第1天: SpringMVC

一、学习目标

1. 能够介绍SpringMVC框架
2. 能够说出入门案例的执行过程
3. 能够说出SpringMVC常用组件
4. 掌握@RequestMapping的使用
5. 掌握SpringMVC的参数绑定

二、SpringMVC的概念

2.1 MVC和三层架构

2.1.1 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。

序号	三层	职责
1	表现层	表现层指的是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：展示层负责结果的展示，控制层负责接收请求。 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。 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）
2	业务层	业务层指的是 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保证事务一致性。 业务层还需要事务，事务应该放到业务层来控制）
3	持久层	持久层指的是 dao 层。 负责数据持久化，包括数据层即数据库和数据访问层。 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，对数据库表进行曾删改查的。

2.1.2 MVC模型

> MVC 全名是 Model View Controller，是模型(model) - 视图(view) - 控制器(controller)的缩写，是一种用于设计创建 Web 应用程序表现层的模式。

号		
1	**Model**（模型）	通常指的就是我们的数据模型。作用一般情况下用于封装数据。
2	**View**（视图）	通常指的就是页面。作用一般就是展示数据的。
3	**Controller**（控制器）	通常指的就是后端的控制器。控制着页面跳转, 页面显示和数据处理。

2.2 SpringMVC概诉

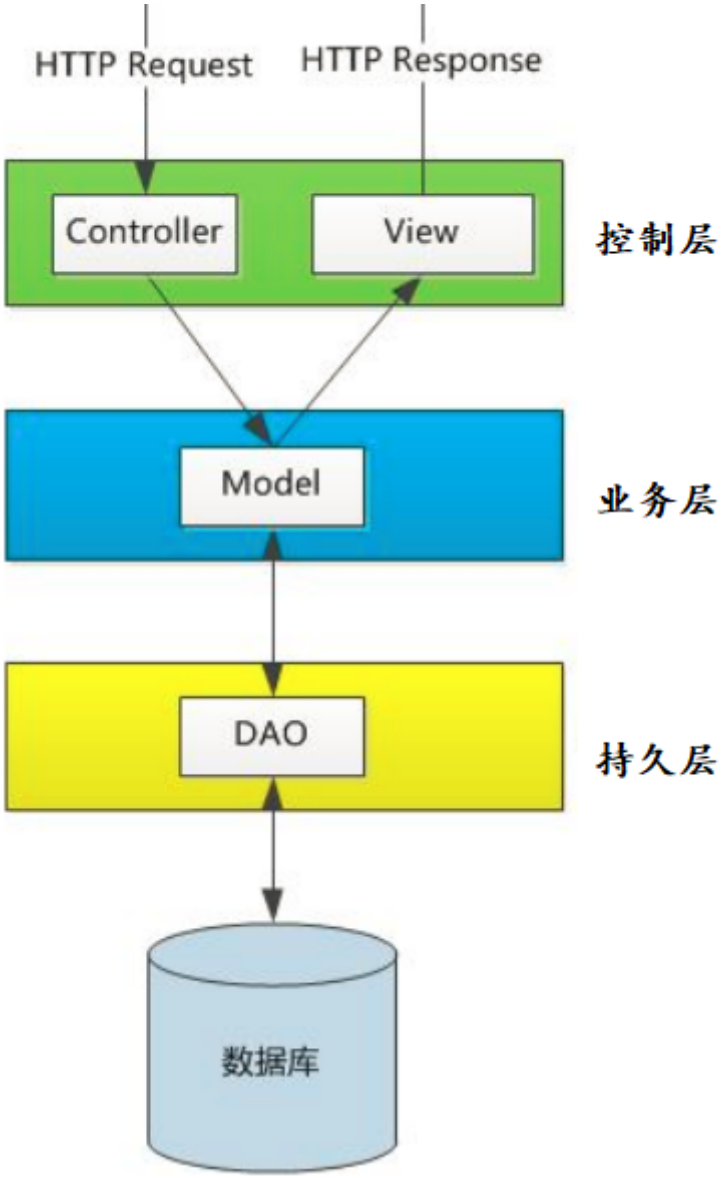
2.2.1 SpringMVC是什么

> Springmvc 是一种基于 MVC 设计模型的请求驱动类型的轻量级 Web 框架， 属于 SpringFrameWork 的后续产品，已经融合在 Spring Web Flow 里面。

> Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 spring 可插入的 MVC 架构，从而在使用 spring 进行 WEB 开发时，可以选择使用 spring的 spring MVC 框架或集成其他 MVC 开发框架，如 Struts1(现在一般不用)， Struts2 等。

> Springmvc 已经成为目前最主流的 MVC 框架之一， 从 Spring3.0 的发布， 就已全面超越 Struts2，成为最优秀的 MVC 框架。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。同时它还支持RESTful 编程风格的请求。

2.2.2 在3层架构的位置



2.2.3 SpringMVC的优点

序号	优点	详情
1	清晰的角色划分	前端控制器（DispatcherServlet） 处理器映射器（HandlerMapping） 处理器适配器（HandlerAdapter） 视图解析器（ViewResolver） 处理器或页面控制器（Controller） 验证器（Validator） 命令对象（Command 请求参数绑定到的对象就叫命令对象） 表单对象（Form Object 提供给表单展示和表单提交的对象）
2	可扩展性好	可以很容易扩展，虽然几乎不需要
3	与Spring 框架无缝集成	这是其它web框架不具备的
4	可适配性好	通过 HandlerAdapter 可以支持任意的类作为处理器

6	单元测试方便	利用 Spring 提供的 Mock 对象能够非常简单的进行 Web 层单元测试
7	本地化、主题的解析的支持	使我们更容易进行国际化和主题的切换
8	jsp标签库	强大的 JSP 标签库，使 JSP 编写更容易

三、SpringMVC入门案例

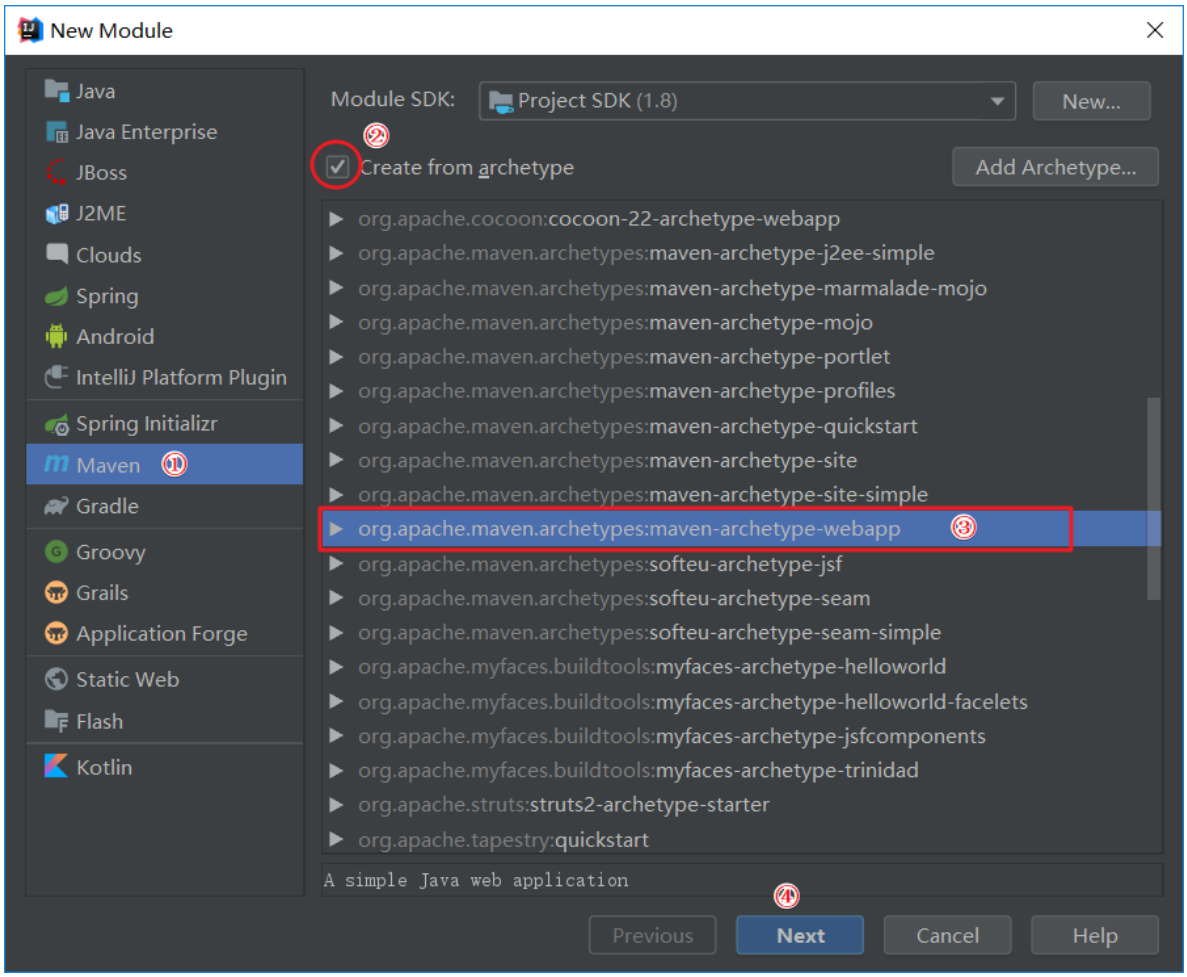
3.1 环境准备

序号	环境名称	环境要求
1	开发环境	1.8
2	开发工具	idea2019.1.3
3	web容器	tomcat8.x

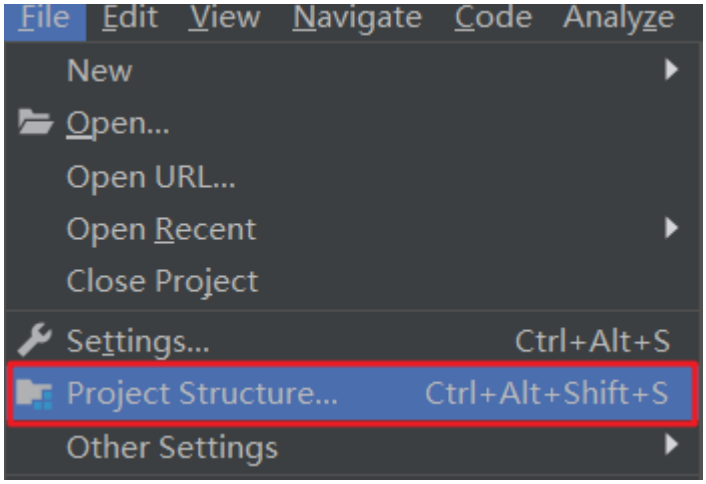
3.2 入门案例

3.2.1 创建web工程

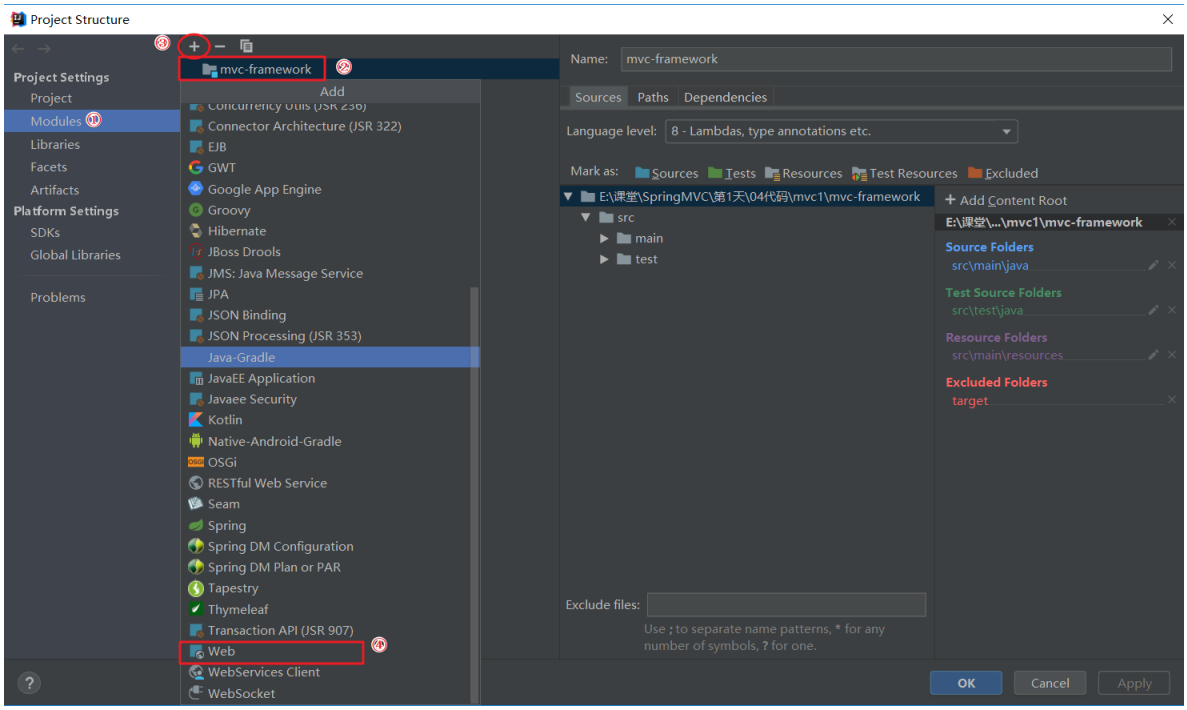
使用骨架创建web工程



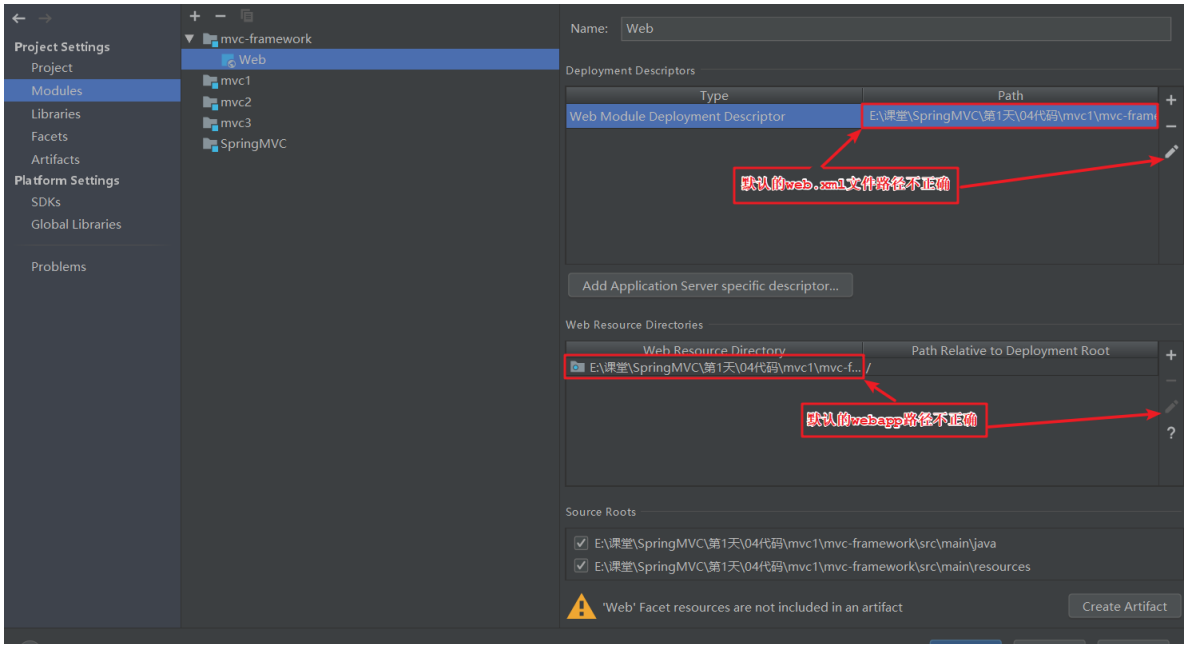
普通工程升级web工程

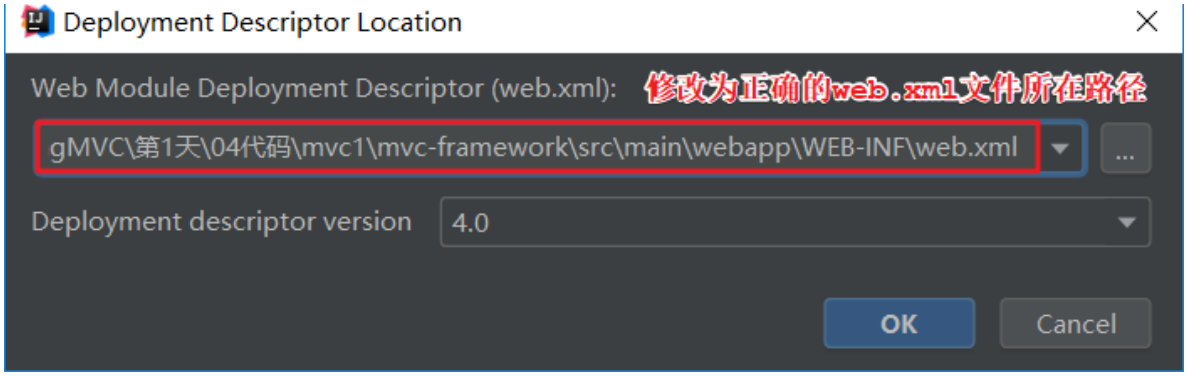


- ② 选择模块添加框架支持

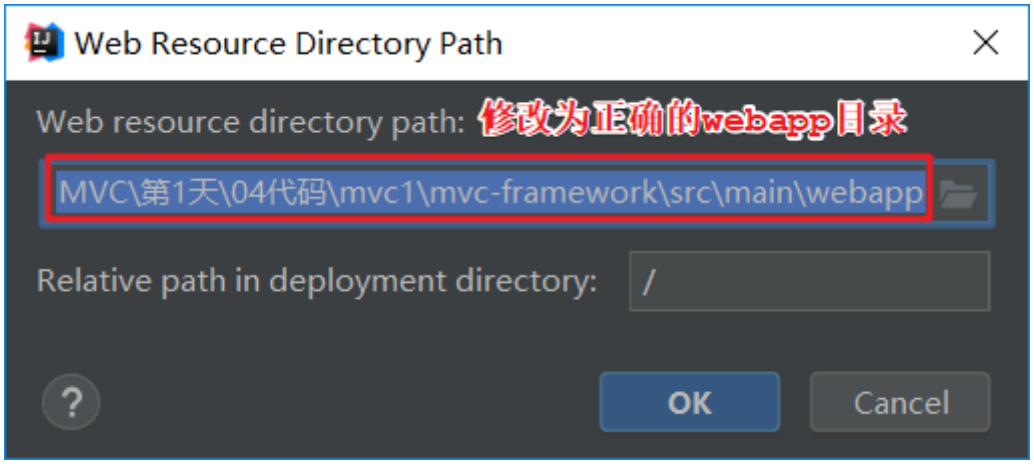


- ③ 需要手动修正的错误点

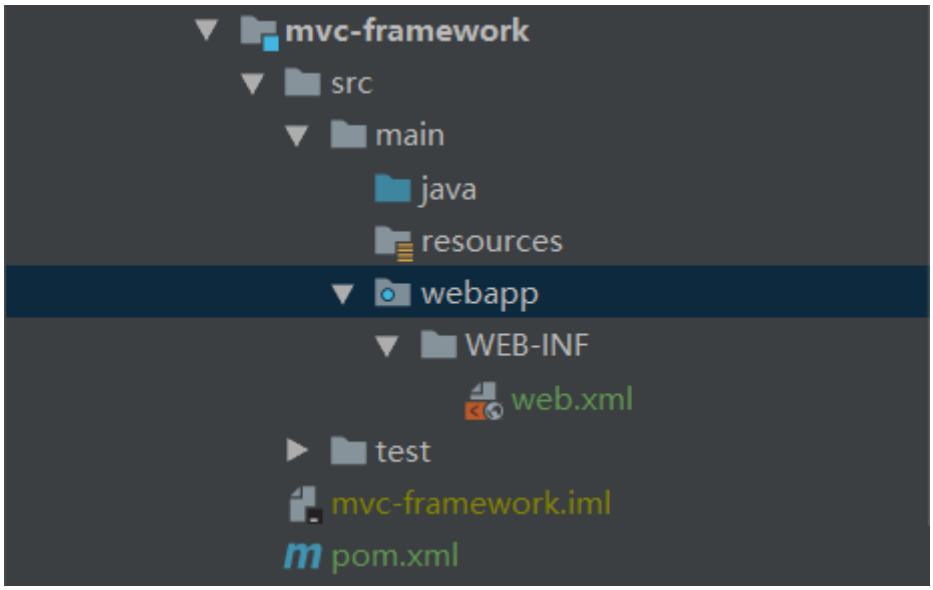




- ⑤ webapp目录修正

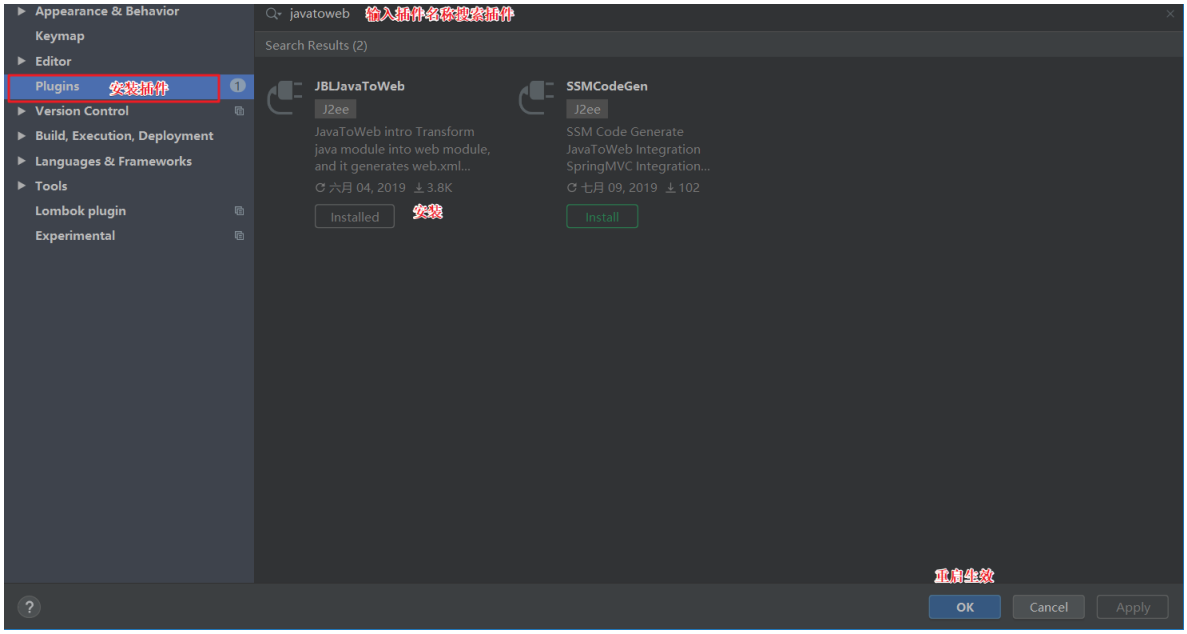


- ⑥ 完成

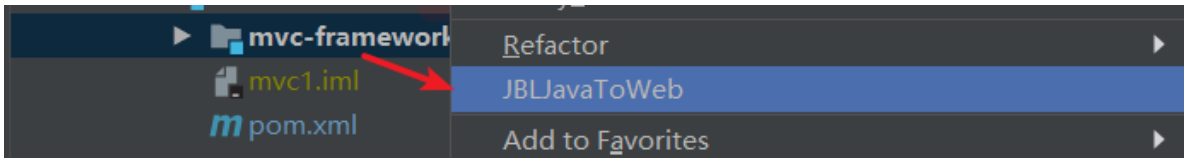


普通工程变身web工程

- 使用插件将普通工程转换成web工程



- 插件使用



3.2.2 添加依赖

- pom.xml

```
<!-- 打包类型 -->
<packaging>war</packaging>

<!-- 依赖配置 -->
<dependencies>
    <!-- 添加SpringMVC 依赖 -->
    <!-- 会间接依赖spring-context, spring-web等 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.1.7.RELEASE</version>
    </dependency>
</dependencies>
```

3.2.3 准备页面

- pages/success.jsp

```
<%--
  Created by IntelliJ IDEA.
  User: Jason
  Date: 2019/7/16
```

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>OK</title>
</head>
<body>
    hello !
</body>
</html>
```

3.2.4 创建控制器

- com.itheima.web.HelloController

```
package com.itheima.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * SpringMVC控制器案例.
 *
 * @Controller:
 *  专用于修饰SpringMVC的控制器,有特定的解析器解析该注解
 */
@Controller
public class HelloController {


    /**
     * WEB项目
     *  1. 请求示例: http://localhost:8080/hello.do
     *  2. 响应示例: /pages/success.jsp(可以拆分为3部分)
     *      2.1 前缀: /pages/
     *      2.2 后缀: .jsp
     *
     *  @RequestMapping: 将方法绑定到指定的请求路径 (处理指定的请求)
     *  @return: 返回值为响应的页面路径
     */
    @RequestMapping("hello.do")
    public String hello(){
        System.out.println("处理请求..");
        // 返回的页面路径
        return "success";
    }
}
```

3.2.5 添加配置

- springMVC.xml

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

    <!-- 1. 开启注解扫描，识别Controller控制器 -->
    <context:component-scan base-package="com.itheima.web"/>

    <!-- 2. 配置视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!-- 响应资源的路径前缀 -->
        <property name="prefix" value="/pages/"/>
        <!-- 响应资源的路径后缀 -->
        <property name="suffix" value=".jsp"/>
    </bean>

    <!-- 3. 开启SpringMVC注解驱动 -->
    <mvc:annotation-driven/>
    <!--
        <mvc:annotation-driven/>: 主要注册以下Bean
        <bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMapping
HandlerMapping"/>
        <bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMapping
HandlerAdapter"/>
        <bean
class="org.springframework.web.servlet.mvc.method.annotation.ExceptionHandl
erExceptionResolver"/>
    -->
</beans>
```

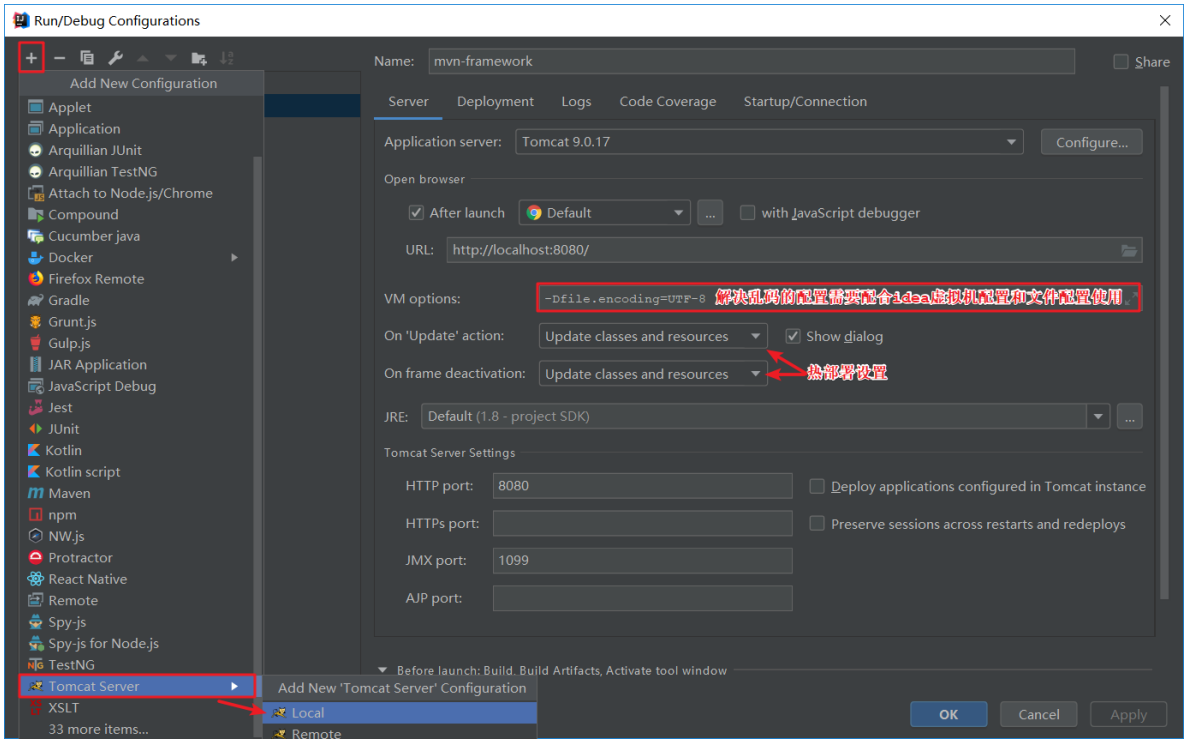
- web.xml

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

    <!-- 配置前端控制器 Start.. -->

    <!-- 1. 配置请求路径的拦截(映射) -->
    <servlet-mapping>
        <!-- servlet名称 -->
        <servlet-name>dispatcherServlet</servlet-name>
        <!-- 拦截的路径(*指任意字符) -->
```

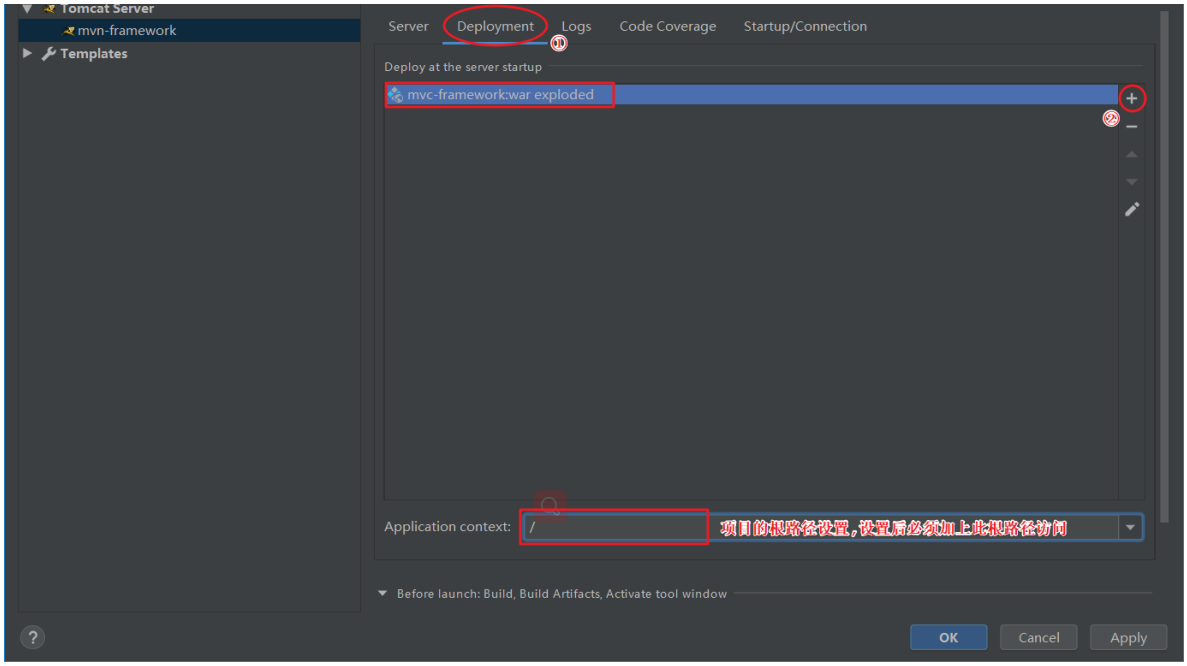
```
    <!-- 2. 配置DispatcherServlet(前端控制器) -->
    <servlet>
        <!-- servlet名称 -->
        <servlet-name>dispatcherServlet</servlet-name>
        <!-- servlet全限定类名称 -->
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 加载springMVC配置文件 -->
        <init-param>
            <!-- 创建Spring容器加载的配置名称 -->
            <param-name>contextConfigLocation</param-name>
            <!-- 创建Spring容器加载的配置文件 -->
            <param-value>classpath:springMVC.xml</param-value>
        </init-param>
        <!-- servlet加载顺序(正整数表示容器启动时加载) -->
        <!-- 取值: 0及小于0的值表示第一次请求是加载; 正整数值越小加载顺序越早 -->
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- 配置前端控制器 End.. -->
</web-app>
```

3.2.6 发布项目

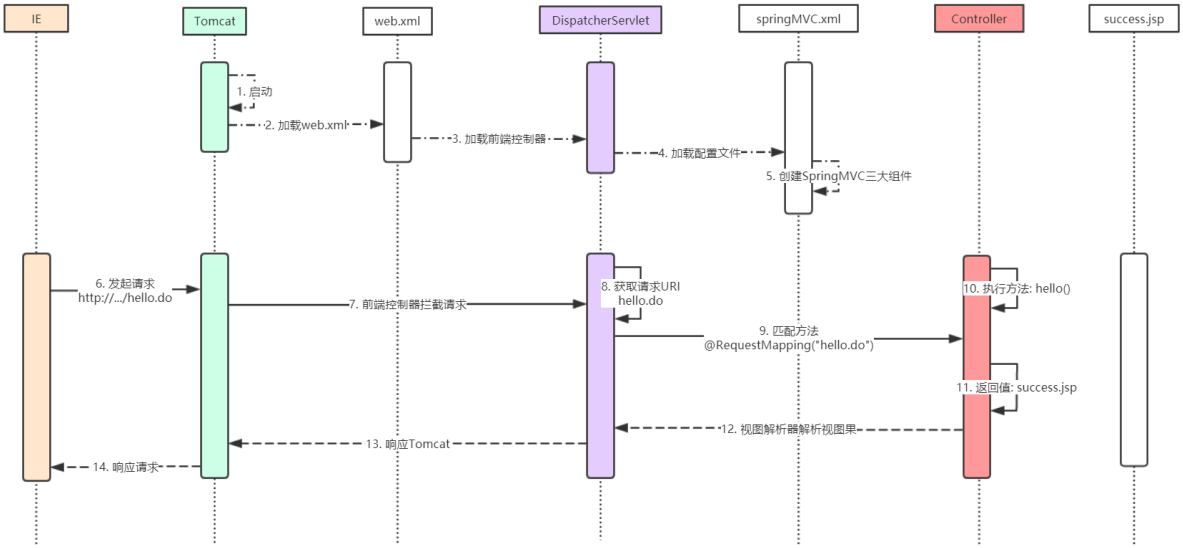
- Tomcat环境设置



- 部署工程选择



3.3 执行流程



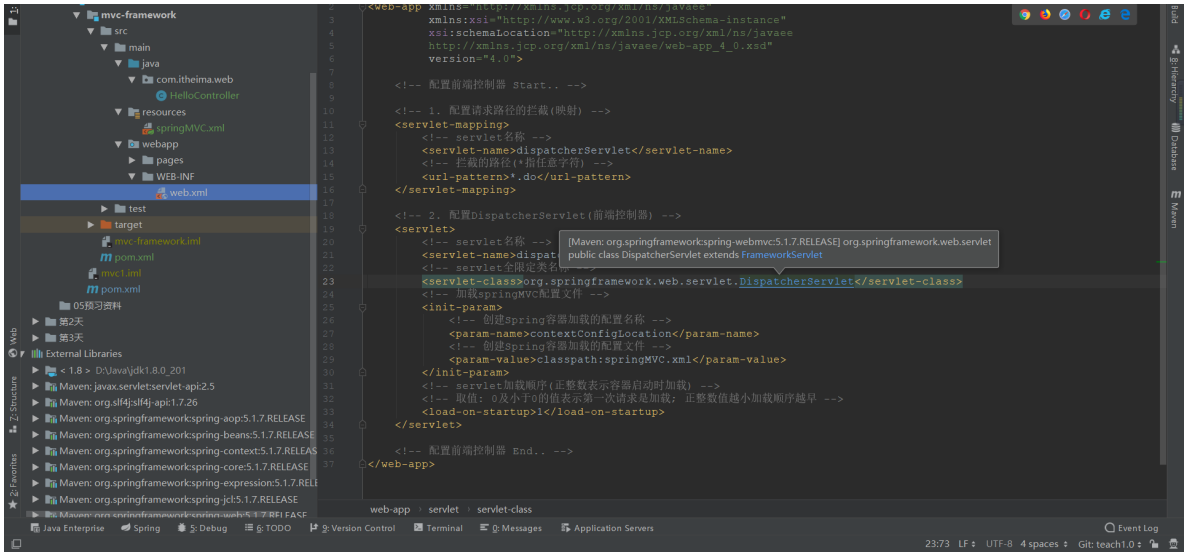
3.4 源码分析

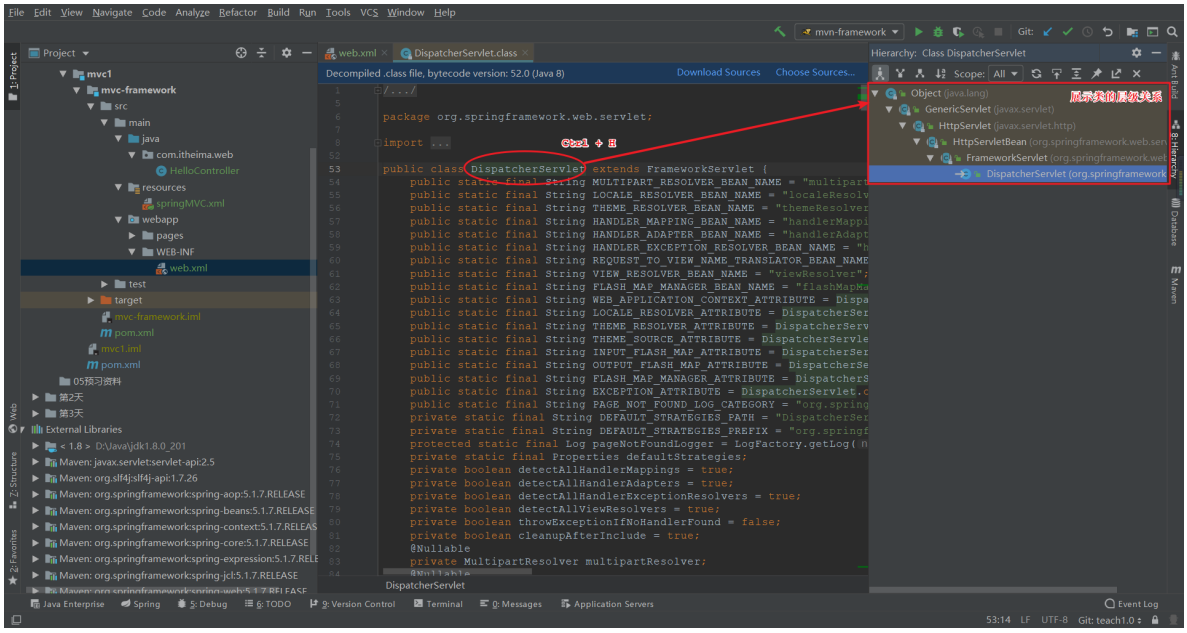
- SpringMVC源码版本: 5.1.7.RELEASE

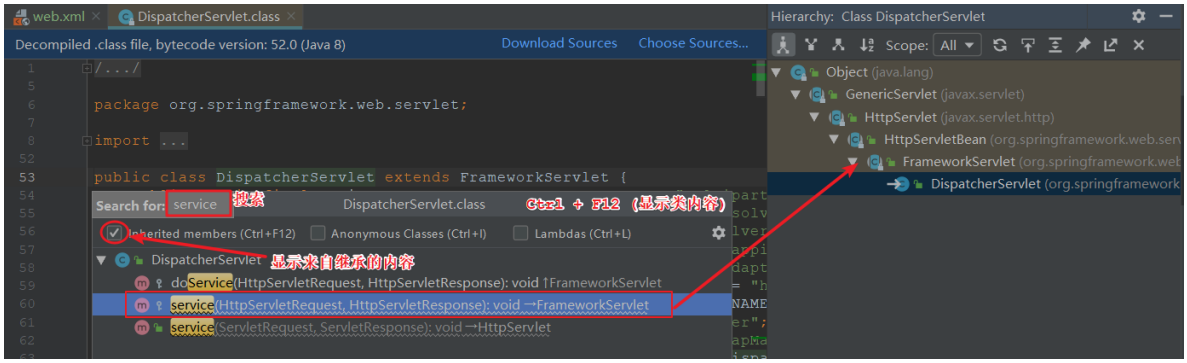
3.4.1 源码分析: Servlet

```java
public interface Servlet {
    // 初始化方法,构造方法后执行
    void init(ServletConfig var1);

    // 每次访问servlet时候执行
    void service(ServletRequest var1, ServletResponse var2);

    // 停止服务器时候执行
    void destroy();
}
```

3.4.2 源码追踪: DispatcherServlet







3.4.3 追踪源码: FrameworkServlet

```
    HttpMethod httpMethod = HttpMethod.resolve(request.getMethod());
    if (httpMethod != HttpMethod.PATCH && httpMethod != null) {
        super.service(request, response);
    } else {
        // 最终都会执行处理请求的方法
        this.processRequest(request, response);
    }
}
```

```
protected final void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
    long startTime = System.currentTimeMillis();
    Throwable failureCause = null;
    LocaleContext previousLocaleContext =
LocaleContextHolder.getLocaleContext();
    LocaleContext localeContext = this.buildLocaleContext(request);
    RequestAttributes previousAttributes =
RequestContextHolder.getRequestAttributes();
    ServletRequestAttributes requestAttributes =
this.buildRequestAttributes(request, response, previousAttributes);
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

 asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(),
new FrameworkServlet.RequestBindingInterceptor());
    this.initContextHolders(request, localeContext, requestAttributes);

    try {
        // 最终会调用doService方法; 该方法被DispatcherServlet重写了
        this.doService(request, response);
    } catch (IOException | ServletException var16) {
        failureCause = var16;
        throw var16;
    } catch (Throwable var17) {
        failureCause = var17;
        throw new NestedServletException("Request processing failed",
var17);
    } finally {
        this.resetContextHolders(request, previousLocaleContext,
previousAttributes);
        if (requestAttributes != null) {
            requestAttributes.requestCompleted();
        }

        this.logResult(request, response, (Throwable)failureCause,
asyncManager);
        this.publishRequestHandledEvent(request, response, startTime,
(Throwable)failureCause);
    }

}
```

3.4.4 回到源码: DispatcherServlet

```
    this.logRequest(request);
    Map<String, Object> attributesSnapshot = null;
    if (WebUtils.isIncludeRequest(request)) {
        attributesSnapshot = new HashMap();
        Enumeration attrNames = request.getAttributeNames();

        label95:
        while(true) {
            String attrName;
            do {
                if (!attrNames.hasMoreElements()) {
                    break label95;
                }

                attrName = (String)attrNames.nextElement();
            } while(!this.cleanupAfterInclude &&
!attrName.startsWith("org.springframework.web.servlet"));

            attributesSnapshot.put(attrName,
request.getAttribute(attrName));
        }
    }

    request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
this.getWebApplicationContext());
    request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
    request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
    request.setAttribute(THEME_SOURCE_ATTRIBUTE, this.getThemeSource());
    if (this.flashMapManager != null) {
        FlashMap inputFlashMap =
this.flashMapManager.retrieveAndUpdate(request, response);
        if (inputFlashMap != null) {
            request.setAttribute(INPUT_FLASH_MAP_ATTRIBUTE,
Collections.unmodifiableMap(inputFlashMap));
        }

        request.setAttribute(OUTPUT_FLASH_MAP_ATTRIBUTE, new FlashMap());
        request.setAttribute(FLASH_MAP_MANAGER_ATTRIBUTE,
this.flashMapManager);
    }

    try {
        // 最终都会执行doDispatch方法
        this.doDispatch(request, response);
    } finally {
        if
(!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted() &&
attributesSnapshot != null) {
            this.restoreAttributesAfterInclude(request,
attributesSnapshot);
        }

    }

}
```

```
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

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

            try {
                processedRequest = this.checkMultipart(request);
                multipartRequestParsed = processedRequest != request;
                // 根据URI获取到具体的处理器方法 (三大组件之一: 处理器映射器)
                mappedHandler = this.getHandler(processedRequest);
                if (mappedHandler == null) {
                    this.noHandlerFound(processedRequest, response);
                    return;
                }
                // 获取到处理器适配器 (三大组件之一: 处理器适配器)
                HandlerAdapter ha =
this.getHandlerAdapter(mappedHandler.getHandler());
                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if (isGet || "HEAD".equals(method)) {
                    long lastModified = ha.getLastModified(request,
mappedHandler.getHandler());
                    if ((new ServletWebRequest(request,
response)).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }

                if (!mappedHandler.applyPreHandle(processedRequest,
response)) {
                    return;
                }
                // 适配器通过反射调用处理器方法
                // 并返回视图和模型
                mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
                if (asyncManager.isConcurrentHandlingStarted()) {
                    return;
                }
                // 如果视图是空采用默认视图
                this.applyDefaultViewName(processedRequest, mv);
                // 拦截器处理
                mappedHandler.applyPostHandle(processedRequest, response,
mv);
            } catch (Exception var20) {
                dispatchException = var20;
            } catch (Throwable var21) {
                dispatchException = new NestedServletException("Handler
dispatch failed", var21);
            }
```

```
mappedHandler, mv, (Exception)dispatchException);
52          } catch (Exception var22) {
53              this.triggerAfterCompletion(processedRequest, response,
mappedHandler, var22);
54          } catch (Throwable var23) {
55              this.triggerAfterCompletion(processedRequest, response,
mappedHandler, new NestedServletException("Handler processing failed",
var23));
56          }
57
58      } finally {
59          if (asyncManager.isConcurrentHandlingStarted()) {
60              if (mappedHandler != null) {
61
 mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest,
response);
62              }
63          } else if (multipartRequestParsed) {
64              this.cleanupMultipart(processedRequest);
65          }
66
67      }
68  }
```

3.4.5 追踪源码: 处理器映射器

```
1   @Nullable
2   protected HandlerExecutionChain getHandler(HttpServletRequest request)
    throws Exception {
3       if (this.handlerMappings != null) {
4           // handlerMappings: 处理器映射器集合
5           Iterator var2 = this.handlerMappings.iterator();
6
7           while(var2.hasNext()) {
8               HandlerMapping mapping = (HandlerMapping)var2.next();
9               HandlerExecutionChain handler = mapping.getHandler(request);
10              if (handler != null) {
11                  return handler;
12              }
13          }
14      }
15
16      return null;
17  }
```

3.4.6 追踪源码: 处理器适配器

```
1   protected HandlerAdapter getHandlerAdapter(Object handler) throws
    ServletException {
2       if (this.handlerAdapters != null) {
3           // handlerAdapters: 处理器适配器集合
4           Iterator var2 = this.handlerAdapters.iterator();
5
6           while(var2.hasNext()) {
```

```
            }
        }
    }

    throw new ServletException("No adapter for handler [" + handler + "]:
The DispatcherServlet configuration needs to include a HandlerAdapter that
supports this handler");
}
```

3.4.7 追踪源码: 视图解析器

```
@Nullable
protected View resolveViewName(String viewName, @Nullable Map<String,
Object> model, Locale locale, HttpServletRequest request) throws Exception
{
    if (this.viewResolvers != null) {
        Iterator var5 = this.viewResolvers.iterator();

        while(var5.hasNext()) {
            ViewResolver viewResolver = (ViewResolver)var5.next();
            View view = viewResolver.resolveViewName(viewName, locale);
            if (view != null) {
                return view;
            }
        }
    }

    return null;
}
```

3.4.8 追踪源码: DispatcherServlet

```
protected void render(ModelAndView mv, HttpServletRequest request,
HttpServletResponse response) throws Exception {
    Locale locale = this.localeResolver != null ?
this.localeResolver.resolveLocale(request) : request.getLocale();
    response.setLocale(locale);
    String viewName = mv.getViewName();
    View view;
    if (viewName != null) {
        // 根据视图解析器创建视图对象
        view = this.resolveViewName(viewName, mv.getModelInternal(),
locale, request);
        if (view == null) {
            throw new ServletException("Could not resolve view with name '"
+ mv.getViewName() + "' in servlet with name '" + this.getServletName() +
"'");
        }
    } else {
        view = mv.getView();
        if (view == null) {
            throw new ServletException("ModelAndView [" + mv + "] neither
contains a view name nor a View object in servlet with name '" +
this.getServletName() + "'");
```

```
    if (this.logger.isTraceEnabled()) {
        this.logger.trace("Rendering view [" + view + "] ");
    }

    try {
        if (mv.getStatus() != null) {
            response.setStatus(mv.getStatus().value());
        }
        // 视图渲染: AbstractView
        view.render(mv.getModelInternal(), request, response);
    } catch (Exception var8) {
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Error rendering view [" + view + "]", var8);
        }

        throw var8;
    }
}
```

3.4.9 追踪源码: AbstractView

```
public void render(@Nullable Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("View " + this.formatViewName() + ", model " + (model != null ? model : Collections.emptyMap()) + (this.staticAttributes.isEmpty() ? "" : ", static attributes " + this.staticAttributes));
    }

    Map<String, Object> mergedModel = this.createMergedOutputModel(model, request, response);
    this.prepareResponse(request, response);
    // 合并视图与数据模型: InternalResourceView
    this.renderMergedOutputModel(mergedModel, this.getRequestToExpose(request), response);
}
```

3.4.10 追踪源码: InternalResourceView

```
protected void renderMergedOutputModel(Map<String, Object> model, HttpServletRequest request, HttpServletResponse response) throws Exception {
    this.exposeModelAsRequestAttributes(model, request);
    this.exposeHelpers(request);
    String dispatcherPath = this.prepareForRendering(request, response);
    RequestDispatcher rd = this.getRequestDispatcher(request, dispatcherPath);
    if (rd == null) {
        throw new ServletException("Could not get RequestDispatcher for [" + this.getUrl() + "]: Check that the corresponding file exists within your web application archive!");
    } else {
        if (this.useInclude(request, response)) {
```

```
            }

            rd.include(request, response);
        } else {
            if (this.logger.isDebugEnabled()) {
                this.logger.debug("Forwarding to [" + this.getUrl() + "]");
            }
            // 转发页面
            rd.forward(request, response);
        }

    }
}
```

3.4.11 追踪源码: RedirectView

```
protected void renderMergedOutputModel(Map<String, Object> model,
HttpServletRequest request, HttpServletResponse response) throws IOException
{
    String targetUrl = this.createTargetUrl(model, request);
    targetUrl = this.updateTargetUrl(targetUrl, model, request, response);
    RequestContextUtils.saveOutputFlashMap(targetUrl, request, response);
    // 重定向地址
    this.sendRedirect(request, response, targetUrl, this.http10Compatible);
}
```

```
protected void sendRedirect(HttpServletRequest request, HttpServletResponse
response, String targetUrl, boolean http10Compatible) throws IOException {
    String encodedURL = this.isRemoteHost(targetUrl) ? targetUrl :
response.encodeRedirectURL(targetUrl);
    HttpStatus attributeStatusCode;
    if (http10Compatible) {
        attributeStatusCode =
(HttpStatus)request.getAttribute(View.RESPONSE_STATUS_ATTRIBUTE);
        if (this.statusCode != null) {
            response.setStatus(this.statusCode.value());
            response.setHeader("Location", encodedURL);
        } else if (attributeStatusCode != null) {
            response.setStatus(attributeStatusCode.value());
            response.setHeader("Location", encodedURL);
        } else {
            // 重定向地址
            response.sendRedirect(encodedURL);
        }
    } else {
        attributeStatusCode = this.getHttp11StatusCode(request, response,
targetUrl);
        response.setStatus(attributeStatusCode.value());
        response.setHeader("Location", encodedURL);
    }

}
```

3.4.12 源码说明

3. HandlerMapping根据请求URI找到具体的处理器， 生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
5. 执行处理器(Controller， 也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
11. DispatcherServlet响应用户 (实际上是视图做的跳转(转发,重定向等..))

四、SpringMVC三大组件

- HandlerMapping：处理器映射器 （三大组件之一）
- HandlAdapter：处理器适配器（三大组件之一）
- ViewResolver：视图解析器（三大组件之一）

4.1 DispatcherServlet：前端控制器

> 用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

4.2 HandlerMapping：处理器映射器

> HandlerMapping负责根据用户请求url找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

4.3 Handler：处理器 (后端控制器)

> Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。
>
> 由于Handler涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发Handler。

4.4 HandlAdapter：处理器适配器

> 通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

下图是许多不同的适配器，最终都可以使用usb接口连接

4.5 ViewResolver：视图解析器

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

4.6 View：视图

> springmvc框架提供了很多的View视图类型的支持，包括：jstlView、freemarkerView、pdfView等。我们最常用的视图就是jsp。
>
> 一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由开发者根据业务需求开发具体的页面。

五、@RequestMapping

- 源码示例

```
@Target({ElementType.METHOD, ElementType.TYPE}) // 修饰方法,类
@Retention(RetentionPolicy.RUNTIME) // 保存到运行中
@Documented // 记录注释成文档
@Mapping // 标识为映射注解
public @interface RequestMapping {
}
```

5.1 指定请求路径

```
package com.itheima.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class RequestMappingController {



    /**
     * @RequestMapping: 修饰方法
     *  value: 映射访问路径后缀: /hello.do
     *      "/" 可缺省
     *      ".do" 可缺省
     *  method: 限制请求的方法只能是{RequestMethod.GET}
```

```
    @RequestMapping(value = "hello"
            , method = {RequestMethod.GET}
            , params = {"id=10","money!=0"}
            , headers = {"cookie"}
    )
    public String hello(){
        return "success";
    }
}
```

5.2 指定请求父路径

```
package com.itheima.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * @RequestMapping: 修饰类
 *  value: 映射访问的根路径 /account
 *      "/" 可缺省
 *  作用: 模块化区分
 *      account/add
 *      account/update
 *      account/del
 *      account/get
 */
@Controller
@RequestMapping("account")
public class RequestMappingController {


    /**
     * @RequestMapping: 修饰方法
     *  value: 映射访问路径后缀: /hello.do
     *      "/" 可缺省
     *      ".do" 可缺省
     *  method: 限制请求的方法只能是{RequestMethod.GET}
     *  params: 限制请求的参数需要有{"id=10","money!=0"}
     *  headers: 限制请求的头部需要有{"cookie"}
     */
    @RequestMapping(value = "hello"
            , method = {RequestMethod.GET}
            , params = {"id=10","money!=0"}
            , headers = {"cookie"}
    )
    public String hello(){
        return "success";
    }
}
```

5.3 指定请求方法

- pages/commit.jsp

```jsp
<%--
  Created by IntelliJ IDEA.
  User: Jason
  Date: 2019/7/18
  Time: 15:55
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>COMMIT</title>
</head>
<body>
    <h4>POST方法</h4>
    <%-- 限定GET方法后,其他方法不能访问 --%>
    <form action="/account/hello.do" method="post">
        <input type="submit"/>
    </form>
</body>
</html>

```

六、SpringMVC参数绑定

6.1 案例演示

- com.itheima.web.ParamController

```java
package com.itheima.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * 参数绑定案例代码
 */
@Controller
@RequestMapping("param")
public class ParamController {
}

```

6.2 默认支持参数类型

6.2.1 HttpServletRequest

```java
/**
 * 默认支持参数
 * @param req 请求对象：操作request域
 * @return
 */
@RequestMapping("hello")
public String hello(HttpServletRequest req) {
    String id = req.getParameter("id");
    req.setAttribute("name", id);
    return "success";
}
```

6.2.2 HttpServletResponse

作用：通过response，执行响应。

```java
/**
 * 默认支持参数
 * @param res 响应对象：执行响应
 * @return
 */
@RequestMapping("hello")
public void hello(HttpServletResponse res) throws IOException {
    res.sendRedirect("/pages/success.jsp");
}
```

6.2.3 HttpSession

作用：通过session，获取和保存会话域数据。

```java
/**
 * 默认支持参数
 * @param session 会话对象：操作session域
 * @return
 */
@RequestMapping("hello")
public String hello(HttpSession session) {
    session.setAttribute("name", "session OK");
    return "success";
}
```

6.2.4 ServletApi

- Locale: zh_CN (本地化对象)
- TimeZone: 时区时间对象
- InputStream: 输入流
- OutputStream: 输出流
- Reader: 字符输入流
- Writer: 字符输出流
- java.security.Principal (jdk安全认证的登录信息)

6.2.5 Model/ModelMap

- Model是一个接口，是模型，用于封装响应的模型数据
- ModelMap是实现类，使用Model和使用ModelMap，效果是一样的
- 使用Model封装响应的模型数据，就可以不使用ModelAndView，页面视图可以使用字符串String响应。

```
/**
 * 默认支持参数
 * @param model 数据模型: 保存获取数据
 * @return
 */
@RequestMapping("hello")
public String hello(Model model) {
    model.addAttribute("name", "model OK");
    return "success";
}
```

6.3 简单参数类型

```
/**
 * 基本数据类型: 参数不能为空(null)
 * 包装数据类型: 参数可以是空
 *
 * @RequestParam: 绑定指定名称的简单类型参数, 名称区分大小写
 *  value: 属性名称
 *  required: 是否必传参数 true默认值表示必须携带
 *  defaultValue: 默认值 当未传递此参数将采用默认值 (前提是包装类型)
 */
@RequestMapping("hello")
public String hello(@RequestParam("id") Integer id
            , @RequestParam(defaultValue = "Jason") String name
            , @RequestParam(required = false) Integer sex
) {
        System.out.println(id);
        System.out.println(name);
        System.out.println(sex);
        // 返回的页面路径
        return "success";
}
```

6.3.1 参数乱码问题

- pages/commit.jsp

```
<%--
  Created by IntelliJ IDEA.
  User: Jason
  Date: 2019/7/18
  Time: 15:55
  To change this template use File | Settings | File Templates.
--%>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
```

```
</head>
<body>
    <h4>POST方法</h4>
    <%-- 限定GET方法后,其他方法不能访问 --%>
    <form action="/account/hello.do" method="post">
        <input type="submit"/>
    </form>
    <h4>请求中文乱码</h4>
    <%-- get方法tomcat8以上没有乱码 --%>
    <%-- post方法又乱码问题 --%>
    <form action="/param/hello.do" method="post">
        <input name="id" value="1"/>
        <input name="name" value="中文"/>
        <input type="submit"/>
    </form>
</body>
</html>
```

6.3.2 解决乱码问题

- web.xml

```
<!-- 1. 解决参数乱码的过滤器 -->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <!-- 指定字符集编码 -->
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<!-- 2. 过滤需要解决参数乱码的资源-->
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

6.4 对象参数类型

```
     *
     * @param account 对象参数
     * @return
     */
    @RequestMapping("hello")
    public String hello(Account account){
        System.out.println(account);
        // 返回的页面路径
        return "success";
    }
```

- com.itheima.web.Account

```
package com.itheima.web;

/**
 * Account.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class Account {
    private Integer id;
    private Integer uid;
    private Double money;

    public Integer getId() {
        return id;
    }

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

    public Integer getUid() {
        return uid;
    }

    public void setUid(Integer uid) {
        this.uid = uid;
    }

    public Double getMoney() {
        return money;
    }

    public void setMoney(Double money) {
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
```

```
                '}';
    }
}
```

6.5 嵌套对象类型

```
/**
 * 对象类型
 *  赋值对象属性值示例: hello.do?id=1
 *  赋值嵌套对象属性值示例: hello.do?user.id=1
 *
 * @param account 对象参数
 * @return
 */
@RequestMapping("hello")
public String hello(Account account){
    System.out.println(account);
    // 返回的页面路径
    return "success";
}
```

- com.itheima.web.Account

```
package com.itheima.web;

/**
 * Account.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class Account {
    private Integer id;
    private Integer uid;
    private Double money;

    private User user;

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public Integer getId() {
        return id;
    }

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

```

```
    }

    public void setUid(Integer uid) {
        this.uid = uid;
    }

    public Double getMoney() {
        return money;
    }

    public void setMoney(Double money) {
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", uid=" + uid +
                ", money=" + money +
                ", user=" + user +
                '}';
    }
}
```

- com.itheima.web.User

```
package com.itheima.web;

import java.util.Date;

/**
 * User.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class User {

    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    public Integer getId() {
        return id;
    }

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

    public String getUsername() {
```

```
    public void setUsername(String username) {
        this.username = username;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", birthday=" + birthday +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```