AOP

面向切面编程，扩展功能不修改源代码实现

采取横向轴机制进而取代传统纵向继承体系

aop：面向切面编程，采取横向抽取机制，取代了传统纵向继承体系重复性代码（缺点：当父类的方法名称名称发生变化时，子类调用方法也需要变）

将程序中的交叉业务逻辑（比如安全，日志，事务等）封装成一个切面，然后注入到目标对象（具体业务逻辑）中去。比如：很多方法可能会抛异常，你要记录这个异常到日志中去，可以写个拦截器类，在这个类中记录日志，在spring.xml中配置一个对这些要记录日志的方法的aop拦截器 在这个方法执行后调用这个拦截器，记录日志。

1.底层使用：动态代理方式实现（增强类中方法而不修改源代码）

- jdk动态代理：针对有接口的情况，创建接口实现类的代理对象
- cglib动态代理：没有接口情况，创建子类的代理对象

2.AOP操作术语：Joinpoint(连接点）：类里面可以被增强的方法，这些方法称为连接点

pointcut(切入点)：在类里面有很多方法可以被增强，而在实际操作中实际增强的方法称为切入点

advice(通知/增强）)：增强的逻辑或者扩展的功能，称为增强

aspect(切面)：把增强应用到具体方法上面的过程称为切面（把增强用到切入点的过程）

target(目标对象）：增强方法所在的类称为目标对象

weaving(织入）：把advice应用到target的过程

Proxy(代理）：一个类被织入增强后，产生的结果代理类

3.通知的方式：前置通知：在方法之前执行

后置通知：在方法之后执行

异常通知：方法出现异常

最终通知：在后置之后来执行

环绕通知：在方法之前和之后来执行

4.spring的aop操作

- 使用Aspectj来实现：AspectJ不是spring的一部分，只是一起使用，2.0以后增加了对他的支持
- 使用aspectj实现aop有两种方式：1.基于aspectj的xml配置方式　2.基于aspectj的注解方式

5.基于aspectj的xml配置方式

切入点的配置：常用的表达式

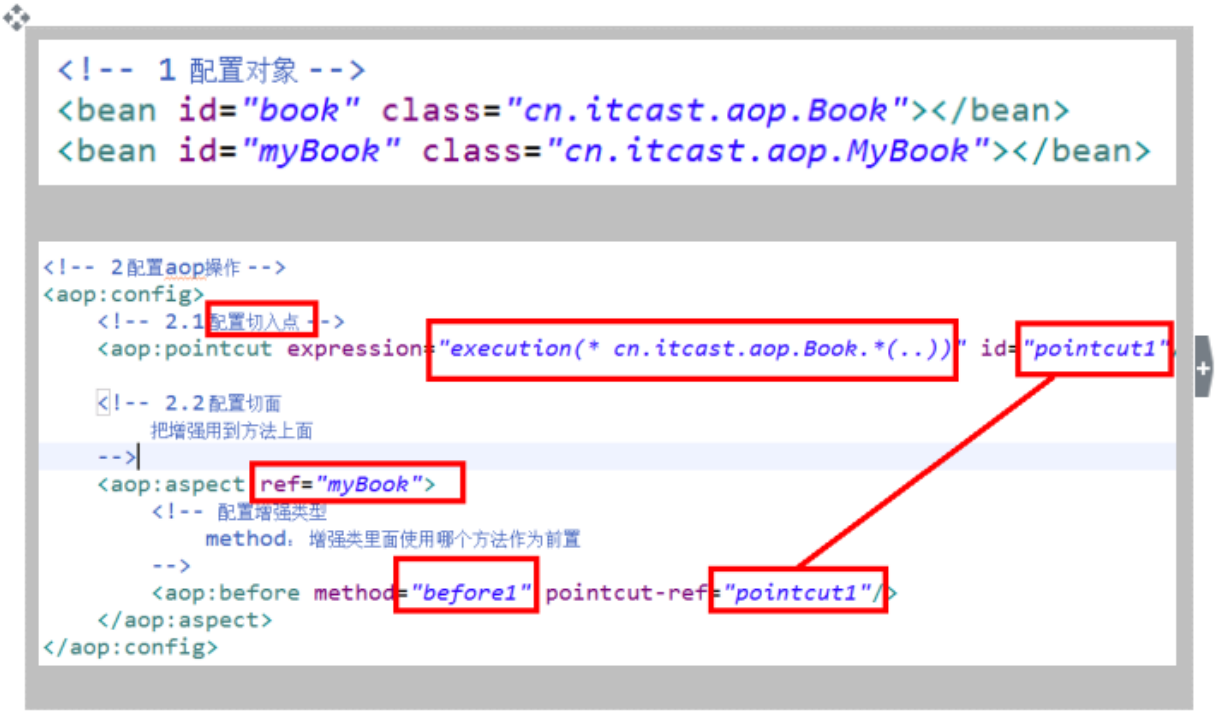
execution(<访问修饰符>?<返回类型><方法名>(<参数>)<异常>)

例如：

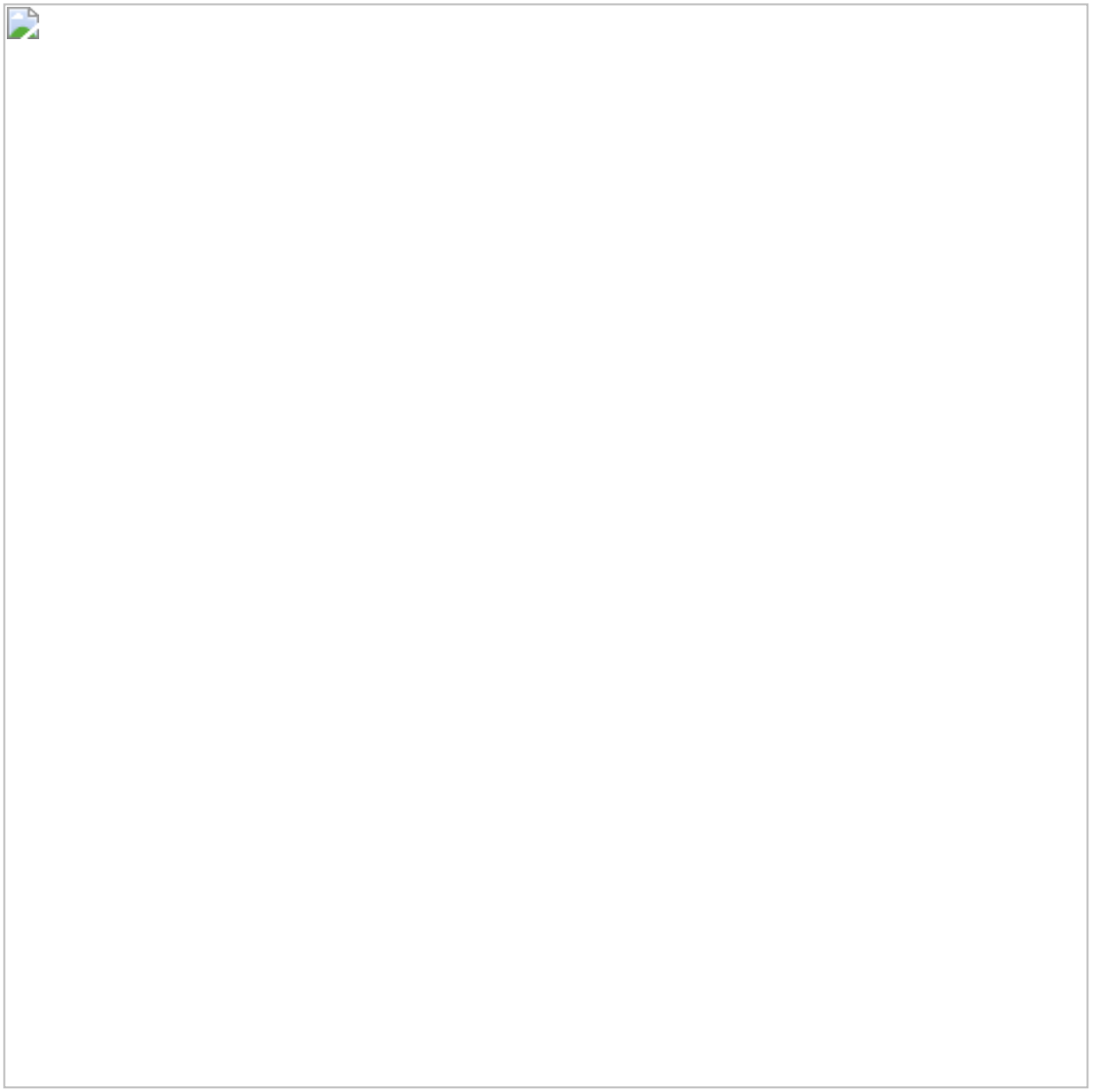
- execution(* cn.itcast.aop.Book.add(..))
- execution(* cn.itcast.aop.Book.*(..))
- execution(* *.*(..))
- 匹配所有save开头的方法 execution(* save*(..))

配置切面：把增强用到方法上面

配置增强类型：method：增强类中哪个方法作为前置、后置或者环绕



6.基于aspectj的注解方式



1 使用注解方式实现 aop 操作

第一步 创建对象

```
<!-- 创建对象 -->
<bean id="book" class="cn.itcast.aop.Book"></bean>
<bean id="myBook" class="cn.itcast.aop.MyBook"></bean>
```

第二步 在 spring 核心配置文件中，开启 aop 操作

```
<!-- 开启aop操作 -->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

第三步 在增强类上面使用注解完成 aop 操作

```
@Aspect
public class MyBook {

    //在方法上面使用注解完成增强配置
    @Before(value="execution(* cn.itcast.aop.Book.*(..))")
    public void before1() {
        System.out.println("before.............");
    }
}
```

7.两种代理方式

1）JDK动态代理：

具体实现原理：

- 通过实现InvocationHandlet接口创建自己的调用处理器
- 通过为Proxy类指定ClassLoader对象和一组interface来创建动态代理
- 通过反射机制获取动态代理类的构造函数，其唯一参数类型就是调用处理器接口类型
- 通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数参入
- JDK动态代理是面向接口的代理模式，如果被代理目标没有接口那么Spring也无能为力，
- Spring通过java的反射机制生产被代理接口的新的匿名实现类，重写了其中AOP的增强方法。

java.lang.reflect包中的两个类：Proxy和InvocationHandler。
其中InvocationHandler只是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。
而Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。

2）CGLib动态代理

CGLib是一个强大、高性能的Code生产类库，可以实现运行期动态扩展java类，Spring在运行期间通过
CGlib继承要被动态代理的类，重写父类的方法，实现AOP面向切面编程呢。

3）两者对比：

JDK动态代理是面向接口，在创建代理实现类时比CGLib要快，创建代理速度快。
CGLib动态代理：对没有生成接口的类。产生代理对象。是通过字节码底层继承要代理类来实现（如果被代理类被final关键字所修饰，那么抱歉会失败），在创建代理这一块没有JDK动态代理快，但是运行速度比JDK动态代理要快。

其他相关：

实现AOP的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。
Spring AOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice 和回调目标对象的方法所组成,并将该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但 AOP 代理中的方法与目标对象的方法存在差异，AOP 方法在特定切入点添加了增强处理，并回调了目标对象的方法。 Spring AOP使用动态代理技术在运行期织入增强代码。
使用两种代理机制：基于JDK的动态代理（JDK本身只提供接口的代理）；基于CGlib的动态代理。

动态代理（cglib 与 JDK)

- JDK的动态代理主要涉及java.lang.reflect包中的两个类：Proxy和InvocationHandler。其中InvocationHandler只是一个接口，可以通过

实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。而Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。 其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。只能针对实现接口的类生成代理，而不能针对类

- CGLib采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类的调用方法，并顺势织入横切逻辑。它运行期间生成的代理对象是目标类的扩展子类，所以无法通知final的方法，因为它们不能被覆写，是针对类实现代理，主要是为指定的类生成一个子类覆盖其中方法。
- 在spring中默认情况下使用JDK动态代理实现AOP，如果proxy-target-class设置为true或者使用了优化策略那么会使用CGLIB来创建动态代理。SpringAOP在这两种方式的实现上基本一样。以JDK代理为例，会使用JdkDynamicAopProxy来创建代理，在invoke()方法首先需要织入到当前类的增强器封装到拦截器链中，然后递归的调用这些拦截器完成功能的织入．最终返回代理对象。