1.AOP切入点表达式

支持切点标识符

Spring AOP支持使用以下AspectJ切点标识符(PCD),用于切点表达式：

- **execution**: 用于匹配方法执行连接点。 这是使用Spring AOP时使用的主要切点标识符。 可以匹配到方法级别， 细粒度
- **within**: 只能匹配类这级，只能指定类， 类下面的某个具体的方法无法指定， 粗粒度
- this: 匹配实现了某个接口：`this(com.xyz.service.AccountService)`
- target: 限制匹配到连接点（使用Spring AOP时方法的执行），其中目标对象（正在代理的应用程序对象）是给定类型的实例。
- args: 限制与连接点的匹配（使用Spring AOP时方法的执行），其中变量是给定类型的实例。 AOP) where the arguments are instances of the given types.
- @target: 限制与连接点的匹配（使用Spring AOP时方法的执行），其中执行对象的类具有给定类型的注解。
- @args: 限制匹配连接点（使用Spring AOP时方法的执行），其中传递的实际参数的运行时类型具有给定类型的注解。
- @within: 限制与具有给定注解的类型中的连接点匹配（使用Spring AOP时在具有给定注解的类型中声明的方法的执行）。
- @annotation:限制匹配连接点（在Spring AOP中执行的方法具有给定的注解）。

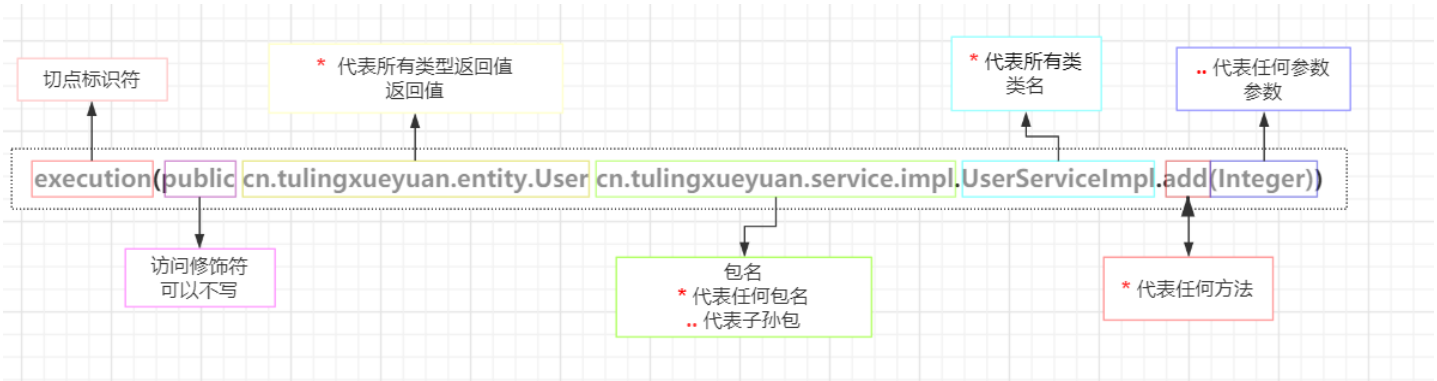
查看文档

语法：

Examples

Spring AOP users are likely to use the `execution` pointcut designator the most often. The format of an execution expression

```
访问修饰符   方法返回值（必须）   包名、类名   方法名称   方法参数
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)
            throws-pattern?)
```

访问修饰符：可不写 可以匹配任何一个访问修饰符

返回值：如果是jdk自带类型可以不用写完整限定名，如果是自定义类型需要写上完整限定名，如果被切入的方法返回值不一样可以使用* 代表所有的方法值都能匹配

包名：cn.* == cn.tulingxuyuean == cn.任意名字 但是只能匹配一级 比如 cn.tulingxueyuan.service就无法匹配

如果要cn.tulingxueyuan.service ==>cn.tulingxueyuan.service , cn.tulingxueyuan.* ==>cn.tulingxueyuan.service.impl就无法匹配

cn.tulingxueyuan..* ==>cn.tulingxueyuan.service.impl 可以匹配

类名： 可以写*，代表任何名字的类名。 也可以模糊匹配 *ServiceImpl==> UserServiceImpl ==>RoleServiceImpl
方法名：可以写*，代表任何方法。 也可以模糊匹配 *add==> useradd ==>roleadd
参数：如果是jdk自带类型可以不用写完整限定名，如果是自定义类型需要写上完整限定名。 如果需要匹配任意参数 可以写：..

1.within表达式
通过类名进行匹配 粗粒度的切入点表达式

within(包名.类名)

则这个类中的所有的连接点都会被表达式识别，成为切入点。

```
1         <aop:pointcut expression="within(cn.tulingxueyuan.service.UserServiceImpl)"
```

在within表达式中可以使用*号匹配符，匹配指定包下所有的类，注意，只匹配当前包，不包括当前包的子孙包。

```
1         <aop:pointcut expression="within(cn.tulingxueyuan.service.*)"
```

在within表达式中也可以用*号匹配符，匹配包

```
1         <aop:pointcut expression="within(cn.tulingxueyuan.*.*)"
```

在within表达式中也可以用..*号匹配符，匹配指定包下及其子孙包下的所有的类

```
1         <aop:pointcut expression="within(cn.tulingxueyuan..*)"
```

2.execution()表达式

细粒度的切入点表达式，可以以方法为单位定义切入点规则

语法:execution(返回值类型 包名.类名.方法名(参数类型,参数类型...))

例子1：

```
<aop:pointcut expression="execution(void cn.tulingxueyuan.service.UserServiceImpl.addUser(ja
```

该切入点规则表示，切出指定包下指定类下指定名称指定参数指定返回值的方法。

例子2：

```
<aop:pointcut expression="execution(* cn.tulingxueyuan.service.*.query())" id="pc1"/>
```

该切入点规则表示，切出指定包下所有的类中的query方法，要求无参，但返回值类型不限。

例子3：

```
<aop:pointcut expression="execution(* cn.tulingxueyuan.service..*.query())" id="pc1"/>
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的query方法，要求无参，但返回值类型不限。

例子4：

```
<aop:pointcut expression="execution(* cn.tulingxueyuan.service..*.query(int,java.lang.String
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的query方法，要求参数为int java.langString类型，但返回值类型不限。

例子5：

```
<aop:pointcut expression="execution(* cn.tulingxueyuan.service..*.query(..))" id="pc1"/>
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的query方法，参数数量及类型不限，返回值类型不限。

例子6：

```
<aop:pointcut expression="execution(* cn.tulingxueyuan.service..*.*(..))" id="pc1"/>
```

该切入点规则表示，切出指定包及其子孙包下所有的类中的任意方法，参数数量及类型不限，返回值类型不限。这种写法等价于within表达式的功能。

例子7：

```
<aop:pointcut expression="execution(* cn.tulingxueyuan.service..*.del*(..))" id="pc1"/>
```

3.合并切点表达式

您可以使用 &&, || 和 !等符号进行合并操作。也可以通过名字来指向切点表达式。

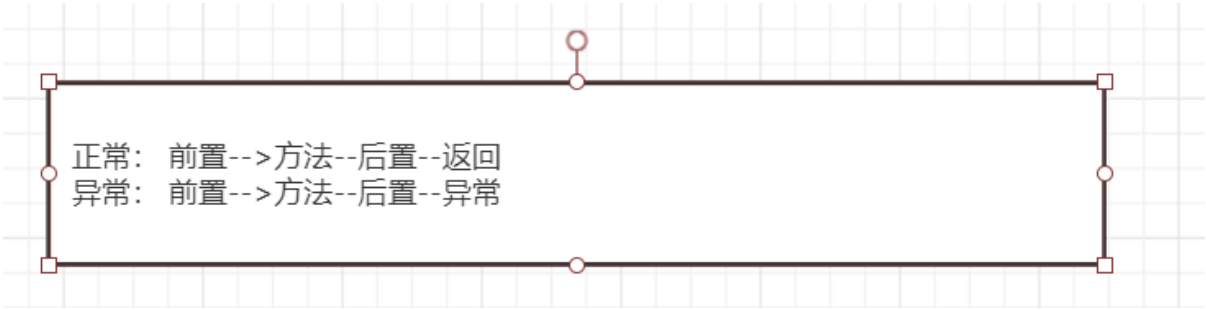
```
//&&：两个表达式同时
execution( public int cn.tulingxueyuan.inter.MyCalculator.*(..)) && execution(* *.*(int,int)
//||：任意满足一个表达式即可
execution( public int cn.tulingxueyuan.inter.MyCalculator.*(..)) && execution(* *.*(int,int)
//!：只要不是这个位置都可以进行切入
//&&：两个表达式同时
execution( public int cn.tulingxueyuan.inter.MyCalculator.*(..))
```

2、通知方法的执行顺序

在之前的代码中大家一直对通知的执行顺序有疑问，其实执行的结果并没有错，大家需要注意：

1、正常执行：@Before--->@After--->@AfterReturning

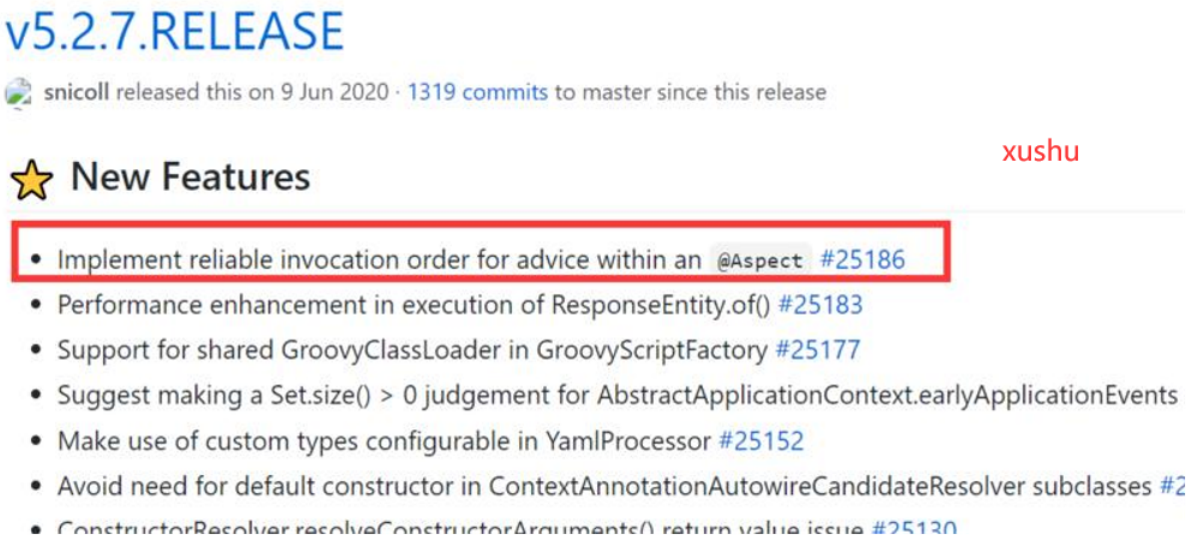
2、异常执行：@Before--->@After--->@AfterThrowing



Spring在5.2.7之后就改变的advice 的执行顺序。 在github官网版本更新说明中有说明：如图

1、正常执行：@Before--->@AfterReturning--->@After

2、异常执行：@Before--->@AfterThrowing--->@After



更新说明：https://github.com/spring-projects/spring-framewor... #25186链接：https://github.com/spring-projects/spring-framewor...

3、获取方法的详细信息

在上面的案例中，我们并没有获取Method的详细信息，例如方法名、参数列表等信息，想要获取的话其实非常简单，只需要添加JoinPoint参数即可。

LogUtil.java

```
1  package cn.tulingxueyuan.util;
2
3  import org.aspectj.lang.JoinPoint;
4  import org.aspectj.lang.annotation.*;
5  import org.springframework.stereotype.Component;
6
7  import java.util.Arrays;
8
9  @Component
10 @Aspect
```

```java
public class LogUtil {

    @Before("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
    public static void start(JoinPoint joinPoint){
        Object[] args = joinPoint.getArgs();
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法开始执行，参数是："+ Arrays.asList(args));
    }

    @AfterReturning("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))"
    public static void stop(JoinPoint joinPoint){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法执行完成，结果是：");

    }

    @AfterThrowing("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
    public static void logException(JoinPoint joinPoint){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法出现异常：");
    }

    @After("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
    public static void end(JoinPoint joinPoint){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法执行结束了.....");
    }
}
```

刚刚只是获取了方法的信息，但是如果需要获取**结果**，还需要添加另外一个方法参数，并且告诉spring使用哪个参数来进行结果接收

LogUtil.java

```java
    @AfterReturning(value = "execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int
                    returning = "result")
    public static void stop(JoinPoint joinPoint,Object result){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法执行完成，结果是："+result);

    }
```

也可以通过相同的方式来获取**异常的信息**

LogUtil.java

```
    @AfterThrowing(value = "execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,
    public static void logException(JoinPoint joinPoint,Exception exception){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法出现异常: "+exception);
    }
```

4、spring对通过方法的要求

spring对于通知方法的要求并不是很高，你可以任意改变方法的返回值和方法的访问修饰符，但是唯一不能修改的就是方法的参数，会出现参数绑定的错误，原因在于通知方法是spring利用反射调用的，每次方法调用得确定这个方法的参数的值。

LogUtil.java

```
    @After("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
    private int end(JoinPoint joinPoint,String aa){
        String name = joinPoint.getSignature().getName();
        System.out.println(name+"方法执行结束了......");
        return 0;
    }
```

5、表达式的抽取

如果在实际使用过程中，多个方法的表达式是一致的话，那么可以考虑将切入点表达式抽取出来：

a、随便生命一个没有实现的返回void的空方法

b、给方法上标注@Potintcut注解

```
package cn.tulingxueyuan.util;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;

import java.util.Arrays;

@Component
@Aspect
public class LogUtil {

    @Pointcut("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
    public void myPoint(){}
```

```
15
16      @Before("myPoint()")
17      public static void start(JoinPoint joinPoint){
18          Object[] args = joinPoint.getArgs();
19          String name = joinPoint.getSignature().getName();
20          System.out.println(name+"方法开始执行，参数是："+ Arrays.asList(args));
21      }
22
23      @AfterReturning(value = "myPoint()",returning = "result")
24      public static void stop(JoinPoint joinPoint,Object result){
25          String name = joinPoint.getSignature().getName();
26          System.out.println(name+"方法执行完成，结果是："+result);
27
28      }
29
30      @AfterThrowing(value = "myPoint()",throwing = "exception")
31      public static void logException(JoinPoint joinPoint,Exception exception){
32          String name = joinPoint.getSignature().getName();
33          System.out.println(name+"方法出现异常："+exception.getMessage());
34      }
35
36      @After("myPoint()")
37      private int end(JoinPoint joinPoint){
38          String name = joinPoint.getSignature().getName();
39          System.out.println(name+"方法执行结束了.....");
40          return 0;
41      }
42  }
```

6、环绕通知的使用

```
1  LogUtil.java
2  package cn.tulingxueyuan.util;
3
4  import org.aspectj.lang.JoinPoint;
5  import org.aspectj.lang.ProceedingJoinPoint;
6  import org.aspectj.lang.annotation.*;
7  import org.springframework.stereotype.Component;
8
9  import java.util.Arrays;
```

```
10
```

```java
@Component
@Aspect
public class LogUtil {
    @Pointcut("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
    public void myPoint(){}

    /**
     * 环绕通知是spring中功能最强大的通知
     * @param proceedingJoinPoint
     * @return
     */
    @Around("myPoint()")
    public Object myAround(ProceedingJoinPoint proceedingJoinPoint){
        Object[] args = proceedingJoinPoint.getArgs();
        String name = proceedingJoinPoint.getSignature().getName();
        Object proceed = null;
        try {
            System.out.println("环绕前置通知:"+name+"方法开始，参数是"+Arrays.asList(args));
            //利用反射调用目标方法，就是method.invoke()
            proceed = proceedingJoinPoint.proceed(args);
            System.out.println("环绕返回通知:"+name+"方法返回，返回值是"+proceed);
        } catch (Throwable e) {
            System.out.println("环绕异常通知"+name+"方法出现异常，异常信息是："+e);
        }finally {
            System.out.println("环绕后置通知"+name+"方法结束");
        }
        return proceed;
    }
}
```

总结：环绕通知的执行顺序是优于普通通知的，具体的执行顺序如下：
环绕前置-->普通前置-->目标方法执行-->环绕正常结束/出现异常-->环绕后置-->普通后置-->普通返回或者异常。
但是需要注意的是，如果出现了异常，那么环绕通知会处理或者捕获异常，普通异常通知是接收不到的，因此最好的方式是在环绕异常通知中向外抛出异常。
异常特殊说明：由于使用反射调用方法捕捉到的异常ex.getMessage=null ； 需要通过ex.getCause() 这一点细节注意一下

```java
public static void main(String[] args) throws Exception {
    try{
        Class<?> aClass = OrderController.class;
```

```
        Method add = aClass.getMethod("error");

        add.invoke(aClass.newInstance());
    }catch (Exception ex){
        ex.printStackTrace();
        System.out.println(ex.getCause().getMessage());
    }
}
```

3、基于配置的AOP配置

之前我们讲解了基于注解的AOP配置方式，下面我们开始讲一下基于xml的配置方式，虽然在现在的企业级开发中使用注解的方式比较多，但是你不能不会，因此需要简单的进行配置，注解配置快速简单，配置的方式共呢个完善。

1、将所有的注解都进行删除

2、添加配置文件

```
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:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/aop
       https://www.springframework.org/schema/aop/spring-aop.xsd
">

    <context:component-scan base-package="cn.tulingxueyuan"></context:component-scan>
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

    <bean id="logUtil" class="cn.tulingxueyuan.util.LogUtil2"></bean>
    <bean id="securityAspect" class="cn.tulingxueyuan.util.SecurityAspect"></bean>
    <bean id="myCalculator" class="cn.tulingxueyuan.inter.MyCalculator"></bean>
    <aop:config>
        <aop:pointcut id="globalPoint" expression="execution(public int cn.tulingxueyuan.int
        <aop:aspect ref="logUtil">
            <aop:pointcut id="mypoint" expression="execution(public int cn.tulingxueyuan.int
```

```
25              <aop:before method="start" pointcut-ref="mypoint"></aop:before>
26              <aop:after method="end" pointcut-ref="mypoint"></aop:after>
27              <aop:after-returning method="stop" pointcut-ref="mypoint" returning="result"></a
28              <aop:after-throwing method="logException" pointcut-ref="mypoint" throwing="excep
29              <aop:around method="myAround" pointcut-ref="mypoint"></aop:around>
30          </aop:aspect>
31      </aop:config>
32  </beans>
```

面试题

- Spring通知有哪些类型?
- 解释基于XML Schema方式的切面实现
- 解释基于注解的切面实现