[**Spring 4 官方文档学习（十一）Web MVC 框架**](http://www.cnblogs.com/larryzeal/p/6001000.html)

1. [介绍Spring Web MVC 框架](http://www.cnblogs.com/larryzeal/p/6001000.html#c1)
   1. [Spring Web MVC的特性](http://www.cnblogs.com/larryzeal/p/6001000.html#c1-1)
   2. [其他MVC实现的可插拔性](http://www.cnblogs.com/larryzeal/p/6001000.html#c1-2)
2. [DispatcherServlet](http://www.cnblogs.com/larryzeal/p/6001000.html#c2)
   1. [在WebApplicationContext中的特殊的bean types](http://www.cnblogs.com/larryzeal/p/6001000.html#c2-1)
   2. [默认的DispatcherServlet配置](http://www.cnblogs.com/larryzeal/p/6001000.html#c2-2)
   3. [DispatcherServlet处理顺序](http://www.cnblogs.com/larryzeal/p/6001000.html#c2-3)
3. [实现Controller](http://www.cnblogs.com/larryzeal/p/6001000.html#c3)
   1. [使用@Controller定义一个Controller](http://www.cnblogs.com/larryzeal/p/6001000.html#c3-1)
   2. [使用@RequestMapping映射requests](http://www.cnblogs.com/larryzeal/p/6001000.html#c3-2)
   3. [定义@RequestMapping handler methods](http://www.cnblogs.com/larryzeal/p/6001000.html#c3-3)
   4. [异步请求的处理](http://www.cnblogs.com/larryzeal/p/6001000.html#c3-4)
   5. [测试controllers](http://www.cnblogs.com/larryzeal/p/6001000.html#c3-5)
4. [Handler mappings](http://www.cnblogs.com/larryzeal/p/6001000.html#c4)
   1. [使用HandlerInterceptor拦截requests](http://www.cnblogs.com/larryzeal/p/6001000.html#c4-1)
5. [resolving views 解析视图](http://www.cnblogs.com/larryzeal/p/6128312.html)
6. [Spring 4 官方文档学习（十一）Web MVC 框架之Flash Attributes](http://www.cnblogs.com/larryzeal/p/6128456.html)
7. [Spring 4 官方文档学习（十一）Web MVC 框架之URI Builder](http://www.cnblogs.com/larryzeal/p/6131664.html)
8. [Spring 4 官方文档学习（十一）Web MVC 框架之locales](http://www.cnblogs.com/larryzeal/p/6132244.html)
9. [Spring 4 官方文档学习（十一）Web MVC 框架之themes](http://www.cnblogs.com/larryzeal/p/6132881.html)
10. [Spring 4 官方文档学习（十一）Web MVC 框架之multipart（文件上传）支持](http://www.cnblogs.com/larryzeal/p/6139470.html)
11. [Spring 4 官方文档学习（十一）Web MVC 框架之异常处理](http://www.cnblogs.com/larryzeal/p/6159631.html)
12. [Spring 4 官方文档学习（十一）Web MVC 框架之约定优于配置](http://www.cnblogs.com/larryzeal/p/6160030.html)
13. [Spring 4 官方文档学习（十一）Web MVC 框架之HTTP caching support](http://www.cnblogs.com/larryzeal/p/6160361.html)
14. [Spring 4 官方文档学习（十一）Web MVC 框架之编码式Servlet容器初始化](http://www.cnblogs.com/larryzeal/p/6160426.html)
15. [Spring 4 官方文档学习（十一）Web MVC 框架之配置Spring MVC](http://www.cnblogs.com/larryzeal/p/6160762.html)

**1、介绍Spring Web MVC 框架**

Spring Web MVC 框架是围绕**DispatcherServlet**设计的，所谓DispatcherServlet就是将请求分发到handler，需要有配置好的handler映射、视图解析、本地化、时区、theme解决方案、还有上传文件的支持。默认的handler是基于@Controller和@RequestMapping注解。自Spring 3.0 起，@Controller注解还能用于RESTful，需要配合@PathVariable以及其他特性。

Spring Web MVC 和 Spring的设计准则是“对扩展开放，对修改关闭”--可以扩展，不能修改。

Spring Web MVC中核心类的一些方法被标记为final。开发者不能覆盖这些方法以提供自己的行为。这不是任性，而是遵从设计准则。

在Spring MVC中，你不能为final方法添加advice。

在Spring Web MVC中，你可以使用任何对象作为**命令对象**或者**form-backing对象**；你不需要实现框架的特定接口或者基类。Spring的数据绑定是高度弹性的：例如，它将类型错误匹配视为校验错误，而非系统错误，从而可被应用evaluate。

Spring的view resolution是非常弹性的。Controller负责准备一个model Map，其中带有数据，还负责挑选一个view name -- 也可以直接写到响应流而完成一个请求。view name resolution是高度可定制的，使用文件扩展名或者Accept header content type negotiation，通过bean names、properties文件、或者，甚至一个自定义的ViewResolver实现。model（MVC中的M）是一个Map接口，允许view技术的完全抽象。你可以直接集成基于模板的rendering技术，例如JSP、Velocity和Freemarker、或者直接生成XML、JSON、Atom、以及很多其他内容类型。model Map会被简单的变成合适的格式，如JSP请求属性、如Velocity模板模型。

**1.1、Spring Web MVC的特性**

Spring Web Flow

目标是成为管理web应用页面流程的最佳解决方案。

SWF集成了现有的框架，如Spring MVC和JSF，在Servlet和Portlet环境中。如果你有一个业务process（或多个）想从conversational model中受益（与纯request model相比），那么SWF就是这种解决方案。

SWF允许你捕获logical page flows，将其作为self-contained modules，可以在不同环境下复用，因此，这是构建web应用模块的理想方式，能够引导用户-- 使用驱动业务processes的controlled navigations。

Spring的web模块包含许多独特的web支持特性：

* 角色完全隔离。 每个角色-- controller、validator、command object、form object、model object、DispatcherServlet、handler mapping、view resolver、等等 -- 都能使用特殊的对象来实现。
* 框架和应用classes的强大且条理清晰的配置。 这个配置能力包括不同contexts中的简单引用，例如从web controllers到业务对象和validators的引用。
* 适用能力、非侵入式、弹性。 可以定义任何需要的controller 方法签名，针对特定的情景 可能需要使用某个参数注解（如@RequestParam、@RequestHeader、@PathVariable等等）。
* 可复用的业务代码，不需要重复。 将现有业务对象作为命令对象或者form对象，而非将它们镜像到一个扩展类。
* 可定制的绑定和校验。Type mismatches as application-level validation errors that keep the offending value, localized date and number binding, and so on instead of String-only form objects with manual parsing and conversion to business objects.
* 可定制的handler mapping和view resolution。 handler mapping 和 view resolution策略：从简单的基于URL的配置，到复杂的构建目的的解决方案策略。相比只使用一种技术的web MVC框架来说，Spring更弹性
* 弹性的模型传输。 使用name/value Map的模型传输支持与任意view技术的简单集成。
* 可定制的locale、time zone和theme resolution，支持JSP（使用/不使用Spring标签），支持JSTL，支持Velocity--不需要额外的桥梁，等等。
* 简单且强大的JSP标签库--Spring标签库，支持的特性包括数据绑定和themes。
* JSP form标签库，Spring 2.0引入的， 使得在JSP页面中写forms更简单。
* lifecycle被限定在当前HTTP请求或HTTP Session的beans。这不是Spring MVC独有的技术，而是Spring MVC使用的WebApplicationContext的特性。

**1.2、其他MVC实现的可插拔性**

对于某些项目来说，非Spring的MVC实现更适合。很多团队希望借用已有的投资（囧，真抽象），例如，使用JSF。

如果你不想使用Spring Web MVC，而想使用Spring提供的其他解决方案，你可以将 *你选择的web MVC框架*集成到Spring中，这很简单。通过Spring的ContextLoaderListener启动Spring的root application context，可以在任意action object中通过它的ServletContext属性来获取它 -- 也可以使用Spring的相应帮助方法来获取。

你注册的beans和Spring的服务随时可用 -- 哪怕没有Spring MVC。这种情景下，Spring不会同其他web框架竞争。它只是简单的致力于纯web MVC框架没有关注的地方，从bean配置到数据访问和事务处理。所以，哪怕你只是想配合JDBC或Hibernate来使用Spring的事务抽象，仍然可以将Spring中间件 **和/或** 数据访问中间件作为你应用的一部分。

**2、DispatcherServlet**

像很多其他web MVC框架一样，Spring MVC框架也是请求驱动的，围绕**一个中心Servlet**来设计，该中心Servlet可以分发请求到Controllers，并提供其他功能。然而Spring的DispatcherServlet做的更多。它被彻底地与Spring IoC容器集成到了一起，所以，你可以使用Spring的所有特性。

Spring MVC DispatcherServlet处理请求的工作流程如下图所示。聪明的读者会认出**DispatcherServlet是Front Controller设计模式的一种实现**。

DispatcherServlet是一个具体的Servlet（继承自HttpServlet基类），所以你需要使用一个URL mapping来映射请求 -- 就是你想让DispatcherServlet处理的请求。这是一个标准Java EE Servlet配置，在Servlet 3.0+ 环境下可以这样注册该Servlet：

[复制代码](javascript:void(0);)

public class MyWebApplicationInitializer implements WebApplicationInitializer { // 这个接口，或者其抽象实现类

@Override

public void onStartup(ServletContext container) {

ServletRegistration.Dynamic registration = container.addServlet("example", new DispatcherServlet());

registration.setLoadOnStartup(1);

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

}

}

[复制代码](javascript:void(0);)

上面的例子中，所有以 /example开头的请求 都会由名字为example的DispatcherServlet实例来处理。

WebApplicationInitializer是Spring MVC提供的接口，可以确保基于代码的配置被探测到、且被自动用于初始化Servlet 3 容器。该接口的一个abstract基类是AbstractAnnotationConfigDispatcherServletInitializer，该abstract基类注册DispatcherServlet更简便，只需要指定映射、列出配置类即可 -- 这是**设置Spring MVC项目的一种推荐的方式（java config）**。

或者，传统模式，web.xml中的设置方式：

[复制代码](javascript:void(0);)

<web-app>

<servlet>

<servlet-name>example</servlet-name>

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

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

</servlet>

<servlet-mapping>

<servlet-name>example</servlet-name>

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

</servlet-mapping>

</web-app>

[复制代码](javascript:void(0);)

在前面曾讲过，在Spring中，ApplicationContext实例可以被scoped （就是有scope）。

而在Spring MVC框架中，每个DispatcherServlet都有其自有的WebApplicationContext，它会继承在root WebApplicationContext中定义的beans。 root WebApplicationContext应该包含所有基础beans，以被其他contexts 和 Servlet实例分享。被继承的beans可以被特定Servlet scope重写，你可以定义针对给定Servlet实例（其scope）的beans。

在DispatcherServlet初始化过程中，Spring MVC会在web应用的/WEB-INF文件夹下查找名字为 *[servlet-name]-servlet.xml*的文件，创建其中定义的bean，并会重写在全局scope中已经定义的任何beans的定义。

看一下下面的DispatcherServlet配置：

[复制代码](javascript:void(0);)

<web-app>

<servlet>

<servlet-name>golfing</servlet-name>

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

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

</servlet>

<servlet-mapping>

<servlet-name>golfing</servlet-name>

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

</servlet-mapping>

</web-app>

[复制代码](javascript:void(0);)

根据上面的Servlet配置，还需要一个文件：/WEB-INF/golfing-servlet.xml。该文件会包含所有的Spring MVC特定的组件（beans）。当然，你可以更改该路径，通过特定的Servlet初始化参数（详见下面）。

对于**单DispatcherServlet情景**来说，也可以只有一个root context。

这可以通过设置一个空的contextConfigLocation servlet init 参数来配置，如下：

[复制代码](javascript:void(0);)

<web-app>

<context-param>

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

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

</context-param>

<servlet>

<servlet-name>dispatcher</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>dispatcher</servlet-name>

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

</servlet-mapping>

<listener>

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

</listener>

</web-app>

[复制代码](javascript:void(0);)

WebApplicationContext是简单的ApplicationContext的一个扩展，针对web应用拥有一些额外的特性。

它不同于normal ApplicationContext的地方是它能够resolve themes，它还知道关联哪个Servlet（通过到ServletContext的连接）。

WebApplicationContext被束缚在ServletContext中，通过使用**RequestContextUtils类的静态方法**，你可以随时查找WebApplicationContext。

注意，我们可以使用基于Java的配置达到同样的目的：

[复制代码](javascript:void(0);)

public class GolfingWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer { //

@Override

protected Class<?>[] getRootConfigClasses() {

// GolfingAppConfig defines beans that would be in root-context.xml

return new Class[] { GolfingAppConfig.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

// GolfingWebConfig defines beans that would be in golfing-servlet.xml

return new Class[] { GolfingWebConfig.class };

}

@Override

protected String[] getServletMappings() {

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

}

}

[复制代码](javascript:void(0);)

**2.1、在WebApplicationContext中的特殊的bean types**

Spring DispatcherServlet使用特殊的beans来处理请求并渲染视图。这些beans是Spring MVC的一部分。你可以选择使用哪个 -- 只需要在WebApplicationContext中简单的配置一些即可。

Spring MVC维护了一个默认beans列表以供使用，下一部分会讲。

现在先来看看**DispatcherServlet依赖的特殊的bean types**：

|  |  |
| --- | --- |
| **bean type** | **解释** |
| HandlerMapping | 基于criteria（不同的HandlerMapping实现有不同的criteria）将incoming requests映射到handlers以及一个pre和post-processors（handler interceptors）。最流行的实现支持注解controllers。 |
| HandlerAdapter | 帮助DispatcherServlet调用映射到一个request的handler，无论handler是否实际被调用了。例如，调用一个注解Controller需要处理不同的注解。所以，HandlerAdapter的主要目的是让DispatcherServlet远离这些细节。 |
| HandlerExceptionResolver | 将exceptions映射到views，也允许更复杂的exception处理代码。 |
| ViewResolver | 处理基于字符串的逻辑视图的名字，将其转成实际的View 类型。 |
| LocaleResolver & LocaleContextResolver | Resolves the locale a client is using and possibly their time zone, in order to be able to offer internationalized views |
| ThemeResolver | Resolves themes your web application can use, for example, to offer personalized layouts |
| MultipartResolver | 解析multi-part请求，例如，支持处理来自HTML forms的文件上传。 |
| FlashMapManager | 存储和获取input 和 output的FlashMap -- 可用于从一个request将attributes传递至另一个，通常跨域一个redirect。 |

**2.2、默认的DispatcherServlet配置**

上面有提到，对每个特殊的bean，DispatcherServlet默认都维护了一个实现列表以供使用。这个信息保存在 **DispatcherServlet.properties** 中，在org.springframework.web.servlet包下。

所有的特殊的beans都有其合理的默认（bean还是设置？）。或早或晚，你总会需要定制这些beans提供的一个或多个properties。例如，配置InternalResourceViewResolver的prefix property 。

这里最需要理解的概念就是：一旦你在你的WebApplicationContext中配置了一个特殊的bean，如InternalResourceViewResolver，你就完全重写了某个类型的默认的实现列表。例如，如果你配置了InternalResourceViewResolver，那默认的ViewResolver实现列表会被忽略！

**2.3、DispatcherServlet处理顺序**

当你set up了一个DispatcherServlet，然后一个由其负责的request进来了，这时该DispatcherServlet会以如下流程开始处理请求：

1. 搜索WebApplicationContext，并将其绑定到该request，作为其attribute，controller和其他元素在该process中可能用到。默认键是 **DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE** 。
2. 将locale resolver绑定到request，以让process中的元素在processing the request时（如rendering the view、preparing data 等等）能够resolve 需要使用的locale。如果你不需要locale resolving，你不需要这个。
3. 将theme resolver绑定到request，让元素（如views）决定使用哪个theme。如果你不使用themes，可以忽略它。
4. 如果你指定了一个multipart file resolver，该request会被检查有无multiparts；如果有，该请求会被封装成MultipartHttpServletRequest以更进一步的处理。
5. 查找恰当合适的handler。如果找到了，关联到该handler的执行链（preprocessors、postprocessors、以及controllers）会被执行，以准备一个model或者rendering。
6. 如果返回了model，view会被rendered。如果没有返回model，（可能由于拦截等原因），没有view会被渲染，因为request可能已经被完成了！

**在WebApplicationContext中声明的handler exception resolvers会pick up 处理request过程中抛出的exceptions。使用这些exception resolvers允许你定义自己的行为来处理exceptions。**

SpringDispatcherServlet也支持返回 last-modification-date（最后修改日期），如同Servlet API中指定的一样。决定特定request的last modification date的处理是简单粗暴的：DispatcherServlet会查找合适的handler mapping，并测试找到的handler是否实现了LastModified接口。如果实现了，long getLastModified(request)方法的值会被返回。

你可以定制自己的DispatcherServlet实例，只要在web.xml的Servlet声明中添加Servlet初始化参数（init-param elements）即可。下面列出了支持的参数。

**DispatcherServlet 初始化参数**

|  |  |
| --- | --- |
| **参数** | **解释** |
| contextClass | 实现了WebApplicationContext的类，实例化了该Servlet使用的context。默认，使用XmlWebApplicationContext。 |
| contextConfigLocation | 传递给context实例（由contextClass指定）的字符串，用于指出在什么地方找到context config。如有多个，以逗号间隔。如果多个，且beans有重复定义，以最后一个为准。 |
| namespace | WebApplicationContext的命名空间。默认是[servlet-name]-servlet。 |

**3.、实现Controller**

Controller将用户的input解释并转成一个model，该model可以通过view描述给用户。

Spring 2.5 为MVC Controllers引入了基于注解的编程模型，使用诸如@RequestMapping、@RequestParam、@ModelAttribute之类的注解。在Servlet MVC和Portlet MVC中都可用。

在 [spring-projects Org on Github](https://github.com/spring-projects/) 里，大量的web应用都使用了这种支持。例如*MvcShowcase*, *MvcAjax*, *MvcBasic*, *PetClinic*, *PetCare*, 以及其他。

[复制代码](javascript:void(0);)

@Controller // 一个注解即可Controller

public class HelloWorldController {

@RequestMapping("/helloWorld") //

public String helloWorld(Model model) {

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

return "helloWorld";

}

}

[复制代码](javascript:void(0);)

**3.1、使用@Controller定义一个Controller**

@Controller注解用于标记一个class拥有controller的角色。

注意，这种**注解Controller需要开启自动探测才能使用**。如下：

[复制代码](javascript:void(0);)

<?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.springframework.samples.petclinic.web"/>**

<!-- ... -->

</beans>

[复制代码](javascript:void(0);)

**3.2、使用@RequestMapping映射requests**

使用@RequestMapping注解将URLs如/app映射到一个类或者特定的handler方法。示例：

[复制代码](javascript:void(0);)

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

private final AppointmentBook appointmentBook;

@Autowired

public AppointmentsController(AppointmentBook appointmentBook) {

this.appointmentBook = appointmentBook;

}

@RequestMapping(method = RequestMethod.GET)

public Map<String, Appointment> get() {

return appointmentBook.getAppointmentsForToday();

}

@RequestMapping(path = "/{day}", method = RequestMethod.GET)

public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {

return appointmentBook.getAppointmentsForDay(day);

}

@RequestMapping(path = "/new", method = RequestMethod.GET)

public AppointmentForm getNewForm() {

return new AppointmentForm();

}

@RequestMapping(method = RequestMethod.POST)

public String add(@Valid AppointmentForm appointment, BindingResult result) {

if (result.hasErrors()) {

return "appointments/new";

}

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}

[复制代码](javascript:void(0);)

class级别上的@RequestMapping不是强制的。

[复制代码](javascript:void(0);)

@Controller

public class ClinicController {

private final Clinic clinic;

@Autowired

public ClinicController(Clinic clinic) {

this.clinic = clinic;

}

@RequestMapping("/")

public void welcomeHandler() {

}

@RequestMapping("/vets")

public ModelMap vetsHandler() {

return new ModelMap(this.clinic.getVets());

}

}

[复制代码](javascript:void(0);)

**@RequestMapping 变体**

Spring 4.3 引入了以下@RequestMapping注解的变体，方法级别的。

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

@PatchMapping

[复制代码](javascript:void(0);)

@Controller

@RequestMapping("/appointments")

public class AppointmentsController {

private final AppointmentBook appointmentBook;

@Autowired

public AppointmentsController(AppointmentBook appointmentBook) {

this.appointmentBook = appointmentBook;

}

@GetMapping

public Map<String, Appointment> get() {

return appointmentBook.getAppointmentsForToday();

}

@GetMapping("/{day}")

public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {

return appointmentBook.getAppointmentsForDay(day);

}

@GetMapping("/new")

public AppointmentForm getNewForm() {

return new AppointmentForm();

}

@PostMapping

public String add(@Valid AppointmentForm appointment, BindingResult result) {

if (result.hasErrors()) {

return "appointments/new";

}

appointmentBook.addAppointment(appointment);

return "redirect:/appointments";

}

}

[复制代码](javascript:void(0);)

**@Controller 和 AOP 代理**

一些情况下，一个controller可能需要在运行时被AOP代理装饰。一个例子是在controller上使用@Transactional。这时，对这些controllers，我们**建议使用基于类的代理**。这也是controller的默认选项。

然而，如果一个controller必须实现一个非Spring Context回调的接口（如InitializingBean、\*Aware等等）的话，你可能需要显式的配置基于类的代理。例如，将 <tx:annotation-driven/> 改成 <tx:annotation-driven proxy-target-class="true"/> 。

**对Spring MVC 3.1中@RequestMapping methods的新的支持类（解析类？）**

Spring 3.1 引入了针对RequestMapping methods的一组新的支持类，叫做：**RequestMappingHandlerMapping** 、**RequestMappingHandlerAdapter**。推荐使用它们，甚至需要利用Spring MVC 3.1及以后的一些新特性。**这些新的支持类默认就由MVC命名空间和MVC Java config启用**，其他情况必须显式的配置。本部分会描述一下新旧支持类之间的一些重要区别。

Spring 3.1之前，type和method -level request mappings是在两个独立的阶段中检查的 -- controller先被DefaultAnnotationHandlerMapping选中，具体的方法调用则由AnnotationMethodHandlerAdapter负责。

Spring 3.1 中新的支持类，只需要使用RequestMappingHandlerMapping。不妨将controller的那些方法看作一个集合，其内容是带有映射的各种端点。

这使得一些新的可能成为现实。For once a HandlerInterceptor or a HandlerExceptionResolver can now expect the Object-based handler to be a HandlerMethod, which allows them to examine the exact method, its parameters and associated annotations. The processing for a URL no longer needs to be split across different controllers.

下面这几件事情已经**不再有效**：

* Select a controller first with a SimpleUrlHandlerMapping or BeanNameUrlHandlerMapping and then narrow the method based on @RequestMapping annotations. -- 先选择一个controller，再根据其@RequestMapping注解窄化请求？为什么不行了？？？几个意思？
* Rely on method names as a fall-back mechanism to disambiguate between two @RequestMapping methods that don’t have an explicit path mapping URL path but otherwise match equally, e.g. by HTTP method. In the new support classes @RequestMapping methods have to be mapped uniquely. --不再支持基于方法名来区分没有显式指定映射URL的@RequestMapping方法。必须显式的指定映射URL。
* Have a single default method (without an explicit path mapping) with which requests are processed if no other controller method matches more concretely. In the new support classes if a matching method is not found a 404 error is raised.  -- 不再支持默认方法处理请求！

上面的特性仍由现有的支持类支持。然而，如果想使用Spring 3.1的新特性，你需要使用新的支持类！

**URI Template Patterns**

URI Template是一个类似URI的字符串，包含一个或多个变量名字。当你使用值将其中的变量全部替换之后，该模板会变成一个URI。

在Spring MVC中，你可以在方法的一个参数上使用@PathVariable注解，就可以将实参绑定到URI模板变量的值。

@GetMapping("/owners/{ownerId}")

public String findOwner(@PathVariable String ownerId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

model.addAttribute("owner", owner);

return "displayOwner";

}

注意，如果方法的参数名与URI模板变量的名字不符，那需要显式的指定；否则可以省略。如下：

@GetMapping("/owners/{ownerId}")

public String findOwner(@PathVariable("ownerId") String theOwner, Model model) {

// implementation omitted

}

一个方法可以拥有任意数量的@PathVariable注解：

[复制代码](javascript:void(0);)

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

public String findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

Pet pet = owner.getPet(petId);

model.addAttribute("pet", pet);

return "displayPet";

}

[复制代码](javascript:void(0);)

当在Map<String, String>实参上使用@PathVariable时，map会被所有的URI模板变量填满。

URI模板可以从type和method级别的@RequestMapping注解中组装。

[复制代码](javascript:void(0);)

@Controller

@RequestMapping("/owners/{ownerId}") // 这里，类中的方法可以使用

public class RelativePathUriTemplateController {

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

public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {

// implementation omitted

}

}

[复制代码](javascript:void(0);)

@PathVariable实参可以是任何简单类型，如int、long、Date等。Spring会自动转换到合适的类型，如果失败会抛出TypeMishmatchException。-- 也可以注册其他数据类型： You can also register support for parsing additional data types. See [the section called “Method Parameters And Type Conversion”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-typeconversion) and [the section called “Customizing WebDataBinder initialization”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-webdatabinder).

**带正则表达式的URI Template Patterns**

有时候你需要更精确的定义URI模板变量。考虑下 URL "/spring-web/spring-web-3.0.5.jar"，你怎么将其拆分成多个部分？

@RequestMapping支持在URI模板变量中使用正则表达式。语法是： {varName:regex} 。如下：

@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]+}")

public void handle(@PathVariable String version, @PathVariable String extension) {

// ...

}

**Path Patterns**

除了URI模板，@RequestMapping注解和其所有变体还支持ant-style的path patterns，如 /mypath/\*.do。

**Path Pattern Comparison**

当一个URL匹配多个patterns时，会使用一种排序来查找最佳匹配。

带有更少URI变量和通配符的pattern ，被认为更匹配。例如，/hotels/{hotel}/\* 比 /hotels/{hotel}/\*\* 更合适，因为其他一样，通配符更少。

如果变量数量一样，更长的被认为更匹配。例如，/foo/bar\* 比 /foo/\* 更匹配。

当两个patterns拥有相同数量的变量和长度时，通配符少的更匹配。例如，/hotels/{hotel} 比 /hotels/\* 更匹配。

另外，还有两个特殊规则：

* /\*\*  匹配度最差。
* 带前缀的pattern，比其他所有不含双通配符的pattern，更差。例如：/public/\*\* 比 /public/path/{a} 更差。

**带有占位符的path patterns**

@RequestMapping注解的patterns还支持 ${...} 占位符。

**后缀pattern匹配**

默认，Spring MVC会执行 “.\*”的匹配，所以，当一个controller的被映射到/person的时候，实际上是隐式的被映射到/person.\*。这样可以使用URL轻松的请求不同的资源表现，如/person.pdf, /person.xml。

后缀pattern匹配可以被关闭，或者被限制在一组为了内容协商目的而注册的路径扩展名中。非常建议使用最普通的请求映射来最小化请求的模糊性，因为有时候“.”不一定代表扩展名，例如/person/{id}，有可能是/person/joe@xx.com。

**后缀pattern匹配和RFD**

Reflected file download （RFD） 攻击，最早由Trustwave在2014年指出。这种攻击类似XSS，都依赖可以反射到响应的输入。然而，不是将js插入到HTML中，RFD攻击依赖于浏览器的下载，并依赖于浏览器将响应视为一个可执行脚本（双击能运行的那种）。

在Spring MVC中，@ResponseBody 和 @ResponseEntity方法同样具备风险，因为它们可以渲染不同内容类型--客户端可能通过URL路径扩展名来请求的类型。需要注意，无论是单独的禁用后缀pattern匹配还是单独的禁用用于内容协商目的的路径扩展名，都不能有效的组织RFD攻击。

为了有效的防护RFD，Spring在渲染响应体之前添加了一个header（Content-Disposition:inline;filename=f.txt）以建议一个固定和安全的下载文件名。这只有在URL路径包含的扩展名既不在白名单中，也不是因内容协商目的而显式注册的文件扩展名时才有效。然而，这样做也有其副作用，有时候可能会直接显示在浏览器中！

默认，很多常用的路径扩展名已经在白名单中。此外，REST API的调用通常不是用作在浏览器中使用的URL。尽管如此，使用自定义HttpMessageConverter实现的应用，可以显式的注册用于内容协商目的的文件扩展名，针对这些扩展名Content-Disposition header（Content-Disposition:inline;filename=f.txt）不会被添加。

This was originally introduced as part of work for [CVE-2015-5211](http://pivotal.io/security/cve-2015-5211). Below are additional recommendations from the report:

* Encode rather than escape JSON responses. This is also an OWASP XSS recommendation. For an example of how to do that with Spring see [spring-jackson-owasp](https://github.com/rwinch/spring-jackson-owasp).
* Configure suffix pattern matching to be turned off or restricted to explicitly registered suffixes only.
* Configure content negotiation with the properties "useJaf" and "ignoreUnknownPathExtensions" set to false which would result in a 406 response for URLs with unknown extensions. Note however that this may not be an option if URLs are naturally expected to have a dot towards the end.
* **Add X-Content-Type-Options: nosniff header to responses. Spring Security 4 does this by default.**

**Matrix Variables**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-matrix-variables>

URI specification [RFC 3986](http://tools.ietf.org/html/rfc3986#section-3.3)定义了在path segments内包含name-value对的可行性。在Spring MVC中，它们被视为matrix Variables。

matrix variables可能出现在任意path segment中，每个matrix variable都由分号隔离。例如："/cars;color=red;year=2012"。多个value的时候，可以使用逗号拼接，如"color=red,green,blue"，也可以重复name，如"color=red;color=green;color=blue"。

如果希望一个URL包含matrix variables，请求映射pattern必须使用URI模板来代表它们。

下面是提取matrix variable ”q”的例子：

[复制代码](javascript:void(0);)

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

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

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

// petId == 42

// q == 11

}

[复制代码](javascript:void(0);)

因为所有的path segments都可能含有matrix variables，某些情况下你需要更精确的信息来确定需要的变量：

[复制代码](javascript:void(0);)

// GET /owners/42;q=11/pets/21;q=22

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

public void findPet(

@MatrixVariable(name="q", pathVar="ownerId") int q1,

@MatrixVariable(name="q", pathVar="petId") int q2) {

// q1 == 11

// q2 == 22

}

[复制代码](javascript:void(0);)

一个matrix variable可以被定义为可选项，可以拥有一个指定的默认值；

[复制代码](javascript:void(0);)

// GET /pets/42

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

public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

// q == 1

}

[复制代码](javascript:void(0);)

所有的matrix variables可以用一个Map来获取：

[复制代码](javascript:void(0);)

// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

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

public void findPet(

@MatrixVariable MultiValueMap<String, String> matrixVars,

@MatrixVariable(pathVar="petId"") MultiValueMap<String, String> petMatrixVars) {

// matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]

// petMatrixVars: ["q" : 11, "s" : 23]

}

[复制代码](javascript:void(0);)

注意：**为了启用matrix variables**，你必须设置RequestMappingHandlerMapping的removeSemicolonContent property为false。其默认是true。

The MVC Java config and the MVC namespace both provide options for enabling the use of matrix variables.

If you are using Java config, The [Advanced Customizations with MVC Java Config](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-config-advanced-java) section describes how the RequestMappingHandlerMapping can be customized.

In the MVC namespace, the <mvc:annotation-driven> element has an enable-matrix-variables attribute that should be set to true. By default it is set to false.

<?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 enable-matrix-variables="true"/>

</beans>

**Consumable Media Types**

通过指定一个consumable media types列表来窄化映射。只有request header中的Content-Type符合指定的媒体类型时，请求才匹配。例如：

@PostMapping(path = "/pets", consumes = "application/json")

public void addPet(@RequestBody Pet pet, Model model) {

// implementation omitted

}

注意，consumable media type表达式可以使用“!”来否定匹配的媒体类型，如使用“!text/plain”来匹配除了text/plain之外的Content-Type。建议使用MediaType中的常量，如 APPLICATION\_JSON\_VALUE、 APPLICATION\_JSON\_UTF8\_VALUE 。

注意，虽然consumes条件支持type和method级别，但是，不同于其他条件，**method级别的会覆盖type级别的类型！！！**

**Producible Media Types**

还可以通过指定一个producible media types列表来窄化请求。仅当request header的Accept匹配时，该request才会匹配。此外，使用produces条件会确保实际的内容类型。如下：

@GetMapping(path = "/pets/{petId}", produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE)

@ResponseBody

public Pet getPet(@PathVariable String petId, Model model) {

// implementation omitted

}

注意produces条件指定的媒体类型，也可以选择性的指定一个字符集。例如，在上面的代码片段中，我们指定了与MappingJackson2HttpMessageConverter中默认配置的媒体类型一致的媒体类型。--**是否可以认为，一种字符集就是一种媒体类型？**

**同consumes类似，produces也可以使用“!”。同样建议使用MediaType中的常量。**

**同consumes类似，方法级别的produces会覆盖类级别的媒体类型！！！**

**请求参数和请求头的值 Request Parameter 、Request Header values**

可以通过请求参数条件来窄化请求的匹配，如："myParam", "!myParam", or "myParam=myValue"。前两个用于测试请求参数中是否出现该参数，第三个则需要请求参数有一个特定的值。例子：

[复制代码](javascript:void(0);)

@Controller

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

public class RelativePathUriTemplateController {

@GetMapping(path = "/pets/{petId}", params = "myParam=myValue")

public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {

// implementation omitted

}

}

[复制代码](javascript:void(0);)

同样的情况还适合请求头：

[复制代码](javascript:void(0);)

@Controller

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

public class RelativePathUriTemplateController {

@GetMapping(path = "/pets", headers = "myHeader=myValue")

public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model model) {

// implementation omitted

}

}

[复制代码](javascript:void(0);)

虽然你可以使用通配符来匹配Content-Type和Accept header values（如headers=*"content-type=text/\*"*，可以匹配*"text/plain"* 和*"text/html"*），但建议使用consumes和produces。这也是它们的设计目的。

**HTTP HEAD 和 HTTP OPTIONS**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-requestmapping-head-options>

@RequestMapping方法映射到“GET”，同时也会隐式的映射到“HEAD”！

@RequestMapping方法内建支持HTTP OPTIONS。略。

**3.3、定义@RequestMapping handler methods**

@RequestMapping handler methods可以有非常灵活的签名。除了BindingResult参数之外的参数可以按任意顺序排放。

Spring 3.1 为 @RequestMapping methods引入了一组新的支持类，分别是：RequestMappingHandlerMapping and RequestMappingHandlerAdapter。建议使用它们，而且，应该使用Spring 3.1及以后的新特性。这些新的支持类默认由MVC命名空间启用，如果是Java config，必须显式的配置--否则无法使用。

**支持的方法参数类型**

* 请求/响应对象（Servlet API）。如ServletRequest 或 HttpServletRequest。
* 会话对象（Servlet API），HttpSession类型的。该类型的参数会强制相应的session出现。就是说，其实参永远非null。

session的访问可能不是线程安全的，特别是在一个Servlet环境中。如果需要多个请求并发访问一个session时，可以考虑将RequestMappingHandlerAdapter的synchronizeOnSession设置为true。

* org.springframework.web.context.request.WebRequest 或者 org.springframework.web.context.request.NativeWebRequest。允许泛型的请求参数访问，以及请求/会话属性访问，不需要绑定到native的Servlet/Portlet API。
* java.util.Locale，用于获取当前请求的locale，由已启用的最符合的locale resolver来决定--实际上是在MVC环境中配置的LocaleResolver/LocacleContextResolver。
* java.util.TimeZone (Java 6+) / java.time.ZoneId (Java 8)，用于获取当前请求的时区，由LocaleContextResolver决定。
* java.io.InputStream / java.io.Reader，用于获取请求的内容。其值是由Servlet API暴露的原生InputStream/Reader。
* java.io.OutputStream / java.io.Writer，用于生成响应的内容。其值是由Servlet API暴露的原生的OutputStream/Writer。
* org.springframework.http.HttpMethod，用于获取HTTP request method。
* java.security.Principle，包含了当前已认证的用户。
* @PathVariable注解的参数，用于获取URI模板变量。
* @MatrixVariable注解的参数，用于获取URI path segments中的name-value对。
* @RequestParam注解的参数，用于获取特定的Servlet 请求参数。参数值会被转换成方法参数类型 -- 类型转换。
* @RequestHeader注解的参数，用于获取特定的Servlet请求HTTP headers。类型转换。
* @RequestBody注解的参数，用于获取HTTP请求体。参数值会使用HttpMessageConverters来进行类型转换。
* @RequestPart注解的参数，用于获取一个multipart/form-data请求的内容。
* @SessionAttribute注解的参数，用于获取已有的、永久的session attributes （例如用户认证对象）。与其对比，通过@SessionAttributes会获取临时存储在session的model attributes。
* @RequestAttribute注解的参数，用于获取request attributes。
* HttpEntity<?> 参数，用于获取Servlet request HTTP headers和contents。request stream会被转成entity body--使用HttpMessageConverters。
* java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap，略。
* org.springframework.web.servlet.mvc.support.RedirectAttributes，用于指定在重定向中使用的一组具体的attributes，也可以用于添加flash attributes（attributes暂存在服务器侧，以在重定向后可用。）。
* command或者form objects，用于绑定请求参数到bean properties，或者直接绑定到字段，通过定制的类型转换--依赖于@InitBinder方法 和/或 HandlerAdapter配置。见RequestMappingHandlerAdapter的webBindingInitializer property。这些command objects和它们的校验结果默认会被暴露成model attributes，使用command class name -- 例如，用model attribute “orderAddress”代表一个“some.package.OrderAddress”类型的command object。@ModelAttribute注解，可以用在方法参数上，以定制model attribute name。
* org.springframework.validation.Errors / org.springframework.validation.BindingResult校验结果，用于其前面的第一个command 或者 form object。
* org.springframework.web.bind.support.SessionStatus，状态处理，用于标记form处理已完成，该标记会导致session attributes被清理 -- 那些由@SessionAttributes注解在type上指定的session attributes！
* org.springframework.web.util.UriComponentsBuilder，一个builder，用于预处理一个URL，关联到当前请求的host、port、scheme、context path、以及servlet mapping的字面值部分。

Errors和BindingResult参数，必须跟在model object后面，因为可能有多个model object，Spring会为每个model object创建一个单独的BindingResult，所以，**下面的例子不会执行**：

@PostMapping // Invalid ordering of BindingResult and @ModelAttribute.

public String processSubmit(@ModelAttribute("pet") Pet pet, Model model, BindingResult result) { ... }

注意，在Pet和BindingResult之间有一个Model参数。必须如下排序才能工作：

@PostMapping

public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result, Model model) { ... }

支持JDK 1.8 的 java.util.Optional 作为方法参数类型，与带有required attribute的注解一起使用（如@RequestParam、@RequestHeader等等）。这些情况下，java.util.Optional 相当于 required = false 。

**支持的方法返回类型**

* ModelAndView对象。with the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* Model对象。with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* Map对象。 for exposing a model, with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* View对象。 with the model implicitly determined through command objects and @ModelAttribute annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).
* 一个String值，能被解释为logical view name。with the model implicitly determined through command objects and @ModelAttribute annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).
* void，如果方法自己处理响应 -- 方法形参里声明ServletResponse / HttpServletResponse。or if the view name is supposed to be implicitly determined through a RequestToViewNameTranslator (not declaring a response argument in the handler method signature).
* 如果方法带有@ResponseBody，返回类型会被写成response HTTP body。返回值会被转成声明的方法参数类型--使用HttpMessageConverters。
* HttpEntity<?>或ResponseEntity<?>对象，用于访问Servlet response HTTP headers 和 contents。该entity body会被转成response stream -- 通过HttpMessageConverters。
* HttpHeaders对象，用于返回一个无body的response。
* 当应用想在一个由Spring MVC管理的线程中异步地produce返回值时，可以返回Callable<?>。
* 当应用想从它自己选择的线程中produce返回值时，可以返回DeferredResult<?>。
* 当应用想从它自己选择的线程中produce返回值时，可以返回ListenableFuture<?>。
* 返回一个ResponseBodyEmittter，以异步的将多个对象写入到response。也可以作为ResponseEntity内的body。
* 返回一个SseEmitter，以异步的将Server-Sent Events写入到response。也可以作为ResponseEntity内的body。
* 返回一个StreamingResponseBody，以异步的写入到response OutputStream。也可以作为ResponseEntity内的body。
* 任意其他返回类型，都被认为是一个单独的model attribute，暴露给view，--使用通过方法级别的@ModelAttribute指定的attribute name（或者，默认的attribute name，基于返回类型的类名字）。The model is implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.

**使用@RequestParameter将请求参数绑定到方法参数**

[复制代码](javascript:void(0);)

@Controller

@RequestMapping("/pets")

@SessionAttributes("pet")

public class EditPetForm {

// ...

@GetMapping

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

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

model.addAttribute("pet", pet);

return "petForm";

}

// ...

}

[复制代码](javascript:void(0);)

默认，@RequestParam的required attribute是true，可以设置为false。@RequestParam(name="id", required=false)

如果目标方法参数的类型不是String，会自动使用类型转换。

当在Map<String, String> 或 MultiValueMap<String, String>实参上使用@RequestParam注解时，会被填入所有的request parameters。

**使用RequestBody注解来映射request body**

方法参数的@RequestBody注解，标识了方法参数应该绑成HTTP request body的值。例如：

@PutMapping("/something")

public void handle(@RequestBody String body, Writer writer) throws IOException {

writer.write(body);

}

可以通过使用一个HttpMessageConverter将request body转成method argument。HttpMessageConverter负责将HTTP request msg转成一个对象以及将对象转成HTTP response body。RequestMappingHandlerAdapter支持@RequestBody的默认HttpMessageConverters：

* ByteArrayHttpMessageConverter converts byte arrays.
* StringHttpMessageConverter converts strings.
* FormHttpMessageConverter converts form data to/from a MultiValueMap<String, String>.
* SourceHttpMessageConverter converts to/from a javax.xml.transform.Source.

**注意，如果使用MVC 命名空间或者使用MVC Java config，默认会注册更多的message converters。**

如果你打算读写XML，你会需要配置一个MarshallingHttpMessageConverter -- 使用org.springframework.oxm包中的特定的Marshaller和Unmarshaller实现。下面的例子演示了如何直接读写XML -- 但是，如果你的应用是通过MVC命名空间或MVC Java config来配置的，见 [Section 22.16.1, “Enabling the MVC Java Config or the MVC XML Namespace”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-config-enable) 。

[复制代码](javascript:void(0);)

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

<property name="messageConverters">

<util:list id="beanList">

<ref bean="stringHttpMessageConverter"/>

<ref bean="marshallingHttpMessageConverter"/>

</util:list>

</property

</bean>

<bean id="stringHttpMessageConverter"

class="org.springframework.http.converter.StringHttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter"

class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">

<property name="marshaller" ref="castorMarshaller"/>

<property name="unmarshaller" ref="castorMarshaller"/>

</bean>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>

[复制代码](javascript:void(0);)

@RequestBody 注解的方法参数还可以使用@Valid注解，Spring会使用配置好的Validator实例来校验该参数。当使用MVC命名空间或MVC Java config时，一个JSR-303 validator会被自定的配置 -- 假如classpath中有一个JSR-303实现。

就像使用@ModelAttribute注解的参数已有，Errors参数可以用来检查errors。如果没有声明这样一个参数，会抛出一个MethodArgumentNotValidException。该异常由DefaultHandlerExceptionResolver来处理，会返回400 error。

**使用@ResponseBody注解来映射response body**

@ResponseBody注解类似于@RequestBody。该注解放在方法上指示返回类型会被写入HTTP response body （没有被放入Model，或被解释成view name）。 例如：

@GetMapping("/something")

@ResponseBody

public String helloWorld() {

return "Hello World";

}

上面的例子，会将字符串写入HTTP response stream。

如同@RequestBody，Spring会将返回的对象转换成一个response body -- 使用一个HttpMessageConverter。

**使用@RestController注解创建一个REST Controller**

使用@RestController代替@ResponseBody与@Controller。它虽然是由后两者组合而成，但在将来会被赋予更多语义。

@RestController也可以配合@ControllerAdvice或@RestControllerAdvice beans。详见 the [the section called “Advising controllers with @ControllerAdvice and @RestControllerAdvice”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-controller-advice) section。

**使用HttpEntity**

HttpEntity 类似于 @RequestBody和@ResponseBody。除了能获取request和response body之外，HttpEntity（以及其response子类：ResponseEntity）还允许获取request和response  headers，如下：

[复制代码](javascript:void(0);)

@RequestMapping("/something")

public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity) throws UnsupportedEncodingException {

String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader");

byte[] requestBody = requestEntity.getBody();

// do something with request header and body

HttpHeaders responseHeaders = new HttpHeaders();

responseHeaders.set("MyResponseHeader", "MyValue");

return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);

}

[复制代码](javascript:void(0);)

The above example gets the value of the MyRequestHeader request header, and reads the body as a byte array. It adds the MyResponseHeader to the response, writes Hello World to the response stream, and sets the response status code to 201 (Created).

As with @RequestBody and @ResponseBody, Spring uses HttpMessageConverter to convert from and to the request and response streams. For more information on these converters, see the previous section and [Message Converters](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/remoting.html#rest-message-conversion).

**在方法上使用@ModelAttribute**

该注解可以用在方法或方法参数上。本部分讲解用在方法上的作用，下一部分会讲解用在方法参数上的作用。

在方法上使用该注解，意味着该方法的一个目的是增加一个或多个model attribute。该方法支持的参数类型与@RequestMapping methods一样，但不能直接映射到请求。相反，同一个Controller中的@ModelAttribute methods会在@RequestMapping methods之前被调用！！！例子：

[复制代码](javascript:void(0);)

// 添加一个 attribute

// 该方法的返回值会被添加到model account中

// 你可以定义该model的名字，例如 @ModelAttribute("myAccount")

@ModelAttribute

public Account addAccount(@RequestParam String number) {

return accountManager.findAccount(number);

}

// 添加多个 attributes

@ModelAttribute

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

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

// add more ...

}

[复制代码](javascript:void(0);)

@ModelAttribute methods被用于将常用的attributes填入model。

**注意两种形式的@ModelAttribute methods。第一个，是隐式的将返回值添加为attribute。第二个，接收一个Model，然后在其中增加任意数量的model attributes。**

一个Controller可以拥有任意数量的@ModelAttribute methods。所有这些方法都会在同一个Controller中的@RequestMapping methods之前被调用！

@ModelAttribute methods 也可以被定义在@ControllerAdvice class内，这样的methods会被用于所有Controllers。--就是在所有Controller的所有@RequestMapping methods之前被调用！

如果没有显式指定一个model attribute name，会发生什么？这种情况下，会基于其类型赋予一个默认的名字。例如，如果方法返回了Account类型，那默认的name就是account。

@ModelAttribute注解也可以用在@RequestMapping methods上。这种情况下，方法的返回值被解释成一个model attribute，而非view name。view name会基于name惯例而获取到，更像是返回了void。 see [Section 22.13.3, “The View - RequestToViewNameTranslator”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-coc-r2vnt)。

**在方法参数上使用@ModelAttribute**

当@ModelAttribute用于方法参数上时，代表该参数应该从该model中获取。如果model中没有，该参数会先被实例化，再被添加到model。一旦出现在model中，该参数的字段会被匹配名字的request parameters填充。这就是Spring MVC中的数据绑定（data binding），一个非常有用的机制，节省了你手动解析每个form字段的时间。

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

public String processSubmit(@ModelAttribute Pet pet) { }

上面的例子，Pet实例从哪里来？有几个选项：

* 可能已经存在于@SessionAttributes的model中。
* 可能已经存在于同一个Controller的@ModelAttribute method的model中。
* 可能基于URI模板变量和类型转换器而获取（稍后详解）。
* 可能使用其默认构造器实例化。

@ModelAttribute method是从数据库中获取attribute的一种常用方式，可能可选的存储于requests之间--通过使用@SessionAttributes。某些情况下，使用URI模板变量和类型转换器更为方便。例子：

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

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

// ...

}

上面的例子，model attribute的name与URI模板变量的名字一致。如果你注册了一个Converter<String, Account>，那么上面的例子就可以不必使用一个@ModelAttribute method。

下一步就是数据绑定。WebDataBinder类会匹配request parameter names -- 包含query string parameters 和 form fields -- 到model attribute fields，根据名字。匹配的字段会被填充--当必要的类型转换被应用了之后。Data binding and validation are covered in [Chapter 9, *Validation, Data Binding, and Type Conversion*](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html). Customizing the data binding process for a controller level is covered in [the section called “Customizing WebDataBinder initialization”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-webdatabinder).

数据绑定的一个结果是，可能存在errors，例如缺失必须的字段或者类型转换错误。为了检查该类错误，需要在@ModelAttribute argument之后紧跟着添加一个BindingResult argument。

[复制代码](javascript:void(0);)

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

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

if (result.hasErrors()) {

return "petForm";

}

// ...

}

[复制代码](javascript:void(0);)

使用BindingResult，你可以检查是否有errors，可以使用Spring的<errors> form tag来在同一个form中显示错误。

注意，某些情况下，不使用数据绑定而获取model中的一个attribute很有用。这些情况下，你可以在Controller中注入Model，或者在注解上使用binding flag，如下：

[复制代码](javascript:void(0);)

@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) {

// ...

}

[复制代码](javascript:void(0);)

In addition to data binding you can also invoke validation using your own custom validator passing the same BindingResult that was used to record data binding errors. That allows for data binding and validation errors to be accumulated in one place and subsequently reported back to the user:

[复制代码](javascript:void(0);)

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

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

new PetValidator().validate(pet, result);

if (result.hasErrors()) {

return "petForm";

}

// ...

}

[复制代码](javascript:void(0);)

-- 就是根据BindingResult的结果进行自己的操作。

或者，可以使用JSR-303 @Valid注解来自动调用校验：

[复制代码](javascript:void(0);)

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

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

if (result.hasErrors()) {

return "petForm";

}

// ...

}

[复制代码](javascript:void(0);)

**使用@SessionAttributes在requests之间的HTTP session中存储model attributes**

type-level @SessionAttributes注解，声明了用于特定handler的session attributes。这会列出model attributes的names或types -- 应该透明的存储于session或某conversational storage，在后续的requests中作为form-backing beans。

[复制代码](javascript:void(0);)

@Controller

@RequestMapping("/editPet.do")

@SessionAttributes("pet")

public class EditPetForm {

// ...

}

[复制代码](javascript:void(0);)

**使用@SessionAttribute访问预存的全局session attributes**

如果需要访问pre-existing global session attributes，就是在controller外部（例如在filter中）管理的 ，且可能或可能不会出现在method parameter上使用@SessionAttribute注解（--什么鬼）（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）。

@RequestMapping("/")

public String handle(@SessionAttribute User user) {

// ...

}

当需要增加或移除session attributes时，可以考虑在controller method上注入 org.springframework.web.context.request.WebRequest 或 javax.servlet.http.HttpSession。

为了在session中临时存储model attributes以作为controller workflow的一部分，可以考虑使用SessionAttributes as described in [the section called “Using @SessionAttributes to store model attributes in the HTTP session between requests”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-sessionattrib).

**使用@RequestAttribute来获取request attributes**

类似于@SessionAttribute，@RequestAttribute也可以用于获取pre-existing request attributes -- 由filter或interceptor创建的。

@RequestMapping("/")

public String handle(@RequestAttribute Client client) {

// ...

}

**处理application/x-www-form-urlencoded data**

前面的部分描述了使用@ModelAttribute来支持来自浏览器客户端的form submission requests。@ModelAttribute注解还被推荐用于处理来自非浏览器客户端的请求。然而，当处理HTTP PUT requests时，有一个显著的不同。浏览器会通过HTTP GET或HTTP POST提交表单数据。非浏览器客户端还可以通过HTTP PUT来提交。这就有一个问题，因为Servlet specification要求 ServletRequest.getParameter\*()方法仅支持HTTP POST的字段获取，而非HTTP PUT。

为了支持HTTP PUT和PATCH 请求，spring-web模块提供了过滤器：**HttpPutFormContentFilter**。

[复制代码](javascript:void(0);)

<filter>

<filter-name>httpPutFormFilter</filter-name>

<filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>httpPutFormFilter</filter-name>

<servlet-name>dispatcherServlet</servlet-name>

</filter-mapping>

<servlet>

<servlet-name>dispatcherServlet</servlet-name>

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

</servlet>

[复制代码](javascript:void(0);)

上面的filter，会拦截content type为 application/x-www-form-urlencoded 的 HTTP PUT和PATCH请求，从其请求体中读取表单数据，封装ServletRequest以让ServletRequest.getParameter\*() 能够使用表单数据。

由于HttpPutFormContentFilter会consume请求体，所以，不应为那些依赖针对 application/x-www-form-urlencoded 的转换器的PUT或PATCH URLs配置该filter。这包括@RequestBody MultiValueMap<String, String> 和 HttpEntity<MultiValueMap<String, String>>。

**使用@CookieValue注解来映射cookie values**

该注解允许一个方法参数绑定一个HTTP cookie的值。

假定从http request接收了如下cookie：

JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84

下面的代码演示了如何获取JSESSIONID cookie：

@RequestMapping("/displayHeaderInfo.do")

public void displayHeaderInfo(@CookieValue("JSESSIONID") String cookie) {

//...

}

如果目标方法参数的类型不是String，会自动应用类型转换。

该注解，也被在Servlet和Portlet环境下的annotated handler methods支持。

**使用@RequestHeader注解来映射request header attributes**

这是一个样例request header：

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

下面的代码演示了如何获取Accept-Encoding和Keep-Alive headers的值：

@RequestMapping("/displayHeaderInfo.do")

public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,

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

//...

}

如果目标方法参数的类型不是String，会自动应用类型转换。

当@RequestHeader注解用于一个Map<String, String>、 MultiValueMap<String, String>，或HttpHeaders argument时，该map会被填入所有的header values。

内建的支持允许转换一个逗号间隔的字符串，将其转成一个字符串或其他类型的数组/集合。例如，@RequestHeader(“Accept”)注解的方法参数，可能是String类型，也可能是String[]或List<String>类型。

该注解，也被在Servlet和Portlet环境下的annotated handler methods支持。

**method parameters 和 type conversion**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-typeconversion>

从request中提取出来的基于字符串的值，包括request parameters、path variables、request headers、还有cookie values，可能需要被转成它们要绑定的method parameter或field的类型。如果目标类型不是String，Spring会自动转成合适的类型。支持所有简单类型，如int、long、Date等等。甚至，你可以使用一个WebDataBinder来定制转换过程，或者通过在FormattingConversionService中注册Formatters。

**定制WebDataBinder 初始化**

通过Spring的WebDataBinder使用PropertyEditors来定制request parameter 的绑定，你可以在你的controller中使用@InitBinder methods，或者在@ControllerAdvice class中使用@InitBinder methods，或者提供一个定制的WebBindingInitializer。 See the [the section called “Advising controllers with @ControllerAdvice and @RestControllerAdvice”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-controller-advice) section for more details.

**使用@InitBinder 定制数据绑定**

在controller方法上使用@InitBinder，允许你在controller内部配置web数据绑定。@InitBinder代表方法初始化了WebDataBinder--会被用于填充被注解的handler方法的command 和 form object arguments。

这些init-binder methods，支持@RequestMapping所支持的所有参数，除了command/form objects和相应的校验结果对象。init-binder methods必须没有返回值。所以，大多被声明为void。 典型的arguments包括WebDataBinder 结合 WebRequest或 java.util.Locale，允许代码注册特定context的editors。

下面的案例演示了使用@InitBinder来配置针对所有java.util.Date form properties的一个CustomDateEditor。

[复制代码](javascript:void(0);)

@Controller

public class MyFormController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

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

dateFormat.setLenient(false);

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

}

// ...

}

[复制代码](javascript:void(0);)

或者，自Spring 4.2起，可以考虑使用addCustomFormatter来指定Formatter实现，取代PropertyEditor实例。

如果你在一个shared FormattingConversionService中有基于Formatter的设置，这会非常有用，只需要同样的做法来复用特定controller针对binding rules的调节。

[复制代码](javascript:void(0);)

@Controller

public class MyFormController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

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

}

// ...

}

[复制代码](javascript:void(0);)

**配置一个定制的WebBindingInitializer**

为了将数据绑定初始化外部化，你可以提供一个定制的WebBindingInitializer实现，然后通过提供一个定制的AnnotationMethodHandlerAdapter的bean configuration来启用它。

下面的例子示意了使用了org.springframework.samples.petclinic.web.ClinicBindingInitializer的配置，该配置配置了几个PetClinic controllers需要的PropertyEditors。

[复制代码](javascript:void(0);)

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

<property name="cacheSeconds" value="0"/>

<property name="webBindingInitializer">

<bean class="org.springframework.samples.petclinic.web.ClinicBindingInitializer"/>

</property>

</bean>

[复制代码](javascript:void(0);)

@InitBinder methods也可以定义在@ControllerAdvice class内，会用于所有的controllers。效果同WebBindingInitializer。See the [the section called “Advising controllers with @ControllerAdvice and @RestControllerAdvice”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-controller-advice) section for more details.

**advising controllers with @ControllerAdvice and @RestControllerAdvice**

@ControllerAdvice注解，是一个component annotation，允许实现类能够在classpath扫描中被自动探测到。当使用MVC namespace或MVC Java config时，自动启用。

@ControllerAdvice class，可以含有@ExceptionHandler、@InitBinder以及@ModelAttribute methods，这些methods，会被应用到所有controller中的@RequestMapping methods，而非仅仅其所声明的controller中。

@RestControllerAdvice，等同于@ExceptionHandler + @ResponseBody methods。

@ControllerAdvice和@RestControllerAdvice 都可以指定作用的controllers：

[复制代码](javascript:void(0);)

// Target all Controllers annotated with @RestController

@ControllerAdvice(annotations = RestController.class)

public class AnnotationAdvice {}

// Target all Controllers within specific packages

@ControllerAdvice("org.example.controllers")

public class BasePackageAdvice {}

// Target all Controllers assignable to specific classes

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

public class AssignableTypesAdvice {}

[复制代码](javascript:void(0);)

详见@ControllerAdvice文档。

**Jackson Serialization View Support**

It can sometimes be useful to filter contextually the object that will be serialized to the HTTP response body. 为了提供这种能力，Spring MVC提供了内建的支持，可以rendering with [Jackson’s Serialization Views](http://wiki.fasterxml.com/JacksonJsonViews).

在一个@Response controller method上，或者在那些返回ResponseEntity的controller methods上，简单的添加@JsonView注解，并指定需要使用的view class或Interface即可。如下：

[复制代码](javascript:void(0);)

@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;

}

}

[复制代码](javascript:void(0);)

注意，尽管@JsonView允许指定多个class，但在controller method上使用时只能指定一个class！可以考虑使用复合接口，如果你需要启用多个views。

对于那些依赖view resolution的controllers，简单的将序列化view class添加到model即可：

[复制代码](javascript:void(0);)

@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";

}

}

[复制代码](javascript:void(0);)

**Jackson JSONP 支持**

为了启用JSONP对@ResponseBody和@ResponseEntity methods的支持，声明一个@ControllerAdvice bean  -- 需要继承AbstractJsonpResponseBodyAdvice，并在其构造中指出JSONP的query parameter name(s)。如下：

[复制代码](javascript:void(0);)

@ControllerAdvice

public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {

public JsonpAdvice() {

super("callback");

}

}

[复制代码](javascript:void(0);)

对于依赖view resolution的controllers，JSONP自动被启用，默认的query parameter name是 jsonp 或 callback。 可以通过其jsonpParameterNames property来定制。

**3.4、异步请求的处理**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-async>

Spring 3.2 引入了基于 Servlet 3 的异步请求处理。不是一直以来的让controller method返回一个值，而是，让controller method返回一个java.util.concurrent.Callable，然后从Spring MVC管理的线程中produce 返回值。同时，main Servlet container thread 会被退出和释放，以处理其他请求。Spring MVC在一个独立的线程调用Callable -- 通过TaskExecutor，当Callable返回时，请求会被分派回Servlet container，从而恢复处理。例子：

[复制代码](javascript:void(0);)

@PostMapping

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

return new Callable<String>() {

public String call() throws Exception {

// ...

return "someView";

}

};

}

[复制代码](javascript:void(0);)

另一个选择是，controller method返回DeferredResult。这种情况下，也可能是任意线程produce的返回值，就是说，非Spring MVC管理的线程！例如，结果可能是响应某个外部事件，如一个JMS message、一个scheduled task等等，而produce的结果。下面是一个例子：

[复制代码](javascript:void(0);)

@RequestMapping("/quotes")

@ResponseBody

public DeferredResult<String> quotes() {

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

// Save the deferredResult somewhere..

return deferredResult;

}

// In some other thread...

deferredResult.setResult(data);

[复制代码](javascript:void(0);)

如果没有Servlet 3.0 异步请求处理特性的相关知识，会很难理解这点。这里是关于底层机制的一些基本的事实：

* 一个ServletRequest可以被放入asynchronous mode，使用request.startAsync()即可。这样做的主要效果就是，该Servlet以及所有Filters，能够退出，但response仍然保持open-- 允许在后面完成处理。
* request.startAsync() 会返回 AsyncContext，可以被用于对async处理的更进一步的控制。例如，它提供了dispatch方法，类似于Servlet API的forward -- 除了它允许应用恢复在一个Servlet container thread中的请求处理。
* ServletRequest提供了对当前DispatcherType的访问，该DispatcherType can be used to distinguish between processing the initial request, an async dispatch, a forward, and other dispatcher types.

With the above in mind, the following is the sequence of events for async request processing with a Callable:

* Controller returns a Callable.
* Spring MVC starts asynchronous processing and submits the Callable to a TaskExecutor for processing in a separate thread.
* The DispatcherServlet and all Filter’s exit the Servlet container thread but the response remains open.
* The Callable produces a result and Spring MVC dispatches the request back to the Servlet container to resume processing.
* The DispatcherServlet is invoked again and processing resumes with the asynchronously produced result from the Callable.

The sequence for DeferredResult is very similar except it’s up to the application to produce the asynchronous result from any thread:

* Controller returns a DeferredResult and saves it in some in-memory queue or list where it can be accessed.
* Spring MVC starts async processing.
* The DispatcherServlet and all configured Filter’s exit the request processing thread but the response remains open.
* The application sets the DeferredResult from some thread and Spring MVC dispatches the request back to the Servlet container.
* The DispatcherServlet is invoked again and processing resumes with the asynchronously produced result.

For further background on the motivation for async request processing and when or why to use it please read [this blog post series](https://spring.io/blog/2012/05/07/spring-mvc-3-2-preview-introducing-servlet-3-async-support).

**async requests 的 Exception处理**

如果，一个由controller method返回的Callable在执行时 抛出了一个Exception，会发生什么？简短的答案是与一个controller method抛出一个异常时相同。会经历常规的异常处理机制。 长的解释是，当Callable抛出一个Exception时，Spring MVC会将该Exception分派到Servlet container，将其作为结果以及恢复request processing的引导，此时request processing会处理Exception，而非controller method return value。当使用DeferredResult时，你还可以选择是否调用setResult或者setErrorResult -- 传入Exception实例。

**拦截async requests**

一个HandlerInterceptor也可以实现AsyncHandlerInterceptor，以实现afterConcurrentHandlingStarted callback，当asynchronous processing开始时，会调用afterConcurrentHandlingStarted ，而非postHandle和afterComplete。

一个HandlerInterceptor也可以注册一个CallableProcessingInterceptor 或 一个 DeferredResultProcessingInterceptor，以更深度地集成asynchronous request的lifecycle，例如，handle 一个 timeout event。详见 AsyncHandlerInterceptor javadoc。

DeferredResult 类型，也提供了诸如 onTimeout(Runnable)、onCompletion(Runnable)之类的方法。详见javadoc。

当使用一个Callable时，你可以将其wrap进一个WebAsyncTask的实例，该实例也可以提供timeout和completion的方法注册。

**HTTP Streaming**

一个controller method可以使用DeferredResult和Callable来异步的produce其返回值，可被用于实现诸如[long polling](http://spring.io/blog/2012/05/08/spring-mvc-3-2-preview-techniques-for-real-time-updates/)之类的技术 -- 这样，服务器可以将一个事件尽快的push到客户端。

如果你想要在一个HTTP response上push多个事件会怎样？这就是与”Long Polling” 有关的技术，也就是HTTP Streaming。 Spring MVC通过ResponseBodyEmitter 返回值类型使其成为可能，该返回值类型可悲用于发送多个对象（而非使用@ResponseBody只发送一个 -- 这种更常见），每个被发送的对象都通过一个HttpMessageConverter被写入到response 。

例子：

[复制代码](javascript:void(0);)

@RequestMapping("/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();

[复制代码](javascript:void(0);)

注意，ResponseBodyEmitter 也可被用做ResponseEntity的body，以便定制response的status 和 headers。

**HTTP Streaming With Server-Sent Events**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-async-sse>

**HTTP Streaming Directly To The OutputStream**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-async-output-stream>

[复制代码](javascript:void(0);)

@RequestMapping("/download")

public StreamingResponseBody handle() {

return new StreamingResponseBody() {

@Override

public void writeTo(OutputStream outputStream) throws IOException {

// write...

}

};

}

[复制代码](javascript:void(0);)

**Configuring Asynchronous Request Processing**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-ann-async-configuration>

**3.5、测试controllers**

spring-test模块提供了第一等级的针对注解controller的测试支持。See [Section 15.6, “Spring MVC Test Framework”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/integration-testing.html#spring-mvc-test-framework).

**4、Handler mappings**

在之前的Spring版本中，用户必须要在web应用上下文中定义一个或者多个HandlerMapping beans 以将incoming web requests映射到合适的handlers。 随着annotated controllers的引入，现在一般可以不必那样做了，因为RequestMappingHandlerMapping 会自动在所有@Controller beans中查找@RequestMapping注解。然而，务必记住，所有的继承自AbstractHandlerMapping的HandlerMapping类，都有以下properties -- 你可以用来定制它们的行为：

**interceptors**，使用的拦截器列表。

**defaultHandler**，默认使用的handler -- 当handler mapping 没有一个匹配的handler时。

**order**，基于该property的值（Ordered接口），Spring将所有可用的handler mappings进行排序，并应用第一匹配的handler。

**alwaysUseFullPath**，如果设为true，Spring会在当前Servlet context中使用全路径来查找合适的handler。如果false（默认就是），会使用当前Servlet mapping内的路径。例如，如果一个Servlet被映射到/testing/\*，当设为true时，使用/testing/viewPage.html，否则，/viewPage.html。

**urlDecode**，默认true，自Spring 2.5起。如果你倾向于对比encoded paths，需要设为false。然而，HttpServletRequest总是以decoded形式暴露Servlet path。注意，当与encoded path对比时，Servlet path不会匹配。

配置拦截器的例子：

[复制代码](javascript:void(0);)

<beans>

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

<property name="interceptors">

<bean class="example.MyInterceptor"/>

</property>

</bean>

<beans>

[复制代码](javascript:void(0);)

**4.1 使用HandlerInterceptor拦截requests**

<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-handlermapping-interceptor>

spring的handler mapping机制包括handler interceptors，当你想要针对特定的requests应用特定的功能时，非常有用。

位于handler mapping内的interceptors，必须实现org.springframework.web.servlet.HandlerInterceptor （或者其实现/子类）。

该接口定义有三个方法preHandle(..) postHandle(..) afterHandle(..)。   见这里。（为知笔记的连接，不知道行不行，以后再说）

preHandle(..)方法会返回一个boolean值，如果false，会破坏执行链的处理过程（不再往下执行）。如果false，DispatcherServlet会认定该拦截器自身来处理请求（例如，渲染视图等），所以不会继续执行其他的拦截器和实际的handler。

拦截器可以在所有继承自AbstractHandlerMapping的类中设置，使用其interceptors属性！ 如下：

[复制代码](javascript:void(0);)

<beans>

<bean id="handlerMapping"

class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">

<property name="interceptors">

<list>

<ref bean="officeHoursInterceptor"/>

</list>

</property>

</bean>

<bean id="officeHoursInterceptor"

class="samples.TimeBasedAccessInterceptor">

<property name="openingTime" value="9"/>

<property name="closingTime" value="18"/>

</bean>

</beans>

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

private int openingTime;

private int closingTime;

public void setOpeningTime(int openingTime) {

this.openingTime = openingTime;

}

public void setClosingTime(int closingTime) {

this.closingTime = closingTime;

}

public boolean preHandle(HttpServletRequest request, HttpServletResponse response,

Object handler) throws Exception {

Calendar cal = Calendar.getInstance();

int hour = cal.get(HOUR\_OF\_DAY);

if (openingTime <= hour && hour < closingTime) {

return true;

}

response.sendRedirect("http://host.com/outsideOfficeHours.html");

return false;

}

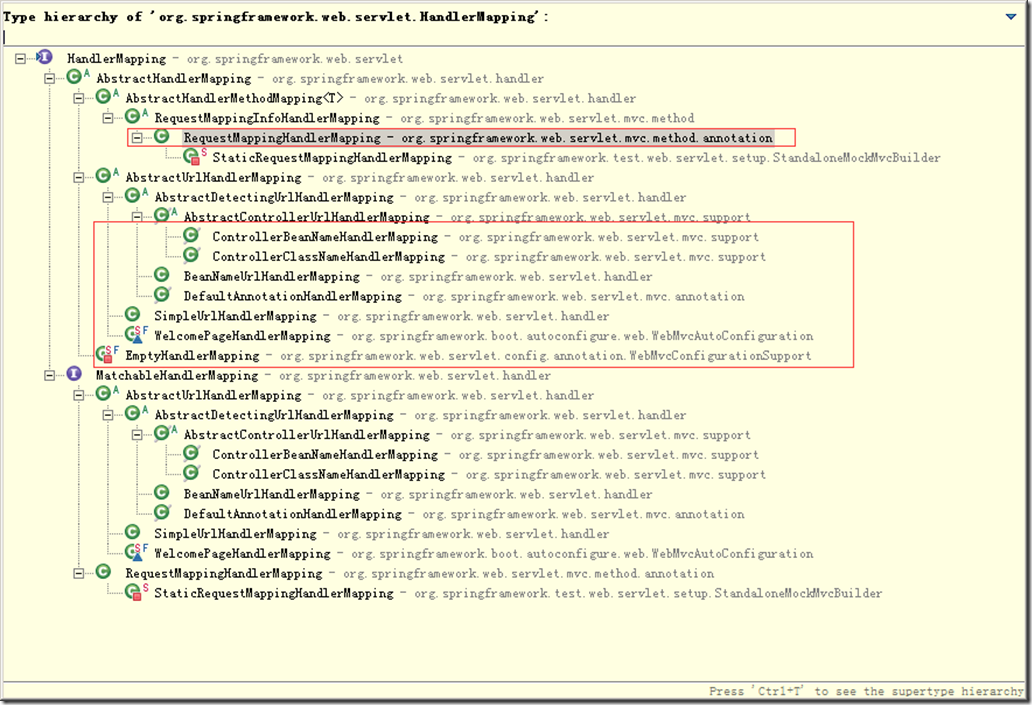
}

[复制代码](javascript:void(0);)

由**该映射**处理的所有请求，都会被 TimeBaseAccessInterceptor 拦截。  该拦截器的就是让你只能在办公时间访问。

注意：当使用RequestMappingHandlerMapping时，实际的handler是HandlerMethod的一个实例，该HandlerMethod会识别要被调用的特定的controller method。

我的补充：handler mapping这个过程，是将request与handler之间映射起来的过程。Spring提供的实现类，能用的也就这几个：

[](http://images2015.cnblogs.com/blog/870358/201612/870358-20161220130104073-1456461973.png)

--- 还有一个RequestMappingHandlerAdapter，不要混淆了。

如你所见，Spring的适配器类 HandlerInterceptorAdapter，让继承HandlerInterceptor更加简单。

在上面的例子中，被配置过的拦截器会被应用到所有由注解过的controller method处理的请求上。如果你想窄化一个拦截器应用的URL路径，你可以使用MVC 命名空间或者MVC Java config，或者声明MappedInterceptor类型的bean实例来完成。See [Section 22.16.1, “Enabling the MVC Java Config or the MVC XML Namespace”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-config-enable).

注意：postHandle方法，通常不能理想地配合@ResponseBody和ResponseEntity方法。    在这种情况下，一个HttpMessageConverter会写入并提交响应 -- 先于postHandle！从而导致无法修改响应，例如，添加一个header。这种情况下，可以实现ResponseBodyAdvice，然后要么将其声明为@ControllerAdvice bean，要么直接在RequestMappingHandlerAdapter中配置。