@Aspect

**public** **class** SystemArchitecture {

@Pointcut("within(com.xyz.someapp.web..\*)")

**public** **void** inWebLayer() {}

@Pointcut("within(com.xyz.someapp.service..\*)")

**public** **void** inServiceLayer() {}

@Pointcut("within(com.xyz.someapp.dao..\*)")

**public** **void** inDataAccessLayer() {}

@Pointcut("execution(\* com.xyz.someapp.service.\*.\*(..))")

**public** **void** businessService() {}

@Pointcut("execution(\* com.xyz.someapp.dao.\*.\*(..))")

**public** **void** dataAccessOperation() {}

<aop:config>

<aop:advisor

pointcut="com.xyz.someapp.SystemArchitecture.businessService()"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

一个官方的例子

格式execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)

**throws**-pattern?)

#### 7.2.3.5 Writing good pointcuts介绍了一些书写pointcut的方法,以及这些格式对性能的影响

#### 一个例子

@Aspect

public class ConcurrentOperationExecutor implements Ordered {

private static final int DEFAULT\_MAX\_RETRIES = 2;

private int maxRetries = DEFAULT\_MAX\_RETRIES;

private int order = 1;

public void setMaxRetries(int maxRetries) {

this.maxRetries = maxRetries;

}

public int getOrder() {

return this.order;

}

public void setOrder(int order) {

this.order = order;

}

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {

int numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

do {

numAttempts++;

try {

return pjp.proceed();

}

catch(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

}

while(numAttempts <= this.maxRetries);

throw lockFailureException;

}

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice. Notice that for the moment we're applying the retry logic to all businessService()s. We try to proceed, and if we fail with an PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.

The corresponding Spring configuration is:

<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"

class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

To refine the aspect so that it only retries idempotent operations, we might define an Idempotent annotation:

@Retention(RetentionPolicy.RUNTIME)

public @interface Idempotent {

// marker annotation

}

#### }

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +

"@annotation(com.xyz.myapp.service.Idempotent)")

public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {

...

}

#### The <aop:config> style of configuration makes heavy use of Spring's [auto-proxying](file:///E:\Java\Java%20EE\libs%20and%20other\libs\Spring\spring-framework-3.0.0.RELEASE\docs\spring-framework-reference\html\aop-api.html#aop-autoproxy) mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use of BeanNameAutoProxyCreator or suchlike. The recommended usage pattern is to use either just the <aop:config> style, or just the AutoProxyCreator style.

#### 不可以重复使用 否则可能会冲突

#### 因为在XML文档里&有特殊用途

When combining pointcut sub-expressions, '&&' is awkward within an XML document, and so the keywords 'and', 'or' and 'not' can be used in place of '&&', '||' and '!' respectively. For example, the previous pointcut may be better written as:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) and this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

#### Note that pointcuts defined in this way are referred to by their XML id and cannot be used as named pointcuts to form composite pointcuts. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

#### 硬编码方式

#### // create a factory that can generate a proxy for the given target object

#### AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

#### // add an aspect, the class must be an @AspectJ aspect

#### // you can call this as many times as you need with different aspects

#### factory.addAspect(SecurityManager.class);

#### // you can also add existing aspect instances, the type of the object supplied must be an @AspectJ aspect

#### factory.addAspect(usageTracker); // now get the proxy object...

#### MyInterfaceType proxy = factory.getProxy();

#### 作者说关于自身调用导致无法代理的 比如

#### public void Foo{

#### public void m1(){

#### this.m2();

#### }

#### public void m2(){

#### …

#### }

#### }

#### 在外界调用m1是会被捕获的,但是在m1内部调用m2是不会被捕获的

#### 最好的解决方式是重构你的代码,避免这种情况

#### 其他的解决方法是 让你的类知道自己被用于Spring AOP中 也就是暴露代理AopContext.getProxy(); 记得要exposeProxy="true"

#### 再来是方法注入(不是通过方法注入)

#### public abstract xxxType getThis();

#### 在bean配置中

#### <bean …>

#### <loopup-method …/>

#### </bean>

#### 当然也可以通过set方法注入代理对象

#### 不过这些方法都无法避免要让你的类知道自己被用于Spring AOP中(总而言之你的类跟普通的类就会有一点不像)