阅读笔记3

第四章 面向切面的Spring

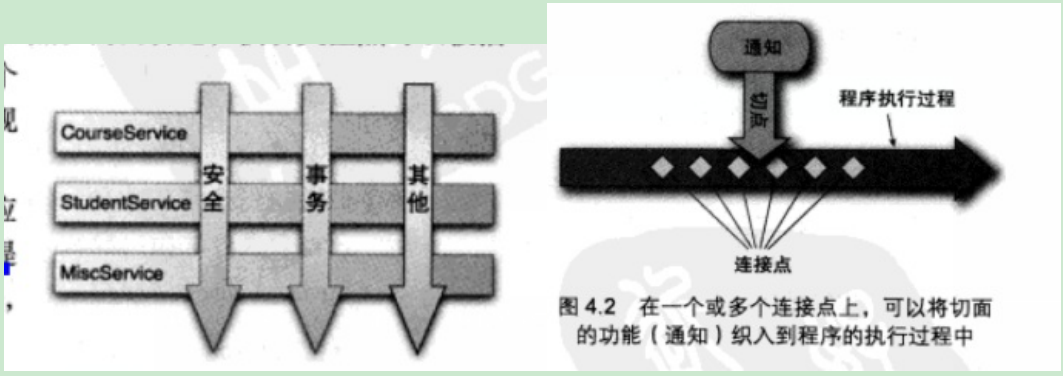


图 4.2 在一个或多个连接点上，可以将切面的功能（通知）织入到程序的执行过程中

面向切面编程

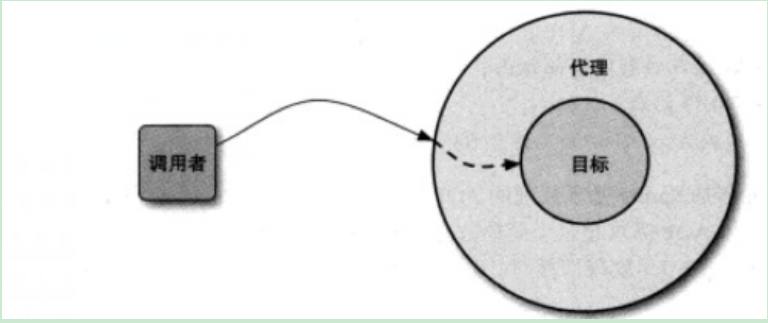
这里单独讲一个 织入

织入就是将切面应用到目标对象来创建新的代理对象的过程

- 编译期——切面在目标类编译时被织入。这种方式需要特殊的编译器。AspectJ 的织入编译器就是以这种方式织入切面的。
- 类加载期——切面在目标类加载到 JVM 时被织入。这种方式需要特殊的类加载器（ClassLoader），它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ 5 的 LTW（load-time weaving）就支持以这种方式织入切面。
- 运行期——切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP 容器会为目标对象动态地创建一个代理对象。Spring AOP 就是以这种方式织入切面的。

Spring通知是java写的 Spring在运行期通知对象 Spring只支持方法连接点

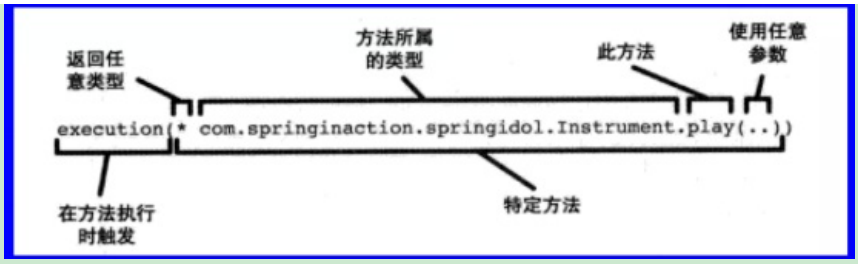
通过在代理类中包裹切面 Spring在运行期将切面织入到Spring管理的Bean中 如下图所示 代理类封装了目标类 并拦截了被通知的方法的调用 再将调用转发给真正的目标Bean 当拦截到方法调用时 在调用目标Bean方法之前 代理会执行切面逻辑



Aspectj切点表达式来定义Spring切面

AspectJ 指示器	描述
arg()	限制连接点匹配参数为指定类型的执行方法
@args()	限制连接点匹配参数由指定注解标注的执行方法
execution()	用于匹配是连接点的执行方法
this()	限制连接点匹配 AOP 代理的 Bean 引用为指定类型的类
target()	限制连接点匹配目标对象为指定类型的类
@target()	限制连接点匹配特定的执行对象，这些对象对应的类要具备指定类型的注解
within()	限制连接点匹配指定的类型
@within()	限制连接点匹配指定注解所标注的类型（当使用 Spring AOP 时，方法定义在由指定的注解所标注的类里）
@annotation	限制匹配带有指定注解连接点

使用Aspectj切面表达式来定定位



http://www.cnblogs.com/byleaf/p/4393626.html注意 &&或者|| 这种符号在xml元素中 会发生错误 因为解析器会把它当作新元素的开始 其实&&或者|| 这种符号 是可以用 and 或者or 来替代的

```
<aop:pointcut id="e" expression="execution(* com.springinaction.knights.Knight.embarkOnQuest(
..)
&& within(com.springinaction.knights.*)
"></aop:pointcut>
```

这是配置方式的一种

```
<aop:pointcut id="e" expression="execution(* com.springinaction.knights.Knight.embarkOnQuest(
..)
&& and bean(eddie)
"></aop:pointcut>
```

这也是一种方式 这是配置bean的指示器 限制只配置特定的bean

```
<aop:pointcut id="e" expression="execution(* com.springinaction.knights.Knight.embarkOnQuest(
..)
&& and  !bean(eddie)
"></aop:pointcut>
```

同理我们配置了操作符 指定除了eddie这个bean以外的都匹配

下面这个代码 是为了说明上面 && 可以用 and 替代写的

```
<aop:pointcut id="c" expression="execution(* com.Spring.Spring_in_action.chapters3_4.*.*(..)) and
!bean(testA_main)"/>
```

在XML中声明切面

AOP 配置元素	描述
<aop:advisor>	定义 AOP 通知器
<aop:after>	定义 AOP 后置通知（不管被通知的方法是否执行成功）
<aop:after-returning>	定义 AOP after-returning 通知
<aop:after-throwing>	定义 after-throwing 通知
<aop:around>	定义 AOP 环绕通知
<aop:aspect>	定义切面
<aop:aspectj-autoproxy>	启用 @AspectJ 注解驱动的切面
<aop:before>	定义 AOP 前置通知
<aop:config>	顶层的 AOP 配置元素。大多数的 <aop:*> 元素必须包含在 <aop:config> 元素内
<aop:declare-parents>	为被通知的对象引入额外的接口，并透明地实现
<aop:pointcut>	定义切点

这里本书介绍了一个特殊的技巧介绍 就是around 环绕通知 这个如果不调用proceed执行方法的话 就变成了阻拦方法的了 我们可以利用忽略调用proceed（）方法来阻止执行被通知的方法

这是配置参数的bean 将通知方法的参数传递给通知 关键在于切点定义与<aop:before>的arg-names属性

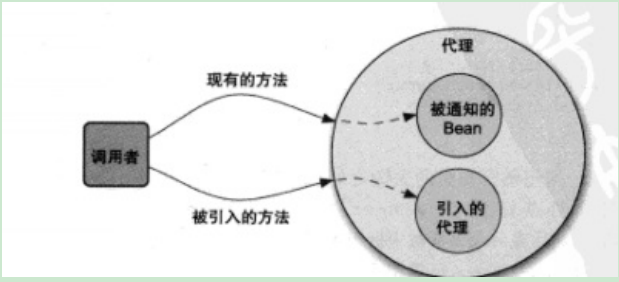
```
<bean id="magician"  class="com.springinaction.springidol.Magician"/>
<aop:config>
    <aop:aspect ref="magician">
        <aop:pointcut id="thinking" expression="execution(* com.springinaction.springidol.Thi
nker.thinkOfSomething(String))
        and args(thoughts)"
/>
```

```
        <aop:before pointcut-ref="thinking"  method="interceptThoughts" arg-names="thoughts"/
>

    </aop:aspect>
</aop:config>
```

这里需要注意 这个切点表达式匹配的方法只能是在接口中定义的（在spring3中是这样） 然后才能用将下面的前置通知获取到参数

关于通过切面引入新功能 过程如图



这是用@Aspect的注解方式声明一个类 下面是需要在xml中进行配置 应用为一个切面的声明 自动代理Bean 这些Bean中的具体内容是在@Aspect中定义的相匹配 这个本质实质上还是一个Spring类型的配置 如果需要Aspect的方式进行配置 那就必须抛弃Spring xml的配置 使用Aspect的方式 不能依赖Spring

```
@Aspect
public class Audience {
@Pointcut(
"execution(* com.springinaction.springidol.Performer.perform(..))")
public void performance() { //<co id="co_definePointcut"/>
}

@Before("performance()")
public void takeSeats() { //<co id="co_takeSeatsBefore"/>
System.out.println("The audience is taking their seats.");
}

@Before("performance()")
public void turnOffCellPhones() { //<co id="co_turnOffCellPhonesBefore"/>
System.out.println("The audience is turning off their cellphones");
}

@AfterReturning("performance()")
public void applaud() { //<co id="co_applaudAfter"/>
System.out.println("CLAP CLAP CLAP CLAP CLAP");
}

@AfterThrowing("performance()")
public void demandRefund() { //<co id="co_demandRefundAfterException"/>
System.out.println("Boo! We want our money back!");
}
```

```
}
```

```
<aop:aspectj-autoproxy />
```

其实这里书里还介绍了一个关于@around的环绕注解

```
@Around("performance()")
public void watchPerformance(ProceedingJoinPoint joinpoint) {
try {
    System.out.println("The audience is taking their seats.");
System.out.println("The audience is turning off their cellphones");

    long start = System.currentTimeMillis();
```

```
joinpoint.proceed();
    long end = System.currentTimeMillis();

System.out.println("CLAP CLAP CLAP CLAP CLAP");

System.out.println("The performance took " + (end - start)
        + " milliseconds.");
} catch(Throwable t) {
    System.out.println("Boo! We want our money back!");
}
}
```

就是这么个模样

这个是上面传递参数的配置用注解的方式写出来

```
<aop:aspectj-autoproxy />
```

扫描一下就好

```
@Aspect
public class Magician implements MindReader {
private String thoughts;

@Pointcut("execution(* com.springinaction.springidol." //<co id="co_parameterizedPointcut"/>
+ "Thinker.thinkOfSomething(String)) && args(thoughts)")
public void thinking(String thoughts) {
  }

@Before("thinking(thoughts)") //<co id="co_passInParameters"/>
public void interceptThoughts(String thoughts) {
    System.out.println("Intercepting volunteer's thoughts : " + thoughts);
    this.thoughts = thoughts;
}



public String getThoughts() {
return thoughts;
}
}
```

这里需要注意的是 在用注解的这种方式实现的时候　我们获取参数的过程在原来的xml中　是需要加上arg-names的　但是现在在这种注解实现的方式中是不需要的　@Aspectj能够依靠java语法来判断为通知所传递参数的细节 所以 这里并不需要与<aop:before>元素的arg-names属性对应的注解

这是与<aop:declare-parents/>对应的注解@DeclareParents

其中value对应type-matching

其中defaultImpl对应default-impl

最下面的注解所标注的static属性指定了被引入的接口 是等同于

<aop:declare-parents/>的implements-interface属性

```
@Aspect
public class ContestantIntroducer {

@DeclareParents( //<co id="co_declareParents"/>
value = "com.springinaction.springidol.Performer+",
defaultImpl = GraciousContestant.class)
public static Contestant contestant;

}
```

Spring AOP虽然能够满足许多应用的切面请求 但是和ApsectJ

相比，还是差了很多　功能比较弱　比如 构造器切点　但是spring　aop无法把通知应用于对象的创建（也就无法创建构造器切点）

所以说我们加入Aspecj也是不错的 一个好的切面一定包含其他类来完成它们的工作 加入Aspectj看样子是很有必要的

注入Aspecjt页面 这是一个Aspectj的例子 这个JudeAspect（评论员）是跟下面的

CriticismEngine（评论池）类来一起合作的 通过下的这个

CriticismEngine来发表评论

```
public aspect JudgeAspect {

public  JudgeAspect(){

    }
pointcut performance() :execution(* perform(..));
    after() returning() :performance(){
        System.out.println(criticismEngine);
}
private  CriticismEngine criticismEngine;
    public  void setCriticismEngine(CriticismEngine criticismEngine1){
this.criticismEngine=criticismEngine1;
}

}
```

我们为了这个CriticismEngine来实现了一个类 这个类就是放这个评论的评论池 到时候随机拿出一条评论来点评。

这个我们是需要通知spring来注入到AspectJ中 但是一定要记住Aspectj不需要Spring也是可以织入到我们的应用的

但是如果想使用spring的依赖注入为Aspectj切面注入协作者，那么就需要在spring中声明为一个bean

并且由于**spring无法创建JudgeAspect 所以我们需要调用一个静态的aspectof（）方法** 该方法返回一个单例 所以为了获得

切面的实例 我们必须使用factory-method来调用asepctof（）方法来代替调用JudgeAspect的构造器方法

简而言之 spring是无法创建一个JudgeAspect的实例的 这个是有Aspectj自己创建的 但是Spring可以通过aspectOf（）工厂方法来获得切面的引用 再来对这个引用进行依赖注入

```
public class CriticismEngineImpl implements  CriticismEngine{
public CriticismEngineImpl() {
    }
public String getCriticism(){
int i= (int) (Math.random()*criticismPool.length);
     return criticismPool[i];
}

private String[] criticismPool;
    public  void setCriticismPool(String[] criticismPool){
this.criticismPool=criticismPool;
}
}
```

```
<bean class="com.springinaction.springidol.JudgeAspect" factory-method="aspectOf">
    <property name="criticismEngine" ref="criticismEngine"></property>
</bean>
```

下面这位个就是关于上面critiismEngine的这个类的Bean配置 下面的配置里有评论的具体内容 之所以设置为bean是为了将这

CriticismEngine（评论池）类注入到spring中并注入给Aspectj的

```
<bean id="criticismEngine" class="com.springinaction.springidol.CriticismEngineImpl">
    <property name="criticismPool">
        <list>
            <value>I am  is 1</value>
            <value>I am  is 2</value>
            <value>I am  is 3</value>
            <value>I am  is 4</value>
        </list>
```

```
    </property>

</bean>
```