
黑马程序员™
www.itheima.com
传智播客旗下
高端IT教育品牌
面向切面编程AOP

目录 Contents

1. Spring 的 AOP 简介

1.1 什么是 AOP

AOP 为 **A**spect **O**riented **P**rogramming 的缩写，意思为面向切面编程，是通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。

AOP 是 OOP 的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

1. Spring 的 AOP 简介

1.2 AOP 的作用及其优势

- 作用：在程序运行期间，在不修改源码的情况下对方法进行功能增强
- 优势：减少重复代码，提高开发效率，并且便于维护

1. Spring 的 AOP 简介

1.3 AOP 的底层实现

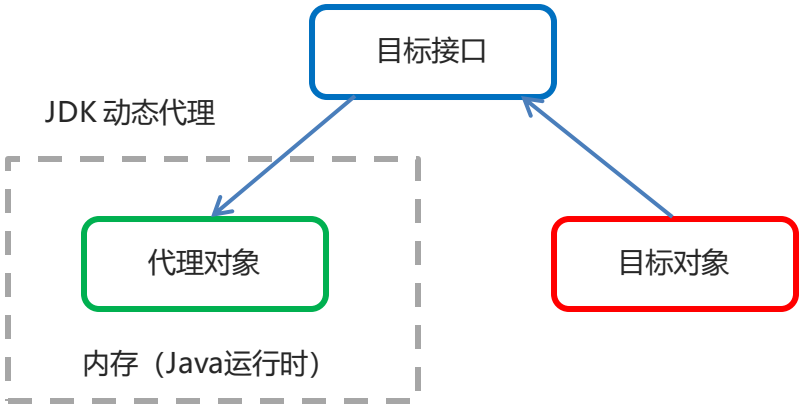
实际上，AOP 的底层是通过 Spring 提供的的动态代理技术实现的。在运行期间，Spring通过动态代理技术动态的生成代理对象，代理对象方法执行时进行增强功能的介入，在去调用目标对象的方法，从而完成功能的增强。

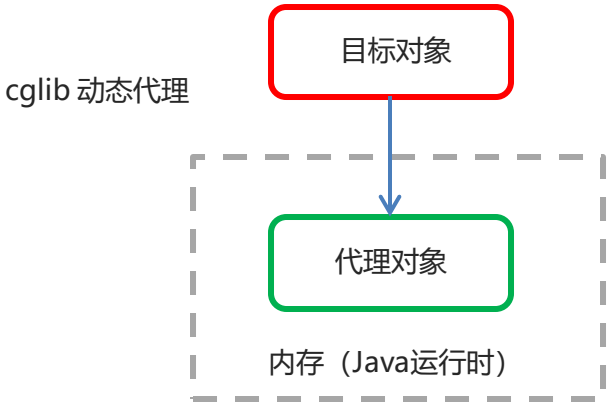
1. Spring 的 AOP 简介

1.4 AOP 的动态代理技术

常用的动态代理技术

- JDK 代理 : 基于接口的动态代理技术
- cglib 代理：基于父类的动态代理技术





1. Spring 的 AOP 简介

1.5 JDK 的动态代理

① 目标类接口

```
public interface TargetInterface {
    public void method();
}
```

② 目标类

```
public class Target implements TargetInterface {
    @Override
    public void method() {
        System.out.println("Target running....");
    }
}
```

1. Spring 的 AOP 简介

1.5 JDK 的动态代理

③ 动态代理代码

```
Target target = new Target(); //创建目标对象
//创建代理对象
TargetInterface proxy = (TargetInterface) Proxy.newProxyInstance(target.getClass()
.getClassLoader(),target.getClass().getInterfaces(),new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
                System.out.println("前置增强代码...");
                Object invoke = method.invoke(target, args);
                System.out.println("后置增强代码...");
                return invoke;
            }
        }
);
```

1. Spring 的 AOP 简介

1.5 JDK 的动态代理

④ 调用代理对象的方法测试

```
// 测试,当调用接口的任何方法时，代理对象的代码都无序修改
proxy.method();
```

```
"C:\Program Files\Java\jdk1.8.0_162\bin\java" ...
前置增强代码...
Target running....
后置增强代码...

Process finished with exit code 0
```

1. Spring 的 AOP 简介

1.6 cglib 的动态代理

① 目标类

```
public class Target {
    public void method() {
        System.out.println("Target running....");
    }
}
```

1. Spring 的 AOP 简介

1.6 cglib 的动态代理

② 动态代理代码

```
Target target = new Target(); //创建目标对象
Enhancer enhancer = new Enhancer();    //创建增强器
enhancer.setSuperclass(Target.class);  //设置父类
enhancer.setCallback(new MethodInterceptor() { //设置回调
    @Override
    public Object intercept(Object o, Method method, Object[] objects,
    MethodProxy methodProxy) throws Throwable {
        System.out.println("前置代码增强....");
        Object invoke = method.invoke(target, objects);
        System.out.println("后置代码增强....");
        return invoke;
    }
});
Target proxy = (Target) enhancer.create(); //创建代理对象
```

1. Spring 的 AOP 简介

1.6 cglib 的动态代理

③ 调用代理对象的方法测试

```
//测试,当调用接口的任何方法时，代理对象的代码都无序修改
proxy.method();
```

```
"C:\Program Files\Java\jdk1.8.0_162\bin\java" ...
前置增强代码...
Target running....
后置增强代码...

Process finished with exit code 0
```

1. Spring 的 AOP 简介

1.7 AOP 相关概念

Spring 的 AOP 实现底层就是对上面的动态代理的代码进行了封装，封装后我们只需要对需要关注的部分进行代码编写，并通过配置的方式完成指定目标的方法增强。

在正式讲解 AOP 的操作之前，我们必须理解 AOP 的相关术语，常用的术语如下：

- Target（目标对象）：代理的目标对象
- Proxy （代理）：一个类被 AOP 织入增强后，就产生一个结果代理类
- Joinpoint（连接点）：所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法，因为spring只支持方法类型的连接点
- Pointcut（切入点）：所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义
- Advice（通知/ 增强）：所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知
- Aspect（切面）：是切入点和通知（引介）的结合
- Weaving（织入）：是指把增强应用到目标对象来创建新的代理对象的过程。spring采用动态代理织入，而AspectJ采用编译期织入和类装载期织入

1.8 AOP 开发明确的事项

1. 需要编写的内容

- 编写核心业务代码（目标类的目标方法）
- 编写切面类，切面类中有通知(增强功能方法)
- 在配置文件中，配置织入关系，即将哪些通知与哪些连接点进行结合

2. AOP 技术实现的内容

Spring 框架监控切入点方法的执行。一旦监控到切入点方法被运行，使用代理机制，动态创建目标对象的代理对象，根据通知类别，在代理对象的对应位置，将通知对应的功能织入，完成完整的代码逻辑运行。

3. AOP 底层使用哪种代理方式

在 spring 中，框架会根据目标类是否实现了接口来决定采用哪种动态代理的方式。

1. Spring 的 AOP 简介

1.9 知识要点

- aop：面向切面编程
- aop底层实现：基于JDK的动态代理 和 基于Cglib的动态代理
- aop的重点概念：

  Pointcut（切入点）：被增强的方法

  Advice（通知/ 增强）：封装增强业务逻辑的方法

  Aspect（切面）：切点+通知

  Weaving（织入）：将切点与通知结合的过程
- 开发明确事项：

  谁是切点（切点表达式配置）

  谁是通知（切面类中的增强方法）

  将切点和通知进行织入配置

目录 Contents

2. 基于 XML 的 AOP 开发

2.1 快速入门

① 导入 AOP 相关坐标
② 创建目标接口和目标类（内部有切点）
③ 创建切面类（内部有增强方法）
④ 将目标类和切面类的对象创建权交给 spring
⑤ 在 applicationContext.xml 中配置织入关系
⑥ 测试代码

2. 基于 XML 的 AOP 开发

2.1 快速入门

① 导入 AOP 相关坐标

```
<!--导入spring的context坐标，context依赖aop-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.0.5.RELEASE</version>
</dependency>
<!-- aspectj的织入 -->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.13</version>
</dependency>
```

2. 基于 XML 的 AOP 开发

2.1 快速入门

② 创建目标接口和目标类（内部有切点）

```
public interface TargetInterface {
    public void method();
}
```

```
public class Target implements TargetInterface {
    @Override
    public void method() {
        System.out.println("Target running....");
    }
}
```

2. 基于 XML 的 AOP 开发

2.1 快速入门

③ 创建切面类（内部有增强方法）

```
public class MyAspect {
    //前置增强方法
    public void before(){
        System.out.println("前置代码增强.....");
    }
}
```

2. 基于 XML 的 AOP 开发

2.1 快速入门

④ 将目标类和切面类的对象创建权交给 spring

```
<!--配置目标类-->
<bean id="target" class="com.itheima.aop.Target"></bean>
<!--配置切面类-->
<bean id="myAspect" class="com.itheima.aop.MyAspect"></bean>
```

2. 基于 XML 的 AOP 开发

2.1 快速入门

⑤ 在 applicationContext.xml 中配置织入关系

导入aop命名空间

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

2. 基于 XML 的 AOP 开发

2.1 快速入门

⑤ 在 applicationContext.xml 中配置织入关系

配置切点表达式和前置增强的织入关系

```
<aop:config>
    <!--引用myAspect的Bean为切面对象-->
    <aop:aspect ref="myAspect">
        <!--配置Target的method方法执行时要进行myAspect的before方法前置增强-->
        <aop:before method="before" pointcut="execution(public void
com.itheima.aop.Target.method())"></aop:before>
    </aop:aspect>
</aop:config>
```

2. 基于 XML 的 AOP 开发

2.1 快速入门

⑥ 测试代码

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AopTest {
    @Autowired
    private TargetInterface target;
    @Test
    public void test1(){
        target.method();
    }
}
```

2. 基于 XML 的 AOP 开发

2.1 快速入门

⑦ 测试结果

```
信息: Loading XML bean definitions from class path resource [applicationContext.xml]
十月 23, 2018 5:55:49 下午 org.springframework.context.support.AbstractApplicationCor
信息: Refreshing org.springframework.context.support.GenericApplicationContext@6f7fd
前置代码增强.....
Target running....

Process finished with exit code 0
```

2. 基于 XML 的 AOP 开发

2.2 XML 配置 AOP 详解

1. 切点表达式的写法

表达式语法：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

- 访问修饰符可以省略
- 返回值类型、包名、类名、方法名可以使用星号* 代表任意
- 包名与类名之间一个点 . 代表当前包下的类，两个点 .. 表示当前包及其子包下的类
- 参数列表可以使用两个点 .. 表示任意个数，任意类型的参数列表

例如：

```
execution(public void com.itheima.aop.Target.method())
execution(void com.itheima.aop.Target.*(..))
execution(* com.itheima.aop.*.*(..))
execution(* com.itheima.aop..*.*(..))
execution(* *..*.*(..))
```

2.2 XML 配置 AOP 详解

2. 通知的类型

通知的配置语法：

```
<aop:通知类型 method=“切面类中方法名” pointcut=“切点表达式"></aop:通知类型>
```

名称	标签	说明
前置通知	<aop:before>	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	<aop:after-returning>	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	<aop:around>	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	<aop:throwing>	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	<aop:after>	用于配置最终通知。无论增强方式执行是否有异常都会执行

2.2 XML 配置 AOP 详解

3. 切点表达式的抽取

当多个增强的切点表达式相同时，可以将切点表达式进行抽取，在增强中使用 pointcut-ref 属性代替 pointcut 属性来引用抽取后的切点表达式。

```
<aop:config>
    <!--引用myAspect的Bean为切面对象-->
    <aop:aspect ref="myAspect">
        <aop:pointcut id="myPointcut" expression="execution(* com.itheima.aop.*.*(..))"/>
        <aop:before method="before" pointcut-ref="myPointcut"></aop:before>
    </aop:aspect>
</aop:config>
```

2. 基于 XML 的 AOP 开发

2.3 知识要点

- aop织入的配置

```
<aop:config>
    <aop:aspect ref="切面类">
        <aop:before method="通知方法名称" pointcut="切点表达式"></aop:before>
    </aop:aspect>
</aop:config>
```

- 通知的类型：前置通知、后置通知、环绕通知、异常抛出通知、最终通知
- 切点表达式的写法：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

目录 Contents

3. 基于注解的 AOP 开发

3.1 快速入门

基于注解的aop开发步骤：

① 创建目标接口和目标类（内部有切点）

② 创建切面类（内部有增强方法）

③ 将目标类和切面类的对象创建权交给 spring

④ 在切面类中使用注解配置织入关系

⑤ 在配置文件中开启组件扫描和 AOP 的自动代理

⑥ 测试

3. 基于注解的 AOP 开发

3.1 快速入门

① 创建目标接口和目标类（内部有切点）

```
public interface TargetInterface {
    public void method();
}
```

```
public class Target implements TargetInterface {
    @Override
    public void method() {
        System.out.println("Target running....");
    }
}
```

3. 基于注解的 AOP 开发

3.1 快速入门

② 创建切面类（内部有增强方法）

```
public class MyAspect {
    //前置增强方法
    public void before(){
        System.out.println("前置代码增强.....");
    }
}
```

3. 基于注解的 AOP 开发

3.1 快速入门

③ 将目标类和切面类的对象创建权交给 spring

```
@Component("target")
public class Target implements TargetInterface {
    @Override
    public void method() {
        System.out.println("Target running....");
    }
}
@Component("myAspect")
public class MyAspect {
    public void before(){
        System.out.println("前置代码增强.....");
    }
}
```

3. 基于注解的 AOP 开发

3.1 快速入门

④ 在切面类中使用注解配置织入关系

```
@Component("myAspect")
@Aspect
public class MyAspect {
    @Before("execution(* com.itheima.aop.*.*(..))")
    public void before(){
        System.out.println("前置代码增强.....");
    }
}
```

3. 基于注解的 AOP 开发

3.1 快速入门

⑤ 在配置文件中开启组件扫描和 AOP 的自动代理

```
<!--组件扫描-->
<context:component-scan base-package="com.itheima.aop"/>

<!--aop的自动代理-->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

3. 基于注解的 AOP 开发

3.1 快速入门

⑥ 测试代码

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AopTest {
    @Autowired
    private TargetInterface target;
    @Test
    public void test1(){
        target.method();
    }
}
```

3. 基于注解的 AOP 开发

3.1 快速入门

⑦ 测试结果

```
信息: Closing org.springframework.context.support.GenericApplicationContext@6f7fd0e6
前置代码增强.....
Target running....

Process finished with exit code 0
```

3.2 注解配置 AOP 详解

1. 注解通知的类型

通知的配置语法：@通知注解(“切点表达式")

名称	注解	说明
前置通知	@Before	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	@AfterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	@Around	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	@AfterThrowing	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	@After	用于配置最终通知。无论增强方式执行是否有异常都会执行

3.2 注解配置 AOP 详解

2. 切点表达式的抽取

同 xml 配置 aop 一样，我们可以将切点表达式抽取。抽取方式是在切面内定义方法，在该方法上使用@Pointcut 注解定义切点表达式，然后在在增强注解中进行引用。具体如下：

```
@@Component("myAspect")
@Aspect
public class MyAspect {
    @Before("MyAspect.myPoint()")
    public void before(){
        System.out.println("前置代码增强.....");
    }
    @Pointcut("execution(* com.itheima.aop.*.*(..))")
    public void myPoint(){}
}
```

3. 基于注解的 AOP 开发

3.3 知识要点

- 注解aop开发步骤
  - ① 使用@Aspect标注切面类
  - ② 使用@通知注解标注通知方法
  - ③ 在配置文件中配置aop自动代理<aop:aspectj-autoproxy/>
- 通知注解类型

前置通知	@Before	用于配置前置通知。指定增强的方法在切入点方法之前执行
后置通知	@AfterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
环绕通知	@Around	用于配置环绕通知。指定增强的方法在切入点方法之前和之后都执行
异常抛出通知	@AfterThrowing	用于配置异常抛出通知。指定增强的方法在出现异常时执行
最终通知	@After	用于配置最终通知。无论增强方式执行是否有异常都会执行

传智播客旗下高端IT教育品牌