Use Aspect-Oriented Programming with Annotations

PROBLEM

You want to use aspect-oriented programming with Spring and annotations.

SOLUTION

You define an aspect by decorating a Java class with the @Aspect annotation. Each of the methods in a class can become an advice with another annotation. You can use five types of advice annotations: @Before, @After, @AfterReturning, @AfterThrowing, and @Around.

To enable annotation support in the Spring IoC container, you have to add @EnableAspectJAutoProxy to one of your configuration classes. To apply AOP, Spring creates proxies, and by default it creates JDK dynamic proxies, which are interface-based. For cases in which interfaces are not available or not used in an application’s design, it’s possible to create proxies by relying on CGLIB. To enable CGLIB, you need to set the attribute proxyTargetClass=true on the @EnableAspectJAutoProxy annotation.

HOW IT WORKS

To support aspect-oriented programming with annotations, Spring uses the same annotations as AspectJ, using a library supplied by AspectJ for pointcut parsing and matching. However, the AOP runtime is still pure Spring AOP, and there is no dependency on the AspectJ compiler or weaver.

To illustrate the enablement of aspect-oriented programming with annotations, you’ll use the following calculator interfaces to define a set of sample POJOs :

**package** com.apress.springrecipes.calculator;  
  
**public interface** ArithmeticCalculator {  
  
    **public** double add(double a, double b);  
    **public** double sub(double a, double b);  
    **public** double mul(double a, double b);  
    **public** double div(double a, double b);  
}  
**package** com.apress.springrecipes.calculator;  
  
**public interface** UnitCalculator {  
  
    **public** double kilogramToPound(double kilogram);  
    **public** double kilometerToMile(double kilometer);  
}

Next, let’s create POJO classes for each interface with println statements to know when each method is executed:

**package** com.apress.springrecipes.calculator;  
  
**import** org.springframework.stereotype.Component;  
  
@Component("arithmeticCalculator")  
**public class** ArithmeticCalculatorImpl **implements** ArithmeticCalculator {  
  
    **public** double add(double a, double b) {  
        double result = a + b;  
        System.out.println(a + " + " + b + " = " + result);  
        **return** result;  
    }  
  
    **public** double sub(double a, double b) {  
        double result = a - b;  
        System.out.println(a + " - " + b + " = " + result);  
        **return** result;  
    }  
    **public** double mul(double a, double b) {  
        double result = a \* b;  
        System.out.println(a + " \* " + b + " = " + result);  
        **return** result;  
    }  
  
    **public** double div(double a, double b) {  
        **if** (b == 0) {  
            **throw new** IllegalArgumentException("Division by zero");  
        }  
        double result = a / b;  
        System.out.println(a + " / " + b + " = " + result);  
        **return** result;  
    }  
}  
**package** com.apress.springrecipes.calculator;  
  
**import** org.springframework.stereotype.Component;  
  
@Component("unitCalculator")  
**public class** UnitCalculatorImpl **implements** UnitCalculator {  
  
    **public** double kilogramToPound(double kilogram) {  
        double pound = kilogram \* 2.2;  
        System.out.println(kilogram + " kilogram = " + pound + " pound");  
        **return** pound;  
    }  
  
    **public** double kilometerToMile(double kilometer) {  
        double mile = kilometer \* 0.62;  
        System.out.println(kilometer + " kilometer = " + mile + " mile");  
        **return** mile;  
    }  
}

Note that each POJO implementation is decorated with the @Component annotation to create bean instances.

*Declare Aspects, Advices, and Pointcuts*

An aspect is a Java class that modularizes a set of concerns (e.g., logging or transaction management) that cuts across multiple types and objects. Java classes that modularize such concerns are decorated with the @Aspect annotation. In AOP terminology, aspects are also complemented by advices, which in themselves have pointcuts. An advice is a simple Java method with one of the advice annotations. AspectJ supports five types of advice annotations : @Before, @After, @AfterReturning, @AfterThrowing, and @Around. A pointcut is an expression that looks for types and objects on which to apply the aspect’s advices.

*Aspect with @Before Advice*

To create a before advice to handle crosscutting concerns before particular program execution points, you use the @Before annotation and include the pointcut expression as the annotation value.

**package** com.apress.springrecipes.calculator;  
  
**import** org.apache.commons.logging.Log;  
**import** org.apache.commons.logging.LogFactory;  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Before;  
**import** org.springframework.stereotype.Component;  
  
@Aspect  
@Component  
**public class** CalculatorLoggingAspect {  
  
    **private** Log log = LogFactory.getLog(**this**.getClass());  
  
    @Before("execution(\* ArithmeticCalculator.add(..))")  
    **public** void logBefore() {  
        log.info("The method add() begins");  
    }  
}

This pointcut expression matches the add() method execution of the ArithmeticCalculator interface. The preceding wildcard in this expression matches any modifier (public, protected, and private) and any return type. The two dots in the argument list match any number of arguments.

For the previous aspect to work (i.e., output its message), you need to set up logging. Specifically, create a logback.xml file with configuration properties like the following.

**<?xml** version="1.0" encoding="UTF-8"**?>**   
**<configuration>**   
  
    **<appender** name="STDOUT" class="ch.qos.logback.core.ConsoleAppender"**>**   
        **<layout** class="ch.qos.logback.classic.PatternLayout"**>**   
            **<Pattern>**%d [%15.15t] %-5p %30.30c - %m%n**</Pattern>**   
        **</layout>**   
    **</appender>**   
  
    **<root** level="INFO"**>**   
        **<appender-ref** ref="STDOUT" **/>**   
    **</root>**   
  
**</configuration>**

**Note**

The @Aspect annotation is not sufficient for autodetection in the classpath. Therefore, you need to add a separate @Component annotation for the POJO to be detected.

Next, you create a Spring configuration to scan all POJOs, including the POJO calculator implementation and aspect and including the @EnableAspectJAutoProxy annotation.

@Configuration  
@EnableAspectJAutoProxy  
@ComponentScan  
**public class** CalculatorConfiguration {  
}

As the last step, you can test the aspect with the following Main class:

**package** com.apress.springrecipes.calculator;  
  
**import** org.springframework.context.ApplicationContext;  
**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
**public class** Main {  
  
    **public** static void main(String[] args) {  
  
        ApplicationContext context =  
            **new** AnnotationConfigApplicationContext(CalculatorConfiguration.class);  
  
        ArithmeticCalculator arithmeticCalculator =  
            context.getBean("arithmeticCalculator", ArithmeticCalculator.class);  
        arithmeticCalculator.add(1, 2);  
        arithmeticCalculator.sub(4, 3);  
        arithmeticCalculator.mul(2, 3);  
        arithmeticCalculator.div(4, 2);  
  
        UnitCalculator unitCalculator = context.getBean("unitCalculator", UnitCalculator.class);  
        unitCalculator.kilogramToPound(10);  
        unitCalculator.kilometerToMile(5);  
    }  
}

The execution points matched by a pointcut are called *join points*. In this term, a pointcut is an expression to match a set of join points, while an advice is the action to take at a particular join point.

For your advice to access the detail of the current join point, you can declare an argument of type JoinPoint in your advice method. Then, you can get access to join point details such as the method name and argument values. Now, you can expand your pointcut to match all methods by changing the class name and method name to wildcards.

**package** com.apress.springrecipes.calculator;  
...  
**import** java.util.Arrays;  
  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Before;  
  
@Aspect  
@Component  
**public class** CalculatorLoggingAspect {  
    ...  
    @Before("execution(\* \*.\*(..))")  
    **public** void logBefore(JoinPoint joinPoint) {  
        log.info("The method " + joinPoint.getSignature().getName()  
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));  
    }  
}

*Use an Aspect with @After Advice*

An after advice is executed after a join point finishes and is represented by a method annotated with @After, whenever it returns a result or throws an exception. The following after advice logs the calculator method ending:

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.After;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @After("execution(\* \*.\*(..))")  
    **public** void logAfter(JoinPoint joinPoint) {  
        log.info("The method " + joinPoint.getSignature().getName()  
            + "() ends");  
    }  
}

*Use an Aspect with @AfterReturning Advice*

An after advice is executed regardless of whether a join point returns normally or throws an exception . If you would like to perform logging only when a join point returns, you should replace the after advice with an after returning advice.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.AfterReturning;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @AfterReturning("execution(\* \*.\*(..))")  
    **public** void logAfterReturning(JoinPoint joinPoint) {  
        log.info("The method {}() ends with {}", joinPoint.getSignature().getName(), result);  
    }  
}

In an after returning advice, you can get access to the return value of a join point by adding a returning attribute to the @AfterReturning annotation. The value of this attribute should be the argument name of this advice method for the return value to pass in. Then, you have to add an argument to the advice method signature with this name. At runtime, Spring AOP will pass in the return value through this argument. Also note that the original pointcut expression needs to be presented in the pointcut attribute instead.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.AfterReturning;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @AfterReturning(  
        pointcut = "execution(\* \*.\*(..))",  
        returning = "result")  
    **public** void logAfterReturning(JoinPoint joinPoint, Object result) {  
        log.info("The method " + joinPoint.getSignature().getName()  
            + "() ends with " + result);  
    }  
}

*Use an Aspect with @AfterThrowing Advice*

An after throwing advice is executed only when an exception is thrown by a join point.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.AfterThrowing;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @AfterThrowing("execution(\* \*.\*(..))")  
    **public** void logAfterThrowing(JoinPoint joinPoint) {  
        log.error("An exception has been thrown in {}()", joinPoint.getSignature().getName());  
    }  
}

Similarly, the exception thrown by the join point can be accessed by adding a throwing attribute to the @AfterThrowing annotation . The type Throwable is the superclass of all errors and exceptions in the Java language. So, the following advice will catch any of the errors and exceptions thrown by the join points:

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.AfterThrowing;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @AfterThrowing(  
        pointcut = "execution(\* \*.\*(..))",  
        throwing = "e")  
    **public** void logAfterThrowing(JoinPoint joinPoint, Throwable e) {  
        log.error("An exception {} has been thrown in {}()", e, joinPoint.getSignature().getName());  
    }  
}

However, if you are interested in one particular type of exception only, you can declare it as the argument type of the exception. Then your advice will be executed only when exceptions of compatible types (i.e., this type and its subtypes) are thrown.

**package** com.apress.springrecipes.calculator;  
...  
**import** java.util.Arrays;  
  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.AfterThrowing;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @AfterThrowing(  
        pointcut = "execution(\* \*.\*(..))",  
        throwing = "e")  
    **public** void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {  
        log.error("Illegal argument {} in {}()", Arrays.toString(joinPoint.getArgs()), joinPoint.getSignature().getName());  
    }  
}

*Use an Aspect with @Around Advice*

The last type of advice is an around advice . It is the most powerful of all the advice types. It gains full control of a join point, so you can combine all the actions of the preceding advices into one single advice. You can even control when, and whether, to proceed with the original join point execution.

The following around advice is the combination of the before, after returning, and after throwing advices you created earlier. Note that for an around advice, the argument type of the join point must be ProceedingJoinPoint. It’s a subinterface of JoinPoint that allows you to control when to proceed with the original join point.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.ProceedingJoinPoint;  
**import** org.aspectj.lang.annotation.Around;  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
**import** org.springframework.stereotype.Component;  
  
**import** java.util.Arrays;  
  
@Aspect  
@Component  
**public class** CalculatorLoggingAspect {  
  
    **private** Logger log = LoggerFactory.getLogger(**this**.getClass());  
  
    @Around("execution(\* \*.\*(..))")  
    **public** Object logAround(ProceedingJoinPoint joinPoint) **throws** Throwable {  
  
        log.info("The method {}() begins with {}", joinPoint.getSignature().getName(), Arrays.toString(joinPoint.getArgs()));  
        **try** {  
            Object result = joinPoint.proceed();  
            log.info("The method {}() ends with ", joinPoint.getSignature().getName(), result);  
            **return** result;  
        } **catch** (IllegalArgumentException e) {  
            log.error("Illegal argument {} in {}()", Arrays.toString(joinPoint.getArgs()) , joinPoint.getSignature().getName());  
            **throw** e;  
        }  
    }  
}

The around advice type is powerful and flexible in that you can even alter the original argument values and change the final return value. You must use this type of advice with great care, as the call to proceed with the original join point may easily be forgotten.

**Tip**

A common rule for choosing an advice type is to use the least powerful one that can satisfy your requirements.

2-14. Access the Join Point Information

PROBLEM

In AOP, an advice is applied to different program execution points that are called *join points*. For an advice to take the correct action, it often requires detailed information about join points.

SOLUTION

An advice can access the current join point information by declaring an argument of type org.aspectj.lang.JoinPoint in the advice method signature.

HOW IT WORKS

For example, you can access the join point information through the following advice. The information includes the join point kind (only method execution in Spring AOP), the method signature (declaring type and method name), and the argument values, as well as the target object and proxy object.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Before;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
**import** org.springframework.stereotype.Component;  
  
**import** java.util.Arrays;  
  
@Aspect  
@Component  
**public class** CalculatorLoggingAspect {  
  
    **private** Logger log = LoggerFactory.getLogger(**this**.getClass());  
  
    @Before("execution(\* \*.\*(..))")  
    **public** void logJoinPoint(JoinPoint joinPoint) {  
  
        log.info("Join point kind : {}", joinPoint.getKind());  
        log.info("Signature declaring type : {}", joinPoint.getSignature().getDeclaringTypeName());  
        log.info("Signature name : {}", joinPoint.getSignature().getName());  
        log.info("Arguments : {}", Arrays.toString(joinPoint.getArgs()));  
        log.info("Target class : {}", joinPoint.getTarget().getClass().getName());  
        log.info("This class : {}", joinPoint.getThis().getClass().getName());  
    }  
}

The original bean that was wrapped by a proxy is called the *target object* , while the proxy object is the this object. They can be accessed by the join point’s getTarget() and getThis() methods. From the following outputs, you can see that the classes of these two objects are not the same:

Join point kind : method-execution  
Signature declaring type : com.apress.springrecipes.calculator.ArithmeticCalculator  
Signature name : add  
Arguments : [1.0, 2.0]  
Target class : com.apress.springrecipes.calculator.ArithmeticCalculatorImpl  
This class : com.sun.proxy.$Proxy6

2-15. Specify Aspect Precedence with the @Order Annotation

PROBLEM

When there’s more than one aspect applied to the same join point, the precedence of the aspects is undefined unless you have explicitly specified it.

SOLUTION

The precedence of aspects can be specified either by implementing the Ordered interface or by using the @Order annotation.

HOW IT WORKS

Suppose you have written another aspect to validate the calculator arguments. There’s only one before advice in this aspect.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.JoinPoint;  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Before;  
  
**import** org.springframework.stereotype.Component;  
  
@Aspect  
@Component  
**public class** CalculatorValidationAspect {  
  
    @Before("execution(\* \*.\*(double, double))")  
    **public** void validateBefore(JoinPoint joinPoint) {  
        **for** (Object arg : joinPoint.getArgs()) {  
            validate((Double) arg);  
        }  
    }  
  
    **private** void validate(double a) {  
        **if** (a < 0) {  
            **throw new** IllegalArgumentException("Positive numbers only");  
        }  
    }  
}

If you apply this aspect and the previous, you can’t guarantee which one is applied first. To guarantee that one aspect is applied before another, you need to specify precedence. To specify precedence, you have to make both aspects implement the Ordered interface or use the @Order annotation.

If you decide to implement the Ordered interface , the lower value returned by the getOrder method represents higher priority. So, if you prefer the validation aspect to be applied first, it should return a value lower than the logging aspect.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.springframework.core.Ordered;  
  
@Aspect  
@Component  
**public class** CalculatorValidationAspect **implements** Ordered {  
    ...  
    **public** int getOrder() {  
        **return** 0;  
    }  
}  
**package** com.apress.springrecipes.calculator;  
...  
**import** org.springframework.core.Ordered;  
  
@Aspect  
@Component  
**public class** CalculatorLoggingAspect **implements** Ordered {  
    ...  
    **public** int getOrder() {  
        **return** 1;  
    }  
}

Another way to specify precedence is through the @Order annotation. The order number should be presented in the annotation value.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.springframework.core.annotation.Order;  
  
@Aspect  
@Component  
@Order(0)  
**public class** CalculatorValidationAspect { **...** }  
**package** com.apress.springrecipes.calculator;  
...  
**import** org.springframework.core.annotation.Order;  
   
@Aspect  
@Component  
@Order(1)  
**public class** CalculatorLoggingAspect { **...** }

2-16. Reuse Aspect Pointcut Definitions

PROBLEM

When writing aspects, you can directly embed a pointcut expression in an advice annotation. You want to use the same pointcut expression in multiple advices without embedding it multiple times.

SOLUTION

You can use the @Pointcut annotation to define a pointcut independently to be reused in multiple advices.

HOW IT WORKS

In an aspect, a pointcut can be declared as a simple method with the @Pointcut annotation. The method body of a pointcut is usually empty because it is unreasonable to mix a pointcut definition with application logic. The access modifier of a pointcut method controls the visibility of this pointcut as well. Other advices can refer to this pointcut by the method name.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
@Component  
**public class** CalculatorLoggingAspect {  
    ...  
    @Pointcut("execution(\* \*.\*(..))")  
    **private** void loggingOperation() {}  
  
    @Before("loggingOperation()")  
    **public** void logBefore(JoinPoint joinPoint) {  
        ...  
    }  
  
    @AfterReturning(  
        pointcut = "loggingOperation()",  
        returning = "result")  
    **public** void logAfterReturning(JoinPoint joinPoint, Object result) {  
        ...  
    }  
  
    @AfterThrowing(  
        pointcut = "loggingOperation()",  
        throwing = "e")  
    **public** void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {  
        ...  
    }  
  
    @Around("loggingOperation()")  
    **public** Object logAround(ProceedingJoinPoint joinPoint) **throws** Throwable {  
        ...  
    }  
}

Usually, if your pointcuts are shared between multiple aspects, it is better to centralize them in a common class. In this case, they must be declared as public.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
**public class** CalculatorPointcuts {  
  
    @Pointcut("execution(\* \*.\*(..))")  
    **public** void loggingOperation() {}  
}

When you refer to this pointcut, you have to include the class name as well. If the class is not located in the same package as the aspect, you have to include the package name also.

**package** com.apress.springrecipes.calculator;  
...  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @Before("CalculatorPointcuts.loggingOperation()")  
    **public** void logBefore(JoinPoint joinPoint) {  
        ...  
    }  
  
    @AfterReturning(  
        pointcut = "CalculatorPointcuts.loggingOperation()",  
        returning = "result")  
    **public** void logAfterReturning(JoinPoint joinPoint, Object result) {  
        ...  
    }  
  
    @AfterThrowing(  
        pointcut = "CalculatorPointcuts.loggingOperation()",  
        throwing = "e")  
    **public** void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {  
        ...  
    }  
  
    @Around("CalculatorPointcuts.loggingOperation()")  
    **public** Object logAround(ProceedingJoinPoint joinPoint) **throws** Throwable {  
        ...  
    }  
}

2-17. Write AspectJ Pointcut Expressions

PROBLEM

Crosscutting concerns can happen at different program execution points called *join points*. Because of the variety of join points, you need a powerful expression language to help match them.

SOLUTION

The AspectJ pointcut language is a powerful expression language that can match various kinds of join points. However, Spring AOP only supports method execution join points for beans declared in its IoC container. For this reason, only those pointcut expressions supported by Spring AOP are presented in this recipe. For a full description of the AspectJ pointcut language, please refer to the AspectJ programming guide available on AspectJ’s web site ([www.eclipse.org/aspectj/](http://www.eclipse.org/aspectj/)). Spring AOP makes use of the AspectJ pointcut language for its pointcut definition and interprets the pointcut expressions at runtime by using a library provided by AspectJ. When writing AspectJ pointcut expressions for Spring AOP, you must keep in mind that Spring AOP only supports method execution join points for the beans in its IoC container. If you use a pointcut expression out of this scope, an IllegalArgumentException is thrown.

HOW IT WORKS

Lets explore the, by Spring, supported patterns for writing pointcut expression. First you will see how to write pointcuts based on message signatures, type patterns and how to use (and access) method arguments.

*Use Method Signature Patterns*

The most typical pointcut expressions are used to match a number of methods by their signatures . For example, the following pointcut expression matches all of the methods declared in the ArithmeticCalculator interface. The initial wildcard matches methods with any modifier (public, protected, and private) and any return type. The two dots in the argument list match any number of arguments.

execution(\* com.apress.springrecipes.calculator.ArithmeticCalculator.\*(..))

You can omit the package name if the target class or interface is located in the same package as the aspect.

execution(\* ArithmeticCalculator.\*(..))

The following pointcut expression matches all the public methods declared in the ArithmeticCalculator interface:

execution(public \* ArithmeticCalculator.\*(..))

You can also restrict the method return type. For example, the following pointcut matches the methods that return a double number:

execution(public double ArithmeticCalculator.\*(..))

The argument list of the methods can also be restricted. For example, the following pointcut matches the methods whose first argument is of primitive double type. The two dots then match any number of followed arguments.

execution(public double ArithmeticCalculator.\*(double, ..))

Or, you can specify all the argument types in the method signature for the pointcut to match.

execution(public double ArithmeticCalculator.\*(double, double))

Although the AspectJ pointcut language is powerful in matching various join points, sometimes you may not be able to find any common characteristics (e.g., modifiers, return types, method name patterns, or arguments) for the methods you want to match. In such cases, you can consider providing a custom annotation for them. For instance, you can define the following marker annotation. This annotation can be applied to both method level and type level.

**package** com.apress.springrecipes.calculator;  
  
**import** java.lang.annotation.Documented;  
**import** java.lang.annotation.ElementType;  
**import** java.lang.annotation.Retention;  
**import** java.lang.annotation.RetentionPolicy;  
**import** java.lang.annotation.Target;  
  
@Target( { ElementType.METHOD, ElementType.TYPE })  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
**public @interface** LoggingRequired {  
}

Next, you can annotate all methods that require logging with this annotation or the class itself to apply the behavior to all methods. Note that the annotations must be added to the implementation class but not the interface, as they will not be inherited.

**package** com.apress.springrecipes.calculator;  
  
@LoggingRequired  
**public class** ArithmeticCalculatorImpl **implements** ArithmeticCalculator {  
  
    **public** double add(double a, double b) {  
        ...  
    }  
  
    **public** double sub(double a, double b) {  
        ...  
    }  
  
    ...  
}

Then you can write a pointcut expression to match a class or methods with the @LoggingRequired annotation using the annotation keyword on the @Pointcut annotation.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
**public class** CalculatorPointcuts {  
  
    @Pointcut("annotation(com.apress.springrecipes.calculator.LoggingRequired)")  
    **public** void loggingOperation() {}  
  
}

*Use Type Signature Patterns*

Another kind of pointcut expression matches all join points within certain types. When applied to Spring AOP, the scope of these pointcuts will be narrowed to matching all method executions within the types. For example, the following pointcut matches all the method execution join points within the com.apress.springrecipes.calculator package:

within(com.apress.springrecipes.calculator.\*)

To match the join points within a package and its subpackage, you have to add one more dot before the wildcard.

within(com.apress.springrecipes.calculator..\*)

The following pointcut expression matches the method execution join points within a particular class:

within(com.apress.springrecipes.calculator.ArithmeticCalculatorImpl)

Again, if the target class is located in the same package as this aspect, the package name can be omitted.

within(ArithmeticCalculatorImpl)

You can match the method execution join points within all classes that implement the ArithmeticCalculator interface by adding a plus symbol.

within(ArithmeticCalculator+)

The custom annotation @LoggingRequired can be applied to the class or method level, as illustrated previously.

**package** com.apress.springrecipes.calculator;  
  
@LoggingRequired  
**public class** ArithmeticCalculatorImpl **implements** ArithmeticCalculator {  
    ...  
}

Then you can match the join points within the classes or methods that have been annotated with @LoggingRequired using the within keyword on the @Pointcut annotation.

@Pointcut("within(com.apress.springrecipes.calculator.LoggingRequired)")  
public void loggingOperation() {}

*Combine Pointcut Expressions*

In AspectJ, pointcut expressions can be combined with the operators && (and), || (or), and ! (not). For example, the following pointcut matches the join points within classes that implement either the ArithmeticCalculator or UnitCalculator interface:

within(ArithmeticCalculator+) || within(UnitCalculator+)

The operands of these operators can be any pointcut expressions or references to other pointcuts.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
**public class** CalculatorPointcuts {  
  
    @Pointcut("within(ArithmeticCalculator+)")  
    **public** void arithmeticOperation() {}  
  
    @Pointcut("within(UnitCalculator+)")  
    **public** void unitOperation() {}  
  
    @Pointcut("arithmeticOperation() || unitOperation()")  
    **public** void loggingOperation() {}  
}

*Declare Pointcut Parameters*

One way to access join point information is by reflection (i.e., via an argument of type org.aspectj.lang.JoinPoint in the advice method). Besides, you can access join point information in a declarative way by using some kinds of special pointcut expressions . For example, the expressions target() and args() capture the target object and argument values of the current join point and expose them as pointcut parameters. These parameters are passed to your advice method via arguments of the same name.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Before;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @Before("execution(\* \*.\*(..)) && target(target) && args(a,b)")  
    **public** void logParameter(Object target, double a, double b) {  
        log.info("Target class : {}", target.getClass().getName());  
        log.info("Arguments : {}, {}", a,b);  
    }  
}

When declaring an independent pointcut that exposes parameters, you have to include them in the argument list of the pointcut method as well.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
**public class** CalculatorPointcuts {  
    ...  
    @Pointcut("execution(\* \*.\*(..)) && target(target) && args(a,b)")  
    **public** void parameterPointcut(Object target, double a, double b) {}  
}

Any advice that refers to this parameterized pointcut can access the pointcut parameters via method arguments of the same name.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.Before;  
  
@Aspect  
**public class** CalculatorLoggingAspect {  
    ...  
    @Before("CalculatorPointcuts.parameterPointcut(target, a, b)")  
    **public** void logParameter(Object target, double a, double b) {  
        log.info("Target class : {}", target.getClass().getName());  
        log.info("Arguments : {}, {}"a,b);  
    }  
}

2-18. Use AOP for introductions for POJOs

PROBLEM

Sometimes you may have a group of classes that share a common behavior. In OOP, they must extend the same base class or implement the same interface. This issue is actually a crosscutting concern that can be modularized with AOP. In addition, the single inheritance mechanism of Java only allows a class to extend one base class at most. So, you cannot inherit behaviors from multiple implementation classes at the same time.

SOLUTION

An introduction is a special type of advice in AOP. It allows objects to implement an interface dynamically by providing an implementation class for that interface. It seems as if objects extend an implementation class at runtime. Moreover, you are able to introduce multiple interfaces with multiple implementation classes to your objects at the same time. This can achieve the same effect as multiple inheritance.

HOW IT WORKS

Suppose you have two interfaces, MaxCalculator and MinCalculator, to define the max() and min() operations.

**package** com.apress.springrecipes.calculator;  
  
**public interface** MaxCalculator {  
  
    **public** double max(double a, double b);  
}  
**package** com.apress.springrecipes.calculator;  
  
**public interface** MinCalculator {  
  
    **public** double min(double a, double b);  
}

Then you have an implementation for each interface with println statements to let you know when the methods are executed.

**package** com.apress.springrecipes.calculator;  
  
**public class** MaxCalculatorImpl **implements** MaxCalculator {  
  
    **public** double max(double a, double b) {  
        double result = (a >= b) ? a : b;  
        System.out.println("max(" + a + ", " + b + ") = " + result);  
        **return** result;  
    }  
}  
**package** com.apress.springrecipes.calculator;  
  
**public class** MinCalculatorImpl **implements** MinCalculator {  
  
    **public** double min(double a, double b) {  
        double result = (a <= b) ? a : b;  
        System.out.println("min(" + a + ", " + b + ") = " + result);  
        **return** result;  
    }  
}

Now, suppose you want ArithmeticCalculatorImpl to perform the max() and min() calculation also. As the Java language supports single inheritance only, it is not possible for the ArithmeticCalculatorImpl class to extend both the MaxCalculatorImpl and MinCalculatorImpl classes at the same time. The only possible way is to extend either class (e.g., MaxCalculatorImpl) and implement another interface (e.g., MinCalculator), either by copying the implementation code or by delegating the handling to the actual implementation class. In either case, you have to repeat the method declarations.

With an introduction, you can make ArithmeticCalculatorImpl dynamically implement both the MaxCalculator and MinCalculator interfaces by using the implementation classes MaxCalculatorImpl and MinCalculatorImpl. It has the same effect as multiple inheritance from MaxCalculatorImpl and MinCalculatorImpl. The idea behind an introduction is that you needn’t modify the ArithmeticCalculatorImpl class to introduce new methods. That means you can introduce methods to your existing classes even without source code available.

**Tip**

You may wonder how an introduction can do that in Spring AOP. The answer is a dynamic proxy. As you may recall, you can specify a group of interfaces for a dynamic proxy to implement. Introduction works by adding an interface (e.g., MaxCalculator) to the dynamic proxy. When the methods declared in this interface are called on the proxy object, the proxy will delegate the calls to the back-end implementation class (e.g., MaxCalculatorImpl).

Introductions, like advices, must be declared within an aspect. You may create a new aspect or reuse an existing aspect for this purpose. In this aspect, you can declare an introduction by annotating an arbitrary field with the @DeclareParents annotation.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.DeclareParents;  
  
**import** org.springframework.stereotype.Component;  
  
@Aspect  
@Component  
**public class** CalculatorIntroduction {  
  
    @DeclareParents(  
        value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",  
        defaultImpl = MaxCalculatorImpl.class)  
    **public** MaxCalculator maxCalculator;  
  
    @DeclareParents(  
        value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",  
        defaultImpl = MinCalculatorImpl.class)  
    **public** MinCalculator minCalculator;  
}

The value attribute of the @DeclareParents annotation type indicates which classes are the targets for this introduction. The interface to introduce is determined by the type of the annotated field. Finally, the implementation class used for this new interface is specified in the defaultImpl attribute.

Through these two introductions, you can dynamically introduce a couple of interfaces to the ArithmeticCalculatorImpl class. Actually, you can specify an AspectJ type-matching expression in the value attribute of the @DeclareParents annotation to introduce an interface to multiple classes.

As you have introduced both the MaxCalculator and MinCalculator interfaces to your arithmetic calculator, you can cast it to the corresponding interface to perform the max() and min() calculations.

**package** com.apress.springrecipes.calculator;  
  
**public class** Main {  
  
    **public** static void main(String[] args) {  
        ...  
        ArithmeticCalculator arithmeticCalculator =  
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");  
        ...  
        MaxCalculator maxCalculator = (MaxCalculator) arithmeticCalculator;  
        maxCalculator.max(1, 2);  
  
        MinCalculator minCalculator = (MinCalculator) arithmeticCalculator;  
        minCalculator.min(1, 2);  
    }  
}

2-19. Introduce States to Your POJOs with AOP

PROBLEM

Sometimes you may want to add new states to a group of existing objects to keep track of their usage, such as the calling count, the last modified date, and so on. It should not be a solution if all the objects have the same base class. However, it’s difficult to add such states to different classes if they are not in the same class hierarchy.

SOLUTION

You can introduce a new interface to your objects with an implementation class that holds the state field. Then, you can write another advice to change the state according to a particular condition.

HOW IT WORKS

Suppose you want to keep track of the calling count of each calculator object. Since there is no field for storing the counter value in the original calculator classes, you need to introduce one with Spring AOP. First, let’s create an interface for the operations of a counter .

**package** com.apress.springrecipes.calculator;  
  
**public interface** Counter {  
  
    **public** void increase();  
    **public** int getCount();  
}

Next, just write a simple implementation class for this interface. This class has a count field for storing the counter value.

**package** com.apress.springrecipes.calculator;  
  
**public class** CounterImpl **implements** Counter {  
  
    **private** int count;  
  
    **public** void increase() {  
        count++;  
    }  
  
    **public** int getCount() {  
        **return** count;  
    }  
}

To introduce the Counter interface to all your calculator objects with CounterImpl as the implementation, you can write the following introduction with a type-matching expression that matches all the calculator implementations:

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.annotation.Aspect;  
**import** org.aspectj.lang.annotation.DeclareParents;  
  
@Aspect  
@Component  
**public class** CalculatorIntroduction {  
    ...  
    @DeclareParents(  
        value = "com.apress.springrecipes.calculator.\*CalculatorImpl",  
        defaultImpl = CounterImpl.class)  
    **public** Counter counter;  
}

This introduction introduces CounterImpl to each of your calculator objects. However, it’s still not enough to keep track of the calling count. You have to increase the counter value each time a calculator method is called. You can write an after advice for this purpose. Note that you must get this object but not the target object, as only the proxy object implements the Counter interface.

**package** com.apress.springrecipes.calculator;  
...  
**import** org.aspectj.lang.annotation.After;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
@Component  
**public class** CalculatorIntroduction {  
    ...  
    @After("execution(\* com.apress.springrecipes.calculator.\*Calculator.\*(..))"  
        + " && this(counter)")  
    **public** void increaseCount(Counter counter) {  
        counter.increase();  
    }  
}

In the Main class , you can output the counter value for each of the calculator objects by casting them into the Counter type.

**package** com.apress.springrecipes.calculator;  
  
**public class** Main {  
  
    **public** static void main(String[] args) {  
        ...  
        ArithmeticCalculator arithmeticCalculator =  
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");  
        ...  
  
        UnitCalculator unitCalculator =  
            (UnitCalculator) context.getBean("unitCalculator");  
        ...  
  
        Counter arithmeticCounter = (Counter) arithmeticCalculator;  
        System.out.println(arithmeticCounter.getCount());  
  
        Counter unitCounter = (Counter) unitCalculator;  
        System.out.println(unitCounter.getCount());  
    }  
}

2-20. Use Load-Time Weaving AspectJ Aspects in Spring

PROBLEM

The Spring AOP framework supports only limited types of AspectJ pointcuts and allows aspects to apply to beans declared in the IoC container. If you want to use additional pointcut types or apply your aspects to objects created outside the Spring IoC container, you have to use the AspectJ framework in your Spring application.

SOLUTION

*Weaving* is the process of applying aspects to your target objects. With Spring AOP, weaving happens at runtime through dynamic proxies. In contrast, the AspectJ framework supports both compile-time and load-time weaving.

AspectJ *compile-time weaving* is done through a special AspectJ compiler called *ajc*. It can weave aspects into your Java source files and output woven binary class files. It can also weave aspects into your compiled class files or JAR files. This process is known as *post-compile-time weaving*. You can perform compile-time and post-compile-time weaving for your classes before declaring them in the Spring IoC container. Spring is not involved in the weaving process at all. For more information on compile-time and post-compile-time weaving, please refer to the AspectJ documentation.

AspectJ load-time weaving (also known as LTW) happens when the target classes are loaded into JVM by a class loader. For a class to be woven, a special class loader is required to enhance the bytecode of the target class. Both AspectJ and Spring provide load-time weavers to add load-time weaving capability to the class loader. You need only simple configurations to enable these load-time weavers.

HOW IT WORKS

To understand the AspectJ load-time weaving process in a Spring application, let’s consider a calculator for complex numbers. First, you create the Complex class to represent complex numbers. You define the toString() method for this class to convert a complex number into the string representation (a + bi).

**package** com.apress.springrecipes.calculator;  
  
**public class** Complex {  
  
    **private** int real;  
    **private** int imaginary;  
  
    **public** Complex(int real, int imaginary) {  
        **this**.real = real;  
        **this**.imaginary = imaginary;  
    }  
  
    *// Getters and Setters*   
    ...  
  
    **public** String toString() {  
        **return** "(" + real + " + " + imaginary + "i)";  
    }  
}

Next, you define an interface for the operations on complex numbers. For simplicity’s sake, only add() and sub() are supported.

**package** com.apress.springrecipes.calculator;  
  
**public interface** ComplexCalculator {  
  
    **public** Complex add(Complex a, Complex b);  
    **public** Complex sub(Complex a, Complex b);  
}

The implementation code for this interface is as follows. Each time, you return a new complex object as the result.

**package** com.apress.springrecipes.calculator;  
  
**import** org.springframework.stereotype.Component;  
  
@Component("complexCalculator")  
**public class** ComplexCalculatorImpl **implements** ComplexCalculator {  
  
    **public** Complex add(Complex a, Complex b) {  
        Complex result = **new** Complex(a.getReal() + b.getReal(),  
            a.getImaginary() + b.getImaginary());  
        System.out.println(a + " + " + b + " = " + result);  
        **return** result;  
    }  
  
    **public** Complex sub(Complex a, Complex b) {  
        Complex result = **new** Complex(a.getReal() - b.getReal(),  
            a.getImaginary() - b.getImaginary());  
        System.out.println(a + " - " + b + " = " + result);  
        **return** result;  
    }  
}

Now, you can test this complex number calculator with the following code in the Main class:

**package** com.apress.springrecipes.calculator;  
  
**import** org.springframework.context.ApplicationContext;  
**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;  
  
**public class** Main {  
  
    **public** static void main(String[] args) {  
  
        ApplicationContext context =  
            **new** AnnotationConfigApplicationContext(CalculatorConfiguration.class);  
  
        ComplexCalculator complexCalculator =  
            context.getBean("complexCalculator", ComplexCalculator.class);  
  
        complexCalculator.add(**new** Complex(1, 2), **new** Complex(2, 3));  
        complexCalculator.sub(**new** Complex(5, 8), **new** Complex(2, 3));  
    }  
}

So far, the complex calculator is working fine. However, you may want to improve the performance of the calculator by caching complex number objects. As caching is a well-known crosscutting concern, you can modularize it with an aspect.

**package** com.apress.springrecipes.calculator;  
  
**import** java.util.Collections;  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
**import** org.aspectj.lang.ProceedingJoinPoint;  
**import** org.aspectj.lang.annotation.Around;  
**import** org.aspectj.lang.annotation.Aspect;  
  
@Aspect  
**public class** ComplexCachingAspect {  
  
    **private** final Map<String, Complex> cache = **new** ConcurrentHashMap<>();  
  
    @Around("call(public Complex.new(int, int)) && args(a,b)")  
    **public** Object cacheAround(ProceedingJoinPoint joinPoint, int a, int b)  
        **throws** Throwable {  
        String key = a + "," + b;  
        Complex complex = cache.get(key);  
        **if** (complex == **null**) {  
            System.out.println("Cache MISS for (" + key + ")");  
            complex = (Complex) joinPoint.proceed();  
            cache.put(key, complex);  
        }  
        **else** {  
            System.out.println("Cache HIT for (" + key + ")");  
        }  
        **return** complex;  
    }  
}

In this aspect, you cache the complex objects in a map with their real and imaginary values as keys. Then, the most suitable time to look up the cache is when a complex object is created by invoking the constructor. You use the AspectJ pointcut expression call to capture the join points of calling the Complex(int,int) constructor.

Next, you need an around advice to alter the return value. If a complex object of the same value is found in the cache, you return it to the caller directly. Otherwise, you proceed with the original constructor invocation to create a new complex object. Before you return it to the caller, you cache it in the map for subsequent usages.

The call pointcut is not supported by Spring AOP , so if you attempt to let Spring scan the pointcut annotation, you’ll get the error “unsupported pointcut primitive call*.*”

Because this type of pointcut is not supported by Spring AOP, you have to use the AspectJ framework to apply this aspect. The configuration of the AspectJ framework is done through a file named aop.xml in the META-INF directory of the classpath root.

<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"  
    "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">  
  
**<aspectj>**   
    **<weaver>**   
        **<include** within="com.apress.springrecipes.calculator.\*" **/>**   
    **</weaver>**   
  
    **<aspects>**   
        **<aspect**   
            name="com.apress.springrecipes.calculator.ComplexCachingAspect" **/>**   
    **</aspects>**   
**</aspectj>**

In this AspectJ configuration file, you have to specify the aspects and which classes you want your aspects to weave in. Here, you specify weaving ComplexCachingAspect into all the classes in the com.apress.springrecipes.calculator package.

Finally, to make this load-time weaving, you need to run the application in one of two ways, as described in the next sections.

*Implement Load-Time Weaving with the AspectJ Weaver*

AspectJ provides a load-time weaving agent to enable load-time weaving. You need only to add a VM argument to the command that runs your application. Then your classes will get woven when they are loaded into the JVM.

java -javaagent:lib/aspectjweaver-1.9.0.jar -jar Recipe\_2\_19\_ii-4.0.0.jar

If you run your application with the preceding argument, you will get the following output and cache status. The AspectJ agent advises all calls to the Complex(int,int) constructor.

Cache MISS for (1,2)  
Cache MISS for (2,3)  
Cache MISS for (3,5)  
(1 + 2i) + (2 + 3i) = (3 + 5i)  
Cache MISS for (5,8)  
Cache HIT for (2,3)  
Cache HIT for (3,5)  
(5 + 8i) - (2 + 3i) = (3 + 5i)

*Implement Load-Time Weaving with Spring Load-Time Weaver*

Spring has several load-time weavers for different runtime environments. To turn on a suitable load-time weaver for your Spring application, you need only to add @EnableLoadTimeWeaving to your configuration class.

Spring will be able to detect the most suitable load-time weaver for your runtime environment. Some Java EE application servers have class loaders that support the Spring load-time weaver mechanism, so there’s no need to specify a Java agent in their startup commands.

However, for a simple Java application , you still require a weaving agent provided by Spring to enable load-time weaving. You have to specify the Spring agent in the VM argument of the startup command.

java -javaagent:lib/spring-instrument-5.0.0.jar -jar Recipe\_2\_19\_iii-4.0.0.jar

However, if you run your application, you will get the following output and cache status:

Cache MISS for (3,5)  
(1 + 2i) + (2 + 3i) = (3 + 5i)  
Cache HIT for (3,5)  
(5 + 8i) - (2 + 3i) = (3 + 5i)

This is because the Spring agent advises only the Complex(int,int) constructor calls made by beans declared in the Spring IoC container. As the complex operands are created in the Main class, the Spring agent will not advise their constructor calls.

2-21. Configure AspectJ Aspects in Spring

PROBLEM

Aspects used in the AspectJ framework are instantiated by the AspectJ framework itself. Therefore, you have to retrieve the aspect instances from the AspectJ framework to configure them.

SOLUTION

Each AspectJ aspect provides a factory class called Aspects that has a static factory method called aspectOf(), which allows you to access the current aspect instance. In the Spring IoC container, you can declare a bean created by this factory method by calling Aspects.aspectOf(ComplexCachingAspect.class).

HOW IT WORKS

For instance, you can allow the cache map of ComplexCachingAspect to be preconfigured via a setter method.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.ProceedingJoinPoint;  
**import** org.aspectj.lang.annotation.Around;  
**import** org.aspectj.lang.annotation.Aspect;  
  
**import** java.util.Map;  
**import** java.util.concurrent.ConcurrentHashMap;  
  
@Aspect  
**public class** ComplexCachingAspect {  
  
    **private** Map<String, Complex> cache = **new** ConcurrentHashMap<>();  
  
    **public** void setCache(Map<String, Complex> cache) {  
        **this**.cache.clear();  
        **this**.cache.putAll(cache);  
    }  
  
    @Around("call(public Complex.new(int, int)) && args(a,b)")  
    **public** Object cacheAround(ProceedingJoinPoint joinPoint, int a, int b) **throws** Throwable {  
        String key = a + "," + b;  
        Complex complex = cache.get(key);  
        **if** (complex == **null**) {  
            System.out.println("Cache MISS for (" + key + ")");  
            complex = (Complex) joinPoint.proceed();  
            cache.put(key, complex);  
        } **else** {  
            System.out.println("Cache HIT for (" + key + ")");  
        }  
        **return** complex;  
    }  
  
}

To configure the aspect, create an @Bean annotated method that calls the aforementioned factory method Aspects.aspectOf; this will give you the instance of the aspect. This instance can in turn be configured.

**package** com.apress.springrecipes.calculator;  
  
**import** org.aspectj.lang.Aspects;  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.ComponentScan;  
**import** org.springframework.context.annotation.Configuration;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
@Configuration  
@ComponentScan  
**public class** CalculatorConfiguration {  
  
    @Bean  
    **public** ComplexCachingAspect complexCachingAspect() {  
  
        Map<String, Complex> cache = **new** HashMap<>();  
        cache.put("2,3", **new** Complex(2,3));  
        cache.put("3,5", **new** Complex(3,5));  
  
        ComplexCachingAspect complexCachingAspect =            Aspects.aspectOf(ComplexCachingAspect.class);  
        complexCachingAspect.setCache(cache);  
        **return** complexCachingAspect;  
    }  
}

To run the application , you use AspectJ’s weaver.

java -javaagent:lib/aspectjweaver-1.9.0.jar -jar Recipe\_2\_20-4.0.0.jar

2-22. Inject POJOs into Domain Objects with AOP

PROBLEM

Beans declared in the Spring IoC container can wire themselves to one another through Spring’s dependency injection capability. However, objects created outside the Spring IoC container cannot wire themselves to Spring beans via configuration. You have to perform the wiring manually with programming code.

SOLUTION

Objects created outside the Spring IoC container are usually domain objects. They are often created using the new operator or from the results of database queries. To inject a Spring bean into domain objects created outside Spring, you need the help of AOP. Actually, the injection of Spring beans is also a kind of crosscutting concern. As the domain objects are not created by Spring, you cannot use Spring AOP for injection. Spring supplies an AspectJ aspect specialized for this purpose. You can enable this aspect in the AspectJ framework.

HOW IT WORKS

Suppose you have a global formatter to format complex numbers. This formatter accepts a pattern for formatting and uses the standard @Component and @Value annotations to instantiate a POJO .

**package** com.apress.springrecipes.calculator;  
  
@Component  
**public class** ComplexFormatter {  
  
    @Value("(a + bi)")  
    **private** String pattern;  
  
    **public** void setPattern(String pattern) {  
        **this**.pattern = pattern;  
    }  
  
    **public** String format(Complex complex) {  
        **return** pattern.replaceAll("a", Integer.toString(complex.getReal()))  
                .replaceAll("b", Integer.toString(complex.getImaginary()));  
    }  
}

In the Complex class, you want to use this formatter in the toString() method to convert a complex number into a string. It exposes a setter method for ComplexFormatter.

**package** com.apress.springrecipes.calculator;  
  
**public class** Complex {  
  
    **private** int real;  
    **private** int imaginary;  
    ...  
    **private** ComplexFormatter formatter;  
  
    **public** void setFormatter(ComplexFormatter formatter) {  
        **this**.formatter = formatter;  
    }  
  
    **public** String toString() {  
        **return** formatter.format(**this**);  
    }  
}

However, because Complex objects are not instantiated by the Spring IoC container, they cannot be configured for dependency injection in the regular manner. Spring includes AnnotationBeanConfigurerAspect in its aspect library to configure the dependencies of any objects, even if they were not created by the Spring IoC container.

First, you have to annotate your object type with the @Configurable annotation to declare that this type of object is configurable.

**package** com.apress.springrecipes.calculator;  
  
**import** org.springframework.beans.factory.annotation.Configurable;  
**import** org.springframework.beans.factory.annotation.Configurable;  
**import** org.springframework.context.annotation.Scope;  
  
@Configurable  
@Component  
@Scope("prototype")  
**public class** Complex {  
    ...  
    @Autowired  
    **public** void setFormatter(ComplexFormatter formatter) {  
        **this**.formatter = formatter;  
    }  
  
}

In addition to the @Configurable annotation, you decorate the POJO with the standard @Component, @Scope, and @Autowired annotations so the bean gets its standard Spring behaviors. However, the @Configurable annotation is the most important configuration piece, and for this Spring defines a convenient annotation, @EnableSpringConfigured, for you to enable the mentioned aspect.

@Configuration  
@EnableSpringConfigured  
@ComponentScan  
**public class** CalculatorConfiguration {}

When a class with the @Configurable annotation is instantiated, the aspect will look for a prototype-scoped bean definition whose type is the same as this class. Then, it will configure the new instances according to this bean definition. If there are properties declared in the bean definition, the new instances will also have the same properties set by the aspect.

Finally, to run the application, you weave the aspect into your classes at load time with the AspectJ agent.

java -javaagent:lib/aspectjweaver-1.9.0.jar -jar Recipe\_2\_21-4.0.0.jar