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。

第 2 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。

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。

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、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。

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。

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：

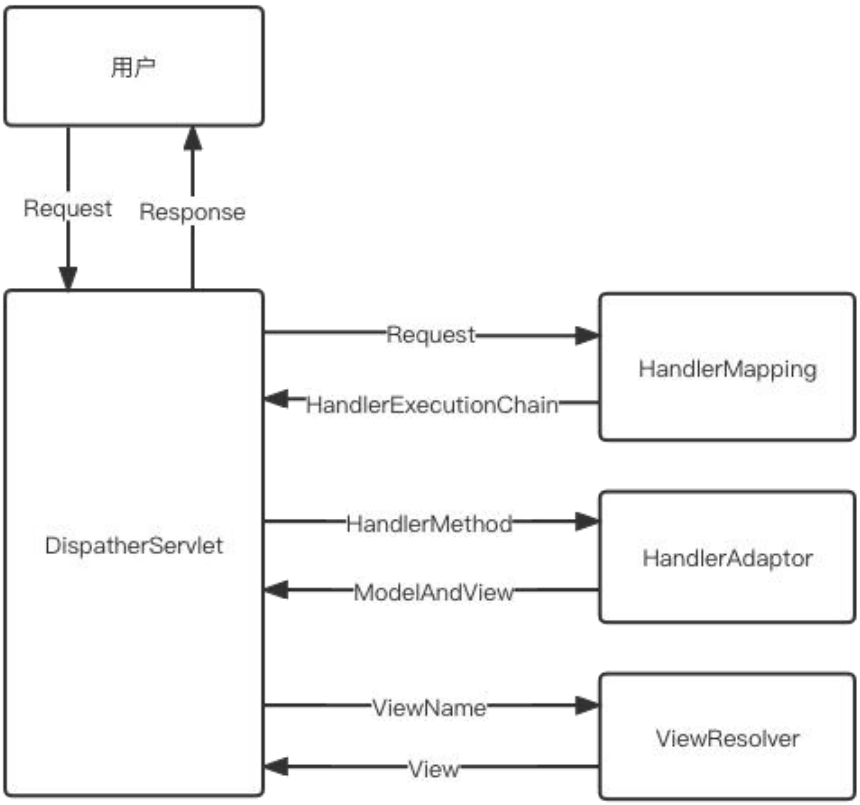

图 4-1 系统总体架构图

每次 HTTP 请求都会经过如下这些执行流程：

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

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。

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。

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。

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。

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。

最后由视图对象将数据渲染到视图中去。

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

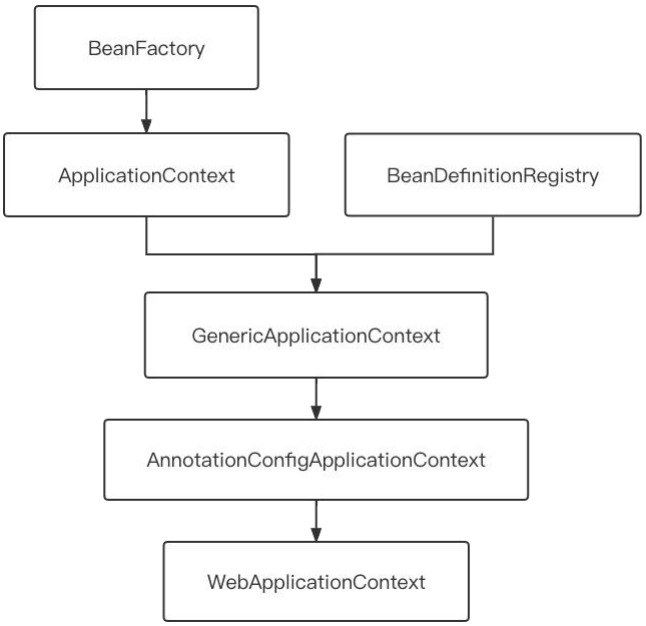

图 5-1 控制反转容器继承设计图

控制反转容器的继承关系架构如下：

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。

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

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。

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、omponent 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。

9、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。

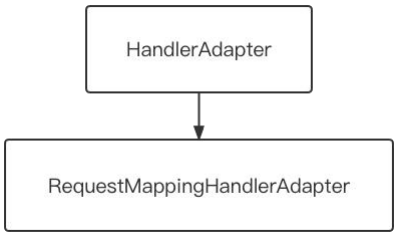

图 5-2 处理器适配器继承设计图

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：

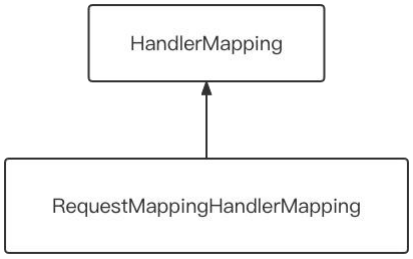

图 5-3 处理器映射器继承图

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。

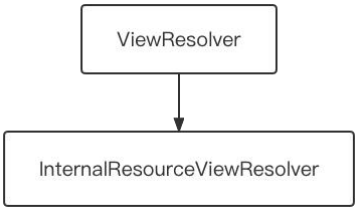

图 5-4 视图解决器

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。

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。

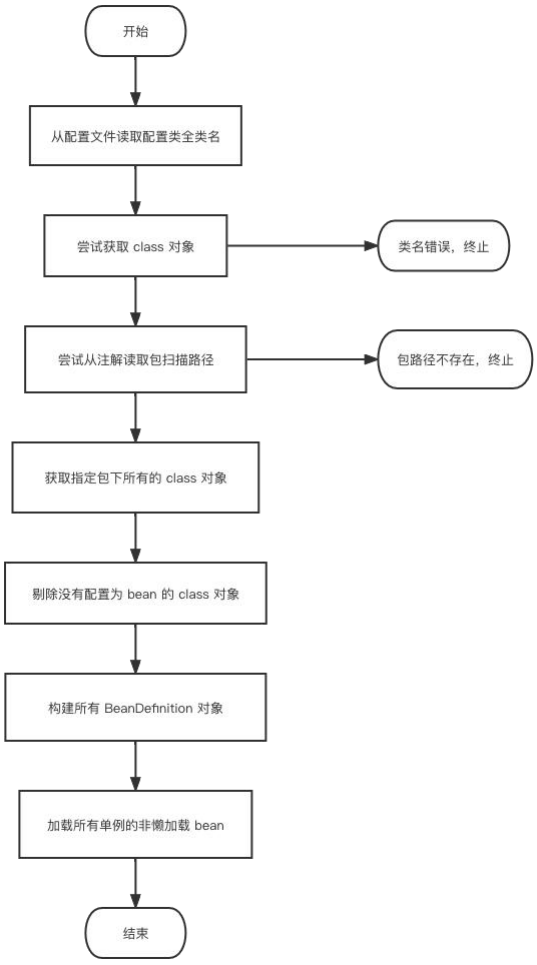

图 6-4 控制反转容器初始化流程图

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

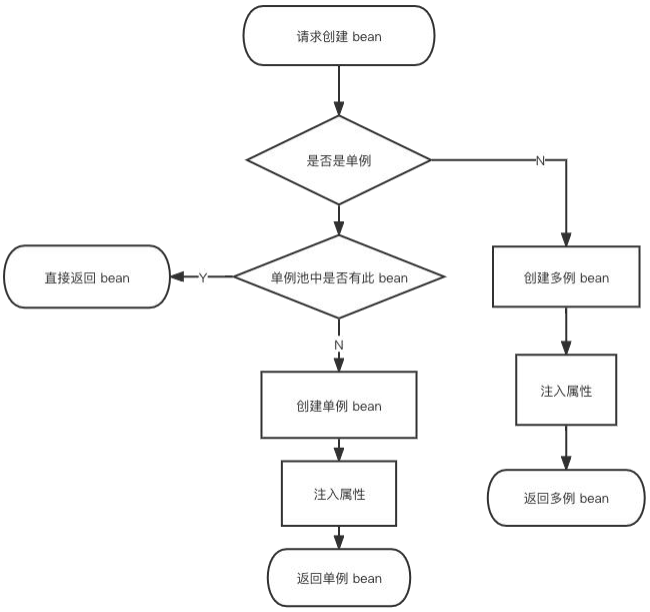

图 6-6 对象创建流程图

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。

4、若单例池中有，直接返回，若单例池中没有，则创建

5、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。

6、递归创建对象完成，如果是单例 Bean，放入单例池中。

代码 6-7 对象创建关键代码

```
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。

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);
    }
}
```

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	正确输出日志信息，表示配	通过，和预期结

	为单例在容器创建 Bean 定义时，日志输出 Bean 的作用域。	置 Bean 为单例成功。	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。

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		处理器被正确配置。		通过，和预期结果一致。

	理器中。		
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	多个拦截器和处理器正确	通过，和预期结

	多个拦截器是否和处理一同封装为处理器执行链对象。	被封装为处理器执行链对象。	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。

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。

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。

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。

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。