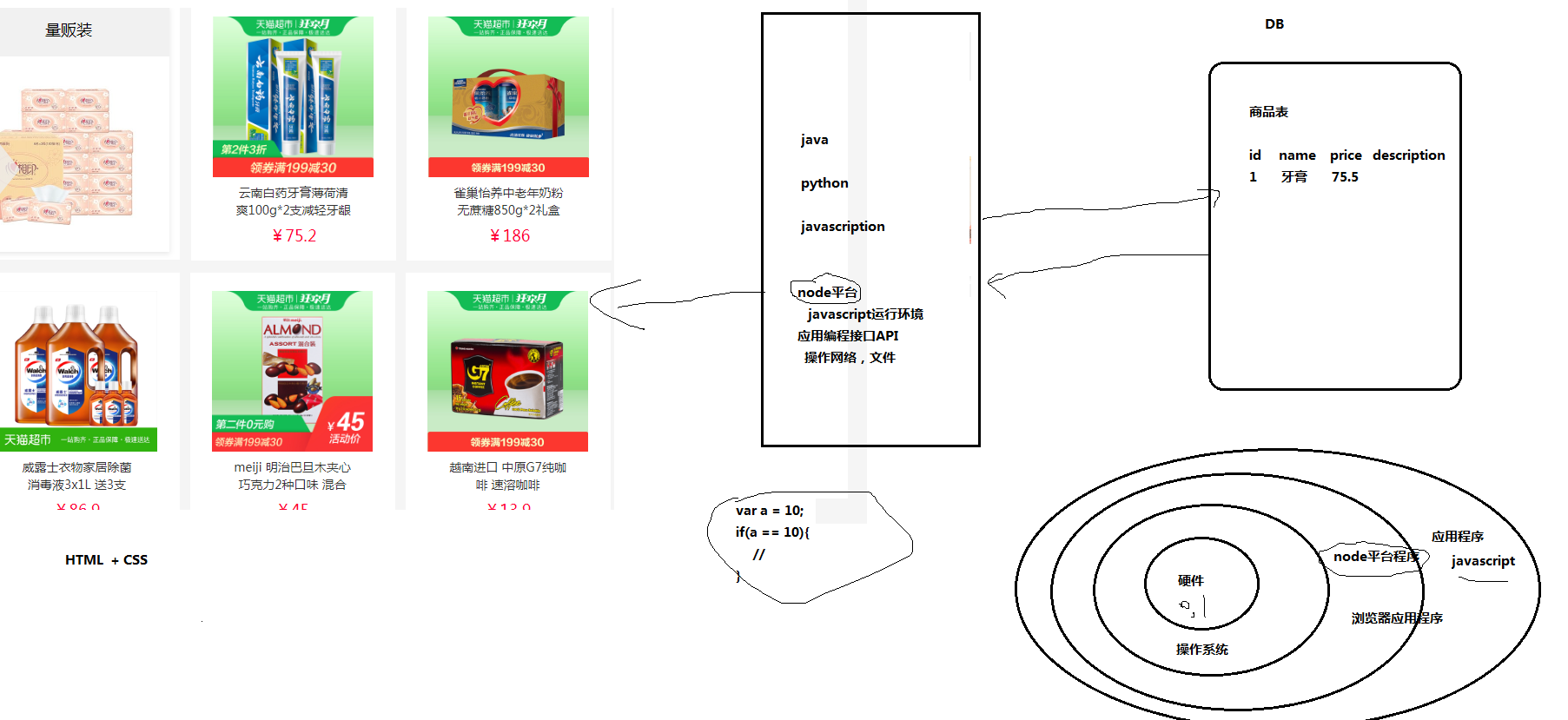
# Nodejs*学习笔记*

1. NodeJs介绍
2. 什么是NodeJS，在应用程开发中起什么作用？

Nodejs是一个应用编程平台，能运行javascript语言编写的代码，提供了javascript运行环境，基于Google的V8引擎，V8引擎执行Javascript的速度非常快，性能非常好。提供了大量应用编程接口API,在处理http网络、数据库、文件等操作时非常方便.



1. NodeJS开发环境安装
2. 官网下载地址<http://nodejs.cn/download/>



1. 检测是否安装成功

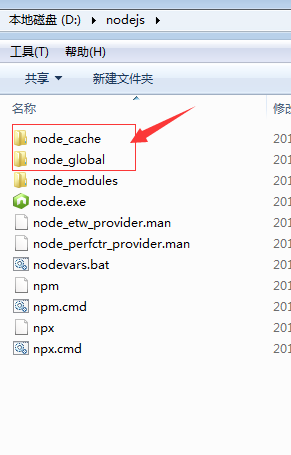
打开命令窗口:输入 node –v

#>v10.11.3

出现如上提示表示安装成功

1. 环境配置(若不配置第三方全局安装包和缓存默认在c盘下)

第一步：在nodejs安装目录下新建: ”node\_global”及”node\_cache”两个文件夹

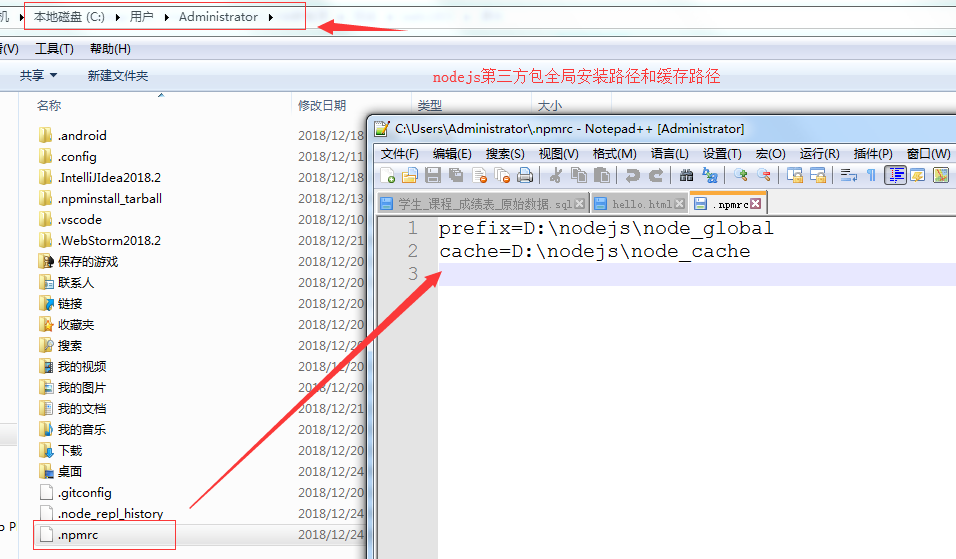


第二步: cmd 中输入如下命令

$ npm config set prefix “d:\nodejs\node\_global” //设置全局安装路径

$ npm config set cache “d:\nodejs\node\_cache” //设置缓存路径

* 第三步： 查看路径设置

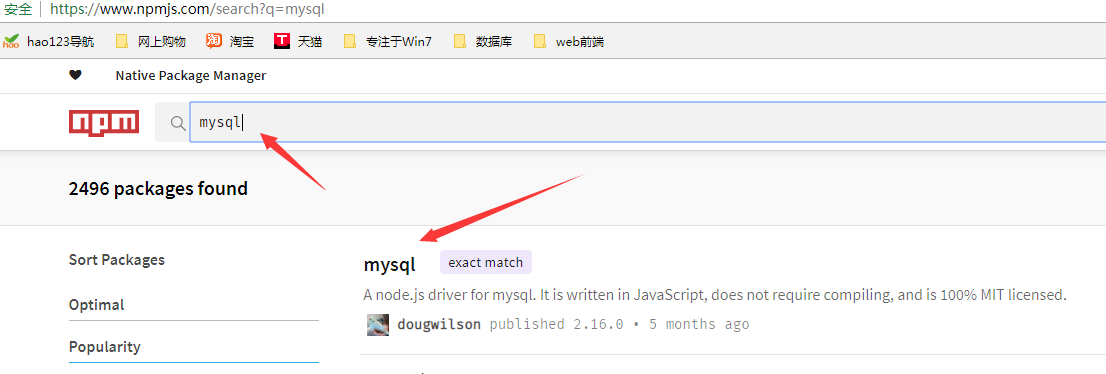


## 3.NPM nodejs包管理工具

1. 什么是npm?

中文文档： <https://www.npmjs.cn/>

官网: <https://www.npmjs.com/>





1. 检查npm是否安装

$ npm –v

1. 设置npm服务器为淘宝镜像服务器

$ npm config set registry https://registry.npm.taobao.org

1. npm包安装方式: 全局安装 和本地安装

npm install 包名 //本地安装 -安装包在当前项目目录下

npm install -g 包名(mysql) //全局安装-安装包在配置的nodejs全局路径下,如: D:\nodejs\node\_global\node\_modules

注: 如果全局安装包命令无效，需设置NODE\_PATH环境变量

NODE\_PATH: D:\nodejs\node\_global //全局安装包所在路径

Path: %NODE\_PATH%

1. cnpm 使用淘宝 NPM 镜像

$ npm install –g

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

$ cnpm install –g mysql

# Javascript

## 导论

### JavaScript历史

要了解JavaScript，我们首先要回顾一下JavaScript的诞生。

在上个世纪的1995年，当时的网景公司正凭借其Navigator浏览器成为Web时代开启时最著名的第一代互联网公司。

由于网景公司希望能在静态HTML页面上添加一些动态效果，于是叫Brendan Eich这哥们在两周之内设计出了JavaScript语言。你没看错，这哥们只用了10天时间。

为什么起名叫JavaScript？原因是当时Java语言非常红火，所以网景公司希望借Java的名气来推广，但事实上JavaScript除了语法上有点像Java，其他部分基本上没啥关系。

### ECMAScript

因为网景开发了JavaScript，一年后微软又模仿JavaScript开发了JScript，为了让JavaScript成为全球标准，几个公司联合ECMA（European Computer Manufacturers Association[欧洲计算机制造商协会]）组织定制了JavaScript语言的标准，被称为ECMAScript标准。

所以简单说来就是，**ECMAScript是一种语言标准，而JavaScript是网景公司对ECMAScript标准的一种实现**。

那为什么不直接把JavaScript定为标准呢？因为JavaScript是网景的注册商标。

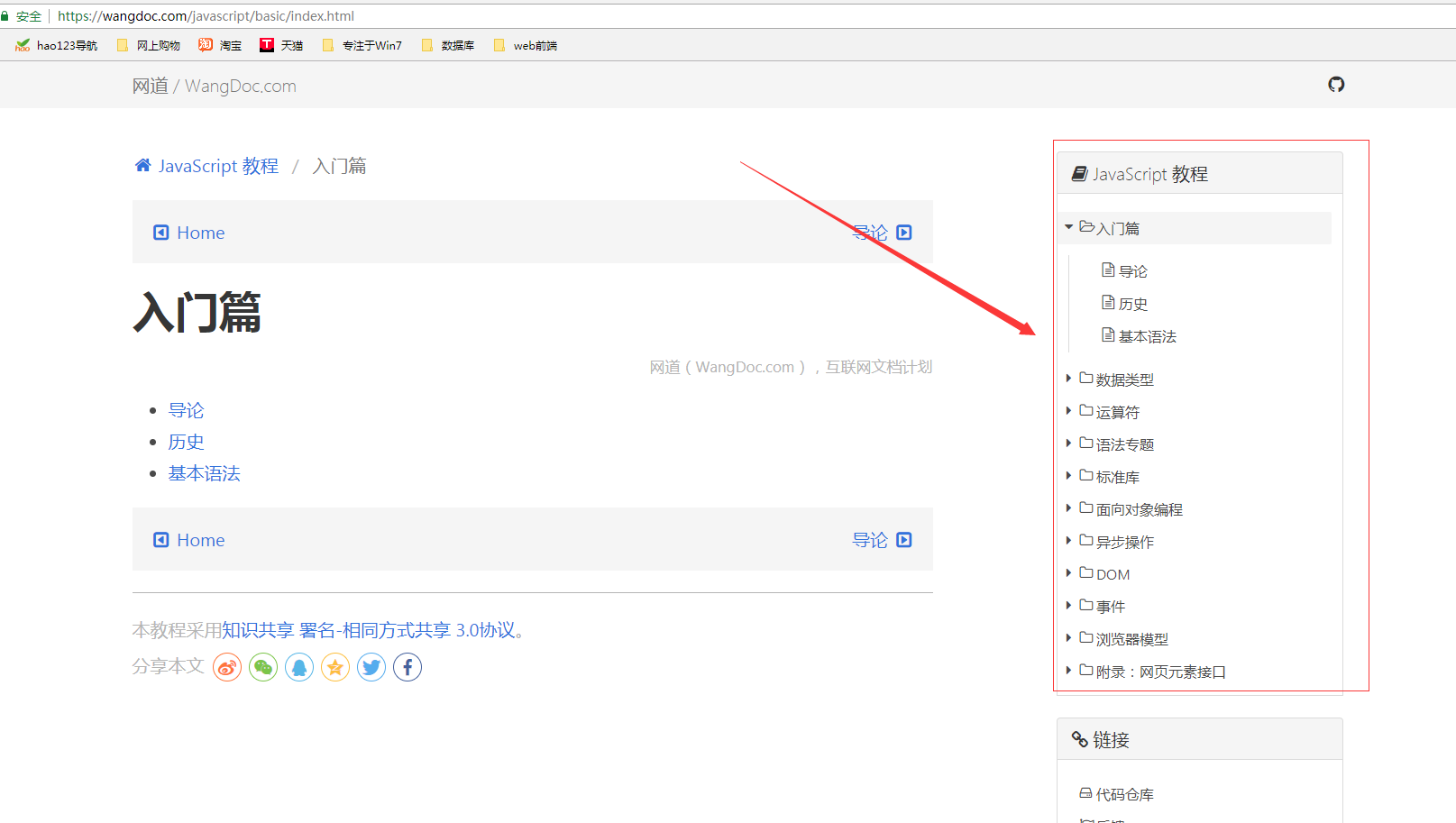
不过大多数时候，我们还是用JavaScript这个词。如果你遇到ECMAScript这个词，简单把它替换为JavaScript就行了。

### JavaScript版本

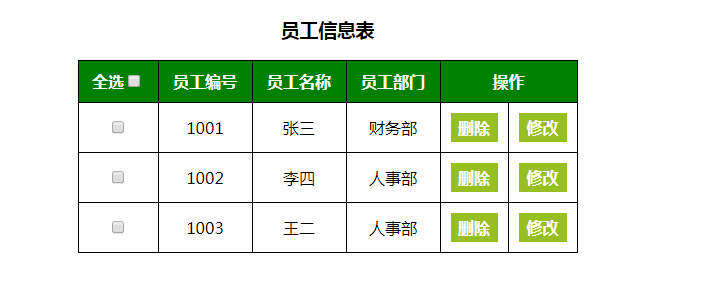
JavaScript的标准——ECMAScript在不断发展，最新版ECMAScript 6标准（简称ES6）已经在2015年6月正式发布了

## 2.语法教程

https://wangdoc.com/javascript/basic/index.html



### 练习1-全选框练习





|  |
| --- |
| <script type="text/javascript">  /\*\*全选按钮\*/  function allCheckboxOne(){  //document.getElement系列  var checkbox = document.getElementById("allcheckbox"); //全选框  var singleCheckboxs = document.getElementsByName("checkbox"); //所有选项  for(var i = 0; i < singleCheckboxs.length; i++){  if(checkbox.checked == true){  singleCheckboxs[i].checked = true;  }else{  singleCheckboxs[i].checked = false;  }  }  }    /\*\*所有选项\*/  function selectSingle(){  var checkbox = document.getElementById("allcheckbox"); //全选框  var singleCheckboxs = document.getElementsByName("checkbox"); //所有选项  var k = 1; //全选框是否选中 0不选中，1选中  for(var i = 0; i < singleCheckboxs.length; i++){  if(singleCheckboxs[i].checked == false){  k = 0;  break;  }  }  if(k == 0){  checkbox.checked = false;  }else{  checkbox.checked = true;  }    }    </script> |

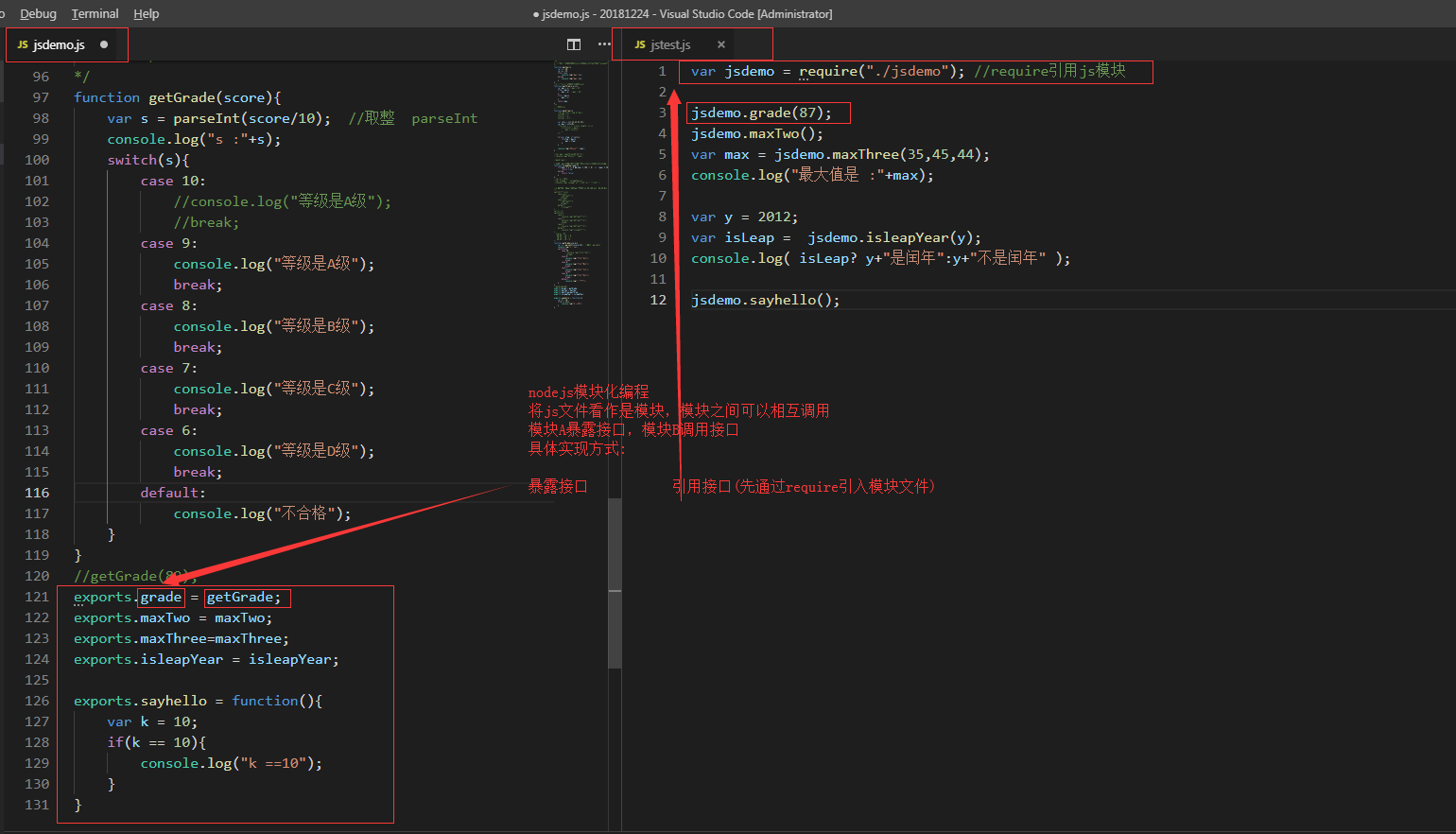
### js基础语法练习2

# 三、nodejs

## 1.模块化编程

将每个js文件看作是一个模块，它们之间可以相互调用.

Node.js 提供了 exports 和 require 两个对象，其中 exports 是模块公开接口对象（用于定义公开模块接口），require 用于从外部获取一个模块的接口，即所获取模块的 exports 对象。



有时候我们只是想把一个对象封装到模块中，格式如下：

module.exports = function() {

// ...

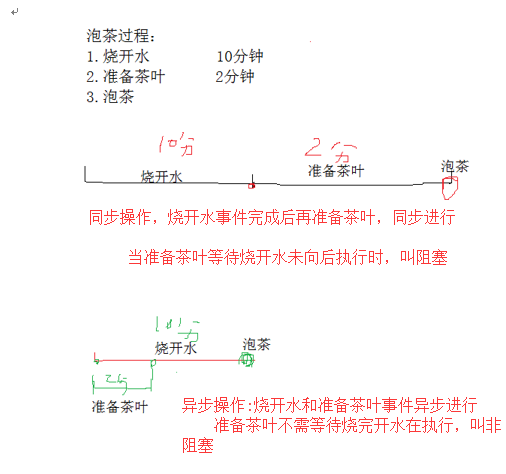
}

模块接口的唯一变化是使用 module.exports = Hello 代替了exports.world = function(){}。 在外部引用该模块时，其接口对象就是要输出的 Hello 对象本身，而不是原先的 exports。

|  |
| --- |
| module.exports与exports的介绍  module.exports与exports都是将函数或者是方法暴露出去，require的时候进行调用  module.export 暴露整个对象  module.export 如果后面跟函数表示将函数使用对象暴露  a.js  module.exports = function() {  // ...  }    b.js  var oneObj = requery(‘a.js’);  oneObj(); //执行函数  exports暴露指定函数,可看作module.exports对象引用  exports.函数=函数|属性；  a.js  exports.one = function() {  // ...  }  exports.two = function(){  //…  }  Var k1 = 10;  exports.k = k1;    b.js  var exc = requery(‘a.js’);  exc.one();  exc.two();  exc.k; |

## 异步与同步, 理解回调函数、阻塞与非阻塞

### 理解异步同步和阻塞非阻塞



### 文件模块

var fs = require("fs");

//同步读文件

var data = fs.readFileSync('src/test.json'); //10分钟

console.log("准备茶叶>>>"); //2分钟

console.log(data.toString());

//异步读文件

fs.readFile("src/test.json", function(err,data){

for(var i =0; i < 100; i++){

console.log(" ");

}

console.log(data.toString());

});

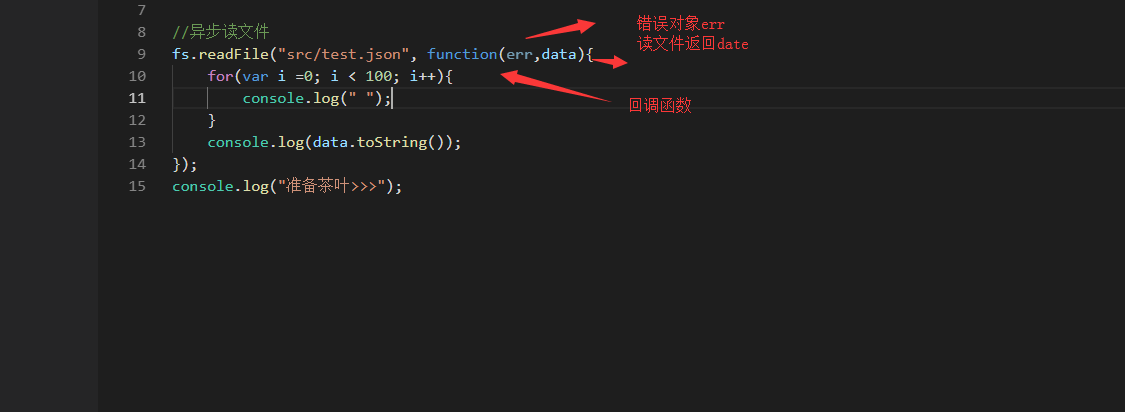
console.log("准备茶叶>>>");

### 回调函数

Node.js 异步编程的直接体现就是回调。

回调函数在完成任务后就会被调用

例如，我们可以一边读取文件，一边执行其他命令，在文件读取完成后，我们将文件内容作为回调函数的参数返回。这样在执行代码时就没有阻塞或等待文件 I/O 操作。这就大大提高了 Node.js 的性能，可以处理大量的并发请求。



## package.json文件说明解释

### 1、package.json是什么？

1. 什么是Node.js的模块（Module）？在Node.js中，模块是一个库或框架，也是一个Node.js项目。Node.js项目遵循模块化的架构，当我们创建了一个Node.js项目，意味着创建了一个模块，这个模块的描述文件，被称为package.json。作为一个描述文件，描述了你的项目依赖哪些包和项目模块相关信息

### 2、package.json属性说明

**name** - 包名.  
    **version** - 包的版本号。  
    **description** - 包的描述。  
    **homepage** - 包的官网URL。  
    **author** - 包的作者，它的值是你在[https://npmjs.org](https://link.jianshu.com?t=https%3A%2F%2Fnpmjs.org%2F" \t "_blank)网站的有效账户名，遵循“账户名<邮件>”的规则，例如：zhangsan <zhangsan@163.com>。  
    **contributors** - 包的其他贡献者。  
    **dependencies / devDependencies** - **生产**/**开发**环境依赖包列表。它们将会被安装在 node\_module 目录下。  
    **repository** - 包代码的Repo信息，包括type和URL，type可以是git或svn，URL则是包的Repo地址。  
    **main** - main 字段指定了程序的主入口文件，require('moduleName') 就会加载这个文件。这个字段的默认值是模块根目录下面的 index.js。  
    **keywords** - 关键字

       上述参数是极为常见的参数，另外还可以设置script、license等等。除了官方必须的一些参数外，我们也可以存储我们自己的关于模块的描述信息在package.json。

### 3、生成package.json文件

$ npm init

### 4、关于版本号的描述 https://www.jianshu.com/p/b3d86ddfd555

### 5. 指定依赖的包

安装包时 我们需要在 package.json 文件中指定项目依赖的包，这样别人在拿到这个项目时才可以使用 npm install 下载。

包有两种依赖方式：

dependencies：在生产环境中需要用到的依赖

devDependencies：在开发、测试环境中用到的依赖

### 6.安装参数 --save 和 --save -dev

添加依赖时我们可以手动修改 package.json 文件，添加或者修改 dependencies devDependencies 中的内容即可。

另一种更酷的方式是用命令行，在使用 npm install 时增加 --save 或者 --save -dev 后缀：

npm install <package\_name> --save //表示将这个包名及对应的版本添加到 package.json的 dependencies

npm install <package\_name> --save-dev //表示将这个包名及对应的版本添加到 package.json的 devDependencies

## 第三方模块-mysql数据库操作

var mysql = require("mysql");

//创建数据库连接对象

var connection = mysql.createConnection(

{

host: "localhost",

user: "root",

password: "b123",

database: "ex1db"

}

);

connection.connect(); //打开数据库连接

var sql = "SELECT e\_no,e\_name,e\_sex,dept\_no,e\_salary,e\_job,hire\_date FROM employee";

connection.query(sql,function(err,data){

if(err){

console.log("数据库操作出错"+err);

return;

}

console.log("姓名\t性别\t 薪酬\t工作" );

for(var i = 0; i < data.length; i++){

console.log(data[i].e\_name +"\t"+ data[i].e\_sex +"\t"+ data[i].e\_salary+ "\t" + data[i].e\_job );

}

});

connection.end(); //关闭数据库连接

## 5.数据库封装-模块化操作

### 1.创建数据库公共模块db.js

var mysql = require("mysql");

function query(sql,parmeters callFunction) {

//创建数据库连接对象

var connection = mysql.createConnection(

{

host: "192.168.40.114",

user: "test",

password: "b123",

database: "ex1db"

}

);

connection.connect(); //打开数据库连接

connection.query(sql, parmeters ,callFunction); //执行sql

connection.end(); //关闭数据连接

}

exports.query = query;

//module.exports = query;

### 封装具体数据库表操作模块 departmentDao.js

var mysqldb = require("./db");

/\*\*

\* 查询部门信息

\*/

function findDepartment() {

var sql = "SELECT \* FROM department";

mysqldb.query(sql, function (err, data) {

if (err) {

console.log("数据库出错" + err);

return;

}

console.log(JSON.stringify(data));

for (var department of data) {

console.log(department.d\_name + "\t" + department.d\_location);

}

});

}

/\*\*

\* 插入部门数据

\*/

function addDepartment() {

var sql1 = "INSERT INTO department (d\_no,d\_name,d\_location)VALUES(50,'技术部','成都')";

mysqldb.query(sql1, function (err, data) {

if (err) {

console.log("插入数据出错" + err);

return;

}

console.log("插入数据成功!");

});

}

exports.findDepartment = findDepartment;

exports.addDepartment = addDepartment;

### 实现测试数据库操作模块 departmentDaoTest.js

var userDao = require("./departmentDao ");

userDao.findDepartment();

### 4.传参使用占位符

/\*\*

\* 插入部门数据

\* @param {编号} no

\* @param {部门名} name

\* @param {位置} location

\*/

function addDepartment(no, name, location) {

// var sql1 = "INSERT INTO department (d\_no,d\_name,d\_location)VALUES("+no+",'"+name+"','"+location+"')"; //SQL 注入

var sql1 = "INSERT INTO department (d\_no,d\_name,d\_location)VALUES(?,?,?)"; //占位符?

var parmeters = [no, name, location]; //参数 数组对象

query(sql1, parmeters, function (err, data) {

if (err) {

console.log("插入数据出错" + err);

return;

}

console.log("插入数据成功!");

});

}

# Node.js开发Web服务器程序

## 1、Web服务器

为什么需要web服务器,作用是什么?

web服务器也叫网站服务器是基于网站架设的服务器，我们平时可以浏览的网页都是在别人的服务器上面保存的文件．

Web服务器有哪些?

nginx 、 apache、 IIS 等

IIS是Microsoft公司的它是一个收费的服务器软件，而Apache是基于开源代码的服务器软件，它的核心都是用代码来实现的，而且是开源的可以免费使用．

## 2、Nginx服务器

### 1.简介

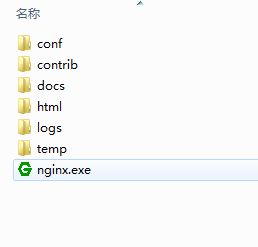
Nginx是一款[轻量级](https://baike.baidu.com/item/%E8%BD%BB%E9%87%8F%E7%BA%A7/10002835" \t "_blank)的[Web](https://baike.baidu.com/item/Web/150564" \t "_blank) 服务器/[反向代理](https://baike.baidu.com/item/%E5%8F%8D%E5%90%91%E4%BB%A3%E7%90%86" \t "_blank)服务器及[电子邮件](https://baike.baidu.com/item/%E7%94%B5%E5%AD%90%E9%82%AE%E4%BB%B6" \t "_blank)（IMAP/POP3）代理服务器，其特点是占有内存少，[并发](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91" \t "_blank)能力强，事实上nginx的并发能力确实在同类型的网页服务器中表现较好，中国大陆使用nginx网站用户有：百度、[京东](https://baike.baidu.com/item/%E4%BA%AC%E4%B8%9C" \t "_blank)、[新浪](https://baike.baidu.com/item/%E6%96%B0%E6%B5%AA" \t "_blank)、[网易](https://baike.baidu.com/item/%E7%BD%91%E6%98%93" \t "_blank)、[腾讯](https://baike.baidu.com/item/%E8%85%BE%E8%AE%AF" \t "_blank)、[淘宝](https://baike.baidu.com/item/%E6%B7%98%E5%AE%9D" \t "_blank)等。

### 2.下载安装

下载地址: <http://nginx.org/>



### 3. 目录结构



### 4. 启动服务

打开命令窗口，进入nginx安装目录，执行命令:nginx



打开浏览器，输入localhost 或者你的IP地址

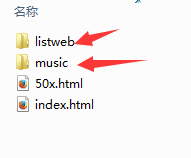
出现如下页面表示服务启动成功



### 5.部署网站

拷贝你写好的网页到nginx安装目录下html子目录





访问网页: <http://192.168.41.16/listweb/list.html>

### 6. nginx常用命令

nginx常用命令：

验证配置是否正确: nginx -t

查看Nginx的版本号：nginx -V

启动Nginx：start nginx

快速停止或关闭Nginx：nginx -s stop

正常停止或关闭Nginx：nginx -s quit

配置文件修改重装载命令：nginx -s reload

错误

nginx: [error] CreateFile() "E:\nginx-1.13.5/logs/nginx.pid" failed

nginx: [error] Open() "E:\nginx-1.13.5/logs/nginx.pid" failed

解决方法:

使用命令创建/logs/nginx.pid文件:

nginx -c conf/nginx.conf

## 3、http协议

### 1、HTTP（超文本传输协议）

    HTTP是一个基于请求与响应模式的、无状态的、应用层的协议，常基于TCP的连接方式，绝大多数的Web开发，都是构建在HTTP协议之上的Web应用。

### 2、HTTP请求响应模式



### 3、HTTP协议的主要特点

1.支持客户/服务器模式。

2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

3.灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

5.无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

### 4、HTTP协议之URL

HTTP URL (URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息)的格式如下：

**http://host[":"port][abs\_path]**

http表示要通过HTTP协议来定位网络资源；host表示合法的Internet主机域名或者IP地址；port指定一个端口号，为空则使用缺省端口80；abs\_path指定请求资源的URI；如果URL中没有给出abs\_path，那么当它作为请求URI时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。

eg:

1、输入：www.guet.edu.cn

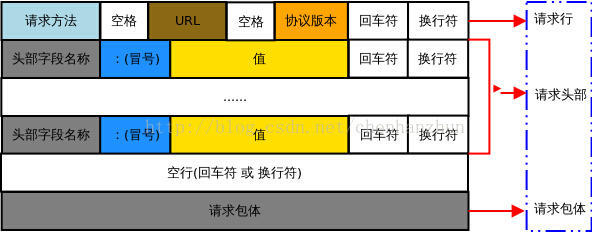
浏览器自动转换成：http://www.guet.edu.cn/

2、http://192.168.0.116:8080/index.html

### 5、HTTP请求request

一次请求就是向目标服务器发送一串文本。什么样的文本？有下面结构的文本。

1.HTTP请求包结构



http请求由三部分组成，分别是：请求行、请求头部、请求包体(正文)

**1>请求行**以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，

格式如下：Method   Request-URL    HTTP-Version   CRLF

其中 Method表示请求方法；Request-URI是一个统一资源标识符；HTTP-Version表示请求的HTTP协议版本；CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

**2>Http请求方式**

方法             描述

**GET**             请求指定url的数据,请求体为空(例如打开网页)。

**POST**           请求指定url的数据，同时传递参数(在请求体中)。

HEAD           类似于get请求，只不过返回的响应体为空，用于获取响应头。

PUT             从客户端向服务器传送的数据取代指定的文档的内容。

DELETE        请求服务器删除指定的页面。

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。

OPTIONS      允许客户端查看服务器的性能。

TRACE         回显服务器收到的请求，主要用于测试或诊断。

常用只有Post与Get。

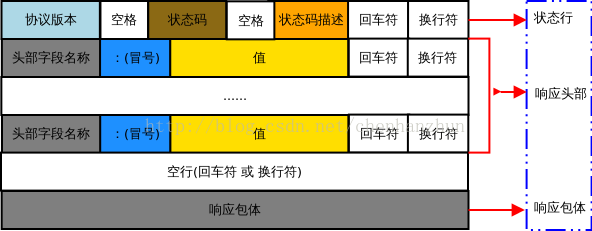
例子：

|  |
| --- |
| POST  /home/user/login.html  HTTP/1.1                                   //请求行      Host: 114.215.86.90      Cache-Control: no-cache      Postman-Token: bd243d6b-da03-902f-0a2c-8e9377f6f6ed      Content-Type: application/x-www-form-urlencoded  username=admin&password=123456                          //请求内容正文  {username:admin,password:123456} |

|  |
| --- |
| GET /home/user/login.html?username=admin&password=123456  HTTP/1.1       //请求行      Host: 114.215.86.90      Cache-Control: no-cache      Postman-Token: bd243d6b-da03-902f-0a2c-8e9377f6f6ed      Content-Type: application/x-www-form-urlencoded |

### 6、HTTP响应response

 HTTP响应包结构



HTTP响应也是由三个部分组成，分别是：状态行、消息报头、响应正文

1、状态行格式如下：

HTTP-Version   Status-Code   Reason-Phrase   CRLF

其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。

状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：

1xx：指示信息--表示请求已接收，继续处理

2xx：成功--表示请求已被成功接收、理解、接受

3xx：重定向--要完成请求必须进行更进一步的操作

4xx：客户端错误--请求有语法错误或请求无法实现

5xx：服务器端错误--服务器未能实现合法的请求

常见状态代码、状态描述、             说明：

**200**             OK                    //客户端请求成功

**400**             Bad Request        //客户端请求有语法错误，不能被服务器所理解

**401**    Unauthorized      //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用

**403**     Forbidden           //服务器收到请求，但是拒绝提供服务

**404**             Not Found           //请求资源不存在，eg：输入了错误的URL

**405**                                       //方法不被允许

**500**             Internal Server Error   //服务器发生不可预期的错误

**503** Server Unavailable  //服务器当前不能处理客户端的请求，一段时间后,可能恢复正常

例子：

    HTTP/1.1  200  OK

    Date: Sat, 02 Jan 2016 13:20:55 GMT

    Server: Apache/2.4.6 (CentOS) PHP/5.6.14

    X-Powered-By: PHP/5.6.14

    Content-Length: 78

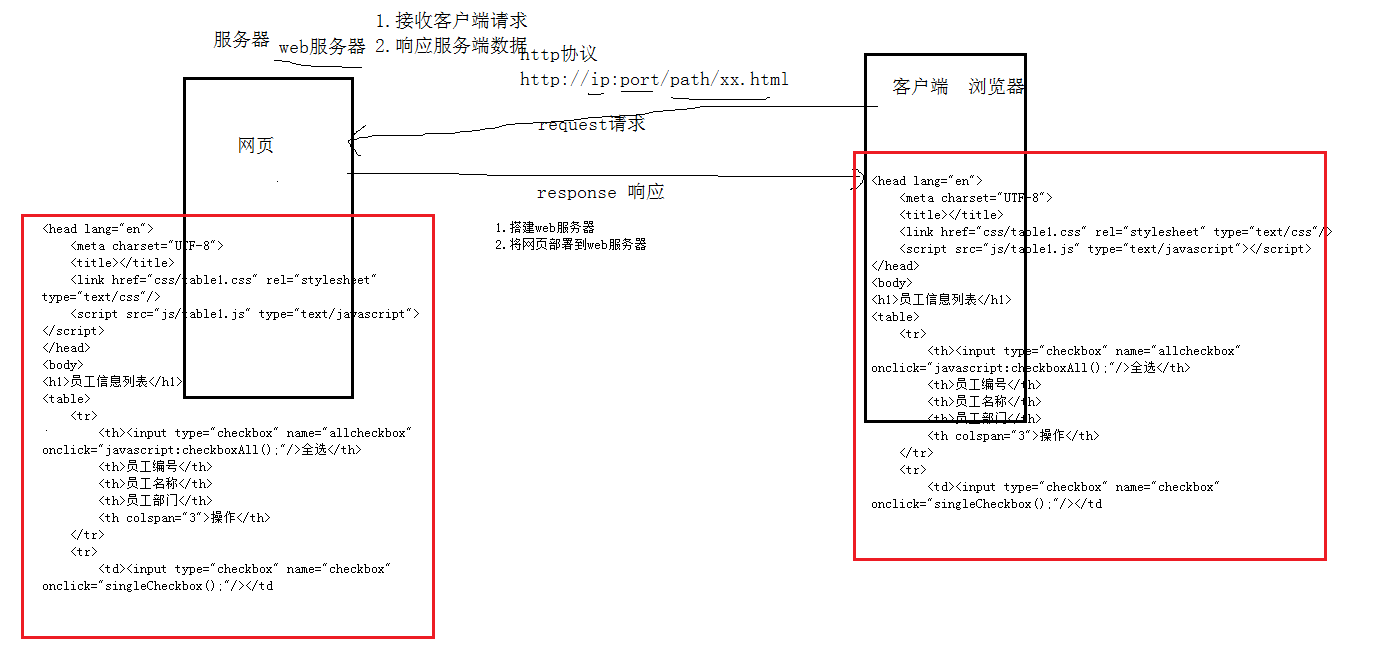
    Keep-Alive: timeout=5, max=100

    Connection: Keep-Alive

    Content-Type: text/html; charset=utf-8

    {"status":202752,"info":"\u6b64\u 8\u6237\u4e0d\u5b58\u5728\uff01","data":null}

### 7、请求响应示例



## 4、用Node.js实现一个HTTP服务器程序

### 1.示例

var http = require("http");

var server = http.createServer(function (request, response) {

response.writeHead(200, {'Content-Type': 'text/plain; charset=utf-8'});//解决中文乱码问题

response.end('{"student ": [{"name":"admin","number":1001},{"name":"小明","number":1002}]}');

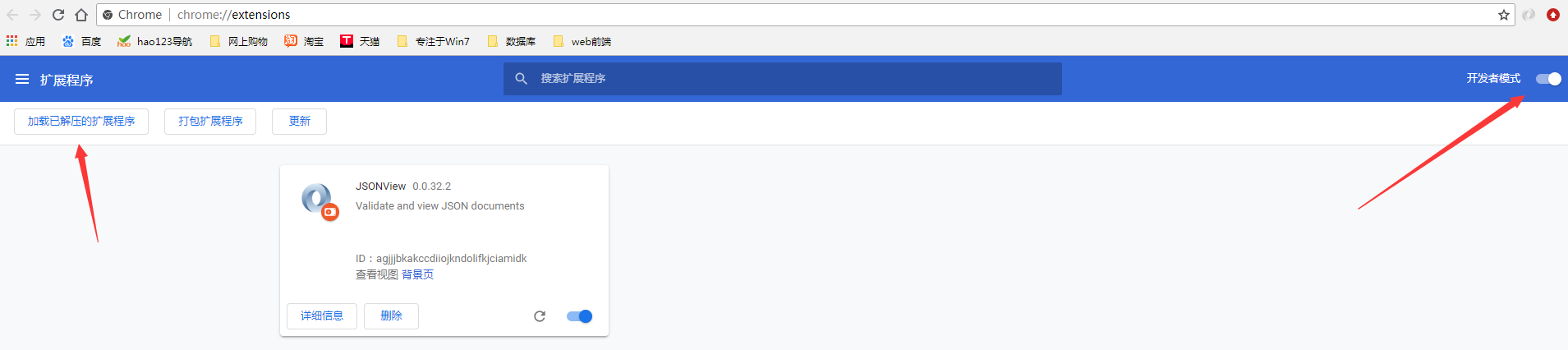
});

server.listen(8080);

console.log("启动web服务器,监听8080; http://192.168.40.114:8080");

### 2.chrom json插件安装

1. 下载离线jsonview插件包
2. 选择更多工具->扩展程序
3. 打开开发者模式->选择离线包



### 3.读取文件数据响应给浏览器客端

var http = require("http");

var fs = require("fs"); //文件模块

var server = http.createServer(function (request, response) {

var filedir ="./data.json";

fs.readFile(filedir, function (err, data) {

if (err) {

console.log("读文件出错" + err);

return;

}

response.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });

response.end(data.toString());

});

});

server.listen(8080);

console.log("启动web服务器,监听8080; <http://192.168.40.114:8080>");

### 4.读取数据库数据响应给浏览器客户端

var http = require("http");

var db = require("./db/db");

var server = http.createServer(function(request,response){

var sql = "SELECT \* FROM department";

db.query(sql, function (err, data) {

if (err) {

console.log("数据库出错" + err);

return;

}

// console.log(JSON.stringify(data));

response.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });

response.end(JSON.stringify(data));

});

});

server.listen(8089);

### Path模块使用

var path = require("path");

//解析当前目录:

var currenDir = path.resolve('.');

console.log(currenDir);

var dir = path.join(currenDir,"public","index.html"); //httpdemo\src\public\index.html

console.log(dir);

### 6.处理浏览器客户端请求,动态拼接访问资源路径

var http = require("http");

var fs = require("fs"); //文件模块

var url = require("url"); //url模块 url.parse(request.url);

var path = require("path"); //文件路径模块 path.resolve("."); path.join()

var util = require("util"); //工具模块 util.insepct(object) =>将对象参数传换为字符串输出

var server = http.createServer(function (request, response) {

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

console.log(util.inspect(reqUrl)); //util.insepct(object) =>将对象参数传换为字符串输出

console.log("request.reqUrl :"+reqUrl.pathname+ " query :"+reqUrl.query); ///public/js/my.js

var currentDir = path.resolve("."); //当前目录

var dir = path.join(currentDir,reqUrl.pathname); //路径拼接 e://hellodemo/public/js/my.js

console.log("dir :"+dir);

});

server.listen(8080);

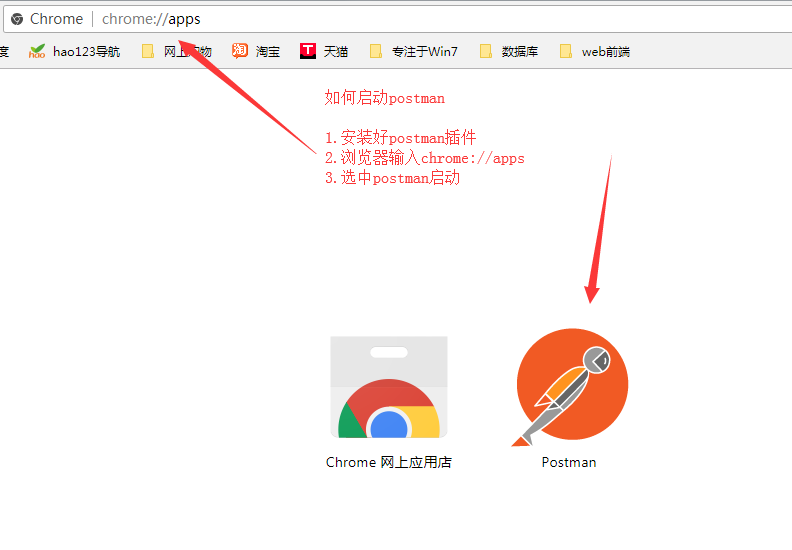
console.log("启动web服务器,监听8080; http://192.168.40.114:8080");

## 5、http请求响应模式深入理解

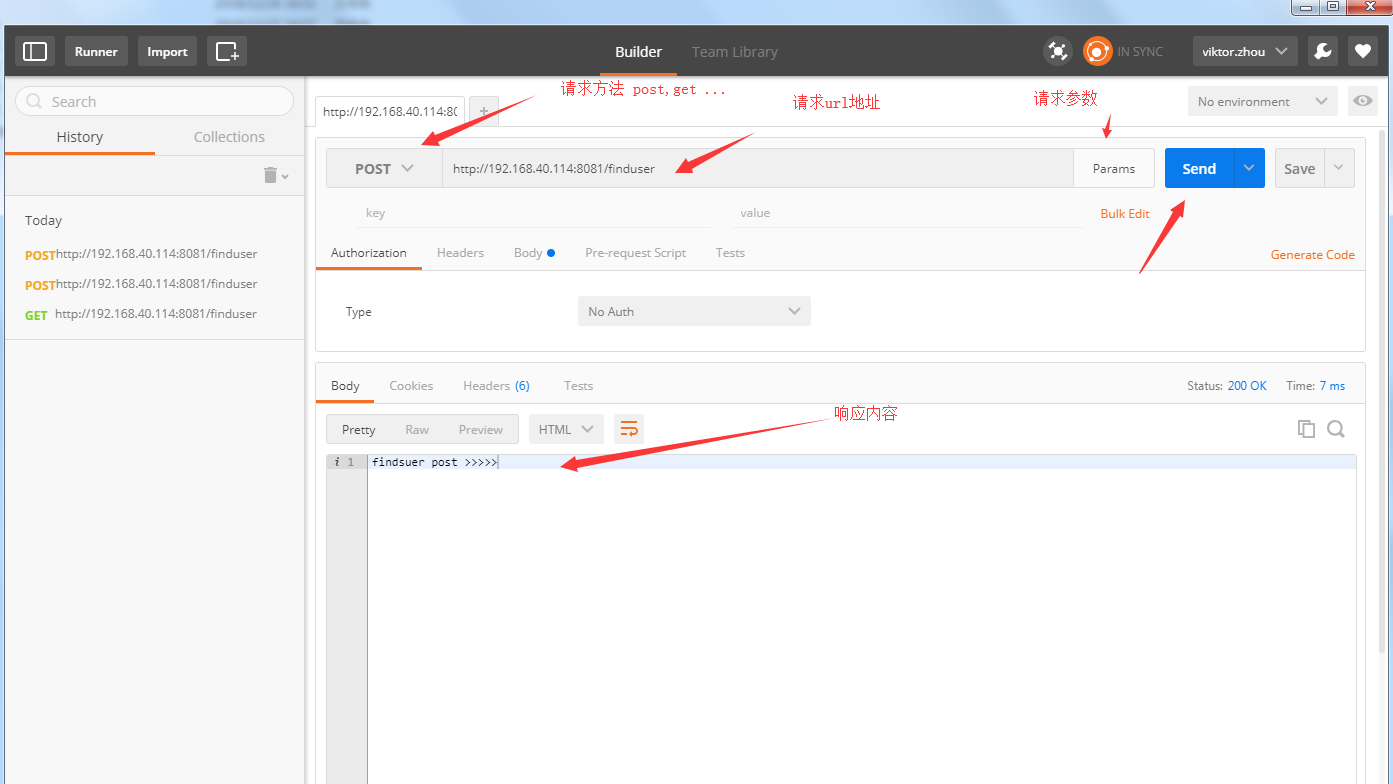
### 1. postman插件使用

1.安装方法参见jsonview插件

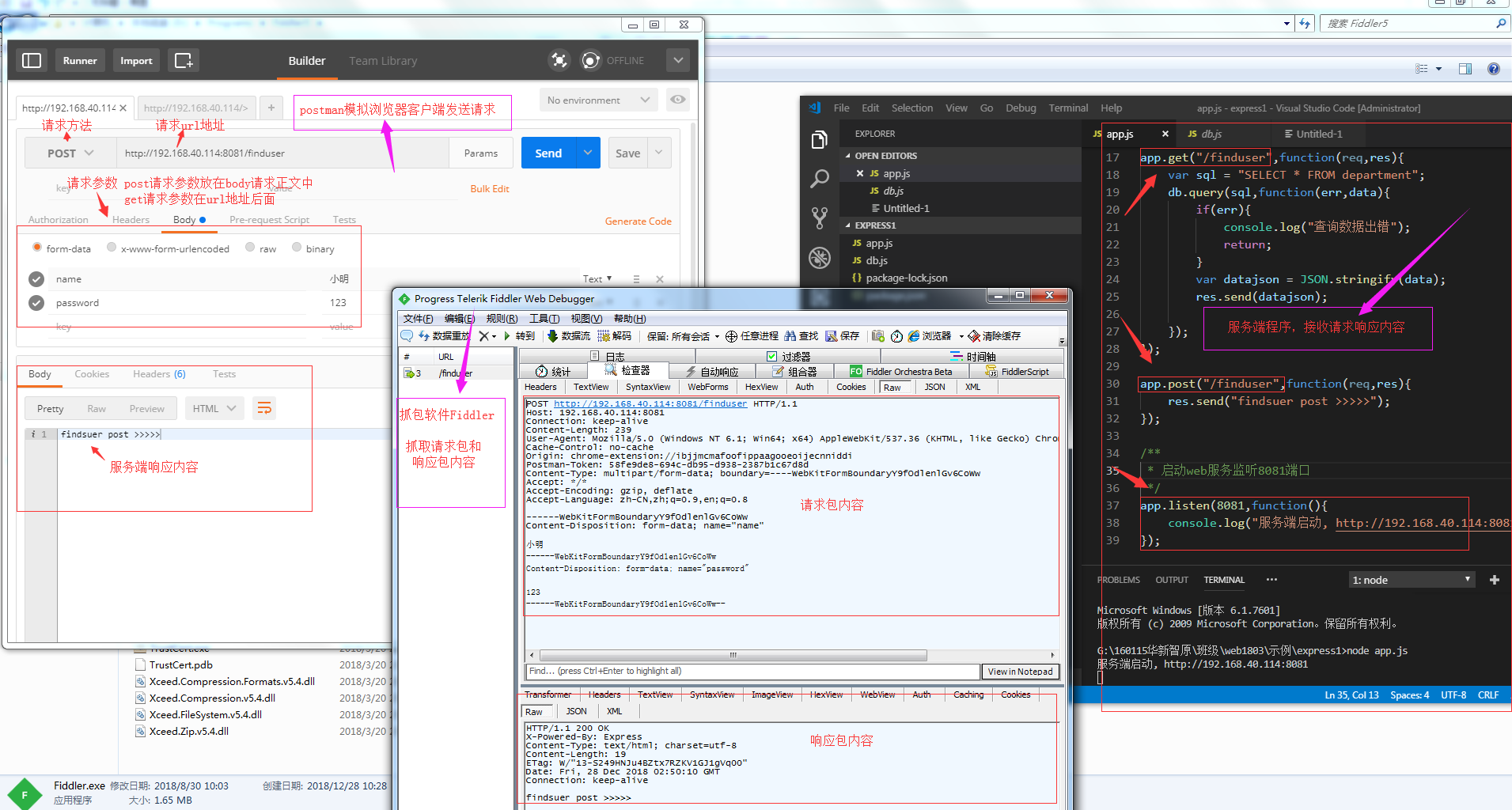
2.启动插件



3.使用

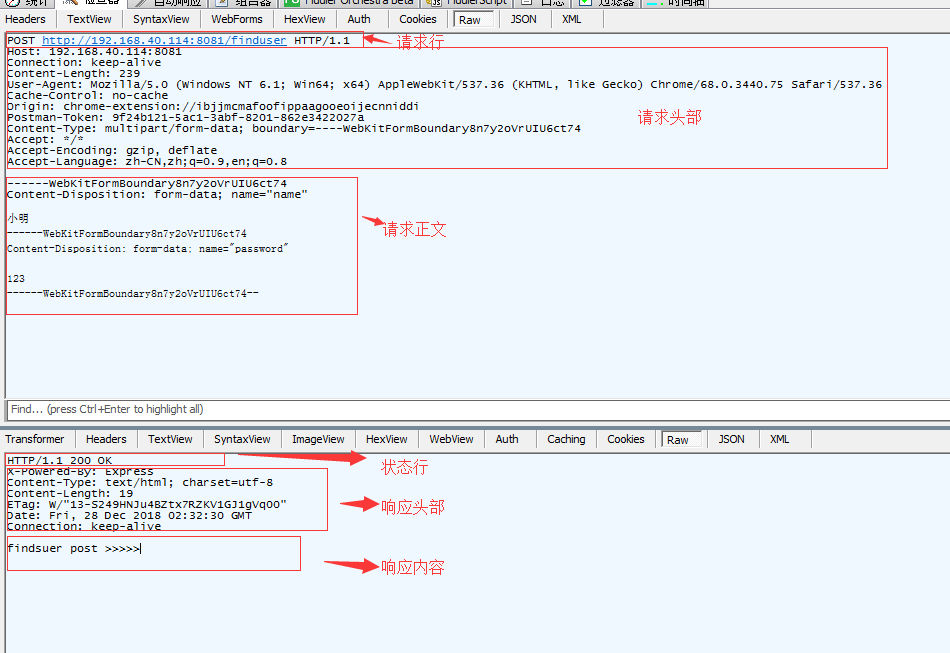
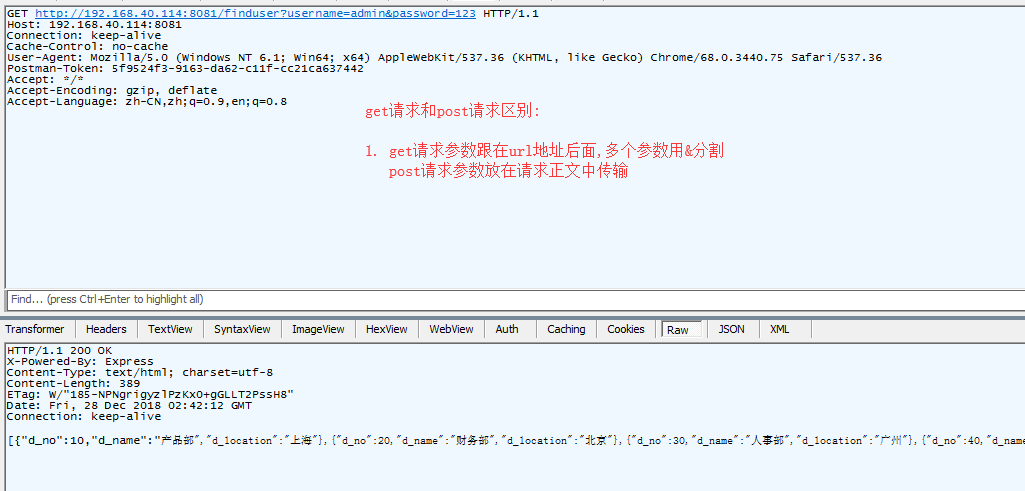


### 2.通过抓包工具理解http请求响应过程



### Get请求和post请求区别

1. 参数位置不同



## 6、Express框架

### 1、Express介绍

官网: <http://www.expressjs.com.cn/>



### 2、基础使用

#### 1安装

首先假定你已经安装了 [Node.js](https://nodejs.org/)，接下来为你的应用创建一个目录，然后进入此目录并将其作为当前工作目录。

$ mkdir myapp

$ cd myapp

通过 npm init 命令为你的应用创建一个 package.json 文件。 欲了解 package.json 是如何起作用的，请参考 [Specifics of npm’s package.json handling](https://docs.npmjs.com/files/package.json).

$ npm init

此命令将要求你输入几个参数，例如此应用的名称和版本。 你可以直接按“回车”键接受大部分默认设置即可，

接下来在 myapp 目录下安装 Express 并将其保存到依赖列表中。如下：

$ npm install express --save

#### 2启动服务响应Helloword

var express = require("express");

var db = require("./db");

var app = express();

/\*\*

\* http://192.168.40.114:8081/hello

\*/

app.get("/hello",function(req,res){

res.send("hello"); //响应内容

});

/\*\*

\* 启动web服务监听8081端口

\*/

app.listen(8081,function(){

console.log("服务端启动, http://192.168.40.114:8081");

});

### 3、Route路由定义格式

app.METHOD(PATH, HANDLER)

说明

* app  是express的一个实例instance
* METHOD  是 [HTTP request method](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol" \l "Request_methods), in lowercase. 如GET, POST…
* PATH  是服务器请求地址路径
* HANDLER  路由匹配时执行的函数.

### 4、路由分类

var express = require('express');

var app = express();

// 路由：字符串类型

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

res.send('book');

});

// 路由：字符串模式

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

res.send('user'); // 比如： /user/man, /user/woman

});

### 5、使用示例

#### Post请求处理

var express = require("express");

var db = require("./db");

var app = express();

//get请求方法,

//find请求url地址

app.get("/find",function(req,res){

res.send("find get >>>>>"); //响应内容

});

//post请求方法,

//find请求url地址

app.post("/find",function(req,res){

res.send("find post >>>>>"); //响应内容

});

/\*\*

\* 启动web服务监听8081端口

\*/

app.listen(8081,function(){

console.log("服务端启动, http://192.168.40.114:8081");

});

#### 响应数据库查询数据

db.js 模块

var mysql = require("mysql");

function query(sql,parmeters, callFunction) {

//创建数据库连接对象

var connection = mysql.createConnection(

{

host: "192.168.40.114",

user: "test",

password: "b123",

database: "ex1db"

}

);

connection.connect(); //打开数据库连接

connection.query(sql,parmeters,callFunction); //执行sql

connection.end(); //关闭数据连接

}

exports.query = query;

//module.exports = query;

app.js 模块

var express = require("express");

var db = require("./db");

var app = express();

app.get("/find",function(req,res){

var sql = "SELECT \* FROM department";

db.query(sql,function(err,data){

if(err){

console.log("查询数据出错");

return;

}

var datajson = JSON.stringify(data);

res.send(datajson);

});

});

/\*\*

\* 启动web服务监听8081端口

\*/

app.listen(8081,function(){

console.log("服务端启动, http://192.168.40.114:8081");

});

### 6、路由拆分

当你用的应用越来越复杂，不可避免的，路由规则也会越来越复杂。这个时候，对路由进行拆分是个不错的选择。

我们分别看下两段代码，路由拆分的好处就直观的体现出来了。

**路由拆分前**

var express = require('express');

var app = express();

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

res.send('/list');

});

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

res.send('/detail');

});

app.listen(3000);

这样的代码会带来什么问题呢？无论是新增还是修改路由，都要带着/user前缀，这对于代码的可维护性来说是大忌。这对小应用来说问题不大，但应用复杂度一上来就会是个噩梦。

**路由拆分后**

可以看到，通过express.Router()进行了路由拆分，新增、修改路由都变得极为便利。

var express = require('express');

var app = express();

var user = express.Router();

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

res.send('/list');

});

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

res.send('/detail');

});

app.use('/user', user); // mini app，通常做应用拆分

app.listen(3000);

### 7、body-parser express中间件使用

body-parser是非常常用的一个express中间件，作用是对http请求体进行解析。使用非常简单，以下两行代码已经覆盖了大部分的使用场景。

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: false })); //extended: false 表是只支持string|arrya类型，为true表示支持所有类型

#### 1.入门基础

在正式讲解前，我们先来看一个POST请求的报文，如下所示。

POST /test HTTP/1.1

Host: 127.0.0.1:3000

Content-Type: application/x-www-form-urlencoded; charset=utf8

Content-Encoding: gzip

name=helloword&password=123

其中需要我们注意的有Content-Type、Content-Encoding以及报文主体：

* Content-Type：请求报文主体的类型、编码。常见的类型有text/plain、application/json、application/x-www-form-urlencoded。常见的编码有utf8、gbk等。
* Content-Encoding：声明报文主体的压缩格式，常见的取值有gzip、deflate、identity。
* 报文主体：这里是个普通的文本字符串hello。

#### 2.body-parser主要做了什么

body-parser实现的要点如下：

1. 处理不同类型的请求体：比如text、json、urlencoded等，对应的报文主体的格式�不同。
2. 处理不同的编码：比如utf8、gbk等。
3. 处理不同的压缩类型：比如gzip、deflare等。
4. 其他边界、异常的处理。

#### 3.处理不同类型请求体

var express = require("express");

var bodyParser = require("body-parser");

var app = express(); //创建一个express对象实例

app.use(bodyParser.json()); //解析json类型文本 content-type:application/json

app.use(bodyParser.urlencoded({extended:false})); //解析 名称值对类型文本 content-type:application/x-www-form-urlencoded

app.post("/login.do",function(req,res){

var body = req.body; //请求正文

console.log(body);

res.send(body);

});

app.get("/login.do",function(req,res){

var query = req.query; //请求正文

console.log(query);

res.send(query);

});

app.listen(8080,function(){

console.log("启动服务器成功! http://192.168.40.114:8080");

})

#### 4.body-parser模块使用步骤

第一步: 安装body-parser模块  
 npm install body-parser  
 第二步: 获取模块，创建解析器  
 var bodyParser = require('body-parser') //获取模块  
 var jsonParser = bodyParser.json(); // 创建 application/json 解析 如： {名称1：值1,名称2：值2}  
 var urlencodedParser = bodyParser.urlencoded({ extended: false });// 创建 application/x-www-form-urlencoded 解析 如: 名称1=值1&名称2=值2  
 第三步: body-parser与exprerss web框架整合  
 app.use(jsonParser);  
 app.use(urlencodedParser);  
 第四步: 获取参数值  
 1. get请求  
 值1 = req.query.名称1  
 2. post请求  
 值1 = req.body.名称1

### 8、利用 Express 托管静态文件

为了提供诸如图像、CSS 文件和 JavaScript 文件之类的静态文件，请使用 Express 中的 express.static 内置中间件函数。

此函数特征如下：

express.static(root, [options])

The root argument specifies the root directory from which to serve static assets. For more information on the options argument, see [express.static](http://www.expressjs.com.cn/4x/api.html" \l "express.static).

例如，通过如下代码就可以将 public 目录下的图片、CSS 文件、JavaScript 文件对外开放访问了：

app.use(express.static('public'))

现在，你就可以访问 public 目录中的所有文件了：

http://localhost:3000/images/kitten.jpg

http://localhost:3000/css/style.css

http://localhost:3000/js/app.js

http://localhost:3000/images/bg.png

http://localhost:3000/hello.html

Express 在静态目录查找文件，因此，存放静态文件的目录名不会出现在 URL 中。

如果要使用多个静态资源目录，请多次调用 express.static 中间件函数：

app.use(express.static('public'))

app.use(express.static('files'))

访问静态资源文件时，express.static 中间件函数会根据目录的添加顺序查找所需的文件。

### 9、核心概念：中间件

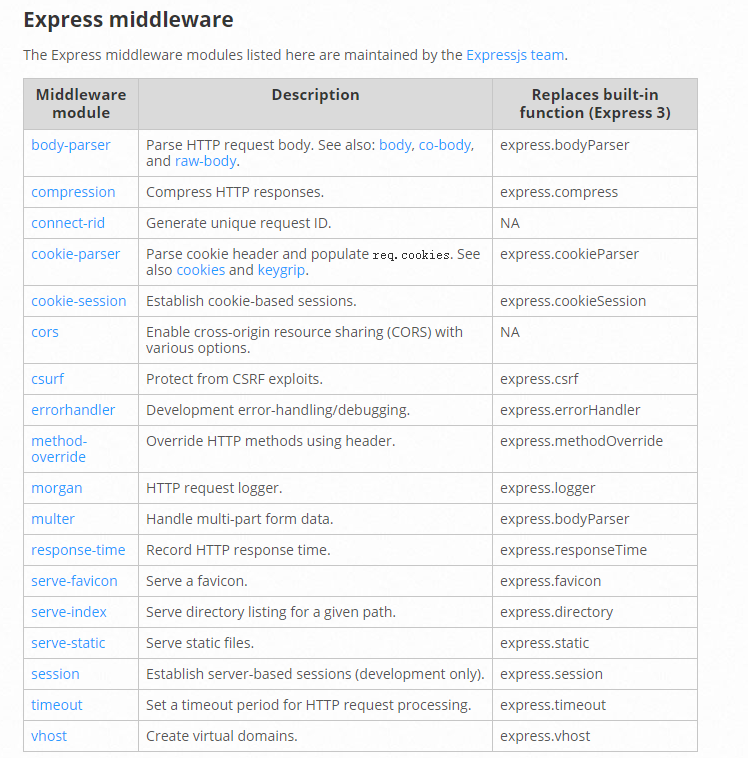
一般学习面向对象编程的时候，我们都会听到一句话：一切皆对象。而在学习express的过程中，很深的一个感受就是：一切皆中间件。比如常见的请求参数解析、cookie解析、gzip等，都可以通过中间件来完成。

#### 1.常用中间件

包括但不限于如下。body-parser

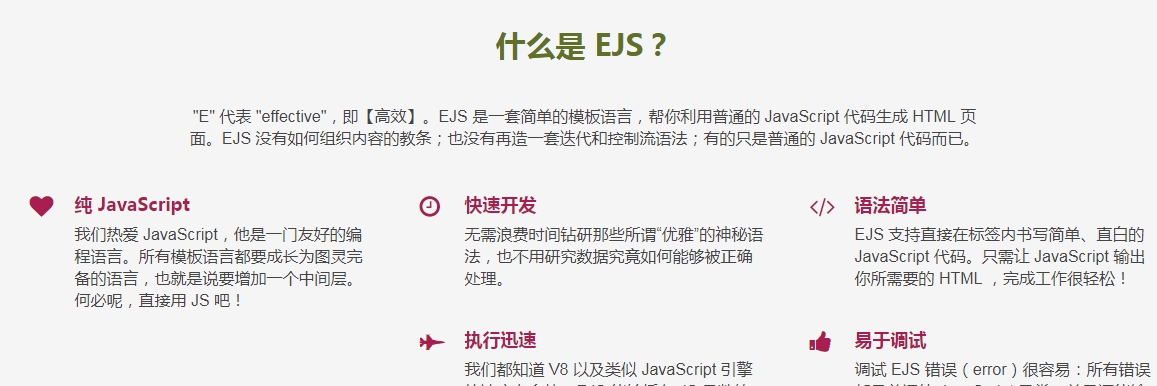
* compression
* serve-static
* session
* cookie-parser
* morgan

更多常用中间件，可以点击 [这里](http://expressjs.com/en/resources/middleware.html) http://expressjs.com/en/resources/middleware.html



### 10、ejs模板引擎

<https://ejs.bootcss.com/>



#### 1、什么是EJS

EJS是一个JavaScript模板库，用来从JSON数据中生成HTML字符串。

#### 2、为什么要使用EJS

与最初的JavaScript相比较，一些不太了解你的代码的人可以更容易地通过EJS模板代码看得懂你的代码。

示例:

最终要实现的效果如下:

用户列表

.张三

.李四

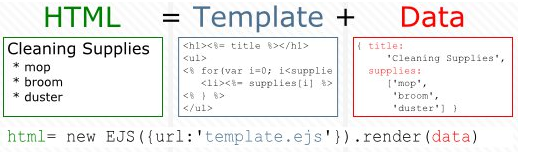
.王二

不使用模块情况

这是用javascript实现的代码

**var** express = require("express");  
**var** app = express();  
*/\*\*  
 用户列表  
 .张三  
 .李四  
 .王二  
 \*  
 \*/*app.get("/list",**function** (request,response) {  
 **var** userJson = '{"title":"用户列表一","list":[{"name":"张三"},{"name":"李四"},{"name":"王二"}]}';  
 **var** userObj = JSON.parse(userJson); //json字符串=》json对象  
  
 //var jsonStr = JSON.stringify(json对象) //json对角 =>json字符串  
 userObj.list  
 **var** html = '<h1>'+userObj.title+'</h1>' +  
 '<ul>' +  
 '<li>张三</li>' +  
 '<li>李四</li>' +  
 '<li>王二</li>' +  
 '</ul>';  
 response.send(html);  
});  
  
app.listen(8080,**function** () {  
 console.log("启动服务成功,监听8080端口");  
});

下面学习EJS同样实现上面的功效，它的工作原理如下：



使用EJS来找回你的明确、维护性良好的HTML代码结构。  
注:data是json对象，不能使json字符串。

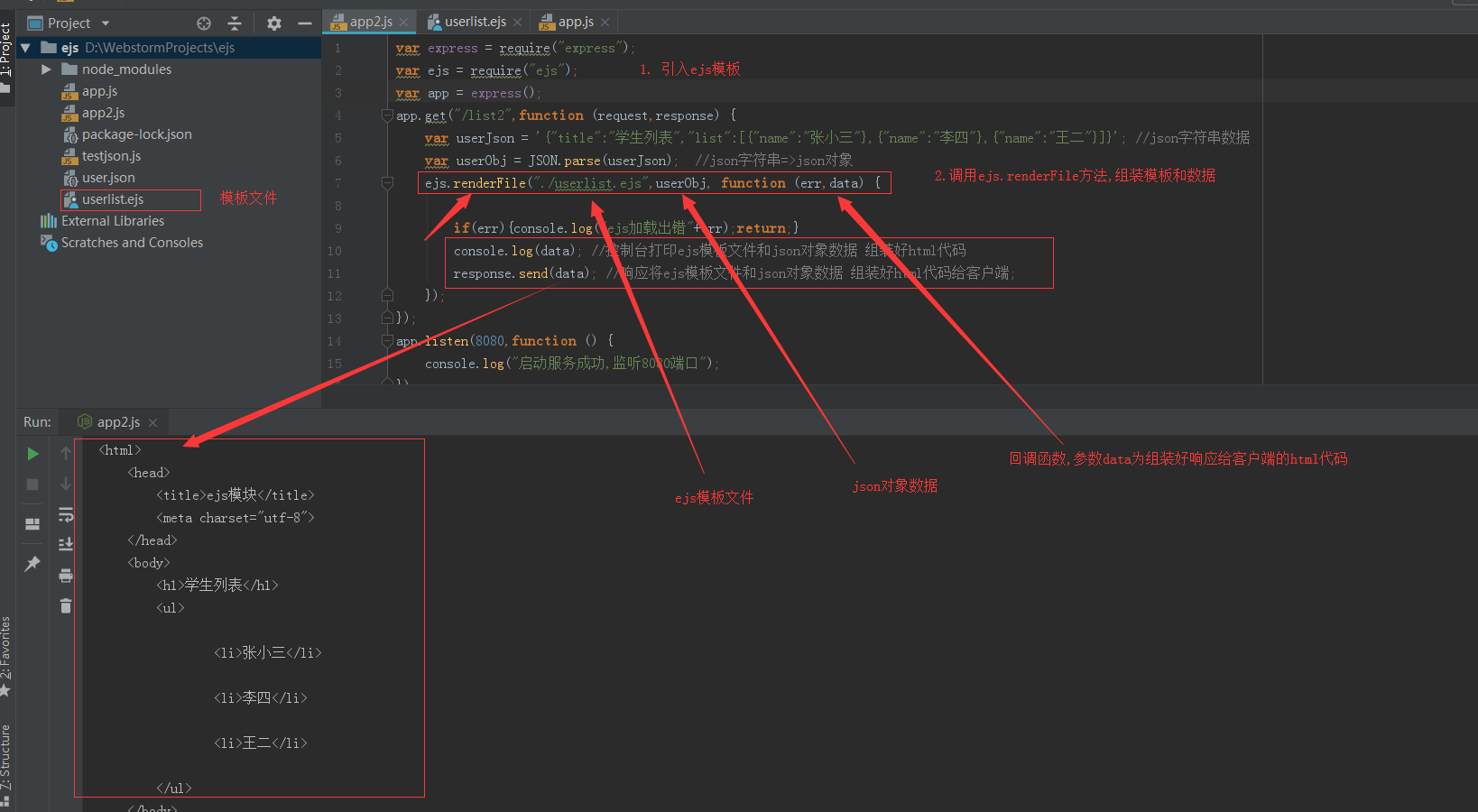
创建一个EJS模板,命名为userlist.ejs文件，内容如下：

<!---  
 %=变量 %  
 % js代码 %  
-->  
  
<html>  
<head>  
 <title>ejs模块</title>  
 <meta charset="utf-8">  
</head>  
<body>  
<h1><%= title %></h1>  
<ul>  
 <% **for**(**var** i = 0; i<list.length; i++){ %>  
 <li><%= list[i].name %></li>  
 <%}%>  
</ul>  
</body>  
</html>

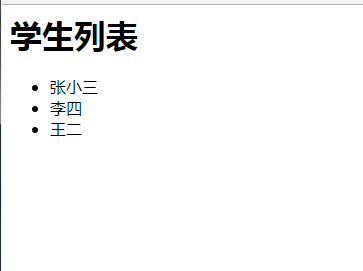
组装ejs模板和json数据响应给客户端

**var** express = require("express");  
**var** ejs = require("ejs");  
**var** app = express();  
app.get("/list",**function** (request,response) {  
 **var** userJson = '{"title":"学生列表","list":[{"name":"张小三"},{"name":"李四"},{"name":"王二"}]}'; //json字符串数据  
 **var** userObj = JSON.parse(userJson); //json字符串=>json对象

ejs.renderFile("./userlist.ejs",userObj, **function** (err,data) {  
  
 **if**(err){console.log("ejs加载出错"+err);**return**;}  
 console.log(data); //控制台打印ejs模板文件和json对象数据 组装好html代码  
 response.send(data); //响应将ejs模板文件和json对象数据 组装好html代码给客户端;  
 });  
});  
app.listen(8080,**function** () {  
 console.log("启动服务成功,监听8080端口");  
});



http://192.168.40.114:8080/list



#### 3、ejs模板语法

EJS是一个javascript模板库，用来从json数据中生成HTML字符串

* 功能：缓存功能，能够缓存HTML模板；
* <% code %>用来执行javascript代码

模板中输出变量

* 声明变量<% title="EJS Template engine" %>
* 输出变量<%= username %> 特殊字符将进行转义

<!-- 定义一个数组，遍历输出 -->

<% users=["Jack","Rose","Alice","Ave"] %>

<ul>

<% for(var item in users)

{ %>

<li> <%= users[item] %> </li>

<% }

%>

</ul>

#### 4、标签含义

* <% '脚本' 标签，用于流程控制，无输出。
* <%\_ 删除其前面的空格符
* <%= 输出数据到模板（输出是转义 HTML 标签）
* <%- 输出非转义的数据到模板
* <%# 注释标签，不执行、不输出内容
* <%% 输出字符串 '<%'
* %> 一般结束标签
* -%> 删除紧随其后的换行符
* \_%> 将结束标签后面的空格符删除

#### 5、包含（include）

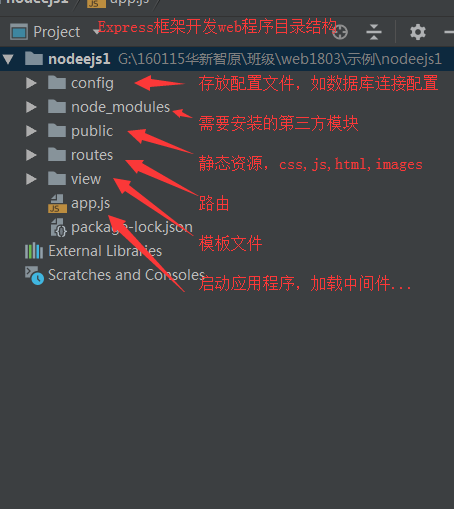
通过 include 指令将相对于模板路径中的模板片段包含进来。(需要提供 'filename' 参数。) 例如，如果存在 "./views/users.ejs" 和 "./views/user/show.ejs" 两个模板文件，你可以通过 <%- include('user/show'); %> 代码包含后者。

你可能需要能够输出原始内容的标签 (<%-) 用于 include 指令，避免对输出的 HTML 代码做转义处理。

<ul>  
 <% users.forEach(**function**(user){ %>  
 <%- include('user/show', {user: user}); %>  
 <% }); %>  
</ul>

### 11、Express框架开发web应用程序示例

#### 项目目录结构



#### 默认模板与路径更改

//设置默认模板引擎是html  
app.engine('.html', ejs.\_\_express);  
app.set('view engine', 'html');  
app.set("views","view"); //模板所在的目录为view

#### 静态资源所有目录

app.use(express.static('public')); //静态资源所有目录

#### Response对象方法

**（1）**response.render方法用于渲染网页模板。

app.get("/", function(request, response) {

response.render("index", { message: "Hello World" });});

上面代码使用render方法，将message变量传入index模板，渲染成HTML网页。

response.render('user/user',{"title":"学生列表","list": [ { "name":"张三"},{"name":"李四"}]});

上述代码等价于下面代码ejs.renderFile， 将指定模板文件与json数据组装成html代码，发送给客户端

ejs.renderFile("./view/user/user.html", {"title":"学生列表","list": [ { "name":"张三"},{"name":"李四"}]} , **function** (err,data) {  
 **if**(err){  
 console.log("ejs模板加载出错"+err);  
 **return**;  
 }  
 response.send(data);  
});

**（2）response.redirect方法**

response.redirect方法允许网址的重定向。可以重定向到静态文件

response.redirect("/hello/anime");

response.redirect("http://www.example.com");

response.redirect(301, "http://www.example.com");

**（3）response.sendFile方法**

response.sendFile方法用于发送文件。

response.sendFile("/path/to/anime.mp4");

#### 参数解析-body-parser模块使用

app.use(bodyParaser.json()); //解析json类型数据  
app.use(bodyParaser.urlencoded({extended:**false**})); //expanded:false表示解析值类型是string|Array, true表示解析值是任意类型

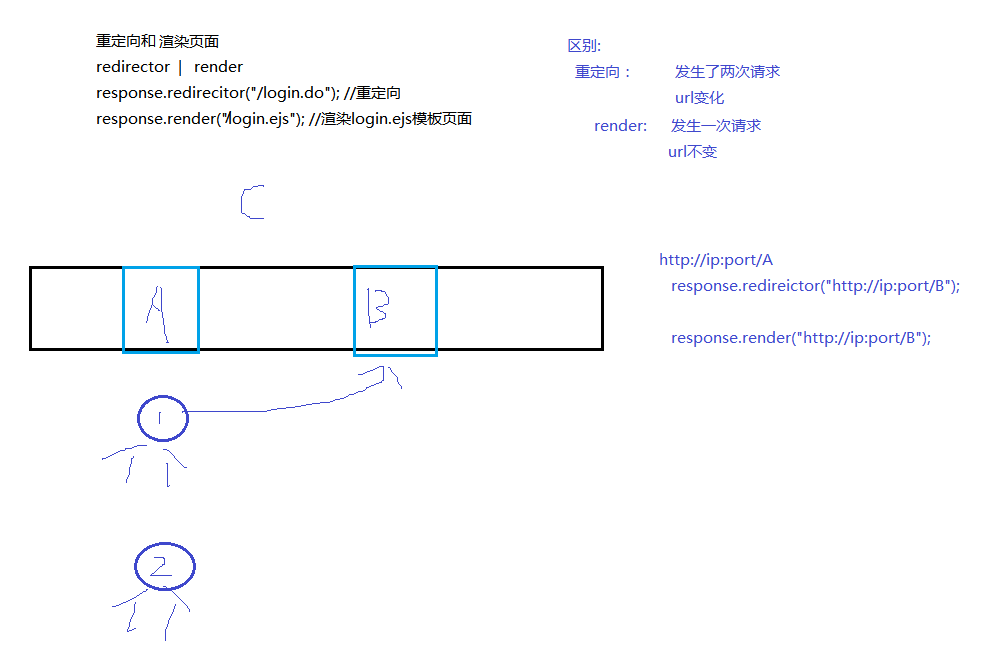
接收Post请求参数

**var** userName = req.body.name;  
**var** passWord = req.body.password;

接收Get请求参数

**var** userName = req.query.name;  
**var** passWord = req.query.password;

#### 重定向与页面渲染



#### 错误

问题1

下述代码会报错

app.use(indexRouter);  
app.use(bodyParser.urlencoded({extended:**false**})); //expanded:false表示解析值类型是string|Array, true表示解析值是任意类型

改为交换两行代码位置后正常

//必须放到路由前面  
app.use(bodyParser.urlencoded({extended:**false**})); //expanded:false表示解析值类型是string|Array, true表示解析值是任意类型  
app.use(indexRouter);

### 12、pug模板引擎

在node常用的模板引擎一般是   
1、jade ——破坏式的、侵入式、强依赖（对原有的html体系不友好，走自己的一套体系）  
2、ejs ——温和的、非侵入式的、弱依赖

[Pug](https://pugjs.org/api/getting-started.html" \t "_blank)原名不叫Pug，是大名鼎鼎的jade，后来由于商标的原因，改为Pug，哈巴狗。其实只是换个名字，语法都与jade一样。丑话说在前面，Pug有它本身的缺点——可移植性差，调试困难，性能并不出色，但使用它可以加快开发效率。本文将详细介绍pug模板引擎

#### 1.pug中文文档参考

https://pug.bootcss.com/api/getting-started.html



#### 2.入门示例

##### 1.在原始目录下建立一个views目录，来存放该 test.pug

html  
 head  
 script  
 style  
 body

##### 2.把该代码输出成一个html文件,我们可以在源目录下新建一个**build目录**来存放生成的文件 那就将刚才的**app.js**写成

**Var** jade = require('pug');//加载jade引擎  
**var** fs = require('fs')  
  
**var** str = jade.renderFile('./views/test.pug' ,{pretty : **true** }); //pretty : ture 相当于beauty格式化一下输出的代码  
fs.writeFile('./build/test.html' ,str , **function**(err){  
 **if** (err)  
 console.log("编译失败");  
 **else** console.log("编译成功");  
})

##### 3.生成的html页面

<html>  
 <head>  
 <script></script>  
 <style></style>  
 </head>  
 <body></body>  
</html>

##### **4.关于class/style的写法——属性放在（）里面，用逗号分隔**

html  
 head  
 script  
 style  
 body  
 div(class=['aaa','bbb','ccc'])  
 //class也可以写成div(style="aaa bbb ccc")  
 div(style={width:'200px' ,height:'300px' ,background:'red'})  
 //style也可以写成div(style="width:200px;xxxx")

运行一下node.js，则输出结果为

<html>  
 <head>  
 <script></script>  
 <style></style>  
 </head>  
 <body>  
 <div class="aaa bbb ccc"></div>  
 <!--class也可以写成div(style="aaa bbb ccc")-->  
 <div style="width:200px;height:300px;background:red;"></div>  
 <!--style也可以写成div(style="width:200px;xxxx")-->  
 </body>  
</html>

关于上方输出格式，可以发现，**style是可以用json传输的**，**class是可以采用数组传输进去的**。  
因此可以在node.js中直接插入相关属性数据，然后在jade文件调用，这样就可以很方便的生成不同框架的模板文件

如果你**想插入相关属性数据**，并**调用**的话，应当在node.js中的 **jade.renderFile中如此添加数据**

var pug = require('pug');

var fs = require('fs');

var str = pug.renderFile('./test.pug', {

pretty: true,

divsytle1: {

width: '200px',

height: '300px',

background: 'blue'

},

classarrays:[

'classname1',

'classname2'

]

});

console.log(str);

fs.writeFile('./build/test.html', str, function (err) {

if (err)

console.log("编译失败"+err);

else

console.log("编译成功");

});

**并在test.jade文件中修改如下：**

html  
 head  
 script  
 style  
 body  
 div(class=arr)  
 div(style=cls)

运行一下，结果是跟刚才的相同

##### **5、在pug标签中输入数据时，记得在相应标签后，加一个空格**

我们通常前端编程的时候，一般都会写到

<div>名称：DobTink  
 <div>年龄：15</div>  
 <script src="a.js"></script>  
 <script>  
 window.onload = **function** () {  
 console.log('测试输出');}  
 </script>  
</div><a href="http://www.dobtink.com">首页</a>

而在jade中，我们需要这样来写这实际上是最常见的情况，文本只需要和标签名隔开一个空格即可,

有时可能想要写一个大段文本块。比如**嵌入脚本或者样式。只需在标签后面接一个 .即可**

div 名称：DobTink  
 div 年龄：15  
 script(src='a.js')  
 script. //注意在script后面加个点"."  
 window.onload = **function** () {  
 console.log('测试输出');}  
a(href="http://www.dobtink.com") 首页

##### **6、在pug中使用if else switch while 等语句**

**pug中的 if 使用**

Pug 的条件判断的一般形式的括号是可选的，所以可以省略掉开头的 -，效果完全相同。类似一个常规的 JavaScript 语法形式

html  
 head  
 body  
 -**var** a = 15;  
 **if** a%2 == 0  
 div(style={background:'red'})  
 **else** div(style={background:'green'})

**pug中的 switch 使用**

html  
 head  
 body  
 -**var** a = 3;  
 **case** a  
 **when** 0  
 div aaa  
 **when** 1  
 div bbb  
 **when** 2  
 div ccc  
 **when** 3  
 div ddd  
 **default** div default

##### 7.循环

　　Pug 目前支持两种主要的迭代方式： each 和 while

ul  
 each val **in** [1, 2, 3, 4, 5]  
 li= val

<ul>  
 <li>1</li>  
 <li>2</li>  
 <li>3</li>  
 <li>4</li>  
 <li>5</li>  
</ul>

##### 代码1

html

head

title pug学习

meta(charset='utf-8')

script.

var message = 'hello pug';

window.onload=function(){

// alert(message);

}

body

div(id='contaner' class='style1')

p(id='p1' style={color:'red',fontsize:'20px'}) 用户名:

input(name='username' type='text' placeholder='请输入用户名' value=username)

br

p 密码:

input(name='password' type="password" placeholder='请输入密码' value=password)

-var k = 10;

if(k % 2 === 0)

p 登录成功

else

p 用户名密码出错

div

ul

each item in week

li= item

##### Pug使用css样式

//-引入内部样式表

style.

#container{

background-color: gray;

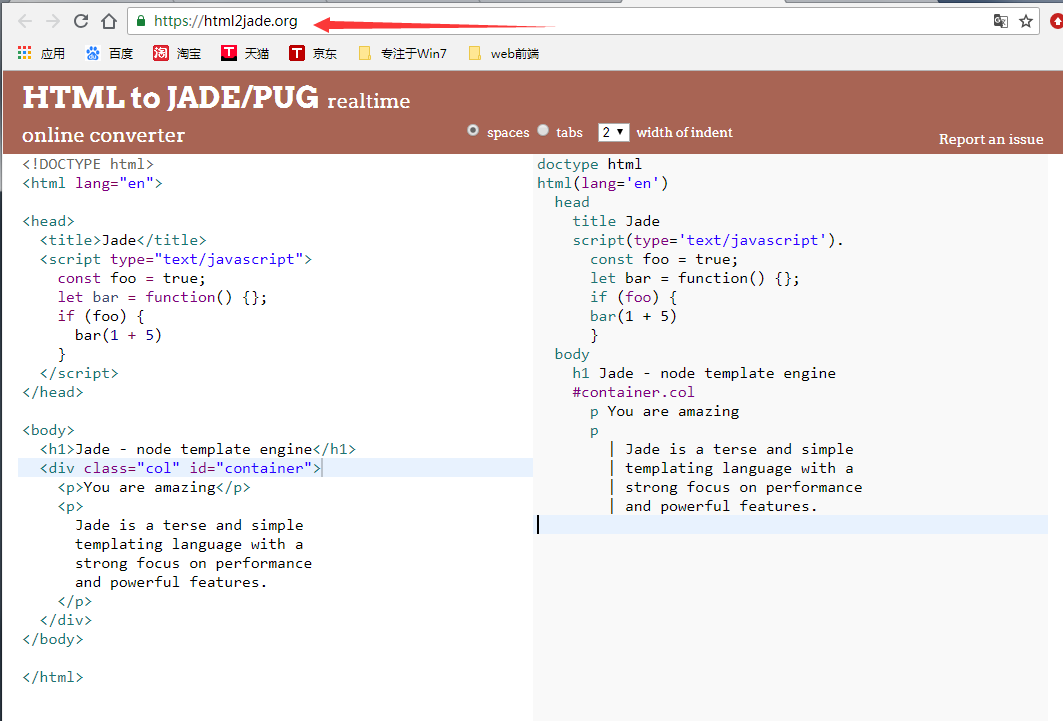
}

//-引入外部样式表

link(rel="stylesheet", href="css/my.css")

#### 3.Html代码转pug代码

<https://html2jade.org/>



#### 4.继承

Pug 支持使用 block 和 extends 关键字进行模板的继承。一个称之为“块”（block）的代码块，可以被子模板覆盖、替换

示例: 定义父模板 layout.pug

//- layout.pug

html

head

title 我的站点 - #{title}

block scripts

script(src='/jquery.js')

body

block content

block foot

#footer

p 一些页脚的内容

子模板 page-a.pug

//- page-a.pug

extends layout.pug

block scripts

script(src='/jquery.js')

script(src='/my.js')

block content

div 正文内容

p 子内容

#### 包含 include

//- includes/head.pug

head

title 我的网站

meta(charset='utf-8')

script.

var message = 'hello pug';

window.onload=function(){

// alert(message);

}

html

//- head

//- title pug学习

//- meta(charset='utf-8')

//- script.

//- var message = 'hello pug';

//- window.onload=function(){

//- // alert(message);

//- }

include head.pug

body

div(id='contaner' class='style1')

#### 注释

//单行注释，会显示在编译后的html代码中 <!--单行注释，会显示在编译后的html代码中 -->

div 单行注释 <div>单行注释</div>

//- 单行注释，不会显示在编译后的html代码中

div 单行注释 <div>单行注释</div>

//-

多单注释

不显示在html代码中

#### Express框架集成pug模板

//设置默认模板引擎是pug

app.engine('.pug', pug.\_\_express);

app.set('view engine', 'pug');

### Express错误处理

#### Express 内置的异步错误处理

<http://expressjs.com/zh-cn/guide/error-handling.html>



示例:

使用Express 框架use方法添加错误处理句柄error handler

// error handler; 处理所有错误

app.use(function (err, req, res, next) {

console.log('err :' + err + ' err.status :' + err.status);

res.status(err.status || 500); // 响应状态码如果没有响应500

// 判断状态码显示404错误页面

if (err.status === 404) {

res.render('error/error404');

} else {

res.render('error', {

message: err.message,

error: {

status: err.status,

stack: err.stack,

},

});

}

创建404资源找不到错误,通过next方法交给error handler处理

var createError = require('http-errors');

// catch 404 and forward to error handler

app.use(function (req, res, next) {

next(createError(404)); // 创建404资源找不到错误,next方法交给error handler处理

});

同步异常测试 /test1

userRouter.get('/test1', function (req, res) {

throw new Error('请求出错');

});

上述代处理同步异常没问题但在处理异步异常时，error handler不能捕获异步异常

异步异常处理方式一:使用next函数，将错误信息交给error handler

例如:

userRouter.get('/find', function (req, res, next) {

var sql = 'SELECT id,username,password FROM user WHERE ids = ?';

var parameters = [12];

db.query(sql, parameters, function (err, data) {

if (err) {

//throw new Error('请求出错'); //异步异常，throw子句无法处理

return next(err); // 交异常交给错误处理句柄 error handler

}

res.send(data[0]);

});

});

#### 利用 Promise 传递异步错误

Promise机制

<https://www.cnblogs.com/superAnny/p/7622862.html>

#### 用 Generators 简化代码

### 热部署加载，自动监听并同步的的 node工具——supervisor

使用express开发时，每次修改文件后，都需要使用npm start命令重启服务，然后才能查看输出结果（在代码中使用**console.log()**），感觉非常的麻烦。有没有一种方法可以使修改后的文件立马生效呢，答案是肯定的。这篇就介绍一个 自动监听并同步的的 node工具——supervisor。

全局安装 supervisor

直接用npm安装既可，输入指令 :

$ npm -g install supervisor

这里注意一点的就是，supervisor必须安装到全局，如果你不安装到全局，错误命令会提示你安装到全局。

用supervisor启动express项目

安装完以后就可以 用 supervisor 来启动 express项目了，**进入项目根目录**，执行：

supervisor ./bin/www

这样你的项目一旦发生更改（只要不是语法错误），supervisor会帮你自动同步到该进程里来，不需要重新启动。

如下图所示，当你修改一个项目文档并保存时，node服务会重新启动。

# 工具软件

## Eslint - js代码检查工具

<https://cn.eslint.org/>



使用步骤:

1. 安装:

npm install eslint -g

1. 编写测试text.js文件

var name = "张三";

if (name === '小明') {

console.log('hello小明');

} else {

console.log('出错');

}

1. 生成.eslintrc.js配置文件

eslint --init

1. 执行编译命令:

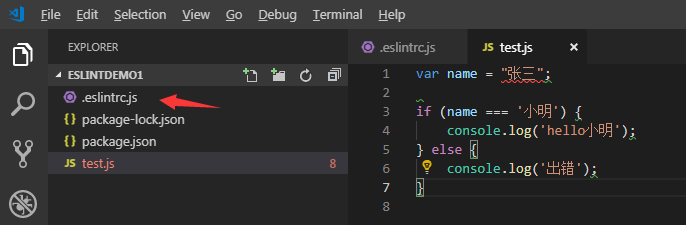
eslint test.js

注:如果eslint命令找不到，需设置环境变量, path： D:\nodejs\node\_global(eslint命令所在目录)

1. 与vscode集成

1>Vscode安装 ESLint插件

2>编写 .eslintrc.js 配置文件 注.eslintrc.js配置文件可通过命令生成: eslint --init

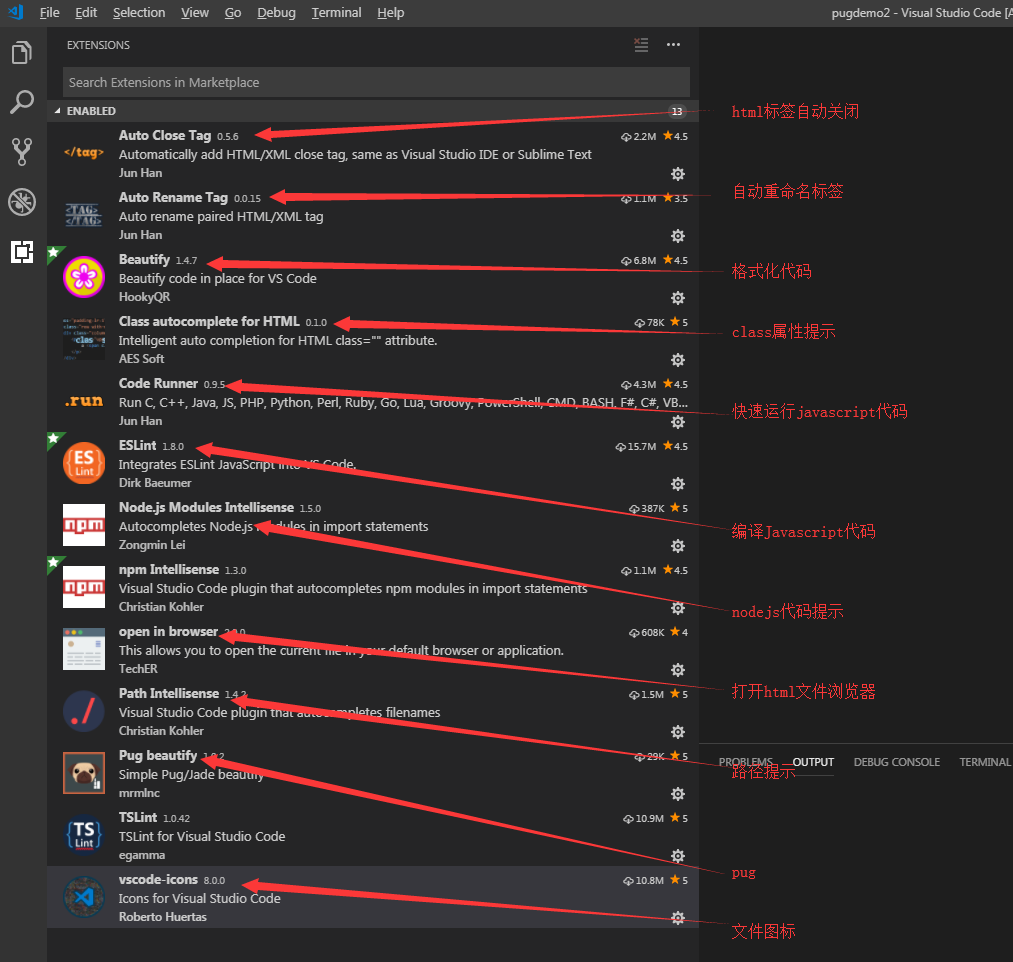


1. 与webstome集成



## vscode常用插件

<https://segmentfault.com/a/1190000011779959> 参考



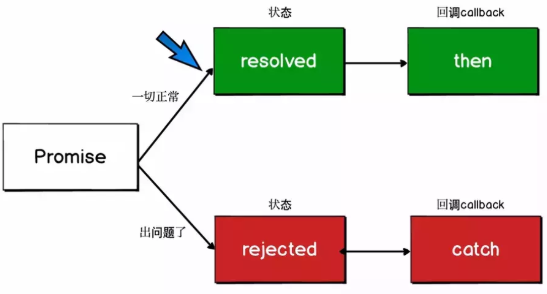
# js异步编程

## 1、Promise

Promise本意是承诺，在程序中的意思就是承诺我过一段时间后会给你一个结果。 什么时候会用到过一段时间？答案是异步操作，异步是指可能比较长时间才有结果的才做，例如网络请求、读取本地文件等

1.Promise的三种状态

* Pending----Promise对象实例创建时候的初始状态
* Fulfilled----可以理解为成功的状态
* Rejected----可以理解为失败的状态



### Promise入门

一、为什么要学习promise?

Javascript是一种单线程的语言， 所有的代码必须按照所谓的“ 自上而下” 的顺序来执行。 本特性带来的问题， 在处理耗时操作时， 程序会阻塞， 这时需要使用异步编程；

异步编程的一种实现方式就是回调函数;

回调函数是 一个 解决方案， 但它并不完美。 两个很大的问题是：

1. 颠倒的控制；

2. 糟糕的错误处理.

3. 异步任务回调嵌套.

fs.readFile(filename, function (err, data) {

if (err) {

console.log('读文件出错' + err);

} else {

console.log(data.toString());

fs.readFile(filename, function (err, data) {

if (err) {

console.log('读文件出错' + err);

} else {

console.log(data.toString());

fs.readFile(filename, function (err, data) {

if (err) {

console.log('读文件出错' + err);

} else {

console.log(data.toString());

}

});

}

});

}

});

二、什么是promise？

相当是异步操作还未兑现的一个承若; promise对象

两种状态:

成功兑现 resolve

失败 rejected

程序创建promise

var promise = new Promise(function(resolve,rejcted){

//具体异步

db.query(sql,parameter,function(err,data){

if(err){

rejected(err);

} else{

resove(data);

}

});

});

promise.then(function(data){

console.log(data); //成功兑现

}).catch(function(err){

//处理错误

});

示例

//创建一个promise对象，处理异步读文件操作

var promise = new Promise(function (resolve, rejected) {

//处理异步操作 异步买书，异步读文件

fs.readFile(filename, function (err, data) {

if (err) {

rejected(err);

} else {

resolve(data);

}

});

});

//then catch

promise.then(function (data) {

console.log(data.toString());

}).catch(function (err) {

console.log('出错' + err);

});

### 链式操作的用法

var fs = require('fs');

var filename = 'test.txt';

/\*\*

\* 读文件

\*/

function readMyFile(filePath) {

//创建一个promise对象，处理异步读文件操作

return new Promise(function (resolve, rejected) {

//处理异步操作 异步买书，异步读文件

fs.readFile(filePath, function (err, data) {

if (err) {

rejected(err);

} else {

resolve(data);

}

});

});

};

//then catch

readMyFile('test.txt')

.then(function (data) {

console.log('文件1内容');

console.log(data.toString());

return readMyFile('test1.txt');

})

.then(function (data2) {

console.log('文件2内容');

console.log(data2.toString());

return readMyFile('test2.txt');

})

.then(function (data3) {

console.log('文件3内容');

console.log(data3.toString());

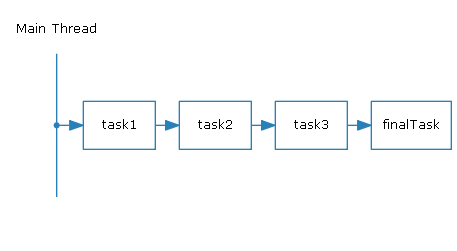
})

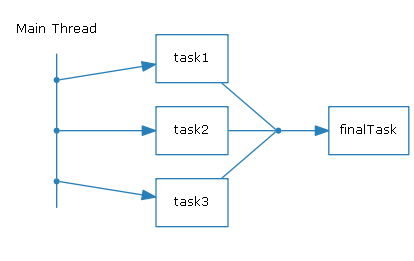
.catch(function (err) {

console.log('出错' + err);

});

### 并发执行异步操作





var fs = require('fs');

var filename = 'test.txt';

/\*\*

\* 读文件

\* 创建异步读文件未兑现的成若-promise对象

\*/

function readMyFile(filePath) {

//创建一个promise对象，处理异步读文件操作

return new Promise(function (resolve, rejected) {

//处理异步操作 异步买书，异步读文件

fs.readFile(filePath, function (err, data) {

console.log('读文件 '+filePath);

if (err) {

rejected(err);

} else {

resolve(data);

}

});

});

};

/\*\*

\* 并发执行任务

\*

\* 所有任务执行完后一起处理

\*/

Promise.all([readMyFile('test.txt'), readMyFile('test1.txt'), readMyFile('test2.txt')])

.then(function(results){

console.log('处理结果');

console.log(results.toString());

});

/\*\*

\* then函数处理最快执行完的异步执行

\*/

Promise.race([readMyFile('test.txt'), readMyFile('test1.txt'), readMyFile('test2.txt')])

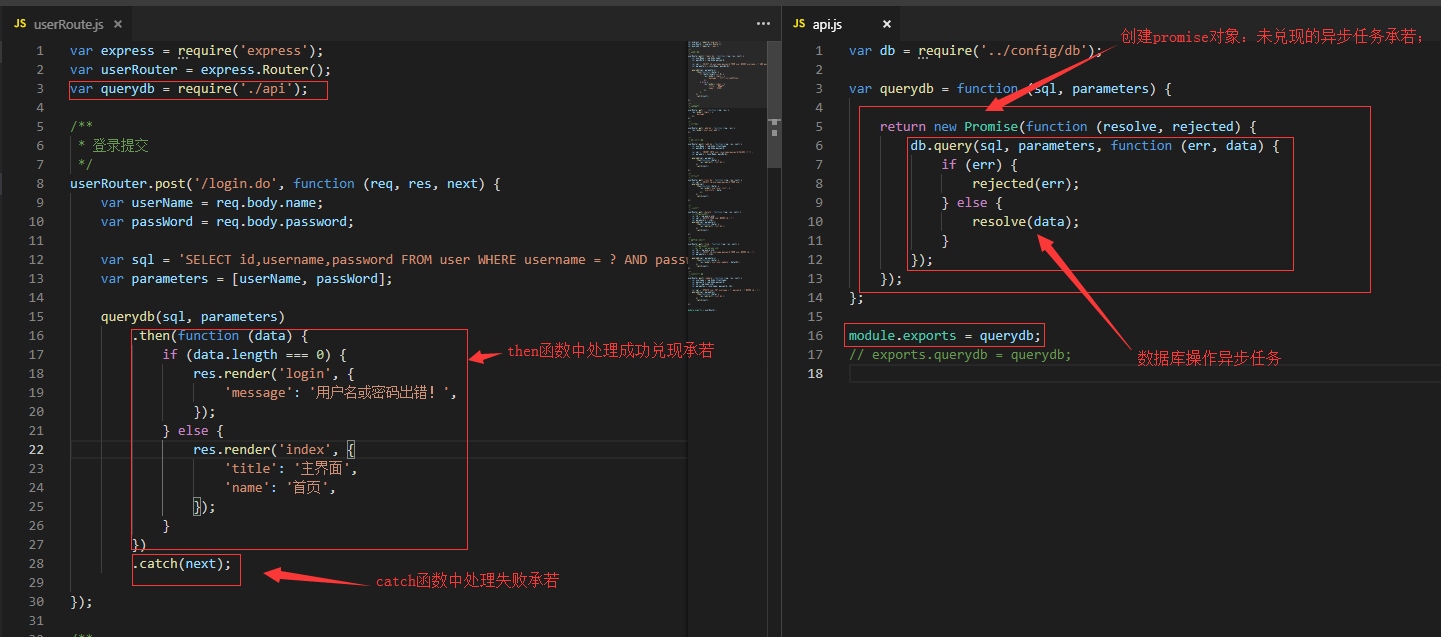
.then(function (results) {

console.log('处理结果');

console.log(results.toString());

});

### Promise重构用户管理系统数据持久化操作代码(即数据库操作)



## async/await

### 1.Async/Await简介

使用async/await，你可以轻松地达成之前promise做到的工作,它有如下特点：

* async/await是基于Promise实现的，它不用于普通的回调函数。
* async/await与Promise一样，是非阻塞的。
* async/await使得异步代码看起来像同步代码，这正是它的魔力所在。

**一个函数如果加上 async ，那么该函数就会返回一个 Promise**

async function async1() {

return "1"

}

console.log(async1()) // -> Promise {<resolved>: "1"}

async1().then(function(data){

Console.log(data); // -> 1

});

**依次调用三个文件那个例子用async/await写法，只需几句话便可实现**

var fs = require('fs');

function read(file) {

return new Promise(function(resolve, reject) {

fs.readFile(file, 'utf8', function(err, data) {

if (err) reject(err)

resolve(data)

})

})

}

async function readResult(params) {

try {

var p1 = await read(params, 'utf8')//await后面跟的是一个Promise实例

var p2 = await read(p1, 'utf8')

var p3 = await read(p2, 'utf8')

console.log('p1', p1)

console.log('p2', p2)

console.log('p3', p3)

return p3

} catch (error) {

console.log(error)

}

}

readResult('1.txt').then( // async函数返回的也是个promise

data => {

console.log(data)

},

err => console.log(err)

)

### 2.Async/Await并发请求

如果请求两个文件，毫无关系，可以通过并发请求

var fs = require('fs');

function read(file) {

return new Promise(function(resolve, reject) {

fs.readFile(file, 'utf8', function(err, data) {

if (err) reject(err)

resolve(data)

})

})

}

function readAll() {

read1()

read2()//这个函数同步执行

}

async function read1() {

let r = await read('1.txt','utf8')

console.log(r)

}

async function read2() {

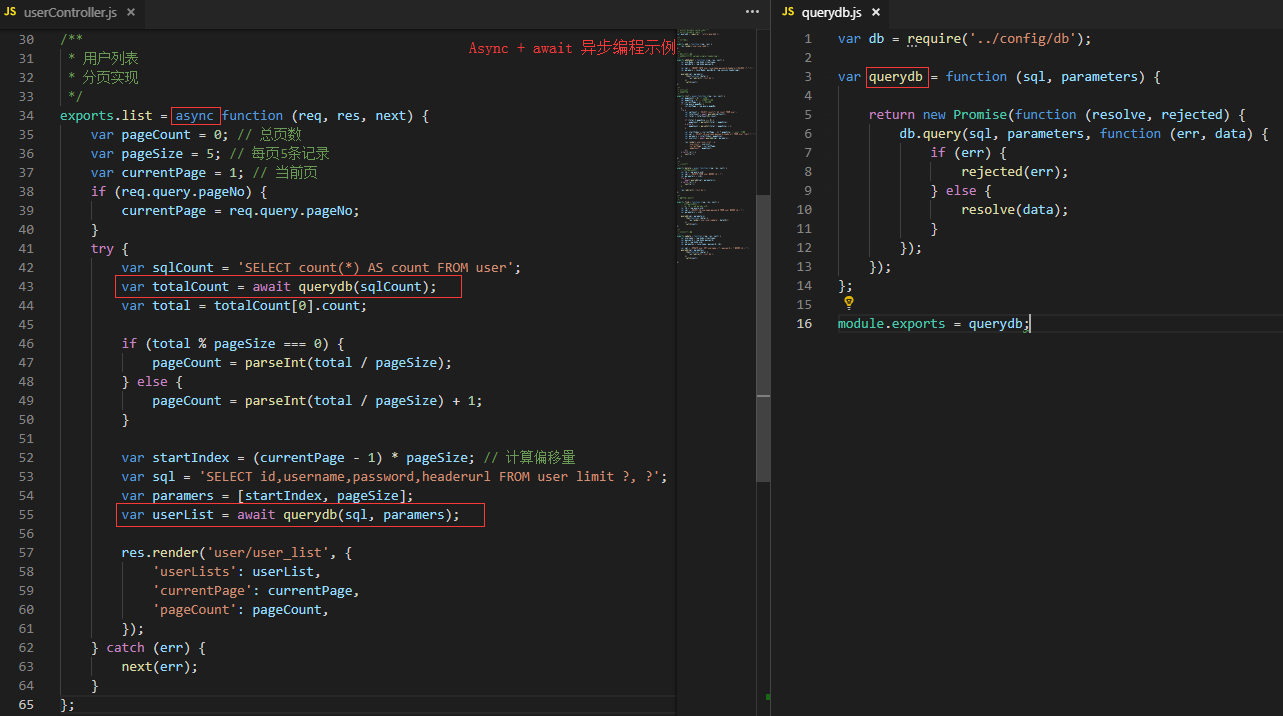
let r = await read('2.txt','utf8')

console.log(r)

}

readAll();

### Async/await重构用户管理系统数据持久化操作代码(即数据库操作)



## 3、总结

**1.JS 异步编程进化史：callback -> promise -> generator -> async + await**

**2.async/await 函数的实现，就是将 Generator 函数和自动执行器，包装在一个函数里。**

**3.async/await可以说是异步终极解决方案了。**

**(1) async/await函数相对于Promise，优势体现在**：

* 处理 then 的调用链，能够更清晰准确的写出代码
* 并且也能优雅地解决回调地狱问题。

当然async/await函数也存在一些缺点，因为 await 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 await 会导致性能上的降低，代码没有依赖性的话，完全可以使用 Promise.all 的方式。

# Cookie、session和拦截器-实现登录认证

## 1、cookie技术

### 1. 什么是cookie

说起来，Cookie应该是一种应用较久的技术了。早在HTML刚刚出现的时候，在每个独立的页面之间没有办法记录和标识不同的用户。后来人们就发明了 Cookie技术，当用户访问网页时，它能够在访问者的机器上创立一个文件，我们把它叫作Cookie，写一段内容进去，来标识不同的用户。如果下次用户再访问这个网页的时候，它又能够读出这个文件里面的内容，这样网页就知道上次这个用户已经访问过该网页了。

**简言之cookie是web服务器保存在客户端的一系列文本信息.**

### 2、cookie作用

   1.对特定对象的追踪. 商城网站记录购物者浏览对象。

   2.保存用户网页浏览记录与习惯。

   3.简化登录。

### 3、浏览器查看cookie

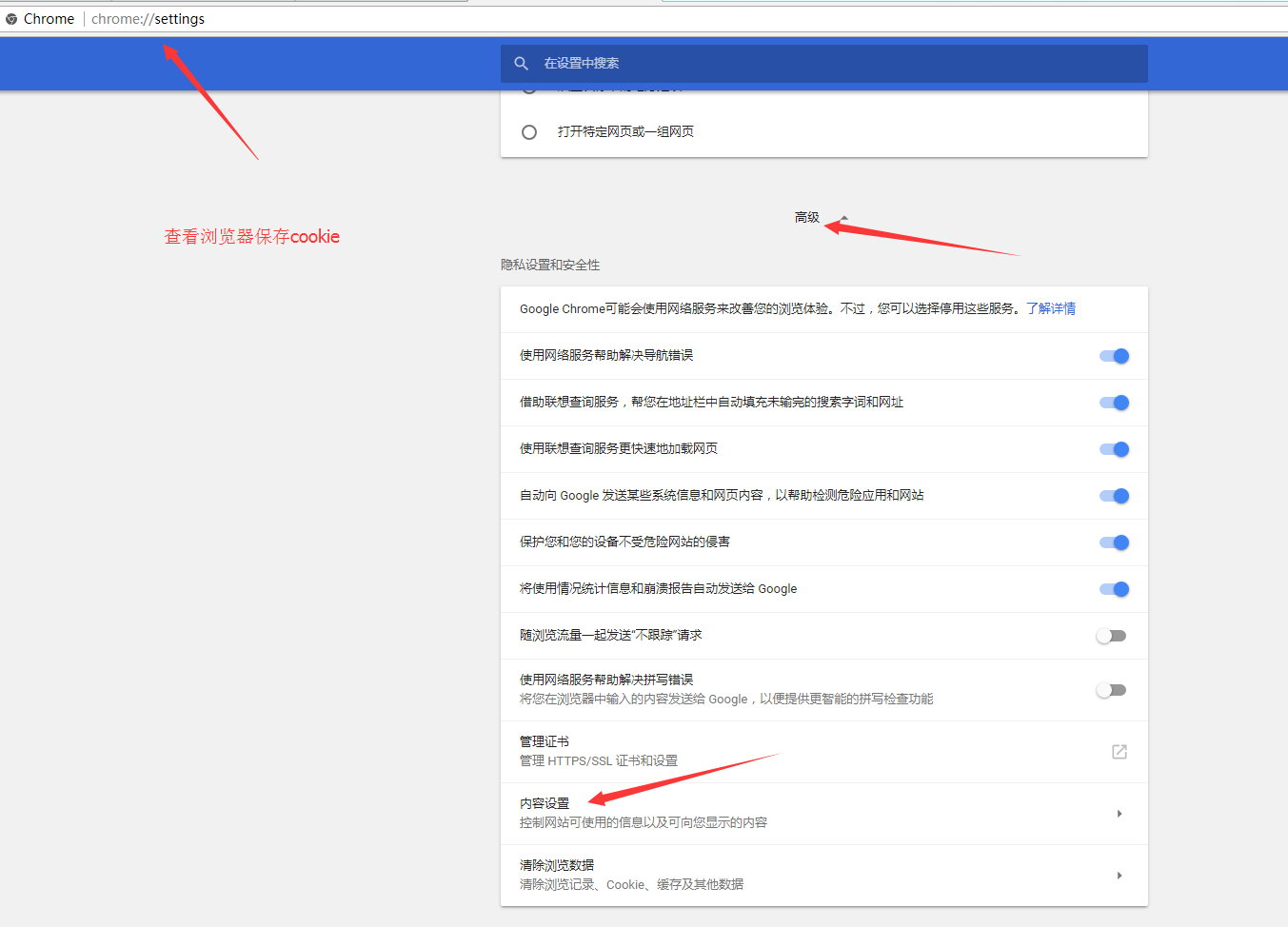
crome浏览器

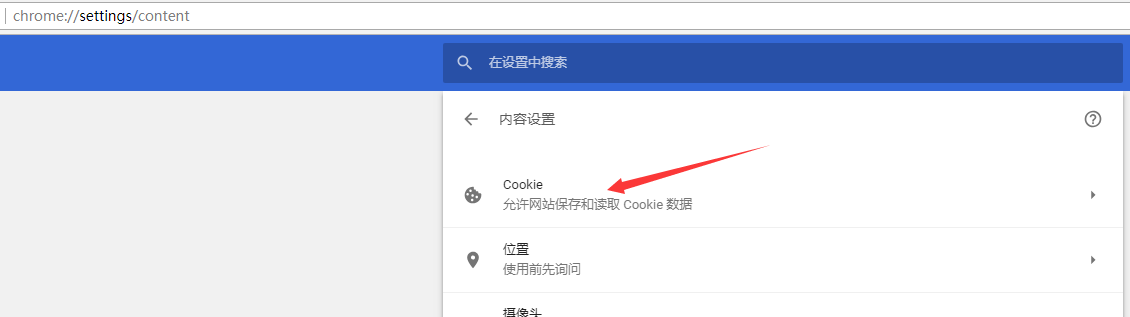
打开Chrome

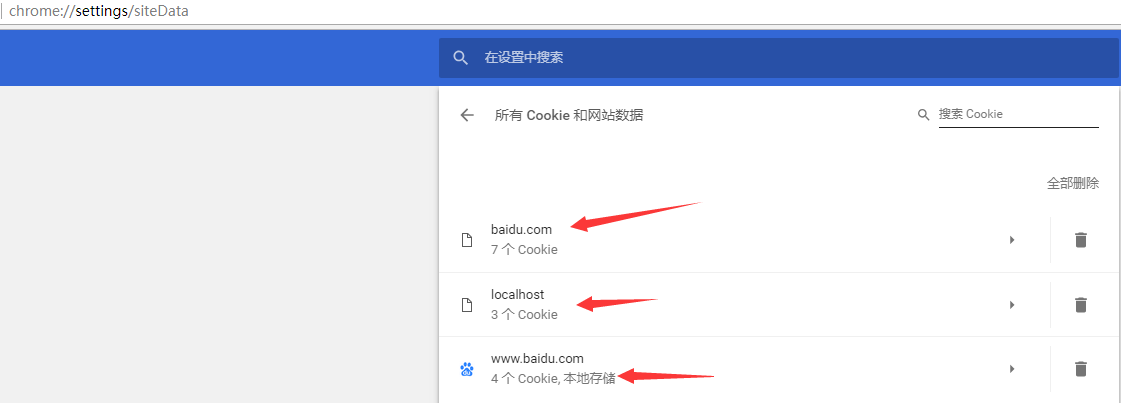
1.进入Chrome设置

2.找到【隐私设置】

3.进入【内容设置】







### 4.cookie-parser中间件

name=value：键值对，可以设置要保存的 Key/Value，注意这里的 name 不能和其他属性项的名字一样

Expires： 过期时间（秒），在设置的某个时间点后该 Cookie 就会失效，如 expires=Wednesday, 09-Nov-99 23:12:40 GMT

maxAge： 最大失效时间（毫秒），设置在多少后失效

secure： 当 secure 值为 true 时，cookie 在 HTTP 中是无效，在 HTTPS 中才有效

Path： 表示 cookie 影响到的路，如 path=/。如果路径不能匹配时，浏览器则不发送这个Cookie

npm install cookie-parser;

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

app.use(cookie()); // 引cookie中间件

发帝cookie信息到客户端

res.cookie('user', 'admin' , //名称,值

{

maxAge: 1000 \* 60 \* 60,

expires: '2020-1-1',

path: '/',

secure: false,

});

获取保存在客户端的cookie信息

req.cookies.cookie-name

清除cookie

res.clearCookie(cookie-name);

### 实现记住密码-cookie



/\*

\* 登录提交

\*/

userRouter.post('/login.do', function (req, res, next) {

var userName = req.body.name;

var passWord = req.body.password;

var autoLogin = req.body.autoLogin;

var sql = 'SELECT id,username,password FROM user WHERE username = ? AND password = ?';

var parameters = [userName, passWord];

querydb(sql, parameters)

.then(function (data) {

if (data.length === 0) {

res.render('login', {

'message': '用户名或密码出错！',

});

} else {

// 登陆成功后将用户和密码写入Cookie，maxAge为cookie过期时间

if (autoLogin === 'on') {

res.cookie('user', {

'username': req.body.name,

'password': req.body.password,

}, {

maxAge: 1000 \* 60 \* 60,

});

} else {

// 删除cookie

res.clearCookie('user');

}

res.redirect('/main.do');

}

}).catch(next);

});

/\*\*

\* 登录界面

\*/

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

if (req.cookies.user) {

res.render('login', {

'user': {

'username': req.cookies.user.username,

'password': req.cookies.user.password,

},

});

} else {

res.render('login');

}

});

<form action="login.do" method="post" enctype="application/x-www-form-urlencoded">

<h1>后台管理中心</h1>

<div >

<input type="text" name="name" placeholder="登录账号" value="<%= locals.user != null?user.username:'' %>"/>

</div>

<div >

<input type="password" name="password" placeholder="登录密码"value="<%= locals.user != null?user.password:'' %>" />

</div>

<div >

<input type="checkbox" name="autoLogin" checked> 记住密码

</div>

<div style="color: red;"> <%= locals.message !=null?message:'' %></div>

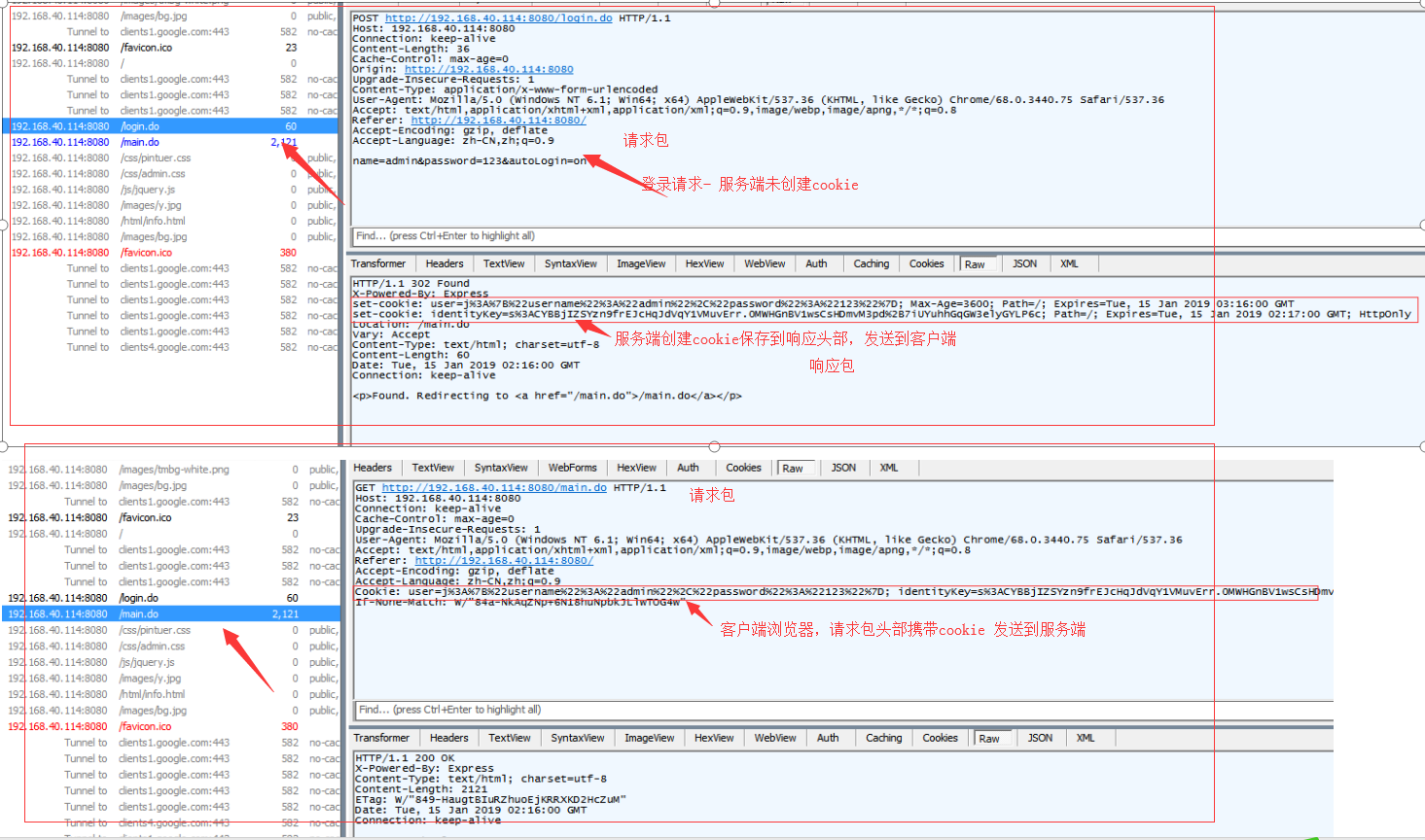
<div>

<input type="submit" value="登录">

</div>

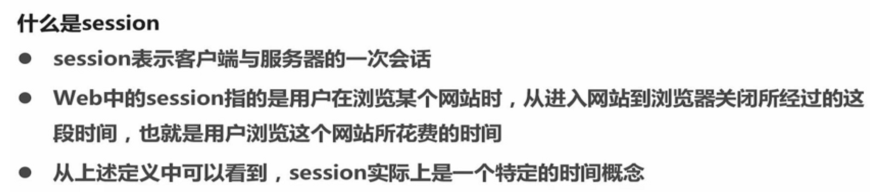
</form>

### 抓包查看cookie



## 2、session技术

### 1.什么是session



graphic

### 2.理解session

Session是在客户端请求到达服务器时，服务器为此请求发出的客户所创建的一个对象，保存在服务器端。购物车是一个很好的例子，一个用户可以有很多session，但每个session只针对一个用户，这就保证了不同session之间的信息独立。

        首先说明一点，在通常意义上，session所能发挥作用是基于cookie机制。针对所需要解释的问题，做这样一个假设：我们第一次访问一个网页。当客户端发送请求后，服务端会建立一个针对此请求发出客户的session对象，而且每个session都会有一个sessionID。服务端会自动将这个sessionID作为一个cookie附加到response上返回给客户端，这个cookie存放在浏览器内存中。我们每次对此网页发送的request都会附带着这个cookie，服务端收到这个请求后会都去cookie中取得这个sessionID，然后查询服务端是否存在一个对应此ID的session对象。如果有，可以直接使用此session；如果没有，则会新建一个。当浏览器关闭后，其所占的内存就会是放掉，cookie自然也就被清除了，此时我们不再保存有这个sessionID。所以再打开浏览器访问同一个页面时，由于没有sessionID，也就查不到对应的session对象，此时重新创建一个新的session对象。

        那当我们关闭浏览器之后，服务器端原来的session对象是否还存在呢？答案是肯定的。服务端根本不知道我们是否关闭了浏览器，也不关心这个。客户端与服务端之间进行通信的唯一途径就是通过请求。服务器有自己的一套机制来管理session，比如多长时间会清除没有使用过的session对象，等等。

        所以说，关闭浏览器session就被清除只是我们所看到的表面现象(实际上是新建了一个session对象)，通常情况下，服务器并不会马上清除session对象，但这个根据服务端的设定而不同。

        PS：cookie一般分为两种：一种是会话cookie，即服务端为session自动创建的cookie，这个cookie存放在浏览器进程中。另一种是可以存放在硬盘上的，可以通过服务端的某些设置，将一些信息放到cookie中并返回给客户端存放在硬盘上。

### 3.express-session中间件

express-session中间件将会话数据存储在服务器上；它仅将会话标识（而非会话数据）保存在 cookie 中。从1.5.0版本开始, express-session不再依赖cookie-parser,直接通过req/res读取/写入;默认存储位置内存存储(服务器端),

安装: npm install express-session

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

// 引入session中间件

app.use(session({ // 这里的name值得是cookie的name，默认cookie的name是：connect.sid

name: 'web1803',

secret: 'websecret',

cookie: ({

maxAge: 1000 \* 60 \* 60 \* 24,

}),

// 重新保存：强制会话保存即使是未修改的。默认为true但是得写上

resave: true,

// 强制“未初始化”的会话保存到存储。

saveUninitialized: true,

}));

主要方法 : **session(options)**

通过option来设置session存储，除了session ID外，session中的任何数据都不存储在cookie中。  
options可选参数:

name - cookie的名字（ 原属性名为 key）。（ 默认：’ connect.sid’）

store - session存储实例

secret - 用它来对session cookie签名， 防止篡改

cookie - session cookie设置（ 默认： { path: ‘ /‘, httpOnly: true,secure: false, maxAge: null }）

genid - 生成新session ID的函数（ 默认使用uid2库）

rolling - 在每次请求时强行设置cookie， 这将重置cookie过期时间（ 默认： false）

resave - 强制保存session即使它并没有变化（ 默认： true）

proxy - 当设置了secure cookies（ 通过” x - forwarded - proto” header） 时信任反向代理。 当设定为true时，” x - forwarded - proto” header 将被使用。 当设定为false时， 所有headers将被忽略。 当该属性没有被设定时， 将使用Express的trust proxy。

saveUninitialized - 强制将未初始化的session存储。 当新建了一个session且未设定属性或值时， 它就处于未初始化状态。 在设定一个cookie前， 这对于登陆验证， 减轻服务端存储压力， 权限控制是有帮助的。（ 默认： true）

unset - 控制req.session是否取消（ 例如通过 delete， 或者将它的值设置为null）。 这可以使session保持存储 状态但忽略修改或删除的请求（ 默认： keep）

express-session的一些方法:

1. Session.destroy():删除session，当检测到客户端关闭时调用。

2. Session.reload():当session有修改时，刷新session。

3. Session.regenerate()：将已有session初始化。

4. Session.save()：保存session。

### 4.登录权限验证

// 保存登录状态到session

req.session.user = userName;

/\*\*

\* 主界面

\*/

userRouter.get('/main.do', function (req, res, next) {

console.log('req.session.user :' + req.session.user);

if (req.session.user) {

res.render('index', {

'title': '主界面',

'name': '首页',

});

} else {

res.redirect('/'); // 重定向到登录界面

}

});

/\*\*

\* 退出

\*/

userRouter.get('/logout.do', function (req, res, next) {

req.session.destroy(); // 销毁session

res.redirect('/');

});

## 3、cookie与session区别



## 4、重定向redirect传参与flash一起使用

flash方法在express 3.X的时候分离出去了，现在使用需要引用connect-flash模块，且只能放在session模块引用下面。

使用时先设置session，然后在路由前设置flash，

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

// 设置flash，这步要在路由之前

userRouter.use(flash());

/\*\*

\* 主界面

\*/

userRouter.get('/main.do', function (req, res, next) {

// 判断session 状态，如果有效，则返回主页，否则转到登录页面

if (req.session.username) {

res.render('index', {

'title': '主界面',

'name': '首页',

});

} else {

req.flash('message', '你还没有登录'); // 保存信息

//req.session.msg = '您还没有登录,请登录11!';

res.redirect('/');

}

});

/\*\*

\* 登录界面

\*/

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

var msg = req.flash('message'); // 获取信息之后销毁session中message对象

//var msg = req.session.msg;

console.log('msg :' + msg);

res.render('login', {

'message': msg,

'user': {

'username': req.cookies.user.username,

'password': req.cookies.user.password,

},

});

});

## 5、四大对象 - app.locals与res.locals

生命周期

Application对象 web服务器启动时创建 服务器停止时结束 var app = express(); app.set(name,value); var value = app.get(name);

Session对象 第一次http请求时创建 关闭浏览器时 | 退出程序 session.destory() session时间到期结束 var sess = req.session; sess.name = value; var value = sess.name;

Request对象 发送请求时创建，请求结束销毁 请求行, 头部,正文

Response对象 响应时创建，响应结束销毁 响应行,头部，正文

Application####

app对象一般是Express应用程序，通过使用入口函数express()导出

var express=require("express");

var app=express();

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

res.send("hello world");

});

app.listen(3000);

app对象有处理以下事务的一些方法：

* 处理http请求的路由：请看app.METHOD和app.param
* 配置中间件：请看app.route
* 渲染HTML视图：请看app.render
* 设置模板引擎：请看app.engine  
  app同样可以设置一些影响app工作的属性，获取更多信息，请看Application settings

属性#####

**app.locals**  
app.locals对象是一个JavaScript对象，并且它的属性是应用程序的本地变量

app.locals.title

// =>"My App"

app.locals.email

// =>"me@myapp.com"

一旦设置了app.locals的属性，这个属性就会存在于应用程序的整个生命周期，二而不同于res.locals设置的属性只存在请求的生命周期中

你可以在应用程序的模板中使用本地变量，这对于模板和应用级数据来说提供了很有用的辅助功能，但是你在中间件中并不能访问本地变量

* app.locals.title="My App";
* app.locals.strftime=require("strftime");
* [app.locals.email="me@myapp.com";](mailto:app.locals.email=\"me@myapp.com\";)

res.locals.name = '首页';

if (req.session.user) {

res.render('index', {

'title': '主界面',

// 'name': '首页',

});

}

在调用 res.render 的时候，express 合并（merge）了 3 处的结果后传入要渲染的模板，优先级：res.render 传入的对象> res.locals 对象 > app.locals 对象，所以 app.locals 和 res.locals 几乎没有区别，都用来渲染模板，使用上的区别在于：app.locals 上通常挂载常量信息（如博客名、描述、作者信息），res.locals 上通常挂载变量信息，即每次请求可能的值都不一样（如请求者信息，res.locals.user = req.session.user）。

## 6、统一登录认证-拦截器

/\*\*

\* 统一登录认证--拦截

\*/

userRouter.all('/\*', function (req, res, next) {

console.log('登录认证拦截 url : ' + req.url);

// url处理方式一

// var arr = req.url.split('/'); // 解析用户请求的路径

// for (var i = 0, length = arr.length; i < length; i++) { // 去除 GET 请求路径上携带的参数

// arr[i] = arr[i].split('?')[0];

// }

// if (arr.length > 1) {

// if (arr[1] === 'login.do' || arr[1] === 'logout.do' || arr[1] === 'register.do') {

// next();

// } else {

// logger.error('intercept：用户未登录执行登录拦截，路径为：' + arr[1]);

// res.redirect('/maindo'); // 将用户重定向到登录页面

// res.end();

// }

// }

// url处理方式二

var reqUrl = req.url + ''; // 对象转字符串 /user/list/main.do?username=admin

if (reqUrl.indexOf('?') !== -1) { // 判断reqUrl有没有?,如果没有返回-1

reqUrl = reqUrl.substring(0, reqUrl.lastIndexOf('?')); // /main.do

}

console.log('Url :' + reqUrl);

// 指定请求放行

if (reqUrl.endsWith('login.do') || reqUrl.endsWith('logout.do') || reqUrl.endsWith('/') || reqUrl === '') {

next();

} else {

if (req.session.user) { // 登录认证

next();

} else {

req.session.originalUrl = req.originalUrl; // 保存原始请求url地址到session中

req.flash('msg', '您还没有登录,请登录!');

res.redirect('/'); // 重定向到登录界面

}

}

});

/\*

\* 登录提交

\*/

userRouter.post('/login.do', function (req, res, next) {

var userName = req.body.name;

var passWord = req.body.password;

var autoLogin = req.body.autoLogin;

var sql = 'SELECT id,username,password FROM user WHERE username = ? AND password = ?';

var parameters = [userName, passWord];

querydb(sql, parameters)

.then(function (data) {

if (data.length === 0) {

res.render('login', {

'message': '用户名或密码出错！',

});

} else {

// 判断是否需要记住密码

if (autoLogin === 'on') {

res.cookie('user', {

'username': userName,

'password': passWord,

}, {

maxAge: 1000 \* 60 \* 60 \* 24, // cookie信息保存一天

});

} else {

res.clearCookie('user'); // 清除cookie

}

// 保存登录状态到session

req.session.user = userName;

// req.app.user = userName;

//-----实现登录后跳转到原始请求页面----

var redirectUrl = '/main.do';

if (req.session.originalUrl) {

redirectUrl = req.session.originalUrl;

req.session.originalUrl = null;

}

// 重定向到主界面

res.redirect(redirectUrl);

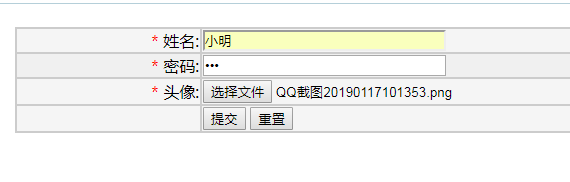
}

})

.catch(next);

});

# 八、文件上传



概览

图片上传是web开发中经常用到的功能，node社区在这方面也有了相对完善的支持。

常用的开源组件有****multer****、****formidable****等，借助这两个开源组件，可以轻松搞定图片上传。

下面以****multer为例****

fileupload\_util.js

var multer = require('multer')

var fs = require('fs');

// 指定上传文件名

var uploadFolder = './uploaddir/';

var createFolder = function (folder) {

try {

fs.accessSync(folder); //如果文件目录存在覆盖

} catch (e) {

fs.mkdirSync(folder);

}

};

createFolder(uploadFolder); // 创建图片保存目录

var storage = multer.diskStorage(

{

destination: function (req, file, cb) {

cb(null, uploadFolder); // 保存的路径，备注：需要自己创建

},

filename: function (req, file, cb) {

// 将保存文件名设置为 字段名 + 时间戳，比如 logo-1478521468943

//文件类型：%s', file.mimetype

//原始文件名：%s', file.originalname

//文件大小：%s', file.size

//文件保存路径：%s', file.path

cb(null, file.fieldname + '-' + Date.now() + '.jpg');

}

});

// 通过 storage 选项来对 上传行为 进行定制化

var myUpload = multer({

storage: storage

})

module.exports = myUpload;

app.js

var express = require('express');

var multer = require('multer')

var fileUpload = require('../utils/fileupload\_util');

var indexRoute = express.Router();

indexRoute.get('/file', function (req, res, next) {

res.render('form');

});

indexRoute.post('/upload2', fileUpload.single('logo'), function (req, res, next) {

res.send("文件上传成功");

});

form.ejs

<html>

<head>

<title>文件上传</title>

<meta charset="utf-8">

</head>

<body>

<div>

<form action="/upload2" method="post" enctype="multipart/form-data">

<h2>单图上传</h2>

<input type="file" name="logo">

<input type="submit" value="提交">

</form>

</div>

</body>

</html>

# 九、分页



### 分页原理分析： 根据页号获取记录

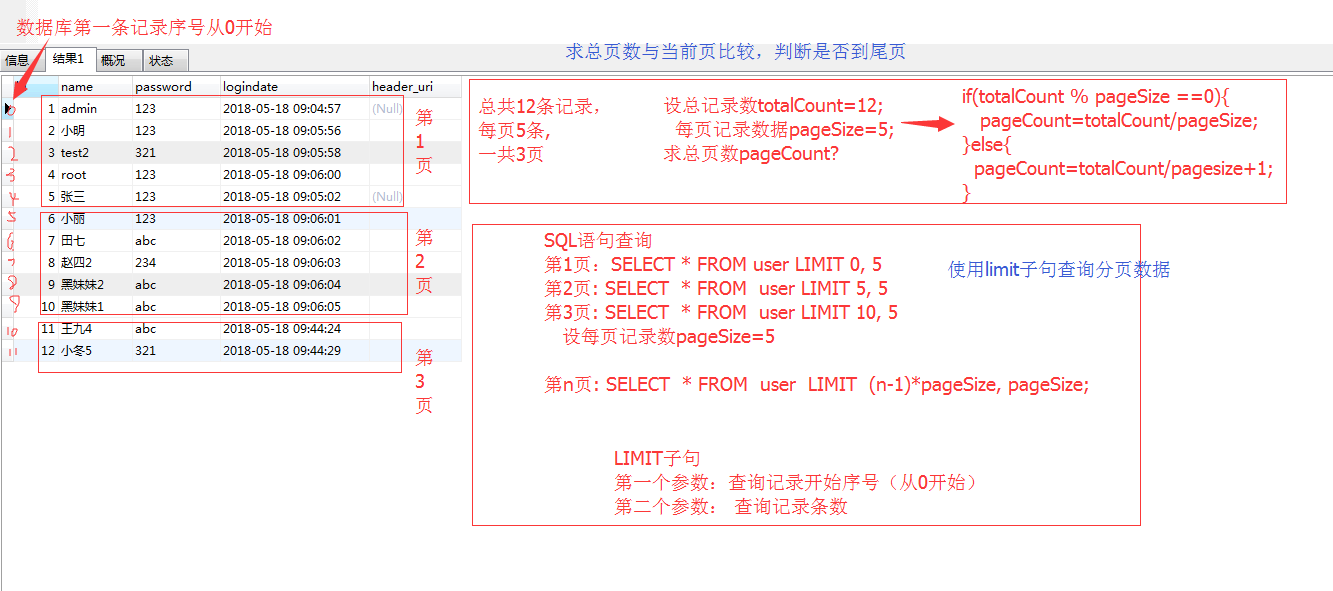
SELECT id,username,password,headerurl FROM user limit (1-1)\*5 , 5 //第1页

SELECT id,username,password,headerurl FROM user limit (2-1)\*5 , 5 //第2页

SELECT id,username,password,headerurl FROM user limit (3-1)\*5 , 5 //第3页

...................................................

SELECT id,username,password,headerurl FROM user limit (n-1)\*5 , 5 //第n页



var pageSize = 5; // 每页5条记录

var currentPage = 1; // 当前页

var startIndex = (currentPage - 1) \* pageSize; // 计算偏移量

var sql = 'SELECT id,username,password,headerurl FROM user limit ?, ?';

var paramers = [startIndex, pageSize];

querydb(sql, paramers)

.then(function (data) {

res.render('user/user\_list', {

'userLists': data,

'currentPage': currentPage,

});

})

.catch(next);

计算总页数

var pageSize = 5; 每页5条记录

var pageCount; 总页数

var total = 12; 总记录数

if(total % pageSize === 0){

pageCount = parseInt(total / pageSize); // 2

}else{

pageCount = parseInt(total / pageSize) + 1; // 3

}

### 2.完整示例

分页后端代码 userRoute.js

userRouter.get('/list.do', function (req, res, next) {

var pageCount = 0; // 总页数

var pageSize = 5; // 每页5条记录

var currentPage = 1; // 当前页

if (req.query.pageNo) {

currentPage = req.query.pageNo;

}

var sqlCount = 'SELECT count(\*) AS count FROM user';

querydb(sqlCount)

.then(function (data) {

console.log(data[0].count);

// ---计算总页数----

var total = data[0].count; // 总记录条数

if (total % pageSize === 0) {

pageCount = parseInt(total / pageSize);

} else {

pageCount = parseInt(total / pageSize) + 1;

}

// ---计算总页数----

var startIndex = (currentPage - 1) \* pageSize; // 计算偏移量

var sql = 'SELECT id,username,password,headerurl FROM user limit ?, ?';

var paramers = [startIndex, pageSize];

return querydb(sql, paramers);

})

.then(function (data) {

res.render('user/user\_list', {

'userLists': data,

'currentPage': currentPage,

'pageCount': pageCount,

});

})

.catch(next);

});

Html页面 list.ejs

<div>

<a onclick="first()" href="#">首页</a>

<a onclick="previousPage()" href="#">上一页</a>

<a onclick="next()" href="#">下一页</a>

<a onclick="lastPage()" href="#">尾页</a>

</div>

<script>

//首页

function first() {

window.location.href = 'list.do?pageNo=1';

}

//下一页

function next() {

var currentPage = '<%= currentPage %>'; //在javascript代码中获取模板数据，需加单引号 '<%= currentPage %>'

var pageCount = ' <%= pageCount %> ';

if (++currentPage > pageCount) {

alert('已经到尾页');

return;

}

window.location.href = 'list.do?pageNo=' + currentPage;

}

// 上一页

function previousPage() {

var currentPage = '<%= currentPage %>';

if (--currentPage == 0) {

alert("已经到第一页");

currentPage = 1;

return;

}

window.location.href = 'list.do?pageNo=' + currentPage;

}

// 尾页

function lastPage() {

var pageCount = '<%= pageCount %> ';

window.location.href = 'list.do?pageNo=' + pageCount;

}

</script>

### 3.分页插件



//分页条显示位置html代码

<div>

<ul class="pagination" id="pagination"></ul>

</div>

分页插件jqPaginator.js 需要 Boostrap样式,jquery支持

<link href="http://cdn.bootcss.com/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">

<script src="js/jquery.js" type="text/javascript"></script>

<script type="text/javascript" src="js/jqPaginator.js"></script>

<script type="text/javascript">

$.jqPaginator('#pagination', {

totalPages: 100, //总页数; 获取模板数据方法, html代码: <%= 变量名 %> ; js代码: '<%= 变量名 %>';

visiblePages: 2,

currentPage: 1, //当前页

first: '<li class="first"><a href="javascript:void(0);">首页</a></li>',

prev: '<li class="prev"><a href="javascript:;">前一页</a></li>',

next: '<li class="next"><a href="javascript:void(0);">下一页</a></li>',

last: '<li class="last"><a href="javascript:void(0);">尾页</a></li>',

page: '<li class="page"><a href="javascript:;">{{page}}</a></li>',

onPageChange: function (num, type) {

if (type == 'change') {

//这里是点击分页的回调

window.location.href = 'list.do?pageNo=' + num;

}

}

});

</script>

# 十、AJAX

## AJAX介绍

### ajax是什么？

Ajax：现在允许浏览器与服务器通信而无须刷新当前页面的技术都被叫做Ajax. 或者是不需要刷新整个页面，只刷新局部页面的一种异步通讯技术；



### Ajax技术的核心: XMLHttpRequest对象：发送请求到服务器并获得返回结果

#### 常用方法:

|  |  |
| --- | --- |
| **方法名** | **说    明** |
| **open(method,URL,async)** | **建立与服务器的连接**  **method参数指定请求的HTTP方法,典型的值是GET或POST**  **URL参数指定请求的地址**  **async参数指定是否使用异步请求，其值为true或false** |
| **send(content)** | **发送请求**  **content参数指定请求的参数** |
| **setRequestHeader(header,value)** | **设置请求的头信息,把首部设置为所提供的值，在设置首部之前必须先调用open()** |
| **Onreadystatechange(e)** | **指定回调函数** |

#### 常用属性

**onreadystatechange：指定回调函数**

**readystate: XMLHttpRequest的状态信息**

**status：HTTP的状态码（200对应OK，404对应NotFount等）**

**statusText：HTTP状态码的相应文本（OK或NotFount等）**

**responseText：获得响应的文本内容，表示一个字符串**

**responseXML：获得响应的XML文档对象，可以解析为DOM对象**

|  |  |  |  |
| --- | --- | --- | --- |
| **就绪状态码 readystate** | | **说    明** | |
| **0** | | **XMLHttpRequest对象没有完成初始化** | |
| **1** | | **XMLHttpRequest对象开始发送请求** | |
| **2** | | **XMLHttpRequest对象的请求发送完成** | |
| **3** | | **XMLHttpRequest对象开始读取响应，还没有结束** | |
| **4** | | **XMLHttpRequest对象读取响应结束** | |
| **状态码status** | **说    明** | |
| **200** | | **服务器响应正常** |
| **400** | | **无法找到请求的资源** |
| **403** | | **没有访问权限** |
| **404** | | **访问的资源不存在** |
| **500** | | **服务器内部错误** |

### 示例

//获取XMLHttpRequest对象

function getXHR() {

if (window.XMLHttpRequest) {

return new window.XMLHttpRequest;

} else {

// 兼容IE浏览器低版本

new window.AtiveXObject("Microsoft.XMLHTTP");

}

}

var xhr = getXHR(); // 获取 XMLHttpRequest对象

//发送请求

xhr.open('post', '/checkUser.do', true); //建立连接

xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded"); // 请求头部

xhr.send('username=' + nameElement.value); //发送请求

//接收axja响应

xhr.onreadystatechange = function (e) {

if (e.target.readyState == 4) { // Ajax请求的状态 0, 1, 2, 3, 4:响应完成

if (e.target.status == 200) { // 响应状态码

alter(e.target.responseText);

} else {

alert(e.target.status + 'e.target.status ' + e);

}

}

}

#### 动态添加元素检查用户名是否存在

var xhr = getXHR(); // 获取 XMLHttpRequest对象

xhr.open('post', '/checkUser.do', true); //建立连接

xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded"); // 请求头部

xhr.send('username=' + nameElement.value); //发送请求

//接收axja响应

xhr.onreadystatechange = function (e) {

if (e.target.readyState == 4) { // Ajax请求的状态 0, 1, 2, 3, 4:响应完成

if (e.target.status == 200) { // 响应状态码

console.log(e.target.responseText);

var json = JSON.parse(e.target.responseText);

if (json.code === 1) { //用户已经存在

var newSpan = document.createElement('span'); //创建新节点 span <span></span>

newSpan.innerHTML = json.msg; // 给新节点添加内容 <span>用户已经存在</span>

tdElement.insertBefore(newSpan, nameElement.nextSibling); //在tdElement元素节点下，nameElement节点后一个元素前插入新节点newSpan,

} else {

//判断tdElement最后一个元素的前一个元素节点名称是否是SPAN,如果是SPAN将其移除

if (tdElement.lastChild.previousSibling.nodeName == 'SPAN') {

tdElement.removeChild(tdElement.lastChild.previousSibling); //移除tdElement最后一个元素的前一个元素节点,即新添加的元素节点

}

}

} else {

alert(e.target.status + 'e.target.status ' + e);

}

}

}

#### 封装原生ajax

/myajax.js

/\*\*

\* 封装原生ajax

{

method: 'post',

url: '/checkuser.do',

content: '发送内容'，

success: function (e) {},

}

\*/

function ajax(json) {

var method = json.method;

var url = json.url;

var content = json.content;

var xhr = null;

if (window.XMLHttpRequest) {

xhr = new window.XMLHttpRequest();

} else {

// 兼容IE浏览器低版本

xhr = new window.AtiveXObject('Microsoft.XMLHTTP');

}

if (xhr !== null) {

xhr.open(method, url, true); // 建立连接

xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded'); // 请求头部

xhr.send(content);

xhr.onreadystatechange = function (e) {

if (e.target.readyState === 4) { // Ajax请求的状态 0, 1, 2, 3, 4:响应完成

if (e.target.status === 200) { // 响应状态码

var text = e.target.responseText;

json.success(text);

}

}

};

}

}

调用代码 login.html

<script src="js/myajax.js" type="text/javascript"></script>

<script>

var tdElement = document.getElementById('boxtd');

var nameElement = document.getElementById('clientname');

nameElement.onblur = function () {

ajax({

method: 'post',

url: '/checkUser.do',

content: 'username=' + nameElement.value,

success: function (data) {

console.log(data);

var json = JSON.parse(data);

//判断tdElement最后一个元素的前一个元素节点名称是否是SPAN,如果是SPAN将其移除

if (tdElement.lastChild.previousSibling.nodeName == 'SPAN') {

tdElement.removeChild(tdElement.lastChild.previousSibling); //移除tdElement最后一个元素的前一个元素节点,即新添加的元素节点

}

if (json.code === 1) {

var newSpan = document.createElement('span'); //创建新节点 span

newSpan.innerHTML = json.msg; // 给新节点添加内容

tdElement.insertBefore(newSpan, nameElement.nextSibling); //在tdElement元素节点下，nameElement节点后一个元素前插入新节点newSpan,

}

}

});

}

</script>

## jquery 实现AJAX

$("#clientname").blur(function () {

//load get post ajax

$.ajax({

url: "/checkUser.do", //规定发送请求的 URL。默认是当前页面。

type: "post", //http请求类型 get|post

data: "username=" + $(this).val(), //规定要发送到服务器的数据。

dataType: "json", //预期的服务器响应的数据类型; json|html/text

success: function (data) { //   当请求成功时运行的函数。

$("#msg").empty(); //清除 id =msg元素

if (data.code == 1) {

$("#clientname").after(

"<span id='msg' style='color:red;font-size:14px'>"+data.msg+"</span>");

}

},

error: function (xhr, status, error) { //如果请求失败要运行的函数。

alert(error + " " + status);

}

});

});

post/get请求方式

$.post(

'/checkUser.do',

'username=' + $(this).val(),

function (result) {

$('#msg').empty();

if (result.code === 1) {

$('#clientname').after('<span id="msg" style="color:red; font-size:14px">' + result.msg + '</span>');

}

}

);

$.get('/checkusr.do?username=' + $('#clientname').val(), function (result) {

alter(result);

});

# 十一、加密技术

## **1.密码常用术语**

明文： 待加密码信息

密文： 经过加密后的明文

加密： 明文转为密文的过程

加密算法： 明文转为密文的转换算法

加密密钥： 通过加密算法进行加密操作用的密钥

解密： 将密文转为明文的过程

解密算法： 密文转为明文的算法

解密密钥：通过解密算法进行解密操作用的密钥

柯克霍夫原则： 数据的安全基于密钥而不是算法的保密。即系统的安全取决于密钥，对密钥保密，对算法共开。——现代密码学设计原则

明文:

指挥部                               作战部队

             2019.6.20攻打海湾

             4154\*rqfasfv13wrm

## 2.常用加密码算法

### **1.**Hash算法(散列算法)

　　Hash算法特别的地方在于它是一种单向算法，用户可以通过Hash算法对目标信息生成一段特定长度的唯一的Hash值，却不能通过这个Hash值重新获得目标信息。因此Hash算法常用在不可还原的密码存储、信息完整性校验等。

　　常见的Hash算法有MD2、MD4、MD5、HAVAL、SHA

打印支持的Hash算法

var crypto = require('crypto'); // 加载crypto库

console.log(crypto.getHashes()); // 打印支持的hash算法

输出结果:

[ 'RSA-MD4',

'RSA-MD5',

'RSA-MDC2',

'RSA-RIPEMD160',

'RSA-SHA1',

'blake2b512',

'blake2s256',

'md4',

'md4WithRSAEncryption',

'md5',

'md5-sha1',

'md5WithRSAEncryption',

'mdc2',

.....

]

加密示例

var content = 'b123'; // 加密的明文；

var md5 = crypto.createHash('md5'); // 定义加密方式:md5不可逆,此处的md5可以换成任意hash加密的方法名称；

md5.update(content);// 加密

var d = md5.digest('hex'); // 加密后的值d

console.log('加密的结果：' + d);

### 2.对称加密

指加密和解密使用相同密钥的加密算法。对称加密算法的原理很容易理解，通信一方用KEK加密明文，另一方收到之后用同样的KEY来解密就可以得到明文。

　　常见的对称加密算法有DES、3DES、Blowfish、IDEA、RC4、RC5、RC6和AES

打印支持的对称加密算法

var crypto = require('crypto'); // 加载crypto库

console.log(crypto.getCiphers()); // 打印支持的cipher算法

结果:

[ 'aes-128-cbc',

'aes-128-cbc-hmac-sha1',

'aes-128-cbc-hmac-sha256',

'aes-128-ccm',

'aes-128-cfb',

'aes-128-cfb1',

'aes-128-cfb8',

'aes-128-ctr',

'aes-128-ecb',

'aes-128-gcm',

'aes-128-ocb',

.....

]

示例:

var crypto = require('crypto'); // 加载crypto库

// console.log(crypto.getCiphers()); // 打印支持的cipher算法

var secret = 'web123key'; //秘钥

var str = 'helloword'; // 明文

var cipher = crypto.createCipher('aes192', secret); // 定义加密方式，两个参数分别是加密算法、密码

cipher.update(str, 'utf8', 'hex'); // 加密，编码方式从utf-8转为hex;

var enc = cipher.final('hex'); // 编码方式重转为hex;

console.log(enc);

var str1 = enc; // 密文

var decipher = crypto.createDecipher('aes192', secret);

decipher.update(str1, 'hex', 'utf8'); // 编码方式从hex转为utf-8;

var dec = decipher.final('utf8'); // 编码方式重转utf-8;

console.log(dec);

输出结果:

6a0bc4c22b0674d9551344bbd62c2312

helloword

### 非对称加密

不对称加密算法，使用两把完全不同但又是完全匹配的一对Key:公钥和私钥。在使用不对称加密算法加密文件时，只有使用匹配的一对公钥和私钥，才能完成对明文的加密和解密过程。

封装加密工具类

var crypto = require('crypto'); // 加载crypto库

exports.md5 = function (content) {

// md5加密password

var md5 = crypto.createHash('md5');

md5.update(content); // 加密

return md5.digest('hex');

};

var secret = 'web123key'; //秘钥

// 加密

exports.cipher = function (content) {

var str = content; // 明文

var cipher = crypto.createCipher('aes192', secret); // 定义加密方式，两个参数分别是加密算法、密码

cipher.update(str, 'utf8', 'hex'); // 加密，编码方式从utf-8转为hex;

var enc = cipher.final('hex'); // 编码方式重转为hex;

return enc;

};

// 解密

exports.decipher = function (enc) {

var str1 = enc; // 密文

var decipher = crypto.createDecipher('aes192', secret);

decipher.update(str1, 'hex', 'utf8'); // 编码方式从hex转为utf-8;

var dec = decipher.final('utf8'); // 编码方式重转utf-8;

return dec;

};

# 日志

## 1.Node.js 模块之 morgan中间件记录日志

NodeJs中Express框架使用morgan中间件记录日志

Express中的app.js文件已经默认引入了该中间件var logger = require('morgan');

使用app.use(logger('dev'));可以将请求信息打印在控制台，便于开发调试，但实际生产环境中，需要将日志记录在log文件里，可以使用如下代码

var express = require('express');

var fs = require('fs');

var logger = require('morgan');

var app = express();

var accessLog = fs.createWriteStream('../access.log', {flags : 'a'});

var errorLog = fs.createWriteStream('../error.log', {flags : 'a'});

app.use(logger('dev')); //打印到控制台

app.use(logger('combined', {stream : accessLog})); //打印到log日志

## 2..log4js

Console.log() 控制台显示日志，一旦server重启日志就丢失了。对于程序开发来说，这样的输出已经够用了。但是在生产环境上，我们希望能把控制台的输出，保存到文件中，而且需要更多的信息，不仅仅是默认的简化的日志信息。由于express框架没有日志功能，我们需要引入log4js包来完成这个功能。

### 安装

npm install log4js

### 基本使用

var log4js = require('log4js'); // 引入日志包

var logger = log4js.getLogger(); // 获取日志对象

logger.level = 'trace'; // 设置日志级别

// 打印日志

logger.trace('trace >>>>');

logger.debug('debug >>>>');

logger.info('info >>>>');

logger.warn('warn >>>>');

logger.error('error >>>>');

logger.fatal('fatal >>>>');

日志的输出级别,共: ALL < TRACE < DEBUG < INFO < WARN < ERROR < FATAL < MARK < OFF八个级别,default level is OFF

只有大于等于日志配置级别的信息才能输出出来:

如果日志级别设为 info 那么大于等info级别的日志信息会输出，小于则不会；上述代码只能输出 info>>>>, warn>>>> ,error>>>, fatal>>>

### 配置

日志信息默认输出到控制台，在log4js中可以配置输出到文件;

appenders:

   配置文件的输出源，一般日志输出type共有console、file、dateFile三种

 \*  console:普通的控制台输出

 \*  file:输出到文件内，以文件名-文件大小-备份文件个数的形式rolling生成文件

  \*  dateFile:输出到文件内，以pattern属性的时间格式，以时间的生成文件

使用配置

log4js.configure({

// 定义输出源

appenders: {

'rule\_console': {

type: 'console',

},

'access': {

type: 'file',

filename: 'log/access.log',

},

'rule-file': {

'type': 'dateFile',

'filename': 'log/server-',

'encoding': 'utf-8',

'maxLogSize': 1024 \* 1024, // 文件大小

'numBackups': 3,

'pattern': 'yyyy-MM-dd.log',

'alwaysIncludePattern': true,

},

},

// 输出种类

categories: {

// 默认输出 var logger = log4js.getLogger();

default: {

appenders: ['rule\_console'],

level: 'debug', // 日志输出级别

},

// 名为 development的种类 var logger = log4js.getLogger('development');

development: {

appenders: ['rule\_console', 'access', 'rule-file'],

level: 'debug', // 日志输出级别

},

},

});

日志配置信息可以设置到json配置文件中引用

var log4jsConfig = require('./config/logConf.json');

log4js.configure(log4jsConfig);

Logconf.json

{

"appenders": {

"rule-console": {

"type": "console"

},

"access": {

"type": "file",

"filename": "log/access.log"

},

"rule-file": {

"type": "dateFile",

"filename": "log/server-",

"encoding": "utf-8",

"maxLogSize": 10000000,

"numBackups": 3,

"pattern": "yyyy-MM-dd.log",

"alwaysIncludePattern": true

},

"rule-error": {

"type": "dateFile",

"filename": "log/error-",

"encoding": "utf-8",

"maxLogSize": 1000000,

"numBackups": 3,

" pattern": "yyyy-MM-dd.log",

"alwaysIncludePattern": true

}

},

"categories": {

"default": {

"appenders": [

"rule-console",

"rule-file",

"rule-error"

],

"level": "debug"

},

"http": {

"appenders": [

"access"

],

"level": "info"

}

},

}

代替console.log()  
增加replaceConsole配置，让所有console输出到日志中，以[INFO] console代替console默认样式。

log4js.configure({

appenders: {

'rule\_console': {

type: 'console',

}

},

categories: {

default: {

appenders: ['rule\_console'],

level: 'INFO',

},

},

replaceConsole: true, // 增加replaceConsole配置，让所有console输出到日志中，以[INFO] console代替console默认样式。

});

使用

var logger = log4js.getLogger('development');

logger.trace('Entering cheese testing');

logger.debug('Got cheese.');

logger.info(is Gouda.');

logger.warn(is quite smelly.');

logger.error(is too ripe!');

logger.fatal(was breeding ground for listeria.');

### 与express集成

var logger = log4js.getLogger();

app.use(log4js.connectLogger(logger, {

level: 'auto', // 日志级别自动输出,指定日志级别： level：log4js.levels.INFOlog4js.levels.INFO

format: ':method :url',

}));

var log = require('log4js').getLogger();

log.debug('debug >>>>');

log.info('info >>>>');

log.warn('warn >>>>');

日志参考:https://blog.csdn.net/xjtroddy/article/details/51697077