

# 案例-登录认证

在前面的课程中，我们已经实现了部门管理、员工管理的基本功能，但是大家会发现，我们并没有登录，就直接访问到了Tlias智能学习辅助系统的后台。 这是不安全的，所以我们今天的主题就是登录认证。 最终我们要实现的效果就是用户必须登录之后，才可以访问后台系统中的功能。



## 1. 登录功能

### 1.1 需求



在登录界面中，我们可以输入用户的用户名以及密码，然后点击“登录”按钮就要请求服务器，服务端判断用户输入的用户名或者密码是否正确。如果正确，则返回成功结果，前端跳转至系统首页面。

## 1.2 接口文档

我们参照接口文档来开发登录功能

- 基本信息

```
1  请求路径： /login
2
3  请求方式： POST
4
5  接口描述：该接口用于员工登录Tlias智能学习辅助系统，登录完毕后，系统下发JWT令牌。
```

- 请求参数

参数格式： application/json

参数说明：

名称	类型	是否必须	备注
username	string	必须	用户名
password	string	必须	密码

请求数据样例：

```
1  {
2      "username": "jinyong",
3      "password": "123456"
4  }
```

- 响应数据

参数格式： application/json

参数说明：

名称	类型	是否必须	默认值	备注	其他信息
code	number	必须		响应码， 1 成功 ； 0 失败	
msg	string	非必须		提示信息	
data	string	必须		返回的数据 ， jwt令牌	

响应数据样例：

```

1  {
2      "code": 1,
3      "msg": "success",
4      "data":
5          "eyJhbGciOiJIUzI1NiJ9.eyJ1Ijoi6YeR5bq4IiwiaWQiOiJEsInVzZXJuYXW1IjoiamlueW9uZyIsImV4cCI6MTY2MjIwNzA0OH0.KkUc_CXJZJ8Dd063eImx4H9Ojfrr6XMJ-yVzaWCVZCo"
6  }

```

### 1.3 思路分析



登录服务端的核心逻辑就是：接收前端请求传递的用户名和密码，然后再根据用户名和密码查询用户信息，如果用户信息存在，则说明用户输入的用户名和密码正确。如果查询到的用户不存在，则说明用户输入的用户名和密码错误。

### 1.4 功能开发

#### LoginController

```

1  @RestController
2  public class LoginController {
3
4      @Autowired
5      private EmpService empService;
6
7      @PostMapping("/login")
8      public Result login(@RequestBody Emp emp) {
9          Emp e = empService.login(emp);
10         return e != null ? Result.success() : Result.error("用户名或密码错误");
11     }
12 }

```

#### EmpService

```

1  public interface EmpService {
2
3      /**
4       * 用户登录
5       * @param emp
6       * @return
7       */
8      public Emp login(Emp emp);
9
10     //省略其他代码...
11 }

```

### EmpServiceImpl

```

1  @Slf4j
2  @Service
3  public class EmpServiceImpl implements EmpService {
4      @Autowired
5      private EmpMapper empMapper;
6
7      @Override
8      public Emp login(Emp emp) {
9          //调用dao层功能：登录
10         Emp loginEmp = empMapper.getByUsernameAndPassword(emp);
11
12         //返回查询结果给Controller
13         return loginEmp;
14     }
15
16     //省略其他代码...
17 }

```

### EmpMapper

```

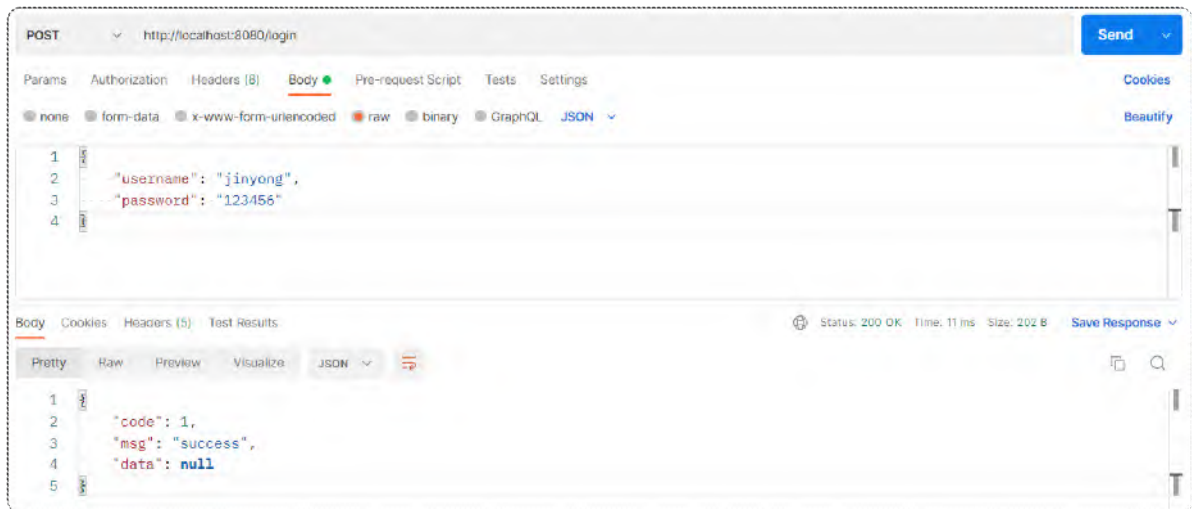
1  @Mapper
2  public interface EmpMapper {
3
4      @Select("select id, username, password, name, gender, image,
5              job, entrydate, dept_id, create_time, update_time " +
6              "from emp " +
7              "where username=#{username} and password=#{password}")
8      public Emp getByUsernameAndPassword(Emp emp);
9
10     //省略其他代码...
11 }

```

## 1.5 测试

功能开发完毕后，我们就可以启动服务，打开postman进行测试了。

发起POST请求，访问：<http://localhost:8080/login>



postman测试通过了，那接下来，我们就可以结合着前端工程进行联调测试。

先退出系统，进入到登录页面：



在登录页面输入账户密码：



登录成功之后进入到后台管理系统页面：

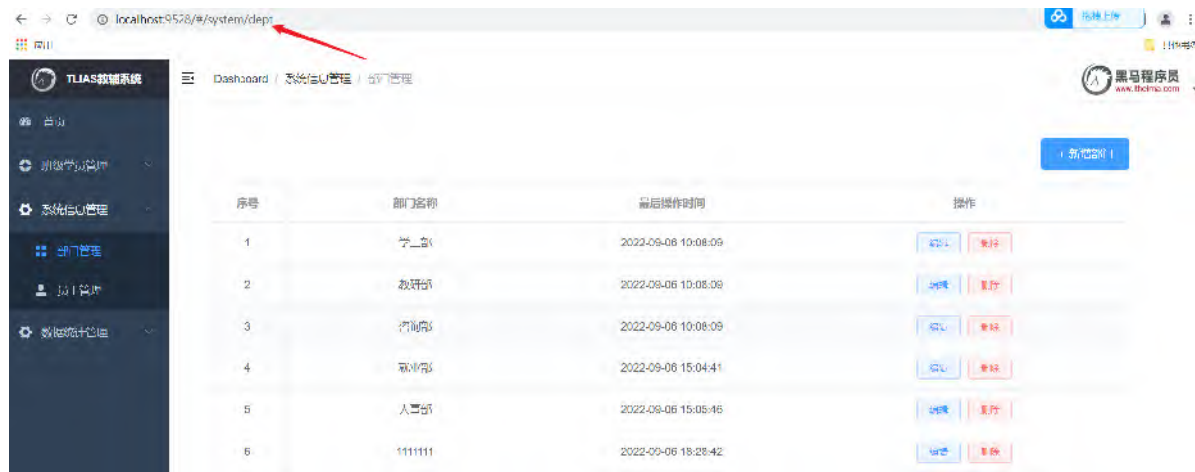


## 2. 登录校验

### 2.1 问题分析

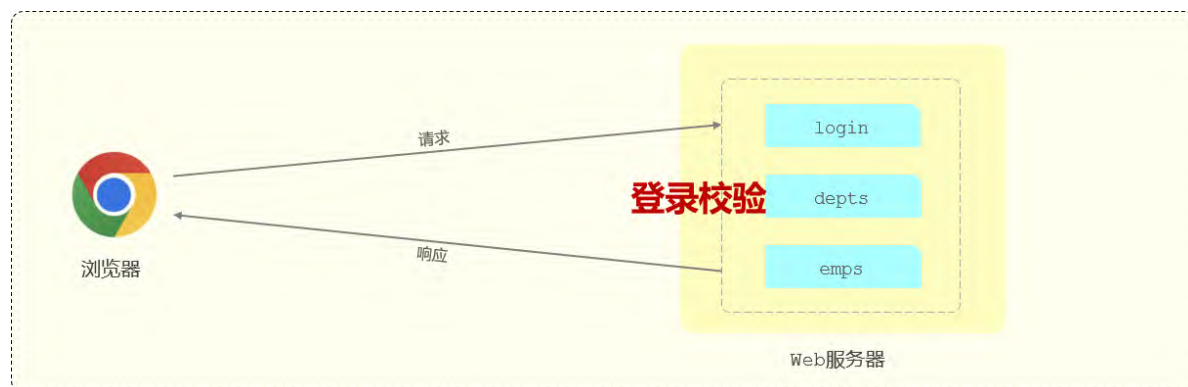
我们已经完成了基础登录功能的开发与测试，在我们登录成功后就可以进入到后台管理系统中进行数据的操作。

但是当我们在浏览器中新的页面上输入地址：<http://localhost:9528/#/system/dept>，发现没有登录仍然可以进入到后端管理系统页面。



而真正的登录功能应该是：登陆后才能访问后端系统页面，不登陆则跳转登陆页面进行登陆。

为什么会出现这个问题？其实原因很简单，就是因为针对于我们当前所开发的部门管理、员工管理以及文件上传等相关接口来说，我们在服务器端并没有做任何的判断，没有去判断用户是否登录了。所以无论用户是否登录，都可以访问部门管理以及员工管理的相关数据。所以我们目前所开发的登录功能，它只是徒有其表。而我们要想解决这个问题，我们就需要完成一步非常重要的操作：登录校验。



什么是登录校验？

- 所谓登录校验，指的是我们在服务器端接收到浏览器发送过来的请求之后，首先我们要对请求进行校验。先要校验一下用户登录了没有，如果用户已经登录了，就直接执行对应的业务操作就可以了；如果用户没有登录，此时就不允许他执行相关的业务操作，直接给前端响应一个错误的结果，最终跳转到登录页面，要求他登录成功之后，再来访问对应的数据。

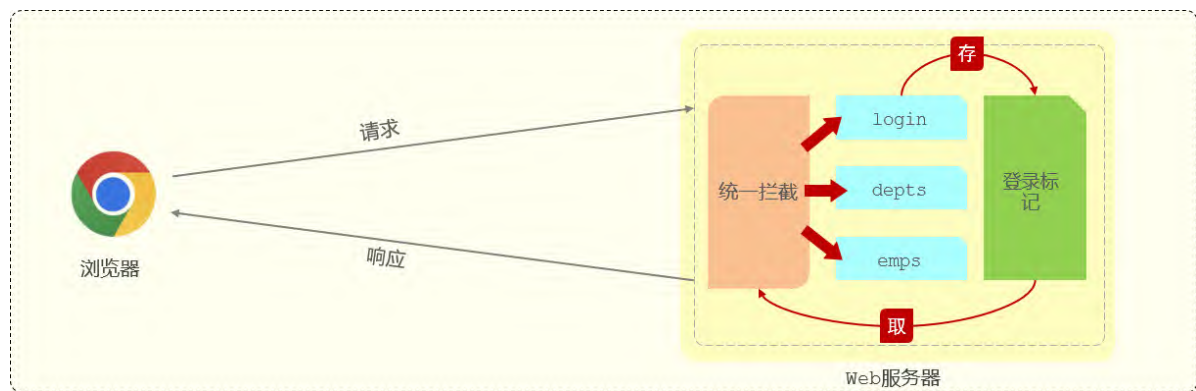
了解完什么是登录校验之后，接下来我们分析一下登录校验大概的实现思路。

首先我们在宏观上先有一个认知：

前面在讲解HTTP协议的时候，我们提到HTTP协议是无状态协议。什么又是无状态的协议？

所谓无状态，指的是每一次请求都是独立的，下一次请求并不会携带上一次请求的数据。而浏览器与服务器之间进行交互，基于HTTP协议也就意味着现在我们通过浏览器来访问了登陆这个接口，实现了登陆的操作，接下来我们在执行其他业务操作时，服务器也并不知道这个员工到底登陆了没有。因为HTTP协议是无状态的，两次请求之间是独立的，所以是无法判断这个员工到底登陆了没有。





那应该怎么来实现登录校验的操作呢？具体的实现思路可以分为两部分：

1. 在员工登录成功后，需要将用户登录成功的信息存起来，记录用户已经登录成功的标记。
2. 在浏览器发起请求时，需要在服务端进行统一拦截，拦截后进行登录校验。

想要判断员工是否已经登录，我们需要在员工登录成功之后，存储一个登录成功的标记，接下来在每一个接口方法执行之前，先做一个条件判断，判断一下这个员工到底登录了没有。如果是登录了，就可以执行正常的业务操作，如果没有登录，会直接给前端返回一个错误的信息，前端拿到这个错误信息之后会自动的跳转到登录页面。

我们程序中所开发的查询功能、删除功能、添加功能、修改功能，都需要使用以上套路进行登录校验。此时就会出现：相同代码逻辑，每个功能都需要编写，就会造成代码非常繁琐。

为了简化这块操作，我们可以使用一种技术：统一拦截技术。

通过统一拦截的技术，我们可以来拦截浏览器发送过来的所有的请求，拦截到这个请求之后，就可以通过请求来获取之前所存入的登录标记，在获取到登录标记且标记为登录成功，就说明员工已经登录了。如果已经登录，我们就直接放行（意思就是可以访问正常的业务接口了）。

我们要完成以上操作，会涉及到web开发中的两个技术：

1. 会话技术
2. 统一拦截技术

而统一拦截技术现实方案也有两种：

1. Servlet规范中的Filter过滤器
2. Spring提供的interceptor拦截器

下面我们先学习会话技术，然后再学习统一拦截技术。



## 2.2 会话技术

介绍了登录校验的大概思路之后，我们先来学习下会话技术。

### 2.2.1 会话技术介绍

什么是会话？

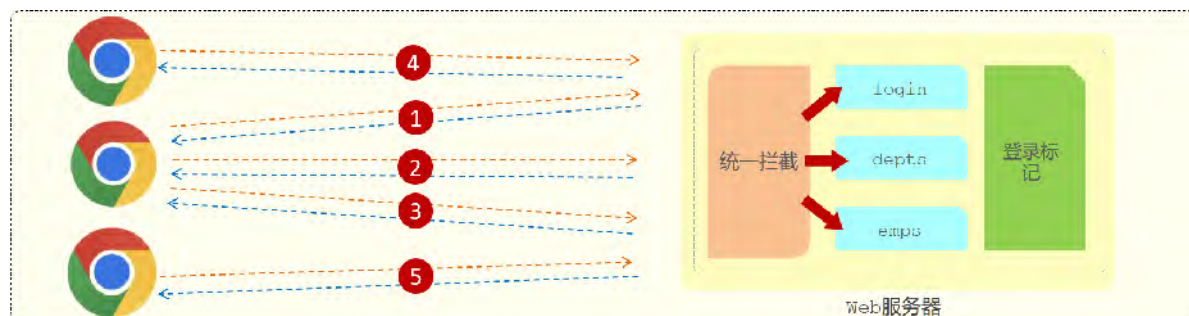
- 在我们日常生活当中，会话指的就是谈话、交谈。
- 在web开发当中，会话指的就是浏览器与服务器之间的一次连接，我们就称为一次会话。

在用户打开浏览器第一次访问服务器的时候，这个会话就建立了，直到有任何一方断开连接，此时会话就结束了。在一次会话当中，是可以包含多次请求和响应的。

比如：打开了浏览器来访问web服务器上的资源（浏览器不能关闭、服务器不能断开）

- 第1次：访问的是登录的接口，完成登录操作
- 第2次：访问的是部门管理接口，查询所有部门数据
- 第3次：访问的是员工管理接口，查询员工数据

只要浏览器和服务器都没有关闭，以上3次请求都属于一次会话当中完成的。



需要注意的是：会话是和浏览器关联的，当有三个浏览器客户端和服务端建立了连接时，就会有三个会话。同一个浏览器在未关闭之前请求了多次服务器，这多次请求是属于同一个会话。比如：1、2、3这三个请求都是属于同一个会话。当我们关闭浏览器之后，这次会话就结束了。而如果我们直接把web服务器关了，那么所有的会话就都结束了。

知道了会话的概念了，接下来我们再了解下会话跟踪。

会话跟踪：一种维护浏览器状态的方法，服务器需要识别多次请求是否来自于同一浏览器，以便在同一次会话的多次请求间共享数据。

服务器会接收很多的请求，但是服务器是需要识别出这些请求是不是同一个浏览器发出来的。比如：1和2这两个请求是不是同一个浏览器发出来的，3和5这两个请求是不是同一个浏览器发出来的。如果是同一个浏览器发出来的，就说明是同一个会话。如果是不同的浏览器发出来的，就说明是不同的会话。而识别多次请求是否来自于同一浏览器的过程，我们就称为会话跟踪。

我们使用会话跟踪技术就是要完成在同一个会话中，多个请求之间进行共享数据。

为什么要共享数据呢？

由于HTTP是无状态协议，在后面请求中怎么拿到前一次请求生成的数据呢？此时就需要在一次会话的多次请求之间进行数据共享

会话跟踪技术有两种：

1. Cookie（客户端会话跟踪技术）
  - 数据存储在客户端浏览器当中
2. Session（服务端会话跟踪技术）
  - 数据存储在服务端
3. 令牌技术

## 2.2.2 会话跟踪方案

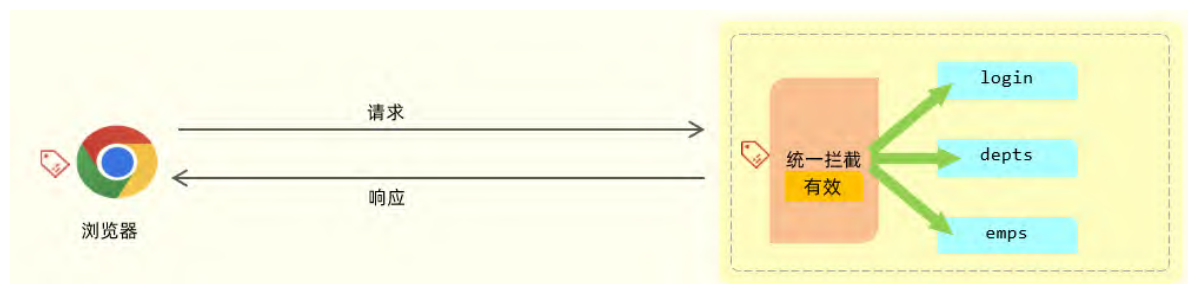
上面我们介绍了什么是会话，什么是会话跟踪，并且也提到了会话跟踪 3 种常见的技术方案。接下来，我们就来对比一下这 3 种会话跟踪的技术方案，来看一下具体的实现思路，以及它们之间的优缺点。

### 2.2.2.1 方案一 - Cookie

cookie 是客户端会话跟踪技术，它是存储在客户端浏览器的，我们使用 cookie 来跟踪会话，我们就可以在浏览器第一次发起请求来请求服务器的时候，我们在服务器端来设置一个cookie。

比如第一次请求了登录接口，登录接口执行完成之后，我们就可以设置一个cookie，在 cookie 当中我们就可以来存储用户相关的一些数据信息。比如我可以在 cookie 当中来存储当前登录用户的用户名，用户的ID。

服务器端在给客户端在响应数据的时候，会**自动的**将 cookie 响应给浏览器，浏览器接收到响应回来的 cookie 之后，会**自动的**将 cookie 的值存储在浏览器本地。接下来在后续每一次请求当中，都会将浏览器本地所存储的 cookie **自动地**携带到服务端。



接下来在服务端我们就可以获取到 cookie 的值。我们可以去判断一下这个 cookie 的值是否存在，如果不存在这个 cookie，就说明客户端之前是没有访问登录接口的；如果存在 cookie 的值，就说明客户端之前已经登录完成了。这样我们就可以基于 cookie 在同一次会话的不同请求之间来共享数据。

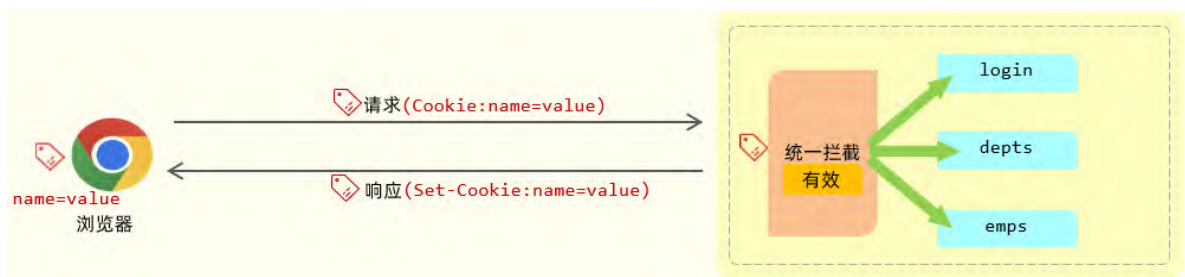
我刚才在介绍流程的时候，用了 3 个自动：

- 服务器会 **自动** 的将 cookie 响应给浏览器。
- 浏览器接收到响应回来的数据之后，会 **自动** 的将 cookie 存储在浏览器本地。
- 在后续的请求当中，浏览器会 **自动** 的将 cookie 携带到服务器端。

### 为什么这一切都是自动化进行的？

是因为 cookie 它是 HTTP 协议当中所支持的技术，而各大浏览器厂商都支持了这一标准。在 HTTP 协议官方给我们提供了一个响应头和请求头：

- 响应头 Set-Cookie：设置Cookie数据的
- 请求头 Cookie：携带Cookie数据的



### 代码测试

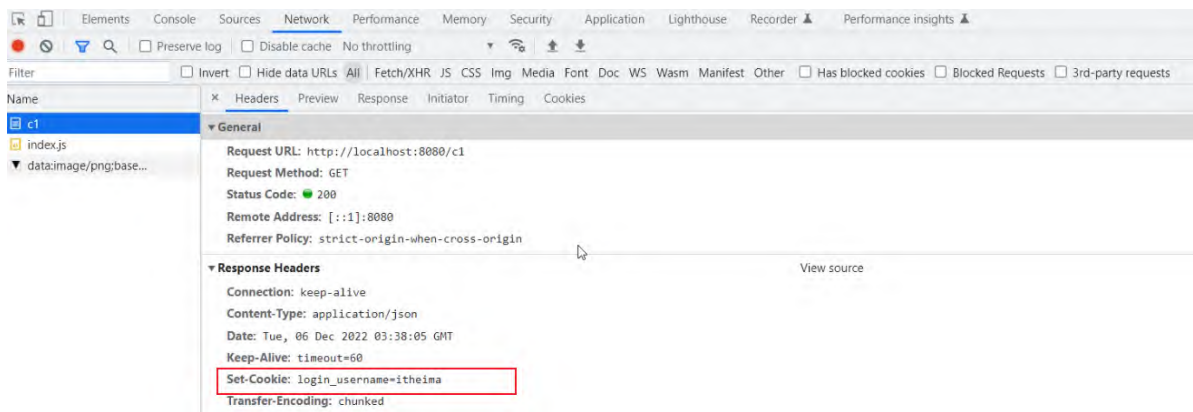
```
1  @Slf4j
2  @RestController
3  public class SessionController {
4
5      //设置Cookie
6      @GetMapping("/c1")
7      public Result cookie1(HttpServletResponse response){
8          response.addCookie(new Cookie("login_username","itheima"));
9          //设置Cookie/响应Cookie
10         return Result.success();
11     }
12
13     //获取Cookie
```

```

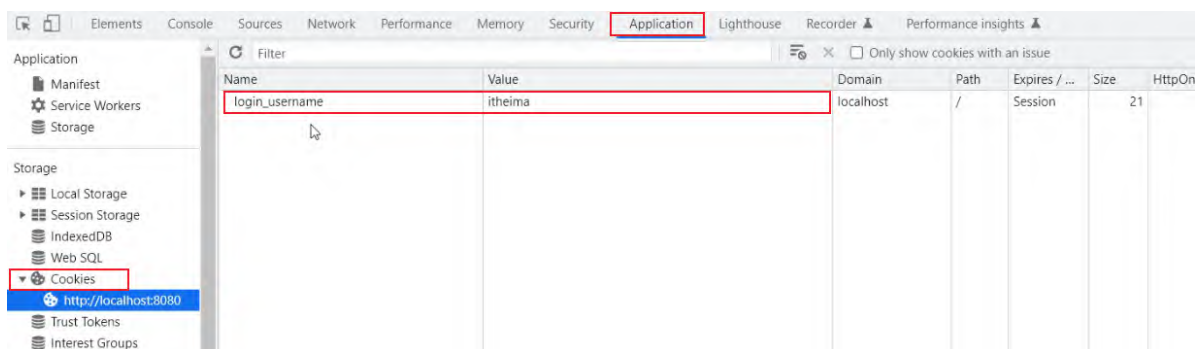
13     @GetMapping("/c2")
14     public Result cookie2(HttpServletRequest request){
15         Cookie[] cookies = request.getCookies();
16         for (Cookie cookie : cookies) {
17             if(cookie.getName().equals("login_username")){
18                 System.out.println("login_username:
19                 "+cookie.getValue()); //输出name为login_username的cookie
20             }
21         }
22         return Result.success();
23     }

```

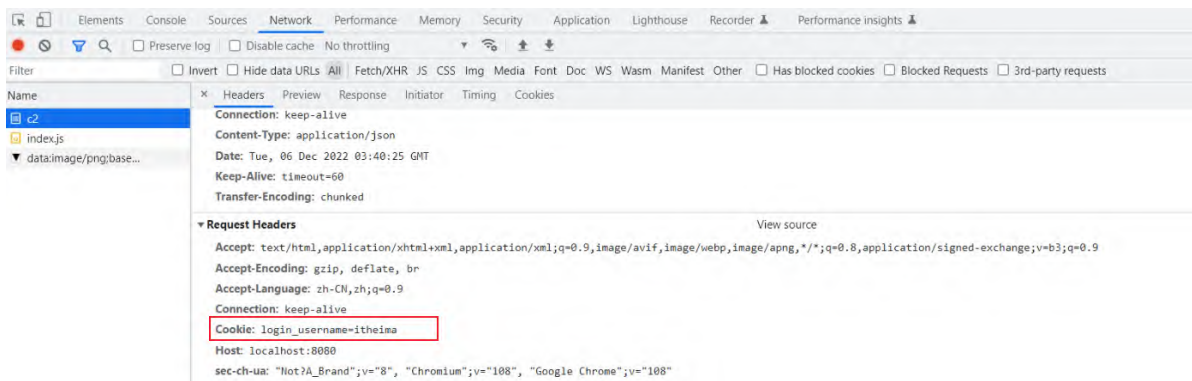
A. 访问c1接口, 设置Cookie, <http://localhost:8080/c1>



我们可以看到, 设置的cookie, 通过响应头Set-Cookie响应给浏览器, 并且浏览器会将Cookie, 存储在浏览器端。



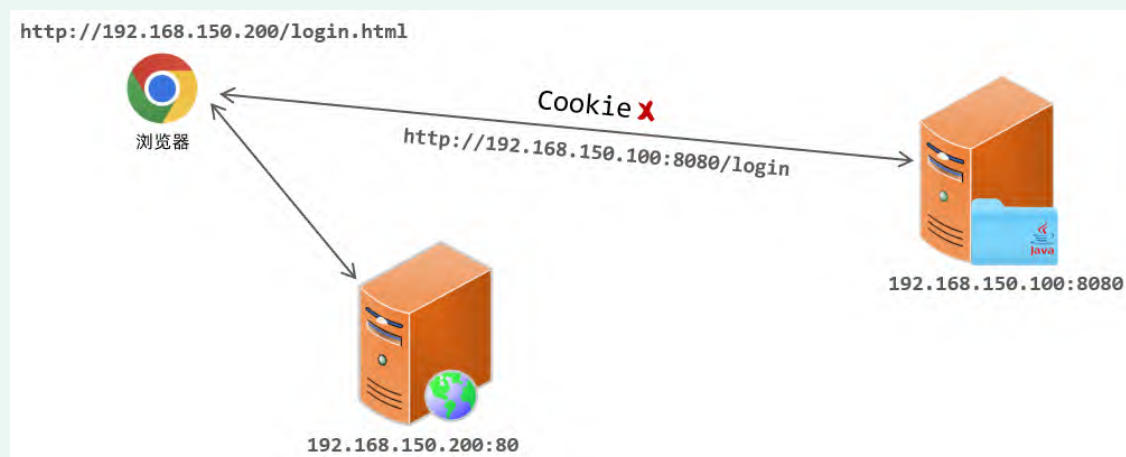
B. 访问c2接口 <http://localhost:8080/c2>, 此时浏览器会自动的将Cookie携带到服务端, 是通过请求头Cookie, 携带的。



## 优缺点

- 优点：HTTP协议中支持的技术（像Set-Cookie 响应头的解析以及 Cookie 请求头数据的携带，都是浏览器自动进行的，是无需我们手动操作的）
- 缺点：
  - 移动端APP (Android、IOS) 中无法使用Cookie
  - 不安全，用户可以自己禁用Cookie
  - Cookie不能跨域

跨域介绍：



- 现在的项目，大部分都是前后端分离的，前后端最终也会分开部署，前端部署在服务器 192.168.150.200 上，端口 80，后端部署在 192.168.150.100上，端口 8080
- 我们打开浏览器直接访问前端工程，访问url: <http://192.168.150.200/login.html>
- 然后在该页面发起请求到服务端，而服务端所在地址不再是localhost，而是服务器的IP地址192.168.150.100，假设访问接口地址为: <http://192.168.150.100:8080/login>

- 那此时就存在跨域操作了，因为我们是在 `http://192.168.150.200/login.html` 这个页面上访问了 `http://192.168.150.100:8080/login` 接口
- 此时如果服务器设置了一个Cookie，这个Cookie是不能使用的，因为Cookie无法跨域

区分跨域的维度：

- 协议
- IP/协议
- 端口

只要上述的三个维度有任何一个维度不同，那就是跨域操作

举例：

`http://192.168.150.200/login.html` -----> `https://192.168.150.200/login`  
[协议不同，跨域]

`http://192.168.150.200/login.html` -----> `http://192.168.150.100/login`  
[IP不同，跨域]

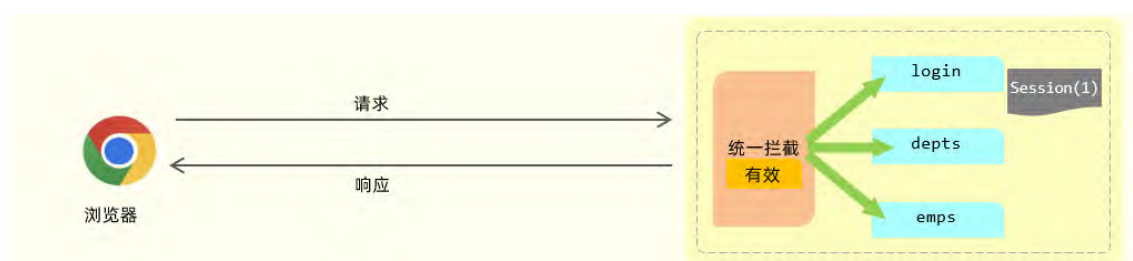
`http://192.168.150.200/login.html` -----> `http://192.168.150.200:8080/login` [端口不同，跨域]

`http://192.168.150.200/login.html` -----> `http://192.168.150.200/login`  
[不跨域]

#### 2.2.2.2 方案二 - Session

前面介绍的时候，我们提到Session，它是服务器端会话跟踪技术，所以它是存储在服务器端的。而Session 的底层其实就是基于我们刚才所介绍的 Cookie 来实现的。

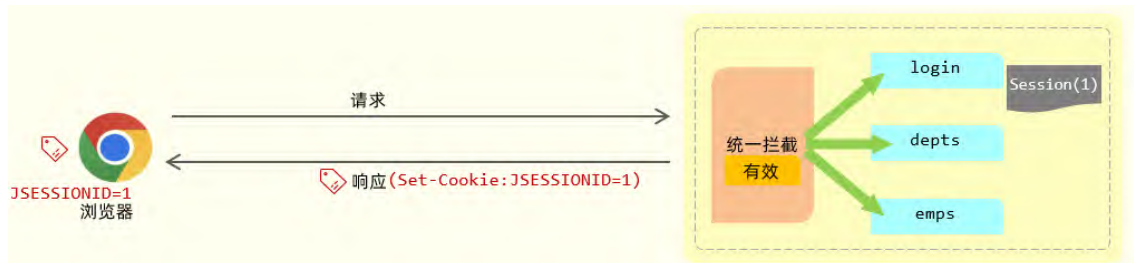
- 获取Session





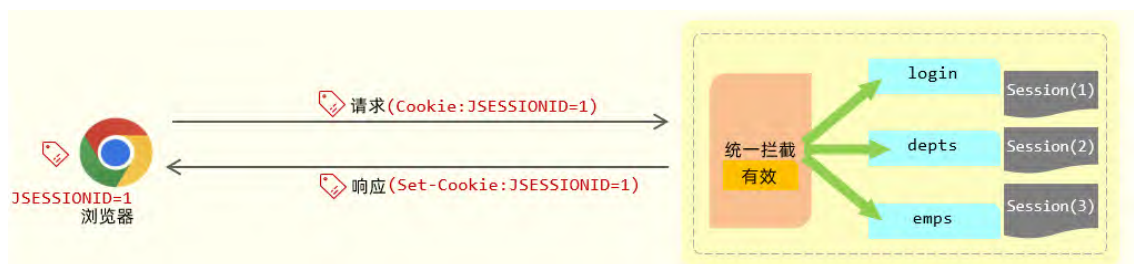
如果我们现在要基于 Session 来进行会话跟踪，浏览器在第一次请求服务器的时候，我们就可以直接在服务器当中来获取到会话对象Session。如果是第一次请求Session，会话对象是不存在的，这个时候服务器会自动的创建一个会话对象Session。而每一个会话对象Session，它都有一个ID（示意图中Session后面括号中的1，就表示ID），我们称之为 Session 的ID。

- 响应Cookie (JSESSIONID)



接下来，服务器端在给浏览器响应数据的时候，它会将 Session 的 ID 通过 Cookie 响应给浏览器。其实在响应头当中增加了一个 Set-Cookie 响应头。这个 Set-Cookie 响应头对应的值是不是cookie？cookie 的名字是固定的 JSESSIONID 代表的服务器端会话对象 Session 的 ID。浏览器会自动识别这个响应头，然后自动将Cookie存储在浏览器本地。

- 查找Session



接下来，在后续的每一次请求当中，都会将 Cookie 的数据获取出来，并且携带到服务端。接下来服务器拿到JSESSIONID这个 Cookie 的值，也就是 Session 的ID。拿到 ID 之后，就会从众多的 Session 当中来找到当前请求对应的会话对象Session。

这样我们是不是就可以通过 Session 会话对象在同一次会话的多次请求之间来共享数据了？好，这就是基于 Session 进行会话跟踪的流程。

## 代码测试

```
1  @Slf4j
2  @RestController
3  public class SessionController {
4
5      @GetMapping("/s1")
6      public Result session1(HttpSession session){
```

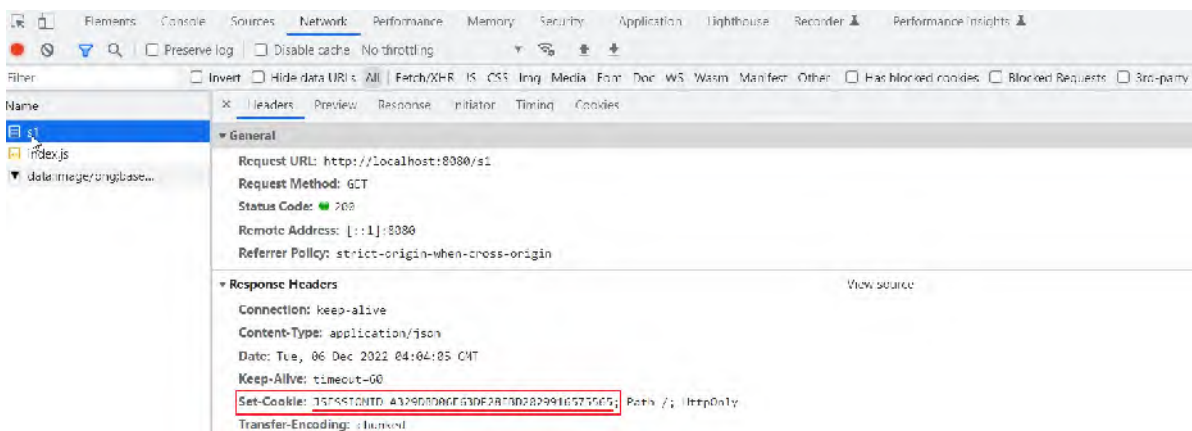


```

7         log.info("HttpSession-s1: {}", session.hashCode());
8
9         session.setAttribute("loginUser", "tom"); //往session中存储数据
10
11         return Result.success();
12     }
13
14     @GetMapping("/s2")
15     public Result session2(HttpServletRequest request){
16         HttpSession session = request.getSession();
17         log.info("HttpSession-s2: {}", session.hashCode());
18
19         Object loginUser = session.getAttribute("loginUser"); //从session中获取数据
20
21         log.info("loginUser: {}", loginUser);
22         return Result.success(loginUser);
23     }
24 }

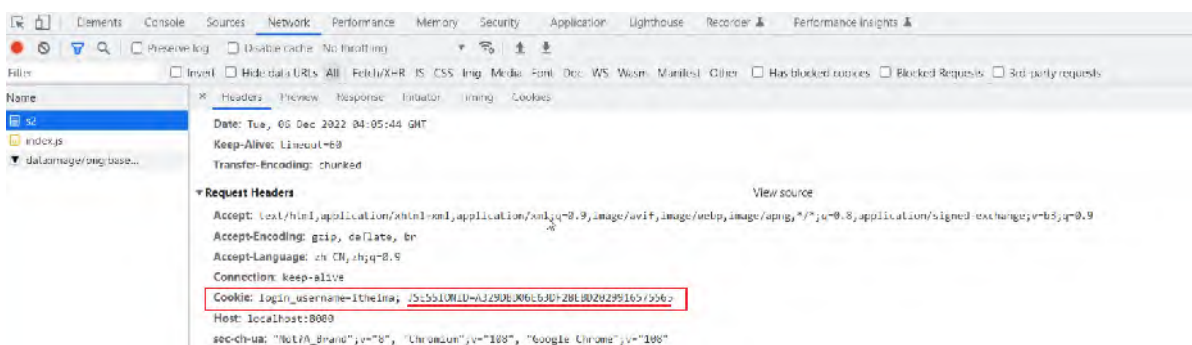
```

#### A. 访问 s1 接口, <http://localhost:8080/s1>



请求完成之后, 在响应头中, 就会看到有一个Set-Cookie的响应头, 里面响应回来了一个Cookie, 就是JSESSIONID, 这个就是服务端会话对象 Session 的ID。

#### B. 访问 s2 接口, <http://localhost:8080/s2>



接下来，在后续的每次请求时，都会将Cookie的值，携带到服务端，那服务端呢，接收到Cookie之后，会自动的根据JSESSIONID的值，找到对应的会话对象Session。

那经过这两步测试，大家也会看到，在控制台输出如下日志：

```
12:04:05.331 INFO 13456 --- [io-8080-exec-10] c.itheima.controller.SessionController : HttpSession-s1: 1675666466
12:05:44.115 INFO 13456 --- [nio-8080-exec-3] c.itheima.controller.SessionController : HttpSession-s2: 1675666466
12:05:44.115 INFO 13456 --- [nio-8080-exec-3] c.itheima.controller.SessionController : loginUser: tom
```

两次请求，获取到的Session会话对象的hashCode是一样的，就说明是同一个会话对象。而且，第一次请求时，往Session会话对象中存储的值，第二次请求时，也获取到了。那这样，我们就可以通过Session会话对象，在同一个会话的多次请求之间来进行数据共享了。

## 优缺点

- 优点：Session是存储在服务端的，安全
- 缺点：
  - 服务器集群环境下无法直接使用Session
  - 移动端APP(Android、IOS)中无法使用Cookie
  - 用户可以自己禁用Cookie
  - Cookie不能跨域

PS：Session 底层是基于Cookie实现的会话跟踪，如果Cookie不可用，则该方案，也就失效了。

服务器集群环境为何无法使用Session？



- 首先第一点，我们现在所开发的项目，一般都不会只部署在一台服务器上，因为一台服务器会存在一个很大的问题，就是单点故障。所谓单点故障，指的就是一旦这台服务器挂了，整个应用都没法访问了。



- 所以在现在的企业项目开发当中，最终部署的时候都是以集群的形式来进行部署，也就是同一个项目它会部署多份。比如这个项目我们现在就部署了 3 份。
- 而用户在访问的时候，到底访问这三台其中的哪一台？其实用户在访问的时候，他会访问一台前置的服务器，我们叫负载均衡服务器，我们在后面项目当中会详细讲解。目前大家先有一个印象负载均衡服务器，它的作用就是将前端发起的请求均匀的分发给后面的这三台服务器。



- 此时假如我们通过 `session` 来进行会话跟踪，可能就会存在这样一个问题。用户打开浏览器要进行登录操作，此时会发起登录请求。登录请求到达负载均衡服务器，将这个请求转给了第一台 Tomcat 服务器。

Tomcat 服务器接收到请求之后，要获取到会话对象 `session`。获取到会话对象 `session` 之后，要给浏览器响应数据，最终在给浏览器响应数据的时候，就会携带这么一个 `cookie` 的名字，就是 `JSESSIONID`，下一次再请求的时候，是不是又会将 `Cookie` 携带到服务端？

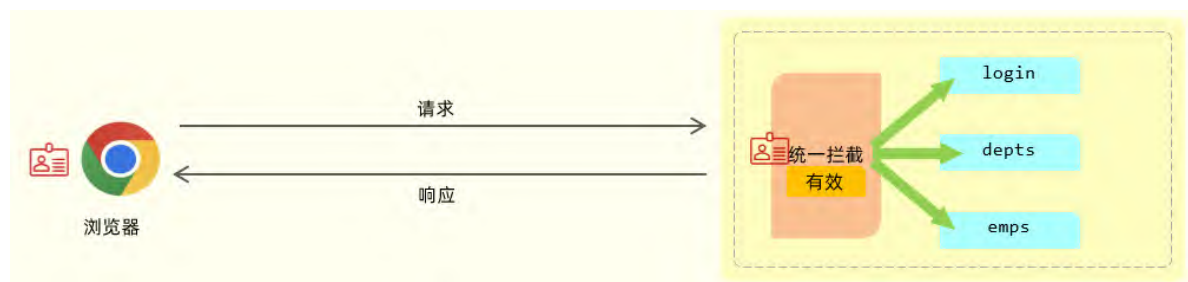
好。此时假如又执行了一次查询操作，要查询部门的数据。这次请求到达负载均衡服务器之后，负载均衡服务器将这次请求转给了第二台 Tomcat 服务器，此时他就要到第二台 Tomcat 服务器当中。根据 `JSESSIONID` 也就是对应的 `session` 的 ID 值，要找对应的 `session` 会话对象。

我想请问在第二台服务器当中有没有这个ID的会话对象 `Session`，是没有的。此时是不是就出现问题了？我同一个浏览器发起了 2 次请求，结果获取到的不是同一个会话对象，这就是 `Session` 这种会话跟踪方案它的缺点，在服务器集群环境下无法直接使用 `Session`。

大家会看到上面这两种传统的会话技术，在现在的企业开发当中是不是会存在很多的问题。为了解决这些问题，在现在的企业开发当中，基本上都会采用第三种方案，通过令牌技术来进行会话跟踪。接下来我们就来介绍一下令牌技术，来看一下令牌技术又是如何跟踪会话的。

### 2.2.2.3 方案三 - 令牌技术

这里我们所提到的令牌，其实它就是一个用户身份的标识，看似很高大上，很神秘，其实本质就是一个字符串。



如果通过令牌技术来跟踪会话，我们就可以在浏览器发起请求。在请求登录接口的时候，如果登录成功，我就可以生成一个令牌，令牌就是用户的合法身份凭证。接下来我在响应数据的时候，我就可以直接将令牌响应给前端。

接下来我们在前端程序当中接收到令牌之后，就需要将这个令牌存储起来。这个存储可以存储在 `cookie` 当中，也可以存储在其他存储空间 (比如: `localStorage`) 当中。

接下来，在后续的每一次请求当中，都需要将令牌携带到服务端。携带到服务端之后，接下来我们就需要来校验令牌的有效性。如果令牌是有效的，就说明用户已经执行了登录操作，如果令牌是无效的，就说明用户之前并未执行登录操作。

此时，如果是在同一次会话的多次请求之间，我们想共享数据，我们就可以将共享的数据存储在令牌当中就可以了。

### 优缺点

- 优点：
  - 支持PC端、移动端
  - 解决集群环境下的认证问题
  - 减轻服务器的存储压力（无需在服务器端存储）
- 缺点：需要自己实现（包括令牌的生成、令牌的传递、令牌的校验）

针对于这三种方案，现在企业开发当中使用的最多的就是第三种令牌技术进行会话跟踪。而前面的这两种传统的方案，现在企业项目开发当中已经很少使用了。所以在我们的课程当中，我们也将会采用令牌技术来解决案例项目当中的会话跟踪问题。

## 2.3 JWT令牌

前面我们介绍了基于令牌技术来实现会话追踪。这里所提到的令牌就是用户身份的标识，其本质就是一个字符串。令牌的形式有很多，我们使用的是功能强大的 JWT令牌。

### 2.3.1 介绍

JWT全称：JSON Web Token （官网：<https://jwt.io/>）

- 定义了一种简洁的、自包含的格式，用于在通信双方以json数据格式安全的传输信息。由于数字签名的存在，这些信息是可靠的。

简洁：是指jwt就是一个简单的字符串。可以在请求参数或者是请求头当中直接传递。

自包含：指的是jwt令牌，看似是一个随机的字符串，但是我们是可以根据自身的需求在jwt令牌中存储自定义的数据内容。如：可以直接在jwt令牌中存储用户的相关信息。

简单来讲，jwt就是将原始的json数据格式进行了安全的封装，这样就可以直接基于jwt在通信双方安全的进行信息传输了。

JWT的组成：（JWT令牌由三个部分组成，三个部分之间使用英文的点来分割）

- 第一部分：Header(头)，记录令牌类型、签名算法等。 例如：  
`{"alg": "HS256", "type": "JWT"}`
- 第二部分：Payload(有效载荷)，携带一些自定义信息、默认信息等。 例如：  
`{"id": "1", "username": "Tom"}`
- 第三部分：Signature(签名)，防止Token被篡改、确保安全性。将header、payload，并加入指定密钥，通过指定签名算法计算而来。

签名的目的就是防止jwt令牌被篡改，而正是因为jwt令牌最后一个部分数字签名的存在，所以整个jwt令牌是非常安全可靠的。一旦jwt令牌当中任何一个部分、任何一个字符被篡改了，整个令牌在校验的时候都会失败，所以它是非常安全可靠的。

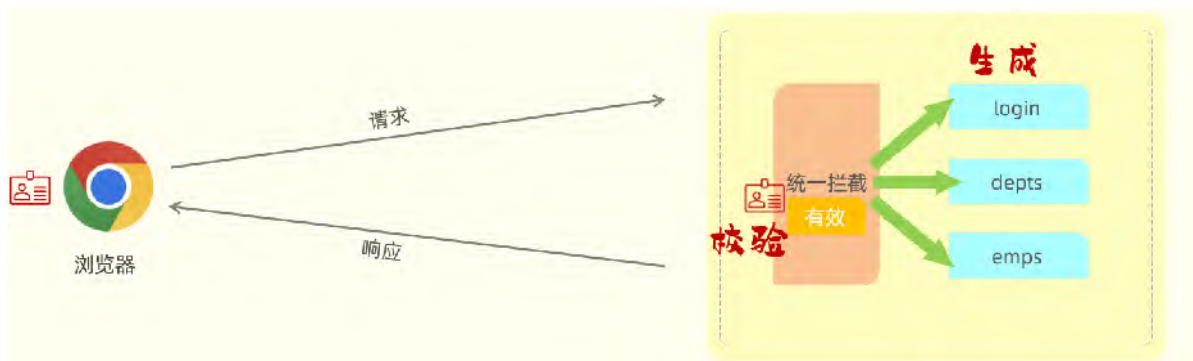
```
{"alg": "HS256", "type": "JWT"} . {"name": "Tom", "iat": 1516239022} . 数字签名(header.payload, secret)
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmFtZSI6ImRvbSUiLCJ1eWYyMm1hdCUyMiUzQTE1MTYyMzkwMjI1N0Q=.Sf1KxwR3SMekKF2QT4fupMeJf...
```

JWT是如何将原始的JSON格式数据，转变为字符串的呢？

其实在生成JWT令牌时，会对JSON格式的数据进行一次编码：进行base64编码

Base64：是一种基于64个可打印的字符来表示二进制数据的编码方式。既然能编码，那也就意味着也能解码。所使用的64个字符分别是A到Z、a到z、0-9，一个加号，一个斜杠，加起来就是64个字符。任何数据经过base64编码之后，最终就会通过这64个字符来表示。当然还有一个符号，那就是等号。等号它是一个补位的符号

需要注意的是Base64是编码方式，而不是加密方式。



JWT令牌最典型的应用场景就是登录认证：

1. 在浏览器发起请求来执行登录操作，此时会访问登录的接口，如果登录成功之后，我们需要生成一个jwt令牌，将生成的 jwt令牌返回给前端。
2. 前端拿到jwt令牌之后，会将jwt令牌存储起来。在后续的每一次请求中都会将jwt令牌携带到服务端。
3. 服务端统一拦截请求之后，先来判断一下这次请求有没有把令牌带过来，如果没有带过来，直接拒绝访问，如果带过来了，还要校验一下令牌是否是有效。如果有效，就直接放行进行请求的处理。

在JWT登录认证的场景中我们发现，整个流程当中涉及到两步操作：

1. 在登录成功之后，要生成令牌。
2. 每一次请求当中，要接收令牌并对令牌进行校验。

稍后我们再来学习如何来生成jwt令牌，以及如何来校验jwt令牌。



### 2.3.2 生成和校验

简单介绍了JWT令牌以及JWT令牌的组成之后，接下来我们就来学习基于Java代码如何生成和校验JWT令牌。

首先我们先来实现JWT令牌的生成。要想使用JWT令牌，需要先引入JWT的依赖：

```
1  <!-- JWT依赖-->
2  <dependency>
3      <groupId>io.jsonwebtoken</groupId>
4      <artifactId>jjwt</artifactId>
5      <version>0.9.1</version>
6  </dependency>
```

在引入完JWT来赖后，就可以调用工具包中提供的API来完成JWT令牌的生成和校验

工具类：Jwts

生成JWT代码实现：


```
1  @Test
2  public void genJwt() {
3      Map<String, Object> claims = new HashMap<>();
4      claims.put("id", 1);
5      claims.put("username", "Tom");
6
7      String jwt = Jwts.builder()
8          .setClaims(claims) //自定义内容(载荷)
9          .signWith(SignatureAlgorithm.HS256, "itheima") //签名算法
10
11          .setExpiration(new Date(System.currentTimeMillis() +
12              24*3600*1000)) //有效期
13          .compact();
14
15      System.out.println(jwt);
16  }
```

运行测试方法：


```
1  eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwizXhwIjoxNjc5NzI5NzAwfQ.fHi0Ub8npbyt71
   UqLXdDdLyipptLgxBUg_mSuGJtXtBk
```

输出的结果就是生成的JWT令牌，，通过英文的点分割对三个部分进行分割，我们可以将生成的令牌复制一下，然后打开JWT的官网，将生成的令牌直接放在Encoded位置，此时就会自动的将令牌解析出来。



 **JWT**

DebuggerLibrariesIntroductionAsk

Crafted by  auth0

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjc5NzI5NzMwfQ.fHi0Ub8npbyt71UqLXDdLyipptLgxBUG_mSuGJtXtBk
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256"}
```

PAYLOAD: DATA

```
{  "id": 1,  "exp": 1672729730}
```

VERIFY SIGNATURE

```
 HMACSHA256(  base64UrlEncode(header) + ".",  base64UrlEncode(payload),  your-256-bit-secret)  secret base64 encoded
```

第一部分解析出来，看到JSON格式的原始数据，所使用的签名算法为HS256。

第二个部分是我们自定义的数据，之前我们自定义的数据就是id，还有一个exp代表的是我们所设置的过期时间。

由于前两个部分是base64编码，所以是可以直接解码出来。但最后一个部分并不是base64编码，是经过签名算法计算出来的，所以最后一个部分是不会解析的。

实现了JWT令牌的生成，下面我们接着使用Java代码来校验JWT令牌(解析生成的令牌)：

```
1  @Test
2  public void parseJwt() {
3      Claims claims = Jwts.parser()
4          .setSigningKey("itheima") //指定签名密钥（必须保证和生成令牌时使用
           相同的签名密钥）
5
6          .parseClaimsJws("eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjc5NzI5NzMwfQ.fHi0Ub8npbyt71UqLXDdLyipptLgxBUG_mSuGJtXtBk")
7
8          .getBody();
9
10     System.out.println(claims);
11 }
```

运行测试方法：

```
1  {id=1, exp=1672729730}
```

令牌解析后，我们可以看到id和过期时间，如果在解析的过程当中没有报错，就说明解析成功了。

下面我们做一个测试：把令牌header中的数字9变为8，运行测试方法后发现报错：

原header: eyJhbGciOiJIUzI1NiJ9

修改为: eyJhbGciOiJIUzI1NiJ8

```
io.jsonwebtoken.MalformedJwtException: Unable to read JSON value: {"alg":"HS256"}
```

```
at io.jsonwebtoken.impl.DefaultJwtParser.readValue(DefaultJwtParser.java:554)
at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:252)
at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:481)
at io.jsonwebtoken.impl.DefaultJwtParser.parseClaimsJws(DefaultJwtParser.java:541)
at com.itheima.TliasWebManagementApplicationTests.parseJwt(TliasWebManagementApplicationTests.java:43) <31 internal lines>
```

结论：篡改令牌中的任何一个字符，在对令牌进行解析时都会报错，所以JWT令牌是非常安全可靠的。

我们继续测试：修改生成令牌的时指定的过期时间，修改为1分钟

```
1  @Test
2  public void genJwt() {
3      Map<String, Object> claims = new HashMap<>();
4      claims.put("id", 1);
5      claims.put("username", "Tom");
6      String jwt = Jwts.builder()
7          .setClaims(claims) //自定义内容(载荷)
8          .signWith(SignatureAlgorithm.HS256, "itheima") //签名算法
9
10         .setExpiration(new Date(System.currentTimeMillis() +
11             60*1000)) //有效期60秒
12         .compact();
13
14         System.out.println(jwt);
15         //输出结果:
16         eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjc2MDA5NzU0fQ.RcVIR65AkGia
17         x-ID6FjW60eLFH3tPTKdoK7UtE4Alro
18     }
19
20     @Test
21     public void parseJwt() {
22         Claims claims = Jwts.parser()
23             .setSigningKey("itheima") //指定签名密钥
24             .parseClaimsJws("eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwiZXhwIjoxNjc2MDA5Nz
25             U0fQ.RcVIR65AkGiax-ID6FjW60eLFH3tPTKdoK7UtE4Alro")
26     }
```

```
21         .getBody();
22
23         System.out.println(claims);
24     }
```

等待1分钟之后运行测试方法发现也报错了，说明：JWT令牌过期后，令牌就失效了，解析的为非法令牌。

通过以上测试，我们在使用JWT令牌时需要注意：

- JWT校验时使用的签名密钥，必须和生成JWT令牌时使用的密钥是配套的。
- 如果JWT令牌解析校验时报错，则说明 JWT令牌被篡改 或 失效了，令牌非法。

### 2.3.3 登录下发令牌

JWT令牌的生成和校验的基本操作我们已经学习完了，接下来我们就需要在案例当中通过JWT令牌技术来跟踪会话。具体的思路我们前面已经分析过了，主要就是两步操作：

1. 生成令牌
  - 在登录成功之后来生成一个JWT令牌，并且把这个令牌直接返回给前端
2. 校验令牌
  - 拦截前端请求，从请求中获取到令牌，对令牌进行解析校验

那我们首先来完成：登录成功之后生成JWT令牌，并且把令牌返回给前端。

JWT令牌怎么返回给前端呢？此时我们就需要再来看一下接口文档当中关于登录接口的描述（主要看响应数据）：

- 响应数据

参数格式：application/json

参数说明：

名称	类型	是否必须	默认值	备注	其他信息
code	number	必须		响应码，1 成功；0 失败	
msg	string	非必须		提示信息	
data	string	必须		返回的数据，jwt令牌	

响应数据样例：

```
1  {
2      "code": 1,
3      "msg": "success",
4      "data":
5          "eyJhbGciOiJIUzI1NiJ9.eyJ1YXlwYXN0eXkiOiJmcm9udG91cyIsImV4cCI6MTYyMjMjIiwiaWF0IjE5MjA0OH0.KkUc_CXJZJ8Dd063eImx4H9Ojfr6XMJ-yVzaWCVZCo"
6  }
```

- 备注说明

用户登录成功后，系统会自动下发JWT令牌，然后在后续的每次请求中，都需要在请求头header中携带到服务端，请求头的名称为 `token` ， 值为 登录时下发的JWT令牌。

如果检测到用户未登录，则会返回如下固定错误信息：

```
1  {
2      "code": 0,
3      "msg": "NOT_LOGIN",
4      "data": null
5  }
```

解读完接口文档中的描述了，目前我们先来完成令牌的生成和令牌的下发，我们只需要生成一个令牌返回给前端就可以了。

## 实现步骤：

1. 引入JWT工具类

- 在项目工程下创建`com.itheima.utils`包，并把提供JWT工具类复制到该包下

2. 登录完成后，调用工具类生成JWT令牌并返回

## JWT工具类

```
1  public class JwtUtils {
2
3      private static String signKey = "itheima";//签名密钥
4      private static Long expire = 43200000L; //有效时间
5
6      /**
7       * 生成JWT令牌
```

```

8      * @param claims JWT第二部分负载 payload 中存储的内容
9      * @return
10     */
11     public static String generateJwt(Map<String, Object> claims){
12         String jwt = Jwts.builder()
13             .addClaims(claims)//自定义信息（有效载荷）
14             .signWith(SignatureAlgorithm.HS256, signKey)//签名算
            法（头部）
15             .setExpiration(new Date(System.currentTimeMillis() +
            expire))//过期时间
16             .compact();
17         return jwt;
18     }
19
20     /**
21     * 解析JWT令牌
22     * @param jwt JWT令牌
23     * @return JWT第二部分负载 payload 中存储的内容
24     */
25     public static Claims parseJWT(String jwt){
26         Claims claims = Jwts.parser()
27             .setSigningKey(signKey)//指定签名密钥
28             .parseClaimsJws(jwt)//指定令牌Token
29             .getBody();
30         return claims;
31     }
32 }
33

```

### 登录成功，生成JWT令牌并返回

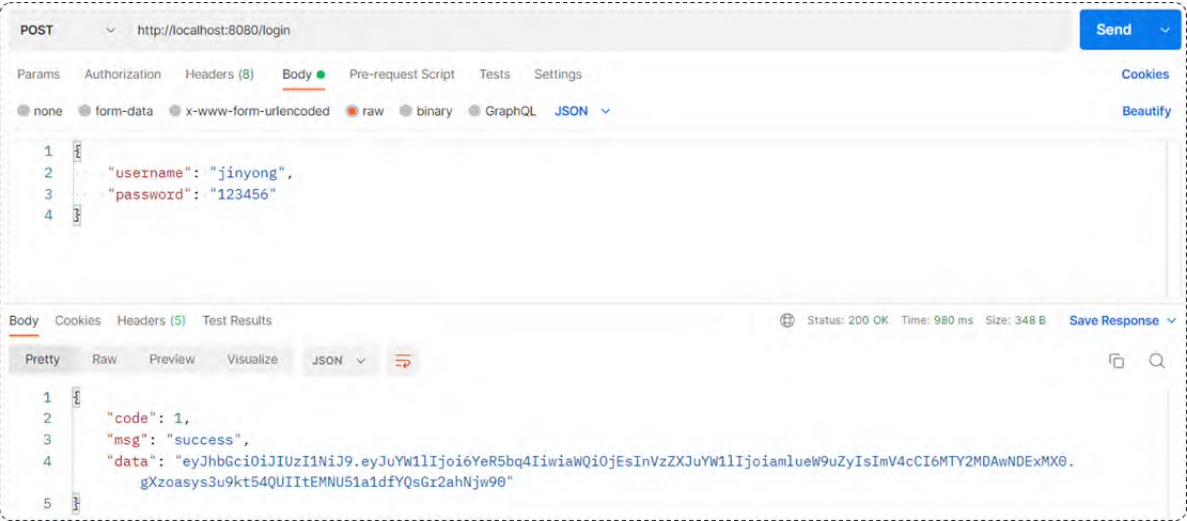
```

1  @RestController
2  @Slf4j
3  public class LoginController {
4      //依赖业务层对象
5      @Autowired
6      private EmpService empService;
7
8      @PostMapping("/login")
9      public Result login(@RequestBody Emp emp) {
10         //调用业务层：登录功能
11         Emp loginEmp = empService.login(emp);
12

```

```
13         //判断：登录用户是否存在
14         if(loginEmp !=null ){
15             //自定义信息
16             Map<String , Object> claims = new HashMap<>();
17             claims.put("id", loginEmp.getId());
18             claims.put("username",loginEmp.getUsername());
19             claims.put("name",loginEmp.getName());
20
21             //使用JWT工具类，生成身份令牌
22             String token = JwtUtils.generateJwt(claims);
23             return Result.success(token);
24         }
25         return Result.error("用户名或密码错误");
26     }
27 }
```

重启服务，打开postman测试登录接口：



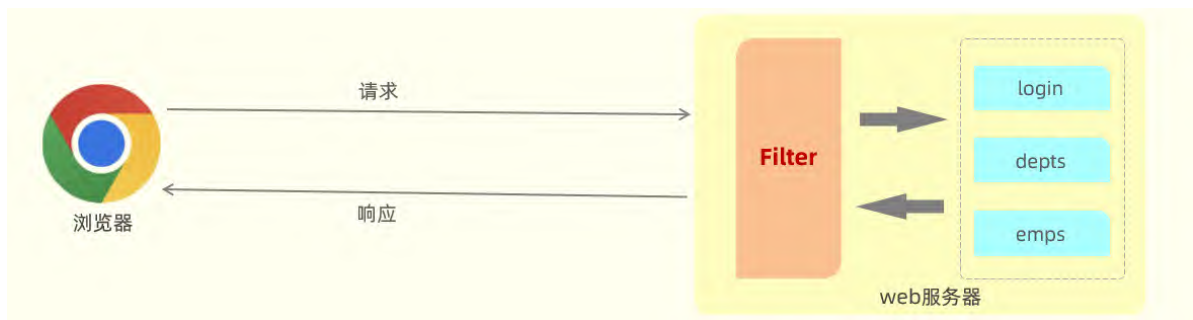
打开浏览器完成前后端联调操作：利用开发者工具，抓取一下网络请求







- 过滤器一般完成一些通用的操作，比如：登录校验、统一编码处理、敏感字符处理等。



下面我们通过Filter快速入门程序掌握过滤器的基本使用操作：

- 第1步，定义过滤器：1.定义一个类，实现 Filter 接口，并重写其所有方法。
- 第2步，配置过滤器：Filter类上加 @WebFilter 注解，配置拦截资源的路径。引导类上加 @ServletComponentScan 开启Servlet组件支持。

## 定义过滤器

```
1 //定义一个类，实现一个标准的Filter过滤器的接口
2 public class DemoFilter implements Filter {
3     @Override //初始化方法，只调用一次
4     public void init(FilterConfig filterConfig) throws
5         ServletException {
6         System.out.println("init 初始化方法执行了");
7     }
8     @Override //拦截到请求之后调用，调用多次
9     public void doFilter(ServletRequest request, ServletResponse
10         response, FilterChain chain) throws IOException, ServletException {
11         System.out.println("Demo 拦截到了请求...放行前逻辑");
12         //放行
13         chain.doFilter(request, response);
14     }
15     @Override //销毁方法，只调用一次
16     public void destroy() {
17         System.out.println("destroy 销毁方法执行了");
18     }
19 }
```

- init方法：过滤器的初始化方法。在web服务器启动的时候会自动的创建Filter过滤器对象，在创建过滤器对象的时候会自动调用init初始化方法，这个方法只会被调用一次。

- doFilter方法：这个方法是在每一次拦截到请求之后都会被调用，所以这个方法是被调用多次的，每拦截到一次请求就会调用一次doFilter()方法。
- destroy方法：是销毁的方法。当我们关闭服务器的时候，它会自动的调用销毁方法destroy，而这个销毁方法也只会调用一次。

在定义完Filter之后，Filter其实并不会生效，还需要完成Filter的配置，Filter的配置非常简单，只需要在Filter类上添加一个注解：@WebFilter，并指定属性urlPatterns，通过这个属性指定过滤器要拦截哪些请求

```
1  @WebFilter(urlPatterns = "/*") //配置过滤器要拦截的请求路径（ /* 表示拦截浏览器的所有请求 ）
2  public class DemoFilter implements Filter {
3      @Override //初始化方法，只调用一次
4      public void init(FilterConfig filterConfig) throws
ServletException {
5          System.out.println("init 初始化方法执行了");
6      }
7
8      @Override //拦截到请求之后调用，调用多次
9      public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
10         System.out.println("Demo 拦截到了请求...放行前逻辑");
11         //放行
12         chain.doFilter(request, response);
13     }
14
15     @Override //销毁方法，只调用一次
16     public void destroy() {
17         System.out.println("destroy 销毁方法执行了");
18     }
19 }
```

当我们在Filter类上面加了@WebFilter注解之后，接下来我们还需要在启动类上面加上一个注解@ComponentScan，通过这个@ComponentScan注解来开启SpringBoot项目对于Servlet组件的支持。

```

1  @ServletComponentScan
2  @SpringBootApplication
3  public class TliasWebManagementApplication {
4
5      public static void main(String[] args) {
6          SpringApplication.run(TliasWebManagementApplication.class,
7              args);
8      }
9  }

```

重新启动服务，打开浏览器，执行部门管理的请求，可以看到控制台输出了过滤器中的内容：

```

2022-12-08 17:11:36.343 INFO 13144 [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
拦截到了请求
拦截到了请求
拦截到了请求
拦截到了请求
拦截到了请求

```

注意事项：

在过滤器Filter中，如果不执行放行操作，将无法访问后面的资源。 放行操作：

```
chain.doFilter(request, response);
```

现在我们已经完成了Filter过滤器的基本使用，下面我们将学习Filter过滤器在使用过程中的一些细节。

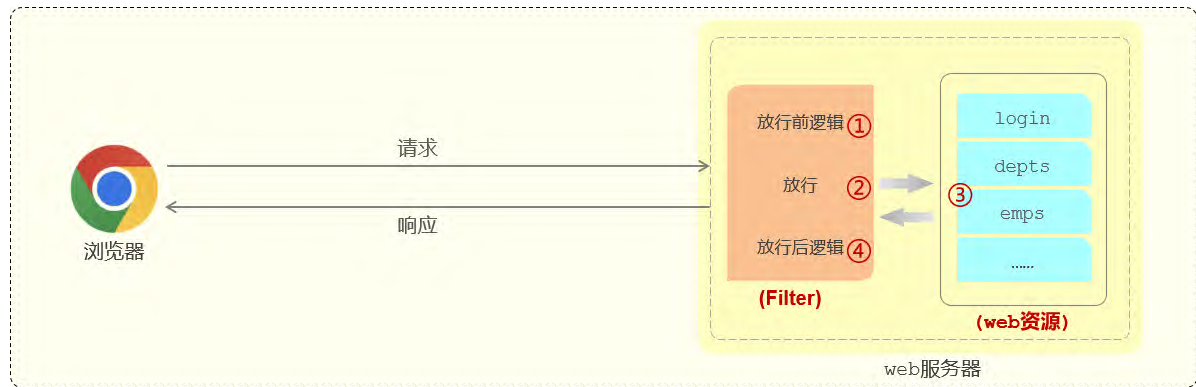
### 2.4.2 Filter详解

Filter过滤器的快速入门程序我们已经完成了，接下来我们就要详细的介绍一下过滤器Filter在使用中的一些细节。主要介绍以下3个方面的细节：

1. 过滤器的执行流程
2. 过滤器的拦截路径配置
3. 过滤器链

### 2.4.2.1 执行流程

首先我们先看下过滤器的执行流程：



过滤器当中我们拦截到了请求之后，如果希望继续访问后面的web资源，就要执行放行操作，放行就是调用 `FilterChain`对象当中的`doFilter()`方法，在调用`doFilter()`这个方法之前所编写的代码属于放行之前的逻辑。

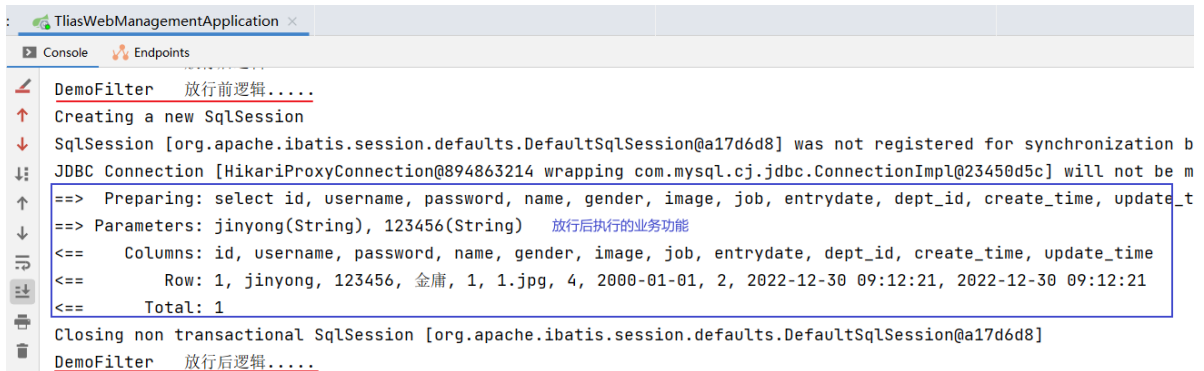
在放行后访问完 web 资源之后还会回到过滤器当中，回到过滤器之后如有需求还可以执行放行之后的逻辑，放行之后的逻辑我们写在`doFilter()`这行代码之后。

```
1  @WebFilter(urlPatterns = "/*")
2  public class DemoFilter implements Filter {
3
4      @Override //初始化方法，只调用一次
5      public void init(FilterConfig filterConfig) throws
ServletException {
6          System.out.println("init 初始化方法执行了");
7      }
8
9      @Override
10     public void doFilter(ServletRequest servletRequest,
ServletResponse servletResponse, FilterChain filterChain) throws
IOException, ServletException {
11
12         System.out.println("DemoFilter 放行前逻辑.....");
13
14         //放行请求
15         filterChain.doFilter(servletRequest,servletResponse);
16
17         System.out.println("DemoFilter 放行后逻辑.....");
18
19     }
20 }
```

```

21      @Override //销毁方法，只调用一次
22      public void destroy() {
23          System.out.println("destroy 销毁方法执行了");
24      }
25  }

```



#### 2.4.2.2 拦截路径

执行流程我们搞清楚之后，接下来再来介绍一下过滤器的拦截路径，Filter可以根据需求，配置不同的拦截资源路径：

拦截路径	urlPatterns值	含义
拦截具体路径	/login	只有访问 /login 路径时，才会被拦截
目录拦截	/emps/*	访问/emps下的所有资源，都会被拦截
拦截所有	/*	访问所有资源，都会被拦截

下面我们来测试"拦截具体路径"：

```

1  @WebFilter(urlPatterns = "/login") //拦截/login具体路径
2  public class DemoFilter implements Filter {
3      @Override
4      public void doFilter(ServletRequest servletRequest,
5                          ServletResponse servletResponse, FilterChain filterChain) throws
6      IOException, ServletException {
7          System.out.println("DemoFilter 放行前逻辑.....");
8          //放行请求
9          filterChain.doFilter(servletRequest, servletResponse);
10         System.out.println("DemoFilter 放行后逻辑.....");
11     }
12
13

```

```

14     @Override
15     public void init(FilterConfig filterConfig) throws
ServletException {
16         Filter.super.init(filterConfig);
17     }
18
19     @Override
20     public void destroy() {
21         Filter.super.destroy();
22     }
23 }

```

测试1：访问部门管理请求，发现过滤器没有拦截请求

The screenshot shows a web application interface for 'TJAS智能学习辅助系统' (TJAS Intelligent Learning Assistance System). The '部门管理' (Department Management) page displays a table of departments:

序号	部门名称	最后操作时间	操作
1	咨询部	2022-12-30 09:12:11	<a href="#">新增</a> <a href="#">删除</a>
2	就业部	2022-12-30 09:12:11	<a href="#">新增</a> <a href="#">删除</a>
3	人事部	2022-12-30 09:12:11	<a href="#">新增</a> <a href="#">删除</a>

Below the table, the database logs for the 'TliasWebManagementApplication' show the following SQL query and results:

```

Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@19d01c1d] was not registered for synchronization because synchronization is not active
JDBC Connection [HikariProxyConnection@526710043 wrapping com.mysql.cj.jdbc.ConnectionImpl@10692f96] will not be managed by Spring
==> Preparing: select id, name, create_time, update_time from dept
==> Parameters:
<== Columns: id, name, create_time, update_time
<== Row: 3, 咨询部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
<== Row: 4, 就业部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
<== Row: 5, 人事部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
<== Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@19d01c1d]

```

测试2：访问登录请求/login，发现过滤器拦截请求

The screenshot shows the database logs for the 'TliasWebManagementApplication' for a login request. The logs indicate that the request was intercepted by the 'DemoFilter' before reaching the database:

```

DemoFilter 放行前逻辑....
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@45eb9fe6] was not registered for synchronization because
JDBC Connection [HikariProxyConnection@1883710508 wrapping com.mysql.cj.jdbc.ConnectionImpl@10692f96] will not be managed
==> Preparing: select id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time from
==> Parameters: jinyong(String), 123456(String)
<== Columns: id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time
<== Row: 1, jinyong, 123456, 金庸, 1, 1.jpg, 4, 2000-01-01, 2, 2022-12-30 09:12:21, 2022-12-30 09:12:21
<== Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@45eb9fe6]
DemoFilter 放行后逻辑....

```

下面我们来测试"目录拦截"：



```

1  @WebFilter(urlPatterns = "/depts/*") //拦截所有以/depts开头，后面是什么
   无所谓
2  public class DemoFilter implements Filter {
3      @Override
4      public void doFilter(ServletRequest servletRequest,
5                          ServletResponse servletResponse, FilterChain filterChain) throws
6                          IOException, ServletException {
7          System.out.println("DemoFilter 放行前逻辑.....");
8          //放行请求
9          filterChain.doFilter(servletRequest,servletResponse);
10         System.out.println("DemoFilter 放行后逻辑.....");
11     }
12
13
14     @Override
15     public void init(FilterConfig filterConfig) throws
16     ServletException {
17         Filter.super.init(filterConfig);
18     }
19
20     @Override
21     public void destroy() {
22         Filter.super.destroy();
23     }
24 }

```

测试1：访问部门管理请求，发现过滤器拦截了请求

```

TliasWebManagementApplication x
Console Endpoints
DemoFilter 放行前逻辑.....
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4efcbd97] was not registered for synchronization
JDBC Connection [HikariProxyConnection@1810969442 wrapping com.mysql.cj.jdbc.ConnectionImpl@54f2d16a] will not be
==> Preparing: select id, name, create_time, update_time from dept
==> Parameters:
<== Columns: id, name, create_time, update_time
<== Row: 3, 咨询部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
<== Row: 4, 就业部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
<== Row: 5, 人事部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
<== Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@4efcbd97]
DemoFilter 放行后逻辑.....

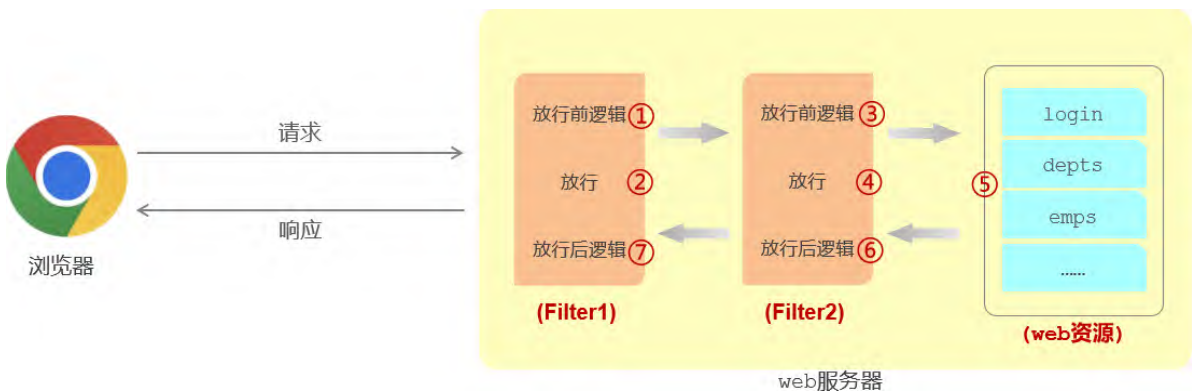
```

测试2：访问登录请求/login，发现过滤器没有拦截请求

```
TliasWebManagementApplication x
Console Endpoints
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@6f1077b9] was not registered for synchronization
JDBC Connection [HikariProxyConnection@581719332 wrapping com.mysql.cj.jdbc.ConnectionImpl@54f2d16a] will not be
==> Preparing: select id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update
==> Parameters: jinyong(String), 123456(String)
<== Columns: id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time
<== Row: 1, jinyong, 123456, 金庸, 1, 1.jpg, 4, 2000-01-01, 2, 2022-12-30 09:12:21, 2022-12-30 09:12:21
<== Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@6f1077b9]
```

### 2.4.2.3 过滤器链

最后我们在来介绍下过滤器链，什么是过滤器链呢？所谓过滤器链指的是在一个web应用程序当中，可以配置多个过滤器，多个过滤器就形成了一个过滤器链。



比如：在我们web服务器当中，定义了两个过滤器，这两个过滤器就形成了一个过滤器链。

而这个链上的过滤器在执行的时候会一个一个的执行，会先执行第一个Filter，放行之后再执行第二个Filter，如果执行到了最后一个过滤器放行之后，才会访问对应的web资源。

访问完web资源之后，按照我们刚才所介绍的过滤器的执行流程，还会回到过滤器当中来执行过滤器放行后的逻辑，而在执行放行后的逻辑的时候，顺序是反着的。

先要执行过滤器2放行之后的逻辑，再来执行过滤器1放行之后的逻辑，最后在给浏览器响应数据。

以上就是当我们在web应用当中配置了多个过滤器，形成了这样一个过滤器链以及过滤器链的执行顺序。下面我们通过idea来验证下过滤器链。

验证步骤：

1. 在filter包下再来新建一个Filter过滤器类：AbcFilter
2. 在AbcFilter过滤器中编写放行前和放行后逻辑
3. 配置AbcFilter过滤器拦截请求路径为：/\*
4. 重启SpringBoot服务，查看DemoFilter、AbcFilter的执行日志



### AbcFilter过滤器

```
1  @WebFilter(urlPatterns = "/*")
2  public class AbcFilter implements Filter {
3      @Override
4      public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws IOException, ServletException {
5          System.out.println("Abc 拦截到了请求... 放行前逻辑");
6
7          //放行
8          chain.doFilter(request, response);
9
10         System.out.println("Abc 拦截到了请求... 放行后逻辑");
11     }
12 }
```

### DemoFilter过滤器

```
1  @WebFilter(urlPatterns = "/*")
2  public class DemoFilter implements Filter {
3      @Override
4      public void doFilter(ServletRequest servletRequest,
        ServletResponse servletResponse, FilterChain filterChain) throws
        IOException, ServletException {
5          System.out.println("DemoFilter 放行前逻辑.....");
6
7          //放行请求
8          filterChain.doFilter(servletRequest, servletResponse);
9
10         System.out.println("DemoFilter 放行后逻辑.....");
11     }
12 }
```

打开浏览器访问登录接口：

通过控制台日志的输出，大家发现AbcFilter先执行DemoFilter后执行，这是为什么呢？

其实是和过滤器的类名有关系。以注解方式配置的Filter过滤器，它的执行优先级是按时过滤器类名的自动排序确定的，类名排名越靠前，优先级越高。

假如我们想让DemoFilter先执行，怎么办呢？答案就是修改类名。

测试：修改AbcFilter类名为XbcFilter，运行程序查看控制台日志

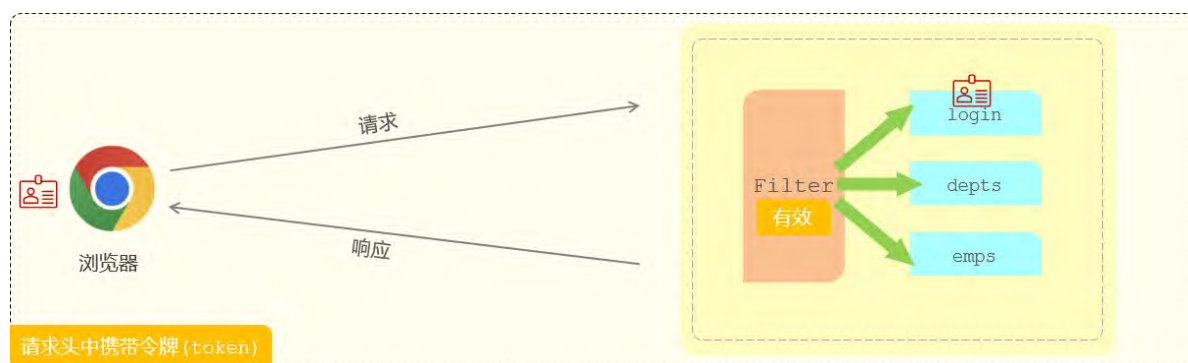
```
1  @WebFilter(urlPatterns = "/*")
2  public class XbcFilter implements Filter {
3      @Override
4      public void doFilter(ServletRequest request, ServletResponse
        response, FilterChain chain) throws IOException, ServletException {
5          System.out.println("Xbc 拦截到了请求...放行前逻辑");
6
7          //放行
8          chain.doFilter(request, response);
9
10         System.out.println("Xbc 拦截到了请求...放行后逻辑");
11     }
12 }
13
```

到此，关于过滤器的使用细节，我们已经全部介绍完毕了。

### 2.4.3 登录校验-Filter

#### 2.4.3.1 分析

过滤器Filter的快速入门以及使用细节我们已经介绍完了，接下来最后一步，我们需要使用过滤器Filter来完成案例当中的登录校验功能。



我们先来回顾下前面分析过的登录校验的基本流程：

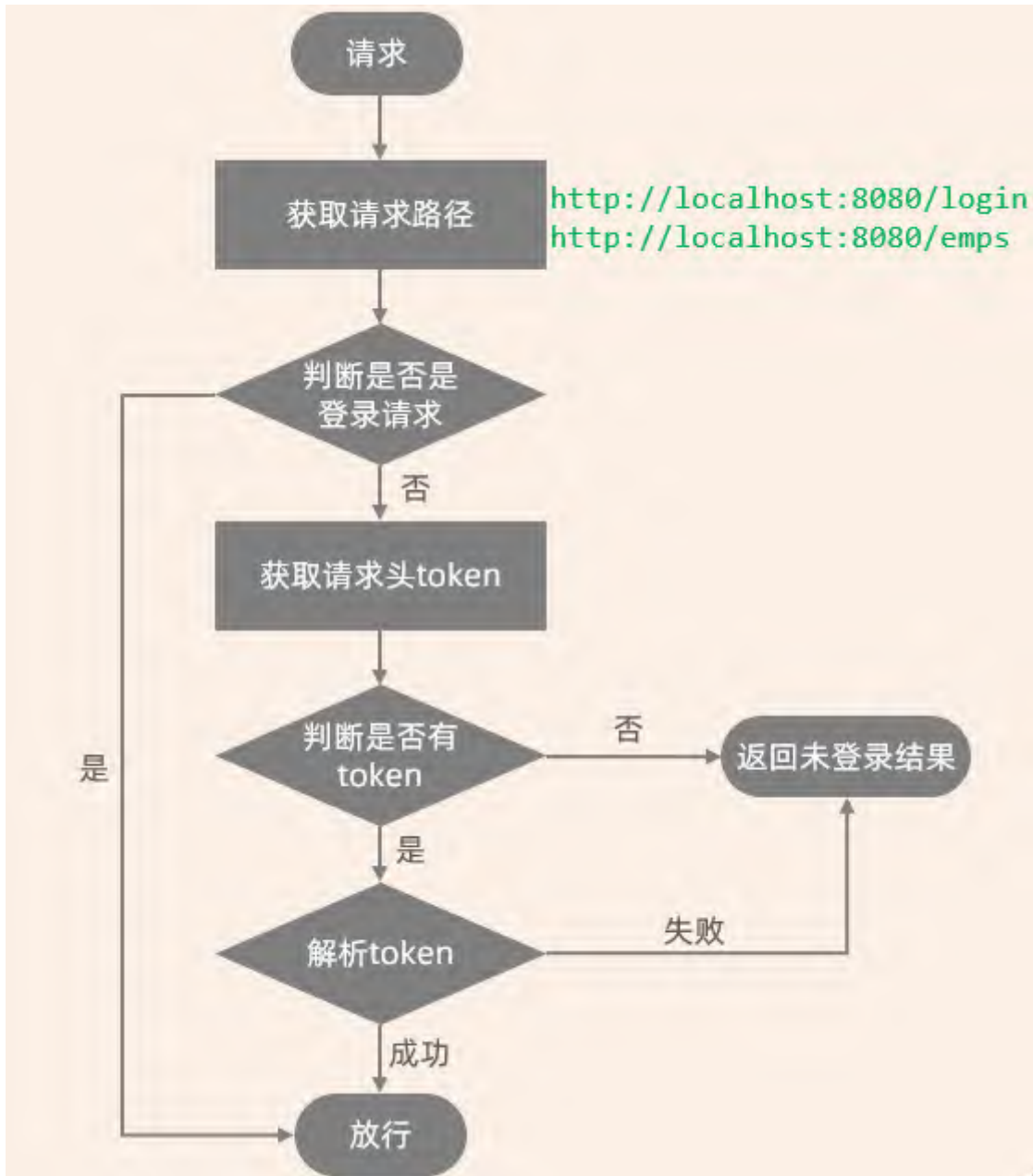
- 要进入到后台管理系统，我们必须先完成登录操作，此时就需要访问登录接口login。
- 登录成功之后，我们会在服务端生成一个JWT令牌，并且把JWT令牌返回给前端，前端会将JWT令牌存储下来。
- 在后续的每一次请求当中，都会将JWT令牌携带到服务端，请求到达服务端之后，要想去访问对应的业务功能，此时我们必须先要校验令牌的有效性。
- 对于校验令牌的这一块操作，我们使用登录校验的过滤器，在过滤器当中来校验令牌的有效性。如果令牌是无效的，就响应一个错误的信息，也不会再去放行访问对应的资源了。如果令牌存在，并且它是有效的，此时就会放行去访问对应的web资源，执行相应的业务操作。

大概清楚了在Filter过滤器的实现步骤了，那在正式开发登录校验过滤器之前，我们思考两个问题：

1. 所有的请求，拦截到了之后，都需要校验令牌吗？
  - 答案：登录请求例外
2. 拦截到请求后，什么情况下才可以放行，执行业务操作？
  - 答案：有令牌，且令牌校验通过（合法）；否则都返回未登录错误结果

#### 2.4.3.2 具体流程

我们要完成登录校验，主要是利用Filter过滤器实现，而Filter过滤器的流程步骤：



基于上面的业务流程，我们分析出具体的操作步骤：

1. 获取请求url
2. 判断请求url中是否包含login，如果包含，说明是登录操作，放行
3. 获取请求头中的令牌 (token)
4. 判断令牌是否存在，如果不存在，返回错误结果 (未登录)
5. 解析token，如果解析失败，返回错误结果 (未登录)
6. 放行

2.4.3.3 代码实现

分析清楚了以上的问题后，我们就参照接口文档来开发登录功能了，登录接口描述如下：

• 基本信息

```
1  请求路径：/login
2
3  请求方式：POST
4
5  接口描述：该接口用于员工登录Tlias智能学习辅助系统，登录完毕后，系统下发JWT令牌。
```

• 请求参数

参数格式：application/json

参数说明：

名称	类型	是否必须	备注
username	string	必须	用户名
password	string	必须	密码

请求数据样例：

```
1  {
2      "username": "jinyong",
3      "password": "123456"
4  }
```

• 响应数据

参数格式：application/json

参数说明：

名称	类型	是否必须	默认值	备注	其他信息
code	number	必须		响应码，1 成功；0 失败	
msg	string	非必须		提示信息	
data	string	必须		返回的数据，jwt令牌	

响应数据样例：



```

1  {
2      "code": 1,
3      "msg": "success",
4      "data":
        "eyJhbGciOiJIUzI1NiJ9.eyJ1Ijoi6YeR5bq4IiwiaWQiOiJEsInVzZXJuYW11IjoiamlueW9uZyIsImV4cCI6MTY2MjIwNzA0OH0.KkUc_CXJZJ8Dd063eImx4H9Ojfr6XMJ-yVzaWCVZCo"
5  }

```

- 备注说明

用户登录成功后，系统会自动下发JWT令牌，然后在后续的每次请求中，都需要在请求头header中携带到服务端，请求头的名称为 `token`，值为 登录时下发的JWT令牌。

如果检测到用户未登录，则会返回如下固定错误信息：

```

1  {
2      "code": 0,
3      "msg": "NOT_LOGIN",
4      "data": null
5  }

```

### 登录校验过滤器: LoginCheckFilter

```

1  @Slf4j
2  @WebFilter(urlPatterns = "/*") //拦截所有请求
3  public class LoginCheckFilter implements Filter {
4
5      @Override
6      public void doFilter(ServletRequest servletRequest,
7                          ServletResponse servletResponse, FilterChain chain) throws
8                          IOException, ServletException {
9
10         //前置：强制转换为http协议的请求对象、响应对象 （转换原因：要使用子类中特有方法）
11
12         HttpServletRequest request = (HttpServletRequest)
13             servletRequest;
14
15         HttpServletResponse response = (HttpServletResponse)
16             servletResponse;
17
18         //1.获取请求url
19         String url = request.getRequestURL().toString();
20         log.info("请求路径: {}", url); //请求路径:
21         http://localhost:8080/login
22     }
23 }

```

```
16 //2.判断请求url中是否包含login，如果包含，说明是登录操作，放行
17 if(url.contains("/login")){
18     chain.doFilter(request, response);//放行请求
19     return;//结束当前方法的执行
20 }
21
22
23 //3.获取请求头中的令牌（token）
24 String token = request.getHeader("token");
25 log.info("从请求头中获取的令牌: {}",token);
26
27
28 //4.判断令牌是否存在，如果不存在，返回错误结果（未登录）
29 if(!StringUtils.hasLength(token)){
30     log.info("Token不存在");
31
32     Result responseResult = Result.error("NOT_LOGIN");
33     //把Result对象转换为JSON格式字符串（fastjson是阿里巴巴提供的
    用于实现对象和json的转换工具类）
34     String json = JSONObject.toJSONString(responseResult);
35     response.setContentType("application/json;charset=utf-
    8");
36     //响应
37     response.getWriter().write(json);
38
39     return;
40 }
41
42 //5.解析token，如果解析失败，返回错误结果（未登录）
43 try {
44     JwtUtils.parseJWT(token);
45 }catch (Exception e){
46     log.info("令牌解析失败!");
47
48     Result responseResult = Result.error("NOT_LOGIN");
49     //把Result对象转换为JSON格式字符串（fastjson是阿里巴巴提供的
    用于实现对象和json的转换工具类）
50     String json = JSONObject.toJSONString(responseResult);
51     response.setContentType("application/json;charset=utf-
    8");
52     //响应
53     response.getWriter().write(json);
54
55     return;
```

```

56         }
57
58
59         //6.放行
60         chain.doFilter(request, response);
61
62     }
63 }

```

在上述过滤器的功能实现中，我们使用到了一个第三方json处理的工具包fastjson。我们要想使用，需要引入如下依赖：

```

1  <dependency>
2      <groupId>com.alibaba</groupId>
3      <artifactId>fastjson</artifactId>
4      <version>1.2.76</version>
5  </dependency>

```

登录校验的过滤器我们编写完成了，接下来我们就可以重新启动服务来做一个测试：

测试前先把之前所编写的测试使用的过滤器，暂时注释掉。直接将@WebFilter注解给注释掉即可。

- 测试1：未登录是否可以访问部门管理页面

首先关闭浏览器，重新打开浏览器，在地址栏中输入：<http://localhost:9528/#/system/dept>

由于用户没有登录，登录校验过滤器返回错误信息，前端页面根据返回的错误信息结果，自动跳转到登录页面了



- 测试2：先进行登录操作，再访问部门管理页面

登录校验成功之后，可以正常访问相关业务操作页面



## 2.5 拦截器Interceptor

学习完了过滤器Filter之后，接下来我们继续学习拦截器Interceptor。

拦截器我们主要分为三个方面进行讲解：

1. 介绍下什么是拦截器，并通过快速入门程序上手拦截器
2. 拦截器的使用细节
3. 通过拦截器Interceptor完成登录校验功能

我们先学习第一块内容：拦截器快速入门

### 2.5.1 快速入门

什么是拦截器？

- 是一种动态拦截方法调用的机制，类似于过滤器。
- 拦截器是Spring框架中提供的，用来动态拦截控制器方法的执行。

拦截器的作用：

- 拦截请求，在指定方法调用前后，根据业务需要执行预先设定的代码。

在拦截器当中，我们通常也是做一些通用性的操作，比如：我们可以通过拦截器来拦截前端发起的请求，将登录校验的逻辑全部编写在拦截器当中。在校验的过程当中，如发现用户登录了(携带JWT令牌且是合法令牌)，就可以直接放行，去访问spring当中的资源。如果校验时发现并没有登录或是非法令牌，就可以直接给前端响应未登录的错误信息。

下面我们通过快速入门程序，来学习下拦截器的基本使用。拦截器的使用步骤和过滤器类似，也分为两步：

1. 定义拦截器
2. 注册配置拦截器

**自定义拦截器：**实现HandlerInterceptor接口，并重写其所有方法

```
1  //自定义拦截器
2  @Component
3  public class LoginCheckInterceptor implements HandlerInterceptor {
4      //目标资源方法执行前执行。 返回true: 放行    返回false: 不放行
5      @Override
6      public boolean preHandle(HttpServletRequest request,
7      HttpServletResponse response, Object handler) throws Exception {
8          System.out.println("preHandle .... ");
9          return true; //true表示放行
10     }
11
12     //目标资源方法执行后执行
13     @Override
14     public void postHandle(HttpServletRequest request,
15     HttpServletResponse response, Object handler, ModelAndView
16     modelAndView) throws Exception {
17         System.out.println("postHandle ... ");
18     }
19
20     //视图渲染完毕后执行，最后执行
21     @Override
22     public void afterCompletion(HttpServletRequest request,
23     HttpServletResponse response, Object handler, Exception ex) throws
24     Exception {
25         System.out.println("afterCompletion .... ");
26     }
27 }
```

注意：

preHandle方法：目标资源方法执行前执行。 返回true：放行 返回false：不放行

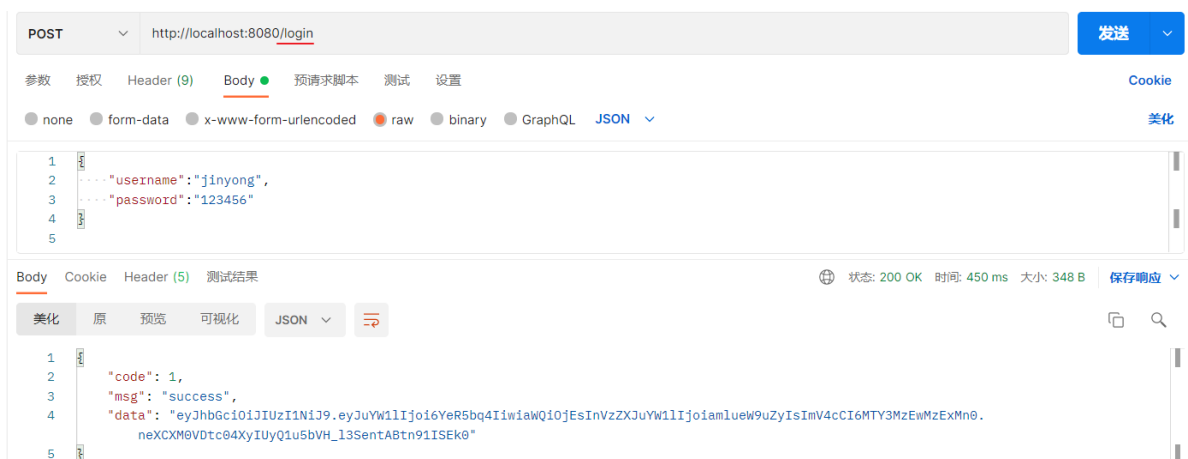
postHandle方法：目标资源方法执行后执行

afterCompletion方法：视图渲染完毕后执行，最后执行

**注册配置拦截器：**实现WebMvcConfigurer接口，并重写addInterceptors方法

```
1  @Configuration
2  public class WebConfig implements WebMvcConfigurer {
3
4      //自定义的拦截器对象
5      @Autowired
6      private LoginCheckInterceptor loginCheckInterceptor;
7
8
9      @Override
10     public void addInterceptors(InterceptorRegistry registry) {
11         //注册自定义拦截器对象
12
13         registry.addInterceptor(loginCheckInterceptor).addPathPatterns("/**
14         "); //设置拦截器拦截的请求路径（ /** 表示拦截所有请求）
15     }
16 }
```

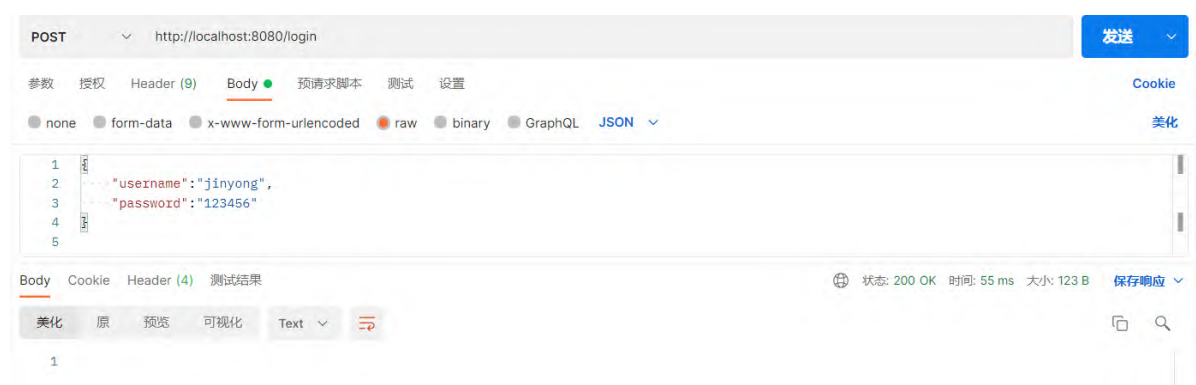
重新启动SpringBoot服务，打开postman测试：



```
TliasWebManagementApplication x
Console Endpoints
preHandle ....
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@65c184ed] was not registered for synchronizat
JDBC Connection [HikariProxyConnection@766471240 wrapping com.mysql.cj.jdbc.ConnectionImpl@6796de18] will not
==> Preparing: select id, username, password, name, gender, image, job, entrydate, dept_id, create_time, upda
==> Parameters: jinyong(String), 123456(String)
<== Columns: id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time
<== Row: 1, jinyong, 123456, 金庸, 1, 1.jpg, 4, 2000-01-01, 2, 2022-12-30 09:12:21, 2022-12-30 09:12:21
<== Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@65c184ed]
postHandle ...
afterCompletion ....
```

接下来我们再来做一个测试：将拦截器中返回值改为false

使用postman，再次点击send发送请求后，没有响应数据，说明请求被拦截了没有放行



## 2.5.2 Interceptor详解

拦截器的入门程序完成之后，接下来我们来介绍拦截器的使用细节。拦截器的使用细节我们主要介绍两个部分：

1. 拦截器的拦截路径配置
2. 拦截器的执行流程

### 2.5.2.1 拦截路径

首先我们先来看拦截器的拦截路径的配置，在注册配置拦截器的时候，我们要指定拦截器的拦截路径，通过 `addPathPatterns("要拦截路径")` 方法，就可以指定要拦截哪些资源。

在入门程序中我们配置的是 `/**`，表示拦截所有资源，而在配置拦截器时，不仅可以指定要拦截哪些资源，还可以指定不拦截哪些资源，只需要调用 `excludePathPatterns("不拦截路径")` 方法，指定哪些资源不需要拦截。



```

1  @Configuration
2  public class WebConfig implements WebMvcConfigurer {
3
4      //拦截器对象
5      @Autowired
6      private LoginCheckInterceptor loginCheckInterceptor;
7
8      @Override
9      public void addInterceptors(InterceptorRegistry registry) {
10         //注册自定义拦截器对象
11         registry.addInterceptor(loginCheckInterceptor)
12             .addPathPatterns("/**")//设置拦截器拦截的请求路径（ /**
            表示拦截所有请求）
13             .excludePathPatterns("/login");//设置不拦截的请求路径
14     }
15 }

```

在拦截器中除了可以设置 `/**` 拦截所有资源外，还有一些常见拦截路径设置：

拦截路径	含义	举例
<code>/*</code>	一级路径	能匹配/depts, /emps, /login, 不能匹配/depts/1
<code>/**</code>	任意级路径	能匹配/depts, /depts/1, /depts/1/2
<code>/depts/*</code>	/depts下的一级路径	能匹配/depts/1, 不能匹配/depts/1/2, /depts
<code>/depts/**</code>	/depts下的任意级路径	能匹配/depts, /depts/1, /depts/1/2, 不能匹配/emps/1

下面主要来演示下 `/**` 与 `/*` 的区别：

- 修改拦截器配置，把拦截路径设置为 `/*`

```

1  @Configuration
2  public class WebConfig implements WebMvcConfigurer {
3
4      //拦截器对象
5      @Autowired
6      private LoginCheckInterceptor loginCheckInterceptor;

```

```

7
8     @Override
9     public void addInterceptors(InterceptorRegistry registry) {
10         //注册自定义拦截器对象
11         registry.addInterceptor(loginCheckInterceptor)
12             .addPathPatterns("/*")
13             .excludePathPatterns("/login");//设置不拦截的请求路径
14     }
15 }

```

使用postman测试: <http://localhost:8080/emp/1>

The screenshot shows a Postman interface with a GET request to `http://localhost:8080/emp/1`. The response is a JSON object with the following structure:

```

{
  "code": 1,
  "msg": "success",
  "data": {
    "id": 1,
    "username": "jinyong",
    "password": "123456",
    "name": "金庸",
    "gender": 1,
    "image": "1.jpg",
    "job": 4,
    "entrydate": "2000-01-01",
    "deptId": 2,
    "createTime": "2022-12-30T09:12:21",
    "updateTime": "2022-12-30T09:12:21"
  }
}

```

控制台没有输出拦截器中的日志信息,说明 `/*` 没有匹配到拦截路径 `/emp/1` 。

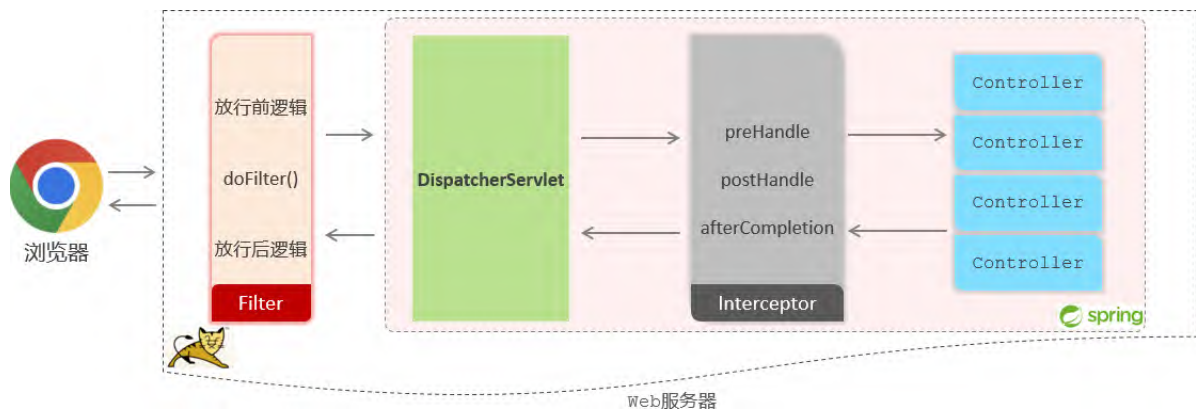
```

SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@76e008f9] was not registered for synchronization because synchronization i
 JDBC Connection [HikariProxyConnection@667772951 wrapping com.mysql.cj.jdbc.ConnectionImpl@5160b3f7] will not be managed by Spring
==> Preparing: select id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time from emp where id = ?
==> Parameters: 1(Integer)
<==      Columns: id, username, password, name, gender, image, job, entrydate, dept_id, create_time, update_time
<==      Row: 1, jinyong, 123456, 金庸, 1, 1.jpg, 4, 2000-01-01, 2, 2022-12-30 09:12:21, 2022-12-30 09:12:21
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@76e008f9]

```

## 2.5.2.2 执行流程

介绍完拦截路径的配置之后, 接下来我们再来介绍拦截器的执行流程。通过执行流程, 大家就能够清晰的知道过滤器与拦截器的执行时机。



- 当我们打开浏览器来访问部署在web服务器当中的web应用时，此时我们所定义的过滤器会拦截到这次请求。拦截到这次请求之后，它会先执行放行前的逻辑，然后再执行放行操作。而由于我们当前是基于springboot开发的，所以放行之后是进入到了spring的环境当中，也就是要来访问我们所定义的controller当中的接口方法。
- Tomcat并不识别所编写的Controller程序，但是它识别Servlet程序，所以在Spring的Web环境中提供了一个非常核心的Servlet：DispatcherServlet（前端控制器），所有请求都会先进行到DispatcherServlet，再将请求转给Controller。
- 当我们定义了拦截器后，会在执行Controller的方法之前，请求被拦截器拦截住。执行 `preHandle()` 方法，这个方法执行完成后需要返回一个布尔类型的值，如果返回true，就表示放行本次操作，才会继续访问controller中的方法；如果返回false，则不会放行（controller中的方法也不会执行）。
- 在controller当中的方法执行完毕之后，再回过头来执行 `postHandle()` 这个方法以及 `afterCompletion()` 方法，然后再返回给DispatcherServlet，最终再来执行过滤器当中放行后的这一部分逻辑的逻辑。执行完毕之后，最终给浏览器响应数据。

接下来我们就来演示下过滤器和拦截器同时存在的执行流程：

- 开启LoginCheckInterceptor拦截器

```

1  @Component
2  public class LoginCheckInterceptor implements HandlerInterceptor {
3      @Override
4      public boolean preHandle(HttpServletRequest request,
5                               HttpServletResponse response, Object handler) throws Exception {
6          System.out.println("preHandle .... ");
7          return true; //true表示放行
8      }
9
10     @Override

```

```

11     public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler, ModelAndView
    modelAndView) throws Exception {
12         System.out.println("postHandle ... ");
13     }
14
15     @Override
16     public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex) throws
    Exception {
17         System.out.println("afterCompletion .... ");
18     }
19 }

```

```

1  @Configuration
2  public class WebConfig implements WebMvcConfigurer {
3
4      //拦截器对象
5      @Autowired
6      private LoginCheckInterceptor loginCheckInterceptor;
7
8      @Override
9      public void addInterceptors(InterceptorRegistry registry) {
10         //注册自定义拦截器对象
11         registry.addInterceptor(loginCheckInterceptor)
12             .addPathPatterns("/**") //拦截所有请求
13             .excludePathPatterns("/login");//不拦截登录请求
14     }
15 }

```

- 开启DemoFilter过滤器

```

1  @WebFilter(urlPatterns = "/*")
2  public class DemoFilter implements Filter {
3      @Override
4      public void doFilter(ServletRequest servletRequest,
5                          ServletResponse servletResponse, FilterChain filterChain) throws
6                          IOException, ServletException {
7          System.out.println("DemoFilter    放行前逻辑.....");
8
9          //放行请求
10         filterChain.doFilter(servletRequest,servletResponse);
11
12         System.out.println("DemoFilter    放行后逻辑.....");
13     }
14 }

```

重启SpringBoot服务后，清空日志，打开Postman，测试查询部门：

The screenshot displays two windows from a development environment. The top window is the Postman interface, showing a GET request to `http://localhost:8080/depts` with a status of 200 OK. The response body is a JSON object:

```

{
  "code": 1,
  "msg": "success",
  "data": [
    {
      "id": 3,
      "name": "咨询部",
      "createTime": "2022-12-30T09:12:11",
      "updateTime": "2022-12-30T09:12:11"
    }
  ]
}

```

The bottom window is the Spring Boot IDE's console, showing the execution of the `DemoFilter`. The logs indicate that the filter's `doFilter` method is called, and the request is successfully processed. The console output includes the following messages:

```

DemoFilter    放行前逻辑.....
preHandle ....
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@59b81e8c] was not registered for synch
JDBC Connection [HikariProxyConnection@1886794728 wrapping com.mysql.cj.jdbc.ConnectionImpl@714a4677] w
==> Preparing: select id, name, create_time, update_time from dept
==> Parameters:
Columns: id, name, create_time, update_time
Row: 3, 咨询部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
Row: 4, 就业部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
Row: 5, 人事部, 2022-12-30 09:12:11, 2022-12-30 09:12:11
Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@59b81e8c]
postHandle ...
afterCompletion ....
DemoFilter    放行后逻辑.....

```

以上就是拦截器的执行流程。通过执行流程分析，大家应该已经清楚了过滤器和拦截器之间的区别，其实它们之间的区别主要是两点：

- 接口规范不同：过滤器需要实现Filter接口，而拦截器需要实现HandlerInterceptor接口。
- 拦截范围不同：过滤器Filter会拦截所有的资源，而Interceptor只会拦截Spring环境中的资源。

### 2.5.3 登录校验- Interceptor

讲解完了拦截器的基本操作之后，接下来我们需要完成最后一步操作：通过拦截器来完成案例当中的登录校验功能。

登录校验的业务逻辑以及操作步骤我们前面已经分析过了，和登录校验Filter过滤器当中的逻辑是完全一致的。现在我们只需要把这个技术方案由原来的过滤器换成拦截器interceptor就可以了。

#### 登录校验拦截器

```
1  //自定义拦截器
2  @Component //当前拦截器对象由Spring创建和管理
3  @Slf4j
4  public class LoginCheckInterceptor implements HandlerInterceptor {
5      //前置方式
6      @Override
7      public boolean preHandle(HttpServletRequest request,
8          HttpServletResponse response, Object handler) throws Exception {
9          System.out.println("preHandle .... ");
10         //1.获取请求url
11         //2.判断请求url中是否包含login，如果包含，说明是登录操作，放行
12         //3.获取请求头中的令牌（token）
13         String token = request.getHeader("token");
14         log.info("从请求头中获取的令牌：{}",token);
15
16         //4.判断令牌是否存在，如果不存在，返回错误结果（未登录）
17         if(!StringUtils.hasLength(token)){
18             log.info("Token不存在");
19
20             //创建响应结果对象
21             Result responseResult = Result.error("NOT_LOGIN");
```

```

22         //把Result对象转换为JSON格式字符串（fastjson是阿里巴巴提供的
    用于实现对象和json的转换工具类）
23         String json = JSONObject.toJSONString(responseResult);
24         //设置响应头（告知浏览器：响应的数据类型为json、响应的数据编码
    表为utf-8）
25         response.setContentType("application/json;charset=utf-
    8");
26         //响应
27         response.getWriter().write(json);
28
29         return false;//不放行
30     }
31
32     //5.解析token，如果解析失败，返回错误结果（未登录）
33     try {
34         JwtUtils.parseJWT(token);
35     } catch (Exception e) {
36         log.info("令牌解析失败!");
37
38         //创建响应结果对象
39         Result responseResult = Result.error("NOT_LOGIN");
40         //把Result对象转换为JSON格式字符串（fastjson是阿里巴巴提供的
    用于实现对象和json的转换工具类）
41         String json = JSONObject.toJSONString(responseResult);
42         //设置响应头
43         response.setContentType("application/json;charset=utf-
    8");
44         //响应
45         response.getWriter().write(json);
46
47         return false;
48     }
49
50     //6.放行
51     return true;
52 }

```

## 注册配置拦截器

```

1  @Configuration
2  public class WebConfig implements WebMvcConfigurer {
3      //拦截器对象
4      @Autowired
5      private LoginCheckInterceptor loginCheckInterceptor;
6

```



```

7      @Override
8      public void addInterceptors(InterceptorRegistry registry) {
9          //注册自定义拦截器对象
10         registry.addInterceptor(loginCheckInterceptor)
11             .addPathPatterns("/**")
12             .excludePathPatterns("/login");
13     }
14 }
15

```

登录校验的拦截器编写完成后，接下来我们就可以重新启动服务来做一个测试：（关闭登录校验Filter过滤器）

- 测试1：未登录是否可以访问部门管理页面

首先关闭浏览器，重新打开浏览器，在地址栏中输入：<http://localhost:9528/#/system/dept>

由于用户没有登录，校验机制返回错误信息，前端页面根据返回的错误信息结果，自动跳转到登录页面了



- 测试2：先进行登录操作，再访问部门管理页面

登录校验成功之后，可以正常访问相关业务操作页面



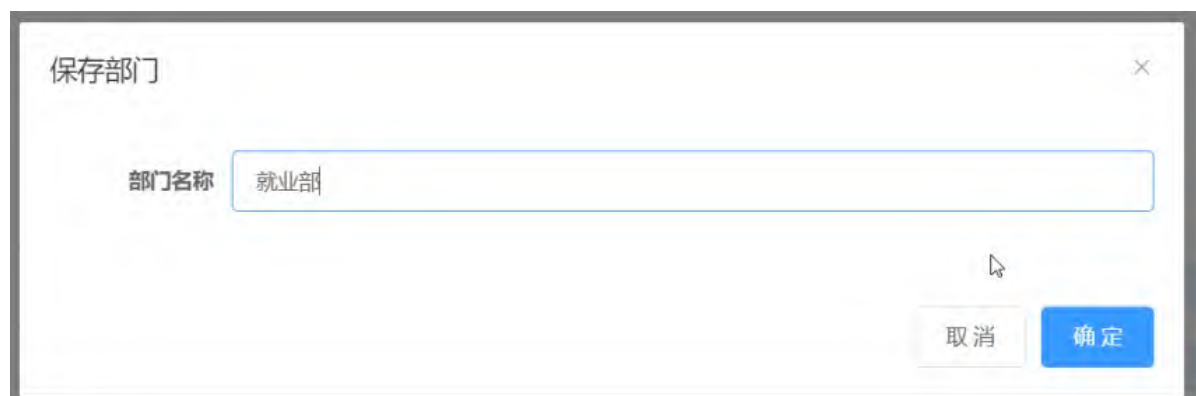
到此我们也就验证了所开发的登录校验的拦截器也是没问题的。登录校验的过滤器和拦截器，我们只需要使用其中的一种就可以了。

## 3. 异常处理

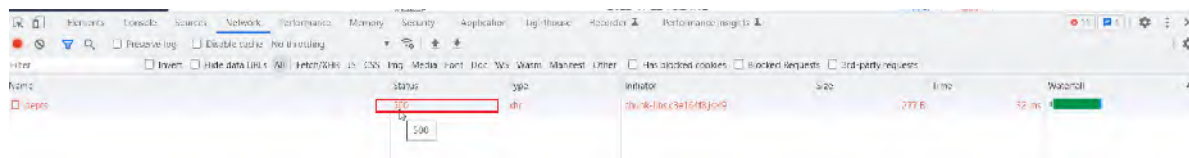
### 3.1 当前问题

登录功能和登录校验功能我们都实现了，下面我们学习下今天最后一块技术点：异常处理。首先我们先来看一下系统出现异常之后会发生什么现象，再来介绍异常处理的方案。

我们打开浏览器，访问系统中的新增部门操作，系统中已经有了“就业部”这个部门，我们再来增加一个就业部，看看会发生什么现象。



点击确定之后，窗口关闭了，页面没有任何反应，就业部也没有添加上。而此时，大家会发现，网络请求报错了。



状态码为500，表示服务器端异常，我们打开idea，来看一下，服务器端出了什么问题。

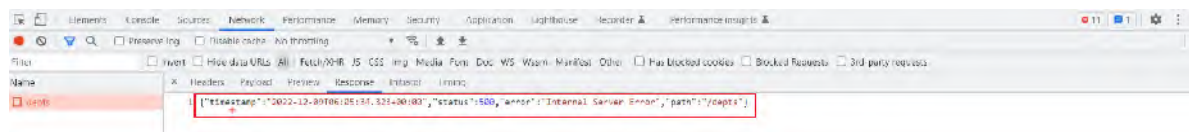
```
### SQL: insert into dept(name, create time, update time) values(?,?,?)
### Cause: java.sql.SQLException: Duplicate entry '就业部' for key 'dept.name'
; Duplicate entry '就业部' for key 'dept.name'; nested exception is java.sql.SQLException: Duplicate entry '就业部' for key 'dept.name'

java.sql.SQLException: Create breakpoint: Duplicate entry '就业部' for key 'dept.name'
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122) ~[mysql-connector-j-8.0.31.jar:8.0.31]
    at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:122) ~[mysql-connector-j-8.0.31.jar:8.0.31]
    at com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPreparedStatement.java:916) ~[mysql-connector-j-8.0.31.jar:8.0.31]
    at com.mysql.cj.jdbc.ClientPreparedStatement.execute(ClientPreparedStatement.java:558) ~[mysql-connector-j-8.0.31.jar:8.0.31]
    at com.zaxxer.hikari.pool.ProxyPreparedStatement.execute(ProxyPreparedStatement.java:44) ~[HikariCP-4.0.3.jar:na]
```

上述错误信息的含义是，dept部门表的name字段的值 就业部 重复了，因为在数据库表dept中已经有了就业部，我们之前设计这张表时，为name字段建议了唯一约束，所以该字段的值是不能重复的。

而当我们再添加就业部，这个部门时，就违反了唯一约束，此时就会报错。

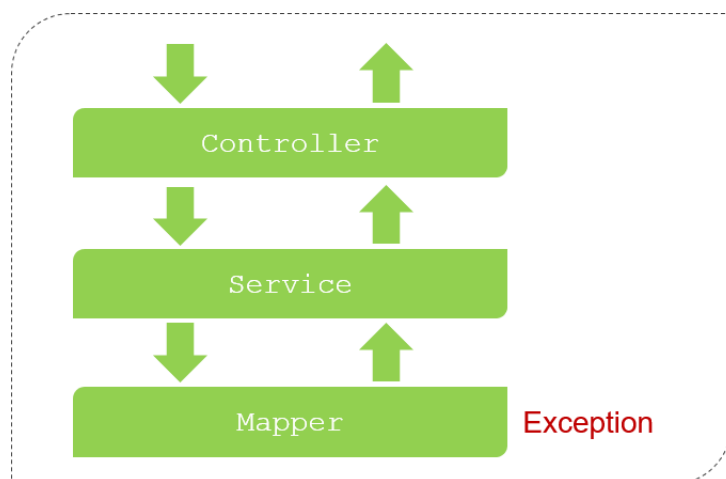
我们来看一下出现异常之后，最终服务端给前端响应回来的数据长什么样。



响应回来的数据是一个JSON格式的数据。但这种JSON格式的数据还是我们开发规范当中所提到的统一响应结果Result吗？显然并不是。由于返回的数据不符合开发规范，所以前端并不能解析出响应的JSON数据。

接下来我们需要思考的是出现异常之后，当前案例项目的异常是怎么处理的？

- 答案：没有做任何的异常处理



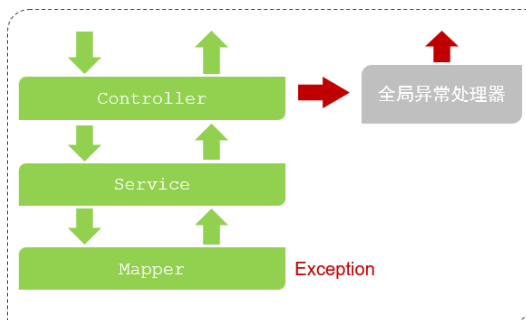
当我们没有做任何的异常处理时，我们三层架构处理异常的方案：

- Mapper接口在操作数据库的时候出错了，此时异常会往上抛（谁调用Mapper就抛给谁），会抛给service。
- service 中也存在异常了，会抛给controller。
- 而在controller当中，我们也没有做任何异常处理，所以最终异常会再往上抛。最终抛给框架之后，框架就会返回一个JSON格式的数据，里面封装的就是错误的信息，但是框架返回的JSON格式的数据并不符合我们的开发规范。

## 3.2 解决方案

那么在三层构架项目中，出现了异常，该如何处理？

- 方案一：在所有Controller的所有方法中进行try...catch处理
  - 缺点：代码臃肿（不推荐）
- 方案二：全局异常处理器
  - 好处：简单、优雅（推荐）



## 3.3 全局异常处理器

我们该怎么样定义全局异常处理器？

- 定义全局异常处理器非常简单，就是定义一个类，在类上加上一个注解  
`@RestControllerAdvice`，加上这个注解就代表我们定义了一个全局异常处理器。
- 在全局异常处理器当中，需要定义一个方法来捕获异常，在这个方法上需要加上注解  
`@ExceptionHandler`。通过`@ExceptionHandler`注解当中的`value`属性来指定我们要捕获的是哪一类型的异常。

```

1  @RestControllerAdvice
2  public class GlobalExceptionHandler {
3
4      //处理异常
5      @ExceptionHandler(Exception.class) //指定能够处理的异常类型
6      public Result ex(Exception e){
7          e.printStackTrace(); //打印堆栈中的异常信息
8
9          //捕获到异常之后，响应一个标准的Result
10         return Result.error("对不起,操作失败,请联系管理员");
11     }
12 }

```

@RestControllerAdvice = @ControllerAdvice + @ResponseBody

处理异常的方法返回值会转换为json后再响应给前端

重新启动SpringBoot服务，打开浏览器，再来测试一下添加部门这个操作，我们依然添加已存在的"就业部" 这个部门：



此时，我们可以看到，出现异常之后，异常已经被全局异常处理器捕获了。然后返回的错误信息，被前端程序正常解析，然后提示出了对应的错误提示信息。

以上就是全局异常处理器的使用，主要涉及到两个注解：

- @RestControllerAdvice //表示当前类为全局异常处理器

- `@ExceptionHandler` //指定可以捕获哪种类型的异常进行处理