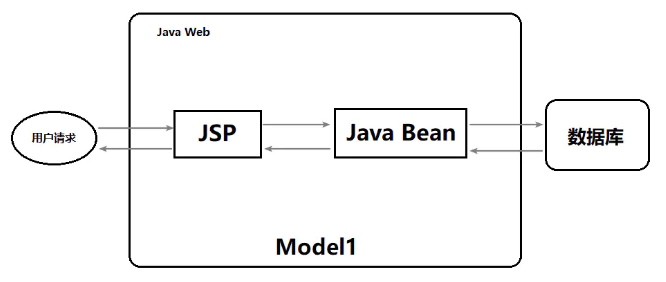
# Spring核心概念

## SSM简介

SSM框架，是Spring + Spring MVC + MyBatis的缩写，Spring依赖注入DI来管理各层的组件，使用面向切面编程AOP管理事物、日志、权限等。SpringMVC代表了Model(模型)View(视图)Controller(控制)接收外部请求,进行分发和处理。Mybatis是基于jdbc的框架,主要用来操作数据库,并且将业务实体和数据表联系起来。

## MVC设计概述

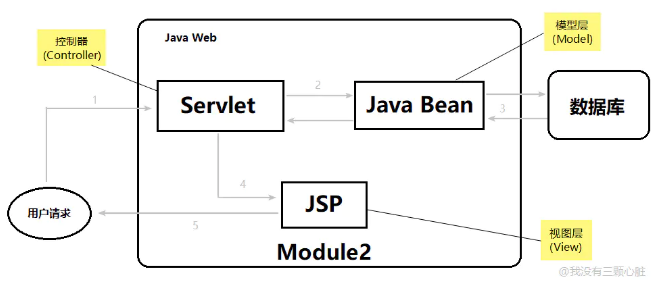
在早期 Java Web 的开发中，统一把显示层、控制层、数据层的操作全部交给 JSP 或者 JavaBean 来进行处理，我们称之为 Model1：



出现的弊端：

* JSP 和 Java Bean 之间严重耦合，Java 代码和 HTML 代码也耦合在了一起
* 要求开发者不仅要掌握 Java ，还要有高超的前端水平
* 前端和后端相互依赖，前端需要等待后端完成，后端也依赖前端完成，才能进行有效的测试
* 代码难以复用

正因为上面的种种弊端，所以很快这种方式就被 Servlet + JSP + Java Bean 所替代了，早期的 MVC 模型（Model2）就像下图这样：



首先用户的请求会到达 Servlet，然后根据请求调用相应的 Java Bean，并把所有的显示结果交给 JSP 去完成，这样的模式我们就称为 MVC 模式。

**M 代表 模型（Model）**

模型是什么呢？ 模型就是数据，就是 dao,bean

**V 代表 视图（View）**

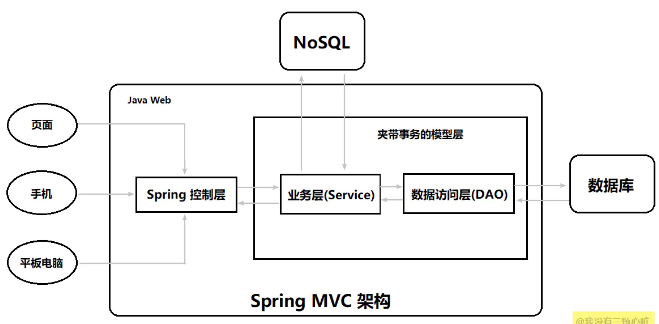
视图是什么呢？ 就是网页, JSP，用来展示模型中的数据

**C 代表 控制器（controller)**

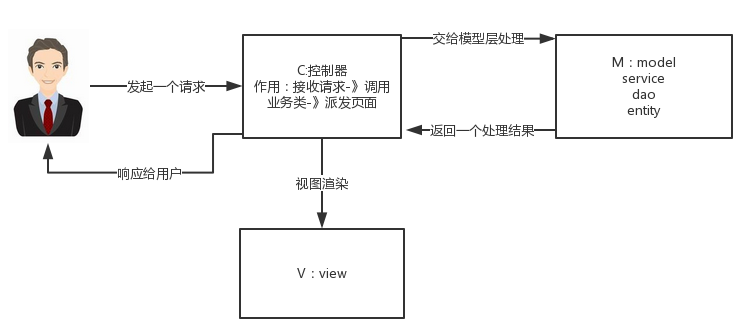
控制器是什么？ 控制器的作用就是把不同的数据(Model)，显示在不同的视图(View)上，Servlet 扮演的就是这样的角色。

## Spring MVC 的架构

为解决持久层中一直未处理好的数据库事务的编程，又为了迎合 NoSQL 的强势崛起，Spring MVC 给出了方案：



MVC原理图：



## 企业级应用

企业级系统的特点：

* 大规模：用户数量多、数据规模大、功能众多
* 性能和安全要求高
* 业务复杂
* 灵活应变

Java技术的应对方式：



Spring是一个当前Java EE轻量级框架，目的是使现有技术更加易用。Spring的诞生是为了满足企业级系统的一些需求，那么，Spring究竟带来了那些好处呢？

引入Spring之后，Spring的依赖注入可以统一管理和生成javabean，哪有需要调用就往哪注入，这种方式大大降低了开发难度，降低了代码的耦合度，给后期的维护也带来了方便。

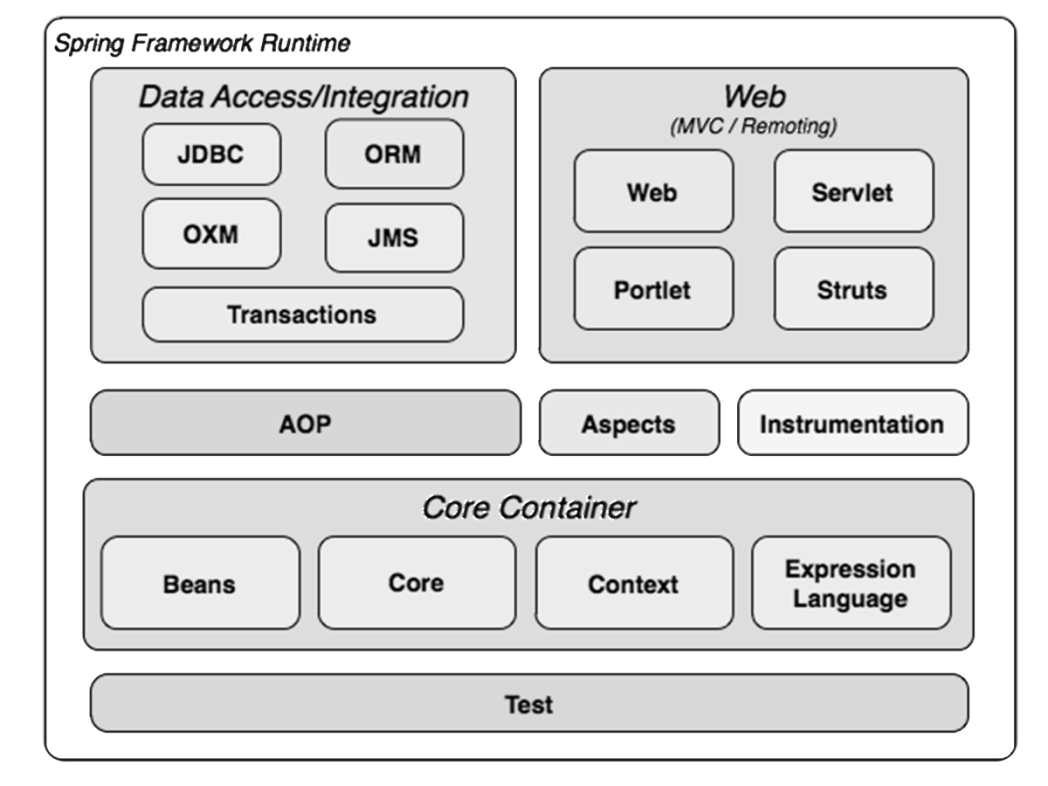
同时spring的AOP还能将系统中那些类似于日志管理，事务等分布性比较强，但又必须有的代码集中生成，无需开发人员关注，提高工作效率。

Spring的主要内容：

* IoC容器
* AOP实现
* 数据访问支持：简化JDBC/ORM框架（Hibernate、Mybatis）、声明式事务。
* Web集成

## Spring的体系结构

Spring一共有十几个组件，但是真正的核心组件只有几个。



从这个图中我们可以看出Spring框架的核心组件只有三个：Core、Context和Beans。他们构建起了整个Spring的骨骼架构，没有他们就不可能有AOP、Web等上层的特性功能。上面这些是Spring特性功能。

其中比较重要的几个包：

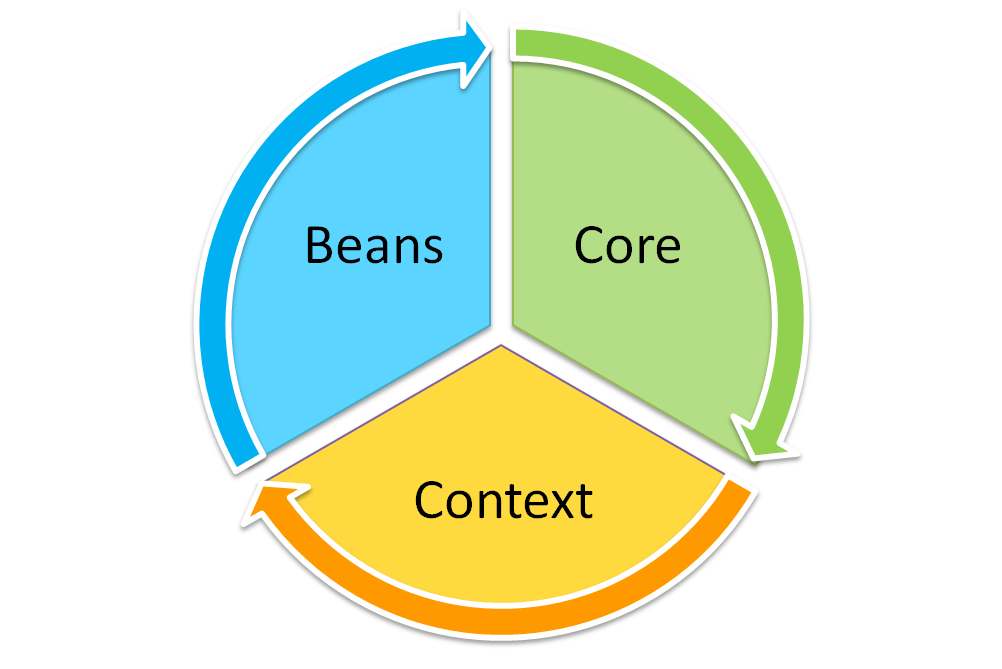
* AOP包(主要提供面向切面编程的实现)；
* Web(主要提供了Web应用开发的支持及针对Web应用的MVC思想实现) 、ORM（为Hibernate、Mybatis这类持久化框架提供支持）；
* 还有Spring MVC（这个是它自带的一个web视图层，可以替代到Sturts2）。

Spring最最核心的就是AOP和下面Spring核心包，也是我们学习的重点。

## Spring核心技术

Spring 两大核心技术：

* 控制反转(IoC：Inversion of Control )，也叫依赖注入(DI：Dependency Injection )。
* 面向切面编程(AOP：Aspect Oriented Programming)。



Spring三个核心组件（Core、Context、Beans）。如果再在他们三个中选一个核心来，那就非Beans莫属了，为何这样说？其实Spring就是面向Bean的编程（BOP，Bean Oriented Programming），Bean才是Spring中的真正主角。

Spring就是面向Bean的编程，在Spring中所有对象都可以看成一个Bean。

Bean在Spring 中作用就像Object对OOP的意义一样，没有对象的概念就没有面向对象编程，Spring中没有Bean也就没有Spring存在意义。就像一次演出舞台都准备好了但是却没有演员一样。为什么要Bean这种角色？

思考下，你会发现原来Spring解决了一个非常关键的问题，他可以让你把对象之间的关系转而使用配置文件来管理，也就是他的**依赖注入**机制，而这个注入关系在一个叫Ioc的容器中管理。Spring正是通过把对象包装在Bean中从而达到对这些对象管理以及一系列额外操作的目的。

Spring的这些核心组件如何协同工作？

前面说Bean是Spring中的关键因素，那Context和Core又有何作用？前面把Bean比作一场演出中的**演员**的话，那么Context就是这场演出的**舞台背景**，而Core就是应该是演出的**道具**了。只有他们在一起才能具备演出一场好戏的最基本条件。当然有最基本的条件还不能使这场演出脱颖而出，还要他表演的节目足够的精彩，这些节目就是Spring提供的特色功能了。

Spring是怎么管理这些Bean的呢？

Spring把所有的Bean及它们之间的依赖关系以配置文件的方式组装起来，在一个叫IoC（Inversion of Control）的容器中进行管理，这也就是Spring的核心设计思想之一依赖注入机制，Spring的另一个核心设计思想叫做AOP，这两个概念在后续会有详细讲解。

一个Context基本上把spring具有的核心功能都包裹起来了，那么这就是spring框架运行需要的环境，也就是常说的上下文。任何一个框架运行都通过一个类来进行描述它执行时的环境，ServletContext也是一样，就是Servlet环境信息。

## Spring的优点

总结起来，Spring作为一个开源的轻量级的IOC和AOP容器框架，具有以下优点：

1、低侵入式设计：非入侵式设计，基于Spring开发的应用一般不依赖于Spring的类

2、独立于各种应用服务器，真正实现：一次编写，到处运行。

3、Spring的依赖注入特性使Bean与Bean之间的依赖关系变的完全透明，降低了耦合度：使用SpringIOC容器，将对象之间的依赖关系交给Spring，降低组件之间的耦合性，让我们更专注于应用逻辑。

4、它的面向切面编程（AOP）特性允许将一些通用任务如安全、事务、日志等进行集中式处理。

5、并且它还提供了与第三方持久层框架的良好整合，并简化了底层数据库访问。

6、高度的开放性（可以和Struts2、Hibernate、MyBatis、CXF等很多主流第三方框架无缝整合）。

## 控制反转（依赖注入）

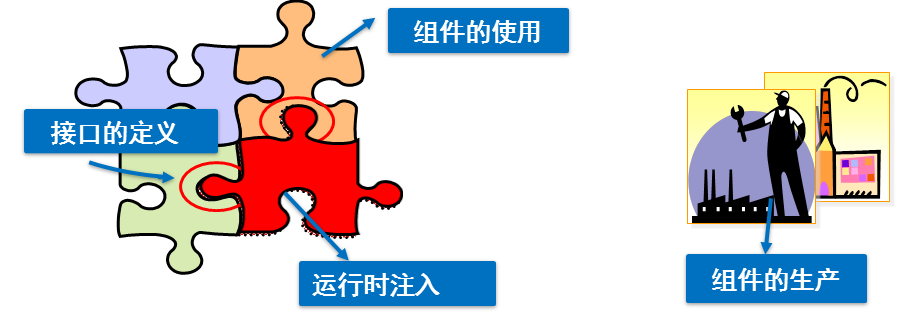
### 什么是控制反转

将组件对象的控制权从代码本身转移到外部容器。

* 组件化的思想：分离关注点，使用接口，不再关注实现。



* 依赖的注入：将组件的构建和使用分开。

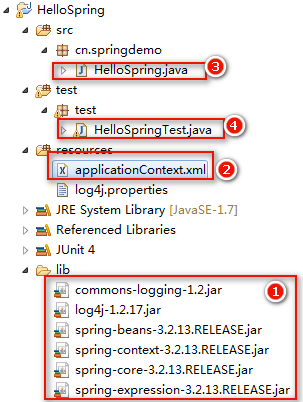


当某个角色（比如一个java实例，调用者）需要另一个角色（另一个java实例，被调用者）的协助时，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。但是在spring里，创建被调用者的工作不再由调用者来完成。因此被称为控制反转；创建被调用者实例的工作通常由spring容器来完成，然后注入调用者，因此也称为依赖注入。这样给程序带来很大的灵活性，这样也实现了我们的接口和实现的分离。

简而言之也就是说我们要获得一个对象，不由我们开发者自己创建，而是由我们的容器来注入给我们的程序来使用。

### 示例：使用Spring实现“控制反转”。

新建Java项目，目录结构如下：



Spring配置文件applicationContext.xml代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

<!-- 通过bean元素声明需要Spring创建的实例。该实例的类型通过class属性指定，并通过id属性为该实例指定一个名称，以便在程序中使用 -->

<bean id="helloSpring" class="cn.springdemo.HelloSpring">

<!-- property元素用来为实例的属性赋值,此处实际是调用setWho()方法实现赋值操作-->

<property name="who">

<!-- 此处将字符串"Spring"赋值给who属性 -->

<value>Spring</value>

</property>

</bean>

</beans>

类HelloSpring.java代码如下：

package cn.springdemo;

/\*\*

\* 第一个Spring,输出"Hello,Spring!"。

\*/

public class **HelloSpring** {

// 定义who属性，该属性的值将通过Spring框架进行设置

private String who = null;

/\*\*

\* 定义打印方法，输出一句完整的问候。

\*/

public void print() {

System.out.println("Hello," + this.getWho() + "!");

}

public String getWho() {

return who;

}

public void setWho(String who) {

this.who = who;

}

}

测试类HelloSpringTest.java代码如下：

package test;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.springdemo.HelloSpring;

public class HelloSpringTest {

@Test

public void helloSpring() {

// 通过ClassPathXmlApplicationContext实例化Spring的上下文

ApplicationContext context = new ClassPathXmlApplicationContext(

"applicationContext.xml");

// 通过ApplicationContext的getBean()方法，根据id来获取bean的实例

HelloSpring helloSpring = (HelloSpring) context.getBean("helloSpring");

// 执行print()方法

helloSpring.print();

}

}

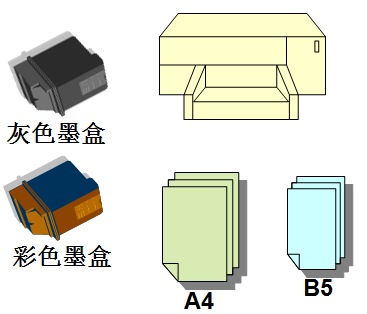
输出结果：Hello,Spring!

### 示例：组装打印机

1. 需求

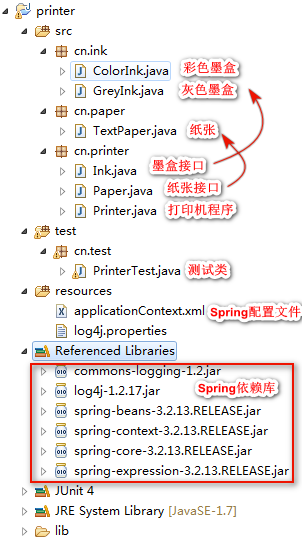
开发一个打印机程序，可灵活配置使用彩色墨盒或灰色墨盒，可灵活配置打印页面的纸张大小。

1. 开发步骤
2. 定义墨盒和纸张的接口标准。
3. 使用接口标准开发打印机。
4. 组装打印机。
5. 运行打印机。



1. 编码

新建Java项目，目录结构如下：



1. 定义组件接口

墨盒接口代码：

/\*\*

\* 墨盒接口。

\*/

public interface **Ink** {

/\*\*

\* 定义打印采用的颜色的方法。

\*

\* @param r 红色值

\* @param g 绿色值

\* @param b 蓝色值

\* @return 返回打印采用的颜色

\*/

public String getColor(int r, int g, int b);

}

纸张接口代码：

/\*\*

\* 纸张接口。

\*/

public interface **Paper** {

public static final String newline = "\r\n";

/\*\*

\* 输出一个字符到纸张。

\*/

public void putInChar(char c);

/\*\*

\* 得到输出到纸张上的内容。

\*/

public String getContent();

}

1. 使用接口开发打印机（面向接口编程）

墨盒接口的实现类——彩色墨盒，代码：

package cn.ink;

import java.awt.Color;

import cn.printer.Ink;

/\*\*

\* 彩色墨盒。ColorInk实现Ink接口。

\*/

public class **ColorInk** implements Ink {

// 打印采用彩色

public String getColor(int r, int g, int b) {

Color color = new Color(r, g, b);

return "#" + Integer.toHexString(color.getRGB()).substring(2);

}

}

墨盒接口的实现类——灰色墨盒，代码：

import java.awt.Color;

import cn.printer.Ink;

/\*\*

\* 灰色墨盒。GreyInk实现Ink接口。

\*/

public class **GreyInk** implements Ink {

// 打印采用灰色

public String getColor(int r, int g, int b) {

int c = (r + g + b) / 3;

Color color = new Color(c, c, c);

return "#" + Integer.toHexString(color.getRGB()).substring(2);

}

}

纸张接口的实现类——文本纸张，代码：

package cn.paper;

import cn.printer.Paper;

/\*\*

\* 文本打印纸张实现。TextPaper实现Paper接口。

\*/

public class **TextPaper** implements Paper {

// 每行字符数

private int charPerLine = 16;

// 每页行数

private int linePerPage = 5;

// 纸张中内容

private String content = "";

// 当前横向位置，从0到charPerLine-1

private int posX = 0;

// 当前行数，从0到linePerPage-1

private int posY = 0;

// 当前页数

private int posP = 1;

public String getContent() {

String ret = this.content;

// 补齐本页空行，并显示页码

if (!(posX == 0 && posY == 0)) {

int count = linePerPage - posY;

for (int i = 0; i < count; ++i) {

ret += Paper.newline;

}

ret += "== 第" + posP + "页 ==";

}

return ret;

}

public void putInChar(char c) {

content += c;

++posX;

// 判断是否换行

if (posX == charPerLine) {

content += Paper.newline;

posX = 0;

++posY;

}

// 判断是否翻页

if (posY == linePerPage) {

content += "== 第" + posP + "页 ==";

content += Paper.newline + Paper.newline;

posY = 0;

++posP;

}

}

// setter方法，用于属性注入

public void setCharPerLine(int charPerLine) {

this.charPerLine = charPerLine;

}

// setter方法，用于属性注入

public void setLinePerPage(int linePerPage) {

this.linePerPage = linePerPage;

}

}

1. 组装打印机

打印机程序类，代码：

package cn.printer;

/\*\*

\* 打印机程序。

\*/

public class **Printer** {

// 面向接口编程，而不是具体的实现类

private Ink ink = null;

// 面向接口编程，而不是具体的实现类

private Paper paper = null;

/\*\*

\* 设值注入所需的setter方法。

\* @param ink 传入墨盒参数

\*/

public void setInk(Ink ink) {

this.ink = ink;

}

/\*\*

\* 设值注入所需的setter方法。

\* @param paper 传入纸张参数

\*/

public void setPaper(Paper paper) {

this.paper = paper;

}

/\*\*

\* 打印机打印方法。

\* @param str 传入打印内容

\*/

public void print(String str) {

// 输出颜色标记

System.out.println("使用" + ink.getColor(255, 200, 0) + "颜色打印:\n");

// 逐字符输出到纸张

for (int i = 0; i < str.length(); ++i) {

paper.putInChar(str.charAt(i));

}

// 将纸张的内容输出

System.out.print(paper.getContent());

}

}

1. 注入组装打印机时所依赖的对象

相当于把创建和组装的需求通过配置文件告诉Spring，由Spring负责实施，而不是通过硬编码实现（需要提前为Printer类的ink和paper属性增加setter方法，Spring是通过调用setter方法实现的注入）。

编辑applicationContext.xml文件，使用Spring创建Ink、Paper和Printer的实例并进行组装。

Spring配置文件applicationContext.xml代码：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

<!-- 定义彩色墨盒bean，该bean的id是colorInk，class指定该bean实例的实现类 -->

<bean id="colorInk" class="cn.ink.ColorInk" />

<!-- 定义灰色墨盒bean，该bean的id是greyInk，class指定该bean实例的实现类 -->

<bean id="greyInk" class="cn.ink.GreyInk" />

<!-- 定义A4纸张bean，该bean的id是a4Paper，class指定该bean实例的实现类 -->

<bean id="a4Paper" class="cn.paper.TextPaper">

<!-- property元素用来指定需要容器注入的属性，charPerLine需要容器注入， TextPaper类必须拥有setCharPerLine()方法。-->

<!-- 注入每行字符数 -->

<property name="charPerLine" value="10" />

<!-- property元素用来指定需要容器注入的属性，linePerPage需要容器注入，TextPaper类必须拥有setLinePerPage()方法。 -->

<!-- 注入每页行数 -->

<property name="linePerPage" value="8" />

</bean>

<!-- 定义B5纸张bean，该bean的id是b5Paper，class指定该bean实例的实现类 -->

<bean id="b5Paper" class="cn.paper.TextPaper">

<!-- property元素用来指定需要容器注入的属性，charPerLine需要容器注入， TextPaper类必须拥有setCharPerLine()方法。注入每行字符数 -->

<property name="charPerLine" value="6" />

<!-- property元素用来指定需要容器注入的属性，linePerPage需要容器注入， TextPaper类必须拥有setLinePerPage()方法。注入每页行数 -->

<property name="linePerPage" value="5" />

</bean>

<!-- **组装打印机**。定义打印机bean，该bean的id是printer， class指定该bean实例的实现类 -->

<bean id="printer" class="cn.printer.Printer">

<!-- 通过ref属性注入已经定义好的bean -->

<!-- 注入彩色墨盒 -->

<property name="ink" ref="colorInk"></property>

<!-- 注入B5打印纸张 -->

<property name="paper" ref="b5Paper"></property>

</bean>

</beans>

1. 运行打印机

运行打印机的测试类代码：

package cn.test;

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.printer.Printer;

/\*\*

\* 测试运行打印机。

\*/

public class **PrinterTest** {

@Test

public void printerTest() {

ApplicationContext context = new ClassPathXmlApplicationContext(

"applicationContext.xml");

// 通过Printer bean的id来获取Printer实例

Printer printer = (Printer) context.getBean("printer");

String content = "几位轻量级容器的作者曾骄傲地对我说：这些容器非常有"

+ "用，因为它们实现了“控制反转”。这样的说辞让我深感迷惑：控"

+ "制反转是框架所共有的特征，如果仅仅因为使用了控制反转就认为"

+ "这些轻量级容器与众不同，就好像在说“我的轿车是与众不同的，" + "因为它有4个轮子。”";

printer.print(content);

}

}

运行测试类中的printerTest()方法，根据applicationContext.xml的注入不同，输出结果不同。

通过使用Spring，可以象更换打印机的墨盒和打印纸一样地更换程序组件。这就是依赖注入带来的魔力！！

## 面向切面编程（AOP）

### 什么是AOP

下面的代码中，除了业务代码之外，还加入了事务控制、记录日志的代码。

public class **UserServiceImpl** implements UserService {

private static final Logger log = Logger.getLogger(UserServiceImpl.class);

public boolean addNewUser(User user) {

log.info(“添加用户 ” + user.getUsername());//记录日志

SqlSession sqlSession = null;

boolean flag = false;

//异常处理

try {

业务代码

sqlSession = MyBatisUtil.createSqlSession();

if (sqlSession.getMapper(UserMapper.class).add(user) > 0)

flag = true;

sqlSession.commit(); //事务控制

} catch (Exception e) {

log.error("添加用户 " + user.getUsername() + "失败", e); //记录日志

sqlSession.rollback(); //事务控制

flag = false;

} finally {

MyBatisUtil.closeSqlSession(sqlSession);

}

return flag;

}

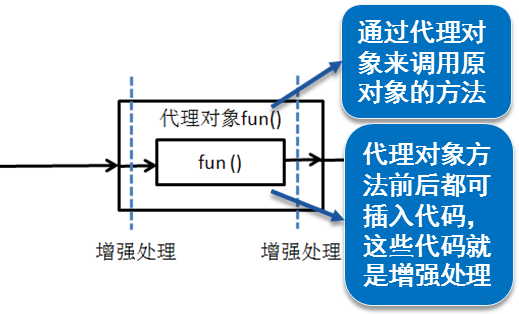
}

我们构建系统的目的是解决业务问题，现在却要维护多方面的事情，怎样才能“专心”做事？

AOP的目标，是让我们可以“专心做事”。

AOP的原理：

* 将复杂的需求分解出不同方面，将散布在系统中的公共功能集中解决。
* 采用代理机制组装起来运行，在不改变原程序的基础上对代码段进行增强处理，增加新的功能。



通过动态代理实现AOP（采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行）。

**AOP是一种思想，它与具体的实现技术无关，任何一种符合AOP的思想的技术实现，都可以看做是AOP的实现。通过java的动态代理机制，就可以很容易实现AOP的思想，实际上Spring的AOP也是建立在Java的代理机制上。**——我们发现AOP实际上是由目标类的代理类实现的。AOP代理其实是由AOP框架动态生成的一个对象，该对象可作为目标对象使用。AOP代理包含了目标对象的全部方法，但是AOP代理中的方法与目标对象的方法存在差异，AOP方法在特定切入点添加了增强处理，并回调了目标对象的方法。

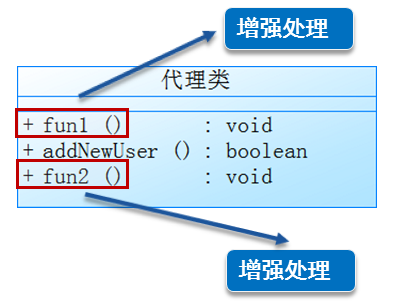
总结下对AOP理解：

业务处理的主要流程就是核心关注点，与之关系不大的部分就是横切关注点。横切关注点的一个特点就是：他们经常发生在核心关注点的多处，而各处基本相似，比如权限认证、日志、事务处理。AOP的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

所谓面向切面编程，是一种通过预编译和运行期动态代理的方式实现在不修改源代码的情况下给程序动态添加功能的技术。

### AOP相关术语

**增强（Advice）：**又翻译成通知，定义了切面是什么以及何时使用，描述了切面要完成的工作和何时需要执行这个工作。是织入到目标类连接点上的一段程序代码。增强包含了用于添加到目标连接点上的一段执行逻辑，又包含了用于定位连接点的方位信息。（所以spring提供的增强接口都是带方位名：BeforeAdvice（表示方法调用前的位置）、AfterReturninAdvice（表示访问返回后的位置）、ThrowAdvice等等，所以只有结合切点和增强两者一起才能确定特定的连接点并实施增强逻辑）。增强又分**前置增强**、**后置增强**、**环绕增强**、**异常抛出增强**、**最终增强**等类型。



Advice直译为“通知”，但这种叫法并不确切，在此处翻译成“增强处理”更便于大家理解。

**切入点（Pointcut）：**Advice定义了切面要发生“故事”和时间，那么切入点就定义了“故事”发生的地点。例如某个类或者方法名，Spring中允许我们使用正则来指定。

**连接点（Joinpoint）：**切入点匹配的执行点称作连接点。如果说切入点是查询条件，那连接点就是被选中的具体的查询结果。程序执行的某个特定位置，程序能够应用增强代码的一个“时机”，比如方法调用或者特定异常抛出。

**切面（Aspect）：**切点和增强组成切面。它包括了横切逻辑的定义，也包括了连接点的定义。Spring AOP就是负责实施切面的框架，它将切面所定义的横切逻辑织入到切面所指定的连接点中。

**代理（Proxy）：**AOP框架创建的对象。一个类被AOP织入增强之后，就产生了一个结果类，它是融合了原类和增强逻辑的代理类。

**目标对象（Target）：**增强逻辑的织入的目标类。

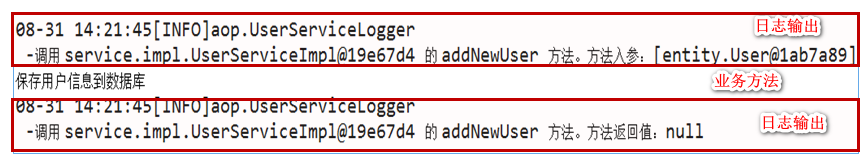
**织入（Weaving）：**将增强添加到目标类具体连接点上的过程。AOP有三种织入的方式：编译期织入、类装载期织入、动态代理织入（spring采用动态代理织入）。

面向切面编程主要关心两个问题，即：在什么位置，执行什么功能。Spring AOP是负责实施切面的框架，即由Spring AOP完成织入工作。

### 示例：使用AOP实现日志功能

1. 需求

在业务方法执行前后添加日志功能。



1. 开发步骤
2. 在项目中添加Spring AOP的jar文件。

* spring-aop-3.2.13.RELEASE.jar
* aopalliance-1.0.jar
* aspectjweaver-1.6.9.jar
* cglib-nodep-2.1.3.jar（已内联在Spring core中）

1. 编写前置增强和后置增强实现日志功能。

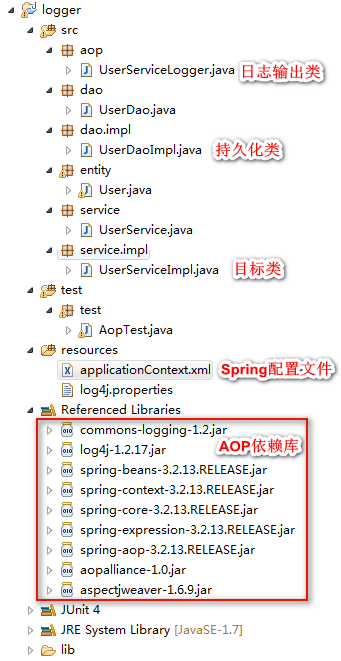
编写目标方法和增强处理。

1. 编写Spring配置文件，对业务方法进行增强处理。

在Spring配置文件中定义切入点，在切入点织入增强处理。

1. 编写代码获取带有增强处理的业务对象。
2. 编码

新建Java项目，目录结构如下：



1. 目标方法

public class UserServiceImpl implements UserService {

//…省略代码

public void addNewUser(User user) {

dao.save(user);

前置增强

}

}

连接点对象

1. 增强处理

public class UserServiceLogger {

private static Logger log=Logger.getLogger(UserServiceLogger.class);

后置增强

public void before(JoinPoint jp) {

log.info("调用 " + jp.getTarget() + " 的 " + jp.getSignature().

getName() + " 方法。方法入参：" + Arrays.toString(jp.getArgs()));

}

连接点方法参数数组

public void afterReturning(JoinPoint jp, Object result) {

log.info("调用 " + jp.getTarget() + " 的 " + jp.getSignature().

getName() + " 方法。方法返回值：" + result);

连接点方法信息

}

目标对象

}

连接点方法返回值

JoinPoint：连接点对象。此处指UserServiceImpl对象。

jp.getArgs()：连接点方法的参数数组。

jp.getTarget()：目标类。

jp.getSignature()：连接点方法信息。

result：连接点方法返回值。

UserServiceLogger类中定义了before( )和afterReturning( )两个方法。我们希望把before()方法作为前置增强使用，即将该方法添加到目标方法之前执行；afterReturning()方法作为后置增强使用，即将该方法添加到目标方法正常返回之后执行。

为了能够在增强方法中获得当前连接点的信息，以便实施相关的判断和处理，可以在增强方法中声明一个JoinPoint类型的参数，Spring会自动注入该实例。通过该实例的getTarget( )方法可以得到被代理的目标对象，getSignature( )方法返回被代理的目标方法，getArgs( )方法返回传递给目标方法的参数数组。对于实现后置增强的afterReturning( )方法，还可以定义一个参数用于接收目标方法的返回值。

1. 定义切入点

<aop:config>

<aop:pointcut id="pointcut"

expression="execution(public void addNewUser(entity.User))"/>

</aop:config>

配置切入点的标签<aop:pointcut>。<aop:pointcut>的expression属性可以配置切入点表达式，切入点表达式支持模糊匹配。表达式匹配规则举例：

public \* addNewUser(entity.User)： “\*”表示匹配所有类型的返回值。

public void \*(entity.User)： “\*”表示匹配所有方法名。

public void addNewUser(..)： “..”表示匹配所有参数个数和类型。

\* com.service.\*.\*(..)：匹配com.service包下所有类的所有方法。

\* com.service..\*.\*(..)：匹配com.service包及其子包下所有类的所有方法。

1. 织入增强处理

织入：在切入点插入增强处理。

将增强处理和切入点结合在一起，在切入点处插入增强处理，即可完成"织入"。

<aop:config>

增强处理对象

<aop:pointcut id="pointcut"

expression="execution(public void addNewUser(entity.User))" />

被织入的方法

<aop:aspect ref="userServiceLogger">

切入点

<aop:before method="before"

前置增强处理

pointcut-ref="pointcut"></aop:before>

<aop:after-returning method="afterReturning"

后置增强处理

pointcut-ref="pointcut" returning="result"/>

</aop:aspect>

</aop:config>

向result变量中注入返回值

在<aop:config>中使用<aop:aspect>引用包含增强方法的Bean。然后分别通过<aop:before>和<aop:after-returning>将方法声明为前置增强和后置增强，在<aop:after-returning>中可以通过returning属性指定需要注入返回值的属性名。方法的JoinPoint类型参数无须特殊处理，Spring会自动为其注入连接点实例。

很明显UserService的addNewUser()方法可以和切入点pointcut相匹配，Spring会生成代理对象在它执行前后分别调用before()和afterReturning()方法，这样就完成了日志输出。

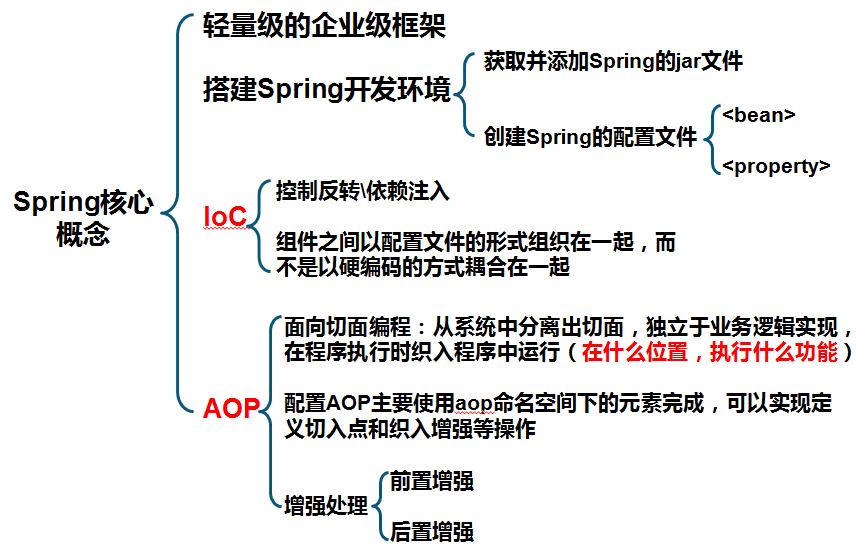
## 常用增强处理类型

|  |  |
| --- | --- |
| **增强处理类型** | **特 点** |
| Before | 前置增强处理，在目标方法前织入增强处理 |
| AfterReturning | 后置增强处理，在目标方法正常执行（不出现异常）后织入增强处理 |
| AfterThrowing | 异常增强处理，在目标方法抛出异常后织入增强处理 |
| After | 最终增强处理，不论方法是否抛出异常，都会在目标方法最后织入增强处理 |
| Around | 环绕增强处理，在目标方法的前后都可以织入增强处理 |

## AOP常用配置元素

|  |  |
| --- | --- |
| AOP配置元素 | 描 述 |
| <aop:config> | AOP配置的顶层元素，大多数的<aop:\*>元素必须包含在<aop:config>元素内 |
| <aop:pointcut> | 定义切点 |
| <aop:aspect> | 定义切面 |
| <aop:after> | 定义最终增强（不管被通知的方法是否执行成功） |
| <aop:after-returning> | 定义after-returning增强 |
| <aop:after-throwing> | 定义after-throwing增强 |
| <aop:around> | 定义环绕增强 |
| <aop:before> | 定义前置增强 |
| <aop:aspectj-autoproxy> | 启动@AspectJ注解驱动的切面 |

## 总结



# 使用注解实现IOC和AOP

## 使用注解实现IOC

注解方式可以将Bean的定义信息和Bean实现类结合在一起，Spring提供的注解有：

* + @Component：实现Bean组件的定义。
  + @Repository ：用于标注DAO类。
  + @Service ：用于标注业务类。
  + @Controller ：用于标注控制器类。

例如以下代码，指明userDao是UserDaoImpl的实例名称：

@Repository("userDao")

public class UserDaoImpl implements UserDao {

…

}

上述与在XML配置文件中编写

<bean id="userDao" class="dao.impl.UserDaoImpl" />

等效。

使用@Autowired注解实现Bean的自动装配，默认按类型匹配，可以使用@Qualifier指定Bean的名称，对类的成员变量进行标注：

@Service("userService")

public class UserServiceImpl implements UserService {

@Autowired

@Qualifier("userDao")

private UserDao dao; //对类的成员变量进行标注

……

}

也可以对方法的入参进行标注：

@Service("userService")

public class UserServiceImpl implements UserService {

private UserDao dao;

@Autowired

public void setDao((@Qualifier("userDao") UserDao dao) {

this.dao = dao;

}

……

}

配置文件中指定需要扫描的基类包，多个包可用逗号隔开：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="......

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.2.xsd">

<!-- 扫描包中注解标注的类 -->

<context:component-scan base-package="service,dao" />

</beans>

## 示例：使用注解实现IOC

见代码。

## 使用注解实现AOP

需求：使用注解来实现日志切面。

步骤：

1. 使用注解定义前置增强和后置增强实现日志功能。

* @Aspect：定义切面。
* @Before：定义前置增强。
* @AfterReturning：定义后置增强。

1. 编写Spring配置文件，完成切面织入。

<aop:aspectj-autoproxy />：启用对于@AspectJ注解的支持。

定义切面的代码如下：

package aop;

import java.util.Arrays;

import org.apache.log4j.Logger;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.aspectj.lang.annotation.AfterReturning;

import org.aspectj.lang.annotation.Pointcut;

/\*\*

\* 使用注解定义切面

\*/

@Aspect

public class **UserServiceLogger** {

private static final Logger log = Logger.getLogger(UserServiceLogger.class);

@Pointcut("execution(\* service.UserService.\*(..))")

public void pointcut() {

}

@Before("pointcut()")

public void before(JoinPoint jp) {

log.info("调用 " + jp.getTarget() + " 的 " + jp.getSignature().getName()

+ " 方法。方法入参：" + Arrays.toString(jp.getArgs()));

}

@AfterReturning(pointcut = "pointcut()", returning = "returnValue")

public void afterReturning(JoinPoint jp, Object returnValue) {

log.info("调用 " + jp.getTarget() + " 的 " + jp.getSignature().getName()

+ " 方法。方法返回值：" + returnValue);

}

}

配置文件代码：

<context:component-scan base-package="service,dao" />

<bean class="aop.UserServiceLogger"></bean>

<!-- 启用对于@AspectJ注解的支持 -->

<aop:aspectj-autoproxy />

# SpringMVC

## SpringMVC框架特点

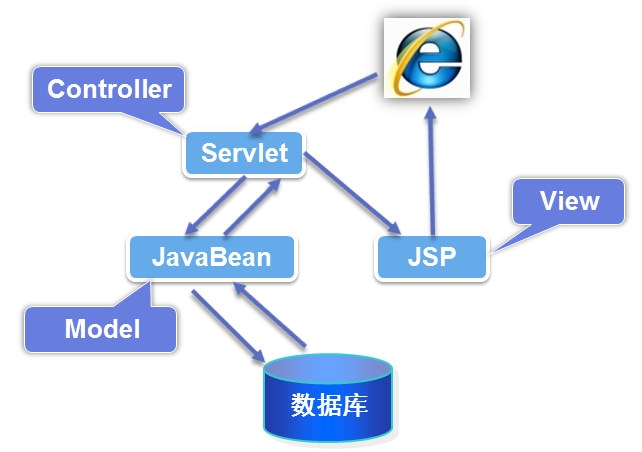
* 清晰地角色划分。
* 灵活的配置功能。
* 提供了大量的控制器接口和实现类。
* 真正做到与View层的实现无关（JSP、Velocity、Xslt等）。
* 国际化支持。
* 面向接口编程。
* Spring提供了Web应用开发的一整套流程，不仅仅是MVC，他们之间可以很方便的结合一起。

## MVC设计模式

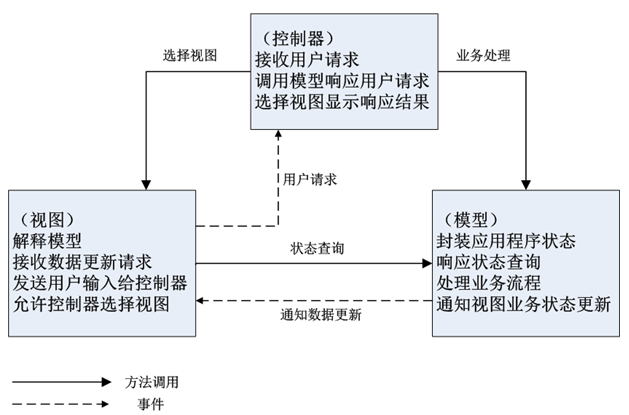
模型（Model）-对应组件：JavaBean。

视图（View）-对应组件：JSP或者HTML文件。

控制器（Controller）-对应组件：Servlet。



MVC处理过程：



MVC 优点：

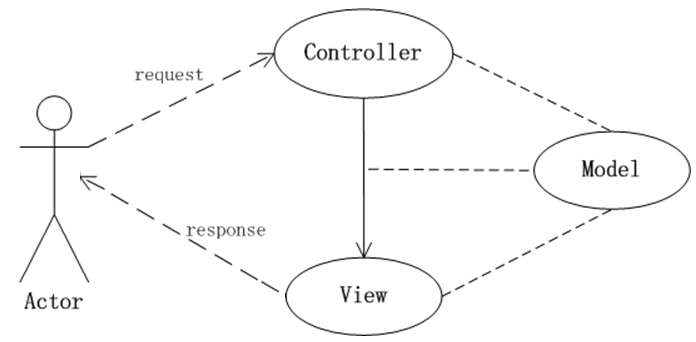
* 多视图共享一个模型，大大提高代码的可重用性。
* MVC三个模块相互独立，松耦合架构。
* 控制器提高了应用程序的灵活性和可配置性。
* 有利于软件工程化管理。

MVC 缺点：

* 原理复杂。
* 增加了系统结构和实现的复杂性。
* 视图对模型数据的低效率访问。

完美的系统架构 = 松耦合+高重用性+高扩展性

## Spring MVC架构



## 示例：第一个SpringMVC程序

Spring MVC框架搭建步骤：

(1) 下载jar文件并导入工程

spring-web-3.2.13.RELEASE.jar

spring-webmvc-3.2.13.RELEASE.jar

(2) 配置文件

在web.xml中配置Servlet，定义DispatcherServlet。

创建Spring MVC的配置文件。

(3) 创建Controller（处理请求的控制器）

(4) 创建View（JSP视图）

(5) 部署运行

控制器代码：

import org.apache.log4j.Logger;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

@Controller

public class IndexController {

private Logger logger = Logger.getLogger(IndexController.class);

// RequestMapping表示用哪个url来对应(此处："/index")

@RequestMapping("/index")

public String index() {

// System.out.println("hello,SpringMVC!");

logger.info("hello,SpringMVC!");

return "index";

}

}

SpringMVC配置文件spring-servlet.xml代码：

<context:component-scan base-package="cn.hyg.controller" />

<mvc:annotation-driven />

<!-- 完成视图的对应 -->

<!-- 对转向页面的路径解析。prefix：前缀， suffix：后缀 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

web.xml代码：

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">

<display-name>springMVC</display-name>

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springmvc-servlet.xml</param-value>

</init-param>

<!-- 标记容器在启动的时候就加载这个servlet(实例化并调用其init()方法)。它的值必须是一个整数，表示servlet应该被载入的顺序 -->

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<context-param>

<param-name>log4jConfigLocation</param-name>

<param-value>classpath:log4j.properties</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.util.Log4jConfigListener

</listener-class>

</listener>

</web-app>

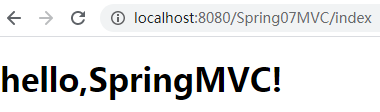
<load-on-startup>1</load-on-startup>这个配置有什么作用呢？作用如下：

1. load-on-startup元素标记容器是否在启动的时候就加载这个servlet(实例化并调用其init()方法)。
2. 它的值必须是一个整数，表示servlet应该被载入的顺序。；
3. 当值为0或者大于0时，表示容器在启动时就加载并初始化这个servlet。
4. 当值小于0或者没有指定时，则表示容器在该Servlet被请求时，才会去加载。
5. 正数的值越小，该Servlet的优先级就越高，应用启动时就优先加载。
6. 当值相同的时候，容器就会自己选择优先加载。

所以，<load-on-startuo>x</load-on-startuo>中x的取值1,2,3,4,5代表的是优先级，而非启动延迟时间。

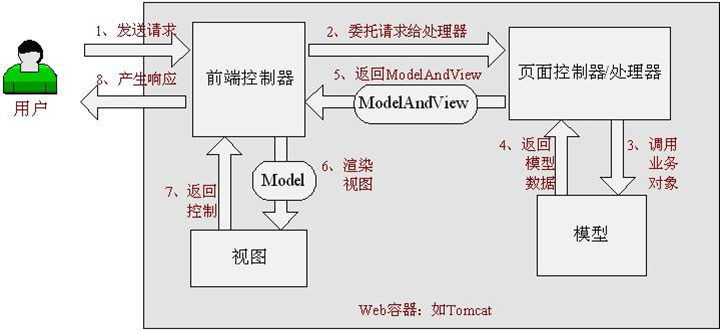
通常大多数Servlet是在用户第一次请求的时候由应用服务器创建并初始化，但<load-on-startup>n</load-on-startup>可以用来改变这种状况，根据自己需要改变加载的优先级！

发布项目到Tomcat，访问地址<http://localhost:8080/Spring07MVC/index>，结果如下：



## SpringMVC体系结构

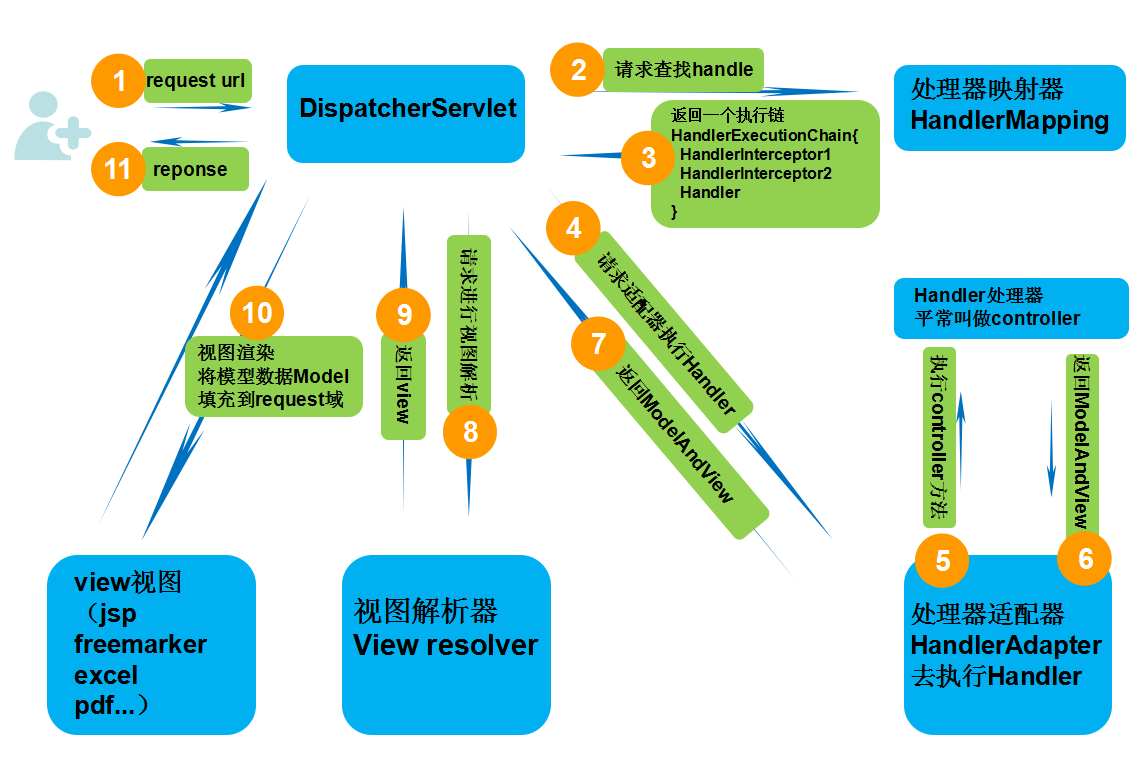
### SpringMVC请求处理流程



具体步骤：

1. 首先用户发送请求到前端控制器，前端控制器根据请求信息（如 URL）来决定选择哪一个页面控制器进行处理并把请求委托给它，即以前的控制器的控制逻辑部分；图中的 1、2 步骤；
2. 页面控制器接收到请求后，进行功能处理，首先需要收集和绑定请求参数到一个对象，这个对象在 Spring Web MVC 中叫命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理完毕后返回一个 ModelAndView（模型数据和逻辑视图名）；图中的 3、4、5 步骤；
3. 前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图进行渲染，并把模型数据传入以便视图渲染；图中的步骤 6、7；
4. 前端控制器再次收回控制权，将响应返回给用户，图中的步骤 8；至此整个结束。

### SpringMVC体系结构



* DispatcherServlet（前端控制器）：Spring MVC最核心的类，在web.xml中配置。
* Handler（处理器）：对应MVC中C(Controller层)，处理实际请求。标注了@RequestMapping的所有方法都可以看作是一个Handler。
* ModelAndView：逻辑视图名、模型对象。

具体步骤：

第一步：发起请求到前端控制器(DispatcherServlet)。

第二步：前端控制器请求HandlerMapping查找 Handler （可以根据xml配置、注解进行查找）。

第三步：处理器映射器HandlerMapping向前端控制器返回Handler，HandlerMapping会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象，多个HandlerInterceptor拦截器对象），通过这种策略模式，很容易添加新的映射策略。

第四步：前端控制器调用处理器适配器去执行Handler。

第五步：处理器适配器HandlerAdapter将会根据适配的结果去执行Handler。

第六步：Handler执行完成给适配器返回ModelAndView。

第七步：处理器适配器向前端控制器返回ModelAndView （ModelAndView是springmvc框架的一个底层对象，包括 Model和view）。

第八步：前端控制器请求视图解析器去进行视图解析 （根据逻辑视图名解析成真正的视图(jsp)），通过这种策略很容易更换其他视图技术，只需要更改视图解析器即可。

第九步：视图解析器向前端控制器返回View。

第十步：前端控制器进行视图渲染 （视图渲染将模型数据(在ModelAndView对象中)填充到request域）。

第十一步：前端控制器向用户响应结果。

下面我们对出现的一些组件进行详细的介绍：

(1) 前端控制器DispatcherServlet（不需要程序员开发）。

作用：接收请求，响应结果，相当于转发器，中央处理器。有了DispatcherServlet减少了其它组件之间的耦合度。

(2) 处理器映射器HandlerMapping（不需要程序员开发）。

作用：根据请求的url查找Handler。

(3) 处理器适配器HandlerAdapter（不需要程序员开发）。

作用：按照特定规则（HandlerAdapter要求的规则）去执行Handler。

(4) 处理器Handler（需要程序员开发）。

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler。

(5) 视图解析器ViewResolver（不需要程序员开发）。

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）

(6) 视图View（需要程序员开发jsp）。

注意：View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf…）

可以总结出：需要我们开发的工作只有处理器 Handler 的编写以及视图，比如JSP页面的编写。可能你还对诸如前端控制器、处理器映射器等等名词不太理解，那么接下来我们对其进行详细的介绍。

接下来就介绍下如何配置和开发相关的组件。

## SpringMVC组件配置（了解即可）

1. 配置前端控制器（DispatcherServlet）

在web项目的web.xml中配置：

<!-- 配置前端控制器DispatcherServlet -->

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!--springmvc.xml 是自己创建的SpringMVC全局配置文件，用contextConfigLocation作为参数名来加载，如果不配置 contextConfigLocation，那么默认加载的是/WEB-INF/servlet名称-servlet.xml，在这里也就是springmvc-servlet.xml 参数多个值使用逗号隔开，如：a.xml,b.xml -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:springmvc.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<!--第一种配置：\*.do,还可以写\*.action等等，表示以.do结尾的或者以.action结尾的URL都由前端控制器DispatcherServlet来解析

第二种配置：/,所有访问的 URL 都由DispatcherServlet来解析，但是这里最好配置静态文件不由DispatcherServlet来解析，需要对静态资源单独处理

错误配置：/\*,注意这里是不能这样配置的，因为如果这样写，最后转发到 jsp 页面的时候，仍然会由DispatcherServlet进行解析，而这时候会找不到对应的Handler，从而报404！！！ -->

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

1. 配置处理器映射器（HandlerMapping）

在 springmvc.xml 文件中配置。通俗来讲就是请求的 URL 怎么能被 SpringMVC 识别，从而去执行我们编写好的 Handler。

(1) 第一种方法

<!-- 配置Handler -->

<bean name="/hello.do" class="com.asiainfo.spring.controller.HelloController" />

<!-- 配置处理器映射器 将bean的name作为url进行查找，需要在配置Handler时指定bean name（就是url）-->

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

这样配置的话，那么请求的 URL，必须为 <http://ip:port/项目名/hello.do>。

(2) 第二种方法

<!-- 配置Handler -->

<bean id="hello" class="com.asiainfo.spring.controller.HelloController" />

<bean id="hello2" class="com.asiainfo.spring.controller.HelloController2" />

<!-- 简单URL配置处理器映射器 -->

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<prop key="/hello.do">hello</prop>

<prop key="/hello2.do">hello2</prop>

</props>

</property>

</bean>

1. 配置视图解析器（ViewResolver）

第一种配置：

<!-- 配置视图解析器 进行jsp解析，默认使用jstl标签，classpath下得有jstl的包-->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" />

如果这样配，那么在 Handler 中返回的必须是路径+jsp页面名称+".jsp"。

第二种配置：

<!--配置视图解析器 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!-- 返回视图页面的前缀 -->

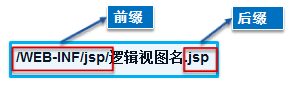
<property name="prefix" value="/WEB-INF/jsp/"></property>

<!-- 返回页面的后缀 -->

<property name="suffix" value=".jsp"></property>

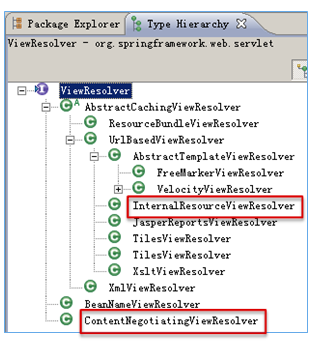
</bean>

如下图：



如果这样配，那么在 Handler 中只需要返回在jsp文件夹下的jsp 页面名就可以了。

视图解析器源码结构：



上面我们讲解了各种配置，可能有人会问这么多配置，万一少配置了一样，那不就不能运行了，那我们能不能不配置呢？答案是肯定的，SpringMVC 给我们提供了一个 DispatcherServlet.properties 文件。系统会首先加载这里面的配置，如果我们没有配置，那么就默认使用这个文件的配置；如果我们配置了，那么就优先使用我们手动配置的。

在 SpringMVC 运行之前，会首先加载 DispatcherServlet.properties 文件里面的内容

①处理器适配器默认：org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter

②处理器映射器默认：org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping

③视图解析器默认：org.springframework.web.servlet.view.InternalResourceViewResolver

详见：

<https://www.cnblogs.com/myitnews/p/11565941.html>

## 参数传递（View to Controller）

### @RequestParam

Controller方法中参数前加@RequestParam可以直接传递参数。

@RequestMapping("/welcome")

public String welcome(@RequestParam String username){

logger.info("welcome, " + username);

return "index";

}

浏览器访问地址：http://localhost:8080/SpringMVC08RequestParam/welcome?username=admin

将成功跳转到index.jsp页面。如下图：



使用required=true保证必须包含参数，required=false为非必须：

@RequestMapping("/welcome")

public String welcome(@RequestParam(value="username",required=true) String username){

logger.info("welcome," + username);

return "index";

}

### @RequestMapping

@RequestMapping可以标注在类定义处，也可以标注在方法定义处。@RequestMapping映射的请求信息必须保证全局唯一。

映射规则：

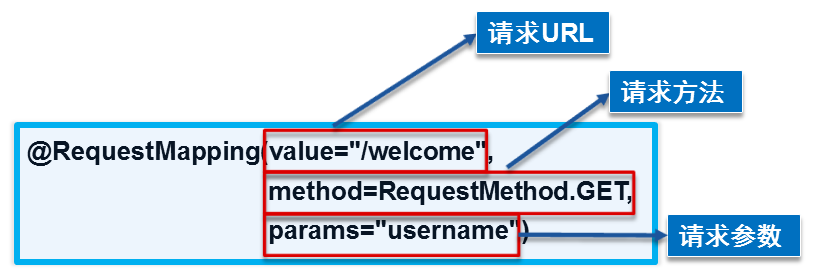
* 通过请求URL进行映射
* 通过请求参数进行映射

若选择方法参数直接入参的话，方法入参名必须与请求中参数名保持一致。

* 通过请求方法进行映射

GET

POST



上图指明：请求地址是“/welcome”，请求方法只能是“GET”，请求参数必须包含“username”。详细代码如下：

@RequestMapping(value="/welcome",method=RequestMethod.GET,params="username")

public String welcome(String username){

保持一致

logger.info("welcome, " + username);

return "index";

}

若选择方法参数直接入参的话，方法入参名必须与请求中参数名保持一致。

## ModelAndView（Controller to view）

ModelAndView包含视图信息和模型数据信息。

常用方法：

1. 添加模型数据，相当于将数据添加到request中。

ModelAndView addObject(String attributeName,Object attributeValue);

ModelAndView addAllObjects(Map<String,?> modelMap);

1. 设置视图

void setView(View view);

void setViewName(String viewName);

示例代码：

@RequestMapping("/index1")

public ModelAndView index(String username){

logger.info("welcome! username: " + username);

//返回前端

ModelAndView mView = new ModelAndView();

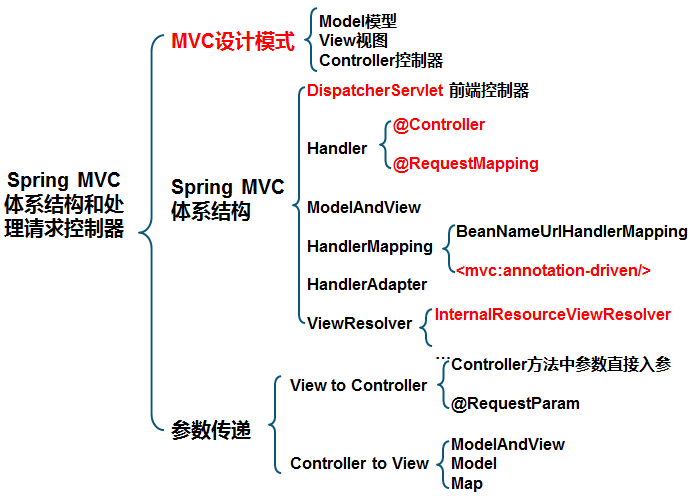
mView.addObject("username", username);

mView.setViewName("index");

return mView;

}

## 总结



## SpringMVC拦截器

### 拦截器与过滤器的区别

（1）过滤器Filter：

依赖于servlet容器。在实现上基于函数回调，可以对几乎所有请求进行过滤，但是缺点是一个过滤器实例只能在容器初始化时调用一次。使用过滤器的目的是用来做一些过滤操作，获取我们想要获取的数据，比如：在过滤器中修改字符编码；在过滤器中修改HttpServletRequest的一些参数，包括：过滤低俗文字、危险字符等。

（2）拦截器Interceptor：

依赖于web框架，在SpringMVC中就是依赖于SpringMVC框架。在实现上基于Java的反射机制，属于面向切面编程（AOP）的一种运用。由于拦截器是基于web框架的调用，因此可以使用Spring的依赖注入进行一些业务操作，同时一个拦截器实例在一个controller生命周期之内可以多次调用。但是缺点是只能对controller请求进行拦截，对其他的一些比如直接访问静态资源的请求则没办法进行拦截处理。拦截器可以获取IOC容器中的各个bean，而过滤器就不行，这点很重要，在拦截器里注入一个service，可以调用业务逻辑。

SpringMVC中使用Interceptor+cookie实现在一定天数之内自动登录：<https://www.zifangsky.cn/700.html>

### 拦截器的使用

1. 新建拦截器类，实现HandlerInterceptor接口，并实现其中定义的方法：

package cn.hyg.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

/\*\*

\* 拦截器类

\*/

public class **MyInterceptor** implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

System.out.println("进入 preHandle 方法..." + request.getRequestURL().toString() + "," + request.getRequestURI());

return true;//若返回false则请求不会向下进行

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {

System.out.println("进入 postHandle 方法..." + request.getRequestURL().toString() + "," + request.getRequestURI());

}

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {

System.out.println("进入 afterCompletion 方法..." + request.getRequestURL().toString() + "," + request.getRequestURI());

}

}

1. SpringMVC配置文件中添加以下配置：

<mvc:interceptors>

<mvc:interceptor>

<!-- 拦截所有的请求，这个必须写在前面 -->

<mvc:mapping path="/\*\*"/>

<!-- 但是排除下面这些，也就是不拦截请求 -->

<mvc:exclude-mapping path="/user/welcome"/>

<mvc:exclude-mapping path="/login.html"/>

<!-- 拦截器类 -->

<bean class="cn.hyg.interceptor.MyInterceptor"/>

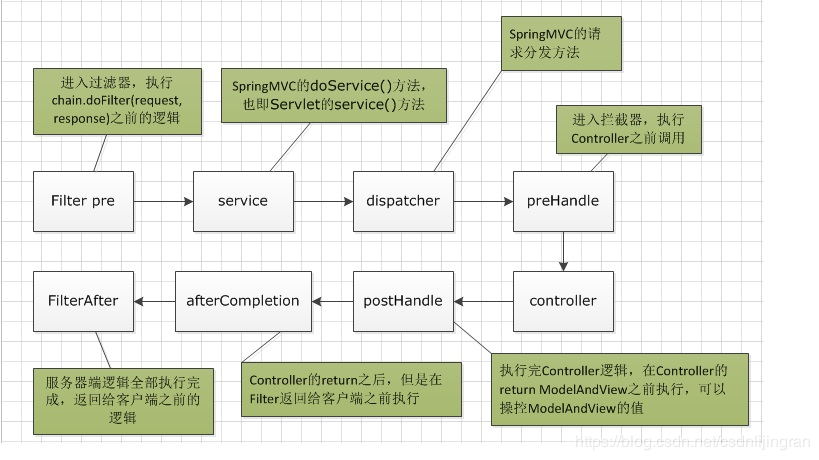
</mvc:interceptor>

</mvc:interceptors>

当访问符合条件的Controller地址时，将先进入到拦截器方法。

### 拦截器各方法执行顺序

SpringMVC的机制是由同一个Servlet（DispatcherServlet）来分发请求给不同的Controller，其实这一步是在Servlet的service()方法中执行的。拦截器是被包裹在过滤器之中的，执行顺序为：过滤器、拦截器、service()方法，dispatcher()方法。

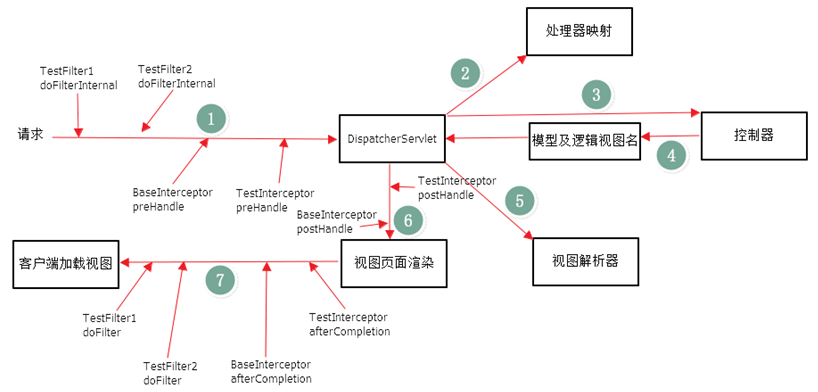


a. preHandle()这个方法是在进入Controller之前执行。

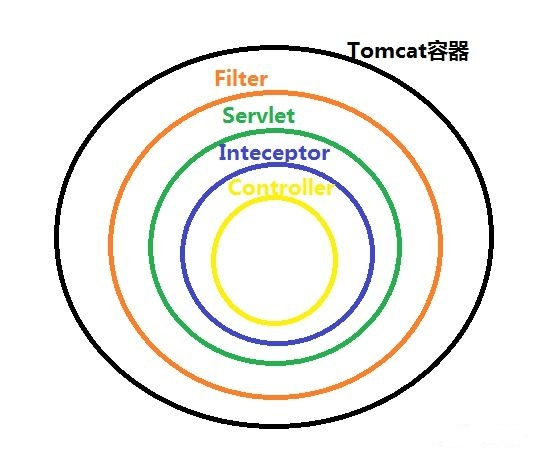
b. postHandle()方法，在return ModelAndView之前进行，可以操控Controller的ModelAndView内容。

c. afterCompletion()方法是在过滤器返回给前端前一步执行，也就是在chain.doFilter(request, response)之后执行。

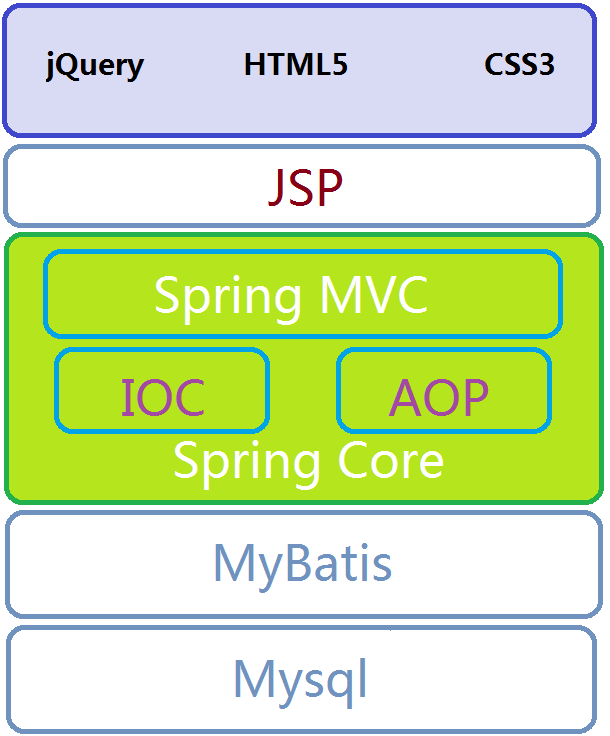
对于整个SpringMVC的执行流程来说，如果同时加上拦截器和过滤器，其最终的执行流程就如下图所示（详见https://www.zifangsky.cn/710.html）：



在实际开发中，如果使用在SpringMVC中使用过滤器（Filter）容易引发XSS攻击的安全问题。详见：<https://www.zifangsky.cn/683.html>



## SSM框架



JavaWeb: sql集成与Java代码里面。

Mybatis：分离sql到配置文件（xml文件）

SSM：

* ORM：MyBatis框架
* Spring Core：基于IoC和AOP的处理方式统一管理所有的JavaBean
* Web框架：Spring MVC

数据存储：MySQL

前端框架：

* JSP
* jQuery框架
* HTML5、CSS3