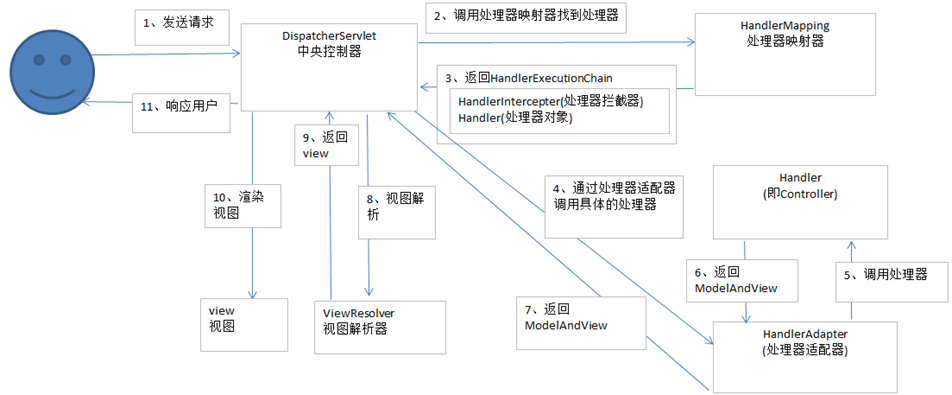
# SpringMVC原理

### SpringMVC的运行流程



⑴ 用户发送请求至前端控制器DispatcherServlet

⑵ DispatcherServlet收到请求调用HandlerMapping处理器映射器。

⑶ 处理器映射器根据请求url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。

1. DispatcherServlet通过HandlerAdapter处 理器适配器调用处理器

⑸ 执行处理器(Controller，也叫后端控制器)。

⑹ Controller执行完成返回ModelAndView

⑺ HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet

⑻ DispatcherServlet将ModelAndView传给ViewReslover视图解析器

⑼ ViewReslover解析后返回具体View

⑽ DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。

⑾ DispatcherServlet响应用户。

### 回顾Servet知识

#### 什么是Servlet

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。

使用 Servlet，您可以收集来自网页表单的用户输入，呈现来自数据库或者其他源的记录，还可以动态创建网页。

Java Servlet 通常情况下与使用 CGI（Common Gateway Interface，公共网关接口）实现的程序可以达到异曲同工的效果。但是相比于 CGI，Servlet 有以下几点优势：

性能明显更好。

Servlet 在 Web 服务器的地址空间内执行。这样它就没有必要再创建一个单独的进程来处理每个客户端请求。

Servlet 是独立于平台的，因为它们是用 Java 编写的。

服务器上的 Java 安全管理器执行了一系列限制，以保护服务器计算机上的资源。因此，Servlet 是可信的。

Java 类库的全部功能对 Servlet 来说都是可用的。它可以通过 sockets 和 RMI 机制与 applets、数据库或其他软件进行交互。

#### 什么是Servlet生命周期

### Servlet 生命周期：

Servlet 加载—>实例化—>服务—>销毁。

**init（）：**

在Servlet的生命周期中，仅执行一次init()方法。它是在服务器装入Servlet时执行的，负责初始化Servlet对象。可以配置服务器，以在启动服务器或客户机首次访问Servlet时装入Servlet。无论有多少客户机访问Servlet，都不会重复执行init（）。

**service（）：**

它是Servlet的核心，负责响应客户的请求。每当一个客户请求一个HttpServlet对象，该对象的Service()方法就要调用，而且传递给这个方法一个“请求”（ServletRequest）对象和一个“响应”（ServletResponse）对象作为参数。在HttpServlet中已存在Service()方法。默认的服务功能是调用与HTTP请求的方法相应的do功能。

**destroy（）：**

仅执行一次，在服务器端停止且卸载Servlet时执行该方法。当Servlet对象退出生命周期时，负责释放占用的资源。一个Servlet在运行service()方法时可能会产生其他的线程，因此需要确认在调用destroy()方法时，这些线程已经终止或完成。

### 手写SpringMVC思路

#### 1.web.xml加载

 为了读取web.xml中的配置，我们用到ServletConfig这个类，它代表当前Servlet在web.xml中的配置信息。通过web.xml中加载我们自己写的MyDispatcherServlet和读取配置文件。

#### 2、初始化阶段

  在前面我们提到DispatcherServlet的initStrategies方法会初始化9大组件，但是这里将实现一些SpringMVC的最基本的组件而不是全部，按顺序包括：

* 加载配置文件
* 扫描用户配置包下面所有的类
* 拿到扫描到的类，通过反射机制，实例化。并且放到ioc容器中(Map的键值对 beanName-bean) beanName默认是首字母小写
* 初始化HandlerMapping，这里其实就是把url和method对应起来放在一个k-v的Map中,在运行阶段取出

#### 3、运行阶段

  每一次请求将会调用doGet或doPost方法，所以统一运行阶段都放在doDispatch方法里处理，它会根据url请求去HandlerMapping中匹配到对应的Method，然后利用反射机制调用Controller中的url对应的方法，并得到结果返回。按顺序包括以下功能：

* 异常的拦截
* 获取请求传入的参数并处理参数
* 通过初始化好的handlerMapping中拿出url对应的方法名，反射调用

### 手写SpringMVC基本实现

|  |
| --- |
| /\*\*  \* 手写SpringMVC框架 作者:每特教育-余胜军<br>  \*  \* **@QQ644064779** 1.自定义DispatcherServlet<br>  \* 2.servlet init()方法初始化###只会执行一次<br>  \* ######2.1获取当前包下所有的类<br>  \* ######2.2初始化当前包下所有的类,使用Java反射机制初始化对象存放在SpringMVC容器中key(beanId)-  \* value( 当前实例对象) <br>  \* ######2.3初始化HandlerMapping方法,将url和方法对应上 <br>  \* ########2.3.1使用Java反射技术读取类的信息,存放在map集合中key为url请求地址,value为对应方法  \* <br>  \* ########2.3.2使用Java反射技术读取类的信息,存放在map集合中key为url请求地址,value为对应实例对象  \* <br>  \* 3.servlet get或者post请求<br>  \* ######## 3.1.1获取请求地址,使用Java反射技术找到对应的方法和实例对象进行执行 <br>  \*/  **public** **class** ExtDispatcherServlet **extends** HttpServlet {  // mvc bean key=beanid ,value=对象  **private** ConcurrentHashMap<String, Object> mvcBeans = **new** ConcurrentHashMap<String, Object>();  // mvc 请求方法 key=requestUrl,value=对象  **private** ConcurrentHashMap<String, Object> mvcBeanUrl = **new** ConcurrentHashMap<String, Object>();  // mvc 请求方法 key=requestUrl,value=方法  **private** ConcurrentHashMap<String, String> mvcMethodUrl = **new** ConcurrentHashMap<String, String>();  /\*\*  \* 初始化自定义SpringMVC容器  \*/  **public** **void** init() **throws** ServletException {  **try** {  // 1.获取当前包下所有的类  List<Class<?>> classes = ClassUtil.*getClasses*("com.itmayiedu.ext.controller");  // 2.初始化当前包下所有的类,使用Java反射机制初始化对象存放在SpringMVC容器中key(beanId)-value(  // 当前实例对象)  findClassMVCBeans(classes);  // 3.初始化HandlerMapping方法,将url和方法对应上  handlerMapping(mvcBeans);  } **catch** (Exception e) {  }  }  // 2.初始化当前包下所有的类,使用Java反射机制初始化对象存放在SpringMVC容器中key(beanId)-value(  // 当前实例对象)  **public** **void** findClassMVCBeans(List<Class<?>> classes)  **throws** ClassNotFoundException, InstantiationException, IllegalAccessException {  mvcBeans = **new** ConcurrentHashMap<String, Object>();  **for** (Class<?> classInfo : classes) {  ExtController extController = classInfo.getDeclaredAnnotation(ExtController.**class**);  **if** (extController != **null**) {  // 默认类名小写 作为bean的名称  String beanId = ClassUtil.*toLowerCaseFirstOne*(classInfo.getSimpleName());  mvcBeans.put(beanId, ClassUtil.*newInstance*(classInfo));  }  }  }  // 3.初始化HandlerMapping方法,将url和方法对应上  **public** **void** handlerMapping(ConcurrentHashMap<String, Object> mvcBeans) {  // 遍历mvc bean对象  **for** (Map.Entry<String, Object> entry : mvcBeans.entrySet()) {  // springmvc 注入object对象  Object mvcObject = entry.getValue();  // 判断类上是否有@ExtRequestMapping注解  Class<? **extends** Object> classInfo = mvcObject.getClass();  String requestBaseUrl = **null**;  ExtRequestMapping classExtRequestMapping = classInfo.getAnnotation(ExtRequestMapping.**class**);  **if** (classExtRequestMapping != **null**) {  requestBaseUrl = classExtRequestMapping.value();  }  // 遍历当前类的所有方法,判断方法上是否有注解  Method[] declaredMethods = classInfo.getDeclaredMethods();  **for** (Method method : declaredMethods) {  ExtRequestMapping methodExtRequestMapping = method.getDeclaredAnnotation(ExtRequestMapping.**class**);  **if** (methodExtRequestMapping != **null**) {  String httpRequestUrl = methodExtRequestMapping.value();  mvcBeanUrl.put(requestBaseUrl + httpRequestUrl, mvcObject);  mvcMethodUrl.put(requestBaseUrl + httpRequestUrl, method.getName());  }  }  }  }  @Override  **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  doPost(req, resp);  }  @Override  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **try** {  doDispatch(req, resp);  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  **public** **void** doDispatch(HttpServletRequest req, HttpServletResponse resp) **throws** IOException, ServletException {  // 1.获取请求url地址  String requestUrl = req.getRequestURI();  // 2.使用请求url查找对应mvc 控制器bean  Object object = mvcBeanUrl.get(requestUrl);  **if** (object == **null**) {  resp.getWriter().println("http ext not found controller 404");  **return**;  }  // 3.获取对应的请求方法  String methodName = mvcMethodUrl.get(requestUrl);  **if** (StringUtils.*isEmpty*(methodName)) {  resp.getWriter().println("http ext not found Method 404");  **return**;  }  // 4.使用java反射技术执行方法  Class<? **extends** Object> classInfo = object.getClass();  String resultPage = (String) methodInvoke(classInfo, object, methodName);  // 5.视图展示  viewdisplay(resultPage, req, resp);  }  // 执行方法··  **public** Object methodInvoke(Class<? **extends** Object> classInfo, Object object, String methodName) {  **try** {  Method method = classInfo.getMethod(methodName);  Object result = method.invoke(object);  **return** result;  } **catch** (Exception e) {  e.printStackTrace();  **return** **null**;  }  }  // 视图展示  **public** **void** viewdisplay(String pageName, HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  // 获取后缀信息  String suffix = ".jsp";  // 页面目录地址  String prefix = "/";  req.getRequestDispatcher(prefix + pageName + suffix).forward(req, res);  }  } |