AOP support using @AspectJ annotation

Summary

@AspectJ is a method of defining aspects as general Java class annotated with Java 5 annotations. @AspectJ style was introduced by the AspectJ5 version. Spring 2.0 supports AspectJ5 annotation. The Spring AOP runtime supports @AspectJ annotation without depdendency on AspectJ complier or weaver.

Description

Enabling @AspectJ Support

To use @AspectJ, you need to add next codes to the Spring configuration.

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

Declaring an aspect

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the @Aspect annotation) will be automatically detected by Spring and used to configure Spring AOP.

```
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class AspectUsingAnnotation {
  ..
}
```

Declaring a pointcut

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. *AOP only supports method execution join points for Spring beans*, so you can think of a pointcut as matching the execution of methods on Spring beans.

An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named `'anyOldTransfer'` that will match the execution of any method named `'transfer'`:

```
@Aspect
public class AspectUsingAnnotation {
    ...
    @Pointcut("execution(public * egovframework.rte.fdl.aop.sample.*Sample.*(..))")
    public void targetMethod() {
        // a dummy method to refer the pointcut annotation value
    }
    ...
}
```

Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

- execution: for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
- within: limits matching to join points within certain types is an instance of the given type

- this: limits matching to join points where the bean reference is an instance of the given type.
- target: limits matching to join points where the target object is an instance of the given type.
- args: limits matching to join points where the arguments are instances of the given types.
- @target: limits matching to join points where the lass of the executing object has an annotation of the given type
- @args: limits matching to join points where the runtime type of the actual arguments passed have annotations of the given types
- @within: limits matching to join points within types that have the given annotation
- @annotation: limits matching to join points where the subject of the join point

Combining pointcut expressions

Pointcut expressions can be combined using '&&', '||' and '!'. It is also possible to refer to pointcut expressions by name.

```
@Pointcut("execution(public * *(..))")
    private void anyPublicOperation() {}

    @Pointcut("within(com.xyz.someapp.trading..*)")
    private void inTrading() {}

    @Pointcut("anyPublicOperation() && inTrading()")
    private void tradingOperation() {}
```

Example

Some examples of common pointcut expressions are given below

Pointcut	Selected join points
execution(public * *(..))	the execution of any public method
execution(* set*(..))	The execution of any method with a name beginning with ‘set’
execution(* set*(..))	The execution of any method with a name beginning with ‘set’
execution(* com.xyz.service.AccountService.*(..))	The execution of any method defined by the AccountService interface
execution(* com.xyz.service.*.*(..))	The execution of any method defined in the service pacakge
execution(* com.xyz.service..*.*(..))	The execution of any method defined in the service package or a sub-package
within(com.xyz.service.*)	Any join point within the service package
within(com.xyz.service..*)	Any join point within the service package
this(com.xyz.service.AccountService)	Any join point where the proxy implements the AccountService interface
target(com.xyz.service.AccountService)	Any join point where the target object implements the AccountService interface
args(java.io.Serializable)	Any join point which takes a singular parameter, and where the argument passed at runtime is Serializable
@target(org.springframework.transaction.annotation.Transactional)	Any join point where the target object has an @Transactional annotation
@within(org.springframework.transaction.annotation.Transactional)	Any join point where the executing method has an @Transactional annotation
@annotation(org.springframework.transaction.annotation.Transactional)	Any join point where the executing method has an @Transactional annotation
@args(com.xyz.security.Classified)	Any join point which takes a single parameter, and

	where the runtime type of the argument passed has the @Classified annotation
bean(accountRepository)	"accountRepository" bean
!bean(accountRepository)	Any bean excluding "accountRepository"
bean(*)	Any bean
bean(account*)	Any bean which the name starts with 'account'
bean(*Repository)	Any bean which the name ends with "Repository"
bean(accounting/*)	Any bean which the name starts with "accounting/"
bean(*dataSource) || bean(*DataSource)	Any bean which the name ends with "dataSource" or "DataSource"

Declaring advice

Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut, or a pointcut expression declared in place.

Before advice

Before advice is declared in an aspect using the @Before annotation.

Next is an example of using the Before advance. The beforeTargetMethod() method is executed before the pointcut declared as targetMethod().

```
@Aspect
public class AspectUsingAnnotation {
      ..
      @Before("targetMethod()")
      public void beforeTargetMethod(JoinPoint thisJoinPoint) {
            Class clazz = thisJoinPoint.getTarget().getClass();
            String className = thisJoinPoint.getTarget().getClass().getSimpleName();
            String methodName = thisJoinPoint.getSignature().getName();
            System.out.println("AspectUsingAnnotation.beforeTargetMethod executed.");
            System.out.println(className + "." + methodName + " executed.");
      }
}
```

After returning advice

After returning advice runs when a matched method execution returns normally. It is declared using the @AfterReturning annotation.

Next is an example of using the After returning advance. The afterReturningTargetMethod() advice is executed after the pointcut declared as targetMethod(). The result of executing targetMethod() is stored in retVal variable to be passed.

```
@Aspect
public class AspectUsingAnnotation {
      ..
      @AfterReturning(pointcut = "targetMethod()", returning = "retVal")
      public void afterReturningTargetMethod(JoinPoint thisJoinPoint,
                  Object retVal) {
            System.out.println("AspectUsingAnnotation.afterReturningTargetMethod executed." +
                                    " return value is [" + retVal + "]");

      }
}
```

After throwing advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the @AfterThrowing annotation.

Next is an example of using the After throwing advice. The afterThrowingTargetMethod() advice is executed after the exeption occurred in the pointcut declared as targetMethod(). The exeption occurred in targetMethod() pointcut is stored in the exception variable and passed. The example binds the passed exception to make the user to identify easily for returning.

```
@Aspect
public class AspectUsingAnnotation {
    ..
    @AfterThrowing(pointcut = "targetMethod()", throwing = "exception")
    public void afterThrowingTargetMethod(JoinPoint thisJoinPoint,
            Exception exception) throws Exception {
        System.out.println("AspectUsingAnnotation.afterThrowingTargetMethod executed.");
        System.out.println("error.", exception);

        throw new BizException("error.", exception);
    }
}
```

After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the `@After` annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

Next is an example of using the After(finally) advice. The afterTargetMethod() advice is used after the pointcut declared as targetMethod().

```
@Aspect
public class AspectUsingAnnotation {
    ..
    @After("targetMethod()")
    public void afterTargetMethod(JoinPoint thisJoinPoint) {
        System.out.println("AspectUsingAnnotation.afterTargetMethod executed.");
    }
}
```

Around advice

Around advice is executed before and after the method execution. Aroundd advice is declared using the @Around annotation.

Next is an example that uses the Around advice. The aroundTargetMethod() passes the ProceedingJoinPoint as a parameter and executes the pointcut through calling the proceed() method. This shows that the execution result value, retVal can be converted within the Around advice.

```
@Aspect
public class AspectUsingAnnotation {
    ..
    @Around("targetMethod()")
    public Object aroundTargetMethod(ProceedingJoinPoint thisJoinPoint)
            throws Throwable {
        System.out.println("AspectUsingAnnotation.aroundTargetMethod start.");
        long time1 = System.currentTimeMillis();
        Object retVal = thisJoinPoint.proceed();

        System.out.println("ProceedingJoinPoint executed. return value is [" + retVal + "]");

        retVal = retVal + "(modified)";
        System.out.println("return value modified to [" + retVal + "]");

        long time2 = System.currentTimeMillis();
        System.out.println("AspectUsingAnnotation.aroundTargetMethod end. Time(" + (time2 -
time1) + ")");
        return retVal;
    }
}
```

Implementing aspects

Use the test code to confirm normal activities of defined aspects. The AnnotationAspectTest class tests two cases: normal execution without exceptions in executing the method; and with exceptions.

Normal Execution

The testAnnotationAspect() function shows the example which the method is executed normally. The before, after returning, after finally, around advice is applied to the someMethod() of the AnnotationAdviceSample class within the egovframework.rte.fdl.aop.sample.

```
public class AnnotationAspectTest {
    @Resource(name = "annotationAdviceSample")
    AnnotationAdviceSample annotationAdviceSample;

    @Test
    public void testAnnotationAspect() throws Exception {
        SampleVO vo = new SampleVO();
        ..
        String resultStr = annotationAdviceSample.someMethod(vo);

        assertEquals("someMethod executed.(modified)", resultStr);
    }
}
```

The result log from executing the test code is as following:

```
AspectUsingAnnotation.beforeTargetMethod executed.
AspectUsingAnnotation.aroundTargetMethod start.
ProceedingJoinPoint executed. return value is [someMethod executed.]
return value modified to [someMethod executed.(modified)]
AspectUsingAnnotation.aroundTargetMethod end. Time(78)
AspectUsingAnnotation.afterTargetMethod executed.
AspectUsingAnnotation.afterReturningTargetMethod executed. return value is [someMethod
executed.(modified)]
```

The order of applying the advice in the console log output is as following:

- @Before
- @Around (before executing the method)
- Method
- @Around (after executing the method)
- @After(finally)
- @AfterReturning

Note that @Around advice can change the method return value but the after returning advice can only refer to and cannot change the return value.

Exceptions

testAnnotationAspectWithException() function shows an example of errors in the method.Before, after throwing, after finally, around advices are applied to someMethod() of the Annotation AdviceSample class within the egovframework.rte.fdl.aop.sample.

```
public class AnnotationAspectTest {
    @Resource(name = "annotationAdviceSample")
    AnnotationAdviceSample annotationAdviceSample;

    @Test
    public void testAnnotationAspectWithException() throws Exception {
        SampleVO vo = new SampleVO();
        // set the flag to generate exceptions
         vo.setForceException(true);
        ..
        try {
            // when the forceException flag of vo is true, it forces - / by zero situation
            resultStr = annotationAdviceSample.someMethod(vo);

            fail("this line cannot be executed because the exception is forced.");
        } catch (Exception e) {
            ..
        }
    }
}
```

The result log from executing the test code is as following:

```
AspectUsingAnnotation.beforeTargetMethod executed.
AspectUsingAnnotation.aroundTargetMethod start.
AspectUsingAnnotation.afterTargetMethod executed.
AspectUsingAnnotation.afterThrowingTargetMethod executed.
Error.
java.lang.ArithmeticException: / by zero
...
```

The order of applying the advice in the console log output is as following:

- @Before
- @Around (before executing the method)
- Method (ArithmeticException is generated)
- @After(finally)
- @AfterThrowing

The advice declared as after is executed even if the exception occurs. The After Throwing advice can reset the error message, generate new exceptions to pass.

Reference

- Spring 2.5 Reference Documentation