毕业设计（论文）检测系统

文本复制检测报告单(全文标明引文)

№:BC20220502213725292645 3108　　检测时间:2022-05-02 21:37:25

篇名: **基于MVC模式的框架研究与实现**

作者: 张炼(1840611416;计算机学院;软件工程)

指导教师: 唐开山

检测机构: 电子科技大学成都学院

提交论文IP: 125.***.***.***

文件名: 论文.docx

检测系统: 毕业设计（论文）检测系统（毕业设计（论文）管理系统）

检测类型: 毕业设计论文

检测范围: 中国学术期刊网络出版总库
中国博士学位论文全文数据库/中国优秀硕士学位论文全文数据库
中国重要会议论文全文数据库
中国重要报纸全文数据库
中国专利全文数据库
图书资源
优先出版文献库
大学生论文联合比对库
互联网资源(包含贴吧等论坛资源)
英文数据库(涵盖期刊、博硕、会议的英文数据以及德国Springer、英国Taylor&Francis 期刊数据库等)
港澳台学术文献库
互联网文档资源
源代码库
CNKI大成编客-原创作品库

时间范围: 1900-01-01至2022-05-02

检测结果

去除本人文献复制比: 3.6%　　跨语言检测结果: 0%

去除引用文献复制比: 3.6%　　总文字复制比: 3.6%

单篇最大文字复制比: 0.7%　（一种轻量级的JavaWeb框架的设计与实现）

重复字数: [1367]　　总段落数: [10]

总字数: [37678]　　疑似段落数: [7]

单篇最大重复字数: [255]　　前部重合字数: [266]

疑似段落最大重合字数: [441]　　后部重合字数: [1101]

疑似段落最小重合字数: [51]

指标: ☑ 疑似剽窃观点　☑ 疑似剽窃文字表述　☐ 疑似整体剽窃　☐ 过度引用

相似表格: 0　　相似公式: 没有公式　　疑似文字的图片: 0

0%(0)	0%(0)	基于MVC模式的框架研究与实现_第1部分（总806字）
4.1%(74)	4.1%(74)	基于MVC模式的框架研究与实现_第2部分（总1798字）
20.5%(192)	20.5%(192)	基于MVC模式的框架研究与实现_第3部分（总937字）
0%(0)	0%(0)	基于MVC模式的框架研究与实现_第4部分（总1067字）

0%(0)	0%(0)	基于MVC模式的框架研究与实现_第5部分（总1548字）
3.8%(147)	3.8%(147)	基于MVC模式的框架研究与实现_第6部分（总3870字）
1.5%(207)	1.5%(207)	基于MVC模式的框架研究与实现_第7部分（总13498字）
28.7%(255)	28.7%(255)	基于MVC模式的框架研究与实现_第8部分（总887字）
0.7%(51)	0.7%(51)	基于MVC模式的框架研究与实现_第9部分（总7729字）
8%(441)	8%(441)	基于MVC模式的框架研究与实现_第10部分（总5538字）

（注释：无问题部分　　文字复制部分　　引用部分）

疑似剽窃观点（1）

基于MVC模式的框架研究与实现_第10部分

1. 简而言之，无论任何时候发生什么样的数据更改，控制器都会将这些更改通知所有相关视图，并更新相关显示。

1. 基于MVC模式的框架研究与实现_第1部分　　总字数：806

相似文献列表

去除本人文献复制比：0%(0)　　文字复制比：0%(0)　　疑似剽窃观点：(0)

原文内容

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和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

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 and 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1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。

MVC意味着三层模块的组织架构，分别为控制层，视图层，模型层。三层各自负责自己的模块，将代码进行总体的解耦。

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，只要是拥有页面和数据的程序都可以使用MVC框架。

因此，对MVC框架的研究与实现具有十分重要的意义。而良好的可维护的MVC框架对于整个MVC框架的发展起到推动作用。

1.2国内外研究现状

Struts框架是2001年发布，2004年开始逐渐火热，成为Java Web应用开发最流行的框架之一。拥有广泛的市场占用率和丰富的开发人群。但是随着经济的发展和技术的不断更新，Struts框架的局限性开始暴露出来。由于JSP，Servlet的耦合性非常的紧密，导致了许多的问题。

为了满足更加灵活，高效的开发需求，Struts2应运而生。作为新一代的Struts框架，其本质是在Struts和Webwork技术的基础之上进行合并后的全新框架。Struts2是以WebWork为核心，通过应用拦截器的机制原理来处理用户的请求，这样使得业务的逻辑控制模块能够与ServletAPI完全脱离开来。但由于Struts框架出现过许多次致命的漏洞，造成了安全问题，许多企业放弃了Struts框架。

随着Spring快速的发展，使用SpringMVC的人也越来越多，Spring MVC也确实更加灵活，Spring MVC逐渐淘汰了Struts2.

1.3本报告主要工作

本报告主要介绍MVC框架的设计和实现过程，包括：

1.系统需求分析

详细分析系统所需要的功能，以及对应功能的性能需求和通用型需求。

2.系统概要设计

因为是一个较为复杂的代码组织架构，所以需要先进行大的模块的设计，主要包括系统主要模块的概要设计极易不同模块之间的架构设计。

3.系统详细设计

在系统概要设计的基础之上，详细设计包括：控制反转容器的详细设计，处理器的详细设计，处理器拦截器的详细设计，处理器映射器的详细设计，视图解决器的详细设计。

4.系统功能实现

在系统详细设计的基础之上，根据设计的架构介绍实现的关键代码，包括：控制反转容器的代码介绍与实现，处理器映射器的代码介绍与实现，请求参数自动注入模块的代码介绍与实现，处理器拦截器的代码介绍与实现，处理器适配器的代码介绍与实现，视图解决器的代码介绍与实现。

5.系统测试

完成代码的编写后，进行系统的测试，包括测试方法的介绍和详细的测试用例表。

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。

3. 基于MVC模式的框架研究与实现_第3部分　　总字数：937

相似文献列表

去除本人文献复制比：20.5%(192)　　文字复制比：20.5%(192)　　疑似剽窃观点：(0)

1	fc27_吕炳劭_管道管理系统 肖倩 - 《高职高专院校联合比对库》- 2020-05-20	12.3%（115） 是否引证：否
2	基于视频识别的小区智能停车位管理系统的设计与实现 赵家辉(导师：白恩健) - 《东华大学硕士论文》- 2020-06-03	9.9%（93） 是否引证：否
3	8997881_王启名_中小企业网络考勤管理系统的设计与实现 王启名 - 《高职高专院校联合比对库》- 2020-05-27	5.1%（48） 是否引证：否
4	口腔诊所预约挂号系统的设计与实现 陈海权 - 《大学生论文联合比对库》- 2019-05-31	4.7%（44） 是否引证：否
5	MPLS VPN用户行为挖掘系统的分析与设计 张俊新(导师：罗守山) - 《北京邮电大学硕士论文》- 2011-05-01	3.3%（31） 是否引证：否

原文内容

第2章相关技术研究

2.1相关技术介绍

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。

Java语言具有众多的版本，有针对企业的，针对个人的，Java在今天也在不断的发展中，通常人们会使用的Java版本一般是Java8或者Java11。Java8提供的常用的功能，对于大多数开发者已经足够使用了；而Java11由于是一个长期维护版本，同时较新的版本也能够对开发提供更好的支持。本框架使用的Java版本为Java11。

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。

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。

指　　标

疑似剽窃文字表述

1. 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
2. Tomcat最早由Sun公司的软件架构师James Duncan Davidson开发，

4. 基于MVC模式的框架研究与实现_第4部分　　总字数：1067

相似文献列表

去除本人文献复制比：0%(0)　　文字复制比：0%(0)　　疑似剽窃观点：(0)

原文内容

第3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。

3.2.2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。

3.3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。

5. 基于MVC模式的框架研究与实现_第5部分　　总字数：1548

相似文献列表

去除本人文献复制比：0%(0)　　文字复制比：0%(0)　　疑似剽窃观点：(0)

原文内容

第4章系统概要设计

4.1系统主体架构设计

此框架是基于Servlet5.0接口标准实现的MVC框架，整体采用模块化的设计，框架主要有4个模块，分别是DispatcherServlet模块，HandlerMapping模块，HandlerAdaptor模块，ViewResolver模块。每个模块负责实现框架中的一部分功能。

系统总体架构图如图4-1：

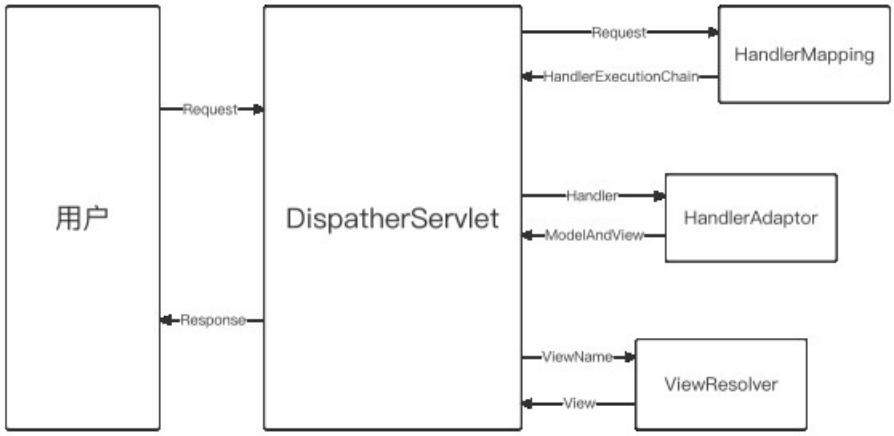

图4-1 系统总体架构图

每次HTTP请求都会经过如下这些执行流程：

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，判断是否有拦截器，如果有则按照配置顺序调用所有拦截器的前置处理。如果前置拦截中有任何一个处理器拦截器拦截了请求，则直接返回响应，不再进行下面的流程。

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.在调用处理器后，按照处理器拦截器配置顺序调用所有处理器拦截器的后置处理。

6.DispatcherServlet通过处理器返回的ModelAndView中的视图名向视图处理器请求视图对象，如果没有视图对象，则抛出异常。

7.最后由视图对象将数据渲染到视图中去。

8.最后Servlet容器返回经过框架处理的响应对象。

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.接受HTTP请求，并将请求按照框架设计的处理器流程处理，并返回HTTP响应。DispatcherServlet本质是一个实现了Servlet接口的类，所以必须实现Servlet的基本功能。

2.初始化HandlerMapping模块，HandlerAdapter模块，ViewResolver模块，确保这些模块能够正常运行，因为这三个模块不能够自己初始化，初始化也较为复杂，所以需要DispatcherServlet对这三个模块进行初始化。

3.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。

HandlerMapping模块是作为一大组件在DispatcherServlet中的，它主要控制处理器的映射关系，它的主要功能如下：

1.作为控制所有处理器的模块，它要提供注册处理器的功能，也要提供获取处理器的功能。

2.作为控制所有拦截器的模块，它要提供注册拦截器的功能，也要提供获取拦截器的功能。

3.它能够提供接口，只要实现此接口，就能定义不同的处理器映射器，增强框架的扩展性。

处理器适配器模块也是整个框架的主要模块之一，它主要是提供不同的调用各种处理器的功能，它的主要功能如下：

1.提供检测处理器是否支持的功能，由于可能有不同的处理器类型，所以必须检测是否支持处理器。

2.通过处理器适配器调用处理器的功能，可以传入处理器的对象，使得处理器适配器调用处理器，并返回处理器的结果。

视图解决器模块也是作为整个框架主要的组件之一，它主要的功能是根据视图名生成视图对象。主要功能如下：

1.根据视图名，生成视图对象。

2.可以提供默认的视图名前缀和后缀设置功能。

第5章系统详细设计

5.1控制反转容器模块详细设计

5.1.1控制反转容器的功能设计

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。

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。

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。

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。

5.1.2控制反转容器类的继承设计

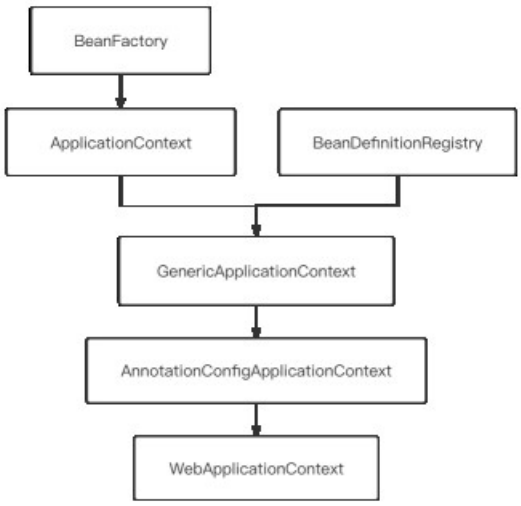

图5-1 控制反转容器继承设计图

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。

5.1.3控制反转容器的注解配置设计

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。

5.2处理器详细设计

5.2.1处理器注解定义详细设计

处理器是整个MVC框架最重要的部分之一，处理器也可以理解为在MVC模式中的控制层，所以对于处理器的设计十分重要，也决定了整个MVC框架的好坏。

控制层的模式其实是面向过程的，需要一个语句块即可实现，所以处理器的实现只需要一个类的一个方法即可，所以处理器其实是对一个方法的映射，每一个被标为控制者的了的一个方法都是一个处理器。

处理器是框架使用者需要使用提供的注解API去定义的处理链中的模块，处理器的定义的粒度是方法级的，也就是说一个类可以定义多个处理器，使用控制者注解可以将某个类标记为一个控制类，如果一个类被标注为一个控制类，则其中的方法才能被声明为处理器方法。

在控制类中使用请求映射注解可以标记某一个类为处理器方法，而处理器方法即可处理一个http请求。在使用请求映射注解的时候可以指定映射的URI和URI对应的请求方法，请求方法也可以指定多个，如果不指定请求方法，那么默认的请求方法是GET方法。

5.2.2处理器适配器详细设计

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。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，用于根据请求生成参数以便调用处理器。

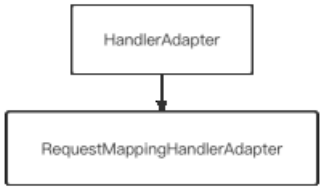

图5-2 处理器适配器继承设计图

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

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。

参数解决器有一个接口和三个实现类。

一个接口为HandlerMethodArgumentResolver，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。

NormalMethodArgumentResolver是一个基于HandlerMethodArgument Resolver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。

RequestParamMethodArgumentResolver是一个基于HandlerMethod ArgumentResolver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。

PathVariableMethodArgumentResolver是一个基于HandlerMethodArgument Resolver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。

5.3处理器拦截器详细设计

处理器拦截器被设计的原因是因为单纯的一个方法作为处理器的流程无法解决合并相同代码的目的。当多个处理器具有相同的代码需求和代码实现时，代码编写者必须在不同的处理器也就是控制者的方法中编写相同的代码。以这种形式极度不利于代码的维护，良好的设计思想是将相同的代码合并到一处，提高代码的可维护性。

因为这个原因，设计了处理器拦截器，命名为处理器拦截器是从框架的流程于处理器拦截器本身的流程行为命名的，实际在业务中处理器拦截器所扮演的角色仅仅是合并相同的代码或进行不同业务流程的解耦。

处理器拦截器被设计为可以使用多个，目的也是为了解耦，不同的业务流程可以封装在一个处理器拦截器中，也可以多个处理器拦截器共同组成同一个处理器流程。

处理器拦截器被成功定义后，还需要使用一个接口去将定义好了处理器拦截器注册到框架中去，使其生效。设计的拦截器配置器具有一个方法，方法的参数是处理器拦截器的注册中心，使用注册中心提供的方法即可注册处理器拦截器。当处理器拦截被配置后，会自动在框架中生效。

5.4处理器映射器详细设计

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。

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。

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如图5.4所示：

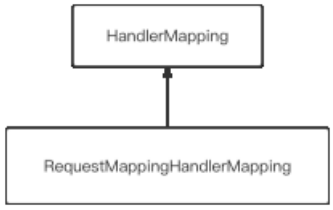

图5-3 处理器映射器继承图

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

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。

5.5视图解决器的详细设计

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。

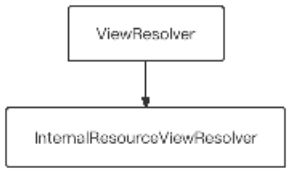

图5-4 视图解决器

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。

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。

指　　标
疑似剽窃文字表述

1. 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。
   同时，由于在处理器适配器执行处理器的过程中，
2. 处理器映射器继承图
   HandlerMapping 是一个接口，定义与DispatcherServlet交互的

7. 基于MVC模式的框架研究与实现_第7部分　　总字数：13498

相似文献列表

去除本人文献复制比：1.5%(207)　　文字复制比：1.5%(207)　　疑似剽窃观点：(0)

1	基于数据库技术的在线商城网站设计与实现 崔申申 - 《大学生论文联合比对库》- 2021-02-22	1.5%（202） 是否引证：否
2	罗定中学师生交流管理系统的设计与实现 李海萍(导师：宁宁;李呐) - 《电子科技大学硕士论文》- 2015-09-01	0.5%（72） 是否引证：否
3	10081406-林佳丽-智能手机被盗防泄密系统设计与实现 林佳丽 - 《大学生论文联合比对库》- 2014-06-08	0.5%（66） 是否引证：否
4	20190197259_李梦颖_基于Android的在线租房交易社区的设计与实现 李梦颖 - 《大学生论文联合比对库》- 2019-06-02	0.5%（66） 是否引证：否
5	1501657542_杨文轩_电子商城系统的设计与实现 王碧莹 - 《大学生论文联合比对库》- 2019-05-14	0.5%（66） 是否引证：否
6	吴忠扬_软件杰普实验班_5720171521_基于JAVA的小游戏开发 软件杰普实验班 - 《大学生论文联合比对库》- 2021-05-27	0.5%（66） 是否引证：否
7	宠物店管理系统的设计与实现 王海帅 - 《大学生论文联合比对库》- 2021-05-13	0.5%（66） 是否引证：否
8	AES、RSA相结合的数据加密方案在Java中的实现 赵航涛; - 《电脑知识与技术》- 2008-05-23	0.5%（63） 是否引证：否

9	租房管理系统的设计与开发 胡艳 - 《大学生论文联合比对库》- 2020-06-02	0.5%（63） 是否引证：否
10	基于java的昌吉学院图书管理系统 阿尔孜古丽·克热木 - 《大学生论文联合比对库》- 2020-06-03	0.5%（63） 是否引证：否
11	9046185_格桑次旺_基于安卓的手机点单系统_10.正文(1) 格桑次旺 - 《大学生论文联合比对库》- 2020-05-30	0.5%（63） 是否引证：否
12	胡艳_租房管理系统的设计与开发 胡艳 - 《大学生论文联合比对库》- 2020-06-01	0.5%（63） 是否引证：否
13	基于人脸识别身份认证的开发与实现 闻泽 - 《大学生论文联合比对库》- 2021-04-06	0.5%（63） 是否引证：否
14	基于人脸识别身份认证的开发与实现 闻泽 - 《大学生论文联合比对库》- 2021-04-06	0.5%（63） 是否引证：否
15	基于人脸识别身份认证的开发与实现 闻泽 - 《大学生论文联合比对库》- 2021-04-13	0.5%（63） 是否引证：否
16	120160386_蔡念桦_基于SSM的学生管理系统 蔡念桦 - 《大学生论文联合比对库》- 2021-05-04	0.5%（63） 是否引证：否
17	sit0415 旦增晋美 - 《大学生论文联合比对库》- 2021-05-14	0.5%（63） 是否引证：否
18	17160222史正义基于微信小程序的学生活动管理系统的设计与实现 史正义 - 《大学生论文联合比对库》- 2021-04-28	0.5%（63） 是否引证：否
19	17010123+刘奇峰+微信小程序-家长访校管理系统 刘奇峰 - 《大学生论文联合比对库》- 2021-05-02	0.5%（63） 是否引证：否
20	基于ssm的学生管理系统论文1 司林玲 - 《大学生论文联合比对库》- 2021-05-04	0.5%（63） 是否引证：否
21	201604011052_解伟光_大用户负荷预测系统设计与开发 解伟光 - 《大学生论文联合比对库》- 2016-04-01	0.4%（60） 是否引证：否

原文内容

第6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bean定义实现

控制反转容器是整个MVC框架的基础支持。所以必须有一个设计良好的控制反转容器。在控制反转容器类的继承设计中，设计了较为复杂的继承关系。因为控制反转容器作为一个基础服务的组件，大量的其他组件对控制反转容器有比较多的依赖关系，所以为了提高控制反转容器的扩展性以及代码的可维护性，设计了较为复杂的继承关系用于提供多样的实现。

在这些实现类中最为关键的实现类是GenericApplicationContext，这个实现类就有控制反转最基本的功能，也是控制反转容器最重要的代码实现。

控制反转容器的设计实现是工厂模式，所以必须有一个字段存贮所有bean的定义，并使用一个id标记不同的bean，而为了实现单例模式的bean，控制反转容器中还有一个单例池存储单例，同时，它也作为一个缓存使用，为了解决循环依赖，还应有一个创建中的缓存池。

控制反转容器bean定义实现关键代码如代码6-1所示：

代码6-1 控制反转容器主要实现类

```
public class GenericApplicationContext
implements ApplicationContext, BeanDefinitionRegistry {
protected BeanDefinitionGenerator bdg = new BeanDefinitionGenerator();
protected Map<String, BeanDefinition> beanDefinitionMap = null;
protected Map<String, Object> singletonObjects = new HashMap<>();
protected Map<String, Object> earlySingletonObjects = new HashMap<>();
protected Map<String, Object> earlyPrototypeObjects = new HashMap<>();
}
```

控制反转容器的一个最核心的需要解决的问题就是，不同bean之间的循环依赖，如果不解决循环依赖问题，就可能会造成程序的死循环。

控制反转容器实现解决循环依赖关键代码如代码6-2所示：

代码6-2 解决循环依赖关键代码

```
a = bd.getBeanClass().getDeclaredConstructor().newInstance();
this.earlySingletonObjects.put(beanId, a);
for (Map.Entry<Field,Object> entry : bd.getFieldMap().entrySet()) {
Field field = entry.getKey();
field.setAccessible(true);
String fieldBeanId = ((BeanId) entry.getValue()).getValue();
field.set( a, this.getBean(fieldBeanId) );
}
this.earlySingletonObjects.remove(beanId);
this.singletonObjects.put(beanId, a);
return processingSingletonObject;
```

解决了循环依赖的问题，还有一个问题需要解决，那就是多线程并发的问题，在多线程的代码环境下，可能会同时有多个线程去向控制反转容器请求bean和进行一些其他操作。并且控制反转容器极易使用在多线程的环境下，所以支持并发的线程安全也是控制反转容器需要支持的。

一般解决并发问题都会使用锁来进行同步，使用了锁后，同时只会有一个线程拥有锁，也就是在同一时间中，只会有一个线程与控制反转容器进行交互。此框架也使用这种策略去解决多线程并发数据一致性的问题。

控制反转容器并发兼容实现关键代码如代码6-3所示：

代码6-3 并发数据一致性关键代码

```
public synchronized <T> T getBean(Class<T> requiredType) {
for (BeanDefinition beanDefinition : this.beanDefinitionMap.values()) {
boolean isMatched = requiredType.isAssignableFrom(beanDefinition.getBeanClass());
if (isMatched) return requiredType.cast(this.getBean(beanDefinition.getId()));
}
return null;
}
```

6.1.2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。

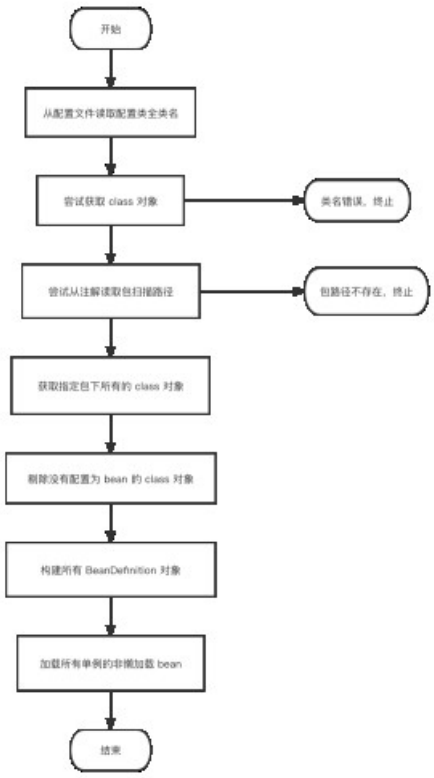

图6-4 控制反转容器初始化流程图

在控制反转容器容器的启动流程中会有许多需要解决的问题，其中最重要的一个问题就是，因为控制反转容器是通过系统提供的文件支持服务去扫描获取类的反射对象实现扫描功能的，但是与文件系统交互就意味着需要考虑不同系统的兼容性问题，需要着重考虑代码的兼容性写法，Java提供了兼容不同系统的不同文件支持的代码写法。

其中，主要需要使用的写法是文件系统文件分割符的问题，只要使用Java提供的针对与不同系统自动变化的文件静态常量

系统分隔符，既可以解决不同系统的问题。

解决文件兼容实现的关键代码如代码6-5所示：

代码6-5 不同系统文件兼容

```
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> list = ToolMethods.getPathsUnderDirectory(file);
if (processingClassNameList == null) return null;
```

6.1.3容器管理对象Bean创建实现

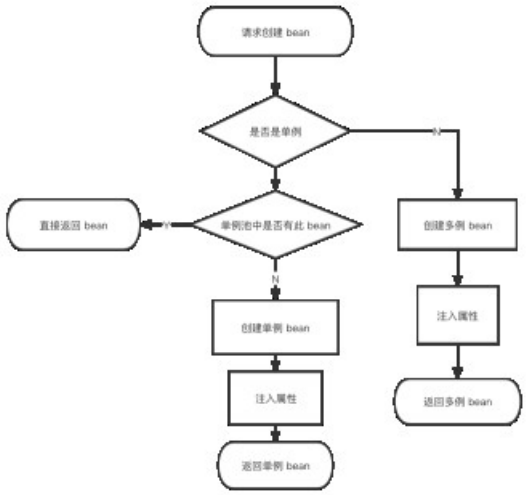

图6-6 Bean创建流程图

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。

代码6-7 Bean创建关键代码

```
String beanId = bd.getId();
Object b;
b = this.singletonObjects.getOrDefault(beanId, null);
if (b != null) return b;
b = this.earlySingletonObjects.getOrDefault(beanId, null);
if (b != null) return b;
b = bd.getBeanClass().getDeclaredConstructor().newInstance();
this.earlySingletonObjects.put(beanId, b);
for (Map.Entry<Field,Object> entry : bd.getFieldMap().entrySet()) {
Field field = entry.getKey();
field.setAccessible(true);
String fieldBeanId = ((BeanId) entry.getValue()).getValue();
field.set( b, this.getBean(fieldBeanId) );
}
this.earlySingletonObjects.remove(beanId);
this.singletonObjects.put(beanId, b);
return processingSingletonObject;
```

6.2处理器映射器模块实现

处理器映射器是起到根据请求中的URI和对应的请求方法，返回对应的处理器和拦截器，并将处理器和拦截器封装为处理器执行链对象。

所以在处理器映射器工作之前，必须将所有的处理器和拦截器注册到处理器映射器中去，这是一个比较复杂的过程，要实现这一过程也需要花费一定的心思，良好的设计这一过程，才能使得处理器映射器正确工作。

关于如何初始化处理器部分，可以通过从控制反转容器中获取在类上绑定有控制者注解的类，获取后读取请求映射的注解的方法，将其封装为处理器方法对象，注册到处理器映射器中。

关于如何初始化拦截器部分，这拦截器模块设计中，我们已经设计了一个处理器拦截器注册中心，所以相比于处理器的初始化，处理器拦截器的初始化比较简单，只需要将处理器拦截器注册中心交由处理器映射器控制即可。

处理器映射器初始化关键代码如代码6.8所示：

代码6-8 处理器映射器初始化关键代码

```
List<BeanDefinition> bds = context.getControllers();
if (bds == null)
return;
for (BeanDefinition bd : bds)
for ( Method method : bd.getBeanClass().getDeclaredMethods() ) {
if ( !method.isAnnotationPresent(RequestMapping.class) )
continue;
String URI = "";
if ( bd.getBeanClass().isAnnotationPresent(RequestMapping.class) )
URI = bd.getBeanClass().getAnnotation(RequestMapping.class).value();
URI += method.getAnnotation(RequestMapping.class).value();
Object controller = context.getBean( bd.getId() );
HandlerMethod handlerMethod = new HandlerMethod();
handlerMethod.setRequestMapping(URI);
handlerMethod.setController(controller);
handlerMethod.setMethod(method);
handlerMapping.registerHandlerMethod(handlerMethod.getRegexURI(), handlerMethod);
}
}
```

处理器映射器初始化后，便可以作为一个成熟的组件良好的运行。

处理器映射器作为一个组件，对外或者更准确的说，对DispatcherServlet提供的接口，是获取处理器执行链，其中包含处理器和符合条件的处理器拦截器。DispatcherServlet获取到处理器执行链后，处理器映射器提供的一次服务即完成。

处理器映射器实现接口的关键代码如代码6-9所示：

代码6-9 处理器映射器实现接口关键代码

```
protected boolean addHandler(HandlerExecutionChain chain, HttpServletRequest request) {
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;
}
}
}
```

6.3请求参数自动注入模块实现

每一个请求编写代码的过程中都具有请求参数的获取和参数类型的转换，因为http请求默认的参数类型都是字符串类型的。

在编写这一过程的代码中出现了大量的重复代码，从代码的复用性角度分析，完全可以将这一个重复的代码封装为一个可以重复利用的模块使用。

但是，在各种各样的参数类型转换中，有许多种类的类型转换方式，只使用单个的方式无法办到应对各种各样的情况，所以请求参数自动注入模块设计为一个接口，以及多种实现类。

其中，最基本的实现类是NormalMethodArgumentResolver，他可以将请求和响应，以及需要返回的ModelAndView自动在调用是注入进处理器的参数。

代码6-10 普通参数解决器关键代码

```
public Object resolveArgument(
HttpServletRequest request,
HttpServletResponse response,
HandlerMethod handlerMethod,
MethodParameter parameter) {
if ( !this.supportsParameter(parameter) )
return null;
Class<?> parameterType = parameter.getParameterType();
if (parameterType == HttpServletRequest.class)
return (Object) request;
else if (parameterType == HttpServletResponse.class)
return (Object) response;
else if (parameterType == ModelAndView.class)
return (Object) new ModelAndView();
else
return null;
}
```

普通参数处理器只解决了实现ServletAPI的请求和响应参数的注入，以及ModelAndView返回对象的自动创建注入。

很多时候，我们需要从路径中获取参数，此时，路径不仅仅作为请求处理器的映射判断条件，还作为请求参数参与业务的逻辑中去，所以此框架默认也提供了这种类型的参数自动注入处理器。

代码6-11 路径参数关键代码

```
public Object resolveArgument(
HttpServletRequest request,
HttpServletResponse response,
HandlerMethod handlerMethod,
MethodParameter parameter) {
if ( !this.supportsParameter(parameter) )
return null;
String argumentName = parameter.getParameterAnnotation(PathVariable.class).value();
if ( argumentName.equals("") )
return null;
String regex = handlerMethod.getRegexURI();
Matcher matcher = Pattern.compile(regex).matcher(request.getRequestURI());
String[] argumentNames = handlerMethod.getArgumentNames();
List<String> argumentValues = new ArrayList<String>();
if ( matcher.find() )
for (int i = 1; i <= matcher.groupCount(); i++)
argumentValues.add(matcher.group(i));
for (int i = 0; i < argumentNames.length; i++)
if (argumentNames[i].equals(argumentName))
return (Object) argumentValues.get(i);
return null;
}
```

除了以上的请求参数类型转换外，还有一个最基本的参数类型转换，那就是请求参数的类型转换，这个参数类型转换器将字符串类型的请求参数转化为在处理器中对应类型的参数，支持的转换类型有字符串，整数，浮点数，基本整数，这4中类型。

代码6-12 路径参数解决器关键代码

```
public Object resolveArgument(
HttpServletRequest request,
HttpServletResponse response,
HandlerMethod handlerMethod,
MethodParameter parameter) {
if ( !this.supportsParameter(parameter) )
return null;
String argumentName = parameter.getParameterAnnotation(RequestParam.class).value();
if ( argumentName.equals("") )
return null;
String argumentValue = request.getParameter(argumentName);
Class<?> parameterType = parameter.getParameterType();
```

```
if (parameterType != String.class &&
parameterType != Integer.class &&
parameterType != Float.class &&
parameterType != int.class)
return null;
}
```

实现了上述4中类型的参数解决器，还要实现，在处理器适配器中调用这些参数类型处理器。

在处理器适配器中调用这些参数解决器采用的设计方法是策略模式。采用策略模式，使得框架在这里多了一个扩展接口，框架使用者或者框架维护者可以使用这个扩展接口自定义一些参数解决器，从而增强框架的功能。

代码6-13 请求参数解决器关键代码

```
public interface HandlerInterceptor {
default boolean preHandle(
HttpServletRequest request,
HttpServletResponse response,
Object handler ) {
return true;
}
default void postHandle(
HttpServletRequest request,
HttpServletResponse response,
Object handler,
ModelAndView modelAndView
) {
}
}
```

6.4处理器拦截器代码实现

处理器拦截器在设计时适合处理器高度耦合的，所以处理器拦截器的实现也和处理器高度耦合。

处理器拦截器是由用户定义的模块，所以在这种模块的实现当中，需要和框架使用者进行交互。

为此，需要定义一个接口，提供给框架使用者使用，用来定义处理器拦截器。在接口的规范当中，需要实现两个方法，一个是前置拦截，一个是后置拦截，提供多样的拦截方法是为了更好的提高框架的多样性。

代码6-14 处理器拦截器关键代码

```
protected Object invokeHandlerMethod(
HttpServletRequest request,
HttpServletResponse response,
HandlerMethod handlerMethod ) {
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 a : this.argumentResolverList) {
if ( a.supportsParameter(methodParameter) ) {
arguments[i] = a.resolveArgument(request, response, handlerMethod);
break;
}
}
}
Object returnObject = null;
Object controller = handlerMethod.getController();
return method.invoke(controller, arguments);
}
```

处理器拦截器的定义已经实现了，但是处理器拦截器的使用还有两个必须要实现的要素，一个是处理器拦截器拦截URI的定义，必须要有这个定义，处理器拦截器才能够正确的生效。第二个是处理器拦截器如果在框架中声明的问题，处理器拦截器必须通过一些接口融入到框架中去。

首先，解决实现第二个问题，我们需要一个处理器拦截器的注册中心，去注册拦截器。

这个拦截器注册中心，主要需要实现两个功能，一个是针对拦截器注册的功能，用于在拦截器注册接口中将想要注册的处理器拦截器注册到拦截器注册中心中。第二个功能是，处理器拦截器注册中心需要向外提供一个获取处理器拦截器的接口，实现了这个接口，框架才能正确融合处理器拦截器注册中心。

处理器器拦截器注册中心注册处理器拦截器关键代码实现如代码6-15所示：

代码6-15 处理器拦截器注册关键代码

```
public InterceptorRegistration addInterceptor(HandlerInterceptor interceptor) {
```

```
InterceptorRegistration registration = new InterceptorRegistration(interceptor);
this.registrations.add(registration);
return registration;
}
```

处理器拦截器注册中心对外提供获取拦截器的关键代码如代码6-16所示：

代码6-16 处理器拦截器注册中心对外提供服务关键代码

```
public List<HandlerInterceptor> getInterceptors(String URI) {
List<HandlerInterceptor> a = new ArrayList<HandlerInterceptor>();
for (InterceptorRegistration registration : this.registrations) {
if (registration.matches(URI)) {
handlerInterceptors.add(registration.getHandlerInterceptor());
}
}
return handlerInterceptors;
}
```

处理器拦截器的定义，注册。拦截URI指定都已经实现了。处理器拦截器模块的基本功能就已经成型了。

接下来还要实现DispatcherServlet向处理器注册中心获取对应的处理器拦截器，以及在调用处理器的同时，调用所以对应的处理器拦截器。

要实现调用处理器拦截器，必须既调用处理器拦截器的前置拦截，又调用处理器拦截器的后置拦截，并且处理器拦截器的设计为可以多个处理器拦截器同时拦截同一个处理器的请求，那么处理器拦截器的前置拦截和后置拦截就必须考虑调用的顺序。

处理器拦截器调用的关键代码实现如代码6-17所示：

代码6-17 处理器拦截器调用代码

```
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 mav = this.handlerAdapter.handle(
request, response handlerExecutionChain.getHandler() );
handlerExecutionChain.applyPostHandle(request, response, modelAndView);
```

6.5处理器适配器模块实现

处理器适配器的实现分为两个部分，因为处理器适配器的初始化配置较为复杂，并且需要同控制反转容器交互实现，所以需要良好的实现。

处理器适配器初始化代码实现如代码6-18所示：

代码6-18 处理器适配器初始化关键代码

```
protected void initHandlerAdapter(WebApplicationContext context) {
this.handlerAdapter = (RequestMappingHandlerAdapter)
context.getBean("reque stMappingHandlerAdapter");
PathVariableMethodArgumentResolver a = (PathVariableMethodArgumentResol ver)
context.getBean("pathVariableMethodArgumentResolver");
NormalMethodArgumentResolver b = (NormalMethodArgumentResolver) conte
xt.getBean("normalMethodArgumentResolver");
RequestParamMethodArgumentResolver c = (RequestParamMethodArgumentR esolver)
context.getBean("requestParamMethodArgumentResolver");
this.handlerAdapter.registerArgumentResolver(a);
this.handlerAdapter.registerArgumentResolver(b);
this.handlerAdapter.registerArgumentResolver(c);
}
```

处理器适配器的初始化代码已经实现了。那么在使用处理器适配器去调用处理器也需要实现。在这个实现中，前面参数解决器的章节已经说明了处理器适配器调用参数解决器的代码，此处主要是处理接口参数规范的代码的实现。

处理器适配器执行处理器关键代码如代码6-19所示：

代码6-19 处理器适配器调用关键代码

```
public ModelAndView handle(
HttpServletRequest request,
HttpServletResponse response,
Object handler) {
Object controllerResult = this.invokeHandlerMethod(
request ,response, (HandlerMethod) handler );
if (controllerResult == null)
return new ModelAndView();
else if (controllerResult.getClass() == ModelAndView.class)
return (ModelAndView) controllerResult;
else
return new ModelAndView();
}
```

6.6视图解决器模块实现

视图解决模块的关键实现有两个关键的需要实现的部分。

一个部分是视图解决器返回的视图对象将模型中的数据渲染进页面的关键代码实现，这一部分主要是将模型中的数据获取出来，与Servlet接口交互，将数据存储进请求的属性中。

另一部分是视图解决器本身针对JSP视图支持的对应的视图解决器实现类，这也是框架默认使用的视图解决器，主要是内部资源视图对象的实现。

8. 基于MVC模式的框架研究与实现_第8部分　　总字数：887

相似文献列表

去除本人文献复制比：28.7%(255)　　文字复制比：28.7%(255)　　疑似剽窃观点：(0)

1	一种轻量级的JavaWeb框架的设计与实现 何易晟 - 《大学生论文联合比对库》- 2018-05-23	28.7%（255） 是否引证：否

原文内容

第一部分视图解决器的视图对象渲染页面关键代码实现如代码6-20所示：

代码6-20 视图对象渲染关键代码

```
public void render(
HttpServletRequest request,
HttpServletResponse response,
Map<String,Object> model) {
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();
}
}
```

内部资源视图解决器的作用主要也就是穿件内部资源视图对象，其中比较重要的实现是，有视图对象名的前缀和后缀的实现，在处理器中，可能我们需要在每个处理器中都指定视图名，而我们习惯于将内部资源视图对象对象的文件放在某一个路径下，这样实现减少了框架使用者需要编写的代码量。

内部资源视图解决器关键代码实现如代码6-21所示：

代码6-21 视图对象渲染关键代码

```
public class InternalResourceViewResolver implements ViewResolver {
private String prefix = "";
private String suffix = "";
public View resolveViewName(String viewName) {
viewName = this.prefix + viewName + this.suffix;
return new InternalResourceView(viewName);
}
```

原文内容

第7章系统测试

7.1 测试方法

7.1.1白盒测试

白盒测试是一种常见的测试方法，通过检查程序的逻辑去判断程序是否正确。在知晓代码的逻辑的情况下去执行代码。白盒指的是盒子是透明的，也就是测试者明确的知道源代码以及其逻辑，在这种情况下去测试代码去针对程序每种可能的运行逻辑去测试代码。

7.1.2黑盒测试

黑盒测试也是一种常见的测试方法。黑盒的意思是完全不可见的盒子，也就是测试者不知道程序的源代码，就像一个黑盒子完全不知道里面有什么，只是针对程序对外暴露的接口进行测试。黑盒测试是一种只针对接口的测试，所以这种测试方法也不可能完全良好的测试每一程序。所以即使程序通过了黑盒测试检测，也会有许多未知的错误不能被发现。

7.2 功能测试

7.2.1控制反转容器功能测试

测试控制反转容器模块的初始化功能是否正常，在各种情况下，是否能够正确初始化，控制反转容器的初始化测试如表7-1所示：

表7-1 控制反转容器初始化功能测试

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	2022-01-15	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。
9	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。		通过，和预期结果一致。
10	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。		通过，和预期结果一致。
11	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。		通过，和预期结果一致。
12	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。		通过，和预期结果一致。
13	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。		通过，和预期结果一致。
14	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。		通过，和预期结果一致。
15	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。		通过，和预期结果一致。
16	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。		通过，和预期结果一致。
17	使用Autowired注解。		引用被正确的被注入。		通过，和预期结果一致。
18	在单例中使用Autowired注解，指定一个单利bean。		单例的引用被正确的注入。		通过，和预期结果一致。
19	在单例中使用Autowired注解，指定一个原型bean。		原型bean无法被注入，抛出异常。		通过，和预期结果一致。
20	在原型类中使用Autowired注解，指定一个单例bean。		单例的引用被正确的注入。		通过，和预期结

			果一致。
21	在原型类中使用Autowired注解，指定一个原型bean。	原型bean的应用被正确的注入。	通过，和预期结果一致。
22	在单例类中使用Autowired注解，依赖自身。	循环依赖正确被引用。	通过，和预期结果一致。
23	在原型类中使用Autowired注解，依赖自身。	原型类无法依赖自身，排除异常。	通过，和预期结果一致。
24	在单例依赖树中，使用多个类的循环依赖。	多个单例类的循环依赖成功，循环依赖被正确实现。	通过，和预期结果一致。
25	在原型类依赖树中，循环依赖原型类。	原型类循环依赖链无法实现，抛出异常。	通过，和预期结果一致。
26	创建多个线程从控制反转容器中获取bean，查看bean获取是否异常。	bean并发获取不会出现错误，	通过，和预期结果一致。

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 2022-01-15 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。

9 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。 通过，和预期结果一致。

10 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。 通过，和预期结果一致。

11 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。 通过，和预期结果一致。

12 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。 通过，和预期结果一致。

13 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。 通过，和预期结果一致。

14 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。 通过，和预期结果一致。

15 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。 通过，和预期结果一致。

16 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。 通过，和预期结果一致。

17 使用Autowired注解。 引用被正确的被注入。 通过，和预期结果一致。

18 在单例中使用Autowired注解，指定一个单利bean。 单例的引用被正确的注入。 通过，和预期结果一致。

19 在单例中使用Autowired注解，指定一个原型bean。 原型bean无法被注入，抛出异常。 通过，和预期结果一致。

20 在原型类中使用Autowired注解，指定一个单例bean。 单例的引用被正确的注入。 通过，和预期结果一致。

21 在原型类中使用Autowired注解，指定一个原型bean。 原型bean的应用被正确的注入。 通过，和预期结果一致。

22 在单例类中使用Autowired注解，依赖自身。 循环依赖正确被引用。 通过，和预期结果一致。

23 在原型类中使用Autowired注解，依赖自身。 原型类无法依赖自身，排除异常。 通过，和预期结果一致。

24 在单例依赖树中，使用多个类的循环依赖。 多个单例类的循环依赖成功，循环依赖被正确实现。 通过，和预期结果一致。

25 在原型类依赖树中，循环依赖原型类。 原型类循环依赖链无法实现，抛出异常。 通过，和预期结果一致。

26 创建多个线程从控制反转容器中获取bean，查看bean获取是否异常。 bean并发获取不会出现错误， 通过，和预期结果一致。

7.2.2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7-2所示：

表7-2 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	2022-01-15	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。
10	不进行任何配置，查看处理器映射器能否被正确初始化。	处理器映射器被正确初始化	通过，和预期结果一致。
11	配置一个处理器，查看处理器映射器能否被正确初始化。	处理器映射器被正确初始化。	通过，和预期结果一致。
12	配置多个处理器，查看处理器映射器能否被正确初始化	处理器映射器被正确初始化。	通过，和预期结果一致。
13	配置一个拦截器，查看拦截器注册中心是否被正确注册拦截器。	处理器拦截器被正确注册到拦截中心中。	通过，和预期结果一致。
14	配置多个拦截器，查看处理器拦截器注册中心是否被正确注册拦截器。	处理器拦截器被正确注册到拦截中心中。	通过，和预期结果一致。
15	不进行任何配置，查看视图解决器是否被正确初始化。	视图解决器被正确初始化。	通过，和预期结果一致。

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 2022-01-15 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。
10 不进行任何配置，查看处理器映射器能否被正确初始化。 处理器映射器被正确初始化通过，和预期结果一致。
11 配置一个处理器，查看处理器映射器能否被正确初始化。 处理器映射器被正确初始化。 通过，和预期结果一致。
12 配置多个处理器，查看处理器映射器能否被正确初始化处理器映射器被正确初始化。 通过，和预期结果一致。
13 配置一个拦截器，查看拦截器注册中心是否被正确注册拦截器。 处理器拦截器被正确注册到拦截中心中。 通过，和预期结果一致。
14 配置多个拦截器，查看处理器拦截器注册中心是否被正确注册拦截器。 处理器拦截器被正确注册到拦截中心中。 通过，和预期结果一致。
15 不进行任何配置，查看视图解决器是否被正确初始化。 视图解决器被正确初始化。 通过，和预期结果一致。
7.2.3处理器模块功能测试
表7-3 处理器模块功能测试

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	2022-01-15	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。
10	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。	通过，和预期结果一致。
11	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。	通过，和预期结果一致。
12	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。	通过，和预期结果一致。
13	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。	通过，和预期结果一致。
14	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。	通过，和预期结果一致。
15	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。	通过，和预期结果一致。
16	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。	通过，和预期结果一致。

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 2022-01-15 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。

10 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。 通过，和预期结果一致。

11 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。 通过，和预期结果一致。

12 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。 通过，和预期结果一致。

13 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。 通过，和预期结果一致。

14 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。 通过，和预期结果一致。

15 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。 通过，和预期结果一致。

16 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。 通过，和预期结果一致。

7.2.4处理器映射器模块测试

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7-4所示：

表7-4 处理器映射器模块功能测试

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	2022-01-15	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。
9	配置处理器和多个拦截器，在Servlet中输出处理器执行链，判断多个处理器拦截器是否正确封装。	正确输出处理器执行链。	通过，和预期结果一致。
10	配置处理器和一个处理器拦截器，判断处理器拦截器是否被正确执行。	一个处理器拦截器被正确执行。	通过，和预期结果一致。
11	配置处理器和多个处理器拦截器，判断多个处理器拦截器是否被正确执行。	多个处理器拦截器被正确执行。	通过，和预期结果一致。
12	配置处理器并且使用路径参数解决器，输出路径URI，判断是否出书正确的正则表达式。	正确输出正则表达式。	通过，和预期结果一致。

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 2022-01-15 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。

9 配置处理器和多个拦截器，在Servlet中输出处理器执行链，判断多个处理器拦截器是否正确封装。 正确输出处理器执行链。 通过，和预期结果一致。

10 配置处理器和一个处理器拦截器，判断处理器拦截器是否被正确执行。 一个处理器拦截器被正确执行。 通过，和预期结果一致。

11 配置处理器和多个处理器拦截器，判断多个处理器拦截器是否被正确执行。 多个处理器拦截器被正确执行。 通过，和预期结果一致。

12 配置处理器并且使用路径参数解决器，输出路径URI，判断是否出书正确的正则表达式。 正确输出正则表达式。 通过，和预期结果一致。

7.2.5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7-5所示：

表7-5 处理器适配器模块功能测试

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	2022-01-15	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	配置一个处理器，请求处理器，查看处理器是否能够被正确执行。		处理器能够被正确执行		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。

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 2022-01-15 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 配置一个处理器，请求处理器，查看处理器是否能够被正确执行。 处理器能够被正确执行通过，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。

7.2.6参数解决器功能测试

参数解决器功能测试如表7-6所示：

表7-6 参数解决器功能测试

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	2022-01-15	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	在处理器中同时使用多个参数，请求，测试处理器参数是否被正确注入。		处理器参数被正确注入。		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。
5	在处理器中使用其他的返回值类型，		抛出异常，不能使用此返回值。		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@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。

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 2022-01-15 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 在处理器中同时使用多个参数，请求，测试处理器参数是否被正确注入。 处理器参数被正确注入。 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。

5 在处理器中使用其他的返回值类型， 抛出异常，不能使用此返回值。 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@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。

7.2.7视图解决器模块功能测试

参数解决器功能测试如表7-7所示：

表7-7 视图解决器功能测试

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	2022-01-15	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。

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 2022-01-15 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。

5	攀枝花市电信计费系统设计与实现 郝华来(导师：饶建珍;刘忠) - 《电子科技大学硕士论文》- 2013-04-29	1.2%（69） 是否引证：否
6	永州市乡村旅游发展研究 邓险夷 - 《大学生论文联合比对库》- 2018-05-27	1.2%（67） 是否引证：否
7	成都市农家乐消费研究 余尧尧(导师：赵吉林) - 《西南财经大学硕士论文》- 2011-04-01	1.0%（58） 是否引证：否
8	基于IPTV及安卓平板电脑的多屏互动网络学习系统设计与实现 林文峰(导师：左保河;袁海洲) - 《华南理工大学硕士论文》- 2013-05-01	1.0%（53） 是否引证：否
9	基础设施对海岛型乡村旅游地开发的影响与对策 ——以“大陈岛”为例 陈雪洁 - 《大学生论文联合比对库》- 2017-12-20	0.7%（41） 是否引证：否

原文内容

第8章结束语

8.1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。

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

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.

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。

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。

8.2不足与下一步工作

不系统具有MVC框架的基本功能，也有一些提高开发者效率的功能。同时由于缺乏经验，整个MVC框架的耦合性还是没有到一个比较低的比较满意的水平，这也在可以预测的范围之内。

针对于此系统最大的不足，下一步的工作为，学习设计模式以及去看一些成熟开源项目的代码，学习如何才能够在复杂系统中编写扩展性好，可维护性高的代码，并针对此系统中不合理的代码进行重构与优化。

参考文献

[1]曾悠,杨明. MVC模式在数据可视化组件中的研究与应用[J]. 软件导刊,2021,20(08):155-159.
[2]陈恒,楼偶俊,巩庆志,张立杰. Spring MVC开发技术指南[J]. 计算机教育,2021,(07):194.
[3]娄元润. 基于MVC的芯片可视化配置系统的设计与实现[D].西安电子科技大学,2021.
[4]陈诚. 基于MVC的某企业员工健康服务管理系统的设计与实现[D].江西师范大学,2021.
[5]刘璐. 基于MVC模式电力设备安全巡检信息管理系统的设计与实现[D].电子科技大学,2021.
[6]杨晨晓. 基于MVC架构的公文流转系统的设计与实现[D].内蒙古大学,2020.
[7]袁琳. 基于MVC的锡盟气象局OA系统研究与实现[D].内蒙古农业大学,2020.
[8]于冰. 基于MVC的治安综合管理信息系统的设计与实现[D].北京交通大学,2020.
[9]陈帅. 基于MVC的NOSS+报表管理系统的设计与实现[D].吉林大学,2020.
[10]邱前绮. 基于MVC技术架构的中职生安全教育在线学习系统[D].广东工业大学,2020.
[11]杜雯. 基于MVC模式的ATS仿真系统的研究与实现[D].西南交通大学,2020.
[12]何煜. 基于MVC架构的生鲜电子商务交易系统的设计与实现[D].大连交通大学,2019.
[13]张煜. 基于MVC架构模式的服务开通系统设计与实现[D].南京邮电大学,2018.
[14]程洁. 基于MVC的政府审计系统的设计与实现[D].南京邮电大学,2018.
[15]宋超. MVC850B型数控铣床摩擦补偿与伺服优化研究[D].南华大学,2019.
[16]杨瑞涛. 基于ASP.NET MVC的CSM管理系统设计与实现[D].内蒙古大学,2019.

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。

外文资料原文

Design and Management of Control System for Rural Tourism Network Information Based on MVC Model

Dong Hanlin

Informatization of Rural Tourism Network

Rural tourism began in the 1830s, and after the 1980s, rural tourism began to develop on a large scale. Now, it has a considerable scale in some western developed countries. In some highly urbanized regions and countries, rural tourism can account for 10%－25% of all tourism activities. The development of rural tourism has effectively changed the phenomenon of rural economic downturn. The contribution of rural tourism to the local economy and the significance of local development have been well proven. In many countries, it is agreed that rural tourism is the driving force of economic development and economic diversification in remote rural areas.

In addition to the support of national policies, rural tourism development also has a profound background. Nowadays, the economic level of urban residents has improved, and their leisure time has increased. Also, urban residents' physical and mental needs to return to nature are more urgent. The open space, fresh air, beautiful

environment, and rich local culture in the rural areas can meet the desire of urban tourists to return to nature and return to the basics. Rural tourism is rapidly formed and developed under such conditions. Today, rural tourism is in the ascendant. After a long development period, rural tourism has developed from the initial spontaneous stage to the present conscious stage. Rural tourism can greatly promote rural economic development. Rural tourism development actively utilizes the agricultural natural environment, agricultural production and management activities, and human resources. After planning and design, the formation of leisure tourism and holiday park with pastoral pleasures can effectively perform agricultural production functions and increase agricultural income.

MVC Model

The MVC pattern is a software architecture pattern. It is a software architecture pattern that separates the three modules of view, controller, and model. The advantage of this design is that system developers and system designers can perform their maintenance. Therefore, it improves the reuse rate of system code and also improves the scalability of system applications. The most significant advantage of the system is that it brings great convenience to system development. MVC is the ideal way to use three different parts to construct a software or component. It provides a powerful object separation mechanism, makes the program more object-oriented, and handles the design of the software architecture and the development of the program. The core idea of the model is to combine effectively “model,” “view,” and “controller”. The model is used to store data objects. The view provides the data display object for the model. The controller is responsible for specific business logic operations and is responsible for matching various operations performed by the “view layer” to the corresponding data of the “data layer” and displaying the results. The structure of the MVC model is shown in Figure 1. The user interacts with the view page, and some requests input by the user are first received by the controller. Then, it is responsible for selecting the corresponding model for processing. The model processes the user’s request through business logic and returns the processed data. Finally, the controller selects the appropriate view to format the returned data. The separation between the three components allows a model to be displayed in multiple different views. When the user changes the data in the model through the controller of a particular view, all other views that depend on the data should reflect these changes. In short, no matter what data changes occur at any time, the controller will notify these changes to all associated views and update the related displays. It is a change propagation mechanism of the model.

译文

基于MVC模式的乡村旅游网络信息控制系统设计与管理

Dong Hanlin

乡村旅游网络信息化

乡村旅游始于19世纪30年代，20世纪80年代后，乡村旅游开始大规模发展。现在，它在一些西方发达国家已经有了相当的规模。在一些高度城市化的地区和国家，乡村旅游占所有旅游活动的10%-25%。乡村旅游的发展有效地改变了农村经济下滑的现象。乡村旅游对当地经济的贡献和当地发展的重要性已得到充分证明。在许多国家，人们一致认为乡村旅游是偏远农村地区经济发展和经济多样化的驱动力。

除了国家政策的支持外，乡村旅游的发展也有着深刻的背景。如今，城市居民的经济水平提高了，闲暇时间增加了。此外，城市居民回归自然的身心需求更为迫切。乡村地区的开放空间、新鲜空气、优美环境和丰富的地方文化可以满足城市游客回归自然、回归本源的愿望。乡村旅游正是在这种条件下迅速形成和发展起来的。如今，乡村旅游方兴未艾。乡村旅游经过长期的发展，已经从最初的自发阶段发展到现在的自觉阶段。乡村旅游可以极大地促进农村经济的发展。乡村旅游开发积极利用农业自然环境、农业生产和管理活动以及人力资源。经过规划设计，形成具有田园乐趣的休闲旅游度假公园，可以有效发挥农业生产功能，增加农业收入。

MVC模式

MVC模式是一种软件架构模式。它是一种软件体系结构模式，将视图、控制器和模型这三个模块分开。这种设计的优点是，系统开发人员和系统设计师可以进行维护。因此，它提高了系统代码的重用率，也提高了系统应用程序的可扩展性。该系统最大的优点是为系统开发带来了极大的便利。MVC是使用三个不同部分构建软件或组件的理想方式。它提供了强大的对象分离机制，使程序更加面向对象，并处理软件体系结构的设计和程序的开发。该模型的核心思想是有效地结合“模型”、“视图”和“控制器”。该模型用于存储数据对象。视图提供了模型的数据显示对象。控制器负责特定的业务逻辑操作，并负责将“视图层”执行的各种操作与“数据层”的相应数据进行匹配，并显示结果。MVC模型的结构如图1所示。用户与视图页面交互，控制器首先接收用户输入的一些请求。然后，它负责选择相应的模型进行处理。该模型通过业务逻辑处理用户的请求，并返回处理后的数据。最后，控制器选择适当的视图来格式化返回的数据。三个组件之间的分离允许在多个不同视图中显示模型。当用户通过特定视图的控制器更改模型中的数据时，依赖于该数据的所有其他视图都应反映这些更改。简而言之，无论任何时候发生什么样的数据更改，控制器都会将这些更改通知所有相关视图，并更新相关显示。这是模型的一种变化传播机制。

指　　标

疑似剽窃观点

1. 简而言之，无论任何时候发生什么样的数据更改，控制器都会将这些更改通知所有相关视图，并更新相关显示。

疑似剽窃文字表述

1. The contribution of rural tourism to the local economy and
2. 乡村旅游方兴未艾。乡村旅游经过长期的发展，已经从最初的自发阶段发展到现在的自觉阶段。
3. 形成具有田园乐趣的休闲旅游度假公园，可以有效发挥农业生产功能，增加农业收入。
4. 模型用于存储数据对象。视图提供了模型的数据显示对象。控制器负责特定的业务逻辑操作，
5. 用户与视图页面交互，控制器首先接收用户输入的一些请求。然后，它负责选择相应的模型进行处理。该模型通过业务逻辑处理用户的请求，并返回处理后的数据。最后，控制器选择适当的视图来格式化返回的数据。三个组件之间的分离允许在多个不同视图中显示模型。当用户通过特定视图的控制器更改模型中的数据时，依赖于该数据的所有其他视图都应反映这些更改。

说明：1.总文字复制比：被检测论文总重合字数在总字数中所占的比例

2.去除引用文献复制比：去除系统识别为引用的文献后，计算出来的重合字数在总字数中所占的比例

3.去除本人文献复制比：去除作者本人文献后，计算出来的重合字数在总字数中所占的比例

4.单篇最大文字复制比：被检测文献与所有相似文献比对后，重合字数占总字数的比例最大的那一篇文献的文字复制比

5.复制比：按照“四舍五入”规则，保留1位小数

6.指标是由系统根据《学术论文不端行为的界定标准》自动生成的

7.红色文字表示文字复制部分;绿色文字表示引用部分;棕灰色文字表示系统依据作者姓名识别的本人其他文献部分

8.本报告单仅对您所选择的比对时间范围、资源范围内的检测结果负责

amlc@cnki.net
https://check.cnki.net/