# Spring的设计理念和整体架构

# Spring Framework的核心：IoC容器的实现

# Spring AOP的实现

## Spring AOP概述

### AOP概念回顾

1、AOP是Aspect-Oriented Pogramming(面向切面编程)的简称

2、模块化

* 初学编程时喜欢把所有代码写进一个main函数里，这种编码方式造成了一个很不好的结果---程序维护性很差
* 于是，开始用子函数来对程序进行模块划分，并对一些基本的功能进行封装
* 后来，为了让代码维护更方便，又把不同的子函数的实现放到了不同的文件中。这样更方便了，不仅不用再一长串的代码文件里查找和维护，还可以让不同的开发人员并行开发和维护，大大提高开发效率
* 面向对象设计其实也是一种模块化的方法，它把相关的数据及其处理方法放在了一起。与单纯使用子函数进行封装相比，面向对象的模块化特性更完备，它体现了计算的一个基本原则---让计算尽可能靠近数据。一个类基本就是一个基本的模块。利用继承关系使得类得到重用，提高开发效率
* 但是程序中总会出现一些重复代码，而且不太方便使用继承的方法把他们重用和管理起来。它们功能重复并且需要用在不同的地方，虽然可以对这些代码左一些简单的封装，使之成为公共函数，但是在这种显式调用中，使用它们不是很方便
* 在使用这些公共函数的时候，往往也需要进行一些逻辑设计，也就是需要代码实现来支持，而且这些逻辑代码也是需要维护的
* 使用AOP后，可以将这些重复的代码抽取出来单独维护，在需要使用时统一调用，还可以为如何使用这些公共代码提供丰富灵活的手段

### Advice增强

1、Advice定义在连接点做什么，为切面增强提供织入接口。Advice是AOP联盟定义的一个接口如下。

1. **public** **interface** Advice {
2. }

2、在Spring AOP的实现中，使用了这个统一接口，并通过这个接口，为AOP切面增强的织入功能做了更多的细化和扩展，比如BeforeAdvice、AfterAdvice、ThrowsAdvice等

1. **public** **interface** BeforeAdvice **extends** Advice {
2. }
3. **public** **interface** AfterAdvice **extends** Advice {
4. }

3、在BeforeAdvice的继承关系中，定义了为待增强的目标方法设置的前置增强接口MethodBeforeAdvice，使用这个前置接口需要实现一个回调函数

1. **public** **interface** MethodBeforeAdvice **extends** BeforeAdvice {
2. **void** before(Method var1, Object[] var2, Object var3) **throws** Throwable;
3. }

* 作为回调函数，before方法的实现在Advice中被配置到目标方法后，会在调用目标方法时被回调
* Method对象：这个参数是目标方法的反射对象
* Object[]对象数组：这个对象数组中包含方法的输入参数

4、在Advice的实现体系中，Spring还提供了AfterAdvice这种增强类型，其常用的子接口有AfterReturningAdvice，ThrowsAdvice

1. **public** **interface** AfterReturningAdvice **extends** AfterAdvice {
2. **void** afterReturning(Object var1, Method var2, Object[] var3, Object var4) **throws** Throwable;
3. }

* afterReturning也是一个回调函数，AOP应用需要在这个接口实现中提供切面增强的具体设计，在这个Advice通知被正确配置后，在目标方法调用结束并成功返回的时候，接口会被Spring AOP回调

1. **public** **interface** ThrowsAdvice **extends** AfterAdvice {
2. }

* 对于ThrowsAdvice，并没有指定需要实现的接口方法，它在抛出异常时被回调，这个回调是AOP使用反射机制来完成的

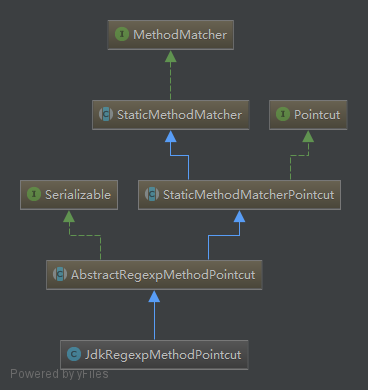
### Pointcut切点

1、Pointcut(切点)决定Advice通知应该作用于哪个连接点，也就是说通过Pointcut来定义需要增强的方法的集合，这些集合的选取可以按照一定的规则来完成。

1. **public** **interface** Pointcut {
2. Pointcut TRUE = TruePointcut.INSTANCE;
4. ClassFilter getClassFilter();
6. MethodMatcher getMethodMatcher();
7. }

* 通过getMethodMatcher返回一个MethodMatcher，对于Point的匹配判断功能，具体是由这个返回的MethodMatcher来完成的。即由这个MethodMatcher来判断是否需要对当前方法调用进行增强，或者是否需要对当前调用方法应用到配置好的Advice通知

2、在Pointcut的继承关系中，以正则表达式切点JdkRegexpMethodPointcut的实现原理为例，来具体了解切点Pointcut的工作原理。JdkRegexpMethodPointcut类通过正则表达式对方法名进行匹配的功能



3、StaticMethodMatcherPointcut如下

1. **public** **abstract** **class** StaticMethodMatcherPointcut **extends** StaticMethodMatcher **implements** Pointcut {
2. **private** ClassFilter classFilter;
4. **public** StaticMethodMatcherPointcut() {
5. **this**.classFilter = ClassFilter.TRUE;
6. }
8. **public** **void** setClassFilter(ClassFilter classFilter) {
9. **this**.classFilter = classFilter;
10. }
12. **public** ClassFilter getClassFilter() {
13. **return** **this**.classFilter;
14. }
16. **public** **final** MethodMatcher getMethodMatcher() {
17. **return** **this**;
18. }
19. }

* 返回的MethodMatcher就是StaticMethodMatcherPointcut的实例本身
* classFilter也是一个单例

1. **public** **interface** ClassFilter {
2. ClassFilter TRUE = TrueClassFilter.INSTANCE;
4. **boolean** matches(Class<?> var1);
5. }
6. **class** TrueClassFilter **implements** ClassFilter, Serializable {
7. **public** **static** **final** TrueClassFilter INSTANCE = **new** TrueClassFilter();
9. **private** TrueClassFilter() {
10. }
12. **public** **boolean** matches(Class<?> clazz) {
13. **return** **true**;
14. }
16. **private** Object readResolve() {
17. **return** INSTANCE;
18. }
20. **public** String toString() {
21. **return** "ClassFilter.TRUE";
22. }
23. }

### Advisor切面

1、完成对目标方法的切面增强设计(Advice)和关注点的设计(Pointcut)以后，需要一个对象把它们接合起来，完成这个作用的就是Advisor(切面)

2、通过Advisor，可以定义应该使用哪个增强并在哪个切点使用它

3、以DefaultPointcutAdvisor进行分析

1. **public** **class** DefaultPointcutAdvisor **extends** AbstractGenericPointcutAdvisor **implements** Serializable {
2. **private** Pointcut pointcut;
4. **public** DefaultPointcutAdvisor() {
5. **this**.pointcut = Pointcut.TRUE;
6. }
8. **public** DefaultPointcutAdvisor(Advice advice) {
9. **this**(Pointcut.TRUE, advice);
10. }
12. **public** DefaultPointcutAdvisor(Pointcut pointcut, Advice advice) {
13. **this**.pointcut = Pointcut.TRUE;
14. **this**.pointcut = pointcut;
15. **this**.setAdvice(advice);
16. }
18. **public** **void** setPointcut(Pointcut pointcut) {
19. **this**.pointcut = pointcut != **null**?pointcut:Pointcut.TRUE;
20. }
22. **public** Pointcut getPointcut() {
23. **return** **this**.pointcut;
24. }
26. **public** String toString() {
27. **return** **this**.getClass().getName() + ": pointcut [" + **this**.getPointcut() + "]; advice [" + **this**.getAdvice() + "]";
28. }
29. }

* pointcut默认被设置为Pointcut.Ture，这个Pointcut.True在Pointcnt接口中被定义
* TruePointcut的定义如下

1. **class** TruePointcut **implements** Pointcut, Serializable {
2. **public** **static** **final** TruePointcut INSTANCE = **new** TruePointcut();
4. **private** TruePointcut() {
5. }
7. **public** ClassFilter getClassFilter() {
8. **return** ClassFilter.TRUE;
9. }
11. **public** MethodMatcher getMethodMatcher() {
12. **return** MethodMatcher.TRUE;
13. }
15. **private** Object readResolve() {
16. **return** INSTANCE;
17. }
19. **public** String toString() {
20. **return** "Pointcut.TRUE";
21. }
22. }

* 可以看到TruePoint的INSTANCE是一个单例，其中methodMatcher实现返回的也是一个单例

1. **public** **interface** MethodMatcher {
2. MethodMatcher TRUE = TrueMethodMatcher.INSTANCE;
4. **boolean** matches(Method var1, Class<?> var2);
6. **boolean** isRuntime();
8. **boolean** matches(Method var1, Class<?> var2, Object... var3);
9. }

* TrueMethodMatcher如下

1. **class** TrueMethodMatcher **implements** MethodMatcher, Serializable {
2. **public** **static** **final** TrueMethodMatcher INSTANCE = **new** TrueMethodMatcher();
4. **private** TrueMethodMatcher() {
5. }
7. **public** **boolean** isRuntime() {
8. **return** **false**;
9. }
11. **public** **boolean** matches(Method method, Class<?> targetClass) {
12. **return** **true**;
13. }
15. **public** **boolean** matches(Method method, Class<?> targetClass, Object... args) {
16. **throw** **new** UnsupportedOperationException();
17. }
19. **public** String toString() {
20. **return** "MethodMatcher.TRUE";
21. }
23. **private** Object readResolve() {
24. **return** INSTANCE;
25. }
26. }

## Spring AOP的设计与实现

### JVM的动态代理特性

1、在Spring AOP实现中，使用的核心技术是动态代理，而这种动态代理实际上是JDK的一个特性，通过JDK动态代理特性，可以为任意Java对象创建代理对象，对于具体使用来说，这个特性是通过Java Reflection API来完成的