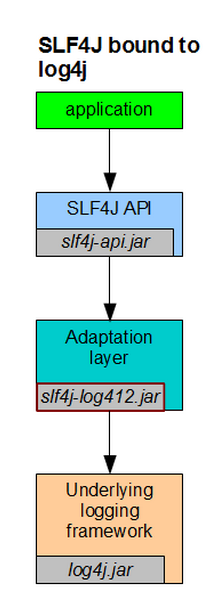
slf4j:Simple Logging Facade for Java，为java提供的简单日志Facade。Facade：门面，更底层一点说就是接口。他允许用户以自己的喜好，在工程中通过slf4j接入不同的日志系统。更直观一点，slf4j是个数据线，一端嵌入程序，另一端链接日志系统，从而实现将程序中的信息导入到日志系统并记录。

   因此，slf4j入口就是众多接口的集合，他不负责具体的日志实现，只在编译时负责寻找合适的日志系统进行绑定。具体有哪些接口，全部都定义在slf4j-api中。查看slf4j-api源码就可以发现，里面除了public final class LoggerFactory类之外，都是接口定义。因此，slf4j-api本质就是一个接口定义。

下图比较清晰的描述了他们之间的关系：



  当系统采用log4j作为日志框架实现的调用关系：

    首先系统包含slf4j-api作为日志接入的接口；

    at compile时slf4j-api中public final class LoggerFactor类中

    private final static void bind() 方法会寻找具体的日志实现类绑定，主要通过

    StaticLoggerBinder.getSingleton();语句调用

   slf4j-log4j12:链接slf4j-api和log4j中间的适配器。它实现了slf4j-apiz中StaticLoggerBinder接口，从而使得在编译时绑定的是slf4j-log4j12的getSingleton()方法

log4j:这个是具体的日志系统。通过slf4j-log4j12初始化Log4j，达到最终日志的输出。

[] SSM框架 之 Spring-AOP（面向切面编程）](http://blog.csdn.net/onlylove_longshao/article/details/53560785)

标签： [spring](http://www.csdn.net/tag/spring)[aop](http://www.csdn.net/tag/aop)[基于注解](http://www.csdn.net/tag/%e5%9f%ba%e4%ba%8e%e6%b3%a8%e8%a7%a3)[基于xml](http://www.csdn.net/tag/%e5%9f%ba%e4%ba%8exml)

2016-12-10 20:48 1600人阅读 评论(0) [收藏](javascript:void(0);) 举报

 分类：

【SSM】（4） 

目录(?)

[+]

**1AOP概述**

AOP(Aspect-Oriented Programming，面向切面编程)：是一种新的方法论，是对传统 OOP(Object-Oriented Programming，面向对象编程)的补充。

两种编程思想。是对面向对象编程的一种补充。

面向切面编程：是指在程序运行期间将某段代码，动态的切入到某个类的指定方法的指定位置的这种编程思想叫做面向切面编程。

AOP编程操作的主要对象是切面(aspect)，而切面模块化横切关注点。

在应用AOP编程时，仍然需要定义公共功能，但可以明确的定义这个功能应用在哪里，以什么方式应用，并且不必修改受影响的类。这样一来横切关注点就被模块化到特殊的类里——这样的类我们通常称之为“切面”。

AOP的好处：

每个事物逻辑位于一个位置，代码不分散，便于维护和升级

业务模块更简洁，只包含核心业务代码

**2AOP术语**

2.1横切关注点:从每个方法中抽取出来的同一类非核心业务。

2.2切面(Aspect):封装横切关注点信息的类，每个关注点体现为一个通知方法。

2.3通知(Advice):切面必须要完成的各个具体工作

2.4目标(Target):被通知的对象

2.5代理(Proxy):向目标对象应用通知之后创建的代理对象

2.6连接点(Joinpoint)

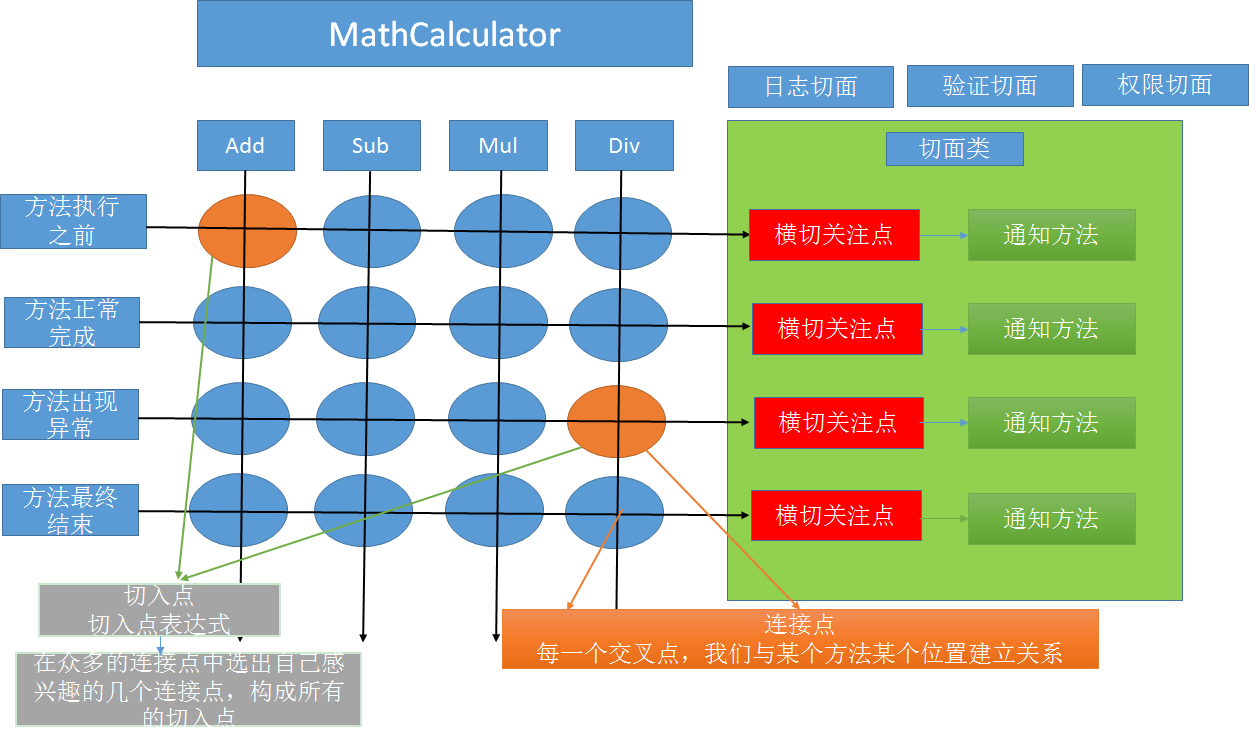
横切关注点在程序代码中的具体体现，对应程序执行的某个特定位置。例如：类某个方法调用前、调用后、方法捕获到异常后等。

在应用程序中可以使用横纵两个坐标来定位一个具体的连接点：

2.7切入点(pointcut)：

定位连接点的方式。每个类的方法中都包含多个连接点，所以连接点是类中客观存在的事物。如果把连接点看作数据库中的记录，那么切入点就是查询条件——AOP可以通过切入点定位到特定的连接点。切点通过org.springframework.aop.Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件。

为了方便理解以上专业术语，给出下图：



**3AspectJ**

**3.1简介**

AspectJ：Java社区里最完整最流行的AOP框架。

在Spring2.0以上版本中，可以使用基于AspectJ注解或基于XML配置的AOP。

**3.2在Spring中启用AspectJ注解支持**

①导入JAR包

commons-logging-1.1.3.jar

spring-beans-4.0.0.RELEASE.jar

spring-context-4.0.0.RELEASE.jar

spring-core-4.0.0.RELEASE.jar

spring-expression-4.0.0.RELEASE.jar

AOP依赖的包

spring-aop-4.0.0.RELEASE.jar

spring-aspects-4.0.0.RELEASE.jar

//增强版

com.springsource.net.sf.cglib-2.2.0.jar

com.springsource.org.aopalliance-1.0.0.jar

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

②引入aop名称空间

③配置

<aop:aspectj-autoproxy>

当Spring IOC容器侦测到bean配置文件中的<aop:aspectj-autoproxy>元素时，会自动为与AspectJ切面匹配的bean创建代理

**3.3用AspectJ注解声明切面**

①要在Spring中声明AspectJ切面，只需要在IOC容器中将切面声明为bean实例。②当在Spring IOC容器中初始化AspectJ切面之后，Spring IOC容器就会为那些与 AspectJ切面相匹配的bean创建代理。

③在AspectJ注解中，切面只是一个带有@Aspect注解的Java类，它往往要包含很多通知。

④通知是标注有某种注解的简单的Java方法。

⑤AspectJ支持5种类型的通知注解：

[1]@Before：前置通知，在方法执行之前执行

[2]@After：后置通知，在方法执行之后执行

[3]@AfterRunning：返回通知，在方法返回结果之后执行

[4]@AfterThrowing：异常通知，在方法抛出异常之后执行

[5]@Around：环绕通知，围绕着方法执行

**3.4AOP版的可以日志记录的计算器**

LogAspect中的代码

**[java]** view plain copy

print?

1. /\*\*
2. \* 1、告诉ioc这个组件的存在
3. \* 2、告诉ioc这是一个切面使用@Aspect
4. \* @author syl
5. \*
6. \*/
7. @Aspect
8. @Component
9. **public** **class** LogAspect {
11. /\*\*
12. \* try{
13. \*      @Before前置通知
14. \*      method.invoke();
15. \*      @AfterRunning返回通知
16. \* }catch(e){
17. \*      @AfterThrowing：异常通知，
18. \* }
19. \* @After
20. \*
21. \* 告诉Spring这些放在都在那个方法的哪个位置执行
22. \* 1）、告诉位置
23. [1]@Before：前置通知，在方法执行之前执行
24. [2]@After：后置通知，在方法执行最终结束之后执行。
25. 如果没异常
26. [3]@AfterRunning：返回通知，在方法返回结果之后执行
27. [4]@AfterThrowing：异常通知，在方法抛出异常之后执行

30. 1、编写切入点表达式，来告诉spring是切入哪个方法的这个位置
31. \*/
32. @Before(value="execution(public \* \*.add(int, int))")
33. **public** **void** logStart(){
34. System.out.println("AOP日志，方法开始");
35. }
37. @After(value="execution(public \* \*.add(int, int))")
38. **public** **void** logEnd(){
39. System.out.println("AOP日志，方法最终结束");
40. }
42. @AfterThrowing(value="execution(public \* \*.add(int, int))")
43. **public** **void** logException(){
44. System.out.println("AOP日志，方法出现异常");
45. }
47. @AfterReturning(value="execution(public \* \*.add(int, int))")
48. **public** **void** logReturn(){
49. System.out.println("AOP日志，方法正常执行");
50. }

**applicationContext中的内容**

**[html]** view plain copy

print?

1. <!--1、自动扫描所有的组件  -->
2. **<context:component-scan** base-package="com.atguigu"**></context:component-scan>**
3. <!--2、开启基于注解的aop功能  -->
4. **<aop:aspectj-autoproxy></aop:aspectj-autoproxy>**
5. <!--
6. 0）、被代理对象也得先加入到ioc中
7. 1）、编写日志切面，并加入到ioc中，@Component @Aspect
8. 2）、给切面编写通知方法。
9. 3）、加上何时何地切入的注解。切入点表达式
10. 4）、开启基于注解的aop功能即可
11. --**>**

**4、AOP细节**

**1切入点表达式**

**1.1作用**

通过表达式的方式定位一个或多个具体的连接点。

**1.2语法细节**

①切入点表达式的语法格式:execution([权限修饰符] [返回值类型] [简单类名/全类名] [方法名]([参数列表]))

②举例说明

表达式 execution(\* com.atguigu.spring.ArithmeticCalculator.\*(..))

含义 ArithmeticCalculator接口中声明的所有方法；第一个“\*”代表任意修饰符及任意返回值；第二个“\*”代表任意方法；

“..”匹配任意数量、任意类型的参数；若目标类、接口与该切面类在同一个包中可以省略包名。

表达式 execution(public \* ArithmeticCalculator.\*(..))

含义 ArithmeticCalculator接口的所有公有方法

表达式 execution(public double ArithmeticCalculator.\*(..))

含义 ArithmeticCalculator接口中返回double类型数值的方法

表达式 execution(public double ArithmeticCalculator.\*(double, ..))

含义 第一个参数为double类型的方法；“..” 匹配任意数量、任意类型的参数。

表达式 execution(public double ArithmeticCalculator.\*(double, double))

含义 参数类型为double，double类型的方法

③在AspectJ中，切入点表达式可以通过 “&&”、“||”、“!”等操作符结合起来。

表达式 execution (\* \*.add(int,..)) || execution(\* \*.sub(int,..))

含义 任意类中第一个参数为int类型的add方法或sub方法

**1.3AOP细节-切入点表达式应用到实际的切面类中**

**[java]** view plain copy

print?

1. //1、不能用本身的类型去找，只能用接口类型
2. //2、IOC容器中保存的不是这个对象的本身，而是代理对象
3. //3、为什么接口类型又是可以的.因为jdk在创建动态代理的时候，
4. //需要被代理对象的接口
5. /\*Calculator bean = ioc.getBean(Calculator.class);
6. bean.add(1, 2);
7. System.out.println(bean.getClass());\*/
8. MathCalculator bean = ioc.getBean(MathCalculator.**class**);
9. bean.add(1, 2);
10. System.out.println(bean.getClass());

**2当前连接点细节**

**2.1概述**

切入点表达式通常都会是从宏观上定位一组方法，和具体某个通知的注解结合起来就能够确定对应的连接点。那么就一个具体的连接点而言，我们可能会关心这个连接点的一些具体信息，

例如：当前连接点所在方法的方法名、当前传入的参数值等等。这些信息都封装在JoinPoint接口的实例对象中。

**3通知**

**3.1概述**

在具体的连接点上要执行的操作。

一个切面可以包括一个或者多个通知。

通知所使用的注解的值往往是切入点表达式。

**3.2前置通知**

前置通知：在方法执行之前执行的通知

使用@Before注解

**3.3后置通知**

后置通知：后置通知是在连接点完成之后执行的，即连接点返回结果或者抛出异常的时候

使用@After注解

**3.4返回通知**

返回通知：无论连接点是正常返回还是抛出异常，后置通知都会执行。如果只想在连接点返回的时候记录日志，应使用返回通知代替后置通知。

使用@AfterReturning注解

在返回通知中访问连接点的返回值

在返回通知中，只要将returning属性添加到@AfterReturning注解中，就可以访问连接点的返回值。该属性的值即为用来传入返回值的参数名称

必须在通知方法的签名中添加一个同名参数。在运行时Spring AOP会通过这个参数传递返回值

原始的切点表达式需要出现在pointcut属性中

**[java]** view plain copy

print?

1. @AfterReturning(value = "com.atguigu.aspect.LogAspect.mypoint()", returning = "res")
2. **public** **void** validReturn(JoinPoint joinPoint, Object res) {
3. Signature signature = joinPoint.getSignature();
4. String name = signature.getName();
5. System.out.println("AOP参数验证，【" + name + "】方法正常返回,返回值为：" + res);
6. }

**3.5异常通知**

异常通知：只在连接点抛出异常时才执行异常通知

将throwing属性添加到@AfterThrowing注解中，也可以访问连接点抛出的异常。Throwable是所有错误和异常类的顶级父类，所以在异常通知方法可以捕获到任何错误和异常。

如果只对某种特殊的异常类型感兴趣，可以将参数声明为其他异常的参数类型。然后通知就只在抛出这个类型及其子类的异常时才被执行

**[java]** view plain copy

print?

1. @AfterThrowing(value = "execution(public \* \*.\*(int, int))", throwing = "e")
2. **public** **void** logException(JoinPoint joinPoint, Throwable e) {
3. // 获取方法名
4. String name = joinPoint.getSignature().getName();
5. System.out.println("AOP日志，【" + name + "】方法出现异常：异常对象：" + abc);
6. }

**3.6环绕通知**

环绕通知是所有通知类型中功能最为强大的，能够全面地控制连接点，甚至可以控制是否执行连接点。

对于环绕通知来说，连接点的参数类型必须是ProceedingJoinPoint。它是 JoinPoint的子接口，允许控制何时执行，是否执行连接点。

在环绕通知中需要明确调用ProceedingJoinPoint的proceed()方法来执行被代理的方法。如果忘记这样做就会导致通知被执行了，但目标方法没有被执行。

注意：环绕通知的方法需要返回目标方法执行之后的结果，即调用 joinPoint.proceed();的返回值，否则会出现空指针异常。

**[java]** view plain copy

print?

1. /\*\*
2. \* 最强大通知。 一般不常用
3. \*
4. \* @param proceedingJoinPoint
5. \* @return
6. \*/
7. @Around(value = "com.atguigu.aspect.LogAspect.mypoint()")
8. **public** Object vaildAround(ProceedingJoinPoint proceedingJoinPoint) {
9. // proceedingJoinPoint封装了连接点的详细信息
10. // proceed，执行目标方法 method.invoke
11. Object proceed = **null**;
12. Object[] args = proceedingJoinPoint.getArgs();
13. **try** {
14. // //传入目标执行时需要的参数列表
15. // 前置通知
16. System.out.println("proceed...之前");

19. // method.invoke
20. // 目标方法执行完成后会有返回值，这个返回值一定return出去
21. proceed = proceedingJoinPoint.proceed(args);
22. // 返回通知
23. System.out.println("proceed...之后");
24. } **catch** (Throwable e) {
25. // e.printStackTrace();
26. // 异常通知
27. System.out.println("proceed...异常");
28. // 1、注意：
29. // 一定将这个异常继续抛出去，以方便外界都能收到这个异常
30. **throw** **new** RuntimeException(e);
31. } **finally** {
32. // 后置通知
33. System.out.println("proceed...结束");
34. }

37. **return** proceed;
38. }

**3.7重用切入点定义**

在编写AspectJ切面时，可以直接在通知注解中书写切入点表达式。但同一个切点表达式可能会在多个通知中重复出现。

在AspectJ切面中，可以通过@Pointcut注解将一个切入点声明成简单的方法。切入点的方法体通常是空的，因为将切入点定义与应用程序逻辑混在一起是不合理的。

切入点方法的访问控制符同时也控制着这个切入点的可见性。如果切入点要在多个切面中共用，最好将它们集中在一个公共的类中。在这种情况下，它们必须被声明为public。

在引入这个切入点时，必须将类名也包括在内。如果类没有与这个切面放在同一个包中，还必须包含包名。

其他通知可以通过方法名称引入该切入点

**[java]** view plain copy

print?

1. @Pointcut(value = "execution(public \* \*.\*(int, int))")
2. **public** **void** mypoint() {
3. }

6. @Before(value = "mypoint()")
7. **public** **void** logStart(JoinPoint joinPoint) {
8. Object[] args = joinPoint.getArgs();
9. String name = joinPoint.getSignature().getName();
10. System.out.println("AOP日志：【" + name + "】方法开始运行,参数是:"
11. + Arrays.asList(args));
12. }

15. @After("mypoint()")
16. **public** **void** logEnd(JoinPoint joinPoint) {
17. String name = joinPoint.getSignature().getName();
18. System.out.println("AOP日志：【" + name + "】方法运行结束!");
19. }

22. @AfterThrowing(value = "mypoint()", throwing = "e")
23. **public** **void** logException(JoinPoint joinPoint, Exception e) {
24. String name = joinPoint.getSignature().getName();
25. System.out.println("AOP日志：【" + name + "】方法运行出现异常:" + e);
26. }

29. @AfterReturning(value = "mypoint()", returning = "res")
30. **public** **void** logReturn(JoinPoint joinPoint, Object res) {
31. String name = joinPoint.getSignature().getName();
32. System.out.println("AOP日志：【" + name + "】方法正常结束,返回值为:" + res);
33. }

**3.8指定切面的优先级**

在同一个连接点上应用不止一个切面时，除非明确指定，否则它们的优先级是不确定的。

切面的优先级可以通过实现Ordered接口或利用@Order注解指定。

实现Ordered接口，getOrder()方法的返回值越小，优先级越高。

若使用@Order注解，序号出现在注解中

**[java]** view plain copy

print?

1. @Aspect
2. @Component
3. @Order(2)
4. **public** **class** LogAspect {
5. }
7. @Component
8. @Aspect
9. @Order(1)
10. **public** **class** ValidatorAspect {
11. }

先切入的最后出去。

**5、 以XML方式配置切面**

**1概述**

除了使用AspectJ注解声明切面，Spring也支持在bean配置文件中声明切面。这种声明是通过aop名称空间中的XML元素完成的。

正常情况下，基于注解的声明要优先于基于XML的声明。通过AspectJ注解，切面可以与AspectJ兼容，而基于XML的配置则是Spring专有的。由于AspectJ得到

越来越多的 AOP框架支持，所以以注解风格编写的切面将会有更多重用的机会。

**2配置细节**

在bean配置文件中，所有的Spring AOP配置都必须定义在<aop:config>元素内部。对于每个切面而言，都要创建一个<aop:aspect>元素来为具体的切面实现

引用后端bean实例。切面bean必须有一个标识符，供<aop:aspect>元素引用。

**[plain]** view plain copy

print?

1. <bean id="logAspect" class="com.atguigu.aspect.LogAspect"></bean>
2. <bean id="validatorAspect" class="com.atguigu.aspect.ValidatorAspect"></bean>
3. <bean id="mathCalulator" class="com.atguigu.calulator.MathCalulator"></bean>

6. <aop:config>
7. <aop:aspect ref="logAspect"></aop:aspect>
8. <aop:aspect ref="validatorAspect"></aop:aspect>
9. </aop:config>

**3声明切入点**

切入点使用<aop:pointcut>元素声明。

切入点必须定义在<aop:aspect>元素下，或者直接定义在<aop:config>元素下。

定义在<aop:aspect>元素下：只对当前切面有效

定义在<aop:config>元素下：对所有切面都有效

基于XML的AOP配置不允许在切入点表达式中用名称引用其他切入点。

**[plain]** view plain copy

print?

1. <aop:pointcut expression="execution(\* \*.\*(..))" id="mypoint"/>

**4声明通知**

在aop名称空间中，每种通知类型都对应一个特定的XML元素。

通知元素需要使用<pointcut-ref>来引用切入点，或用<pointcut>直接嵌入切入点表达式。

method属性指定切面类中通知方法的名称

**[plain]** view plain copy

print?

1. <aop:before method="logStart" pointcut="execution(\* \*.\*(..))"/

**5完整代码如下**

**[plain]** view plain copy

print?

1. <!--基于注解的AOP
2. 1、将被代理的对象和切面类都要加入ioc容器中
3. 2、配置切面类的每个通知方法，都在何时何地执行
4. 3、开启基于注解的aop功能
5. -->
6. <!--基于XML的AOP  -->
7. <!--1、将这些组件加入到ioc中  -->
8. <bean id="mathCalculator" class="com.atguigu.inf.MathCalculator"></bean>
9. <bean id="aalidatorAspect" class="com.atguigu.aspect.AalidatorAspect"></bean>
10. <bean id="logAspect" class="com.atguigu.aspect.LogAspect"></bean>

13. <!--2、配置切面类的详细信息,依赖于aop名称空间  -->
14. <aop:config>
15. <aop:pointcut expression="execution(\* \*.\*(..))" id="mypoint"/>
16. <aop:pointcut expression="execution(\* \*.add(..))" id="mypoint1"/>
17. <!-- 配置第二个切面，默认根据切面配置的优先顺序执行-->
18. <aop:aspect ref="aalidatorAspect" order="1">
19. <aop:before method="validStart" pointcut-ref="mypoint"/>
20. <aop:after-returning method="validReturn" pointcut-ref="mypoint" returning="res"/>
21. </aop:aspect>
22. <!--使用aop:aspect配置一个切面  -->
23. <aop:aspect ref="logAspect" order="0">
24. <!-- 配置前置通知是哪个方法 -->
25. <aop:before method="logStart" pointcut="execution(\* \*.\*(..))"/>
26. <aop:after method="logEnd" pointcut-ref="mypoint"/>
27. <!--returning指定返回值  -->
28. <aop:after-returning method="logReturn" pointcut-ref="mypoint" returning="res"/>
29. <!--throwing指定异常参数的名 -->
30. <aop:after-throwing method="logException" pointcut-ref="mypoint" throwing="abc"/>
31. </aop:aspect>
32. </aop:config>