AOP

一，概述

- DI有利于应用对象之间的解耦，而AOP可以实现横切关注点与它们影响的对象之间的解耦。
- AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。
- AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。
    - AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。
    - Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。
- Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：
    - JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。
    - 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。
- 静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

二，5种类型的通知

- 前置通知：在目标方法前调用通知功能
- 后置通知：在目标方法后调用通知功能
- 返回通知：在目标成功执行之后调用通知
- 异常通知：在目标方法抛出异常之后调用通知
- 环绕通知：通知包裹了被通知的方法，在被通知的方法调用前和调用后执行自定义行为。

三，实现方式

- 注入式AspectJ切面

```
@Aspect
public class AspectClass {
    @Before("execution(** com_7_4_3.TrackCounter.Count(int))")
    public void Count_test(int count_num) {
        System.out.println("前置通知");
```

```
    }
    //后置，环绕同理。
}


//通过注解来实现自定义切口函数
@Pointcut("execution(** com_7_4_3.TrackCounter.Count(int))")
public void perform(){}

@Before("performance()")
public void Count_test(int count_num) {
    System.out.println("前置通知");
}

//通过XML来实现自定义切口函数，aop直接能够让我们在Spring中声明切面，不用再去启动代理操做
<bean id="audience" class="com_8_4_4.Audience"></bean>
    <!-- 通过XML配置切面以及向切点向连接点传入参数 -->
    <aop:config>
        <aop:aspect ref="audience">
            <aop:pointcut id="performance"
                expression="execution(** com_8_4_4.Performance.perform(int)) and
args(count)"></aop:pointcut>     //自定义切点函数名称
            <aop:before method="silenceCePhone" pointcut-ref="performance" />
        </aop:aspect>
    </aop:config>
```

如果只是注册了切面是远远不行的，必须得让其他的文件能够知道它是切面，可以在配置类中加上

@EnableAspectJAutoProxy,并且声明切面类组件，代码如下:

```
@Configuration
@EnableAspectJAutoProxy //启动Aspect代理
@ComponentScan

public class BeanConfig {
  @Bean
  public AspectClass getAspectClass() {
      return new AspectClass();
  }
}
```

如果要使用XML来实现的话，则:

```
<aop:aspectj-autoproxy/>
<bean class="包名.AspectClass"></bean>
```

环绕通知

环绕通知是最为强大的通知类型，它能够让你所编写的逻辑被通知的目标方法完全包装起来，实际上就像在一个通知方法中编写前置通知和后置通知。

```
public class Audience{
    @Arount("execution(** com_7_4_3.TrackCounter.Count(int))")
    public void watchPerformance(ProceedingJoinPOint jp){
        try{
            System.out.println("前置通知");
            jp.proceed();
            System.out.println("后置通知");
        } catch(Throwable e){
            System.out.println("Demading a refund");
        }
    }
}
```

- 处理通知中的参数

  通过切点函数中的参数向连接点的函数传参数

  ```
  @Pointcut("execution(** com_7_4_3.TrackCounter.Count(int))"+"&& args(trackNUmber)")
  public void trackPlayed(int trackNumber){}

  @Beafore("trackPlayed(int trackNumber)")
  public void count(int trackNumber){
      System.out.println(trackNUmber);
  }
  ```

  通过XML的方式来实现该功能

  ```
  //通过XML来实现自定义切口函数，aop直接能够让我们在Spring中声明切面，不用再去启动代理操做
  <bean id="audience" class="com_8_4_4.Audience"></bean>
      <!-- 通过XML配置切面以及向切点向连接点传入参数 -->
      <aop:config>
          <aop:aspect ref="audience">
              <aop:pointcut id="performance"
                  expression="execution(** com_8_4_4.Performance.perform(int)) and
  args(count)"></aop:pointcut>      //自定义切点函数名称
              <aop:before method="silenceCePhone" pointcut-ref="performance" />
          </aop:aspect>
      </aop:config>
  ```