15. Spring 统一功能处理

本篇文章我们来讲解如何使用 Spring 拦截器来处理项目中常用的统一功能处理：

* 统一用户登录权限验证
* 统一异常处理
* 统一数据格式返回

这时有人可能反问到，不是可以用Spring AOP来对这些功能进行处理吗？那我还要去学拦截器干嘛？

还是一句话：“存在即合理”，Spring拦截器的存在能弥补一些Spring AOP的不足，且听我细细道来。

1. 用户登录权限验证

这里我将分别使用三种方式实现这个登录权限验证的功能，从而引出为什么要使用 Spring 拦截器。

1.1 最原始的登录验证

在学习Spring AOP之前，也就是Servlet的时代我们会在所有需要登录验证的路由方法下添加登录验证的业务：

```
HttpSession session = req.getSession(false);
if (session == null) {
    resp.setContentType(”text/html;charset=utf-8”);
    resp.getWriter().write(”用户未登陆，请登录！！”);
    return;
}
User user = (User) session.getAttribute(”user”);
```

```
if (user == null) {
    resp.setContentType("text/html;charset=utf-8");
    resp.getWriter().write("用户未登陆，请登录！！");
    return;
}
```

1.2 Spring AOP实现

学习了Spring AOP后，我们就想着使用切面来消除冗余代码。

编写`UserController`类，类中目前有两个方法`login()`和`getall()`

```
@RestController
@RequestMapping("/user")
public class UserController {

    @RequestMapping("/login")
    public String login(HttpServletRequest request, String username) {
        //验证账密的伪代码
        System.out.println("进行账号密码验证");
        //存储session
        HttpSession session = request.getSession(true);
        session.setAttribute("user", username);
        return "登录成功";
    }

    @RequestMapping("/getall")
    public String getAll() {
        return "查询用户信息";
    }
}
```

编写`Aspect`类：

```
@Component
```

```
@Aspect
public class LoginAspect {
    @Pointcut("execution(*
com.chenshu.intercept_demo.controller..*.*(..))")
    private void myPointcut(){};

    @Around("myPointcut()")
    public String loginValidate(ProceedingJoinPoint joinPoint) throws
Throwable {
        HttpServletRequest request =
((ServletRequestAttributes)RequestContextHolder.getRequestAttributes())
.getRequest();
        HttpSession session = request.getSession(false);
        if (session == null) {
            return "用户未登陆，请登录！！";
        }
        String username = (String) session.getAttribute("user");
        if (username == null) {
            return "用户未登陆，请登录！！";
        }
        return (String) joinPoint.proceed();
    }
}

```

第一个弊端：获取Session的过程中我们发现，需要使用九牛二虎之力才能成功在切面中取到`HttpServletRequest`对象。

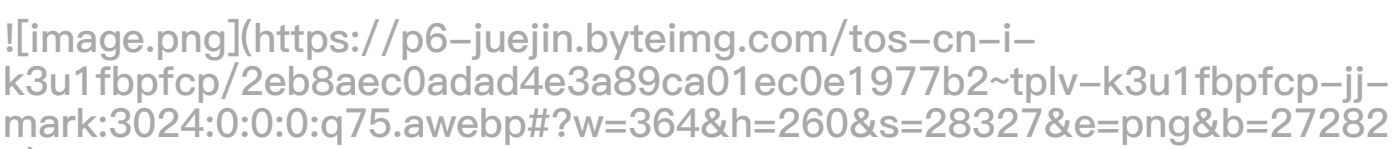
首先通过`RequestContextHolder.getRequestAttributes()`方法获取到当前线程的请求上下文，然后通过强制类型转换为`ServletRequestAttributes`类型，再通过`getRequest()`方法获取到`HttpServletRequest`对象。

```
HttpServletRequest request =
((ServletRequestAttributes)RequestContextHolder.getRequestAttributes())
.getRequest();

```

第二个弊端：一般情况下`Usercontroller`类中会存放所有用户相关的业务，有些业务可能不需要身份权限验证，比如登录、注册业务，下面的登录业务中就遇到了这种状况：

这种状况有两种解决方案，一种是**编写更复杂的切点表达式**，一种是**将不需要登录权限验证的业务另外分包**。

这里我就使用分包来解决上面的问题：



`validate`包下的`UserController1`类：

```
@RestController
@RequestMapping("/user1")
public class UserController1 {

    @RequestMapping("/getall")
    public String getAll() {
        return "查询用户信息";
    }
}
```

`novalidate`包下的`UserController2`类：

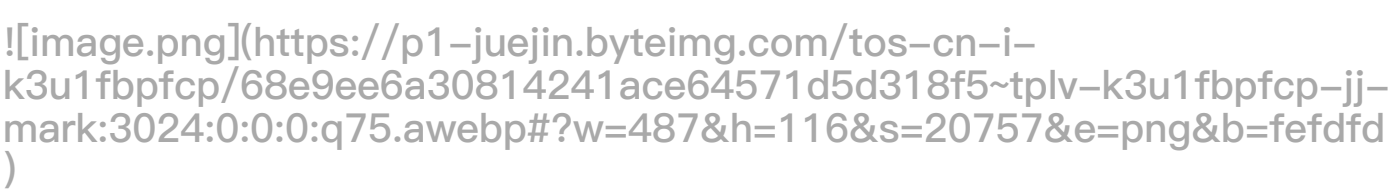
```
@RestController
@RequestMapping("user2")
```

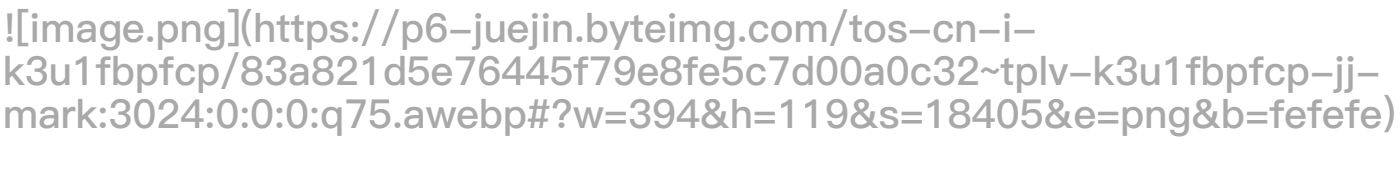
```
public class UserController2 {
    @RequestMapping(”/login”)
    public String login(HttpServletRequest request, String username) {
        //伪代码
        System.out.println(”进行账号密码验证”);
        //存储session
        HttpSession session = request.getSession(true);
        session.setAttribute(”user”, username);
        return ”登录成功”;
    }

}
```

修改切点表达式：只对`validate`包内的所有类进行身份权限验证

```
@Pointcut(”execution(*
com.chenshu.intercept_demo.controller.validate.*.*(..))”)
private void myPointcut(){};
```

此时问题就得以解决了：用户成功在登录后查询到用户信息





一顿操作下来问题虽然解决了，但是想必大家写完不禁感慨道：这也太麻烦了吧！

别担心，你遇到的问题前人一定早就遇到了，官方给出了解决方案 —— Spring 拦截器

1.3 Spring 拦截器

对于以上问题，Spring中提供了具体的实现拦截器：`HandlerInterceptor`，对比与`Spring AOP`，拦截器在Spring Web项目中可以更方便使用请求和响应的包装类，并且可以更方便地配置拦截规则，的如果用户访问的路由中有拦截器的话，就会先走拦截器，拦截器会决定是否放行给`Controller`：

拦截器的实现分为以下两个步骤：

1. 创建自定义拦截器：实现HandlerInterceptor接口并重写preHandle（执行具体方法之前的预处理）方法。
2. 配置拦截器并添加拦截规则：将自定义拦截器加入WebMvcConfigurer的addInterceptors方法中。

接下来我将对拦截器的实现进行代码演示：

1. 先把刚刚在controller包中拆分的UserController还原回来，并稍作修改：

```
@RestController
@RequestMapping(”/user”)
public class UserController {

    @RequestMapping(”/login”)
    public String login(HttpServletRequest request, String username, String password) {
        //1. 非空判断，这里直接使用Spring提供的工具方法来实现
        if (StringUtils.hasLength(username) && StringUtils.hasLength(password)) {
            //2. 验证密码是否正确
            if (”admin”.equals(username) && ”admin”.equals(password)) {
                HttpSession session = request.getSession(true);
                session.setAttribute(”user”, ”admin”);
```

```
            return ”登录成功！”;
        } else {
            return ”用户名或密码输入错误，登录失败！”;
        }
    }
    return ”用户名或密码为空，请重新输入！”;
}

@RequestMapping(”/reg”)
public String reg() {
    return ”用户进行注册操作！”;
}

@RequestMapping(”/getall”)
public String getAll() {
    return ”查询用户信息成功！”;
}
}
```

1.3.1 创建自定义拦截器

1. 新建一个interceptors包
2. 在interceptors包中创建`LoginInterceptor`类（自行命名），并实现`HandlerInterceptor`接口（成为拦截器）
3. 重写`preHandle()`，在其中编写用户登录权限验证的功能
4. 如果`preHandle()`方法返回true就继续执行后续业务，返回false则相反

```
public class LoginInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        HttpSession session = request.getSession(false);
        if (session != null && session.getAttribute(”user”) != null) {
            return true;
        }
        response.setStatus(401);
        return false;
    }
}
```

1.3.2 配置拦截器并添加拦截规则：

1. 创建一个config包
2. 在config包中创建一个`InterceptorConfig`类（自行命名），并实现`WebMvcConfigurer`接口（成为Mvc配置类）
3. 添加五大类注解，使其在项目启动时就生效，这里使用`@Configuration`
4. 重写`addInterceptors()`，在其中使用`registry`对象的`addInterceptor`的方法将自定义拦截器加入到系统配置信息中
5. 添加拦截规则：通过`addInterceptor()`返回的对象调用`addPathPatterns()`拦截规则
6. 排除拦截规则：通过`addPathPatterns()`返回的对象调用`excludePathPatterns()`排除拦截规则

```
@Configuration
public class MyMvcConfigurer implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor())
                .addPathPatterns("/**") // 拦截所有请求
                .excludePathPatterns("/user/login") //排除不拦截的路径
                .excludePathPatterns("/user/reg"); // 排除不拦截的路径
    }
}
```

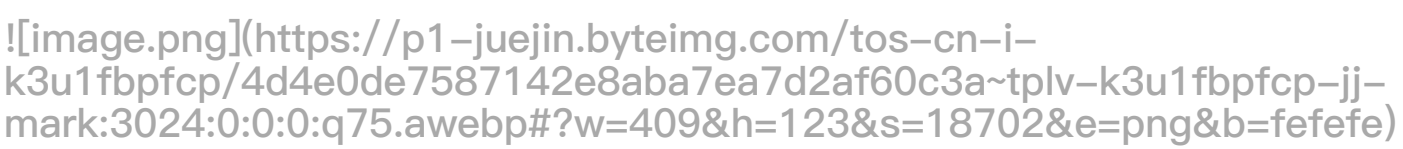
1.3.3 【引入】拦截规则的通配符

1. `/*`：匹配单个路径级别，如`/admin`、`/user`、`/product`
2. `/**`：匹配多个路径级别，包括子路径，如`/admin`、`/admin/user`、`/admin/user/edit`
3. `/* + 文件后缀名`：某路径下所有某后缀名的文件，如`/*.html`

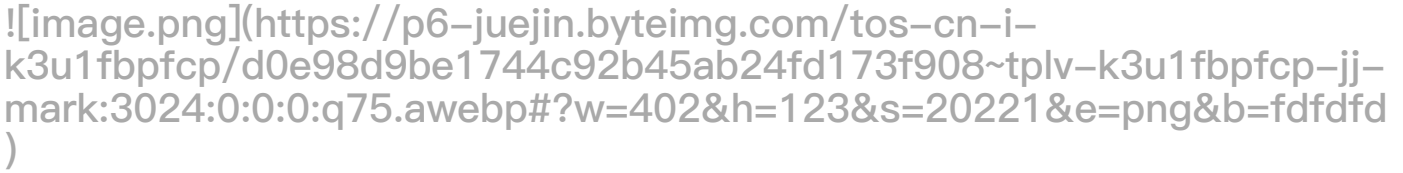
运行并测试：

成功拦截user/getall，并将状态码设置为401：

不拦截user/reg，执行到UserController业务代码：



不拦截user/login，执行到UserController业务代码：



1.3.4 拦截器源码分析

项目启动后服务器接收到第一个HTTP请求时，可以在console中查看到初始化`DispatcherServlet`(前端控制器)的日志信息：

```
2024-04-22 00:41:58.898  INFO 45311 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-04-22 00:41:58.898  INFO 45311 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
```

`DispatcherServlet`的解释：当客户端发送一个HTTP请求到服务器时，请求首先到达 Servlet 容器。然后会到达`DispatcherServlet`(前端控制器)，它是一个Servlet，负责拦截所有的请求并将请求分发给相应的`Controller`。

而所有请求到达`DispatcherServlet`的时候，会调用一个`doDispatch()`方法，它的源码如下（重点看标注了注解的地方）：

```
```

```
protected void doDispatch(HttpServletRequest request,
HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;
    WebAsyncManager asyncManager =
WebAsyncUtils.getAsyncManager(request);

    try {
        try {
            ModelAndView mv = null;
            Exception dispatchException = null;

            try {
                processedRequest = this.checkMultipart(request);
                multipartRequestParsed = processedRequest != request;
                mappedHandler = this.getHandler(processedRequest);
                if (mappedHandler == null) {
                    this.noHandlerFound(processedRequest, response);
                    return;
                }
                //获取HandlerAdapter执行处理器，这是用来执行Controller中的
业务的
                HandlerAdapter ha =
this.getHandlerAdapter(mappedHandler.getHandler());
                String method = request.getMethod();
                boolean isGet = HttpMethod.GET.matches(method);
                if (isGet || HttpMethod.HEAD.matches(method)) {
                    long lastModified = ha.getLastModified(request,
mappedHandler.getHandler());
                    if ((new ServletWebRequest(request,
response)).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }
                //调用预处理
                if (!mappedHandler.applyPreHandle(processedRequest,
response)) {
                    return;
                }
                //通过HandlerAdapter执行Controller中的业务
                mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
                if (asyncManager.isConcurrentHandlingStarted()) {
                    return;
                }
                //调用后处理
                this.applyDefaultViewName(processedRequest, mv);
```

```
                mappedHandler.applyPostHandle(processedRequest,
response, mv);
            } catch (Exception var20) {
                dispatchException = var20;
            } catch (Throwable var21) {
                dispatchException = new NestedServletException("Handler
dispatch failed", var21);
            }

            this.processDispatchResult(processedRequest, response,
mappedHandler, mv, (Exception)dispatchException);
        } catch (Exception var22) {
            this.triggerAfterCompletion(processedRequest, response,
mappedHandler, var22);
        } catch (Throwable var23) {
            this.triggerAfterCompletion(processedRequest, response,
mappedHandler, new NestedServletException("Handler processing
failed", var23));
        }

    } finally {
        if (asyncManager.isConcurrentHandlingStarted()) {
            if (mappedHandler != null) {
mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest
, response);
            }
        } else if (multipartRequestParsed) {
            this.cleanupMultipart(processedRequest);
        }

    }
}
```

提取重点内容：

```
//调用预处理
if (!mappedHandler.applyPreHandle(processedRequest, response)) {
    return;
}
//通过HandlerAdapter执行Controller中的业务
mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
if (asyncManager.isConcurrentHandlingStarted()) {
```

```
    return;
}
//调用后处理
this.applyDefaultViewName(processedRequest, mv);
mappedHandler.applyPostHandle(processedRequest, response, mv);
```

根据源码我们发现在预处理的时候，如果返回的值为false就直接就直接return，不继续走Controller层的业务；返回的值为true之后，才会通过`HandlerAdapter`执行Controller中的业务。

点进`applyPreHandle(processedRequest, response)`方法的源码中查看一下它做了什么事：

```
boolean applyPreHandle(HttpServletRequest request,
HttpServletResponse response) throws Exception {
    for(int i = 0; i < this.interceptorList.size(); this.interceptorIndex = i++) {
        HandlerInterceptor interceptor =
(HandlerInterceptor)this.interceptorList.get(i);
        if (!interceptor.preHandle(request, response, this.handler)) {
            this.triggerAfterCompletion(request, response, (Exception)null);
            return false;
        }
    }

    return true;
}
```

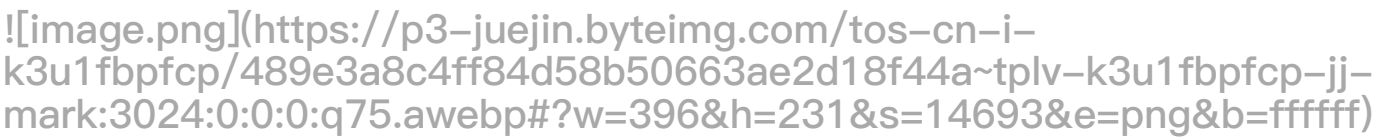
这个方法位于`HandlerExecutionChain`这个类中，拉到最上面，可以看到该类有一个属性`interceptorList`，也就是拦截器列表，我们在`WebMvcConfigurer`中添加的拦截器都在这里面：

```
private final List<HandlerInterceptor> interceptorList;
```

```

然后我们再回头看`applyPreHandle(processedRequest, response)`方法，就能知道它到底做了什么事了：无非就是遍历拦截器列表，并且调用它们的`preHandle()`方法，只要有一个拦截器的`preHandle()`方法返回false，那就都别玩，直接返回false给到`doDispatcher()`方法中，不继续执行`Controller`层的代码。

## 2. 统一异常处理

Spring 框架的统一异常处理是基于 Spring MVC 模块的特性来实现的。

### 2.1 统一异常处理是什么

统一异常处理是通过`@ControllerAdvice`+`@ExceptionHandler`这两个注解来实现的。

1. **@ControllerAdvice 注解**：`@ControllerAdvice` 注解是 Spring MVC 提供的一种特殊类型的控制器，它可以用于定义全局的控制器通知。被 `@ControllerAdvice` 注解标记的类中的方法可以拦截应用程序中抛出的异常。
2. **@ExceptionHandler 注解**：`@ExceptionHandler` 注解用于标记方法，指示该方法应该处理特定类型的异常。当被标记的方法抛出对应类型的异常时，Spring MVC 将调用该方法来处理异常，并返回适当的响应。

### 2.2 统一异常处理的使用

在讲解使用前，我先举个例子来描述一下为什么要使用统一异常处理。

在`/reg`路由中构造一个算数异常：

```
@RequestMapping(”/reg”)
public String reg() {
    int ret = 10/0;
```
```

```
    return ”用户进行注册操作！”;
}
```

当我们在浏览器中访问该路由时，发现状态码直接变成`401`，并显示该页面不存在，这样太过于暴力，对于程序猿来说倒还好，但是用户看到这么一串信息直接就懵逼了：



但是通过统一异常的处理，在后端程序抛异常的时候，`@ControllerAdvice`标注的类中的方法可以拦截应用程序中抛出的异常；该类中的方法又需要使用一个`@ExceptionHandler` 注解来对相应的异常进行处理，在方法中，后端程序猿可以返回自定义的错误响应（如错误码+错误信息）给前端，前端的程序猿就可以合理地根据响应，将合适的页面展现给用户。

扯了这么多，接下来我就来演示下如何实现统一异常的处理，并返回自定义的错误响应给前端：

1. 首先在`interceptors`包新建了一个`ExceptionAdvice`类，并在类上添加了一个`@ControllerAdvice`使该类中的方法可以拦截到异常，然后又在类上添加了一个`@ResponseBody`的注解表示返回非视图对象：

```
@ResponseBody
@ControllerAdvice
public class ExceptionAdvice {

}
```

---

2. 编写方法，并在方法上添加`@ExceptionHandler`注解标识拦截哪种异常：

```
@ControllerAdvice
public class ExceptionAdvice {
    @ExceptionHandler(Exception.class)
    public HashMap<String, Object> ExceptionHandler() {
        HashMap<String, Object> retMap = new HashMap<>();
        return retMap;
    }
}
```

由于我想返回一个自定义的Json对象给前端，因此上面的代码中我返回一个`Map`对象，由Spring MVC来帮我自动转换为Json对象并返回。

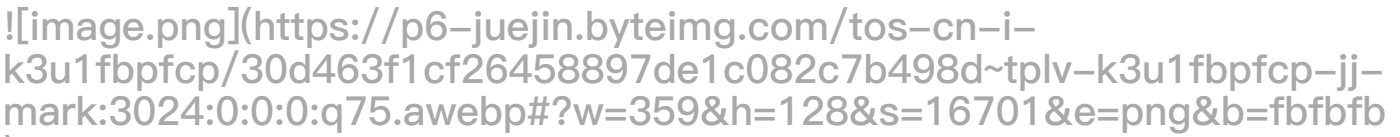
---

3. 此时我们需要往map对象中添加你想要返回的键值对，这里我想返回一个`-1错误码`以及`异常信息`给前端：

```
@ControllerAdvice
public class ExceptionAdvice {
    @ExceptionHandler(Exception.class)
    public HashMap<String, Object> ExceptionHandler(Exception e) {
        HashMap<String, Object> retMap = new HashMap<>();
        retMap.put(”code”, ”-1”);
        retMap.put(”msg”, e.getMessage());
        return retMap;
    }
}
```

由于我想要在返回的Json对象中添加异常信息的键值对，而`@ExceptionHandler`注解修饰的方法可以直接在参数中获取到异常对象，这里我直接通过异常对象获取异常信息并返回。

此时再通过浏览器访问后端，发现显示了下面信息：



2.3 粒度细化

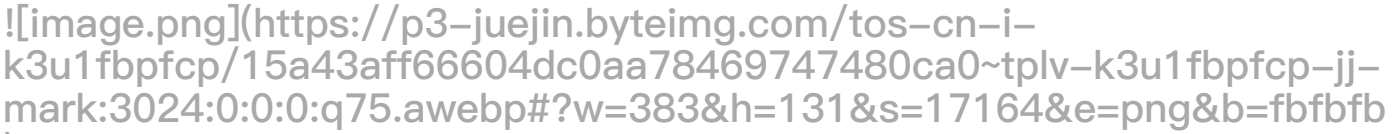
前面说过`@ExceptionHandler` 标记的方法会处理特定类型的异常。

因此可以再编写一个方法并在该注解的参数中传入一个 `ArithmeticException.class`，从而使拦截的异常粒度细化：

这里将错误码设为`-2`，表示算数异常的错误码：

```
@ExceptionHandler(ArithmeticException.class)
public HashMap<String, Object>
ArithmeticExceptionHandler(ArithmeticException e) {
    HashMap<String, Object> retMap = new HashMap<>();
    retMap.put(”code”, ”-2”);
    retMap.put(”msg”, e.getMessage());
    return retMap;
}
```

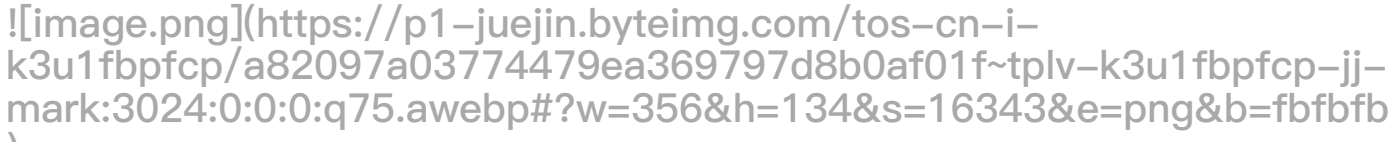
测试发现，这次算数异常的处理优先被刚刚编写的方法所拦截：



在构造一个空指针异常来测试：

```
@RequestMapping(”/reg”)
public String reg() {
    Object obj = null;
    obj.hashCode();
    return ”用户进行注册操作！”;
}
```

此时不是算数异常，就被最初写的`ExceptionHandler`方法拦截了



3. 统一数据返回格式
==========

为什么要统一数据返回格式呢？其实和异常处理一样，是为了降低和前端程序猿的沟通成本，后端程序猿将所有响应都分装成一个固定格式的Json对象，前端程序猿可以通过Json对象中的状态码来辨别是正常的响应还是异常的响应。

统一异常处理是通过`@ControllerAdvice`注解 + 实现`ResponseBodyAdvice`接口来实现的

1. `@ControllerAdvice`注解就不多说了，统一异常处理中已经提到
2. **编写 ResponseBodyAdvice 实现类**：创建一个实现`ResponseBodyAdvice` 接口的类，该类可以在响应返回给客户端之前进行处理。

3.1 实现统一数据返回格式处理

前面我们通过统一的异常处理，将异常的响应信息的状态码设为负数，这里我们就可以将正常响应信息的状态码设为正数，并且在Json对象中新增一个body的属性来填写响应的具体内容。

1. 在`interceptors`包中新增一个`ResponseAdvice`类，给该类添加

`@ControllerAdvice`注解并且该类需实现`ResponseBodyAdvice`接口

> 注意事项：实现`ResponseBodyAdvice`接口后默认返回非视图对象，因此这里不需要添加`@ResponseBody`注解

```
@ControllerAdvice
public class ResponseAdvice implements ResponseBodyAdvice {

}
```

2. 必须实现接口中的`support()`和`beforeBodyWrite()`方法

support()方法： 用于确定Controller层返回的类型是否需要后续的`beforeBodyWrite()`对其进行封装，如果返回true，就继续走`beforeBodyWrite()`方法，返回false，则直接返回Controller层原生的数据格式

beforeBodyWrite()方法： 在这里可以编写你具体要讲响应分装成什么形式的逻辑代码。里面有一个`body`参数，是用来获取到Controller中返回的对象的。

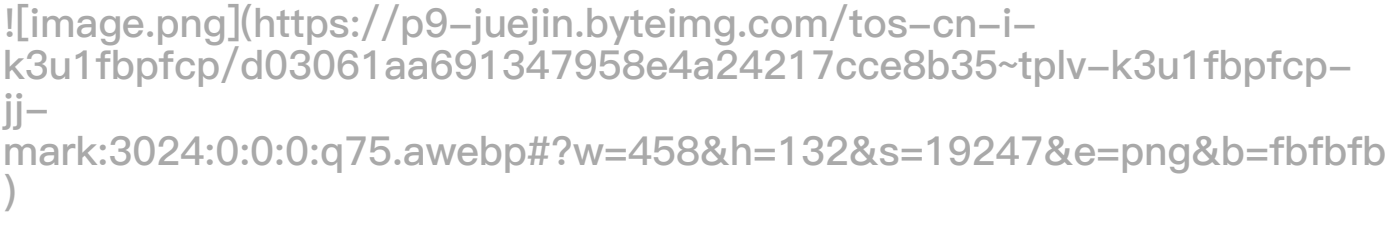
了解了两个方法是做什么的就可以开始编写代码了：

```
@ControllerAdvice
public class ResponseAdvice implements ResponseBodyAdvice {

    //对所有请求都进行封装
    @Override
    public boolean supports(MethodParameter returnType, Class
converterType) {
        return true;
    }

    //处理统一数据返回格式的逻辑
    @Override
```

```
    public Object beforeBodyWrite(Object body, MethodParameter
returnType, MediaType selectedContentType, Class
selectedConverterType, ServerHttpRequest request, ServerHttpResponse
response) {
        HashMap<String, Object> retMap = new HashMap<>();
        retMap.put(”code”, ”200”);
        retMap.put(”msg”, ””);
        retMap.put(”body”, body);
        return retMap;
    }
}
```

3.2 【引入】响应嵌套问题

当我们编写完代码，满心欢喜的进行测试`/reg`路由后，发现前面统一异常处理的错误响应信息被嵌套在统一数据返回格式的响应的`body`属性中了：

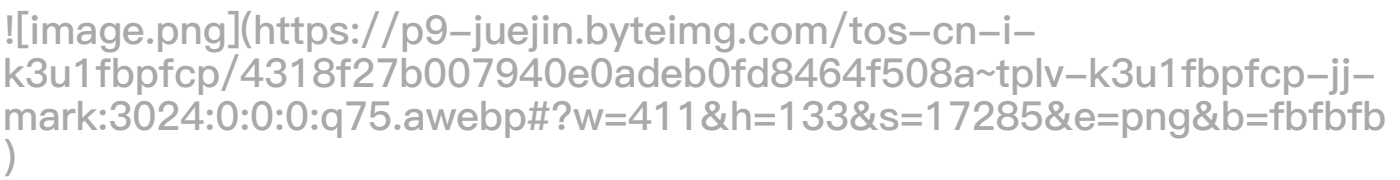


出现这种结果显然不符合我们的预期，这时`supports()`方法就派上用场了，修改方法：

```
//只封装非HashMap对象
@Override
public boolean supports(MethodParameter returnType, Class
converterType) {
    return !returnType.getParameterType().equals(HashMap.class);
}
```

在写程序前，我们可以提前约定好，将所有**响应信息**都使用`HashMap`进行返回，这样我们就可以使用上面的代码，在封装响应前先判断是否为已经封装好的响应信息，如果是，就不继续嵌套封装。

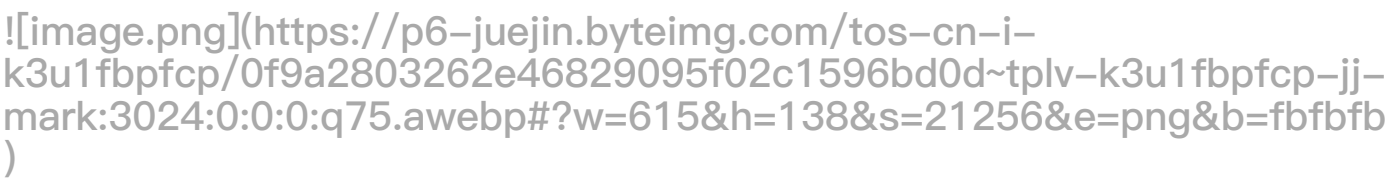
成功解决响应嵌套问题：



3.3 【引入】Controller层返回String时的异常

此时我们在测试一下，如果Controller层返回的类型非HashMap，会将响应封装成什么样式：

访问`/login`路由，我们惊奇的发现返回的是错误响应信息，并且错误信息是`"java.util.HashMap cannot be cast to java.lang.String"`



这个问题在初次遇到的时候困扰了我很久，这是怎么回事呢？原因如下：

在Controller中添加了@ResponseBody之后，Spring就会根据返回的类型选择合适的消息转化器进行转换。

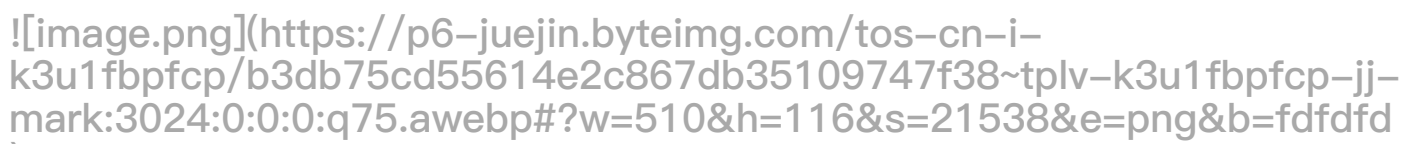
在前面的Spring MVC中讲过返回String本质上是返回一个html页面，而不是一个Json对象。如果要返回一个Json对象的话，会使用内置的`Jackson`框架进行处理。

由于返回String比较特殊，当Controller层返回一个String的时候，就会使用`StringHttpMessageConverter`这个消息转化器进行处理，但是我们进行统一数据返回格式处理的业务后，在将响应返回给前端之前，会对Controller进行拦截，然后将其分装成一个`HashMap`对象后返回，返回响应后，`StringHttpMessageConverter`一看，咋返回的不是个`String`，然后就将HashMap对象进行强行转换为String，于是就抛出了`"java.util.HashMap cannot be cast to java.lang.String"`的异常。

解决方案：修改`beforeBodyWrite()`方法，当body为String的时候，自行调用Jackson的 `ObjectMapper`对象进行特殊处理，并返回给前端。

```
//处理统一数据返回格式的逻辑
@Override
public Object beforeBodyWrite(Object body, MethodParameter
returnType, MediaType selectedContentType, Class
selectedConverterType, ServerHttpRequest request, ServerHttpResponse
response) {
    HashMap<String, Object> retMap = new HashMap<>();
    retMap.put(”code”, ”200”);
    retMap.put(”msg”, ””);
    retMap.put(”body”, body);

    if (body instanceof String) {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            return objectMapper.writeValueAsString(retMap);
        } catch (JsonProcessingException e) {
            throw new RuntimeException(e);
        }
    }
    return retMap;
}
```

成功获取到封装好的响应信息：



4. 总结
=====

本篇文章中讲解了三种统一功能的实现：用户登录权限验证、统一异常处理、统一数据返回格式。

对上面三种统一功能详细讲解了：

1. 如何使用拦截器实现用户登录权限验证
2. 通过Spring MVC中`@ControllerAdvice`和`@ExceptionHandler`实现统一异常处理
3. 通过`@ControllerAdvice`和`ResponseBodyAdvice`接口实现统一数据返回格式的处理

原文链接: https://juejin.cn/post/7360512664316231714