一、AOP的实现机制

1、动态代理：

在运行期间，为相应的接口动态生成对应的代理对象。可以将横切关注点逻辑封装到动态代理的InvocationHandler中，然后在系统运行期间，根据横切关注点需要织入的模块位置，将横切逻辑织入到相应的代理类中。以动态代理类为载体的横切逻辑的优点是所需要织入横切关注点逻辑的模块类都得是实现相应的接口，因为动态代理只针对接口开放。缺点是，在运行期间使用反射，性能较差。Spring AOP默认采用动态代理。

2、动态字节码增强:

只要class文件符合Java规范，就可以使用CGLIB工具，在程序运行期间，动态构建字节码的class文件。这样就可以为需要织入横切逻辑的模块类在运行期间，通过该技术，为这些系统模块类生成相应的子类，而将横切逻辑加到这些子类中，让程序在执行期间使用的是这些动态生成的子类，从而达到将横切逻辑织入系统的目的。使用该技术，即使模块类没有实现相应的接口，也可以对其进行扩展。final类型的类除外。当动态代理无法使用时，Spring采用CGLIB库动态字节码增强支持来实现AOP。

二、AOP的范畴

1、Joinpoint：将要在其上进行织入操作的系统执行点。Spring AOP仅支持"方法调用执行"的Joinpoint。

> 1.方法调用：某个方法被调用时所处的程序执行点。处在调用对象上的执行点。
>
> 2.方法调用执行：某个方法内部执行开始时点。处在被调用到的方法逻辑执行的时点。
>
> 3.构造方法执行：某个对象构造方法内部执行的开始时点。
>
> 4.字段设置：对象的某个属性通过setter方法被设置或者直接被设置的时点。
>
> 5.字段获取：某个对象相应属性被访问的时点。可通过getter访问，也可以直接访问。
>
> 6.异常处理执行：在异常抛出后，对应的异常处理逻辑执行的时点。
>
> 7.类初始化：类中某些静态类型或静态代码块的初始化时点。

2、Pointcut：代表Joinpoint的表述方式。将横切逻辑织入当前系统的过程中，需要参照Pointcut规定的Joinpoint信息，才可以知道应该往系统的哪些Joinpoint上织入横切逻辑。

> a.Pointcut的三种表述方式：
>
> - 直接指定Joinpoint所在的方法名称
> - 正则表达式：通过正则，来归纳表述需要符合某种条件的多组Joinpoint。
> - 使用特定的Pointcut表述语言：功能最强大，AspectJ使用这种方式来指定Pointcut。
>
> b.Pointcut运算：Pointcut之间可以进行逻辑运算。它们就相当于一个集合。
>
> 如Pointcut.Males || Pointcut.Females 代表全体学生

3、Advice：单一横切关注点逻辑的载体，它代表将会织入到Joinpoint的横切逻辑。把Aspect比作OOP的Class，那么Advice就相当于Class中的Method。

> - Before Advice：在Joinpoint指定位置之前执行的Advice类型。通常不会中断程序执行流程。通过它做一些初始化工作。
> - After Advice：在相应Joinpoint之后执行的Advice类型，可以分为以下三种
>
> a.After returning Advice：只有当前Joinpoint处执行流程正常完成后，该Advice才会执行。
>
> b.After throwing Advice：只有当期Joinpoint执行过程中抛出异常的情况下，该Advice才会执行

> c.After (Finally) Advice：不管Joinpoint处执行流程是正常结束还是抛出异常，该Advice都会执行。
>
> - Around Advice：对附加其上的Joinpoint进行包裹，可以在Joinpont之前和之后都指定相应的逻辑，甚至中断或者忽略Joinpoint处原来程序流程的执行。
> - Introduction：它不是根据横切逻辑在Joinpoint处执行时机来区分的，而是根据它可以完成的功能而区别于其他Advice类型。它可以为原有的对象添加新的特性或行为。

4、Aspect：是对系统中的横切关注点逻辑进行模块化封装的AOP概念实体。通常，Aspectd可以包含多个Pointcut以及相关Advice定义。Spring AOP最初实体和Aspect相对应，直到引入AspectJ后，可以使用@AspectJ的注解并结合普通的POJO来声明Aspect。

```
aspect AjStyleAspect{
    // pointcut定义
    pointcut query():call(public * get*(..));
    pointcut update():execution(public void updte*(..));

    // advice定义
    before():query(){
        ...
    }
    after() returnint:update(){
        ...
    }
}
```

5、织入和织入器：只有经过织入过程后，以Aspect模块化的横切关注点才会集成到OOP的系统中，完成织入过程的东西被称为织入器。

> AspectJ的ajc编译器就是它的织入器；Spring AOP使用一组类来完成织入操作，ProxyFactory是最通用的织入器。

6、目标对象：符合Pointcut所指定的条件，将在织入过程中被织入横切逻辑的对象。

7、小结

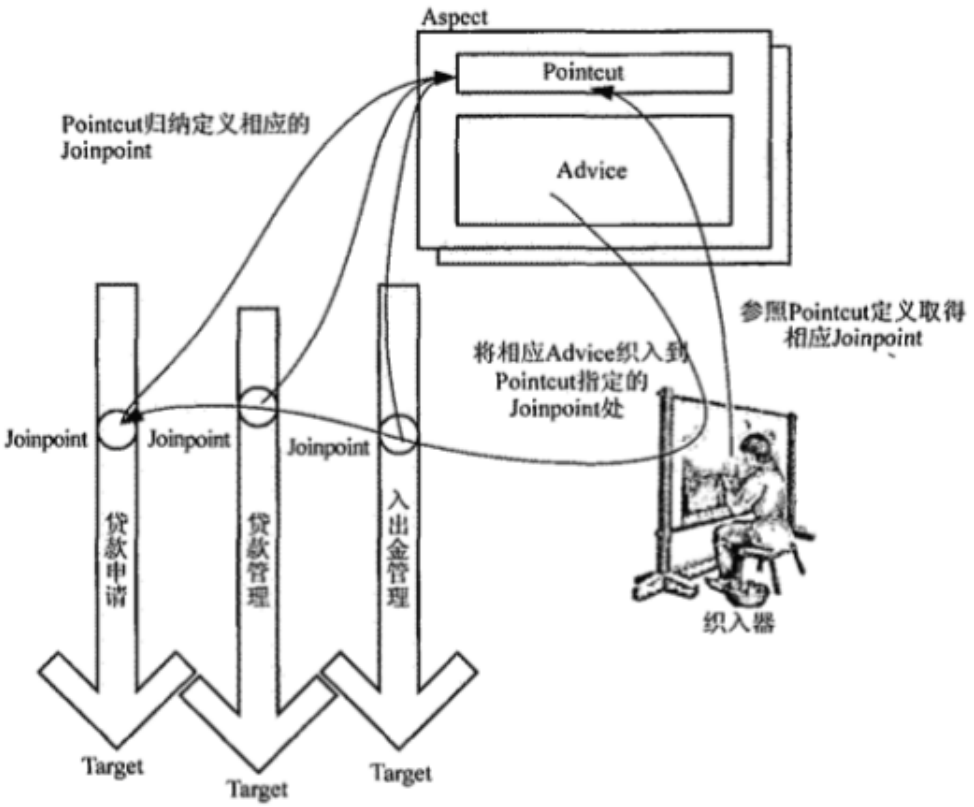


图7-9　AOP各个概念所处的场景

三、Spring AOP

1.实现机制：

采用动态代理和字节码生成技术，它们都是在运行期间为目标对象生成一个代理对象，而将横切逻辑织入到这个代理对象中，系统最终使用的是织入了横切逻辑的代理对象，而不是真正的目标对象。

1.1动态代理

> 代理模式：
>
> 1.抽象接口、具体实现类、代理类和客户端是代理模式的4中角色。代理类持有一个具体实现类的实例，并且也实现了抽象接口，客户端通过访问代理来间接访问具体类。将请求转发给被代理对象之前或之后，都可以根据情况插入其他处理逻辑。如果具体实现类是系统中的Joinpoint所在的对象，即目标对象，那么就可以为这个目标对象创建一个代理对象，然后将横切逻辑添加到这个代理对象中。

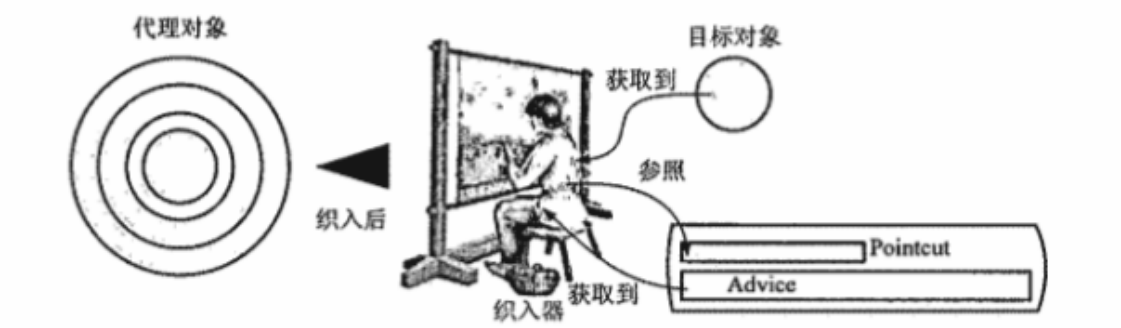


图8-4　AOP场景中的代理对象

BeanFactory,DefaultListableBeanFactory,AnnotationConfigApplicationContext和AppClient也是这种模式。

动态代理：上面的代理是为对应的目标对象创建静态代理的方法，实现上需要为每个具有同方法名的不同接口创建不同的代理类，实现上不可行。而动态代理可以为指定的接口在运行期间动态地生成代理对象。

动态代理机制的实现主要由一个类和一个接口组成：java.lang.reflect.Proxy类和java.lang.reflect.InvocationHandler接口。InvocationHandler调用处理程序。举个例子

```
public interface IRequest {
 void request();
}

public interface IRequestable {
 void request();
}

public class RequestImpl implements IRequest{
 @Override
 public void request() {
     System.out.println("这是IRequest的实现。");
 }
}

public class StaticProxy1 implements IRequest{

 private RequestImpl request = new RequestImpl();

 @Override
 public void request() {
     System.out.println("对IRequest的静态代理");
     request.request();
 }

}

public class StaticProxy2 implements IRequestable{

 private RequestableImpl requestable = new RequestableImpl();

 @Override
 public void request() {
     System.out.println("对IRequestable的静态代理");
     requestable.request();
```

```
 }
}


public class RequestableImpl implements IRequestable{
 @Override
 public void request() {
     System.out.println("这是IRequestable的实现。");
 }
}

public class RequestCtrlInvocationHandler implements InvocationHandler {

 private Object target;

 public RequestCtrlInvocationHandler(Object target){
     this.target = target;
 }

 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
     if (method.getName().equals("request")) {
         System.out.println("开始拦截request方法");
         return method.invoke(target, args);
     }
     return null;
 }

}


public class Client {

 public static void main(String[] args){
     // 静态代理：需要为每个request方法创建一个静态代理类
     StaticProxy1 staticProxy1 = new StaticProxy1();
     StaticProxy2 staticProxy2 = new StaticProxy2();
     staticProxy1.request();
     staticProxy2.request();

     System.out.println();
     System.out.println("===============>动态代理=============>..");
     // 动态代理：为所有的request方法，创建动态代理
     // 使用Proxy和RequestCtrlInvocationHandler创建不同类型目标对象的动态代理
     IRequest dynamicProxy1 = (IRequest) Proxy.newProxyInstance(
                          IRequest.class.getClassLoader(),
                         new Class[]{IRequest.class},
                         new RequestCtrlInvocationHandler(new
RequestImpl()));
     dynamicProxy1.request();


     IRequestable dynamicProxy2 = (IRequestable)
Proxy.newProxyInstance(IRequestable.class.getClassLoader(),
                          new Class[]{IRequestable.class},
                         new RequestCtrlInvocationHandler(new
RequestableImpl()));
```

```
        dynamicProxy2.request();

    }

}
```

即使还有更多的目标对象类型，只要它们依然织入的横切逻辑相同，用这个RequestCtrlInvocationHandler一个类并通过Proxy为它们生成相应的动态代理类实例就可以满足需求。InvocationHandler就是实现横切逻辑的地方，它是横切逻辑的载体，作用跟Advice是一样的。所以在使用动态代理机制实现AOP的过程中，可以在InvocationHandler的基础上细化程序结构，并根据Advice的类型，分化出对应不同Advice类型的程序结构。

默认情况下如果Spring AOP发现目标对象实现了相应的接口，则采用动态代理机制为其生成代理对象实例。而如果目标对象没有实现任何接口，Spring AOP会尝试使用CGLIB（Code Generation Library）为目标对象生成动态的代理对象实例。

1.2动态字节码生成

原理：可以对目标对象进行继承扩展，为其生成相应的子类，而子类就可以通过覆写来扩展父类的行为，只要将横切逻辑的实现放到子类中，然后让系统使用扩展后的目标对象的子类，就可以达到与代理模式相同的效果了。

```
public class Requestable {

public void request(){
   System.out.println("hello request...");
}

}

public class RequestCtrlCallback implements MethodInterceptor {

@Override
public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
   if (method.getName().equals("request")) {
       System.out.println("开始拦截request方法");
       return methodProxy.invokeSuper(o, objects); // 调用父类（目标对象）的方法
   }
   return null;
}
}

public class Client {

public static void main(String[] args){

   Enhancer enhancer = new Enhancer();
   enhancer.setSuperclass(Requestable.class);
   enhancer.setCallback(new RequestCtrlCallback());
   Requestable proxy = (Requestable) enhancer.create();
   proxy.request();

}

}
```

2.Spring AOP中的Pointcut

3.Spring AOP中的 Advice

Spring的Advice实现全部遵循AOP Alliance组织的接口规范。

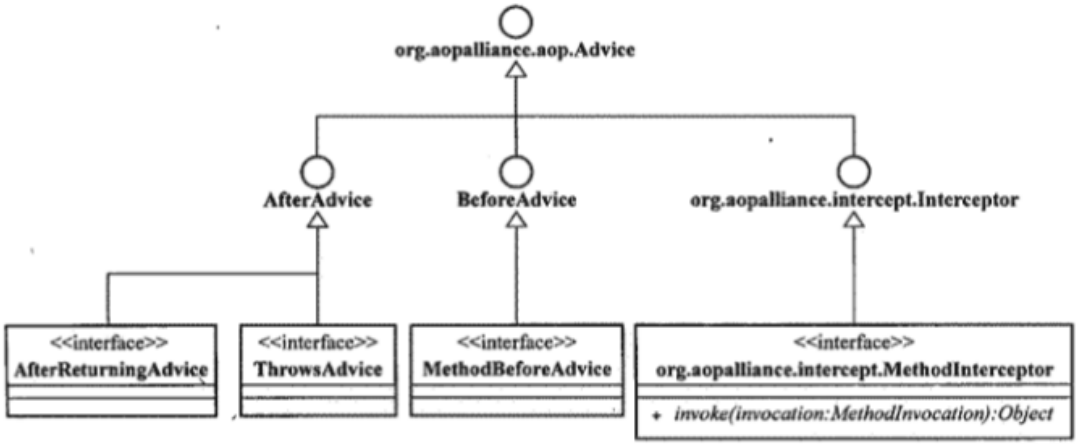


图9-4 Spring中Advice略图

3.1Spring Advice类型

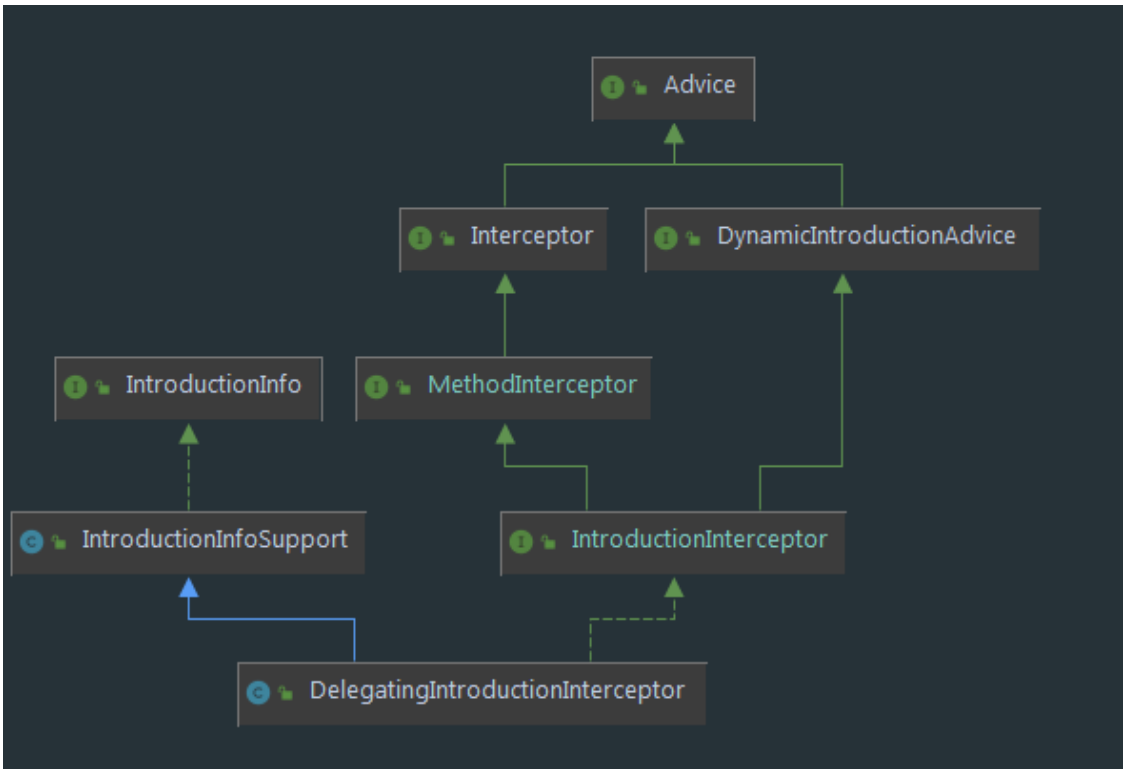
按照其自身实例能够在目标对象类的所有实现中共享这一标准，可以分为per-class和per-instance类型。

> per-class类型的Advice
>
> 可以在目标对象的所有实例之间共享。通常只是提供方法拦截的功能，不会为目标对象类保存任何状态或者添加新的特性。处理上图没有列出的Introduction类型的Advice不属于per-class类型的Advice之外，其他的都是。
>
> 1. Before Advice：所实现的横切逻辑将在相应的Joinpoint之前执行。
> 2. ThrowsAdvice：以统一的方式对异常进行处理。执行相应的Joinpoint发生异常时，执行这部分横切逻辑。
> 3. AfterReturningAdvice：只有方法正常返回的情况下，它才会被执行。Spring不支持对方法返回值进行修改。
> 4. AroundAdvice：Spring没有直接定义对应Around Advice的接口，而是直接采用AOP Alliance的接口标准即org.aopalliance.intercept.MethodInterceptor。它能完成前几种Advice干的事。

```
class TracingInterceptor implements MethodInterceptor {
 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(); // 让相应的调用链继续执行，这句是必须的。
   System.out.println("method "+i.getMethod()+" returns "+ret);
   return ret;
 }
}
```

> per-instance类型的Advice

> IntroductionInterceptor就是Spring AOP的Introduction



4.Spring AOP中的Aspect

Advisor代表Spring的Aspect，与正常的Aspect不同，Advisor只持有一个Pointcut和一个Advice。理论上的Aspect定义中可以有多个Pointcut和多个Advice。

Advisor的分支

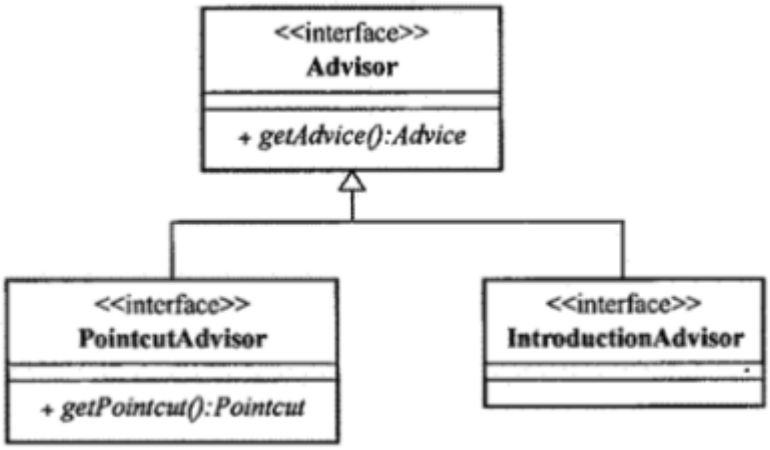


图9-6 Advisor分支

4.1PointcutAdvisor族谱

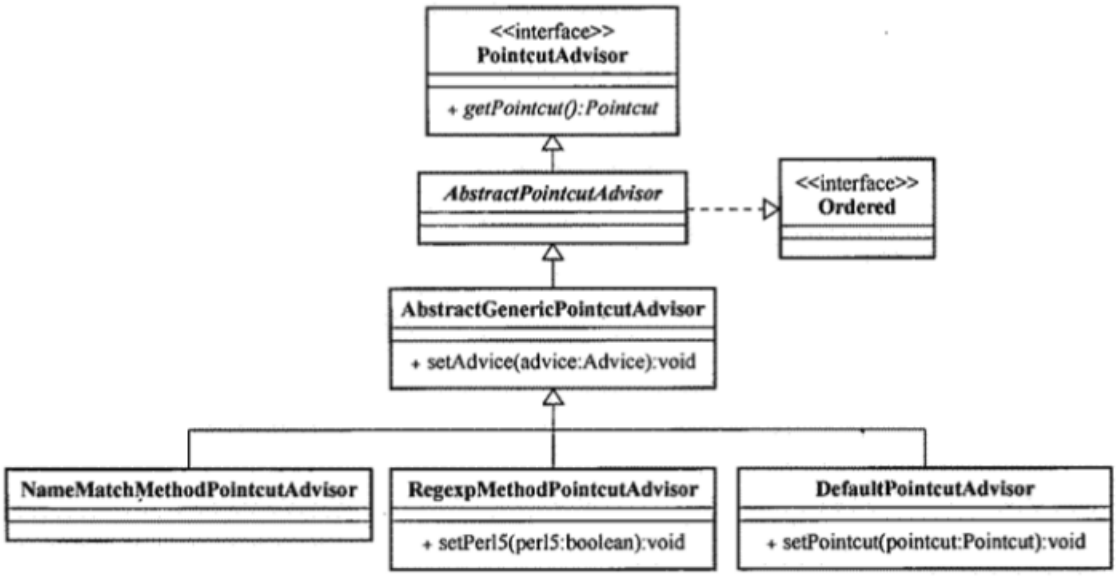


图9-7 PointcutAdvisor及相关子类

> DefaultPointcutAdvisor是最通用的PointcutAdvisor实现，除了不能为其指定Introduction类型的Advice之外，剩下的任何类型的Pointcut、任何类型的Advice都可以通过它来使用。在构造它的时候，传入Pointcut和Advice实例。
>
> 其他类型的xxxAdvisor都把Pointcut固化了，且不能更改。

4.2 IntroductionAdvisor分支

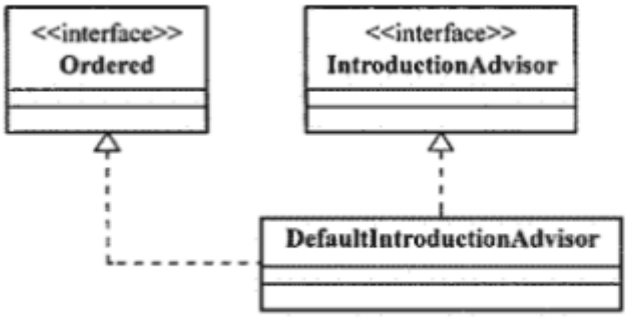


图9-8 IntroductionAdvisor类结构图

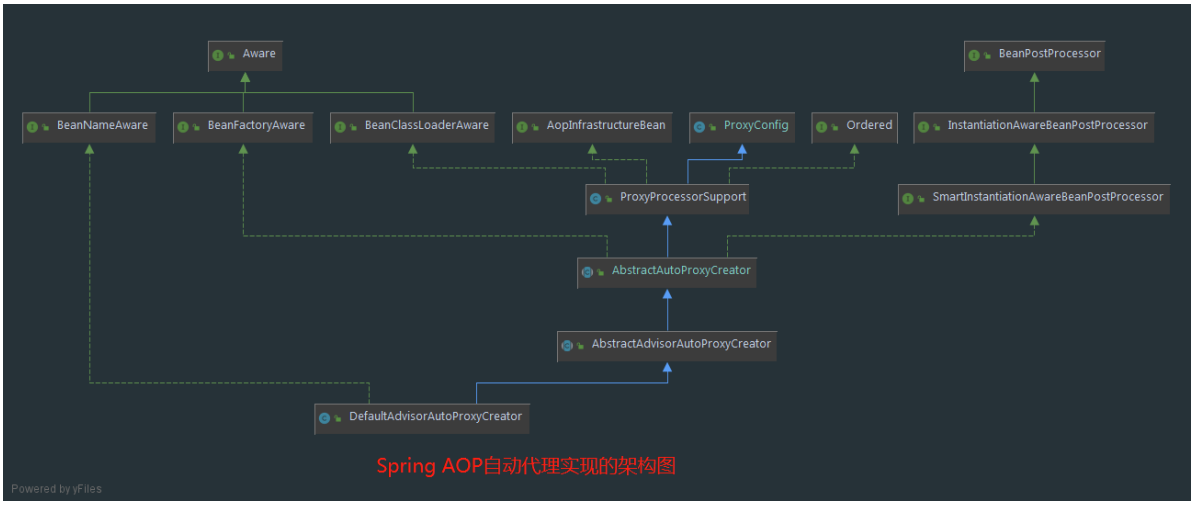
它们只能用于类级别的拦截，只能使用Introduction型的Advice。

5.Spring AOP的织入

AspectJ采用ajc编译器作为织入器，而Spring AOP中使用ProxyFactory作为织入器。

使用ProxyFactory只需要指定目标对象和Aspect即可。

6.自动代理



Spring AOP自动代理实现的架构图

> 所有的AutoProxyCreator都是InstantiationAwareBeanPostProcessor，它与普通的BeanPostProcessor不同。当Spring IoC容器检测到InstantiationAwareBeanPostProcessor的时候，会直接通过InstantiationAwareBeanPostProcessor中的逻辑构造对象实例返回，而不会走正常的对象实例化流程，造成“短路”的现象，AutoProxyCreator则直接返回目标对象的代理对象。