

第四天

- 路由的概念
 - 路由就是**映射关系**
 - 根据不同的用户 URL 请求，返回不同的内容
 - 有点类似现实中的路标
 - 本质：URL 请求地址与服务器资源之间的对应关系
- 了解express中路由的概念及组成部分
 - 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**

```
1 app.METHOD(PATH, HANDLER)
```

```
1 // 匹配 GET 请求, 且请求 URL 为 /
2 app.get('/', function (req, res) {
3   res.send('Hello World!')
4 })
5
6 // 匹配 POST 请求, 且请求 URL 为 /
7 app.post('/', function (req, res) {
8   res.send('Got a POST request')
9 })
```

- 注意事项
 - **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
- 上面的写法存在的问题
- 优化路由写法
 - 单独提取到一个路由对象当中
- 上面的写法存在的问题
- 优化路由的写法
 - 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，将路由抽离为单独模块的步骤如下
 - 创建路由模块对应的 .js 文件
 - 调用 `express.Router()` 函数创建路由对象
 - 向路由对象上挂载具体的路由
 - 使用 `module.exports` 向外共享路由对象
 - 使用 `app.use()` 函数注册路由模块

- 代码落地

```
// 1. 导入 express
const express = require('express')
// 2. 创建路由对象
const router = express.Router()

// 3. 挂载获取用户列表的路由
router.get('/user/list', (req, res) => {
  res.send('用户列表')
})

// 4. 挂载添加用户列表的路由
router.post('/user/add', (req, res) => {
  res.send('添加用户')
})

// 5. 向外导出路由对象
module.exports = router
```

```
const express = require('express')
const app = express()

// 导入路由模块
const userRouter = require('./002-router')

// 使用 app.use() 注册路由模块
app.use(userRouter)

app.listen(3000, () => {
  console.log('running.....')
})
```

- 为路由模块添加前缀
 - 类似于托管静态资源写法，为静态资源统一挂载访问前缀一样

- 注意，添加了路由前缀后，访问的路由的时候，也应该加上前缀
- 代码落地

```
const express = require('express')
const app = express()

// 导入路由模块
const userRouter = require('./002-router')

// 使用 app.use() 注册路由模块
// 给路由模块添加统一得到访问前缀 /api
app.use('/api', userRouter)

app.listen(3000, () => {
  console.log('running.....')
})
```

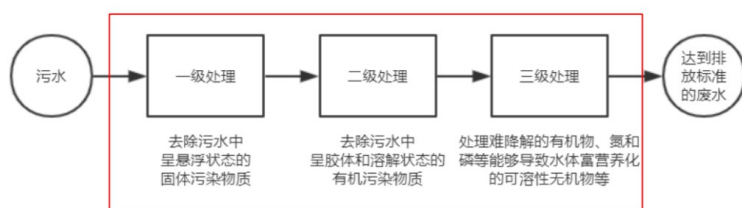
• 中间件概述

• 什么是中间件

- software 软件
- microsoft 微软
- 所谓的中间件（Middleware），特指业务流程的中间处理环节

• 现实中的中间件

- 在处理污水的时候，一般都要经过三个处理环节，从而保证处理过后的废水，达到排放标准，处理污水的这三个中间处理环节，就可以叫做中间件



• 中间件的作用

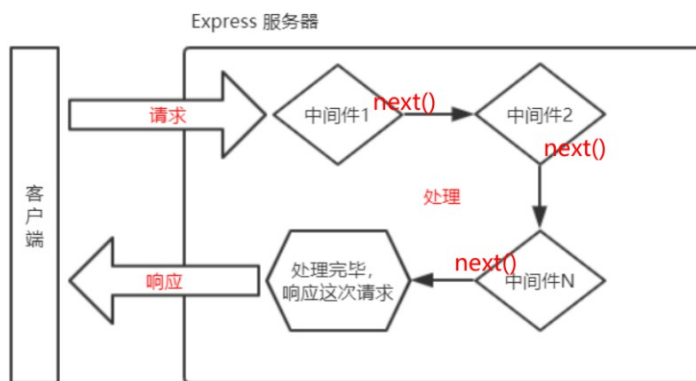
- 可以把多个路由要做的同样的逻辑提取出来放到统一的位置，逻辑只要写一次就行了

• 中间件的语法格式

- Express 的中间件，本质上就是一个 **function 处理函数**，只不过这个函数有三个固定的参数：分别是 req, res, next

• 注意事项

- **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



- 代码落地

```
const express = require('express')
const app = express()

// 定义一个最简单的中间件函数
const middleware = (req, res, next) => {
  console.log('这是最简单的中间件函数')
  // 把流转关系，转交给下一个中间件或者路由
  next()
}

// 全局生效的中间件
app.use(middleware)

app.get('/', (req, res) => {
  console.log('调用了 / 这个路由')

  res.send('Home page')
})
```

- 可以同时调用多个中间件

```

const express = require('express')
const app = express()

// 第一个中间件
app.use((req, res, next) => {
  console.log('调用了第一个全局的中间件')
  next()
})

// 第二个中间件
app.use((req, res, next) => {
  console.log('调用了第二个全局的中间件')
  next()
})

app.get('/user', (req, res) => {
  res.send('User Page')
})

app.listen(3000, () => {
  console.log('running.....')
})

```

- 注意
 - 顺序变了，执行结果不一样
 - 所有的中间件和路由共享同一份req和res
 - 必须在最后一行写next()把控制权交给下一个中间件
- 中间件的注意事项
 - 一定要在路由之前注册中间件
 - 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
 - 执行完中间件的业务代码之后，不要忘记调用 next() 函数
 - 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
 - 连续调用多个中间件时，多个中间件之间，共享 req 和 res 对象
- 按作用到路由的数量划分为两种类型中间件
 - 全局中间件
 - app.use(中间件)
 - 局部中间件
 - 局部中间件的语法两种写法
 - app.get(url,mw,callback)
 - app.get(url,[mw1,mw2],callback)
- 按功能划分为五种类型中间件
 - 应用级别中间件

```

JS app.js > ...
1  const express = require('express')
2  const app = express()
3  const middleware = function (req, res, next) {
4    console.log('当前访问的时间为' + Date.now());
5    next()
6  }
7  app.use(middleware)
8  const userRouter = require('./userRouter')
9  const articleRouter = require('./articleRouter')
10 app.use('/user', userRouter)
11 app.use('/article', articleRouter)
12 app.listen(3000, () => {
13   console.log('服务器开启成功');
14 })

```

• 路由级别中间件

```

文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)  userRouter.js - code1112
JS app.js  JS userRouter.js x  JS articleRouter.js
JS userRouter.js > ...
1  const express = require('express')
2  const router = express.Router()
3  const middleware = function (req, res, next) {
4    console.log('当前访问的时间为' + Date.now());
5    next()
6  }
7  router.use(middleware)
8  router.get('/index', (req, res) => {
9    res.send('用户被访问')
10 })
11 module.exports = router

```

1. 一个应用(app)下面可以有多个路由模块

// 2. 应用级别中间件(app.use(中间件))对所有的路由规则全部有效果

// 3. 路由级别中间件(router.use(中间件))只会对当前这个路由模块有效果

• 错误级别中间件

- 错误级别中间件的作用：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)
- 注意：错误级别的中间件，必须注册在所有路由之后


```

    router.use(middleware)
    router.get('/index', (req, res) => {
      try {
        const str = '{name:"zs"}'
        JSON.parse(str)
      } catch (err) {
        console.log(err.message);
        res.send(err.message)
        return
      }
      res.send('用户被访问')
    })
  })
  module.exports = router

```

```

5   next()
6 }
7 router.use(middleware)
8 router.get('/index', (req, res) => {
9   JSON.parse('{name:"zs"}')
10  res.send('用户被访问')
11 })
12 module.exports = router

```

```

3 const userRouter = require('./userRouter')
4 const articleRouter = require('./articleRouter')
5 app.use('/user', userRouter)
6 app.use('/article', articleRouter)
7 app.use((err, req, res, next) => {
8   res.send(err.message)
9 })
10 app.listen(3000, () => {
11   console.log('服务器开启成功');
12 })

```

- express内置的中间件
 - express.static 快速托管静态资源的内置中间件，例如：HTML 文件、图片、CSS 样式等
 - express.urlencoded 解析 URL-encoded 格式的请求体数据
 - express.urlencoded 解析 URL-encoded 格式的请求体数据

```
const express = require('express')
const app = express()

// 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('/book', (req, res) => {
  console.log(req.body)
  res.send(req.body)
})

app.listen(3000, () => {
  console.log('running.....')
})
```

- 第三方中间件

- 非 Express 官方内置，而是由第三方开发出来的中间件，叫做第三方中间件。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

- 第三方中间件举例

- cors 跨域
- express-jwt 身份认证
- jsonwebtoken

- 扩展：为什么我们写的中间件是一个函数的定义，那些内置或第三方的中间件怎么看起来是函数的调用？

- 基于 Express 写接口

- 接口的数据格式规范

- 代码落地

```
// index.js
// 导入 express 模块
const express = require('express')

// 创建 express 的服务器实例
const app = express()

// 配置解析表单数据的中间件
//注意：如果要获取 URL-encoded 格式的请求体数据，必须配置中间件 app.use(express.urlencoded({ extended: false })))
app.use(express.urlencoded({ extended: false })))

// 导入路由模块
const router = require('./020-apiRouter')
// 把路由模块，注册到 app 上
app.use('/api', router)

// 调用 app.listen 方法，指定端口号并启动 web 服务器
app.listen(3000, () => {
  console.log('running.....')
})
```



```
const express = require('express')
const router = express.Router()

// 在这里挂载对应的路由
router.get('/book', (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, 想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', // 状态描述
    data: query // 需要响应给客户端的数据
  })
})

module.exports = router
```

```
const express = require('express')
const router = express.Router()

// 在这里挂载对应的路由

// 定义 POST 接口
router.post('/book', (req, res) => {
  // 通过 req.body 获取请求中包含的 url-encoded 格式的数据
  const body = req.body

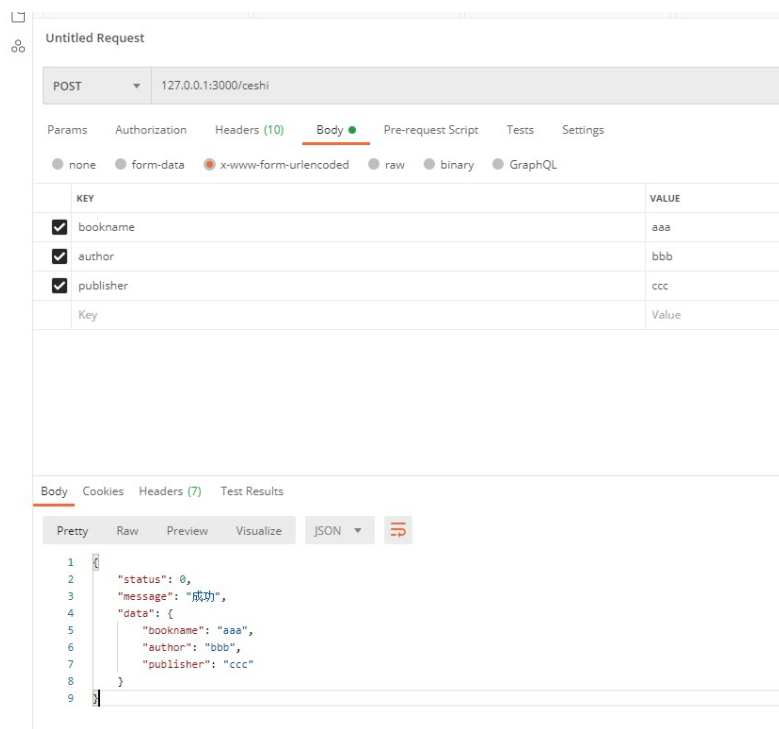
  // 调用 res.send() 方法, 向客户端响应结果
  res.send({
    status: 0,
    msg: 'Post 请求成功',
    data: body
  })
})

module.exports = router
```

- 接口的跨域问题
 - 到目前为止, 我们编写的 GET 和 POST 接口, 存在一个很严重的问题: **不支持跨域请求**
 - 解决接口跨域问题的方案主要有三种
 - **JSONP** (有缺陷的解决方案: 只支持 GET 请求)
 - 在企业中没有人使用了, 但是面试还是需要掌握的一个知识点
 - **CORS** (主流的解决方案, 推荐使用)
 - 代理 proxy (后面 vue 人力资源项目中会讲)
 - 跨域解决方案一: JSONP
 - 理解

- 原理：利用script标签可以跨域
- 实现思路
 - 前端 提前准备一个全局函数 前端还要通过script标签向后端发送一个请求，通过callback把全局函数名字传递给后端
 - 后端 通过req.query.callback获取到函数的名字，然后把json数据包装成jsonp形式（函数调用，数据作为实参）
- 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。这种请求数据的方式叫做 JSONP
- 画图理解
- 特点
 - JSONP 不属于真正的 Ajax 请求，因为它没有使用 XMLHttpRequest 这个对象
 - JSONP 仅支持 GET 请求，不支持 POST、DELETE、PUT、PATCH等请求
- 跨域解决方案二：CORS
 - 缺点：兼容性只有高级浏览器才支持
 - 所谓的CORS很简单，就是后台接口在返回json数据的时候，要顺便在响应头中加一行响应头信息
 - `Access-Control-Allow-Origin:*`
- 第一步、准备测试代码并用postman测试

```
const express = require('express')
const app = express()
app.use(express.urlencoded({ extended: false }))
app.post('/ceshi', (req, res) => {
  res.send({
    status: 0,
    message: '成功',
    data: req.body
  })
})
app.listen(3000, () => {
  console.log('服务器开启成功');
})
```



- 第二步、写前端代码发ajax拿数据报错



- 第三步 为什么大事件不报错?设置和大事件同样的东西

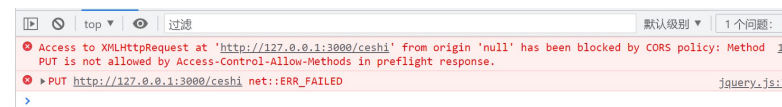
- <http://www.escook.cn:8086/>



- 第四步、优化

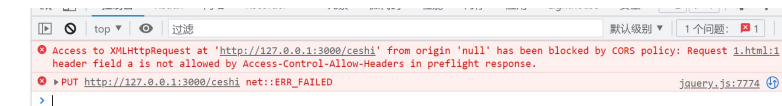
```
JS app.js x
JS app.js > ...
1  const express = require('express')
2  const app = express()
3  app.use(express.urlencoded({ extended: false }))
4  app.use((req, res, next) => {
5    res.setHeader('Access-Control-Allow-Origin', '*')
6    next()
7  })
8  app.post('/ceshi', (req, res) => {
9    res.send({
10     status: 0,
11     message: '成功',
12     data: req.body,
13   })
14 })
15 app.listen(3000, () => {
16   console.log('服务器开启成功')
17 })
18
```

• 第五步、把POST修改为PUT及解决



```
2  const app = express()
3  app.use(express.urlencoded({ extended: false }))
4  app.use((req, res, next) => {
5    res.setHeader('Access-Control-Allow-Origin', '*')
6    res.setHeader('Access-Control-Allow-Methods', '*')
7    next()
8  })
9  app.put('/ceshi', (req, res) => {
```

• 第六步、添加自定义头信息



```
1  const express = require('express')
2  const app = express()
3  app.use(express.urlencoded({ extended: false }))
4  app.use((req, res, next) => {
5    res.setHeader('Access-Control-Allow-Origin', '*')
6    res.setHeader('Access-Control-Allow-Methods', '*')
7    res.setHeader('Access-Control-Allow-Headers', '*')
8    next()
9  })
10 app.put('/ceshi', (req, res) => {
11   res.send({
```

• 第七步、直接使用第三方中间件 cors

• 简单请求和预检请求的区别

- <https://developer.mozilla.org/zh-CN/docs/web/http/cors#%E8%8B%A5%E5%B9%B2%E8%AE%BF%E9%97%AE%E6%8E%A7%E5%88%B6%E5%9C%BA%E6%99%AF>

• ajax请求有两种

- 简单请求
 - 只有GET、POST并且没有自定义请求头信息

- 预检请求
 - 不是GET或POST，或者有自定义头信息
 - 特点
 - 请求会发送二次，第一次是OPTION预检请求