Rays's Note

of

Spring

目 录

[1. Spring概述 3](#_Toc16459526)

[1.1. Spring功能 3](#_Toc16459527)

[1.2. Bean容器 3](#_Toc16459528)

[1.3. Bean生命周期 4](#_Toc16459529)

[1.4. Spring模块 5](#_Toc16459530)

[2. Bean 6](#_Toc16459531)

[2.1. Hello World 6](#_Toc16459532)

[2.2. XML配置 7](#_Toc16459533)

[2.3. 代码配置 7](#_Toc16459534)

[2.4. 自动化装配 8](#_Toc16459535)

[2.5. 导入和混合 8](#_Toc16459536)

[2.5.1. 代码配置 8](#_Toc16459537)

[2.5.2. xml配置 8](#_Toc16459538)

[3. 高级装配 9](#_Toc16459539)

[3.1. 环境和Profile 9](#_Toc16459540)

[3.1.1. 配置 profile 9](#_Toc16459541)

[3.1.2. 激活 profile 9](#_Toc16459542)

[3.2. 条件化的bean 9](#_Toc16459543)

[3.3. 处理自动装配歧义 10](#_Toc16459544)

[3.4. Bean的作用域 10](#_Toc16459545)

[3.5. 运行时注入 10](#_Toc16459546)

[3.5.1. 注入外部的值 10](#_Toc16459547)

[4. 切面 11](#_Toc16459548)

[4.1. 基础 11](#_Toc16459549)

[4.1.1. 切面术语 11](#_Toc16459550)

[4.1.2. 动态代理 11](#_Toc16459551)

[4.1.3. SpringAOP 11](#_Toc16459552)

[4.2. 切点选择 12](#_Toc16459553)

[4.3. 注解创建切面 13](#_Toc16459554)

[4.3.1. 定义通知 13](#_Toc16459555)

[4.3.2. 环绕通知 14](#_Toc16459556)

[4.3.3. 添加方法 14](#_Toc16459557)

[4.4. XML创建切面 14](#_Toc16459558)

[5. Spring MVC 构建 16](#_Toc16459559)

[5.1. 请求过程 16](#_Toc16459560)

[5.2. Spring MVC配置 16](#_Toc16459561)

[5.2.1. 代码配置 16](#_Toc16459562)

[5.2.2. 启用SpringMVC 17](#_Toc16459563)

[5.3. 控制器编写 18](#_Toc16459564)

[5.3.1. 传递模型数据到视图 19](#_Toc16459565)

[5.3.2. 接受请求参数 19](#_Toc16459566)

[5.3.3. 处理表单 19](#_Toc16459567)

[5.3.4. 表单验证 20](#_Toc16459568)

[6. 视图 21](#_Toc16459569)

[6.1. 视图解析 21](#_Toc16459570)

[6.2. JSP视图创建 21](#_Toc16459571)

[6.2.1. 配置JSP视图解析 21](#_Toc16459572)

[6.2.2. Spring JSP库 21](#_Toc16459573)

[6.2.2.1. 表单绑定模型 21](#_Toc16459574)

[6.2.2.2. Spring通用标签库 22](#_Toc16459575)

[6.3. Apache Tiles 23](#_Toc16459576)

[6.4. Thymeleaf 23](#_Toc16459577)

[7. SpringMVC 高级技术 24](#_Toc16459578)

[7.1. 配置其他Servlet、Filter等 24](#_Toc16459579)

[7.2. 文件上传 24](#_Toc16459580)

[7.2.1. HTTP muti-part简介 24](#_Toc16459581)

[7.2.2. 配置mutipart解析器 24](#_Toc16459582)

[7.2.3. 处理muti-part请求 25](#_Toc16459583)

[7.3. 异常处理 26](#_Toc16459584)

[7.4. Flash属性 26](#_Toc16459585)

[8. Spring Security 27](#_Toc16459586)

[8.1. Spring Security的模块 27](#_Toc16459587)

[8.2. 配置 27](#_Toc16459588)

[8.3. 其他 28](#_Toc16459589)

[9. Spring JDBC 29](#_Toc16459590)

[9.1. Spring数据访问设计 29](#_Toc16459591)

[9.2. 使用SpringJDBC 29](#_Toc16459592)

[10. Spring事务管理 31](#_Toc16459593)

[10.1. Spring事务设计 31](#_Toc16459594)

[10.2. 声明式事务管理 32](#_Toc16459595)

[10.2.1. 理解声明式事务 32](#_Toc16459596)

[10.2.2. 声明式事务配置 32](#_Toc16459597)

[10.2.3. 回滚事务 33](#_Toc16459598)

[10.2.4. 事务传播 34](#_Toc16459599)

[10.3. 编程事务管理 35](#_Toc16459600)

[11. Spring缓存 36](#_Toc16459601)

[11.1. 启用缓存 36](#_Toc16459602)

[11.1.1. 缓存管理器 36](#_Toc16459603)

[11.2. 启用方法缓存 37](#_Toc16459604)

[11.3. XML配置缓存 37](#_Toc16459605)

# Spring概述

## Spring功能

一个网站，有一个模块负责处理订单：

class TradeHandler {

TradeDao tradeDao;

void dealTrade(Trade trade) {

trade.done() // 订单完毕，处理完成

tradeDao.insert(trade); // 记录工单信息

}

}

这个模块需要外部的一个对象给他传递工单，我们还需要一个对象来获取工单并将传递工单到处理模块：

class TradeReceive {

TradeHandler tradeHandler;

dispatch() { // 工单处理方法

tradeHandler.dealTrade(selectTrade());

}

}

那么当我们这个外卖网站启动的时候，首先需要初始化TradeHandler对象，然后需要初始化TradeReceice对象，同时将TradeHandler对象传递给TradeReceive。当对象越来越多时，初始化的顺序以及对象的传递会变得越来越复杂。

Spring将这些对象都定义为Bean，并且负责初始化和处理bean之间的依赖关系。我们可以通过配置，将TradeReceive和TradeHandler配置为Bean，并且告诉Spring他们的关系。Spring会创建好TradeHandler，然后注入给TradeReceive。这种自动处理bean关系的机制，被称为依赖注入（IOC）。一个类要成为Spring的Bean，不需要实现任何Spring的接口，就是一个普通的java类即可。

Spring还有一个强大的支持，被称为切面（AOP）。比如TradeHandler类，我们现在希望再接收到一个工单的是时候，可以给用户发一条短信。当然我们可以在handler方法中加入短信的类比如Message，然后在Handler类方法开始前调用Message.send（）。在Spring中有更好的方式来实现这种需求，就是切面。我们可以定义一个切面叫做beforHandle，然后将这个切面应用到handle方法前，同时配置这个前面的任务是执行Message.send。这一切都不需要改动handle方法就可以实现。

## Bean容器

之前我们说过Spring负责所有Bean的创建，那么在创建之后必定需要一个地方来保存这些Bean。在Spring中提供两种容器供选择，来保存对象：

1.Bean工厂，由BeanFactory接口定义，提供基本的支持。

2.上下文，由ApplicationContext接口定义,基于BeanFactory提供框架级别的服务。

大多数的应用程序中，都使用上下文来管理Bean，因为BeanFactory较为低级,Spring也提供了多种上下文供选择：

1. AnnotationConfigApplicationContext：从注解配置中加载上下文

2. AnnotationConfigWebApplicationContext：从注解加载Web应用上下文

3. XmlWebApplicationContext：从xml文件加载web应用上下文

4. ClassPathXmlApplicationContext：从类路径下的配置文件加载web应用上下文

## Bean生命周期



1.Spring创建bean实例

2.Spring将属性值和对其他bean的引用注入到当前bean对应的属性中

3.如果bean实现了BeanNameAware接口，Spring会将bean的id传入

4.如果bean实现了BeanFactoryAware接口，Spring会将BeanFactory容器实例传入

5.如果bean实现了ApplicationContextAware接口，Spring会将上下文实例传入

6.如果bean实现了BeanPostProcessor接口，Spring会调用postBefor…

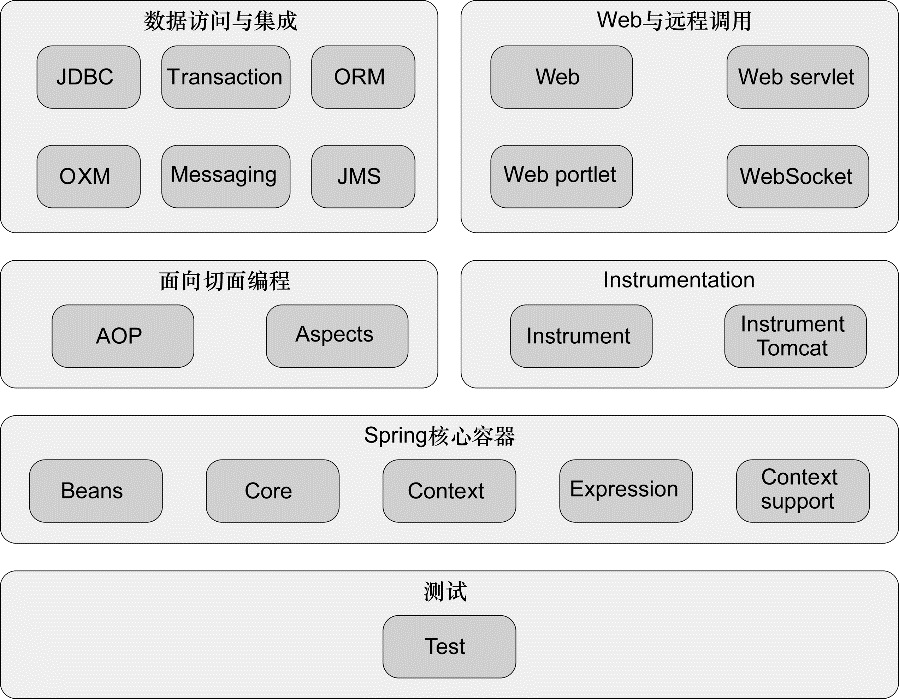
7.如果bean实现了InitializingBean接口，Spring会调用其afterPropertiesSet()方法，同时也会调用init-method中定义的方法

8.如果bean实现了BeanPostProcessor，Spring会调用postAfter…

9.此时bean已经初始化完毕

10.如果bean实现了DisposableBean，Spring将调用它的destroy()接口方法，同时也会调用destory-method中定义的方法

## Spring模块



# Bean

要使用Spring的bean功能，我们只要指定需要创建bean的类即可，然后在一个合适的上下文中加载这些配置，Spring就能够为我们创建好所有的bean，并且自动的处理好他们的依赖关系。

在Spring中配置Bean有三种方式：xml配置、代码配置和自动扫描。

## Hello World

@Component

class Dog { String name = "dog"; } // 指定dog的bean

---------------------------------------------------------------------

@Component

class DogMaster { // 指定DogMaster的bean

@Autowired Dog d;

void saySomething() { // 指定DogMaster依赖Dog

System.out.printf(

"I am master, my dog's name is %s \n", d.name); }

}

---------------------------------------------------------------------

@Configuration() // 指定配置文件

@ComponentScan(basePackageClasses=HelloWorld.class)

class Config {}

---------------------------------------------------------------------

public static void main(String[] args) {

@SuppressWarnings("resource")

ApplicationContext context // 使用上下文加载配置

= new AnnotationConfigApplicationContext(Config.class);

DogMaster dm = context.getBean(DogMaster.class); // 获取并使用bean

dm.saySomething();

}

我们创建三个类，Dog，DogMaster是需要作为bean的类，Config则是配置类。演示代码中使用的是注解配置，并且利用了自动扫描功能。

## XML配置

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。使用 <bean> 元素配置Bean，<constructor-arg> 或 -c 标签配置构造器依赖，<property> 或 -p 配置属性依赖。

<beans>

<bean name="tradeDao" class="com.ray.TradeDao">

<property name="ds" ref="datasource" />

</bean>

</breans>

## 代码配置

Spring可以直接使用Java代码来进行配置。为某个类添加注解@Configuration，表明某个类是配置类。配置bean的方式是，编写一个方法创建并返回Bean，然后为方法添加@Bean注解。在配置Bean的方法中，加入对其他Bean的依赖即可有两种方式：一种是在配置方法中调用其他Bean的配置方法；一种是配置方法将依赖的Bean定义为方法参数。

@Configuration

class Config {

@Bean

public DataSource ds() {

return new DataSource();

}

@Bean

public TradeDao() {

return new TradeDao(ds()); // 使用方法调用配置依赖

}

@Bean

public TradeService tradeService(TradeDao dao) {

return new TradeService(dao); // 使用方法参数配置依赖

}

}

## 自动化装配

自动化装配是通过注解@Component（或JDI规范的@Named），直接指定某个类需要配置为bean，然后告诉Spring这些类所在的位置需要扫描。可以在配置类上使用@ComponentScan，也可以在xml文件中配置<context:component-scan>,来启用自动配置。

@Autowired 注解（或者JDI规范的@Inject）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，Spring在扫描时会按照关系设置属性。

## 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

PropertyPlaceholderConfigurer bean或PropertySourcesPlaceholderConfigurer 可以处理占位符。

# 切面

## 基础

Spring使用bean来管理程序，bean就是普通的java对象。切面则是这些对象的一些通用需求的抽象，比如某些对象需要安全管理，现实方法调用，某对象对象需要缓存数据等等。

### 切面术语

* **通知Advice**：就是希望实现的功能，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，共有五种通知：前置通知、后置通知、方法返回通知、方法异常通知和绕通知。
* **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，切点是连接点的子集
* **引入**：向现有的类添加新方法或属性
* **织入**：将通知应用到切点， Spring中通过动态代理实现,即拦截目标的方法调用，在调用前后调用通知代码。

### 动态代理

代理是一种设计模式。比如一个类A实现了接口I，A中的实现是我们真正需要的功能。

I obj;

public void bussMethid {

obj.doSomeThing();

}

如果我们再编写一个类B也实现接口I，然后B持有一个对象A。我们在程序中使用B对象来代替对象A，就相当于间接的使用对象A。这样我们可以在B的中加入其他的逻辑，来加强方法A的原有功能，而且不需要修改A的方法。

java的动态代理则不需要实现编写类B来继承接口，我们编写一个通用的调用处理类，这个类实现了InvocationHandler接口。然后通过Proxy.newInstance方法，可以创建一个动态代理的对象，使用这个对象调用的接口方法都会被传递到调用处理类的对象的invoke方法中。调用处理类内部持有被代理对象，在invoke方法中，我们可以加入一些新的逻辑，然后将方法调用传递给被代理对象，这样就实现了动态代理。

### 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。

切面通知作为需求，在代码中表现为一段代码。Spring的AOP的过程就是，创建业务对象的代理，在invok方法中加入通知代码。然后在业务代码中使用到业务对象的地方，替换为使用业务对象的代理。

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

既然Spring的AOP通过动态代理实现，那么我们需要告诉Spring哪些类的哪些方法需要动态代理，这就是切点配置。Spring使用指示器来描述配置，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 |

可以使用&&,||和！逻辑运算符或者and,or和not来对指示器进行逻辑运算。指示器内部值是一种类似于正则表达式的模糊字符串，例如要指定切点是某个方法执行时，execution指示器可以这样来指定：

execution(\* concert.Performance.perform())

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：

execution(\* concert.Performance.perform(..)) && within(concert.\*)

## 注解创建切面

切点时指示哪些方法（bean）需要代理，通知则执行代理了这些方法后，具体做的事情。Spring的通知定义在切面内，使用@Aspect可以标注一个类为切面。切面功能需要在配置文件中启用，@EnableAspectJAutoProxy或者在xml中使用<aop:aspectj-autoproxy>可以启用SpringAOP配置。

### 定义通知

Spring的通知是切面内的一个方法，在方法上增加注解来配置通知：

|  |  |
| --- | --- |
| 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 |

配置通知时，可以直接在通知内使用指示器配置代理目标：

@Aspect

@Compoment // 注意切面本身除了Aspect注解之外，还要将其配置为bean

class PetMaster {

@Before("execution(\*\* Pet.play(..))")

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

}

也可以先定义切点，在通知内引用切点标识来配置代理目标：

@Aspect

class PetMaster {

@Pointcut("execution(\*\* Pet.play(..))")

public void petPlay(){}

@Before("petPlay()")

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

}

Spring会为切点指示器指示的对象创建代理，当指示对象的执行的方法被调用时，执行对应的通知。

### 环绕通知

使用@Around可以创建环绕通知，环绕通知的不同之处在于它需要在通知内调用被代理的方法。Spring使用ProceedingJoinPoint来表示目标方法调用，环绕通知方法需要接受该类型的参数jp，然后在通知内使用jp.proceed()类调用被代理方法。

@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

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 MyAspect {

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

public static Listener dogListener;

}

在切面内定义一个要引入的接口，并使用@DeclareParents注解指定功能要引入的bean和功能的实现类。

* **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）。

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的切面配置的顶层元素时aop:config，所有的切面配置都在此元素内。和注解类似，我们需要编写切面类，将其配置为bean，然后编写通知方法。然后在aop:config中配置切面，并且引用切面类的bean。

如之前的PetMaster的配置可以如下

<aop:config ref="petMaster">

<aop:aspect ref="petMaster">

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

</aop:aspect>

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

<aop:config>

也可以使用通知的pointcut属性直接编写切点表达式。

# 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。

## Spring MVC配置

### 代码配置

SpringMVC的核心是DispatchServlet，配置Spring的第一步是将需要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，在Servlet3规范中，容器会在类路径中查找实现ServletContainerInitializer接口的配置类。Spring实现了这个接口并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。

因此在Spring中我们不需要实现Servlet规范接口，可以继承AbstractAnnotationConfigDispatcherServletInitializer来配置程序，主要需要实现三个方法：

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

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。

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() {

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

}

@Override

protected String[] getServletMappings() {

return new String[] {"/"};

}

}

在以前的程序中会使用web.xml来配置应用，应用会有两个上下文：appName-servlet.xml文件定义的servlet上下文和web.xml中配置的ContextLoaderListener加载的应用上下文。在servlet上下文中，一般配置前端控制器、视图解析器等bean，在应用上下文中一般配置了服务层和持久层的对象。

在代码的配置中，我们使用WebConfig配置servlet上下文，而RootConfig则定义应用上下文。

### 启用SpringMVC

简单的mvc配置需要启用SpringWebmvc，然后配置视图解析器，配置静态资源映射，然后配置一些控制器bean。在WebConfig配置一个简单的mvc如下：

@Configuration

@EnableWebMvc // 启用Spring MVC

@ComponentScan(basePackageClasses= {com.rays.controller.PackageInfo.class}) // 组件扫描

public class WebConfig extends WebMvcConfigurerAdapter {

@Override

public void configureDefaultServletHandling(

DefaultServletHandlerConfigurer configurer) {

// 将静态资源的请求转发到其他的Servlet来处理

configurer.enable();

}

@Bean

public InternalResourceViewResolver internalResourceViewResolver() { // JSP 视图解析器

Log.p("加载 JSP 视图解析器");

InternalResourceViewResolver res = new InternalResourceViewResolver();

res.setPrefix("/views/");

res.setSuffix(".jsp");

res.setExposeContextBeansAsAttributes(true);

return res;

}

}

这个简单的配置中包含了前端控制器扫描、jsp视图解析器和静态资源映射。

## 前端控制器Controller

容器接收的请求被转发给了DispatcherServlet，这个servlet会进一步的将请求转发给前端控制器。前端控制器就是一个普通的java类，其内部的方法将请求再委托给其他的服务对象来处理。

* @Controller注解标注其一个类为控制器
* @RequestMapping注解标注其处理何种请求，和请求的url，@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，控制器可以调用服务对象获取数据，然后传递给视图。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。然后向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中使用${thing}就可以获取到属性。

### 接受请求参数

Spring可以处理查询参数和路径变量。

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) {

// 业务逻辑

}

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) {

// 业务逻辑

}

### 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 createQuestion(Question question) {

// 业务逻辑

}

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);

}

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);

}

视图解析器根据逻辑名称构建视图对象，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会将逻辑视图名称解析为jsp文件。配置InternalResourceViewResolver时，需要同时配置其prefix和suffix属性，prefix+logicName+suffix就是jsp的物理路径。

@Bean

public InternalResourceViewResolver viewReslover() { // JSP 视图解析器

Log.p("加载 JSP 视图解析器");

InternalResourceViewResolver res

= new InternalResourceViewResolver();

res.setPrefix("/views/");

res.setSuffix(".jsp");

res.setExposeContextBeansAsAttributes(true);

return res;

}

如果jsp中使用了jstl标签库，那么需要视图解析为JstlView，可以使用setViewClass来指定使用的视图的类。

### Spring JSP库

Spring提供了两个JSP标签库，用来帮助定义Spring MVC Web的视图。其中一个标签库会用来渲染HTML表单标签，这些标签可以绑定model中的某个属性。另外一个标签库包含了一些工具类标签，我们随时都可以非常便利地使用它们。

#### 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" %>

表单库大部分标签都用于渲染html表单，Spring的标签可以绑定一个对象，根据对象的内容来渲染html。比一个页面如下：

<sf:form method="POST" action="user\_login" commandName="user">

用户名 <sf:input path="username" ><br/>

密码 <sf:password path="password" /><br/>

<input type="submit" value="登陆">

</sf:form>

标签<sf:form>对应html的form标签，commandName属性用来绑定表单和Model中的对象。因此渲染视图时，需要控制器在model中放置名为user的对象。视图解析器会结合这个对象来渲染表单视图,然后写入响应报文。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，结合表单验证API如果验证失败，Spring会在页面渲染一些错误信息。

#### 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，需要在不同语言环境下显示不同的文本，可以如下配置：

<h1><s:message code="prop.name" /></h1>

Spring在渲染文本时会使用MessageSource来获取prop.name对应的消息。需要在上下文配置MessageSource为bean，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.<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。

# 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) {

// 其他配置

}

}

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));

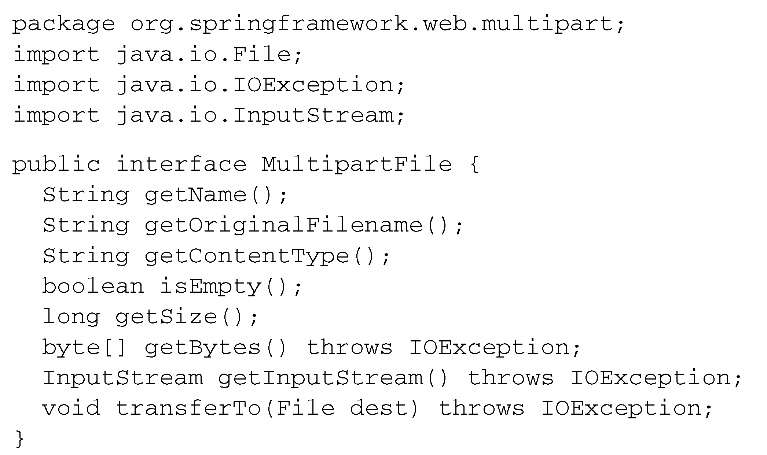
}

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：

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(){}

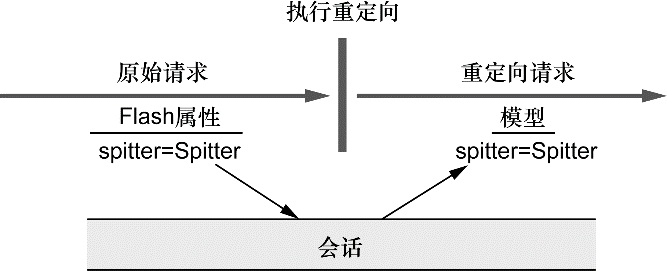
这样的方法会处理该控制器中所有方法抛出该异常后进行处理。

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 JDBC

## JDBC

JDBC定义了java访问数据库的基本接口，其使用连接来操作数据库。一般来说操作一个数据库需要知道数据库的地址、用户名和密码。JDBC将数据库的操作抽象为获取连接、准备语句、读取结果几个步骤，定义好这些操作的接口。具体数据库的操作实现了这些接口，称为驱动。访问不同的数据库，只需要使用不同的驱动即可。

访问数据库的步骤如下：

Class.forName("\*.Driver");

Connection conn = DriverManager.getConnection(url,username,password);

当我们获取了连接以后，就可以使用连接来构造sql对象，然后向数据库发送语句，并接收结果。JDBC的问题在于操作数据库时需要大量的资源管理和异常处理操作，而真正的数据库操作只占代码的小部分。比如一个查询的用例如下：

try {

Connection conn = DriverManager.getConnection(url, u, p);

Statement stat = conn.createStatement();

stat.execute("select \* from a");

stat.close();

conn.close();

} catch (SQLException e) {

e.printStackTrace();

}

注意到我们实际上只需要执行一句sql，但却附加了很多连接关闭、异常处理的代码。实际上这个代码还不是一个很健壮的代码，完整的业务代码还要考虑更多的东西。除此之外还有事务管理、连接池等等设计，因此我们需要一些持久化的框架来减少业务代码的复杂度。实际上本来JDBC的初衷是统一数据库的访问操作，我们也不应该要求JDBC来做过多的设计。

## Spring数据访问设计

Spring将数据库访问中的固定的操作（例如JDBC的获取连接、捕获异常等）抽象为模板，将其中可变的部分抽象为回调。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。

Spring不只是可以简化JDBC代码的复杂度，也可以和其他的持久化框架配合。

## 数据源

在企业级应用中，一般将数据库抽象为数据源DataSource，我们从DataSource获取数据库连接。在Spring中，使用数据源的方式是将数据源配置为bean。

### 数据源连接池

Spring没有提供连接池实现，但是大多数的连接池都可以配置为Spring的数据源，比如c3p0,Apache Commons DBCP。一个DBCP的连接池数据源配置如下：

@Bean

public BasicDataSource datasource() {

BasicDataSource ds = new BasicDataSource();

ds.setDriverClassName("xx");

// 其他配置

return ds;

}

### JDBC数据源

Spring为JDBC数据源提供了三个实现：

DriverManagerDataSource：每次请求都返回一个新的数据库连接

SingleConnectionDataSource：每次请求都返回同一个数据库连接

SimpleDriverDataSource：用于解决特定问题的数据源实现

配置JDBC数据源的方式和配置连接池的方式类似，只要将其配置为bean，设置为相关属性即可。

### 嵌入式数据源

嵌入式数据库是一种运行在内存中的数据库，数据库本身是程序的一部分。嵌入式数据库特别适合用于开发环境，因为每次重启应用或者测试时，数据库都会重新填充内容。配置嵌入式数据库的方式如下：

@Bean

public DataSource datasource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript("classpath:schema.sql")

.build();

}

## 使用SpringJDBC

由于版本更迭，Spring目前常用的JDBC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。JdbcTemplate是JdbcOperations接口的实现，在程序中持有JdbcOperations引用，运行时注入实际的模板对象。



在Dao中如果需要更新插入操作，可以直接执行如下代码：

jdbcOpt.update(

"inster into user (username, password) values (?, ?)",

"abc",

"1234");

查询操作需要将结果集映射到实体，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;

}

//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;

}

}

然后在Dao中查询方法如下：

jdbcOpt.queryForObject("select \* from user where id = ?", new UserRowMapper(), id);

# Spring ORM

持久化框架基于JDBC提供了更多可选的功能，Spring对多个持久化框架提供了支持，不同的持久化框架这些支持是相似的。了解Spring对一种持久化框架的支持，可以很快的切换到另一种持久化框架。

## Spring集成Hibernate

### 声明Session工厂

Hibernate的Session接口提供了数据库访问的功能，标准的Session获取方式是SessionFactory接口的实现类。Spring对Hibernate的支持是实现了SessionFactory接口，在程序中可以将该实现配置为bean。需要注意的是不同版本的Hibernate，这个接口由不同的类来实现。如Hibernate4中，可以使用3中LocalSessionFactoryBean，同时支持xml和注解配置的hibernate。

@Bean

public SessionFactory sessionFactory() {

SessionFactory sf = new LocalSessionFactoryBean()

// 其他设置方法

return sf;

}

### 持久层

在Dao层，我们可以注入SessionFactory，然后使用SessionFactory操作Session来实现持久化。

@Repository

public class UserDaoImpl implements UserDao {

@Autowired

SessionFactory sf;

public Session currentSession() {

return sf.getCurrentSession();

}

public void save(User user) {

return currentSession().save(user);

}

}

## Spring集成MyBatis

MyBatis的核心类是SqlSessionFactory，和Hibernate的SessionFactory类似，该类也用于打开Session，而Session则用于操作数据库。

Spring集成MyBatis的方式是将类SqlSessionFactoryBean配置为bean，然后设置其配置属性，该类实现了InitializingBean接口，因此Spring会调用接口方法来使用配置初始化SqlSessionFactory。同时该类是实现了FactoryBean接口，意思是在配置该类为bean的时候，使用时Spring会按照工厂模式来创建bean，即返回其getObject的对象，而不是这个对象本身。

# Spring事务管理

## Spring事务设计

Spring将事务的管理抽象为PlatformTransactionManager接口：

public interface PlatformTransactionManager {

TransactionStatus getTransaction(

TransactionDefinition definition) throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

}

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接口可以配置事务的一些特征：

隔离(Isolation)：事务和其他的事务的隔离程度。如：该事务能否感知其他事务未提交的写操作。

传播(Propagation)：定义事务的传播行为。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>

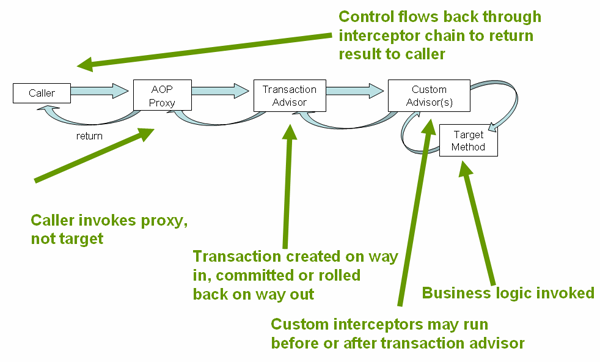
只需要修改配置就能够为程序提供事务支持而不需要修改程序代码。

## 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

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。可以如下配置：

<!-- 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();

}

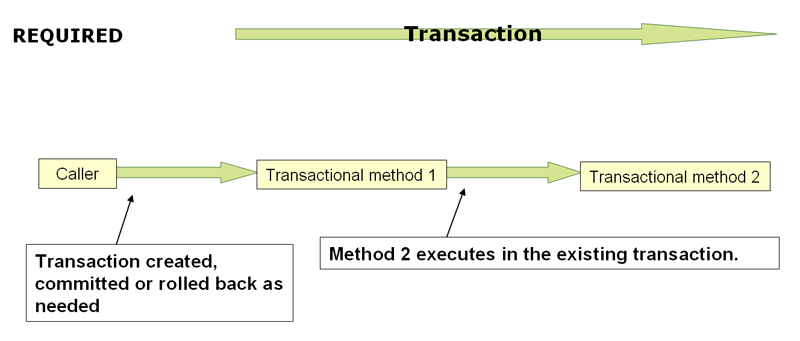
### 事务传播

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。

1. PROPAGATION\_REQUIRED级别

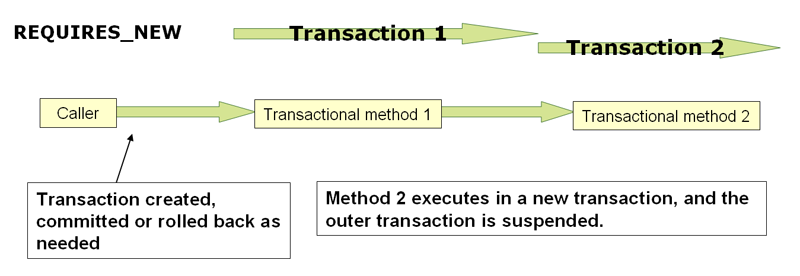
当传播级别设置为PROPAGATION\_REQUIRED时，Spring会为每个方法创建一个逻辑事务作用域。每个逻辑事务作用域都可以单独确定仅回滚状态，外部事务作用域再逻辑上独立于内部事务作用域。当然，在标准REQUIRED类型的情况下，所有这些作用域都将映射到同一物理事务。所以内部事务设置的仅回滚状态确实会影响到外部事物的提交机会（这也是被期望的）。

但是内部事务设置仅回滚时，外部事务并未决定回滚自身，因此回滚时意外的。此时会触发一个UnexpectedRollbackException。



2. PROPAGATION\_REQUIRES\_NEW

该模式下，会为每一个配置了事务的方法设置新的事务，并且其物理事务也是独立的。



## 编程事务管理

略

# 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>