@AspectJ 어노테이션을 이용한 AOP 지원

개요

@AspectJ는 Java 5 어노테이션을 사용한 일반 Java 클래스로 관점(Aspect)를 정의하는 방식이다. @AspectJ 방식은 AspectJ 5 버전에서 소개되었으며, Spring은 2.0 버전부터 AspectJ 5 어노테이션을 지원한다. Spring AOP 실행환경은 AspectJ 컴파일러나 직조기(Weaver)에 대한 의존성이 없이 @AspectJ 어노테이션을 지원한다.

설명

@AspectJ 설정하기

@AspectJ를 사용하기 위해서 다음 코드를 Spring 설정에 추가한다.

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

관점(Aspect) 정의하기

클래스에 @Aspect 어노테이션을 추가하여 Aspect를 생성한다. @Aspect 설정이 되어 있는 경우 Spring은 자동적으로 @Aspect 어노테이션을 포함한 클래스를 검색하여 Spring AOP 설정에 반영한다.

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

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

포인트컷(Pointcut) 정의하기

포인트컷은 결합점(Join points)을 지정하여 충고(Advice)가 언제 실행될지를 지정하는데 사용된다. Spring AOP는 Spring 빈에 대한 메소드 실행 결합점만을 지원하므로, Spring에서 포인트컷은 빈의 메소드 실행점을 지정하는 것으로 생각할 수 있다.

다음 예제는 egovframework.rte.fdl.aop.sample 패키지 하위의 Sample 명으로 끝나는 클래스의 모든 메소드 수행과 일치할 'targetMethod' 라는 이름의 pointcut을 정의한다.

```
@Aspect
public class AspectUsingAnnotation {
    ...
    @Pointcut("execution(public * egovframework.rte.fdl.aop.sample.*Sample.*(..))")
    public void targetMethod() {
        // pointcut annotation 값을 참조하기 위한 dummy method
    }
    ...
}
```

포인트컷 지정자(Designators)

Spring에서 포인트컷 표현식에 사용될 수 있는 지정자는 다음과 같다. 포인트컷은 모두 public 메소드를 대상으로 한다.

- execution: 메소드 실행 결합점(join points)과 일치시키는데 사용된다.
- within: 특정 타입에 속하는 결합점을 정의한다.
- this: 빈 참조가 주어진 타입의 인스턴스를 갖는 결합점을 정의한다.

- target: 대상 객체가 주어진 타입을 갖는 결합점을 정의한다.
- args: 인자가 주어진 타입의 인스턴스인 결합점을 정의한다.
- @target: 수행중인 객체의 클래스가 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
- @args: 전달된 인자의 런타입 타입이 주어진 타입의 어노테이션을 갖는 결합점을 정의한다.
- @within: 주어진 어노테이션을 갖는 타입 내 결합점을 정의한다.
- @annotation: 결합점의 대상 객체가 주어진 어노테이션을 갖는 결합점을 정의한다.

포인트컷 표현식 조합하기

포인트컷 표현식은 '&&', '||' 그리고 '!' 를 사용하여 조합할 수 있다.

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

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

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

포인트컷 정의 예제

Spring AOP에서 자주 사용되는 포인트컷 표현식의 예를 살펴본다.

Pointcut	**선택된 Joinpoints**
execution(public * *(..))	public 메소드 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* set*(..))	이름이 set으로 시작하는 모든 메소드명 실행
execution(* com.xyz.service.AccountService.*(..))	AccountService 인터페이스의 모든 메소드 실행
execution(* com.xyz.service.*.*(..))	service 패키지의 모든 메소드 실행
execution(* com.xyz.service..*.*(..))	service 패키지와 하위 패키지의 모든 메소드 실행
within(com.xyz.service.*)	service 패키지 내의 모든 결합점
within(com.xyz.service..*)	service 패키지 및 하위 패키지의 모든 결합점
this(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 프록시 개체의 모든 결합점
target(com.xyz.service.AccountService)	AccountService 인터페이스를 구현하는 대상 객체의 모든 결합점
args(java.io.Serializable)	하나의 파라미터를 갖고 전달된 인자가 Serializable인 모든 결합점
@target(org.springframework.transaction.annotation.Transactional)	대상 객체가 @Transactional 어노테이션을 갖는 모든 결합점
@within(org.springframework.transaction.annotation.Transactional)	대상 객체의 선언 타입이 @Transactional 어노테이션을 갖는 모든 결합점
@annotation(org.springframework.transaction.annotation.Transactional)	실행 메소드가 @Transactional 어노테이션을 갖는 모든 결합점
@args(com.xyz.security.Classified)	단일 파라미터를 받고, 전달된 인자 타입이 @Classified 어노테이션을 갖는 모든 결합점
bean(accountRepository)	"accountRepository" 빈
!bean(accountRepository)	"accountRepository" 빈을 제외한 모든 빈
bean(*)	모든 빈
bean(account*)	이름이 'account'로 시작되는 모든 빈
bean(*Repository)	이름이 "Repository"로 끝나는 모든 빈
bean(accounting/*)	이름이 "accounting/"로 시작하는 모든 빈
bean(*dataSource) || bean(*DataSource)	이름이 "dataSource" 나 "DataSource" 으로 끝나는 모든 빈

충고(Advice) 정의하기

충고(Advice)는 관점(Aspect)의 실제 구현체로 포인트컷 표현식과 일치하는 결합점에 삽입되어 동작할 수 있는 코드이다. 충고는 결합점과 결합하여 동작하는 시점에 따라 before advice, after advice, around advice 타입으로 구분된다.

Before advice

Before advice는 @Before 어노테이션을 사용한다.

다음은 Before 충고를 사용하는 예제이다. Before 충고인 beforeTargetMethod() 메소드는 targetMethod()로 정의된 포인트컷 전에 수행된다.

```
@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 returing 충고는 정상적으로 메소드가 실행될 때 수행된다. After returning 충고는 @AfterReturing 어노테이션을 사용한다.

다음은 After returning 충고를 사용하는 예제이다. afterReturningTargetMethod() 충고는 targetMethod()로 정의된 포인트컷 후에 수행된다. targetMethod() 포인트컷의 실행 결과는 retVal 변수에 저장되어 전달된다.

```
@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 충고는 메소드가 수행 중 예외사항을 반환하고 종료하는 경우 수행된다. After throwing 충고는 @AfterThrowing 어노테이션을 사용한다.

다음은 After throwing 충고를 사용하는 예제이다. afterThrowingTargetMethod() 충고는 targetMethod()로 정의된 포인트컷에서 예외가 발생한 후에 수행된다. targetMethod() 포인트컷에서 발생된 예외는 exception 변수에 저장되어 전달된다. 예제에서는 전달 받은 예외를 한번 더 감싸서 사용자가 쉽게 알아 볼 수 있도록 메시지를 설정하여 반환한다.

```
@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("에러가 발생했습니다.", exception);

        throw new BizException("에러가 발생했습니다.", exception);
```

```
    }
}
```

After (finally) advice

After (finally) 충고는 메소드 수행 후 무조건 수행된다. After (finally) 충고는 @After 어노테이션을 사용한다. After 충고는 정상 종료와 예외 발생 경우를 모두 처리해야 하는 경우에 사용된다. 리소스 해제와 같은 작업이 해당된다.

다음은 After (finally) 충고를 사용하는 예제이다. afterTargetMethod() 충고는 targetMethod()로 정의된 포인트컷 이후에 수행된다.

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

Around advice

Around 충고는 메소드 수행 전후에 수행된다. Around 충고는 @Around 어노테이션을 사용한다.

다음은 Around 충고를 사용하는 예제이다. aroundTargetMethod() 충고는 파라미터로 ProceedingJoinPoint을 전달하며 proceed() 메소드 호출을 통해 대상 포인트컷을 실행한다. 포인트컷 수행 결과값인 retVal을 Around 충고 내에서 변환하여 반환할 수 있음을 보여준다.

```
@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;
    }
}
```

관점(Aspect) 실행하기

앞서 정의한 관점(Aspect)가 정상적으로 동작하는지 확인하기 위해 테스트 코드를 이용해 확인해 본다. AnnotationAspectTest 클래스는 대상 메소드 수행시 예외없이 정상 실행하는 경우와 예외 발생의 경우를 구분해서 테스트 한다.

정상 실행의 경우

testAnnotationAspect() 함수는 대상 메소드가 정상 수행되는 사례를 보여준다. egovframework.rte.fdl.aop.sample 패키지에 속하는 AnnotationAdviceSample 클래스의 someMethod() 메소드는 before, after returning, after finally, around 충고(Advice)가 적용된다.

```
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);
    }
}
```

테스트 코드를 수행한 결과 로그는 다음과 같다.

```
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)]
```

콘솔 로그 출력을 보면 충고(Advice)가 적용되는 순서는 다음과 같다.

- @Before
- @Around (대상 메소드 수행 전)
- 대상 메소드
- @Around (대상 메소드 수행 후)
- @After(finally)
- @AfterReturning

주의할 점은 @Around 충고는 대상 메소드의 반환 값(return value)를 변경 가능하지만, After returning 충고는 반환 값을 참조 가능하지만 변경할 수 없다.

예외 발생의 경우

testAnnotationAspectWithException() 함수는 대상 메소드에 오류가 발생한 사례를 보여준다. egovframework.rte.fdl.aop.sample 패키지에 속하는 AnnotationAdviceSample 클래스의 someMethod() 메소드는 before, after throwing, after finally, around 충고(Advice)가 적용된다.

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

    @Test
    public void testAnnotationAspectWithException() throws Exception {
        SampleVO vo = new SampleVO();
        // exception 을 발생시키도록 플래그 설정
         vo.setForceException(true);
        ..
        try {
            // vo 의 forceException 플래그가 true 이면 - / by zero 상황을 강제로 처리함
            resultStr = annotationAdviceSample.someMethod(vo);

            fail("exception 을 강제로 발생시켜 이 라인이 수행될 수 없습니다.");
        } catch (Exception e) {
            ..
        }
    }
}
```

테스트 코드를 수행한 결과 로그는 다음과 같다.

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

콘솔 로그 출력을 보면 충고(Advice)가 적용되는 순서는 다음과 같다.

- @Before
- @Around (대상 메소드 수행 전)
- 대상 메소드 (ArithmeticException 예외가 발생한다)
- @After(finally)
- @AfterThrowing

예외가 발생하더라도 after 로 정의한 충고(Advice)는 수행되는 것을 확인할 수 있다. After Throwing 충고(Advice)는 에러 메시지를 재설정하고 새로운 예외를 생성하여 전달할 수 있다.

참고자료

- Spring 2.5 Reference Documentation [http://static.springframework.org/spring/docs/2.5.x/reference/aop.html]

egovframework/rte/fdl/aop/aspectj.txt · 마지막 수정: 2015/04/14 11:24 (외부 편집기)