Spring AOP API

这是Spring1.2的历史用法，现在（V4.0）仍然支持
这是SpringAOP基础，不得不了解
现在的用法也是基于历史的，只是更简便了

只需要了解一下过一下就好了

现在相对以前，是有很大的简化

这是第一个

Pointcut

- 实现之一：NameMatchMethodPointcut，根据方法名字进行匹配
- 成员变量：mappedNames，匹配的方法名集合

```
<bean id="pointcutBean" class="org.springframework.aop.support.NameMatchMethodPointcut">
  <property name="mappedNames">
    <list>
      <value>sa*</value>
    </list>
  </property>
</bean>
```

list是指表示当前这个属性，是一个集合，这个可以写多个，value的意思是sa开头的这种所有的方法，*是通配符的方式

Before advice

- 一个简单的通知类型
- 只是在进入方法之前被调用，不需要MethodInvocation对象
- 前置通知可以在连接点执行之前插入自定义行为，但不能改变返回值

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {

    private int count;

    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
```

下面贴代码：

```
public class MoocBeforeAdvice implements  MethodBeforeAdvice {

@Override
public void before(Method method, Object[] args, Object target)
throws Throwable {
      System.out.println("MoocBeforeAdvice : " + method.getName() + "      " +
            target.getClass().getName());
}

}
```

Throw advice

下面贴代码：

```
public class MoocThrowsAdvice  implements  ThrowsAdvice {
public  void afterThrowing(Exception e) throws Throwable{
      System.out.println("MoocThrowsAdvice   afterThrowing  1");
}

public  void afterThrowing(Method method,Object[] objects,Object target,Exception e)
 throws
Throwable{
     System.out.println("MoocThrowsAdvice  afterThrowing2  "+method.getName()+"  "+t
arget.getClass().getName());
}

}
```

After Returning advice

After Returning advice

- 后置通知必须实现org.springframework.aop.AfterReturningAdvice接口

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

- 可以访问返回值（但不能进行修改）、被调用的方法、方法的参数和目标
- 如果抛出异常，将会抛出拦截器链，替代返回值

下面贴代码：

```
public class MoocAfterReturningAdvice  implements  AfterReturningAdvice {

@Override
public void afterReturning(Object returnValue, Method method,
Object[] args, Object target) throws Throwable {
      System.out.println(" MoocAfterReturningAdvice  afterReturning"+method+"  " +
"  "+target.getClass().getName()+"    "+ returnValue);
}

}
```

Interception around advice

- Spring的切入点模型使得切入点可以独立与advice重用，以针对不同的advice可以使用相同的切入点

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

下面贴代码：

```
public class MoocMethodInterceptor implements MethodInterceptor {

@Override
public Object invoke(MethodInvocation invocation) throws Throwable {
      System.out.println("MoocMethodInterceptor :"+invocation.getMethod()+"   "
+invocation.getStaticPart().getClass().getName());
Object obj=invocation.proceed();
System.out.println("MethodInvocation2 :"+obj);
      return  obj;
}
```

```
}
```

Introduction advice

- Spring把引入通知作为一种特殊的拦截通知
- 需要IntroductionAdvisor和IntroductionInterceptor
- 仅适用于类，不能和任何切入点一起使用

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

    Class[] getInterfaces();
```

Introduction advice

- 一个Spring test suite的例子
- 如果调用lock()方法，希望所有的setter方法抛出LockedException异常（如使物体不可变，AOP典型例子）
- 需要一个完成繁重任务的IntroductionInterceptor，这种情况下，可以使用org.springframework.aop.support.DelegatingIntroductionInterceptor

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

Introduction advice

```
public class LockMixin extends DelegatingIntroductionInterceptor implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0) {
            throw new LockedException();
        }
        return super.invoke(invocation);
    }

}
```

下面贴代码：

```
public interface Lockable {

void lock();

   void unlock();

   boolean locked();

}
```

```
public class LockMixin extends DelegatingIntroductionInterceptor implements Lockable
 {

private static final long serialVersionUID = 6943163819932660450L;

   private boolean locked;

    public void lock() {
this.locked = true;
}

public void unlock() {
this.locked = false;
}

public boolean locked() {
return this.locked;
}

public Object invoke(MethodInvocation invocation) throws Throwable {
if (locked() && invocation.getMethod().getName().indexOf("set") == 0) {
throw new RuntimeException();
}
return super.invoke(invocation);
}

}
```

这个就是说呢，我们继承了这个

```
DelegatingIntroductionInterceptor
```

类，然后我们就可以对我们作用的地方，进行查看，如果是调用lock方法，所有的setter都抛出异常

就是在下面的地方，有获取到方法名，如果是set，就抛出异常，这个父类的作用是里面有

```
MethodInvocation
```

下面实现一个方法，来进行我们的目标类进行环绕通知。然后按照我们想的来

- introduction advisor比较简单，持有独立的LockMixin实例

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

Advisor API in Spring

- Advisor是仅包含一个切入点表达式关联的单个通知的方面
- 除了introductions，advisor可以用于任何通知
- org.springframework.aop.support.DefaultPointcutAdvisor是最常用的advisor类，它可以与MethodInterceptor, BeforeAdvice或者ThrowsAdvice一起使用
- 它可以混合在Spring同一个AOP代理的advisor和advice

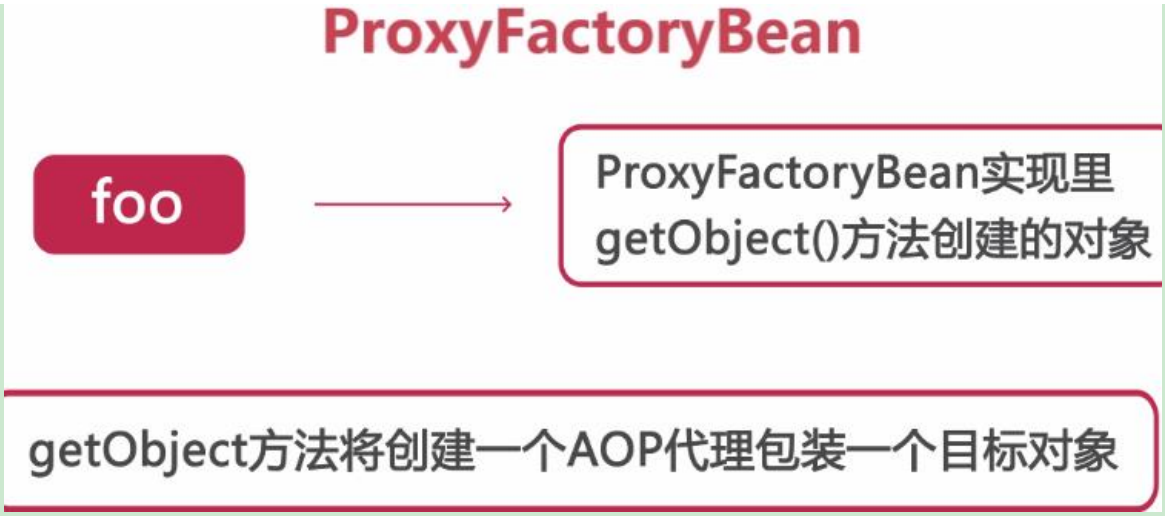
```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

private static final long serialVersionUID = -171332350782163120L;

   public LockMixinAdvisor() {
super(new LockMixin(), Lockable.class);
}
}
```

ProxyFactoryBean

- 创建Spring AOP代理的基本方法是使用org.springframework.aop.framework.ProxyFactoryBean
- 这可以完全控制切入点和通知(advice)以及他们的顺序



- 使用ProxyFactoryBean或者其它IoC相关类来创建AOP代理的最重要好处是通知和切入点也可以由IoC来管理
- 被代理类没有实现任何接口，使用CGLIB代理，否则JDK代理
- 通过设置proxyTargetClass为true，可强制使用CGLIB
- 如果目标类实现了一个（或者多个）接口，那么创建代理的类型将依赖ProxyFactoryBean的配置
- 如果ProxyFactoryBean的proxyInterfaces属性被设置为一个或者多个全限定接口名，基于JDK的代理将被创建

上面这个ProxyFactoryBean的配置，是配置其中的proxyInterfaces的属性

关于

CGLIB(Code Generation Library)是一个开源项目！

是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。Hibernate用它来实现PO(Persistent Object 持久化对象)字节码的动态生成。

- 如果ProxyFactoryBean的proxyInterfaces属性没有被设置，但是目标类实现了一个（或者更多）接口，那么ProxyFactoryBean将自动检测到这个目标类已经实现了至少一个接口，创建一个基于JDK的代理。

下面的是Proxying interfaces

是一种实现的方法

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name" value="Tony"/>
    <property name="age" value="51"/>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>

    <property name="target" ref="personTarget"/>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref bean="person"/></property>
```

为target指定类的时候不用ref 而是在target这个property里定义一个bean 改bean指向target要指定的类，这样阻止直接访问到被指定的那个类。

```
1.  <property name="interceptorNames">
```

```
            <list>
                <value>decorateBeforeAdvice</value>
                <value>transactionAdvisor</value>
                <value>roleDecorateInterceptor</value>
            </list>
        </property>
```

interceptorNames 属性的list集合里面只能放 通知/通知者 。
通知/通知者被称为拦截者（拦截器）。
下面贴代码:

```
<bean id="moocBeforeAdvice" class=" aop.api.MoocBeforeAdvice"></bean>
<!--//这是配置了before-->
<bean id="moocAfterReturningAdvice" class="aop.api.MoocAfterReturningAdvice"></bean>
<!--//这是配置了返回-->
<bean id="moocMethodInterceptor" class="aop.api.MoocMethodInterceptor"></bean>
<!--这是环绕方式-->
<bean id="moocThrowsAdvice" class=" aop.api.MoocThrowsAdvice"></bean>
<!--这个是出错的情况下-->
<bean id="bizLogicImplTarget" class="aop.api.BizLogicImpl"/>
<!--这是下面我们对应生成的需要target类-->
<bean id="pointcutBean" class="org.springframework.aop.support.NameMatchMethodPointcut">
        <property name="mappedNames">
            <list>
                <value>sa*</value>
            </list>
        </property>
    </bean>
<!--

pointcutBean 切入点 如果满足的话，就是下面的情况，触发moocBeforAdvice&ndash

-->
<bean id="defaultAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
          <property name="advice" ref="moocBeforeAdvice" />
          <property name="pointcut" ref="pointcutBean" />
       </bean>

<!--这指定默认的前面实现的地方 如果有满足pointcutBean这个bean配置的情况下-->


    <bean id="bizLogicImpl" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target">
           <ref bean="bizLogicImplTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>defaultAdvisor</value>
                <value>moocAfterReturningAdvice</value>
                <value>moocMethodInterceptor</value>
                <value>moocThrowsAdvice</value>
            </list>
        </property>
<!--这是对应的代理工厂模式-->
</bean>
</beans>
```

这里我们可以指定一个属性 就是proxyInterfaces这个属性

```
<propertyname="proxyInterfaces">
  <value>aop.api.BizLogic</value>
</property>
```

这个属性可以指定目标类的接口，然后默认是生成JDK代理
如果不是目标类的接口，就会出错

Proxying interfaces

使用匿名内部bean来隐藏目标和代理之间的区别

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>
    <!-- Use inner bean, not local reference to target -->
    <property name="target">
        <bean class="com.mycompany.PersonImpl">
            <property name="name" value="Tony"/>
            <property name="age" value="51"/>
        </bean>
    </property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
```

这个的意思是，不用引用一个bean，可以直接在内部定义一个Bean
下面贴代码：

```
<property name="target">
    <bean    class="aop.api.BizLogicImpl"/>
</property>
```

下面是之前的：

```
<property name="target">
   <ref bean="bizLogicImplTarget"/>
</property>
```

在外面定义一个Bean的话，有一个坏处，可以直接get到bean对象 可以被别人处理过。所以这种方式的好处，可以直接用。

- 前面的例子中如果没有Person接口，这种情况下Spring会使用CGLIB代理，而不是JDK动态代理
- 如果想，可以强制在任何情况下使用CGLIB，即使有接口
- CGLIB代理的工作原理是在运行时生成目标类的子类，Spring配置这个生成的子类委托方法调用到原来的目标
- 子类是用来实现Decorator模式，织入通知

CGLIB的代理对用户是透明的，需要注意：

- final方法不能被通知，因为它们不能被覆盖
- 不用把CGLIB添加到classpath中，在Spring3.2中，CGLIB被重新包装并包含在Spring核心的JAR（即基于CGLIB的AOP就像JDK动态代理一样“开箱即用”）

这个的意思是不用再手工的去引用CGLIB代理了，因为已经包含在了Spring中

使用*global* advisors

- 用*做通配，匹配所有拦截器加入通知链

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="service"/>
    <property name="interceptorNames">
        <list>
            <value>global*</value>
        </list>
    </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

使用global advisors 用*做通配符 使用这个的时候必须得是拦截器，**如果使advice是不管用的**

- 使用父子bean定义，以及内部bean定义，可能会带来更清洁和更简洁的代理定义（抽象属性标记父bean定义为抽象的这样它不能被实例化）

```
<bean id="txProxyTemplate" abstract="true"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

下面来简化的

```
<bean id="myService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MyServiceImpl">
        </bean>
    </property>
</bean>
```

```
<bean id="mySpecialService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MySpecialServiceImpl">
        </bean>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="store*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

使用ProxyFactory

- 使用Spring AOP而不必依赖于Spring IoC

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addAdvice(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

- 大多数情况下最佳实践是用IoC容器创建AOP代理
- 虽然可以硬编码方式实现，但是Spring推荐使用配置或注解方式实现

自动代理 使用**auto-proxy** 这个角色自动代理 简化配置与开发

- Spring也允许使用“自动代理”的bean定义，它可以自动代理选定的bean，这是建立在Spring的“bean post processor”功能基础上的（在加载bean的时候就可以修改）
- BeanNameAutoProxyCreator

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames" value="jdk*,onlyJdk"/>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

- DefaultAdvisorAutoProxyCreator，当前IoC容器中自动应用，不用显示声明引用advisor的bean定义

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
    <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

如果在IOC容器中声明了上面的第一行，DefaultAdvisorAutoProxyCreator这个类的话，它会在当前的IOC容器中，自动的应用，来达到代理的效果，就是不用显式声明advisor的bean的定义