**[Spring AOP 详解](http://pandonix.iteye.com/blog/336873)**

**博客分类：**

* [J2EE](http://pandonix.iteye.com/category/20309)

[AOP](http://www.iteye.com/blogs/tag/AOP)[Spring](http://www.iteye.com/blogs/tag/Spring)[junit](http://www.iteye.com/blogs/tag/junit)[Bean](http://www.iteye.com/blogs/tag/Bean)[JDK](http://www.iteye.com/blogs/tag/JDK)

      此前对于AOP的使用仅限于声明式事务，除此之外在实际开发中也没有遇到过与之相关的问题。最近项目中遇到了以下几点需求，仔细思考之后，觉得采用AOP 来解决。一方面是为了以更加灵活的方式来解决问题，另一方面是借此机会深入学习Spring AOP相关的内容。本文是权当本人的自己AOP学习笔记，以下需求不用AOP肯定也能解决，至于是否牵强附会，仁者见仁智者见智。

1. 对部分函数的调用进行日志记录，用于观察特定问题在运行过程中的函数调用情况
2. 监控部分重要函数，若抛出指定的异常，需要以短信或邮件方式通知相关人员
3. 金控部分重要函数的执行时间

    事实上，以上需求没有AOP也能搞定，只是在实现过程中比较郁闷摆了。

1. 需要打印日志的函数分散在各个包中，只能找到所有的函数体，手动添加日志。然而这些日志都是临时的，待问题解决之后应该需要清除打印日志的代码，只能再次手动清除^\_^!
2. 类 似1的情况，需要捕获异常的地方太多，如果手动添加时想到很可能明天又要手动清除，只能再汗。OK，该需求相对比较固定，属于长期监控的范畴，并不需求临 时添加后再清除。然而，客户某天要求，把其中20%的异常改为短信提醒，剩下的80%改用邮件提醒。改之，两天后，客户抱怨短信太多，全部改成邮件提 醒...
3. 该需求通常用于监控某些函数的执行时间，用以判断系统执行慢的瓶颈所在。瓶颈被解决之后，烦恼同情况1

    终于下定决心，采用AOP来解决！代码如下：

    切面类TestAspect

**Java代码  [收藏代码](javascript:void())**

1. **package** com.spring.aop;
2. /\*\*
3. \* 切面
4. \*
5. \*/
6. **public** **class** TestAspect {
8. **public** **void** doAfter(JoinPoint jp) {
9. System.out.println("log Ending method: "
10. + jp.getTarget().getClass().getName() + "."
11. + jp.getSignature().getName());
12. }
14. **public** Object doAround(ProceedingJoinPoint pjp) **throws** Throwable {
15. **long** time = System.currentTimeMillis();
16. Object retVal = pjp.proceed();
17. time = System.currentTimeMillis() - time;
18. System.out.println("process time: " + time + " ms");
19. **return** retVal;
20. }
22. **public** **void** doBefore(JoinPoint jp) {
23. System.out.println("log Begining method: "
24. + jp.getTarget().getClass().getName() + "."
25. + jp.getSignature().getName());
26. }
28. **public** **void** doThrowing(JoinPoint jp, Throwable ex) {
29. System.out.println("method " + jp.getTarget().getClass().getName()
30. + "." + jp.getSignature().getName() + " throw exception");
31. System.out.println(ex.getMessage());
32. }
34. **private** **void** sendEx(String ex) {
35. //TODO 发送短信或邮件提醒
36. }
37. }

**Java代码  [收藏代码](javascript:void())**

1. **package** com.spring.service;
2. /\*\*
3. \* 接口A
4. \*/
5. **public** **interface** AService {
7. **public** **void** fooA(String \_msg);
9. **public** **void** barA();
10. }

**Java代码  [收藏代码](javascript:void())**

1. **package** com.spring.service;
2. /\*\*
3. \*接口A的实现类
4. \*/
5. **public** **class** AServiceImpl **implements** AService {
7. **public** **void** barA() {
8. System.out.println("AServiceImpl.barA()");
9. }
11. **public** **void** fooA(String \_msg) {
12. System.out.println("AServiceImpl.fooA(msg:"+\_msg+")");
13. }
14. }

**Java代码  [收藏代码](javascript:void())**

1. **package** com.spring.service;
3. /\*\*
4. \*   Service类B
5. \*/
6. **public** **class** BServiceImpl {
8. **public** **void** barB(String \_msg, **int** \_type) {
9. System.out.println("BServiceImpl.barB(msg:"+\_msg+" type:"+\_type+")");
10. **if**(\_type == 1)
11. **throw** **new** IllegalArgumentException("测试异常");
12. }
14. **public** **void** fooB() {
15. System.out.println("BServiceImpl.fooB()");
16. }
18. }

    ApplicationContext

**Java代码  [收藏代码](javascript:void())**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="
6. http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
8. http://www.springframework.org/schema/aop
9. http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"
10. **default**-autowire="autodetect">
11. <aop:config>
12. <aop:aspect id="TestAspect" ref="aspectBean">
13. <!--配置com.spring.service包下所有类或接口的所有方法-->
14. <aop:pointcut id="businessService"
15. expression="execution(\* com.spring.service.\*.\*(..))" />
16. <aop:before pointcut-ref="businessService" method="doBefore"/>
17. <aop:after pointcut-ref="businessService" method="doAfter"/>
18. <aop:around pointcut-ref="businessService" method="doAround"/>
19. <aop:after-throwing pointcut-ref="businessService" method="doThrowing" throwing="ex"/>
20. </aop:aspect>
21. </aop:config>
23. <bean id="aspectBean" **class**="com.spring.aop.TestAspect" />
24. <bean id="aService" **class**="com.spring.service.AServiceImpl"></bean>
25. <bean id="bService" **class**="com.spring.service.BServiceImpl"></bean>
27. </beans>

    测试类AOPTest

**Java代码  [收藏代码](javascript:void())**

1. **public** **class** AOPTest **extends** AbstractDependencyInjectionSpringContextTests {
3. **private** AService aService;
5. **private** BServiceImpl bService;
7. **protected** String[] getConfigLocations() {
8. String[] configs = **new** String[] { "/applicationContext.xml"};
9. **return** configs;
10. }

13. /\*\*
14. \* 测试正常调用
15. \*/
16. **public** **void** testCall()
17. {
18. System.out.println("SpringTest JUnit test");
19. aService.fooA("JUnit test fooA");
20. aService.barA();
21. bService.fooB();
22. bService.barB("JUnit test barB",0);
23. }
25. /\*\*
26. \* 测试After-Throwing
27. \*/
28. **public** **void** testThrow()
29. {
30. **try** {
31. bService.barB("JUnit call barB",1);
32. } **catch** (IllegalArgumentException e) {
34. }
35. }
37. **public** **void** setAService(AService service) {
38. aService = service;
39. }
41. **public** **void** setBService(BServiceImpl service) {
42. bService = service;
43. }
44. }

    运行结果如下：

**Java代码  [收藏代码](javascript:void())**

1. log Begining method: com.spring.service.AServiceImpl.fooA
2. AServiceImpl.fooA(msg:JUnit test fooA)
3. log Ending method: com.spring.service.AServiceImpl.fooA
4. process time: 0 ms
5. log Begining method: com.spring.service.AServiceImpl.barA
6. AServiceImpl.barA()
7. log Ending method: com.spring.service.AServiceImpl.barA
8. process time: 0 ms
9. log Begining method: com.spring.service.BServiceImpl.fooB
10. BServiceImpl.fooB()
11. log Ending method: com.spring.service.BServiceImpl.fooB
12. process time: 0 ms
13. log Begining method: com.spring.service.BServiceImpl.barB
14. BServiceImpl.barB(msg:JUnit test barB type:0)
15. log Ending method: com.spring.service.BServiceImpl.barB
16. process time: 0 ms
18. log Begining method: com.spring.service.BServiceImpl.barB
19. BServiceImpl.barB(msg:JUnit call barB type:1)
20. log Ending method: com.spring.service.BServiceImpl.barB
21. method com.spring.service.BServiceImpl.barB **throw** exception
22. 测试异常

    《Spring参考手册》中定义了以下几个AOP的重要概念，结合以上代码分析如下：

* 切面（Aspect） ：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”，在本例中，“切面”就是类TestAspect所关注的具体行为，例如，AServiceImpl.barA()的调用就是切面TestAspect所关注的行为之一。“切面”在ApplicationContext中<aop:aspect>来配置。
* 连接点（Joinpoint） ：程序执行过程中的某一行为，例如，AServiceImpl.barA()的调用或者BServiceImpl.barB(String \_msg, int \_type)抛出异常等行为。
* 通知（Advice） ：“切面”对于某个“连接点”所产生的动作，例如，TestAspect中对com.spring.service包下所有类的方法进行日志记录的动作就是一个Advice。其中，一个“切面”可以包含多个“Advice”，例如TestAspect
* 切入点（Pointcut） ：匹配连接点的断言，在AOP中通知和一个切入点表达式关联。例如，TestAspect中的所有通知所关注的连接点，都由切入点表达式execution(\* com.spring.service.\*.\*(..))来决定
* 目标对象（Target Object） ：被一个或者多个切面所通知的对象。例如，AServcieImpl和BServiceImpl，当然在实际运行时，Spring AOP采用代理实现，实际AOP操作的是TargetObject的代理对象。
* AOP代理（AOP Proxy） 在Spring AOP中有两种代理方式，JDK动态代理和CGLIB代理。默认情况下，TargetObject实现了接口时，则采用JDK动态代理，例如，AServiceImpl；反之，采用CGLIB代理，例如，BServiceImpl。强制使用CGLIB代理需要将 <aop:config> 的 proxy-target-class 属性设为true

       通知（Advice）类型

* 前置通知（Before advice） ：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext中在<aop:aspect>里面使用<aop:before>元素进行声明。例如，TestAspect中的doBefore方法
* 后通知（After 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）。ApplicationContext中在<aop:aspect>里面使用<aop:after>元素进行声明。例如，TestAspect中的doAfter方法，所以AOPTest中调用BServiceImpl.barB抛出异常时，doAfter方法仍然执行
* 返回后通知（After return advice） ：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext中在<aop:aspect>里面使用<after-returning>元素进行声明。
* 环绕通知（Around advice） ：包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext中在<aop:aspect>里面使用<aop:around>元素进行声明。例如，TestAspect中的doAround方法。
* 抛出异常后通知（After throwing advice） ： 在方法抛出异常退出时执行的通知。 ApplicationContext中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。例如，TestAspect中的doThrowing方法。

       切入点表达式

* 通常情况下，表达式中使用”execution“就可以满足大部分的要求。表达式格式如下：

**Java代码  [收藏代码](javascript:void())**

1. execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) **throws**-pattern?)

modifiers-pattern：方法的操作权限

ret-type-pattern：返回值

declaring-type-pattern：方法所在的包

name-pattern：方法名

parm-pattern：参数名

throws-pattern：异常

其中，除ret-type-pattern和name-pattern之外，其他都是可选的。上例中，execution(\* com.spring.service.\*.\*(..))表示com.spring.service包下，返回值为任意类型；方法名任意；参数不作限制的所有方法。

* 通知参数

可以通过args来绑定参数，这样就可以在通知（Advice）中访问具体参数了。例如，<aop:aspect>配置如下

**Java代码  [收藏代码](javascript:void())**

1. <aop:config>
2. <aop:aspect id="TestAspect" ref="aspectBean">
3. <aop:pointcut id="businessService"
4. expression="execution(\* com.spring.service.\*.\*(String,..)) and args(msg,..)" />
5. <aop:after pointcut-ref="businessService" method="doAfter"/>
6. </aop:aspect>
7. </aop:config>

TestAspect的doAfter方法中就可以访问msg参数，但这样以来AService中的barA()和BServiceImpl中的barB()就不再是连接点，因为execution(\* com.spring.service.\*.\*(String,..))只配置第一个参数为String类型的方法。其中，doAfter方法定义如下：

**Java代码  [收藏代码](javascript:void())**

1. **public** **void** doAfter(JoinPoint jp,String msg)

* 访问当前的连接点

任何通知（Advice）方法可以将第一个参数定义为 org.aspectj.lang.JoinPoint 类型。JoinPoint 接口提供了一系列有用的方法， 比如 getArgs() （返回方法参数）、getThis() （返回代理对象）、getTarget() （返回目标）、getSignature() （返回正在被通知的方法相关信息）和 toString()（打印出正在被通知的方法的有用信息。