

12 | 唯有套路得人心：谈谈Java EE的那些模式

2019-10-07 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 18:50 大小 15.09M



你好，我是四火。

本章我们以 MVC 架构为核心，已经介绍了很多设计模式，今天我们将进一步泛化，谈论更多的 Java EE 模式。这些模式，就是我们在搭建全栈架构、设计的工作过程中，不断总结和应用的“套路”。

背景和概念

我相信很多人都接触过面向对象模式，可是，模式是个通用词，面向对象只是其中的一个分支而已。事实上，我们本章的重点 MVC 本身就是一种典型的模式，介绍过的 CQRS 是模式，学习过的 AOP、IoC，这些其实也都是模式。

因此，和其它领域的技术相比，作为全栈工程师的我们，更有机会接触到各种模式。这些模式可以帮助我们在设计开发工作中拓宽思路，使用精巧的代码结构来解决实际问题。

说到这里，你可能会问，为什么这次谈论模式的时候，要使用 Java EE 这个编程语言前缀？模式不是应该和语言无关吗？

一点都没错，模式就是和语言无关的，但是，诞生模式最多的温床，就是 Java 语言。

世界上没有任何一门语言，像 Java 一样，几乎一直被黑，但是生态圈一直在壮大，且在工业界具备如此统治力。**很多人说，Java 是一门平庸的语言，这可能没错，但是它对于底层细节的封装和语言本身的难度做到了很好的平衡**，它不一定会有多精巧、多出彩，但是新手也可以顺利完成工作，且不容易写出破坏性强、其他人难以接手的代码，这对于要规模、要量产的工业界来说，简直是超级福音。

使用 Java 的人可以快速上手，也可以把精力专注在高层的架构和设计上面，这就是为什么使用 Java 的人往往对模式特别敏感的原因。

当然，语言本身的江湖地位也和生态圈密切相关，更先进、更合理的语言一直在出现，但要把整个语言推翻另起炉灶，其难度可想而知，毕竟一门语言还涉及到社区、厂商、开源库、标准等等。

在互联网的战场上，我们一直能看到类似的例子，比如在前端领域 JavaScript 就是一个相对“草率”，有着诸多缺陷的语言，在它之后有许多更先进的语言尝试把它替代（比如 Google 强推的 [Dart](#)），但是这件事情是极其困难的。

那么，什么是 Java EE，为什么是 Java EE？

Java EE，全称为 Java Platform Enterprise Edition，即 Java 平台企业版，是 Java 的三大平台之一，另两大是 Java SE（标准版）和 Java ME（微型版）。企业市场对软件的需求和大众市场是完全不同的，尤其是在互联网的早些时候，对吞吐量、数据规模和服务质量等都有着更高级别的要求，而且企业花钱多，因而带来的回报也高得多。

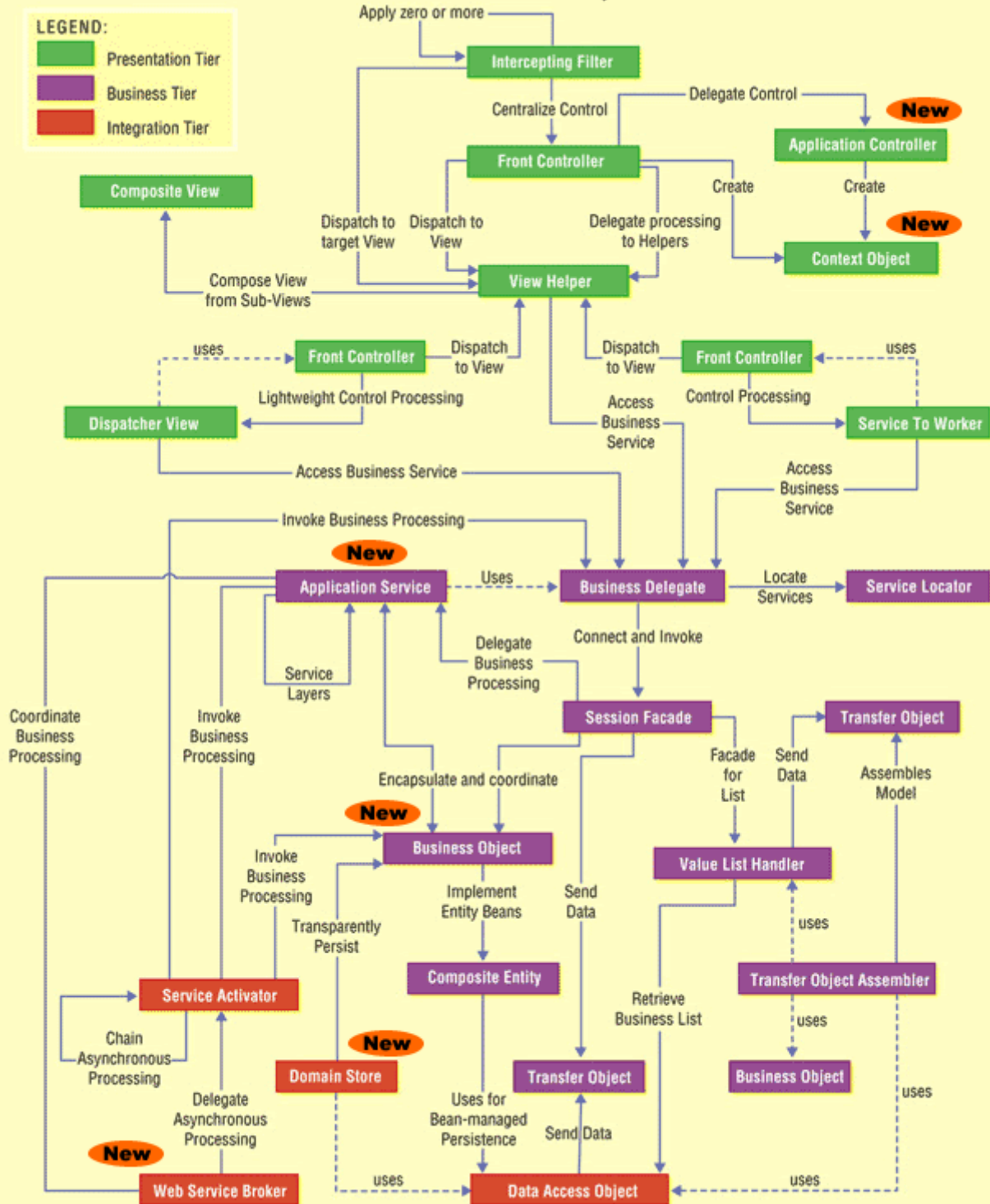
但如今这个特点已经有了变化，但是从当时开始逐步确立下来的企业级规范和技术标准，直到现在还在广泛使用，并不断发展。它使得 Java EE 平台孕育了比其它语言和平台更多的

软件架构和设计思想，而这些优秀的思想，以及通用的“套路”，在这个过程中不断被程序员总结成“模式”。

概览

Java EE 的模式涉及的面非常广泛，下图是来自经典的 [Core J2EE Patterns: Best Practices and Design Strategies](#) 一书，对我们从宏观上理解 Java EE 模式有一定的指导意义。但是请不要以为这就是一个完整的 Java EE 的模式列表，它只是列出了在当时比较常见的一些而已。

Core J2EE Patterns, 2nd Edition



(c) 2003 corej2eepatterns.com. All Rights Reserved.

从图中我们可以看到，这些“核心模式”大致分为呈现层（Presentation Tier，绿色部分）、业务层（Business Tier，紫色部分）和集成层（Integration Tier，红色部分）三大部分，模式之间有的通过实线箭头连接，表示着不同模式之间的单向关联关系，有的通过虚线箭头连接，表示着模式之间“使用包含”的依赖关系。

这里面的内容其实有很多在本章已经涉及到了，比如 Front Controller 和 Business Object，但是，我还想补充和细化其中的两个模式，它们在网站开发的项目中非常常用：

Intercepting Filter 和 Data Access Object。


拦截过滤器

拦截过滤器 (Intercepting Filter) 正如图中的 “Apply zero or more” 和 Servlet 规范所述一样，应当具备一个链式结构。这个链式结构中的每个过滤器，互相之间应当是一个互不依赖的松耦合关系，以便于组合容易。这个过滤器链条，出现的位置通常在控制器 Front Controller 之前，在还没有进行到 Servlet 的 URL 映射前，请求需要先通过它的过滤逻辑。

Tomcat 中配置过滤器

还记得我们在 [第 10 讲] 的加餐中使用 Servlet、JSP 和 JavaBean 实现的简单 MVC 系统吗？现在，让我们来动动手，添加一个基于 URL 映射的过滤器。


首先，打开 `${CATALINA_HOME}/webapps/ROOT/WEB-INF/web.xml`，在我们原本的 BookServlet 配置前，添加如下内容：

 复制代码

```
1 <filter>
2   <filter-name>AuthFilter</filter-name>
3   <filter-class>AuthFilter</filter-class>
4 </filter>
5 <filter-mapping>
6   <filter-name>AuthFilter</filter-name>
7   <url-pattern>/*</url-pattern>
8 </filter-mapping>
9
10 <filter>
11   <filter-name>BookFilter</filter-name>
12   <filter-class>BookFilter</filter-class>
13 </filter>
14 <filter-mapping>
15   <filter-name>BookFilter</filter-name>
16   <url-pattern>/books/*</url-pattern>
17 </filter-mapping>
```

你看，为了显示过滤器链的效果，我们配置了两个过滤器，第一个 AuthFilter 用来对所有的请求实施权限控制，因此 URL 使用 `/*` 匹配所有请求；第二个 BookFilter 我们希望它只对访问图书的请求实施权限控制。


现在建立 AuthFilter, 创建 \${CATALINA_HOME}/webapps/ROOT/WEB-INF/classes/AuthFilter.java, 写入:

 复制代码

```
1 import javax.servlet.*;
2 import java.util.logging.Logger;
3 import java.io.IOException;
4
5 public class AuthFilter implements Filter {
6     private Logger logger = Logger.getLogger(AuthFilter.class.getName());
7     public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
8         logger.info("Check permission...");
9         chain.doFilter(request, response);
10    }
11 }
```

这个用于鉴权的过滤器, 现在只打印日志, 未来我们可以加入真正的鉴权逻辑。

接着建立 BookFilter, 创建 \${CATALINA_HOME}/webapps/ROOT/WEB-INF/classes/BookFilter.java, 写入:


 复制代码

```
1 import javax.servlet.*;
2 import java.io.IOException;
3 import java.util.logging.Logger;
4 import java.util.concurrent.atomic.AtomicInteger;
5
6 public class BookFilter implements Filter {
7     private Logger logger = Logger.getLogger(BookFilter.class.getName());
8     private AtomicInteger count = new AtomicInteger();
9     public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
10         logger.info("Add book accessing count...");
11         int current = this.count.incrementAndGet();
12         request.setAttribute("count", current);
13         chain.doFilter(request, response);
14     }
15 }
```

在这个过滤器中, 我们先打印了日志, 接着创建了一个计数器, 使用 AtomicInteger 而不是 int 这个原语类型的目的是为了正确处理在多线程情况下并发计数器的情形, 再把当前对


books 请求的计数放到 request 中。

编译一下：

 复制代码


```
1 javac AuthFilter.java BookFilter.java -classpath ${CATALINA_HOME}/lib/servlet-api.jar
```

我们再回到曾经建立的 `${CATALINA_HOME}/webapps/ROOT/book.jsp`，在页面尾部添加一行输出计数器的计数：

 复制代码


```
1 Count: ${count}
```

现在启动 Tomcat：

 复制代码


```
1 catalina run
```

最后在浏览器中访问 <http://localhost:8080/books?category=art>，你将看到类似这样的输出，并且每刷新一次页面，这个计数就加 1。

 复制代码

```
1 Category name: art, date: 2019-8-11 Count: 1
```

再回到控制台，你应该能看到类似这样的日志，从中可见过滤器的调用顺序：

 复制代码


```
1 11-Aug-2019 11:08:50.131 INFO [http-nio-8080-exec-1] AuthFilter.doFilter Check permissi
2 11-Aug-2019 11:08:50.132 INFO [http-nio-8080-exec-1] BookFilter.doFilter Add book acces:
```

好，动手实践暂时就先到这里。就如同上面的例子这样，过滤器也是可以配置映射关系的，并且，在过滤器中，我们可以实现一组不同类型请求的处理所共有的逻辑。学到这里，不知道你有没有联想到一个相关的，且在这一讲之前我们才学过的模式？对，它就是 AOP，过滤器本质上就是面向切面编程这种模式的一种子模式。

Struts 的拦截器


Struts 提供了拦截器（Interceptor）这样功能更加强大的组件，对于一些常见的功能，它已经预置了数十种常见的拦截器，比如异常、参数验证、文件上传和国际化支持等等，既包括预处理（Action 执行之前），也包括后处理（Action 执行之后）的拦截逻辑，只需要配置使用即可。

举例来说，如果定义了这样一个的拦截器栈，它包含了两个拦截器，一个是异常拦截器，一个是校验拦截器，并且配置了 ping 方法不需要经过拦截器的校验，这两个拦截器组合成为 commonInterceptorStack 这个拦截器栈：

 复制代码

```
1 <interceptor-stack name="commonInterceptorStack">
2   <interceptor-ref name="exception"/>
3   <interceptor-ref name="validation">
4     <param name="excludeMethods">ping</param>
5   </interceptor-ref>
6 </interceptor-stack>
```

配置完毕后就可以使用了，对于一个控制器层的 bookAction，我们规定请求必须经过一个 alias 拦截器，和刚才定义的 commonInterceptorStack 拦截器栈：

 复制代码

```
1 <action name="bookAction" class="BookAction">
2   <interceptor-ref name="alias"/>
3   <interceptor-ref name="commonInterceptorStack"/>
4 </action>
```

数据访问对象

我们在 [\[第 08 讲\]](#) 中介绍持久层框架的时候，已经谈到了 DAO（Data Access Object），今天让我们进一步学习一下。

DAO 本质上是能够为某种特定数据持久化的机制提供抽象结构的对象。虽然我们谈论 DAO 基本上是默认这里的数据持久化的介质就是数据库，但需要明确的是，实际上并没有这样的约束。换句话说，DAO 可以把数据持久化到数据库中，但也可以持久化到文件里，甚至会以网络请求的方式把数据持久化到某个远程服务中去。

数据访问对象最大的好处依然是我们反复强调的“解耦”，业务代码不需要关心数据是怎样持久化的。在测试其上方的 Service 层的时候，只要把实际的 DAO 替换成“桩代码”，就可以不实际执行持久化逻辑而完成测试；如果哪一天希望更换 DAO 的实现，例如把关系数据库存储改为更一般的键值存储，其上方的 Service 层不修改逻辑就可以实现。

但事物都有两面性，DAO 也不是完美的，比如说，**多加一层就会从额外的抽象层次上带来软件的复杂性，它经常和“抽象泄露（Leaky Abstraction）”这样的现象联系起来。**

这里是说，理想状况下，程序员只需要关心“某一抽象层之上”的逻辑和调用，这也是我们分层的一大好处。可是，现实总是和理想有距离的，一旦抽象之下的部分出错，程序员很可能必须去了解和深入这部分的内容，这就违背了抽象分层的初衷，但是在很多情况下这是不可避免的，这也是整个软件体系日渐复杂，我们需要学习的内容越来越多的原因之一。

总结思考

今天我们了解了 Java EE 的各种模式，并且重点学习了拦截过滤器这个模式。模式的学习有一个特点，在理论学习的基础上，我们需要反复地实践强化，以及反复地思考。可以说，实践和思考这二者缺一不可。如果只有实践而没有思考，就没有办法灵活地将理论应用在复杂的实际项目中；如果只有思考而没有实践，那么到实际动手的时候还是很难顺利地实施想法。

对于今天的内容，留两个问题：

我们介绍了 DAO 层的两面性，那么，在你经历的项目中，你是怎样访问数据存储设施（例如文件和数据库）的，能说说吗？

今天我们学到了，通过使用基于 URL 映射的过滤器，是可以给业务代码增加 AOP 的切面逻辑的。那么，为什么我们还需要之前所介绍的，通过匹配代码类和方法的表达式来嵌

入切面逻辑的方式呢？

有道是，技术进程多风雨，唯有套路得人心。回看本章，从 MVC 开始，我们一直都在和“模式”打交道，不知道你是不是认真学习了，是不是收获一些代码设计上的“套路”了呢？

选修课堂：MyBatis vs Hibernate

在 DAO 的设计过程中，我们经常需要处理模型实体对象和关系数据库的表记录之间的双向转换问题，怎样将对象的属性和关联关系映射到数据库表上去？有许多持久化框架都给出了自己的解决办法，今天我就来介绍两种最经典的解决思路，MyBatis 和 Hibernate。


MyBatis 的思路是使用 XML 或注解的方式来配置 ORM，把 SQL 用标签管理起来，但不关心，也不干涉实际 SQL 的书写。

在这种思路下框架轻量，很容易集成，又因为我们可以使用 SQL 所有的特性，可以写存储过程，也可以写 SQL 方言 (Dialect)，所以灵活度相当高。当然，灵活也意味着在具体实现功能的时候，你需要做得更多，不但需要关心模型层、SQL，还需要关心这二者怎样映射起来，具体包括：

请求参数映射，即模型的值怎样映射到 SQL 语句的变参里面；

返回值映射，即怎样将数据库查询的返回记录映射到模型对象。

我们来看一个最简单的 XML 配置片段：

 复制代码

```
1 <mapper namespace="xxx.BookDAO">
2     <insert id="add" parameterType="Book">
3         insert into BOOKS(NAME, DESC) values(#{name}, #{desc})
4     </insert>
5
6     <select id="get" resultType="Book" parameterType="java.lang.String">
7         select * from BOOKS where ID=#{id}
8     </select>
9 </mapper>
```

你看，SQL 原模原样地写在了配置文件里面。对于写入语句，比如这里的 insert，需要显式告知参数类型 Book 对象，接着就可以直接使用 Book 的 name 和 desc 对应的 get 方法来获得具体值并注入 SQL 了。对于简单的对象，默认的映射规则就可以解决问题，反之，也可以在 XML 中定义映射规则。

Hibernate 则是另一种思路，如果你已经习惯于和模型层打交道，那么它就将 SQL 层对你隐藏起来了。换言之，你只需要写模型代码和 HQL（Hibernate Query Language）这种面向对象的查询语言就可以了，至于 SQL 的生成，框架可以帮你完成。

这种方式的一大好处就是具体数据库的透明性，你今天使用的数据库是 MySQL，明天就可以换成 Oracle，并且不用改代码。在分析设计的时候，你只需要做自己习惯的模型驱动编程就可以了。

但值得注意的是，Hibernate 是把双刃剑，有利也有弊。它也带来了很多问题，比如较高的学习曲线，在出现问题的时候，无论是功能问题还是性能问题，它需要更多的知识储备来进行问题的定位和性能的调试。

MyBatis 和 Hibernate 到这就讲解清楚了，再总结延伸一下。

从框架本身的角度来说，Hibernate 提供的特性远比 MyBatis 更丰富、更完整。如果你是一位有着一定 ORM 经验的程序员，那么 Hibernate 很可能会使你的开发效率更高。

可对于一个新项目而言，在技术选型的过程中，如果你的团队中没有非常多的经验丰富的程序员，我通常建议持久层的框架不要去考虑 Hibernate。简单说来，就是因为它的“水比较深”。我相信大多数程序员朋友还是更习惯于实打实地接触 SQL，流程到哪一步，执行了什么语句，该怎么调整，都十分清晰和直接。

扩展阅读

文中提到了 Java EE 平台的一系列标准和技术，维基百科上有一个[简单的列表](#)供参考。

[Core J2EE Patterns: Best Practices and Design Strategies](#) 这本书对于你学习 Java EE 的模式会提供不错的指导性帮助，属于权威之一，但是内容比较抽象，如果你在设计方面有一定追求，它是很好的阅读材料。好几年前我曾经读过纸质的第一版，但这是第二版，已经可以在网上公开阅读。

如果对于文中提到的 Struts 拦截器感兴趣，请看 Struts 官方文档中[对于拦截器的介绍](#)。

文中提到了抽象泄露的概念，如果你感兴趣的话，请阅读 [The Law of Leaky Abstractions](#)，作者 Joel Spolsky 就是那本著名的《软件随想录》的作者。

[MyBatis 的官网](#)，是的，MyBatis 的教程我就推荐官网上的，清晰简洁，而且具备中文版，不需要去找什么第三方的资料了；如果是需要中文的 Hibernate 入门资料，我推荐 W3Cschool 上的 [Hibernate 教程](#)。



全栈工程师修炼指南

从全栈入门到技能实战

熊焱

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 剑走偏锋：面向切面编程

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。