post');

app.use('/post', postBlog); //提交博客

此时发表博客，会提示：



查看数据库，数据库对应的表是有内容的，说明成功发表了。



**【25】博客的加载**

博客要满足这些功能；

①初始加载一些博客内容；

②当滚动条拖到最下面的时候，发起ajax请求，取得更多博客内容；

③将博客内容添加到最下方；

④当读取完所有内容时，提示无新博客；

首先，在路由之中添加：

app.use('/loadblog', users); //用户主页，users来处理

将博客的发送交给user来处理；

将以下代码添加进models/blog.js

//读取文本内容（读取所有的），从第count条偏移量开始取9条  
BlogSubmit.**prototype**.getAll = function (count, callback) {  
 var self = this;  
 db.con(function (connect) {  
 //第一次查询数据库还有多少数据  
 connect.**query**("SELECT count(\*) FROM text", null, function (err, result) {  
 if (err) {  
 **console**.log("SELECT \* FROM text limit :" + count + ", 9 error, the err information is " + err);  
 return callback(err);  
 }  
 //console.log(result[0]['count(\*)']); //这个是查询出来的值  
 //只有当请求数据的编号（已经查询到多少个），小于最大数据量的时候，才会从数据库内读取数据  
 if (Number(count) < result[0]['count(\*)']) {  
 connect.**query**("SELECT \* FROM text limit 0,9", [count.toString()], function (err, result) {  
 if (err) {  
 **console**.log("SELECT \* FROM blogtext limit :" + count + ", 9 error, the err information is " + err);  
 return callback(err);  
 }  
 result.**offset** = Number(count) + result.**length**;  
 callback(null, result);  
 })  
 } else {  
 callback(null, null);  
 }  
 })  
 })  
}

修改routes/users.js

var express = require('express');  
var router = express.**Router**();  
var BlogSubmit = require("../models/blog");  
  
/\* GET users listing. \*/  
router.**get**('/', function (req, res, next) {  
 res.send('respond with a resource');  
});  
  
router.**get**('/:count', function (req, res, next) {  
 var blog = new BlogSubmit({  
 **username**: null,  
 **text**: null  
 })  
 blog.getAll(req.**params**.count, function (err, result) {  
 if (err) {  
 res.send({  
 **code**: 500,  
 **data**: "错误描述" + err  
 })  
 } else if (result === null) { //空数据  
 res.send({  
 **code**: 501,  
 **data**: "服务器已经没有更多数据了"  
 })  
 } else {  
 res.send({**code**: 200, **data**: result, **offset**: result.**offset**});  
 }  
 })  
});  
  
module.exports = router;

修改public/javascripts/blog.js，添加以下内容：

var lastLoad = 0; //初始时间设置为0  
var lastLoadCount = 0; //偏移量，防止读取到重复数据  
//加载更多的内容  
function loadMoreContent() {  
 //控制加载间隔，时间差大于2秒，且当前不处于加载中  
 if (new Date() - lastLoad > 2000 && **$**("#scrollToFoot").**innerHTML** !== "加载中~~~") {  
 lastLoad = new Date();  
 **$**("#scrollToFoot").**innerHTML** = "加载中~~~";  
 //设置重置，假如超时10秒，那么允许再次提交  
 setTimeout(function () {  
 if (**$**("#scrollToFoot").**innerHTML** === "加载中~~~") {  
 **$**("#scrollToFoot").**text**("加载失败");  
 }  
 }, 10000);  
 **$**.get('/loadblog/' + lastLoadCount, function (obj) {  
 //这里假设item的格式：  
 if (obj.**code** === 500) {  
 **$**("#scrollToFoot").**text**("加载失败");  
 } else if (obj.**code** === 501) {  
 **$**("#scrollToFoot").**text**(obj.**data**);  
 lastLoad = new Date() + 10000000; //禁止继续请求  
 } else {  
 obj.**data**.forEach(function (item) {  
 **$**("#content").append('<div class="span4"><h2>' + item.**user** + '说</h2>' +  
 '<p>' + item.**text** + '</p></div>');  
 })  
 **$**("#scrollToFoot").**innerHTML** = "滚动到底部然后加载内容";  
 lastLoadCount += obj.**offset**;  
 }  
 })  
 }  
}  
  
//加载完毕时，加载一次内容  
loadMoreContent();  
//向下滚动时加载内容  
**$**(**window**).scroll(function () {  
 if (**$**(**document**).**height**() - **$**(this).**scrollTop**() - **$**(this).**height**() < 100) {  
 loadMoreContent();  
 }  
})

修改index.jade中的

**div**.*row*.*content*

为：

**div**.*row*.*content*#content

到目前为止，可以自动无限加载博客了。

**【26】路由控制**

现在，就差最后一步了。

毫无疑问，在用户未登录的时候，应该禁止掉用户发送只有登录过的用户才可以做的事情；而用户在登录之后，也应该禁止掉用户在非登录情况下才能做的事情；

前者典型是登出，而后者典型是登录和注册。

比如这样就是不对的



解决办法，利用“路由中间件”来解决问题，这就是我们在router.get之类的函数中，第二个参数的第三个参数next；

所谓路由中间件，首先要理解请求；

假如有一个url为/reg 的post请求，当他遇见router.post(‘/reg’,xxxxx)时，执行完这行代码的回调函数后便停止了。

假如我们需要他被两段这样的代码来先后处理，那么显然是不可行的，路由会自动结束请求。

这个时候在第一段代码最后执行回调函数中的第三个参数next，那么就会继续执行第二段代码。

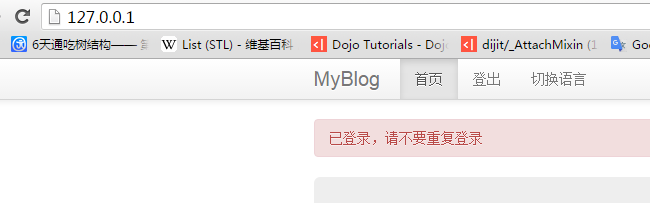
更具体的请看我的博客：

<http://blog.csdn.net/qq20004604/article/details/51817625>

打开app.js，在之前所有路由代码之前，添加以下代码：

//路由的处理  
app.use('/login', checkNotLogin);  
app.use('/reg', checkNotLogin);  
  
//必须在已登录情况下才能访问  
app.use('/logout', checkLogin);  
//未登录检测（已登录情况下执行）  
function checkNotLogin(req, res, next) {  
 if (req.session.**user**) {  
 req.session.**err** = "已登录，请不要重复登录";  
 return res.redirect('/');  
 }  
 next();  
}  
//已登录检测（未登录情况下执行）  
function checkLogin(req, res, next) {  
 if (!req.session.**user**) {  
 req.session.**err** = "你还没有登录，请登录";  
 return res.redirect('/login');  
 }  
 next();  
}

此时重启nodejs程序，登录博客，然后url地址输入/reg，会有以下提示



这说明，我们的路由权限控制已经顺利完成了。

这篇博客的总体结构也基本完成了。

虽然还有很多需要修补的问题，但那已经是下一篇博客的故事了。

【27】bug修复

之前因为更改为定时刷新之后，只有当处于首页时，js才能正常自行；

当进入登录、注册页时，由于无法找到#textarea这个dom，因此会报错，导致代码无法正常运行。

因此，将blog.js分拆，新建一个index.js在public\javascripts目录下，代码分拆后结果如下：

blog.js

**[javascript]** [view plain](http://blog.csdn.net/qq20004604/article/details/52019904" \o "view plain) [copy](http://blog.csdn.net/qq20004604/article/details/52019904" \o "copy)

1. $(document).ready(**function** () {
2. //根据url，设置高亮
3. **if** (window.location.pathname === '/') {
4. $(".nav-collapse .nav li:first-child").addClass("active");
5. }
6. **else** **if** (window.location.pathname === '/login') {
7. $(".nav-collapse .nav li:nth-child(2)").addClass("active");
8. }
9. **else** **if** (window.location.pathname === '/reg') {
10. $(".nav-collapse .nav li:nth-child(3)").addClass("active");
11. }
12. })

index.js

**[javascript]** [view plain](http://blog.csdn.net/qq20004604/article/details/52019904" \o "view plain) [copy](http://blog.csdn.net/qq20004604/article/details/52019904" \o "copy)

1. $(document).ready(**function** () {
3. //清空输入框
4. $("#clearBlog").click(**function** () {
5. $("#textarea").val("");
6. })
7. **var** lastSubmit = **null**;
8. **var** timeer = **null**;
9. **var** successSubmit = **null**;
10. //防止连续点击
11. **function** ErrorAlert() {
12. **if** (**new** Date() - lastSubmit < 3000) {
13. clearTimeout(timeer);
14. }
15. lastSubmit = **new** Date();
16. timeer = setTimeout(**function** () {
17. $("#submitError").addClass("displayNONE");
18. }, 3000);
19. }
21. //提交输入框
22. $("#postBlog").click(**function** () {
23. //防止重复发表，因此需要间隔10秒
24. **if** (successSubmit && **new** Date() - successSubmit < 10000) {
25. //这里是警告提示，防止连续发送消息
26. $("#submitError").text("你发的太快了，喝喝茶吧！距离下次可以发送消息的时间还有：" + (10 - (**new** Date() - successSubmit) / 1000).toFixed(0) + " 秒。");
27. $("#submitSuccess").addClass("displayNONE");
28. $("#submitError").removeClass("displayNONE");
29. //防止连续点击
30. ErrorAlert()
31. **return**;
32. }
34. **var** text = $("#textarea").val();
35. **if** (text.length === 0) {    //禁止发送空内容
36. $("#submitError").text("请填写输入内容");
37. $("#submitError").removeClass("displayNONE");
38. //防止连续点击
39. ErrorAlert()
40. **return**;
41. }
42. **var** length = 0;
43. //获取输入长度，英文字母为1，中文汉字为2
44. **for** (**var** i = 0; i < text.length; i++) {
45. **if** (text[i].match(/[^\x00-\xff]/ig) != **null**)
46. length += 2;
47. **else**
48. length += 1;
49. }
50. **if** (length > 255) {
51. $("#submitError").text("字符长度过长，限制字符长度为255个字节，你的文本长度为" + length + "个字节");
52. $("#submitError").removeClass("displayNONE");
53. ErrorAlert()
54. **return**;
55. }
57. //先清除输入框再提交
58. $("#textarea").val("");
59. successSubmit = **new** Date();
60. $.post('/post', {text: text}, **function** (item) {
61. console.log(item);
62. **if** (item.code == 403) {
63. location.href = item.data;
64. }
65. **else** **if** (item.code == 500) {
66. successSubmit = 0;
67. $("#submitError").text(item.data);
68. $("#submitError").removeClass("displayNONE");
69. setTimeout(**function** () {
70. $("#submitError").addClass("displayNONE");
71. }, 3000);
72. } **else** **if** (item.code == 200) {
73. $("#submitSuccess").text(item.data);
74. $("#submitError").addClass("displayNONE");
75. $("#submitSuccess").removeClass("displayNONE");
76. setTimeout(**function** () {
77. $("#submitSuccess").addClass("displayNONE");
78. }, 3000);
79. }
80. })
81. })

84. **var** lastLoad = 0;   //初始时间设置为0
85. **var** lastLoadCount = 0;  //偏移量，防止读取到重复数据
86. **var** loadingBlog = **null**; //加载用的定时器
87. //加载更多的内容
88. **function** loadMoreContent() {
89. //控制加载间隔，时间差大于2秒，且当前不处于加载中
90. **if** (**new** Date() - lastLoad > 2000 && $("#scrollToFoot").innerHTML !== "加载中~~~") {
91. lastLoad = **new** Date();
92. $("#scrollToFoot")[0].innerHTML = "加载中~~~";
93. //设置重置，假如超时10秒，那么允许再次提交
94. setTimeout(**function** () {
95. **if** ($("#scrollToFoot").innerHTML === "加载中~~~") {
96. $("#scrollToFoot").text("加载失败");
97. }
98. }, 10000);
99. $.get('/loadblog/' + lastLoadCount, **function** (obj) {
100. console.log(obj);
101. //这里假设item的格式：
102. **if** (obj.code === 500) {
103. $("#scrollToFoot").text("加载失败");
104. } **else** **if** (obj.code === 501) {  //加载完毕
105. $("#scrollToFoot").text(obj.data);
106. lastLoad = **new** Date() + 10000000;   //禁止继续请求
107. clearInterval(loadingBlog);
108. } **else** {
109. obj.data.forEach(**function** (item) {
110. $("#content").append('<div class="span4"><h2>' + item.user + '说</h2>' +
111. '<p>' + item.text + '</p></div>');
112. })
113. $("#scrollToFoot").innerHTML = "滚动到底部然后加载内容";
114. lastLoadCount += obj.offset;
115. }
116. })
117. }
118. }
120. //加载完毕时，加载一次内容
121. loadMoreContent();
122. //向下滚动时加载内容
123. loadingBlog = setInterval(**function** () {
124. **if** ($(document).height() - $(window).scrollTop() - $(window).height() < 200) {
125. loadMoreContent();
126. }
127. }, 100);
128. })

**（41）博客的优化（1）**

【1】全局样式和独立样式的分离

我将样式表分割为全局表和独立表；

* 在全局的样式里，是一些通用的样式，他们的特点是复用率高，例如设置隐藏效果的.displayNONE，他只有一条属性那就是display:none;加载全局样式时，样式表在head部分。
* 在独立的样式里，是一些当前页面专有的样式，其加载链接时放在jade的对应的模块里面进行加载。为了防止污染，将当前jade里的dom结点其尽量在同一个div下面，这个div主要用于使用后代样式选择器识别。

【2】初步调节：

index.jade

extends layout  
**block content  
 link**(rel='stylesheet', href='./stylesheets/index.css')  
 **div**.*Blog-index* **div**.*hero-unit* **h1 #{**index.**title}  
 p #{**index.**description}  
 br  
 br** *//修改以下部分* if(!user())  
 **br  
 br  
 a**.*btn*.*btn-primary*.*btn-large*(href="/login") **#{**index.**login}  
 a**.*btn*.*btn-large*(href="/reg") **#{**index.**regNow}** else  
 **textarea**#textarea.*uneditable-input* **button**#postBlog.*btn*.*btn-large* **#{**index.**submit}  
 button**#clearBlog.*btn*.*btn-large* **#{**index.**emptyInput}  
 div**#submitError.*alert*.*alert-error*.*displayNONE*.*myalert* **div**#submitSuccess.*alert*.*alert-success*.*displayNONE* **div**.*row*.*content*#content  
 **div**.*span4* **h2 烟雨江南说  
 p 当欲望没有了枷锁，就没有了向前的路  
 p 只有转左，或者向右  
 p 左边是地狱，右边也是地狱  
 div**.*span4* **h2 烟雨江南说  
 p 那些都是极好极好的  
 p 可是我偏偏不喜欢  
 div**.*span4* **h2 烟雨江南说  
 p 我不怕傻  
 p 只怕  
 p 遇不到  
 p 可以让我变傻的人  
 div**.*span4* **h2 烟雨江南说  
 p 人在年轻的时候总会有些莫名的坚持，  
 p 并且以此感动着自己，  
 p 却时常会在不经意间让真正重要的东西从指间流走。  
 div**.*span4* **h2 烟雨江南说  
 p 记忆真是一种奇怪的东西，  
 p 有时候会涤荡所有的苦难，只留下温情，  
 p 有时候却磨灭掉曾有的欢乐，唯剩下苍白和丑陋。  
 div**.*span4* **h2 烟雨江南说  
 p 那存在的，都是幻影。  
 p 那永恒的，终将毁灭。  
 p 世界万物，缤纷色彩，都是被蒙蔽的人心罢了。  
 div**.*span4* **h2 烟雨江南说  
 p 诸神以真相示人，而世人却视而不见  
 div**.*span4* **h2 烟雨江南说  
 p 只有绵羊会向狮子要求平等，  
 p 而狮子们从来不会这样想。  
 div**.*span4* **h2 烟雨江南说  
 p 愿迷途的旅人，从此得享安息。  
 p 因理想而不朽，因归返而救赎。  
 div**#scrollToFoot **滚动到底部然后加载内容**

layout.jade

**doctype html  
html  
 head  
 title #{**layout.**title}  
 link**(rel='stylesheet', href='./stylesheets/bootstrap.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/bootstrap-responsive.min.css')  
 **link**(rel='stylesheet', href='./stylesheets/global.css')  
 **script**(type="text/javascript",src='javascripts/jq.js')  
 **script**(type="text/javascript",src='javascripts/bootstrap.min.js')  
 **script**(type="text/javascript",src='javascripts/blog.js')  
 **body  
 div**.*navbar*.*navbar-fixed-top* **div**.*navbar-inner* **div**.*container* **a**.*btn*.*btn-navbar*(data-toggle="collapse",data-target=".nav-collapse")  
 **span**.*icon-bar* **span**.*icon-bar* **span**.*icon-bar* **a**.*brand*(href="/") **#{**layout.**topTitle}  
 div**.*nav-collapse* **ul**.*nav* **li  
 a**(href="/") **#{**layout.**indexButton}** *//以下是修改后的部分* if(user())  
 **li  
 a**(href="/logout") **#{**layout.**logout}** else  
 **li  
 a**(href="/login") **#{**layout.**login}  
 li  
 a**(href="/reg") **#{**layout.**reg}  
 li  
 a**(href="/language") **#{**layout.**language}  
 span**.*brand*(style='float:right') **#{**layout.**foot}  
 div**#contrainer.*container* - var ErrorAlert = err()  
 if(ErrorAlert)  
 **div**.*alert*.*alert-error* **#{**ErrorAlert**}  
 block content  
 hr  
 fotter  
 p #{**layout.**foot}**

global.css

body {  
 padding-top: 60px;  
 padding-botom: 40px;  
}  
  
.displayNONE {  
 display: none;  
}  
  
fotter p {  
 margin-top: 40px;  
 padding-top: 20px;  
 border-top: 1px solid #aaa;  
 font-size: 18px;  
 text-align: center;  
}

index.css

.Blog-index #textarea {  
 resize: none;  
 width: 300px;  
 height: 100px;  
 cursor: text;  
}  
  
.Blog-index #postBlog {  
 position: relative;  
 left: 20px;  
 vertical-align: top;  
}  
  
.Blog-index #clearBlog {  
 position: relative;  
 left: -90px;  
 top: 27px;  
 width: 110px;  
 height: 44px;  
}  
  
.Blog-index .myalert {  
 position: absolute;  
}  
  
.Blog-index #scrollToFoot {  
 border: 1px solid #ccc;  
 text-align: center;  
 font-size: 18px;  
 padding: 20px 0;  
}  
  
  
.Blog-index .row {  
 color: #555;  
}

样式目前依然和之前保持一致

【3】添加localStorage用于计数用户第几次访问本网站

简单介绍localStorage：他类似cookie，用于存储数据。他的特点是：

①存储于本地，不发送到服务器；

②使用简单，localStorage.变量名 即可

③清除历史记录后被清空；

④浏览器关闭后依然存在。

⑤HTML5新增

因此，只需要更改前端代码即可。

修改blog.js

**$**(**document**).ready(function () {  
 //根据url，设置高亮  
 if (**window**.**location**.**pathname** === '/') {  
 **$**(".nav-collapse .nav li:first-child").addClass("active");  
 }  
 else if (**window**.**location**.**pathname** === '/login') {  
 **$**(".nav-collapse .nav li:nth-child(2)").addClass("active");  
 }  
 else if (**window**.**location**.**pathname** === '/reg') {  
 **$**(".nav-collapse .nav li:nth-child(3)").addClass("active");  
 }  
  
  
 //给原生的Date添加一个方法，传递的参数是格式  
 Date.**prototype**.Format = function (fmt) { //author: meizz  
 var obj = {  
 **"M+"**: this.getMonth() + 1, //月份  
 **"d+"**: this.getDate(), //日  
 **"H+"**: this.getHours(), //小时  
 **"m+"**: this.getMinutes(), //分  
 **"s+"**: this.getSeconds(), //秒  
 **"q+"**: **Math**.floor((this.getMonth() + 3) / 3), //季度  
 **"S"**: this.getMilliseconds() //毫秒  
 };  
 if (/(y+)/.test(fmt))  
 fmt = fmt.replace(RegExp.**$1**, (this.getFullYear() + "").substr(4 - RegExp.**$1**.**length**));  
 for (var key in obj)  
 if (new RegExp("(" + key + ")").test(fmt)) fmt = fmt.replace(RegExp.**$1**, (RegExp.**$1**.**length** == 1) ? (obj[key]) : (("00" + obj[key]).substr(("" + obj[key]).**length**)));  
 return fmt; //返回值是格式化好之后的时间  
 }  
  
 var testLocalStorage = function () { //鉴别浏览器是否支持localStorage，如果支持则返回true  
 try {  
 return 'localStorage' in **window** && **window**['localStorage'] !== null;  
 } catch (e) {  
 return false;  
 }  
 }  
 if (testLocalStorage()) {  
 if (!**localStorage**.**visits**) {  
 **localStorage**.**visits** = 1;  
 } else {  
 **localStorage**.**visits**++;  
 }  
  
 **$**("fotter p")[0].**innerHTML** += "<br>在上次清除历史记录后，localStorage技术告诉我，你是第" + **localStorage**.**visits** + "次打开本页面。" + (**localStorage**.**lastVisitTime** ? ("你上次访问时间是：" +  
 **localStorage**.**lastVisitTime**) : (""));  
 **localStorage**.**lastVisitTime** = new Date().Format("yyyy-MM-dd HH:mm:ss");  
 }  
})

【4】轮播图

轮播图写于index.jade的div.Blog-index之下，和div.hero-unit平级

因此，需要固定hero-unit的行高，确保其不会因为内容量的变化而变化。

（42）**博客的优化（2）——用fs模块来记录访问情况**

讲道理说呢，我觉得用mysql来记录访问情况更好一点，不过为了学习使用fs模块，我这里就使用fs模块好了。

需求：

①以ip为参考；

②重复ip两次访问时间间隔在一小时内的，算一次访问，并且更新最新访问时间；

③同一ip两次访问时间间隔超过1个小时的，算是一次新的访问；

④只保留最近一小时的访问记录；（具体实施时，采取下次添加新的访问记录时，删除超过1个小时的记录）；

步骤：

①首先，无论访问哪个页面时，都应该对访问ip进行处理，因此打开app.js，在所有路由之前添加如下代码：

var visitCount = require('./models/visit');  
app.use('/', visitCount);

这段代码的意思是，无论你访问什么路径，都先交给visitCount这个模块进行处理；

②在models文件夹下建立visit.js文件；

首先，我们打算利用fs模块，那么要引入；

其次，路由模块也得正常引入；

第三，我们要让无论是什么路径都能处理，因此要在路径时使用“/:xxx”这样的方式，来匹配任何路径；

第四，要支持响应任何类型请求，比如get、post等，因此应该用all；

因此写下以下代码：

var fs = require("fs");  
var express = require('express'); // 调用express模块  
var router = express.**Router**(); // 调用模块的Router方法  
  
router.all("/:anyurl", function (req, res, next) {  
 //处理代码  
 next(); //交给之后的路由继续处理  
})

module.exports = router;

这里没有写处理代码，我们需要将对访问进行记录的代码，放在处理代码的位置，即next()命令之前；

③处理访问记录；

第一个，我们要能获取访问ip，方法是：

req.connection.remoteAddress

其次，我们要能读取文件，这个文件存储了总的访问次数和最近一小时的访问ip；

文件以json格式存储，方便**JSON.parse()**解析，因此存储时需要将对象转为JSON字符串，即使用**JSON.stringify()**命令；

第三，为了方便，存储时间时，将Date类型对象转为毫秒的number类型；

第四，调用fs.readFile来读取，fs.writeFile来写入；

第五，要清除过期的访问记录

声明一个函数，他有一个方法：

function visitCount() {  
 this.visitCheck = function (ip) {  
 fs.readFile("./LastVisitIP.json", "utf8", function (err, data) {  
 if (err) {  
 throw err;  
 }  
 // 因此读取的文件可能存在字符串开头是空白字符，如果不移除则无法解析  
 // 推测是因为UTF8格式的txt文件会在开头占用三个字节；  
 data = data.replace(/^\s\*/, "");  
 var obj = **JSON**.parse(data);  
  
 // 先检查当前的，如果时间过期（一小时），则删除这条访问记录  
 var nowTime = new Date().getTime();  
 for (var i in obj) {  
 if (i != "visitNumber" & nowTime - obj[i] > 3600000) {  
 delete obj[i];  
 }  
 }  
  
 if (ip in obj) { //如果还有记录，说明是重复访问  
 **console**.log("IP:" + ip + ", last visit time is " + new Date(obj[ip]))  
 } else { //否则是非重复访问  
 obj[ip] = nowTime; //添加访问记录，形式是key为ip，val为访问时间转为毫秒后的形式  
 obj.visitNumber++; //访问计数+1  
 **console**.log("IP:" + ip + " is the first visit.The visit time is " + new Date(obj[ip]));  
 }  
   
 //将新的数据写入  
 fs.writeFile("./LastVisitIP.json", **JSON**.stringify(obj), "utf8", function (err) {  
 if (err) {  
 throw err;  
 }  
 })  
 })  
 };  
}

使用时，生成一个他的实例，然后调用它的visitCheck方法，参数传当前访问的ip；

var count = new visitCount();  
count.visitCheck(req.connection.remoteAddress);

将以上代码放在路由中的next()之前，便能满足我们的需求。