[今日课程大纲]

动态代理设计模式(JDK 和 cglib)

AOP 详解

AOP 中几种通知类型

AOP 两种实现方式(Schema-base 和 AspectJ)

[知识点详解]

一.AOP

1.AOP:中文名称面向切面编程

2.英文名称:(Aspect Oriented Programming)

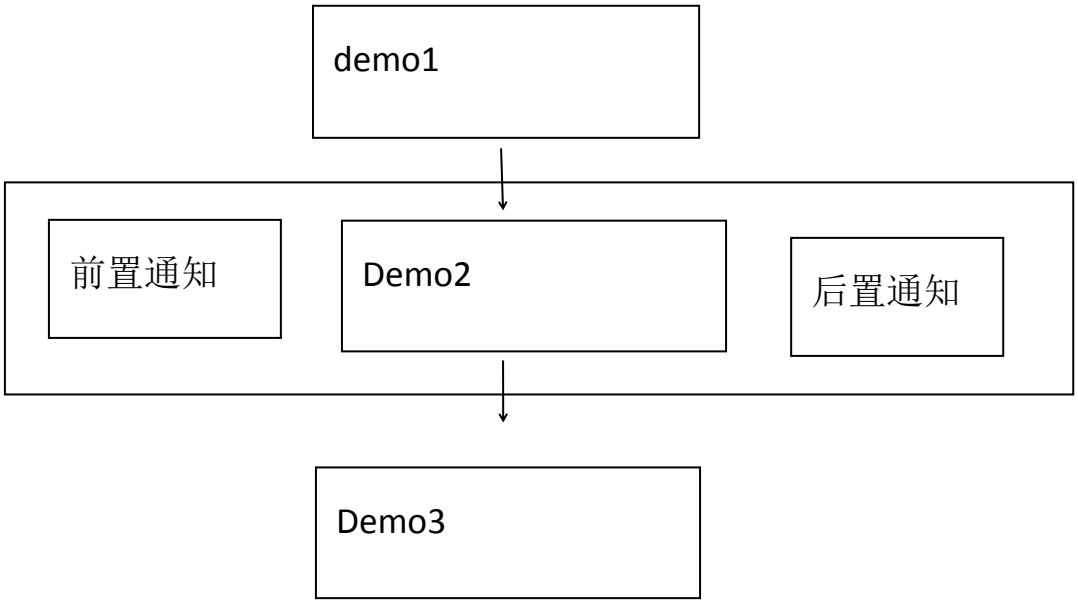
3.正常程序执行流程都是纵向执行流程

3.1 又叫面向切面编程,在原有纵向执行流程中添加横切面

3.2 不需要修改原有程序代码

3.2.1 高扩展性

3.2.2 原有功能相当于释放了部分逻辑.让职责更加明确.



4.面向切面编程是什么?

4.1 在程序原有纵向执行流程中,针对某一个或某一些方法添加通知,形成横切面过程就叫做面向切面编程.

5.常用概念

5.1 原有功能: 切点, pointcut

5.2 前置通知: 在切点之前执行的功能. before advice

5.3 后置通知: 在切点之后执行的功能,after advice

5.4 如果切点执行过程中出现异常,会触发异常通知.throws advice

5.5 所有功能总称叫做切面.

5.6 织入: 把切面嵌入到原有功能的过程叫做织入

6.spring 提供了 2 种 AOP 实现方式

6.1 Schema-based

6.1.1 每个通知都需要实现接口或类

6.1.2 配置 spring 配置文件时在<aop:config>配置

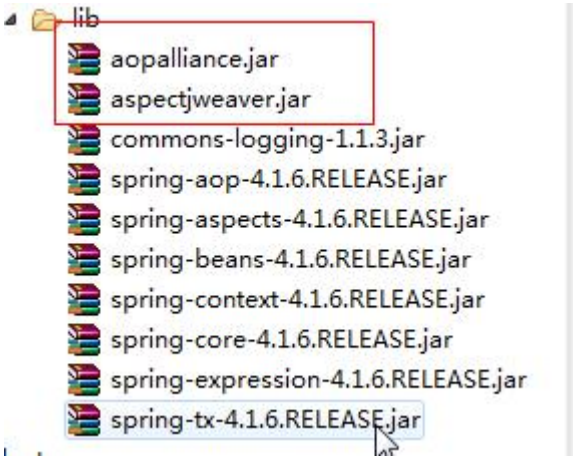
6.2 AspectJ

6.2.1 每个通知不需要实现接口或类

6.2.2 配置 spring 配置文件是在<aop:config>的子标签<aop:aspect>中配置

二. Schema-based 实现步骤

1. 导入 jar



2. 新建通知类

2.1 新建前置通知类

2.1.1 arg0: 切点方法对象 Method 对象

2.1.2 arg1: 切点方法参数

2.1.3 arg2:切点在哪个对象中

```
public class MyBeforeAdvice implements
MethodBeforeAdvice {
    @Override
    public void before(Method arg0, Object[] arg1, Object
```

```
arg2) throws Throwable {
        System.out.println("执行前置通知");
    }
}
```

2.2 新建后置通知类

2.2.1 arg0: 切点方法返回值

2.2.2 arg1:切点方法对象

2.2.3 arg2:切点方法参数

2.2.4 arg3:切点方法所在类的对象

```
public class MyAfterAdvice implements
AfterReturningAdvice {
    @Override
    public void afterReturning(Object arg0, Method arg1,
Object[] arg2, Object arg3) throws Throwable {
        System.out.println("执行后置通知");
    }
}
```

3. 配置 spring 配置文件

3.1 引入 aop 命名空间

3.2 配置通知类的<bean>

3.3 配置切面

3.4 * 通配符,匹配任意方法名,任意类名,任意一级包名

3.5 如果希望匹配任意方法参数 (..)

```
<?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:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/sc
hema/beans

http://www.springframework.org/schema/beans/spring-be
ans.xsd
        http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.
xsd">
    <!-- 配置通知类对象,在切面中引入 -->
    <bean id="mybefore"
class="com.bjsxt.advice.MyBeforeAdvice"></bean>
```

```
    <bean id="myafter"
class="com.bjsxt.advice.MyAfterAdvice"></bean>


    <!-- 配置切面 -->
    <aop:config>
      <!-- 配置切点 -->
      <aop:pointcut expression="execution(*
com.bjsxt.test.Demo.demo2())" id="mypoint"/>
      <!-- 通知 -->
      <aop:advisor advice-ref="mybefore"
pointcut-ref="mypoint"/>
      <aop:advisor advice-ref="myafter"
pointcut-ref="mypoint"/>
    </aop:config>
    <!-- 配置 Demo 类,测试使用 -->
    <bean id="demo" class="com.bjsxt.test.Demo"></bean>
</beans>
```

4. 编写测试代码

```
public class Test {
   public static void main(String[] args) {
//    Demo demo = new Demo();
//    demo.demo1();
```

```
//    demo.demo2();
//    demo.demo3();
      ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xm
l");
      Demo demo = ac.getBean("demo",Demo.class);
      demo.demo1();
      demo.demo2();
      demo.demo3();
   }
}
```

5. 运行结果:

三. 配置异常通知的步骤(AspectJ 方式)

1. 只有当切点报异常才能触发异常通知
2. 在 spring 中有 AspectJ 方式提供了异常通知的办法.

   2.1 如果希望通过 schema-base 实现需要按照特定的要求自己编写方法.

3. 实现步骤:

3.1 新建类,在类写任意名称的方法

```
public class MyThrowAdvice{
   public void myexception(Exception e1){
      System.out.println("执行异常通知
"+e1.getMessage());
   }
}
```

3.2 在 spring 配置文件中配置

3.2.1 <aop:aspect>的 ref 属性表示:方法在哪个类中.

3.2.2 <aop: xxxx/> 表示什么通知

3.2.3 method: 当触发这个通知时,调用哪个方法

3.2.4 throwing: 异常对象名,必须和通知中方法参数名相同(可以不在通知中声明异常对象)

```
    <bean id="mythrow"
class="com.bjsxt.advice.MyThrowAdvice"></bean>
    <aop:config>
      <aop:aspect ref="mythrow">
         <aop:pointcut expression="execution(*
com.bjsxt.test.Demo.demo1())" id="mypoint"/>
         <aop:after-throwing method="myexception"
pointcut-ref="mypoint" throwing="e1"/>
```

```
    </aop:aspect>
  </aop:config>
  <bean id="demo" class="com.bjsxt.test.Demo"></bean>
```

四. 异常通知(Schema-based 方式)

1. 新建一个类实现 throwsAdvice 接口

   1.1 必须自己写方法,且必须叫 afterThrowing

   1.2 有两种参数方式

      1.2.1 必须是 1 个或 4 个

   1.3 异常类型要与切点报的异常类型一致

```
public class MyThrow  implements ThrowsAdvice{
// public void afterThrowing(Method m, Object[] args,
Object target, Exception ex) {
//    System.out.println("执行异常通知");
// }
   public void afterThrowing(Exception ex) throws
Throwable {
      System.out.println("执行异常通过-schema-base 方式
");
    }
}
```

2. 在 ApplicationContext.xml 配置

```
    <bean id="mythrow"
class="com.bjsxt.advice.MyThrow"></bean>
    <aop:config>
      <aop:pointcut expression="execution(*
com.bjsxt.test.Demo.demo1())" id="mypoint"/>
      <aop:advisor advice-ref="mythrow"
pointcut-ref="mypoint" />
    </aop:config>
    <bean id="demo" class="com.bjsxt.test.Demo"></bean>
```

五.环绕通知(Schema-based 方式)

1. 把前置通知和后置通知都写到一个通知中,组成了环绕通知

2. 实现步骤

   2.1 新建一个类实现 MethodInterceptor

```
public class MyArround implements MethodInterceptor {
   @Override
   public Object invoke(MethodInvocation arg0) throws
Throwable {
      System.out.println("环绕-前置");
      Object result = arg0.proceed();//放行,调用切点方式
```

```
        System.out.println("环绕-后置");
        return result;
    }
}
```

2.2 配置 applicationContext.xml

```
 <bean id="myarround"
class="com.bjsxt.advice.MyArround"></bean>
    <aop:config>
        <aop:pointcut expression="execution(*
com.bjsxt.test.Demo.demo1())" id="mypoint"/>
        <aop:advisor advice-ref="myarround"
pointcut-ref="mypoint" />
    </aop:config>
    <bean id="demo" class="com.bjsxt.test.Demo"></bean>
```

六.使用 AspectJ 方式实现

1. 新建类,不用实现

1.1 类中方法名任意

```
public class MyAdvice {
    public void mybefore(String name1,int age1){
```

```
        System.out.println("前置"+name1 );
    }
    public void mybefore1(String name1){
        System.out.println("前置:"+name1);
    }
    public void myaftering(){
        System.out.println("后置 2");
    }
    public void myafter(){
        System.out.println("后置 1");
    }
    public void mythrow(){
        System.out.println("异常");
    }
    public Object myarround(ProceedingJoinPoint p) throws
Throwable{
        System.out.println("执行环绕");
        System.out.println("环绕-前置");
        Object result = p.proceed();
        System.out.println("环绕后置");
        return result;
    }
```

```
}
```

1.2 配置 spring 配置文件

1.2.1 <aop:after/> 后置通知,是否出现异常都执行

1.2.2 <aop:after-returing/> 后置通知,只有当切点正确执行时执行

1.2.3 <aop:after/> 和 <aop:after-returing/> 和 <aop:after-throwing/>执行顺序和配置顺序有关

1.2.4 execution() 括号不能扩上 args

1.2.5 中间使用 and 不能使用&& 由 spring 把 and 解析成&&

1.2.6 args(名称) 名称自定义的.顺序和 demo1(参数,参数)对应

1.2.7 <aop:before/> arg-names=”名称” 名称来源于 expression=”” 中 args(),名称必须一样

1.2.7.1 args() 有几个参数,arg-names 里面必须有几个参数

1.2.7.2 arg-names=”” 里面名称必须和通知方法参数名对应

```
    <aop:config>
      <aop:aspect ref="myadvice">
         <aop:pointcut expression="execution(*
com.bjsxt.test.Demo.demo1(String,int)) and
args(name1,age1)" id="mypoint"/>
         <aop:pointcut expression="execution(*
com.bjsxt.test.Demo.demo1(String)) and args(name1)"
id="mypoint1"/>
```

```
        <aop:before method="mybefore"
pointcut-ref="mypoint" arg-names="name1,age1"/>
        <aop:before method="mybefore1"
pointcut-ref="mypoint1" arg-names="name1"/>
        <!-- <aop:after method="myafter"
pointcut-ref="mypoint"/>
        <aop:after-returning method="myaftering"
pointcut-ref="mypoint"/>
        <aop:after-throwing method="mythrow"
pointcut-ref="mypoint"/>
        <aop:around method="myarround"
pointcut-ref="mypoint"/>-->
      </aop:aspect>
    </aop:config>
```

七. 使用注解(基于 Aspect)

1. spring 不会自动去寻找注解,必须告诉 spring 哪些包下的类中可能有注解

1.1 引入 xmlns:context

```
<context:component-scan
base-package="com.bjsxt.advice"></context:component-s
```

```
can>
```

2. @Component

2.1 相当于<bean/>

2.2 如果没有参数,把类名首字母变小写,相当于<bean id=””/>

2.3 @Component(“自定义名称”)

3. 实现步骤:

3.1 在 spring 配置文件中设置注解在哪些包中

```
<context:component-scan
base-package="com.bjsxt.advice,com.bjsxt.test"></cont
ext:component-scan>
```

3.2 在 Demo 类中添加@Componet

3.2.1 在方法上添加@Pointcut(“”) 定义切点

```
@Component
public class Demo {
   @Pointcut("execution(*
com.bjsxt.test.Demo.demo1())")
   public void demo1() throws Exception{
//    int i = 5/0;
      System.out.println("demo1");
   }
}
```

3.3 在通知类中配置

3.3.1 @Component 类被 spring 管理

3.3.2 @Aspect 相当于<aop:aspect/>表示通知方法在当前类中

```
@Component
@Aspect
public class MyAdvice {
   @Before("com.bjsxt.test.Demo.demo1()")
   public void mybefore(){
      System.out.println("前置");
   }
   @After("com.bjsxt.test.Demo.demo1()")
   public void myafter(){
      System.out.println("后置通知");
   }
   @AfterThrowing("com.bjsxt.test.Demo.demo1()")
   public void mythrow(){
      System.out.println("异常通知");
   }
   @Around("com.bjsxt.test.Demo.demo1()")
   public Object myarround(ProceedingJoinPoint p) throws
Throwable{
      System.out.println("环绕-前置");
```

```
        Object result = p.proceed();
        System.out.println("环绕-后置");
        return result;
    }
}
```

八.代理设计模式

1. 设计模式:前人总结的一套解决特定问题的代码.
2. 代理设计模式优点:
    2.1 保护真实对象
    2.2 让真实对象职责更明确.
    2.3 扩展
3. 代理设计模式
    3.1 真实对象.(老总)
    3.2 代理对象(秘书)
    3.3 抽象对象(抽象功能),谈小目标

九. 静态代理设计模式

1. 由代理对象代理所有真实对象的功能.
    1.1 自己编写代理类

1.2 每个代理的功能需要单独编写

2. 静态代理设计模式的缺点:

2.1 当代理功能比较多时,代理类中方法需要写很多.

十. 动态代理

1. 为了解决静态代理频繁编写代理功能缺点.

2. 分类:

2.1 JDK 提供的

2.2 cglib 动态代理

十一. JDK 动态代理

1. 和 cglib 动态代理对比

1.1 优点:jdk 自带,不需要额外导入 jar

1.2 缺点:

1.2.1 真实对象必须实现接口

1.2.2 利用反射机制.效率不高.

2. 使用 JDK 动态代理时可能出现下面异常

2.1 出现原因:希望把接口对象转换为具体真实对象

```
true
Exception in thread "main" java.lang.ClassCastException: com.sun.proxy.$Proxy0 cannot be cast to com.bjsxt.Laozong
        at com.bjsxt.Women.main(Women.java:14)
```

十二: cglib 动态代理

1. cglib 优点:
    1.1 基于字节码,生成真实对象的子类.
        1.1.1 运行效率高于 JDK 动态代理.
    1.2 不需要实现接口
2. cglib 缺点:
    2.1 非 JDK 功能,需要额外导入 jar
3. 使用 spring aop 时,只要出现 Proxy 和真实对象转换异常
    3.1 设置为 true 使用 cglib
    3.2 设置为 false 使用 jdk(默认值)

```
<aop:aspectj-autoproxy
proxy-target-class="true"></aop:aspectj-autoproxy>
```