

05-SpringAOP的使用详解

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

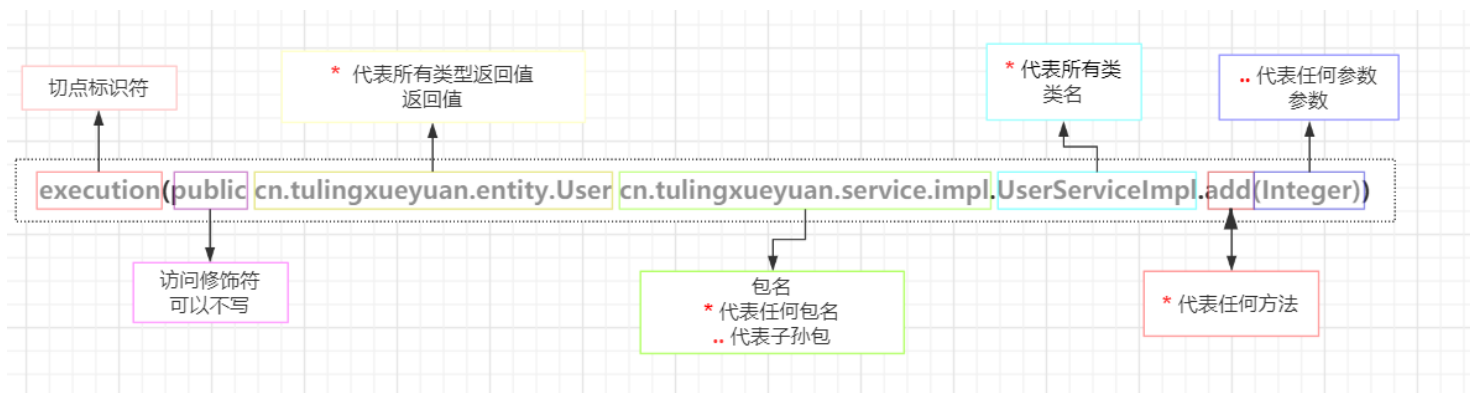
语法:

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?)`

<https://blog.csdn.net/Huangyuhua068>



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:

```
1 <aop:pointcut expression="execution(void cn.tulingxueyuan.service.UserServiceImpl.addUser(java.lang.String))" id="pc1"/>
```

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:

```
1 <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:

```
1 <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:

```
1 <aop:pointcut expression="execution(* cn.tulingxueyuan.service..*.query(int,java.lang.String))" 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，要求参数为int java.lang.String类型，但返回值类型不限。

例子5:

```
1 <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:

```
1 <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:

```
1 <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。

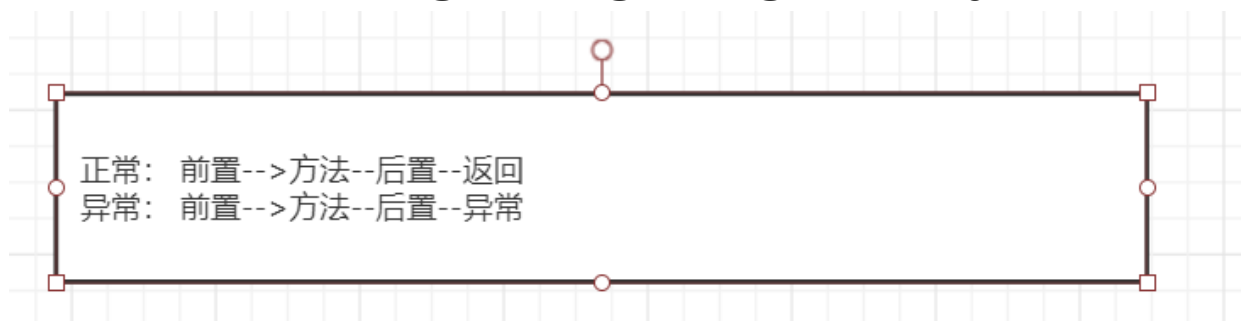
```
1 //&&: 两个表达式同时
2 execution( public int cn.tulingxueyuan.inter.MyCalculator.*(..)) && execution(* *.*(int,int) )
3 //||: 任意满足一个表达式即可
4 execution( public int cn.tulingxueyuan.inter.MyCalculator.*(..)) && execution(* *.*(int,int) )
5 //! : 只要不是这个位置都可以进行切入
6 //&&: 两个表达式同时
7 execution( public int cn.tulingxueyuan.inter.MyCalculator.*(..))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

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

v5.2.7.RELEASE

 snicoll released this on 9 Jun 2020 · 1319 commits to master since this release

★ New Features

xushu

- Implement reliable invocation order for advice within an `@Aspect` #25186
- Performance enhancement in execution of `ResponseEntity.of()` #25183
- Support for shared `GroovyClassLoader` in `GroovyScriptFactory` #25177
- Suggest making a `Set.size() > 0` judgement for `AbstractApplicationContext.earlyApplicationEvents`
- Make use of custom types configurable in `YamlProcessor` #25152
- Avoid need for default constructor in `ContextAnnotationAutowireCandidateResolver` subclasses #25151
- `ConstructorResolver.resolveConstructorArguments()` return value issue #25130

更新说明: <https://github.com/spring-projects/spring-framework...>

#25186链接: <https://github.com/spring-projects/spring-framework...>

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
11 public class LogUtil {
12
13     @Before("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
14     public static void start(JoinPoint joinPoint){
15         Object[] args = joinPoint.getArgs();
16         String name = joinPoint.getSignature().getName();
17         System.out.println(name+"方法开始执行, 参数是: "+ Arrays.asList(args));
18     }
19
20     @AfterReturning("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
21     public static void stop(JoinPoint joinPoint){
22         String name = joinPoint.getSignature().getName();
23         System.out.println(name+"方法执行完成, 结果是: ");
24     }
25
26
27     @AfterThrowing("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
28     public static void logException(JoinPoint joinPoint){
29         String name = joinPoint.getSignature().getName();
30         System.out.println(name+"方法出现异常: ");
31     }
32
33     @After("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
34     public static void end(JoinPoint joinPoint){
35         String name = joinPoint.getSignature().getName();
36         System.out.println(name+"方法执行结束了.....");
37     }
38 }
```

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

```
1  @AfterReturning(value = "execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))",
2      returning = "result")
3  public static void stop(JoinPoint joinPoint, Object result){
4      String name = joinPoint.getSignature().getName();
5      System.out.println(name+"方法执行完成，结果是: "+result);
6
7  }
```

也可以通过相同的方式来获取**异常的信息**

LogUtil.java

```
1  @AfterThrowing(value = "execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))", throwing = "exception")
2  public static void logException(JoinPoint joinPoint, Exception exception){
3      String name = joinPoint.getSignature().getName();
4      System.out.println(name+"方法出现异常: "+exception);
5  }
```

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

```
1  @After("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
2  private int end(JoinPoint joinPoint, String aa){
3      String name = joinPoint.getSignature().getName();
4      System.out.println(name+"方法执行结束了.....");
5      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

```
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
11 public class LogUtil {
12
13     @Pointcut("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
14     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
11 @Component
12 @Aspect
13 public class LogUtil {
14     @Pointcut("execution( public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))")
15     public void myPoint(){}
16
17     /**
18      * 环绕通知是spring中功能最强大的通知
19      * @param proceedingJoinPoint
20      * @return
21      */
22     @Around("myPoint()")
23     public Object myAround(ProceedingJoinPoint proceedingJoinPoint){
24         Object[] args = proceedingJoinPoint.getArgs();
25         String name = proceedingJoinPoint.getSignature().getName();
26         Object proceed = null;
27         try {
28             System.out.println("环绕前置通知:"+name+"方法开始, 参数是"+Arrays.asList(args));
29             //利用反射调用目标方法, 就是method.invoke()
30             proceed = proceedingJoinPoint.proceed(args);
31             System.out.println("环绕返回通知:"+name+"方法返回, 返回值是"+proceed);
32         } catch (Throwable e) {
33             System.out.println("环绕异常通知"+name+"方法出现异常, 异常信息是: "+e);
34         }finally {
35             System.out.println("环绕后置通知"+name+"方法结束");
36         }
37         return proceed;
38     }
39 }
```

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

```
1 public static void main(String[] args) throws Exception {
2     try{
3         Class<?> aClass = OrderController.class;
4         Method add = aClass.getMethod("error");
5
6         add.invoke(aClass.newInstance());
7     }catch (Exception ex){
8         ex.printStackTrace();
9         System.out.println(ex.getCause().getMessage());
10    }
11 }
```

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

```
1 aop.xml
2 <?xml version="1.0" encoding="UTF-8"?>
3 <beans xmlns="http://www.springframework.org/schema/beans"
4         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
5      xmlns:context="http://www.springframework.org/schema/context"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7      xsi:schemaLocation="http://www.springframework.org/schema/beans
8      http://www.springframework.org/schema/beans/spring-beans.xsd
9      http://www.springframework.org/schema/context
10     http://www.springframework.org/schema/context/spring-context.xsd
11     http://www.springframework.org/schema/aop
12     https://www.springframework.org/schema/aop/spring-aop.xsd
13 ">
14
15     <context:component-scan base-package="cn.tulingxueyuan"></context:component-scan>
16     <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
17
18     <bean id="logUtil" class="cn.tulingxueyuan.util.LogUtil2"></bean>
19     <bean id="securityAspect" class="cn.tulingxueyuan.util.SecurityAspect"></bean>
20     <bean id="myCalculator" class="cn.tulingxueyuan.inter.MyCalculator"></bean>
21     <aop:config>
22         <aop:pointcut id="globalPoint" expression="execution(public int cn.tulingxueyuan.inter.MyCalculator.*(int,int))"/>
23         <aop:aspect ref="logUtil">
24             <aop:pointcut id="mypoint" expression="execution(public int cn.tulingxueyuan.inter.MyCalculator.*(int,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"></aop:after-returning>
28             <aop:after-throwing method="logException" pointcut-ref="mypoint" throwing="exception"></aop:after-throwing>
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