回首页

Spring AOP 使用介绍，从前世到今生

创建时间: 2018-06-19 00:00:00

前面写过 Spring IOC 的源码分析，很多读者希望可以出一个 Spring AOP 的源码分析，不过 Spring AOP 的源码还是比较多的，写出来不免篇幅会大些。

本文不介绍源码分析，而是介绍 Spring AOP 中的一些概念，以及它的各种配置方法，涵盖了 Spring AOP 发展到现在出现的全部 3 种配置方式。

由于 Spring 强大的向后兼容性，实际代码中往往会出现很多配置混杂的情况，而且居然还能工作，本文希望帮助大家理清楚这些知识。

本文使用的测试源码已上传到 Github: hongjiev/spring-aop-learning。

目录：

AOP, AspectJ, Spring AOP

我们先来把它们的概念和关系说说清楚。

AOP 要实现的是在我们原来写的代码的基础上，进行一定的包装，如在方法执行前、方法返回后、方法抛出异常后等地方进行一定的拦截处理或者叫增强处理。

AOP 的实现并不是因为 Java 提供了什么神奇的钩子，可以把方法的几个生命周期告诉我们，而是我们要实现一个代理，实际运行的实例其实是生成的代理类的实例。

作为 Java 开发者，我们都很熟悉 **AspectJ** 这个词，甚至于我们提到 AOP 的时候，想到的往往就是 AspectJ，即使你可能不太懂它是怎么工作的。这里，我们把 AspectJ 和 Spring AOP 做个简单的对比：

Spring AOP：

- 它基于动态代理来实现。默认地，如果使用接口的，用 JDK 提供的动态代理实现，如果没有接口，使用 CGLIB 实现。大家一定要明白背后的意思，包括什么时候会不用 JDK 提供的动态代理，而用 CGLIB 实现。

- Spring 3.2 以后，spring-core 直接就把 CGLIB 和 ASM 的源码包括进来了，这也是为什么我们不需要显式引入这两个依赖
- Spring 的 IOC 容器和 AOP 都很重要，Spring AOP 需要依赖于 IOC 容器来管理。
- 如果你是 web 开发者，有些时候，你可能需要的是一个 Filter 或一个 Interceptor，而不一定是 AOP。
- Spring AOP 只能作用于 Spring 容器中的 Bean，它是使用纯粹的 Java 代码实现的，只能作用于 bean 的方法。
- Spring 提供了 AspectJ 的支持，后面我们会单独介绍怎么使用，一般来说我们用**纯的** Spring AOP 就够了。
- 很多人会对比 Spring AOP 和 AspectJ 的性能，Spring AOP 是基于代理实现的，在容器启动的时候需要生成代理实例，在方法调用上也会增加栈的深度，使得 Spring AOP 的性能不如 AspectJ 那么好。

AspectJ：

- AspectJ 出身也是名门，来自于 Eclipse 基金会，link：https://www.eclipse.org/aspectj
- 属于静态织入，它是通过修改代码来实现的，它的织入时机可以是：
  - Compile-time weaving：编译期织入，如类 A 使用 AspectJ 添加了一个属性，类 B 引用了它，这个场景就需要编译期的时候就进行织入，否则没法编译类 B。
  - Post-compile weaving：也就是已经生成了 .class 文件，或已经打成 jar 包了，这种情况我们需要增强处理的话，就要用到编译后织入。
  - **Load-time weaving**：指的是在加载类的时候进行织入，要实现这个时期的织入，有几种常见的方法。1、自定义类加载器来干这个，这个应该是最容易想到的办法，在被织入类加载到 JVM 前去对它进行加载，这样就可以在加载的时候定义行为了。2、在 JVM 启动的时候指定 AspectJ 提供的 agent：`-javaagent:xxx/xxx/aspectjweaver.jar`。
- AspectJ 能干很多 Spring AOP 干不了的事情，它是 **AOP 编程的完全解决方案**。Spring AOP 致力于解决的是企业级开发中最普遍的 AOP 需求（方法织入），而不是力求成为一个像 AspectJ 一样的 AOP 编程完全解决方案。
- 因为 AspectJ 在实际代码运行前完成了织入，所以大家会说它生成的类是没有额外运行时开销的。

- ~~很快我会专门写一篇文章介绍 AspectJ 的使用，以及怎么在 Spring 应用中使用 AspectJ。~~

  已成文：https://www.javadoop.com/post/aspectj

AOP 术语解释

在这里，不准备解释那么多 AOP 编程中的术语了，我们碰到一个说一个吧。

Advice、Advisor、Pointcut、Aspect、Joinpoint 等等。

Spring AOP

首先要说明的是，这里介绍的 Spring AOP 是纯的 Spring 代码，和 AspectJ 没什么关系，但是 Spring 延用了 AspectJ 中的概念，包括使用了 AspectJ 提供的 jar 包中的注解，但是不依赖于其实现功能。

> 后面介绍的如 @Aspect、@Pointcut、@Before、@After 等注解都是来自于 AspectJ，但是功能的实现是纯 Spring AOP 自己实现的。

下面我们来介绍 Spring AOP 的使用方法，先从最简单的配置方式开始说起，这样读者想看源码也会比较容易。

目前 Spring AOP 一共有三种配置方式，Spring 做到了很好地向下兼容，所以大家可以放心使用。

- Spring 1.2 **基于接口的配置**：最早的 Spring AOP 是完全基于几个接口的，想看源码的同学可以从这里起步。
- Spring 2.0 **schema-based 配置**：Spring 2.0 以后使用 XML 的方式来配置，使用 命名空间 `<aop />`
- Spring 2.0 **@AspectJ 配置**：使用注解的方式来配置，这种方式感觉是最方便的，还有，这里虽然叫做 `@AspectJ`，但是这个和 AspectJ 其实没啥关系。

Spring 1.2 中的配置

这节我们将介绍 Spring 1.2 中的配置，这是最古老的配置，但是由于 Spring 提供了很好的向后兼容，以及很多人根本不知道什么配置是什么版本的，以及是否有更新更好的配置方法替代，所以还是会有很多代码是采用这种古老的配置方式的，这里说的古老并没有贬义的意思。

下面用一个简单的例子来演示怎么使用 Spring 1.2 的配置方式。

首先，我们先定义两个接口 `UserService` 和 `OrderService`，以及它们的实现类 `UserServiceImpl` 和 `OrderServiceImpl`：

```
public interface UserService {

    User createUser(String firstName, String lastName, int age);

    User queryUser();
}
```

```
public class UserServiceImpl implements UserService {

    private static User user = null;

    @Override
    public User createUser(String firstName, String lastName, int age) {
        user = new User();
        user.setFirstName(firstName);
        user.setLastName(lastName);
        user.setAge(age);
        return user;
    }

    @Override
    public User queryUser() {
        return user;
    }
}
```

```
public interface OrderService {

    Order createOrder(String username, String product);

    Order queryOrder(String username);
}
```

```
public class OrderServiceImpl implements OrderService {

    private static Order order = null;

    @Override
    public Order createOrder(String username, String product) {
        order = new Order();
        order.setUsername(username);
        order.setProduct(product);
        return order;
    }

    @Override
    public Order queryOrder(String username) {
        return order;
    }
}
```

接下来，我们定义两个 advice，分别用于拦截**方法执行前**和**方法返回后**：

> advice 是我们接触的第一个概念，记住它是干什么用的。

```
public class LogArgsAdvice implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, @Nullable Object target) throws Throwable {
        System.out.println("准备执行方法: " + method.getName() + ", 参数列表: " + Arrays.toString(args));
    }
}
```

```
public class LogResultAdvice implements AfterReturningAdvice {

    @Override
    public void afterReturning(Object returnValue, Method method, Object[] args, Object target)
            throws Throwable {
        System.out.println("方法返回: " + returnValue);
    }
}
```

上面的两个 Advice 分别用于方法调用前输出参数和方法调用后输出结果。

现在可以开始配置了，我们配置一个名为 spring_1_2.xml 的文件：

```
    <bean id="userServiceImpl" class="com.javadoop.springaoplearning.service.imple.UserServiceImpl"/>
    <bean id="orderServiceImpl" class="com.javadoop.springaoplearning.service.imple.OrderServiceImpl"/>

    <!--定义两个 advice-->
    <bean id="logArgsAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogArgsAdvice"/>
    <bean id="logResultAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogResultAdvice"/>

    <bean id="userServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <!--代理的接口-->
        <property name="proxyInterfaces">
            <list>
                <value>com.javadoop.springaoplearning.service.UserService</value>
            </list>
        </property>
        <!--代理的具体实现-->
        <property name="target" ref="userServiceImpl"/>

        <!--配置拦截器，这里可以配置 advice、advisor、interceptor，这里先介绍 advice-->
        <property name="interceptorNames">
            <list>
                <value>logArgsAdvice</value>
                <value>logResultAdvice</value>
            </list>
        </property>
    </bean>

    <!--==========================================-->
    <!--同理，我们也可以配置一个 orderServiceProxy.....-->
    <!--==========================================-->
```

接下来，我们跑起来看看：

```
public class Spring_1_2_Application {

    public static void main(String[] args) {

        // 启动 Spring 的 IOC 容器
        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring_1_2.xml");

        // 我们这里要取 AOP 代理: userServiceProxy, 这非常重要
        UserService userService = (UserService) context.getBean("userServiceProxy");

        userService.createUser("Tom", "Cruise", 55);
        userService.queryUser();
    }
}
```

查看输出结果：

```
准备执行方法：createUser, 参数列表：[Tom, Cruise, 55]
方法返回：User{firstName='Tom', lastName='Cruise', age=55, address='null'}
准备执行方法：queryUser, 参数列表：[]
方法返回：User{firstName='Tom', lastName='Cruise', age=55, address='null'}
```

从结果可以看到，对 UserService 中的两个方法都做了前、后拦截。这个例子理解起来应该非常简单，就是一个代理实现。

> 代理模式需要一个接口、一个具体实现类，然后就是定义一个代理类，用来包装实现类，添加自定义逻辑，在使用的时候，需要用代理类来生成实例。

此中方法有个致命的问题，如果我们需要拦截 OrderService 中的方法，那么我们还需要定义一个 OrderService 的代理。如果还要拦截 PostService，得定义一个 PostService 的代理.....

而且，我们看到，我们的拦截器的粒度只控制到了类级别，类中所有的方法都进行了拦截。接下来，我们看看怎么样**只拦截特定的方法**。

在上面的配置中，配置拦截器的时候，interceptorNames 除了指定为 Advice，是还可以指定为 Interceptor 和 Advisor 的。

这里我们来理解 **Advisor** 的概念，它也比较简单，**它内部需要指定一个 Advice**，Advisor 决定该拦截哪些方法，拦截后需要完成的工作还是内部的 Advice 来做。

它有好几个实现类，这里我们使用实现类 **NameMatchMethodPointcutAdvisor** 来演示，从名字上就可以看出来，它需要我们给它提供方法名字，这样符合该配置的方法才会做拦截。

```
    <bean id="userServiceImpl" class="com.javadoop.springaoplearning.service.imple.UserServiceImpl"/>
    <bean id="orderServiceImpl" class="com.javadoop.springaoplearning.service.imple.OrderServiceImpl"/>

    <!--定义两个 advice-->
    <bean id="logArgsAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogArgsAdvice"/>
    <bean id="logResultAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogResultAdvice"/>

    <!--定义一个只拦截queryUser方法的 advisor-->
    <bean id="logCreateAdvisor" class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
        <!--advisor 实例的内部会有一个 advice-->
        <property name="advice" ref="logArgsAdvice" />
        <!--只有下面这两个方法才会被拦截-->
        <property name="mappedNames" value="createUser,createOrder" />
    </bean>

    <bean id="userServiceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
        <!--代理的接口-->
        <property name="proxyInterfaces">
            <list>
                <value>com.javadoop.springaoplearning.service.UserService</value>
            </list>
        </property>
        <!--代理的具体实现-->
        <property name="target" ref="userServiceImpl"/>

        <!--配置拦截器，这里可以配置 advice、advisor、interceptor-->
        <property name="interceptorNames">
            <list>
                <value>logCreateAdvisor</value>
            </list>
        </property>
    </bean>

    <!--=========================================-->
    <!--同理，我们也可以配置一个 orderServiceProxy......-->
    <!--=========================================-->
```

> 我们可以看到，userServiceProxy 这个 bean 配置了一个 advisor，advisor 内部有一个 advice。advisor 负责匹配方法，内部的 advice 负责实现方法包装。
>
> 注意，这里的 mappedNames 配置是可以指定多个的，用逗号分隔，可以是不同类中的方法。相比直接指定 advice，advisor 实现了更细粒度的控制，因为在这里配置 advice 的话，所有方法都会被拦截。

```
public class Spring_1_2_Advisor_Application {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring_1_2_advisor.xml");

        UserService userService = (UserService) context.getBean("userServiceProxy");

        userService.createUser("Tom", "Cruise", 55);
        userService.queryUser();
    }
}
```

输出结果如下，只有 createUser 方法被拦截：

```
准备执行方法：createUser, 参数列表：[Tom, Cruise, 55]
```

到这里，我们已经了解了 **Advice** 和 **Advisor** 了，前面也说了还可以配置 **Interceptor**。

对于 Java 开发者来说，对 Interceptor 这个概念肯定都很熟悉了，这里就不做演示了，贴一下实现代码：

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        // 执行 真实实现类 的方法
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

上面，我们介绍完了 **Advice**、**Advisor**、**Interceptor** 三个概念，相信大家应该很容易就看懂它们了。

它们有个共同的问题，那就是我们得为每个 bean 都配置一个代理，之后获取 bean 的时候需要获取这个代理类的 bean 实例（如 `(UserService) context.getBean("userServiceProxy")`），这显然非常不方便，不利于我们之后要使用的自动根据类型注入。下面介绍 autoproxy 的解决方案。

autoproxy：从名字我们也可以看出来，它是实现自动代理，也就是说当 Spring 发现一个 bean 需要被切面织入的时候，Spring 会自动生成这个 bean 的一个代理来拦截方法的执行，确保定义的切面能被执行。

这里强调**自动**，也就是说 Spring 会自动做这件事，而不用像前面介绍的，我们需要显式地指定代理类的 bean。

我们去掉原来的 **ProxyFactoryBean** 的配置，改为使用 **BeanNameAutoProxyCreator** 来配置：

```xml
    <bean id="userServiceImpl" class="com.javadoop.springaoplearning.service.imple.UserServiceImpl"/>
    <bean id="orderServiceImpl" class="com.javadoop.springaoplearning.service.imple.OrderServiceImpl"/>

    <!--定义两个 advice-->
    <bean id="logArgsAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogArgsAdvice"/>
    <bean id="logResultAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogResultAdvice"/>

    <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="interceptorNames">
            <list>
                <value>logArgsAdvice</value>
                <value>logResultAdvice</value>
            </list>
        </property>
        <property name="beanNames" value="*ServiceImpl" />
    </bean>
```

配置很简单，beanNames 中可以使用正则来匹配 bean 的名字。这样配置出来以后，userServiceBeforeAdvice 和 userServiceAfterAdvice 这两个拦截器就不仅仅可以作用于 UserServiceImpl 了，也可以作用于 OrderServiceImpl、PostServiceImpl、ArticleServiceImpl..... 等等，也就是说不再是配置某个 bean 的代理了。

> 注意，这里的 InterceptorNames 和前面一样，也是可以配置成 Advisor 和 Interceptor 的。

然后我们修改下使用的地方：

```java
public class Spring_1_2_BeanNameAutoProxy_Application {

    public static void main(String[] args) {

        // 启动 Spring 的 IOC 容器
        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring_1_2_BeanNameAutoProxy.xml");

        // 注意这里，不再需要根据代理找 bean
        UserService userService = context.getBean(UserService.class);
        OrderService orderService = context.getBean(OrderService.class);

        userService.createUser("Tom", "Cruise", 55);
        userService.queryUser();

        orderService.createOrder("Leo", "随便买点什么");
        orderService.queryOrder("Leo");
    }
}
```

发现没有，**我们在使用的时候，完全不需要关心代理了**，直接使用原来的类型就可以了，这是非常方便的。

输出结果就是 OrderService 和 UserService 中的每个方法都得到了拦截：

```
准备执行方法：createUser, 参数列表：[Tom, Cruise, 55]
方法返回：User{firstName='Tom', lastName='Cruise', age=55, address='null'}
准备执行方法：queryUser, 参数列表：[]
方法返回：User{firstName='Tom', lastName='Cruise', age=55, address='null'}
准备执行方法：createOrder, 参数列表：[Leo, 随便买点什么]
方法返回：Order{username='Leo', product='随便买点什么'}
```

```
准备执行方法：queryOrder, 参数列表：[Leo]
方法返回：Order{username='Leo', product='随便买点什么'}
```

到这里，是不是发现 BeanNameAutoProxyCreator 非常好用，它需要指定被拦截类名的模式(如 *ServiceImpl)，它可以配置多次，这样就可以用来匹配不同模式的类了。

另外，在 BeanNameAutoProxyCreator 同一个包中，还有一个非常有用的类 **DefaultAdvisorAutoProxyCreator**，比上面的 BeanNameAutoProxyCreator 还要方便。

之前我们说过，advisor 内部包装了 advice，advisor 负责决定拦截哪些方法，内部 advice 定义拦截后的逻辑。所以，仔细想想其实就是只要让我们的 advisor 全局生效就能实现我们需要的自定义拦截功能、拦截后的逻辑处理。

> BeanNameAutoProxyCreator 是自己匹配方法，然后交由内部配置 advice 来拦截处理；
>
> 而 DefaultAdvisorAutoProxyCreator 是让 ioc 容器中的所有 advisor 来匹配方法，advisor 内部都是有 advice 的，让它们内部的 advice 来执行拦截处理。

1、我们需要再回头看下 Advisor 的配置，上面我们用了 NameMatchMethodPointcutAdvisor 这个类：

```
<bean id="logCreateAdvisor" class="org.springframework.aop.support.NameMatchMet
    <property name="advice" ref="logArgsAdvice" />

    <property name="mappedNames" value="createUser,createOrder" />
</bean>
```

其实 Advisor 还有一个更加灵活的实现类 **RegexpMethodPointcutAdvisor**，它能实现正则匹配，如：

```
<bean id="logArgsAdvisor" class="org.springframework.aop.support.RegexpMethodPc
    <property name="advice" ref="logArgsAdvice" />
    <property name="pattern" value="com.javadoop.*.service.*.create.*" />
</bean>
```

也就是说，我们能通过配置 Advisor，精确定位到需要被拦截的方法，然后使用内部的 Advice 执行逻辑处理。

2、之后，我们需要配置 DefaultAdvisorAutoProxyCreator，它的配置非常简单，直接使用下面这段配置就可以了，它就会使得**所有的 Advisor 自动生效**，无须其他配置。

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProx
```

```
    <bean id="userServiceImpl" class="com.javadoop.springaoplearning.service.imple.UserServiceImpl"/>
    <bean id="orderServiceImpl" class="com.javadoop.springaoplearning.service.imple.OrderServiceImpl"/>

    <!--定义两个 advice-->
    <bean id="logArgsAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogArgsAdvice"/>
    <bean id="logResultAdvice" class="com.javadoop.springaoplearning.aop_spring_1_2.LogResultAdvice"/>

    <!--定义两个 advisor-->
    <!--记录 create* 方法的传参-->
    <bean id="logArgsAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice" ref="logArgsAdvice" />
        <property name="pattern" value="com.javadoop.*.service.*.create.*" />
    </bean>
    <!--记录 query* 的返回值-->
    <bean id="logResultAdvisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice" ref="logResultAdvice" />
        <property name="pattern" value="com.javadoop.*.service.*.query.*" />
    </bean>


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

然后我们运行一下：

```
public class Spring_1_2_DefaultAdvisorAutoProxy_Application {

    public static void main(String[] args) {

        // 启动 Spring 的 IOC 容器
        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring_1_2_DefaultAdvisorAutoProxy.xml");

        UserService userService = context.getBean(UserService.class);
        OrderService orderService = context.getBean(OrderService.class);

        userService.createUser("Tom", "Cruise", 55);
        userService.queryUser();

        orderService.createOrder("Leo", "随便买点什么");
        orderService.queryOrder("Leo");
    }
}
```

输出：

```
准备执行方法：createUser, 参数列表：[Tom, Cruise, 55]
方法返回：User{firstName='Tom', lastName='Cruise', age=55, address='null'}
准备执行方法：createOrder, 参数列表：[Leo, 随便买点什么]
方法返回：Order{username='Leo', product='随便买点什么'}
```

从结果可以看出，create *方法使用了 logArgsAdvisor 进行传参输出，query* 方法使用了 logResultAdvisor 进行了返回结果输出。

到这里，Spring 1.2 的配置就要介绍完了。本文不会介绍得面面俱到，主要是关注最核心的配置，如果读者感兴趣，要学会自己去摸索，比如这里的 Advisor 就不只有我这里介绍的 NameMatchMethodPointcutAdvisor 和 RegexpMethodPointcutAdvisor，AutoProxyCreator 也不仅仅是 BeanNameAutoProxyCreator 和 DefaultAdvisorAutoProxyCreator。

> 读到这里，我想对于很多人来说，就知道怎么去阅读 Spring AOP 源码了。

Spring 2.0 @AspectJ 配置

Spring 2.0 以后，引入了 @AspectJ 和 Schema-based 的两种配置方式，我们先来介绍 @AspectJ 的配置方式，之后我们再来看使用 xml 的配置方式。

注意了，**@AspectJ 和 AspectJ 没多大关系**，并不是说基于 AspectJ 实现的，而仅仅是使用了 AspectJ 中的概念，包括使用的注解也是直接来自于 AspectJ 的包。

首先，我们需要依赖 `aspectjweaver.jar` 这个包，这个包来自于 AspectJ：

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.11</version>
</dependency>
```

如果是使用 Spring Boot 的话，添加以下依赖即可：

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

在 @AspectJ 的配置方式中，之所以要引入 aspectjweaver 并不是因为我们需要使用 AspectJ 的处理功能，而是因为 Spring 使用了 AspectJ 提供的一些注解，实际上**还是纯的 Spring AOP 代码**。

说了这么多，明确一点，@AspectJ 采用注解的方式来配置使用 Spring AOP。

首先，我们需要**开启** @AspectJ 的注解配置方式，有两种方式：

1、在 xml 中配置：

```
<aop:aspectj-autoproxy/>
```

- 使用 @EnableAspectJAutoProxy

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

一旦开启了上面的配置，那么所有使用 @Aspect 注解的 **bean** 都会被 Spring 当做**用来实现 AOP 的配置类**，我们称之为一个 **Aspect**。

> 注意了，@Aspect 注解要作用在 bean 上面，不管是使用 @Component 等注解方式，还是在 xml 中配置 bean，首先它需要是一个 bean。

比如下面这个 bean，它的类名上使用了 @Aspect，它就会被当做 Spring AOP 的配置。

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
    <!-- configure properties of aspect here as normal -->
</bean>
```

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

接下来，我们需要关心的是 @Aspect 注解的 bean 中，我们需要配置哪些内容。

首先，我们需要配置 Pointcut，Pointcut 在大部分地方被翻译成切点，用于定义哪些方法需要被增强或者说需要被拦截，有点类似于之前介绍的 **Advisor** 的方法匹配。

Spring AOP 只支持 bean 中的方法（不像 AspectJ 那么强大），所以我们可以认为 **Pointcut** 就是用来匹配 Spring 容器中的所有 bean 的方法的。

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

我们看到，@Pointcut 中使用了 **execution** 来正则匹配方法签名，这也是最常用的，除了 execution，我们再看看其他的几个比较常用的匹配方式：

- within：指定所在类或所在包下面的方法（Spring AOP 独有）

  > 如 @Pointcut("within(com.javadoop.springaoplearning.service..*)")

- @annotation：方法上具有特定的注解，如 @Subscribe 用于订阅特定的事件。

  > 如 @Pointcut("execution(.*(..)) && @annotation(com.javadoop.annotation.Subscribe)")

- bean(idOrNameOfBean)：匹配 bean 的名字（Spring AOP 独有）

  > 如 @Pointcut("bean(*Service)")

Tips：上面匹配中，通常 "." 代表一个包名，".." 代表包及其子包，方法参数任意匹配使用两个点 ".."。

对于 web 开发者，Spring 有个很好的建议，就是定义一个 **SystemArchitecture**：

```
@Aspect
public class SystemArchitecture {

    // web 层
    @Pointcut("within(com.javadoop.web..*)")
    public void inWebLayer() {}

    // service 层
    @Pointcut("within(com.javadoop.service..*)")
    public void inServiceLayer() {}

    // dao 层
    @Pointcut("within(com.javadoop.dao..*)")
    public void inDataAccessLayer() {}

    // service 实现，注意这里指的是方法实现，其实通常也可以使用 bean(*ServiceImpl)
```

```
    @Pointcut("execution(* com.javadoop..service.*.*(..))")
    public void businessService() {}

    // dao 实现
    @Pointcut("execution(* com.javadoop.dao.*.*(..))")
    public void dataAccessOperation() {}
}
```

上面这个 SystemArchitecture 很好理解，该 Aspect 定义了一堆的 Pointcut，随后在任何需要 Pointcut 的地方都可以直接引用（如 xml 中的 pointcut-ref=""）。

配置 pointcut 就是配置我们需要拦截哪些方法，接下来，我们要配置需要对这些被拦截的方法做什么，也就是前面介绍的 Advice。

接下来，我们要配置 Advice。

下面这块代码示例了各种常用的情况：

> 注意，实际写代码的时候，不要把所有的切面都揉在一个 class 中。

```
@Aspect
public class AdviceExample {

    // 这里会用到我们前面说的 SystemArchitecture
    // 下面方法就是写拦截 "dao层实现"

    @Before("com.javadoop.aop.SystemArchitecture.dataAccessOperation()")
    public void doAccessCheck() {
        // ... 实现代码
    }

    // 当然，我们也可以直接"内联"Pointcut，直接在这里定义 Pointcut
    // 把 Advice 和 Pointcut 合在一起了，但是这两个概念我们还是要区分清楚的
    @Before("execution(* com.javadoop.dao.*.*(..))")
    public void doAccessCheck() {
        // ... 实现代码
    }

    @AfterReturning("com.javadoop.aop.SystemArchitecture.dataAccessOperation()"
    public void doAccessCheck() {
```

```
public void doAccessCheck() {
    // ...
}

@AfterReturning(
    pointcut="com.javadoop.aop.SystemArchitecture.dataAccessOperation()",
    returning="retVal")
public void doAccessCheck(Object retVal) {
    // 这样，进来这个方法的处理时候，retVal 就是相应方法的返回值，是不是非常方便
    //  ... 实现代码
}

// 异常返回
@AfterThrowing("com.javadoop.aop.SystemArchitecture.dataAccessOperation()")
public void doRecoveryActions() {
    // ... 实现代码
}

@AfterThrowing(
    pointcut="com.javadoop.aop.SystemArchitecture.dataAccessOperation()",
    throwing="ex")
public void doRecoveryActions(DataAccessException ex) {
    // ... 实现代码
}


// 注意理解它和 @AfterReturning 之间的区别，这里会拦截正常返回和异常的情况
@After("com.javadoop.aop.SystemArchitecture.dataAccessOperation()")
public void doReleaseLock() {
    // 通常就像 finally 块一样使用，用来释放资源。
    // 无论正常返回还是异常退出，都会被拦截到
}

// 感觉这个很有用吧，既能做 @Before 的事情，也可以做 @AfterReturning 的事情
@Around("com.javadoop.aop.SystemArchitecture.businessService()")
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
```

```
        return retval;
    }

}
```

细心的读者可能发现了有些 Advice 缺少方法传参，如在 @Before 场景中参数往往是非常有用的，比如我们要用日志记录下来被拦截方法的入参情况。

Spring 提供了非常简单的获取入参的方法，使用 org.aspectj.lang.JoinPoint 作为 Advice 的第一个参数即可，如：

```
@Before("com.javadoop.springaoplearning.aop_spring_2_aspectj.SystemArchitecture
public void logArgs(JoinPoint joinPoint) {
    System.out.println("方法执行前，打印入参：" + Arrays.toString(joinPoint.getArgs
}
```

> 注意：第一，必须放置在第一个参数上；第二，如果是 @Around，我们通常会使用其子类 ProceedingJoinPoint，因为它有 procceed()/procceed(args[]) 方法。

到这里，我们介绍完了 @AspectJ 配置方式中的 **Pointcut** 和 **Advice** 的配置。对于开发者来说，其实最重要的就是这两个了，定义 Pointcut 和使用合适的 Advice 在各个 Pointcut 上。

下面，我们用这一节介绍的 @AspectJ 来实现上一节实现的**记录方法传参**和**记录方法返回值**。

```
@Aspect
public class LogArgsAspect {

    // 这里可以设置一些自己想要的属性，到时候在配置的时候注入进来

    @Before("com.javadoop.springaoplearning.aop_spring_2_aspectj.SystemArchitecture.businessService()")
    public void logArgs(JoinPoint joinPoint) {
        System.out.println("方法执行前，打印入参：" + Arrays.toString(joinPoint.getArgs()));
    }
}
```

```
@Aspect
public class LogResultAspect {

    @AfterReturning(pointcut = "com.javadoop.springaoplearning.aop_spring_2_aspectj.SystemArchitecture.businessService()",
            returning = "result")
    public void logResult(Object result) {
        System.out.println(result);
    }
}
```

xml 的配置非常简单：

```
    <bean id="userService" class="com.javadoop.springaoplearning.service.imple.UserServiceImpl"/>
    <bean id="orderService" class="com.javadoop.springaoplearning.service.imple.OrderServiceImpl"/>

    <!--开启 @AspectJ 配置-->
    <aop:aspectj-autoproxy/>

    <bean class="com.javadoop.springaoplearning.aop_spring_2_aspectj.LogArgsAspect">
        <!--如果需要配置参数，和普通的 bean 一样操作-->
    </bean>
    <bean class="com.javadoop.springaoplearning.aop_spring_2_aspectj.LogResultAspect" />
```

> 这里是示例，所以 bean 的配置还是使用了 xml 的配置方式。

测试一下：

```
public class Spring_2_0_AspectJ {

    public static void main(String[] args){

        // 启动 Spring 的 IOC 容器
        ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring_2_0_aspectj.xml");

        UserService userService = context.getBean(UserService.class);
        OrderService orderService = context.getBean(OrderService.class);

        userService.createUser("Tom", "Cruise", 55);
        userService.queryUser();
    }
}
```

输出结果：

```
方法执行前，打印入参：[Tom, Cruise, 55]
User{firstName='Tom', lastName='Cruise', age=55, address='null'}
方法执行前，打印入参：[]
User{firstName='Tom', lastName='Cruise', age=55, address='null'}
```

JoinPoint 除了 getArgs() 外还有一些有用的方法，大家可以进去稍微看一眼。

最后提一点，@Aspect 中的配置不会作用于使用 @Aspect 注解的 bean。

Spring 2.0 schema-based 配置

本节将介绍的是 Spring 2.0 以后提供的基于 `<aop />` 命名空间的 XML 配置。这里说的 schema-based 就是指基于 `aop` 这个 schema。

> 介绍 IOC 的时候也介绍过 Spring 是怎么解析各个命名空间的（各种 *NamespaceHandler），解析 `<aop />` 的源码在 org.springframework.aop.config.AopNamespaceHandler 中。

有了前面的 @AspectJ 的配置方式的知识，理解 xml 方式的配置非常简单，所以我们就可以废话少一点了。

这里先介绍**配置 Aspect**，便于后续理解：

```
<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
    </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
    ...
</bean>
```

> 所有的配置都在 `<aop:config>` 下面。
>
> `<aop:aspect >` 中需要指定一个 bean，和前面介绍的 LogArgsAspect 和 LogResultAspect 一样，我们知道该 bean 中我们需要写处理代码。
>
> 然后，我们写好 Aspect 代码后，将其“织入”到合适的 Pointcut 中，这就是面向切面。

然后，我们需要**配置 Pointcut**，非常简单，如下：

```
<aop:config>


    <aop:pointcut id="businessService"
        expression="execution(* com.javadoop.springaoplearning.service.*.*(..))

    <!--也可以像下面这样-->
    <aop:pointcut id="businessService2"
        expression="com.javadoop.SystemArchitecture.businessService()"/>

</aop:config>
```

> 将 `<aop:pointcut>` 作为 `<aop:config>` 的直接子元素，将作为全局 Pointcut。

我们也可以在 `<aop:aspect />`内部配置 Pointcut，这样该 Pointcut 仅用于该 Aspect：

```
<aop:config>
    <aop:aspect ref="logArgsAspect">
```

```
        <aop:pointcut id="internalPointcut"
                      expression="com.javadoop.SystemArchitecture.businessService()"
    </aop:aspect>
</aop:config>
```

接下来，我们应该配置 Advice 了，为了避免废话过多，我们直接上实例吧，非常好理解，将上一节用 @AspectJ 方式配置的搬过来：

```
<bean id="userService" class="com.javadoop.springaoplearning.service.imple.UserServiceImpl"/>
<bean id="orderService" class="com.javadoop.springaoplearning.service.imple.OrderServiceImpl"/>

<!--定义 bean，将作为 Aspect 使用，我们需要处理的逻辑代码都在里面-->
<bean id="logArgsAspect" class="com.javadoop.springaoplearning.aop_spring_2_schema_based.LogArgsAspect" />
<bean id="logResultAspect" class="com.javadoop.springaoplearning.aop_spring_2_schema_based.LogResultAspect" />

<aop:config>
    <!--下面这两个 Pointcut 是全局的，可以被所有的 Aspect 使用-->
    <!--这里示意了两种 Pointcut 配置-->
    <aop:pointcut id="logArgsPointcut" expression="execution(* com.javadoop.springaoplearning.service.*.*(..))" />
    <aop:pointcut id="logResultPointcut" expression="
    com.javadoop.springaoplearning.aop_spring_2_schema_based.SystemArchitecture.businessService()" />

    <aop:aspect ref="logArgsAspect">
        <!--在这里也可以定义 Pointcut，不过这是局部的，不能被其他的 Aspect 使用-->
        <aop:pointcut id="internalPointcut"
                      expression="
                      com.javadoop.springaoplearning.aop_spring_2_schema_based.SystemArchitecture.businessService()" />
        <aop:before method="logArgs" pointcut-ref="internalPointcut" />
    </aop:aspect>

    <aop:aspect ref="logArgsAspect">
        <aop:before method="logArgs" pointcut-ref="logArgsPointcut" />
    </aop:aspect>

    <aop:aspect ref="logResultAspect">
        <aop:after-returning method="logResult" returning="result" pointcut-ref="logResultPointcut" />
    </aop:aspect>

</aop:config>
```

上面的例子中，我们配置了两个 LogArgsAspect 和一个 LogResultAspect。

其实基于 XML 的配置也是非常灵活的，这里没办法给大家演示各种搭配，大家抓住基本的 Pointcut、Advice 和 Aspect 这几个概念，就很容易配置了。

小结

到这里，本文介绍了 Spring AOP 的三种配置方式，我们要知道的是，到目前为止，我们使用的都是 Spring AOP，和 AspectJ 没什么关系。

下一篇文章，将会介绍 AspectJ 的使用方式，以及怎样在 Spring 应用中使用 AspectJ。之后差不多就可以出 Spring AOP 源码分析了。

《AspectJ 使用介绍》介绍了 AspectJ 的 3 种用法，感兴趣的读者可以去看一看，那篇文章稍微短一些。

附录

本文使用的测试源码已上传到 Github: hongjiev/spring-aop-learning。

建议读者 clone 下来以后，通过命令行进行测试，而不是依赖于 IDE，因为 IDE 太"智能"了：

- mvn clean package
- java -jar target/spring-aop-learning-1.0-jar-with-dependencies.jar

  > pom.xml 中配置了 assembly 插件，打包的时候会将所有 jar 包依赖打到一起。

- 修改 Application.java 中的代码，或者其他代码，然后重复 1 和 2

（全文完）

留下你的评论

回复

xue 8 months ago

网站图片都丢失了

回复

zouyu 9 months ago

看完IOC了 这两天死磕Spring

回复

yexufeijun a year ago

看完了ioc的文章，感觉好厉害

回复