自：https://blog.csdn.net/hxpjava1/article/details/55504513/

**spring多个AOP执行先后顺序**

##### 众所周知，spring声明式事务是基于AOP实现的，那么，如果我们在同一个方法自定义多个AOP，我们如何指定他们的执行顺序呢？网上很多答案都是指定order，order越小越是最先执行，这种也不能算是错，但有些片面。

* 配置AOP执行顺序的三种方式：

1. 通过实现org.springframework.core.Ordered接口

**[java]** [view plain](http://blog.csdn.net/qqxhwwqwq/article/details/51678595) [copy](http://blog.csdn.net/qqxhwwqwq/article/details/51678595)

* 1. @Component
  2. @Aspect
  3. @Slf4j
  4. **public** **class** MessageQueueAopAspect1 **implements** Ordered{@Override
  5. **public** **int** getOrder() {
  6. // TODO Auto-generated method stub
  7. **return** 2;
  8. }
  10. }

1. 通过注解

**[java]** [view plain](http://blog.csdn.net/qqxhwwqwq/article/details/51678595) [copy](http://blog.csdn.net/qqxhwwqwq/article/details/51678595)

* 1. @Component
  2. @Aspect
  3. @Slf4j
  4. @Order(1)
  5. **public** **class** MessageQueueAopAspect1{
  7. ...
  8. }

1. 通过配置文件配置

**[html]** [view plain](http://blog.csdn.net/qqxhwwqwq/article/details/51678595) [copy](http://blog.csdn.net/qqxhwwqwq/article/details/51678595)

* 1. **<aop:config** expose-proxy="true"**>**
  2. **<aop:aspect** ref="aopBean" order="0"**>**
  3. **<aop:pointcut** id="testPointcut"  expression="@annotation(xxx.xxx.xxx.annotation.xxx)"**/>**
  4. **<aop:around** pointcut-ref="testPointcut" method="doAround" **/>**
  5. **</aop:aspect>**
  6. **</aop:config>**

我们在同一个方法上加以下两个AOP，看看究竟。

**[java]** [view plain](http://blog.csdn.net/qqxhwwqwq/article/details/51678595) [copy](http://blog.csdn.net/qqxhwwqwq/article/details/51678595)

1. @Component
2. @Aspect
3. @Slf4j
4. **public** **class** MessageQueueAopAspect1 **implements** Ordered{
6. @Resource(name="actionMessageProducer")
7. **private** IProducer<MessageQueueInfo> actionProducer;
9. @Pointcut("@annotation(com.xxx.annotation.MessageQueueRequire1)")
10. **private** **void** pointCutMethod() {
11. }
13. //声明前置通知
14. @Before("pointCutMethod()")
15. **public** **void** doBefore(JoinPoint point) {
16. log.info("MessageQueueAopAspect1:doBefore");
17. **return**;
18. }
20. //声明后置通知
21. @AfterReturning(pointcut = "pointCutMethod()", returning = "returnValue")
22. **public** **void** doAfterReturning(JoinPoint point,Object returnValue) {
23. log.info("MessageQueueAopAspect1:doAfterReturning");
24. }
26. //声明例外通知
27. @AfterThrowing(pointcut = "pointCutMethod()", throwing = "e")
28. **public** **void** doAfterThrowing(Exception e) {
29. log.info("MessageQueueAopAspect1:doAfterThrowing");
30. }
32. //声明最终通知
33. @After("pointCutMethod()")
34. **public** **void** doAfter() {
35. log.info("MessageQueueAopAspect1:doAfter");
36. }
38. //声明环绕通知
39. @Around("pointCutMethod()")
40. **public** Object doAround(ProceedingJoinPoint pjp) **throws** Throwable {
41. log.info("MessageQueueAopAspect1:doAround-1");
42. Object obj = pjp.proceed();
43. log.info("MessageQueueAopAspect1:doAround-2");
44. **return** obj;
45. }
47. @Override
48. **public** **int** getOrder() {
49. **return** 1001;
50. }
51. }

**[java]** [view plain](http://blog.csdn.net/qqxhwwqwq/article/details/51678595) [copy](http://blog.csdn.net/qqxhwwqwq/article/details/51678595)

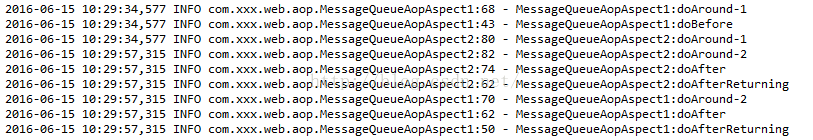
1. @Component
2. @Aspect
3. @Slf4j
4. **public** **class** MessageQueueAopAspect2 **implements** Ordered{
6. @Resource(name="actionMessageProducer")
7. **private** IProducer<MessageQueueInfo> actionProducer;
9. @Pointcut("@annotation(com.xxx.annotation.MessageQueueRequire2)")
10. **private** **void** pointCutMethod() {
11. }

14. //声明前置通知
15. @Before("pointCutMethod()")
16. **public** **void** doBefore(JoinPoint point) {
17. log.info("MessageQueueAopAspect2:doBefore");
18. **return**;
19. }
21. //声明后置通知
22. @AfterReturning(pointcut = "pointCutMethod()", returning = "returnValue")
23. **public** **void** doAfterReturning(JoinPoint point,Object returnValue) {
24. log.info("MessageQueueAopAspect2:doAfterReturning");
25. }
27. //声明例外通知
28. @AfterThrowing(pointcut = "pointCutMethod()", throwing = "e")
29. **public** **void** doAfterThrowing(Exception e) {
30. log.info("MessageQueueAopAspect2:doAfterThrowing");
31. }
33. //声明最终通知
34. @After("pointCutMethod()")
35. **public** **void** doAfter() {
36. log.info("MessageQueueAopAspect2:doAfter");
37. }
39. //声明环绕通知
40. @Around("pointCutMethod()")
41. **public** Object doAround(ProceedingJoinPoint pjp) **throws** Throwable {
42. log.info("MessageQueueAopAspect2:doAround-1");
43. Object obj = pjp.proceed();
44. log.info("MessageQueueAopAspect2:doAround-2");
45. **return** obj;
46. }
48. @Override
49. **public** **int** getOrder() {
50. **return** 1002;
51. }
52. }

**[java]** [view plain](http://blog.csdn.net/qqxhwwqwq/article/details/51678595) [copy](http://blog.csdn.net/qqxhwwqwq/article/details/51678595)

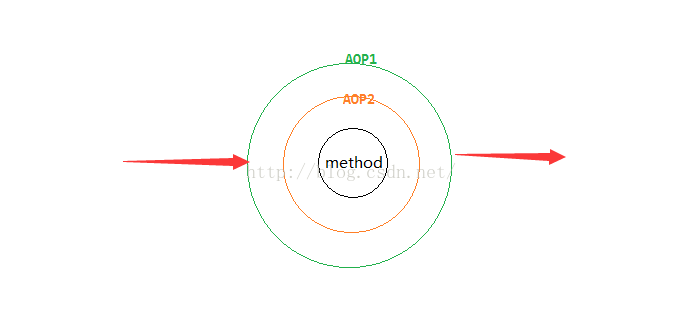
1. @Transactional(propagation=Propagation.REQUIRES\_NEW)
2. @MessageQueueRequire1
3. @MessageQueueRequire2
4. **public** PnrPaymentErrCode bidLoan(String id){
5. ...
6. }

看看执行结果：



从上面的测试我们看到，确实是order越小越是最先执行，但更重要的是最先执行的最后结束。

这个不难理解，[**spring**](http://lib.csdn.net/base/17) AOP就是面向切面编程，什么是切面，画一个图来理解下：



**由此得出：spring aop就是一个同心圆，要执行的方法为圆心，最外层的order最小。从最外层按照AOP1、AOP2的顺序依次执行doAround方法，doBefore方法。然后执行method方法，最后按照AOP2、AOP1的顺序依次执行doAfter、doAfterReturn方法。也就是说对多个AOP来说，先before的，一定后after。**

**如果我们要在同一个方法事务提交后执行自己的AOP，那么把事务的AOP order设置为2，自己的AOP order设置为1，然后在doAfterReturn里边处理自己的业务逻辑。**