

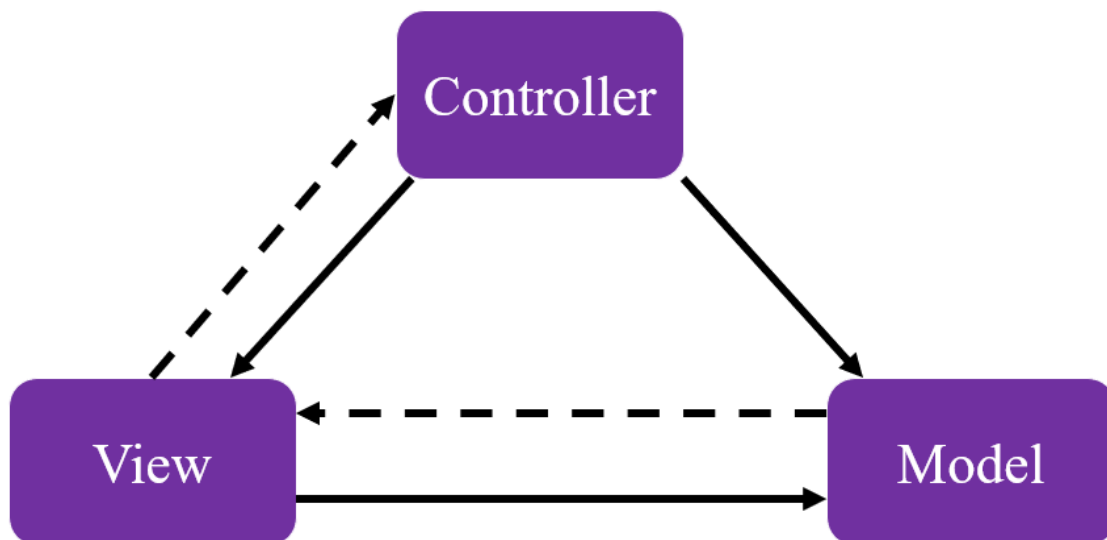
Part 3：功能分析与建模研讨

- 在Part1中：
 - 主要是对Spring MVC核心组件的功能进行了介绍，并且从“流程化”的角度介绍了一下Spring MVC的核心流程和核心功能，并以DispatcherServlet类为代表进行了功能分析。
- 在Part2中：
 - 我对Spring MVC核心流程的设计从“源码分析”的角度进行了详细的分析介绍；
 - 并分析了Spring MVC的“重要组件”的类的设计以及类之间的具体关系和交互，并以此为基础分析面向对象的具体应用和思想。
- 在Part3中：
 - 首先，我会将Spring MVC回归到MVC框架上；
 - 其次，我会从“整体流程”的角度给出一个整体类图，并分析类之间的具体关系和交互；
 - 再次，我会针对Spring MVC的核心流程，具体分析流程中使用到的面向对象设计模式，即阐明Spring MVC的高级设计意图。
 - 最后，针对Spring MVC的核心流程，我将依据设计模式分析其遵循的面向对象原则，并阐述面向对象的核心思想。

一、MVC思想：回溯Spring MVC中的核心流程并回归MVC框架

1、回顾MVC框架模式

MVC (Model - View - Controller) 框架模式是web层的一种复合设计模式，三个核心部件之间的关系图如下所示：



注：上图中实线箭头代表调用，虚线箭头代表事件

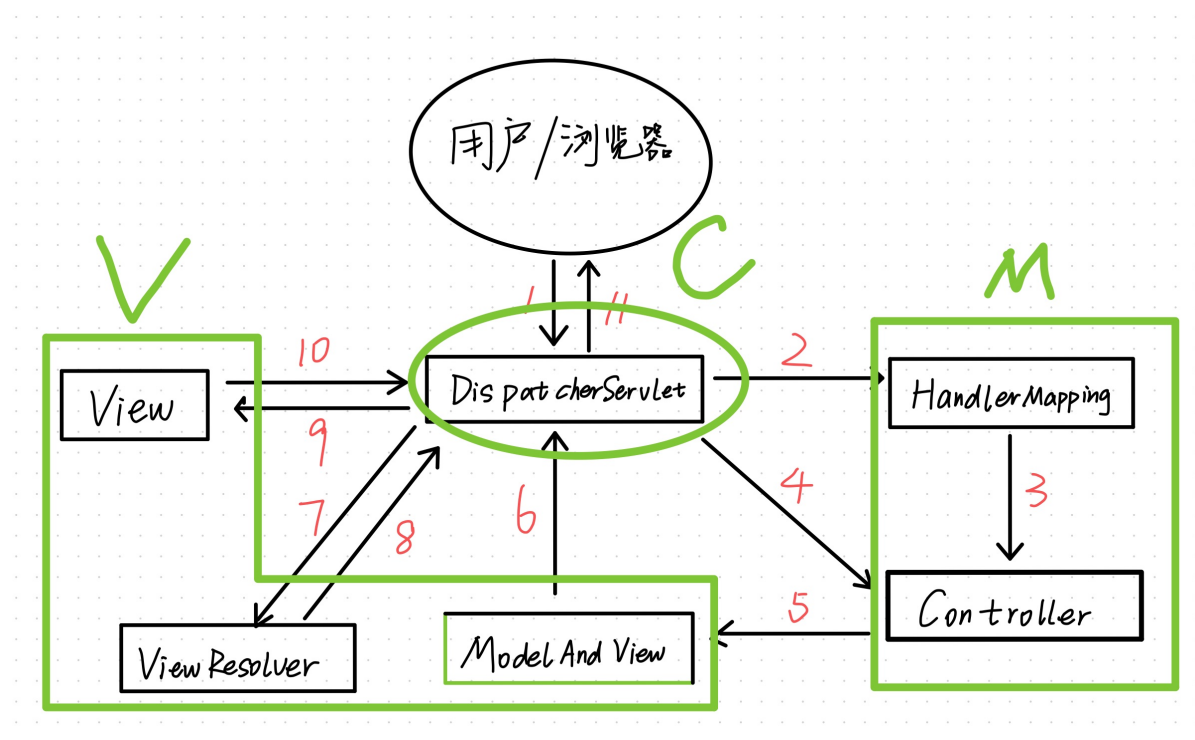
由上图可知，MVC中的Controller类似于一个中转站，会决定调用哪个Model（模型）处理用户请求以及调用哪个View（视图）呈现给用户；MVC框架将M和V分离，实现了前后端代码的分离；

其中三个核心部件的具体描述如下：

- (1) **Model：模型**，所有的用户数据、状态以及程序逻辑
- (2) **View：视图**，呈现模型，类似于Web程序中的界面
- (3) **Controller：控制器**，作用于Model和View上，负责获取用户的输入信息，进行解析并反馈给模型

2、将Spring MVC回归到MVC框架上

我们来回顾一下在Part1中给出的整体流程图：



在Spring MVC框架中，DispatcherServlet为前端控制器，是Spring MVC的中枢，负责从视图中获取用户请求并且分派给相应的处理器处理，并决定用哪个视图去把数据呈现给用户

在上图中，我们来将MVC框架复现到Spring MVC中：

DispatcherServlet对应MVC中的C（Controller）

HandlerMapping和Controller对应MVC中的M（Model）

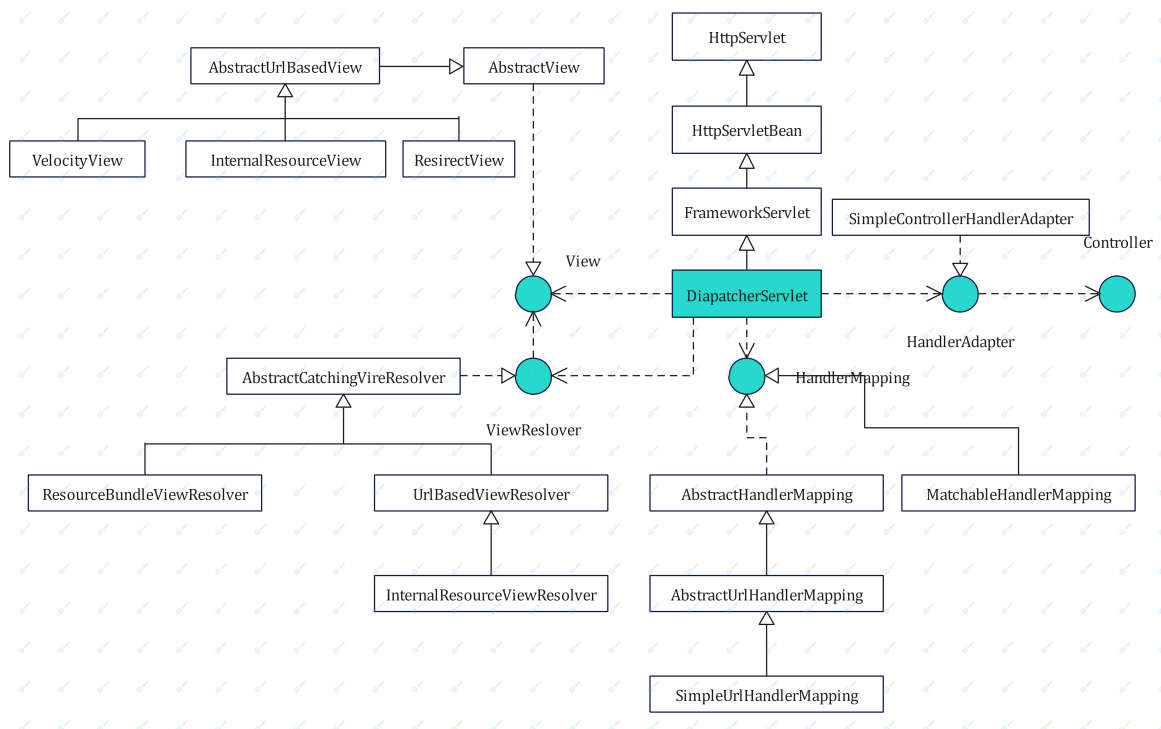
View、ViewResolver和ModelAndView对应MVC中的V（View）

注：MVC中的Controller和Spring MVC中的Controller是不同的，在Spring MVC中，前端控制器DispatcherServlet对应MVC框架中的Controller，是核心控制器；而Spring MVC的组件Controller负责业务和数据的处理，是MVC框架中的Model的一部分。

通过这样的回归，结合Part2中我们对于Spring MVC一个请求的核心流程的详细分析，我们就可以清晰的知道接收到HTTP请求之后Spring MVC框架是如何调用Controller、Model和View来处理请求并返回的。

二、Spring MVC：整体类图及类之间的交互关系分析

1、Spring MVC的整体类图



从上面类图中我们可以很清晰地划分出6个核心组件，其中DispatcherServlet是Spring MVC的核心，其他5个组件需要依赖从DispatcherServlet中得到的请求消息进行工作，所以**DispatcherServlet与其他5个组件的类之间是依赖关系**；

对于每个组件而言，其内部的类通过继承关系和实现关系来进行交互；

注：其中 `HandlerMapping` 和 `HandlerAdapter` 的内部的类间关系在Part2中已经给出过，这里只挑选几个类象征性的表示一下其中的类间关系。

2、类之间的交互关系

(1) 依赖关系

Spring MVC以DispatcherServlet类为核心，其他5个核心组件HandlerAdapter类、Controller类、HandlerMapping类、View类和ViewResolver类**依赖**于核心类DispatcherServlet；

(2) 继承关系和实现关系

继承关系：继承关系是出现频率最高的类间关系，6个核心组件各自的内部类间关系中都包括很多继承关系；

实现关系：对于一个组件而言，其相当于是一个接口，每个组件都需要依靠其内部的类来进行实现。

注：我的源码分析关注的主要是 *Spring MVC* 对于一个请求的核心组件和核心流程，在这个流程中，*Spring MVC* 框架主要采用的就是继承、依赖和实现关系，其他关系比如组合、聚合等在这里没有很明显的体现，但是在*Spring*的其他模块中是有很多体现的，由于这不是我源码分析的内容，这里我们不予以赘述。

三、高级设计意图：设计模式及实例

1、策略模式

- **意图：**定义一系列的算法,把它们一个个封装成类,并且使它们可相互替换，使算法独立于客户而独立变化。
- **主要解决：**在有多种算法相似的情况下，使用 if-else 难以维护。
- **应用场景：**一个系统有许多许多类，而区分它们的只是它们直接的行为。
- **实例：**

策略模式在Spring MVC中经常能看到：

在DispatcherServlet初始化组件时，其父类FrameworkServlet会调用DispatcherServlet中的onRefresh方法：

```
1 @Override
2     protected void onRefresh(ApplicationContext context) {
3         initStrategies(context);
4     }
```

而onRefresh会调用initStrategies函数来对九大组件进行初始化：

```
1 protected void initStrategies(ApplicationContext context) {
2     initMultipartResolver(context);
3     initLocaleResolver(context);
4     initThemeResolver(context);
5     initHandlerMappings(context);
6     initHandlerAdapters(context);
7     initHandlerExceptionResolvers(context);
8     initRequestToViewNameTranslator(context);
9     initViewResolvers(context);
10    initFlashMapManager(context);
11 }
```

我们可以看到DispatcherServlet在初始化组件时会传入相应的接口从而获取该组件的实现类集合，这里对于每个组件的初始化函数都是策略模式的应用场景，以initLocaleResolver为例我们看一下LocaleResolver初始化一个默认的本地化处理组件时：

```
1 private void initLocaleResolver(ApplicationContext context) {
2     try {
3         ///!
4         this.localeResolver = context.getBean(LOCALE_RESOLVER_BEAN_NAME,
5 LocaleResolver.class);
6         if (logger.isTraceEnabled()) {
7             logger.trace("Detected " + this.localeResolver);
8         }
9         else if (logger.isDebugEnabled()) {
10            logger.debug("Detected " +
11 this.localeResolver.getClass().getSimpleName());
12        }
13    }
14    catch (NoSuchBeanDefinitionException ex) {
15        // we need to use the default.
16        ///!
17    }
```

```

15         this.localeResolver = getDefaultStrategy(context,
LocaleResolver.class);
16         if (logger.isTraceEnabled()) {
17             logger.trace("No LocaleResolver '" +
LOCALE_RESOLVER_BEAN_NAME +
18                 "': using default [" +
this.localeResolver.getClass().getSimpleName() + "]");
19         }
20     }
21 }

```


由上面代码可知传入了LocaleResolver.class，而这个类有多个实现类，包括AcceptHeaderLocaleResolver、CookieLocaleResolver、FixedLocaleResolver等，对应多种不同的处理方式，可以通过绑定对应组件来决定使用哪一种处理方式。

但是试想如果使用if-else的处理方式，仅仅一个LocaleResolver组件的代码就会变得冗长。

2、责任链模式

- **意图：**定义多个可以处理请求（承担责任）的类，并将它们连成一条链，沿着该链向下传递请求（责任），直到有能力解决问题（承担责任）的类处理之。
- **主要解决：**使得请求者和处理者之前关系解耦，提高了代码灵活性。
- **应用场景：**一个请求有多个类可以处理，但具体由哪个类处理要在运行时刻确定；
- **实例：**

在核心类DispatcherServlet中的HandlerExecutionChain类，就是责任链模式的具体类，其贯穿在DispatcherServlet处理请求的最核心方法doDispatch中，HandlerExecutionChain类的定义如下：


HandlerExecutionChain ▶

```

SF logger : Log
F handler : Object
  interceptorIndex : int
  interceptorList : List<HandlerInterceptor>
  interceptors : HandlerInterceptor[]
  HandlerExecutionChain(Object)
  HandlerExecutionChain(Object, HandlerInterceptor[])
  addInterceptor(HandlerInterceptor) : void
  addInterceptors(HandlerInterceptor[]) : void
  applyAfterConcurrentHandlingStarted(HttpServletRequest, HttpServletResponse) : void
  applyPostHandle(HttpServletRequest, HttpServletResponse, ModelAndView) : void
  applyPreHandle(HttpServletRequest, HttpServletResponse) : boolean
  getHandler() : Object
  getInterceptors() : HandlerInterceptor[]
  initInterceptorList() : void
  toString() : String
  triggerAfterCompletion(HttpServletRequest, HttpServletResponse, Exception) : void

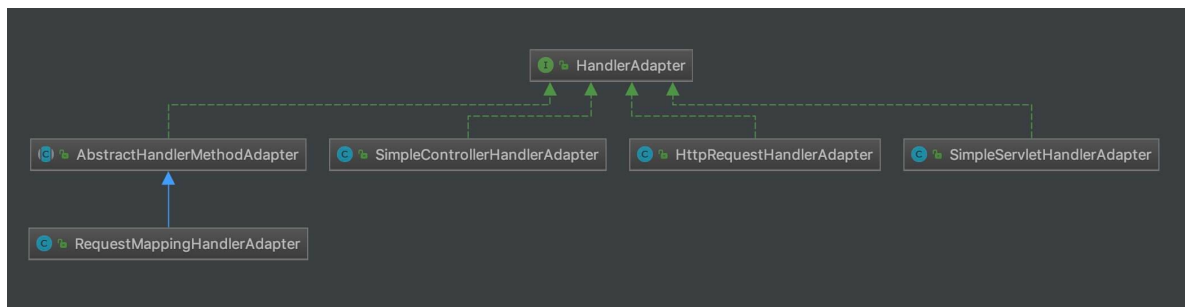
```

HandlerExecutionChain本身不处理请求，它主要负责请求的拦截器的执行和请求的处理，只是将请求分配给在链上注册的处理器执行，这便体现了一种责任链的实现方式，减少了责任链本身与处理逻辑之间的耦合的同时，规范了整个处理请求的流程。

3、适配器模式

- **意图**：把一个类的接口扩展成客户端所期待的另外一个接口。
- **主要解决**：使得原来不能一起使用的两个类（一个接口和一个使用该接口的类）一起工作。
- **应用场景**：需要一个统一的输入接口，而输入端的类型不可知。这个时候，我们可以暴露一个接口适配器给客户端，让客户端自己定制。
- **实例**：

回顾一下组件HandlerAdapter的类图：



由上图可知，它有四个实现类，而对于这四个实现类中不同类型的Handler，Spring MVC都实现了不同的HandlerAdapter。即通过HandlerAdapter接口，springMVC还将Servlet、Controller、HttpRequestHandler等类作为handler，实现了相应的适配器类。

当我们需要不同handler处理请求时，我们只需要关注HandlerAdapter的实现类，重写其中的handler方法，就可以完成请求的处理，而不需要关注handler的本身的类型或方法等。

4、组合模式

- **意图**：将对象组合成树形结构以表示“部分-整体”的层次结构，用户对单个对象和组合对象的使用具有一致性。
- **主要解决**：使高层调用简单。
- **应用场景**：
 - 维护和展示部分-整体关系的场景（如树形菜单、文件和文件夹管理）。
 - 从一个整体中能够独立出部分模块或功能的场景。
- **实例**：（这一部分不是我源码分析的重点部分，所以简要提一下，主要是想介绍一下组合模式的面向对象思想）

springMVC通过组合模式，使得用户或者说框架本身在进行配置时，就通过操作WebMvcConfigurer类及其衍生类这个整体就可以。

四、面向对象的原则和思想核心要素

- **单一职责原则**：一个类只负责一件事情。
- **开放关闭原则**：软件中的对象（类，模块，函数等）对于扩展是开放的，对于修改是关闭的。
- **里氏替换原则**：所有引用基类的地方必须能够透明地使用其子类的对象（子类可以扩展父类的功能，但是不能改变父类原有的功能）。
- **依赖倒置原则**：高层模块不依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。
- **接口隔离原则**：客户端不应该依赖他不需要的接口，类间的依赖关系应该建立在最小的接口上。
- **迪米特原则**：一个对象应该对其他对象保持最小的了解。

事实上，不同的设计模式就蕴含并体现了面向对象的原则和核心思想，于是依据我们上面针对不同设计模式举的例子，这里我们根据不同的设计模式来阐述一下Spring MVC框架中体现的面向对象原则和核心思想。

1、策略模式：接口、继承和多态思想，开放封闭原则

在上面举的LocaleResolver例子中，使用LocaleResolver.class类包含的多个实现类来对应多种不同的处理方式，就利用**接口、继承和多态**来替代原来的if-else实现方法，增加了代码的可读性和可维护性；**类是可扩展的，符合面向对象的开放封闭原则。**

2、责任链模式：解耦（低耦合、高内聚），单一职责原则、依赖倒置原则

在上面举的HandlerExecutionChain例子中，责任链将顺序过程处理的代码和逻辑，**解耦**成执行的顺序逻辑和对应的负责人，**请求者和处理者之间关系解耦**，对应的责任链只需要关心责任处理的顺序，**解耦后抽象度更高，灵活性更好，模块依赖于抽象，符合依赖倒置原则**；并且在责任链中，**每个类只负责一件事情，符合单一职责原则。**

责任链模式的解耦，满足“低耦合、高内聚”的原则。

3、适配器模式：解耦、高复用，依赖倒置原则

在HandlerAdapter中，适配器模式将**所需要用的类与使用者相解耦**，使用者只需关注相应的适配器接口提供是接口方法即可，达到了**高复用的目的**。**使用者只需要依赖抽象出来的接口即可，满足依赖倒置原则。**

4、组合模式：解耦

组合模式依据整体-部分的关系很好的简化了代码量并实现了解耦，满足“低耦合、高内聚”原则。

通过Spring MVC所使用的设计模式，我们可以得出Spring MVC框架的核心流程满足**单一职责原则、依赖倒置原则和开放封闭原则**；当然，通过Spring MVC的整体类图，我们也可以看出来其满足**接口隔离原则**，类之间的依赖关系都建立的最小的接口上。

在Spring MVC的核心流程中，处处体现着面向对象的最核心思想：**高内聚、低耦合，通过高抽象化、接口、继承、多态和最重要的解耦思想，将“面向对象”体现的淋漓尽致。**

五、结语

对Spring MVC的源码分析阅读就告一段落了，在这个过程中，我不但对Spring MVC的MVC架构有了深刻理解，更是自己手动进行了一次详尽的源码阅读，真正了解了Spring MVC框架是如何处理利用核心部件一个用户端请求的（即核心流程），在这个过程中，我也体会到了Spring MVC中无处不在的面向对象思想和原则，受益匪浅。