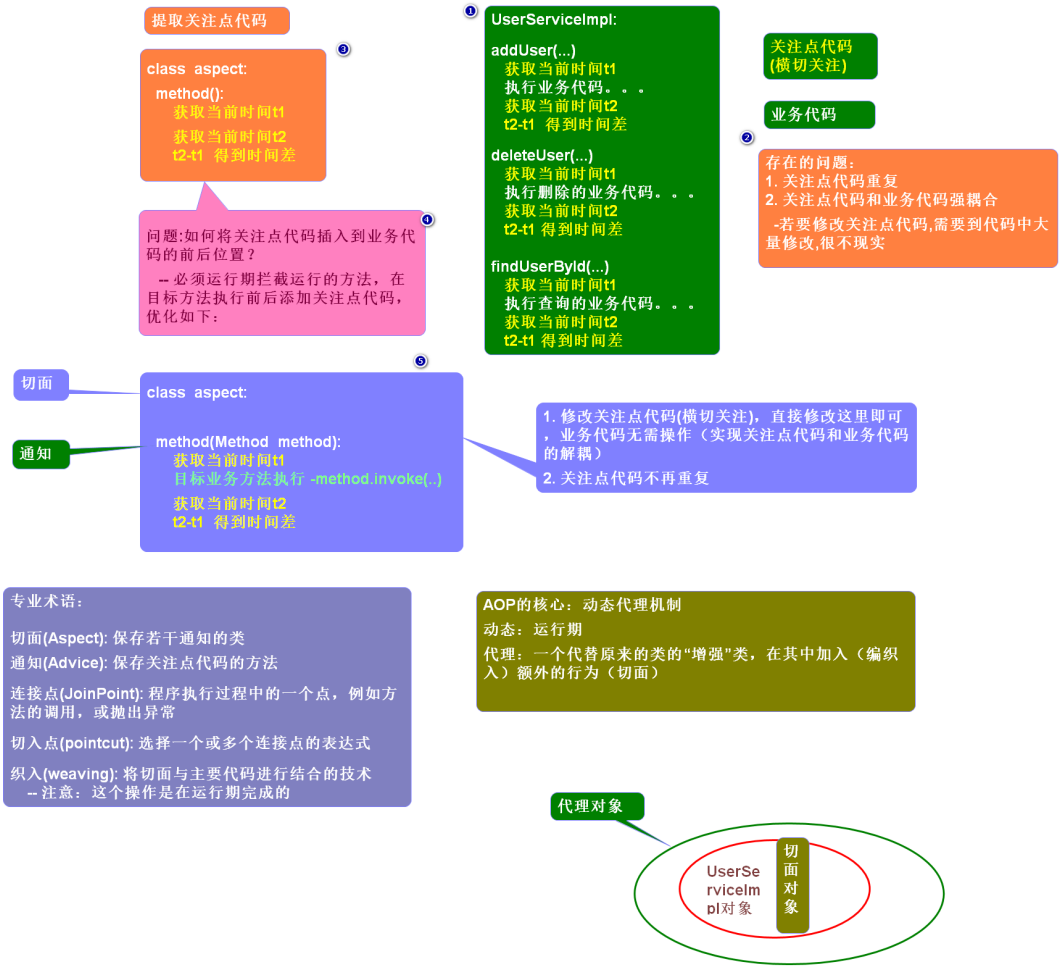
AOP

- 定义

> Aop是面向切面编程，用于实现关注点代码和业务代码的分离解耦的，实现关注点代码(横切关注)的模块化，解决代码缠绕(关注点和业务代码耦合)和代码分散(代码重复)问题。

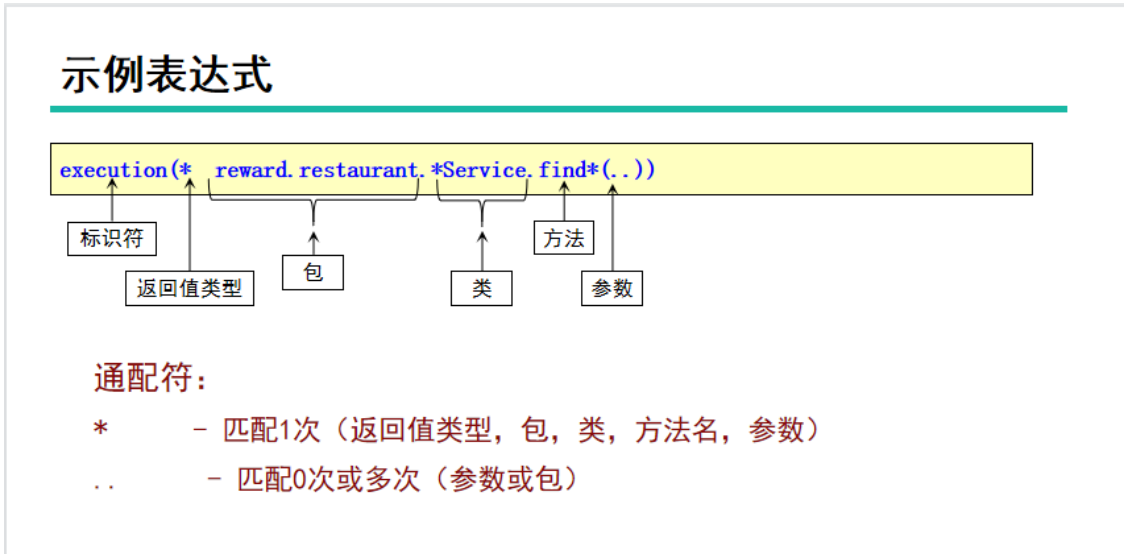


- 通知分类：

> 前置通知：在业务代码之前执行，若Advice抛出异常，则目标不会被调用 -- @Before
> 后置通知：在业务代码之后执行，无论目标是否抛出异常，都会被调用 -- @After
> 环绕通知：在业务代码前后环绕执行，可以停止异常的传播 -- @Around
> 返回后通知：方法返回指定对象后，执行通知代码 --@AfterReturning，带有returning属性
> 抛出异常后通知：方法抛出指定异常时才会调用该通知 -- @AfterThrowing，带有属性throwing

- 切入点表达式

> Spring AOP使用AspectJ的切入点表达式语言 -- AOP集成了AspectJ框架



- 注意点：

```
切入点表达式可以指定某注解
execution(@javax.annotation.security.RolesAllowed  void  send*(..))
任何名称使用send开头的void方法，且添加了@RolesAllowed注解

execution(@org..transaction.annotation.Transactional  *  *(..))
任何使用@Transactional注解进行标记的方法
```

代码实现

- 环绕通知：

```
 定义切面类，添加注解@Aspect和@Component
    @Component
    @Aspect
    public class TimeAspect {

        @Around("execution(* *..service.*.*(..))")
        public Object getTime(ProceedingJoinPoint joinPoint) throws
Throwable {
            long beforeTime = System.currentTimeMillis();
            Object returnVal = joinPoint.proceed();
            long afterTime = System.currentTimeMillis();

 System.out.println(joinPoint.getSignature().getDeclaringType()+"耗时："+
(afterTime-beforeTime)+"毫秒");
            return returnVal;
        }
    }

注意点：
    1. 通知方法返回值必须为Object，将原业务方法的返回值返回
    2. 通知方法要将proceed方法的异常抛出，表示若业务代码中发生异常，这里也抛出去，以便让控制器层的异常处理能够处理。
```

- 前置通知

```
@Before("execution(* *..service.*.*(..))")
public void log(){
    System.err.println("前置通知。。。。。");
    int i = 10/0;
}
```

注意点：若前置通知抛出异常，则目标方法不会被调用

- 后置通知

```
//切入点表达式外部化
@Pointcut("execution(* *..service.*.*(..))")
public void pointcutMethod(){}
```

无论目标是否抛出异常，都会被调用

- JoinPoint和ProceedingJoinPoint的区别

```
ProceedingJoinPoint只能用于around通知，可以让目标方法执行 -- proceed()
JoinPoint可以用于其他通知，用户获取信息 -- 提供了一些列get方法
   getThis()和getTarget()作用相同，都是获取目标对象
   getSignatrue():获取方法签名 -- 调用的目标方法的签名

getThis()=com.example.springaop2206demo.service.impl.PersonServiceImpl@528a5
b5c
getSignatrue=List
com.example.springaop2206demo.service.impl.PersonServiceImpl.list()
getTarget=com.example.springaop2206demo.service.impl.PersonServiceImpl@528a5
b5c

ProceedingJoinPoint继承自JoinPoint
```

- 返回后通知

```
@AfterReturning(value = "pointcutMethod()",returning = "list")
    public void afterReturningAdvice(List<Person> list){
//        System.out.println("getThis()="+joinPoint.getThis());
//        System.out.println("getSignatrue="+joinPoint.getSignature());
//        System.out.println("getTarget="+joinPoint.getTarget());
//        System.out.println("getClass="+joinPoint.getClass());
        System.out.println(list.size());
        System.out.println("返回值后通知。。。");

    }
```

可以使用JoinPoint也可以不使用,若目标方法中抛出异常，则目标方法不会返回指定类型，该通知不会执行

- 抛出异常后通知

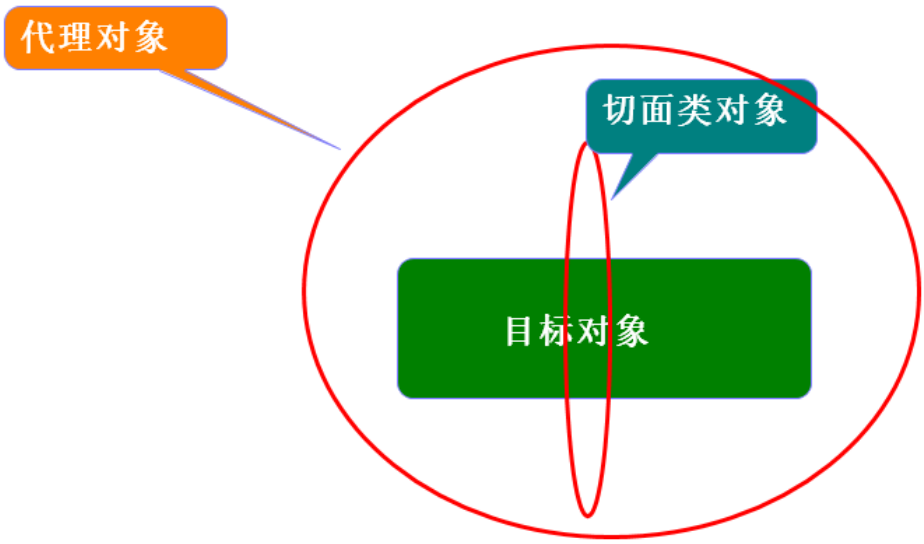
```
 @AfterThrowing(value = "pointcutMethod()",throwing = "e")
    public void afterThrowingAdvice(UserNotFoundException e){
        System.out.println("抛出异常后通知。。。");
//        throw new RuntimeException("用户不存在异常。。。。");
    }
```

@AfterThrowing的Advice不会停止异常的传播,但是它可以抛出一个类型不同的异常

AOP动态代理机制

1. aop是如何实现将切面类中的通知织入到切入点中的呢?

- AOP的底层实质上是通过使用动态代理机制将通知织入到连接点中的，如图所示：



- 如图所示，在运行期我们想在调用目标对象中的某些方法时，将切面类中的通知代码织入，将二者组合起来，这就是动态代理机制能帮我们完成的，并且将二者组合之后，会产生一个新对象，这个新对象就叫做代理对象，即代理对象是目标对象和切面类对象组合后的新对象。
- 动态代理机制具体是如何将目标对象和切面类对象织入到一起的呢？接下来，我们一起来学习动态代理机制。

2. 动态代理机制

- AOP采用的代理机制共有两种，分别为JDK动态代理与CGLIB动态代理，不同情况下使用不同的代理机制，接下来我们来学习这两种动态代理机制。
- JDK动态代理、

```
JDK动态代理，实质为使用JDK提供的api来生成代理对象，所以叫做JDK动态代理。
适用于为实现了接口的目标对象生成代理机制。
```

- CGLIB动态代理

> Cglib动态代理机制和jdk动态代理机制类似，是使用Cglib的API方法来生成代理对象的，所以叫做Cglib动态代理机制，适用于目标对象没有实现接口的情况。

3 AOP中两种代理机制的应用场景

- 其实，Spring是很智能的，在得到目标对象后，Spring会判断目标对象是否实现了接口，若实现了接口，则自动采用JDK动态代理机制为其生成代理对象；若目标对象没有实现接口，则自动采用CGLIB动态代理机制为其生成代理对象。
- 注意：从SpringBoot2.0版本开始，AOP的代理机制的选择发生了改变，默认的代理机制变为了CGLIB动态代理机制，不再根据目标对象是否实现接口而自动选择代理机制。

3. AOP的应用

- AOP在编程中应用场景非常多，经常见到的有：
    - Spring的声明式事务管理
    - 性能统计
    - 记录日志
    - 安全方面