Spring基础 - Spring核心之面向切面编程(AOP)

引入

> 我们在Spring基础 - Spring简单例子引入Spring的核心中向你展示了AOP的基础含义，同时以此发散了一些AOP相关知识点。

1. Spring 框架通过定义切面, 通过拦截切点实现了不同业务模块的解耦，这个就叫**面向切面编程 - Aspect Oriented Programming (AOP)**
2. 为什么@Aspect注解使用的是aspectj的jar包呢？这就引出了**Aspect4J和Spring AOP的历史渊源**，只有理解了Aspect4J和Spring的渊源才能理解有些注解上的兼容设计
3. 如何支持**更多拦截方式**来实现解耦， 以满足更多场景需求呢？ 这就是@Around, @Pointcut... 等的设计
4. 那么Spring框架又是如何实现AOP的呢？ 这就引入**代理技术，分静态代理和动态代理**，动态代理又包含JDK代理和CGLIB代理等

本节将在此基础上进一步解读AOP的含义以及AOP的使用方式；后续的文章还将深入AOP的实现原理：

- Spring进阶 - Spring AOP实现原理详解之切面实现
- Spring进阶 - Spring AOP实现原理详解之AOP代理

如何理解AOP

> AOP的本质也是为了解耦，它是一种设计思想； 在理解时也应该简化理解。

AOP（Aspect-Oriented Programming，面向方面编程），可以说是OOP（Object-Oriented Programing，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需 要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日 志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种 散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为 “Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低 模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为； 那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手 将这些剖开的切面复原，不留痕迹。

AOP是什么

> AOP为Aspect Oriented Programming的缩写，意为：面向切面编程

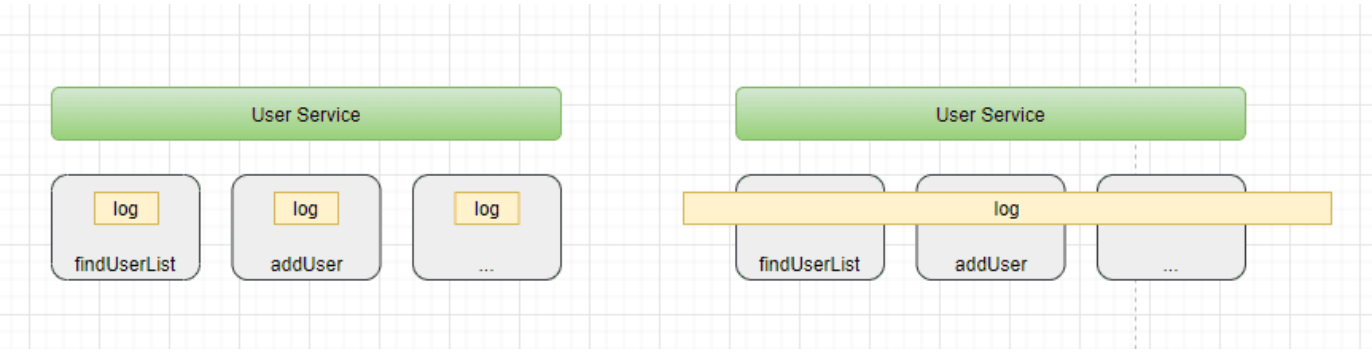
AOP最早是AOP联盟的组织提出的,指定的一套规范,spring将AOP的思想引入框架之中,通过**预编译方式**和**运行期间动态代理**实现程序的统一维护的一种技术,

- 先来看一个例子， 如何给如下UserServiceImpl中所有方法添加进入方法的日志，

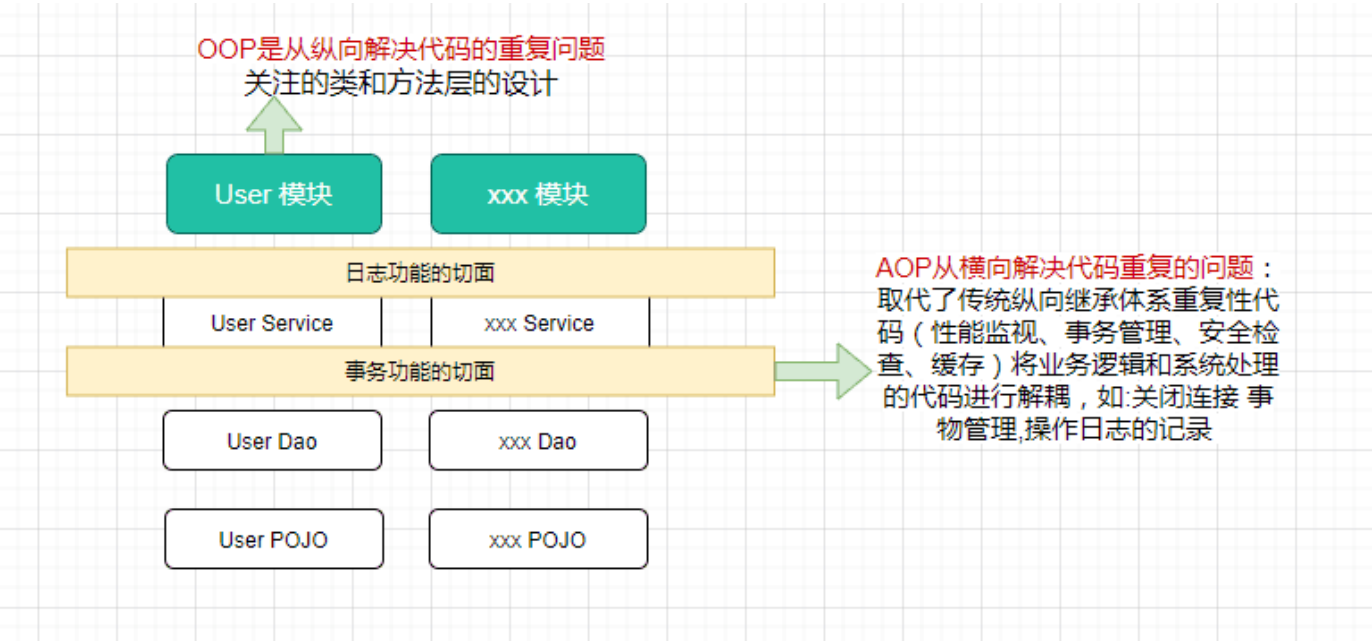
```java
/**
 * @author pdai
 */
public class UserServiceImpl implements IUserService {

    /**
     * find user list.
     *
     * @return user list
     */

```

```
    @Override
    public List<User> findUserList() {
        System.out.println("execute method:  findUserList");
        return Collections.singletonList(new User("pdai", 18));
    }

    /**
     * add user
     */
    @Override
    public void addUser() {
        System.out.println("execute method:  addUser");
        // do something
    }

}
```

我们将记录日志功能解耦为日志切面，它的目标是解耦。进而引出AOP的理念：就是将分散在各个业务逻辑代码中相同的代码通过**横向切割**的方式抽取到一个独立的模块中！



OOP面向对象编程，针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。而AOP则是针对业务处理过程中的切面进行提取，它所面对的是处理过程的某个步骤或阶段，以获得逻辑过程的中各部分之间低耦合的隔离效果。这两种设计思想在目标上有着本质的差异。



AOP术语

> 首先让我们从一些重要的AOP概念和术语开始。**这些术语不是Spring特有的。**

- **连接点（Jointpoint）**：表示需要在程序中插入横切关注点的扩展点，**连接点可能是类初始化、方法执行、方法调用、字段调用或处理异常等等**，Spring只支持方法执行连接点，在AOP中表示为**在哪里干**；
- **切入点（Pointcut）**： 选择一组相关连接点的模式，即可以认为连接点的集合，Spring支持perl5正则表达式和AspectJ切入点模式，Spring默认使用AspectJ语法，在AOP中表示为**在哪里干的集合**；
- **通知（Advice）**：在连接点上执行的行为，通知提供了在AOP中需要在切入点所选择的连接点处进行扩展现有行为的手段；包括前置通知（before advice）、后置通知(after advice)、环绕通知（around advice），在Spring中通过代理模式实现AOP，并通过拦截器模式以环绕连接点的拦截器链织入通知；在AOP中表示为**干什么**；
- **方面/切面（Aspect）**：横切关注点的模块化，比如上边提到的日志组件。可以认为是通知、引入和切入点的组合；在Spring中可以使用Schema和@AspectJ方式进行组织实现；在AOP中表示为**在哪干和干什么集合**；
- **引入（inter-type declaration）**：也称为内部类型声明，为已有的类添加额外新的字段或方法，Spring允许引入新的接口（必须对应一个实现）到所有被代理对象（目标对象），在AOP中表示为**干什么（引入什么）**；
- **目标对象（Target Object）**：需要被织入横切关注点的对象，即该对象是切入点选择的对象，需要被通知的对象，从而也可称为被通知对象；由于Spring AOP 通过代理模式实现，从而这个对象永远是被代理对象，在AOP中表示为**对谁干**；
- **织入（Weaving）**：把切面连接到其它的应用程序类型或者对象上，并创建一个被通知的对象。这些可以在编译时（例如使用AspectJ编译器），类加载时和运行时完成。Spring和其他纯Java AOP框架一样，在运行时完成织入。在AOP中表示为**怎么实现的**；
- **AOP代理（AOP Proxy）**：AOP框架使用代理模式创建的对象，从而实现在连接点处插入通知（即应用切面），就是通过代理来对目标对象应用切面。在Spring中，AOP代理可以用JDK动态代理或CGLIB代理实现，而通过拦截器模型应用切面。在AOP中表示为**怎么实现的一种典型方式**；

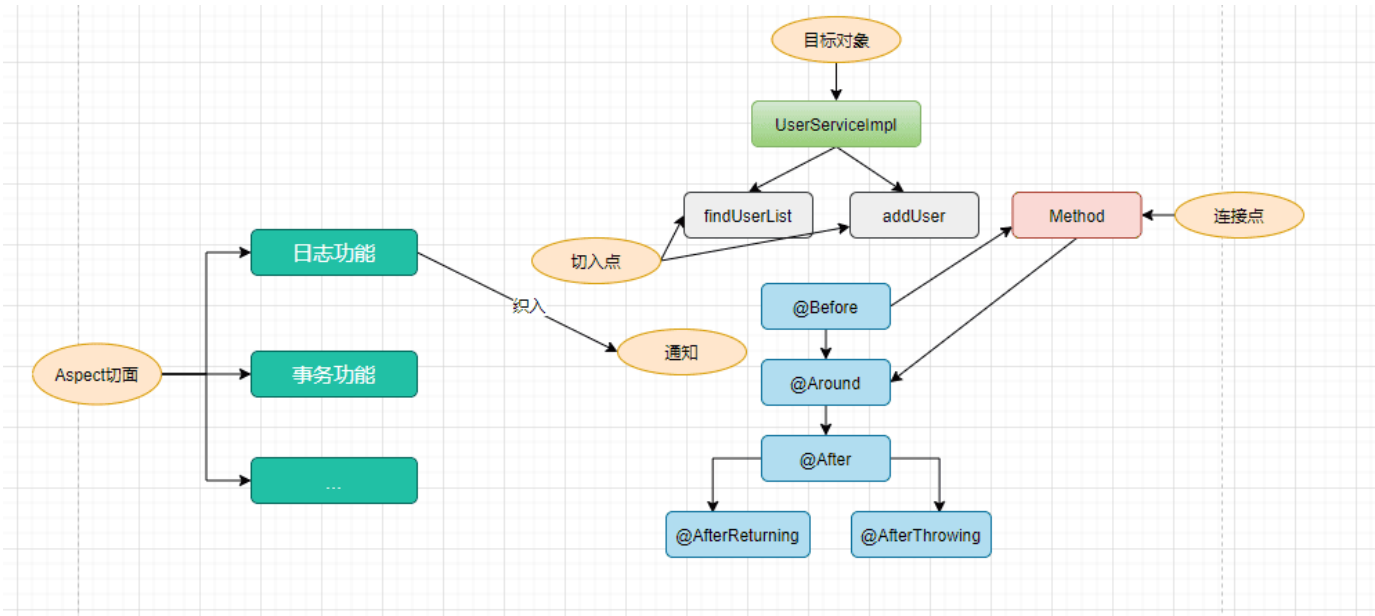
> **通知类型**：

- **前置通知（Before advice）**：在某连接点之前执行的通知，但这个通知不能阻止连接点之前的执行流程（除非它抛出一个异常）。

- **后置通知（After returning advice）**：在某连接点正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。
- **异常通知（After throwing advice）**：在方法抛出异常退出时执行的通知。
- **最终通知（After (finally) advice）**：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。
- **环绕通知（Around Advice）**：包围一个连接点的通知，如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它自己的返回值或抛出异常来结束执行。

环绕通知是最常用的通知类型。和AspectJ一样，Spring提供所有类型的通知，我们推荐你使用尽可能简单的通知类型来实现需要的功能。例如，如果你只是需要一个方法的返回值来更新缓存，最好使用后置通知而不是环绕通知，尽管环绕通知也能完成同样的事情。用最合适的通知类型可以使得编程模型变得简单，并且能够避免很多潜在的错误。比如，你不需要在JoinPoint上调用用于环绕通知的proceed()方法，就不会有调用的问题。

> 我们把这些术语串联到一起，方便理解



Spring AOP和AspectJ是什么关系

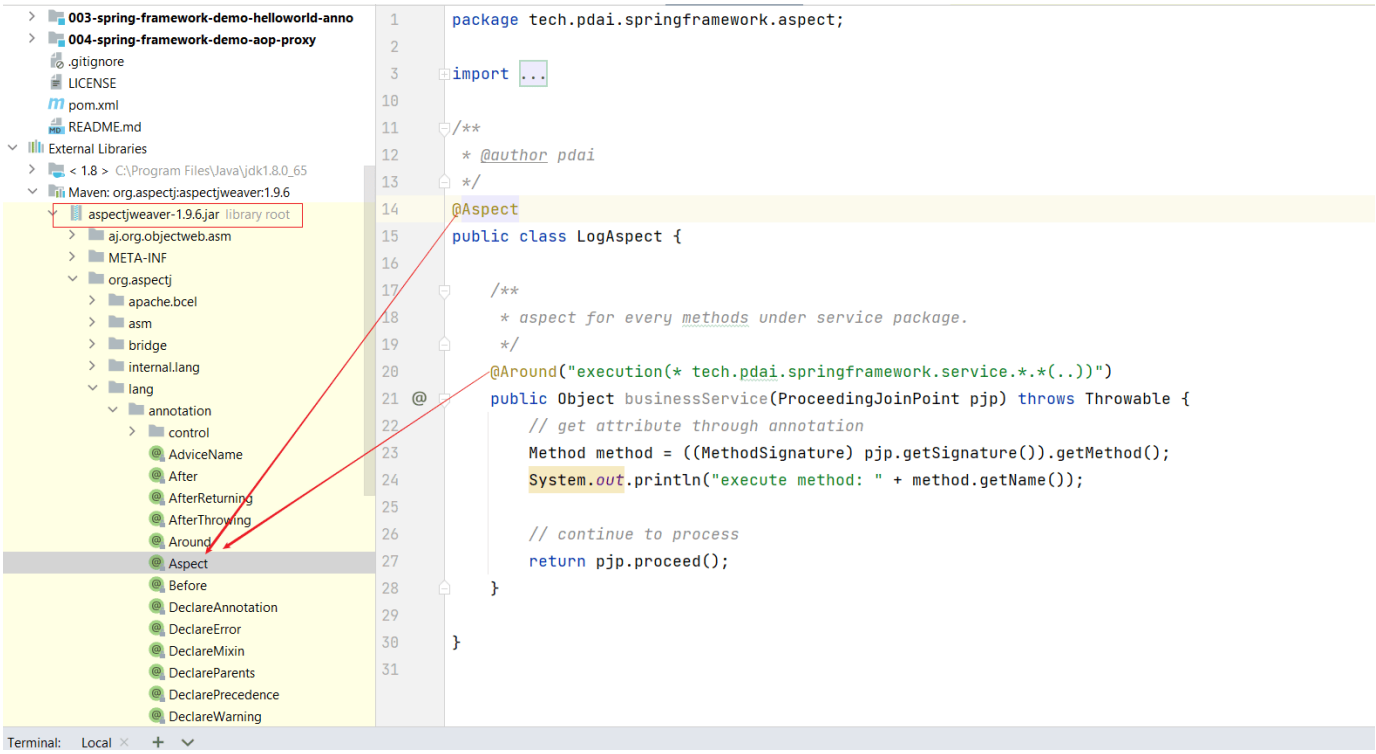
- **首先AspectJ是什么**?

AspectJ是一个java实现的AOP框架，它能够对java代码进行AOP编译（一般在编译期进行），让java代码具有AspectJ的AOP功能（当然需要特殊的编译器）

可以这样说AspectJ是目前实现AOP框架中最成熟，功能最丰富的语言，更幸运的是，AspectJ与java程序完全兼容，几乎是无缝关联，因此对于有java编程基础的工程师，上手和使用都非常容易。

- **其次，为什么需要理清楚Spring AOP和AspectJ的关系**?

我们看下@Aspect以及增强的几个注解，为什么不是Spring包，而是来源于aspectJ呢?



- **Spring AOP和AspectJ是什么关系**?

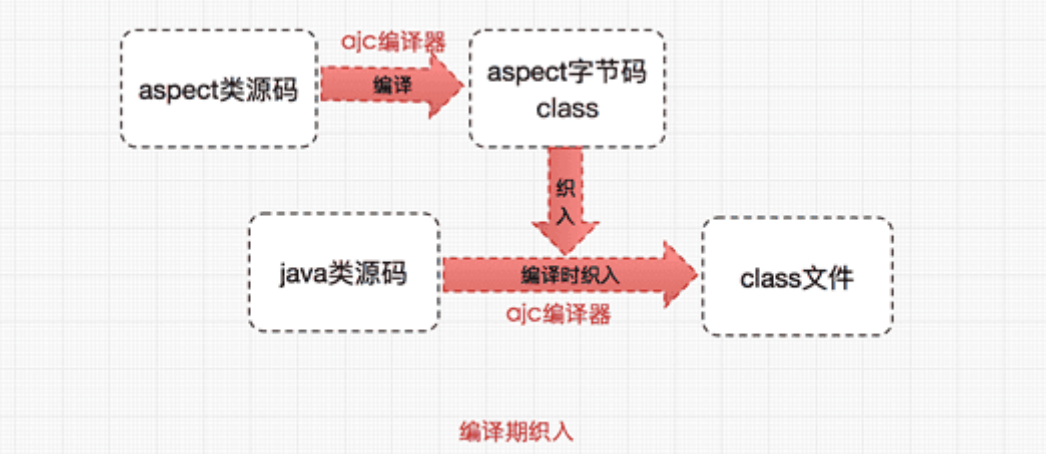
1. AspectJ是更强的AOP框架，是实际意义的**AOP标准**;
2. Spring为何不写类似AspectJ的框架？ Spring AOP使用纯Java实现, 它不需要专门的编译过程, 它一个**重要的原则就是无侵入性（non-invasiveness）**; Spring 小组完全有能力写类似的框架，只是Spring AOP从来没有打算通过提供一种全面的AOP解决方案来与AspectJ竞争。Spring的开发小组相信无论是基于代理（proxy-based）的框架如Spring AOP或者是成熟的框架如AspectJ都是很有价值的，他们之间应该是**互补而不是竞争的关系。**
3. Spring小组喜欢@AspectJ注解风格更胜于Spring XML配置； 所以**在Spring 2.0使用了和AspectJ 5一样的注解，并使用AspectJ来做切入点解析和匹配。但是，AOP在运行时仍旧是纯的Spring AOP，并不依赖于AspectJ的编译器或者织入器（weaver）。**
4. Spring 2.5对AspectJ的支持：在一些环境下，增加了对AspectJ的装载时编织支持，同时提供了一个新的bean切入点。

- **更多关于AspectJ**?

了解AspectJ应用到java代码的过程（这个过程称为织入），对于织入这个概念，可以简单理解为aspect(切面)应用到目标函数(类)的过程。

对于这个过程，一般分为**动态织入**和**静态织入**:

1. 动态织入的方式是在运行时动态将要增强的代码织入到目标类中，这样往往是通过动态代理技术完成的，如Java JDK的动态代理(Proxy，底层通过反射实现)或者CGLIB的动态代理(底层通过继承实现)，Spring AOP采用的就是基于运行时增强的代理技术
2. ApectJ采用的就是静态织入的方式。ApectJ主要采用的是编译期织入，在这个期间使用AspectJ的acj编译器(类似javac)把aspect类编译成class字节码后，在java目标类编译时织入，即先编译aspect类再编译目标类。



AOP的配置方式

> Spring AOP 支持对XML模式和基于@AspectJ注解的两种配置方式。

XML Schema配置方式

Spring提供了使用"aop"命名空间来定义一个切面，我们来看个例子(例子代码)：

- **定义目标类**

```java
package tech.pdai.springframework.service;

/**
 * @author pdai
 */
public class AopDemoServiceImpl {

    public void doMethod1() {
        System.out.println("AopDemoServiceImpl.doMethod1()");
    }

    public String doMethod2() {
        System.out.println("AopDemoServiceImpl.doMethod2()");
        return "hello world";
    }

    public String doMethod3() throws Exception {
        System.out.println("AopDemoServiceImpl.doMethod3()");
        throw new Exception("some exception");
    }
}
```

- **定义切面类**

```java
package tech.pdai.springframework.aspect;

import org.aspectj.lang.ProceedingJoinPoint;

/**
 * @author pdai
 */
public class LogAspect {

    /**
     * 环绕通知.
     *
     * @param pjp pjp
     * @return obj
     * @throws Throwable exception
     */
    public Object doAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("-----------------------");
        System.out.println("环绕通知: 进入方法");
        Object o = pjp.proceed();
        System.out.println("环绕通知: 退出方法");
        return o;
    }

    /**
     * 前置通知.
     */
    public void doBefore() {
        System.out.println("前置通知");
    }

    /**
     * 后置通知.
     *
     * @param result return val
     */
    public void doAfterReturning(String result) {
        System.out.println("后置通知, 返回值: " + result);
    }

    /**
     * 异常通知.
     *
     * @param e exception
     */
    public void doAfterThrowing(Exception e) {
        System.out.println("异常通知, 异常: " + e.getMessage());
    }

    /**

```

```
     * 最终通知.
     */
    public void doAfter() {
        System.out.println("最终通知");
    }

}
```

- **XML配置AOP**

```xml
<?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="tech.pdai.springframework" />

    <aop:aspectj-autoproxy/>

    <!-- 目标类 -->
    <bean id="demoService" class="tech.pdai.springframework.service.AopDemoServiceImpl">
        <!-- configure properties of bean here as normal -->
    </bean>

    <!-- 切面 -->
    <bean id="logAspect" class="tech.pdai.springframework.aspect.LogAspect">
        <!-- configure properties of aspect here as normal -->
    </bean>

    <aop:config>
        <!-- 配置切面 -->
        <aop:aspect ref="logAspect">
            <!-- 配置切入点 -->
            <aop:pointcut id="pointCutMethod" expression="execution(* tech.pdai.springframe
            <!-- 环绕通知 -->
            <aop:around method="doAround" pointcut-ref="pointCutMethod"/>
            <!-- 前置通知 -->
            <aop:before method="doBefore" pointcut-ref="pointCutMethod"/>
            <!-- 后置通知：returning属性：用于设置后置通知的第二个参数的名称，类型是Object -->
            <aop:after-returning method="doAfterReturning" pointcut-ref="pointCutMethod" re
            <!-- 异常通知：如果没有异常，将不会执行增强；throwing属性：用于设置通知第二个参数的的名
```

```
            <aop:after-throwing method="doAfterThrowing" pointcut-ref="pointCutMethod" thro
            <!-- 最终通知 -->
            <aop:after method="doAfter" pointcut-ref="pointCutMethod"/>
        </aop:aspect>
    </aop:config>

    <!-- more bean definitions for data access objects go here -->
</beans>
```

- **测试类**

```java
/**
  * main interfaces.
  *
  * @param args args
  */
public static void main(String[] args) {
    // create and configure beans
    ApplicationContext context = new ClassPathXmlApplicationContext("aspects.xml");

    // retrieve configured instance
    AopDemoServiceImpl service = context.getBean("demoService", AopDemoServiceImpl.class);

    // use configured instance
    service.doMethod1();
    service.doMethod2();
    try {
        service.doMethod3();
    } catch (Exception e) {
        // e.printStackTrace();
    }
}
```

- **输出结果**

```java
----------------------
环绕通知: 进入方法
前置通知
AopDemoServiceImpl.doMethod1()
环绕通知: 退出方法
最终通知
----------------------
环绕通知: 进入方法
前置通知
AopDemoServiceImpl.doMethod2()
环绕通知: 退出方法

```

```
最终通知
后置通知，返回值: hello world
-----------------------
环绕通知: 进入方法
前置通知
AopDemoServiceImpl.doMethod3()
最终通知
异常通知，异常: some exception
```

AspectJ注解方式

基于XML的声明式AspectJ存在一些不足，需要在Spring配置文件配置大量的代码信息，为了解决这个问题，Spring 使用了@AspectJ框架为AOP的实现提供了一套注解。

注解名称	解释
@Aspect	用来定义一个切面。
@pointcut	用于定义切入点表达式。在使用时还需要定义一个包含名字和任意参数的方法签名来表示切入点名称，这个方法签名就是一个返回值为void，且方法体为空的普通方法。
@Before	用于定义前置通知，相当于BeforeAdvice。在使用时，通常需要指定一个value属性值，该属性值用于指定一个切入点表达式(可以是已有的切入点，也可以直接定义切入点表达式)。
@AfterReturning	用于定义后置通知，相当于AfterReturningAdvice。在使用时可以指定pointcut / value和returning属性，其中pointcut / value这两个属性的作用一样，都用于指定切入点表达式。
@Around	用于定义环绕通知，相当于MethodInterceptor。在使用时需要指定一个value属性，该属性用于指定该通知被植入的切入点。
@After-Throwing	用于定义异常通知来处理程序中未处理的异常，相当于ThrowAdvice。在使用时可指定pointcut / value和throwing属性。其中pointcut/value用于指定切入点表达式，而throwing属性值用于指定-一个形参名来表示Advice方法中可定义与此同名的形参，该形参可用于访问目标方法抛出的异常。
@After	用于定义最终final 通知，不管是否异常，该通知都会执行。使用时需要指定一个value属性，该属性用于指定该通知被植入的切入点。
@DeclareParents	用于定义引介通知，相当于IntroductionInterceptor (不要求掌握)。

> Spring AOP的实现方式是动态织入，动态织入的方式是在运行时动态将要增强的代码织入到目标类中，这样往往是通过动态代理技术完成的；**如Java JDK的动态代理(Proxy，底层通过反射实现)或者CGLIB的动态代理(底层通过继承实现)**，Spring AOP采用的就是基于运行时增强的代理技术。所以我们看下如下的两个例子(例子代码 中05模块）：

> - 基于JDK代理例子
> - 基于Cglib代理例子

接口使用JDK代理

- 定义接口

```java
/**
 * Jdk Proxy Service.
 *
 * @author pdai
 */
public interface IJdkProxyService {

    void doMethod1();

    String doMethod2();

    String doMethod3() throws Exception;
}
```

- 实现类

```java
/**
 * @author pdai
 */
@Service
public class JdkProxyDemoServiceImpl implements IJdkProxyService {

    @Override
    public void doMethod1() {
        System.out.println("JdkProxyServiceImpl.doMethod1()");
    }

    @Override
    public String doMethod2() {
        System.out.println("JdkProxyServiceImpl.doMethod2()");
        return "hello world";
    }

    @Override
    public String doMethod3() throws Exception {
        System.out.println("JdkProxyServiceImpl.doMethod3()");
        throw new Exception("some exception");


```

```
    }
}
```

- **定义切面**

```java
package tech.pdai.springframework.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.stereotype.Component;

/**
 * @author pdai
 */
@EnableAspectJAutoProxy
@Component
@Aspect
public class LogAspect {

    /**
     * define point cut.
     */
    @Pointcut("execution(* tech.pdai.springframework.service.*.*(..))")
    private void pointCutMethod() {
    }


    /**
     * 环绕通知.
     *
     * @param pjp pjp
     * @return obj
     * @throws Throwable exception
     */
    @Around("pointCutMethod()")
    public Object doAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("----------------------");
        System.out.println("环绕通知: 进入方法");
        Object o = pjp.proceed();
        System.out.println("环绕通知: 退出方法");
        return o;
    }
```

```java
    /**
     * 前置通知.
     */
    @Before("pointCutMethod()")
    public void doBefore() {
        System.out.println("前置通知");
    }


    /**
     * 后置通知.
     *
     * @param result return val
     */
    @AfterReturning(pointcut = "pointCutMethod()", returning = "result")
    public void doAfterReturning(String result) {
        System.out.println("后置通知, 返回值: " + result);
    }

    /**
     * 异常通知.
     *
     * @param e exception
     */
    @AfterThrowing(pointcut = "pointCutMethod()", throwing = "e")
    public void doAfterThrowing(Exception e) {
        System.out.println("异常通知, 异常: " + e.getMessage());
    }

    /**
     * 最终通知.
     */
    @After("pointCutMethod()")
    public void doAfter() {
        System.out.println("最终通知");
    }

}
```

- **输出**

```java
----------------------
环绕通知: 进入方法
前置通知
JdkProxyServiceImpl.doMethod1()
最终通知
环绕通知: 退出方法
----------------------
```

```
环绕通知: 进入方法
前置通知
JdkProxyServiceImpl.doMethod2()
后置通知, 返回值: hello world
最终通知
环绕通知: 退出方法
----------------------
环绕通知: 进入方法
前置通知
JdkProxyServiceImpl.doMethod3()
异常通知, 异常: some exception
最终通知
```

非接口使用Cglib代理

- **类定义**

```java
/**
 * Cglib proxy.
 *
 * @author pdai
 */
@Service
public class CglibProxyDemoServiceImpl {

    public void doMethod1() {
        System.out.println("CglibProxyDemoServiceImpl.doMethod1()");
    }

    public String doMethod2() {
        System.out.println("CglibProxyDemoServiceImpl.doMethod2()");
        return "hello world";
    }

    public String doMethod3() throws Exception {
        System.out.println("CglibProxyDemoServiceImpl.doMethod3()");
        throw new Exception("some exception");
    }
}
```

- **切面定义**

和上面相同

- **输出**

```java
----------------------
环绕通知: 进入方法
```

```
前置通知
CglibProxyDemoServiceImpl.doMethod1()
最终通知
环绕通知: 退出方法
-----------------------
环绕通知: 进入方法
前置通知
CglibProxyDemoServiceImpl.doMethod2()
后置通知, 返回值: hello world
最终通知
环绕通知: 退出方法
-----------------------
环绕通知: 进入方法
前置通知
CglibProxyDemoServiceImpl.doMethod3()
异常通知, 异常: some exception
最终通知
```

AOP使用问题小结

> 这里总结下实际开发中会遇到的一些问题：

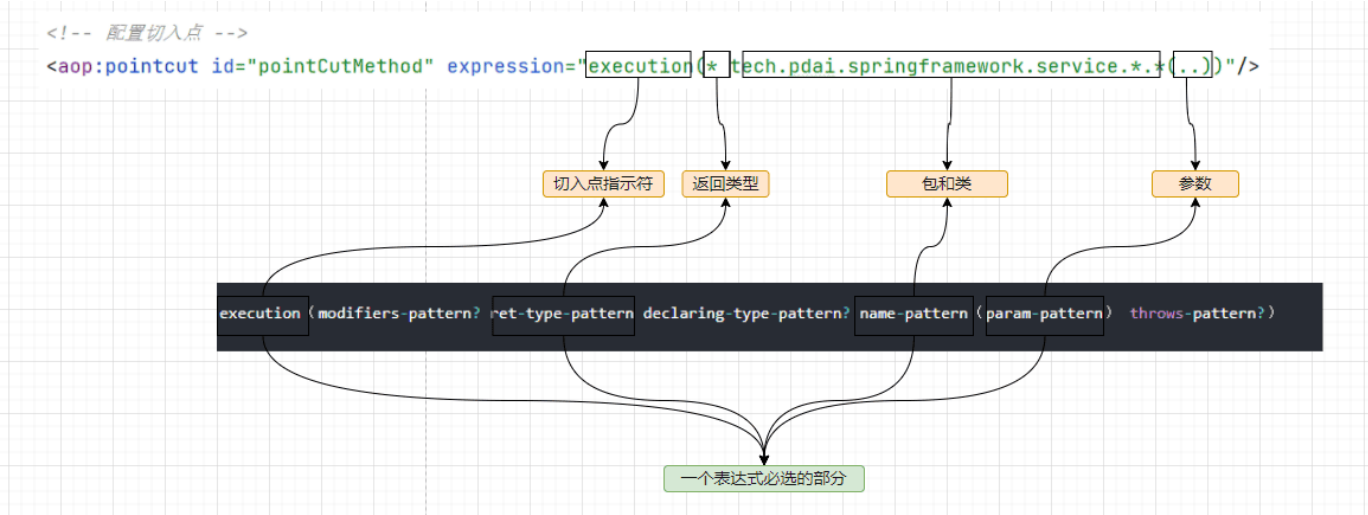
切入点（pointcut）的申明规则?

Spring AOP 用户可能会经常使用 execution切入点指示符。执行表达式的格式如下：

```java
execution（modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern（param-p
```

- ret-type-pattern 返回类型模式, name-pattern名字模式和param-pattern参数模式是必选的， 其它部分都是可选的。返回类型模式决定了方法的返回类型必须依次匹配一个连接点。 你会使用的最频繁的返回类型模式是`*`，**它代表了匹配任意的返回类型。**
- declaring-type-pattern, 一个全限定的类型名将只会匹配返回给定类型的方法。
- name-pattern 名字模式匹配的是方法名。 你可以使用`*`通配符作为所有或者部分命名模式。
- param-pattern 参数模式稍微有点复杂：()匹配了一个不接受任何参数的方法， 而(..)匹配了一个接受任意数量参数的方法（零或者更多）。 模式*()匹配了一个接受一个任何类型的参数的方法。* *模式*(*,String)匹配了一个接受两个参数的方法，第一个可以是任意类型， 第二个则必须是String类型。

对应到我们上面的例子：



下面给出一些通用切入点表达式的例子。

```java
// 任意公共方法的执行：
execution(public * *(..))

// 任何一个名字以“set”开始的方法的执行：
execution(* set*(..))

// AccountService接口定义的任意方法的执行：
execution(* com.xyz.service.AccountService.*(..))

// 在service包中定义的任意方法的执行：
execution(* com.xyz.service.*.*(..))

// 在service包或其子包中定义的任意方法的执行：
execution(* com.xyz.service..*.*(..))

// 在service包中的任意连接点（在Spring AOP中只是方法执行）：
within(com.xyz.service.*)

// 在service包或其子包中的任意连接点（在Spring AOP中只是方法执行）：
within(com.xyz.service..*)

// 实现了AccountService接口的代理对象的任意连接点 （在Spring AOP中只是方法执行）：
this(com.xyz.service.AccountService) // 'this'在绑定表单中更加常用

// 实现AccountService接口的目标对象的任意连接点 （在Spring AOP中只是方法执行）：
target(com.xyz.service.AccountService) // 'target'在绑定表单中更加常用

// 任何一个只接受一个参数，并且运行时所传入的参数是Serializable 接口的连接点（在Spring AOP中只是方法
args(java.io.Serializable) // 'args'在绑定表单中更加常用; 请注意在例子中给出的切入点不同于 exec

// 目标对象中有一个 @Transactional 注解的任意连接点 （在Spring AOP中只是方法执行）
@target(org.springframework.transaction.annotation.Transactional) // '@target'在绑定表单中更

// 任何一个目标对象声明的类型有一个 @Transactional 注解的连接点 （在Spring AOP中只是方法执行）：
@within(org.springframework.transaction.annotation.Transactional) // '@within'在绑定表单中更
```

```
// 任何一个执行的方法有一个 @Transactional 注解的连接点 （在Spring AOP中只是方法执行）
@annotation（org.springframework.transaction.annotation.Transactional） // '@annotation'在绑

// 任何一个只接受一个参数，并且运行时所传入的参数类型具有@Classified 注解的连接点（在Spring AOP中只
@args（com.xyz.security.Classified） // '@args'在绑定表单中更加常用

// 任何一个在名为'tradeService'的Spring bean之上的连接点 （在Spring AOP中只是方法执行）
bean（tradeService）

// 任何一个在名字匹配通配符表达式'*Service'的Spring bean之上的连接点 （在Spring AOP中只是方法执行）
bean（*Service）
```

此外Spring 支持如下三个逻辑运算符来组合切入点表达式

```java
&&：要求连接点同时匹配两个切入点表达式
||：要求连接点匹配任意个切入点表达式
!:：要求连接点不匹配指定的切入点表达式
```

多种增强通知的顺序?

如果有多个通知想要在同一连接点运行会发生什么？Spring AOP遵循跟AspectJ一样的优先规则来确定通知执行的顺序。 在"进入"连接点的情况下，最高优先级的通知会先执行（所以给定的两个前置通知中，优先级高的那个会先执行）。 在"退出"连接点的情况下，最高优先级的通知会最后执行。（所以给定的两个后置通知中， 优先级高的那个会第二个执行）。

当定义在不同的切面里的两个通知都需要在一个相同的连接点中运行， 那么除非你指定，否则执行的顺序是未知的。你可以通过指定优先级来控制执行顺序。 在标准的Spring方法中可以在切面类中实现org.springframework.core.Ordered 接口或者用**Order注解**做到这一点。在两个切面中， Ordered.getValue()方法返回值（或者注解值）较低的那个有更高的优先级。

当定义在相同的切面里的两个通知都需要在一个相同的连接点中运行， 执行的顺序是未知的（因为这里没有方法通过反射javac编译的类来获取声明顺序）。 考虑在每个切面类中按连接点压缩这些通知方法到一个通知方法，或者重构通知的片段到各自的切面类中 - 它能在切面级别进行排序。

Spring AOP 和 AspectJ 之间的关键区别?

AspectJ可以做Spring AOP干不了的事情，**它是AOP编程的完全解决方案，Spring AOP则致力于解决企业级开发中最普遍的AOP**（方法织入）。

下表总结了 Spring AOP 和 AspectJ 之间的关键区别:

Spring AOP	AspectJ
在纯 Java 中实现	使用 Java 编程语言的扩展实现
不需要单独的编译过程	除非设置 LTW，否则需要 AspectJ 编译器 (ajc)
只能使用运行时织入	运行时织入不可用。支持编译时、编译后和加载时织入
功能不强-仅支持方法级编织	更强大 - 可以编织字段、方法、构造函数、静态初始值设定项、最终类/方法等……。
只能在由 Spring 容器管理的 bean 上实现	可以在所有域对象上实现
仅支持方法执行切入点	支持所有切入点
代理是由目标对象创建的, 并且切面应用在这些代理上	在执行应用程序之前 (在运行时) 前, 各方面直接在代码中进行织入
比 AspectJ 慢多了	更好的性能
易于学习和应用	相对于 Spring AOP 来说更复杂

Spring AOP还是完全用AspectJ?

以下Spring官方的回答：（总结来说就是 **Spring AOP更易用，AspectJ更强大**）。

- Spring AOP比完全使用AspectJ更加简单， 因为它不需要引入AspectJ的编译器 / 织入器到你开发和构建过程中。 如果你**仅仅需要在Spring bean上通知执行操作，那么Spring AOP是合适的选择。**
- 如果你需要通知domain对象或其它没有在Spring容器中管理的任意对象，那么你需要使用AspectJ。
- 如果你想通知除了简单的方法执行之外的连接点（如：调用连接点、字段get或set的连接点等等）， 也需要使用AspectJ。

当使用AspectJ时，你可以选择使用AspectJ语言（也称为"代码风格"）或@AspectJ注解风格。 如果切面在你的设计中扮演一个很大的角色，并且你能在Eclipse等IDE中使用AspectJ Development Tools (AJDT)， 那么首选AspectJ语言 :- 因为该语言专门被设计用来编写切面，所以会更清晰、更简单。如果你没有使用 Eclipse等IDE，或者在你的应用中只有很少的切面并没有作为一个主要的角色，你或许应该考虑使用@AspectJ风格 并在你的IDE中附加一个普通的Java编辑器，并且在你的构建脚本中增加切面织入（链接）的段落。

参考文章

http://shouce.jb51.net/spring/aop.html#aop-ataspectj

https://www.cnblogs.com/linhp/p/5881788.html

https://www.cnblogs.com/bj-xiaodao/p/10777914.html