My github: **https://github.com/wh1291104857**

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： 计科 11402

姓　　名： 王豪

学　　号： 201403639

设计时间： 2017.6.12 - 2017.6.25

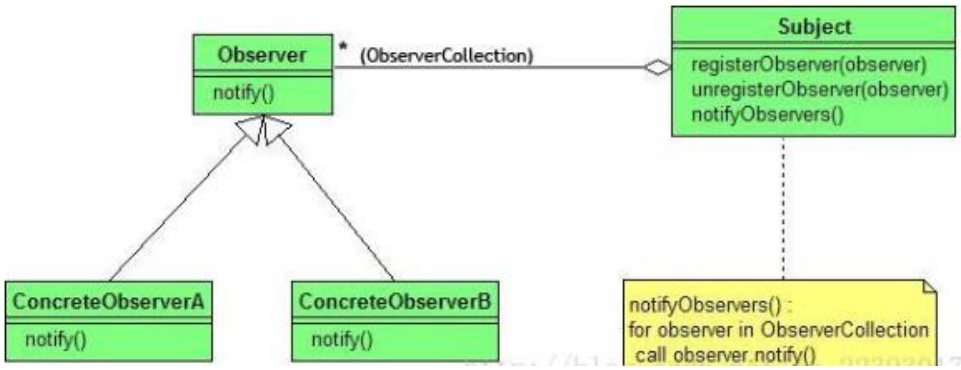
作业一

Github 地址：https://github.com/xiaosong520/ObserverPatternDemo

介绍

观察者模式也被称为**发布-订阅**（Publish/Subscribe）模式，它属于**行为型模式**的一种。观察者模式定义了**一种一对多的依赖关系**，一个主题对象可被**多个观察者**对象同时监听。当这个主题对象状态变化时，会通知所有观察者对象并作出相应处理逻辑。

UML



观察者模式定义了四种角色：抽象主题、具体主题、抽象观察者、具体观察者。

1. **抽象主题（Subject）**：该角色是一个抽象类或接口，定义了增加、删除、通知观察者对象的方法。
2. **具体主题（ConcreteSubject）**：该角色继承或实现了抽象主题，定义了一个集合存入注册过的具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知。
3. **抽象观察者（Observer）**：该角色是具体观察者的抽象类，定义了一个更新方法。
4. **具体观察者（ConcrereObserver）**：该角色是具体的观察者对象，在得到具体主题更改通知时更新自身的状态。

实现代码

抽象观察者（Observer）接口

```
public interface Observer {

     public void update(String msg, TextView tv);
}
```

具体观察者（Person）类

```
public class Person implements Observer {

    // 用户名
    private String name;
    public Person(String name) {
        this.name = name;
    }

    @Override
    public void update(String msg, TextView tv) {
        tv.setText(tv.getText()+name+":"+ msg +"\n");
    }

}
```

抽象主题（Subject）接口

```
public interface Subject {
    /**
     * 增加观察者
     * @param observer
     */
    public void attach(Observer observer);
    /**
     * 删除观察者
     * @param observer
     */
    public void detach(Observer observer);
    /**
     * 通知观察者
     */
    public void notify(String message, TextView v);
}
```

具体主题（XiaosongSubject）类

```
public class XiaosongSubject implements Subject {

    //用于保存订阅了小嵩的博客的用户
    private List<Observer> mPersonList = new ArrayList<>();

    @Override
    public void attach(Observer observer) {
        mPersonList.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        mPersonList.remove(observer);
    }

    @Override
    public void notify(String message, TextView tv) {
        for (Observer observer : mPersonList) {
            observer.update(message,tv);
        }
    }
}
```

MainActivity 类

```
public class MainActivity extends AppCompatActivity implements
View.OnClickListener{

    private Person pMing,pQing,pLiang;
    private XiaosongSubject mSubject;

    private TextView tv_output;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initView();
        initEvent();
```

```
    }

    private void initView() {

        findViewById(R.id.btn_add_one).setOnClickListener(this);
        findViewById(R.id.btn_add_two).setOnClickListener(this);
        findViewById(R.id.btn_add_three).setOnClickListener(this);

        findViewById(R.id.btn_notify).setOnClickListener(this);
        findViewById(R.id.btn_delete).setOnClickListener(this);

        tv_output = (TextView)findViewById(R.id.tv_output);
    }

    private void initEvent() {
        //创建被观察者(具体主题)
        mSubject = new XiaosongSubject();
    }


    @Override
    public void onClick(View v) {
        switch (v.getId()){
            case R.id.btn_add_one://注册观察者  小明
                if (pMing==null){
                    pMing = new Person("小明");
                    mSubject.attach(pMing);
                    Toast.makeText(this,"小明关注了我
",Toast.LENGTH_SHORT).show();
                }else {
                    Toast.makeText(this,"小明已关注我了，不能再重复操作
",Toast.LENGTH_SHORT).show();
                }
                break;

            case R.id.btn_add_two://注册观察者  小琴
                if (pQing==null){
                    pQing = new Person("小琴");
                    mSubject.attach(pQing);
                    Toast.makeText(this,"小琴关注了我
",Toast.LENGTH_SHORT).show();
                }else {
                    Toast.makeText(this,"小琴已关注我了，不能再重复操作
",Toast.LENGTH_SHORT).show();
```

```
                }
                break;

            case R.id.btn_add_three://注册观察者  阿亮
                if (pLiang==null){
                    pLiang = new Person("阿亮");
                    mSubject.attach(pLiang);
                    Toast.makeText(this,"阿亮关注了我
",Toast.LENGTH_SHORT).show();
                }else {
                    Toast.makeText(this,"阿亮已关注我了，不能再重复操作
",Toast.LENGTH_SHORT).show();
                }
                break;

            case R.id.btn_notify://主题(被观察者)更新了内容，通知所有观察者
                tv_output.setText("");
                mSubject.notify("小嵩更新微博啦~ 快来看看吧", tv_output);
                break;

            case R.id.btn_delete://注销观察者  小明
                if (pMing!=null){
                    mSubject.detach(pMing);
                    pMing = null;
                }

                if (pQing!=null){//注销观察者  小琴
                    mSubject.detach(pQing);
                    pQing = null;
                }

                if (pLiang!=null){//注销观察者  阿亮
                    mSubject.detach(pLiang);
                    pLiang = null;
                }

                break;

            default:
                break;
        }
    }
}
```

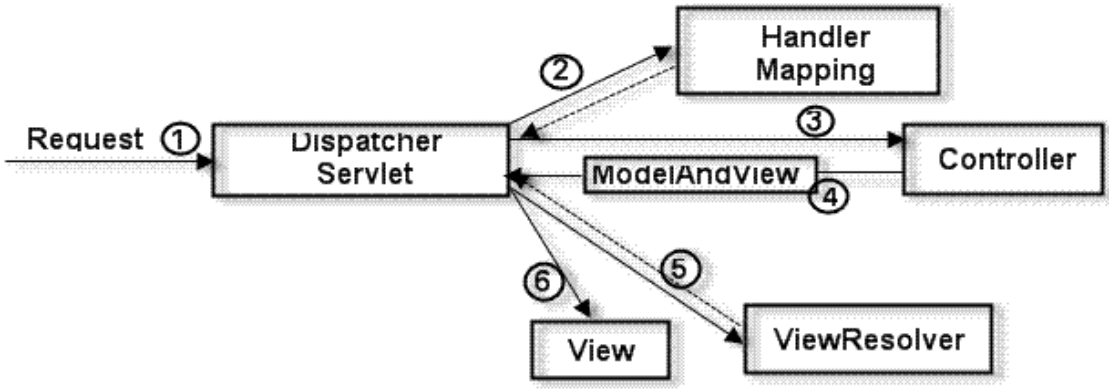
总结

运行界面

在 **Android** 中，有很多场景使用了观察者模式，比如 **android** 的源码里： OnClickListener、 ContentObserver、 android.database.Observable 等；还有第三方开源库 EventBus、RxJava、RxAndroid 等。具体可阅读它们的源码去理解和体会。

作业二

Github ：https://github.com/mybatis/spring



什么是 MyBatis-Spring?

MyBatis-Spring 将 MyBatis 与 Spring 无缝集成。该库允许 MyBatis 参与 Spring 事务，负责构建 MyBatis mappers 和 SqlSession，并将它们注入到其他 bean 中，将 MyBatis 异常转换为 Spring DataAccessException，最后，它可以让您构建您的应用程序代码，而不依赖于 MyBatis， Spring 或 MyBatis-Spring。

动机

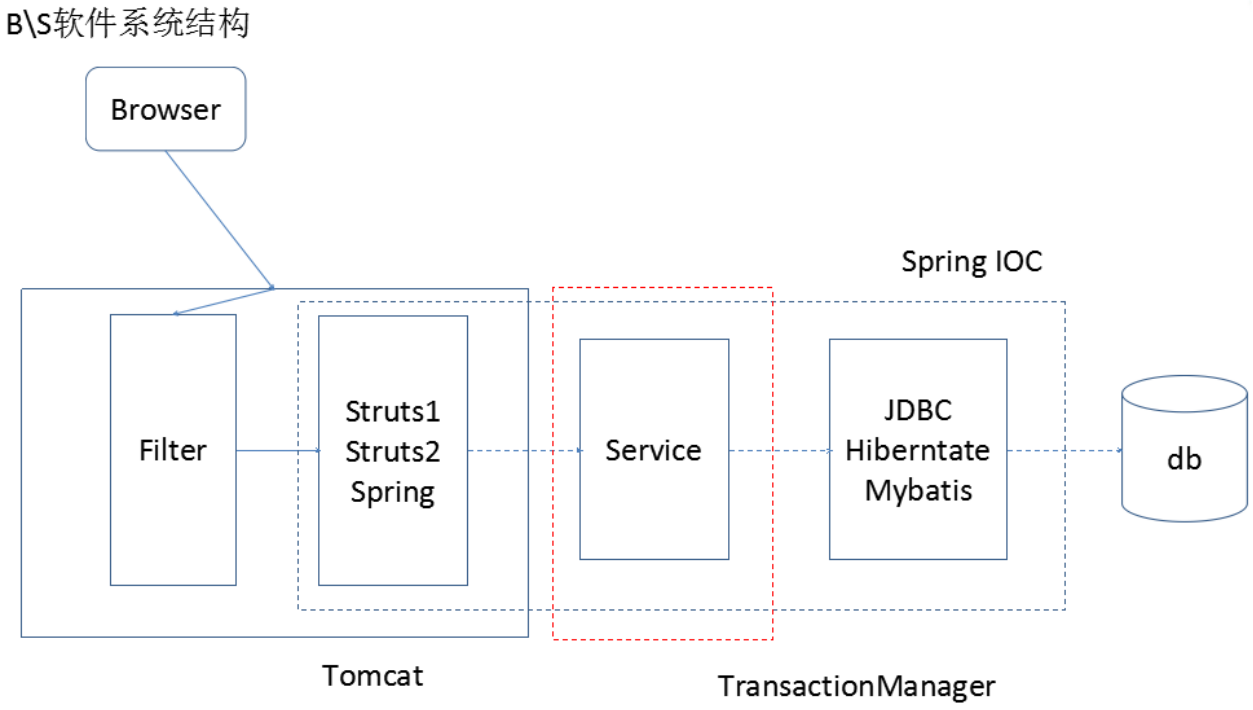
Spring 版本 2 仅支持 iBATIS 版本 2.试图将 MyBatis 3 支持添加到 Spring 3（请参阅 Spring Jira 问题）。不幸的是，Spring 3 发展在 MyBatis 3 正式发布之前结束。因为 Spring 团队不想发布基于 MyBatis 的非发布版本的代码，所以官方的 Spring 支持将不得不等待。鉴于 Spring 对 MyBatis 的支持感兴趣，MyBatis 社区决定是时候重新组织感兴趣的贡献者，并将 Spring 集成添加为 MyBatis 的社区子项目。

要求

在开始使用 MyBatis-Spring 集成之前，非常重要的是您熟悉 MyBatis 和 Spring 术语。本文档不会为 MyBatis 或 Spring 提供背景信息或基本设置和配置教程。

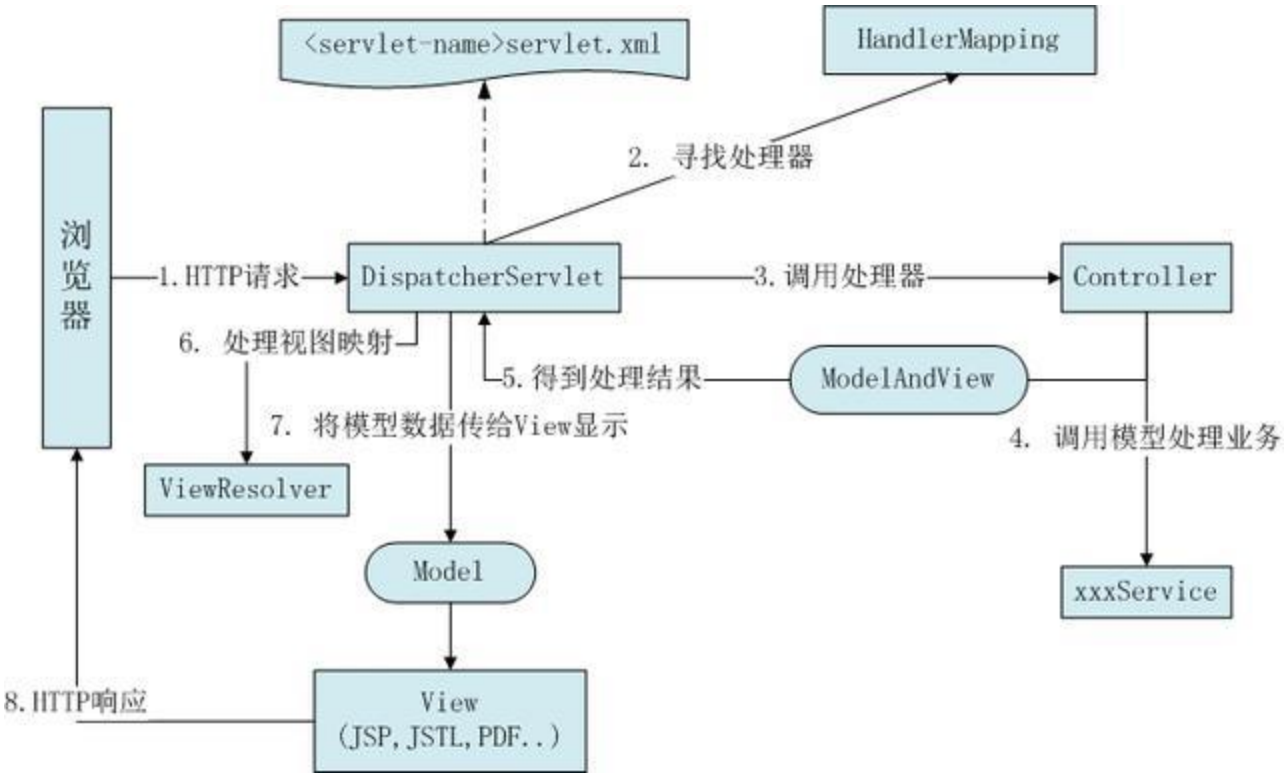
1. Spring mvc 所有的请求都提交给 DispatcherServlet,它会委托应用系统的其他模块负责负责对请求进行真正的处

理工作。



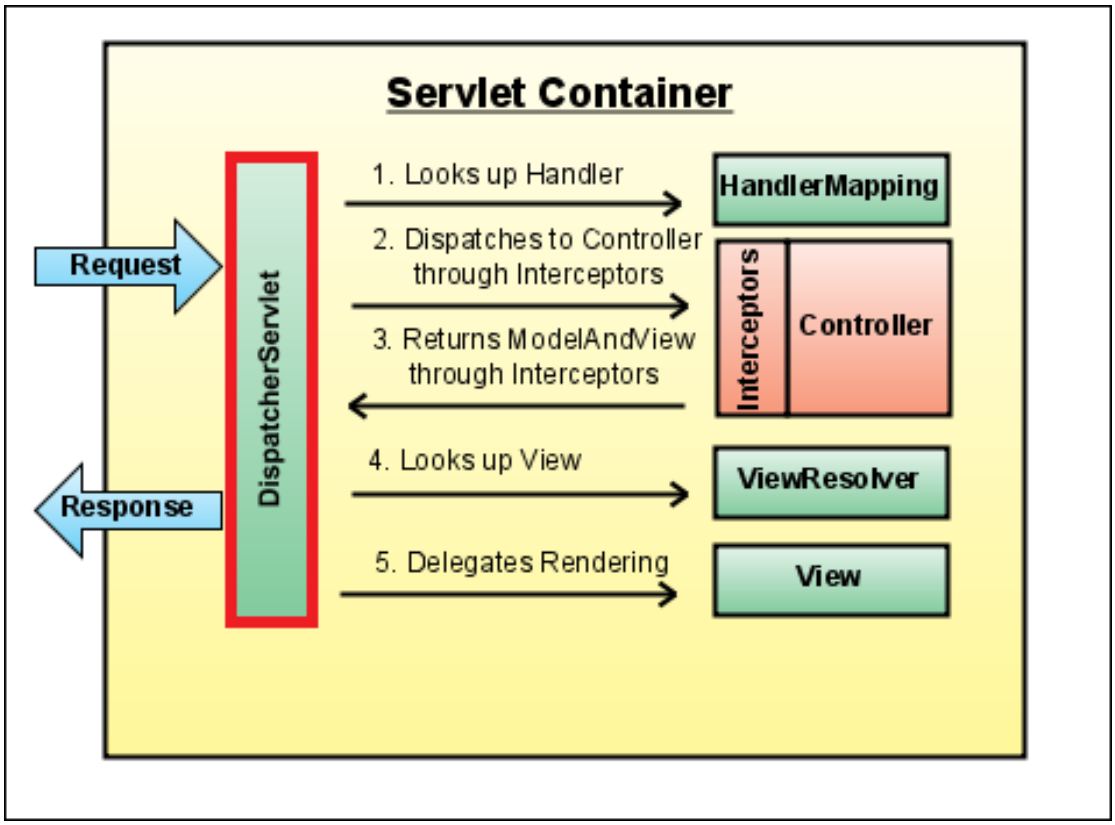
SpringMVC：负责表现层
Service 接口：处理业务
Mapper:持久层
spring 负责将各层之间整合
通过 Spring 管理持久层的 mapper(相当于 Dao 接口)
通过 Spring 管理业务层的 service,service 中可以调用 mapper 接口
Spring 进行事务控制
通过 Spring 管理表现层 handler,handler 中可以调用 service 接口
mapper、service、handler 都属于 javabean
第一步：整合 dao 层
mybatis 和 spring 整合，通过 spring 管理 mapper 接口。
使用 mapper 的扫描器自动扫描 mapper 接口在 spring 中进行注册。
第二步：整合 service 层
通过 spring 管理 service 接口。
使用配置方式将 service 接口配置在 spring 配置文件中。
实现事务控制。
第三步：整合 springmvc
由于 springmvc 是 spring 的模块，不需要整合。
没看懂?没关系再来看张图放松一下。

2. DispatcherServlet 查询一个或多个 HandlerMapping,找到处理请求的 Controller。



相关接口解释 DispatcherServlet 接口： Spring 提供的前端控制器，所有的请求都有经过它来统一分发。在 DispatcherServlet 将请求分发给 Spring Controller 之前，需要借助于 Spring 提供的 HandlerMapping 定位到具体的 Controller。HandlerMapping 接口： 能够完成客户请求到 Controller 映射。 Controller 接口：需要为并发用户处理上述请求，因此实现 Controller 接口时，必须保证线程安全并且可重用。 Controller 将处理用户请求，这和 Struts Action 扮演的角色是一致的。一旦 Controller 处理完用户请求，则返回 ModelAndView 对象给 DispatcherServlet 前端控制器，ModelAndView 中包含了模型（Model）和视图（View）。 从宏观角度考虑，DispatcherServlet 是整个 Web 应用的控制器；从微观考虑，Controller 是单个 Http 请求处理过程中的控制器，而 ModelAndView 是 Http 请求过程中返回的模型(Model)和视图(View)。ViewResolver 接口：Spring 提供的视图解析器（ViewResolver）在 Web 应用中查找 View 对象，从而将相应结果渲染给客户。

3. DispatcherServlet 请请求提交到目标 Controller。



DispatcherServlet 是前端控制器设计模式的实现，提供 Spring Web MVC 的集中访问点，而且负责职责的分派，而且与 Spring IoC 容器无缝集成，从而可以获得 Spring 的所有好处。

DispatcherServlet 主要用作职责调度工作，本身主要用于控制流程，主要职责如下：

1、文件上传解析，如果请求类型是 multipart 将通过 MultipartResolver 进行文件上传解析；

2、通过 HandlerMapping，将请求映射到处理器（返回一个 HandlerExecutionChain，它包括一个处理器、多个 HandlerInterceptor 拦截器）；

3、通过 HandlerAdapter 支持多种类型的处理器(HandlerExecutionChain 中的处理器)；

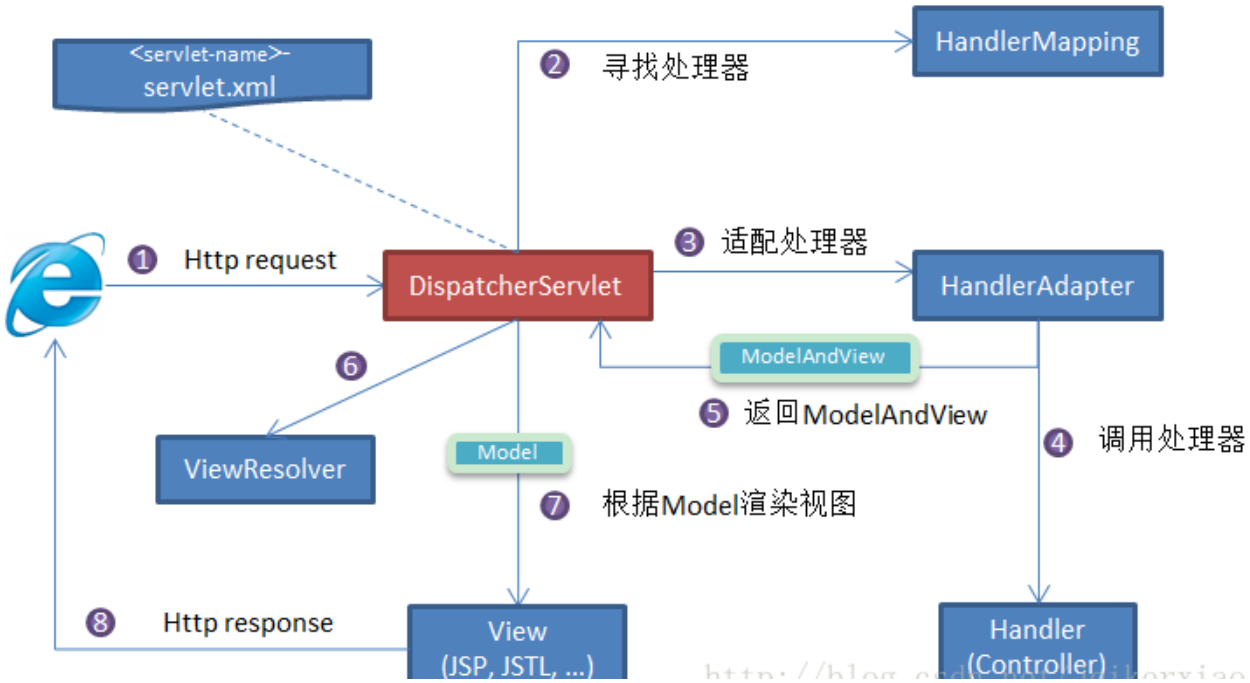
4、通过 ViewResolver 解析逻辑视图名到具体视图实现；

5、本地化解析；

6、渲染具体的视图等；

7、如果执行过程中遇到异常将交给 HandlerExceptionResolver 来解析。

4. Controller 进行业务逻辑处理后，会返回一个 ModelAndView.



给每一个 controller 绑定一个 manager 类，用来处理业务逻辑

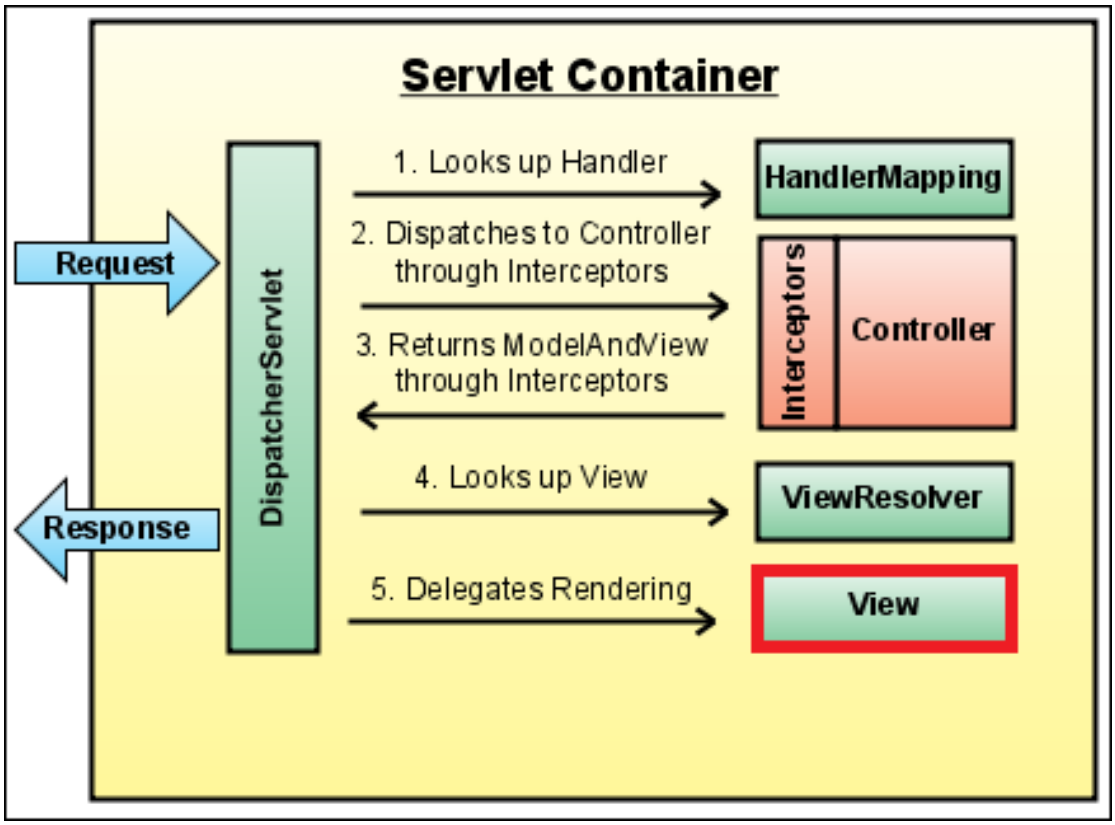
@property (nonatomic,strong)Cinema_ListManager *manager;

通过 manager 回调把网络请求的结果回调给 controller，在 manager 里面处理其他业务逻辑，数据持久化之类的，也可以根据网络请求到的数据来做业务逻辑处理。这样 controller 就不用再去关心业务逻辑，只需要拿到数据刷界面就行了。

```
- (void)loadData {
    self.manager = [[Cinema_ListManager alloc] init];
    __weak typeof(self) weakSelf = self;
    [self.manager getCinema_ListWithBlcok:^(NSArray *result) {
        __strong typeof(weakSelf)strongSelf = weakSelf;
        strongSelf.dataArray = result;
        [strongSelf.tableView reloadData];
    }];
}
```

这么做的好处，一方面避免了 ViewController 处理过多业务逻辑导致，viewController 过于臃肿。另一方面，方便多人协作开发。比如有些东西，界面在另一个人那，而业务逻辑却在我这。这样我就可以在 manager 里面写好业务逻辑，然后告诉另一个人去调用我的 manager 里的某个方法就可以了。他不用关心是怎么实现的，只需要根据 block 回调的数据来处理界面就行了。

5. Dispathcher 查询一个或多个 ViewResolver 视图解析器，找到 ModelAndView 对象指定的视图对象。



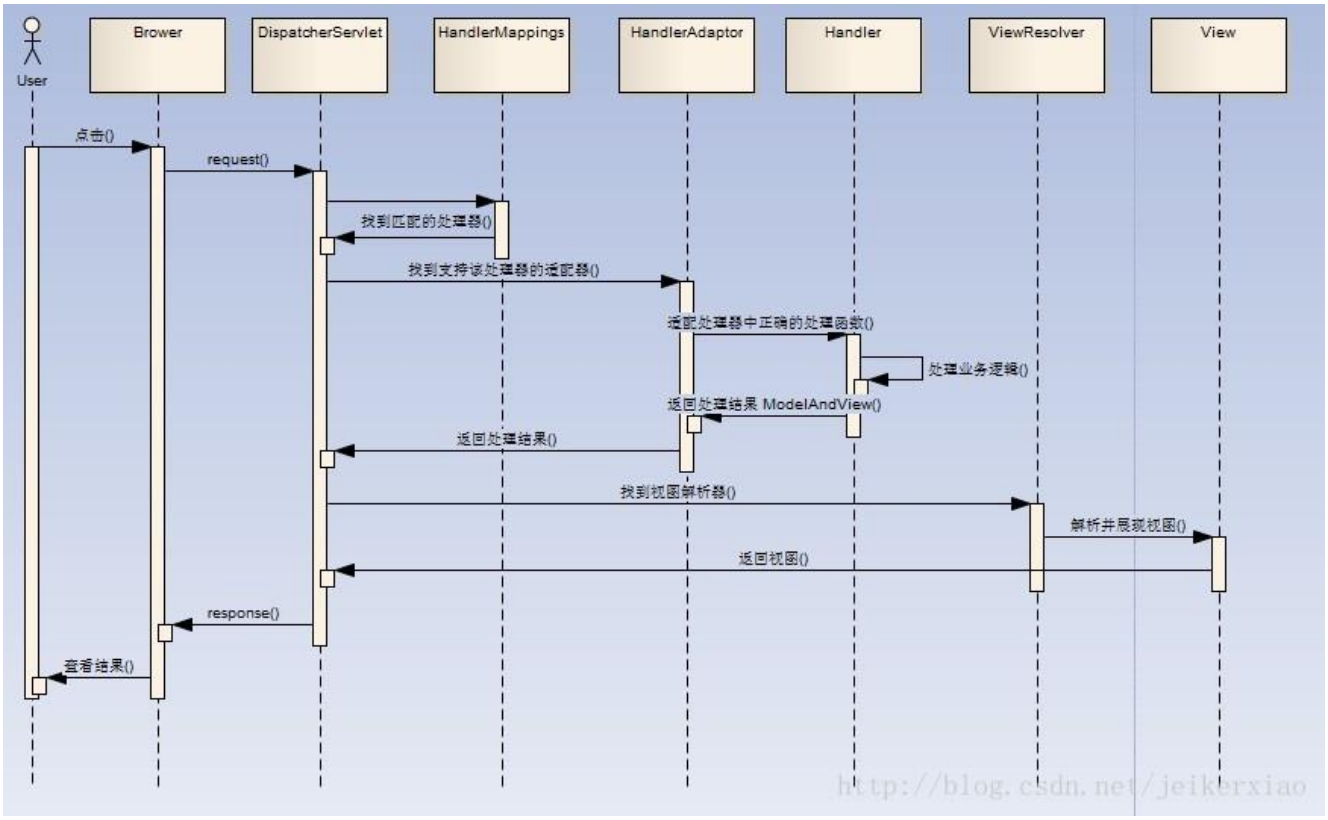

```
<!-- 定义 JSP 视图解析器-->
<bean                                                    id="jspViewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/"></property>
    <property name="suffix" value=".jsp"></property>
    <property name="order" value="1" />
</bean>

<bean                                                    id="freemarkerViewResolver"
class="com.founder.web.commom.springmvc.view.ExtFreeMarkerViewResolver">
    <property name="contentType" value="text/html; charset=UTF-8"/>
    <property name="exposeRequestAttributes" value="false"/>
    <property name="exposeSessionAttributes" value="false"/>
    <property name="exposeSpringMacroHelpers" value="true"/>
    <property name="cacheUnresolved" value="false"/>
    <property name="order" value="0" />
</bean>
```

如果某个解析器没有找到合适的视图，Spring 会在上下文中寻找是否配置了其它的解析器。 如果有，它会继续进行解析，否则，Srping 会抛出一个 Exception。要记住，当一个视图解析器找不到合适的视图时，它可能 返回 null 值。 但是，不是每个解析器都这么做。这是因为，在某些情况下，解析器可能无法侦测出符合要求的视图是否存在。 比如，InternalResourceViewResolver 在内部调用了 RequestDispatcher。 请求分发是检查一个 JSP 文件是否存在的唯一方法，不幸的是，这个方法只能用一次。 同样的问题在 VelocityViewResolver 和其它解析器中

也有。 当使用这些解析器时，最好仔细阅读它们的 Javadoc，看看需要的解析器是否无法发现不存在的视图。 这个问题产生的副作用是，如果 InternalResourceViewResolver 解析器没有放在链的末端，InternalResourceViewResolver 后面的那些解析器根本得不到使用， 因为 InternalResourceViewResolver 总是返回一个视图！

6. 视图对象负责渲染返回给客户端。




```
public void render(Map<String, ?> model, HttpServletRequest
request, HttpServletResponse response) throws Exception {
        if (logger.isTraceEnabled()) {
            logger.trace("Rendering view with name '" +
this.beanName + "' with model " + model +
                " and static attributes " +
this.staticAttributes);
        }
        Map<String, Object> mergedModel =
createMergedOutputModel(model, request, response);
        prepareResponse(request, response);
        renderMergedOutputModel(mergedModel,
getRequestToExpose(request), response);
    }
```

在进行视图渲染时，会获取到上一节构造的 view 和 model 对象作为参数传递给 renderMergedOutputModel,不同 view 的子类，做具体的渲染操作。如 InternalResourceView 就是进行 redirect 或者 forword 操作,这属于 servlet 的范畴。

不做详细解释。

SpringMVC 的理解

MVC 是什么就不用我多说了.对于现有较成熟的 Model-View-Control(MVC)框架而言,其注意的主要问题无外乎下面这些:

Model:

模型应该包含由视图显示的数据.在 J2EE Web 应用中,数据通常应该由普通的 javabean 组成.一旦一个控制器选择了视图,模型就要包含视图相应的数据.模型本身不应该进一步的访问数据,也不应该和业务对象相联系.

模型要解决的问题包括:

- 封装要显示的数据
- 我不认为模型要依赖于特定的框架
- 不一定非得是 javabean

View:

视图负责显示出模型包含的信息,视图不必了解控制器或是底层业务对象的具体实现

视图要解决的问题包括:

- 在显示给定数据模型的情况下显示内容
- 不应该包含有业务逻辑
- 可能需要执行显示逻辑,比如颜色交替的显示某个数组的各行
- 视图最好不处理验证的错误,数据的验证应该在由其他组件完成
- 视图不应该处理参数,参数应该交由控制器集中处理

Control:

控制器就好像 MVC 里的中枢神经,它也许会需要一些助手来帮助它比如解析视图,解析参数等.控制器可以访问到业务对象或者是它的代理是很重要的,比如 Struts 里的 Action.

控制器要解决的问题包括:

- 检查和抽取请求参数
- 调用业务对象,传递从请求中获取的参数
- 创建模型,视图讲显示对应的模型
- 选择一个合适的视图发送给客户端
- 控制器有时不会只有一个