第 9 章 Spring AOP一世 9

本章内容

- Spring AOP中的Joinpoint
- Spring AOP中的Pointcut
- Spring AOP中的Advice
- Spring AOP中的Aspect
- Spring AOP的织入
- `TargetSource`

在动态代理和CGLIB的支持下，Spring AOP框架的实现经过了两代。从Spring的AOP框架第一次发布，到Spring 2.0发布之前的AOP实现，是Spring第一代AOP实现。Spring 2.0发布后的AOP实现是第二代。不过划分归划分，Spring AOP的底层实现机制却一直没变。唯一改变的，是各种AOP概念实体的表现形式以及Spring AOP的使用方式。

下面，我们先从第一代的Spring AOP相关的概念实体说起。

9.1 Spring AOP 中的 Joinpoint

之前我们已经提到，AOP的Joinpoint可以有许多种类型，如构造方法调用、字段的设置及获取、方法调用、方法执行等。但是，在Spring AOP中，仅支持方法级别的Joinpoint。更确切地说，只支持方法执行（Method Execution）类型的Joinpoint。这一点我们从8.2节就能看出来。

虽然Spring AOP仅提供方法拦截，但是在实际的开发过程中，这已经可以满足80%的开发需求了。所以，我们不用过于担心Spring AOP的能力。

Spring AOP之所以如此，主要有以下几个原因。

(1) 前面说过了，Spring AOP要提供一个简单而强大的AOP框架，并不想因大而全使得框架本身过于臃肿。如果能够仅付出20%的努力，就能够得到80%的回报，这难道不是很好吗？*Keep It Simple, Stupid*原则指导我们抛弃旧有的EJB2时代的思想和模式，它同样适用在这里；否则，事倍功半，并不是想看到的结果。

(2) 对于类中属性（Field）级别的Joinpoint，如果提供这个级别的拦截支持，那么就破坏了面向对象的封装，而且，完全可以通过对setter和getter方法的拦截达到同样的目的。

(3) 如果应用需求非常特殊，完全超出了Spring AOP提供的那80%的需求支持，不妨求助于现有的其他AOP实现产品，如AspectJ。目前来看，AspectJ是Java平台对AOP支持最完善的产品，同时，Spring AOP也提供了对AspectJ的支持。（不过，需要注意的是，AspectJ也不是完全支持所有类型的Joinpoint，如程序中的循环结构。部分原因应该归结为要实现这20%的需求，可能需要付出80%的工作和努力。）

9.2 Spring AOP 中的 Pointcut

Spring中以接口定义org.springframework.aop.Pointcut作为其AOP框架中所有Pointcut的最顶层抽象，该接口定义了两个方法用来帮助捕捉系统中的相应Joinpoint，并提供了一个TruePointcut类型实例。如果Pointcut类型为TruePointcut，默认会对系统中的所有对象，以及对象上所有被支持的Joinpoint进行匹配。org.springframework.aop.Pointcut接口定义如以下代码所示：

```
public interface Pointcut {
   ClassFilter getClassFilter();
   MethodMatcher getMethodMatcher();
   Pointcut TRUE = TruePointcut.INSTANCE;
}
```

ClassFilter和MethodMatcher分别用于匹配将被执行织入操作的对象以及相应的方法。之所以将类型匹配和方法匹配分开定义，是因为可以重用不同级别的匹配定义，并且可以在不同的级别或者相同的级别上进行组合操作，或者强制让某个子类只覆写（Override）相应的方法定义等。

ClassFilter接口的作用是对Joinpoint所处的对象进行Class级别的类型匹配，其定义如下：

```
public interface ClassFilter {
   boolean matches(Class clazz);
   ClassFilter TRUE = TrueClassFilter.INSTANCE;
}
```

当织入的目标对象的Class类型与Pointcut所规定的类型相符时，matches方法将会返回true，否则，返回false，即意味着不会对这个类型的目标对象进行织入操作。比如，如果我们仅希望对系统中Foo类型的类执行织入，则可以如下这样定义ClassFilter：

```
public class FooClassFilter{
   public boolean matches(Class clazz){
      return Foo.class.equals(clazz);
   }
}
```

当然，如果类型对我们所捕捉的Joinpoint无所谓，那么Pointcut中使用的ClassFilter可以直接使用“ClassFilter TRUE = TrueClassFilter.INSTANCE;”。当Pointcut中返回的ClassFilter类型为该类型实例时，Pointcut的匹配将会针对系统中所有的目标类以及它们的实例进行。

相对于ClassFilter的简单定义，MethodMatcher则要复杂得多。毕竟，Spring主要支持的就是方法级别的拦截——“重头戏”可不能单薄啊！MethodMatcher定义如下：

```
public interface MethodMatcher {
   boolean matches(Method method, Class targetClass);
   boolean isRuntime();
   boolean matches(Method method, Class targetClass, Object[] args);
   MethodMatcher TRUE = TrueMethodMatcher.INSTANCE;
}
```

MethodMatcher通过重载（Overload），定义了两个matches方法，而这两个方法的分界线就是isRuntime()方法。在对对象具体方法进行拦截的时候，可以忽略每次方法执行的时候调用者传入的参数，也可以每次都检查这些方法调用参数，以强化拦截条件。假设对以下方法进行拦截：

```
public boolean login(String username,Sring password);
```

如果只想在login方法之前插入计数功能，那么login方法的参数对于Joinpoint捕捉就是可以忽略的。而如果想在用户登录的时候对某个用户做单独处理，如不让其登录或者给予特殊权限，那么这个方法的参数就是在匹配Joinpoint的时候必须要考虑的。

(1) 在前一种情况下，isRuntime返回false，表示不会考虑具体Joinpoint的方法参数，这种类型的MethodMatcher称之为StaticMethodMatcher。因为不用每次都检查参数，那么对于同样类型的方法匹配结果，就可以在框架内部缓存以提高性能。isRuntime方法返回false表明当前的MethodMatcher为StaticMethodMatcher的时候，只有boolean matches(Method method, Class targetClass);方法将被执行，它的匹配结果将会成为其所属的Pointcut主要依据。

(2) 当isRuntime方法返回true时，表明该MethodMatcher将会每次都对方法调用的参数进行匹配检查，这种类型的MethodMatcher称之为DynamicMethodMatcher。因为每次都要对方法参数进行检查，无法对匹配的结果进行缓存，所以，匹配效率相对于StaticMethodMatcher来说要差。而且大部分情况下，StaticMethodMatcher已经可以满足需要，最好避免使用DynamicMethodMatcher类型。如果一个MethodMatcher为DynamicMethodMatcher（isRuntime()返回true），并且当方法boolean matches(Method method, Class targetClass);也返回true的时候，三个参数的matches方法将被执行，以进一步检查匹配条件。如果方法boolean matches(Method method,Class targetClass);返回false，那么不管这个MethodMatcher是StaticMethodMatcher还是DynamicMethodMatcher，该结果已经是最终的匹配结果——你可以猜得到，三个参数的matches方法那铁定是执行不了了。

在MethodMatcher类型的基础上，Pointcut可以分为两类，即StaticMethodMatcherPointcut和DynamicMethodMatcherPointcut。因为StaticMethodMatcherPointcut具有明显的性能优势，所以，Spring为其提供了更多支持。图9-1给出了Spring AOP中各Pointcut类型之间的一个局部“族谱”。

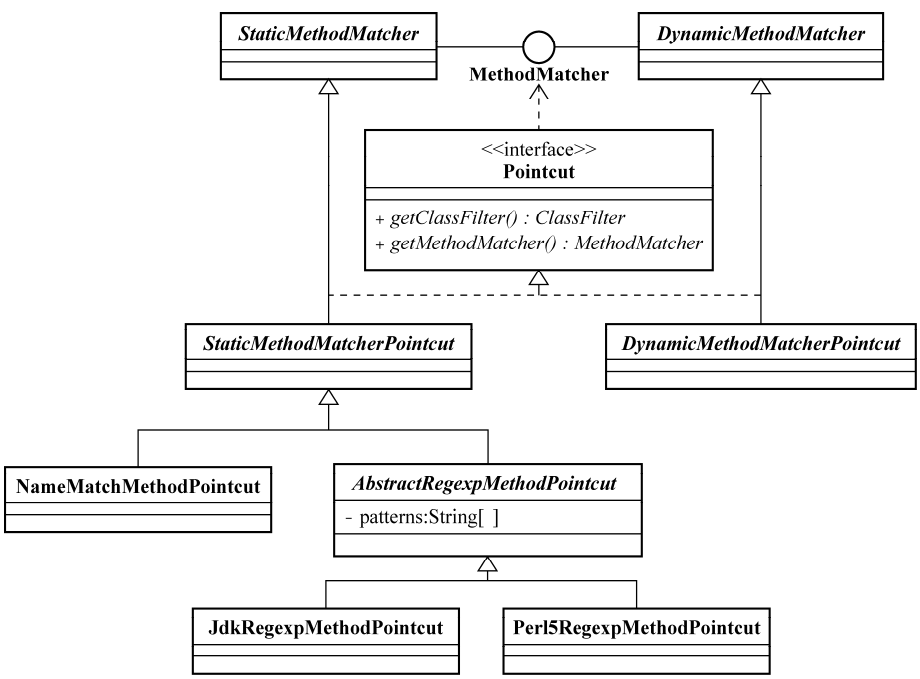

图9-1 Pointcut局部“族谱”

在深入这个“族谱”之前，我们先来看看Spring的AOP框架提供了哪些常见的Pointcut。毕竟，谁都有点儿想“坐享其成”嘛。

9.2.1 常见的 Pointcut

总地来说，图9-2给出了较为常用的几种Pointcut实现。

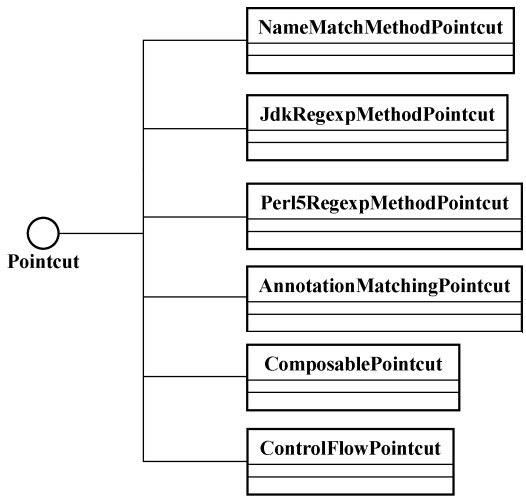

图9-2 常见的Pointcut

1. **NameMatchMethodPointcut**

这是最简单的Pointcut实现，属于StaticMethodMatcherPointcut的子类，可以根据自身指定的一组方法名称与Joinpoint处的方法的方法名称进行匹配，比如：

```
NameMatchMethodPointcut pointcut = new NameMatchMethodPointcut();
pointcut.setMappedName("matches");
// 或者传入多个方法名
pointcut.setMappedNames(new String[]{"matches","isRuntime"});
```

但是，NameMatchMethodPointcut无法对重载（Overload）的方法名进行匹配，因为它仅对方法名进行匹配，不会考虑参数相关信息，而且也没有提供可以指定参数匹配信息的途径。

NameMatchMethodPointcut除了可以指定方法名，以对指定的Joinpoint进行匹配，还可以使用“*”通配符，实现简单的模糊匹配，如下所示：

```
pointcut.setMappedNames(new String[]{"match*","*matches","mat*es"});
```

如果基于“*”通配符的NameMatchMethodPointcut依然无法满足对多个特定Joinpoint的匹配需要，那么使用正则表达式好了。

2. **JdkRegexpMethodPointcut和Perl5RegexpMethodPointcut**

StaticMethodMatcherPointcut的子类中有一个专门提供基于正则表达式的实现分支，以抽象类AbstractRegexpMethodPointcut为统帅。与NameMatchMethodPointcut相似，AbstractRegexp-MethodPointcut声明了pattern和patterns属性，可以指定一个或者多个正则表达式的匹配模式（Pattern）。其下设JdkRegexpMethodPointcut和Perl5RegexpMethodPointcut两种具体实现，就如

你在图9-1中所看到的那样。

JdkRegexpMethodPointcut的实现基于JDK 1.4之后引入的JDK标准正则表达式。如果想使用该Pointcut实现，那么首先需要保证应用程序是运行在1.4或者更高版本的JVM之上。JdkRegexpMethodPointcut的简单使用示例如下：

```
JdkRegexpMethodPointcut pointcut = new JdkRegexpMethodPointcut();
pointcut.setPattern(".*match.*");
// 或者
pointcut.setPatterns(new String[]{".*match.*",".*matches"});
```

注意，使用正则表达式来匹配相应的Joinpoint所处的方法时，正则表达式的匹配模式必须以匹配整个方法签名（Method Signature）的形式指定，而不能像NameMatchMethodPointcut那样仅给出匹配的方法名称。也就是说，如果有对象定义如下：

```
package cn.spring21.sample;
public class Bar
{
   public void doSth()
   {
      ...
   }
}
```

那么，使用正则表达式.*doSth.*则会匹配Bar的doSth方法，即相当于cn.spring21.demo.Bar.doSth。但如果Pointcut使用doSth.*作为匹配的正则表达式模式，就无法捕捉到Bar的doSth方法的执行。当然，也可以通过正则表达式中的转义，更确切地指定对doSth方法的匹配：cn\.spring21\.sample\.Bar\.doSth。不过，这已经属于正则表达式如何使用的范畴了。

注意 更多有关JDK中正则表达式的信息，可以参考1.4或者更高版本的JDK的Javadoc中java.util.regex.Pattern类定义，其中有很详细的有关正则表达式匹配模式如何指定的信息。

如果当前应用还无法使用JDK 1.4或者更高版本，或者我们更喜欢perl5风格的正则表达式，那么可以使用Perl5RegexpMethodPointcut。该Pointcut实现使用Jakarta ORO提供正则表达式支持，所以，在使用之前，请先将JakartaORO的jar包加入应用程序的classpath中。除了正则表达式的语法上可能有少许差异，Perl5RegexpMethodPointcut的使用和需要注意的问题与JdkRegexpMethodPointcut几乎相同，如下所示。

(1) 可以通过pattern或者patterns对象属性指定一个或者多个正则表达式的匹配模式。

(2) 指定的正则表达式匹配模式应该覆盖匹配整个方法签名，而不是只指定到方法名称部分。

有关该类使用的正则表达式的具体风格，请参照JakartaORO的相应文档。

3. AnnotationMatchingPointcut

不好意思，JDK又升级了。AnnotationMatchingPointcut只能用在使用JDK 5或者更高版本的应用中，因为注解是在Java 5（Tiger）发布后才有的。

AnnotationMatchingPointcut根据目标对象中是否存在指定类型的注解来匹配Joinpoint，要使用该类型的Pointcut，首先需要声明相应的注解。代码清单9-1给出了两个我们将用到的注解的定义。

代码清单9-1 用于Pointcut的ClassLevelAnnotation和MethodLevelAnnotation的定义

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
```

```
public @interface ClassLevelAnnotation {
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MethodLevelAnnotation {
}
```

注解定义中的@Target指定该注解可以标注的类型。对于我们声明的两个注解来说就是，ClassLevelAnnotation用于类层次，而MethodLevelAnnotation只能用于方法层次。代码清单9-2给出了这两个注解的使用示例。

代码清单9-2 ClassLevelAnnotation和MethodLevelAnnotation使用示例

```
@ClassLevelAnnotation
public class GenericTargetObject {

   @MethodLevelAnnotation
   public void gMethod1()
   {
      System.out.println("gMethod1");
   }
   public void gMethod2()
   {
      System.out.println("gMethod2");
   }
}
```

针对代码清单9-2中的GenericTargetObject类型，不同的AnnotationMatchingPointcut定义会产生不同的匹配行为。

- AnnotationMatchingPointcut pointcut = new AnnotationMatchingPointcut(ClassLevelAnnotation.class)

AnnotationMatchingPointcut仅指定类级别的注解，GenericTargetObject类中所有方法执行的时候，将全部匹配，不管该方法指定了注解还是没有指定。也可以通过AnnotationMatchingPointcut的静态方法，来构建类级别的注解对应的AnnotationMatchingPointcut实例：

```
AnnotationMatchingPointcut pointcut = AnnotationMatchingPointcut.forClassAnnotation ➥
(ClassLevelAnnotation.class);
```

- AnnotationMatchingPointcut pointcut = AnnotationMatchingPointcut.for MethodAnnotation(MethodLevelAnnotation.class)

如果只指定方法级别的注解而忽略类级别的注解，则该Pointcut仅匹配特定的标注了指定注解的方法定义，而忽略其他方法。对于GenericTargetObject，或者其他类中任何方法来说，只要它标注了@MethodLevelAnnotation，则这些方法将都会被匹配并织入相应的横切逻辑。

- AnnotationMatchingPointcut pointcut = new AnnotationMatchingPointcut(ClassLevelAnnotation.class,MethodLevelAnnotation.class)

通过同时限定类级别的注解和方法级别的注解，我们可以进一步缩小“包围圈”。现在，只有标注了@ClassLevelAnnotation的类定义中同时标注了@MethodLevelAnnotation的方法才会被匹配，二者缺一不可。对于我们的GenericTargetObject来说，只有它的gMethod1()方法才会被拦截，而gMethod2()则不会。

注解作为外部配置方式的另一种应用配置方式（或者更高一步，Attribute-Oriented Programming），

已经愈发受到更多开发人员的青睐。从早期的xdoclet、Jakarta Commons Attributes，到Java 5引入正式的注解支持，我们现在几乎到处都能看到注解的身影——Guice、AspectJ甚至这里的AnnotationMatchingPointcut。不过，注解配置方式与外部的配置方式并不冲突，只要合理利用，二者都可以发挥更大的作用。稍后，我们就可以看到这两种方式的强大结合。

4. ComposablePointcut

之前在介绍Pointcut概念的时候，我们说过，Pointcut通常还提供逻辑运算功能，而ComposablePointcut就是Spring AOP提供的可以进行Pointcut逻辑运算的Pointcut实现。它可以进行Pointcut之间的“并”以及“交”运算，如：

```
ComposablePointcut pointcut1 = new ComposablePointcut(classFilter1,methodMatcher1);
ComposablePointcut pointcut2 = ...;

ComposablePointcut unitedPoincut        = pointcut1.union(pointcut2);
ComposablePointcut intersectionPointcut = pointcut1.intersection(unitedPoincut);

assertEquals(pointcut1,intersectionPointcut);
```

我们说了，Pointcut定义根据ClassFilter和MethodMatcher划分为两部分，一部分是为了重用这些定义，另一部分是为了可以相互组合。而通过ComposablePointcut，我们可以很明白地看出这两个目的，以下代码演示了这一点：

```
ComposablePointcut pointcut3 = pointcut2.union(classFilter1).intersection(methodMatcher1);
```

我们在pointcut1和pointcut3中复用了classFilter1和methodMatcher1以及pointcut2的定义，同时，还进行了Pointcut同ClassFilter以及MethodMatcher之间的逻辑组合运算。

当然，如果只想进行Pointcut与Pointcut之间的逻辑组合运算，Spring AOP提供了org.springframework.aop.support.Pointcuts工具类，其简单使用示例如下所示：

```
Pointcut pointcut1 = ...;
Pointcut pointcut2 = ...;

Pointcut unitedPoincut        = Pointcuts.union(pointcut1,pointcut2);
Pointcut intersectionPointcut = Pointcuts.intersection(pointcut1,pointcut2);
```

5. ControlFlowPointcut

较之其他类型的Pointcut类型，最最最特殊的Pointcut类型，ControlFlowPointcut在理解和使用上都需要我们多付出点儿脑细胞。虽然ControlFlowPointcut不是很常用，但某些场合可能需要用到，所以，还是应搞清楚这个Pointcut的特点和使用方式。

ControlFlowPointcut匹配程序的调用流程，不是对某个方法执行所在的Joinpoint处的单一特征进行匹配。

假设我们所拦截的目标对象（Target Object）有方法声明如代码清单9-3所示。

代码清单9-3 TargetObject及其调用类定义

```
public class  TargetObject
{
   public void method1()
   {
      ...
   }
   ...
}
```

```
调用类
public class TargetCaller
{
   private TargetObject target;

   public void callMethod()
   {
      target.method1();
   }
   public void setTarget(TargetObject target)
   {
      this.target = target;
   }
}
```

如果使用之前的任何Pointcut实现，我们只能指定在TargetObject的method1方法每次执行的时候，都织入相应横切逻辑。也就是说，一旦通过Pointcut指定method1处为Joinpoint，那么对该方法的执行进行拦截是必定的，不管method1是被谁调用。而通过ControlFlowPointcut，我们可以指定，只有当TargetObject的method1方法在TargetCaller类所声明的方法中被调用的时候，才对method1方法进行拦截，其他地方调用method1的话，不对method1进行拦截。如图9-3所示。

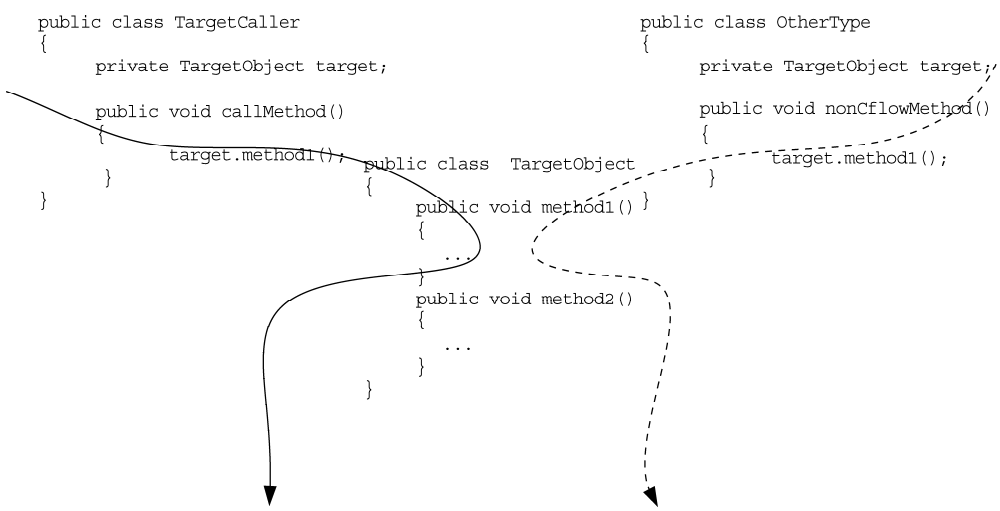

图9-3 程序的调用流程

虽然method1可以被多个对象在不同的执行流程内被调用，但是通过ControlFlowPointcut，我们可以指定，只有按照图9-3中实线所标注的执行流程，才会触发method1处的Joinpoint，而其他经过method1这个Joinpoint处的程序调用执行流程，则不会触发该Joinpoint处的横切逻辑织入操作。

使用ControlFlowPointcut，我们对以上需求的Pointcut表述，如代码清单9-4所示。

代码清单9-4 ControlFlowPointcut的使用示例

```
ControlFlowPointcut pointcut = new ControlFlowPointcut(TargetCaller.class);
Advice advice = ...;

TargetObject target = new TargetObject();
```

```
TargetObject targetObjectToUse = weaver.weave(advice).to(target).accordingto(pointcut);

// advice的逻辑在这里将不会被触发执行
targetObjectToUse.method1();

// advice的逻辑在这里将被触发执行
// 因为TargetCaller的callMethod()将调用method1
// 正像ControlFlowPointcut(TargetCaller.class)所指定的那样
TargetCaller caller = new TargetCaller();
caller.setTarget(targetObjectToUse);
caller.callMethod();
```

当织入器按照Pointcut的规定，将Advice织入到目标对象之后，从任何其他地方调用method1，是不会触发Advice所包含的横切逻辑的执行的。只有在ControlFlowPointcut规定的类内部调用目标对象的method1，才会触发Advice中横切逻辑的执行。

注意 实例代码中的weaver为虚构概念，不是Spring中的织入器实现。稍后马上会为你奉上Spring AOP使用什么原理以及使用什么工具作为织入器的相关内容。

如果在ControlFlowPointcut的构造方法中单独指定Class类型的参数，那么ControlFlowPointcut将尝试匹配指定的Class中声明的所有方法，跟目标对象的Joinpoint处的方法流程组合。所以，如果只是想完成“TargetCaller.callMethod()调用TargetObject.method1()”这样的流程匹配，而忽略TargetCaller中其他方法与TargetObject中方法的Control Flow匹配，我们可以同时在构造ControlFlowPointcut的时候，传入第二个参数，即调用方法的名称：

```
ControlFlowPointcut pointcut = new ControlFlowPointcut(TargetCaller.class,"callMethod");
```

我们的TargetCaller现在就声明了一个方法（即callMethod）调用了TargetObject的方法，所以，new ControlFlowPointcut(TargetCaller.class,"callMethod");形式的声明与newControlFlowPointcut(TargetCaller.class);在当前场景下会产生相同的效果。

因为ControlFlowPointcut类型的Pointcut需要在运行期间检查程序的调用栈，而且每次方法调用都需要检查，所以性能比较差。如果不是十分必要，应该尽量避免这种Pointcut的使用。

9.2.2 扩展 Pointcut（Customize Pointcut）

虽然我认为以上Spring AOP提供的Pointcut已经足够使用，但却无法保证一定就没有更加特殊的需求，以致于以上Pointcut类型都无法满足要求。这种情况下，我们可以扩展Spring AOP的Pointcut定义，给出自定义的Pointcut实现。

要自定义Pointcut，不用白手起家，Spring AOP已经提供了相应的扩展抽象类支持，我们只需要继承相应的抽象父类，然后实现或者覆写相应方法逻辑即可。前面已经讲过，Spring AOP的Pointcut类型可以划分为StaticMethodMatcherPointcut和DynamicMethodMatcherPointcut两类。要实现自定义Pointcut，通常在这两个抽象类的基础上实现相应子类即可。

1. 自定义StaticMethodMatcherPointcut

StaticMethodMatcherPointcut根据自身语意，为其子类提供了如下几个方面的默认实现。

- 默认所有StaticMethodMatcherPointcut的子类的ClassFilter均为ClassFilter. TRUE，即忽略类的类型匹配。如果具体子类需要对目标对象的类型做进一步限制，可以通过public void setClassFilter(ClassFilter classFilter)方法设置相应的ClassFilter实现。

- 因为是StaticMethodMatcherPointcut，所以，其MethodMatcher的isRuntime方法返回false，同时三个参数的matches方法抛出UnsupportedOperationException异常，以表示该方法不应该被调用到。

最终，我们需要做的就是实现两个参数的matches方法了。

如果我们想提供一个Pointcut实现，捕捉系统里数据访问层的数据访问对象中的查询方法所在的Joinpoint，那么可以实现一个StaticMethodMatcherPointcut，如下：

```
public class QueryMethodPointcut extends StaticMethodMatcherPointcut {
   public boolean matches(Method method, Class clazz) {
      return method.getName().startsWith("get")
           && clazz.getPackage().getName().startsWith("...dao");
   }
}
```

很简单，不是吗？

注意 使用现有的Pointcut类型完全可以满足相同的需求，所以在实现自定义的Pointcut之前，务必先查看一下是否已经有可用的Pointcut实现！

2. 自定义DynamicMethodMatcherPointcut

DynamicMethodMatcherPointcut也为其子类提供了部分便利。

(1) getClassFilter()方法返回ClassFilter.TRUE，如果需要对特定的目标对象类型进行限定，子类只要覆写这个方法即可。

(2) 对应的MethodMatcher的isRuntime总是返回true，同时，StaticMethodMatcherPointcut提供了两个参数的matches方法的实现，默认直接返回true。

要实现自定义DynamicMethodMatcherPointcut，通常情况下，我们只需要实现三个参数的matches方法逻辑即可。代码清单9-5给出了一个自定义的PKeySpecificQueryMethodPointcut实现类。

代码清单9-5 自定义的DynamicMethodMatcherPointcut实现示例

```
public class PKeySpecificQueryMethodPointcut extends DynamicMethodMatcherPointcut {
   public boolean matches(Method method, Class clazz, Object[] args) {
      if(method.getName().startsWith("get") ➥
      && clazz.getPackage().getName().startsWith("...dao"))
      {
         if(!ArrayUtils.isEmpty(args))
         {
            return StringUtils.equals("12345", args[0].toString());
         }
      }
      return false;
   }
}
```

如果愿意，我们也可以覆写一下两个参数的matches方法，这样，不用每次都得到三个参数的matches方法执行的时候才检查所有的条件。

9.2.3 IoC 容器中的 Pointcut

Spring中的Pointcut实现都是普通的Java对象，所以，我们同样可以通过Spring的IoC容器来注册并

使用它们。

如果某个Pointcut自身需要某种依赖，可以通过IoC容器为其注入。或者如果容器中的某个对象需要依赖于某个Pointcut，也可以把这个Pointcut注入到依赖对象中。不过，通常在使用Spring AOP的过程中，不会直接将某个Pointcut注册到容器，然后公开给容器中的对象使用。这一点稍后将详细讲述。只不过，需要说明的就是，将各个Pointcut以独立的形式注册到容器使用是完全合情合理的，如下所示：

```
<bean id="nameMatchPointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">
   <property name="mappedNames">
      <list>
         <value>methodName1</value>
         <value>methodName2</value>
      </list>
   </property>
</bean>
```

9.3 Spring AOP 中的 Advice

Spring AOP加入了开源组织AOP Alliance（http://aopalliance.sourceforge.net/）[①]，目的在于标准化AOP的使用，促进各个AOP实现产品之间的可交互性。鉴于此，Spring中的Advice实现全部遵循AOP Alliance规定的接口。图9-4中就是Spring中各种Advice类型实现与AOP Alliance中标准接口之间的关系（Introduction型的Advice将单独讲解）。

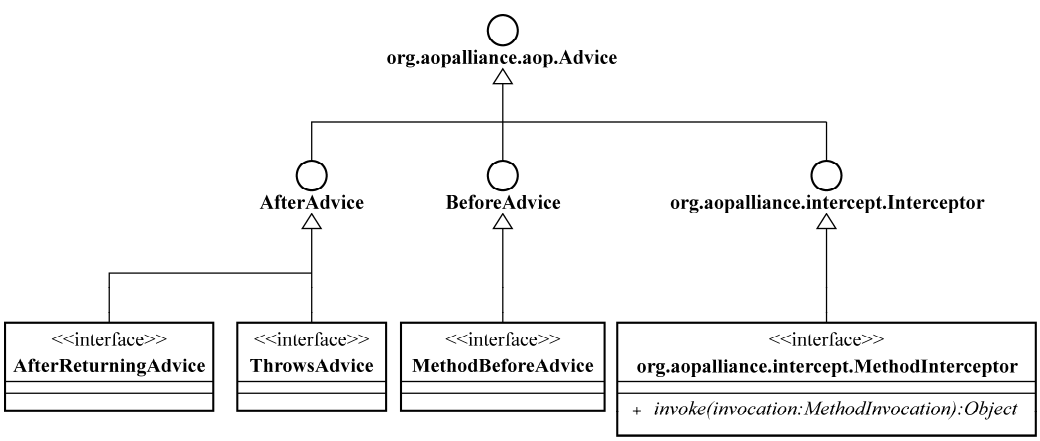

图9-4 Spring中Advice略图

Advice实现了将被织入到Pointcut规定的Joinpoint处的横切逻辑。在Spring中，Advice按照其自身实例（instance）能否在目标对象类的所有实例中共享这一标准，可以划分为两大类，即per-class类型的Advice和per-instance类型的Advice。

9.3.1 per-class 类型的 Advice

per-class类型的Advice是指，该类型的Advice的实例可以在目标对象类的所有实例之间共享。这种类型的Advice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。除

① 更确切地说，是Rod Johnson本人。

了图9-4 中没有列出的Introduction类型的Advice不属于per-class类型的Advice之外，图9-4中的所有Advice均属此列。

per-class类型的Advice将会是我们最常接触的Advice类型，所以，先从它们开刀！

1. Before Advice

本着“由简入奢”，哦，不，是“由简入深”的原则，我们先从最简单的Advice类型——Before Advice说起。

Before Advice所实现的横切逻辑将在相应的Joinpoint之前执行，在Before Advice执行完成之后，程序执行流程将从Joinpoint处继续执行，所以Before Advice通常不会打断程序的执行流程。但是如果必要，也可以通过抛出相应异常的形式中断程序流程。

要在Spring中实现Before Advice，我们通常只需要实现org.springframework.aop.MethodBeforeAdvice接口即可，该接口定义如下：

```
public interface MethodBeforeAdvice extends BeforeAdvice {
   void before(Method method, Object[] args, Object target) throws Throwable;
}
```

就像我们在图9-4中所看到的那样，MethodBeforeAdvice继承了BeforeAdvice，而BeforeAdvice与AOP Alliance的Advice一样，都是标志接口，其中没有定义任何方法。

注意 org.springframework.aop.BeforeAdvice接口没有定义方法的另一个原因在于考虑到将来的可扩展性。如果必要，可以引入支持属性级别拦截的Before Advice支持。

我们可以使用Before Advice进行整个系统的某些资源初始化或者其他一些准备性的工作。当然，其应用场景并非仅限于此，各位可以根据具体情况选择是否使用Before Advice。

假设我们的系统需要在文件系统的指定位置生成一些数据文件（系统实现中可能存在多处这样的位置），创建之前，我们需要首先检查这些指定位置是否存在，不存在则需要去创建它们。为了避免不必要的代码散落，我们可以为系统中相应目标类提供一个Before Advice，对文件系统的指定路径进行统一的检查或者初始化。代码清单9-6给出了用于初始化指定的资源路径的ResourceSetupBeforeAdvice定义。

代码清单9-6 ResourceSetupBeforeAdvice定义

```
public class ResourceSetupBeforeAdvice implements MethodBeforeAdvice {

   private Resource resource;
   public ResourceSetupBeforeAdvice(Resource resource)
   {
      this.resource = resource;
   }
   public void before(Method method, Object[] args, Object target)
         throws Throwable {
      if(!resource.exists())
      {
         FileUtils.forceMkdir(resource.getFile());
      }
   }
}
```

2. ThrowsAdvice

Spring中以接口定义org.springframework.aop.ThrowsAdvice对应通常AOP概念中的After-ThrowingAdvice。虽然该接口没有定义任何方法，但是在实现相应的ThrowsAdvice的时候，我们的方法定义需要遵循如下规则：

```
void afterThrowing([Method, args, target], ThrowableSubclass);
```

其中，[]中的三个参数可以省略。我们可以根据将要拦截的Throwable的不同类型，在同一个ThrowsAdvice中实现多个afterThrowing方法。框架将会使用Java反射机制（Java Reflection）来调用这些方法，如代码清单9-7所示。

代码清单9-7 声明了多个afterThrowing方法的ExceptionBarrierThrowsAdvice定义

```
public class ExceptionBarrierThrowsAdvice implements ThrowsAdvice {

   public void afterThrowing(Throwable t)
   {
      // 普通异常处理逻辑
   }

   public void afterThrowing(RuntimeException e)
   {
      // 运行时异常处理逻辑
   }

   public void afterThrowing(Method m,Object[] args,Object target,ApplicationException e)
   {
      // 处理应用程序生成的异常
   }
   ...
}
```

ThrowsAdvice通常用于对系统中特定的异常情况进行监控，以统一的方式对所发生的异常进行处理，我们可以在一种称之为Fault Barrier的模式中使用它。当然，我们更应该根据具体应用的场景来发挥ThrowsAdvice的最大能量。

能够马上想到的例子可能就是对系统中的运行时异常（RuntimeException）进行监控，一旦捕捉到异常，需要马上以某种方式通知系统的监控人员或者运营人员。假设我们通过email的方式发送通知，我们可以实现如代码清单9-8所示的一个ThrowsAdvice。

代码清单9-8 ExceptionBarrierThrowsAdvice的定义

```
public class ExceptionBarrierThrowsAdvice implements ThrowsAdvice {
   private JavaMailSender mailSender;
   private String[]   receiptions;

   public void afterThrowing(Method m,Object[] args,Object target,RuntimeException e)
   {
        final String exceptionMessage = ExceptionUtils.getFullStackTrace(e);
        getMailSender().send(new MimeMessagePreparator(){
        public void prepare(MimeMessage message) throws Exception {
        MimeMessageHelper helper =
          new MimeMessageHelper(message);
        helper.setSubject("...");
```

```
            helper.setTo(getReceiptions());
            helper.setText(exceptionMessage);
            }});
        }

        public JavaMailSender getMailSender() {
                return mailSender;
        }

        public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
        }

        public String[] getReceiptions() {
                return receiptions;
        }

        public void setReceiptions(String[] receiptions) {
                this.receiptions = receiptions;
        }
    }
```

该ThrowsAdvice实现中使用了Spring为JavaMail服务提供的抽象层，我们将在后面详细介绍该特性。你可以在当前ExceptionBarrierThrowsAdvice的基础上进行扩展，如添加更多配置项，或者进一步丰富邮件内容。

3. AfterReturningAdvice

org.springframework.aop.AfterReturningAdvice接口定义了Spring的AfterReturningAdvice，其定义如下：

```
public interface AfterReturningAdvice extends AfterAdvice {
   void afterReturning(Object returnValue, Method method, Object[] args, Object target) ➥
   throws Throwable;
}
```

通过Spring中的AfterReturningAdvice，我们可以访问当前Joinpoint的方法返回值、方法、方法参数以及所在的目标对象。

因为只有方法正常返回的情况下，AfterReturningAdvice才会执行，所以用来处理资源清理之类的工作并不合适。不过，如果有需要方法成功执行后进行的横切逻辑，使用AfterReturningAdvice倒比较合适。另外，虽然Spring的AfterReturningAdvice可以访问到方法的返回值，但不可以更改返回值。这一点与通常的AfterReturningAdvice的特性有所出入。好在，如果真想这么做，通过其他Advice类型还是可以做到的，如稍后即将隆重推出的Spring中的Around Advice实现。

因为Spring中的AfterReturningAdvice不能对方法返回值进行更改，所以其应用场景大受影响。不过，或多或少，还是可以寻找到某些应用的场景的。想一个更有魅力的示例有些费脑筋，所以，直接拿我遇到的场景来说吧。为了便于运营人员验证系统状态，FX的批处理程序在正常完成之后会往数据库的指定表中插入运行状态，运营人员可以通过验证这些状态判断相应的批处理任务是否成功执行，所以，我们可以实现一个AfterReturningAdvice对所有批处理任务的执行进行拦截。该AfterReturningAdvice实现见代码清单9-9。

代码清单9-9 TaskExecutionAfterReturningAdvice的定义

```
public class TaskExecutionAfterReturningAdvice implements AfterReturningAdvice {
   private SqlMapClientTemplate sqlMapClientTemplate;
```

```
    public void afterReturning(Object returnValue, Method m, Object[] args, ➥
    Object target) throws Throwable {
      Class clazz = target.getClass();
      getSqlMapClientTemplate().insert("BATCH.insertTaskStatus",clazz.getName());
    }

    public SqlMapClientTemplate getSqlMapClientTemplate() {
      return sqlMapClientTemplate;
    }

    public void setSqlMapClientTemplate(SqlMapClientTemplate sqlMapClientTemplate) {
      this.sqlMapClientTemplate = sqlMapClientTemplate;
    }
}
```

很简单，是吧！不过，如果不用AOP的话，散落到各个批处理任务中的类似逻辑可就不是两行代码可以搞定的了。

4. Around Advice

Spring AOP没有提供`After(Finally)Advice`，使得我们没有一个合适的Advice类型来承载类似于系统资源清除之类的横切逻辑。Spring AOP的`AfterReturningAdvice`不能更改Joinpoint所在方法的返回值，使得我们在方法正常返回后无法对其进行更多干预。不过，有了Around Advice，这些问题就都不是问题了。

Spring中没有直接定义对应Around Advice的实现接口，而是直接采用AOP Alliance的标准接口，即`org.aopalliance.intercept.MethodInterceptor`，该接口定义如下：

```
public interface MethodInterceptor extends Interceptor {
  Object invoke(MethodInvocation invocation) throws Throwable;
}
```

当然，从本节一开始的UML类图（图9-4）中就可以看出这个苗头了。

`MethodInterceptor`作为Around Advice那可是神通广大！之前提到的几种Advice能完成的事情，对于`MethodInterceptor`来说简直是不在话下。所以，`MethodInterceptor`，或者说Around Advice可以应用的场景那是相当地多啊！系统安全验证及检查、系统各处的性能检测、简单的日志记录[①]以及系统附加行为的添加等。

为了演示`MethodInterceptor`的使用以及需要注意的问题，我们以“简单的检测系统某些方法的执行性能”为例，实现一个`PerformanceInterceptor`，如代码清单9-10所示。

代码清单9-10 `PerformanceMethodInterceptor`的定义

```
public class PerformanceMethodInterceptor implements MethodInterceptor {
  private final Log logger = LogFactory.getLog(this.getClass());
  public Object invoke(MethodInvocation invocation) throws Throwable {
    StopWatch watch = new StopWatch();
    try
```

① 虽然大多数讲述AOP的资料或者书籍都喜欢以日志记录作为例子，但是那只是理论上的，在实际的操作过程中，实现的难度却比较大。因为系统中日志记录点很多，位置也很灵活，不可能只限于方法调用的开始或者结束等位置，即使像AspectJ这样支持很多Joinpoint类型的AOP实现产品，也无法保证能够捕捉到程序流程中的任何一个点。而且，日志记录需要的某些特定的上下文信息在取得的方式上也无定规。这两点造成日志记录无法完全的以AOP的方式进行，但简单的日志记录功能以现有的AOP产品支持还是可以的。

```
        {
          watch.start();
          return invocation.proceed();
        }
        finally
        {
          watch.stop();
          if(logger.isInfoEnabled())
          {
            logger.info(watch.toString());
          }
        }
      }
    }
```

通过MethodInterceptor的invoke方法的MethodInvocation参数，我们可以控制对相应Joinpoint的拦截行为。通过调用MethodInvocation的proceed()方法，可以让程序执行继续沿着调用链传播，这是通常我们所希望的行为。如果我们在哪一个MethodInterceptor中没有调用proceed()，那么程序的执行将会在当前MethodInterceptor处“短路”，Joinpoint上的调用链将被中断，同一Joinpoint上的其他MethodInterceptor的逻辑以及Joinpoint处的方法逻辑将不会被执行。除非你真的知道自己在做什么，否则，不要忘记调用proceed()方法哦！另外，我们还可以通过MethodInvocation对象取得Joinpoint的更多信息。

正如在PerformanceMethodInterceptor中所看到的那样，我们可以在proceed()方法，也就是Joinpoint处的逻辑执行之前或者之后插入相应的逻辑，甚至捕获proceed()方法可能抛出的异常。到现在，你是否可以理解为什么MethodInterceptor可以完成其他类型Advice可以完成的任务了呢？

为了进一步演示MethodInterceptor的使用，我们可以设想这样的场景： 如果某个销售系统规定，在商场优惠期间，所有商品一律8折出售（或者其他折扣条件），那么我们就应该在系统中所有取得商品价格的位置插入这样的横切逻辑。之前使用AfterReturningAdvice无法做的事情，现在我们使用Spring的Around Advice，也就是MethodInterceptor来做好了。代码清单9-11中声明的DiscountMethodInterceptor即用于此目的。

代码清单9-11 DiscountMethodInterceptor类的定义

```
Public class DiscountMethodInterceptor implements MethodInterceptor {
  private static final Integer  DEFAULT_DISCOUNT_RATIO = 80;
  private static final IntRange RATIO_RANGE = new IntRange(5,95);

  private Integer discountRatio = DEFAULT_DISCOUNT_RATIO;
  private boolean campaignAvailable;

  public Object invoke(MethodInvocation invocation) throws Throwable {
    Object returnValue = invocation.proceed();
    if(RATIO_RANGE.containsInteger(getDiscountRatio()) && isCampaignAvailable())
    {
      return ((Integer)returnValue)*getDiscountRatio()/100;
    }
    return returnValue;
  }

  private boolean isCampaignAvailable() {
    return campaignAvailable;
  }
```

```
    public void setCampaignAvailable(boolean campaignAvailable) {
      this.campaignAvailable = campaignAvailable;
    }

    public Integer getDiscountRatio() {
      return discountRatio;
    }

    public void setDiscountRatio(Integer discountRatio) {
      this.discountRatio = discountRatio;
    }
}
```

我们可以直接通过编程的方式来使用该类，如以下代码所示：

```
DiscountMethodInterceptor interceptor = new DiscountMethodInterceptor();
interceptor.setCampaignAvailable(true);
interceptor.setDiscountRatio(90);
...
// 现在可以将其添加到相应的Aspect中使用
```

既然我们使用了Spring框架并且这些Advice实现都是普通的POJO，更多时候，会直接将其集成到IoC容器中，如下所示：

```
<bean id="discountInterceptor" class="...DiscountMethodInterceptor">
   <property name="campaignAvailable" value="true"/>
   <property name="discountRatio" value="90"/>
</bean>
```

好啦！关于MethodInterceptor，我能说的就这些了。关于MethodInterceptor的更多应用场景，还有待你去挖掘……

9.3.2 per-instance 类型的 Advice

与per-class类型的Advice不同，per-instance类型的Advice不会在目标类所有对象实例之间共享，而是会为不同的实例对象保存它们各自的状态以及相关逻辑。就拿上班族为例（或许是比较痛苦的例子，呵呵），如果员工是一类人的话，那么公司的每一名员工就是员工类的不同对象实例。每个员工上班之前，公司设置了一个per-class类型的Advice进行“上班活动”的一个拦截，即打卡机，所有的员工都公用一个打卡机。当每个员工进入各自的位置之后，他们就会使用各自的电脑进行工作，而他们各自的电脑就好像per-instance类型的Advice一样，每个电脑保存了每个员工自己的资料。

在Spring AOP中，Introduction就是唯一的一种per-instance型Advice。

Introduction

Introduction可以在不改动目标类定义的情况下，为目标类添加新的属性以及行为。这就好比我们开发人员，如果公司人员紧张，没有配备测试人员，那么，通常就会给我们扣上一顶“测试人员”的帽子，让我们同时进行系统的测试工作，实际上，你还是你，只不过多了点儿事情而已。

在Spring中，为目标对象添加新的属性和行为必须声明相应的接口以及相应的实现。这样，再通过特定的拦截器将新的接口定义以及实现类中的逻辑附加到目标对象之上。之后，目标对象（确切地说是目标对象的代理对象）就拥有了新的状态和行为。这个特定的拦截器就是org.springframework.aop.IntroductionInterceptor，其定义如下：

```
public interface IntroductionInterceptor extends MethodInterceptor, ➥
DynamicIntroductionAdvice {
}
```

```
public interface DynamicIntroductionAdvice extends Advice {
   boolean implementsInterface(Class intf);
}
```

IntroductionInterceptor继承了MethodInterceptor以及DynamicIntroductionAdvice。通过DynamicIntroductionAdvice，我们可以界定当前的IntroductionInterceptor为哪些接口类提供相应的拦截功能。通过MethodInterceptor，IntroductionInterceptor就可以处理新添加的接口上的方法调用了。毕竟，原来的目标对象不会处理自己认为没有的东西啊。另外，通常情况下，对于IntroductionInterceptor来说，如果是新增加的接口上的方法调用，不必去调用MethodInterceptor的proceed()方法。毕竟，当前位置已经是“航程”的终点了（当前被拦截的方法实际上就是整个调用链中要最终执行的唯一方法）。

如果把每个目标对象实例看作盒装牛奶生产线上的那一盒盒牛奶的话，那么生产合格证就是新的Introduction逻辑，而IntroductionInterceptor就是把这些生产合格证贴到一盒盒牛奶上的那个“人”。

因为Introduction较之其他Advice有些特殊，所以，我们有必要从总体上看一下Spring中对Introduction的支持结构，图9-5给出了Introduction相关的类图结构。

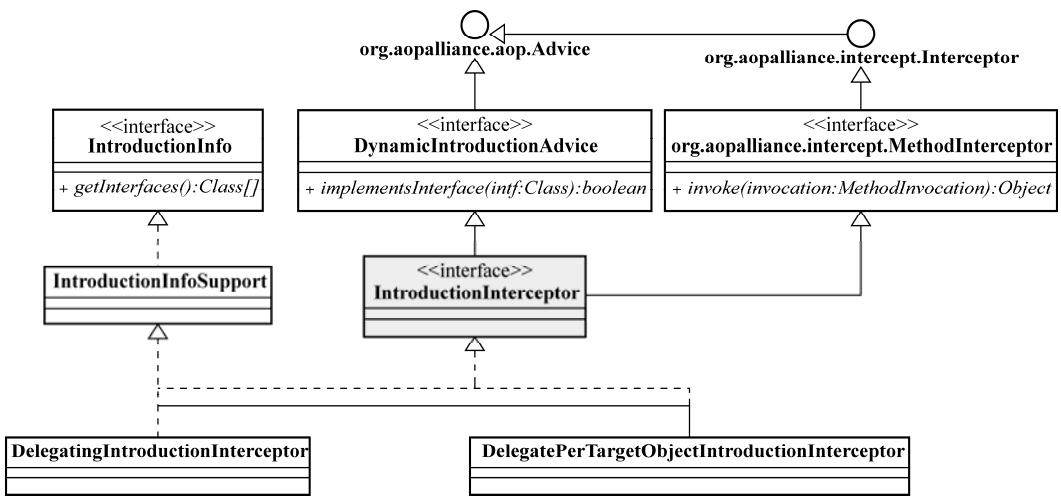

图9-5 Introduction相关类结构图

IntroductionInterceptor是从哪里出来的，前面已经讲过了，从图9-5中也可以看出相关的继承层次关系。我们要说的是实现Introduction型的Advice的两条分支，即以DynamicIntroductionAdvice为首的动态分支和以IntroductionInfo为首的静态可配置分支。从上面DynamicIntroductionAdvice的定义中可以看出，使用DynamicIntroductionAdvice，我们可以到运行时再去判定当前Introduction可应用到的目标接口类型，而不用预先就设定。而IntroductionInfo类型则完全相反，其定义如下：

```
public interface IntroductionInfo {
   Class[] getInterfaces();
}
```

实现类必须返回预定的目标接口类型，这样，在对IntroductionInfo型的Introduction进行织入

的时候，实际上就不需要指定目标接口类型了，因为它自身就带有这些必要的信息。

要对目标对象进行拦截并添加Introduction逻辑，我们可以直接扩展IntroductionInterceptor，然后在子类的invoke方法中实现所有的拦截逻辑。不过，除非特殊状况下需要去直接扩展IntroductionInterceptor，大多数时候，直接使用Spring提供的两个现成的实现类就可以了。

- DelegatingIntroductionInterceptor

从名字也可以看的出来，DelegatingIntroductionInterceptor不会自己实现将要添加到目标对象上的新的逻辑行为，而是委派（delegate）给其他实现类。不过这样也好，职责划分可以更加明确嘛！

就以简化的开发人员为例来说明DelegatingIntroductionInterceptor的用法吧！我们声明IDeveloper接口及其相关类，如代码清单9-12所示。

代码清单9-12 IDeveloper接口声明及其相关实现类定义

```
public interface IDeveloper {
   void developSoftware();
}

public class Developer implements IDeveloper {

   public void developSoftware() {
      System.out.println("I am happy with programming.");
   }
}
```

使用DelegatingIntroductionInterceptor为Developer添加新的状态或者行为，我们可以按照如下步骤进行。

(1) 为新的状态和行为定义接口。我们要为Developer添加测试人员的职能，首先需要将需要的职能以接口定义的形式声明。这样，就有了ITester声明，如下：

```
public interface ITester {
   boolean isBusyAsTester();
   void testSoftware();
}
```

(2) 给出新接口的实现类。接口实现类给出将要添加到目标对象的具体逻辑。当目标对象将要行使新的职能的时候，会通过该实现类寻求帮忙。代码清单9-13给出了针对ITester的实现类。

代码清单9-13 ITester的实现类定义

```
public class Tester implements ITester {
   private boolean busyAsTester;

   public void testSoftware() {
      System.out.println("I will ensure the quality.");
   }

   public boolean isBusyAsTester() {
      return busyAsTester;
   }

   public void setBusyAsTester(boolean busyAsTester) {
      this.busyAsTester = busyAsTester;
   }
}
```

我们可以在接口实现类中添加相应的属性甚至辅助方法，就跟实现通常的业务对象一样。

(3) 通过DelegatingIntroductionInterceptor进行Introduction的拦截。有了新增加职能的接口定义以及相应实现类，使用DelegatingIntroductionInterceptor，我们就可以把具体的Inroduction拦截委托给具体的实现类来完成，如下代码演示了这一过程：

```
ITester delegate = new Tester()
DelegatingIntroductionInterceptor interceptor = new ➥
DelegatingIntroductionInterceptor(delegate);
// 进行织入
ITester tester = (ITester)weaver.weave(developer).with(interceptor).getProxy();
tester.testSoftware();
```

(4) Introduction的最终织入过程在细节上有需要注意的地方，我们将在后面提到。虽然，DelegatingIntroductionInterceptor是Introduction型Advice的一个实现，但你可能料想不到的是，它其实是个“伪军”，因为它的实现根本就没有兑现Introduction作为per-instance型Advice的承诺。实际上，DelegatingIntroductionInterceptor会使用它所持有的同一个“delegate”接口实例，供同一目标类的所有实例共享使用。你想啊，就持有一个接口实现类的实例对象，它往哪里去放对应各个目标对象实例的状态啊？所以，如果要真的想严格达到Introduction型Advice所宣称的那样的效果，我们不能使用DelegatingIntroductionInterceptor，而是要使用它的兄弟，DelegatePerTargetObjectIntroductionInterceptor。

- DelegatePerTargetObjectIntroductionInterceptor

与DelegatingIntroductionInterceptor不同，DelegatePerTargetObjectIntroductionInterceptor会在内部持有一个目标对象与相应Introduction逻辑实现类之间的映射关系。当每个目标对象上的新定义的接口方法被调用的时候，DelegatePerTargetObjectIntroductionInterceptor会拦截这些调用，然后以目标对象实例作为键，到它持有的那个映射关系中取得对应当前目标对象实例的Introduction实现类实例。剩下的当然就是，让当前目标对象实例吃自己家锅里的饭了。如果根据当前目标对象实例没有找到对应的Introduction实现类实例，DelegatePerTargetObjectIntroductionInterceptor将会为其创建一个新的，然后添加到映射关系中。

使用DelegatePerTargetObjectIntroductionInterceptor与使用DelegatingIntroductionInterceptor没有太大的差别，唯一的区别可能就在于构造方式上。现在我们不是自己构造delegate接口实例，而只需要告知DelegatePerTargetObjectIntroductionInterceptor相应的delegate接口类型和对应实现类的类型。剩下的工作留给DelegatePerTargetObjectIntroductionInterceptor就可以了，如以下代码所示：

```
DelegatePerTargetObjectIntroductionInterceptor interceptor = ➥
new DelegatePerTargetObjectIntroductionInterceptor(DelegateImpl.class,IDelegate.class);
```

当然啦，如果DelegatingIntroductionInterceptor和DelegatePerTargetObjectIntroductionInterceptor默认的invoke方法实现逻辑无法满足你的需求，你也可以直接扩展这两个类，覆写（Override）相应的方法。不过，不知为什么，DelegatingIntroductionInterceptor和DelegatePerTargetObjectIntroductionInterceptor自身实现上对扩展有所限制，实例变量没有提供可以公开给子类的途径，一些应该声明为protected以便子类共享的方法也没有放开，而是声明为private。DelegatingIntroductionInterceptor倒是可以通过它的无参数的构造方法进行扩展，但要求子类必须同时实现新的Introduction逻辑的接口。DelegatePerTargetObjectIntroductionInterceptor干脆就没有发现什么有用的可扩展点。所以，给我的感觉就是，直接扩展这两个类跟直接扩展

IntroductionInterceptor相比，好像也没有太多优势。希望Spring Team之后能够修改这两个类，以便能够更方便地进行扩展。

要扩展IntroductionInterceptor或者DelegatingIntroductionInterceptor和DelegatePerTargetObjectIntroductionInterceptor，通常是因为目标对象的行为，与新附加到目标对象的状态和行为相关联。这时，在处理两方面的方法调用的时候，就需要根据情况添加新的调用处理逻辑——假设Developer要进行开发的时候，检测到其作为Tester本身也在忙活，Developer要“罢工”，我们可以实现拥有类似逻辑的IntroductionInterceptor实现，如代码清单9-14所示。

代码清单9-14 扩展DelegatingIntroductionInterceptor的示例

```
public class TesterFeatureIntroductionInterceptor extends ➥
DelegatingIntroductionInterceptor implements ITester
{
   private static final long serialVersionUID = -3387097489523045796L;
   private boolean busyAsTester;

   @Override
      public Object invoke(MethodInvocation mi) throws Throwable {
         if(isBusyAsTester()
         && StringUtils.contains(mi.getMethod().getName(), "developSoftware"))
         {
            throw new RuntimeException("你想累死我呀？");
         }
         return super.invoke(mi);
   }

   public boolean isBusyAsTester() {
      return busyAsTester;
   }

   public void setBusyAsTester(boolean busyAsTester) {
      this.busyAsTester = busyAsTester;
   }

   public void testSoftware() {
      System.out.println("I will ensure the quality.");
   }
}
```

最后要说的是Introduction的性能问题。与AspectJ直接通过编译器将Introduction织入目标对象不同，Spring AOP采用的是动态代理机制，在性能上，Introduction型的Advice要逊色不少。如果有必要，可以考虑采用AspectJ的Introduction实现。

9.4 Spring AOP 中的 Aspect

当所有的Pointcut和Advice准备好之后，就到了该把它们分门别类地装进箱子的时候了。你知道我说的箱子是什么，对吧？当然是Aspect。

在解释Aspect的概念的时候曾经提到过，Spring中最初没有完全明确的Aspect的概念，但是，这并不意味着就没有。只不过，Spring中的这个Aspect在实现和特性上有所特殊而已。

Advisor代表Spring中的Aspect，但是，与正常的Aspect不同，Advisor通常只持有一个Pointcut和一个Advice。而理论上，Aspect定义中可以有多个Pointcut和多个Advice，所以，我们可以认为Advisor是一种特殊的Aspect。

为了能够更清楚Advisor的实现结构体系，我们可以将Advisor简单划分为两个分支，一个分支以org.springframework.aop.PointcutAdvisor为首，另一个分支则以org.springframework.aop.IntroductionAdvisor为头儿，如图9-6所示。

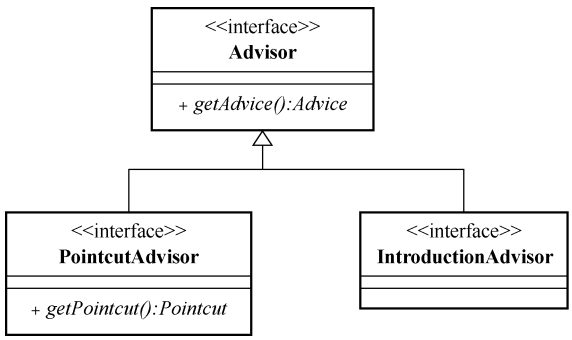

图9-6 Advisor分支

9.4.1 PointcutAdvisor 家族

实际上，org.springframework.aop.PointcutAdvisor才是真正的定义一个Pointcut和一个Advice的Advisor，大部分的Advisor实现全都是PointcutAdvisor的“部下”（见图9-7）。

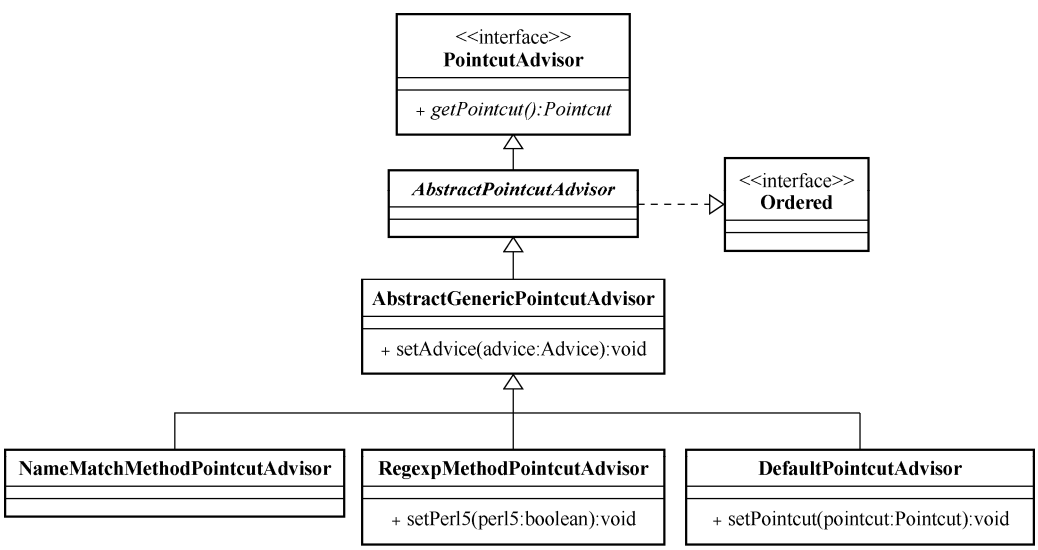

图9-7 PointcutAdvisor及相关子类

下面我们就来看一下几个常用的PointcutAdvisor实现。

1. DefaultPointcutAdvisor

DefaultPointcutAdvisor是PointcutAdvisor的“大弟子”，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DefaultPointcutAdvisor来使用。我们可以在构造DefaultPointcutAdvisor的时候，就明确指定属于当前DefaultPointcutAdvisor实例的Pointcut和Advice，也可以在DefaultPointcut-Advisor实例构造完成后，再通过setPointcut以及setAdvice方法设置相应的Pointcut和Advice（使用示例见代码清单9-15）。

代码清单9-15 DefaultPointcutAdvisor使用示例

```
Pointcut pointcut = ...;     // 任何Pointcut类型
Advice   advice   = ...;     // 除Introduction类型外的任何Advice类型

DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(pointcut,advice);
// 或者
DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor(advice);
advisor.setPointcut(pointcut);
// 或者
DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor();
advisor.setPointcut(pointcut);
advisor.setAdvice(advice);
```

此处给出代码并不是让你在实际的环境中就这么用，而是为了演示事实的真相。实际上，Spring中任何的bean都可以通过IoC容器来管理，Spring AOP中的任何概念对此也同样适用。大多数时候，我们会通过IoC容器来注册和使用Spring AOP的各种概念实体。

通常使用Spring的IoC容器注册管理DefaultPointcutAdvisor的情形，如代码清单9-16所示。

代码清单9-16 通过IoC容器注册管理DefaultPointcutAdvisor相关类

```
<bean id="pointcut" class="...">
   ...
</bean>
<bean id="advice" class="...">
   ...
</bean>

<bean id="advisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
   <property name="pointcut" ref="pointcut"/>
   <property name="advice" ref="advice"/>
</bean>
```

2. NameMatchMethodPointcutAdvisor

NameMatchMethodPointcutAdvisor是细化后的DefaultPointcutAdvisor，它限定了自身可以使用的Pointcut类型为NameMatchMethodPointcut，并且外部不可更改。不过，对于使用的Advice来说，除了Introduction，其他任何类型的Advice都可以使用。

NameMatchMethodPointcutAdvisor内部持有一个NameMatchMethodPointcut类型的Pointcut实例。当通过NameMatchMethodPointcutAdvisor公开的setMappedName和setMappedNames方法设置将被拦截的方法名称的时候，实际上是在操作NameMatchMethodPointcutAdvisor所持有的这个NameMatchMethodPointcut实例。

NameMatchMethodPointcutAdvisor的使用也很简单，通过编程方式还是通过IoC容器都可以，编程方式使用示例如下：

```
Advice advice = ...; // 任何类型的Advice，Introduction类型除外
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor(advice);
advisor.setMappedName("methodName2Intercept");
```

```
// 或者
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor(advice);
advisor.setMappedNames(new String[]{"method1","method2"});
```

通过IoC容器使用的情形，见代码清单9-17。

代码清单9-17 通过IoC容器配置使用NameMatchMethodPointcutAdvisor

```
<bean id="advice" class="...">
   ...
</bean>

<bean id="advisor" class="org.springframework.aop.support. ➥
NameMatchMethodPointcutAdvisor">
   <property name="advice">
     <ref bean="advice"/>
   </property>
   <property name="mappedNames">
     <list>
       <value>method1</value>
       ...
     </list>
   </property>
</bean>
```

3. RegexpMethodPointcutAdvisor

与NameMatchMethodPointcutAdvisor类似，RegexpMethodPointcutAdvisor也限定了自身可以使用的Pointcut的类型，即只能通过正则表达式为其设置相应的Pointcut。

RegexpMethodPointcutAdvisor自身内部持有一个AbstractRegexpMethodPointcut的实例。希望你还记得，AbstractRegexpMethodPointcut有两个实现类，即Perl5RegexpMethodPointcut和JdkRegexpMethodPointcut。默认情况下，RegexpMethodPointcutAdvisor会使用JdkRegexpMethodPointcut。如果要强制使用Perl5RegexpMethodPointcut，那么可以通过RegexpMethodPointcutAdvisor的setPerl5(boolean)达成所愿。

RegexpMethodPointcutAdvisor提供了许多构造方法，我们可以在构造时就指定Pointcut的正则表达式匹配模式以及相应的Advice，也可以构造完成之后再指定，在使用上与其他的Advisor实现并无太多差别。我们这里只演示在IoC容器中的配置使用方式（见代码清单9-18）。

代码清单9-18 通过IoC容器配置使用RegexpMethodPointcutAdvisor示例

```
<bean id="advice" class="...">
   ...
</bean>
<bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
   <property name="pattern">
     <value>cn\.spring21\..*\.methodNamePattern</value>
   </property>
   <property name="advice">
     <ref bean="advice"/>
   </property>
   <property name="perl5">
     <value>false</value>
   </property>
</bean>
```

关于正则表达式的更多信息可以参照JDK 1.4或者更高版本的Javadoc以及Jakarta ORO项目文档。

4. DefaultBeanFactoryPointcutAdvisor

DefaultBeanFactoryPointcutAdvisor是使用比较少的一个Advisor实现，因为自身绑定到了BeanFactory，所以，要使用DefaultBeanFactoryPointcutAdvisor，我们的应用铁定要绑定到Spring的IoC容器了。而且，通常情况下，DefaultPointcutAdvisor已经完全可以满足需求。

DefaultBeanFactoryPointcutAdvisor的作用是，我们可以通过容器中的Advice注册的beanName来关联对应的Advice。只有当对应的Pointcut匹配成功之后，才去实例化对应的Advice，减少了容器启动初期Advisor和Advice之间的耦合性。

要使用DefaultBeanFactoryPointcutAdvisor，我们通常需要在容器的配置文件中进行如代码清单9-19所示的配置。

代码清单9-19 DefaultBeanFactoryPointcutAdvisor使用示例

```
<bean id="advice" class="...">
</bean>

<bean id="pointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">
   <property name="mappedName" value="doSth"/>
</bean>

<bean id="advisor" class="org.springframework.aop.support. ➥
DefaultBeanFactoryPointcutAdvisor">
   <property name="pointcut" ref="pointcut"/>
   <property name="adviceBeanName" value="advice"/>
</bean>
```

注意，对应advice的配置属性名称为“adviceBeanName”，而它的值就对应advice的beanName。除了这一点，与DefaultPointcutAdvisor使用并无二致。

9.4.2 IntroductionAdvisor 分支

IntroductionAdvisor与PointcutAdvisor最本质上的区别就是，IntroductionAdvisor只能应用于类级别的拦截，只能使用Introduction型的Advice，而不能像PointcutAdvisor那样，可以使用任何类型的Pointcut，以及差不多任何类型的Advice。也就是说，IntroductionAdvisor纯粹就是为Introduction而生的。

IntroductionAdvisor的类层次比较简单，只有一个默认实现DefaultIntroductionAdvisor[①]，其继承层次见图9-8。

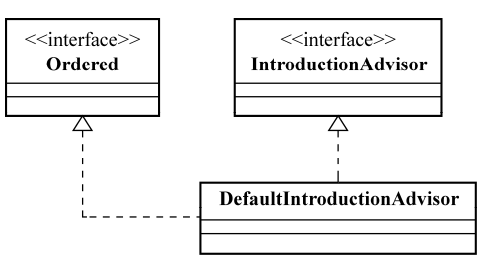

图9-8 IntroductionAdvisor类结构图

既然IntroductionAdvisor仅限于Introduction的使用场景，那么DefaultIntroductionAdvisor

① 另一个通过AspectJ继承扩展的实现，将在第10章介绍。

的使用也比较简单，只可以指定Introduction型的Advice（即IntroductionInterceptor）以及将被拦截的接口类型。其使用示例见代码清单9-20。

代码清单9-20 DefaultIntroductionAdvisor使用示例

```
<bean id="introductionInterceptor" class="org.springframework.aop.support. ➥
DelegatingIntroductionInterceptor">
   <constructor-arg>
     <bean class="...DelegateImpl">
     </bean>
   </constructor-arg>
</bean>

<bean id="introductionAdvisor" class="org.springframework.aop.support. ➥
DefaultIntroductionAdvisor">
   <constructor-arg><ref bean="introductionInterceptor"/></constructor-arg>
   <constructor-arg><value>...IDelegateInterface</value></constructor-arg>
</bean>
```

我们也可以指定Advice以及一个IntroductionInfo对象类来构造DefaultIntroductionAdvisor，因为IntroductionInfo可以提供必要的目标接口类型。代码清单9-21是结合IntroductionInfo使用的DefaultIntroductionAdvisor的使用示例。

代码清单9-21 结合IntroductionInfo使用的DefaultIntroductionAdvisor的使用示例

```
<bean id="introductionInterceptor" class="org.springframework.aop.support. ➥
DelegatingIntroductionInterceptor">
   <constructor-arg>
     <bean class="...DelegateImpl">
     </bean>
   </constructor-arg>
</bean>

<bean id="introductionAdvisor" class="org.springframework.aop.support. ➥
DefaultIntroductionAdvisor">
   <constructor-arg index="0"><ref bean="introductionInterceptor"/></constructor-arg>
   <constructor-arg index="1"><ref bean="introductionInterceptor"/></constructor-arg>
</bean>
```

不用奇怪为什么我们在构造DefaultIntroductionAdvisor的时候传入两个“introducetion-Interceptor”，它们两个其实是不一样的，前者是作为Introduction型的Advice实例，后者则是作为IntroductionInfo的实例。不要忘了DelegatingIntroductionInterceptor实现了Introduciton-Info接口哦！

9.4.3 Ordered 的作用

系统中只存在单一的横切关注点的情况很少，大多数时候，都会有多个横切关注点需要处理，那么，系统实现中就会有多个Advisor存在。当其中的某些Advisor的Pointcut匹配了同一个Joinpoint的时候，就会在这同一个Joinpoint处执行多个Advice的横切逻辑。如果这些Advisor所关联的Advice之间没有很强的优先级依赖关系，那么谁先执行，谁后执行都不会造成任何影响。而一旦这几个需要在同一Joinpoint处执行的Advice逻辑存在优先顺序依赖的话，就需要我们来干预了，否则，系统的行为就会偏离我们的预想。

记得有一天，同事大鹏突然问我，说：“头儿，这个任务初始化的时候抛出异常但没被我们的ThrowsAdvice截获，帮看一下呗？”我心里纳闷，不能吧？查了一下Pointcut的正则表达式定义，没错

啊，应该能捕获到啊。最后查到抛出异常的是应用到同一个方法的Advice所抛出的，我才猛然醒悟……

现在假设有两个Advisor，一个进行权限检查，当检查到当前调用没有权限的时候，抛出相应异常，称为PermissionAuthAdvisor；另一个Advisor使用一个ThrowsAdvice对系统中的所有需要检测的异常进行拦截，称其为ExceptionBarrierAdvisor。如果以如下形式声明这两个Advisor，就不会有问题：

```
<bean id="exceptionBarrierAdvisor" class="...ExceptionBarrierAdvisor">
   ...
</bean>

<bean id="pemissionAuthAdvisor" class="...PermissionAuthAdvisor">
   ...
</bean>
```

即使PermissionAuthAdvisor的Advice抛出异常，我们的ExceptionBarrierAdvisor也可以捕获该异常并进行系统内的统一处理。而如果我们像如下这样，颠倒它们两个的声明顺序，那就有问题了：

```
<bean id="pemissionAuthAdvisor" class="...PermissionAuthAdvisor">
   ...
</bean>

<bean id="exceptionBarrierAdvisor" class="...ExceptionBarrierAdvisor">
   ...
</bean>
```

在PermissionAuthAdvisor中的Advice抛出异常之后，ExceptionBarrierAdvisor并没有起作用，问题出在哪儿呢？

Spring在处理同一Joinpoint处的多个Advisor的时候，实际上会按照指定的顺序和优先级来执行它们，顺序号决定优先级，顺序号越小，优先级越高，优先级排在前面的，将被优先执行。我们可以从0或者1开始指定，因为小于0的顺序号原则上由Spring AOP框架内部使用。默认情况下，如果我们不明确指定各个Advisor的执行顺序，那么Spring会按照它们的声明顺序来应用它们，最先声明的顺序号最小但优先级最大，其次次之。

有了这些前提，我们就可以知道为什么仅颠倒两个Advisor的顺序就会造成某个Advisor失效。让我们来看图9-9。

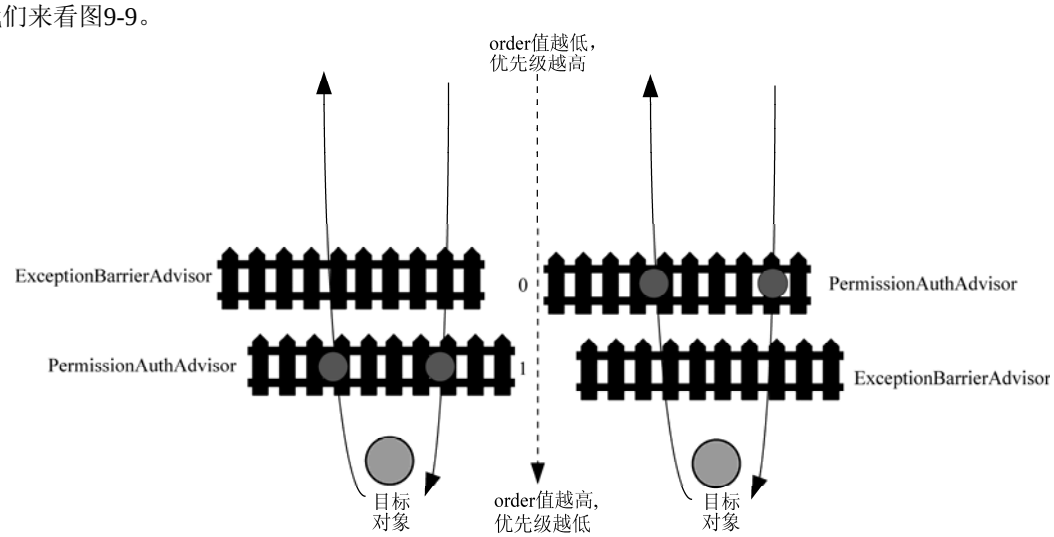

图9-9 Ordered作用示意图

在图9-9中，左边是正常的情况，当调用流程在PermissionAuthAdvisor中出现问题时，甚至是在目标对象上出现问题时，ExceptionBarrierAdvisor会在调用流程返回的时候捕获到相应异常；而右边就是调换顺序后的结果，可以看到现在PermissionAuthAdvisor上出现问题的话，因为调用流程已经经过了ExceptionBarrierAdvisor，所以，ExceptionBarrierAdvisor根本无法捕获PermissionAuthAdvisor上的异常（虽然目标对象上的问题可以捕获到）。

虽然我们可以通过调整配置中各个Advisor声明的顺序来避免以上问题，但是，这并非最彻底的解决方法。最彻底的方法就是，为每个Advisor明确指定顺序号。在Spring框架中，我们可以通过让相应的Advisor以及其他顺序紧要的bean实现org.springframework.core.Ordered接口来明确指定相应顺序号。不过，从图9-7中也应该看到了，各个Advisor实现类，其实已经实现了Ordered接口。我们无需自己去实现这个接口了，唯一要做的是直接在配置的时候指定顺序号。代码清单9-22中的配置为我们的两个Advisor指定了明确的顺序号，从而避免了最初问题的出现。

代码清单9-22 明确指定各个Advisor的顺序号的演示

```
<bean id="pemissionAuthAdvisor" class="...PermissionAuthAdvisor">
   <property name="order" value="1"/>
   ...
</bean>

<bean id="exceptionBarrierAdvisor" class="...ExceptionBarrierAdvisor">
   <property name="order" value="0"/>
   ...
</bean>
```

9.5 Spring AOP 的织入

俗话说得好，“万事俱备，只欠东风”！各个模块我们已经实现好了，剩下的工作，就是拼装各个模块。

要进行织入，AspectJ采用ajc编译器作为它的织入器；JBoss AOP使用自定义的ClassLoader作为它的织入器；而在Spring AOP中，使用类org.springframework.aop.framework.ProxyFactory作为织入器。

9.5.1 如何与 ProxyFactory 打交道

首先需要声明的是，ProxyFactory并非Spring AOP中唯一可用的织入器，而是最基本的一个织入器实现，所以，我们就从最基本的这个织入器开始，来窥探一下Spring AOP的织入过程到底是一个什么样子。

使用ProxyFactory来进行横切逻辑的织入很简单。我们知道，Spring AOP是基于代理模式的AOP实现，织入过程完成后，会返回织入了横切逻辑的目标对象的代理对象。为ProxyFactory提供必要的“生产原材料”之后，ProxyFactory就会返回那个织入完成的代理对象（如以下代码所示）：

```
ProxyFactory weaver = new ProxyFactory(yourTargetObject);
// 或者
// ProxyFactory weaver = new ProxyFactory();
// weaver.setTarget(task);
Advisor advisor = ...;
weaver.addAdvisor(advisor);
Object proxyObject = weaver.getProxy();
// 现在可以使用proxyObject了
```

使用ProxyFactory只需要指定如下两个最基本的东西。

- 第一个是要对其进行织入的目标对象。我们可以通过ProxyFactory的构造方法直接传入，也可以在ProxyFactory构造完成之后，通过相应的setter方法进行设置。
- 第二个是将要应用到目标对象的Aspect。哦，在Spring里面叫做Advisor，呵呵。不过，除了可以指定相应的Advisor之外，还可以使用如下代码，直接指定各种类型的Advice。

```
weaver.addAdvice(...);
```

  - 对于Introduction之外的Advice类型，ProxyFactory内部就会为这些Advice构造相应的Advisor，只不过在为它们构造的Advisor中使用的Pointcut为Pointcut.TRUE，即这些“没穿衣服”的Advice将被应用到系统中所有可识别的Joinpoint处;
  - 而如果添加的Advice类型是Introduction类型，则会根据该Introduction的具体类型进行区分：如果是IntroductionInfo的子类实现，因为它本身包含了必要的描述信息，框架内部会为其构造一个DefaultIntroductionAdvisor；而如果是DynamicIntroductionAdvice的子类实现，框架内部将抛出AopConfigException异常（因为无法从DynamicIntroductionAdvice取得必要的目标对象信息）。

但是，在不同的应用场景下，我们可以指定更多ProxyFactory的控制属性，以便让ProxyFactory帮我们生成必要的代理对象。我们知道，Spring AOP在使用代理模式实现AOP的过程中采用了动态代理和CGLIB两种机制，分别对实现了某些接口的目标类和没有实现任何接口的目标类进行代理，所以，在使用ProxyFactory对目标类进行代理的时候，会通过ProxyFactory的某些行为控制属性对这两种情况进行区分。

在继续下面内容之前，有必要先设定一个简单的场景，以便大家结合实际情况来查看和分析在不同场景下，ProxyFactory在使用方式上的细微差异。假设我们的目标类型定义如下：

```
public interface ITask {
   void execute(TaskExecutionContext ctx);
}

public class MockTask implements ITask {

   public void execute(TaskExecutionContext ctx) {
      System.out.println("task executed.");
   }
}
```

有了要拦截的目标类，还得有织入到Joinpoint处的横切逻辑，也就是要用到某个Advice实现。我们就把之前的PerformanceMethodInterceptor先拿来一用（见代码清单9-23）。

代码清单9-23 PerformanceMethodInterceptor定义

```
public class PerformanceMethodInterceptor implements MethodInterceptor {
   private final Log logger = LogFactory.getLog(this.getClass());
   public Object invoke(MethodInvocation invocation) throws Throwable {
      StopWatch watch = new StopWatch();
      try
      {
         watch.start();
         return invocation.proceed();
      }
      finally
      {
         watch.stop();
```

```
            if(logger.isInfoEnabled())
            {
              logger.info(watch.toString);
            }
          }
        }
    }
```

有了这些之后，让我们来看一下使用ProxyFactory对实现了ITask接口的目标类，以及没有实现任何接口的目标类如何进行代理。

1. 基于接口的代理

MockTask实现了ITask接口，要对这种实现了某些接口的目标类进行代理，我们可以为ProxyFactory明确指定代理的接口类型，如下所示：

```
MockTask task = new MockTask();
ProxyFactory weaver = new ProxyFactory(task);
weaver.setInterfaces(new Class[]{ITask.class});
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
ITask proxyObject =(ITask)weaver.getProxy();
proxyObject.execute(null);
```

通过setInterfaces()方法可以明确告知ProxyFactory，我们要对ITask接口类型进行代理。另外，在这里，我们通过NameMatchMethodPointcutAdvisor来指定Pointcut和相应的Advice（PerformanceMethodInterceptor）。至于什么类型的Pointcut、Advice以及Advisor，我们完全可以根据个人的喜好或者具体场景来使用，举一反三嘛！

不过，如果没有其他行为属性的干预，我们也可以不使用setInterfaces()方法明确指定具体的接口类型。这样，默认情况下，ProxyFactory只要检测到目标类实现了相应的接口，也会对目标类进行基于接口的代理，如下所示：

```
MockTask task = new MockTask();
ProxyFactory weaver = new ProxyFactory(task);
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
ITask proxyObject =(ITask)weaver.getProxy();
proxyObject.execute(null);
```

这两种形式最终的结果是等效的：

```
task executed.
60 [main] INFO ...PerformanceMethodInterceptor  - 0:00:00.000
```

简单点儿说，如果目标类实现了至少一个接口，不管我们有没有通过ProxyFactory的setInterfaces()方法明确指定要对特定的接口类型进行代理，只要不将ProxyFactory的optimize和proxyTargetClass两个属性的值设置为true（这两个属性稍后将谈到），那么ProxyFactory都会按照面向接口进行代理。

看你是否真正了解了动态代理

在我们对MockTask实例进行代理之后，我们通过如下代码将取得的代理对象强制转型为

ITask，然后执行execute，那么，将取得代理对象强制转型为MockTask是否可以呢？

```
ITask proxyObject =(ITask)weaver.getProxy();
```

如果你不确定，可以把这行代码改一下试试。呵呵，答案是不可以，程序最终将抛出java.lang.ClassCastException异常。我想，你已经猜到问题出在哪里了。

请回头看一下图8-1，在代理模式的场景中，接口的具体实现类和这个具体实现类的代理对象是两个不同的对象，我们可以将接口实现类和它的代理对象都强制转型为接口类型，但是无法将代理对象类型强制转型为接口实现类类型。到我们的场景中就是，MockTask可以强制转型为ITask，MockTask的代理对象也可以强制转型为ITask，但是，要将代理对象强制转型为MockTask，一定会出问题的。

如果我们在代码最后再添加如下一行代码，来查看代码中proxyObject的类型的话：

```
System.out.println(proxyObject.getClass());
```

就会发现结果是：

```
class $Proxy0
```

(MockTask)java.lang.reflect.Proxy这样的强制转型，你觉得能行吗？

2. 基于类的代理

如果目标类没有实现任何接口，那么，默认情况下，ProxyFactory会对目标类进行基于类的代理，即使用CGLIB。假设我们现在有一个对象，定义如下：

```
public class Executable {
   public void execute(){
      System.out.println("Executable without any Interfaces");
   }
}
```

如果使用Executable作为目标对象类，那么，ProxyFactory就会对其进行基于类的代理，如下代码演示了使用ProxyFactory对Executable进行织入的过程：

```
ProxyFactory weaver = new ProxyFactory(new Executable());
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
Executable proxyObject =(Executable)weaver.getProxy();
proxyObject.execute();
System.out.println(proxyObject.getClass());
```

从输出结果我们也可以看出来，最终的代理对象是基于CGLIB的：

```
Executable without any Interfaces
2143 [main] INFO ...PerformanceMethodInterceptor  - 0:00:00.000
class ...Executable$$EnhancerByCGLIB$$9e62fc83
```

但是，即使目标对象类实现了至少一个接口，我们也可以通过proxyTargetClass属性强制ProxyFactory采用基于类的代理。以MockTask为例，它实现了ITask接口，默认情况下ProxyFactory对其会采用基于接口的代理，但是，通过proxyTargetClass，我们可以改变这种默认行为（见如下代码）：

```
ProxyFactory weaver = new ProxyFactory(new MockTask());
```

```
weaver.setProxyTargetClass(true);
NameMatchMethodPointcutAdvisor advisor = new NameMatchMethodPointcutAdvisor();
advisor.setMappedName("execute");
advisor.setAdvice(new PerformanceMethodInterceptor());
weaver.addAdvisor(advisor);
MockTask proxyObject =(MockTask)weaver.getProxy();
proxyObject.execute(null);
System.out.println(proxyObject.getClass());
```

现在，我们可以直接将代理对象强制转型为MockTask类型，并且，从输出结果也可以看到，最终的代理对象是基于CGLIB的，而不是动态代理的：

```
task executed.
311 [main] INFO ...PerformanceMethodInterceptor  - 0:00:00.000
class ...MockTask$$EnhancerByCGLIB$$4bf6056
```

除此之外，如果将ProxyFactory的optimize属性设定为true的话，ProxyFactory也会采用基于类的代理机制。关于optimize属性的更多信息，我们将在后面给出。

总地来说，如果满足以下列出的三种情况中的任何一种，ProxyFactory将对目标类进行基于类的代理。

- 如果目标类没有实现任何接口，不管proxyTargetClass的值是什么，ProxyFactory会采用基于类的代理。
- 如果ProxyFactory的proxyTargetClass属性值被设置为true，ProxyFactory会采用基于类的代理。
- 如果ProxyFactory的optimize属性值被设置为true，ProxyFactory会采用基于类的代理。

3. Introduction的织入

之所以将Introduction的织入单独列出，是因为Introduction型Advice比较特殊，如下所述。

- Introduction可以为已经存在的对象类型添加新的行为，只能应用于对象级别的拦截，而不是通常Advice的方法级别的拦截，所以，进行Introduction的织入过程中，不需要指定Pointcut，而只需要指定目标接口类型。
- Spring的Introduction支持只能通过接口定义为当前对象添加新的行为，所以，我们需要在织入的时机，指定新织入的接口类型。

鉴于以上两点，使用ProxyFactory进行Introduction的织入代码示例如代码清单9-24所示。

代码清单9-24 使用ProxyFactory进行Introduction的织入过程示例

```
ProxyFactory weaver = new ProxyFactory(new Developer());
weaver.setInterfaces(new Class[]{IDeveloper.class,ITester.class});
TesterFeatureIntroductionInterceptor advice = new TesterFeatureIntroductionInterceptor();
weaver.addAdvice(advice);
//DefaultIntroductionAdvisor advisor = new DefaultIntroductionAdvisor(advice,advice);
//weaver.addAdvisor(advisor);

Object proxy = weaver.getProxy();
((ITester)proxy).testSoftware();
((IDeveloper)proxy).developSoftware();
```

如果我们不使用Advisor而直接为ProxyFactory指定Advice的话，还记得ProxyFactory会如何处理的嘛？ ProxyFactory会在自身内部构建相应的Advisor来使用，对吧？因为TesterFeature-IntroductionInterceptor是IntroductionInfo的子类，所以，ProxyFactory内部会创建一个默认的DefaultIntroductionAdvisor实例，就跟我们注释掉的两行代码效果一样。

对Introduction进行织入，与基于接口的代理形式有点像，但有少许差异。对Introduction进行织入，新添加的接口类型必须是通过setInterfaces指定的，而原来的目标对象，是采用基于接口的代理形式还是采用基于类的代理形式，完全是可以自由选择的。上面我们通过setInterfaces同时指定了目标对象实现的接口和新添加的接口类型，在进行Introduction织入的同时使用了基于接口的代理形式。我们同样可以在织入Introduction的同时，使用基于类的代理形式（见代码清单9-25）。

代码清单9-25 使用ProxyFactory进行基于类的代理方式的Introduction织入过程示例

```
ProxyFactory weaver = new ProxyFactory(new Developer());
weaver.setProxyTargetClass(true);
weaver.setInterfaces(new Class[]{ITester.class});
TesterFeatureIntroductionInterceptor advice = new TesterFeatureIntroductionInterceptor();
weaver.addAdvice(advice);
//DefaultIntroductionAdvisor advisor = new DefaultIntroductionAdvisor(advice,advice);
//weaver.addAdvisor(advisor);

Object proxy = weaver.getProxy();
((ITester)proxy).testSoftware();
((Developer)proxy).developSoftware();
```

我们通过weaver.setProxyTargetClass(true);强制使用了基于类的代理，所以，现在得将代理对象转型为Developer而不是IDeveloper。

从介绍Advice类型到介绍Advisor类型，针对Introduction的部分都是单独陈述的，或许你已经猜到，Introduction的Advice以及Advisor是不能跟其他Advice和Advisor混用的，要织入Introduction，你只能使用IntroductionAdvisor或者其子类，而不能使用其他的组合。

9.5.2 看清 ProxyFactory 的本质

知其表而不知其里，充其量你只能算一个画匠，而不是画师； 只懂得如何使用API，而不知道这些API为何如此设计，使你迈不出从“画匠”到“画师”的那一步，如果你想迈出这一步，那不妨随我看一下这ProxyFactory内部到底有何“猫腻儿”，何如？

认识ProxyFactory的本质，不仅可以让我们清楚它如何实现，帮助我们在以后的系统设计中吸取宝贵的经验，而且可以进一步帮助我们更好地使用ProxyFactory。

注意 因为本节剩下的内容涉及的都是ProxyFactory或者Spring AOP框架的实现，所以，大部分类全部来自org.springframework.aop.framework包。

要了解ProxyFactory，我们得先从它的“根”说起，即org.springframework.aop.framework.AopProxy，该接口定义如下：

```
public interface AopProxy {
   Object getProxy();
   Object getProxy(ClassLoader classLoader);
}
```

Spring AOP框架内使用AopProxy对使用的不同的代理实现机制进行了适度的抽象，针对不同的代理实现机制提供相应的AopProxy子类实现。目前，Spring AOP框架内提供了针对JDK的动态代理和CGLIB两种机制的AopProxy实现（见图9-10）。

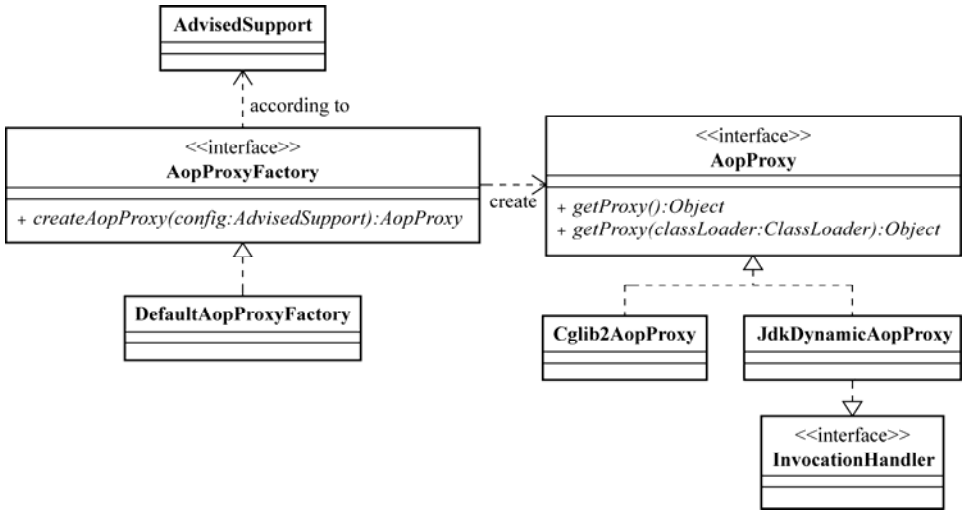

图9-10 AopProxy相关结构图

当前，AopProxy有Cglib2AopProxy和JdkDynamicAopProxy两种实现。因为动态代理需要通过InvocationHandler提供调用拦截，所以，JdkDynamicAopProxy同时实现了InvocationHandler接口。不同AopProxy实现的实例化过程采用工厂模式（确切地说是抽象工厂模式）进行封装，即通过org.springframework.aop.framework.AopProxyFactory进行。AopProxyFactory接口的定义如下所示：

```
public interface AopProxyFactory {
   AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException;
}
```

AopProxyFactory根据传入的AdvisedSupport实例提供的相关信息，来决定生成什么类型的AopProxy。不过，具体工作会转交给AopProxyFactory的具体实现类。而实际上这个AopProxyFactory实现类现在就一个，即org.springframework.aop.framework.DefaultAopProxyFactory。

DefaultAopProxyFactory的实现逻辑很简单，如以下伪代码所示：

```
if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces (config))
{
   // 创建Cglib2AopProxy实例，并返回;
}
else
{
   // 创建JdkDynamicAopProxy实例，并返回;
}
```

也就是说，如果传入的AdvisedSupport实例config的isOptimize或者isProxyTargetClass方法返回true，或者目标对象没有实现任何接口，则采用CGLIB生成代理对象，否则使用动态代理。还记得ProxyFactory会采用基于类的代理形式生成代理对象需要满足的条件吗？这里是一个关键点，但是走到这里，你还是无法理解为什么ProxyFactory会有这种好像偶然的行为。别急，我们接着看！

AopProxyFactory需要根据createAopProxy方法传入的AdvisedSupport实例信息，来构建相应的AopProxy。下面我们要看看这个AdvisedSupport到底是何方神圣。

说得简单一点儿，AdvisedSupport其实就是一个生成代理对象所需要的信息的载体，该类相关的类层次图，见图9-11。

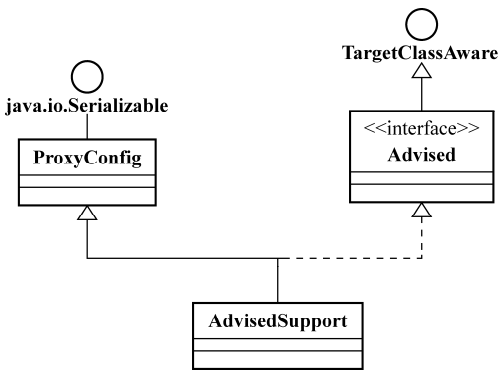

图9-11 AdvisedSupport类层次图

AdvisedSupport所承载的信息可以划分为两类，一类以org.springframework.aop.framework.ProxyConfig为统领，记载生成代理对象的控制信息；一类以org.springframework.aop.framework.Advised为旗帜，承载生成代理对象所需要的必要信息，如相关目标类、Advice、Advisor等。

ProxyConfig其实就是一个普通的JavaBean，它定义了5个boolean型的属性，分别控制在生成代理对象的时候，应该采取哪些行为措施，下面是这5个属性的详细情况。

- proxyTargetClass。在9.5.1节的第2小节中已经提到过这个属性，如果proxyTargetClass属性设置为true，则ProxyFactory将会使用CGLIB对目标对象进行代理，默认值为false。
- optimize。该属性的主要用于告知代理对象是否需要采取进一步的优化措施，如代理对象生成之后，即使为其添加或者移除了相应的Advice，代理对象也可以忽略这种变动。另外，我们也曾经提到，当该属性为true时，ProxyFactory会使用CGLIB进行代理对象的生成。默认情况下，该属性为false。更多信息参照Spring的Javadoc以及参考文档。
- opaque。该属性用于控制生成的代理对象是否可以强制转型为Advised，默认值为false，表示任何生成的代理对象都可强制转型为Advised，我们可以通过Advised查询代理对象的一些状态。
- exposeProxy。设置exposeProxy，可以让Spring AOP框架在生成代理对象时，将当前代理对象绑定到ThreadLocal。如果目标对象需要访问当前代理对象，可以通过AopContext.currentProxy()取得。该属性的用途将在后文中详细讲述。出于性能方面考虑，该属性默认值为false。
- frozen。如果将frozen设置为true，那么一旦针对代理对象生成的各项信息配置完成，则不容许更改。比如，如果ProxyFactory的设置完毕，并且fronzen为true，则不能对Advice进行任何变动，这样可以优化代理对象生成的性能。默认情况下，该值为false。

要生成代理对象，只有ProxyConfig提供的控制信息还不够，我们还需要生成代理对象的一些具体信息，比如，要针对哪些目标类生成代理对象，要为代理对象加入何种横切逻辑等，这些信息可以通过org.springframework.aop.framework.Advised设置或者查询。默认情况下，Spring AOP框架返回的代理对象都可以强制转型为Advised，以查询代理对象的相关信息。Advised的接口定义代码

太长，我们就不在此罗列了，你可以参照它的Javadoc。简单点儿说，我们可以使用Advised接口访问相应代理对象所持有的Advisor，进行添加Advisor、移除Advisor等相关动作。即使代理对象已经生成完毕，也可对其进行这些操作。直接操作Advised，更多时候用于测试场景，可以帮助我们检查生成的代理对象是否如所期望的那样。（有关Advised的更多信息，请参照Spring的参考文档，因为与我们的主题相关性不大，这里不进行详细讲述。）

回到之前的AdvisedSupport话题，AdvisedSupport继承了ProxyConfig，我们可以通过AdvisedSupport设置代理对象生成的一些控制属性。AdvisedSupport同时实现了Advised接口，我们也可以通过AdvisedSupport设置生成代理对象相关的目标类、Advice等必要信息。这样，具体的AopProxy实现在生成代理对象时，可以从AdvisedSupport这里取得所有这些必要信息。

现在回到主题ProxyFactory。AopProxy、AdvisedSupport与ProxyFactory是什么关系呢？先看图9-12。

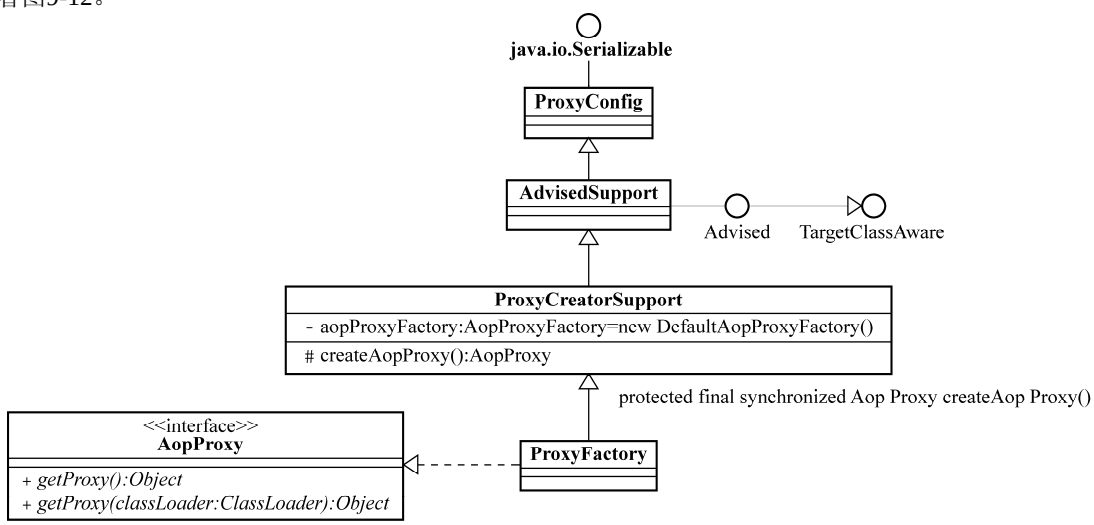

图9-12 ProxyFactory继承层次类图

ProxyFactory集AopProxy和AdvisedSupport于一身，所以，可以通过ProxyFactory设置生成代理对象所需要的相关信息，也可以通过ProxyFactory取得最终生成的代理对象。前者是AdvisedSupport的职责，后者是AopProxy的职责。

为了重用相关逻辑，Spring AOP框架在实现的时候，将一些公用的逻辑抽取到了org.springframework.aop.framework.ProxyCreatorSupport中，它自身就继承了AdvisedSupport，所以，生成代理对象的必要信息从其自身就可以搞到。为了简化子类生成不同类型AopProxy的工作，ProxyCreatorSupport内部持有一个AopProxyFactory实例，默认采用的是DefaultAopProxyFactory（也可以通过构造方法或者setter方法设置其他实现，如果有的话）。DefaultAopProxyFactory的默认行为前面已经讲述过了。ProxyFactory作为一个ProxyCreatorSupport自然继承了这种行为，从它的使用中我们已经领略过了。

前面已经说过了，ProxyFactory只是Spring AOP中最基本的织入器实现。实际上，ProxyFactory还有几个“兄弟”，这从ProxyCreatorSupport的继承类图（图9-13）中可以看到。

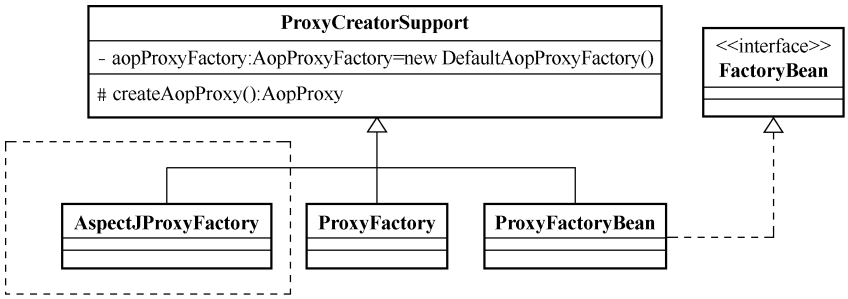

图9-13 ProxyFactory的“兄弟”

后文中将详细讲述AspectJProxyFactory。当前，我们还是先来看看ProxyFactoryBean。

9.5.3 容器中的织入器——ProxyFactoryBean

虽然使用ProxyFactory，可以让我们能够独立于Spring的IoC容器之外来使用Spring的AOP支持，但是，将Spring AOP与Spring的IoC容器支持相结合，才是发挥Spring AOP更大作用的最佳途径。通过结合Spring的IoC容器，我们可以在容器中对Pointcut和Advice等进行管理，即使它们依赖于其他业务对象，也可以很容易地注入其中。

在IoC容器中，使用org.springframework.aop.framework.ProxyFactoryBean作为织入器，它的使用与ProxyFactory无太大差别。不过在演示ProxyFactoryBean的使用之前，我们有必要在看清了ProxyFactory本质的前提下，进一步弄明白ProxyFactoryBean的本质。

1. ProxyFactoryBean的本质

对于ProxyFactoryBean，我们应该这样断词，即Proxy+FactoryBean，而不是ProxyFactory+ Bean。也就是说，ProxyFactoryBean本质上是一个用来生产Proxy的FactoryBean。还记得IoC容器中的FactoryBean的作用吧？如果容器中的某个对象持有某个FactoryBean的引用，它取得的不是FactoryBean本身，而是FactoryBean的getObject()方法所返回的对象。所以，如果容器中某个对象依赖于ProxyFactoryBean，那么它将会使用到ProxyFactoryBean的getObject()方法所返回的代理对象，这就是ProxyFactryBean得以在容器中游刃有余的原因。

要让ProxyFactoryBean的getObject()方法返回相应目标对象类的代理对象其实很简单。因为ProxyFactoryBean继承了与ProxyFactory共有的父类ProxyCreatorSupport，而ProxyCreatorSupport基本上已经把要做的事情（如设置目标对象、配置其他部件、生成对应的AopProxy等）全部完成了。我们只需在ProxyFactoryBean的getObject()方法中通过父类的createAopProxy()取得相应的AopProxy，然后“return AopProxy.getProxy()”即可。

因为涉及FactoryBean，所以在实现getObject()时，逻辑上还得点缀一下。我们来看ProxyFactoryBean的getObject()定义（见代码清单9-26）。

代码清单9-26 ProxyFactoryBean的getObject()方法逻辑

```
public Object getObject() throws BeansException {
   initializeAdvisorChain();
   if (isSingleton()) {
      return getSingletonInstance();
   }
   else
```

```
    {
    if (this.targetName == null)
    {
      logger.warn("Using non-singleton proxies with singleton targets is often undesirable." +➥
      "Enable prototype proxies by setting the 'targetName' property.");
    }
    return newPrototypeInstance();
    }
}
```

FactoryBean定义中要求标明返回的对象是以singleton的scope返回，还是以prototype的scope返回。所以，得针对这两种情况分别返回不同的代理对象，以满足FactoryBean的isSingleton()方法的语义。

如果将ProxyFactoryBean的singleton属性设置为true，则ProxyFactoryBean在第一次生成代理对象之后，会通过内部实例变量singletonInstance（Object类型）缓存生成的代理对象。之后，所有的请求将会返回这一缓存实例，从而满足singleton的语义。反之，如果将ProxyFactoryBean的singleton属性设置为false，那么，ProxyFactoryBean每次都会重新检测各项设置，并为当前调用准备一套新的环境，然后再根据最新的环境数据，返回一个新的代理对象。因此，如果singleton属性为false，在生成代理对象的性能上存在损失。如果非要这么做，请确保有充足的理由。singleton默认值为true，即返回同一个代理对象实例。

如果对ProxyFactoryBean的细节感兴趣，可以读一下ProxyFactoryBean的代码。

2. ProxyFactoryBean的使用

与ProxyFactory一样，通过ProxyFactoryBean，我们可以在生成目标对象的代理对象的时候，指定使用基于接口的代理还是基于类的代理方式，而且，因为它们全部继承自同一个父类，大部分可设置项目都相同。不过，ProxyFactoryBean在继承了父类ProxyCreatorSupport的所有配置属性之外，还添加了几个自己独有的，如下所示。

- proxyInterfaces。如果我们要采用基于接口的代理方式，那么需要通过该属性配置相应的接口类型，这是一个Collection类型实例，所以我们可以通过配置元素<list>来指定一个或者多个接口类型。实际上，这与通过Interfaces属性指定接口类型是等效的，我们完全可以随个人喜好来使用，虽然使用proxyInterfaces可以保持使用上的统一风格。另外，如果目标对象实现了某个或者多个接口，即使我们不通过该属性指定要代理的接口类型，ProxyFactroyBean也可以自动检测到目标对象所实现的接口，并对其进行基于接口的代理。因为ProxyFactoryBean有一个autodetectInterfaces属性，该属性默认值为true，即如果没有明确指定要代理的接口类型，ProxyFactoryBean会自动检测目标对象所实现的接口类型并进行代理。
- interceptorNames。通过该属性，我们可以指定多个将要织入到目标对象的Advice、拦截器以及Advisor，而再也不用通过ProxyFactory那样的addAdvice或者addAdvisor方法一个一个地添加了。因为该属性属于Collection类型，所以通常我们会使用配置元素<list>添加需要的拦截器名称。该属性有两个特性需要提及，如以下所述。
  - 如果没有通过相应的设置目标对象的方法明确为ProxyFactoryBean设置目标对象，那么可以在interceptorNames的最后一个元素位置，放置目标对象的bean定义名称。这是个特例，大部分情况下，还是建议明确指定目标对象，而避免这种配置方式。
  - 通过在指定的interceptorNames某个元素名称之后添加*通配符，可以让ProxyFactory-Bean在容器中搜寻符合条件的所有的Advisor并应用到目标对象。这些符合条件的Advisor，

Spring参考文档中称之为global advisor。代码清单9-27给出了这种用法的示例。

- singleton。因为ProxyFactoryBean本质上是一个FactoryBean，所以我们可以通过singleton属性，指定每次getObject调用是返回同一个代理对象，还是返回一个新的。通常情况下是返回同一个代理对象，即singleton为true。只有在需要返回有状态的代理对象的情况下，才会将singleton设置为false，如使用Introduction的场合。

代码清单9-27 包含*通配符的interceptorNames属性使用示例

```
<bean id="proxy"
   class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="target" ref="..."/>
   <property name="interceptorNames">
      <list>
         <value>global*</value>
      </list>
   </property>
</bean>
<bean id="global_debug" ➥
class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor. ➥
PerformanceMonitorInterceptor"/>
```

要在容器中通过ProxyFactoryBean使用基于接口的代理方式，通常可以采用代码清单9-28所示的配置方式。

代码清单9-28 通过ProxyFactoryBean使用基于接口的代理方式的配置示例

```
<bean id="pointcut" class="org.springframework.aop.support.NameMatchMethodPointcut">
   <property name="mappedName" value="execute"/>
</bean>

<bean id="performanceInterceptor" class="...advice.PerformanceMethodInterceptor">
</bean>

<bean id="performanceAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
   <property name="pointcut">
      <ref bean="pointcut"/>
   </property>
   <property name="advice">
      <ref bean="performanceInterceptor"/>
   </property>
</bean>

<bean id="task" class="...MockTask">
</bean>

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="target">
      <ref bean="task"/>
   </property>
   <property name="proxyInterfaces">
      <list>
         <value>...ITask</value>
      </list>
   </property>
   <property name="interceptorNames">
```

```
      <list>
        <value>performanceAdvisor</value>
      </list>
    </property>
  </bean>
```

现在，从Pointcut到Advice再到Advisor，从目标对象到相应的代理对象，全部都由IoC容器统一管理。为ProxyFactoryBean指定目标对象、要代理的接口类型以及相应的Advisor或Advice，ProxyFactoryBean就会返回目标对象的代理对象供调用者使用。我们可以将生成的代理对象直接注入到依赖的主体对象中，但是这里有一个初学者容易犯的错误，就是通常会将目标对象task注入依赖的主体对象，而不是目标对象的代理对象taskProxy。通过之前有关代理模式的讲解，现在应该不会犯这种错误了。将没有织入任何横切逻辑的目标对象，而不是代理对象注入依赖的主体对象，一定不会产生任何拦截效果。为了避免这种问题，如果没有依赖于目标对象的依赖关系，可以将目标对象的bean定义声明为内部bean，这样，就不会出现该引用目标对象代理对象的地方，反而因不慎或者其他原因而引用目标对象本身的情况。代码清单9-29演示了这种好的实践方式。

代码清单9-29 使用内部bean定义避免错误的依赖注入引用

```
...

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <bean class="...MockTask"/>
  </property>
  <property name="proxyInterfaces">
    <list>
      <value>...ITask</value>
    </list>
  </property>
  <property name="interceptorNames">
    <list>
      <value>performanceAdvisor</value>
    </list>
  </property>
</bean>
```

因为autodetectInterfaces的默认值为true，如果确认目标对象所实现的接口就是要代理的接口，那么，完全可以省略通过interfaces或者proxyInterfaces明确指定代理接口的配置。代码清单9-29的配置内容可以精简如下：

```
...

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <bean class="...MockTask"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>performanceAdvisor</value>
    </list>
  </property>
</bean>
```

如果没有指定要代理的接口类型，并且目标对象也没有实现任何接口，那么，ProxyFactoryBean会采用基于类的代理方式为目标对象生成代理对象。不过，即使目标对象实现了某些接口，我们

也可以强制ProxyFactoryBean采用基于类的代理方式来生成代理对象。与ProxyFactory一样，只要指定proxyTargetClass为true就可以了（见代码清单9-30）。

代码清单9-30 强制ProxyFactoryBean使用基于类的代理方式的配置示例

```
...
<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="target">
      <bean class="...MockTask"/>
   </property>
   <property name="proxyTargetClass">
      <value>true</value>
   </property>
   <property name="interceptorNames">
      <list>
         <value>performanceAdvisor</value>
      </list>
   </property>
</bean>
```

不过，现在客户端代码不能将代理对象强制转型为ITask，而应该强制转型为目标对象的具体类型，即MockTask，如下所示：

```
ApplicationContext ctx = ...;
// ITask task = (ITask)ctx.getBean("taskProxy");
// 错误!

MockTask task = (MockTask)ctx.getBean("taskProxy");
task.execute(null);
...
```

提示 有时，我们的应用可能需要依赖于第三方库，这些库中可能有些对象是出于简单实用的目的，就是没有进行面向接口编程，自然就没有实现任何接口。而且，我们自己设计和实现的类，可能出于某种目的，也是没有实现接口的必要，这时，就需要通过将proxyTargetClass设置为true来解决代理的问题。

说完了如何通过ProxyFactoryBean生成目标对象的代理对象（使用“基于接口的代理”方式也好，使用“基于类的代理”方式也好）。下面该说一下Introduction的代理了，因为它一直比较特立独行嘛！

为了演示Introduction的织入，我们引入一个ICounter接口定义以及一个简单实现类，然后将这个接口的行为和状态添加到ITask相应实现类中。ICounter接口以及相关实现类定义见代码清单9-31。

代码清单9-31 ICounter接口以及相关实现类定义

```
public interface ICounter {
   void resetCounter();
   int getCounter();
}

public class CounterImpl implements ICounter {
   private int counter;

   public int getCounter() {
      counter++;
```

```
        return counter;
    }

    public void resetCounter() {
        counter = 0;
    }
}
```

要将ICounter的行为添加到ITask相应实现类中，可以采用代码清单9-32所示的配置。

代码清单9-32 将ICounter行为添加到ITask的配置示例

```
<bean id="task" class="...MockTask" singleton="false">
</bean>

<bean id="introducedTask" class="org.springframework.aop.framework.ProxyFactoryBean" ➥
singleton="false">
    <property name="targetName">
        <value>task</value>
    </property>
    <property name="proxyInterfaces">
        <list>
            <value>...ITask</value>
            <value>...ICounter</value>
        </list>
    </property>
    <property name="interceptorNames">
        <list>
            <value>introductionInterceptor</value>
        </list>
    </property>
</bean>

<bean id="introductionInterceptor" ➥
class="org.springframework.aop.support.DelegatingIntroductionInterceptor"
singleton="false">
    <constructor-arg>
        <bean class="...CounterImpl">
        </bean>
    </constructor-arg>
</bean>
```

请注意，我们将目标对象的bean定义、ProxyFactoryBean的bean定义，以及相应Introduction-Interceptor的bean定义的scope，全部声明为prototype，也就是singleton="false"，并且，这种情况下，我们使用的是"taskName"而不是"task"来指定目标对象（使用task通过ref指定prototype类型的依赖会有什么效果，在Spring的IoC容器部分已经讲述过了。）。这样才能保证每次取得的代理对象都持有各自独有的状态和行为，如下是调用执行的代码示例：

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("...");
Object proxy1 = ctx.getBean("introducedTask");
Object proxy2 = ctx.getBean("introducedTask");

System.out.println(((ICounter)proxy1).getCounter());
System.out.println(((ICounter)proxy1).getCounter());
System.out.println(((ICounter)proxy2).getCounter());
```

因为proxy1和proxy2各自拥有独立的状态，所以，输出为：

```
1
```

```
2
1
```

我们之前说过，DelegatingIntroductionInterceptor是一个“伪军”，如果不是采用prototype的scope为每一个代理对象都分配一个该类型实例，则无法保证各代理对象拥有各自的状态。不过，如果使用DelegatePerTargetObjectIntroductionInterceptor，那么可以共用一个该类型的Advice实例（即使用sigleton的scope），见代码清单9-33。

代码清单9-33 使用DelegatePerTargetObjectIntroductionInterceptor代替DelegatingIntroductionInterceptor后的配置实例

```
<bean id="task" class="...MockTask" singleton="false">
</bean>

<bean id="introducedTask" class="org.springframework.aop.framework.ProxyFactoryBean" ➥
singleton="false">
   <property name="target"><ref bean="task"/></property>
   <property name="proxyInterfaces">
      <list>
         <value>...ITask</value>
         <value>...ICounter</value>
      </list>
   </property>
   <property name="interceptorNames">
      <list>
         <value>introductionInterceptor</value>
      </list>
   </property>
</bean>

<bean id="introductionInterceptor" class="org.springframework.aop.support. ➥
DelegatePerTargetObjectIntroductionInterceptor">
   <constructor-arg index="0">
      <value>...CounterImpl</value>
   </constructor-arg>
   <constructor-arg index="1">
      <value>...ICounter</value>
   </constructor-arg>
</bean>
```

至于你的自定义IntroductionInterceptor，在应用的时候，请根据情况设置IntroductionInterceptor的scope以保证状态的独立性。有关ProxyFactoryBean的更多配置项细节，请参照对应的Javadoc，这里就不赘述了。我们得加快织入的速度了，毕竟，一个一个地配置ProxyFactoryBean可不是什么令人感到轻松、愉快的事情。

9.5.4 加快织入的自动化进程

在IoC容器中使用ProxyFactoryBean进行横切逻辑的织入固然不错，但是，我们都是针对每个目标对象，然后给出它们各自所对应的ProxyFactoryBean配置。如果目标对象就那么几个，那还应付得过来。但系统中那么多的业务对象可能都是目标对象，如果还是用ProxyFactoryBean一个个地进行配置，估计得累得吐血，所以，我们得寻求更加简单快捷的方式。

注意 当然，如果系统小，而且横切关注点不多，目标对象少，那么简单地使用ProxyFacto-

ryBean也不失为合适的方式。

Spring AOP给出了自动代理（AutoProxy）机制，用以帮助我们解决使用ProxyFactoryBean配置工作量比较大的问题。

1. 自动代理得以实现的原理

要使用自动代理机制，需要以Spring的IoC容器为依托。更进一步说，需要使用ApplicationContext类型的IoC容器。虽然可以通过进一步的编码，让BeanFactory也支持这一功能，但是既然要自动，还是一步到位的好。

Spring AOP的自动代理的实现建立在IoC容器的BeanPostProcessor概念之上。还记得我们可以使用BeanPostProcessor干预bean的实例化过程吗？通过BeanPostProcessor，我们可以在遍历容器中所有bean的基础上，对遍历到的bean进行一些操作。有了这个前提条件，要实现自动代理就很容易了。

其实我们不难想到，只要提供一个BeanPostProcessor，然后在这个BeanPostProcessor内部实现这样的逻辑，即当对象实例化的时候，为其生成代理对象并返回，而不是实例化后的目标对象本身，从而达到代理对象自动生成的目的。该逻辑如果以伪代码演示，如代码清单9-34所示。

代码清单9-34 自动代理实现原理的伪代码示例

```
for(bean in IoC container)
{
   检查当前bean定义是否符合拦截条件;
   如果符合拦截条件，则
   {
      Object proxy = createProxyFor(bean);
      return proxy;
   }
   否则
   {
      Object instance = createInstance(bean);
      return instance;
   }
}
```

而对于Object proxy = createProxyFor(bean);这行代码，如何根据目标对象生成相应的代理对象的细节，我想就不用说了吧（不就是使用ProxyFactory或者ProxyFactoryBean嘛）！我要说的是第一行，即检查当前bean定义是否符合拦截条件。

要检查当前bean是否符合拦截条件，首先需要知道拦截条件是什么，那么我们就要通过某种方式，告知具体的自动代理实现类都有哪些拦截条件：

- 可以通过外部配置文件传入这些拦截条件信息，比如我们在容器的配置文件中注册的有关Pointcut以及Advisor等就包括这些信息；
- 还可以在具体类的定义文件中，通过元数据来指明具体的拦截条件是什么，比如可以通过Jakarta Commons Attributes或者Java 5的注解，直接在代码类中标注Pointcut等拦截信息。

但不管采用什么方式来提供拦截信息，我们都可以提供相应的自动代理实现类来读取这些信息，并为相应的目标对象自动生成代理对象。

下面，让我们先看一下Spring AOP都提供了哪些可以使用的自动代理实现类，免得做一些无用功。

2. 可用的AutoProxyCreator

Spring AOP在org.springframework.aop.framework.autoproxy包中提供了两个常用的AutoProxyCreator，即BeanNameAutoProxyCreator和DefaultAdvisorAutoProxyCreator。

- BeanNameAutoProxyCreator

使用BeanNameAutoProxyCreator，我们可以通过指定一组容器内的目标对象对应的beanName，将指定的一组拦截器应用到这些目标对象之上。代码清单9-35给出了通常情况下使用BeanNameAutoProxyCreator的配置演示。

代码清单9-35 使用BeanNameAutoProxyCreator进行自动代理的配置示例

```
<bean id="target1" class="..."/>
<bean id="target2" class="..."/>

<bean id="mockTask" class="..."/>
<bean id="fakeTask" class="..."/>

<bean id="taskThrowsAdvice" class="...TaskThrowsAdvice">
</bean>

<bean id="performanceInterceptor" class="...PerformanceMethodInterceptor">
</bean>

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
   <property name="beanNames">
      <list>
         <value>target1</value>
         <value>target2</value>
      </list>
   </property>
   <property name="interceptorNames">
      <list>
         <value>taskThrowsAdvice</value>
      </list>
   </property>
</bean>

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
   <property name="beanNames">
      <list>
         <value>mockTask</value>
         <value>fakeTask</value>
      </list>
   </property>
   <property name="interceptorNames">
      <list>
         <value>performanceInterceptor</value>
      </list>
   </property>
</bean>
```

通过beanNames，我们可以指定要对容器中的哪些bean自动生成代理对象。通过interceptorNames，我们可以指定将要应用到目标对象的拦截器、Advice或者Advisor等。实际上，我们可以将以上两个BeanNameAutoProxyCreator并作一个使用。给出至少两个BeanNameAutoProxyCreator定义，只是为了表明，我们可以使用多个BeanNameAutoProxyCreator以细化横切逻辑的织入范围。

如果一类目标对象的bean定义名称相似，如可能我们系统中服务层的bean定义名称都以service结尾，那么我们可以如代码清单9-36所示，在beanNames属性中指定*通配符以简化配置。

代码清单9-36 在beanNames属性中指定*通配符以简化配置

```
...
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
   <property name="beanNames">
      <list>
         <value>target*</value>
         <value>*service</value>
      </list>
   </property>
   <property name="interceptorNames">
      <list>
         <value>performanceInterceptor</value>
      </list>
   </property>
</bean>
...
```

如果在使用*通配符的情况下，我们还是要指定一长串的beanNames，那么使用一下配置诀窍吧。对于String[]型的数组，我们也可以不用<list>元素，而是如下所示，直接使用逗号分隔数组的各个元素即可：

```
...
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
   <property name="beanNames">
      <value>target*,*Task,*service</value>
   </property>
   <property name="interceptorNames">
      <list>
         <value>performanceInterceptor</value>
      </list>
   </property>
</bean>
```

当然，这是容器配置中的小诀窍，与BeanNameAutoProxyCreator可没太大的关系。因为我们可以在任何类似场合使用这种配置方式。

- DefaultAdvisorAutoProxyCreator

如果把BeanNameAutoProxyCreator比作半自动步枪的话，那么DefaultAdvisorAutoProxyCreator可算全自动步枪啦。使用DefaultAdvisorAutoProxyCreator，我们只需要在ApplicationContext的配置文件中注册一下DefaultAdvisorAutoProxyCreator的bean定义就可以了。剩下的事情，DefaultAdvisorAutoProxyCreator全部搞定。代码清单9-37给出了使用DefaultAdvisorAutoProxyCreator的配置演示。

代码清单9-37 DefaultAdvisorAutoProxyCreator的使用配置示例

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean id="target1" class="..."/>
<bean id="target2" class="..."/>

<bean id="mockTask" class="..."/>
<bean id="fakeTask" class="..."/>

<bean id="logAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
   <property name="pointcut">
      ...
   </property>
```

```
    <property name="advice">
      <bean id="performanceInterceptor" class="...PerformanceMethodInterceptor">
      </bean>
    </property>
</bean>

<bean id="logAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut">
      ...
    </property>
    <property name="advice">
      <bean id="taskThrowsAdvice" class="...TaskThrowsAdvice">
      </bean>
    </property>
</bean>
```

将DefaultAdvisorAutoProxyCreator注册到容器后，它就会自动搜寻容器内的所有Advisor，然后根据各个Advisor所提供的拦截信息，为符合条件的容器中的目标对象生成相应的代理对象。注意，DefaultAdvisorAutoProxyCreator只对Advisor有效，因为只有Advisor才既有Pointcut信息以捕捉符合条件的目标对象，又有相应的Advice。

使用DefaultAdvisorAutoProxyCreator对容器内所有bean定义对应的对象进行自动代理之后，我们从容器中取得的对象实例，就都是代理后已经包含了织入的横切逻辑的代理对象了，除非该对象不符合Pointcut规定的拦截条件。

因为DefaultAdvisorAutoProxyCreator的处理范围比较大，所以，为了避免将不必要的横切逻辑织入到不需要的目标对象之上，应该尽量细化各个Advisor的定义，或者，转而使用BeanNameAutoProxyCreator来进行更加细粒度的织入范围控制。

提示 某些时候，如果系统中有许多目标对象类无法采用“基于接口的代理”形式进行拦截，那么我们就需要明确告知AOP框架，在生成代理对象的时候应该采用“基于类的代理”形式。使用自动代理之前，我们可能需要依次为每一个ProxyFactoryBean都指定一下proxyTargetClass属性，但是使用了自动代理之后，因为所有AutoProxyCreator的父类都继承自ProxyConfig，所以只需要通过具体的AutoProxyCreator指定一下proxyTargetClass属性，然后就可以控制所有代理对象的生成是采用“基于类的代理”形式了。对于DefaultAdvisorAutoProxyCreator来说，是如下这个样子：

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
    <property name="proxyTargetClass">
        <value>true</value>
    </property>
</bean>
```

3. 扩展AutoProxyCreator

如果BeanNameAutoProxyCreator和DefaultAdvisorAutoProxyCreator不能满足我们的要求，或者我们想要一种其他方式的自动代理，比如基于元数据的方式，那么可以在Spring AOP提供的AbstractAutoProxyCreator或者AbstractAdvisorAutoProxyCreator基础之上，实现相应的子类，而不用什么都重新开始。

下面我们看一下Spring AOP框架中有关自动代理的实现架构（见图9-14）。

所有的AutoProxyCreator都是InstantiationAwareBeanPostProcessor，这种类型的

BeanPostProcessor与普通的BeanPostProcessor有所不同。当Spring IoC容器检测到有InstantiationAwareBeanPostProcessor类型的BeanPostProcessor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，也就是我说的“短路”。这样，相应的AutoProxyCreator会直接构造目标对象的代理对象返回，而不是原来的目标对象。

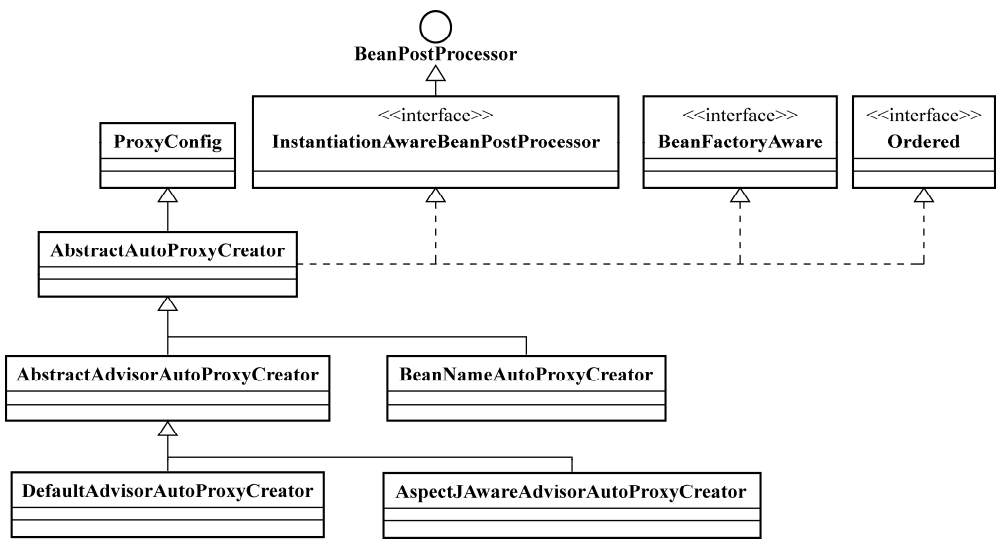

图9-14 AutoProxyCreator实现结构类图

要实现自定义的AutoProxyCreator，可以像BeanNameAutoProxyCreator那样直接继承AbstractAutoProxyCreator，也可以像DefaultAdvisorAutoProxyCreator那样继承AbstractAdvisorAutoProxyCreator。不管如何，子类中只需要提供规则匹配一类的逻辑，如果必要，也可以覆写相应逻辑。

AspectJAwareAdvisorAutoProxyCreator是Spring 2.0之后的AutoProxyCreaotr实现，也算是一个AutoProxyCreator的自定义实现。它还有一个子类AnnotationAwareAspectJAutoProxyCreator，可以支持根据Java 5的注解捕获信息以完成自动代理。我们将在10.1.1节给出更多相关信息。

注意 Spring AOP还支持基于Jakarta Commons Attributes的元数据的自动代理机制。但随着Java 5的普及，Java 5中的注解作为标准将迅速覆盖Java开发领域，所以，基于Jakarta Commons Attribute的自动代理我们就不做解释了，其原理，我们在上面的讲解中也已经提及了，无非是拦截信息的提供方式不同而已。但如果你想使用基于元数据的自动代理，而又不能升级到Java 5，那么可以参照Spring的参考文档，以取得有关在Spring Aop中使用Jakarta Commons Attributes进行自动代理的更多信息。

9.6 TargetSource

通常，在使用ProxyFactory的时候，我们都是通过setTarget()方法指定具体的目标对象。使用ProxyFactoryBean也是如此，或者ProxyFactoryBean还可以通过setTargetName()指定目标对象在IoC容器中的bean定义名称。但除此之外，还有一种方式没有说，那就是还可以通过setTargetSource()来指定目标对象。

TargetSource的作用就好像是为目标对象在外面加了一个壳，或者说，它就像是目标对象的容器。当每个针对目标对象的方法调用经历层层拦截而到达调用链的终点的时候，就该调用目标对象上定义的方法了。但这时，Spring AOP做了点儿手脚，它不是直接调用这个目标对象上的方法，而是通过“插足于”调用链与实际目标对象之间的某个TargetSource来取得具体目标对象，然后再调用从TargetSource中取得的目标对象上的相应方法。整个情形如图9-15所示。

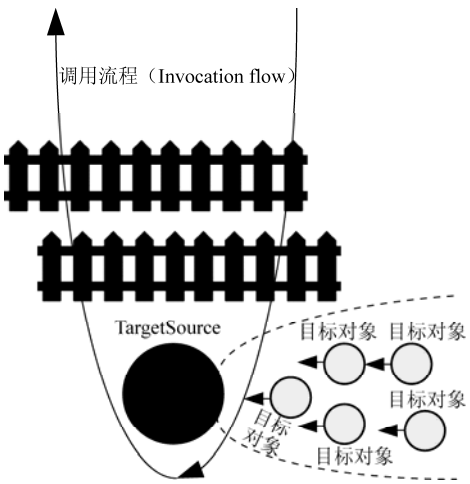

图9-15 TargetSource示意图

在通常情况下，无论是通过setTarget()，还是通过setTargetName()等方法设置的目标对象，框架内部都会通过一个TargetSource实现类对这个设置的目标对象进行封装，也就是说，框架内部会以统一的方式处理调用链终点的目标对象。

TargetSource最主要的特性就是，每次的方法调用都会触发TargetSource的getTarget()方法，getTarget()方法将从相应的TargetSource实现类中取得具体的目标对象，这样，我们就可以控制每次方法调用作用到的具体对象实例：

- 提供一个目标对象池，每次从TargetSource取得的目标对象都从这个目标对象池中取得。
- 让一个TargetSource实现类持有多个目标对象实例，然后按照某种规则，在每次方法调用时，返回相应的目标对象实例。

当然，还可以让TargetSource只持有一个目标对象实例，这样，每次的方法调用就都会针对这一个目标对象实例。其实，这就是通常ProxyFactory或者ProxyFactoryBean处理目标对象的方式，它们内部会构造一个org.springframework.aop.target.SingletonTargetSource实例，而SingletonTargetSource则会针对每次方法调用都返回同一个目标对象实例的引用。

9.6.1 可用的 TargetSource 实现类

在深入TargetSource的定义之前，我们还是先来看一下有哪些现成的TargetSource实现类。如果这些现有的TargetSource实现类不能满足需求，我们再寻求自定义的TargetSource实现。以下所有的TargetSource实现类全部位于org.springframework.aop.target包中。

1. SingletonTargetSource

org.springframework.aop.target.SingletonTargetSource是使用最多的TargetSource实现类，虽然我们可能并不知道。因为在通过ProxyFactoryBean的setTarget()设置完目标对象之后，ProxyFactoryBean内部会自行使用一个SingletonTargetSource对设置的目标对象进行封装。

SingletonTargetSource的实现其实很简单，就是内部只持有一个目标对象，当每次方法调用到达时，SingletonTargetSource都会返回这同一个目标对象。

所有的TargetSource都可以通过ProxyFactoryBean的setTargetSource()方法进行设置（ProxyFactory同样可以）。为了演示TargetSource的使用，我们来看一个简单的例子（见代码清单9-38）。

代码清单9-38 SingletonTargetSource使用示例

```
<bean id="target" class="..."/>

<bean id="singletonTargetSource" class="org.springframework.aop.target.SingletonTargetSource">
   <constructor-arg>
     <ref bean="target"/>
   </constructor-arg>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="targetSource">
     <ref bean="singletonTargetSource"/>
   </property>
   <property name="interceptorNames">
     <list>
       <value>anyInterceptor</value>
     </list>
   </property>
</bean>
```

该实例其实没有什么太大意义，因为设置一个SingletonTargetSource型的TargetSource，其实跟直接setTarget是等效的。

2. **PrototypeTargetSource**

与SingletonTargetSource正好相反，如果为ProxyFactory或者ProxyFactoryBean设置一个PrototypeTargetSource类型的TargetSource，那么每次方法调用到达调用链终点，并即将调用目标对象上的方法的时候，PrototypeTargetSource都会返回一个新的目标对象实例供调用。代码清单9-39给出了PrototypeTargetSource的简单使用演示。

代码清单9-39 PrototypeTargetSource使用示例

```
<bean id="target" class="..." singleton="false"/>

<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">
   <property name="targetBeanName">
     <value>target</value>
   </property>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="targetSource">
     <ref bean="prototypeTargetSource"/>
   </property>
   <property name="interceptorNames">
     <list>
       <value>anyInterceptor</value>
     </list>
   </property>
</bean>
```

注意，因为PrototypeTargetSource每次都需要返回新的对象实例，所以，需要注意以下两点。

(1) 目标对象的bean定义声明的scope必须为prototype。

(2) 通过targetBeanName属性指定目标对象的bean定义名称，而不是引用。原因我想就不用多说了，在第4章讲解Spring的IoC容器的时候，我们已经领教过了。

如果对以上配置所取得的targetProxy进行单元测试，那么代码清单9-40所示的测试代码运行之后，应该是可以得到可爱的Green Bar的。（不要告诉我你不知道Green Bar在TDD中意味着什么哦！）

代码清单9-40 针对PrototypeTargetSource的单元测试代码示例

```
...

public void testPrototypeTargetSource() throws Exception
{
   Object proxy = ctx.getBean("targetProxy");
   Object targetObject0 = ((Advised)proxy).getTargetSource().getTarget();
   Object targetObject1 = ((Advised)proxy).getTargetSource().getTarget();
   Object targetObject2 = ((Advised)proxy).getTargetSource().getTarget();
   assertNotSame(targetObject0, targetObject1);
   assertNotSame(targetObject1, targetObject2);
   assertNotSame(targetObject0, targetObject2);
}
```

3. HotSwappableTargetSource

这是我觉得比较有用的一个TargetSource实现。使用HotSwappableTargetSource封装目标对象，可以让我们在应用程序运行的时候，根据某种特定条件，动态地替换目标对象类的具体实现，比如，IService有多个实现类，如果程序启动之后，默认的IService实现类出现了问题，我们可以马上切换到IService的另一个实现上，而所有这些对于调用者来说都是透明的。

使用HotSwappableTargetSource的swap方法，可以用新的目标对象实例将旧的目标对象实例替换掉。该方法的声明如下所示：

```
public Object swap(Object newTarget)
```

该方法会返回被替换的旧的目标对象实例。

要使用HotSwappableTargetSource，我们得在它构造的时候，就提供一个默认的目标对象实例，如代码清单9-41所示。

代码清单9-41 HotSwappableTargetSource的初始化

```
<bean id="task" class="org.darrenstudio.books.unveilspring.aop.advisor.MockTask">
</bean>

<bean id="hotSwapTargetSource" class="org.springframework.aop.target.HotSwappableTargetSource">
   <constructor-arg>
      <ref bean="task"/>
   </constructor-arg>
</bean>

<bean id="taskProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="targetSource" ref="hotSwapTargetSource">
   </property>
   <property name="interceptorNames">
      <list>
         <value>anyInterceptor</value>
      </list>
   </property>
```

```
</bean>
```

我们使用构造方法注入为HotSwappableTargetSource注入了初始的目标对象，之后，就可以在程序中对其进行操作，如代码清单9-42所示。

代码清单9-42 HotSwappableTargetSource使用示例

```
...

Object proxy = ctx.getBean("taskProxy");
Object initTarget = ((Advised)proxy).getTargetSource().getTarget();

HotSwappableTargetSource hotSwapTargetSource = ➥
(HotSwappableTargetSource)ctx.getBean("hotSwapTargetSource");
Object oldTarget = hotSwapTargetSource.swap(new ITask(){
   public void execute(TaskExecutionContext ctx) {
      // 省略
   }});
Object newTarget = ((Advised)proxy).getTargetSource().getTarget();

assertSame(initTarget, oldTarget);
assertNotSame(initTarget,newTarget);
```

能最快记起来的使用HotSwappableTargetSource的场景，就是我曾经在2005年使用它解决了双数据源的互换问题：在有两个数据库双机热备的情况下，如果一个数据库挂掉，则将程序迅速地切换到另一个数据库。当时，使用了ThrowsAdvice对数据库相关异常进行捕捉，在捕捉到必要的切换信息后，就调用HotSwappableTargetSource的swap方法使用新的数据源替换旧的数据源，具体详情可以参照我的博客文章《对双数据源互换的实现的改进》（http://darrenwang. blogcn.com/diary,101689446.shtml）。

总之，使用HotSwappableTargetSource，我们可以在任何合适的地方、合适的时机对旧的目标对象进行替换，比如可以在某个拦截器中持有相关HotSwappableTargetSource的引用，一旦满足相应的条件，就可以调用swap方法动态替换新的目标对象，也可以设置一个定时任务，让它也持有一个HotSwappableTargetSource的引用，每隔一段时间就使用新的目标对象置换掉旧的，等等诸如此类的场景。有关HotSwappableTargetSource的使用，还是有待你自己去挖掘吧！

4. CommonsPoolTargetSource

某些时候，我们可能不想每次都返回新的目标对象，而是想返回有限数目的目标对象实例，这些目标对象实例的“地位”是平等的，就好像数据库连接池中的那些Connection一样，我们可以提供一个目标对象的对象池，然后让某个TargetSource实现每次都从这个目标对象池中去取得目标对象。

CommonsPoolTargetSource就是这么一个TargetSource实现，它使用现有的Jakarta Commons Pool提供对象池支持。使用它，跟使用PrototypeTargetSource没什么太大差别，如代码清单9-43所示。

代码清单9-43 CommonsPoolTargetSource使用配置示例

```
<bean id="target" class="..." singleton="false"/>

<bean id="poolingTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
   <property name="targetBeanName">
      <value>target</value>
   </property>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="targetSource">
      <ref bean="poolingTargetSource"/>
```

```
    </property>
    <property name="interceptorNames">
      <list>
        <value>anyInterceptor</value>
      </list>
    </property>
  </bean>
```

在CommonsPoolTargetSource的使用上，需要注意的问题跟PrototypeTargetSource差不多，即注意它要使用prototype型scope的bean定义。CommonsPoolTargetSource还有许多控制对象池的可配置属性，比如对象池的大小、初始对象数量等，都可以在配置中指定。详情可参照CommonsPoolTargetSource的Javadoc文档。

如果CommonsPoolTargetSource不能满足要求，或者因为其他原因不能使用Jakarta Commons Pool，那么也可以通过扩展org.springframework.aop.target.AbstractPoolingTargetSource类，实现相应的提供对象池化功能的TargetSource。有关扩展AbstractPoolingTargetSource的更多信息，可以参考它的Javadoc或者源码，毕竟Spring是开源的嘛！

5. **ThreadLocalTargetSource**

如果想为不同的线程调用提供不同的目标对象，那么可以使用org.springframework.aop.target.ThreadLocalTargetSource。它可以保证各自线程上对目标对象的调用，可以被分配到当前线程对应的那个目标对象实例上。其实，ThreadLocalTargetSource无非就是对JDK标准的ThreadLocal进行了简单的封装而已。

与其他TargetSource类似，ThreadLocalTargetSource的使用也比较简单，见代码清单9-44。

代码清单9-44 ThreadLocalTargetSource使用配置示例

```
<bean id="target" class="..." singleton="false"/>

<bean id="threadLocalTargetSource" ➥
class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName">
    <value>target</value>
  </property>
</bean>

<bean id="targetProxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource">
    <ref bean="threadLocalTargetSource"/>
  </property>
  <property name="interceptorNames">
    <list>
      <value>anyInterceptor</value>
    </list>
  </property>
</bean>
```

我想你不会忘记将目标对象的bean定义的scope设置成prototype型吧。毕竟人家每个线程起码得有自己的目标对象实例不是？不过没关系，要是非设置成singleton的，Spring AOP会“给你好看”的啦，呵呵，异常伺候！

9.6.2 自定义 TargetSource

说了这么多可以使用的TargetSource实现，大部分情况下应该够用了。不过，永远也不能排除

特殊情况，我们还得做好实现自定义TargetSource的准备。

要实现自定义的TargetSource，我们可以直接扩展TargetSource接口，好在这个接口定义的方法不多，如下所示：

```
public interface TargetSource extends TargetClassAware {
   Class getTargetClass();
   boolean isStatic();
   Object getTarget() throws Exception;
   void releaseTarget(Object target) throws Exception;
}
```

从下面的方法名称上，我估计各位就能猜出个大概了。

- getTargetClass()方法返回目标对象类型；
- isStatic()用于表明是否要返回同一个目标对象实例，SingletonTargetSource的这个方法肯定是返回true，其他的实现根据情况，通常返回false；
- getTarget()是核心，要返回哪个目标对象实例，完全由它说了算；
- 具体调用过程的结束，如果isStatic()为false，则会调用releaseTarget()以释放当前调用的目标对象。但是否需要释放，完全是由实现的需要决定的，大部分时候，该方法可以空着不实现。

为了演示TargetSource的特性以及如何实现一个TargetSource，我实现了一个简单的AlternativeTargetSource。它内部有一个计数器，当计数器为奇数的时候，TargetSource将针对当前调用返回第一个目标对象实例；否则，返回第二个目标对象实例。AlternativeTargetSource的定义如代码清单9-45所示。

代码清单9-45 AlternativeTargetSource定义

```
public class AlternativeTargetSource implements TargetSource {
   private ITask alternativeTaskOne;
   private ITask alternativeTaskTwo;

   private int counter;

   public AlternativeTargetSource(ITask task1,ITask task2)
   {
      this.alternativeTaskOne = task1;
      this.alternativeTaskTwo = task2;
   }

   public Object getTarget() throws Exception {
      try
      {
         if(counter %2 == 0)
            return alternativeTaskTwo;
         else
            return alternativeTaskOne;
      }
      finally
      {
         counter++;
      }
   }

   public Class getTargetClass() {
      return ITask.class;
```

```
    }

    public boolean isStatic() {
      return false;
    }

    public void releaseTarget(Object arg0) throws Exception {
      // 什么也不做
    }

}
```

在使用AlternativeTargetSource的时候（见代码清单9-46），就会发现，多次的方法调用所发生的目标对象实际上是交错变换的。

代码清单9-46 AlternativeTargetSource的使用示例

```
ITask task1 = new ITask()
{
  public void execute(TaskExecutionContext ctx) {
    System.out.println("execute in Task1.");
  }
};
ITask task2 = new ITask()
{
  public void execute(TaskExecutionContext ctx) {
    System.out.println("execute in Task2.");
  }
};

ProxyFactory pf = new ProxyFactory();
TargetSource targetSource = new AlternativeTargetSource(task1,task2);
pf.setTargetSource(targetSource);
Object proxy = pf.getProxy();
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
((ITask)proxy).execute(null);
...
```

程序输出，如下所示：

```
execute in Task2.
execute in Task1.
execute in Task2.
execute in Task1.
execute in Task2.
...
```

其实，这个TargetSource实现没有太大的意义，不过扩展一下就说不定了。最主要的，我们可以根据应用的具体场景来给出特定的TargetSource实现。

如果我们的TargetSource自定义实现不在乎是否依赖于Spring的IoC容器，也不妨考虑一下org.springframework.aop.target包中的几个抽象类，直接在这些抽象类的基础上进行扩展，可以省却部分劳烦。

9.7 小结

本章我们详尽剖析了Spring AOP中的各种概念和实现原理，这些概念和实现原理是Spring AOP发布之初就确定的，是整个框架的基础。纵使框架版本如何升级，甚至为Spring AOP加入更多的特性，在升级和加入更多更多特性的过程中，也将一直秉承Spring AOP的这些理念。

了解Spring AOP框架发布之初就确立的各种概念和原理，可以帮助我们更好地理解和使用Spring AOP。甚至，可以帮助我们去扩展Spring AOP。而接下来要讲述的，就是Spring 2.0之后对Spring AOP进行的扩展。