Rays's

Personal Study Note

of

Spring

目 录

[1. 装配Bean 3](#_Toc7164537)

[1.1. 自动化装配 3](#_Toc7164538)

[1.2. 代码配置 3](#_Toc7164539)

[1.3. XML配置 3](#_Toc7164540)

[1.4. 导入和混合 4](#_Toc7164541)

[1.4.1. 代码配置 4](#_Toc7164542)

[1.4.2. xml配置 4](#_Toc7164543)

[2. 高级装配 5](#_Toc7164544)

[2.1. 环境和Profile 5](#_Toc7164545)

[2.1.1. 配置 profile 5](#_Toc7164546)

[2.1.2. 激活 profile 5](#_Toc7164547)

[2.2. 条件化的bean 5](#_Toc7164548)

[2.3. 处理自动装配歧义 6](#_Toc7164549)

[2.4. Bean的作用域 6](#_Toc7164550)

[2.5. 运行时注入 6](#_Toc7164551)

[2.5.1. 注入外部的值 6](#_Toc7164552)

[2.5.2. SpringEL 7](#_Toc7164553)

[3. 面向切面 8](#_Toc7164554)

[3.1. 术语 8](#_Toc7164555)

[3.1.1. 术语 8](#_Toc7164556)

[3.1.2. 动态代理 8](#_Toc7164557)

[3.1.3. SpringAOP 8](#_Toc7164558)

[3.2. 切点选择 9](#_Toc7164559)

[3.2.1. 编写切点 9](#_Toc7164560)

[3.3. 注解创建切面 10](#_Toc7164561)

[3.3.1. 定义切面 10](#_Toc7164562)

[3.3.2. 启用切面 11](#_Toc7164563)

[3.3.3. 环绕通知 11](#_Toc7164564)

[3.3.4. 添加方法 11](#_Toc7164565)

[3.4. XML创建切面 12](#_Toc7164566)

[4. Spring MVC 构建 14](#_Toc7164567)

[4.1. 请求过程 14](#_Toc7164568)

[4.2. 搭建Spring MVC 14](#_Toc7164569)

[4.2.1. 代码配置 14](#_Toc7164570)

[4.2.2. 启用SpringMVC 15](#_Toc7164571)

[4.3. 控制器编写 15](#_Toc7164572)

[4.3.1. 传递模型数据到视图 16](#_Toc7164573)

[4.3.2. 接受请求参数 16](#_Toc7164574)

[4.3.3. 处理表单 17](#_Toc7164575)

[4.3.4. 表单验证 17](#_Toc7164576)

[5. 视图 18](#_Toc7164577)

[5.1. 视图解析 18](#_Toc7164578)

[5.2. JSP视图创建 18](#_Toc7164579)

[5.2.1. 配置JSP视图解析 18](#_Toc7164580)

[5.2.2. Spring JSP库 18](#_Toc7164581)

[5.2.2.1. 表单绑定模型 18](#_Toc7164582)

[5.2.2.2. Spring通用标签库 19](#_Toc7164583)

[5.3. Apache Tiles 20](#_Toc7164584)

[5.4. Thymeleaf 20](#_Toc7164585)

[6. SpringMVC 高级技术 21](#_Toc7164586)

[6.1. 配置其他Servlet、Filter等 21](#_Toc7164587)

[6.2. 文件上传 21](#_Toc7164588)

[6.2.1. HTTP muti-part简介 21](#_Toc7164589)

[6.2.2. 配置mutipart解析器 21](#_Toc7164590)

[6.2.3. 处理muti-part请求 22](#_Toc7164591)

[6.3. 异常处理 22](#_Toc7164592)

[6.4. Flash属性 23](#_Toc7164593)

[7. Spring WEB Flow 24](#_Toc7164594)

# 装配Bean

## 自动化装配

自动化装配的两大要素

* 组件扫描（Component Scan）：自动发现Bean
* 自动装配（Autowiring）：自动处理Bean之间的关系

1.组件扫描

@ComponentScan或者<context:component-scan> 开启自动扫描功能，通过basepackage指定扫描的范围，使用 @Component 注解标记需要的Bean对应的类（或者使用JDI规范@Named）。

2.自动装配

@Autowired 注解表明Bean和其他Bean的依赖关系，可以用于构造函数、set方法和直接用于属性（或者使用JDI规范@Inject），Spring在扫描时会按照关系设置属性。

## 代码配置

Spring可以直接使用Java代码来进行配置。

1.启动代码配置

为某个类添加注解@Configuration，表明某个类是配置类。

2.配置Bean

编写方法创建并返回Bean，为方法添加@Bean注解

@bean

A bean1(){ return new A(); }

3.配置注入

在配置Bean的方法中，加入对其他Bean的依赖即可有两种方式：

一种是配置方法依赖其他Bean的配置方法

B bean2() { return new B(bean1());}

一种是配置方法依赖其他的Bean的参数传入

B bean2(A a) { return new B(a); }

## XML配置

新的Spring项目应该尽量不再使用xml配置，了解xml配置主要用于维护旧项目。

1.规范

Spring的配置xml文件以<beans>元素为根，并且需要在xml顶部声明多个xml模式（XSD）文件。

2.配置<bean>

使用 <bean> 元素配置Bean

3.配置依赖

<constructor-arg> 或 -c 标签配置构造器依赖

<property> 或 -p 配置属性依赖

## 导入和混合

### 代码配置

@Import 引入其他配置类

@ImportResource 引入配置文件

### xml配置

<Import > 元素引入其他配置文件

<bean > 元素直接将配置类作为Bean配置

# 高级装配

## 环境和Profile

对于同一个bean，希望在不同的场景下使用不同的配置方案，可以使用profile功能。

常见的场景就是数据源的配置，就算在普通情况下，也会有生产和测环境的区别，这时候数据源会希望使用不同的地址、用户名等。

### 配置 profile

@Profile注解，可以用于类、配置方法表明配置的生效范围。

如 @Profile("dev") 表明在 dev profile 激活后生效。

在xml中使用 <beans> 元素的 profile 可以配置生效范围。也可以再 <beans> 中嵌套 <beans> 而不用创建多个文件来配置生效范围。

### 激活 profile

Spring 依赖两个属性来确定当前的 profile：

spring.profiles.active和spring.profiles.default

优先使用 active 若该值没有设置则使用default，若default也没有设置，则只创建不包含 profile 的bean。

这两个属性的设置方式有多种

* 作为DispatcherServlet的初始化参数；
* 作为Web应用的上下文参数；
* 作为JNDI条目；
* 作为环境变量；
* 作为JVM的系统属性；
* 在集成测试类上，使用@ActiveProfiles注解设置。

可以同时激活多个profile，一般用于多个profile彼此无关的场景可能需要。

## 条件化的bean

@Conditional 注解用于特殊场景，只有当某个条件符合时才创建某个Bean。该注解使用 Condition 接口进行条件比对。

public interface Condition {

boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata);

}

任何实现该接口的方法都可以用于条件比对，只有matches方法返回true时，才创建bean。matches方法的参数 ConditionContext 也是一个接口，大致方法如下：

* 借助getRegistry()返回的BeanDefinitionRegistry检查bean定义；
* 借助getBeanFactory()返回的ConfigurableListableBeanFactory检查bean是

否存在，甚至探查bean的属性；

* 借助getEnvironment()返回的Environment检查环境变量是否存在以及它的值是什么；
* 读取并探查getResourceLoader()返回的ResourceLoader所加载的资源；
* 借助getClassLoader()返回的ClassLoader加载并检查类是否存在。

AnnotatedTypeMetadata 也是一个接口，可以用来检查要配置的bean是否包含其他注解。在Spring 4 之后的profile 的代码也重构为基于Conditional来创建。

## 处理自动装配歧义

Spring在处理自动装配时，若没有无歧义的选项时，会抛出NoUniqueBeanDefinitionException，

@Primary 用于Bean 定义，标记该Bean是首选Bean

@Qualifier 用于注入定义，标记将注入的值首选某个Bean

## Bean的作用域

默认情况下Spring的所有的Bean均为单例模式。

Spring定义的作用域有：

单例：整个应用中，只创建一个

原型：每次注入或获取均创建新的对象。

会话：web应用中，每个会话中只有一个。

请求：web应用中，每次请求只有一个。

使用 @Scope注解，并配置ConfigurableBeanFactory中的常量，可以设置Bean的作用域。

## 运行时注入

### 注入外部的值

1. 使用 @PropertySource 声明外部属性文件，使用 Environment 类来检索属性值。

2. 属性占位符

在配置中使用 @Value("${ ... }") 表明该处是一个属性的值

然后 配置 PropertyPlaceholderConfigurer bean或PropertySourcesPlaceholderConfigurer 来处理处理占位符。

### SpringEL

略

# 面向切面

## 术语

### 术语

面向切面：希望程序在某处做某事，将这些期望统一管理则形成切面。

* **通知Advice**，就是希望做的事情，Spring切面中有5种通知：

1. 前置、2. 后置、3. 返回、4. 异常、5. 环绕

从定义可以看出Spring的通知和Method是密切相关的，实际上就是在方法执行过程中寻找时机完成功能。

* **连接点Join Point**，可以运行通知的时机
* **切点PointCut**，需要运行通知的时机，就是某处
* **引入**，向现有的类添加新方法或属性。
* **织入**，实现新的切面功能，必然要对已经存在的代码进行修改，不过这种修改是模式化的。织入大致可以在以下几个阶段：

编译器：编译类的时候，加入切面的代码，这时候需要特殊的编译器。

类加载期：加载类的时候，加入切面的代码，需要特殊的加载器（ClassLoader）

运行期：应用运行的时候，加入切面的代码，一般是通过动态代理实现。Spring就是基于该方式。

### 动态代理

代理是一种设计模式， java为代理提供了动态代理的语法支持。

假设一个类Impl实现了接口Intf，现在要实现代理Impl，可以有两种方式：

1. 编写类IntfProxy，添加成员Impl并实现接口Intf。然后在每个Intf方法实现中，使用Impl调用对应的方法。直接创建类IntfProxy对象，就得到了Impl的代理，这就是静态代理。

2. 编写类IntfHandler实现InvocationHandler接口，添加成员Impl。在invoke方法中使用method.invoke(implObj, args)来调用Impl的方法。使用Proxy. newProxyInstance创建Impl的代理对象，这就是动态代理。

动态代理和静态代理的区别在于，动态代理不需要预先实现被代理对象的接口，一个动态代理的调用可以为任何接口的对象提供代理支持。

### SpringAOP

Spring的AOP是基于动态代理完成的，java 中动态代理的核心功能就是拦截方法调用，因此SpringAOP也是在方法拦截的基础上完成。

切面实际上是需要添加的功能，表现在代码中也只是一个方法。因此Spring的AOP的过程就是，先有业务方法，然后有切面方法，然后确定切面方法是在业务方法执行前、执行后或者异常后来执行。

基于动态代理实现的过程是这样的，业务Bean作为被代理对象，切面成为代理，对业务Bean的方法调用都会被代理拦截。代码实现上Spring使用ReflectiveMethodInvocation接管方法调用，然后递归执行proceed方法，每层递归会执行不同的Invocation，而前置、后置等通知的区别就在于是递归前还是递归后执行。



## 切点选择

前面已经介绍了连接点和切点，在Spring中切点实际上就是方法。Spring的切点指示器是AspectJ的一个子集，具体如下：

|  |  |
| --- | --- |
| 指示器 | 描述 |
| args() | 指定切点的参数类型 |
| @args() | 指定切点的参数的类型包含指定注解 |
| execution() | 指定切点方法 |
| this() | 指定切点代理引用的类，用于Bean未实现接口的情况，Spring会使用CGLib来实现代理 |
| target() | 指定切点所在的对象类，用于Bean实现了接口的情况，Spring会使用动态代理 |
| @target() | 指定切点所在的对象的类需要包含的注解 |
| within() | 指定切点所在的类或者包 |
| @within() | 指定切点所在类需要包含的注解 |
| @annotation | 指定切点方法的需要包含的注解 |
| bean() | 指定切点所在的bean的id或名称 |

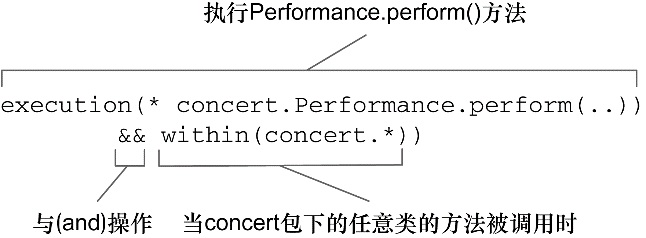
### 编写切点

编写切点的过程就是使用指示器指定切点方法的过程。Spring中可以使用&&,||和！逻辑运算符来对指示器进行逻辑运算，也可以使用 and， or 和 not。

只是器内部值是一种类似于正则表达式的模糊字符串，例如execution指示器可以这样来指定：



在所有指示器中，只有execution是直接指定方法，其他的指示器主要是在execution的基础上做二次筛选，因此可以结合其他指示器，对切点做进一步的限制，如：



## 注解创建切面

### 定义切面

使用@Aspect标注可以创建一个切面，使用具体的注解可以指定切点和通知，由于通知的不同，指定通知的方式也有所不同。具体的通知指定方式有以下几种：

|  |  |
| --- | --- |
| 注解 | 通知 |
| @After | 指定通知在方法结束或异常后调用 |
| @AfterReturning | 指定通知在方法结束后调用 |
| @AfterThrowing | 指定通知在方法异常后调用 |
| @Around | 指定通知环绕切点方法 |
| @Before | 指定通知在目标方法之前调用 |

使用不同的注解指定通知执行的时机，使用指示器指定通知覆盖的范围，Spring会根据这些配置为需要的Bean创建代理。如一个Dog类实现了Pet接口，其中有一个play()方法。然后有一个PetMaster的类，他希望自己的宠物在玩耍时不要受伤，因此他会格外关注宠物的玩耍的情况，这时候就可以创建一个关注宠物玩耍的切面。

@Aspect

class PetMaster {

@Before("execution(\*\* com.rays.entity.Pet.play(..))")

public void beforePlay(){ // watch pet play }

}

这样Spring会为Dog的bean创建一个代理，代理中在执行play方法前，会先执行PetMaster Bean的beforePlay方法。如果要为一个方法创建多个通知，那么不可避免的会重复的配置多个指示器表达式，对于Spring来说也将有一个冗余的解析过程。此时可以使用@Pointcut注解来指定切点，然后在其他的通知表达式中直接使用切点配置方法即可。

@Aspect

class PetMaster {

@Pointcut("execution(\*\* com.rays.entity.Pet.play(..))")

public void petPlay(){}

@Before("petPlay()")

public void beforePlay(){ // watch pet play }

}

### 启用切面

只是做了3.3.1中的操作并不能使切面生效，首先为了让Spring能够知道这个切面的存在，需要将PetMaster也配置成一个Bean，然后需要在配置中其中AOP功能。在配置类上使用注解@EnableAspectJAutoProxy或者在xml中使用<aop:aspectj-autoproxy>可以启用SpringAOP配置。

### 环绕通知

其他的通知中，我们只需要编写通知功能就能够起作用，环绕通知则需要多做一些操作。使用@Around可以创建环绕通知，但是必然的，我们需要在环绕通知内指定目标方法在何时调用（或者不调用）。Spring使用ProceedingJoinPoint来表示目标方法调用。因此环绕通知方法必须接受一个ProceedingJoinPoint类型的参数jp，然后在通知内需要的时候，使用jp.proceed()类调用目标方法。

@Around("execution(\*\* com.rays.entity.Pet.play(..))")

public void aroundPlay(ProceedingJoinPoint pjp) {

try {

Out.p("Another watching begin");

pjp.proceed();

Out.p("Another watching end");

} catch (Throwable e) {

e.printStackTrace();

}

}

### 添加方法

对于已经实现的Dog类，我们需要为其添加Listener接口的listen方法。常规的方式是修改Dog类，实现Listener接口，然后添加的新的方法。使用Spring可以在不修改原来类的情况下，直接为原来类的Bean添加的方法。

首先配置一个切面

@Aspect

@Component

public class DogListener {

@DeclareParents(value="com.rays.entity.Dog+", defaultImpl=CommonListener.class)

public static Listener dogListener;

}

使用@DeclareParents注解将Listener的功能引入到Dog的bean中去。

* **value**属性指定哪些类型的Bean要引入新的功能。
* **defaultImpl**指定要引入的具体功能（实现了这些功能的类）。
* **@ DeclareParents**本身标记的静态属性指定了需要引入的接口。

最后，和之前的切面一样，将切面配置为一个Bean以实现切面功能。在代码中就可以将dog bean 作为一个Listener来使用。

Pet p1 = (Pet) context.getBean("dog");

Listener l = (Listener)p1;

l.listener();

## XML创建切面

Spring的aop命名空间中，提供多个元素用来在xml中声明切面，如下：

|  |  |
| --- | --- |
| AOP配置元素 | 用途 |
| <aop:config> | 顶层的AOP配置元素 |
| <aop:advisor> | 定义通知 |
| <aop:before><aop:after>  <aop:after-returning>  <aop:after-throwing><aop:around> | 具体通知 |
| <aop:aspect> | 定义切面 |
| <aop:pointcut> | 定义切点 |
| <aop:declare-parents> | 引入额外接口方法 |

xml的配置和注解配置略有不同，注解本身所在的配置也是一种配置信息，而xml的配置则没有了这些优势。如果要通过xml指定一个切面，那么必须显示的标注切面Bean的id。然后在指定通知时，无论何种通知，都要指定切点和通知方法。

如之前的PetMaster的配置可以如下

<aop:config ref="petMaster">

<aop:before pointcut="execution(xxx)" method="beforePlay">

<aop:config>

也可以使用定义切点的方法指定切点

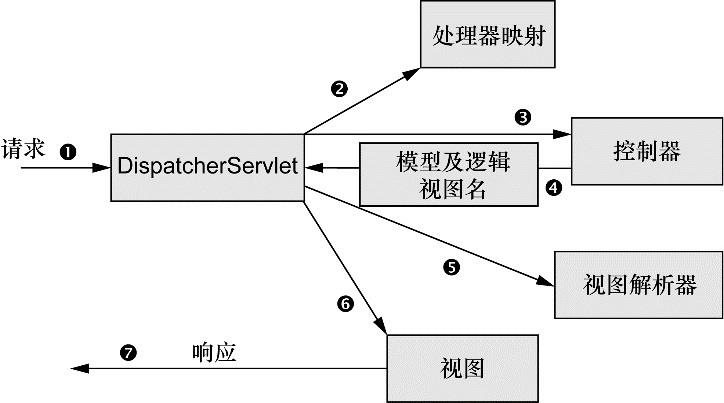
<aop:pointcut id="p1" expression="xx"/>

<aop:before pointcut-ref="p1" xx />

# Spring MVC 构建

Spring MVC是基于servlet的一个MVC框架，其使用DispatchServlet转发所有请求。

## 请求过程



通过配置请求映射到Spring的DispatchServlet，然后编写控制器和视图等模块，Spring将自动的让请求在这些模块中依次处理。请求大致经过的流程有以下几个步骤：

1.请求到达容器，容器转交请求给DispatchSetvlet

2.DispatchSetvler查询handler mapping确定本次请求的控制器

3.转交请求到具体的控制器，等候其处理

4.控制器处理完毕提供模型和逻辑视图名

5.视图解析器根据逻辑视图名解析出最终实际的视图

6.转交模型到最终视图，这个视图可以是JSP也可以是其他技术实现。

7.最终视图根据模型中的资料渲染最终视图并返回响应报文。

## 搭建Spring MVC

### 代码配置

SpringMVC的核心是DispatchServlet，配置Spring的第一步是通过配置，将需要使用Sping处理的请求映射到DispatchServlet。在Servlet3之前，容器只会从web.xml中获取servlet的映射配置，因此只能使用xml文件来配置DispatchServlet。在Servlet3规范中，容器会在类路径中查找配置类，这个配置类需要实现ServletContainerInitializer接口，Spring实现了这个接口并且将配置工作再次转交给实现了WebApplicationInitializer的类来处理配置，Spring同时提供了一个基础实现AbstractAnnotationConfigDispatcherServletInitializer类。总之，借助于此，自定义的程序可以继承以上配置类，进而通过代码来配置DispatchServlet映射和一些其他的配置。

继承的配置类，主要需要实现三个方法：

1. getServletMappings() ： 用于配置哪些请求会映射到DispatchServlet

2. getServletConfigClasses()：用于配置web组件的配置、如视图解析器、控制器、处理器映射等。

3. getRootConfigClasses()：用于配置其他Bean，如一些服务对象、数据层组件等

一个配置类的例子如下：

public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class<?>[] {RootConfig.class};

}

@Override

protected Class<?>[] getServletConfigClasses() { // 指定 servlet 配置类

return new Class<?>[] {WebConfig.class};

}

@Override

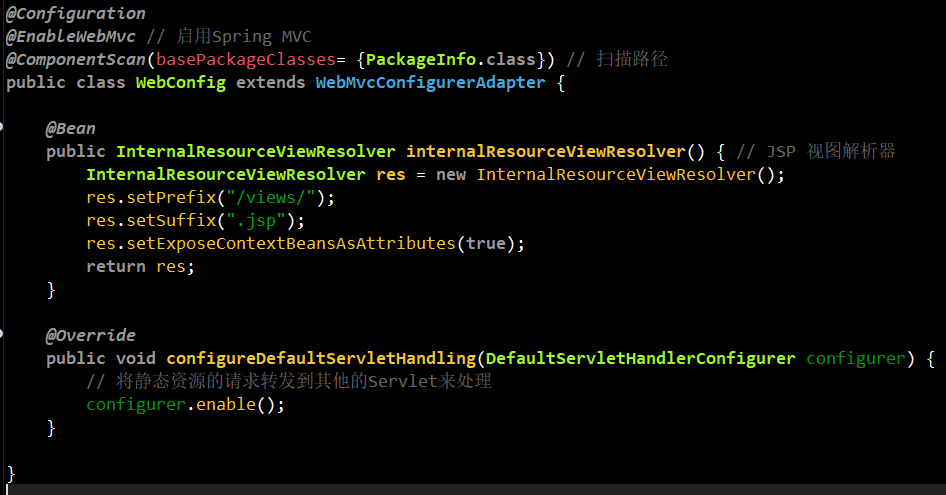
protected String[] getServletMappings() { // 映射 DispatchServlet

return new String[] {"/"};

}

}

### 启用SpringMVC



启用SpingMVC组件的相关配置如图。

## 控制器编写

控制器是处理请求的地方，实际上控制器也只是叫请求转交给其他服务对象来处理请求，编写一个控制器就是编写一个普通的java类。

* 使用@Controller注解标注其是一个控制器
* 使用@RequestMapping注解标注其处理何种请求，@RequestMapping可以用于类，也可以用于控制器的具体方法。
* 控制器方法处理完毕后，一般需要返回视图的逻辑名称。

一个简单的控制器配置如下：

@Controller // 声明为一个控制器

@RequestMapping(value="/") // 声明处理对 "/" 的请求

public class HomeController {

@RequestMapping(method=RequestMethod.GET) // 声明处理 GET 方法

public String homePage() {

return "home"; // 返回逻辑视图名称

}

}

这样配置的控制器，会处理对"/"的GET请求，然后返回"home"。经过视图解析器InternalResourceViewResolver处理后会处理为"views/home.jsp"。

### 传递模型数据到视图

一般视图都需要具体的数据来渲染具体的页面，控制器方法调用其他服务对象能够获取到一些数据，这些数据一般都需要传递给视图。在Spring中可以用控制器方法接受一个Model类型的参数，Model实际上是一个map。然后像Model中添加属性，Spring会将Model对象传递给视图。在控制器方法中如下:

public void getSomething(Model model) {model.addAttribute("thing", something);}

这些属性会被放入request作用域，在JSP使用${thing}就可以获取到属性。

### 接受请求参数

Spring可以处理以下类型的请求参数：

1.查询参数(Query Parameter)

查询参数的请求形式为http:/xxxx/App?id=123&&name=abc，在controller中可以使用@RequestParam标注入参，Spring则会将请求参数作为入参传入。

public String get(

@RequestParam("id") int id,

@RequestParam("name") String name) {..}

2.路径变量(Path Parameter)

查询参数的请求形式为http:/xxxx/App/user/1234，其中1234为参数，在controller中可以使用@RequestMapping中在请求Url中的参数的位置使用{paramName}占位符替代参数，然后使用@PathVariable标注方法入参对应的占位符。

@RequestMapping(url="/user/{user\_id}" ..)

public String get(

@PathVariable("userId") int userId) {..}

3.表单参数(Form Parameter)

### 处理表单

Spring可以自动处理表单和POJO之间的关系，只要表单的参数名称和java的实体名称一致，Spring就能自动将表单参数设置到java实体中。

如一个表单如下：

<form action="create" method="POST">

标题 <input type="text" name="title"/><br/><br/>

内容 <textarea rows="5" cols="20" name="describle"></textarea>

<input type="submit">

</form>

在Spring中使用如下Controller方法就能实现参数绑定

@RequestMapping(method=RequestMethod.POST, value={"/create"})

public String createForm(Question question) { … }

其中Question包含title和describle属性。

### 表单验证

Spring支持Java Validation API，只要在类路径下包含实现了java验证API的类即可（如Hibernate Validator）。java校验API包含多个注解，可以用于属性以限制属性的值，这里不一一列举。

在Controller的方法中可以使用Error类型的参数来接受校验结果。

# 视图

## 视图解析

Spring视图解析接受来自控制器的逻辑视图名称和模型，将其渲染输出到响应报文中。Spring定义了ViewResolver接口来抽象视图解析工作，大致如下：

public interface ViewResolver {

View resolveViewName(String viewName, Locale locale) throws Exception;

}

视图解析器返回View对象，View接口定义如下：

public interface View {

void render(Map<String, ?> model,

HttpServletRequest request,

HttpServletResponse response) throws Exception;

}

总结而言视图解析器负责根据逻辑名称将模型传递到指定视图对象，视图对象将模型渲染为页面输出到响应报文。Spring内置的视图解析器和视图实现已经可以满足绝大部分的需求，一般而言不需要自己编写实现。大部分的视图解析器都对应Web应用中特定的视图技术。

## JSP视图创建

### 配置JSP视图解析

InternalResourceViewResolver的约定是在逻辑视图的前后添加前缀和后缀，这样就能够确定JSP文件在应用中的物理位置。因此在配置InternalResourceViewResolver时，需要同时配置其prefix和suffix属性。

### Spring JSP库

#### 表单绑定模型

使用Spring表单库需要声明：

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="sf" %>

标签<sf:form>对应html的form标签，sf:form有一个 commandName属性，用来绑定表单和Model中的属性对象。当某个控制器方法向用户返回表单视图的时候，需要在模型中放置一个用来和表单绑定的对象。视图解析器会结合这个对象来渲染表单视图，渲染后这个对象会被丢弃而渲染后的视图将会被写入报文发送给客户端。像<sf:input><sf:password><sf:textarea>会被解析为对应html的文本、密码和文本区域标签。表单是结合着实体对象渲染而来，当客户端填写相关内容后，Spring负责将表单参数解析为实体对象的属性。在渲染视图时，结合表单验证API。Spring会有选择的渲染错误显示相关的标签，如在验证失败的时候<sf:errors>会被渲染到最终html和标签的class属性会被设置成sf标签的cssErrorClass属性。

sf:error标签的path属性指定当哪一个属性校验失败时，需要渲染该标签。如果path被设置为\*则该标签会显示所有校验失败的属性的验证消息。属性验证消息可以自行定义，在类路径的根目录下添加文件ValidationMessages.properties，在需要校验的属性上使用注解的message属性，可以通过属性文件映射来定义该属性的验证消息。如：

@Size(min=4,message="{question.describle.size}")

String describle;

当该属性验证失败时，Spring会去属性文件中查找question.describle.size对应的消息，然后渲染到html页面中。

#### Spring通用标签库

使用通用标签库需要如下声明

<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>

简要介绍一些通用标签：

1. <s:message>

用以提供消息支持，例如在国际化环境中，需要在不同环境下显示不同的文本就可以使用<s:message code="prop.name">嵌入消息。Spring在渲染文本时会使用MessageSource来获取prop.name对应的消息。MessageSource主要包含两个实现，ReloadableResourceBundleMessageSource与ResourceBundleMessageSource，二者的工作方式有区别，ReloadableResourceBundleMessageSource能够重新读取属性文件而不重启应用。同时在配置baseName时，ReloadableResourceBundleMessageSource可以配置应用外的目录（用file前缀指定），也可以配置类路径下（用classpath前缀指定）。创建不同后缀的属性文件，这样在不同的语言环境，MessageSource会使用不同的属性文件来获取文本。

2.创建URL

<s:url value="">用于创建url，如：

<a href="<s:url value="/question/123"/>">123</a>

会被渲染为

<a href="/RaysSpring/question/123">123</a>

Spring标签处理会自动的添加上Servlet上下文到Url中。使用<s:param>可以为生成url添加参数，同时支持用占位符替换的路径参数。使用<s:url>的htmlEscape可以将生成的url做转义处理。

3. <s:escapeBody>

用以对html内容做转义处理。

## Apache Tiles

## Thymeleaf

# SpringMVC 高级技术

## 配置其他Servlet、Filter等

使用Spring代码配置，可以配置任意数量的初始化器，添加新的类实现WebApplicationInitializer接口。

public class OtherServletConfig implements WebApplicationInitializer {

@Override

public void onStartup(ServletContext servletContext) throws ServletException {}

}

使用入参servletContext可以实现添加servlet、添加Filter、添加Listener和添加init-param等功能。

## 文件上传

### HTTP muti-part简介

HTTP的文件上传是使用Content-Type为multipart/form-data类型的POST报文，使用mutipart的一些要素如下：

1.表单

HTML可以使用form来生成mutipart报文，使用form的enctype属性，指定表单的类型为multipart/form-data。在表单域中使用input type=file可以创建文本上传组件。

2. Content-Type: multipart/form-data; boundary=----xxxx

内容类型中还有一个boundary属性，一般是随机生成的一串字符串，因为mutipart的报文是将HTTP报文分为了多个部分，boundary中的字符串就是用来分割这些部分的边界字符串，自然的这个字符串和报文每部分内容都不能存在重复。

3.请求体

请求体则是报文的实际内容，其中可以包含普通的表单属性，最主要的是可以包含二进制的文件编码。每个部分由boundary分割开来，每个部分可以包含独立的Content-type和其他的一些属性。

### 配置mutipart解析器

现在我们知道mutipart类型的报文和普通的form报文的区别，因此为了处理mutipart的请求，需要一些不同的处理方式。Spring提供了MultipartResolver接口处理该类请求并提供了两类实现：

1. CommonsMultipartResolver

容器不支持Servlet 3.0或者Spring的版本在3.0以下时的选择。

2. StandardServletMultipartResolver

如果环境具备，则优先选择使用StandardServletMultipartResolver。将其配置为一个Bean，该类没有额外的参数配置。在自定义的初始化类中，可以定制mutipart的一些细节参数，如下：

@Override

protected void customizeRegistration(ServletRegistration.Dynamic registration) {

// 配置 mutipart 的上传路径

registration.setMultipartConfig(

new MultipartConfigElement("/x/x", 2097152, 4194304, 0));

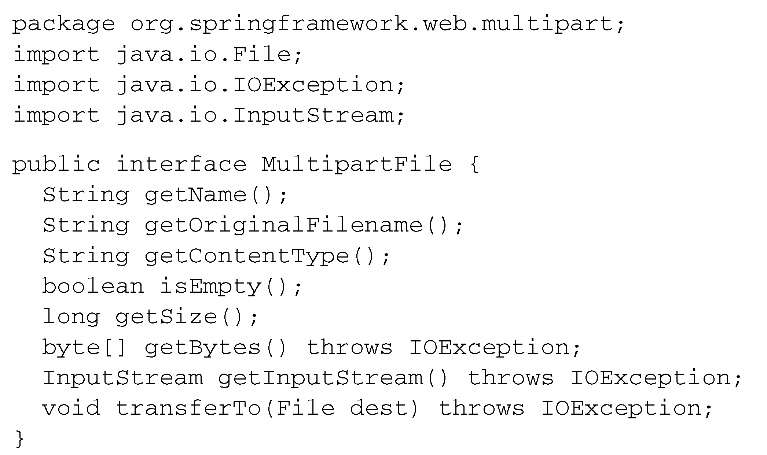
}

方法中配置了mutipart文件上传到服务器的临时路径，同时限制了其大小。

### 处理muti-part请求

请求会经过Mutipart解析器的处理，但最终还是会转交给控制器来处理，控制器中在接受文件类型时，需要一些不同于普通表单参数的处理方式。

在控制器方法中可以使用@RequestPart标注一个byte[]类型的参数，然后这个参数在方法调用时会传入文件的字节流。除此之外，还可以用MultipartFile类型的参数来接受mutipart类型的参数，使用接口提供的方法可以实现将文件写入本地或者写入其他服务器的操作。



一个controller的方法如下：

public String userRegPost(@RequestPart("profilePic") MultipartFile profilePic) {

profilePic.transferTo(new File("/root/" + profilePic.getOriginalFilename()));

return "user\_main\_page";

}

该方法可以实现将接受的文件写到本地的功能。

## 异常处理

无论程序处理的结果如何，最终程序都会对请求生成一个响应，异常控制的目的是在程序发生异常时，能够实现特定功能，然后返回合适的响应。例如如果数据库没有查询数据，并不需要给用户返回具体的数据库异常，而是返回更易明白的提示信息更为合适。

Spring提供了以下方式来将异常转换为响应：

1.将特定的异常转换为响应

Spring对一些自身的异常提供了一套转换逻辑。

2.在异常上添加@ResponseStatus注解

如下所示

@ResponseStatus(value=HttpStatus.NOT\_FOUND, reason="User not found")

public class UserNotFoundException extends RuntimeException {

}

当程序抛出UserNotFoundException时则会生成404错误报文。

3.在方法上添加@ExceptionHandler使其用来处理异常

@ExceptionHandler(UserNotFoundException.class)

public String exceptionHandler(){}

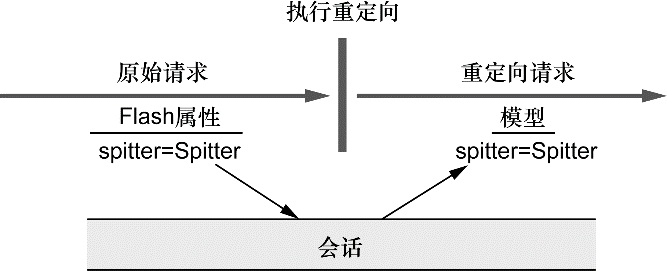
这样的方法会处理该控制器中所有方法抛出该异常后进行处理。

4.控制器通知

使用@ControllerAdvice标注的类也会成为一个Bean，该Bean内可以编写@ExceptionHandler标注的方法，这样任意控制器的异常都会被该方法处理。

## Flash属性

使用redirect前缀的逻辑视图名会使得客户端发生重定向，重定向的请求到达服务器时，之前保留的模型并不能传递到本次请求，使用Session上下文可以使得模型在请求之间传递。Spring中使用RedirectAttributes可以简洁的使用Session上下文，使用addFlashAttribute()可以添加属性到Session上下文，然后重定向的请求到达时从Session上下文取出模型。



# Spring Security

Spring Security的安全防护主要有两个手段：

1.使用Servlet规范的Filter对请求做安全处理。

2.使用SpringAOP对方法调用做安全性的处理。

## Spring Security的模块

SpringSecurity共有11个模块，使用结合SpringMVC的WEB应用需要至少以下几个模块：

1.Core：核心基本库

2.Configuration：解析XML和Java形式的SpringSecurity配置。

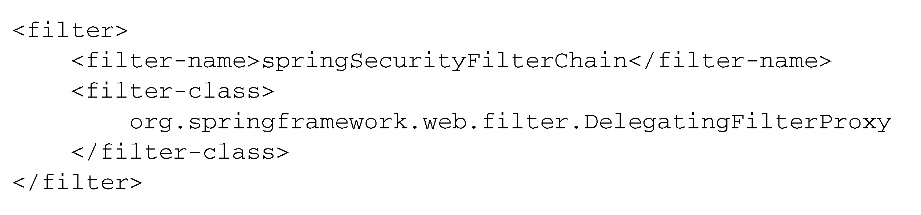
3.Web：提供基于Filter的安全配置

4.Tag Library：标签库，提供一些额外的JSP标签

## 配置

SpringSecurity使用一系列的Filter实现安全功能，在配置时只需要配置DelegatingFilterProxy，该类会将过滤工作委托给注册在Spring应用上下问中的名为springSecurityFilterChain的过滤器。配置DelegatingFilterProxy的方式有两种：

1.xml配置



2.java配置

编写初始化类，实现AbstractSecurityWebApplicationInitializer，该类实现了接口WebApplicationInitializer，其初始化方法中注册了DelegatingFilterProxy过滤器。

以上这些工作配置了过滤调用的入口或者说框架，初次之外还要配置具有实际功能的过滤器。这些过滤器不需要显示的配置，而是通过一些声明安全性的方法调用来实现。

Spring Security必须配置在一个实现了WebSecurityConfigurer的bean中，或者扩展类WebSecurityConfigurerAdapter。@EnableWebSecurity可以启用WEB安全性功能。一个配置类格式如下：

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {}

如果扩展了WebSecurityConfigurerAdapter，可以通过重载器三个方法来配置安全性功能，方法如下：

1. configure(WebSecurity)：配置Filter链

2. configure(HttpSecurity)：配置如何通过拦截器保护请求。

3. configure(AuthenticationManagerBuilder)：配置user-detail服务

## 其他

SpringSecurity属于Spring提供的安全功能，其他的功能性配置以后再做学习。

# Spring JDBC

## Spring数据访问设计



Spring访问数据库的设计遵循面向对象原则中的针对接口编程，来实现松耦合。具体实现是在服务对象中持有DAO接口，运行时注入DAO实现。这不是强制，实际使用时也可以不使用接口注入。

Spring为数据库访问提供的大量的异常设计，这里不一一介绍。

具体到数据访问，Spring将数据库访问中的固定的操作抽象为模板类，将其中可变的部分抽象为回调类。模板中一般处理事务控制、资源管理、异常处理等固定的操作，在回调中则主要负责数据访问，如准备语句、绑定参数、整理结果等。

## 使用SpringJDBC

Spring为JDBC准备模板由三个，由于版本更迭，3.1以后常用的模板有两个：

* JdbcTemplate：基本的模板，支持索引参数的查询。
* NamedParameterJdbcTemplate:：可以用命名参数来绑定SQL来查询。

使用JdbcTemplate的方式是将其配置为Bean，并为其注入数据源，然后将模板注入到Dao层中去。JdbcTemplate是JdbcOperations接口的实现，在注入时一般基于接口来注入。

1.模板插入

在Dao中引用jdbcOpt在运行时被注入JdbcTemplate，那么如果要插入数据在模板中可以使用如下方式：

jdbcOpt.update("inster into user (username, password) values (?, ?)", "abc", "1234");

2.模板查询

模板查询相对插入需要增加从结果集到实体的映射，SpringJDBC使用RowMapper这一接口抽象这一过程。

public interface RowMapper<T> {

T mapRow(ResultSet rs, int rowNum) throws SQLException;

}

如果映射User可以编写如下实现：

class UserRowMapper implements RowMapper<User> {

User mapRow(ResultSet rs, int rowNum) throws SQLException {

User user = new User();

user.setUsername(rs.getString("username"));

user.setPassword(rs.getString("password"));

return user;

}

}

然后查询方法如下：

jdbcOpt.queryForObject(QUERY\_USER\_BY\_ID, **new UserRowMapper()**, id);

# Spring事务管理

## Spring事务设计

Spring事务抽象的关键是事务策略的概念。事务策略由PlatformTransactionManager接口定义：

public interface PlatformTransactionManager {

TransactionStatus getTransaction(

TransactionDefinition definition) throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

}

该接口本质上是一个服务接口，也可以应用到程序代码中。其对象和Spring中的其他bean并无二质。和Spring的哲学一致，该接口的方法抛出的TransactionException是非检查异常。事务底层的失败通常都是不可恢复的。

getTransaction(..)方法返回一个事务状态（TransactionStatus）对象，具体的对象由TransactionDefinition参数决定。返回的事务状态对象可能表示一个新的事务，如果是JavaEE的事务上下文，也可能表示一个在当前调用栈中已经存在的事务。

TransactionDefinition接口定制：

隔离(Isolation)：此事务和其他的事物的隔离程度。例如：是否该事务能够感知其他事务未提交的写操作。

传播(Propagation)：通常一个事务内的代码应该只在当前事务上运行。但是可以通过选项来定义一个事务方法执行时，另一个事务上下文已经存在时的行为。例如，代码可以在已经存在的事务上下文中继续运行或者放弃当前事务重建一个新的事务。

超时(Timeout)：在底层的事务自动回退前，事务能够执行多久。

只读(Read-only)：只读事务用于当代码只有读操作，没有写操作的场景。

这些配置是典型事务理论的反应。如果必要，可以查看关于事务隔离层级和其他一些核心的事务概念。理解这些概念是使用Spring框架事务解决方案的基础。

TransactionStatus接口提供一些简单的方式来控制事务执行和查询事务状态。这些概念和其他的一些常用的事务接口相似：

public interface TransactionStatus extends SavepointManager {

boolean isNewTransaction();

boolean hasSavepoint();

void setRollbackOnly();

boolean isRollbackOnly();

void flush();

boolean isCompleted();

}

在Spring中使用事务管理的基础是配置正确的PlatformTransactionManager。

使用JDBC的事务管理器配置如下：

首先配置了一个数据源 datasource，然后将datasource注入给事务管理器。

<bean id="txManager" class="..DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

Hibernate中需要先定义LocalSessionFactoryBean（程序代码用以获得Hinernate Session实例），然后将sessionFactory 注入Hibernate的事务管理器。

<bean id="sessionFactory" class="...LocalSessionFactoryBean">...</bean>

<bean id="txManager" class="...HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory"/>

</bean>

只需要修改配置就能够为程序提供事务支持而不需要修改程序代码。

## 事务同步资源

### 低级同步实现

像一些类如DataSourceUtils，SesstionFactoryUtils等等都处于一个较低的层级，用于在程序中直接操作原生的持久化API同时也能够使用Sping的框架的事务同步。

例如，使用JDBC时可以使用：

DataSourceUtils.getConnection(ds);

替代：

ds.getConnection();

如果一个事务已经和一个连接关联，那么方法将会返回这个连接实例，否则方法将返回一个新的连接。

### 事务感知数据源代理

TransactionAwareDataSourceProxy存在于最低层级，这个代理类用于代理数据源，为数据源添加Spring框架管理的事务。

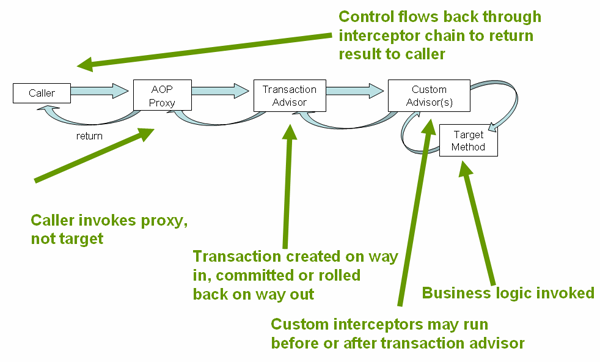
## 声明式事务管理

Spring框架的声明式事务管理通过AOP实现，但事务的代码作为Spring框架的一部分，以样板方式使用，理解AOP并不是必须的。

### 理解声明式事务

仅仅告诉你在类上添加@Transactional注解，在配置上添加@EnableTransactionManagement注解，就能够理解事务管理如何工作是不现实的。这一节介绍声明式事务管理的内部工作原理。

Spring声明式事务管理式通过AOP代理实现的，而声明则是由元数据实现（XML或注解）。AOP和事务声明组合生成AOP代理，该代理使用TransactionInterceptor结合PlatformTransactionManager围绕方法调用驱动事务。



### 声明式事务配置

考虑一下接口和实现，为这些实现添加声明式的事务支持。

public interface CatService {

Cat getCat();

void updateCat();

void insertCat();

}

public class CatServiceImpl implements CatService {…}

接口的get方法应该在只读语义的事务上下文中执行，而insert和update方法应该在读写语义的上下文中执行。以下配置将在接下来的几段中详细介绍：

<!-- from the file 'context.xml' -->

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

<beans ..>

<!-- 事务建议 -->

<tx:advice id="txAdvice" transaction-manager="txManager">

<!-- 事务语义 -->

<tx:attributes>

<!-- 所有get\*方法为read-only -->

<tx:method name="get\*" read-only="true"/>

<!-- 其他方法使用默认事务定义 -->

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<!-- 应用事务建议-->

<aop:config>

<aop:pointcut id="catOpt" expression="execution(\* CatService.\*(..))"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="catOpt"/>

</aop:config>

<!-- 数据源配置 -->

<bean id="dataSource" … />

<!-- 事务管理器 -->

<bean id="txManager" class="...DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

</beans>

在<tx:advice>中配置了所使用的事务管理器、事务语义和代理方法的模式。

### 回滚事务

推荐的方式是在事务中执行代码时抛出异常，来告诉Spring框架需要回滚事务。Spring框架会捕获所有未处理的异常，并且根据异常决定是否回滚事务。默认情况下。Spring框架只会在Runtime、Unchecked异常时回滚，可以通过如下配置来指定某些异常发生时需要回滚：

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="get\*" read-only="true"

rollback-for="NoProductInStockException"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

也可以通过no-rollback-for="InstrumentNotFoundException"来指定一个例外。初次之外也可以在代码中直接使用Spring的代码来显示的回滚事务：

try {

// 业务逻辑

} catch (NoProductInStockException ex) {

// 回滚

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

}

### 事务传播

在Spring的事务管理中，需要注意物理事务和逻辑事务之间的区别，以及传播设置如何应用于此差异。

PROPAGATION\_REQUIRED级别

当传播级别设置为PROPAGATION\_REQUIRED时，每一个配置了事务的方法创建一个逻辑事务作用域。每个逻辑事务作用域单独的确定回滚状态外部事物的作用域逻辑上和内部事务作用域独立。标准情况下，这些逻辑事务作用域都将映射到同一个物理事务。所以内部事务设置的仅回滚状态确实会影响到外部事务的提交。

When the propagation setting is PROPAGATION\_REQUIRED, a logical transaction scope is created for each method upon which the setting is applied. Each such logical transaction scope can determine rollback-only status individually, with an outer transaction scope being logically independent from the inner transaction scope. Of course, in case of standard PROPAGATION\_REQUIRED behavior, all these scopes will be mapped to the same physical transaction. So a rollback-only marker set in the inner transaction scope does affect the outer transaction’s chance to actually commit (as you would expect it to).

However, in the case where an inner transaction scope sets the rollback-only marker, the outer

transaction has not decided on the rollback itself, and so the rollback (silently triggered by the inner transaction scope) is unexpected. A corresponding UnexpectedRollbackException is thrown at that point. This is expected behavior so that the caller of a transaction can never be misled to assume that a commit was performed when it really was not. So if an inner transaction (of which the outer caller is not aware) silently marks a transaction as rollback-only, the outer caller still calls commit. The outer caller needs to receive an UnexpectedRollbackException to indicate clearly that a rollback was performed instead.

# Spring缓存

## 启用缓存

在配置类上使用注解@EnableCaching可以启用注解驱动的缓存。XML中可以使用<cache:annotation-driven>元素启用注解驱动的缓存。两种配置方式都会创建一个切面，该切面会在包含缓存注解的方法被调用时触发，这个切面会从缓存中获取数据、将数据添加到缓存或者从缓存中移除。

Spring对于缓存仅提供了声明式的支持，而缓存的具体实现则需要继承其他解决方案。

### 缓存管理器

声明了缓存配置之后还需要配置缓存管理器，Spring内置了一些缓存管理器：

* SimpleCacheManager
* NoOpCacheManager
* ConcurrentMapCacheManager
* CompositeCacheManager
* EhCacheCacheManager
* RedisCacheManager（来自于Spring Data Redis项目）
* GemfireCacheManager（来自于Spring Data GemFire项目）

具体选择哪一个取决于底层缓存供应商。ConcurrentMapCacheManager，这个简单的缓存管理器使用java.util.concurrent.ConcurrentHashMap作为其缓存存储。它非常简单，适用于开发、测试或基础的应用。

这里使用EhCache缓存实现为代表，可以配置缓存管理器为EhCacheCacheManager。在缓存的配置类中按照如下配置：

@Bean

public EhCacheCacheManager cacheManager (CacheManager cm) {

return new EhCacheCacheManager(cm);

}

@Bean

public EhCacheManagerFactoryBean ehcache() {

EhCacheManagerFactoryBean ecfb = new EhCacheManagerFactoryBean();

ecfb.setConfigLocation(new ClassPathResource("ehcache.xml"));

return ecfb;

}

这里需要注意的是Spring和EhCache都定义了CacheManager类，Ehcache的CacheManger是通过工厂方法创建的，ehcache()方法配置了工厂Bean并指定了其配置文件的路径。Spring的CacheManager类由cacheManager()方法配置，并注入了EhCache的CacheManager（从工厂生成）。

除了配置Bean以外，EhCache也定义了自己的xml配置模式，在bean的配置中，使用的配置文件是ehcache.xml，具体的配置不做介绍。

## 启用方法缓存

缓存的工作机制是使用切面，一般情况下当方法调用时，方法签名和入参会构成一个key，使用这个key去缓存中查询是否存在缓存值，如果存在则直接返回，否则需要实际执行方法，再将方法的返回值加入缓存。

Spring提供了四个注解来声明缓存规则：

* @Cacheable：声明方法会先查询缓存，未命中时也会向缓存添加值。
* @CachePut：声明方法只会像缓存中添加值。
* @CacheEvict：声明方法用于从缓存中清除值。
* @Caching：分组的注解，能够同时应用多个其他的缓存注解。

使用这些注解和其属性值，可以具体的定义方法的缓存策略，和缓存使用的key。

## XML配置缓存

有时候使用其他的jar包时，无法添加注解，或者有时候并不想在代码上添加注解时，可以使用xml来配置缓存。Spring的cache命名空间提供了一些元素来配置：

<cache:advice>：执行缓存的通知，需要结合aop:advisor应用到切点。

<cache:caching>：定义具体的缓存的规则

<cache:cacheable><cache:cache-put><cache:cache-evict>：同注解

一个简单的xml配置如下：

<cache:advice id="questionCacheAdvice" cache-manager="cacheManager">

<cache:caching cache="questionCache">

<cache:cacheable method="select\*" /> 配置方法支持缓存

</cache:caching>

</cache:advice>

<aop:config>

<aop:advisor advice-ref="questionCacheAdvice" 应用通知到切点

pointcut="bean(questionRepository\*)"

/>

</aop:config>