SpringMVC教程

#一、什么是SpringMVC

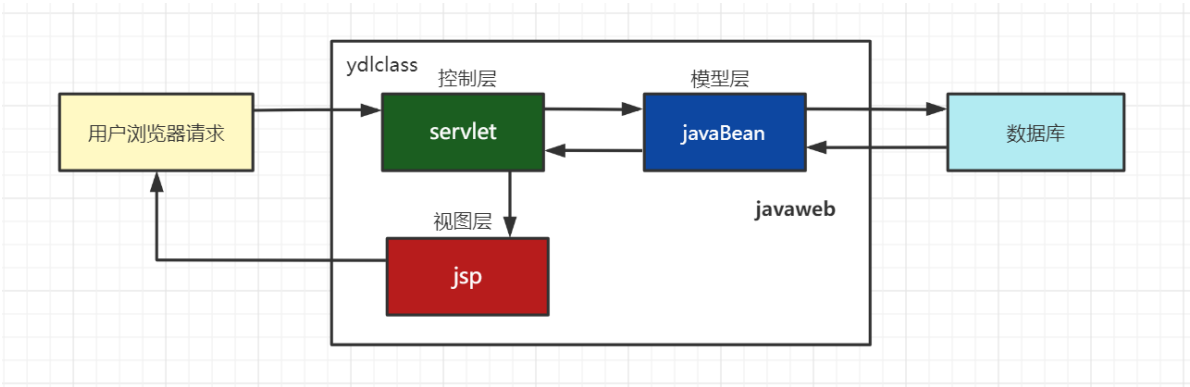
【Spring Web MVC】是最初建立在 Servlet API 之上的 Web 框架，从一开始就包含在【Spring Framework】中。正式名称【Spring Web MVC】来自其源模块的名称 (spring-webmvc)，但它更常被称为【Spring MVC】。

> MVC 设计概述

回顾mvc：

#1、mvc的发展历程

> 我们之前学习的mvc模式就是这种【Servlet + JSP + Java Bean】 模式，早期的 MVC 模型如下图所示：



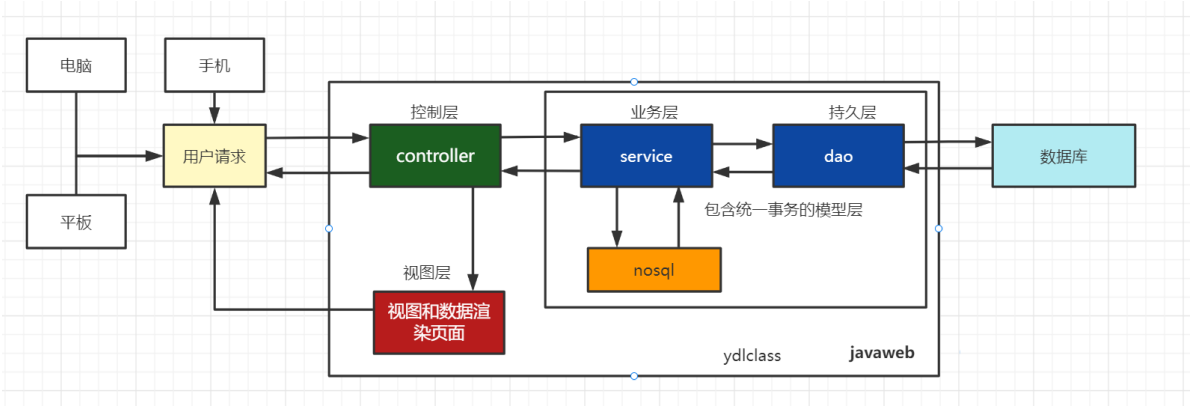
首先用户的请求会到达 Servlet，然后根据请求调用相应的 JavaBean，并把所有的显示结果交给 JSP 去完成，这样的模式我们就称为 MVC 模式：

- **M 代表 模型（Model）** 模型是什么呢？ 完成具体的业务，进行数据的查询。
- **V 代表 视图（View）** 视图是什么呢？ 就是用来做展示的，比如我们学过的JSP技术，用来展示模型中的数据。
- **C 代表 控制器（controller)**

  控制器是什么？ 控制器的作用就是搜集页面传来的原始数据，或者调用模型获得数据交给视图层处理，Servlet 扮演的就是这样的角色。

> Spring MVC 的架构

Spring MVC 给出了自己的mvc方案：



传统的模型层被拆分为了业务层（Service）和数据访问层（DAO,Data Access Object）。同时，在 Service 层下可以通过 Spring 的声明式事务操作数据访问层。

spring的mvc有以下特点：

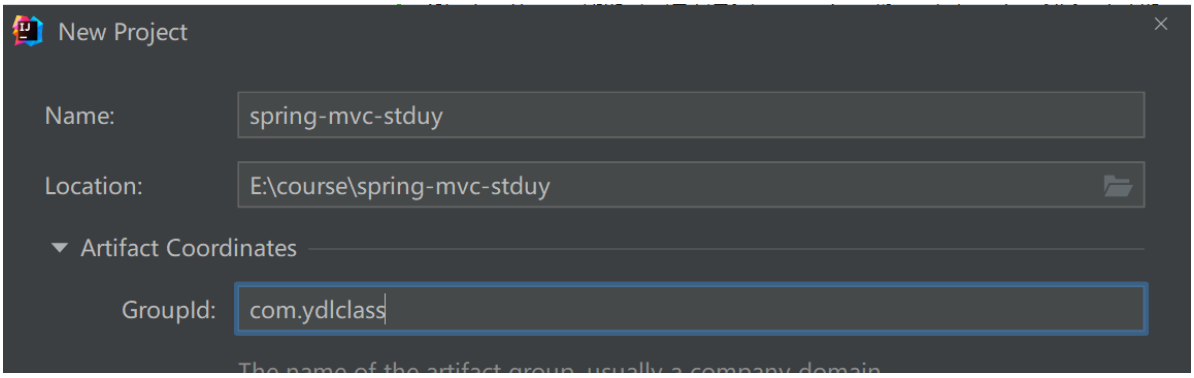
- 结构松散，几乎可以在 Spring MVC 中使用各类视图，不仅仅是jsp。
- 松耦合，各个模块分离
- 与 Spring 无缝集成

> 现在使用springmvc的公司越来越多，已经成为了霸主地位，基本上取代了早年的struts2，但是我们不能否仍依然有一些公司在使用老的框架，但是触类旁通，希望有大家可以去自行了解。

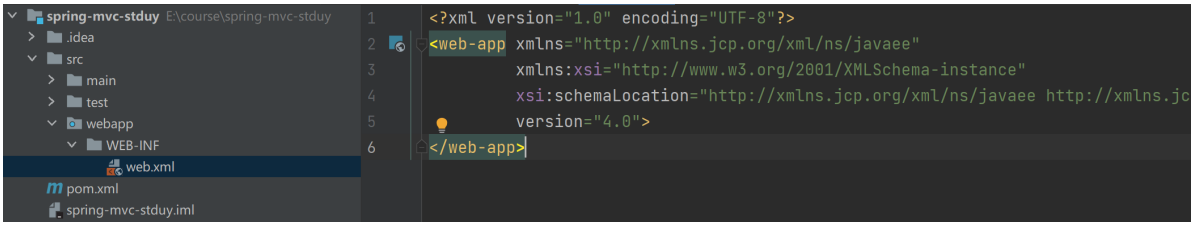
#二、直接上代码

#1、创建基础web工程

> 创建工程



> 完善一个webapp工程所必备的目录：

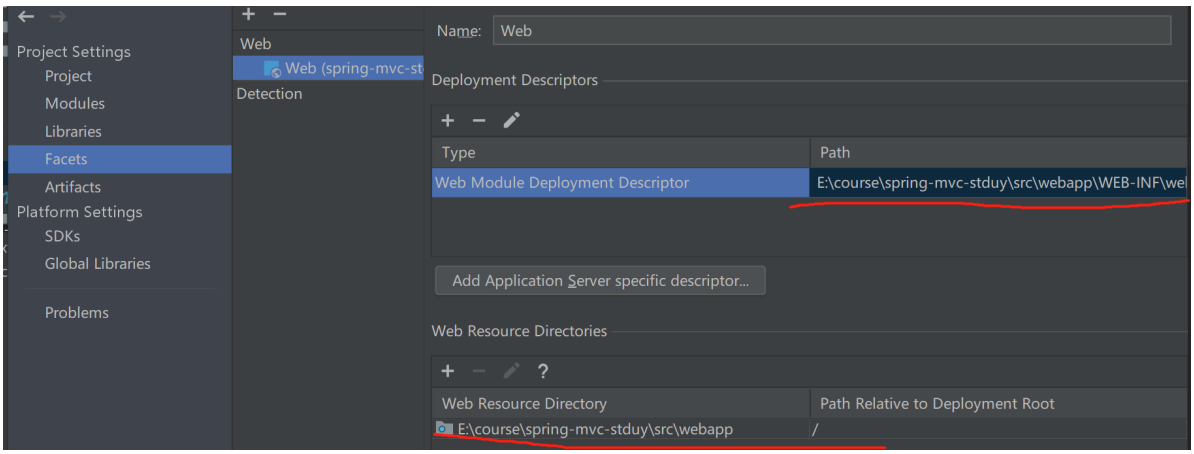


> 添加一个最小的必须依赖

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.ydlclass</groupId>
    <artifactId>spring-mvc-stduy</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <!--servlet api-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.1</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <target>${maven.compiler.target}</target>
                    <source>${maven.compiler.source}</source>
                    <encoding>utf-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

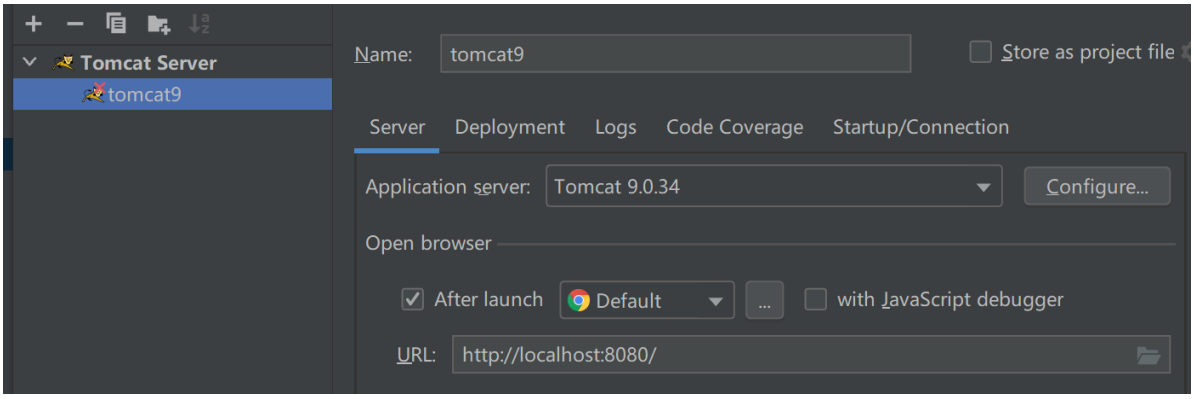
> 构建web项目:



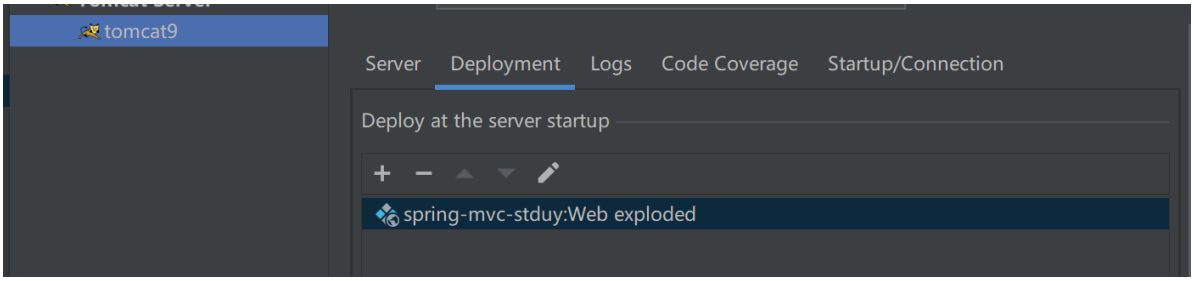
web.xml模板：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
</web-app>
```

配置tomcat，推荐使用tomcat9：



部署项目

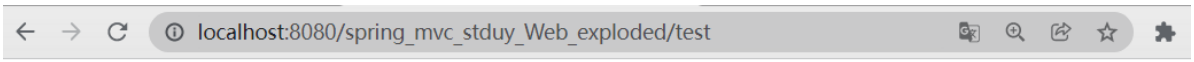


启动tomcat

写一个setvlet进行测试

```java
@WebServlet("/test")
public class TestServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
        try {
            resp.getWriter().println("hello servlet!!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

> 在浏览器中进行测试，web项目构建成功：



hello servlet!!

#2、搭建springmvc环境

（1）首先完整的pom

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.ydlclass</groupId>
    <artifactId>spring-mvc-stduy</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <!--servlet api-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.1</version>
            <scope>provided</scope>
        </dependency>
        <!--springmvc的依赖，会自动传递spring的其他依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.18.RELEASE</version>
        </dependency>
    </dependencies>
```

```
    <!--编译插件-->
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <target>${maven.compiler.target}</target>
                    <source>${maven.compiler.source}</source>
                    <encoding>utf-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

（2）配置web.xml

> 注册一个叫DispatcherServlet的servlet，这玩意是spring给我们提供的，我们先复制，后边会细讲：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <!--配置一个ContextLoaderListener，他会在servlet容器启动时帮我们初始化spring容器-->
    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!--指定启动spring容器的配置文件-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/app-context.xml</param-value>
    </context-param>

    <!--注册DispatcherServlet，这是springmvc的核心-->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>WEB-INF/app-context.xml</param-value>
        </init-param>
        <!--加载时先启动-->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--/ 匹配所有的请求；（不包括.jsp）-->
    <!--/* 匹配所有的请求；（包括.jsp）-->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
```

```
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

（3）编写配置文件

> 名称：app-context.xml (其实就是个spring和springmvc共享的配置文件)，我们可以建立在/WEB-INF/目录下：

小知识：在视图解析器中我们把所有的视图都存放在/WEB-INF/目录下，这样可以保证视图安全，因为这个目录下的文件，客户端不能直接访问，必须通过请求转发。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 处理映射器 -->
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
    <!-- 处理器适配器 -->
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
    <!-- 视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/page/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

（4）编写Controller

> 注意：这个实现了Controller接口的类需要返回一个ModelAndView，这个对象封装了视图和模型；

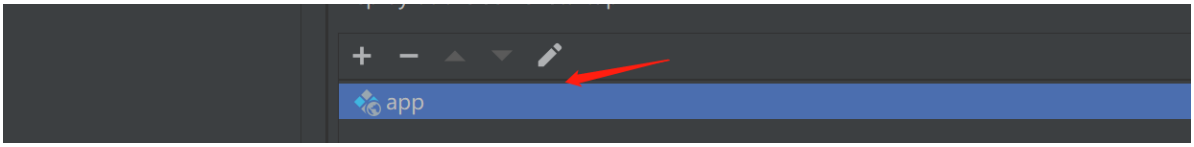
```
public class FirstController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse) throws Exception {
        // ModelAndView 封装了模型和视图
        ModelAndView mv = new ModelAndView();
        // 模型里封装数据
        mv.addObject("hellomvc","Hello springMVC!");
        // 封装跳转的视图名字
        mv.setViewName("hellomvc");
        // 不是有个视图解析器吗？
        // 这玩意可以自动给你加个前缀后缀，可以将hellomvc拼装成/jsp/hellomvc.jsp
        return mv;
    }
}
```

（5）注入容器

> 将这个实现了controller接口的bean注入到容器中，注意此时的id就成了你要访问的url了

```
    <bean id="/hellomvc" class="cn.itnanls.controller.FirstController"/>
```

> 我们可以给项目换一个简单的名字：



（6）创建jsp页面

> 使用el表达式获取模型中的数据

```
<body>
    ${hellomvc}
</body>
```

（7）配置Tomcat，并启动测试

#3、使用注解来一波

> 记住一点，只要用注解就得去扫包，让专业的负责解析的类来进行解析：

（1）配置文件

```
<?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:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       https://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/mvc
       https://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- 自动扫包 -->
    <context:component-scan base-package="com.ydlclass"/>
    <!-- 让Spring MVC不处理静态资源，负责静态资源也会走我们的前端控制器、试图解析器 -->
    <mvc:default-servlet-handler />
    <!--  让springmvc自带的注解生效  -->
    <mvc:annotation-driven />

     <!-- 处理映射器 -->
```

```
    <bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
    <!-- 处理器适配器 -->
    <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
    <!-- 视图解析器 -->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" id="internalResourceViewResolver">
        <!-- 前缀 -->
        <property name="prefix" value="/WEB-INF/page/" />
        <!-- 后缀 -->
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

（2）编写controller

```
@Controller
public class AnnotationController {

    @RequestMapping("/hello")
    public ModelAndView testAnnotation(){
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("hello","hello annotationMvc");
        modelAndView.setViewName("hello");
        return modelAndView;
    }

}
```

（3）启动tomcat测试

#三、初识springmvc

#1、组件说明

> DispatcherServlet：中央控制器，前端控制器

用户请求到达前端控制器（dispatcherServlet），他是整个流程控制的中心，由它负责调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

这玩意可以理解成一个【咨询处】，你去某个地方办事，先去咨询处问问我们应该先干什么，等第一件事做完了，可以接着去咨询处咨询，你的下一步工作应该是什么。

> handler：处理器

Handler也叫后端控制器，在DispatcherServlet的控制下Handler对【具体的用户请求】进行处理，由于Handler涉及到【具体的用户业务请求】，所以一般情况需要程序员【根据业务需求开发Handler】。这玩意就是你写的controller，别把他想成啥高级玩意，你也能写个处理器。

> View：视图

一般情况下，需要通过【页面标签或页面模版技术】将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。目前我们接触过得视图技术就是jsp，当然还有Freemarker，Thymeleaf等。

> HandlerMapping：处理器映射器

HandlerMapping负责根据【用户请求url】找到【Handler】即处理器，springmvc提供了不同的【处理器映射器】实现，如配置文件方式，实现接口方式，注解方式等。

> HandlAdapter：处理器适配器

HandlerAdapter负责调用具体的处理器，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。我们写的controller中的方法，将来就是会由处理器适配器调用。

> ViewResolver：视图解析器

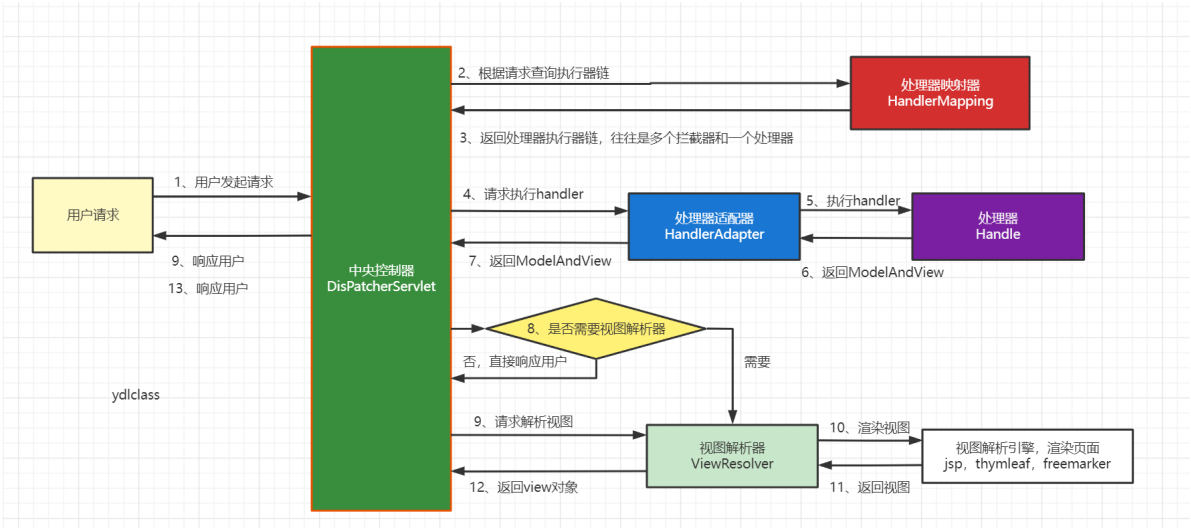
View Resolver负责将处理结果生成View视图，View Resolver首先根据【逻辑视图名】解析成【物理视图名】即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。

#2、执行流程

> Springmvc的是围绕DispatcherServlet进行设计的：

- DispatcherServlet的作用是将请求分发到不同的处理器。从Spring 2.5开始，使用Java 5或者以上版本的用户可以采用基于注解的controller声明方式。
- Spring MVC框架像许多其他MVC框架一样, 以【请求为驱动】, 围绕一个【核心Servlet】进行请求分派及提供其他功能， DispatcherServlet仅仅是一个的Servlet（它继承自HttpServlet）。

> 分发的流程大致如下：



> 我们甚至可以大致看一下源码：

众所周知，servlet中的核心方法是【service方法】，当请求一个servlet时会主动调用service方法，而在DispatcherServlet的service方法中，其核心时调用了一个doDispatch的方法，如下：

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

        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

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

            try {
                // 判断请求中有没有文件
                processedRequest = checkMultipart(request);
                multipartRequestParsed = (processedRequest != request);

                // 获得一个过滤器链，这就是处理器适配器的工作
                mappedHandler = getHandler(processedRequest);
                if (mappedHandler == null) {
                    noHandlerFound(processedRequest, response);
                    return;
                }

                // 确定当前请求的处理程序适配器
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

                // 省略一些
                ...

                // 处理器链调用所有拦截器的前置处理程序，如有不满足的直接返回：
                if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                    return;
                }

                // 此处由处理器适配器调用我们写的controller。
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

                if (asyncManager.isConcurrentHandlingStarted()) {
                    return;
                }

                applyDefaultViewName(processedRequest, mv);
                // 处理器链调用所有拦截器的后置处理程序，如有不满足的直接返回：
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            }
            catch (Exception ex) {
                dispatchException = ex;
            }
            catch (Throwable err) {
                dispatchException = new NestedServletException("Handler dispatch failed", err);
            }
```

```
52                // 处理最终结果，视图解析器处理mv，还要做统一的异常处理
53                processDispatchResult(processedRequest, response, mappedHandler, mv,
    dispatchException);
54            }
55        ...
56        }
```

> 为了理解拦截器，虽然没有学习，但是我们可以看一下这个接口：

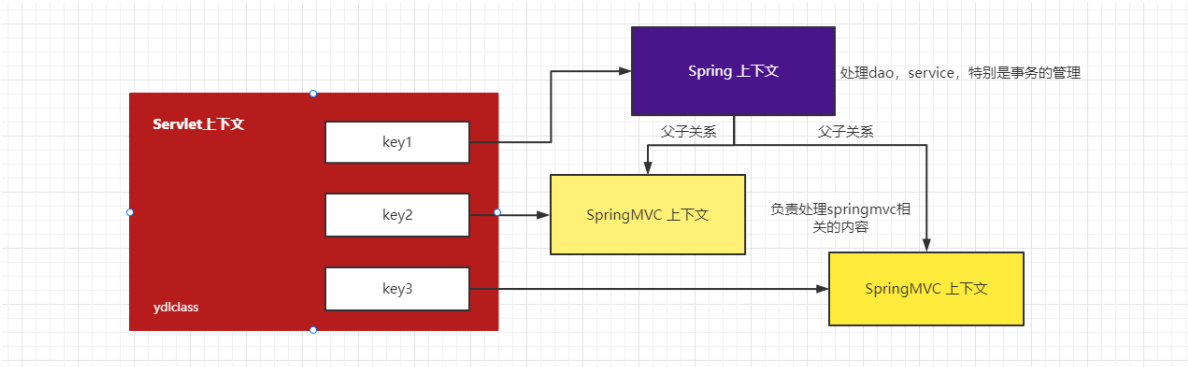
```
1   public interface HandlerInterceptor {
2       //处理器执行之前
3       default boolean preHandle(HttpServletRequest request, HttpServletResponse
    response, Object handler) throws Exception {
4           return true;
5       }
6       //处理器执行之后
7       default void postHandle(HttpServletRequest request, HttpServletResponse
    response, Object handler, @Nullable ModelAndView modelAndView) throws Exception {
8       }
9       //完成之后
10      default void afterCompletion(HttpServletRequest request, HttpServletResponse
    response, Object handler, @Nullable Exception ex) throws Exception {
11      }
12  }
```

> 其实这个处理过程简单一点回答总结如下：

1. 通过url匹配一个过滤器链，其中包含多个过滤器和一个处理器
2. 第一步调用拦截器的preHandle方法
3. 第二步执行handler方法
4. 第三部调用拦截器的postHandle方法
5. 将结果给视图解析器进行处理
6. 处理完成后调用afterCompletion

#3、三个上下文

在我们的web项目中存在至少三个上下文，分别是【servlet上下文】，【spring上下文】以及【springmvc上下文】，具体如下：



（1）ServletContext

- 对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是我们的ServletContext，其为后面的spring IoC容器提供一个宿主环境。

（2）spring上下文

- 在web.xml的配置中，我们需要提供一个监听器【ContextLoaderListener】。在web容器启动时，会触发【容器初始化】事件，此时contextLoaderListener会监听到这个事件，其contextInitialized方法会被调用。
- 在这个方法中，spring会初始化一个【上下文】，这个上下文被称为【根上下文】，即【WebApplicationContext】，这是一个接口类，其实际的实现类是XmlWebApplicationContext。这个就是spring的IoC容器，其对应的Bean定义的配置由web.xml中的【context-param】配置指定，默认配置文件为【/WEB-INF/applicationContext.xml】。
- 在这个IoC容器初始化完毕后，spring以【WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE】为属性Key，将其存储到ServletContext中，便于将来获取；

```
public interface WebApplicationContext extends ApplicationContext {
    String ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE = WebApplicationContext.class.getName() + ".ROOT";
```

相关配置：

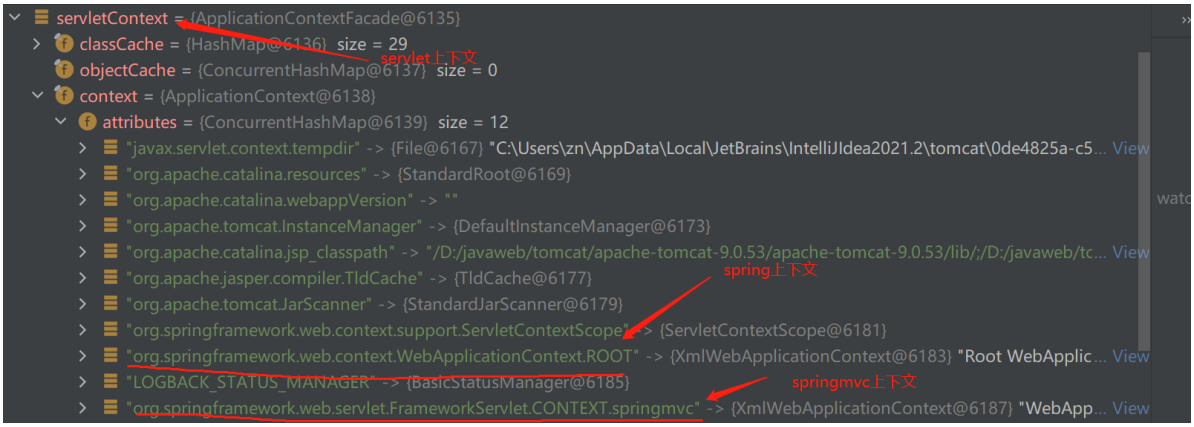
```
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/app-context.xml</param-value>
</context-param>
```

（3）springmvc上下文

- contextLoaderListener监听器初始化完毕后，开始初始化web.xml中配置的Servlet，这个servlet可以配置多个，通常只配置一个，以最常见的DispatcherServlet为例，这个servlet实际上是一个【标准的前端控制器】，用以转发、匹配、处理每个servlet请求。
- DispatcherServlet在初始化的时候会建立自己的IoC上下文，用以持有【spring mvc相关的bean】。在建立DispatcherServlet自己的IoC上下文时，会利用【WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE】先从ServletContext中获取之前的【根上下文】作为自己上下文的【parent上下文】。有了这个parent上下文之后，再初始化自己持有的上下文，这个上下文本质上也是XmlWebApplicationContext，默认读取的配置文件是【/WEB-INF/springmvc-servlet.xml】，当然我们也可以使用init-param标签的【contextConfigLocation属性】进行配置。
- DispatcherServlet初始化自己上下文的工作在其【initStrategies】方法中可以看到，大概的工作就是初始化处理器映射、视图解析等。这个servlet自己持有的上下文默认实现类也是xmlWebApplicationContext。初始化完毕后，spring以【"org.springframework.web.servlet.FrameworkServlet.CONTEXT"+Servlet名称】为Key，也将其存到ServletContext中，以便后续使用。这样每个servlet就持有自己的上下文，即拥有自己独立的bean空间，同时各个servlet还可以共享相同的bean，即根上下文(第2步中初始化的上下文)定义的那些bean。

注：springMVC容器只负责创建Controller对象，不会创建service和dao，并且他是一个子容器。而spring的容器只负责Service和dao对象，是一个父容器。子容器可以看见父容器的对象，而父容器看不见子容器的对象，这样各司其职。

我们可以通过debug，使用`ServletContext servletContext = req.getServletContext()`查方法看ServletContext，如下：



#四、核心技术篇

#1、视图和模型拆分

视图和模型相伴相生，但是springmvc给我们提供了更好的，更优雅的解决方案：

- Model会在调用handler时通过参数的形式传入
- View可以简化为字符串形式返回

这样的解决方案才是企业开发中最常用的：

```
@RequestMapping("/test1")
public String testAnnotation(Model model){
    model.addAttribute("hello","hello annotationMvc as string");
    return "annotation";
}
```

#2、重定向和转发

在返回的字符串中，默认使用视图解析器进行视图跳转：

springmvc给我们提供了更好的解决【重定向和转发】的方案：

> 返回视图字符串加前缀redirect就可以进行重定向：

```
redirect:/redirectController/redirectTest
redirect:https://www.baidu.cm
```

> 返回视图字符串加前缀forward就可以进行请求转发，而不走视图解析器：

```
// 会将请求转发至/a/b
forward:/a/b
```

#3、RequestMapping和衍生注解

在刚才的小练习中，我们看到了这个注解【@RequestMapping】

- 这个注解很关键，他不仅仅是一个方法级的注解，还是一个类级注解。
- 如果放在类上，相当于给每个方法默认都加上一个前缀url。

```
@Controller
@RequestMapping("/user/")
public class AnnotationController {

    @RequestMapping("register")
    public String register(Model model){
        ......
        return "register";
    }

    @RequestMapping("login")
    public String login(){
        ......
        return "register";
    }
}
```

> 好处

- 一个类一般处理一类业务，可以统一加上前缀，好区分
- 简化书写复杂度

> RequestMapping注解有六个属性，如下

1、`value`，`method`；

- value： 指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；
- method： 指定请求的method类型， GET、POST、PUT、DELETE等；

2、`consumes`，`produces`；

- consumes：指定处理中的请求的内容类型（Content-Type），例如application/json；
- produces：指定返回响应的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回

```
@GetMapping(value = "{id}",produces = {"application/json;charset=utf-8"})
```

3、`params`，`headers`；

- params： 指定request中必须包含某些参数值处理器才会继续执行。
- headers： 指定request中必须包含某些指定的header值处理器才会继续执行。

```
@RequestMapping(value = "add",method = RequestMethod.POST,
                consumes = "application/json",produces = "text/plain",
                headers = "name",params = {"age","times"}
               )
@ResponseBody
public String add(Model model){
    model.addAttribute("user","add user");
    return "user";
}equestMapping还有几个衍生注解，用来处理特定方法的请求：
```

```
 @GetMapping("getOne")
 public String getOne(){
     return "user";
 }

 @PostMapping("insert")
 public String insert(){
     return "user";
 }

 @PutMapping("update")
 public String update(){
     return "user";
 }

 @DeleteMapping("delete")
 public String delete(){
     return "user";
 }
```

源码中能看带GetMapping注解中有@RequestMapping作为元注解修饰：

```
@RequestMapping(
    method = {RequestMethod.GET}
)
public @interface GetMapping {
```

#4、URI 模式匹配

`@RequestMapping`可以支持【URL模式匹配】，为此，spring提供了两种选择（两个类）：

- `PathPattern` — `PathPattern`是 Web 应用程序的推荐解决方案，也是 Spring WebFlux 中的唯一选择，比较新。
- `AntPathMatcher` — 使用【字符串模式与字符串路径】匹配。这是Spring提供的原始解决方案，用于选择类路径、文件系统和其他位置上的资源。

小知识：二者目前都存在于Spring技术栈内，做着相同的事。虽说现在还鲜有同学了解到PathPattern，我认为淘汰掉AntPathMatcher只是时间问题（特指web环境哈），毕竟后浪总归有上岸的一天。但不可否认，二者将在较长时间内共处，那么它俩到底有何区别呢？

- 出现时间，AntPathMatcher是一个早在2003年（Spring的第一个版本）就已存在的路径匹配器，而PathPattern是Spring 5新增的，旨在用于替换掉较为“古老”的AntPathMatcher。
- 功能差异，PathPattern去掉了Ant字样，但保持了很好的向下兼容性：除了不支持将`**`写在path中间之外，其它的匹配规则从行为上均保持和AntPathMatcher一致，并且还新增了强大的{*pathVariable}的支持，他能匹配最后的多个路劲，并获取路径的值。
- 性能差异，Spring官方说PathPattern的性能优于AntPathMatcher。

> 以下是一些模式匹配的示例：

- `"/resources/ima?e.png"` - 匹配路径段中的一个字符
- `"/resources/*.png"` - 匹配路径段中的零个或多个字符
- `"/resources/**"` - 匹配多个路径段
- `"/projects/{project}/versions"` - 匹配路径段并将其【捕获为变量】
- `"/projects/{project:[a-z]+}/versions"` - 使用正则表达式匹配并【捕获变量】

捕获的 URI 变量可以使用 `@PathVariable` 注解，示例例如：

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

您还可以在类和方法级别声明 URI 变量，如以下示例所示：

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class OwnerController {

    @GetMapping("/pets/{petId}")
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

> 有时候会遇到一个url可以匹配到多个路由的情况，这个时候就是由Spring的AntPatternComparator完成优先级处理，大致规律如下：

比如：有两个匹配规则一个是 /a/，**一个是 /a/b/**，还有一个是/a/b/*，如果访问的url是/a/b/c，其实这三个路由都能匹配到，在匹配优先级中，有限级如下：

匹配方式	优先级
全路径匹配，例如：配置路由/a/b/c	第一优先级
带有{}路径的匹配，例如：/a/{b}/c	第二优先级
正则匹配，例如：/a/{regex:\d{3}}/c	第三优先级
带有 * 路径的匹配，例如：/a/b/*	第四优先级
带有 ** 路径的匹配，例如：/a/b/**	第五优先级
仅仅是双通配符：/**	最低优先级

注意：

1. 当有多个*和多个'{}'时，命中单个路径多的，优先越高。
2. 多 的优先级高于'**'，会优先匹配带有

> 我们还可以从一个类中看出，当一个url匹配了多个处理器时，优先级是如何考虑的，这个类是AntPathMatcher的一个内部类：

```
protected static class AntPatternComparator implements Comparator<String> {
```

```java

    @Override
    public int compare(String pattern1, String pattern2) {
        PatternInfo info1 = new PatternInfo(pattern1);
        PatternInfo info2 = new PatternInfo(pattern2);
        .....

        boolean pattern1EqualsPath = pattern1.equals(this.path);
        boolean pattern2EqualsPath = pattern2.equals(this.path);
        // 完全相等，是无法比较的
        if (pattern1EqualsPath && pattern2EqualsPath) {
            return 0;
        }
        // pattern1和urlequals,返回负数 1胜出
        else if (pattern1EqualsPath) {
            return -1;
        }
        // pattern2和urlequals,返回正数，2胜出
        else if (pattern2EqualsPath) {
            return 1;
        }

        // 都是前缀匹配，长的优先   /a/b/**  /a/**
        if (info1.isPrefixPattern() && info2.isPrefixPattern()) {
            return info2.getLength() - info1.getLength();
        }
        // 非前缀匹配的优先级高
        else if (info1.isPrefixPattern() && info2.getDoubleWildcards() == 0) {
            return 1;
        }
        else if (info2.isPrefixPattern() && info1.getDoubleWildcards() == 0) {
            return -1;
        }

        // 匹配数越少，优先级越高
        if (info1.getTotalCount() != info2.getTotalCount()) {
            return info1.getTotalCount() - info2.getTotalCount();
        }

        // 路径越短越好
        if (info1.getLength() != info2.getLength()) {
            return info2.getLength() - info1.getLength();
        }

        // 单通配符个数，数量越少优先级越高
        if (info1.getSingleWildcards() < info2.getSingleWildcards()) {
            return -1;
        }
        else if (info2.getSingleWildcards() < info1.getSingleWildcards()) {
            return 1;
        }
        // url参数越少越优先
        if (info1.getUriVars() < info2.getUriVars()) {
            return -1;
        }
        else if (info2.getUriVars() < info1.getUriVars()) {
            return 1;
        }
```

```
60
61              return 0;
62          }
63      }
```

> 源码中我们看到的信息如下：

1、完全匹配者，优先级最高

2、都是前缀匹配（/a/**），匹配路由越长，优先级越高

3、前缀匹配优先级，比非前缀的低

4、需要匹配的数量越少，优先级越高，this.uriVars + this.singleWildcards + (2 * this.doubleWildcards);

5、路劲越短优先级越高

6、*越少优先级越高

7、{}越少优先级越高

#5、牛逼的传参

> 在学习servlet时，我们是这样获取请求参数的：

```
1   @PostMapping("insert")
2   public String insert(HttpServletRequest req){
3       String username = req.getParameter("username");
4       String password = req.getParameter("password");
5       // 其他操作
6       return "success";
7   }
```

> 有了springmvc之后，我们以后再也不需要使用 `getParamter` 一个一个获取参数了：

```
1   @Controller
2   @RequestMapping("/user/")
3   public class LoginController {
4
5       @RequestMapping("login")
6       public String login(String username,String password){
7           System.out.println(username);
8           System.out.println(password);
9           return "login";
10      }
11  }
```

> 那么问题又来了，如果一个表单几十个参数怎么获取啊？更牛的来了方式他来了：

需要提前定义一个User对象：

```
1   public class User {
2
3       private String username;
4       private String password;
5       private int age;
```

```
6
7       public String getUsername() {
8           return username;
9       }
10
11      public void setUsername(String username) {
12          this.username = username;
13      }
14
15      public String getPassword() {
16          return password;
17      }
18
19      public void setPassword(String password) {
20          this.password = password;
21      }
22
23      public int getAge() {
24          return age;
25      }
26
27      public void setAge(int age) {
28          this.age = age;
29      }
30  }
```

> 直接在参数中申明user对象

```
1   @Controller
2   @RequestMapping("/user/")
3   public class LoginController {
4
5       @RequestMapping("register")
6       public String register(User user){
7           System.out.println(user);
8           return "register";
9       }
10
11      @RequestMapping("login")
12      public String login(String username,String password){
13          System.out.println(username);
14          System.out.println(password);
15          return "login";
16      }
17  }
```

(1) @RequestParam

您可以使用 `@RequestParam` 注解将【请求参数】（即查询参数或表单数据）绑定到控制器中的方法参数。

```java
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }
}
```

默认情况下，使用此注解的方法参数是必需的，但我们可以通过将`@RequestParam`注解的【`required`标志设置】为`false`来指定方法参数是可选的。如果目标方法参数类型不是String，则应用会自动进行类型转换，这个后边会讲。

请注意，使用`@RequestParam`是可选的。默认情况下，任何属于简单值类型且未被任何其他参数解析器解析的参数都被视为使用【`@RequestParam`】。

(2) `@RequestHeader`

您可以使用`@RequestHeader`注解将请求的首部信息绑定到控制器中的方法参数中：

假如我们的请求header如下：

```
Host localhost:8080
Accept text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding gzip,deflate
Accept-Charset ISO -8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive 300
```

以下示例获取`Accept-Encoding`和`Keep-Alive`标头的值：

```java
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding,
        @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```

小知识：当`@RequestHeader`注解上的使用`Map<String, String>`，`MultiValueMap<String, String>`或`HttpHeaders`参数，则map会被填充有所有header的值。当然，我们依然可以使用requied的属性来执行该参数不是必须的。

(3) `@CookieValue`

我们可以使用`@CookieValue`注解将请求中的 cookie 的值绑定到控制器中的方法参数。

假设我们的请求中带有如下cookie：

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

以下示例显示了如何获取 cookie 值：

```java
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) {
    //...
}
```

(4) @ModelAttribute

您可以使用 `@ModelAttribute` 注解在方法参数上来访问【模型中的属性】，或者在不存在的情况下对其进行实例化。模型的属性会覆盖来自 HTTP Servlet 请求参数的值，其名称与字段名称匹配，这称为数据绑定，它使您不必【处理解析】和【转换单个查询参数】和表单字段。以下示例显示了如何执行此操作：

```java
@RequestMapping("/register")
public String register(@ModelAttribute("user") UserForm user) {
    ...
}
```

> 还有一个例子

@ModelAttribute 和 @RequestMapping 注解同时应用在方法上时，有以下作用：

1. 方法的【返回值】会存入到 Model 对象中，key为 ModelAttribute 的 value 属性值。
2. 方法的返回值不再是方法的访问路径，访问路径会变为 @RequestMapping 的 value 值，例如：@RequestMapping(value = "/index") 跳转的页面是 index.jsp 页面。

```java
@Controller
public class ModelAttributeController {
    // @ModelAttribute和@RequestMapping同时放在方法上
    @RequestMapping(value = "/index")
    @ModelAttribute("name")
    public String model(@RequestParam(required = false) String name) {
        return name;
    }
}
```

(5) @SessionAttribute

如果您需要访问全局管理的预先存在的会话属性，并且可能存在或可能不存在，您可以 `@SessionAttribute` 在方法参数上使用注解，如下所示示例显示：

```java
@RequestMapping("/")
public String handle(@SessionAttribute User user) {
    // ...
}
```

(6) @RequestAttribute

和 `@SessionAttribute` 一样，您可以使用 `@RequestAttribute` 注解来访问先前创建的存在与请求中的属性（例如，由 Servlet `Filter` 或 `HandlerInterceptor`）创建或在请求转发中添加的数据：

```java
@GetMapping("/")
public String handle(@RequestAttribute Client client) {
    // ...
}
```

(7) @SessionAttributes

@SessionAttributes注解应用到Controller上面，可以将Model中的属性同步到session当中：

```
@Controller
@RequestMapping("/Demo.do")
@SessionAttributes(value={"attr1","attr2"})
public class Demo {

    @RequestMapping(params="method=index")
    public ModelAndView index() {
        ModelAndView mav = new ModelAndView("index.jsp");
        mav.addObject("attr1", "attr1Value");
        mav.addObject("attr2", "attr2Value");
        return mav;
    }

    @RequestMapping(params="method=index2")
    public ModelAndView index2(@ModelAttribute("attr1")String attr1,
@ModelAttribute("attr2")String attr2) {
        ModelAndView mav = new ModelAndView("success.jsp");
        return mav;
    }
}
```

> 附加一个注解使用的案例：

```
@RequestMapping("insertUser")
    public String insertUser(
            @RequestParam(value = "age",required = false) Integer age,
            @RequestHeader(value = "Content-Type",required = false) String
contentType,
            @RequestHeader(required = false) String name,
            @CookieValue(value = "company",required = false) String company,
            @SessionAttribute(value = "username",required = false) String
onlineUser,
            @RequestAttribute(required = false) Integer count,
            @ModelAttribute("date") Date date,
            @SessionAttribute(value = "date",required = false) Date sessionDate
    ) {
        System.out.println("sessionDate = " + sessionDate);
        System.out.println("date = " + date);
        System.out.println("count = " + count);
        System.out.println("onlineUser = " + onlineUser);
        System.out.println("age = " + age);
        System.out.println("contentType = " + contentType);
        System.out.println("name = " + name);
        System.out.println("company = " + company);
        return "user";
    }
```

（8）数组的传递

在类似批量删除的场景中，我们可能需要传递一个id数组，此时我们仅仅需要将方法的参数指定为数组即可：

```
@GetMapping("/array")
public String testArray(@RequestParam("array") String[] array) throws Exception {
    System.out.println(Arrays.toString(array));
    return "array";
}
```

我们可以发送如下请求，可以是多个名称相同的key，也可以是一个key，但是值以逗号分割的参数：

```
http://localhost:8080/app/hellomvc?array=1,2,3,4
```

或者

```
http://localhost:8080/app/hellomvc?array=1&array=3
```

结果都是没有问题的：

```
[2021-12-27 12:07:08,079] Artifact app: Deploy took 1,359 milliseconds
[1, 2, 3, 4]
27-Dec-2021 00:07:16.413 信息 [Catalina-utility-1] org.apache.catalina.
27-Dec-2021 00:07:16.439 信息 [Catalina-utility-1] org.apache.catalina.
[1, 3]
```

（9）复杂参数的传递

当然我们在进行参数接收的时候，其中可能包含很复杂的参数，一个请求中可能包含很多项内容，比如以下表单：

当然我们要注意表单中的name（参数中key）的写法：

```
<form action="user/queryParam" method="post">
    排序字段：<br>
    <input type="text" name="sortField">
    <hr>
    数组：<br>
    <input type="text" name="ids[0]"> <br>
    <input type="text" name="ids[1]">
    <hr>
    user对象：<br>
    <input type="text" name="user.username" placeholder="姓名"><br>
    <input type="text" name="user.password" placeholder="密码">
    <hr>
    list集合<br>
    第一个元素：<br>
    <input type="text" name="userList[0].username" placeholder="姓名"><br>
    <input type="text" name="userList[0].password" placeholder="密码"><br>
    第二个元素： <br>
    <input type="text" name="userList[1].username" placeholder="姓名"><br>
    <input type="text" name="userList[1].password" placeholder="密码">
    <hr>
    map集合<br>
    第一个元素：<br>
    <input type="text" name="userMap['user1'].username" placeholder="姓名"><br>
    <input type="text" name="userMap['user1'].password" placeholder="密码"><br>
```

```
        第二个元素：<br>
        <input type="text" name="userMap['user2'].username" placeholder="姓名"><br>
        <input type="text" name="userMap['user2'].password" placeholder="密码"><br>
        <input type="submit" value="提交">
    </form>
```

然后我们需要搞一个实体类用来接收这个表单的参数：

```
@Data
public class QueryVo {
    private String sortField;
    private User user;
    private Long[] ids;
    private List<User> userList;
    private Map<String, User> userMap;
}
```

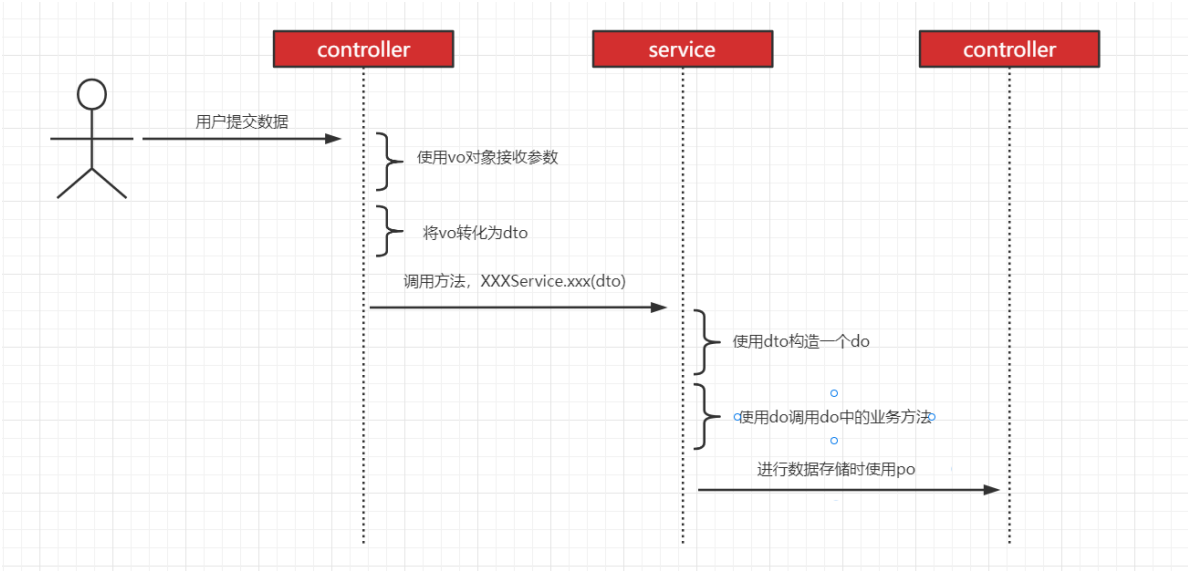
编写接口进行测试，我们发现表单的数据已经尽数传递了进来：

```
@PostMapping("queryParam")
public String queryParam(QueryVo queryVo) {
    System.out.println(queryVo);
    return "user";
}
```

> 拓展知识：

- VO（View Object）：视图对象，用于展示层，它的作用是把某个指定页面（或组件）的所有数据封装起来。
- DTO（Data Transfer Object）：数据传输对象，这个概念来源于J2EE的设计模式，原来的目的是为了EJB的分布式应用提供粗粒度的数据实体，以减少分布式调用的次数，从而提高分布式调用的性能和降低网络负载，但在这里，我泛指用于展示层与服务层之间的数据传输对象。
- DO（Domain Object）：领域对象，就是从现实世界中抽象出来的有形或无形的业务实体。
- PO（Persistent Object）：持久化对象，它跟持久层（通常是关系型数据库）的数据结构形成一一对应的映射关系，如果持久层是关系型数据库，那么，数据表中的每个字段（或若干个）就对应PO的一个（或若干个）属性。

下面以一个时序图建立简单模型来描述上述对象在三层架构应用中的位置：



大致流程如下：

- 用户发出请求（可能是填写表单），表单的数据在展示层被匹配为VO；
- 展示层把VO转换为服务层对应方法所要求的DTO，传送给服务层；
- 服务层首先根据DTO的数据构造（或重建）一个DO，调用DO的业务方法完成具体业务；
- 服务层把DO转换为持久层对应的PO（可以使用ORM工具，也可以不用），调用持久层的持久化方法，把PO传递给它，完成持久化操作；
- 数据传输顺序：VO `=> DTO =` > DO ===> PO

相对来说越是靠近显示层的概念越不稳定，复用度越低。分层的目的，就是复用和相对稳定性。

小知识：一般的简单工程中，并不会进行这样的设计，我们可能有一个User类就可以了，并不需要什么VO、DO啥的。但是，随着项目工程的复杂化，简单的对象已经没有办法在各个层的使用，项目越是复杂，就需要越是复杂的设计方案，这样才能满足高扩展性和维护性。

#6、设定字符集

> springmvc内置了一个统一的字符集处理过滤器，我们只要在`web.xml`中配置即可：

```
<filter>
    <filter-name>CharacterEncodingFilter</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>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

> 看看他的核心源码，是不是和我们之前自己写的很像呢？

```
@Override
protected void doFilterInternal(
    HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
    throws ServletException, IOException {

    String encoding = getEncoding();
    if (encoding != null) {
        if (isForceRequestEncoding() || request.getCharacterEncoding() == null) {
            request.setCharacterEncoding(encoding);
        }
        if (isForceResponseEncoding()) {
            response.setCharacterEncoding(encoding);
        }
    }
    filterChain.doFilter(request, response);
}
```

#7、返回json数据（序列化）

> 我们经常需要使用ajax请求后台获取数据，而不需要访问任何的页面，这种场景在前后分离的项目当中尤其重要：

这种做法其实很简单，大致步骤如下：

- 将我们的对象转化为json字符串。
- 将返回的内容直接写入响应体，不走视图解析器。
- 然后将Content-Type设置为 `application/json` 即可。

为了实现这个目的，我们可以引入fastjson：

```
<dependency>
   <groupId>com.alibaba</groupId>
   <artifactId>fastjson</artifactId>
   <version>1.2.68</version>
</dependency>
```

```
// produces指定了响应的Content-Type
@RequestMapping(value = "getUsers",produces = {"application/json;charset=utf-8"})
@ResponseBody  // 将返回的结果直接写入响应体，不走视图解析器
public String getUsers(){
    List<User> users =  new ArrayList<User>(){{
        add(new User("Tom","2222"));
        add(new User("jerry","333"));
    }};
    return JSONArray.toJSONString(users);
}
```

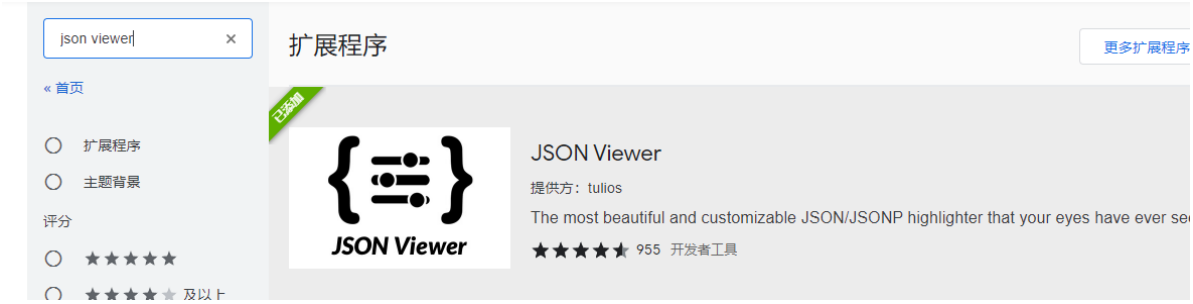
测试：成功！

注意：@ResponseBody能将返回的结果直接放在响应体中，不走视图解析器。

127.0.0.1:8088/app/user/getAll

```
// 20220104165846
// http://127.0.0.1:8088/app/user/getAll

[
  {
    "username": "tom",
    "password": "123"
  },
  {
    "username": "jerry"
  }
]
```

浏览器中添加插件json viewer可以有如上显示：



> 当然springmvc也考虑到了，每次这样写也其实挺麻烦，我们还可以向容器注入一个专门处理消息转换的bean：

这个转化器的作用就是：当不走视图解析器时，如果发现【返回值是一个对象】，就会自动将返回值转化为json字符序列：

```
<mvc:annotation-driven >
        <mvc:message-converters>
            <bean id="fastjson"
class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter">
                <property name="supportedMediaTypes">
                    <list>
                        <!-- 这里顺序不能反，一定先写text/html,不然ie下会出现下载提示 -->
                    <value>text/html;charset=UTF-8</value>
                    <value>application/json;charset=UTF-8</value>
                    </list>
                </property>
            </bean>
        </mvc:message-converters>
</mvc:annotation-driven>
```

以后我们的controller就可以写成下边的样子了：

```
@RequestMapping(value = "getUsersList")
@ResponseBody
public List<User> getUsersList(){
    return   new ArrayList<User>(){{
        add(new User("邸智伟","2222"));
        add(new User("刘展鹏","333"));
    }};
}
```

> 当然我们还可以使用一个更加流行的组件jackson来处理，他的工作和fastjson一致，首先需要引入以下依赖：

```
<!--jackson-->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

我们还可以对序列化的过程进行额外的一些配置：

```
public class CustomObjectMapper extends ObjectMapper {

    public CustomObjectMapper() {
        super();
        //去掉默认的时间戳格式
```

```
        configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
        //设置为东八区
        setTimeZone(TimeZone.getTimeZone("GMT+8"));
        //设置日期转换yyyy-MM-dd HH:mm:ss
        setDateFormat(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
        // 设置输入:禁止把POJO中值为null的字段映射到json字符串中
        configure(SerializationFeature.WRITE_NULL_MAP_VALUES, false);
        // 空值不序列化
        setSerializationInclusion(JsonInclude.Include.NON_NULL);
        // 反序列化时，属性不存在的兼容处理

 getDeserializationConfig().withoutFeatures(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
        // 序列化枚举是以toString()来输出，默认false，即默认以name()来输出
        configure(SerializationFeature.WRITE_ENUMS_USING_TO_STRING, true);
    }
}
```

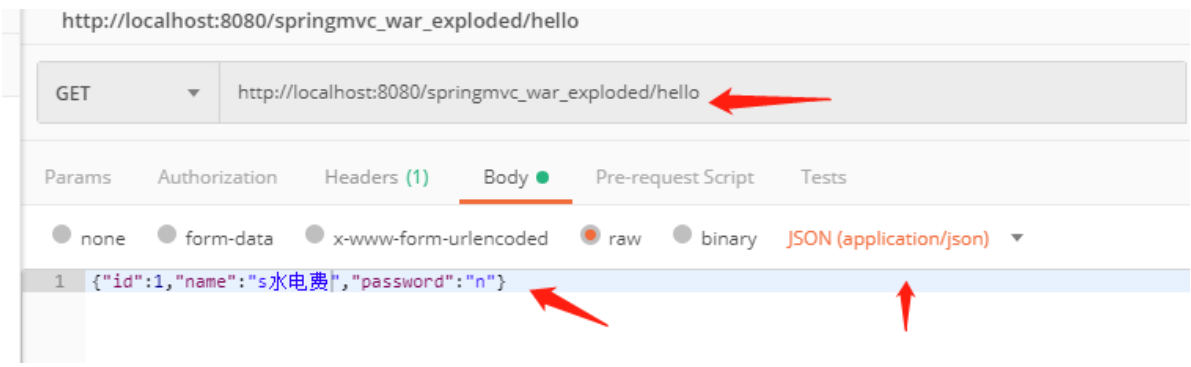
编写配置文件：

```
<mvc:annotation-driven>

    <mvc:message-converters>
        <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
            <!-- 自定义Jackson的objectMapper -->
            <property name="objectMapper" ref="customObjectMapper" />
            <property name="supportedMediaTypes">
                <list>
                    <value>text/plain;charset=UTF-8</value>
                    <value>application/json;charset=UTF-8</value>
                </list>
            </property>
        </bean>
    </mvc:message-converters>

</mvc:annotation-driven>
<!--注入我们写的对jackson的配置的bean-->
<bean name="customObjectMapper" class="com.ydlclass.CustomObjectMapper"/>
```

测试成功：

#8、获取请求中的json数据

在前端发送的数据中可能会如如下情况，Contetn-Type是application/json，请求体中是json格式数据：



@RequestBody注解可以【直接获取请求体的数据】。

如果我们配置了消息转化器，消息转化器会将请求体中的json数据反序列化成目标对象，如下所示：

```
@PostMapping("insertUser")
public String insertUser(@RequestBody User user) {
    System.out.println(user);
    return "user";
}
```

当然，我们可以吧消息转化器注解掉，直接使用一个String来接收请求体的内容：

#9、数据转化

假如有如下场景，前端传递过来一个日期字符串，但是后端需要使用Date类型进行接收，这时就需要一个类型转化器进行转化。

自定义的类型转化器只支持从requestParam获取的参数进行转化，我们可以定义如下，其实学习spring时我们已经接触过这个Converter接口：

```
public class StringToDateConverter implements Converter<String, Date> {
    @Override
    public Date convert(String source) {
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy_MM_dd hh,mm,ss");
        try {
            return simpleDateFormat.parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

然后，我们需要在配置文件中进行配置：

```
<!-- 开启mvc的注解 -->
<mvc:annotation-driven conversion-service="conversionService" />

<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <set>
            <bean id="stringToDateConverter" class="cn.itnanls.convertors.StringToDateConverter"/>
        </set>
    </property>
</bean>
```

> 对于时间类型的处理，springmvc给我们提供了一个比较完善的解决方案，使用注解@DateTimeFormat，同时配合jackson提供的@JsonFormat注解几乎可以满足我们的所有需求。

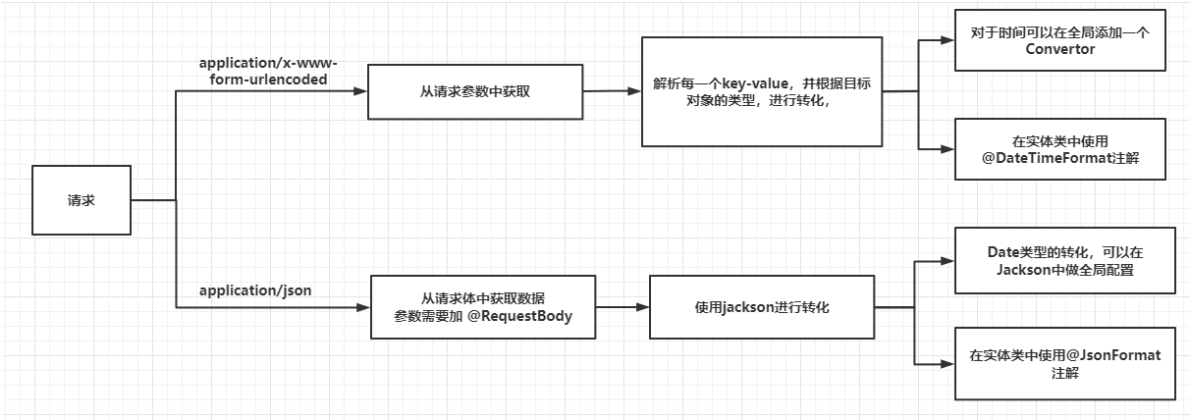
@DateTimeFormat：当从requestParam中获取string参数并需要转化为Date类型时，会根据此注解的参数pattern的格式进行转化。

@JsonFormat：当从请求体中获取json字符序列，需要反序列化为对象时，时间类型会按照这个注解的属性内容进行处理。

这两个注解需要加在实体类的对应字段上即可：

```java
// 对象和json互相转化的过程当中按照此转化方式转哈
@JsonFormat(
            pattern = "yyyy年MM月dd日",
            timezone = "GMT-8"
    )
// 从requestParam中获取参数并且转化
@DateTimeFormat(pattern = "yyyy年MM月dd日")
private Date birthday;
```

> 处理的过程大致如下：



#10、数据校验

- JSR 303 是 Java 为 Bean 数据合法性校验提供的标准框架，它包含在 JavaEE 6.0 中。
- JSR 303 通过在 Bean 属性上标注类似于 @NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证。

Constraint	详细信息
`@Null`	被注解的元素必须为 `null`
`@NotNull`	被注解的元素必须不为 `null`
`@AssertTrue`	被注解的元素必须为 `true`
`@AssertFalse`	被注解的元素必须为 `false`
`@Min(value)`	被注解的元素必须是一个数字，其值必须大于等于指定的最小值
`@Max(value)`	被注解的元素必须是一个数字，其值必须小于等于指定的最大值
`@DecimalMin(value)`	被注解的元素必须是一个数字，其值必须大于等于指定的最小值
`@DecimalMax(value)`	被注解的元素必须是一个数字，其值必须小于等于指定的最大值
`@Size(max, min)`	被注解的元素的大小必须在指定的范围内
`@Digits (integer, fraction)`	被注解的元素必须是一个数字，其值必须在可接受的范围内
`@Past`	被注解的元素必须是一个过去的日期
`@Future`	被注解的元素必须是一个将来的日期
`@Pattern(value)`	被注解的元素必须符合指定的正则表达式

> Hibernate Validator 扩展注解

Hibernate Validator 是 JSR 303 的一个参考实现，除支持所有标准的校验注解外，它还支持以下的扩展注解

Hibernate Validator 附加的 constraint

Constraint	详细信息
`@Email`	被注解的元素必须是电子邮箱地址
`@Length`	被注解的字符串的大小必须在指定的范围内
`@NotEmpty`	被注解的字符串的必须非空
`@Range`	被注解的元素必须在合适的范围内

> Spring MVC 数据校验

Spring MVC 可以对表单参数进行校验，并将结果保存到对应的【BindingResult】或 【Errors 】对象中。

> 要实现数据校验，需要引入已下依赖

```xml
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.9.Final</version>
</dependency>
```

并在实体类加上特定注解

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserVO {

    @NotNull(message = "用户名不能为空")
    private String username;

    @NotNull(message = "用户名不能为空")
    private String password;

    @Min(value = 0, message = "年龄不能小于{value}")
    @Max(value = 120,message = "年龄不能大于{value}")
    private int age;

    @JsonFormat(
            pattern = "yyyy-MM-dd",
            timezone = "GMT-8"
    )
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    @Past(message = "生日不能大于今天")
    private Date birthday;

    @Pattern(regexp = "^1([358][0-9]|4[579]|66|7[0135678]|9[89])[0-9]{8}$",
message = "手机号码不正确")
    private String phone;

    @Email
    private String email;
}
```

在配置文件中配置如下内容，增加hibernate校验：

```xml
<bean id="localValidator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <property name="providerClass"
value="org.hibernate.validator.HibernateValidator"/>
</bean>
<!--注册注解驱动-->
<mvc:annotation-driven validator="localValidator"/>
```

controller使用@Validated标识验证的对象，紧跟着的BindingResult获取错误信息

```java
@PostMapping("insert")
public String insert(@Validated UserVO user, BindingResult br) {
    List<ObjectError> allErrors = br.getAllErrors();
    Iterator<ObjectError> iterator = allErrors.iterator();
    // 打印以下错误结果
    while (iterator.hasNext()){
        ObjectError error = iterator.next();
        log.error("user数据校验错误:{}",error.getDefaultMessage());
    }

    if(allErrors.size() > 0){
        return "error";
    }

    System.out.println(user);
    return "user";
}
```

记住：永远不要相信用户的输入，我们开发的系统凡是涉及到用户输入的地方，都要进行校验，这里的校验分为前台校验和后台校验，前台校验通常由javascript来完成，后台校验主要由java来负责，这里我们可以通过spring mvc+hibernate validator完成。

#11、视图解析器详解

我们默认的视图解析器是如下的配置，它主要是处理jsp页面的映射渲染：

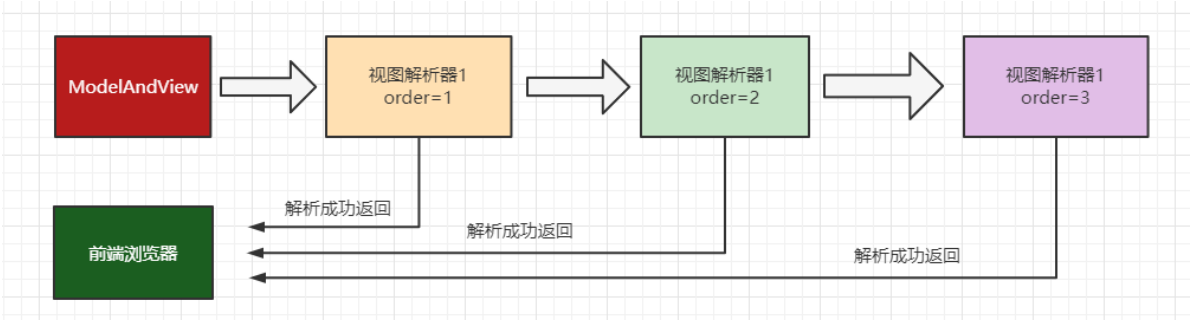
```xml
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
 id="internalResourceViewResolver">
    <!-- 前缀 -->
    <property name="prefix" value="/WEB-INF/page/" />
    <!-- 后缀 -->
    <property name="suffix" value=".jsp" />
</bean>
```

如果我们想添加新的视图解析器，则需要给旧的新增一个order属性，或者直接删除原有的视图解析器：

```xml
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="internalResourceViewResolver">
    <!-- 前缀 -->
    <property name="prefix" value="/WEB-INF/page/" />
    <!-- 后缀 -->
    <property name="suffix" value=".jsp" />
    <property name="order" value="10"/>
</bean>
```

- 这里的order表示视图解析的【优先级】，数字越小优先级越大（即：0为优先级最高，所以优先进行处理视图），InternalResourceViewResolver在项目中的优先级一般要设置为最低，也就是order要最大。不然它会影响其他视图解析器。
- 当处理器返回逻辑视图时（也就是return “string”），要经过**视图解析器链**，如果前面的解析器能处理，就不会继续往下传播。如果不能处理就要沿着解析器链继续寻找，直到找到合适的视图解析器。

如下图所示：



然后，我们可以配置一个新的Tymeleaf视图解析器，order设置的低一些，这样两个视图解析器都可以生效：

```
<!--thymeleaf的视图解析器-->
<bean id="templateResolver"

 class="org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver">
    <property name="prefix" value="/WEB-INF/templates/" />
    <property name="suffix" value=".html" />
    <property name="templateMode" value="HTML" />
    <property name="cacheable" value="true" />
</bean>
<!--thymeleaf的模板引擎配置-->
<bean id="templateEngine"
      class="org.thymeleaf.spring4.SpringTemplateEngine">
    <property name="templateResolver" ref="templateResolver" />
    <property name="enableSpringELCompiler" value="true" />
</bean>
<bean id="viewResolver" class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
    <property name="order" value="1"/>
    <property name="characterEncoding" value="UTF-8"/>
    <property name="templateEngine" ref="templateEngine"/>
</bean>
```

> 添加两个相关依赖

```
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf</artifactId>
    <version>3.0.14.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.thymeleaf/thymeleaf-spring4 -->
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring4</artifactId>
    <version>3.0.14.RELEASE</version>
</dependency>
```

> 模板中需要添加对应的命名空间

```
<html xmlns:th="http://www.thymeleaf.org" >
```

thymeleaf官网： **Thymeleafopen in new window**

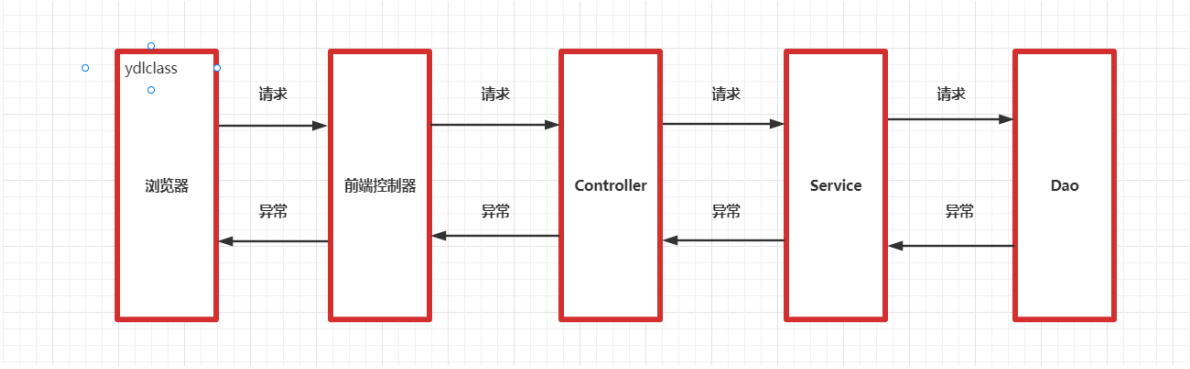
thymeleaf语法详解：

#12、全局异常捕获

（1）HandlerExceptionResolver

在Java中，对于异常的处理一般有两种方式：

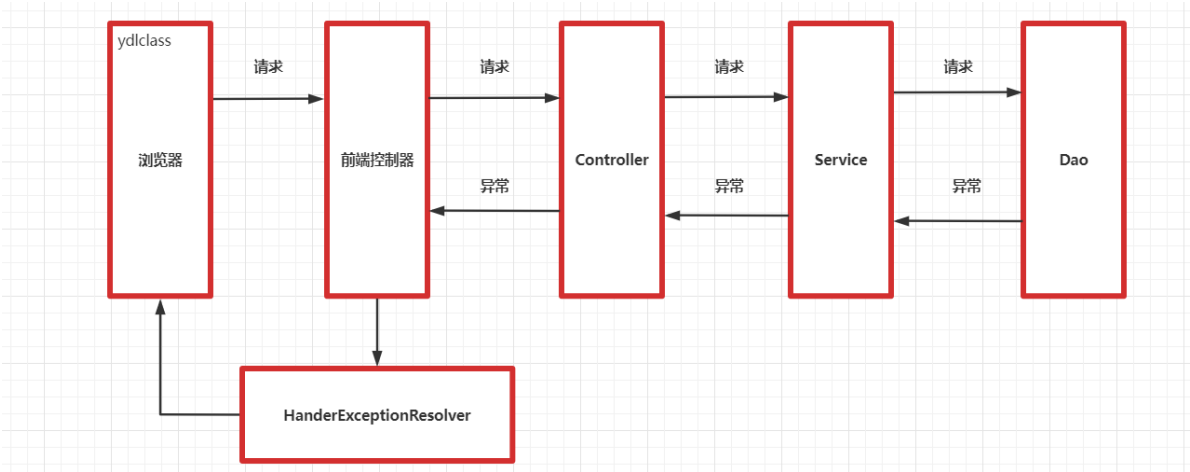
- 一种是当前方法捕获处理（try-catch），这种处理方式会造成业务代码和异常处理代码的耦合。
- 另一种是自己不处理，而是抛给调用者处理（throws），调用者再抛给它的调用者，也就是一直向上抛，指导传递给浏览器。



被异常填充的页面是长这个样子的：

在这种方法的基础上，衍生出了SpringMVC的异常处理机制。系统的dao、service、controller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，如下图：

小知识：service层尽量不要处理异常，如果自己捕获并处理了，异常就不生效了。特别是不要生吞异常。



Spring MVC的Controller出现异常的默认处理是响应一个500状态码，再把错误信息显示在页面上，如果用户看到这样的页面，一定会觉得你这个网站太LOW了。

要解决Controller的异常问题，当然也不能在每个处理请求的方法中加上异常处理，那样太繁琐了。

通过源码我们得知，需要写一个HandlerExceptionResolver，并实现其方法：

```
public class GlobalExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request,
                                         HttpServletResponse response, Object
handler, Exception ex) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("error", ex.getMessage());
        modelAndView.setViewName("error");
        return modelAndView;
    }
}
```

```
<bean id="globalExecptionResovler"
class="com.lagou.exception.GlobalExecptionResovler"></bean>
```

```
@Component
public class GlobalExecptionResovler implements HandlerExceptionResolver {}
```

小知识：当然在web中我们也能对异常进行统一处理：

```
<!--处理500异常-->
<error-page>
    <error-code>500</error-code>
    <location>/500.jsp</location>
</error-page>
<!--处理404异常-->
<error-page>
    <error-code>404</error-code>
    <location>/404.jsp</location>
</error-page>
```

（2）@ControllerAdvice

该注解同样能实现异常的全局统一处理，而且实现起来更加简单优雅，当然使用这个注解有一下三个功能：

- 处理全局异常
- 预设全局数据
- 请求参数预处理

我们主要学习其中的全局异常处理，`@ControllerAdvice` 配合 `@ExceptionHandler` 实现全局异常处理：

```
@Slf4j
@ControllerAdvice
public class GlobalExceptionResolverController  {

    @ExceptionHandler(ArithmeticException.class)
    public String processArithmeticException(ArithmeticException ex){
        log.error("发生了数学类的异常：",ex);
        return "error";
    }

    @ExceptionHandler(BusinessException.class)
    public String processBusinessException(BusinessException ex){
        log.error("发生了业务相关的异常：",ex);
        return "error";
    }
```

```
16
17      @ExceptionHandler(Exception.class)
18      public String processException(Exception ex){
19          log.error("发生了其他的异常：",ex);
20          return "error";
21      }
22  }
```

#13、处理资源

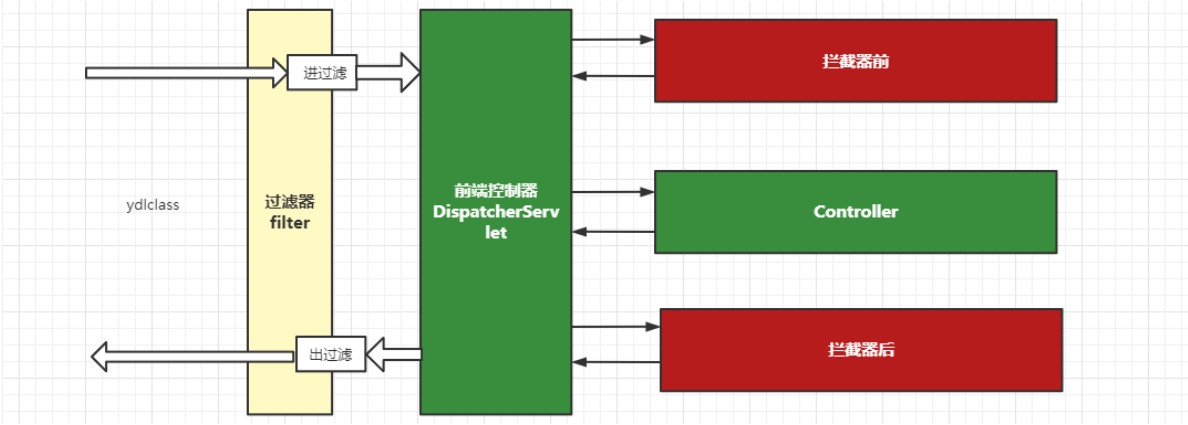
当我们使用了springmvc后，所有的请求都会交给springmvc进行管理，当然也包括静态资源，比如 `/static/js/index.js` ，这样的请求如果走了中央处理器，必然会抛出异常，因为没有与之对应的controller，这样我们可以使用一下配置进行处理：

```
1   <mvc:resources mapping="/js/**" location="/static/js/"/>
2   <mvc:resources mapping="/css/**" location="/static/css/"/>
3   <mvc:resources mapping="/img/**" location="/static/img/"/>
```

经过这样的配置后，我们直接配置了请求url和路径的映射关系，就不会再走我们的前端控制器了。

#14、拦截器

1. SpringMVC提供的拦截器类似于JavaWeb中的过滤器，只不过**SpringMVC拦截器只拦截被前端控制器拦截的请求**，而过滤器拦截从前端发送的【任意】请求。
2. 熟练掌握 `SpringMVC` 拦截器对于我们开发非常有帮助，在没使用权限框架(`shiro，spring security`)之前，一般使用拦截器进行认证和授权操作。
3. SpringMVC拦截器有许多应用场景，比如：登录认证拦截器，字符过滤拦截器，日志操作拦截器等等。



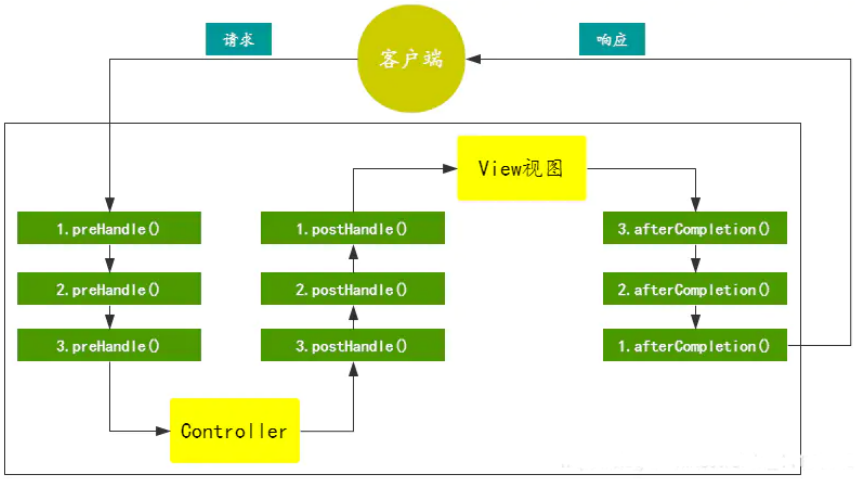
（1）自定义拦截器

> SpringMVC拦截器的实现一般有两种方式

1. 自定义的 `Interceptor` 类要实现了Spring的HandlerInterceptor接口。
2. 继承实现了 `HandlerInterceptor` 接口的类，比如Spring已经提供的实现了HandlerInterceptor接口的抽象类HandlerInterceptorAdapter。

```java
public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {}

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {}
}
```

（2）拦截器拦截流程



（3）拦截器规则

我们可以配置多个拦截器，每个拦截器中都有三个方法。下面将总结多个拦截器中的方法执行规律。

1. preHandle：Controller方法处理请求前执行，根据拦截器定义的顺序，正向执行。
2. postHandle：Controller方法处理请求后执行，根据拦截器定义的顺序，逆向执行。需要所有的preHandle方法都返回true时才会调用。
3. afterCompletion：View视图渲染后处理方法：根据拦截器定义的顺序，逆向执行。preHandle返回true也会调用。

（4）登录拦截器

接下来编写一个登录拦截器，这个拦截器可以实现认证操作。就是当我们还没有登录的时候，如果发送请求访问我们系统资源时，拦截器不放行，请求失败。只有登录成功后，拦截器放行，请求成功。登录拦截器只要在preHandle()方法中编写认证逻辑即可，因为是在请求执行前拦截。代码实现如下：

```java
/**
 *  登录拦截器
 */
public class LoginInterceptor implements HandlerInterceptor {

    /**
        在执行Controller方法前拦截，判断用户是否已经登录，
        登录了就放行，还没登录就重定向到登录页面
```

```
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        HttpSession session = request.getSession();
        User user = session.getAttribute("user");
        if (user == null){
            //还没登录，重定向到登录页面
            response.sendRedirect("/toLogin");
        }else {
            //已经登录，放行
            return true;
        }
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {}

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {}
}
```

编写完SpringMVC拦截器，我们还需要在springmvc.xml配置文件中，配置我们编写的拦截器，配置代码如下：

1. 配置需要拦截的路径
2. 配置不需要拦截的路径
3. 配置我们自定义的拦截器类

```
<mvc:interceptors>
    <mvc:interceptor>
        <!--
            mvc:mapping：拦截的路径
            /**：是指所有文件夹及其子孙文件夹
            /*：是指所有文件夹，但不包含子孙文件夹
            /：Web项目的根目录
        -->
        <mvc:mapping path="/**"/>
        <!--
                mvc:exclude-mapping：不拦截的路径,不拦截登录路径
                /toLogin：跳转到登录页面
                /login：登录操作
            -->
        <mvc:exclude-mapping path="/toLogin"/>
        <mvc:exclude-mapping path="/login"/>
        <!--class属性就是我们自定义的拦截器-->
        <bean id="loginInterceptor" class="com.ydlclass.interceptor.LoginInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

#15、全局配置类

springmvc有一个可用作用于做全局配置的接口，这个接口是`WebMvcConfigurer`，在这个接口中有很多默认方法，每一个默认方法都可以进行一项全局配置，这些配置可以和我们配置文件的配置一一对应：这些配置在全局的xml中也可以进行配置：

> 列举几个xml的配置

```
<!--处理静态资源-->
<mvc:resources mapping="/js/**" location="/static/js/"/>
<mvc:resources mapping="/css/**" location="/static/css/"/>
<mvc:resources mapping="/./image/**" location="/static/./image/"/>

<!--配置页面跳转-->
<mvc:view-controller path="/toGoods" view-name="goods"/>
<mvc:view-controller path="/toUpload" view-name="upload"/>
<mvc:view-controller path="/websocket" view-name="websocket"/>

<mvc:cors>
    <mvc:mapping path="/goods/**" allowed-methods="*"/>
</mvc:cors>
```

> 列举几个常用的WebMvcConfigurer的配置

```
@Configuration
@EnableWebMvc
public class MvcConfiguration implements WebMvcConfigurer {

    // 拦截器进行配置
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LoginInterceptor())
                .addPathPatterns("/**")
                .excludePathPatterns(List.of("/toLogin","/login"))
                .order(1);
    }

    // 资源的配置
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {

 registry.addResourceHandler("/js/**").addResourceLocations("/static/js/");

 registry.addResourceHandler("/css/**").addResourceLocations("/static/css/");
    }

    // 跨域的全局配置
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins("*")
                .allowedMethods("GET","POST","PUT","DELETE")
                .maxAge(3600);
    }

    // 页面跳转的配置
```

```
31      @Override
32      public void addViewControllers(ViewControllerRegistry registry) {
33          registry.addViewController("/index").setViewName("index");
34      }
35
36  }
```

#五、跨域

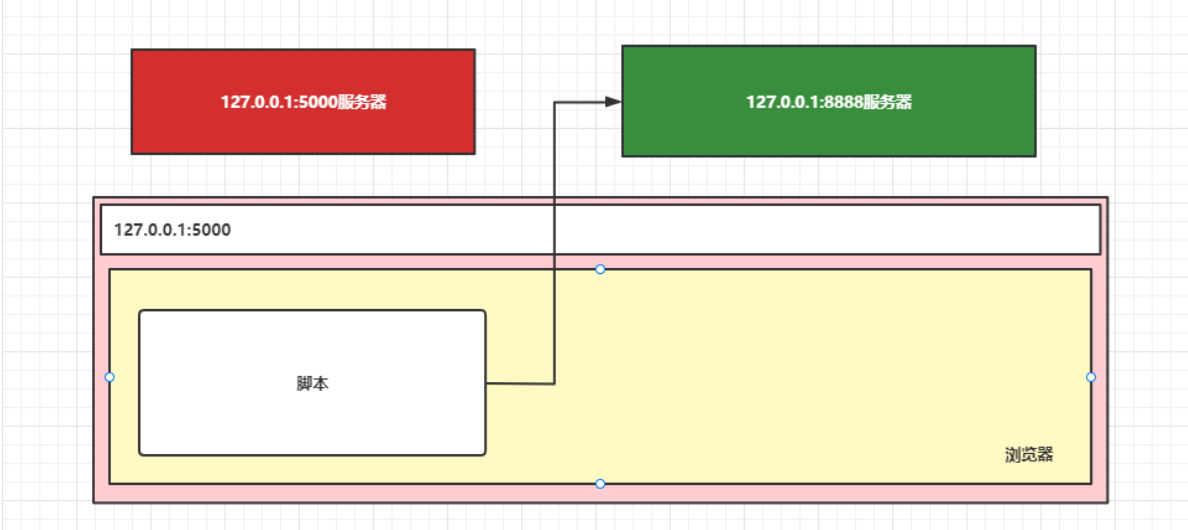
更多更详细的跨域的问题可以看我的另一个视频： https://www.bilibili.com/video/BV1nU4y1W7Rf ，这个视频最后学完springboot以后看。

#1、同源策略

同源策略（Sameoriginpolicy）是一种约定，它是浏览器最核心也最基本的安全功能。同源策略会阻止一个域的javascript脚本和另外一个域的内容进行交互。所谓同源（即指在同一个域）就是两个页面具有相同的协议（protocol），主机（host）和端口号（port）。

#2、什么是跨域

当一个请求url的协议、域名、端口三者之间任意一个与当前页面url不同时，就会产生跨域。



举一个例子：从127.0.0.1:5000访问的页面中，有Javascript使用ajax访问127.0.0.1:8888的接口就会产生跨域；

当前页面url	被请求页面url	是否跨域	原因
http://www.ydlclass.com/	http://www.ydlclass.com/index.html	不跨域	同源（协议、域名、端口号相同）
http://www.ydlclass.com/	https://www.ydlclass.com/index.html	跨域	协议不同（http/https）
http://www.ydlclass.com/	http://www.baidu.com/	跨域	主域名不同（test/baidu）
http://www.ydlclass.com/	http://blog.ydlclass.com/	跨域	子域名不同（www/blog）
http://www.ydlclass.com:8080/	http://www.ydlclass.com:7001/	跨域	端口号不同（8080/7001）

> 非同源限制

- 无法读取非同源网页的 Cookie、LocalStorage 和 IndexedDB。
- 无法接触非同源网页的 DOM
- 无法向非同源地址发送 AJAX 请求

#3、两种请求

全称是"跨域资源共享"(Cross-origin resource sharing)；

浏览器将CORS请求分成两类：简单请求（simple request）和非简单请求（not-so-simple request）。

只要同时满足以下两大条件，就属于简单请求：

(1) 请求方法是以下三种方法之一：

- HEAD
- GET
- POST

(2) HTTP的头信息不超出以下几种字段：

- Accept
- Accept-Language
- Content-Language
- Last-Event-ID
- Content-Type：只限于三个值`application/x-www-form-urlencoded`、`multipart/form-data`、`text/plain`

这是为了兼容表单（form），因为历史上表单一直可以发出跨域请求。AJAX 的跨域设计就是，只要表单可以发，AJAX 就可以直接发。

凡是不同时满足上面两个条件，就属于非简单请求。

浏览器对这两种请求的处理，是不一样的。

（1）简单请求

> 基本流程

对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个`Origin`字段。

下面是一个例子，浏览器发现这次跨源AJAX请求是简单请求，就自动在头信息之中，添加一个`Origin`字段。

```
GET /cors HTTP/1.1
Origin: http://api.bob.com
Host: api.ydlclass.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

上面的头信息中，`Origin`字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）。服务器根据这个值，决定是否同意这次请求。

如果`Origin`指定的源，不在许可范围内，服务器会返回一个正常的HTTP回应。浏览器发现，这个回应的头信息没有包含`Access-Control-Allow-Origin`字段（详见下文），就知道出错了，从而抛出一个错误，被`XMLHttpRequest`的`onerror`回调函数捕获。注意，这种错误无法通过状态码识别，因为HTTP回应的状态码有可能是200。

如果`Origin`指定的域名在许可范围内，服务器返回的响应，会多出几个头信息字段。

```
Access-Control-Allow-Origin: http://api.bob.com
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: FooBar
Content-Type: text/html; charset=utf-8
```

上面的头信息之中，有三个与CORS请求相关的字段，都以`Access-Control-`开头。

（1）Access-Control-Allow-Origin

该字段是必须的。它的值要么是请求时`Origin`字段的值，要么是一个`*`，表示接受任意域名的请求。

（2）Access-Control-Allow-Credentials

该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为`true`，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为`true`，如果服务器不要浏览器发送Cookie，删除该字段即可。

（3）Access-Control-Expose-Headers

该字段可选。CORS请求时，`XMLHttpRequest`对象的`getResponseHeader()`方法只能拿到6个基本字段：`Cache-Control`、`Content-Language`、`Content-Type`、`Expires`、`Last-Modified`、`Pragma`。如果想拿到其他字段，就必须在`Access-Control-Expose-Headers`里面指定。上面的例子指定，`getResponseHeader('FooBar')`可以返回`FooBar`字段的值。

（4）withCredentials 属性

上面说到，CORS请求默认不发送Cookie和HTTP认证信息。如果要把Cookie发到服务器，一方面要服务器同意，指定`Access-Control-Allow-Credentials`字段。

```
Access-Control-Allow-Credentials: true
```

另一方面，开发者必须在AJAX请求中打开`withCredentials`属性。

```
var xhr = new XMLHttpRequest();
xhr.withCredentials = true;
```

否则，即使服务器同意发送Cookie，浏览器也不会发送。或者，服务器要求设置Cookie，浏览器也不会处理。

但是，如果省略`withCredentials`设置，有的浏览器还是会一起发送Cookie。这时，可以显式关闭`withCredentials`。

```
xhr.withCredentials = false;
```

需要注意的是，如果要发送Cookie，`Access-Control-Allow-Origin`就不能设为星号，必须指定明确的、与请求网页一致的域名。同时，Cookie依然遵循同源政策，只有用服务器域名设置的Cookie才会上传，其他域名的Cookie并不会上传，且（跨源）原网页代码中的`document.cookie`也无法读取服务器域名下的Cookie。

（2）非简单请求

> 预检请求

非简单请求是那种对服务器有特殊要求的请求，比如请求方法是`PUT`或`DELETE`，或者`Content-Type`字段的类型是`application/json`。OPTIONS

非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight）。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的`XMLHttpRequest`请求，否则就报错。

下面是一段浏览器的JavaScript脚本。

```
var url = 'http://api.ydlclass.com/cors';
var xhr = new XMLHttpRequest();
xhr.open('PUT', url, true);
xhr.setRequestHeader('X-Custom-Header', 'value');
xhr.send();
```

上面代码中，HTTP请求的方法是`PUT`，并且发送一个自定义头信息`X-Custom-Header`。

浏览器发现，这是一个非简单请求，就【自动】发出一个"预检"请求，要求服务器确认可以这样请求。下面是这个"预检"请求的HTTP头信息。

```
OPTIONS /cors HTTP/1.1
Origin: http://api.bob.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: api.ydlclass.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

"预检"请求用的请求方法是`OPTIONS`，表示这个请求是用来询问的。头信息里面，关键字段是`Origin`，表示请求来自哪个源。

除了`Origin`字段，"预检"请求的头信息包括两个特殊字段。

（1）Access-Control-Request-Method

该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是`PUT`。

（2）Access-Control-Request-Headers

该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是`X-Custom-Header`。

> 预检请求的响应

服务器收到"预检"请求以后，检查了`Origin`、`Access-Control-Request-Method`和`Access-Control-Request-Headers`字段以后，确认允许跨源请求，就可以做出回应。

```
 HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://api.bob.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

上面的HTTP回应中，关键的是`Access-Control-Allow-Origin`字段，表示`http://api.bob.com`可以请求数据。该字段也可以设为星号，表示同意任意跨源请求。

```
Access-Control-Allow-Origin: *
```

如果服务器否定了"预检"请求，会返回一个正常的HTTP回应，但是没有任何CORS相关的头信息字段。这时，浏览器就会认定，服务器不同意预检请求，因此触发一个错误，被`XMLHttpRequest`对象的`onerror`回调函数捕获。控制台会打印出如下的报错信息。

```
XMLHttpRequest cannot load http://api.ydlclass.com.
Origin http://api.bob.com is not allowed by Access-Control-Allow-Origin.
```

服务器回应的其他CORS相关字段如下。

```
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Access-Control-Allow-Credentials: true
Access-Control-Max-Age: 1728000
```

（1）Access-Control-Allow-Methods

该字段必需，它的值是逗号分隔的一个字符串，表明服务器支持的所有跨域请求的方法。注意，返回的是所有支持的方法，而不单是浏览器请求的那个方法。这是为了避免多次"预检"请求。

（2）Access-Control-Allow-Headers

如果浏览器请求包括`Access-Control-Request-Headers`字段，则`Access-Control-Allow-Headers`字段是必需的。它也是一个逗号分隔的字符串，表明服务器支持的所有头信息字段，不限于浏览器在"预检"中请求的字段。

（3）Access-Control-Allow-Credentials

该字段与简单请求时的含义相同。

（4）Access-Control-Max-Age

该字段可选，用来指定本次预检请求的有效期，单位为秒。上面结果中，有效期是20天（1728000秒），即允许缓存该条回应1728000秒（即20天），在此期间，不用发出另一条预检请求。

#4、解决方案

首先想到的就是使用过滤器进行统一的处理，当然在单个的servlet或者controller中也可以单独处理，基本的逻辑就是在响应的首部信息中加入需要的首部信息字段，解决方案如下：

```
public class CORSFilter implements Filter{

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        response.setHeader("Access-Control-Allow-Origin", "*");
        response.setHeader("Access-Control-Allow-Methods", "POST, GET");
        response.setHeader("Access-Control-Max-Age", "3600");
        response.setHeader("Access-Control-Allow-Headers", "Content-Type, Access-Control-Allow-Headers, Authorization, X-Requested-With");
        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void destroy() {

    }
}
```

对api为前缀的请求都进行处理：

```
<!-- CORS Filter -->
<filter>
    <filter-name>CORSFilter</filter-name>
    <filter-class>com.ydlclass.filter.CORSFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>CORSFilter</filter-name>
    <url-pattern>/api/*</url-pattern>
</filter-mapping>
```

到这里，就可以简单的实现 CORS 跨域请求了，上面的过滤器将会为所有请求的响应加上 `Access-Control-Allow-*` 首部，换言之就是允许来自任意源的请求来访问该服务器上的资源。而在实际开发中可以根据需要开放跨域请求权限以及控制响应头部等等。

> springmvc给我们提供了更加简单的解决方案

- 在Controller 上使用 `@CrossOrigin` 注解就可以实现跨域，这个注解是一个类级别也是方法级别的注解：

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("goods")
public class GoodsController{
}
```

如果同时在 Controller 和方法上都有使用 `@CrossOrigin` 注解，那么在具体某个方法上的 CORS 属性将是两个注解属性合并的结果，如果属性的设置发生冲突，那么Controller 上的主机属性将被覆盖。

我们也可以使用配置类进行全局的配置：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("*")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(false).maxAge(3600);
    }
}
```

基于 XML 配置文件与上等效：

```
<mvc:cors>
  <mvc:mapping path="/api/**"
        allowed-origins="*"
        allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3"
        exposed-headers="header1, header2" allow-credentials="false"
        max-age="123" />

    <mvc:mapping path="/resources/**"
        allowed-origins="http://domain1.com" />

</mvc:cors>
```

#六、 restful

自从Roy Fielding博士在2000年他的博士论文中提出 **RESTopen in new window** （Representational State Transfer）风格的软件架构模式后，REST就基本上成为Web API的标准了。

> restful是一种风格，可以遵循，也可以不遵循，但是现在他已经变成主流。

#1、Rest架构的主要原则

- 网络上的所有事物都被抽象为资源。
- 每个资源都有一个唯一的资源标识符。
- 同一个资源具有多种表现形式他可能是xml，也可能是json等。
- 对资源的各种操作不会改变资源标识符。
- 所有的操作都是无状态的。
- 符合REST原则的架构方式即可称为RESTful。

#2、什么是Restful

Restful web service是一种常见的rest的应用,是遵守了rest风格的web服务，rest式的web服务是一种ROA(The Resource-Oriented Architecture)(面向资源的架构).

在restful风格中，我们将互联网的资源抽象成资源，将获取资源的方式定义为方法，从此请求再也不止get和post了：

客户端请求	原来风格URL地址	RESTful风格URL地址
查询所有用户	/user/findAll	GET /user
查询编号为1的用户	/user/findById?id=1	GET /user/1
新增一个用户	/user/save	POST /user
修改编号为1的用户	/user/update	PUT /user/1
删除编号为1的用户	/user/delete?id=1	DELETE /user/1

> Spring MVC 对 RESTful应用提供了以下支持

- 利用@RequestMapping 指定要处理请求的URI模板和HTTP请求的动作类型
- 利用@PathVariable讲URI请求模板中的变量映射到处理方法参数上
- 利用Ajax,在客户端发出PUT、DELETE动作的请求

#3、数据过滤

我们想获取所有用户，使用如下url即可 `/user` 。但是真是场景下，我们可能需要需要一些条件进行过滤：

例如：我们需要查询名字叫张三的前10条数据，使用以下场景即可：

```
/user?name=jerry&pageSize=10&page=1
```

第一：查询的url不变，变的是条件，我们只需要同伙url获取对应的参数就能实现复杂的多条件查询。

#4、RequestMapping中指定请求方法

```
@RequestMapping(value = "/{id}", method = RequestMethod.GET)
@RequestMapping(value = "/add", method = RequestMethod.POST)
@RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
@RequestMapping(value = "/{id}", method = RequestMethod.PUT)
```

当然还有更好用的

```
@GetMapping("/user/{id}")
@PostMapping("/user")
@DeleteMapping("/user/{id}")
@PutMapping("/user/{id}")
```

#4、ajax还能这么玩

可以采用Ajax方式发送PUT和DELETE请求

> 我们可以使用当下比较流行的axios组件测试

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    const instance = axios.create({
        baseURL: 'http://127.0.0.1:8088/app/'
    });
    // 为给定 ID 的 user 创建请求
    instance.get('goods')
        .then(function (response) {
        console.log(response);
    }).catch(function (error) {
        console.log(error);
    });

    instance.get('goods/1')
        .then(function (response) {
        console.log(response);
    })
        .catch(function (error) {
        console.log(error);
    });

    instance.post('goods', {
        name: '洗发露',
        price: 25454
    }).then(function (response) {
        console.log(response);
    }).catch(function (error) {
        console.log(error);
    });

    instance.put('goods', {
        name: '洗发露',
        price: 25454
    }).then(function (response) {
        console.log(response);
    }).catch(function (error) {
        console.log(error);
    });

    instance.delete('goods/1')
        .then(function (response) {
        console.log(response);
    }).catch(function (error) {
        console.log(error);
    });
```

```
46
47  </script>
```

> 当然，使用jquery同样可以发送如下请求

```
$.ajax( {
    type : "GET",
    url : "http://localhost:8080/springmvc/user/rest/1",
    dataType : "json",
    success : function(data) {
        console.log("get请求! ----------------------")
        console.log(data)
    }
});

$.ajax( {
    type : "DELETE",
    url : "http://localhost:8080/springmvc/user/rest/1",
    dataType : "json",
    success : function(data) {
        console.log("delete请求! ----------------------")
        console.log(data)
    }
});

$.ajax( {
    type : "put",
    url : "http://localhost:8080/springmvc/user/rest/1",
    dataType : "json",
    data: {id:12,username:"楠哥",password:"123"},
    success : function(data) {
        console.log("get请求! ----------------------")
        console.log(data)
    }
});

$.ajax( {
    type : "post",
    url : "http://localhost:8080/springmvc/user/rest",
    dataType : "json",
    data: {id:12,username:"楠哥",password:"123"},
    success : function(data) {
        console.log("get请求! ----------------------")
        console.log(data)
    }
});
41
```

#七、文件上传和下载

#一、文件上传

【MultipartResolver】用于处理文件上传。当收到请求时，DispatcherServlet 的 checkMultipart() 方法会调用 MultipartResolver 的 isMultipart() 方法判断请求中【是否包含文件】。如果请求数据中包含文件，则调用 MultipartResolver 的 resolveMultipart() 方法对请求的数据进行解析，然后将文件数据解析成 MultipartFile 并封装在 MultipartHttpServletRequest (继承了 HttpServletRequest) 对象中，最后传递给 Controller。

我们可以看到DispatcherServlet的核心方法中第一句就是如下的代码：

```
        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);
```

注： MultipartResolver 默认不开启，需要手动开启。

文件上传对前端表单有如下要求：为了能上传文件，必须将表单的【method设置为POST】，并将enctype设置为【multipart/form-data】。只有在这样的情况下，浏览器才会把用户选择的文件以二进制数据发送给服务器。

这里，我们对表单中的 enctype 属性做个详细的说明：

- application/x-www-form-urlencoded：默认方式，只处理表单域中的 value 属性值，采用这种编码方式的表单会将表单域中的值处理成 URL 编码方式。
- multipart/form-data：这种编码方式会以二进制流的方式来处理表单数据，这种编码方式会把文件域指定文件的内容也封装到请求参数中，不会对字符编码。

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

一旦设置了enctype为multipart/form-data，浏览器即会采用二进制流的方式来处理表单数据，而对于文件上传的处理则涉及在服务器端解析原始的HTTP响应。

> 在2003年，Apache Software Foundation发布了开源的Commons FileUpload组件，其很快成为Servlet/JSP程序员上传文件的最佳选择。

1、我们同样需要导入这个jar包【commons-fileupload】，Maven会自动帮我们导入他的依赖包【commons-io】；

```
<!--文件上传-->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
```

2、配置bean：multipartResolver

【注意！！！这个bena的id必须为：multipartResolver，否则上传文件会报400的错误！在这里栽过坑，教训！】

```
<!--文件上传配置-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
   <!-- 请求的编码格式，必须和jSP的pageEncoding属性一致，以便正确读取表单的内容，默认为ISO-8859-1 -->
   <property name="defaultEncoding" value="utf-8"/>
   <!-- 上传文件大小上限，单位为字节（10485760=10M） -->
   <property name="maxUploadSize" value="10485760"/>
   <property name="maxInMemorySize" value="40960"/>
</bean>
```

CommonsMultipartFile 的常用方法：

- **String getOriginalFilename()：获取上传文件的原名**
- **InputStream getInputStream()：获取文件流**
- **void transferTo(File dest)：将上传文件保存到一个目录文件中**

我们去实际测试一下

3、编写前端页面

```
<form action="/upload" enctype="multipart/form-data" method="post">
 <input type="file" name="file"/>
 <input type="submit" value="upload">
</form>
```

4、`Controller`

```
@PostMapping("/upload")
@ResponseBody
public R upload(@RequestParam("file") CommonsMultipartFile file,
HttpServletRequest request) throws Exception{
    //获取文件名 : file.getOriginalFilename();
    String uploadFileName = file.getOriginalFilename();
    System.out.println("上传文件名 : "+uploadFileName);

    //上传路径保存设置
    String path = "D:/upload";
    //如果路径不存在，创建一个
    File realPath = new File(path);
    if (!realPath.exists()){
        realPath.mkdir();
    }
    System.out.println("上传文件保存地址："+realPath);
    //就问香不香，就和你写读流一样
    file.transferTo(new File(path+"/"+uploadFileName));

    return R.success();
}
```

5、测试上传文件，OK!

小知识：我们在文件上传可以考虑以下几点：

1、文件的原始信息，或者叫文件的元数据是不是可以存在数据库，具体应该怎么做?

2、文件的上传目录能不能写在配置文件当中，这个应该怎么做?

3、文件上传到服务器后可不可以安装一定的规则分目录存储，比如日期?

4、思考怎么使用阿里云的oss进行图片存储?

二、文件下载

- 第一种可以直接向response的输出流中写入对应的文件流
- 第二种可以使用 ResponseEntity<byte[]>来向前端返回文件

1、传统方式

```
@GetMapping("/download1")
@ResponseBody
public R download1(HttpServletResponse response){
    FileInputStream fileInputStream = null;
    ServletOutputStream outputStream = null;
    try {
        // 这个文件名是前端传给你的要下载的图片的id
        // 然后根据id去数据库查询出对应的文件的相关信息，包括url，文件名等
        String  fileName = "楠老师.jpg";

        //1、设置response 响应头，处理中文名字乱码问题
        response.reset(); //设置页面不缓存,清空buffer
        response.setCharacterEncoding("UTF-8"); //字符编码
        response.setContentType("multipart/form-data"); //二进制传输数据
        //设置响应头，就是当用户想把请求所得的内容存为一个文件的时候提供一个默认的文件名。
        //Content-Disposition属性有两种类型：inline 和 attachment
        //inline ：将文件内容直接显示在页面
        //attachment：弹出对话框让用户下载具体例子：
        response.setHeader("Content-Disposition",
                           "attachment;fileName="+ URLEncoder.encode(fileName,
"UTF-8"));

        // 通过url获取文件
        File file = new File("D:/upload/"+fileName);
        fileInputStream = new FileInputStream(file);
        outputStream = response.getOutputStream();

        byte[] buffer = new byte[1024];
        int len;
        while ((len = fileInputStream.read(buffer)) != -1){
            outputStream.write(buffer,0,len);
            outputStream.flush();
        }

        return R.success();
    } catch (IOException e) {
        e.printStackTrace();
        return R.fail();
    }finally {
        if( fileInputStream != null ){
            try {
                fileInputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if( outputStream != null ){
            try {
```

```
                outputStream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

#2、使用ResponseEntity

```
@GetMapping("/download2")
public ResponseEntity<byte[]> download2(){
    try {
        String fileName = "楠老师.jpg";
        byte[] bytes = FileUtils.readFileToByteArray(new
File("D:/upload/"+fileName));
        HttpHeaders headers=new HttpHeaders();
        // Content-Disposition就是当用户想把请求所得的内容存为一个文件的时候提供一个默认的文
件名。
        headers.set("Content-Disposition","attachment;filename="+
URLEncoder.encode(fileName, "UTF-8"));
        headers.set("charsetEncoding","utf-8");
        headers.set("content-type","multipart/form-data");
        ResponseEntity<byte[]> entity=new ResponseEntity<>(bytes,headers,
HttpStatus.OK);
        return entity;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

#八、WebSocket

#1、WebSocket 简介

WebSocket 协议提供了一种标准化方式，可通过单个 TCP 连接在客户端和服务器之间建立全双工、双向通信通道。它是与 HTTP 不同的 TCP 协议，但旨在通过 HTTP 工作，使用端口 80 和 443。

WebSocket 交互以 HTTP 请求开始，HTTP请求中包含`Upgrade: websocket`时，会切换到 WebSocket 协议。以下示例显示了这样的交互：

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

成功握手后，HTTP 升级请求底层的 TCP 套接字保持打开状态，客户端和服务器都可以继续发送和接收消息。

（1）HTTP 与 WebSocket

尽管 WebSocket 被设计为与 HTTP 兼容并从 HTTP 请求开始，但这两种协议会产生不同的架构和应用程序编程模型。

在 HTTP 和 REST 中，一个应用程序被建模为多个 URL。为了与应用程序交互，客户端访问这些 URL，请求-响应样式。服务器根据 HTTP URL、方法和请求头将请求路由到适当的处理程序。而在 WebSocket 中，通常只有一个 URL 用于初始连接。随后，所有应用程序消息都在同一个 TCP 连接上流动。

> 我们现在有一个需要，就是页面用实时显示当前的库存信息：

短轮询:

最简单的一种方式，就是你用JS写个死循环（setInterval），不停的去请求服务器中的库存量是多少，然后刷新到这个页面当中，这其实就是所谓的短轮询。

这种方式有明显的坏处，那就是你很浪费服务器和客户端的资源。客户端还好点，现在PC机配置高了，你不停的请求还不至于把用户的电脑整死，但是服务器就很蛋疼了。如果有1000个人停留在某个商品详情页面，那就是说会有1000个客户端不停的去请求服务器获取库存量，这显然是不合理的。

长轮询:

长轮询这个时候就出现了，其实长轮询和短轮询最大的区别是，短轮询去服务端查询的时候，不管库存量有没有变化，服务器就立即返回结果了。而长轮询则不是，在长轮询中，服务器如果检测到库存量没有变化的话，将会把当前请求挂起一段时间（这个时间也叫作超时时间，一般是几十秒）。在这个时间里，服务器会去检测库存量有没有变化，检测到变化就立即返回，否则就一直等到超时为止。

而对于客户端来说，不管是长轮询还是短轮询，客户端的动作都是一样的，就是不停的去请求，不同的是服务端，短轮询情况下服务端每次请求不管有没有变化都会立即返回结果，而长轮询情况下，如果有变化才会立即返回结果，而没有变化的话，则不会再立即给客户端返回结果，直到超时为止。

这样一来，客户端的请求次数将会大量减少（这也就意味着节省了网络流量，毕竟每次发请求，都会占用客户端的上传流量和服务端的下载流量），而且也解决了服务端一直疲于接受请求的窘境。

但是长轮询也是有坏处的，因为把请求挂起同样会导致资源的浪费，假设还是1000个人停留在某个商品详情页面，那就很有可能服务器这边挂着1000个线程，在不停检测库存量，这依然是有问题的。

（2）何时使用 WebSocket

WebSockets 可以使网页具有动态性和交互性。但是，在许多情况下，Ajax 和 HTTP 长轮询的组合可以提供简单有效的解决方案。

例如，新闻、邮件和社交提要需要动态更新，但每隔几分钟更新一次可能也完全没问题。另一方面，协作、游戏和金融应用程序需要更接近实时。

延迟本身并不是决定因素。如果消息量相对较低（例如监控网络故障），HTTP轮询可以提供有效的解决方案。低延迟、高频率和高容量的组合是使用 WebSocket 的最佳案例。

#2、实战案例

Spring Framework 提供了一个 WebSocket API，您可以使用它来编写处理 WebSocket 消息的客户端和服务器端应用程序。

（1）引入依赖

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-websocket</artifactId>
    <version>5.2.18.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-messaging</artifactId>
    <version>5.2.18.RELEASE</version>
</dependency>
```

（2）创建 WebSocket 服务器需要实现 `WebSocketHandler` 接口或者直接扩展 `TextWebSocketHandler` 或 `BinaryWebSocketHandler` 这两个类，使用起来相对简单一点。以下示例使用 `TextWebSocketHandler`：

```java
public class MessageHandler extends TextWebSocketHandler {

    Logger log = LoggerFactory.getLogger(MessageHandler.class);

    //用来保存连接进来session
    private List<WebSocketSession> sessions = new CopyOnWriteArrayList<>();

    /**
     * 关闭连接进入这个方法处理，将session从 list中删除
     */
    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {
        sessions.remove(session);
        log.info("{} 连接已经关闭，现从list中删除 ,状态信息{}", session, status);
    }

    /**
     * 三次握手成功，进入这个方法处理，将session 加入list 中
     */
    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws Exception {
        sessions.add(session);
        log.info("用户{}连接成功.... ",session);
    }

    /**
     * 处理客户发送的信息，将客户发送的信息转给其他用户
     */
    @Override
    public void handleMessage(WebSocketSession session, WebSocketMessage<?> message) throws Exception {
        log.info("收到来自客户端的信息: {}",message.getPayload());
        session.sendMessage(new TextMessage("当前时间："+
                LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss")) +",收到来自客户端的信息!"));
        for(WebSocketSession wss : sessions)
            if(!wss.getId().equals(session.getId())){
                wss.sendMessage(message);
            }
    }
```

```
}
```

（3）有专用的 WebSocket Java 配置和 XML 命名空间支持，用于将前面的 WebSocket 处理程序映射到特定的 URL，如以下示例所示：

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer{

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(new MessageHandler(), "/message")
        .addInterceptors(new HttpSessionHandshakeInterceptor())
        .setAllowedOrigins("*"); //允许跨域访问
    }
}
```

以下示例显示了与前面示例等效的 XML 配置：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/message" handler="myHandler"/>
        <websocket:handshake-interceptors>
            <bean
class="org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor"/>
        </websocket:handshake-interceptors>
    </websocket:handlers>

    <bean id="myHandler" class="com.ydlclass.MessageHandler"/>

</beans>
```

（4）使用原生js，用来访问websocket：

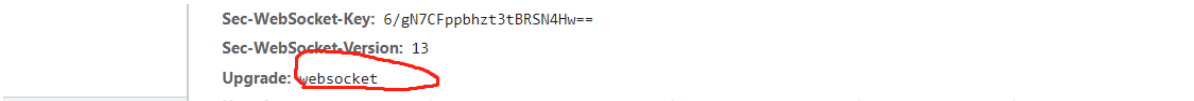
```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>websocket调试页面</title>
</head>
<body>
<div style="float: left; padding: 20px">
  <strong>location:</strong> <br />
  <input type="text" id="serverUrl" size="35" value="" /> <br />
  <button onclick="connect()">connect</button>
  <button onclick="wsclose()">disConnect</button>
  <br /> <strong>message:</strong> <br /> <input id="txtMsg" type="text"
size="50" />
```

```
    <br />
    <button onclick="sendEvent()">发送</button>
  </div>

  <div style="float: left; margin-left: 20px; padding-left: 20px; width: 350px; border-left: solid 1px #cccccc;"> <strong>消息记录</strong>
    <div style="border: solid 1px #999999;border-top-color: #CCCCCC;border-left-color: #CCCCCC; padding: 5px;width: 100%;height: 172px;overflow-y: scroll;" id="echo-log"></div>
    <button onclick="clearLog()" style="position: relative; top: 3px;">清除消息</button>
  </div>

  </div>
  </body>
  <!-- 下面是h5原生websocket js写法 -->
  <script type="text/javascript">
    let output ;
    let websocket;
    function connect(){ //初始化连接
      output = document.getElementById("echo-log")
      let inputNode = document.getElementById("serverUrl");
      let wsUri = inputNode.value;
      try{
        websocket = new WebSocket(wsUri);
      }catch(ex){
        console.log(ex)
        alert("对不起websocket连接异常")
      }

      connecting();
      window.addEventListener("load", connecting, false);
    }


    function connecting()
    {
      websocket.onopen = function(evt) { onOpen(evt) };
      websocket.onclose = function(evt) { onClose(evt) };
      websocket.onmessage = function(evt) { onMessage(evt) };
      websocket.onerror = function(evt) { onError(evt) };
    }

    function sendEvent(){
      let msg = document.getElementById("txtMsg").value
      doSend(msg);
    }

    //连接上事件
    function onOpen(evt)
    {
      writeToScreen("CONNECTED");
      doSend("WebSocket 已经连接成功! ");
    }

    //关闭事件
    function onClose(evt)
    {
```

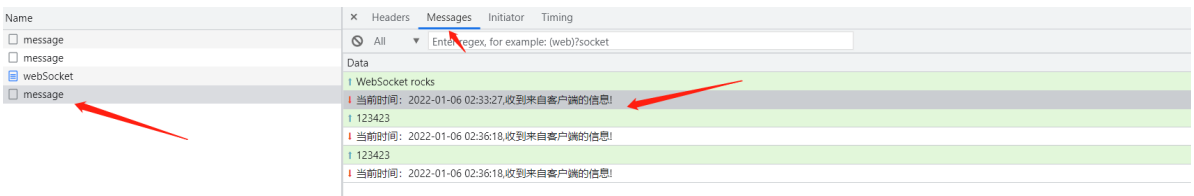
```
68        writeToScreen("连接已经断开！");
69      }
70
71      //后端推送事件
72      function onMessage(evt)
73      {
74        writeToScreen('<span style="color: blue;">服务器: ' + evt.data+'</span>');
75      }
76
77      function onError(evt)
78      {
79        writeToScreen('<span style="color: red;">异常信息:</span> ' + evt.data);
80      }
81
82      function doSend(message)
83      {
84        writeToScreen("客户端A: " + message);
85        websocket.send(message);
86      }
87
88      //清除div的内容
89      function clearLog(){
90        output.innerHTML = "";
91      }
92
93      //浏览器主动断开连接
94      function wsclose(){
95        websocket.close();
96      }
97
98      function writeToScreen(message)
99      {
100       let pre = document.createElement("p");
101       pre.innerHTML = message;
102       output.appendChild(pre);
103     }
104   </script>
105   </html>
```

```
1   ws:127.0.0.1:8088/app/message
```

我们可以看到在websocket的请求中有这样的首部信息：



而且我们多次发送消息，并没有新的请求产生：



小知识：我们经常看到有很多地方使用sockjs完成websocket的建立，原因是一些浏览器中缺少对WebSocket的支持。SockJS是一个浏览器JavaScript库，它提供了一个连贯的、跨浏览器的Javascript API，它在浏览器和web服务器之间创建了一个低延迟、全双工、跨域通信通道。

#九、整合数据库，ssm结束

> 其实就是spring整合mybatis，咱们尽量使用注解完成工作

#1、完整的依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>ssm-study</artifactId>
        <groupId>org.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>spring-mvc-study</artifactId>
    <packaging>war</packaging>

    <dependencies>
        <!-- 测试相关 -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <!-- springmvc -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.6.RELEASE</version>
        </dependency>
        <!-- servlet -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.2</version>
            <scope>provided</scope>
        </dependency>

        <!--文件上传-->
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.4</version>
        </dependency>
        <!-- jackson -->
        <dependency>
```

```
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
            <version>2.11.2</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
            <version>2.11.2</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.11.2</version>
        </dependency>

        <!-- mybatis 相关 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.2</version>
        </dependency>
        <!-- 数据库连接驱动 相关 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>

        <!-- 提供了对JDBC操作的完整封装 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.10.RELEASE</version>
        </dependency>
        <!-- 织入 相关 -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.4</version>
        </dependency>
        <!-- spring，mybatis整合包 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>2.0.2</version>
        </dependency>
        <!-- 集成德鲁伊使用 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.18</version>
        </dependency>
        <!-- 日志 -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.3</version>
        </dependency>
```

```
107
108         <dependency>
109             <groupId>org.projectlombok</groupId>
110             <artifactId>lombok</artifactId>
111             <version>1.18.12</version>
112         </dependency>
113
114         <dependency>
115             <groupId>com.alibaba</groupId>
116             <artifactId>druid</artifactId>
117             <version>1.1.18</version>
118         </dependency>
119
120         <dependency>
121             <groupId>org.slf4j</groupId>
122             <artifactId>slf4j-api</artifactId>
123             <version>1.7.30</version>
124         </dependency>
125         <dependency>
126             <groupId>ch.qos.logback</groupId>
127             <artifactId>logback-classic</artifactId>
128             <version>1.2.3</version>
129         </dependency>
130     </dependencies>
131
132     <build>
133         <plugins>
134             <plugin>
135                 <groupId>org.apache.maven.plugins</groupId>
136                 <artifactId>maven-compiler-plugin</artifactId>
137                 <version>3.1</version>
138                 <configuration>
139                     <source>1.8</source> <!-- 源代码使用的JDK版本 -->
140                     <target>1.8</target> <!-- 需要生成的目标class文件的编译版本 -->
141                     <encoding>utf-8</encoding><!-- 字符集编码 -->
142                 </configuration>
143             </plugin>
144         </plugins>
145         <resources>
146             <resource>
147                 <directory>src/main/java</directory>
148                 <includes>
149                     <include>**/*.properties</include>
150                     <include>**/*.xml</include>
151                 </includes>
152                 <filtering>false</filtering>
153             </resource>
154             <resource>
155                 <directory>src/main/resources</directory>
156                 <includes>
157                     <include>**/*.properties</include>
158                     <include>**/*.xml</include>
159                 </includes>
160                 <filtering>false</filtering>
161             </resource>
162         </resources>
163     </build>
164
```

```
</project>
```

#2、mybatis的配置文件

```
mybatis-config.xml
<?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>

</configuration>
```

#3、springmvc配置文件

```
 springmvc-servlet.xml
 <?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:mvc="http://www.springframework.org/schema/mvc"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- 自动扫包 -->
    <context:component-scan base-package="cn.itnanls"/>
    <!-- 让Spring MVC不处理静态资源 -->
    <mvc:default-servlet-handler />
    <!--  让springmvc自带的注解生效  -->
    <mvc:annotation-driven >


    </mvc:annotation-driven>
    <bean id="fastjson"
 class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter">
        <property name="supportedMediaTypes">
            <list>
                <value>text/html;charset=UTF-8</value>
                <value>application/json;charset=UTF-8</value>
            </list>
        </property>
    </bean>

    <!--文件上传配置-->
    <bean id="multipartResolver"
 class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!-- 请求的编码格式，必须和jSP的pageEncoding属性一致，以便正确读取表单的内容，默认为
 ISO-8859-1 -->
        <property name="defaultEncoding" value="utf-8"/>
        <!-- 上传文件大小上限，单位为字节（10485760=10M） -->
        <property name="maxUploadSize" value="10485760"/>
        <property name="maxInMemorySize" value="40960"/>
    </bean>
```

```
39
40      <!-- 视图解析器 -->
41      <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    id="internalResourceViewResolver">
42          <!-- 前缀 -->
43          <property name="prefix" value="/WEB-INF/page/" />
44          <!-- 后缀 -->
45          <property name="suffix" value=".jsp" />
46      </bean>
47  </beans>
```

#4、数据源配置

```
1   driver=com.mysql.cj.jdbc.Driver
2   url=jdbc:mysql://localhost:3306/ssm?
    useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai
3   username=root
4   password=root
```

#5、spring配置文件

> application.xml

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xmlns:aop="http://www.springframework.org/schema/aop"
5          xmlns:context="http://www.springframework.org/schema/context"
6          xmlns:tx="http://www.springframework.org/schema/tx"
7          xsi:schemaLocation="http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         https://www.springframework.org/schema/context/spring-context.xsd
11         http://www.springframework.org/schema/aop
12         http://www.springframework.org/schema/aop/spring-aop.xsd
13         http://www.springframework.org/schema/tx
14         http://www.springframework.org/schema/tx/spring-tx.xsd">
15
16      <!-- 加载外部的数据库信息 classpath:不叫会报错具体原因下边解释-->
17      <context:property-placeholder location="classpath:db.properties"/>
18      <!-- 加入springmvc的配置 -->
19      <import resource="classpath:springmvc-servlet.xml"/>
20
21      <!-- Mapper 扫描器 -->
22      <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
23          <!-- 扫描 cn.wmyskxz.mapper 包下的组件 -->
24          <property name="basePackage" value="cn.itnanls.dao"/>
25      </bean>
26
27      <!--配置数据源：数据源有非常多，可以使用第三方的，也可使使用Spring的-->
28      <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
29          <property name="driverClassName" value="${jdbc.driver}"/>
30          <property name="url" value="${jdbc.url}"/>
```

```
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!--配置SqlSessionFactory-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <!--关联Mybatis-->
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <property name="mapperLocations" value="classpath:mappers/*.xml"/>
    </bean>

    <!--注册sqlSessionTemplate , 关联sqlSessionFactory-->
    <bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
        <!--利用构造器注入-->
        <constructor-arg index="0" ref="sqlSessionFactory"/>
    </bean>

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


    <!--配置事务通知-->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!--配置哪些方法使用什么样的事务,配置事务的传播特性-->
            <tx:method name="add*" propagation="REQUIRED"/>
            <tx:method name="delete*" propagation="REQUIRED"/>
            <tx:method name="update*" propagation="REQUIRED"/>
            <tx:method name="search*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="get*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="find*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
    <!--配置aop织入事务-->
    <aop:config>
        <aop:pointcut id="txPointcut" expression="execution(*
com.ydlclass.service.impl.*.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
    </aop:config>

</beans>
```

#6、web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <!--注册DispatcherServlet，这是springmvc的核心，就是个servlet-->
```

```
    <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:application.xml</param-value>
        </init-param>
        <!--加载时先启动-->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--/ 匹配所有的请求；（不包括.jsp）-->
    <!--/* 匹配所有的请求；（包括.jsp）-->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

Maven项目，application-context.xml、db.properties文件均放置在src/main/resources目录下，Tomcat部署项目，src/main/resources目录下的配置文件默认位置为：{项目名}/WEB-INF/classes，而Spring却在项目根目录下寻找，肯定找不到，因此，配置时指定classpath目录下寻找即可。

解决方案如下：

```
 <context:property-placeholder location="**classpath:db.properties**" />
```

配置文件完

#7、编写entity实体类

```
/**
 * @author IT楠老师
 * @date 2020/5/19
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User implements Serializable{

    private int id;
    private String username;
    private String password;
}
```

#8、UserDao接口编写

```
/**
 * @author IT楠老师
 * @date 2020/5/19
 */
@Mapper
public interface UserMapper {
    /**
     * 根据id查找用户
     * @param id
```

```
     * @return
     */
    User findUserById(int id);

    /**
     * 获取所有的用户
     * @return
     */
    List<User> findAllUsers();
}
```

#9、Mapper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="cn.itnanls.dao.UserMapper">
    <select id="findUserById" resultType="cn.itnanls.entity.User">
      select id,username,password from user where id = #{id}
    </select>

    <select id="findAllUsers" resultType="cn.itnanls.entity.User">
      select id,username,password from user
    </select>
</mapper>
```

#10、编写service

```
/**
 * @author IT楠老师
 * @date 2020/5/19
 */
public interface IUserService {
    /**
     * 获取一个用户的信息
     * @param id
     * @return
     */
    User getUserInfo(int id);

    /**
     * 获取所有用户信息
     * @return
     */
    List<User> getAllUsers();
}

/**
 * @author IT楠老师
 * @date 2020/5/19
 */
@Service
public class UserServiceImpl implements IUserService {

    @Resource
```

```
    private UserMapper userMapper;

    @Override
    public User getUserInfo(int id) {
        return userMapper.findUserById(id);
    }

    @Override
    public List<User> getAllUsers() {
        return userMapper.findAllUsers();
    }
}
```

#11、编写controller

```
/**
 * @author IT楠老师
 * @date 2020/5/19
 */
@Controller
@RequestMapping("/user/")
public class UserController {

    @Resource
    IUserService userService;

    @GetMapping("{id}")
    @ResponseBody
    public User getUserInfo(@PathVariable int id){
        return userService.getUserInfo(id);
    }

    @GetMapping("/all")
    @ResponseBody
    public List<User> getUserInfo(){
        return userService.getAllUsers();
    }

}
```

#12、测试

#13、集成一个德鲁伊

（1）更换数据源

```
<!--配置数据源：数据源有非常多，可以使用第三方的，也可使使用Spring的-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>

    <property name = "filters" value = "${filters}" />
    <!-- 最大并发连接数 -->
```

```
    <property name = "maxActive" value = "${maxActive}" />
    <!-- 初始化连接数量 -->
    <property name = "initialSize" value = "${initialSize}" />
    <!-- 配置获取连接等待超时的时间 -->
    <property name = "maxWait" value = "${maxWait}" />
    <!-- 最小空闲连接数 -->
    <property name = "minIdle" value = "${minIdle}" />
    <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
    <property name = "timeBetweenEvictionRunsMillis" value
="${timeBetweenEvictionRunsMillis}" />
    <!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->
    <property name = "minEvictableIdleTimeMillis" value
="${minEvictableIdleTimeMillis}" />
    <!--        <property name = "validationQuery" value = "${validationQuery}"
/>     -->
    <property name = "testWhileIdle" value = "${testWhileIdle}" />
    <property name = "testOnBorrow" value = "${testOnBorrow}" />
    <property name = "testOnReturn" value = "${testOnReturn}" />
    <property name = "maxOpenPreparedStatements" value
="${maxOpenPreparedStatements}" />
    <!-- 打开 removeAbandoned 功能 -->
    <property name = "removeAbandoned" value = "${removeAbandoned}" />
    <!-- 1800 秒，也就是 30 分钟 -->
    <property name = "removeAbandonedTimeout" value ="${removeAbandonedTimeout}"
/>
    <!-- 关闭 abanded 连接时输出错误日志 -->
    <property name = "logAbandoned" value = "${logAbandoned}" />
</bean>
```

(2) 增加`db.properties`内容

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

filters=wall,stat
maxActive=20
initialSize=3
maxWait=5000
minIdle=3
maxIdle=15
timeBetweenEvictionRunsMillis=60000
minEvictableIdleTimeMillis=300000
validationQuery=SELECT 'x'
testWhileIdle=true
testOnBorrow=false
testOnReturn=false
maxOpenPreparedStatements=20
removeAbandoned=true
removeAbandonedTimeout=1800
logAbandoned=true
```

（3）开启web监控

> 在web.xml中启动web服务

```
<!-- 连接池 启用 Web 监控统计功能    start-->
<filter>
    <filter-name>DruidWebStatFilter</filter-name>
    <filter-class>com.alibaba.druid.support.http.WebStatFilter</filter-class>
    <init-param>
        <param-name>exclusions</param-name>
        <param-value>*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>DruidWebStatFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<servlet>
    <servlet-name>DruidStatView </servlet-name>
    <servlet-class>com.alibaba.druid.support.http.StatViewServlet</servlet-class>
    <init-param>
        <!-- 用户名 -->
        <param-name>loginUsername</param-name>
        <param-value>itnanls</param-value>
    </init-param>
    <init-param>
        <!-- 密码 -->
        <param-name>loginPassword</param-name>
        <param-value>123</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>DruidStatView</servlet-name>
    <url-pattern>/druid/*</url-pattern>
</servlet-mapping>
```

（4）测试，成功。

14、集成日志框架

（1）引入依赖

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.2.3</version>
</dependency>
```

（2）新建配置文件

在类路径下（src/main/resources）新建一个logback.xml文件 这里贴出一个模板，下面会有解释

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="60 seconds" debug="false">
    <!-- 定义参数常量 -->
    <!-- 日志级别 TRACE<DEBUG<INFO<WARN<ERROR -->
```

```
    <!-- logger.trace("msg") logger.debug... -->
    <property name="log.level" value="debug" />
    <property name="log.maxHistory" value="30" />
    <property name="log.filePath" value="D:/log" />
    <property name="log.pattern"
              value="%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} -
%msg%n" />
    <!-- 控制台输出设置 -->
    <appender name="consoleAppender" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>${log.pattern}</pattern>
        </encoder>
    </appender>
    <!-- DEBUG级别文件记录 -->
    <appender name="debugAppender"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 文件路径 -->
        <file>${log.filePath}/debug.log</file>
        <!-- 滚动日志文件类型，就是每天都会有一个日志文件 -->
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- 文件名称 -->
            <fileNamePattern>${log.filePath}/debug/debug.%d{yyyy-MM-dd}.log.gz
            </fileNamePattern>
            <!-- 文件最大保存历史数量 -->
            <maxHistory>${log.maxHistory}</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${log.pattern}</pattern>
        </encoder>
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>DEBUG</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
    </appender>
    <!-- INFO -->
    <appender name="infoAppender"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 文件路径 -->
        <file>${log.filePath}/info.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- 文件名称 -->
            <fileNamePattern>${log.filePath}/info/info.%d{yyyy-MM-dd}.log.gz
            </fileNamePattern>
            <!-- 文件最大保存历史数量 -->
            <maxHistory>${log.maxHistory}</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${log.pattern}</pattern>
        </encoder>
        <filter class="ch.qos.logback.classic.filter.LevelFilter">
            <level>INFO</level>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
    </appender>
```

```
58
59      <!-- com.ydlclass开头的日志对应形式 -->
60      <logger name="com.ydlclass" level="${log.level}" additivity="true">
61          <appender-ref ref="debugAppender"/>
62          <appender-ref ref="infoAppender"/>
63      </logger>
64
65      <!-- <root> 是必选节点，用来指定最基础的日志输出级别，只有一个level属性 -->
66      <root level="info">
67          <appender-ref ref="consoleAppender"/>
68      </root>
69
70      <!-- 捕捉sql开头的日志 -->
71      <appender name="MyBatis"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
72          <file>${log.filePath}/sql_log/mybatis-sql.log</file>
73          <rollingPolicy
    class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
74              <FileNamePattern>${log.filePath}/sql_log/mybatis-sql.log.%d{yyyy-MM-
    dd}</FileNamePattern>
75              <maxHistory>30</maxHistory>
76          </rollingPolicy>
77          <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
78              <pattern>%thread|%d{yyyy-MM-dd
    HH:mm:ss.SSS}|%level|%logger{36}|%m%n</pattern>
79          </encoder>
80      </appender>
81
82      <logger name="sql" level="DEBUG">
83          <appender-ref ref="MyBatis"/>
84      </logger>
85
86  </configuration>
```

mybatis配置文件

```
1   <settings>
2       <setting name="logPrefix" value="sql."/>
3   </settings>
```

（3）使用

> 注意引入的包必须是org.slf4j.Logger和org.slf4j.LoggerFactory

必须定义一个log变量才能打log，参数就填本类的class，这样打印日志才能准确定位啊

```
1   private final Logger log = LoggerFactory.getLogger(UserController.class);
```

```
1   import org.slf4j.Logger;
2   import org.slf4j.LoggerFactory;
3   public class UserController {
4
5       //定义一个log
6       private final Logger log = LoggerFactory.getLogger(UserController.class);
7
8       ....
9       public void ....
```

```
}


//在方法中合理使用log，使用哪个级别看这个日志的重要性
@GetMapping(value = "{id}")
@ResponseBody
public User getUserInfo(@PathVariable Integer id){
    log.info("info");
    log.trace("trace");
    log.debug("debug");
    log.warn("warn");
    log.error("error");

    return userService.getUserInfo(id);
}
```

（4）使用lombok

在类上加注解：

```
@Slf4j
public class UserController {}
```

会在编译的时候自动加上

```
    private final Logger log = LoggerFactory.getLogger(UserController.class);
```

所以这句话就不用写了。

结束

#九、全部的配置文件：

#1、pom文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>ssm-study</artifactId>
        <groupId>org.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>spring-mvc-study</artifactId>
    <packaging>war</packaging>

    <dependencies>
        <!-- 测试相关 -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
```

```
        <!-- springmvc -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.6.RELEASE</version>
        </dependency>
        <!-- servlet -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.2</version>
            <scope>provided</scope>
        </dependency>

        <!--文件上传-->
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.4</version>
        </dependency>
        <!-- jackson -->
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
            <version>2.11.2</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-annotations</artifactId>
            <version>2.11.2</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.11.2</version>
        </dependency>

        <!-- mybatis 相关 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.2</version>
        </dependency>
        <!-- 数据库连接驱动 相关 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.47</version>
        </dependency>

        <!-- 提供了对JDBC操作的完整封装 -->
        <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>5.1.10.RELEASE</version>
        </dependency>
        <!-- 织入 相关 -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.4</version>
        </dependency>
        <!-- spring，mybatis整合包 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>2.0.2</version>
        </dependency>
        <!-- 集成德鲁伊使用 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.18</version>
        </dependency>

        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.3</version>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.12</version>
        </dependency>

        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.1.18</version>
        </dependency>

        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.3</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
```

```
                <version>3.1</version>
                <configuration>
                    <source>1.8</source> <!-- 源代码使用的JDK版本 -->
                    <target>1.8</target> <!-- 需要生成的目标class文件的编译版本 -->
                    <encoding>utf-8</encoding><!-- 字符集编码 -->
                </configuration>
            </plugin>
        </plugins>
        <resources>
            <resource>
                <directory>src/main/java</directory>
                <includes>
                    <include>**/*.properties</include>
                    <include>**/*.xml</include>
                </includes>
                <filtering>false</filtering>
            </resource>
            <resource>
                <directory>src/main/resources</directory>
                <includes>
                    <include>**/*.properties</include>
                    <include>**/*.xml</include>
                </includes>
                <filtering>false</filtering>
            </resource>
        </resources>
    </build>

</project>
```

#2、mybatis-config.xml

```
<?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>
    <settings>
        <setting name="lazyLoadingEnabled" value="true"/>
        <setting name="aggressiveLazyLoading" value="false"/>
        <!-- 下划线转驼峰式 -->
        <setting name="mapUnderscoreToCamelCase" value="true"/>
        <setting name="logPrefix" value="sql."/>
    </settings>

    <typeAliases>
        <package name="cn.itnanls.entity"/>
    </typeAliases>

</configuration>
```

#3、springmvc-servlet.xml

```
<?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:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       https://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/mvc
       https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- 让Spring MVC不处理静态资源 -->
    <mvc:default-servlet-handler />
    <!--  让springmvc自带的注解生效  -->
    <mvc:annotation-driven >
        <mvc:message-converters>
            <bean id="fastjson"
class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter">
                <property name="supportedMediaTypes">
                    <list>
                        <value>text/html;charset=UTF-8</value>
                        <value>application/json;charset=UTF-8</value>
                    </list>
                </property>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>


    <!--文件上传配置-->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!-- 请求的编码格式，必须和jSP的pageEncoding属性一致，以便正确读取表单的内容，默认为
ISO-8859-1 -->
        <property name="defaultEncoding" value="utf-8"/>
        <!-- 上传文件大小上限，单位为字节（10485760=10M） -->
        <property name="maxUploadSize" value="10485760"/>
        <property name="maxInMemorySize" value="40960"/>
    </bean>

    <!-- 处理映射器 -->
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
    <!-- 处理器适配器 -->
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
    <!--视图解析器:DispatcherServlet给他的ModelAndView-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/page/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
```

```
    </bean>
</beans>
```

#4、db.properties

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

filters=wall,stat
maxActive=20
initialSize=3
maxWait=5000
minIdle=3
maxIdle=15
timeBetweenEvictionRunsMillis=60000
minEvictableIdleTimeMillis=300000
validationQuery=SELECT 'x'
testWhileIdle=true
testOnBorrow=false
testOnReturn=false
maxOpenPreparedStatements=20
removeAbandoned=true
removeAbandonedTimeout=1800
logAbandoned=true
```

#5、application.xml

```
<?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:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       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
       https://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">

    <!-- 加载外部的数据库信息 classpath:不叫会报错具体原因下边解释-->
    <context:property-placeholder location="classpath:db.properties"/>
    <!-- 加入springmvc的配置 -->
    <import resource="classpath:springmvc-servlet.xml"/>

    <context:component-scan base-package="cn.itnanls"/>

    <!-- Mapper 扫描器 -->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!-- 扫描 cn.wmyskxz.mapper 包下的组件 -->
        <property name="basePackage" value="cn.itnanls.dao"/>
```

```
    </bean>

    <!--配置数据源：数据源有非常多，可以使用第三方的，也可使使用Spring的-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>

        <property name = "filters" value = "${filters}" />
        <!-- 最大并发连接数 -->
        <property name = "maxActive" value = "${maxActive}" />
        <!-- 初始化连接数量 -->
        <property name = "initialSize" value = "${initialSize}" />
        <!-- 配置获取连接等待超时的时间 -->
        <property name = "maxWait" value = "${maxWait}" />
        <!-- 最小空闲连接数 -->
        <property name = "minIdle" value = "${minIdle}" />
        <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
        <property name = "timeBetweenEvictionRunsMillis" value
="${timeBetweenEvictionRunsMillis}" />
        <!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->
        <property name = "minEvictableIdleTimeMillis" value
="${minEvictableIdleTimeMillis}" />
        <!--        <property name = "validationQuery" value =
"${validationQuery}" />     -->
        <property name = "testWhileIdle" value = "${testWhileIdle}" />
        <property name = "testOnBorrow" value = "${testOnBorrow}" />
        <property name = "testOnReturn" value = "${testOnReturn}" />
        <property name = "maxOpenPreparedStatements" value
="${maxOpenPreparedStatements}" />
        <!-- 打开 removeAbandoned 功能 -->
        <property name = "removeAbandoned" value = "${removeAbandoned}" />
        <!-- 1800 秒，也就是 30 分钟 -->
        <property name = "removeAbandonedTimeout" value
="${removeAbandonedTimeout}" />
        <!-- 关闭 abanded 连接时输出错误日志 -->
        <property name = "logAbandoned" value = "${logAbandoned}" />
    </bean>

    <!--配置SqlSessionFactory-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <!--关联Mybatis-->
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <property name="mapperLocations" value="classpath:mappers/*.xml"/>
    </bean>

    <!--注册sqlSessionTemplate , 关联sqlSessionFactory-->
    <bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
        <!--利用构造器注入-->
        <constructor-arg index="0" ref="sqlSessionFactory"/>
    </bean>

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

```
    </bean>


    <!--配置事务通知-->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!--配置哪些方法使用什么样的事务,配置事务的传播特性-->
            <tx:method name="add*" propagation="REQUIRED"/>
            <tx:method name="delete*" propagation="REQUIRED"/>
            <tx:method name="update*" propagation="REQUIRED"/>
            <tx:method name="search*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="get*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="find*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="*" propagation="REQUIRED"/>
        </tx:attributes>
    </tx:advice>
    <!--配置aop织入事务-->
    <aop:config>
        <aop:pointcut id="txPointcut" expression="execution(*
  cn.itnanls.service.impl.*.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
    </aop:config>

</beans>
```

#6、logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="60 seconds" debug="false">
    <!-- 定义参数常量 -->
    <!-- 日志级别 TRACE<DEBUG<INFO<WARN<ERROR -->
    <!-- logger.trace("msg") logger.debug... -->
    <property name="log.level" value="debug" />
    <property name="log.maxHistory" value="30" />
    <property name="log.filePath" value="D:/log" />
    <property name="log.pattern"
              value="%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} -
%msg%n" />
    <!-- 控制台输出设置 -->
    <appender name="consoleAppender" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>${log.pattern}</pattern>
        </encoder>
    </appender>
    <!-- DEBUG级别文件记录 -->
    <appender name="debugAppender"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <!-- 文件路径 -->
        <file>${log.filePath}/debug.log</file>
        <!-- 滚动日志文件类型，就是每天都会有一个日志文件 -->
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- 文件名称 -->
            <fileNamePattern>${log.filePath}/debug/debug.%d{yyyy-MM-dd}.log.gz
            </fileNamePattern>
            <!-- 文件最大保存历史数量 -->
            <maxHistory>${log.maxHistory}</maxHistory>
```

```
28          </rollingPolicy>
29          <encoder>
30              <pattern>${log.pattern}</pattern>
31          </encoder>
32          <filter class="ch.qos.logback.classic.filter.LevelFilter">
33              <level>DEBUG</level>
34              <onMatch>ACCEPT</onMatch>
35              <onMismatch>DENY</onMismatch>
36          </filter>
37      </appender>
38      <!-- INFO -->
39      <appender name="infoAppender"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
40          <!-- 文件路径 -->
41          <file>${log.filePath}/info.log</file>
42          <rollingPolicy
    class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
43              <!-- 文件名称 -->
44              <fileNamePattern>${log.filePath}/info/info.%d{yyyy-MM-dd}.log.gz
45              </fileNamePattern>
46              <!-- 文件最大保存历史数量 -->
47              <maxHistory>${log.maxHistory}</maxHistory>
48          </rollingPolicy>
49          <encoder>
50              <pattern>${log.pattern}</pattern>
51          </encoder>
52          <filter class="ch.qos.logback.classic.filter.LevelFilter">
53              <level>INFO</level>
54              <onMatch>ACCEPT</onMatch>
55              <onMismatch>DENY</onMismatch>
56          </filter>
57      </appender>
58
59      <!-- com.ydlclass开头的日志对应形式 -->
60      <logger name="com.ydlclass" level="${log.level}" additivity="true">
61          <appender-ref ref="debugAppender"/>
62          <appender-ref ref="infoAppender"/>
63      </logger>
64
65      <!-- <root> 是必选节点，用来指定最基础的日志输出级别，只有一个level属性 -->
66      <root level="info">
67          <appender-ref ref="consoleAppender"/>
68      </root>
69
70      <!-- 捕捉sql开头的日志 -->
71      <appender name="MyBatis"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
72          <file>${log.filePath}/sql_log/mybatis-sql.log</file>
73          <rollingPolicy
    class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
74              <FileNamePattern>${log.filePath}/sql_log/mybatis-sql.log.%d{yyyy-MM-
    dd}</FileNamePattern>
75              <maxHistory>30</maxHistory>
76          </rollingPolicy>
77          <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
78              <pattern>%thread|%d{yyyy-MM-dd
    HH:mm:ss.SSS}|%level|%logger{36}|%m%n</pattern>
79          </encoder>
```

```
80        </appender>
81
82        <logger name="sql" level="DEBUG">
83            <appender-ref ref="MyBatis"/>
84        </logger>
85
86    </configuration>
```