# Node

## 初识Node

### 概念

为什么JavaScript可以操作DOM和BOM?

每个浏览器都内置了DOM、BOM这样的API函数，因此，浏览器中的JavaScript才可以调用它们。

运行环境是指代码正常运行所需的必要环境。

步骤一：内置API，DOM、BOM、Canvas、XMLHttpRequest、JS内置对象、etc...

步骤二：待执行的JavaScript代码

步骤三：Chrome浏览器运行环境，V8引擎，V8引擎负责解析和执行JavaScript代码。

内置API是由运行环境提供的特殊接口，只能在所属的运行环境中被调用。

### 什么是Node.js

Node.js是一个基于ChromeV8引擎的JavaScript运行环境。

官网：<https://nodejs.org/zh-cn/>

浏览器是JavaScript的前端运行环境

Node.js是JavaScript的后端运行环境

Node.js中无法调用DOM和BOM等浏览器内置API

Node.js运行环境

步骤一：内置API，fs、path、http、querystring、JS内置对象、etc...

步骤二：待执行的JavaScript代码

步骤三：Chrome浏览器运行环境，V8引擎，V8引擎负责解析和执行JavaScript代码。

Node.js可以做什么

1. 基于Epress框架(http://www.expressjs.com.cn/)，可以快速构建Web应用
2. 基于Electron框架(https://electronjs.org/),可以构建跨平台的桌面应用
3. 基于restify框架(http://restify.com/),可以快速构建API接口项目
4. 读写和操作数据库、创建实用的命令行工具辅助前端开发、etc...

Node.js的学习路径：

JavaScript基础语法+Node.js内置API模块(fs、path、http等)+第三方API模块(express、mysql等)

### 安装Node.js

官网：[https://nodejs.org/en/](https://nodejs.org/zh-cn/)

LTS为长期稳定版，对于追求稳定性的企业级项目来说，推荐安装LTS版本的Node.js。

Current为新特性尝鲜版，对于热衷于尝试新特性的用户来说，推荐安装Current版本的Node.js。但是，Current版本可能存在隐藏的Bug或安全性漏洞，因此不推荐在企业级项目中使用Current版本的Node.js。

打开终端，cmd中输入命令node -v

### 使用Node运行JS代码

终端(terminal)是专门为开发人员设计的，用于实现人机交互的一种方式。

在Node.js环境中执行JavaScript代码

方式一：

1. 打开终端
2. 输入node要执行的js文件的路径

切换磁盘d: 切换到d盘

cd到要执行的js文件的路径

方式二：

直接在js文件路径上cmd

node 123456.js

powershell终端也可以

终端中的快捷键

1. 使用↑，可以快速定位到上一次执行的命令
2. 使用tab键，能够快速补全路径
3. 使用esc键，能够快速清空当前已输入的命令
4. 输入cls命令，可以清空终端

## fs

### 读取文件内容

fs模块是Node.js官方提供的、用来操作文件的模块。它提供了一系列的方法和属性，用来满足用户对文件的操作需求。

fs.readFile()方法，用来读取指定文件中的内容

fs.writeFile()方法，用来向指定的文件中写入内容

const fs=require(‘fs’)

使用fs.readFile()方法，可以读取指定文件中的内容，语法格式如下：

fs.readFile(path[,options],callback)

参数1：必选参数，字符串，表示文件的路径

参数2：可选参数，表示以什么编码格式来读取文件

参数3：必选参数，文件读取完成后，通过回调函数拿到读取的结果

创建1.txt文件

// 1. 导入 fs 模块，来操作文件

const *fs* = require('fs')

// 2. 调用 fs.readFile() 方法读取文件

//    参数1：读取文件的存放路径

//    参数2：读取文件时候采用的编码格式，一般默认指定 utf8

//    参数3：回调函数，拿到读取失败和成功的结果  err  dataStr

*fs*.readFile('./files/11.txt', 'utf8', function(*err*, *dataStr*) {

  // 2.1 打印失败的结果

  // 如果读取成功，则 err 的值为 null

  // 如果读取失败，则 err 的值为 错误对象，dataStr 的值为 undefined

  console.log(*err*)

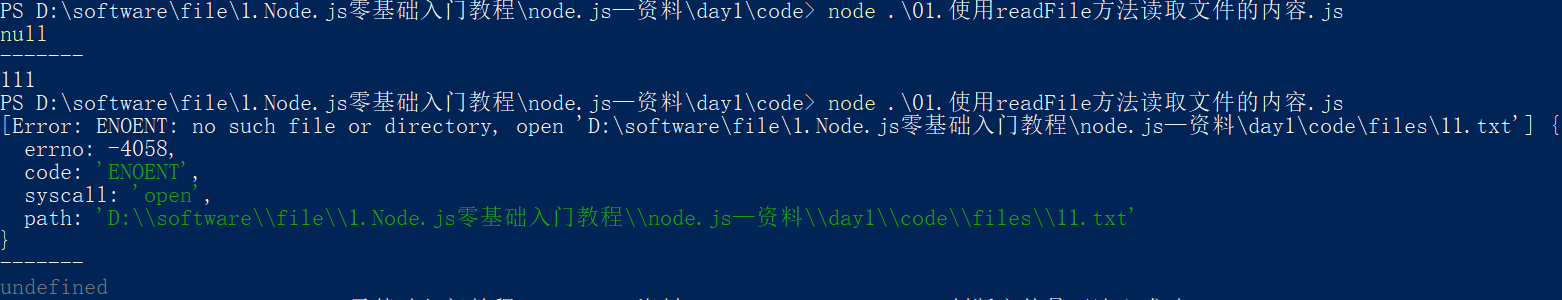
  console.log('-------')

  // 2.2 打印成功的结果

  console.log(*dataStr*)

})

运行截图



判断文件是否读取成功

可以判断err对象是否为null，从而知晓文件读取的结果：

const *fs* = require('fs')

*fs*.readFile('./files/11.txt', 'utf8', function(*err*, *dataStr*) {

  if (*err*) {

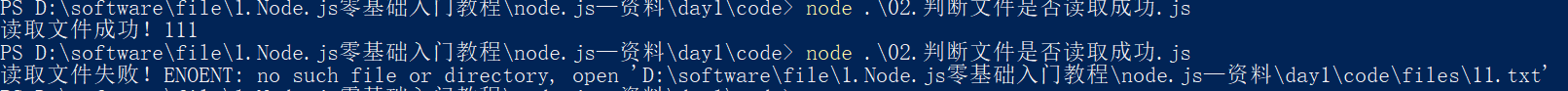
    return console.log('读取文件失败！' + *err*.message)

  }

  console.log('读取文件成功！' + *dataStr*)

})

运行截图



null的布尔值为false

### 写入文件内容

向指定的文件中写入内容

使用fs.writeFile()方法，可以向指定的文件中写入内容，语法格式如下：

fs.writeFile(file,data[,options],callback)

参数1：必选参数，需要指定一个文件路径的字符串，表示文件的存放路径

参数2：必选参数，表示要写入的内容

参数3：可选参数，表示以什么格式写入文件内容，默认值是utf8

参数4：必选参数，文件写入完成后的回调函数

// 1. 导入 fs 文件系统模块

const *fs* = require('fs')

// 2. 调用 fs.writeFile() 方法，写入文件的内容

//    参数1：表示文件的存放路径

//    参数2：表示要写入的内容

//    参数3：回调函数

*fs*.writeFile('./files/3.txt', 'ok123', function(*err*) {

  // 2.1 如果文件写入成功，则 err 的值等于 null

  // 2.2 如果文件写入失败，则 err 的值等于一个 错误对象

  // console.log(err)

  if (*err*) {

    return console.log('文件写入失败！' + *err*.message)

  }

  console.log('文件写入成功！')

})

写入成功后会自动创建一个3.txt文件。覆盖原来的文件内容

判断文件是否写入成功

### 整理成绩的案例

整理之前

成绩.txt

小红=99 小白=100 小黄=70 小黑=66 小绿=88

整理之后

成绩-ok.txt

小红：99

小白：100

小黄：70

小黑：66

小绿：88

核心实现步骤

1. 导入需要的fs文件系统模块
2. 使用fs.readFile()方法，读取素材目录下的成绩.txt文件
3. 判断文件是否读取失败
4. 文件读取成功后，处理成绩数据
5. 将处理完成的成绩数据，调用fs.writeFile()方法，写入到新文件成绩-ok.txt中

// 1. 导入 fs 模块

const *fs* = require('fs')

// 2. 调用 fs.readFile() 读取文件的内容

*fs*.readFile('../素材/成绩.txt', 'utf8', function(*err*, *dataStr*) {

  // 3. 判断是否读取成功

  if (*err*) {

    return console.log('读取文件失败！' + *err*.message)

  }

  // console.log('读取文件成功！' + dataStr)

  // 4.1 先把成绩的数据，按照空格进行分割

  const arrOld = *dataStr*.split(' ')

  // 4.2 循环分割后的数组，对每一项数据，进行字符串的替换操作

  const arrNew = []

  arrOld.forEach(*item* => {

    arrNew.push(*item*.replace('=', '：'))

  })

  // 4.3 把新数组中的每一项，进行合并，得到一个新的字符串

  const newStr = arrNew.join('\r\n')

  // 5. 调用 fs.writeFile() 方法，把处理完毕的成绩，写入到新文件中

*fs*.writeFile('./files/成绩-ok.txt', newStr, function(*err*) {

    if (*err*) {

      return console.log('写入文件失败！' + *err*.message)

    }

    console.log('成绩写入成功！')

  })

})

\r\n回车换行

### 路径动态拼接的问题

在使用fs模块操作文件时，如果提供的操作路径是以./或.//开头的相对路径时，很容易出现路径动态拼接错误的问题。

原因：代码在运行的时候，会以执行node命令时所处的目录，动态拼接出被操作文件的完整路径。

不在day1\code目录下出现的问题，会在day1目录下拼接files\1.txt

const *fs* = require('fs')

// 出现路径拼接错误的问题，是因为提供了 ./ 或 ../ 开头的相对路径

// 如果要解决这个问题，可以直接提供一个完整的文件存放路径就行

// 移植性非常差、不利于维护

// \_\_dirname 表示当前文件所处的目录

*fs*.readFile(\_\_dirname + '/files/1.txt', 'utf8', function(*err*, *dataStr*) {

  if (*err*) {

    return console.log('读取文件失败！' + *err*.message)

  }

  console.log('读取文件成功！' + *dataStr*)

})

一个\代表转义，两个\代表斜杠

D:\\software\\file\\1.Node.js零基础入门教程\\node.js—资料\\day1\\code\\files\\1.txt 绝对路径



## path

### 使用path模块处理路径

path模块是Node.js官方提供的、用来处理路径的模块。它提供了一系列的方法和属性，用来满足用户对路径的处理需求。

path.join()方法，用来将多个路径片段拼接成一个完整的路径字符串。

path.basename()方法，用来从路径字符串中，将文件名解析出来。

const path=require(‘path’)

path.join([...paths])

...paths<string>路径片段的序列

返回值:<string>

const path = require('path')

const *fs* = require('fs')

// 注意：  ../ 会抵消前面的路径

// const pathStr = path.join('/a', '/b/c', '../../', './d', 'e')

// console.log(pathStr)  // \a\b\d\e

// fs.readFile(\_\_dirname + '/files/1.txt')

*fs*.readFile(path.join(\_\_dirname, '/files/1.txt'), 'utf8', function(*err*, *dataStr*) {

  if (*err*) {

    return console.log(*err*.message)

  }

  console.log(*dataStr*)

})

注意：今后凡是涉及到路径拼接的操作，都要使用path.join()方法进行处理。不要直接使用+进行字符串的拼接。

path.join可以解析./files/1.txt中的.

### 获取路径中的文件名

使用path.basename()方法，可以获取路径中的最后一部分，经常通过这个方法获取路径中的文件名，语法格式如下：

path.basename(path[,ext])

path<string>必选参数，表示一个路径的字符串

ext<string>可选参数，表示文件扩展名

返回:<string>表示路径中的最后一部分

const path = require('path')

// 定义文件的存放路径

const fpath = '/a/b/c/index.html'

// const fullName = path.basename(fpath)

// console.log(fullName)//输出index.html

const nameWithoutExt = path.basename(fpath, '.html')

console.log(nameWithoutExt)//输出index

### 获取路径中的文件扩展名

使用path.extname()方法，可以获取路径中的扩展名部分，语法格式如下：

path.extname(path)

path<string>必选参数，表示一个路径的字符串

const path = require('path')

// 这是文件的存放路径

const fpath = '/a/b/c/index.html'

const fext = path.extname(fpath)

console.log(fext)//输出.html

### 时钟案例

将素材目录下的index.html页面拆分成三个文件，分别是index.html、index.js、index.css，并且将拆分出来的3个文件存放到clock目录中

案例的实现步骤

1. 创建两个正则表达式，分别用来匹配<style>和<script>标签
2. 使用fs模块，读取需要被处理的HTML文件
3. 自定义resolveCSS方法，来写入index.css样式文件
4. 自定义resolveJS方法，来写入index.js脚本文件
5. 自定义resolveHTML方法，来写入index.html文件

// 1.1 导入 fs 模块

const fs = require('fs')

// 1.2 导入 path 模块

const path = require('path')

// 1.3 定义正则表达式，分别匹配 <style></style> 和 <script></script> 标签

//其中\s表示空白字符;\S表示非空白字符;\*表示匹配任意次

const regStyle = /<style>[\s\S]\*<\/style>/

const regScript = /<script>[\s\S]\*<\/script>/

 // 2.3 读取文件成功后，调用对应的三个方法，分别拆解出 css, js, html 文件

  resolveCSS(*dataStr*)

  resolveJS(*dataStr*)

  resolveHTML(*dataStr*)

// 4.1 定义处理 js 脚本的方法

function resolveJS(*htmlStr*) {

  // 4.2 通过正则，提取对应的 <script></script> 标签内容

  const r2 = regScript.exec(*htmlStr*)

  // 4.3 将提取出来的内容，做进一步的处理

  const newJS = r2[0].replace('<script>', '').replace('</script>', '')

  // 4.4 将处理的结果，写入到 clock 目录中的 index.js 文件里面

*fs*.writeFile(path.join(\_\_dirname, './clock/index.js'), newJS, function(*err*) {

    if (*err*) return console.log('写入 JavaScript 脚本失败！' + *err*.message)

    console.log('写入 JS 脚本成功！')

  })

}

// 5.1 定义处理 HTML 结构的方法

function resolveHTML(*htmlStr*) {

  // 5.2 将字符串调用 replace 方法，把内嵌的 style 和 script 标签，替换为外联的 link 和 script 标签

  const newHTML = *htmlStr*.replace(regStyle, '<link rel="stylesheet" href="./index.css" />').replace(regScript, '<script src="./index.js"></script>')

  // 5.3 写入 index.html 这个文件

*fs*.writeFile(path.join(\_\_dirname, './clock/index.html'), newHTML, function(*err*) {

    if (*err*) return console.log('写入 HTML 文件失败！' + *err*.message)

    console.log('写入 HTML 页面成功！')

  })

}

1. fs.writeFile()方法只能用来创建文件，不能用来创建路径
2. 重复调用fs.writeFile()写入同一个文件，新写入的内容会覆盖之前的旧内容

## http

### 服务器相关的概念

在网络节点中，负责消费资源的电脑，叫做客户端；负责对外提供网络资源的电脑，叫做服务器。

http模块是Node.js官方提供的、用来创建web服务器的模块。通过http模块提供的http.createServer()方法，就可以方便的把一台普通的电脑，变成一台Web服务器，从而对外提供Web资源服务。

const http=require(‘http’)

web服务器软件：IIS、Apache

IP地址就是互联网上每台计算机的唯一地址，因此IP地址具有唯一性。

IP地址和域名是一一对应的关系，这份对应关系存放在一种叫做域名服务器(DNS,Domain name server)的电脑中。域名服务器就是提供IP地址和域名之间的转换服务的服务器。

每个web服务都对应一个唯一的端口号。

每个端口号不能同时被多个web服务占用。

在实际应用中，URL中的80端口可以被忽略(默认端口)。

### 创建最基本的web服务器

基本步骤

1. 导入http模块
2. 创建web服务器实例
3. 为服务器实例绑定request事件，监听客户端的请求
4. 启动服务器

// 1. 导入 http 模块

const http = require('http')

// 2. 创建 web 服务器实例

const server = http.createServer()

// 3. 为服务器实例绑定 request 事件，监听客户端发送过来的网络请求

server.on('request', function (*req*, *res*) {

//只要有客户端来请求我们自己的服务器，就会触发request事件，从而调用这个事件处理函数

  console.log('Someone visit our web server.')

})

// 4. 启动服务器，即可启动当前的web服务器实例

server.listen(8080, function () {

  console.log('server running at http://127.0.0.1:8080')

})

Ctrl+c停止终端

req请求对象：如果想在事件处理函数中，访问与客户端相关的数据或属性

res响应对象：在服务器的request事件处理函数中，如果想访问与服务器相关的数据或属性

const *http* = require('http')

const server = *http*.createServer()

// req 是请求对象，包含了与客户端相关的数据和属性

server.on('request', (*req*, *res*) => {

  // req.url 是客户端请求的 URL 地址

  const url = *req*.url

  // req.method 是客户端请求的 method 类型

  const method = *req*.method

  const str = `Your request url is ${url}, and request method is ${method}`

  console.log(str)

  // 调用 res.end() 方法，向客户端响应一些内容

*res*.end(str)

})

server.listen(8080, () => {

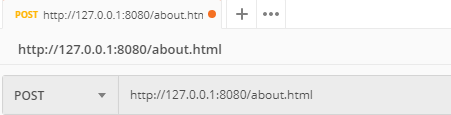
  console.log('server running at http://127.0.0.1:8080')

})

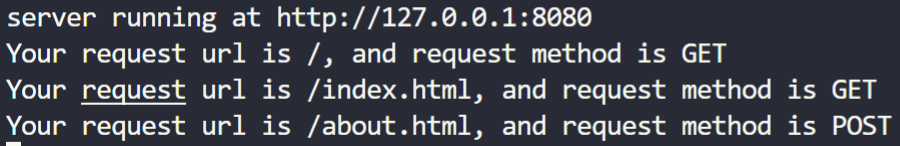
浏览器里get请求



Postman里post请求



终端req请求内容(客户端向服务器请求内容)



浏览器res响应内容(服务器向客户端响应内容)



Postman中body中



解决中文乱码问题

当调用res.end()方法，向客户端发送中文内容的时候，会出现乱码问题，此时，需要手动设置内容的编码格式。

 // 调用 res.setHeader() 方法，设置 Content-Type 响应头的值为text/html;charset=utf-8，解决中文乱码的问题

*res*.setHeader('Content-Type', 'text/html; charset=utf-8')

### 根据不同的url响应不同的html内容

1. 获取请求的url地址
2. 设置默认的响应内容为404 Not found
3. 判断用户请求是否为/或/index.html首页
4. 判断用户请求的是否为/about.html关于页面
5. 设置Content-Type响应头，防止中文乱码
6. 使用res.end()把内容响应给客户端

server.on('request', (*req*, *res*) => {

  // 1. 获取请求的 url 地址

  const url = *req*.url

  // 2. 设置默认的响应内容为 404 Not found

  let content = '<h1>404 Not found!</h1>'

  // 3. 判断用户请求的是否为 / 或 /index.html 首页

  // 4. 判断用户请求的是否为 /about.html 关于页面

  if (url === '/' || url === '/index.html') {

    content = '<h1>首页</h1>'

  } else if (url === '/about.html') {

    content = '<h1>关于页面</h1>'

  }

  // 5. 设置 Content-Type 响应头，防止中文乱码

*res*.setHeader('Content-Type', 'text/html; charset=utf-8')

  // 6. 使用 res.end() 把内容响应给客户端

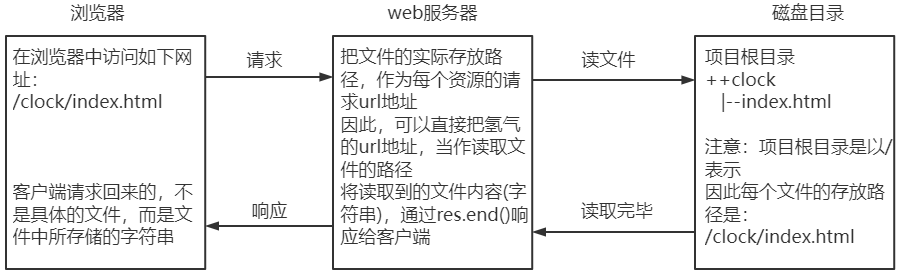
*res*.end(content)

})

### 时钟web服务器案例

把文件的实际存放路径，作为每个资源的请求url地址

服务器充当的角色就是一个字符串的搬运工



1. 导入需要的模块
2. 创建基本的web服务器
3. 将资源的请求url地址映射为文件的存放路径
4. 读取文件内容并响应给客户端
5. 优化资源的请求路径

// 1.1 导入 http 模块

const *http* = require('http')

// 1.2 导入 fs 模块

const *fs* = require('fs')

// 1.3 导入 path 模块

const path = require('path')

// 2.1 创建 web 服务器

const server = *http*.createServer()

// 2.2 监听 web 服务器的 request 事件,监听客户端的请求

server.on('request', (*req*, *res*) => {})

// 2.3 启动服务器

server.listen(8080, () => {})

将资源的请求url地址映射为文件的存放路径

// 3.1 获取到客户端请求的 URL 地址

  const url = *req*.url

 // 3.2 把请求的 URL 地址映射为具体文件的存放路径

  // const fpath = path.join(\_\_dirname, url)

读取文件的内容并响应给客户端

  // 4.1 根据“映射”过来的文件路径读取文件的内容

*fs*.readFile(fpath, 'utf8', (*err*, *dataStr*) => {

    // 4.2 读取失败，向客户端响应固定的“错误消息”

    if (*err*) return *res*.end('404 Not found.')

    // 4.3 读取成功，将读取成功的内容，响应给客户端

*res*.end(*dataStr*)

  })

  // 5.1 预定义一个空白的文件存放路径

  let fpath = ''

  if (url === '/') {

    fpath = path.join(\_\_dirname, './clock/index.html')

  } else {

    fpath = path.join(\_\_dirname, '/clock', url)

  }

## 模块化

### 概念

模块化是指解决一个复杂问题时，自顶向下逐层把系统划分若干模块的过程。对于整个系统来说，模块是可组合、分解和更换的单元。

编程领域中的模块化，就是遵守固定的规则，把一个大文件拆成独立并互相依赖的多个小模块。

把代码进行模块化拆分的好处：

1. 提高了代码的复用性
2. 提高了代码的可维护性
3. 可以实现按需加载

模块化规范

使用什么样的语法格式来引用模块

在模块中使用什么样的语法格式向外暴露成员

模块化规范的好处：大家都遵守同样的模块化来规范写代码，降低了沟通的成本，极大方便了各个模块之间的相互调用。

### 模块化的分类&require的使用

Node.js中根据模块来源的不同，将模块分为了3大类，分别是：

内置模块(内置模块是由Node.js官方提供的，例如fs、path、http等)

自定义模块(用户创建的每个js文件，都是自定义模块)

第三方模块(由第三方开发出来的模块，并非官方提供的内置模块，也不是用户创建的自定义模块，使用前需要先下载)

加载模块

使用强大的require()方法，可以加载需要的内置模块、用户自定义模块、第三方模块进行使用。

使用require()方法加载其他模块时，会执行被加载模块中的代码。

加载用户自定义模块时要给相对路径

// 可以省略 .js 的后缀名

const m1 = require('./06.m1.js')

### 模块作用域和module对象

什么是模块作用域

和函数作用域类似，在自定义模块中定义的变量、方法等成员，只能在当前模块内被访问，这种模块级别的访问限制，叫做模块作用域。

模块作用域的好处

防止全局变量污染的问题

const username = '张三'

function sayHello() {

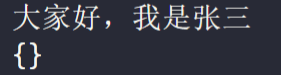
  console.log('大家好，我是' + username)

}

sayHello()

const custom = require('./08.模块作用域')

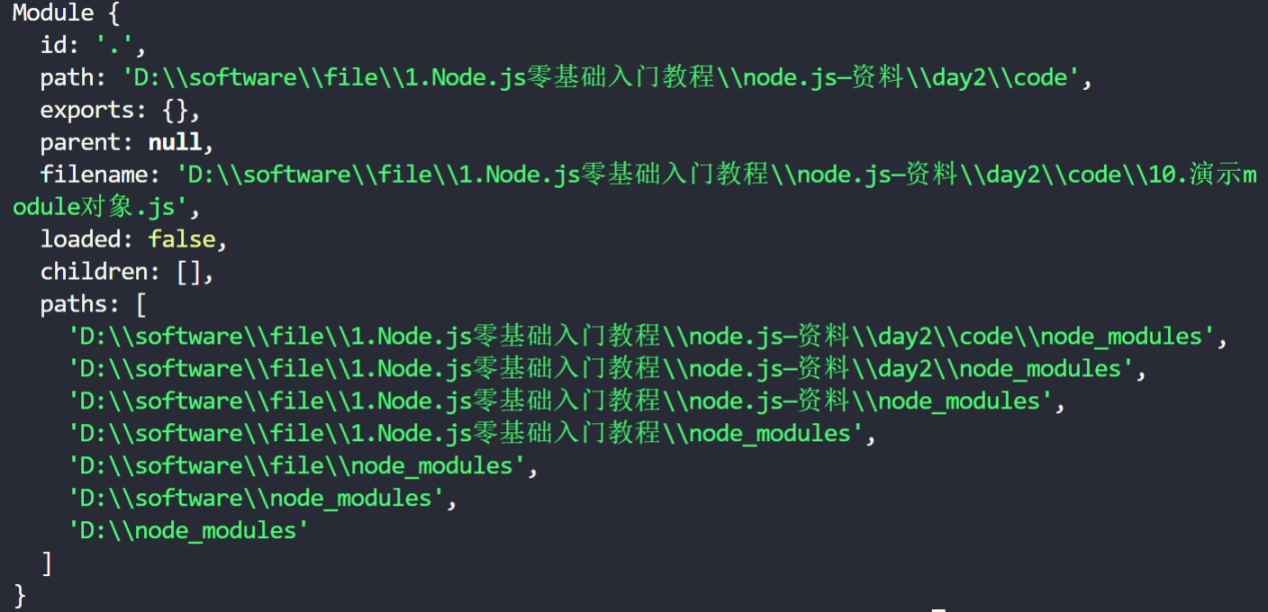
console.log(custom)



向外共享模块作用域中的成员

module对象

在每个.js自定义模块中都有一个module对象，它里面存储了和当前模块有关的信息



### module.exports对象的使用

在自定义模块中，可以使用module.exports对象，将模块内的成员共享出去，供外界使用。

外界require()方法导入自定义模块时，得到的就是module.exports所指向的对象。

// 在一个自定义模块中，默认情况下， module.exports = {}

// 向 module.exports 对象上挂载 username 属性

*module*.*exports*.username = 'zs'

// 让 module.exports 指向一个全新的对象

*module*.*exports* = {

  nickname: '小黑',

  sayHi() {

    console.log('Hi!')

  }

}



使用require()方法导入模块时，导入的结果，永远以module.exports指向的对象为准。exports和module.exports指向同一个对象。

为了防止混乱，建议大家不要在同一个模块中同时使用exports和module.exports

### CommonJS模块化规范

Node.js遵循了CommonJS模块化规范，CommonJS规定了模块的特性和各模块之间如何相互依赖。

CommonJS规定：

1. 每个模块内部，module变量代表当前模块
2. module变量是一个对象，它的exports属性(即module.exports)是对外的接口
3. 加载某个模块，其实是加载该模块的module.exports属性。require()方法用于加载模块

## 包与npm

### 包的概念

Node.js中的第三方模块又叫做包

包的来源

不同于Node.js的内置模块与自定义模块，包是由第三方个人或团队开发出来的，免费供所有人使用。

包是基于内置模块封装出来的，提供了更高级、更方便的API，极大的提高了开发效率。

<https://www.npmjs.com/网站上搜索自己所需要的包>

### 格式化时间的传统做法

1. 创建格式化时间的自定义模块
2. 定义格式化时间的方法
3. 创建补零函数
4. 从自定义模块中导出格式化时间的函数
5. 导入格式化时间的自定义模块
6. 调用格式化时间的函数

// 1. 定义格式化时间的方法

function dateFormat(*dtStr*) {

  const dt = **new** *Date*(*dtStr*)

  const y = dt.getFullYear()

  const m = padZero(dt.getMonth() + 1)

  const d = padZero(dt.getDate())

  const hh = padZero(dt.getHours())

  const mm = padZero(dt.getMinutes())

  const ss = padZero(dt.getSeconds())

  return `${y}-${m}-${d} ${hh}:${mm}:${ss}`

}

// 定义补零的函数

function padZero(*n*) {

  return *n* > 9 ? *n* : '0' + *n*

}

// 导入自定义的格式化时间的模块

const TIME = require('./15.dateFormat')

// 调用方法，进行时间的格式化

const dt = **new** *Date*()

const newDT = TIME.dateFormat(dt)

### 格式化时间的高级做法

1. 使用npm包管理工具，在项目中安装格式化时间的包moment
2. 使用require()导入格式化时间的包
3. 参考moment的官方API文档对时间进行格式化

// 1. 导入需要的包

// 注意：导入的名称，就是装包时候的名称

const moment = require('moment')

//参考moment官方API文档，调用对应的方法，对时间进行格式化

//调用moment()方法，得到当前的时间

//针对当前的时间，调用format()方法，按照指定的格式进行时间的格式化

const dt = moment().format('YYYY-MM-DD HH:mm:ss')

在项目中安装包的命令

npm install 包的完整名称

npm i 完成的包名称

终端中npm install moment

自动安装最新版本的包，如果需要指定版本的包，可以包名之后，通过@符号指定具体的版本 例如npm install moment@2.22.2

初次装包完成后，在项目文件夹下多一个叫做node\_modules的文件夹和package-lock.json的配置文件

其中node\_modules文件夹用来存放所有已安装到项目中的包。require()导入第三方包时，就是从这个目录中查找并加载包。

Package-lock.json配置文件用来记录node\_modules目录下的每一个包的下载信息，例如包的名字、版本号、下载地址等。

程序员不要手动修改node\_modules或package-lock.json文件中的任何代码，npm包管理工具会自动维护它们。

包的语义化版本规范

包的版本是以”点分十进制”形式进行定义的，总共有三位数字，例如2.24.0

第一位数字：大版本

第二位数字：功能版本

第三位数字：Bug修复版本

版本号提升规则：只要前面的版本号增长了，则后面的版本号归零

### 包管理配置文件

npm规定，在项目根目录中，必须提供一个叫做package.json的管理配置文件。用来记录与项目有关的一些配置信息，例如：

1. 项目的名称、版本号、描述等
2. 项目中都用到了哪些包
3. 哪些包只在开发期间会用到
4. 哪些包在开发和部署时都需要用到

整个项目的体积是30.4M

第三方包的体积是28.8M

项目源代码的体积1.6M

遇到的问题：第三方包的体积过大，不方便团队成员之间共享项目源代码

解决方案：共享时剔除node\_modules

今后在项目开发中，一定把node\_modules文件夹，添加到.gitignore忽略文件中

npm包管理工具提供了一个快捷命令，可以在执行命令时所处的目录中，快速创建package.json这个包管理配置文件：npm init -y

上述命令只能在英文的目录问下成功运行！所以，项目文件夹的名称一定要使用英文命名，不要使用中文，不能出现空格。

运行npm install 命令安装包的时候，npm包管理工具会自动把包的名称和版本号，记录到package.json中。

package.json文件中，有一个dependencies节点，专门用来记录您使用npm install命令安装了哪些包。

如果node\_modules删除了，可以运行npm install命令(或npm i)一次性安装所有的依赖包：

一次性安装所有的包：执行npm install命令时，npm包管理工具会先读取package.json中的dependencies节点，读取到记录的所有依赖包名称和版本号之后，npm包管理工具会把这些包一次性下载到项目中。

可以运行npm uninstall命令，来卸载指定的包

npm unstall命令执行成功后，会把卸载的包，自动从package.json的dependencies中移除掉。

如果某些包只在项目开发阶段用到，在项目上线之后不会用到，则建议把这些包记录到devDependencies节点中。

与之对应的，如果某些包在开发和项目上线之后都需要用到，则建议把这些包记录到dependencies节点中。

安装指定的包，并记录到devDependencies节点中

npm i 包名 -D等价npm install 包名 --save-dev

为什么下包速度慢

在使用npm下包的时候，默认从国外的https://registry.npmjs.org/服务器进行下载，此时，网络数据的传输需要经过漫长的海底光缆，因此下包速度会很慢。

解决下包速度慢的问题

淘宝NPM镜像服务器

镜像(Mirroring)是一种文件存储形式，一个磁盘上的数据在另一个磁盘上存在一个完全相同的副本即为镜像。

切换npm的下包镜像源

下包的镜像源，指的就是下包的服务器地址

cmd终端中

查看当前的下包镜像源

npm config get registry

将下包的镜像源切换为淘宝镜像源

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

检查镜像源是否下载成功

npm config get registry

nrm为了更方便的切换下包的镜像源，我们可以安装nrm这个小工具，利用nrm提供的终端命令，可以快速查看和切换下包的镜像源。

通过npm包管理器，将nrm安装为全局可用的工具

cmd终端中

npm i nrm -g

查看所有可用的镜像源

nrm ls

将下包的镜像源切换为taobao镜像

nrm use taobao

### 包的分类和规范的包结构

使用npm包管理工具下载的包，共分为两大类，分别是：

项目包

全局包

项目包：那些被安装到项目的node\_modules目录中的包，都是项目包。

项目包又分为两类，分别是：

开发依赖包(被记录到devDependencies节点中的包，只在开发期间会用到)

核心依赖包(被记录dependencies节点中的包，在开发期间和项目上线之后都会用到)

全局包：在执行npm install命令时，如果提供了-g参数，则会把包安装为全局包。全局包会被安装到C:\Users\用户目录\AppData\Roaming\npm\node\_modules目录下

我的电脑中C:\Users\许高峰\AppData\Roaming\npm\node\_modules

npm i 包名 -g #全局安装指定的包

npm uninstall 包名 -g #卸载全局安装的包

1. 只有工具性质的包，才有全局安装的必要性。因为它们提供了好用的终端命令。
2. 判断某个包是否需要全局安装后才能使用，可以参考官方提供的使用说明即可。

i5ting\_toc是一个可以把md文档转为html页面的小工具，使用步骤如下：

将i5ting\_toc安装为全局包

npm install i5ting\_toc -g

调用i5ting\_toc，轻松实现md转html的功能

i5ting\_toc -f 要转换的md文件路径 -o

-f指定 -o转换完成后自动在浏览器打开

一个规范的包，它的1组成结构，必须符合以下3点要求：

1. 包必须以单独的目录而存在
2. 包的顶级目录下要必须包含package.json这个包管理配置文件
3. package.json中必须包含name，version，main这三个属性，分别代表包的名字、版本号、包的入口。

### 初始化基础的包结构

开发属于自己的包

1. 格式化日期
2. 转义HTML中的特殊符号
3. 还原HTML中的特殊字符

初始化包的基本结构

1. 新建itheima-tools文件夹，作为包的根目录
2. 在itheima-tools文件夹中，新建如下三个文件：

package.json(包管理配置文件)

Index.js(包的入口文件)

README.md(包的说明文档)

初始化package.json

{

  "name": "itheima-tools",

  "version": "1.1.0",

  "main": "index.js",

  "description": "提供了格式化时间、HTMLEscape相关的功能",

  "keywords": [

    "itheima",

    "dateFormat",

    "escape"

  ],

  "license": "ISC" //开源许可协议

}

### 开发属于自己的包

模块化拆分

1. 将格式化时间的功能，拆分到src->dateFormat.js中
2. 将处理HTML字符串的功能，拆分到src->htmlEscape.js中
3. 在index.js中，导入两个模块，得到需要向外共享的方法
4. 在index.js中，使用module.exports把对应的方法共享出去

es6语法中的...的作用？

两个作用：

1.展开运算符号

2.剩余操作符

//1.展开运算符

const a = [2,3];

const b = [1,...a,4];

console.log(b)//结果是：[1,2,3,4]

//2.剩余操作符

const a = [1,2,3,4,5]

const [b,...c] = a;

console.log(b) //结果是1

console.log(c) //结果是2，3，4，5

在index.js中定义格式化时间的方法

// 这是包的入口文件

const date = require('./src/dateFormat')

const escape = require('./src/htmlEscape')

// 向外暴露需要的成员

//...展开运算符，展开这个对象

*module*.*exports* = {

  ...date,

  ...escape

}

const itheima = require('./itheima-tools')

导入的itheima-tools会搜索到package.json里面的main属性，自动导入index.js文件

在index.js中定义转义HTML的方法

// 定义转义 HTML 字符的函数

function htmlEscape(*htmlstr*) {

//|表示或，match是匹配的结果

  return *htmlstr*.replace(/<|>|"|&/g, *match* => {

    switch (*match*) {

      case '<':

        return '&lt;'

      case '>':

        return '&gt;'

      case '"':

        return '&quot;'

      case '&':

        return '&amp;'

    }

  })

}

// 定义还原 HTML 字符串的函数

*module*.*exports* = {

  htmlEscape,

  htmlUnEscape

}

转义HTML结果：



还原HTML结果：



编写包的说明文档

包根目录中的README.md文件，是包的使用说明文档。通过它，我们可以事先把包的使用说明，以markdown的格式写出来，方便用户参考。

README文件中具体写什么内容，没有强制性的要求；只要能够清晰地把包地作用、用法、注意事项等描述描述清楚即可。包含以下6项内容：

安装方式、导入方式、格式化时间、转义HTML中地特殊字符、还原HTML中的特殊字符、开源协议

**## 安装**

```

npm install itheima-tools

```

**## 导入**

```js

const itheima = require('itheima-tools')

```

**## 格式化时间**

```js

// 调用 dateFormat 对时间进行格式化

const dtStr = itheima.dateFormat(**new** Date())

// 结果  2020-04-03 17:20:58

console.log(dtStr)

```

**## 转义 HTML 中的特殊字符**

```js

// 带转换的 HTML 字符串

const htmlStr = '<h1 title="abc">这是h1标签<span>123&nbsp;</span></h1>'

// 调用 htmlEscape 方法进行转换

const str = itheima.htmlEscape(htmlStr)

// 转换的结果 &lt;h1 title=&quot;abc&quot;&gt;这是h1标签&lt;span&gt;123&amp;nbsp;&lt;/span&gt;&lt;/h1&gt;

console.log(str)

```

**## 还原 HTML 中的特殊字符**

```js

// 待还原的 HTML 字符串

const str2 = itheima.htmlUnEscape(str)

// 输出的结果 <h1 title="abc">这是h1标签<span>123&nbsp;</span></h1>

console.log(str2)

```

**## 开源协议**

ISC

### 发布包

注册npm账号

1.访问https://www.npmjs.com/网站，点击sign up按钮，进入注册用户界面

2.填写账号相关的信息：Full Name、Public Email、Username、Password

1. 点击Create an Account按钮，注册账号
2. 登录邮箱，点击验证链接，进行账号的验证

登录npm账号

npm账号注册完成后，可以在终端中执行npm login命令，依次输入用户名、密码、邮箱后，即可登录成功。

在运行npm login命令之前，必须先把下包的服务器地址切换为npm的官方服务器。否则会导致发布包失败！

vscode终端中：

nrm use npm

npm login

把包发布到npm上

将终端切换到包的根目录之后，运行npm publish命令，即可将包发布到npm上(注意：包名不能雷同)

cd itheima-tools

npm publish

删除已发布的包

运行npm unpublish包名 --force命令，即可从npm删除已发布的包

1. npm unpublish命令只能删除72小时以内发布的包
2. npm unpublish删除的包，在24小时内不允许重复发布
3. 发布包的时候要慎重，尽量不要往npm上发布没有意义的包

### 模块的加载机制

优先从缓存中加载

模块在第一次加载后会被缓存。这也意味着多次调用require()不会导致模块的代码被执行多次。

不论是内置模块、用户自定义模块、还是第三方模块，它们都会优先从缓存中加载，从而提高模块的加载效率。

require('./03.自定义模块')

require('./03.自定义模块')

require('./03.自定义模块')

只会执行一次require()

内置模块的加载机制

内置模块是由Node.js官方提供的模块，内置模块的加载优先级最高。

例如，require(‘fs’)始终返回内置的fs模块，即使在node\_modules目录下有名字相同的包也叫做fs。

自定义模块的加载机制

使用require()加载自定义模块时，必须指定以./或../开头的路径标识符。在加载自定义模块时，如果没有指定./或../这样的路径标识符，则node会把它当作内置模块或第三方模块进行加载。

同时，在使用require()导入自定义模块时，如果忽略了文件的扩展名，则Node.js会按顺序分别尝试加载以下的文件：

1. 按照确切的文件名进行加载
2. 补全.js扩展名进行加载
3. 补全.json扩展名进行加载
4. 补全.node扩展名进行加载
5. 加载失败，终端报错

第三方模块的加载机制

如果传递给require()的模块标识符不是一个内置模块，也没有以’./’或’../’开头，则Node.js会从当前模块的父目录开始，尝试从/node\_modules文件夹中加载第三方模块。

如果没有找到对应的第三方模块，则移动到再上一层父目录中，进行加载，直到文件系统的根目录。

例如：加载在C:\Users\itheima\project\foo.js文件里调用require(“tools”)，则Node.js会按以下顺序查找：

1. C:\Users\itheima\project\node\_modules\tools
2. C:\Users\itheima\node\_modules\tools
3. C:\Users\node\_modules\tools
4. C:\node\_modules\tools

目录作为模块

当把目录作为模块标识符，传递给require()进行加载的时候，有三种加载方式：

1. 在被加载的目录下查找一个叫做package.json的文件，并寻找main属性，作为require()加载的入口。
2. 如果目录里没有package.json文件，或者main入口不存在或无法解析，则Node.js会试图加载目录下的index.js文件。
3. 如果以上两步都失败了，则Node.js会在终端打印错误消息，报告模块的缺失：Error:Cannot find module ‘xxx’

## express

### 认识express并创建基本web服务器

使用express.static()快速托管静态资源

使用express路由精简项目结构

使用常见的express中间件

使用express创建API接口

能够在express中启用cors跨域资源共享

1. 初识Express
2. Express路由
3. Express中间件
4. 使用Express写接口

Express是基于Node.js平台，快速、开发、极简的Web开发框架。

Express的作用和Node.js内置的http模块类似，是专门用来创建Web服务器的。

Express的本质：就是一个npm上的第三方包，提供了快速创建Web服务器的便捷方法。

Express的中文官网：<http://www.expressjs.com.cn/>

不使用Express能否创建Web服务器？

答案：能，使用Node.js提供的原生http模板即可。

有了http内置模块，为什么还有用Express？

答案：http内置模块用起来很复杂，开发效率低；Express是基于内置的http模块进一步封装出来的，能够极大的提高开发效率。

http内置模块与Express是什么关系？

答案：类似于浏览器中Web API和jQuery的关系。后者是基于前者进一步封装出来。

对于前端程序员来说，最常见的两种服务器，分别是：

Web网站服务器：专门对外提供Web网页资源的服务器。

API接口服务器：专门对外提供API接口的服务器。

使用Express，我们可以方便、快捷的创建Web网站的服务器或API接口的服务器。

安装：npm i express @4.17.1

创建基本Web服务器

// 1. 导入 express

const express = require('express')

// 2. 创建 web 服务器

const app = express()

// 3. 启动 web 服务器

app.listen(80, () => {

  console.log('express server running at http://127.0.0.1')

})

### 监听GET请求和POST请求&处理参数

通过app.get()方法，可以监听客户端的GET请求，具体的语法格式如下：

参数1：客户端的URL地址

参数2：请求对应的处理函数

req:请求对象(包含了与请求相关的属性与方法)

res:响应对象(包含了与响应相关的属性与方法)

通过res.send()方法，可以把处理好的内容，发送给客户端：

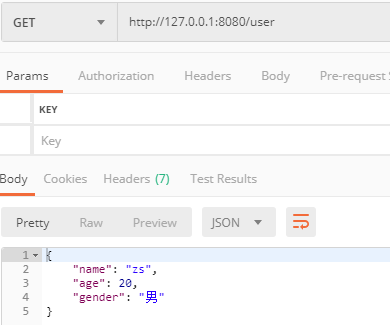
// 4. 监听客户端的 GET 和 POST 请求，并向客户端响应具体的内容

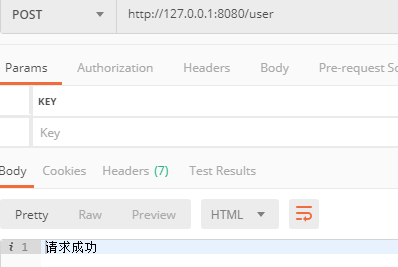
app.get('/user', (*req*, *res*) => {

  // 调用 express 提供的 res.send() 方法，向客户端响应一个 JSON 对象，向客户端响应一个 文本字符串

*res*.send({ name: 'zs', age: 20, gender: '男' })

})





获取URL中携带的查询参数

通过req.query对象，可以访问到客户端通过查询字符串的形式，发送到服务器的参数：

app.get('/', (*req*, *res*) => {

//req.query默认是一个空对象

//客户端使用？name=zs&age=20这种查询字符串形式，发送到服务器的参数

//可以通过req.query对象访问到，例如

//req.query.name req.query.age

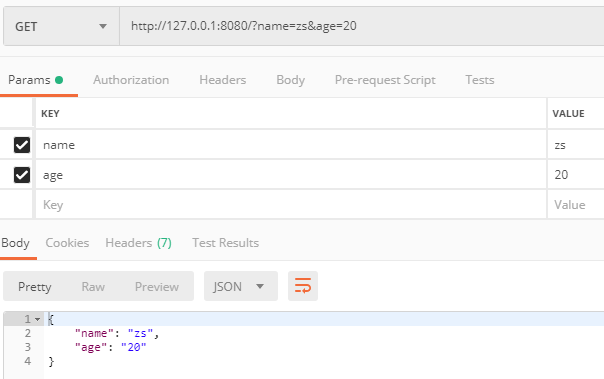
  // 通过 req.query 可以获取到客户端发送过来的 查询参数

  // 注意：默认情况下，req.query 是一个空对象

  console.log(*req*.query)

*res*.send(*req*.query)

})



获取URL中的动态参数

通过req.params对象，可以访问到URL中，通过：匹配到的动态参数：

// 注意：这里的 :id 是一个动态的参数

//URL地址中，可以通过:参数名的形式，匹配动态参数值

app.get('/user/:ids/:username', (*req*, *res*) => {

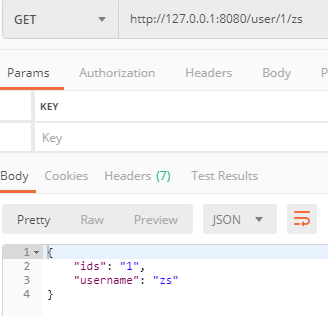
  // req.params 是动态匹配到的 URL 参数，默认也是一个空对象

//里面存放着通过:动态匹配到的参数值

  console.log(*req*.params)

*res*.send(*req*.params)

})



### 资源静态处理

托管静态资源

express提供了一个非常好用的函数，叫做express.static(),通过它，我们可以非常方便地创建一个静态资源服务器，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/bg.jpg>

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

http://localhost:3000/js/login.js

Express在指定的静态目录中查找文件，并对外提供资源的访问路径。因此，存放静态文件的目录名不会出现在URL中。

// 在这里，调用 express.static() 方法，快速的对外提供静态资源

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

app.use(express.static('./clock'))

如果要托管多个静态资源目录，请多次调用express.static()函数

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

挂载路径前缀

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

### nodemon

在编写调试Node.js项目的时候，如果修改了项目的代码，则需要频繁的手动close掉，然后再重新启动，非常繁琐。

现在，我们可以使用nodemon(https://www.npmjs.com/package/nodemon)这个工具，它能够监听项目文件的变动，当代码被修改后，nodemon会自动帮我们重启项目，极大方便了开发和调试。

cmd终端中安装

安装:npm install nodemon -g

当基于Node.js编写一个网站应用的时候，传统的方式，是运行node app.js命令，来启动项目，这样做的坏处是：代码被修改之后，需要手动重启项目。

现在，我们可以将node命令修改为nodemon命令，使用nodemon app.js来启动项目。这样做的好处是：代码被修改之后，会被nodemon监听到，从而实现自动重启项目的效果。

## Express路由

### 路由的概念

路由是按键与服务之间的关系

在Express中，路由指的是客户端的请求与服务器处理函数之间的映射关系。

Express中的路由分3部分组成，分别是请求的类型、请求的URL地址、处理函数，格式如下：

app.METHOD(PATH,HANDLER)

// 挂载路由

app.get('/', (*req*, *res*) => {

*res*.send('hello world.')

})

app.post('/', (*req*, *res*) => {

*res*.send('Post Request.')

})

每当一个请求到达服务器之后，需要先经过路由的匹配，只有匹配成功之后，才会调用对应的处理函数。

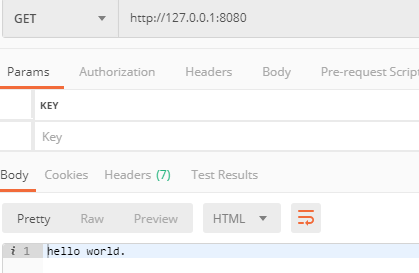
在匹配时，会按照路由的顺序进行匹配，如果请求类型和请求的URL同时匹配成功，则Express会将这次请求，转交给对应的function函数进行处理。

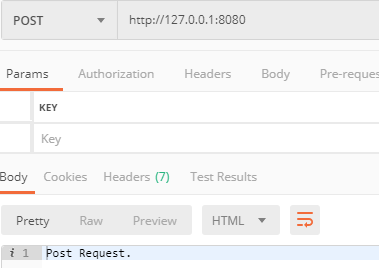
1. 按照定义的先后顺序进行匹配
2. 请求类型和请求的URL同时匹配成功，才会调用对应的处理函数。

vscode终端中cd code

npm init -y

npm i [express@4.17.1](mailto:express@4.17.1)





### 路由的使用

模块化路由

为了方便对路由进行模块化的管理，Express不建议将路由直接挂载到app上，而是推荐将路由抽离为单独的模块。

1. 创建路由模块对应的js文件
2. 调用express.Router()函数创建路由对象
3. 向路由对象上挂载具体的路由
4. 使用module.exports向外共享路由对象
5. 使用app.use()函数注册路由模块

03.router.js

// 这是路由模块

// 1. 导入 express

const express = require('express')

// 2. 创建路由对象

const router = express.Router()

// 3. 挂载具体的路由

router.get('/user/list', (*req*, *res*) => {

*res*.send('Get user list.')

})

// 4. 向外导出路由对象

*module*.*exports* = router

02.模块化路由.js

// 1. 导入路由模块

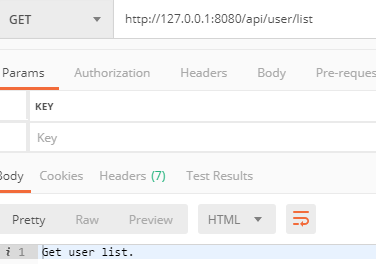
const app = express()

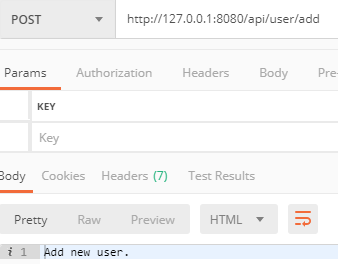
const router = require('./03.router')

// 2. 注册路由模块,并添加统一的访问前缀/api

app.use('/api', router)

// 注意： app.use() 函数的作用，就是来注册全局中间件





### Express中间件

中间件(Middleware)，特指业务流程的中间处理环节。

调用流程：当一个请求到达Express的服务器之后，可以连续调用多个中间件，从而对这次请求进行预处理。

中间件的格式

app.get(‘/’,function(req,res,next){

next();

})

中间件函数的形参列表中，必须包含next参数。而路由处理函数中只包含req和res。

next函数是实现多个中间件连续调用的关键，它表示把流转关系转交给下一个中间件或路由。

定义中间件函数

// // 定义一个最简单的中间件函数

//常量mv所指向的，就是一个中间件函数

// const mw = function (req, res, next) {

//   console.log('这是最简单的中间件函数')

//在当前中间件的业务处理完毕后，必须调用next()函数

//   // 把流转关系，转交给下一个中间件或路由

//   next()

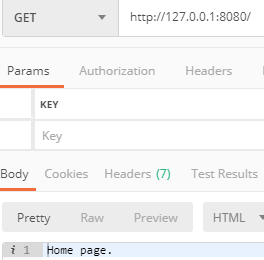
// }

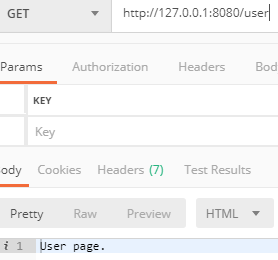
客户端发起的任何请求，到达服务器后，都会触发的中间件，叫做全局生效的中间件。

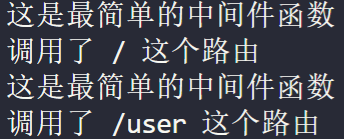
通过调用app.use(中间件函数)，即可定义一个全局生效的中间件。

// // 将 mw 注册为全局生效的中间件

// app.use(mw)







// 这是定义全局中间件的简化形式

app.use((*req*, *res*, *next*) => {

  console.log('这是最简单的中间件函数')

  next()

})

中间件的作用

多个中间件之间，共享同一份req和res，基于这样的特性，我们可以在上游的中间件中，统一为req和res对象添加自定义的属性和方法，供下游的中间件或路由进行使用。

// 这是定义全局中间件的简化形式

app.use((*req*, *res*, *next*) => {

  // 获取到请求到达服务器的时间

  const time = *Date*.now()

  // 为 req 对象，挂载自定义属性，从而把时间共享给后面的所有路由

*req*.startTime = time

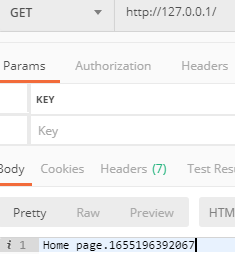
  next()

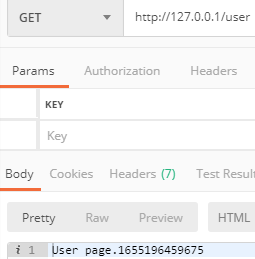
})

app.get('/', (*req*, *res*) => {

*res*.send('Home page.' + *req*.startTime)

})





定义多个全局中间件

可以使用app.use()连续定义多个全局中间件。客户端请求到达服务器之后，会按照定义的先后顺序依次进行调用

// 定义第一个全局中间件

app.use((*req*, *res*, *next*) => {

  console.log('调用了第1个全局中间件')

  next()

})

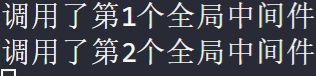
// 定义第二个全局中间件

app.use((*req*, *res*, *next*) => {

  console.log('调用了第2个全局中间件')

  next()

})



### 局部生效的中间件

不使用app.use()定义的中间件，叫做局部生效的中间件

// 1. 定义中间件函数

const mw1 = (*req*, *res*, *next*) => {

  console.log('调用了局部生效的中间件')

  next()

}

// 2. 创建路由

app.get('/', mw1, (*req*, *res*) => {

*res*.send('Home page.')

})

可以在路由中，通过如下两种等价的方式，使用多个局部中间件：

// 2. 创建路由

app.get('/', mw1, mw2, (*req*, *res*) => {

*res*.send('Home page.')

})

// 2. 创建路由

app.get('/', [mw1, mw2], (*req*, *res*) => {

*res*.send('Home page.')

})

了解中间件的5个使用注意事项

1. 一定要在路由之前注册中间件
2. 客户端发送过来的请求，可以连续调用多个中间件进行处理
3. 执行完中间件的业务代码之后，不要忘记调用next()函数
4. 为了防止代码逻辑混乱，调用next()函数后不要再写额外的代码
5. 连续调用多个中间件时，多个中间件之间，共享req和res对象

### 中间件的分类

1. 应用级别的中间件
2. 路由级别的中间件
3. 错误级别的中间件
4. Express内置的中间件
5. 第三方的中间件

应用级别的中间件：通过app.use()或app.get()或app.post()，绑定到app实例上的中间件。包括全局中间件和局部中间件。

路由级别的中间件：绑定到express.Router()实例上的中间件，叫做路由级别的中间件。应用级别中间件是绑定到app实例上，路由级别中间件绑定到router实例上。

错误级别的中间件：专门用来捕获整个项目中发生的异常错误，从而防止项目异常崩溃的问题。

格式：错误级别中间件的function处理函数中，必须有4个形参，形参顺序从前到后，分别是(err,req,res,next)。

// 1. 定义路由

app.get('/', (*req*, *res*) => {

  // 1.1 人为的制造错误

  throw **new** *Error*('服务器内部发生了错误！')

*res*.send('Home page.')

})

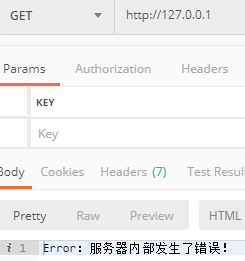
// 2. 定义错误级别的中间件，捕获整个项目的异常错误，从而防止程序的崩溃

app.use((*err*, *req*, *res*, *next*) => {

  console.log('发生了错误！' + *err*.message)

*res*.send('Error：' + *err*.message)

})



错误级别的中间件必须注册在所有路由之后！

Express内置的中间件：自Express4.16.0版本开始，Express内置了3个常用的中间件，极大的提高了Express项目的开发效率和体验

1. express.static快速托管静态资源的内置中间件，例如：HTML文件、图片、CSS样式(无兼容性)
2. express.json解析JSON格式的请求体资源(有兼容性，仅在4.16.0+版本中可用)
3. express.urlencoded解析URL-encoded格式的请求体数据(有兼容性，仅在4.16.0+版本中可用)

// 注意：除了错误级别的中间件，其他的中间件，必须在路由之前进行配置

// 通过 express.json() 这个中间件，解析表单中的 JSON 格式的数据

app.use(express.json())

// 通过 express.urlencoded() 这个中间件，来解析 表单中的 url-encoded 格式的数据

app.use(express.urlencoded({ extended: false }))

app.post('/user', (*req*, *res*) => {

  // 在服务器，可以使用 req.body 这个属性，来接收客户端发送过来的请求体数据

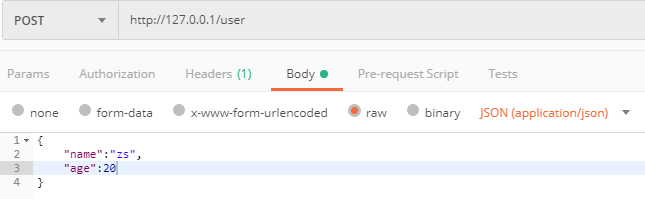
  // 默认情况下，如果不配置解析表单数据的中间件，则 req.body 默认等于 undefined

 // 在服务器端，可以通过 req,body 来获取 JSON 格式的表单数据和 url-encoded 格式的数据

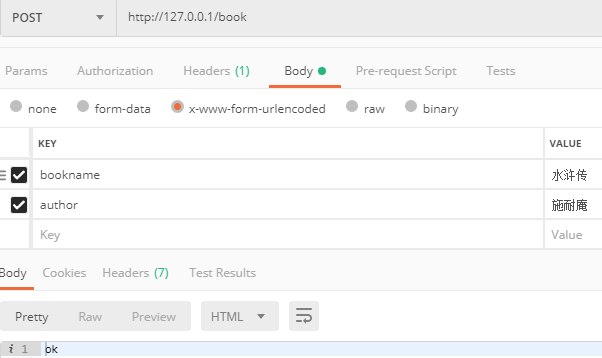
  console.log(*req*.body)

*res*.send('ok')

})









第三方的中间件：非Express官方内置的，而是由第三方开发出来的中间件。在项目中，大家可以按需下载并配置第三方中间件，从而提高项目的开发效率。

例如：在express@4.16.0之前的版本中，经常使用body-parser这个第三方中间件，来解析请求体数据。使用步骤如下：

1. 运行npm install body-parser安装中间件
2. 使用require导入中间件
3. 调用app.use()注册并使用中间件

// 1. 导入解析表单数据的中间件 body-parser

const parser = require('body-parser')

// 2. 使用 app.use() 注册中间件

app.use(parser.urlencoded({ extended: false }))

// app.use(express.urlencoded({ extended: false }))

注意：Express内置的express.urlencoded中间件，就是基于body-parser这个第三方中间进一步封装出来的。

### 自定义中间件

需求描述与实现步骤：

自己手动模拟一个类似于express.urlencoded这样的中间件，来解析POST提交到服务器的表单数据。

1. 定义中间件
2. 监听req的data事件
3. 监听req的end事件
4. 使用querystring模块解析请求体数据
5. 将解析出来的数据对象挂载为req.body
6. 将自定义中间件封装为模块

监听req的data事件：在中间件中，需要监听req对象的data事件，来获取客户端发送到服务器的数据。

如果数据量比较大，无法一次性发送完毕，则客户端会把数据切割后，分批发送到服务器。所以data事件可能会触发多次，每一次触发data事件时，获取到数据只是完整数据的一部分，需要手动对接收到的数据进行拼接。

监听req的end事件：当请求体数据接收完毕之后，会自动触发req的end事件。因此，我们可以在req的end事件中，拿到并处理完整的请求体数据。

使用querystring模块解析请求体数据

Node.js内置了一个querystring模块，专门来处理查询字符串。通过这个模块提供的parse()函数，可以轻松把查询字符串，解析成对象的格式。

将解析出来的数据对象挂载为req.body：上游的中间件和下游的中间件及路由之间，共享同一份req和res。因此，我们可以将解析出来的数据，挂载为req的自定义属性，命名为req.body，供下游使用。

// 导入 Node.js 内置的 querystring 模块

const *qs* = require('querystring')

// 这是解析表单数据的中间件

app.use((*req*, *res*, *next*) => {

  // 定义中间件具体的业务逻辑

  // 1. 定义一个 str 字符串，专门用来存储客户端发送过来的请求体数据

  let str = ''

  // 2. 监听 req 的 data 事件(客户端发送过来的新的请求体数据)

*req*.on('data', (*chunk*) => {

//拼接请求体数据，隐式转换为字符串

    str += *chunk*

  })

  // 3. 监听 req 的 end 事件(请求体发送完毕后自动触发)

*req*.on('end', () => {

    // 在 str 中存放的是完整的请求体数据

    // console.log(str)

    // TODO: 把字符串格式的请求体数据，解析成对象格式

    const body = *qs*.parse(str)

*req*.body = body

    next()

  })

})

打印str



打印body



body的JSON格式



将自定义中间件封装为模块：为了优化代码结构，我们可以把自定义的中间件函数，封装为独立的模块。

在14.custom-body-parser.js中

const bodyParser = (*req*, *res*, *next*) => {

  // 定义中间件具体的业务逻辑

  // 1. 定义一个 str 字符串，专门用来存储客户端发送过来的请求体数据

}

*module*.*exports* = bodyParser //向外导出解析请求体数据的中间件函数

在13.对自定义的中间件进行模块化的拆分.js中

// 1. 导入自己封装的中间件模块

const customBodyParser = require('./14.custom-body-parser')

// 2. 将自定义的中间件函数，注册为全局可用的中间件

app.use(customBodyParser)

### 使用Express写接口

创建基本的服务器

创建API路由模16.apiRouter.js

编写GET接口

// 在这里挂载对应的路由

router.get('/get', (*req*, *res*) => {

  // 通过 req.query 获取客户端通过查询字符串，发送到服务器的数据

  const query = *req*.query

  // 调用 res.send() 方法，向客户端响应处理的结果

*res*.send({

    status: 0, // 0 表示处理成功，1 表示处理失败

    msg: 'GET 请求成功！', // 状态的描述

    data: query, // 需要响应给客户端的数据

  })

})



编写POST接口

// 定义 POST 接口

router.post('/post', (*req*, *res*) => {

  // 通过 req.body 获取请求体中包含的 url-encoded 格式的数据

  const body = *req*.body

  // 调用 res.send() 方法，向客户端响应结果

*res*.send({

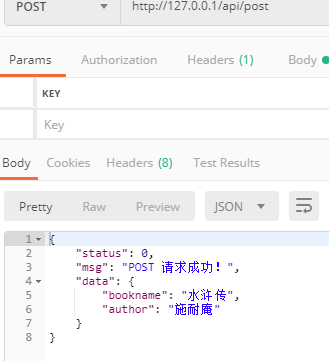
    status: 0,

    msg: 'POST 请求成功！',

    data: body,

  })

})



// 定义 DELETE 接口

router.delete('/delete', (*req*, *res*) => {

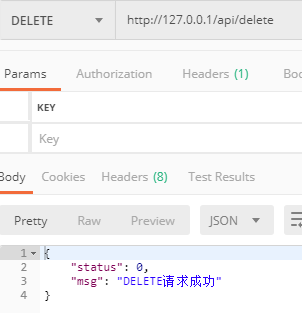
*res*.send({

    status: 0,

    msg: 'DELETE请求成功',

  })

})



*module*.*exports* = router

// 导入路由模块

const router = require('./16.apiRouter')

// 把路由模块，注册到 app 上

app.use('/api', router)

### CORS跨域资源共享

接口的跨域问题

刚才编写的GET和POST接口，存在一个很严重的问题：不支持跨域请求

当页面被载入并且DOM树完成初始化后触发。

$(document).ready(function () {

/\*对document设置事件当页面加载完毕，

DOM树生成完毕之后就会触发事件执行后面的js函数\*；/

})；

//简略写法

$(function(){

//后续需要执行的js函数；

})

$(function () {

// 1. 测试GET接口

        $('#btnGET').on('click', function () {

          $.ajax({

            type: 'GET',

            url: 'http://127.0.0.1/api/get',

            data: { name: 'zs', age: 20 },

            success: function (*res*) {

              console.log(*res*)

            },

          })

        })}

协议、域名、端口号任意一项不同就存在跨域

协议不同,http和https

解决接口跨域问题的方案主要有两种：

1. CORS(主流的解决方案，推荐使用)
2. JSONP(有缺陷的解决方案：只支持GET请求)

使用cors中间件解决跨域问题

cors是Express的一个第三方中间件。通过安装和配置cors中间件，可以很方便地解决跨域问题。

1. 运行npm install cors 安装中间件
2. 使用const cors=require(“cors”)导入中间件
3. 在路由之前调用app.use(cors())配置中间件

// 一定要在路由之前，配置 cors 这个中间件，从而解决接口跨域的问题

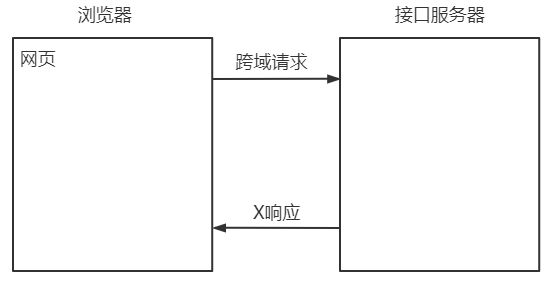
const cors = require('cors')

app.use(cors())

什么是CORS

CORS(Cross-Origin Resource Sharing,跨域资源共享)由一系列HTTP响应头组成，这些HTTP响应头决定浏览器是否阻止前端JS代码跨域获取资源。

浏览器的同源安全策略默认会阻止网页”跨域”获取资源。但如果接口服务器配置了CORS相关的HTTP响应头，就可以解除浏览器端的跨域访问限制。



响应的结果被浏览器拦截，网页无法获取到跨域响应的数据

在服务器端，配置cors相关的响应头，从而解除浏览器端的跨域访问限制，配置Access-Control-Allow-\*

1. CORS主要在服务器端进行配置。客户端浏览器无需做任何额外的配置，即可请求开启了CORS的接口。
2. CORS在浏览器中有兼容性。只有支持XMLHttpRequest Level2的浏览器，才能正常访问开启了CORS的服务端接口(例如：IE10+、Chrome4+、FileFox3.5+)

### cors相关的三个响应头

CORS响应头部-Access-Control-Allow-Origin

语法：Access-Control-Allow-Origin:<origin> | \*

其中，origin参数的值指定了允许访问该资源的外域URL。

例如，下面的字段值将只允许来自http://itcast.cn的请求：

res.setHeader(‘Access-Control-Allow-Origin’,’http://itcast.cn’)

如果指定了Access-Control-Allow-Origin字段的值为通配符\*，表示允许来自任何域的请求

res.setHeader(‘Access-Control-Allow-Origin’,’\*’)

CORS响应头部-Access-Control-Allow-Headers

默认情况下，CORS仅支持客户端向服务端发送如下所示的9个请求头：

Accept、Accept-Language、Content-Language、DPR、Downlink、Save-Data、Viewport-Width、Width、Content-Type(值仅限于text/plain、multipart/form-data、application/x-www-form-urlencoded三者之一)

如果客户端向服务器发送了额外的请求头信息，则需要在服务器端，通过Access-Control-Allow-Headers对额外的请求头进行声明，否则这次请求会失败！

//允许客户端额外向服务器发送Content-Type请求头和X-Custom-Header请求头

//注意：多个请求头之间使用英文的逗号进行分割

res.setHeader(‘Access-Control-Allow-Headers’,’Content-Type,X-Custom-Header’)

CORS响应头部-Access-Control-Allow-Methods

默认情况下，CORS仅支持客户端发起GET、POST、HEAD请求。

如果客户端希望通过PUT、DELETE等方式请求服务器的资源，则需要在服务器端，通过Access-Control-Allow-Methods来指明实际请求所允许使用的HTTP方法。

//只允许POST、GET、DELETE、HEAD请求方法

res.setHeader(‘Access-Control-Allow-Methods’,’POST,GET,DELETE,HEAD’)

//允许所有的HTTP请求方法

res.setHeader(‘Access-Control-Allow-Methods’,’\*’)

### cors的简单请求与预检请求

CORS请求的分类

客户端在请求CORS接口时，根据请求方式和请求头的不同，可以将CORS的请求分为两大类，分别是简单请求和预检请求

简单请求

同时满足以下两大条件的请求，就属于简单请求：

1. 请求方式：GET、POST、HEAD三者之一
2. HTTP头部信息不超过以下几种字段：无自定义头部字段、Accept、Accept-Language、Content-Language、DPR、Downlink、Save-Data、Viewport-Width、Width、Content-Type(只有三个值text/plain、multipart/form-data、application/x-www-form-urlencoded)

预检请求

只要符合以下任何一个条件的请求，都需要进行预检请求：

1. 请求方式为GET、POST、HEAD之外的请求Method类型
2. 请求头中包含自定义头部字段
3. 向服务器发送了application/json格式的数据

在浏览器与服务器正式通信之前，浏览器会先发送OPTION请求进行预检，以获知服务器是否允许实际请求，所以这一次的OPTION请求称为”预检请求”。服务器成功响应预检请求后，才会发送真正的请求，并且携带真实数据。

简单请求和预检请求的区别

简单请求的特点：客户端与服务器之间只会发生一次请求。

预检请求的特点：客户端与服务器之间会发生两次请求，OPTION预检请求成功后，才会发起真正的请求。

使用火狐浏览器

预检请求

 // 3. 为删除按钮绑定点击事件处理函数

        $('#btnDelete').on('click', function () {

          $.ajax({

            type: 'DELETE',

            url: 'http://127.0.0.1/api/delete',

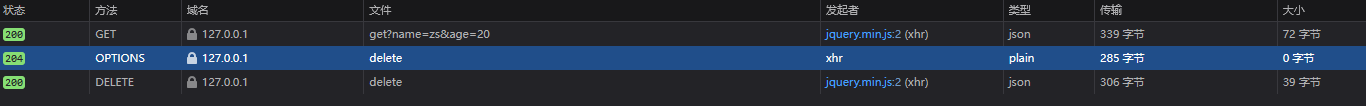
            success: function (*res*) {

              console.log(*res*)

            },

          })

        })



### JSONP接口

概念：浏览器端通过<script>标签的src属性，请求服务器上的数据，同时，服务器返回一个函数的调用。这种请求数据数据的方式叫做JSONP。

1. JSONP不属于真正的Ajax请求，因为它没有使用XMLHttpRequest这个对象。
2. JSONP仅支持GET请求，不支持POST、PUT、DELETE等请求。

创建JSONP接口的注意事项

如果项目中已经配置了CORS跨域资源共享，为了防止冲突，必须在配置CORS中间件之前声明JSONP的接口。否则JSONP接口会被处理成开启了CORS的接口。

实现JSONP接口的步骤

1. 获取客户端发送过来的回调函数的名字
2. 得到要通过JSONP形式发送给客户端的数据
3. 根据前两步得到的数据，拼接出一个函数调用的字符串
4. 把上一步拼接得到的字符串，响应给客户端的<script>标签进行解析执行

// 必须在配置 cors 中间件之前，配置 JSONP 的接口

app.get('/api/jsonp', (*req*, *res*) => {

  // TODO: 定义 JSONP 接口具体的实现过程

  // 1. 获取客户端发送过来的回调函数的名称

//callback是客户端传过来的key jQuery34103029150767417774\_1655230348156

  const funcName = *req*.query.callback

  // 2. 通过JSONP形式定义要发送到客户端的数据对象

  const data = { name: 'zs', age: 22 }

  // 3. 拼接出一个函数的调用的字符串

//JSON.stringify() 方法用于将 JavaScript 值转换为 JSON 字符串。

//jQuery34109706661533436063\_1655230526353({"name":"zs","age":22})

  const scriptStr = `${funcName}(${JSON.stringify(data)})`

  // 4. 把拼接的字符串，响应给客户端

*res*.send(scriptStr)

})

// 一定要在路由之前，配置 cors 这个中间件，从而解决接口跨域的问题

const cors = require('cors')

app.use(cors())

get-query

post-body

在网页中使用jQuery发起JSONP请求

调用$.ajax()函数，提供JSONP的配置选项，从而发起JSONP请求

 // 4. 为 JSONP 按钮绑定点击事件处理函数

        $('#btnJSONP').on('click', function () {

          $.ajax({

            type: 'GET',

            url: 'http://127.0.0.1/api/jsonp',

            dataType: 'jsonp',

            success: function (*res*) {

              console.log(*res*)

            },

          })

        })

## 数据库

### 数据库的基本概念

数据库(database)是用来组织、存储和管理数据的仓库。

用户可以对数据库中的数据进行新增、查询、更新、删除等操作。

常见的数据库有如下几个：

MySQL数据库(目前使用最广泛、流行度最高的开源免费数据库；Community+Enterprise)

Oracle数据库(收费)

SQL Server数据库(收费)

Mongodb数据库(Community+Enterprise)

其中，MySQL、Oracle、SQL Server属于传统型数据库(又叫做：关系型数据库或SQL数据库)，这三者的设计理念相同，用法比较类似。

而Mongodb属于新型数据库(又叫做：非关系型数据库或NoSQL数据库)，它在一定程度上弥补了传统型数据库的缺陷。

数据的组织结构：指的就是数据库以什么样的结构进行存储。

传统型数据库的数据组织结构对比Excel中数据的组织结构

Excel的数据组织结构：工作簿、工作表、数据行、列

1. 整个Excel叫做工作簿
2. Users和books是工作表
3. Users工作表中有几行数据
4. 每行数据由几列信息组成
5. 每列信息都有对应的数据类型

传统型数据库的数据组织结构：数据库(database)、数据表(table)、数据行(row)、字段(field列)，与Excel的数据组织结构组成一一对应，每个字段都有对应的数据类型。

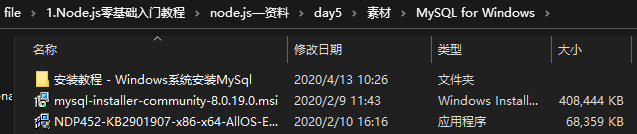
1. 在实际项目开发中，一般情况下，每个项目都对应独立的数据库
2. 不同的数据，要存储到数据库的不同表中
3. 每个表中具体存储哪些信息，由字段来决定
4. 表中的行，代表每一条具体的数据

### 安装MySQL数据库

对于开发人员来说，只需要安装MySQL Server和MySQL Workbench这两个软件，就能满足开发的需要了。

MySQL Server：专门用来提供数据存储和服务的软件。

MySQL Workbench：可视化的MySQL管理工具，通过它，可以方便的操作存储在MySQL Server中的数据。



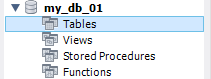
在Windows环境下的安装

只需要运行mysql-installer-community-8.0.19.0.msi这个安装包，就能一次性将MySQL Server和MySQL Workbench安装到自己的电脑上。

可以参考D:\software\file\1.Node.js零基础入门教程\node.js—资料\day5\素材\MySQL for Windows\安装教程 - Windows系统安装MySql\README.md

### MySQL-Workbench的基本用法

在services.msc打开MySQL服务



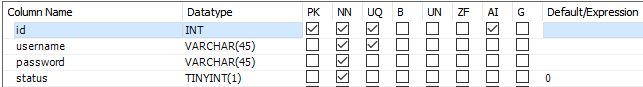
创建数据表

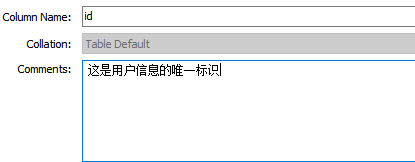
DataType数据类型：

1. int整数
2. varchar(len)字符串
3. tinyint(1)布尔值

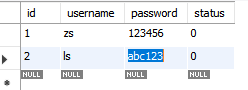
字段的特殊标识：

1. PK(Primary Key)主键、唯一标识
2. NN(Not Null)值不允许为空
3. UQ(Unique)值唯一
4. AI(Auto Increment)值自动增长





向表中写入数据



### SQL

SQL(英文全称：Structured Query Language)是结构化查询语言，专门用来访问和处理数据库的编程语言。能够让我们以编程的形式，操作数据库里面的数据。

三个关键点：

1. SQL是一门数据库编程语言
2. 使用SQL语言编写出来的代码，叫做SQL语句
3. SQL语句只能在关系型数据库中使用(例如MySQL、Oracle、SQL Server)。非关系型数据库(例如Mongodb)不支持SQL语言。
4. 从数据库中查询数据
5. 向数据库中插入更新数据
6. 更新数据库中的数据
7. 从数据库删除数据
8. 可以创建新数据库
9. 可在数据库中创建新表
10. 可在数据库中创建存储过程、视图
11. etc...

查询数据(select)、插入数据(insert into)、更新数据(update)、删除数据(delete)

额外需要掌握的4中SQL语法：

where条件、and和or运算符、order by排序、count(\*)函数

SELECT语句用于从表中查询数据。执行的结果被存储在一个结果表(称为结果集)

-- 这是注释

-- 从FROM指定表中，查询出所有的数据。\*表示所有列

SELECT \* FROM 表名称

-- 从FROM指定的表中，查询出指定列名称(字段)的数据

SELECT 列名称 FROM 表名称

注意：SQL语句中的关键字对大小写不敏感。

INSERT INTO语句用于向数据表中插入新的数据行

-- 语法解读：向指定的表中，插入如下几列数据，列的值通过values一一指定

-- 注意：列和值要一一对应，多个列和多个值之间，使用英文逗号分隔

INSERT INTO 表名称 (列1,列2,....) VALUES (值1,值2,....)

Update语句用于修改表中的数据

-- 语法解读：

1. 用UPDATE指定要更新哪个表中的数据
2. 用SET 指定列对应的新值
3. 用WHERE指定更新的条件

UPDATE 表名称 SET 列名称=新值 WHERE 列名称=某值

UPDATE 表名称 SET 列名称1=新值,列名称2=新值 WHERE 列名称=某值

DELETE语句用于删除表中的行

-- 从指定的表中，根据WHERE条件，删除对应的数据行

DELETE FROM 表名称 WHERE 列名称=值

-- select \* from users

-- select username,password from users

-- insert into users (username,password) values ('tony stark','098123')

-- select \* from users

-- update users set password='888888' where id=4

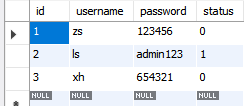
-- select \* from users

-- update users set password='admin123',status=1 where id=2

-- select \* from users

-- delete from users where id=4

-- select \* from users



### 额外SQL语法

WHERE子句用于限定选择的标准。在SELECT、UPDATE、DELETE语句中，皆可使用WHERE子句来限定选择的标准。

-- 查询语句中的WHERE条件

SELECT 列名称 FROM 表名称 WHERE 列 运算符 值

运算符

=等于 <>不等于 >大于 <小于 >=大于等于 <=小于等于

BETWEEN在某个范围内 LIKE搜索某种模式

注意：在某些版本的SQL中，操作符<>可以写为!=

AND和OR可在WHERE子语句中把两个或多个条件结合起来。

AND表示同时满足多个条件，相当于JavaScript中的&&运算符

OR表示只要满足任意一个条件即可，相当于JavaScript中的||运算符

-- select \* from users where status=1

-- select \* from users where id>2

-- select \* from users where username!="ls"

-- select \* from users where username<>"ls"

-- select \* from users where id<3 and status=0

-- select \* from users where username='zs' or status=1

ORDER BY 语句用于根据指定的列对结果集进行排序(由上到下)

ORDER BY 语句默认按照升序对记录进行排序(ASC关键字)

如果您希望按照降序对记录进行排序，可以使用DESC关键字

-- select \* from users order by status

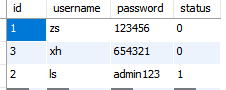
-- select \* from users order by status desc

-- select \* from users order by status asc

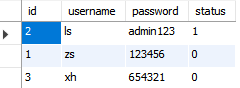
-- select \* from users order by id desc

-- select \* from users order by status desc,username asc

升序：



降序：



COUNT(\*)函数用于返回查询结果的总数据条数

-- select count(\*) from users where status=0

如果希望给查询出来的列名称设置别名，可以使用AS关键字

-- select count(\*) as total from users where status=0



第一个创建编写SQL语言文本，第二个打开保存的.sql文件

## mysql模块

### 安装并配置mysql模块

1. 安装操作MySQL数据库的第三方模块(mysql)
2. 通过mysql模块连接到MySQL数据库
3. 通过mysql模块执行SQL语句

mysql模块是托关于npm上的第三方模块。它提供了在Node.js项目中连接和操作MySQL数据库的能力。

安装npm install mysql

在使用mysql模块操作MySQL数据库之前，必须先对mysql模块进行必要的配置

// 1. 导入 mysql 模块

const mysql = require('mysql')

// 2. 建立与 MySQL 数据库的连接关系

const db = mysql.createPool({

  host: '127.0.0.1', // 数据库的 IP 地址

  user: 'root', // 登录数据库的账号

  password: '123456', // 登录数据库的密码

  database: 'my\_db\_01', // 指定要操作哪个数据库

})

测试mysql模块能否正常工作

调用db.query()函数，指定要执行的SQL语句，通过回调函数拿到执行的结果

// 测试 mysql 模块能否正常工作

/\* db.query('select 1', (err, results) => {

  // mysql 模块工作期间报错了

  if(err) return console.log(err.message)

  // 能够成功的执行 SQL 语句

//只要能打印出[RowDataPacket{‘1’:1}]的结果，就证明数据库连接正常

  console.log(results)

}) \*/



### 查询和插入数据

// 查询 users 表中所有的数据

/\* const sqlStr = 'select \* from users'

db.query(sqlStr, (err, results) => {

  // 查询数据失败

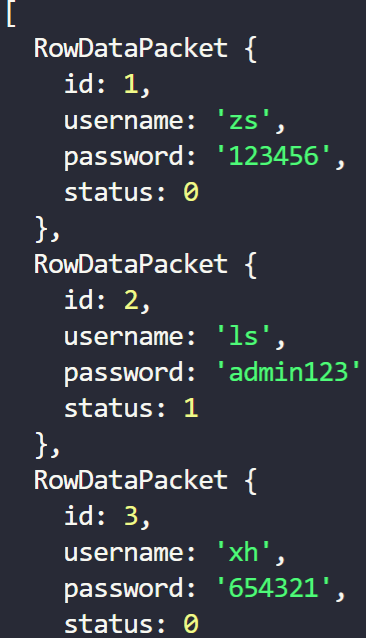
  if (err) return console.log(err.message)

  // 查询数据成功

  // 注意：如果执行的是 select 查询语句，则执行的结果是数组

  console.log(results)

}) \*/



插入数据

// 向 users 表中，新增一条数据，其中 username 的值为 Spider-Man，password 的值为 pcc123

//要插入到users表中的数据对象

/\* const user = { username: 'Spider-Man', password: 'pcc123' }

// 定义待执行的 SQL 语句

//待执行的SQL语句，其中英文的?表示占位符

const sqlStr = 'insert into users (username, password) values (?, ?)'

// 执行 SQL 语句

//使用数组的形式，依次为?占位符指定具体的值

db.query(sqlStr, [user.username, user.password], (err, results) => {

  // 执行 SQL 语句失败了

  if (err) return console.log(err.message)

  // 成功了

  // 注意：如果执行的是 insert into 插入语句，则 results 是一个对象

  // 可以通过 affectedRows 属性，来判断是否插入数据成功

  if (results.affectedRows === 1) {

    console.log('插入数据成功!')

  }

}) \*/

因为插入的是个对象不是一个数组，查询返回的是一个数组

插入数据的便捷方式：向表中新增数据时，如果数据对象的每个属性和数据表的字段一一对应，则可以通过如下方式快速插入数据

// 演示插入数据的便捷方式

//要插入到users表中的数据对象

/\* const user = { username: 'Spider-Man2', password: 'pcc4321' }

// 定义待执行的 SQL 语句

//待执行的SQL语句，其中英文的?表示占位符

const sqlStr = 'insert into users set ?'

// 执行 SQL 语句

//直接将数据对象当作占位符的值

db.query(sqlStr, user, (err, results) => {

  if (err) return console.log(err.message)

  if (results.affectedRows === 1) {

    console.log('插入数据成功')

  }

}) \*/

### 更新和删除数据

更新数据

// 演示如何更新用户的信息

//要更新的数据对象

/\* const user = { id: 6, username: 'aaa', password: '000' }

// 定义 SQL 语句

const sqlStr = 'update users set username=?, password=? where id=?'

// 执行 SQL 语句

//调用db.query()执行SQL语句的同时，使用数组依次为占位符指定具体的值

db.query(sqlStr, [user.username, user.password, user.id], (err, results) => {

  if (err) return console.log(err.message)

  // 注意：执行了 update 语句之后，执行的结果，也是一个对象，可以通过 affectedRows 判断是否更新成功

  if (results.affectedRows === 1) {

    console.log('更新成功')

  }

}) \*/

更新表数据时，如果数据对象的每个属性和数据表的字段一一对应，则可以通过如下方式快速更新表数据

// 演示更新数据的便捷方式

/\* const user = { id: 8, username: 'aaaa', password: '0000' }

// 定义 SQL 语句

const sqlStr = 'update users set ? where id=?'

// 执行 SQL 语句

db.query(sqlStr, [user, user.id], (err, results) => {

  if (err) return console.log(err.message)

  if (results.affectedRows === 1) {

    console.log('更新数据成功')

  }

})

\*/

删除数据

在删除数据时，推荐根据id这样的唯一标识，来删除对应的数据。

// 删除 id 为 5 的用户

/\* const sqlStr = 'delete from users where id=?'

//调用db.query()执行SQL语句的同时，为占位符指定具体的值

//如果SQL语句中有多个占位符，则必须使用数组为每个占位符指定具体的值

//如果SQL语句中只有一个占位符，则可以省略数组

db.query(sqlStr, 5, (err, results) => {

  if (err) return console.log(err.message)

  // 注意：执行 delete 语句之后，结果也是一个对象，也会包含 affectedRows 属性

  if (results.affectedRows === 1) {

    console.log('删除数据成功')

  }

}) \*/

标记删除

使用DELETE语句，会把真正的把数据从表中删除掉，为了保险起见，推荐使用标记删除的形式，来模拟删除的动作。

所谓的标记删除，就是在表中设置类似于status这样的状态字段，来标记当前这条数据是否被删除。

当用户执行了删除的动作时，我们并没有执行DELETE语句把数据删除掉，而是执行了UPDATE语句，将这条数据对应的status字段标记为删除即可。

// 标记删除

// const sqlStr = 'update users set status=? where id=?'

// db.query(sqlStr, [1, 6], (err, results) => {

//   if (err) return console.log(err.message)

//   if (results.affectedRows === 1) {

//     console.log('标记删除成功')

//   }

// })

## web

### web开发模式

目前主流的Web开发模式有两种，分别是：

1. 基于服务端渲染的传统Web开发模式
2. 基于前后端分离的新型Web开发模式

服务端渲染的Web开发模式：服务器发送给客户端的HTML页面，是在服务器通过字符串的拼接，动态生成的。因此，客户端不需要使用Ajax这样的技术额外请求页面的数据。

app.get(‘/index.html’,(req,res)=>{

//要渲染的数据

const user={name:’zs’,age:20}

//服务器端通过字符串的拼接，动态生成HTML内容

const html=’<h1>姓名：${user.name},年龄：${user.age}</h1>’

//把生成好的页面内容响应给客户端。因此，客户端拿到的是带有真实数据的//HTML页面

res.send(html)

})

服务端渲染的优缺点

优点：

1. 前端耗时少。因为服务器端负责动态生成HTML内容，浏览器只需要直接渲染页面即可。尤其是移动端，更省电。
2. 有利于SEO。因为服务器端响应是完整的HTML页面内容，所以爬虫更容易爬取去获得信息，更有利于SEO。

缺点：

1. 占用服务器端资源。即服务器端完成HTML页面内容的拼接，如果请求较多，会对服务器造成一定的访问压力。
2. 不利于前后端分离，开发效率低。使用服务器端渲染，则无法进行分工合作，尤其对于前端复杂度高的项目，不利于项目高效开发。

前后端分离的Web开发模式：依赖于Ajax技术的广泛应用。简而言之，前后端分离的Web开发模式，就是后端只负责提供API接口，前端使用Ajax调用接口的开发模式。

前后端分离的优缺点：

1. 开发体验好。前端专注于UI页面的开发，后端专注于api的开发，且前端有更多的选择性。
2. 用户体验好。Ajax技术的广泛应用，极大的提高了用户的体验，可以轻松实现页面的局部刷新。
3. 减轻了服务器端的渲染压力。因页面最终是在每个用户的浏览器中生成的。

缺点：

1. 不利于SEO。因为完整的HTML页面需要在客户端动态拼接完成，所以爬虫对无法爬取页面的有效信息。(解决方案：利用Vue、React等前端框架SSR(server side render)技术能够很好的解决SEO问题)

如何选择Web开发模式

不谈业务场景而盲目选择使用何种开发模式都是耍流氓

比如企业级网站，主要功能是展示而没有复杂的交互，并且需要良好的SEO，则这时我们就需要使用服务器端渲染；

而类似后台管理项目，交互性比较强，不需要考虑SEO，那么就可以使用前后端分离的开发模式。

另外，具体使用何种开发模式并不是绝对的，为了同时兼顾首页的渲染速度和前后端分离的开发效率，一些网站采用了首屏服务器端渲染+其他页面前后端分离的开发模式。

### 身份验证的概念

身份验证(Authentication)又称”身份验证”、”鉴权”，是指通过一定的手段，完成对用户身份的确认。

在Web开发中，也涉及到用户身份的认证，例如：各大网站的手机验证码登录、邮箱密码登录、二维码登录等。

身份认证的目的，就是为了确认当前所声称为某种身份的用户，确实是所声称的用户。例如，你去找快递员取快递，你要怎样证明这份快递是你的。

对于服务端渲染和前后端分离这两种开发模式来说，分别有着不同的身份认证方案：

1. 服务端渲染推荐使用Session认证机制
2. 前后端分离推荐使用JWT认证机制

### session

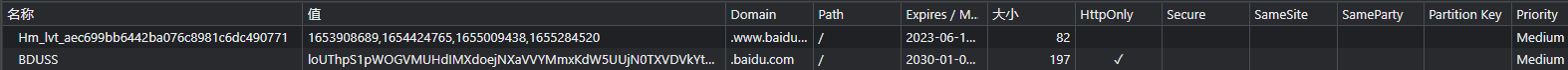
HTTP协议的无状态性，指的是客户端的每次HTTP请求都是独立的，连续多个请求之间没有直接的关系，服务器不会主动保留每次HTTP请求的状态。

如何突破HTTP无状态的限制

现实生活中的会员卡身份认证方式，在Web开发中的专业术语叫做Cookie

什么是Cookie

Cookie是存储在用户浏览器中的一般不超过4KB的字符串。它由一个名称(Name)、一个值(Value)和其他几个用于控制Cookie有效期、安全性、使用范围的可选属性组成。应用->存储->Cookie



不同域名下的Cookie各自独立，每当客户端发起请求时，会自动把当前域名下所有未过期的Cookie一同发送到服务器。

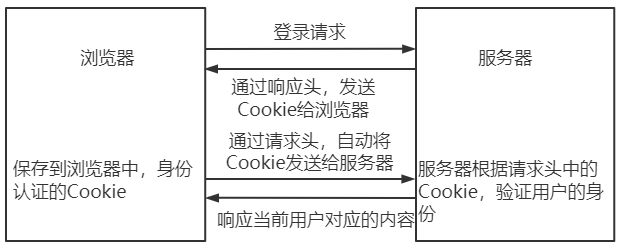
Cookie的几大特性：

1. 自动发送
2. 域名独立
3. 过期时限
4. 4KB限制

Cookie在身份认证中的作用

客户端第一次请求服务器的时候，服务器通过响应头的形式，向客户端发送一个身份认证的Cookie，客户端自动将Cookie保存在浏览器中。

随后，当客户端浏览器每次请求服务器的时候，浏览器会自动将身份验证相关的Cookie，通过请求头的形式发送给服务器，服务器即可验明客户端的身份。



网络->www.baidu.com

请求头中的Cookie

BIDUPSID=A704CF63EA6784574AA4C4E0FF3AFDCE;

PSTM=1604029909; \_\_yjs\_duid=1\_b38a2d56f71b3f88cfef862612c490931621689831487;

在应用->存储->Cookie里面一一对应

Cookie不具有安全性

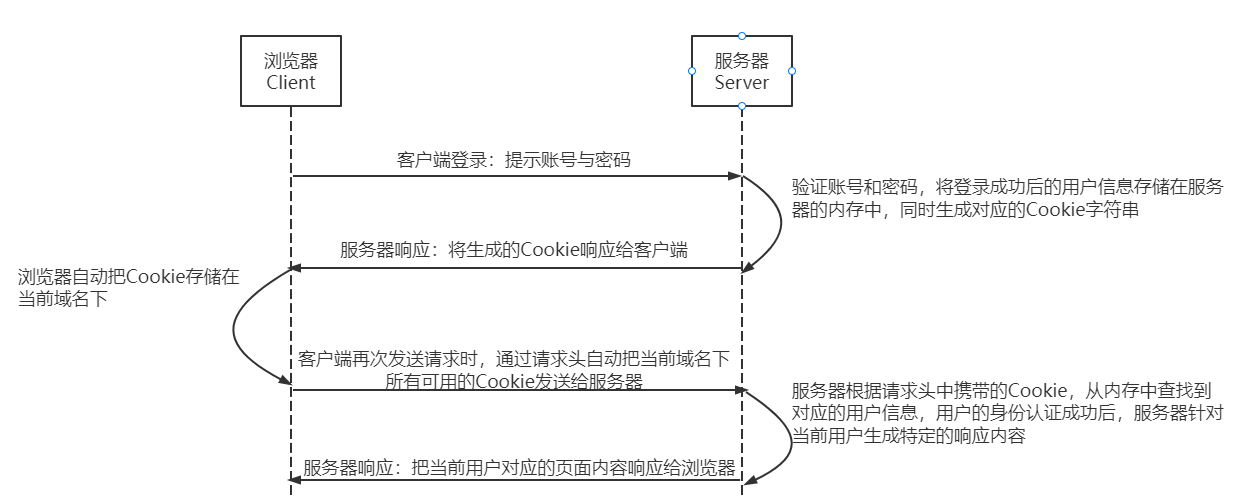
由于Cookie是存储在浏览器中的，而且浏览器也提供了读写Cookie的API，因此Cookie很容易被伪造，不具有安全性。因此不建议服务器将重要的隐私数据，通过Cookie的形式发送给浏览器。

千万不要使用Cookie存储重要且隐私的数据！比如用户的身份信息、密码等。

提高身份证的安全性

为了防止客户伪造会员卡，收银员在拿到客户出示的会员卡之后，可以在收银机上确认存在的会员卡，才能被正常使用。

这种”会员卡+刷卡认证”的设计理念，就是Session认证机制的精髓。



### session中间件的使用

在Express项目中，只需要安装express-session中间件，即可在项目中使用Session认证：

npm install express-session

配置express-session中间件

express-session中间件安装成功后，需要通过app.use()来注册session中间件

// TODO\_01：请配置 Session 中间件

const session = require('express-session')

app.use(

  session({

    secret: 'itheima',//secret属性可以为任意字符串

    resave: false,//固定写法

    saveUninitialized: true,//固定写法

  })

)

// 托管静态页面

app.use(express.static('./pages'))

// 解析 POST 提交过来的表单数据

app.use(express.urlencoded({ extended: false }))

D:\software\file\1.Node.js零基础入门教程\node.js—资料\day6\code\01.session案例>npm i express-session

向session中存数据

当express-session中间件配置成功后，即可通过req.session来访问和使用session对象，从而存储用户的关键信息：

// 登录的 API 接口

app.post('/api/login', (*req*, *res*) => {

  // 判断用户提交的登录信息是否正确

  if (*req*.body.username !== 'admin' || *req*.body.password !== '000000') {

    return *res*.send({ status: 1, msg: '登录失败' })

  }

  // TODO\_02：请将登录成功后的用户信息，保存到 Session 中

  // 注意：只有成功配置了 express-session 这个中间件之后，才能够通过 req 点出来 session 这个属性

*req*.session.user = *req*.body // 用户的信息

*req*.session.islogin = true // 用户的登录状态

*res*.send({ status: 0, msg: '登录成功' })

})

从session中获取数据

可以直接从req.session对象上获取之前存储的数据

// 获取用户姓名的接口

app.get('/api/username', (*req*, *res*) => {

  // TODO\_03：请从 Session 中获取用户的名称，响应给客户端

  if (!*req*.session.islogin) {

    return *res*.send({ status: 1, msg: 'fail' })

  }

*res*.send({

    status: 0,

    msg: 'success',

    username: *req*.session.user.username,

  })

})

清空session

调用req.session.destroy()函数，即可清空当前服务器保存的session信息。

// 退出登录的接口

app.post('/api/logout', (*req*, *res*) => {

  // TODO\_04：清空 Session 信息

*req*.session.destroy()

*res*.send({

    status: 0,

    msg: '退出登录成功',

  })

})

.serialize()从表单数据中获取参数

   // 发起 POST 登录请求

        $.post('/api/login', $(*this*).serialize(), function (*res*) {

          // status 为 0 表示登录成功；否则表示登录失败！

          if (*res*.status === 0) {

            location.href = './index.html'

          } else {

            alert('登录失败！')

          }

        })

### jwt-了解token的原理

了解Session认证的局限性

Session认证机制需要配合Cookie才能实现。由于Cookie默认不支持跨域访问，所以，当涉及到前端跨域请求后端接口的时候，需要做很多额外的配置，才能实现跨域Session认证。

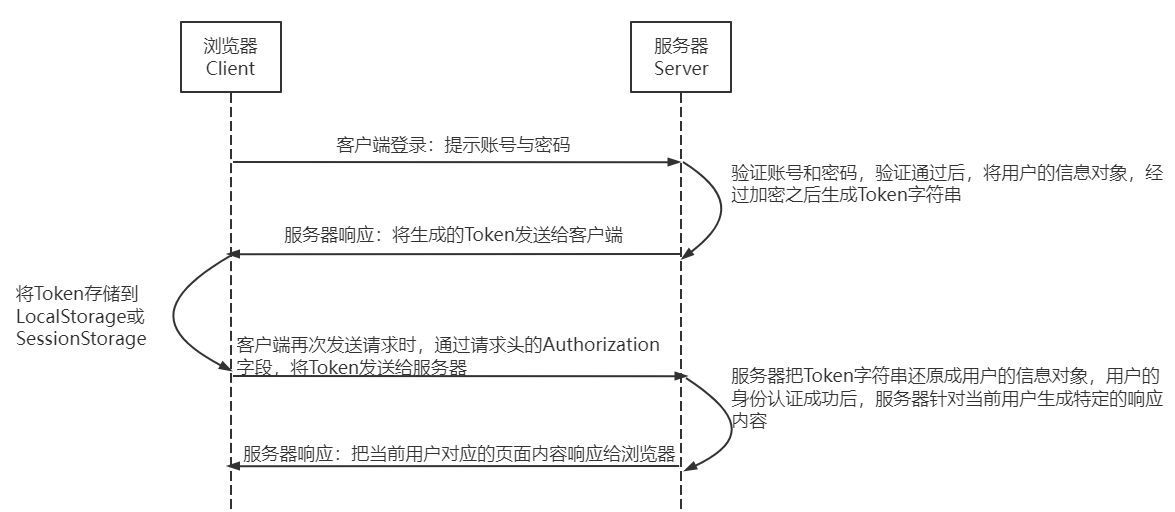
当前端请求后端接口不存在跨域问题的时候，推荐使用Session身份认证机制

当前端需要跨域请求后端接口的时候，不推荐使用Session身份认证机制，推荐是使用JWT认证机制。

什么是JWT

JWT(英文全称：JSON Web Token)是目前最流行的跨域认证解决方案。

JWT工作原理



总结：用户的信息通过Token字符串的形式，保存在客户端浏览器中。服务器通过还原Token字符串的形式来认证用户的身份。

区别：在JWT中用户信息保存在浏览器里面，在Session中用户信息保存在服务器里面。

JWT通常由三部分组成，分别是Header(头部)、Payload(有效荷载)、Signature(签名)。三者之间使用英文的”.”分割，格式如下：

Header.Payload.Signature

Payload部分才是真正的用户信息，它是用户信息经过加密之后生成的字符串。

Header和Signature是安全性相关的部分，只是为了保证Token的安全性。

{id:3,username:’admin’}---->用户信息经过加密后，就是Payload部分对应的字符串---->Header.Payload.Signature---->Header和Signature只是为了保证Token的安全性。

JWT的使用方式

客户端收到服务器返回的JWT之后，通常会将它储存在localStorage或sessionStorage中。此后，客户端每次与服务器通信，都要带上这个JWT的字符串，从而进行身份认证。推荐的做法是把JWT放在HTTP请求头的Authorization字段中，格式如下：

Authorization:Bearer<token>

### jwt-在express中生成token

安装JWT相关的包

npm install jsonwebtoken express-jwt

jsonwebtoken用于生成JWT字符串

express-jwt用于将JWT字符串解析还原成JSON对象

使用require()函数，分别导入JWT相关的两个包：

// TODO\_01：安装并导入 JWT 相关的两个包，分别是 jsonwebtoken 和 express-jwt

const jwt = require('jsonwebtoken')

const expressJWT = require('express-jwt')

// 允许跨域资源共享

const cors = require('cors')

app.use(cors())

// 解析 post 表单数据的中间件

const bodyParser = require('body-parser')

app.use(bodyParser.urlencoded({ extended: false }))

express.urlencoded()

返回仅解析urlencode主体并且仅查看Content-Type标头与type选项匹配的请求的中间件。该解析器仅接受主体的UTF-8编码，并支持gzip和deflate编码的自动填充。

body-parser中间件

先在中间件中解析传入的请求主体，然后再在处理程序中对其进行处理，该请求主体可在req.body属性下找到。express.urlencoded是基于Nodejs body-parser的。

定义secret密钥

为了保证JWT字符串的安全性，防止JWT字符串在网络传输过程中被别人破解，我们需要专门定义一个用于加密和解密的secret密钥：

1. 当生成JWT字符串的时候，需要使用secret密钥对用户信息进行加密，最终得到加密好的JWT字符串
2. 当把JWT字符串解析还原成JSON对象的时候，需要使用secret密钥进行解密

// TODO\_02：定义 secret 密钥，建议将密钥命名为 secretKey

const secretKey = 'itheima No1 ^\_^'

在登录成功后生成JWT字符串

调用jsonwebtoken包提供的sign()方法，将用户的信息加密成JWT字符串，响应给客户端：

// 登录接口

app.post('/api/login', function (*req*, *res*) {

  // 将 req.body 请求体中的数据，转存为 userinfo 常量

  const userinfo = *req*.body

  // 登录失败

  if (userinfo.username !== 'admin' || userinfo.password !== '000000') {

    return *res*.send({

      status: 400,

      message: '登录失败！',

    })

  }

  // 登录成功

  // TODO\_03：在登录成功之后，调用 jwt.sign() 方法生成 JWT 字符串。并通过 token 属性发送给客户端

  // 参数1：用户的信息对象

  // 参数2：加密的秘钥

  // 参数3：配置对象，可以配置当前 token 的有效期

  // 记住：千万不要把密码加密到 token 字符中

  const tokenStr = jwt.sign({ username: userinfo.username }, secretKey, { expiresIn: '30s' })

*res*.send({

    status: 200,

    message: '登录成功！',

    token: tokenStr, // 要发送给客户端的 token 字符串

  })

})

将JWT字符串还原为JSON对象

客户端每次在访问那些有权限接口的时候，都需要主动通过请求头中的Authorization字段，将Token字符串发送到服务器进行身份认证。

此时，服务器可以通过express-jwt这个中间件，自动将客户端发送过来的Token解析还原成JSON对象；

// TODO\_04：注册将 JWT 字符串解析还原成 JSON 对象的中间件

// 注意：只要配置成功了 express-jwt 这个中间件，就可以把解析出来的用户信息，挂载到 req.user 属性上

//expressJWT({ secret: secretKey })就是用来解析Token的中间件

//unless({ path: [/^\/api\//] })用来指定哪些接口不需要访问权限

app.use(expressJWT({ secret: secretKey }).unless({ path: [/^\/api\//] }))

当express-jwt这个中间件配置成功之后，即可在那些有权限的接口中，使用req.user对象，来访问从JWT字符串中解析出来的用户信息了。

// 这是一个有权限的 API 接口

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

  // TODO\_05：使用 req.user 获取用户信息，并使用 data 属性将用户信息发送给客户端

  console.log(*req*.user)

*res*.send({

    status: 200,

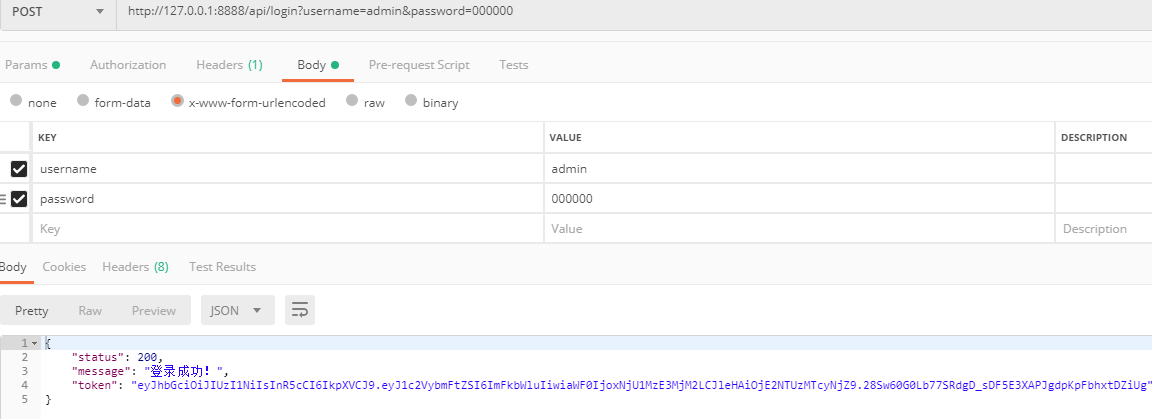
    message: '获取用户信息成功！',

    data: *req*.user, // 要发送给客户端的用户信息

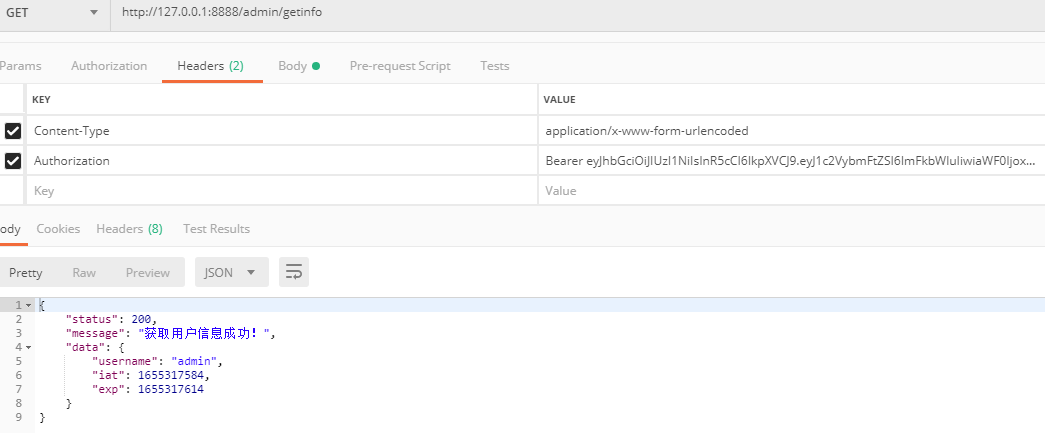
  })

})

先登录



登录成功后30s内将payload复制到authorization的value里面(Bearer)



捕获解析JWT失败后产生的错误

当使用express-jwt解析Token字符串时，如果客户端发送过来的Token字符串过期或不合法，会产生一个解决失败的错误，影响项目的正常运行。我们可以通过Express的错误中间件，捕获这个错误并运行这个错误进行相关的处理



// TODO\_06：使用全局错误处理中间件，捕获解析 JWT 失败后产生的错误

app.use((*err*, *req*, *res*, *next*) => {

  // 这次错误是由 token 解析失败导致的

  if (*err*.name === 'UnauthorizedError') {

    return *res*.send({

      status: 401,

      message: '无效的token',

    })

  }

*res*.send({

    status: 500,

    message: '未知的错误',

  })

})

# 项目

## 初始化项目

### 创建项目

创建api\_server文件，在终端中该文件目录下npm init -y

安装npm install [express@4.17.1](mailto:express@4.17.1)

创建app.js文件

导入express

创建服务器的实例对象

启动服务器

### 配置cors跨域

npm i [cors@2.8.5](mailto:cors@2.8.5)

导入配置cors中间件

### 配置解析表单数据的中间件

// 配置解析表单数据的中间件，注意：这个中间件，只能解析 application/x-www-form-urlencoded 格式的表单数据

app.use(express.urlencoded({ extended: false }))

### 初始化路由相关的文件夹

创建router文件夹(存放客户端的请求与处理函数之间的映射关系)

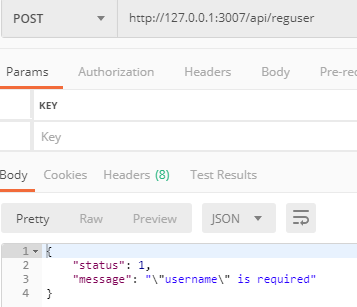
和router\_handler文件夹(负责存放每个路由对应的处理函数)

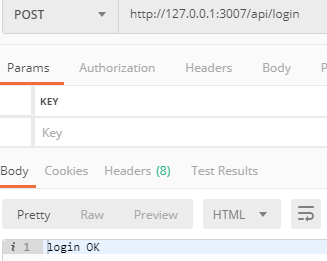
### 初始化用户路由模块

在router文件夹中，新建user.js文件，作为用户的路由模块，并初始化

在app.js中，导入并使用用户路由模块

Postman是一个[接口测试](https://so.csdn.net/so/search?q=%E6%8E%A5%E5%8F%A3%E6%B5%8B%E8%AF%95&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/youtiao_hulatang/article/details/_blank)工具，在做接口测试的时候，Postman相当于一个客户端。它可以模拟用户发起的各类HTTP/HTTPS请求，将请求数据发送至服务端，获取对应的响应结果，从而验证响应中的结果数据是否和预期值相匹配。





### 抽离用户路由模块中的处理函数

为了保证路由模块的纯粹性

新建/router\_handler/user.js，使用exports对象，分别向外共享两个路由处理函数

// 登录的处理函数

*exports*.login = (*req*, *res*) => {

*res*.send('login OK')

}

在router/user.js中

// 导入用户路由处理函数对应的模块

const user\_handler = require('../router\_handler/user')

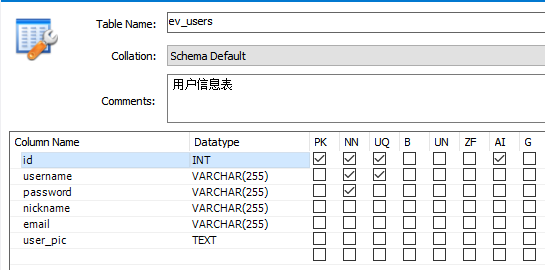
// 登录

router.post('/login', user\_handler.login)

ES6用的是export，commonJS用的是exports

## 登录和注册

### 新建ev\_users表



### 安装并配置mysql模块

安装npm i [mysql@2.18.1](mailto:mysql@2.18.1)

在项目根目录中新建/db/index.js文件

const mysql = require('mysql')

const db = mysql.createPool({

  host: '127.0.0.1',

  user: 'root',

  password: '123456',

  database: 'my\_db\_01',

})

*module*.*exports* = db

在router\_handler/user.js里面导入

// 导入数据库操作模块

const db = require('../db/index')

### 注册(API接口)

实现步骤

1. 检测表单数据是否合法
2. 检测用户名是否被占用
3. 对密码进行加密处理
4. 插入新用户

一、检测表单数据是否合法

判断用户名和密码是否为空

在router\_handler/user.js进行处理

注册新用户的处理函数

获取客户端提交到服务器的用户信息

const userinfo = *req*.body

对表单中的数据，进行合法性的校验

if (!userinfo.username || !userinfo.password)

1. 检测用户名是否被占用

导入数据库操作模块：

// 定义 SQL 语句，查询用户名是否被占用

  const sqlStr = 'select \* from ev\_users where username=?'

执行SQL语句

db.query(sqlStr, userinfo.username, (*err*, *results*) => {}

执行SQL语句失败

 if (*err*)

用户名被占用

 if (*results*.length > 0)

1. 对密码进行加密处理

为了保证密码的安全性，不建议在数据库以明文的形式保存用户密码，推荐对密码进行加密存储

在当前项目中，使用bcryptjs对用户密码进行加密，优点：

加密之后的密码，无法被逆向破解

同一明文密码多次加密，得到的加密结果各不相同，保证了安全性

安装：npm i [bcryptjs@2.4.3](mailto:bcryptjs@2.4.3)

在/router\_handler/user.js中导入bcryptjs

在注册用户的处理函数中，确认用户名可用之后，调用bcryptjs.hashSync(明文密码,随机盐的长度)方法，对用户的密码进行加密处理

// 调用 bcrypt.hashSync() 对密码进行加密

    userinfo.password = bcrypt.hashSync(userinfo.password, 10)

1. 插入新用户

// 定义插入新用户的 SQL 语句

    const sql = 'insert into ev\_users set ?'

    // 调用 db.query() 执行 SQL 语句

    db.query(sql, { username: userinfo.username, password: userinfo.password }, (*err*, *results*) => {}

判断SQL语句是否执行成功

 if (*err*)

判断影响行数是否为1

 if (*results*.affectedRows !== 1)

注册用户成功

### 代码优化

优化res.send()代码

在处理函数中，需要多次调用res.send()向客户端响应处理失败的结果，可以手动封装一个res.cc()函数。

在app.js中，所有路由之前，声明一个全局中间件，为res对象挂载一个res.cc()函数

// 一定要在路由之前，封装 res.cc 函数

app.use((*req*, *res*, *next*) => {

  // status 默认值为 1，表示失败的情况

  // err 的值，可能是一个错误对象，也可能是一个错误的描述字符串

*res*.cc = function (*err*, *status* = 1) {

*res*.send({

      status,

      message: *err* instanceof *Error* ? *err*.message : *err*,

    })

  }

  next()

})

res.send({ status: 1, message: err.message })

res.send({ status: 1, message: '用户名被占用，请更换其他用户名！' })

res.send({ status: 1, message: err.message })

res.send({ status: 1, message: '注册用户失败，请稍后再试！' })

// res.send({ status: 0, message: '注册成功！' })

优化表单数据验证

表单验证的原则：前端验证为辅，后端验证为主，后端永远不要相信前端提交过来的任何内容

在实际开发中，前后端都需要对表单的数据进行合法性的验证，而且，后端作为数据合法性验证的最后一个关口，在拦截非法数据方面，起到了至关重要的作用。

单纯的使用if...else...的形式对数据合法性进行验证，效率低下、出错率高、维护性差。因此，推荐使用第三方数据验证模块，来降低出错率、提高验证的效率与可维护性，让后端程序员把更多的精力放在核心业务逻辑的处理上。

安装@hapi/joi包，为表单中携带的每个数据项，定义验证规则：

npm install @hapi/joi@17.1.0

安装@escook/express-joi中间件，来实现自动对表单数据进行验证的功能：

npm i @escook/express-joi

在app.js文件中

在 app.js 的全局错误级别中间件中，捕获验证失败的错误，并把验证失败的结果响应给客户端：

// 导入定义验证规则的包

const joi = require('@hapi/joi')

// 定义错误级别的中间件

app.use((*err*, *req*, *res*, *next*) => {

  // 验证失败导致的错误

  if (*err* instanceof *joi*.ValidationError) return *res*.cc(*err*)

  // 未知的错误

*res*.cc(*err*)

})

在router/user.js中

// 1. 导入验证数据的中间件

const expressJoi = require('@escook/express-joi')

// 2. 导入需要的验证规则对象

const { reg\_login\_schema } = require('../schema/user')

新建/schema/user.js用户信息验证规则模块，并初始化代码如下

// 导入定义验证规则的包

const joi = require('@hapi/joi')

// 定义用户名和密码的验证规则

const username = joi.string().alphanum().min(1).max(10).required()

const password = joi

  .string()

  .pattern(/^[\S]{6,12}$/)

  .required()

// 定义验证注册和登录表单数据的规则对象

*exports*.reg\_login\_schema = {

  body: {

    username,

    password,

  },

}

// 2.1 校验 req.body 中的数据

body: {

username: Joi.string().alphanum().min(3).max(12).required(),

password: Joi.string()

.pattern(/^[\S]{6,15}$/)

.required(),

repassword: Joi.ref('password')

},

用户名必须为字符串由数字和字母组成长度3-9之间

密码必须为字符串不包含空格长度6-15之间

重复密码必须与密码一致

// 3. 在路由中通过 expressJoi(userSchema) 的方式

// 调用中间件进行参数验证

app.post('/adduser/:id', expressJoi(userSchema), function (req, res) {

const body = req.body

res.send(body)})

// 4.1 错误级别中间件app.use(function (err, req, res, next) {

// 4.1 Joi 参数校验失败

if (err instanceof Joi.ValidationError) {

return res.send({

status: 1,

message: err.message

})

}

// 4.2 未知错误

res.send({

status: 1,

message: err.message

})})

### body、query、params对比

表单数据req.body

查询字符串的数据req.query

url所携带的数据req.params

req.params,req.query是用在get请求当中，而req.body是用在post请求中的

1. req.params

app.get('/api/getUserListById/:id', (req, res) => {

console.log('id is:' + req.params.id);

})

IMG_256

通过req.params.id，我们会得到 1

2. req.query

http://localhost:3000/?id=2,用req.query.id,我们会得到 2，如果有两个或以上参数，用 & 连接，如：/?id=2&name=node, 获取参数则是：req.query.id 得到 1 , req.query.name 得到 node .

3. req.body

req.body是用在post请求当中的

那么req.body.name就会得到表单输入的登录名，req.body.password就会得到表单输入的密码.

### 登录

实现步骤

1. 检测表单数据是否合法
2. 根据用户名查询用户的数据
3. 判断用户输入的密码是否正确
4. 生成JWT的Token字符串
5. 检测登录表单的数据

将/router/user.js中的登录代码修改如下：

// 登录

router.post('/login', expressJoi(reg\_login\_schema), user\_handler.login)

1. 根据用户名查询用户的数据

接收表单数据

定义SQL语句

执行SQL语句，查询用户的数据

 // 执行 SQL 语句成功，但是获取到的数据条数不等于 1

    if (*results*.length !== 1) return *res*.cc('登录失败！')

1. 判断用户输入的密码是否正确

核心实现思路：调用bcrypt.compareSync(用户提交的密码，数据库中的密码)方法比较密码是否一致，返回值是布尔值(true一致、false不一致)

// TODO：判断密码是否正确

const compareResult = bcrypt.compareSync(userinfo.password, *results*[0].password)

//如果对比的结果等于false，则证明用户输入的密码错误

    if (!compareResult) return *res*.cc('登录失败！')

### Token的生成与验证

生成JWT的Token字符串

核心注意点：在生成Token字符串的时候，一定要剔除密码和头像的值

通过ES6的高级语法，快速剔除密码和头像的值：

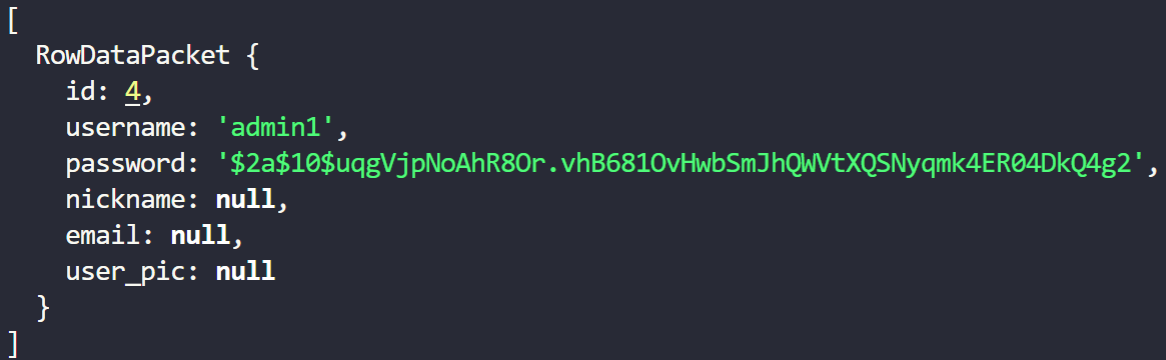
删除完之后，user中只保留了用户的id,username,nickname,email的值

通过展开运算符展开user，用password和user\_pic将展开的user里面的覆盖掉

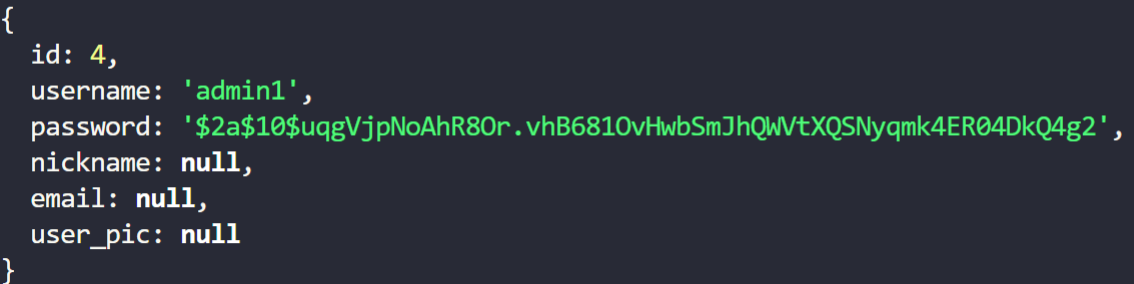
    // TODO：在服务器端生成 Token 的字符串

const user = { ...*results*[0], password: '', user\_pic: '' }

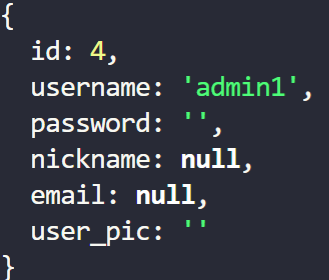
打印results



打印{...results[0]}



打印user



安装生成token字符串的包：

npm i [jsonwebtoken@8.5.1](mailto:jsonwebtoken@8.5.1)

在/router\_handler/user.js模块的头部区域导入jsonwebtoken包

// 导入生成 Token 的包

const jwt = require('jsonwebtoken')

创建config.js文件，并向外分享加密和还原Token的jwtSecretKey字符串

// 这是一个全局的配置文件

*module*.*exports* = {

  // 加密和解密 Token 的秘钥

  jwtSecretKey: 'itheima No1. ^\_^',

  // token 的有效期

  expiresIn: '10h',

}

导入config.js配置文件

// 对用户的信息进行加密，生成 Token 字符串

    const tokenStr = jwt.sign(user, config.jwtSecretKey, { expiresIn: config.expiresIn })

    // 调用 res.send() 将 Token 响应给客户端

*res*.send({

      status: 0,

      message: '登录成功！',

      token:  'Bearer ' + tokenStr,

    })

配置解析Token的中间件

在app.js中注册路由之前，配置解析Token的中间件

安装npm i express-jwt@5.3.3

const expressJWT = require('express-jwt')

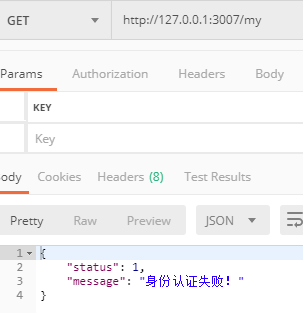
const config = require('./config')

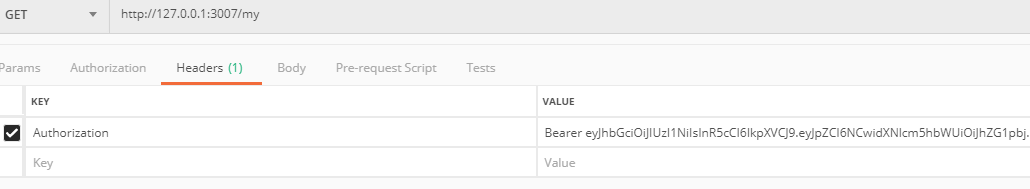
app.use(expressJWT({ secret: config.jwtSecretKey }).unless({ path: [/^\/api/] }))

判断Headers中有没有携带Authorization的字段

// 身份认证失败后的错误

  if (*err*.name === 'UnauthorizedError') return *res*.cc('身份认证失败！')





没有这个接口

@hapi/joi

@escook/express-joi

jsonwebtoken

express-jwt

Bcryptjs

bcrypt.hashSync

bcrypt.compareSync

## 个人中心

### 获取用户的基本信息

1.初始化路由模块

2.初始化路由处理函数模块

3.获取用户的基本信息

一、初始化路由模块

创建/router/user/userinfo.js路由模块，并初始化如下的代码结构

导入express

创建路由对象

获取用户基本信息的路由

向外共享路由对象

在app.js中导入并使用用户信息的路由模块

1. 初始化路由处理函数模块

创建/router\_handler/userinfo.js路由处理函数模块，并初始化如下的代码结构

// 获取用户基本信息的处理函数

*exports*.getUserInfo = (*req*, *res*) => {}

1. 获取用户的基本信息

在/router\_handler/userinfo.js头部导入数据库操作模块

定义SQL语句

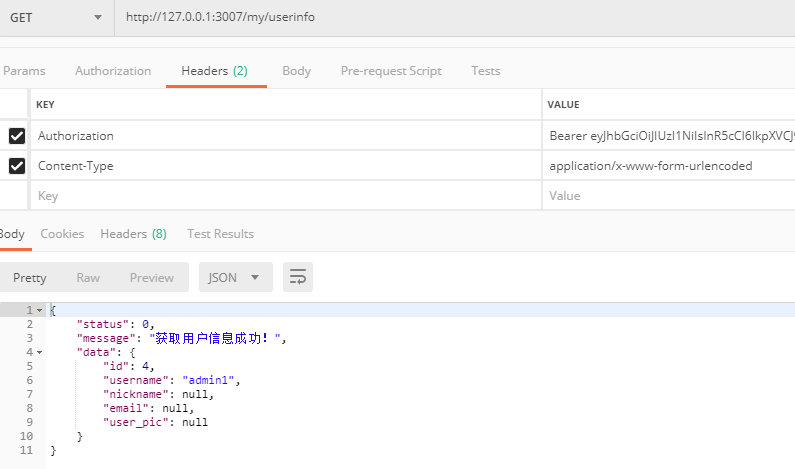
根据用户的id查询用户的基本信息

为了防止用户的密码泄露，需要排除password字段

 const sql = `select id, username, nickname, email, user\_pic from ev\_users where id=?`

调用db.query()执行SQL语句

 db.query(sql, *req*.user.id, (*err*, *results*) => {}



### 更新用户的基本信息

1. 定义路由和处理函数
2. 验证表单数据
3. 实现更新用户基本信息的功能
4. 定义路由和处理函数

在/router/userinfo.js模块中，新增更新用户基本信息的路由

在/router\_handler/userinfo.js模块中，定义并向外共享，更新用户基本信息的路由处理函数

1. 验证表单数据

在/schema/user.js验证规则模块中，定义id,nickname,email的验证规则如下：

// 定义 id, nickname, email 的验证规则

const id = joi.number().integer().min(1).required()

const nickname = joi.string().required()

const user\_email = joi.string().email().required()

// 验证规则对象 - 更新用户基本信息

*exports*.update\_userinfo\_schema = {

// 需要对 req.body 里面的数据进行验证

}

在/router/userinfo.js模块中，导入验证数据合法性的中间件

// 导入验证数据的中间件

const expressJoi = require('@escook/express-joi')

// 导入需要的验证规则对象

const { update\_userinfo\_schema} = require('../schema/user')

// 更新用户信息的路由

router.post('/userinfo', expressJoi(update\_userinfo\_schema), userinfo\_handler.updateUserInfo)

在/router\_handler/userinfo.js中

// 定义待执行的 SQL 语句

  const sql = `update ev\_users set ? where id=?`

  // 调用 db.query() 执行 SQL 语句并传递参数

  db.query(sql, [*req*.body, *req*.body.id], (*err*, *results*) => {}

### 重置密码

1. 定义路由和处理函数
2. 验证表单数据
3. 实现重置密码的功能
4. 定义路由和处理函数

在/router/userinfo.js模块中，新增重置密码的路由

// 更新密码的路由

router.post('/updatepwd', expressJoi(update\_password\_schema), userinfo\_handler.updatePassword)

在/router\_handler/userinfo.js模块中，定义并向外分享重置密码的路由处理函数

1. 验证表单数据

核心验证思路：旧密码与新密码，必须符合密码的验证规则，并且新密码不能与旧密码一致。

在/schema/user.js模块中，使用exports向外共享如下的验证规则对象

// 验证规则对象 - 更新密码

*exports*.update\_password\_schema = {

  body: {

oldPwd: password,

//joi.ref(‘oldPwd’)表示newPwd的值必须和oldPwd的值保持一致

//joi.not(joi.ref(‘oldPwd’))表示newPwd的值不能等于oldPwd的值

//.concat()用于合并joi.not(joi.ref(‘oldPwd’))和password两条验证规则

    newPwd: joi.not(joi.ref('oldPwd')).concat(password),

  },

}

在/router/userinfo.js模块中导入需要的验证规则对象

根据id查询用户数据的SQL语句是否存在

// 根据 id 查询用户的信息

  const sql = `select \* from ev\_users where id=?`

 db.query(sql, *req*.user.id, (*err*, *results*) => {}

即可使用bcrypt.compareSync(提交的密码,数据库中的密码)方法验证密码是否正确，compareSync()函数的返回值为布尔值，true表示密码正确，false表示密码错误。

 // 判断密码是否正确

    const compareResult = bcrypt.compareSync(*req*.body.oldPwd, *results*[0].password)

    if (!compareResult) return *res*.cc('旧密码错误！')

对新密码进行bcrypt加密之后，更新到数据库中

// 定义更新密码的 SQL 语句

    const sql = `update ev\_users set password=? where id=?`

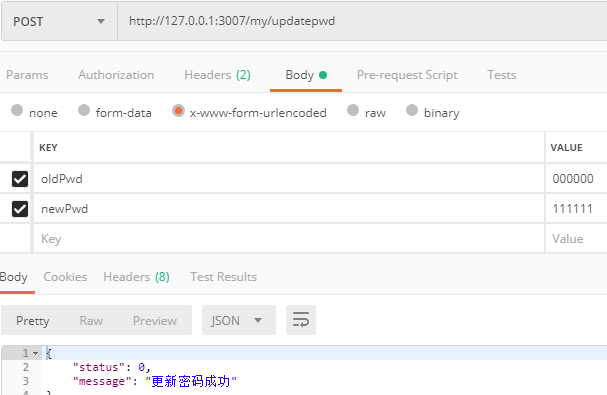
    // 对新密码进行加密处理

    const newPwd = bcrypt.hashSync(*req*.body.newPwd, 10)

    // 调用 db.query() 执行 SQL 语句

    db.query(sql, [newPwd, *req*.user.id], (*err*, *results*) => {}

req.user.id为验证身份成功了就会往req挂载一个用户的信息user对象(解析token后自动挂载的，现在是req.authorization，获取token里的用户信息)



### 更换头像

1. 定义路由和处理函数
2. 验证表单数据
3. 实现更新用户头像的功能
4. 定义路由和处理函数

在/router/userinfo.js模块中，新增更新用户头像的路由

在/router\_handler/userinfo.js模块中，定义并向外共享更新用户头像的路由处理函数

1. 验证表单数据

在/schema/user.js验证规则模块中，定义avatar的验证规则如下：

// dataUri() 指的是如下格式的字符串数据：

// data:image/png;base64,VE9PTUFOWVNFQ1JFVFM=

// 定义验证 avatar 头像的验证规则

const avatar = joi.string().dataUri().required()

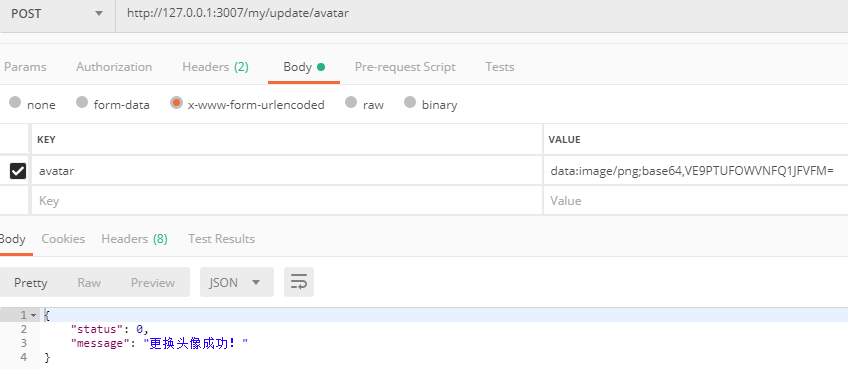
并使用exports向外共享如下的验证规则对象

在/router/userifo.js模块中，导入需要的验证规则对象

在/router/userinfo.js模块中，修改更新用户头像的路由如下：

// 更换头像的路由

router.post('/update/avatar', expressJoi(update\_avatar\_schema), userinfo\_handler.updateAvatar)



// 1. 定义更新头像的 SQL 语句

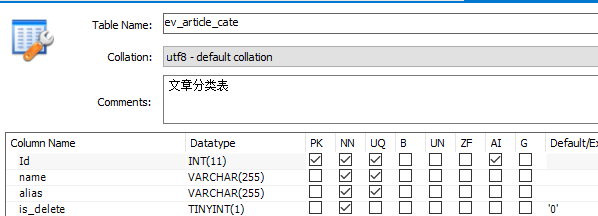
  const sql = `update ev\_users set user\_pic=? where id=?`

  // 2. 调用 db.query() 执行 SQL 语句

  db.query(sql, [*req*.body.avatar, *req*.user.id], (*err*, *results*) => {}

## 文章分类管理

### 新建ev\_article\_cate表





新增两条初始数据



### 获取文章分类列表的接口

1. 初始化路由模块

创建/router/artcate.js路由模块，并初始化如下的代码结构

导入文章分来的路由处理函数模块

// 获取文章分类列表数据的路由

router.get('/cates', artCate\_handler.getArtCates)

在app.js中

// 导入并使用文章分类的路由模块

const artCateRouter = require('./router/artcate')

app.use('/my/article', artCateRouter)

1. 初始化路由处理函数模块

在/router\_handler/artcate.js路由处理函数模块

// 获取文章分类列表的处理函数

*exports*.getArtCates = (*req*, *res*) => {

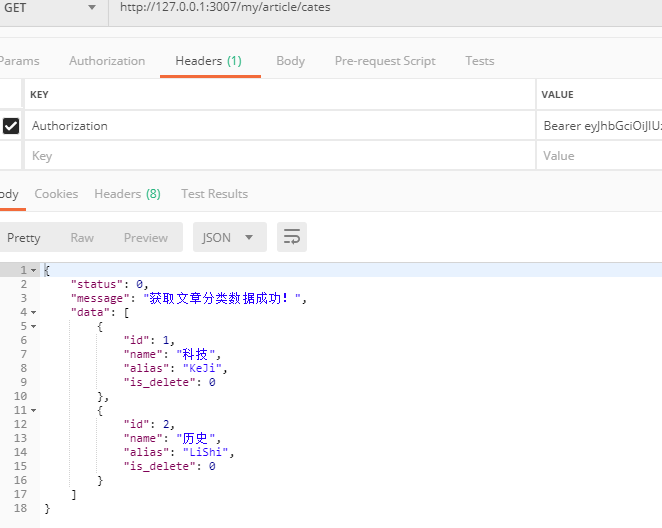
  // 定义查询分类列表数据的 SQL 语句

  const sql = `select \* from ev\_article\_cate where is\_delete=0 order by id asc`

}

1. 获取文章分类列表数据

在/router\_handler/artcate.js头部导入数据库操作



### ES6解构赋值

ES6之前的代码是如何实现的。无非是通过点(.)语法或者下标([])的方式来获取对象的属性值，代码分别如下。

通过点语法(.)获取对象的属性值。

const user = {

userName: "光脚丫思考",

blog: "https://blog.csdn.net/gjysk"

}

console.log(user.userName);

console.log(user.blog);

通过下标([])方式获取对象属性值的代码如下：

const user = {

userName: "光脚丫思考",

blog: "https://blog.csdn.net/gjysk"

}

console.log(user["userName"]);

console.log(user["blog"]);

可以看到，下标中的字符串，实际上就是属性的名称。

当然了，如果与ES6语法比较起来，上面两种玩法都还是显的就没有那么简洁了。接下来，我们看看ES6是如何来获取对象的属性值的？

const user = {

userName: "光脚丫思考",

blog: "https://blog.csdn.net/gjysk"

}

const {userName, blog} = user;

console.log(userName);

console.log(blog);

上述代码演示了如何通过ES6语法的解构赋值来快速地获取属性值。示例中创建与对象属性名同名的变量，并将对象中同名的属性值分配给新创建的同名变量，大大地减少了许多重复的代码。

### 新增文章分类

1. 定义路由和处理函数

在/router/artcate.js模块中，添加新增文章分类的路由

解构赋值出add\_cate\_schema里面的对象

// 导入需要的验证规则对象

const { add\_cate\_schema, delete\_cate\_schema, get\_cate\_schema, update\_cate\_schema } = require('../schema/artcate')

// 新增文章分类的路由

router.post('/addcates', expressJoi(add\_cate\_schema), artCate\_handler.addArticleCates)

1. 验证表单数据

在/schema/artcate.js

// 1. 导入定义验证规则的模块

const joi = require('@hapi/joi')

// 2. 定义 name 和 alias 的验证规则

const name = joi.string().required()

const alias = joi.string().alphanum().required()

1. 查询分类名称与分类别名是否被占用

// 新增文章分类的处理函数

*exports*.addArticleCates = (*req*, *res*) => {

  // 1. 定义查重的 SQL 语句

  const sql = `select \* from ev\_article\_cate where name=? or alias=?`

  // 2. 执行查重的 SQL 语句

  db.query(sql, [*req*.body.name, *req*.body.alias], (*err*, *results*) => {}

// 4.1 判断数据的 length

    if (*results*.length === 2) return *res*.cc('分类名称与分类别名被占用，请更换后重试！')

    // 4.2 length 等于 1 的三种情况

    if (*results*.length === 1 && *results*[0].name === *req*.body.name && *results*[0].alias === *req*.body.alias) return *res*.cc('分类名称与分类别名被占用，请更换后重试！')

    if (*results*.length === 1 && *results*[0].name === *req*.body.name) return *res*.cc('分类名称被占用，请更换后重试！')

    if (*results*.length === 1 && *results*[0].alias === *req*.body.alias) return *res*.cc('分类别名被占用，请更换后重试！')

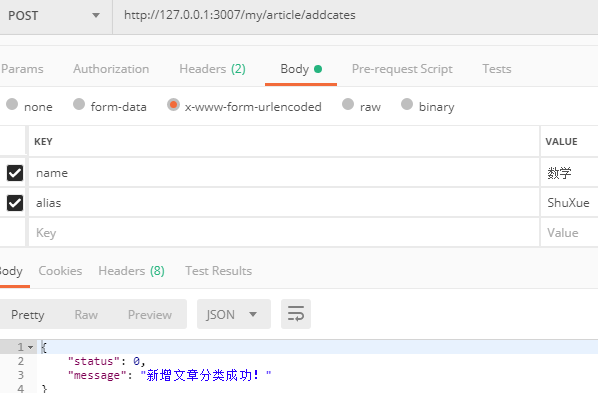
1. 实现新增文章分类的功能

 // 定义插入文章分类的 SQL 语句

    const sql = `insert into ev\_article\_cate set ?`

    // 执行插入文章分类的 SQL 语句

db.query(sql, *req*.body, (*err*, *results*) => {}



### 删除文章分类的接口

1. 定义路由和处理函数

在/router/artcate.js模块中

// 根据 Id 删除文章分类的路由

router.get('/deletecate/:id', expressJoi(delete\_cate\_schema), artCate\_handler.deleteCateById)

1. 验证表单函数

在/schema/artcate.js模块中

// id 的校验规则

const id = joi.number().integer().min(1).required()

1. 实现删除文章分类的功能

在/router\_handler/artcate.js模块中

// 删除文章分类的处理函数

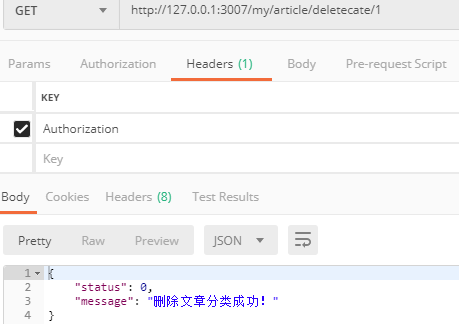
*exports*.deleteCateById = (*req*, *res*) => {

  // 定义标记删除的 SQL 语句

  const sql = `update ev\_article\_cate set is\_delete=1 where id=?`

  // 调用 db.query() 执行 SQL 语句

  db.query(sql, *req*.params.id, (*err*, *results*) => {}



### 根据id获取文章分类的接口

1.定义路由和处理函数

在/router/artcate.js模块中

// 根据 Id 获取文章分类的路由

router.get('/cates/:id', expressJoi(get\_cate\_schema), artCate\_handler.getArtCateById)

2.验证表单函数

在/schema/artcate.js模块中

// id 的校验规则

const id = joi.number().integer().min(1).required()

3.实现获取文章分类的功能

在/router\_handler/artcate.js模块中

// 根据 Id 获取文章分类的处理函数

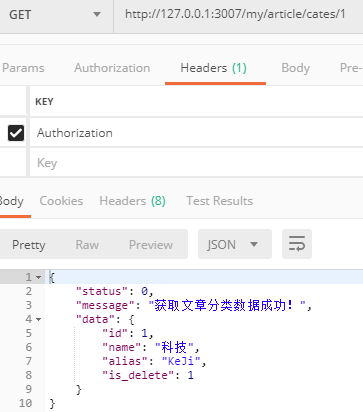
*exports*.getArtCateById = (*req*, *res*) => {

  // 定义根据 ID 获取文章分类数据的 SQL 语句

  const sql = `select \* from ev\_article\_cate where id=?`

  // 调用 db.query() 执行 SQL 语句

  db.query(sql, *req*.params.id, (*err*, *results*) => {}



登录注册新增修改用(提交)post

查询删除用(url上的参数)get

### 根据id更新文章分类的接口

1.定义路由和处理函数

在/router/artcate.js模块中

// 根据 Id 更新文章分类的路由

router.post('/updatecate', expressJoi(update\_cate\_schema), artCate\_handler.updateCateById)

2.验证表单函数

在/schema/artcate.js模块中

// 2. 定义 name 和 alias 的验证规则

const name = joi.string().required()

const alias = joi.string().alphanum().required()

// id 的校验规则

const id = joi.number().integer().min(1).required()

3.查询分类名称与分类别名是否被占用

不包括自身，自身名称和别名可以被自己占用

// 根据 Id 更新文章分类的处理函数

*exports*.updateCateById = (*req*, *res*) => {

  // 定义查重的 SQL 语句

  const sql = `select \* from ev\_article\_cate where Id<>? and (name=? or alias=?)`

  // 调用 db.query() 执行查重的 SQL 语句

  db.query(sql, [*req*.body.Id, *req*.body.name, *req*.body.alias], (*err*, *results*) => {

    // 执行 SQL 语句失败

    if (*err*) return *res*.cc(*err*)

    // 判断名称和别名被占用的4种情况

    if (*results*.length === 2) return *res*.cc('分类名称与别名被占用，请更换后重试！')

    if (*results*.length === 1 && *results*[0].name === *req*.body.name && *results*[0].alias === *req*.body.alias) return *res*.cc('分类名称与别名被占用，请更换后重试！')

    if (*results*.length === 1 && *results*[0].name === *req*.body.name) return *res*.cc('分类名称被占用，请更换后重试！')

    if (*results*.length === 1 && *results*[0].alias === *req*.body.alias) return *res*.cc('分类别名被占用，请更换后重试！')

4.实现更新文章分类的功能

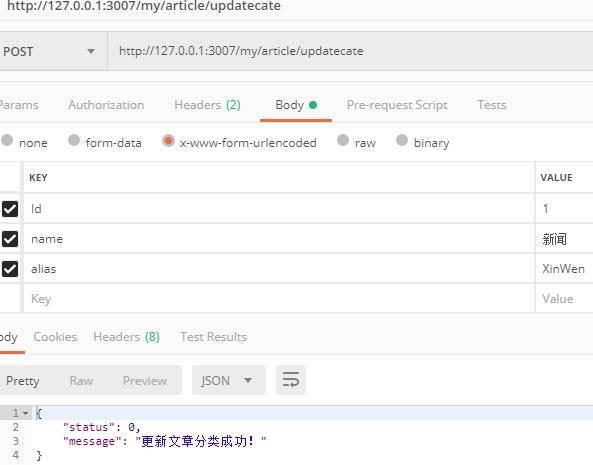
在/router\_handler/artcate.js模块中

// 定义更新文章分类的 SQL 语句

    const sql = `update ev\_article\_cate set ? where Id=?`

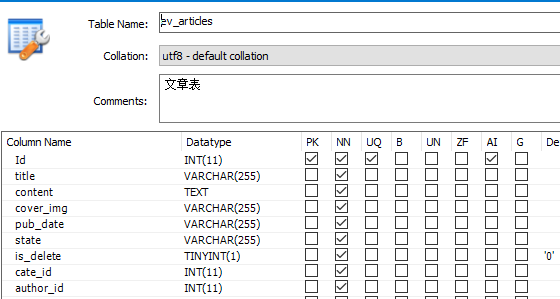
    // 执行更新文章分类的 SQL 语句

    db.query(sql, [*req*.body, *req*.body.Id], (*err*, *results*) => {}



## 文件管理

### 新建ev\_articles表



1. 初始化路由模块

// 发布新文章的路由 // upload.single() 是一个局部生效的中间件，用来解析 FormData 格式的表单数据 // 将文件类型的数据，解析并挂载到 req.file 属性中 // 将文本类型的数据，解析并挂载到 req.body 属性中

// 发布新文章的路由 // 注意：在当前的路由中，先后使用了两个中间件： // 先使用 multer 解析表单数据 // 再使用 expressJoi 对解析的表单数据进行验证

// 发布文章的路由

router.post('/add', uploads.single('cover\_img'), expressJoi(add\_article\_schema), article\_handler.addArticle)

1. 初始化路由处理函数模块

// 发布文章的处理函数

*exports*.addArticle = (*req*, *res*) => {

  console.log(*req*.file)

  if (!*req*.file || *req*.file.fieldname !== 'cover\_img') return *res*.cc('文章封面是必选参数！')

  // TODO：证明数据都是合法的，可以进行后续业务逻辑的处理

  // 处理文章的信息对象

  const articleInfo = {

    // 标题、内容、发布状态、所属分类的Id

    ...*req*.body,

    // 文章封面的存放路径

    cover\_img: path.join('/uploads', *req*.file.filename),

    // 文章的发布时间

    pub\_date: **new** *Date*(),

    // 文章作者的Id

    author\_id: *req*.user.id,

  }}

1. 使用multer解析表单数据

注意：使用 express.urlencoded() 中间件无法解析 multipart/form-data 格式的请求体数据。

当前项目，推荐使用 multer 来解析 multipart/form-data 格式的表单数据。[https://www.npmjs.com/package/multer](https://www.npmjs.com/package/multer" \t "http://www.escook.cn:8088/" \l "/mds/_blank)

// 创建 multer 的实例对象，通过 dest 属性指定文件的存放路径

// 导入 multer 和 path

const multer = require('multer')

const path = require('path')

// 创建 multer 的实例

const uploads = multer({ dest: path.join(\_\_dirname, '../uploads') })

1. 验证表数据

const joi = require('@hapi/joi')

// 分别定义 标题、分类Id、内容、发布状态的校验规则

const title = joi.string().required()

const cate\_id = joi.number().integer().min(1).required()

const content = joi.string().required().allow('')

const state = joi.string().valid('已发布', '草稿').required()

1. 实现发布文章的功能

const sql = `insert into ev\_articles set ?`

  db.query(sql, articleInfo, (*err*, *results*) => {

    if (*err*) return *res*.cc(*err*)

    if (*results*.affectedRows !== 1) return *res*.cc('发布新文章失败！')

*res*.cc('发布文章成功！', 0)

  })

在app.js中

// 托管静态资源文件

app.use('/uploads', express.static('./uploads'))

在素材大事件项目里面Ctrl+shiift+p输入express