依赖

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

切入点表达式（@Pointcut）和通知

1. 切入点表达式（bean, within, execution, annotation）
2. bean

bean(“UserServiceImpl”) 具体指某一个类

bean(“\*ServiceImpl”) 指所有的类

1. within

within(“com.service.UserServiceImpl”) 具体指某一个类里面的所有的方法

within(“com.service.\*”) 当前包下面的所有的类中的所有的方法

within(“com.service..\*”) 当前包以及当前包下面的所有的子包里面的所有的方法

1. execution

语法（返回值类型 包名.类名.方法名(参数)）

execution(void com.service.UserServiceImpl.addUser())

execution(void com.service.UserServiceImpl.addUser(String))

execution(\* com.service..\*.\*(..))

1. annotation

见下面的代码示例 二

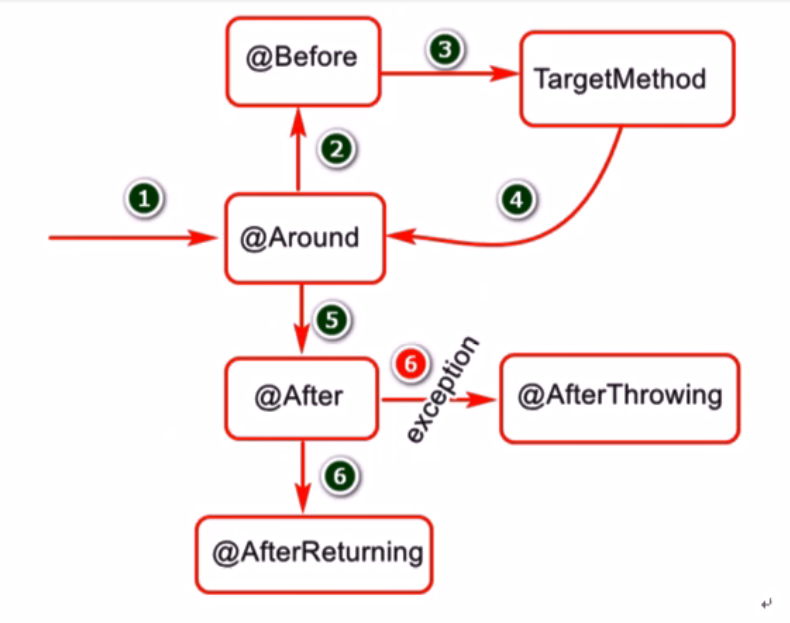
1. 通知 （@round，@Before， @Around， @AfterReturning， @AfterThrowing 等5种）

正确的执行顺序

@Around执行proceed() 方法之前 --> @Before --> @Around执行proceed() 方法之后 --> @After --> @AfterReturning

异常的执行顺序

@Around执行proceed() 方法之前 --> @Before --> --> @After --> @AfterThrowing



代码示例： 类上面加上 @Aspect @Component 这两个注解

一、作用在所有的类上面或者具体某一个类上面的代码

// 1. 切入点表达式

@Pointcut("bean(\*ServiceImpl)") // 可以写一个具体的serviceImpl，也可以用\*表示所有的

public void doPointcut() {}

// 环绕通知

@Around("doPointcut()")

public Object around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable{

log.info("start:" + System.currentTimeMillis());

Object[] args = proceedingJoinPoint.getArgs();

String name = proceedingJoinPoint.getSignature().getName();

Object proceed = proceedingJoinPoint.proceed();

log.info("end: "+ System.currentTimeMillis());

return proceed;

}

1. 使用自定义注解作用在某个方法上面
2. 新建一个自定义注解

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface RequireCache {

String operation() default “”;

}

1. 写aop切面编程

@Aspect

@Component

public class SysCacheAspect {

@Pointcut("@annotation(com.cy.dong\_ba.pj.sys.aop.RequireCache)") // 此处指向自定//义注解的包名+类名上

public void doCaches() {}

@Around("doCaches()")

public Object doCache(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

System.err.println("目标方法前");

Object proceed = proceedingJoinPoint.proceed();

System.err.println("执行目标方法之后");

// 获取目标方法

Method realMethod = proceedingJoinPoint.getTarget().getClass().getDeclaredMethod(proceedingJoinPoint.getSignature().getName(), ((MethodSignature)proceedingJoinPoint.getSignature()).getParameterTypes());

// 获取自定义注解

RequireCache annotation = realMethod.getAnnotation(RequireCache.class);

// 获取到自定义注解上面的字段值

String operation = annotation.operation();

return proceed;

}

}

3、在需要的方法上面加上 @RequireCache(operation=”在上面可以拿到这个值”) 注解即可实现

事务：

原子性， 一致性， 隔离性， 持久性

@Transactional(timeout=30, // 时间

rollbackFor=RuntimeException.class,//回滚异常的类型

isolation=”” // 隔离级别（见下面的详细解释）

)

多个事务并发执行时可能会出现 1、脏读，2、幻读，3、不可重复读

1. 脏读：读到了一条别人没有提交（commit）的数据（实际数据库里面没有）

避免脏读：隔离级别设置为 isolation=”Isolation.READ\_COMMITED”

1. 幻读：统计100条数据， 在读的时候别人增数据或者删数据了，导致读的数据不正确

避免幻读：隔离级别设置为 isolation=”Isolation.READ\_COMMITED”，就是锁表

1. 不可重复度：同一个事务对同一条数据多次读取不一致

避免不可重复度：隔离级别设置为 isolation=”Isolation.REPEATABLE.READ”

就是在读的时候不让别人改这条数据（加行锁）

事务的传播性

不同的业务之间（比如UserServiceImpl里面的方法调用RoleServiceImpl里面的方法）的调用会有事务的传播性优先级较高的会影响优先级低的，解决的办法可以在优先级低的方法上开启一个新的事务 @Transactional(propagation = Propagation.REQUIRES\_NEW)