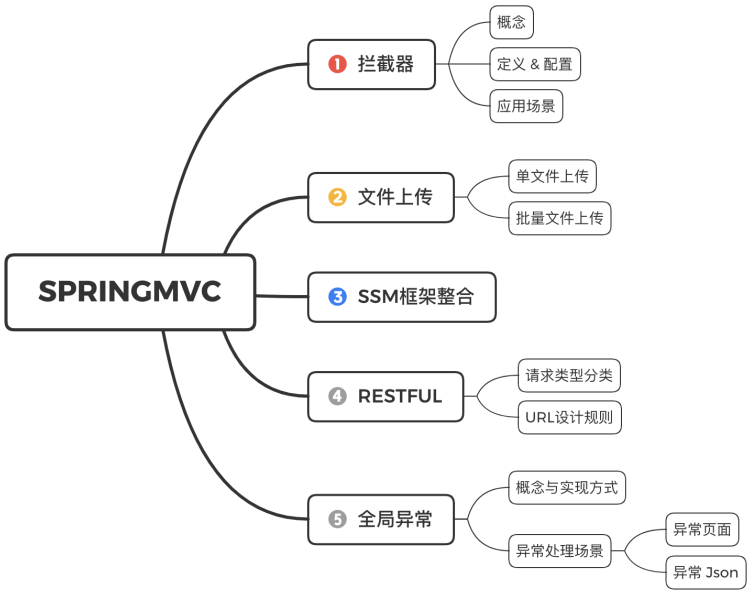
SpringMVC - 第二天

1. 学习目标



2. 拦截器

2.1. 基本概念

SpringMVC 中的 Interceptor 拦截器也是相当重要和相当有用的，它的主要作用是拦截用户的请求并进行相应的处理。比如通过它来进行权限验证，或者是来判断用户是否登陆等操作。对于 SpringMVC 拦截器的定义方式有两种：

实现接口：org.springframework.web.servlet.HandlerInterceptor

继承适配器：org.springframework.web.servlet.handler.HandlerInterceptorAdapter

2.2. 拦截器实现

2.2.1. 实现 HandlerInterceptor 接口

- 接口实现类

```
/**
 * 拦截器实现
 *       实现 HandlerInterceptor 接口
 */
public class MyInterceptor01 implements HandlerInterceptor {

    /**
     * 在 目标Handler(方法)执行前 执行
     *       返回 true：执行handler方法
     *       返回 false：阻止目标handler方法执行
     * @param request
```

```
     * @param response
     * @param handler
     * @return
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("目标Handler执行前执行MyInterceptor01 --> preHandle方
法...");
        /**
         * 返回 true：执行handler方法
         * 返回 false：阻止目标handler方法执行
         */
        return true;
    }

    /**
     * 在 目标Handler(方法)执行后，视图生成前 执行
     * @param request
     * @param response
     * @param handler
     * @param modelAndView
     * @throws Exception
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("目标Handler执行后，视图生成前执行MyInterceptor01 -->
postHandle方法...");
    }

    /**
     * 在 目标Handler(方法)执行后，视图生成后 执行
     * @param request
     * @param response
     * @param handler
     * @param ex
     * @throws Exception
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("目标Handler执行后，视图生成后执行MyInterceptor01 -->
afterCompletion方法...");
    }
}
```

- 拦截器xml配置

```
<!-- 拦截器配置：方式一 -->
<mvc:interceptors>
    <!--
        使用bean定义一个Interceptor
        直接定义在mvc:interceptors根下面的Interceptor将拦截所有的请求
    -->
    <bean class="com.xxxx.springmvc.interceptor.MyInterceptor01"/>
</mvc:interceptors>
```

```
<!-- 拦截器配置：方式二 (推荐使用) -->
<mvc:interceptors>
    <!--
        定义在 mvc:interceptor 下面，可以自定义需要拦截的请求
        如果有多个拦截器满足拦截处理的要求，则依据配置的先后顺序来执行
    -->
    <mvc:interceptor>
        <!-- 通过 mvc:mapping 配置需要拦截的资源。支持通配符。可配置多个。 -->
        <mvc:mapping path="/**"/> <!-- "/**"表示拦截所有的请求。 -->
        <!-- 通过 mvc:mapping 配置不需要被拦截的资源。支持通配符。可配置多个。 -->
        <mvc:exclude-mapping path="/url/*"/>  <!-- "/url/*"表示放行url路径下的请求。
-->
        <bean class="com.xxxx.springmvc.interceptor.MyInterceptor01"/>
    </mvc:interceptor>
</mvc:interceptors>
```

2.2.2. 继承 HandlerInterceptorAdapter

实际上最终还是 HandlerInterceptor 接口实现。

- 子类实现类

```
/**
 * 拦截器实现
 *      继承 HandlerInterceptorAdapter 适配器
 */
public class MyInterceptor02 extends HandlerInterceptorAdapter {

    /**
     * 在 目标Handler(方法)执行前 执行
     *      返回 true：执行handler方法
     *      返回 false：阻止目标handler方法执行
     * @param request
     * @param response
     * @param handler
     * @return
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

        System.out.println("目标Handler执行前执行MyInterceptor02 --> preHandle方
法...");
        /**
```

```
            * 返回 true：执行handler方法
            * 返回 false：阻止目标handler方法执行
            */
           return true;
       }
}
```

- 拦截器xml配置

```
<mvc:interceptors>
  <mvc:interceptor>
    <!-- 拦截的资源 -->
    <mvc:mapping path="/**"/>
    <!-- 放行的资源 -->
    <mvc:exclude-mapping path="/url/test01"/>
    <mvc:exclude-mapping path="/url/test02"/>
    <bean class="com.xxxx.springmvc.interceptor.MyInterceptor02"/>
  </mvc:interceptor>
</mvc:interceptors>
```

2.2.3. 多个拦截器实现

SpringMVC 框架支持多个拦截器配置，从而构成拦截器链，对客户端请求进行多次拦截操作。

- 拦截器代码实现

  这里参考MyInterceptor01、MyInterceptor02代码

- 拦截器xml配置

```
<mvc:interceptors>
  <!--
      拦截器链（多个拦截器）
          如果有多个拦截器满足拦截处理的要求，则依据配置的先后顺序来执行
          先配置的拦截器的 preHandle 方法先执行
          先配置的拦截器的 postHandle、afterCompletion 方法后执行
  -->
  <mvc:interceptor>
    <!-- 拦截所有请求 -->
    <mvc:mapping path="/**" />
    <bean class="com.xxxx.springmvc.interceptor.MyInterceptor01" />
  </mvc:interceptor>
  <mvc:interceptor>
    <!-- 拦截所有请求 -->
    <mvc:mapping path="/**" />
    <bean class="com.xxxx.springmvc.interceptor.MyInterceptor02" />
  </mvc:interceptor>
</mvc:interceptors>
```

2.3. 拦截器应用 - 非法请求拦截

使用拦截器完成用户是否登录请求验证功能

2.3.1. 用户控制器

UserInfoController 定义

```
/**
 * 用户操作模拟实现
 *       用户登录（无需登录）
 *       用户添加（需要登录）
 *       用户修改（需要登录）
 *       用户删除（需要登录）
 */
@Controller
@RequestMapping("/userInfo")
public class UserInfoController {

    /**
     * 用户登录
     * @return
     */
    @RequestMapping("/login")
    public ModelAndView userLogin(HttpSession session){
        System.out.println("用户登录...");
        ModelAndView mv = new ModelAndView();
        // 设置视图
        mv.setViewName("success");

        // 用户登录后，设置对应的session域对象
        User user = new User();
        user.setId(1);
        user.setUserName("admin");
        user.setUserPwd("123456");
        session.setAttribute("user",user);

        return mv;
    }

    /**
     * 用户添加
     * @return
     */
    @RequestMapping("/add")
    public ModelAndView userAdd(){
        System.out.println("用户添加...");
        ModelAndView mv = new ModelAndView();
        // 设置视图
        mv.setViewName("success");

        return mv;
    }

    /**
     * 用户修改
     * @return
```

```
     */
    @RequestMapping("/update")
    public ModelAndView userUpdate(){
        System.out.println("用户更新...");
        ModelAndView mv = new ModelAndView();
        // 设置视图
        mv.setViewName("success");

        return mv;
    }

    /**
     * 用户删除
     * @return
     */
    @RequestMapping("/delete")
    public ModelAndView userDelete(){
        System.out.println("用户删除...");
        ModelAndView mv = new ModelAndView();
        // 设置视图
        mv.setViewName("success");

        return mv;
    }

}
```

2.3.2. 页面定义

success.jsp 定义

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h3>欢迎登录！</h3>
</body>
</html>
```

2.3.3. 非法请求拦截器定义

LoginInterceptor 定义

```
/**
 * 非法访问拦截
 */
public class LoginInterceptor extends HandlerInterceptorAdapter {

    /**
     * 在 目标方法执行前 执行
     * @param request
     * @param response
     * @param handler
     * @return
```

```
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response, Object handler) throws
Exception {
        // 获取 session 域对象中的user对象
        User user= (User) request.getSession().getAttribute("user");
        // 判断session域对象中的 user 是否为空
        if(null == user){ // 如果为空，表示用户未登录
            // 拦截用户跳转到登录页面
            response.sendRedirect(request.getContextPath() + "/login.jsp");
            // 不执行目标方法
            return false;
        }
        // 用户已登录，执行目标方法
        return true;
    }
}
```

2.3.4. 拦截器xml配置

servlet-context.xml 配置

```
<!-- 拦截所有请求 -->
<mvc:interceptors>
    <mvc:interceptor>
        <!-- 拦截所有请求 -->
        <mvc:mapping path="/**" />
        <!-- 放行用户登录请求 -->
        <mvc:exclude-mapping path="/userInfo/login"/>
        <!--目标拦截器-->
        <bean class="com.xxxx.springmvc.interceptor.LoginInterceptor" />
    </mvc:interceptor>
</mvc:interceptors>
```

3. 文件上传

3.1. 环境配置

3.1.1. pom.xml文件修改

```
<!-- 添加 commons-fileupload 依赖 -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.2</version>
</dependency>
```

3.1.2. servlet-context.xml修改

```
<!-- 文件上传 -->
<bean id="multipartResolver"

 class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 允许文件上传的最大尺寸 -->
    <property name="maxUploadSize">
        <value>104857600</value>
    </property>
    <!--
        设置文件放入临时文件夹的最大大小限制。
        此值是阈值，低于此值，则保存在内存中，如高于此值，则生成硬盘上的临时文件。
    -->
    <property name="maxInMemorySize">
        <value>4096</value>
    </property>
</bean>
```

3.2. 代码实现

3.2.1. 单文件上传

3.2.1.1. 页面表单

- input 的type设置为file
- form 表单的method设为post,
- form 表单的enctype设置为multipart/form-data，以二进制的形式传输数据

```
<form action="uploadFile" method="post" enctype="multipart/form-data">
    <input type="file" name="file" />
    <button type="submit"> 提交</button>
</form>
```

3.2.1.2. 代码实现

```
/**
 * 文件上传
 */
@Controller
public class FileController {

    /**
     * 单文件上传
     *      使用MultipartFile对象作为参数，接收前端发送过来的文件
     * @param file
     * @param request
     * @return
     */
    @RequestMapping("/uploadFile")
    public String uploadFile(@RequestParam("file") MultipartFile file123435,
                             HttpServletRequest request){

        // 判断文件是否为空，如果不为空进行对应的文件上传操作
```

```
        if (!file.isEmpty()) {
            try {
                // 获取项目的所在的路径 （绝对路径）
                String path = request.getServletContext().getRealPath("/");
                // 设置上传的文件存放的目录
                File uploadFile = new File(path + "/upload");
                // 判断文件目录是否存在，不存在则新建目录
                if (!uploadFile.exists()) {
                    // 新建目录
                    uploadFile.mkdir();
                }
                // 获取上传文件的原文件名
                String originalName = file.getOriginalFilename();
                // 获取上传的文件的后缀
                String suffix =
originalName.substring(originalName.lastIndexOf("."));
                // 通过系统当前时间的毫秒数，生成随机文件名 （避免上传的文件名重复）
                String fileName = System.currentTimeMillis() + suffix;
                // 上传文件 (转存文件到指定目录)
                file.transferTo(new File(uploadFile, fileName));

                // 设置成功的域对象
                request.setAttribute("msg","文件上传成功！");

            } catch (IOException e) {
                e.printStackTrace();
                //  如果报错，设置的域对象
                request.setAttribute("msg","文件上传失败！");
            }

        } else {
            // 上传文件不存在，设置的域对象
            request.setAttribute("msg","文件不存在，上传失败！");
        }

        return "result";
    }
}
```

3.2.2. 多文件上传

3.2.2.1. 页面表单

```
<form action="uploadFiles" method="post" enctype="multipart/form-data">
  <input type="file" name="files" />
  <input type="file" name="files" />
  <input type="file" name="files" />
  <button type="submit"> 提交</button>
</form>
```

3.2.2.2. 代码实现

```
/**
  * 上传文件
  * @param file
  * @param request
  */
```

```java
public void saveFile(MultipartFile file, HttpServletRequest request) {
  // 判断文件是否为空，如果不为空进行对应的文件上传操作
  if (!file.isEmpty()) {
    try {
      // 获取项目的所在的路径 （绝对路径）
      String path = request.getServletContext().getRealPath("/");
      // 设置上传的文件存放的目录
      File uploadFile = new File(path + "/upload");
      // 判断文件目录是否存在，不存在则新建目录
      if (!uploadFile.exists()) {
        // 新建目录
        uploadFile.mkdir();
      }
      // 获取上传文件的原文件名
      String originalName = file.getOriginalFilename();
      // 获取上传的文件的后缀
      String suffix = originalName.substring(originalName.lastIndexOf("."));
      // 通过系统当前时间的毫秒数，生成随机文件名 （避免上传的文件名重复）
      String fileName = System.currentTimeMillis() + suffix;
      // 上传文件 (转存文件到指定目录)
      file.transferTo(new File(uploadFile, fileName));

      // 设置成功的域对象
      request.setAttribute("msg","文件上传成功！");

    } catch (IOException e) {
      e.printStackTrace();
      //  如果报错，设置的域对象
      request.setAttribute("msg","文件上传失败！");
      response.senRedirect("error.jsp");
    }

  } else {
    // 上传文件不存在，设置的域对象
    request.setAttribute("msg","文件不存在，上传失败！");
  }
}

/**
  * 多文件上传
  * @param files
  * @param request
  * @return
  */
@RequestMapping("/uploadFiles")
public String uploadFiles(@RequestParam("files") List<MultipartFile> files,
                          HttpServletRequest request) {
  // 判断文件集合是否为空
  if (files != null && files.size() > 0) {
    // 循环上传
    for (MultipartFile file:files) {
      // 上传文件
      saveFile(file, request);
    }
  }
  return "result";
}
```

4. SSM 框架集成与测试

4.1. 环境配置

4.1.1. IDEA 下创建项目

创建Maven对应的Web项目

4.1.2. 配置 pom.xml

4.1.2.1. 修改 JDK 版本

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
```

4.1.2.2. 添加坐标依赖

```
<dependencies>
    <!-- junit 测试 -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>

    <!-- spring 核心jar -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.2.4.RELEASE</version>
    </dependency>

    <!-- spring 测试jar -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>5.2.4.RELEASE</version>
    </dependency>

    <!-- spring jdbc -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.2.4.RELEASE</version>
    </dependency>

    <!-- spring事务 -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>5.2.4.RELEASE</version>
```

```
    </dependency>

    <!-- aspectj切面编程的jar -->
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.9.5</version>
    </dependency>


    <!-- c3p0 连接池 -->
    <dependency>
      <groupId>com.mchange</groupId>
      <artifactId>c3p0</artifactId>
      <version>0.9.5.2</version>
    </dependency>

    <!-- mybatis -->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis</artifactId>
      <version>3.5.3</version>
    </dependency>

    <!-- 添加mybatis与Spring整合的核心包 -->
    <dependency>
      <groupId>org.mybatis</groupId>
      <artifactId>mybatis-spring</artifactId>
      <version>2.0.3</version>
    </dependency>

    <!-- mysql 驱动包 -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.19</version>
    </dependency>

    <!-- 日志打印相关的jar -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.2</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.2</version>
    </dependency>

    <!-- 分页插件 -->
    <dependency>
      <groupId>com.github.pagehelper</groupId>
      <artifactId>pagehelper</artifactId>
      <version>5.1.10</version>
    </dependency>

    <!-- spring web -->
```

```
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>5.2.4.RELEASE</version>
    </dependency>

    <!-- spring mvc -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.2.4.RELEASE</version>
    </dependency>

    <!-- web servlet -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.0.1</version>
      <scope>provided</scope>
    </dependency>

    <!-- 添加json 依赖jar包（注意版本问题） -->
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-core</artifactId>
      <version>2.10.0</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.10.0</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-annotations</artifactId>
      <version>2.10.0</version>
    </dependency>

    <!-- commons-fileupload -->
    <dependency>
      <groupId>commons-fileupload</groupId>
      <artifactId>commons-fileupload</artifactId>
      <version>1.3.2</version>
    </dependency>

</dependencies>
```

4.1.2.3. 设置资源目录和插件

```
  <build>
    <finalName>ssm</finalName>

    <!--
      Maven 项目:如果源代码(src/main/java)存在xml、properties、tld 等文件
      Maven 默认不会自动编译该文件到输出目录,如果要编译源代码中xml properties tld 等文件
      需要显式配置 resources 标签
   -->
```

```
    <resources>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
      <resource>
        <directory>src/main/java</directory>
        <includes>
          <include>**/*.xml</include>
          <include>**/*.properties</include>
          <include>**/*.tld</include>
        </includes>
        <filtering>false</filtering>
      </resource>
    </resources>

    <plugins>
      <!-- 编译环境插件 -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>11</source>
          <target>11</target>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
      <!-- jetty插件 -->
      <plugin>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>9.4.27.v20200227</version>
        <configuration>
          <scanIntervalSeconds>10</scanIntervalSeconds>
          <!-- 设置端口 -->
          <httpConnector>
            <port>8080</port>
          </httpConnector>
          <!-- 设置项目路径 -->
          <webAppConfig>
            <contextPath>/ssm</contextPath>
          </webAppConfig>
        </configuration>
      </plugin>
    </plugins>

  </build>
```

4.1.3. 配置 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="3.0"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
```

```
  <!-- 启动spring容器-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring.xml</param-value>
  </context-param>
  <!-- 设置监听器 -->
  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!-- 编码过滤 utf-8 -->
  <filter>
    <description>char encoding filter</description>
    <filter-name>encodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <!-- servlet请求分发器 -->
  <servlet>
    <servlet-name>springMvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:servlet-context.xml</param-value>
    </init-param>
    <!-- 表示启动容器时初始化该Servlet -->
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>springMvc</servlet-name>
    <!-- 这是拦截请求，  "/"代表拦截所有请求，"*.do"拦截所有.do请求 -->
     <url-pattern>/</url-pattern>
    <!--<url-pattern>*.do</url-pattern>-->
  </servlet-mapping>
</web-app>
```

4.1.4. 配置 servlet-context.xml

在项目的 src/main/resources 下创建 servlet-context.xml 文件， 内容如下

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
        http://www.springframework.org/schema/mvc
```

```
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 开启扫描器 -->
    <context:component-scan base-package="com.xxxx.ssm.controller" />

    <!-- mvc 注解驱动 并添加json 支持 -->
    <mvc:annotation-driven>
        <mvc:message-converters>
            <!-- 返回信息为字符串时 处理 -->
            <bean
class="org.springframework.http.converter.StringHttpMessageConverter"/>
            <!-- 将对象转换为json 对象 -->
            <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConvert
er"/>
        </mvc:message-converters>
    </mvc:annotation-driven>


    <!-- 使用默认的 Servlet 来响应静态文件 -->
    <mvc:default-servlet-handler/>


    <!-- 配置视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          id="internalResourceViewResolver">
        <!-- 前缀：在WEB-INF目录下的jsp目录下 -->
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!-- 后缀：以.jsp结尾的资源 -->
        <property name="suffix" value=".jsp"/>
    </bean>


    <!-- 文件上传 -->
    <bean id="multipartResolver"

 class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!-- 允许文件上传的最大尺寸 -->
        <property name="maxUploadSize">
            <value>104857600</value>
        </property>
        <!--
          设置文件放入临时文件夹的最大大小限制。
          此值是阈值，低于此值，则保存在内存中，如高于此值，则生成硬盘上的临时文件。
        -->
        <property name="maxInMemorySize">
            <value>4096</value>
        </property>
    </bean>

</beans>
```

4.1.5. 配置 spring.xml

在项目的 src/main/resources 下创建 spring.xml 文件， 内容如下

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!-- 扫描基本包 -->
    <context:component-scan base-package="com.xxxx.ssm" >
        <!--  context:exclude-filter标签：排除对某个注解的扫描 （过滤controller层） -->
        <context:exclude-filter type="annotation"

 expression="org.springframework.stereotype.Controller" />
    </context:component-scan>

    <!-- 加载properties 配置文件 -->
    <context:property-placeholder location="classpath:db.properties" />

    <!-- aop -->
    <aop:aspectj-autoproxy />

    <!-- 配置c3p0 数据源 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${jdbc.driver}"></property>
        <property name="jdbcUrl" value="${jdbc.url}"></property>
        <property name="user" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>

    <!-- 配置事务管理器 -->
    <bean id="txManager"

 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!-- 设置事物增强 -->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="add*" propagation="REQUIRED" />
            <tx:method name="insert*" propagation="REQUIRED" />
            <tx:method name="update*" propagation="REQUIRED" />
            <tx:method name="delete*" propagation="REQUIRED" />
        </tx:attributes>
    </tx:advice>
```

```
    <!-- aop 切面配置 -->
    <aop:config>
        <aop:pointcut id="servicePointcut"
                      expression="execution(* com.xxxx.ssm.service..*.*(..))" />
        <aop:advisor advice-ref="txAdvice" pointcut-ref="servicePointcut" />
    </aop:config>

    <!-- 配置 sqlSessionFactory -->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        <property name="configLocation" value="classpath:mybatis.xml" />
        <property name="mapperLocations"
value="classpath:com/xxxx/ssm/mapper/*.xml" />
    </bean>

    <!-- 配置扫描器 -->
    <bean id="mapperScanner"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!-- 扫描com.xxxx.ssm.dao这个包以及它的子包下的所有映射接口类 -->
        <property name="basePackage" value="com.xxxx.ssm.dao" />
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />
    </bean>
</beans>
```

4.1.6. 配置 mybatis.xml

在项目的 src/main/resources 下创建 mybatis.xml 文件， 内容如下

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <typeAliases>
        <package name="com.xxxx.ssm.po"/>
    </typeAliases>
    <plugins>
        <plugin interceptor="com.github.pagehelper.PageInterceptor"></plugin>
    </plugins>
</configuration>
```

4.1.7. 配置 db.properties

在项目的 src/main/resources 下创建 db.properties 文件，内容如下(mysql 创建数据库ssm)

```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/ssm?
useUnicode=true&characterEncoding=utf8&serverTimezone=GMT%2B8&useSSL=false
jdbc.username=root
jdbc.password=root
```

4.1.8. 添加 log4j.properties

在项目的 src/main/resources 下创建 log4j.properties 文件，内容如下

```
log4j.rootLogger=DEBUG, Console
# Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n
log4j.logger.java.sql.ResultSet=INFO
log4j.logger.org.apache=INFO
log4j.logger.java.sql.Connection=DEBUG
log4j.logger.java.sql.Statement=DEBUG
log4j.logger.java.sql.PreparedStatement=DEBUG
```

4.2. 添加源代码

4.2.1. 添加包

在项目的 src/main/java 下创建对应的包结构

com.xxxx.ssm.controller

com.xxxx.ssm.service

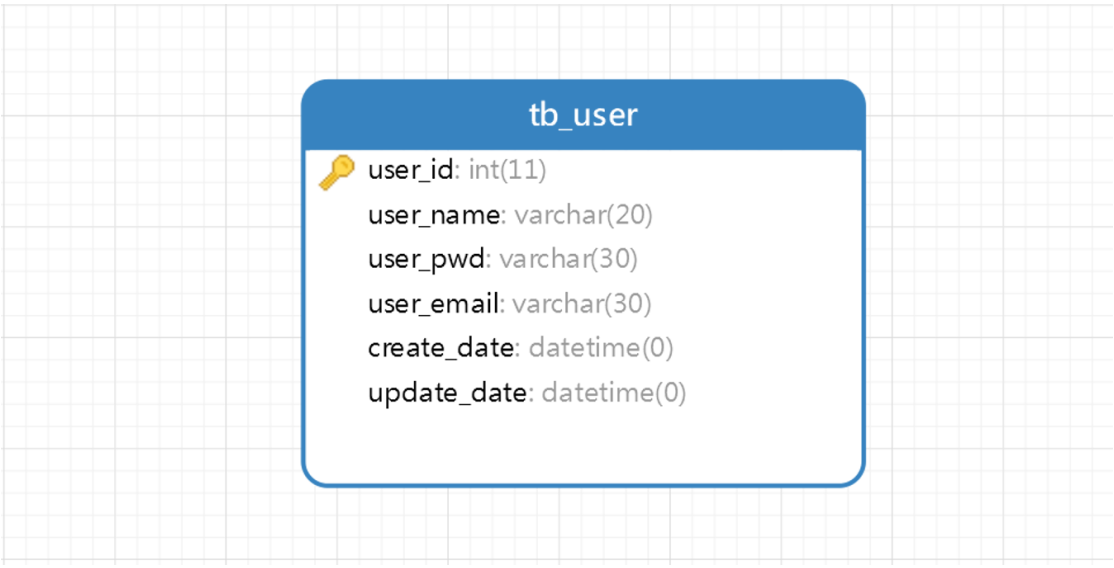
com.xxxx.ssm.mapper

com.xxxx.ssm.dao

com.xxxx.ssm.po

4.2.2. 添加 User.java

在 com.xxxx.ssm.po 包下创建 JavaBean 文件 User.java (数据库字段对应如下)



```
public class User {

    private Integer userId;
    private String userName;
    private String userPwd;
    private String userEmail;
    private Date createDate;
    private Date updateDate;

    /**
      set get 方法省略
    **/
}
```

4.2.3. 添加UserDao.java 接口

com.xxxx.ssm.dao 包下创建 UserDao.java 文件，提供对应的用户详情查询功能

```
public interface UserDao {
    User queryUserByUserId(Integer userId);
}
```

4.2.4. 添加UserMapper.xml 映射文件

com.xxxx.ssm.mapper 包下创建 UserMapper.xml 文件，提供select 查询标签配置

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.xxxx.ssm.dao.UserDao">
      <select id="queryUserByUserId" parameterType="int"
resultType="com.xxxx.ssm.po.User">
           select user_id as userId,user_name as userName,user_pwd as userPwd
           from tb_user
           where user_id = #{userId}
      </select>
</mapper>
```

4.2.5. 添加 UserService.java

com.xxxx.ssm.service 包下创建UserService.java 文件，提供用户详情查询方法

```
@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    public User queryUserByUserId(Integer userId){
        return userDao.queryUserByUserId(userId);
    }
}
```

4.2.6. 添加 HelloController.java

在 com.xxxx.ssm.controller 包下创建 HelloController.java 文件

```
@Controller
public class HelloController {
    // 注入userService
    @Autowired
    private UserService userService;

    @RequestMapping("/hello")
    public ModelAndView hello(){
        ModelAndView mv = new ModelAndView();
        // 调用service 层查询方法
        User user = userService.queryUserByUserId(1);
        mv.addObject("user", user);
        mv.setViewName("hello");
        return mv;
    }
}
```

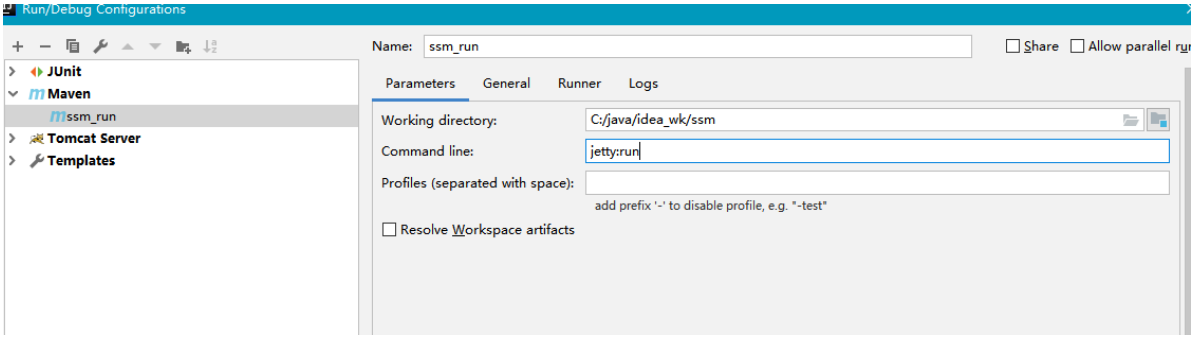
4.2.7. 添加 hello.jsp 视图文件

在src/main/webapp/WEB-INF 创建jsp 目录，并在该目下创建hello.jsp ， 展示查询的用户信息

```
<body>
    欢迎你，${user.userName}
</body>
```

4.3. 执行测试

4.3.1. Idea 下配置 jetty 启动命令



4.3.2. 启动 jetty 浏览器访问http://localhost:8080/ssm/hello查看结果

5. RestFul URL

5.1. 基本概念

模型 - 视图 - 控制器（MVC）是一个众所周知的以设计界面应用程序为基础的设计思想。

Restful 风格的 API 是一种软件架构风格，设计风格而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

在Restful 风格中，用户请求的 url 使用同一个 url，用请求方式：get，post，delete，put...等方式对请求的处理方法进行区分，这样可以在前后台分离式的开发中使得前端开发人员不会对请求的资源地址产生混淆和大量的检查方法名的麻烦，形成一个统一的接口。

在 Restful 风格中，现有规定如下：

- GET（SELECT）：从服务器查询，可以在服务器通过请求的参数区分查询的方式。
- POST（CREATE）：在服务器端新建一个资源，调用 insert 操作。
- PUT（UPDATE）：在服务器端更新资源，调用 update 操作。
- PATCH（UPDATE）：在服务器端更新资源（客户端提供改变的属性）。(目前 jdk7 未实现，tomcat7 不支持)。
- DELETE（DELETE）：从服务器端删除资源，调用 delete 语句。

5.2. SpringMVC 支持 RestFul URL 风格设计

案例：如何使用 Java 构造没有扩展名的RESTful url，如 /forms/1?

SpringMVC 是通过 @RequestMapping 及 @PathVariable 注解提供的。

通过如@RequestMapping(value="/blog /{id}", method = RequestMethod.DELETE)，即可处理 /blog/1 的 delete 请求。

5.3. RestFul URL 映射地址配置

5.3.1. 准备环境

5.3.1.1. 添加 Account

在 src/resources/java 对应的 com.xxxx.ssm.po 目录下新建 Account.java 实体类

```
public class Account {
    private Integer accountId;
    private String accountName;
    private String accountType;
    private Double money;
    private Integer userId;
    private Date createDate;
    private Date updateDate;
    private String remark;

    /* get set 方法省略 */
}
```

5.3.1.2. 添加 AccountDao

在 src/resources/java 对应的 com.xxxx.ssm.dao 目录下新建 AccountDao.java 接口类

```
public interface AccountDao {

    public Account selectById(Integer id);

    public int save(Account account);

    public int update(Account account);

    public int delete(Integer id);

}
```

5.3.1.3. 添加 AccountMapper

在 src/resources/java 对应的 com.xxxx.ssm.mapper 目录下新建 AccountMapper.xml 映射文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.xxxx.ssm.dao.AccountDao" >
  <resultMap id="BaseResultMap" type="com.xxxx.ssm.po.Account" >
    <id column="id" property="id" jdbcType="INTEGER" />
    <result column="aname" property="aname" jdbcType="VARCHAR" />
    <result column="type" property="type" jdbcType="VARCHAR" />
    <result column="money" property="money" jdbcType="DOUBLE" />
    <result column="user_id" property="userId" jdbcType="INTEGER" />
    <result column="create_time" property="createTime" jdbcType="DATE" />
    <result column="update_time" property="updateTime" jdbcType="DATE" />
    <result column="remark" property="remark" jdbcType="VARCHAR" />
  </resultMap>
  <sql id="Base_Column_List" >
    id, aname, type, money, user_id, create_time, update_time, remark
  </sql>
  <!-- 查询操作 -->
  <select id="selectById" resultMap="BaseResultMap"
parameterType="java.lang.Integer" >
    select
    <include refid="Base_Column_List" />
    from tb_account
    where id = #{id,jdbcType=INTEGER}
  </select>

  <!-- 删除操作 -->
  <delete id="delete" parameterType="java.lang.Integer" >
    delete from tb_account
    where id = #{id,jdbcType=INTEGER}
  </delete>

  <!-- 添加操作 -->
  <insert id="save" parameterType="com.xxxx.ssm.po.Account" >
    insert into tb_account
    <trim prefix="(" suffix=")" suffixOverrides="," >
      <if test="id != null" >
```

```
        id,
      </if>
      <if test="aname != null" >
        aname,
      </if>
      <if test="type != null" >
        type,
      </if>
      <if test="money != null" >
        money,
      </if>
      <if test="userId != null" >
        user_id,
      </if>
      <if test="createTime != null" >
        create_time,
      </if>
      <if test="updateTime != null" >
        update_time,
      </if>
      <if test="remark != null" >
        remark,
      </if>
    </trim>
    <trim prefix="values (" suffix=")" suffixOverrides="," >
      <if test="id != null" >
        #{id,jdbcType=INTEGER},
      </if>
      <if test="aname != null" >
        #{aname,jdbcType=VARCHAR},
      </if>
      <if test="type != null" >
        #{type,jdbcType=VARCHAR},
      </if>
      <if test="money != null" >
        #{money,jdbcType=DOUBLE},
      </if>
      <if test="userId != null" >
        #{userId,jdbcType=INTEGER},
      </if>
      <if test="createTime != null" >
        #{createTime,jdbcType=DATE},
      </if>
      <if test="updateTime != null" >
        #{updateTime,jdbcType=DATE},
      </if>
      <if test="remark != null" >
        #{remark,jdbcType=VARCHAR},
      </if>
    </trim>
  </insert>

  <!-- 更新操作 -->
  <update id="update" parameterType="com.xxxx.ssm.po.Account" >
    update tb_account
    <set >
      <if test="aname != null" >
        aname = #{aname,jdbcType=VARCHAR},
```

```
      </if>
      <if test="type != null" >
        type = #{type,jdbcType=VARCHAR},
      </if>
      <if test="money != null" >
        money = #{money,jdbcType=DOUBLE},
      </if>
      <if test="userId != null" >
        user_id = #{userId,jdbcType=INTEGER},
      </if>
      <if test="createTime != null" >
        create_time = #{createTime,jdbcType=DATE},
      </if>
      <if test="updateTime != null" >
        update_time = #{updateTime,jdbcType=DATE},
      </if>
      <if test="remark != null" >
        remark = #{remark,jdbcType=VARCHAR},
      </if>
    </set>
    where id = #{id,jdbcType=INTEGER}
  </update>
</mapper>
```

5.3.1.4. 添加 AccountService

在 src/resources/java 对应的 com.xxxx.ssm.service 目录下新建 AccountService.java

```
@Service
public class AccountService {

    @Autowired
    private AccountDao accountDao;

    public  Account selectById(Integer id){
        return accountDao.selectById(id);
    }

    public int saveAccount(Account account){
        return accountDao.save(account);
    }

    public int updateAccount(Account account){
        return accountDao.update(account);
    }

    public int delAccount(Integer id){
        return accountDao.delete(id);
    }

}
```

5.3.2. URL 映射地址配置

5.3.2.1. Get 请求配置

```
/**
 * restful --> get 请求，执行查询操作
 * @param id
 * @return
 */
@GetMapping("account/{id}")
@ResponseBody
public Account queryAccountById(@PathVariable Integer id){
    return accountService.selectById(id);
}
```

5.3.2.2. Delete 请求配置

```
/* restful-->delete 请求 执行删除操作
 * @param id
 * @return
 */
@DeleteMapping("account/{id}")
@ResponseBody
public Map<String,Object> deleteAccount(@PathVariable Integer id){
    int result = accountService.delAccount(id);

    Map<String,Object> map=new HashMap<String,Object>();
    if(result == 1 ){
        map.put("msg","success");
        map.put("code",200);
    } else {
        map.put("msg","error");
        map.put("code",500);
    }
    return map;
}
```

5.3.2.3. Post 请求配置

```
/* restful --> post 请求，执行添加操作
 * @return
 */
@PostMapping("account")
@ResponseBody
public Map<String,Object> saveAccount(@RequestBody  Account account){
    int result = accountService.saveAccount(account);
    Map<String,Object> map=new HashMap<String,Object>();
    if(result == 1 ){
        map.put("msg","success");
        map.put("code",200);
    } else {
        map.put("msg","error");
        map.put("code",500);
    }
    return map;
}
```

5.3.2.4. Put 请求配置

```
/* restful-->put 请求执行更新操作
 * @param id
 * @param account
 * @return
 */
@PutMapping("account")
@ResponseBody
public Map<String,Object> updateAccount(@RequestBody  Account account){
    int result = accountService.updateAccount(account);
    Map<String,Object> map=new HashMap<String,Object>();
    if(result == 1 ){
        map.put("msg","success");
        map.put("code",200);
    } else {
        map.put("msg","error");
        map.put("code",500);
    }
    return map;
}
```

6. 全局异常统一处理

6.1. 全局异常概念

在 JavaEE 项目的开发中，不管是对底层的数据库操作过程，还是业务层的处理过程，还是控制层的处理过程，都不可避免会遇到各种可预知的、不可预知的异常需要处理。每个过程都单独处理异常，系统的代码耦合度高，工作量大且不好统一，维护的工作量也很大。

SpringMVC 对于异常处理这块提供了支持，通过 SpringMVC 提供的全局异常处理机制，能够将所有类型的异常处理从各处理过程解耦出来，既保证了相关处理过程的功能较单一，也实现了异常信息的统一处理和维护。

全局异常实现方式 Spring MVC 处理异常有 3 种方式：

1. 使用 Spring MVC 提供的简单异常处理器 SimpleMappingExceptionResolver
2. 实现 Spring 的异常处理接口 HandlerExceptionResolver 自定义自己的异常处理器
3. 使用 @ExceptionHandler 注解实现异常处理

6.2. 异常处理实现

6.2.1. 全局异常处理方式一

6.2.1.1. 配置简单异常处理器

配置 SimpleMappingExceptionResolver 对象

```
<!-- 配置全局异常统一处理的 Bean （简单异常处理器） -->
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
  <!-- 页面在转发时出现异常，设置默认的错误页面 （error代表的是一个视图） -->
  <property name="defaultErrorView" value="error"></property>
  <!-- 异常发生时，设置异常的变量名 -->
  <property name="exceptionAttribute" value="ex"></property>
</bean>
```

可以在处理异常的页面获取异常信息

```
${ex}
```

6.2.1.2. 使用自定义异常

参数异常

```
/**
 * 自定义异常：参数异常
 */
public class ParamsException extends RuntimeException {
    private Integer code = 300;
    private String msg = "参数异常!";


    public ParamsException() {
        super("参数异常!");
    }

    public ParamsException(String msg) {
        super(msg);
        this.msg = msg;
    }

    public ParamsException(Integer code) {
        super("参数异常!");
        this.code = code;
    }

    public ParamsException(Integer code, String msg) {
        super(msg);
        this.code = code;
        this.msg = msg;
    }

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }

    public String getMsg() {
        return msg;
    }
```

```
    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

业务异常

```
/**
 * 自定义异常：业务异常
 */
public class BusinessException extends RuntimeException {
    private Integer code=400;
    private String msg="业务异常!";


    public BusinessException() {
        super("业务异常!");
    }

    public BusinessException(String msg) {
        super(msg);
        this.msg = msg;
    }

    public BusinessException(Integer code) {
        super("业务异常!");
        this.code = code;
    }

    public BusinessException(Integer code, String msg) {
        super(msg);
        this.code = code;
        this.msg = msg;
    }

    public Integer getCode() {
        return code;
    }

    public void setCode(Integer code) {
        this.code = code;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

6.2.1.3. 设置自定义异常与页面的映射

```
<!-- 设置自定义异常与页面的映射  -->
<property name="exceptionMappings">
    <props>
        <!-- key：自定义异常对象的路径；  标签中设置具体的处理页面的视图名-->
        <prop key="com.xxxx.ssm.exception.BusinessException">buss_error</prop>
        <prop key="com.xxxx.ssm.exception.ParamsException">params_error</prop>
    </props>
</property>
```

使用 SimpleMappingExceptionResolver 进行异常处理，具有集成简单、有良好的扩展性、对已有代码没有入侵性等优点，但该方法仅能获取到异常信息，若在出现异常时，对需要获取除异常以外的数据的情况不适用。

6.2.2. 全局异常处理方式二(推荐)

6.2.2.1. 实现 HandlerExceptionResolver 接口

```
/**
 * 全局异常统一处理
 */
@Component
public class GlobalExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object handler, Exception ex) {
        ModelAndView mv = new ModelAndView("error");
        mv.addObject("ex","默认错误信息");
        return mv;
    }
}
```

6.2.2.2. 自定义异常处理

```
/**
 * 全局异常统一处理
 */
@Component
public class GlobalExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse, Object handler, Exception ex) {
        ModelAndView mv = new ModelAndView("error");
        mv.addObject("ex","默认错误信息");

        // 判断是否是自定义异常
        if (ex instanceof ParamsException) {
            mv.setViewName("params_error");
            ParamsException e = (ParamsException) ex;
            mv.addObject("ex", e.getMsg());
        }

        if (ex instanceof BusinessException) {
```

```
            mv.setViewName("business_error");
            BusinessException e = (BusinessException) ex;
            mv.addObject("ex", e.getMsg());
        }
        return mv;
    }
}
```

使用实现 HandlerExceptionResolver 接口的异常处理器进行异常处理，具有集成简单、有良好的扩展性、对已有代码没有入侵性等优点，同时，在异常处理时能获取导致出现异常的对象，有利于提供更详细的异常处理信息。

6.2.3. 全局异常处理方式三

页面处理器继承 BaseController

```
public class BaseController {
    @ExceptionHandler
    public String exc(HttpServletRequest request,HttpServletResponse
response,Exception ex){
        request.setAttribute("ex", ex);
        if(ex instanceof ParamsException){
            return "error_param";
        }
        if(ex instanceof BusinessException){
            return "error_business";
        }
        return "error";
    }
}
```

使用 @ExceptionHandler 注解实现异常处理，具有集成简单、有扩展性好（只需要将要异常处理的 Controller 类继承于 BaseController 即可）、不需要附加Spring 配置等优点，但该方法对已有代码存在入侵性(需要修改已有代码，使相关类继承于 BaseController)，在异常处理时不能获取除异常以外的数据。

6.3. 未捕获异常的处理

对于 Unchecked Exception 而言，由于代码不强制捕获，往往被忽略，如果运行期产生了 Unchecked Exception，而代码中又没有进行相应的捕获和处理，则我们可能不得不面对尴尬的 404、500……等服务器内部错误提示页面。

此时需要一个全面而有效的异常处理机制。目前大多数服务器也都支持在 web.xml 中通过(Websphere/Weblogic)或者(Tomcat)节点配置特定异常情况的显示页面。修改 web.xml 文件，增加以下内容：

```
<!-- 出错页面定义 -->
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/500.jsp</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/500.jsp</location>
</error-page>
<error-page>
    <error-code>404</error-code>
    <location>/404.jsp</location>
</error-page>
```