SpringMVC

主讲：崔译

一、MVC简介

- 是一种Web程序架构
- 是一种Web程序的设计思想
- MVC
    - Model 数据模型，是真正意义上要展示的数据
    - View 视图 用于实现 显示逻辑 数据的展示方式（HTML，JSP，FTL，XLS， ……）
    - Controller 控制器，用于控制请求流程，即：浏览器发送任意一个请求，请求同一个Servlet (即Controller)，由该Servlet 根据请求地址 决定后续要执行的方法以及响应
- 市面上存在很多实现了(大部分)MVC思想的框架（SpringMVC,SpringBoot,Struts1,Struts2..）

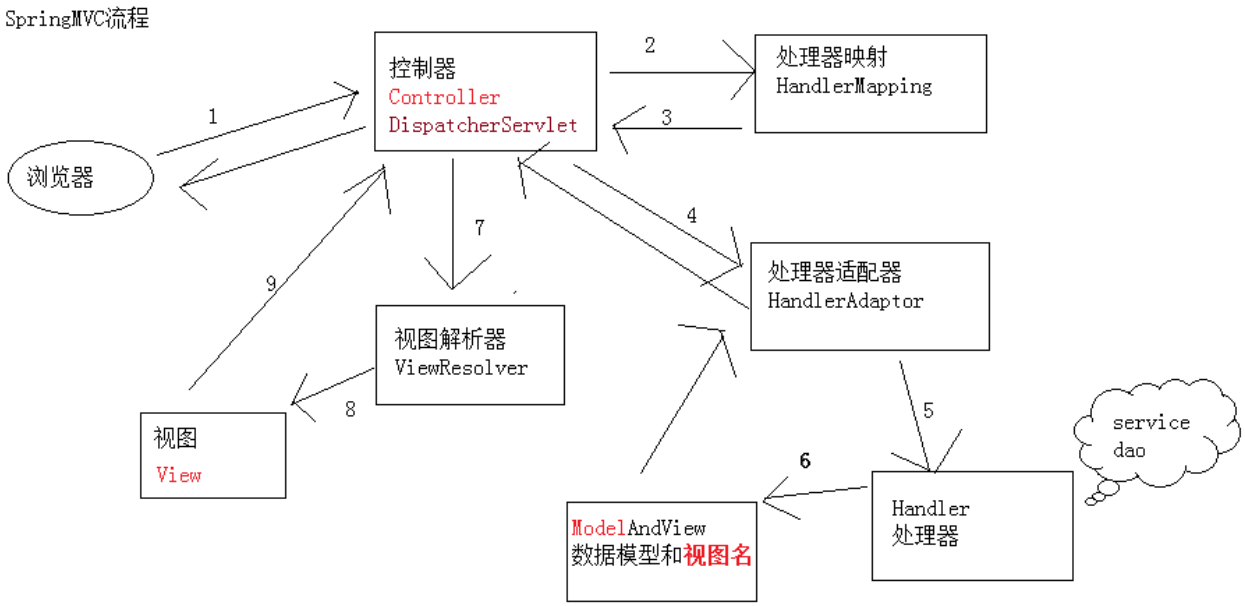
二、MVP简介

MVP 中的P就是对MVC 中的C 的扩展

Model View Presenter

> 简称：MVP 全称：Model-View-Presenter ；MVP 是从经典的模式MVC演变而来，它们的基本思想有相通的地方：Controller/Presenter负责逻辑的处理，Model提供数据，View负责显示。
>
> 在MVP中View并不直接使用Model，它们之间的通信是通过Presenter (MVC中的Controller)来进行的，所有的交互都发生在Presenter内部，而在MVC中View会直接从Model中读取数据而不是通过 Controller。

三、SpringMVC



1. 浏览器发送请求到控制器（DispatcherServlet）
2. 控制器根据请求地址到 处理器映射（HandlerMapping）中寻找对应的Handler
3. HandlerMapping 返回找到的Handler
4. 控制器根据返回的Handler 找对应的 处理器适配器（HandlerAdaptor）
5. 执行对应的Handler
6. Handler 将处理结果封装成 Model 对象 ， 然后将Model对象和 视图名 封装成ModelAndView对象返回
7. 控制器根据返回的ViewName ， 找到对应的视图解析器（ViewResolver）
8. 视图解析器 将 数据模型 渲染到 对应视图中
9. 视图解析器将渲染结果返回给Controller
10. 控制器将结果响应给客户端浏览器

四、HelloWorld

1、添加依赖

```
<!--SpringMVC核心包-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.3.9.RELEASE</version>
</dependency>

<!--J2EE 相关jar包-->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.0.1</version>
  <scope>provided</scope>
</dependency>

<dependency>

  <groupId>javax.servlet</groupId>
```

```
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>


 <!--Freemarker 依赖包-->
<dependency>
  <groupId>org.freemarker</groupId>
  <artifactId>freemarker</artifactId>
  <version>2.3.23</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>4.3.9.RELEASE</version>
</dependency>
<!--Freemarker 依赖包-->
```

2、配置DispatcherServlet

在web.xml中配置

注意：Maven的web项目模板的web.xml 文件的版本是web2.0

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

```
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--
      SpringMVC 的配置文件的默认位置是
        /WEB-INF/[SERVLET-NAME]-servlet.xml
      注意：
        1、Spring配置文件是Spring配置文件
        2、SpringMVC配置文件是SpringMVC配置文件
        3、真正Spring配置文件的加载方式是listener
        4、很多公司会使用SpringMVC来加载Spring配置文件（本质上两种配置文件都是Spring配置文件）
    -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:application/app*.xml</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>

    <servlet-name>springmvc</servlet-name>
```

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


  <!--Spring和 Web 整合-->
  <!--<listener>-->
    <!--<listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>-->
  <!--</listener>-->
  <!---->
  <!--<context-param>-->
    <!--<param-name>contextConfigLocation</param-name>-->
    <!--<param-value>classpath:application/applicationContext-mvc.xml</param-value>-->
  <!--</context-param>-->
```

3、HelloWorld-1

3-1 配置HandlerMapping

```
<!-- handlerMapping 开始-->
<bean
     class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
</bean>
<!-- handlerMapping 结束-->
```

3-2 配置HandlerAdaptor

```
<!-- HandlerAdaptor 开始-->
<bean
    class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter">
</bean>
<!-- HandlerAdaptor 结束-->
```

3-3 编写Handler

```
public class HelloWorld1 implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
                                      HttpServletResponse httpServletResponse)
            throws Exception {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");

        mv.addObject("abc","老王");

        return mv;
    }
}
```

3-4 配置Handler

```
<!-- Handler 开始-->
<bean name="/hello" class="day01.HelloWorld1"></bean>
 <!-- Handler 结束-->
```

3-5 配置视图解析器

```
<!-- ViewResolver 开始-->
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
                  value="org.springframework.web.servlet.view.JstlView">
        </property>
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
</bean>
<!-- ViewResolver 结束-->
```

4、HelloWorld-2

4-1 配置HandlerMapping 和 HandlerAdaptor

```
<!--配置注解驱动,用于替换  handlerMapping HandlerAdaptor-->
<mvc:annotation-driven />
```

4-2 编写，配置Handler

```
@Controller
public class HelloWorld2 {
    @RequestMapping("/hello")
    public ModelAndView hello()
    {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.addObject("mike","老谢");
        return mv;
    }
}
```

```
<context:component-scan base-package="day01"></context:component-scan>
```

4-3 配置视图解析器

```
 <!-- Freemarker 配置 开始-->
<!--Freemarker全局设置-->
```

```
<bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <!--模板文件的加载路径，相当于prefix-->
  <property name="templateLoaderPath" value="/WEB-INF/templates"></property>
  <!--字符集-->
  <property name="defaultEncoding" value="utf-8"></property>
  <!--Freemarker设置-->
  <property name="freemarkerSettings">
    <props>
      <!--模板更新延迟-->
      <prop key="template_update_delay">1</prop>
      <prop key="locale">zh_CN</prop>
      <prop key="date_format">yyyy-MM-dd</prop>
      <prop key="number_format">#.##</prop>
    </props>
  </property>
</bean>

<!--Freemarker视图解析器-->
<bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
  <!--开启Freemarker缓存-->
  <property name="cache" value="true"></property>
  <!--后缀-->
  <property name="suffix" value=".ftl"></property>
  <!--前缀，由于前面配过templateLoaderPath，此处可以置空-->
  <property name="prefix" value=""></property>
  <!--设置模板文件类型-->
  <property name="contentType" value="text/html;charset=utf8"></property>
  <!--允许freemarker 解析器从request中取值-->
  <property name="allowRequestOverride" value="true"></property>
  <!--允许freemarker 解析器从session中取值-->
  <property name="allowSessionOverride" value="true"></property>
  <property name="exposeSpringMacroHelpers" value="true"></property>
  <property name="exposeSessionAttributes" value="true"></property>
  <property name="exposeRequestAttributes" value="true"></property>
  <!--取绝对路径-->
  <property name="requestContextAttribute" value="request"></property>
</bean>
<!-- Freemarker 配置 结束-->
```

五、请求访问视图

```
<!--配置请求访问视图-->
<!--
        view-controller 视图控制器
        用于请求映射到视图中（不是映射到handler中）
        path ： 请求路径
        view-name: 视图名
-->
<mvc:view-controller path="/showLogin" view-name="login"></mvc:view-controller>
```

六、访问静态资源

```
<!--配置静态资源访问-->
<!--
        静态资源：
            mapping：请求路径
            location：资源位置
              * 通配符，通配一层路径
                    abc/*  abc/aaa(Y)  abc/aaa/bbb (N)
              ** 通配符，通配的是多层路径
-->
<!--<mvc:resources mapping="/stylesheet/**" location="/css/"></mvc:resources>-->
<!--<mvc:resources mapping="/js/**" location="/js/"></mvc:resources>-->
<!--
    <mvc:resources mapping="/images/**" location="/images/">
    </mvc:resources>
-->
<!--默认的Servlet处理器，将静态资源交给默认servlet-handler处理-->
<mvc:default-servlet-handler/>
```

七、Handler方法返回值类型

返回值类型	作用
ModelAndView	数据模型和视图名
字符串	视图名
字符串, forward:/字符串	转发，字符串是请求地址, `/`代表的是绝对路径
字符串, redirect:/字符串	重定向，字符串是请求地址, `/`代表的是绝对路径
void	将请求地址作为视图名

```
@Controller
@RequestMapping("/retType")
public class ReturnTypeController {

    @RequestMapping("/f1")
    public ModelAndView f1()
    {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("index");
        mv.addObject("mike","老谢");
        return mv;
    }
    @RequestMapping("/f2")
    public String f2()
    {

        return "index";
```

```
    }
    @RequestMapping("/f3")
    public String f3()
    {
        return "forward:/showLogin";
    }
    @RequestMapping("/f4")
    public String f4()
    {
        return "redirect:/showLogin";
    }
    @RequestMapping("/f5")
    public void f5()
    {

    }
}
```

八、Handler方法请求格式

1、@RequestMapping注解

用于定义请求地址，请求限制

- 加在类上，表示该类中的所有的handler方法的请求前缀
- 加在方法上，表示该方法的请求地址

```
@Controller
@RequestMapping("/rmc")
public class RequestMappingController {

    // 发送f1请求，并且请求方式是post请求
    @RequestMapping(value = "/f1",method = RequestMethod.POST)
    public String f1()
    {
        return "index";
    }

    // 多个url 映射到同一个handler方法
    @RequestMapping(value = {"/f2","/f3"})
    public String f2()
    {
        return "index";
    }

    //必须包含参数id
    @RequestMapping(value = "/f4",params = "id")
    public String f4()
    {
        return "index";
    }
```

```
    //必须包含参数id，并且值是4
    @RequestMapping(value = "/f5",params = "id=4")
    public String f5()
    {
        return "index";
    }

    //请求头中必须包含User-Agent,并且值是.....
    @RequestMapping(value = "/f6",headers="User-Agent=Mozilla/5.0 (Windows NT 6.1;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.99 Safari/537.36")
    public String f6()
    {
        return "index";
    }
}
```

2、使用Ant风格的URL

在请求地址中使用通配符

通配符	作用
*	匹配任意字符，但是不包含路径分隔符，能且只能匹配一层路径
**	匹配任意多层（包括0层）
?	匹配任意一个字符

```
@Controller
@RequestMapping("/ant")
public class AntController {

    @RequestMapping("/f1/*")
    public String f1()
    {
        return "index";
    }

    @RequestMapping("/f2/**")
    public String f2()
    {
        return "index";
    }

    @RequestMapping("/f3?")
    public String f3()
    {
        return "index";
    }

    @RequestMapping("/f4*/*")
```

```
    public String f4()
    {
        return "index";
    }
}
```

3、使用REST风格URL

将请求参数作为URL 的一部分

> deleteById?id=3 ---> REST --> deleteById/3

```
@Controller
@RequestMapping("/rest")
public class RestController {
    @RequestMapping("/delete/{id}")
    public String f1( @PathVariable("id") int id )
    {
        System.out.println(id);
        return "index";
    }
    @RequestMapping("/findById/{id:\\d+}")
    public String f2( @PathVariable("id") int id )
    {
        System.out.println(id);
        return "index";
    }
}
```

九、Handler方法参数

1、所有的Servlet API

> HttpServletRequest HttpServletResponse HttpSession
>
> ImputStream / OutputStream -> request.getInputStream() response.getOutputStream
>
> Reader / Writer -> request.getReader / response.getWriter()

```
@RequestMapping("/f1")
public String f1(HttpServletRequest request)
{
  System.out.println(request.getParameter("cmd"));

  return "index";
}
```

SpringMVC官方不建议使用ServletAPI

2、数据模型参数

在Handler方法中，可以提供 `Map` / `Model` / `ModelMap` 类型的参数，不论哪一个 ， 都是 `Map` 的结构，**用于替代 `request.setAttribute()` 方法** ， 向 `request` 作用域 中 放值

```
@Controller
@RequestMapping("/param")
public class ParamController {

    @RequestMapping("/f1")
    public String f1(HttpServletRequest request)
    {
        System.out.println(request.getParameter("cmd"));
        return "index";
    }

    @RequestMapping("/f2")
    public String f2(Map map, Model model, ModelMap modelMap)
    {
        map.put("username","xyl");
        model.addAttribute("author","cy");
        List list = new ArrayList();
        list.add("xyl你的鼻子有两个孔");
        list.add("感冒时还流着鼻涕妞纽");
        list.add("xyl你有着黑漆漆的眼");
        list.add("望呀望呀望，也望不到边");
        list.add("xyl你的耳朵大又大");
        list.add("忽闪忽闪也听不见我在骂你傻");
        list.add("xyl你的尾巴卷又卷");
        list.add("原来蹦蹦跳跳，也离不开它");
        modelMap.addAttribute("data",list);
        return "forward:/showParam";
    }
}
```

3、对象类型参数

3-1 请求参数名和方法参数名（属性）一致

```
@RequestMapping("/f3")
public String f3(String username,String address)
{
  System.out.println(username+";" + address);
  return "index";
}

@RequestMapping("/f4")
public String f4(User user)
{
  System.out.println(user);
  return "index";
}
```

3-2 请求参数名和方法参数名（属性）不一致

```
@RequestMapping("/f5")
public String f5(
  @RequestParam("username") String name,
  @RequestParam("address")   String addr
)
{
  System.out.println(name+";" + addr);
  return "index";
}
```

3-3 关于RequestParam注解

- 如果Handler方法添加了RequestParam注解，那么请求中**默认**必须存在该参数，否则报错
- RequestParam注解中存在三个属性:
  - value / name : 指定请求参数名
  - required：true | false 设置该参数是否是必须传递的参数
  - defaultValue : 指定默认值，defaultValue 的值的类型是字符串，但是不影响参数类型

```
@RequestMapping("/f6")
public String f6(
  @RequestParam(value = "pn",required = false,defaultValue = "1")
    Integer pageNo
)
{
  System.out.println(pageNo);
  return "index";
}
```

十、@ModelAttribute注解

对于Handler方法中的User 参数，作用：

1. 接收请求参数
2. 作为request作用域中的属性(attribute) --> key?
3. 作为下一次请求中的请求参数(parameter)

```
public String f1(User user){}
```

1、添加在Handler方法参数上

```
@RequestMapping("/f1")
// 接收请求参数
// 将 模型属性 中的user 注入到参数usera上
// 将 usera 放到 模型属性 中， 属性的key 是 user
public String f1( @ModelAttribute("user") User usera)
{
   return "forward:/model/f4";
}
```

2、添加在一个返回值类型不为void的方法上

```
// 在 ModelAttribute中添加一个key 为someUser ， value 是user对象的属性
// 当访问f4所在controlle中的其他Handler方法前，都会先执行该方法
@ModelAttribute("someUser")
public User f4()
{
  User user = new User();
  return user;
}
```

十一、@SessionAttributes

- 将ModelAttribute中的属性 提升到session 的读写级别
- 在request中和session 中都存在key相同的属性，并且是同一个对象

```
@Controller
@RequestMapping("/session")
// 将ModelAttribute中key为sessionUser的对象拷贝到session中（model和session中是同一个对象）
// 前提是Model中一定要存在该对象
@SessionAttributes("user")
// 将ModelAttribute中所有类型为User的对象拷贝到session中
//@SessionAttributes(User.class)
public class SessionAttributesController {

    @RequestMapping("/f1")
    public String f1(Model model)
    {
        User user = new User();
        user.setAddress("aaaaa");
        user.setUsername("abc");

        model.addAttribute("user",user);
        return "redirect:/showSession";
    }

    @RequestMapping("/f2")
    public String f2( User user)
    {

        System.out.println(user);
```

```
        return "session";
    }

}
```

SessionStatus

```
@RequestMapping("/logout")
public String logout(SessionStatus status)
{
  status.setComplete();// session.invalidate()
  return "session";
}
```

十二、@ExceptionHandler

- SpringMVC 建议Handler方法将异常抛出
- 使用ExceptionHandler注解处理异常

```
@Controller
@RequestMapping("/exc")
public class ExceptionHandlerController {

    // 处理当前类中的handler方法抛出的IOException
    @ExceptionHandler(IOException.class)
    public String ioHandler(Exception e)
    {
        System.out.println(e.getMessage());
        return "session";
    }
}
```

```
// 定义全局 控制器的通知（异常处理器）
@ControllerAdvice
public class GlobalExceptionHandler {

    // 处理所有Controller类中的handler方法抛出的Throwable
    @ExceptionHandler(Throwable.class)
    public String Ce(Exception e)
    {
        System.out.println(e.getMessage());
        return "index";
    }

}
```

十三、处理Ajax请求

1、添加对象转JSON依赖

```
<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.49</version>
</dependency>
```

2、配置MessageConverter

```
<!--配置消息转换器===》方式1
        必须在annotation-driven的上面
-->
<bean class="
            org.springframework.web.servlet.mvc
            .method.annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean class="
                    com.alibaba.fastjson
                    .support.spring.FastJsonHttpMessageConverter">
                <property name="supportedMediaTypes">
                      <list>
                          <value>application/json;charset=UTF-8</value>
                      </list>
                 </property>
             </bean>
         </list>
     </property>
</bean>
```

```
<!--配置消息转换器===》方式2 -->
<mvc:annotation-driven>
  <mvc:message-converters>
    <bean class="
       com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter
    ">
      <property name="supportedMediaTypes">
        <list>
          <value>application/json;charset=UTF-8</value>
        </list>
      </property>
    </bean>
  </mvc:message-converters>
</mvc:annotation-driven>
```

3、编写代码，返回JSON

```
@Controller
@RequestMapping("/ajax")
public class AjaxController {
    @RequestMapping("/f1")
    @ResponseBody
    public Map f1()
    {
        Map map = new HashMap();
        map.put("aa","bb");
        map.put("cc","dd");
        return map;
    }
    @RequestMapping("/f2")
    @ResponseBody
    public User f2()
    {
       User user = new User("anc","addr",new Date());
       return user;
    }
}
```

```
public class User {
    private String username;
    private String address;
    @JSONField(format = "yyyy-MM-dd")
    private Date birthday;
}
```

使用@RestController 替代 @Controller

```
// 当前类中的所有的方法都自动添加@ResponseBody 注解
@RestController
@RequestMapping("/ajax")
public class AjaxController {
    @RequestMapping("/f1")
    public Map f1()
    {
        Map map = new HashMap();
        map.put("aa","bb");
        map.put("cc","dd");
        return map;
    }
    @RequestMapping("/f2")
    public User f2()
    {
       User user = new User("anc","addr",new Date());
       return user;
    }
}
```

十四、文件上传

1、添加依赖

SpringMVC底层使用commons-fileupload实现文件上传

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

2、配置文件上传的bean

```
 <!-- 文件上传 开始-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <!--上传临时目录-->
  <!--<property name="uploadTempDir" value="WEB-INF"></property>-->
  <!--文件最大大小-->
  <!--<property name="maxUploadSize" value="10240000"></property>-->
  <!--单文件最大大小-->
  <!--<property name="maxUploadSizePerFile" value="5000000"></property>-->
</bean>
<!-- 文件上传 结束-->
```

3、编写代码

```
@RequestMapping("/upload")
public String upload(MultipartFile myFile, HttpServletRequest request) throws
IOException {
  System.out.println(myFile.getOriginalFilename());
  String path =  request.getServletContext().getRealPath("upload");
  path += File.separator +
            UUID.randomUUID().toString() +
            myFile.getOriginalFilename()
                  .substring(myFile.getOriginalFilename().lastIndexOf("."));
  File dest = new File(path);

  myFile.transferTo(dest);

  return "session";
}
```

4、ajax文件上传

```
$(function(){
```

```
  $(":button").bind("click",function(){
    // 表单的dom对象
    let form = $("form")[0];
    // 创建表单域对象
    let formData = new FormData(form);
    $.ajax({
      url:"${ctx}/upload/ajaxupload",
      data:formData,
      contentType : false,// 告诉jQuery不要去设置Content-Type请求头
      processData: false,// 告诉jQuery不要去处理发送的数据
      dataType:'json',
      type:"post",
      success:function(data){
        console.log(data);
      }
    })
  });
});
```

十五、文件下载

```
@RequestMapping("/load")
public ResponseEntity<byte[]> f() throws IOException {
  File file =
    new ClassPathResource("application/applicationContext-mvc.xml")
    .getFile();

  byte[] body = FileUtils.readFileToByteArray(file);

  HttpHeaders header = new HttpHeaders();
  header.setContentType(MediaType.APPLICATION_OCTET_STREAM);
  header.setContentDispositionFormData("attachment","app.xml");

  //code 响应状态码
  HttpStatus code = HttpStatus.CREATED;
  ResponseEntity<byte[]> entity = new ResponseEntity<>(body,header,code);
  return entity;
}
```

十六、Interceptor

SpringMVC的 拦截器，另类的 AOP ， 可以认为是Controller 的 环绕通知

1、编写代码

```
public class SomeInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest httpServletRequest,
```

```
                                HttpServletResponse httpServletResponse, Object o) throws
Exception {
        // 在进入handler方法之前执行
        System.out.println("preHandle.....");

        // false 代表不继续执行handler，
        // true 代表继续执行handler，chain.doFilter() / pjp.proceed()
        return true;
    }

    private ModelAndView mv;

    @Override
    public void postHandle(HttpServletRequest httpServletRequest,
                           HttpServletResponse httpServletResponse,
                           Object o, ModelAndView modelAndView) throws Exception {
        // 在Handler你方法执行结束后，执行  after-returning
        modelAndView.addObject("abc","456");
        mv = modelAndView;
        System.out.println("postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest httpServletRequest,
                                HttpServletResponse httpServletResponse,
                                Object o, Exception e) throws Exception {

        // handler方法执行结束，准备响应页面时执行 after-throwing
        mv.addObject("abc","789");
        System.out.println("afterCompletion");

    }
}
```

2、配置interceptor

```
<!-- interceptor 开始-->
<mvc:interceptors>
  <mvc:interceptor>
    <!--要拦截的url-->
    <mvc:mapping path="/inter/f1/*"/>
    <!--不要拦截的url-->
    <mvc:exclude-mapping path="/inter/f1/c"/>
    <bean class="day02.SomeInterceptor"></bean>
  </mvc:interceptor>

</mvc:interceptors>
<!-- interceptor 结束-->
```

十七、乱码问题解决方案

- **参照web笔记，乱码问题解决方案**
- 配置tomcat字符集

```
 <plugin>
   <groupId>org.apache.tomcat.maven</groupId>
   <artifactId>tomcat7-maven-plugin</artifactId>
   <version>2.2</version>
   <configuration>
     <!--Application Context / pageContext.request.contextPath-->
     <path>/</path>
     <!--指定tomcat端口号-->
     <port>8080</port>
     <!--设置get请求字符集-->
     <uriEncoding>utf-8</uriEncoding>
   </configuration>
</plugin>
```

- 配置字符集

```
<!-- 字符集过滤器 开始-->
  <filter>
    <filter-name>charsetFilter</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>
    <init-param>
      <param-name>forceRequestEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>forceResponseEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>charsetFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <!-- 字符集过滤器 结束-->
```

十八、SpringMVC注解版

作为 帮助手册 使用

```
@Configuration
@EnableWebMvc
//<context:component-scan base-package="day01"></context:component-scan>
@ComponentScan(basePackages = {"day01","day02"})
```

```
public class SpringMVCConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //<mvc:view-controller path="/showLogin" view-name="login"></mvc:view-
controller>
        registry.addViewController("/showRegist").setViewName("regist");
        registry.addViewController("/showIndex").setViewName("index");
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        //freeMarkerViewResolver
        FreeMarkerViewResolver resolver = new FreeMarkerViewResolver();
        resolver.setAllowRequestOverride(true);
        resolver.setAllowSessionOverride(true);
        resolver.setExposeSpringMacroHelpers(true);
        resolver.setExposeSessionAttributes(true);
        resolver.setExposeRequestAttributes(true);
        resolver.setRequestContextAttribute("request");
        resolver.setCache(true);
        resolver.setContentType("text/html;charset=utf8");
        resolver.setSuffix(".ftl");
        resolver.setPrefix("");
    }

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer()
    {
        FreeMarkerConfigurer config = new FreeMarkerConfigurer();
        config.setTemplateLoaderPath("/WEB-INF/templates");
        config.setDefaultEncoding("utf-8");

        Properties p = new Properties();
        p.setProperty("template_update_delay","1");
        p.setProperty("locale","zh_CN");
        p.setProperty("date_format","yyyy-MM-dd");
        p.setProperty("number_format","#.##");

        config.setFreemarkerSettings(p);

        return  config;
    }

    @Bean
    public SomeInterceptor someInterceptor()
    {
        return new SomeInterceptor();
    }


    @Override

    public void addInterceptors(InterceptorRegistry registry) {
```

```
        //<mvc:interceptors>
        registry.addInterceptor(someInterceptor())
                .addPathPatterns("/inter/f1/*")
                .excludePathPatterns("/inter/f1/c");

    }

    @Bean
    public CommonsMultipartResolver multipartResolver()
    {
        //<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        CommonsMultipartResolver resolver = new CommonsMultipartResolver();
        return resolver;
    }


    @Override
    public void extendMessageConverters(List<HttpMessageConverter<?>> converters) {
        //<mvc:message-converters>
        FastJsonHttpMessageConverter converter = new FastJsonHttpMessageConverter();
        List<MediaType> list = new ArrayList();
        list.add(MediaType.APPLICATION_JSON_UTF8);
        converter.setSupportedMediaTypes(list);
        converters.add(converter);
    }

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        // <mvc:default-servlet-handler/>
        configurer.enable();
    }
}
```