第3天: Spring

一、目标

1. 理解代理设计模式
2. 掌握动态代理的两种实现方式
3. 完成动态代理案例
4. 理解AOP相关概念和术语
5. 理解AspectJ表达式语言
6. 编写AOP的通知代码
7. 掌握AOP的注解实现

七、代理模式

- 为目标对象提供一个代理对象以便控制外部对目标对象的访问

2.1 静态代理

- 代理类是程序运行前准备好的代理方式被成为 静态代理
- interface: Star.java

```
package com.itheima.proxy;

/**
 * 明星接口.
 *
 * @author : Jason.lee
 */
public interface Star {
    /**
     * 唱歌.
     *
     * @param money the money
     */
    void sing(Integer money);

    /**
     * 拍戏.
     *
     * @param money the money
     */
    void act(Integer money);
}
```

- target: LiuStar.java

```
package com.itheima.proxy;
```

```
 * 刘德华.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class LiuStar implements Star {
    @Override
    public void sing(Integer money) {
        System.out.println("刘德华唱歌真好听，收费: "+money);
    }

    @Override
    public void act(Integer money) {
        System.out.println("刘德华拍戏棒棒哒，收费: "+money);
    }
}
```

- proxy: LiuStarProxy.java

```
package com.itheima.proxy;

/**
 * 刘德华代理人(经纪人).
 *  静态代理:
 *      特点:
 *          1. 需要实现与目标对象相同的接口
 *          2. 需要在内部维护目标对象
 *      缺点:
 *          1. 接口方法较多时，代理实现较麻烦 (可能只需要代理个别方法)
 *          2. 接口扩展方法后，目标对象与代理对象都需要更改
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class LiuStarProxy implements Star {

    Star star = new LiuStar();

    @Override
    public void sing(Integer money) {
        if (money > 10000) {
            star.sing(money);
        } else {
            System.out.println("档期忙!");
        }
    }

    @Override
    public void act(Integer money) {
        if (money > 20000) {
            star.act(money);
        } else {
            System.out.println("档期忙!");
        }
    }
```

- 单元测试

```
package com.itheima.proxy;

import org.junit.Test;

public class LiuStarProxyTest {

    @Test
    public void sing() {
        new LiuStarProxy().sing(10001);
        new LiuStarProxy().act(10001);
    }
}
```

- 静态代理的缺点
  - 接口方法较多时, 代理实现较麻烦 (可能只需要代理个别方法)
  - 接口扩展方法后, 目标对象与代理对象都需要更改

2.2 动态代理

- 代理类在程序运行时创建的代理方式被称为 动态代理

2.2.1 Proxy动态代理

- JDK提供的动态代理API

```
@Test
public void testProxy() {
    // 目标对象
    LiuStar ls = new LiuStar();

    /**
      * 动态创建代理对象:
      *  Proxy: JDK动态代理的Api也是生成的代理对象的超类
      *      loader: 类加载器
      *      interfaces: 代理对象需要实现的接口组
      *      h:  代理对象方法调用的处理程序
      */
    Star star = (Star) Proxy.newProxyInstance(
        // 默认类加载器
        this.getClass().getClassLoader(),
        // 实现的接口组
        new Class[]{Star.class},
        // 调用处理程序
        new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                // 获取方法名
                String name = method.getName();
                // 获取参数值
```

```
                    || (name.equals("act") && arg > 20000)) {
                    // 调用目标方法
                    return method.invoke(ls, args);
                }
                return null;
            }
        }
    );

    // 代理类继承了Proxy
    System.out.println(star instanceof Proxy);
    // 代理类实现了Star
    System.out.println(star instanceof Star);
    // 查看类字节码
    System.out.println(star.getClass()); // class com.sun.proxy.$Proxy4

    // 调用代理方法
    star.sing(10001);
    star.act(10001);

}
```

- JDK动态代理也称之为 **接口代理**：代理类实现了与目标对象相同的接口 ($Proxy4 implement Star)

2.2.2 Cglib动态代理

- 第三方框架实现的动态代理

```
@Test
public void testCglib() {

    // 目标对象
    LiuStar ls = new LiuStar();

    /**
      * 动态创建代理对象:
      *  Enhancer: Cglib动态代理的Api
      *      superclass: 超类字节码
      *      interfaces: 代理对象需要实现的接口组
      *      callback:  代理对象方法调用的拦截器
      */
    Star star = (Star) Enhancer.create(
        LiuStar.class,
        new Class[]{ Star.class },
        new MethodInterceptor() {
            @Override
            public Object intercept(Object o, Method method, Object[]
objects, MethodProxy methodProxy) throws Throwable {
                // 获取方法名
                String name = method.getName();
                // 获取参数值
                Integer arg = (Integer) objects[0];
                // 拦截方法
```

```
                    return method.invoke(ls, objects);
                }
                return null;
            }
        }
    );

    // 代理类继承了LiuStar
    System.out.println(star instanceof LiuStar);
    // 代理类实现了Star
    System.out.println(star instanceof Star);
    // 查看类字节码
    System.out.println(star.getClass()); // class
com.itheima.proxy.LiuStar$$EnhancerByCGLIB$$13a93339
    // 调用代理方法
    star.sing(10001);
    star.act(10001);
}
```

- Cglib动态代理也称之为 **子类代理**：代理类继承了目标对象 ($$EnhancerByCGLIB extends LiuStar)

2.2.3 动态代理对比

	JDK	CGLIB
实现原理	利用 **实现** 接口方法来拦截目标方法	利用 **继承** 覆写方法来拦截目标方法
实现前提	**需要目标对象实现接口**	需要目标方法未使用final/static修饰
实现效率	JDK1.6以前较Cglib慢; 1.6以及1.7大量调用时较慢	1.8时被JDK代理超越

二、AOP概念与术语

3.1 AOP概念

- AOP (Aspect Oriented Programming) 即面向切面编程
- 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

3.2 AOP的作用

- 非侵入式编程: 在不修改源码的情况下对已有方法进行**增强**
- 提高代码复用: 增强的内容抽象成方法或者对象可**重复使用**
- 统一管理维护: 抽象成**独立**的方法或对象方便后期维护管理

3.3 AOP的原理

- Spring AOP 实现的原理是动态代理技术
- 底层支持两种动态代理
  - 当目标实现接口时采用JDK动态代理
  - 当目标没有实现接口采用Cglib动态代理 (可配置统一使用Cglib)

3.4 AOP相关术语

- 连接点（Joinpoint）

> 程序执行的某个特定位置：如类开始初始化前、类初始化后、类某个方法调用前、调用后、方法抛出异常后。一个类或一段程序代码拥有一些具有边界性质的特定点，这些点中的特定点就称为“连接点”。Spring仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时以及方法调用前后这些程序执行点织入增强。连接点由两个信息确定：第一是用方法表示的程序执行点；第二是用相对点表示的方位。

- 切点（Pointcut）

> 每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点，即连接点是程序类中客观存在的事物。AOP通过“切点”定位特定的连接点。连接点相当于数据库中的记录，而切点相当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。在Spring中，切点通过org.springframework.aop.Pointcut接口进行描述，它使用类和方法作为连接点的查询条件，Spring AOP的规则解析引擎负责切点所设定的查询条件，找到对应的连接点。其实确切地说，不能称之为查询连接点，因为连接点是方法执行前、执行后等包括方位信息的具体程序执行点，而切点只定位到某个方法上，所以如果希望定位到具体连接点上，还需要提供方位信息。

- 通知（Advice）

> 通知也叫增强，增强是织入到目标类连接点上的一段程序代码，在Spring中，增强除用于描述一段程序代码外，还拥有另一个和连接点相关的信息，这便是执行点的方位。结合执行点方位信息和切点信息，我们就可以找到特定的连接点。

- 目标对象（Target）

> 增强逻辑的织入目标类。如果没有AOP，目标业务类需要自己实现所有逻辑，而在AOP的帮助下，目标业务类只实现那些非横切逻辑的程序逻辑，而性能监视和事务管理等这些横切逻辑则可以使用AOP动态织入到特定的连接点上。

- 引介（Introduction）

> 引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过AOP的引介功能，我们可以动态地为该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。

> 织入是将增强添加对目标类具体连接点上的过程。AOP像一台织布机，将目标类、增强或引介通过AOP这台织布机天衣无缝地编织到一起。根据不同的实现技术，AOP有三种织入的方式：
>
> a、编译期织入，这要求使用特殊的Java编译器。
>
> b、类装载期织入，这要求使用特殊的类装载器。
>
> c、动态代理织入，在运行期为目标类添加增强生成子类的方式。
>
> 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。

- 代理（Proxy）

> 一个类被AOP织入增强后，就产出了一个结果类，它是融合了原类和增强逻辑的代理类。根据不同的代理方式，代理类既可能是和原类具有相同接口的类，也可能就是原类的子类，所以我们可以采用调用原类相同的方式调用代理类。

- 切面（Aspect）

> 切面由切点和增强（引介）组成，它既包括了横切逻辑的定义，也包括了连接点的定义，Spring AOP就是负责实施切面的框架，它将切面所定义的横切逻辑织入到切面所指定的连接点中。

三、AOP配置案例

- 使用Spring AOP实现自动记录日志的功能

4.1 基于XML的AOP配置

4.1.1 环境搭建

- 工程名称: spring-day02-xml
- pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>spring3</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-day03-xml</artifactId>

    <dependencies>
        <!-- Spring IOC 依赖以及Cglib支持 -->
        <dependency>
            <groupId>org.springframework</groupId>
```

```
        <!-- AspectJ切面表达式支持 -->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.8.13</version>
        </dependency>
        <!-- 环境测试 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

- AccountServiceImpl.java

```
package com.itheima.xml.impl;

import com.itheima.xml.AccountService;

/**
 * 账户业务实现类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class AccountServiceImpl implements AccountService {
    @Override
    public void save() {
        System.out.println("保存账户");
    }
}
```

- logAdvice.java

```
package com.itheima.xml.advice;

/**
 * 记录日志功能的通知/切面类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class LogAdvice {

    public void log(){
        System.out.println("记录操作日志..");
    }
```

- applicationContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- 创建AccountServiceImpl对象并加入到IOC容器 -->
    <bean id="accountService"
class="com.itheima.xml.impl.AccountServiceImpl"/>

    <!-- 创建并加入IOC容器 -->
    <bean id="logAdvice" class="com.itheima.xml.advice.LogAdvice"/>
</beans>
```

- XmlTests.java

```java
// 打印对象的字节码
// class com.itheima.xml.impl.AccountServiceImpl
System.out.println(accountService.getClass());
```

4.1.2 AOP配置

- applicationContext.xml

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

    <!-- 创建AccountServiceImpl对象并加入到IOC容器 -->
    <bean id="accountService"
class="com.itheima.xml.impl.AccountServiceImpl"/>


    <!-- 创建并加入IOC容器 -->
    <bean id="logAdvice" class="com.itheima.xml.advice.LogAdvice"/>

    <!--
        1. 导入aop命名空间和约束文件:
            xmlns:aop="http://www.springframework.org/schema/aop"
            http://www.springframework.org/schema/aop
            https://www.springframework.org/schema/aop/spring-aop.xsd"
        2. 声明AOP配置: <aop:config>..
        3. 配置切面: <aop:aspect..
```

```
            id: 切入点唯一标识; expression: 指定切入点表达式;
            表达式组成: [修饰符] 返回值类型 包名.类名.方法名(参数)
            execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?)
        5. 配置通知与切入点关系: <aop:after/before/after-throwing/after-returning..
            method: 通知方法名称; pointcut-ref: 切入点唯一标识;
    -->
    <aop:config>
        <!-- 切面 -->
        <aop:aspect id="logAspect" ref="logAdvice">
            <!-- 切入点 -->
            <aop:pointcut id="pt" expression="execution(public void com.itheima.xml.impl.AccountServiceImpl.save())"/>
            <!-- 通知与切入点关系 -->
            <aop:after method="log" pointcut-ref="pt"/>
        </aop:aspect>
    </aop:config>
</beans>
```

- 单元测试

```
import com.itheima.xml.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * AOP XML测试类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class XmlTests {

    @Autowired
    AccountService accountService;

    @Test
    public void testAop (){
        // 打印对象的字节码
        // class com.sun.proxy.$Proxy15
        System.out.println(accountService.getClass());
        accountService.save();
    }
}
```

4.2 切入点表达式

- 通用符号说明: 表达式中的特殊符号
  - `*`: 通配符, 表示1个或多个
  - `.`: 路径符, 表示包以及子包
  - `..`: 参数符, 表示参数可以0个可以1个也可以多个

- 指示符: AspectJ pointcut designators (PCD)

指示符	作用
execution	匹配表达式指定的所有方法执行的连接点
within	匹配指定一个或多个类型的方法
bean	匹配指定类型的方法

4.2.1 execution

> execution是方法级别的，细粒度的控制

execution(void com.itheima..AccountServiceImpl.save())	匹配方法
execution(void com.itheima..AccountServiceImpl.save(Integer))	方法参数必须为整型
execution(* *(..))	万能配置

- execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?)
- **modifiers-pattern**: 权限访问修饰符 (可填)
- **ret-type-pattern**: 返回值类型 (必填)
- **declaring-type-pattern**: 全限定类名 (可填)
- **name-pattern**: 方法名 (必填)
- **param-pattern**: 参数名 (必填)
- **throws-pattern**: 异常类型 (可填)

4.2.2 within

> within表达式也是应用于类级别,实现粗粒度的(对类中的所有方法进行)控制的，指定相关类时，需要指定具体包名。

within(com.itheima..AccountServiceImpl)	指定一个类
within(com.itheima.*)	只包含当前目录下的类
within(com.itheima..*)	包含当前目录及所有子目录下的类

4.2.3 bean

bean(*Service)	匹配一类对象
bean(*)	包含当前目录及所有子目录下的类

4.3 常用标签说明

4.3.1 <aop:config

- 声明AOP配置

4.3.2 <aop:aspect

- 配置切面
  - id: 切面名称/唯一标识
  - ref: 通知对象的引用

4.3.3 <aop:pointcut

- 配置切入点表达式
  - id: 切入点表达式名称/唯一标识
  - expression: 切入点表达式

4.3.4 <aop:before

- 配置前置通知
  - method: 指定通知方法的名称
  - pointcut: 定义切入点表达式
  - pointcut-ref: 引用切入点表达式

4.3.5 <aop:after-returning

- 配置后置通知
  - method: 指定通知方法的名称
  - pointcut: 定义切入点表达式
  - pointcut-ref: 引用切入点表达式

4.3.6 <aop:after-throwing

- 配置异常通知
  - method: 指定通知方法的名称
  - pointcut: 定义切入点表达式
  - pointcut-ref: 引用切入点表达式

4.3.7 <aop:after

- 配置最终通知
  - method: 指定通知方法的名称
  - pointcut: 定义切入点表达式
  - pointcut-ref: 引用切入点表达式

4.3.8 <aop:around

- pointcut-ref: 引用切入点表达式

4.4 通知

4.4.1 通知类型

- 伪代码

```
try{
    [前置通知]
    执行目标对象方法..
    [后置通知]
}catch(Exception e){
    [异常通知]
}finally{
    [最终通知]
}
```

- 前置通知: 在目标方法执行前执行
- 后置通知: 在目标方法正常返回后执行 (和异常通知2选1)
- 异常通知: 在目标方法发生一场后执行 (和后置通知2选1)
- 最终通知: 无论目标方法正常执行还是发生异常都会执行
- 环绕通知: 在目标方法执行的任意时间点自由选择执行 (通用)

4.4.2 通知案例

- 改造: LogAdvice.java

```
package com.itheima.xml.advice;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.Signature;

/**
 * 记录日志功能的通知/切面类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class LogAdvice {

    public void log(){
        System.out.println("记录操作日志..");
    }

    public void before(){
        System.out.println("【前置通知】..");
    }

    public void afterReturning(){
        System.out.println("【后置通知】..");
    }

```

```
    }

    public void after(){
        System.out.println("【最终通知】..");
    }

    /**
     * 环绕通知
     * @param pjp 正在执行的连接点对象 (目标方法)
     * @return 目标方法执行后的返回值
     * @throws Throwable 目标方法执行发生的异常
     * JoinPoint:
     *  getArgs()：返回方法参数。
     *  getThis()：返回代理对象。
     *  getTarget()：返回目标对象。
     *  getSignature()：返回正在执行的方法的描述。
     *  toString()：打印通知方法的有用说明。
     */
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        // 获取目标方法的参数
        Object[] args = pjp.getArgs();
        // 实现环绕通知
        try {
            // 前置通知
            before();
            Object ret = pjp.proceed(args);// 执行目标方法
            // 后置通知
            afterReturning();
            return ret;
        } catch (Exception e){
            // 异常通知
            afterThrowing();
            throw e;
        } finally {
            // 最终通知
            after();
        }
    }
}
```

- advice.xml

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

    <!-- 创建AccountServiceImpl对象并加入到IOC容器 -->
    <bean id="accountService"
class="com.itheima.xml.impl.AccountServiceImpl"/>
```

```
    <bean id="logAdvice" class="com.itheima.xml.advice.LogAdvice"/>

    <aop:config>
        <aop:aspect id="logAspect" ref="logAdvice">
            <aop:pointcut id="pt" expression="execution(* *())"/>
            <!-- 前置通知 -->
            <aop:before method="before" pointcut-ref="pt"/>
            <!-- 后置通知 -->
            <aop:after-returning method="afterReturning" pointcut-
ref="pt"/>
            <!-- 异常通知 -->
            <aop:after-throwing method="afterThrowing" pointcut-ref="pt"/>
            <!-- 最终通知 -->
            <aop:after method="after" pointcut-ref="pt"/>
            <!-- 环绕通知 -->
            <!--<aop:around method="around" pointcut-ref="pt"/>-->
        </aop:aspect>
    </aop:config>
</beans>
```

- XmlTests.java

```
import com.itheima.xml.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * AOP XML测试类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
@RunWith(SpringJUnit4ClassRunner.class)
//@ContextConfiguration(locations = "classpath:applicationContext.xml") //
AOP案例配置
@ContextConfiguration(locations = "classpath:advice.xml") // AOP通知案例配置
public class XmlTests {

    @Autowired
    AccountService accountService;

    @Test
    public void testAop (){
        // 打印对象的字节码
        System.out.println(accountService.getClass());
        accountService.save();
    }
}
```

5.1 使用注解改造案例

5.1.1 改造工程

- 工程名称: spring-day03-anno

5.2.2 改造通知类

- LogAdvice.java

```
package com.itheima.xml.advice;

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

/**
 * @Component: 创建对象并添加到IOC容器
 * @Aspect: 定义对象为切面/通知类
 *  value: 名称
 */
@Component
@Aspect
public class LogAdvice {

    /**
     * @Pointcut: 定义切入点表达式
     */
    @Pointcut("execution(* *(..))")
    public void pt(){

    }

    /**
     * @Before: 定义前置通知和切入点的关系
     *  value: 切入点表达式或者对切入点表达式的引用
     */
    @Before("pt()")
    public void before(){
        System.out.println("【前置通知】..");
    }


    /**
     * @Before: 定义后置通知和切入点的关系
     *  value: 切入点表达式或者对切入点表达式的引用
     */
    @AfterReturning("pt()")
    public void afterReturning(){
        System.out.println("【后置通知】..");
    }


    /**
```

```
    @AfterThrowing("pt()")
    public void afterThrowing(){
        System.out.println("【异常通知】..");
    }


    /**
     * @Before: 定义最终通知和切入点的关系
     *  value: 切入点表达式或者对切入点表达式的引用
     */
    @After("pt()")
    public void after(){
        System.out.println("【最终通知】..");
    }


    /**
     * @Before: 定义环绕通知和切入点的关系
     *  value: 切入点表达式或者对切入点表达式的引用
     */
//    @Around("pt()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        // 获取目标方法的参数
        Object[] args = pjp.getArgs();
        // 实现环绕通知
        try {
            // 前置通知
            before();
            Object ret = pjp.proceed(args);// 执行目标方法
            // 后置通知
            afterReturning();
            return ret;
        } catch (Exception e){
            // 异常通知
            afterThrowing();
            throw e;
        } finally {
            // 最终通知
            after();
        }
    }
}
```

5.1.3 改造配置

- applicationContext.xml

```
<?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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

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

    <!-- 开启注解扫描 -->
    <context:component-scan base-package="com.itheima.xml"/>

    <!-- 开启自动代理
            proxy-target-class: 是否通用Cglib代理
     -->
    <aop:aspectj-autoproxy proxy-target-class="true"/>
</beans>
```

5.1.4 单元测试

- AnnoTests.java

```
import com.itheima.xml.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

/**
 * AOP XML测试类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class AnnoTests {

    @Autowired
    AccountService accountService;

    @Test
    public void testAop (){
        // 打印对象的字节码
        System.out.println(accountService.getClass());
        accountService.save();
    }
}
```