深入剖析 Spring Web 源码

（第二版）

作者： *罗伯特.李*
邮件： *robertleepeak@gmail.com*
博客： *http://blog.csdn.net/robertleepeak*

1 前言 4

2 WEB MVC介绍 5

2.1 MVC体系结构 5
2.2 WEB MVC体系结构 6
2.3 本章小结 7

3 SPRING WEB MVC工作流 8

3.1 组件以及组件的接口 8
3.2 组件间的协调通信 12
3.3 本章小结 13

4 SPRING WEB MVC的架构实现 14

4.1 DISPATCHERSERVLET的实现 14
4.1.1 通用Servlet和HTTP Servlet 15
4.1.2 派遣器Servlet及其父类 23
4.1.3 根共享环境的加载 45
4.1.4 其他Servlet 55
4.2 处理器映射，处理器适配器以及处理器的实现 55
4.2.1 横向剖析 56
4.2.1.1 基于简单控制器流程的实现 56
4.2.1.2 基于注解控制器流程的实现 87
4.2.1.3 基于HTTP请求处理器流程的实现 137
4.2.2 纵向剖析 157
4.2.2.1 处理器映射的实现架构 157
4.2.2.2 处理器适配器的实现架构 167
4.2.2.3 处理器的实现架构 169
4.2.2.4 拦截器的实现架构 196
4.2.2.5 HTTP消息转换器的实现架构 203
4.3 视图解析和视图显示 203
4.3.1 基于URL的视图解析器和视图 203
4.3.1.1 内部资源视图解析器和内部资源视图 214
4.3.1.2 瓦块视图解析器和瓦块视图 219
4.3.1.3 模板视图解析器和模板视图 222
4.3.1.4 Jasper报表视图解析器和Jaspter报表视图 225
4.3.1.5 XSLT视图解析器和XSLT视图 225
4.3.2 更多的视图解析器 233
4.3.2.1 Bean名视图解析器 233
4.3.2.2 内容选择视图解析器 234
4.3.2.3 资源绑定视图解析器 237
4.3.2.4 XML视图解析器 239
4.3.3 更多的视图 240
4.4 其他的SPRING WEB MVC组件 240

5 SPRING WEBFLOW 241

6 SPRING安全 242

6.1 横向剖析 242

6.1.1 首次登陆流程 251

6.1.2 登录失败流程 251

6.1.3 登录成功流程 251

6.1.4 退出登录流程 251

6.2 纵向剖析 251

6.2.1 过滤器和过滤器应用的顺序 251

6.2.2 领域模型 252

6.2.3 应用程序层次的安全 252

7 SPRING集成 253

8 SPRING远程调用, SPRING RMI和SPRING WEB SERVICE 254

9 与其他WEB MVC框架的对比与集成 255

10 APPFUSE的实现 256

11 SPRING WEB与设计模式 257

12 附录 258

12.1 如何开始学习SPRING源代码 258

12.1.1 J2SE/J2EE通用文档 258

12.1.2 Spring参考文档 258

12.1.3 Demo代码 258

12.1.4 Spring源代码 258

12.1.4.1 Spring Web相关项目组织结构 258

12.2 书写习惯 258

1 前言

Spring 是一个优秀的轻量级企业应用开发框架，是 Java 开发中最流行的工具之一，Spring 在 Java 企业开发中具有举足轻重的作用。

Spring 是基于 JavaBeans 的控制翻转（Inversion of Control，IoC）原则进行配置管理，使得应用程序的组建更加快捷简易。并且对 AOP 程序设计有较好的支持，使程序接口清晰易读，并且通过 AOP 实现了声明式的安全和事物。Spring 的抽象的数据存取层次支持多种 ORM 框架。并且对 J2EE 的其他技术提供了简单而有效的接口。使 Spring 能够和任何 J2EE 进行无缝的结合。

除此之外，Spring Web MVC 的体系架构更加优秀。本书从 Spring Web MVC 的架构原理和设计思想进行了全面剖析，从源代码分析入手，让读者更深入、更彻底地认识 Spring，领略 Spring 的架构之美和设计之美，从而使得我们在利用 Spring 进行开发时，不仅能知其然，还能知其所以然，从本质上提升我们对 Spring 的理解和开发水平。

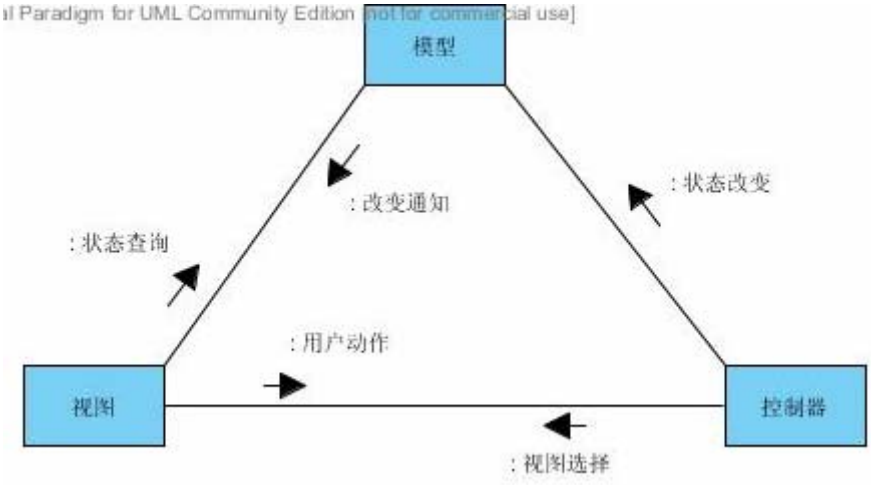
这不是一本 Spring 初学者的使用教材，这也不是一本 Spring 应用教科书，这是一本适合于想要深入研究和学习 Spring 实现和架构的学术性书籍。这本书专注于讨论 Spring Web 技术，尤其是 Spring Web MVC 的架构实现。希望对读者有所帮助。

2 Web MVC介绍

这一章，我们将首先介绍经典的 MVC 体系结构，通过对比 MVC 和 Web MVC 的区别，来深入理解 Web MVC 的由来和特点。

2.1 MVC体系结构

首先，我们介绍一下 MVC 的体系结构。MVC 是软件工程中的一种软件架构模式。它把软件系统分为三个基本部分：模型，视图和控制器。如图表 2-1。



图表 2-2

MVC 体系结构实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外此模式通过对复杂度的简化使程序结构更加直观。软件系统通过对自身基本部份分离的同时也赋予了各个基本部分应有的功能。

控制器

- 定义应用程序行为和流程
- 映射用户动作到模型改变
- 选择用于响应的视图

控制器起到不同层面间的组织作用，用于控制应用程序的流程。它处理事件并作出响应。响应包括更新数据模型和选择视图解释数据模型。

视图

- 解释模型
- 请求模型更新
- 发送用户动作到控制器

视图层能够实现数据有目的的显示（理论上，这不是必需的）。在视图中一般没有程序上的逻辑。为了实现视图上的刷新功能，视图需要访问它监视的数据模型（Model），因此应该事先在被它监视的数据那里注册。

模型

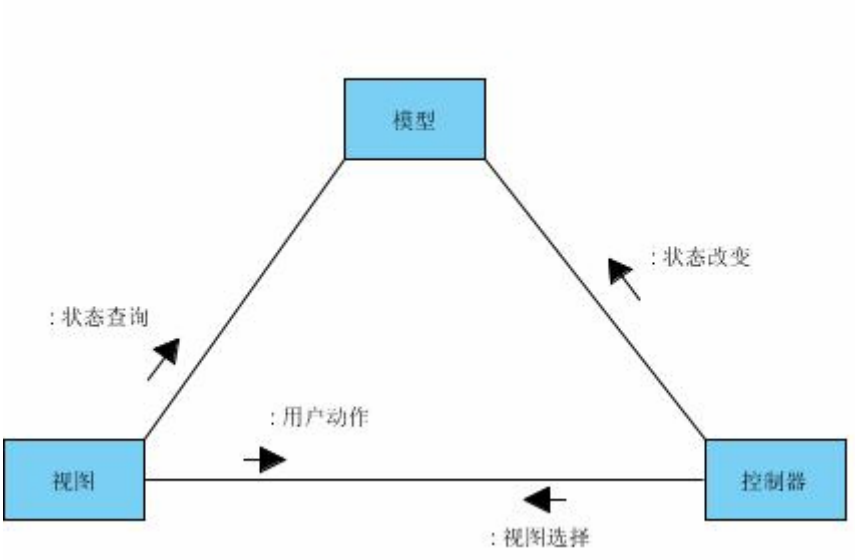
- 封装应用程序状态
- 提供应用程序功能
- 响应状态查询
- 通知视图改变

模型用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。“模型”有对数据直接访问的权力，例如对数据库的访问。“模型”不依赖“视图”和“控制器”，也就是说，模型不关心它会被如何显示或是如何被操作。但是模型中数据的变化一般会通过一种刷新机制被公布。为了实现这种机制，那些用于监视此模型的视图必须事先在此模型上注册，从而，视图可以了解在数据模型上发生的改变。

上面我们介绍了 MVC 体系结构的组件以及组件的职责，我们得知，MVC 的各个组件是协同工作，而又互相独立，这样最大程度上实现了组件的重用和高可扩展性。MVC 主要是应用在传统的 C/S 体系架构中。

2.2 Web MVC体系结构

随着 B/S 体系结构的应用程序的流行和快速发展，MVC 体系结构思想被应用到 Web 应用程序设计。Web 应用程序是基于 HTTP 协议的，而 HTTP 协议的最大特点就是无连接的。对于一个 Web 客户端程序，每次和服务器的通信都是通过一次完整的 HTTP 请求和响应来完成的。我们无法实现 C/S 应用程序中实现的在视图层注册监听器来监听模型层改变的通知。取而代之的是视图每次需要主动的查询用户数据改变。如图表 2-3。



图表 2-4

我们能看出，唯一不同的就是模型层不再通知视图层是否存在着状态改变。而是要求视图层主动的去通过控制器查询模型层的改变。如果模型状态改变，控制器则选择一个新的视图解释模型的改变。通过 MVC 在 Web 应用程序的应用，使 Web 应用程序同样可以有层的概念并且有较好的体系结构，易于维护和容易扩展。

2.3 本章小结

这一章，介绍 MVC 的体系结构和工作原理，通过对比 MVC 和 Web MVC 来深入理解 Web MVC 的由来和特点。

下一章，我们将从整体架构上分析 Spring 是如何实现 Web MVC 体系结构的。

3 Spring Web MVC工作流

这一章，我们将描述 Spring Web MVC 的各个组件，组件接口，以及各个组件之间如何协调通信，进而理解 Spring Web MVC 框架的基本工作原理。

3.1 组件以及组件的接口

Spring Web MVC 是由若干组件组成的，这些组件相互独立又相互协调工作共同完成 Spring Web MVC 工作流。每个组件都有清晰的接口定义，接口后面都有一个设计良好的类实现体系结构，清晰的抽象出公用的逻辑并且实现在通用的抽象类里，同时提供常用的具体实现类。进而实现一个清晰的，高可扩展的，可插拔的 Web MVC 体系结构。

下面我们介绍这些典型的组件，组件的功能以及组件所定义的接口。

- 派遣器处理器 Servlet 对象(Dispatcher Servlet)

  派遣器处理器 Servlet 对象(Dispatcher Servlet)是 Spring Web MVC 中最核心组件，它是一个实现类而并不是一个接口。

  从类的继承角度来看，派遣器处理器对象(Dispatcher Servlet)最终继承自 HttpServlet 对象。通过若干个抽象类的划分，是派遣器处理器对象(Dispatcher Servlet)的类体系接口清晰明了，任务分明，容易扩展，我们将在后续章节详细分析派遣器处理器对象(Dispatcher Servlet)的类体系接口。

  初始化时， 它通过内部的 Spring Web 应用程序环境，找到相应的 Spring Web MVC 的各个组件，如果这些组件没有显式配置，Spring Web MVC 将会根据默认加载策略初始化各个模块的默认实现。

  服务时，它通过一套注册的处理器映射对象(HandlerMapping)找到一个处理器对象(Handler)，然后，从一套注册的处理器适配器对象（HandlerAdapter）中找到一个支持这个处理器对象(Handler)的处理器适配器对象（HandlerAdapter），并且通过它把控制流转发给这个处理器对象(Hanlder)，处理器对象(Handler)结束业务逻辑的调用后把模型数据和逻辑视图传回派遣器处理器 Servlet 对象(Dispatcher Servlet)。

  派遣器处理器 Servlet 对象(Dispatcher Servlet)再通过视图解析器对象(ViewResolver)得到真正的视图对象（View），把控制权交给视图对象（View），同事传入模型数据，视图对象（View）会按照一定的视图层定义展现这些数据到用户响应对象里。

  下一小节中我们将对这个流程进行完整的描述，并且在下一章中详细的分析 Servlets 的类体系结构。

- 处理器映射对象(HandlerMapping)

处理器映射对象(HandlerMapping)是用于映射一个请求对象(Request)到一个处理器对象(Handler)。一个 Spring Web MVC 实例可能会包含多个处理器映射对象(HandlerMapping)。按照处理器映射对象所在的顺序，第一个返回非空处理器执行链对象(HandlerExecutionChain)的处理器映射，会被当作为有效的处理器映射对象。处理器执行链对象(HandlerExecutionChain)包含一个处理器对象(Hanlder)和一组能够应用在处理器对象(Handler)上的拦截器对象(Interceptor).

它的接口定义如下,

图表 3-1

在这个接口中，通过输入一个 HTTP 请求对象(HttpServletRequest)给方法(getHandler)，这个方法就会输出一个处理器执行链对象(HandlerExecutionChain)。进而我们能够从处理器执行链对象(HandlerExecutionChain)得到一个处理器对象(Handler)。

- 处理器适配器对象(HandlerAdaptor)

处理器适配器(HandlerAdaptor)是用来转接一个控制流到一个指定类型的处理器对象（Handler）。一个类型的处理器对象（Handler）通常会对应一个处理器适配器对象(HandlerAdaptor)的一个实现。处理器适配器对象(HandlerAdaptor)能够判断自己是否支持某个处理器对象（Hanlder）。如果一个处理器适配器对象(HandlerAdaptor)支持这种类型的处理器(Handler)，那么这个处理器适配器对象(HandlerAdaptor)就可以使用这个处理器对象（handler）处理当前的这个 HTTP 请求。

然而，处理器对象(Hanlder)是一个通用的 Java 对象类型，这样做是允许 Spring Web MVC 可以任意的去集成其他框架的处理器对象(handler)，只需要为这个需要支持的处理器对象(Handler)提供一个客户化的处理器适配器对象(HanlderAdaptor)，就可以在不改变任何上层派遣器 Servlet 对象(DispatcherServlet)代码以及下层控制器对象(Controller)代码的前提下，实现任意集成。后来证明，基于注释的控制器，以及 Spring Web Flow 都是通过这样的适配器的实现来完成和 Spring Web MVC 工作流集成的，我们将在后面章节详细讨论。

它的接口定义如下,

图表 3-2

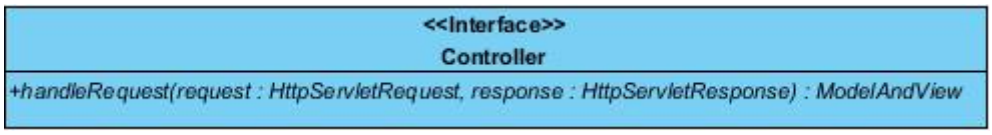
在这个接口中，通过输入一个处理器对象（Handler）给 supports()方法，这个方法就会返回是否支持这个处理器。通过传入一个 HttpServletRequese，HttpServletResponse 和一个 Hanlder 对象给 handle()方法，这个方法就会就会传递控制权给处理器对象(Handler)。当处理器对象（Handler）处理这个 HTTP 请求后返回数据模型和逻辑视图名称的组合对象，处理器适配器会把这个返回结果进而返回给派遣器 Servlet(DispatcherServlet)。getLastModified 方法是用来处理一个带有 lastModified 头信息的 HTTP 请求的。不是所有的处理器适配器（HandlerAdapter）都需要支持这个方法的。我们将在下一章详细讨论它的实现。

- 处理器对象 - 控制器对象（Handler - Controller)

处理器对象（Handler）是用于处理业务逻辑的一个基本单元。它通过传入的 HTTP 请求对象(HttpServletRequese)来决定如何处理业务逻辑和执行哪些服务，执行服务后返回相应的模型数据和逻辑视图名。处理器对象(Handler)是一个通用的对象类型。控制流是由处理器适配器(HandlerAdaptor)传递给处理器的。所以，一个类型的处理器对象(Hanlder)一定会对应一个支持的处理器适配器(HandlerAdaptor)。这样可以实现，处理器适配器（HandlerAdaptor）和处理器(Handler)的任意插拔。

控制器是 Spring Web MVC 中最简单的一个处理器。这个处理器有清晰的接口定义，通常这个类型的处理器对象(Handler)是通过一个简单控制处理适配器对象(SimpleControllerHandlerAdapter)来传递控制的。

它的接口定义如下,



图表 3-3

在这个接口中，通过输入一个 HTTP 请求对象（HttpServletRequese）和 HTTP 响应对象(HttpServletResponse)给方法 handleRequest()方法，这个方法就会返回一个处理后的模型数据和逻辑视图名的组合对象。这个对象将通过处理器适配器(HandlerAdapter)进而返回给派遣器 Servlet(DispatcherServlet)。

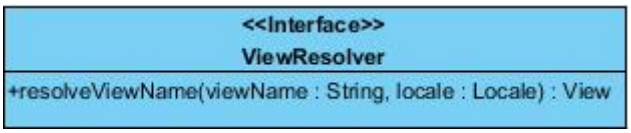
Spring Web MVC 之所以具有高可扩展性，在于处理器适配器（HandlerAdapter）和处理器(Handler)的设计。这样可以让任意的一个处理器对象插入到 Spring Web MVC 的体系

结构中，是它具有无限的扩展性。一个例子就是自从 Spring 2.5 新引进的注释方法处理器适配器对象(AnnotationMethodHandlerAdapter)和注释处理器。Spring Web Flow 也是器 Spring Web MVC 扩展的一个典型例子，我们讲在下面章节中详细讨论他们的实现。

- 视图解析器对象(ViewResolver)

视图解析器对象(ViewResolver)是用于映射一个逻辑视图名称到一个真正的视图对象（View）。当控制器对象(Controller)处理业务逻辑之后，通常会返回需要显示的数据模型对象和视图的逻辑名称，这样就需要一个支持的视图解析器对象(ViewResolver)解析出一个真正的视图对象(View)，然后传递控制流给这个视图对象。

它的接口定义如下,



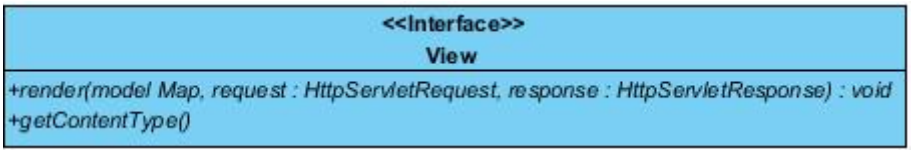
图表 3-4

在这个接口中，通过输入一个逻辑视图名称和本地对象(Locale)给 resolveViewName 方法，这个方法就会返回一个事实上的物理视图对象（View）。本地对象(Locale)可以用来查找本地化的资源或者资源包。

- 视图对象(View)

View 是用来把模型数据通过某种显示方式反馈给客户，通常是通过执行 JSP 页面来完成的。也可以通过其他的更复杂的显示技术来完整这个展示过程，例如，JSTL 视图，Tiles 视图，报表视图和二进制文件视图等等。

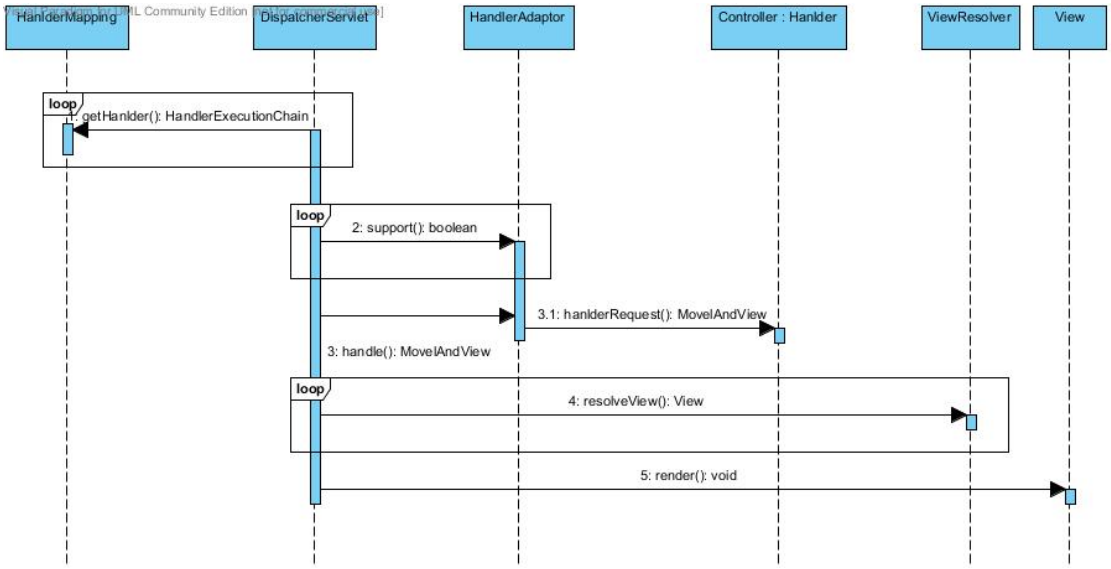
它的接口定义如下,



图表 3-5

在这个接口中，通过输入 HTTP 请求对象（HttpServletRequese），HTTP 响应对象(HttpServletResponse)和一个模型 Map 对象，这个组件就会把模型 Map 通过一定的显示方式包含到客户的 HTTP 响应对象(HttpServletResponse)，这就是提供给用户请求的最终响应。getContentType()能够返回这个视图所支持的内容类型，例如，XML，JSON，text/html 等等。

3.2 组件间的协调通信



图表 3-6

众所周知，一个 HTTP 请求发送到 Web 容器，Web 容器就会封装一个 HTTP 请求对象(HttpServletRequest)，这个对象包含所有的 HTTP 请求信息，例如，HTTP 参数以及参数值，HTTP 请求头的各种元数据。同时，Web 容器会创建一个 HTTP 响应对象（HttpServletResponse），用以发送 HTTP 响应给客户端用户。然后，Web 容器传递 HTTP 请求对象(HttpServletRequest)和 HTTP 响应对象（HttpServletResponse）给 Servlet 对象的 service()方法。

实际上，Spring Web MVC 的入口就是一个客户化的 Servlet，称为派遣器 Servlet 对象(DispatcherServlet)。这个派遣器 Servlet 对象（DispatcherServlet）得到 HTTP 请求对象(HttpServletRequest)和 HTTP 响应对象（HttpServletResponse）后，一个典型的 Spring Web MVC 工作流就开始了。

派遣器 Servlet 对象（DispatcherServlet）首先查找所有注册的处理器映射器对象（HandlerMapping），然后，遍历所有的处理器映射器对象（HandlerMapping），直到一个处理器映射器对象（HandlerMapping）返回一个非空的处理器执行链对象(HandlerExecutionChain)。那么，处理器执行链对象(HandlerExecutionChain)就包含着一个需要处理当前 HTTP 请求的一个处理器对象（Handler）。如图表 3-6 第 1 步。

这里，一个处理器对象(Handler)被设计成了一个通用的对象类型，所以，这里需要一个处理器适配器（HandlerAdaptor）去派遣这个控制流到一个处理器对象(Handler)，因为只有支持这种类型的处理器（Handler）的处理器适配器(HandlerAdapter)才知道如何去传递控制流给这个类型的处理器（Hanlder）。

拿到了处理器对象(Hanlder)以后，派遣器 Servlet 对象（DispatcherServlet）则查找所有注册的处理器适配器对象（HandlerAdapter），然后，遍历所有的处理器适配器对象

（HandlerAdapter）查询是否有一个处理器适配器对象（HandlerAdapter）支持这个处理器对象（Handler）。如图表 3-6 第 2 步。

如果有这样的一个处理器适配器对象(HandlerAdapter)，则派遣器 Servlet 对象（DispatcherServlet）将控制权转交给这个派遣器适配器对象(HandlerAdapter)。如图表 3-6 第 3 步。派遣器适配器对象（HandlerAdapter）和真正的处理器对象(Handler)是成对出现的，所以，这个支持的处理器适配器对象（HandlerAdapter）则知道如何去使用这个处理器(Handler)去处理这个请求。最简单的一个处理器则是控制器对象（Controller）。处理器适配器对象(HandlerAdapter)将传递 HTTP 请求对象（HttpServletRequest）和 HTTP 响应对象（HttpServletResponse）给控制器，并且期待控制器返回模型和视图对象（ModelAndView）。如图表 3-6 第 3.1 步。这个模型和视图对象（ModelAndView）对象包含着一组模型数据和视图逻辑名称，并且最终返回给派遣器 Servlet 对象（DispatcherServlet）.

派遣器 Servlet 对象（DispatcherServlet）然后查找所有注册的视图解析器对象，并且遍历所有的视图解析器对象（ViewResolver），直到一个视图解析器对象（ViewResolver）返回一个物理的视图对象（View）。如图表 3-6 第 4 步。

最后，派遣器 Servlet（DispatcherServlet）把得到的一组模型数据传递给得到的物理视图对象（View）。如图表 3-6 第 5 步。然后，视图对象则会使用响应的表现层技术，把模型数据展现成 UI 界面，并且通过 HTTP 响应对象（HttpServletResponse）发送给 HTTP 客户。

3.3 本章小结

这一章，我们介绍了 Web MVC 在 Spring 中是如何实现的。具体分析了 Spring Web MVC 架构中的各个组件，组件的功能，以及组件的接口。最后，大体的分析了 Spring Web MVC 的工作流是如何传递和工作的。

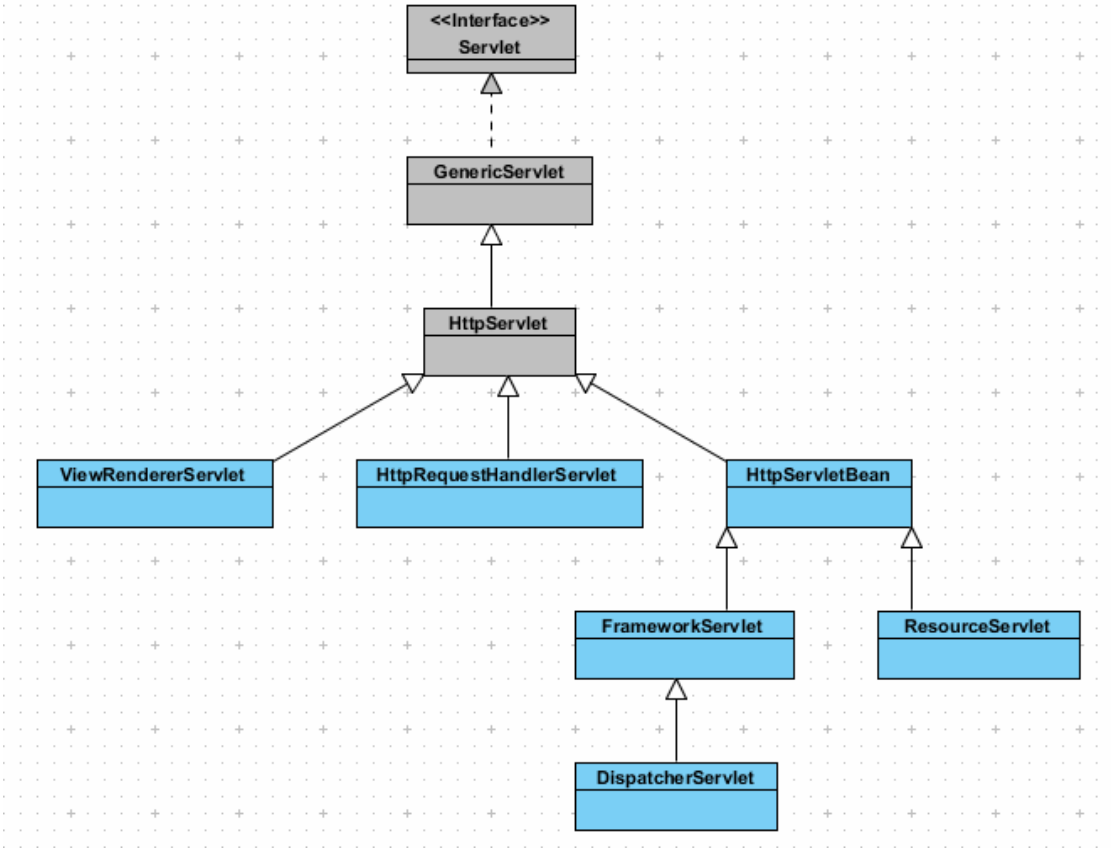
下一章，我们将深入抛析 Spring Web MVC 的架构的具体实现。

4 Spring Web MVC的架构实现

Spring 是一个轻量级 J2EE 框架。它可以运行在任何一个 Web 容器上。事实上，Spring 的入口点就是实现或者继承了 Web 容器规范中的 Servlet, 监听器(Listener)和过滤器(Filter)。这一章我们将详细讨论 Spring Web MVC 架构的具体实现。

4.1 DispatcherServlet的实现

DispatcherServlet 是一个经过多个层次最终继承自 Servlet 规范中的 HttpServlet, 进而实现 Servlet 规范中定义的 Servlet 接口。这些继承和实现组成了一个复杂的树形结构，在树形结构中的每个层次的类完成一个特定的初始化功能，服务功能或者清理资源的功能，每个层次的类之间分工合理，易于扩展。如图表 4 - 1，



图表 4-1

Servlet 是 Servlet 规范中规定的一个服务器组件的接口，任何一个可以处理用户请求的服务器组件需要实现这个接口，Web 容器就是根据 URL 到 Servlet 的映射派遣一个 HTTP 请求到这个 Servlet 组件的实现，进而对这个 HTTP 请求进行处理，并且产生 HTTP 响应。

通用 Servlet(GenericServlet)是 Servlet 的一个抽象实现。这个实现是和协议无关的。它提供了 Servlet 应该具有的基础功能。例如，保存 Servlet 配置，为后来的操作提供初始化参数和

信息等等。

HTTP Servlet(HttpServlet)是针对 HTTP 协议对通用 Servlet 的实现。它实现了 HTTP 协议的一些基本操作。例如，根据 HTTP 请求的方法分发不同类型的 HTTP 请求到不同的方法进行处理。对于简单的 HTTP 方法(HEAD, OPTIONS, TRACE)提供了通用的实现，这些实现在子类中通常是不需要重写的。而对其他的业务服务类型的方法(GET, POST, PUT, DELETE)提供了占位符方法。子类应该有选择的根据业务逻辑重写这些服务类型方法的实现。

HTTP Servlet Bean(HttpServletBean)是 Spring Web MVC 的一个抽象实现。它提供了一个特殊的功能，可以将 Servlet 配置的初始化参数作为 Bean 的属性，自动赋值给 Servlet 的属性。子类 Servlet 的很多属性都是通过这个功能进行配置的。

Framework Servlet(FrameworkServlet)也是一个抽象的实现。在这个是西安层次上，它提供了加载一个对应的 Web 应用程序环境的功能。这个 Web 应用程序环境可以存在一个根环境，这个根环境可以是共享的也可以是这个 Servlet 或者几个 Servlet 专用的。它也提供了功能将 HTTP GET, POST, PUT, DELETE 方法统一派遣到 Spring Web MVC 的控制器方法进行派遣。在派遣前到处请求环境等信息到线程的局部存储。

派遣器 Servlet(DispatcherServlet)是这个继承链中最后一个类，它是 Spring Web MVC 的核心实现类，它在框架 Servlet 中加载的 Web 应用程序环境中查找一切 Spring Web MVC 所需要的并且注册的组件，如果一个需要的 Spring Web MVC 组件没有注册，则通过缺省策略的配置创建并且初始化这些组件。在一个 HTTP 请求被派遣的时候，它使用得到的 Spring Web MVC 组件进行处理和响应。具体的处理和相应将在下面的章节具体分析。

资源 Servlet(ResourceServlet)是一个用来存取 Web 应用中内部静态资源的一个 Servlet 实现。

HTTP 请求处理器 Servlet(HttpRequestHandlerServlet)是用来派遣一个请求直接到 HttpRequestHandler 的特殊的 Serlvet。可以用以基于 HTTP 协议的远程调用的实现。

视图绘制 Servlet(ViewRenderServlet)是用来与 Portlet 集成而设计的一个实现类。

从上图可以看出，前三个类被标识为灰色，因为他们都是 Servlet 规范中规定接口或者实现，并不是 Spring 的实现。而后面的所有类都是 Spring Web MVC 的实现类。下面的第一小节中我们讲分析 Servlet 规范中对 HTTP 协议的实现。第二小节则深入剖析 Spring Web MVC 控制器（派遣器 Servlet）的实现。

4.1.1 通用Servlet和HTTP Servlet

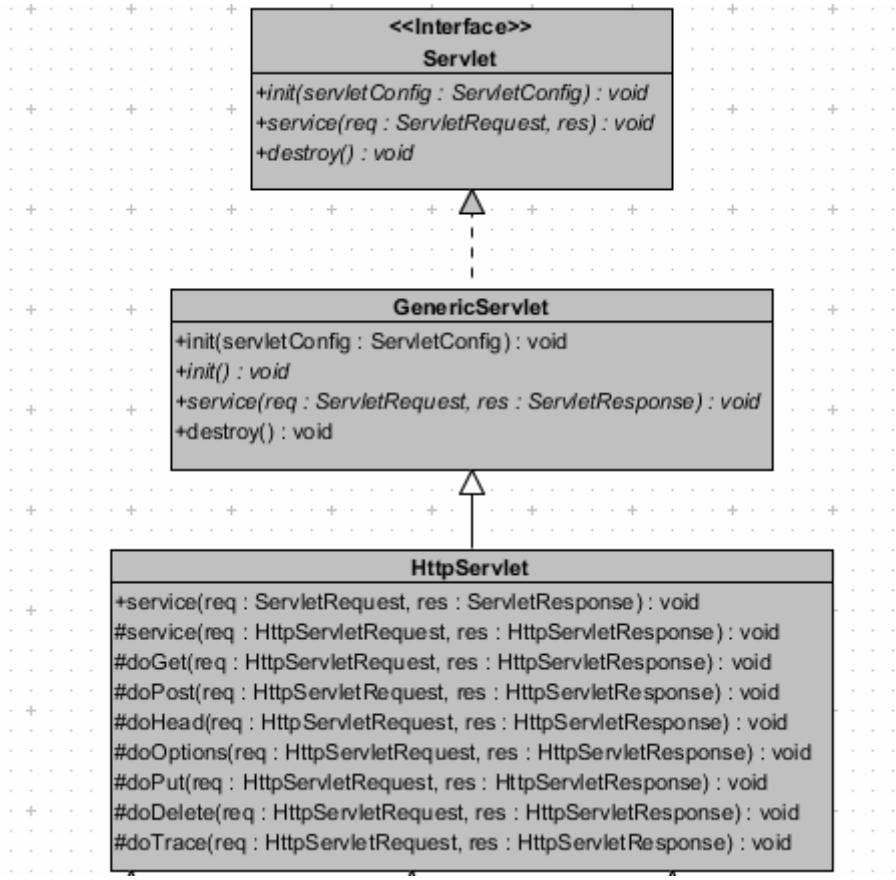
HTTP（Hyper Text Transfer Protocol）是超文本传输协议的缩写，它用于传送 WWW 方式的数据，关于 HTTP 协议的详细内容请参考 RFC2616。HTTP 协议采用了请求/响应模型。客户端向服务器发送一个请求，请求头包含请求的方法、URI、协议版本、以及包含请求修饰符、客户信息和内容的类似于 MIME 的消息结构。服务器以一个状态行作为响应，相应的内容包括消息协议的版本，成功或者错误编码加上包含服务器信息、实体元信息以及可能的

实体内容。

HTTP 协议支持各种类型的方法，其中包括，GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE。

- GET 方法将请求参数放在请求的头中，请求服务器做某些服务操作并返回响应。
- POST 方法从客户机向服务器传送数据，并可能要求服务器做出某些服务操作进行响应。
- PUT 方法请求将一个资源放在服务器的某个路径下。
- DELETE 方法请求将服务器某路径下的一个资源删除。
- HEAD 要求服务器查找某对象的头信息，包括应该包含的请求体的长度，而不是对象本身。
- OPTION 方法用来查询服务器的实现信息。
- TRACE 多数情况下用在调试目操作上。

下图是 Servlet 规范的接口和实现类的继承结构，每个类的方法包括实体方法，抽象方法或者占位符方法。抽象方法和占位符方法由子类实现。



图表 4-2

从上图我们可以看出，Servlet 接口定义了 3 个重要的接口方法，init()方法是在 Sevlet 初始化的时候调用的，提供给 Servlet 组件进行初始化自己的机会。与此相对应， detroy()方法是在 Servlet 析构时候调用的，提供给 Servlet 组件进行释放使用过的资源的机会。而 sevice()

方法是用来处理每一个 Web 容器传递进来的请求与响应的。

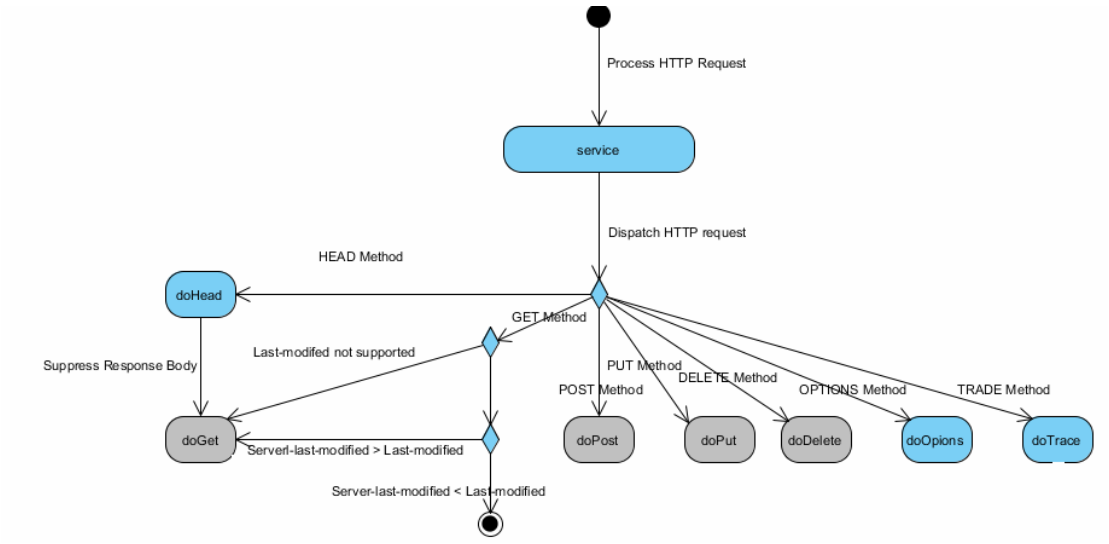
通用 Servlet 实现了 Servlet 的接口方法 init(), 方法中保存了 Servlet 容器传递过来的 ServletConfig 对象。如下图程序片段所示，

```
public void init(ServletConfig config) throws ServletException {
    //保存Servlet配置对象，对于处理一个HTTP请求的许多操作都需要的Servlet配置所包含的信息，例如，
    Servlet名字，Servlet配置的参数等等
    this.config = config;

    //代理到另外一个无参数的init方法，这个方法是一个抽象方法，它是一个占位符，提供给子类重写并初始化的
    机会
    this.init();
}
```

通用Servlet的方法service()是一个显示定义的抽象方法，要求实现类必须重写这个方法的实现。因为不同的Servlet实现会依赖不同的协议，实现各不相同。D

destroy()是一个方法占位符，子类可以有选择的实现进而进行资源的清理。

HTTP Servlet 正如我们所愿，实现了通用 Servlet 的 service()方法，根据 HTTP 请求中所标识的方法，把 HTTP 请求派遣到不同的处理方法中。如下图所示，



图表 4-3

这些不同的方法有不同的实现，这些处理方法中的大部分是占位符，但是，它为 doOptions()和 doTrace()提供了具体实现，因为对于不同的 HTTP Servlet 组件，这两个方法的行为基本是不变的。他们都是用于返回服务器信息和调试目的。如下图代码注释，

```
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException
```

```
{
    HttpServletRequest  request;
    HttpServletResponse response;

    try {
        //既然是HTTP协议绑定的Serlvet，强制转换到HTTP的领域模型
        request = (HttpServletRequest) req;
        response = (HttpServletResponse) res;
    } catch (ClassCastException e) {
        //如果传入的HTTP请求和HTTP响应不是HTTP的领域模型，则抛出Servlet异常，这个异常会被Servlet容
        器所处理
        throw new ServletException("non-HTTP request or response");
    }

    //如果传入的请求和响应是预期的HTTP请求和HTTP响应，则调用service()方法。
    service(request, response);
}

protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    //从HTTP请求中取得这次请求所使用的HTTT方法
    String method = req.getMethod();

    if (method.equals(METHOD_GET)) {
        //如果这次请求使用GET方法

        //取得这个Servlet的最后修改的时间
        long lastModified = getLastModified(req);
        if (lastModified == -1) {
            //-1代表这个Servlet不支持最后修改操作，直接调用doGet()进行处理HTTP GET请求
            doGet(req, resp);
        } else {
            //如果这个Servlet支持最后修改操作，取得请求头中包含的请求的最后修改时间
            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);

            if (ifModifiedSince < (lastModified / 1000 * 1000)) {
                //如果请求头中包含的修改时间早于这个Servlet的最后修改时间，说明这个Servlet自从客户上
                一次HTTP请求已经被修改了，设置最新修改时间到响应头中
                maybeSetLastModified(resp, lastModified);

                // 调用doGet进行进行处理HTTP GET请求
                doGet(req, resp);
            } else {
```

```
                //如果请求头中包含修改时间晚于这个Servlet的最后修改时间，说明这个Servlet自从请求的最
                后修改时间后没有更改过，这种情况下，仅仅返回一个HTTP响应状态SC_NOT_MODIFIED
                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
        }

    } else if (method.equals(METHOD_HEAD)) {
        //如果这次请求使用POST方法

        //如果这个Servlet支持最后修改操作，则设置这个Servlet的最后修改时间到响应头中
        long lastModified = getLastModified(req);
        maybeSetLastModified(resp, lastModified);

            //和对HTTP GET方法处理不同的是，无论请求头中的修改时间是不是早于这个Sevlet的最后修改时间，
            都会发HEAD响应给客户，因为HTTP HEAD响应是用来查询Servlet头信息的操作
        doHead(req, resp);

    } else if (method.equals(METHOD_POST)) {
        //如果这次请求使用POST方法
        doPost(req, resp);

    } else if (method.equals(METHOD_PUT)) {
        //如果这次请求使用PUT方法
        doPut(req, resp);

    } else if (method.equals(METHOD_DELETE)) {
        //如果这次请求使用DELETE方法
        doDelete(req, resp);

    } else if (method.equals(METHOD_OPTIONS)) {
        //如果这次请求使用OPTIONS方法
        doOptions(req,resp);

    } else if (method.equals(METHOD_TRACE)) {
        //如果这次请求使用TRACE方法
        doTrace(req,resp);

    } else {
            //如果这次请求是其他未知方法，返回错误代码SC_NOT_IMPLEMENTED给HTTP响应，并且显示一个错误
            消息，说明这个操作是没有实现的
        String errMsg = lStrings.getString("http.method_not_implemented");
        Object[] errArgs = new Object[1];
        errArgs[0] = method;
        errMsg = MessageFormat.format(errMsg, errArgs);
```

```
        resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
    }
}
```

从上面两个方法的实现中我们可以看到，HTTP Servlet 根据不同的 HTTP 方法进行了 HTTP 请求的分发。这样，不同方法的请求会使用不同的处理方法进行处理。事实上 doGet() doPost(), doPut(), doDelete()都是占位符实现，子类应该有选择的重写这些方法来实现真正的服务逻辑。Spring Web MVC 就是通过重写这些方法，开始控制流的实现的。

下面是 doGet()方法的代码注释。

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    //取得请求头包含的HTTP协议版本
    String protocol = req.getProtocol();

        //直接发送错误消息，可见，一个子类需要重写这些占位符方法doGet(), doPost(), doPut(), doDelete()
        中的一个或者多个
    String msg = lStrings.getString("http.method_get_not_supported");
    if (protocol.endsWith("1.1")) {
        //如果是HTTP 1.1, 发送SC_METHOD_NOT_ALLOWED
        resp.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, msg);
    } else {
        //如果是HTTP的更早版本则发送SC_BAD_REQUEST
        resp.sendError(HttpServletResponse.SC_BAD_REQUEST, msg);
    }
}
```

doHead()方法的实现通过对 HTTP 响应类进行包装，实现了 NoBodyReponse 类，这个类忽略了对 HTTP 响应体的输出。重用了 doGet()方法的实现，并且保留了 HTTP 头信息的输出。所以，如果一个子类 Servlet 重写了 doGet()方法，这个方法 doHead()是不需要重写的。代码注释如下，

```
protected void doHead(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    //构造一个特殊的响应类，这个类内部忽略了所有的响应体的输出
    NoBodyResponse response = new NoBodyResponse(resp);

    //重用doGet()处理器罗杰
    doGet(req, response);

```

```
    //设置响应体的字节大小，尽管响应体并没有输出，但是客户端可能关系这个信息
    response.setContentLength();
}
```

doPost(), doPut(), doDelete()方法的实现和doGet()方法的实现是类似的，他们都是一个占位符的实现，子类Servlet需要有选择的进行重写进而实现真正需要的HTTP服务。

然而， doOptions()和 doTrace()对任何 Servlet 的实现，基本是不变的，他们是用来查询服务器信息和调试所用，他们的实现如下，

```
//这个方法doOptions()设置支持的HTTP方法名称到相应头中
protected void doOptions(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException
{
    //取得当前这个Servlet以及父类Servlet声明的所有方法，这些方法不包括本类HTTP Servlet所生命的方法
    Method[] methods = getAllDeclaredMethods(this.getClass());

    //初始化的时候，假设它不支持任何HTTP方法
    boolean ALLOW_GET = false;
    boolean ALLOW_HEAD = false;
    boolean ALLOW_POST = false;
    boolean ALLOW_PUT = false;
    boolean ALLOW_DELETE = false;
    boolean ALLOW_TRACE = true;
    boolean ALLOW_OPTIONS = true;

    //根据子类Servlet是否重写了HTTP Servlet的占位符方法，判断是否这个Servlet实现支持这种HTTP方法，
例如，如果子类Servlet实现了doGet(), 然后HTTP GET方法是支持的
    for (int i=0; i<methods.length; i++) {
        //遍历得到的所有生命的方法
        Method m = methods[i];

        //如果名字是doGet(), doPost(),  doPut()或者doDelete(), 它支持相应的方法
        if (m.getName().equals("doGet")) {
            ALLOW_GET = true;
            ALLOW_HEAD = true;
        }
        if (m.getName().equals("doPost"))
            ALLOW_POST = true;
        if (m.getName().equals("doPut"))
            ALLOW_PUT = true;
        if (m.getName().equals("doDelete"))
            ALLOW_DELETE = true;
```

```
    }

    //把支持的HTTP方法名称拼接成逗号分割的字符串，例如，“GET, POST”, “GET, POST, PUT, DELETE”
    String allow = null;
    if (ALLOW_GET)
        if (allow==null) allow=METHOD_GET;
    if (ALLOW_HEAD)
        if (allow==null) allow=METHOD_HEAD;
        else allow += ", " + METHOD_HEAD;
    if (ALLOW_POST)
        if (allow==null) allow=METHOD_POST;
        else allow += ", " + METHOD_POST;
    if (ALLOW_PUT)
        if (allow==null) allow=METHOD_PUT;
        else allow += ", " + METHOD_PUT;
    if (ALLOW_DELETE)
        if (allow==null) allow=METHOD_DELETE;
        else allow += ", " + METHOD_DELETE;
    if (ALLOW_TRACE)
        if (allow==null) allow=METHOD_TRACE;
        else allow += ", " + METHOD_TRACE;
    if (ALLOW_OPTIONS)
        if (allow==null) allow=METHOD_OPTIONS;
        else allow += ", " + METHOD_OPTIONS;

    //把支持的方法拼接成的字符串设置到HTTP协议的相应头中，这个值的key是"Allow"
    resp.setHeader("Allow", allow);
}

//这个方法返回一个字符串到HTTP响应体里面，这个字符串包含请求URL，版本信息以及请求的头信息，主要是用来
调试
protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{

    int responseLength;


    //连接URI字符串和协议版本信息字符串
    String CRLF = "\r\n";
    String responseString = "TRACE "+ req.getRequestURI()+
        " " + req.getProtocol();

    Enumeration reqHeaderEnum = req.getHeaderNames();
```

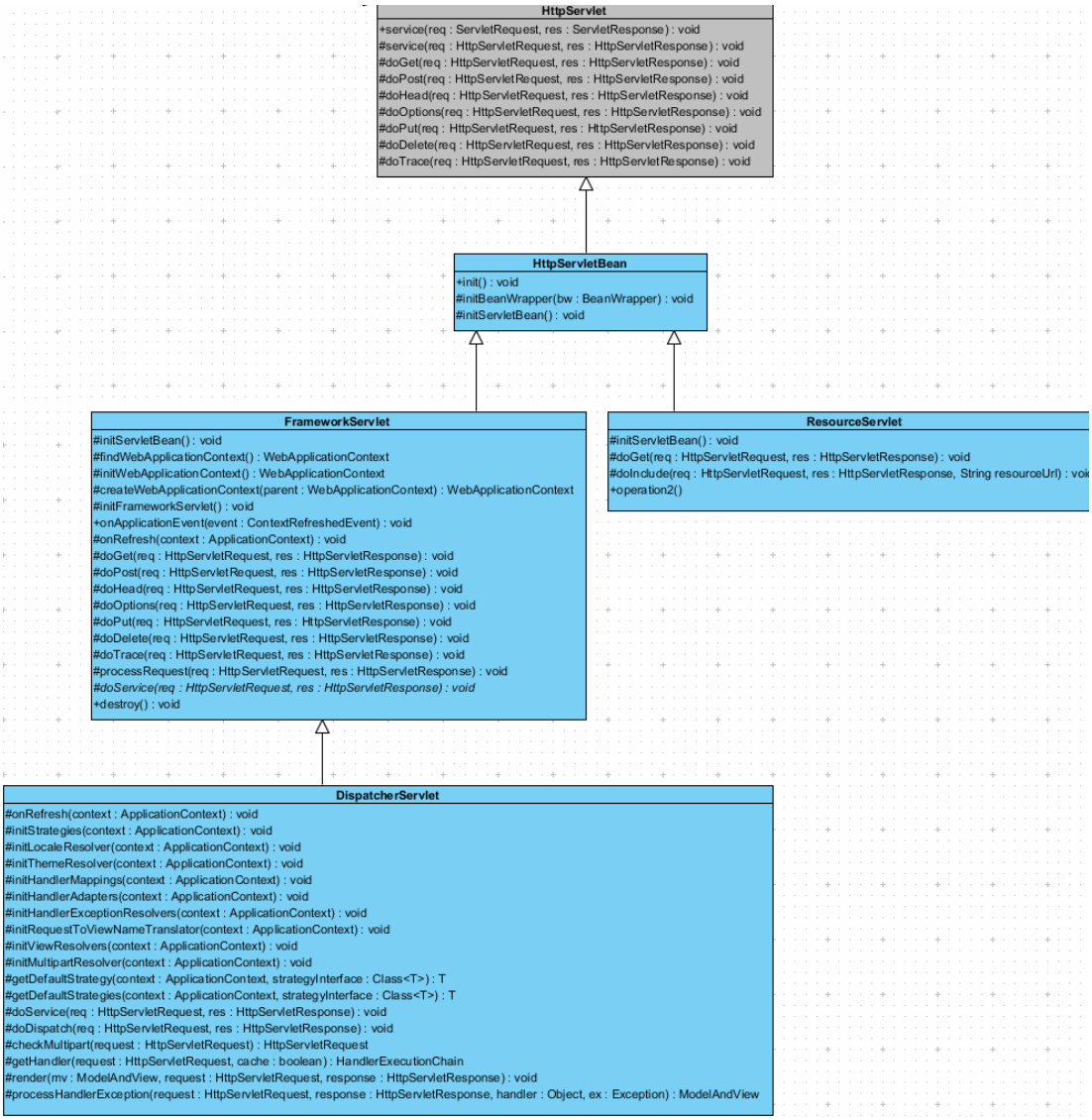
```
    //遍历所有的请求头信息
    while( reqHeaderEnum.hasMoreElements() ) {
        String headerName = (String)reqHeaderEnum.nextElement();

        //拼接所有的请求头到字符串中，并且使用：分割名值对，每对头信息之间使用回车换行进行分隔
        responseString += CRLF + headerName + ": " +
        req.getHeader(headerName);
    }

    //附着回车换行符到字符串结尾
    responseString += CRLF;

    //取得字符串字节长度信息
    responseLength = responseString.length();

    //设置响应类型为message/http
    resp.setContentType("message/http");

    //设置响应体的长度
    resp.setContentLength(responseLength);

    //输出字符串消息到响应中
    ServletOutputStream out = resp.getOutputStream();
    out.print(responseString);

    //关闭相应流，结束操作
    out.close();
    return;
}
```

从上面的代码注释中，我们可以看到 Servlet 规范中的 HTTP Servlet 的实现只是一个占位符实现，并不包含完全的服务实现，一些服务的实现是由子类 Servlet 完成的。Spring Web MVC 就是通过实现这些占位符方法来派遣 HTTP 请求到 Spring Web MVC 的控制器组件方法的。

下面一节我们将深入剖析 Spring Web MVC 的控制器是如何进行派遣和处理 HTTP 请求的。

4.1.2 派遣器Servlet及其父类

上节我们介绍了 Servlet 容易规范中的 HTTP Servlet 的实现，它对于各种 HTTP 方法使用了占位符的实现，这些占位符方法需要子类进一步重写，这一小节中我们将讨论，派遣器 Servet 是如何实现这些占位符方法来完成 Spring Web MVC 的工作流的。下图是派遣器 Servlet 完整的实现体系，



图表 4-4

从上面实现类图可以看到，继承自 HTTP Servlet 的直接子类就是 Http Servlet Bean。这个类的唯一功能就是把 Servlet 配置的参数作为一个 Bean 的属性对 Servlet 的属性字段进行自动的初始化。(Spring 的派遣器 Servlet 会默认加载一个子环境，这个子环境的位置可以用 Spring 的初始化参数指定，就是这个功能实现的。) 这些属性需要有 getter 和 setter 方法。

上面这个特点是通过重写通用 Servlet 的 init()方法提供实现的，如下代码注释，

```
//重写通用Servlet的init()方法占位符进行初始化Servlet Bean，这些初始化信息来自于Serlvet配置的参数，
我们通常通过Servlet初始化参数给一个Serlvet指定一个非默认名字的Spring Context的文件路径，就是这里实
现的
@Override
public final void init() throws ServletException {
    if (logger.isDebugEnabled()) {
        //这个Servlet名字就是从Servlet配置对象中取得的，而Serlvet配置是在通用Servlet初始化阶段保存的
        logger.debug("Initializing servlet '" + getServletName() + "'");
```

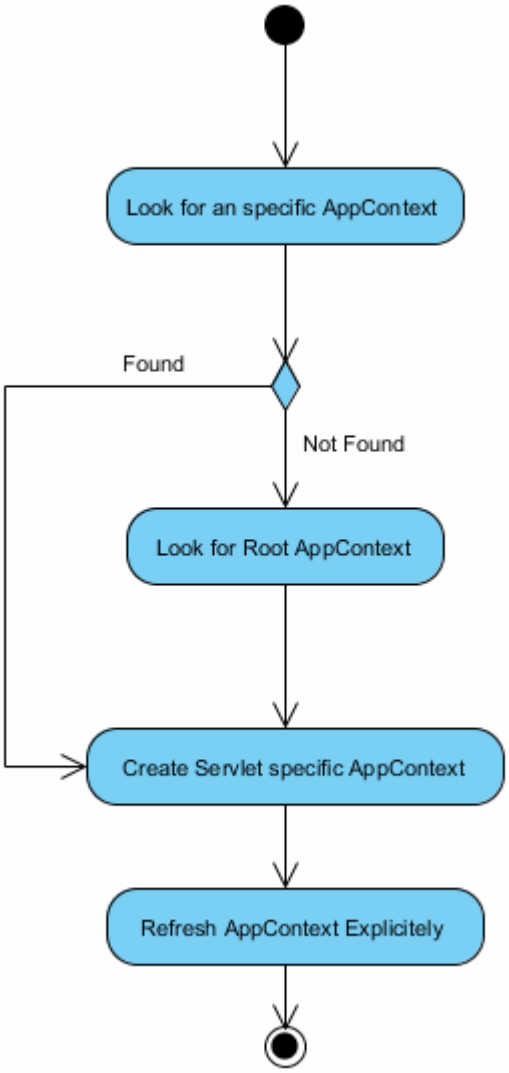
```
    }

    //设置Servlet初始化参数作为Servlet Bean的属性
    try {

        //使用Servlet配置的初始化参数创建一个PropertyValues对象,PropertyValues对象是名值对的
        集合, 子类也可以指定哪些属性是必须的
        PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(),
this.requiredProperties);

        //把当前的Servlet当作一个Bean, 把Bean的属性以及属性的存取方法信息放入BeanWrapper对象
        BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);

        //注册一个可以在资源和路径之间进行转化的客户化编辑器，这些资源是这个Web应用的内部资源，例如,
        一个文件，一个图片等等
        ResourceLoader resourceLoader = new
ServletContextResourceLoader(getServletContext());
        bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader));

        //提供给子类机会增加更多的客户化的编辑器，或者对BeanWrapper进行更多的初始化
        initBeanWrapper(bw);

        //把初始化制定的参数值赋值到Servlet的属性中，第二个参数true表明忽略位置属性
        bw.setPropertyValues(pvs, true);
    }
    catch (BeansException ex) {
        logger.error("Failed to set bean properties on servlet '" + getServletName() + "'",
ex);
        throw ex;
    }

    //给子类一个机会去初始化子类需要的资源，同样是一个占位符方法
    initServletBean();

    if (logger.isDebugEnabled()) {
        logger.debug("Servlet '" + getServletName() + "' configured successfully");
    }
}
```

从上图可以看出，HTTP Servlet Bean初始化自己特殊的资源以后，留下了另外一个占位符方法initServletBean(),这个方法提供子类初始化的机会。

在这个类体系结构中的下一个实现类是框架Servlet，框架Servlet提供的主要功能就是加载一个Web应用程序环境，这是通过实现父类的占位符方法initServletBean()实现的。并且重

写HTTP Servlet中的占位符方法，派遣HTTP请求到统一的Spring Web MVC的控制器方法，进而派遣器Servlet派遣这个HTTP请求到不同的处理器进行处理和响应。

首先，我们分析框架Servlet是如何加载Web应用程序环境的，如下图流程所示，



图表 4-5

从上图可以看出，框架Servlet试图去查找一个专用的根环境，但是，如果这个专用的根环境不存在，这个Servlet则会查找共享的根环境，使用共享的根环境是我们常见的一种配置。如下程序注解，

```
//重写了Http Servlet Bean的初始化占位符方法initServletBean()，进而初始化框架Serlvet所需的资源，
在这里就是Web应用程序环境
@Override
protected final void initServletBean() throws ServletException {
```

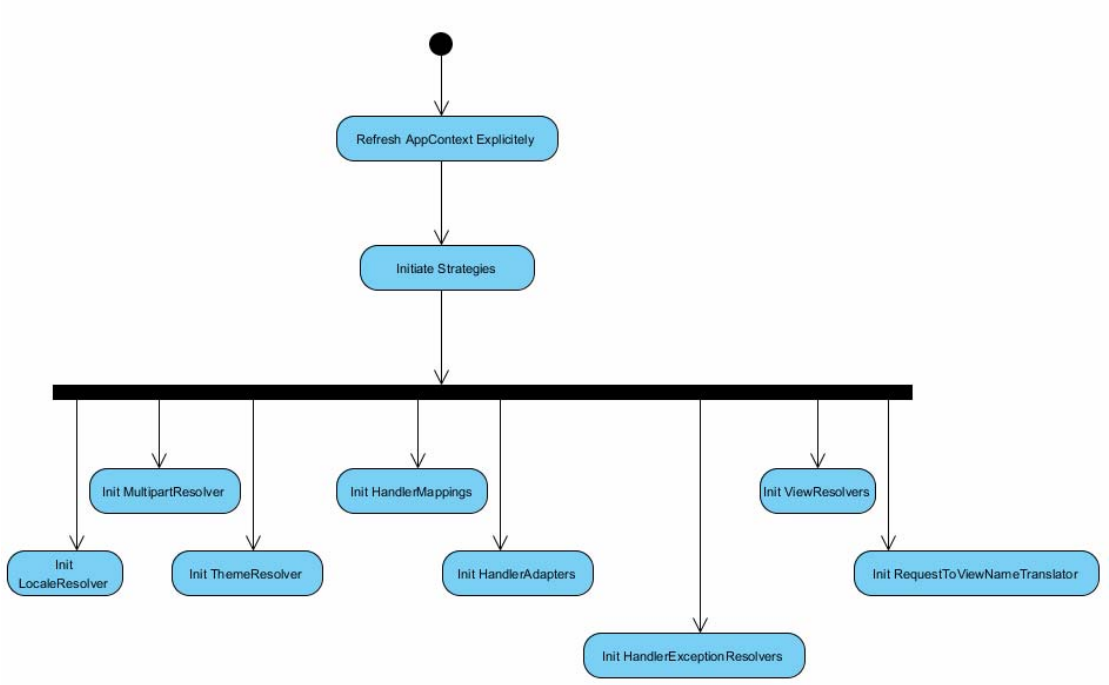
```
        //打印初始化信息到Servlet容易的日志
        getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName()
+ "'");

        if (this.logger.isInfoEnabled()) {
            this.logger.info("FrameworkServlet '" + getServletName() + "': initialization
started");
        }

        //取得初始化环境的开始时间
        long startTime = System.currentTimeMillis();

        try {
            //初始化Servlet的环境，这个方法的流程见上图
            this.webApplicationContext = initWebApplicationContext();

            //同样调用一个占位符方法，这个占位符方法给子类机会去初始化子类指定的资源，这是这个方法在派遣
器Servlet中并没有覆盖
            initFrameworkServlet();
        }
        catch (ServletException ex) {
            this.logger.error("Context initialization failed", ex);
            throw ex;
        }
        catch (RuntimeException ex) {
            this.logger.error("Context initialization failed", ex);
            throw ex;
        }

        if (this.logger.isInfoEnabled()) {
            //取得初始化环境的结束时间
            long elapsedTime = System.currentTimeMillis() - startTime;

            //log初始化Web应用程序环境所需的总体时间
            this.logger.info("FrameworkServlet '" + getServletName() + "': initialization
completed in " +
                    elapsedTime + " ms");
        }
    }

protected WebApplicationContext initWebApplicationContext() {
    //首先查找是否这个派遣器Servlet有一个专用的根环境，这个根环境是通过一个属性（contextAttribute）
作为关键字存储在Servlet环境里的，这个属性可以在Servlet的初始化参数中指定，因为在HTTP Servlet Bean
的初始化过程中，初始化参数将被当作为Bean属性进行赋值
```

```
    WebApplicationContext wac = findWebApplicationContext();

    //如果这个Servlet不存在专用的根环境
    //通常我们不需要这个环境，因为我们通常使用一个Web监听器进行加载一个默认的共享的根环境
    if (wac == null) {
        //取得默认的共享的根环境，这个根环境通过关键字ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE保存
在Servlet环境对象里
        WebApplicationContext parent =
                WebApplicationContextUtils.getWebApplicationContext(getServletContext());

        //创建派遣器Servlet的子环境，这个子环境引用得到的主环境，这个猪环境是可选的
        wac = createWebApplicationContext(parent);
    }

    //Web应用程序环境在创建之后，指定了这个类作为Web应用程序环境事件处理的监听器，如果这个Web应用程序
环境支持刷新，这个onRefresh方法应该已经调用，否则，我们需要手工激发初始化事件
    //这个刷新方法将被派遣器Servlet重写，它将提取并初始化Spring Web MVC的各个组件。

    //框架Servlet初始化为子类准备了初始化占位符方法initFrameworkServlet()，但是，同时也准备了
onRefresh()方法，因为一个ConfigurableApplicationContext是支持动态刷新的，依赖于Web应用程序环境的
子类组件应该监听这个方法进行重新初始化，派遣器Servlet就是这样实现的
    if (!this.refreshEventReceived) {
        onRefresh(wac);
    }

    //如果设置了发布环境属性，则把这个Web应用程序环境以ServletContextAttributeName的值作为关键字
保存到Servlet环境对象里，这个关键字是
org.springframework.web.servlet.FrameworkServlet.CONTEXT. + Servlet名称
    //这样做可以在其他的Servlet中共享这个Web应用程序环境
    if (this.publishContext) {
        // Publish the context as a servlet context attribute.
        String attrName = getServletContextAttributeName();
        getServletContext().setAttribute(attrName, wac);
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Published WebApplicationContext of servlet '" +
getServletName() +
                    "' as ServletContext attribute with name [" + attrName + "]");
        }
    }

    return wac;
}
```

我们可以看出框架 Servlet 初始化后，定义了初始化占位符方法 initFrameworkServlet()，子类

可以实现这个占位符方法进行初始化，然而，它还定义了另外的初始化的占位符方法 onRefresh(), 这个方法是在 Web 应用程序环境创建时或者刷新时调用的。派遣器 Servlet 通过重写这个占位符方法进行查找或者初始化 Spring Web MVC 所需要的各种组件。使用 onRefresh()初始化的好处就是可以使 Spring Web MVC 的组件可以动态的重新加载。

下面是框架 Serlvet 的初始化全过程，如下图所示，



图表 4-6

派遣器 Servlet 通过监听事件得知 Servlet 的 Web 应用程序环境初始化或者刷新后，首先在加载的 Web 应用程序环境(包括主环境和子环境)中查找是不是已经注册了相应的组件，如果查找到注册的组件，就会使用这些组件。如果没有查找到已经注册的相应的组件，派遣器 Servlet 将会加载缺省的配置策略，这些缺省的配置策略保存在一个属性文件里，这个属性文件和派遣器 Servlet 在同一个目录里，文件名是 DispatcherServlet.properties，通过读取不同组件配置的实现类名，实例化并且初始化这些组件的实现。

Spring Web MVC 的组件通过数量分为可选组件，单值组件和多值组件。

- 可选组件是在整个流程里可能需要也可能不需要。例如，MultipartResolver。
- 单值组件是在整个流程里只需要一个这样的组件。例如，ThemeResolver, LocaleResolver 和 RequestToViewNameTranslator。
- 多值组件是在整个流程里可以配置多个这样的实现组件。运行时，轮询查找哪个组件支持当前的 HTTP 请求，如果找到一个支持的组件则使用这个组件进行处理。

以下方法 initStrategies()是在 Web 应用程序环境初始化或者刷新的时候调用的，如上图所示，这个方法加载了所有的 Spring Web MVC 所需要的组件。如下代码注释，

```
protected void initStrategies(ApplicationContext context) {
    //初始化多部请求解析器，没有默认的实现
    initMultipartResolver(context);

    //初始化地域解析器，默认实现是AcceptHeaderLocaleResolver
    initLocaleResolver(context);

    //初始化主题解析器，默认实现是FixedThemeResolver
    initThemeResolver(context);

    //初始化处理器映射，这是个集合， 默认实现是BeanNameUrlHandlerMapping和
DefaultAnnotationHandlerMapping
    initHandlerMappings(context);

    //初始化处理器适配器，这是个集合，默认实现是HttpRequestHandlerAdapter，
SimpleControllerHandlerAdapter和AnnotationMethodHandlerAdapter
    initHandlerAdapters(context);

    //初始化处理器异常解析器，这是个集合，默认实现是
AnnotationMethodHandlerExceptionResolver,ResponseStatusExceptionResolver和
DefaultHandlerExceptionResolver
    initHandlerExceptionResolvers(context);

    //初始化请求到视图名解析器，默认实现是DefaultRequestToViewNameTranslator
    initRequestToViewNameTranslator(context);

    //初始化视图解析器，这是个集合，默认实现是InternalResourceViewResolver
    initViewResolvers(context);
}
```

对于可选组件，请看如下代码注释，

```
private void initMultipartResolver(ApplicationContext context) {
    try {
        //从配置的Web应用程序环境中查找多部请求解析器
        this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME,
MultipartResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using MultipartResolver [" + this.multipartResolver + "]");
        }
    }
    catch (NoSuchBeanDefinitionException ex) {
        this.multipartResolver = null;
```

```
        //如果没有多部请求解析器在Web应用程序环境中注册，忽略这种情况，毕竟不是所有的应用程序都需要使用
它，多部请求通常会应用到文件上传的情况
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate MultipartResolver with name '" +
MULTIPART_RESOLVER_BEAN_NAME +
                    "': no multipart request handling provided");
        }
    }
}
```

对于单值组件，请看如下代码注释，

```
private void initLocaleResolver(ApplicationContext context) {
    try {
        //从配置的Web应用程序环境中查找地域请求解析器
        this.localeResolver = context.getBean(LOCALE_RESOLVER_BEAN_NAME,
LocaleResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using LocaleResolver [" + this.localeResolver + "]");
        }
    }
    catch (NoSuchBeanDefinitionException ex) {
        //如果没有地域请求解析器在Web应用程序环境中注册，则查找缺省配置的策略，并且根据配置初始化缺省的
地域请求解析器，后面将代码注解如何加载默认策略
        this.localeResolver = getDefaultStrategy(context, LocaleResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate LocaleResolver with name '" +
LOCALE_RESOLVER_BEAN_NAME +
                    "': using default [" + this.localeResolver + "]");
        }
    }
}
```

initThemeResolver()和initRequestToViewNameTranslator()同样是对单值组件进行初始化，他们的实现和initLocaleResolver()具有相同的实现，这里将不进行代码注解。

下面是对多值组件进行初始化的代码注释，

```
private void initHandlerMappings(ApplicationContext context) {
    this.handlerMappings = null;

    if (this.detectAllHandlerMappings) {
        //如果配置成为自动检测所有的处理器映射,则在加载的Web应用程序环境中查找所有实现HandlerMapping
接口的Bean
        Map<String, HandlerMapping> matchingBeans =
```

```
            BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
HandlerMapping.class, true, false);
        if (!matchingBeans.isEmpty()) {
            this.handlerMappings = new ArrayList<HandlerMapping>(matchingBeans.values());

            // 根据这些Bean所实现的Order接口进行排序
            OrderComparator.sort(this.handlerMappings);
        }
    }
    else {
        //如果没有配置成为自动检测所有的处理器映射，则在Web应用程序环境中查找指定名字
为"handlerMapping"的Bean作为处理器映射
        try {
            HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME,
HandlerMapping.class);

            //构造单个Bean的集合
            this.handlerMappings = Collections.singletonList(hm);
        }
        catch (NoSuchBeanDefinitionException ex) {
            //忽略异常，后面将使用对象引用是否为空判断是否查找成功
        }
    }

    if (this.handlerMappings == null) {
        //如果仍然没有查找到注册的HanlderMapping的实现，则使用缺省的配置策略加载处理器映射
        this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);
        if (logger.isDebugEnabled()) {
            logger.debug("No HandlerMappings found in servlet '" + getServletName() + "':
using default");
        }
    }
}
```

initHandlerAdapters(), initHandlerExceptionResolvers()和 initViewResolvers ()同样是对多值组件进行初始化，他们和 initHandlerMappings()具有相同的实现，这里将不进行代码注解。

下面的两个代码注释阐述了，缺省的配置策略是如何进行加载的。

```
protected <T> T getDefaultStrategy(ApplicationContext context, Class<T> strategyInterface)
{
    //对于单值的组件加载，首先重用了多值的组件的加载方法，然后判断是否只有一个组件配置返回，期待的结果是
有且只有一个组件被配置，否则将抛出异常
    List<T> strategies = getDefaultStrategies(context, strategyInterface);
```

```
    if (strategies.size() != 1) {
        throw new BeanInitializationException(
                "DispatcherServlet needs exactly 1 strategy for interface [" +
strategyInterface.getName() + "]");
    }
    return strategies.get(0);
}


protected <T> List<T> getDefaultStrategies(ApplicationContext context, Class<T>
strategyInterface) {
    //取得组件接口的完全的类名，缺省策略是通过组件接口的类名作为关键字存储在属性文件里的
    String key = strategyInterface.getName();

    //取得以这个接口名为关键字配置的所有实现类的名字，这些类的名字是通过逗号分隔的
    String value = defaultStrategies.getProperty(key);
    if (value != null) {
        //把逗号分隔的类名字符串转化成字符串的数组
        String[] classNames = StringUtils.commaDelimitedListToStringArray(value);

        //对每一个配置的类名字，加载这个类并且初始化这个组件
        List<T> strategies = new ArrayList<T>(classNames.length);
        for (String className : classNames) {
            try {
                //通过类的名字加载这个类
                Class clazz = ClassUtils.forName(className,
DispatcherServlet.class.getClassLoader());

                //在Web应用程序环境中创建这个组件的类的对象
                Object strategy = createDefaultStrategy(context, clazz);

                //将初始化的组件Bean加入到返回的集合结果中
                strategies.add((T) strategy);
            }
            catch (ClassNotFoundException ex) {
                //如果找不到这个类的定义
                throw new BeanInitializationException(
                        "Could not find DispatcherServlet's default strategy class [" +
className +
                                "] for interface [" + key + "]", ex);
            }
            catch (LinkageError err) {
                //如果找不到加载的类的依赖类
                throw new BeanInitializationException(
```

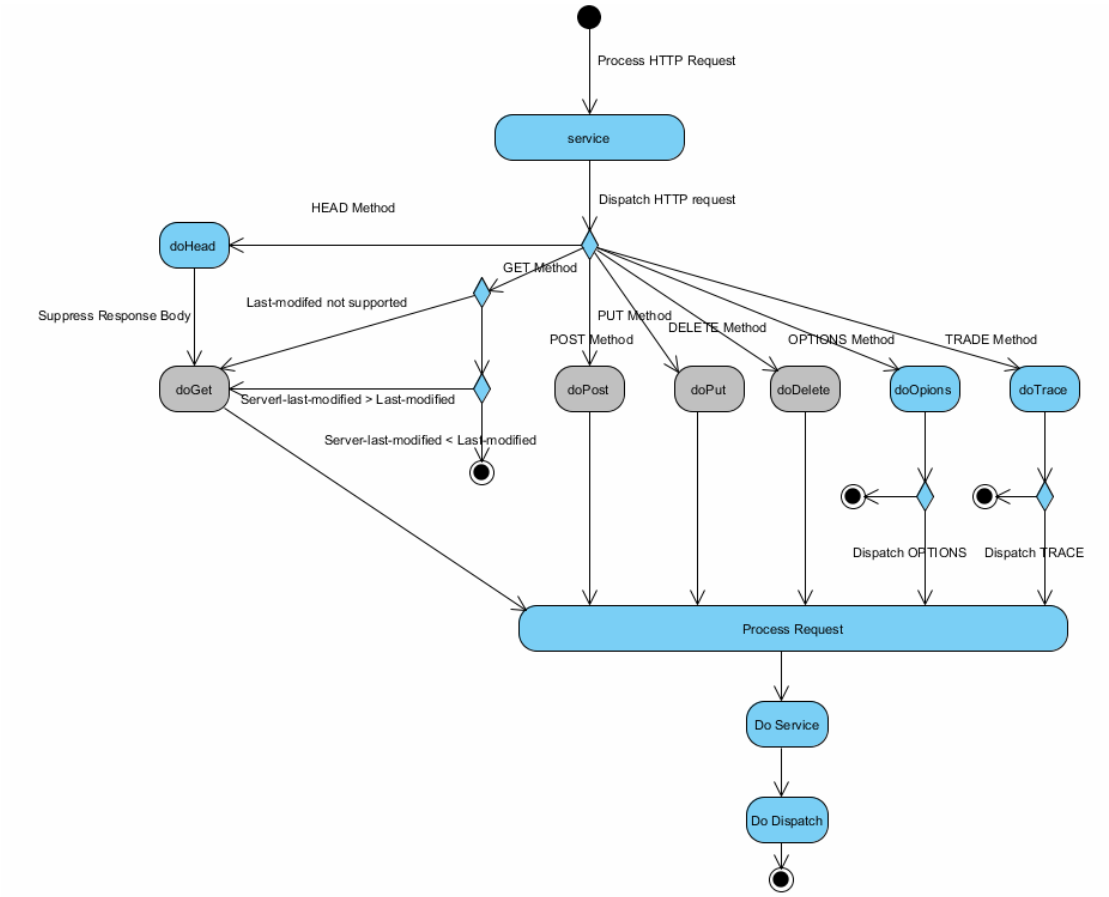
```
                    "Error loading DispatcherServlet's default strategy class [" +
className +
                            "] for interface [" + key + "]: problem with class file or
dependent class", err);
                }
            }
            return strategies;
        }
        else {
            //如果没有缺省策略配置，则返回一个空列表
            return new LinkedList<T>();
        }
    }
```

上面我们讨论了派遣器 Servlet 以及父类是如何进行初始化的。从 HTTP Servlet Bean,框架 Servlet 到派遣器 Servlet 逐层的初始化，每个层次的初始化完成一个特定的功能。当派遣器 Servlet 初始化完毕后，所有的 Spring Web MVC 的组件都初始化完毕并且准备进行对 HTTP 请求进行服务。

接下来，如果一个 HTTP 请求被派遣到派遣器 Servlet,那么派遣器 Servlet 就开始了真正的 Spring Web MVC 的工作流，这是一个复杂而又清晰的流程。下面我们将深入分析这个流程。

在上一节中，我们通过研究 Servlet 规范中的 HTTP Servlet 的实现得知，HTTP Servlet 已经根据 HTTP 请求所指定的 HTTP 方法将不同的 HTTP 请求分发到不同的占位符方法去处理，这些占位符方法是，doGet(), doPost(), doPut(), doDelete()。doHead()方法是通过 doGet()方法实现，子类通常是不需要改写的。doOptions()和 doTrace()方法的实现基本是不变的，子类通常条件下也不需要改写。

在本书开始章节里介绍了 Web MVC 模型的组成，我们知道控制器层是 Web MVC 中不可缺少的一部分，所以，在 Spring Web MVC 的实现中，改写了这些占位符方法，把 HTTP 请求重新统一的派遣分发到派遣器 Servlet 的控制器方法，由框架 Servlet 中的 handleRequest()方法统一进行处理，如下图所示，



图表 4-7

在框架 Servlet 中可以配置是否将 OPTIONS 和 TRACE 方法派遣到 Spring Web MVC 的控制流中，通常的 Spring Web MVC 是不需要派遣和重新实现这两个操作的行为的。请看 doGet() 的代码注释，

```
//这个方法被定义为final,HTTP GET请求应该派往到Spring Web MVC流程去处理，子类不应该改写这个方法
protected final void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
   //简单的派遣HTTP GET请求到Spring Web MVC的控制流
   processRequest(request, response);
}
```

doPost(), doPut(), doDelete()具有相同的实现。也就是说，doGet(), doPost, doPut, doDelete()把HTTP的 GET, POST, PUT和DELETE请求统一的分发到Spring Web MVC的控制器方法进行处理。这里不再做代码注释。

下面是doOptions()的代码注释。

```
//Spring Web MVC并不需要对OPTIONS请求进行任何特殊的处理，所以，这个方法是可以被子类改写的，也就是说，
改写后不会对Spring Web MVC流程有任何的影响
protected void doOptions(HttpServletRequest request, HttpServletResponse response)
```

```
        throws ServletException, IOException {
    //调用HTTP Servlet中对OPTIONS方法的默认实现
    super.doOptions(request, response);

    //通过配置可以决定是否将OPTIONS和TRACE方法派遣到Spring Web MVC的控制流中，通常是不需要的
    if (this.dispatchOptionsRequest) {
        processRequest(request, response);
    }
}
```

doTrace()的实现和doOptions()的实现是相似的，这里不再做代码注释。

流程走到这里，我们看到所有的HTTP GET, POST, PUT, DELETE甚至OPTIONS和TRACE请求都被统一传递到processRequest()方法进行统一的处理，下面的代码注释了这些请求是如何进一步被Spring Web MVC处理的。

```
//服务前保存线程局部存储的信息，服务后回复这些信息
protected final void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
    //记录处理请求的开始时间
    long startTime = System.currentTimeMillis();

    //记录产生的异常在finally语句里打印出来
    Throwable failureCause = null;

    //因为一个线程可能处理不同的请求,这经常发生在forward,include的操作上，所以在处理之前需要保存一些
容易被覆盖的信息，请求结束后进行恢复

    //保存当前线程局部存储的地域信息，以备处理完这个请求后进行恢复
    LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();

    //导出当前的请求的地域信息到线程的局部存储，所以尽管你不能得到HTTP Request对象， 在Spring Web MVC
流程中的任意位置都可以取得这个地域信息的值
    LocaleContextHolder.setLocaleContext(buildLocaleContext(request),
this.threadContextInheritable);

    //保存当前线程局部存储的请求属性，以备处理完这个请求后进行恢复
    RequestAttributes previousRequestAttributes =
RequestContextHolder.getRequestAttributes();
    ServletRequestAttributes requestAttributes = null;

    //如果当前线程局部存储不包含请求属性，或者包含着同样的ServletRequestAttributes的实例，则导出新
的请求属性，这个新的请求环境能够连接到request以及session对象的
```

```
    //如果这个Servlet是第一个组件处理这个请求，previousRequestAttributes一定为空
    //对于包含请求的情况，previousRequestAttributes并不是空，而且是类ServletRequestAttributes
的一个实例，因为在前一个Servlet组件的处理中建立了一个ServletRequestAttributes的实例， 所以需要创建
一个新的请求属性并且保存在环境中，这个新的请求属性是供这个Servlet使用，并且需要保存前一个请求属性的
    if (previousRequestAttributes == null ||
previousRequestAttributes.getClass().equals(ServletRequestAttributes.class)) {
        //基于当前的请求对象创建一个请求属性对象
        requestAttributes = new ServletRequestAttributes(request);

        //把新创建的请求属性对象放入线程的局部存储中
        RequestContextHolder.setRequestAttributes(requestAttributes,
this.threadContextInheritable);
    }

    if (logger.isTraceEnabled()) {
        logger.trace("Bound request context to thread: " + request);
    }

    try {
        //开始Spring Web MVC的真正的派遣工作流，这个方法在框架Servlet中定义为抽象方法，在派遣器
Servlet中实现
        doService(request, response);
    }
    catch (ServletException ex) {
        //保存异常在finally语句里打印log
        failureCause = ex;
        throw ex;
    }
    catch (IOException ex) {
        //保存异常在finally语句里打印log
        failureCause = ex;
        throw ex;
    }
    catch (Throwable ex) {
        //保存异常在finally语句里打印log
        failureCause = ex;
        throw new NestedServletException("Request processing failed", ex);
    }

    finally {
        //请求处理完，恢复先前的线程局部存储中的地域信息
        LocaleContextHolder.setLocaleContext(previousLocaleContext,
this.threadContextInheritable);
```

```
        //如果在处理HTTP请求前导出了新的请求属性，这里恢复原来的线程局部存储的请求属性
        if (requestAttributes != null) {
            RequestContextHolder.setRequestAttributes(previousRequestAttributes,
this.threadContextInheritable);
            requestAttributes.requestCompleted();
        }
        if (logger.isTraceEnabled()) {
            logger.trace("Cleared thread-bound request context: " + request);
        }

        if (failureCause != null) {
            //异常发生时保存异常，在finally语句里打印出来
            this.logger.debug("Could not complete request", failureCause);
        }
        else {
            this.logger.debug("Successfully completed request");
        }

        if (this.publishEvents) {
            // 计算这个请求的总处理时间，将时间传递给应用程序环境，注册事件监听器的Bean就会接收到这个事
件，可以用作为统计分析
            long processingTime = System.currentTimeMillis() - startTime;
            this.webApplicationContext.publishEvent(
                    new ServletRequestHandledEvent(this,
                            request.getRequestURI(), request.getRemoteAddr(),
                            request.getMethod(), getServletConfig().getServletName(),
                            WebUtils.getSessionId(request), getUsernameForRequest(request),
                            processingTime, failureCause));
        }
    }
}
```

框架 Servlet 准备好请求环境，并且把请求以及请求属性保存在线程局部存储后，则把控制流传递给派遣器 Servlet，这样 Spring Web MVC 的任何一个角落都能访问到请求以及请求属性。

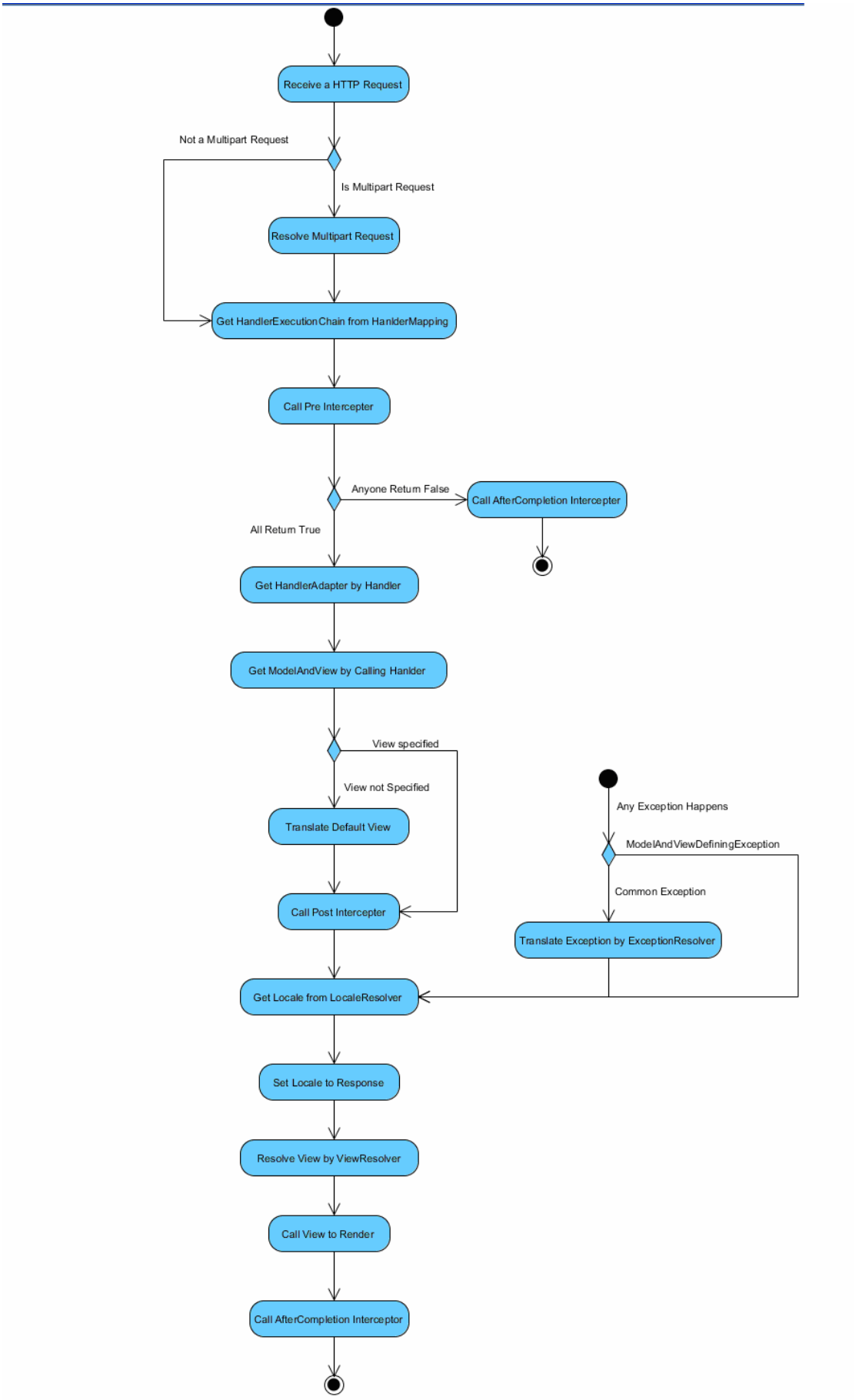
下面的代码注释用来分析派遣器 Servlet 如何派遣 HTTP 请求的。请注意，派遣器 Servlet 在派遣之前，保存了请求的属性信息，在服务过后恢复了这些信息。

```
protected void doService(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    if (logger.isDebugEnabled()) {
        String requestUri = new UrlPathHelper().getRequestUri(request);
```

```
        logger.debug("DispatcherServlet with name '" + getServletName() + "' processing "
+ request.getMethod() +
                " request for [" + requestUri + "]");
    }

    //对于一个include请求，除了需要保存和恢复请求环境信息，还需要保存请求属性，待请求处理完毕后，如果其
中某一个属性改变了，我们需要恢复这些属性
    Map<String, Object> attributesSnapshot = null;
    if (WebUtils.isIncludeRequest(request)) {

        //如果是一个包含请求，则遍历所有请求的属性
        logger.debug("Taking snapshot of request attributes before include");
        attributesSnapshot = new HashMap<String, Object>();
        Enumeration attrNames = request.getAttributeNames();
        while (attrNames.hasMoreElements()) {
            String attrName = (String) attrNames.nextElement();
            //如果包含请求清除属性打开，默认是打开的，则把所有spring指定的属性进行保存，这些属性的关键
字是以org.springframework.web.servlet开头的
            if (this.cleanupAfterInclude ||
attrName.startsWith("org.springframework.web.servlet")) {
                attributesSnapshot.put(attrName, request.getAttribute(attrName));
            }
        }
    }

    // 在包含请求的情况下，例如，一个URI /action/process1包含另外一个URI /action/process2,上面的
这些属性如果和主请求重复，包含请求结束后，这些属性的值将会被恢复

    // 存储Web应用程序环境在request属性里
    request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
getWebApplicationContext());

    // 存储地域解析器在request属性里
    request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);

    // 存储主题解析器在request属性里
    request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);

    // 存储主题源在request属性里
    request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());


    try {
        //开始Spring Web MVC的真正的派遣工作流
```

```
            doDispatch(request, response);
        }
        finally {
            if (attributesSnapshot != null) {
                //恢复保存的Spring指定的请求属性
                restoreAttributesAfterInclude(request, attributesSnapshot);
            }
        }
    }
```

程序执行到这里，Spring Web MVC 的工作流正式开始，这个工作流利用前面加载的各个 Spring Web MVC 的组件协调工作，开始派遣 HTTP 请求，处理 HTTP 请求，返回 HTTP 响应等等，具体步骤如下图，


Receive a HTTP Request
Not a Multipart Request
Is Multipart Request
Resolve Multipart Request
Get HandlerExecutionChain from HanlderMapping
Call Pre Intercepter
Anyone Return False
Call AfterCompletion Intercepter
All Return True
Get HandlerAdapter by Handler
Get ModelAndView by Calling Hanlder
View specified
View not Specified
Translate Default View
Call Post Intercepter
Any Exception Happens
ModelAndViewDefiningException
Common Exception
Translate Exception by ExceptionResolver
Get Locale from LocaleResolver
Set Locale to Response
Resolve View by ViewResolver
Call View to Render
Call AfterCompletion Interceptor

图表 4-8

这个活动图精确的阐述了，派遣器 Servlet 是如果通过初始化的 Spring Web MVC 的各个组件去处理一个 HTTP 请求的。我们可以看到，这个过程主要分为 2 个阶段完成的，

1. 通过映射处理器查找得到处理器对象，在通过支持的处理器对象调用得到的处理器，在调用处理器之前和之后都对应用在这个处理器的拦截器进行了调用。给客户化的处理器机会进行初始化或者析构资源。
2. 解析视图并且显示视图，发送 HTTP 响应。

上面的 2 个阶段描述了处理的整体流程，这些流程针对不同的技术有不同的实现。例如，在第 1 步处理器处理的实现上，有基于简单的 Spring 控制器的实现，也有基于注解的控制器的实现，还有用于实现远程 HTTP 调用的实现。对于视图解析和显示，也有不同的实现，例如，基于 JSP 页面的实现，基于 Tiles 的实现，基于报表的实现等等。我们将在后面的章节详细讨论这些实现的架构和流程。如下代码注释，

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    int interceptorIndex = -1;

    try {
        ModelAndView mv;
        boolean errorView = false;

        try {
            //如果是HTTP多部请求，则转换并且封装成一个简单的HTTP请求
            processedRequest = checkMultipart(request);

            // 根据处理器映射的配置，取得处理器执行链对象
            mappedHandler = getHandler(processedRequest, false);
            if (mappedHandler == null || mappedHandler.getHandler() == null) {

                //如果没有发现任何处理器，发送错误信息
                noHandlerFound(processedRequest, response);
                return;
            }

            // 开始调用前置拦截器
            HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();
            if (interceptors != null) {
                for (int i = 0; i < interceptors.length; i++) {
                    HandlerInterceptor interceptor = interceptors[i];
```

```
                //依次调用前置拦截器，如果任何一个拦截器返回false,则结束整个流程，反序掉用前面执行过的前置拦截器的所有完成拦截器，然后返回， 停止处理流程。
                if (!interceptor.preHandle(processedRequest, response, mappedHandler.getHandler())) {
                    triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, null);
                    return;
                }
                interceptorIndex = i;
            }
        }

        // 查找支持的处理器适配器
        HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

        // 通过得到的处理器适配器，调用得到的处理器，处理器适配器和处理器类型是成对出现的，因为处理器适配器知道如何调用它所支持的处理器
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

        if (mv != null && !mv.hasView()) {
            //如果控制器没有返回任何逻辑视图名，则用缺省的请求到视图名翻译器得到视图名
            mv.setViewName(getDefaultViewName(request));
        }

        // 应用后置拦截器
        if (interceptors != null) {
            for (int i = interceptors.length - 1; i >= 0; i--) {
                HandlerInterceptor interceptor = interceptors[i];
                //反序调用所有的后置拦截器，因为后置拦截器是用来释放资源的，如果初始化资源时用的是正序，那么起清除资源最好使用反序调用，这样就会很好的解决了资源依赖的问题
                interceptor.postHandle(processedRequest, response, mappedHandler.getHandler(), mv);
            }
        }
    }
    catch (ModelAndViewDefiningException ex) {
        //这个异常是用来跳过后续的处理过程，直接进入视图解析和显示阶段
        logger.debug("ModelAndViewDefiningException encountered", ex);
        mv = ex.getModelAndView();
    }
    catch (Exception ex) {
        //如果产生任何没有处理的异常，则调用处理器异常解析器，取得异常情况下的模型和视图对象，然后进入视图解析和显示阶段
        Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
```

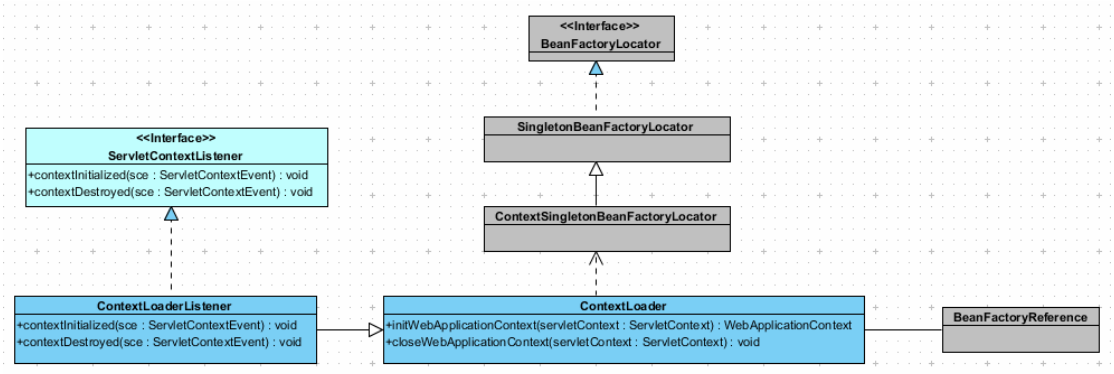
```
            mv = processHandlerException(processedRequest, response, handler, ex);
            errorView = (mv != null);
        }

        if (mv != null && !mv.wasCleared()) {
            //如果返回了一个视图对象，则解析视图和显示视图
            render(mv, processedRequest, response);
            if (errorView) {
                WebUtils.clearErrorRequestAttributes(request);
            }
        }
        else {
            //如果没有返回了一个视图对象，则不进行试图解析和显示
            if (logger.isDebugEnabled()) {
                logger.debug("Null ModelAndView returned to DispatcherServlet with name '"
+ getServletName() +
                        "': assuming HandlerAdapter completed request handling");
            }
        }

        //如果处理成功，调用完成拦截器
        triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest,
response, null);
    }

    catch (Exception ex) {
        //如果处理失败并且产生异常，调用完成拦截器，并且传入产生的异常，进行异常处理
        triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest,
response, ex);
        throw ex;
    }
    catch (Error err) {
        ServletException ex = new NestedServletException("Handler processing failed", err);
        //如果处理失败并且产生错误，调用完成拦截器，并且传入产生的错误，进行错误处理
        triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest,
response, ex);
        throw ex;
    }

    finally {
        //清除Multipart资源
        if (processedRequest != request) {
            cleanupMultipart(processedRequest);
        }
```

```
    }
}
```

4.1.3 根共享环境的加载

上一节中我们在分析框架 Sevlet 是如何初始化 Web 应用程序环境的时候得知，一个 Servlet 拥有一个专用的子环境，但是这个子环境可以而且通常引用一个根共享环境，这个根共享环境是通过 Servlet 环境监听器加载的。也就是说，当一个 Servlet 环境，也就是一个应用程序被容器加载时，监听器通过监听这个初始化事件初始化根共享 Web 应用程序，而当一个 Servlet 环境析构时，监听器功过监听这个析构时间析构共享的 Web 应用程序环境。

下面是整个根共享环境加载的类图，



图表 4-9

我们可以看到，类环境加载监听器类实现了 Servlet 规范中定义的 Servlet 环境监听器用以处理初始化事件和析构事件。而真正的根共享环境的创建的实现是环境加载类中实现的。在环境加载类中，通过 Servlet 初始化参数配置的根共享环境位置加载 Web 应用程序环境，并且将这个环境以 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE 为关键字保存在 Servlet 环境中，这个根共享环境在 Servlet 加载专用子环境中被引用作为父环境。

```
public void contextInitialized(ServletContextEvent event) {
    //这个方法实现的本意是提供一个占位符方法createContextLoader（）给子类机会创建客户化的环境加载，
但是，后来这个证明不是非常有用的，已经鼓励不再使用了，事实上，子类可以通过重写本方法达到同样的目的
    this.contextLoader = createContextLoader();

    //没有子类实现createContextLoader（）占位符 方法，则使用超类的缺省实现，超类 就是环境 加载类
    if (this.contextLoader == null) {
        //实际上是为了使用超类的默认实现
        this.contextLoader = this;
    }

    //调用超类的加载根共享Web应用程序环境的默认实现
```

```
        this.contextLoader.initWebApplicationContext(event.getServletContext());
    }

    public void contextDestroyed(ServletContextEvent event) {
        //如果环境加载存在，那么关闭环境加载的Web应用程序环境
        if (this.contextLoader != null) {
            this.contextLoader.closeWebApplicationContext(event.getServletContext());
        }

        //清除保存在Servlet环境中的任何可释放的Bean
        ContextCleanupListener.cleanupAttributes(event.getServletContext());
    }

    public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {
        //如果已经存在了根共享Web应用程序环境，则抛出异常提示客户
        if
    (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIB
    UTE) != null) {
            throw new IllegalStateException(
                    "Cannot initialize context because there is already a root application
    context present - " +
                    "check whether you have multiple ContextLoader* definitions in your
    web.xml!");
        }

        Log logger = LogFactory.getLog(ContextLoader.class);
        servletContext.log("Initializing Spring root WebApplicationContext");
        if (logger.isInfoEnabled()) {
            logger.info("Root WebApplicationContext: initialization started");
        }

        //记录创建根Web应用程序环境的开始时间
        long startTime = System.currentTimeMillis();

        try {
            //决定根Web应用程序环境是否存在父应用程序环境
            ApplicationContext parent = loadParentContext(servletContext);

            //创建根Web应用程序环境，如果父环境存在则引用父环境，通常情况下父环境是不存在的
            this.context = createWebApplicationContext(servletContext, parent);

            //把创建的根Web应用程序环境保存到Servlet环境中，每个派遣器Servlet加载的子环境会应用这个环
    境作为父环境
```

```
    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATT
RIBUTE, this.context);


        //取得线程的类加载器
        ClassLoader ccl = Thread.currentThread().getContextClassLoader();
        if (ccl == ContextLoader.class.getClassLoader()) {
            //如果线程和本类拥有相同的类加载器，则使用静态变量保存即可，因为同一类加载器加载同一份静
态变量
            currentContext = this.context;
        }
        else if (ccl != null) {
            //如果线程和本类拥有不同的类加载器，则使用线程的类加载器作为关键在保存在一个映射对象里，
保证析构时能拿到Web应用程序环境进行关闭操作
            currentContextPerThread.put(ccl, this.context);
        }

        if (logger.isDebugEnabled()) {
            logger.debug("Published root WebApplicationContext as ServletContext
attribute with name [" +
                    WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE +
"]");
        }
        if (logger.isInfoEnabled()) {
            long elapsedTime = System.currentTimeMillis() - startTime;
            logger.info("Root WebApplicationContext: initialization completed in " +
elapsedTime + " ms");
        }

        return this.context;
    }
    catch (RuntimeException ex) {
        logger.error("Context initialization failed", ex);
        //如果产生任何异常，则保存异常对象到Servlet环境里

    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATT
RIBUTE, ex);
        throw ex;
    }
    catch (Error err) {
        logger.error("Context initialization failed", err);
        //如果产生任何错误，则保存错误对象到Servlet环境里

    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATT
```

```
RIBUTE, err);
            throw err;
        }
    }

    protected WebApplicationContext createWebApplicationContext(ServletContext sc,
    ApplicationContext parent) {
        //取得配置的Web应用程序环境类，如果没有配置，则使用缺省的类XmlWebApplicationContext
        Class<?> contextClass = determineContextClass(sc);

        //如果配置的Web应用程序环境类不是可配置的Web应用程序环境的子类，则抛出异常，停止初始化
        if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
            throw new ApplicationContextException("Custom context class [" +
    contextClass.getName() +
                    "] is not of type [" + ConfigurableWebApplicationContext.class.getName()
    + "]");
        }

       //否则实例化Web应用程序环境类
        ConfigurableWebApplicationContext wac =
                (ConfigurableWebApplicationContext)
    BeanUtils.instantiateClass(contextClass);

        //设置Web应用程序环境的ID
        if (sc.getMajorVersion() == 2 && sc.getMinorVersion() < 5) {
            // 如果 Servlet规范 <= 2.4，则使用web.xml里定义的应用程序名字定义Web应用程序名
            String servletContextName = sc.getServletContextName();

            //设置ID
            wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX +
                    ObjectUtils.getDisplayString(servletContextName));
        }
        else {
            // 如果Servlet规范是 2.5， 则使用配置的ContextPath定义Web应用程序名
            try {
                String contextPath = (String)
    ServletContext.class.getMethod("getContextPath").invoke(sc);

                //设置ID
                wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX
    +
                        ObjectUtils.getDisplayString(contextPath));
            }
            catch (Exception ex) {
```

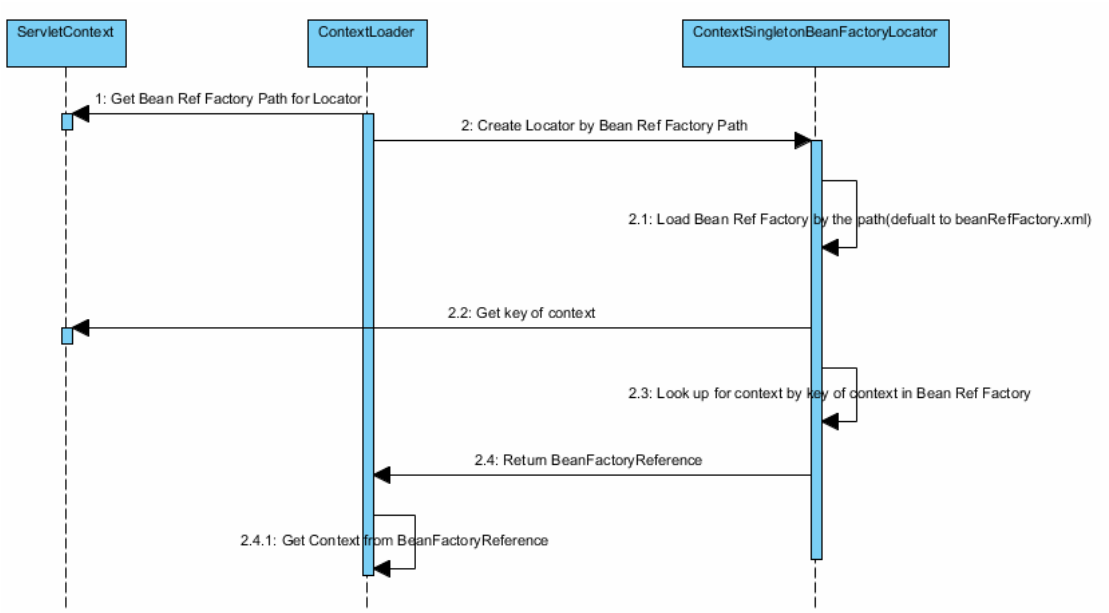
```
            //如果Servlet规范是2.5,但是不能取得ContextPath，抛出异常
            throw new IllegalStateException("Failed to invoke Servlet 2.5 getContextPath
method", ex);
        }
    }

    //如果父环境存在，则引用使用父环境
    wac.setParent(parent);

    //保存Servlet环境
    wac.setServletContext(sc);

    //设置环境的位置
    wac.setConfigLocation(sc.getInitParameter(CONFIG_LOCATION_PARAM));

    //提供子类可互换Web应用程序环境的机会
    customizeContext(sc, wac);

    //刷新Web应用程序环境以加载Bean定义
    wac.refresh();
    return wac;
}

protected Class<?> determineContextClass(ServletContext servletContext) {
    //首先检查是否初始化参数中定义了Web应用程序环境的类名
    String contextClassName = servletContext.getInitParameter(CONTEXT_CLASS_PARAM);
    if (contextClassName != null) {
        try {
            //如果初始化参数中定义了Web应用程序环境的类名，加载定义的类名
            return ClassUtils.forName(contextClassName,
ClassUtils.getDefaultClassLoader());
        }
        catch (ClassNotFoundException ex) {
            throw new ApplicationContextException(
                    "Failed to load custom context class [" + contextClassName + "]",
ex);
        }
    }
    else {
        //如果初始化参数中定义了Web应用程序环境的类名，加载缺省策略中定义的类名，缺省策略保存在
ContextLoader.properties文件里
        contextClassName =
defaultStrategies.getProperty(WebApplicationContext.class.getName());
        try {
```

```
            //加载缺省策略中定义的类名
            return ClassUtils.forName(contextClassName,
ContextLoader.class.getClassLoader());
        }
        catch (ClassNotFoundException ex) {
            throw new ApplicationContextException(
                    "Failed to load default context class [" + contextClassName + "]",
ex);
        }
    }
}

public void closeWebApplicationContext(ServletContext servletContext) {
    servletContext.log("Closing Spring root WebApplicationContext");
    try {
        //如果是可配置的Web应用程序环境
        if (this.context instanceof ConfigurableWebApplicationContext) {
            //关闭可配置的Web应用程序环境
            ((ConfigurableWebApplicationContext) this.context).close();
        }
    }
    finally {
        //取得当前线程的类加载器
        ClassLoader ccl = Thread.currentThread().getContextClassLoader();
        if (ccl == ContextLoader.class.getClassLoader()) {
            //如果当前线程和本类的公用一个类加载器，则清空静态变量引用
            currentContext = null;
        }
        else if (ccl != null) {
            //否则根据线程的类加载器移除保存的Web应用程序环境
            currentContextPerThread.remove(ccl);
        }

        //移除Servlet环境中的Web应用程序环境的引用

    servletContext.removeAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_
ATTRIBUTE);

        //如果父环境存在则释放父环境
        if (this.parentContextRef != null) {
            this.parentContextRef.release();
        }
    }
}
```

事实上，根共享环境的加载时同样可以加载一个父环境。尽管这种情况是不常见的，但是 Spring Web MVC 提供了这样的扩展性。在 Servlet 初始化参数中可以配置一个 Bean 工厂路径（locatorFactorySelector），这个 Bean 工厂路径会被 Bean 工厂定位器所加载，Bean 工厂定位器会在这个 Bean 工厂中查找以另外一个 Servlet 参数（parentContextKey）为名字的 Bean 工厂对象，最后得到的 Bean 工厂对象则是根共享环境的父环境。如果在初始化参数中没有配置 Bean 工厂路径，则用缺省的 Bean 工厂路径 classpath*:beanRefFactory.xml。

```
protected ApplicationContext loadParentContext(ServletContext servletContext) {
    ApplicationContext parentContext = null;

    //取得Web.xml初始化参数配置中对LOCATOR_FACTORY_SELECTOR_PARAM的配置串，这是Bean工厂定位器
使用的Bean工厂的路径，如果这个值没有配置，则使用缺省的classpath*:beanRefFactory.xml
    String locatorFactorySelector =
servletContext.getInitParameter(LOCATOR_FACTORY_SELECTOR_PARAM);

    //取得Web.xml初始化参数配置中对LOCATOR_FACTORY_KEY_PARAM的配置串，这是用来取得Bean工厂的关
键字
    String parentContextKey =
servletContext.getInitParameter(LOCATOR_FACTORY_KEY_PARAM);

    if (parentContextKey != null) {
        //locatorFactorySelector如果为空，则使用缺省值classpath*:beanRefFactory.xml初始化
Bean工厂定位器
        BeanFactoryLocator locator =
ContextSingletonBeanFactoryLocator.getInstance(locatorFactorySelector);
        Log logger = LogFactory.getLog(ContextLoader.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Getting parent context definition: using parent context key of
'" +
                    parentContextKey + "' with BeanFactoryLocator");
        }

        //Bean工厂定位器从配置的Bean工厂中找到制定关键字（参数LOCATOR_FACTORY_KEY_PARAM的值）的
工厂
        this.parentContextRef = locator.useBeanFactory(parentContextKey);

        //进而取得一个应用程序环境，这个应用程序环境作为根共享应用程序环境的父环境
        parentContext = (ApplicationContext) this.parentContextRef.getFactory();
    }

    return parentContext;
}
```

Bean 工厂定位器的实现中，加载了一个指定的 Bean 引用工厂，然后在加载的 Bean 引用工厂中查找指定名字的 Bean 工厂对象，这个 Bean 工厂对象会被返回，作为根共享环境的父环境。这些实现属于 Spring 环境项目范围，以下给出序列图，将不在这里做代码注释。



图表 4-10

根据上面的分析，我们发现 Spring Web MVC 是依赖于 Spring 环境的定义的，而每一个 Spring 环境可以最多有一个父环境的引用。这些特点同样应用到了 Spring Web MVC 的体系结构里。下面我们总结以下 Spring Web MVC 里面环境的三个层次。其中 Servlet 专用跟环境和根共享主环境在同一个层次。

Servlet专用子环境

加载组件：派遣器Servlet(框架Servlet)
配置路径：Servlet初始化参数contextConfigLocation指定的路径
缺省路径： WEB-INF/[servlet_name] -servlet.xml
保存位置：在框架Servlet对象内部，也以关键字*FrameworkServlet全类名.CONTEXT.Servlet名*保存在Servlet环境里

Servlet专用根环境

这是一个需要定制实现的组件，组件实现需要把加载的环境以某个关键字保存在 Servlet 环境里。

这样，如果在某个派遣器 Servlet 初始化参数 contextAttribute 指定这个关键字， Servlet 专用子环境会引用这个加载的专用根环境作为父环境。

根共享主环境

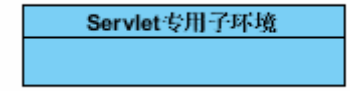
加载组件：环境加载监听器
配置路径：Servlet环境初始化参数contextConfigLocation指定的路径
缺省路径： 没有缺省路径
保存位置：WebApplicationContext全类名.ROOT

根共享环境主环境的父环境

加载组件：环境加载监听器和Bean工厂定位器
配置路径：Servlet环境初始化参数locatorFactorySelector指定Bean工厂定位器使用的给BeanFactory，Servlet环境初始化参数parentContextKey指定Bean工厂定位器用于查找BeanFactory的关键字
缺省路径： parentContextKey的缺省路径是classpath*:beanRefFactory.xml，如果parentContextKey没有制定，则超找所有ApplicationContext的子类实现
保存位置：WebApplicationContext全类名.ROOT

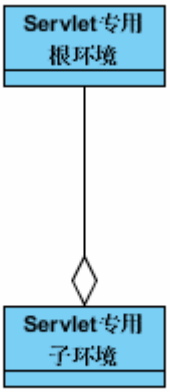
除了 Servlet 专用子环境，其他的父环境都是可选的。根据上面层次的组合，一共有 4 种环境配置，如下，

1. 单个 Servlet 专用子环境



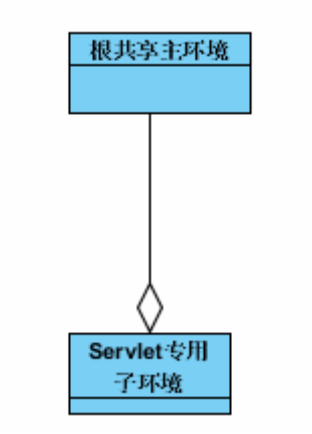
图表 4-11

2. Servlet 专用子环境引用到 Servlet 专用根环境



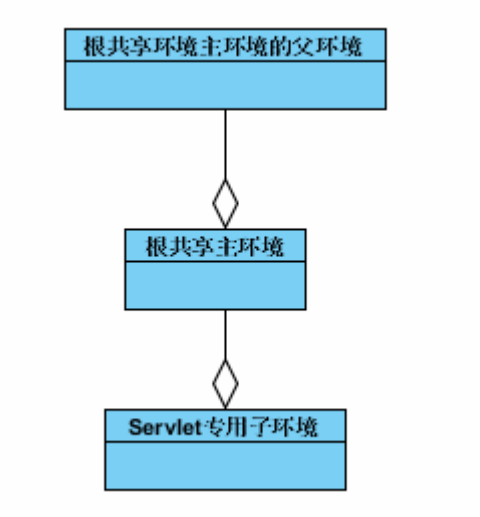
图表 4-12

3. Servlet 专用子环境引用到根共享主环境



图表 4-13

4. Servlet 专用子环境引用到根共享主环境及其父环境



图表 4-14

其中，配置 1 和配置 3 是我们在开发中经常使用到的。但是在业务逻辑更复杂的情况下，我们可以选择配置 2 和配置 4。配置 2 能够使多个 Servlet 共享一个根环境。配置 4 能使共享的跟环境通过一个 Serlvet 配置参数转换它的父环境。

4.1.4 其他Servlet

ResourceServlet 是用于存取 Web 应用程序的内部资源的。它也继承自 HttpServletBean，所以能够自动的将 Servlet 的初始化参数作为属性值来初始化 Servlet 对象。它改写了 HTTP Servlet 的 Get 方法来处理 HTTP 对资源的请求。

HttpRequestHandlerServlet 是用来直接将一个 HTTP 请求转发给 HttpRequestHandler。

ViewRenderSevlet 是用来与 Portlet 进行集成的 Sevlet。

4.2 处理器映射，处理器适配器以及处理器的实现

上一节，深入的剖析了作为总控制器的派遣器 Servlet 如何初始化 Web 应用程序环境，进而初始化 Spring Web MVC 所需要的各个组件。在一个 HTTP 请求从 Web 容器派发到派遣器 Servlet，派遣器 Servlet 进一步分发和派遣 HTTP 请求到 Spring Web MVC 的控制器组件的。

这一节，我们将分析 Spring Web MVC 的控制器组件如何传递和处理 HTTP 请求，并且返回模型数据和逻辑视图给总控制器派遣器 Servlet,。随后，派遣器 Servlet 将解析视图和显示视图，下一节将深入剖析解析视图和显示视图的实现。

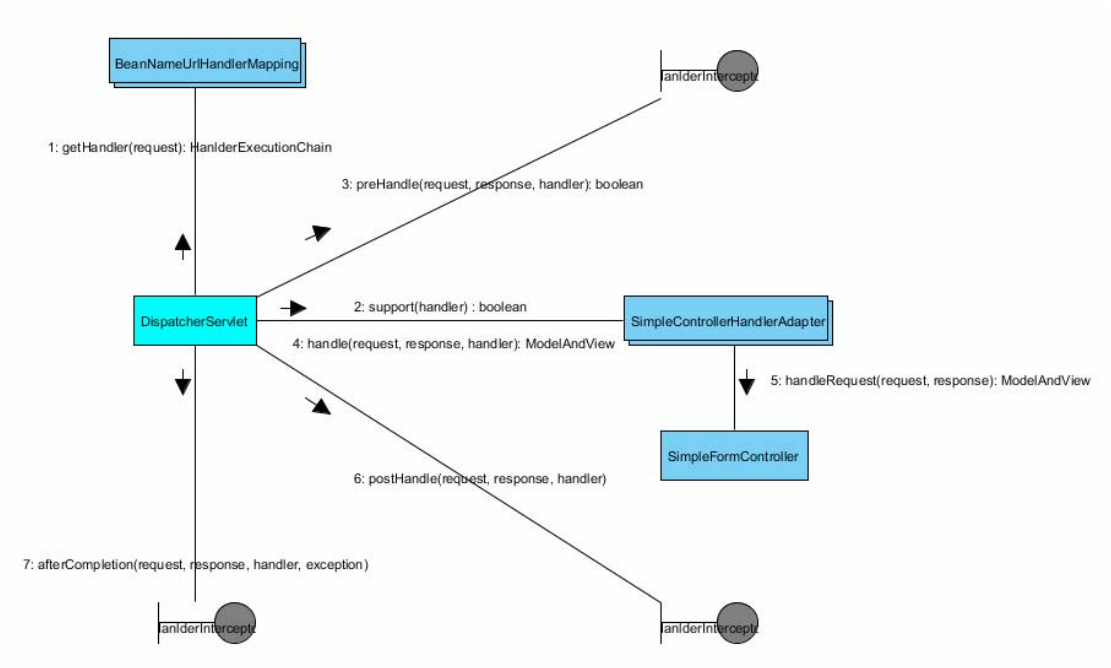
4.2.1 横向剖析

回顾第三章的概述中，Spring Web MVC 的控制器组件是由处理器映射，处理器适配器以及处理器三个组件协同工作完成的。为了能让 Spring Web MVC 设计结构清晰，易于扩展，能够轻松的与其他 Web MVC 实现相结合，Spring Web MVC 使用了通用的对象类型作为处理器，同时为一个类型的处理器提供一个专用处理器适配器进行请求传递，这样使在不改变任何 Spring Web MVC 架构或者实现的基础上插入新的控制器的实现成为可能。

这一小节将深入分析 Spring Web MVC 基于不同处理器类型的流程的实现。其中，最简单和实用的处理器类型是简单的控制器。

4.2.1.1基于简单控制器流程的实现

简单控制器的流程定义了简单的控制器接口(Controller)，在流程的开始，它通常通过 Bean 名字 URL 处理器映射(BeanNameUrlHandlerMapping)来获得支持此 HTTP 请求的一个控制器实例和支持这个控制器的处理器拦截器(HandlerInterceptor)。通过简单控制处理适配器(SimpleControllerHandlerAdapter)传递 HTTP 请求到简单的控制器（SimpleFormController）来实现 Spring Web MVC 的控制流程。这个流程如下图所示，

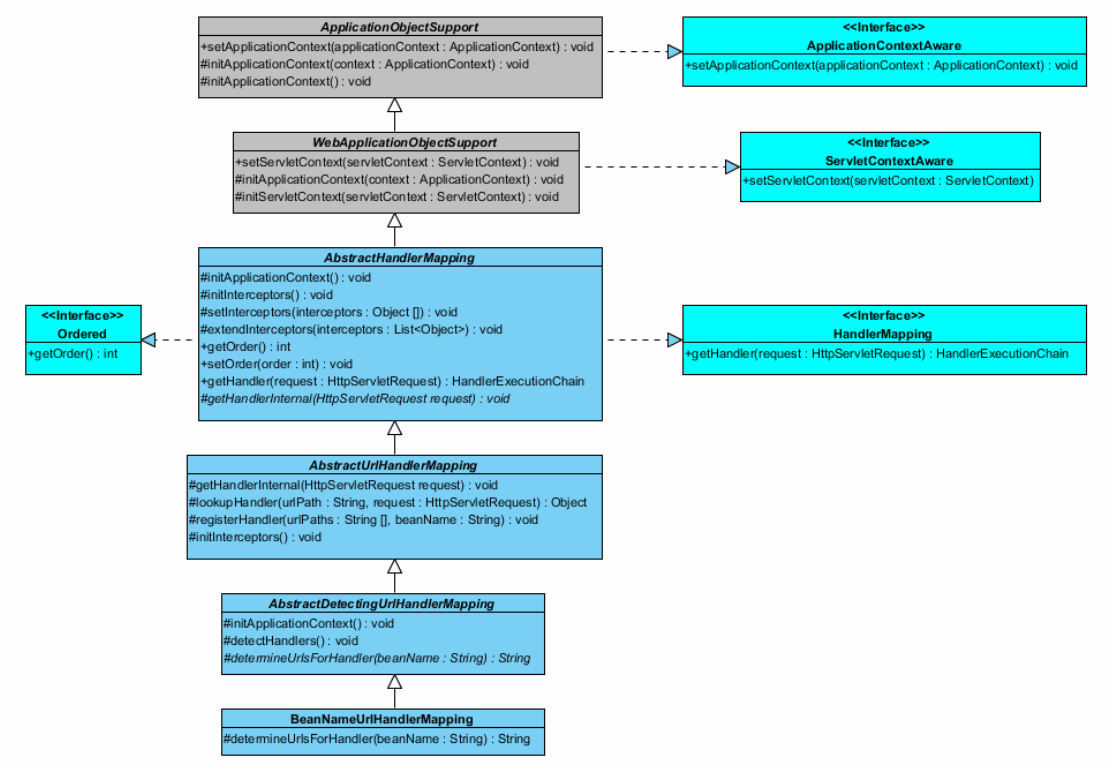


图表 4-15

首先，我们分析派遣器 Servlet 是如何通过 Bean 名字 URL 处理器映射获得处理器执行链，也就是上图中的第一步。

Bean 名字 URL 处理器映射是通过一些列的父类继承最终实现处理器映射接口的。其中不同的父类抽象出一个独立的类级别，一个类级别完成一个最小化而又完善的功能。下图是它的

类继承实现的树结构。



图表 4-16

上图中灰色类，包括 Web 应用程序对象支持类和应用程序对象支持类是 Spring 环境项目的实现，Web 应用程序对象支持类用来初始化时得到 Web 应用程序环境,而应用程序对象支持类用来初始化时得到 Servlet 环境。这里我们不再剖析他们的实现。

抽象处理器映射是这个体系结构中直接实现处理器映射接口的抽象类，这个类继承了 Web 应用程序对象支持类，目的是监听 Web 应用程序环境初始化事件。在初始化事件中，初始化拦截器，这些拦截器是应用到所有处理器的。如下程序注释，

```
public void setInterceptors(Object[] interceptors) {
    //通过注入的方式设置通用拦截器，这些拦截器是对象类型，其中真正支持的类型包括HandlerInterceptor
和WebRequestInterceptor，这些通用拦截器是应用在所有的处理器上的
    this.interceptors.addAll(Arrays.asList(interceptors));
}

@Override
protected void initApplicationContext() throws BeansException {
    //提供占位符方法让子类添加新的拦截器对象
    extendInterceptors(this.interceptors);


    //初始化拦截器，因为拦截器有不同的是是实现，这里需要将不同的拦截器适配到最终的HandlerInterceptor
```

```
的是实现，这是通过HandlerInterceptorAdapter来实现的
    initInterceptors();
}

protected void initInterceptors() {
    //如果配置的通用拦截器不为空
    if (!this.interceptors.isEmpty()) {
        //对配置的通用拦截器进行适配
        this.adaptedInterceptors = new HandlerInterceptor[this.interceptors.size()];
        for (int i = 0; i < this.interceptors.size(); i++) {
            Object interceptor = this.interceptors.get(i);
            //对空的拦截器进行校验
            if (interceptor == null) {
                throw new IllegalArgumentException("Entry number " + i + " in interceptors
array is null");
            }
            //对每个拦截器进行适配
            this.adaptedInterceptors[i] = adaptInterceptor(interceptor);
        }
    }
}

protected HandlerInterceptor adaptInterceptor(Object interceptor) {
    if (interceptor instanceof HandlerInterceptor) {
        //如果拦截器是HandlerInterceptor本身的是实现，不需要适配
        return (HandlerInterceptor) interceptor;
    }
    else if (interceptor instanceof WebRequestInterceptor) {
        //如果拦截器是WebRequestHandlerInterceptorAdapter，则适配到通用的HandlerInterceptor
的实现
        return new WebRequestHandlerInterceptorAdapter((WebRequestInterceptor)
interceptor);
    }
    else {
        //不支持其他类型的拦截器
        throw new IllegalArgumentException("Interceptor type not supported: " +
interceptor.getClass().getName());
    }
}
```

当派遣器 Servlet 要求处理器映射翻译一个请求到处理器执行链的时候，抽象处理器则映射一个内部的处理器，连同初始化的拦截器一起构成处理器执行链返回。抽象处理器定义映射内部的处理器逻辑作为一个抽象的方法，子类需要实现这个方法来解析处理器。如下程序注释，

```
//实现处理器映射的方法，这个方法是派遣器Servlet要求翻译HTTP请求到处理器执行链的入口
public final HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception
{
    //使用某种映射逻辑，将请求映射到一个真正的处理器，这个方法定义为抽象的，子类必须实现它，例如，实现基于URL到Beean名称的映射逻辑
    Object handler = getHandlerInternal(request);

    //如果没有映射的处理器，则使用缺省的处理器
    if (handler == null) {
        //子类可以设置缺省的处理器，也可以通过注射的方式设置缺省的处理器
        handler = getDefaultHandler();
    }

    //如果没有发现任何处理器，则返回空处理器，派遣器Servlet将发送HTTP错误响应SC_NOT_FOUND(404)
    if (handler == null) {
        return null;
    }

    //如果内部映射逻辑实现返回一个字符串，则认为这个字符串是Bean的名字
    if (handler instanceof String) {
        String handlerName = (String) handler;

        //在应用程序环境中通过Bean名字查找这个Bean
        handler = getApplicationContext().getBean(handlerName);
    }

    //连同处理器拦截器一起构造处理器执行链
    return getHandlerExecutionChain(handler, request);
}

//抽象的处理器映射逻辑，这个逻辑的目的是映射一个HTTP请求到一个处理器对象，子类应该根据某些规则进行实现，通常是根据URL匹配Bean的名字来实现的，当然也可以匹配Bean的类或者其他的特征，我们将在第四小结剖析所有子类的实现
protected abstract Object getHandlerInternal(HttpServletRequest request) throws Exception;

protected HandlerExecutionChain getHandlerExecutionChain(Object handler,
HttpServletRequest request) {
    //判断处理器对象的类型
    if (handler instanceof HandlerExecutionChain) {
        //如果处理器已经是处理器对象
        HandlerExecutionChain chain = (HandlerExecutionChain) handler;

        //添加初始化的拦截器
```

```
            chain.addInterceptors(getAdaptedInterceptors());

            //返回处理器执行链
            return chain;
        }
        else {
            //否则使用处理器和初始化的拦截器构造处理器执行链的对象
            return new HandlerExecutionChain(handler, getAdaptedInterceptors());
        }
    }
```

我们看见抽象的处理器映射也实现了 Ordered 接口，这个接口是用来在有多个处理器映射可供派遣器 Servlet 使用时的优先级。

类体系结构中的下一个类是抽象 URL 处理器映射，正如我们所愿，抽象 URL 处理器映射实现了 getHandlerInternal()方法，在方法实现里通过请求的 URL 匹配相应的映射处理器和映射拦截器来返回处理器执行链对象的。

那么，这些映射拦截器和映射处理器是如何进行初始化的呢？映射的拦截器是在初始化的时候，通过在 Web 应用程序环境中查找得到的。这些映射拦截器必须是 MappedInterceptor 的子类，而且注册在 Web 应用程序环境中。它也提供了一个方法 registerHandler()，供子类调用注册响应的处理器。如下程序注释，

```
//改写了抽象处理器的拦截器初始化的方法，初始化更多配置在Web应用程序环境中的映射拦截器，映射拦截器是一个从URL到处理器拦截器对象映射的实现
@Override
protected void initInterceptors() {
    //初始化父类的通用拦截器，这些拦截器应用到所有的处理器上
    super.initInterceptors();

    //查找所有在Web应用程序环境中注册的MappedInterceptor的实现
    Map<String, MappedInterceptor> mappedInterceptors =
BeanFactoryUtils.beansOfTypeIncludingAncestors(
            getApplicationContext(), MappedInterceptor.class, true, false);

    //如果找到任何MappedInterceptor的实现
    if (!mappedInterceptors.isEmpty()) {
        //构造MappedInterceptor的集合类并且存储，这个集合类提供了通过URL过滤拦截器的功能
        this.mappedInterceptors = new
MappedInterceptors(mappedInterceptors.values().toArray(
                new MappedInterceptor[mappedInterceptors.size()]));
    }

}
```

```
protected void registerHandler(String[] urlPaths, String beanName) throws BeansException,
IllegalStateException {
    //注册的处理器必须映射到一个URL路径上
    Assert.notNull(urlPaths, "URL path array must not be null");

    //对于拥有多个URL的处理器，分别注册URL到处理器的映射
    for (String urlPath : urlPaths) {
        registerHandler(urlPath, beanName);
    }
}

//注册一个URL到一个处理器的映射
protected void registerHandler(String urlPath, Object handler) throws BeansException,
IllegalStateException {
    //URL和处理器都不能为空
    Assert.notNull(urlPath, "URL path must not be null");
    Assert.notNull(handler, "Handler object must not be null");

    //开始解析处理器
    Object resolvedHandler = handler;

    //如果没有配置懒惰初始化处理器选项，则把使用的处理器名字转换为Web应用程序环境中的Bean
    //如果配置懒惰初始化处理器选项，则这个转换是在返回处理器执行链的过程中实现的
    if (!this.lazyInitHandlers && handler instanceof String) {
        String handlerName = (String) handler;

        //如果这个Bean使用单例模式，则在初始化的时候进行转换，在后续的服务方法中不再进行转换，提高了
效率
        //但是，如果非单例模式，我们不能在初始化的时候进行转换，例如，如果Bean的范围是Session,服务
方法(getHanlderInternal)会的到不同的处理器实例对于不同的Session,在这种情况，如果初始化的时候进行转
换，则每次返回同一个Bean，变成了单例模式了
        if (getApplicationContext().isSingleton(handlerName)) {
            resolvedHandler = getApplicationContext().getBean(handlerName);
        }
    }

    //查看是否这个URL已经注册过处理器了
    Object mappedHandler = this.handlerMap.get(urlPath);
    if (mappedHandler != null) {
        //如果这个URL确实已经注册过处理器了
        if (mappedHandler != resolvedHandler) {
            //抛出异常，提示用户配置错误
            throw new IllegalStateException(
```

```
                    "Cannot map handler [" + handler + "] to URL path [" + urlPath +
                    "]: There is already handler [" + resolvedHandler + "] mapped.");
            }
        }
        else {
            //如果这个URL没有注册过处理器了
            if (urlPath.equals("/")) {
                //如果是根处理器
                if (logger.isInfoEnabled()) {
                    logger.info("Root mapping to handler [" + resolvedHandler + "]");
                }
                //设置根处理器
                setRootHandler(resolvedHandler);
            }
            else if (urlPath.equals("/*")) {
                //如果是默认处理器
                if (logger.isInfoEnabled()) {
                    logger.info("Default mapping to handler [" + resolvedHandler + "]");
                }
                //设置默认处理器
                setDefaultHandler(resolvedHandler);
            }
            else {
                //设置正常的处理器
                this.handlerMap.put(urlPath, resolvedHandler);
                if (logger.isInfoEnabled()) {
                    logger.info("Mapped URL path [" + urlPath + "] onto handler [" +
resolvedHandler + "]");
                }
            }
        }
    }
```

我们可以看到这些映射拦截器是在 Web 应用程序环境中查找得到的，他们必须实现 MappedInterceptor 接口。得到的这些 MappedInterceptor 接口保存在 MappedInterceptors 集合类中，这个类同时提供了基于 URL 路径过滤的功能。如下程序注释，

```
public Set<HandlerInterceptor> getInterceptors(String lookupPath, PathMatcher pathMatcher)
{
    //构造过滤结果集合
    Set<HandlerInterceptor> interceptors = new LinkedHashSet<HandlerInterceptor>();

    //遍历所有配置的映射拦截器
    for (MappedInterceptor interceptor : this.mappedInterceptors) {
```

```
            //如果配置的拦截器匹配此路径
            if (matches(interceptor, lookupPath, pathMatcher)) {
                //添加当前的拦截器到过滤结果集合中
                interceptors.add(interceptor.getInterceptor());
            }
        }

        //返回过滤结构
        return interceptors;
    }

    private boolean matches(MappedInterceptor interceptor, String lookupPath, PathMatcher
pathMatcher) {
        //映射拦截器保存这从URL到处理器拦截器的映射关系，这个方法取得这个处理器映射器支持的所有URL
Pattern
        String[] pathPatterns = interceptor.getPathPatterns();

        //判断是否配置了URL Pattern
        if (pathPatterns != null) {
            //如果配置了URL Pattern
            for (String pattern : pathPatterns)
                //查看是否存在一个URL Pattern匹配此路径
                if (pathMatcher.match(pattern, lookupPath)) {
                    //如果匹配此路径，则使用当前映射拦截器
                    return true;
                }
            }
           // 如果没有匹配此路径的URL Pattern，则不使用当前映射拦截器
            return false;
        } else {
            //在没有配置任何URL Pattern情况下，这个映射拦截器默认成为通用的拦截器，会被应用到所有的处理
器上
            return true;
        }
    }
```

抽象URL处理器映射初始化了拦截器和处理器之后，它将如何实现映射请求到处理器执行链的逻辑呢？正如我们所想，它通过URL精确匹配或者最佳匹配查找注册的拦截器和处理器，然后构造处理器执行链对象。如下程序注释，

```
//实现抽象处理器映射的抽象发放，提供基于URL匹配的实现
@Override
protected Object getHandlerInternal(HttpServletRequest request) throws Exception {
    //通过实用方法获得查找路径，这个查找路径 = URI - "http://" - hostname:port - application
```

```
context - servlet mapping prefix
    //例如 http://www.robert.com/jpetstore/petstore/insert/ - "http://" -
"www.robert.com" - "jpetstore" - "petstore" = "/insert"
    String lookupPath = this.urlPathHelper.getLookupPathForRequest(request);

    //匹配最佳匹配处理器
    Object handler = lookupHandler(lookupPath, request);
    if (handler == null) {
        //如果没有最佳匹配处理器
        Object rawHandler = null;
        if ("/".equals(lookupPath)) {
            //如果查找路径是根路径，则使用根处理器
            rawHandler = getRootHandler();
        }
        if (rawHandler == null) {
            //否则使用缺省处理器
            rawHandler = getDefaultHandler();
        }

        if (rawHandler != null) {
            //如果是根路径或者配置了缺省处理器

            if (rawHandler instanceof String) {
                //翻译处理器Bean名字到Bean对象本身，如果配置了懒惰加载为false，而且处理器是单例
模式，这个转换在初始化的时候已经做完了
                String handlerName = (String) rawHandler;
                rawHandler = getApplicationContext().getBean(handlerName);
            }

            //定义占位符方法校验处理器
            validateHandler(rawHandler, request);

            //增加新的处理器拦截器导出最佳匹配路径和查找路径，既然我们使用了根处理器或者缺省处理器，
这两个值都是查找路径
            handler = buildPathExposingHandler(rawHandler, lookupPath, lookupPath,
null);
        }
    }
    if (handler != null && this.mappedInterceptors != null) {
        //如果存在最佳匹配处理器，过滤映射拦截器，得到所有匹配的处理器拦截器，匹配过程在前文中已经分
析
        Set<HandlerInterceptor> mappedInterceptors =
                this.mappedInterceptors.getInterceptors(lookupPath,
```

```
this.pathMatcher);
        if (!mappedInterceptors.isEmpty()) {
            HandlerExecutionChain chain;
            if (handler instanceof HandlerExecutionChain) {
                //如果处理器拦截器是处理器执行链对象，则使用已存对象
                chain = (HandlerExecutionChain) handler;
            } else {
                //否则创建新的处理器执行链
                chain = new HandlerExecutionChain(handler);
            }
            //添加过滤得到的处理器拦截器到处理器执行链中
            chain.addInterceptors(mappedInterceptors.toArray(new
HandlerInterceptor[mappedInterceptors.size()]));
        }
        //【问题】既然方法开始调用了lookupHandler()，那么返回处理器映射器对象或者null,所以创建新
的处理器执行链逻辑不会执行。如果子类改写了lookupHandler()返回一个普通的处理器对象，创建新的处理器执行
链逻辑不被执行，但是新的处理器执行链对象没有返回，所以，添加的处理器拦截器将会丢失，这里程序设计有些不妥
    }
    if (handler != null && logger.isDebugEnabled()) {
        //记录日志成功的映射了处理器
        logger.debug("Mapping [" + lookupPath + "] to handler '" + handler + "'");
    }
    else if (handler == null && logger.isTraceEnabled()) {
        //记录日志映射处理器失败
        logger.trace("No handler mapping found for [" + lookupPath + "]");
    }

    //返回处理器，可能为空
    return handler;
}


protected Object lookupHandler(String urlPath, HttpServletRequest request) throws
Exception {
    // 首先执行精确匹配，查找路径和处理器配置的URL完全相同
    Object handler = this.handlerMap.get(urlPath);
    if (handler != null) {
        // 精确匹配成功
        if (handler instanceof String) {
            //翻译Bean名到Bean对象本身
            String handlerName = (String) handler;
            handler = getApplicationContext().getBean(handlerName);
        }
        //调用占位符方法校验处理器
```

```
            validateHandler(handler, request);

            //增加新的处理器拦截器导出最佳匹配路径和查找路径，既然精确匹配成功，这两个值都是查找路径
            return buildPathExposingHandler(handler, urlPath, urlPath, null);
        }
        // 执行最佳匹配方案
        List<String> matchingPatterns = new ArrayList<String>();
        for (String registeredPattern : this.handlerMap.keySet()) {
            //取得所有匹配的处理器注册的URL Pattern
            if (getPathMatcher().match(registeredPattern, urlPath)) {
                matchingPatterns.add(registeredPattern);
            }
        }

        //决定最佳匹配
        String bestPatternMatch = null;
        if (!matchingPatterns.isEmpty()) {
            //对匹配的URL Pattern进行排序
            Collections.sort(matchingPatterns,
getPathMatcher().getPatternComparator(urlPath));
            if (logger.isDebugEnabled()) {
                logger.debug("Matching patterns for request [" + urlPath + "] are " +
matchingPatterns);
            }

            //排序后数组的第一个匹配为最佳匹配
            bestPatternMatch = matchingPatterns.get(0);
        }
        if (bestPatternMatch != null) {
            //如果存在最佳匹配，则找到最佳匹配URL Pattern的处理器
            handler = this.handlerMap.get(bestPatternMatch);

            // 翻译Bean名到Bean对象本身
            if (handler instanceof String) {
                String handlerName = (String) handler;
                handler = getApplicationContext().getBean(handlerName);
            }

            //调用占位符方法校验处理器
            validateHandler(handler, request);

            //从URL中提取去除URL Pattern前缀的剩余部分，例如，URL Pattern是/petstore/*,而查找路径
是/petstore/insert,则结构是/insert
            String pathWithinMapping =
```

```
getPathMatcher().extractPathWithinPattern(bestPatternMatch, urlPath);

        //得到模板变量，并且添加新的处理器拦截器到处到HTTP请求中，这些模板变量在控制器的实现中会被用到
        //例如， URL Pattern是/petstore/insert/{id}, 查找路径是insert/1,则解析出一个模板变量id=1,并且到处到HTTP请求对象里
        Map<String, String> uriTemplateVariables =
                getPathMatcher().extractUriTemplateVariables(bestPatternMatch, urlPath);

        //创建处理器执行链对象
        return buildPathExposingHandler(handler, bestPatternMatch, pathWithinMapping, uriTemplateVariables);
    }
    // No handler found...
    return null;
}

protected Object buildPathExposingHandler(Object rawHandler,
        String bestMatchingPattern,
        String pathWithinMapping,
        Map<String, String> uriTemplateVariables) {

    //创建处理器执行器链对象
    HandlerExecutionChain chain = new HandlerExecutionChain(rawHandler);

    //添加路径到处理器拦截器,类定义如下
    chain.addInterceptor(new PathExposingHandlerInterceptor(bestMatchingPattern, pathWithinMapping));

    //添加模板变量处理器拦截器,类定义如下
    if (!CollectionUtils.isEmpty(uriTemplateVariables)) {
        chain.addInterceptor(new UriTemplateVariablesHandlerInterceptor(uriTemplateVariables));
    }
    return chain;
}

//导出最佳匹配的URL Pattern和查找路径中去除URL Pattern所匹配的那部分
private class PathExposingHandlerInterceptor extends HandlerInterceptorAdapter {

    private final String bestMatchingPattern;

    private final String pathWithinMapping;
```

```
    private PathExposingHandlerInterceptor(String bestMatchingPattern, String
pathWithinMapping) {
        this.bestMatchingPattern = bestMatchingPattern;
        this.pathWithinMapping = pathWithinMapping;
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) {
        //导出是在处理器拦截器的前置拦截器中实现的
        exposePathWithinMapping(this.bestMatchingPattern, this.pathWithinMapping,
request);
        return true;
    }

}

protected void exposePathWithinMapping(String bestMatchingPattern, String
pathWithinMapping, HttpServletRequest request) {
    //导出到请求属性中,在控制器中这些路径可以用来解析缺省的视图名
    request.setAttribute(HandlerMapping.BEST_MATCHING_PATTERN_ATTRIBUTE,
bestMatchingPattern);
    request.setAttribute(HandlerMapping.PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE,
pathWithinMapping);
}

//导出模板变量名值对
private class UriTemplateVariablesHandlerInterceptor extends HandlerInterceptorAdapter {

    private final Map<String, String> uriTemplateVariables;

    private UriTemplateVariablesHandlerInterceptor(Map<String, String>
uriTemplateVariables) {
        this.uriTemplateVariables = uriTemplateVariables;
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) {
        //导出是在处理器拦截器的前置拦截器中实现的
        exposeUriTemplateVariables(this.uriTemplateVariables, request);
        return true;
    }
```

```
}

protected void exposeUriTemplateVariables(Map<String, String> uriTemplateVariables,
HttpServletRequest request) {
    //导出到请求属性中,在控制器中这些参数可能成为业务逻辑的输入
    request.setAttribute(HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE,
uriTemplateVariables);
}
```

我们看到，抽象 URL 处理器映射是在 Web 应用程序环境初始化的时候初始化了拦截器，并且提供了通过 URL 对拦截器过滤的功能。同时提供方法注册处理器。现在我们将分析，一个子类应该如何注册处理器。在类的体系结构中的下一个类是抽象探测 URL 处理器映射，它探测 Web 应用程序环境中的所有 Bean，通过某种规则对 Bean 名字进行过滤来决定是否注册这个 Bean 作为一个处理器。如下程序所示，

```
//改写应用程序初始化方法，获得注册处理器的机会
@Override
public void initApplicationContext() throws ApplicationContextException {
    //保持原来的初始化实现
    super.initApplicationContext();

    //从Web应用程序环境中探测处理器
    detectHandlers();
}

protected void detectHandlers() throws BeansException {
    if (logger.isDebugEnabled()) {
        logger.debug("Looking for URL mappings in application context: " +
getApplicationContext());
    }

    //找到所有的对象类的实现，其实是eb应用程序环境中所有的Bean,并且返回Bean名字
    String[] beanNames = (this.detectHandlersInAncestorContexts ?

    BeanFactoryUtils.beanNamesForTypeIncludingAncestors(getApplicationContext(),
Object.class) :
            getApplicationContext().getBeanNamesForType(Object.class));

    // 对于每一个Bean的名字
    for (String beanName : beanNames) {
        //映射Bean的名字到一个或者多个URL
        String[] urls = determineUrlsForHandler(beanName);
        if (!ObjectUtils.isEmpty(urls)) {
            //如果这个Bean的名字能映射到一个或者多个URL，则注册Bean作为一个处理器
```

```
            registerHandler(urls, beanName);
        }
        else {
            //否则打印日志
            if (logger.isDebugEnabled()) {
                logger.debug("Rejected bean name '" + beanName + "': no URL paths
identified");
            }
        }
    }
}

//提供子类机会选择不同的策略映射名字到URL Pattern
protected abstract String[] determineUrlsForHandler(String beanName);
```

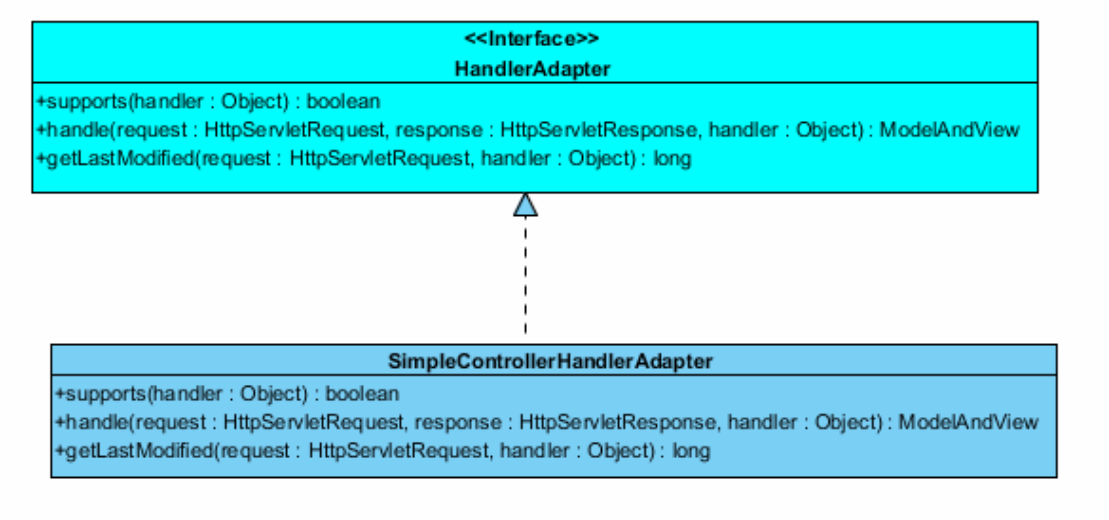
在上面的实现中，处理器是在应用程序环境中探测得到的。所以，我们称为这个类作为抽象探测 URL 处理器映射。但是，在抽象 URL 处理器映射中，初始化的实现也自动的在应用程序环境中探测了处理器拦截器的实现，所以，我认为把探测处理器拦截器的实现加入当前这个类中更合理。

事实上，映射 Bean 的名字到 URL Pattern 的实现是非常简单的，子类 Bean 名 URL 处理器映射通过查看是否一个 Bean 名字或者别名以字符/开头，如果是以字符/开头，则认为是一个处理器。如下程序所示，

```
protected String[] determineUrlsForHandler(String beanName) {
    List<String> urls = new ArrayList<String>();

    //如果Bean名以/开头
    if (beanName.startsWith("/")) {
        //则此Bean是一个处理器，Bean名字是此Bean的一个匹配的URL Pattern
        urls.add(beanName);
    }

    //取得Bean的所有别名
    String[] aliases = getApplicationContext().getAliases(beanName);
    for (int i = 0; i < aliases.length; i++) {
        //如果Bean的别名以/开头
        if (aliases[i].startsWith("/")) {
            //则此Bean是一个处理器，Bean的别名是此Bean的一个匹配的URL Pattern
            urls.add(aliases[i]);
        }
    }

    //返回此Bean定义的所有URL Pattern
```

```
    return StringUtils.toStringArray(urls);
}
```

流程分析到这里，派遣器 Servlet 已经通过处理器映射得到了处理器执行链对象。处理器执行链对象包含着一个以 Object 为类型的处理器，和一套应用在处理器上的处理器拦截器。接下来派遣器 Servlet 则轮询所有注册的处理器适配器（如图 4-15 第二步），查找是否有一个处理器适配器支持此处理器。Bean 名 URL 处理器映射(BeanNameUrlHandlerMapping)通常用于映射简单的控制器对象，所以，返回的处理器执行链对象里面通常包含着控制器接口（Controller）的实现类。这个轮询结果将返回简单控制处理适配器(SimpleControllerHandlerAdapter)，并且通过它将 HTTP 请求传递给控制器进行处理（如果 4-15 第四步和第五步）。简单的控制处理适配器的实现非常简单，正如它的名字所示，它仅仅是个适配器，请看如下类图，



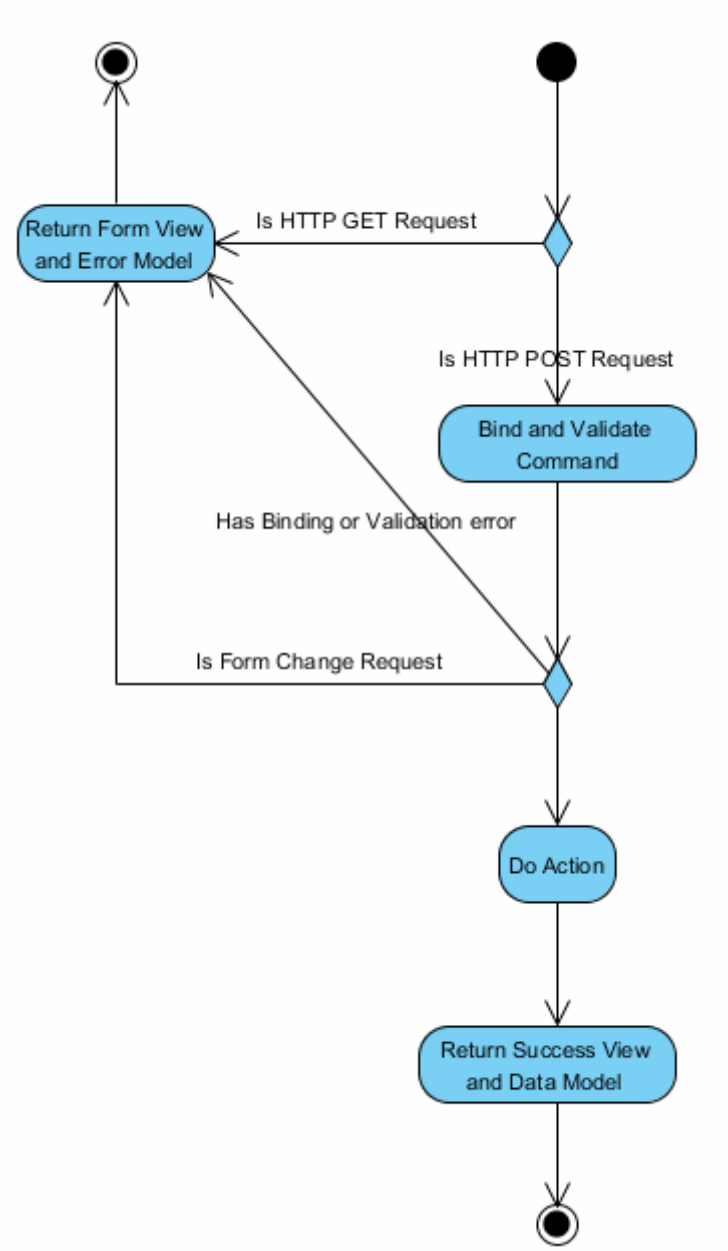
图表 4-17

如下程序注释，

```
public class SimpleControllerHandlerAdapter implements HandlerAdapter {

    public boolean supports(Object handler) {
        支持任何控制器接口的实现类
        return (handler instanceof Controller);
    }

    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response,
Object handler)
            throws Exception {

        适配HTTP请求，HTTP响应到处理器，这个处理器一定是控制器接口的实现，并且返回模型和视图对象，派遣器Servlet讲用模型和视图对象构造HTTP响应
        return ((Controller) handler).handleRequest(request, response);
    }
```

```
    public long getLastModified(HttpServletRequest request, Object handler) {
        if (handler instanceof LastModified) {
            如果一个控制器实现了最后修改接口，则适配最后修改请求到控制器
            return ((LastModified) handler).getLastModified(request);
        }

        如果没有实现了最后修改接口，则返回不支持最后修改操作，每次都返回一个全新的HTTP GET请求，尽管这个资源自从上次请求没有改变过
        return -1L;
    }

}
```

现在我们理解，一个 HTTP 请求经过处理器适配器传递到简单控制器的实现，简单控制器将实现任何必要的业务逻辑，最后返回模型数据和逻辑视图给作为总控制器的派遣器 Servlet。简单控制器接口有许多抽象和具体的实现，每个抽象和具体的实现都能完成一个特定的功能，结构清晰合理，易于扩展，使客户程序能够根据业务逻辑的需要选择不同的类继承处理不同的 HTTP 请求。下面我们以简单 Form 控制器为例说明它是如何完成对 HTTP 请求的处理并且返回模型数据和逻辑视图对象给派遣器 Servlet 的。下图是简单 Form 控制器的实现流程，

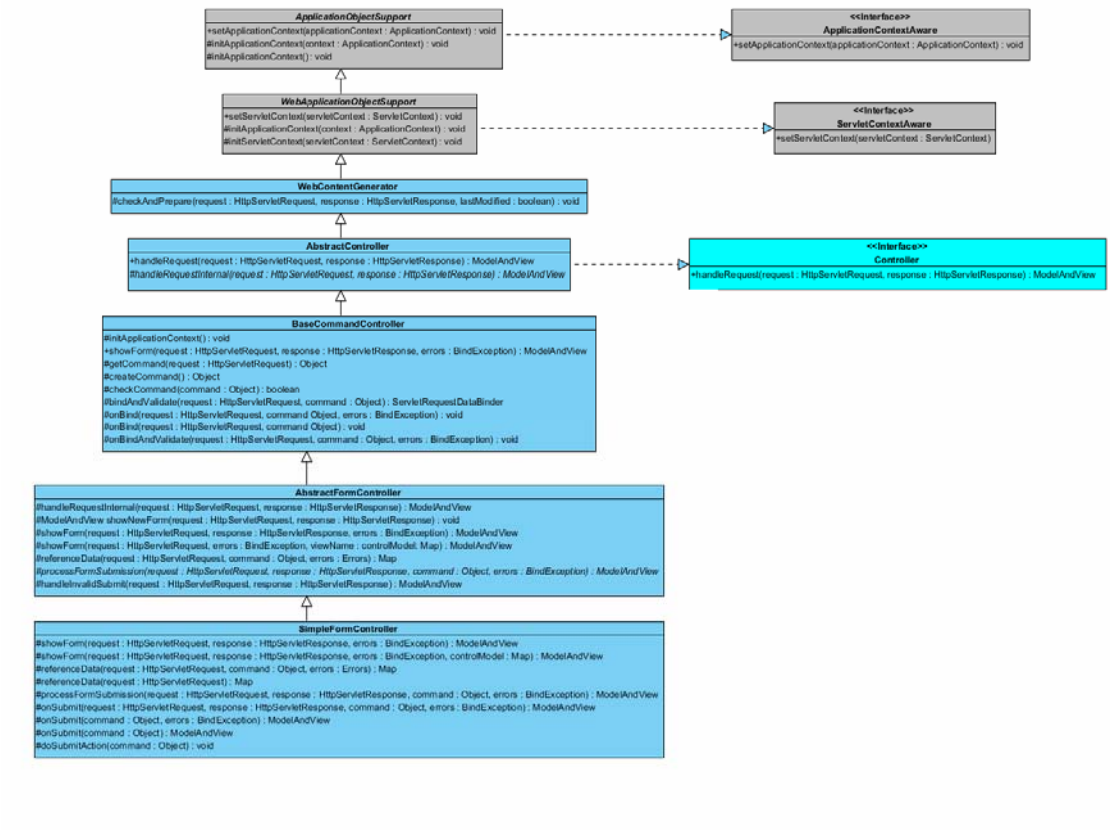


图表 4-18

从上图我们可以看到，简单 Form 控制器是通过 HTTP 请求方法来判断执行初始化操作还是业务逻辑操作。如果请求是 HTTP GET 方法，说明这次请求是这个模块的第一次加载，那么我们应该显示一个 Form 给用户，以至于用户可以填写业务逻辑的输入数据。当用户填写了业务逻辑数据后，提交 Form 给此模块，那么用户提交使用的一定是 HTTP POST 请求，这个请求包含着此模块业务逻辑的输入数据，所以，简单 Form 控制器会绑定这些输入数据到业务逻辑模型对象中，这里称为一个命令(Command)对象 ，以下分析中的命令对象，Form 的 Backing Bean 和业务逻辑模型对象指同一个事物，不再做区分。然后，对命令对象进行校验。如果在绑定或者校验过程中出现任何错误，则导出错误对象到 HTTP 请求的属性里，接下来在显示 Form 视图的时候，同时显示错误，于是，用户得知哪些输入是不合法的。当用户提交了完整而且有效的输入数据后，简单 Form 控制器在绑定和校验后，获得了此模块需要的输入数据后，使用服务层的服务进行业务逻辑的处理，处理后会返回处理结果数据，

也就是模型数据，这些模型数据连同成功视图一起返回给作为总控制器的派遣器 Servlet.

为了让程序具有可重用性和可扩展性，上面的流程并不是通过一个类实现的。而是通过多个类的继承最终由简单 Form 控制器实现的。如下是这些类的实现类图，



图表 4-19

我们在分析处理器映射的实现中得知上图中被标识为灰色的类和接口是用来对一个对象注入 Web 应用程序环境和 Servlet 环境的，这里我们不再详述其实现。继承自 Web 应用程序对象支持类的第一个类就是 Web 内容产生器，这个类用于校验支持的 HTTP 方法，也会产生 HTTP 缓存头信息等。如下程序注释，

```
//这个方法实现检查HTTP请求方法的合法性和产生缓存头信息，这个类是抽象类，这个方法是实体方法给子类使用的
//lastModified通常是根据控制器是不是实现了LastModified接口来决定的
protected final void checkAndPrepare(
        HttpServletRequest request, HttpServletResponse response, boolean lastModified)
    throws ServletException {

    // 代理另外一个方法并且传入更多的参数(配置的缓存时间)
    checkAndPrepare(request, response, this.cacheSeconds, lastModified);
}
```

```
protected final void checkAndPrepare(
        HttpServletRequest request, HttpServletResponse response, int cacheSeconds,
boolean lastModified)
    throws ServletException {

    // 检查是不是支持当前的HTTP请求方法
    String method = request.getMethod();
    if (this.supportedMethods != null && !this.supportedMethods.contains(method)) {
        // 如果不支持，则抛出异常，终止处理
        throw new HttpRequestMethodNotSupportedException(
                method, StringUtils.toStringArray(this.supportedMethods));
    }

    // 如果Session不存在，则抛出特定的异常HttpSessionRequiredException，这个异常通常被捕获，捕获
后创建Session,然后重试
    if (this.requireSession) {
        if (request.getSession(false) == null) {
            throw new HttpSessionRequiredException("Pre-existing session required but
none found");
        }
    }

    // 添加缓存信息到响应头中
    // 如果控制器支持最后修改操作，则设置必须重新校验信息到响应头中
    applyCacheSeconds(response, cacheSeconds, lastModified);
}

protected final void applyCacheSeconds(HttpServletResponse response, int seconds, boolean
mustRevalidate) {
    // 如果缓存的时间大于0
    if (seconds > 0) {
        // 设置缓存时间到响应头中
        cacheForSeconds(response, seconds, mustRevalidate);
    }
    else if (seconds == 0) {
        // 设置绝不缓存信息到响应头中
        preventCaching(response);
    }

    // 如果缓存时间小于0， 服务器不决定是否缓存，由客户自己决定
}

protected final void cacheForSeconds(HttpServletResponse response, int seconds, boolean
mustRevalidate) {
```

```
    // 这里我们需要兼容HTTP 1.0和HTTP 1.1
    if (this.useExpiresHeader) {
        // 如果是 HTTP 1.0 头信息，设置缓存的时间
        response.setDateHeader(HEADER_EXPIRES, System.currentTimeMillis() + seconds *
1000L);
    }
    if (this.useCacheControlHeader) {
        // 如果是HTTP 1.1 头信息，设置缓存时间
        String headerValue = "max-age=" + seconds;

        //如果支持最后修改操作，则设置标识重新校验
        if (mustRevalidate) {
            headerValue += ", must-revalidate";
        }

        response.setHeader(HEADER_CACHE_CONTROL, headerValue);
    }
}


protected final void preventCaching(HttpServletResponse response) {
    //在HTTP响应头中，设置不使用缓存
    response.setHeader(HEADER_PRAGMA, "no-cache");
    if (this.useExpiresHeader) {
        // 如果是HTTP 1.0 头信息，则使用1代表不使用缓存
        response.setDateHeader(HEADER_EXPIRES, 1L);
    }
    if (this.useCacheControlHeader) {
        // 如果是HTTP 1.1 头信息: "no-cache" 是标准值, "no-store" 用在firefox浏览器里的 , 他们
都是用來告诉浏览器不需要缓存网页的
        response.setHeader(HEADER_CACHE_CONTROL, "no-cache");
        if (this.useCacheControlNoStore) {
            response.addHeader(HEADER_CACHE_CONTROL, "no-store");
        }
    }
}
```

类层次的下一个类抽象控制器类实现了控制器接口。但是，这个类除了对方法 handleRequest() 进行同步没有做任何实现，并且适配到一个抽象方法 handleRequestInternal()，这个方法是由子类来实现的。如下程序所示，

```
public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response)
        throws Exception {
```

```
    // 调用Web内容产生器进行检查HTTP方法和产生缓存信息的响应头
    checkAndPrepare(request, response, this instanceof LastModified);

    // 如果Session内同步标识打开，则对handleRequestInternal()进行同步调用
    if (this.synchronizeOnSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            // 如果Session存在则取得同步对象，缺省是Session对象自己
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                // 在Session内同步处理请求
                return handleRequestInternal(request, response);
            }
        }
    }

    // 如果不需要Session内同步，直接调用了抽象方法handleRequestInternal()
    return handleRequestInternal(request, response);
}

// 这个方法需要由子类实现，来处理不同的业务流程
protected abstract ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
   throws Exception;
```

类实现体系结构中的下一个类基本命令控制器中引入了领域对象模型命令对象(Command)的概念，它是一个通用的对象类型，用来存储输入的参数信息。并且引入了对这个领域对象模型进行初始化和校验的逻辑，如下程序注释，

```
//继承自应用程序支持类的初始化方法，实现更多的初始化逻辑
protected void initApplicationContext() {
    //如果注册了校验器
    if (this.validators != null) {
        for (int i = 0; i < this.validators.length; i++) {
            //如果存在着不支持配置的命令类的校验器，则抛出异常，停止处理
            if (this.commandClass != null
&& !this.validators[i].supports(this.commandClass))
                throw new IllegalArgumentException("Validator [" + this.validators[i] +
                        "] does not support command class [" +
                        this.commandClass.getName() + "]");
        }
    }
}
```

```
//提供了取得命令的方法给子类使用
protected Object getCommand(HttpServletRequest request) throws Exception {
    //默认实现是通过配置的类实例化一个新的命令对象
    return createCommand();
}

protected final Object createCommand() throws Exception {
    //如果没有配置命令类，则抛出异常推出
    if (this.commandClass == null) {
        throw new IllegalStateException("Cannot create command without commandClass being
set - " +
                "either set commandClass or (in a form controller) override
formBackingObject");
    }
    if (logger.isDebugEnabled()) {
        logger.debug("Creating new command of class [" + this.commandClass.getName() +
"]");
    }

    //通过配置的类实例化一个命令对象
    return BeanUtils.instantiateClass(this.commandClass);
}

protected final boolean checkCommand(Object command) {
    //查看命令是否和配置的命令类匹配
    return (this.commandClass == null || this.commandClass.isInstance(command));
}

//提供方法进行绑定命令和校验命令给子类使用
protected final ServletRequestDataBinder bindAndValidate(HttpServletRequest request,
Object command)
throws Exception {
    //通过当前HTTP请求和命令对象创建绑定对象
    ServletRequestDataBinder binder = createBinder(request, command);

    //把绑定结构放入到版定异常中，erros和binder共享一个绑定结构的引用，校验中会把错误存贮在这个绑定
结果上
    BindException errors = new BindException(binder.getBindingResult());

    //查看是否配置了跳过绑定
    if (!suppressBinding(request)) {
        //如果没有跳过绑定
```

```
            //绑定请求数据到命令数据里
            binder.bind(request);

            //绑定后，激发绑定后事件
            onBind(request, command, errors);

            //查看校验设置
            if (this.validators != null && isValidateOnBinding()
&& !suppressValidation(request, command, errors)) {
                //如果绑定时校验开启和跳过校验关闭，则使用校验器进行校验命令
                for (int i = 0; i < this.validators.length; i++) {
                    //在校验器里，如果有错误，就会存入errors对象里面，而errors是从绑定的绑定结果构造
的，传递的是引用，所以共享一个结果对象，这个对象最后会被逐层返回
                    ValidationUtils.invokeValidator(this.validators[i], command, errors);
                }
            }

            //绑定和校验后，激发绑定和校验后时间
            onBindAndValidate(request, command, errors);
        }

        //如果设置了跳过绑定，则直接将绑定对象返回
        return binder;
    }

    protected boolean suppressBinding(HttpServletRequest request) {
        //默认情况下是进行绑定操作的
        return false;
    }

    protected void onBind(HttpServletRequest request, Object command, BindException errors)
throws Exception {
        //响应绑定操作执行后的事件方法，子类可以改写实现特殊的绑定操作
        onBind(request, command);
    }

    protected void onBind(HttpServletRequest request, Object command) throws Exception {
        //占位符方法，同上方法同能
    }

    protected void onBindAndValidate(HttpServletRequest request, Object command, BindException
errors)
throws Exception {
        //响应绑定和校验操作执行后的事件方法，子类可以改写实现特殊的校验操作
```

```
}

protected ServletRequestDataBinder createBinder(HttpServletRequest request, Object
command)
throws Exception {
    //实例化一个Servlet请求数据绑定
    ServletRequestDataBinder binder = new ServletRequestDataBinder(command,
getCommandName());

    //对绑定进行简单的初始化，例如，绑定使用的消息代码解析器，绑定错误处理器，客户化编辑器等
    prepareBinder(binder);

    //提供机会通过Web绑定初始化器初始化绑定对象
    initBinder(request, binder);

    //返回绑定对象
    return binder;
}

protected final void prepareBinder(ServletRequestDataBinder binder) {
    //设置是否使用直接字段存取
    if (useDirectFieldAccess()) {
        binder.initDirectFieldAccess();
    }

    //设置消息代码解析器
    if (this.messageCodesResolver != null) {
        binder.setMessageCodesResolver(this.messageCodesResolver);
    }

    //设置绑定错误处理器
    if (this.bindingErrorProcessor != null) {
        binder.setBindingErrorProcessor(this.bindingErrorProcessor);
    }

    //这是客户化
    if (this.propertyEditorRegistrars != null) {
        for (int i = 0; i < this.propertyEditorRegistrars.length; i++) {
            this.propertyEditorRegistrars[i].registerCustomEditors(binder);
        }
    }
}
protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder)
throws Exception {
```

```
    if (this.webBindingInitializer != null) {
        //使用Web绑定初始化器对绑定进行初始化
        this.webBindingInitializer.initBinder(binder, new ServletWebRequest(request));
    }
}
```

由此可见，基本命令控制器的实现中提供了实用方法创建命令对象，绑定命令对象和校验命令对象。那么，在下一个类层次中，抽象 Form 控制器则使用这些方法进行创建命令对象，校验命令对象，进而实现整体的显示 Form，处理 Form 和显示成功试图的流程。如下程序注释，

```
protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
        throws Exception {

    // 判断是提交Form还是显示Form
    if (isFormSubmission(request)) {
        // 如果是提交Form(HTTP POST)
        try {
            // 取得命令对象，新创建的命令对象或者从Session中提取的命令对象，取决与配置
            Object command = getCommand(request);
            //
            使用父类提供的实用方法进行绑定和校验
            ServletRequestDataBinder binder = bindAndValidate(request, command);

            // 构造绑定错误对象，这个对象是简历在绑定结构智商的
            BindException errors = new BindException(binder.getBindingResult());

            // 处理提交Form逻辑
            return processFormSubmission(request, response, command, errors);
        }
        catch (HttpSessionRequiredException ex) {
            // 如果没有form-bean(命令对象）存在在Session里
            if (logger.isDebugEnabled()) {
                logger.debug("Invalid submit detected: " + ex.getMessage());
            }

            // 如果需要Session存在或者使用了Session From但是没有Session Form存在，则初始化
Session Form,然后重试
            return handleInvalidSubmit(request, response);
        }
    }

    else {
```

```
            如果是显示Form(HTTP GET)
            // 则显示Form输入视图
            return showNewForm(request, response);
        }
    }

    protected boolean isFormSubmission(HttpServletRequest request) {
        // 只有HTTP POST才会使用提交Form的处理逻辑
        return "POST".equals(request.getMethod());
    }



    protected final Object getCommand(HttpServletRequest request) throws Exception {
        // 如果不是session-form 模式, 穿件一个全新的form-backing命令对象.
        if (!isSessionForm()) {
            return formBackingObject(request);
        }

        // 如果是Session-form 模式
        HttpSession session = request.getSession(false);
        if (session == null) {
            // 如果Session不存在，则抛出特殊的异常HttpSessionRequiredException
            throw new HttpSessionRequiredException("Must have session when trying to bind (in
    session-form mode)");
        }
        String formAttrName = getFormSessionAttributeName(request);
        // 取得用来保存在Session里的Session Form的关键字
        Object sessionFormObject = session.getAttribute(formAttrName);
        if (sessionFormObject == null) {
            //如果关键字不存在，则抛出特殊的异常HttpSessionRequiredException
            throw new HttpSessionRequiredException("Form object not found in session (in
    session-form mode)");
        }

        // 抛出特殊的异常HttpSessionRequiredException,会被抓住，然后，创建Session和命令本身，然后重
    试


        if (logger.isDebugEnabled()) {
            logger.debug("Removing form session attribute [" + formAttrName + "]");
        }

        // 流程处理后，将命令对象从Session中移除，如果再次需要命令对象，则需要重建命令对象，然后绑定，校
```

```
验等等
    session.removeAttribute(formAttrName);

    // 调用占位符方法取得当前的命令对象
    return currentFormObject(request, sessionFormObject);
}

protected Object formBackingObject(HttpServletRequest request) throws Exception {
    //使用父类的使用方法创建一个全新的form-backing命令对象
    return createCommand();
}

protected Object currentFormObject(HttpServletRequest request, Object sessionFormObject)
throws Exception {
    // 返回传入的命令对象本身，也可能对命令对象做客户化的改变
    return sessionFormObject;
}

//抽象方法，子类用来实现对HTTP请求的处理逻辑
protected abstract ModelAndView processFormSubmission(
        HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors)
        throws Exception;

//如果使用Session Form模式，但是没有Session或者Session里不存在命令对象
protected ModelAndView handleInvalidSubmit(HttpServletRequest request,
HttpServletResponse response)
throws Exception {
    // 创建一个命令对象
    Object command = formBackingObject(request);

    // 校验和绑定
    ServletRequestDataBinder binder = bindAndValidate(request, command);
    BindException errors = new BindException(binder.getBindingResult());

    // 进行对HTTP提交请求的处理，这个方法是抽象方法，子类需要实现响应的对HTTP请求的处理逻辑
    return processFormSubmission(request, response, command, errors);
}


protected final ModelAndView showNewForm(HttpServletRequest request, HttpServletResponse
response)
throws Exception {
    logger.debug("Displaying new form");
```

```
    // 显示一个Form视图
    return showForm(request, response, getErrorsForNewForm(request));
}

protected final BindException getErrorsForNewForm(HttpServletRequest request) throws
Exception {
    // 穿件命令对象(form-back对象)
    Object command = formBackingObject(request);

    // 这个命令对象不能为空
    if (command == null) {
        throw new ServletException("Form object returned by formBackingObject() must not
be null");
    }

    // 简单命令对象和配置的命令类是否兼容
    if (!checkCommand(command)) {
        throw new ServletException("Form object returned by formBackingObject() must match
commandClass");
    }

    //创建绑定对象，但是不需要真正的绑定和校验
    ServletRequestDataBinder binder = createBinder(request, command);
    BindException errors = new BindException(binder.getBindingResult());

    // 缺省情况下对于显示Form视图，不绑定命令对象
    if (isBindOnNewForm()) {
        // 手工配置为显示Form视图时，进行绑定
        logger.debug("Binding to new form");
        binder.bind(request);

        // 传递绑定时间
        onBindOnNewForm(request, command, errors);
    }

    // 返回绑定结果错误对象，这里面可能不包含错误
    return errors;
}

//子类实现用来决定如何显示Form视图
protected abstract ModelAndView showForm(
        HttpServletRequest request, HttpServletResponse response, BindException errors)
        throws Exception;
```

如此可见，抽象 Form 控制器实现了处理一个 Web Form 的主要流程。也就是说，在 Form 初始化的时候，则显示 Form 视图，而在 Form 提交的时候，则处理 Form 提交，最后显示成功视图。因此，定义了两个抽象的方法，showForm()和 processFormSubmission()。也正如我们所想，子类通过实现这两个方法来处理不同的流程。在简单 Form 控制器的实现中，showForm()简单的显示了配置的 Form 视图。而 processFormSubmission()的实现则判断是否有绑定和校验错误，如果有错误，则转发请求到 Form 视图，在 Form 视图中显示错误并且提示用户重新输入。如果用户输入了正确有效的数据并且提交 Form，简单 Form 控制器则使用服务层的服务处理逻辑，并且连同包含处理结果的模型数据和成功视图返回给作为主控制器的派遣器 Servlet。如下程序注释，

```
//实现父类的抽象方法，最终处理一个对Form初始化HTTP请求
@Override
protected ModelAndView showForm(
        HttpServletRequest request, HttpServletResponse response, BindException errors)
        throws Exception {

    return showForm(request, response, errors, null);
}

protected ModelAndView showForm(
        HttpServletRequest request, HttpServletResponse response, BindException errors,
Map controlModel)
        throws Exception {

    //显示配置的Form视图
    return showForm(request, errors, getFormView(), controlModel);
}

@Override
protected ModelAndView processFormSubmission(
        HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors)
        throws Exception {

    if (errors.hasErrors()) {
        if (logger.isDebugEnabled()) {
            logger.debug("Data binding errors: " + errors.getErrorCount());
        }
        //如果绑定和校验返回错误，则转发到Form视图，并且显示错误，提示用户重新输入
        return showForm(request, response, errors);
    }
    else if (isFormChangeRequest(request, command)) {
        logger.debug("Detected form change request -> routing request to onFormChange");
```

```
            //如果是Form改变请求，例如一个相关对话框数据改变等
            onFormChange(request, response, command, errors);

            //转发到Form视图，继续用户输入
            return showForm(request, response, errors);
        }
        else {
            logger.debug("No errors -> processing submit");

            //处理提交Form逻辑并且返回模型和视图对象
            return onSubmit(request, response, command, errors);
        }
    }

    protected ModelAndView onSubmit(Object command, BindException errors) throws Exception {
        //执行业务逻辑处理，返回模型和视图对象
        ModelAndView mv = onSubmit(command);
        if (mv != null) {
            // 如果返回了模型和视图对象，则直接返回它给派遣器Servlet，缺省情况下并不返回模型和视图对象
            return mv;
        }
        else {
            // 缺省情况下返回配置的成功视图
            if (getSuccessView() == null) {
                throw new ServletException("successView isn't set");
            }
            return new ModelAndView(getSuccessView(), errors.getModel());
        }
    }

    protected ModelAndView onSubmit(Object command) throws Exception {
        //业务逻辑处理的缺省实现，调用一个占位符方法
        //子类可以改写此方法的实现，调用服务层处理逻辑，返回模型和视图对象
        doSubmitAction(command);
        return null;
    }

    protected void doSubmitAction(Object command) throws Exception {
        //如果子类不需要返回特殊的视图，那么仅仅需要改写此方法
    }
```

简单 Form 控制器是这个实现体系结构中的最后一个类，在使用它之前，需要为它配置 Form 视图和成功视图，它就可以开始工作了。但是，一个真正的控制器类应该改写它的 doActionSubmit()方法，从而实现需要的业务逻辑调用。

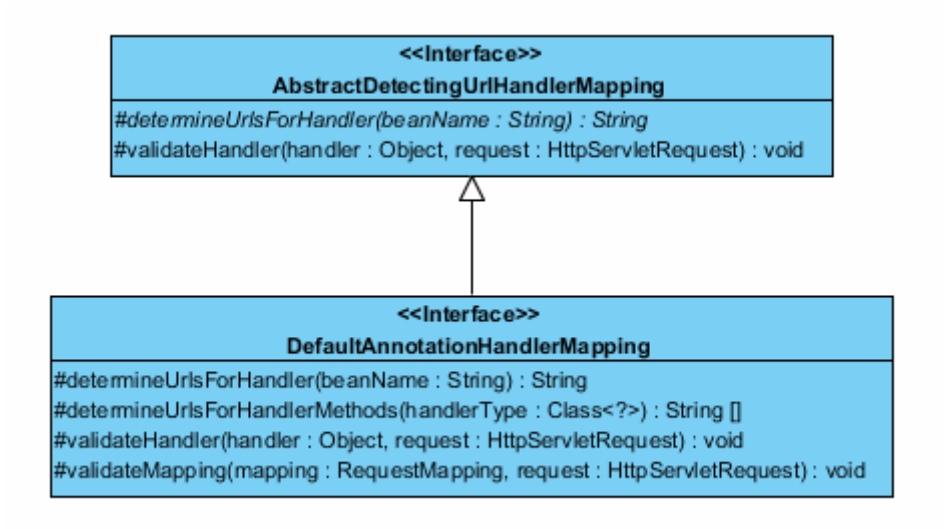
最总我们可以看见，简单 Form 控制器实现并不是单一的类实现，在实现上有很多的层次，每个层次完成一个相对独立的功能，下一层紧紧的依赖于上一层。当你选择实现一个控制器的时候，可以根据需求选择实现哪个层次的抽象类控制器，甚至控制器接口本身。

4.2.1.2基于注解控制器流程的实现

上一节，我们详细的分析了基于简单控制器流程的实现，事实上，许多的简单控制器的实现已经不被推荐使用。自从 2.5 发布以后，Spring 开始鼓励使用基于注解控制器的流程。基于注解的控制器流程具有实现方法简单，程序代码清晰易读等特点。

基于注解控制器的流程和基于简单控制器的流程的实现非常相似，派遣器 Servlet 在处理一个 HTTP 请求的时候，它通过缺省注解处理器映射(DefaultAnnotationHandlerMapping)把 HTTP 请求映射到响应的注解控制器(@Contoller)，然后，把控制流传给注解方法处理器适配器(AnnotationMethodHandlerAdapter)。注解方法处理器适配器并不是简单的传递控制流给注解控制器，而是以一定规则查找注解控制器里面的处理器方法，并且通过反射的方式映射 HTTP 请求信息到方法参数，然后使用反射调用方法，得到方法的返回结果后，再根据一定的规则把返回结果映射到模型和视图对象，进而返回给作为总控制器的派遣器 Servlet。

事实上，缺省注解处理器映射的实现重用了简单控制器流程的处理器映射的实现体系结构。回顾上一小结中分析的 Bean 名 URL 处理器映射继承自抽象探测 URL 处理器映射，实现了其抽象方法 determineUrlsForHandler(),在这个方法实现中，把所有以左划线(/)开头的 Bean 名字注册作为一个简单的控制器。缺省注解处理器映射的实现同样也实现了抽象方法 determineUrlsForHandler()，如果某个 Bean 中使用了请求映射(@RequestMappings)注解，则注册这个 Bean 作为一个注解控制器。如下类图所示，



图表 4-20

如上图所示，缺省注解处理器映射实现了方法 determineUrlsForHandler(),查找 Bean 类型级别

的请求映射注解和方法级别的请求映射注解，如果两个级别的请求映射注解都存在，结合两个级别的请求映射注解，否则使用方法级别的请求映射注解，构造出一个 URL Pattern 集合，并且返回这个 URL Pattern 集合，抽象探测 URL 处理器映射就会使用这个集合的每一个元素作为关键字注册当前的 Bean 作为一个注解控制器。如下程序注释，

```
@Override
protected String[] determineUrlsForHandler(String beanName) {
    // 取得Web应用程序环境
    ApplicationContext context = getApplicationContext();

    // 取得当前Bean的类型
    Class<?> handlerType = context.getType(beanName);

    // 找到类型级别的请求映射注解(@RequestMapping)
    RequestMapping mapping = context.findAnnotationOnBean(beanName,
RequestMapping.class);

    if (mapping != null) {
        // 如果类型级别声明了请求映射注解(@RequestMapping)

        // 缓存处理器类型和类型级别请求映射注解(@RequestMapping)
        this.cachedMappings.put(handlerType, mapping);

        // 开始查找所有应用在当前Bean的URL Pattern,这些URL Pattern是根据类型级别和方法级别的请求
映射注解(@RequestMapping)计算得到的
        Set<String> urls = new LinkedHashSet<String>();

        // 取得类型级别的URL Pattern
        String[] typeLevelPatterns = mapping.value();
        if (typeLevelPatterns.length > 0) {
            // 如果类型级别的请求映射注解(@RequestMapping)指定了URL Pattern，结合类型级别的URL
Pattern和方法级别的URL Pattern
            String[] methodLevelPatterns = determineUrlsForHandlerMethods(handlerType);

            // 对于每个类型级别的URL Pattern结合每一个方法级别的URL Pattern
            for (String typeLevelPattern : typeLevelPatterns) {
                // 如果没有使用斜线/开头，则添加一个斜线/开头，因为匹配时候的URI是以/开头的
                if (!typeLevelPattern.startsWith("/")) {
                    typeLevelPattern = "/" + typeLevelPattern;
                }

                for (String methodLevelPattern : methodLevelPatterns) {
                    // 结合任意一个类型级别的URL Pattern和任意一个方法级别的URL Pattern
                    String combinedPattern = getPathMatcher().combine(typeLevelPattern,
```

```
methodLevelPattern);

                    // 添加结合的URL Pattern到结果集合
                    addUrlsForPath(urls, combinedPattern);
                }

                // 添加类型级别的URL Pattern到结果集合
                addUrlsForPath(urls, typeLevelPattern);
            }

            // 返回URL Pattern结果集合
            return StringUtils.toStringArray(urls);
        }
        else {
            // 如果类型级别的请求映射注解(@RequestMapping)没有配置URL Pattern，则直接返回所有方法级别的请求映射注解(@RequestMapping)中的URL Pattern
            return determineUrlsForHandlerMethods(handlerType);
        }
    }
    else if (AnnotationUtils.findAnnotation(handlerType, Controller.class) != null) {
        // 如果没有声明类型级别的请求映射注解(@RequestMapping)，但是声明了类型级别的控制器注解(@Controller)，则直接返回所有方法级别的请求映射注解(@RequestMapping)中的URL Pattern
        return determineUrlsForHandlerMethods(handlerType);
    }
    else {
        // 如果既没有方法级别的请求映射注解(@RequestMapping)也没有类型级别的控制器注解(@Controller)，则这个Bean不是注解控制器，返回null
        return null;
    }
}

protected String[] determineUrlsForHandlerMethods(Class<?> handlerType) {
    // 声明成final是为了您名类的访问
    final Set<String> urls = new LinkedHashSet<String>();

    // 如果是代理类则使用代理类实现的接口，否则用类本身来探测处理器方法
    Class<?>[] handlerTypes =
            Proxy.isProxyClass(handlerType) ? handlerType.getInterfaces() : new Class<?>[]{handlerType};

    // 对于每个类型，遍历类型中的每个处理器方法
    for (Class<?> currentHandlerType : handlerTypes) {
        ReflectionUtils.doWithMethods(currentHandlerType, new ReflectionUtils.MethodCallback() {
```

```
            public void doWith(Method method) {
                // 取得当前方法级别声明的请求映射注解(@RequestMapping)
                RequestMapping mapping = AnnotationUtils.findAnnotation(method,
RequestMapping.class);
                if (mapping != null) {
                    // 如果方法级别的请求映射注解(@RequestMapping)存在

                    // 遍历每一个请求映射注解(@RequestMapping)中注册的URL Pattern
                    String[] mappedPaths = mapping.value();
                    for (String mappedPath : mappedPaths) {
                        // 添加到返回URL Pattern集合中
                        addUrlsForPath(urls, mappedPath);
                    }
                }
            }
        });
    }

    // 返回结果集合
    return StringUtils.toStringArray(urls);
}

protected void addUrlsForPath(Set<String> urls, String path) {
    // 直接添加到结果集合中
    urls.add(path);

    // 如果使用缺省后缀，则添加另外两个URL到URL Pattern集合中，用于匹配扩展名后缀和斜线/后缀
    if (this.useDefaultSuffixPattern && path.indexOf('.') == -1 && !path.endsWith("/"))
{
        urls.add(path + ".*");
        urls.add(path + "/");
    }
}
```

在缺省注解处理器映射中除了实现了提取注解处理器中配置的URL Pattern外，还改写了一个校验处理器的方法validateHandler()，这个方法的实现根据类型级别的请求映射的配置，校验当前的请求是否能够应用到这个处理器上。这个校验方法是在派遣器Servlet将一个请求映射到响应的处理器的时候调用的。具体逻辑如下，

- 如果处理器类型级别请求映射定义了HTTP方法，则当前HTTP请求方法必须是请求映射定义的这些HTTP方法之一。
- 如果处理器类型级别请求映射定义了HTTP参数，则当前HTTP请求参数必须包含请求映射定义的所有的HTTP参数。
- 如果处理器类型级别请求映射定义了HTTP头，则HTTP请求头必须包含请求映射定义的所有的HTTP头。

否则，这个处理器不能处理当前的HTTP请求，抛出异常，终止处理。可见，类型级别的请求映射是优先校验的，方法级别的请求映射是后来校验的，所以，我们得出一下结论。

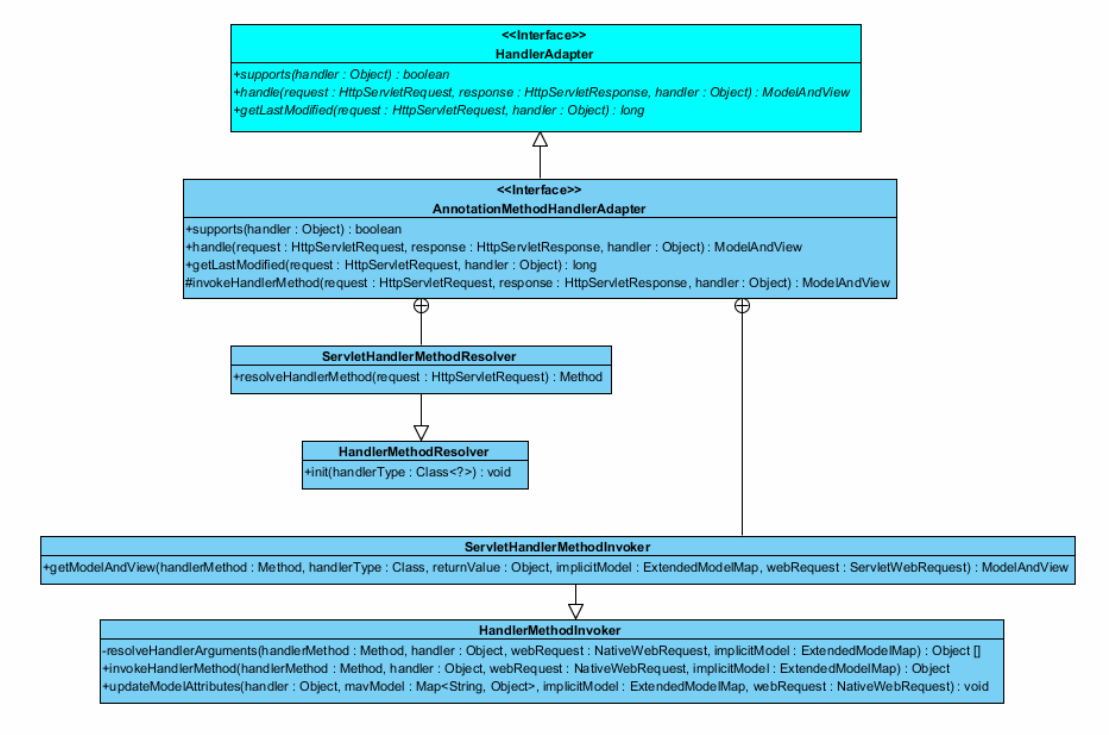
- 方法级别请求映射定义的HTTP方法必须是类型级别请求映射定义的HTTP方法的子集，否则没有意义。
- 方法级别请求映射定义的HTTP参数可以多余类型级别请求映射定义的HTTP参数。
- 方法级别请求映射定义的HTTP头可以多余类型级别请求映射定义的HTTP头。

如下程序注释，

```
//这个方法是在派遣器Servlet查找到一个处理器对象时调用的
@Override
protected void validateHandler(Object handler, HttpServletRequest request) throws
Exception {
    // 从缓存中得到声明在处理器类级别的请求映射注解(@ReuqestMapping)
    RequestMapping mapping = this.cachedMappings.get(handler.getClass());
    if (mapping == null) {
        // 如果缓存中没有，则直接取得，通常情况下，初始化的时候加入的
        mapping = AnnotationUtils.findAnnotation(handler.getClass(),
RequestMapping.class);
    }
    if (mapping != null) {
        // 校验方法级别的映射信息
        validateMapping(mapping, request);
    }
}

protected void validateMapping(RequestMapping mapping, HttpServletRequest request) throws
Exception {
    // 取得类型级别声明的HTTP方法
    RequestMethod[] mappedMethods = mapping.method();
    if (!ServletAnnotationMappingUtils.checkRequestMethod(mappedMethods, request)) {
        // 如果当前请求的HTTP方法不是类型级别声明的HTTP方法，而且类型级别声明了一个或者一个以上的
HTTP方法
        String[] supportedMethods = new String[mappedMethods.length];

        // 取得类型级别声明的HTTP方法，这些方法是支持的HTTP方法
        for (int i = 0; i < mappedMethods.length; i++) {
            supportedMethods[i] = mappedMethods[i].name();
        }

        // 抛出异常，提示哪些HTTP方法是支持的
        throw new HttpRequestMethodNotSupportedException(request.getMethod(),
supportedMethods);


    // 取得类型级别声明的HTTP参数
```

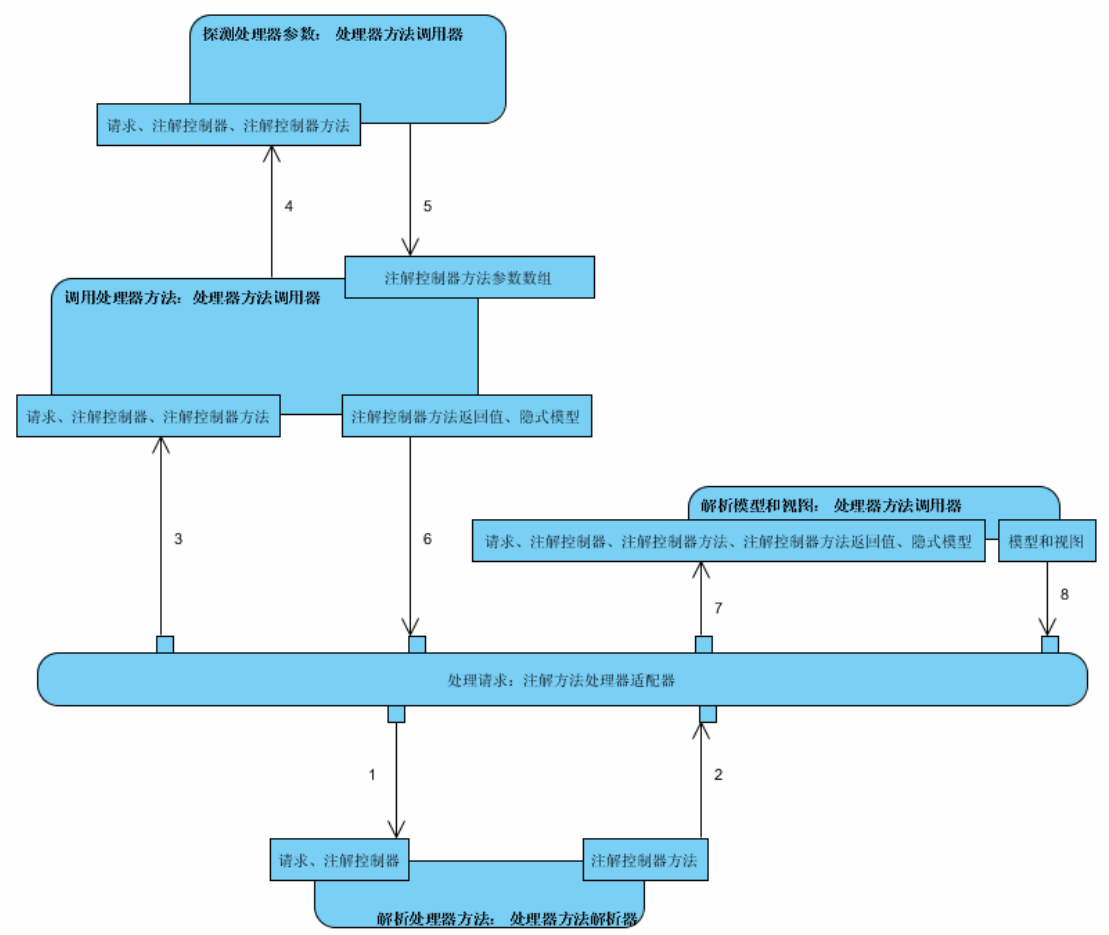
```
    String[] mappedParams = mapping.params();

    if (!ServletAnnotationMappingUtils.checkParameters(mappedParams, request)) {
        // 如果当前请求的HTTP参数不包含类型级别声明的HTTP参数，而且类型级别声明了一个或者一个以上的
HTTP参数

        // 抛出异常，提示那些参数应该包含在HTTP请求中
        throw new UnsatisfiedServletRequestParameterException(mappedParams,
request.getParameterMap());
    }

    // 取得类型级别声明的HTTP头
    String[] mappedHeaders = mapping.headers();
    if (!ServletAnnotationMappingUtils.checkHeaders(mappedHeaders, request)) {
        // 如果当前请求的HTTP头不包含类型级别声明的HTTP头，而且类型级别声明了一个或者一个以上的HTTP
头

        // 抛出异常，提示哪些头信息应该包含在在HTTP请求中

        throw new ServletRequestBindingException("Header conditions \"" +
                StringUtils.arrayToDelimitedString(mappedHeaders, ", ") +
                "\" not met for actual request");
    }

    //可见，如果类型级别声明了支持的HTTP方法，那么，方法中声明的方法应该是类型级别声明HTTP方法的子集，
参数和头信息的声明则可以是递增的
}
```

通过以上我们的分析，我们理解在基于注解控制器流程的实现中，缺省注解处理器映射是通过处理器中声明的请求映射注解注册注解控制器以及查找注解控制器的。作为总控制器的派遣器 Servlet 通过 HTTP 请求得到一个注解控制器，将注解控制器等传给注解方法处理器适配器进行处理器方法的调用，对处理器方法的调用是通过反射实现的，在调用之前，需要通过反射从请求参数，请求头等探测所有需要的参数，再调用后返回方法结果后，再通过一定规则映射结果到模型和视图对象。这个流程是在注解方法处理器适配器类和相关的支持类实现的，如下类图所示，



图表 4-21

如上图所示，注解方法处理器适配器类实现了处理器适配器接口。在 handle()方法的实现中，使用两个辅助类 Servlet 处理器方法解析器(ServletHandlerMethodResolver)和 Servlet 处理器方法调用器(ServletHandlerMethodInvoker)利用反射的原理调用注解处理器中的处理器方法。在处理器方法调用之前，通过参数注解从 HTTP 请求中提取参数值，在处理器方法调用之后通过注解映射方法的返回值到模型和视图对象，最后返回给派遣器 Servlet 进行视图解析和视图显示。如下流程图所示，



图表 4-22

根据上面流程图显示的顺序，我们将深入的对代码进行剖析，派遣器 Servlet 从缺省注解处理器映射得到了注解控制器后，将控制器传递给注解方法处理器适配器的 handle()方法，handle()方法在进行通用的 HTTP 请求方法检查和设置 HTTP 响应缓存信息后，根据需要对处理器方法进行同步或者非同步的调用，如下代码注释，

```
public ModelAndView handle(HttpServletRequest request, HttpServletResponse response,
Object handler)
        throws Exception {

    // 如果在处理器类型级别声明了Session属性(@SessionAttributes)，为这个处理器使用特殊的缓存时
间，这个特殊的缓存时间通过属性cacheSecondsForSessionAttributeHandlers配置
    if (AnnotationUtils.findAnnotation(handler.getClass(), SessionAttributes.class) !=
null) {
        // 如果使用Session属性管理，缺省情况下，指导用户不使用缓存
        checkAndPrepare(request, response,
this.cacheSecondsForSessionAttributeHandlers, true);
    }
    如果处理器级别没有声明Session属性(@SessionAttributes)，则使用缺省的缓存配置
    else {
        // 缺省的缓存配置是通过属性cacheSeconds配置的
```

```
            checkAndPrepare(request, response, true);
        }

        // 如果配置了Session内同步
        if (this.synchronizeOnSession) {
            // 取得Session对象，如果没有并不创建新的Session
            HttpSession session = request.getSession(false);

            // 如果Session存在
            if (session != null) {
                // 取得Session互斥锁对象，缺省是Session对象自己
                Object mutex = WebUtils.getSessionMutex(session);

                // 同步调用处理器方法
                synchronized (mutex) {
                    return invokeHandlerMethod(request, response, handler);
                }
            }
        }

        // 如果没有配置Session内同步，或者还没有创建Session对象，则直接调用处理器方法，不需要Session内同步
        return invokeHandlerMethod(request, response, handler);
    }

    protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
    HttpServletResponse response, Object handler)
            throws Exception {

        // 通过传入的注解控制器构造一个方法解析器，一个类型的处理器对应一个Servlet处理器方法解析器
        ServletHandlerMethodResolver methodResolver = getMethodResolver(handler);

        // 根据URL Pattern匹配，解析得到注解控制器中能够处理当前请求的处理器方法
        Method handlerMethod = methodResolver.resolveHandlerMethod(request);

        // 构造方法调用器，一个类型的处理器对应一个Servlet处理器方法调用器，处理器方法的调用逻辑就是在这个类中实现的
        ServletHandlerMethodInvoker methodInvoker = new
    ServletHandlerMethodInvoker(methodResolver);

        // 构造Web请求对象，它是一个代理对象，包含请求和响应引用和信息
        ServletWebRequest webRequest = new ServletWebRequest(request, response);

        // 构造空的模型对象，用于方法调用过程中存储必要的数据，状态，结果等等
```

```
    ExtendedModelMap implicitModel = new BindingAwareModelMap();

    // 通过反射调用处理器方法，在这个方法实现中，通过反射和声明在参数上的注解探测得到参数值，通过反射调用处理器方法得到返回值，更多的模型数据通过隐式模型返回
    Object result = methodInvoker.invokeHandlerMethod(handlerMethod, handler, webRequest, implicitModel);

    // 通过方法结果的类型和其上声明的注解，把结果和模型数据映射成为模型和视图对象
    ModelAndView mav = methodInvoker.getModelAndView(handlerMethod, handler.getClass(), result, implicitModel, webRequest);

    // 导出Session属性从模型到Session中，如果是绑定对象，同时导出绑定结果
    methodInvoker.updateModelAttributes(handler, (mav != null ? mav.getModel() : null), implicitModel, webRequest);

    return mav;
}
```

如上代码注释，对于如何解析处理器方法，如何解析参数，如何调用处理器方法以及如果映射返回值等的实现都是封装在处理器方法解析器和处理器方法调用器的实现中的，我们稍后会深入剖析这些逻辑的实现。

注解方法处理器适配器也对处理器适配器接口的另外两个方法进行实现，如下代码注释，

```
public boolean supports(Object handler) {
    // 对于一个Bean对象，只要其中有一个方法声明了请求映射注解(@RequestMapping)，则这个Bean对象是注解控制器
    return getMethodResolver(handler).hasHandlerMethods();
}

public long getLastModified(HttpServletRequest request, Object handler) {
    // 注解方法控制器适配器不支持最后修改操作，这可能是因为注解控制器基本应用到Form的处理过程，不需要支持最后修改操作，最后修改操作更多应用到请求资源上

    // 【问题】下面的两个实现可能会更好，
    //        1. 可以通过反射判断注解控制器是不是已经实现了LastModified接口
    //        2. 也可以扩展一个方法注解，声明为这个注解的方法则用来返回实现返回最后修改的时间

    return -1;
}
```

如上程序注释可见，supports()和 getLastModified()的实现是非常简单的，这里不再详细分析。通过上面的流程图和代码注释，我们也已经大体的了解了通过反射调用处理器方法的总体步骤，现在我们开始深入剖析注解方法处理器适配器是如何实现方法解析，方法调用以及模型结果数据映射的。

如何解析处理器方法呢？

正如注解控制器的名字所示，它是基于注解信息的控制器，这个控制器是一个普通的 Bean,不需要实现任何接口或者继承抽象类。一个注解控制器可能包含一个或者更多的处理器方法，这些处理器方法是用请求映射注解(@RequestMapping)标志的，请求映射注解包含着用于匹配 HTTP 请求的 URI Pattern，请求方法，请求参数，请求头的信息。这些在请求映射中声明的信息会用于匹配 HTTP 请求，如果匹配成功，则会使用匹配的处理器方法处理请求。我们首先分析请求映射注解(@RequestMapping)都包含哪些属性信息。

```
public @interface RequestMapping {

    // 用于匹配查找路径的URI Pattern，HTTP请求的查找路径必须匹配URI Pattern之一
    String[] value() default {};

    // 处理器方法所支持的HTTP方法，这些方法包括 GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE，
HTTP请求方法必须是其中之一
    RequestMethod[] method() default {};

    // HTTP请求必须包含所有的这些参数
    String[] params() default {};

    // HTTP请求必须包含所有的这些头
    String[] headers() default {};

}
```

如上代码所示，声明在一个处理器方法或者处理器类型级别的请求映射注解可以包含 URI Pattern, 请求 方法，请求 参数， 请求头等信息。在匹配的时候，URI Pattern 是最重要的匹配信息，如果没有指定 URI Pattern，则使用其余信息匹配。下面我们分析注解方法处理器适配器是如何使用这些信息匹配一个 HTTP 请求到一个处理器方法的。

首先，注解方法处理器适配器为每一个处理器类型创建一个处理器方法解析器，处理器方法解析器通过反射分析处理器类型，并且取得所有声明了请求映射注解的处理器方法，初始化绑定方法，模型属性方法。如下代码所示，

```
private ServletHandlerMethodResolver(Class<?> handlerType) {
    // 使用父类的初始化方法进行初始化，一个处理器类型对应一个Servlet处理器方法解析器
    init(handlerType);
}

public void init(Class<?> handlerType) {
    // 如果处理器类型是代理类，则获得创建代理类指定的接口，否则使用处理器类型本身，这样可以忽略代理类本
身的方法invoke()
    Class<?>[] handlerTypes =
            Proxy.isProxyClass(handlerType) ? handlerType.getInterfaces() : new
Class<?>[] {handlerType};

    // 遍历获得的每一个类型
```

```
    for (final Class<?> currentHandlerType : handlerTypes) {
        // 遍历每个类型的每个方法
        ReflectionUtils.doWithMethods(currentHandlerType, new
ReflectionUtils.MethodCallback() {
            public void doWith(Method method) {
                // 如果这是接口的方法，取得实现接口的代理类中的对象的方法，这样主要是忽略代理类本身
的方法invoke()
                Method specificMethod = ClassUtils.getMostSpecificMethod(method,
currentHandlerType);

                // 如果是处理器方法，存储处理器方法
                if (isHandlerMethod(method)) {
                    handlerMethods.add(specificMethod);
                }
                // 如果是初始化绑定方法，存储初始化绑定方法
                else if (method.isAnnotationPresent(InitBinder.class)) {
                    initBinderMethods.add(specificMethod);
                }
                // 如果是模型属性方法，存储模型属性方法
                else if (method.isAnnotationPresent(ModelAttribute.class)) {
                    modelAttributeMethods.add(specificMethod);
                }
            }
            // 跳过桥梁方法，桥梁方法是编译器产生，用来解决模板的方法的重载问题的
        }, ReflectionUtils.NON_BRIDGED_METHODS);
    }

    // 取得类型级别的请求映射注解
    this.typeLevelMapping = AnnotationUtils.findAnnotation(handlerType,
RequestMapping.class);

    // 取得类型级别的Session属性注解
    SessionAttributes sessionAttributes =
handlerType.getAnnotation(SessionAttributes.class);

    this.sessionAttributesFound = (sessionAttributes != null);

    // 如果存在类型级别的Session属性注解
    if (this.sessionAttributesFound) {
        // 保存类型级别的Session属性注解声明的属性名字或者类型
        this.sessionAttributeNames.addAll(Arrays.asList(sessionAttributes.value()));
        this.sessionAttributeTypes.addAll(Arrays.asList(sessionAttributes.types()));
    }
}
```

```
protected boolean isHandlerMethod(Method method) {
    // 如果方法上存在请求映射注解，则这个方法是处理器方法
    return AnnotationUtils.findAnnotation(method, RequestMapping.class) != null;
}
```

我们看到，对于一个处理器类型初始化一个处理器方法解析器，处理器方法解析器在解析处理器方法时使用了一个复杂的逻辑决定这些方法中的哪些方法可以处理当前 HTTP 请求。如果多个处理器方法可以处理当前的请求，那么选择最佳匹配的处理器方法。以下详细分析这个工作流程。

在初始化阶段，我们已经存储了所有声明了请求映射注解的处理器方法。现在我们遍历所有的处理器方法，判断是否此方法支持当前的 HTTP 请求。

如果请求映射注解的处理器方法包含 URI Pattern 的信息，那么对于每一个 URI Pattern，则查看是否存在类型级别的 URI Pattern，如果存在，则结合类型级别的 URI Pattern。否则，如果最佳匹配的 URI Pattern 存在，则结合最佳匹配的 URI Pattern。否则单独使用处理器方法级别的 URI Pattern 匹配当前的查找路径。如果 URI Pattern 匹配成功，查看是否 HTTP 请求匹配声明的请求方法，请求参数和请求头。如果这些信息都匹配成功，则添加当前 URI Pattern 到匹配路径集合中，并通过路径匹配对比器对匹配路径集合进行排序，同事标识当前处理器方法为匹配。

如果请求映射注解的处理器方法不包含 URI Pattern 的信息，则只需要查看是否匹配声明的请求方法，请求参数和请求头。如果这些信息匹配，则认为当前处理器方法为匹配。一种特殊情况是，如果请求映射注解中没有声明 HTTP 方法和参数，那么首先使用处理器方法名解析器解析处理器方法名，如果解析的方法名和当前处理器方法相同，则认为当前处理器方法为匹配。缺省的方法名解析器是通过去掉 URI 最后一部分的文件名扩展名得到的。

如果某一个处理器方法匹配，存储这个处理器方法到匹配的处理器方法集合中。如果处理器集合中已经存在一个处理器方法，而且已存处理器方法和现在处理器方法不是同一个处理器方法，那么我们需要解析冲突。在这种情况下，如果没有路径信息，我们需要使用方法名解析器解析最佳处理器方法。处理规则如下，

1. 如果已存处理器方法和当前处理器方法名相同，则使用后解析的方法。
2. 否则，如果解析的方法名和已存处理器方法同名，继续使用已存处理器方法。如果解析的方法名和当前处理器方法同名，则使用当前处理器方法。
3. 如果解析的方法名既不等于当前处理器方法也不等于已存的处理器方法，则抛出异常，终止处理。

根据上面的逻辑分析，最终如果有一个或者多个处理器方法匹配当前 HTTP 请求，则通过请求映射信息对比器找到最佳匹配的处理器方法。如下代码所示，

```
public Method resolveHandlerMethod(HttpServletRequest request) throws ServletException {
    // 取得查找路径，查找路径是URI去掉应用程序环境部分和Servlet环境部分的剩余部分
    String lookupPath = urlPathHelper.getLookupPathForRequest(request);

    // 取得URL对比器，如果有多个URI Pattern匹配，用于对比找到最佳匹配
    Comparator<String> pathComparator = pathMatcher.getPatternComparator(lookupPath);

    // 用于存储请求映射信息到处理器方法的映射
```

```
    Map<RequestMappingInfo, Method> targetHandlerMethods = new
LinkedHashMap<RequestMappingInfo, Method>();
    Set<String> allowedMethods = new LinkedHashSet<String>(7);
    String resolvedMethodName = null;

    // 遍历每一个处理器方法
    for (Method handlerMethod : getHandlerMethods()) {
        RequestMappingInfo mappingInfo = new RequestMappingInfo();

        // 取得处理器方法的请求映射注解
        RequestMapping mapping = AnnotationUtils.findAnnotation(handlerMethod,
RequestMapping.class);

        // 取得声明在请求映射注解上的路径信息，也就是URI Pattern
        mappingInfo.paths = mapping.value();

        // 如果处理器类型级别没有声明处理器映射或者方法级别处理器映射包含的HTTP方法信息不同于类型级
别处理器映射包含的HTTP方法信息，则使用方法级别的声明信息
        if (!hasTypeLevelMapping() || !Arrays.equals(mapping.method(),
getTypeLevelMapping().method())) {
            mappingInfo.methods = mapping.method();
        }

        // 如果处理器类型级别没有声明处理器映射或者方法级别处理器映射包含的HTTP参数信息不同于类型级
别处理器映射包含的HTTP参数信息，则使用方法级别的声明信息
        if (!hasTypeLevelMapping() || !Arrays.equals(mapping.params(),
getTypeLevelMapping().params())) {
            mappingInfo.params = mapping.params();
        }

        // 如果处理器类型级别没有声明处理器映射或者方法级别处理器映射包含的HTTP头信息不同于类型级别
处理器映射包含的HTTP头信息，则使用方法级别的声明信息
        if (!hasTypeLevelMapping() || !Arrays.equals(mapping.headers(),
getTypeLevelMapping().headers())) {
            mappingInfo.headers = mapping.headers();
        }
        boolean match = false;

        // 如果URI Pattern存在，则先匹配URI
        if (mappingInfo.paths.length > 0) {
            List<String> matchedPaths = new
ArrayList<String>(mappingInfo.paths.length);
            for (String methodLevelPattern : mappingInfo.paths) {
                // 取得匹配的URI Pattern
```

```
                String matchedPattern = getMatchedPattern(methodLevelPattern,
lookupPath, request);

                // 如果取得的URI pattern不为空，则说明这个URI Pattern匹配成功
                if (matchedPattern != null) {
                    // 匹配HTTP方法，参数和头等信息
                    if (mappingInfo.matches(request)) {
                        match = true;
                        // 记录匹配的URI Pattern
                        matchedPaths.add(matchedPattern);
                    }
                    else {
                        // 否则如果匹配失败，记录支持的HTTP方法，用来构造提示用户信息
                        for (RequestMethod requestMethod : mappingInfo.methods) {
                            allowedMethods.add(requestMethod.toString());
                        }
                        break;
                    }
                }
            }
            // 排序匹配的URI Pattern集合
            Collections.sort(matchedPaths, pathComparator);

            // 设置匹配的URI Pattern信息到请求映射信息中
            mappingInfo.matchedPaths = matchedPaths;
        }
        // 如果URI Pattern不存在，则只需要匹配HTTP方法，参数和请求头
        else {
            // 则只需要匹配HTTP方法，参数和请求头
            match = mappingInfo.matches(request);

            // 如果没有声明请求方法和请求参数（可能仅仅请求头匹配），则使用方法名解析器解析处理器方
法，如果方法名解析器解析的方法和当前方法不同名，则匹配失败
            if (match && mappingInfo.methods.length == 0 && mappingInfo.params.length ==
0 &&
                    resolvedMethodName != null
&& !resolvedMethodName.equals(handlerMethod.getName())) {
                match = false;
            }
            else {
                // 否则如果匹配失败，否则记录支持的HTTP方法，用来构造提示用户信息
                for (RequestMethod requestMethod : mappingInfo.methods) {
                    allowedMethods.add(requestMethod.toString());
                }
```

```
            }
        }

        // 如果这个处理器方法和当前的HTTP请求匹配成功
        if (match) {
            // 存储请求映射信息到当前处理器方法的映射，如果有已存的处理器方法，则取得已存的处理器方法，这意味着对同一个请求映射信息配置了两个处理器方法
            Method oldMappedMethod = targetHandlerMethods.put(mappingInfo, handlerMethod);

            // 如果已存的处理器方法不同于当前处理器方法
            if (oldMappedMethod != null && oldMappedMethod != handlerMethod) {
                // 如果请求映射注解中没有URI Pattern声明信息，则使用方法名解析器解析冲突
                if (methodNameResolver != null && mappingInfo.paths.length == 0) {
                    // 如果已存处理器方法和当前处理器方法名不同
                    if (!oldMappedMethod.getName().equals(handlerMethod.getName())) {
                        if (resolvedMethodName == null) {
                            // 使用方法名解析器解析方法名
                            resolvedMethodName = methodNameResolver.getHandlerMethodName(request);
                        }

                        // 如果解析方法名不等于已存方法名，则抛弃已存方法
                        if (!resolvedMethodName.equals(oldMappedMethod.getName())) {
                            oldMappedMethod = null;
                        }
                        // 如果解析方法名不等于当前方法名
                        if (!resolvedMethodName.equals(handlerMethod.getName())) {
                            // 如果已存方法名没有被抛弃，也就是解析方法名等于已存方法名
                            if (oldMappedMethod != null) {
                                // 恢复并仍然使用已存方法名
                                targetHandlerMethods.put(mappingInfo, oldMappedMethod);
                                oldMappedMethod = null;
                            }
                            else {
                                // 解析方法名既不等于已存方法名也不等于当前方法名
                                targetHandlerMethods.remove(mappingInfo);
                            }
                        }
                    }
                }

                // 解析方法名既不等于已存方法名也不等于当前方法名
```

```
                if (oldMappedMethod != null) {
                    throw new IllegalStateException(
                            "Ambiguous handler methods mapped for HTTP path '" +
lookupPath + "': {" +
                                    oldMappedMethod + ", " + handlerMethod +
                                    "}. If you intend to handle the same path in
multiple methods, then factor " +
                                    "them out into a dedicated handler class with that
path mapped at the type level!");
                }
            }
        }
    }

    // 如果有一个或者多个处理器方法匹配
    if (!targetHandlerMethods.isEmpty()) {
        // 取得所有匹配的请求映射信息
        List<RequestMappingInfo> matches = new
ArrayList<RequestMappingInfo>(targetHandlerMethods.keySet());

        // 获得可以对请求映射信息排列的对比器
        RequestMappingInfoComparator requestMappingInfoComparator =
                new RequestMappingInfoComparator(pathComparator);

        // 排序
        Collections.sort(matches, requestMappingInfoComparator);

        // 第一个为最佳匹配处理器方法
        RequestMappingInfo bestMappingMatch = matches.get(0);

        // 取得最佳匹配处理器方法的最佳匹配路径
        String bestMatchedPath = bestMappingMatch.bestMatchedPath();

        if (bestMatchedPath != null) {
            // 如果存在最佳匹配路径，则提取模板变量
            extractHandlerMethodUriTemplates(bestMatchedPath, lookupPath, request);
        }

        // 返回处理器方法
        return targetHandlerMethods.get(bestMappingMatch);
    }
    // 如果没有处理器方法匹配
    else {
        // 当URI Pattern匹配或者没有声明URI Pattern的时候，如果HTTP方法，请求参数或者头信息不匹
```

```
配，则提示HTTP方法不支持，并且提示那些HTTP方法支持
        if (!allowedMethods.isEmpty()) {
            throw new HttpRequestMethodNotSupportedException(request.getMethod(),
                    StringUtils.toStringArray(allowedMethods));
        }
        // 否则提示用户没有请求处理器方法
        else {
            throw new NoSuchRequestHandlingMethodException(lookupPath,
request.getMethod(),
                    request.getParameterMap());
        }
    }
}

private String getMatchedPattern(String methodLevelPattern, String lookupPath,
HttpServletRequest request) {
    // 如果类型级别声明了请求映射注解的URI Pattern
    if (hasTypeLevelMapping() && (!ObjectUtils.isEmpty(getTypeLevelMapping().value())))
{
        // 取得类型级别声明的请求映射注解的URI Pattern
        String[] typeLevelPatterns = getTypeLevelMapping().value();
        // 遍历每一个URI Pattern
        for (String typeLevelPattern : typeLevelPatterns) {
            // 补充URI前缀
            if (!typeLevelPattern.startsWith("/")) {
                typeLevelPattern = "/" + typeLevelPattern;
            }

            // 结合方法级别的URL Pattern
            String combinedPattern = pathMatcher.combine(typeLevelPattern,
methodLevelPattern);

            // 如果查找路径匹配结合的URL Pattern
            if (isPathMatchInternal(combinedPattern, lookupPath)) {
                // 返回匹配结合的URL Pattern
                return combinedPattern;
            }
        }
        // 如果无法匹配，则返回空
        return null;
    }

    // 取得最佳匹配URI Pattern
    String bestMatchingPattern = (String)
```

```
request.getAttribute(HandlerMapping.BEST_MATCHING_PATTERN_ATTRIBUTE);
    // 如果存在最佳匹配URL Pattern
    if (StringUtils.hasText(bestMatchingPattern)) {
        // 结合最佳匹配URL Pattern和方法级别的URI Pattern
        String combinedPattern = pathMatcher.combine(bestMatchingPattern,
methodLevelPattern);
        // 如果结合URI Pattern匹配，则使用这个结合的Pattern
        if (!combinedPattern.equals(bestMatchingPattern) &&
                (isPathMatchInternal(combinedPattern, lookupPath))) {
            return combinedPattern;
        }
    }
    // 【问题】问什么我们需要使用最佳匹配Pattern进行结合

    // 如果类型级别没有声明请求映射注解的URI Pattern，则使用方法级别的URI Pattern进行匹配
    if (isPathMatchInternal(methodLevelPattern, lookupPath)) {
        return methodLevelPattern;
    }
    return null;
}

private boolean isPathMatchInternal(String pattern, String lookupPath) {
    // 如果查找路径和URI Pattern相等或者匹配
    if (pattern.equals(lookupPath) || pathMatcher.match(pattern, lookupPath)) {
        return true;
    }

    // 增加后缀进行匹配
    boolean hasSuffix = pattern.indexOf('.') != -1;
    if (!hasSuffix && pathMatcher.match(pattern + ".*", lookupPath)) {
        return true;
    }

    // 增加前缀进行匹配
    boolean endsWithSlash = pattern.endsWith("/");
    if (!endsWithSlash && pathMatcher.match(pattern + "/", lookupPath)) {
        return true;
    }
    return false;
}

private void extractHandlerMethodUriTemplates(String mappedPath,
        String lookupPath,
        HttpServletRequest request) {
```

```
    Map<String, String> variables = null;
    boolean hasSuffix = (mappedPath.indexOf('.') != -1);

    // 如果路径不存在.字符(例如.do)，则使用模糊后缀匹配，增加.*进行匹配，意味着可以以任何字符串结尾
    if (!hasSuffix && pathMatcher.match(mappedPath + ".*", lookupPath)) {
        String realPath = mappedPath + ".*";
        if (pathMatcher.match(realPath, lookupPath)) {
            variables = pathMatcher.extractUriTemplateVariables(realPath, lookupPath);
        }
    }

    // 使用模糊前缀进行匹配，如果URI Pattern没有以/开始，则表示可以以任意路径开始
    if (variables == null && !mappedPath.startsWith("/")) {
        String realPath = "/**/" + mappedPath;
        if (pathMatcher.match(realPath, lookupPath)) {
            variables = pathMatcher.extractUriTemplateVariables(realPath, lookupPath);
        }
        else {
            // 如果路径没有匹配成功则增加模糊后缀继续匹配
            realPath = realPath + ".*";
            if (pathMatcher.match(realPath, lookupPath)) {
                variables = pathMatcher.extractUriTemplateVariables(realPath,
lookupPath);
            }
        }
    }

    // 导出得到的模板变量到请求属性中
    if (!CollectionUtils.isEmpty(variables)) {
        Map<String, String> typeVariables =
                (Map<String, String>)
request.getAttribute(HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE);
        if (typeVariables != null) {
            variables.putAll(typeVariables);
        }
        request.setAttribute(HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE,
variables);
    }
}
```

如何解析处理器方法参数呢？

注解方法处理器适配器是通过请求映射注解解析处理器方法的。解析得到了处理器方法，在调用处理器方法之前我们必须首先解析所有的处理器方法参数。处理器方法参数也是通过各种注解标记的，不同的注解包含着信息指导注解方法处

理器适配器从不同的数据源取得数据。下面我们详细分析，处理器方法参数所支持的所有注解。

最常用的应用在处理器方法参数上的注解是请求参数注解(@RequestParam)。 请求参数注解指导注解方法处理器适配器通过参数名字找到请求参数值，并且赋值给当前方法参数。如下代码注释，

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestParam {

    // 请求参数名字
    String value() default "";

    // 请求参数是不是必须的
    boolean required() default true;

    // 如果提供了缺省值，则请求参数自动变成非必须的
    String defaultValue() default ValueConstants.DEFAULT_NONE;

}
```

请求头注解(@RequestHeader)指导注解方法处理器适配器通过头名字找到请求头的值，并且赋值给当前方法参数。如下代码注释，

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestHeader {

    // 请求头名字
    String value() default "";

    // 请求头是不是必须的
    boolean required() default true;

    // 如果提供了缺省值，则请求头自动变成非必须的
    String defaultValue() default ValueConstants.DEFAULT_NONE;

}
```

请求体注解(@RequestBody)指导注解方法处理器适配器通过消息转换器将请求体转换成 Java 对象作为当前方法参数的值。但是请求注解并没有声明任何属性信息，它只是个标志，指导注解方法处理器适配器为当前参数解析请求体。如下代码注释，

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RequestBody {

}
```

Cookie 值注解(@CookieValue) 指导注解方法处理器适配器通过 Cookie 名字找到 Cookie 的值，并且赋值给当前方法参数。如下代码注释，

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CookieValue {
    // Cookie名字
    String value() default "";

    // Cookie值是不是必须的
    boolean required() default true;

    // 如果提供了缺省值，则Cookie值自动变成非必须的
    String defaultValue() default ValueConstants.DEFAULT_NONE;

}
```

路径变量注解(@PathVariable)指导注解方法处理器适配器通过路径变量名字找到路径变量的值(路径变量通常被称为模板变量)，并且赋值给当前方法参数。如下代码注释，

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface PathVariable {

   // 路径变量的名字
    String value() default "";

}
```

模型属性注解(@ModelAttribute) 指导注解方法处理器适配器通过模型属性名字找到模型属性的值，并且赋值给当前方法参数。模型属性注解也可以用于声明在方法上，这种情况下把方法的返回值作为模型数据值放入隐式模型中。如下代码注释，

```
// 可以生命在参数上，作为参数的输入值，也可以生命在方法上，方法的返回值会被加入到隐式模型中
@Target({ElementType.PARAMETER, ElementType.METHOD})
```

```
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ModelAttribute {

    // 模型属性的名字
    String value() default "";

}
```

对于一个处理器方法参数，只能声明上面的注解中的一个，或者不声明注解。如果声明了上面注解中的一个，则根据注解解析处理器方法参数的值。如果一个处理器方法参数没有声明任何注解，则查看是否配置有客户化Web参数解析器，如果存在则使用客户化Web参数解析器进行解析。如果没有配置客户化Web参数解析器或者客户化Web参数解析器不能解析当前参数，则判断是不是标准的WebRequest类型，如果是则返回标准的WebRequest类型。如果上面的情况仍然不能解析参数值，则使用默认值。

如果通过上面的流程仍然没有解析出参数值，则做如下判断，

- 如果参数是模型类型或者Map类型，则使用当前隐式模型。
- 如果参数是Session状态类型，则使用当前的Session状态，它表明当前Session完成或者没有完成。
- 如果参数是HTTP实体，则解析HTTP请求体作为HTTP实体。
- 如果是错误对象，则必须前一个参数是绑定对象。否则，抛出异常，终止处理。下面将介绍绑定对象。
- 如果是简单数据类型，则初始化参数名为空。

最后，如果参数值仍然没有被解析，那么这个参数是一个需要绑定的Bean。如果这个方法参数声明了模型属性或者这个模型属性是Session属性，则从模型中或者Session中取得当前值，否则根据类型创建一个对象，再使用HTTP请求参数进行绑定操作。

此外，值注解(@Value)和校验注解(@Valid)是两个辅助的注解，并不直接绑定方法参数到任何HTTP请求信息。用来声明方法参数缺省值或者校验的。

值注解(@Value)用于给参数值指定缺省值，如果没有指定任何注解或者指定的注解所表达的数据为空，而且注解中没有指定缺省值，则使用这个指定的缺省值。如下代码注释，

```
@Retention(RetentionPolicy.RUNTIME)
// 可以应用到参数，字段和方法上
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
public @interface Value {
    // 指定缺省值
    String value();

}
```

校验注解(@Valid) 指导注解方法处理器适配器对当前的属性进行校验。这个注解没有任何属性，仅仅是一个标志。

```
@Target({ METHOD, FIELD, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface Valid {
}
```

整个流程实现在处理器方法解析器类中。如下代码所示，

```
private Object[] resolveHandlerArguments(Method handlerMethod, Object handler,
        NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws Exception {

    // 根据参数类型个数创建参数值数组
    Class[] paramTypes = handlerMethod.getParameterTypes();
    Object[] args = new Object[paramTypes.length];

    // 遍历所有参数解析参数值
    for (int i = 0; i < args.length; i++) {
        // 构造一个方法参数对象，方法参数对象包含着方法参数的所以信息
        MethodParameter methodParam = new MethodParameter(handlerMethod, i);

        // 使用参数名解析器解析参数名，这个实现需要读取字节码文件，才能解析到方法参数名
        methodParam.initParameterNameDiscovery(this.parameterNameDiscoverer);

        // 解析参数类型，如果是模板参数则解析原始类型
        GenericTypeResolver.resolveParameterType(methodParam, handler.getClass());

        String paramName = null;
        String headerName = null;
        boolean requestBodyFound = false;
        String cookieName = null;
        String pathVarName = null;
        String attrName = null;
        boolean required = false;
        String defaultValue = null;
        boolean validate = false;
        int annotationsFound = 0;
        Annotation[] paramAnns = methodParam.getParameterAnnotations();

        // 遍历当前参数的所有注解
        for (Annotation paramAnn : paramAnns) {
            // 请求参数注解
            if (RequestParam.class.isInstance(paramAnn)) {
                RequestParam requestParam = (RequestParam) paramAnn;
                paramName = requestParam.value();
                required = requestParam.required();
```

```
                    defaultValue =
parseDefaultValueAttribute(requestParam.defaultValue());
                    annotationsFound++;
                }
                // 请求头注解
                else if (RequestHeader.class.isInstance(paramAnn)) {
                    RequestHeader requestHeader = (RequestHeader) paramAnn;
                    headerName = requestHeader.value();
                    required = requestHeader.required();
                    defaultValue =
parseDefaultValueAttribute(requestHeader.defaultValue());
                    annotationsFound++;
                }
                // 请求体注解
                else if (RequestBody.class.isInstance(paramAnn)) {
                    requestBodyFound = true;
                    annotationsFound++;
                }
                // Cookie值注解
                else if (CookieValue.class.isInstance(paramAnn)) {
                    CookieValue cookieValue = (CookieValue) paramAnn;
                    cookieName = cookieValue.value();
                    required = cookieValue.required();
                    defaultValue = parseDefaultValueAttribute(cookieValue.defaultValue());
                    annotationsFound++;
                }
                // 路径变量注解
                else if (PathVariable.class.isInstance(paramAnn)) {
                    PathVariable pathVar = (PathVariable) paramAnn;
                    pathVarName = pathVar.value();
                    annotationsFound++;
                }
                // 模型属性注解
                else if (ModelAttribute.class.isInstance(paramAnn)) {
                    ModelAttribute attr = (ModelAttribute) paramAnn;
                    attrName = attr.value();
                    annotationsFound++;
                }
                // 缺省值注解
                else if (Value.class.isInstance(paramAnn)) {
                    defaultValue = ((Value) paramAnn).value();
                }
                // 校验注解
                else if ("Valid".equals(paramAnn.annotationType().getSimpleName())) {
```

```
                    validate = true;
                }
            }

            // 不允许一个参数上有多个注解，但是@Value和@Valid除外
            if (annotationsFound > 1) {
                throw new IllegalStateException("Handler parameter annotations are exclusive
choices - " +
                        "do not specify more than one such annotation on the same parameter:
" + handlerMethod);
            }

            // 如果参数上没有注解
            if (annotationsFound == 0) {
                // 解析通用参数
                Object argValue = resolveCommonArgument(methodParam, webRequest);

                // 如果解析通用参数成功，则使用解析值
                if (argValue != WebArgumentResolver.UNRESOLVED) {
                    args[i] = argValue;
                }
                // 如果解析通用参数不成功，则使用缺省值
                else if (defaultValue != null) {
                    args[i] = resolveDefaultValue(defaultValue);
                }
                // 如果解析通用参数不成功，而且没有缺省值，则解析特殊类型的对象
                else {
                    // 【问题】 如果参数声明类型过于细节化，则会出现类型转换异常，例如，你声明了一个客
户化的模型类实现，这种情况下尽量生命更通用的类型，例如，Model, Map, SessionStatus等等

                    Class paramType = methodParam.getParameterType();
                    // 如果参数是Model或者Map的子类或者实现类，则使用隐式模型
                    if (Model.class.isAssignableFrom(paramType) ||
Map.class.isAssignableFrom(paramType)) {
                        args[i] = implicitModel;
                    }
                    // 如果参数是Session状态的子类或者实现类，则使用当前的Session状态对象
                    else if (SessionStatus.class.isAssignableFrom(paramType)) {
                        args[i] = this.sessionStatus;
                    }
                    // 如果参数是HTTP实体的子类或者实现类，则使用消息转换器将请求体转换作为HTTP实体对
象，它类似于对请求体注解(@RequestBody)解析
                    else if (HttpEntity.class.isAssignableFrom(paramType)) {
                        args[i] = resolveHttpEntityRequest(methodParam, webRequest);
```

```
                    }
                    // 如果参数是错误的子类或者实现类，那么前一个参数一定是绑定对象，当绑定对象处理是，这个对象会被掠过，所以，对于一个单独出现的错误对象，则是异常情况
                    else if (Errors.class.isAssignableFrom(paramType)) {
                        throw new IllegalStateException("Errors/BindingResult argument declared " +
                                "without preceding model attribute. Check your handler method signature!");
                    }
                    // 对于没有解析的简单属性，默认为参数名为空的参数，解析的值也是空
                    else if (BeanUtils.isSimpleProperty(paramType)) {
                        paramName = "";
                    }
                    // 否则，是一个没有模型属性注解的绑定对象，把属性名设置为空串后，后面进行绑定和解析
                    else {
                        attrName = "";
                    }
                }
            }

            // 解析请求参数值
            if (paramName != null) {
                args[i] = resolveRequestParam(paramName, required, defaultValue, methodParam, webRequest, handler);
            }
            // 解析请求头值
            else if (headerName != null) {
                args[i] = resolveRequestHeader(headerName, required, defaultValue, methodParam, webRequest, handler);
            }
            // 解析请求体值，这是通过HTTP消息转换器完成的
            else if (requestBodyFound) {
                args[i] = resolveRequestBody(methodParam, webRequest, handler);
            }
            // 解析Cookie值
            else if (cookieName != null) {
                args[i] = resolveCookieValue(cookieName, required, defaultValue, methodParam, webRequest, handler);
            }
            // 解析路径变量值(模板变量值)
            else if (pathVarName != null) {
                args[i] = resolvePathVariable(pathVarName, methodParam, webRequest, handler);
            }
```

```
        // 如果属性值不为空，则是一个绑定对象，可能声明了模型属性注解，也可能没有声明
        else if (attrName != null) {
            // 如果声明了模型属性注解，或者此属性是Session属性，则可以从模型或者Session中取得，否则创建一个新的属性对象，然后进行绑定
            WebDataBinder binder =
                    resolveModelAttribute(attrName, methodParam, implicitModel,
webRequest, handler);
            boolean assignBindingResult = (args.length > i + 1 &&
Errors.class.isAssignableFrom(paramTypes[i + 1]));
            if (binder.getTarget() != null) {
                doBind(binder, webRequest, validate, !assignBindingResult);
            }

            // 取得绑定的参数值
            args[i] = binder.getTarget();

            // 将绑定结果赋值给下一个参数值，对于一个绑定参数，下一个参数一定是绑定结果或者错误对象
            if (assignBindingResult) {
                args[i + 1] = binder.getBindingResult();
                i++;
            }

            // 将绑定结果放入到隐式模型中
            implicitModel.putAll(binder.getBindingResult().getModel());
        }
    }

    return args;
}

protected Object resolveCommonArgument(MethodParameter methodParameter, NativeWebRequest
webRequest)
throws Exception {

    // 首先使用客户化的Web参数解析器进行解析
    if (this.customArgumentResolvers != null) {
    for (WebArgumentResolver argumentResolver : this.customArgumentResolvers) {
        Object value = argumentResolver.resolveArgument(methodParameter, webRequest);
        if (value != WebArgumentResolver.UNRESOLVED) {
            // 存在一个客户化的Web参数解析器解析到参数值，则返回此解析的参数值
            return value;
        }
    }
    }
```

```
    // 取得参数类型
    Class paramType = methodParameter.getParameterType();

    // 解析标准类型
    Object value = resolveStandardArgument(paramType, webRequest);

    // 如果解析得到标准类型值，但是标准类型值不是参数类型或者其子类对象，则抛出异常，这表明参数应该声明作为更通用的类型
    if (value != WebArgumentResolver.UNRESOLVED
&& !ClassUtils.isAssignableValue(paramType, value)) {
        throw new IllegalStateException("Standard argument type [" + paramType.getName()
+
                "] resolved to incompatible value of type [" + (value != null ?
value.getClass() : null) +
                "]. Consider declaring the argument type in a less specific fashion.");
    }
    return value;
}

protected Object resolveStandardArgument(Class parameterType, NativeWebRequest webRequest)
throws Exception {
    // 如果期待Web请求类的子类，则返回NativeWebRequest参数
    if (WebRequest.class.isAssignableFrom(parameterType)) {
    return webRequest;
    }
    return WebArgumentResolver.UNRESOLVED;
}
```

我们理解，对于不同的参数注解，注解方法处理器适配器从不同的数据源提取数据，例如，请求参数，请求头或者请求体等，下面我们具体分析对于不同的参数注解，注解方法处理器适配器是如何解析方法参数值的。

下面是通过 HTTP 方法参数解析处理器方法参数值的(@RequestParam)。如下代码注释，

```
private Object resolveRequestParam(String paramName, boolean required, String
defaultValue,
        MethodParameter methodParam, NativeWebRequest webRequest, Object
handlerForInitBinderCall)
        throws Exception {

    Class<?> paramType = methodParam.getParameterType();

    // 如果参数是映射类型，这种情况下请求参数注解没有必要指定任何参数名字，返回所有参数名值对映射
    if (Map.class.isAssignableFrom(paramType)) {
```

```
            // 解析参数映射，这个参数映射包含所有参数名值对
            return resolveRequestParamMap((Class<? extends Map>) paramType, webRequest);
        }

        // 如果参数名是空，请求参数注解没有指定任何参数名，则使用参数名解析器解析的参数名，这是通过解析方法
参数名实现的
        if (paramName.length() == 0) {
            paramName = getRequiredParameterName(methodParam);
        }

        Object paramValue = null;
        // 如果是多部请求体，则解析文件参数
        if (webRequest.getNativeRequest() instanceof MultipartRequest) {
            paramValue = ((MultipartRequest)
webRequest.getNativeRequest()).getFile(paramName);
        }

        if (paramValue == null) {
            // 如果是普通请求，则取得参数值
            String[] paramValues = webRequest.getParameterValues(paramName);
            // 如果参数类型不是数组，但是返回了参数数组值，则只使用数组中的第一个元素
            if (paramValues != null && !paramType.isArray()) {
                paramValue = (paramValues.length == 1 ? paramValues[0] : paramValues);
            }
            // 否则，使用数组赋值
            else {
                paramValue = paramValues;
            }
        }

        if (paramValue == null) {
            // 如果请求参数注解中制定了缺省值，或者通过值注解(@value)中制定了缺省值
            if (StringUtils.hasText(defaultValue)) {
                // 使用指定的缺省值
                paramValue = resolveDefaultValue(defaultValue);
            }
            else if (required) {
                // 如果请求参数注解中制定了是必须的参数，则抛出异常，终止处理
                raiseMissingParameterException(paramName, paramType);
            }

            // 检查参数值的合法性
            paramValue = checkValue(paramName, paramValue, paramType);
        }
```

```
    // 通过绑定器转换参数值到需要的类型
    WebDataBinder binder = createBinder(webRequest, null, paramName);
    initBinder(handlerForInitBinderCall, paramName, binder, webRequest);
    return binder.convertIfNecessary(paramValue, paramType, methodParam);
}

private Map resolveRequestParamMap(Class<? extends Map> mapType, NativeWebRequest
webRequest) {
    Map<String, String[]> parameterMap = webRequest.getParameterMap();

    // 如果是多值映射
    if (MultiValueMap.class.isAssignableFrom(mapType)) {
        MultiValueMap<String, String> result = new LinkedMultiValueMap<String,
String>(parameterMap.size());
        // 遍历所有的HTTP请求参数
        for (Map.Entry<String, String[]> entry : parameterMap.entrySet()) {
            // 遍历每一个HTTP请求参数的值
            for (String value : entry.getValue()) {
                // 加多个值到同一个参数名中，所以成为多值映射
                result.add(entry.getKey(), value);
            }
        }
        return result;
    }
    // 如果是单值映射，对于HTTP数组参数，则仅仅使用第一个元素
    else {
        Map<String, String> result = new LinkedHashMap<String,
String>(parameterMap.size());
        for (Map.Entry<String, String[]> entry : parameterMap.entrySet()) {
            if (entry.getValue().length > 0) {
                result.put(entry.getKey(), entry.getValue()[0]);
            }
        }
        return result;
    }
}


private String getRequiredParameterName(MethodParameter methodParam) {
    String name = methodParam.getParameterName();
    // 如果不能解析处理器方法参数名，则抛出异常
    if (name == null) {
        throw new IllegalStateException(
```

```
                    "No parameter name specified for argument of type [" +
methodParam.getParameterType().getName() +
                            "], and no parameter name information found in class file
either.");
        }
        return name;
    }

    private Object checkValue(String name, Object value, Class paramType) {
        // 把应用空值制定为布尔false, 如果对于私有类型解析了空值，则抛出异常，终止处理
        if (value == null) {
            if (boolean.class.equals(paramType)) {
                return Boolean.FALSE;
            }
            else if (paramType.isPrimitive()) {
                throw new IllegalStateException("Optional " + paramType + " parameter '" +
name +
                        "' is not present but cannot be translated into a null value due to
being declared as a " +
                        "primitive type. Consider declaring it as object wrapper for the
corresponding primitive type.");
            }
        }
        return value;
    }
```

解析 HTTP 请求头作为处理器方法参数值的实现和解析 HTTP 请求参数作为处理器方法参数值的实现相似(@RequestHeader)。这里不再进行代码注释。

下面是通过 HTTP 请求体解析处理器方法参数值的(@RequestBody)。如下代码注释，

```
    protected Object resolveRequestBody(MethodParameter methodParam, NativeWebRequest
webRequest, Object handler)
            throws Exception {

        // 创建HTTP请求的代理对象，可以从这个对象取得请求体的流对象,取得方法参数类型，然后使用消息转换器进
行解析
        return readWithMessageConverters(methodParam, createHttpInputMessage(webRequest),
methodParam.getParameterType());
    }

    private Object readWithMessageConverters(MethodParameter methodParam, HttpInputMessage
inputMessage, Class paramType)
            throws Exception {
```

```
    // 确定HTTP请求的内容类型
    MediaType contentType = inputMessage.getHeaders().getContentType();

    // 如果没有指定HTTP请求的内容类型，抛出不支持的HTTP媒体类型异常
    if (contentType == null) {
        StringBuilder builder = new
StringBuilder(ClassUtils.getShortName(methodParam.getParameterType()));
        String paramName = methodParam.getParameterName();
        if (paramName != null) {
            builder.append(' ');
            builder.append(paramName);
        }
        throw new HttpMediaTypeNotSupportedException(
                "Cannot extract parameter (" + builder.toString() + "): no Content-Type
found");
    }

    List<MediaType> allSupportedMediaTypes = new ArrayList<MediaType>();
    if (this.messageConverters != null) {
        // 遍历所有的消息转换器
        for (HttpMessageConverter<?> messageConverter : this.messageConverters) {
            // 保存支持的媒体类型，如果解析请求体失败，则提示用户那些媒体类型是支持的
            allSupportedMediaTypes.addAll(messageConverter.getSupportedMediaTypes());

            // 判断是否有消息转换器可以解析HTTP请求体
            if (messageConverter.canRead(paramType, contentType)) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Reading [" + paramType.getName() + "] as \"" +
contentType
                            +"\" using [" + messageConverter + "]");
                }
                // 如果存在，则返回转换的对象引用
                return messageConverter.read(paramType, inputMessage);
            }
        }
    }

    // 如果解析失败，则抛出异常
    throw new HttpMediaTypeNotSupportedException(contentType, allSupportedMediaTypes);
}
```

事实上，有另外一个方法可以用来解析请求体作为处理器方法参数，那就是通过声明请求参数作为 HttpEntity 类型，HttpEntity 类型可以指定模板类型进行泛化。对比请求体注解，HttpEntity 参数不需要注解，注解方法处理器适配器是通

过类型来判断是否需要解析方法体的。如下代码注释，

```
private HttpEntity resolveHttpEntityRequest(MethodParameter methodParam,
NativeWebRequest webRequest)
throws Exception {

    // 创建HTTP请求的代理对象
    HttpInputMessage inputMessage = createHttpInputMessage(webRequest);

    // 返回声明的参数类型，例如， 如果参数是HttpEntity<Resource>, 这个方法返回Reource.class, 如
果参数是HttpEntity<Resource[].class>, 这个方法返回Resource[].class
    Class<?> paramType = getHttpEntityType(methodParam);

    // 使用方法转换器进行解析
    Object body = readWithMessageConverters(methodParam, inputMessage, paramType);

    // 解析请求体后，构造HTTP实体对象，因为真正需要返回的是HTTP实体对象，而不是参数类型对象
    return new HttpEntity<Object>(body, inputMessage.getHeaders());
}

private Class<?> getHttpEntityType(MethodParameter methodParam) {
    // 校验参数类型是HTTP实体类型或者子类
    Assert.isAssignable(HttpEntity.class, methodParam.getParameterType());

    // 取得模板参数化类型
    ParameterizedType type = (ParameterizedType) methodParam.getGenericParameterType();

    // 校验模板参数化类型只有一个参数类型
    if (type.getActualTypeArguments().length == 1) {
        Type typeArgument = type.getActualTypeArguments()[0];
        // 如果参数类型是单值，则直接返回
        if (typeArgument instanceof Class) {
            return (Class<?>) typeArgument;
        }
        // 如果参数类型是数组，则通过构造一个空数组返回数组类型
        else if (typeArgument instanceof GenericArrayType) {
            Type componentType = ((GenericArrayType)
typeArgument).getGenericComponentType();
            if (componentType instanceof Class) {
                // Surely, there should be a nicer way to do this
                Object array = Array.newInstance((Class<?>) componentType, 0);
                return array.getClass();
            }
        }
```

```
    }

    // 如果模板参数化类型没有参数或者有多个参数，则抛出异常，终止处理
    throw new IllegalArgumentException(
            "HttpEntity parameter (" + methodParam.getParameterName() + ") is not
parameterized");

}
```

我们知道，请求体注解(@RequestBody)和 HTTP 实体(HttpEntity)的实现都是通过消息转换器进行解析 HTTP 请求体作为处理器方法参数的。消息转换器的设计和处理器适配器十分相似，消息转换器有一个方法判断是否一个消息转换器可以转换某个类型的参数，当然，还有另外一个方法用于事实上转换请求体到一个指定类型的对象。后面我们将有一小节讨论 HTTP 消息转换器的实现体系结构。

下面是通过 Cookie 值解析处理器方法参数值的(@CookieValue)。如下代码注释，

```
private Object resolveCookieValue(String cookieName, boolean required, String
defaultValue,
        MethodParameter methodParam, NativeWebRequest webRequest, Object
handlerForInitBinderCall)
        throws Exception {

    // 取得参数类型
    Class<?> paramType = methodParam.getParameterType();

    // 如果没有指定Cookie名，则默认使用处理器方法参数名
    if (cookieName.length() == 0) {
        cookieName = getRequiredParameterName(methodParam);
    }

    // 通过Cookie名解析Cookie值
    Object cookieValue = resolveCookieValue(cookieName, paramType, webRequest);

    // 如果解析失败
    if (cookieValue == null) {
        // 如果Cookie值注解中指定了缺省值，或者使用值注解(@Value)指定了缺省值，则使用缺省值
        if (StringUtils.hasText(defaultValue)) {
            cookieValue = resolveDefaultValue(defaultValue);
        }
        // 如果是必须的处理器方法值，则抛出异常，终止处理
        else if (required) {
            raiseMissingCookieException(cookieName, paramType);
        }
```

```
            // 校验解析的Cookie值
            cookieValue = checkValue(cookieName, cookieValue, paramType);
        }

        // 使用数据绑定进行数据类型转换
        WebDataBinder binder = createBinder(webRequest, null, cookieName);
        initBinder(handlerForInitBinderCall, cookieName, binder, webRequest);
        return binder.convertIfNecessary(cookieValue, paramType, methodParam);
    }

    //这是声明在处理器调用器中的占位符方法，子类必须改写其实现
    protected Object resolveCookieValue(String cookieName, Class paramType, NativeWebRequest
    webRequest)
            throws Exception {

        throw new UnsupportedOperationException("@CookieValue not supported");
    }

    //子类改写父类的占位符，从请求中解析Cookie值
    protected Object resolveCookieValue(String cookieName, Class paramType, NativeWebRequest
    webRequest)
    throws Exception {

        // 取得HTTP请求对象
        HttpServletRequest servletRequest = (HttpServletRequest)
    webRequest.getNativeRequest();

        // 取Cookie值
        Cookie cookieValue = WebUtils.getCookie(servletRequest, cookieName);

        // 如果处理器方法参数是Cookie类型，则直接返回
        if (Cookie.class.isAssignableFrom(paramType)) {
            return cookieValue;
        }
        // 否则返回字符串值
        else if (cookieValue != null) {
            return cookieValue.getValue();
        }
        else {
            return null;
        }
    }
```

下面是通过模板变量解析处理器方法参数值的(@PathVariant)。如下代码注释，

```
private Object resolvePathVariable(String pathVarName, MethodParameter methodParam,
        NativeWebRequest webRequest, Object handlerForInitBinderCall) throws Exception
{

    // 取得参数类型
    Class<?> paramType = methodParam.getParameterType();

    // 如果没有制定路径变量名，则使用参数名本身
    if (pathVarName.length() == 0) {
        pathVarName = getRequiredParameterName(methodParam);
    }

    // 解析路径变量值，这些值是在请求映射实现中保存的
    String pathVarValue = resolvePathVariable(pathVarName, paramType, webRequest);

    // 使用数据绑定转换到期望的类型对象
    WebDataBinder binder = createBinder(webRequest, null, pathVarName);
    initBinder(handlerForInitBinderCall, pathVarName, binder, webRequest);
    return binder.convertIfNecessary(pathVarValue, paramType, methodParam);
}

//占位符方法
protected String resolvePathVariable(String pathVarName, Class paramType, NativeWebRequest
webRequest)
        throws Exception {

    throw new UnsupportedOperationException("@PathVariable not supported");
}

//以上占位符方法的实现
protected String resolvePathVariable(String pathVarName, Class paramType, NativeWebRequest
webRequest)
throws Exception {
    // 取得HTTP请求
    HttpServletRequest servletRequest = (HttpServletRequest)
webRequest.getNativeRequest();

    // 取得路径变量（模板变量），这些变量是在请求映射的实现中保存的
    Map<String, String> uriTemplateVariables =
            (Map<String, String>)
servletRequest.getAttribute(HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE);

    // 失败找到模板变量,所有模板变量都是必须的
```

```
    if (uriTemplateVariables == null || !uriTemplateVariables.containsKey(pathVarName))
{
        throw new IllegalStateException(
                "Could not find @PathVariable [" + pathVarName + "] in @RequestMapping");
    }

    // 返回模板变量值
    return uriTemplateVariables.get(pathVarName);
}
```

最后我们分析一下注解方法处理器适配器是如何绑定领域模型对象，初始化领域模型对象，和管理领域模型对象的。

当一个参数上没有声明任何注解，而且参数没有缺省值并且不是一个简单的属性，也就是说，它是一个领域对象模型对象，也就是一个 Bean 对象。那么则需要数据绑定对象把请求参数数据绑定到 Bean 的属性上。请看解析参数主流程的代码，这个片段的代码如下注释所示，

```
// 如果领域对象模型Bean存在在隐式模型中，则取得领域对象模型Bean，否则，根据领域对象模型类型，实例一个对象
WebDataBinder binder =
        resolveModelAttribute(attrName, methodParam, implicitModel, webRequest,
handler);

// 如果有下一个参数，而且下一个参数是错误类型的子类，则同时复制绑定结果
boolean assignBindingResult = (args.length > i + 1 &&
Errors.class.isAssignableFrom(paramTypes[i + 1]));

// 如果将要绑定的领域对象模型Bean不为空，则绑定Web参数到对象模型中
if (binder.getTarget() != null) {
    doBind(binder, webRequest, validate, !assignBindingResult);
}

// 取得绑定过后的结果做为参数
args[i] = binder.getTarget();

// 如果下一个参数是绑定结果对象，则把绑定结果赋值给下一个参数
if (assignBindingResult) {
    args[i + 1] = binder.getBindingResult();
    i++;
}

// 把绑定结果放入隐式模型中
implicitModel.putAll(binder.getBindingResult().getModel());
```

首先，它在隐式模型中查找领域对象模型Bean，因为这个领域对象模型可能是被标志有模型属性注解(@ModelAttribute)的方法返回的。如果隐式模型中不包含领域对象模型Bean，而且当前领域对象模型Bean是一个Session属性，则在Session中取得领域对象模型Bean。如果仍然不能解析领域对象模型Bean，则通过反射实例化一个全新的类型进行绑定。如下代码注释，

```
private WebDataBinder resolveModelAttribute(String attrName, MethodParameter methodParam,
        ExtendedModelMap implicitModel, NativeWebRequest webRequest, Object handler)
throws Exception {

    // 如果此参数没有生命模型属性注解，则此参数没有属性名，使用类型自动生成的名字
    String name = attrName;
    if ("".equals(name)) {
        name = Conventions.getVariableNameForParameter(methodParam);
    }

    // 取得参数类型
    Class<?> paramType = methodParam.getParameterType();
    Object bindObject;

    // 如果隐式模型中存在此领域对象模型Bean，则使用它
    if (implicitModel.containsKey(name)) {
        bindObject = implicitModel.get(name);
    }
    // 如果是Session属性，则使用Session中的Bean
    else if (this.methodResolver.isSessionAttribute(name, paramType)) {
        bindObject = this.sessionAttributeStore.retrieveAttribute(webRequest, name);
        if (bindObject == null) {
            raiseSessionRequiredException("Session attribute '" + name + "' required -
not found in session");
        }
    }
    // 如果隐式模型和Session中都没有此领域对象模型Bean,则实例化一个新的
    else {
        bindObject = BeanUtils.instantiateClass(paramType);
    }

    // 创建绑定对象
    WebDataBinder binder = createBinder(webRequest, bindObject, name);

    // 初始化绑定，首先使用客户化的绑定初始化器进行初始化，这个初始化器是所有处理器共享的，然后，再使用
处理器生命初始化绑定方法(@InitBinder)进行初始化，这些初始化是处理器指定的
    initBinder(handler, name, binder, webRequest);
    return binder;
}
```

```
protected WebDataBinder createBinder(NativeWebRequest webRequest, Object target, String
objectName)
throws Exception {
    // 创建一个Web请求数据绑定，这个类库扩展了数据绑定类，实现了绑定HTTP请求参数到Bean对象
    return new WebRequestDataBinder(target, objectName);
}


protected void initBinder(Object handler, String attrName, WebDataBinder binder,
NativeWebRequest webRequest)
throws Exception {

    // 首先使用配置的初始化绑定器进行初始化,绑定初始化器有一个缺省的实现
ConfigurableWebBindingInitializer，这个实现支持许多配置选项
    if (this.bindingInitializer != null) {
        this.bindingInitializer.initBinder(binder, webRequest);
    }

    // 取得生命在处理器中的初始化绑定方法，也就是标记有初始化绑定器注解(@InitBinder)的方法
    if (handler != null) {
        Set<Method> initBinderMethods = this.methodResolver.getInitBinderMethods();

    // 如果存在初始化绑定方法
    if (!initBinderMethods.isEmpty()) {
        boolean debug = logger.isDebugEnabled();
        // 遍历调用每一个方法
        for (Method initBinderMethod : initBinderMethods) {
            // 找到桥梁方法
            Method methodToInvoke =
BridgeMethodResolver.findBridgedMethod(initBinderMethod);

            // 初始化绑定器注解可以声明应用在哪些属性上
            String[] targetNames = AnnotationUtils.findAnnotation(methodToInvoke,
InitBinder.class).value();

            // 如果初始化绑定注解声明中包含此属性名或者没有生命任何属性名
           if (targetNames.length == 0 || Arrays.asList(targetNames).contains(attrName))
{
                // 解析初始化绑定器的所需要的方法
                Object[] initBinderArgs =
                        resolveInitBinderArguments(handler, methodToInvoke, binder,
webRequest);
                if (debug) {
```

```
                        logger.debug("Invoking init-binder method: " + methodToInvoke);
                    }

                    // 设置方法可以存取属性
                    ReflectionUtils.makeAccessible(methodToInvoke);

                    // 调用方法，并且禁止放回任何返回值
                    Object returnValue = methodToInvoke.invoke(handler, initBinderArgs);
                    if (returnValue != null) {
                        throw new IllegalStateException(
                                "InitBinder methods must not have a return value: " +
methodToInvoke);
                    }
                }
            }
        }
}
```

现在我们看到，一个绑定对象初始化完毕后，这包括应用在所有处理器上的绑定初始化器的初始化，和声明在处理器内部的绑定初始化方法，然后开始做真正的绑定操作。这个操作就是把HTTP请求参数设置到绑定的目标对象中。如下代码所示，

```
private void doBind(WebDataBinder binder, NativeWebRequest webRequest, boolean validate,
boolean failOnErrors)
        throws Exception {
    // 代理到事实的绑定方法中
    doBind(binder, webRequest);

    // 如果设置了校验，则校验绑定，绑定结果会赋值给下一个Errors对象，并且会放入到隐式模型中给试图进行
展示
    if (validate) {
        binder.validate();
    }
    if (failOnErrors && binder.getBindingResult().hasErrors()) {
        throw new BindException(binder.getBindingResult());
    }
}

protected void doBind(WebDataBinder binder, NativeWebRequest webRequest) throws Exception
{
    // 既然创建的是Web请求数据绑定，则调用Web请求数据绑定的绑定方法
    ((WebRequestDataBinder) binder).bind(webRequest);
}
```

Web数据绑定是数据绑定的子类，主要是分析HTTP请求参数包含的数据，这包括多部文件HTTP请求，然后，将收集的参数信息赋值给绑定的目标对象。如下代码注释，

```
public class WebRequestDataBinder extends WebDataBinder {
    // 提供API方法绑定目标对象到HTTP请求对象
    public void bind(WebRequest request) {
        // 从HTTP请求中提取请求参数，这些包括写在URL中的参数和普通POST体的参数
        MutablePropertyValues mpvs = new
MutablePropertyValues(request.getParameterMap());

        // 如果是多部文件HTTP请求，则也需要绑定多部文件HTTP请求体的文件参数，这些不包含URL中的参数
        if (request instanceof NativeWebRequest) {
            MultipartRequest multipartRequest = ((NativeWebRequest)
request).getNativeRequest(MultipartRequest.class);
            if (multipartRequest != null) {
                bindMultipartFiles(multipartRequest.getFileMap(), mpvs);
            }
        }

        doBind(mpvs);
    }
}

public class WebDataBinder extends DataBinder {
    protected void doBind(MutablePropertyValues mpvs) {
        // 检查是否存在缺省字段，缺省字段指定的值是在此字段没有值的时候应用的，这个缺省字段是以!开头
的
        checkFieldDefaults(mpvs);
        // 检查是否存在标记字段，标记字段指对那些没有指定的请求参数，需要清空原来的已有的值
        checkFieldMarkers(mpvs);
        super.doBind(mpvs);
    }

    protected void bindMultipartFiles(Map<String, MultipartFile> multipartFiles,
MutablePropertyValues mpvs) {
        for (Map.Entry<String, MultipartFile> entry : multipartFiles.entrySet()) {
            String key = entry.getKey();
            MultipartFile value = entry.getValue();
            // 可以配置是否绑定空的文件参数
            if (isBindEmptyMultipartFiles() || !value.isEmpty()) {
                mpvs.add(key, value);
            }
        }
    }
```

```
}

public class DataBinder implements PropertyEditorRegistry, TypeConverter {

    protected void doBind(MutablePropertyValues mpvs) {
        // 可以配置哪些字段是允许的，哪些字段是必须的
        checkAllowedFields(mpvs);
        checkRequiredFields(mpvs);

        // 事实上赋值属性
        applyPropertyValues(mpvs);
    }

    protected void applyPropertyValues(MutablePropertyValues mpvs) {
        try {
            //getPropertyAccessor()返回的是BeanWrapper实现， BeanWrapper是用來存取Bean属性的封装类
            getPropertyAccessor().setPropertyValues(mpvs, isIgnoreUnknownFields(), isIgnoreInvalidFields());
        }
        catch (PropertyBatchUpdateException ex) {
            // 使用绑定错误处理器处理绑定错误结果
            for (PropertyAccessException pae : ex.getPropertyAccessExceptions()) {
                getBindingErrorProcessor().processPropertyAccessException(pae, getInternalBindingResult());
            }
        }
    }
}
```

数据绑定，属性校验，属性编辑器的实现是一个复杂的话题，属于 Spring 框架的核心实现，我们不在这里进行分析和代码注释。

如何调用处理器方法呢？

在得到方法参数以前，注解方法处理器适配器首先处理模型属性方法，初始化隐式模型，如果存在 Session 属性，导入 Session 属性，然后使用解析得到的方法参数调用处理器方法。如下代码所示，

```
public final Object invokeHandlerMethod(Method handlerMethod, Object handler,
        NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws Exception {

    // 取得真正的处理器方法，而不是桥梁方法，桥梁方法是编译器产生的方法
    Method handlerMethodToInvoke = BridgeMethodResolver.findBridgedMethod(handlerMethod);
```

```
    try {
        boolean debug = logger.isDebugEnabled();

        // 如果有Session属性，则提取Session属性到隐式模型属性中
        for (String attrName : this.methodResolver.getActualSessionAttributeNames()) {
            Object attrValue = this.sessionAttributeStore.retrieveAttribute(webRequest,
attrName);
            if (attrValue != null) {
                implicitModel.addAttribute(attrName, attrValue);
            }
        }

        // 遍历所有的模型属性方法，将其返回的模型属性放入隐式模型中
        for (Method attributeMethod : this.methodResolver.getModelAttributeMethods()) {
            // 取得模型属性方法的真正的处理器方法
            Method attributeMethodToInvoke =
BridgeMethodResolver.findBridgedMethod(attributeMethod);

            // 解析参数，这个实现过程和处理器方法参数的解析过程相似，不再进行代码注释
            Object[] args = resolveHandlerArguments(attributeMethodToInvoke, handler,
webRequest, implicitModel);
            if (debug) {
                logger.debug("Invoking model attribute method: " +
attributeMethodToInvoke);
            }

            // 如果模型属性方法指定的属性已经存在在隐式模型中，则忽略此模型属性
            String attrName = AnnotationUtils.findAnnotation(attributeMethodToInvoke,
ModelAttribute.class).value();
            if (!"".equals(attrName) && implicitModel.containsAttribute(attrName)) {
                continue;
            }

            // 调用模型属性方法，得到属性值
            Object attrValue = doInvokeMethod(attributeMethodToInvoke, handler, args);

            // 如果没有制定模型属性名，则使用模型类型创建一个唯一的名字
            if ("".equals(attrName)) {
                Class resolvedType =

    GenericTypeResolver.resolveReturnType(attributeMethodToInvoke,
handler.getClass());
                attrName =
```

```
    Conventions.getVariableNameForReturnType(attributeMethodToInvoke, resolvedType,
attrValue);
                }

                // 添加属性名值对到隐式模型中
                if (!implicitModel.containsAttribute(attrName)) {
                    implicitModel.addAttribute(attrName, attrValue);
                }
            }

            // 通过注解，缺省值等注解解析处理器参数
            Object[] args = resolveHandlerArguments(handlerMethodToInvoke, handler,
webRequest, implicitModel);
            if (debug) {
                logger.debug("Invoking request handler method: " + handlerMethodToInvoke);
            }

            return doInvokeMethod(handlerMethodToInvoke, handler, args);
        }
        catch (IllegalStateException ex) {
            // Throw exception with full handler method context...
            throw new HandlerMethodInvocationException(handlerMethodToInvoke, ex);
        }
    }
```

```
private Object doInvokeMethod(Method method, Object target, Object[] args) throws Exception
{
    // 使方法可见
    ReflectionUtils.makeAccessible(method);
    try {
        // 调用方法
        return method.invoke(target, args);
    }
    catch (InvocationTargetException ex) {
        ReflectionUtils.rethrowException(ex.getTargetException());
    }
    throw new IllegalStateException("Should never get here");
}
```

如果映射处理器方法返回值和隐式模型到模型和视图对象呢？

注解方法处理器适配器调用一个处理器方法之后，得到了处理器方法的返回值，它根据返回值的类型解析模型和视图对

象，如果声明了消息响应体，它将使用消息转换器转换返回值到 HTTP 响应体。我们已经分析了消息转换器的实现体系结构，这里不再进行分析。如下代码所示，

```
public ModelAndView getModelAndView(Method handlerMethod, Class handlerType, Object
returnValue,
        ExtendedModelMap implicitModel, ServletWebRequest webRequest) throws Exception
{

    // 如果声明了响应状态注解
    ResponseStatus responseStatusAnn = AnnotationUtils.findAnnotation(handlerMethod,
ResponseStatus.class);
    if (responseStatusAnn != null) {
        // 从响应状态注解中解析设置的HTTP响应状态代码
        HttpStatus responseStatus = responseStatusAnn.value();

        // to be picked up by the RedirectView
        webRequest.getRequest().setAttribute(View.RESPONSE_STATUS_ATTRIBUTE,
responseStatus);

        // 设置响应状态代码
        webRequest.getResponse().setStatus(responseStatus.value());
        responseArgumentUsed = true;
    }

    // Invoke custom resolvers if present...
    if (customModelAndViewResolvers != null) {
        for (ModelAndViewResolver mavResolver : customModelAndViewResolvers) {
            ModelAndView mav = mavResolver
                    .resolveModelAndView(handlerMethod, handlerType, returnValue,
implicitModel, webRequest);
            if (mav != ModelAndViewResolver.UNRESOLVED) {
                return mav;
            }
        }
    }

    // 如果参数类型是HTTP实体类型
    if (returnValue instanceof HttpEntity) {
        handleHttpEntityResponse((HttpEntity<?>) returnValue, webRequest);
        return null;
    }
    //如果声明了请求体注解，则使用消息转换器解析返回值到相应体
    else if (AnnotationUtils.findAnnotation(handlerMethod, ResponseBody.class) != null)
{
```

```
            handleResponseBody(returnValue, webRequest);
            return null;
        }
        //如果返回值已经是模型和视图类型，则合并隐式模型的值
        else if (returnValue instanceof ModelAndView) {
            ModelAndView mav = (ModelAndView) returnValue;
            mav.getModelMap().mergeAttributes(implicitModel);
            return mav;
        }    // 如果返回值是模型，则合并它和隐式模型，并且构造一个模型和视图对象
        else if (returnValue instanceof Model) {
            return new ModelAndView().addAllObjects(implicitModel).addAllObjects(((Model)
returnValue).asMap());
        }
        // 如果返回值是视图，则合并它和隐式模型，并且构造一个模型和视图对象
        else if (returnValue instanceof View) {
            return new ModelAndView((View) returnValue).addAllObjects(implicitModel);
        }
        // 如果是模型属性方法，则加返回值到模型中，并且合并隐式模型
        else if (AnnotationUtils.findAnnotation(handlerMethod, ModelAttribute.class) != null)
{
            addReturnValueAsModelAttribute(handlerMethod, handlerType, returnValue,
implicitModel);
            return new ModelAndView().addAllObjects(implicitModel);
        }
        // 如果返回值是映射，则合并它和隐式模型，构造模型和视图对象
        else if (returnValue instanceof Map) {
            return new ModelAndView().addAllObjects(implicitModel).addAllObjects((Map)
returnValue);
        }
        // 如果返回值是字符串，它是视图的逻辑名，构造一个模型和视图对象
        else if (returnValue instanceof String) {
            return new ModelAndView((String) returnValue).addAllObjects(implicitModel);
        }
        // 如果返回值为空，
        else if (returnValue == null) {
            // 如果设置了响应状态， 或者没有修改状态， 返回空值
            if (this.responseArgumentUsed || webRequest.isNotModified()) {
                return null;
            }
            else {
                // Assuming view name translation...
                return new ModelAndView().addAllObjects(implicitModel);
            }
        }
```

```
    // 如果是一个Bean值，加入到模型中，并且合并隐式模型，返回模型和视图对象
    else if (!BeanUtils.isSimpleProperty(returnValue.getClass())) {
        // Assume a single model attribute...
        addReturnValueAsModelAttribute(handlerMethod, handlerType, returnValue,
implicitModel);
        return new ModelAndView().addAllObjects(implicitModel);
    }
    else {
        throw new IllegalArgumentException("Invalid handler method return value: " +
returnValue);
    }
}

private void handleResponseBody(Object returnValue, ServletWebRequest webRequest)
        throws Exception {
    if (returnValue == null) {
        return;
    }
    HttpInputMessage inputMessage = createHttpInputMessage(webRequest);
    HttpOutputMessage outputMessage = createHttpOutputMessage(webRequest);

    // 使用消息转换器写返回值
    writeWithMessageConverters(returnValue, inputMessage, outputMessage);
}

private void handleHttpEntityResponse(HttpEntity<?> responseEntity, ServletWebRequest
webRequest)
        throws Exception {
    if (responseEntity == null) {
        return;
    }
    HttpInputMessage inputMessage = createHttpInputMessage(webRequest);
    HttpOutputMessage outputMessage = createHttpOutputMessage(webRequest);

    // 如果返回值是相应体，则包含相应状态，需要写入HTTP响应里
    if (responseEntity instanceof ResponseEntity && outputMessage instanceof
ServerHttpResponse) {
        ((ServerHttpResponse)outputMessage).setStatusCode(((ResponseEntity)
responseEntity).getStatusCode());
    }

    // 把返回的HTTP实体的头信息写入HTTP相应里
    HttpHeaders entityHeaders = responseEntity.getHeaders();
    if (!entityHeaders.isEmpty()) {
```

```
            outputMessage.getHeaders().putAll(entityHeaders);
        }
        Object body = responseEntity.getBody();
        if (body != null) {
            // 转换返回值并且写入HTTP相应体里
            writeWithMessageConverters(body, inputMessage, outputMessage);
        }
    }

    @SuppressWarnings("unchecked")
    private void writeWithMessageConverters(Object returnValue,
            HttpInputMessage inputMessage, HttpOutputMessage outputMessage)
            throws IOException, HttpMediaTypeNotAcceptableException {
        List<MediaType> acceptedMediaTypes = inputMessage.getHeaders().getAccept();

        // 如果HTTP请求没有制定接受的媒体类型，默认接受所有媒体类型
        if (acceptedMediaTypes.isEmpty()) {
            acceptedMediaTypes = Collections.singletonList(MediaType.ALL);
        }
        MediaType.sortByQualityValue(acceptedMediaTypes);
        Class<?> returnValueType = returnValue.getClass();
        List<MediaType> allSupportedMediaTypes = new ArrayList<MediaType>();
        if (getMessageConverters() != null) {
            // 使用HTTP消息转换器转换返回值作为HTTP相应体
            for (MediaType acceptedMediaType : acceptedMediaTypes) {
                for (HttpMessageConverter messageConverter : getMessageConverters()) {
                    if (messageConverter.canWrite(returnValueType, acceptedMediaType)) {
                        messageConverter.write(returnValue, acceptedMediaType,
outputMessage);
                        if (logger.isDebugEnabled()) {
                            MediaType contentType =
outputMessage.getHeaders().getContentType();
                            if (contentType == null) {
                                contentType = acceptedMediaType;
                            }
                            logger.debug("Written [" + returnValue + "] as \"" + contentType
+
                                    "\" using [" + messageConverter + "]");
                        }
                        this.responseArgumentUsed = true;
                        return;
                    }
                }
            }
```

```
            for (HttpMessageConverter messageConverter : messageConverters) {
                allSupportedMediaTypes.addAll(messageConverter.getSupportedMediaTypes());
            }
        }
        throw new HttpMediaTypeNotAcceptableException(allSupportedMediaTypes);
    }
```

如何更新模型数据呢？

最后，如果存在 Session 属性，则导出 Session 属性从模型到 Session 中，如下代码所示，

```
    public final void updateModelAttributes(Object handler, Map<String, Object> mavModel,
            ExtendedModelMap implicitModel, NativeWebRequest webRequest) throws Exception {

        // 如果Session状态标识成结束，则清除Session属性，默认情况下Session始终是未完成的
        if (this.methodResolver.hasSessionAttributes() && this.sessionStatus.isComplete()) {
            for (String attrName : this.methodResolver.getActualSessionAttributeNames()) {
                this.sessionAttributeStore.cleanupAttribute(webRequest, attrName);
            }
        }

        // Expose model attributes as session attributes, if required.
        // Expose BindingResults for all attributes, making custom editors available.
        Map<String, Object> model = (mavModel != null ? mavModel : implicitModel);
        for (String attrName : new HashSet<String>(model.keySet())) {
            Object attrValue = model.get(attrName);
            boolean isSessionAttr =
                    this.methodResolver.isSessionAttribute(attrName, (attrValue != null ?
    attrValue.getClass() : null));
            // 如果Session属性，而且Session状态是未完成的，则导出Session属性从模型到Session中
            if (isSessionAttr && !this.sessionStatus.isComplete()) {
                this.sessionAttributeStore.storeAttribute(webRequest, attrName,
    attrValue);
            }

            // 同时导出绑定结构到Session中
            if (!attrName.startsWith(BindingResult.MODEL_KEY_PREFIX) &&
                    (isSessionAttr || isBindingCandidate(attrValue))) {
                String bindingResultKey = BindingResult.MODEL_KEY_PREFIX + attrName;
                if (mavModel != null && !model.containsKey(bindingResultKey)) {
                    WebDataBinder binder = createBinder(webRequest, attrValue, attrName);
                    initBinder(handler, attrName, binder, webRequest);
                    mavModel.put(bindingResultKey, binder.getBindingResult());
                }
```

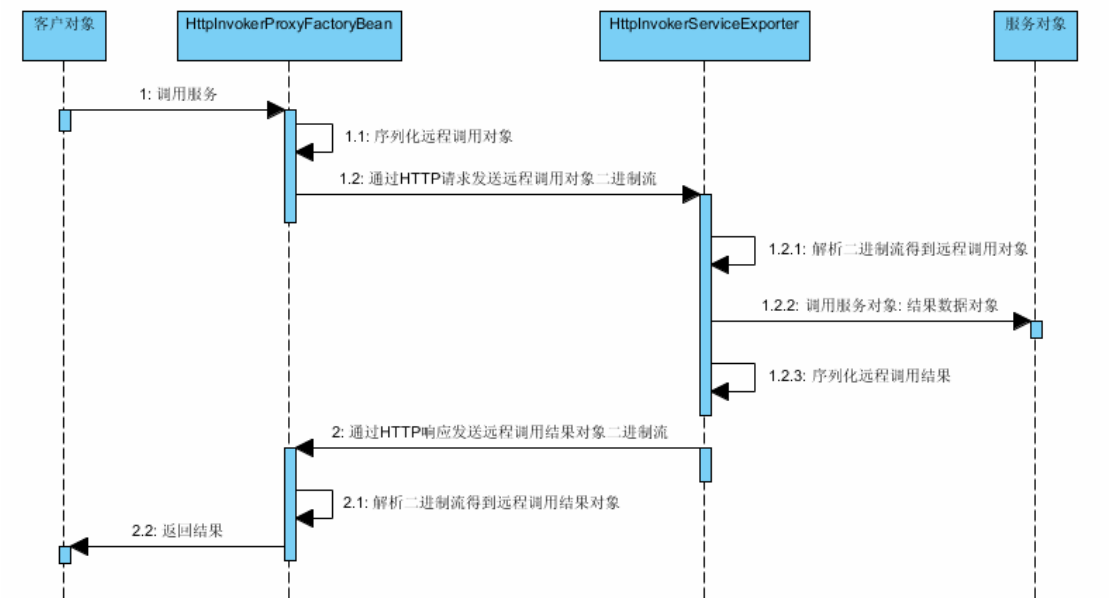
```
        }
    }
}
```

基于简单控制器流程的实现和基于注解控制器流程的实现是两个用来处理 HTTP 请求的 Spring Web MVC 流程的实现。他们使用了不同的方式实现了相同的功能和流程，简单控制器流程是经典的实现，而基于注解控制器流程的实现是从版本 2.5 新引入的，其简单易用又不失于功能强大，使基于注解控制器流程的实现成为被推荐使用的流程。

4.2.1.3基于HTTP请求处理器流程的实现

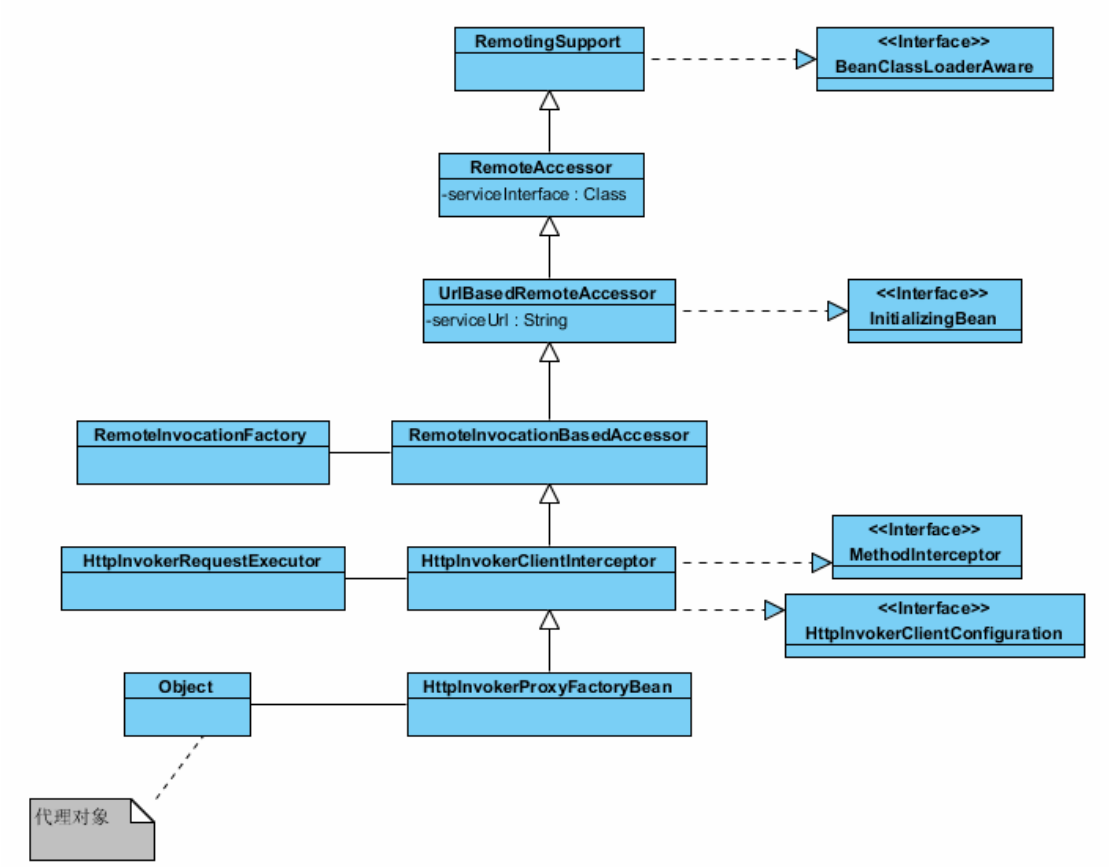
这一节，我们将分析 Spring Web MVC 中的另外一个流程，它是用来支持基于 HTTP 协议的远程调用的流程。它并不是应用到一个简单的 HTTP 请求，它使用 Servlet 通过 HTTP 协议导出一个服务，服务的客户端可以通过 HTTP 协议使用和调用此服务。Spring Web MVC 支持基于 HTTP 协议的三个类型的远程调用，基于 Burlap 的远程调用，基于 Hessian 的远程调用和基于序列化的远程调用。这里我们只分析基于序列化的远程调用，在后面小节中我们将分析所有远程调用流程的架构实现。

既然这是一个远程调用流程的实现，那么必然存在客户端和服务器端的实现，客户端是通过一个工厂 Bean 的实现导出服务器接口的代理，代理的实现把对方法的调用封装成远程调用对象并且进行序列化，通过配置的 HTTP URL 把序列化的远程调用对象通过 HTTP 请求体传入到服务器，服务器解析序列化的远程调用对象，然后通过反射调用远程调用对象里包含的方法，最后发送返回值到客户端。如下流程图所示，



图表 4-23

首先我们深入分析基于 HTTP 请求处理器流程的远程调用的客户端的实现体系结构。如下类图所示，



图表 4-24

客户端的实现是通过 AOP 代理拦截方法调用，然后，将传入的方法参数和正在调用的方法封装到远程调用对象并且序列化为字节流。最后，通过 HTTP 调用器请求执行器写入到远程主机的 HTTP 服务中的。下面我们根据流程的先后顺序对实现代码进行分析，如下代码注释，

当在客户端的 Spring 环境中配置一个 HTTP 服务，首先我们需要声明一个 HTTP 唤起器代理工厂 Bean(HttpInvokerProxyFactoryBean)，然后指定需要代理的接口类和服务器端服务的 URL。如下 Spring 配置文件所示，

```
<bean                                                    id="httpInvokerProxy"
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

HTTP 唤起器代理工厂 Bean 实现了工厂 Bean 接口，当它被初始化后进行 Bean 连接的时候，它会提供一个代理对象。如下代码注释，

```
// 这是一个工厂Bean，返回一个代理对象，用于拦截对于提供的接口方法的调用，然后，实现远程调用
public class HttpInvokerProxyFactoryBean extends HttpInvokerClientInterceptor
        implements FactoryBean<Object> {
```

```
    // 返回的代理对象
    private Object serviceProxy;


    // Spring环境中的Bean的占位符方法，初始化后属性后，构造代理对象
    @Override
    public void afterPropertiesSet() {
        super.afterPropertiesSet();

        // 将要代理的接口为空，抛出异常，停止处理
        if (getServiceInterface() == null) {
            throw new IllegalArgumentException("Property 'serviceInterface' is
required");
        }

        // 使用AOP代理工厂创建代理对象，对这个代理对象的所有方法调用最后都会被拦截，然后调用到超类对
MethodInterceptor的实现方法invoke()
        // 如果将要代理类是接口，则使用JDK的动态代理，如果将要代理类是类类型，则需要使用CGLIB产生子
类创建代理
        this.serviceProxy = new ProxyFactory(getServiceInterface(),
this).getProxy(getBeanClassLoader());
    }


    // 返回代理对象
    public Object getObject() {
        return this.serviceProxy;
    }

    // 返回代理类型
    public Class<?> getObjectType() {
        return getServiceInterface();
    }

    // 既然是服务，则是单例模式
    public boolean isSingleton() {
        return true;
    }

}
```

产生了代理对象后，客户端会根据业务需要调用代理对象的服务方法。当调用任何方法的时候，代理机制会统一的调用到超类 HTTP 唤起器客户拦截器(HttpInvokerClientInterceptor)中

对方法拦截器接口(MethodInterceptor)中的 invoke 方法的是实现。如下代码所示，

```
// 当调用一个代理对象的方法，AOP代理机制会将调用信息，包括调用的方法，参数等封装成MethodInvocation对象传递给MethodInterceptor的实现里
public Object invoke(MethodInvocation methodInvocation) throws Throwable {
    // 如果正在调用toString()方法，则返回服务URL信息
    if (AopUtils.isToStringMethod(methodInvocation.getMethod())) {
        return "HTTP invoker proxy for service URL [" + getServiceUrl() + "]";
    }

    // 创建远程调用对象，远程调用对象是方法调用对象的一个简单封装
    RemoteInvocation invocation = createRemoteInvocation(methodInvocation);
    RemoteInvocationResult result = null;
    try {
        // 使用HTTP调用器请求执行器调用远程的HTTP服务，并且获得返回结果
        result = executeRequest(invocation, methodInvocation);
    }
    // 这些异常是在调用过程中产生的，例如网络连接错误，返回值解析错误等等
    catch (Throwable ex) {
        throw convertHttpInvokerAccessException(ex);
    }

    try {
        // 远程调用结果可能包含异常信息，如果有异常发生在服务器中，则需要在客户端中重现
        return recreateRemoteInvocationResult(result);
    }
    // 这些异常是发生在服务器中的异常，异常被传回了客户端，这里需要重现异常
    catch (Throwable ex) {
        // 如果结果中是唤起目标异常，则直接抛出异常
        if (result.hasInvocationTargetException()) {
            throw ex;
        }
        // 【问题】为什么唤起目标异常不需要疯涨？

        // 否则抛出封装的异常
        else {
            throw new RemoteInvocationFailureException("Invocation of method [" +
methodInvocation.getMethod() +
                    "] failed in HTTP invoker remote service at [" + getServiceUrl() +
"]", ex);
        }
    }
}
```

```
//这些异常是发生在客户端和服务器端
protected RemoteAccessException convertHttpInvokerAccessException(Throwable ex) {
    // HTTP连接错误
    if (ex instanceof ConnectException) {
        throw new RemoteConnectFailureException(
                "Could not connect to HTTP invoker remote service at [" + getServiceUrl()
+ "]", ex);
    }
    // 是否有返回值对象是不可解析的类型
    else if (ex instanceof ClassNotFoundException || ex instanceof NoClassDefFoundError
||
            ex instanceof InvalidClassException) {
        throw new RemoteAccessException(
                "Could not deserialize result from HTTP invoker remote service [" +
getServiceUrl() + "]", ex);
    }
    // 其他未知调用过程中的异常
    else {
        throw new RemoteAccessException(
            "Could not access HTTP invoker remote service at [" + getServiceUrl() + "]",
ex);
    }
}

public class RemoteInvocationResult implements Serializable {
    // 如果存在异常则重现异常，否则返回远程结果
    public Object recreate() throws Throwable {
        // 如果异常存在
        if (this.exception != null) {
            Throwable exToThrow = this.exception;

            // 如果是唤起目标异常，则取得发生在方法调用时真正的异常
            if (this.exception instanceof InvocationTargetException) {
                exToThrow = ((InvocationTargetException)
this.exception).getTargetException();
            }

            // 添加客户端异常堆栈
            RemoteInvocationUtils.fillInClientStackTraceIfPossible(exToThrow);

            // 抛出异常
            throw exToThrow;
        }
        // 否则返回服务器端结果
```

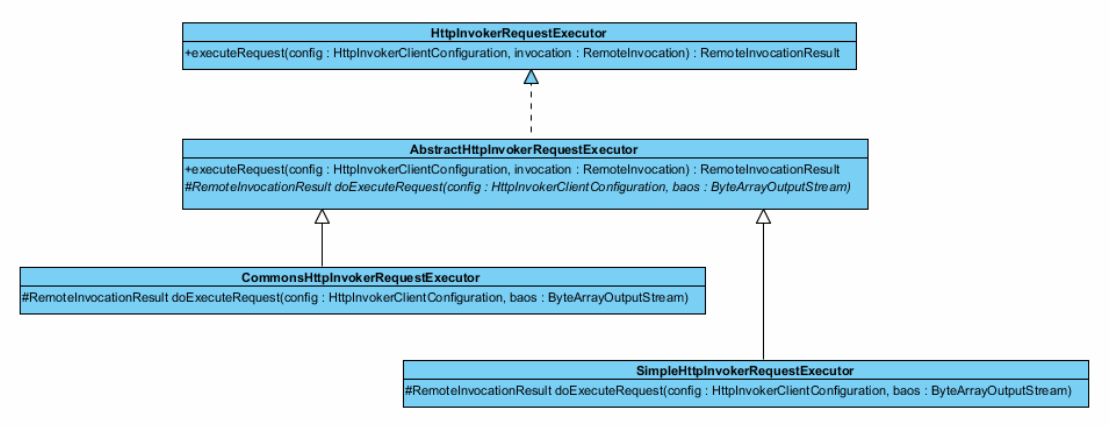
```
        else {
            return this.value;
        }
    }

}
```

我们看到拦截器方法除了实现对远程 HTTP 服务的调用外，还对返回结果进行了处理。如果在调用过程中产生任何连接异常，结果解析异常等，则直接翻译这些异常并且抛出。然后，有些情况下，一个调用在服务器端产生了异常，异常信息通过返回的调用结果返回给客户端，这样客户端需要解析这些异常，并且重现他们。

我们也理解，对远程 HTTP 服务的调用是通过 HTTP 调用器请求执行器实现的。它将远程调用序列化成对象流后，通过 HTTP 连接类的支持，写入到远程服务。当远程服务处理这个请求后，返回了远程调用结果对象，并且通过反序列化解析远程调用结果对象。如下代码所示，

```
// 代理方法
protected RemoteInvocationResult executeRequest(
        RemoteInvocation invocation, MethodInvocation originalInvocation) throws
Exception {

    return executeRequest(invocation);
}

protected RemoteInvocationResult executeRequest(RemoteInvocation invocation) throws
Exception {
    // 调用HTTP调用器请求执行器实现远程调用的
    return getHttpInvokerRequestExecutor().executeRequest(this, invocation);
}
```

程序分析到这里，我们理解所有远程调用的处理过程是通过 HTTP 调用器请求执行器实现的。HTTP 调用器请求执行器并不是一个单一的实现，它是一个接口，并且有两个常用的实现，一个使用 JDK 自带的 HTTP 客户端进行通讯，而另外一个使用 Apache Commons 的 HTTP 客户端，如下类图所示，



图表 4-25

为了使我们的分析简单明白，我们这里仅仅分析基于 JDK 自带的 HTTP 客户端进行通讯的实现。首先，我们分析一下如下抽象 HTTP 唤起器请求执行器（AbstractHttpInvokerRequestExecutor）的实现。如下代码注释，

```
public final RemoteInvocationResult executeRequest(
        HttpInvokerClientConfiguration config, RemoteInvocation invocation) throws
Exception {

    // 将远程调用对象序列化并且输出到字节数组输出流中
    ByteArrayOutputStream baos = getByteArrayOutputStream(invocation);

    if (logger.isDebugEnabled()) {
        logger.debug("Sending HTTP invoker request for service at [" +
config.getServiceUrl() +
                "], with size " + baos.size());
    }

    // 代理到其他方法处理
    return doExecuteRequest(config, baos);
}

// 抽象方法供子类实现
protected abstract RemoteInvocationResult doExecuteRequest(
        HttpInvokerClientConfiguration config, ByteArrayOutputStream baos)
        throws Exception;


protected RemoteInvocationResult readRemoteInvocationResult(InputStream is, String
codebaseUrl)
throws IOException, ClassNotFoundException {
    // 从HTTP请求体的输入流创建对象输入流
```

```
    ObjectInputStream ois = createObjectInputStream(decorateInputStream(is),
codebaseUrl);
    try {
        // 代理到其他方法进行读操作
      return doReadRemoteInvocationResult(ois);
    }
    finally {
      ois.close();
    }
}

// 代理方法，子类可以客户换
protected InputStream decorateInputStream(InputStream is) throws IOException {
    return is;
}

protected ObjectInputStream createObjectInputStream(InputStream is, String codebaseUrl)
throws IOException {
    // 使用特殊的对象输入流，如果本地不能解析放回结果中的某个类，则试图通过网络URL配置的资源解析类
    return new CodebaseAwareObjectInputStream(is, getBeanClassLoader(), codebaseUrl);
}

protected RemoteInvocationResult doReadRemoteInvocationResult(ObjectInputStream ois)
throws IOException, ClassNotFoundException {
    // 从对象流中读对象，如果没有读到远程调用结果对象，则抛出异常，终止处理
    Object obj = ois.readObject();
    if (!(obj instanceof RemoteInvocationResult)) {
    throw new RemoteException("Deserialized object needs to be assignable to type [" +
            RemoteInvocationResult.class.getName() + "]: " + obj);
    }

    // 返回远程调用对象
    return (RemoteInvocationResult) obj;
}
```

如此可见，对于一个远程调用抽象 HTTP 唤起器请求执行器实现了整个流程框架，但是留下了一个抽象方法，这个方法是提供给子类使用不同的方式与 HTTP 远程服务通信的。下面我们分析简单 HTTP 唤起器请求执行器（SimpleHttpInvokerRequestExecutor）是如何通过 JDK 自带的 HTTP 客户端来完成这个任务的。如下代码所示，

```
// 通过JDK自带的HTTP客户端实现基于HTTP协议的远程调用
public class SimpleHttpInvokerRequestExecutor extends AbstractHttpInvokerRequestExecutor
{

```

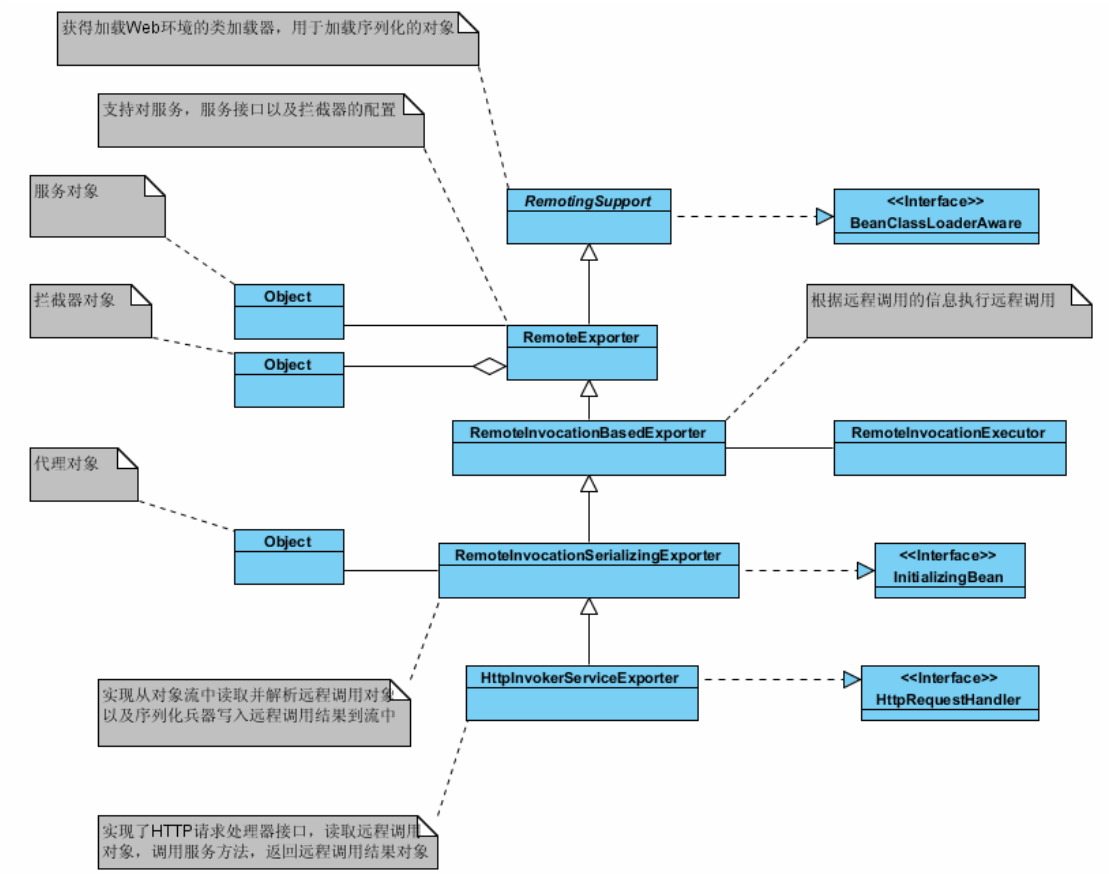
```
    @Override
    protected RemoteInvocationResult doExecuteRequest(
            HttpInvokerClientConfiguration config, ByteArrayOutputStream baos)
            throws IOException, ClassNotFoundException {

        // 打开远程的HTTP连接
        HttpURLConnection con = openConnection(config);

        // 设置HTTP连接信息
        prepareConnection(con, baos.size());

        // 把准备好的序列化的远程方法调用对象的字节流写入到HTTP请求体中
        writeRequestBody(config, con, baos);

        // 校验HTTP响应
        validateResponse(config, con);

        // 获得HTTP相应体的流对象
        InputStream responseBody = readResponseBody(config, con);

        // 读取远程调用结果对象并返回
        return readRemoteInvocationResult(responseBody, config.getCodebaseUrl());
    }


    protected HttpURLConnection openConnection(HttpInvokerClientConfiguration config)
throws IOException {
        // 打开远程服务的URL连接
        URLConnection con = new URL(config.getServiceUrl()).openConnection();

        // 这必须是基于HTTP协议的服务
        if (!(con instanceof HttpURLConnection)) {
            throw new IOException("Service URL [" + config.getServiceUrl() + "] is not
an HTTP URL");
        }

        return (HttpURLConnection) con;
    }


    protected void prepareConnection(HttpURLConnection con, int contentLength) throws
IOException {
        // 我们需要写入远程调用的序列化对象的二进制流，所以，需要使用HTTP POST方法，并且需要输出信息
到服务器
```

```
        con.setDoOutput(true);
        con.setRequestMethod(HTTP_METHOD_POST);

        // 内容类型是application/x-java-serialized-object，长度是程调用的序列化对象的二进制流的长度
        con.setRequestProperty(HTTP_HEADER_CONTENT_TYPE, getContentType());
        con.setRequestProperty(HTTP_HEADER_CONTENT_LENGTH, Integer.toString(contentLength));

        // 如果地域信息存在，则设置接收语言
        LocaleContext locale = LocaleContextHolder.getLocaleContext();
        if (locale != null) {
            con.setRequestProperty(HTTP_HEADER_ACCEPT_LANGUAGE, StringUtils.toLanguageTag(locale.getLocale()));
        }

        // 如果接受压缩选项，则使用压缩的HTTP通信
        if (isAcceptGzipEncoding()) {
            con.setRequestProperty(HTTP_HEADER_ACCEPT_ENCODING, ENCODING_GZIP);
        }
    }

    protected void writeRequestBody(
            HttpInvokerClientConfiguration config, HttpURLConnection con, ByteArrayOutputStream baos)
            throws IOException {
        // 远程调用的序列化对象的二进制流写到HTTP连接的输出流中
        baos.writeTo(con.getOutputStream());
    }


    protected void validateResponse(HttpInvokerClientConfiguration config, HttpURLConnection con)
            throws IOException {

        // 如果接受HTTP响应代码出错，则抛出异常终止处理，说明服务器没有开启，或者服务配置错误
        if (con.getResponseCode() >= 300) {
            throw new IOException(
                    "Did not receive successful HTTP response: status code = " + con.getResponseCode() +
                    ", status message = [" + con.getResponseMessage() + "]");
        }
    }
```

```
    protected InputStream readResponseBody(HttpInvokerClientConfiguration config,
HttpURLConnection con)
            throws IOException {

        if (isGzipResponse(con)) {
            // GZIP response found - need to unzip.
            return new GZIPInputStream(con.getInputStream());
        }
        else {
            // Plain response found.
            return con.getInputStream();
        }
    }

    protected boolean isGzipResponse(HttpURLConnection con) {
        // 通过HTTP头判断是否这是一个压缩过的HTTP相应
        String encodingHeader = con.getHeaderField(HTTP_HEADER_CONTENT_ENCODING);
        return (encodingHeader != null &&
encodingHeader.toLowerCase().indexOf(ENCODING_GZIP) != -1);
    }

}
```

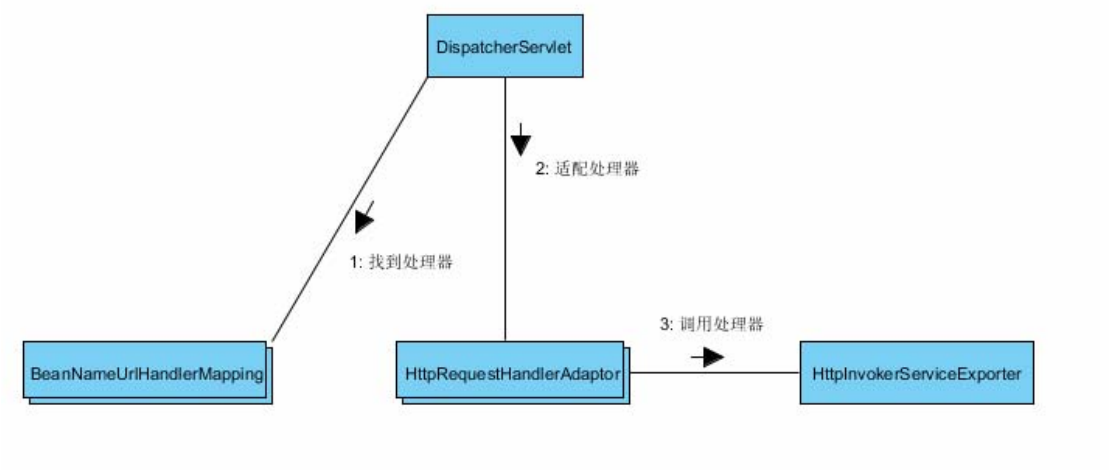
由此可见，简单 HTTP 唤起器请求执行器通过 JDK 自带的 URL 连接类实现了远程调用所需要 HTTP 通信的流程。这个流程支持 HTTP 请求体的压缩，并且能够使用地域信息进行一定程度的客户化。

代码分析到这里，我们理解一个远程调用（RemoteInvocation）对象写入了 HTTP 服务中后，服务器就会返回一个远程调用结果（RemoteInvocationResult）对象。下面我们分析服务器端是如何实现这个过程的。如下类图所示，



图表 4-26

服务器端对基于 HTTP 请求处理器流程的实现仍然是建立在 Spring Web MVC 的实现架构中的。它和基于简单控制器流程的实现和基于注解控制器流程的实现非常相似，同样有处理器映射，处理器适配器和处理器的实现，但是唯一不同的是它并没有特殊的处理器映射的实现，因为客户端是通过配置的 URL 查找和调用服务器端的服务的，所以简单基于简单控制器流程的实现中的 Bean 名 URL 处理器映射在这里可以被完全重用。如下流程图所示，



图表 4-27

下面我们根据上图中流程的先后顺序分析它的实现。首先，派遣器 Servlet 接收到一个包含有远程调用序列化的二进制流的 HTTP 请求，它会使用配置的处理器映射查找能够处理当前 HTTP 请求的处理器。通常情况下，Bean 名 URL 处理器映射会返回配置的 HTTP 调用器服务导出器(HttpInvokerServiceExporter)。如下配置代码所示，

```
<bean name="/AccountService"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
    <property name="service" ref="accountService"/>
    <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

Bean 名 URL 处理器映射会发现这个 Bean 是一个处理器 Bean，然后使用 Bean 作为 URL 注册此处理器。客户端 HTTP 请求发送到这个 URL，作为总控制器的派遣器 Servlet 要求 Bean 名 URL 处理器映射解析当前请求所需要的处理器，于是返回配置的 HTTP 调用器服务导出器。

既然 HTTP 调用器服务导出器实现了 HTTP 请求处理器接口(HttpRequestHandler)，HTTP 请求处理器适配器(HttpRequestHandlerAdapter)支持这个类型的处理器，所以，调用的控制流通过 HTTP 请求处理器适配器传递给 HTTP 调用器服务导出器。如下代码注释，

```
// 用于适配HTTP请求处理器的处理器适配器，主要用于实现基于HTTP的远程调用
public class HttpRequestHandlerAdapter implements HandlerAdapter {

    public boolean supports(Object handler) {
        // 支持类型任何实现了接口HttpRequestHandler的处理器
        return (handler instanceof HttpRequestHandler);
    }

    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response,
Object handler)
            throws Exception {

        // 传递控制给HttpRequestHandler，不需要返回值，响应是在HttpRequestHandler直接生成的
        ((HttpRequestHandler) handler).handleRequest(request, response);
        return null;
    }

    public long getLastModified(HttpServletRequest request, Object handler) {
        // 通用的实现HttpInvokerServiceExporter不支持最后修改操作
        if (handler instanceof LastModified) {
            return ((LastModified) handler).getLastModified(request);
        }
        return -1L;
```

```
    }

}
```

程序分析到这里，我们看到包含有远程调用序列化的二进制流的 HTTP 请求传递给 HTTP 调用器服务导出器，它将从序列化的二进制流中解析远程调用对象，进而解析方法调用对象，最后使用反射调用配置的服务中相应的方法，返回结果给客户端。如下代码所示，

```
// 方法没有返回值，响应是在处理请求时生成和返回给客户端的
public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    try {
        // 从请求中读取远程调用对象
        RemoteInvocation invocation = readRemoteInvocation(request);

        // 根据远程调用对象的信息，调用配置服务的相应服务方法
        RemoteInvocationResult result = invokeAndCreateResult(invocation, getProxy());

        // 写响应结果
        writeRemoteInvocationResult(request, response, result);
    }
    catch (ClassNotFoundException ex) {
        // 从客户端发送的二进制序列化流可能包含有不可识别的类类型，这是抛出异常终止处理
        throw new NestedServletException("Class not found during deserialization", ex);
    }
}
```

上述方法中可以看到实现的总体流程，这个流程分为三个步骤，一个步骤是读取远程调用对象，第二个步骤是调用相关的服务方法，第三个步骤则是写入返回的结果。

第一个步骤，读取远程调用对象代码注释如下，

```
protected RemoteInvocation readRemoteInvocation(HttpServletRequest request)
        throws IOException, ClassNotFoundException {
    // 从HTTP请求体中的流中读取远程调用对象
    return readRemoteInvocation(request, request.getInputStream());
}

protected RemoteInvocation readRemoteInvocation(HttpServletRequest request, InputStream
is)
        throws IOException, ClassNotFoundException {

    // 从简单的流构造对象流
```

```
    ObjectInputStream ois = createObjectInputStream(decorateInputStream(request, is));
    try {
        // 从对象流中读取远程调用对象
        return doReadRemoteInvocation(ois);
    }
    finally {
        ois.close();
    }
}

protected InputStream decorateInputStream(HttpServletRequest request, InputStream is)
throws IOException {
    // 占位符方法，仅仅返回HTTP请求体中的流对象
    return is;
}

protected ObjectInputStream createObjectInputStream(InputStream is) throws IOException {
    // 特殊的对象输入流，可以配置客户化的类加载器，如果某些类不存在，可以视图通过一个配置的URL加载此类
    return new CodebaseAwareObjectInputStream(is, getBeanClassLoader(), null);
}


protected RemoteInvocation doReadRemoteInvocation(ObjectInputStream ois)
throws IOException, ClassNotFoundException {
    // 读取HTTP请求体的序列化的对象
    Object obj = ois.readObject();

    // 如果不是远程调用的序列化流，则终止异常
    if (!(obj instanceof RemoteInvocation)) {
        throw new RemoteException("Deserialized object needs to be assignable to type ["
+
                RemoteInvocation.class.getName() + "]: " + obj);
    }

    return (RemoteInvocation) obj;
}
```

第二个步骤，调用相关的服务方法代码注释如下所示，

```
protected RemoteInvocationResult invokeAndCreateResult(RemoteInvocation invocation,
Object targetObject) {
    try {
        // 事实上调用服务方法
        Object value = invoke(invocation, targetObject);
```

```
            // 返回正确的远程结果
            return new RemoteInvocationResult(value);
        }
        catch (Throwable ex) {
            // 调用过程中如果有异常发生，则返回带有异常的远程调用结果，客户端将会重现这个异常
            return new RemoteInvocationResult(ex);
        }
    }


protected Object invoke(RemoteInvocation invocation, Object targetObject)
throws NoSuchMethodException, IllegalAccessException, InvocationTargetException {

    if (logger.isTraceEnabled()) {
            logger.trace("Executing " + invocation);
    }
    try {
            // 使用远程调用执行器执行服务方法，抛出产生的任何异常
            return getRemoteInvocationExecutor().invoke(invocation, targetObject);
    }
    catch (NoSuchMethodException ex) {
            if (logger.isDebugEnabled()) {
                logger.warn("Could not find target method for " + invocation, ex);
            }
            throw ex;
    }
    catch (IllegalAccessException ex) {
            if (logger.isDebugEnabled()) {
                logger.warn("Could not access target method for " + invocation, ex);
            }
            throw ex;
    }
    catch (InvocationTargetException ex) {
            if (logger.isDebugEnabled()) {
                logger.debug("Target method failed for " + invocation,
ex.getTargetException());
            }
            throw ex;
    }
}

//远程调用执行器只有一个缺省的实现，缺省的实现简单的调用远程调用对象的默认实现
public class DefaultRemoteInvocationExecutor implements RemoteInvocationExecutor {
```

```
    public Object invoke(RemoteInvocation invocation, Object targetObject)
            throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException{

        // 校验远程调用对象和配置的服务对象存在
        Assert.notNull(invocation, "RemoteInvocation must not be null");
        Assert.notNull(targetObject, "Target object must not be null");

        // 调用远程调用的默认实现
        return invocation.invoke(targetObject);
    }

}


public class RemoteInvocation implements Serializable {
    public Object invoke(Object targetObject) throws NoSuchMethodException,
            IllegalAccessException, InvocationTargetException {
        // 通过远程调用对象包含的方法名和参数类型找到服务对象的方法
        Method method = targetObject.getClass().getMethod(this.methodName,
                this.parameterTypes);

        // 调用找到的方法
        return method.invoke(targetObject, this.arguments);
    }
}
```

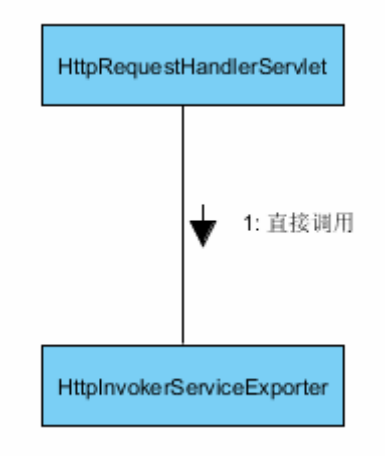
第三个步骤，写入返回的结果的代码注释如下所示，

```
protected void writeRemoteInvocationResult(
        HttpServletRequest request, HttpServletResponse response, RemoteInvocationResult
result)
        throws IOException {
    // 设置内容类型为application/x-java-serialized-object
    response.setContentType(getContentType());

    // 写远程调用结果到响应的输出流
    writeRemoteInvocationResult(request, response, result, response.getOutputStream());
}


protected void writeRemoteInvocationResult(
        HttpServletRequest request, HttpServletResponse response, RemoteInvocationResult
```

```
result, OutputStream os)
        throws IOException {
    // 创建对象输出流
    ObjectOutputStream oos = createObjectOutputStream(decorateOutputStream(request,
response, os));
    try {
        // 将远程调用结果对象写入到代表响应的对象输出流
        doWriteRemoteInvocationResult(result, oos);
        //
        将缓存刷新到HTTP响应体中
        oos.flush();
    }
    finally {
        oos.close();
    }
}

protected ObjectOutputStream createObjectOutputStream(OutputStream os) throws IOException
{
    // 创建简单的对象输出流，用于序列化远程调用结果对象
    return new ObjectOutputStream(os);
}

protected void doWriteRemoteInvocationResult(RemoteInvocationResult result,
ObjectOutputStream oos)
throws IOException {
    // 既然远程调用结果对象是可序列化的，直接写远程调用结果对象到对象流
    oos.writeObject(result);
}
```

如上分析，我们知道服务端的实现是讲客户端传递过来的远程调用的序列化对象进行反序列化，根据远程调用对象信息调用业务逻辑方法，最后同样以序列化的方法将调用结果传递回客户端。

事实上，还有另外一种配置方法配置基于 HTTP 请求处理器流程的实现方式。这种方法不需要任何的处理器映射和处理器适配器的实现，而是使用特殊的 HTTP 请求处理器 Servlet（HttpRequestHandlerServlet）。这个 Servlet 会将 HTTP 请求直接发送给相应的 HTTP 调用器服务导出器进行处理。如下图所示，



图表 4-28

如下代码所示，

```
public class HttpRequestHandlerServlet extends HttpServlet {

    private HttpRequestHandler target;


    @Override
    public void init() throws ServletException {
        // 取得根Web应用程序环境
        WebApplicationContext wac =
WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext());

        // 取得和此Servlet具有同名的Bean,这个Bean必须是HttpRequestHandler接口的实现
        this.target = (HttpRequestHandler) wac.getBean(getServletName(),
HttpRequestHandler.class);
    }


    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        // 设置地域信息
        LocaleContextHolder.setLocale(request.getLocale());
        try {
            // 讲控制流传递给HttpRequestHandler
            this.target.handleRequest(request, response);
        }
        // 既然我们重写了service方法，那么所有的HTTP方法的请求都可能会发送到这里，所以
```

```
HttpRequestHandler的实现需要检查是否当前请求使用了支持的HTTP方法
        catch (HttpRequestMethodNotSupportedException ex) {
            String[] supportedMethods = ((HttpRequestMethodNotSupportedException)
ex).getSupportedMethods();
            if (supportedMethods != null) {
                // 如果异常中保存了支持的HTTP方法信息
                response.setHeader("Allow",
StringUtils.arrayToDelimitedString(supportedMethods, ", "));
            }
            // 发送方法禁止错误状态
            response.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED,
ex.getMessage());
        }
        finally {
            LocaleContextHolder.resetLocaleContext();
        }
    }

}
```

根据上面的这个简单 Servlet 的实现，我们可以这样配置我们的远程调用导出器。如下代码所示，

跟环境

```
<bean name="accountExporter"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
   <property name="service" ref="accountService"/>
   <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

web.xml

```
<servlet>
    <servlet-name>accountExporter</servlet-name>
    <servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</
servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>accountExporter</servlet-name>
    <url-pattern>/remoting/AccountService</url-pattern>
</servlet-mapping>
```

我们可以看到这种实现方法涉及组件会更少，不需要处理器映射和处理器适配器的参与，直接实现了 Servlet 到处理器适配器的调用，相对于第一种方法而言可以称为是一个捷径。但是它需要单独配置一个 Servlet，除此之外，HTTP 请求处理器 Servlet 仅仅实现了 HTTP Servlet, 它并不拥有专用的子环境，所以需要在根环境声明服务导出 Bean。

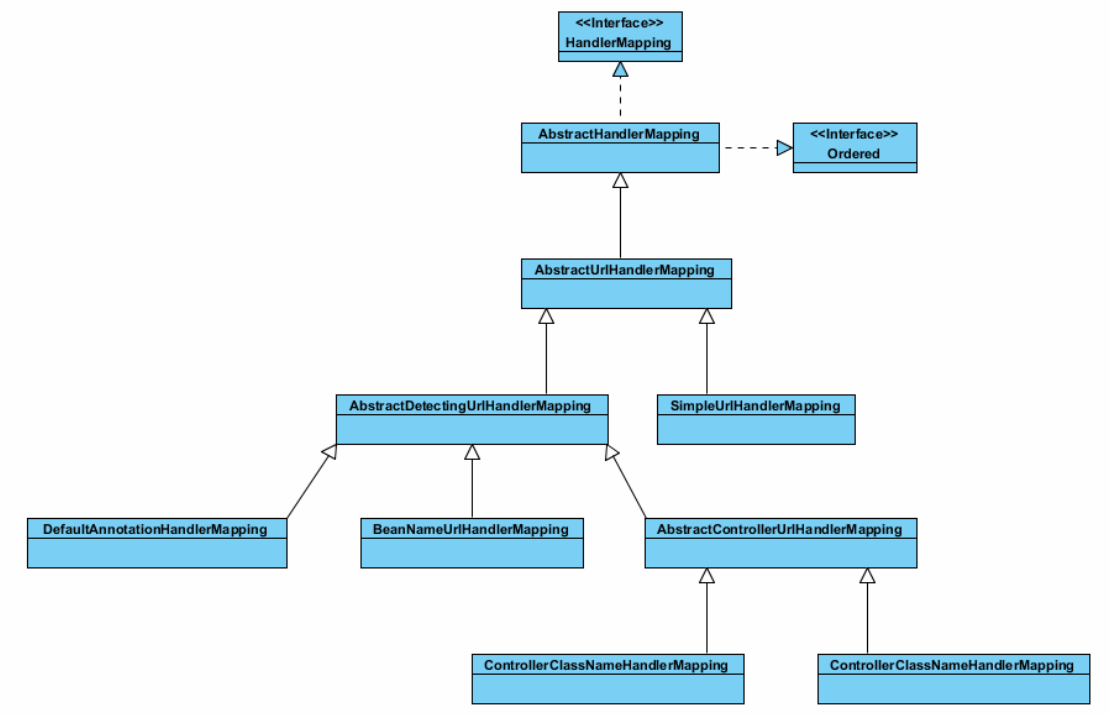
4.2.2 纵向剖析

前一小节中，我们根据不同流程的实现分析了基于简单控制器流程的实现，基于注解控制器流程的实现和基于 HTTP 请求处理器流程的实现。对不同流程的实现使用不同的处理器映射，处理器适配器和处理器的模块实现。事实上，不同流程中使用的处理器映射，处理器适配器和处理器的实现并不是完全隔离的。对于其中的每一个模块，通过面向对象的方法和分析，抽象出一个合理的，可扩展的，最大限度可重用的树形实现体系结构。体系结构中的每一个层次会完成一个特定的功能，这样使插入更多流程的实现成为可能。

下面我们针对不同模块，包括处理器映射，处理器适配器和处理器等，深入分析每个模块的架构实现，最终我们会理解这些模块的实现是如何支撑上一节讨论的三个流程的实现，这些实现会在最大限度重用的基础上具有相对的独立性。

4.2.2.1处理器映射的实现架构

作为总控制器的派遣器 Servlet 首先会轮询处理器映射模块，查找能够处理当前请求的处理器，处理器映射模块根据当前请求的 URL 返回简单的控制器类型，注解控制器类型或者远程调用处理器类型。前一小节中，我们根据流程的实现分析了 Bean 名 URL 处理器映射(BeanNameUrlHandlerMapping)和缺省注解处理器映射(DefaultAnnotationHandlerMapping)。事实上，在处理器映射的实现体系结构中还有其他的实现，用来实现根据不同的规则查找相应的处理器的逻辑。不同层次的处理器映射的实现，无论是抽象类还是实现类，都关联着特殊而又完整的逻辑。如下类图所示，



图表 4-29

上一节流程分析中讨论的 Bean 名 URL 处理器映射（BeanNameUrlHandlerMapping）和缺省注解处理器映射（DefaultAnnotationHandlerMapping），他们是经常使用到的处理器映射的实现，下面我们介绍所有的处理器映射的具体实现类的逻辑功能。

- Bean 名 URL 处理器映射(BeanNameUrlHandlerMapping)

  这个实现类通过识别 Web 应用程序环境中以 URL 为名字声明的 Bean 为处理器。URL 是通过以斜线(/)开头的并且以斜线(/)分隔的字符串。然后，使用 Bean 名中声明的 URL 和请求的 URL 进行匹配，如果匹配成功，则使用匹配的 Bean 作为处理器返回。

- 缺省注解处理器映射(DefaultAnnotationHandlerMapping)

  这个实现类通过声明在 Web 应用程序环境中 Bean 类型中的请求映射注解(@RequestMapping)来注册处理器映射的。请求映射注解声明有匹配请求 URL 所用的 URL Pattern。然后，使用方法级别的请求映射注解中声明的 URL Pattern 和类型级别的请求映射注解中声明的 URL Pattern 结合并且匹配请求的 URL，如果匹配成功，则使用匹配的 Bean 作为处理器返回。

- 控制器类名处理器映射(ControllerClassNameHandlerMapping)

  这个实现类通过声明在 Web 应用程序环境中的控制器类型来注册处理器映射的。它从控制器的类型转换出控制器所服务的 URL Pattern。这个转换规则是，把点号分割的具有包前缀的类名替换成斜线(/)分割的具有包前缀的字符串，再加上前缀和后缀构成 URL Pattern，然后，使用得到的 Pattern 匹配请求的 URL，如果匹配成功，则使用匹配的 Bean

作为处理器返回。

- 控制器 Bean 名处理器映射(ControllerBeanNameHandlerMapping)

  这个实现类通过声明在 Web 应用程序环境中的控制器类型来注册处理器映射的。它从控制器的 Bean 名字转换出控制器所服务的 URL Pattern。这个转换规则是，把 Bean 名字加上前缀和后缀构成 URL Pattern，然后，使用得到的 Pattern 匹配请求的 URL，如果匹配成功，则使用匹配的 Bean 作为处理器返回。

- 简单 URL 处理器映射(SimpleUrlHandlerMapping)

  这个实现类通过配置一套 URL Pattern 到处理器的映射而实现的。它使用配置的映射中的 URL Pattern 匹配请求中的 URL，如果匹配成功，则使用匹配 URL Pattern 映射的 Bean 作为处理器返回。

我们看到具体的实现类并不是直接实现处理器映射接口的，而是通过一系列的抽象类的实现最终完成的，在每个抽象类的实现层次上完成不同的独立的逻辑功能。下面我们分析这些抽象类和实现他们的具体实现类是如何分工并且最终完成必要的业务逻辑的。

- 抽象处理器映射(AbstractHandlerMapping)

  抽象处理器映射是处理器映射实现中的最底层的实现，它直接实现处理器映射接口，并且继承 Web 应用程序环境支持对象。它提供了配置缺省处理器以及应用到所有处理器上的处理器拦截器的功能。

  它把具体如何取得一个处理器抽象并且留给子类进行特殊化的实现。

- 抽象 URL 处理器映射(AbstractUrlHandlerMapping)

  抽象 URL 处理器映射继承自抽象处理器映射，实现了取得一个处理器的抽象方法，提供了功能根据 URL 进行匹配处理器的功能。也提供了根据 URL 查找应用在处理器上特殊的拦截器。并且提供了方法实现注册 URL 到处理器的映射。

  如何获得 URL 到处理器的映射的逻辑留给子类进行完成。

  简单 URL 处理器映射就是通过应用程序环境中 Bean 串联配置直接注射 URL 到处理器映射来实现抽象 URL 处理器映射的。

- 抽象探测 URL 处理器映射(AbstractDetectingUrlHandlerMapping)

  抽象探测 URL 处理器映射通过一定的规则在 Web 应用程序环境中自动发现 URL 到处理器的映射。

  使用什么样的规则在 Web 应用程序环境中自动发现 URL 到处理器的映射并没有直接实

现，因为这会有很多的映射规则，并且根据需求可以自由扩展。这个规则留给子类进行实现。

Bean 名 URL 处理器映射就是根据把 Bean 名声明作为 URL 来发现处理器的。而缺省注解处理器映射是根据声明在控制器中的请求映射注解中包含的 URL Pattern 信息来解析处理器的。

- 抽象控制器 URL 处理器映射(AbstractControllerUrlHandlerMapping)

  这是抽象探测 URL 处理器映射的另外一个实现，这个实现也是一个抽象的实现，它是通过在 Bean 环境中找到合适的控制器类型，根据一定的规则将控制器类型映射到一个或者多个 URL Pattern 来实现的。

  它有两个具体的实现类，控制器类名处理器映射是根据类名解析出映射的 URL Pattern。而控制器 Bean 名处理器映射则是根据控制器在 Web 应用程序环境中声明的 Bean 名映射到 URL Pattern 的。

在上一节的分析中，我们已经对 Bean 名 URL 处理器映射（BeanNameUrlHandlerMapping）和缺省注解处理器映射（DefaultAnnotationHandlerMapping）以及他们的父类进行分析。下面我们将对剩余的其他的类进行代码分析，首先回顾一下抽象探测 URL 处理器映射的实现，如下代码注释，

```
protected abstract String[] determineUrlsForHandler(String beanName);
```

它留下了一个抽象方法，对于 Web 应用程序环境中的每一个 Bean，都将使用此方法找到 Bean 所映射到的 URL Pattern。子类需要根据具体的映射规则来实现这个方法。除了上一节中分析的 Bean 名 URL 处理器映射和缺省注解处理器映射实现了这个方法外，存在另外一套根据控制器类型映射的实现。这套根据控制器类型映射的实现包含一个抽象实现和两个具体实现，抽象实现是抽象控制器处理器映射。如下代码所示，

```
public abstract class AbstractControllerUrlHandlerMapping extends
AbstractDetectingUrlHandlerMapping  {

    // 通过控制器接口类型或者控制器注解类型查找简单控制器和注解控制器的实用类
    private ControllerTypePredicate predicate = new AnnotationControllerTypePredicate();

    // 声明的包中的控制器不会作为处理器进行注册
    private Set<String> excludedPackages =
Collections.singleton("org.springframework.web.servlet.mvc");

    // 声明的类不会作为处理器进行注册
    private Set<Class> excludedClasses = Collections.emptySet();

    // 实现根据Bean映射出URL的逻辑
```

```
    @Override
    protected String[] determineUrlsForHandler(String beanName) {
        // 取得Bean的类型
        Class beanClass = getApplicationContext().getType(beanName);

        // 判断是否Bean可以作为控制器处理器
        if (isEligibleForMapping(beanName, beanClass)) {
            // 根据Bean名字或者类型名映射出URL的逻辑
            return buildUrlsForHandler(beanName, beanClass);
        }
        else {
            return null;
        }
    }


    protected boolean isEligibleForMapping(String beanName, Class beanClass) {
        // 如果Bean类型是空，则不映射此Bean
        if (beanClass == null) {
            if (logger.isDebugEnabled()) {
                logger.debug("Excluding controller bean '" + beanName + "' from class name
mapping " +
                        "because its bean type could not be determined");
            }
            return false;
        }

        // 如果Bean的包配置为排除包，则不映射此Bean
        if (this.excludedClasses.contains(beanClass)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Excluding controller bean '" + beanName + "' from class name
mapping " +
                        "because its bean class is explicitly excluded: " +
beanClass.getName());
            }
            return false;
        }

        // 如果Bean的包配置为排除类，则不映射此Bean
        String beanClassName = beanClass.getName();
        for (String packageName : this.excludedPackages) {
            if (beanClassName.startsWith(packageName)) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Excluding controller bean '" + beanName + "' from class
```

```
name mapping " +
                                "because its bean class is defined in an excluded package:
" + beanClass.getName());
                    }
                    return false;
                }
            }

            // 必须是控制器类型，才映射作为控制器处理器
            return isControllerType(beanClass);
        }

        protected boolean isControllerType(Class beanClass) {
            return this.predicate.isControllerType(beanClass);
        }

        protected boolean isMultiActionControllerType(Class beanClass) {
            return this.predicate.isMultiActionControllerType(beanClass);
        }

        // 子类可以选择根据Bean名字还是根据Bean类来映射URL Pattern
        protected abstract String[] buildUrlsForHandler(String beanName, Class beanClass);

}
```

抽象控制器处理器映射有两个实现，一个是根据 Bean 名字映射到 URL Pattern 的实现，另外一个是根据 Bean 的类型映射到 URL Pattern 的实现。

以下是控制器 Bean 名处理器映射(ControllerBeanNameHandlerMapping)的代码注释，

```
public class ControllerBeanNameHandlerMapping extends
AbstractControllerUrlHandlerMapping {

    private String urlPrefix = "";

    private String urlSuffix = "";

    @Override
    protected String[] buildUrlsForHandler(String beanName, Class beanClass) {
        List<String> urls = new ArrayList<String>();
```

```
        // 根据Bean名产生URL Pattern
        urls.add(generatePathMaping(beanName));

        // 对于Bean名的别名，以同样的规则产生URL Pattern
        String[] aliases = getApplicationContext().getAliases(beanName);
        for (String alias : aliases) {
            urls.add(generatePathMapping(alias));
        }

        // 返回URL Pattern数组
        return StringUtils.toStringArray(urls);
    }

    /**
     * Prepends a '/' if required and appends the URL suffix to the name.
     */
    protected String generatePathMapping(String beanName) {
        // 如果bean名不是以斜线(/)开头，则增加斜线(/)
        String name = (beanName.startsWith("/") ? beanName : "/" + beanName);
        StringBuilder path = new StringBuilder();

        // 添加前缀
        if (!name.startsWith(this.urlPrefix)) {
            path.append(this.urlPrefix);
        }

        path.append(name);

        // 添加后缀
        if (!name.endsWith(this.urlSuffix)) {
            path.append(this.urlSuffix);
        }
        return path.toString();
    }

}
```

以下是控制器类名处理器映射(ControllerClassNameHandlerMapping)的代码注释，

```
public class ControllerClassNameHandlerMapping extends
AbstractControllerUrlHandlerMapping {
    // 控制器名的后缀
    private static final String CONTROLLER_SUFFIX = "Controller";
```

```
    // 通过类型映射的路径是否保持大写字母的存在
    private boolean caseSensitive = false;

    private String pathPrefix;

    private String basePackage;

    public void setPathPrefix(String prefixPath) {
        this.pathPrefix = prefixPath;

        // 一个路径应该保证有斜线(/)开头，但是没有斜线(/)结尾
        if (StringUtils.hasLength(this.pathPrefix)) {
            if (!this.pathPrefix.startsWith("/")) {
                this.pathPrefix = "/" + this.pathPrefix;
            }
            if (this.pathPrefix.endsWith("/")) {
                this.pathPrefix = this.pathPrefix.substring(0, this.pathPrefix.length()
- 1);
            }
        }
    }


    public void setBasePackage(String basePackage) {
        this.basePackage = basePackage;

        // 设置缺省的包前缀
        if (StringUtils.hasLength(this.basePackage) && !this.basePackage.endsWith("."))
{
            this.basePackage = this.basePackage + ".";
        }
    }

    @Override
    protected String[] buildUrlsForHandler(String beanName, Class beanClass) {
        // 仅仅使用类名进行映射
        return generatePathMappings(beanClass);
    }

    protected String[] generatePathMappings(Class beanClass) {
        // 产生路径前缀
        StringBuilder pathMapping = buildPathPrefix(beanClass);

        // 取得不包含包名的类名
```

```
        String className = ClassUtils.getShortName(beanClass);

        // 如果以控制器后缀(Controller)结尾，则移除控制器后缀
        String path = (className.endsWith(CONTROLLER_SUFFIX) ?
                className.substring(0, className.lastIndexOf(CONTROLLER_SUFFIX)) :
classNAme);

        if (path.length() > 0) {
            // 如果保持路径大小写，则把类名的第一个字符小写
            if (this.caseSensitive) {
                pathMapping.append(path.substring(0,
1).toLowerCase()).append(path.substring(1));
            }
            // 否则使所有路径字符变成小写
            else {
                pathMapping.append(path.toLowerCase());
            }
        }

        // 如果是多行为控制器类型，则加URL本身和所有的子URL
        if (isMultiActionControllerType(beanClass)) {
            return new String[] {pathMapping.toString(), pathMapping.toString() + "/*"};
        }
        // 否则只加URL本身
        else {
            return new String[] {pathMapping.toString() + "*"};
        }
    }


    private StringBuilder buildPathPrefix(Class beanClass) {
        StringBuilder pathMapping = new StringBuilder();

        // 第一部分是路径前缀
        if (this.pathPrefix != null) {
            pathMapping.append(this.pathPrefix);
            pathMapping.append("/");
        }
        else {
            pathMapping.append("/");
        }

        // 第二部分是包名中逗点替换成斜线的结果
        if (this.basePackage != null) {
```

```
            String packageName = ClassUtils.getPackageName(beanClass);
            if (packageName.startsWith(this.basePackage)) {
                String subPackage =
packageName.substring(this.basePackage.length()).replace('.', '/');
                pathMapping.append(this.caseSensitive ? subPackage :
subPackage.toLowerCase());
                pathMapping.append("/");
            }
        }
        return pathMapping;
    }

}
```

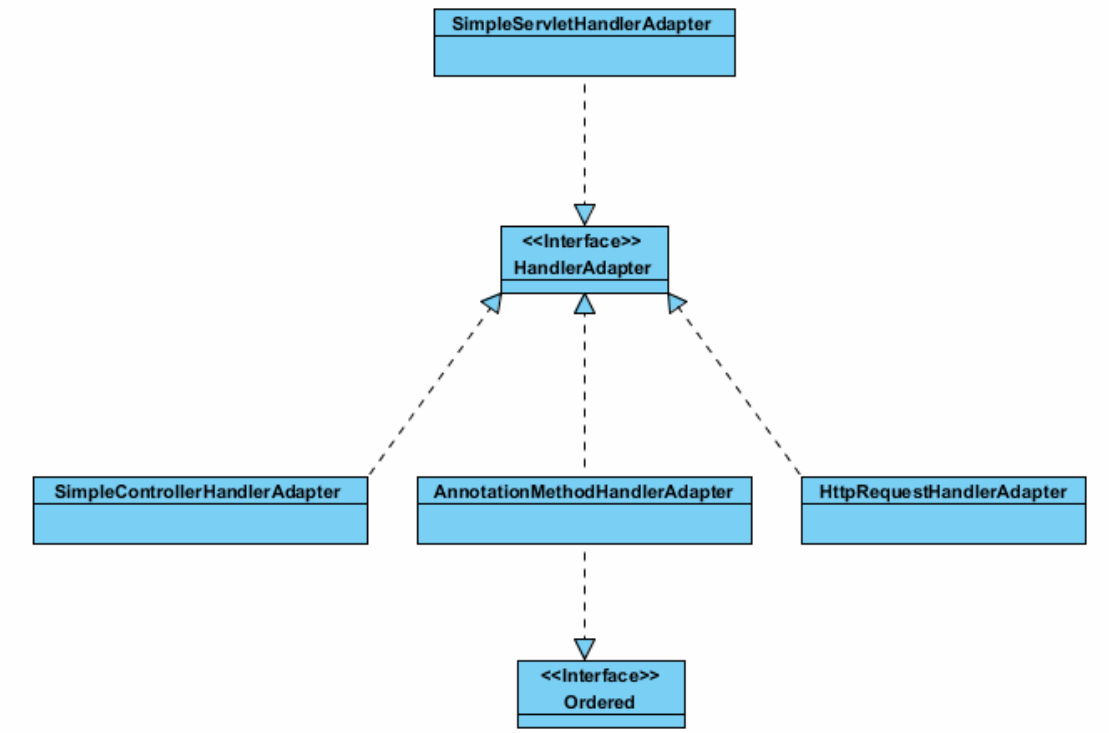
抽象 URL 处理器映射有另外一个具体实现简单 URL 处理器映射，它根据配置的 URL Pattern 到处理器的映射来查找处理器，如下代码所示，

```
public class SimpleUrlHandlerMapping extends AbstractUrlHandlerMapping {

    private final Map<String, Object> urlMap = new HashMap<String, Object>();

    // 通过属性配置URL到Bean名的映射
    public void setMappings(Properties mappings) {
        CollectionUtils.mergePropertiesIntoMap(mappings, this.urlMap);
    }

    // 配置URL到Bean的映射
    public void setUrlMap(Map<String, ?> urlMap) {
        this.urlMap.putAll(urlMap);
    }

    public Map<String, ?> getUrlMap() {
        return this.urlMap;
    }

    @Override
    public void initApplicationContext() throws BeansException {
        super.initApplicationContext();
        // 初始化的时候注册处理器
        registerHandlers(this.urlMap);
    }

    protected void registerHandlers(Map<String, Object> urlMap) throws BeansException {
        // 如果配置的处理器映射为空，则警告
```

```
        if (urlMap.isEmpty()) {
            logger.warn("Neither 'urlMap' nor 'mappings' set on
SimpleUrlHandlerMapping");
        }
        else {
            // 对于没一个配置的URL到处理器的映射，如果URL不是以斜线(/)开头，则追加斜线开头，则注册
处理器
            for (Map.Entry<String, Object> entry : urlMap.entrySet()) {
                String url = entry.getKey();
                Object handler = entry.getValue();
                // Prepend with slash if not already present.
                if (!url.startsWith("/")) {
                    url = "/" + url;
                }
                // Remove whitespace from handler bean name.
                if (handler instanceof String) {
                    handler = ((String) handler).trim();
                }
                registerHandler(url, handler);
            }
        }
    }

}
```

4.2.2.2处理器适配器的实现架构

作为总控制器的派遣器 Servlet 通过处理器映射得到处理器后，会轮询处理器适配器模块，查找能够处理当前 HTTP 请求的处理器适配器的实现，处理器适配器模块根据处理器映射返回的处理器类型，例如，简单的控制器类型，注解控制器类型或者远程调用处理器类型来选择某一个适当的处理器适配器的实现来适配当前的 HTTP 请求。前一小节中，我们根据流程的实现分析了简单控制器处理器适配器 (SimpleControllerHandlerAdapter)，注解方法处理器适配器 (AnnotationMethodHandlerAdapter) 和 HTTP 请求处理器适配器 (HttpRequestHandlerAdapter)的架构实现。事实上，还有另外一个简单的处理器适配器可以将请求适配到一个已存 Servlet 的实现。如下类图所示，



图表 4-30

处理器适配器的实现体系架构简单明了，体系架构中的针对不同流程的处理器适配器的实现基本不共用任何逻辑和抽象类，它们互相独立自成独立的模块。我们已经在上一小节对基于流程的实现的剖析中已经对处理器适配器的三个实现进行了深入剖析，事实上在处理器适配器架构上还有另外一个实现，就是简单 Servlet 处理器适配器的实现，它简单的将 HTTP 请求适配到 Servlet 进行处理。下面我们对每一个处理器适配器的功能实现进行分析。

- 简单控制器处理器适配器(SimpleControllerHandlerAdapter)

这个实现类将 HTTP 请求适配到一个控制器的实现进行处理。这里控制器的实现是一个简单的控制器接口的实现。简单控制器处理器适配器被设计成一个框架类的实现，不需要被改写，客户化的业务逻辑通常是在控制器接口的实现类中实现的。

- 注解方法处理器适配器(AnnotationMethodHandlerAdapter)

从上一小节对流程的分析得知，这个类的实现是基于注解的实现，它需要结合注解方法映射和注解方法处理器协同工作。它通过解析声明在注解控制器的请求映射信息来解析相应的处理器方法来处理当前的 HTTP 请求。在处理的过程中，它通过反射来发现探测处理器方法的参数，调用处理器方法，并且映射返回值到模型和控制器对象，最后返回模型和控制器对象给作为主控制器的派遣器 Servlet。

- HTTP 请求处理器适配器(HttpRequestHandlerAdapter)

HTTP 请求处理器适配器仅仅支持对 HTTP 请求处理器的适配。它简单的将 HTTP 请求对象和响应对象传递给 HTTP 请求处理器的实现，它并不需要返回值。它主要应用在基于 HTTP

的远程调用的实现上。

- 简单 Servlet 处理器适配器(SimpleServletHandlerAdapter)

这个实现能够将一个 HTTP 请求传递给一个 Servlet 规范中定义的 Servlet 的实现进行处理。它的应用并不广泛，主要应用在适配到一个已有的 Servlet 的实现以达到重用的目的。基于简单控制器流程的实现中，有个相似的类 ServletWrappingController 实现了同样的业务逻辑。

既然我们在前一些小节中对大部分的实现类进行了架构分析和代码注释，这里我们只对简单 Servlet 处理器适配器进行分析，如下代码注释，

```
public class SimpleServletHandlerAdapter implements HandlerAdapter {

    public boolean supports(Object handler) {
        // 仅仅支持实现了Servlet的处理器，这个处理器需要在Web应用程序中声明，但是， Servlet的初始化方法和析构方法不会被调用
        return (handler instanceof Servlet);
    }

    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        // 适配到Servlet的service方法进行处理，不需要返回值，返回值在Servlet的服务方法直接写入HTTP响应对象
        ((Servlet) handler).service(request, response);
        return null;
    }

    public long getLastModified(HttpServletRequest request, Object handler) {
        // 不支持最后修改行为
        return -1;
    }

}
```

4.2.2.3处理器的实现架构

作为总控制器的派遣器 Servlet 将得到的处理器传递给支持此处理器的处理器适配器，处理器适配器然后调用处理器中适当的处理器方法，最后返回处理结果给派遣器 Serlvet。

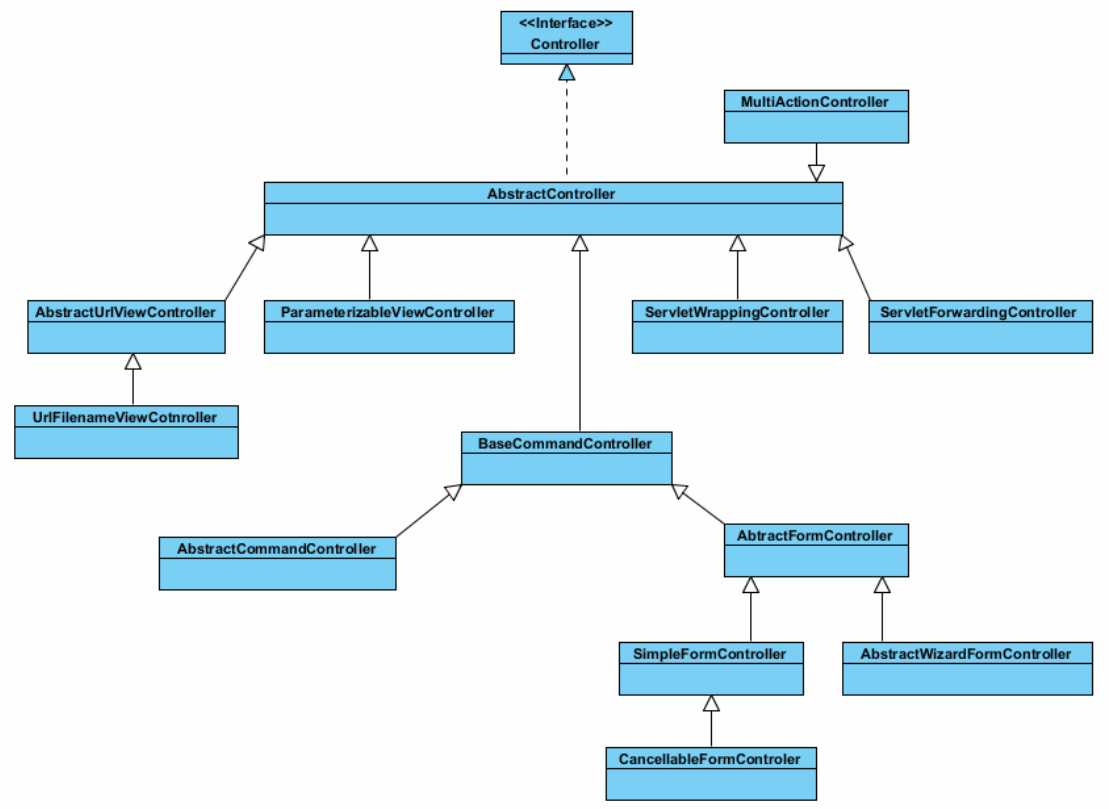
处理器架构中并没有简单的处理器接口定义，任何一个对象类型都可以成为处理器，每个类型的处理器都有一个对应的处理器适配器，用于将 HTTP 请求适配给一定类型的处理器。

处理器根据类型分为简单控制器，注解控制器和 HTTP 请求处理器。我们在讨论基于流程的实现的小节中，已经对每种类型中的典型实现类进行了剖析。下面我们就对不同类型的处理器进行详细的剖析。

4.2.2.3.1 简单控制器

简单控制器是最常用的处理器类型，它有一个简单的接口定义，接口有唯一的处理器方法，方法接受 HTTP 请求对象和 HTTP 响应对象作为参数，并且返回模型和视图对象。当派遣器 Servlet 派遣一个 HTTP 请求到简单控制器处理器适配器，简单控制器处理器适配器就会传递 HTTP 请求对象和 HTTP 响应对象给简单控制器的处理器方法，控制器处理器方法调用服务层的业务逻辑处理方法后返回模型和视图对象，模型和视图对象最后返回给派遣器 Servlet。

简单控制器接口有很多抽象的或者具体的实现类，每个层次的类都实现一个独立的逻辑功能。如下类图所示，



图表 4-31

从上图我们可以看到，简单控制器架构的实现比较复杂，它有很多的抽象实现类和具体的实现类，有些类之间互相独立，有些类是继承自其他类，设计和实现这些类有着不同的目的，实现不同的功能。下面通过功能对这些类实现进行分类。

- 抽象命令控制器(AbstractCommandController)

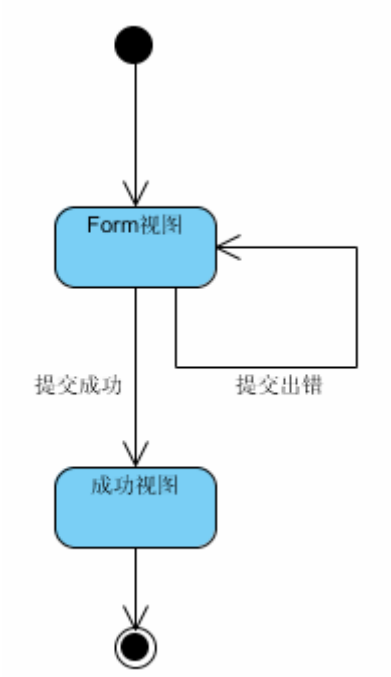
抽象命令控制器根据请求参数自动绑定一个命令类实例，子类实现根据绑定的命令类实例完成服务层的业务逻辑的调用后，最后决定跳转到某一个视图。它用来处理一个单一的直线式的流程。

- 抽象 Form 控制器(AbstractFormController)

抽象 Form 控制器也根据请求参数自动绑定一个命令类实例，它用来处理一个基于 Form 的流程，包括 Form 的显示，提交和取消。它是一个抽象类，但是它有不同的实现类实现不同的 Form 流程。下面我们具体介绍其三个实现类。

- 简单 Form 控制器(SimpleFormController)

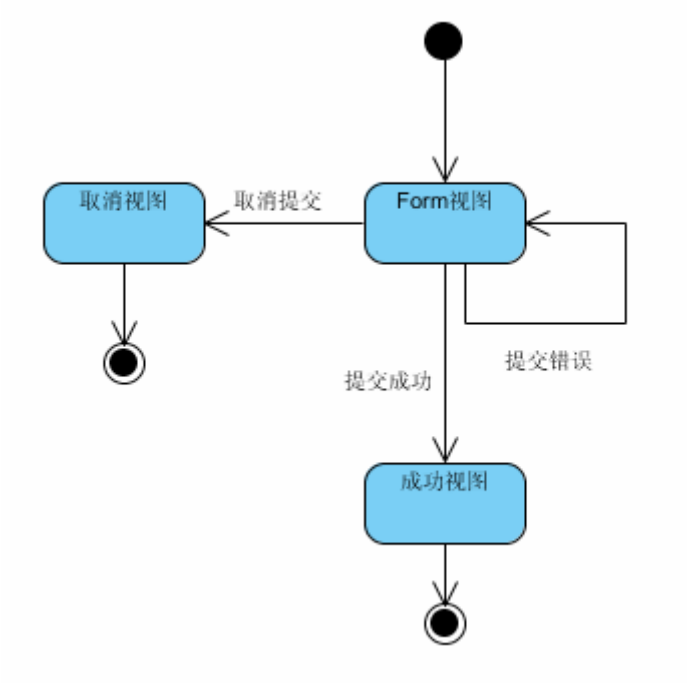
简单的 Form 控制器用来处理一个具有两个状态的 Form 处理流程。他们是显示 Form 状态和提交 Form 状态。通过配置 Form 视图和成功视图，简单的 Form 控制器就可以开始工作了。第一次请求页面时，它使用 HTTP GET 协议，简单 Form 控制器自动显示 Form 视图。当 Form 提交的时候，它使用 HTTP POST 协议，简单 Form 控制器就进行服务层次的逻辑调用，最后显示成功视图。



图表 4-32

- 可取消的 Form 控制器(CancellableFormController)

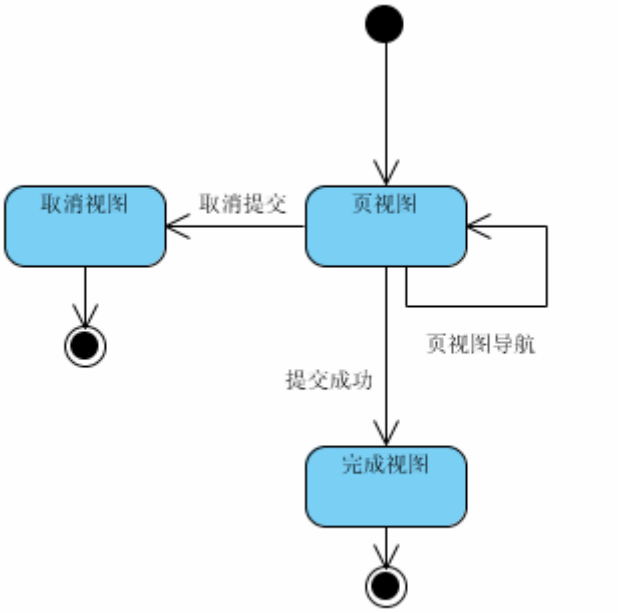
可取消的 Form 控制器是简单 Form 控制器的子类。它除了拥有显示 Form 状态和提交 Form 状态还增加了一个取消 Form 状态。如果一个 Form 提交中带有取消参数，则显示取消视图。



图表 4-33

- 抽象导航控制器(AbstractWizardFormController)

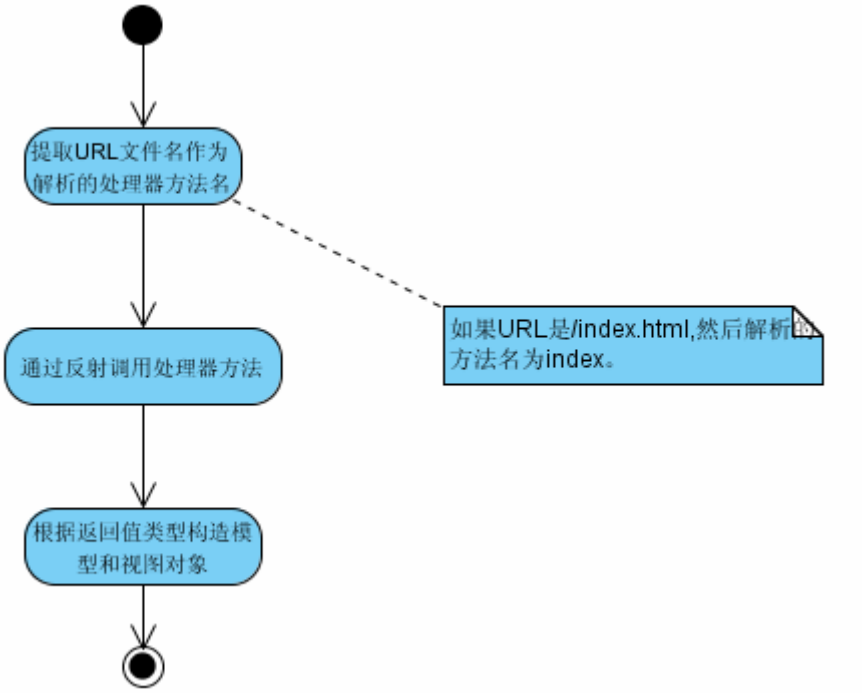
  抽象导航控制器更加复杂，它拥有更多的状态，它拥有一个或者多个页面状态，一个取消状态和一个完成状态。当一个 Form 加载时，它显示第一个页面，然后，通过页面提交的参数可以在不同的页面进行导航。当一个提交中包含取消参数或者完成参数，则显示取消视图或者完成视图。这是一个抽象类，子类可以根据业务需要在显示取消和完成试图前进行服务层次的业务逻辑的处理。



图表 4-34

- 多动作控制器(MultiActionController)

  多动作控制器是用于处理多个 HTTP 请求的处理器。它根据 HTTP 请求 URL 映射得到应该调用的处理器方法，通过反射调用处理器方法，并且封装返回结果作为模型和视图返回给简单控制器适配器。每个处理器方法可以有一个对应的最后修改方法，最后修改方法名是处理器方法名加上 LastModified 后缀构成的。最后修改方法也是通过反射调用并且返回结果的。



图表 4-35

- Servlet 相关控制器

  Servlet 相关控制器和简单 Servlet 处理器适配器功能类似，他们都实现将 HTTP 请求适配到一个已存的 Servlet 实现。但是，简单 Servlet 处理器适配器需要在 Web 应用程序环境中定义 Servlet Bean，并且 Servlet 没有机会进行初始化和析构。

  - Servlet 包装控制器(ServletWrappingController)

    Servlet 包装控制器内部封装了一个 Servlet 实例，内部封装的 Servlet 实例对外并不开放，对于程序的其他范围是不可见的。封装的 Servlet 实例有机会进行初始化和析构。Servlet 包装控制器适配所有的 HTTP 请求到内部封装的 Servlet 实例进行处理。它通常用于对已存 Servlet 的逻辑重用上。

  - Servlet 转发控制器(ServletForwardingController)

    Servlet 包装控制器将所有的 HTTP 请求转发给一个在 web.xml 中定义的 Servlet。Web 容器会对这个定义在 web.xml 的标准 Servlet 进行初始化和析构。

  下面我们将用一个表格来总结 Servlet 封装相关对象异同。

Servlet 封装对象	*管理范围*	*初始化和析构*	*服务调用方式*
SimpleServletHandlerAdaptor	Web 应用程序环境	没有	直接
ServletWrappingController	控制器内部	有	直接
ServletForwardingController	web.xml	有	Servlet 派遣器

- 视图转发控制器
  - URL 文件名视图控制器(UrlFilenameViewController)

    URL 文件名视图控制器通过将 URL 翻译成为视图名，并且返回。
  - 可参数化视图控制器(ParameterizableViewController)

    可参数化视图控制器简单的返回配置的视图名。

在前面一些小结中，分析了基于不同流程的实现，我们已经对简单 Form 控制器及其父类的实现进行了深入剖析和代码注释，这里将只对其他类进行剖析和代码注释。

抽象命令控制器继承自基本命令控制器。前面小结分析到，基本命令控制器提供了创建命令对象和通过 HTTP 请求参数对命令对象进行绑定和校验的功能方法。抽象命令控制器使用这些方法实现一个线性的流程，首先创建命令对象，然后绑定校验命令对象，最后传递命令对象到抽象的方法进行业务逻辑的处理。子类应该根据业务逻辑实现此方法。如下代码所示，

```
public abstract class AbstractCommandController extends BaseCommandController {

    public AbstractCommandController() {
    }

    public AbstractCommandController(Class commandClass) {
        setCommandClass(commandClass);
    }

    public AbstractCommandController(Class commandClass, String commandName) {
        setCommandClass(commandClass);
        setCommandName(commandName);
    }


    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {
```

```
        // 根据命令类型，创建命令对象
        Object command = getCommand(request);

        // 绑定命令对象到HTTP请求参数，并且校验
        ServletRequestDataBinder binder = bindAndValidate(request, command);

        // 构造绑定结果对象
        BindException errors = new BindException(binder.getBindingResult());

        // 调用业务逻辑处理方法
        return handle(request, response, command, errors);
    }

    // 子类根据业务逻辑实现此方法，它通常不适用于Form流程，它适用于一个简单的AJAX请求
    protected abstract ModelAndView handle(
            HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors)
            throws Exception;

}
```

可取消的 Form 控制器继承自简单的 Form 控制器。我们知道，简单 Form 控制器支持具有两种状态的 Form 流程，既显示 Form 视图状态和显示成功视图状态。可取消的 Form 控制器则增加了一个显示取消视图状态。如下代码所示，

```
public class CancellableFormController extends SimpleFormController {

    // 缺省的代表取消请求的参数键值
    private static final String PARAM_CANCEL = "_cancel";

    private String cancelParamKey = PARAM_CANCEL;

    private String cancelView;

    public final void setCancelParamKey(String cancelParamKey) {
        this.cancelParamKey = cancelParamKey;
    }

    public final String getCancelParamKey() {
        return this.cancelParamKey;
    }

    // 当接收到一个取消请求，显示这个取消视图
    public final void setCancelView(String cancelView) {
```

```
        this.cancelView = cancelView;
    }

    public final String getCancelView() {
        return this.cancelView;
    }


    @Override
    protected boolean isFormSubmission(HttpServletRequest request) {
        // 除了普通的Form提交请求，还包括具有取消参数的请求
        return super.isFormSubmission(request) || isCancelRequest(request);
    }

    @Override
    protected boolean suppressValidation(HttpServletRequest request, Object command) {
        // 取消请求不需要校验参数
        return super.suppressValidation(request, command) || isCancelRequest(request);
    }

    @Override
    protected ModelAndView processFormSubmission(
            HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors)
            throws Exception {

        // 特殊处理取消请求
        if (isCancelRequest(request)) {
            return onCancel(request, response, command);
        }
        else {
            return super.processFormSubmission(request, response, command, errors);
        }
    }

    protected boolean isCancelRequest(HttpServletRequest request) {
        // 存在取消参数，则是取消请求
        return WebUtils.hasSubmitParameter(request, getCancelParamKey());
    }

    protected ModelAndView onCancel(HttpServletRequest request, HttpServletResponse
response, Object command)
            throws Exception {
        // 代理到处理取消的处理器
```

```
        return onCancel(command);
    }

    protected ModelAndView onCancel(Object command) throws Exception {
        // 简单的返回取消视图
        return new ModelAndView(getCancelView());
    }

}
```

抽象导航控制器继承自抽象 Form 控制器。抽象 Form 控制器定义了两个状态，显示 Form 视图状态和显示成功视图状态。抽象导航控制器扩展了它的实现，拥有多个页面状态也一个取消视图状态和一个完成视图状态。它能够在不同的页面之间进行导航。它适用于一个模块有非常多的输入数据，以至于需要多个 Tab 页面进行输入。如下代码注释，

```
public abstract class AbstractWizardFormController extends AbstractFormController {

    // 代表完成请求的参数
    public static final String PARAM_FINISH = "_finish";

    // 代表取消请求的参数
    public static final String PARAM_CANCEL = "_cancel";

    // 代表导航到一个新页面的参数
    public static final String PARAM_TARGET = "_target";

    // 代表当前的页面，这个参数也存在在request或者session里
    public static final String PARAM_PAGE = "_page";

    // 所有的页面视图
    private String[] pages;

    private String pageAttribute;

    private boolean allowDirtyBack = true;

    private boolean allowDirtyForward = false;


    public AbstractWizardFormController() {
        // AbstractFormController sets default cache seconds to 0.
        super();

        // Always needs session to keep data from all pages.
```

```
        // 需要在Session中保存命令对象
        setSessionForm(true);

        // Never validate everything on binding ->
        // wizards validate individual pages.
        setValidateOnBinding(false);
    }

    public final void setPages(String[] pages) {
        // 至少有一个页面，第一个页面表现作为传统的Form视图
        if (pages == null || pages.length == 0)  {
            throw new IllegalArgumentException("No wizard pages defined");
        }
        this.pages = pages;
    }

    public final String[] getPages() {
        return this.pages;
    }

    protected final int getPageCount() {
        return this.pages.length;
    }

    public final void setPageAttribute(String pageAttribute) {
        this.pageAttribute = pageAttribute;
    }

    public final String getPageAttribute() {
        return this.pageAttribute;
    }

    public final void setAllowDirtyBack(boolean allowDirtyBack) {
        this.allowDirtyBack = allowDirtyBack;
    }

    public final boolean isAllowDirtyBack() {
        return this.allowDirtyBack;
    }

    public final void setAllowDirtyForward(boolean allowDirtyForward) {
        this.allowDirtyForward = allowDirtyForward;
    }
```

```
    public final boolean isAllowDirtyForward() {
        return this.allowDirtyForward;
    }

    @Override
    protected final void onBindAndValidate(HttpServletRequest request, Object command,
BindException errors)
        throws Exception {

        onBindAndValidate(request, command, errors, getCurrentPage(request));
    }

    // 针对不同的页面进行特殊的绑定和校验
    protected void onBindAndValidate(HttpServletRequest request, Object command,
BindException errors, int page)
        throws Exception {
    }

    @Override
    protected boolean isFormSubmission(HttpServletRequest request) {
        // 除了Form提交请求，完成请求和取消请求都被视为Form提交
        return super.isFormSubmission(request) || isFinishRequest(request) ||
isCancelRequest(request);
    }

    @Override
    protected final Map referenceData(HttpServletRequest request, Object command, Errors
errors)
        throws Exception {

        return referenceData(request, command, errors, getCurrentPage(request));
    }

    protected Map referenceData(HttpServletRequest request, Object command, Errors errors,
int page)
        throws Exception {

        return referenceData(request, page);
    }

    // 针对不同的页面有不同的引用数据
    protected Map referenceData(HttpServletRequest request, int page) throws Exception {
        return null;
    }
```

```
    @Override
    protected ModelAndView showForm(
            HttpServletRequest request, HttpServletResponse response, BindException
errors)
        throws Exception {

        // 第一次请求显示第一个页面，相当于传统的Form视图
        return showPage(request, errors, getInitialPage(request, errors.getTarget()));
    }

    protected final ModelAndView showPage(HttpServletRequest request, BindException
errors, int page)
        throws Exception {

        // 校验它是一个合法的页面
        if (page >= 0 && page < getPageCount(request, errors.getTarget())) {
            if (logger.isDebugEnabled()) {
                logger.debug("Showing wizard page " + page + " for form bean '" +
getCommandName() + "'");
            }

            // Set page session attribute, expose overriding request attribute.
            Integer pageInteger = new Integer(page);
            String pageAttrName = getPageSessionAttributeName(request);
            if (isSessionForm()) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Setting page session attribute [" + pageAttrName + "]
to: " + pageInteger);
                }
                request.getSession().setAttribute(pageAttrName, pageInteger);
            }
            request.setAttribute(pageAttrName, pageInteger);

            // Set page request attribute for evaluation by views.
            Map controlModel = new HashMap();
            if (this.pageAttribute != null) {
                controlModel.put(this.pageAttribute, new Integer(page));
            }

            // 取得具体某一页的视图名
            String viewName = getViewName(request, errors.getTarget(), page);

            // 显示视图
```

```
            return showForm(request, errors, viewName, controlModel);
        }

        else {
            throw new ServletException("Invalid wizard page number: " + page);
        }
    }

    protected int getPageCount(HttpServletRequest request, Object command) {
        return getPageCount();
    }

    // 每一个页面对应一个视图名，他们按照顺序存储
    protected String getViewName(HttpServletRequest request, Object command, int page) {
        return getPages()[page];
    }

    // 默认情况下，第一个页面就是初始化页面
    protected int getInitialPage(HttpServletRequest request, Object command) {
        return getInitialPage(request);
    }

    protected int getInitialPage(HttpServletRequest request) {
        return 0;
    }

    protected String getPageSessionAttributeName(HttpServletRequest request) {
        return getPageSessionAttributeName();
    }

    protected String getPageSessionAttributeName() {
        return getClass().getName() + ".PAGE." + getCommandName();
    }

    @Override
    protected ModelAndView handleInvalidSubmit(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {

        return showNewForm(request, response);
    }

    @Override
    protected final ModelAndView processFormSubmission(
```

```
            HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors)
            throws Exception {

        int currentPage = getCurrentPage(request);
        // Remove page session attribute, provide copy as request attribute.
        String pageAttrName = getPageSessionAttributeName(request);
        if (isSessionForm()) {
            if (logger.isDebugEnabled()) {
                logger.debug("Removing page session attribute [" + pageAttrName + "]");
            }
            request.getSession().removeAttribute(pageAttrName);
        }
        request.setAttribute(pageAttrName, new Integer(currentPage));

        // cancel?
        if (isCancelRequest(request)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Cancelling wizard for form bean '" + getCommandName() +
"'");
            }
            return processCancel(request, response, command, errors);
        }

        // finish?
        if (isFinishRequest(request)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Finishing wizard for form bean '" + getCommandName() + "'");
            }
            return validatePagesAndFinish(request, response, command, errors,
currentPage);
        }

        // Normal submit: validate current page and show specified target page.
        if (!suppressValidation(request, command, errors)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Validating wizard page " + currentPage + " for form bean
'" + getCommandName() + "'");
            }
            validatePage(command, errors, currentPage, false);
        }

        // Give subclasses a change to perform custom post-procession
        // of the current page and its command object.
```

```
        postProcessPage(request, command, errors, currentPage);

        int targetPage = getTargetPage(request, command, errors, currentPage);
        if (logger.isDebugEnabled()) {
            logger.debug("Target page " + targetPage + " requested");
        }
        if (targetPage != currentPage) {
            if (!errors.hasErrors() || (this.allowDirtyBack && targetPage < currentPage)
||
                    (this.allowDirtyForward && targetPage > currentPage)) {
                // Allowed to go to target page.
                return showPage(request, errors, targetPage);
            }
        }

        // Show current page again.
        return showPage(request, errors, currentPage);
    }

    protected int getCurrentPage(HttpServletRequest request) {
        // Check for overriding attribute in request.
        String pageAttrName = getPageSessionAttributeName(request);
        Integer pageAttr = (Integer) request.getAttribute(pageAttrName);
        if (pageAttr != null) {
            return pageAttr.intValue();
        }
        // Check for explicit request parameter.
        String pageParam = request.getParameter(PARAM_PAGE);
        if (pageParam != null) {
            return Integer.parseInt(pageParam);
        }
        // Check for original attribute in session.
        if (isSessionForm()) {
            pageAttr = (Integer) request.getSession().getAttribute(pageAttrName);
            if (pageAttr != null) {
                return pageAttr.intValue();
            }
        }
        throw new IllegalStateException(
                "Page attribute [" + pageAttrName + "] neither found in session nor in
request");
    }

    protected boolean isFinishRequest(HttpServletRequest request) {
```

```
        return WebUtils.hasSubmitParameter(request, PARAM_FINISH);
    }

    protected boolean isCancelRequest(HttpServletRequest request) {
        return WebUtils.hasSubmitParameter(request, PARAM_CANCEL);
    }

    protected int getTargetPage(HttpServletRequest request, Object command, Errors errors,
int currentPage) {
        return getTargetPage(request, currentPage);
    }

    protected int getTargetPage(HttpServletRequest request, int currentPage) {
        // 缺省情况下，_target参数代表要导航的目标页面
        return WebUtils.getTargetPage(request, PARAM_TARGET, currentPage);
    }

    private ModelAndView validatePagesAndFinish(
            HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors,
            int currentPage) throws Exception {

        // In case of binding errors -> show current page.
        if (errors.hasErrors()) {
            return showPage(request, errors, currentPage);
        }

        if (!suppressValidation(request, command, errors)) {
            // In case of remaining errors on a page -> show the page.
            for (int page = 0; page < getPageCount(request, command); page++) {
                validatePage(command, errors, page, true);
                if (errors.hasErrors()) {
                    return showPage(request, errors, page);
                }
            }
        }

        // No remaining errors -> proceed with finish.
        return processFinish(request, response, command, errors);
    }

    protected void validatePage(Object command, Errors errors, int page, boolean finish)
{
        validatePage(command, errors, page);
```

```
    }

    protected void validatePage(Object command, Errors errors, int page) {
    }

    // 子类根据业务逻辑改写
    protected void postProcessPage(HttpServletRequest request, Object command, Errors
errors, int page)
            throws Exception {
    }

    // 子类根据业务逻辑改写
    protected abstract ModelAndView processFinish(
            HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors)
            throws Exception;

    // 子类根据业务逻辑改写
    protected ModelAndView processCancel(
            HttpServletRequest request, HttpServletResponse response, Object command,
BindException errors)
            throws Exception {

        throw new ServletException(
                "Wizard form controller class [" + getClass().getName() + "] does not
support a cancel operation");
    }

}
```

Servlet 包装控制器简单的封装了一个内部的 Servlet 实例。如下代码所示，

```
public class ServletWrappingController extends AbstractController
    implements BeanNameAware, InitializingBean, DisposableBean {

    // 包装的Servlet类名，它必须是一个Servlet的完全实现类
    private Class servletClass;

    // Servlet的名称
    private String servletName;

    // Servlet的初始化参数
    private Properties initParameters = new Properties();
```

```
    // 可选的Bean名
    private String beanName;

    // 创建的内部Servlet实例
    private Servlet servletInstance;


    public void setServletClass(Class servletClass) {
        this.servletClass = servletClass;
    }


    public void setServletName(String servletName) {
        this.servletName = servletName;
    }

    public void setInitParameters(Properties initParameters) {
        this.initParameters = initParameters;
    }

    public void setBeanName(String name) {
        this.beanName = name;
    }

    // 初始化方法继承
    public void afterPropertiesSet() throws Exception {
        // servlet类必须存在，而且是Servlet的一个实现类
        if (this.servletClass == null) {
            throw new IllegalArgumentException("servletClass is required");
        }
        if (!Servlet.class.isAssignableFrom(this.servletClass)) {
            throw new IllegalArgumentException("servletClass [" +
this.servletClass.getName() +
                    "] needs to implement interface [javax.servlet.Servlet]");
        }

        // 如果Servlet名不存在，使用Bean名
        if (this.servletName == null) {
            this.servletName = this.beanName;
        }

        // 创建一个Servlet实例
        this.servletInstance = (Servlet) this.servletClass.newInstance();

```

```
        // 模拟调用Servlet的初始化方法
        this.servletInstance.init(new DelegatingServletConfig());
    }

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
        throws Exception {

        // 适配Servlet的服务方法，用于重用已存的Servlet实现的逻辑
        this.servletInstance.service(request, response);
        return null;
    }


    public void destroy() {
        // 模拟调用析构方法
        this.servletInstance.destroy();
    }


    private class DelegatingServletConfig implements ServletConfig {

        public String getServletName() {
            // 配置的Sevlet名或者Bean名
            return servletName;
        }

        public ServletContext getServletContext() {
            // 过Web应用程序环境传递进来的真正的Sevlet环境
            return ServletWrappingController.this.getServletContext();
        }

        public String getInitParameter(String paramName) {
            // 可配置的初始化参数
            return initParameters.getProperty(paramName);
        }

        public Enumeration getInitParameterNames() {
            return initParameters.keys();
        }
    }

}
```

Servlet 转发控制器通过 Servlet 转发派遣器转发 HTTP 请求到一个标准的 web.xml 定义的 Servlet 组件。如下代码所示，

```
public class ServletForwardingController extends AbstractController implements
BeanNameAware {

    // Servlet名称
    private String servletName;

    // 如果没有制定Servlet名称，则使用Bean名称
    private String beanName;

    public void setServletName(String servletName) {
        this.servletName = servletName;
    }

    public void setBeanName(String name) {
        this.beanName = name;
        if (this.servletName == null) {
            this.servletName = name;
        }
    }


    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response)
            throws Exception {

        // 取得Servlet的转发派遣器
        RequestDispatcher rd =
getServletContext().getNamedDispatcher(this.servletName);
        if (rd == null) {
            throw new ServletException("No servlet with name '" + this.servletName + "'
defined in web.xml");
        }
        // If already included, include again, else forward.
        if (useInclude(request, response)) {
            rd.include(request, response);
            if (logger.isDebugEnabled()) {
                logger.debug("Included servlet [" + this.servletName +
                        "] in ServletForwardingController '" + this.beanName + "'");
            }
```

```
        }
        else {
            // 使用容器提供的请求派遣功能，转发HTTP请求到Servlet进行处理
            rd.forward(request, response);
            if (logger.isDebugEnabled()) {
                logger.debug("Forwarded to servlet [" + this.servletName +
                        "] in ServletForwardingController '" + this.beanName + "'");
            }
        }
        return null;
    }

   protected boolean useInclude(HttpServletRequest request, HttpServletResponse response)
{
        // 如果它是显示的包含请求或者它已经设置的响应状态代码
        return (WebUtils.isIncludeRequest(request) || response.isCommitted());
    }

}
```

URL 文件名视图控制器继承自抽象 URL 视图控制器。抽象 URL 视图控制器通过 URL 决定视图名称，并且返回包含此视图的模型和视图对象。然后，具体如何映射 URL 到视图名，子类需要根据业务逻辑进行实现。如下代码所示，

```
public abstract class AbstractUrlViewController extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response) {
        // 取得查找路径，用于日志
        String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);

        // 根据请求URL决定视图名
        String viewName = getViewNameForRequest(request);
        if (logger.isDebugEnabled()) {
            logger.debug("Returning view name '" + viewName + "' for lookup path [" +
lookupPath + "]");
        }
        return new ModelAndView(viewName);
    }

    //子类根据业务逻辑实现映射过程
    protected abstract String getViewNameForRequest(HttpServletRequest request);
```

```
}

public class UrlFilenameViewController extends AbstractUrlViewController {
    protected String getViewNameForRequest(HttpServletRequest request) {
        // 提取基于请求映射的路径或者查找路径
        String uri = extractOperableUrl(request);
        return getViewNameForUrlPath(uri);
    }

    protected String extractOperableUrl(HttpServletRequest request) {
        // 首先使用基于请求映射的路径
        String urlPath = (String)
request.getAttribute(HandlerMapping.PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE);

        // 如果没有基于请求映射的路径，则使用查找路径
        if (!StringUtils.hasText(urlPath)) {
            urlPath = getUrlPathHelper().getLookupPathForRequest(request);
        }
        return urlPath;
    }

    protected String getViewNameForUrlPath(String uri) {
        String viewName = this.viewNameCache.get(uri);
        if (viewName == null) {
            // 从URI中提取视图名
            viewName = extractViewNameFromUrlPath(uri);
            // 添加配置的前缀和后缀
            viewName = postProcessViewName(viewName);
            this.viewNameCache.put(uri, viewName);
        }
        return viewName;
    }

    protected String extractViewNameFromUrlPath(String uri) {
        // 去除前缀/好后缀.*
        int start = (uri.charAt(0) == '/' ? 1 : 0);
        int lastIndex = uri.lastIndexOf(".");
        int end = (lastIndex < 0 ? uri.length() : lastIndex);
        return uri.substring(start, end);
    }

    protected String postProcessViewName(String viewName) {
        return getPrefix() + viewName + getSuffix();
    }
```

}

可参数化视图控制器的代码非常简单，这里不再进行代码注释。

4.2.2.3.2注解控制器

上一小结中，我们剖析了简单控制器的实现体系结构。事实上，自从 Spring 2.5 引进了注解方法控制器, 已经不推荐使用简单控制器的实现体系结构中的基于 Form 的实现类。

注解控制器是通过在一个简单的 Java 对象上声明具有元信息的注解来实现的。所以，注解控制器并没有接口，抽象类或者实体类的实现体系结构。然而，声明在简单的 Java 对象上的注解则扮演着至关重要的角色。

下面我们根据不同类型的注解进行剖析，

1. 标识控制器和控制器方法的注解

   @Controller

   一个注解控制器是通过注解@Controller 标识的。注解@Controller 比注解@Component 具有更具体的语义，代表它不仅是应用程序环境中的一个 Bean，而且还是一个注解控制器类型。注解@Component 还有两个其他具有更具体语义的注解，@Repository 和 @Service。

   注解@Componet 和它具有更具体语义的注解都是通过 XML 环境中的标记 <context:component-scan/>加载进入应用程序环境的。

   @RequestMapping

   注解控制器的处理器方法是通过注解@RequestMapping 映射到一个适当的 HTTP 请求的。在注解@RequestMapping 中，可以声明 HTTP 方法，HTTP 参数，HTTP 头信息以及更加重要的 URL Pattern 信息。这些信息全部是用来匹配一个 HTTP 请求的，如果匹配成功，则用当前声明注解@RequestMapping 的处理器方法来处理当前 HTTP 请求。

   注解@RequestMapping 既能够声明在注解控制器类级别也能生命在注解控制器的处理器方法中。如果存在类级别的注解@RequestMapping，它将会结合方法级别的注解 @RequestMapping 一起参与匹配 HTTP 请求。

2. 声明初始化数据绑定方法的注解

   @InitBinder

   注解@InitBinder 用于声明一个初始化数据绑定对象的方法，数据绑定对象用于对一个

业务逻辑对象进行赋值的辅助对象。在一个处理器方法参数没有任何注解声明的时候，它就是一个需要绑定的业务逻辑对象，在调用处理器方法之前，需要使用 Web 数据绑定对象对此业务逻辑对象进行绑定，注解@InitBinder 就是用来给 Web 数据绑定进行初始化的。

3. 声明模型属性相关注解

@SessionAttribute

注解@SessionAttribute 可以声明保存某些模型对象到 Session 范围内。这些 Session 范围内的对象可以在接下来的请求中继续可得。

@ModelAttribute

注解@ModelAttribute 有两个用途，它可以用来声明一个方法的返回值作为模型属性。它也能用来声明某个处理器方法参数能够从某个模型属性中得到期望的值。

4. 注解控制器处理器方法参数和返回值注解

前面我们已经在基于流程的剖析的章节中，对注解控制器的处理器方法的参数和返回值的注解进行了详细的分析和代码注释。这里将不再重复分析。

最后我们对一个典型的注解控制器进行分析，如下代码注释，

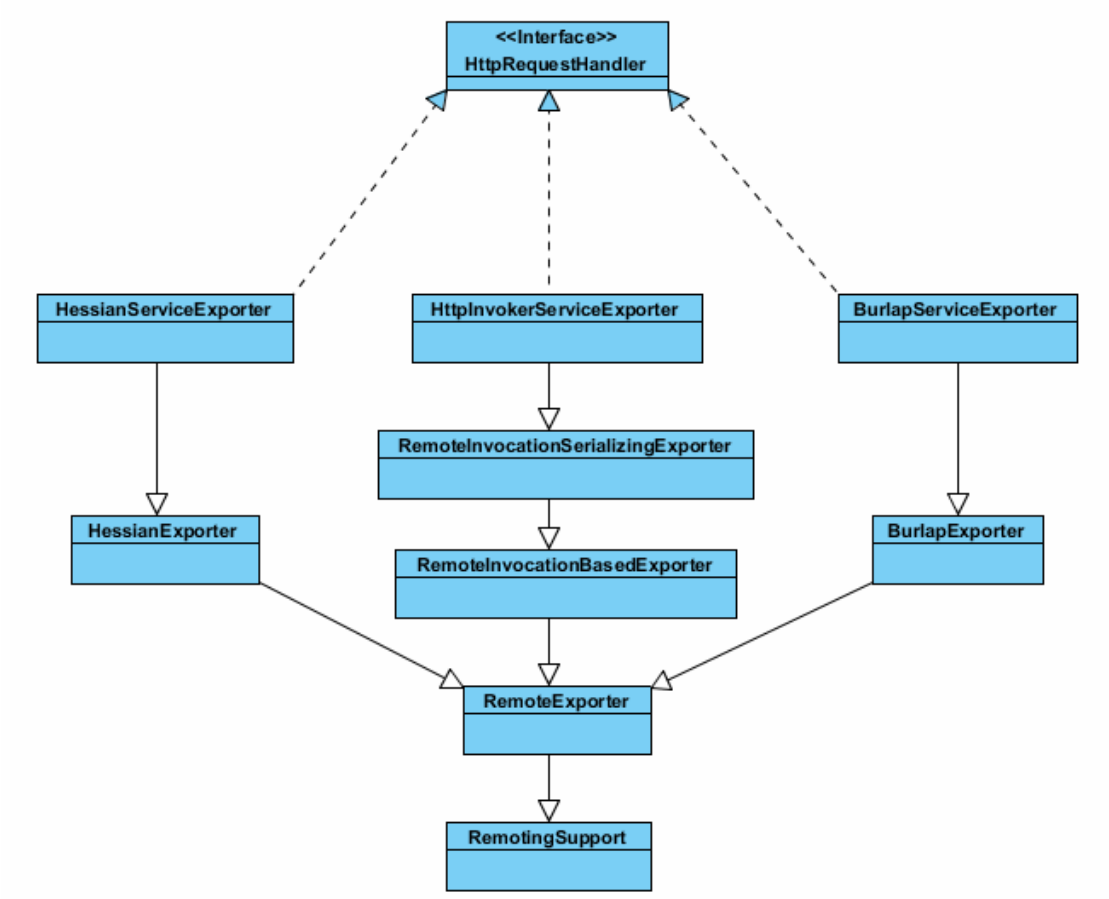
```
// 声明作为注解控制器组件
@Controller

// 映射带有模板参数的URL到所有的当前控制器的处理器方法中
@RequestMapping("/owners/{ownerId}/pets/new")

// 将pet对象存储到Session范围内
@SessionAttributes("pet")
public class AddPetForm {
    // 引用持久层业务逻辑服务对象，注解控制器推荐使用控制器对持久层业务逻辑的直接调用，在不复杂的情况下，不再需要服务层和DAO层次
    private final Clinic clinic;

    // 自动注入持久层业务逻辑服务对象
    @Autowired
    public AddPetForm(Clinic clinic) {
        this.clinic = clinic;
    }

    // 声明模型属性，这些模型属性会在视图显示中使用
```

```
    @ModelAttribute("types")
    public Collection<PetType> populatePetTypes() {
        return this.clinic.getPetTypes();
    }

    // 用来初始化绑定Pet对象
    @InitBinder
    public void setAllowedFields(WebDataBinder dataBinder) {
        dataBinder.setDisallowedFields("id");
    }

    // HTTP GET请求用来显示Form
    @RequestMapping(method = RequestMethod.GET)
    // 从路径变量中提取OWNER ID
    public String setupForm(@PathVariable("ownerId") int ownerId, Model model) {
        // 取得Owner信息
        Owner owner = this.clinic.loadOwner(ownerId);

        // 创建一个Pet对象加入到Owner中
        Pet pet = new Pet();
        owner.addPet(pet);

        // 加Pet对象到模型中，并且Pet对象是Session属性，所以，它也会保存在Session中
        model.addAttribute("pet", pet);
        return "pets/form";
    }

    // HTTP POST请求用来创建Pet对象
    @RequestMapping(method = RequestMethod.POST)
    // Pet对象在显示Form的请求中存储在Session中，所以，它在模型中也是可得
    public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result,
SessionStatus status) {
        // 对Pet对象绑定和校验
        new PetValidator().validate(pet, result);

        // 如果出错继续显示Form视图
        if (result.hasErrors()) {
            return "pets/form";
        }
        // 如果成功则创建Pet对象
        else {
            this.clinic.storePet(pet);

            // 结束一个Session周期，清楚Session中保存的属性
```

```
            status.setComplete();

            // 成功后显示Owner视图
            return "redirect:/owners/" + pet.getOwner().getId();
        }
    }

}
```

4.2.2.3.3 HTTP请求处理器

HTTP 请求处理器是用来实现基于 HTTP 请求的远程调用的处理器。在前面章节基于流程的分析中，我们已经对典型的 HTTP 请求处理器进行了详细的分析和代码注释。事实上，除了 HTTP 调用服务导出器(HttpInvokerServiceExporter)的实现，还有更多的基于 HTTP 请求的远程调用的实现。如下类图所示，



图表 4-36

如上图所示，除了 HTTP 调用服务导出器以外，还有两个远程调用的实现类，他们是 Hessian 服务导出器和 Burlap 服务导出器，他们的实现流程和 HTTP 调用服务导出器的实现流程非

常类似，区别在于他们并没有在 HTTP 协议上传递远程调用对象和远程调用对象结果的序列化数据，而是使用他们专用的在 HTTP 协议上传输的数据格式来实现远程调用。

Hessian 通过在 HTTP 协议上传输二进制的数据来实现远程调用。Burlap 通过在 HTTP 协议上传输 XML 数据来实现远程调用。我们不讲在本书对他们本身的实现进行分析。

4.2.2.4拦截器的实现架构

从前面的章节分析中得知，处理器映射机制支持处理器拦截器功能。处理器拦截器应用一定的功能在满足一定条件的请求上。

处理器拦截器必须实现 HandlerInterceptor 接口，它定义了三个方法如下，

preHandle()

任何处理器调用之前调用的方法。它返回一个布尔值，如果返回真，则继续调用处理器链的其他处理器或者拦截器，如果返回假，则停止调用处理器链的其他处理器或者拦截器，在这种情况下，它假设拦截器已经处理了 HTTP 请求，而且写入了 HTTP 响应。

postHandle()

任何处理器调用之后调用的方法。

afterCompletion()

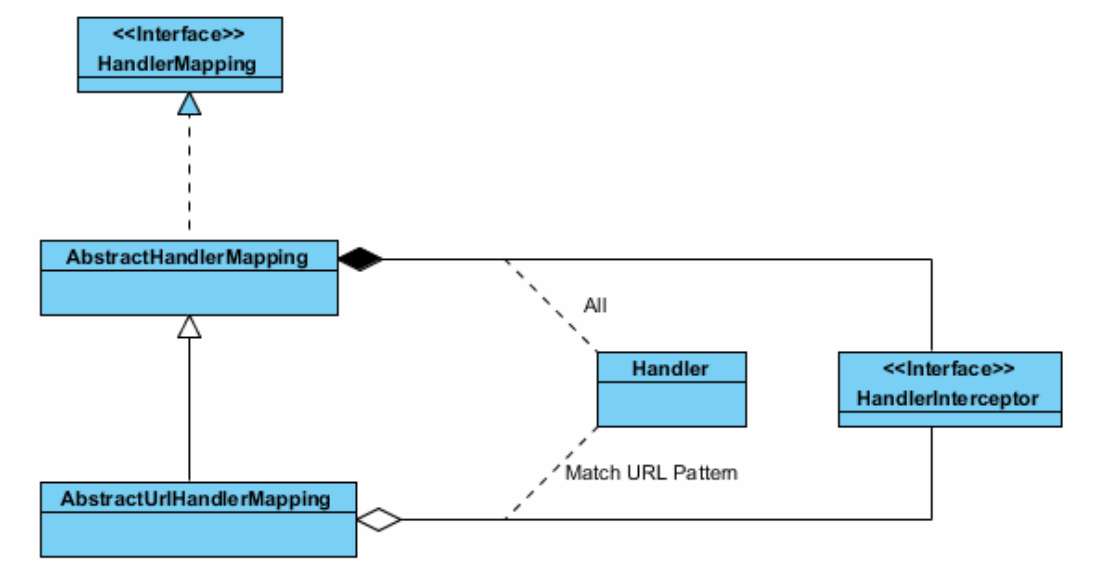
整个请求处理之后调用的方法。

如下代码注释，

```
public interface HandlerInterceptor {

    // 处理器执行之前调用的方法，可以用来对处理器进行预处理，如果返回布尔值假，则终止处理请求
    boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler)
        throws Exception;

    // 处理器执行之后调用的方法，可以用来对处理器进行后置处理
    void postHandle(
            HttpServletRequest request, HttpServletResponse response, Object handler,
ModelAndView modelAndView)
            throws Exception;

    // 当整个请求完成时候（成功或者失败）的方法，如果有任何异常产生，则通过异常参数传入
    void afterCompletion(
```

```
            HttpServletRequest request, HttpServletResponse response, Object handler,
Exception ex)
            throws Exception;

}
```

在对流程进行分析的时候，我们看到有两个层次的处理器拦截器的实现。在抽象的处理器映射的层次上可以配置处理器拦截器。这些拦截器会应用到所有的处理器上。然后，在抽象 URL 处理器映射的层次上可以根据 URL Pattern 配置处理器拦截器，这些拦截器只应用到匹配 URL Pattern 的处理器上。如下图所示，



图表 4-37

在抽象处理器映射类中，它声明了一个处理器拦截器的数组，但是并没有提供这个数组的存取方法。这个处理器映射是通过另外一个数组属性适配得到的。如下代码所示，

```
public abstract class AbstractHandlerMapping extends WebApplicationObjectSupport
        implements HandlerMapping, Ordered {
    // 通用类型的拦截器对象
    // 包含处理器拦截器，它用标准Servlet HTTP请求对象和标准Servlet HTTP响应对象作为方法参数
    // 也包含Web Request拦截器， 它使用Spring对请求和响应的封装类型， WebRequest和WebResponse
    private final List<Object> interceptors = new ArrayList<Object>();

    // 适配后的处理器适配器
    private HandlerInterceptor[] adaptedInterceptors;

    // 提供方法设置HandlerInterceptor和WebRequestInterceptor
    public void setInterceptors(Object[] interceptors) {
        this.interceptors.addAll(Arrays.asList(interceptors));
```

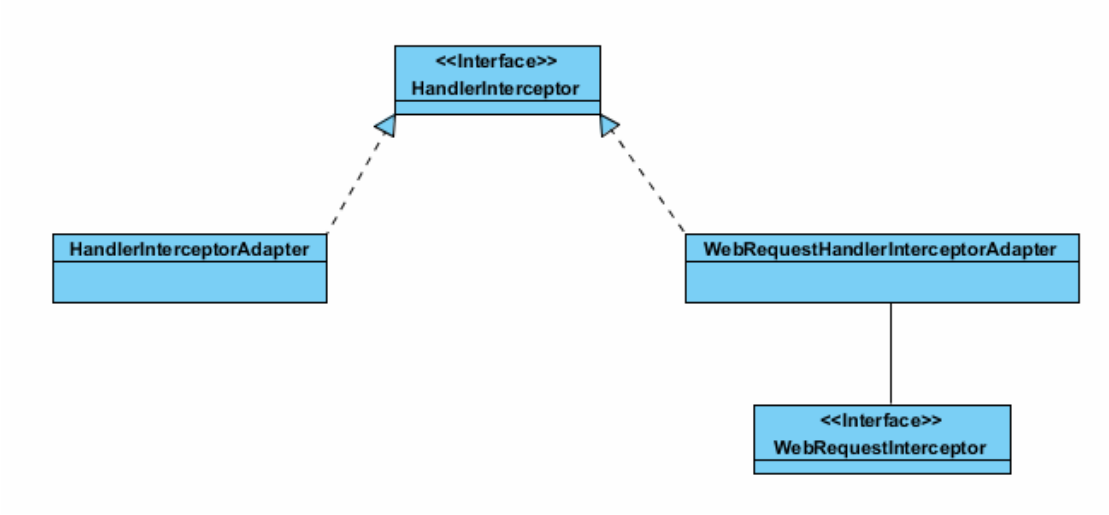
```
    }

    // 当应用程序环境初始化的时候，初始化拦截器
    protected void initApplicationContext() throws BeansException {
        extendInterceptors(this.interceptors);
        initInterceptors();
    }

    // 子类可以改写添加新的拦截器
    protected void extendInterceptors(List<Object> interceptors) {
    }


    protected void initInterceptors() {
        if (!this.interceptors.isEmpty()) {
            this.adaptedInterceptors = new
HandlerInterceptor[this.interceptors.size()];
            // 对于每个对象类型的拦截器适配到标准的处理器拦截器
            for (int i = 0; i < this.interceptors.size(); i++) {
                Object interceptor = this.interceptors.get(i);
                if (interceptor == null) {
                    throw new IllegalArgumentException("Entry number " + i + " in
interceptors array is null");
                }
                this.adaptedInterceptors[i] = adaptInterceptor(interceptor);
            }
        }
    }

    protected HandlerInterceptor adaptInterceptor(Object interceptor) {
        // 如果已经是处理器拦截器类型，不需要适配
        if (interceptor instanceof HandlerInterceptor) {
            return (HandlerInterceptor) interceptor;
        }
        // 如果是Web请求拦截器，需要使用WebRequestHandlerInterceptorAdapter进行适配
        else if (interceptor instanceof WebRequestInterceptor) {
            return new WebRequestHandlerInterceptorAdapter((WebRequestInterceptor)
interceptor);
        }
        // 不支持其他类型的拦截器
        else {
            throw new IllegalArgumentException("Interceptor type not supported: " +
interceptor.getClass().getName());
        }
```

```
    }

}
```

我们可以看到，它支持两种类型的拦截器，处理器拦截器和 Web 请求拦截器。处理器拦截器使用标准的 HTTP 请求和 HTTP 响应作为参数，而 Web 请求拦截器使用了 Spring 封装类型 WebRequest 和 WebResponse。这是因为除了 Servlet 规范中的请求和响应以外，它还要支持 Portlet 中的请求和响应。如下 Web 请求拦截器代码注释，

```
public interface WebRequestInterceptor {

    // 处理器执行前调用，不支持在拦截器中结束处理器链的操作
    void preHandle(WebRequest request) throws Exception;

    // 处理器执行后调用
    void postHandle(WebRequest request, ModelMap model) throws Exception;

    // 请求处理完成后调用
    void afterCompletion(WebRequest request, Exception ex) throws Exception;

}
```

有的时候，我们并不希望实现所有的三个方法，而是希望实现其中某一个，或者某些方法，Spring 框架提供了处理器拦截器的适配器类来达到这样的目的。如下类图所示，



图表 4-38

上图中，处理器拦截器适配器实现了处理器拦截器接口，子类只需要继承自处理器拦截器适配器并且根据业务逻辑需要改写其中的某些方法即可。

Web 请求处理器拦截器适配器用来将一个 Web 请求拦截器适配成为一个标准的处理器拦截器。它应用在抽象处理器拦截器的内部实现上。

下面我们分析抽象 URL 处理器映射层次的处理器拦截器。这个层次的拦截器根据配置的 URL Pattern 应用到一部分的处理器上。如下代码所示，

```
public abstract class AbstractUrlHandlerMapping extends AbstractHandlerMapping {

    // 映射拦截器集合，映射拦截器保存从URL Pattern到处理器拦截器的映射信息
    private MappedInterceptors mappedInterceptors;

    // 可以在外部进行配置映射的拦截器
    public void setMappedInterceptors(MappedInterceptor[] mappedInterceptors) {
        this.mappedInterceptors = new MappedInterceptors(mappedInterceptors);
    }


    @Override
    // 从Web应用程序环境中查找映射的拦截器
    protected void initInterceptors() {
        super.initInterceptors();

        // 查找Web应用程序环境中所有的映射的拦截器
        Map<String, MappedInterceptor> mappedInterceptors =
BeanFactoryUtils.beansOfTypeIncludingAncestors(
                getApplicationContext(), MappedInterceptor.class, true, false);
        if (!mappedInterceptors.isEmpty()) {
            this.mappedInterceptors = new
MappedInterceptors(mappedInterceptors.values().toArray(
                    new MappedInterceptor[mappedInterceptors.size()]));
        }

    }
}
```

映射的拦截器是一个保存从 URL Pattern 到处理器拦截器的映射信息。如下代码所示，

```
public final class MappedInterceptor {

    // 匹配的URL Pattern
    private final String[] pathPatterns;

    // 匹配的URL Pattern应该应用的处理器拦截器
    private final HandlerInterceptor interceptor;
```

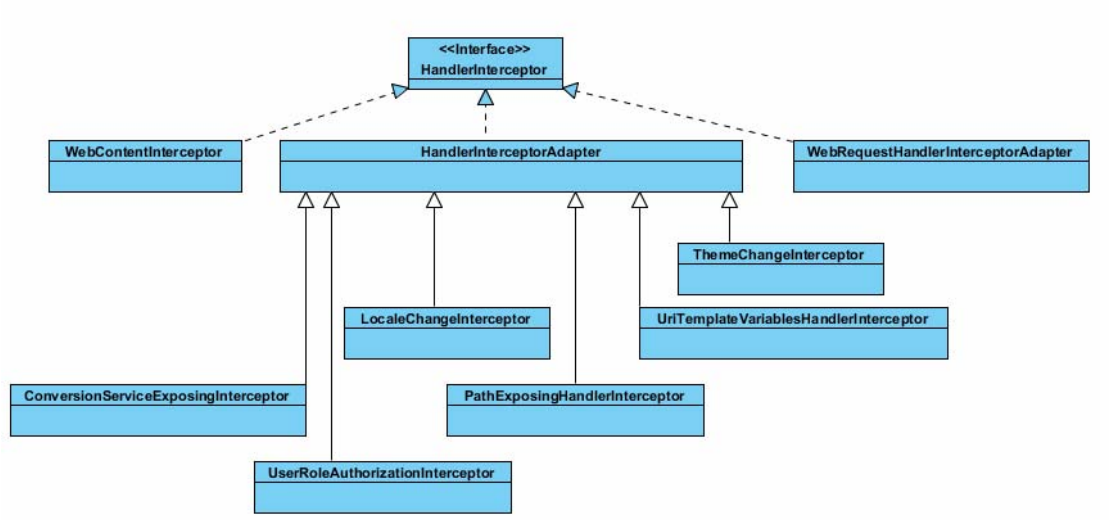
```
    //省略了构造器和存取方法
}
```

有了这些映射信息，处理器映射在构造处理器执行链的时候，就会使用这些信息进行匹配当前的 HTTP 请求，如果匹配成功，则使用此处理器拦截器。过滤功能是在映射的拦截器集合类中实现的。如下代码注释，

```
class MappedInterceptors {
    // 所有可得的映射的拦截器
    private MappedInterceptor[] mappedInterceptors;

    public MappedInterceptors(MappedInterceptor[] mappedInterceptors) {
        this.mappedInterceptors = mappedInterceptors;
    }

    // 通过请求的查找路径进行过滤
    public Set<HandlerInterceptor> getInterceptors(String lookupPath, PathMatcher
pathMatcher) {
        Set<HandlerInterceptor> interceptors = new LinkedHashSet<HandlerInterceptor>();
        for (MappedInterceptor interceptor : this.mappedInterceptors) {
            // 如果映射的拦截器匹配查找路径，则使用此映射的拦截器
            if (matches(interceptor, lookupPath, pathMatcher)) {
                interceptors.add(interceptor.getInterceptor());
            }
        }
        return interceptors;
    }

    private boolean matches(MappedInterceptor interceptor, String lookupPath, PathMatcher
pathMatcher) {
        // 一个映射的拦截器包含有多个URL Pattern，如果其中一个URL Pattern匹配查找路径，则匹配成功
        String[] pathPatterns = interceptor.getPathPatterns();
        if (pathPatterns != null) {
            for (String pattern : pathPatterns) {
                if (pathMatcher.match(pattern, lookupPath)) {
                    return true;
                }
            }
            return false;
        } else {
            return true;
        }
    }
```

```
}
```

Spring 框架实现了一些通用的处理器拦截器。其中，每个拦截器的实现都实现了一个独立完整的逻辑功能。如下类图所示，



图表 4-39

我们已经在前面分析了适配器 WebRequestHandlerInterceptorAdapter 和 HandlerInterceptorAdapter 的用途。这里我们将对剩余的类进行功能性的描述，既然，他们的实现并不复杂，这里将不再进行代码注释。

- WebContentInterceptor

  它能用于根据配置设置用户的 HTTP 缓存，并且对 HTTP 请求进行简单的方法校验。事实上它是在处理器调用之前对 WebContentGenerator 的实现进行调用。

- ConversionServiceExposingInterceptor

  它在处理器调用之前导出配置的转换服务到 HTTP 请求属性中。

- UserRoleAuthorizationInterceptor

  它用于限制一类控制器仅仅可以在一部分的角色中使用。角色校验是在处理器调用之前完成的。

- LocaleChangeInterceptor

  它在处理器调用之前根据请求的地域信息配置地域解析器。

- ThemeChangeInterceptor

它在处理器调用之前根据请求的主题信息配置主题解析器。

- PathExposingHandlerInterceptor
  处理器内部使用用于导出处理器匹配的路径信息到请求属性中。

- UriTemplateVariablesHandlerInterceptor

  处理器内部使用用于导出处理器匹配的路径信息中的模板变量到请求属性中。

4.2.2.5HTTP消息转换器的实现架构

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

本章节将包含下面的内容。

- @RequestBody
- @ResponseBody
- @HttpEntity
- @ResponseEntity
- HttpMessageConverter 实现体系结构

4.3 视图解析和视图显示

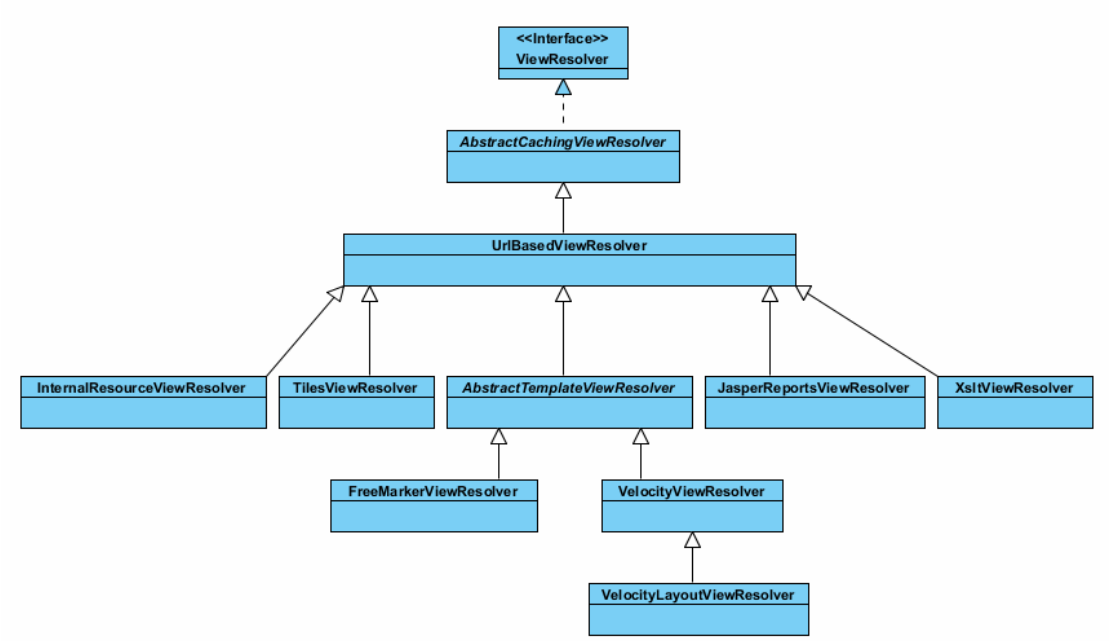
上一节，深入的剖析了作为总控制器的派遣器 Servlet 如何通过处理器映射查找处理器，并且通过处理器适配器进行适配调用处理器实现的业务逻辑服务，进而返回逻辑视图名和模型数据对象。

这一节，我们将分析作为总控制器的派遣器 Servlet 如何解析视图和显示视图。这是 Spring Web MVC 流程中最后一个关键步骤。

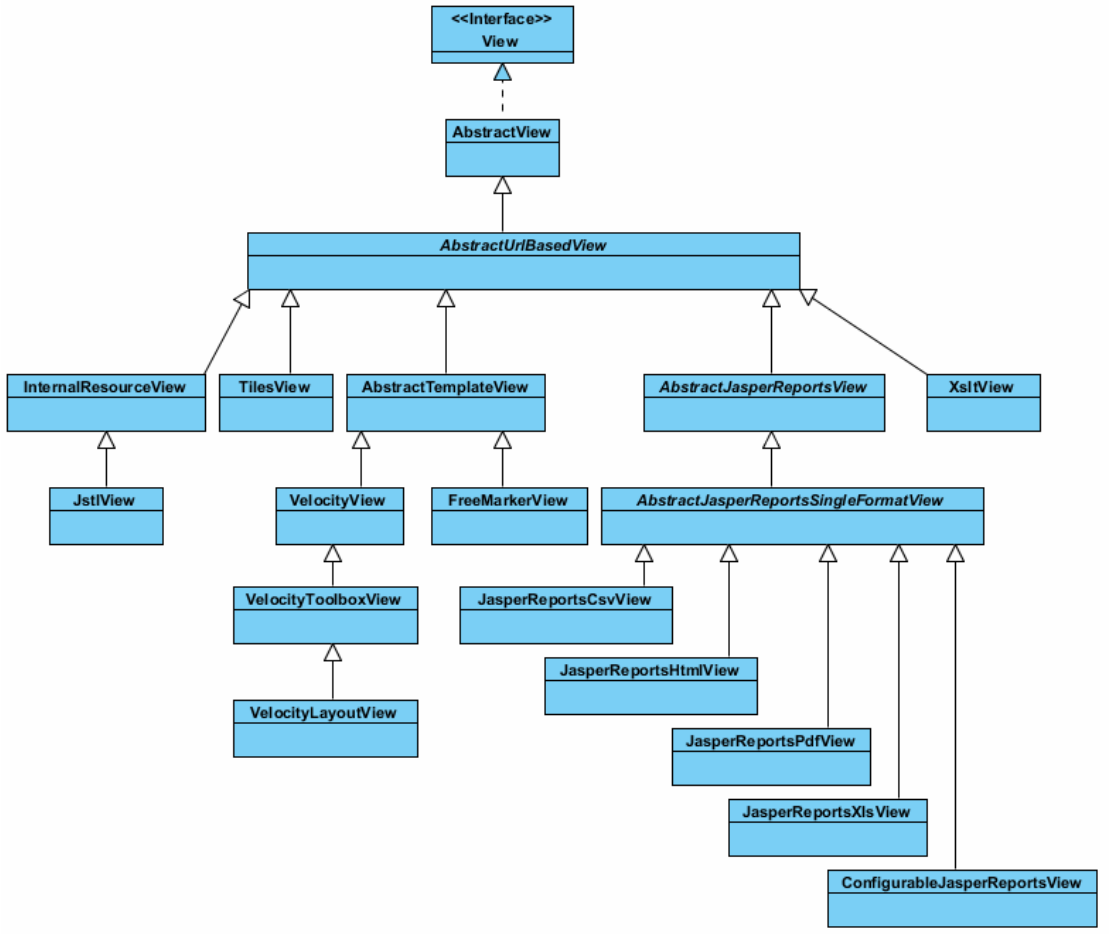
由于视图显示技术的多样性，存在很多视图解析器和视图的实现，这一节我们将分析典型的视图解析器和视图显示的实现。他们是基于 URL 的视图解析器，然后，对于其他更多的视图解析器和视图的实现进行剖析。

4.3.1 基于URL的视图解析器和视图

基于 URL 的视图解析器是视图解析器接口的简单实现，它不需要显示的映射定义，而是直接的把逻辑视图名解析成为资源 URL。它通常使用在基于 URL 的简单的视图显示技术上，并且逻辑视图名是 URL 的一部分。如下视图解析器和视图的类图所示，



图表 4-40



图表 4-41

抽象缓存视图解析器是视图解析器的直接的抽象实现类。它实现了对视图的内存缓存。如下代码所示，

```java
public abstract class AbstractCachingViewResolver extends WebApplicationObjectSupport
implements ViewResolver {
    // 设置缓存是否开启
    private boolean cache = true;

    // 缓存视图的映射对象
    private final Map<Object, View> viewCache = new HashMap<Object, View>();

    public void setCache(boolean cache) {
        this.cache = cache;
    }

    public boolean isCache() {
        return this.cache;
    }

    // 解析视图方法的逻辑实现
    public View resolveViewName(String viewName, Locale locale) throws Exception {
        // 如果视图缓存关闭，则每次都创建视图
        if (!isCache()) {
            return createView(viewName, locale);
        }
        // 如果视图缓存打开
        else {
            // 首先取得缓存中存储视图的关键字对象
            Object cacheKey = getCacheKey(viewName, locale);
            // 需要同步存取
            synchronized (this.viewCache) {
                // 检查缓存中是否已经存在视图
                View view = this.viewCache.get(cacheKey);
                if (view == null) {
                    // Ask the subclass to create the View object.
                    view = createView(viewName, locale);
                    // 将创建的视图加入到缓存中
                    this.viewCache.put(cacheKey, view);
                    if (logger.isTraceEnabled()) {
                        logger.trace("Cached view [" + cacheKey + "]");
                    }
                }
                // 返回从缓存中获得的视图或者新创建的视图
                return view;
            }
        }
```

```
    }

    protected Object getCacheKey(String viewName, Locale locale) {
        // 使用逻辑视图名和地域作为视图的关键字
        return viewName + "_" + locale;
    }

    // 提供功能移除缓存的视图
    public void removeFromCache(String viewName, Locale locale) {
        if (!this.cache) {
            logger.warn("View caching is SWITCHED OFF -- removal not necessary");

        }
        else {
            Object cacheKey = getCacheKey(viewName, locale);
            Object cachedView;
            synchronized (this.viewCache) {
                cachedView = this.viewCache.remove(cacheKey);
            }
            if (cachedView == null) {
                // Some debug output might be useful...
                if (logger.isDebugEnabled()) {
                    logger.debug("No cached instance for view '" + cacheKey + "' was
found");
                }
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Cache for view " + cacheKey + " has been cleared");
                }
            }
        }
    }

    // 提供功能清除所有缓存的视图
    public void clearCache() {
        logger.debug("Clearing entire view cache");
        synchronized (this.viewCache) {
            this.viewCache.clear();
        }
    }

    protected View createView(String viewName, Locale locale) throws Exception {
        return loadView(viewName, locale);
```

```
    }

    // 让子类根据具体实现创建不同的视图
    protected abstract View loadView(String viewName, Locale locale) throws Exception;

}
```

视图解析器实现体系结构中下一个层次是基于 URL 的视图解析器，它除了根据重定向前缀和转发前缀构造一个重定向视图和转发视图以外，它还为每个通常的 URL 创建一个抽象的基于 URL 视图的子类对象并且返回。如下代码注释，

```
public class UrlBasedViewResolver extends AbstractCachingViewResolver implements Ordered
{
    // 重定向前缀
    public static final String REDIRECT_URL_PREFIX = "redirect:";

    // 转发前缀
    public static final String FORWARD_URL_PREFIX = "forward:";

    // 视图类名，应该是抽象的基于URL视图的子类
    private Class viewClass;

    // URL前缀
    private String prefix = "";

    // URL后缀
    private String suffix = "";

    // 能够处理的视图逻辑名字集合，如果为空，任何视图逻辑名都可以处理
    private String[] viewNames = null;

    // 支持的内容类型
    private String contentType;

    // 以/开头的URL是相对于Web Server根还是应用程序根
    private boolean redirectContextRelative = true;

    // 使用HTTP 1.0兼容还是使用HTTP1.1兼容，不同版本的HTTP协议重定向方法不同
    private boolean redirectHttp10Compatible = true;

    // 保存请求环境属性的关键字
    private String requestContextAttribute;

    // 最大顺序
```

```
    private int order = Integer.MAX_VALUE;

    // 静态的属性，应用到所有的解析的视图上
    private final Map<String, Object> staticAttributes = new HashMap<String, Object>();

    // 设置视图类，需要的视图必须是抽象的基于URL的视图
    public void setViewClass(Class viewClass) {
        if (viewClass == null || !requiredViewClass().isAssignableFrom(viewClass)) {
            throw new IllegalArgumentException(
                    "Given view class [" + (viewClass != null ? viewClass.getName() :
null) +
                    "] is not of type [" + requiredViewClass().getName() + "]");
        }
        this.viewClass = viewClass;
    }

    protected Class requiredViewClass() {
        return AbstractUrlBasedView.class;
    }

    @Override
    protected void initApplicationContext() {
        super.initApplicationContext();
        // 初始化是检查必要的是视图类
        if (getViewClass() == null) {
            throw new IllegalArgumentException("Property 'viewClass' is required");
        }
    }

    @Override
    protected Object getCacheKey(String viewName, Locale locale) {
        // 仅仅使用视图逻辑名作为缓存的关键字
        return viewName;
    }

    @Override
    protected View createView(String viewName, Locale locale) throws Exception {
        // If this resolver is not supposed to handle the given view,
        // return null to pass on to the next resolver in the chain.
        if (!canHandle(viewName, locale)) {
            return null;
        }
        // Check for special "redirect:" prefix.
        if (viewName.startsWith(REDIRECT_URL_PREFIX)) {
```

```
            String redirectUrl = viewName.substring(REDIRECT_URL_PREFIX.length());
            return new RedirectView(redirectUrl, isRedirectContextRelative(),
isRedirectHttp10Compatible());
        }
        // Check for special "forward:" prefix.
        if (viewName.startsWith(FORWARD_URL_PREFIX)) {
            String forwardUrl = viewName.substring(FORWARD_URL_PREFIX.length());
            return new InternalResourceView(forwardUrl);
        }
        // Else fall back to superclass implementation: calling loadView.
        return super.createView(viewName, locale);
    }

    protected boolean canHandle(String viewName, Locale locale) {
        // 如果声明了可以处理的视图结合，则进行匹配，如果匹配成功，则能够处理，否则，不能处理当前逻辑
视图
        String[] viewNames = getViewNames();
        return (viewNames == null || PatternMatchUtils.simpleMatch(viewNames, viewName));
    }

    @Override
    protected View loadView(String viewName, Locale locale) throws Exception {
        // 创建抽象的基于URL的视图
        AbstractUrlBasedView view = buildView(viewName);
        // 将创建的视图放入到Web应用程序环境中，并且进行初始化
        View result = (View)
getApplicationContext().getAutowireCapableBeanFactory().initializeBean(view, viewName);
        // 如果视图能处理当前的地域，返回当前视图
        return (view.checkResource(locale) ? result : null);
    }

    protected AbstractUrlBasedView buildView(String viewName) throws Exception {
        // 根据配置的视图类名实例化视图对象
        AbstractUrlBasedView view = (AbstractUrlBasedView)
BeanUtils.instantiateClass(getViewClass());

        // URL = 前缀 + 逻辑视图名 + 后缀
        view.setUrl(getPrefix() + viewName + getSuffix());

        // 设置内容类型
        String contentType = getContentType();
        if (contentType != null) {
            view.setContentType(contentType);
        }
```

```
        // 设置请求环境属性的关键字值
        view.setRequestContextAttribute(getRequestContextAttribute());

        // 这是静态属性
        view.setAttributesMap(getAttributesMap());
        return view;
    }

}
```

我们看到，抽象的基于 URL 的视图解析器返回抽象的基于 URL 的视图。现在我们分析一下抽象的基于 URL 的视图的类的实现。实现视图的第一层的类是抽象视图，它合并了派遣器 Servlet 传进来的模型数据和视图解析器传进来的静态模型数据。将这些数据准备给子类实现视图显示逻辑。如下代码所示，

```
public abstract class AbstractView extends WebApplicationObjectSupport implements View,
BeanNameAware {
    // 缺省的内容类型为HTML
    public static final String DEFAULT_CONTENT_TYPE = "text/html;charset=ISO-8859-1";

    // 每次输出的数据长度
    private static final int OUTPUT_BYTE_ARRAY_INITIAL_SIZE = 4096;

    // 视图作为Bean的名字
    private String beanName;

    // 视图的能处理的内容类型
    private String contentType = DEFAULT_CONTENT_TYPE;

    // 保存请求环境属性的关键字
    private String requestContextAttribute;

    // 静态属性集合
    private final Map<String, Object> staticAttributes = new HashMap<String, Object>();

    // 使用CSV格式的字符串设置静态属性
    public void setAttributesCSV(String propString) throws IllegalArgumentException {
        if (propString != null) {
            StringTokenizer st = new StringTokenizer(propString, ",");
            while (st.hasMoreTokens()) {
                String tok = st.nextToken();
                int eqIdx = tok.indexOf("=");
                if (eqIdx == -1) {
```

```
                    throw new IllegalArgumentException("Expected = in attributes CSV
string '" + propString + "'");
                }
                if (eqIdx >= tok.length() - 2) {
                    throw new IllegalArgumentException(
                            "At least 2 characters ([]) required in attributes CSV string
'" + propString + "'");
                }
                String name = tok.substring(0, eqIdx);
                String value = tok.substring(eqIdx + 1);

                // Delete first and last characters of value: { and }
                value = value.substring(1);
                value = value.substring(0, value.length() - 1);

                addStaticAttribute(name, value);
            }
        }
    }

    public void render(Map<String, ?> model, HttpServletRequest request,
HttpServletResponse response) throws Exception {
        if (logger.isTraceEnabled()) {
            logger.trace("Rendering view with name '" + this.beanName + "' with model "
+ model +
                " and static attributes " + this.staticAttributes);
        }

        // Consolidate static and dynamic model attributes.
        Map<String, Object> mergedModel =
                new HashMap<String, Object>(this.staticAttributes.size() + (model !=
null ? model.size() : 0));
        mergedModel.putAll(this.staticAttributes);
        if (model != null) {
            mergedModel.putAll(model);
        }

        // Expose RequestContext?
        if (this.requestContextAttribute != null) {
            mergedModel.put(this.requestContextAttribute,
createRequestContext(request, response, mergedModel));
        }

        // 准备响应
```

```
        prepareResponse(request, response);

        // 生成响应
        renderMergedOutputModel(mergedModel, request, response);
    }

    // 如果设置了请求环境属性的关键字值，导出请求环境
    protected RequestContext createRequestContext(
            HttpServletRequest request, HttpServletResponse response, Map<String,
Object> model) {

        return new RequestContext(request, response, getServletContext(), model);
    }

    // 解决IE的一个bug
    protected void prepareResponse(HttpServletRequest request, HttpServletResponse
response) {
        if (generatesDownloadContent()) {
            response.setHeader("Pragma", "private");
            response.setHeader("Cache-Control", "private, must-revalidate");
        }
    }

    // 默认不产生下载内容
    protected boolean generatesDownloadContent() {
        return false;
    }

    // 子类需要实现显示视图的逻辑
    protected abstract void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request, HttpServletResponse
response) throws Exception;

    // 提供功能导出模型数据到请求属性中
    protected void exposeModelAsRequestAttributes(Map<String, Object> model,
HttpServletRequest request) throws Exception {
        for (Map.Entry<String, Object> entry : model.entrySet()) {
            String modelName = entry.getKey();
            Object modelValue = entry.getValue();
            if (modelValue != null) {
                request.setAttribute(modelName, modelValue);
                if (logger.isDebugEnabled()) {
                    logger.debug("Added model object '" + modelName + "' of type [" +
modelValue.getClass().getName() +
```

```
                                "] to request in view with name '" + getBeanName() + "'");
                    }
                }
                else {
                    request.removeAttribute(modelName);
                    if (logger.isDebugEnabled()) {
                        logger.debug("Removed model object '" + modelName +
                                "' from request in view with name '" + getBeanName() + "'");
                    }
                }
            }
        }

    // 创建临时的内存输出流
    protected ByteArrayOutputStream createTemporaryOutputStream() {
        return new ByteArrayOutputStream(OUTPUT_BYTE_ARRAY_INITIAL_SIZE);
    }

    // 将一个内存输出流写入响应中
    protected void writeToResponse(HttpServletResponse response, ByteArrayOutputStream
baos) throws IOException {
        // Write content type and also length (determined via byte array).
        response.setContentType(getContentType());
        response.setContentLength(baos.size());

        // Flush byte array to servlet output stream.
        ServletOutputStream out = response.getOutputStream();
        baos.writeTo(out);
        out.flush();
    }

}
```

视图实现的体系结构中的下一个实现类是抽象的基于 URL 的视图。它没有实现如何显示 URL，而是实现了保存一个 URL 的值，子类需要根据 URL 的值来完成视图的显示。如下代码所示，

```
public abstract class AbstractUrlBasedView extends AbstractView implements
InitializingBean {
    // 存储一个将要显示的URL
    private String url;


    protected AbstractUrlBasedView() {
```

```
    }

    protected AbstractUrlBasedView(String url) {
        this.url = url;
    }

    public void afterPropertiesSet() throws Exception {
        // 默认URL是必须的
        if (isUrlRequired() && getUrl() == null) {
            throw new IllegalArgumentException("Property 'url' is required");
        }
    }

    protected boolean isUrlRequired() {
        return true;
    }

    // 占位符方法，子类需要判断是否此视图支持某一个地域信息
    public boolean checkResource(Locale locale) throws Exception {
        return true;
    }

}
```

抽象的基于 URL 的视图解析器和抽象的基于 URL 的视图是体系结构中的基础抽象类，所有其他的实体类都是通过实现他们来实现一个具体视图解析的逻辑。

4.3.1.1 内部资源视图解析器和内部资源视图

内部资源视图解析器和内部资源视图是最常用的视图解析器和视图的实现。他们通过转发请求给 JSP, Servlet 或者具有 JSTL 标记的 JSP 组件进行处理。

内部资源视图解析器通过改写抽象的基于 URL 的视图解析器的构建视图方法，构建专用的内部资源视图或者 JSTL 视图。如下代码所示，

```
public class InternalResourceViewResolver extends UrlBasedViewResolver {

    // 判断是否JSTL相关类存在
    private static final boolean jstlPresent = ClassUtils.isPresent(
            "javax.servlet.jsp.jstl.core.Config",
InternalResourceViewResolver.class.getClassLoader());

    // 是否总是使用包含派遣HTTP请求
    private Boolean alwaysInclude;
```

```
    // 是否导出环境的Bean作为属性
    private Boolean exposeContextBeansAsAttributes;

    // 需要导出Web应用程序环境中的Bean的名字
    private String[] exposedContextBeanNames;


    public InternalResourceViewResolver() {
        // 如果JSTL相关类存在，则使用JSTL视图，否则使用内部资源视图
        Class viewClass = requiredViewClass();
        if (viewClass.equals(InternalResourceView.class) && jstlPresent) {
            viewClass = JstlView.class;
        }
        setViewClass(viewClass);
    }

    @Override
    protected Class requiredViewClass() {
        // 默认使用内部资源视图
        return InternalResourceView.class;
    }

    @Override
    protected AbstractUrlBasedView buildView(String viewName) throws Exception {
        // 使用超类创建内部资源视图或者JSTL视图
        InternalResourceView view = (InternalResourceView) super.buildView(viewName);

        // 设置是否总是使用包含请求操作
        if (this.alwaysInclude != null) {
            view.setAlwaysInclude(this.alwaysInclude);
        }

        // 设置是否导出环境的Bean作为属性
        if (this.exposeContextBeansAsAttributes != null) {

    view.setExposeContextBeansAsAttributes(this.exposeContextBeansAsAttributes);
        }

        // 需要导出Web应用程序环境中的Bean的名字
        if (this.exposedContextBeanNames != null) {
            view.setExposedContextBeanNames(this.exposedContextBeanNames);
        }
```

```
        // 防止派遣死循环
        view.setPreventDispatchLoop(true);
        return view;
    }

}
```

从上面视图解析器的实现可以看出，它通过检查是否存在 JSTL 相关类，来创建内部资源视图还是 JSTL 视图。事实上，JSTL 视图是内部资源视图的子类，它不但支持 JSP 和 Servlet 服务器断组件，还支持包含 JSTL 的 JSP 展示层的组件。如下代码所示，

```
public class InternalResourceView extends AbstractUrlBasedView {
    // 是否总是使用包含派遣请求
    private boolean alwaysInclude = false;

    // 是否导出转发属性
    private volatile Boolean exposeForwardAttributes;

    // 是否导出应用程序环境中的Bean作为属性
    private boolean exposeContextBeansAsAttributes = false;

    // 导出那些Bean作为属性
    private Set<String> exposedContextBeanNames;

    // 是否阻止派遣死循环
    private boolean preventDispatchLoop = false;

    @Override
    protected void initServletContext(ServletContext sc) {
        // 如果早于Servlet 2.5，需要手动导出转发属性
        if (this.exposeForwardAttributes == null && sc.getMajorVersion() == 2 &&
sc.getMinorVersion() < 5) {
            this.exposeForwardAttributes = Boolean.TRUE;
        }
    }

    @Override
    protected void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request, HttpServletResponse
response) throws Exception {

        // Determine which request handle to expose to the RequestDispatcher.
        HttpServletRequest requestToExpose = getRequestToExpose(request);
```

```
        // Expose the model object as request attributes.
        exposeModelAsRequestAttributes(model, requestToExpose);

        // Expose helpers as request attributes, if any.
        exposeHelpers(requestToExpose);

        // Determine the path for the request dispatcher.
        String dispatcherPath = prepareForRendering(requestToExpose, response);

        // Obtain a RequestDispatcher for the target resource (typically a JSP).
        RequestDispatcher rd = getRequestDispatcher(requestToExpose, dispatcherPath);
        if (rd == null) {
            throw new ServletException("Could not get RequestDispatcher for [" + getUrl()
+
                    "]: Check that the corresponding file exists within your web
application archive!");
        }

        // If already included or response already committed, perform include, else forward.
        if (useInclude(requestToExpose, response)) {
            response.setContentType(getContentType());
            if (logger.isDebugEnabled()) {
                logger.debug("Including resource [" + getUrl() + "] in
InternalResourceView '" + getBeanName() + "'");
            }
            rd.include(requestToExpose, response);
        }

        else {
            // Note: The forwarded resource is supposed to determine the content type
itself.
            exposeForwardRequestAttributes(requestToExpose);
            if (logger.isDebugEnabled()) {
                logger.debug("Forwarding to resource [" + getUrl() + "] in
InternalResourceView '" + getBeanName() + "'");
            }
            rd.forward(requestToExpose, response);
        }
    }

    protected HttpServletRequest getRequestToExpose(HttpServletRequest originalRequest)
{
        // 如果导出Bean作为属性，则创建一个代理HTTP Servlet请求，这个代理请求包含导出的Bean作为属
性
```

```
        if (this.exposeContextBeansAsAttributes || this.exposedContextBeanNames != null)
{
            return new ContextExposingHttpServletRequest(
                    originalRequest, getWebApplicationContext(),
this.exposedContextBeanNames);
        }
        // 否则导出原来的HTTP Servlet请求
        return originalRequest;
    }

    protected String prepareForRendering(HttpServletRequest request, HttpServletResponse
response)
            throws Exception {

        String path = getUrl();
        // 如果设置了防止派遣死循环，并且检测到派遣到最初的URL，则阻止死循环，抛出异常，终止处理
        if (this.preventDispatchLoop) {
            String uri = request.getRequestURI();
            if (path.startsWith("/") ? uri.equals(path) :
uri.equals(StringUtils.applyRelativePath(uri, path))) {
                throw new ServletException("Circular view path [" + path + "]: would
dispatch back " +
                        "to the current handler URL [" + uri + "] again. Check your
ViewResolver setup! " +
                        "(Hint: This may be the result of an unspecified view, due to
default view name generation.)");
            }
        }
        return path;
    }

    protected RequestDispatcher getRequestDispatcher(HttpServletRequest request, String
path) {
        // 使用容器的请求派遣器
        return request.getRequestDispatcher(path);
    }

    protected boolean useInclude(HttpServletRequest request, HttpServletResponse response)
{
        // 如果设置了总是使用包含，或者请求是包含请求，或者已经发送了响应代码
        return (this.alwaysInclude || WebUtils.isIncludeRequest(request) ||
response.isCommitted());
    }
```

```
    protected void exposeForwardRequestAttributes(HttpServletRequest request) {
        if (this.exposeForwardAttributes != null && this.exposeForwardAttributes) {
            try {
                // 导出转发请求属性FORWARD_REQUEST_URI_ATTRIBUTE,
FORWARD_CONTEXT_PATH_ATTRIBUTE, FORWARD_SERVLET_PATH_ATTRIBUTE,
FORWARD_PATH_INFO_ATTRIBUTE, FORWARD_QUERY_STRING_ATTRIBUTE
                WebUtils.exposeForwardRequestAttributes(request);
            }
            catch (Exception ex) {
                // Servlet container rejected to set internal attributes, e.g. on TriFork.
                this.exposeForwardAttributes = Boolean.FALSE;
            }
        }
    }

}
```

子类 JSTL 视图能够重用了所有的内部资源视图的逻辑，而且增加了导出本地化环境的实现。如下代码所示，

```
public class JstlView extends InternalResourceView {
    // 设置的消息源
    private MessageSource messageSource;

    @Override
    protected void exposeHelpers(HttpServletRequest request) throws Exception {
        // 如果设置了消息源，则导出设置的消息源
        if (this.messageSource != null) {
            JstlUtils.exposeLocalizationContext(request, this.messageSource);
        }
        // 否则导出应用程序环境作为消息源，应用程序环境也实现了消息源接口
        else {
            JstlUtils.exposeLocalizationContext(new RequestContext(request,
getServletContext()));
        }
    }

}
```

4.3.1.2 瓦块视图解析器和瓦块视图

瓦块视图解析器和瓦块视图把 HTTP 请求派遣给瓦块容器进行处理，瓦块容器使用不同的瓦块定义来显示一个完整的瓦块页面。

瓦块视图解析器继承自抽象的基于 URL 的视图解析器。简单的通过返回瓦块视图类来实现的。如下代码所示，

```
public class TilesViewResolver extends UrlBasedViewResolver {

    public TilesViewResolver() {
        setViewClass(requiredViewClass());
    }

    @Override
    protected Class requiredViewClass() {
        // 简单的返回瓦块视图的定义
        return TilesView.class;
    }

}
```

瓦块视图则把当前的 URL 交给瓦块容器进行响应。如下代码注释，

```
public class TilesView extends AbstractUrlBasedView {

    @Override
    public boolean checkResource(final Locale locale) throws Exception {
        // 从应用程序对象中取得瓦块容器
        TilesContainer container = ServletUtil.getContainer(getServletContext());
        // 瓦块容器应该是基本瓦块容器类型，如果不是，则做乐观处理
        if (!(container instanceof BasicTilesContainer)) {
            // Cannot check properly - let's assume it's there.
            return true;
        }

        // 创建瓦块请求环境
        BasicTilesContainer basicContainer = (BasicTilesContainer) container;
        TilesApplicationContext appContext = new
ServletTilesApplicationContext(getServletContext());
        TilesRequestContext requestContext = new ServletTilesRequestContext(appContext,
null, null) {
            @Override
            public Locale getRequestLocale() {
                return locale;
            }
        };

        // 检查是否存在瓦块页面定义可以处理当前的URL
```

```
        return (basicContainer.getDefinitionsFactory().getDefinition(getUrl(),
requestContext) != null);
    }

    @Override
    protected void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request, HttpServletResponse
response) throws Exception {

        ServletContext servletContext = getServletContext();
        // 从应用程序对象中取得瓦块容器
        TilesContainer container = ServletUtil.getContainer(servletContext);
        // 如果瓦块容器不存在，则不能处理当前的请求，终止处理
        if (container == null) {
            throw new ServletException("Tiles container is not initialized. " +
                    "Have you added a TilesConfigurer to your web application context?");
        }

        // 导出模型映射数据作为请求属性以供瓦块容器使用
        exposeModelAsRequestAttributes(model, request);

        // 导出本地化环境
        JstlUtils.exposeLocalizationContext(new RequestContext(request,
servletContext));

        if (!response.isCommitted()) {
            // Tiles is going to use a forward, but some web containers (e.g. OC4J 10.1.3)
            // do not properly expose the Servlet 2.4 forward request attributes...
However,
            // must not do this on Servlet 2.5 or above, mainly for GlassFish compatibility.
            ServletContext sc = getServletContext();
            if (sc.getMajorVersion() == 2 && sc.getMinorVersion() < 5) {
                WebUtils.exposeForwardRequestAttributes(request);
            }
        }

        // 把请求交给瓦块容器进行处理
        container.render(getUrl(), request, response);
    }

}
```

4.3.1.3模板视图解析器和模板视图

模板视图解析器和模板视图把 HTTP 请求传递给模板技术的容器进行处理，它支持 Velocity 和 FreeMaker。

抽象的模板视图解析器继承自基于 URL 的视图解析器。并且可以对创建的抽象模板视图进行设置，如下代码注释，

```
public class AbstractTemplateViewResolver extends UrlBasedViewResolver {

    private boolean exposeRequestAttributes = false;

    private boolean allowRequestOverride = false;

    private boolean exposeSessionAttributes = false;

    private boolean allowSessionOverride = false;

    private boolean exposeSpringMacroHelpers = true;

    @Override
    protected Class requiredViewClass() {
        // 支持抽象模板视图类
        return AbstractTemplateView.class;
    }

    @Override
    protected AbstractUrlBasedView buildView(String viewName) throws Exception {
        // 对抽象模板视图进行客户化的设置
        AbstractTemplateView view = (AbstractTemplateView) super.buildView(viewName);
        view.setExposeRequestAttributes(this.exposeRequestAttributes);
        view.setAllowRequestOverride(this.allowRequestOverride);
        view.setExposeSessionAttributes(this.exposeSessionAttributes);
        view.setAllowSessionOverride(this.allowSessionOverride);
        view.setExposeSpringMacroHelpers(this.exposeSpringMacroHelpers);
        return view;
    }

}
```

抽象的模板视图根据客户化的设置对请求进行一系列的预处理，然后，把 HTTP 请求传递给子类，子类根据具体的模板实现技术交给模板容器进行处理。

```
public abstract class AbstractTemplateView extends AbstractUrlBasedView {
```

```
    public static final String SPRING_MACRO_REQUEST_CONTEXT_ATTRIBUTE =
"springMacroRequestContext";


    private boolean exposeRequestAttributes = false;

    private boolean allowRequestOverride = false;

    private boolean exposeSessionAttributes = false;

    private boolean allowSessionOverride = false;

    private boolean exposeSpringMacroHelpers = true;

    @Override
    protected final void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request, HttpServletResponse
response) throws Exception {

        if (this.exposeRequestAttributes) {
            // 导出请求属性到Spring模型中
            for (Enumeration en = request.getAttributeNames(); en.hasMoreElements();) {
                String attribute = (String) en.nextElement();
                if (model.containsKey(attribute) && !this.allowRequestOverride) {
                    throw new ServletException("Cannot expose request attribute '" +
attribute +
                            "' because of an existing model object of the same name");
                }
                Object attributeValue = request.getAttribute(attribute);
                if (logger.isDebugEnabled()) {
                    logger.debug("Exposing request attribute '" + attribute +
                            "' with value [" + attributeValue + "] to model");
                }
                model.put(attribute, attributeValue);
            }
        }

        if (this.exposeSessionAttributes) {
            // 导出Session属性到Spring模型中
            HttpSession session = request.getSession(false);
            if (session != null) {
                for (Enumeration en = session.getAttributeNames();
en.hasMoreElements();) {
```

```
                        String attribute = (String) en.nextElement();
                        if (model.containsKey(attribute) && !this.allowSessionOverride) {
                            throw new ServletException("Cannot expose session attribute '"
+ attribute +
                                    "' because of an existing model object of the same name");
                        }
                        Object attributeValue = session.getAttribute(attribute);
                        if (logger.isDebugEnabled()) {
                            logger.debug("Exposing session attribute '" + attribute +
                                    "' with value [" + attributeValue + "] to model");
                        }
                        model.put(attribute, attributeValue);
                    }
                }
            }

            if (this.exposeSpringMacroHelpers) {
                if (model.containsKey(SPRING_MACRO_REQUEST_CONTEXT_ATTRIBUTE)) {
                    throw new ServletException(
                            "Cannot expose bind macro helper '" +
SPRING_MACRO_REQUEST_CONTEXT_ATTRIBUTE +
                            "' because of an existing model object of the same name");
                }
                // Expose RequestContext instance for Spring macros.
                model.put(SPRING_MACRO_REQUEST_CONTEXT_ATTRIBUTE,
                        new RequestContext(request, response, getServletContext(),
model));
            }

            // 应用响应内容类型
            applyContentType(response);

            // 由子类实现使用模板技术进行处理
            renderMergedTemplateModel(model, request, response);
        }

        protected void applyContentType(HttpServletResponse response){
            if (response.getContentType() == null) {
                response.setContentType(getContentType());
            }
        }

        protected abstract void renderMergedTemplateModel(
                Map<String, Object> model, HttpServletRequest request, HttpServletResponse
```

```
response) throws Exception;

}
```

抽象模板视图解析器和抽象模板视图有两个类型的子类，两个类型的子类各自支持 Velocity 和 FreeMaker。

FreeMakerViewResolver 的实现非常简单，它仅仅返回 FreeMakerView 类。而 FreeMakerView 则将 HTTP 请求传递给 FreeMaker 容器来处理并且响应产生展示层的界面。

VelocityViewResolver 的实现非常简单，它仅仅返回 VelocityView 类。而 VelocityView 则将 HTTP 请求传递给 Velocity 容器来处理并且响应产生展示层的界面。

VelocityLayoutViewResolver 和 VelocityLayoutView 是 VelocityViewResolver 和 VelocityView 的子类，它增加了对 Layout 的支持。

既然这些类的实现需要更多的 FreeMaker 和 Velocity 的知识，我们这里不做详细的代码分析和注释。

4.3.1.4Jasper报表视图解析器和Jaspter报表视图

由于本人时间有限，将在本书第二版完成本节的内容。如您愿意提供帮助请发邮件到 robertleepeak@gmail.com。

4.3.1.5XSLT视图解析器和XSLT视图

XSLT 视图解析器和 XSLT 视图使用 XSLT 技术将 XML DOM 数据转换成为一种支持的其他数据格式，这包括其他的 XML 格式，也包括类似 HTML 的展示层的数据显示格式。

XSLT 视图解析器继承自基于 URL 的视图解析器，并且返回 XSLT 视图。如下代码注释，

```
public class XsltViewResolver extends UrlBasedViewResolver {

    private String sourceKey;

    private URIResolver uriResolver;

    private ErrorListener errorListener;

    private boolean indent = true;

    private Properties outputProperties;
```

```
    private boolean cacheTemplates = true;

    @Override
    protected Class requiredViewClass() {
        // 返回XSLT视图类，父类负责创建此类实例
        return XsltView.class;
    }

    @Override
    protected AbstractUrlBasedView buildView(String viewName) throws Exception {
        // 调用父类创建XSLT视图类的实例
        XsltView view = (XsltView) super.buildView(viewName);

        // 设置数据源的属性关键字
        view.setSourceKey(this.sourceKey);

        // 设置URI解析器
        if (this.uriResolver != null) {
            view.setUriResolver(this.uriResolver);
        }

        // 设置错误监听器
        if (this.errorListener != null) {
            view.setErrorListener(this.errorListener);
        }

        // 设置其他属性
        view.setIndent(this.indent);
        view.setOutputProperties(this.outputProperties);
        view.setCacheTemplates(this.cacheTemplates);
        return view;
    }

}
```

XSLT 视图则使用 XSLT 技术将一种 XML 格式的数据源转换成为另外一种，如下代码注释所示，

```
public class XsltView extends AbstractUrlBasedView {
    // 可以配置一个非缺省的转换工厂类
    private Class transformerFactoryClass;

    private String sourceKey;
```

```
    private URIResolver uriResolver;

    private ErrorListener errorListener = new SimpleTransformErrorListener(logger);

    private boolean indent = true;

    private Properties outputProperties;

    private boolean cacheTemplates = true;

    private TransformerFactory transformerFactory;

    private Templates cachedTemplates;

    @Override
    protected void initApplicationContext() throws BeansException {
        // 创建转换工厂类的实例并且初始化
        this.transformerFactory = newTransformerFactory(this.transformerFactoryClass);
        this.transformerFactory.setErrorListener(this.errorListener);
        if (this.uriResolver != null) {
            this.transformerFactory.setURIResolver(this.uriResolver);
        }
        // 如果缓存模板，则初始化时加载所有模板
        if (this.cacheTemplates) {
            this.cachedTemplates = loadTemplates();
        }
    }

    protected TransformerFactory newTransformerFactory(Class transformerFactoryClass) {
        // 如果配置了非缺省的转换工厂类，则实例化配置的转换工厂类
        if (transformerFactoryClass != null) {
            try {
                return (TransformerFactory) transformerFactoryClass.newInstance();
            }
            catch (Exception ex) {
                throw new TransformerFactoryConfigurationError(ex, "Could not
instantiate TransformerFactory");
            }
        }
        // 否则使用缺省的转换工厂类
        else {
            return TransformerFactory.newInstance();
        }
    }
```

```
    @Override
    protected void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request, HttpServletResponse
response)
            throws Exception {
        // 如果没有缓存XSLT模板，则加载模板
        Templates templates = this.cachedTemplates;
        if (templates == null) {
            templates = loadTemplates();
        }

        // 根据模板穿件转换对象
        Transformer transformer = createTransformer(templates);
        // 配置转换对象
        configureTransformer(model, response, transformer);
        // 配置响应
        configureResponse(model, response, transformer);
        Source source = null;
        try {
            // 找到XSLT的数据源
            source = locateSource(model);
            if (source == null) {
                throw new IllegalArgumentException("Unable to locate Source object in
model: " + model);
            }
            // 进行事实的转换
            transformer.transform(source, createResult(response));
        }
        finally {
            closeSourceIfNecessary(source);
        }
    }

    protected Result createResult(HttpServletResponse response) throws Exception {
        // 使用响应的输出流构造数据结果
        return new StreamResult(response.getOutputStream());
    }

    protected Source locateSource(Map<String, Object> model) throws Exception {
        // 如果配置了数据源的关键字，则使用模型属性中以此关键字标记的属性
        if (this.sourceKey != null) {
            return convertSource(model.get(this.sourceKey));
        }
```

```
        // 否则找到第一个能够作为数据源类型的属性作为数据源
        Object source = CollectionUtils.findValueOfType(model.values(),
getSourceTypes());
        return (source != null ? convertSource(source) : null);
    }

    protected Class[] getSourceTypes() {
        // 这些类型都可以作为输入的数据源
        return new Class[] {Source.class, Document.class, Node.class, Reader.class,
InputStream.class, Resource.class};
    }

    protected Source convertSource(Object source) throws Exception {
        // 需要把不同类型的数据源转换为XSLT的标准数据源类型
        if (source instanceof Source) {
            return (Source) source;
        }
        else if (source instanceof Document) {
            return new DOMSource(((Document) source).getDocumentElement());
        }
        else if (source instanceof Node) {
            return new DOMSource((Node) source);
        }
        else if (source instanceof Reader) {
            return new StreamSource((Reader) source);
        }
        else if (source instanceof InputStream) {
            return new StreamSource((InputStream) source);
        }
        else if (source instanceof Resource) {
            Resource resource = (Resource) source;
            return new StreamSource(resource.getInputStream(),
resource.getURI().toASCIIString());
        }
        else {
            throw new IllegalArgumentException("Value '" + source + "' cannot be converted
to XSLT Source");
        }
    }

    protected void configureTransformer(Map<String, Object> model, HttpServletResponse
response, Transformer transformer) {
        // 把模型参数拷贝到XSLT转换对象中
```

```
        copyModelParameters(model, transformer);
        // 拷贝输出属性
        copyOutputProperties(transformer);
        // 配置是否缩进
        configureIndentation(transformer);
    }

    protected final void configureIndentation(Transformer transformer) {
        // 打开或者关闭缩进
        if (this.indent) {
            TransformerUtils.enableIndenting(transformer);
        }
        else {
            TransformerUtils.disableIndenting(transformer);
        }
    }

    protected final void copyOutputProperties(Transformer transformer) {
        // 拷贝输出属性
        if (this.outputProperties != null) {
            Enumeration en = this.outputProperties.propertyNames();
            while (en.hasMoreElements()) {
                String name = (String) en.nextElement();
                transformer.setOutputProperty(name,
this.outputProperties.getProperty(name));
            }
        }
    }

    protected final void copyModelParameters(Map<String, Object> model, Transformer
transformer) {
        // 把模型参数拷贝到XSLT转换对象中
        for (Map.Entry<String, Object> entry : model.entrySet()) {
            transformer.setParameter(entry.getKey(), entry.getValue());
        }
    }

    protected void configureResponse(Map<String, Object> model, HttpServletResponse
response, Transformer transformer) {
        String contentType = getContentType();
        String mediaType = transformer.getOutputProperty(OutputKeys.MEDIA_TYPE);
        String encoding = transformer.getOutputProperty(OutputKeys.ENCODING);
        if (StringUtils.hasText(mediaType)) {
            contentType = mediaType;
```

```
        }
        if (StringUtils.hasText(encoding)) {
            // Only apply encoding if content type is specified but does not contain charset
clause already.
            if (contentType != null
&& !contentType.toLowerCase().contains(WebUtils.CONTENT_TYPE_CHARSET_PREFIX)) {
                contentType = contentType + WebUtils.CONTENT_TYPE_CHARSET_PREFIX +
encoding;
            }
        }
        // 根据XSLT模板支持的类型设置响应的内容类型
        response.setContentType(contentType);
    }

    private Templates loadTemplates() throws ApplicationContextException {
        // 取得模板的数据源
        Source stylesheetSource = getStylesheetSource();
        try {
            // 根据模板的数据源创建模板对象
            Templates templates =
this.transformerFactory.newTemplates(stylesheetSource);
            if (logger.isDebugEnabled()) {
                logger.debug("Loading templates '" + templates + "'");
            }
            return templates;
        }
        catch (TransformerConfigurationException ex) {
            throw new ApplicationContextException("Can't load stylesheet from '" + getUrl()
+ "'", ex);
        }
        finally {
            closeSourceIfNecessary(stylesheetSource);
        }
    }

    protected Transformer createTransformer(Templates templates) throws
TransformerConfigurationException {
        // 根据模板创建转换器
        Transformer transformer = templates.newTransformer();
        if (this.uriResolver != null) {
            transformer.setURIResolver(this.uriResolver);
        }
        return transformer;
    }
```

```
    protected Source getStylesheetSource() {
        // 根据请求的URL创建数据源
        String url = getUrl();
        if (logger.isDebugEnabled()) {
            logger.debug("Loading XSLT stylesheet from '" + url + "'");
        }
        try {
            Resource resource = getApplicationContext().getResource(url);
            return new StreamSource(resource.getInputStream(),
resource.getURI().toASCIIString());
        }
        catch (IOException ex) {
            throw new ApplicationContextException("Can't load XSLT stylesheet from '" +
url + "'", ex);
        }
    }

    private void closeSourceIfNecessary(Source source) {
        // 关闭一个流类型的数据源
        if (source instanceof StreamSource) {
            StreamSource streamSource = (StreamSource) source;
            if (streamSource.getReader() != null) {
                try {
                    streamSource.getReader().close();
                }
                catch (IOException ex) {
                    // ignore
                }
            }
            if (streamSource.getInputStream() != null) {
                try {
                    streamSource.getInputStream().close();
                }
                catch (IOException ex) {
                    // ignore
                }
            }
        }
    }

}
```

4.3.2 更多的视图解析器

上一节中，我们介绍了基于 URL 的视图解析器和视图的实现。事实上，还存在这一些视图解析器，他们不仅支持某一种类型的视图实现，而是能在多种视图实现中互相转换和选择。这一节中我们将分析四种这样类型的视图解析器的实现。

4.3.2.1Bean名视图解析器

Bean 名视图解析器通过把逻辑视图名作为 Web 应用程序环境中的 Bean 名来解析视图。如下代码所示，

```
public class BeanNameViewResolver extends WebApplicationObjectSupport implements
ViewResolver, Ordered {

    private int order = Integer.MAX_VALUE;  // default: same as non-Ordered


    public void setOrder(int order) {
        this.order = order;
    }

    public int getOrder() {
        return order;
    }


    public View resolveViewName(String viewName, Locale locale) throws BeansException {
        取得Web应用程序环境
        ApplicationContext context = getApplicationContext();

        如果Web应用程序环境中不包含一个Bean，它的名字是逻辑视图名，则返回空的视图
        if (!context.containsBean(viewName)) {
            // Allow for ViewResolver chaining.
            return null;
        }

        如果Web应用程序环境中包含这样的一个Bean，则返回这个Bean作为视图
        return (View) context.getBean(viewName, View.class);
    }

}
```

4.3.2.2内容选择视图解析器

内容选择视图解析器根据 HTTP 请求所指定的媒体类型来选择一个合适的视图解析器来解析视图，但是它自己并不解析视图。一个 HTTP 请求可以通过下列的方式之一来制定媒体类型，

1. 根据 URL 路径的扩展名。
2. 根据制定的参数值。
3. 根据 HTTP 头的接受内容类型。

首先在初始化的时候它加载了所有的其他的视图解析器。如下代码注释，

```
@Override
protected void initServletContext(ServletContext servletContext) {
    if (this.viewResolvers == null) {
        // 在Web应用程序环境中找到所有的视图解析器的实现
        Map<String, ViewResolver> matchingBeans =

    BeanFactoryUtils.beansOfTypeIncludingAncestors(getApplicationContext(),
ViewResolver.class);
        this.viewResolvers = new ArrayList<ViewResolver>(matchingBeans.size());
        for (ViewResolver viewResolver : matchingBeans.values()) {
            // 保存所有的视图解析器，并且排除自己
            if (this != viewResolver) {
                this.viewResolvers.add(viewResolver);
            }
        }
    }

    // 如果不能找到视图解析器，则打印警告日志
    if (this.viewResolvers.isEmpty()) {
        logger.warn("Did not find any ViewResolvers to delegate to; please configure them
using the " +
                "'viewResolvers' property on the ContentNegotiatingViewResolver");
    }

    // 排序视图解析器，ContentNegotiatingViewResolver的优先级是最高的
    OrderComparator.sort(this.viewResolvers);
}
```

当处理一个 HTTP 请求的时候，它首先通过上述的三个规则解析请求所指定的媒体类型，如下代码所示，

```
protected List<MediaType> getMediaTypes(HttpServletRequest request) {
```

```
    // 如果扩展名解析优先，从URL的扩展名中解析媒体类型
    if (this.favorPathExtension) {
        String requestUri = urlPathHelper.getRequestUri(request);
        String filename = WebUtils.extractFullFilenameFromUrlPath(requestUri);
        MediaType mediaType = getMediaTypeFromFilename(filename);
        if (mediaType != null) {
            if (logger.isDebugEnabled()) {
                logger.debug("Requested media type is '" + mediaType + "' (based on
filename '" + filename + "')");
            }
            List<MediaType> mediaTypes = new ArrayList<MediaType>();
            mediaTypes.add(mediaType);
            return mediaTypes;
        }
    }

    // 如果参数解析优先，从参数中解析媒体类型
    if (this.favorParameter) {
        if (request.getParameter(this.parameterName) != null) {
            String parameterValue = request.getParameter(this.parameterName);
            MediaType mediaType = getMediaTypeFromParameter(parameterValue);
            if (mediaType != null) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Requested media type is '" + mediaType + "' (based on
parameter '" +
                            this.parameterName + "'='" + parameterValue + "')");
                }
                List<MediaType> mediaTypes = new ArrayList<MediaType>();
                mediaTypes.add(mediaType);
                return mediaTypes;
            }
        }
    }

    // 如果不忽略HTTP头中支持的接受头信息，则从HTTP头中解析媒体类型
    if (!this.ignoreAcceptHeader) {
        String acceptHeader = request.getHeader(ACCEPT_HEADER);
        if (StringUtils.hasText(acceptHeader)) {
            List<MediaType> mediaTypes = MediaType.parseMediaTypes(acceptHeader);
            if (logger.isDebugEnabled()) {
                logger.debug("Requested media types are " + mediaTypes + " (based on Accept
header)");
            }
            return mediaTypes;
```

```
        }
    }

    // 如果解析不成功，则使用缺省内容类型
    if (this.defaultContentType != null) {
        return Collections.singletonList(this.defaultContentType);
    }
    else {
        return Collections.emptyList();
    }
}
```

在得知一个请求所能接受的媒体类型后，它将要选择一个最佳的能处理当前媒体类型的视图解析器来解析具体的视图，如下代码注释，

```
public View resolveViewName(String viewName, Locale locale) throws Exception {
    // 取得Servlet请求属性
    RequestAttributes attrs = RequestContextHolder.getRequestAttributes();
    Assert.isInstanceOf(ServletRequestAttributes.class, attrs);
    ServletRequestAttributes servletAttrs = (ServletRequestAttributes) attrs;

    // 从Servlet请求属性中得到HTTP请求对象
    List<MediaType> requestedMediaTypes = getMediaTypes(servletAttrs.getRequest());
    // 如果支持的媒体类型多余一个，则进行排序
    if (requestedMediaTypes.size() > 1) {
        // avoid sorting attempt for empty list and singleton list
        Collections.sort(requestedMediaTypes);
    }

    // 如果某个视图解析器能够解析当前请求，解析的视图则成为候选视图
    SortedMap<MediaType, View> views = new TreeMap<MediaType, View>();
    List<View> candidateViews = new ArrayList<View>();
    for (ViewResolver viewResolver : this.viewResolvers) {
        View view = viewResolver.resolveViewName(viewName, locale);
        if (view != null) {
            candidateViews.add(view);
        }
    }

    // 缺省视图也是候选视图
    if (!CollectionUtils.isEmpty(this.defaultViews)) {
        candidateViews.addAll(this.defaultViews);
    }
```

```
    // 遍历所有候选视图
    for (View candidateView : candidateViews) {
        // 取得候选试图支持的内容类型
        String contentType = candidateView.getContentType();
        if (StringUtils.hasText(contentType)) {
            // 转换成为媒体类型对象
            MediaType viewMediaType = MediaType.parseMediaType(contentType);
            // 遍历HTTP请求支持的媒体类型
            for (MediaType requestedMediaType : requestedMediaTypes) {
                // 如果HTTP请求支持的媒体类型包含候选视图的媒体类型，则使用这个视图
                if (requestedMediaType.includes(viewMediaType)) {
                    if (!views.containsKey(requestedMediaType)) {
                        views.put(requestedMediaType, candidateView);
                        break;
                    }
                }
            }
        }
    }

    // 如果解析到一个或者多个视图,则使用第一个
    if (!views.isEmpty()) {
        MediaType mediaType = views.firstKey();
        View view = views.get(mediaType);
        if (logger.isDebugEnabled()) {
            logger.debug("Returning [" + view + "] based on requested media type '" +
mediaType + "'");
        }
        return view;
    }
    else {
        return null;
    }
}
```

4.3.2.3资源绑定视图解析器

资源绑定视图解析器从资源绑定中加载 Bean 定义，然后通过视图逻辑名来解析一个定义的 Bean 作为视图名，如下代码注释，

```
@Override
protected View loadView(String viewName, Locale locale) throws Exception {
    从资源绑定中加载Bean工厂
    BeanFactory factory = initFactory(locale);
```

```
    try {
        在Bean工厂中解析Bean
        return factory.getBean(viewName, View.class);
    }
    catch (NoSuchBeanDefinitionException ex) {
        // to allow for ViewResolver chaining
        return null;
    }
}

protected synchronized BeanFactory initFactory(Locale locale) throws BeansException {
    // Try to find cached factory for Locale:
    // Have we already encountered that Locale before?
    if (isCache()) {
        BeanFactory cachedFactory = this.localeCache.get(locale);
        if (cachedFactory != null) {
            return cachedFactory;
        }
    }

    // Build list of ResourceBundle references for Locale.
    List<ResourceBundle> bundles = new LinkedList<ResourceBundle>();
    for (String basename : this.basenames) {
        ResourceBundle bundle = getBundle(basename, locale);
        bundles.add(bundle);
    }

    // Try to find cached factory for ResourceBundle list:
    // even if Locale was different, same bundles might have been found.
    if (isCache()) {
        BeanFactory cachedFactory = this.bundleCache.get(bundles);
        if (cachedFactory != null) {
            this.localeCache.put(locale, cachedFactory);
            return cachedFactory;
        }
    }

    // Create child ApplicationContext for views.
    GenericWebApplicationContext factory = new GenericWebApplicationContext();
    factory.setParent(getApplicationContext());
    factory.setServletContext(getServletContext());

    // Load bean definitions from resource bundle.
    PropertiesBeanDefinitionReader reader = new PropertiesBeanDefinitionReader(factory);
```

```
    reader.setDefaultParentBean(this.defaultParentView);
    for (ResourceBundle bundle : bundles) {
        reader.registerBeanDefinitions(bundle);
    }

    factory.refresh();

    // Cache factory for both Locale and ResourceBundle list.
    if (isCache()) {
        this.localeCache.put(locale, factory);
        this.bundleCache.put(bundles, factory);
    }

    return factory;
}
```

4.3.2.4XML视图解析器

XML 视图解析器从一个 XML 资源文件中加载 Bean 定义，然后通过视图逻辑名来解析一个定义的 Bean 作为视图名，如下代码注释，

```
protected synchronized BeanFactory initFactory() throws BeansException {
    检查是否已经缓存
    if (this.cachedFactory != null) {
        return this.cachedFactory;
    }

    从配置的位置或者缺省的位置加载XML资源
    Resource actualLocation = this.location;
    if (actualLocation == null) {
        actualLocation = getApplicationContext().getResource(DEFAULT_LOCATION);
    }

    // Create child ApplicationContext for views.
    GenericWebApplicationContext factory = new GenericWebApplicationContext();
    factory.setParent(getApplicationContext());
    factory.setServletContext(getServletContext());

    // Load XML resource with context-aware entity resolver.
    XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
    reader.setEntityResolver(new ResourceEntityResolver(getApplicationContext()));
    reader.loadBeanDefinitions(actualLocation);

    factory.refresh();
```

```
    if (isCache()) {
        this.cachedFactory = factory;
    }
    return factory;
}
```

4.3.3 更多的视图

由于本人时间有限，将在本书第二版完成本节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

4.4 其他的Spring Web MVC组件

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

本章节将包含下面的内容。

- Multipart Resolver
- Local Resolver
- Theme Resolver
- HandlerExceptionResolvers
- RequestToViewNameTranslator

5 Spring Webflow

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到 robertleepeak@gmail.com。

本章节将对 Spring 的分离项目 Spring Webflow 进行深入的剖析。

6 Spring安全

Spring 安全是一个功能强大的高可插拔的认证和赋权框架。它是 Spring 应用程序的领域内的标准的安全策略。

Spring 安全是最成熟的并且被广泛应用的 Spring 项目之一。自从 2003 年创建以来，它是一个独立于 Spring 框架的一个单独项目，这个项目同样是通过阿帕奇 2.0 协议发布，所以，它可以应用到任何项目，这包括开源项目，免费项目和商业项目。

这一章，我们将深入的彻底剖析 Spring 安全框架的流程和架构。

6.1 横向剖析

Spring 安全框架主要应用在需要认证与赋权的 Web 应用程序上。它使用 Servlet 标准中的过滤器拦截所有需要安全认证与赋权的 HTTP 请求，并且根据 HTTP 请求的 URL，参数以及其他信息自动识别登录的流程，这个包括首次登录流程，登录失败流程，登录成功流程，退出登录流程，识别不同的流程，它会作出不同的处理操作。

这一节，我们将针对不同的流程进行深入的分析，了解每个流程是如何被识别以及如何被处理的。

既然 Spring 安全框架是通过 Servlet 标准中的过滤器来实现的，在开始流程分析之前，我们首先介绍过滤器的功能和实现。

过滤器技术是 Servlet 2.3 新增加的功能， 它不同于 Servlet,它的设计不是用来产生响应, 但是，它能够在一个请求到达 Servlet 之前预处理请求,也可以在离开 Servlet 时处理响应。

一个过滤器包括如下功能:

1. 在 Servlet 被调用之前截获;。
2. 在 Servlet 被调用之前检查 Servlet 请求;
3. 根据需要修改请求头和请求数据;
4. 根据需要修改响应头和响应数据;
5. 在 Servlet 被调用之后截获.

你能够配置一个过滤器到一个或多个 Servlet，单个 Servlet 或 Servlet 组能够被多个过滤器使用，过滤器也可以通过 URL 进行配置。

既然过滤器能够在请求到达一个特定的 Servlet 之前处理请求和相应，那么，在 Spring 安全框架中正式使用这一特点来完成认证和赋权的流程的。

Spring 安全框架首先需要配置一个过滤器，这个过滤器应用在所有需要安全保护的请求 URL 上，通常应用到 Web 应用程序的跟环境上。如下 web.xml 文件配置所示，

```
<filter>
    <filter-name>springSecurity</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    <init-param>
        <param-name>targetBeanName</param-name>
        <param-value>springSecurityFilterChain</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>springSecurity</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

从配置代码中我们可以看到，转接过滤器代理在拦截了每个 Web 应用程序的请求后，将会把控制权转接给 Spring 应用程序环境中的一个特殊的 Bean，它的名字是 springSecurityFilterChain。如下代码注释，

```
public class DelegatingFilterProxy extends GenericFilterBean {
    // 是否使用专用的Spring跟环境
    private String contextAttribute;

    // 代理过滤器的Bean名字
    private String targetBeanName;

    // 是否代理生命周期方法到代理过滤器Bean，这包括初始化过滤器和卸载过滤器事先
    private boolean targetFilterLifecycle = false;

    // 代理目标过滤器
    private Filter delegate;

    // 同步对象
    private final Object delegateMonitor = new Object();

    @Override
    protected void initFilterBean() throws ServletException {
        // If no target bean name specified, use filter name.
        // 这个Bean名字通常是springSecurityFilterChain，它是Spring自动生成的一个Bean
        if (this.targetBeanName == null) {
            this.targetBeanName = getFilterName();
        }
```

```
        // Fetch Spring root application context and initialize the delegate early, if
possible. If the root application context will be started after this filter proxy, we'll
have to resort to lazy initialization.
        // 如果这个过滤器定义在web.xml中，它将在过滤器的初始化的时候调用，如果这个过滤器定义在Web应
用程序环境做，并且作为一个Bean，它是在Bean初始化中调用，因此很可能被重复调用，所以这里需要同步
        synchronized (this.delegateMonitor) {
            WebApplicationContext wac = findWebApplicationContext();

            // 如果过滤器初始化时或者过滤器Bean初始化是，Web应用程序环境还没有初始化，则使用懒惰加
载
            if (wac != null) {
                this.delegate = initDelegate(wac);
            }
        }
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain
filterChain)
            throws ServletException, IOException {

        // Lazily initialize the delegate if necessary.
        // 如果代理不存在，则懒惰加载
        Filter delegateToUse = null;
        synchronized (this.delegateMonitor) {
            if (this.delegate == null) {
                WebApplicationContext wac = findWebApplicationContext();
                if (wac == null) {
                    throw new IllegalStateException("No WebApplicationContext found: no
ContextLoaderListener registered?");
                }
                this.delegate = initDelegate(wac);
            }
            delegateToUse = this.delegate;
        }

        // Let the delegate perform the actual doFilter operation.
        invokeDelegate(delegateToUse, request, response, filterChain);
    }

    @Override
    public void destroy() {
        // 卸载过滤器事件处理
        Filter delegateToUse = null;
```

```
        synchronized (this.delegateMonitor) {
            delegateToUse = this.delegate;
        }
        if (delegateToUse != null) {
            destroyDelegate(delegateToUse);
        }
    }

    protected WebApplicationContext findWebApplicationContext() {
        // 找到共享跟环境或者专用跟环境，如果是专用跟环境，需要有客户化监听器或者Servlet初始化专用跟环境
        String attrName = getContextAttribute();
        if (attrName != null) {
            return WebApplicationContextUtils.getWebApplicationContext(getServletContext(), attrName);
        }
        else {
            return WebApplicationContextUtils.getWebApplicationContext(getServletContext());
        }
    }

    protected Filter initDelegate(WebApplicationContext wac) throws ServletException {
        Filter delegate = wac.getBean(getTargetBeanName(), Filter.class);

        // 如果配置代理过滤器生命周期方法，则代理过滤器初始化方法
        if (isTargetFilterLifecycle()) {
            delegate.init(getFilterConfig());
        }
        return delegate;
    }

    protected void invokeDelegate(
            Filter delegate, ServletRequest request, ServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        delegate.doFilter(request, response, filterChain);
    }

    protected void destroyDelegate(Filter delegate) {
        // 如果配置代理过滤器生命周期方法，则代理过滤器卸载方法
        if (isTargetFilterLifecycle()) {
            delegate.destroy();
```

```
        }
    }
}
```

由此可以看到，转接过滤器代理的最重要的功能就是把过滤器拦截到的请求转交给代理目标的过滤器的实现，目标过滤器是配置在 Spring 根共享环境的一个实现过滤器接口的 Bean。

另外，转接过滤器代理还根据配置进行代理过滤器的初始化和卸载事件。

转接过滤器代理将截获的 HTTP 请求代理给目标过滤器 Bean 的实现，这个实现是过滤器链代理类。过滤器链代理类首先调用 Web 应用程序环境中配置的 Spring 安全相关的过滤器，然后，再调用过滤器链中的其他过滤器。

在 Web 应用程序环境中配置了多个 Spring 安全相关的过滤器，这些过滤器用来处理登录，登录失败，登录成功以及退出登录等事件。如下 Web 应用程序环境配置所示，

```
<http auto-config="true">
    <!-- Authentication policy -->
    <form-login login-page="/users/login"
login-processing-url="/users/login/authenticate" default-target-url="/main"
authentication-failure-url="/users/login?login_error=1"/>
    <logout logout-url="/users/logout" logout-success-url="/users/logoutSuccess"/>

    <intercept-url pattern="/static/**" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
    <intercept-url pattern="/users/login" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
    <intercept-url pattern="/users/logoutSuccess"
access="IS_AUTHENTICATED_ANONYMOUSLY"/>
   <intercept-url pattern="/**" access="ROLE_USER" />
</http>

<authentication-manager>
   <authentication-provider>
     <user-service>
       <user name="jimi" password="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
       <user name="bob" password="bobspassword" authorities="ROLE_USER" />
     </user-service>
   </authentication-provider>
</authentication-manager>
```

我们看到上面的 Web 应用程序环境的配置，包含了登录页面 URL，登录失败页面 URL，登录成功页面 URL 等信息，它也包含了用来处理登录事件的 URL 和用于推出登录事件的 URL，它同样包含了哪些 URL 需要什么样的角色才能访问和是否匿名访问等信息。

Spring 安全框架在 Web 应用程序环境初始化的过程中根据上述信息生成相应的安全过滤器，

这些安全过滤器按照顺序对截获的 HTTP 请求进行检查和处理，这些检查和处理操作如下，

- 如果当前 HTTP 请求是首次登录请求，则匿名过滤器则将请求转发到登录页面。
- 如果当前 HTTP 请求是登录请求，则用户名密码校验过滤器根据用户细节服务对当前用户进行认证和校验。如果认证和校验通过，则显示登录成功 URL 或者用户初始请求 URL。如果不成功，则显示登录失败页面，通常登录失败页面也是登录页面，因此，用户能够修改登录信息进行重试。
- 如果当前 HTTP 请求是退出登录请求，则推出登录过滤器进行推出登录操作。

我们会在下面的小节里详细的剖析不同的流程是如何实现的，这里只对过滤器链代理类代码进行解析，过滤器链代理类根据各种过滤器的顺序，逐个调用相应的过滤器，来完成安全认证的过程。如下代码注释，

```
public class FilterChainProxy extends GenericFilterBean {
    // 原始的安全过滤器链集合,一个路径Pattern对应多个过滤器映射,在web.xml中配置的一个过滤器映射对应这里的一个入口项
    private Map<String, List<Filter>> uncompiledFilterChainMap;

    // 对路径Pattern处理过后的过滤器链映射
    private Map<Object, List<Filter>> filterChainMap;

    // 蚂蚁路径匹配器
    private UrlMatcher matcher = new AntUrlPathMatcher();

    // 匹配是是否使用包含参数的路径
    private boolean stripQueryStringFromUrls = true;

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

     // 构造一个包含请求，相应和过滤器链的对象
       FilterInvocation fi = new FilterInvocation(request, response, chain);

       // 根据当前请求的URL找到所有应该处理的安全过滤器，这包含匿名过滤器，用户名密码校验过滤器，退出登录过滤器等等
       List<Filter> filters = getFilters(fi.getRequestUrl());

       // 如果没有找到任何关联到当前URL的安全过滤器
       if (filters == null || filters.size() == 0) {
          if (logger.isDebugEnabled()) {
             logger.debug(fi.getRequestUrl() +
                   filters == null ? " has no matching filters" : " has an empty filter list");
```

```
            }

            chain.doFilter(request, response);

            return;
        }
        // 虚拟过滤器链先对附加的过滤器进行调用，然后再调用原来的过滤器链
        VirtualFilterChain virtualFilterChain = new VirtualFilterChain(fi, filters);
        virtualFilterChain.doFilter(fi.getRequest(), fi.getResponse());
    }

    public List<Filter> getFilters(String url)  {
     // 如果配置移除查询串，则从URl中移除？之后的所有参数信息
        if (stripQueryStringFromUrls) {
            // String query string - see SEC-953
            int firstQuestionMarkIndex = url.indexOf("?");

            if (firstQuestionMarkIndex != -1) {
                url = url.substring(0, firstQuestionMarkIndex);
            }
        }

        for (Map.Entry<Object, List<Filter>> entry : filterChainMap.entrySet()) {
            Object path = entry.getKey();
            // 根据情况决定是否使用小写路径匹配
            if (matcher.requiresLowerCaseUrl()) {
                url = url.toLowerCase();

                if (logger.isDebugEnabled()) {
                    logger.debug("Converted URL to lowercase, from: '" + url + "'; to: '" +
url + "'");
                }
            }

            boolean matched = matcher.pathMatchesUrl(path, url);

            if (logger.isDebugEnabled()) {
                logger.debug("Candidate is: '" + url + "'; pattern is " + path + "; matched="
+ matched);
            }

            // 找到匹配额过滤器集合
            if (matched) {
```

```
                return entry.getValue();
            }
        }

        return null;
    }

    // 在Web应用程序初始化的时候，安全框架根据配置的Bean信息生成过滤器链集合
    @SuppressWarnings("unchecked")
    public void setFilterChainMap(Map filterChainMap) {
     // 检查过滤器映射集合是否包含合法的内容
        checkContents(filterChainMap);
        uncompiledFilterChainMap = new LinkedHashMap<String,
List<Filter>>(filterChainMap);

        // 检查过滤器映射集合的路径是否有正确的顺序
        checkPathOrder();
        createCompiledMap();
    }

    @SuppressWarnings("unchecked")
    private void checkContents(Map filterChainMap) {
        for (Object key : filterChainMap.keySet()) {
          // 路径必须是字符串值
            Assert.isInstanceOf(String.class, key, "Path key must be a String but found "
+ key);
            Object filters = filterChainMap.get(key);

            // 过滤器集合必须是列表类型
            Assert.isInstanceOf(List.class, filters, "Value must be a filter list");
            // Check the contents
            Iterator filterIterator = ((List)filters).iterator();

            // 过滤器必须实现过滤器接口
            while (filterIterator.hasNext()) {
                Object filter = filterIterator.next();
                Assert.isInstanceOf(Filter.class, filter, "Objects in filter chain must be
of type Filter. ");
            }
        }
    }

    private void checkPathOrder() {
        // Check that the universal pattern is listed at the end, if at all
```

```
    // 万能匹配路径必须在最后一个
        String[] paths = (String[]) uncompiledFilterChainMap.keySet().toArray(new
String[0]);
        String universalMatch = matcher.getUniversalMatchPattern();

        for (int i=0; i < paths.length-1; i++) {
            if (paths[i].equals(universalMatch)) {
                throw new IllegalArgumentException("A universal match pattern " +
universalMatch + " is defined " +
                        " before other patterns in the filter chain, causing them to be ignored.
Please check the " +
                        "ordering in your <security:http> namespace or FilterChainProxy bean
configuration");
            }
        }
    }

    private void createCompiledMap() {
        filterChainMap = new LinkedHashMap<Object,
List<Filter>>(uncompiledFilterChainMap.size());

        // 对路径Pattern进行转换，缺省实现下没有转换
        for (String path : uncompiledFilterChainMap.keySet()) {
            filterChainMap.put(matcher.compile(path),
uncompiledFilterChainMap.get(path));
        }
    }

    private static class VirtualFilterChain implements FilterChain {
        private FilterInvocation fi;
        private List<Filter> additionalFilters;
        private int currentPosition = 0;

        private VirtualFilterChain(FilterInvocation filterInvocation, List<Filter>
additionalFilters) {
            this.fi = filterInvocation;
            this.additionalFilters = additionalFilters;
        }

        public void doFilter(ServletRequest request, ServletResponse response) throws
IOException, ServletException {
          // 如果附加的安全过滤器全部处理完毕，则调用原来的过滤器链上的过滤器
            if (currentPosition == additionalFilters.size()) {
                if (logger.isDebugEnabled()) {
```

```
                logger.debug(fi.getRequestUrl()
                        + " reached end of additional filter chain; proceeding with original
chain");
            }

            fi.getChain().doFilter(request, response);
        } else {
            currentPosition++;

            // 对附加的安全过滤器逐个进行调用
            Filter nextFilter = additionalFilters.get(currentPosition - 1);

            if (logger.isDebugEnabled()) {
                logger.debug(fi.getRequestUrl() + " at position " + currentPosition + "
of "
                        + additionalFilters.size() + " in additional filter chain; firing
Filter: '"
                        + nextFilter + "'");
            }

            nextFilter.doFilter(request, response, this);
        }
    }
}
```

6.1.1 首次登陆流程

6.1.2 登录失败流程

6.1.3 登录成功流程

6.1.4 退出登录流程

6.2 纵向剖析

6.2.1 过滤器和过滤器应用的顺序

org.springframework.security.web.context.SecurityContextPersistenceFilter@1e65cf8,
org.springframework.security.web.authentication.logout.LogoutFilter@e1cbe4,

org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@10f7a0,
org.springframework.security.web.authentication.www.BasicAuthenticationFilter@1f134b1,
org.springframework.security.web.savedrequest.RequestCacheAwareFilter@ba6b1b,
org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@cdca66,
org.springframework.security.web.authentication.AnonymousAuthenticationFilter@a3068b,
org.springframework.security.web.session.SessionManagementFilter@102b4ee,
org.springframework.security.web.access.ExceptionTranslationFilter@b499ae,
org.springframework.security.web.access.intercept.FilterSecurityInterceptor@d56fe5

6.2.2 领域模型

6.2.3 应用程序层次的安全

7 Spring集成

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

本章节将对 Spring 的分离项目 Spring 集成进行深入的剖析。

8 Spring远程调用，Spring RMI和Spring Web Service

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

本章节将对 Spring 支持的远程调用(Hessian, Burlap)，RMI 和 Web Service 进行深入的剖析。

9 与其他Web MVC框架的对比与集成

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

本章节分析 Spring 与 JSF, Struts, Webwork, Portlet 的对比和集成，包含下面的内容。

- JSF
- Struts
- Webwork
- Portlet

10 Appfuse的实现

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

本章节将对 Appfuse 的架构进行深入的剖析。

11 Spring Web与设计模式

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到robertleepeak@gmail.com。

本章节包含使用源码解析 Golf 设计模式。

12 附录

12.1 如何开始学习Spring源代码

12.1.1 J2SE/J2EE通用文档

12.1.2 Spring参考文档

Spring 框架提供了完成的参考文档，最新英文版的参考文档地址是，

http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/

12.1.3 Demo代码

Spring 框架提供了丰富的样例来阐述如何使用 Spring 开发轻量级的 JEE 应用程序，这些代码可以通过 SVN 来下载。SVN 地址是，

https://src.springframework.org/svn/spring-samples/

12.1.4 Spring源代码

由于本人时间有限，将在本书第二版完成本章节的内容。如您愿意提供帮助请发邮件到 robertleepeak@gmail.com。

12.1.4.1 Spring Web相关项目组织结构

12.2 书写习惯

在阅读 Spring 源代码的过程中发现，作者在字里行间已经写了简单的英文注释，索性保留了这些注释，让大家可以在阅读代码过程中，也能学习几句外国程序员写的英文。

（init）初始化方法和(destroy)卸载方法。