Spring 框架之 AOP 原理剖析

前言

AOP（Aspect Oriented Programming）面向切面编程是 Spring 框架最核心的组件之一，它通过对程序结构的另一种考虑，补充了 OOP（Object-Oriented Programming）面向对象编程。在 OOP 中模块化的关键单元是类，而在 AOP 中，模块化单元是切面。也就是说 AOP 关注的不再是类，而是一系列类里面需要共同能力的行为。

本文内容主要包括：

- 讲解 OOP 与 AOP 的简单对比，以及 AOP 的基础名词，比如面试中经常会被问到的 point cut、advice、join point 等。
- 讲解面试经常会被问到的 JDK 动态代理和 CGLIB 代理原理以及区别。
- 讲解 Spring 框架中基于 Schema 的 AOP 实现原理。
- 讲解 Spring 框架中如何基于 AOP 实现的事务管理。

AOP 基础概念

我们知道在 OOP 中模块化的关键单元是类，类封装了一类对象的行为和状态，当多个类有共同的属性和行为时候我们把这些共同的东西封装为一个基类，然后多个类可以通过继承基类的方式来复用这些共同的东西，如果子类需要定制基类行为则可以使用多态。OOP 中使用类来提供封装，继承，多态三个特性。

当我们需要在多个不相关的类的某些已有的行为里面添加一个共同的非业务逻辑时候，比如我们需要统计一些业务方法的执行耗时时候，以往做法需要在统计耗时的行为里面写入计算耗时的代码，在 OOP 里面这种不涉及业务的散落在多个类的行为里面的代码叫做横切（Cross-cutting）代码，OOP 中这种方式缺点一是业务逻辑行为受到了计算耗时代码干扰（业务逻辑行为应该只专注业务），缺点二是计算耗时的代码不能被复用。

而在 AOP 中模块化单元是切面（Aspect），它将那些影响多个类的共同行为封装到可重用的模块中，然后你就可以决定在什么时候对哪些类的哪些行为执行进行拦截（切点），并使用封装好的可重用模块里面的行为（通知）对其拦截的业务行为进行功能增强，而不需要修改业务模块的代码，切面就是对此的一个抽象描述。

AOP 中有以下基础概念：

- Join point（连接点）：程序执行期间的某一个点，例如执行方法或处理异常时候的点。在 Spring AOP 中，连接点总是表示方法的执行。

- Advice（通知）：通知是指一个切面在特定的连接点要做的事情。通知分为方法执行前通知，方法执行后通知，环绕通知等。许多 AOP 框架（包括 Spring）都将通知建模为拦截器，在连接点周围维护一系列拦截器（形成拦截器链），对连接点的方法进行增强。
- Pointcut（切点）:一个匹配连接点（Join point）的谓词表达式。通知（Advice）与切点表达式关联，并在切点匹配的任何连接点（Join point）（例如，执行具有特定名称的方法）上运行。切点是匹配连接点（Join point）的表达式的概念，是AOP的核心，并且 Spring 默认使用 AspectJ 作为切入点表达式语言。
- Aspect（切面）：它是一个跨越多个类的模块化的关注点，它是通知（Advice）和切点（Pointcut）合起来的抽象，它定义了一个切点（Pointcut）用来匹配连接点（Join point），也就是需要对需要拦截的那些方法进行定义；它定义了一系列的通知（Advice）用来对拦截到的方法进行增强；
- Target object（目标对象）：被一个或者多个切面（Aspect）通知的对象，也就是需要被 AOP 进行拦截对方法进行增强（使用通知）的对象，也称为被通知的对象。由于在 AOP 里面使用运行时代理，所以目标对象一直是被代理的对象。
- AOP proxy（AOP 代理）：为了实现切面（Aspect）功能使用 AOP 框架创建一个对象，在 Spring 框架里面一个 AOP 代理要么指 JDK 动态代理，要么指 CgLIB 代理。
- Weaving（织入）：是将切面应用到目标对象的过程，这个过程可以是在编译时（例如使用 AspectJ 编译器），类加载时，运行时完成。Spring AOP 和其它纯 Java AOP 框架一样，是在运行时执行植入。
- Advisor：这个概念是从 Spring 1.2的 AOP 支持中提出的，一个 Advisor 相当于一个小型的切面，不同的是它只有一个通知（Advice），Advisor 在事务管理里面会经常遇到，这个后面会讲到。

相比 OOP，AOP 有以下优点：

- 业务代码更加简洁，例如当需要在业务行为前后做一些事情时候，只需要在该行为前后配置切面进行处理，无须修改业务行为代码。
- 切面逻辑封装性好，并且可以被复用，例如我们可以把打日志的逻辑封装为一个切面，那么我们就可以在多个相关或者不相关的类的多个方法上配置该切面。

JDK 动态代理和 CGLIB 代理原理以及区别

在 Spring 中 AOP 代理使用 JDK 动态代理和 CGLIB 代理来实现，默认如果目标对象是接口，则使用 JDK 动态代理，否则使用 CGLIB 来生成代理类，本节就简单来介绍这两种代理的原理和区别。

JDK 动态代理

由于 JDK 代理是对接口进行代理，所以首先写一个接口类：

```
public interface UserServiceBo {
```

```
    public  int add();
}
```

然后实现该接口如下：

```
public class UserServiceImpl implements UserServiceBo {

    @Override
    public int add() {
        System.out.println("--------------------add--------------
--------");
        return 0;
    }
}
```

JDK 动态代理是需要实现 InvocationHandler 接口，这里创建一个 InvocationHandler 的实现类：

```
public class MyInvocationHandler implements InvocationHandler {

    private Object target;

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

    @Override
    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
        //(1)
        System.out.println("-----------------begin
"+method.getName()+"-----------------");
        //(2)
        Object result = method.invoke(target, args);
        //(3)
        System.out.println("-----------------end
"+method.getName()+"-----------------");
        return result;
    }

    public Object getProxy(){
        //(4)
        return
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoad
er(), target.getClass().getInterfaces(), this);
    }

}
```

建立一个测试类：

```
public static void main(String[] args) {

  //(5)打开这个开关，可以把生成的代理类保存到磁盘
 System.getProperties().put("sun.misc.ProxyGenerator.saveGenerate
dFiles", "true");
  //(6)创建目标对象（被代理对象）
  UserServiceBo service = new UserServiceImpl();
  //(7)创建一个InvocationHandler实例，并传递被代理对象
  MyInvocationHandler handler = new MyInvocationHandler(service);
  //(8)生成代理类
  UserServiceBo proxy = (UserServiceBo) handler.getProxy();
  proxy.add();
}
```

其中代码（6）创建了一个 UserServiceImpl 的实例对象，这个对象就是要被代理的目标对象。

代码（7）创建了一个 InvocationHandler 实例，并传递被代理目标对象 service 给内部变量 target。

代码（8）调用 MyInvocationHandler 的 getProxy 方法使用 JDK 生成一个代理对象。

代码（5）设置系统属性变量 sun.misc.ProxyGenerator.saveGeneratedFiles 为 true，这是为了让代码（8）生成的代理对象的字节码文件保存到磁盘。

运行上面代码会在项目的 com.sun.proxy 下面会生成 $Proxy0.class 类，经反编译后核心 Java 代码如下：

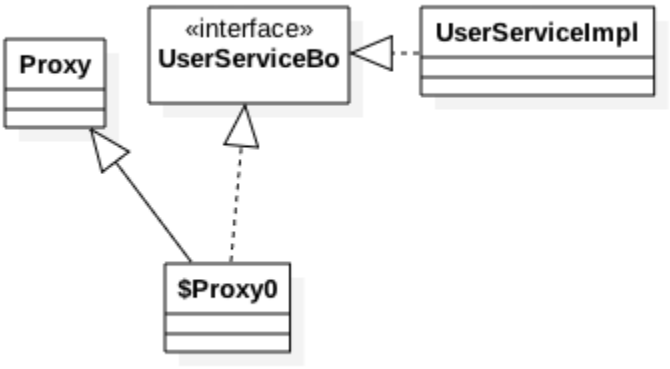
```
package com.sun.proxy;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;
import proxy.JDK.UserServiceBo;

public final class $Proxy0
  extends Proxy
  implements UserServiceBo
{
  private static Method m1;
  private static Method m3;
  private static Method m0;
  private static Method m2;

  public $Proxy0(InvocationHandler paramInvocationHandler)
```

```
    {
      super(paramInvocationHandler);
    }

    public final int add()
    {
      try
      {
      //（9）第一个参数是代理类本身，第二个是实现类的方法，第三个是参数
        return h.invoke(this, m3, null);
      }
      catch (Error|RuntimeException localError)
      {
        throw localError;
      }
      catch (Throwable localThrowable)
      {
        throw new UndeclaredThrowableException(localThrowable);
      }
    }

   ...

    static
    {
      try
      {

        m3 =
Class.forName("proxy.JDK.UserServiceBo").getMethod("add", new
Class[0]);
      ...
        return;
      }
      catch (NoSuchMethodException localNoSuchMethodException)
      ...
    }
}
```

代理类 `$Proxy0` 中代码（9）中的 h 就是 main 函数里面创建的 MyInvocationHandler 的实例，h.invoke(this, m3, null) 实际就是调用的 MyInvocationHandler 的 invoke 方法，而后者则是委托给被代理对象进行执行，这里可以对目标对象方法进行拦截，然后对其功能进行增强。

另外代码（8）生成的 proxy 对象实际就是 `$Proxy0` 的一个实例，当调用 proxy.add() 时候，实际是调用的代理类 `$Proxy0` 的 add 方法，后者内部则委托给 MyInvocationHandler 的 invoke 方法，invoke 方法内部有调用了目标对象 service 的 add 方法。

那么接口（ UserServiceBo ）、目标对象（ 被代理对象 UserServiceImpl ）、代理对象（ $Proxy0 ）三者具体关系可以使用下图表示：



可知 JDK 动态代理是对接口进行的代理；代理类实现了接口，并继承了 Proxy 类；目标对象与代理对象没有什么直接关系，只是它们都实现了接口，并且代理对象执行方法时候内部最终是委托目标对象执行具体的方法。

CGLIB 动态代理

相比 JDK 动态代理对接口进行代理，CGLIB 则是对实现类进行代理，这意味着无论目标对象是否有接口，都可以使用 CGLIB 进行代理。

下面结合一个使用 CGLIB 对实现类进行动态代理的简单代码来讲解。

使用 CGLIB 进行代理需要实现 MethodInterceptor，创建一个方法拦截器 CglibProxy 类：

```
public class CglibProxy implements MethodInterceptor {
    //(10)
    private Enhancer enhancer = new Enhancer();
    //(11)
    public Object getProxy(Class clazz) {
        //（12）设置被代理类的Class对象
        enhancer.setSuperclass(clazz);
        //(13)设置拦截器回调
        enhancer.setCallback( this);
        return enhancer.create();
    }


    @Override
    public Object intercept(Object obj, Method method, Object[]
args, MethodProxy proxy) throws Throwable {

System.out.println(obj.getClass().getName()+"."+method.getName())
;
        Object result = proxy.invokeSuper(obj, args);
```

```
        return result;
    }
}


public void testCglibProxy() {

//（14）生成代理类到本地
 System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY,
 "/Users/zhuizhumengxiang/Downloads");
    //(15)生成目标对象
    UserServiceImpl service = new UserServiceImpl();
    //（16）创建CglibProxy对象
    CglibProxy cp = new CglibProxy();
    //(17)生成代理类
    UserServiceBo proxy = (UserServiceBo)
cp.getProxy(service.getClass());
    proxy.add();
}
```

执行上面代码会在 /Users/zhuizhumengxiang/Downloads 目录生成代理类 UserServiceImpl$$EnhancerByCGLIB$$d0bce05a.class 文件，反编译后部分代码如下：

```
public class UserServiceImpl$$EnhancerByCGLIB$$d0bce05a extends
UserServiceImpl
  implements Factory
{
static void CGLIB$STATICHOOK1()
  {
    //(18)空参数
    CGLIB$emptyArgs = new Object[0];

    //(19)获取UserServiceImpl的add方法列表
    Method[] tmp191_188 = ReflectUtils.findMethods(new String[] {
"add", "()I" }, (localClass2 =
Class.forName("zlx.cglib.zlx.UserServiceImpl")).getDeclaredMethod
s());
    CGLIB$add$0$Method = tmp191_188[0];

    //(20)创建CGLIB$add$0，根据创建一个MethodProxy
    CGLIB$add$0$Proxy = MethodProxy.create(localClass2,
localClass1, "()I", "add", "CGLIB$add$0");

  }
  static
  {
    CGLIB$STATICHOOK1();
```

```
    }
    //(21)
    final int CGLIB$add$0()
    {
      return super.add();
    }
    //(22)
    public final int add()
    {
      ...
      //(23)获取拦截器，这里为CglibProxy的实例
      MethodInterceptor tmp17_14 = this.CGLIB$CALLBACK_0;
      if (tmp17_14 != null)
      { //(24)调用拦截器的intercept方法
        Object tmp36_31 = tmp17_14.intercept(this,
CGLIB$add$0$Method, CGLIB$emptyArgs, CGLIB$add$0$Proxy);
        tmp36_31;
        return tmp36_31 == null ? 0 :
((Number)tmp36_31).intValue();
      }
      return super.add();
    }
  }
```

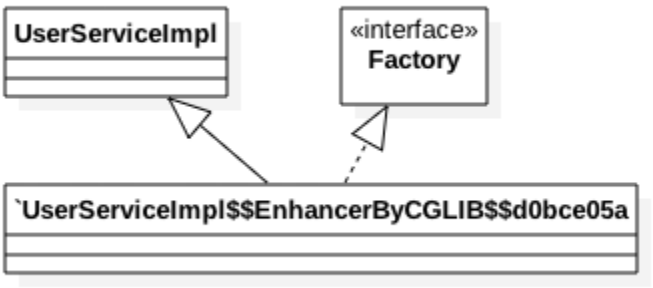
代码（18）创建了一个 `CGLIB$emptyArgs` ，因为 add 方法是无入参的，所以这里创建的 Object 对象个数为0，这对象是 CglibProxy 拦截器的 intercept 的第三个参数。

代码（19）获取 UserServiceImpl 的 add 方法列表，并把唯一方法赋值给 `CGLIB$add$0$Method` ，这个对象是 CglibProxy 拦截器的 intercept 第二个参数。

代码（20）创建了一个 MethodProxy 对象，当调用 MethodProxy 对象的 invokeSuper 方法时候会直接调用代理对象的 `CGLIB$add$0` 方法，也就是直接调用父类 UserServiceImpl 的 add 方法，这避免了一次反射调用，创建的 MethodProxy 对象是 CglibProxy 拦截器的 intercept 的第四个参数。

代码（22）是代理类的 add 方法，代码（17）生成的代理对象 proxy 就是 `UserServiceImpl$$EnhancerByCGLIB$$d0bce05a` 的一个实例，当调用 proxy.add() 方法时候就是调用的代码（22），从代码（22）可知内部调用了拦截器 CglibProxy 的 intercept 方法，可知 intercept 的第一个参数就是代理对象本身。

那么接口（UserServiceBo）、目标对象（被代理对象 UserServiceImpl），代理对象（`UserServiceImpl$$EnhancerByCGLIB$$d0bce05a` ）三者具体关系可以使用下图表示：



可知接口和代理对象没有啥关系，代理对象是继承了目标对象和实现了 Factory 接口。

总结

JDK 动态代理机制只能对接口进行代理，其原理是动态生成一个代理类，这个代理类实现了目标对象的接口，目标对象和代理类都实现了接口，但是目标对象和代理类的 Class 对象是不一样的，所以两者是没法相互赋值的。

CGLIB 是对目标对象本身进行代理，所以无论目标对象是否有接口，都可以对目标对象进行代理，其原理是使用字节码生成工具在内存生成一个继承目标对象的代理类，然后创建代理对象实例。由于代理类的父类是目标对象，所以代理类是可以赋值给目标对象的，自然如果目标对象有接口，代理对象也是可以赋值给接口的。

CGLIB 动态代理中生成的代理类的字节码相比 JDK 来说更加复杂。

JDK 使用反射机制调用目标类的方法，CGLIB 则使用类似索引的方式直接调用目标类方法，所以 JDK 动态代理生成代理类的速度相比 CGLIB 要快一些，但是运行速度比 CGLIB 低一些，并且 JDK 代理只能对有接口的目标对象进行代理。

Spring 框架中基于 Schema 的 AOP 实现原理

Spring 提供了两种方式对 AOP 进行支持：基于 Schema 的 AOP，基于注解的 AOP。

基于 Schema 的 AOP 允许您基于 XML 的格式配置切面功能，Spring 2.0 提供了新的“aop”命名空间标记来定义切面的支持，基于注解的 AOP 则允许您使用 @Aspect 风格来配置切面。

本文就来先讲讲基于 Schema 的 AOP 的实现原理。

AOP 简单使用

一个切面实际上是一个被定义在 Spring application context 里面的一个正常的 Java 对象，配置切面对目标对象进行增强时候，一般使用下面配置格式：

```
    <!--(1) -->
    <bean id="helloService"
class="zlx.test.aop.HelloServiceBoImpl" />

    <!--(2) -->
    <bean id="myAspect" class="zlx.test.aop.MyAspect" />

    <aop:config>
        <!--(3) -->
        <aop:pointcut id="pointcut"
            expression="execution(* *..*BoImpl.sayHello(..))"/>
        <!--(4) -->
        <aop:aspect ref="myAspect">
            <!--(4.1) -->
            <aop:before pointcut-ref="pointcut"
method="beforeAdvice" />
            <!--(4.2) -->
            <aop:after pointcut="execution(*
*..*BoImpl.sayHello(..))"
                method="afterAdvice" />
            <!--(4.3) -->
             <aop:after-returning
                pointcut="execution(*
*..*BoImpl.sayHelloAfterReturn(..))"
method="afterReturningAdvice"
                arg-names="content" returning="content" />
            <!--(4.4) -->
            <aop:after-throwing pointcut="execution(*
*..*BoImpl.sayHelloThrowExecption(..))"
                method="afterThrowingAdvice" arg-names="e"
throwing="e" />
            <!--(4.5) -->
            <aop:around pointcut="execution(*
*..*BoImpl.sayHelloAround(..))"
                method="aroundAdvice" />
        </aop:aspect>
    </aop:config>
    <!--(5) -->
    <bean id = "tracingInterceptor"
class="zlx.test.aop.TracingInterceptor"/>

    <!--(6) -->
    <aop:config>
        <aop:pointcut id="pointcutForadVisor"
expression="execution(* *..*BoImpl.sayHelloAdvisor(..))" />
        <aop:advisor pointcut-ref="pointcutForadVisor" advice-
ref="tracingInterceptor" />
    </aop:config>
```

代码（1）创建一个要被 AOP 进行功能增强的目标对象（Target object），HelloServiceBoImpl 的代码如下：

```
public class HelloServiceBoImpl implements HelloServiceBo{

    @Override
    public void sayHello(String content) {
        System.out.println("sayHello:" + content);
    }

    @Override
    public String sayHelloAround(String content) {
        System.out.println(" sayHelloAround:" + content);
        return content;
    }

    @Override
    public String sayHelloAfterReturn(String content) {
        System.out.println("sayHelloAround:" + content);
        return content;
    }

    @Override
    public void sayHelloThrowExecption() {
        System.out.println("sayHelloThrowExecption");
        throw new RuntimeException("hello Im an exception");
    }
}


public interface HelloServiceBo {

    public void sayHello(String content);

    public String sayHelloAround(String content);

    public String sayHelloAfterReturn(String content);

    public void sayHelloThrowExecption();

}
```

代码（2）实质是定义了切面要使用的一系列的通知方法（Advice），用来对目标对象（Target object）的方法进行增强，MyAspect 的代码如下：

```
public class MyAspect {
    public void beforeAdvice(String content) {
        System.out.println("---before advice "+ "---");
    }
```

```
    public void afterAdvice(JoinPoint jp) {
        System.out.println("---after advice " + jp.getArgs()
[0].toString()+"---");
    }

    public Object afterReturningAdvice(Object value) {
        System.out.println("---afterReturning advice " + value+"-
--");
        return value + " ha";
    }

    public void afterThrowingAdvice(Exception e) {
        System.out.println("---after throwing advice exception:"
+ e+"---");
    }

    public Object aroundAdvice(ProceedingJoinPoint pjp) throws
Throwable {

        Object[] obj = pjp.getArgs();
        String content = (String) obj[0];
        System.out.println("---before sayHelloAround execute---
");
        String  retVal = (String) pjp.proceed();
        System.out.println("---after sayHelloAround execute---");

        return retVal+ " suffix";
    }
}
```

代码（3）定义了一个切点（pointcut），这里是对满足 `* *..*BoImpl` 表达式的类里面的方法名称为 sayHello 的方法进行拦截。

代码（4）定义一个切面，一个切面中可以定义多个拦击器，其中（4.1）定义了一个前置拦截器，这个拦截器对满足代码（3）中定义的切点的连接点（方法）进行拦截，并使用 MyAspect 中定义的通知方法 beforeAdvice 进行功能增强。其中（4.2）定义了一个后置拦截器（finally），对满足 `execution(* *..*BoImpl.sayHello(..))` 条件的连接点方法使用 MyAspect 中的通知方法 afterAdvice 进行功能增强。其中（4.3）定义了一个后置拦截器，对满足 `execution(* *..*BoImpl.sayHelloAfterReturn(..))` 条件的连接点方法使用 MyAspect 中的通知方法 afterReturningAdvice 进行功能增强，这个后置连接器与（4.2）不同在于如果被拦截方法抛出了异常，则这个拦截器不会被执行，而（4.2)的拦截器一直会被执行。其中（4.4）定义了一个当被拦截方法抛出异常后对异常进行拦截的拦截器，具体拦截哪些方法由 `execution(* *..*BoImpl.sayHelloThrowExecption(..))` 来决定，具体的通知方法是 MyAspect 中的 afterThrowingAdvice 方法。其中（4.5）对满足 `execution(* *..*BoImpl.sayHelloAround(..))` 条件的连接点方法使用 MyAspect 中的通知方法 aroundAdvice 进行增强。

代码（5）创建一个方法拦截器，它是一个通知，代码如下：

```
class TracingInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation i) throws Throwable {
        System.out
                .println("---method " + i.getMethod() + " is
called on " + i.getThis() + " with args " + i.getArguments()+"---
");
        Object ret = i.proceed();
        System.out.println("---method " + i.getMethod() + "
returns " + ret+"---");
        return ret;
    }
}
```

代码（6）创建了一个新的 aop:config 标签，内部首先创建了一个切点，然后创建了一个 advisor（一个小型切面），它对应的通知方法是 tracingInterceptor，对应的切点是 pointcutForadVisor。

需要注意的是为了能够使用 AOP 命名空间下的 aop 标签，您需要在 XML 引入下面的 spring-aop schema：

```
<?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-4.3.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
<!-- <bean/> definitions here -->
</beans>
```

把上面配置收集起来，放入到 beanaop.xml 配置文件后，写下下面代码，就可以进行测试：

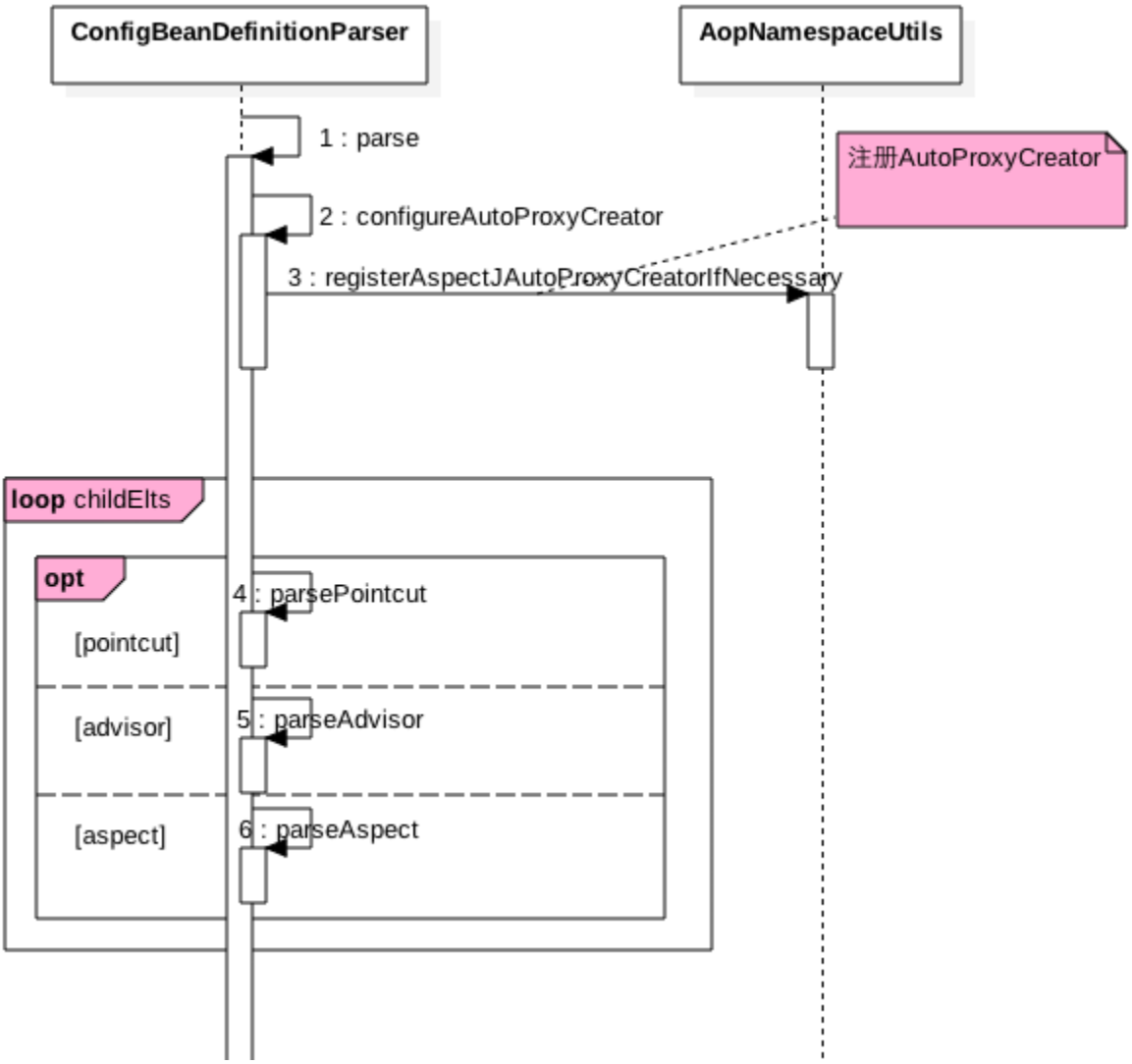
```
public class TestAOP {

    public static final String xmlpath = "beanaop.xml";
    public static void main(String[] args) {

        ClassPathXmlApplicationContext cpxa = new
ClassPathXmlApplicationContext(xmlpath);
        HelloServiceBo serviceBo =
cpxa.getBean("helloService",HelloServiceBo.class);
        serviceBo.sayHello(" I love you");
```

```
            String result = serviceBo.sayHelloAround("I love you");
            System.out.println(result);

            result = serviceBo.sayHelloAfterReturn("I love you");
            System.out.println(result);

            serviceBo.sayHelloAdvisor("I love you");

            serviceBo.sayHelloThrowExecption();

        }
    }
```

原理剖析

aop:config 标签的解析

既然本文讲解基于 XML 配置的 AOP 的原理，那么我们就先从解析 XML 里面配置的 aop:config 讲起。首先看看 Spring 框架的 ConfigBeanDefinitionParser 类是如何解析 aop:config 标签，主要时序图如下：



代码（3）注册了一个 AspectJAwareAdvisorAutoProxyCreator 类到 Spring IOC 容器，该类的作用是自动创建代理类，这个后面会讲。这里需要注意的是在 registerAspectJAutoProxyCreatorIfNecessary的useClassProxyingIfNecessary 方法里面解析了 aop:config 标签里面的 proxy-target-class 和 expose-proxy 属性值，代码如下：

```
    private static void
useClassProxyingIfNecessary(BeanDefinitionRegistry registry,
@Nullable Element sourceElement) {
        //解析proxy-target-class属性
        if (sourceElement != null) {
            boolean proxyTargetClass =
Boolean.valueOf(sourceElement.getAttribute(PROXY_TARGET_CLASS_ATT
RIBUTE));  //设置proxy-target-class属性值到
AspectJAwareAdvisorAutoProxyCreator里面
            if (proxyTargetClass) {

AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
            }
            //解析expose-proxy属性值
            boolean exposeProxy =
Boolean.valueOf(sourceElement.getAttribute(EXPOSE_PROXY_ATTRIBUTE
));
            //设置expose-proxy属性属性值到
AspectJAwareAdvisorAutoProxyCreator
            if (exposeProxy) {
    AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
            }
        }
    }
```

针对 proxy-target-class 属性（默认 false），如果设置为 true，则会强制使用 CGLIB 生成代理类；对于 expose-proxy 属性（默认 false）如果设置为 true，则对一个类里面的嵌套方法调用的方法也进行代理，具体说是如果一个 ServiceImpl 类里面有 a 和 b 方法，如果 a 里面调用了 b：

```
public void a(){
  this.b();
}
```

那么默认情况下在对方法 a 进行功能增强时候，里面的 b 方法是不会被增强的，如果也需要对 b 方法进行增强则可以设置 expose-proxy 为 true，并且在调用 b 方法时候使用下面方式：

```
public void a(){
 ((ServiceBo)AopContext.currentProxy()).b();
}
```

需要注意的是 AopContext.currentProxy() 返回的是代理后的类，如果使用 JDK 代理则这里类型转换时候必须要用接口类，如果是 CGLIB 代理则这里类型转换为实现类 ServiceImpl。

代码（4）是对 aop:config 标签里面的 aop:pointcut 元素进行解析，每个 pointcut 元素会创建一个 AspectJExpressionPointcut 的 bean 定义，并注册到Spring IOC，parsePointcut 的主干代码如下：

```
private AbstractBeanDefinition parsePointcut(Element
pointcutElement, ParserContext parserContext) {
    //获取aop:pointcut标签的id属性值
    String id = pointcutElement.getAttribute(ID);
    //获取aop:pointcut标签的expression属性值
    String expression = pointcutElement.getAttribute(EXPRESSION);

    AbstractBeanDefinition pointcutDefinition = null;

    try {
        //创建AspectJExpressionPointcut的bean定义，并设置属性
        pointcutDefinition =
createPointcutDefinition(expression);

pointcutDefinition.setSource(parserContext.extractSource(pointcut
Element));

        //如果aop:pointcut标签的id属性值不为空，则id作为该bean在Spring
容器里面的bean名字，并注册该bean到Spring容器。
        String pointcutBeanName = id;
        if (StringUtils.hasText(pointcutBeanName)) {

parserContext.getRegistry().registerBeanDefinition(pointcutBeanNa
me, pointcutDefinition);
        }
        else {//否者系统自动生成一个名字，并注入该bean到Spring容器
            pointcutBeanName =
parserContext.getReaderContext().registerWithGeneratedName(pointc
utDefinition);
        }
        ...
    }
    ...
    return pointcutDefinition;
}
```

注： AspectJExpressionPointcut 的成员变量 expression 保存了 aop:pointcut 标签的 expression 属性值，这个在自动代理时候会用到。

代码（5）是对 aop:config 标签里面的 aop:advisor 元素进行解析，每个 advisor 元素会创建一个 DefaultBeanFactoryPointcutAdvisor 的 bean 定义，并注册到 Spring IOC，

parseAdvisor代码如下：

```
private void parseAdvisor(Element advisorElement, ParserContext
parserContext) {
    //创建DefaultBeanFactoryPointcutAdvisor的定义，并设置对advice的引
用
    AbstractBeanDefinition advisorDef =
createAdvisorBeanDefinition(advisorElement, parserContext);
    String id = advisorElement.getAttribute(ID);

    try {
        ...
        //注册DefaultBeanFactoryPointcutAdvisor到Spring容器
        String advisorBeanName = id;
        if (StringUtils.hasText(advisorBeanName)) {

parserContext.getRegistry().registerBeanDefinition(advisorBeanNam
e, advisorDef);
        }
        else {
            advisorBeanName =
parserContext.getReaderContext().registerWithGeneratedName(adviso
rDef);
        }
        //解析aop:advisor标签中引用的切点，并设置到
DefaultBeanFactoryPointcutAdvisor的定义
        Object pointcut = parsePointcutProperty(advisorElement,
parserContext);
        if (pointcut instanceof String) {
                advisorDef.getPropertyValues().add(POINTCUT, new
RuntimeBeanReference((String) pointcut));
                ...
        }
        ...
    }
    ...
}
```

注：每个 DefaultBeanFactoryPointcutAdvisor 对象里面通过成员变量 adviceBeanName 保存引用通知在 BeanFactory 里面的 Bean 名称，通过成员变量 pointcut 保存切点。

DefaultBeanFactoryPointcutAdvisor 继承自 Advisor 接口，这里需要注意的是 adviceBeanName 保存的只是通知的字符串名称，那么如何获取真正的通知对象呢，其实 DefaultBeanFactoryPointcutAdvisor 实现了 BeanFactoryAware，其内部保证着 BeanFactory 的引用，既然有了 BeanFactory，那么就可以根据 Bean 名称拿到想要的 Bean，这个可以参考 Chat：Spring 框架常用扩展接口揭秘。

代码（6）是对 aop:config 标签里面的 aop:aspect 元素进行解析，会解析 aop:aspect 元素内的子元素，每个子元素会对应创建一个 AspectJPointcutAdvisor 的 bean 定义，并注册

到 Spring IOC，parseAspect的代码如下：

```
private void parseAspect(Element aspectElement, ParserContext
parserContext) {
        //获取 aop:aspect 元素的id属性
        String aspectId = aspectElement.getAttribute(ID);
        //获取 aop:aspect 元素的ref属性
        String aspectName = aspectElement.getAttribute(REF);

        try {
            this.parseState.push(new AspectEntry(aspectId,
aspectName));
            List<BeanDefinition> beanDefinitions = new
ArrayList<>();
            List<BeanReference> beanReferences = new ArrayList<>
();


            ...
            //循环解析```aop:aspect```元素内所有通知
            NodeList nodeList = aspectElement.getChildNodes();
            boolean adviceFoundAlready = false;
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                //判断是否为通知节点，比如前置，后者通知等
                if (isAdviceNode(node, parserContext)) {
                    if (!adviceFoundAlready) {
                        adviceFoundAlready = true;
                        ...
                        beanReferences.add(new
RuntimeBeanReference(aspectName));
                    }
                    //根据通知类型的不同，创建不同的通知对象,最后封装为
AspectJPointcutAdvisor的bean定义，并注册到Spring容器
                    AbstractBeanDefinition advisorDefinition =
parseAdvice(
                            aspectName, i, aspectElement,
(Element) node, parserContext, beanDefinitions, beanReferences);
                    beanDefinitions.add(advisorDefinition);
                }
            }
            //循环解析```aop:aspect```元素内所有切点，并注册到Spring容
器
            List<Element> pointcuts =
DomUtils.getChildElementsByTagName(aspectElement, POINTCUT);
            for (Element pointcutElement : pointcuts) {
                parsePointcut(pointcutElement, parserContext);
            }
            ...
        }
        finally {
```

```
                ...
            }
    }
```

需要注意的是 parseAdvice 函数内部会根据通知类型的不同创建不同的 advice 对象，对应 before 通知会创建通知对象为 AspectJMethodBeforeAdvice，对应 after 通知创建的通知对象为 AspectJAfterAdvice，对应 after-returning 通知创建的通知对象为 AspectJAfterReturningAdvice，对应 after-throwing 通知创建的通知对象为 AspectJAfterThrowingAdvice，对应 around 通知创建的通知对象为 AspectJAroundAdvice，一个共同点是这些通知对象都实现了 MethodInterceptor 接口。

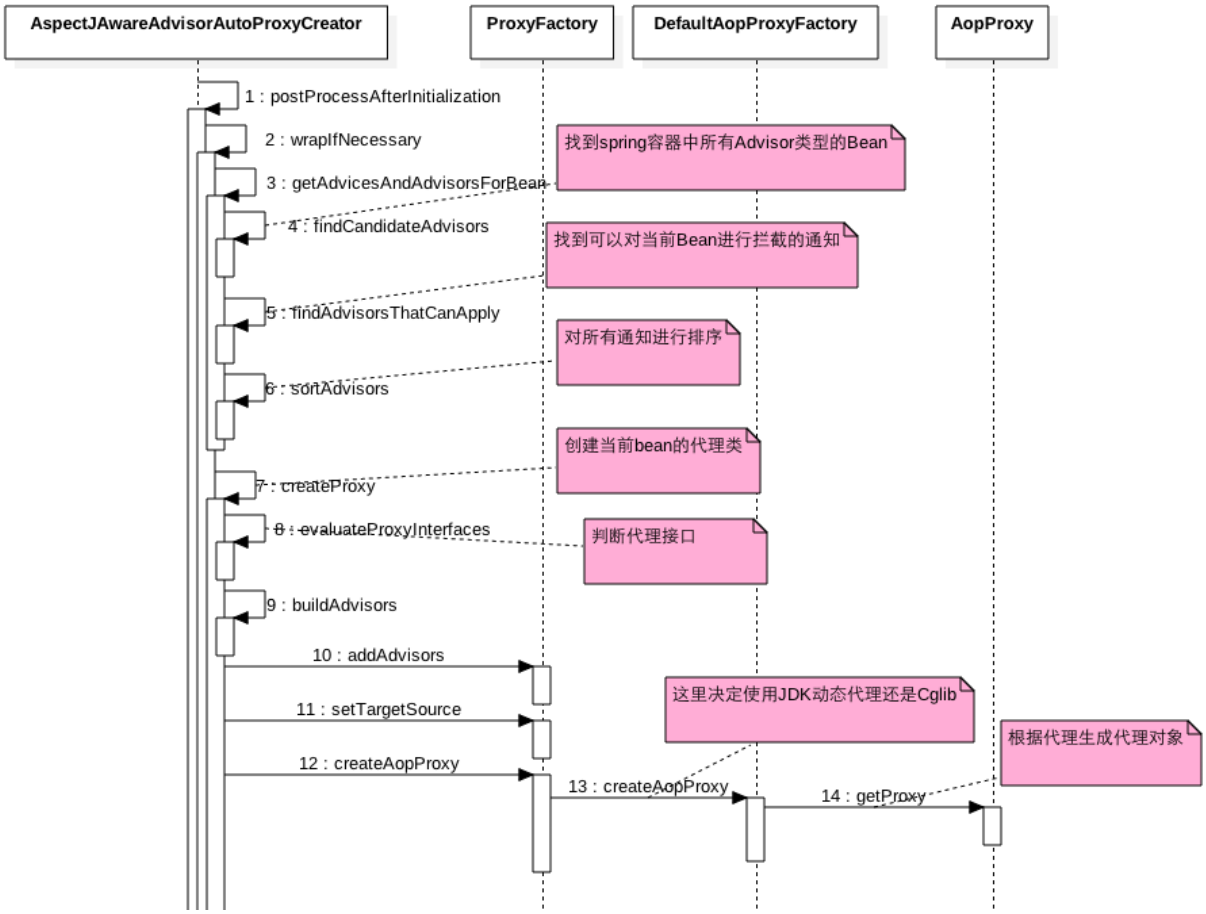
注：每个 AspectJPointcutAdvisor 对象里面通过 advice 维护着一个通知，通过 pointcut 维护这么一个切点，AspectJPointcutAdvisor 继承自 Advisor 接口。

至此 Spring 框架把 aop:config 标签里面的配置全部转换为了 Bean 定义的形式并注入到 Spring 容器了，需要注意的是对应一个 Spring 应用程序上下文的 XML 配置里面可以配置多个 aop:config 标签，每次解析一个 aop:config 标签的时候都会重新走这个流程。

代理类的生成

（1）代理类生成的主干流程

上节在解析 aop:config 标签时候注入了一个 AspectJAwareAdvisorAutoProxyCreator 类到 Spring 容器，其实这个类就是实现动态生成代理类的，AspectJAwareAdvisorAutoProxyCreator 实现了 BeanPostProcessor 接口（这个接口是 Spring 框架在 bean 初始化前后做事情的扩展接口，具体可以参考：http://gitbook.cn/gitchat/activity/5a84589a1f42d45a333f2a8e ），所以会有 Object postProcessAfterInitialization(@Nullable Object bean, String beanName) 方法，下面看下这个方法代码执行时序图：



代码（3）在 Spring 容器中查找可以对当前 bean 进行增强的通知 bean，内部首先执行代码（4）在 Spring 容器查找所有实现了 Advisor 接口的 Bean，也就是上节讲解的从 aop:config 标签里面解析的所有 DefaultBeanFactoryPointcutAdvisor 和 AspectJPointcutAdvisor 的实例都会被找到。代码（5）则看找到的 Advisor 里面哪些可以应用到当前 bean 上，这个是通过切点表达式来匹配的。代码（6）则对匹配的 Advisor 进行排序，根据每个 Advisor 对象的 getOrder 方法的值进行排序。

代码（13）这里根据一些条件决定是使用 JDK 动态代理，还是使用 CGLIB 进行代理，属于设计模式里面的策略模式。

```
public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
    //如果optimize =true或者proxyTargetClass=ture 或者没有指定代理接
口，则使用CGLIB进行代理
    if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
        Class<?> targetClass = config.getTargetClass();
        //如果目标类为接口或者目标类是使用JDK动态代理生成的类，则是要使用
JDK对其进行代理
        if (targetClass.isInterface() ||
Proxy.isProxyClass(targetClass)) {
            return new JdkDynamicAopProxy(config);
        }
        //否者使用CGLIB进行代理
        return new ObjenesisCglibAopProxy(config);
    }
```

```
        //使用JDK进行代理
        else {
            return new JdkDynamicAopProxy(config);
        }
```

对于 proxyTargetClass 前面已经讲过了可以通过在 aop:config 这个标签里面配置，那么 optimize 是在哪里配置的呢？其实除了本文讲的基于标签的 AOP 模式，Spring 还提供了比如下面的方式进行动态代理：

```
 <bean
class="org.springframework.aop.framework.autoproxy.BeanNameAutoPr
oxyCreator">
        <property name="beanNames" value="*impl"></property>  <!-
- 只为后缀为"impl"的bean生产代理 -->
        <property name="interceptorNames" value="myAdvisor">
</property>   <!--自定义方法拦击器-->
        <property name="optimize" value="true"></property>
  </bean>
```

BeanNameAutoProxyCreator 可以针对指定规则的 beanName 的 bean 使用 interceptorNames 进行增强，由于这时候不能确定匹配的 bean 是否有接口，所以这里 optimize 设置为 true，然后在创建代理工厂时候具体看被代理的类是否是接口决定是使用 JDK 代理还是 CGLIB 代理。

代码（14）则是具体调用 JdkDynamicAopProxy 或者 ObjenesisCglibAopProxy 的 getProxy 方法获取代理类，下面两节具体介绍如何生成。

（2）JDK 动态代理生成代理对象

首先看下 JdkDynamicAopProxy 的 getProxy 方法：

```
public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isDebugEnabled()) {
            logger.debug("Creating JDK dynamic proxy: target
source is " + this.advised.getTargetSource());
        }
        Class<?>[] proxiedInterfaces =
AopProxyUtils.completeProxiedInterfaces(this.advised, true);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader,
proxiedInterfaces, this);
    }
```

由于 JdkDynamicAopProxy 类实现了 InvocationHandler 接口，所以这里使用 Proxy.newProxyInstance 创建代理对象时候第三个参数传递的为 this。下面看下 JdkDynamicAopProxy 的 invoke 方法：

```
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        ...
        try {
            ...
            //获取可以运用到该方法上的拦截器列表
            List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(method,
targetClass);


            //创建一个invocation方法，这个里面是一个interceptor链，是一
个责任链模式
            invocation = new ReflectiveMethodInvocation(proxy,
target, method, args, targetClass, chain);
            //使用拦截器链处理这个连接点方法
            retVal = invocation.proceed();
            ...
            return retVal;
        }
        finally {
            ...
        }
}
```

下面看下上面小节“AOP 简单使用”例子中调用 serviceBo.sayHello 时候的时序图从而加深理解：

- “AOP 简单使用”这一小节例子中我们在 sayHello 方法执行前加了一个前置拦截器，在 sayHello 方法执行后加了个后置拦截器；
- 当执行 serviceBo.sayHello 时候实际上执行的代理类的 sayHello 方法，所以会被 JdkDynamicAopProxy 的 invoke 方法拦截，invoke 方法内首先调用 getInterceptorsAndDynamicInterceptionAdvice 方法获取配置到 sayHello 方法上的拦截器列表，然后创建一个 ReflectiveMethodInvocation 对象（内部是一个基于数数组的责任链），然后调用该对象的 procced 方法激活拦截器链对 sayHello 方法进行增强，这里是首先调用了前置连接器对 sayHello 进行增强，然后调用实际业务方法 sayHello，最后调用了后置拦截器对 sayHello 进行增强。

（3）CGLIB 动态代理生成代理对象

首先看下 ObjenesisCglibAopProxy 的 getProxy 方法：

```
public Object getProxy(@Nullable ClassLoader classLoader) {

    try {
        Class<?> rootClass = this.advised.getTargetClass();
```

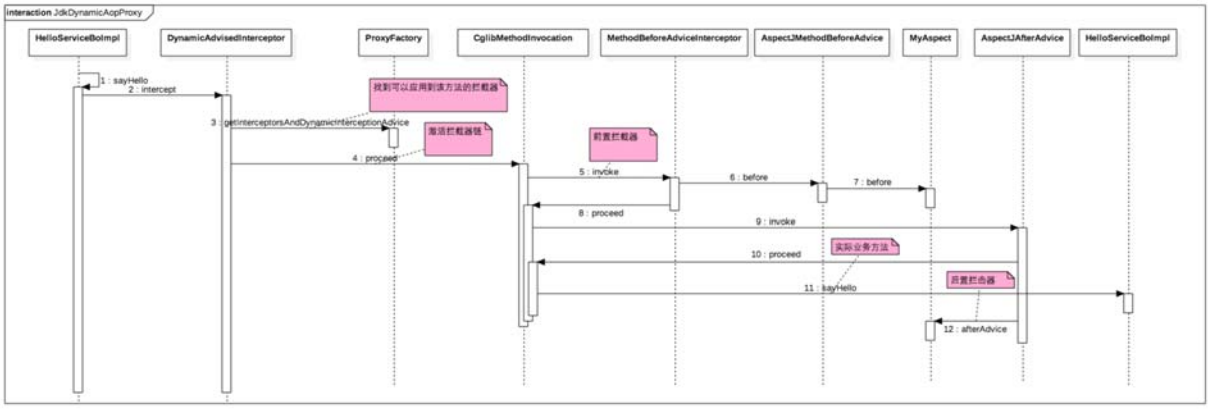
```
            //创建CGLIB Enhancer
            Enhancer enhancer = createEnhancer();
            ...
            enhancer.setSuperclass(proxySuperClass);

    enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfaces(th
    is.advised));
            enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
            ...
            //获取callback,主要是DynamicAdvisedInterceptor
            Callback[] callbacks = getCallbacks(rootClass);
                Class<?>[] types = new Class<?>[callbacks.length];
                for (int x = 0; x < types.length; x++) {
                    types[x] = callbacks[x].getClass();
            }
            //设置callback
            enhancer.setCallbackTypes(types);

            //产生代理类并创建一个代理实例
            return createProxyClassAndInstance(enhancer, callbacks);
        }
        catch (CodeGenerationException | IllegalArgumentException ex)
    {
            ...
        }
        ...
    }
```

下面看下拦截器 DynamicAdvisedInterceptor 的 intercept 方法，代码如下：

```
    public Object intercept(Object proxy, Method method, Object[]
    args, MethodProxy methodProxy) throws Throwable {
        ...
        TargetSource targetSource = this.advised.getTargetSource();
        try {
            ...
            //获取可以运用到该方法上的拦截器列表
            List<Object> chain =
    this.advised.getInterceptorsAndDynamicInterceptionAdvice(method,
    targetClass);
            Object retVal;
            //如果拦截器列表为空，则直接反射调用业务方法
            if (chain.isEmpty() &&
    Modifier.isPublic(method.getModifiers())) {
                Object[] argsToUse =
    AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                retVal = methodProxy.invoke(target, argsToUse);
            }
            else {
                //创建一个方法invocation,其实这个是个拦截器链，这里调用
```

```
proceed激活拦截器链对当前方法进行功能增强
            retVal = new CglibMethodInvocation(proxy, target,
method, args, targetClass, chain, methodProxy).proceed();
        }
        //处理返回值
        retVal = processReturnType(proxy, target, method,
retVal);
        return retVal;
    }
    finally {
        ...
    }
}
```

下面看“AOP 简单使用”这一节例子中调用 serviceBo.sayHello 时候的调用链路时序图从而加深理解：



- 由于默认使用的 JDK 动态代理，要使用 CGLIB 进行代理，需要在 aop:config 标签添加属性如下：

```
<aop:config  proxy-target-class="true">
```

- 执行流程类似于 JDK 动态代理，这里不再累述。

Spring 框架中如何基于 AOP 实现的事务管理

事务的简单配置

XML 使用标签配置事务，一般是按照下面方式进行配置：

```
<aop:config>
  <!--(1) -->
  <aop:pointcut id="businessService"
```

```
            expression="execution(* com.xyz.myapp.service.*.*(..))"/>
    <!--(2) -->
    <aop:advisor
        pointcut-ref="businessService"
        advice-ref="tx-advice"/>
  </aop:config>
    <!--(3) -->
  <tx:advice id="tx-advice">
    <tx:attributes>
      <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
  </tx:advice>
```

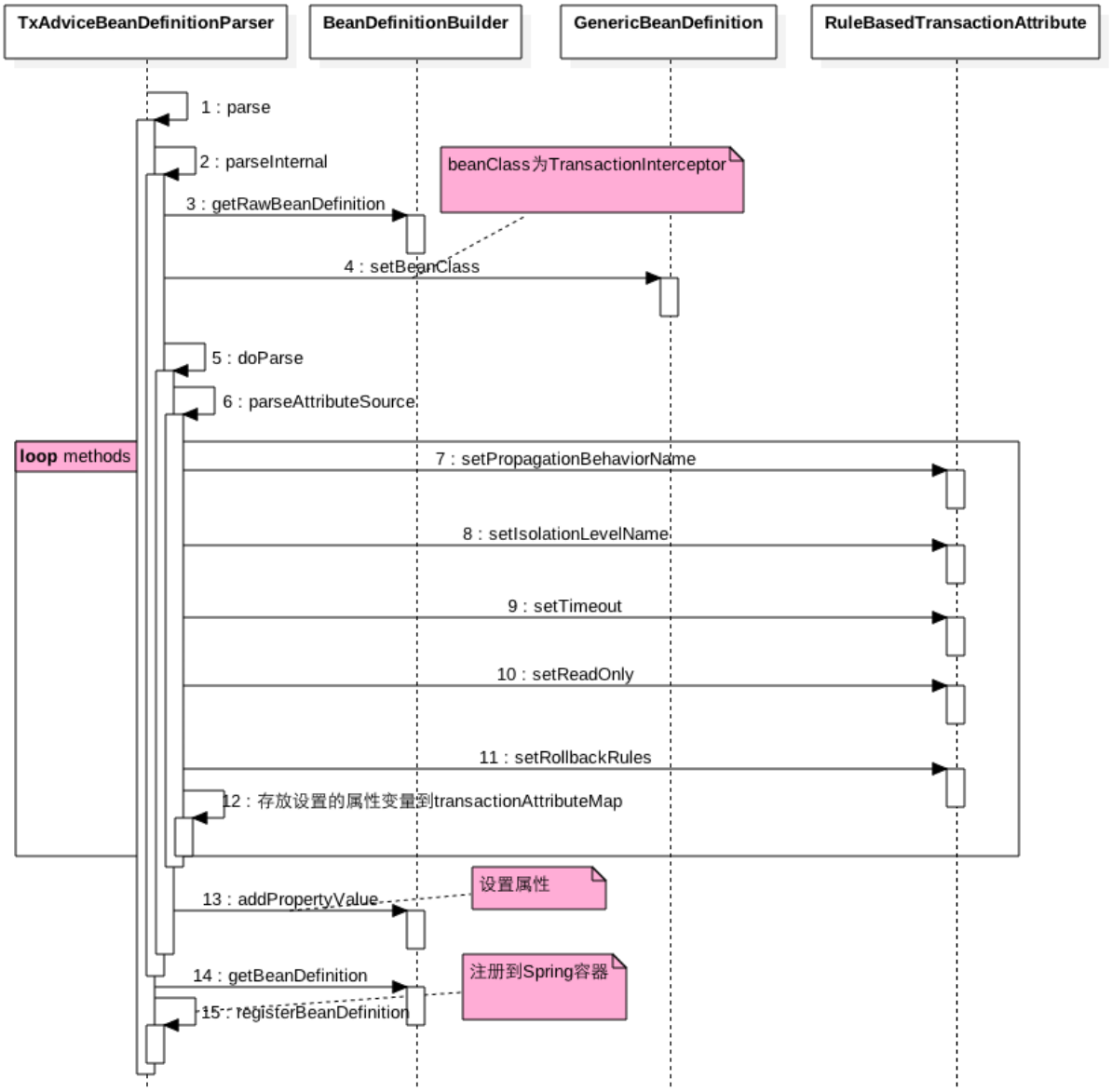
- 如上配置（1），配置了一个 id 为 businessService 的切点用来匹配要对哪些方法进行事务增强；
- 配置（2）配置了一个 advisor ，使用 businessService 作为切点，tx-advice 作为通知方法；
- 配置（3）则使用 tx:advice 标签配置了一个通知，这个是重点，下节专门讲解。

注：这个配置作用是对满足 id 为 businessService 的切点条件的方法进行事务增强，并且设置事务传播性级别为 REQUIRED。

原理剖析

tx:advice 标签的解析

tx:advice 标签是使用 TxAdviceBeanDefinitionParser 进行解析的，下面看看解析时序图：



- 如上时序图 BeanDefinitionBuilder 是一个建造者模式，用来构造一个 bean 定义，这个 bean 定义最终会生成一个 TransactionInterceptor 的实例；
- 步骤（5）通过循环解析 tx:attributes 标签里面的所有 tx:method 标签，每个 tx:method 对应一个 RuleBasedTransactionAttribute 对象，其中 tx:method 标签中除了可以配置事务传播性，还可以配置事务隔离级别，超时时间，是否只读，和回滚策略。
- 步骤（12）把所有的 method 标签的 RuleBasedTransactionAttribute 对象存在到了一个 map 数据结构，步骤（13）设置解析的所有属性到建造者模式的对象里面，步骤（14）使用建造者对象创建一个 bean 定义，步骤（15）则注册该 bean 到 Spring 容器。

注： tx:advice 作用是创建一个 TransactionInterceptor 拦截器，内部维护事务配置信息。

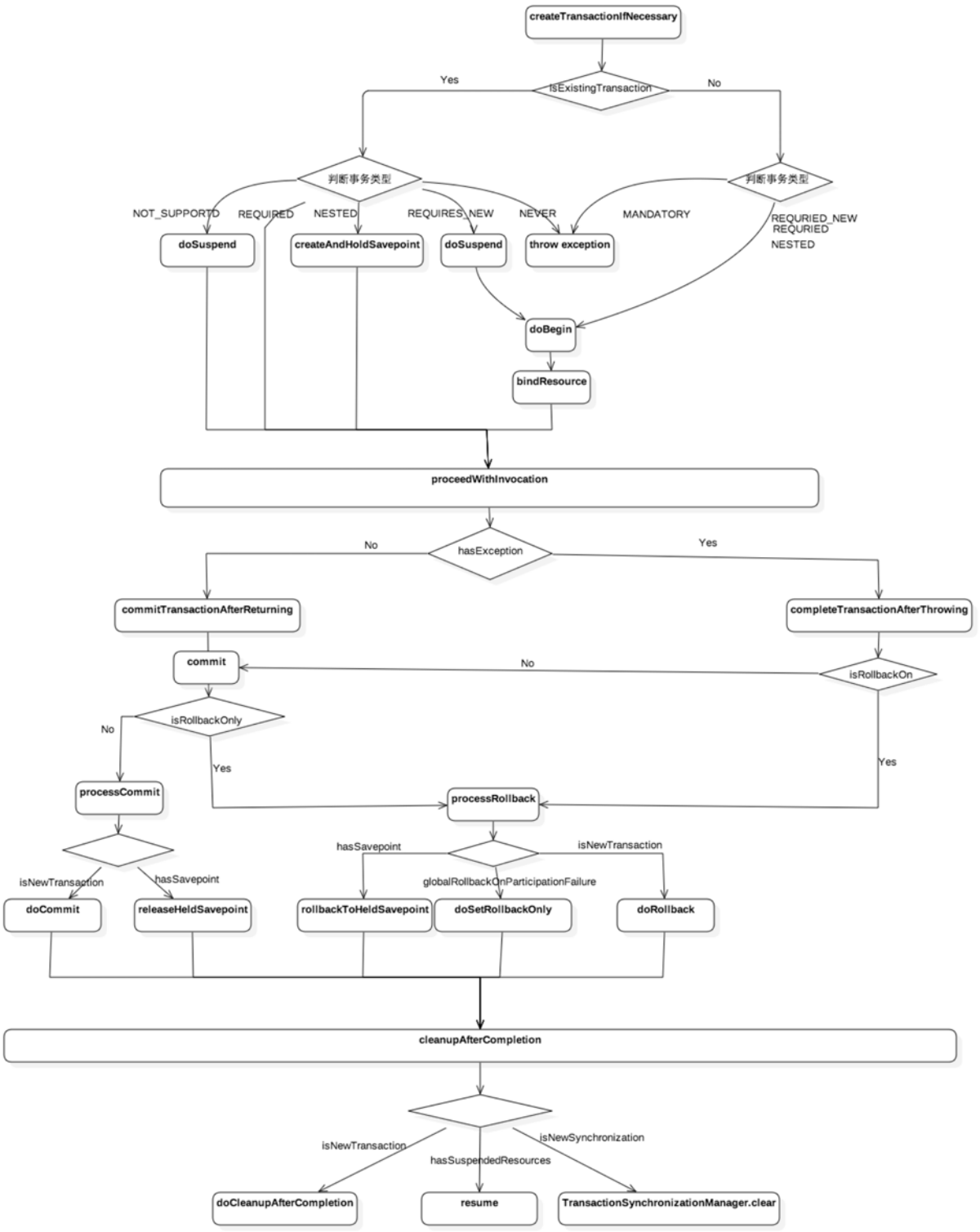
事务拦截器原理简单剖析

下面看下 TransactionInterceptor 的 invoke 方法：

```
public Object invoke(final MethodInvocation invocation) throws
Throwable {
```

```
    ...
    return invokeWithinTransaction(invocation.getMethod(),
targetClass, invocation::proceed);
}

protected Object invokeWithinTransaction(Method method, @Nullable
Class<?> targetClass,
            final InvocationCallback invocation) throws Throwable
{

        // If the transaction attribute is null, the method is
non-transactional.
        TransactionAttributeSource tas =
getTransactionAttributeSource();
        final TransactionAttribute txAttr = (tas != null ?
tas.getTransactionAttribute(method, targetClass) : null);
        final PlatformTransactionManager tm =
determineTransactionManager(txAttr);
        final String joinpointIdentification =
methodIdentification(method, targetClass, txAttr);

        if (txAttr == null || !(tm instanceof
CallbackPreferringPlatformTransactionManager)) {
            // 标准事务，内部有getTransaction（开启事务） 和commit（提
交）/rollback（回滚）事务被调用.
            TransactionInfo txInfo =
createTransactionIfNecessary(tm, txAttr,
joinpointIdentification);
            Object retVal = null;
            try {
                //这是一个环绕通知，调用proceedWithInvocation激活拦截
器链里面的下一个拦击器
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                // target invocation exception
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                cleanupTransactionInfo(txInfo);
            }
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }

        ...
}
```

其中 createTransactionIfNecessary 是重要方法，其内部逻辑处理流程如下图：



注： Spring 事务管理通过配置一个 AOP 切面来实现，其中定义了一个切点用来决定对哪些方法进行方法拦截，定义了一个 TransactionInterceptor 通知，来对拦截到的方法进行事务增强，具体事务拦击器里面是怎么做的，读者可以结合上面的 TransactionInterceptor 方法的流程图结合源码来研究下，如果必要后面会再开一个 Chat 专门讲解 Spring 事务的实现，以及事务隔离性与传播性。

总结

本文以大纲的形式剖析了 AOP 原理，以及事务拦击器如何使用 AOP 来实现，由于篇幅限制并没有深入到每个实现细节进行讲解，希望读者能够依靠本文作为大纲，对照源码深入到大纲里面的细节进行研究，以便加深对 AOP 原理的理解和掌握。