

**设计模式与系统架构分析报告**



**系 （院）：**  计算机科学学院

**课 程：** 设计模式与系统架构

**指导教师：**  向华

**专业班级：** 计算机科学与技术11403班

**姓 名：** 王 岩

**学 号：** 201403680

**设计时间：** 2017.6.12 - 2017.6.25

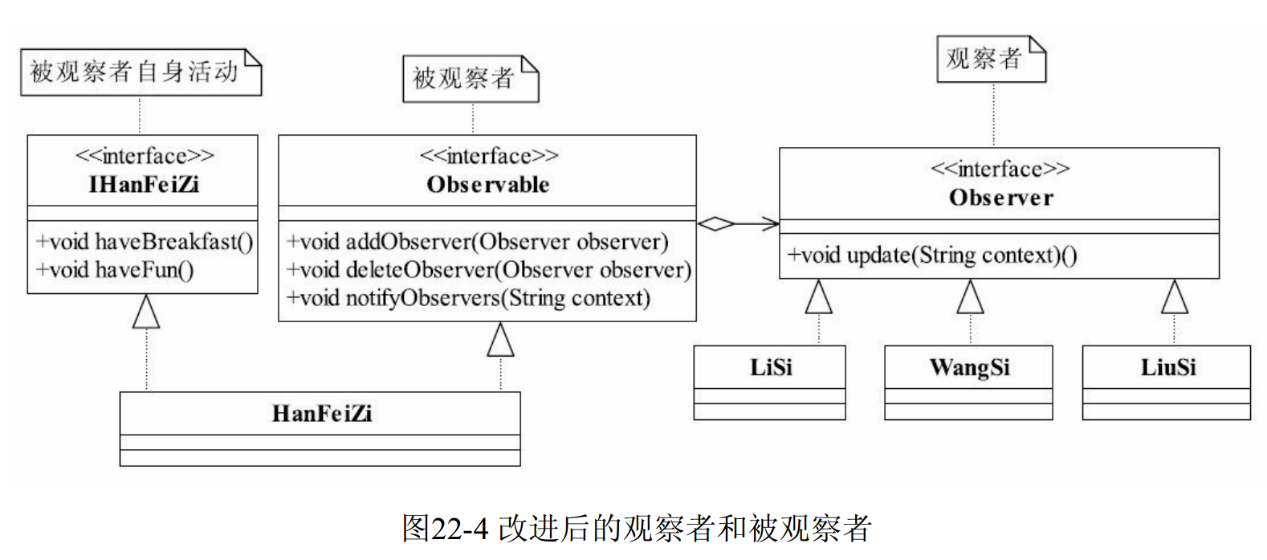
观察者模式

1.

2.代码介绍

该代码是作者专门用来介绍观察者模式而设计的的Demo。简明清晰地分析了观察这模式的结构，让观察者模式易于理解，用来认识和学习观察者模式非常好。

3.类结构图



4.由上图可以清晰的看出：观察者模式拥有三个角色，分别是：被观察者、观察者、观察者对象。观察者通过update（）方法监听被观察者的活动，当被观察者发生活动时，就调用该方法。具体的观察者：观察者接口的具体实现，在这个角色中，将定义被观察者对象状态发生变化时所要处理的逻辑。

观察者（Observer）将自己注册到被观察对象（Subject）中，被观察对象将观察者存放在一个容器（Container）里。被观察对象发生了某种变化（如图中的SomeChange），从容器中得到所有注册过的观察者，将变化通知观察者。观察者告诉被观察者要撤销观察，被观察者从容器中将观察者去除。观察者将自己注册到被观察者的容器中时，被观察者不应该过问观察者的具体类型，而是应该使用观察者的接口。这样的优点是：假定程序中还有别的观察者，那么只要这个观察者也是相同的接口实现即可。一个被观察者可以对应多个观察者，当被观察者发生变化的时候，他可以将消息一一通知给所有的观察者。基于接口，而不是具体的实现——这一点为程序提供了更大的灵活性。

5.代码

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*被观察者\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **interface** IHanFeiZi {

/\*\*

\*韩非子吃饭

\*/

**public** **void** haveBreakfast();

/\*\*

\* 韩非子娱乐

\*/

**public** **void** haveFun();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**import** java.util.ArrayList;

**public** **class** HanFeiZi **implements** Observable,IHanFeiZi{

**private** ArrayList<Observer> observableList = **new** ArrayList<>();

@Override

**public** **void** addObserver(Observer observable) {

**this**.observableList.add(observable);

}

@Override

**public** **void** deleteObserver(Observer observable) {

**this**.observableList.remove(observable);

}

@Override

**public** **void** notifyObserviers(String context) {

**for**(Observer observer : observableList) {

observer.update(context);

}

}

@Override

**public** **void** haveBreakfast() {

System.***out***.println("韩非子：开始吃饭");

**this**.notifyObserviers("韩非子在吃饭");

}

@Override

**public** **void** haveFun() {

System.***out***.println("韩非子：开始娱乐");

**this**.notifyObserviers("韩非子在娱乐");

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*观察者\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*被观察者接口

\* **@author** qiaojy

\*

\*/

**public** **interface** Observable {

/\*\*

\* 增加一个观察者

\* **@param** observable

\*/

**public** **void** addObserver(Observer observable);

/\*\*

\* 删除一个观察者

\* **@param** observable

\*/

**public** **void** deleteObserver(Observer observable);

/\*\*

\* 通知观察者

\* **@param** context

\*/

**public** **void** notifyObserviers(String context);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* 观察者接口

\* **@author** qiaojy

\*/

**public** **interface** Observer {

**public** **void** update(String context);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*观察者对象\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** LiSi **implements** Observer{

@Override

**public** **void** update(String context) {

System.***out***.println("李斯：观察到韩非子活动，开始向老师汇报");

**this**.reportToQinShihuang(context);

}

**private** **void** reportToQinShihuang(String reportcontent) {

System.***out***.println("报告，韩非子有活动:"+reportcontent);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** WangSi **implements** Observer{

**private** **final** **static** String ***TAG*** = WangSi.**class**.getSimpleName();

@Override

**public** **void** update(String context) {

// **TODO** Auto-generated method stub

System.***out***.println(***TAG***+"观察到韩非子有变化");

report(context);

}

**private** **void** report(String context) {

System.***out***.println("报告韩非子的状态"+context);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*main函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** ClientMain {

**public** **static** **void** main(String[] args) {

Observer lisi = **new** LiSi();

Observer wangsi = **new** WangSi();

HanFeiZi hanFeiZi = **new** HanFeiZi();

hanFeiZi.addObserver(lisi);

hanFeiZi.addObserver(wangsi);

hanFeiZi.haveBreakfast();

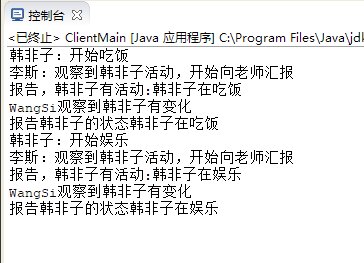
hanFeiZi.haveFun();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*code ending\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

6.运行结果



7.小结

观察者模式（Observer）完美的将观察者和被观察的对象分离开。举个例子，用户界面可以作为一个观察者，业务数据是被观察者，用户界面观察业务数据的变化，发现数据变化后，就显示在界面上。面向对象设计的一个原则是：系统中的每个类将重点放在某一个功能上，而不是其他方面。一个对象只做一件事情，并且将他做好。观察者模式在模块之间划定了清晰的界限，提高了应用程序的可维护性和重用性。

观察者设计模式定义了对象间的一种一对多的依赖关系，以便一个对象的状态发生变化时，所有依赖于它的对象都得到通知并自动刷新。

String MVC

Spring Web MVC是一种基于Java的实现了Web MVC设计模式的请求驱动类型的轻量级Web框架，即使用了MVC架构模式的思想，将web层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC也是要简化我们日常Web开发的。

另外还有一种基于组件的、事件驱动的Web框架在此就不介绍了，如Tapestry、JSF等。

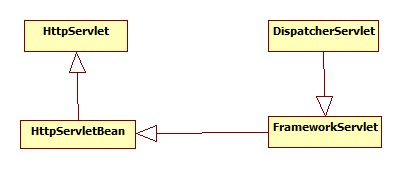
Spring Web MVC也是服务到工作者模式的实现，但进行可优化。前端控制器是DispatcherServlet；应用控制器其实拆为处理器映射器(Handler Mapping)进行处理器管理和视图解析器(View Resolver)进行视图管理；页面控制器/动作/处理器为Controller接口（仅包含ModelAndView handleRequest(request, response) 方法）的实现（也可以是任何的POJO类）；支持本地化（Locale）解析、主题（Theme）解析及文件上传等；提供了非常灵活的数据验证、格式化和数据绑定机制；提供了强大的约定大于配置（惯例优先原则）的契约式编程支持

Spring Web MVC核心架构图

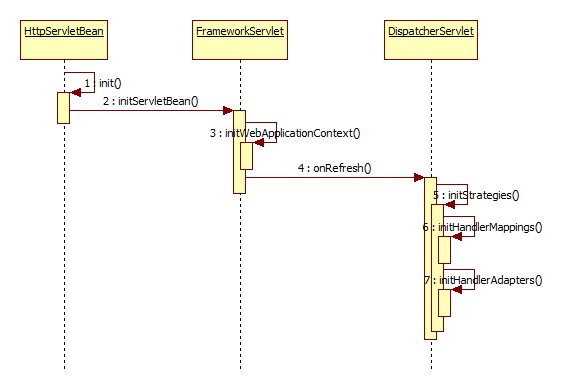


 Spring MVC代码分析

DispatcherServlet的继承体系如：



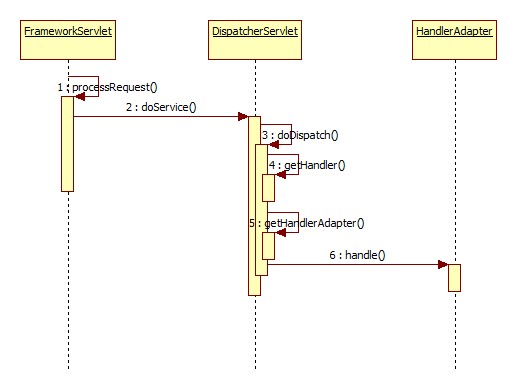
看到它们继承自HttpServlet，你就知道初始化过程应该是从init方法开始了，整个初始化的流程为：



上面的时序图中，在5的initStragegies()中除了调用6,7的函数外，还有几个类似的初始化函数.

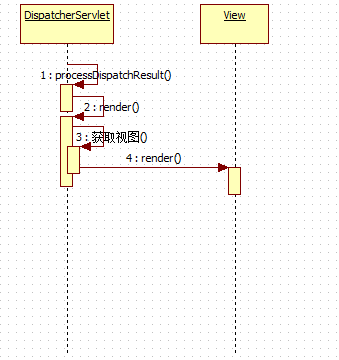
请求处理过程

   如POST、GET的实现方法在FrameworkServlet中，分别是doPost、doGet等请求的处理跳转到了processRequest函数中，最终进入DispatcherServlet的doService函数，详细的流程如：



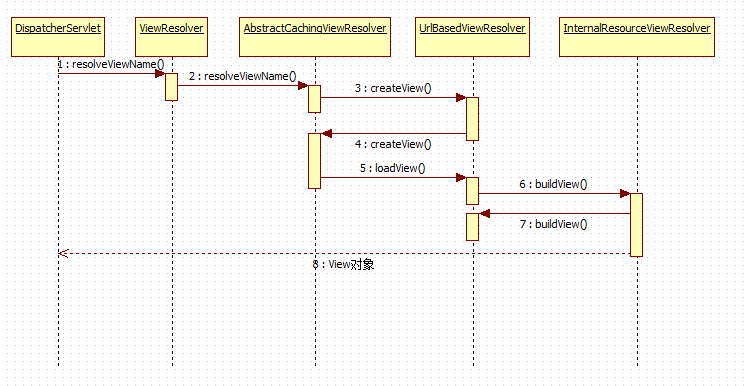
上面的时序图展示了详细的请求处理流程，其中最重要的是doDispatch函数，其中包含了整个的处理逻辑

视图的解析在”processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);“中完成，处理流程图如下：



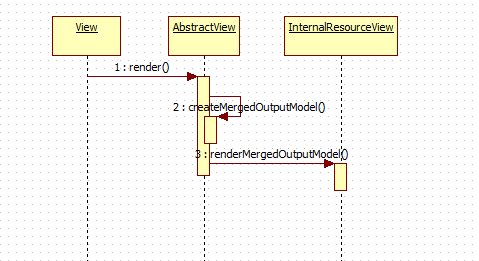
        processDispatchResult主要处理异常、请求状态及触发请求完成事件，视图的渲染工作交给了render().

视图解析器InternalResourceViewResolver，视图的获取过程为：



这里有几处子类覆写了父类函数，并回调其覆盖的函数，主要是为了针对特定的实现类添加相关的应用逻辑。

以视图InternalResourceView为例，则视图的渲染过程为：



Spring MVC是一个典型的教科书式的mvc构架，而不像[struts](http://baike.baidu.com/item/struts)等都是变种或者不是完全基于mvc系统的框架，对于初学者或者想了解mvc的人来说我觉得 spring是最好的，它的实现就是教科书！第二它和[tapestry](http://baike.baidu.com/item/tapestry)一样是一个纯正的servlet系统，这也是它和tapestry相比 struts所具有的优势。而且框架本身有代码，看起来容易理解。

String MVC的优势：

1. 清晰的角色划分：控制器(controller)、验证器(validator)、命令对象(command obect)、表单对象(form object)、模型对象(model object)、Servlet分发器(DispatcherServlet)、处理器映射(handler mapping)、试图解析器(view resoler)等等。每一个角色都可以由一个专门的对象来实现。
2. 强大而直接的配置方式：将框架类和应用程序累都能作为JavaBean配置，支持跨多个context的引用，例如，在web控制器中对业务对象和验证器validator)的引用。
3. 可适配、非侵入：可以根据不同的应用场景，选择何事的控制器子类(simple型、command型、from型、wizard型、multi-action型或者自定义)，而不是一个单一控制器(比如Action/ActionForm)继承。
4. 可重用的业务代码：可以使用现有的业务对象作为命令或表单对象，而不需要去扩展某个特定框架的基类。
5. 可定制的绑定(binding)和验证(validation):比如将类型不匹配作为应用级的验证错误，这可以保证错误的值。再比如本地化的日期和数字绑定等等。在其他某些框架中，你只能使用字符串表单对象，需要手动解析它并转换到业务对象。
6. 可定制的handler mapping和view resolution：[**spring**](http://lib.csdn.net/base/javaee)提供从最简单的URL映射，到复杂的、专用的定制策略。与某些web MVC框架强制开发人员使用单一特定技术相比，Spring显得更加灵活。
7. 灵活的model转换：在Springweb框架中，使用基于Map的键/值对来达到轻易的与各种视图技术集成。
8. 可定制的本地化和主题(theme)解析：支持在JSP中可选择地使用Spring标签库、支持JSTL、支持Velocity(不需要额外的中间层)等等。
9. 简单而强大的JSP标签库(Spring Tag Library)：支持包括诸如数据绑定和主题(theme)之类的许多功能。他提供在标记方面的最大灵活性。
10. JSP表单标签库：在Spring2.0中引入的表单标签库，使用在JSP编写表单更加容易。
11. Spring Bean的生命周期可以被限制在当前的HTTp Request或者HTTp Session。准确的说，这并非Spring MVC框架本身特性，而应归属于Spring MVC使用的WebApplicationContext容器。