**SpringMVC介绍**

**为什么要使用springMVC？**

很多应用程序的问题在于处理业务数据和显示业务数据的视图的对象之间存在紧密耦合。通常，更新业务对象的命令都是从视图本身发起的，使视图对任何业务对象更改都有高度敏感性。而且，当多个视图依赖于同一个业务对象时是没有灵活性的。

Spring Web MVC是一种基于Java的实现了Web MVC设计模式的请求驱动类型的轻量级Web框架，即使用了MVC架构模式的思想，将web层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC也是要简化我们日常Web开发的。

**MVC设计模型**

MVC 是一种著名的设计模式，特别是在 Web 应用程序领域。模式全都是关于将包含业务数据的模块与显示模块的视图解耦的。这是怎样发生的？视图（例如，JSP 页面）怎样能够与其模型（例如，包含数据的 JavaBean）解耦？记得这句格言么？一个层次的重定向几乎可以解决计算机业中的所有问题。确实，在模型和视图之间引入重定向层可以解决问题。此重定向层是控制器。控制器将接收请求，执行更新模型的操作，然后通知视图关于模型更改的消息。依赖于模型的状态并且依赖于请求的控制器可以决定要显示哪个视图。

**springMVC的强大之处**

1.Spring MVC 实现了即用的 MVC 的核心概念。它为控制器和处理程序提供了大量与此模式相关的功能。并且当向 MVC 添加反转控制（Inversion of Control，IoC）时，它使应用程序高度解耦，提供了通过简单的配置更改即可动态更改组件的灵活性。Spring MVC 为您提供了完全控制应用程序的各个方面的力量。

2.Spring 的 Web MVC 模块是围绕 DispatcherServlet 而设计的。DispatcherServlet 给处理程序分派请求，执行视图解析，并且处理语言环境和主题解析，此外还为上传文件提供支持。

3.DispatcherServlet 通过使用处理程序映射来决定哪一个处理程序应当处理传入的请求。处理程序映射只是用于标识使用哪一个处理程序来处理特定 URL 模式的映射。处理程序是只有一种方法 ModelAndView handleRequest(request,response) 的控制器接口的实现。Spring 还有一些可用的高级处理程序实现；其中一个重要的高级处理程序实现是 SimpleFormController，它提供了将命令对象绑定到表单、对其执行验证等功能。

4.您已经在本系列教程的先前教程中使用了 DispatcherServlet 和简单的处理程序。在下一个部分中，将使用 SimpleFormController 并说明 Spring MVC 提供的各种即用功能。

**springMVC优势**

1、清晰的角色划分：前端控制器（DispatcherServlet）、请求到处理器映射（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 提供给表单展示和提交到的对象就叫表单对象）。

2、分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要；

3、由于命令对象就是一个POJO，无需继承框架特定API，可以使用命令对象直接作为业务对象；

4、和Spring 其他框架无缝集成，是其它Web框架所不具备的；

5、可适配，通过HandlerAdapter可以支持任意的类作为处理器；

6、可定制性，HandlerMapping、ViewResolver等能够非常简单的定制；

7、功能强大的数据验证、格式化、绑定机制；

8、利用Spring提供的Mock对象能够非常简单的进行Web层单元测试；

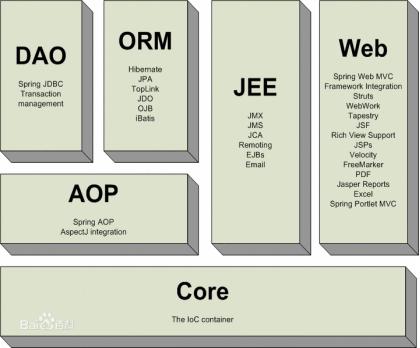
9、本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。

10、强大的JSP标签库，使JSP编写更容易。

………………还有比如RESTful风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配置支持等等。

**springMVC的运行原理**

**架构图**

****

**首先让我们了解下 MVC（Model-View-Controller）三元组的概念：**

Model（模型）：数据模型，提供要展示的数据，因此包含数据和行为，可以认为是领域模型或 JavaBean 组件（包含数据和行为），不过现在一般都分离开来：Value Object（数据） 和 服务层（行为）。也就是模型提供了模型数据查询和模型数据的状态更新等功能，包括数据和业务。

领域模型

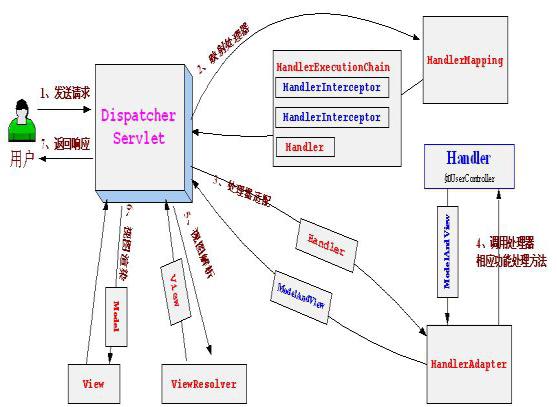
javaBean组件等价于 域模型层 + 业务逻辑层 + 持久层

View(视图):负责进行模型的展示，一般就是我们见到的用户界面，客户想看到的东西。

Controller(控制器):接收用户请求，委托给模型进行处理（状态改变），处理完毕后把返回的模型数据返回给视图，

由视图负责展示。 也就是说控制器做了个调度员的工作，。

**Springmvc运行原理流程**

****

核心架构的具体流程步骤如下：

1、  首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；

2、DispatcherServlet——>HandlerMapping,HandlerMapping将会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象，通过这种策略模式，很容易添加新的映射策略；

3、  DispatcherServlet——>HandlerAdapter，HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；

4、  HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个ModelAndView对象（包含模型数据、逻辑视图名）；

5、  ModelAndView的逻辑视图名——> ViewResolver， ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；

6、  View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数据结构，因此很容易支持其他视图技术；

7、返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

**组件说明**

以下组件通常使用框架提供实现：

u 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。

u HandlerMapping：处理器映射器

HandlerMapping负责根据用户请求找到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。

u Handler：处理器

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。

u 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。

u View Resolver：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。

u 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，需要由程序员根据业务需求开发具体的页面。

注解描述：

@RequestMapping：定义请求url到处理器功能方法的映射

**RequestMappingHandlerAdapter**

注解式处理器适配器，对标记@ResquestMapping的方法进行适配。

从spring3.1版本开始，废除了AnnotationMethodHandlerAdapter的使用，推荐使用RequestMappingHandlerAdapter完成注解式处理器适配。

配置如下：

<!--注解适配器 -->

<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"/>

**<mvc:annotation-driven>**

springmvc使用<mvc:annotation-driven>自动加载RequestMappingHandlerMapping和RequestMappingHandlerAdapter，可用在springmvc.xml配置文件中使用<mvc:annotation-driven>替代注解处理器和适配器的配置。

**1.1 视图解析器**

在springmvc.xml文件配置如下：

[复制代码](javascript:void(0);)

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass"

value="org.springframework.web.servlet.view.JstlView" />

<property name="prefix" value="/WEB-INF/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

[复制代码](javascript:void(0);)

InternalResourceViewResolver：支持JSP视图解析

viewClass：JstlView表示JSP模板页面需要使用JSTL标签库，所以classpath中必须包含jstl的相关jar 包。此属性可以不设置，默认为JstlView。

prefix 和suffix：查找视图页面的前缀和后缀，最终视图的址为：

前缀+**逻辑视图名**+后缀，逻辑视图名需要在controller中返回ModelAndView指定，比如逻辑视图名为hello，则最终返回的jsp视图地址 “WEB-INF/jsp/hello.jsp”

**DispatcherServlet核心代码分析**

[复制代码](javascript:void(0);)

//前端控制器分派方法

protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {

        HttpServletRequest processedRequest = request;

        HandlerExecutionChain mappedHandler = null;

        int interceptorIndex = -1;

        try {

            ModelAndView mv;

            boolean errorView = false;

            try {

        //检查是否是请求是否是multipart（如文件上传），如果是将通过MultipartResolver解析

                processedRequest = checkMultipart(request);

         //步骤2、请求到处理器（页面控制器）的映射，通过HandlerMapping进行映射

                mappedHandler = getHandler(processedRequest, false);

                if (mappedHandler == null || mappedHandler.getHandler() == null) {

                    noHandlerFound(processedRequest, response);

                    return;

                }

 //步骤3、处理器适配，即将我们的处理器包装成相应的适配器（从而支持多种类型的处理器）

                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

                // 304 Not Modified缓存支持

                //此处省略具体代码

                // 执行处理器相关的拦截器的预处理（HandlerInterceptor.preHandle）

                //此处省略具体代码

                // 步骤4、由适配器执行处理器（调用处理器相应功能处理方法）

                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

                // Do we need view name translation?

                if (mv != null && !mv.hasView()) {

                    mv.setViewName(getDefaultViewName(request));

                }

                // 执行处理器相关的拦截器的后处理（HandlerInterceptor.postHandle）

                //此处省略具体代码

            }

            catch (ModelAndViewDefiningException ex) {

                logger.debug("ModelAndViewDefiningException encountered", ex);

                mv = ex.getModelAndView();

            }

            catch (Exception ex) {

                Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);

                mv = processHandlerException(processedRequest, response, handler, ex);

                errorView = (mv != null);

            }

//步骤5 步骤6、解析视图并进行视图的渲染

//步骤5 由ViewResolver解析View（viewResolver.resolveViewName(viewName, locale)）

步骤6 视图在渲染时会把Model传入（view.render(mv.getModelInternal(), request, response);）

            if (mv != null && !mv.wasCleared()) {

                render(mv, processedRequest, response);

                if (errorView) {

                    WebUtils.clearErrorRequestAttributes(request);

                }

            }

            else {

                if (logger.isDebugEnabled()) {

                    logger.debug("Null ModelAndView returned to DispatcherServlet with name '" + getServletName() +

                            "': assuming HandlerAdapter completed request handling");

                }

            }

            // 执行处理器相关的拦截器的完成后处理（HandlerInterceptor.afterCompletion）

            //此处省略具体代码

        catch (Exception ex) {

            // Trigger after-completion for thrown exception.

            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex)            throw ex;

        }

        catch (Error err) {

            ServletException ex = new NestedServletException("Handler processing failed", err);

            // Trigger after-completion for thrown exception.

            triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex);

            throw ex;

        }

        finally {

            // Clean up any resources used by a multipart request.

            if (processedRequest != request) {

                cleanupMultipart(processedRequest);

            }

        }

    }