Spring 实践 -AOP

2016/04/07 | 分类： [基础技术](http://www.importnew.com/cat/basic) | [0 条评论](http://www.importnew.com/19041.html#respond) | 标签： [AOP](http://www.importnew.com/tag/aop), [SPRING](http://www.importnew.com/tag/spring)

分享到：**2**

原文出处： [翡青](http://blog.csdn.net/zjf280441589/article/details/50816802)

AOP引介

AOP(*Aspect Oriented Programing*)**面向切面编程**采用**横向抽取机制**,以取代传统的**纵向继承体系**的重复性代码(如性能监控/事务管理/安全检查/缓存实现等).

*横向抽取代码复用: 基于代理技术,在不修改原来代码的前提下,对原有方法进行增强.*

Spring AOP 历史

* 1.2开始, Spring开始支持AOP技术(Spring AOP)  
  Spring AOP使用纯Java实现,不需要专门的编译过程和类加载器,在运行期通过代理方式向目标类**织入**增强代码.
* 2.0之后, 为了简化AOP开发, Spring开始支持AspectJ(一个基于Java的AOP框架)框架.

AOP相关术语

| **术语** | **中文** | **描述** |
| --- | --- | --- |
| *Joinpoint* | 连接点 | 指那些被拦截到的点.在Spring中,这些点指方法(因为Spring只支持方法类型的连接点). |
| *Pointcut* | 切入点 | 指需要(配置)被增强的*Joinpoint*. |
| *Advice* | 通知/增强 | 指拦截到*Joinpoint*后要做的**操作**.通知分为*前置通知/后置通知/异常通知/最终通知/环绕通知*等. |
| *Aspect* | 切面 | 切入点和通知的结合. |
| *Target* | 目标对象 | 需要被代理(增强)的对象. |
| *Proxy* | 代理对象 | 目标对象被AOP **织入** 增强/通知后,产生的对象. |
| *Weaving* | 织入 | 指把**增强/通知**应用到**目标对象**来创建**代理对象**的过程(Spring采用动态代理织入,AspectJ采用编译期织入和类装载期织入). |
| *Introduction* | 引介 | 一种特殊通知,在不修改类代码的前提下,可以在运行期为类动态地添加一些Method/Field(不常用). |

*其他关于AOP理论知识可参考*[*AOP技术研究*](http://www.cnblogs.com/wayfarer/archive/2005/09/21/241038.html)*.*

AOP实现

Spring AOP代理实现有两种:**JDK动态代理**和**Cglib框架动态代理**, JDK动态代理可以参考博客[代理模式](http://blog.csdn.net/zjf280441589/article/details/50411737#t3)的动态代理部分, 在这里仅介绍CGLib框架实现.

cglib 动态代理

cglib(Code Generation Library)是一个开源/高性能/高质量的Code生成类库,可以在运行期动态扩展Java类与实现Java接口.  
cglib比java.lang.reflect.Proxy更强的在于它**不仅可以接管接口类的方法,还可以接管普通类的方法**(cglib[项目](https://github.com/cglib/cglib)).从3.2开始, spring-core包中内置cglib类,因此可以不用添加额外依赖.

* UserDAO(并没有实现接口)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | /\*\*   \* @author jifang   \* @since 16/3/3 上午11:16.   \*/  public class UserDAO {        public void add(Object o) {          System.out.println("UserDAO -> Add: " + o.toString());      }        public void get(Object o) {          System.out.println("UserDAO -> Get: " + o.toString());      }  } |

* CGLibProxyFactory

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | public class CGLibProxyFactory {        private Object target;        public CGLibProxyFactory(Object target) {          this.target = target;      }        private Callback callback = new MethodInterceptor() {            /\*\*           \*           \* @param obj   代理对象           \* @param method    当期调用方法           \* @param args  方法参数           \* @param proxy 被调用方法的代理对象(用于执行父类的方法)           \* @return           \* @throws Throwable           \*/          @Override          public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {                // 前置增强              System.out.println("+ Before Advice ...");                // 执行目标方法              //Object result = method.invoke(target, args);              Object result = proxy.invoke(target, args);                // 后置增强              System.out.println("+ After Advice ...");                return result;          }      };        public Object createProxy() {            // 1. 创建Enhancer对象          Enhancer enhancer = new Enhancer();            // 2. cglib创建代理, 对目标对象创建子对象          enhancer.setSuperclass(target.getClass());            // 3. 传入回调接口, 对目标增强          enhancer.setCallback(callback);            return enhancer.create();      }        public static void main(String[] args) {          UserDAO proxy = (UserDAO) new CGLibProxyFactory(new UserDAO()).createProxy();          proxy.get("hello");          proxy.add("world");      }  } |

AOP小结

* Spring AOP的底层通过JDK/cglib动态代理为目标对象进行横向织入:  
  1) 若目标对象实现了接口,则Spring使用JDK的java.lang.reflect.Proxy代理.  
  2) 若目标对象没有实现接口,则Spring使用cglib库生成目标对象的子类.
* Spring只支持方法连接点,不提供属性连接.
* 标记为final的方法不能被代理,因为无法进行覆盖.
* 程序应优先对针对接口代理,这样便于程序解耦/维护.

Spring AOP

AOP联盟为通知Advice定义了org.aopalliance.aop.Advice接口, Spring在Advice的基础上,根据通知在目标方法的连接点位置,扩充为以下五类:

| **通知** | **接口** | **描述** |
| --- | --- | --- |
| 前置通知 | MethodBeforeAdvice | 在目标方法执行***前***实施增强 |
| 后置通知 | AfterReturningAdvice | …执行***后***实施增强 |
| 环绕通知 | MethodInterceptor | ..执行***前后***实施增强 |
| 异常抛出通知 | ThrowsAdvice | …抛出***异常***后实施增强 |
| 引介通知 | IntroductionInterceptor | 在目标类中***添加新的方法和属性***(少用) |

* 添加Spring的AOP依赖  
  使用Spring的AOP和AspectJ需要在pom.xml中添加如下依赖:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-aop</artifactId>      <version>${spring.version}</version>  </dependency>  <dependency>      <groupId>org.springframework</groupId>      <artifactId>spring-aspects</artifactId>      <version>${spring.version}</version>  </dependency> |

* 定义Target

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*\*   \* @author jifang   \* @since 16/3/3 下午2:50.   \*/  public interface OrderService {        void save();        Integer delete(Integer param);  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class OrderServiceImpl implements OrderService {        @Override      public void save() {          System.out.println("添加...");      }        @Override      public Integer delete(Integer param) {          System.out.println("删除...");          return param;      }  } |

* 定义Advice

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | /\*\*   \* 实现MethodInterceptor接口定义环绕通知   \*   \* @author jifang   \* @since 16/3/6 下午2:54.   \*/  public class ConcreteInterceptor implements MethodInterceptor {        @Override      public Object invoke(MethodInvocation invocation) throws Throwable {          System.out.println("前置通知 -> ");            Object result = invocation.proceed();            System.out.println("<- 后置通知");            return result;      }  } |

Spring手动代理

* 配置代理  
  Spring最原始的AOP支持, 手动指定目标对象与通知(没有使用AOP名称空间).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"         xsi:schemaLocation="http://www.springframework.org/schema/beans    http://www.springframework.org/schema/beans/spring-beans.xsd">        <!-- target -->      <bean id="service" class="com.fq.service.impl.OrderServiceImpl"/>      <!-- advice -->      <bean id="advice" class="com.fq.advice.ConcreteInterceptor"/>        <bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">          <property name="target" ref="service"/>          <property name="interceptorNames" value="advice"/>          <property name="proxyTargetClass" value="false"/>      </bean>  </beans> |

* Client

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations = "classpath:spring/applicationContext.xml")  public class AOPClient {        @Autowired      // 必须指定使用代理对象名称, 否则不予代理      @Qualifier("serviceProxy")      private OrderService service;        @Test      public void client() {          service.save();          service.delete(88);      }  } |

*这种方式的缺陷在于每个Target都必须手动指定ProxyFactoryBean对其代理(不能批量指定),而且这种方式会在Spring容器中存在两份Target对象(代理前/代理后),浪费资源,且容易出错(比如没有指定@Qualifier).*

Spring自动代理 – 引入AspectJ

通过AspectJ引入Pointcut切点定义

* Target/Advice同前
* 定义切面表达式

*通过execution函数定义切点表达式(定义切点的方法切入)   
execution(<访问修饰符> <返回类型><方法名>(<参数>)<异常>)   
如:   
1) execution(public \* \*(..)) # 匹配所有public方法.   
2) execution(\* com.fq.dao.\*(..)) # 匹配指定包下所有类方法(不包含子包)   
3) execution(\* com.fq.dao..\*(..)) # 匹配指定包下所有类方法(包含子包)   
4) execution(\* com.fq.service.impl.OrderServiceImple.\*(..)) # 匹配指定类所有方法   
5) execution(\* com.fq.service.OrderService+.\*(..)) # 匹配实现特定接口所有类方法   
6) execution(\* save\*(..)) # 匹配所有save开头的方法*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"         xmlns:aop="http://www.springframework.org/schema/aop"         xsi:schemaLocation="http://www.springframework.org/schema/beans    http://www.springframework.org/schema/beans/spring-beans.xsd      http://www.springframework.org/schema/aop      http://www.springframework.org/schema/aop/spring-aop.xsd">        <!-- target -->      <bean id="service" class="com.fq.service.impl.OrderServiceImpl"/>      <!-- advice -->      <bean id="advice" class="com.fq.advice.ConcreteInterceptor"/>        <!-- 配置切面 : proxy-target-class确定是否使用CGLIB -->      <aop:config proxy-target-class="true">          <!--              aop:pointcut : 切点定义              aop:advisor: 定义Spring传统AOP的切面,只支持一个pointcut/一个advice              aop:aspect : 定义AspectJ切面的,可以包含多个pointcut/多个advice          -->          <aop:pointcut id="pointcut" expression="execution(\* com.fq.service.impl.OrderServiceImpl.\*(..))"/>          <aop:advisor advice-ref="advice" pointcut-ref="pointcut"/>      </aop:config>    </beans> |

* Client同前

AspectJ AOP

AspectJ是一个基于Java的AOP框架,提供了强大的AOP功能,其他很多AOP框架都借鉴或采纳了AspectJ的一些思想,Spring2.0以后增加了对AspectJ切点表达式支持(如上),并在Spring3.0之后与AspectJ进行了很好的集成.  
在Java领域,AspectJ中的很多语法结构基本上已成为AOP领域的标准, 他定义了如下几类通知类型:

| **通知** | **接口** | **描述** |
| --- | --- | --- |
| 前置通知 | @Before | 相当于BeforeAdvice |
| 后置通知 | @AfterReturning | 相当于AfterReturningAdvice |
| 环绕通知 | @Around | 相当于MethodInterceptor |
| 抛出通知 | @AfterThrowing | 相当于ThrowAdvice |
| 引介通知 | @DeclareParents | 相当于IntroductionInterceptor |
| 最终final通知 | @After | 不管是否异常,该通知都会执行 |

*新版本Spring,建议使用AspectJ方式开发以简化AOP配置.*

AspectJ-XML-AOP

使用AspectJ编写Advice无需实现任何接口,而且可以将多个通知写入一个切面类.

前置通知

* 定义通知

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | /\*\*   \* @author jifang   \* @since 16/3/3 下午5:38.   \*/  public class Aspect {        /\*\*       \* 无返回值       \*/      public void before1() {          System.out.println("前置增强before1");      }        /\*\*       \* 还可以传入连接点参数 JoinPoint       \*       \* @param point       \*/      public void before2(JoinPoint point) {          System.out.printf("前置增强before2 %s%n", point.getKind());      }  } |

* 装配

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"         xmlns:aop="http://www.springframework.org/schema/aop"         xmlns:context="http://www.springframework.org/schema/context"         xsi:schemaLocation="http://www.springframework.org/schema/beans    http://www.springframework.org/schema/beans/spring-beans.xsd      http://www.springframework.org/schema/aop      http://www.springframework.org/schema/aop/spring-aop.xsd      http://www.springframework.org/schema/context      http://www.springframework.org/schema/context/spring-context.xsd">        <context:component-scan base-package="com.fq.service"/>        <!-- 配置切面通知 -->      <bean id="advice" class="com.fq.advice.Aspect"/>        <!-- AOP切面配置 -->      <aop:config>          <aop:aspect ref="advice">              <aop:pointcut id="pointcut" expression="execution(\* com.fq.service.impl.OrderServiceImpl.\*(..))"/>              <aop:before method="before1" pointcut-ref="pointcut"/>              <aop:before method="before2" pointcut-ref="pointcut"/>          </aop:aspect>      </aop:config>    </beans> |

* 前置通知小结
  + 前置通知会保证在目标方法执行前执行;
  + 前置通知默认不能阻止目标方法执行(但如果通知抛出异常,则目标方法无法执行);
  + 可以通过JoinPoint参数获得当前拦截对象和方法等信息.

后置通知

* 定义通知

|  |  |
| --- | --- |
| 1  2  3 | public void afterReturning(JoinPoint point, Object result) {      System.out.printf("后置增强, 结果为 %s%n", result);  } |

* 装配

|  |  |
| --- | --- |
| 1 | <aop:after-returning method="afterReturning" returning="result" pointcut-ref="pointcut"/> |

*后置通知可以获得方法返回值,但在配置文件定义返回值参数名必须与后置通知方法参数名一致(如result).*

环绕通知

* 定义通知

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public Object around(ProceedingJoinPoint point) throws Throwable {      System.out.printf("环绕前置增强 method: %s, args: %s%n", point.toShortString(), Arrays.toString(point.getArgs()));        Object result = point.proceed(point.getArgs());        System.out.printf("环绕后置增强 result: %s%n", result);        return result;  } |

* 装配

|  |  |
| --- | --- |
| 1 | <aop:around method="around" arg-names="point" pointcut-ref="pointcut"/> |

*环绕通知可以实现任何通知的效果, 甚至可以阻止目标方法的执行.*

抛出通知

* 定义通知

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private static final Logger LOGGER = LoggerFactory.getLogger(Aspect.class);    public void afterThrowing(JoinPoint point, Throwable ex) {      String message = new StringBuilder("method ").append(point.getSignature().getName()).append(" error").toString();      System.out.println(message);        LOGGER.error("{},", message, ex);  } |

* 装配

|  |  |
| --- | --- |
| 1 | <aop:after-throwing method="afterThrowing" throwing="ex" pointcut-ref="pointcut"/> |

*throwing属性指定异常对象名, 该名称应和方法定义参数名一致.*

最终通知

* 定义通知

|  |  |
| --- | --- |
| 1  2  3 | public void after(JoinPoint point) {      System.out.println("最终通知, 释放资源");  } |

* 装配

|  |  |
| --- | --- |
| 1 | <aop:after method="after" pointcut-ref="pointcut"/> |

*无论目标方法是否出现异常,该通知都会执行(类似finally代码块, 应用场景为释放资源).*

AspectJ-Annotation-AOP

@AspectJ是AspectJ 1.5新增功能,可以通过JDK注解技术,直接在Bean类中定义切面.  
AspectJ预定义的注解有:@Before/@AfterReturning/@Around/@AfterThrowing/@DeclareParents/@After.描述同前.  
使用AspectJ注解AOP需要在applicationContext.xml文件中开启注解自动代理功能:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"         xmlns:aop="http://www.springframework.org/schema/aop"         xmlns:context="http://www.springframework.org/schema/context"         xsi:schemaLocation="http://www.springframework.org/schema/beans    http://www.springframework.org/schema/beans/spring-beans.xsd      http://www.springframework.org/schema/aop      http://www.springframework.org/schema/aop/spring-aop.xsd      http://www.springframework.org/schema/context      http://www.springframework.org/schema/context/spring-context.xsd">        <!-- 批量扫描@Component -->      <context:component-scan base-package="com.fq"/>      <!-- 启用注解自动代理@Aspect-->      <aop:aspectj-autoproxy/>  </beans> |

* OrderService/Client同前

@Before

* Aspect

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*\*   \* @Aspect: 指定是一个切面   \* @Component: 指定可以被Spring容器扫描到   \*/  @Aspect  @Component  public class CustomAspect {        @Before("execution(\* com.fq.service.impl.OrderServiceImpl.\*(..))")      public void before(JoinPoint point) {          System.out.printf("前置增强before2 %s%n", point.getKind());      }  } |

@AfterReturning

|  |  |
| --- | --- |
| 1  2  3  4 | @AfterReturning(value = "execution(\* com.fq.service.impl.OrderServiceImpl.d\*(..))", returning = "result")  public void afterReturning(JoinPoint point, Object result) {      System.out.printf("后置增强, 结果为 %s%n", result);  } |

@Around

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Around("execution(\* com.fq.service.impl.OrderServiceImpl.\*(..))")  public Object around(ProceedingJoinPoint point) throws Throwable {      long start = System.currentTimeMillis();      Object result = point.proceed(point.getArgs());      long time = System.currentTimeMillis() - start;        System.out.printf("method %s invoke consuming %d ms%n", point.toLongString(), time);        return result;  } |

*如果不调用ProceedingJoinPoint的proceed方法,那么目标方法就不执行了.*

@AfterThrowing

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @AfterThrowing(value = "execution(\* com.fq.service.impl.OrderServiceImpl.\*(..))", throwing = "ex")  public void afterThrowing(JoinPoint point, Throwable ex) {      String message = new StringBuilder("method ").append(point.getSignature().getName()).append(" error").toString();      System.out.println(message);        LOGGER.error("{},", message, ex);  } |

@After

|  |  |
| --- | --- |
| 1  2  3  4 | @After("execution(\* com.fq.service.impl.OrderServiceImpl.\*(..))")  public void after(JoinPoint point) {      System.out.println("最终通知, 释放资源");  } |

@Pointcut定义切点

对于重复的切点,可以使用@Pointcut进行定义, 然后在通知注解内引用.

* 定义切点方法  
  无参/无返回值/方法名为切点名:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*\*   \* @author jifang   \* @since 16/3/4 上午11:47.   \*/  public class OrderServicePointcut {        @Pointcut("execution(\* com.fq.service.impl.OrderServiceImpl.\*(..))")      public void pointcut() {      }  } |

* 引用切点  
  在Advice上像调用方法一样引用切点:

|  |  |
| --- | --- |
| 1  2  3  4 | @After("OrderServicePointcut.pointcut()")  public void after(JoinPoint point) {      System.out.println("最终通知, 释放资源");  } |

*1) 如果切点与切面在同一个类内, 可省去类名前缀;   
2) 当需要通知多个切点时,可以使用||/&&进行连接.*

小结

| **通知** | **描述** |
| --- | --- |
| 前置通知 | 权限控制(少用) |
| 后置通知 | 少用 |
| 环绕通知 | 权限控制/性能监控/缓存实现/事务管理 |
| 异常通知 | 发生异常后,记录错误日志 |
| 最终通知 | 释放资源 |