

课程主题

Spring AOP源码阅读&循环依赖问题

课程目标

1. 重点掌握AOP的核心概念（画图）
2. 了解AOP的各种用法
3. 重点理解动态代理模式的原理及源码（源码、代码及图）
4. 重点理解spring中AOP产生代理对象和执行代理对象的底层原理（画图）
5. 重点阅读Spring中AOP产生代理对象和执行代理对象的底层源码
6. 循环依赖的理解

课程回顾

1. IoC流程的源码阅读
 1. 基础容器---基于XML
 2. 高级容器---基于XML和注解配置类
 3. BeanDefinition注册流程
 4. getBean流程
 5. 至少阅读三遍（首先搞清楚流程，其次搞清楚类的作用，再其次去理解设计模式的应用）
 6. 学会自己独立阅读完源码
2. 单例模式的讲解
 1. 双重检查锁（DCL）的实现方式
 1. 重点分析可能存在的问题（性能问题、安全问题）
 2. volatile
 1. 解决有序性
 2. 解决可见性
 2. 静态内部类实现方式
 3. 枚举实现方式
 4. 单例破坏
 1. 反射破坏
 2. 序列化破坏
 5. 防止单例破坏
 1. 枚举天然防止了反射破坏、序列化破坏
 2. DCL防止单例破坏

```
private Object readResolve() {  
    return instance;  
}
```

```
private DoubleCheckLockSingleton() {  
    System.out.println("====对象构建开始====");  
    synchronized (DoubleCheckLockSingleton.class) {  
        if (false == flag) {  
            flag = !flag;  
        } else {  
            throw new RuntimeException("单例模式正在被攻击");  
        }  
    }  
    System.out.println("====对象构建结束====");  
}
```

课程内容

一、AOP核心概念介绍

1、什么是AOP

AOP (面向切面编程)

在软件业，AOP为Aspect Oriented Programming的缩写，意为：[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。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。

- 在软件业，AOP为[Aspect Oriented Programming](#)的缩写，意为：[面向切面编程](#)
- 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
- 作用：[在不修改【目标类代码】的前提下，可以通过AOP技术去【增强目标类的功能】。通过【预编译方式】或者【运行期动态代理】实现程序功能的统一维护的一种技术](#)
 - 对目标类进行无感知的功能增强。
- AOP是一种编程范式，隶属于软工范畴，指导开发者如何组织程序结构

- AOP最早由**AOP联盟**的组织提出的,制定了一套规范。Spring将AOP思想引入到框架中,必须遵守AOP联盟的规范
- 利用AOP可以对业务代码中【**业务逻辑**】和【**系统逻辑**】进行隔离,从而使得【**业务逻辑**】和【**系统逻辑**】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。

微服务框架(SpringCloudAlibaba) ---> 针对业务层面的拆分 ----> 增加了系统处理的复杂性(服务治理) ----> 服务网格(istio, 在服务网格架构基础之上, 将业务实现和服务治理进行了拆分。)

如果应用架构又改变了 ----> 微服务框架技术就不吃香了 -----> 不变的东西(架构思想、强大的基础[数据结构与算法、JVM、并发编程、Netty、MySQL])

2、为什么使用AOP

业务层存在的问题:

```
public class UserServiceImpl{
    public void saveUser(){

        // C 程序员: 日志监控
        // B 程序员: 开启事务--系统逻辑

        // A 程序员: 编写业务逻辑代码--程序员更应该关注业务的实现, 而不需要关心系统逻辑

        // B 程序员: 关闭事务--系统逻辑
    }
}

public class OrderServiceImpl{
    public void saveOrder(){

        // 开启事务

        // 编写业务逻辑代码

        // 关闭事务
    }
}
```

需要改善的地方：

1. 抽象共性代码（系统功能）
2. 将代码职责分离（系统功能、业务功能）
3. 保证代码的扩展性

- 作用：

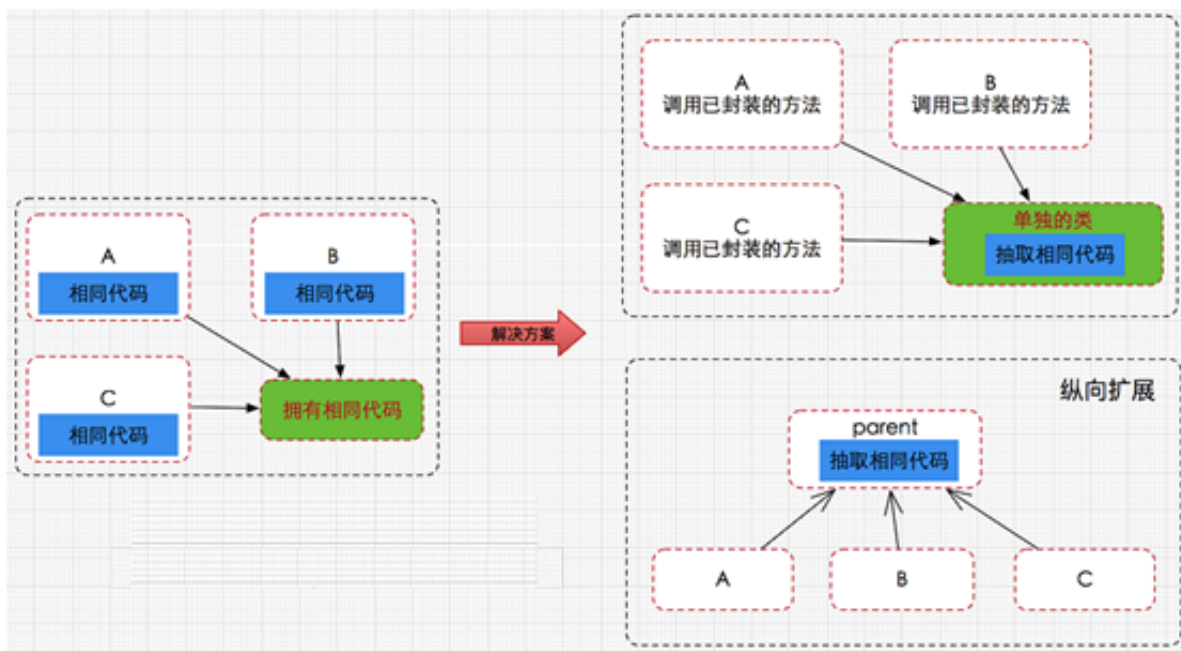
AOP采取**横向抽取机制**，补充了 传统**纵向继承体系**（OOP）无法解决的重复性代码优化（性能监视、事务管理、安全检查、缓存），将业务逻辑和系统处理的代码（关闭连接、事务管理、操作日志记录）解耦。

- 优势：

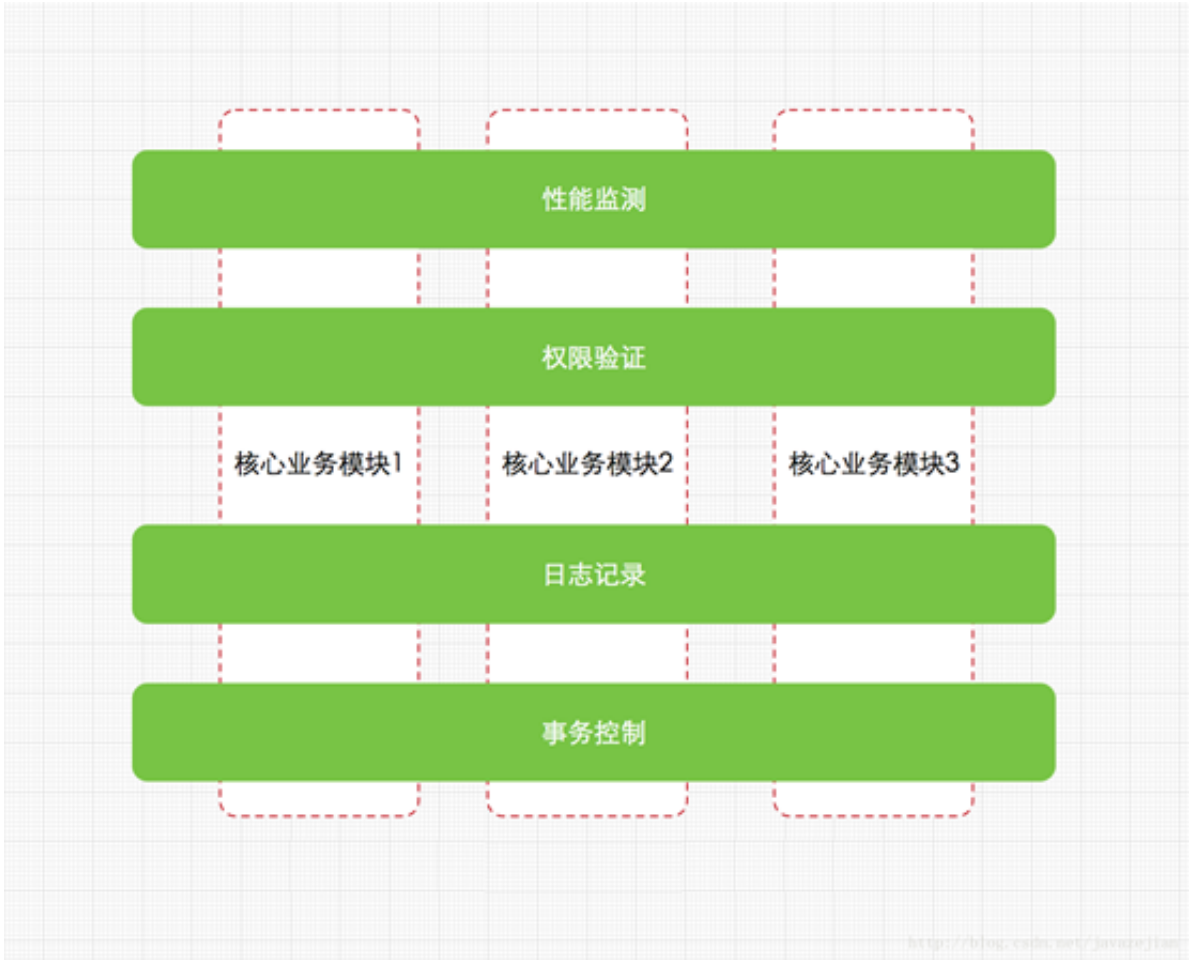
重复性代码被抽取出来之后，**维护更加方便**

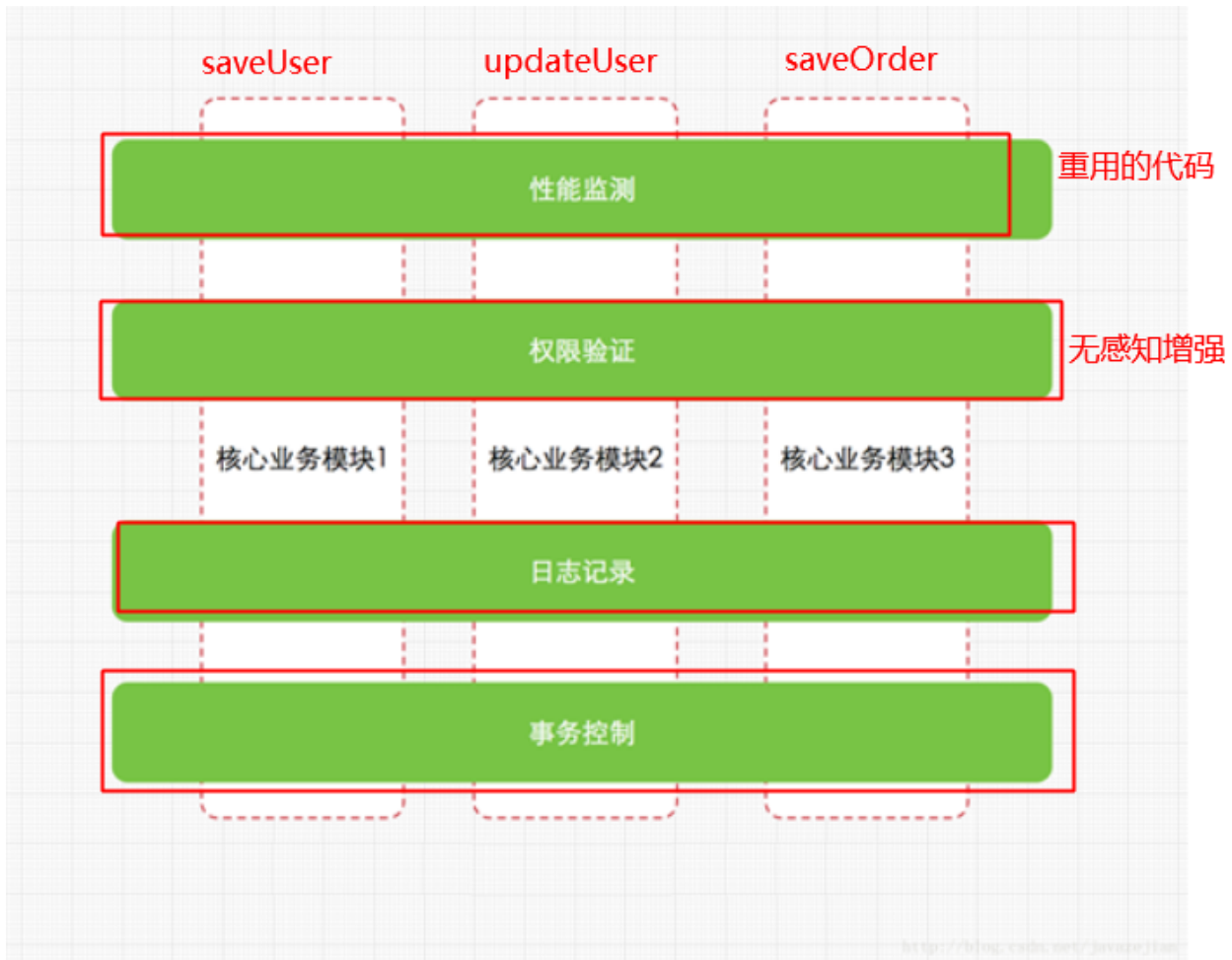
不想修改原有代码前提下，可以动态横向添加共性代码。

- 纵向继承体系(OOP):



- 横向抽取机制(AOP):





3、AOP相关术语介绍

3.1 术语解释

- **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。通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

- **Introduction(引介)**

引介是一种特殊的通知在不修改类代码的前提下，`Introduction`可以在运行期为类动态地添加一些方法或Field

- **Target(目标对象)**

代理的目标对象

- **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

- **Proxy (代理)**

一个类被AOP织入增强后，就产生一个结果代理类

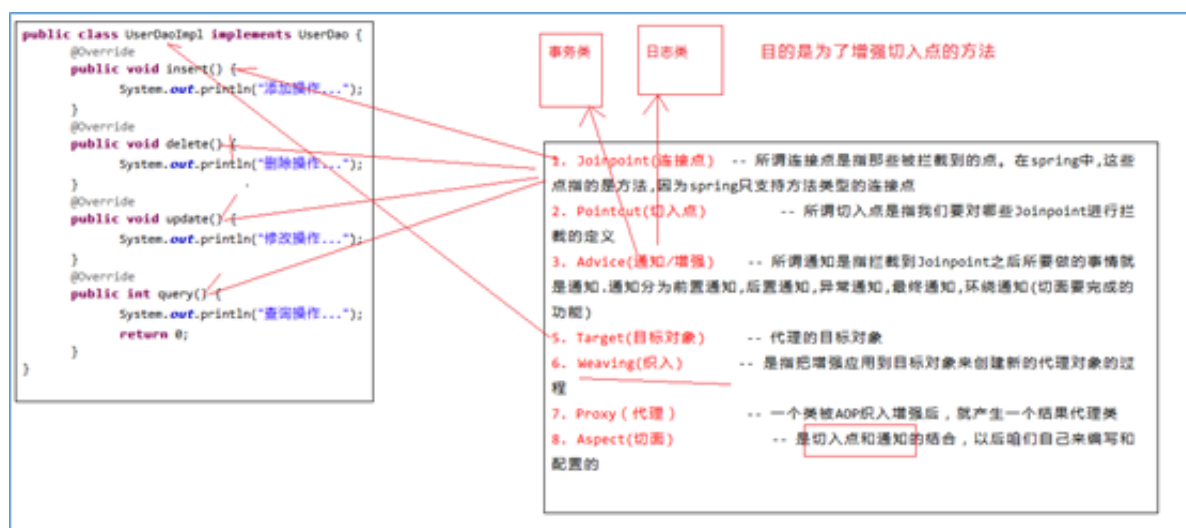
- **Aspect(切面)**

是切入点 and 通知的结合，以后咱们自己来编写和配置的

- **Advisor (通知器、顾问)**

和Aspect很相似

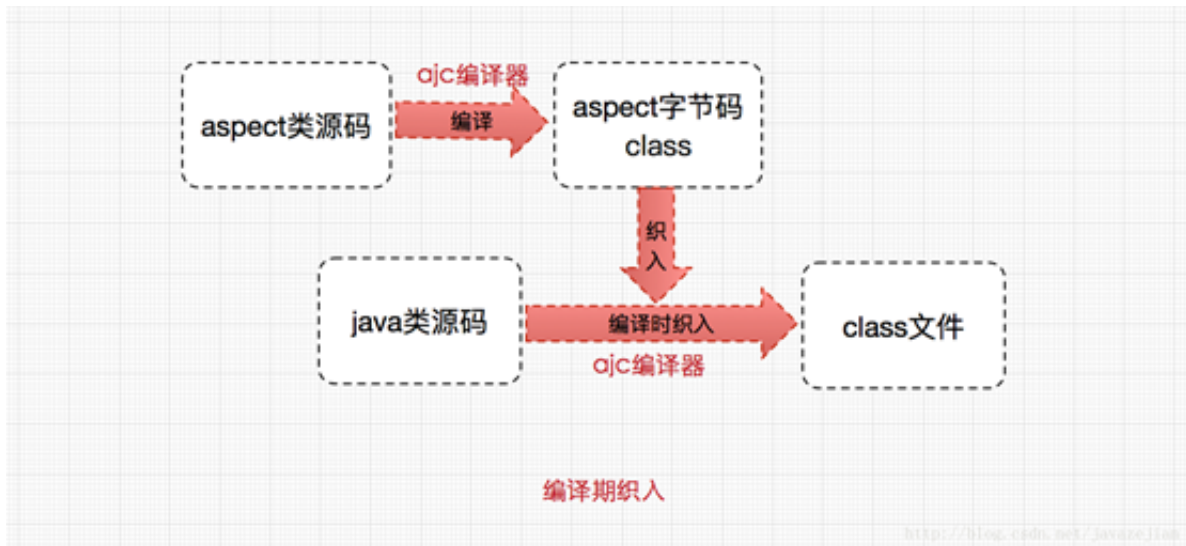
3.2 图示说明



4、AOP实现之AspectJ（了解）

- [AspectJ](#)是一个Java实现的AOP框架，它能够对java代码进行AOP编译（一般在编译期进行），让java代码具有AspectJ的AOP功能（当然需要特殊的编译器）
- 可以说AspectJ是目前实现AOP框架中最成熟，功能最丰富的语言。更幸运的是，AspectJ与java程序完全兼容，几乎是无缝关联，因此对于有java编程基础的工程师，上手和使用都非常容易。
- 了解AspectJ应用到java代码的过程（这个过程称为[织入](#)），对于织入这个概念，可以简单理解为aspect(切面)应用到目标函数(类)的过程。
- 对于织入这个过程，一般分为[动态织入](#)和[静态织入](#)，[动态织入的方式是在运行时动态将要增强的代码织入到目标类中](#)，这样往往是[通过动态代理技术完成的](#)，如Java JDK的动态代理(Proxy，底层通过反射实现)或者CGLIB的动态代理(底层通过继承实现)，[Spring AOP采用的就是基于运行时增强的代理技术](#)
- [AspectJ采用的就是静态织入的方式](#)。[AspectJ主要采用的是编译期织入](#)，在这个期间使用AspectJ的ajc编译器(类似javac)把aspect类编译成class字节码后，在java目标类编译时织入，即先编译

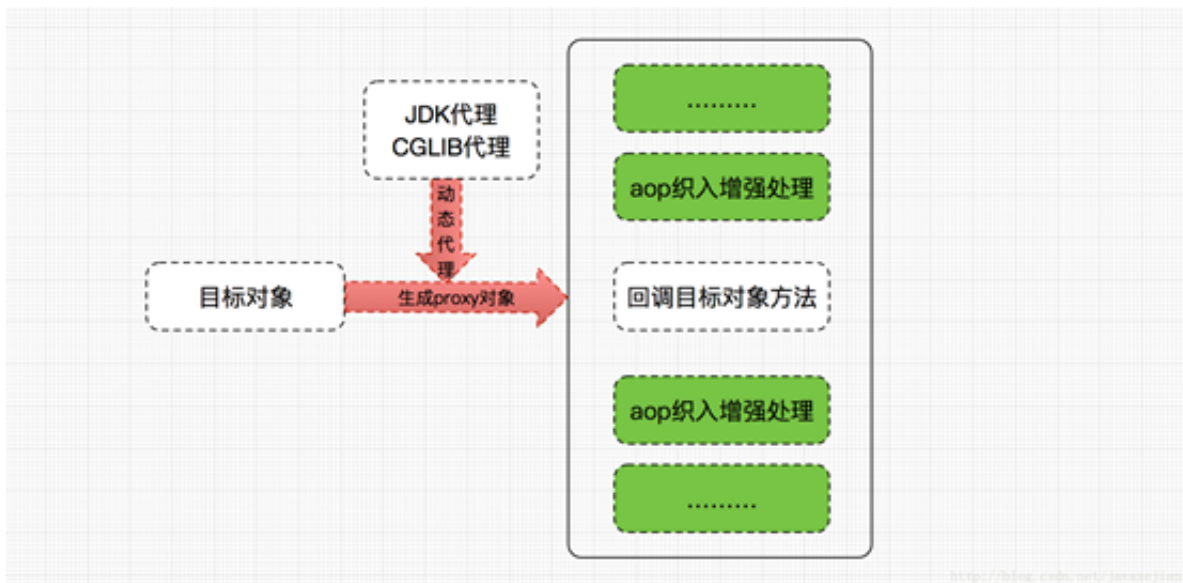
aspect类再编译目标类。



5、AOP实现之Spring AOP(了解)

5.1 实现原理分析

- Spring AOP是通过动态代理技术实现的
- 而动态代理是基于反射设计的。（关于反射的知识，请自行学习）
- 动态代理技术的实现方式有两种：基于接口的JDK动态代理和基于继承的CGLIB动态代理。



1) JDK动态代理

目标对象必须实现接口

```
// JDK代理对象工厂&代理对象方法调用处理器
public class JDKProxyFactory implements InvocationHandler {

    // 目标对象的引用
    private Object target;

    // 通过构造方法将目标对象注入到代理对象中
```

```

public JDKProxyFactory(Object target) {
    super();
    this.target = target;
}

/**
 * @return
 */
public Object getProxy() {

    // 如何生成一个代理类呢?
    // 1、编写源文件
    // 2、编译源文件为class文件
    // 3、将class文件加载到JVM中(ClassLoader)
    // 4、将class文件对应的对象进行实例化(反射)

    // Proxy是JDK中的API类
    // 第一个参数: 目标对象的类加载器
    // 第二个参数: 目标对象的接口
    // 第二个参数: 代理对象的执行处理器
    Object object = Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(),
        this);

    return object;
}

/**
 * 代理对象会执行的方法
 */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    Method method2 = target.getClass().getMethod("saveUser", null);
    Method method3 =
Class.forName("com.sun.proxy.$Proxy4").getMethod("saveUser", null);
    System.out.println("目标对象的方法:" + method2.toString());
    System.out.println("目标接口的方法:" + method.toString());
    System.out.println("代理对象的方法:" + method3.toString());
    System.out.println("这是jdk的代理方法");
    // 下面的代码, 是反射中的API用法
    // 该行代码, 实际调用的是[目标对象]的方法
    // 利用反射, 调用[目标对象]的方法
    Object returnValue = method.invoke(target, args);

    return returnValue;
}
}

```

2) Cglib动态代理

- 目标对象不需要实现接口
- 底层是通过继承目标对象产生代理子对象（代理子对象中继承了目标对象的方法，并可以对该方法进行增强）

```
public class CgLibProxyFactory implements MethodInterceptor {

    /**
     * @param clazz
     * @return
     */
    public Object getProxyByCgLib(Class clazz) {
        // 创建增强器
        Enhancer enhancer = new Enhancer();
        // 设置需要增强的类的类对象
        enhancer.setSuperclass(clazz);
        // 设置回调函数
        enhancer.setCallback(this);
        // 获取增强之后的代理对象
        return enhancer.create();
    }

    /**
     * Object proxy:这是代理对象，也就是[目标对象]的子类
     * Method method:[目标对象]的方法
     * Object[] arg:参数
     * MethodProxy methodProxy: 代理对象的方法
     */
    @Override
    public Object intercept(Object proxy, Method method, Object[] arg,
        MethodProxy methodProxy) throws Throwable {
        // 因为代理对象是目标对象的子类
        // 该行代码，实际调用的是父类目标对象的方法
        System.out.println("这是cglib的代理方法");

        // 通过调用子类[代理类]的invokeSuper方法，去实际调用[目标对象]的方法
        Object returnValue = methodProxy.invokeSuper(proxy, arg);
        // 代理对象调用代理对象的invokeSuper方法，而invokeSuper方法会去调用目标类的invoke方法完成目标对象的调用

        return returnValue;
    }
}
```

ASM API使用：

```
ClassWriter classWriter = new ClassWriter(0);
classWriter.visit(Opcodes.V1_8, Opcodes.ACC_PUBLIC, className, null,
    "java/lang/Object", null);
MethodVisitor initVisitor = classWriter.visitMethod(Opcodes.ACC_PUBLIC, "<init>",
    "()V", null, null);
initVisitor.visitCode();//访问开始
initVisitor.visitVarInsn(Opcodes.ALOAD, 0);//this指针入栈
initVisitor.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/lang/Object", "<init>",
    "()V");//调用构造函数
initVisitor.visitInsn(Opcodes.RETURN);
initVisitor.visitMaxs(1, 1);//设置栈长、本地变量数
```

5.2 使用

- 其使用**ProxyFactoryBean**创建：
- 使用 **<aop:advisor>** 定义通知器的方式实现AOP则需要通知类实现Advice接口
- 增强（通知）的类型有：

- 前置通知：org.springframework.aop.MethodBeforeAdvice
- 后置通知：org.springframework.aop.AfterReturningAdvice
- 环绕通知：org.aopalliance.intercept.MethodInterceptor
- 异常通知：org.springframework.aop.ThrowsAdvice

二、基于AspectJ的AOP使用

其实就是指的Spring + AspectJ整合，不过Spring已经将AspectJ收录到自身的框架中了，并且底层织入依然是采取的动态织入方式。

1 添加依赖

```
<!-- 基于AspectJ的aop依赖 -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>5.0.7.RELEASE</version>
</dependency>
<dependency>
  <groupId>aopalliance</groupId>
  <artifactId>aopalliance</artifactId>
  <version>1.0</version>
</dependency>
```

2 编写目标类和目标方法

编写接口和实现类（目标对象）

UserService接口

UserServiceImpl实现类

配置目标类，将目标类交给spring IoC容器管理

```
<context:component-scan base-package="sourcecode.ioc" />
```

3 使用XML实现

1) 实现步骤

编写通知（增强类，一个普通的类）

```
public class MyAdvice {

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

配置通知，将通知类交给spring IoC容器管理

```
<!-- 配置通知、增强 -->
<bean name="myAdvice" class="cn.spring.advice.MyAdvice"></bean>
```

配置AOP 切面

```
<!-- 配置通知、增强 -->
<bean name="myAdvice" class="cn.spring.advice.MyAdvice"></bean>

<!-- AOP配置 -->
<aop:config>
  <aop:aspect ref="myAdvice">
    <!-- method: 指定要增强的方法，也就是指定通知类中的增强功能方法 -->
    <!-- pointcut: 指定切入点，需要通过表达式来指定 -->
    <aop:before method="log"
      pointcut="execution(void cn.spring.dao.UserDaoImpl.insert())" />
  </aop:aspect>
</aop:config>
```

2) 切入点表达式

切入点表达式的格式：

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

表达式格式说明：

- execution：必须要
- 修饰符：可省略
- 返回值类型：必须要，但是可以使用*通配符
- 包名：

** 多级包之间使用.分割

** 包名可以使用*代替，多级包名可以使用多个*代替

** 如果想省略中间的包名可以使用 ..

- 类名

** 可以使用*代替

** 也可以写成*DaoImpl

- 方法名：

** 也可以使用*好代替

** 也可以写成add*

- 参数：

** 参数使用*代替

** 如果有多个参数，可以使用 ..代替

3) 通知类型

通知类型（五种）：前置通知、后置通知、最终通知、环绕通知、异常抛出通知。

前置通知：

- * 执行时机：目标对象方法之前执行通知
- * 配置文件：<aop:before method="before" pointcut-ref="myPointcut"/>
- * 应用场景：方法开始时可以进行校验

后置通知：

- * 执行时机：目标对象方法之后执行通知，有异常则不执行了
- * 配置文件：<aop:after-returning method="afterReturning" pointcut-ref="myPointcut"/>
- * 应用场景：可以修改方法的返回值

最终通知：

- * 执行时机：目标对象方法之后执行通知，有没有异常都会执行
- * 配置文件：<aop:after method="after" pointcut-ref="myPointcut"/>
- * 应用场景：例如像释放资源

环绕通知：

- * 执行时机：目标对象方法之前和之后都会执行。
- * 配置文件：<aop:around method="around" pointcut-ref="myPointcut"/>
- * 应用场景：事务、统计代码执行时机

异常抛出通知：

- * 执行时机：在抛出异常后通知
- * 配置文件：<aop:after-throwing method="afterThrowing" pointcut-ref="myPointcut"/>
- * 应用场景：包装异常

4 使用注解实现

1) 实现步骤

编写切面类（注意不是通知类，因为该类中可以指定切入点）

```
/**
 * 切面类（通知+切入点）
 *
 * @author think
 *
 */
// @Aspect：标记该类是一个切面类
@Component("myAspect")
@Aspect
public class MyAspect {

    // @Before：标记该方法是一个前置通知
    // value：切入点表达式
    @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    public void log() {
        System.out.println("记录日志...");
    }
}
```

配置切面类

```
<context:component-scan base-package="com.kkb.spring"/>
```

开启AOP自动代理

```
<!-- AOP基于注解的配置-开启自动代理 -->
<aop:aspectj-autoproxy />
```

2) 环绕通知注解配置

@Around

作用：

把当前方法看成是环绕通知。属性：

value：

用于指定切入点表达式，还可以指定切入点表达式的引用。


```
@Around(value = "execution(* *.*(..))")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) {
    // 定义返回值
    Object rtValue = null;
    try {
        // 获取方法执行所需的参数
        Object[] args = joinPoint.getArgs();
        // 前置通知：开启事务beginTransaction();
        // 执行方法
        rtValue = joinPoint.proceed(args);
        // 后置通知：提交事务commit();
    } catch (Throwable e) {
        // 异常通知：回滚事务rollback(); e.printStackTrace();
    } finally {
        // 最终通知：释放资源release();
    }
    return rtValue;
}
```

3) 定义通用切入点

使用@PointCut注解在切面类中定义一个通用的切入点，其他通知可以引用该切入点

```

// @Aspect：标记该类是一个切面类
@Aspect
public class MyAspect {

    // @Before：标记该方法是一个前置通知
    // value：切入点表达式
    // @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    @Before(value = "MyAspect.fn()")
    public void log() {
        System.out.println("记录日志...");
    }

    // @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    @Before(value = "MyAspect.fn()")
    public void validate() {
        System.out.println("进行后台校验...");
    }

    // 通过@Pointcut定义一个通用的切入点
    @Pointcut(value = "execution(* *..*.*DaoImpl.*(..))")
    public void fn() {
    }
}

```

5 纯注解方式

```

@Configuration
@ComponentScan(basePackages="com.kkb")
@EnableAspectJAutoProxy
public class SpringConfiguration {
}

```

三、代理模式

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作。代理又分为[动态代理](#)和[静态代理](#)

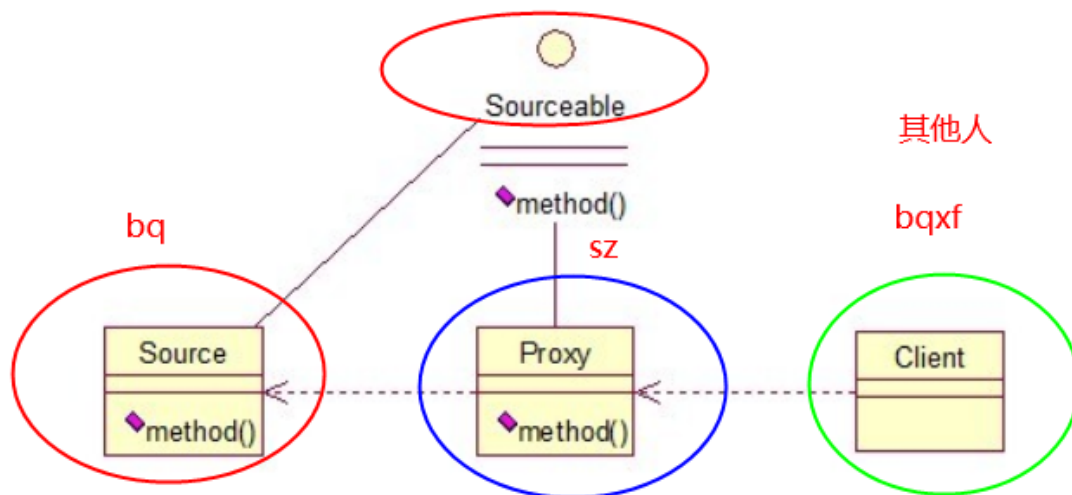
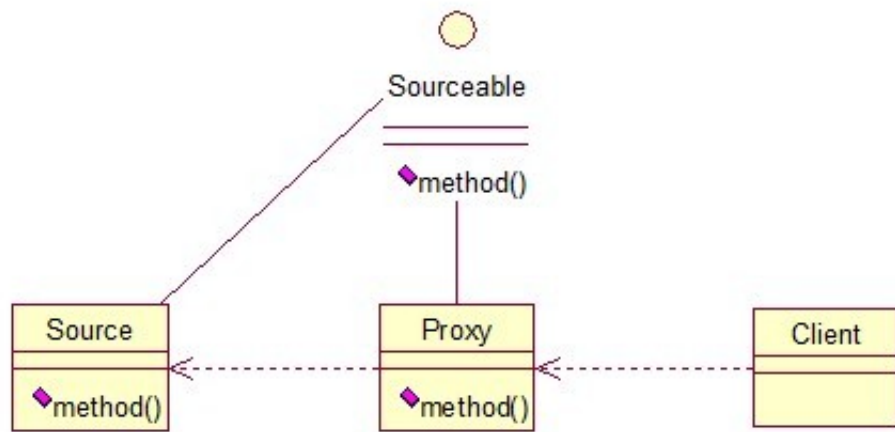
1 静态代理

比如我们在租房子的时候回去找中介，为什么呢？

因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。

再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。

先来看看关系图：



根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

```

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

```

public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
  
```

静态代理的重点：

- [需要为源类手动编写一个代理类](#)
- 代理类和源类实现同一接口。

- 代理对象持有源对象的引用。

静态代理的缺点：

- 会产生大量的代理类。

```
public class Proxy implements Sourceable {
    // 持有源对象的引用
    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
```

测试类：

```
public class ProxyTest {
    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }
}
```

输出：

```
before proxy!
the original method!
after proxy!
```

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1. 修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。
2. 就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

2 动态代理

动态代理在编译期间，不需要为源类去手动编写一个代理类。

只会再运行期间，去为源对象产生一个代理对象。

JDK动态代理和Cglib动态代理的区别

1. JDK动态代理是Java自带的，cglib动态代理是第三方jar包提供的。
2. JDK动态代理是针对【拥有接口的目标类】进行动态代理的，而Cglib是【非final类】都可以进行动态代理。但是Spring【优先使用JDK动态代理】。
3. JDK动态代理实现的逻辑是【目标类】和【代理类】都【实现同一个接口】，目标类和代理类【是平级的】。而Cglib动态代理实现的逻辑是给【目标类】生个孩子（子类，也就是【代理类】），【目标类】和【代理类】是父子继承关系】。
4. JDK动态代理在早期的JDK1.6左右性能比cglib差，但是在JDK1.8以后cglib和jdk的动态代理性能基本上差不多。反而jdk动态代理性能更加的优越。

动态代理主要关注两个点：

代理对象如何创建的[底层原理](#)？

代理对象如何执行的[原理分析](#)？

2.1 JDK动态代理

方式1（两个类）

产生代理对象

```
// 为指定的目标对象，产生代理对象的工厂
public class JDKProxyFactory {

    private Object target;

    public JDKProxyFactory(Object target){
        this.target = target ;
    }
}
```

```
// 只能为实现了接口的类产生代理对象
public Object getProxy() {
    // 1.目标类的类加载器
    // 2.目标类的接口集合
    // 3.代理对象被调用时的调用处理器
    Object proxy = Proxy.newProxyInstance(target.getClass().getClassLoader(),
    target.getClass().getInterfaces(),
                                new MyInvocationHandler(target));

    // 底层原理分析
    // 第一步：JDK编写java源代码
    // 第二步：编译源代码（JDK自带API）

    // 参考根据原类如何写出静态代理类的逻辑，去写出来一个要产生的代理对象对应的.java文件
    // .java---> .class ---> Class ---> Object

    return proxy;
}
}
```

底层原理分析

```
// 1.JDK在运行时根据代理类的规则以及目标类相关信息，编写代理类的源代码（.java）
// 2.JDK对java源代码进行编译（.class）
// 3.JDK根据目标类的类加载器，去将代理类的class文件加载到JVM中
// 4.JVM会根据加载到的class信息，产生一个代理对象

Object proxy = Proxy.newProxyInstance(target.getClass().getClassLoader(),
    target.getClass().getInterfaces(),
                                new MyInvocationHandler(target));
```

程序员A使用静态代理思路推演：

- 只有目标类

```
public interface UserService {

    //原对象target
    void saveUser() ;
}
```

```
public class UserServiceImpl implements UserService {  
  
    @Override  
    public void saveUser() {  
        System.out.println("添加用户");  
    }  
}
```

- 会根据需求写出代理类

```
getSourceCode(){  
    StringBuffer sb;  
  
  
}
```



```

public class $Proxy6 implements 目标类的接口集合{

    // 构造注入
    private InvocationHandler h;

    private Method m1;

    // 遍历接口中的方法
    public void saveUser() {
        this.h.invoke(this,m1, saveUser方法参数);
    }

    static{
        m1 = Class.forName("目标类的全限定名").getDeclaredMethod("saveUser");
    }

    // ...

}

```

以上代码，只需要传递给它目标类的接口集合，以及InvocationHandler的实现类，那么代理类的代码，JDK就可以自行写出。

执行代理对象方法

```

// [代理对象]方法调用处理器
public class MyInvocationHandler implements InvocationHandler {

    // 目标对象的引用
    private Object target;

    // 通过构造方法将目标对象注入到代理对象中
    public MyInvocationHandler(Object target) {
        super();
        this.target = target;
    }

    /**
     * 代理对象会执行的方法
     * 1.代理对象
     * 2.目标对象方法
     * 3.方法参数
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {

```

```

        // 增强代码....(日志记录)

        // 下面的代码，是反射中的API用法
        // 该行代码，实际调用的是[目标对象]的方法
        // 利用反射，调用[目标对象]的方法

        // 配置文件+反射的方式+容器 (Map、List)
        // 将容器中的功能进行有序的调用即可 (MethodInvocation.proceed())和
        MethodInvocation.proceed())
        Object returnValue = method.invoke(target, args);

        // 增强代码....(性能监控)

        return returnValue;
    }
}

```

方式2（一个类，推荐）

```

// [代理对象]方法调用处理器
public class JDKProxyFactory implements InvocationHandler {

    // 目标对象的引用
    private Object target;

    // 通过构造方法将目标对象注入到代理对象中
    public JDKProxyFactory(Object target) {
        super();
        this.target = target;
    }

    // 只能为实现了接口的类产生代理对象
    public Object getProxy() {
        // 1.目标类的类加载器
        // 2.目标类的接口集合
        // 3.代理对象被调用时的调用处理器
        Object proxy = Proxy.newProxyInstance(target.getClass().getClassLoader(),
        target.getClass().getInterfaces(),
        this);

        return proxy;
    }

    /**
     * 代理对象会执行的方法
     * 1.代理对象
     * 2.目标对象方法
     * 3.方法参数
     */
}

```

```

    */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {
        // 增强代码....(日志记录)

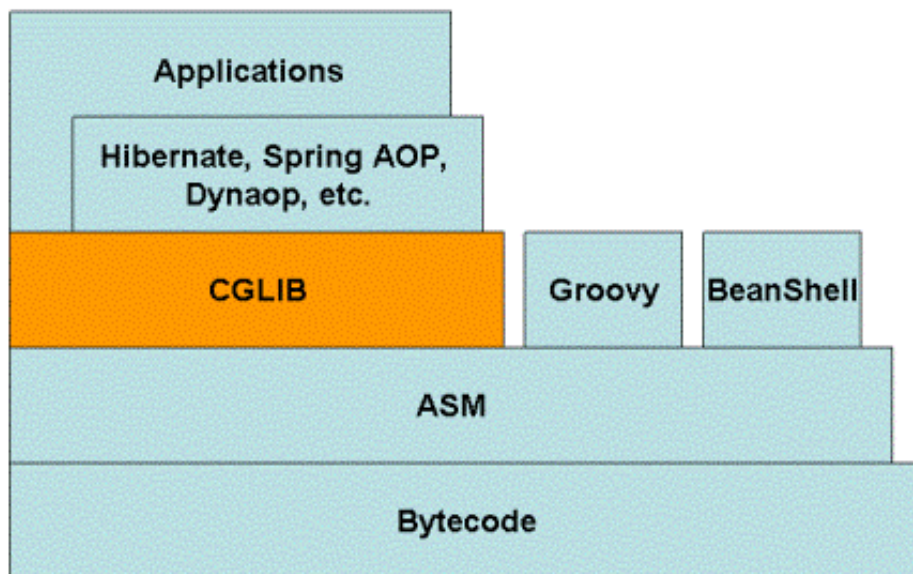
        // 下面的代码, 是反射中的API用法
        // 该行代码, 实际调用的是[目标对象]的方法
        // 利用反射, 调用[目标对象]的方法
        Object returnValue = method.invoke(target, args);

        // 增强代码....(性能监控)

        return returnValue;
    }
}

```

2.2 CGLib动态代理(ASM)



是通过子类继承父类的方式去实现的动态代理，不需要接口。

```

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

import org.objectweb.asm.ClassWriter;

```

```

import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;

public class MyGenerator {

    public static void main(String[] args) throws IOException {

        System.out.println();
        ClassWriter classWriter = new ClassWriter(0);
        // 通过visit方法确定类的头部信息
        classWriter.visit(Opcodes.V1_7, // java版本
            Opcodes.ACC_PUBLIC, // 类修饰符
            "Programmer", // 类的全限定名
            null, "java/lang/Object", null);

        //创建构造函数
        MethodVisitor mv = classWriter.visitMethod(Opcodes.ACC_PUBLIC, "<init>", "()", null, null);
        mv.visitCode();
        mv.visitVarInsn(Opcodes.ALOAD, 0);
        mv.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/lang/Object", "<init>", "());");
        mv.visitInsn(Opcodes.RETURN);
        mv.visitMaxs(1, 1);
        mv.visitEnd();

        // 定义code方法
        MethodVisitor methodVisitor =
classWriter.visitMethod(Opcodes.ACC_PUBLIC, "code", "()", null, null);
        methodVisitor.visitCode();
        methodVisitor.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System",
"out", "Ljava/io/PrintStream;");
        methodVisitor.visitLdcInsn("I'm a Programmer,Just Coding.....");
        methodVisitor.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
"java/io/PrintStream", "println", "(Ljava/lang/String;)V");
        methodVisitor.visitInsn(Opcodes.RETURN);
        methodVisitor.visitMaxs(2, 2);
        methodVisitor.visitEnd();
        classWriter.visitEnd();
        // 使classWriter类已经完成
        // 将classWriter转换成字节数组写到文件里面去
        byte[] data = classWriter.toByteArray();
        File file = new File("D://Programmer.class");
        FileOutputStream fout = new FileOutputStream(file);
        fout.write(data);
        fout.close();

    }
}

```

方式1

产生代理对象的代码：

```
public class CgLibProxyFactory {

    /**
     * @param clazz
     * @return
     */
    public Object getProxy(Class clazz) {
        // 创建增强器
        Enhancer enhancer = new Enhancer();
        // 设置需要增强的类的类对象
        enhancer.setSuperclass(clazz);
        // 设置回调函数
        enhancer.setCallback(new MyMethodInterceptor());
        // 获取增强之后的代理对象
        return enhancer.create();
    }
}
```

执行代码对象方法的代码：

```
public class MyMethodInterceptor implements MethodInterceptor {

    /**
     * Object proxy:这是代理对象，也就是[目标对象]的子类
     * Method method:[目标对象]的方法
     * Object[] arg:参数
     * MethodProxy methodProxy:代理对象的方法
     */
    @Override
    public Object intercept(Object proxy, Method method, Object[] arg,
        MethodProxy methodProxy) throws Throwable {
        // 因为代理对象是目标对象的子类
        // 该行代码，实际调用的是父类目标对象的方法
        System.out.println("这是cglib的代理方法");

        // 通过调用子类[代理类]的invokeSuper方法，去实际调用[目标对象]的方法
        // Object returnValue = method.invoke(目标对象,arg);

        // FastClass机制 优于 反射调用方法
        Object returnValue = methodProxy.invokeSuper(proxy, arg);
        // 代理对象调用代理对象的invokeSuper方法，而invokeSuper方法会去调用目标类的invoke方法完成目标对象的调用
    }
}
```

```

        return returnValue;
    }
}

```

方式2

```

public class CgLibProxyFactory implements MethodInterceptor {

    /**
     * @param clazz
     * @return
     */
    public Object getProxy(Class clazz) {
        // 创建增强器
        Enhancer enhancer = new Enhancer();
        // 设置需要增强的类的类对象
        enhancer.setSuperclass(clazz);
        // 设置回调函数
        enhancer.setCallback(this);
        // 获取增强之后的代理对象
        return enhancer.create();
    }

    /**
     * Object proxy:这是代理对象，也就是[目标对象]的子类
     * Method method:[目标对象]的方法
     * Object[] arg:参数
     * MethodProxy methodProxy: 代理对象的方法
     */
    @Override
    public Object intercept(Object proxy, Method method, Object[] arg,
        MethodProxy methodProxy) throws Throwable {
        // 因为代理对象是目标对象的子类
        // 该行代码，实际调用的是父类目标对象的方法
        System.out.println("这是cglib的代理方法");

        // 通过调用子类[代理类]的invokeSuper方法，去实际调用[目标对象]的方法
        Object returnValue = methodProxy.invokeSuper(proxy, arg);
        // 代理对象调用代理对象的invokeSuper方法，而invokeSuper方法会去调用目标类的invoke方法完成目标对象的调用
        return returnValue;
    }
}

```

四、AOP源码阅读

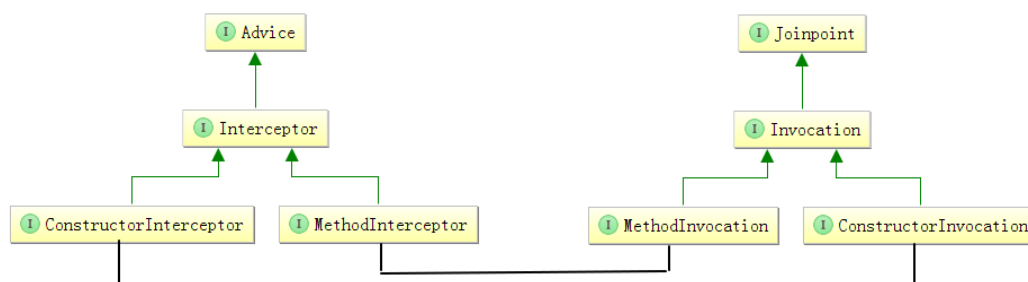
2.1 Spring AOP核心类解析



2.1.1 SpringAOP基础解析类

类名	作用概述
AopNamespaceHandler	AOP命名空间解析类。我们在用AOP的时候，会在Spring配置文件的beans标签中引入：xmlns:aop
AspectJAutoProxy BeanDefinitionParser	解析<aop:aspectj-autoproxy />标签的类。在AopNamespaceHandler中创建的类。
Config BeanDefinitionParser	解析<aop:config /> 标签的类。同样也是在AopNamespaceHandler中创建的类。
AopNamespaceUtils	AOP命名空间解析工具类，在上面两个中被引用。
AopConfigUtils	AOP配置工具类。主要是向Spring容器中注入可以生成Advisor和创建代理对象的bean

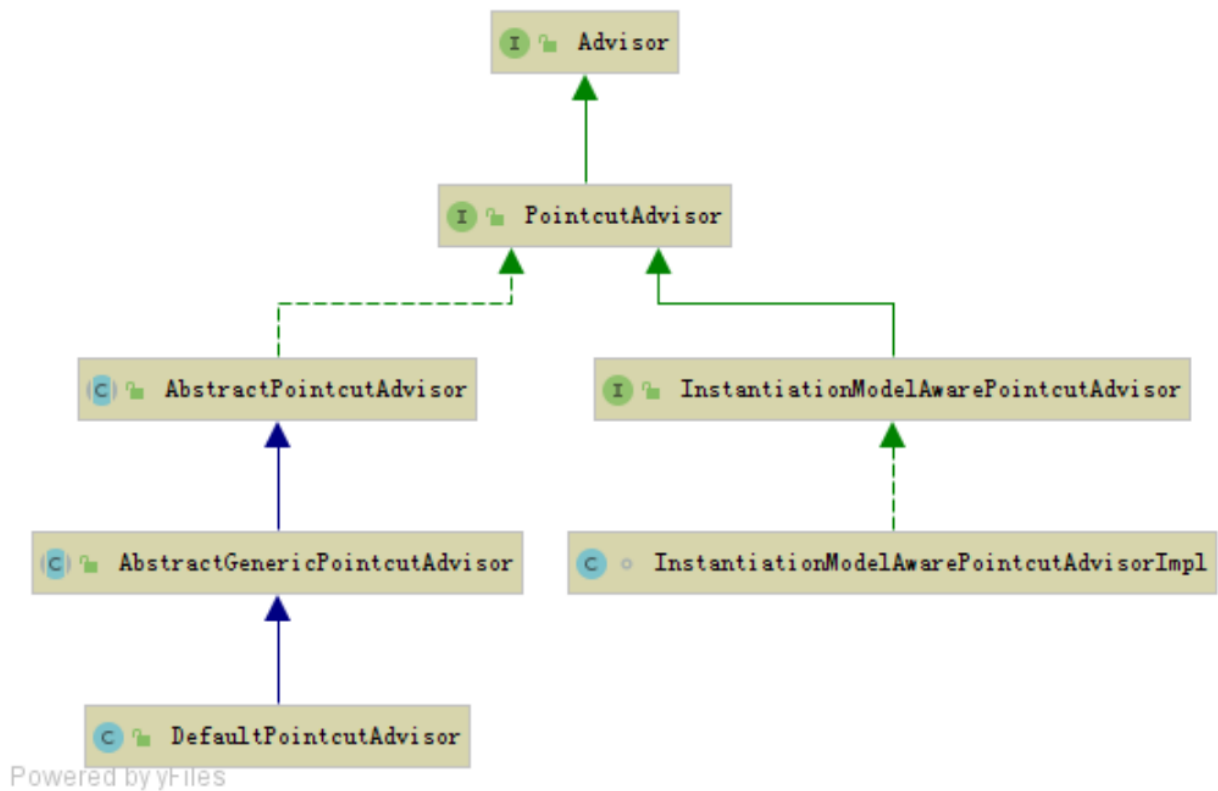
2.1.2 AOP联盟定义的类



类名	作用概述
Advice	AOP联盟中的一个标识接口。 通知和Interceptor顶级类 。我们说的各种通知类型都要实现这个接口。
Interceptor	AOP联盟中进行方法拦截的一个标识接口。是Advice的子类。
MethodInterceptor	方法拦截器 。是Interceptor的一个重要子类。 主要方法：invoke。入参为：MethodInvocation
ConstructorInterceptor	构造方法拦截器 。是Interceptor的另一个重要的子类。在AOP联盟中是可以对构造方法进行拦截的。这样的场景我们应该很少用到。主要方法为:construct入参为ConstructorInvocation
分割线--分割线	
Joinpoint	AOP联盟中的连接点类 。主要的方法是：proceed()执行下一个拦截器。getThis()获取目标对象。
Invocation	AOP拦截的执行类 。是Joinpoint的子类。主要方法：getArguments()获取参数。
MethodInvocation	Invocation的一个重要实现类 。真正执行AOP方法的拦截。主要方法：getMethod()目标方法。
ConstructorInvocation	Invocation的另一个重要实现类。执行构造方法的拦截。主要方法：getConstructor()返回构造方法。

2.1.3 SpringAOP中定义的类

Advisor系列(重点)

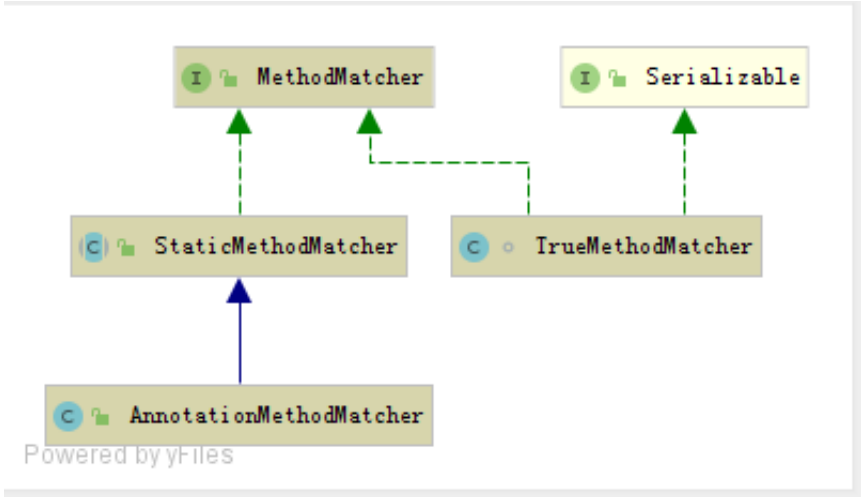


类名	作用概述
Advisor	SpringAOP中的核心类。组合了Advice。
PointcutAdvisor	SpringAOP中Advisor的重要子类。 组合了切点Pointcut和Advice 。
InstantiationModelAwarePointcutAdvisorImpl	PointcutAdvisor的一个重要实现子类。
DefaultPointcutAdvisor	PointcutAdvisor的另一个重要实现子类。可以将Advice包装为Advisor。在SpringAOP中是以Advisor为主线。向Advice靠拢。

Pointcut系列（重点）

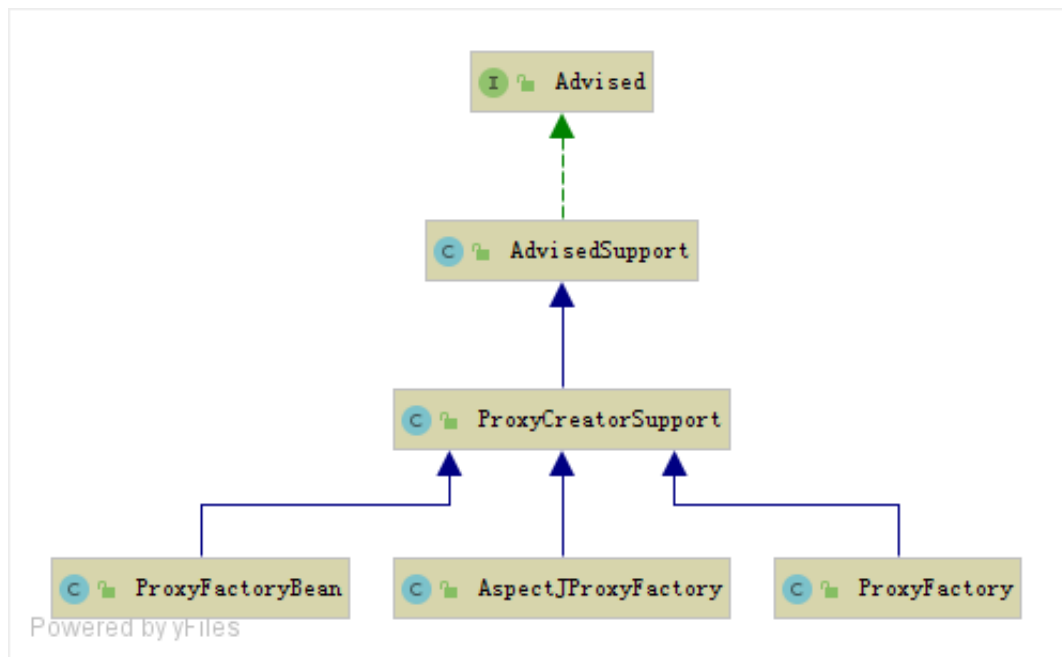
类名	作用概述
Pointcut	SpringAOP中切点的顶级抽象类。
TruePointcut	Pointcut的一个重要实现类。在DefaultPointcutAdvisor中使用的是TruePointcut。在进行切点匹配的时候永远返回true
AspectJExpressionPointcut	Pointcut的一个重要实现类。AspectJ语法切点类。同时实现了 MethodMatcher ，AspectJ语法切点的匹配在这个类中完成。
AnnotationMatchingPointcut	Pointcut的一个重要实现类。注解语法的切点类。
JdkRegexpMethodPointcut	Pointcut的一个重要实现类。正则语法的切点类。

MethodMatcher系列（重点）



类名	作用概述
MethodMatcher	切点匹配连接点的地方。 即类中的某个方法和我们定义的切点表达式是否匹配、能不能被AOP拦截
TrueMethodMatcher	用于返回true
AnnotationMethodMatcher	带有注解的方法的匹配器

Advised系列



类名	作用概述
Advised	SpringAOP中的又一个核心类。 它组合了Advisor和TargetSource即目标对象
AdvisedSupport	Advised的一个实现类。SpringAOP中的一个核心类。继承了 ProxyConfig 实现了Advised。
ProxyCreatorSupport	AdvisedSupport的子类。引用了AopProxyFactory用来创建代理对象。
ProxyFactory	ProxyCreatorSupport的子类。用来创建代理对象。在SpringAOP中用的最多。
ProxyFactoryBean	ProxyCreatorSupport的子类。用来创建代理对象。它实现了BeanFactoryAware、FactoryBean接口
AspectJProxyFactory	ProxyCreatorSupport的子类。用来创建代理对象。使用AspectJ语法。

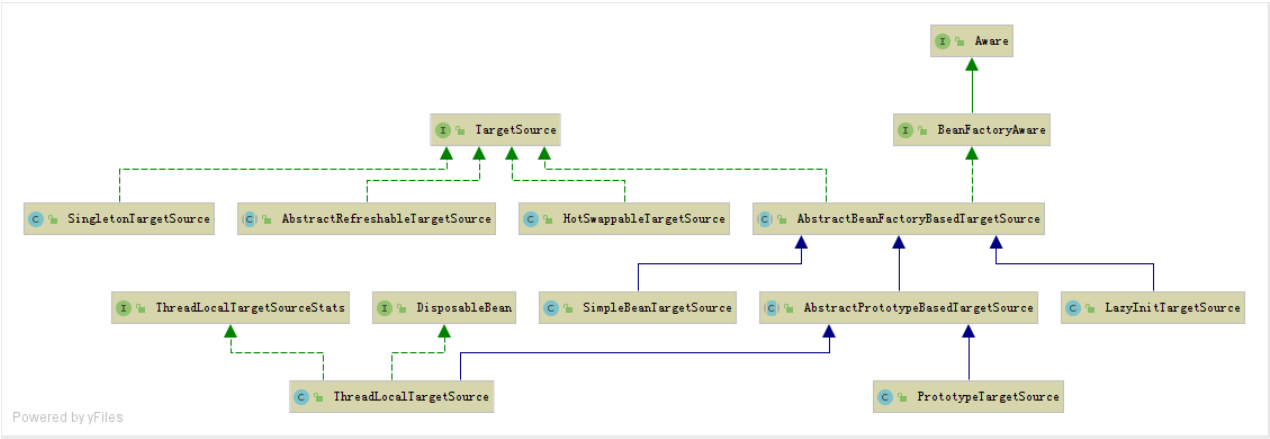
注意：

ProxyFactory、ProxyFactoryBean、AspectJProxyFactory这三个类的使用场景各不相同。但都是生成Advisor和TargetSource、代理对象的关系。

ProxyConfig系列

类名	作用概述
ProxyConfig	SpringAOP中的一个核心类。 在Advised中定义了一系列的配置接口，像：是否暴露对象、是否强制使用CGlib等。 ProxyConfig是对这些接口的实现，但是ProxyConfig却不是Advised的实现类
ProxyProcessorSupport	ProxyConfig的子类
AbstractAutoProxyCreator	ProxyProcessorSupport的重要子类。SpringAOP中的核心类。实现了 SmartInstantiationAwareBeanPostProcessor 、BeanFactoryAware接口。自动创建代理对象的类。我们在使用AOP的时候基本上都是用的这个类来进程Bean的拦截，创建代理对象。
AbstractAdvisorAutoProxyCreator	AbstractAutoProxyCreator的子类。SpringAOP中的核心类。用来创建Advisor和代理对象。
AspectJAwareAdvisorAutoProxyCreator	AbstractAdvisorAutoProxyCreator的子类。使用AspectJ语法创建Advisor和代理对象。
AnnotationAwareAspectJAutoProxyCreator	AspectJAwareAdvisorAutoProxyCreator的子类。使用AspectJ语法创建Advisor和代理对象的类。<aop:aspectj-autoproxy />标签默认注入到SpringAOP中的BeanDefinition。
InfrastructureAdvisorAutoProxyCreator	AbstractAdvisorAutoProxyCreator的子类。SpringAOP中的核心类。基础建设类。Spring事务默认创建代理对象的类。

TargetSource系列



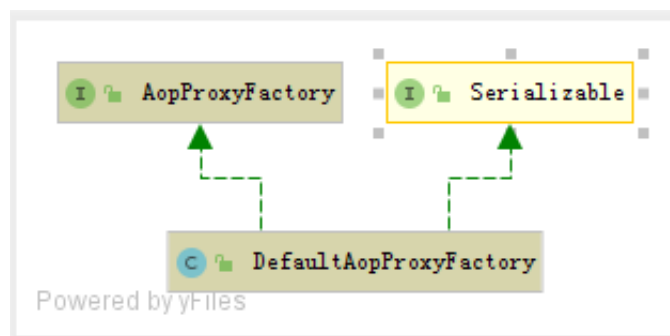
类名	作用概述
TargetSource	持有目标对象的接口。
SingletonTargetSource	TargetSource的子类。适用于单例目标对象。
HotSwappableTargetSource	TargetSource的子类。支持热交换的目标对象
AbstractRefreshableTargetSource	TargetSource的子类。支持可刷新的热部署的目标对象。
AbstractBeanFactoryBasedTargetSource	TargetSource的子类。实现了BeanFactoryAware接口。
SimpleBeanTargetSource	AbstractBeanFactoryBasedTargetSource的子类。从BeanFactory中获取单例Bean。
LazyInitTargetSource	AbstractBeanFactoryBasedTargetSource的子类。从BeanFactory中获取单例Bean。支持延迟初始化。
AbstractPrototypeBasedTargetSource	AbstractBeanFactoryBasedTargetSource的子类。对Prototype类型的Bean的支持。
ThreadLocalTargetSource	AbstractPrototypeBasedTargetSource的子类。和线程上下文相结合的类。
PrototypeTargetSource	AbstractPrototypeBasedTargetSource的子类。从BeanFacory中获取Prototype类型的Bean。

AopProxy系列（重点）

类名	作用概述
AopProxy	生成AOP代理对象的类。
JdkDynamicAopProxy	AopProxy的子类。使用JDK的方式创建代理对象。它持有 Advised 对象。实现了 AopProxy 接口和 InvocationHandler 接口。
CglibAopProxy	AopProxy的子类。使用Cglib的方法创建代理对象。它持有Advised对象。
ObjenesisCglibAopProxy	CglibAopProxy的子类。使用Cglib的方式创建代理对象。它持有Advised对象。

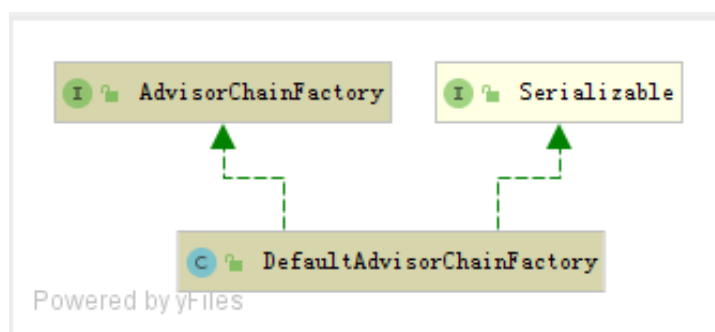
Object getProxy(); ----由JDK或者CGLIB实现

AopProxyFactory系列



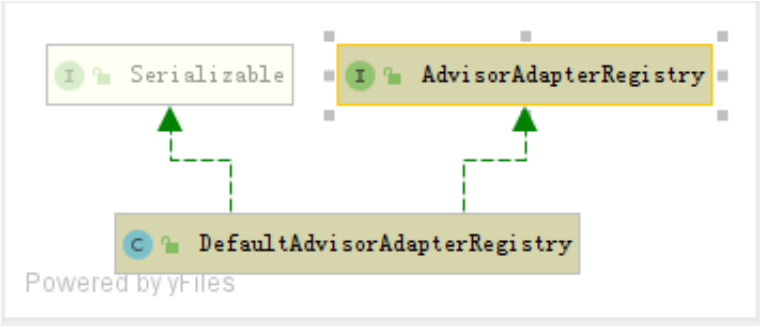
类名	作用概述
AopProxyFactory	创建AOP代理对象的工厂类。选择使用JDK还是Cglib的方式来创建代理对象。
DefaultAopProxyFactory	AopProxyFactory的子类，也是SpringAOP中唯一默认的实现类。

AdvisorChainFactory系列



类名	作用概述
AdvisorChainFactory	获取Advisor链的接口。
DefaultAdvisorChainFactory	AdvisorChainFactory的实现类。也是SpringAOP中唯一默认的实现类。

AdvisorAdapterRegistry系列



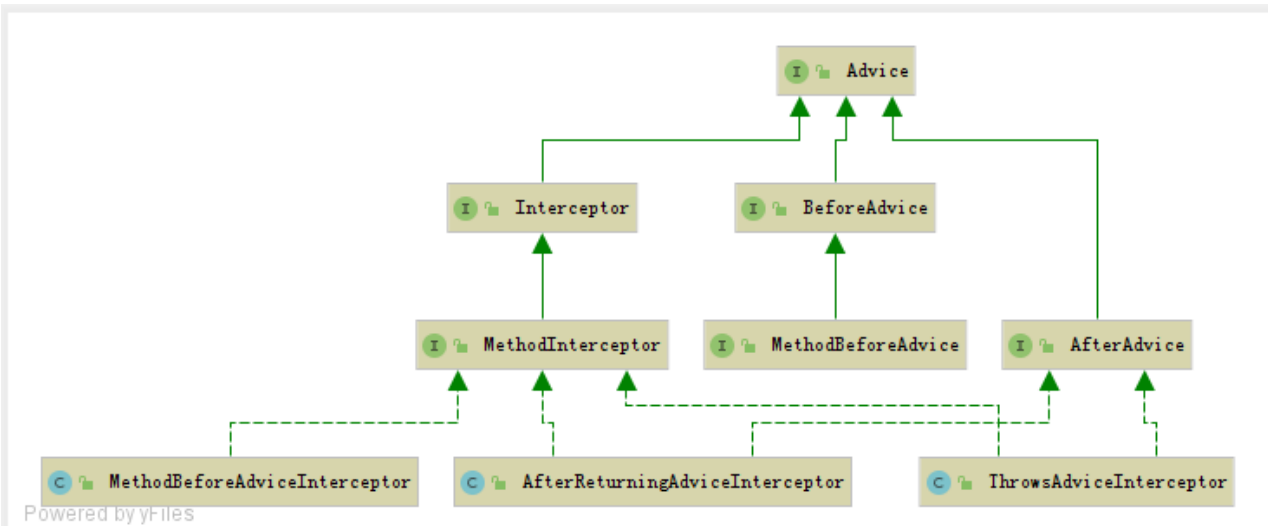
类名	作用概述
AdvisorAdapterRegistry	Advisor适配注册器类。用来将Advice适配为Advisor。将Advisor适配为 MethodInterceptor 。
DefaultAdvisorAdapterRegistry	AdvisorAdapterRegistry的实现类。也是SpringAOP中唯一默认的实现类。持有：MethodBeforeAdviceAdapter、AfterReturningAdviceAdapter、ThrowsAdviceAdapter实例。

AutoProxyUtils系列

类名	作用概述
AutoProxyUtils	SpringAOP自动创建代理对象的工具类。

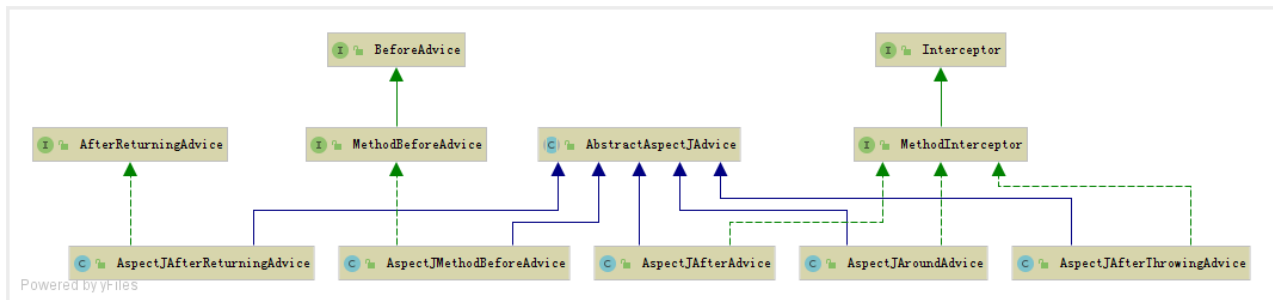
Advice实现系列（重点）

AspectJ有五种通知类型，其中三种是直接可以强转成MethodInterceptor接口。



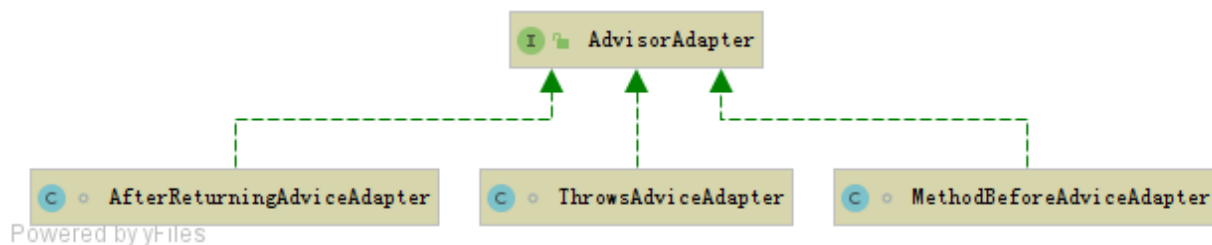
类名	作用概述
BeforeAdvice	前置通知类。直接继承了Advice接口。
MethodBeforeAdvice	BeforeAdvice的子类。定义了方法before。执行前置通知。
MethodBeforeAdviceInterceptor	MethodBefore前置通知Interceptor。实现了MethodInterceptor接口。持有MethodBefore对象。
AfterAdvice	后置通知类。直接继承了Advice接口。
ThrowsAdvice	后置异常通知类。直接继承了AfterAdvice接口。
AfterReturningAdvice	后置返回通知类。直接继承了AfterAdvice接口。
AfterReturningAdviceInterceptor	后置返回通知Interceptor。实现了MethodInterceptor和AfterAdvice接口。持有AfterReturningAdvice实例
ThrowsAdviceInterceptor	后置异常通知Interceptor。实现了MethodInterceptor和AfterAdvice接口。要求方法名为：afterThrowing

AbstractAspectJAdvice系列（重点）



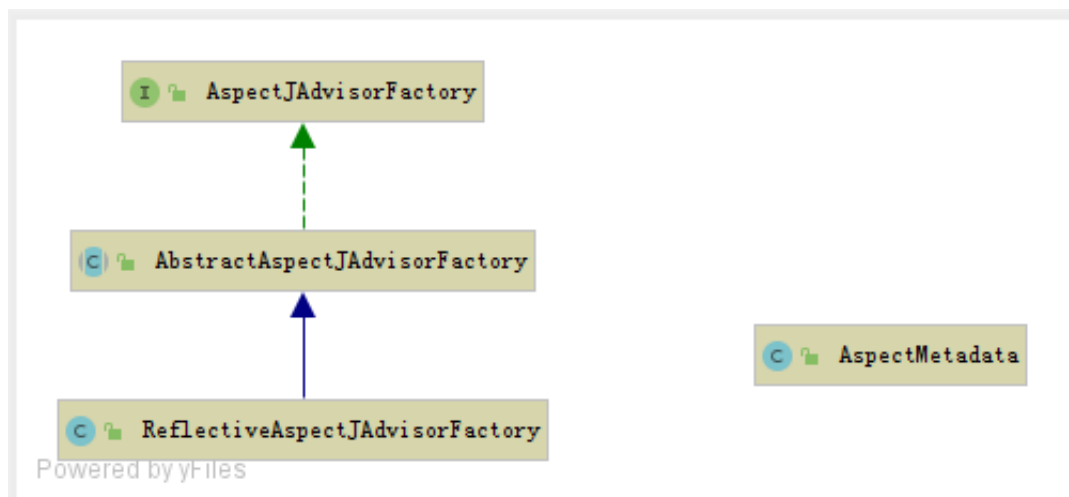
类名	作用概述
AbstractAspectJAdvice	使用AspectJ注解的通知类型顶级父类
AspectJMethodBeforeAdvice	使用AspectJ Before注解的前置通知类型。实现了MethodBeforeAdvice继承了AbstractAspectJAdvice。
AspectJAfterAdvice	使用AspectJ After注解的后置通知类型。实现了MethodInterceptor、AfterAdvice接口。继承了AbstractAspectJAdvice。
AspectJAfterReturningAdvice	使用AspectJ AfterReturning注解的后置通知类型。实现了AfterReturningAdvice、AfterAdvice接口。继承了AbstractAspectJAdvice。
AspectJAroundAdvice	使用AspectJ Around注解的后置通知类型。实现了MethodInterceptor接口。继承了AbstractAspectJAdvice。
AspectJAfterThrowingAdvice	使用AspectJ Around注解的后置通知类型。实现了MethodInterceptor、AfterAdvice接口。继承了AbstractAspectJAdvice。

AdvisorAdapter系列(重点)



类名	作用概述
AdvisorAdapter	Advisor适配器。判断此接口的是不是能支持对应的Advice。五种通知类型，只有三种通知类型适配器。这里可以想一下为什么只有三种。
MethodBeforeAdviceAdapter	前置通知的适配器。支持前置通知类。有一个getInterceptor方法：将Advisor适配为MethodInterceptor。Advisor持有Advice类型的实例，获取MethodBeforeAdvice，将MethodBeforeAdvice适配为MethodBeforeAdviceInterceptor。AOP的拦截过程通过MethodInterceptor来完成。
AfterReturningAdviceAdapter	后置返回通知的适配器。支持后置返回通知类。有一个getInterceptor方法：将Advisor适配为MethodInterceptor。Advisor持有Advice类型的实例，获取AfterReturningAdvice，将AfterReturningAdvice适配为AfterReturningAdviceInterceptor。AOP的拦截过程通过MethodInterceptor来完成。
ThrowsAdviceAdapter	后置异常通知的适配器。支持后置异常通知类。有一个getInterceptor方法：将Advisor适配为MethodInterceptor。Advisor持有Advice类型的实例，获取ThrowsAdvice，将ThrowsAdvice适配为ThrowsAdviceInterceptor。AOP的拦截过程通过MethodInterceptor来完成。

AspectJAdvisorFactory系列

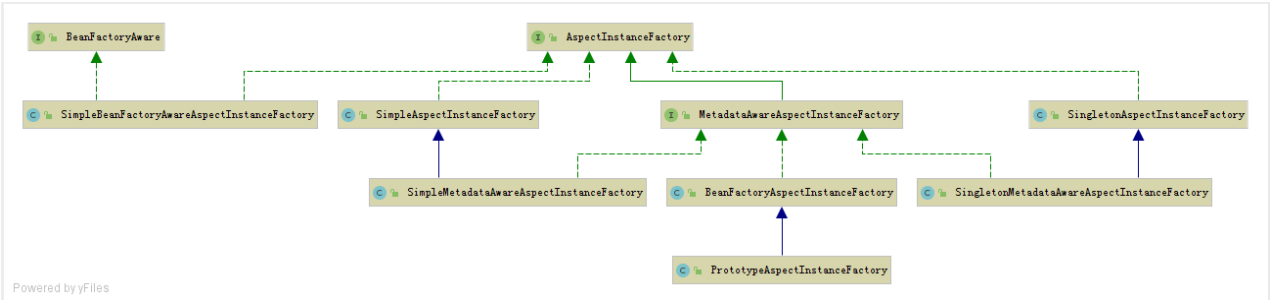


类名	作用概述
AspectJAdvisorFactory	使用AspectJ注解 生成Advisor工厂类
AbstractAspectJAdvisorFactory	AspectJAdvisorFactory的子类。使用AspectJ注解 生成Advisor的工厂类
ReflectiveAspectJAdvisorFactory	AbstractAspectJAdvisorFactory的子类。使用AspectJ注解 生成Advisor的具体实现类。
AspectMetadata	使用AspectJ Aspect注解的切面元数据类。

BeanFactoryAspectJAdvisorsBuilder系列

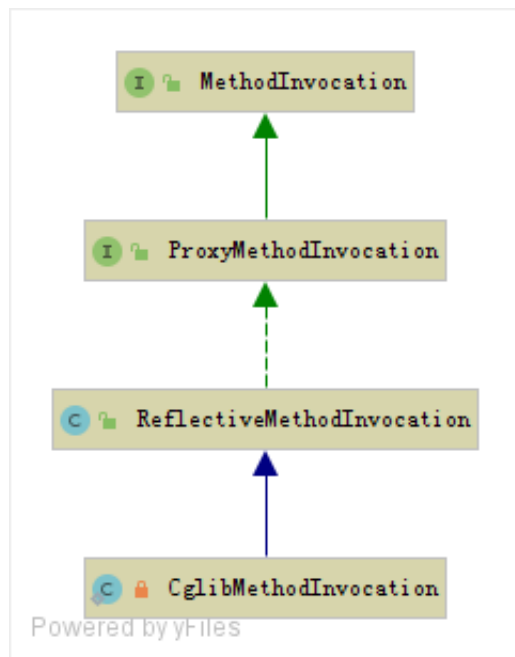
类名	作用概述
BeanFactoryAspectJAdvisorsBuilder	工具类。负责构建Advisor、Advice。SpringAOP核心类

AspectInstanceFactory系列



类名	作用概述
AspectInstanceFactory	Aspect实例工厂类
MetadataAwareAspectInstanceFactory	AspectInstanceFactory的子类。含有Aspect注解元数据 Aspect切面实例工厂类。
BeanFactoryAspectInstanceFactory	MetadataAwareAspectInstanceFactory的子类。持有BeanFactory实例。从BeanFactory中获取Aspect实例。
PrototypeAspectInstanceFactory	BeanFactoryAspectInstanceFactory的子类。获取Prototype类型的Aspect实例。
SimpleMetadataAwareAspectInstanceFactory	MetadataAwareAspectInstanceFactory的实例。在AspectJProxyFactory中有使用。
SingletonMetadataAwareAspectInstanceFactory	MetadataAwareAspectInstanceFactory的子类。继承了SimpleAspectInstanceFactory。单例Aspect实例类。在AspectJProxyFactory中有使用。
SimpleBeanFactoryAwareAspectInstanceFactory	AspectInstanceFactory的子类。实现了BeanFactoryAware接口。和 aop:config 配合使用的类。

ProxyMethodInvocation系列



类名	作用概述
ProxyMethodInvocation	含有代理对象的。MethodInvocation的子类。
ReflectiveMethodInvocation	ProxyMethodInvocation的子类。 AOP拦截的执行入口类 。
CglibMethodInvocation	ReflectiveMethodInvocation的子类。对Cglib反射调用目标方法进行了一点改进。

ClassFilter系列（重点）

2.2 查找BeanDefinitionParser流程分析

根据自定义标签，找到对应的[BeanDefinitionParser](#)，比如[aop:config](#)标签，就对应着ConfigBeanDefinitionParser。

阅读经验分享：

根据自定义标签名称冒号前面的值去直接找NamespaceHandler，然后再根据自定义标签名称冒号后面的值去找BeanDefinitionParser。

2.2.1 找入口

DefaultBeanDefinitionDocumentReader#parseBeanDefinitions 方法的第16行或者23行：

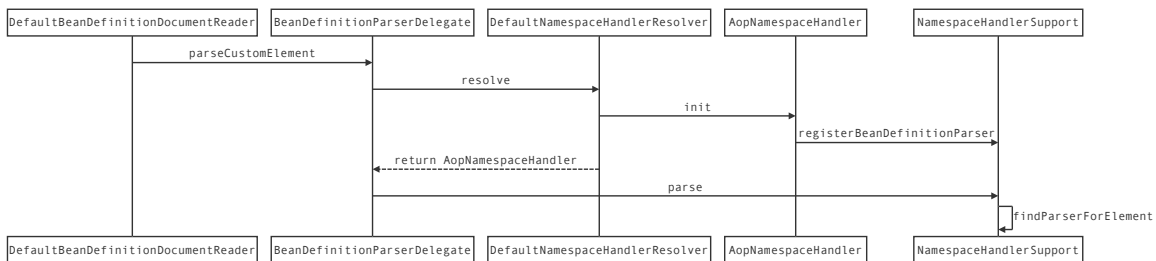
```
protected void parseBeanDefinitions(Element root,
BeanDefinitionParserDelegate delegate) {
    // 加载的Document对象是否使用了Spring默认的XML命名空间（beans命名空间）
    if (delegate.isDefaultNamespace(root)) {
        // 获取Document对象根元素的所有子节点（bean标签、import标签、alias标签和其他自定义
        标签context、aop等）
    }
}
```

```

NodeList nl = root.getChildNodes();
for (int i = 0; i < nl.getLength(); i++) {
    Node node = nl.item(i);
    if (node instanceof Element) {
        Element ele = (Element) node;
        // bean标签、import标签、alias标签，则使用默认解析规则
        if (delegate.isDefaultNamespace(ele)) {
            parseDefaultElement(ele, delegate);
        }
        //像context标签、aop标签、tx标签，则使用用户自定义的解析规则解析元素节点
        else {
            delegate.parseCustomElement(ele);
        }
    }
}
}
else {
    // 如果不是默认的命名空间，则使用用户自定义的解析规则解析元素节点
    delegate.parseCustomElement(root);
}
}

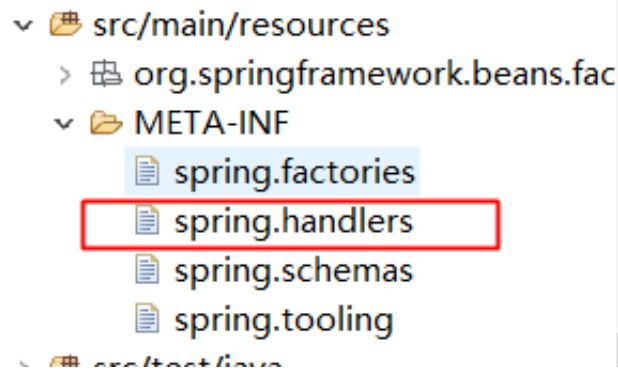
```

2.2.2 流程图



2.2.3 流程相关类的说明

2.2.4 流程解析



```
http://www.springframework.org/schema/c=org.springframework.beans.factory.xml.  
SimpleConstructorNamespaceHandler  
http://www.springframework.org/schema/p=org.springframework.beans.factory.xml.  
SimplePropertyNamespaceHandler  
http://www.springframework.org/schema/util=org.springframework.beans.factory.x  
ml.UtilNamespaceHandler
```

BeanDefinitionParserDelegate#parseCustomElement

```
@Nullable  
public BeanDefinition parseCustomElement(Element ele, @Nullable  
BeanDefinition containingBd) {  
    // 获取命名空间URI (就是获取beans标签的xmlns:aop或者xmlns:context属性的值)  
    // http://www.springframework.org/schema/aop  
    String namespaceUri = getNamespaceURI(ele);  
  
    // 根据不同的命名空间URI,去匹配不同的NamespaceHandler(一个命名空间对应一个  
    NamespaceHandler)  
    // 此处会调用DefaultNamespaceHandlerResolver类的resolve方法  
    // 两步操作: 查找NamespaceHandler、调用NamespaceHandler的init方法进行初始化(针对  
    不同自定义标签注册相应的BeanDefinitionParser)  
    NamespaceHandler handler =  
    this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);  
  
    // 调用匹配到的NamespaceHandler的解析方法  
    return handler.parse(ele, new ParserContext(this.readerContext, this,  
    containingBd));  
}
```

DefaultNamespaceHandlerResolver#resolve

```
public NamespaceHandler resolve(String namespaceUri) {
```

```

// 读取spring所有工程的META-INF/spring.handlers文件,
// 获取namespaceUri和NamespaceHandler的映射关系
Map<String, Object> handlerMappings = getHandlerMappings();
// 获取 指定namespaceUri对应的namespaceHandler
Object handlerOrClassName = handlerMappings.get(namespaceUri);

// META-INF/spring.handlers文件中存储的value都是String类型的类名
String className = (String) handlerOrClassName;
try {
    // 根据类名通过反射获取到NamespaceHandler的Class类对象
    Class<?> handlerClass = ClassUtils.forName(className,
this.classLoader);

    // 实例化NamespaceHandler
    NamespaceHandler namespaceHandler =
        (NamespaceHandler) BeanUtils.instantiateClass(handlerClass);
    // 调用NamespaceHandler类的init方法,
    // 初始化一些专门处理指定标签的BeanDefinitionParsers类
    namespaceHandler.init();
    // 将namespaceUri对应的String类型的类名, 替换为NamespaceHandler对象
    // 下一次再获取的话, 就不会重复创建实例
    handlerMappings.put(namespaceUri, namespaceHandler);
    return namespaceHandler;
}
}

```

AopNamespaceHandler#init()

```

public void init() {
    // In 2.0 XSD as well as in 2.1 XSD.
    // <aop:config></aop:config>对应的BeanDefinitionParser
    registerBeanDefinitionParser("config", new ConfigBeanDefinitionParser());
    registerBeanDefinitionParser("aspectj-autoproxy", new
AspectJAutoProxyBeanDefinitionParser());
    registerBeanDefinitionDecorator("scoped-proxy", new
ScopedProxyBeanDefinitionDecorator());

    // Only in 2.0 XSD: moved to context namespace as of 2.1
    registerBeanDefinitionParser("spring-configured", new
SpringConfiguredBeanDefinitionParser());
}

```

至此，找到了解析对应的BeanDefinitionParser

2.3 执行BeanDefinitionParser流程分析

解析标签和，最终解析10个类对应的BeanDefinition。

- 产生代理对象的类对应的BeanDefinition（一种）：[AspectJAwareAdvisorAutoProxyCreator](#)
- 通知BeanDefinition（五种）：[AspectJMethodBeforeAdvice](#)、[AspectJAfterAdvice](#)、[AspectJAfterReturningAdvice](#)、[AspectJAfterThrowingAdvice](#)、[AspectJAroundAdvice](#)
- 通知器BeanDefinition（一种）：[DefaultBeanFactoryPointcutAdvisor](#)、[AspectJPointcutAdvisor](#)
- 切入点BeanDefinition（一种）：[AspectJExpressionPointcut](#)
- 自定义增强功能（BeanDefinition，是由ioc流程确定的BeanDefinition，不是我们这个环节确定的）

* 类的实例

* 方法（一个方法对应一个增强功能） 反射调用 `[method]().invoke([bean>(),args);`

- 用于产生自定义增强类实例的类对应的BeanDefinition [实例工厂](#)去产生自定义功能对应的类的实例

[SimpleBeanFactoryAwareAspectInstanceFactory](#)----增强类的实例。

- 用于调用自定义增强类方法对应的BeanDefinition 使用一个封装增强方法的
[BeanDefinition](#)去封装Method方法

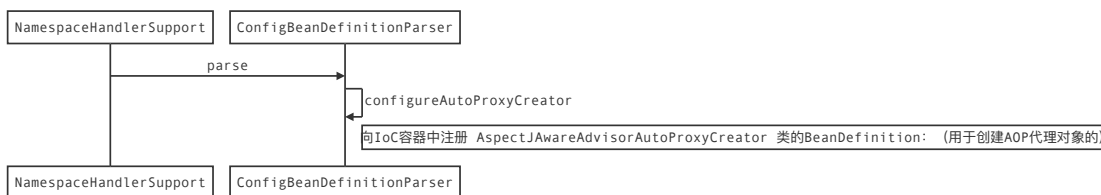
[MethodLocatingFactoryBean](#)---Method对象

2.3.1 找入口

[NamespaceHandlerSupport](#)类的 `parse` 方法第6行代码：

```
public BeanDefinition parse(Element element, ParserContext parserContext) {  
    // NamespaceHandler里面初始化了大量的BeanDefinitionParser来分别处理不同的自定义标  
    签  
    // 从指定的NamespaceHandler中，匹配到指定的BeanDefinitionParser  
    BeanDefinitionParser parser = findParserForElement(element, parserContext);  
    // 调用指定自定义标签的解析器，完成具体解析工作  
    return (parser != null ? parser.parse(element, parserContext) : null);  
}
```


2.3.2 流程图



2.3.3 流程相关类的说明

2.3.4 流程解析

ConfigBeanDefinitionParser#parse

```
public BeanDefinition parse(Element element, ParserContext parserContext) {

    // 向IoC容器中注册 AspectJAwareAdvisorAutoProxyCreator 类的BeanDefinition: (用于创建AOP代理对象的)
    // BeanPostProcessor可以对实例化之后的bean进行一些操作
    // AspectJAwareAdvisorAutoProxyCreator 实现了BeanPostProcessor接口,可以对目标对象实例化之后, 创建对应的代理对象
    configureAutoProxyCreator(parserContext, element);

    // 获取<aop:config>标签的子标签<aop:aspect>、<aop:advisor> 、<aop:pointcut>
    List<Element> childElts = DomUtils.getChildElements(element);
    for (Element elt: childElts) {
        // 获取子标签的节点名称或者叫元素名称
        String localName = parserContext.getDelegate().getLocalName(elt);
        if (POINTCUT.equals(localName)) {
            // 解析<aop:pointcut>标签
            // 产生一个AspectJExpressionPointcut的BeanDefinition对象, 并注册
            parsePointcut(elt, parserContext);
        }
        else if (ADVISOR.equals(localName)) {
            // 解析<aop:advisor>标签
            // 产生一个DefaultBeanFactoryPointcutAdvisor的BeanDefinition对象, 并注册
            parseAdvisor(elt, parserContext);
        }
        else if (ASPECT.equals(localName)) {
            // 解析<aop:aspect>标签
            // 产生了很多BeanDefinition对象
            // aop:after等标签对应5个BeanDefinition对象
            // aop:after标签的method属性对应1个BeanDefinition对象
            // 最终的AspectJPointcutAdvisor BeanDefinition类

            parseAspect(elt, parserContext);
        }
    }
}
```

```

    }
}

return null;
}

```

ConfigBeanDefinitionParser#parsePointcut

```

private AbstractBeanDefinition parsePointcut(Element pointcutElement,
ParserContext parserContext) {

    // 此处创建一个 AspectJExpressionPointcut 类对应的BeanDefinition对象, 处理
    pointcut
    pointcutDefinition = createPointcutDefinition(expression);

    return pointcutDefinition;
}

protected AbstractBeanDefinition createPointcutDefinition(String expression) {
    RootBeanDefinition beanDefinition =
        new RootBeanDefinition(AspectJExpressionPointcut.class);

    // 设置切入点表达式
    beanDefinition.getPropertyValues().add(EXPRESSION, expression);
    return beanDefinition;
}

```

spring-aop.xml配置文件

```

<aop:config>
    <aop:aspect ref="myAdvice">
        <aop:after method="after"
            pointcut="execution(* *..*ServiceImpl.*(..))" />

        <aop:before method="before"
            pointcut="execution(* *..*ServiceImpl.*(..))" />
    </aop:aspect>
</aop:config>

```

ConfigBeanDefinitionParser#parseAspect

```

private void parseAspect(Element aspectElement, ParserContext parserContext) {

```

```

try {

    // 获取<aop:aspect>标签的所有子标签
    NodeList nodeList = aspectElement.getChildNodes();
    boolean adviceFoundAlready = false;
    for (int i = 0; i < nodeList.getLength(); i++) {
        Node node = nodeList.item(i);
        // 判断是否是<aop:before>、<aop:after>、<aop:after-returning>、
        // <aop:after-throwing method="">、<aop:around method="">这五个标签
        if (isAdviceNode(node, parserContext)) {
            // 解析<aop:before>等五个子标签
            // 方法主要做了三件事：
            // 1、根据织入方式（before、after这些）创建RootBeanDefinition,
            // 名为adviceDef即advice定义
            // 2、将上一步创建的RootBeanDefinition写入一个新的
RootBeanDefinition,
            // 构造一个新的对象，名为advisorDefinition，即advisor定义
            // 3、将advisorDefinition注册到DefaultListableBeanFactory中
            AbstractBeanDefinition advisorDefinition =
                parseAdvice(aspectName, i, aspectElement, (Element) node,
                    parserContext, beanDefinitions,
beanReferences);

        }
    }

    // 得到所有<aop:aspect>下的<aop:pointcut>子标签
    List<Element> pointcuts =
        DomUtils.getChildElementsByTagName(aspectElement, POINTCUT);
    for (Element pointcutElement : pointcuts) {
        // 解析<aop:pointcut>子标签
        parsePointcut(pointcutElement, parserContext);
    }

}

}

```

```

public class MyAdvice {

    // 演示前置通知
    public void before() {
        System.out.println("前置通知...");
    }
}

```

```

}

// 演示后置通知
public void afterReturning() {
    System.out.println("后置通知...");
}

// 演示最终通知
public void after() {
    System.out.println("最终通知...");
}

// 演示异常抛出通知
public void afterThrowing() {
    System.out.println("异常抛出通知...");
}

/**
 * 环绕通知 场景使用：事务管理
 *
 * @param joinPoint
 * @throws Throwable
 */
public void around(ProceedingJoinPoint joinPoint) {
    System.out.println("环绕通知---前置通知");
    try {
        // 调用目标对象的方法
        joinPoint.proceed();
        System.out.println("环绕通知---后置通知");
    } catch (Throwable e) { // 相当于实现异常通知
        System.out.println("环绕通知---异常抛出配置");
        e.printStackTrace();
    } finally {
        System.out.println("环绕通知---最终通知");
    }
}
}

```

ConfigBeanDefinitionParser#parseAdvice (产生了8个BeanDefinition)

```

private AbstractBeanDefinition parseAdvice(
    String aspectName, int order, Element aspectElement,
    Element adviceElement, ParserContext parserContext,
    List<BeanDefinition> beanDefinitions, List<BeanReference> beanReferences)
{

```

```

try {

    // create the method factory bean
    // 创建方法工厂Bean的BeanDefinition对象：用于获取Advice增强类的Method对象
    RootBeanDefinition methodDefinition =
        new RootBeanDefinition(MethodLocatingFactoryBean.class);
    // 设置MethodLocatingFactoryBean的targetBeanName为advice类的引用名称
    methodDefinition.getPropertyValues().add("targetBeanName", aspectName);
    // 设置MethodLocatingFactoryBean的methodName为<aop:after>标签的method属性
    // 值（也就是advice方法名称）
    methodDefinition.getPropertyValues().add("methodName",

adviceElement.getAttribute("method"));

    // create instance factory definition
    // 创建实例工厂BeanDefinition：用于创建增强类的实例
    RootBeanDefinition aspectFactoryDef =
        new
RootBeanDefinition(SimpleBeanFactoryAwareAspectInstanceFactory.class);
    // 设置SimpleBeanFactoryAwareAspectInstanceFactory的aspectBeanName为
advice类的引用名称
    aspectFactoryDef.getPropertyValues().add("aspectBeanName", aspectName);

    //以上的两个BeanDefinition的作用主要是通过反射调用Advice对象的指定方法
    // method.invoke(obj,args)

    // register the pointcut
    // 通知增强类的BeanDefinition对象（核心）
    AbstractBeanDefinition adviceDef = createAdviceDefinition(
        adviceElement, parserContext, aspectName, order, methodDefinition,
        aspectFactoryDef, beanDefinitions, beanReferences);

    // configure the advisor
    // 通知器类的BeanDefinition对象
    RootBeanDefinition advisorDefinition =
        new RootBeanDefinition(AspectJPointcutAdvisor.class);

    // 给通知器类设置Advice对象属性值
    advisorDefinition.getConstructorArgumentValues()
        .addGenericArgumentValue(adviceDef);

    if (aspectElement.hasAttribute(ORDER_PROPERTY)) {
        advisorDefinition.getPropertyValues().add(
            ORDER_PROPERTY, aspectElement.getAttribute(ORDER_PROPERTY));
    }

    return advisorDefinition;
}

```

```
}  
}
```

ConfigBeanDefinitionParser#createAdviceDefinition

```
private AbstractBeanDefinition createAdviceDefinition(  
    Element adviceElement, ParserContext parserContext, String aspectName,  
    int order,  
    RootBeanDefinition methodDef, RootBeanDefinition aspectFactoryDef,  
    List<BeanDefinition> beanDefinitions, List<BeanReference> beanReferences)  
{  
  
    // 根据通知类型的不同, 分别创建对应的BeanDefinition对象(可以去看看getAdviceClass方法)  
    RootBeanDefinition adviceDefinition =  
        new RootBeanDefinition(getAdviceClass(adviceElement,  
parserContext));  
  
    // 设置构造参数  
    ConstructorArgumentValues cav =  
adviceDefinition.getConstructorArgumentValues();  
    // 设置第一个构造参数: 方法工厂对象的BeanDefinition  
    cav.addIndexedArgumentValue(METHOD_INDEX, methodDef);  
  
    // 解析<aop:before>、<aop:after>、<aop:after-returning>标签中的pointcut或者  
pointcut-ref属性  
    Object pointcut = parsePointcutProperty(adviceElement, parserContext);  
    // 设置第二个构造参数: 切入点BeanDefinition  
    if (pointcut instanceof BeanDefinition) {  
        cav.addIndexedArgumentValue(POINTCUT_INDEX, pointcut);  
        beanDefinitions.add((BeanDefinition) pointcut);  
    }  
    else if (pointcut instanceof String) {  
        RuntimeBeanReference pointcutRef = new RuntimeBeanReference((String)  
pointcut);  
        cav.addIndexedArgumentValue(POINTCUT_INDEX, pointcutRef);  
        beanReferences.add(pointcutRef);  
    }  
    // 设置第三个构造参数: 实例工厂BeanDefinition  
    cav.addIndexedArgumentValue(ASPECT_INSTANCE_FACTORY_INDEX,  
aspectFactoryDef);  
  
    return adviceDefinition;  
}  
  
private Class<?> getAdviceClass(Element adviceElement, ParserContext  
parserContext) {
```

```

        // 获取标签名称, 比如aop:before标签对应的标签名是before
        String elementName =
parserContext.getDelegate().getLocalName(adviceElement);
        // 处理<aop:before>标签
        if (BEFORE.equals(elementName)) {
            return AspectJMethodBeforeAdvice.class;
        }
        // 处理<aop:after>标签
        else if (AFTER.equals(elementName)) {
            return AspectJAfterAdvice.class;
        }
        // 处理<aop:after-returning>标签
        else if (AFTER_RETURNING_ELEMENT.equals(elementName)) {
            return AspectJAfterReturningAdvice.class;
        }
        // 处理<aop:after-throwing>标签
        else if (AFTER_THROWING_ELEMENT.equals(elementName)) {
            return AspectJAfterThrowingAdvice.class;
        }
        // 处理<aop:around>标签
        else if (AROUND.equals(elementName)) {
            return AspectJAroundAdvice.class;
        }
    }
}

```

2.4 产生AOP代理流程分析

2.4.1 AspectJAwareAdvisorAutoProxyCreator的继承体系

```

|--BeanPostProcessor (初始化)
    postProcessBeforeInitialization---初始化之前调用
    postProcessAfterInitialization---初始化之后调用

|--InstantiationAwareBeanPostProcessor (实例化)
    postProcessBeforeInstantiation---实例化之前调用
    postProcessAfterInstantiation---实例化之后调用
    postProcessPropertyValues---后置处理属性值

|---SmartInstantiationAwareBeanPostProcessor
    predictBeanType
    determineCandidateConstructors
    getEarlyBeanReference

```

```
|-----AbstractAutoProxyCreator
    postProcessBeforeInitialization
    postProcessAfterInitialization----AOP功能入口
    postProcessBeforeInstantiation
    postProcessAfterInstantiation
    postProcessPropertyValues
    ...
|-----AbstractAdvisorAutoProxyCreator
    getAdvicesAndAdvisorsForBean
    findEligibleAdvisors
    findCandidateAdvisors
    findAdvisorsThatCanApply

|-----AspectJAwareAdvisorAutoProxyCreator
    extendAdvisors
    sortAdvisors
```

2.4.2 找入口

AbstractAutoProxyCreator类的 `postProcessAfterInitialization` 方法第6行代码:

```
public Object postProcessAfterInitialization(@Nullable Object bean, String
beanName) throws BeansException {
    if (bean != null) {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);
        if (!this.earlyProxyReferences.contains(cacheKey)) {
            // 使用动态代理技术, 产生代理对象
            return wrapIfNecessary(bean, beanName, cacheKey);
        }
    }
    return bean;
}
```

2.4.2 流程图

2.5 代理对象执行流程

主要去针对jdk产生的动态代理对象进行分析, 其实就是去分析InvocationHandler的invoke方法

入口: JdkDynamicAopProxy#invoke方法

