MVC 体系结构
    三层架构
        表现层：
        业务层：
        持久层：
SpringMVC 是什么?
SpringMVC开发流程:
SpringMVC请求处理过程
        流程图
        流程说明
Spring MVC 九大组件
        HandlerMapping（处理器映射器）
        HandlerAdapter（处理器适配器）
        HandlerExceptionResolver
        ViewResolve
        RequestToViewNameTranslator
        LocaleResolver
        ThemeResolver
        MultipartResolver
        FlashMapManager
url-pattern配置和原理剖析:
ModelAndView、Model、ModelMap用法
请求参数绑定
SpringMVC请求参数示例
        简单数据类型参数
        绑定Pojo类型参数
        绑定Pojo包装对象参数（嵌套pojo）
        前端传日期参数：需自己注册时间转换器
理解Rest风格请求
        什么是 REST
        Restful 的优点
        Restful 的特性
        RESTful 的示例
GET、POST请求乱码解决
@RequestBody回顾
@ResponseBody 回顾
拦截器(Inteceptor)使用
        监听器示例：https://www.cnblogs.com/ygj0930/p/6374384.html
拦截器示例
多个拦截器执行示例
SpringMVC文件上传分析
SpringMVC文件上传代码:
SpringMVC异常处理机制:
重定向和转发的区别:
SpringMVC核心源码流程
Spring MVC 必备设计模式
Spring Data JPA 框架简介
Spring Data JPA，JPA规范和Hibernate之间的关系
JPQL示例
SpringMVC 的控制器是不是单例模式,如果是,有什么问题,怎么解决?
怎么样把 ModelMap 里面的数据放入 Session 里面?
SpringMVC 怎么和 AJAX 相互调用的?
Get和Post的区别?
SpringMVC使用适配器模式:
SpringMVC是怎么解决并发问题的?
SpringMVC中的拦截器和Servlet中的Filter有什么区别?

springmvc怎么前后端分离：
SpringMVC的常用注解?

MVC 体系结构

三层架构

我们的开发架构一般都是基于两种形式，一种是 C/S 架构，也就是客户端/服务器；另一种是 B/S 架构，也就是浏览器服务器。在 JavaEE 开发中，几乎全都是基于 B/S 架构的开发。那么在 B/S 架构中，系统标准的三层架构包括：表现层、业务层、持久层。三层架构在我们的实际开发中使用的非常多，所以我们课程中的案例也都是基于三层架构设计的。

三层架构中，每一层各司其职，接下来我们就说说每层都负责哪些方面：

表现层：

也就是我们常说的web 层。它负责接收客户端请求，向客户端响应结果，通常客户端使用http 协议请求web 层，web 需要接收 http 请求，完成 http 响应。

表现层包括展示层和控制层：控制层负责接收请求，展示层负责结果的展示。
表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。
表现层的设计一般都使用 MVC 模型。（MVC 是表现层的设计模型，和其他层没有关系）

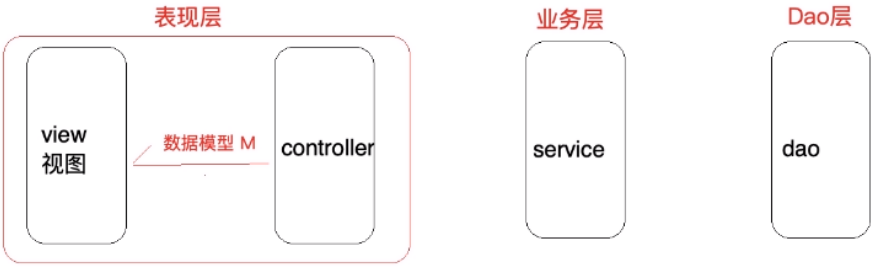
业务层：

也就是我们常说的 service 层。它负责业务逻辑处理，和我们开发项目的需求息息相关。web 层依赖业务层，但是业务层不依赖 web 层。
业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务一致性。（也就是我们说的， 事务应该放到业务层来控制）

持久层：

也就是我们是常说的 dao 层。负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库中。通俗的讲，持久层就是和数据库交互，对数据库表进行增删改查的。

经典三层（代码架构）

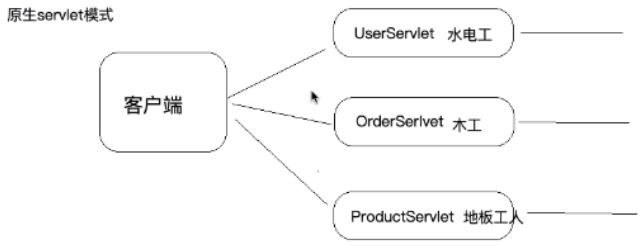


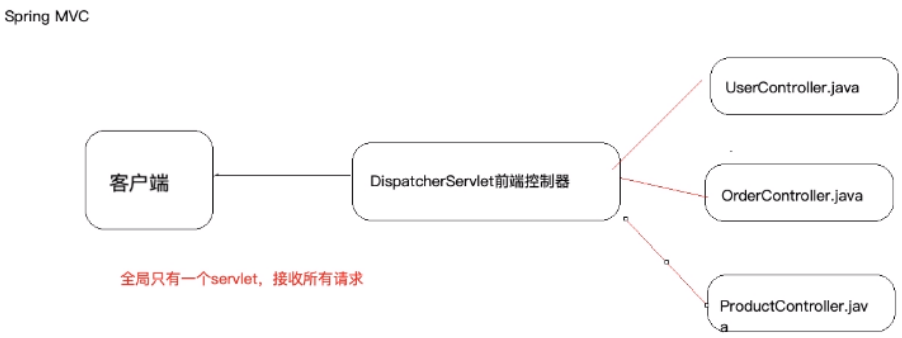
MVC 模式（代码的组织方式/形式）
M model模型（数据模型[pojo、vo、po]+业务模型[业务逻辑]）
V view视图（jsp、html）
C controller控制器（servlet）

Spring MVC 框架是一个应用于表现层的框架

SpringMVC 是什么?

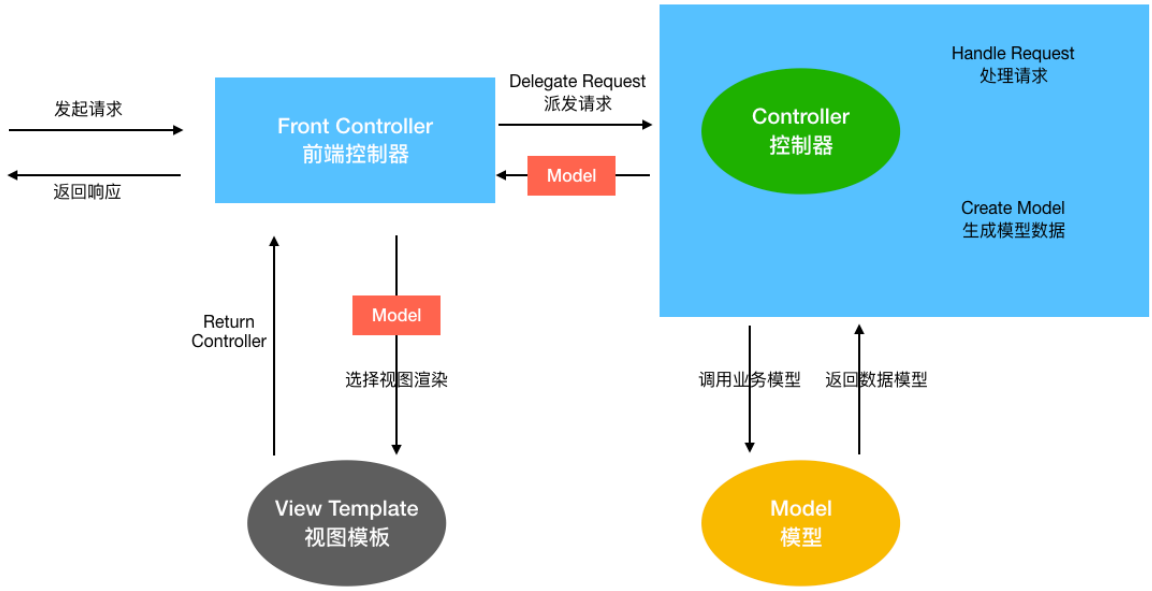
SpringMVC和原生Servlet的区别：它通过一套注解，让一个简单的 Java 类成为处理请求的控制器，而无须实现任何接口。





Spring MVC和Struts2一样，都是 为了解决表现层问题 的Web框架，它们都是基于 MVC 设计模式的。而这些表现层框架的主要职责就是处理前端HTTP请求。 Spring MVC 本质可以认为是对servlet的封装，简化了我们serlvet的开发 作用：1）接收请求 2）返回响应，跳转页面

by 应癫



SpringMVC开发流程：

1）配置DispatcherServlet前端控制器

2）开发处理具体业务逻辑的Handler（@Controller、@RequestMapping）

3）xml配置文件配置controller扫描，配置SpringMVC三大件

4）将xml文件路径告诉SpringMVC（DispatcherServlet）

web.xml配置：

```
<web-app>
  <display-name>Archetype Created Web Application</display-name>


  <!--SpringMVC提供的针对post请求的编码过滤器-->
  <filter>
    <filter-name>encoding</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>


  <!--配置SpringMVC请求方式转换过滤器，会检查请求参数中是否有_method参数，如果有就按照指定的请求方式进行转换-->
  <filter>
    <filter-name>hiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
  </filter>


  <filter-mapping>
```

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


  <filter-mapping>
    <filter-name>hiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <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:springmvc.xml</param-value>//配置相关xml文件
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <!--拦截匹配规则的url请求，进入SpringMVC框架处理-->
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>

```

总结：

以上拦截url请求有三种方式：

1. 带后缀，比如*.action *.do *.aaa，拦截这些后缀的请求。
2. 配置/，这种方式不会拦截jsp，但是会拦截.html等静态资源（静态资源：除了servlet和jsp之外的js、css、png等）。

   为什么会拦截静态资源： 因为tomcat容器中有一个web.xml（父），你的项目中也有一个web.xml（子），是一个继承关系，父web.xml中有一个DefaultServlet，url-pattern 是一个 /，此时我们自己的web.xml中也配置了一个 / ,覆写了父web.xml的配置

   为什么不拦截.jsp： 因为父web.xml中有一个JspServlet，这个servlet拦截.jsp文件，而我们并没有覆写这个配置，所以SpringMVC此时不拦截jsp，jsp的处理交给了tomcat。
3. 配置/*，这种方式拦截所有，包括jsp。

```
  <!--配置SpringMVC请求方式转换过滤器，会检查请求参数中是否有_method参数，如果有就按照指定的请求方式进
行转换-->
  <filter>
    <filter-name>hiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
  </filter>
```

如上图，_method参数是运用在前端指定请求类型，如下代码：

```
请求方式value的值为 GET、POST、 HEAD、OPTIONS、PUT、DELETE、TRACE中的一个。
<form action="..." method="post">
        <input type="hidden" name="_method" value="put" />
</form>
```

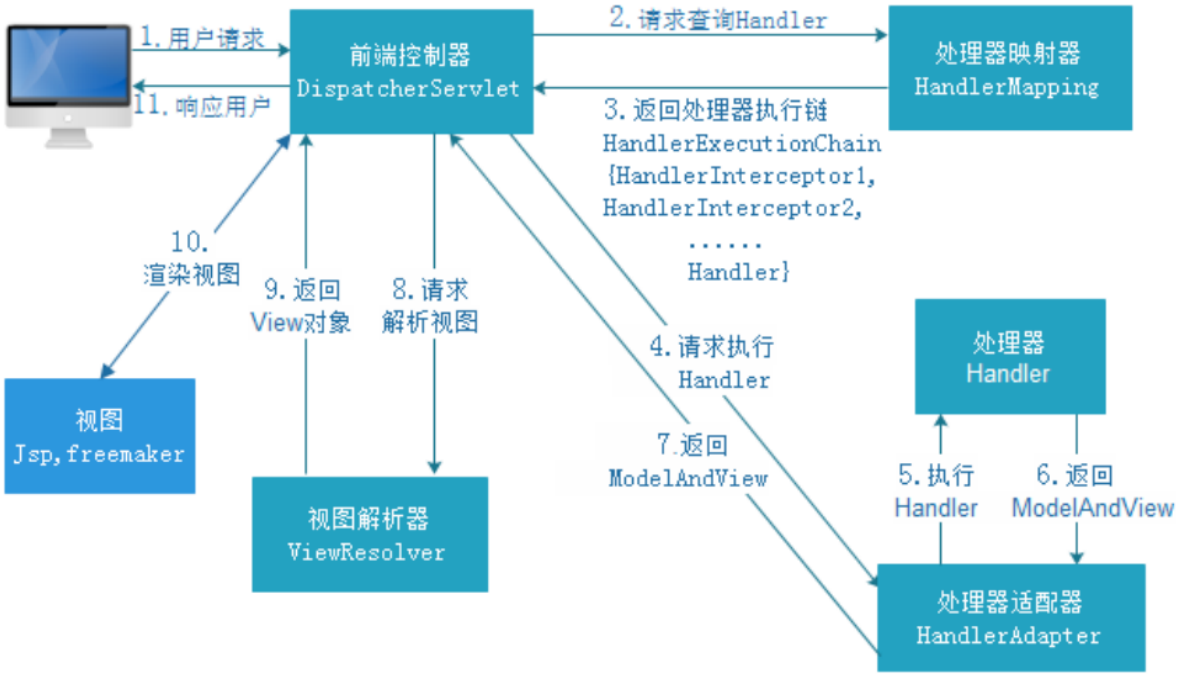
SpringMVC.xml配置：

```
<!--开启controller扫描-->
    <context:component-scan base-package="com.lagou.edu.controller"/>


    <!--配置SpringMVC的视图解析器-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
    <!--配置这个就不用加前缀-->
    modelAndView.setViewName("/WEB-INF/jsp/success.jsp");
    <!--而是使用-->
    modelAndView.setViewName("success");

    <!--
        自动注册最合适的处理器映射器，处理器适配器(调用handler方法)
    -->
    <mvc:annotation-driven conversion-service="conversionServiceBean"/>


    <!--注册自定义类型转换器-->
    <bean id="conversionServiceBean"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="com.lagou.edu.converter.DateConverter"></bean>
            </set>
        </property>
    </bean>


    <!--静态资源配置，方案一-->
    <!--
        原理：添加该标签配置之后，会在SpringMVC上下文中定义一个
DefaultServletHttpRequestHandler对象
             这个对象如同一个检查人员，对进入DispatcherServlet的url请求进行过滤筛查，如
果发现是一个静态资源请求
             那么会把请求转由web应用服务器（tomcat）默认的DefaultServlet来处理，如果不
是静态资源请求，那么继续由
             SpringMVC框架处理
    -->
    <!--<mvc:default-servlet-handler/>-->


    <!--静态资源配置，方案二，SpringMVC框架自己处理静态资源
        mapping:约定的静态资源的url规则
        location：指定的静态资源的存放位置

    -->
    <mvc:resources location="classpath:/" mapping="/resources/**"/>
    <mvc:resources location="/WEB-INF/js/" mapping="/js/**"/>


    <mvc:interceptors>
        <!--拦截所有handler-->
        <!--<bean class="com.lagou.edu.interceptor.MyIntercepter01"/>-->

        <mvc:interceptor>
```

```
55              <!--配置当前拦截器的url拦截规则，**代表当前目录下及其子目录下的所有url-->
56              <mvc:mapping path="/**"/>
57              <!--exclude-mapping可以在mapping的基础上排除一些url拦截-->
58              <!--<mvc:exclude-mapping path="/demo/**"/>-->
59              <bean class="com.lagou.edu.interceptor.MyIntercepter01"/>
60          </mvc:interceptor>
61
62
63          <mvc:interceptor>
64              <mvc:mapping path="/**"/>
65              <bean class="com.lagou.edu.interceptor.MyIntercepter02"/>
66          </mvc:interceptor>
67
68      </mvc:interceptors>
69
70
71      <!--多元素解析器
72          id固定为multipartResolver
73      -->
74      <bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
75          <!--设置上传文件大小上限，单位是字节，-1代表没有限制也是默认的-->
76          <property name="maxUploadSize" value="5000000"/>
77      </bean>
```

SpringMVC请求处理过程

流程图



流程说明

第一步：用户发送请求至前端控制器DispatcherServlet（就是负责分发控制流程的作用）
第二步：DispatcherServlet收到请求调用HandlerMapping处理器映射器
第三步：处理器映射器根据请求Url找到具体的Handler（后端控制器），生成处理器对象及处理器拦截器(如果 有则生成)一并返回DispatcherServlet
第四步：DispatcherServlet调用HandlerAdapter处理器适配器去调用Handler
第五步：处理器适配器执行Handler
第六步：Handler执行完成给处理器适配器返回ModelAndView
第七步：处理器适配器向前端控制器返回 ModelAndView，ModelAndView 是SpringMVC 框架的一个底层对象，包括 Model 和 View
第八步：前端控制器请求视图解析器去进行视图解析，根据逻辑视图名来解析真正的视图。
第九步：视图解析器向前端控制器返回View
第十步：前端控制器进行视图渲染，就是将模型数据（在 ModelAndView 对象中）填充到 request 域
第十一步：前端控制器向用户响应结果

Spring MVC 九大组件

HandlerMapping（处理器映射器）

HandlerMapping 是用来查找 Handler 的，也就是处理器，具体的表现形式可以是类，也可以是方法。比如，标注了@RequestMapping的每个方法都可以看成是一个Handler。Handler负责具体实际的请求处理，在请求到达后，HandlerMapping 的作用便是找到请求相应的处理器Handler 和 Interceptor.（处理handler（相当于一个@RequestMapping）和url的关系）；

HandlerAdapter（处理器适配器）

HandlerAdapter 是一个适配器。因为 Spring MVC 中 Handler 可以是任意形式的，只要能处理请求即可。但是把请求交给 Servlet 的时候，由于 Servlet 的方法结构都doService(HttpServletRequest req,HttpServletResponse resp)形式的，要让固定的 Servlet 处理方法调用 Handler 来进行处理，便是 HandlerAdapter 的职责。

HandlerExceptionResolver

HandlerExceptionResolver 用于处理 Handler 产生的异常情况。它的作用是根据异常设置 ModelAndView，之后交给渲染方法进行渲染，渲染方法会将 ModelAndView 渲染成页面。

ViewResolve

ViewResolver即视图解析器，用于将String类型的视图名和Locale解析为View类型的视图，只有一个resolveViewName()方法。从方法的定义可以看出，Controller层返回的String类型视图名 viewName 最终会在这里被解析成为View。View是用来渲染页面的，也就是说，它会将程序返回的参数和数据填入模板中，生成html文件。ViewResolver 在这个过程主要完成两件事情：ViewResolver 找到渲染所用的模板（第一件大事）和所用的技术（第二件大事，其实也就是找到视图的类型，如JSP）并填入参数。默认情况下，Spring MVC会自动为我们配置一个 InternalResourceViewResolver,是针对 JSP 类型视图的。（拼接前置的后缀，不用每次都写路径和.jsp，如下图）

```
<!--配置SpringMVC的视图解析器-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

RequestToViewNameTranslator

RequestToViewNameTranslator 组件的作用是从请求中获取 ViewName.因为 ViewResolver 根据 ViewName 查找 View，但有的 Handler 处理完成之后,没有设置 View，也没有设置 ViewName，便要通过这个组件从请求中查找 ViewName。(把逻辑的得到的值当成url，比如上面success会出现在浏览器路径中)

LocaleResolver

ViewResolver 组件的 resolveViewName 方法需要两个参数，一个是视图名，一个是 Locale。LocaleResolver 用于从请求中解析出 Locale，比如中国 Locale 是 zh-CN，用来表示一个区域。这个组件也是 i18n 的基础。（国际化处理区域，不重要）

ThemeResolver

ThemeResolver 组件是用来解析主题的。主题是样式、图片及它们所形成的显示效果的集合。Spring MVC 中一套主题对应一个 properties文件，里面存放着与当前主题相关的所有资源，如图片、CSS样式等。创建主题非常简单，只需准备好资源，然后新建一个“主题名.properties”并将资源设置进去，放在classpath下，之后便可以在页面中使用了。SpringMVC中与主题相关的类有ThemeResolver、ThemeSource和Theme。ThemeResolver负责从请求中解析出主题名，ThemeSource根据主题名找到具体的主题，其抽象也就是Theme，可以通过Theme来获取主题和具体的资源。

MultipartResolver

MultipartResolver 用于上传请求，通过将普通的请求包装成 MultipartHttpServletRequest 来实现。MultipartHttpServletRequest 可以通过 getFile() 方法 直接获得文件。如果上传多个文件，还可以调用 getFileMap()方法得到Map<FileName，File>这样的结构，MultipartResolver 的作用就是封装普通的请求，使其拥有文件上传的功能。(上传请求,可用于文件上传)

FlashMapManager

FlashMap 用于重定向时的参数传递，比如在处理用户订单时候，为了避免重复提交，可以处理完post请求之后重定向到一个get请求，这个get请求可以用来显示订单详情之类的信息。这样做虽然可以规避用户重新提交订单的问题，但是在这个页面上要显示订单的信息，这些数据从哪里来获得呢？因为重定向时么有传递参数这一功能的，如果不想把参数写进URL（不推荐），那么就可以通过FlashMap来传递。只需要在重定向之前将要传递的数据写入请求（可以通过ServletRequestAttributes.getRequest()方法获得）的属性OUTPUT_FLASH_MAP_ATTRIBUTE中，这样在重定向之后的Handler中Spring就会自动将其设置到Model中，在显示订单信息的页面上就可以直接从Model中获取数据。FlashMapManager 就是用来管理 FalshMap 的。（用于重定向时的参数传递）

url-pattern配置和原理剖析：

```
<web-app>
 <servlet-mapping>
    <servlet-name>springmvc</servlet-name>

    <!--
      方式一：带后缀，比如*.action  *.do *.aaa
              该种方式比较精确、方便，在以前和现在企业中都有很大的使用比例
```

```
      方式二：/ 不会拦截 .jsp，但是会拦截.html等静态资源（静态资源：除了servlet和jsp之外的js、css、png等）

            为什么配置为/ 会拦截静态资源？？？
                因为tomcat容器中有一个web.xml（父），你的项目中也有一个web.xml（子），是一个继承关系
                        父web.xml中有一个DefaultServlet， url-pattern 是一个 /
                        此时我们自己的web.xml中也配置了一个 / ,覆写了父web.xml的配置
            为什么不拦截.jsp呢？
                因为父web.xml中有一个JspServlet，这个servlet拦截.jsp文件，而我们并没有覆写这个配置，所以SpringMVC此时不拦截jsp，jsp的处理交给了tomcat

            如何解决/拦截静态资源这件事？
            有两种：具体看下面文件

      方式三：/* 拦截所有，包括.jsp，如果是jsp，他会返回/WEB-INF/jsp/success.jsp，然后再去找RequestMapping，肯定是错的
    -->
    <!--拦截匹配规则的url请求，进入SpringMVC框架处理-->
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

```
<!--静态资源配置，方案一-->
    <!--原理：添加该标签配置之后，会在SpringMVC上下文中定义一个DefaultServletHttpRequestHandler对象，这个对象如同一个检查人员，对进入DispatcherServlet的url请求进行过滤筛查，如果发现是一个静态资源请求那么会把请求转由web应用服务器（tomcat）默认的DefaultServlet来处理，如果不是静态资源请求，那么继续由SpringMVC框架处理-->

    <!--<mvc:default-servlet-handler/>该方法有局限，页面只能放在webapp文件夹下，不能放在WEB-INF下面-->

    <!--静态资源配置，方案二，SpringMVC框架自己处理静态资源
        mapping:约定的静态资源的url规则
        location：指定的静态资源的存放位置？相当于项目文件的resources文件夹下
        mapping="/resources/**"：路径带有这个就去上面的location去找
    -->
    <mvc:resources location="classpath:/"  mapping="/resources/**"/>
    <mvc:resources location="/WEB-INF/js/" mapping="/js/**"/>

```

ModelAndView、Model、ModelMap用法

ModelAndView包括参数、跳转路径；Model、ModelMap、Map只。

```
  //ModelAndView方式
    @RequestMapping("/handle01")
    public ModelAndView handle01(@ModelAttribute("name") String name) {

        int c = 1/0;


        Date date = new Date();// 服务器时间
```

```
        // 返回服务器时间到前端页面
        // 封装了数据和页面信息的 ModelAndView
        ModelAndView modelAndView = new ModelAndView();
        // addObject 其实是向请求域中request.setAttribute("date",date);
        modelAndView.addObject("date",date);
        // 视图信息(封装跳转的页面信息) 逻辑视图名
        modelAndView.setViewName("success");
        return modelAndView;
    }
    //ModelMap方式
    @RequestMapping("/handle11")
    public String handle11(ModelMap modelMap) {
        Date date = new Date();// 服务器时间
        modelMap.addAttribute("date",date);
        System.out.println("=================modelmap:" + modelMap.getClass());
        return "success";
    }
    //直接声明形参Model，封装数据
    @RequestMapping("/handle12")
    public String handle12(Model model) {
        Date date = new Date();
        model.addAttribute("date",date);
        System.out.println("=================model:" + model.getClass());
        return "success";
    }
   //直接声明形参Map集合，封装数据
   @RequestMapping("/handle13")
    public String handle13(Map<String,Object> map) {
        Date date = new Date();
        map.put("date",date);
        System.out.println("=================map:" + map.getClass());
        return "success";
    }
   /**

SpringMVC在handler方法上传入Map、Model和ModelMap参数，并向这些参数中保存数据（放入  到请求域），都可以在页面获取到

它们之间是什么关系？

运行时的具体类型都是BindingAwareModelMap，相当于给BindingAwareModelMap中保存的数据都会放在请求域中

Map(jdk中的接口) Model（spring的接口） ModelMap(class,实现Map接口)

BindingAwareModelMap继承了ExtendedModelMap，ExtendedModelMap继承了ModelMap,实现了Model接口
      **/
```

请求参数绑定

请求参数绑定：说白了SpringMVC如何接收请求参数

http协议（超文本传输协议）
原生servlet接收一个整型参数：
1）String ageStr = request.getParameter("age");
2) Integer age = Integer.parseInt(ageStr);

SpringMVC框架对Servlet的封装，简化了servlet的很多操作
SpringMVC在接收整型参数的时候，直接在Handler方法中声明形参即可
@RequestMapping("xxx")
public String handle(Integer age) {
System.out.println(age);
}
参数绑定：取出参数值绑定到handler方法的形参上（原生servlet需要通过方法得到参数，而mvc不用是因为mvc框架内部通过反射将参数值绑定到handler方法的形参上）

SpringMVC请求参数示例

简单数据类型参数

```
//SpringMVC对原生servlet api是支持的
///url: /demo/handle03?id=1
@RequestMapping("/handle02")
    public ModelAndView handle02(HttpServletRequest request, HttpServletResponse response,HttpSession session) {
        String id = request.getParameter("id");//原生servlet方式
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("date",date);
        modelAndView.setViewName("success");
        return modelAndView;
    }
   /**
     * url: /demo/handle03?id=1
     * 要求: 传递的参数名和声明的形参名称保持一致
     * @RequestParam的意义是传递的参数名和声明的形参名称不一致时使用
     * 对于布尔类型的参数，请求的参数值为true或false。或者1或0
     */
  @RequestMapping("/handle03")
    public ModelAndView handle03(@RequestParam("ids") Integer id,Boolean flag) {

        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("date",date);
        modelAndView.setViewName("success");
        return modelAndView;
    }

```

绑定Pojo类型参数

```
/*
     * SpringMVC接收pojo类型参数  url: /demo/handle04?id=1&username=zhangsan
     * 接收pojo类型参数，直接形参声明即可，类型就是Pojo的类型，形参名user无所谓
     * 但是要求传递的参数名必须和Pojo的属性名保持一致（通过反射调用set方法）
     */
    @RequestMapping("/handle04")
    public ModelAndView handle04(User user) {
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();

        modelAndView.setViewName("success");

```

```
        return modelAndView;
    }

```

绑定Pojo包装对象参数（嵌套pojo）

```
/*
     * SpringMVC接收pojo包装类型参数  url：/demo/handle05?user.id=1&user.username=zhangsan
     * 不管包装Pojo与否，它首先是一个pojo，那么就可以按照上述pojo的要求来
     * 1、绑定时候直接形参声明即可
     * 2、传参参数名和pojo属性保持一致，如果不能够定位数据项，那么通过属性名 + "." 的方式进一步锁定数据
     *
     */
    @RequestMapping("/handle05")
    public ModelAndView handle05(QueryVo queryVo) {
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("date",date);
        modelAndView.setViewName("success");
        return modelAndView;
    }

public class QueryVo {
 private String mail;
 private String phone;
 // 嵌套了另外的Pojo对象
 private User user;
}
```

前端传日期参数：需自己注册时间转换器

```
    /**
     * url：/demo/handle06?birthday=2019-10-08
     * 绑定日期类型参数
     * 定义一个SpringMVC的类型转换器  接口，扩展实现接口接口，注册你的实现
     * 意思是前端传的参数是2019-10-08格式，你得转成Date类型才能接收到
     */
    @RequestMapping("/handle06")
    public ModelAndView handle06(Date birthday) {
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("date",date);
        modelAndView.setViewName("success");
        return modelAndView;
    }
```

```
   /**
     * 自定义类型转换器
     * S：source，源类型
     * T：target：目标类型
     */
    public class DateConverter implements Converter<String, Date> {
        @Override
        public Date convert(String source) {
            // 完成字符串向日期的转换
```

```
            SimpleDateFormat simpleDateFormat = new
                    SimpleDateFormat("yyyy-MM-dd");
            try {
                Date parse = simpleDateFormat.parse(source);
                return parse;
            } catch (ParseException e) {
                e.printStackTrace();
            }
            return null;
        }
    }
}
```

```
    <!--
        自动注册最合适的处理器映射器，处理器适配器(调用handler方法)
    -->
    <mvc:annotation-driven conversion-service="conversionServiceBean"/>

    <!--注册自定义类型转换器-->
    <bean id="conversionServiceBean"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="com.lagou.edu.converter.DateConverter"></bean>
                <bean>...</bean>//可配置多个转换器
            </set>
        </property>
    </bean>
```

理解Rest风格请求

什么是 REST

REST（英文：Representational State Transfer，简称 REST）描述了一个架构样式的网络系统， 比如 web 应用程序。它首次出现在 2000 年 Roy Fielding 的博士论文中，他是 HTTP 规范的主要编写者之一。在目前主流的三种 Web 服务交互方案中，REST 相比于 SOAP（Simple Object Access protocol，简单对象访问协议）以及 XML-RPC 更加简单明了，无论是对 URL 的处理还是对 Payload 的编码，REST 都倾向于用更加简单轻量的方法设计和实现。值得注意的是 REST 并没有一个明确的标准，而更像
是一种设计的风格。
它本身并没有什么实用性，其核心价值在于如何设计出符合 REST 风格的网络接口。
资源 表现层 状态转移

Restful 的优点

它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

Restful 的特性

资源（Resources）：网络上的一个实体，或者说是网络上的一个具体信息。
它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。可以用一个 URI（统一资源定位符）指向它，每种资源对应一个特定的 URI 。要获取这个资源，访问它的 URI 就可以，因此URI 即为每一个资源的独一无二的识别符。
表现层（Representation）：把资源具体呈现出来的形式，叫做它的表现层 （Representation）。比如，

文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式。

状态转化（State Transfer）：每发出一个请求，就代表了客户端和服务器的一次交互过程。

HTTP 协议，是一个无状态协议，即所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转化”（State Transfer）。而这种转化是建立在表现层之上的，所以就是“表现层状态转化”。具体说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源，PUT 用来更新资源，DELETE 用来删除资源

RESTful 的示例

rest是一个url请求的风格，基于这种风格设计请求的url
没有rest的话，原有的url设计
http://localhost:8080/user/queryUserById.action?id=3
url中定义了动作（操作），参数具体锁定到操作的是谁
有了rest风格之后
rest中，认为互联网中的所有东西都是资源，既然是资源就会有一个唯一的uri标识它，代表它
http://localhost:8080/user/3 代表的是id为3的那个用户记录（资源）

```
/*
 * restful  get   /demo/handle/15
 * 注解的使用@PathVariable，可以帮助我们从uri中取出参数
 */
@RequestMapping(value = "/handle/{id}",method = {RequestMethod.GET})
public ModelAndView handleGet(@PathVariable("id") Integer id) {

    Date date = new Date();
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("date",date);
    modelAndView.setViewName("success");
    return modelAndView;
}

/*
 * restful  post  /demo/handle
 */
@RequestMapping(value = "/handle",method = {RequestMethod.POST})
public ModelAndView handlePost(String username) {

    Date date = new Date();
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("date",date);
    modelAndView.setViewName("success");
    return modelAndView;
}
```

锁定资源之后如何操作它呢？常规操作就是增删改查
根据请求方式不同，代表要做不同的操作
get 查询，获取资源
post 增加，新建资源
put 更新
delete 删除资源

配置get、post、put、delete如下图：

```
    <form method="post" action="/demo/handle">
        <input type="text" name="username"/>
        <input type="submit" value="提交rest_post请求"/>
    </form>


    <form method="post" action="/demo/handle/15/lisi">
        <input type="hidden" name="_method" value="put"/>
        <input type="submit" value="提交rest_put请求"/>
    </form>


    <form method="post" action="/demo/handle/15">
        <input type="hidden" name="_method" value="delete"/>
        <input type="submit" value="提交rest_delete请求"/>
    </form>
</fieldset>
```

```
<filter>
  <filter-name>encoding</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>


<!--配置springmvc请求方式转换过滤器，会检查请求参数中是否有_method参数，如果有就按照指定的请求方式进行转换-->
<filter>
  <filter-name>hiddenHttpMethodFilter</filter-name>
  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>


<filter-mapping>
  <filter-name>encoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>


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

rest风格带来的直观体现：就是传递参数方式的变化，参数可以在uri中了

/account/1 HTTP GET ： 得到 id = 1 的 account

/account/1 HTTP DELETE：删除 id = 1 的 account

/account/1 HTTP PUT：更新 id = 1 的 account

URL：资源定位符，通过URL地址去定位互联网中的资源（抽象的概念，比如图片、视频、app服务等）。

RESTful 风格 URL：互联网所有的事物都是资源，要求URL中只有表示资源的名称，没有动词。

RESTful风格资源操作：使用HTTP请求中的method方法put、delete、post、get来操作资源。分别对应添加、删除、修改、查询。不过一般使用时还是 post 和 get。put 和 delete几乎不使用。

RESTful 风格资源表述：可以根据需求对URL定位的资源返回不同的表述（也就是返回数据类型，比如XML、JSON等数据格式）。

Spring MVC 支持 RESTful 风格请求，具体讲的就是使用 @PathVariable 注解获取 RESTful 风格的请求URL中的路径变量。

GET、POST请求乱码解决

Post请求乱码，web.xml中加入过滤器

```
<!-- 解决post乱码问题 -->
<filter>
 <filter-name>encoding</filter-name>
 <filter-class>
 org.springframework.web.filter.CharacterEncodingFilter
 </filter-class>
 <!-- 设置编码参是UTF8 -->
 <init-param>
 <param-name>encoding</param-name>
 <param-value>UTF-8</param-value>
 </init-param>
 <init-param>
 <param-name>forceEncoding</param-name>
 <param-value>true</param-value>
 </init-param>
</filter>
<filter-mapping>
 <filter-name>encoding</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Get请求乱码（Get请求乱码需要修改tomcat下server.xml的配置）

```
<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080"
protocol="HTTP/1.1" redirectPort="8443"/>
```

@RequestBody回顾

@RequstBody 和@ReponseBody 区别：

交互：两个方向

1）前端到后台：前端ajax发送json格式字符串，后台直接接收为pojo参数，使用注解@RequstBody 2）后台到前端：后台直接返回pojo对象，前端直接接收为json对象或者字符串，使用注解 @ResponseBody

```
<mvc:resources location="/WEB-INF/js/" mapping="/js/**"/>//扫描到静态资源
```

```

<script type="text/javascript" src="/js/jquery.min.js"></script>//引入jQuery框架
<div>
        <h2>Ajax json交互</h2>
        <fieldset>
            <input type="button" id="ajaxBtn" value="ajax提交"/>
        </fieldset>
 </div>
<script>
    $(function () {

        $("#ajaxBtn").bind("click",function () {
            // 发送ajax请求
            $.ajax({
                url: '/demo/handle07',
                type: 'POST',
                data: '{"id":"1","name":"李四"}',
```

```
                contentType: 'application/json;charset=utf-8',//上送格式
                dataType: 'json',//返回数据格式
                success: function (data) {
                    alert(data.name);
                }
            })

        })
    })
</script>
```

```
    //@RequestBody将前端发送的JSON格式转为对象
    @RequestMapping("/handle07")
    public ModelAndView handle07(@RequestBody User user) {
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("date", date);
        modelAndView.setViewName("success");
        return modelAndView;
    }

```

@ResponseBody 回顾

```
@RequestMapping("/handle07")
    // 添加@ResponseBody之后，不再走视图解析器那个流程，而是等同于response直接输出数据
    public @ResponseBody User handle07(@RequestBody User user) {
        // 业务逻辑处理，修改name为张三丰
        user.setName("张三丰");
        return user;
    }
```

拦截器(Inteceptor)使用

1.1 监听器、过滤器和拦截器对比

Servlet：处理Request请求和Response响应

过滤器（Filter）：对Request请求起到过滤的作用，作用在Servlet之前，如果配置为/*可以对所有的资源访问（servlet、js/css静态资源等）进行过滤处理。

监听器（Listener）：实现了javax.servlet.ServletContextListener 接口的服务器端组件，它随Web应用的启动而启动，只初始化一次，然后会一直运行监视，随Web应用的停止而销毁

作用一： 做一些初始化工作，web应用中spring容器启动ContextLoaderListener

作用二：监听web中的特定事件，比如HttpSession,ServletRequest的创建和销毁；变量的创建、销毁和修改等。可以在某些动作前后增加处理，实现监控，比如**统计在线人数**，利用HttpSessionLisener等。

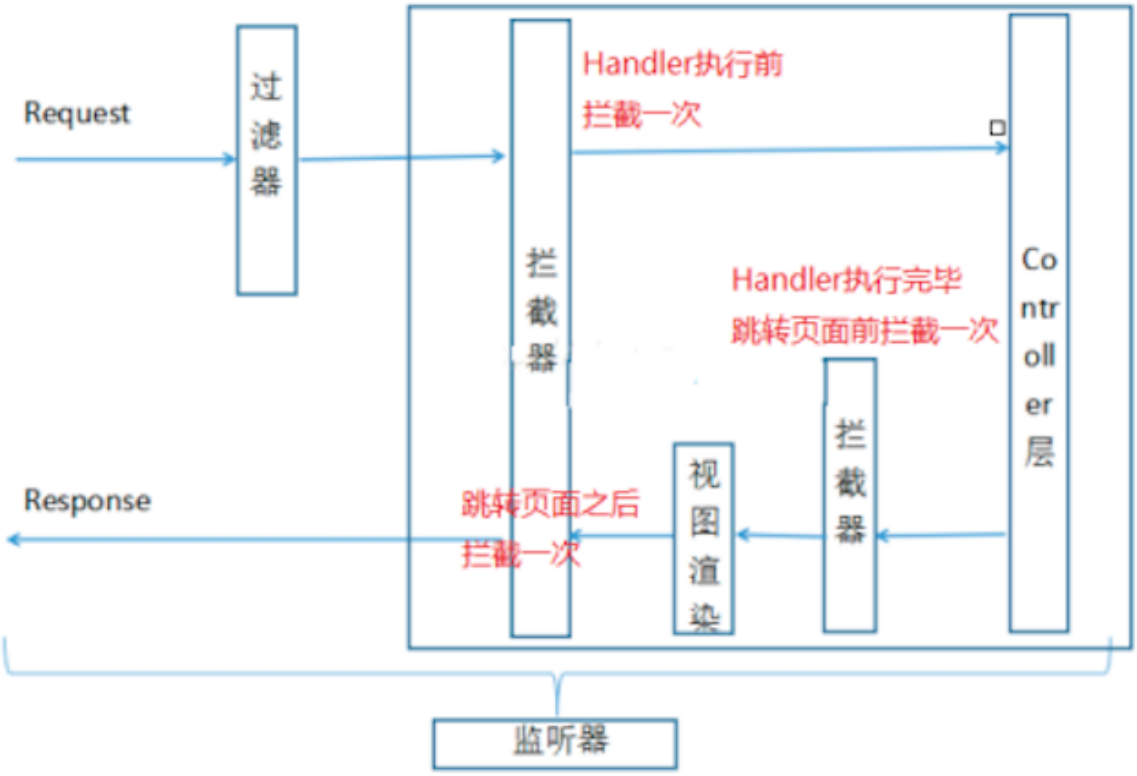
拦截器（Interceptor）：是SpringMVC、Struts等表现层框架自己的， 不会拦截jsp/html/css/image的访问等，只会拦截访问的控制器方法（Handler）。

从配置的角度也能够总结发现： serlvet、filter、listener是配置在web.xml中的，而interceptor是配置在表现层框架自己的配置文件中的

在Handler业务逻辑执行之前拦截一次

在Handler逻辑执行完毕但未跳转页面之前拦截一次

在跳转页面之后拦截一次



监听器示例：https://www.cnblogs.com/ygj0930/p/6374384.html

拦截器示例

```
/**
 * 自定义SpringMVC拦截器
 */
public class MyIntercepter01 implements HandlerInterceptor {


    /**
     * 会在handler方法业务逻辑执行之前执行
     * 往往在这里完成权限校验工作
     * @param request
     * @param response
     * @param handler
     * @return  返回值boolean代表是否放行，true代表放行，false代表中止
     * @throws Exception
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        System.out.println("MyIntercepter01 preHandle......");
        return true;
    }

```

```
22
23      /**
24       * 会在handler方法业务逻辑执行之后尚未跳转页面时执行
25       * @param request
26       * @param response
27       * @param handler
28       * @param modelAndView  封装了视图和数据，此时尚未跳转页面呢，你可以在这里针对返回
的数据和视图信息进行修改
29       * @throws Exception
30       */
31      @Override
32      public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
33          System.out.println("MyIntercepter01 postHandle......");
34      }
35
36      /**
37       * 页面已经跳转渲染完毕之后执行
38       * @param request
39       * @param response
40       * @param handler
41       * @param ex  可以在这里捕获异常
42       * @throws Exception
43       */
44      @Override
45      public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
46          System.out.println("MyIntercepter01 afterCompletion......");
47      }
48  }
49
```

```
1   <mvc:interceptors>
2           <!--拦截所有handler-->
3           <!--<bean class="com.lagou.edu.interceptor.MyIntercepter01"/>-->
4
5           <mvc:interceptor>
6               <!--配置当前拦截器的url拦截规则，**代表当前目录下及其子目录下的所有url-->
7               <mvc:mapping path="/**"/>
8               <!--exclude-mapping可以在mapping的基础上排除一些url拦截-->
9               <!--<mvc:exclude-mapping path="/demo/**"/>-->
10              <bean class="com.lagou.edu.interceptor.MyIntercepter01"/>
11          </mvc:interceptor>
12
13      </mvc:interceptors>
```

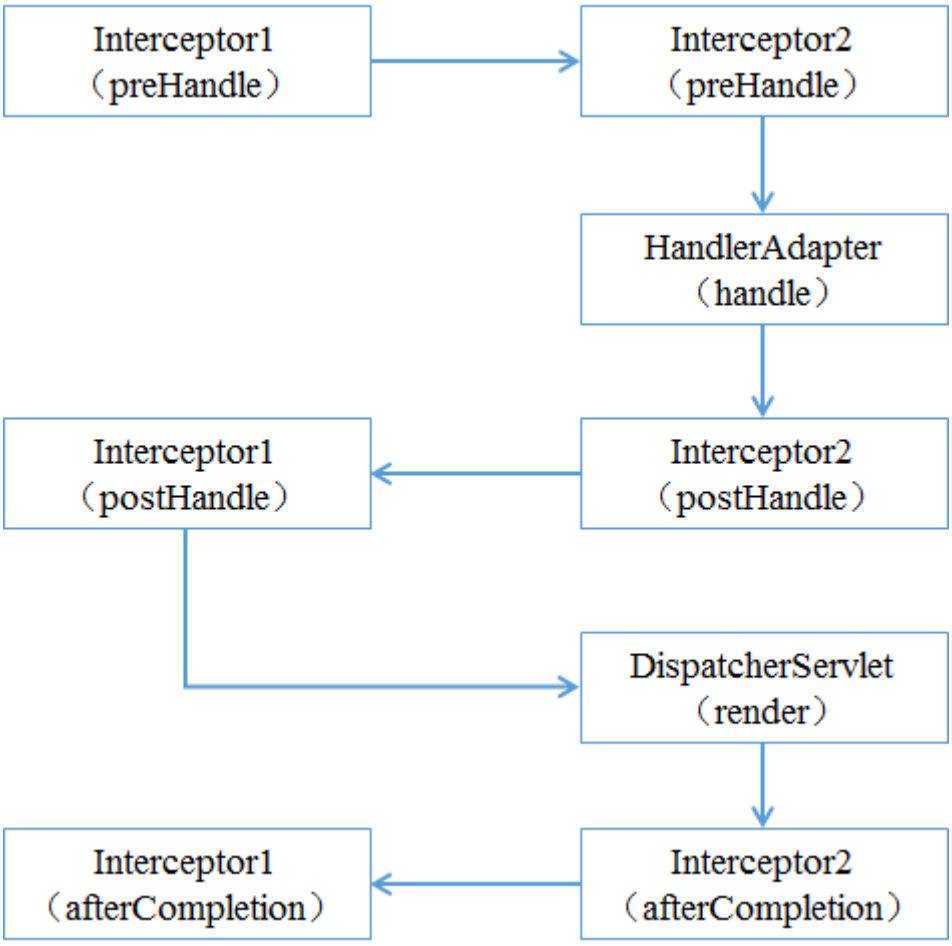
多个拦截器执行示例

```
1   <mvc:interceptors>
2           <!--拦截所有handler-->
3           <!--<bean class="com.lagou.edu.interceptor.MyIntercepter01"/>-->
4
5           <mvc:interceptor>
6               <!--配置当前拦截器的url拦截规则，**代表当前目录下及其子目录下的所有url-->
7               <mvc:mapping path="/**"/>
8               <!--exclude-mapping可以在mapping的基础上排除一些url拦截-->
```

```
9           <!--<mvc:exclude-mapping path="/demo/**"/>-->
10          <bean class="com.lagou.edu.interceptor.MyIntercepter01"/>
11      </mvc:interceptor>1
12
13      <mvc:interceptor>
14          <mvc:mapping path="/**"/>
15          <bean class="com.lagou.edu.interceptor.MyIntercepter02"/>
16      </mvc:interceptor>
17
18  </mvc:interceptors>
```

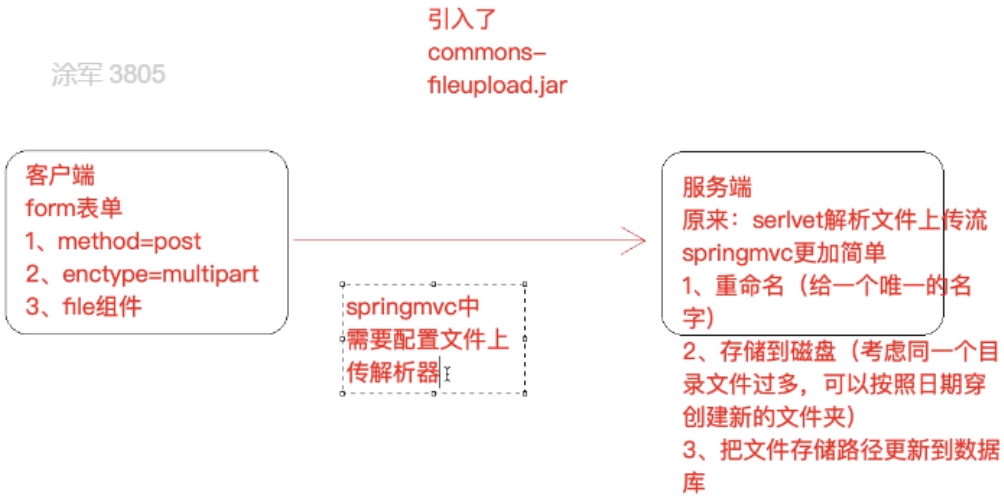
执行结果：

```
信息: Initializing Spring DispatcherServlet
[INFO] Completed initialization in 1731 ms
MyIntercepter01 preHandle......
MyIntercepter02 preHandle......
MyIntercepter02 postHandle......
MyIntercepter01 postHandle......
MyIntercepter02 afterCompletion......
MyIntercepter01 afterCompletion......
```



SpringMVC文件上传分析

```xml
<!--文件上传所需坐标-->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
```



SpringMVC文件上传代码：

```jsp
 <div>
        <h2>multipart 文件上传</h2>
        <fieldset>
            <%--
                1 method="post"
                2 enctype="multipart/form-data"
                3 type="file"
            --%>
            <form method="post" enctype="multipart/form-data"
action="/demo/upload">
                <input type="file" name="uploadFile"/>
                <input type="submit" value="上传"/>
            </form>
        </fieldset>
  </div>

```

```java

    /**
     * 文件上传
     * @return
     */
    @RequestMapping(value = "/upload")
    public ModelAndView upload(MultipartFile uploadFile, HttpSession session)
throws IOException {

        // 处理上传文件
        // 重命名，原名123.jpg ，获取后缀
```

```
        String originalFilename = uploadFile.getOriginalFilename();// 原始名称
        // 扩展名  jpg
        String ext = originalFilename.substring(originalFilename.lastIndexOf(".") + 1, originalFilename.length());
        String newName = UUID.randomUUID().toString() + "." + ext;

        // 存储,要存储到指定的文件夹，/uploads/yyyy-MM-dd，考虑文件过多的情况按照日期，生成一个子文件夹
        String realPath = session.getServletContext().getRealPath("/uploads");
        String datePath = new SimpleDateFormat("yyyy-MM-dd").format(new Date());
        File folder = new File(realPath + "/" + datePath);
        if (!folder.exists()) {
            folder.mkdirs();
        }

        // 存储文件到目录
        uploadFile.transferTo(new File(folder, newName));

        // TODO 文件磁盘路径要更新到数据库字段
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("date", date);
        modelAndView.setViewName("success");
        return modelAndView;
    }
```

```
<!--多元素解析器
       id固定为multipartResolver
   -->
   <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
       <!--设置上传文件大小上限，单位是字节，-1代表没有限制也是默认的-->
       <property name="maxUploadSize" value="5000000"/>
   </bean>

```

SpringMVC异常处理机制：

以下两种异常处理方式，mvc框架提供注解可以对相应的Exception做相应的处理。

当前controller类生效：

```
    // SpringMVC的异常处理机制（异常处理器）
    // 注意：写在这里只会对当前controller类生效
    @ExceptionHandler(ArithmeticException.class)
    public void handleException(ArithmeticException exception,HttpServletResponse response) {
        // 异常处理逻辑
        try {
            response.getWriter().write(exception.getMessage());//如果有ArithmeticException的异常会走到该方法
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

```

设置全局异常（所有controller类生效）：

```
// 可以让我们优雅的捕获所有Controller对象handler方法抛出的异常
@ControllerAdvice
public class GlobalExceptionResolver {
    @ExceptionHandler(ArithmeticException.class)
    public ModelAndView handleException(ArithmeticException exception,
HttpServletResponse response) {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("msg",exception.getMessage());
        modelAndView.setViewName("error");//转到报错页面
        return modelAndView;
    }
}
```

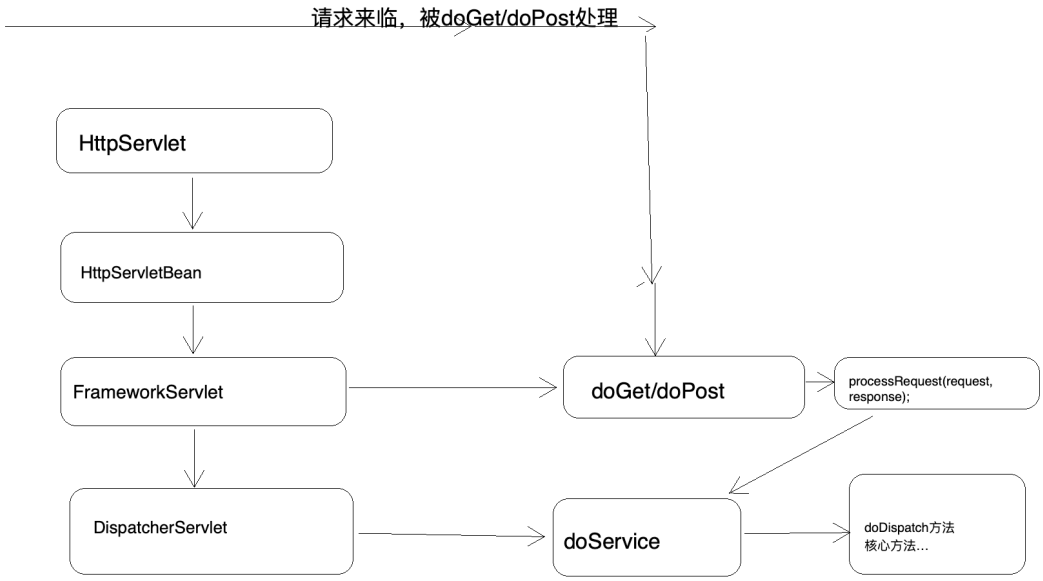
重定向和转发的区别：

```
   /**
     * SpringMVC 重定向时参数传递的问题
     * 转发：A 找 B 借钱400，B没有钱但是悄悄的找到C借了400块钱给A
     *    url不会变,参数也不会丢失,一个请求
     * 重定向：A 找 B 借钱400，B 说我没有钱，你找别人借去，那么A 又带着400块的借钱需求找
到C
     *    url会变,参数会丢失需要重新携带参数,两个请求
     */
转发：
    @RequestMapping("test_forward.do")
    public String testForward(Model model){
        model.addAttribute("msgBefore", "转发前的输出的信息");
        //转发到同一个控制器下的test.do
        return "forward:test.do";
    }

    @RequestMapping("test.do")
    public String test(Model model){
        model.addAttribute("msgAfter", "转发后的输出的信息");
        //跳转到/user/test.jsp页面
        return "/user/test";
    }

重定向：
    @RequestMapping("/handleRedirect")
    public String handleRedirect(String name, RedirectAttributes
redirectAttributes) {

        //return "redirect:handle01?name=" + name;  // 拼接参数安全性、参数长度都有局
限
        // addFlashAttribute方法设置了一个flash类型属性，该属性会被暂存到session中，在
跳转到页面之后该属性销毁
        redirectAttributes.addFlashAttribute("name", name);
        return "redirect:handle01";

    }
```

总结：

区别	转发forward()	重定向sendRedirect()
根目录	包含项目访问地址	没有项目访问地址
地址栏	不会发生变化	会发生变化
哪里跳转	服务器端进行的跳转	浏览器端进行的跳转
请求域中数据	不会丢失	会丢失

SpringMVC核心源码流程



doDispatch的核心步骤：
1）调用getHandler()获取到能够处理当前请求的执行链 HandlerExecutionChain（Handler+拦截器）但是如何去getHandler的？--得到Handler

2）调用getHandlerAdapter()；获取能够执行1）中Handler的适配器
但是如何去getHandlerAdapter的？--得到执行Handler的适配器

3）适配器调用Handler执行ha.handle（总会返回一个ModelAndView对象）--执行Handler

4）调用processDispatchResult()方法完成视图渲染跳转--返回前端

Spring MVC 必备设计模式

参见讲义（策略模式、模板方法模式、适配器模式）

Spring Data JPA 框架简介

Spring Data Jpa 是应用于Dao层的一个框架，简化数据库开发的，作用和Mybatis框架一样，但是在使用方式和底层机制是有所不同的。最明显的一个特点，Spring Data Jpa 开发Dao的时候，很多场景我们连sql语句都不需要开发。由Spring出品。

主要课程内容

- Spring Data JPA 介绍回顾
- Spring Data JPA、JPA规范和Hibernate之间的关系
- Spring Data JPA 应用（基于案例）
  - 使用步骤
  - 接口方法、使用方式
- Spring Data JPA 执行过程源码分析

Spring Data JPA 是 Spring 基于JPA 规范的基础上封装的一套 JPA 应用框架，可使开发者用极简的代码即可实现对数据库的访问和操作。它提供了包括增删改查等在内的常用功能！学习并使用 Spring Data JPA 可以极大提高开发效率。

说明：Spring Data JPA 极大简化了数据访问层代码。

如何简化呢？使用了Spring Data JPA，我们Dao层中只需要写接口，不需要写实现类，就自动具有了增删改查、分页查询等方法。

使用Spring Data JPA 很多场景下不需要我们自己写sql语句

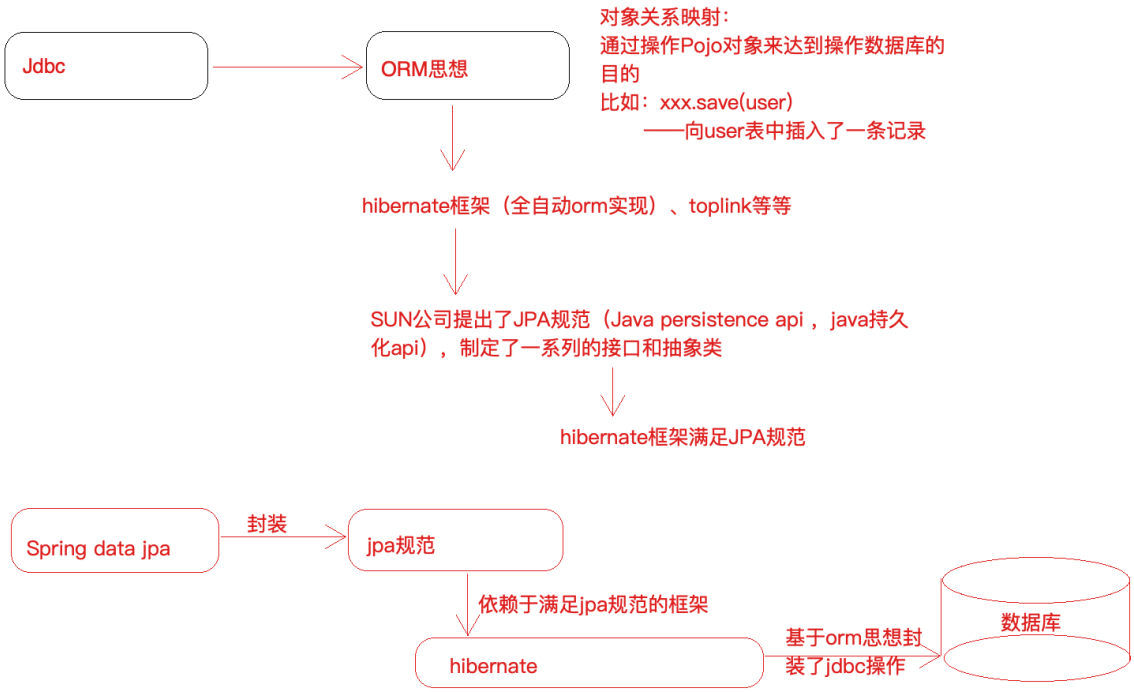
Spring Data 家族：

Currently, the release train contains the following modules:

- Spring Data Commons
- Spring Data JPA
- Spring Data KeyValue
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data Redis
- Spring Data REST
- Spring Data for Apache Cassandra
- Spring Data for Apache Geode
- Spring Data for Apache Solr
- Spring Data for Pivotal GemFire
- Spring Data Couchbase (community module)
- Spring Data Elasticsearch (community module)
- Spring Data Neo4j (community module)

Spring Data JPA，JPA规范和Hibernate之间的关系

Spring Data JPA 是 Spring 提供的一个封装了JPA 操作的框架，而 JPA 仅仅是规范，单独使用规范无法具体做什么，那么Spring Data JPA 、 JPA规范 以及 Hibernate （JPA 规范的一种实现）之间的关系是什么?



JPA 是一套规范，内部是由接口和抽象类组成的，Hiberanate 是一套成熟的 ORM 框架，而且 Hiberanate 实现了 JPA 规范，所以可以称 Hiberanate 为 JPA 的一种实现方式，我们使用 JPA 的 API 编程，意味着站在更高的角度去看待问题（面向接口编程）。

Spring Data JPA 是 Spring 提供的一套对 JPA 操作更加高级的封装，是在 JPA 规范下的专门用来进行数据持久化的解决方案。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
https://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/data/jpa
https://www.springframework.org/schema/data/jpa/spring-jpa.xsd
">
    <!--对Spring和SpringDataJPA进行配置-->
    <!--1、创建数据库连接池druid-->
    <!--引入外部资源文件-->
    <context:property-placeholder
            location="classpath:jdbc.properties"/>
    <!--第三方jar中的bean定义在xml中-->
    <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}"/>
    </bean>
    <!--2、配置一个JPA中非常重要的对象,entityManagerFactory
    entityManager类似于mybatis中的SqlSession：提供增删改查方法。
    entityManagerFactory类似于Mybatis中的SqlSessionFactory
    -->
    <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <!--配置一些细节.....-->
```

```
        <!--配置数据源-->
        <property name="dataSource" ref="dataSource"/>
        <!--配置包扫描（pojo实体类所在的包）-->
        <property name="packagesToScan"
                  value="com.lagou.edu.pojo"/>
        <!--指定jpa的具体实现，也就是hibernate-->
        <property name="persistenceProvider">
            <bean class="org.hibernate.jpa.HibernatePersistenceProvider"></bean>
        </property>
        <!--jpa方言配置,不同的jpa实现对于类似于beginTransaction等细节实现
        起来是不一样的，所以传入JpaDialect具体的实现类-->
        <property name="jpaDialect">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect">
</bean>
        </property>
        <!--配置具体provider，hibearnte框架的执行细节-->
        <property name="jpaVendorAdapter">
           <bean
 class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <!--定义hibernate框架的一些细节-->
                <!--
                配置数据表是否自动创建
                因为我们会建立pojo和数据表之间的映射关系
                程序启动时，如果数据表还没有创建，是否要程序给创建一下
                -->
                <property name="generateDdl" value="false"/>
                <!--
                指定数据库的类型
                hibernate本身是个dao层框架，可以支持多种数据库类型
                的，这里就指定本次使用的什么数据库
                -->
                <property name="database" value="MYSQL"/>
                <!--
                配置数据库的方言
                hiberante可以帮助我们拼装sql语句，但是不同的数据库sql
                语法是不同的，所以需要我们注入具体的数据库方言
                -->
                <property name="databasePlatform"
                          value="org.hibernate.dialect.MySQLDialect"/>
                <!--是否显示sql
                操作数据库时，是否打印sql
                -->
                <property name="showSql" value="true"/>
            </bean>
        </property>
    </bean>
    <!--3、引用上面创建的entityManagerFactory
    <jpa:repositories> 配置jpa的dao层细节
    base-package:指定dao层接口所在包
    -->
    <jpa:repositories base-package="com.lagou.edu.dao" entity-managerfactory-
                      ref="entityManagerFactory"
                      transaction-manager-ref="transactionManager"/>
    <!--4、事务管理器配置
    编写实体类 Resume，使用 JPA 注解配置映射关系
    jdbcTemplate/mybatis 使用的是DataSourceTransactionManager
    jpa规范：JpaTransactionManager
    -->
    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
```

```
    </bean>
    <!--5、声明式事务配置-->
    <!--
    <tx:annotation-driven/>
    -->
    <!--6、配置spring包扫描-->
    <context:component-scan base-package="com.lagou.edu"/>
</beans>
```

```
/**
 * 简历实体类（在类中要使用注解建立实体类和数据表之间的映射关系以及属性和字段的映射关系）
 * 1、实体类和数据表映射关系
 * @Entity
 * @Table
 * 2、实体类属性和表字段的映射关系
 * @Id 标识主键
 * @GeneratedValue 标识主键的生成策略
 * @Column 建立属性和字段映射
 */
@Entity
@Table(name = "tb_resume")
public class Resume {

    @Id
    /**
     * 生成策略经常使用的两种：
     * GenerationType.IDENTITY:依赖数据库中主键自增功能  Mysql
     * GenerationType.SEQUENCE:依靠序列来产生主键      Oracle
     */
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;
    @Column(name = "name")
    private String name;
    @Column(name = "address")
    private String address;
    @Column(name = "phone")
    private String phone;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
```

```
        this.address = address;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }


    @Override
    public String toString() {
        return "Resume{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", address='" + address + '\'' +
                ", phone='" + phone + '\'' +
                '}';
    }
}
```

```
/**
 * 一个符合SpringDataJpa要求的Dao层接口是需要继承JpaRepository和JpaSpecificationExecutor
 *
 * JpaRepository<操作的实体类类型,主键类型>
 *      封装了基本的CRUD操作
 *
 * JpaSpecificationExecutor<操作的实体类类型>
 *      封装了复杂的查询（分页、排序等）
 *
 */
public interface ResumeDao extends JpaRepository<Resume,Long>, JpaSpecificationExecutor<Resume> {


    @Query("from Resume  where id=?1 and name=?2")//使用该注解就是引用JPQL，可以不用上面继承的包
    public List<Resume> findByJpql(Long id,String name);


    /**
     * 使用原生sql语句查询，需要将nativeQuery属性设置为true，默认为false（jpql）
     * @param name
     * @param address
     * @return
     */
    @Query(value = "select * from tb_resume  where name like ?1 and address like ?2",nativeQuery = true)
    public List<Resume> findBySql(String name,String address);


    /**
     * 方法命名规则查询
     * 按照name模糊查询（like）
     *  方法名以findBy开头
     *          -属性名（首字母大写）
     *                  -查询方式（模糊查询、等价查询），如果不写查询方式，默认等价查询
```

```
    */
    public List<Resume> findByNameLikeAndAddress(String name,String address);

}

```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:applicationContext.xml"})
public class ResumeDaoTest {
    // 要测试IOC哪个对象注入即可
    @Autowired
    private ResumeDao resumeDao;

    /**
     * dao层接口调用，分成两块：
     * 1、基础的增删改查
     * 2、专门针对查询的详细分析使用
     */
    @Test
    public void testFindById(){
        // 早期的版本 dao.findOne(id);

        /*
            select resume0_.id as id1_0_0_,
                resume0_.address as address2_0_0_, resume0_.name as name3_0_0_,
                 resume0_.phone as phone4_0_0_ from tb_resume resume0_ where
resume0_.id=?
         */

        Optional<Resume> optional = resumeDao.findById(1l);
        Resume resume = optional.get();
        System.out.println(resume);
    }
    @Test
    public void testFindOne(){
        Resume resume = new Resume();
        resume.setId(1l);
        resume.setName("张三");
        Example<Resume> example = Example.of(resume);
        Optional<Resume> one = resumeDao.findOne(example);
        Resume resume1 = one.get();
        System.out.println(resume1);
    }
    @Test
    public void testSave(){
        // 新增和更新都使用save方法，通过传入的对象的主键有无来区分，没有主键信息那就是
新增，有主键信息就是更新
        Resume resume = new Resume();
        resume.setId(5l);
        resume.setName("赵六六");
        resume.setAddress("成都");
        resume.setPhone("132000000");
        Resume save = resumeDao.save(resume);
        System.out.println(save);
    }

    @Test
    public void testDelete(){
        resumeDao.deleteById(5l);
    }
```

```
53
54      @Test
55      public void testFindAll(){
56          List<Resume> list = resumeDao.findAll();
57          for (int i = 0; i < list.size(); i++) {
58              Resume resume =  list.get(i);
59              System.out.println(resume);
60          }
61      }
62
63      @Test
64      public void testSort(){
65          Sort sort = new Sort(Sort.Direction.DESC,"id");
66          List<Resume> list = resumeDao.findAll(sort);
67          for (int i = 0; i < list.size(); i++) {
68              Resume resume =  list.get(i);
69              System.out.println(resume);
70          }
71      }
72      @Test
73      public void testPage(){
74          /**
75           * 第一个参数：当前查询的页数，从0开始
76           * 第二个参数：每页查询的数量
77           */
78          Pageable pageable  = PageRequest.of(0,2);
79          //Pageable pageable = new PageRequest(0,2);
80          Page<Resume> all = resumeDao.findAll(pageable);
81          System.out.println(all);
82          /*for (int i = 0; i < list.size(); i++) {
83              Resume resume =  list.get(i);
84              System.out.println(resume);
85          }*/
86      }
87
88      /**
89       * ========================针对查询的使用进行分析=======================
90       * 方式一：调用继承的接口中的方法  findOne(),findById()
91       * 方式二：可以引入jpql（jpa查询语言）语句进行查询 (=====>>>> jpql 语句类似于
sql，只不过sql操作的是数据表和字段，jpql操作的是对象和属性，比如 from Resume where
id=xx)  hql
92       * 方式三：可以引入原生的sql语句
93       * 方式四：可以在接口中自定义方法，而且不必引入jpql或者sql语句，这种方式叫做方法命
名规则查询，也就是说定义的接口方法名是按照一定规则形成的，那么框架就能够理解我们的意图
94       * 方式五：动态查询
95       *       service层传入dao层的条件不确定，把service拿到条件封装成一个对象传递给Dao
层，这个对象就叫做Specification（对条件的一个封装）
96       *
97       *
98       *           // 根据条件查询单个对象
99       *           Optional<T> findOne(@Nullable Specification<T> var1);
100      *           // 根据条件查询所有
101      *           List<T> findAll(@Nullable Specification<T> var1);
102      *           // 根据条件查询并进行分页
103      *           Page<T> findAll(@Nullable Specification<T> var1, Pageable var2);
104      *           // 根据条件查询并进行排序
105      *           List<T> findAll(@Nullable Specification<T> var1, Sort var2);
106      *           // 根据条件统计
107      *           long count(@Nullable Specification<T> var1);
108      *
109      *       interface Specification<T>
```

```
     *              toPredicate(Root<T> var1, CriteriaQuery<?> var2,
CriteriaBuilder var3);用来封装查询条件的
     *                  Root:根属性（查询所需要的任何属性都可以从根对象中获取）
     *                  CriteriaQuery 自定义查询方式 用不上
     *                  CriteriaBuilder 查询构造器，封装了很多的查询条件（like =
等）
     *
     *
     */

    @Test
    public void testJpql(){
        List<Resume> list = resumeDao.findByJpql(1l, "张三");
        for (int i = 0; i < list.size(); i++) {
            Resume resume =  list.get(i);
            System.out.println(resume);
        }
    }

    @Test
    public void testSql(){
        List<Resume> list = resumeDao.findBySql("李%", "上海%");
        for (int i = 0; i < list.size(); i++) {
            Resume resume =  list.get(i);
            System.out.println(resume);
        }
    }

    @Test
    public void testMethodName(){
        List<Resume> list = resumeDao.findByNameLikeAndAddress("李%","上海");
        for (int i = 0; i < list.size(); i++) {
            Resume resume =  list.get(i);
            System.out.println(resume);
        }

    }

    // 动态查询，查询单个对象
    @Test
    public void testSpecfication(){

        /**
         * 动态条件封装
         * 匿名内部类
         *
         * toPredicate：动态组装查询条件
         *
         *      借助于两个参数完成条件拼装，，，  select * from tb_resume where
name='张三'
         *      Root：获取需要查询的对象属性
         *      CriteriaBuilder：构建查询条件，内部封装了很多查询条件（模糊查询，精准
查询）
         *
         *      需求：根据name（指定为"张三"）查询简历
          */

        Specification<Resume> specification = new Specification<Resume>() {
            @Override
            public Predicate toPredicate(Root<Resume> root, CriteriaQuery<?>
criteriaQuery, CriteriaBuilder criteriaBuilder) {
```

```
                // 获取到name属性
                Path<Object> name = root.get("name");

                // 使用CriteriaBuilder针对name属性构建条件（精准查询）
                Predicate predicate = criteriaBuilder.equal(name, "张三");
                return predicate;
            }
        };


        Optional<Resume> optional = resumeDao.findOne(specification);
        Resume resume = optional.get();
        System.out.println(resume);

    }

    @Test
    public void testSpecficationMultiCon(){

        /**

         *      需求：根据name（指定为"张三"）并且，address 以"北"开头（模糊匹配），查询简历
         */

        Specification<Resume> specification = new Specification<Resume>() {
            @Override
            public Predicate toPredicate(Root<Resume> root, CriteriaQuery<?> criteriaQuery, CriteriaBuilder criteriaBuilder) {
                // 获取到name属性
                Path<Object> name = root.get("name");
                Path<Object> address = root.get("address");
                // 条件1：使用CriteriaBuilder针对name属性构建条件（精准查询）
                Predicate predicate1 = criteriaBuilder.equal(name, "张三");
                // 条件2：address 以"北"开头（模糊匹配）
                Predicate predicate2 = criteriaBuilder.like(address.as(String.class), "北%");

                // 组合两个条件
                Predicate and = criteriaBuilder.and(predicate1, predicate2);

                return and;
            }
        };
        Optional<Resume> optional = resumeDao.findOne(specification);
        Resume resume = optional.get();
        System.out.println(resume);
    }
}

```

JPQL示例

在hebinate中又称HQL

```java
    //引用JPQL，可以不用上面继承的包
    @Query("from Resume  where id=?1 and name=?2")
    public List<Resume> findByJpql(Long id,String name);

    //这个不是JPQL
    //使用原生sql语句查询，需要将nativeQuery属性设置为true，默认为false-(jpql)
    @Query(value = "select * from tb_resume  where name like ?1 and address like ?2",nativeQuery = true)
    public List<Resume> findBySql(String name,String address);
```

SpringMVC 的控制器是不是单例模式,如果是,有什么问题,怎么解决?

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方 案是在控制器里面不能写字段。

怎么样把 ModelMap 里面的数据放入 Session 里面?

答：可以在类上面加上@SessionAttributes 注解,里面包含的字符串就是要放入 session 里面的 key

SpringMVC 怎么和 AJAX 相互调用的?

答： 通过 Jackson 框架就可以把 Java 里面的对象直接转化成 Js 可以识别的 Json 对象。

具体步骤如下：

1）加入 Jackson.jar

2）在配置文件中配置 json 的映射

3）在接受 Ajax 方法里面可以直接返回 Object,List 等,但方法前面要加上@ResponseBody

Get和Post的区别?

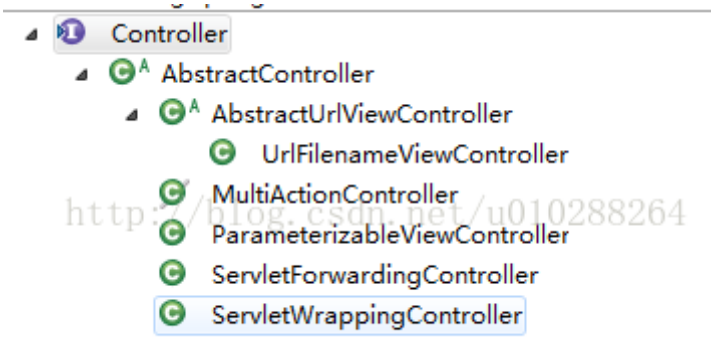
（大多数）浏览器通常都会限制url长度在2K个字节，而（大多数）服务器最多处理64K大小的url。超过的部分，恕不处理。

1. GET 在浏览器回退时是无害的，而 POST 会再次提交请求
2. GET 产生的 URL 地址可以被 Bookmark，而 POST 不可以
3. GET 请求会被浏览器主动 cache，而 POST 不会，除非手动设置
4. GET 请求只能进行 URL 编码，而 POST 支持多种编码方式（支持json、xml、form、浏览器原生form表单编码方式）
5. GET 请求参数会被完整保留在浏览器历史记录里，而 POST 中的参数不会被保留
6. GET 请求在 URL 中传送的长度是有限制的，而 POST 没有
7. 对参数的数据类型，GET 只接受 ASCII 字符，而 POST 没有限制
8. GET 比 POST 更不安全，因为参数直接暴露在 URL 上，所以不能用来传递敏感数据

9. GET 参数通过 URL 传递，POST 放在 Request body 中
10. GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包。【先发送 header，服务器响应 100（continue），浏览器再发送data，服务器响应200 ok（返回数据），理论上发两次包的时间和一次差别基本可以无视，网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点，并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次】

SpringMVC使用适配器模式:

Controller可以理解为Adaptee（被适配者）其中之一



可以看到处理器（宽泛的概念Controller，以及HttpRequestHandler，Servlet，等等）的类型不同，有多重实现方式，那么调用方式就不是确定的，如果需要直接调用Controller方法，需要调用的时候就得不断是使用if else来进行判断是哪一种子类然后执行。那么如果后面要扩展（宽泛的概念Controller，以及HttpRequestHandler，Servlet，等等）Controller，就得修改原来的代码，这样违背了开闭原则（对修改关闭，对扩展开放）。

Spring创建了一个适配器接口（HandlerAdapter）使得每一种处理器（宽泛的概念Controller，以及HttpRequestHandler，Servlet，等等）有一种对应的适配器实现类，让适配器代替（宽泛的概念Controller，以及HttpRequestHandler，Servlet，等等）执行相应的方法。这样在扩展Controller 时，只需要增加一个适配器类就完成了SpringMVC的扩展了，**每一种Controller有一种对应的适配器实现类。**

SpringMVC是怎么解决并发问题的?

SpringMVC的controller是singleton的（非线程安全的），这也许就是他和struts2的区别吧！和Struts一样，Spring的Controller默认是Singleton的，这意味着每个request过来，系统都会用原有的instance去处理，这样导致了两个结果:一是我们不用每次创建Controller，二是减少了对象创建和垃圾收集的时间;由于只有一个Controller的instance，当多个线程调用它的时候，它里面的**instance变量就不是线程安全的了，会发生窜数据的问题。**当然大多数情况下，我们根本不需要考虑线程安全的问题，比如dao,service等，除非在bean中声明了实例变量。因此，我们在使用Spring mvc 的contrller时，应避免在controller中定义实例变量。

如果控制器是使用单例形式，且controller中有一个私有的变量a,所有请求到同一个controller时，使用的a变量是共用的，即若是某个请求中修改了这个变量a，则，在别的请求中能够读到这个修改的内容。。

有几种解决方法:
1、在Controller中使用ThreadLocal变量
2、在Spring配置文件Controller中声明 scope="prototype"，每次都创建新的controller

所在在使用Spring开发web 时要注意，默认Controller、Dao、Service都是单例的。

SpringMVC中的拦截器和Servlet中的Filter有什么区别?

a. 首先最核心的一点他们的拦截侧重点是不同的，SpringMVC中的拦截器是依赖JDK的反射实现的，SpringMVC的拦截器主要是进行拦截请求，通过对Handler进行处理的时候进行拦截，先声明的拦截器中的preHandle方法会先执行，然而它的postHandle方法（他是介于处理完业务之后和返回结果之前）和afterCompletion方法却会后执行。并且Spring的拦截器是按照配置的先后顺序进行拦截的。

b. Servlet的filter是基于函数回调实现的过滤器，Filter主要是针对URL地址做一个编码的事情、而过滤掉没用的参数、

安全校验（比较泛的，比如登录不登录之类）

springmvc怎么前后端分离:

通过nginx

SpringMVC的常用注解?

1. @Component 在类定义之前添加@Component注解，他会被spring容器识别，并转为bean。
2. @Repository 对Dao实现类进行注解 (特殊的@Component)
3. @Service 用于对业务逻辑层进行注解， (特殊的@Component)
4. @Controller 用于控制层注解 ， (特殊的@Component)
5. @RequestMapping：用于处理请求地址映射，可以作用于类和方法上。
6. @RequestParam：用于获取传入参数的值
7. @PathViriable：用于定义路径参数值
8. @ResponseBody：作用于方法上，可以将整个返回结果以某种格式返回，如json或xml格式。
9. @CookieValue：用于获取请求的Cookie值