* 学习目标

 * 能够掌握SpringMVC实现restful风格

 * 能够掌握SpringMVC响应数据和结果视图

 * 能够掌握SpringMVC文件上传

 * 能够掌握SpringMVC的拦截器

 * 能够掌握SpringMVC异常处理

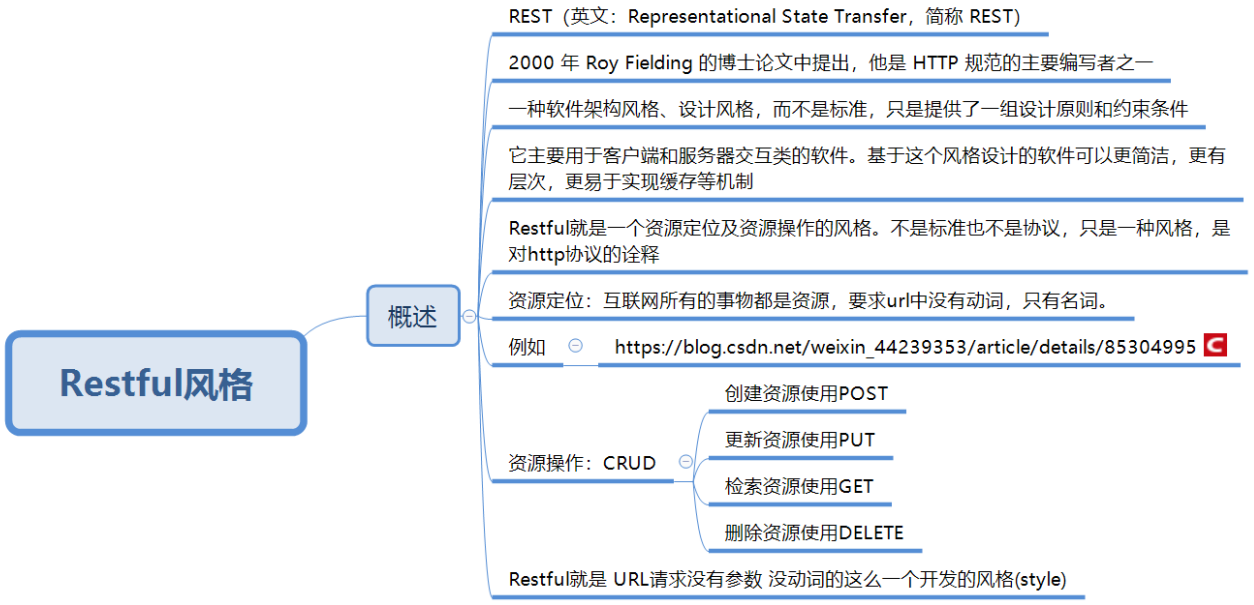
 * 能够了解SpringMVC与Struts2区别

---

 * 回顾

  * ModleAttribute

* 能够掌握SpringMVC实现restful风格

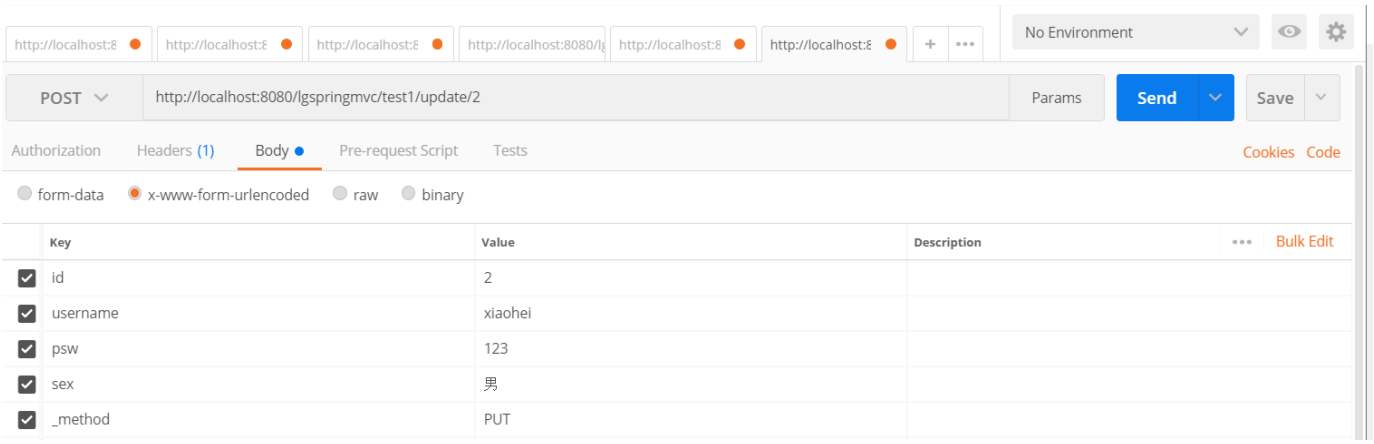


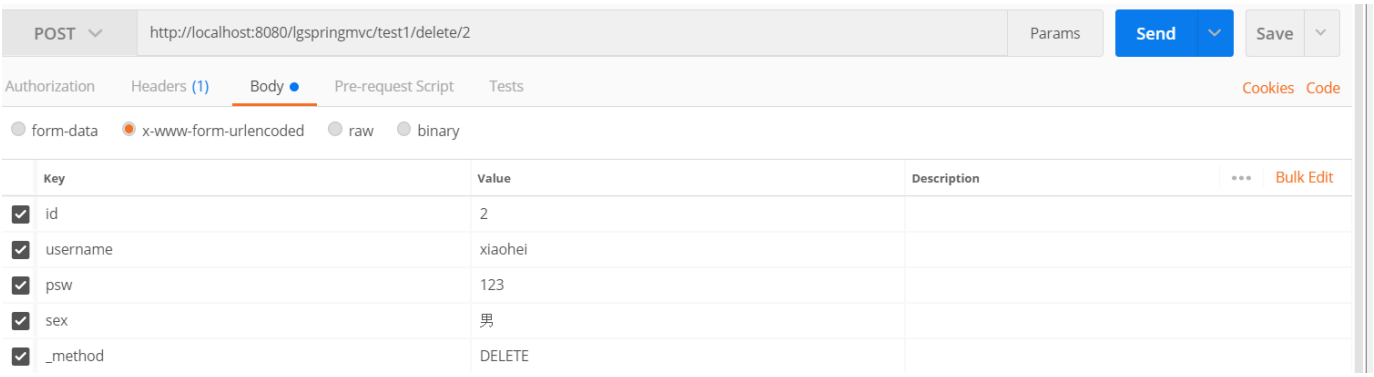

```
* 案例
@Controller
```

```java
@RequestMapping("/test1")
public class RestFulController {

    @PostMapping("/insert")
    public String testPost(User user){
        System.out.println(user);
        return "success";
    }
    @GetMapping("/get/{id}")
    public String testGet(@PathVariable("id") Integer id){
        System.out.println("Get:"+id);
        return "success";
    }

    @PutMapping("/update/{id}")
    public String testPut(@PathVariable("id") Integer id,User user){
        System.out.println(id+":"+user);
        return "success";
    }
    @DeleteMapping("/delete/{id}")
    public String testDelete(@PathVariable("id") Integer id){
        System.out.println("Delete:"+id);
        return "success";
    }
}

* 单元测试
  @Test
    public void test13() throws Exception {
        String result = mockMvc.perform(post("/test1/insert")
                .param("id","2")
                .param("username","xiaohei")
                .param("psw","123")
                .param("sex","男")).
                andExpect(status().isOk()).
                andDo(print()).andReturn().getResponse().getContentAsString();
        System.out.println(result);
    }

    @Test
```

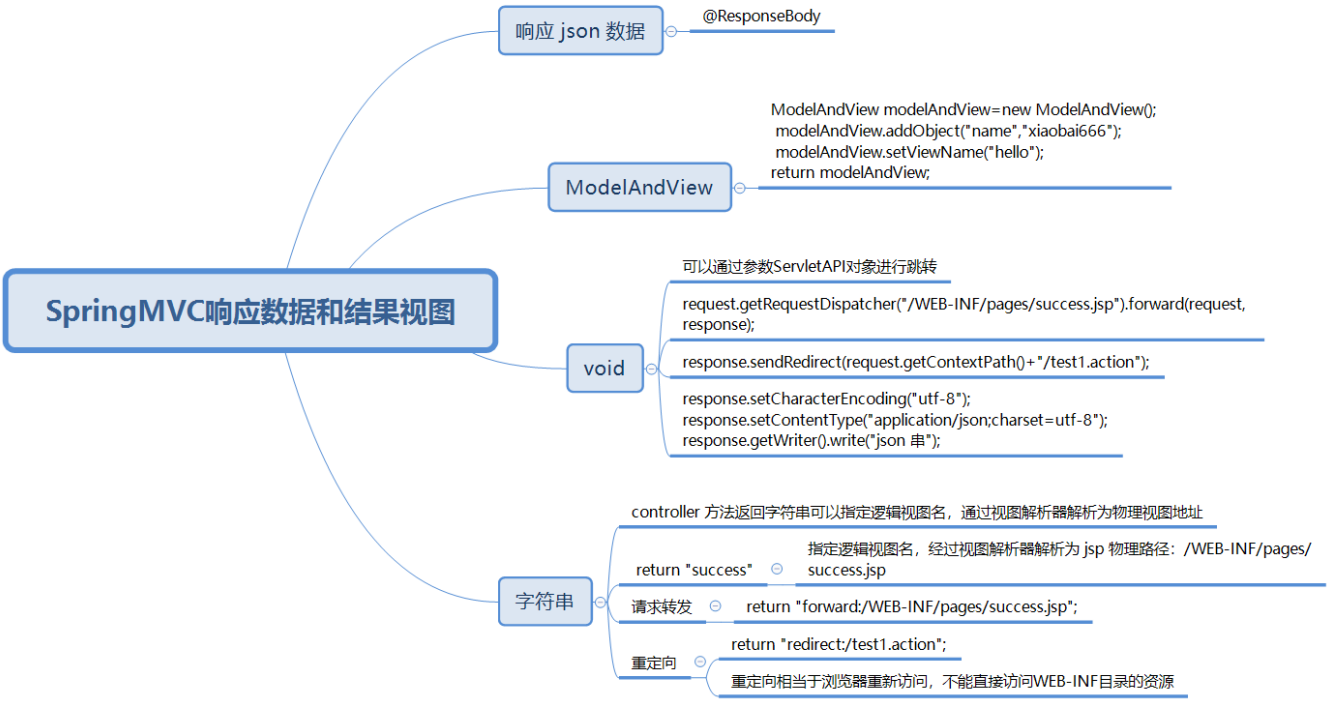
```
    public void test14() throws Exception {
        String result = mockMvc.perform(get("/test1/get/2")).
                andExpect(status().isOk()).
                andDo(print()).andReturn().getResponse().getContentAsString();
        System.out.println(result);
    }

    @Test
    public void test15() throws Exception {
        String result = mockMvc.perform(put("/test1/update/2")
                .param("id","2")
                .param("username","xiaohei")
                .param("psw","123")
                .param("sex","男")).
                andExpect(status().isOk()).
                andDo(print()).andReturn().getResponse().getContentAsString();
        System.out.println(result);
    }

    @Test
    public void test16() throws Exception {
        String result = mockMvc.perform(delete("/test1/delete/8")).
                andExpect(status().isOk()).
                andDo(print()).andReturn().getResponse().getContentAsString();
        System.out.println(result);
    }
* 测试结果
  * 例如/test1/get/2，需要获取2的参数值，需要加上注解@PathVariable，否则获取参数值为

* 再通过postman测试
  * 发现测试put的时候，值设置的不完全
  * 原因：form 表单只支持 GET 与 POST 请求，而 DELETE、PUT 等 method 并不支持，Spri
    加了一个过滤器，可以将浏览器请求改为指定的请求方式，发送给我们的控制器方法，使得支
   与 DELETE 请求
  * PUT和DELETE没办法跳转视图但是返回json是可以的

* 添加过滤器
 <filter>
    <filter-name>hiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter
```

```
83    </filter>
84    <filter-mapping>
85      <filter-name>hiddenHttpMethodFilter</filter-name>
86      <url-pattern>/*</url-pattern>
87    </filter-mapping>
88  * 查看过滤器的源码
89      * 发现post请求，添加_method=PUT|DELETE
90
```





* 能够掌握SpringMVC响应数据和结果视图




```
@RequestMapping("/test10")
public void test10(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
   request.getRequestDispatcher("/WEB-INF/jsps/hello.jsp").forward(request,resp
}
@RequestMapping("/test11")
public String test11(){
   return "forward:/WEB-INF/jsps/hello.jsp";
 }
@RequestMapping("/test12")
public void test12(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
  response.sendRedirect(request.getContextPath()+"/test1.action");
}
@RequestMapping("/test13")
public String test13(){
   return "redirect:/test1.action";
}

* 浏览器进行测试
```

* 能够掌握SpringMVC文件上传

* 复习文件上传：10-文件上传

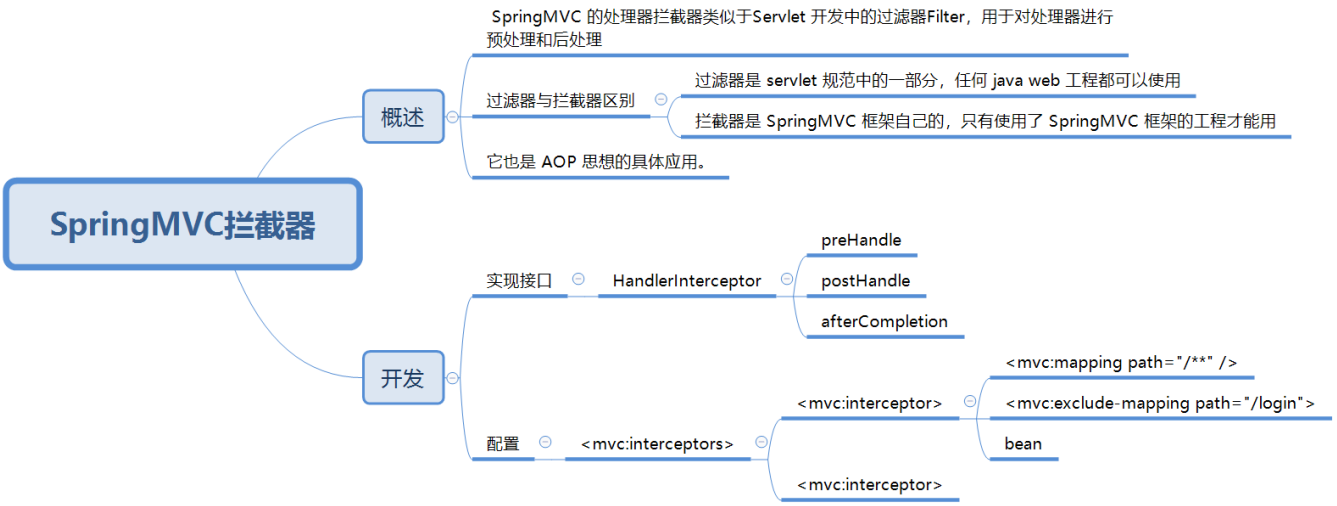
```
* 案例一：（传到应用服务器）
* 添加依赖
 <dependency>
   <groupId>commons-fileupload</groupId>
   <artifactId>commons-fileupload</artifactId>
   <version>1.3.3</version>
 </dependency>
* 页面
<%@ page contentType="text/html;charset=UTF-8" language="java" isELIgnored="fal
<html>
<head>
    <title>测试</title>
</head>
<body>
<form action="${pageContext.request.contextPath}/fileUpload"
    method="post" enctype="multipart/form-data">
    图片：<input type="file" name="uploadFile"/><br/>
    <input type="submit" value="上传"/>
</form>
</body>
</html>

<%@ page contentType="text/html;charset=UTF-8" language="java" isELIgnored="fal
<html>
<body>
<h2>文件${result}</h2>
</body>
</html>

* 代码
@Controller
public class UploadController {
    @RequestMapping("/fileUpload")
    public String fileUpload(MultipartFile uploadFile, Model model){
        try {
        //生成唯一的文件名(UUID)
        String filename = UUID.randomUUID().toString();
        System.out.println(filename);
        //获取原文件名：b.jpg
```

```
        String originalFilename = uploadFile.getOriginalFilename();
        System.out.println(originalFilename);
        //获取带.的文件后缀名
        String extName = originalFilename.substring(originalFilename.lastIndexO
        System.out.println(extName);
        //上传
        //指定上传的图片的路径
        File file= new File("D:\\wook666\\upload\\"+filename+extName);
        uploadFile.transferTo(file);
        model.addAttribute("result","上传成功");
        } catch (Exception e) {
            e.printStackTrace();
            model.addAttribute("result","上传失败");
        }
        return "uploadsuccess";
    }
}

* 配置
<bean id="multipartResolver" class="
    org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <!-- 设置上传文件的最大尺寸为5MB -->
  <property name="maxUploadSize">
     <value>5242880</value>
  </property>
</bean>

* 案例二：（传到文件服务器）
* 构建文件服务项目lgimage
   * 构建不同tomcat：端口为8082，其他的端口对应改一下
 * 在lgimage下webapp下面
   * 构建uploads文件夹
    * 温馨提醒：里面新建个文件空白文件，不然有时候打包没有打包uploads目前过去
 * tomcat默认是只读，修改tomcat的web.xml文件
  * 添加
   <init-param>
       <param-name>readonly</param-name>
       <param-value>false</param-value>
   </init-param>

```

```
* 在应用服务中
 * 添加依赖
 <dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.19.4</version>
 </dependency>
 * 代码
  public static final String FILE_SERVER_URL = "http://localhost:8082/lgimage/u
    @RequestMapping("/fileUpload")
    public String fileUpload2(MultipartFile uploadFile, Model model){
        try {
            //生成唯一的文件名(UUID)
            String filename = UUID.randomUUID().toString();
            System.out.println(filename);
            //获取原文件名
            String originalFilename = uploadFile.getOriginalFilename();
            System.out.println(originalFilename);
            //获取带.的文件后缀名
            String extName = originalFilename.substring(originalFilename.lastIr
            System.out.println(extName);
            //上传
           // 创建 sun 公司提供的 jersey 包中的 Client 对象
            Client client = Client.create();
            // 指定上传文件的地址，该地址是 web 路径
            WebResource resource = client.resource(FILE_SERVER_URL+filename+ext
            // 实现上传
            resource.put(uploadFile.getBytes());
            model.addAttribute("result","上传成功");
        } catch (Exception e) {
            e.printStackTrace();
            model.addAttribute("result","上传失败");
        }
        return "uploadsuccess";
    }
 * 访问进行测试：
  * http://localhost:8080/lgspringmvc/upload.jsp
```

* 能够掌握SpringMVC的拦截器

* 复习过滤器：07-Fitler&Listener




```
* 案例一:(一个拦截器)
public class LgHandlerInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        // 返回值:返回false，不执行handler，返回true执行handler
        System.out.println("LgHandlerInterceptor--preHandle");
        return true;
    }
    /**
     * 在业务处理器处理完请求后，但是 DispatcherServlet 向客户端返回响应前被调用
     */
    @Override
    public void postHandle(HttpServletRequest request,
       HttpServletResponse response,
       Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("LgHandlerInterceptor--postHandle");
    }
    /**
     * 在 DispatcherServlet 完全处理完请求后被调用
     * 可以在该方法中进行一些资源清理的操作。
     */
    @Override
    public void afterCompletion(HttpServletRequest request,
```

```
          HttpServletResponse response,
         Object handler, Exception ex) throws Exception {
        System.out.println("LgHandlerInterceptor--afterCompletion");
    }
}
* 配置
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**"/>
    <bean id="lgHandlerInterceptor" class="com.lg.interceptor.LgHandlerIntercep
  </mvc:interceptor>
</mvc:interceptors>
* 单元测试
 @Test
public void test1() throws Exception {
  mockMvc.perform(post("/user/testsa"));
}
* 结果
 * preHandle 返回值为true的执行结果
LgHandlerInterceptor--preHandle
handler执行...
LgHandlerInterceptor--postHandle
LgHandlerInterceptor--afterCompletion
 * preHandle返回值为false的执行结果（不会执行handler）
LgHandlerInterceptor--preHandle

* 案例二：两个拦截器
 * 在上面基础上再编写拦截器并配置
public class LgHandlerInterceptor2 implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {
        // 返回值:返回false，不执行handler，返回true执行handler
        System.out.println("LgHandlerInterceptor2--preHandle");
        return true;
    }
    /**
     * 在业务处理器处理完请求后，但是 DispatcherServlet 向客户端返回响应前被调用
     */
    @Override
```

```
    public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler,
    ModelAndView modelAndView) throws Exception {
        System.out.println("LgHandlerInterceptor2--postHandle");
    }

    /**
     * 在 DispatcherServlet 完全处理完请求后被调用
     * 可以在该方法中进行一些资源清理的操作。
     */
    @Override
    public void afterCompletion(HttpServletRequest
    request, HttpServletResponse response,
    Object handler, Exception ex) throws Exception {
        System.out.println("LgHandlerInterceptor2--afterCompletion");
    }
}
* 配置
 <mvc:interceptors>
   <mvc:interceptor>
     <mvc:mapping path="/**"/>
     <bean id="lgHandlerInterceptor" class="com.lg.interceptor.LgHandlerInterce
   </mvc:interceptor>
   <mvc:interceptor>
     <mvc:mapping path="/**"/>
     <bean id="lgHandlerInterceptor2" class="com.lg.interceptor.LgHandlerInterc
   </mvc:interceptor>
</mvc:interceptors>
* 单元测试
* 测试结果
 * 当拦截器1和2的preHandle返回值为true
   * 拦截器1preHandle-->拦截器2preHandle-->handler
     -->拦截器2postHandle-->拦截器1postHandle
     -->拦截器2afterCompletion-->拦截器1afterCompletion
LgHandlerInterceptor--preHandle
LgHandlerInterceptor2--preHandle
handler执行...
LgHandlerInterceptor2--postHandle
LgHandlerInterceptor--postHandle
LgHandlerInterceptor2--afterCompletion
```

```
LgHandlerInterceptor--afterCompletion
 * 当拦截器1的preHandle返回值为false
 LgHandlerInterceptor--preHandle
 * 拦截器1的preHandle返回值为true和拦截器2的preHandle返回值为false
  * 拦截器1preHandle-->拦截器2preHandle-->handler
   -->拦截器1afterCompletion
 LgHandlerInterceptor--preHandle
 LgHandlerInterceptor2--preHandle
 LgHandlerInterceptor--afterCompletion

* 案例三：
  * 如何用户的登录过就放行访问，如果不没有登录过，跳转到的登录页面。
  * 开发思路：
   * 登录的页面（url:login）不需要拦截
   * 其他的任何的url都应该由拦截器拦截。
     * 拦截器中的逻辑：（preHandle） 方法中做验证
      * 获取URL判断下：如果 是login就放行。
      * 如果不是登录的URL，
       * 判断：session是否有用户信息，
         * 如果有，已经登录，验证通过。
         * 如果没有，重定向到登录的页面
 * 前期准备
  * copy之前登录小米页面
   * 改成jsp放在/WEB-INF/jsps目录
   * 注意修改路径
  * 假如访问看不到css样式，图片，js之类
  !-- location 表示路径，mapping 表示文件，**表示该目录下的文件以及子目录的文件 -->
    <mvc:resources location="/css/" mapping="/css/**"/>
    <mvc:resources location="/image/" mapping="/image/**"/>
    <mvc:resources location="/js/" mapping="/js/**"/>
  * 测试http://localhost:8080/lgspringmvc/user/login
* 代码
@RequestMapping("/login")
public String login(){
   return "login";
 }
@RequestMapping(value = "/loginsubmit",method = RequestMethod.POST)
public String loginSubmit(HttpSession session,String username,String password){
        session.setAttribute("loginStatus","success");
        System.out.println(username+":"+password);
```

```
        System.out.println("登录成功...");
        return "redirect:/index.jsp";
}
public class LoginHandlerInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response,
            Object handler) throws Exception {
        System.out.println("启动登录拦截器...");
        Object loginStatus=request.getSession().getAttribute("loginStatus");
        //如果不是登录的URL，
        //判断：session是否有用户信息，
        //如果有，已经登录，验证通过。
        //如果没有重定向到登录的页面
        if(loginStatus!=null){
            return true;
        }else {
            response.sendRedirect(request.getContextPath()+"/user/login");
            return false;
        }
    }
}
* 配置
  <mvc:interceptor>
      <mvc:mapping path="/**"/>
      <mvc:exclude-mapping path="/user/login"/>
      <mvc:exclude-mapping path="/user/loginsubmit"/>
      <mvc:exclude-mapping path="/css/**"/>
      <mvc:exclude-mapping path="/image/**"/>
      <mvc:exclude-mapping path="/js/**"/>
      <bean id="loginHandlerInterceptor" class="com.lg.interceptor.LoginHandler
  </mvc:interceptor>

* 测试(测试Controller)
```

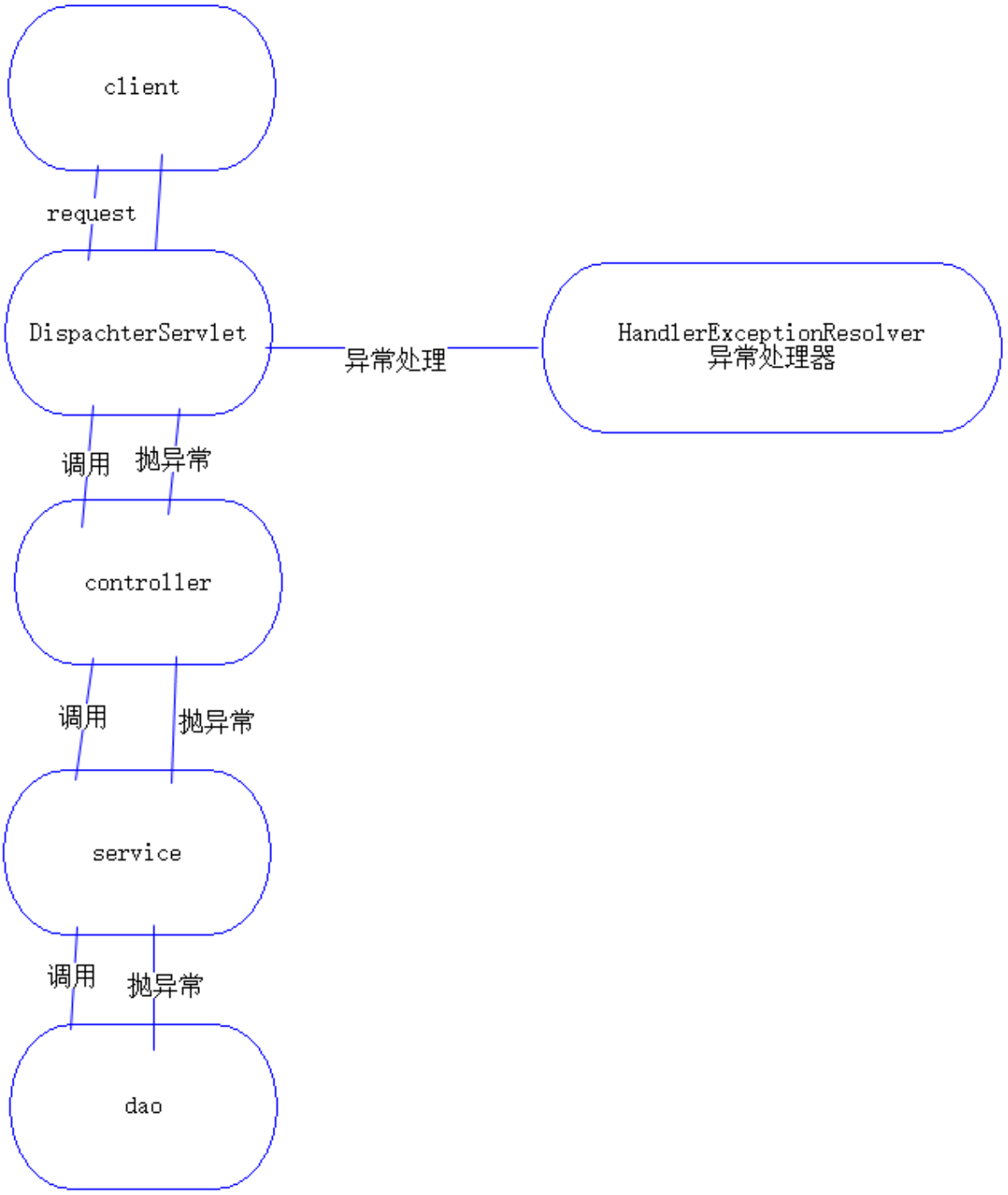
* 能够掌握SpringMVC异常处理

  * 复习异常：02-异常处理&Log4j

  * 异常处理思路

* 系统的dao、service、controller出现都通过throws Exception向上抛出，

最后由springmvc前端控制器交由异常处理器进行异常处理




```
* 案例一：（XML配置）
* 自定义异常代码
public class CustomException extends Exception {
    private String message;
    public CustomException(String message) {
        this.message = message;
    }
```

```
    public String getMessage() {
        return message;
    }
}
public class CustomHandlerExceptionResolver implements HandlerExceptionResolver
    @Override
    public ModelAndView resolveException(HttpServletRequest request, HttpServle
                                         Object handler, Exception ex) {
        ex.printStackTrace();
        CustomException customException = null;
        //如果抛出的是系统自定义异常则直接转换
        if (ex instanceof CustomException) {
            customException = (CustomException) ex;
        } else {
            //如果抛出的不是系统自定义异常则重新构造一个系统错误异常。
            customException = new CustomException("系统错误，请与系统管理 员联系!
        }
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("message", customException.getMessage());
        modelAndView.setViewName("error");
        return modelAndView;
    }
}
* 测试代码
@RequestMapping("testError1")
public void testError1(){
    int i=100/0;
}
@RequestMapping("testError2")
public void testError2() throws CustomException {
    throw new CustomException("用户不存在");
}
* 配置
<bean id="customHandlerExceptionResolver" class="com.lg.exception.CustomHandler
* 界面
 * 在/WEB-INF/jsps/ 新建error.jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" isELIgnored="fal
<html>
<head>
    <title>错误提示页面</title>
```

```
</head>
<body>
    <h1>${message}</h1>
    <img width="85%" height="85%" src="${pageContext.request.contextPath}/image
</body>
</html>
 * 在wepapp 下新建testError.jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" isELIgnored="fal
<html>
<head>
    <title>测试</title>
</head>
<body>
<a href="${pageContext.request.contextPath}/user/testError1">错误1</a>
<hr/>
<a href="${pageContext.request.contextPath}/user/testError2">错误2</a>
<hr/>
</body>
</html>
* 测试
 * http://localhost:8080/lgspringmvc/testError.jsp

* 案例二：(注解形式)
@ControllerAdvice
public class CustomHandlerExceptionResolver2 {

    @ExceptionHandler
    public ModelAndView resolveException(Exception ex) {
        ex.printStackTrace();
        CustomException customException = null;
        //如果抛出的是系统自定义异常则直接转换
        if (ex instanceof CustomException) {
            customException = (CustomException) ex;
        } else {
            //如果抛出的不是系统自定义异常则重新构造一个系统错误异常。
            customException = new CustomException("系统错误，请与系统管理 员联系!
        }
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("message", customException.getMessage());
        modelAndView.setViewName("error");
```

```
88          return modelAndView;
89      }
90  }
91
```

* 能够了解SpringMVC与Struts2区别