Spring的AOP

课程安排

- AOP的概述
- AOP 的底层实现
- Spring 的传统AOP
  - 不带切入点的切面
  - 带有切入点的切面
- Spring 的传统AOP的自动代理
  - 基于Bean名称的自动代理
  - 基于切面信息的自动代理

什么是AOP

- AOP Aspect Oriented Programing 面向切面编程
- AOP采取横向抽取机制，取代了传统纵向继承体系重复性代码（性能监视、事务管理、安全检查、缓存）

什么是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

AOP相关术语

Joinpoint(连接点):所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点.

Pointcut(切入点):所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.

Advice(通知/增强):所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

Introduction(引介):引介是一种特殊的通知在不修改类代码的前提下,Introduction可以在运行期为类动态地添加一些方法或Field.

AOP相关术语

Target(目标对象):代理的目标对象

Weaving(织入):是指把增强应用到目标对象来创建新的代理对象的过程.

spring采用动态代理织入，而AspectJ采用编译期织入和类装在期织入

Proxy（代理）:一个类被AOP织入增强后，就产生一个结果代理类

Aspect(切面): 是切入点和通知（引介）的结合

JDK动态代理

```
public class MyJdkProxy implements InvocationHandler {
 private Object object;
 public MyJdkProxy(Object object) {
     this.object = object;
 }
 public Object createProxy() {
     Object proxy = Proxy.newProxyInstance(userDao.getClass()
             .getClassLoader(), userDao.getClass().getInterfaces(), this);
     return proxy;
 }
 @Override
 public Object invoke(Object proxy, Method method, Object[] args)
         throws Throwable {
     if("save".equals(method.getName())){
         System.out.println("==================权限校验===============");
         return method.invoke(object, args);
     }
     return method.invoke(object, args);
 }
}
```

使用CGLIB生成代理

- 对于不使用接口的业务类，无法使用JDK动态代理
- CGlib采用非常底层字节码技术，可以为一个类创建子类，解决无接口代理问题

使用CGLIB生成代理

```
public class MyCglibProxy implements MethodInterceptor{
 private Object object;
 public MyCglibProxy(Object object) {
     this.object = object;
 }
 public Object createProxy(){
     // 1.创建一个 CGLIB 的核心类:
     Enhancer enhancer = new Enhancer();
     // 2.设置父类:
     enhancer.setSuperclass(object.getClass());
     // 3.设置回调:
     enhancer.setCallback(this);
     // 4.生成代理 :
     Object proxy = enhancer.create();
     return proxy;
 }
 public Object intercept(Object proxy, Method method, Object[] args,
         MethodProxy methodProxy) throws Throwable {
     if("update".equals(method.getName())){
         System.out.println("==================权限校验===============");
         return methodProxy.invokeSuper(proxy, args);
     }
     return methodProxy.invokeSuper(proxy, args);
 }
}
```

代理知识总结

- Spring在运行期，生成动态代理对象，不需要特殊的编译器
- Spring AOP的底层就是通过JDK动态代理或CGLib动态代理技术 为目标Bean执行横向织入

  1.若目标对象实现了若干接口，spring使用JDK的java.lang.reflect.Proxy类代理。

  2.若目标对象没有实现任何接口，spring使用CGLIB库生成目标对象的子类。

代理知识总结

- 程序中应优先对接口创建代理，便于程序解耦维护
- 标记为final的方法，不能被代理，因为无法进行覆盖
  - JDK动态代理，是针对接口生成子类，接口中方法不能使用final修饰
  - CGLib 是针对目标类生产子类，因此类或方法 不能使final的
- Spring只支持方法连接点，不提供属性连接

Spring AOP增强类型

- AOP联盟为通知Advice定义了org.aopalliance.aop.Interface.Advice
- Spring按照通知Advice在目标类方法的连接点位置，可以分为5类
  - 前置通知 org.springframework.aop.MethodBeforeAdvice
    - 在目标方法执行前实施增强
  - 后置通知 org.springframework.aop.AfterReturningAdvice
    - 在目标方法执行后实施增强

Spring AOP增强类型

- 环绕通知 org.aopalliance.intercept.MethodInterceptor
  - 在目标方法执行前后实施增强
- 异常抛出通知 org.springframework.aop.ThrowsAdvice
  - 在方法抛出异常后实施增强
- 引介通知 org.springframework.aop.IntroductionInterceptor
  - 在目标类中添加一些新的方法和属性

Spring AOP切面类型

- Advisor：代表一般切面，Advice本身就是一个切面，对目标类所有方法进行拦截
- PointcutAdvisor：代表具有切点的切面，可以指定拦截目标类哪些方法
- IntroductionAdvisor：代表引介切面，针对引介通知而使用切面（不要求掌握）

Advisor切面案例

- ProxyFactoryBean常用可配置属性
  - target : 代理的目标对象
  - proxyInterfaces : 代理要实现的接口
    - 如果多个接口可以使用以下格式赋值

```
<list>
    <value></value>
    ....
</list>
```

Advisor切面案例

- proxyTargetClass：是否对类代理而不是接口，设置为true时，使用CGLib代理
- interceptorNames：需要织入目标的Advice
- singleton：返回代理是否为单实例，默认为单例
- optimize：当设置为true时，强制使用CGLib

PointcutAdvisor 切点切面

- 使用普通Advice作为切面，将对目标类所有方法进行拦截，不够灵活，在实际开发中常采用 带有切点的切面
- 常用PointcutAdvisor 实现类
  - DefaultPointcutAdvisor 最常用的切面类型，它可以通过任意Pointcut和Advice 组合定义切面
  - JdkRegexpMethodPointcut 构造正则表达式切点

自动创建代理

- 前面的案例中，每个代理都是通过ProxyFactoryBean织入切面代理，在实际开发中，非常多的Bean每个都配置ProxyFactoryBean开发维护量巨大
- 解决方案：自动创建代理
  - BeanNameAutoProxyCreator 根据Bean名称创建代理
  - DefaultAdvisorAutoProxyCreator 根据Advisor本身包含信息创建代理
  - AnnotationAwareAspectJAutoProxyCreator 基于Bean中的AspectJ注解进行自动代理

BeanNameAutoProxyCreator 举例

- 对所有以DAO结尾Bean所有方法使用代理

```
* 只留下目标类和通知:
    <!-- 配置目标类: -->
    <bean id="productDao" class="com.imooc.spring.demo3.ProductDaoImpl"/>
    <bean id="customerDao" class="com.imooc.spring.demo4.CustomerDao"/>

    <!-- 配置通知:(前置通知) -->
    <bean id="beforeAdvice" class="com.imooc.spring.demo3.MyBeforeAdvice"/>
    <!-- 配置通知:(环绕通知) -->
    <bean id="myAroundAdvice" class="com.imooc.spring.demo4.MyAroundAdvice"/>

* 通过配置完整自动代理:
    <!-- 配置基于 Bean 名称的自动代理 -->
    <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <!-- 配置 Bean 名称 -->
        <property name="beanNames" value="*Dao"/>
        <!-- 配置拦截的名称 -->
        <property name="interceptorNames" value="beforeAdvice"/>
    </bean>
```

DefaultAdvisorAutoProxyCreator 举例

- 配置环绕代理案例

```
* 只留下目标类和增强:
    <!-- 配置目标类: -->
    <bean id="productDao" class="com.imooc.spring.demo3.ProductDaoImpl"/>
    <bean id="customerDao" class="com.imooc.spring.demo4.CustomerDao"/>

    <!-- 配置通知:(前置通知) -->
    <bean id="beforeAdvice" class="com.imooc.spring.demo3.MyBeforeAdvice"/>
    <!-- 配置通知:(环绕通知) -->
    <bean id="myAroundAdvice" class="com.imooc.spring.demo4.MyAroundAdvice"/>

* 配置切面:
    <!-- 配置切面 -->
    <bean id="myAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <!-- 表达式 -->
        <property name="pattern" value="com\.imooc\.spring\.demo4\.CustomerDao\.save"/>
        <!-- 配置增强 -->
        <property name="advice" ref="myAroundAdvice"/>
    </bean>

* 配置生成代理:
    <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

课程总结