█

使用http模块建立一个服务

const http = require('http')

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

console.log(req.url)

if (req.url == '/aa') {

res.write('aaa')

} else if (req.url == '/bb') {

res.write('bbb')

} else {

res.write('404')

}

// switch(req.url) {

// case '/aa':

// res.write('aaa')

// break

// case '/bb':

// res.write('bbb')

// break

// default:

// res.write('404')

// break

// }

res.end()

})

server.listen(8080)

█使用fs模块来管理文件

●基本用法

const fs = require('fs')

// 读取文件

fs.readFile('a.txt', function (err, data) {

if (err) {

console.log('读取失败')

} else {

console.log(data)

// 输出：<Buffer 31 32 33 34 35 36 37 0d 0a 68 61 68 61>

console.log(data.toString())

// 输出：

// 1234567

// haha

}

})

// 写文件

fs.writeFile('b.txt', 'haha123', function (err) {

console.log(err)

})

// 如没有b.txt则自动生成文件b.txt然后将第二个参数'haha123'作为内容写入b.txt文件

●在服务中应用fs模块

const http = require('http')

const fs = require('fs')

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

var file\_name = './www' + req.url

// 例如客户端输入localhost:8080/a.html 则会读取www目录下的a.html文件

fs.readFile(file\_name, function (err, data) {

if (err) {

res.write('404')

} else {

res.write(data)

}

res.end()

})

})

server.listen(8080)

█处理用户请求

●get

发请求的页面

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<form action="http://localhost:8087" method="get">

用户：<input type="text" name="user" value=""><br/>

密码：<input type="text" name="pass" value=""><br/>

<input type="submit" value="提交">

</form>

</body>

</html>

使用querystring模块来序列号get请求

const http = require('http')

const querystring = require('querystring')

http.createServer(function (req, res) {

// console.log(req.url)

// 输出：

// /?user=aa&pass=11

// /favicon.ico

// 由于有favicon.ico存在，所以要加一个if判断，有？的再执行下面操作

if (req.url.indexOf('?')!=-1) {

var arr = req.url.split('?')

var url = arr[0]

// 使用querystring模块将/?user=aa&pass=11序列化成一个json对象

var get = querystring.parse(arr[1])

} else {

var url = req.url

}

console.log(url, get)

// 输出：

// / { user: 'bbb', pass: '222' }

// /favicon.ico undefined

res.write('aaa')

res.end()

}).listen(8087)

使用url模块来序列化get请求

const http = require('http')

const urlLib = require('url')

http.createServer(function (req, res) {

// console.log(req.url)

// 输出：

// /?user=aa&pass=11

// /favicon.ico

// 第二个参数true是将pathname变成对象，没有true则默认pathname是一个字符串

var obj = urlLib.parse(req.url, true)

var url = obj.pathname

var get = obj.query

console.log(url, get)

// 输出：

// / { user: 'cc', pass: '11' }

// /favicon.ico {}

res.write('aaa')

res.end()

}).listen(8087)

●post

发请求的页面

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<form action="http://localhost:8087" method="post">

用户：<input type="text" name="user" value=""><br/>

密码：<input type="text" name="pass" value=""><br/>

<input type="submit" value="提交">

</form>

</body>

</html>

使用querystring来序列号post

const http = require('http')

const querystring = require('querystring')

http.createServer(function (req, res) {

var str = ''

req.on('data', function (data) {

// 如果post数据很大，会分多次发送，所以每次触发发送事件，都把str += 发送的data

str += data

})

req.on('end', function () {

// 数据都发送完毕后，会触发end事件

console.log(str)

// 输出：user=%E9%98%BF%E5%8F%91&pass=1212

var aa = querystring.parse(str)

console.log(aa)

// 输出：{ user: '阿发', pass: '1212' }

})

}).listen(8081)

█express

●基本用法

const express = require('express')

var server = express()

// 类似.get和.post,而.use可以接受get和post请求

server.use('/', function (req, res) {

// send和原生的write类似，但是write只能发送string或者buffer对象，而send发送的类型则比write要多

res.send('abc')

res.end()

})

server.listen(8081)

●express-static中间件

const express = require('express')

const expressStatic = require('express-static')

var server = express()

server.listen(8082)

// 这样，可以通过url来直接访问www文件夹里的静态文件

// 例如localhost:8082/index.html 则会访问www文件夹里的index.html文件

server.use(expressStatic('./www'))

◎设置访问端口时直接响应www目录中的index.html文件

const express = require('express')

var server = express()

server.use(express.static('./www', {index:"index.html"}))

server.listen(80)

●express应用案例

前端页面 www/index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<h1>Hello World!</h1>

用户：<input type="text" id="user"><br/>

密码：<input type="password" id="pass"><br/>

<input type="button" value="登陆" id="btn1">

<input type="button" value="注册" id="btn2">

<script src="./js/ajax.js"></script>

<script>

window.onload = function () {

var oUser = document.getElementById('user')

var oPass = document.getElementById('pass')

var oBtn = document.getElementById('btn1')

var oBtn2 = document.getElementById('btn2')

oBtn.onclick = function () {

ajax({

url: '/login',

data: {user: oUser.value, pass: oPass.value},

success: function (str) {

console.log(str)

// 这里str是字符串，需要将其转化成json对象

// var json = eval('(' + str + ')')

var json = JSON.parse(str)

console.log(json)

if (json.ok) {

alert('登陆成功')

} else {

alert('登陆失败:' + json.msg)

}

},

error: function () {

alert('网络故障')

}

})

}

oBtn2.onclick = function () {

ajax({

url: '/sigin',

data: {user: oUser.value, pass: oPass.value},

success: function (str) {

console.log(str)

// 这里str是字符串，需要将其转化成json对象

// var json = eval('(' + str + ')')

var json = JSON.parse(str)

console.log(json)

if (json.ok) {

alert('注册成功')

} else {

alert('注册失败：' + json.msg)

}

}

})

}

}

</script>

</body>

</html>

后台 et1.js

const express = require('express')

const expressStatic = require('express-static')

var server = express()

server.listen(8089)

// 模仿数据库来存放用户数据，这里用一个json对象来存

var users = {

'tom': '123456',

'jerry': '777777',

'jojo': '12321'

}

// 登陆接口

server.get('/login', function (req, res) {

// req.query会把url里的的参数返回成一个json对象

// 例如在浏览器输入localhost:8085/login?a=1&b=7 后台则会输出如下结果

// 输出：{ a: '1', b: '7' }

console.log(req.query)

var user = req.query['user']

var pass = req.query['pass']

if (users[user] == null) {

res.send({ok:false, msg:'此用户不存在'})

} else {

if (users[user] != pass) {

res.send({ok:false, msg:'密码错误'})

} else {

res.send({ok:true, msg:'登陆成功'})

}

}

})

// 注册接口

server.get('/sigin', function (req, res) {

var user = req.query['user']

var pass = req.query['pass']

if (users[user]) {

res.send({ok:false, msg:'此用户已经存在'})

} else {

users[user] = pass

res.send({ok:true, msg:'注册成功'})

}

})

// 这样，可以通过url来直接访问www文件夹里的静态文件

// 例如localhost:8082/index.html 则会访问www文件夹里的index.html文件

server.use(expressStatic('./www'))

使用

命令行输入

node et1.js

启动服务

然后用浏览器 localhost:8089/index.html

●处理get和post请求

请求的页面

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<form action="http://localhost:8081/pc" method="post">

姓名：<input type="text" name="name" value=""><br/>

年龄：<input type="text" name="age" value=""><br/>

<input type="submit" value="提交">

</form>

</body>

</html>

后台页面

var express = require('express')

var bodyParser = require('body-parser')

var server = express()

server.listen(8081)

// 使用body-parser中间件，需要先设置一下，然后才使用req.body

server.use(bodyParser.urlencoded({

// 扩展模式

extended: false,

// 限制post大小，例如2M

limit: 2 \* 1024 \* 1024

}))

server.get('/gc', function (req, res) {

// 无需中间件，req.query会自动将get请求返回为json对象

// 如在浏览器键入url http://localhost:8080/gc?a=1&b=2&c=7

// 后台会输出 { a: '1', b: '2', c: '7' }

console.log(req.query)

})

server.post('/pc', function (req, res) {

// 使用body-parser中间件后，req.body会将post请求转化为json对象

// 例如{ name: 'bb', age: '22' }

console.log(req.body)

})

总结

GET-无需中间件

req.query

POST-需要"body-parser"中间件

server.use(bodyParser.urlencoded({}));

server.use(function (){

req.body

});

●链式操作

const express=require('express');

const bodyParser=require('body-parser');

var server=express();

server.listen(8080);

server.use('/', function (req, res, next){

console.log('a');

// 如果不写next()则浏览器访问localhost:8080时后台只会打印a，而写了next()才会执行下一个打印b的指令

next();

});

server.use('/', function (req, res, next){

console.log('b');

});

█模块化

●

m1.js

exports.a = 111

exports.b = function (n) {

console.log(n)

}

mz.js

var m1 = require('./m1')

console.log(m1.a)

// 输出111

console.log(m1.b(222))

// 输出222

●以上可以使用module.exports来暴露多个对象

m1.js使用module.exports暴露多个对象

module.exports = {

'a': 111,

'b': function (n) {

console.log(n)

}

}

mz.js

var m1 = require('./m1')

console.log(m1.a)

// 输出111

console.log(m1.b(222))

// 输出222

●小案例，通过模块化将所有接口分离到一个模块m1.js中

接口文件 m1.js

const express=require('express');

var server=express();

server.listen(8088);

module.exports = {

'a': function () {

server.get('/b1', function (req, res) {

console.log(req.query)

})

},

'b': function () {

server.get('/b2', function (req, res) {

console.log(req.query)

})

}

}

主程序mz.js

var m1 = require('./m1')

m1.a()

m1.b()

运行node mz.js后

可通过浏览器输入url

<http://localhost:8088/b1?a=11&b=52>

或

<http://localhost:8088/b2?a=11&b=52>

来测试b1和b2两个接口

●上传文件

◎案例1 ：Form标签的enctype属性为application/x-www-form-urlencoded时

利用body-parser中间件只能解析post请求的文本，而不能解析post过来的文件如视频，音频等

前端页面

Form标签的enctype属性为application/x-www-form-urlencoded时，上传的文件只是文件名，可以用body-parser来解析

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<form action="http://localhost:8082/pf" method="post" enctype="application/x-www-form-urlencoded">

文件<input type="file" name="f1">

<input type="submit" value="提交">

</form>

</body>

</html>

后台

var express = require('express')

const bodyParser=require('body-parser');

var server = express()

server.listen(8082)

server.use(bodyParser.urlencoded({

// 扩展模式

extended: false,

// 限制post大小，例如2M

limit: 2 \* 1024 \* 1024

}))

server.use('/pf', function (req, res) {

console.log(req.body)

// 打印：{ f1: 'cs.txt' }

})

◎一个上传文件的例子

前端页面，要上传文件，需要将form的enctype属性设置成multipart/form-data

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

</head>

<body>

<form action="http://localhost:8083/pf" method="post" enctype="multipart/form-data">

文件<input type="file" name="f1">

<input type="submit" value="提交">

</form>

</body>

</html>

后端

这里使用multer中间件来代替body-parser来接收post请求，因为multer可以接受文件上传，而body-parser只能接受post的字符串

这里还使用path中间件来获取文件的扩展名

const express = require('express')

const bodyParser=require('body-parser');

const multer = require('multer')

const fs = require('fs')

const pathLib = require('path')

// 设置multer中间件，设置接受上传文件的路径

var objMulter = multer({dest: './www/upload/'})

var server = express()

server.use(bodyParser.urlencoded({

extended: false,

limit: 2 \* 1024 \* 1024

}))

server.use(objMulter.any());

server.post('/pf', function (req, res){

// req.file是上传文件的具体信息

// [ { fieldname: 'f1',

// originalname: 'cs.txt',

// encoding: '7bit',

// mimetype: 'text/plain',

// destination: './www/upload/',

// filename: '5a0a97f6edb843c7c4b76182135fc8fd',

// path: 'www\\upload\\5a0a97f6edb843c7c4b76182135fc8fd',

// size: 30 } ]

console.log(req.files)

//新文件名

//'./www/upload/dfb33662df86c75cf4ea8197f9d419f9' + '.png'

console.log(pathLib.parse(req.files[0].originalname))

// 输出：{ root: '', dir: '', base: 'cs.txt', ext: '.txt', name: 'cs' }

var newName=req.files[0].path+pathLib.parse(req.files[0].originalname).ext;

fs.rename(req.files[0].path, newName, function (err){

if(err)

res.send('上传失败');

else

res.send('成功');

});

//1.获取原始文件扩展名

//2.重命名临时文件

});

server.listen(8083)

●路由

Router是express内置的，无需安装

const express = require('express')

var server = express()

//目录1：/user/

var routeUser=express.Router();

server.use('/user', routeUser);

routeUser.get('/1.html', function (req, res){ //http://xxx.com/user/1.html

res.send('user1');

});

routeUser.get('/2.html', function (req, res){ //http://xxx.com/user/2.html

res.send('user22222');

});

//目录2：/article/

var articleRouter=express.Router();

server.use('/article', articleRouter);

articleRouter.get('/10001.html', function (req, res){ //http://xxxx.com/article/10001.html

res.send('asdfasdfasdf');

});

server.listen(8085)

█数据库

●mysql基础

Npm install mysql可以安装node的mysql模块，这是一个客户端模块，用来向数据库服务端发起请求

一个简单例子

const mysql = require('mysql')

// 建立连接

// var db=mysql.createConnection(host, port, user, password, database)

// 这里port是默认端口3306，所以可以省略

var db = mysql.createConnection({host: 'localhost', user: 'root', password: 'qaz123456', database: 'mytest'})

// 查询

// db.query(SQL, (err, data)=>{})

db.query('SELECT \* FROM `user\_table`', (err, data) => {

if (err)

console.log('出错了', err)

else

console.log('成功')

console.log(JSON.stringify(data))

})

◎4大查询语句——增删改查

增-INSERT

INSERT INTO 表 (字段列表) VALUES(值列表)

INSERT INTO `user\_table` (`ID`, `username`, `password`) VALUES(0, 'blue2', '987654');

删-DELETE

改-UPDATE

查-SELECT

SELECT 什么 FROM 表

SELECT \* FROM `user\_table`;

◎SQL标准写法：

1.关键字大写

2.库、表、字段需要加上` （就是数字1键左边的那个键）

●mysql基本用法

◎四大操作语句-

1.删 DELETE

DELETE FROM 表 WHERE 条件

2.增 INSERT

INSERT INTO 表 (字段列表) VALUES(值列表)

INSERT INTO `user\_table` (`ID`, `username`, `password`) VALUES(0, 'blue2', '987654');

注：ID设为自动递增，所以值为0时，自动加到最后一条数据之后

3.改 UPDATE

UPDATE 表 SET 字段=值,字段=值,... WHERE 条件

4.查 SELECT

SELECT \* FROM 表 WHERE 条件

●子句

◎ WHERE 条件

WHERE name='blue'

WHERE age>18

WHERE age<=18

WHERE age>=18 AND score<60

WHERE cach>100 OR score>10000

* ORDER 排序

ORDER BY age ASC/DESC

ASC-升序(从小到大)

DESC-降序(从大到小)

◎ ORDER BY price ASC

\*价格(price)升序排序，如果价格相同，再按销量(sales)降序排序

ORDER BY price ASC, sales DESC

* GROUP 聚类 合并相同
* LIMIT 限制输出条数，一般用于分页

LIMIT 10; 前10条

LIMIT 5,8; 从5开始，要8个

分页：

例如每页20条

第1页 0,20 0~19

第2页 20,20 20~39

第3页 40,20

第n页 (n-1)\*20,20

●子句的顺序

子句之间是有顺序

WHERE GROUP ORDER LIMIT

筛选 合并 排序 限制

例如

SELECT class, COUNT(class) FROM students\_table WHERE mark>80 GROUP BY class ORDER BY COUNT(class) DESC;

|  |  |
| --- | --- |
| Class | COUNT(class) |
| 1 | 2 |
| 3 | 2 |
| 2 | 1 |

●函数

COUNT

MIN

MAX

AVG平均值

◎案例练习

数据库

|  |  |  |  |
| --- | --- | --- | --- |
| ID | class | name | mark |
| 1 | 1 | 小红 | 99 |
| 2 | 2 | 小强 | 76 |
| 3 | 3 | 小文 | 88 |
| 4 | 2 | 小明 | 85 |
| 5 | 2 | 小兰 | 66 |
| 6 | 1 | 小土 | 92 |
| 7 | 3 | 小张 | 87 |

使用COUNT来统计表有多少条记录

SELECT COUNT(\*) FROM students\_table;

|  |
| --- |
| COUNT(\*) |
| 7 |

列出班级和班级人数的统计

SELECT class, COUNT(class) FROM students\_table GROUP BY class;

|  |  |
| --- | --- |
| class | COUNT(class) |
| 1 | 2 |
| 2 | 3 |
| 3 | 2 |

列出班级和每个班的平均分

SELECT class, AVG(mark) FROM students\_table GROUP BY class;

|  |  |
| --- | --- |
| Class | AVG(mark) |
| 1 | 54.25 |
| 2 | 98.5 |
| 3 | 56.5 |

每个班级的最高和最低分

SELECT class, MAX(mark), MIN(mark) FROM students\_table GROUP BY class;

|  |  |  |
| --- | --- | --- |
| Class | MAX(mark) | MIN(mark) |
| 1 | 99 | 92 |
| 2 | 85 | 66 |
| 3 | 88 | 87 |

每个班级的分数之和

SELECT class, SUM(mark) FROM students\_table GROUP BY class;

|  |  |
| --- | --- |
| Class | SUM(mark) |
| 1 | 191 |
| 2 | 227 |
| 3 | 175 |

统计每个班级的分数之和并排序

SELECT class, SUM(mark) FROM students\_table GROUP BY class ORDER BY SUM(mark) DESC;

|  |  |
| --- | --- |
| Class | SUM(mark) |
| 2 | 227 |
| 1 | 191 |
| 3 | 175 |