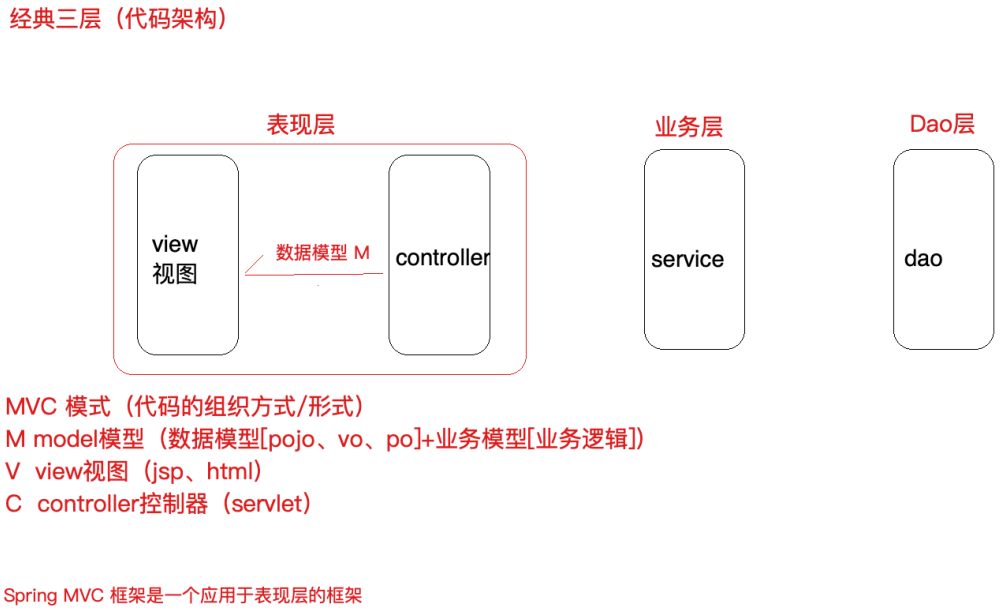
课程主要内容：

- Spring MVC 应用(常规使用)
- Spring MVC 高级技术(拦截器、异常处理器等)
- 手写 MVC 框架(自定义 MVC 框架，难点/重点)
- Spring MV 源码深度剖析(难点/重点)
- SSM 整合

Part I. Spring MVC基础知识

1. Spring MVC 简介

1.1 经典三层架构



经典三层是一种宏观组织架构。

1. 表现层：即Web层。
    1. 包括展示层（结果展示）和控制层（接收请求等）。
    2. 表现层的设计一般都使用 MVC 模型(MVC 是表现层的设计模型，是代码的组织方式，和其他层没有关系)：
        1. Model：数据模型 + 业务模型
        2. View：jsp，html

    3. Controller：处理用户交互的部分。作用一般就是处理程序逻辑。
2. 业务层：即Service层，负责业务逻辑处理。
3. 持久层：即Dao层，负责数据持久化，包括数据层（即数据库）和数据访问层。通俗的讲，持久层就是和数据库交互(增删改查)。

1.2 What is Spring MVC?

- 全名 Spring Web MVC，应用于经典三层中的表现层，是一种基于 Java 的 实现 MVC 设计模型的 请求驱动类型的轻量级Web开发框架，属于 SpringFrameWork 的后续产品。（Struts2也是为了解决表现层问题 的web框架，但次于Spring3.0中的Spring MVC。）
- 主要职责：处理前端HTTP请求。
- **本质**：不同于原生servlet模式，Web层不再需要逐个跟各种功能的servlet打交道，Spring MVC把这些各种各样的servlet（e.g. UserServlet, OrderServlet, ProductServlet, etc.）封装成为了一个全局的DispatchServlet（前端控制器），从此，Web层只与Front Controller打交道即可。
- 作用:
    - 接收请求
    - 返回响应
    - 跳转页面

2. Spring Web MVC 工作流程

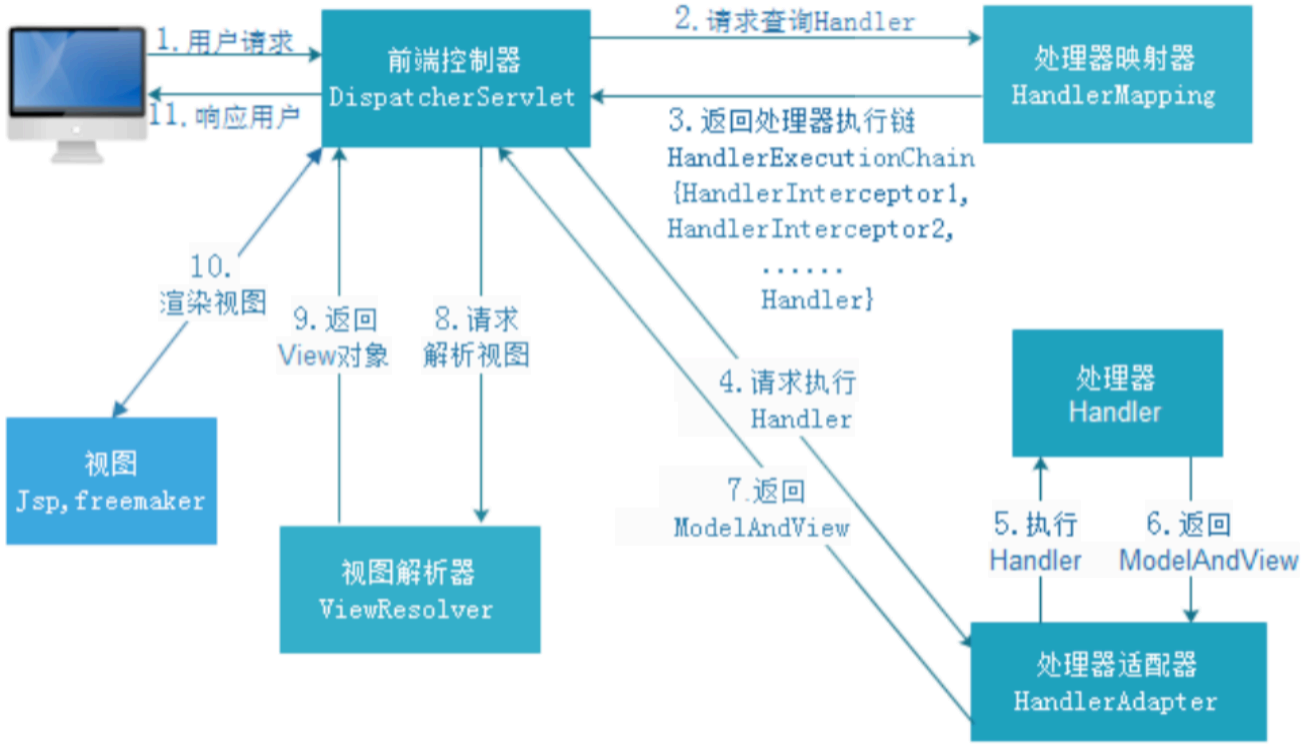
需求：前端浏览器请求URL（http://localhost:8080/demo/handle01），前端页面显示后台服务器的时间。

(`springmvc-demo`工程创建见笔记最后Extra 1.1)

开发过程：

- 配置DispatcherServlet前端控制器
- 开发处理具体业务逻辑的Handler (@Controller、@RequestMapping)
- xml配置文件配置controller扫描，配置springmvc 核心三大件（即ViewResolver, HandlerMapping, HandlerAdapter）
- 将xml文件路径告诉springmvc(DispatcherServlet)

2.1 Spring MVC 处理请求的流程



流程说明

第一步: 用户发送请求至前端控制器DispatcherServlet

第二步: DispatcherServlet收到请求调用HandlerMapping处理器映射器

第三步: HandlerMapping根据请求Url找到具体的Handler(后端控制器)，生成 Handler 对象及 HandlerInterceptor 对象 (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ViewResolver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

2.2 九大组件

1. HandlerMapping

HandlerMapping 是用来查找 Handler 的。具体的表现形式可以是类，也可以是方法：

1. 方法 -- 标注了@RequestMapping的每个方法都可以看成是一个Handler。一个类中可以定义多个Handler，Handler负责具体实际的请求处理。
2. 类 -- 下面这个OneController class就只是一个Handler。该写法并不方便，故不推荐。

```
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class OneController implements Controller {
    @Override
    public ModelAndView
handleRequest(javax.servlet.http.HttpServletRequest httpServletRequest,
javax.servlet.http.HttpServletResponse httpServletResponse) throws
Exception {
      // 业务逻辑

        return null;
    }
}
```

2. HandlerAdapter

   职责：让固定的 Servlet 处理方法调用各种各样的 Handler 来进行请求处理。

   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，Servlet 的方法结构是固定的 `doService(HttpServletRequest req,HttpServletResponse resp)` 形式。

3. HandlerExceptionResolver

   职责：用于处理 Handler 产生的异常情况，可以根据不同异常设置ModelAndView。

   这样之后的渲染方法才能够将 ModelAndView 渲染成页面。

4. ViewResolver

   ViewResolver相当于翻译了Controller的返回结果然后封装返回给View。

   职责1：找到渲染所用的模板

   职责2：找到渲染所用的技术(例如视图的类型，如JSP)并填入参数

   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。

   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。

5. RequestToViewNameTranslator

   职责：把请求转换成视图名称。

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。

6. LocaleResolver

   职责：从请求中解析出 Locale。

7. ThemeResolver

   职责：解析主题。主题是样式、图片及它们所形成的显示效果的集合。

8. MultipartResolver

   职责：用于上传请求，通过将普通的请求包装成 MultipartHttpServletRequest 来实现。

9. FlashMapManager

   职责：重定向时的参数传递

3. 请求参数绑定 - SpringMVC如何接收请求参数

3.1 原生servlet vs. SpringMVC 框架

原生servlet接收一个整型参数:

```
String ageStr = request.getParameter("age");
Integer age = Integer.parseInt(ageStr);
```

SpringMVC框架接收一个整型参数，直接在Handler方法对input中 声明形参即可：

```
@RequestMapping("/myurl")
public String handle(Integer age) {
  System.out.println(age);
}
```

所以SpringMVC的参数绑定即：**取出参数值 绑定到Handler方法的形参上。**

3.2 参数绑定

访问 http://localhost:8080/ ，就能看到index.jsp 渲染出的页面。

3.2.1 默认支持原生Servlet API 作为Handler方法参数

```
@Controller
@RequestMapping("/demo")
public class DemoController2 {

    /**
     *
     * SpringMVC 对原生servlet api的支持  url：/demo/handle02?id=1
     *
     * 如果要在SpringMVC中使用servlet原生对象，比如
HttpServletRequest\HttpServletResponse\HttpSession，
     * 直接在Handler方法形参中声明使用即可
     *
     */
    @RequestMapping("/handle02")
    public ModelAndView handle02(HttpServletRequest request, HttpServletResponse
response, HttpSession session) {
        String id = request.getParameter("id");

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

3.2.2 简单类型参数

简单数据类型 即 八种基本数据类型及其包装类型。

```
 /**
     * SpringMVC 接收简单数据类型参数  url：/demo/handle03?id=1
     *
     * 注意：接收简单数据类型参数，直接在handler方法的形参中声明即可，框架会取出参数值然后绑定
到对应参数上
     * 要求：传递的参数名和声明的形参名称保持一致
     * 如果不一致（url里是ids,后端是id），就取不到，除非用 @RequestParam("ids")，就可以把
ids的值 mapping给 id。
     * Best practice：建议用包装类型，不要用int，用Integer。
     *
     * Boolean 类型 只接收 4 个参数：  0，  1，  true，false （其他数字或character会报错）
     */
    @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;
    }
```

3.2.3 Pojo类型参数

```
public class User {

    private Integer id;
    private String name;

// setter & getter methods
  ...
}
```

```
/**
 * SpringMVC接收pojo类型参数  url：/demo/handle04?id=1&name=zhangsan
 *
 * 接收pojo类型参数，直接形参声明即可，类型就是Pojo的类型，形参名无所谓（即input为 User asuka 也行）
 * 但是要求传递的参数名(即url中的 id，name)必须和Pojo的属性名(id，name)保持一致，
 * 这样才能通过反射来set值。
 */
@RequestMapping("/handle04")
public ModelAndView handle04(User user) {

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

3.2.4 Pojo包装对象参数

```
public class QueryVo {

    private String mail;
    private String phone;

    // 嵌套了另外的Pojo对象
    private User user;

  // setter & getter methods
  ...
}
```

```
/**
 * SpringMVC接收pojo包装类型参数  url：/demo/handle05?user.id=1&user.name=zhangsan
 *
 * 这里的 QueryVo 把 User 包了进来。
 * 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
 * user 也是pojo，所以也要让url中 user.xxx的传参xxx与 user中的属性 一致。
 */
@RequestMapping("/handle05")
public ModelAndView handle05(QueryVo queryVo) {

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

3.2.5 日期类型参数 （考虑format转换）

由于不能将url中的String翻译成java.util.Date （日期的format太多了），所以需要手动自定义类型转换器DateConverter class：

```
import org.springframework.core.convert.converter.Converter;

import java.text.ParseException;
import java.text.SimpleDateFormat;
```

```
import java.util.Date;


/*
*  implements Converter<S, T>
*  S: source, 源类型
 * T: target: 目标类型
* */
public class DateConverter implements Converter<String, Date> {
    @Override
    public Date convert(String source) {
        // String to Date
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
// format即url中的format

        try {
            Date parse = simpleDateFormat.parse(source);
            return parse;
        } catch (ParseException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

然后，在springmvc.xml 中，注册这个实现类进SpringMVC：

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

然后，关联给HandlerAdapter：

```
<!--
        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)
        由于自己建的conversionServiceBean是run在Handler方法中的，所以要通过conversion-
service 属性进行绑定。
    -->
<mvc:annotation-driven conversion-service="conversionServiceBean"/>
```

4. 对Restful风格请求支持

4.1 什么是Restful风格请求?

Rest是一个url请求的风格，基于这种风格设计请求的url。

资源：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。

对资源的操作(method)：使用HTTP请求中的4种方法put、delete、**post**、**get**来操作资源。不过一般使用时还是 post 和 get，put 和 delete几乎不使用。

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，**每种资源对应一个独一无二的 URI** 。要获取这个资源，访问它的 URI 就可以。
- 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。

4.2 Spring MVC如何支持Restful风格请求

使用 @PathVariable 注解获取 RESTful 风格的请求url中取出uri的参数。

URL一样，但RequestMethod 不同，使用的Handler方法就不同。

- 后端Handler方法群:

```
/*
```

```
 * restful  get   /demo/handle/15
 *
 * @RequestMapping 的value 中的 {xxx} 表示xxx是个动态参数，xxx是参数名字
 * method 来限定请求的操作方法
 * @PathVariable 可取出url中的参数名字
 */
@RequestMapping(value = "/handle/{id}",method = {RequestMethod.GET})
public ModelAndView handleGet(@PathVariable("id") Integer id) {

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

/*
* restful  post  /demo/handle
* 在web.xml中添加了过滤器来应对POST方法的乱码问题。
*/
@RequestMapping(value = "/handle",method = {RequestMethod.POST})
public ModelAndView handlePost(String username) {...}


/*
* restful  put  /demo/handle/15/lisi
*/
@RequestMapping(value = "/handle/{id}/{name}",method = {RequestMethod.PUT})
public ModelAndView handlePut(@PathVariable("id") Integer
id,@PathVariable("name") String username) {...}

/*
 * restful  delete  /demo/handle/15
 */
@RequestMapping(value = "/handle/{id}",method = {RequestMethod.DELETE})
public ModelAndView handleDelete(@PathVariable("id") Integer id) {...}
```

- 在web.xml中添加了过滤器来应对POST方法的乱码问题：

```
 <!--springmvc提供的针对post请求的编码过滤器，并且指明编码方式用UTF-8-->
  <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-mapping>
    <filter-name>encoding</filter-name>
    <!-- SpringMVC 会拦截吸收所有 url   -->
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

> GET 方法乱码问题需要修改tomcat下的 server.xml 配置（见讲义 第六部分 附录一）

- 【注意】为了使用PUT和DELETE方法：
  - 前端的

    block 需要 method="post", type="hidden"， 且 name = "_method":

    ```
      <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>
    ```

  - web.xml file中，增加HiddenHttpMethodFilter 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>hiddenHttpMethodFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

5. Ajax JSON 交互

交互，两个方向：

- 前端到后台: 前端ajax发送json格式字符串，后台接收的是pojo参数，使用注解@RequstBody
- 后台到前端: 后台直接返回pojo对象，前端接收的是json对象或者字符串，使用注解 @ResponseBody

实现Spring MVC 使用Json交互：

- 前端（一个按钮）：

```
<div>
    <h2>Ajax json交互</h2>
    <fieldset>
        <input type="button" id="ajaxBtn" value="ajax提交"/>
    </fieldset>
</div>
```

- 使用jQuery（path="WEB-INF/js/jquery.min.js"）：直接复制粘贴吧。

- 在index.jsp中引入jQuery并给按钮绑定一个event，被触发后会发送一个请求给后端：

```
<script type="text/javascript" src="/js/jquery.min.js"></script>
```

```
<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',
                dataType: 'json',
                success: function (data) {
                    alert(data.name);
                }
            })
        })
    })
</script>
```

- 引入jar包：

```
<!--json数据交互所需jar，start-->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.9.0</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.0</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-annotations</artifactId>
  <version>2.9.0</version>
</dependency>
<!--json数据交互所需jar，end-->
```

- 前端到后台（前端发json过来，使用 @RequestBody 后台就能直接接收到pojo User）：

```
    @RequestMapping("/handle07")
    public ModelAndView handle07(@RequestBody User user) {
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("myDate",date);
        modelAndView.setViewName("success");
        return modelAndView;
     }
```

- 后台到前端（后台返回pojo对象，使用 @ResponseBody 前端就能直接收到json格式的response）：

```
@RequestMapping("/handle07")
public @ResponseBody User handle07(@RequestBody User user) {
// 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

  // 业务逻辑处理，修改name为张三丰
     user.setName("张三丰");
  // 这里返回pojo对象
     return user;
}
```

Part II. Spring MVC高级技术

1. Interceptor 使用

1.1 监听器（Listener）vs. 过滤器（Filter）vs. 拦截器（Interceptor）

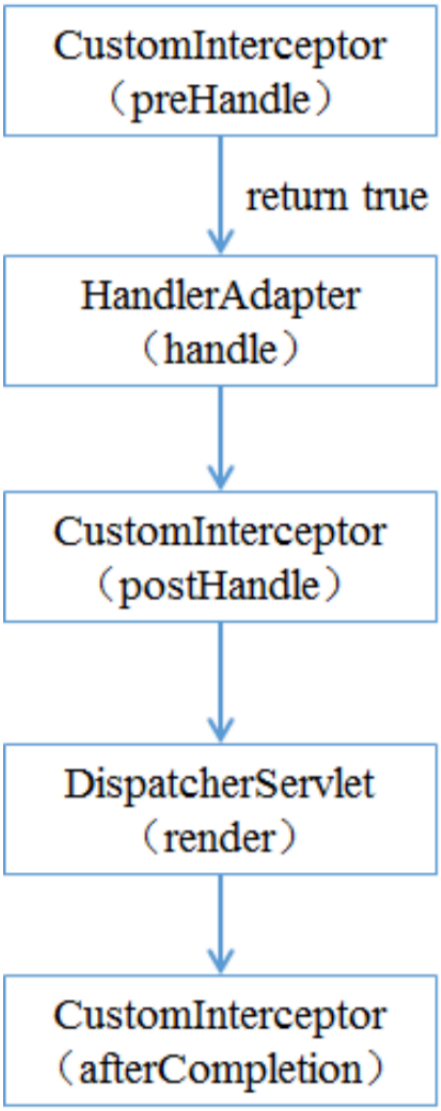
1. Servlet：处理Request请求和Response响应。**配置在web.xml中**。
2. Filter：对Request请求起到过滤的作用，作用在Servlet之前，如果他的配置为 /* 就可以对所有的资源访问 (包括servlet、js/css静态资源等) 进行过滤处理。**配置在web.xml中**。
3. 例如：针对post请求的编码过滤器 CharacterEncodingFilter， 针对put/delete的HiddenHttpMethodFilter 等。
4. Listener：实现了javax.servlet.ServletContextListener Interface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。**配置在web.xml中**。
    1. 作用一：做一些初始化工作。例如 web应用中spring容器启动ContextLoaderListener

2. 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。

5. Interceptor：是SpringMVC、Struts等表现层框架自己的组件，**配置在springmvc自己的配置文件中**。且不会拦截对 jsp/html/css/image 等静态文件的访问等，**只会拦截访问的Handler方法**。有3个地方可以拦截：
    1. 在Handler的业务逻辑执行之前 拦截一次　【常用，比如权限检查】
    2. 在Handler所有业务逻辑执行完毕但未跳转页面之前 拦截一次
    3. 在跳转页面之后（即视图渲染完毕后） 拦截一次

1.2 单个Interceptor执行流程

1.2.1 流程



1.2.2 自定义SpringMVC Interceptor

Step 1: 创建实现类implements HandlerInterceptor Interface：

```
public class MyIntercepter01 implements HandlerInterceptor {

    /**
     * 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;
    }


    /**
     * 在handler方法 业务逻辑执行之后 尚未跳转页面时 执行
     *
     * @param request
     * @param response
     * @param handler
     * @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
     * @throws Exception
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("MyIntercepter01 postHandle......");
    }

    /**
     * 页面已经跳转渲染完毕之后执行
     *
     * @param request
     * @param response
     * @param handler
```

```
     * @param ex  可以在这里捕获异常
     * @throws Exception
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("MyIntercepter01 afterCompletion......");
    }
}
```

Step 2：在springmvc.xml中注册自制Interceptor：

```
<!-- 注册自定义Interceptor -->
  <mvc:interceptors>
    <!--拦截所有handler-->
    <!--<bean class="com.lagou.edu.interceptor.MyIntercepter01"/>-->

    <mvc:interceptor>
      <!--配置当前拦截器的url拦截规则，/表示url根目录，**代表当前目录下及其子目录下的所有url-
->
      <mvc:mapping path="/**"/>

      <!--exclude-mapping可以在mapping的基础上排除一些url拦截，exclude之前必须已经有
<mvc:mapping>block。这里拦截了demo目录下的所有url-->
      <!--<mvc:exclude-mapping path="/demo/**"/>-->

      <bean class="com.lagou.edu.interceptor.MyInterceptor01"/>
    </mvc:interceptor>

  </mvc:interceptors>
```

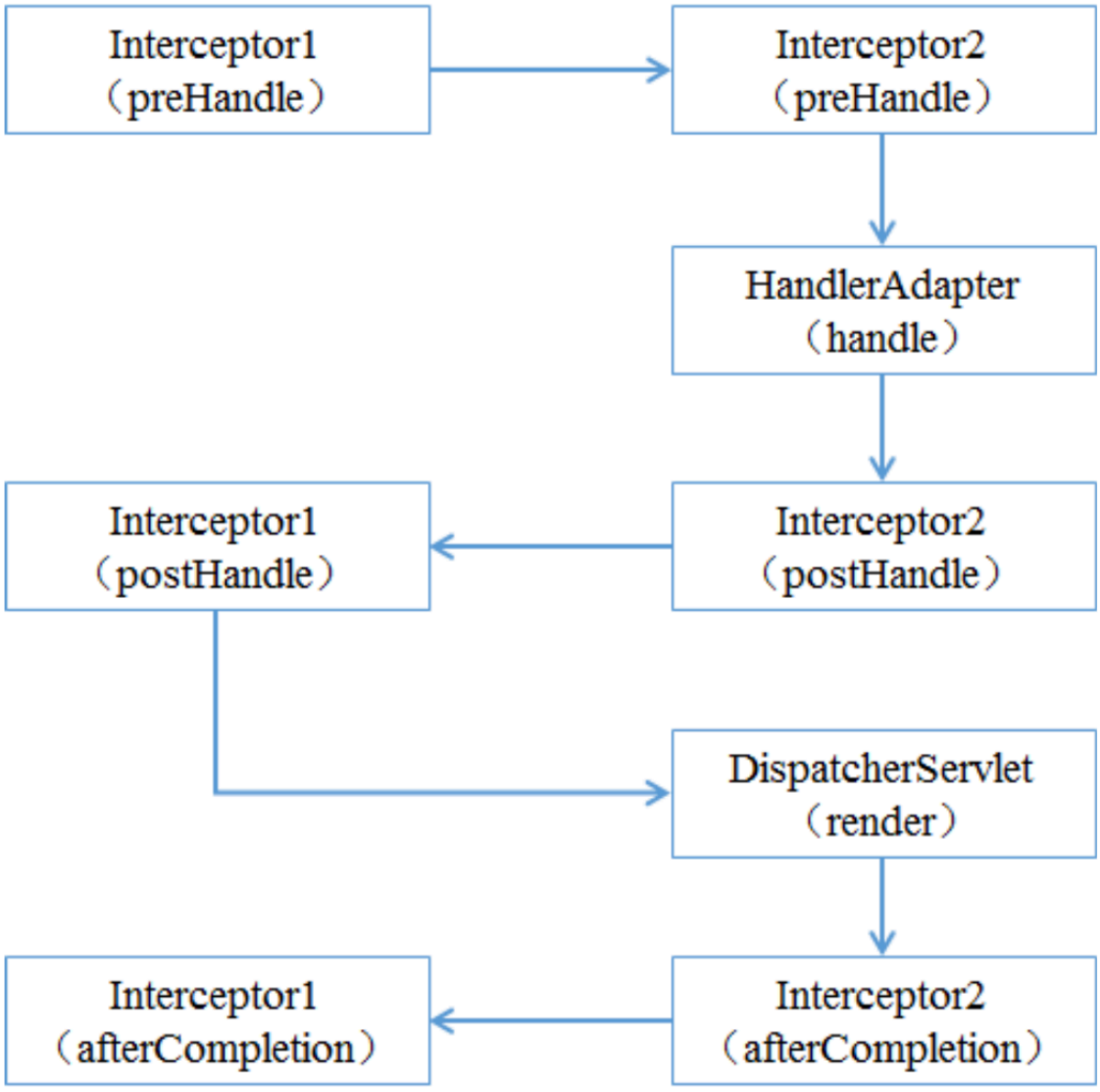
1.3 多个Interceptor的执行流程

每个Interceptor有3个拦截时机。添加 MyInterceptor02.java并注册【注意此时01在02之前】：

```
<!-- 注册自定义Interceptor   -->
<mvc:interceptors>
   <!--拦截所有handler-->
   <!--<bean class="com.lagou.edu.interceptor.MyIntercepter01"/>-->
```

```
    <!-- 注册 MyInterceptor01    -->
    <mvc:interceptor>
        <!--配置当前拦截器的url拦截规则，/表示url根目录，**代表当前目录下及其子目录下的所有url-
->
        <mvc:mapping path="/**"/>

        <!--exclude-mapping可以在mapping的基础上排除一些url拦截，比如demo目录下的所有url-->
        <!--<mvc:exclude-mapping path="/demo/**"/>-->

        <bean class="com.lagou.edu.interceptor.MyInterceptor01"/>
    </mvc:interceptor>

    <!-- 注册 MyInterceptor02    -->
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.lagou.edu.interceptor.MyInterceptor02"/>
    </mvc:interceptor>

</mvc:interceptors>
```

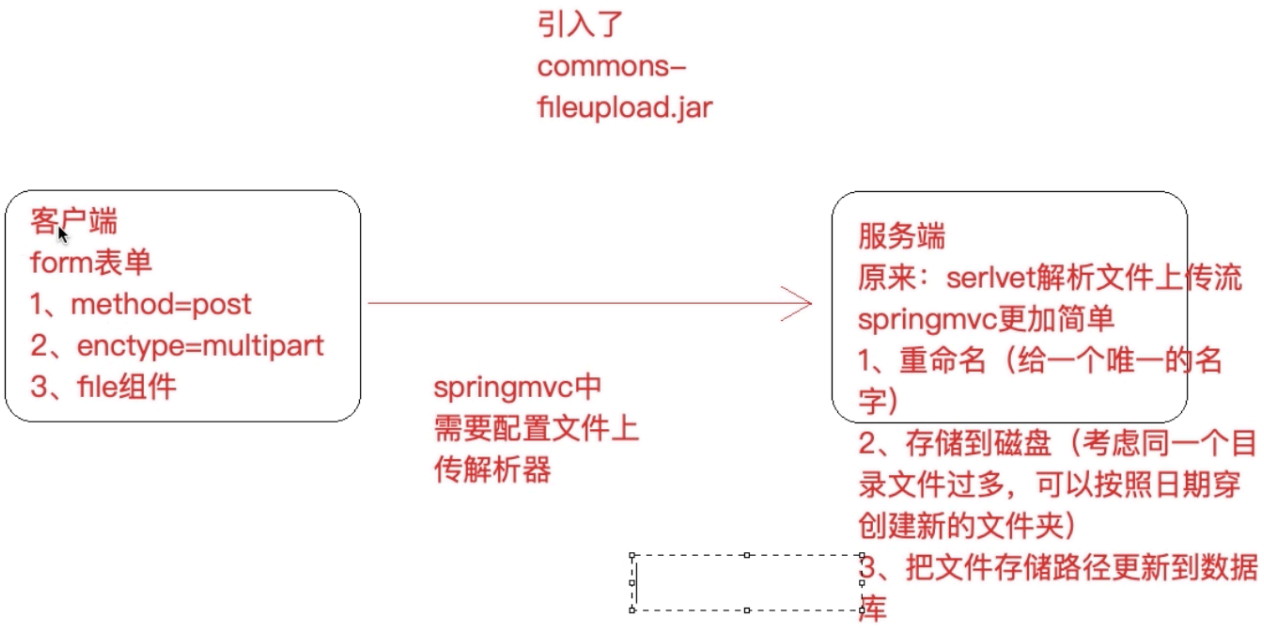
运行顺序见下图（与xml中的配置顺序有关）：



2. 处理multipart形式的数据（即如何上传文件）

原生servlet处理上传的文件数据的，springmvc又是对serlvet的封装。

enctype=multipart 是默认的形式。



Step 1: 引入dependency (pom.xml)

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

Step 2: 开发客户端 （index.jsp）

【注意】method="post" enctype="multipart/form-data" type="file"

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

Step 3: 开发服务端 （FileUploadController.java）

input中的“uploadFile”需要和 index.jsp中的 `<input type="file" name="uploadFile"/>` 的name一致。

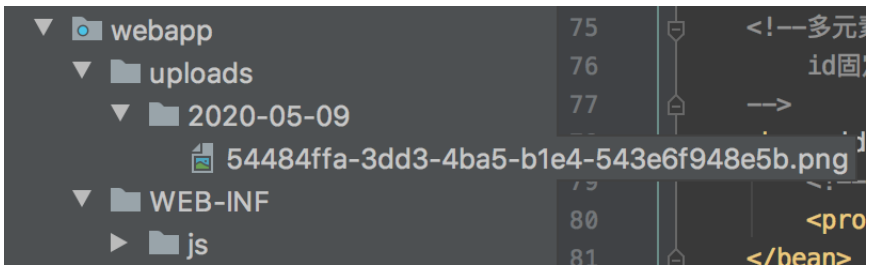
```
@RequestMapping(value = "/upload")
public ModelAndView upload(MultipartFile uploadFile, HttpSession session) throws
IOException {

     // Step 1: 处理上传文件
        // 原名"123.jpg" ，获取后缀
        String originalFilename = uploadFile.getOriginalFilename();
        // 扩展名  jpg
        String ext = originalFilename.substring(originalFilename.lastIndexOf(".")
+ 1, originalFilename.length());
        String newName = UUID.randomUUID().toString() + "." + ext;

     // Step 2: 存储到指定的文件夹，比如webapp下面的 /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

        // 通过session来获得 相对于当前项目（即webapp）的路径，/uploads 即相当于 webapp下
面的uploads
        String realPath = session.getServletContext().getRealPath("/uploads");

        // 解析现在的日子，然后create出以它命名的folder （注意判断folder存在与否）
        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));

      // Step 3: TODO 文件磁盘路径要更新到数据库字段

        // 若成功会返回下面这个modelAndView页面
        Date date = new Date();
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("myDate",date);
        modelAndView.setViewName("success");
        return modelAndView;
}
```

Step 4: 配置上传解析器 （springmvc.xml） id固定为multipartResolver,这样sprinMVC才能去找到对应的bean

```
<!--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，value类型是long，单位是字节，-1代表没有限制也是默认的-->
    <property name="maxUploadSize" value="5000000"/>
</bean>
```

Test:

随便上传个东西后，可以发现webapp下面会新增一个uploads folder，且其中有日期命名的folder并包含上传的文件，上传的文件的名字是UUID.扩展名



3. 处理异常（全局的Handler更优雅）

Dao层异常往Service层抛，Service层收到异常往表现层抛，所以可以用SpringMVC的Handler来捕获并处理异常。

可以单独在某个Controller class中写一个ExceptionHandler来只handle这个class中的某个或多个Exception，但是这样写过于局限了：

```
@Controller
@RequestMapping("/demo")
public class DemoController  {

    // 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());
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
    ...
```

应该自定义一个全局的Handler（背后implements了HttpExceptionResolver Interface）来捕获并处理所有Controller中的Handler方法可能抛出的异常：

```
// 可以优雅捕获所有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");  // 会跳转到新创建error.jsp文件
        return modelAndView;
    }
}
```

4. 用flash属性来构建重定向handler

转发 (forward) vs. 重定向 (redirect)：

- 转发 (forward)
  - 类似场景：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。
  - 对A来说，从头到尾相当于只有一个请求，所以URL不会变, 参数也不会丢失。
- 重定向 (redirect)
  - 类似场景：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。
  - 对A来说，这里需要发两个请求，所以url会变, 参数会丢失因为新请求需要重新携带参数。

Step 1: 创建最终访问的url的handler：

```
/*
    * URL = localhost:8080/demo/handleRedirect?name=Jenny
    *
    * Input 的 @ModelAttribute("name") String name
    * 意思是将 Model中的"name" attribute 绑定给 参数name，然后下面的方法就可以用这个参数
name
    * */
    @RequestMapping("/handle001")
    public ModelAndView handle001(@ModelAttribute("name") String name) {

        int c = 1/0; // 制造异常
...
```

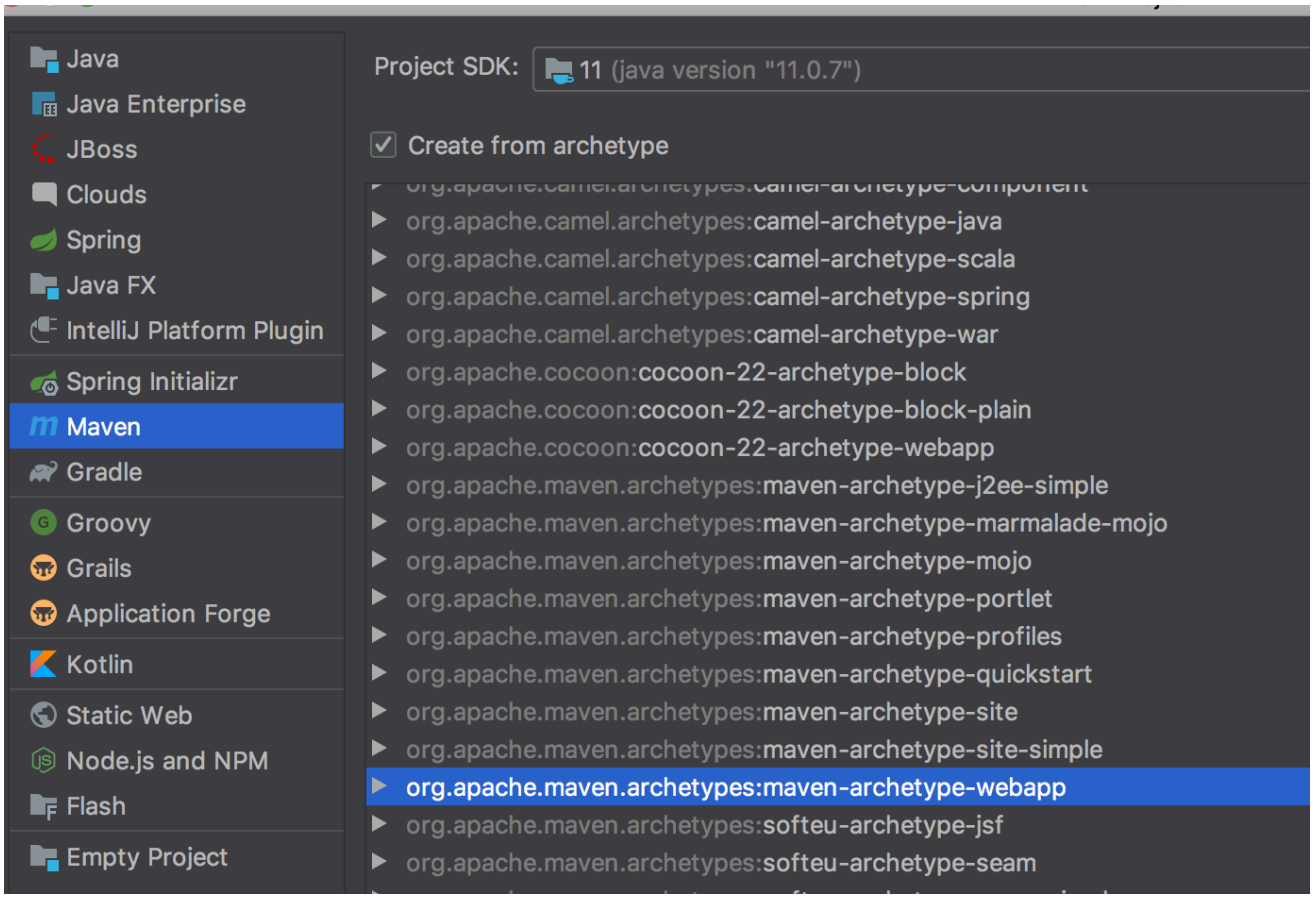
Step 2：创建重定向handler：

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

        /*
        * 直接return的话，handle001 的handler并拿不到name的值，因为这是重定向，参数会丢失
        * */
        // return "redirect:handle001";

        /*
         * 把参数拼接到return的url上，这样可以让handle001 的handler拿到name的值，
         * 但是安全性、参数长度都有局限
         * */
        // return "redirect:handle01?name=" + name;
        // 相当于 redirectAttributes.addAttribute("?name" + name);

        /*
        * 在input中增加 RedirectAttributes 参数，
        * 其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:handle001";

    }
```

Extra 1. Miscellaneous

1. 创建工程 `springmvc-demo`

Step 1: Create Web Application by maven



Step 2: 清理自动生成的配置。

1. maven complier的version改成11

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

2. 删除 block
3. junit version 改成12

Step 3: 补全code structure。

Create directory "java" and "resources" in "main" folder. Then 右键 mark "java" folder as "Sources Root", "resources" as "Resources Root".

Step 4：pom.xml中引入build plugin for tomcat:

```
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.2</version>
        <configuration>
          <port>8080</port>
          <path>/</path>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

Step 5: pom.xml中引入Spring web mvc的依赖：

```
 <!--引入spring webmvc的依赖-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>5.1.12.RELEASE</version>
    </dependency>
```

Step 6: web.xml中，给webapp配置全局的DispatcherServlet：

```
 <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
```

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

    <!--
 url-pattern标签写法：
      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，
      方式三：/* 拦截所有，包括.jsp
    -->
    <url-pattern>/</url-pattern>
  </servlet-mapping>
```

Step 7:

- 开始写DispatcherServlet背后的Controller，创建com.lagou.edu.controller.DemoController

  url: http://localhost:8080/demo/handle01 从根（类）开始匹配，再到方法匹配。

```
@Controller
@RequestMapping("/demo")
public class DemoController  {

    @RequestMapping("/handle01")
    public ModelAndView handle01(@ModelAttribute("name") String name) {
        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("myDate",date);

        // 视图信息(封装跳转的页面信息)
        modelAndView.setViewName("/WEB-INF/jsp/success.jsp");

        return modelAndView;
    }
```

- 创建跳转页面的success.jsp file (path = “src/main/webapp/WEB-INF/jsp/success.jsp”)

Step 8: 创建Spring的config文件（springmvc.xml）在“resources”folder中，来扫描pkg 中的annotation，并且加入视图解析器(InternalResourceViewResolver):

```
<?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"
     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
">
  <!--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>

</beans>
```

Step 9: 如果不单独配置处理器映射器，处理器适配器，会使用默认的。这里额外引入了Spring mvc的处理器映射器，处理器适配器：

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

    <!--
        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/>

</beans>
```

Step 10: 在web.xml中关联springmvc.xml配置文件

```
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <!--  关联springmvc.xml配置文件  -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
  </servlet>
```

Step 11: Test & Debug

- Test result:

  服务器时间是：${myDate}

- Bug Fix：需要Add isELIgnored="false" attribute into page directive tag. So that the web page will treat ${...} as static text instead of being evaluated by JSP:

  ```
  <%@ page isELIgnored="false" contentType="text/html;charset=UTF-8"
  language="java" %>
  ```

  Refer to - https://www.tutorialspoint.com/jsp/page_directive.htm

- Now test result:

  服务器时间是：Fri May 08 16:40:08 PDT 2020

2. web.xml中的 `<servlet-mapping>`中的`<url-pattern>` （三种方式）

```
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <!--  关联springmvc.xml配置文件  -->
    <init-param>
```

```
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>


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

说明：

此处servlet是用的SpringMVC，其servlet-mapping中的url-pattern标签表明的意思为 只要遇到这种pattern的url，就将其转给SpringMVC框架处理，即这里的url-pattern是给SpringMVC框架吸收并控制的。

2.1 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

2.2 方式二：/

- 不会拦截 .jsp 文件，即不会被SpringMVC框架吸收，那么tomcat就能够顺利实现 .jsp 文件的页面跳转。
- 会拦截 .html 等静态资源（除了servlet和jsp之外的js、css、png等），即会被SpringMVC框架吸收，tomcat无法跳转到静态资源定义的页面。

> 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），是一个子继承父的继承关系。
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，从而不能够直接拿到静态资源。

> 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的处理就绕过了SpringMVC交给了tomcat。

> 如何解决 / 拦截静态资源这件事?

```
<!-- 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。

    限制：静态资源只能放在"/webapp" folder中，而不能放在"WEB-INF" 或者 classpath（即jar包
等）
-->
 <mvc:default-servlet-handler/>
```

```
<!--静态资源配置 方案二 - SpringMVC框架自己处理静态资源
    语法：遇到mapping定义的url pattern时，就去location定义的路径里面找。

    所以，可以定义如下映射规则：
        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/**"/>

<!-- 会去 "/webapp" folder 和 "com.lagou.edu" pkg中去找  -->
<mvc:resources location="/webapp,classpath:/"  mapping="/resources/**"/>

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

2.3 方式三：/*

- SpringMVC框架会拦截并吸收所有，包括.jsp，tomcat啥都拿不到。

3. 数据输出机制之Model、Map及ModelMap回顾

```
@Controller
@RequestMapping("/demo")
public class DemoController {
   /*
    * 声明 ModelMap
    * url: http://localhost:8080/demo/handle11
     * */
```

```
    @RequestMapping("/handle11")
    public String handle11(ModelMap modelMap) {
        Date date = new Date();
        modelMap.addAttribute("myDate", date);

        return "success";
    }

    /*
     * 声明 Model
     * url: http://localhost:8080/demo/handle12
     * */
    @RequestMapping("/handle12")
    public String handle12(Model model) {
        Date date = new Date();
        model.addAttribute("myDate", date);

        return "success";
    }

    /*
     * 声明 Map 集合
     * url: http://localhost:8080/demo/handle13
     * */
    @RequestMapping("/handle13")
    public String handle13(Map<String, Object> map) {
        Date date = new Date();
        map.put("myDate", date);

        return "success";
    }
}
```

SpringMVC 在Handler方法上，input中传入Map，Model和ModelMap参数，并向这些参数中存入数据，所有方式都能通过url访问到存入的数据。

通过打印运行时的class能发现，这3个东西在运行时候的具体类型相同，都是`org.springframework.validation.support.BindingAwareModelMap`,都相当于给这个`BindingAwareModelMap`中保存的数据 都会放在请求域中。

三者关系：

```
Map(jdk中的Interface)        Model (spring的Interface)
ModelMap(class,实现Map Interface)

BindingAwareModelMap 继承了ExtendedModelMap,
ExtendedModelMap extends了 ModelMap class 并 实现了Model Interface
```

可见三者都能用**BindingAwareModelMap**来实现出来。