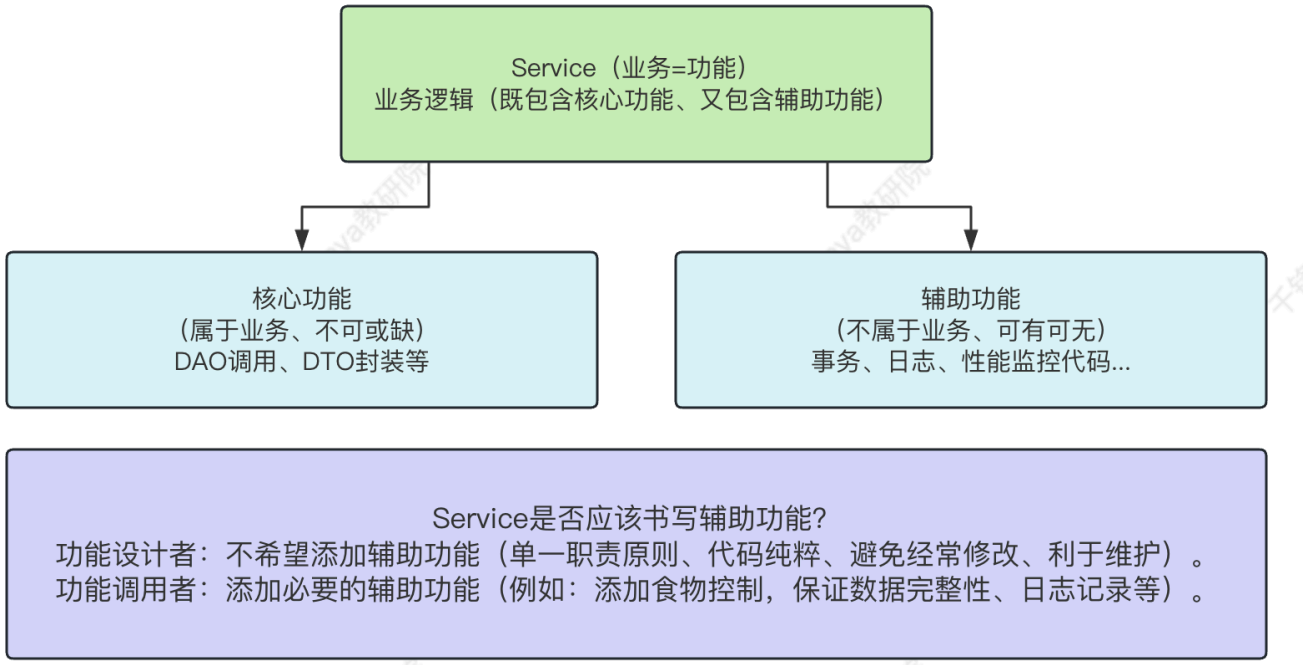
1.代理设计模式

1.1 概念

> 将核心功能与辅助功能（事务、日志、性能监控代码）分离，达到核心业务功能更纯粹、辅助业务功能可复用。

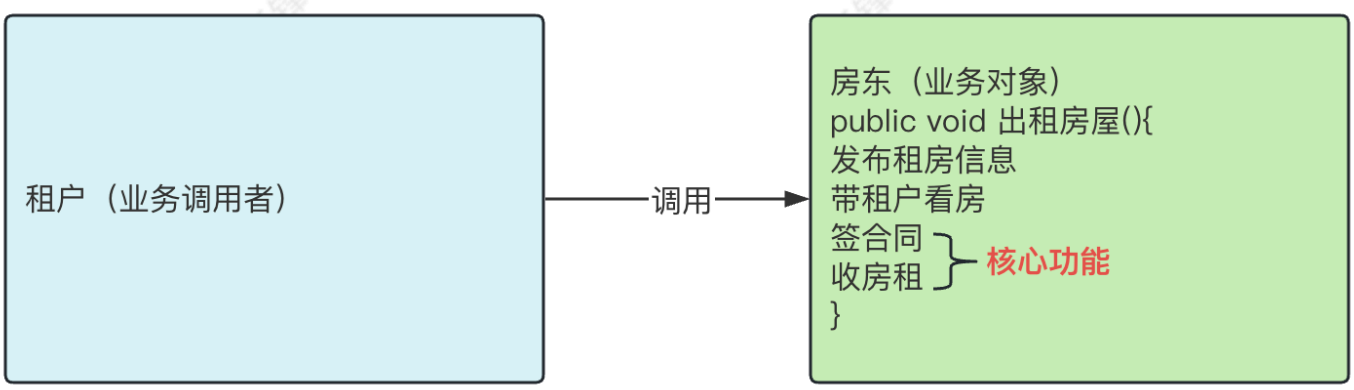
功能分离



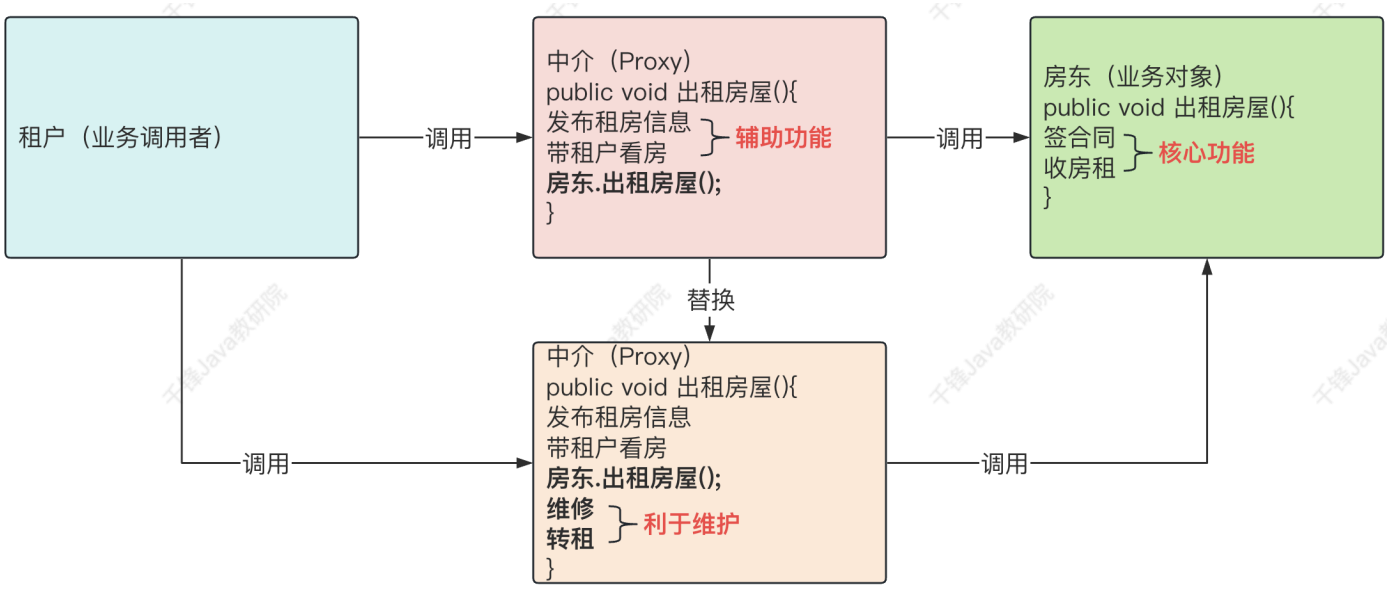
1.2 静态代理设计模式

> 通过代理类的对象，为原始类的对象（目标类的对象）添加辅助功能，更容易更换代理实现类、利于维护。

使用代理之前：



使用代理之后：



- 代理类 = 实现原始类相同接口 + 添加辅助功能 + 调用原始类的业务方法。
- 静态代理的问题
  - 代理类数量过多，不利于项目的管理。
  - 多个代理类的辅助功能代码冗余，修改时，维护性差。

1.3 动态代理设计模式

> 动态创建代理类的对象，为原始类的对象添加辅助功能。

1.3.1 JDK动态代理实现（基于接口）

```
//目标
final OrderService os = new OrderServiceImpl();
//额外功能
InvocationHandler handler = new InvocationHandler(){//1.设置回调函数（额外功能代码）
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        System.out.println("start...");
        method.invoke(os, args);
         System.out.println("end...");
        return null;
    }
};
//2.创建动态代理类
Object proxyObj = Proxy.newProxyInstance(ClassLoader , Interfaces , InvocationHandler);
```

1.3.2 CGlib动态代理实现（基于继承）

```
final OrderService os = new OrderServiceImpl();
Enhancer enh = new Enhancer();//1.创建字节码曾强对象
enh.setSuperclass(os.getClass());//2.设置父类（等价于实现原始类接口）
enh.setCallback(new InvocationHandler(){//3.设置回调函数（额外功能代码）
    @Override
    public Object invoke(Object proxy , Method method, Object[] args) throws Throwable{
        System.out.println("start...");
        Object ret = method.invoke(os,args);
        System.out.println("end...");
        return ret;
    }
});
OrderService proxy = (OrderService)enh.create();//4.创建动态代理类
proxy,createOrder();
```

在这里不要忘了引入Cglib的类库

```
        <dependency>
            <groupId>cglib</groupId>
            <artifactId>cglib</artifactId>
            <version>3.2.10</version>
        </dependency>
```

2. 使用Spring面向切面编程

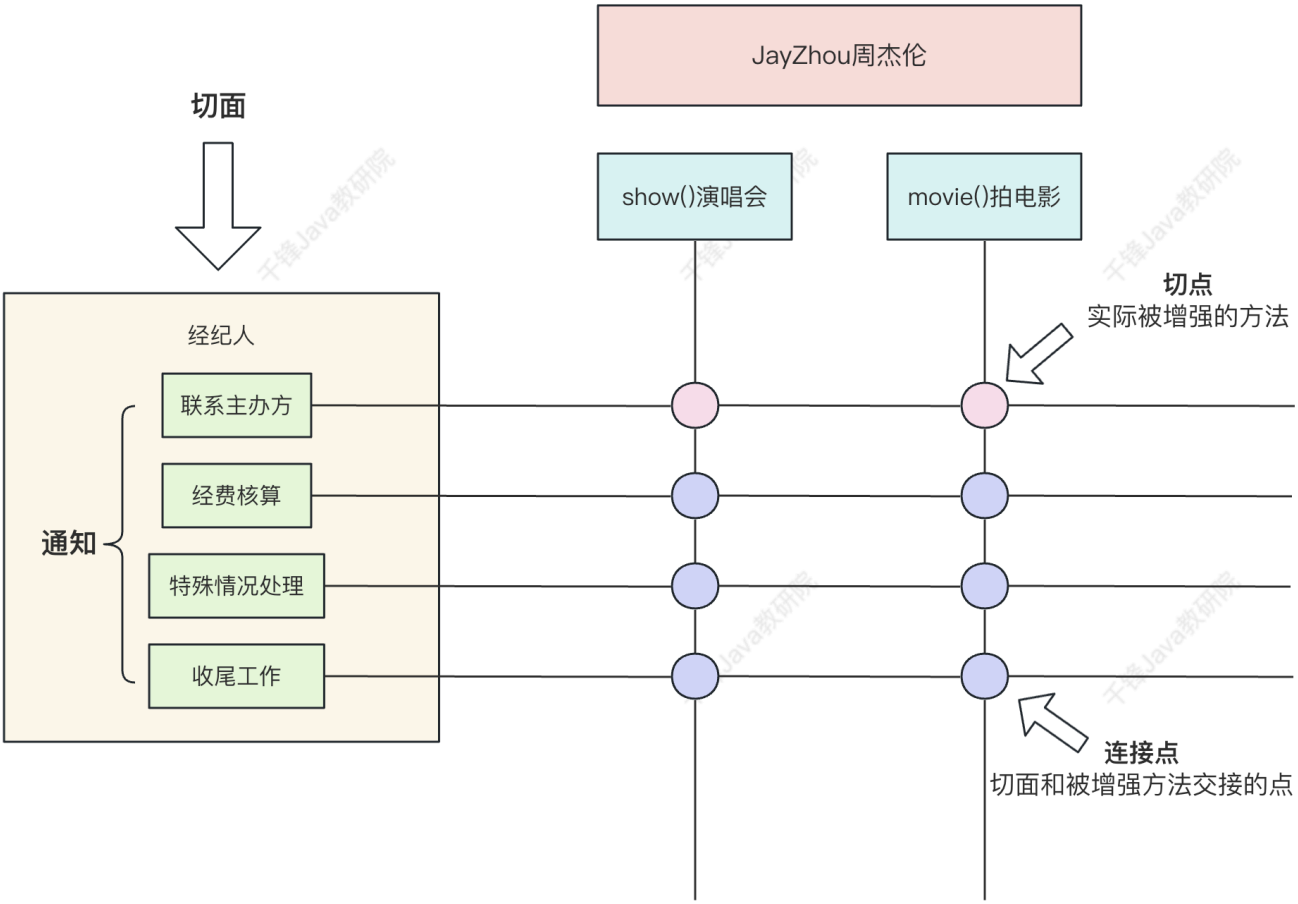
AOP（Aspect Oriented Programming），即面向切面编程，利用一种称为"横切"的技术，剖开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为"Aspect"，即切面。所谓"切面"，简单说就是那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块之间的耦合度，并有利于未来的可操作性和可维护性。

2.1 AOP 概念

让我们从定义一些核心AOP概念和术语开始。 这些术语不是特定于Spring的。 不幸的是，AOP术语不是特别直观。 但是，如果Spring使用自己的术语，那将更加令人困惑。

- 切面（Aspect）：指关注点模块化，这个关注点可能会横切多个对象。事务管理是企业级Java应用中有关横切关注点的例子。 在Spring AOP中，切面可以使用通用类基于模式的方式（schema-based approach）或者在普通类中以`@Aspect`注解（@AspectJ 注解方式）来实现。
- 连接点（Join point）：在程序执行过程中某个特定的点，例如某个方法调用的时间点或者处理异常的时间点。在Spring AOP中，一个连接点总是代表一个方法的执行。

- 通知（Advice）：在切面的某个特定的连接点上执行的动作。通知有多种类型，包括“around”, “before” and “after”等等。通知的类型将在后面的章节进行讨论。 许多AOP框架，包括Spring在内，都是以拦截器做通知模型的，并维护着一个以连接点为中心的拦截器链。
- 切点（Pointcut）：匹配连接点的断言。通知和切点表达式相关联，并在满足这个切点的连接点上运行（例如，当执行某个特定名称的方法时）。切点表达式如何和连接点匹配是AOP的核心：Spring默认使用AspectJ切点语义。
- 引入（Introduction）：声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被通知的对象上。例如，可以使用引入来使bean实现 `IsModified` 接口， 以便简化缓存机制（在AspectJ社区，引入也被称为内部类型声明（inter））。
- 目标对象（Target object）：被一个或者多个切面所通知的对象。也被称作被通知（advised）对象。既然Spring AOP是通过运行时代理实现的，那么这个对象永远是一个被代理（proxied）的对象。
- AOP代理（AOP proxy）:AOP框架创建的对象，用来实现切面契约（aspect contract）（包括通知方法执行等功能）。在Spring中，AOP代理可以是JDK动态代理或CGLIB代理。
- 织入（Weaving）：把切面连接到其它的应用程序类型或者对象上，并创建一个被被通知的对象的过程。这个过程可以在编译时（例如使用AspectJ编译器）、类加载时或运行时中完成。 Spring和其他纯Java AOP框架一样，是在运行时完成织入的。



Spring AOP包含以下类型的通知:

- 前置通知（Before advice）：在连接点之前运行但无法阻止执行流程进入连接点的通知（除非它引发异常）。
- 后置返回通知（After returning advice）:在连接点正常完成后执行的通知（例如，当方法没有抛出任何异常并正常返回时）。

- 后置异常通知（After throwing advice）：在方法抛出异常退出时执行的通知。
- 后置通知（总会执行）（After (finally) advice）：当连接点退出的时候执行的通知（无论是正常返回还是异常退出）。
- 环绕通知（Around Advice）:环绕连接点的通知，例如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它可以选择是否继续执行连接点或直接返回自定义的返回值又或抛出异常将执行结束。

环绕通知是最常用的一种通知类型。与AspectJ一样，在选择Spring提供的通知类型时，团队推荐开发者尽量使用简单的通知类型来实现需要的功能。例如， 如果只是需要使用方法的返回值来作缓存更新，虽然使用环绕通知也能完成同样的事情，但是仍然推荐使用后置返回通知来代替。使用最合适的通知类型可以让编程模型变得简单，还能避免很多潜在的错误。例如，开发者无需调用于环绕通知（用`JoinPoint`）的`proceed()`方法，也就不会产生调用的问题。

在Spring 2.0中，所有通知参数都是静态类型的，因此您可以使用相应类型的通知参数（例如，方法执行的返回值的类型）而不是`Object`数组。

切点和连接点匹配是AOP的关键概念，这个概念让AOP不同于其它仅仅提供拦截功能的旧技术。切入点使得通知可以独立于面向对象的层次结构进行定向。 例如，您可以将一个提供声明式事务管理的通知应用于跨多个对象（例如服务层中的所有业务操作）的一组方法。

2.3.Spring基于XML的AOP配置

@AspectJ会将切面声明为常规Java类的注解类型。 AspectJ项目引入了@AspectJ风格，并作为AspectJ 5发行版的一部分。Spring使用的注解类似于AspectJ 5， 使用AspectJ提供的库用来解析和匹配切点。

2.3.1 引入依赖

```
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.8</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.3.28</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.28</version>
        </dependency>
```

2.3.2 XML配置

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--扫描所有包-->
    <context:component-scan base-package="com.qf"/>

    <!--开启注解AOP-->
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

</beans>
```

2.3.3 打上注解

在作为切面的类上打上@Aspectj注解和@Component注解。

```
package com.qf.aspect;

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

/**
 * 经纪人
 * @Author: 索尔 VX：214490523
 * @技术交流社区： qfjava.cn
 */
@Aspect
@Component
public class Agent {


    @Before("execution(* com.qf.star..*.*(..))")
    public void before(){
        System.out.println("联系主办方");
    }
```

```
    @After("execution(* com.qf.star..*.*(..))")
    public void after(){
        System.out.println("经费核算");
    }

    @AfterThrowing("execution(* com.qf.star..*.*(..))")
    public void afterThrowing(){
        System.out.println("特殊情况处理");
    }

    @AfterReturning("execution(* com.qf.star..*.*(..))")
    public void afterReturning(){
        System.out.println("收尾工作");
    }


}
```

2.3.4 编写测试类

```
package com.qf.test;

import com.qf.star.JayZhou;
import com.qf.star.MayDay;
import com.qf.star.SuperStar;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @Author: 索尔 VX：214490523
 * @技术交流社区： qfjava.cn
 */
public class AopTest {
    @Test
    public void test1(){
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("classpath:spring.xml");
        SuperStar jay = context.getBean(SuperStar.class);
        System.out.println(jay.show("杭州"));
        System.out.println(jay.getClass());
    }

    @Test
    public void test2(){
```

```
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("classpath:spring.xml");
        MayDay mayDay = context.getBean(MayDay.class);
        System.out.println(mayDay.show("杭州"));
        System.out.println(mayDay.getClass());
    }
}
```

3.切点表达式

Spring AOP用户可能最常使用`execution`切点标识符，执行表达式的格式为：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-
pattern(param-pattern)throws-pattern?)
//注释
execution(访问权限修饰符 返回值类型 包名.类名.方法名(参数列表) 抛出异常类型)
```

除返回类型模式（上面片段中的`ret-type-pattern`）以外的所有部件、名称模式和参数模式都是可选的。返回类型模式确定要匹配的连接点的方法的返回类型必须是什么。 通常，可以使用`*`作为返回类型模式，它匹配任何返回类型。只有当方法返回给定类型时，完全限定的类型名称才会匹配。名称模式与方法名称匹配，可以将`*`通配符用作名称模式的全部或部分。 如果指定声明类型模式，则需要有后缀`.`将其加入到名称模式组件中。参数模式稍微复杂一点。`()`匹配没有参数的方法。`(..)`匹配任意个数的参数（0个或多个）。`(*)`匹配任何类型的单个参数。`(*,String)`匹配有两个参数而且第一个参数是任意类型，第二个必须是`String`的方法。

3.1 常见的切点表达式

以下示例显示了一些常见的切点表达式：

- 匹配任意公共方法的执行:

  ```
  execution(public * *(..))
  ```

- 匹配任意以`set`开始的方法:

  ```
  execution(* set*(..))
  ```

- 匹配定义了`AccountService`接口的任意方法:

  ```
  execution(* com.xyz.service.AccountService.*(..))
  ```

- 匹配定义在`service`包中的任意方法:

  ```
  execution(* com.xyz.service.*.*(..))
  ```

- 匹配定义在service包和其子包中的任意方法:

```
execution(* com.xyz.service..*.*(..))
```

- 匹配在service包中的任意连接点（只在Spring AOP中的方法执行）：

```
within(com.xyz.service.*)
```

- 匹配在service包及其子包中的任意连接点（只在Spring AOP中的方法执行）：

```
within(com.xyz.service..*)
```

- 匹配代理实现了`AccountService`接口的任意连接点（只在Spring AOP中的方法执行）：

```
this(com.xyz.service.AccountService)
```

- 匹配当目标对象实现了`AccountService`接口的任意连接点（只在Spring AOP中的方法执行）：

```
target(com.xyz.service.AccountService)
```

- 匹配使用了单一的参数，并且参数在运行时被传递时可以序列化的任意连接点（只在Spring的AOP中的方法执行）。：

```
args(java.io.Serializable)
```

  注意在这个例子中给定的切点不同于`execution(* *(java.io.Serializable))`. 如果在运行时传递的参数是可序列化的，则与execution匹配，如果方法签名声明单个参数类型可序列化，则与args匹配。
- 匹配当目标对象有`@Transactional`注解时的任意连接点（只在Spring AOP中的方法执行）。

```
@target(org.springframework.transaction.annotation.Transactional)
```

- 匹配当目标对象的定义类型有`@Transactional`注解时的任意连接点（只在Spring的AOP中的方法执行）：

```
@within(org.springframework.transaction.annotation.Transactional)
```

- 匹配当执行的方法有`@Transactional`注解的任意连接点（只在Spring AOP中的方法执行）：

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

- 匹配有单一的参数并且在运行时传入的参数类型有`@Classified`注解的任意连接点（只在Spring AOP中的方法执行）：

```
@args(com.xyz.security.Classified)
```

- 匹配在名为`tradeService`的Spring bean上的任意连接点（只在Spring AOP中的方法执行）：

```
bean(tradeService)
```

- 匹配以`Service`结尾的Spring bean上的任意连接点（只在Spring AOP中方法执行）：

```
bean(*Service)
```

3.2 切点表达式类型

表达式类型	描述
execution	匹配方法切入点
within	匹配指定类型
this	匹配代理对象实例的类型
target	匹配目标对象实例的类型
args	匹配方法参数
bean	匹配 bean 的 id 或名称
@within	匹配类型是否含有注解
@target	匹配目标对象实例的类型是否含有注解
@annotation	匹配方法是否含有注解
@args	匹配方法参数类型是否含有注解

```
/**
 * 日志功能
 * @Author: 索尔 VX：214490523
 * @技术交流社区：  qfjava.cn
 */
@Aspect
@Component
public class LogAspect {


    @Before("args(java.lang.Long)")
    public void before(){
        System.out.println("前置通知");
    }
```

```
    @After("within(com.qf.service..*)")
    public void after(){
        System.out.println("后置通知");
    }

    @After("target(com.qf.service.IProductService)")
    public void afterReturning(){
        System.out.println("后置返回通知");
    }

    public void afterThrowing(){
        System.out.println("后置异常通知");
    }
}
```

3.3 切点表达式组合

使用 `&&`、`||` 和 `!` 来组合多个切点表达式，表示多个表达式“与”、“或”和“非”的逻辑关系。

```
    @Before("execution(public com.qf.entity.Product com.qf..*.*(..)) &&
args(java.lang.Long)")
    public void before(){
        System.out.println("前置通知");
    }
```

4.通知的参数

Spring提供了全部类型的通知，这意味着需在通知签名中声明所需的参数，接着将会看到怎么声明参数以及上下文的值是如何在通知实体中被使用的。

4.1 获得切入方法的信息

任何通知方法都可以声明一个类型为 org.aspectj.lang.JoinPoint的参数作为其第一个参数。JoinPoint接口提供很多有用的方法：：

- getArgs(): 返回方法参数.
- getThis(): 返回代理对象.
- getTarget(): 返回目标对象.
- getSignature():返回正在通知的方法的描述.
- toString(): 打印方法被通知的有用描述.

```
    @Before("execution(public com.qf.entity.Product com.qf..*.*(..)) &&
args(java.lang.Long)")
    public void before(JoinPoint joinPoint){
        //获得方法名
        String methodName = joinPoint.getSignature().getName();
        //获得方法的参数
        Object[] args = joinPoint.getArgs();
        System.out.println("前置通知，方法名："+methodName+",参数："+ Arrays.asList(args));
    }
```

4.2 获得切入方法的返回值

获得切入点方法的返回值，需要在通知参数中添加returning参数，并在修饰的方法中加入返回值参数。

```
  @AfterReturning(value="target(com.qf.service.IProductService)",returning="returnValue
")
    public void afterReturning(Object returnValue){
        System.out.println("后置返回通知,返回结果："+returnValue);
    }
```

4.3 获得切入方法的异常信息

捕获切入点方法的异常信息，需要在通知参数中添加throwing参数，并在修饰的方法中加入异常参数。

```
    @AfterThrowing(value="execution(* *(..))",throwing = "exception")
    public void afterThrowing(Exception exception){
        System.out.println("后置异常通知,异常信息："+exception.getMessage());
    }
```

4.4 复用切点表达式

将切点表达式抽取成用@Pointcut注解修饰的方法。在@Pointcut注解中声明切点表达式。在通知中使用@Pointcut注解修饰的方法的方法名。

```
    @Pointcut("execution(public com.qf.entity.Product com.qf..*.*(..)) &&
args(java.lang.Long)")
    public void myPointcut(){
    }

    @Before("myPointcut()")
    public void before(JoinPoint joinPoint){
        //获得方法名
        String methodName = joinPoint.getSignature().getName();
        //获得方法的参数
        Object[] args = joinPoint.getArgs();
        System.out.println("前置通知，方法名："+methodName+",参数："+ Arrays.asList(args));
    }
```

5.环绕通知

环绕通知 “around” 匹配的方法执行运行。它有机会在方法执行之前和之后进行工作，并确定方法何时、 如何以及甚至是否真正执行。环绕通知经常用于需要在方法执行前或后在线程安全的情况下共享状态（例如开始和结束时间）。确认可使用的通知形式， 要符合最小匹配原则。

可以使用`aop:around`元素声明环绕通知。通知方法的第一个参数必须是`ProceedingJoinPoint`类型。在通知代码体中，调用`ProceedingJoinPoint`实现的`proceed()`会使匹配的方法继续执行。 `proceed`方法也可以通过传递 `Object[]`- 数组的值给原方法作为传入参数。

```
    @Around("myPointcut()")
    public Object around(ProceedingJoinPoint joinPoint){
        //获得方法名
        String methodName = joinPoint.getSignature().getName();
        //获得方法的参数
        Object[] args = joinPoint.getArgs();
        Object proceed = null;
        try {
            System.out.println("环绕前置通知，方法名："+methodName+",参数："+
Arrays.asList(args));
            //切入方法的调用。
            proceed = joinPoint.proceed(args);
            System.out.println("环绕后置返回通知。返回结果："+proceed);
        } catch (Throwable e) {
            System.out.println("环绕后置异常通知：方法名："+methodName+"，异常信
息："+e.getMessage());
        } finally {
            System.out.println("环绕后置通知，方法名："+methodName);
        }
        return proceed;
    }
```

6.使用XML配置SpringAOP

使用XML也能实现通知的配置。通过在XML里完成通知的切点表达式、参数等的配置。

```
    <aop:config>
        <aop:aspect ref="logAspectByXML">
            <aop:pointcut id="pc" expression="execution(public com.qf.entity.Product
com.qf..*.*(..)) &amp;&amp; args(java.lang.Long)"/>
            <aop:before method="before" pointcut-ref="pc" ></aop:before>
            <aop:after-returning method="afterReturning"
pointcut="target(com.qf.service.IProductService)" returning="returnValue"></aop:after-
returning>
            <aop:after-throwing method="afterThrowing" pointcut="execution(* *(..))"
throwing="exception"></aop:after-throwing>
            <aop:after method="after" pointcut="within(com.qf.service..*)"></aop:after>
            <aop:around method="around" pointcut-ref="pc"></aop:around>
        </aop:aspect>
    </aop:config>
```

注意，需要先关闭注解AOP。