10. SpringBoot 统一功能处理

本节目标

1. 掌握拦截器的使用, 及其原理
2. 学习统一数据返回格式和统一异常处理的操作
3. 了解一些Spring的源码

1. 拦截器

上个章节我们完成了强制登录的功能, 后端程序根据Session来判断用户是否登录, 但是实现方法是比较麻烦的

- 需要修改每个接口的处理逻辑
- 需要修改每个接口的返回结果
- 接口定义修改, 前端代码也需要跟着修改

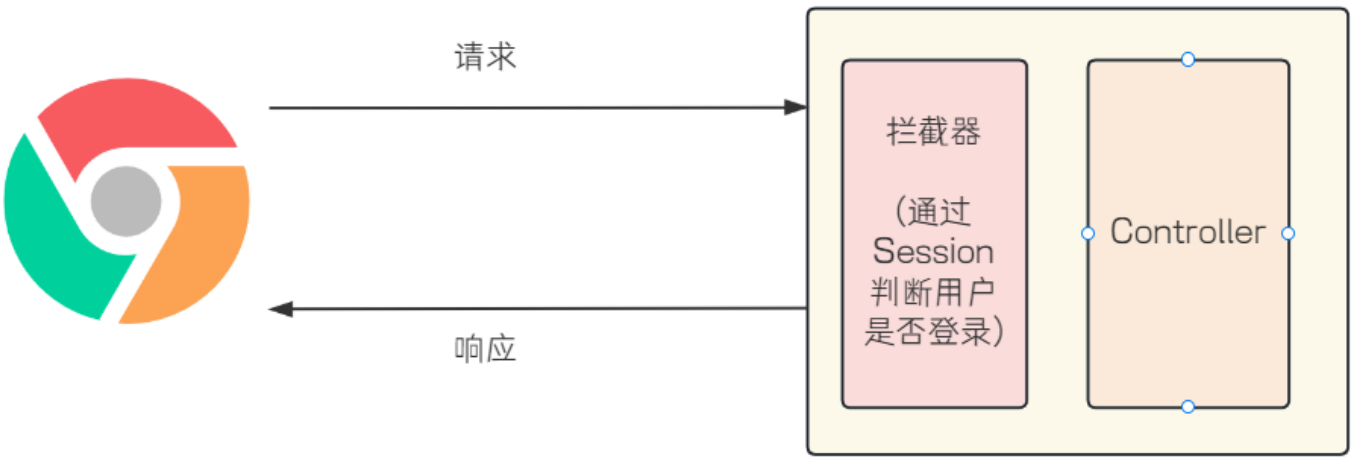
有没有更简单的办法, 统一拦截所有的请求, 并进行Session校验呢, 这里我们学习一种新的解决办法: 拦截器

1.1 拦截器快速入门

什么是拦截器?

拦截器是Spring框架提供的核心功能之一, 主要用来拦截用户的请求, 在指定方法前后, 根据业务需要执行预先设定的代码.

> 也就是说, 允许开发人员提前预定义一些逻辑, 在用户的请求响应前后执行. 也可以在用户请求前阻止其执行.
>
> 在拦截器当中，开发人员可以在应用程序中做一些通用性的操作, 比如通过拦截器来拦截前端发来的请求, 判断Session中是否有登录用户的信息. 如果有就可以放行, 如果没有就进行拦截.



比如我们去银行办理业务, 在办理业务前后, 就可以加一些拦截操作

办理业务之前, 先取号, 如果带身份证了就取号成功

业务办理结束, 给业务办理人员的服务进行评价.

这些就是"拦截器"做的工作.

下面我们先来学习下拦截器的基本使用.

拦截器的使用步骤分为两步：

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

```
@Slf4j
@Component
public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        log.info("LoginInterceptor 目标方法执行前执行..");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        log.info("LoginInterceptor 目标方法执行后执行");
    }

    @Override
```

```
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws Exception {
        log.info("LoginInterceptor 视图渲染完毕后执行，最后执行");
    }
}
```

- preHandle()方法：目标方法执行前执行. 返回true: 继续执行后续操作; 返回false: 中断后续操作.
- postHandle()方法：目标方法执行后执行
- afterCompletion()方法：视图渲染完毕后执行，最后执行(后端开发现在几乎不涉及视图, 暂不了解)

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

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    //自定义的拦截器对象
    @Autowired
    private LoginInterceptor loginInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //注册自定义拦截器对象
        registry.addInterceptor(loginInterceptor)
                .addPathPatterns("/**");//设置拦截器拦截的请求路径（ /** 表示拦截所有请求）
    }
}
```

启动服务, 试试访问任意请求, 观察后端日志

```
2023-09-27 16:56:04.171  INFO 29584 --- [nio-8080-exec-2] c.e.demo.interceptor.LoginInterceptor    : LoginInterceptor 目标方法执行前执行.
2023-09-27 16:56:04.183  INFO 29584 --- [nio-8080-exec-2] c.e.demo.controller.BookController       : 查询图书信息queryBookById, Id:20
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@1efc546a] was not registered for synchronization because synchronization is not acti
2023-09-27 16:56:04.197  INFO 29584 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2023-09-27 16:56:04.324  INFO 29584 --- [nio-8080-exec-2] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
JDBC Connection [HikariProxyConnection@1107736237 wrapping com.mysql.cj.jdbc.ConnectionImpl@51c8e388] will not be managed by Spring
==>  Preparing: select id, book_name, author, count, price, publish, `status`, create_time, update_time from book_info where id=? and status<>0
==> Parameters: 20(Integer)
<==    Columns: id, book_name, author, count, price, publish, status, create_time, update_time
<==        Row: 20, 图书17, 作者2, 29, 22.00, 出版社1, 1, 2023-09-24 17:45:50, 2023-09-27 16:40:50
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@1efc546a]
2023-09-27 16:56:04.411  INFO 29584 --- [nio-8080-exec-2] c.e.demo.interceptor.LoginInterceptor    : LoginInterceptor 目标方法执行后执行
2023-09-27 16:56:04.412  INFO 29584 --- [nio-8080-exec-2] c.e.demo.interceptor.LoginInterceptor    : LoginInterceptor 视图渲染完毕后执行，最后执行
```

可以看到preHandle 方法执行之后就放行了, 开始执行目标方法, 目标方法执行完成之后执行postHandle和afterCompletion方法.

我们把拦截器中preHandle方法的返回值改为false, 再观察运行结果

```
2023-09-27 17:04:02.552  INFO 35708 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-09-27 17:04:02.553  INFO 35708 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2023-09-27 17:04:02.553  INFO 35708 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
2023-09-27 17:04:02.568  INFO 35708 --- [nio-8080-exec-1] c.e.demo.interceptor.LoginInterceptor    : LoginInterceptor 目标方法执行前执行..
```

可以看到, 拦截器拦截了请求, 没有进行响应.

1.2 拦截器详解

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. 拦截器实现原理

1.2.1 拦截路径

拦截路径是指我们定义的这个拦截器, 对哪些请求生效.

我们在注册配置拦截器的时候, 通过 `addPathPatterns()` 方法指定要拦截哪些请求. 也可以通过 `excludePathPatterns()` 指定不拦截哪些请求.

> 上述代码中, 我们配置的是 `/**` , 表示拦截所有的请求.

比如用户登录校验, 我们希望可以对除了登录之外所有的路径生效.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {
    //自定义的拦截器对象
    @Autowired
    private LoginInterceptor loginInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //注册自定义拦截器对象
        registry.addInterceptor(loginInterceptor)
                .addPathPatterns("/**")
                .excludePathPatterns("/user/login");//设置拦截器拦截的请求路径(/** 表示拦截所有请求)
    }
```

```
19 }
```

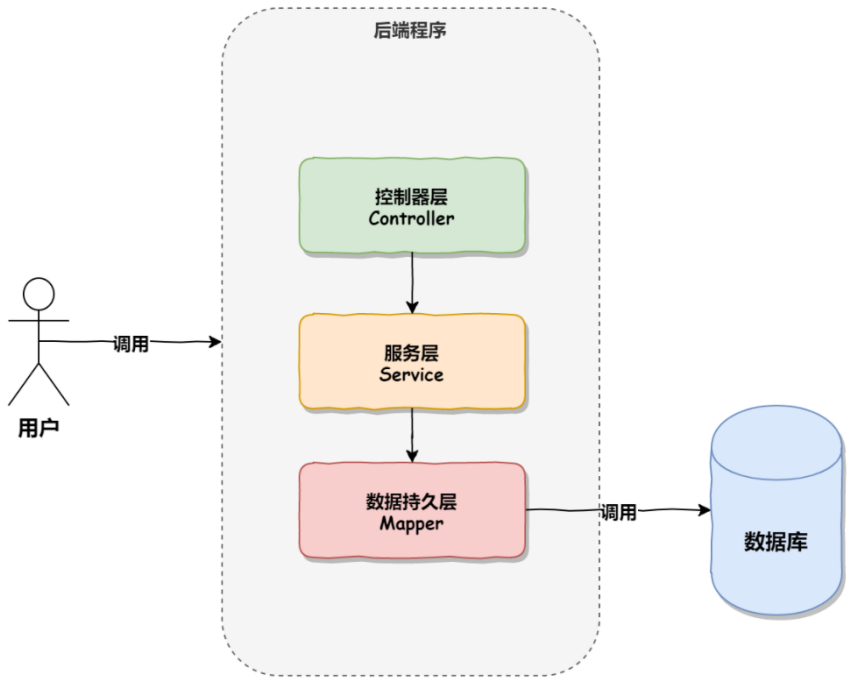
在拦截器中除了可以设置 `/**` 拦截所有资源外，还有一些常见拦截路径设置：

拦截路径	含义	举例
/*	一级路径	能匹配/user，/book，/login，不能匹配 /user/login
/**	任意级路径	能匹配/user，/user/login，/user/reg
/book/*	/book下的一级路径	能匹配/book/addBook，不能匹配/book/addBook/1，/book
/book/**	/book下的任意级路径	能匹配/book，/book/addBook，/book/addBook/2，不能匹配/user/login

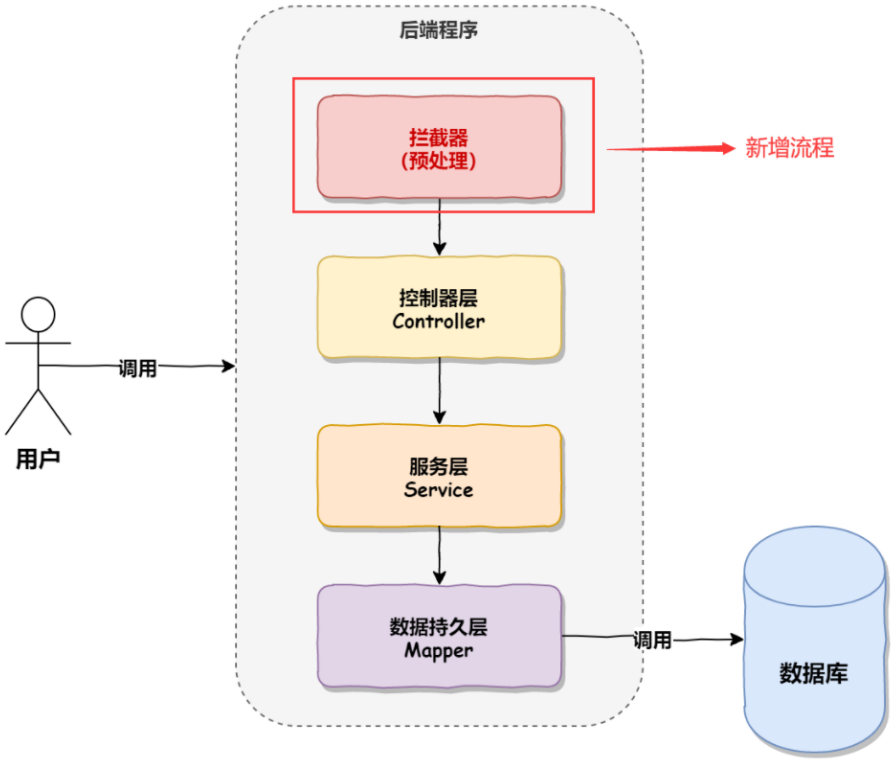
> 以上拦截规则可以拦截此项目中的使用 URL，包括静态文件(图片文件, JS 和 CSS 等文件).

1.2.2 拦截器执行流程

正常的调用顺序:



有了拦截器之后，会在调用 Controller 之前进行相应的业务处理，执行的流程如下图



1. 添加拦截器后, 执行Controller的方法之前, 请求会先被拦截器拦截住. 执行 `preHandle()` 方法，这个方法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, 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).
2. controller当中的方法执行完毕后，再回过来执行 `postHandle()` 这个方法以及 `afterCompletion()` 方法，执行完毕之后，最终给浏览器响应数据.

1.3 登录校验

学习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：通过拦截器来完成图书管理系统中的登录校验功能

1.3.1 定义拦截器

从session中获取用户信息, 如果session中不存在, 则返回false,并设置http状态码为401, 否则返回true.

```
1  import com.example.demo.constant.Constants;
2  import lombok.extern.slf4j.Slf4j;
3  import org.springframework.stereotype.Component;
4  import org.springframework.web.servlet.HandlerInterceptor;
5  import javax.servlet.http.HttpServletRequest;
6  import javax.servlet.http.HttpServletResponse;
7  import javax.servlet.http.HttpSession;
8
9  @Slf4j
10 @Component
```

```
11 public class LoginInterceptor implements HandlerInterceptor {
12
13     @Override
14     public boolean preHandle(HttpServletRequest request, HttpServletResponse
   response, Object handler) throws Exception {
15         HttpSession session = request.getSession(false);
16         if (session != null &&
   session.getAttribute(Constants.SESSION_USER_KEY) != null) {
17             return true;
18         }
19         response.setStatus(401);
20         return false;
21     }
22 }
```

> http状态码401: Unauthorized
>
> Indicates that authentication is required and was either not provided or has failed. If the request already included authorization credentials, then the 401 status code indicates that those credentials were not accepted.
>
> 中文解释: **未经过认证**. 指示身份验证是必需的, 没有提供身份验证或身份验证失败. 如果请求已经包含授权凭据，那么401状态码表示不接受这些凭据。

1.3.2 注册配置拦截器

```
1  import org.springframework.beans.factory.annotation.Autowired;
2  import org.springframework.context.annotation.Configuration;
3  import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
4  import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
5
6  @Configuration
7  public class WebConfig implements WebMvcConfigurer {
8      //自定义的拦截器对象
9      @Autowired
10     private LoginInterceptor loginInterceptor;
11
12     @Override
13     public void addInterceptors(InterceptorRegistry registry) {
14         //注册自定义拦截器对象
15         registry.addInterceptor(loginInterceptor)
16                 .addPathPatterns("/**")//设置拦截器拦截的请求路径(/**表示拦截所有请
   求)
17                 .excludePathPatterns("/user/login")//设置拦截器排除拦截的路径
```

```
                .excludePathPatterns("/**/*.js") //排除前端静态资源
                .excludePathPatterns("/**/*.css")
                .excludePathPatterns("/**/*.png")
                .excludePathPatterns("/**/*.html");
    }
}
```

也可以改成

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.Arrays;
import java.util.List;

@Configuration
public class WebConfig implements WebMvcConfigurer {
    //自定义的拦截器对象
    @Autowired
    private LoginInterceptor loginInterceptor;
    private List<String> excludePaths = Arrays.asList(
            "/user/login",
            "/**/*.js",
            "/**/*.css",
            "/**/*.png",
            "/**/*.html"
    );
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //注册自定义拦截器对象
        registry.addInterceptor(loginInterceptor)
                .addPathPatterns("/**")//设置拦截器拦截的请求路径(/** 表示拦截所有请求)
                .excludePathPatterns(excludePaths);//设置拦截器排除拦截的路径
    }
}
```

删除之前的登录校验代码

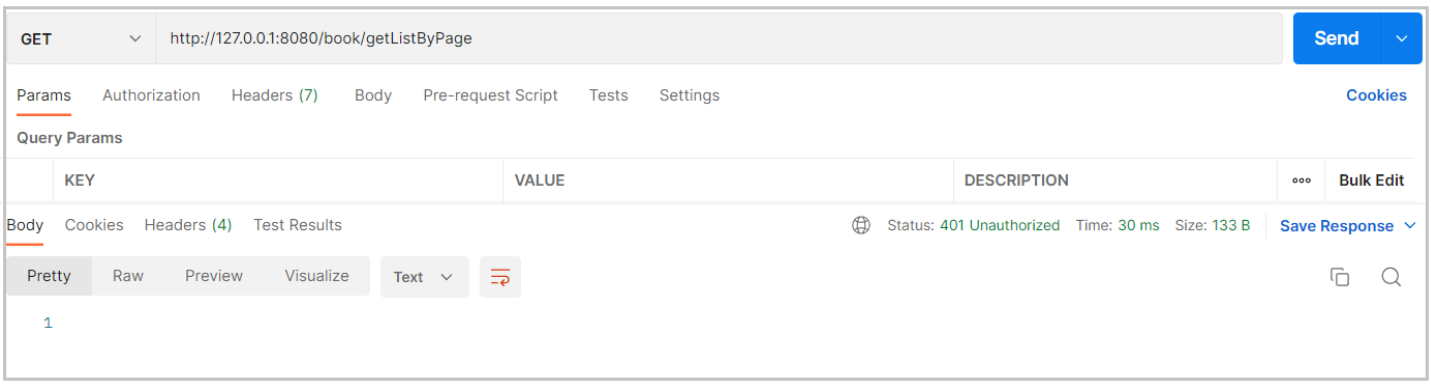
```
@RequestMapping("/getListByPage")
public Result getListByPage(PageRequest pageRequest, HttpSession session) {
```

```
3       log.info("获取图书列表, pageRequest:{}", pageRequest);
4   //      //判断用户是否登录
5   //      if (session.getAttribute(Constants.SESSION_USER_KEY)==null){
6   //          return Result.unlogin();
7   //      }
8   //      UserInfo userInfo = (UserInfo)
    session.getAttribute(Constants.SESSION_USER_KEY);
9   //      if (userInfo==null || userInfo.getId()<0 ||
    "".equals(userInfo.getUserName())){
10  //          return Result.unlogin();
11  //      }
12      //用户登录, 返回图书列表
13      PageResult<BookInfo> pageResult =
    bookService.getBookListByPage(pageRequest);
14      log.info("获取图书列表222, pageRequest:{}", pageResult);
15      return Result.success(pageResult);
16  }
```

运行程序, 通过Postman进行测试:

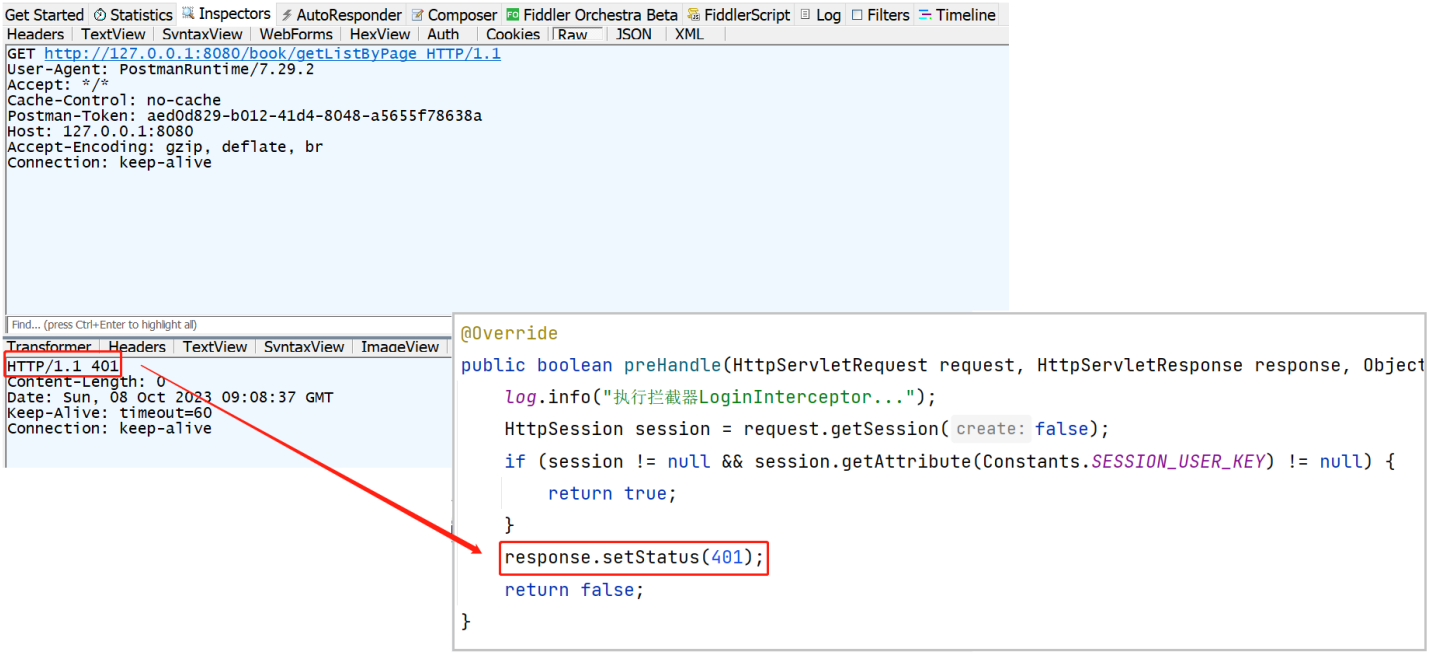
1. 查看图书列表

http://127.0.0.1:8080/book/getListByPage



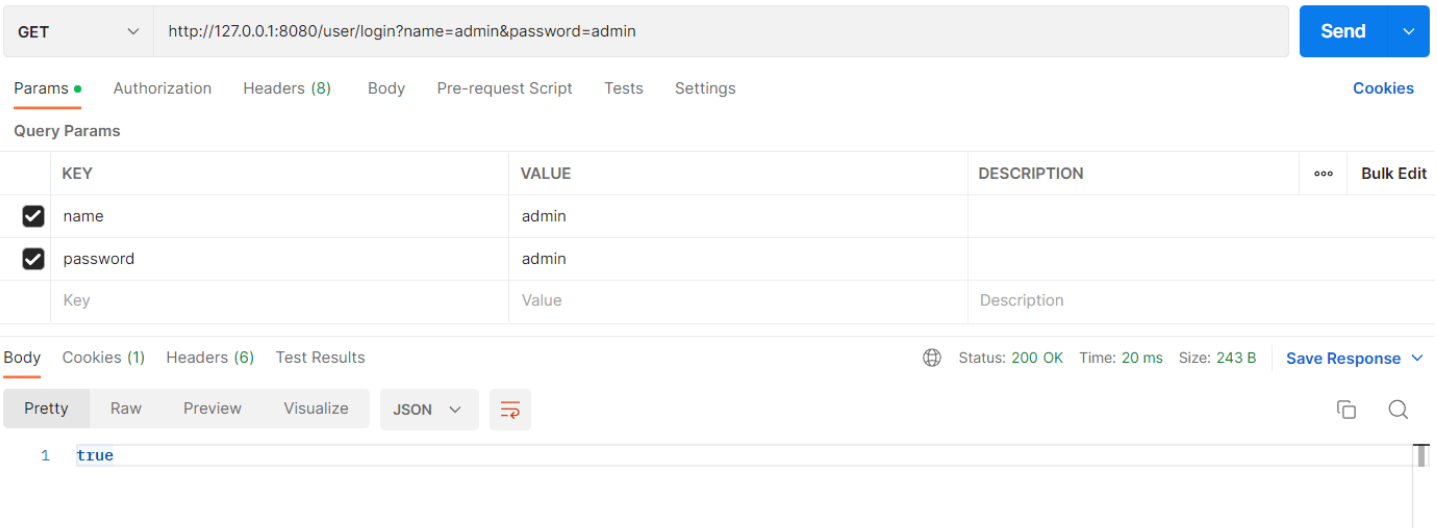
观察返回结果: http状态码401

也可以通过Fiddler抓包观察



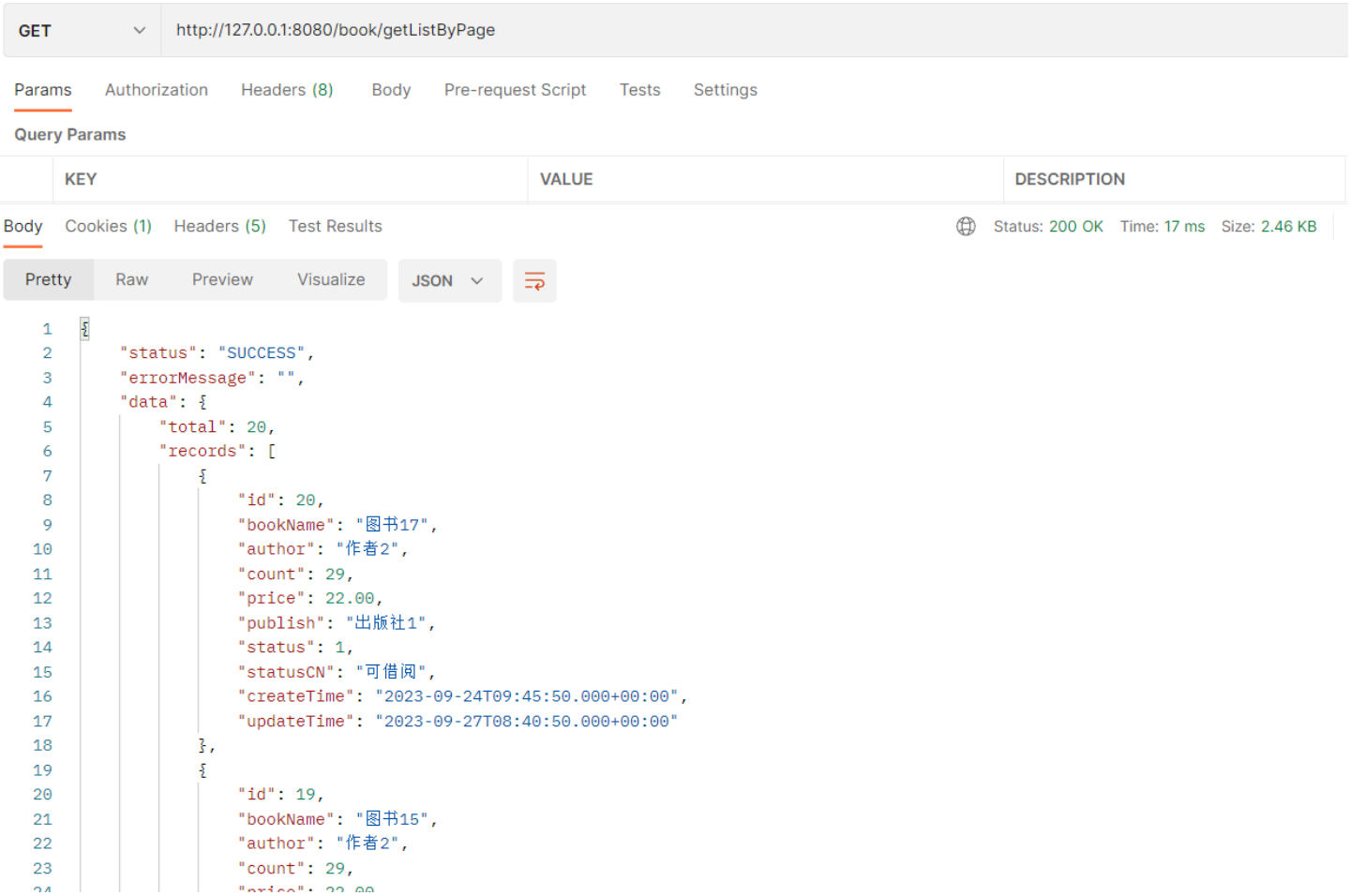
2. 登录

http://127.0.0.1:8080/user/login?name=admin&password=admin



3. 再次查看图书列表

数据进行了返回



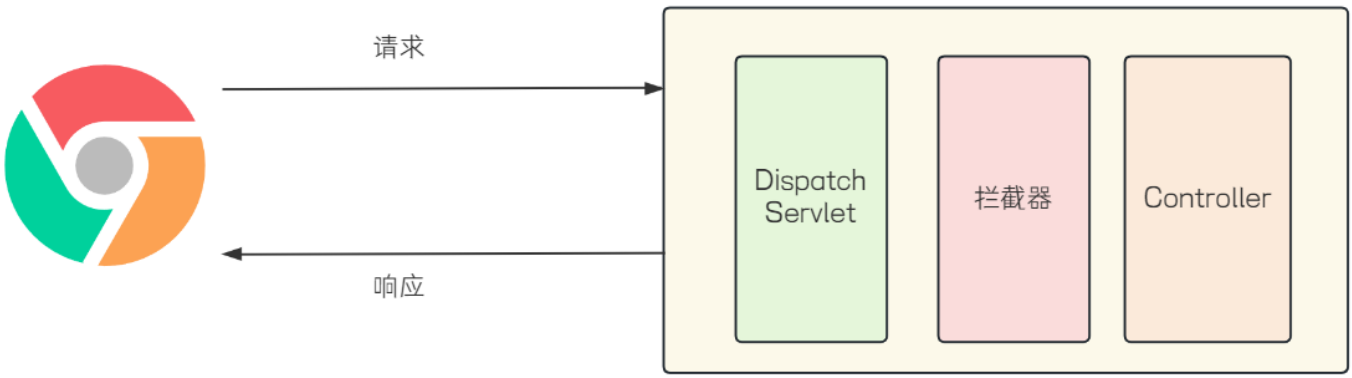
1.4 DispatcherServlet 源码分析(了解)

观察我们的服务启动日志:

```
2023-10-07 11:22:39.377  INFO 68960 --- [           main] com.example.demo.DemoApplication         : No active profile set, falling back to 1 default profile: "default"
2023-10-07 11:22:40.380  INFO 68960 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-10-07 11:22:40.387  INFO 68960 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-10-07 11:22:40.388  INFO 68960 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.79]
2023-10-07 11:22:40.531  INFO 68960 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-10-07 11:22:40.531  INFO 68960 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1115 ms
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
Parsed mapper file: 'file [D:\Git\JavaEE课件相关资料\后端代码\spring-book\target\classes\mapper\BookInfoMapper.xml]'
2023-10-07 11:22:41.126  INFO 68960 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2023-10-07 11:22:41.135  INFO 68960 --- [           main] com.example.demo.DemoApplication         : Started DemoApplication in 2.15 seconds (JVM running for 2.588)
2023-10-07 11:22:49.195  INFO 68960 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-10-07 11:22:49.195  INFO 68960 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2023-10-07 11:22:49.196  INFO 68960 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
```

当Tomcat启动之后, 有一个核心的类DispatcherServlet, 它来控制程序的执行顺序.

所有请求都会先进到DispatcherServlet，执行doDispatch 调度方法. 如果有拦截器, 会先执行拦截器 `preHandle()` 方法的代码, 如果 `preHandle()` 返回true, 继续访问controller中的方法. controller 当中的方法执行完毕后，再回过来执行 `postHandle()` 和 `afterCompletion()` ，返回给 DispatcherServlet, 最终给浏览器响应数据.



1.4.1 初始化(了解)

DispatcherServlet的初始化方法 init() 在其父类 HttpServletBean 中实现的.

主要作用是加载 web.xml 中 DispatcherServlet 的 配置, 并调用子类的初始化.

> web.xml是web项目的配置文件，一般的web工程都会用到web.xml来配置，主要用来配置Listener，Filter，Servlet等, Spring框架从3.1版本开始支持Servlet3.0, 并且从3.2版本开始通过配置DispatcherServlet, 实现不再使用web.xml

init() 具体代码如下:

```
@Override
public final void init() throws ServletException {
    try {
        // ServletConfigPropertyValues 是静态内部类，使用 ServletConfig 获取 web.xml 中配置的参数
        PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
        // 使用 BeanWrapper 来构造 DispatcherServlet
        BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
        ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());
        bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));
        initBeanWrapper(bw);
        bw.setPropertyValues(pvs, true);
    } catch (BeansException ex) {}
    // 让子类实现的方法，这种在父类定义在子类实现的方式叫做模版方法模式
    initServletBean();
}
```

在 `HttpServletBean` 的 `init()` 中调用了 `initServletBean()`, 它是在 FrameworkServlet 类中实现的, 主要作用是建立 WebApplicationContext 容器(有时也称上下文), 并

加载 SpringMVC 配置文件中定义的 Bean到该容器中, 最后将该容器添加到 ServletContext 中. 下面是 initServletBean() 的具体代码:

```
/**
 * Overridden method of {@link HttpServletBean}, invoked after any bean properties
 * have been set. Creates this servlet's WebApplicationContext.
 */
@Override
protected final void initServletBean() throws ServletException {
   getServletContext().log("Initializing Spring " + getClass().getSimpleName() + " '" + getServletName() + "'");
   if (logger.isInfoEnabled()) {
      logger.info("Initializing Servlet '" + getServletName() + "'");
   }
   long startTime = System.currentTimeMillis();

   try {
      //创建ApplicationContext容器
      this.webApplicationContext = initWebApplicationContext();
      initFrameworkServlet();
   }
   catch (ServletException | RuntimeException ex) {
      logger.error("Context initialization failed", ex);
      throw ex;
   }

   if (logger.isDebugEnabled()) {
      String value = this.enableLoggingRequestDetails ?
            "shown which may lead to unsafe logging of potentially sensitive data" :
            "masked to prevent unsafe logging of potentially sensitive data";
      logger.debug("enableLoggingRequestDetails='" + this.enableLoggingRequestDetails +
            "': request parameters and headers will be " + value);
   }

   if (logger.isInfoEnabled()) {
      logger.info("Completed initialization in " + (System.currentTimeMillis() - startTime) + " ms");
   }
}

```

此处打印的日志, 也正是控制台打印出来的日志

```
2023-10-07 11:22:39.377  INFO 68960 --- [           main] com.example.demo.DemoApplication         : No active profile set, falling back to 1 default profile: "default"
2023-10-07 11:22:40.380  INFO 68960 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-10-07 11:22:40.387  INFO 68960 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-10-07 11:22:40.388  INFO 68960 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.79]
2023-10-07 11:22:40.531  INFO 68960 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-10-07 11:22:40.531  INFO 68960 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1115 ms
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
Parsed mapper file: 'file [D:\Git\JavaEE课件相关资料\后端代码\spring-book\target\classes\mapper\BookInfoMapper.xml]'
2023-10-07 11:22:41.126  INFO 68960 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2023-10-07 11:22:41.135  INFO 68960 --- [           main] com.example.demo.DemoApplication         : Started DemoApplication in 2.15 seconds (JVM running for 2.588)
2023-10-07 11:22:49.195  INFO 68960 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-10-07 11:22:49.195  INFO 68960 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2023-10-07 11:22:49.196  INFO 68960 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
```

> 源码跟踪技巧:
>
> 在阅读框架源码的时候, 一定要抓住关键点, 找到核心流程.
>
> 切忌从头到尾一行一行代码去看, 一个方法的去研究, 一定要找到关键流程, 抓住关键点, 先在宏观上对整个流程或者整个原理有一个认识, 有精力再去研究其中的细节.

初始化web容器的过程中, 会通过onRefresh 来初始化SpringMVC的容器

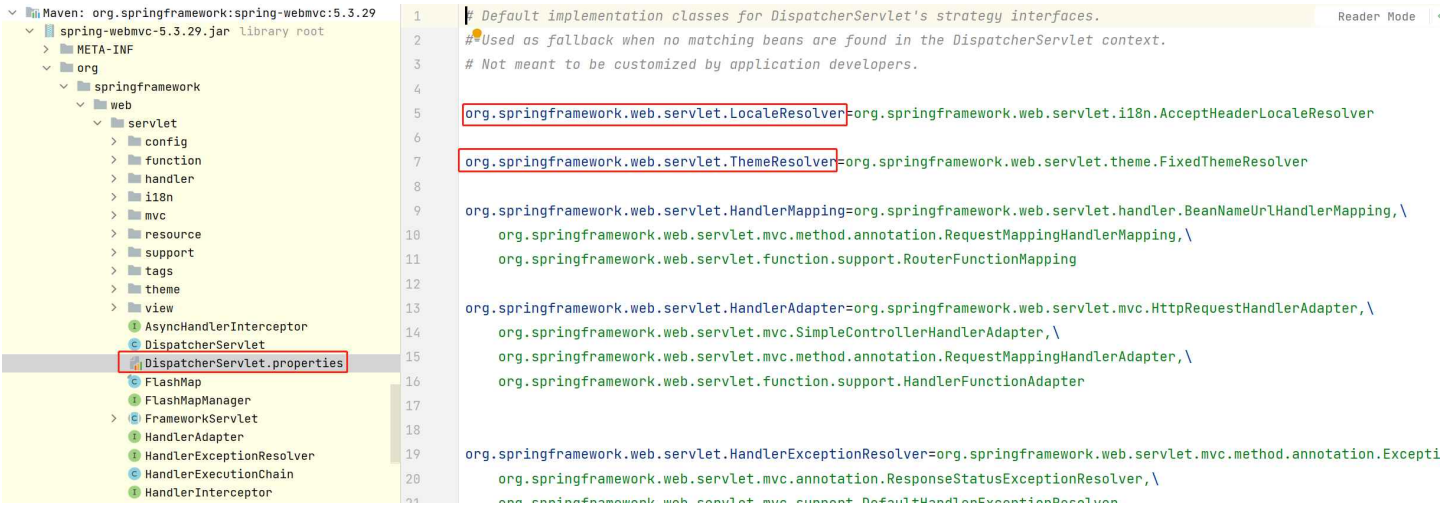
```java
protected WebApplicationContext initWebApplicationContext() {
    //...
    if (!this.refreshEventReceived) {
        //初始化Spring MVC
        synchronized (this.onRefreshMonitor) {
            onRefresh(wac);
        }
    }
    return wac;
}
```

```java
@Override
protected void onRefresh(ApplicationContext context) {
    initStrategies(context);
}

/**
 * Initialize the strategy objects that this servlet uses.
 * <p>May be overridden in subclasses in order to initialize further strategy objects.
 */
protected void initStrategies(ApplicationContext context) {
    initMultipartResolver(context);
    initLocaleResolver(context);
    initThemeResolver(context);
    initHandlerMappings(context);
    initHandlerAdapters(context);
    initHandlerExceptionResolvers(context);
```

```
17      initRequestToViewNameTranslator(context);
18      initViewResolvers(context);
19      initFlashMapManager(context);
20  }
```

在initStrategies()中进行9大组件的初始化, 如果没有配置相应的组件，就使用默认定义的组件(在DispatcherServlet.properties中有配置默认的策略, 大致了解即可)



方法initMultipartResolver、initLocaleResolver、initThemeResolver、initRequestToViewNameTranslator、initFlashMapManager的处理方式几乎都一样(1.2.3.7.8,9),从应用文中取出指定的Bean, 如果没有, 就使用默认的.

方法initHandlerMappings、initHandlerAdapters、initHandlerExceptionResolvers的处理方式几乎都一样(4,5,6)

1. 初始化文件上传解析器MultipartResolver：从应用上下文中获取名称为multipartResolver的Bean，如果没有名为multipartResolver的Bean，则没有提供上传文件的解析器
2. 初始化区域解析器LocaleResolver：从应用上下文中获取名称为localeResolver的Bean，如果没有这个Bean，则默认使用AcceptHeaderLocaleResolver作为区域解析器
3. 初始化主题解析器ThemeResolver：从应用上下文中获取名称为themeResolver的Bean，如果没有这个Bean，则默认使用FixedThemeResolver作为主题解析器
4. 初始化处理器映射器HandlerMappings：处理器映射器作用，1）通过处理器映射器找到对应的处理器适配器，将请求交给适配器处理；2）缓存每个请求地址URL对应的位置（Controller.xxx方法）；如果在ApplicationContext发现有HandlerMappings，则从ApplicationContext中获取到所有的HandlerMappings，并进行排序；如果在ApplicationContext中没有发现有处理器映射器，则默认BeanNameUrlHandlerMapping作为处理器映射器

5. 初始化处理器适配器HandlerAdapter：作用是通过调用具体的方法来处理具体的请求；如果在ApplicationContext发现有handlerAdapter，则从ApplicationContext中获取到所有的HandlerAdapter，并进行排序；如果在ApplicationContext中没有发现处理器适配器，则默认SimpleControllerHandlerAdapter作为处理器适配器
6. 初始化异常处理器解析器HandlerExceptionResolver：如果在ApplicationContext发现有handlerExceptionResolver，则从ApplicationContext中获取到所有的HandlerExceptionResolver，并进行排序；如果在ApplicationContext中没有发现异常处理器解析器，则不设置异常处理器
7. 初始化RequestToViewNameTranslator：其作用是从Request中获取viewName，从ApplicationContext发现有viewNameTranslator的Bean，如果没有，则默认使用DefaultRequestToViewNameTranslator
8. 初始化视图解析器ViewResolvers：先从ApplicationContext中获取名为viewResolver的Bean，如果没有，则默认InternalResourceViewResolver作为视图解析器
9. 初始化FlashMapManager：其作用是用于检索和保存FlashMap（保存从一个URL重定向到另一个URL时的参数信息），从ApplicationContext发现有flashMapManager的Bean，如果没有，则默认使用DefaultFlashMapManager

1.4.2 处理请求(核心)

DispatcherServlet 接收到请求后, 执行doDispatch 调度方法, 再将请求转给Controller.

我们来看doDispatch 方法的具体实现

```
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;
                //1. 获取执行链
                //遍历所有的 HandlerMapping 找到与请求对应的Handler
                mappedHandler = this.getHandler(processedRequest);
```

```
18                 if (mappedHandler == null) {
19                     this.noHandlerFound(processedRequest, response);
20                     return;
21                 }
22                 //2. 获取适配器
23                 //遍历所有的 HandlerAdapter，找到可以处理该 Handler 的
   HandlerAdapter
24                 HandlerAdapter ha =
   this.getHandlerAdapter(mappedHandler.getHandler());
25                 String method = request.getMethod();
26                 boolean isGet = HttpMethod.GET.matches(method);
27                 if (isGet || HttpMethod.HEAD.matches(method)) {
28                     long lastModified = ha.getLastModified(request,
   mappedHandler.getHandler());
29                     if ((new ServletWebRequest(request,
   response)).checkNotModified(lastModified) && isGet) {
30                         return;
31                     }
32                 }
33                 //3. 执行拦截器preHandle方法
34                 if (!mappedHandler.applyPreHandle(processedRequest, response))
   {
35                     return;
36                 }
37
38                 //4. 执行目标方法
39                 mv = ha.handle(processedRequest, response,
   mappedHandler.getHandler());
40                 if (asyncManager.isConcurrentHandlingStarted()) {
41                     return;
42                 }
43
44                 this.applyDefaultViewName(processedRequest, mv);
45                 //5. 执行拦截器postHandle方法
46                 mappedHandler.applyPostHandle(processedRequest, response, mv);
47             } catch (Exception var20) {
48                 dispatchException = var20;
49             } catch (Throwable var21) {
50                 dispatchException = new NestedServletException("Handler
   dispatch failed", var21);
51             }
52             //6. 处理视图，处理之后执行拦截器afterCompletion方法
53             this.processDispatchResult(processedRequest, response,
   mappedHandler, mv, (Exception)dispatchException);
54         } catch (Exception var22) {
55             //7. 执行拦截器afterCompletion方法
```

```
56              this.triggerAfterCompletion(processedRequest, response,
    mappedHandler, var22);
57          } catch (Throwable var23) {
58              this.triggerAfterCompletion(processedRequest, response,
    mappedHandler, new NestedServletException("Handler processing failed", var23));
59          }
60
61      } finally {
62          if (asyncManager.isConcurrentHandlingStarted()) {
63              if (mappedHandler != null) {
64
    mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
65              }
66          } else if (multipartRequestParsed) {
67              this.cleanupMultipart(processedRequest);
68          }
69
70      }
71  }
```

> HandlerAdapter 在 Spring MVC 中使用了适配器模式, 下面详细再介绍
>
> 适配器模式, 也叫包装器模式. 简单来说就是目标类不能直接使用, 通过一个新类进行包装一下, 适配调用方使用.
>
> 把两个不兼容的接口通过一定的方式使之兼容.
>
> HandlerAdapter 主要用于支持不同类型的处理器（如 Controller、HttpRequestHandler 或者 Servlet 等），让它们能够适配统一的请求处理流程。这样，Spring MVC 可以通过一个统一的接口来处理来自各种处理器的请求.

从上述源码可以看出在开始执行 Controller 之前，会先调用 预处理方法 applyPreHandle，而 applyPreHandle 方法的实现源码如下：

```
1  boolean applyPreHandle(HttpServletRequest request, HttpServletResponse
    response) throws Exception {
2      for(int i = 0; i < this.interceptorList.size(); this.interceptorIndex =
    i++) {
3          // 获取项目中使用的拦截器 HandlerInterceptor
4          HandlerInterceptor interceptor =
    (HandlerInterceptor)this.interceptorList.get(i);
5          if (!interceptor.preHandle(request, response, this.handler)) {
6              this.triggerAfterCompletion(request, response, (Exception)null);
```

```
            return false;
        }
    }
    return true;
}
```

在 applyPreHandle 中会获取所有的拦截器 `HandlerInterceptor` , 并执行拦截器中的 preHandle 方法，这样就会咱们前面定义的拦截器对应上了，如下图所示：

```
@Slf4j
@Component
public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throw
        log.info("LoginInterceptor 目标方法执行前执行...");
        return true;
    }
```

如果拦截器返回true, 整个发放就返回true, 继续执行后续逻辑处理

如果拦截器返回fasle, 则中断后续操作

1.4.3 适配器模式

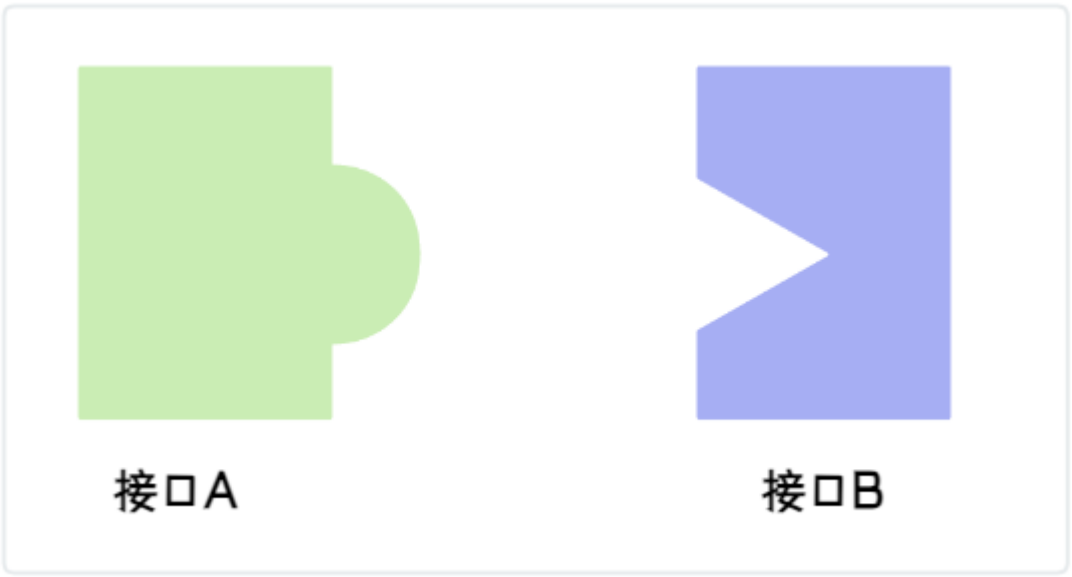
HandlerAdapter 在 Spring MVC 中使用了适配器模式

适配器模式定义

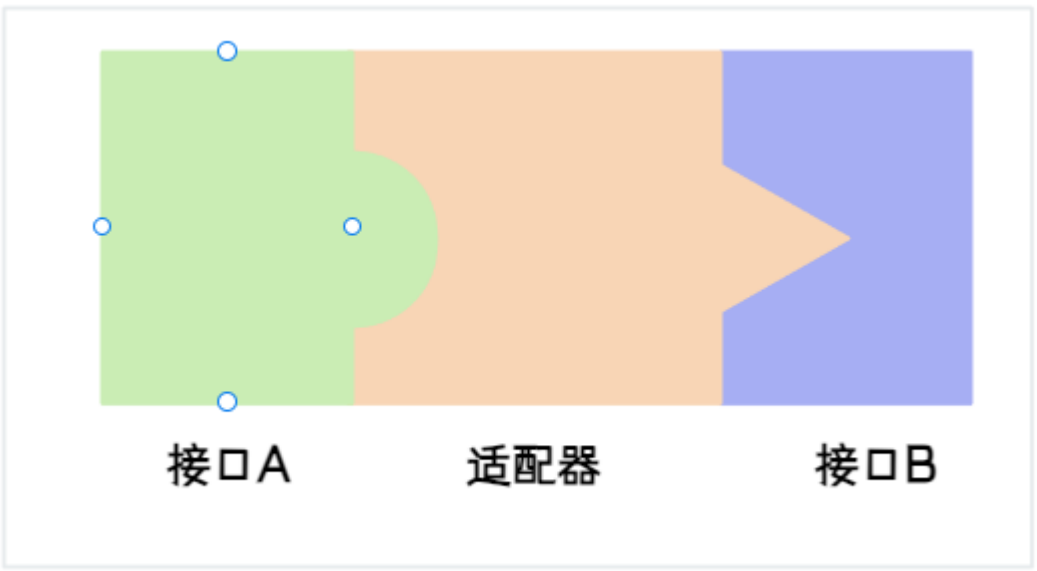
适配器模式, 也叫包装器模式. 将一个类的接口，转换成客户期望的另一个接口, 适配器让原本接口不兼容的类可以合作无间.

简单来说就是目标类不能直接使用, 通过一个新类进行包装一下, 适配调用方使用. 把两个不兼容的接口通过一定的方式使之兼容.

比如下面两个接口, 本身是不兼容的(参数类型不一样, 参数个数不一样等等)



可以通过适配器的方式, 使之兼容



> 日常生活中, 适配器模式也是非常常见的
>
> 比如转换插头, 网络转接头等

出国旅行必备物品之一就是转换插头. 不同国家的插头标准是不一样的, 出国后我们手机/电脑充电器可能就没办法使用了. 比如美国电器 110V，中国 220V，就要有一个适配器将 110V 转化为 220V. 国内也经常使用转换插头把两头转为三头, 或者三头转两头

适配器模式角色

- Target: 目标接口 (可以是抽象类或接口), 客户希望直接用的接口
- Adaptee: 适配者, 但是与Target不兼容
- Adapter: 适配器类, 此模式的核心. 通过继承或者引用适配者的对象, 把适配者转为目标接口
- client: 需要使用适配器的对象

适配器模式的实现

场景: 前面学习的slf4j 就使用了适配器模式, slf4j提供了一系列打印日志的api, 底层调用的是log4j 或者logback来打日志, 我们作为调用者, 只需要调用slf4j的api就行了.

```java
/**
 * slf4j接口
 */
interface Slf4jApi{
    void log(String message);
}

/**
 * log4j 接口
 */
class Log4j{
    void log4jLog(String message){
        System.out.println("Log4j打印:"+message);
    }
}
/**
 * slf4j和log4j适配器
 */

class Slf4jLog4JAdapter implements Slf4jApi{
    private Log4j log4j;

    public Slf4jLog4JAdapter(Log4j log4j) {
        this.log4j = log4j;
    }

    @Override
    public void log(String message) {
        log4j.log4jLog(message);
    }
}
/**
 * 客户端调用
 */
 public class Slf4jDemo {
    public static void main(String[] args) {
       Slf4jApi slf4jApi = new Slf4jLog4JAdapter(new Log4j());
       slf4jApi.log("使用slf4j打印日志");
    }
}
```

可以看出, 我们不需要改变log4j的api,只需要通过适配器转换下, 就可以更换日志框架, 保障系统的平稳运行.

> 适配器模式的实现并不在slf4j-core中(只定义了Logger), 具体实现是在针对log4j的桥接器项目slf4j-log4j12中

> 设计模式的使用非常灵活, 一个项目中通常会含有多种设计模式.

适配器模式应用场景

一般来说，适配器模式可以看作一种"补偿模式"，用来补救设计上的缺陷. 应用这种模式算是"无奈之举", 如果在设计初期，我们就能协调规避接口不兼容的问题, 就不需要使用适配器模式了

所以适配器模式更多的应用场景主要是对正在运行的代码进行改造, 并且希望可以复用原有代码实现新的功能. 比如版本升级等.

2. 统一数据返回格式

强制登录案例中, 我们共做了两部分工作

1. 通过Session来判断用户是否登录
2. 对后端返回数据进行封装, 告知前端处理的结果

> 回顾
>
> 后端统一返回结果

```
@Data
public class Result<T> {
    private int status;
    private String errorMessage;
    private T data;
}
```

> 后端逻辑处理

```
@RequestMapping("/getListByPage")
public Result getListByPage(PageRequest pageRequest) {
    log.info("获取图书列表, pageRequest:{}", pageRequest);
    //用户登录, 返回图书列表
    PageResult<BookInfo> pageResult =
bookService.getBookListByPage(pageRequest);
    log.info("获取图书列表222, pageRequest:{}", pageResult);
    return Result.success(pageResult);
}
```

> Result.success(pageResult) 就是对返回数据进行了封装

拦截器帮我们实现了第一个功能, 接下来看SpringBoot对第二个功能如何支持

2.1 快速入门

统一的数据返回格式使用 `@ControllerAdvice` 和 `ResponseBodyAdvice` 的方式实现

`@ControllerAdvice` 表示控制器通知类

添加类 `ResponseAdvice` , 实现 `ResponseBodyAdvice` 接口, 并在类上添加 `@ControllerAdvice` 注解

```
1  import com.example.demo.model.Result;
2  import org.springframework.core.MethodParameter;
3  import org.springframework.http.MediaType;
4  import org.springframework.http.server.ServerHttpRequest;
5  import org.springframework.http.server.ServerHttpResponse;
6  import org.springframework.web.bind.annotation.ControllerAdvice;
7  import
   org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;
8
9  @ControllerAdvice
10 public class ResponseAdvice implements ResponseBodyAdvice {
11     @Override
12     public boolean supports(MethodParameter returnType, Class converterType) {
13         return true;
14     }
15
16     @Override
17     public Object beforeBodyWrite(Object body, MethodParameter returnType,
   MediaType selectedContentType, Class selectedConverterType, ServerHttpRequest
   request, ServerHttpResponse response) {
18         return Result.success(body);
19     }
20 }
```

- supports方法: 判断是否要执行beforeBodyWrite方法. true为执行, false不执行. 通过该方法可以选择哪些类或哪些方法的response要进行处理, 其他的不进行处理.

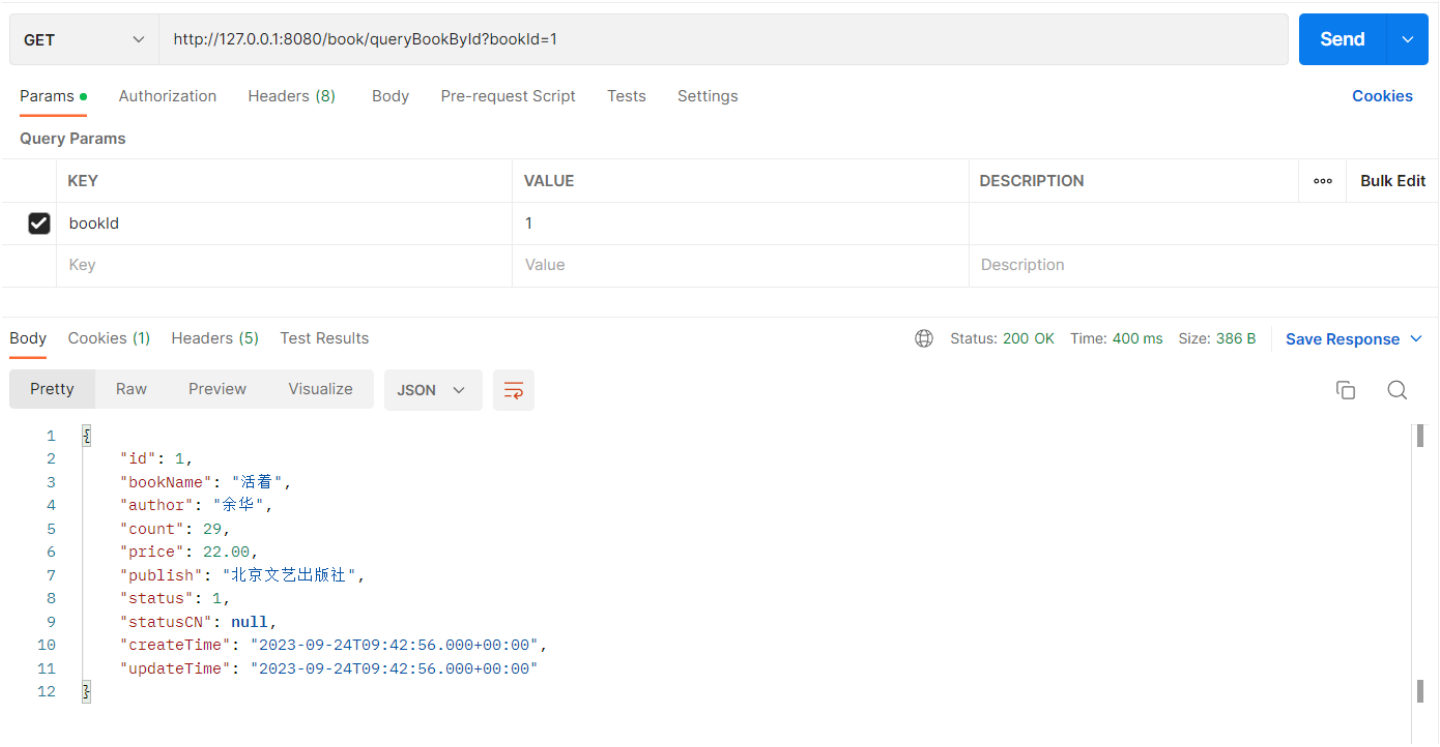
> 从returnType获取类名和方法名
>
> ```
> 1 //获取执行的类
> 2 Class<?> declaringClass = returnType.getMethod().getDeclaringClass();
> 3 //获取执行的方法
> 4 Method method = returnType.getMethod();
> ```

- beforeBodyWrite方法: 对response方法进行具体操作处理

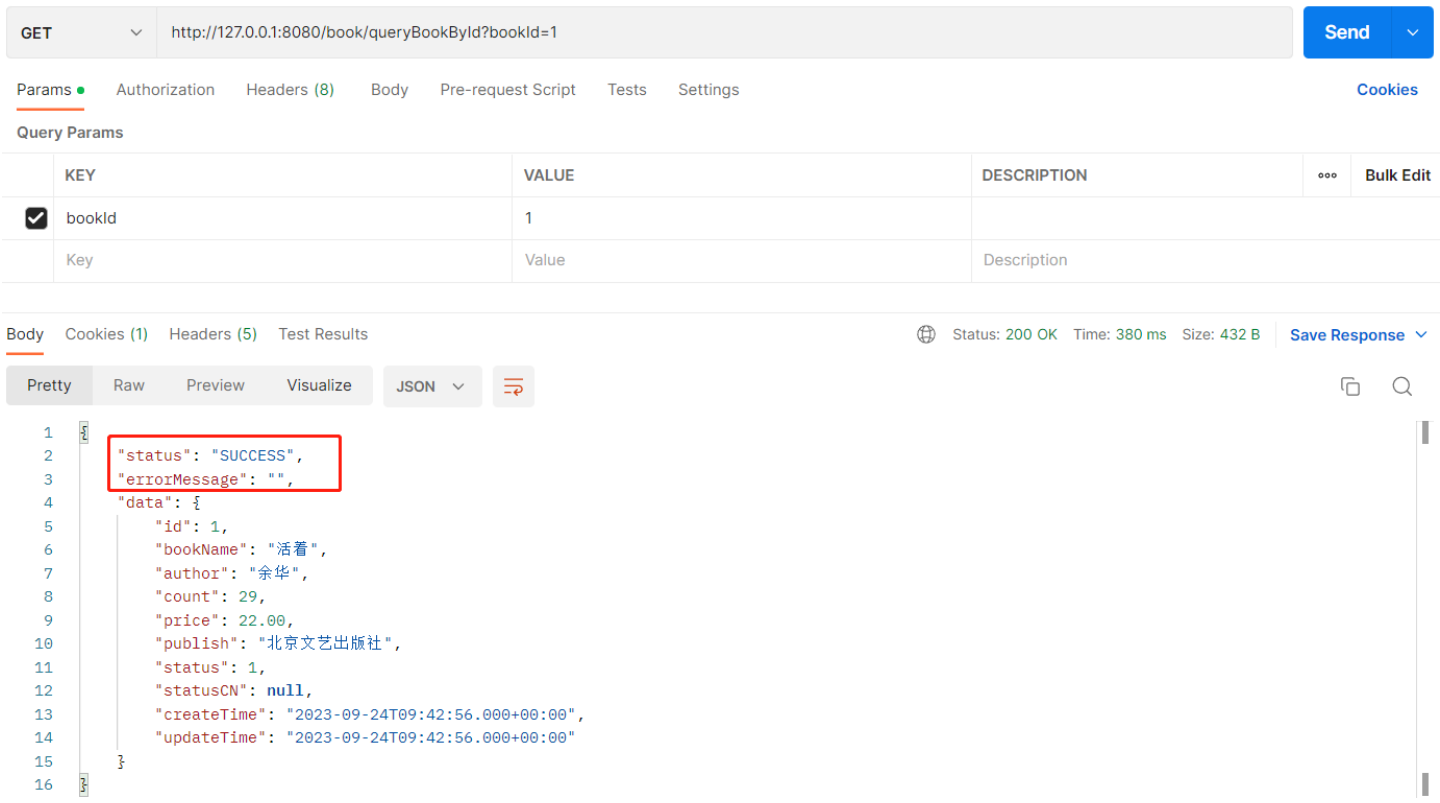
测试

测试接口: http://127.0.0.1:8080/book/queryBookById?bookId=1

添加统一数据返回格式之前:



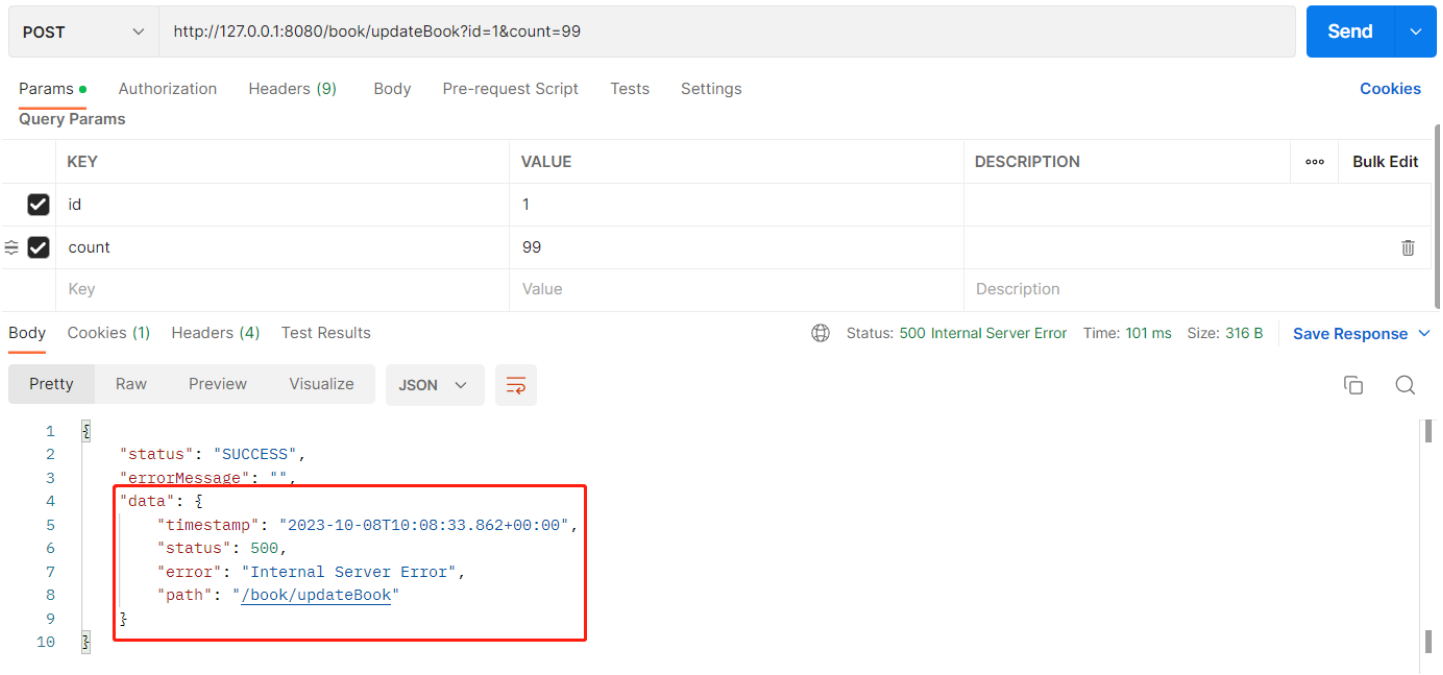
添加统一数据返回格式之后:



2.2 存在问题

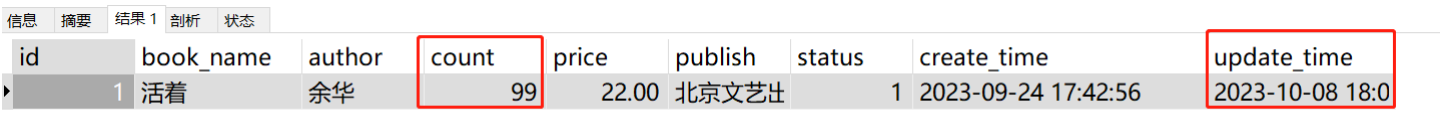
问题现象:

我们继续测试修改图书的接口: http://127.0.0.1:8080/book/updateBook



结果显示, 发生内部错误

查看数据库, 发现数据操作成功



查看日志, 日志报错

```
2023-10-08 18:08:33.856 ERROR 73748 --- [nio-8080-exec-3] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet] in context with
 path [] threw exception [Request processing failed; nested exception is java.lang.ClassCastException: com.example.demo.model.Result cannot be cast to java.lang
 .String] with root cause

java.lang.ClassCastException Create breakpoint : com.example.demo.model.Result cannot be cast to java.lang.String
    at org.springframework.http.converter.StringHttpMessageConverter.addDefaultHeaders(StringHttpMessageConverter.java:44) ~[spring-web-5.3.29.jar:5.3.29]
    at org.springframework.http.converter.AbstractHttpMessageConverter.write(AbstractHttpMessageConverter.java:211) ~[spring-web-5.3.29.jar:5.3.29]
    at org.springframework.web.servlet.mvc.method.annotation.AbstractMessageConverterMethodProcessor.writeWithMessageConverters(AbstractMessageConverterMethodProcessor
     .java:293) ~[spring-webmvc-5.3.29.jar:5.3.29]
    at org.springframework.web.servlet.mvc.method.annotation.RequestResponseBodyMethodProcessor.handleReturnValue(RequestResponseBodyMethodProcessor.java:183)
     ~[spring-webmvc-5.3.29.jar:5.3.29]
    at org.springframework.web.method.support.HandlerMethodReturnValueHandlerComposite.handleReturnValue(HandlerMethodReturnValueHandlerComposite.java:78)
     ~[spring-web-5.3.29.jar:5.3.29]
```

多测试几种不同的返回结果, 发现只有返回结果为String类型时才有这种错误发生.

测试代码:

```java
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RequestMapping("/test")
@RestController
```

```
public class TestController {
    @RequestMapping("/t1")
    public String t1(){
        return "t1";
    }
    @RequestMapping("/t2")
    public boolean t2(){
        return true;
    }
    @RequestMapping("/t3")
    public Integer t3(){
        return 200;
    }
}
```

解决方案:

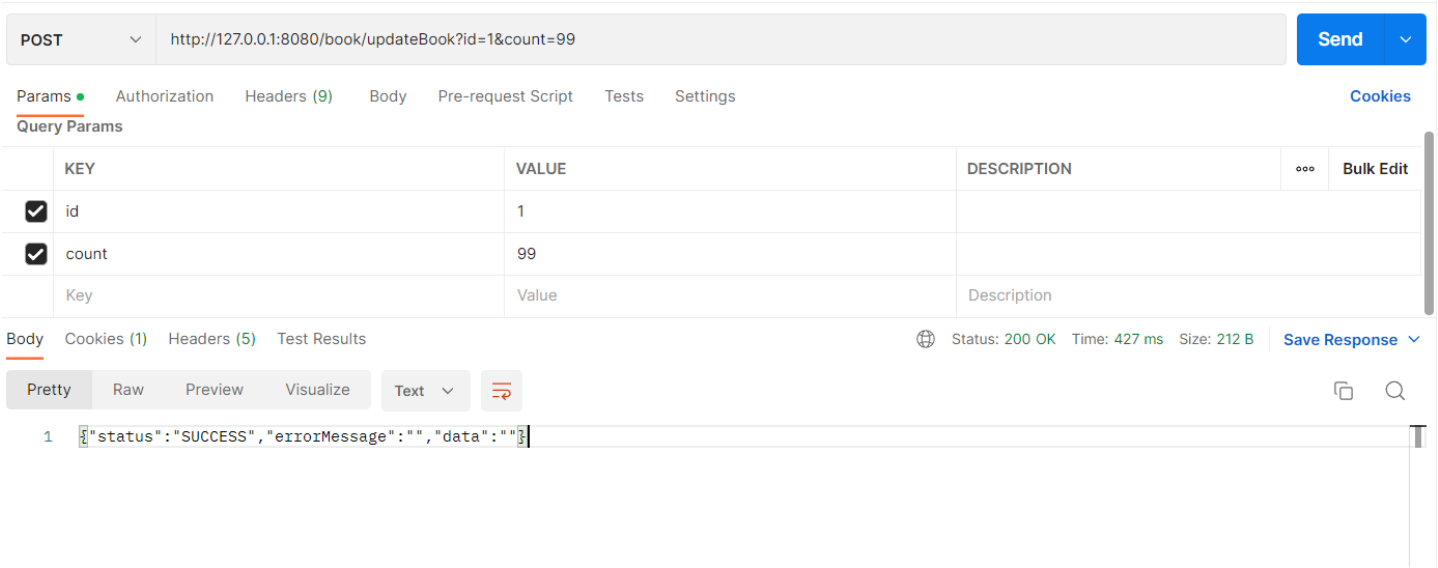
```
import com.example.demo.model.Result;
import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.SneakyThrows;
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.MethodParameter;
import org.springframework.http.MediaType;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.web.bind.annotation.ControllerAdvice;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;

@Slf4j
@ControllerAdvice
public class ResponseAdvice implements ResponseBodyAdvice {
    private static ObjectMapper mapper = new ObjectMapper();
    @Override
    public boolean supports(MethodParameter returnType, Class converterType) {
        return true;
    }

    @SneakyThrows
    @Override
    public Object beforeBodyWrite(Object body, MethodParameter returnType,
MediaType selectedContentType, Class selectedConverterType, ServerHttpRequest
request, ServerHttpResponse response) {
        //如果返回结果为String类型，使用SpringBoot内置提供的Jackson来实现信息的序列化
        if (body instanceof String){
```

```
26              return mapper.writeValueAsString(Result.success(body));
27          }
28          return Result.success(body);
29      }
30  }
```

重新测试, 结果返回正常:



原因分析:

SpringMVC默认会注册一些自带的`HttpMessageConverter`(从先后顺序排列分别为`ByteArrayHttpMessageConverter`,`StringHttpMessageConverter`,`SourceHttpMessageConverter`,`SourceHttpMessageConverter`,`AllEncompassingFormHttpMessageConverter`)

```
1  public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
2        implements BeanFactoryAware, InitializingBean {
3
4        //...
5        public RequestMappingHandlerAdapter() {
6         this.messageConverters = new ArrayList<>(4);
7         this.messageConverters.add(new ByteArrayHttpMessageConverter());
8         this.messageConverters.add(new StringHttpMessageConverter());
9         if (!shouldIgnoreXml) {
10           try {
11              this.messageConverters.add(new SourceHttpMessageConverter<>());
12           }
13           catch (Error err) {
14              // Ignore when no TransformerFactory implementation is available
15           }
16        }
```

```
17          this.messageConverters.add(new
    AllEncompassingFormHttpMessageConverter());
18      }
19        //...
20  }
```

其中AllEncompassingFormHttpMessageConverter 会根据项目依赖情况 添加对应的HttpMessageConverter

```
1  public AllEncompassingFormHttpMessageConverter() {
2     if (!shouldIgnoreXml) {
3        try {
4           addPartConverter(new SourceHttpMessageConverter<>());
5        }
6        catch (Error err) {
7           // Ignore when no TransformerFactory implementation is available
8        }
9
10       if (jaxb2Present && !jackson2XmlPresent) {
11          addPartConverter(new Jaxb2RootElementHttpMessageConverter());
12       }
13    }
14
15    if (kotlinSerializationJsonPresent) {
16       addPartConverter(new KotlinSerializationJsonHttpMessageConverter());
17    }
18    if (jackson2Present) {
19       addPartConverter(new MappingJackson2HttpMessageConverter());
20    }
21    else if (gsonPresent) {
22       addPartConverter(new GsonHttpMessageConverter());
23    }
24    else if (jsonbPresent) {
25       addPartConverter(new JsonbHttpMessageConverter());
26    }
27
28    if (jackson2XmlPresent && !shouldIgnoreXml) {
29       addPartConverter(new MappingJackson2XmlHttpMessageConverter());
30    }
31
32    if (jackson2SmilePresent) {
33       addPartConverter(new MappingJackson2SmileHttpMessageConverter());
34    }
35 }
```

在依赖中引入jackson包后，容器会把`MappingJackson2HttpMessageConverter`自动注册到`messageConverters`链的末尾.

Spring会根据返回的数据类型, 从`messageConverters`链选择合适的`HttpMessageConverter`.

当返回的数据是非字符串时, 使用的`MappingJackson2HttpMessageConverter`写入返回对象.

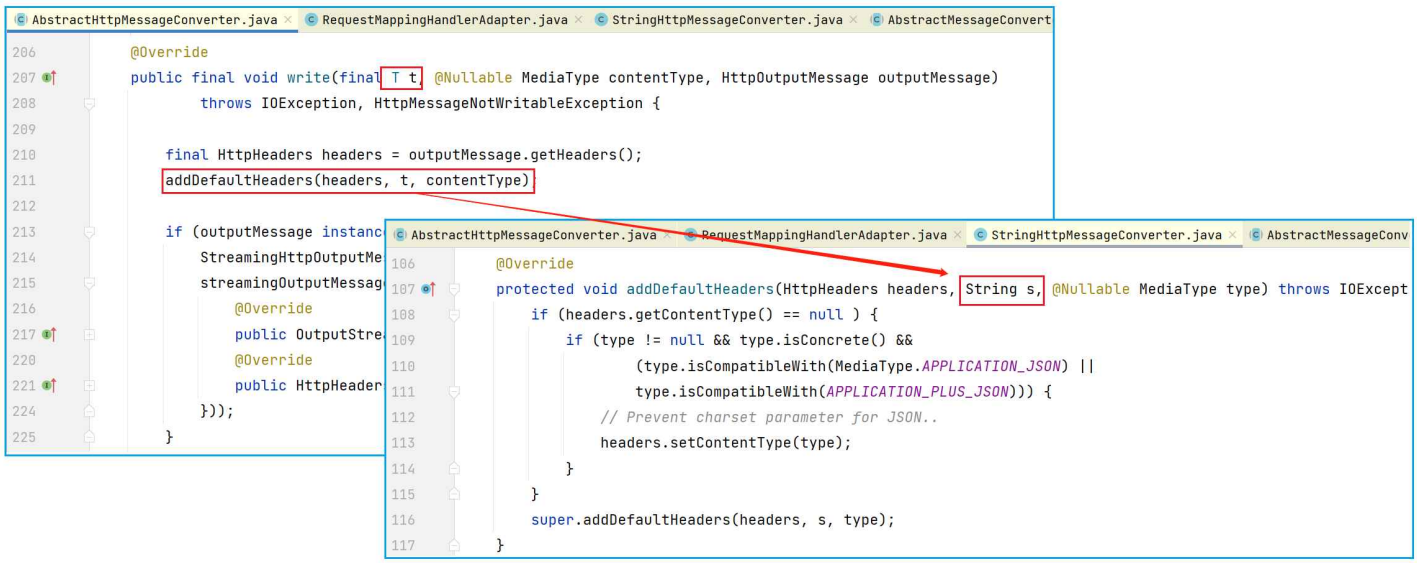
当返回的数据是字符串时，`StringHttpMessageConverter`会先被遍历到，这时会认为`StringHttpMessageConverter`可以使用.

```
1  public abstract class AbstractMessageConverterMethodProcessor extends
   AbstractMessageConverterMethodArgumentResolver
2        implements HandlerMethodReturnValueHandler {
3
4           //...代码省略
5        protected <T> void writeWithMessageConverters(@Nullable T value,
   MethodParameter returnType,
6        ServletServerHttpRequest inputMessage, ServletServerHttpResponse
   outputMessage)
7        throws IOException, HttpMediaTypeNotAcceptableException,
   HttpMessageNotWritableException {
8
9         //...代码省略
10         if (selectedMediaType != null) {
11            selectedMediaType = selectedMediaType.removeQualityValue();
12            for (HttpMessageConverter<?> converter : this.messageConverters) {
13               GenericHttpMessageConverter genericConverter = (converter
   instanceof GenericHttpMessageConverter ?
14                     (GenericHttpMessageConverter<?>) converter : null);
15               if (genericConverter != null ?
16                     ((GenericHttpMessageConverter)
   converter).canWrite(targetType, valueType, selectedMediaType) :
17                     converter.canWrite(valueType, selectedMediaType)) {
18                  //getAdvice().beforeBodyWrite 执行之后, body转换成了Result类型的
   结果
19                  body = getAdvice().beforeBodyWrite(body, returnType,
   selectedMediaType,
20                        (Class<? extends HttpMessageConverter<?>>)
   converter.getClass(),
21                        inputMessage, outputMessage);
22                  if (body != null) {
23                     Object theBody = body;
24                     LogFormatUtils.traceDebug(logger, traceOn ->
25                           "Writing [" + LogFormatUtils.formatValue(theBody,
   !traceOn) + "]");
26                     addContentDispositionHeader(inputMessage, outputMessage);
```

```
27                      if (genericConverter != null) {
28                          genericConverter.write(body, targetType,
    selectedMediaType, outputMessage);
29                      }
30                      else {
31                          //此时cover为StringHttpMessageConverter
32                          ((HttpMessageConverter) converter).write(body,
    selectedMediaType, outputMessage);
33                      }
34                  }
35                  else {
36                      if (logger.isDebugEnabled()) {
37                          logger.debug("Nothing to write: null body");
38                      }
39                  }
40                  return;
41              }
42          }
43      }
44      //...代码省略
45
46  }
47  //...代码省略
48 }
```

在 `((HttpMessageConverter) converter).write(body, selectedMediaType, outputMessage)` 的处理中, 调用父类的write方法

由于 `StringHttpMessageConverter` 重写了addDefaultHeaders方法, 所以会执行子类的方法



然而子类 `StringHttpMessageConverter` 的addDefaultHeaders方法定义接收参数为String, 此时t为Result类型, 所以出现类型不匹配"Result cannot be cast to java.lang.String"的异常

2.3 案例代码修改

如果一些方法返回的结果已经是Result类型了, 那就直接返回Result类型的结果即可

```
@SneakyThrows
@Override
public Object beforeBodyWrite(Object body, MethodParameter returnType,
MediaType selectedContentType, Class selectedConverterType, ServerHttpRequest
request, ServerHttpResponse response) {
    //返回结果更加灵活
    if (body instanceof Result){
        return body;
    }
    //如果返回结果为String类型, 使用SpringBoot内置提供的Jackson来实现信息的序列化
    if (body instanceof String){
        return mapper.writeValueAsString(Result.success(body));
    }
    return Result.success(body);
}
```

2.4 优点

1. 方便前端程序员更好的接收和解析后端数据接口返回的数据
2. 降低前端程序员和后端程序员的沟通成本, 按照某个格式实现就可以了, 因为所有接口都是这样返回的.
3. 有利于项目统一数据的维护和修改.
4. 有利于后端技术部门的统一规范的标准制定, 不会出现稀奇古怪的返回内容.

3. 统一异常处理

统一异常处理使用的是 `@ControllerAdvice` + `@ExceptionHandler` 来实现的, `@ControllerAdvice` 表示控制器通知类， `@ExceptionHandler` 是异常处理器，两个结合表示当出现异常的时候执行某个通知，也就是执行某个方法事件

具体代码如下:

```
import com.example.demo.model.Result;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

@ControllerAdvice
```

```
@ResponseBody
public class ErrorAdvice {

    @ExceptionHandler
    public Object handler(Exception e) {
        return Result.fail(e.getMessage());
    }
}
```

> 类名, 方法名和返回值可以自定义, 重要的是注解
>
> 接口返回为数据时, 需要加 `@ResponseBody` 注解

以上代码表示，如果代码出现Exception异常(包括Exception的子类), 就返回一个 Result的对象, Result对象的设置参考 `Result.fail(e.getMessage())`

```
public static Result fail(String msg) {
    Result result = new Result();
    result.setStatus(ResultStatus.FAIL);
    result.setErrorMessage(msg);
    result.setData("");
    return result;
}

```

我们可以针对不同的异常, 返回不同的结果.

```
import com.example.demo.model.Result;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

@ResponseBody
@ControllerAdvice
public class ErrorAdvice {

    @ExceptionHandler
    public Object handler(Exception e) {
        return Result.fail(e.getMessage());
    }
    @ExceptionHandler
    public Object handler(NullPointerException e) {
        return Result.fail("发生NullPointerException:"+e.getMessage());
    }
```

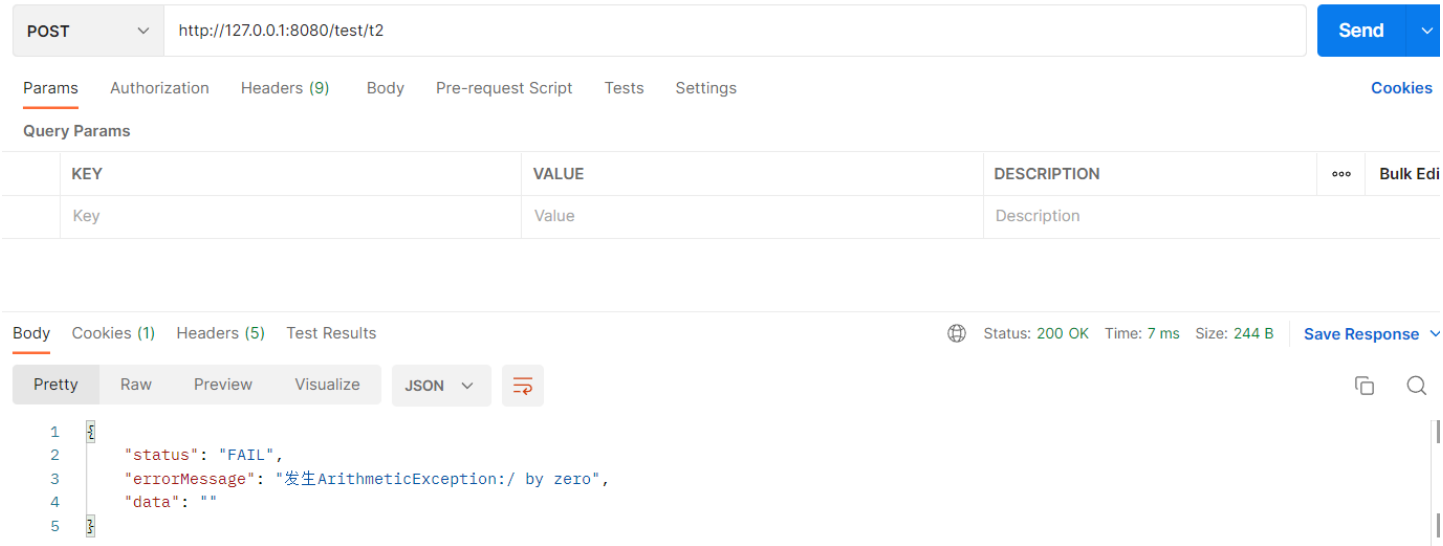
```
18
19      @ExceptionHandler
20      public Object handler(ArithmeticException e) {
21          return Result.fail("发生ArithmeticException:"+e.getMessage());
22      }
23  }
```

模拟制造异常:

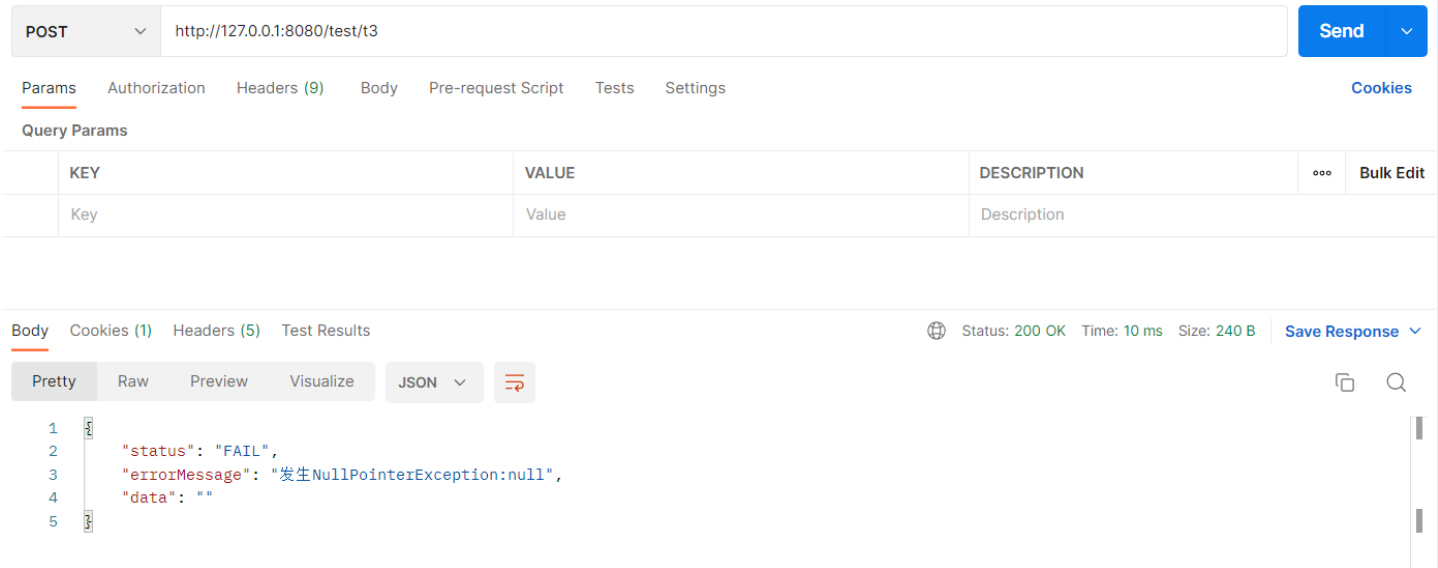
```
1  import org.springframework.web.bind.annotation.RequestMapping;
2  import org.springframework.web.bind.annotation.RestController;
3
4  @RequestMapping("/test")
5  @RestController
6  public class TestController {
7      @RequestMapping("/t1")
8      public String t1(){
9          return "t1";
10     }
11     @RequestMapping("/t2")
12     public boolean t2(){
13         int a = 10/0; //抛出ArithmeticException
14         return true;
15     }
16     @RequestMapping("/t3")
17     public Integer t3(){
18         String a =null;
19         System.out.println(a.length()); //抛出NullPointerException
20         return 200;
21     }
22 }
```

当有多个异常通知时，匹配顺序为当前类及其子类向上依次匹配

/test/t2 抛出ArithmeticException, 运行结果如下:



/test/t3 抛出NullPointerException, 运行结果如下:



4. @ControllerAdvice 源码分析

统一数据返回和统一异常都是基于 `@ControllerAdvice` 注解来实现的, 通过分析 `@ControllerAdvice` 的源码, 可以知道他们的执行流程.

点击 `@ControllerAdvice` 实现源码如下:

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Component
5  public @interface ControllerAdvice {
6      @AliasFor("basePackages")
7      String[] value() default {};
8
9      @AliasFor("value")
10     String[] basePackages() default {};
```

```
11
12      Class<?>[] basePackageClasses() default {};
13
14      Class<?>[] assignableTypes() default {};
15
16      Class<? extends Annotation>[] annotations() default {};
17  }
```

从上述源码可以看出 `@ControllerAdvice` 派生于 @Component 组件, 这也就是为什么没有五大注解, `ControllerAdvice` 就生效的原因.

下面我们看看Spring是怎么实现的, 还是从 `DispatcherServlet` 的代码开始分析.
`DispatcherServlet` 对象在创建时会初始化一系列的对象:

```
1  public class DispatcherServlet extends FrameworkServlet {
2      //...
3      @Override
4      protected void onRefresh(ApplicationContext context) {
5          initStrategies(context);
6      }
7
8      /**
9       * Initialize the strategy objects that this servlet uses.
10      * <p>May be overridden in subclasses in order to initialize further
    strategy objects.
11      */
12     protected void initStrategies(ApplicationContext context) {
13         initMultipartResolver(context);
14         initLocaleResolver(context);
15         initThemeResolver(context);
16         initHandlerMappings(context);
17         initHandlerAdapters(context);
18         initHandlerExceptionResolvers(context);
19         initRequestToViewNameTranslator(context);
20         initViewResolvers(context);
21         initFlashMapManager(context);
22     }
23     //...
24
25 }
```

对于 `@ControllerAdvice` 注解，我们重点关注 `initHandlerAdapters(context)` 和 `initHandlerExceptionResolvers(context)` 这两个方法.

1. initHandlerAdapters(context)

`initHandlerAdapters(context)` 方法会取得所有实现了 `HandlerAdapter` 接口的bean并保存起来，其中有一个类型为 `RequestMappingHandlerAdapter` 的bean，这个bean就是 `@RequestMapping` 注解能起作用的关键，这个bean在应用启动过程中会获取所有被 `@ControllerAdvice` 注解标注的bean对象, 并做进一步处理，关键代码如下:

```
public class RequestMappingHandlerAdapter extends AbstractHandlerMethodAdapter
      implements BeanFactoryAware, InitializingBean {
    //...


    /**
     * 添加ControllerAdvice bean的处理
     */
    private void initControllerAdviceCache() {
       if (getApplicationContext() == null) {
          return;
       }

       //获取所有所有被  @ControllerAdvice 注解标注的bean对象
       List<ControllerAdviceBean> adviceBeans =
ControllerAdviceBean.findAnnotatedBeans(getApplicationContext());

       List<Object> requestResponseBodyAdviceBeans = new ArrayList<>();

       for (ControllerAdviceBean adviceBean : adviceBeans) {
          Class<?> beanType = adviceBean.getBeanType();
          if (beanType == null) {
             throw new IllegalStateException("Unresolvable type for
ControllerAdviceBean: " + adviceBean);
          }
          Set<Method> attrMethods = MethodIntrospector.selectMethods(beanType,
MODEL_ATTRIBUTE_METHODS);
          if (!attrMethods.isEmpty()) {
             this.modelAttributeAdviceCache.put(adviceBean, attrMethods);
          }
          Set<Method> binderMethods =
MethodIntrospector.selectMethods(beanType, INIT_BINDER_METHODS);
          if (!binderMethods.isEmpty()) {
             this.initBinderAdviceCache.put(adviceBean, binderMethods);
          }
          if (RequestBodyAdvice.class.isAssignableFrom(beanType) ||
ResponseBodyAdvice.class.isAssignableFrom(beanType)) {
             requestResponseBodyAdviceBeans.add(adviceBean);
          }
```

```
35          }
36
37          if (!requestResponseBodyAdviceBeans.isEmpty()) {
38             this.requestResponseBodyAdvice.addAll(0,
    requestResponseBodyAdviceBeans);
39          }
40
41          if (logger.isDebugEnabled()) {
42             int modelSize = this.modelAttributeAdviceCache.size();
43             int binderSize = this.initBinderAdviceCache.size();
44             int reqCount = getBodyAdviceCount(RequestBodyAdvice.class);
45             int resCount = getBodyAdviceCount(ResponseBodyAdvice.class);
46             if (modelSize == 0 && binderSize == 0 && reqCount == 0 && resCount
    == 0) {
47                logger.debug("ControllerAdvice beans: none");
48             }
49             else {
50                logger.debug("ControllerAdvice beans: " + modelSize + "
    @ModelAttribute, " + binderSize +
51                      " @InitBinder, " + reqCount + " RequestBodyAdvice, " +
    resCount + " ResponseBodyAdvice");
52             }
53          }
54       }
55          //...
56
57
58    }
```

这个方法在执行时会查找使用所有的 `@ControllerAdvice` 类，把 `ResponseBodyAdvice` 类放在容器中，当发生某个事件时，调用相应的 Advice 方法，比如返回数据前调用统一数据封装

至于DispatcherServlet和RequestMappingHandlerAdapter是如何交互的这就是另一个复杂的话题了，此处不赘述, 源码部分难度比较高, 且枯燥, 大家以了解为主.

2. initHandlerExceptionResolvers(context)

接下来看 `DispatcherServlet` 的 `initHandlerExceptionResolvers(context)` 方法，这个方法会取得所有实现了 `HandlerExceptionResolver` 接口的bean并保存起来，其中就有一个类型为 `ExceptionHandlerExceptionResolver` 的bean，这个bean在应用启动过程中会获取所有被 `@ControllerAdvice` 注解标注的bean对象做进一步处理, 代码如下:

```
1  public class ExceptionHandlerExceptionResolver extends
   AbstractHandlerMethodExceptionResolver
2        implements ApplicationContextAware, InitializingBean {
3
```

```
4       //...
5
6     private void initExceptionHandlerAdviceCache() {
7        if (getApplicationContext() == null) {
8           return;
9        }
10
11       // 获取所有所有被  @ControllerAdvice 注解标注的bean对象
12       List<ControllerAdviceBean> adviceBeans =
ControllerAdviceBean.findAnnotatedBeans(getApplicationContext());
13       for (ControllerAdviceBean adviceBean : adviceBeans) {
14          Class<?> beanType = adviceBean.getBeanType();
15          if (beanType == null) {
16             throw new IllegalStateException("Unresolvable type for
ControllerAdviceBean: " + adviceBean);
17          }
18          ExceptionHandlerMethodResolver resolver = new
ExceptionHandlerMethodResolver(beanType);
19          if (resolver.hasExceptionMappings()) {
20             this.exceptionHandlerAdviceCache.put(adviceBean, resolver);
21          }
22          if (ResponseBodyAdvice.class.isAssignableFrom(beanType)) {
23             this.responseBodyAdvice.add(adviceBean);
24          }
25       }
26
27       if (logger.isDebugEnabled()) {
28          int handlerSize = this.exceptionHandlerAdviceCache.size();
29          int adviceSize = this.responseBodyAdvice.size();
30          if (handlerSize == 0 && adviceSize == 0) {
31             logger.debug("ControllerAdvice beans: none");
32          }
33          else {
34             logger.debug("ControllerAdvice beans: " +
35                   handlerSize + " @ExceptionHandler, " + adviceSize + "
ResponseBodyAdvice");
36          }
37       }
38    }
39       //...
40 }
```

当Controller抛出异常时，`DispatcherServlet`通过`ExceptionHandlerExceptionResolver`来解析异常，而`ExceptionHandlerExceptionResolver`又通过`ExceptionHandlerMethodResolver`

来解析异常， ExceptionHandlerMethodResolver 最终解析异常找到适用的@ExceptionHandler标注的方法是这里：

```
public class ExceptionHandlerMethodResolver {

    //...

    private Method getMappedMethod(Class<? extends Throwable> exceptionType) {
        List<Class<? extends Throwable>> matches = new ArrayList();
        //根据异常类型，查找匹配的异常处理方法
        //比如NullPointerException会匹配两个异常处理方法:
        //handler(Exception e) 和 handler(NullPointerException e)
        for (Class<? extends Throwable> mappedException :
this.mappedMethods.keySet()) {
            if (mappedException.isAssignableFrom(exceptionType)) {
                matches.add(mappedException);
            }
        }
        //如果查找到多个匹配，就进行排序，找到最使用的方法. 排序的规则依据抛出异常相对于
声明异常的深度
        //比如抛出的是NullPointerException(继承于RuntimeException,
RuntimeException又继承于Exception)
        //相对于handler(NullPointerException e) 声明的NullPointerException深度为0,
        //相对于handler(Exception e) 声明的Exception  深度 为2
        //所以 handler(NullPointerException e)标注的方法会排在前面
        if (!matches.isEmpty()) {
            if (matches.size() > 1) {
                matches.sort(new ExceptionDepthComparator(exceptionType));
            }
            return this.mappedMethods.get(matches.get(0));
        }
        else {
            return NO_MATCHING_EXCEPTION_HANDLER_METHOD;
        }
    }
    //...
}
```

5. 案例代码

通过上面统一功能的添加, 我们后端的接口已经发生了变化(后端返回的数据格式统一变成了Result类型), 所以我们需要对前端代码进行修改

> 实际开发中, 后端接口的设计需要经过多方评审检查(review). 在接口设计时就会考虑格式化的统一化,尽可能的避免返工
>
> 当前是学习阶段, 给大家讲了这个接口设计的演变过程

5.1 登录页面

登录界面没有拦截, 只是返回结果发生了变化, 所以只需要根据返回结果修改对应代码即可

登录结果代码修改

```
function login() {
    $.ajax({
        type: "post",
        url: "/user/login",
        data: {
            name: $("#userName").val(),
            password: $("#password").val()
        },
        success: function (result) {
            console.log(result);
            if (result.status=="SUCCESS" && result.data==true) {
                location.href = "book_list.html";
            } else {
                alert("账号或密码不正确!");
            }
        }
    });
}
```

5.2 图书列表

针对图书列表页有两处变化

1. 拦截器进行了强制登录校验, 如果校验失败, 则http状态码返回401, 此时会走ajax的error逻辑处理
2. 接口返回结果发生了变化

图书列表代码修改:

```
function getBookList() {
    $.ajax({
        type: "get",
        url: "/book/getListByPage" + location.search,
        success: function (result) {
            //真实前端代码需要分的更细一点, 此处不做判断
```

```
            if (result == null || result.data == null) {
                location.href = "login.html";
                return;
            }

            var finalHtml = "";
            var data = result.data;
            for (var book of data.records) {
                //...代码省略
            }
        },
        error: function (error) {
            if (error != null && error.status == 401) {
                //用户未登录
                location.href="login.html";
            }
        }
    });
}
```

5.3 其他

参考图书列表, 对删除图书, 批量删除图书,添加图书, 修改图书接口添加用户强制登录以及统一格式返回的逻辑处理

1. 删除图书

```
function deleteBook(id) {
    //...代码省略
    success: function (result) {
        if(result.status=="SUCCESS" || result.data==""){
            //重新刷新页面
            location.href = "book_list.html"
        }else{
            alert(result.data);
        }
    },
    error: function (error) {
        if (error != null && error.status == 401) {
            //用户未登录
            location.href = "login.html";
        }
    }
    //...代码省略
}
```

2. 批量删除图书

```
function batchDelete() {
    //...代码省略
    success: function (result) {
        if (result.status=="SUCCESS" || result.data==true) {
            alert("删除成功");
            //重新刷新页面
            location.href = "book_list.html"
        }
    },
    error: function (error) {
        if (error != null && error.status == 401) {
            //用户未登录
            location.href = "login.html";
        }
    }
    //...代码省略
}
```

3. 添加图书

```
function add() {
    //...代码省略
    success: function (result) {
        console.log(result);
        console.log(result.data);
        if (result.status == "SUCCESS" && result.data == "") {
            location.href = "book_list.html"
        } else {
            console.log(result);
            alert("添加失败:" + result.data);
        }
    },
    error: function (error) {
        if (error != null && error.status == 401) {
            //用户未登录
            alert("用户未登录");
            location.href = "login.html";
        }
    }
    //...代码省略
}
```

4. 获取图书详情

```
$.ajax({
     //...代码省略
    success: function (result) {
        if (result.status == "SUCCESS" && result.data != null) {
            var book = result.data;
            if (book != null) {
                 //...代码省略
            }
        }

    },
    error: function (error) {
        if (error != null && error.status == 401) {
            //用户未登录
            alert("用户未登录");
            location.href = "login.html";
        }
    }
});
```

5. 修改图书

```
function update() {
        //...代码省略
        success: function (result) {
            if (result.status == "SUCCESS" || result.data == "") {
                location.href = "book_list.html"
            } else {
                console.log(result);
                alert("修改失败:" + result.data);
            }
        },
        error: function (error) {
            if (error != null && error.status == 401) {
                //用户未登录
                location.href = "login.html";
            }
        }
        //...代码省略
}
```

总结

本章节主要介绍了SpringBoot 对一些统一功能的处理支持.

1. 拦截器的实现主要分两部分: 1. 定义拦截器(实现HandlerInterceptor 接口) 2. 配置拦截器
2. 统一数据返回格式通过@ControllerAdvice + ResponseBodyAdvice 来实现
3. 统一异常处理使用@ControllerAdvice + @ExceptionHandler 来实现, 并且可以分异常来处理
4. 学习了DispatcherServlet的一些源码.