# 复习

浏览器：用于浏览器服务器上内容的应用程序。

服务器：用于响应浏览器的请求的应用程序。

HTTP协议：

规定浏览器如何发出请求，服务器如何接收请求。以及服务器如何响应请求及浏览器如何解析请求。

HTTP请求：

根据HTTP协议发出的具体的一次请求行为。

组成：请求头、请求首行、请求空行、请求正文

请求首行：URL、请求方式

请求头：一些配置、设置、信息。

请求空行：空白行，用于分割请求头和请求正文

请求正文：存放数据的正式的地方

方式：根据本次发送的请求的目的而划分。

GET：从服务器上“获取”内容

POST：往服务器上“设置”内容

GET请求的特点：

没有请求正文

数据携带在URL的QUERY部分

携带数据量受地址栏的长度的限制

不能携带敏感数据

触发方式多

便于分享

POST请求特点：

具备请求正文

携带数据要带在正文中，容量没有上限

比较GET更加安全一些

触发方式少：AJAX和表单可以发送

不便于分享

状态码：用于标记本次请求的状态（成功还是失败）

200：成功

302：重定向

304：未修改

404：资源未找到

500：服务器错误

缓存：临时的存储机制 通常是由后端设置

强制缓存：如果两次的URL完全一致，并且请求头中没有任何标记，此时会触发强制缓存。（无论服务器上的内容是否修改过，总是使用浏览器缓存中的内容）

协商缓存：由服务器设置，如果要使用缓存中的数据，不会直接使用，而是查看缓存标记，如果标记还在就发送请求到达服务器，询问服务器“数据是否已经修改”如果修改过，就重新请求修改后的数据，携带回浏览器替换之前的缓存。如果没有修改过，则直接返回，不携带任何数据。回到浏览器之后，就让浏览器使用之前的缓存中的数据。

转码：URL中是不可以出现中文等特殊字符的。

解决方案：将中文字符以特定的编码方式转换为英文字符，传递给服务器，服务器再以同样的规则，解码。

加码函数：encodeURIComponent

解码函数：decodeURIComponent

当用户打开浏览器并输入地址按下回车之后：

1 浏览器解析字符串

2 按照域名转换域名为ip地址

查看浏览器的缓存中是否存在

查看系统缓存中是否存在

查看路由器缓存中是否存在

去请求DNS服务器  
3 根据得到的IP地址，申请建立TCP连接

4 发送请求

5 得到响应 （得到第一个页面的源代码）

6 解析这些源代码 渲染成页面供用户查看

从上到下依次解析

构建DOM树

构建样式树

遇见各种外链资源标签，重新发送新的HTTP请求

7 所有请求发送完毕 页面渲染完毕 用户看到页面

NodeJS：它是一个运行JS的环境

特点：单线程、非阻塞I/O、事件驱动

适用场景：大并发、IO多

模块化：

模块分类：内置模块、第三方模块

内置模块：HTTP模块、FS模块、QueryStrings模块、Path模块、HTTPS模块、URL模块

第三方模块：不属于内置模块的都是第三方模块

模块定义：

一个JS文件就是一个模块

模块引入：

通过reuqire引入模块

注：如果被引入的模块并没有处于node\_modules中，则必须以./开头

模块暴露：

通过module.exports或者exports打点添加属性

模块规范：CommonJS规范 属于同步规范

node\_modules文件夹：该文件夹用于存放第三方模块

存在于该文件夹内部的模块可以直接通过名称引入（好像核心模块一样）

该文件夹可以存在于同级、上级、上上级、……直到根目录

HTTP模块：该模块用于搭建HTTP服务器

引入: var http = require("http");

调用方法创建服务器对象:

var server = http.createServer(function(req, res) {

// 当前函数是用于处理前端请求的响应函数

// req 前端请求对象 包含一切前端数据

// res 后端响应对象 返回什么由res决定

})

监听到端口：

server.listen(3000);

# FS模块

FS模块是 File System模块的简称。

后端是可以操作后端服务器上的文件的。

想要使用FS模块，要先引入： var fs = require("fs");

## 1.1 创建文件

fs.appendFile(filePath, content, callback); 该方法用于向文件中追加内容 注：如果文件不存在，先创建再追加

filePath: 目标文件的路径 可以是相对 可以是绝对

如果是相对路径相对于工作目录而不是运行文件所在目录

content：要追加的内容

callback: 回调函数 回调函数中有一个参数 表示错误对象

demo:

|  |
| --- |
| 1. // 引入fs模块 2. var fs = require("fs"); 3. // 调用对应的创建方法 4. fs.appendFile("./index.txt", "内容", function(err) { 5. // err 表示在创建过程中出现的异常 如果有，就是错误对象，如果没有就是null 6. console.log(err); 7. }); |

## 1.2 读取文件

fs.readFile(filePath, callback); 方法用于读取某一个文件的内容

filePath: 目标文件的路径 可以是相对 可以是绝对

callback: 回调函数 回调函数中有两个参数

第一个表示错误对象

第二个表示读取到的文件的内容（buffer类型）

demo:

|  |
| --- |
| 1. // 读取同级别的index.html 2. var fs = require("fs"); 3. fs.readFile("./index1.html", function(err, data) { 4. console.log(err); 5. console.log(data); 6. // 如果成功才可以调用toString 7. // console.log(data.toString()); 8. }); |

## 1.3 删除文件

fs.unlink(filePath, callback); 该方法用于删除某一个指定的文件

filePath: 目标文件路径 可以是相对 可以是绝对

callback: 回调函数 函数有一个错误对象

demo:

|  |
| --- |
| 1. // 删除同级别的index.html文件 如果发现没有该文件 请实现创建一个 2. var fs = require("fs"); 3. // 删除 4. fs.unlink("./index.html", function(err) { 5. if (err) { 6. console.log(err); 7. return; 8. } 9. console.log("删除./index.html文件成功"); 10. }); |

## 1.4 重命名

fs.rename(oldName, newName, callback); 该方法用于将文件或者文件夹重命名

oldName: 原名称

newName: 修改后的名称

callback: 回调函数 有一个错误参数

demo:

|  |
| --- |
| 1. // 引入fs模块 2. var fs = require("fs"); 3. // 调用rename方法重命名 4. fs.rename("a", "b", function(err) { 5. console.log(err); 6. }); |

## 1.5 创建文件夹

fs.mkdir(dirPath, callback); 该方法用于创建新的文件夹

dirPath: 被创建出来的新的文件夹路径

注：一定要确保不重名、路径的最后第二层是存在的

ex: ./a/b/c/d 此时一定要确保 a/b/c是已经存在的

callback: 回调函数 回调函数中有一个错误参数

demo:

|  |
| --- |
| 1. // 异步创建文件夹 2. fs.mkdir("./hello/hello1", function(err) { 3. console.log(err); 4. console.timeEnd(); 5. }); 6. console.time(); |

## 1.6 读取文件夹

fs.readdir(dirPath, callback); 该方法用于读取文件夹中的内容

dirPath: 被读取的文件夹路径

callback: 读取完毕之后的回调函数 有两个参数

第一个参数是表示错误信息

第二个参数是表示读取到的内容 它是一个数组

demo:

|  |
| --- |
| 1. var fs = require("fs"); 2. // 读取 3. fs.readdir("新建文件夹", function(err, arr) { 4. console.log(err); 5. console.log(arr); 6. }); |

## 1.7 判定目标状态

fs.stat(targetPath, callback); 该方法用于获取一个状态对象 该状态对象可以判定是否是文件夹

targetPath: 目标路径

callback: 回调函数 有两个参数

第一个是错误信息

第二个是状态对象

状态对象.isDirectory()方法判定目标是文件夹或者是文件

demo:

|  |
| --- |
| 1. // 引入fs模块 用于判定目标的状态 2. var fs = require("fs"); 3. // 判定a.txt是什么 4. fs.stat("./a.txt", function(err, statObj) { 5. console.log("a.txt是一个" + (statObj.isDirectory() ? "文件夹" : "文件")); 6. }); 7. fs.stat("./aaa", function(err, statObj) { 8. console.log("aaa是一个" + (statObj.isDirectory() ? "文件夹" : "文件")); 9. }); |

## 1.8 删除文件夹

fs.rmdir(dirPath, callback); 该方法用于删除空文件夹

dirPath: 文件夹的地址

callback: 回调函数 有一个错误参数

demo:

|  |
| --- |
| 1. // 删除新建文件夹1 2. fs.rmdir("新建文件夹 1", function(err) { 3. console.log(err); 4. }); |

# URL模块

该模块专门用于处理URL字符串。

引入方式： var url = require("url");

## 2.1 parse方法

url.parse(str, boolean); 该方法用于将URL字符串转为URL对象

str: URL字符串

boolean: 布尔值 如果该布尔值为true 则将query部分转为对象 默认为false

返回值：url对象

demo:

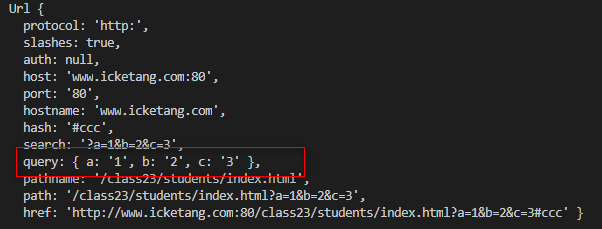
|  |
| --- |
| 1. // 引入url模块 2. var url = require("url"); 3. // 定义url字符串 4. var str = 'http://www.icketang.com:80/class23/students/index.html?a=1&b=2&c=3#ccc'; 5. // 将url字符串转为url对象 6. var obj = url.parse(str); 7. // 输出obj 8. console.log(obj); |

输出：

|  |
| --- |
|  |

注意：上面的对象的每一个部分都是URL的组成部分 query现在为string

此时，我们可以将第二个参数传递： url.parse(str, true);

此时得到的对象：

|  |
| --- |
|  |

注：query已经变为了对象

这样获取get请求提交的数据就变得简单。

## 2.2 format方法

url.format(obj); 该方法用于将URL对象重新转回URL字符串

obj：URL对象

返回值：URL字符串

demo:

|  |
| --- |
| 1. // 引入url模块 2. var url = require("url"); 3. // 定义url字符串 4. var str = 'http://www.icketang.com:80/class23/students/index.html?a=1&b=2&c=3#ccc'; 5. // 将url字符串转为url对象 6. var obj1 = url.parse(str); 7. // 输出obj 8. console.log(obj1); 9. // 修改一下内容 10. // obj1.host = "www.baidu.com"; 11. obj1.search = "?a=8&b=9&c=10"; 12. console.log(obj1); 13. // 将obj1重新转为URL字符串 14. var newStr = url.format(obj1); 15. console.log(newStr); |

最后的输出结果：

|  |
| --- |
|  |

# QueryString模块

该模块专门用于处理query字符串的。或者是类似于query字符串的字符串。

query字符串： a=1&b=2&c=3

类似query字符串的字符串： a#1\*b#2\*c#3

引入方式： var qs = require("querystring");

## 3.1 parse

qs.parse(querystring, bigSplit, smallSplit); 该方法用于将query字符串格式化为对象

querystring: 被格式化的字符串

bigSplit: 大分隔符号 可省略 默认是&

smallSplit: 小分隔符号 可省略 默认是=

demo:

|  |
| --- |
| 1. // 引入querystring模块 2. var qs = require("querystring"); 3. // 定义query字符串 4. var queryString = "a=1&b=2&c=3"; 5. // 定义类似的字符串 6. var likeString = "a#1$b#2$c#3"; 7. // 格式化 8. var obj1 = qs.parse(queryString); 9. console.log(obj1); 10. var obj2 = qs.parse(likeString, "$", "#"); 11. console.log(obj2); |

输出:

|  |
| --- |
|  |

# 下午复习

FS模块：NodeJS中的核心模块之一，只能够在通过node命令调用时，才会有。

创建文件： fs.appendFile(文件路径, 文件内容, 回调函数)

读取文件： fs.readFile(文件路径，回调函数)

删除文件： fs.unlink(文件路径，回调函数)

重命名： fs.rename(旧文件路径，新文件路径， 回调函数)

创建文件夹： fs.mkdir(文件夹路径，回调函数)

读取文件夹： fs.readdir(文件夹路径，回调函数)

判定目标状态：fs.stat(目标路径，回调函数)

删除文件夹：fs.rmdir(文件夹路径，回调函数) 必须是空的

URL模块：NodeJS的核心模块之一，只能够在通过node命令调用时，才会有。

将URL字符串转为URL对象：url.parse(url字符串，是否将query转为对象)

将URL对象转为URL字符串：url.format(url对象) 要修改URL对象的关键属性才会修改

QueryString模块：NodeJS的核心模块之一，只能够在通过node命令调用时，才会有。

将query字符串或者类似的字符串转为对象：

qs.parse(query字符串, 大分隔符，小分隔符)

# 静态服务器

## 4.1 概念介绍

所谓的静态服务器，是“静态资源服务器”的简称。

静态资源：html、css、js、video、audio等资源叫做静态资源。

动态资源：登录功能、注册功能、上传内容、购票、查看实时的购票信息等

## 4.2 MimeType

MimeType指的是每一种资源在互联网中的类型定义。

比如：

html text/html

css text/css

js text/javascript

js application/x-javascript

jpg image/jpeg

jpeg image/jpeg

jpe image/jpeg

……

## 4.3 实现静态服务器

|  |
| --- |
| 1. // 创建服务器对象 2. var server = http.createServer(function(req, res) { 3. // 获取pathname部分 4. var pathname = url.parse(req.url).pathname; 5. // 获取文件的拓展名 6. var extName = pathname.split(".").pop(); 7. fs.readFile("." + pathname, function(err, data) { 8. // 判定是否读取成功 9. if (err) { 10. res.setHeader("content-type", "text/plain;charset=utf-8"); 11. res.end("抱歉，您读取的" + pathname + "资源不存在！404！"); 12. return; 13. } 14. // 指定响应头中的content-type字段 15. res.setHeader("content-type", MimeType[extName] + ";charset=utf-8"); 16. // 给前端返回 17. res.end(data); 18. }); 19. }); |

# 接口

接口就是后台提供的一个URL地址，但是与资源URL不同，接口通常会实现一些功能。

比如：下单、购物车结算、登录、注册、获取物品详情等

## 5.1 获取GET请求数据

获取GET请求的数据要通过URL模块，并传递第二个参数，获取解析对象的query属性。

demo:

|  |
| --- |
| 1. var urlObj = url.parse(req.url, true); 2. var query = urlObj.query; |

输出：

|  |
| --- |
|  |

## 5.2 获取POST请求数据

因为POST请求的携带的数据容量可能很多，所以很可能无法一次性传递完毕。所以需要多次传递。

NodeJS处理POST请求的数据的方式：data事件和end事件

绑定data事件：该事件视数据量的多少，决定触发次数

|  |
| --- |
| 1. req.on("data", function(data\_chunk) { 2. data\_chunk就是本次提交过来的数据块 3. }); |

绑定end事件：当所有的数据块传递完毕之后，触发的事件

|  |
| --- |
| 1. req.on("end", function() { 2. 本事件会在所有的数据传递完毕之后触发。也就是最后一次data事件完毕之后。 3. }); |

demo:

|  |
| --- |
| 1. // 定义变量 用于接收数据 2. var data = ""; 3. // console.log("前端发送post请求注册"); 4. // 绑定data事件 5. req.on("data", function(data\_chunk) { 6. data += data\_chunk; 7. }); 8. // 绑定end事件 9. req.on("end", function() { 10. // 将query字符串格式化成query对象方便获取信息 11. var obj = qs.parse(data); 12. console.log(obj); 13. // 获取用户名 14. var username = obj.username; 15. // 获取密码 16. var password = obj.password; 17. // 拿着username去arr中判定是否已经存在 18. var result = arr.some(function(value) { 19. return value.username === username; 20. }); 21. // 判定result 22. if (result) { 23. // 说明已经存在 24. res.end("抱歉，用户名已经被占用"); 25. return; 26. } 27. // 将数据存储到数据库中 28. arr.push({ 29. username: username, 30. password: password 31. }); 32. // 说明没有被占用 33. res.end("恭喜，可以使用"); 34. }); |

# AJAX

很久之前，浏览器只是一个页面展示的容器。

后来随着web2.0出现。基于网页的应用的需求慢慢的出现。于是人们大量的使用表单。进行数据交互。

但是使用表单有一个特点：会跳转页面。

在某些增加一条数据、点击某一个按钮时，只需要更新一点点内容，但是技术上也需要刷新整个页面。

于是就出现了AJAX。

全称：Asynchronous JavaScript And XML 异步的JS和XML

注：XML是可拓展标记语言，在以前的时候用于前后端之间传递数据。现在基本只在后端使用。

但是XML有：前端解析麻烦、后端生成也麻烦。所以渐渐地被JSON代替。

AJAX不是一门新出现的技术。最早可以追溯到IE5。

总体流程： 通过JS发送HTTP请求，到达服务器，请求数据，带回数据，JS处理，提示、上树

## 6.1 体验AJAX

网易163：验证用户邮箱

|  |
| --- |
|  |

花瓣：瀑布流

|  |
| --- |
|  |

## 6.2 jQuery中的AJAX

发送GET请求：

$.get(url, data, callback, dataType);  
 url: 本次请求发送的URL地址字符串

data: 本次请求携带的数据 它们最终会被格式化为query字符串挂载在url的query部分 可以省略

可以是query字符串

可以是对象

callback: 本次请求结束之后的回调函数

有一个参数，是返回的JSON数据

dataType: 返回的数据的类型 可以是默认的字符串 也可以是json

发送POST请求：

$.post(url, data, callback, dataType);

url: 本次请求发送的URL地址字符串

data: 本次请求携带的数据 它们最终会被格式化为query字符串挂载在请求正文中 可以省略

可以是query字符串

可以是对象

callback: 本次请求结束之后的回调函数

有一个参数，是返回的JSON数据

dataType: 返回的数据的类型 可以是默认的字符串 也可以是json

该方法用于负责发送任何请求

$.ajax(option);

option: 配置对象

url: 本次请求发送的URL地址字符串

data: 本次请求携带的数据 它们最终会被格式化为query字符串

type: 本次请求的类型 get、post

dataType: 返回的数据的类型 可以是默认的字符串 也可以是json

success: 成功时的回调函数

error: 失败时的回调函数

complete: 完成时的回调函数 不论成功失败

processData: 布尔值配置 是否由jQuery来序列化数据

contentType: 布尔值配置 是否由jQuery来决定请求头的content-type字段

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |