# Spring AOP

1. Spring AOP基础
2. AOP概述

使用场景：性能检测、访问控制、事物管理等；

AOP，面向切面的编程；

将分散在业务代码的相同代码（具备横切特性的代码--增强Advice），抽离出来，然后通过相关技术（切面）融合到业务逻辑代码（切点、连接点、目标对象），这便是AOP需要解决的问题。

1.1、连接点（Join Point）

1.2、切点（Point Cut）

1.3、增强（Advice）

1.4、目标对象（Target）

1.5、引介（Introduction）

1.6、织入（Weaving）

1.7、代理（Proxy）--JDK动态代理，基于接口；CGLib提供的动态代理，基于类的；

1.8、切面（Aspect）

AOP的重点工作在于如何（利用Proxy代理技术-Jdk、CGLib）将**增强（具有横切特性的代码，比如事物）**用于**目标对象（Target，比如Dao或是Service）**的**连接点（比如具体的一个Method）**上；

第一，如何通过切点和增强定位到连接点（硬编码；在spring 中，我们使用XML配置）；

第二，如何在增强中编写切面的代码（增强，是放在业务代码的什么位置：前Before、后After、环绕Around、异常Exception）；

1. AOP基础知识

Spring AOP使用动态代理在运行期织入增强的代码，有二种代理机制，基于JDK的动态代码，基于CGLib的动态代理。

（1）具有横切代码的示例：

|  |
| --- |
| **public** **void** removeTopic(**int** topicId) {  /\*\*  \* 具有横切特性的代码，对方法启动性能监控  \*/  PerformanceMonitor.*begin*("com.springaop.proxy.ForumServiceImpl.removeTopic");  System.*out*.println("模拟删除Topic记录:"+topicId);  **try** {  Thread.*currentThread*().*sleep*(20);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  /\*\*  \* 具有横切特性的代码，对方法关闭性能监控  \*/  PerformanceMonitor.*end*();  } |
| **存在问题：**  如果需要给方法增加性能监控，那么必须调整代码，在方法的前后增加开启性能和结束性能的代码；  **这些非业务逻辑的代码，破坏了业务逻辑的纯粹性；**  **如何改进：**  （1）我们希望通过代理的方式，讲业务类中的横切代码，**完全移除**。  （2）通过JDK动态代理技术或CGLib动态代理技术，将**横切代码动态组织**到**目标方法**的**相应位置**。 |

1. JDK的动态代理

JDK1.3，Java提供动态代理技术，允许开发者在运行期间创建接口的代理实现类；

动态代理涉及java.lang.reflect包中的二个类，Proxy和InvocationHandler。

其中，InvocationHandler是一个接口，可以通过实现该接口定义横切逻辑，并通过**反射**机制调用目标类的代码，动态讲横切逻辑和业务逻辑编织在一起。即：由InvocationHandler接口实现了将【增强】织入到【目标对象】的【连接点】。

|  |
| --- |
| **public** **class** PerformaceHandler **implements** InvocationHandler {  /\*\*  \* 目标方法对象  \*/  **private** Object target;  **public** PerformaceHandler(Object target) {  **this**.target = target;  }  /\*\*  \* 原来分散在业务逻辑中的横切逻辑，被抽离到了invoke方法中  \*/  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  /\*\*  \* 横切逻辑，开启方法的性能监控  \*/  PerformanceMonitor.*begin*(target.getClass().getName() + "."  + method.getName());  /\*\*  \* 通过反射的机制，调用业务逻辑  \*/  Object obj = method.invoke(target, args);  /\*\*  \* 横切逻辑，关闭方法的性能监控  \*/  PerformanceMonitor.*end*();  **return** obj;  }  } |
| /\*\*  \* 使用JDK动态代理  \* 具有横切特性的代码，已经完全聪业务逻辑中解耦  \*/  // 第一步：希望被代理的目标业务类  ForumService target = **new** ForumServiceImpl();  // 第二步：将目标业务类和横切代码编织到一起  PerformaceHandler handler = **new** PerformaceHandler(target);  // 第三步：根据编织了目标业务类代码和性能监视横切逻辑的InvocationHandler实例，创建代理实例  ForumService proxy = (ForumService) Proxy.*newProxyInstance*(target.getClass().getClassLoader(),  target.getClass().getInterfaces(), handler);  // 第四步：调用代理实例  proxy.removeForum(10);  proxy.removeTopic(1012); |
| 解决问题：  通过JDK的动态代理机制，实现了横切代码和业务代理的抽离。在代码运行时刻，动态织入到目标对象上。利用的原理是反射。  存在问题：  **只能为接口创建代理实例！** |

1. CGLib动态代理

|  |
| --- |
| CGLib(Code Generation Library),一个强大的，高性能的的Code生产类库。可以在运行期扩展Java类或实现Java接口。被AOP使用很多。  CGLib是通过一个小而快的字节码处理框架（ASM），来转换字节码并生成新的类。 |
| **public** **class** CglibProxy **implements** MethodInterceptor {  **private** Enhancer enhancer = **new** Enhancer();  **public** Object getProxy(Class<?> clazz) {  /\*\*  \* 设置需要创建子类的类  \*/  enhancer.setSuperclass(clazz);  enhancer.setCallback(**this**);  /\*\*  \* 动过字节码技术，动态创建子类实例  \*/  **return** enhancer.create();  }    **public** Object intercept(Object obj, Method method, Object[] args,  MethodProxy proxy) **throws** Throwable {  /\*\*  \* 横切逻辑，织入性能监控代码到业务逻辑中  \*/  PerformanceMonitor.*begin*(obj.getClass().getName() + "."  + method.getName());  /\*\*  \* 拦截父类所有的方法调用，通过代理类调用父类中的方法；实现业务逻辑的调用  \*/  Object result = proxy.invokeSuper(obj, args);  /\*\*  \* 横切逻辑，织入性能监控代码到业务逻辑中  \*/  PerformanceMonitor.*end*();  **return** result;  }  } |
| /\*\*  \* 使用CGLib动态代理  \* 解决jdk动态代理的问题：直接为类对象，创建代理实例  \*/  CglibProxy proxy = **new** CglibProxy();  // 通过动态生成子类的方式，创建代理类  ForumService forumService = (ForumService) proxy.getProxy(ForumServiceImpl.**class**);  forumService.removeForum(10);  forumService.removeTopic(1023); |
| 解决问题：  可以直接为类创建代理对象。 |

1. 代理知识小结

Spring AOP的底层通过JDK动态代理或CGLib动态代理技术为目标Bean织入横切逻辑。但是这种方式存在三个明显的不足，需要改进：

第一，目标类的**所有方法**都增加了“性能监视”的横切逻辑，但是，我们可能只希望对业务类中的**指定方法**增加横切逻辑。

第二，我们通过**硬编码**的方式，指定了织入横切逻辑的织入点，即在业务逻辑的开始前和开始后，织入代码。

第三，我们**手工编写代理实例的创建过程**，为不同类创建代理时，需要分别编写相应的创建代码，无法做到通用。例如控制层调用业务层，必须硬编码创建代理对象，才能将增强织入到目标对象的连接点。

所以，Spring AOP的主要工作就是围绕以上三点展开： 第一,Spring AOP通过切点（Pointcut）指定在那些类的那些方法上织入横切逻辑。

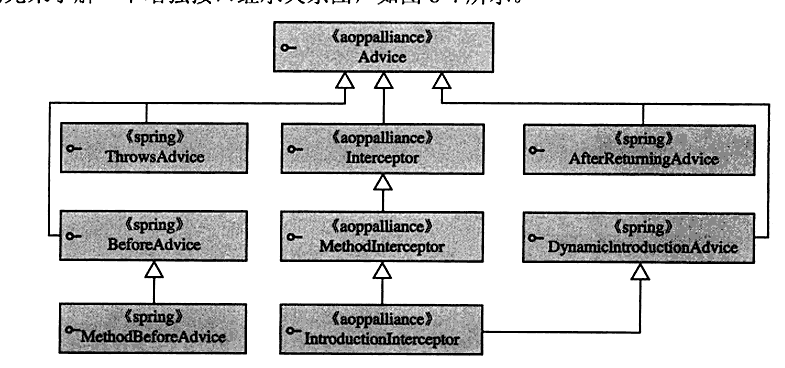
第二，Spring AOP通过Advice（增强）描述横切逻辑和方法的具体织入点（方法前、后、二端等）。

第三，Spring AOP通过切面（Advisor）讲PointCut和Advice二者组装起来。

1. 创建增强类（Advice）

Spring通过**增强类定义横切逻辑**，同时还包括了在方法的那一点上加入横切代码的**方位**信息，所以，增强即包括了横切逻辑，还包含部分连接点信息。

* 前置增强
* 后置增强
* 环绕增强
* 异常抛出增强
* 引介增强



1. 前置增强

|  |
| --- |
| **1、定义前置增强类：**  **public** **class** GreetingBeforeAdvice **implements** MethodBeforeAdvice {  **public** **void** before(Method method, Object[] args, Object obj)  **throws** Throwable {  String clientName = (String) args[0];  System.*out*.println("How are you！Mr." + clientName + ".");  }  } |
| **2、定义目标接口，基于JDK动态代理，只能基于接口实现：**  **纯业务层。**  **public** **interface** Waiter {  **void** greetTo(String name);  **void** serveTo(String name);  }  **public** **class** NaiveWaiter **implements** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("greet to "+name+"...");  }  **public** **void** serveTo(String name){  System.*out*.println("serving "+name+"...");  }  } |
| 3、基于Spring AOP配置，实现增强  <!-- 前置增强 -->  <bean id=*"greetingBefore"* class=*"com.springaop.advice.GreetingBeforeAdvice"* />  <!-- 代理的目标对象，基于接口的动态代理，JDK动态代理-->  <bean id=*"target"* class=*"com.springaop.advice.NaiveWaiter"* />  <!--1. greetingBefore -->  <!-- p:proxyInterfaces:指定代理的接口 -->  <!-- p:target-ref:制定对那个Bean进行代理 -->  <!-- p:interceptorNames:指定使用的增强 -->  <bean id=*"waiter"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:proxyInterfaces=*"com.springaop.advice.Waiter"* p:target-ref=*"target"*  p:interceptorNames=*"greetingBefore"* /> |
| 4、测试验证：不看配置文件，无法从测试代码中发现使用了增强  @Test  **public** **void** testBeforeAdvice2() {  System.*out*.println("---------------前置增强----------------------");  Waiter waiter = (Waiter) context.getBean("waiter");  waiter.greetTo("John");  } |

1. 后置增强

|  |
| --- |
| **public** **class** GreetingAfterAdvice **implements** AfterReturningAdvice {  **public** **void** afterReturning(Object returnObj, Method method, Object[] args,  Object obj) **throws** Throwable {  System.*out*.println("Please enjoy yourself!");  }  } |
| **public** **interface** Waiter {  **void** greetTo(String name);  **void** serveTo(String name);  }  **public** **class** NaiveWaiter **implements** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("greet to "+name+"...");  }  **public** **void** serveTo(String name){  System.*out*.println("serving "+name+"...");  }  } |
| <bean id="greetingAfter" class="com.springaop.advice.GreetingAfterAdvice" />  <bean id="waiter1" class="org.springframework.aop.framework.ProxyFactoryBean" p:proxyInterfaces="com.springaop.advice.Waiter" p:target-ref="target"  p:interceptorNames="greetingBefore,greetingAfter" /> |
| @Test  **public** **void** testBeforeAndAfterAdvice() {  System.*out*.println("---------------前置增强&后置增强----------------------");  Waiter waiter = (Waiter) context.getBean("waiter1");  waiter.greetTo("John");  } |

（3）环绕增强

|  |
| --- |
| **public** **class** GreetingInterceptor **implements** MethodInterceptor {  **public** Object invoke(MethodInvocation invocation) **throws** Throwable {  Object[] args = invocation.getArguments();  String clientName = (String)args[0];  System.*out*.println("How are you！Mr."+clientName+".");  Object obj = invocation.proceed();  System.*out*.println("Please enjoy yourself!");  **return** obj;  }  } |
| **public** **interface** Waiter {  **void** greetTo(String name);  **void** serveTo(String name);  }  **public** **class** NaiveWaiter **implements** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("greet to "+name+"...");  }  **public** **void** serveTo(String name){  System.*out*.println("serving "+name+"...");  }  } |
| <!-- 环绕增强 -->  <bean id=*"greetingAround"* class=*"com.springaop.advice.GreetingInterceptor"* />  <!-- 3. around advice -->  <bean id=*"waiter2"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:proxyInterfaces=*"com.springaop.advice.Waiter"* p:target-ref=*"target"*  p:interceptorNames=*"greetingAround"* /> |
| @Test  **public** **void** testInterceptorAdvice() {  System.*out*.println("---------------环绕增强----------------------");  Waiter waiter = (Waiter) context.getBean("waiter2");  waiter.greetTo("John");  } |

1. 抛出异常增强--(基于CGLib动态代理，基于类的代理)

|  |
| --- |
| **public** **class** TransactionManager **implements** ThrowsAdvice {  **public** **void** afterThrowing(Method method, Object[] args, Object target,  Exception ex) **throws** Throwable {  System.*out*.println("-----------");  System.*out*.println("method:" + method.getName());  System.*out*.println("抛出异常:" + ex.getMessage());  System.*out*.println("成功回滚事务。");  }  } |
| **public** **class** ForumService {  **public** **void** removeForum(**int** forumId) {  // do sth...  **throw** **new** RuntimeException("运行异常。");  }  **public** **void** updateForum(Forum forum) **throws** Exception{  // do sth...  **throw** **new** SQLException("数据更新操作异常。");  }  } |
| <!-- 代理的目标对象，基于类的动态代理，CGLib动态代理-->  <bean id=*"forumServiceTarget"* class=*"com.springaop.advice.ForumService"* />  <!-- 4. 异常抛出增强 -->  <!-- 因为com.springaop.advice.ForumService是类，所以本次使用CGLib代理;所以，p:proxyTargetClass=true -->  <bean id=*"transactionManager"* class=*"com.springaop.advice.TransactionManager"* />  <bean id=*"forumService"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interceptorNames=*"transactionManager"* p:target-ref=*"forumServiceTarget"*  p:proxyTargetClass=*"true"* /> |
| @Test  **public** **void** testThrowsAdvice() {  System.*out*.println("---------------抛出异常增强----------------------");  ForumService forumService = (ForumService) context  .getBean("forumService");  **try** {  forumService.removeForum(10);  } **catch** (Exception e) {  }  **try** {  forumService.updateForum(**new** Forum());  } **catch** (Exception e) {  }  } |

1. 引介增强--动态给目标对象，增加相关功能;

|  |
| --- |
| 引介增强是一种比较特殊的的增强类型，它不是在目标方法周围织入增强，而是为目标类创建新的方法和属性，所以引介增强的连接点是类级别的，而非方法级别。  通过引介增强，可以为目标类增加一个接口的实现，即原来目标类未实现某个接口，通过引介增强可以为目标类创建实现该接口的代理。 |
| **public** **interface** Monitorable {  **void** setMonitorActive(**boolean** active);  } |
| **public** **class** ControllablePerformaceMonitor **extends**  DelegatingIntroductionInterceptor **implements** Monitorable {  **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** ThreadLocal<Boolean> MonitorStatusMap = **new** ThreadLocal<Boolean>();  **public** **void** setMonitorActive(**boolean** active) {  MonitorStatusMap.set(active);  }  **public** Object invoke(MethodInvocation mi) **throws** Throwable {  Object obj = **null**;  **if** (MonitorStatusMap.get() != **null** && MonitorStatusMap.get()) {  PerformanceMonitor.*begin*(mi.getClass().getName() + "."  + mi.getMethod().getName());  obj = **super**.invoke(mi);  PerformanceMonitor.*end*();  } **else** {  obj = **super**.invoke(mi);  }  **return** obj;  }  } |
| **public** **class** ForumService {  **public** **void** removeTopic(**int** topicId) {  System.*out*.println("模拟删除Topic记录:"+topicId);  **try** {  Thread.*currentThread*().*sleep*(20);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  } |
| <bean id=*"pmonitor"* class=*"com.springaop.advice.introduce.ControllablePerformaceMonitor"* />  <!-- 代理的目标对象，基于类的动态代理，CGLib动态代理-->  <bean id=*"forumServiceTarget"* class=*"com.springaop.advice.introduce.ForumService"* />  <bean id=*"forumService"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interfaces=*"com.springaop.advice.introduce.Monitorable"* p:target-ref=*"forumServiceTarget"*  p:interceptorNames=*"pmonitor"* p:proxyTargetClass=*"true"* /> |
| @Test  **public** **void** testIntroduceAdvice() {  System.*out*.println("---------------引介增强----------------------");  ForumService forumService = (ForumService) context  .getBean("forumService");  forumService.removeForum(10);  Monitorable moniterable = (Monitorable) forumService;  moniterable.setMonitorActive(**true**);  forumService.removeForum(10);  } |

1. 创建切面-Advisor=Pointcut(切点) + Adviser（增强）

Spring AOP围绕三点来解决：

（1）第一，对业务类中的指定方法增加横切逻辑。

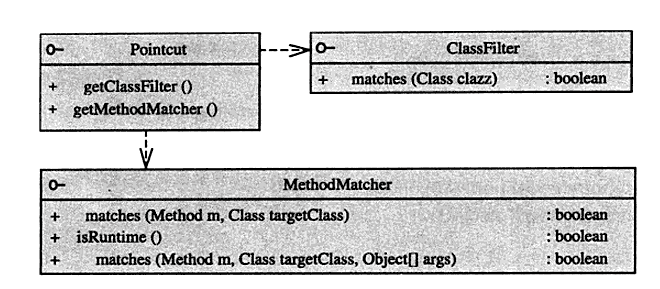
（2）第二，不是硬编码，织入横切逻辑。

（3）第三，通用方式增强代码被织入到目标类；

增强解决了第二和第三。

如何解决问题一，我们就需要使用切点进行目标连接点的定位。

**增强提供了方位信息，前、后、环绕、异常等；而切点进一步描述织入到那些类的那些方法上。**



（Point Cut类关系图）

Spring提供了6种类型的切点：

* 静态方法切点
* 动态方法切点
* 注解切点
* 表达式切点
* 流程切点
* 复合切点

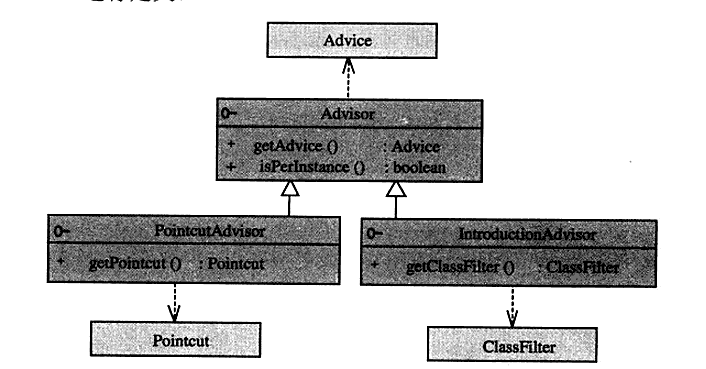
1. 切面类型

增强包含横切代码，有包含**部分连接点**信息（方法前、后等），所以我们可以使用增强生产一个切面（**增强就是一个简单的切面**）。

切点仅包含目标类连接点的部分信息（类和方法的定位），所以仅有切点，我们无法制作一个切面。

Spring使用Advisor接口标识切面，一个切面包含了横切代码和连接点信息，**切面可以分为3类**：

* 一般切面（Advisor）：它仅仅包含一个Advice，但是因为**横切面太宽泛**，因为他包含了横切连接点是所有类的所有方法。**很少使用**。Advice是一个简单的Advisor。
* **切点切面**（PointcutAdvisor）：包含Advice和Pointcut。提供更具灵活特性的切面。具有实际意义的切面。学习的重点。
* 引介切面（IntroductionAdvisor）：和引介增强搭配使用。专门配合引介增强使用。非重点。



（Advisor类关系图）

1. 静态普通方法名匹配切点切面

|  |
| --- |
| **1、定义切面**  **public** **class** GreetingAdvisor **extends** StaticMethodMatcherPointcutAdvisor {  **private** **static** **final** **long** *serialVersionUID* = 1L;  **public** **boolean** matches(Method method, Class<?> clazz) {  **return** "greetTo".equals(method.getName());  }  **public** ClassFilter getClassFilter() {  **return** **new** ClassFilter() {  **public** **boolean** matches(Class<?> clazz) {  **return** Waiter.**class**.isAssignableFrom(clazz);  }  };  }  } |
| **2、定义增强**  **public** **class** GreetingBeforeAdvice **implements** MethodBeforeAdvice {    **public** **void** before(Method method, Object[] args, Object obj)  **throws** Throwable {  String clientName = (String) args[0];  System.*out*.println(obj.getClass().getName() + "." + method.getName());  System.*out*.println("How are you！Mr." + clientName + ".");  }  } |
| **3、定义目标对象**  **public** **class** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("waiter greet to "+name+"...");  }  **public** **void** serveTo(String name){  System.*out*.println("waiter serving "+name+"...");  }  }  **public** **class** Seller {  **public** **void** greetTo(String name) {  System.*out*.println("seller greet to " + name + "...");  }  } |
| 4、配置  <!-- Target：目标对象 -->  <bean id=*"waiterTarget"* class=*"com.springaop.advisor.Waiter"* />  <bean id=*"sellerTarget"* class=*"com.springaop.advisor.Seller"* />  <!-- Advice：前置增强 -->  <bean id=*"greetingAdvice"* class=*"com.springaop.advisor.GreetingBeforeAdvice"* />  <!-- Advisor：切面-->  <bean id=*"greetingAdvisor"* class=*"com.springaop.advisor.GreetingAdvisor"*  p:advice-ref=*"greetingAdvice"* />  <!-- 1.公共配置信息 -->  <bean id=*"parent"* abstract=*"true"*  class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interceptorNames=*"greetingAdvisor"* p:proxyTargetClass=*"true"* />  <bean id=*"waiter"* parent=*"parent"* p:target-ref=*"waiterTarget"* />  <bean id=*"seller"* parent=*"parent"* p:target-ref=*"sellerTarget"* /> |
| 5、测试类  @Test  **public** **void** testStaticMethodAdvisor() {  System.*out*  .println("-------------------测试，静态方法匹配切点切面------------------");  Waiter waiter = (Waiter) context.getBean("waiter");  Seller seller = (Seller) context.getBean("seller");  // 进入切面引入的增强逻辑  waiter.greetTo("John");  // 不会进入切面引入的增强逻辑  waiter.serveTo("John");  // 不会进入切面引入的增强逻辑  seller.greetTo("John");  } |

1. 静态正则表达式方法匹配切点切面

|  |
| --- |
| **1、目标对象**  **public** **class** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("waiter greet to "+name+"...");  }  **public** **void** serveTo(String name){  System.*out*.println("waiter serving "+name+"...");  }  } |
| **2、前置增强**  **public** **class** GreetingBeforeAdvice **implements** MethodBeforeAdvice {  **public** **void** before(Method method, Object[] args, Object obj)  **throws** Throwable {  String clientName = (String) args[0];  System.*out*.println(obj.getClass().getName() + "." + method.getName());  System.*out*.println("How are you！Mr." + clientName + ".");  }  } |
| 3、配置  <!-- 2.正则表达式方法名匹配切面 -->  <bean id=*"regexpAdvisor"* class=*"org.springframework.aop.support.RegexpMethodPointcutAdvisor"*  p:advice-ref=*"greetingAdvice"*>  <!-- 限定方法名的匹配模式串 -->  <property name=*"patterns"*>  <list>  <!-- 匹配模式串 -->  <value>.\*greet.\*</value>  </list>  </property>  </bean>  <bean id=*"waiter1"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interceptorNames=*"regexpAdvisor"* p:target-ref=*"waiterTarget"*  p:proxyTargetClass=*"true"* /> |
| 4、测试类  @Test  **public** **void** testRegexpAdvisor() {  System.*out*  .println("-------------------测试，静态正则表达式匹配切点切面------------------");  Waiter waiter = (Waiter) context.getBean("waiter1");  // 进入切面引入的增强逻辑  waiter.greetTo("John");  // 不会进入切面引入的增强逻辑  waiter.serveTo("John");  } |
| 备注:  相比对静态普通方法名匹配切点切面，静态正则使用配置中的正则表达式，代替Advisor类。 |

1. 动态切点切面

|  |
| --- |
| **1、切面**  **public** **class** GreetingDynamicPointcut **extends** DynamicMethodMatcherPointcut {  /\*\*  \* 配置客户白名单，只对白名单客户的方法启用增强  \*/  **private** **static** List<String> *specialClientList* = **new** ArrayList<String>();    **static** {  *specialClientList*.add("John");  *specialClientList*.add("Tom");  }  /\*\*  \* 对类进行静态切点检查  \*/  **public** ClassFilter getClassFilter() {  **return** **new** ClassFilter() {  **public** **boolean** matches(Class<?> clazz) {  System.*out*.println("调用getClassFilter()对" + clazz.getName()  + "做静态检查.");  **return** Waiter.**class**.isAssignableFrom(clazz);  }  };  }  /\*\*  \* 对方法进行静态切点检查  \*/  **public** **boolean** matches(Method method, Class<?> clazz) {  System.*out*.println("调用matches(method,clazz)对" + clazz.getName() + "."  + method.getName() + "做静态检查.");  **return** "greetTo".equals(method.getName());  }  /\*\*  \* 对方法进行动态切点检查  \* 只对目标方法为greetTo的，且clientName在白名单的启用增强  \*/  **public** **boolean** matches(Method method, Class<?> clazz, Object[] args) {  System.*out*.println("调用matches(method,clazz)对" + clazz.getName() + "."  + method.getName() + "做动态检查.");  String clientName = (String) args[0];  **return** *specialClientList*.contains(clientName);  }  } |
| **2、前置增强**  **public** **class** GreetingBeforeAdvice **implements** MethodBeforeAdvice {  **public** **void** before(Method method, Object[] args, Object obj)  **throws** Throwable {  String clientName = (String) args[0];  System.*out*.println(obj.getClass().getName() + "." + method.getName());  System.*out*.println("How are you！Mr." + clientName + ".");  }  } |
| **3、目标对象**  **public** **class** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("waiter greet to "+name+"...");  }  **public** **void** serveTo(String name){  System.*out*.println("waiter serving "+name+"...");  }  } |
| 4、配置  <!-- 3.动态切面 -->  <bean id=*"dynamicAdvisor"* class=*"org.springframework.aop.support.DefaultPointcutAdvisor"*>  <property name=*"pointcut"*>  <bean class=*"com.springaop.advisor.GreetingDynamicPointcut"* />  </property>  <property name=*"advice"*>  <bean class=*"com.springaop.advisor.GreetingBeforeAdvice"* />  </property>  </bean>  <bean id=*"waiter2"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interceptorNames=*"dynamicAdvisor"* p:target-ref=*"waiterTarget"*  p:proxyTargetClass=*"true"* /> |
| 5、测试  @Test  **public** **void** testDynamicAdvisor() {  System.*out*.println("-------------------测试，动态切面------------------");  Waiter waiter = (Waiter) context.getBean("waiter2");  // 不会进入切面引入的增强逻辑  waiter.serveTo("Peter");  // 不会进入切面引入的增强逻辑  waiter.greetTo("Peter");  // 不会进入切面引入的增强逻辑  waiter.serveTo("Peter");  // 百名单内的客户，引入切面引入的增强逻辑  // Waiter.greetTo方法，而且是白名单内的，才会进入增强逻辑。  waiter.greetTo("John");  } |

1. 流程切点切面

|  |
| --- |
| **1、切面**  **public** **class** WaiterDelegate {  **private** Waiter waiter;  **public** **void** setWaiter(Waiter waiter) {  **this**.waiter = waiter;  }  **public** Waiter getWaiter() {  **return** waiter;  }  // waiter的方法通过改方法发起的调用  **public** **void** service(String clientName) {  waiter.greetTo(clientName);  waiter.serveTo(clientName);  }  } |
| **2、前置增强**  **public** **class** GreetingBeforeAdvice **implements** MethodBeforeAdvice {  **public** **void** before(Method method, Object[] args, Object obj)  **throws** Throwable {  String clientName = (String) args[0];  System.*out*.println(obj.getClass().getName() + "." + method.getName());  System.*out*.println("How are you！Mr." + clientName + ".");  }  } |
| **3、目标对象**  **public** **class** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("waiter greet to "+name+"...");  }    **public** **void** serveTo(String name){  System.*out*.println("waiter serving "+name+"...");  }  } |
| 4、配置  <!-- 4.控制流程切面 -->  <bean id=*"controlFlowPointcut"* class=*"org.springframework.aop.support.ControlFlowPointcut"*>  <!-- 指定流程切面的类 -->  <constructor-arg type=*"java.lang.Class"*  value=*"com.springaop.advisor.WaiterDelegate"* />  <!-- 制定流程切点的方法 -->  <constructor-arg type=*"java.lang.String"* value=*"service"* />  </bean>  <bean id=*"controlFlowAdvisor"* class=*"org.springframework.aop.support.DefaultPointcutAdvisor"*  p:pointcut-ref=*"controlFlowPointcut"* p:advice-ref=*"greetingAdvice"* />  <bean id=*"waiter3"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interceptorNames=*"controlFlowAdvisor"* p:target-ref=*"waiterTarget"*  p:proxyTargetClass=*"true"* /> |
| 5、测试  @Test  **public** **void** testControlFlowAdvisor() {  System.*out*.println("-------------------测试，流程切面------------------");  Waiter waiter = (Waiter) context.getBean("waiter3");  WaiterDelegate wd = **new** WaiterDelegate();  wd.setWaiter(waiter);  // 不会进入切面引入的增强逻辑  waiter.serveTo("Peter");  // 不会进入切面引入的增强逻辑  waiter.greetTo("Peter");  // 进入切面引入的增强逻辑  wd.service("Peter");  } |

1. 复合切点切面

|  |
| --- |
| **1、切面**  **public** **class** GreetingComposablePointcut {  **public** Pointcut getIntersectionPointcut() {  /\*\*  \* 创建一个复合切点  \*/  ComposablePointcut cp = **new** ComposablePointcut();  /\*\*  \* 创建一个流程切点  \*/  Pointcut pt1 = **new** ControlFlowPointcut(WaiterDelegate.**class**, "service");  /\*\*  \* 创建一个方法名切点  \*/  NameMatchMethodPointcut pt2 = **new** NameMatchMethodPointcut();  pt2.addMethodName("greetTo");  /\*\*  \* 将二个切点进行交集操作  \*/  **return** cp.intersection(pt1).intersection((Pointcut) pt2);  }  } |
| **2、前置增强**  **public** **class** GreetingBeforeAdvice **implements** MethodBeforeAdvice {  **public** **void** before(Method method, Object[] args, Object obj)  **throws** Throwable {  String clientName = (String) args[0];  System.*out*.println(obj.getClass().getName() + "." + method.getName());  System.*out*.println("How are you！Mr." + clientName + ".");  }  } |
| **3、目标对象**  **public** **class** Waiter {  **public** **void** greetTo(String name) {  System.*out*.println("waiter greet to "+name+"...");  }    **public** **void** serveTo(String name){  System.*out*.println("waiter serving "+name+"...");  }  } |
| 4、配置  <!-- 5.复合切点切面 -->  <bean id=*"gcp"* class=*"com.springaop.advisor.GreetingComposablePointcut"* />  <bean id=*"composableAdvisor"* class=*"org.springframework.aop.support.DefaultPointcutAdvisor"*  p:pointcut=*"#{gcp.intersectionPointcut}"* p:advice-ref=*"greetingAdvice"* />  <bean id=*"waiter4"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interceptorNames=*"composableAdvisor"* p:target-ref=*"waiterTarget"*  p:proxyTargetClass=*"true"* /> |
| 5、测试  @Test  **public** **void** testComposableAdvisor() {  System.*out*.println("-------------------测试，复合切面------------------");  Waiter waiter = (Waiter) context.getBean("waiter4");  WaiterDelegate wd = **new** WaiterDelegate();  wd.setWaiter(waiter);  // 不会进入切面引入的增强逻辑  waiter.serveTo("Peter");  // 不会进入切面引入的增强逻辑  waiter.greetTo("Peter");  /\*\*  \* 调用waiter.greetTo，不会进入切面引入的增强逻辑  \* 调用waiter.serveTo，进入切面引入的增强逻辑  \*/  wd.service("Peter");  } |

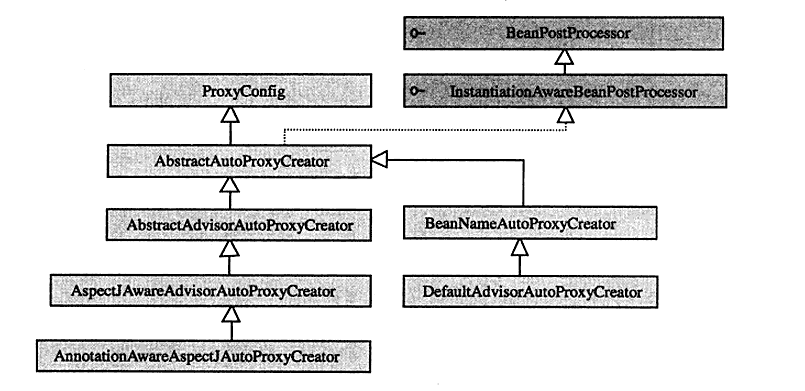
1. 引介切面

|  |
| --- |
| **1、增强**  **public** **class** ControllablePerformaceMonitor **extends**  DelegatingIntroductionInterceptor **implements** Monitorable {  **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** ThreadLocal<Boolean> MonitorStatusMap = **new** ThreadLocal<Boolean>();  **public** **void** setMonitorActive(**boolean** active) {  MonitorStatusMap.set(active);  }  **public** Object invoke(MethodInvocation mi) **throws** Throwable {  Object obj = **null**;  /\*\*  \* 只有在监听打开的时候，才启用方法的性能检测  \*/  **if** (MonitorStatusMap.get() != **null** && MonitorStatusMap.get()) {  PerformanceMonitor.*begin*(mi.getClass().getName() + "."  + mi.getMethod().getName());  obj = **super**.invoke(mi);  PerformanceMonitor.*end*();  } **else** {  obj = **super**.invoke(mi);  }  **return** obj;  }  } |
| **2、目标对象**  **public** **class** ForumService {  **public** **void** removeTopic(**int** topicId) {  System.*out*.println("模拟删除Topic记录:"+topicId);  **try** {  Thread.*currentThread*().*sleep*(20);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  **public** **void** removeForum(**int** forumId) {  System.*out*.println("模拟删除Forum记录:"+forumId);  **try** {  Thread.*currentThread*().*sleep*(40);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  } |
| 3、配置  <!--6.引介切面-->  <bean id=*"introduceAdvisor"*  class=*"org.springframework.aop.support.DefaultIntroductionAdvisor"*>  <constructor-arg>  <bean class=*"com.springaop.advice.introduce.ControllablePerformaceMonitor"* />  </constructor-arg>  </bean>  <bean id=*"forumServiceTarget"* class=*"com.springaop.advice.introduce.ForumService"* />  <bean id=*"forumService"* class=*"org.springframework.aop.framework.ProxyFactoryBean"*  p:interceptorNames=*"introduceAdvisor"*  p:target-ref=*"forumServiceTarget"*  p:proxyTargetClass=*"true"*/> |
| 4、测试  @Test  **public** **void** testIntroduceAdvisor() {  System.*out*.println("-------------------测试，引介切面------------------");  ForumService forumService = (ForumService) context.getBean("forumService");  // 不会进入切面引入的增强逻辑  forumService.removeForum(10);  Monitorable moniterable = (Monitorable) forumService;  moniterable.setMonitorActive(**true**);  // 进入切面引入的增强逻辑  forumService.removeForum(10);  } |

1. 自动创建代理

每一个需要被代理的Bean都需要创建一个org.springframework.aop.framework.ProxyFactoryBean，进行配置。用于创建织入切面的代理。配置起来比较麻烦，特别是对于大型系统来说。

幸运的是，Spring提供了自动代理机制，让容器为我们自动生成代理。



（自动代理创建器实现类图）

6、小结

AOP为程序提供了一个崭新思考角度，可以将重复性的**横切代码抽取到统一的模块**中，解决重复性代码问题。

Spring使用JDK的基于接口的**动态代理**，或是使用CGLib的动态代理，实现了不需要安装任何编译器，在**运行期间织入增强**。JDK动态代理在代理对象性能强于CGLib，如果对象是单例，推荐使用CGLib。

**Spring只支持方法级别的织入增强**。Spring提供了五种增强，**前、后、异常、环绕、以及引介**；增强实际上就是一个简单的切面。但是它的切点是方法的相对位置信息，所以一般增强+切点才会更有实际意义。

**切面是增强和切点的复合体**。我们可以通过ProxyBeanFactory将切面织入到不同的目标类中；为了解决配置上的繁琐，使用自动代理创建器，可以实现容器中的切面织入到目标Bean中。

1. 基于AspectJ和schema的AOP