**摘要**

本文介绍了一个MVC框架，描述了该框架的功能从需求分析，概要设计，详细设计，功能实现，系统测试的过程。采用Java + Servlet的技术。已实现的功能有控制反转容器，处理器注册，处理器映射器，处理器适配器，视图解决器等。通过使用此框架，封装了后端在MVC模式下的大量重复代码，减少了开发人员的工作量，提高了工作效率。

关键词：MVC框架，Java，Servlet，Maven，Logback

**ABSTRACT**

This paper introduces an MVC framework and describes the function of the framework from requirement analysis, outline design, detailed design, function implementation and system testing. Adopt java + Servlet Technology. The implemented functions include control inversion container, processor registration, processor mapper, processor adapter, view solver, etc. By using this framework, a large amount of repetitive code of the back-end in MVC mode is encapsulated, which reduces the workload of developers and improves work efficiency.

Keywords: MVC framework, Java, Servlet, Maven, Logback

目录

[第一章 绪论 1](#_Toc1510982425)

[1.1选题的目的与意义 1](#_Toc1087491200)

[1.2国内外研究现状 1](#_Toc231278783)

[1.3本报告主要工作 1](#_Toc157104811)

[1.4复杂工程问题归纳 1](#_Toc1203156314)

[1.5报告章节安排 2](#_Toc742149246)

[第二章 相关技术研究 3](#_Toc717355746)

[2.1相关技术介绍 3](#_Toc624828764)

[2.2知识技能学习情况 3](#_Toc302002718)

[2.2.1开发基础 3](#_Toc1255823565)

[2.2.2辅助工具 4](#_Toc1157374239)

[第三章 系统需求分析 5](#_Toc81960347)

[3.1功能需求 5](#_Toc970534302)

[3.1.1功能概述 5](#_Toc1631714749)

[3.1.2功能需求 5](#_Toc863614253)

[3.2非功能需求 6](#_Toc2070263745)

[3.2.1性能需求 6](#_Toc1392713521)

[3.2.1通用性需求 6](#_Toc1911878794)

[3.2可行性研究 6](#_Toc149080697)

[第四章 系统概要设计 7](#_Toc1633342077)

[3.1系统总体架构设计 7](#_Toc296828538)

[3.2 系统模块架构设计 7](#_Toc192726185)

[第五章 系统详细设计 9](#_Toc743651619)

[5.1控制反转容器模块详细设计 9](#_Toc197934993)

[5.1处理器模块详细设计 9](#_Toc241258148)

[5.1处理器映射器模块详细设计 9](#_Toc376567900)

[5.2处理器拦截器模块详细设计 10](#_Toc342387591)

[5.3处理器适配器模块详细设计 10](#_Toc1399551624)

[5.3视图解决器模块的详细设计 12](#_Toc875758977)

[第六章 系统功能实现 13](#_Toc28209901)

[6.1控制反转容器模块实现 13](#_Toc1677403767)

[6.1.1控制反转容器启动初始化实现 13](#_Toc2107277800)

[6.1.2容器管理对象Bean创建实现 15](#_Toc717678276)

[6.1处理器映射器模块实现 17](#_Toc1750623180)

[6.1.1处理器映射器基本功能实现 17](#_Toc50338713)

[6.2.2处理器拦截器功能实现 18](#_Toc2081676120)

[6.3请求参数自动注入模块实现 18](#_Toc2074455563)

[6.5处理器适配器模块实现 20](#_Toc977638296)

[6.6视图处理器模块实现 21](#_Toc769457675)

[第七章 系统测试 22](#_Toc128621491)

[7.1 测试方法 22](#_Toc1372850355)

[7.1.2黑盒测试 22](#_Toc931613117)

[7.1.3白盒测试 22](#_Toc318387142)

[7.2 功能测试 22](#_Toc1750930917)

[7.2.1控制反转容器功能测试 22](#_Toc927507178)

[7.2.1 初始化功能测试 24](#_Toc29347073)

[7.2.2处理器模块功能测试 25](#_Toc1462500748)

[7.2.2处理器映射模块器模块测试 26](#_Toc152248074)

[7.2.3处理器适配器模块功能测试 27](#_Toc1180356141)

[7.2.4参数解决器功能测试 28](#_Toc1939214448)

[7.2.5视图解决器模块功能测试 29](#_Toc17917017)

[第八章 结束语 30](#_Toc483594139)

[7.1全文总结 30](#_Toc1688573925)

[7.2不足与下一步工作 30](#_Toc865562370)

[致谢 31](#_Toc452527812)

第一章 绪论

**1.1选题的目的与意义**

MVC框架是一种在应用程序开发中使用的一种框架，它将业务逻辑，

数据和视图分离的方式来组织代码。互联网发达的今天，

MVC框架广泛应用于各种程序中。MVC框架具有极高的可复用性，

因此，对MVC框架的研究与实现具有十分重要的意义。

**1.2国内外研究现状**

**1.3本报告主要工作**

1.实现IoC容器, 并且可以通过注解配置bean以及配置bean是原型还是单例

2.实现DispatcherServler，DispatcherServlet做到良好控制MVC框架的运行流程。

3.实现HandlerMapping，HandlerMapping做到能够根据请求找到特定的Controller。

4.实现ModelAndView，ModelAndView能够封装数据和视图信息。

5.实现ViewResolver，ViewResolver能够根据ModelAndView生成页面。

**1.4复杂工程问题归纳**

第一个问题是并未编写过复杂系统的代码，所以对于架构以及代码的扩展性，代码规范不是特别的了解。开发复杂系统需要比较强的代码架构能力，开发过程中会遇到各种未知的问题和最初没有考虑到的需求需要敏捷的做出调整和变更，所要编写的代码需要较强的可扩展性，预留许多的扩展接口，以及需要充分的考虑耦合性，适当的利用各种方法解耦。

第二个问题是编写复杂系统需要一些编程设计模式的知识。这也是自己比较薄弱的地方。编写代码的过程中，越来越发现设计模式的重要性，为了编写良好的代码，需要较强的设计模式功底。

第三个问题是不太会软件测试，以往的软件开发几乎都没有进行软件测试，为了减少bug的发生，以及增强代码的健壮性，需要各种各样不同的测试方法对软件进行系统的测试，如何高效的学习和使用软件测试的方法是一个问题。

第四个问题是对于日志系统的忽略，以往的开发几乎都不会使用日志系统，此次开发由于系统较为复杂，在dubug排错的过程中浪费了许多的时间，原因是没有良好的使用日志来对程序运行的信息进行反馈输出。

**1.5报告章节安排**

第一章主要介绍国内外研究现状。

第二章主要介绍开发所使用的相关技术。

第三章主要分析的开发框架所需的需求。

第四章主要是如何设计的整个框架的总体架构。

第五章在第四章的基础之上，详细设计了系统每个模块的功能以及大概的实现方式。

第六章主要设计的实现的流程，以及展示具体的关键的代码实现。

第七章主要是介绍测试所使用的测试方法，以及每个小模块具体使用了哪些用例进行测试。

第八章主要是结束语，总结指出不足和下一步工作。

第二章 相关技术研究

**2.1相关技术介绍**

1.开发计算机语言-Java

Java语言是1995年sun公司发布的一门高级计算机语言。Java是一门跨平台的变成语言，可以在各种不同的操作系统上运行。Java还是一个平台，由Java虚拟机和应用编程接口，所以Java几乎可以在任何操作系统中使用，并且可以一次编译，就可以在各种系统中运行。本框架使用的Java版本为Java11，

2.Tomcat容器

Tomcat服务器是一个开源的轻量级的Web应用服务器，在许多中小型和并发量比较下的场合被普遍使用。Tomcat的主要组件为服务器Server，服务Service，连接器Connector，容器Container。Tomcat最早由Sun公司的软件架构师James Duncan Davidson开发，后来于1999年于Apache软件基金会管理，变为今天的Tomcat。

3.Logback

Logback是log4j的改良版，它是log4j创始人设计的新的日志框架，它是基于slf4j接口的一个实现。它的优点非常的多，执行速度快，使用内存小，配置简单，强大的过滤能力。

4.Maven

Maven是一个项目管理工具，它包含一个项目对象模型，一个项目生命周期，一个依赖管理系统。使用Maven的好处非常的多，Maven对象jar进行统一的管理，使得开发人员不必在依赖和版本上话费过多的时间，一心将精力放在其他地方。所以使用Maven可以极大的提高开发效率。

**2.2知识技能学习情况**

**2.2.1开发基础**

1.Java

熟练使用Java的基本语法，连接Java并发多线程，集合，IO流，注解，反射，多态，泛型等多种技术。

2.tomcat服务器

已经学会熟练安装配置使用Tomcat服务器，熟悉使用Tomcat来进行软件开发。

**2.2.2辅助工具**

1.IDEA

已经熟练使用IDEA创建管理项目，集成Maven，git等工具，使用代码提示，代码跳转等操作。

2.Maven

会使用Maven创建管理项目，使用Maven清空并编译Java代码，熟悉使用各种插件。

3.Logback

熟练的使用Logback的日志级别，进行dubug，测试，输出提示信息，代码报错等常用操作。

第三章 系统需求分析

**3.1功能需求**

**3.1.1功能概述**

MVC框架首先应当实现MVC模式的概念模型，具有控制层，视图层，模型层三层模块。在此基础之上，为了提高开发者的开发效率，需要提供一些重复的通用的功能。

**3.1.2功能需求**

为了管理不同对象的依赖关系，框架需要有一个控制反转容器去管理控制各种各样的对象以及它们之间的依赖关系，并且可以动态的注册Bean和获取Bean，为了提高框架的效率以及节约内存，需要提供一些懒加载的功能。由于控制反转容器完成的对象的管理以及常见，所以还必须支持一些对象创建的设计模式，比如单例模式和原型模式。

针对于控制反转容器，由于基于的计算机语言为Java，Java习惯于使用注解做一些配置的工作，所以对于Bean的配置都应当提供基于注解的配置选项。

在有了控制反转容器之后，其他的功能需求都在控制反转容器的基础之上。为了实现代码的高可复用，处理器模块还应当提供一些帮助于代码复用的功能，常见的MVC框架一般提供拦截器的功能，定义拦截器可以把不同的处理器中的相同的代码封装到一个拦截器中。

有了配置处理器的功能，框架内部还应当又一个自动处理处理请求，然后将请求映射到处理的功能，它可以自动的将不同请求和相同请求的不同方法映射到一个配置指定的处理器中，并且还可以将处理器和拦截器封装为一个对象。

现在我们有了控制层和模型层，还差视图层。当控制层返回一个模型数据后，我们应该执行一个视图层去渲染数据，将模型层的数据渲染到视图层中，最后返回给前端，但是此时控制层返回的视图只有名字，所以我们应该有一个模块，可以通过视图的名字去寻找视图对象，所以应当有一个去完成视图名到视图对象的转化。基本的需求就足够了。

**3.2非功能需求**

**3.2.1性能需求**

框架在启动配置初始化的过程中，在配置对象不超过100个，配置处理器不超过20个的情况下，系统启动时间不超过5秒，配置对象和配置处理器过多的情况下，根据比例可以适当延长。

**3.2.1通用性需求**

做一个基于MVC模式的框架，应当具有MVC框架最基本也是对通用的功能，能够根据框架使用者的大多是业务康场景提供易于扩展的，低耦合的功能辅助。

**3.2可行性研究**

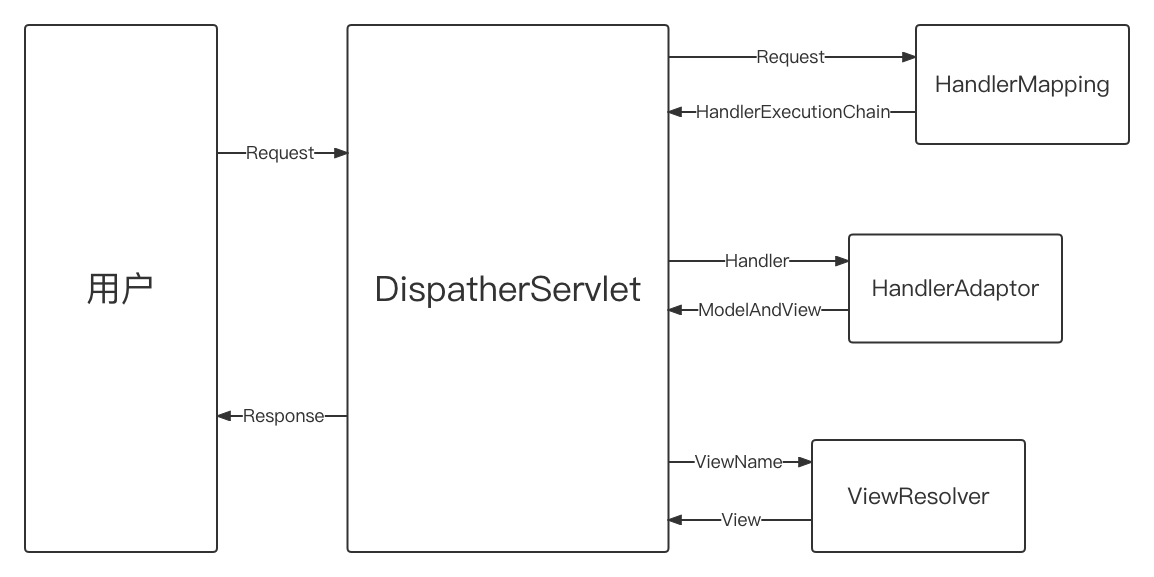
MVC模式最早由Trygve Reenskaug在1978年提出，经过40年的发展，不论是概念模型，还是各种代码的实现，都已经变得相当的成熟，优秀的MVC框架也称出不穷，这种设计模式以及其具体实现都已经非常的常见，所以MVC框架的可行性已经足够，也有非常多的MVC框架用于参考。

第四章 系统概要设计

**4.1系统主题模块概要设计**

此MVC框架

系统总体架构图如下

每次 HTTP请求执行流程：

1. 用户的HTTP请求由Servlet容器处理并分配线程传入DispatcherServlet。

2. DispatcherServlet接受请求后，向HandlerMapping传入Request对象，HandlerMapping模块根据Request对象的URI和HTTP请求方法，将处理器和所有符合条件的拦截器封装为HandlerExecutionChain对象返回给DispatcherServlet。

3. DispathcerServlet拿到处理器执行链后，调用所有拦截器的前置处理。

4. DispatcherServlet使用处理器适配器调用处理器，返回ModelAndView对象，其中包含要响应的数据和视图名。

5. DispatcherServlet向视图处理器发送视图名接受视图对象，

再由视图对象渲染视图将数据渲染到视图中。

6. 最后Servlet容器返回响应对象。

该系统属于框架项目，用于帮助开发人员提高开发效率，

该系统基于MVC模式的实现，架构图如下：

**4.2 系统模块架构设计**

DispatcherServlet是整个MVC框架最顶层的类，也是整个框架最核心的控制者，它作为管理者具有如下功能：

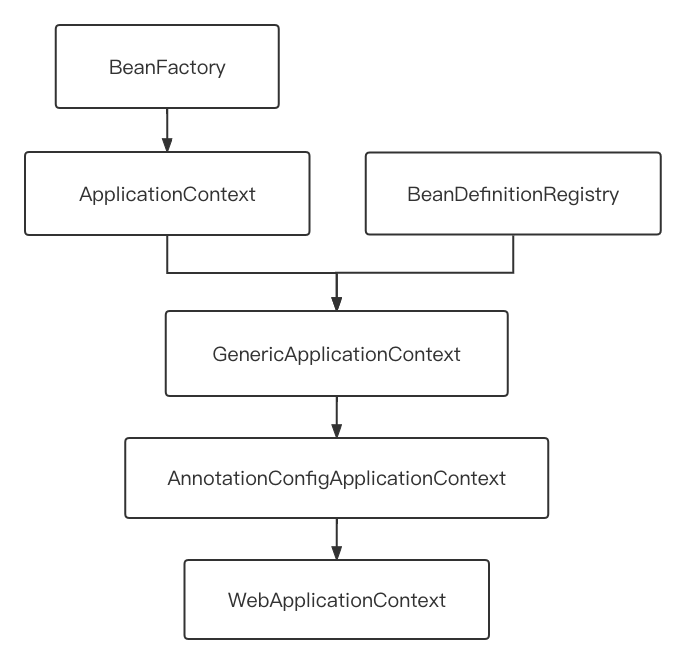
1. 初始化HandlerMapping模块，HandlerAdapter模块，ViewResolver模块，确保这些模块能够正常运行。

2. 实现完整的框架流程控制，作为整个框架饿管理者，此类必须实现良好的流程控制。

3. 此类是依托与ServletAPI实现的类，所以它必须实现Servlet的接口，并提供必要的功能。

第五章 系统详细设计

**5.1控制反转容器模块详细设计**

**5.1.1控制反转容器类的继承设计**

控制反转容器的继承关系架构如下：

1. BeanFactory，ApplicationContext，BeanDefinitionRegistry都是接口定义了，bean工厂，bean定义注册中心等接口规范。

2.GenericApplicationContext是一个抽象类，包含了作为一个工厂的通用的一些方法。

3.AnnotationConfigApplicationContext是一个实现注解定义bean的API实现类，它完全通过注解去配置bean。

4.WebApplicationContext扩展了AnnotationConfigApplicationContext类，添加了一些控制反转容器针对Web应用的方法。

**5.1.2控制反转容器的功能设计**

1.控制反转容器需要控制所有bean的定义已经创建，所以必须有bean定义的注册功能。

2.控制反转容器在初始化后，要对外提供获取对象的功能，所以必须有多种获取bean的方式。

3.控制反转容器要主动去扫描包已经器所含的bean，所以需要有bean定义类扫描的功能。

4.控制反转容器扫描获取了类后，需要从类的信息和类的注解中去获取bean的定义，由于注解的使用具有多样些，有各种各样的注解需要适配，考虑代码的扩展性，需要使用策论模式去架构代码。

5.类的关系中会出现对象的循环依赖问题，需要引入缓存控制解决循环依赖的问题。

**5.1.2控制反转容器的注解API设计**

针对使用常用的类AnnotaiontConfigApplicationContext的注解设计如下：

1.Autowired注解，用于在定义bean时，在定义的类的字段上使用，可以自动的引用其他bean。

2.Component注解，用于定义bean时，标注一个类为一个bean，在控制反转容器扫描bean时会自动配置bean。

3.Controller注解，和Component注解具有相同的功能，只是提供更加语义化的注解。

4.Service注解，和Component注解具有相同的功能，只是提供更加语义化的注解。

5.Repository注解，和Component注解具有相同的功能，只是提供更加语义化的注解。

6.Configuration注解，和Componet注解具有相同的功能，此外还具有配置类的特性。

7.ComponentScan注解，在配置类的类上使用此注解，可以定义扫描的包名，包名中的定义的bean会被自动配置。

8.Scope注解，用于在使用注解定义bean时，在类上使用，可以定义bean的创建的设计模式。

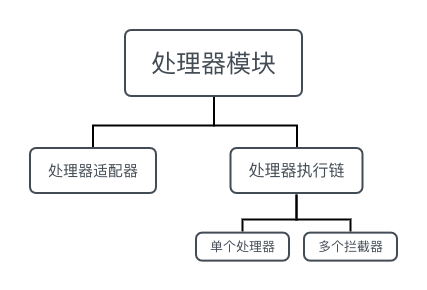
8.Lazy注解，用于使用注解定义bean时，在类上使用，可以定义单例bean在控制反转容器初始化的过程中是否加载，加了Lazy注解的单例bean会在使用时才加载。

**5.1处理器模块详细设计**

**5.2.1处理器模块的组件设计**

处理器模块主要由4个部分组成，分别是处理器，处理器拦截器，参数解决器，处理器适配器，分为4个部分的目的是为了解耦。以及将处理器中相同的代码封装为拦截器以实现代码的复用，而参数解决器存在是因为参数的处理问题一直是每个处理器都必须进行参数的获取以及参数的类型转换等步骤。

这三个部分的关系图如下所示：



处理器模块主要右两个部分组成，分别为处理器适配器和处理器执行链。处理器适配器是为了在调用处理器时进行参数的自动获取和转换，而处理器执行链时封装了处理器和拦截器的部分，拦截器可以配置多个以实现各种各样不同的功能。

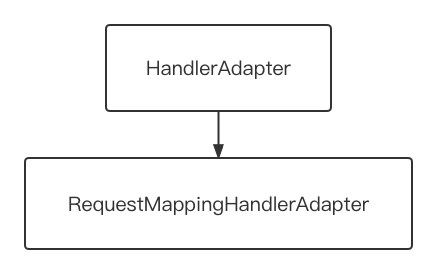
**5.2.2处理器适配器子模块详细设计**

1. 处理器适配器存在的目的是为了针对不同的处理器进行适配，故处理器适配器需要具有适配并执行处理器的功能。

2. 同时，由于在处理器适配器执行处理器的过程中，处理器需要各种各样的参数，所以处理器适配具有参数解决器的子模块，用于根据请求生成参数以便调用处理器。

处理器适配器代码架构详细设计：

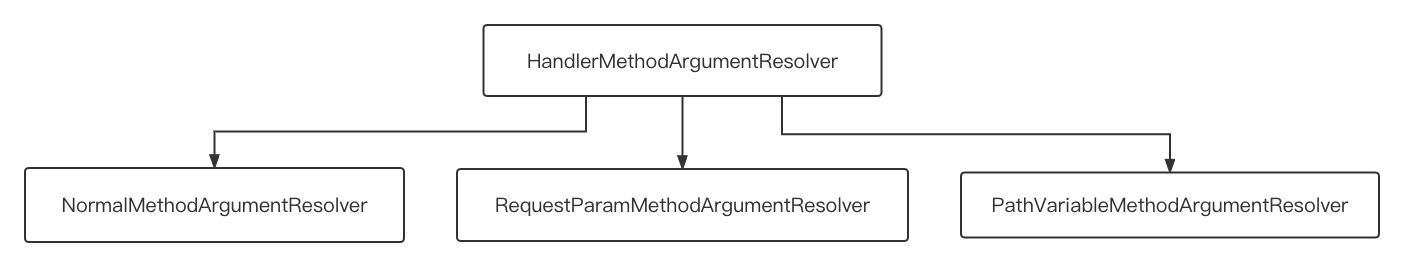
代码架构说明：

1. HandlerAdapter是一个接口，定义了处理器适配器与DispacherServlet交互的规范。

2. RequestMappingHandlerAdapter是一个基于HandlerAdapter接口的实现类，是针对HandlerMethod初期实现的处理器适配器，但处理器是HandlerMethod类型时可以使用此适配器调用处理器。

参数解决器是作为处理器适配器的子模块存在的，它存在的目的是为了解决在处理器中需要手动获取参数的问题，手动获取参数是每一个请求所必须经历的过程，所以参数解决器封装了这样一个过程，提高了代码的复用性，加快了开发的效率。

在此框架中参数解决器总共有三种类型，它们的代码架构图如下：

1. HandlerMethodArgumentResolver是一个接口，定义了参数解决器所必须的功能，以及其与处理器适配器交互的接口。

2. NormalMethodArgumentResolver是一个基于HandlerMethodArgumentResolver的实现类，此实现类是为了自动注入一些常用的与业务无关的参数，比如Request和Response，此实现类自动生效不需要使用注解定义。

3. RequestParamMethodArgumentResolver是一个基于HandlerMethodArgumetnResolver的实现类，此实现类是为了将request中蕴含的参数自动注入给处理器，使用此功能需要使用@RequestParam注解绑定到处理器的参数上，处理器适配器会根据这个注解自动使用此功能。

4. PathVariableMethodArgumentResolver是一个基于HandlerMethodArgumentResolver的实现类，此实现类是为了将URI路径中蕴含的参数自动注入给处理器使用，使用此功能要在处理器的参数前面使用@PathVariable注解，使用此注解后，在处理器适配器调用此处理器时，会将路径中的参数自动注入处理器的参数中。

**5.1处理器映射器模块详细设计**

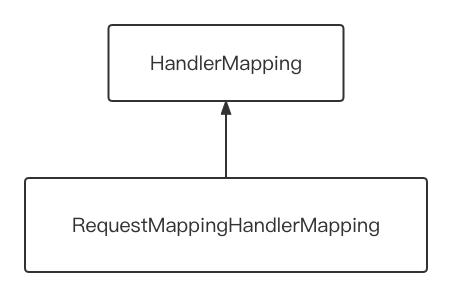
1. 处理器映射器的定位为管理所有的处理器，并为DispatcherServlet提供处理器映射，所以再初始化阶段，处理器映射器必须完成所有处理器创建与管理。

2. 处理器映射器因为要向DispatcherServlet提供获取处理器的服务，所以理所当然应当具有获取处理器执行链的功能。

3. 处理器映射器不仅仅提供最基本的功能，还应当实现同时实现拦截的功能，故处理器映射器应当具有注册拦截器已经在DispatcherServlet请求处理器执行链时，将处理器以及其对应的所有拦截器封装为一个对象返回给DispatcherServlet。

处理器映射器代码继承架构图：

代码架构说明：

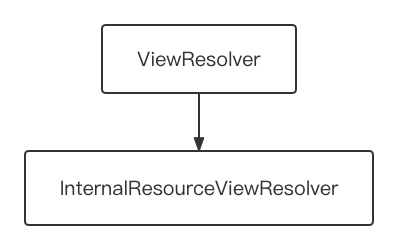
1. HandlerMapping 是一个接口，定义与DispatcherServlet交互的接口规范。

2. RequestMappingHandlerMapping 是一个继承HandlerMapping接口的具体实现类，它是针对使用@RequestMApping注解定义的处理器的处理器映射器。

**5.3视图解决器模块的详细设计**

在处理器适配调用完成处理器，处理器适配器会返回ModelAndView对象，但是在这个对象中，只蕴含了Model和视图名，所以根据MVC模式的定义，还需要通过视图名创建一个视图对象，所以视图解决器模块被设计了出来，他的主要功能是通过视图名创建一个视图对象。

其代码架构图如下：

1. ViewResolver是一个接口，它定义了视图解决器的规范，及其与DispatcherServlet的交互的接口。

2. InternalResourceViewResolver是一个基于ViewResolver的实现类，它存在的目的是为了在MVC模式中的视图模块引入JSP模块，使用此实现类即可将JSP的视图模块整合进入框架。

第六章 系统功能实现

**6.1控制反转容器模块实现**

**6.1.1控制反转容器启动初始化实现**

设计控制反转容器启动初始化流程

1. 读取配置类类名

2. 尝试将通过配置类类名获取配置类class对象，若无法获取抛出异常

3. 尝试读取包扫描路径，若没有配置包扫描路径抛出异常

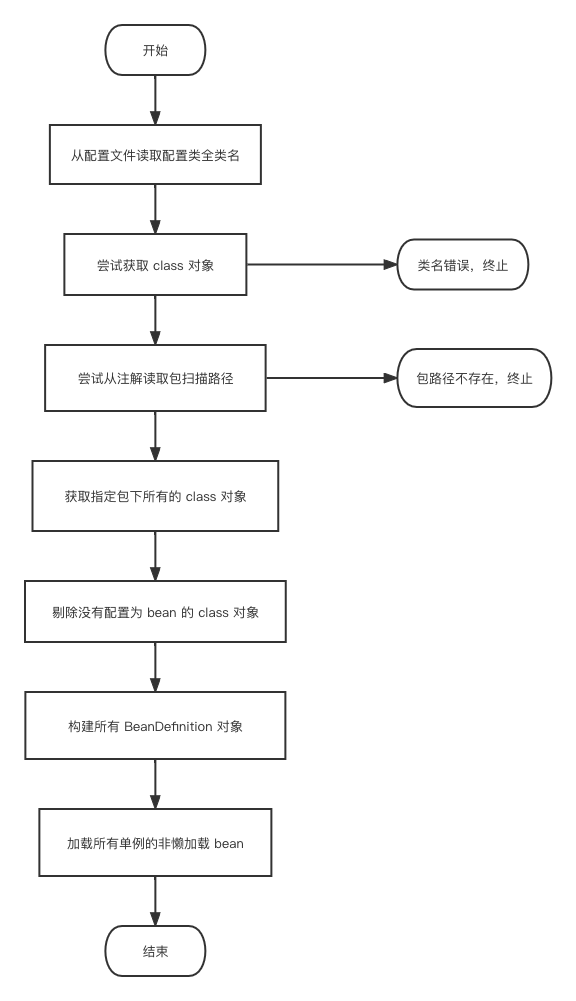
4. 扫描获取class对象。

5. 过滤所有class对象，将没有配置为Bean的class对象剔除。

6. 构建BeanDefinition对象。

7. 创建所有非懒加载单例Bean。

控制反转容器启动初始化流程图：



控制反转容器启动初始化关键代码：

String classpath = Thread.currentThread().getContextClassLoader().getResource("").getPath();

File file = null;

if ( packageName.equals("") ) {

file = new File(classpath);

} else {

String packagePath = classpath + File.separator + packageName.replace(".", File.separator);

file = new File(packagePath);

}

List<String> processingClassNameList = ToolMethods.getPathsUnderDirectory(file);

if (processingClassNameList == null) return null;

for (int index = processingClassNameList.size() - 1; index >= 0; index--) {

if ( !processingClassNameList.get(index).endsWith(".class") ) {

processingClassNameList.remove(index);

}

}

for ( int i = 0; i < processingClassNameList.size(); i++ ) {

String classname = processingClassNameList.get(i)

.substring(classpath.length())

.replace(".class", "").replace(File.separator, ".");

processingClassNameList.set(i, classname);

}

List<Class<?>> classObjects = new ArrayList<Class<?>>();

for (String classname : processingClassNameList) {

try {

Class<?> classObject = Class.forName(classname);

classObjects.add( classObject );

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

return classObjects;

**6.1.2容器管理对象Bean创建实现**

设计Bean创建流程

1. 检测请求获取的Bean是否被容器管理，如果没有返回false。

2. 如果被控制反转容器管理，则开始创建。

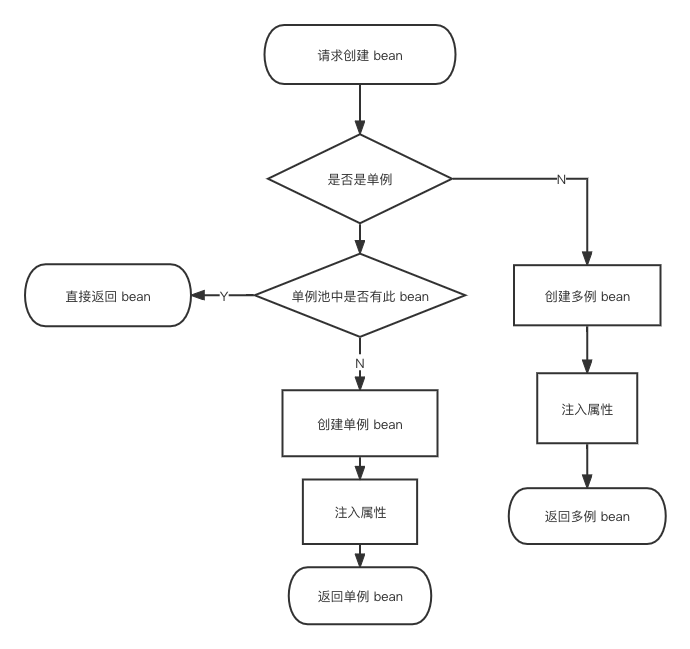
3. 判断请求获取的Bean是单例还是多例。

3. 若单例池中有，直接返回，若单例池中没有，则创建

4. 递归创建对象，创建过程中的对象，放入前期Bean池中，每次获取对象，判断是否在创建中，若是在创建中则检测到循环依赖问题。

5. 递归创建对象完成，如果是单例Bean，放入单例池中。

Bean创建流程图如下：



Bean创建流程关键代码：

String beanId = beanDefinition.getId();

Object processingSingletonObject;

processingSingletonObject = this.singletonObjects.getOrDefault(beanId, null);

if (processingSingletonObject != null) return processingSingletonObject;

processingSingletonObject = this.earlySingletonObjects.getOrDefault(beanId, null);

if (processingSingletonObject != null) return processingSingletonObject;

processingSingletonObject = beanDefinition.getBeanClass().getDeclaredConstructor().newInstance();

this.earlySingletonObjects.put(beanId, processingSingletonObject);

for (Map.Entry<Field,Object> entry : beanDefinition.getFieldMap().entrySet()) {

Field field = entry.getKey();

field.setAccessible(true);

String fieldBeanId = ((BeanId) entry.getValue()).getValue();

field.set( processingSingletonObject, this.getBean(fieldBeanId) );

}

this.earlySingletonObjects.remove(beanId);

this.singletonObjects.put(beanId, processingSingletonObject);

return processingSingletonObject;

**6.1处理器映射器模块实现**

**6.1.1处理器映射器基本功能实现**

关键代码实现

String URI = request.getRequestURI();

String requestMethod = request.getMethod();

HandlerMethod handlerMethod = this.handlerMethodMap.getOrDefault(URI, null);

if (handlerMethod != null) {

for (RequestMethod method : handlerMethod.getRequestMethods()) {

if ( method.toString().equals(requestMethod) ) {

chain.setHandler(handlerMethod);

return true;

}

}

return false;

}

for (Map.Entry<String,HandlerMethod> entry : this.handlerMethodMap.entrySet()) {

boolean isMatching = URI.matches( entry.getKey() );

if (isMatching) {

for ( RequestMethod method : entry.getValue().getRequestMethods() ) {

if ( method.toString().equals(requestMethod) ) {

chain.setHandler( entry.getValue() );

return true;

}

}

return false;

}

}

return false;

**6.2.2处理器拦截器功能实现**

关键代码实现：

HttpServletRequest processedRequest = checkMultipart(request);

HandlerExecutionChain handlerExecutionChain = getHandler(request);

if (handlerExecutionChain == null) {

try {

if (this.tryFindStaticResource(request, response)) {

return;

}

}

catch (IOException exception) {

exception.printStackTrace();

}

response.setStatus(404);

return;

}

if (!handlerExecutionChain.applyPreHandle(request, response)) {

return;

}

ModelAndView modelAndView = this.handlerAdapter.handle( request, response, handlerExecutionChain.getHandler() );

handlerExecutionChain.applyPostHandle(request, response, modelAndView)

if (modelAndView.getViewName() != null) {

this.render(request, response, modelAndView);

}

else {

response.setStatus(404);

}

**6.3请求参数自动注入模块实现**

关键代码

Method method = handlerMethod.getMethod();

Parameter[] parameters = handlerMethod.getParameters();

Object[] arguments = new Object[parameters.length];

for (int i = 0; i < parameters.length; i++) {

MethodParameter methodParameter = new MethodParameter(method, i);

for (HandlerMethodArgumentResolver argumentResolver : this.argumentResolverList) {

if ( argumentResolver.supportsParameter(methodParameter) ) {

arguments[i] = argumentResolver.resolveArgument(request, response, handlerMethod, methodParameter);

break;

}

}

}

Object returnObject = null;

Object controller = handlerMethod.getController();

try {

returnObject = method.invoke(controller, arguments);

}

catch (IllegalAccessException | InvocationTargetException e) {

e.printStackTrace();

}

if (returnObject == null) {

return null;

}

else if (returnObject.getClass() != ModelAndView.class && returnObject.getClass() != String.class) {

return null;

}

else {

return returnObject;

}

**6.5处理器适配器模块实现**

关键代码

Method method = handlerMethod.getMethod();

Parameter[] parameters = handlerMethod.getParameters();

Object[] arguments = new Object[parameters.length];

for (int i = 0; i < parameters.length; i++) {

MethodParameter methodParameter = new MethodParameter(method, i);

for (HandlerMethodArgumentResolver argumentResolver : this.argumentResolverList) {

if ( argumentResolver.supportsParameter(methodParameter) ) {

arguments[i] = argumentResolver.resolveArgument(request, response, handlerMethod, methodParameter);

break;

}

}

}

Object returnObject = null;

Object controller = handlerMethod.getController();

try {

returnObject = method.invoke(controller, arguments);

}

catch (IllegalAccessException | InvocationTargetException e) {

e.printStackTrace();

}

if (returnObject == null) {

return null;

}

else if (returnObject.getClass() != ModelAndView.class && returnObject.getClass() != String.class) {

return null;

}

else {

return returnObject;

}

**6.6视图处理器模块实现**

关键代码

for (Map.Entry<String,Object> entry : model.entrySet()) {

request.setAttribute(entry.getKey(), entry.getValue());

}

try {

request.getRequestDispatcher(this.URL).forward(request, response);

}

catch (ServletException | IOException e) {

e.printStackTrace();

}

第七章 系统测试

**7.1 测试方法**

**7.1.2黑盒测试**

黑盒测试

**7.1.3白盒测试**

白盒测试

**7.2 功能测试**

**7.2.1控制反转容器功能测试**

初始化功能测试如表7.1所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试目的 | 测试控制反转容器功能是否能够正常运行 | | | | | | |
| 编写人 | 张炼 | 时间 | 2018-5-16 | | 状态 | 检测结果为合格 | |
| 前置条件 | 系统能够正常启动 | | | | | | |
| 序号 | 测试步骤 | | | 期待结果 | | | 是否通过 |
| 1 | 不配置控制饭庄容器的配置类，启动容器 | | | 输出配置类未被正确配置，控制反转容器无法正确启动 | | | 通过，和预期结果一致 |
| 2 | 配置不存在的配置类，启动容器 | | | 输出配置类不存在，无法正确启动容器 | | | 通过，和预期结果一致 |
| 3 | 配置正确配置类，但是没有配置扫描包路径 | | | 输出没有配置扫描包路径，无法正确启动容器 | | | 通过，和预期结果一致 |
| 4 | 配置正确配置类，并且正确配置扫描包路径 | | | 容器器正常启动，没有输出任何提示信息 | | | 通过，和预期结果一致 |
| 4 | 使用@Component，并在容器初始化的代码中输出日志信息，查看Bean是否被正确管理 | | | 正确输出对应的日志信息，表示Bean已经被正确管理 | | | 通过，和预期结果一致 |
| 5 | 使用@Controller注解，并在容器初始化的代码中输出日志信息，查看Bean是否被正确管理 | | | 正确输出对应的日志信息，表示Bean已经被正确管理 | | | 通过，和预期结果一致 |
| 6 | 使用@Service注解，并在容器初始化的代码中输出日志信息，查看Bean是否被正确管理 | | | 正确输出对应的日志信息，表示Bean已经被正确管理 | | | 通过，和预期结果一致 |
| 7 | 使用@Repository注解，并在容器初始化的代码中输出日志信息，查看Bean是否被正确管理 | | | 正确输出对应的日志信息，表示Bean已经被正确管理 | | | 通过，和预期结果一致 |
| 8 | 使用@Autowired注解，并在容器初始化的代码中输出日志信息，查看Bean是否被正确管理 | | | 正确输出对应的日志信息，表示字段已经被正确注入 | | | 通过，和预期结果一致 |
|  | 使用@Lazy注解，并在使用注解的Bean被创建时输出日志信息，查看注解是否正确生效 | | | 正确输出日志信息，表示@Lazy注解生效 | | |  |
|  | 使用@Sope注解，并配置Bean为多例在容器创建Bean定义时，日志输出Bean的作用域 | | | 正确输出日志信息，表示配置Bean为多例成功 | | |  |
|  | 使用@Sope注解，并配置Bean为单例在容器创建Bean定义时，日志输出Bean的作用域 | | | 正确输出日志信息，表示配置Bean为单例成功 | | |  |
|  | 在使用依赖注入时，配置一个会循环依赖的单例Bean，查看Bean是否能够被正确定义并创建 | | | 正常输出对应的日志信息，表示配置的单例循环Bean被定义成功 | | |  |
|  | 在使用依赖注入时，配置一个会循环依赖的多例Bean，查看Bean是否能够被正确定义并创建 | | | 正常输出对应的日志信息，表示配置的单例循环Bean被定义成功 | | |  |
|  | 在配置的扫描包路径下，放置非class文件，检测在有不期待文件下，容器的运行是否成功 | | | 容器正常启动，表示容器具有检测未知文件并处理的能力 | | |  |
|  | 同时使用多个定义Bean的注解，并在Bean定义过程中输出使用定义Bean的注解 | | | 容器正常启动，表示容器可能多个定义Bean的注解 | | |  |
|  | 定义多个单例Bean和多个多例Bean，在容器初始化工程中，输出被创建的Bean的id | | | 正确输出所有单例Bean的id，而未输出多例Bean的id | | |  |

**7.2.1 初始化功能测试**

测试在框架初始化的过程中会不会出现任何的问题，需要保证系统能够正常的初始化，初始化功能测试如表7.1所示：

表7.1初始化功能测试

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试目的 | 测试在系统其中后，初始化工程能否正确执行 | | | | | | |
| 编写人 | 张炼 | 时间 | 2018-5-16 | | 状态 | 检测结果为合格 | |
| 前置条件 | 系统能够正常启动 | | | | | | |
| 序号 | 测试步骤 | | | 期待结果 | | | 是否通过 |
| 1 | 直接启动系统，不配置初始化配置类 | | | 启动初始化中断，并其实需要配置初始化配置类 | | | 通过，和预期结果一致 |
| 2 | 配置初始化配置类，然后启动系统 | | | 系统能够正常启动 | | | 通过，和预期结果一致 |
| 3 | 不配置Controller，启动系统 | | | 启动成功，尝试化正常 | | | 通过，和预期结果一致 |
| 4 | 配置Controller，启动系统 | | | 启动成功，初始化正常 | | | 通过，和预期结果一致 |
| 4 | 配置Controller，不配置URI，启动系统 | | | 启动成功，初始化正常 | | | 通过，和预期结果一致 |
| 5 | 配置Controller，配置URI，启动系统 | | | 启动成功，初始化正常 | | | 通过，和预期结果一致 |
| 6 | 配置Controller，配置URI，不配置请求方法 | | | 启动成功，初始化正常 | | | 通过，和预期结果一致 |
| 7 | 配置Controller，配置URI，配置请求方法 | | | 启动成功，初始化正常 | | | 通过，和预期结果一致 |
| 8 | 正确配置Controller，不配置拦截器，启动系统 | | | 启动成功，初始化正常 | | | 通过，和预期结果一致 |
| 9 | 正确配置Controller，配置拦截器 | | | 启动成功，初始化正常 | | | 通过，和预期结果一致 |

**7.2.2处理器模块功能测试**

处理器映射器模块测试如表7.2所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试目的 | 测试处理器模块是否能够正确运行 | | | | | | |
| 编写人 | 张炼 | 时间 | 2018-5-16 | | 状态 | 检测结果为合格 | |
| 前置条件 | 系统能够正常启动 | | | | | | |
| 序号 | 测试步骤 | | | 期待结果 | | | 是否通过 |
| 1 | 在方法上使用请求映射注解，查看URI是否被正确的配置到处理器中 | | | 处理器被正确配置 | | | 通过，和预期结果一致 |
| 2 | 在类上使用请求映射注解，不再方法上使用，看URI是否被正确配置到处理器中 | | | 处理器陪正确配置 | | | 通过，和预期结果一致 |
| 3 | 同时在类和方法上使用请求映射注解，插卡URI是否被正确配置到处理器中 | | | 处理器被正确配置 | | | 通过，和预期结果一致 |
| 4 | 使用请求映射注解但是不配置请求方法，日志输出被配置的所有请求方法 | | | 默认的GET请求方法被正确配置当处理器对象当中 | | | 通过，和预期结果一致 |
| 4 | 使用请求映射注解同时配置单个请求方式，日志输出被配置的所有请求方法 | | | 配置的请求方法被日志正确输出 | | | 通过，和预期结果一致 |
| 5 | 使用请求映射注解同时配置多个请求方法，日志输出被配置的所有请求方法 | | | 配置的多个请求方法被日志正确输出 | | | 通过，和预期结果一致 |
| 6 | 不配置拦截器，日志输出配置的拦截器 | | | 没有任何拦截器被日志输出 | | | 通过，和预期结果一致 |
| 7 | 配置单个拦截器，映射所有的请求URI，日志输出查看不同的处理器是否都同时被拦截 | | | 在不同的请求处理器被执行过程中，同一个拦截器都被正确输出 | | | 通过，和预期结果一致 |
| 8 | 配置多个拦截器，日志输出看容同一个处理器器是否被多个拦截器拦截 | | | 多个拦截器名都被正确输出 | | | 通过，和预期结果一致 |
| 9 | 配置多个拦截器，日志输出查看拦截器前置处理是否安装设计顺序正确执行 | | | 拦截器的前置处理按照循序正确执行 | | | 通过，和预期结果一致 |
|  | 配置多个拦截器，日志输出查看拦截器后置处理是否按照设计顺序正确执行 | | | 拦截器的后置处理按照设计顺序正确执行 | | |  |
|  | 配置单个拦截器，日志输出查看多个拦截器是否和处理一同封装为处理器执行链对象 | | | 多个拦截器和处理器正确被封装为处理器执行链对象 | | |  |
|  | 配置多个拦截器，日志输出查看多个拦截器是否和处理器一同被封装为处理器执行链对象 | | | 多个拦截器和处理器被正确的封装为处理器执行链对象 | | |  |
|  | 配置处理器，返回值设置为非ModelAndView对象，查看日志是否能够检测并输出提示 | | | 日志检测被正确输出 | | |  |
|  | 配置处理器，返回值设置为ModelAndView对象，查看是否能够正常运行 | | | 正常运行 | | |  |
|  | 配置两个处理器，请求URI配置成不冲突的，请求查看运行 | | | 不同的URI请求分配到了不同的处理器 | | |  |
|  | 配置两个处理器，请求URI冲突，请求查看运行 | | | 请求被分配到了先配置的URI | | |  |

**7.2.2处理器映射模块器模块测试**

测试在系统启动后，处理器映射模块能否正常运行，处理器映射器模块测试如表7.2所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试目的 | 测试处理器映射器能否正常配置使用 | | | | | | |
| 编写人 | 张炼 | 时间 | 2018-5-16 | | 状态 | 检测结果为合格 | |
| 前置条件 | 系统能够正常启动 | | | | | | |
| 序号 | 测试步骤 | | | 期待结果 | | | 是否通过 |
| 1 | 不进行任何配置，在代码中输出已经管理的处理器URI | | | 不输出任何处理器URI | | | 通过，和预期结果一致 |
| 2 | 配置一个处理器，在代码中输出已经管理的URI | | | 输出一个对应的处理器的URI | | | 通过，和预期结果一致 |
| 3 | 配置多个处理器，在代码中输出已经管理的URI | | | 输出多个对应的处理器的URI | | | 通过，和预期结果一致 |
| 4 | 在处理中使用路径参数，在代码中输出URI | | | 输出对应的URI | | | 通过，和预期结果一致 |
| 4 | 配置一个处理器，使用多个请求方法，在代码中输出所以请求方法 | | | 正确输出处理器的所有请求方法 | | | 通过，和预期结果一致 |
| 5 | 配置一个处理器，使用一个请求方法，在代码中输出所以请求方法 | | | 正确输出一个处理器的请求方法 | | | 通过，和预期结果一致 |
| 6 | 配置拦截器，在代码中输出拦截器的类名 | | | 正确输出拦截器的类名 | | | 通过，和预期结果一致 |
| 7 | 配置多个拦截器，在代码中输出拦截器的类名 | | | 正确输出多个拦截器的雷名 | | | 通过，和预期结果一致 |
| 8 | 配置处理器和拦截器，在Servlet中输出处理器执行链 | | | 正确输出处理器执行链 | | | 通过，和预期结果一致 |

**7.2.3处理器适配器模块功能测试**

处理器适配器模块功能测试如表7.3所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试目的 | 测试处理器映射器能否正常配置使用 | | | | | | |
| 编写人 | 张炼 | 时间 | 2018-5-16 | | 状态 | 检测结果为合格 | |
| 前置条件 | 系统能够正常启动 | | | | | | |
| 序号 | 测试步骤 | | | 期待结果 | | | 是否通过 |
| 1 | 不进行任何配置，在代码中输出已经管理的处理器URI | | | 不输出任何处理器URI | | | 通过，和预期结果一致 |
| 2 | 配置一个处理器，在代码中输出已经管理的URI | | | 输出一个对应的处理器的URI | | | 通过，和预期结果一致 |
| 3 | 配置多个处理器，在代码中输出已经管理的URI | | | 输出多个对应的处理器的URI | | | 通过，和预期结果一致 |
| 4 | 在处理中使用路径参数，在代码中输出URI | | | 输出对应的URI | | | 通过，和预期结果一致 |
| 4 | 配置一个处理器，使用多个请求方法，在代码中输出所以请求方法 | | | 正确输出处理器的所有请求方法 | | | 通过，和预期结果一致 |
| 5 | 配置一个处理器，使用一个请求方法，在代码中输出所以请求方法 | | | 正确输出一个处理器的请求方法 | | | 通过，和预期结果一致 |
| 6 | 配置拦截器，在代码中输出拦截器的类名 | | | 正确输出拦截器的类名 | | | 通过，和预期结果一致 |
| 7 | 配置多个拦截器，在代码中输出拦截器的类名 | | | 正确输出多个拦截器的雷名 | | | 通过，和预期结果一致 |
| 8 | 配置处理器和拦截器，在Servlet中输出处理器执行链 | | | 正确输出处理器执行链 | | | 通过，和预期结果一致 |

**7.2.4参数解决器功能测试**

参数解决器功能测试如表7.4所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试目的 | 测试参数解决器是否能够正常使用 | | | | | | |
| 编写人 | 张炼 | 时间 | 2018-5-16 | | 状态 | 检测结果为合格 | |
| 前置条件 | 系统能够正常启动 | | | | | | |
| 序号 | 测试步骤 | | | 期待结果 | | | 是否通过 |
| 1 | 不配置任何参数注入选项 | | | 无参数解决器也能正常运行 | | | 通过，和预期结果一致 |
| 2 | 在处理器中使用Request，请求测试处理器 | | | 输出request，request不为空，正常注入 | | | 通过，和预期结果一致 |
| 3 | 在处理器中使用Response，请求测试处理器 | | | response不为空，正常注入 | | | 通过，和预期结果一致 |
| 4 | 在处理器中使用ModelAndView请求参数处理器 | | | modelAndView正常注入 | | | 通过，和预期结果一致 |
| 4 | 在处理器中使用@RequestParam注解，请求处理器 | | | 正确注入注解指定的参数 | | | 通过，和预期结果一致 |
| 5 | 在处理器中使用多个请求参数注解，请求测试处理器URI | | | 正确注入多个注解指定的参数 | | | 通过，和预期结果一致 |
| 6 | 在处理器中使用@PathVariable注解，请求测试处理器URI | | | 正确注入注解指定的路径参数 | | | 通过，和预期结果一致 |
| 7 | 在处理器中使用多个路径参数注解，请求测试处理器URI | | | 正确注入多个注解指定的路径参数 | | | 通过，和预期结果一致 |
| 8 | 在处理器中使用无法注入的请求参数，请求处理器URI | | | 无法注入参数，参数为空 | | | 通过，和预期结果一致 |

**7.2.5视图解决器模块功能测试**

参数解决器功能测试如表7.4所示：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 测试目的 | 测试视图解决器组件功能是否运行正常 | | | | | | |
| 编写人 | 张炼 | 时间 | 2018-5-16 | | 状态 | 检测结果为合格 | |
| 前置条件 | 系统能够正常启动 | | | | | | |
| 序号 | 测试步骤 | | | 期待结果 | | | 是否通过 |
| 1 | 不做任何配置，启动系统 | | | 内部资源视图解决器嫩嫩巩固被默认的自动配置 | | | 通过，和预期结果一致 |
| 2 | 配置视图解决器前缀 | | | 视图解决器前缀配置正确 | | | 通过，和预期结果一致 |
| 3 | 配置视图解决器后缀 | | | 视图解决器后缀配置正确 | | | 通过，和预期结果一致 |
| 4 | 在没有配置前缀后缀的情况下，请求视图对象 | | | 视图对象正确生成 | | | 通过，和预期结果一致 |
| 4 | 在配置前缀的情况下请求视图对象 | | | 视图对象正确生成 | | | 通过，和预期结果一致 |
| 5 | 在配置后缀的情况下请求视图对象 | | | 视图对象正确生成 | | | 通过，和预期结果一致 |
| 6 | 在同时配置前缀和后缀的情况下请求视图对象 | | | 视图对象正确生成 | | | 通过，和预期结果一致 |
| 7 | 在没有模型数据的情况下，使用视图对象调用渲染功能 | | | 渲染功能运行正常，模型数据被正确注入JSP页面的request属性中 | | | 通过，和预期结果一致 |
| 8 | 在有单个模型数据的情况下，使用视图对象调用渲染功能 | | | 渲染功能运行正常，在JSP页面的request属性中有应有模型数据 | | | 通过，和预期结果一致 |
| 9 | 在有多个模型数据的情况下，使用视图对象条用渲染功能 | | | 渲染功能运行正常，在JSP页面的request属性中有应有的模型数据 | | | 通过，和预期结果一致 |

第八章 结束语

**7.1全文总结**

本文从相关技术研究，系统需求分析，系统概要设计，系统功能实现，系统测试5个方面阐述了MVC框架的信息。

1.首先对相关技术进行调研，了解了目前相关的主要MVC框架的设计，以及实现技术

2.根据调研获取的信息分析实现的框架的主要需要实现的功能.

3.根据需求的分析和低耦合，高内聚的设计思想设计除了整体的架构以及实现功能的各个模块。

4.整个框架使用了许多代码设计模式，以便于提高代码的扩展性和可维护性。

**7.2不足与下一步工作**

本系统

# 致谢

论文是在唐开山老师的指导下完成的，十分感谢唐开山老师，在编写论文的过程中提供了很多的帮助。