# SpringMVC官方文档要点记录

Version 5.0.3.RELEASE

## 1.1 Introduction

Spring Web MVC is the original web framework built on the Servlet API and included in the Spring Framework from the very beginning. The formal name "Spring Web MVC" comes from the name of its source module spring-webmvc

Spring Web MVC是在Servlet API上构建的原始Web框架，从一开始就包含在Spring框架中。正式名称“Spring Web MVC”来自其源模块spring-webmvc的名称

## 1.2 DispatcherServlet

Spring MVC, like many other web frameworks, is designed around the front controller pattern where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing while actual work is performed by configurable, delegate components. This model is flexible and supports diverse workflows.

像许多其他web框架一样，SpringMVC围绕着前端控制器模式设计，其中中央Servlet —DispatcherServlet为请求处理提供了共享算法，而实际工作由可配置代理组件执行。这个模型是灵活的，支持多种工作流

The DispatcherServlet, as any Servlet, needs to be declared and mapped according to the Servlet specification using Java configuration or in web.xml. In turn the DispatcherServlet uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling, and more.

DispatcherServlet和任何Servlet一样，都需要根据Servlet规范使用Java配置或在web.xml中进行声明和映射，反过来，DispatcherServlet使用Spring配置来发现请求映射、视图解析、异常处理等所需的代理组件

Spring Boot follows a different initialization sequence. Rather than hooking into the lifecycle of the Servlet container, Spring Boot uses Spring configuration to bootstrap itself and the embedded Servlet container. Filter and Servlet declarations are detected in Spring configuration and registered with the Servlet container

Spring Boot遵循了不同的初始化顺序，Spring Boot使用Spring配置去引导自身和嵌入的Servlet容器，而不是连接到Servlet容器的生命周期。在Spring配置中检测到Filter和Servlet的声明并且注册到Servlet容器中

Below is an example of the Java configuration that registers and initializes the DispatcherServlet. This class is auto-detected by the Servlet container

下面是注册和初始化DispatcherServlet的Java配置示例。此类由Servlet容器自动检测

**public** **class** **MyWebApplicationInitializer** **implements** WebApplicationInitializer {

@Override

**public** **void** onStartup(ServletContext servletCxt) {

*// Load Spring web application configuration*

AnnotationConfigWebApplicationContext ac = **new** AnnotationConfigWebApplicationContext();

ac.register( .class);

ac.refresh();

*// Create and register the DispatcherServlet*

DispatcherServlet servlet = **new** DispatcherServlet(ac);

ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);

registration.setLoadOnStartup(1);

registration.addMapping("/app/\*");

}

}

Below is an example of web.xml configuration to register and initialize the DispatcherServlet:

下面是一个web.xml配置示例。用于注册和初始化DispatcherServlet

<web-app>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

### 1.2.1. Context Hierarchy

DispatcherServlet expects a WebApplicationContext, an extension of a plain ApplicationContext, for its own configuration. WebApplicationContext has a link to the ServletContext and Servlet it is associated with. It is also bound to the ServletContext such that applications can use static methods on RequestContextUtils to look up the WebApplicationContext if they need access to it.

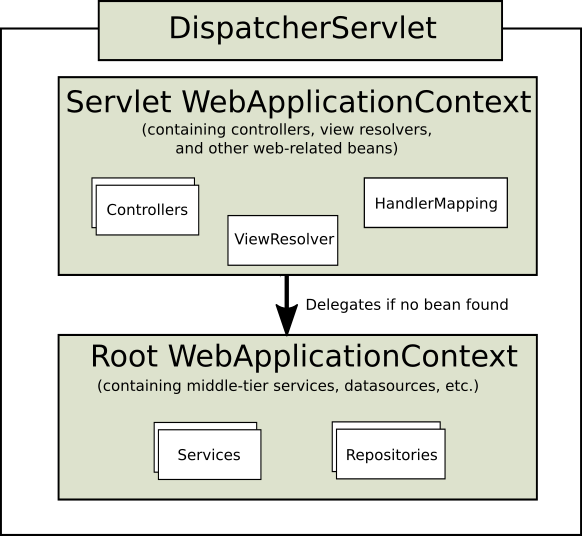
DispatcherServlet需要一个WebApplicationContext（一个普通的ApplicationContext的扩展）来进行它自身的配置，WebApplicationContext具有与ServletContext和Servlet相关联的链接，它也绑定到ServletContext，使得应用程序可以使用RequestContextUtils上的静态方法来查找WebApplicationContext（如果应用程序需要访问它）

For many applications having a single WebApplicationContext is simple and sufficient. It is also possible to have a context hierarchy where one root WebApplicationContext is shared across multiple DispatcherServlet (or other Servlet) instances, each with its own child WebApplicationContext configuration.

对于许多拥有单个WebApplicationContext的应用程序来说，它是简单而且足够的，也有可能有一个上下文层级结构，其中一个根WebApplicationContext被多个DispatcherServlet（或其它Servlet）实例共享

The root WebApplicationContext typically contains infrastructure beans such as data repositories and business services that need to be shared across multiple Servlet instances. Those beans are effectively inherited and could be overridden (i.e. re-declared) in the Servlet-specific, child WebApplicationContext which typically contains beans local to the given Servlet

根WebApplicationContext通常包含需要被多个Servlet实例共享的基础结构bean，如数据存储和业务服务，这些bean被有效的继承，并在特定的Servlet上重载（即重声明），子WebApplicationContext通常包含给定Servlet的本地beans



Below is example configuration with a WebApplicationContext hierarchy:

下面是使用WebApplicationContext层次结构的示例配置：

**public** **class** **MyWebAppInitializer** **extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** Class<?>**[]** getRootConfigClasses() {

**return** **new** Class<?**[]** { RootConfig.class };

}

@Override

**protected** Class<?>**[]** getServletConfigClasses() {

**return** **new** Class<?**[]** { App1Config.class };

}

@Override

**protected** String**[]** getServletMappings() {

**return** **new** String**[]** { "/app1/\*" };

}

}

And the web.xml equivalent:

和web.xml等效

<web-app>

<listener> <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/root-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app1</servlet-name> <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app1-context.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app1</servlet-name>

<url-pattern>/app1/\*</url-pattern>

</servlet-mapping>

</web-app>

### 1.2.2. Special Bean Types

The DispatcherServlet delegates to special beans to process requests and render the appropriate responses. By "special beans" we mean Spring-managed Object instances that implement one of the framework contracts listed in the table below. Spring MVC provides built-in implementations of these contracts but you can also customize, extend, or replace them.

DispatcherServlet委托特殊的beans处理请求并呈现相应的响应，“特殊的beans”是指实现下表中列出的框架协议之一的Spring管理对象实例的实现

#### [HandlerMapping](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-handlermapping)

Map a request to a handler along with a list of [interceptors](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-handlermapping-interceptor) for pre- and post- processing. The mapping is based on some criteria the details of which vary by HandlerMapping implementation.

将一个请求映射到处理程序，以及一个用于前置处理和后置处理的拦截器列表，该映射基于一些标准，其细节因HandlerMapping的实现而异

The two main HandlerMapping implementations are RequestMappingHandlerMapping which supports @RequestMapping annotated methods and SimpleUrlHandlerMapping which maintains explicit registrations of URI path patterns to handlers.

HandlerMapping的两个主要实现是RequestMappingHandlerMapping（其支持@RequestMapping注解方法）和SimpleUrlHandlerMapping（其维护Url路径模式到处理程序的显示注册）

#### HandlerAdapter

Help the DispatcherServlet to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example, invoking an annotated controller requires resolving annotations. The main purpose of a HandlerAdapteris to shield the DispatcherServlet from such details.

帮助DispatcherServlet 去调用映射到请求的处理程序，而不管处理程序是实际如何调用的，例如，调用一个注解控制器需要解析注解，HandlerAdapteris的主要目的是保护DispatcherServlet不受这些细节的影响

#### [HandlerExceptionResolver](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-exceptionhandlers)

Strategy to resolve exceptions possibly mapping them to handlers, or to HTML error views, or other.

解决异常的策略，可能映射它们到处理程序、HTML错误视图或其他

#### [ViewResolver](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-viewresolver)

Resolve logical String-based view names returned from a handler to an actual View to render to the response with.

将从处理程序返回的基于字符串的逻辑视图名称解析为实际视图以呈现响应

#### [LocaleResolver](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-localeresolver), [LocaleContextResolver](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-timezone)

Resolve the Locale a client is using and possibly their time zone, in order to be able to offer internationalized views.

解决客户端正在使用的区域化设置以及可能的时区，以便能够提供国际化视图

#### [ThemeResolver](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-themeresolver)

Resolve themes your web application can use, for example, to offer personalized layouts.

解决您的web应用程序可能使用的主题，例如，提供个性化的布局

#### [MultipartResolver](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-multipart)

Abstraction for parsing a multi-part request (e.g. browser form file upload) with the help of some multipart parsing library.

在一些multipart 解析库的帮助下解析一些multipart 请求的抽象（例如，浏览器表单文件上传）

#### [FlashMapManager](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html" \l "mvc-flash-attributes)

Store and retrieve the "input" and the "output" FlashMap that can be used to pass attributes from one request to another, usually across a redirect.

存储和检索能够用于将属性从一个请求传递到另一个请求的“输入”和“输出”FlashMap，通常通过重定向

### **1.2.3. Framework Config**

For each type of special bean, the DispatcherServlet checks for the WebApplicationContext first. If there are no matching bean types, it falls back on the default types listed in DispatcherServlet.properties.

对每种类型的特殊bean，DispatcherServlet 首先检查WebApplicationContext，如果没有匹配的bean类型，它将回退到DispatcherServlet.properties中列出的默认类型

Applications can declare the special beans they wish to have. Most applications however will find a better starting point in the MVC Java config or the MVC XML namespace which provide a higher level configuration API that in turn make the necessary bean declarations.

应用程序能够声明它们希望拥有的特殊bean，然而大部分程序将会在MVC Java Config或MVC XML命名空间中找到一个更好的起点，它提供了更高级的配置API，从而进行必要的bean声明

### 1.2.4. Container Config

In a Servlet 3.0+ environment, you have the option of configuring the Servlet container programmatically as an alternative or in combination with a web.xml file. Below is an example of registering a DispatcherServlet:

在Servlet 3.0+环境中，你可以选择以编程的方式配置Servlet容器，作为代替方法或与web.xml文件结合使用. 下面是注册DispatcherServlet的示例:

**import** org.springframework.web.WebApplicationInitializer;

**public** **class** **MyWebApplicationInitializer** **implements** WebApplicationInitializer {

@Override

**public** **void** onStartup(ServletContext container) {

XmlWebApplicationContext appContext = **new** XmlWebApplicationContext();

appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

ServletRegistration.Dynamic registration = container.addServlet("dispatcher", **new** DispatcherServlet(appContext));

registration.setLoadOnStartup(1);

registration.addMapping("/");

}

}

WebApplicationInitializer is an interface provided by Spring MVC that ensures your implementation is detected and automatically used to initialize any Servlet 3 container. An abstract base class implementation of WebApplicationInitializer named AbstractDispatcherServletInitializer makes it even easier to register the DispatcherServlet by simply overriding methods to specify the servlet mapping and the location of the DispatcherServlet configuration.

WebApplicationInitializer是SpringMVC提供的一个接口，它确保你的实现被检测到并自动用来初始化任何Servlet3容器，它是一个名为AbstractDispatcherServletInitializer 的抽象基类的实现，通过简单的方法重载来指定Servlet映射和DispatcherServlet配置文件的位置，使得注册DispatcherServlet更加容易

This is recommended for applications that use Java-based Spring configuration

建议应用程序使用基于Java Spring 的配置

**public** **class** **MyWebAppInitializer** **extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** Class<?>**[]** getRootConfigClasses() {

**return** null;

}

@Override

**protected** Class<?>**[]** getServletConfigClasses() {

**return** **new** Class**[]** { MyWebConfig.class };

}

@Override

**protected** String**[]** getServletMappings() {

**return** **new** String**[]** { "/" };

}

}

If using XML-based Spring configuration, you should extend directly from AbstractDispatcherServletInitializer

如果使用基于XML的Spring配置，你应该直接从AbstractDispatcherServletInitializer扩展

**public** **class** **MyWebAppInitializer** **extends** AbstractDispatcherServletInitializer {

@Override

**protected** WebApplicationContext createRootApplicationContext() {

**return** null;

}

@Override

**protected** WebApplicationContext createServletApplicationContext() {

XmlWebApplicationContext cxt = **new** XmlWebApplicationContext();

cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

**return** cxt;

}

@Override

**protected** String**[]** getServletMappings() {

**return** **new** String**[]** { "/" };

}

}

AbstractDispatcherServletInitializer also provides a convenient way to add Filter instances and have them automatically mapped to the DispatcherServlet

AbstractDispatcherServletInitializer 还提供了添加过滤器实例并将其自动映射到DispatcherServlet便捷方式

**public** **class** **MyWebAppInitializer** **extends** AbstractDispatcherServletInitializer {

*// ...*

@Override

**protected** Filter**[]** getServletFilters() {

**return** **new** Filter**[]** {

**new** HiddenHttpMethodFilter(), **new** CharacterEncodingFilter() };

}

}

Each filter is added with a default name based on its concrete type and automatically mapped to the DispatcherServlet.

每个过滤器都根据其具体类型添加了一个默认名称，并自动映射到DispatcherServlet

The isAsyncSupported protected method of AbstractDispatcherServletInitializer provides a single place to enable async support on the DispatcherServlet and all filters mapped to it. By default this flag is set to true.

AbstractDispatcherServletInitializer 的受保护方法isAsyncSupported提供了一个单独的位置来启用异步支持DispatcherServlet和所有的过滤器映射到它，默认情况下，此标志被设为true

Finally, if you need to further customize the DispatcherServlet itself, you can override the createDispatcherServlet method.

最后，如果你需要进一步自定义DispatcherServlet 本身，你能够重载createDispatcherServlet 方法

### 1.2.5. Processing

The DispatcherServlet processes requests as follows:

DispatcherServlet 处理请求如下：

* The WebApplicationContext is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE.

在请求中搜索和绑定WebApplicationContext 作为控制器和流程中其他元素可以使用的属性。它在DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE关键字下被默认绑定

* The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.

区域解析器绑定到请求，以便在处理请求时，流程中的元素能够使解析区域信息来使用

* The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.

主题解析器被绑定到请求，以便让视图等元素决定使用哪个主题，如果你没有使用主题，你能够忽略它

* If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process.

如果指定了multiparts文件解析器，则会检查请求中的multiparts，如果找到multiparts，请求将被包装到一个MultipartHttpServletRequest 中，以供流程中其他元素进一步处理

* An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering. Or alternatively for annotated controllers, the response may be rendered (within the HandlerAdapter) instead of returning a view.

搜索适当的处理程序，如果找到处理程序，与处理程序（预处理程序、后处理程序和控制器）相关联的执行链被执行，以便准备模型或渲染。或者对于带注解的控制器，响应可以被呈现（在HandlerAdapter中），而不是返回一个视图

* If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

如果返回模型，将会呈现视图，如果没有模型返回（也许是由于一个预处理或者后处理拦截了这个请求，可能是出于安全原因），没有视图呈现，因为请求已经被完成了

DispatcherServlet initialization parameters

DispatcherServlet 初始化参数

* **contextClass**：Class that implements WebApplicationContext, which instantiates the context used by this Servlet. By default, the XmlWebApplicationContext is used.

实现WebApplicationContext的类，它实例化这个Servlet使用的上下文，默认情况下，只用XmlWebApplicationContext

* **contextConfigLocation**：String that is passed to the context instance (specified by contextClass) to indicate where context(s) can be found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multiple contexts. In case of multiple context locations with beans that are defined twice, the latest location takes precedence.

传递给上下文实例(由contextClass指定)的字符串，指示在哪儿可以找到上下文，该字符串可能包含多个字符串（用逗号分隔）以支持多个上下文，在定义了两次bean的多个上下文位置的情况下，最新位置的优先

* **Namespace**：Namespace of the WebApplicationContext. Defaults to [servlet-name]-servlet.

WebApplicationContext的Namespace默认为[servlet-name]-servlet

### 1.2.6. Interception

* preHandle(..) — before the actual handler is executed

实际的处理程序之前被执行

* postHandle(..) — after the handler is executed

实际的处理程序之后被执行

* afterCompletion(..) — after the complete request has finished

整个请求完成之后完成

### 1.2.7. View Resolution

spring MVC defines the ViewResolver and View interfaces that enable you to render models in a browser without tying you to a specific view technology. ViewResolver provides a mapping between view names and actual views. View addresses the preparation of data before handing over to a specific view technology.

Spring MVC定义了视图解析器和视图接口，以确保你能够呈现模型到浏览器，而无需将你绑定到特殊的视图技术。视图解析器提供了一个视图名称和实际视图之间的映射，视图在交付给特定的视图技术之前处理准备数据

This table below provides more details on the ViewResolver hierarchy:

下表提供了更多视图解析器层级结构的细节

#### AbstractCachingViewResolver

Sub-classes of AbstractCachingViewResolver cache view instances that they resolve. Caching improves performance of certain view technologies. It’s possible to turn off the cache by setting the cache property to false. Furthermore, if you must refresh a certain view at runtime (for example when a FreeMarker template is modified), you can use the removeFromCache(String viewName, Locale loc) method

AbstractCachingViewResolver 子类缓存它们解析的视图实例，缓存提高了某些视图技术的性能，通过设置缓存属性为false可以关闭缓存，另外，如果你必须在运行时刷新一个某个视图（例如FreeMarker模板被修改），你可以使用removeFromCache（String viewName，Locale loc）方法

#### XmlViewResolver

Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.

ViewResolver的实现类，它接受一个用XML编写的配置文件，其中使用和spring的XML bean工厂相同的DTD，默认配置文件是/WEB-INF/views.xml.

#### ResourceBundleViewResolver

Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name, and for each view it is supposed to resolve, it uses the value of the property [viewname].(class) as the view class and the value of the property [viewname].url as the view url.

ViewResolver的实现类，它使用ResourceBundle中的一个bean定义，通过bundle的基础名称指定，对于它支持解析每种视图，用属性[viewname].(class)的值作为视图类和属性[viewname].url的值 作为视图的Url

#### UrlBasedViewResolver

Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.

ViewResolver 接口的简单实现，它实现了逻辑视图名称到URL的直接解析，没有明确的映射定义。如果你的逻辑视图名称以直观的方式匹配到视图资源的名称，而不需要随意的映射，那么这是适当的

#### InternalResourceViewResolver

Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).

UrlBasedViewResolver 的便捷子类，它支持InternalResourceView （实际上是Servlets和JSPs）和子类（如JS听力View和TilesView）。你可以使用setViewClass（..）为所有这个解析器生成的视图指定视图类

#### FreeMarkerViewResolver

Convenient subclass of UrlBasedViewResolver that supports FreeMarkerView and custom subclasses of them.

UrlBasedViewResolver 的便捷子类，它支持FreeMarkerView 和它们的自定义子类

#### ContentNegotiatingViewResolver

Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept  header

ViewResolver 接口的实现类，它解析基于请求文件名或Accept 头的视图

#### Redirect

The special redirect: prefix in a view name allows you to perform a redirect. The UrlBasedViewResolver (and subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

视图名称中的特殊“redirect:“前缀允许执行重定向，UrlBasedViewResolver （和子类）会将其识别为需要重定向的特殊指示，视图名称的其余部分将被视为重定向的网址

#### Forward

It is also possible to use a special forward: prefix for view names that are ultimately resolved by UrlBasedViewResolver and subclasses. This creates an InternalResourceView which does a RequestDispatcher.forward(). Therefore, this prefix is not useful with InternalResourceViewResolver and InternalResourceView (for JSPs) but it can be helpful if using another view technology, but still want to force a forward of a resource to be handled by the Servlet/JSP engine. Note that you may also chain multiple view resolvers, instead.

对于最终由UrlBasedViewResolver和其子类解析的视图名称，也能够使用特殊的“forward: ”前缀，这创建了一个InternalResourceView ，它执行RequestDispatcher.forward()，因此，这个前缀对InternalResourceViewResolver 和InternalResourceView （对于JSPs）是没用的，但是如果使用其他技术，但是仍然想强迫一个资源的转发由Servlet/JSP引擎来处理，它可能是有帮助的。请注意，你也可以链接多个视图解析器

### **1.2.8. Locale**

DispatcherServlet enables you to automatically resolve messages using the client’s locale. This is done with LocaleResolver objects.

DispatcherServlet 是你能够自动的使用客户端语言环境解析消息，这个是由LocaleResolver 对象完成的

When a request comes in, the DispatcherServlet looks for a locale resolver, and if it finds one it tries to use it to set the locale. Using the RequestContext.getLocale() method, you can always retrieve the locale that was resolved by the locale resolver.

当请求进入时，DispatcherServlet 会查找一个语言环境解析器，如果找到，它会尝试用它去设置语言环境，用RequestContext.getLocale() 方法，你始终能够检索到由语言环境解析器解析的区域设置

In addition to automatic locale resolution, you can also attach an interceptor to the handler mapping to change the locale under specific circumstances, for example, based on a parameter in the request.

除了自动区域环境解析之外，你还能够在特殊的环境下，将拦截器附加到处理程序映射，以更改区域设置（例如，基于请求中的参数）

Locale resolvers and interceptors are defined in the org.springframework.web.servlet.i18n package and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

区域设置解析器和拦截器被定义在rg.springframework.web.servlet.i18n包中，并以常规的方式配置在你的应用程序中。这里是spring包含的语言环境解析器的选择

#### TimeZone

#### Header resolver

#### Cookie resolver

#### Session resolver

#### Locale interceptor

### 1.2.9. Themes

You can apply Spring Web MVC framework themes to set the overall look-and-feel of your application, thereby enhancing user experience. A theme is a collection of static resources, typically style sheets and images, that affect the visual style of the application.

你能够使用Spring Web MVC框架主题来设置应用程序的整体外观，进而提高用户体验，一个主题是静态资源的集合，通常是样式表和图片，它们会影响应用程序的视觉风格

#### Define a theme

定义主题

To use themes in your web application, you must set up an implementation of the org.springframework.ui.context.ThemeSource interface. The WebApplicationContext interface extends ThemeSource but delegates its responsibilities to a dedicated implementation. By default the delegate will be an org.springframework.ui.context.support.ResourceBundleThemeSource implementation that loads properties files from the root of the classpath. To use a custom ThemeSource implementation or to configure the base name prefix of the ResourceBundleThemeSource, you can register a bean in the application context with the reserved name themeSource. The web application context automatically detects a bean with that name and uses it.

为了在你的web应用程序中使用主题，你必须设置org.springframework.ui.context.ThemeSource接口的一个实现，WebApplicationContext 接口扩展了ThemeSource，但将其职责委托给了专用的实现。默认情况下，委托将是一个org.springframework.ui.context.support.ResourceBundleThemeSource 的实现，它从类路径的根目录加载属性文件。要是要自定义ThemeSource实现或配置ResourceBundleThemeSource的基本名称前缀，你能够在应用程序的上下文中用保留名称themeSource注册一个bean，web因此程序自动检测具有该名称的bean并且使用它

When using the ResourceBundleThemeSource, a theme is defined in a simple properties file. The properties file lists the resources that make up the theme. Here is an example:

当使用ResourceBundleThemeSource时，主题定义在简单的属性文件中，这些属性文件列出了组成主题的资源,下面是一个示例:

styleSheet=/themes/cool/style.css

background=/themes/cool/img/coolBg.jpg

#### Resolve themes

解析主题

ThemeResolver implementations

FixedThemeResolver：Selects a fixed theme, set using the defaultThemeName property.

选择一个固定的主题,设置使用defaultThemeName属性

SessionThemeResolver：The theme is maintained in the user’s HTTP session. It only needs to be set once for each session, but is not persisted between sessions

主题被维护在用户的http session中，每次会话只会设置一次，但不会在会话间持续

CookieThemeResolver：The selected theme is stored in a cookie on the client.

选择的主题被存储在客户端的cookie中

Spring also provides a ThemeChangeInterceptor that allows theme changes on every request with a simple request parameter.

Spring也提供了一个ThemeChangeInterceptor ，它允许使用一个简单的请求参数来改变每次请求的主题

### 1.2.10. Multipart resolver

MultipartResolver from the org.springframework.web.multipart package is a strategy for parsing multipart requests including file uploads. There is one implementation based on [Commons FileUpload](https://jakarta.apache.org/commons/fileupload) and another based on Servlet 3.0 multipart request parsing.

来自org.springframework.web.multipart包中MultipartResolver解析包括文件上传在内的multipart 请求的一种策略，有一个基于 Commons FileUpload 的实现，另一个基于Servlet 3.0 multipart 请求解析

To enable multipart handling, you need declare a MultipartResolver bean in your DispatcherServlet Spring configuration with the name "multipartResolver". The DispatcherServlet detects it and applies it to incoming request. When a POST with content-type of "multipart/form-data" is received, the resolver parses the content and wraps the current HttpServletRequest as MultipartHttpServletRequest in order to provide access to resolved parts in addition to exposing them as request parameters.

要启用multipart 处理，你需要在DispatcherServlet Spring配置中声明一个名为“multipartResolver”的MultipartResolver bean，DispatcherServlet 检查它并将其应用到传入的请求，当接收到内容类型为"multipart/form-data"的POST时，解析器解析内容并将当前的HttpServletRequest 包装为MultipartHttpServletRequest ，以便提供对解析部分的访问，并将它们公开为请求参数

## 1.3 Filters

The spring-web module provides some useful filters.

Spring-web模块提供了一些有用的过滤器

### 1.3.1 HTTP PUT Form

Browsers can only submit form data via HTTP GET or HTTP POST but non-browser clients can also use HTTP PUT and PATCH. The Servlet API requires ServletRequest.getParameter\*() methods to support form field access only for HTTP POST.

浏览器仅仅能够通过HTTP GET或HTTP POST提交表单数据，但非浏览器客户端也能使用HTTP PUT和PATCH， Servlet API要求ServletRequest.getParameter\*() 方法仅支持HTTP POST的表单字段访问

The spring-web module provides HttpPutFormContentFilter that intercepts HTTP PUT and PATCH requests with content type application/x-www-form-urlencoded, reads the form data from the body of the request, and wraps the ServletRequest in order to make the form data available through the ServletRequest.getParameter\*() family of methods.

Spring-web模块提供了HttpPutFormContentFilter ，它拦截文本类型为application/x-www-form-urlencoded的HTTP PUT和PATCH请求 ，从请求体中读取表单数据，并封装ServletRequest ，以使表单数据通过 ServletRequest.getParameter\*() 系列的方法可用

### 1.3.2. Forwarded Headers

As a request goes through proxies such as load balancers the host, port, and scheme may change presenting a challenge for applications that need to create links to resources since the links should reflect the host, port, and scheme of the original request as seen from a client perspective.

当请求经过诸如负载均衡等的代理时，主机、端口和方案也许会改变，这对于需要创建资源链接的应用提出了挑战，因为链接应该反映从客户端看原始请求的主机、端口和方案

RFC 7239 defines the "Forwarded" HTTP header for proxies to use to provide information about the original request. There are also other non-standard headers in use such as "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto".

RFC 7239定义了代理用于提供原始请求信息的"Forwarded" HTTP头，还使用了其他非标准报头，例如"X-Forwarded-Host", "X-Forwarded-Port", 和"X-Forwarded-Proto"

ForwardedHeaderFilter detects, extracts, and uses information from the "Forwarded" header, or from "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto". It wraps the request in order to overlay its host, port, and scheme and also "hides" the forwarded headers for subsequent processing.

ForwardedHeaderFilter 发现、扩展并使用来自“Forwarded”报头或来自 "X-Forwarded-Host", "X-Forwarded-Port", 和"X-Forwarded-Proto"的信息. 它封装请求以便覆盖其主机、端口和方案，并且“隐藏”转发的报文头以供后续处理

Note that there are security considerations when using forwarded headers as explained in Section 8 of RFC 7239. At the application level it is difficult to determine whether forwarded headers can be trusted or not. This is why the network upstream should be configured correctly to filter out untrusted forwarded headers from the outside.

请注意，正如 RFC 7239中第八章所述，当使用转发报文头时，存在安全考虑事项，在应用级别很难去区分转发报文头是否可信，这就是为什么网络上游应该被正确配置，以从外部过滤出不可信的转发报文头

Applications that don’t have a proxy and don’t need to use forwarded headers can configure the ForwardedHeaderFilter to remove and ignore such headers.

没有代理和不需要使用转发报文头的应用程序能可以配置ForwardedHeaderFilter 过滤器以移除和忽视这些报文头

### 1.3.3. Shallow ETag

There is a ShallowEtagHeaderFilter. It is called shallow because it doesn’t have any knowledge of the content. Instead it relies on buffering actual content written to the response and computing the ETag value at the end.

有一个ShallowEtagHeaderFilter，它被叫做浅层，因为它对内容一无所知，相反，它依靠缓存写如响应的实际内容，并在末尾计算ETag的值

### 1.3.4. CORS

Spring MVC provides fine-grained support for CORS configuration through annotations on controllers. However when used with Spring Security it is advisable to rely on the built-in CorsFilter that must be ordered ahead of Spring Security’s chain of filters.

Spring MVC通过控制器上的注解为CORS配置提供了细粒度的支持，但是，当与Spring Security一起使用时，建议使用内置的CorsFilter ，它必须排在Spring Security过滤器链之前

## 1.4. Annotated Controllers

Spring MVC provides an annotation-based programming model where @Controller and @RestController components use annotations to express request mappings, request input, exception handling, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces.

SpringMVC提供了一个基于注解的编程模型，其中@Controller 和@RestController组件使用注解来标识请求映射、请求输入、异常处理等等。注解控制器具有灵活的方法签名。不必扩展基类，也不需要实现特殊的接口

@Controller**public** **class** **HelloController** {

@GetMapping("/hello")

**public** String handle(Model model) {

model.addAttribute("message", "Hello World!");

**return** "index";

}

}

### 1.4.1. Declaration

You can define controller beans using a standard Spring bean definition in the Servlet’s WebApplicationContext. The @Controller stereotype allows for auto-detection, aligned with Spring general support for detecting @Component classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.

你可以在Servlet的WebApplicationContext中使用标准的Spring bean定义来定义控制器bean。@Controller原型允许自动检测，与Spring通用支持保持一致，用于在类路径中检查@Component类，并为它们自动注册bean定义。它也充当注解类的原型，表明他作为web组件的角色

To enable auto-detection of such @Controller beans, you can add component scanning to your Java configuration:

要启用诸如@Controller beans的自动检测，可以将组件扫描添加到Java配置中：

@Configuration

@ComponentScan("org.example.web")

**public** **class** **WebConfig** {

*// ...*

}

The XML configuration equivalent:

等效的XML配置:

<?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:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="org.example.web"/>

*<!-- ... -->*

</beans>

@RestController is a composed annotation that is itself annotated with @Controller and @ResponseBody indicating a controller whose every method inherits the type-level @ResponseBody annotation and therefore writes to the response body (vs model-and-vew rendering).

@RestController是一个组合的注解，它自己用@Controller 和@ResponseBody注解来表示一个控制器，它的每个方法都继承类型级别的 @ResponseBody注解，因此写入相应体(vs model-and-vew rendering).

#### AOP proxies

AOP代理

In some cases a controller may need to be decorated with an AOP proxy at runtime. One example is if you choose to have @Transactional annotations directly on the controller. When this is the case, for controllers specifically, we recommend using class-based proxying. This is typically the default choice with controllers. However if a controller must implement an interface that is not a Spring Context callback (e.g. InitializingBean, \*Aware, etc), you may need to explicitly configure class-based proxying. For example with <tx:annotation-driven/>, change to <tx:annotation-driven proxy-target-class="true"/>.

在一些情况在，控制器需要在运行时用AOP代理进行装饰，一个例子是如果你选择在控制器上直接使用@Transactional 注解，当这种情况下，对特定的控制器，我们推荐使用基于类的代理。这通常是控制器默认选择，但是，如果控制器必须实现一个不是Spring Context回调的接口（例如，InitializingBean, \*Aware等），

你也许应该明确地配置基于类的代理，例如，使用<tx:annotation-driven/>改为<tx:annotation-driven proxy-target-class="true"/>

### 1.4.2. Request Mapping

The @RequestMapping annotation is used to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. It can be used at the class-level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

@RequestMapping注解用来将请求映射到控制器方法，它具有通过URL、HTTP方法、请求参数、请求头和媒体类型进行匹配的各种属性，可以再类级别使用它用来表示共享映射，或者在方法级别用于缩小到特点的端点映射

There are also HTTP method specific shortcut variants of @RequestMapping:

@ RequestMapping还有特定于HTTP方法的快捷方式变体:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

The shortcut variants are composed annotations — themselves annotated with @RequestMapping. They are commonly used at the method level. At the class level an @RequestMapping is more useful for expressing shared mappings.

快捷方式变体由注解组成，它们自己用@RequestMapping注解，通常用于方法级别，在类级别，

@RequestMapping对表达共享映射更有用

@RestController@RequestMapping("/persons")**class** **PersonController** {

@GetMapping("/{id}")

**public** Person getPerson(@PathVariable Long id) {

*// ...*

}

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

**public** **void** add(@RequestBody Person person) {

*// ...*

}

}

#### URI patterns

You can map requests using glob patterns and wildcards:

你可以使用glob模式和通配符来映射请求

? matches one character

？匹配一个字符

\* matches zero or more characters within a path segment

\*匹配路径段中的零个或多个字符

\*\* match zero or more path segments

\*\*匹配零个或多个路径段

You can also declare URI variables and access their values with @PathVariable:

你也可以声明URI变量并用 @PathVariable访问它们的值

@GetMapping("/owners/{ownerId}/pets/{petId}")**public** Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

*// ...*

}

URI variables can be declared at the class and method level:

URI变量能被声明在类级别和方法级别

@Controller@RequestMapping("/owners/{ownerId}")

**public** **class** **OwnerController** {

@GetMapping("/pets/{petId}")

**public** Pet findPet(@PathVariable Long ownerId,

@PathVariable Long petId) {

*// ...*

}

}

URI variables are automatically converted to the appropriate type or`TypeMismatchException` is raised. Simple types — int, long, Date, are supported by default and you can register support for any other data type.

URI变量能够自动转换到适当的类型，否在会引发“TypeMismatchException”，简单类型（int，long，Date）被默认支持，你也能注册对任何其他类型的支持

URI path patterns can also have embedded ${…​} placeholders that are resolved on startup via PropertyPlaceHolderConfigurer against local, system, environment, and other property sources. This can be used for example to parameterize a base URL based on some external configuration.

URI路径模式也能嵌入${…​}占位符，它通过PropertyPlaceHolderConfigurer 在启动时解析本地、系统、环境和其他属性资源。这个可能被用于例如基于一些外部配置参数化基本的URL

### 1.4.3. Handler Methods

@RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

@RequestMapping 处理方法有一个灵活的签名，它能够从一系列支持的控制器方法参数和返回值中选择

#### Type Conversion

类型转换

Some annotated controller method arguments that represent String-based request input — e.g. @RequestParam, @RequestHeader, @PathVariable, @MatrixVariable, and @CookieValue, may require type conversion if the argument is declared as something other than String.

表示基于字符串请求输入的一些注解的控制器方法参数，例如@RequestParam, @RequestHeader, @PathVariable, @MatrixVariable, and @CookieValue，如果参数声明为字符串以外的类型，则可能需要进行类型转换

For such cases type conversion is automatically applied based on the configured converters. By default simple types such as int, long, Date, etc. are supported. Type conversion can be customized through a WebDataBinder

在一些情况下，将基于配置的转换器自动应用类型转换，默认支持简单类型，例如int, long, Date等。类型转换能通过WebDataBinder自定义

#### Matrix variables

矩阵变量

*// GET /pets/42;q=11;r=22*

@GetMapping("/pets/{petId}")

**public** **void** findPet(@PathVariable String petId, @MatrixVariable **int** q) {

*// petId == 42*

*// q == 11*

}

#### @RequestParam

Use the @RequestParam annotation to bind Servlet request parameters (i.e. query parameters or form data) to a method argument in a controller.

使用 @RequestParam注解将Servlet请求参数（例如请求参数或表单数）绑定到控制器中的方法参数

The following code snippet shows the usage:

下面的代码片段显示了用法

@Controller

@RequestMapping("/pets")

**public** **class** **EditPetForm** {

*// ...*

@GetMapping

**public** String setupForm(**@RequestParam**(**"petId"**)** int **petId****, Model model) {

Pet pet = this.clinic.loadPet(petId);

model.addAttribute("pet", pet);

**return** "petForm";

}

*// ...*

}

Method parameters using this annotation are required by default, but you can specify that a method parameter is optional by setting @RequestParam's required flag to false or by declaring the argument with an java.util.Optional wrapper.

使用这个注解的方法参数默认是必须的，但是你能通过设置@RequestParam的required 标识为false或者声明用一个 java.util.Optional封装的参数来指定方法参数是可选的

Type conversion is applied automatically if the target method parameter type is not String.

如果目标方法参数类型不是字符串，则自动的进行类型转换

When an @RequestParam annotation is declared as Map<String, String> or MultiValueMap<String, String> argument, the map is populated with all request parameters.

当@RequestParam注解别声明为Map<String, String> 或MultiValueMap<String, String>参数时，映射将填充所有的请求参数

Note that use of @RequestParam is optional, e.g. to set its attributes. By default any argument that is a simple value type, as determined by BeanUtils#isSimpleProperty, and is not resolved by any other argument resolver, is treated as if it was annotated with @RequestParam.

请注意，@ RequestParam的使用是可选的，例如设置其属性。默认情况下，由BeanUtils # issempleproperty确定且未由任何其他参数解析器解析的简单值类型的任何参数将被视为使用@ RequestParam进行注释

#### @RequestHeader

Use the @RequestHeader annotation to bind a request header to a method argument in a controller.

使用@RequestHeader注解将请求头绑定到控制器中的方法参数

Given request with headers:

给定有请求头的请求

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

The following gets the value of the Accept-Encoding and Keep-Alive headers:

下面获取Accept-Encoding和Keep-Alive请求头的值

@GetMapping("/demo")**public** **void** handle(

**@RequestHeader**(**"Accept-Encoding"**)**** String encoding,

**@RequestHeader**(**"Keep-Alive"**)**** **long** keepAlive) {

*//...*

}

Type conversion is applied automatically if the target method parameter type is not String.

如果目标方法参数不是字符串类型，将自动的使用类型转换

When an @RequestHeader annotation is used on a Map<String, String>, MultiValueMap<String, String>, or HttpHeadersargument, the map is populated with all header values.

当@RequestHeader注解用在Map<String, String>, MultiValueMap<String, String>, or HttpHeadersargument上时，映射将填充所有的请求头

#### @CookieValue

Use the @CookieValue annotation to bind the value of an HTTP cookie to a method argument in a controller.

使用@CookieValue注解将HTTP cookie的值绑定到控制器中的方法参数

Given request with the following cookie:

给定含有下列cookie的请求

JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84

The following code sample demonstrates how to get the cookie value:

下面的代码简单的示范了如何获取cookie的值

@GetMapping("/demo")

**public** **void** handle(**@CookieValue**(**"JSESSIONID"**)**** String cookie) {

*//...*

}

Type conversion is applied automatically if the target method parameter type is not String

如果目标方法参数不是字符串类型，将自动的使用类型转换

#### @ModelAttribute

Use the @ModelAttribute annotation on a method argument to access an attribute from the model, or have it instantiated if not present. The model attribute is also overlaid with values from HTTP Servlet request parameters whose names match to field names. This is referred to as data binding and it saves you from having to deal with parsing and converting individual query parameters and form fields.

在方法参数上使用@ModelAttribute注解来访问模型中的属性，或者如果不存在，就将其实例化。模型属性也覆盖了名称和字段名称匹配的HTTP Servlet 请求参数的值，这被称为数据绑定，它使你不必处理解析和转换单个查询参数和表单数据

For example:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")**public** String processSubmit(**@ModelAttribute **Pet pet****) { }

The Pet instance above is resolved as follows:

上面的Pet实例解析如下：

1. From the model if already added via Model Methods.

如果已经通过模型方法添加，则从模型中获取

1. From the HTTP session via @SessionAttributes.

通过@SessionAttributes从HTTP 会话获取

1. From a URI path variable passed through a Converter

通过转换器从URL路径变量获取

1. From the invocation of a default constructor.

从默认构造函数调用

1. From the invocation of a "primary constructor" with arguments matching to Servlet request parameters; argument names are determined via JavaBeans @ConstructorProperties or via runtime-retained parameter names in the bytecode.

从参数匹配Servlet请求参数的“主构造函数”调用，参数名称通过JavaBeans @ConstructorProperties或字节码中运行时保留的参数名称确定

While it is common to use a Model Methods to populate the model with attributes, one other alternative is to rely on a Converter<String, T> in combination with a URI path variable convention.

虽然通常使用模型方法来用属性填充模型，但另一种方法是依靠Converter<String, T>与URI路径变量约定结合使用，

In the example below the model attribute name "account" matches the URI path variable "account" and the Account is loaded by passing the String account number through a registered Converter<String, Account>:

在下面的示例中，模型属性名称“account”和URI路径变量“account”相匹配，通过将String字符账号传递到已注册的Converter<String, Account>来加载账户

@PutMapping("/accounts/{account}")

**public** String save(@ModelAttribute("account") Account account) {

*// ...*

}

After the model attribute instance is obtained, data binding is applied. The WebDataBinder class matches Servlet request parameter names (query parameters and form fields) to field names on the target Object. Matching fields are populated after type conversion is applied where necessary

获得模型属性实例后，应用数据绑定，WebDataBinder类将Servlet请求参数名称（查询参数和表单字段）和目标对象字段名称进行匹配，必要时，应用类型转换后填充匹配字段

Data binding may result in errors. By default a BindException is raised but to check for such errors in the controller method, add a BindingResult argument immediately next to the @ModelAttribute as shown below:

数据绑定可能导致错误，默认情况下会引发一个BindException ，但是要在控制器方法中检查这样的错误，立即在 @ModelAttribute后面添加一个BindingResult 参数，如下所示：

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")**public** String processSubmit(**@ModelAttribute**(**"pet"**) Pet pet****, BindingResult result) {

**if** (result.hasErrors()) {

**return** "petForm";

}

*// ...*

}

In some cases you may want access to a model attribute without data binding. For such cases you can inject the Model into the controller and access it directly or alternatively set @ModelAttribute(binding=false) as shown below:

在一些情况下，你可能需要访问一个没有数据绑定的模型属性，对于这种情况，你可以注入模型到控制器并直接访问它或可以设置@ModelAttribute(binding=false) ，如下所示

@ModelAttribute

**public** AccountForm setUpForm() {

**return** **new** AccountForm();

}

@ModelAttribute

**public** Account findAccount(@PathVariable String accountId) {

**return** accountRepository.findOne(accountId);

}

@PostMapping("update")

**public** String update(@Valid AccountUpdateForm form, BindingResult result,

**@ModelAttribute**(binding=**false**)**** Account account) {

*// ...*

}

Validation can be applied automatically after data binding by adding the javax.validation.Valid annotation or Spring’s @Validated annotation

通过添加javax.validation.Valid注解或者Spring的@Validated注解，可以再数据绑定后自动的应用验证

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

**public** String processSubmit(**@Valid@ModelAttribute**(**"pet"**) Pet pet****, BindingResult result) {

**if** (result.hasErrors()) {

**return** "petForm";

}

*// ...*

}

Note that use of @ModelAttribute is optional, e.g. to set its attributes. By default any argument that is not a simple value type, as determined by BeanUtils#isSimpleProperty, and is not resolved by any other argument resolver, is treated as if it was annotated with @ModelAttribute.

请注意，@ModelAttribute的使用是可选的，例如，设置它的属性，默认情况下，由BeanUtils#isSimpleProperty确定的非简单类型且未由任何其他参数解析器解析的任何参数都将视为使用ModelAttribute进行注释

#### @SessionAttributes

@SessionAttributes is used to store model attributes in the HTTP Servlet session between requests. It is a type-level annotation that declares session attributes used by a specific controller. This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session for subsequent requests to access.

@SessionAttributes用于在请求之间的HTTP Servlet会话中存储模型属性，它是一个类型级别的注解，用于声明特定控制器使用的的会话属性，这通常会列出应该透明的存储在session中的模型属性的名称或者模型属性的类型，以便后续请求访问

For example:

举例：

@Controller**@SessionAttributes**(**"pet"**)****

**public** **class** **EditPetForm** {

*// ...*

}

On the first request when a model attribute with the name "pet" is added to the model, it is automatically promoted to and saved in the HTTP Servlet session. It remains there until another controller method uses a SessionStatus method argument to clear the storage:

在第一次请求中，当名称为“pet”的模型数据被添加到模型中时，它将被自动升级并保存到HTTP Servlet会话中，它保持他们直到另一个控制器方法使用SessionStatus 方法参数清除这个存储

@Controller**@SessionAttributes**(**"pet"**)****

**public** **class** **EditPetForm** {

*// ...*

@PostMapping("/pets/{id}")

**public** String handle(Pet pet, BindingResult errors,

SessionStatus status) {

**if** (errors.hasErrors) {

*// ...*

}

status.setComplete();

*// ...*

}

}

}

#### @SessionAttribute

If you need access to pre-existing session attributes that are managed globally, i.e. outside the controller (e.g. by a filter), and may or may not be present use the @SessionAttribute annotation on a method parameter:

如果你需要访问全局（控制器之外，例如过滤器）管理的预先存在的session属性，并且也许存在，也许不存在，请在方法参数上使用@SessionAttribute

@RequestMapping("/")**public** String handle(**@SessionAttribute** User user) {

*// ...*

}

#### Redirect attributes

By default all model attributes are considered to be exposed as URI template variables in the redirect URL. Of the remaining attributes those that are primitive types or collections/arrays of primitive types are automatically appended as query parameters.

默认情况下，所有模型属性都被认为是作为重定向URL中的URI模板变量公开的，在剩余的属性中，那些原始类型和原始类型的集合/数据都自动的附加为查询参数

Appending primitive type attributes as query parameters may be the desired result if a model instance was prepared specifically for the redirect. However, in annotated controllers the model may contain additional attributes added for rendering purposes (e.g. drop-down field values). To avoid the possibility of having such attributes appear in the URL, an @RequestMapping method can declare an argument of type RedirectAttributes and use it to specify the exact attributes to make available to RedirectView. If the method does redirect, the content of RedirectAttributes is used. Otherwise the content of the model is used.

如果为重定向专门准备了模型实例，那么附加原始类型作为查询参数是期望的结果，但是，在注解控制器中，模型也许包含为渲染目的而添加的额外属性（例如，下拉字段值）。为了避免URL中有这种属性出现的可能性，@RequestMapping方法可以声明一个类型为RedirectAttributes 的参数，并用它指定可能用于RedirectView的确定属性。如果方法确实重定向，则使用RedirectAttributes 的内容，否在使用模型的内容

The RequestMappingHandlerAdapter provides a flag called "ignoreDefaultModelOnRedirect" that can be used to indicate the content of the default Model should never be used if a controller method redirects. Instead the controller method should declare an attribute of type RedirectAttributes or if it doesn’t do so no attributes should be passed on to RedirectView. Both the MVC namespace and the MVC Java config keep this flag set to false in order to maintain backwards compatibility. However, for new applications we recommend setting it to true

RequestMappingHandlerAdapter 提供了一个叫做“ignoreDefaultModelOnRedirect”的标志，

如果控制器方法重定向，可以用来表示默认模型的内容不被使用。相反，控制器方法应该声明类型为RedirectAttributes 的属性，或者如果不这样中，则任何属性不应该传递给RedirectView，MVC命名空间和MVC Java 配置都将这个标识设置为false，以保持向后兼容。然而，对于新的应用程序，我们建议设置为true

Note that URI template variables from the present request are automatically made available when expanding a redirect URL and do not need to be added explicitly neither through Model nor RedirectAttributes. For example:

请注意，当前请求中的URI模板变量在扩展重定向URL时自动可用，不需要通过添加显式的Model或RedirectAttributes，例如：

@PostMapping("/files/{path}")

**public** String upload(...) {

*// ...*

**return** "redirect:files/{path}";

}

#### Flash attributes

Flash attributes provide a way for one request to store attributes intended for use in another. This is most commonly needed when redirecting — for example, the Post/Redirect/Get pattern. Flash attributes are saved temporarily before the redirect (typically in the session) to be made available to the request after the redirect and removed immediately.

Flash属性提供了一种请求存储属性供另一种使用的方法，这是重定向时最常见的需求，举个例子，Post/Redirect/Get模式。在从重定向（通常在会话中）之前临时保存Flash属性，以便在重定向之后对请求可用并立即删除

Spring MVC has two main abstractions in support of flash attributes. FlashMap is used to hold flash attributes while FlashMapManager is used to store, retrieve, and manage FlashMap instances.

Spring MVC有两种主要的抽象来支持flash属性，FlashMap又来保存false属性，而FlashMapManager用来存储、检索和管理FlashMap实例

Flash attribute support is always "on" and does not need to enabled explicitly although if not used, it never causes HTTP session creation. On each request there is an "input" FlashMap with attributes passed from a previous request (if any) and an "output"FlashMap with attributes to save for a subsequent request. Both FlashMap instances are accessible from anywhere in Spring MVC through static methods in RequestContextUtils.

Flash属性支持总是处于“on”状态，不需要显式开启，即使不适用，也不会导致创建HTTP会话。每个请求都有一个包含从前一个请求（如果有）传递属性 的“input”FlashMap，和为下一个请求保存属性的 "output"FlashMap，两个实例都可以从SpringMVC的任何地方通过RequestContextUtils中的静态方法访问

Annotated controllers typically do not need to work with FlashMap directly. Instead an @RequestMapping method can accept an argument of type RedirectAttributes and use it to add flash attributes for a redirect scenario. Flash attributes added via RedirectAttributes are automatically propagated to the "output" FlashMap. Similarly, after the redirect, attributes from the "input" FlashMap are automatically added to the Model of the controller serving the target URL.

注解控制器通常不需要直接的使用FlashMap，相反的，@RequestMapping可以访问类型为RedirectAttributes 的属性并用它为重定向场景添加flash属性，通过RedirectAttributes 添加的Flash属性自动

传播到“output” FlashMap，相似的，在重定向之后，"input" FlashMap 中的属性自动的添加到提供目标URL的控制器模型中

#### Multipart

After a MultipartResolver has been enabled, the content of POST requests with "multipart/form-data" is parsed and accessible as regular request parameters. In the example below we access one regular form field and one uploaded file:

在启用MultipartResolver 后，具有"multipart/form-data"的POST请求的内容作为一个常规的请求参数来进行解析和访问。在下面的例子中，我们访问一个常规的表单字段和一个上传文件

@Controller

**public** **class** **FileUploadController** {

@PostMapping("/form")

**public** String handleFormUpload(@RequestParam("name") String name,

@RequestParam("file") MultipartFile file) {

**if** (!file.isEmpty()) {

**byte[]** bytes = file.getBytes();

*// store the bytes somewhere*

**return** "redirect:uploadSuccess";

}

**return** "redirect:uploadFailure";

}

}

When using Servlet 3.0 multipart parsing you can also use javax.servlet.http.Part as a method argument instead of Spring’s MultipartFile.

当时用Servlet3.0的multipart 解析时，你也可以使用javax.servlet.http.Part作为方法参数而不是Spring的MultipartFile

Multipart content can also be used as part of data binding to a command object. For example the above form field and file could have been fields on a form object:

Multipart 内容也可以用作数据绑定到命令对象的一部分，例如，上面的表单字段和文件有可能是表单对象上的字段

**class** **MyForm** {

**private** String name;

**private** MultipartFile file;

*// ...*

}

@Controller**public** **class** **FileUploadController** {

@PostMapping("/form")

**public** String handleFormUpload(MyForm form, BindingResult errors) {

**if** (!form.getFile().isEmpty()) {

**byte[]** bytes = form.getFile().getBytes();

*// store the bytes somewhere*

**return** "redirect:uploadSuccess";

}

**return** "redirect:uploadFailure";

}

}

Multipart requests can also be submitted from non-browser clients in a RESTful service scenario. For example a file along with JSON:

在RESTFul服务场景中，Multipart 请求也可以从非浏览器客户端提交，例如，一个和JSON在一起的文件

POST /someUrl

Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="meta-data"

Content-Type: application/json; charset=UTF-8

Content-Transfer-Encoding: 8bit

{

"name": "value"

}

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="file-data"; filename="file.properties"

Content-Type: text/xml

Content-Transfer-Encoding: 8bit

... File Data ...

You can access the "meta-data" part with @RequestParam as a String but you’ll probably want it deserialized from JSON (similar to @RequestBody). Use the @RequestPart annotation to access a multipart after converting it with an HttpMessageConverter:

你可以用@RequestParam作为字符串来访问"meta-data" 部分，但你可能需要从JSON反序列化（类似于@RequestBody），在使用HttpMessageConverter转换之后，用 @RequestPart注解来访问multipart

@PostMapping("/")

**public** String handle(**@RequestPart**(**"meta-data"**) MetaData metadata,****

**@RequestPart**(**"file-data"**) MultipartFile file****) {

*// ...*

}

@RequestPart can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a MethodArgumentNotValidException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@RequestPart可以和javax.validation.Valid, 或Spring的 @Validated 注解联合使用,这会到这标准的bean验证应用标准的bean验证。默认情况下，验证错误引发MethodArgumentNotValidException ，该异常转换为400(BAD\_REQUEST)相应，或者，验证错误可以通过Errors或BindingResult 参数在控制器本地处理

@PostMapping("/")

**public** String handle(**@Valid** @RequestPart("meta-data") MetaData metadata,

****BindingResult result****) {

*// ...*

}

#### @RequestBody

Use the @RequestBody annotation to have the request body read and deserialized into an Object through an HttpMessageConverter. Below is an example with an @RequestBody argument:

使用@RequestBody 世界来让请求体通过HttpMessageConverter读取并反序列化成一个对象，下面是一个@RequestBody参数的例子

@PostMapping("/accounts")

**public** **void** handle(@RequestBody Account account) {

*// ...*

}

You can use the Message Converters option of the MVC Config to configure or customize message conversion.

你可以使用MVC配置的消息转换选择来配置或自定义消息转换

@RequestBody can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a MethodArgumentNotValidException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@RequestBody可以与javax.validation.Valid或 Spring的 @Validated注解联合使用，这回导致应用标准bean验证。默认情况下，验证错误会引发MethodArgumentNotValidException ，该异常转换为400(BAD\_REQUEST)相应，或者，验证错误可以通过Errors或BindingResult 参数在控制器本地处理

@PostMapping("/accounts")

**public** **void** handle(@Valid @RequestBody Account account, BindingResult result) {

*// ...*

}

#### HttpEntity

HttpEntity is more or less identical to using @RequestBody but based on a container object that exposes request headers and body. Below is an example:

HttpEntity 或多或少与使用@RequestBody相同，但是基于一个公开请求头和体的容器对象，下面是一个例子

@PostMapping("/accounts")

**public** **void** handle(HttpEntity<Account> entity) {

*// ...*

}

#### @ResponseBody

Use the @ResponseBody annotation on a method to have the return serialized to the response body through anHttpMessageConverter. For example:

在一个方法上用@ResponseBody注解来通过anHttpMessageConverter将返回序列号到响应体，例如

@GetMapping("/accounts/{id}")

@ResponseBody**public** Account handle() {

*// ...*

}

@ResponseBody is also supported at the class level in which case it is inherited by all controller methods. This is the effect of @RestController which is nothing more than a meta-annotation marked with @Controller and @ResponseBody.

在类级别也支持@ResponseBody，在这种情况下，它被所有的控制器方法继承。这个是@RestController的效果，它只不过是用@Controller 和@ResponseBody标注.一个元注解

@ResponseBody may be used with reactive types.

@ResponseBody可以和反应类型一起使用

You can use the Message Converters option of the MVC Config  to configure or customize message conversion.

你可以使用MVC配置的消息转换器选项来配置或自定义消息转换器

@ResponseBody methods can be combined with JSON serialization views.

@ResponseBody方法可以与JSON序列号视图相结合

#### ResponseEntity

ResponseEntity is more or less identical to using @ResponseBody but based on a container object that specifies request headers and body. Below is an example:

ResponseEntity 或多或少与使用@ResponseBody相同，但是基于指定请求头和体的容器对象，下面是一个例子

@PostMapping("/something")**public** ResponseEntity<String> handle() {

*// ...*

URI location = ...

return **new** ResponseEntity.created(location).build();

}

#### Jackson JSON

Spring MVC provides built-in support for Jackson’s Serialization Views which allows rendering only a subset of all fields in an Object. To use it with @ResponseBody or ResponseEntity controller methods, use Jackson’s @JsonView annotation to activate a serialization view class:

SpringMVC为Jackson的序列化视图提供了内置支持，它允许仅渲染一个对象所有字段的子集，

与@ResponseBody 或 ResponseEntity控制器方法一起使用它，用 Jackson的@JsonView注解来激活一个序列化视图类

@RestController

**public** **class** **UserController** {

@GetMapping("/user")

@JsonView(User.WithoutPasswordView.class)

**public** User getUser() {

**return** **new** User("eric", "7!jd#h23");

}

}

**public** **class** **User** {

**public** **interface** **WithoutPasswordView** {};

**public** **interface** **WithPasswordView** **extends** WithoutPasswordView {};

**private** String username;

**private** String password;

**public** User() {

}

**public** User(String username, String password) {

this.username = username;

this.password = password;

}

@JsonView(WithoutPasswordView.class)

**public** String getUsername() {

**return** this.username;

}

@JsonView(WithPasswordView.class)

**public** String getPassword() {

**return** this.password;

}

}

@JsonView allows an array of view classes but you can only specify only one per controller method. Use a composite interface if you need to activate multiple views.

@JsonView 运行一个视图类的数组，但是每个控制器方法只能指定一个。如果你需要激活多视图，请使用复合接口

For controllers relying on view resolution, simply add the serialization view class to the model:

对于依赖视图解析器的控制器，只需要将序列化视图类添加到模型即可

@Controller

**public** **class** **UserController** **extends** AbstractController {

@GetMapping("/user")

**public** String getUser(Model model) {

model.addAttribute("user", **new** User("eric", "7!jd#h23"));

model.addAttribute(JsonView.class.getName(), User.WithoutPasswordView.class);

**return** "userView";

}

}

**Jackson JSONP**

In order to enable JSONP support for @ResponseBody and ResponseEntity methods, declare an @ControllerAdvice bean that extends AbstractJsonpResponseBodyAdvice as shown below where the constructor argument indicates the JSONP query parameter name(s):

为了给@ResponseBody 和ResponseEntity方法启用JSONP支持，声明一个扩展自AbstractJsonpResponseBodyAdvice 得@ControllerAdvice bean，如下所示，其中构造参数表示JSONP查询参数的名称

@ControllerAdvice

**public** **class** **JsonpAdvice** **extends** AbstractJsonpResponseBodyAdvice {

**public** JsonpAdvice() {

super("callback");

}

}

For controllers relying on view resolution, JSONP is automatically enabled when the request has a query parameter named jsonp or callback. Those names can be customized through jsonpParameterNames property.

对于依赖于视图解析器的控制器，当请求有一个名称为jsonp或callback的查询参数时，它会自启动，这些名称可以通过jsonpParameterNames 属性来自定义

### 1.4.4. Model Methods

The @ModelAttribute annotation can be used on @RequestMapping method arguments to create or access an Object from the model and bind it to the request. @ModelAttribute can also be used as a method-level annotation on controller methods whose purpose is not to handle requests but to add commonly needed model attributes prior to request handling.

 可以在@RequestMapping 方法参数使用@ModelAttribute注解，用来从模型创建或访问一个对象并将其绑定到请求。@ModelAttribute 也可以用作控制器方法上的方法级别注解，其目的是不处理请求，而是在请求处理之前添加通常需要的模型属性

A controller can have any number of @ModelAttribute methods. All such methods are invoked before @RequestMapping methods in the same controller. A @ModelAttribute method can also be shared across controllers via @ControllerAdvice.

控制器可以由任意数量的@ModelAttribute 方法，所有这些方法都在同一个控制器中的@RequestMapping方法之前调用，@ModelAttribute 也能通过 @ControllerAdvice在控制器之间共享

@ModelAttribute methods have flexible method signatures. They support many of the same arguments as @RequestMapping methods except for @ModelAttribute itself nor anything related to the request body.

@ModelAttribute方法具有灵活的方法签名，它们支持许多与@RequestMapping方法相同的参数，除了@ModelAttribute自身，或者与请求体没有任何关联的东西

An example @ModelAttribute method:

一个@ModelAttribute方法示例：

@ModelAttribute

**public** **void** populateModel(@RequestParam String number, Model model) {

model.addAttribute(accountRepository.findAccount(number));

*// add more ...*

}

To add one attribute only:

仅添加一个属性

@ModelAttribute

**public** Account addAccount(@RequestParam String number) {

**return** accountRepository.findAccount(number);

}

@ModelAttribute can also be used as a method-level annotation on @RequestMapping methods in which case the return value of the @RequestMapping method is interpreted as a model attribute. This is typically not required, as it is the default behavior in HTML controllers, unless the return value is a String which would otherwise be interpreted as a view name . @ModelAttribute can also help to customize the model attribute name:

@ModelAttribute也能用作@RequestMapping方法上的方法级别的注解，在这种情况下，@RequestMapping 方法的返回值被解释为模型属性。这个通常不是必要的，因为它是HTML控制器中的默认行为，除非返回值是一个字符串，否则将被视为视图名称， @ModelAttribute也可以帮助定制模型属性名称

@GetMapping("/accounts/{id}")@ModelAttribute("myAccount")

**public** Account handle() {

*// ...*

**return** account;

}

### 1.4.5. Binder Methods

@InitBinder methods in an @Controller or @ControllerAdvice class can be used to customize type conversion for method arguments that represent String-based request values (e.g. request parameters, path variables, headers, cookies, and others). Type conversion also applies during data binding of request parameters onto @ModelAttribute arguments (i.e. command objects).

@Controller或@ControllerAdvice类中的@InitBinder方法可以用来自定义表示基于字符串(例如请求参数、路径变量、标头、cookies等)的请求值的方法参数的类型转换。类型转换也适用于将请求参数绑定到@ModelAttribute 参数（即命令对象）的过程

@InitBinder methods can register controller-specific java.bean.PropertyEditor, or Spring Converter and Formatter components. In addition, the MVC config can be used to register Converter and Formatter types in a globally shared FormattingConversionService.

@InitBinder方法可以注册特定于控制器的java.bean.PropertyEditor，或String的转换器和格式化组件。另外，MVC 配置可以用于在全局共享的FormattingConversionService中注册转换器和格式化程序类型

@InitBinder methods support many of the same arguments that a @RequestMapping methods do, except for @ModelAttribute(command object) arguments. Typically they’re are declared with a WebDataBinder argument, for registrations, and a void return value. Below is an example:

@InitBinder方法支持许多与@RequestMapping方法相同的参数，除了@ModelAttribute（command object）参数，通常它们是用一个WebDataBinder 参数来声明的，用于注册和一个void的返回值，下面是一个例子：

@Controller**public** **class** **FormController** {

**@InitBinder**

**public** **void** initBinder(WebDataBinder binder) {

SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd");

dateFormat.setLenient(false);

binder.registerCustomEditor(Date.class, **new** CustomDateEditor(dateFormat, false));

}

*// ...*

}

Alternatively when using a Formatter-based setup through a shared FormattingConversionService, you could re-use the same approach and register controller-specific Formatter's:

或者，当通过共享的FormattingConversionService使用基于Formatter的设置时，可以重新使用相同的方法并注册控制器特定的Formatter：

@Controller

**public** **class** **FormController** {

**@InitBinder**

**protected** **void** initBinder(WebDataBinder binder) {

binder.addCustomFormatter(**new** DateFormatter("yyyy-MM-dd"));

}

*// ...*

}

### 1.4.6. Controller Advice

Typically @ExceptionHandler, @InitBinder, and @ModelAttribute methods apply within the @Controller class (or class hierarchy) they are declared in. If you want such methods to apply more globally, across controllers, you can declare them in a class marked with @ControllerAdvice or @RestControllerAdvice.

通常，@ExceptionHandler, @InitBinder, 和@ModelAttribute方法都适用于声明@Controller类（或类层次结构）中。如果希望跨控制器全局使用这样的方法，你可以在标有@ControllerAdvice 或 @RestControllerAdvice的类上声明它们

@ControllerAdvice is marked with @Component which means such classes can be registered as Spring beans via component scanning. @RestControllerAdvice is also a meta-annotation marked with both @ControllerAdvice and @ResponseBody which essentially means @ExceptionHandler methods are rendered to the response body via message conversion (vs view resolution/template rendering).

@ControllerAdvice标记为@Component，这意味着这些类可以通过组件扫描注册为Spring bean，

@RestControllerAdvice也是一个用@ControllerAdvice 和@ResponseBody标记的元注解， 它本质上意味着@ExceptionHandler方法通过消息转换器（vs视图解析和模板渲染）呈现给响应体

On startup, the infrastructure classes for @RequestMapping and @ExceptionHandler methods detect Spring beans of type @ControllerAdvice, and then apply their methods at runtime. Global @ExceptionHandler methods (from an @ControllerAdvice) are applied after local ones (from the @Controller). By contrast global @ModelAttribute and @InitBinder methods are applied before local ones.

启动时， @RequestMapping和@ExceptionHandler方法的基础结构类检测类型为 @ControllerAdvice的Spring bean，然后再运行时应用它们的方法，全局的@ExceptionHandler方法（来自@ControllerAdvice）应用于本地方法（来自@Controller）之后，相比之下，全局的 @ModelAttribute 和@InitBinder放在应用在本地方法之前

By default @ControllerAdvice methods apply to every request, i.e. all controllers, but you can narrow that down to a subset of controllers via attributes on the annotation:

默认情况下，@ControllerAdvice方法应用于每个请求（例如所有的控制器），但是可以通过注解上的属性缩小到控制器的一个子集

*// Target all Controllers annotated with @RestController*

@ControllerAdvice(annotations = RestController.class)

**public** **class** **ExampleAdvice1** {}

*// Target all Controllers within specific packages*

@ControllerAdvice("org.example.controllers")

**public** **class** **ExampleAdvice2** {}

*// Target all Controllers assignable to specific classes*

@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})

**public** **class** **ExampleAdvice3** {}

Keep in mind the above selectors are evaluated at runtime and may negatively impact performance if used extensively.

请注意，上述选择器在运行时进行评估，如果广泛使用，可能对性能有负面影响

## 1.5. URI Links

Spring MVC provides a mechanism for building and encoding a URI using UriComponentsBuilder and UriComponents.

Spring MVC提供了使用UriComponentsBuilder 和UriComponents构造和编码URI的一个机制

For example you can expand and encode a URI template string:

举个例子，你可以expand 和encode 一个URI模板字符串

UriComponents uriComponents = UriComponentsBuilder.fromUriString(

"http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();

Note that UriComponents is immutable and the expand() and encode() operations return new instances if necessary.

You can also expand and encode using individual URI components:

请注意，UriComponents 是不可变的，如果需要，expand() 和encode()操作返回新的实例

你还可以使用单独的URI组件进行expand 和encode

UriComponents uriComponents = UriComponentsBuilder.newInstance()

.scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}").build()

.expand("42", "21")

.encode();

In a Servlet environment the ServletUriComponentsBuilder subclass provides static factory methods to copy available URL information from a Servlet requests:

在Servlet环境中，ServletUriComponentsBuilder 子类提供了静态工厂方法用来从一个Servlet请求复制可用的URL信息

HttpServletRequest request = ...

*// Re-use host, scheme, port, path and query string*

*// Replace the "accountId" query param*

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)

.replaceQueryParam("accountId", "{id}").build()

.expand("123")

.encode();

Alternatively, you may choose to copy a subset of the available information up to and including the context path:

或者，你可以选择将可用信息的子集复制并包含在上下文路径中

*// Re-use host, port and context path*

*// Append "/accounts" to the path*

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)

.path("/accounts").build()

Or in cases where the DispatcherServlet is mapped by name (e.g. /main/\*), you can also have the literal part of the servlet mapping included:

或者在DispatcherServlet 通过名称（例如/ main / \*）映射的情况下，你也可以包含Servlet映射的文字部分

*// Re-use host, port, context path*

*// Append the literal part of the servlet mapping to the path*

*// Append "/accounts" to the path*

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(request)

.path("/accounts").build()

### 1.5.1. Links to Controllers

Spring MVC also provides a mechanism for building links to controller methods. For example, given:

SpringMVC也提供了构建控制器方法链接的一种机制，举个例子，给出：

@Controller@RequestMapping("/hotels/{hotel}")

**public** **class** **BookingController** {

@GetMapping("/bookings/{booking}")

**public** String getBooking(@PathVariable Long booking) {

*// ...*

}

}

You can prepare a link by referring to the method by name:

你可以通过名称引用方法来准备链接

UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

In the above example we provided actual method argument values, in this case the long value 21, to be used as a path variable and inserted into the URL. Furthermore, we provided the value 42 in order to fill in any remaining URI variables such as the "hotel" variable inherited from the type-level request mapping. If the method had more arguments you can supply null for arguments not needed for the URL. In general only @PathVariable and @RequestParam arguments are relevant for constructing the URL.

在上述的例子中，我们提供了实际方法参数的值，在这种情况下，long值21被用作路径变量插入到URL，此外，我们还提供值42，以填充任意剩余的URI变量，例如继承自类型级别请求映射的“hotel”变量，如果方法有更多的参数，可以给URL不需要的参数提供null值，通常，仅仅 @PathVariable 和@RequestParam参数和构建URL有关联

There are additional ways to use MvcUriComponentsBuilder. For example you can use a technique akin to mock testing through proxies to avoid referring to the controller method by name (the example assumes static import of MvcUriComponentsBuilder.on):

还有其他的方法可以使用MvcUriComponentsBuilder，举个例子，你可以使用类似于通过代理模拟测试的技术，以避免按名称引用控制器（这个示例假定静态导入MvcUriComponentsBuilder）

UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

The above examples use static methods in MvcUriComponentsBuilder. Internally they rely on ServletUriComponentsBuilder to prepare a base URL from the scheme, host, port, context path and servlet path of the current request. This works well in most cases, however sometimes it may be insufficient. For example you may be outside the context of a request (e.g. a batch process that prepares links) or perhaps you need to insert a path prefix (e.g. a locale prefix that was removed from the request path and needs to be re-inserted into links).

上述示例在MvcUriComponentsBuilder中使用静态方法，在内部，它们依靠ServletUriComponentsBuilder 来从当前请求的scheme、主机、端口、上下文路径和Servlet路径中准备一个基本的URL，这在大部分情况下运作良好，但是有时可能不足，举个例子，你也许在请求的上下文之外（例如准备链接的批量处理过程），或者你要插入一个路径前缀（例如一个从请求路径中移除并需要重新插入到链接中的区域前缀 ）

For such cases you can use the static "fromXxx" overloaded methods that accept a UriComponentsBuilder to use base URL. Or you can create an instance of MvcUriComponentsBuilder with a base URL and then use the instance-based "withXxx" methods. For example:

在这种情况下，你可以使用接受一个UriComponentsBuilder 的静态“fromXxx”重载方法来使用基本的URL，或者你可以创建一个带有基本URL的MvcUriComponentsBuilder 实例，然后使用基于实例的"withXxx" 方法，例如：

UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path("/en");

MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);

builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

### 1.5.2. Links in views

You can also build links to annotated controllers from views such as JSP, Thymeleaf, FreeMarker. This can be done using the fromMappingName method in MvcUriComponentsBuilder which refers to mappings by name.

你也可以从JSP,Thymeleaf,FreeMarker视图中建立到注解控制器的链接。这可以使用MvcUriComponentsBuilder 中的fromMappingName 方法来完成，它引用了名称的映射

Every @RequestMapping is assigned a default name based on the capital letters of the class and the full method name. For example, the method getFoo in class FooController is assigned the name "FC#getFoo". This strategy can be replaced or customized by creating an instance of HandlerMethodMappingNamingStrategy and plugging it into your RequestMappingHandlerMapping. The default strategy implementation also looks at the name attribute on @RequestMapping and uses that if present. That means if the default mapping name assigned conflicts with another (e.g. overloaded methods) you can assign a name explicitly on the @RequestMapping.

每个@RequestMapping被分配了一个基于类的大写字母和完整方法名的默认名称，例如，在类FooController的方法getFoo被分配名称“FC#getFoo”，这个策略可以通过创建一个HandlerMethodMappingNamingStrategy 的实例并插入到RequestMappingHandlerMapping来呗替换或定制。默认的策略实现也查看@RequestMapping上的名称属性并使用它（如果存在）。这意味着如果分配的默认路径名称和另一个冲突（例如重载的方法），你可以在@RequestMapping显式的指派一个名称

The Spring JSP tag library provides a function called mvcUrl that can be used to prepare links to controller methods based on this mechanism.

Spring JSP标签库提供了一个叫mvcUrl的方法，它可用于基于此机制准备控制器方法的链接

For example given:

例如，给定：

@RequestMapping("/people/{id}/addresses")

**public** **class** **PersonAddressController** {

@RequestMapping("/{country}")

**public** HttpEntity getAddress(@PathVariable String country) { ... }

}

You can prepare a link from a JSP as follows:

你可以从JSP准备一个链接，如下所示：

<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>

...

<a href="${s:mvcUrl('PAC#getAddress').arg(0,'US').buildAndExpand('123')}">Get Address</a>

## 1.6. Exception Handling

### 1.6.1. Overview

Spring HandlerExceptionResolver implementations deal with unexpected exceptions that occur during controller execution. A HandlerExceptionResolver somewhat resembles the exception mappings you can define in the web application descriptor web.xml. However, they provide a more flexible way to do so. For example they provide information about which handler was executing when the exception was thrown. Furthermore, a programmatic way of handling exceptions gives you more options for responding appropriately before the request is forwarded to another URL (the same end result as when you use the Servlet specific exception mappings).

Spring HandlerExceptionResolver 实现类处理在控制器执行期间发生的意外异常，HandlerExceptionResolver 有点类似于你可以在web应用描述web.xml中定义的异常映射，但是，它们提供了一个更加灵活的方式来做这些，例如，它们提供有关当异常抛出时正在执行哪个处理器的信息，另外，处理异常的编程方式给你更多的选择，以便在请求转发到另一个URL（与使用Servlet指定的异常映射有相同的最终结果）之前进行适当的相应

Besides implementing the HandlerExceptionResolver interface, which is only a matter of implementing the resolveException(Exception, Handler) method and returning a ModelAndView, you may also use the provided SimpleMappingExceptionResolver or create @ExceptionHandler methods. The SimpleMappingExceptionResolver enables you to take the class name of any exception that might be thrown and map it to a view name. This is functionally equivalent to the exception mapping feature from the Servlet API, but it is also possible to implement more finely grained mappings of exceptions from different handlers. The @ExceptionHandler annotation on the other hand can be used on methods that should be invoked to handle an exception. Such methods may be defined locally within an @Controller or may apply to many @Controller classes when defined within an @ControllerAdvice class. The following sections explain this in more detail.

除了实现HandlerExceptionResolver 接口，它仅仅是一种实现resolveException（Exception, Handler）方法并返回ModelAndView，你也可以使用提供的SimpleMappingExceptionResolver 或创建@ExceptionHandler方法，

SimpleMappingExceptionResolver 使您可以将可能引发的任何异常的类名称映射到视图名称，这在功能上等同于Servlet API的异常映射特性，但是也可以实现来自不同处理程序的更精细的异常映射。另一方面，@ExceptionHandler注解能够用于调用处理异常的方法，这些方法可以在@Controller中本地定义，或当定义在@ControllerAdvice类时应用到许多@Controlle类，下面的部分将对此进行更详细的解释

### 1.6.2. @ExceptionHandler

The HandlerExceptionResolver interface and the SimpleMappingExceptionResolver implementations allow you to map Exceptions to specific views declaratively along with some optional Java logic before forwarding to those views. However, in some cases, especially when relying on @ResponseBody methods rather than on view resolution, it may be more convenient to directly set the status of the response and optionally write error content to the body of the response.

HandlerExceptionResolver 接口和SimpleMappingExceptionResolver 实现允许你在转发这些视图之前，将异常连同一些可选择的java逻辑声明性地映射到特定的视图，但是，在某些情况下，特别是当依靠@ResponseBody方法而不是视图解析器时，直接的设置响应的状态并选择性的写错误内容到响应体是更加方便的

You can do that with @ExceptionHandler methods. When declared within a controller such methods apply to exceptions raised by @RequestMapping methods of that controller (or any of its subclasses). You can also declare an @ExceptionHandler method within an @ControllerAdvice class in which case it handles exceptions from @RequestMapping methods from many controllers. Below is an example of a controller-local @ExceptionHandler method:

你可以用 @ExceptionHandler 方法做到这一点，当在一个控制器中声明时，这些方法适用于该控制器（或任何它的子类）的@RequestMapping方法引起的异常，你也可以在@ControllerAdvice中声明@ExceptionHandler方法，这种情况下，它处理来自多个控制器的@RequestMapping 方法的异常，下面是一个本地控制器@ExceptionHandler 方法的例子

@Controller

**public** **class** **SimpleController** {

*// @RequestMapping methods omitted ...*

@ExceptionHandler(IOException.class)

**public** ResponseEntity<String> handleIOException(IOException ex) {

*// prepare responseEntity*

**return** responseEntity;

}

}

The @ExceptionHandler value can be set to an array of Exception types. If an exception is thrown that matches one of the types in the list, then the method annotated with the matching @ExceptionHandler will be invoked. If the annotation value is not set then the exception types listed as method arguments are used.

@ExceptionHandler值能被设置成异常类型的数组，如果抛出一个异常与列表类型中某个类型相匹配，那么，将调用和@ExceptionHandler匹配的注解方法，如果注解值没有设置，那么，将使用列为方法参数的异常类型

Much like standard controller methods annotated with a @RequestMapping annotation, the method arguments and return values of @ExceptionHandler methods can be flexible. For example, the HttpServletRequest can be accessed in Servlet environments. The return type can be a String, which is interpreted as a view name, a ModelAndView object, a ResponseEntity, or you can also add the @ResponseBody to have the method return value converted with message converters and written to the response stream.

就像使用@RequestMapping注解的标准控制器方法注解，@ExceptionHandler方法的方法参数和返回值可能是灵活的，举个例子可以再Servlet环境中访问HttpServletRequest ，返回类型可能是一个字符，它被解释为一个视图名称，一个ModelAndView对象，一个ResponseEntity，或者，你也可以添加@ResponseBody，以使方法返回值转换为消息转换器并写入到响应流

### 1.6.3. Framework exceptions

Spring MVC may raise a number of exceptions while processing a request. The SimpleMappingExceptionResolver can easily map any exception to a default error view as needed. However, when working with clients that interpret responses in an automated way you will want to set specific status code on the response. Depending on the exception raised the status code may indicate a client error (4xx) or a server error (5xx).

在处理请求时，SpringMVC可能会引发一些的异常，SimpleMappingExceptionResolver 可以根据需要容易的将任何异常映射到它们默认异常视图，但是，在与自动解释响应的客户端合作时，你需要在响应中设置特定的状态码，根据引发的异常，状态码可以表示客户端错误（4xx）或服务器错误（5xx）

The DefaultHandlerExceptionResolver translates Spring MVC exceptions to specific error status codes. It is registered by default with the MVC namespace, the MVC Java config, and also by the DispatcherServlet (i.e. when not using the MVC namespace or Java config). Listed below are some of the exceptions handled by this resolver and the corresponding status codes:

DefaultHandlerExceptionResolver 将Spring MVC异常转换为特定的错误状态码，默认情况下，MVC命名空间，MVC java配置以及DispatcherServlet （即不使用MVC命名空间或Java配置时）都被注册，下面列出的是这个解析器处理的一些异常和对应的状态码

| Exception | HTTP Status Code |
| --- | --- |
| BindException | 400 (Bad Request) |
| ConversionNotSupportedException | 500 (Internal Server Error) |
| HttpMediaTypeNotAcceptableException | 406 (Not Acceptable) |
| HttpMediaTypeNotSupportedException | 415 (Unsupported Media Type) |
| HttpMessageNotReadableException | 400 (Bad Request) |
| HttpMessageNotWritableException | 500 (Internal Server Error) |
| HttpRequestMethodNotSupportedException | 405 (Method Not Allowed) |
| MethodArgumentNotValidException | 400 (Bad Request) |
| MissingPathVariableException | 500 (Internal Server Error) |
| MissingServletRequestParameterException | 400 (Bad Request) |
| MissingServletRequestPartException | 400 (Bad Request) |
| NoHandlerFoundException | 404 (Not Found) |
| NoSuchRequestHandlingMethodException | 404 (Not Found) |
| TypeMismatchException | 400 (Bad Request) |

The DefaultHandlerExceptionResolver works transparently by setting the status of the response. However, it stops short of writing any error content to the body of the response while your application may need to add developer-friendly content to every error response for example when providing a REST API. You can prepare a ModelAndView and render error content through view resolution — i.e. by configuring a ContentNegotiatingViewResolver, MappingJackson2JsonView, and so on. However, you may prefer to use @ExceptionHandler methods instead.

DefaultHandlerExceptionResolver 通过设置状态响应码来透明的工作，但是，当你的应用需要添加开发友好的内容到每个错误响应时（例如，当提供一个REST API是），它不在将任何错误内容写入到响应体，你可以准备一个ModelAndView 并通过视图解析器来渲染错误内容——即通过配置ContentNegotiatingViewResolver，MappingJackson2JsonView等，然而，你可能更喜欢使用@ExceptionHandler方法代替

If you prefer to write error content via @ExceptionHandler methods you can extend ResponseEntityExceptionHandler instead. This is a convenient base for @ControllerAdvice classes providing an @ExceptionHandler method to handle standard Spring MVC exceptions and return ResponseEntity. That allows you to customize the response and write error content with message converters.

如果你更喜欢通过@ExceptionHandler方法编写错误内容，则可以扩展ResponseEntityExceptionHandler 代替。这是@ControllerAdvice类的便利基础，它提供一个@ExceptionHandler 方法来处理标准Spring MVC异常并返回ResponseEntity。它允许你自定义响应并使用消息转换器编写错误内容

### 1.6.4. REST API exceptions

An @RestController may use @ExceptionHandler methods that return a ResponseEntity to provide both a response status and error details in the body of the response. Such methods may also be added to @ControllerAdvice classes for exception handling across a subset or all controllers.

@RestController 可以使用@ExceptionHandler方法返回一个ResponseEntity ，在响应体中提供响应状态码和错误细节，这个方法也可以添加到@ControllerAdvice类中，以处理子集或者所有控制器的异常处理

A common requirement is to include error details in the body of the response. Spring does not automatically do this (although Spring Boot does) because the representation of error details in the response body is application specific.

一个常见的需求是在响应体中包含错误信息，Spring没有自动执行此操作（虽然Spring boot会这样做），因为响应体中的错误细节的表现心事是特定于应用程序的

Applications that wish to implement a global exception handling strategy with error details in the response body should consider extending the abstract base class ResponseEntityExceptionHandler which provides handling for the exceptions that Spring MVC raises and provides hooks to customize the response body as well as to handle other exceptions. Simply declare the extension class as a Spring bean and annotate it with @ControllerAdvice.

希望在响应体中实现全局异常处理策略的应用程序应该考虑扩展抽象基类ResponseEntityExceptionHandler ，它提供对Spring MVC引发的异常的处理，并提供钩子来自定义响应主体以及处理其他异常。只需要将扩展类声明为Spring bean，并使用@ControllerAdvice对其注解

### 1.6.5. Annotated Exception

A business exception can be annotated with @ResponseStatus. When the exception is raised, the ResponseStatusExceptionResolver handles it by setting the status of the response accordingly. By default the DispatcherServlet registers the ResponseStatusExceptionResolver and it is available for use.

业务异常可以用@ResponseStatus进行注解，当引发异常时，ResponseStatusExceptionResolver 通过设置相应地响应状态来处理它。默认情况下，DispatcherServlet 注册ResponseStatusExceptionResolver 并可供使用

### 1.6.6. Container error page

When the status of the response is set to an error status code and the body of the response is empty, Servlet containers commonly render an HTML formatted error page. To customize the default error page of the container, you can declare an <error-page> element in web.xml. Up until Servlet 3, that element had to be mapped to a specific status code or exception type. Starting with Servlet 3 an error page does not need to be mapped, which effectively means the specified location customizes the default Servlet container error page.

当响应状态被设置为错误状态码且响应体为空时，Servlet 容器通常呈现一个HTML格式的错误页面。要自定义容器的默认错误页面，可以在web.xml中声明一个<error-page>元素，直到Servlet 3，该元素必须映射到特定的状态码或异常类型，从Servlet 3开始，不需要映射错误页面，这实际上意味着指定的位置定制了默认的Servlet容器错误页面

<error-page>

<location>/error</location></error-page>

Note that the actual location for the error page can be a JSP page or some other URL within the container including one handled through an @Controller method:

请注意，错误页面实际的位置可能是一个JSP页面或容器中的其他URL，包括通过@Controller 方法处理的URL

When writing error information, the status code and the error message set on the HttpServletResponse can be accessed through request attributes in a controller:

当写错误信息时，可以通过控制器中请求属性访问HttpServletResponse 上设置的状态码和错误消息

@Controller

**public** **class** **ErrorController** {

@RequestMapping(path = "/error", produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE)

@ResponseBody

**public** Map<String, Object> handle(HttpServletRequest request) {

Map<String, Object> map = **new** HashMap<String, Object>();

map.put("status", request.getAttribute("javax.servlet.error.status\_code"));

map.put("reason", request.getAttribute("javax.servlet.error.message"));

**return** map;

}

}

or in a JSP:

或在JSP中

<%@ page contentType="application/json" pageEncoding="UTF-8"%>

{

status:<%=request.getAttribute("javax.servlet.error.status\_code") %>,

reason:<%=request.getAttribute("javax.servlet.error.message") %>

}

## 1.7. Async Requests

Spring MVC has an extensive integration with the Servlet 3.0 asynchronous request processing. DeferredResult and Callableprovide basic support for producing return values asynchronously. Controllers can produce response streams including SSE and raw data. Controllers can use reactive clients and return reactive return types to Spring MVC for response handling.

Spring MVC和Servlet 3.0 异步请求处理进行了广泛的集成，DeferredResult 和Callableprovide为异步生成返回值提供了基本支持。控制器能够生成包含SSE和raw data的响应流。控制器可以用响应式的客户端，并将响应式返回类型返回给Spring MVC以进行响应处理。

### 1.7.1. DeferredResult

Once the asynchronous request processing feature is enabled in the Servlet container, controller methods can wrap any supported controller method return value with DeferredResult:

一旦在Servlet容器中启用异步请求处理功能，控制器方法就能使用DeferredResult包装任何受支持的控制器方法返回值

@GetMapping("/quotes")

@ResponseBody

**public** DeferredResult<String> quotes() {

DeferredResult<String> deferredResult = **new** DeferredResult<String>();

*// Save the deferredResult somewhere..*

**return** deferredResult;

}

*// From some other thread...*

deferredResult.setResult(data);

The controller can produce the return value asynchronously, from a different thread, for example in response to an external event (JMS message), a scheduled task, or other.

控制器可以从不同的线程异步地生成返回值，例如，响应外部事件（JMS消息），定时调度任务或其他

### 1.7.2. Callable

A controller may also wrap any supported return value with java.util.concurrent.Callable:

控制器也能够用java.util.concurrent.Callable包装任何受支持的返回类型

@PostMapping

**public** Callable<String> processUpload(**final** MultipartFile file) {

**return** **new** Callable<String>() {

**public** String call() **throws** Exception {

*// ...*

**return** "someView";

}

};

}

The return value will then be obtained by executing the the given task through the configured TaskExecutor.

然后通过配置的TaskExecutor执行给定的任务来获取返回值

### 1.7.3. Processing

Here is a very concise overview of Servlet asynchronous request processing:

下面是Servlet异步请求处理的简要概述

A ServletRequest can be put in asynchronous mode by calling request.startAsync(). The main effect of doing so is that the Servlet, as well as any Filters, can exit but the response will remain open to allow processing to complete later.

ServletRequest 可以通过调用request.startAsync()置于异步模式。这样做的主要效果是，Servlet以及任何过滤器能够退出，但是响应仍旧保持打开状态，以便稍后完成处理

The call to request.startAsync() returns AsyncContext which can be used for further control over async processing. For example it provides the method dispatch, that is similar to a forward from the Servlet API except it allows an application to resume request processing on a Servlet container thread.

 request.startAsync()调用的返回值AsyncContext ，可用于进一步控制异步处理。例如，它提供了方法dispatch，它类似于Servlet API的转发，除了它允许应用在一个Servlet容器线程上程序恢复请求处理

The ServletRequest provides access to the current DispatcherType that can be used to distinguish between processing the initial request, an async dispatch, a forward, and other dispatcher types.

ServletRequest 提供了对当前DispatcherType 类型的访问，可用于区分处理初始请求，异步调度，转发和其他调度类型

#### DeferredResult processing:

Controller returns a DeferredResult and saves it in some in-memory queue or list where it can be accessed.

控制器返回一个DeferredResult 并保存到它能够访问的内存队列或列表中

Spring MVC calls request.startAsync().

Spring MVC 调用request.startAsync()

Meanwhile the DispatcherServlet and all configured Filter’s exit the request processing thread but the response remains open.

与此同时，DispatcherServlet 和所有配置的过滤器退出请求流程线程，但是响应仍旧保持打开

The application sets the DeferredResult from some thread and Spring MVC dispatches the request back to the Servlet container.

应用程序从某个线程设置DeferredResult ， Spring MVC将请求分配回Servlet 容器

The DispatcherServlet is invoked again and processing resumes with the asynchronously produced return value.

DispatcherServlet 再次被调用，并使用异步生成的返回值继续处理

#### Callable processing:

Controller returns a Callable.

控制器返回一个Callable

Spring MVC calls request.startAsync() and submits the Callable to a TaskExecutor for processing in a separate thread.

Spring MVC调用 request.startAsync() 并将Callable 提交到TaskExecutor ，以便在单独的线程中进行处理

Meanwhile the DispatcherServlet and all Filter’s exit the Servlet container thread but the response remains open.

与此同时，DispatcherServlet 和所有的过滤器退出Servlet容器线程，但响应保存打开

Eventually the Callable produces a result and Spring MVC dispatches the request back to the Servlet container to complete processing.

最终，Callable 生成一个结果，SpringMVC将请求分配回Servlet容器，以完成处理

The DispatcherServlet is invoked again and processing resumes with the asynchronously produced return value from the Callable.

DispatcherServlet 被再次调用，并使用从Callable异步生成的返回结果继续处理

### 1.7.4. Exception handling

When using a DeferredResult you can choose whether to call setResult or setErrorResult with an exception. In both cases Spring MVC dispatches the request back to the Servlet container to complete processing. It is then treated either as if the controller method returned the given value, or as if it produced the given exception. The exception then goes through the regular exception handling mechanism, e.g. invoking @ExceptionHandler methods.

在使用DeferredResult 时，你可以选择是否使用异常调用setResult或setErrorResult，在这两种情况下，Spring MVC会将请求发送回Servlet容器来完成处理，然后它将被视为控制器方法返回的给定值，或看做它产生的给定异常，接着，异常通过常规的异常处理机制，例如调用 @ExceptionHandler方法

When using Callable, similar processing logic follows. The main difference being that the result is returned from the Callable or an exception is raised by it.

当使用Callable是，类似的处理逻辑如下。主要的不同是在于返回结果是来自Callable 还是由它引发的异常

### 1.7.5. Interception

HandlerInterceptor's can also be AsyncHandlerInterceptor in order to receive the afterConcurrentHandlingStarted callback on the initial request that starts asynchronous processing instead of postHandle and afterCompletion.

HandlerInterceptor也可以是AsyncHandlerInterceptor ，为了在初始化请求上接收afterConcurrentHandlingStarted 回调，该回调开启异步处理而不是postHandle或afterCompletion

HandlerInterceptor's can also register a CallableProcessingInterceptor or a DeferredResultProcessingInterceptor in order to integrate more deeply with the lifecycle of an asynchronous request for example to handle a timeout event.

HandlerInterceptor也可以注册CallableProcessingInterceptor 或DeferredResultProcessingInterceptor ，以便更深入的集成异步请求的生命周期，例如，处理超时事件

DeferredResult provides onTimeout(Runnable) and onCompletion(Runnable) callbacks. Callable can be substituted for WebAsyncTask that exposes additional methods for timeout and completion callbacks.

DeferredResult 提供了onTimeout(Runnable) 和onCompletion(Runnable)，WebAsyncTask 可以取代Callable ，它公开了用于超时和完成回调的额外方法

### 1.7.6. Streaming response

What if you wanted to push multiple events on a single HTTP response? The ResponseBodyEmitter return value can be used to stream multiple Objects, where each Object sent is serialized with an HttpMessageConverter and written to the response. For example:

如果你想在单个HTTP响应上推送多个事件，该怎么办？ResponseBodyEmitter 返回值能够用于一连串的多个对象，其中每个对象的发送都用HttpMessageConverter 序列号并写入到响应。例如

@GetMapping("/events")

**public** ResponseBodyEmitter handle() {

ResponseBodyEmitter emitter = **new** ResponseBodyEmitter();

*// Save the emitter somewhere..*

**return** emitter;

}

*// In some other thread*

emitter.send("Hello once");

*// and again later on*

emitter.send("Hello again");

*// and done at some point*

emitter.complete();

ResponseBodyEmitter can also be used as the body in a ResponseEntity allowing you to customize the status and headers of the response.

ResponseBodyEmitter 也能被用作ResponseBodyEmitter 中主体，它允许你自定义响应的状态和头

### 1.7.7. Server-Sent Events

SseEmitter is a sub-class of ResponseBodyEmitter that provides support for Server-Sent Events where events sent from the server are formatted according to the W3C SSE specification. In order to produce an SSE stream from a controller simply return SseEmitter:

SseEmitter 是ResponseBodyEmitter 的一个子类，它为Server-Sent事件提供支持，其中从服务器发送的事件根据 W3C SSE 规范进行格式化，为了从控制器生成SSE流，只需返回SseEmitter

@GetMapping(path="/events", produces=MediaType.TEXT\_EVENT\_STREAM\_VALUE)

**public** SseEmitter handle() {

SseEmitter emitter = **new** SseEmitter();

*// Save the emitter somewhere..*

**return** emitter;

}

*// In some other thread*

emitter.send("Hello once");

*// and again later on*

emitter.send("Hello again");

*// and done at some point*

emitter.complete();

While SSE is the main option for streaming into browsers, note that Internet Explorer does not support Server-Sent Events. Consider using Spring’s WebSocket messaging with SockJS fallback transports (including SSE) that target a wide range of browsers.

虽然SSE值流入浏览器的主要选择，请注意，IE浏览器不支持Server-Sent事件，建议使用Spring的WebSocket 消息和SockJS 回滚传输（包括SSE），它针对广泛的浏览器

### 1.7.8. Streaming raw data

Sometimes it is useful to bypass message conversion and stream directly to the response OutputStream for example for a file download. Use the of the StreamingResponseBody return value type to do that:

有时，绕过消息转换并直接流到响应的OutputStream（例如文件下载）是有用的，用StreamingResponseBody 返回值类型执行此操作

@GetMapping("/download")

**public** StreamingResponseBody handle() {

**return** **new** StreamingResponseBody() {

@Override

**public** **void** writeTo(OutputStream outputStream) **throws** IOException {

*// write...*

}

};

}

StreamingResponseBody can be used as the body in a ResponseEntity allowing you to customize the status and headers of the response.

StreamingResponseBody 可以用作ResponseEntity 的主体，允许你自定义响应的状态和头

### 1.7.9. Reactive return values

Spring MVC supports use of reactive client libraries in a controller. This includes the WebClient from spring-webflux and others such as Spring Data reactive data repositories. In such scenarios it is convenient to be able to return reactive types from the controller method .

Spring MVC支持在控制器中使用响应式客户端库，这包括 spring-webflux中的WebClient 和诸如Spring Data响应式数据存储等其他WebClient

Reactive return values are handled as follows:

响应式返回值的处理如下：

* A single-value promise is adapted to and similar to using DeferredResult. Examples include Mono (Reactor) or Single (RxJava).

单值承诺适用于并类似于使用DeferredResult。例如包括Mono (Reactor) 或Single (RxJava).

* A multi-value stream, with a streaming media type such as "application/stream+json" or "text/event-stream", is adapted to and similar to using ResponseBodyEmitter or SseEmitter. Examples include Flux (Reactor) or Observable (RxJava). Applications can also return Flux<ServerSentEvent> or Observable<ServerSentEvent>.

具有流媒体类型（例如"application/stream+json" 或 "text/event-stream"）的多值流，适用于并类似于使用ResponseBodyEmitter 或SseEmitter，例如包括Flux (Reactor) 或Observable (RxJava)。应用程序也可以返回Flux<ServerSentEvent> 或Observable<ServerSentEvent>.

* A multi-value stream, with any other media type (e.g. "application/json"), is adapted to and similar to using DeferredResult<List<?>>.

具有任何其他媒体类型（例如"application/json"）的多值流，适用于并类似于使用 DeferredResult<List<?>>

Spring MVC supports Reactor and RxJava through the ReactiveAdapterRegistry from spring-core which allows it to adapt from multiple reactive libraries.

Spring MVC通过 spring-core中的ReactiveAdapterRegistry 来支持Reactor 和RxJava，使他能够适应多个响应式库

When streaming to the response with a reactive type, Spring MVC performs (blocking) writes to the response through the through the configured MVC TaskExecutor. By default this is a SyncTaskExecutor and not suitable for production. SPR-16203 will provide better defaults. In the mean time please configure the executor through the MVC config.

当流式传输到具有响应式类型的响应时，SpringMVC通过通过配置的MVCTaskExecutor来执行（阻塞）写入到响应。默认情况下，这是一个SyncTaskExecutor ，不适合生产。SPR-16203 提供更好的默认值。同时请通过MVC配置来配置executor

### 1.7.10. Configuration

For asynchronous requests there are minor requirements at the Servlet container level and more controls in Spring MVC configuration.

对于异步请求，在Servlet容器级别有更小的要求，在SpringMVC配置中有更多的控制

#### Servlet container config

For applications configured with a web.xml be sure to update to version 3.0:

对于使用web.xml配置的应用程序，请确保更新到3.0版

<web-app 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"

version="3.0">

</web-app>

Asynchronous support must be enabled on the DispatcherServlet through the <async-supported>true</async-supported> sub-element in web.xml. Additionally any Filter that participates in asyncrequest processing must be configured to support the ASYNC dispatcher type. It should be safe to enable the ASYNC dispatcher type for all filters provided with the Spring Framework since they usually extend OncePerRequestFilter and that has runtime checks for whether the filter needs to be involved in async dispatches or not.

必须在DispatcherServlet 通过web.xml中的 <async-supported>true</async-supported>子元素来启用异步支持。另外，任何参与异步请求处理的过滤器都必须配置为支持ASYNC分发类型。

为 Spring Framework 提供的所有功率去开启ASYNC 分发类型应该是安全的。因为它通常扩展OncePerRequestFilter ，并运行时检查过滤器是否需要参与到异步分派中

Below is some example web.xml configuration:

以下是web.xml配置的一些列子：

<web-app 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"

version="3.0">

<filter>

<filter-name>Spring OpenEntityManagerInViewFilter</filter-name>

<filter-class>org.springframework.~.OpenEntityManagerInViewFilter</filter-class>

<async-supported>true</async-supported>

</filter>

<filter-mapping>

<filter-name>Spring OpenEntityManagerInViewFilter</filter-name>

<url-pattern>/\*</url-pattern>

<dispatcher>REQUEST</dispatcher>

<dispatcher>ASYNC</dispatcher>

</filter-mapping>

</web-app>

If using Servlet 3, Java based configuration for example via WebApplicationInitializer, you’ll also need to set the "asyncSupported" flag as well as the ASYNC dispatcher type just like with web.xml. To simplify all this configuration, consider extending AbstractDispatcherServletInitializer, or better AbstractAnnotationConfigDispatcherServletInitializer which automatically set those options and make it very easy to register Filter instances.

如果使用Servlet 3，Java基本配置（例如通过WebApplicationInitializer），你还需要设置“asyncSupported”标签和ASYNC 分发类型，就像web.xml一样。为了简化所有的配置，建议扩展AbstractDispatcherServletInitializer或更好的AbstractAnnotationConfigDispatcherServletInitializer ，它可以自动设置这些选项并且可以容易的注册过滤器实例

#### Spring MVC config

The MVC Java config and the MVC namespace provide options for configuring asynchronous request processing. WebMvcConfigurer has the method configureAsyncSupport while <mvc:annotation-driven> has an <async-support> sub-element.

MVC Java配置和MVC命名空间为配置异步请求处理提供了选择。WebMvcConfigurer具有configureAsyncSupport 方法，而<mvc:annotation-driven>具有 <async-support>子元素

Those allow you to configure the default timeout value to use for async requests, which if not set depends on the underlying Servlet container (e.g. 10 seconds on Tomcat). You can also configure an AsyncTaskExecutor to use for executing Callableinstances returned from controller methods. It is highly recommended to configure this property since by default Spring MVC uses SimpleAsyncTaskExecutor. The MVC Java config and the MVC namespace also allow you to register CallableProcessingInterceptor and DeferredResultProcessingInterceptor instances.

这些允许你为异步请求配置默认超时值，如果没有设置，则依赖于底层的Servlet容器。你也可以配置AsyncTaskExecutor 用来执行从控制器方法返回的Callableinstances。强烈建议配置这个属性，因为默认情况下Spring MVC使用SimpleAsyncTaskExecutor，MVC Java配置和MVC命名空间也允许你注册CallableProcessingInterceptor 和DeferredResultProcessingInterceptor实例

If you need to override the default timeout value for a specific DeferredResult, you can do so by using the appropriate class constructor. Similarly, for a Callable, you can wrap it in a WebAsyncTask and use the appropriate class constructor to customize the timeout value. The class constructor of WebAsyncTask also allows providing an AsyncTaskExecutor.

如果你需要为特定的DeferredResult覆盖这个默认的超时值，你可以通过使用适当的类构造器来实现，相似的，对于Callable，你能包装它到WebAsyncTask 中，并使用适当的类构造器来自定义超时值。WebAsyncTask 类的构造器也允许提供一个AsyncTaskExecutor

## 1.8. CORS

### 1.8.1. Introduction

For security reasons browsers prohibit AJAX calls to resources outside the current origin. For example you could have your bank account in one tab and evil.com in another. Scripts from evil.com should not be able to make AJAX requests to your bank API with your credentials, e.g. withdrawing money from your account!

出于安全原因，浏览器禁止AJAX调用当前源之外的资源，举个例子，你可能在一个选项卡中有你的银行账号，在另一个标签中有evil.com。来自evil.com的脚本不能用你的凭证向你的你的银行API发送AJAX请求，例如，从你账号提款

Cross-Origin Resource Sharing (CORS) is a W3C specification implemented by most browsers that allows you to specify what kind of cross domain requests are authorized rather than using less secure and less powerful workarounds based on IFRAME or JSONP.

跨域资源共享是由大部分浏览器实现的W3C规范，允许你指定什么类型的跨域请求被授权，而不是使用基于IFRAME或JSONP的不太安全和功能较差的解决方案

### 1.8.2. Processing

The CORS specification distinguishes between preflight, simple, and actual requests.

CORS规范区分预检、简单的和实际的请求

Spring MVC HandlerMapping's provide built-in support for CORS. After successfully mapping a request to a handler, HandlerMapping's check the CORS configuration for the given request and handler and take further actions. Preflight requests are handled directly while simple and actual CORS requests are intercepted, validated, and have required CORS response headers set.

Spring MVC HandlerMapping提供了对CORS的内置支持，成功的映射请求到处理器后，HandlerMapping 检查给定请求和处理器的CORS配置，并且进行进一步的操作，预检请求被直接的处理，而简单的实际的CORS请求被拦截、验证并需要设置CORS响应头

In order to enable cross-origin requests (i.e. the Origin header is present and differs from the host of the request) you need to have some explicitly declared CORS configuration. If no matching CORS configuration is found, preflight requests are rejected. No CORS headers are added to the responses of simple and actual CORS requests and consequently browsers reject them.

为了实现跨越请求（即Origin头是存在的，并且与请求的主机不同），你需要显式的声明CORS配置，如果没有发现匹配的CORS，则拒绝预检请求，没有CORS头被添加到简单的和实际的CORS请求的响应，因此浏览器拒绝它们

Each HandlerMapping can be configured individually with URL pattern based CorsConfiguration mappings. In most cases applications will use the MVC Java config or the XML namespace to declare such mappings, which results in a single, global map passed to all HadlerMappping's.

每个HandlerMapping 可以使用基于URL模式的CorsConfiguration 映射单独配置。大部分情况下，应用程序用MVC Java配置或XML命名空间声明这样的映射，这将导致一个单一的全局映射传递给所有的HadlerMappping

Global CORS configuration at the HandlerMapping level can be combined with more fine-grained, handler-level CORS configuration. For example annotated controllers can use class or method-level @CrossOrigin annotations (other handlers can implement CorsConfigurationSource).

在HandlerMapping 级别的全局的CORS配置能和更细粒度、处理器级别的CORS配置相联合。举个例子，注解控制器能用类或方法级别的@CrossOrigin注解（其他处理器可以实现CorsConfigurationSource）

The rules for combining global and local configuration are generally additive — e.g. all global and all local origins. For those attributes where only a single value can be accepted such as allowCredentials and maxAge, the local overrides the global value.

组合全局和本地配置的规则通常是累加的--例如所有全局的和所有本地的origins。对于那些只接收单个值的属性，例如allowCredentials 和maxAge，本地覆盖全局的值

### 1.8.3. @CrossOrigin

The @CrossOrigin annotation enables cross-origin requests on annotated controller methods:

@CrossOrigin注解在注解控制器方法上启用了跨域请求

@RestController

@RequestMapping("/account")

**public** **class** **AccountController** {

@CrossOrigin

@GetMapping("/{id}")

**public** Account retrieve(@PathVariable Long id) {

*// ...*

}

@DeleteMapping("/{id}")

**public** **void** remove(@PathVariable Long id) {

*// ...*

}

}

By default @CrossOrigin allows:

默认情况下，@CrossOrigin允许：

All origins.

所有的域

All headers.

所有的请求头

All HTTP methods to which the controller method is mapped.

控制器方法映射到的所有HTTP方法

allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.

allowedCredentials 默认情况下是未启用的，因为它建立了一种信任级别，暴露铭感的用户特定信息，例如cookies和CSRF tokens，只能在适当的地方使用

maxAge is set to 30 minutes

maxAge 设置为30分钟

@CrossOrigin is supported at the class level too and inherited by all methods:

@CrossOrigin在类级别也被支持，并且被所有方法继承

@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)

@RestController

@RequestMapping("/account")

**public** **class** **AccountController** {

@GetMapping("/{id}")

**public** Account retrieve(@PathVariable Long id) {

*// ...*

}

@DeleteMapping("/{id}")

**public** **void** remove(@PathVariable Long id) {

*// ...*

}

}

CrossOrigin can be used at both class and method-level:

CrossOrigin 可以在类和方法级别使用

@CrossOrigin(maxAge = 3600)

@RestController@RequestMapping("/account")

**public** **class** **AccountController** {

@CrossOrigin("http://domain2.com")

@GetMapping("/{id}")

**public** Account retrieve(@PathVariable Long id) {

*// ...*

}

@DeleteMapping("/{id}")

**public** **void** remove(@PathVariable Long id) {

*// ...*

}

}

### 1.8.4. Global Config

In addition to fine-grained, controller method level configuration you’ll probably want to define some global CORS configuration too. You can set URL-based CorsConfiguration mappings individually on any HandlerMapping. Most applications however will use the MVC Java config or the MVC XNM namespace to do that.

除了细粒度、控制器方法级别的配置外，你也可能想要定义一些全局的CORS配置。你可以在任何HandlerMapping中单独的设置基于URL的CorsConfiguration 映射，然而，大部分应用程序用MVC Java配置或MVC XLM命名空间来实现这一点

By default global configuration enables the following:

默认情况下全局配置启用以下内容

All origins.

所有的域

All headers.

所有的请求头

GET, HEAD, and POST methods.

GET，HEAD和POST方法

allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.

allowedCredentials 默认情况下未启用，因为它建立了一个信任级别，暴露敏感的用户特定信息，例如cookies和CSRF tokens，只能在适当的地方使用

maxAge is set to 30 minutes.

maxAge 设置为30分钟

#### Java Config

To enable CORS in the MVC Java config, use the CorsRegistry callback:

要在MVC Java配置中启用CORS，请用CorsRegistry 回调

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addCorsMappings(CorsRegistry registry) {

registry.addMapping("/api/\*\*")

.allowedOrigins("http://domain2.com")

.allowedMethods("PUT", "DELETE")

.allowedHeaders("header1", "header2", "header3")

.exposedHeaders("header1", "header2")

.allowCredentials(true).maxAge(3600);

}

}

#### XML Config

To enable CORS in the XML namespace, use the <mvc:cors> element:

要在XML命名空间中启用CORS，请用<mvc:cors>元素

<mvc:cors>

<mvc:mapping path="/api/\*\*"

allowed-origins="http://domain1.com, http://domain2.com"

allowed-methods="GET, PUT"

allowed-headers="header1, header2, header3"

exposed-headers="header1, header2" allow-credentials="true"

max-age="123" />

<mvc:mapping path="/resources/\*\*"

allowed-origins="http://domain1.com" />

</mvc:cors>

### 1.8.5. CORS Filter

You can apply CORS support through the built-in CorsFilter.

你可以通过内置的CorsFilter应用CORS 支持

To configure the filter pass a CorsConfigurationSource to its constructor:

传递CorsConfigurationSource 到它的构造器来配置过滤器

CorsConfiguration config = **new** CorsConfiguration();

*// Possibly...// config.applyPermitDefaultValues()*

config.setAllowCredentials(true);

config.addAllowedOrigin("http://domain1.com");

config.addAllowedHeader("**"**);****

****config.addAllowedMethod(**"**");

UrlBasedCorsConfigurationSource source = **new** UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", config);

CorsFilter filter = **new** CorsFilter(source);

## 1.9. Web Security

The Spring Security project provides support for protecting web applications from malicious exploits

Spring Security 项目为包含web应用程序免受恶意攻击提供支持

HDIV is another web security framework that integrates with Spring MVC.

HDIV是Spring MVC集成的另一个web安全框架

## 1.10. HTTP Caching

A good HTTP caching strategy can significantly improve the performance of a web application and the experience of its clients. The 'Cache-Control' HTTP response header is mostly responsible for this, along with conditional headers such as 'Last-Modified' and 'ETag'.

一个好的HTTP缓存策略能够显著的提高web应用程序的性能和它的客户体验。“Cache-Control”HTTP响应主要负责这个，以及诸如'Last-Modified' 和'ETag'条件标头

The 'Cache-Control' HTTP response header advises private caches (e.g. browsers) and public caches (e.g. proxies) on how they can cache HTTP responses for further reuse.

“Cache-Control” HTTP响应头建议私有的缓存（例如浏览器）和公有的缓存（例如代理），它们如何缓存HTTP响应以供进一步使用

An ETag (entity tag) is an HTTP response header returned by an HTTP/1.1 compliant web server used to determine change in content at a given URL. It can be considered to be the more sophisticated successor to the Last-Modified header. When a server returns a representation with an ETag header, the client can use this header in subsequent GETs, in an If-None-Match header. If the content has not changed, the server returns 304: Not Modified.

ETag (entity tag) 是 HTTP/1.1兼容的web服务器返回的HTTP响应头，用来确定给定URL在内容上的改变，它可以被认为是 Last-Modified头的更复杂的后继者。当服务器返回一个带有ETag头的表示时，客户端可以在随后的GETs请求中使用该头，在If-None-Match头中，如果内容没有改变，服务器返回304：没有修改

This section describes the different choices available to configure HTTP caching in a Spring Web MVC application.

本节描述在Spring Web MVC应用程序中配置HTTP缓存的不同选择

### 1.10.1. Cache-Control

Spring Web MVC supports many use cases and ways to configure "Cache-Control" headers for an application.

Spring Web MVC 支持许多用例和为应用程序配置 "Cache-Control"头的方式

Spring Web MVC uses a configuration convention in several of its APIs: setCachePeriod(int seconds):

Spring Web MVC在其几个API中使用了一个配置约定：setCachePeriod(int seconds)

A -1 value won’t generate a 'Cache-Control' response header.

-1值不会生成 'Cache-Control' 响应头

A 0 value will prevent caching using the 'Cache-Control: no-store' directive.

使用“Cache-Control: no-store”指令时，0值将阻止缓存

An n > 0 value will cache the given response for n seconds using the 'Cache-Control: max-age=n' directive.

n>0值将使用“Cache-Control: max-age=n”指令将给定的响应缓存n秒

The CacheControl builder class simply describes the available "Cache-Control" directives and makes it easier to build your own HTTP caching strategy. Once built, a CacheControl instance can then be accepted as an argument in several Spring Web MVC APIs.

CacheControl 构造器类简单的描述了可用的“Cache-Control”指令，使得构建自己的HTTP缓存策略更加容易。一旦构建完成，CacheControl 实例就可以在几个Spring Web MVC API中被接受为一个参数

*// Cache for an hour - "Cache-Control: max-age=3600"*

CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

*// Prevent caching - "Cache-Control: no-store"*

CacheControl ccNoStore = CacheControl.noStore();

*// Cache for ten days in public and private caches,*

*// public caches should not transform the response*

*// "Cache-Control: max-age=864000, public, no-transform"*

CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS)

.noTransform().cachePublic();

### 1.10.2. Static resources

Static resources should be served with appropriate 'Cache-Control' and conditional headers for optimal performance.Configuring a ResourceHttpRequestHandler for serving static resources not only natively writes 'Last-Modified' headers by reading a file’s metadata, but also 'Cache-Control' headers if properly configured.

静态资源应当使用适当的“Cache-Control”和条件头以获取最佳性能，配置用于静态资源的ResourceHttpRequestHandler 不仅通过读取文件头的元数据来本地写入'Last-Modified'头，如果配置正确还写入 'Cache-Control' 头

You can set the cachePeriod attribute on a ResourceHttpRequestHandler or use a CacheControl instance, which supports more specific directives:

你可以在ResourceHttpRequestHandler 上设置cachePeriod 属性，也可以使用支持更多特殊指令的CacheControl 实例

@Configuration@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public-resources/")

.setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic());

}

}

And in XML:

在XML中：

<mvc:resources mapping="/resources/\*\*" location="/public-resources/">

<mvc:cache-control max-age="3600" cache-public="true"/>

</mvc:resources>

### 1.10.3. @Controller caching

Controllers can support 'Cache-Control', 'ETag', and/or 'If-Modified-Since' HTTP requests; this is indeed recommended if a 'Cache-Control' header is to be set on the response. This involves calculating a lastModified long and/or an Etag value for a given request, comparing it against the 'If-Modified-Since' request header value, and potentially returning a response with status code 304 (Not Modified).

控制器可以支持'Cache-Control', 'ETag', 和/或 'If-Modified-Since' HTTP请求，如果'Cache-Control' 在响应头中设置，这确实是推荐的。这包括计算给定请求的lastModified long和/或 Etag值，并将其与'If-Modified-Since' 请求头值做对比，并可能返回状态码304（未修改）的响应

As described in HttpEntity, controllers can interact with the request/response using HttpEntity types. Controllers returning ResponseEntity can include HTTP caching information in responses like this:

正如HttpEntity中所述，控制器可以使用HttpEntity 类型与请求/响应进行交互。返回ResponseEntity 的控制器能够在响应中包含HTTP缓存信息，如下所示：

@GetMapping("/book/{id}")

**public** ResponseEntity<Book> showBook(@PathVariable Long id) {

Book book = findBook(id);

String version = book.getVersion();

**return** ResponseEntity

.ok()

.cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))

.eTag(version) *// lastModified is also available*

.body(book);

}

Doing this will not only include 'ETag' and 'Cache-Control' headers in the response, it will also convert the response to an HTTP 304 Not Modified response with an empty body if the conditional headers sent by the client match the caching information set by the Controller.

这样做不仅包含响应中的'ETag' 和'Cache-Control'头，如果客户发送条件调和控制器设置的缓存信息相匹配，它也会将响应转换为一个具有空body的HTTP 304未作修改响应

An @RequestMapping method may also wish to support the same behavior. This can be achieved as follows:

@RequestMapping也可能希望支持同样的行为，这可以实现如下：

@RequestMapping

**public** String myHandleMethod(WebRequest webRequest, Model model) {

**long** lastModified = *// 1. application-specific calculation*

**if** (request.checkNotModified(lastModified)) {

*// 2. shortcut exit - no further processing necessary*

**return** null;

}

*// 3. or otherwise further request processing, actually preparing content*

model.addAttribute(...);

**return** "myViewName";

}

There are two key elements here: calling request.checkNotModified(lastModified) and returning null. The former sets the appropriate response status and headers before it returns true. The latter, in combination with the former, causes Spring MVC to do no further processing of the request.

这里有两个关键元素：调用 request.checkNotModified(lastModified) 和返回null，前者在返回true之前设置适当的响应状态和标头。后者与前者结合，使得Spring MVC不再对请求做进一步处理

Note that there are 3 variants for this:

请注意，对这个有3个变量

request.checkNotModified(lastModified) compares lastModified with the 'If-Modified-Since' or 'If-Unmodified-Since' request header

request.checkNotModified(lastModified)和'If-Modified-Since' 或'If-Unmodified-Since' 请求头进行比较

request.checkNotModified(eTag) compares eTag with the 'If-None-Match' request header

request.checkNotModified(eTag) 和‘If-None-Match’请求头进行比较

request.checkNotModified(eTag, lastModified) does both, meaning that both conditions should be valid

request.checkNotModified(eTag, lastModified)同时具有这两个条件，意味着两个条件都得有效

When receiving conditional 'GET'/'HEAD' requests, checkNotModified will check that the resource has not been modified and if so, it will result in a HTTP 304 Not Modified response. In case of conditional 'POST'/'PUT'/'DELETE' requests, checkNotModified will check that the resource has not been modified and if it has been, it will result in a HTTP 409 Precondition Failed response to prevent concurrent modifications.

当接收到有条件'GET'/'HEAD'的请求时，checkNotModified 将检查资源是否未被修改，如果是这样的话，将导致HTTP 304未做修改响应。在条件'POST'/'PUT'/'DELETE'请求的情况下， checkNotModified 检查资源是否未被修改，如果被修改，将导致HTTP 409 前置条件失败响应，以防止并发修改

### 1.10.4. ETag Filter

Support for ETags is provided by the Servlet filter ShallowEtagHeaderFilter. It is a plain Servlet Filter, and thus can be used in combination with any web framework. The ShallowEtagHeaderFilter filter creates so-called shallow ETags by caching the content written to the response and generating an MD5 hash over that to send as an ETag header. The next time a client sends a request for the same resource, it uses that hash as the If-None-Match value. The filter detects this, lets the request be processed as usual, and at the end compares the two hashes. If they are equal, a 304 is returned.

对ETags 的支持由Servlet过滤器ShallowEtagHeaderFilter提供。它是一个普通的Servlet过滤器，因此能和任何web框架联合使用，ShallowEtagHeaderFilter 过滤器通过缓存写入响应的内容来创建所谓的浅ETags，并生成一个MD5 散列最为ETag头来发送。下次客户端发送相同资源的请求时，它将使用散列值作为“If-None-Match ”值，过滤器检测到这一点，让请求像往常一样处理，并在最后比较这两个hash值，如果它们相等，返回304

Note that this strategy saves network bandwidth but not CPU, as the full response must be computed for each request. Other strategies at the controller level, described above, can avoid computation.

请注意，此策略可以节省网络宽带而不是CPU，因为必须为每个请求计算完整响应。上述描述的控制器级别的其他策略能够避免计算

## 1.11. MVC Config

The MVC Java config and the MVC XML namespace provide default configuration suitable for most applications along with a configuration API to customize it.

MVC Java配置和MVC XML命名空间提供了适用于大多数应用程序的默认配置，以及配置API来对其进行自定义  
**1.11.1. Enable MVC Config**

In Java config use the @EnableWebMvc annotation:

在Java配置中用@EnableWebMvc注解

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** {

}

In XML use the <mvc:annotation-driven> element:

在XML中用<mvc:annotation-driven>元素

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<mvc:annotation-driven/>

</beans>

The above registers a number of Spring MVC infrastructure beans also adapting to dependencies available on the classpath: e.g. payload converters for JSON, XML, etc.

上面注册了一些Spring MVC基础设置bean，也适应了类路径上的依赖性，例如对JSON，XML等的有效载荷的转换器

### **1.11.2. MVC Config API**

In Java config implement WebMvcConfigurer interface:

在Java配置中实现WebMvcConfigurer 接口

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

*// Implement configuration methods...*

}

In XML check attributes and sub-elements of <mvc:annotation-driven/>.

在XML中检查<mvc:annotation-driven/>的属性和子集

### 1.11.3. Type conversion

By default formatters for Number and Date types are installed, including support for the @NumberFormat and @DateTimeFormat annotations. Full support for the Joda Time formatting library is also installed if Joda Time is present on the classpath.

默认情况下，对数值和日期类型的转换器已经安装，包括对@NumberFormat 和@DateTimeFormat注解的支持。如果Joda时间在当前类路径出现，则还安装队Joda时间转换库的完全支持

In Java config, register custom formatters and converters:

在Java配置中，注册自定义格式化器和转化器

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addFormatters(FormatterRegistry registry) {

*// ...*

}

}

In XML, the same:

在XML中，同样：

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<mvc:annotation-driven conversion-service="conversionService"/>

<bean id="conversionService"

class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="org.example.MyConverter"/>

</set>

</property>

<property name="formatters">

<set>

<bean class="org.example.MyFormatter"/>

<bean class="org.example.MyAnnotationFormatterFactory"/>

</set>

</property>

<property name="formatterRegistrars">

<set>

<bean class="org.example.MyFormatterRegistrar"/>

</set>

</property>

</bean>

</beans>

### 1.11.4. Validation

By default if Bean Validation is present on the classpath — e.g. Hibernate Validator, the LocalValidatorFactoryBean is registered as a global Validator for use with @Valid and Validated on controller method arguments.

默认情况下，如果bean验证器在类路径上存在（例如Hibernate 验证器），LocalValidatorFactoryBean 将被注册为一个全局的验证器，和 @Valid 一起使用 并验证控制器方法上的参数

In Java config, you can customize the global Validator instance:

在Java配置中，你可以自定义全局的验证器实例：

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** Validator getValidator(); {

*// ...*

}

}

In XML, the same:

在XML中，同样：

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<mvc:annotation-driven validator="globalValidator"/>

</beans>

Note that you can also register Validator's locally:

请注意，你也可以在本地注册Validator

@Controller

**public** **class** **MyController** {

@InitBinder

**protected** **void** initBinder(WebDataBinder binder) {

binder.addValidators(**new** FooValidator());

}

}

### 1.11.5. Interceptors

In Java config, register interceptors to apply to incoming requests:

在Java配置中，注册拦截器应用于传入的请求

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(**new** LocaleInterceptor());

registry.addInterceptor(**new** ThemeInterceptor()).addPathPatterns("/\*\*").excludePathPatterns("/admin/\*\*");

registry.addInterceptor(**new** SecurityInterceptor()).addPathPatterns("/secure/\*");

}

}

In XML, the same:

在XML中，同样：

<mvc:interceptors>

<bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>

<mvc:interceptor>

<mvc:mapping path="/\*\*"/>

<mvc:exclude-mapping path="/admin/\*\*"/>

<bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path="/secure/\*"/>

<bean class="org.example.SecurityInterceptor"/>

</mvc:interceptor></mvc:interceptors>

### 1.11.6. Content Types

You can configure how Spring MVC determines the requested media types from the request — e.g. Accept header, URL path extension, query parameter, etc.

你可以配置Spring MVC如何从请求中确定请求的媒体类型（例如Accept 头，URL路径扩展，查询参数等）

By default the URL path extension is checked first — with json, xml, rss, and atom registered as known extensions depending on classpath dependencies, and the "Accept" header is checked second.

默认情况下，URL路径扩展首先被检查，依靠类路径依赖将 json, xml, rss, 和atom注册为已知的扩展，然后检查“Accept”头

Consider changing those defaults to Accept header only and if you must use URL-based content type resolution consider the query parameter strategy over the path extensions

考虑更改这些默认值为仅接收头，如果你必须使用基于URL的内容类型解析，请考虑路径扩展上的查询参数策略

In Java config, customize requested content type resolution:

在Java配置中，自定义请求的类容类型解析

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.mediaType("json", MediaType.APPLICATION\_JSON);

}

}

In XML, the same:

在XML中，同样：

<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="mediaTypes">

<value>

json=application/json

xml=application/xml

</value>

</property>

</bean>

### 1.11.7. Message Converters

Customization of HttpMessageConverter can be achieved in Java config by overriding configureMessageConverters() if you want to replace the default converters created by Spring MVC, or by overriding extendMessageConverters() if you just want to customize them or add additional converters to the default ones.

如果你想替换Spring MVC创建的默认转换器，可以在Java配置中通过重载configureMessageConverters()来实现HttpMessageConverter 的自定义化。或者如果你仅仅想要自定义它们或添加额外的转换器到默认转换器上，可以通过重载extendMessageConverters() 来实现

Below is an example that adds Jackson JSON and XML converters with a customized ObjectMapper instead of default ones:

以下是一个例子，该示例使用自定义的ObjectMapper而不是默认值的ObjectMapper添加了Jackson JSON和XML转换器

@Configuration

@EnableWebMvc

**public** **class** **WebConfiguration** **implements** WebMvcConfigurer {

@Override

**public** **void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {

Jackson2ObjectMapperBuilder builder = **new** Jackson2ObjectMapperBuilder()

.indentOutput(true)

.dateFormat(**new** SimpleDateFormat("yyyy-MM-dd"))

.modulesToInstall(**new** ParameterNamesModule());

converters.add(**new** MappingJackson2HttpMessageConverter(builder.build()));

converters.add(**new** MappingJackson2XmlHttpMessageConverter(builder.xml().build()));

}

}

In this example, Jackson2ObjectMapperBuilder is used to create a common configuration for both MappingJackson2HttpMessageConverter and MappingJackson2XmlHttpMessageConverter with indentation enabled, a customized date format and the registration of jackson-module-parameter-names that adds support for accessing parameter names (feature added in Java 8).

在这个例子中，Jackson2ObjectMapperBuilder 用于为MappingJackson2HttpMessageConverter 和MappingJackson2HttpMessageConverter 创建通用配置，并启用缩进，自定义日期格式以及添加对访问参数名称支持的jackson-module-parameter-names的注册（Java8中新增的功能）

This builder customizes Jackson’s default properties with the following ones:

这个构造器使用以下属性自定义Jackson的默认属性

DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES is disabled.

DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES被禁用

MapperFeature.DEFAULT\_VIEW\_INCLUSION is disabled.

MapperFeature.DEFAULT\_VIEW\_INCLUSION被禁用

It also automatically registers the following well-known modules if they are detected on the classpath:

如果在类路径上检测到它们，它也自动的注册以下已知模块

jackson-datatype-jdk7: support for Java 7 types like java.nio.file.Path.

jackson-datatype-jdk7: 支持Java7类型，如java.nio.file.Path.

jackson-datatype-joda: support for Joda-Time types.

jackson-datatype-joda: 支持Joda-Time类型

jackson-datatype-jsr310: support for Java 8 Date & Time API types.

jackson-datatype-jsr310: 支持Java8的日期和时间API类型

jackson-datatype-jdk8: support for other Java 8 types like Optional.

jackson-datatype-jdk8: 支持Java的其他类型，如Optional

Other interesting Jackson modules are available:

其他有趣的Jackson模块是可用的：

jackson-datatype-money: support for javax.money types (unofficial module)

jackson-datatype-money: 支持 javax.money类型（非官方模块）

jackson-datatype-hibernate: support for Hibernate specific types and properties (including lazy-loading aspects)

jackson-datatype-hibernate: 支持Hibernate特定类型和属性（包括延迟加载方面）

It is also possible to do the same in XML:

在XML中也可以这样做

<mvc:annotation-driven>

<mvc:message-converters>

<bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

<property name="objectMapper" ref="objectMapper"/>

</bean>

<bean class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter">

<property name="objectMapper" ref="xmlMapper"/>

</bean>

</mvc:message-converters>

</mvc:annotation-driven>

<bean id="objectMapper" class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean"

p:indentOutput="true"

p:simpleDateFormat="yyyy-MM-dd"

p:modulesToInstall="com.fasterxml.jackson.module.paramnames.ParameterNamesModule"/>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true"/>

### 1.11.8. View Controllers

This is a shortcut for defining a ParameterizableViewController that immediately forwards to a view when invoked. Use it in static cases when there is no Java controller logic to execute before the view generates the response.

这是定义一个ParameterizableViewController 的快捷方式，可以再调用时立即转向一个视图。当在视图生成响应事前，没有Java控制器逻辑执行，在静态情况下使用它

An example of forwarding a request for "/" to a view called "home" in Java:

在java中将“/”转发到一个名为“name”视图的示例：

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/").setViewName("home");

}

}

And the same in XML use the <mvc:view-controller> element:

在XML中一样，使用<mvc:view-controller>元素

<mvc:view-controller path="/" view-name="home"/>

### 1.11.9. View Resolvers

The MVC config simplifies the registration of view resolvers.

MVC配置简化了视图解析器的注册

The following is a Java config example that configures content negotiation view resolution using FreeMarker HTML templates and Jackson as a default View for JSON rendering:

以下是一个Java配置的例子，使用FreeMarker HTML模板和Jackson作为JSON呈现的默认视图来配置内容negotiation视图解析器

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureViewResolvers(ViewResolverRegistry registry) {

registry.enableContentNegotiation(**new** MappingJackson2JsonView());

registry.jsp();

}

}

And the same in XML:

在XML中也一样：

<mvc:view-resolvers>

<mvc:content-negotiation>

<mvc:default-views>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

</mvc:default-views>

</mvc:content-negotiation>

<mvc:jsp/>

</mvc:view-resolvers>

Note however that FreeMarker, Tiles, Groovy Markup and script templates also require configuration of the underlying view technology.

但是请注意，FreeMarker, Tiles, Groovy标记和脚本模板也需要配置底层的视图技术

The MVC namespace provides dedicated elements. For example with FreeMarker:

MVC 命名空间提供了专用的元素，例如FreeMarker

<mvc:view-resolvers>

<mvc:content-negotiation>

<mvc:default-views>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

</mvc:default-views>

</mvc:content-negotiation>

<mvc:freemarker cache="false"/>

</mvc:view-resolvers>

<mvc:freemarker-configurer>

<mvc:template-loader-path location="/freemarker"/>

</mvc:freemarker-configurer>

In Java config simply add the respective "Configurer" bean:

在Java配置中只需要添加各自的"Configurer" bean

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureViewResolvers(ViewResolverRegistry registry) {

registry.enableContentNegotiation(**new** MappingJackson2JsonView());

registry.freeMarker().cache(false);

}

@Bean

**public** FreeMarkerConfigurer freeMarkerConfigurer() {

FreeMarkerConfigurer configurer = **new** FreeMarkerConfigurer();

configurer.setTemplateLoaderPath("/WEB-INF/");

**return** configurer;

}

}

### 1.11.10. Static Resources

This option provides a convenient way to serve static resources from a list of Resource-based locations.

这个选项提供了一种便捷方式，可以从基于资源的位置列表中提供静态资源

In the example below, given a request that starts with "/resources", the relative path is used to find and serve static resources relative to "/public" under the web application root or on the classpath under "/static". The resources are served with a 1-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser. The Last-Modified header is also evaluated and if present a 304 status code is returned.

在下面的示例中，给定一个以"/resources"开头的请求，相对路径用于在web应用程序根目录下或”/static”下的类路径上查找和提供相对于”/public”的静态资源 ，这些资源使用期限为一年，来确保浏览器缓存的最大利用并减少浏览器发出的HTTP请求。Last-Modified 头也被评估，如果存在，则返回304状态码

In Java config:

在Java配置中

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public", "classpath:/static/")

.setCachePeriod(31556926);

}

}

In XML:

在XML中

<mvc:resources mapping="/resources/\*\*"

location="/public, classpath:/static/"

cache-period="31556926" />

The resource handler also supports a chain of ResourceResolver's and ResourceResolver's. which can be used to create a toolchain for working with optimized resources.

资源处理器也支持ResourceResolver和ResourceResolver的链，它可以用于创建一个工具链来处理优化的资源

The VersionResourceResolver can be used for versioned resource URLs based on an MD5 hash computed from the content, a fixed application version, or other. A ContentVersionStrategy (MD5 hash) is a good choice with some notable exceptions such as JavaScript resources used with a module loader.

VersionResourceResolver 可用于基于内容、固定应用程序版本或其他的计算过的MD5哈希的版本资源URLs，ContentVersionStrategy (MD5 哈希)是个好选择，但有一些值得注意的例外，例如模块加载器使用的JavaScript资源

For example in Java config;

举个例子，在Java配置中

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public/")

.resourceChain(true)

.addResolver(**new**

VersionResourceResolver().addContentVersionStrategy("/\*\*"));

}

}

In XML, the same:

在XML中，同样：

<mvc:resources mapping="/resources/\*\*" location="/public/">

<mvc:resource-chain>

<mvc:resource-cache/>

<mvc:resolvers>

<mvc:version-resolver>

<mvc:content-version-strategy patterns="/\*\*"/>

</mvc:version-resolver>

</mvc:resolvers>

</mvc:resource-chain>

</mvc:resources>

You can use ResourceUrlProvider to rewrite URLs and apply the full chain of resolvers and transformers — e.g. to insert versions. The MVC config provides a ResourceUrlProvider bean so it can be injected into others. You can also make the rewrite transparent with the ResourceUrlEncodingFilter for Thymeleaf, JSPs, FreeMarker, and others with URL tags that rely on HttpServletResponse#encodeURL.

你可以使用ResourceUrlProvider来重写URL，并应用到解析器和转换器的完整链--例如，插入版本。MVC配置提供了ResourceUrlProvider bean，因此它可以注入到其他bean。你可以用Thymeleaf、JSP、FreeMarker的ResourceUrlEncodingFilter和其他依赖于HttpServletResponse#encodeURL的URL标签来使重新透明

WebJars is also supported via WebJarsResourceResolver and automatically registered when "org.webjars:webjars-locator" is present on the classpath. The resolver can re-write URLs to include the version of the jar and can also match to incoming URLs without versions — e.g. "/jquery/jquery.min.js" to "/jquery/1.2.0/jquery.min.js".

WebJars 也通过WebJarsResourceResolver 支持并在类路径上存在“org.webjars:webjars-locator”时自动注册。这个解析器能够重写URL以包括jar的版本，并可以与没有版本的输入URL匹配--例如"/jquery/jquery.min.js" 到 "/jquery/1.2.0/jquery.min.js".

### 1.11.11. Default Servlet

This allows for mapping the DispatcherServlet to "/" (thus overriding the mapping of the container’s default Servlet), while still allowing static resource requests to be handled by the container’s default Servlet. It configures a DefaultServletHttpRequestHandler with a URL mapping of "/\*\*" and the lowest priority relative to other URL mappings.

这允许将DispatcherServlet映射到“/”(从而覆盖容器默认Servlet的映射)，同时允许静态资源请求通过容器默认的Servlet处理。它用“/\*\*”的URL映射来配置DefaultServletHttpRequestHandler ，相对于其他URL映射最低优先级

This handler will forward all requests to the default Servlet. Therefore it is important that it remains last in the order of all other URL HandlerMappings. That will be the case if you use <mvc:annotation-driven> or alternatively if you are setting up your own customized HandlerMapping instance be sure to set its order property to a value lower than that of the DefaultServletHttpRequestHandler, which is Integer.MAX\_VALUE.

处理器将把所有的请求转发到默认Servlet。因此，重要的是 它保持在其他URL HandlerMappings顺序的最后。如果你使用<mvc:annotation-driven>或你正在设置你自己的自定义HandlerMapping 实例，则确保将其顺序属性设为一个低于DefaultServletHttpRequestHandler（即Integer.MAX\_VALUE）的值

To enable the feature using the default setup use:

使用默认设置启用该功能，请用：

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

}

Or in XML:

或在XML中

<mvc:default-servlet-handler/>

The caveat to overriding the "/" Servlet mapping is that the RequestDispatcher for the default Servlet must be retrieved by name rather than by path. The DefaultServletHttpRequestHandler will attempt to auto-detect the default Servlet for the container at startup time, using a list of known names for most of the major Servlet containers (including Tomcat, Jetty, GlassFish, JBoss, Resin, WebLogic, and WebSphere). If the default Servlet has been custom configured with a different name, or if a different Servlet container is being used where the default Servlet name is unknown, then the default Servlet’s name must be explicitly provided as in the following example:

覆盖"/" Servlet映射的注意事项是，默认Servlet的RequestDispatcher 必须通过名称而不是路径来检索。DefaultServletHttpRequestHandler 将尝试在启动时自动检测容器的默认的Servlet，使用大多数主要容器的已知名称列表（包括 Tomcat, Jetty, GlassFish, JBoss, Resin, WebLogic, and WebSphere） 。如果默认的Servlet用不同的名称自定义配置，或者如果在默认Servlet名称未知的地方使用了不同的Servlet容器，那么默认Servlet的名称必须显式地提供，如下例所示：

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable("myCustomDefaultServlet");

}

}

Or in XML:

或在XML中

<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>

### 1.11.12. Path Matching

This allows customizing options related to URL matching and treatment of the URL

这允许自定义与URL映射和URL处理有关的选项

Example in Java config:

Java配置中的示例

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configurePathMatch(PathMatchConfigurer configurer) {

configurer

.setUseSuffixPatternMatch(true)

.setUseTrailingSlashMatch(false)

.setUseRegisteredSuffixPatternMatch(true)

.setPathMatcher(antPathMatcher())

.setUrlPathHelper(urlPathHelper());

}

@Bean

**public** UrlPathHelper urlPathHelper() {

*//...*

}

@Bean

**public** PathMatcher antPathMatcher() {

*//...*

}

}

In XML, the same:

在XML中，同样

<mvc:annotation-driven>

<mvc:path-matching

suffix-pattern="true"

trailing-slash="false"

registered-suffixes-only="true"

path-helper="pathHelper"

path-matcher="pathMatcher"/></mvc:annotation-driven>

<bean id="pathHelper" class="org.example.app.MyPathHelper"/>

<bean id="pathMatcher" class="org.example.app.MyPathMatcher"/>

### 1.11.13. Advanced Java Config

@EnableWebMvc imports DelegatingWebMvcConfiguration that (1) provides default Spring configuration for Spring MVC applications and (2) detects and delegates to WebMvcConfigurer's to customize that configuration.

@EnableWebMvc 导入DelegatingWebMvcConfiguration ，（1）为Spring MVC 应用程序提供了默认的Spring配置，并且（2）检查并委托给WebMvcConfigurer来定制该配置

For advanced mode, remove @EnableWebMvc and extend directly from DelegatingWebMvcConfiguration instead of implementing WebMvcConfigurer:

对于高级模式，移除@EnableWebMvc并直接从DelegatingWebMvcConfiguration 扩展而不是实现WebMvcConfigurer

@Configuration

**public** **class** **WebConfig** **extends** DelegatingWebMvcConfiguration {

*// ...*

}

You can keep existing methods in WebConfig but you can now also override bean declarations from the base class and you can still have any number of other WebMvcConfigurer's on the classpath.

你可以在WebConfig中保留现有的方法，但是现在你也可以从基类中覆盖bean声明，并且你仍然可以再类路径上拥有任意数量的其他WebMvcConfigurer

### 1.11.14. Advanced XML Config

The MVC namespace does not have an advanced mode. If you need to customize a property on a bean that you can’t change otherwise, you can use the BeanPostProcessor lifecycle hook of the Spring ApplicationContext:

MVC 命名空间没有高级模式，如果你需要自定义一个你不能改变的bean的属性，你可以使用Spring  ApplicationContext的 BeanPostProcessor 声明周期钩子

@Component

**public** **class** **MyPostProcessor** **implements** BeanPostProcessor {

**public** Object postProcessBeforeInitialization(Object bean, String name) **throws** BeansException {

*// ...*

}

}

Note that MyPostProcessor needs to be declared as a bean either explicitly in XML or detected through a <component scan/>declaration.

请注意，MyPostProcessor需要在XML中显式声明为bean，或者通过<component scan />声明来检测。

## 1.12. View Technologies

### 1.12.1. Introduction

One of the areas in which Spring excels is in the separation of view technologies from the rest of the MVC framework. For example, deciding to use Groovy Markup Templates or Thymeleaf in place of an existing JSP is primarily a matter of configuration. This chapter covers the major view technologies that work with Spring and touches briefly on how to add new ones. This chapter assumes you are already familiar with View Resolution which covers the basics of how views in general are coupled to the MVC framework.

Spring擅长的领域之一是将视图技术和MVC框架的其余部分分离。举个例子，决定使用Groovy标记模板或Thymeleaf 替换已有的JSP主要是配置的问题。本章覆盖了和Spring合作的主要的视图技术，并简要的介绍如何添加新技术。本章假定你已经熟悉 View Resolution，它涵盖了一般视图如何与MVC框架耦合的基本知识

### 1.12.2. Thymeleaf

Thymeleaf is a good example of a view technology fitting perfectly in the MVC framework。Support for this integration is not provided by the Spring team but by the Thymeleaf team itself.

Thymeleaf 是MVC框架中 完美匹配视图技术的一个很好的例子，对这种集成的支持不是由Spring团队提供的，而是由Thymeleaf 团队自己提供的

Configuring Thymeleaf for Spring usually requires a few beans defined, like a ServletContextTemplateResolver, a SpringTemplateEngine and a ThymeleafViewResolver.

为Spring配置Thymeleaf 通常需要定义一些beans，如ServletContextTemplateResolver，SpringTemplateEngine 和ThymeleafViewResolver

### 1.12.3. Groovy Markup

The Groovy Markup Template Engine is another view technology, supported by Spring. This template engine is a template engine primarily aimed at generating XML-like markup (XML, XHTML, HTML5, …​​), but that can be used to generate any text based content.

Groovy标记模板引擎是Spring支持的另一个视图技术。这个模板引擎是一个模板引擎，其主要生成类似于XML的标记（XML, XHTML, HTML5, …​），但它能够用于生成任何基于文本的内容

This requires Groovy 2.3.1+ on the classpath.

在类路径上需要Groovy 2.3.1+

#### Configuration

Configuring the Groovy Markup Template Engine is quite easy:

配置Groovy标记模板引擎是非常容易的

@Configuration

@EnableWebMvc

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureViewResolvers(ViewResolverRegistry registry) {

registry.groovy();

}

@Bean

**public** GroovyMarkupConfigurer groovyMarkupConfigurer() {

GroovyMarkupConfigurer configurer = **new** GroovyMarkupConfigurer();

configurer.setResourceLoaderPath("/WEB-INF/");

**return** configurer;

}

}

The XML counterpart using the MVC namespace is:

使用MVC命名空间的对应项是：

<mvc:annotation-driven/>

<mvc:view-resolvers>

<mvc:groovy/></mvc:view-resolvers>

<mvc:groovy-configurer resource-loader-path="/WEB-INF/"/>

#### Example

Unlike traditional template engines, this one relies on a DSL that uses the builder syntax. Here is a sample template for an HTML page:

与传统的模板引擎不同，这个引擎依赖于使用builder 语法的DSL，下面是HTML页面的一个示例模板

yieldUnescaped '<!DOCTYPE html>'

html(lang:'en') {

head {

meta('http-equiv':'"Content-Type" content="text/html; charset=utf-8"')

title('My page')

}

body {

p('This is an example of HTML contents')

}

}

### 1.12.4. FreeMarker

FreeMarker is a templating language that can be used as a view technology within Spring MVC applications.

FreeMarker 是一个模板语言，它能够在Spring MVC应用程序中用作视图技术

#### Dependencies

Your web application will need to include freemarker-2.x.jar in order to work with FreeMarker. Typically this is included in the WEB-INF/lib folder where the jars are guaranteed to be found by a Java EE server and added to the classpath for your application. It is of course assumed that you already have the spring-webmvc.jar in your 'WEB-INF/lib' directory too!

你的web应用程序需要包含 freemarker-2.x.jar，以便与FreeMarker一起使用，通常，它包含在WEB-INF/lib 文件夹下，在该文件夹中，Java EE服务器保证可以找到jar 并 将其添加到你应用程序的类路径。 当然，也假定你的“WEB-INF/lib”目录下也已经有spring-webmvc.jar了

#### Context configuration

A suitable configuration is initialized by adding the relevant configurer bean definition to your '\*-servlet.xml' as shown below:

通过添加相关的配置器bean定义到'\*-servlet.xml'中来初始化一个适当的配置。如下所示：

*<!-- freemarker config -->*

<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">

<property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>

</bean>

*<!--*

*View resolvers can also be configured with ResourceBundles or XML files. If you need*

*different view resolving based on Locale, you have to use the resource bundle resolver.*

*-->*

<bean id="viewResolver" class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">

<property name="cache" value="true"/>

<property name="prefix" value=""/>

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

</bean>

#### Creating templates

Your templates need to be stored in the directory specified by the FreeMarkerConfigurer shown above. If you use the view resolvers highlighted, then the logical view names relate to the template file names in similar fashion to InternalResourceViewResolver for JSP’s. So if your controller returns a ModelAndView object containing a view name of "welcome" then the resolver will look for the /WEB-INF/freemarker/welcome.ftl template.

你的模板需要存放在上述FreeMarkerConfigurer 指定的文件下，如果你使用突出显示的视图解析器，那么逻辑视图名称与模板文件名称的关联类似于JSP的InternalResourceViewResolver 。因此，如果你的控制器名称返回一个包含视图名为“welcome”的ModelAndView对象，那么解析器将查找/WEB-INF/freemarker/welcome.ftl模板

#### Advanced config

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker Configuration object managed by Spring by setting the appropriate bean properties on the FreeMarkerConfigurer bean. The freemarkerSettings property requires a java.util.Properties object and the freemarkerVariables property requires a java.util.Map.

通过在FreeMarkerConfigurer bean上设置适当的bean属性，FreeMarker 的 'Settings' 和'SharedVariables'可以直接传递给由Spring管理的FreeMarker配置对象。freemarkerSettings 设置属性需要一个java.util.Properties对象，freemarkerVariables 属性需要一个java.util.Map

<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">

<property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>

<property name="freemarkerVariables">

<map>

<entry key="xml\_escape" value-ref="fmXmlEscape"/>

</map>

</property>

</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>

#### Form handling

Spring provides a tag library for use in JSP’s that contains (amongst other things) a <spring:bind/> tag. This tag primarily enables forms to display values from form backing objects and to show the results of failed validations from a Validator in the web or business tier. Spring also has support for the same functionality in FreeMarker, with additional convenience macros for generating form input elements themselves.

Spring提供一个在JSP中使用的标签库，其中包含（除其他事项外）一个<spring:bind/>标签。该标签主要

使表单能够显示来自表单返回对象的值，并显示来自web或业务层中验证器返回的失败验证结果。Spring在FreeMarker中也支持相同的功能，并为生成表单输入元素自身提供了其他的外方便的宏

#### The bind macros

A standard set of macros are maintained within the spring-webmvc.jar file for both languages, so they are always available to a suitably configured application.

在两种语言的spring-webmvc.jar文件中都保留着一组标准的宏集合，因此，它们始终可用于适当配置应用程序

Some of the macros defined in the Spring libraries are considered internal (private) but no such scoping exists in the macro definitions making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to be directly calling from within your templates. If you wish to view the macro code directly, the file is called spring.ftl in the package org.springframework.web.servlet.view.freemarker.

Spring库中的一些宏定义被认为是内部的（私有的），但是在宏定义中不存在这样的作用域，使得所有宏在调用代码和用户模板时可见。下面的章节仅仅集中在你需要直接从你的模板中调用的宏，如果你希望直接地看宏代码，该文件被称为spring.ftl，在包org.springframework.web.servlet.view.freemarker中

#### Simple binding

In your HTML forms (vm / ftl templates) which act as a form view for a Spring MVC controller, you can use code similar to the following to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. Example code is shown below for the personForm view configured earlier:

在充当SpringMVC控制器表单视图的HTML表单中（vm/ftl模板），你可以使用以下类似的代码来绑定到字段值，并以与JSP等效的方式为每个输入字段显示错误信息，下面显示了以前配置的personForm 视图的示例代码

*<!-- freemarker macros have to be imported into a namespace. We strongly*

*recommend sticking to 'spring' -->*

<#import "/spring.ftl" as spring/>

<html>

...

<form action="" method="POST">

Name:

<@spring.bind "myModelObject.name"/>

<input type="text"

name="${spring.status.expression}"

value="${spring.status.value?html}"/><br>

<#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>

<br>

...

<input type="submit" value="submit"/>

</form>

...

</html>

<@spring.bind> requires a 'path' argument which consists of the name of your command object (it will be 'command' unless you changed it in your FormController properties) followed by a period and the name of the field on the command object you wish to bind to. Nested fields can be used too such as "command.address.street". The bind macro assumes the default HTML escaping behavior specified by the ServletContext parameter defaultHtmlEscape in web.xml.

<@spring.bind>需要一个“path”参数，它由你的命令对象的名称组成（除非你在FormController 属性中修改它，否则它将是“command”），后面跟着一个句点和要绑定到命令对象上的字段名。嵌套字段也可以被使用，例如“command.address.street” 。绑定宏假定 ServletContext参数defaultHtmlEscape 在web.xml中指定的默认HTML转义行为

The optional form of the macro called <@spring.bindEscaped> takes a second argument and explicitly specifies whether HTML escaping should be used in the status error messages or values. Set to true or false as required. Additional form handling macros simplify the use of HTML escaping and these macros should be used wherever possible. They are explained in the next section.

名为 <@spring.bindEscaped>的宏的可选形式接受第二个参数，并明确指定是否应该在状态错误消息或值中使用HTML转义，根据需要设置为true或false。额外的表单处理宏简化了HTML转义的使用，应该尽可能的使用这些宏。它们将在下一章解释

#### Input macros

Additional convenience macros for both languages simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and they can be mixed and matched with simple HTML or calls direct to the spring bind macros highlighted previously.

这两种语言的其他的便利宏 简化 了绑定和表单生成（包括验证错误显示）。永远不需要使用这些宏来生成表单输入字段，它们可以与简单的HTML混合并匹配， 或直接调用前面突出显示的Spring bind宏

The following table of available macros show the VTL and FTL definitions and the parameter list that each takes.

下面的可用宏表显示了VTL和 FTL定义 并各自需要的参数列表

Table 5. Table of macro definitions

| macro | FTL definition | message (output a string from a resource bundle based on the code parameter) |
| --- | --- | --- |
| <@spring.message code/> | messageText (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter) | <@spring.messageText code, text/> |
| url (prefix a relative URL with the application’s context root) | <@spring.url relativeUrl/> | formInput (standard input field for gathering user input) |
| <@spring.formInput path, attributes, fieldType/> | formHiddenInput \* (hidden input field for submitting non-user input) | <@spring.formHiddenInput path, attributes/> |
| formPasswordInput \* (standard input field for gathering passwords. Note that no value will ever be populated in fields of this type) | <@spring.formPasswordInput path, attributes/> | formTextarea (large text field for gathering long, freeform text input) |
| <@spring.formTextarea path, attributes/> | formSingleSelect (drop down box of options allowing a single required value to be selected) | <@spring.formSingleSelect path, options, attributes/> |
| formMultiSelect (a list box of options allowing the user to select 0 or more values) | <@spring.formMultiSelect path, options, attributes/> | formRadioButtons (a set of radio buttons allowing a single selection to be made from the available choices) |
| <@spring.formRadioButtons path, options separator, attributes/> | formCheckboxes (a set of checkboxes allowing 0 or more values to be selected) | <@spring.formCheckboxes path, options, separator, attributes/> |
| formCheckbox (a single checkbox) | <@spring.formCheckbox path, attributes/> | showErrors (simplify display of validation errors for the bound field) |

In FTL (FreeMarker), these two macros are not actually required as you can use the normal formInput macro, specifying ' hidden’ or ' `password’ as the value for the `fieldType parameter.

在FTL（FreeMarker）中，这两种宏实际上并不需要，因为你可以使用普通的formInput宏，指定' hidden’ 或' `password’作为“fieldType ”参数的值

The parameters to any of the above macros have consistent meanings:

上述任意一个宏参数都有一致的含义：

path: the name of the field to bind to (ie "command.name")

Path：要绑定的字段名称（例如：“command.name”）

options: a Map of all the available values that can be selected from in the input field. The keys to the map represent the values that will be POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from the corresponding values posted back by the form. Usually such a map is supplied as reference data by the controller. Any Map implementation can be used depending on required behavior. For strictly sorted maps, a SortedMap such as a TreeMap with a suitable Comparator may be used and for arbitrary Maps that should return values in insertion order, use a LinkedHashMap or a LinkedMap from commons-collections.

Options：可以从输入字段选择的所有可用值的映射。这个Map的Key表示 从表单传递返回并绑定到命令对象上的值。Key值对应存储的Map对象 是 表单上显示给用户的标签， 可能和表单传递返回的响应值不同。 通常这样的map由控制器作为参考数据提供，任何Map实现可以根据需要的行为来使用，对于严格排序的map。 可以使用SortedMap （例如具有适当比较器的TreeMap），对于应该按插入顺序值返回的任意Map。请使用commons-collections集合中的LinkedHashMap 或者LinkedMap

separator: where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (ie "<br>").

Separator：其中多个选项可用作离散元素（单选按钮或复选框），用于分隔列表中每个元素（例如“<br>”）的字符序列

attributes: an additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field you may supply attributes as 'rows="5" cols="60"' or you could pass style information such as 'style="border:1px solid silver"'.

HTML标签自身包含的任意标签或文本的额外字符串。这个字符串由宏直接回显。举个例子。在文本域的字段中，你能够将属性设置为 'rows="5" cols="60"' ，或者你可以传递样式信息，例如'style="border:1px solid silver"'

classOrStyle: for the showErrors macro, the name of the CSS class that the span tag wrapping each error will use. If no information is supplied (or the value is empty) then the errors will be wrapped in <b></b> tags.

classOrStyle：对于showErrors 宏，将使用 span标签包装每个错误的CSS类名称。如果没有提供信息（或值为空），那么错误信息将被包装的<b></b>标签

Examples of the macros are outlined below some in FTL and some in VTL. Where usage differences exist between the two languages, they are explained in the notes.

宏的示例概述如下，一些在FTL中，一些在VTL中。如果两种语言中存在差异。将在注释中说明

#### Input Fields

The formInput macro takes the path parameter (command.name) and an additional attributes parameter which is empty in the example above. The macro, along with all other form generation macros, performs an implicit spring bind on the path parameter. The binding remains valid until a new bind occurs so the showErrors macro doesn’t need to pass the path parameter again - it simply operates on whichever field a bind was last created for.

formInput 宏接受路径参数（command.name）和一个在上述例子中为空的额外属性。该宏以及所有其他表单生成的宏，在路径参数上执行隐式的Spring绑定。绑定保持有效，直到发生新的绑定，因此showErrors 不需要再次传递路径参数-它只是在 绑定最后创建的任意字段上运行

The showErrors macro takes a separator parameter (the characters that will be used to separate multiple errors on a given field) and also accepts a second parameter, this time a class name or style attribute. Note that FreeMarker is able to specify default values for the attributes parameter.

showErrors宏采用一个分隔参数（用于分隔给定的字段上多个错误的字符），并且接受第二个参数，这次是一个类名或样式属性。请注意，FreeMarker能为属性参数指定默认值

<@spring.formInput "command.name"/>

<@spring.showErrors "<br>"/>

Output is shown below of the form fragment generating the name field, and displaying a validation error after the form was submitted with no value in the field. Validation occurs through Spring’s Validation framework.

输出显示在生成名称字段的表单片段下面，并且在表单提交后字段中没有值的情况下显示验证错误。验证通过Spring的验证框架进行

The generated HTML looks like this:

生成的HTML如下所示：

Name:

<input type="text" name="name" value="">

<br>

<b>required</b>

<br>

<br>

The formTextarea macro works the same way as the formInput macro and accepts the same parameter list. Commonly, the second parameter (attributes) will be used to pass style information or rows and cols attributes for the textarea.

formTextarea 宏和formInput 宏的工作方式相同，并接受相同的参数列表。通常，第二个参数（attributes）用于传递文本域的样式信息或行和列属性

##### Selection Fields

Four selection field macros can be used to generate common UI value selection inputs in your HTML forms.

四个选择字段宏 能够 在HTML表单中 用于生成常用的UI值选择输入

* formSingleSelect
* formMultiSelect
* formRadioButtons
* formCheckboxes

Each of the four macros accepts a Map of options containing the value for the form field, and the label corresponding to that value. The value and the label can be the same.

四个宏中的每一个接受 包含表单字段值的选项的Map，以及与该值相对应的标签。值和标签可以是相同的

An example of radio buttons in FTL is below. The form backing object specifies a default value of 'London' for this field and so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'.

在FTL 中一个单选按钮的示例如下。表单支持对象为这个字段指定了一个默认值“London”，因此不需要验证。当表单呈现时，要选择的整个城市列表 在‘cityMap’名称下作为模型参考数据 提供

...

Town:

<@spring.formRadioButtons "command.address.town", cityMap, ""/><br><br>

This renders a line of radio buttons, one for each value in cityMap using the separator "". No additional attributes are supplied (the last parameter to the macro is missing). The cityMap uses the same String for each key-value pair in the map. The map’s keys are what the form actually submits as POSTed request parameters, map values are the labels that the user sees. In the example above, given a list of three well known cities and a default value in the form backing object, the HTML would be

这将呈现一行单选按钮。cityMap的每个值使用风格符“”。没有提供额外的属性（缺失宏的最后一个参数）。cityMap为map中的每一个键值对使用相同的字符。该map的键是这个表单实际提交的POST请求参数，map的值是用户看到的标签。在上面的例子中，给定三个已知城市的列表和一个表单支持对象的默认值。HTML将会是：

Town:

<input type="radio" name="address.town" value="London">London</input>

<input type="radio" name="address.town" value="Paris" checked="checked">Paris</input>

<input type="radio" name="address.town" value="New York">New York</input>

If your application expects to handle cities by internal codes for example, the map of codes would be created with suitable keys like the example below.

如果你的应用程序想要在内部代码处理城市，例如，通过合适的键创建代码映射，如下例所示：

**protected** Map<String, String> referenceData(HttpServletRequest request) **throws** Exception {

Map<String, String> cityMap = **new** LinkedHashMap<>();

cityMap.put("LDN", "London");

cityMap.put("PRS", "Paris");

cityMap.put("NYC", "New York");

Map<String, String> model = **new** HashMap<>();

model.put("cityMap", cityMap);

**return** model;

}

The code would now produce output where the radio values are the relevant codes but the user still sees the more user friendly city names.

代码将产生输出，其中单选按钮的值是相关的代码，但是用户仍旧能够看到根据用户友好的城市名称

Town:

<input type="radio" name="address.town" value="LDN">London</input>

<input type="radio" name="address.town" value="PRS" checked="checked">Paris</input>

<input type="radio" name="address.town" value="NYC">New York</input>

#### HTML escaping

Default usage of the form macros above will result in HTML tags that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your web.xml as used by Spring’s bind support. In order to make the tags XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model where they will be visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

上面表单宏的默认使用 将 生成 符合 HTML 4.01 的HTML标签， 这些标签使用 Spring绑定支持锁使用的web.xml中定义的HTML转义 的默认值，为了使标签XHTML 兼容或覆盖默认的HTML转义值，你可以在模板中指定另个变量（或在模板中可见的模型中）。在模板中指定它的的优点是，可以在模板处理的稍后 改变它们的不同的值，以在你表单中为不同的字段提供不同的行为

To switch to XHTML compliance for your tags, specify a value of 'true' for a model/context variable named xhtmlCompliant:

为了转换标签的XHTML 合规性，为名称为xhtmlCompliant的模型/内容变量指定一个true值

<#-- for FreeMarker -->

<#assign xhtmlCompliant = true in spring>

Any tags generated by the Spring macros will now be XHTML compliant after processing this directive.

在用这个指令处理之后，由Spring 指令生成的任何标签 将符合XHTML 规范

In similar fashion, HTML escaping can be specified per field:

以类似的方式。每个字段可以指定HTML转义

<-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>

<-- next field will use HTML escaping -->

<@spring.formInput "command.name"/>

<assign htmlEscape = false in spring>

<-- all future fields will be bound with HTML escaping off -->

### 1.12.5. JSP & JSTL

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views. Using JSP or JSTL is done using a normal view resolver defined in the WebApplicationContext. Furthermore, of course you need to write some JSPs that will actually render the view.

Spring为JSP和JSTL视图提供了一些开箱即用的解决方案。使用JSP或JSTL是 使用在WebApplicationContext中定义的一个普通的视图解析器完成的。此外，当然你需要写一些JSP，它将实际渲染视图

#### View resolvers

Just as with any other view technology you’re integrating with Spring, for JSPs you’ll need a view resolver that will resolve your views. The most commonly used view resolvers when developing with JSPs are the InternalResourceViewResolver and the ResourceBundleViewResolver. Both are declared in the WebApplicationContext:

就像其他任何与Spring集成的视图技术一样，对于JSP，你需要视图解析器来解析视图。当使用JSP开发时，最常使用的视图解析器是InternalResourceViewResolver 和ResourceBundleViewResolver。两者都在WebApplicationContext中声明

*<!-- the ResourceBundleViewResolver -->*

<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename" value="views"/>

</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):

welcome.(class)=org.springframework.web.servlet.view.JstlView

welcome.url=/WEB-INF/jsp/welcome.jsp

productList.(class)=org.springframework.web.servlet.view.JstlView

productList.url=/WEB-INF/jsp/productlist.jsp

As you can see, the ResourceBundleViewResolver needs a properties file defining the view names mapped to 1) a class and 2) a URL. With a ResourceBundleViewResolver you can mix different types of views using only one resolver.

正如你看到的。ResourceBundleViewResolver 需要一个属性文件来定义映射到1）类 和 2）URL的视图名称。使用ResourceBundleViewResolver ，你可以仅用一个解析器混合不同的视图类型

<bean id="viewResolver" 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>

The InternalResourceBundleViewResolver can be configured for using JSPs as described above. As a best practice, we strongly encourage placing your JSP files in a directory under the 'WEB-INF' directory, so there can be no direct access by clients.

InternalResourceBundleViewResolver 可以配置为使用上述描述的JSP。作为一个好的实践，我们强烈建议将你的JSP文件放在 'WEB-INF' 目录下的一个目录中。这样客户端就不能直接访问了

#### JSPs versus JSTL

When using the Java Standard Tag Library you must use a special view class, the JstlView, as JSTL needs some preparation before things such as the I18N features will work.

当使用Java标准标签库时，你必须使用特定的视图类，JSTLView，因为JSTL需要一些准备工作，例如I18N 特性才能起作用

#### Spring’s JSP tag library

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags haveHTML escaping features to enable or disable escaping of characters.

Spring提供了请求参数与命令对象的数据绑定，如前面章节描述的 。结合这些数据绑定功能，促进JSP页面的开发。Spring提供了一些标签，这些标签使得事情变得更加简单。所有Spring标签都有HTML转义特性，用来启用或禁用字符串转换

The spring.tld tag library descriptor (TLD) is included in the spring-webmvc.jar

spring.tld标签库描述符（TLD）包含在 spring-webmvc.jar中

#### Spring’s form tag library

As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

自版本2.0开始。Spring 在使用JSP和Spring Web MVC时 为处理表单元素 提供了一套完整的数据绑定识别标签。每个标签都支持其相应的HTML标签对应的属性集，使标签熟悉和直观的来使用，标签生成的HTML符合HTML 4.01/XHTML 1.0 规范

Unlike other form/input tag libraries, Spring’s form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As you will see in the following examples, the form tags make JSPs easier to develop, read and maintain.

不像其他的表单/输入 标签库，Spring的表单标签库与Spring Web MVC集成，使标签可以访问控制器处理的命令对象和引用数据。正如你在下面的示例中所看到的。表单标签使得JSP更容易开发、阅读和维护

Let’s go through the form tags and look at an example of how each tag is used. We have included generated HTML snippets where certain tags require further commentary.

让我们通过表单标签，看看如何使用每个标签的例子。 我们已经包含生成的HTML片段，其中某些标签需要进一步评论

#### Configuration

The form tag library comes bundled in spring-webmvc.jar. The library descriptor is called spring-form.tld.

表单库标签捆绑在 spring-webmvc.jar中，库描述称为 spring-form.tld

To use the tags from this library, add the following directive to the top of your JSP page:

要使用这个库的标签，添加一下指令到你JSP文件的顶部

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

where form is the tag name prefix you want to use for the tags from this library.

其中 form是要用于来自此库的标签的标签名称前缀

#### The form tag

This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding. It puts the command object in the PageContext so that the command object can be accessed by inner tags. All the other tags in this library are nested tags of the form tag.

这个标签呈现一个HTML‘表单’标签，并暴露一个绑定路径到内部标签进行绑定，它将命令对象放在PageContext ，以便命令对象能被内部标签访问。这个库中的所有其他标签是这个表单标签的嵌套标签

Let’s assume we have a domain object called User. It is a JavaBean with properties such as firstName and lastName. We will use it as the form backing object of our form controller which returns form.jsp. Below is an example of what form.jsp would look like:

让我们假定我们有一个称为User的域对象。它是具有firstName 和lastName属性的JavaBean。我们将使用它作为返回form.jsp的表单控制器的表单支持对象。下面是form.jsp的一个例子。

<form:form>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

The firstName and lastName values are retrieved from the command object placed in the PageContext by the page controller.

firstName 和lastName 值从页面控制器放置的PageContext 中的命令对象中检索

The generated HTML looks like a standard form:

生成的HTML看起来像一个标准的form

<form method="POST">

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value="Harry"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value="Potter"/></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form>

The preceding JSP assumes that the variable name of the form backing object is 'command'. If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

前面的JSP假定表单支持对象的变量名是'command'。 如果你已经把表单支持对象以另一个名字（绝对是最佳实践）放到模型中，那么你可以将表单绑定到指定的变量，如下所示：

<form:form modelAttribute="user">

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

#### The input tag

This tag renders an HTML 'input' tag using the bound value and type='text' by default.

这个标签默认使用绑定值和type ='text'呈现一个HTML'input'标签。

#### The checkbox tag

This tag renders an HTML 'input' tag with type 'checkbox'.

这个标签呈现一个‘CheckBox’类型的HTML ‘input’标签

Let’s assume our User has preferences such as newsletter subscription and a list of hobbies. Below is an example of the Preferences class:

让我们假设我们的用户有喜好，如通讯订阅和业余爱好列表。 以下是Preferences类的一个例子：

**public** **class** **Preferences** {

**private** **boolean** receiveNewsletter;

**private** String**[]** interests;

**private** String favouriteWord;

**public** **boolean** isReceiveNewsletter() {

**return** receiveNewsletter;

}

**public** **void** setReceiveNewsletter(**boolean** receiveNewsletter) {

this.receiveNewsletter = receiveNewsletter;

}

**public** String**[]** getInterests() {

**return** interests;

}

**public** **void** setInterests(String**[]** interests) {

this.interests = interests;

}

**public** String getFavouriteWord() {

**return** favouriteWord;

}

**public** **void** setFavouriteWord(String favouriteWord) {

this.favouriteWord = favouriteWord;

}

}

The form.jsp would look like:

form.jsp 看起来像这样：

<form:form>

<table>

<tr>

<td>Subscribe to newsletter?:</td>

<%-- Approach 1: Property is of type java.lang.Boolean --%>

<td><form:checkbox path="preferences.receiveNewsletter"/></td>

</tr>

<tr>

<td>Interests:</td>

<%-- Approach 2: Property is of an array or of type java.util.Collection --%>

<td>

Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>

Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>

Defence Against the Dark Arts: <form:checkbox path="preferences.interests" value="Defence Against the Dark Arts"/>

</td>

</tr>

<tr>

<td>Favourite Word:</td>

<%-- Approach 3: Property is of type java.lang.Object --%>

<td>

Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>

</td>

</tr>

</table>

</form:form>

There are 3 approaches to the checkbox tag which should meet all your checkbox needs.

有三种方法的CheckBox标签，应该满足你所有复选框的需要

* Approach One - When the bound value is of type java.lang.Boolean, the input(checkbox) is marked as 'checked' if the bound value is true. The value attribute corresponds to the resolved value of the setValue(Object) value property.

方法1：当绑定值是 java.lang.Boolean类型时，如果绑定值是true，input(checkbox)被标记为'checked' ，value属性对应 setValue(Object) 值属性解析的值

* Approach Two - When the bound value is of type array or java.util.Collection, the input(checkbox) is marked as 'checked' if the configured setValue(Object) value is present in the bound Collection.

方法2：当绑定类型为array 或者java.util.Collection时， 如果配置的setValue(Object)值存在于绑定的Collection， input(checkbox)标记为'checked'

* Approach Three - For any other bound value type, the input(checkbox) is marked as 'checked' if the configured setValue(Object) is equal to the bound value.

方法3：对于其他绑定值类型，如果配置的setValue(Object)等于绑定的值， input(checkbox)标记为'checked'

Note that regardless of the approach, the same HTML structure is generated. Below is an HTML snippet of some checkboxes:

请注意，不管采用哪种方法，都将生成相同的HTML结构，下面是一些checkboxes的HTML片段

<tr>

<td>Interests:</td>

<td>

Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox" value="Defence Against the Dark Arts"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

</td>

</tr>

What you might not expect to see is the additional hidden field after each checkbox. When a checkbox in an HTML page is not checked, its value will not be sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML in order for Spring form data binding to work. The checkbox tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore ("\_") for each checkbox. By doing this, you are effectively telling Spring that "the checkbox was visible in the form and I want my object to which the form data will be bound to reflect the state of the checkbox no matter what".

您可能不希望看到的是每个复选框后的附加隐藏域。如果未选中 html 页中的复选框, 则在提交表单后, 它的值将不会作为 HTTP 请求参数的一部分发送到服务器, 因此, 为了使 Spring 表单数据绑定工作, 我们需要在 html 中解决这个问题。复选框标记遵循现有的 Spring 约定, 其中包括每个复选框都以下划线 ("\_") 为前缀的隐藏参数。通过这样做, 您可以有效地告诉 Spring "该复选框在表单中是可见的, 并且我希望将表单数据绑定到其上的对象能够反映复选框的状态, 而不管它是什么"。

#### The checkboxes tag

This tag renders multiple HTML 'input' tags with type 'checkbox'.

这个标签使用“复选框”来呈现多个HTML“输入”标签。

Building on the example from the previous checkbox tag section. Sometimes you prefer not to have to list all the possible hobbies in your JSP page. You would rather provide a list at runtime of the available options and pass that in to the tag. That is the purpose of the checkboxes tag. You pass in an Array, a List or a Map containing the available options in the "items" property. Typically the bound property is a collection so it can hold multiple values selected by the user. Below is an example of the JSP using this tag:

基于上一个复选框标签部分的示例。 有时候，您不希望在JSP页面中列出所有可能的兴趣爱好。 您宁愿在运行时提供可用选项列表，并将其传递给标记。 这是复选框标签的用途。 你传递一个Array，一个List或者一个包含“items”属性中的可用选项的Map。 通常绑定属性是一个集合，所以它可以保存用户选择的多个值。 以下是使用此标记的JSP示例：

<form:form>

<table>

<tr>

<td>Interests:</td>

<td>

<%-- Property is of an array or of type java.util.Collection --%>

<form:checkboxes path="preferences.interests" items="${interestList}"/>

</td>

</tr>

</table>

</form:form>

This example assumes that the "interestList" is a List available as a model attribute containing strings of the values to be selected from. In the case where you use a Map, the map entry key will be used as the value and the map entry’s value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

这个例子假设“interestList”是一个可用作模型属性的List变量，包含要从中选择的值的字符串。 在使用Map的情况下，Map输入键将被用作值，并且Map条目的值将被用作要被显示的标签。 您也可以使用自定义对象，您可以在其中使用“itemValue”为值提供属性名称，使用“itemLabel”为该标签提供属性名称。

The radiobutton tag

This tag renders an HTML 'input' tag with type 'radio'.

这个标签呈现一个类型为“radio”的HTML“输入”标签。

A typical usage pattern will involve multiple tag instances bound to the same property but with different values.

一个典型的使用模式将涉及 绑定到相同的属性但具有不同的值 多个标签实例 。

<tr>

<td>Sex:</td>

<td>

Male: <form:radiobutton path="sex" value="M"/> <br/>

Female: <form:radiobutton path="sex" value="F"/>

</td>

</tr>

#### The radiobuttons tag

This tag renders multiple HTML 'input' tags with type 'radio'.

这个标签呈现多个类型为“radio”的HTML“输入”标签。

Just like the checkboxes tag above, you might want to pass in the available options as a runtime variable. For this usage you would use the radiobuttons tag. You pass in an Array, a List or a Map containing the available options in the "items" property. In the case where you use a Map, the map entry key will be used as the value and the map entry’s value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

就像上面的复选框标签一样，您可能想要将可用选项作为运行时变量传递。 对于这种用法，您可以使用radiobuttons 标签。 你传递一个Array，一个List或者一个“items”属性中包含可用选项的Map。 在使用Map的情况下，Map输入键将被用作值，并且Map条目的值将被用作要被显示的标签。 您也可以使用自定义对象，您可以在其中使用“itemValue”为值提供属性名称，使用“itemLabel”为该标签提供属性名称。

<tr>

<td>Sex:</td>

<td><form:radiobuttons path="sex" items="${sexOptions}"/></td>

</tr>

#### The password tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

此标记使用绑定值呈现类型为“password”的HTML“输入”标记。

<tr>

<td>Password:</td>

<td>

<form:password path="password"/>

</td>

</tr>

Please note that by default, the password value is not shown. If you do want the password value to be shown, then set the value of the 'showPassword' attribute to true, like so.

请注意，默认情况下，密码值不显示。 如果您想要显示密码值，则将“showPassword”属性的值设置为true，就像这样。

<tr>

<td>Password:</td>

<td>

<form:password path="password" value="^76525bvHGq" showPassword="true"/>

</td>

</tr>

#### The select tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested option and options tags.

这个标签呈现一个HTML“select”元素。 它支持数据绑定到选定的选项以及使用嵌套的选项和选项标签。

Let’s assume a User has a list of skills.

我们假设一个用户有一个技能列表。

<tr>

<td>Skills:</td>

<td><form:select path="skills" items="${skills}"/></td>

</tr>

If the User’s skill were in Herbology, the HTML source of the 'Skills' row would look like:

如果用户的技能在Herbology中，则“'Skills' ”行的HTML源代码将如下所示:

<tr>

<td>Skills:</td>

<td>

<select name="skills" multiple="true">

<option value="Potions">Potions</option>

<option value="Herbology" selected="selected">Herbology</option>

<option value="Quidditch">Quidditch</option>

</select>

</td>

</tr>

#### The option tag

This tag renders an HTML 'option'. It sets 'selected' as appropriate based on the bound value.

这个标签呈现一个HTML“'option'”。 它根据边界值适当设置“'selected' ”。

<tr>

<td>House:</td>

<td>

<form:select path="house">

<form:option value="Gryffindor"/>

<form:option value="Hufflepuff"/>

<form:option value="Ravenclaw"/>

<form:option value="Slytherin"/>

</form:select>

</td>

</tr>

If the User’s house was in Gryffindor, the HTML source of the 'House' row would look like:

如果用户的房屋位于Gryffindor，则“House”行的HTML源代码将如下所示：

<tr>

<td>House:</td>

<td>

<select name="house">

<option value="Gryffindor" selected="selected">Gryffindor</option>

<option value="Hufflepuff">Hufflepuff</option>

<option value="Ravenclaw">Ravenclaw</option>

<option value="Slytherin">Slytherin</option>

</select>

</td>

</tr>

#### The options tag

This tag renders a list of HTML 'option' tags. It sets the 'selected' attribute as appropriate based on the bound value.

这个标签呈现HTML'option' 标签的列表。 它根据绑定值适当地设置“selected”属性。

<tr>

<td>Country:</td>

<td>

<form:select path="country">

<form:option value="-" label="--Please Select"/>

<form:options items="${countryList}" itemValue="code" itemLabel="name"/>

</form:select>

</td>

</tr>

If the User lived in the UK, the HTML source of the 'Country' row would look like:

如果用户居住在UK，“'Country' ”行的HTML源代码将如下所示：

<tr>

<td>Country:</td>

<td>

<select name="country">

<option value="-">--Please Select</option>

<option value="AT">Austria</option>

<option value="UK" selected="selected">United Kingdom</option>

<option value="US">United States</option>

</select>

</td>

</tr>

As the example shows, the combined usage of an option tag with the options tag generates the same standard HTML, but allows you to explicitly specify a value in the JSP that is for display only (where it belongs) such as the default string in the example: "-- Please Select".

如示例所示，option 标记与option 记的组合使用会生成相同的标准HTML，但允许您在JSP中明确指定一个仅用于显示的值（其所属的位置），例如示例中的默认值：“ - 请选择”。

The items attribute is typically populated with a collection or array of item objects. itemValue and itemLabel simply refer to bean properties of those item objects, if specified; otherwise, the item objects themselves will be stringified. Alternatively, you may specify a Map of items, in which case the map keys are interpreted as option values and the map values correspond to option labels. If itemValue and/or itemLabel happen to be specified as well, the item value property will apply to the map key and the item label property will apply to the map value.

items属性通常由项目对象的集合或数组填充。 itemValue和itemLabel只需引用这些项目对象的bean属性（如果指定的话）; 否则，项目对象本身将被串化。 或者，您可以指定项目的Map，在这种情况下，如果itemValue和/或itemLabel恰好被指定，Map键值将被解释为选项值，Map值对应于选项标签。

#### The textarea tag

This tag renders an HTML 'textarea'.

这个标签呈现一个HTML'textarea'。

<tr>

<td>Notes:</td>

<td><form:textarea path="notes" rows="3" cols="20"/></td>

<td><form:errors path="notes"/></td>

</tr>

#### The hidden tag

This tag renders an HTML 'input' tag with type 'hidden' using the bound value. To submit an unbound hidden value, use the HTML input tag with type 'hidden'.

这个标签使用绑定值呈现一个类型为“hidden”的HTML“input”标签。 要提交未绑定的隐藏值，请使用类型为“'hidden'”的HTML输入标签。

<form:hidden path="house"/>

If we choose to submit the 'house' value as a hidden one, the HTML would look like:

如果我们选择将“house”值作为隐藏值提交，则HTML将如下所示:

<input name="house" type="hidden" value="Gryffindor"/>

#### The errors tag

This tag renders field errors in an HTML 'span' tag. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

此标记在HTML“span”标记中呈现字段错误。它提供对在控制器中创建的错误或由与控制器关联的任何验证器创建的错误的访问。

Let’s assume we want to display all error messages for the firstName and lastName fields once we submit the form. We have a validator for instances of the User class called UserValidator.

假设我们希望在提交表单后显示firstName和lastName字段的所有错误消息。 我们有一个名为UserValidator的User类实例的验证器。

**public** **class** **UserValidator** **implements** Validator {

**public** **boolean** supports(Class candidate) {

**return** User.class.isAssignableFrom(candidate);

}

**public** **void** validate(Object obj, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required", "Field is required.");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field is required.");

}

}

The form.jsp would look like:

form.jsp看起来像这样：

<form:form>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

<%-- Show errors for firstName field --%>

<td><form:errors path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

<%-- Show errors for lastName field --%>

<td><form:errors path="lastName"/></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

If we submit a form with empty values in the firstName and lastName fields, this is what the HTML would look like:

如果我们提交了一个firstName 和lastName字段为空表单， 这就是HTML的样子:

<form method="POST">

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value=""/></td>

<%-- Associated errors to firstName field displayed --%>

<td><span name="firstName.errors">Field is required.</span></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value=""/></td>

<%-- Associated errors to lastName field displayed --%>

<td><span name="lastName.errors">Field is required.</span></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form>

What if we want to display the entire list of errors for a given page? The example below shows that the errors tag also supports some basic wildcarding functionality.

如果我们想要显示给定页面的整个错误列表呢？ 下面的例子显示了错误标签还支持一些基本的通配符功能。

* path="\*" - displays all errors

path="\*" --显示所有错误

* path="lastName" - displays all errors associated with the lastName field

path="lastName" -- 显示与lastName字段关联的所有错误

* if path is omitted - object errors only are displayed

如果path省略--仅显示对象错误

The example below will display a list of errors at the top of the page, followed by field-specific errors next to the fields:

以下示例将在页面顶部显示错误列表，然后在字段旁边显示特定于字段的错误：

<form:form>

<form:errors path="\*" cssClass="errorBox"/>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

<td><form:errors path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

<td><form:errors path="lastName"/></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

The HTML would look like:

HTML看起来像：

<form method="POST">

<span name="\*.errors" class="errorBox">Field is required.<br/>Field is required.</span>

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value=""/></td>

<td><span name="firstName.errors">Field is required.</span></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value=""/></td>

<td><span name="lastName.errors">Field is required.</span></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form>

The spring-form.tld tag library descriptor (TLD) is included in the spring-webmvc.jar.

spring-form.tld标记库描述符（TLD）包含在spring-webmvc.jar中

#### HTTP method conversion

A key principle of REST is the use of the Uniform Interface. This means that all resources (URLs) can be manipulated using the same four HTTP methods: GET, PUT, POST, and DELETE. For each method, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that is has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or simply do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form). This latter trick is what Spring’s HiddenHttpMethodFilter does. This filter is a plain Servlet Filter and therefore it can be used in combination with any web framework (not just Spring MVC). Simply add this filter to your web.xml, and a POST with a hidden \_method parameter will be converted into the corresponding HTTP method request.

REST的一个关键原则是使用统一接口。这意味着所有资源（URL）都可以使用相同的四种HTTP方法进行操作：GET，PUT，POST和DELETE。对于每种方法，HTTP规范都定义了确切的语义。例如，一个GET应该总是一个安全的操作，意思是没有任何副作用，PUT或DELETE应该是幂等的，这意味着你可以重复这些操作，但最终的结果应该是一样的。虽然HTTP定义了这四种方法，但HTML仅支持两种：GET和POST。幸运的是，有两种可能的解决方法：您可以使用JavaScript来执行PUT或DELETE，或者简单地使用'real'方法作为附加参数（在HTML表单中作为隐藏输入字段建模）执行POST。后一个技巧就是Spring的HiddenHttpMethodFilter所做的。这个过滤器是一个普通的Servlet过滤器，因此它可以与任何Web框架（不只是Spring MVC）结合使用。只需将此过滤器添加到您的web.xml中，并将带有隐藏\_method参数的POST转换为相应的HTTP方法请求。

To support HTTP method conversion the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet taken from the updated Petclinic sample

为了支持 http 方法转换, Spring MVC 表单标签被更新以支持设置 http 方法。例如,从更新的 Petclinic 示例中抽取的以下代码段

<form:form method="delete">

<p class="submit"><input type="submit" value="Delete Pet"/></p>

</form:form>

This will actually perform an HTTP POST, with the 'real' DELETE method hidden behind a request parameter, to be picked up by the HiddenHttpMethodFilter, as defined in web.xml:

这实际上会执行一个HTTP POST，在请求参数后面隐藏着'real' DELETE方法，由web.xml中定义的HiddenHttpMethodFilter获取：

<filter>

<filter-name>httpMethodFilter</filter-name>

<filter-**class**>**org**.springframework.web.filter.HiddenHttpMethodFilter</filter-**class**>

</**filter**>

<filter-mapping>

<filter-name>httpMethodFilter</filter-name>

<servlet-name>petclinic</servlet-name>

</filter-mapping>

The corresponding @Controller method is shown below:

相应的@Controller方法如下所示：

@RequestMapping(method = RequestMethod.DELETE)

**public** String deletePet(@PathVariable **int** ownerId, @PathVariable **int** petId) {

this.clinic.deletePet(petId);

**return** "redirect:/owners/" + ownerId;

}

#### HTML5 tags

Starting with Spring 3, the Spring form tag library allows entering dynamic attributes, which means you can enter any HTML5 specific attributes.

从Spring 3开始，Spring表单标签库允许输入动态属性，这意味着您可以输入任何HTML5特定属性。

In Spring 3.1, the form input tag supports entering a type attribute other than 'text'. This is intended to allow rendering new HTML5 specific input types such as 'email', 'date', 'range', and others. Note that entering type='text' is not required since 'text' is the default type.

在Spring 3.1中，表单输入标签支持输入“'text'”以外的类型属性。 这是为了允许呈现新的HTML5特定的输入类型，如“电子邮件”，“日期”，“范围”等。 请注意，输入type ='text'不是必需的，因为'text'是默认的类型。

### 1.12.6. Script views

It is possible to integrate any templating library running on top of a JSR-223 script engine in web applications using Spring. The following describes in a broad way how to do this. The script engine must implement both ScriptEngine and Invocable interfaces.

在使用 Spring 的 web 应用程序中, 可以将运行在 JSR-223 脚本引擎之上的任何模板库集成在一起。以下以广泛的方式描述如何做到这一点。 脚本引擎必须实现ScriptEngine和Invocable接口。

It has been tested with:

它已经过测试：

* Handlebars running on Nashorn
* Mustache running on Nashorn
* React running on Nashorn
* EJS running on Nashorn
* ERB running on JRuby
* String templates running on Jython

#### Requirements

To be able to use script templates integration, you need to have available in your classpath the script engine:

为了能够使用脚本模板集成，您需要在类路径中提供脚本引擎：

* Nashorn Javascript engine is provided builtin with Java 8+. Using the latest update release available is highly recommended.

Java 8+提供了内置的Nashorn Javascript引擎。强烈推荐使用最新的更新版本。

* Rhino Javascript engine is provided builtin with Java 6 and Java 7. Please notice that using Rhino is not recommended since it does not support running most template engines.

Java 6 和Java 7中提供了内置Rhino Javascript 引擎， 请注意，不推荐使用Rhino，因为它不支持运行大多数模板引擎。

* JRuby dependency should be added in order to get Ruby support.

为了获得Ruby支持，应该添加JRuby依赖关系。

* Jython dependency should be added in order to get Python support.

为了获得Python支持，应添加Jython依赖关系。

You should also need to add dependencies for your script based template engine. For example, for Javascript you can use WebJars to add Maven/Gradle dependencies in order to make your javascript libraries available in the classpath.

您还需要为基于脚本的模板引擎添加依赖项。例如，对于Javascript，您可以使用WebJars添加Maven / Gradle依赖项，以便使Javascript库在类路径中可用。

#### Script templates

To be able to use script templates, you have to configure it in order to specify various parameters like the script engine to use, the script files to load and what function should be called to render the templates. This is done thanks to a ScriptTemplateConfigurer bean and optional script files.

为了能够使用脚本模板, 您必须对其进行配置, 以便指定各种参数, 如要使用的脚本引擎、要加载的脚本文件以及应调用哪些函数来呈现模板。这是由 ScriptTemplateConfigurer bean 和可选的脚本文件完成的。

For example, in order to render Mustache templates thanks to the Nashorn Javascript engine provided with Java 8+, you should declare the following configuration:

例如，为了使用Java 8+提供的Nashorn Javascript引擎来呈现Mustache模板，您应该声明以下配置：

@Configuration@EnableWebMvc

**public** **class** **MustacheConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureViewResolvers(ViewResolverRegistry registry) {

registry.scriptTemplate();

}

@Bean

**public** ScriptTemplateConfigurer configurer() {

ScriptTemplateConfigurer configurer = **new** ScriptTemplateConfigurer();

configurer.setEngineName("nashorn");

configurer.setScripts("mustache.js");

configurer.setRenderObject("Mustache");

configurer.setRenderFunction("render");

**return** configurer;

}

}

The XML counterpart using MVC namespace is:

使用MVC名称空间的XML对象是：

<mvc:annotation-driven/>

<mvc:view-resolvers>

<mvc:script-template/>

</mvc:view-resolvers>

<mvc:script-template-configurer engine-name="nashorn" render-object="Mustache" render-function="render">

<mvc:script location="mustache.js"/>

</mvc:script-template-configurer>

The controller is exactly what you should expect:

你所期望的控制器：

@Controller

**public** **class** **SampleController** {

@RequestMapping

**public** ModelAndView test() {

ModelAndView mav = **new** ModelAndView();

mav.addObject("title", "Sample title").addObject("body", "Sample body");

mav.setViewName("template.html");

**return** mav;

}

}

And the Mustache template is:

Mustache模板是:

<html>

<head>

<title>{{title}}</title>

</head>

<body>

<p>{{body}}</p>

</body>

</html>

The render function is called with the following parameters:

使用以下参数调用渲染函数：

* String template: the template content

String template: 模板内容

* Map model: the view model

Map model: 视图模型

* String url: the template url (since 4.2.2)

String url: 模板url（自4.2.2开始）

Mustache.render() is natively compatible with this signature, so you can call it directly.

Mustache.render()是与此签名相兼容的，因此您可以直接调用它。

If your templating technology requires some customization, you may provide a script that implements a custom render function. For example, Handlerbars needs to compile templates before using them, and requires a polyfill in order to emulate some browser facilities not available in the server-side script engine.

如果您的模板技术需要一些定制，您可以提供一个实现自定义渲染功能的脚本。 例如，Handlerbars在使用模板之前需要编译模板，并且需要使用polyfill来模拟服务器端脚本引擎中不可用的某些浏览器工具。

@Configuration

@EnableWebMvc

**public** **class** **MustacheConfig** **implements** WebMvcConfigurer {

@Override

**public** **void** configureViewResolvers(ViewResolverRegistry registry) {

registry.scriptTemplate();

}

@Bean

**public** ScriptTemplateConfigurer configurer() {

ScriptTemplateConfigurer configurer = **new** ScriptTemplateConfigurer();

configurer.setEngineName("nashorn");

configurer.setScripts("polyfill.js", "handlebars.js", "render.js");

configurer.setRenderFunction("render");

configurer.setSharedEngine(false);

**return** configurer;

}

}

polyfill.js only defines the window object needed by Handlebars to run properly:

polyfill.js只定义Handlebars正确运行所需的窗口对象：

**var** window = {};

This basic render.js implementation compiles the template before using it. A production ready implementation should also store and reused cached templates / pre-compiled templates. This can be done on the script side, as well as any customization you need (managing template engine configuration for example).

这个基本的render . js实现在使用模板之前对其进行编译。生产就绪实现还应存储和重用缓存模板/预编译模板。这可以在脚本端完成，也可以在需要的任何自定义(例如管理模板引擎配置)上完成。

**function** **render**(template, model) {

**var** compiledTemplate = Handlebars.compile(template);

**return** compiledTemplate(model);

}

### 1.12.7. XML Marshalling

### 1.12.8. Tiles

### 1.12.9. XSLT

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring Web MVC application.

XSLT是XML的转换语言，在Web应用程序中作为视图技术而广受欢迎。 如果您的应用程序自然处理XML，或者您的模型可以轻松转换为XML，则XSLT可以作为视图技术的不错选择。 以下部分显示如何将XML文档生成为模型数据，并使用Spring Web MVC应用程序中的XSLT进行转换。

This example is a trivial Spring application that creates a list of words in the Controller and adds them to the model map. The map is returned along with the view name of our XSLT view.

这个例子是一个普通的Spring应用程序，它在Controller中创建一个单词列表并将它们添加到模型映射中。 该地图与我们的XSLT视图的视图名称一起返回。

#### Beans

Configuration is standard for a simple Spring application. The MVC configuration has to define a XsltViewResolver bean and regular MVC annotation configuration.

配置是一个简单的Spring应用程序的标准。 MVC配置必须定义一个XsltViewResolver bean和常规的MVC注释配置。

@EnableWebMvc

@ComponentScan

@Configuration

**public** **class** **WebConfig** **implements** WebMvcConfigurer {

@Bean

**public** XsltViewResolver xsltViewResolver() {

XsltViewResolver viewResolver = **new** XsltViewResolver();

viewResolver.setPrefix("/WEB-INF/xsl/");

viewResolver.setSuffix(".xslt");

**return** viewResolver;

}

}

And we need a Controller that encapsulates our word generation logic.

我们需要一个封装我们的词生成逻辑的控制器。

#### Controller

The controller logic is encapsulated in a @Controller class, with the handler method being defined like so…​

控制器逻辑被封装在一个@Controller类中，处理器方法被定义成这样...

@Controller

**public** **class** **XsltController** {

@RequestMapping("/")

**public** String home(Model model) **throws** Exception {

Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();

Element root = document.createElement("wordList");

List<String> words = Arrays.asList("Hello", "Spring", "Framework");

**for** (String word : words) {

Element wordNode = document.createElement("word");

Text textNode = document.createTextNode(word);

wordNode.appendChild(textNode);

root.appendChild(wordNode);

}

model.addAttribute("wordList", root);

**return** "home";

}

}

So far we’ve only created a DOM document and added it to the Model map. Note that you can also load an XML file as a Resource and use it instead of a custom DOM document.

到目前为止，我们只创建了一个DOM文档并将其添加到Model地图中。 请注意，您也可以将XML文件作为资源加载，并使用它来代替自定义的DOM文档。

Of course, there are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

当然，有些软件包可以自动“控制”一个对象图形，但在Spring中，您可以完全灵活地根据您的选择以您的模型创建DOM。 这防止了XML的转换在模型数据的结构中扮演太重要的角色，这在使用工具来管理统一过程时是危险的。

Next, XsltViewResolver will resolve the "home" XSLT template file and merge the DOM document into it to generate our view.

接下来，XsltViewResolver将解析“home”XSLT模板文件并将DOM文档合并到其中以生成我们的视图。

#### Transformation

Finally, the XsltViewResolver will resolve the "home" XSLT template file and merge the DOM document into it to generate our view. As shown in the XsltViewResolver configuration, XSLT templates live in the war file in the 'WEB-INF/xsl' directory and end with a "xslt" file extension.

最后，XsltViewResolver将解析“home”XSLT模板文件并将DOM文档合并到其中以生成我们的视图。 如XsltViewResolver配置所示，XSLT模板位于“WEB-INF / xsl”目录下的war文件中，并以“xslt”文件扩展名结尾。

<?xml version="1.0" encoding="utf-8"?><xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html" omit-xml-declaration="yes"/>

<xsl:template match="/">

<html>

<head><title>Hello!</title></head>

<body>

<h1>My First Words</h1>

<ul>

<xsl:apply-templates/>

</ul>

</body>

</html>

</xsl:template>

<xsl:template match="word">

<li><xsl:value-of select="."/></li>

</xsl:template>

</xsl:stylesheet>

This is rendered as:

这呈现为：

<html>

<head>

<META http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Hello!</title>

</head>

<body>

<h1>My First Words</h1>

<ul>

<li>Hello</li>

<li>Spring</li>

<li>Framework</li>

</ul>

</body>

</html>

### 1.12.10. Document views: PDF, Excel

#### Introduction

Returning an HTML page isn’t always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the correct content type to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.

返回一个HTML页面不总是用户查看模型输出的最好方式，Spring使得动模型数据动态生成PDF文档或Excel电子表格变得简单。该文档是视图，从具有正确内容类型的服务器流式传输到 (希望) 使客户端 PC 能够运行其电子表格或 PDF 查看器应用程序以进行响应。

In order to use Excel views, you need to add the Apache POI library to your classpath, and for PDF generation, the common iText 2.1.7 or its fork OpenPDF (e.g. OpenPDF 1.0.4).

为了使用Excel视图，您需要将Apache POI库添加到您的类路径中，而对于PDF生成，则需要添加常见的iText 2.1.7或其分支的OpenPDF（例如OpenPDF 1.0.4）

#### Configuration

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

基于文档的视图以与XSLT视图几乎相同的方式进行处理，下面几节通过演示如何调用XSLT示例中使用的相同控制器来呈现与PDF文档和Excel电子表格相同的模型 （也可以在Open Office中查看或操作）。

#### View definition

First, let’s amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier:

首先, 让我们修改视图. 属性文件 (或等价的 xml), 并为这两种文档类型添加简单的视图定义。现在, 整个文件的外观与前面显示的 XSLT 视图类似:

home.(class)=xslt.HomePage

home.stylesheetLocation=/WEB-INF/xsl/home.xslt

home.root=words

xl.(class)=excel.HomePage

pdf.(class)=pdf.HomePage

If you want to start with a template spreadsheet or a fillable PDF form to add your model data to, specify the location as the 'url' property in the view definition

如果您想从模板电子表格或可输入的PDF表单开始添加模型数据，请在视图定义中指定位置作为“url”属性

#### Controller

The controller code we’ll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

我们将使用的控制器代码与前面的XSLT示例完全相同，只是更改了视图的名称。当然，您可能很聪明，并根据URL参数或其他一些逻辑来选择，Spring确实非常擅长将视图与控制器分离!

#### Excel views

Exactly as we did for the XSLT example, we’ll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass of org.springframework.web.servlet.view.document.AbstractExcelView (for Excel files generated by POI) or org.springframework.web.servlet.view.document.AbstractJExcelView (for JExcelApi-generated Excel files) and implementing the buildExcelDocument() method.

就像我们为XSLT示例所做的那样，我们将继承合适的抽象类，以便在生成输出文档时实现自定义行为。 对于Excel，这包括编写org.springframework.web.servlet.view.document.AbstractExcelView的子类（用于由POI生成的Excel文件）或org.springframework.web.servlet.view.document.AbstractJExcelView（用于JExcelApi生成的Excel 文件）并执行buildExcelDocument（）方法。

Here’s the complete listing for our POI Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet:

以下是POI Excel视图的完整列表，其中显示了新电子表格的第一列的连续行中模型映射的单词列表：

**package** excel;

*// imports omitted for brevity*

**public** **class** **HomePage** **extends** AbstractExcelView {

**protected** **void** buildExcelDocument(Map model, HSSFWorkbook wb, HttpServletRequest req,

HttpServletResponse resp) **throws** Exception {

HSSFSheet sheet;

HSSFRow sheetRow;

HSSFCell cell;

*// Go to the first sheet*

*// getSheetAt: only if wb is created from an existing document*

*// sheet = wb.getSheetAt(0);*

sheet = wb.createSheet("Spring");

sheet.setDefaultColumnWidth((**short**) 12);

*// write a text at A1*

cell = getCell(sheet, 0, 0);

setText(cell, "Spring-Excel test");

List words = (List) model.get("wordList");

**for** (**int** i=0; i < words.size(); i++) {

cell = getCell(sheet, 2+i, 0);

setText(cell, (String) words.get(i));

}

}

}

And the following is a view generating the same Excel file, now using JExcelApi:

以下是使用JExcelApi生成相同Excel文件的视图：

**package** excel;

*// imports omitted for brevity*

**public** **class** **HomePage** **extends** AbstractJExcelView {

**protected** **void** buildExcelDocument(Map model, WritableWorkbook wb,

HttpServletRequest request, HttpServletResponse response) **throws** Exception {

WritableSheet sheet = wb.createSheet("Spring", 0);

sheet.addCell(**new** Label(0, 0, "Spring-Excel test"));

List words = (List) model.get("wordList");

**for** (**int** i = 0; i < words.size(); i++) {

sheet.addCell(**new** Label(2+i, 0, (String) words.get(i)));

}

}

}

Note the differences between the APIs. We’ve found that the JExcelApi is somewhat more intuitive, and furthermore, JExcelApi has slightly better image-handling capabilities. There have been memory problems with large Excel files when using JExcelApi however.

请注意API之间的差异。 我们发现JExcelApi更加直观，而且JExcelApi的图像处理能力稍好一些。 但是，使用JExcelApi时，大型Excel文件存在内存问题。

If you now amend the controller such that it returns xl as the name of the view ( return new ModelAndView("xl", map);) and run your application again, you should find that the Excel spreadsheet is created and downloaded automatically when you request the same page as before.

如果您现在修改控制器，使其返回xl作为视图的名称（返回新的ModelAndView（“xl”，map）;）并再次运行您的应用程序，您应该会发现Excel电子表格是自动创建和下载的 像以前一样请求相同的页面。

#### PDF views

The PDF version of the word list is even simpler. This time, the class extendsorg.springframework.web.servlet.view.document.AbstractPdfView and implements the buildPdfDocument() method as follows:

单词列表的PDF版本更简单。 这一次，这个类扩展了org.springframework.web.servlet.view.document.AbstractPdfView，并实现了buildPdfDocument（）方法，如下所示：

**package** pdf;

*// imports omitted for brevity*

**public** **class** **PDFPage** **extends** AbstractPdfView {

**protected** **void** buildPdfDocument(Map model, Document doc, PdfWriter writer,

HttpServletRequest req, HttpServletResponse resp) **throws** Exception {

List words = (List) model.get("wordList");

**for** (**int** i=0; i<words.size(); i++) {

doc.add( **new** Paragraph((String) words.get(i)));

}

}

}

Once again, amend the controller to return the pdf view with return new ModelAndView("pdf", map);, and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

再一次，修改控制器返回pdf视图返回新的ModelAndView（“pdf”，map），并重新加载您的应用程序中的URL。 这次应该出现一个PDF文档，列出模型图中的每个单词。

### 1.12.11. Feed views: RSS, Atom

### 1.12.12. Jackson

#### JSON

The MappingJackson2JsonView uses the Jackson library’s ObjectMapper to render the response content as JSON. By default, the entire contents of the model map (with the exception of framework-specific classes) will be encoded as JSON. For cases where the contents of the map need to be filtered, users may specify a specific set of model attributes to encode via the RenderedAttributes property. The extractValueFromSingleKeyModel property may also be used to have the value in single-key models extracted and serialized directly rather than as a map of model attributes.

MappingJackson2JsonView使用Jackson库的ObjectMapper将响应内容呈现为JSON。默认情况下，模型映射的全部内容（除了框架特定的类）将被编码为JSON。对于需要过滤地图内容的情况，用户可以指定一组特定的模型属性，通过RenderedAttributes属性进行编码。 extractValueFromSingleKeyModel属性也可用于直接提取和序列化单键模型中的值，而不是模型属性的映射。

JSON mapping can be customized as needed through the use of Jackson’s provided annotations. When further control is needed, a custom ObjectMapper can be injected through the ObjectMapper property for cases where custom JSON serializers/deserializers need to be provided for specific types.

JSON映射可以根据需要通过使用Jackson提供的注释来定制。当需要进一步控制时，对于需要为特定类型提供自定义JSON序列化器/反序列化器的情况，可以通过ObjectMapper属性注入自定义ObjectMapper。

JSONP is supported and automatically enabled when the request has a query parameter named jsonp or callback. The JSONP query parameter name(s) could be customized through the jsonpParameterNames property.

当请求有一个名为jsonp或callback的查询参数时，支持并自动启用JSONP。 JSONP查询参数名称可以通过jsonpParameterNames属性自定义。

#### XML

The MappingJackson2XmlView uses the Jackson XML extension's XmlMapper to render the response content as XML. If the model contains multiples entries, the object to be serialized should be set explicitly using the modelKey bean property. If the model contains a single entry, it will be serialized automatically.

MappingJackson2XmlView使用Jackson XML扩展的XmlMapper将响应内容呈现为XML。如果模型包含多个条目，则应使用modelKey bean属性显式设置要序列化的对象。如果模型包含单个条目，则会自动序列化。

XML mapping can be customized as needed through the use of JAXB or Jackson’s provided annotations. When further control is needed, a custom XmlMapper can be injected through the ObjectMapper property for cases where custom XML serializers/deserializers need to be provided for specific types.

可以根据需要通过使用JAXB或Jackson提供的注释来定制XML映射。当需要进一步控制时，对于需要为特定类型提供自定义XML序列化器/反序列化器的情况，可以通过ObjectMapper属性注入自定义的XmlMapper。