# Spring 4.x

## 第1篇 基础篇

### chapter1--Spring概述

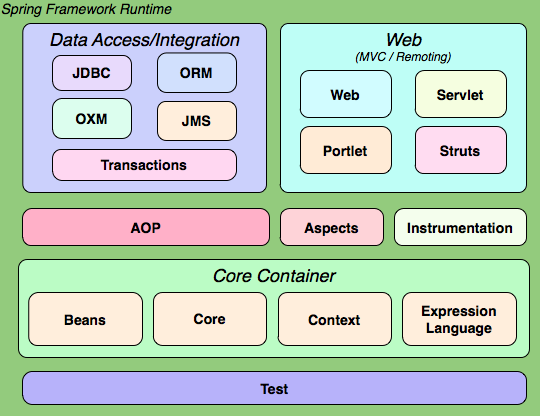
### 1.1 ClassPathXmlApplicationContext

1. **public** **static** **void** main(String[] args) {
2. ApplicationContext ctx = **new** ClassPathXmlApplicationContext("resources/spring/applicationContext-hibernate.xml");
3. DataSource ds = (DataSource) ctx.getBean("dataSource");
4. System.out.println(ds);
5. }

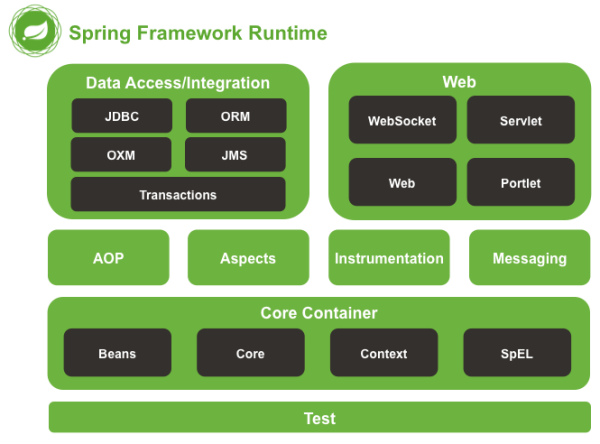
### 1.2体系结构

#### 1.2.1结构图

##### 1.2.1.1 spring3.x

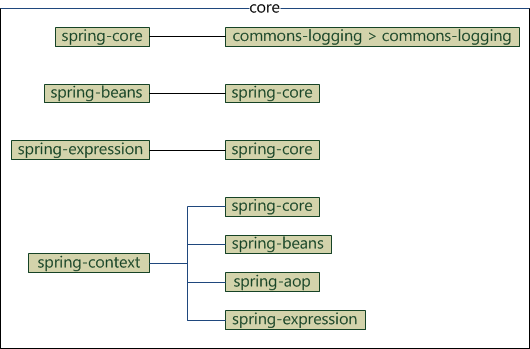


##### 1.2.1.2 spring4.x

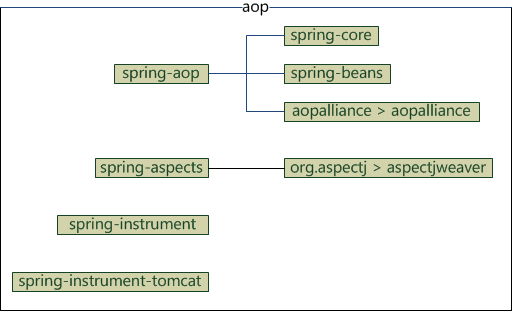


#### 1.2.2依赖关系

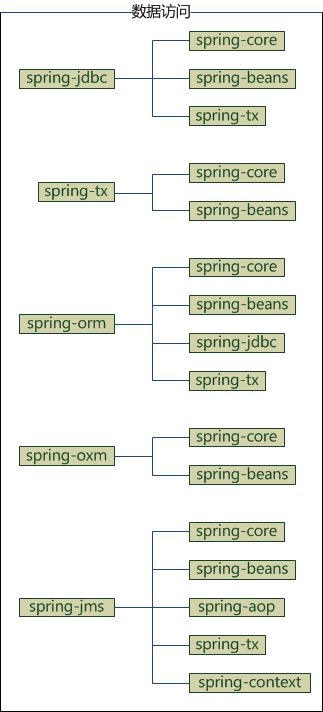
##### 1.2.2.1 Core



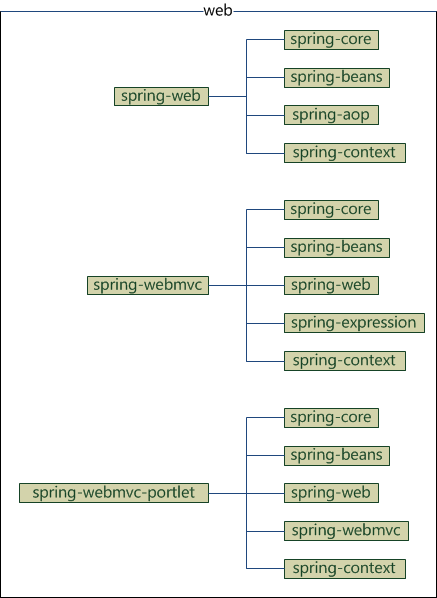
##### 1.2.2.2 aop



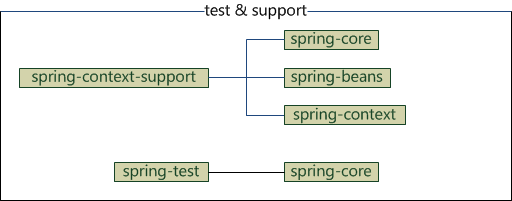
##### 1.2.2.3 data



##### 1.2.2.4 web



##### 1.2.2.5 test & support



### chapter2--快速入门

#### 2.1.web.xml

1. <context-param>
2. <param-name>contextConfigLocation</param-name>
3. <param-value>classpath:smart-context.xml</param-value>
4. </context-param>
5. <listener>
6. <listener-class>
7. org.springframework.web.context.ContextLoaderListener
8. </listener-class>
9. </listener>
10. <servlet>
11. <servlet-name>smart</servlet-name>
12. <servlet-class>
13. org.springframework.web.servlet.DispatcherServlet
14. </servlet-class>
15. <load-on-startup>3</load-on-startup>
16. </servlet>
17. <servlet-mapping>
18. <servlet-name>smart</servlet-name>
19. <url-pattern>\*.html</url-pattern>
20. </servlet-mapping>

#### 2.2.smart-context.xml

##### 2.2.1<!--扫描类包-->

扫描类包，将标注Spring注解的类自动转化Bean，同时完成Bean的注入

1. <context:component-scan base-package="com.smart.dao"/>
2. <context:component-scan base-package="com.smart.service"/>

##### 2.2.2<!-- 配置数据源 -->

1. <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
2. destroy-method="close"
3. p:driverClassName="com.mysql.jdbc.Driver"
4. p:url="jdbc:mysql://localhost:3306/sampledb"
5. p:username="root"
6. p:password="123456" />

##### 2.2.2.3<!-- 配置Jdbc模板 -->

1. <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"
2. p:dataSource-ref="dataSource" />

##### 2.2.4<!-- 配置事务管理器 -->

1. <bean id="transactionManager"
2. class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
3. p:dataSource-ref="dataSource" />

##### 2.2.5<!-- 通过AOP配置提供事务增强，让service包下所有Bean的所有方法拥有事务 -->

1. <aop:config proxy-target-class="true">
2. <aop:pointcut id="serviceMethod"
3. expression="(execution(\* com.smart.service..\*(..))) and (@annotation(org.springframework.transaction.annotation.Transactional))" />
4. <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice" />
5. </aop:config>
6. <tx:advice id="txAdvice" transaction-manager="transactionManager">
7. <tx:attributes>
8. <tx:method name="\*" />
9. </tx:attributes>
10. </tx:advice>

#### 2.3.smart-servlet.xml

##### 2.3.1<!-- 扫描web包，应用Spring的注解 -->

1. <context:component-scan base-package="com.smart.web"/>

##### 2.3.2<!-- 配置视图解析器，将ModelAndView及字符串解析为具体的页面 -->

1. <bean
2. class="org.springframework.web.servlet.view.InternalResourceViewResolver"
3. p:viewClass="org.springframework.web.servlet.view.JstlView"
4. p:prefix="/WEB-INF/jsp/"
5. p:suffix=".jsp" />

### chapter3-Spring Boot

## 第2篇 核心篇

### chapter4--IOC容器

#### 4.1 IOC类型

##### 4.1.1. 构造函数注入

##### 4.1.2.属性注入

##### 4.1.3接口注入

#### 4.2．类装载器 classloader

类装载器就是寻找类的字节码文件并构造出类在JVM内部表示对象的组件，在java中，类装载器把一个类装入JVM需要以下步骤：装载、链接、初始化

JVM装载类时使用 全盘负责委托机制

1. public class ClassLoaderTest {
2. public static void main(String[] args) {
3. ClassLoader loader = ClassLoaderTest.class.getClassLoader();
4. while (loader != null) {
5. System.out.println(loader.getClass().getName());
6. loader = loader.getParent();
7. }
8. System.out.println(loader);
9. }
10. // 先后输出为：
11. // sun.misc.Launcher$AppClassLoader
12. // sun.misc.Launcher$ExtClassLoader
13. // null : 注null就代表是BootStrap类加载器，该加载器是顶级加载器，没有父类加载器

**类加载器之间的父子关系和管辖范围：**

BootStrap -> ExtClassLoader -> AppClassLoader(即通常所说的System ClassLoader)

它们的管辖范围依次是：  
　　BootStrap------>JRE/lib/rt.jar  
　　ExtClassLoader---------->JRE/lib/ext/\*.jar  
　　AppClassLoader---------->CLASSPATH指定的所有jar或目录。

### chapter5--在IOC中装配bean

### 5.0属性注入

1. <bean id="car1" class="com.smart.ditype.Car">
2. <property name=""> </property>
3. </bean>

### 5.1 构造函数注入

##### 5.1.1. 按类型匹配入参

1. <bean id="car1" class="com.smart.ditype.Car">
2. <constructor-arg type="java.lang.String">
3. <value>红旗CA72</value>
4. </constructor-arg>
5. <constructor-arg type="double">
6. <value>20000</value>
7. </constructor-arg>
8. </bean>

##### 5.1.2. 按索引匹配入参

1. <bean id="car2" class="com.smart.ditype.Car">
2. <constructor-arg index="0" value="红旗CA72" /> <constructor-arg
3. index="1" value="中国一汽" /> <constructor-arg index="2" value="20000" />
4. </bean>

##### 5.1.3. 联合使用类型和索引匹配入参

1. <bean id="car3" class="com.smart.ditype.Car">
2. <constructor-arg index="0" type="java.lang.String">
3. <value>红旗CA72</value>
4. </constructor-arg>
5. <constructor-arg index="1" type="java.lang.String">
6. <value>中国一汽</value>
7. </constructor-arg>
8. <constructor-arg index="2" type="int">
9. <value>200</value>
10. </constructor-arg>
11. </bean>
12. <bean id="car4" class="com.smart.ditype.Car">
13. <constructor-arg index="0">
14. <value>红旗CA72</value>
15. </constructor-arg>
16. <constructor-arg index="1">
17. <value>中国一汽</value>
18. </constructor-arg>
19. <constructor-arg index="2" type="int">
20. <value>200</value>
21. </constructor-arg>
22. </bean>

##### 5.1.4. 通过自身类型反射匹配入参

<!--构造函数注入：自动识别入参类型 -->

1. <bean id="boss1" class="com.smart.ditype.Boss">
2. <constructor-arg>
3. <value>John</value>
4. </constructor-arg>
5. <constructor-arg>
6. <ref bean="car" />
7. </constructor-arg>
8. <constructor-arg>
9. <ref bean="office" />
10. </constructor-arg>
11. </bean>

##### 5.1.5. 循环依赖问题

解决：将构造函数注入方式调整为属性注入方式

#### 5.2 工厂方法注入

#### 5.3 注入参数详解

##### 5.3.1. 字面值注入

brand 的value属性包含一个xml特殊符号，因此添加特殊标签cdata

1. <bean id="car" class="com.smart.attr.Car">
2. <property name="maxSpeed">
3. <value>200</value>
4. </property>
5. <property name="brand">
6. <value>
7. <![CDATA[红旗&CA72]]>
8. </value>
9. </property>
10. <property name="price" value="2000.00" />
11. </bean>

或者可以

1. <bean id="car2" class="com.smart.ditype.Car" p:brand="红旗&amp;CA72"
2. p:maxSpeed="200" p:price="100000" />

##### 5.3.2. ref（parent 、bean、local）

1. <bean id="boss1" class="com.smart.ditype.Boss">
2. <constructor-arg>
3. <ref bean="office" />
4. </constructor-arg>
5. </bean>

##### 5.3.3.内部bean

##### 5.3.4.null值

1. <bean id="car" class="com.smart.attr.Car">
2. <property name="brand"><null/></property>
3. </bean>

#### 5.4 集合类型属性

##### 5.4.1 list

1. <bean id="favoriteSet"
2. class="org.springframework.beans.factory.config.SetFactoryBean">
3. <property name="sourceSet">
4. <list>
5. <value>看报</value>
6. <value>赛车</value>
7. <value>高尔夫</value>
8. </list>
9. </property>
10. </bean>

##### 5.4.2 set

1. <bean id="childBoss" parent="parentBoss">
2. <property name="favorites">
3. <set merge="true">
4. <value>爬山</value>
5. <value>游泳</value>
6. </set>
7. </property>
8. </bean>

##### 5.4.3 map

1. <bean id="emails"
2. class="org.springframework.beans.factory.config.MapFactoryBean">
3. <property name="sourceMap">
4. <map>
5. <entry key="AM" value="会见客户" />
6. <entry key="PM" value="公司内部会议" />
7. </map>
8. </property>
9. </bean>

##### 5.4.4 properties

1. <property name="mails">
2. <props>
3. <prop key="jobMail">john-office@smart.com</prop>
4. <prop key="lifeMail">john-life@smart.com</prop>
5. </props>
6. </property>

##### 5.4.5 强类型集合

元素的键为String类型，值为Integer类型，Spring容器在注入强类型集合时会判断元素的类型，将设置值转换为对应的数据库类型。

1. <property name="jobTime">
2. <map>
3. <entry>
4. <key>
5. <value>会见客户</value>
6. </key>
7. <value>124</value>
8. </entry>
9. </map>
10. </property>

##### 5.4.6 集合合并

##### 5.4.7 通过util命名空间配置

1. <beans xmlns="http://www.springframework.org/schema/beans"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xmlns:p="http://www.springframework.org/schema/p"
4. xmlns:util="http://www.springframework.org/schema/util"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
7. http://www.springframework.org/schema/util
8. http://www.springframework.org/schema/util/spring-util-4.0.xsd">

#### 5.5 简化配置方式

##### 5.5.1 字面属性值

表4-3 字面值属性简化配置

|  |  |  |
| --- | --- | --- |
|  | 简化前 | 简化后 |
| 字面值  属性 | <property name="maxSpeed">  <value>200</value>  </property> | <property name="maxSpeed" value="200"/> |
| 构造函数参数 | <constructor-arg type="java.lang.String">  <value>红旗CA72</value>  </constructor-arg> | <constructor-arg type="java.lang.String"  value="红旗CA72" /> |
| 集合元素 | <map>  <entry>  <key><value>AM</value></key>  <value>会见客户</value>  </entry>  </map> | <map>  <entry key="AM" value="会见客户"/>  </map> |

##### 5.5.2 引用对象属性

|  |  |  |
| --- | --- | --- |
|  | 简化前 | 简化后 |
| 字面值  属性 | <property name="car">  <ref bean="car"></ref>  </property> | <property name="car" ref="car"/> |
| 构造函数参数 | <constructor-arg>  <ref bean="car"/>  </constructor-arg> | <constructor-arg ref="car"/> |
| 集合元素 | <map>  <entry>  <key><ref bean="keyBean"/></key>  <ref bean="valueBean"/>  </entry>  </map> | <map>  <entry key-ref="keyBean" value-ref="valueBean"/>  </map> |

##### 5.5.3 p命名空间

1. <beans xmlns="http://www.springframework.org/schema/beans"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xmlns:p="http://www.springframework.org/schema/p"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
6. <bean id="car" class="com.smart.ditype.Car"
7. p:brand="红旗&amp;CA72"
8. p:maxSpeed="200"
9. p:price="20000.00"/>
10. <bean id="boss" class="com.smart.ditype.Boss"
11. p:car-ref="car"/>

#### 5.6 自动装配

|  |  |
| --- | --- |
| 自动装配类型 | 说 明 |
| byName | 根据名称进行自动匹配。假设Boss有一个名为car的属性，如果容器中刚好有一个名为car的Bean，Spring就会自动将其装配给Boss的car属性 |
| byType | 根据类型进行自动匹配。假设Boss有一个Car类型的属性，如果容器中刚好有一个Car类型的Bean，Spring就会自动将其装配给Boss的这个属性 |
| constructor | 与ByType类似，只不过它是针对构造函数注入而言的，如果Boss有一个构造函数，构造函数包含一个Car类型的入参，如果容器中有一个Car类型的Bean，则Spring将自动把这个Bean作为Boss构造函数的入参，如果容器中没有找到和构造函数入参匹配类型的Bean，Spring将抛出异常 |
| autodetect | 根据Bean的自省机制决定采用byType还是constructor进行自动装配：如果Bean提供了默认的构造函数，则采用byType；否则采用constructor |

#### 5.7方法注入

##### 5.7.1 lookup方法注入

##### 5.7.2 方法替换

#### 5.8 <bean> 之间关系

##### 5.8.1继承

##### 5.8.2依赖

##### 5.8.3依赖

#### 5.9 整合多个配置文件

1. <import resource="classpath:com/smart/impt/beans1.xml"/>

#### 5.10 bean 作用域

|  |  |
| --- | --- |
| 类型 | 说明 |
| singleton | <property name="maxSpeed">  <value> 200</value>  </property> |
| protopy | 1 |
| request |  |
| session |  |
| globalsession |  |
|  |  |

1. <bean id="car" class="com.smart.auto.Car" scope="singleton">
2. <property name="brand" value="红旗CA72"/>
3. <property name="price" value="2000"/>
4. </bean>

#### 5.11 基于注解的配置

##### 5.11.1 使用注解定义的Bean

@Component

@Repository

@Service

@Controller

##### 5.11.2 扫描注解定义的Bean

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
7. http://www.springframework.org/schema/context
8. http://www.springframework.org/schema/context/spring-context-4.0.xsd">
9. <context:component-scan base-package="com.smart.anno">
10. <context:include-filter type="aspectj" expression="com.smart.anno.\*Plugin+"/>
11. <context:exclude-filter type="aspectj" expression="com.smart..\*Controller+"/>
12. </context:component-scan>
13. </beans>

##### 5.11.3 自动装配Bean

###### 5.11.3.1 Autowired

1. @Autowired(requird=false)

###### 5.11.3.2 Qualifier

1. @Qualifier（"UserDao"）限定bean的名称

###### 5.11.3.3多个入参的情况

1. @Autowired
2. public void init(@Qualifier("userDao")UserDao userDao,LogDao logDao){
3. System.out.println("multi param inject");
4. this.userDao = userDao;
5. this.logDao =logDao;
6. }

###### 5.11.3.4 对集合进行标注

Spring会将容器中匹配集合元素类型的所有bean都注进来

1. @Autowired(required=false)
2. private List<Plugin> plugins;
3. @Autowired
4. private Map<String,Plugin> pluginMaps;

###### 5.11.3.5 延迟加载@Lazy

延迟加载必须同时标注在属性及目标Bean上，如①、②处

1. @Lazy①
2. @repository
3. public class LogDao{
4. }
5. @Service
6. public class Service{
7. @Lazy②
8. @Autowired
9. public void setLogDao(LogDao logDao){}
10. }

###### 5.11.3.6 标准注解支持@Resource

@resource要求提供一个bean的名称的属性，如果属性为空，则自动采用标注处的变量名或方法名作为bean的名称

1. @Resource（"car"）

###### 5.11.3.7 作用范围@Scope

1. @Scope("prototype")
2. 5.11.3.7 作用范围@Scope

###### 5.11.3.7 生命过程方法

1. @PostConstruct
2. @PreDestory

#### 5.12 基于类的配置

##### 5.12.1 使用java类提供的bean定义信息

1. @Configuration
2. public class AppConf {
3. @Bean
4. public UserDao userDao(){
5. return new UserDao();
6. }
8. @Bean
9. public LogDao logDao(){
10. return new LogDao();
11. }
13. @Bean
14. public LogonService logonService(){
15. LogonService logonService = new LogonService();
16. logonService.setLogDao(logDao());
17. logonService.setUserDao(userDao());
18. return logonService;
19. }
20. }

##### 5.12.2 使用java类的配置信息启动Spring容器

1. public static void main(String[] args) {
2. // 1.通过构造函数加载配置类
3. ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConf.class);
4. // 2.通过编码方式注册配置类
5. AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
6. ctx.register(DaoConfig.class);
7. ctx.register(ServiceConfig.class);
8. ctx.refresh();
9. // 3.通过XML组装@Configuration配置类所提供的配置信息
10. ApplicationContext ctx = new ClassPathXmlApplicationContext("com/smart/conf/beans2.xml");
11. // 4.通过@Configuration组装XML配置所提供的配置信息
12. ApplicationContext ctx = new AnnotationConfigApplicationContext(LogonAppConfig.class);
13. // 5.@Configuration的配置类相互引用
14. ApplicationContext ctx = new AnnotationConfigApplicationContext(DaoConfig.class, ServiceConfig.class);
15. LogonService logonService = ctx.getBean(LogonService.class);
16. System.out.println((logonService.getLogDao() != null));
17. logonService.printHelllo();
18. }

### chapter6--Spring 高级容器主题

#### 6.1 Spring容器技术内幕

##### 6.1.1 内部工作机制

1. // Tell the subclass to refresh the internal bean factory.
2. //1.初始化BeanFactory
3. ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
4. // Allows post-processing of the bean factory in context subclasses.
5. postProcessBeanFactory(beanFactory);
6. // Invoke factory processors registered as beans in the context.
7. //2.调用工厂后处理器
8. invokeBeanFactoryPostProcessors(beanFactory);
9. // Register bean processors that intercept bean creation.
10. //3.注册bean后处理器
11. registerBeanPostProcessors(beanFactory);
12. // Initialize message source for this context.
13. //4.初始化消息源
14. initMessageSource();
15. // Initialize event multicaster for this context.
16. //5.初始化应用上下文广播器
17. initApplicationEventMulticaster();
18. // Initialize other special beans in specific context subclasses.
19. //6.初始化其它特殊的bea：由具体子类实现
20. onRefresh();
21. // Check for listener beans and register them.
22. //7.注册事件监听器
23. registerListeners();
24. // Instantiate all remaining (non-lazy-init) singletons.
25. //8.初始化所有单实例bean，使用懒加载的bean除外
26. finishBeanFactoryInitialization(beanFactory);
27. // Last step: publish corresponding event.
28. //9.完成刷新并发布容器刷新事件
29. finishRefresh();
31. 1.初始化BeanFactory:根据配置文件实例化beanFactory，将Spring配置文件装入容器的bean定义注册表（beanDefinitionRegistry）中，但此时bean还未初始化
32. 2.调用工厂后处理器：根据反射机制从beanDefinitionRegistry中找到所有实现了BeanFactoryPostProcessor接口的bean，并调用其postProccessBeanFactory（）接口中的方法
33. 3.注册bean后处理器:根据反射机制从beanDefinitionRegisty中找到所有实现了beanpostprocessor接口的bean，并将它们注册到容器bean后处理器的注册表中
34. 4.初始化消息源:初始化容器的国际恶化消息资源
35. 5.初始化应用上下文事件广播器。
36. 6.初始化其它特殊的bean:这是一个钩子方法，子类可以借助这个方法执行一些特殊的操作，如abstractRefreshableWebApplicationContext就使用该方法执行初始化themSource的操作
37. 7.注册事件监听器
38. 8.初始化所有的单实例的bean，使用懒加载模式的bean除外:初始化bean后，将它们放入Spring容器的缓存池中
39. 9.发布上下文刷新事件:创建上下文刷新事件，事件广播器负责将这心事件广播到每个注册的事件监听器中

#### 6.2 属性编辑器

#### 6.3 使用外部属性文件

<bean

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"

p:fileEncoding="utf-8">

<property name="locations">

<list>

<value>classpath:com/smart/placeholder/jdbc.properties</value>

</list>

</property>

</bean>

其它属性

1. order
2. placeholderprefix
3. placeholdersuffix

context

1. <context:property-placeholder
2. location="classpath:com/smart/placeholder/jdbc.properties" file-encoding="utf8"/>
3. <bean id="utf8" class="java.lang.String">
4. <constructor-arg value="utf-8"></constructor-arg>
5. </bean>

#### 6.4 使用加密的属性文件

1. <bean class="com.smart.placeholder.EncryptPropertyPlaceholderConfigurer"
2. p:location="classpath:com/smart/placeholder/jdbc.properties"
3. p:fileEncoding="utf-8"/>

#### 6.5 国际化信息

##### 6.5.1 ResourceBundleMessageSource

1. <bean id="myResource1" class="org.springframework.context.support.ResourceBundleMessageSource">
2. <property name="basenames">
3. <list>
4. <value>com/smart/i18n/fmt\_resource</value>
5. </list>
6. </property>
7. </bean>

##### 6.5.2 ReloadableResourceBundleMessageSource

1. <bean id="myResource2" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
2. <property name="basenames">
3. <list>
4. <value>com/smart/i18n/fmt\_resource</value>
5. </list>
6. </property>
7. <property name="cacheSeconds" value="2"/>
8. </bean>

##### 6.5.3 容器级的国际化资源

1. <bean id="messageSource12" class="org.springframework.context.support.ResourceBundleMessageSource">
2. <property name="basenames">
3. <list>
4. <value>com/smart/i18n/fmt\_resource</value>
5. </list>
6. </property>
7. </bean>

#### 6.6 容器事件

### chapter7--Spring AOP基础

#### 7.1 AOP概述

##### 7.1.1 AOP 术语

###### 7.1.1.1 连接点jointpoint

连接点是aop想目标类打入契子的一个选锚点，如类开始初始化前，类初始化后，类的某个方法调用前，调用后

###### 7.1.1.2 切点pointcut

每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点。AOP通过切点定义特殊的连接点。一个切点可以匹配多个连接点

###### 7.1.1.3 增强advice

###### 7.1.1.4 目标对象 target

###### 7.1.1.5 引介 introduction

引介是一种特殊的增强，它为类添加一些属性和方法

###### 7.1.1.6 织入 weaving

###### 7.1.1.7 代理proxy

一个类被aop织入增强后，就产生了一个结果类

#### 7.2 创建增强类

##### 7.2.1 增强类型

###### 7.2.1.1 前置增强

1. public class GreetingBeforeAdvice implements MethodBeforeAdvice {
2. public void before(Method method, Object[] args, Object obj) throws Throwable {
3. String clientName = (String)args[0];
4. System.out.println("How are you锛丮r."+clientName+".");
5. }
6. }
7. <bean
8. id="greetingBeforeAdvice " class="com.smart.advice.GreetingBeforeAdvice" />
9. <bean id="target" class="com.smart.advice.NaiveWaiter" />
10. <bean id="waiter"
11. class="org.springframework.aop.framework.ProxyFactoryBean"
12. p:proxyInterfaces="com.smart.advice.Waiter"
13. p:target-ref="target"
14. p:interceptorNames="greetingBeforeAdvice"
15. />

###### 7.2.1.2 后置增强

1. public class GreetingAfterAdvice implements AfterReturningAdvice {
2. public void afterReturning(Object returnObj, Method method, Object[] args,
3. Object obj) throws Throwable {
4. System.out.println("Please enjoy yourself!");
5. }
6. }
7. <bean id="greetingBefore" class="com.smart.advice.GreetingBeforeAdvice" />
8. <bean id="greetingAfter" class="com.smart.advice.GreetingAfterAdvice" />
9. <bean id="waiter"
10. class="org.springframework.aop.framework.ProxyFactoryBean"
11. p:proxyInterfaces="com.smart.advice.Waiter" p:target-ref="target"
12. p:interceptorNames="greetingBefore,greetingAfter"/>

###### 7.2.1.3 环绕增强

1. public class GreetingInterceptor implements MethodInterceptor {
2. public Object invoke(MethodInvocation invocation) throws Throwable {
3. Object[] args = invocation.getArguments();
4. String clientName = (String)args[0];
5. System.out.println("How are you锛丮r."+clientName+".");
7. Object obj = invocation.proceed();
9. System.out.println("Please enjoy yourself!");
11. return obj;
12. }
13. }
14. <bean id="greetingAround" class="com.smart.advice.GreetingInterceptor" />
15. <bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"
16. p:proxyInterfaces="com.smart.advice.Waiter" p:target-ref="target"
17. p:interceptorNames="greetingAround" />

###### 7.2.1.4 异常抛出增强

1. public class ForumService {
2. public void removeForum(int forumId) {
3. // do sth...
4. throw new RuntimeException("杩愯寮傚父銆�");
5. }
6. public void updateForum(Forum forum) throws Exception{
7. // do sth...
8. throw new SQLException("鏁版嵁鏇存柊鎿嶄綔寮傚父銆�");
10. }
11. }
12. public class TransactionManager implements ThrowsAdvice {
13. public void afterThrowing(Method method, Object[] args, Object target,
14. Exception ex) throws Throwable {
15. System.out.println("-----------");
16. System.out.println("method:" + method.getName());
17. System.out.println("抛出异常:" + ex.getMessage());
18. System.out.println("成功回滚事务。");
19. }
20. }
21. <bean id="forumServiceTarget" class="com.smart.advice.ForumService" />
22. <bean id="transactionManager" class="com.smart.advice.TransactionManager" />
23. <bean id="forumService" class="org.springframework.aop.framework.ProxyFactoryBean"
24. p:interceptorNames="transactionManager"
25. p:target-ref="forumServiceTarget"
26. p:proxyTargetClass="true"/>

###### 7.2.1.5 引介增强

#### 7.4创建切面

##### 7.4.1 静态普通方法匹配切面

1. public class GreetingAdvisor extends StaticMethodMatcherPointcutAdvisor {
2. public boolean matches(Method method, Class clazz) {
3. return "greetTo".equals(method.getName());
4. }
5. public ClassFilter getClassFilter(){
6. return new ClassFilter(){
7. public boolean matches(Class clazz){
8. return Waiter.class.isAssignableFrom(clazz);
9. }
10. };
12. }
13. }
14. <?xml version="1.0" encoding="UTF-8" ?>
15. <beans xmlns="http://www.springframework.org/schema/beans"
16. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
17. xmlns:util="http://www.springframework.org/schema/util"
18. xsi:schemaLocation="http://www.springframework.org/schema/beans
19. http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
20. http://www.springframework.org/schema/util
21. http://www.springframework.org/schema/util/spring-util-4.0.xsd">
22. <bean id="waiterTarget" class="com.smart.advisor.Waiter" />
23. <!-- <bean id="sellerTarget" class="com.smart.advisor.Seller" /> -->
24. <bean id="greetingBeforeAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />
25. <bean id="greetingAdvisor" class="com.smart.advisor.GreetingAdvisor"
26. p:advice-ref="greetingBeforeAdvice" />
28. <bean id="parent" abstract="true"
29. class="org.springframework.aop.framework.ProxyFactoryBean"
30. p:interceptorNames="greetingAdvisor"
31. p:proxyTargetClass="true" />
32. <bean id="waiter" parent="parent" p:target-ref="waiterTarget" />
33. </beans>

##### 7.4.2静态正则表达式匹配切面

##### 7.4.3动态切面

##### 7.4.4 流程切面

##### 7.4.5 复合切点切面

##### 7.4.5 引介漆面

#### 7.5自动创建代理

##### 7.5.1 BeanNameAutoProxyCreator

1. <bean
2. class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"
3. p:beanNames="\*er" p:interceptorNames="greetingAdvice"
4. p:optimize="true"/>

##### 7.5.2 DefaultAdvisorAutoProxyCreator

<!--通过Advisor自动创建代理-->

1. <bean id="regexpAdvisor"
2. class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
3. p:patterns=".\*greet.\*" p:advice-ref="greetingAdvice" />
4. <bean
5. class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" p:proxyTargetClass="true" />

##### 7.5.3 aop:confi

1. <aop:config proxy-target-class="true">
2. <aop:pointcut id="serviceMethod"
3. expression="(execution(\* com..\*(..)))" />
4. <aop:advisor pointcut-ref="serviceMethod" advice-ref="greetingBeforeAdvice" />
5. </aop:config>

### chapter8--基于@AspectJ和Schema的AOP

#### 8.1 使用@AspectJ

##### 8.1.1 通过编码使用@AspectJ

1. Waiter target = new NaiveWaiter();
2. AspectJProxyFactory factory = new AspectJProxyFactory();
3. factory.setTarget(target);
4. factory.addAspect(PreGreetingAspect.class);
5. Waiter proxy = factory.getProxy();
6. proxy.greetTo("John");
7. proxy.serveTo("John");
8. public class NaiveWaiter implements Waiter {
9. public void greetTo(String clientName) {
10. System.out.println("NaiveWaiter:greet to "+clientName+"...");
11. }
12. public void smile(String clientName,int times){
13. System.out.println("NaiveWaiter:smile to "+clientName+ times+"times...");
14. }
15. }
16. @Aspect
17. public class PreGreetingAspect{
18. @Before("execution(\* greetTo(..))")
19. public void beforeGreeting(){
20. System.out.println("How are you");
21. }
22. }

##### 8.1.2 通过配置使用@AspectJ

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
6. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">
7. <aop:aspectj-autoproxy/>
8. <!--bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"/-->
9. <bean id="waiter" class="com.smart.NaiveWaiter" />
10. <bean class="com.smart.aspectj.example.PreGreetingAspect" />
11. </beans>

#### 8.2 @AspectJ语法

##### 8.2.1 切点表达式函数

| **函数** | **入参** | **示例** |
| --- | --- | --- |
| execution() | 方法匹配字符串 | execution(\* com.yc.service.\*.\*(..))  数量（体现在 “(..)“） |
| @annotation() | 方法注解类名 | @anntation(com.yc.controller.needRecord) |
| args() | 类名 | @args(com.yc.annotation.MyAnnotation) |
| @args() | 类型注解类名 | @args(com.yc.annotation.MyAnnotation) |
| within() | 类名匹配串 | within(com.yc.service.\*ServiceImpl)，。 |
| target() | 类名 | target(com.yc.service.IUserService), |
| @within() | 类型注解类名 | @within(com.yc.controller.needRecord), |
| @target() | 类型注解类名 | @target(com.yc.controller.needRecord) |
| this |  |  |

##### 8.2.2 入参中使用通配符

###### 8.2.2.1 \*

\*匹配任意字符，但它智能匹配上下文中的一个元素

###### 8.2.2.2 ..

匹配任意字符，可以匹配上下文中的多个元素，但在表示类时，必须和\*联合使用，但在表示入参时可以单独使用

###### 8.2.2.3 +

表示按类型匹配指定的所欲类，必须跟在类名后面，如com.xxx.xxx.Car+

###### 8.2.2.4 支持程度

（1）支持所有通配符execution（） within（）

（2）仅支持+ args（） this target

（3）不支持通配符 @args（） @within（） @target（） @annotation

##### 8.2.3 逻辑运算符

###### 8.2.3.1 &&

###### 8.2.3.2 ||

###### 8.2.3.3 ！

##### 8.2.4不同的增强类型

###### 8.2.4.1 @before前置增强

value：该成员用于定义切点

argNames：由于无法通过java反射机制获取方法入参名，所以如果在java编译时未启用调试信息，或者需要在运行期解析起点，就必须通过这个成员制定注解标注增强方法的参数名，多个参数名需要用逗号分隔

###### 8.2.4.2 @AfterRunning 后置增强

value：该成员用于定义切点

pointcut：表示切点信息，如果显示地制定pointcut值，将覆盖value的值

returning：将目标对象方法的返回值绑定给增强方法

argName：同前所述

###### 8.2.4.3 @Around

value：

argNames：

###### 8.2.4.4 @AfterThrowing

value：

pointcut：

throwing：将抛出的异常绑定到增强方法中

argNames:

###### 8.2.4.5 @After Final增强，不管抛出异常还是正常退出，该增强都会执行，该增强没有对应的增强接口，可以把它看成throwsAdvice和AfterReturningAdvic的混合物，一般用于释放资源，相当于try{}finally{}的控制流

value:

argNames:

###### 8.2.4.6 @DeclareParents引介增强，

value：该成员用于定义切点，它表示在哪个目标类上添加引介增强

defaultImpl：默认的接口实现类

##### 8.2.5用法

###### 8.2.5.1 引介增强的用法

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
6. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">
7. <aop:aspectj-autoproxy/>
8. <bean id="waiter" class="com.smart.NaiveWaiter"/>
9. <bean class="com.smart.aspectj.basic.EnableSellerAspect"/>
10. </beans>
11. @Aspect
12. public class EnableSellerAspect {
13. @DeclareParents(value="com.smart.NaiveWaiter",
14. defaultImpl=SmartSeller.class)
15. public Seller seller;
16. }

###### 8.2.5.2 execution（）

execution()是最常用的切点函数，其语法如下所示：

整个表达式可以分为五个部分：

1、execution(): 表达式主体。

2、第一个\*号：表示返回类型，\*号表示所有的类型。

3、包名：表示需要拦截的包名，后面的两个句点表示当前包和当前包的所有子包，com.sample.service.impl包、子孙包下所有类的方法。

4、第二个\*号：表示类名，\*号表示所有的类。

5、\*(..):最后这个星号表示方法名，\*号表示所有的方法，后面括弧里面表示方法的参数，两个句点表示任何参数。

execution(<修饰符模式>?<返回类型模式><方法名模式>(<参数模式>)<异常模式>?) 除了返回类型模式、方法名模式和参数模式外，其它项都是可选的。与其直接讲解该方法的使用规则，还不如通过一个个具体的例子进行理解。下面，我们给出各种使用execution()函数实例。

**1)通过方法签名定义切点**

execution(public \* \*(..))l

匹配所有目标类的public方法，但不匹配SmartSeller和protected voidshowGoods()方法。第一个\*代表返回类型，第二个\*代表方法名，而..代表任意入参的方法；

execution(\* \*To(..))l

匹配目标类所有以To为后缀的方法。它匹配NaiveWaiter和NaughtyWaiter的greetTo()和serveTo()方法。第一个\*代表返回类型，而\*To代表任意以To为后缀的方法；

**2)通过类定义切点**

execution(\*com.baobaotao.Waiter.\*(..))l

匹配Waiter接口的所有方法，它匹配NaiveWaiter和NaughtyWaiter类的greetTo()和serveTo()方法。第一个\*代表返回任意类型，com.baobaotao.Waiter.\*代表Waiter接口中的所有方法；

execution(\*com.baobaotao.Waiter+.\*(..))l

匹配Waiter接口及其所有实现类的方法，它不但匹配NaiveWaiter和NaughtyWaiter类的greetTo()和serveTo()这两个Waiter接口定义的方法，同时还匹配NaiveWaiter#smile()和NaughtyWaiter#joke()这两个不在Waiter接口中定义的方法。

**3)通过类包定义切点**

在类名模式串中，“.\*”表示包下的所有类，而“..\*”表示包、子孙包下的所有类。

execution(\* com.baobaotao.\*(..))l

匹配com.baobaotao包下所有类的所有方法；

execution(\* com.baobaotao..\*(..))l

匹配com.baobaotao包、子孙包下所有类的所有方法，如com.baobaotao.dao，com.baobaotao.servier以及com.baobaotao.dao.user包下的所有类的所有方法都匹配。“..”出现在类名中时，后面必须跟“\*”，表示包、子孙包下的所有类；

execution(\* com..\*.\*Dao.find\*(..))l

匹配包名前缀为com的任何包下类名后缀为Dao的方法，方法名必须以find为前缀。如com.baobaotao.UserDao#findByUserId()、com.baobaotao.dao.ForumDao#findById()的方法都匹配切点。

**4)通过方法入参定义切点**

切点表达式中方法入参部分比较复杂，可以使用“\*”和“..”通配符，其中“\*”表示任意类型的参数，而“..”表示任意类型参数且参数个数不限。

execution(\* joke(String,int)))l

匹 配joke(String,int)方法，且joke()方法的第一个入参是String，第二个入参是int。它匹配NaughtyWaiter#joke(String,int)方法。如果方法中的入参类型是java.lang包下的类，可以直接使用类名，否则必须使用全限定类名，如joke(java.util.List,int)；

execution(\* joke(String,\*)))l

匹 配目标类中的joke()方法，该方法第一个入参为String，第二个入参可以是任意类型，如joke(Strings1,String s2)和joke(String s1,double d2)都匹配，但joke(String s1,doubled2,String s3)则不匹配；

execution(\* joke(String,..)))l

匹配目标类中的joke()方法，该方法第 一个入参为String，后面可以有任意个入参且入参类型不限，如joke(Strings1)、joke(String s1,String s2)和joke(String s1,double d2,Strings3)都匹配。

execution(\* joke(Object+)))l

匹 配目标类中的joke()方法，方法拥有一个入参，且入参是Object类型或该类的子类。它匹配joke(Strings1)和joke(Client c)。如果我们定义的切点是execution(\*joke(Object))，则只匹配joke(Object object)而不匹配joke(Stringcc)或joke(Client c)。

###### 8.2.5.3 args()和@args()

args()函数的入参是类名，@args()函数的入参必须是注解类的类名。虽然args()允许在类名后使用+通配符后缀，但该通配符在此处没有意义：添加和不添加效果都一样。

1)args()

该函数接受一个类名，表示目标类方法入参对象按类型匹配于指定类时，切点匹配，如下面的例子：

args(com.baobaotao.Waiter)

表 示运行时入参是Waiter类型的方法，它和execution(\*\*(com.baobaotao.Waiter))区别在于后者是针对类方法的签名而言的，而前者则针对运行时的入参类型而言。如args(com.baobaotao.Waiter)既匹配于addWaiter(Waiterwaiter)，也匹配于addNaiveWaiter(NaiveWaiter naiveWaiter)，而execution(\*\*(com.baobaotao.Waiter))只匹配addWaiter(Waiterwaiter)方法；实际上，args(com.baobaotao.Waiter)等价于execution(\*\*(com.baobaotao.Waiter+))，当然也等价于args(com.baobaotao.Waiter+)。

2)@args()

该函数接受一个注解类的类名，当方法的运行时入参对象标注发指定的注解时，方法匹配切点。这个切点函数的匹配规则不太容易理解，我们通过以下示意图对此进行详细讲解：

8.2.5.4 @within和@target

###### 8.2.5.5 target 和 this

##### 8.2.6 进阶

###### 8.2.6.1切点复合运算

###### 8.2.6.2命名切点

###### 8.2.6.3增强织入的顺序

###### 8.2.6.4访问连接点信息

###### 8.2.6.5绑定连接点方法入参

###### 8.2.6.6 绑定代理对象

###### 8.2.6.7绑定类注解对象

###### 8.2.6.8 绑定返回值

###### 8.2.6.9 绑定抛出异常

#### 8.3 基于schema 配置切面

### chapter9--Spring SPEL

## 第3篇 数据篇

### chapter10--Spring 对DAO的支持

### chapter11--Spring的事务管理

### chapter12--的事务管理难点剖析、

### chapter13--Spring Spring JDBC访问数据库

### chapter14--整合其他ORM框架

## 第4篇 应用篇

### chapter15--Spring cache

### chapter16--任务调度和异步执行器

### chapter17--spring mvc

#### 17.1 dispatcherServlet内部逻辑

##### 17.1.1 voidinitStrategies

1. protected voidinitStrategies(ApplicationContext context) {
2. initMultipartResolver(context);//初始化文件上传解析器
3. initLocaleResolver(context);//初始化本地解析器
4. initThemeResolver(context);//初始化主题解析器
5. initHandlerMappings(context);//初始化处理器映射器
6. initHandlerAdapters(context);//初始化处理器适配器
7. initHandlerExceptionResolvers(context);//初始化异常解析器
8. initRequestToViewNameTranslator(context);//初始化请求到视图名称解析器
9. initViewResolvers(context);//初始化视图解析器
10. initFlashMapManager(context);
11. }

##### 17.1.2 DispatcherServlet.properties

org/springframework/web/servlet 类路径

1. ##本地化解析器
2. org.springframework.web.servlet.LocaleResolver=
3. org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver
4. ##主题解析器
5. org.springframework.web.servlet.ThemeResolver=
6. org.springframework.web.servlet.theme.FixedThemeResolver
7. ##处理器映射（共2个）
8. org.springframework.web.servlet.HandlerMapping=
9. org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
10. org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping
11. ##处理器适配器（共3个）
12. org.springframework.web.servlet.HandlerAdapter=
13. org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
14. org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
15. org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter
16. ##异常处理器（共3个）
17. org.springframework.web.servlet.HandlerExceptionResolver=
18. org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\
19. org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
20. org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver
21. ##视图名称翻译器
22. org.springframework.web.servlet.RequestToViewNameTranslator=
23. org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator
24. ##视图解析器
25. org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

##### 17.1.3 表格

solver类型为MultipartResolver的Bean作为该类型组件；

|  |  |
| --- | --- |
| 组件类型 | 发现机制 |
| 文件上传  解析器  （☆） | 1）查找名为multipartResolver类型为MultipartResolver的Bean作为该类型组件；  2）没有默认的实现类。  如果用户没有在上下文中显式定义这一类型的组件，DispatcherServlet中将不会拥有该类型的组件 |
| 本地化解析器  （☆） | 1）查找名为localeResolver类型为LocaleResolver的Bean作为该类型组件；  2）如果1）找不到，使用默认的实现类（AcceptHeaderLocaleResolver）创建该类型组件 |
| 主题解析器  （☆） | 1）查找名为themeResolver类型为LocaleResolver的Bean作为组件；  2）如果1）找不到，使用默认的实现类（FixedThemeResolver） |
| 处理器映射器  （★） | 1）如果detectAllHandlerMappings属性为true（默认为true），根据类型匹配机制查找上下文及父Spring容器中所有类型为HandlerMapping的Bean，将它们作为该类型组件；  2）如果detectAllHandlerMappings属性为false，查找名为handlerMapping类型为HandlerMapping的Bean作为该类型组件；  3）如果通过以上两种方式都找不到，使用BeanNameUrlHandlerMapping实现类创建该类型的组件 |
| 处理器适配器  （★） | 1）如果detectAllHandlerAdapters属性为true（默认为true），根据类型匹配机制查找上下文及父Spring容器中所有类型为HandlerAdapter的Bean，将它们作为该类型组件；  2）如果detectAllHandlerAdapters属性为false，查找名为handlerAdapter类型为HandlerAdapter的Bean作为该类型组件；  3）如果通过以上两种方式都找不到，使用DispatcherServlet.properties配置文件中指定的三个实现类分别创建一个适配器，添加到适配器列表中 |
| 处理器异常  解析器  （★） | 1）如果detectAllHandlerExceptionResolvers属性为true（默认为true），根据类型匹配（HandlerExceptionResolver）机制查找上下文及父Spring容器中所有匹配的Bean作为该类型组件；  2）如果detectAllHandlerExceptionResolvers属性为false，查找名为handlerExceptionResolver类型为HandlerExceptionResolver的Bean作为该类型组件；  3）如果通过以上两种方式都找不到，查找DispatcherServlet.properties中定义的默认实现类，不过该文件中没有对应处理器异常解析器的默认实现类（用户可以更改属性文件） |
| 视图名翻译器  （☆） | 1）查找名为viewNameTranslator类型为RequestToViewNameTranslator的Bean作为该类型组件；  2）如果1）找不到，使用默认的实现类（DefaultRequestToViewNameTranslator）创建该类型的组件 |
| 视图解析器  （★） | 1）如果detectAllViewResolvers属性为true（默认为true），根据类型匹配（ViewResolver）机制查找上下文及父Spring容器中所有匹配的Bean作为该类型组件；  2）如果detectAllViewResolvers属性为false，查找名为viewResolver类型为ViewResolver的Bean作为该类型组件；  3）如果通过以上两种方式都找不到，通过DispatcherServlet.properties中定义的默认实现类（InternalResourceViewResolver）创建该类型的组件 |

#### 17.2注解驱动的控制器

#### 17.3通过@requestMapping映射需求

##### 17.3.1 url进行映射

1. @RequestMapping(value="/user/createUser")

⦁ /user/\*/createUser：匹配/user/aaa/createUser、/user/bbb/createUser等URL。

⦁ /user/\*\*/createUser：匹配/user/createUser、/user/aaa/bbb/createUser等URL。

⦁ /user/createUser??：匹配/user/createUseraa、/user/createUserbb等URL。

⦁ /user/{userId}：匹配user/123、user/456等URL。

⦁ /user/\*\*/{userId}：匹配user/aaa/bbb/123、user/aaa/456等URL。

⦁ company/{companyId}/user/{userId}/detail：匹配company/123/user/456/detail等的URL。

1. @RequestMapping("/{userId}")
2. public ModelAndView showDetail(@PathVariable("userId") String userId){
3. ModelAndView mav = new ModelAndView();
4. mav.setViewName("user/showDetail");
5. mav.addObject("user", userService.getUserById(userId));
6. return mav;
7. }

##### 17.3.2通过请求参数、请求方法或请求头进行映射

1. @Controller
2. @RequestMapping("/user")
3. public class UserController {
5. @RequestMapping(value="/delete",method=RequestMethod.POST,params="userId") ①
6. public String test1(@RequestParam("userId") String userId){
7. //do sth
8. return "user/test1";
9. }
11. @RequestMapping(value="/show",headers="content-type=text/\*")②
12. public String test2(@RequestParam("userId") String userId){
13. //do sth
14. return "user/test2";
15. }
16. }

###### 请求处理方法签名

1. //①请求参数按名称匹配的方式绑定到方法入参中，方法返回的字符串代表逻辑视图名
2. @RequestMapping(value="/handle1")
3. public String handle1(@RequestParam("userName") String userName,
4. @RequestParam("password") String password,
5. @RequestParam("realName") String realName){
6. return "success";
7. }
8. //②将Cooke值及报文头属性绑定到入参中、方法返回ModelAndView
9. @RequestMapping(value="/handle2")
10. public ModelAndView handle2(@CookieValue("JSESSIONID") String sessionId,
11. @RequestHeader("Accept-Language") String accpetLanguage){
12. ModelAndView mav = new ModelAndView();
13. mav.setViewName("success");
14. mav.addObject("user", new User());
15. return mav;
16. }
18. //③请求参数按名称匹配的方式绑定到user的属性中、方法返回的字符串代表逻辑视图名
19. @RequestMapping(value="/handle3")
20. public String handle3(User user){
21. return "success";
22. }
24. //④直接将HTTP请求对象传递给处理方法、方法返回的字符串代表逻辑视图名
25. @RequestMapping(value="/handle4")
26. public String handle4(HttpServletRequest request){
27. return "success";
28. }

从代码清单15-12中我们可以发现，Spring MVC会结合方法入参类型、入参的注解、返回值的类型按照“契约”进行相应的处理。

##### 17.3.3 使用矩阵变量绑定参数

##### 17.3.4 请求处理方法签名详细说明

###### 17.3.4.1@RequestParam

1. @RequestMapping(value = "/handle11")
2. public String handle11(
3. @RequestParam(value = "userName", required = false) String userName,
4. @RequestParam("age") int age) {
5. …
6. }

###### 17.3.4.2 @CookieValue

1. @RequestMapping(value = "/handle12")
2. public String handle12(
3. @CookieValue(value="sessionId",required=false) String sessionId,
4. @RequestParam("age") int age) {
5. …
6. }

###### 17.3.4.3 @RequestHeader

1. @RequestMapping(value = "/handle13")
2. public String handle13(@RequestHeader("Accept-Encoding") String encoding,
3. @RequestHeader("Keep-Alive") long keepAlive) {
4. …
5. }

###### 17.3.4.4 使用命令/表单对象

1. @RequestMapping(value = "/handle14")
2. public String handle14(User user) {
3. …
4. }

###### 17.3.4.5 IO对象

1. @Controller
2. @RequestMapping("/user")
3. public class UserController {
5. @RequestMapping(value = "/handle31")
6. public void handle31(OutputStream os) throws IOException{
7. Resource res = new ClassPathResource("/image.jpg");//读取类路径下的图片文件
8. FileCopyUtils.copy(res.getInputStream(), os);//将图片写到输出流中
9. }
10. }

###### 17.3.4..6其他类型的参数

控制器处理方法的入参除支持以上类型的参数以外，还支持java.util.Locale、java. security.Principal，可以通过Servlet的HttpServletRequest 的getLocale()及getUserPrincipal()得到相应的值。如果处理方法的入参类型为Locale或Principal，Spring MVC自动从请求对象中获取相应的对象并传递给处理方法的入参。

#### 17.4使用HttpMessageConverter<T>

##### 17.4.1 实现类

|  |  |
| --- | --- |
| 实 现 类 | 功能说明 |
| StringHttpMessageConverter | 用途：将请求信息转换为字符串。  1）T为String类型；  2）可读取所有媒体类型（\*/\*）的请求信息，可通过设置supportedMediaTypes属性指定媒体类型；  3）响应信息的媒体类型为text/plain（即Content-Type的值） |
| FormHttpMessageConverter | 用途：将表单数据读取到MultiValueMap中。  1）T为org.springframework.util.MultiValueMap<String,?>类型；  2）支持读取application/x-www-form-urlencoded的媒体类型，但不支持读取multipart/form-data的媒体；  3）可写application/x-www-form-urlencoded及multipart/form-data媒体类型的响应信息 |
| XmlAwareFormHttpMessageConverter | 扩展于FormHttpMessageConverter，如果部分表单属性是XML数据，可用该转换器进行读取 |
| ResourceHttpMessageConverter | 用途：读写org.springframework.core.io.Resource对象。  1）T为org.springframework.core.io.Resource类型；  2）可读取所有媒体类型（\*/\*）的请求信息；  3）如果类路径下提供了JAF（Java Activation Framework），则根据Resource的类型指定响应的媒体类型，否则响应的媒体类型为application/octet-stream |
| BufferedImageHttpMessageConverter | 用途：读写BufferedImage对象。  1）T为BufferedImage类型；  2）可以读取所有媒体类型；  3）返回BufferedImage相应的媒体类型，也可以通过contentType显式指定 |
| ByteArrayHttpMessageConverter | 用途：读写二进制数据。  1）T为byte[]类型；  2）可读取所有媒体类型（\*/\*）的请求信息，可通过设置supportedMediaTypes属性指定媒体类型；  3）响应信息的媒体类型为application/octet-stream |
| SourceHttpMessageConverter | 用途：读写javax.xml.transform.Source类型的数据。  1）T为javax.xml.transform.Source类型及其扩展类，包括javax.xml. transform.dom.DOMSource、javax.xml.transform.sax.SAXSource及javax.xml. transform.stream.StreamSource；  2）可读取 text/xml和application/xml媒体类型请求；  3）响应信息的媒体类型为text/xml或application/xml |
| MarshallingHttpMessageConverter | 用途：通过Spring的 org.springframework.oxm.Marshaller （将Java对象转换成XML）和Unmarshaller（将XML解析为Java对象）读写XML消息。  1）T为Object类型；  2）可读取 text/xml和application/xml媒体类型请求；  3）响应信息的媒体类型为text/xml或application/xml |
| Jaxb2RootElementHttpMessageConverter | 用途：通过JAXB2读写XML消息，将请求消息转换到标注XmlRootElement 和XmlType注解的类中。  1）T为Object类型；  2）可读取 text/xml和application/xml媒体类型请求；  3）响应信息的媒体类型为text/xml或application/xml |
| MappingJacksonHttpMessageConverter | 用途：利用Jackson开源类包的ObjectMapper读写JSON数据。  1）T为Object类型；  2）可读取 application/json类型的数据；  3）响应信息的媒体类型为application/json |
| RssChannelHttpMessageConverter | 用途：能够读写RSS种子消息。  1）T为com.sun.syndication.feed.rss.Channel类型；  2）可读取 application/rss+xml类型的数据；  3）响应信息的媒体类型为application/rss+xml |

##### 17.4.2 用法

1. <!--① 定义一个AnnotationMethodHandlerAdapter-->
2. <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"
3. p:messageConverters-ref="messageConverters"/>
4. <!--① HttpMessageConverter列表-->
5. <util:list id="messageConverters">
6. <bean class="org.springframework.http.converter.BufferedImageHttpMessageConverter" />
7. <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
8. <bean class="org.springframework.http.converter.StringHttpMessageConverter" />
9. <bean class="org.springframework.http.converter.xml.XmlAwareFormHttpMessageConverter"/>
10. </util:list>
11. @Controller
12. @RequestMapping("/user")
13. public class UserController {
14. …
15. @RequestMapping(value = "/handle41")
16. public String handle41(**@RequestBody String requestBody** ) {①
17. System.out.println(requestBody);
18. return "success";
19. }
20. **@ResponseBody ②**
21. @RequestMapping(value = "/handle42/{imageId}")
22. public byte[] handle42(@PathVariable("imageId") String imageId) throws IOException {
23. System.out.println("load image of "+imageId);
24. Resource res = new ClassPathResource("/image.jpg");
25. byte[] fileData =FileCopyUtils.copyToByteArray(res.getInputStream());
26. return fileData;
27. }
28. }

### chapter18--实战案例开发

## 第5篇 提高篇

### chapter19--Spring OXM

### chapter20--实战单元测试

# 后端路线图

http://blog.csdn.net/wangshouxiang123/article/details/53618471?locationNum=11&fps=1#t4