AOP

SpringTutors January 29, 2016 Features 121 Views

AOP(Aspect Oriented Programming)

What is an AOP ?

Aop is a programming methodology. It means it is programming principle which helps in running the primary business logic and cross cutting logic (secondary logic) written in two different component without referencing each other.

Note:- AOP is not a programming language.
Java is a programming language which supports oops principle so it is called as object oriented programming language. It does not support AOP. Similarly, Any programming languages or Framework which follows the principles of AOP then that programming language or Framework is called as AOP style of programming or Framework.
AOP is not specific to spring. AOP principles are supported by spring like various framework such as AspectJ, JAC, JBossAOP etc. Among various AOP based programming language or framework most popular once are AspectJ and Spring AOP.

Let's suppose there is an application for adding two number as follows-

Business logic

```
package com.demo.bean;
 public class Calculator
 {
 public int add(int a,int b)
 {
 return a + b;
 }
 }
```

Caching logic

```
package com.demo.bean;
 import java.util.Map;
 import java.util.concurrent.ConcurrentHashMap;
 public class Cache
 {
 private static Cache instance;
 private Map<String,Object> mapData;
 private Cache()
 {
 mapData=new ConcurrentHashMap<String, Object>();
 }
 public static synchronized Cache getInstance()
 {
 if(instance==null)
 {
```

```
instance=new Cache();
}
return instance;
}
public void put(String key,Object value)
{
mapData.put(key,value);
}
public Object get(String key)
{
return mapData.get(key);
}
public boolean containsKey(String key)
{
return mapData.containsKey(key);
}
}
```

Now in the above application if user wants to add two same (10,20) value repeatedly then all the time our application has to perform the operation and return the same value (30) to the user. Instead of performing the operation each and every time for the same value if we store the result for that two same value in cache memory then whenever user ask for the operation we can give them from cache memory. It will make our application fast. Similarly if we want to know the control flow of our application then we have to write the logging logic. In this way we can see that to make our business logic work better we require some helper logic which can be called as secondary logic or cross-cutting logic. Here Caching and logging are cross-cutting logic and add(–) method of Calculator application containing business logic. In this way we can say that in every real-time application there will be two types of logic one is primary business logic and other is cross-cutting logic to make the application perform better.

Why we need AOP ?

In java if we have to write the cross-cutting logic for a business logic then we have to write that logic in either business class itself or in another class. when we write the cross-cutting logic in another class then to get benefit of the cross-cutting logic business logic have to refer to that class. In both the cases we have to modify the business code when we do not want to use the cross-cutting logic. In this way we can see that in any cases business logic and cross-cutting logic get tightly coupled. To solve this problem we requires AOP. AOP tells the programmer to write business and cross-cutting logic in two different component I will take care of running the both you need not to refer to any one explicitly. Now by using AOP developers will concentrate on writing business logic more and any time they can write the cross-cutting logic for making business logic perform better. These are some reasons we require AOP.

Aop principles

1.Aspect:- It is the piece of logic or code which has to be applied on a target class joinpoint.

2.Advice:- It tells how to execute the Aspect on joinpoint .

3.JoinPoint:- It is the point on the target class where Advice can be applied to execute the Apect.

4.Pointcut:- It is the set of JoinPoint where advices are applied to execute the Aspect.

5.Weaving:-It is the mixture of Target and Advice which produces Proxy.

6.Proxy:- The result comes after weaving is Proxy.

7.Target:- It is the class on which Aspect is going to be adviced at joinpoint.

When Spring released AOP at that time AspectJ has already accepted in the market and it seems to be more powerful than Spring AOP. Instead of building Spring AOP more powerful than AspectJ Spring people provided integration to the AspectJ to take the benefits of it. From Spring 2.x Spring not only supports its own AOP it allows us to work with AspectJ also.

Initially when spring has not support for AspectJ it provides only one way to work with AOP that is Programmatic approach after integration with AspectJ it provides two more way to work with it one is Spring AspectJ declaretive approach and another is AspectJ Annotation approach.

Types Of Advices

1. Around Advice :- This is the advice which executes the aspect logic (cross-cutting logic) around the target class joinpoint. In this case advice method will be called before the target class method execution and after the target class method execution. This advice can control the target class method execution.
2. Before Advice :- In this advice before executing the target class method it will execute the advice method and after finishing the execution of advice method control will not come back to the advice method.
3. After Returning Advice :- In this advice method will executes after executing the target class method successfully but before returning the value to the caller.
4. Throws Advice :- This advice method will be invoked only when the target class method throws an exception.

Programamatic AOP

To work with programmatic approach we have to use Spring AOP API. Programmatic AOP Supports All type of Advices defined above. let’s start working with one advice after another and understand how it is going to work

AroundAdvice

To work with any advice we need target class

Target Class:-

```
Class Calculator
 {
 public int add(int a, int b)
 {
 return a + b;
 }
 }
```

Spring support joinPoint at method level. Now in the target class we have add(–) methods which is a joinpoint. Now on this joinpoint how we want to apply aspect is known by advice here we want to use around advice. AroundAdvice means before the target class method execution and after the method execution of target class we want to perform some additional functionality on target. Now suppose we want to perform logging operation on target to know the details of execution. Logging operation can be applied on any classes and any methods in our application to get the execution details of application so writing this Logging logic in every class or at every method will be bad approach. To deal with this Aop provides very suitable way-

```
Write the Logging logic(Aspect) in separate component like following-
```

```
Class LoggingAspect imports MethodInterceptor
 {
 Public Object invoke(MethodInvocation methodInvocation)
 {
 String methodName=methodInvocation.getMaethod().getName();
 Object[] args=methodInvocation.getArguments();//this will gives us the arguments by which we
//to get the  type of arguments we have to use getDeclaredType()
 //now after getting the method name and value by which method is called if we want to perform some logging operation we can perform by
 System.out.println("Before executing  "+methodName+"With value "+args[0]+" "+args[1]);
```

```
//Now if we complete our work then we want to send the request to target class method to execute then we have to tell the AOP to proceed
Object sum=methodInvocation.proceed();
//methodInvocation contains all the details about the target method so AOP will execute that method only whose details are with methodIn
//now again we want to perform some Logging operation then do like following-
System.out.println(methodName +" method execution is finished with "+args[0]+" "+args[1]);
// As add(--) is invoked from invoke method of Aspect so target will return the value to Aspect invoke(--)
return sum;//this will return the sum to the programmer.
}
}
```

Now to execute Target and Aspect(Cross cutting logic or cross cutting concerns or secondary logic) together we have to give both to the Aop. Aop will take it and weaved both to generate proxy at run-time(Spring supports both static weaving and dynamic weaving but on integration with AspectJ it supports static Weaving only).
After generating the proxy it gives its reference to programmer.

Let's write weaving class

Weaving class

```
Class Test
{
public void main (String[] args)
{
proxyFactory pf=new ProxyFactory();
Pf.setTarget(new Calculator);
Pf.addAdvice(new LoggingAdvice);
Calculator proxy=(Calculator)pf.getProxy();
proxy.add(10,20);
}
}
```

Now let’s see the control flow of Around Advice

When we call proxy.add(10,20) control does not go to target class add(–) method instead of going to target class add(–) method it will goes to Advice class MethodInterceptor invoke(MethodInvocation methodInvocation) method. Internally Aop will pass the details of target class and its method details on which we want to apply crosscutting logic to methodInvocation. Now we can get the details of target class method like following-

String methodName=methodInvocation.getMethod().getName(); It will give the name of the method on which we want to apply aspect that is cross-cutting logic.
Object[] args=methodInvocation.getArguments(); It will give the arguments value by which target class is called.
Object arg=methodInvocation.getThis(); It will gives the target class object. By using target class we can access the attribute which is declared at class level.

Now by getting the details of method we can perform some action on target class method before executing its method like following-

//By using Logger
System.out.println(“Entering in method ”+methodName+” with value ”+args[0]+”,”+args[1]); Even we can modify the value which is passed by caller to call the target method like following-
args[0]=(Integer)args[0]+1;
args[1]=(Integer)args[1]+1;
Now the modified value will reflect on the object of target when we call the target class from invoke(-) because of calling by reference. After performing the before logic if want to call the target class metod then-
Object ret=methodInvocation.proceed(); When proceed() is called the control goes to target class add(–) method and starts execution. Now it will execute by passing the modified value as argument. After executing the add(–) it will return the control to proceed() method because it is called by the proceed(). We

can also skeep the execution of target class method just by not calling methoodInvocation.proceed(); Now if we want to perform some operation after executing the target class method then we can do. In this case –System.out.println("Exit from "+methodName+" with value "+ret); If we want to modify the ret value we can modify because it is visible in invoke(-)-
ret=(Integer)ret+1; After performing after crosscutting logic we can return the value to caller as ret. Now the caller of the target class add(–)method will get 33 as result. But caller expect the value 30. In this we can see that by using around advice programmer can control the execution of target class method at different places.

Control point in Around Advice

When we use around advice it gives three control points to the programmer as we can see in the above explanation.

1. Programmer can modify the original value passed by caller because the details of target class is supplied by AOP to MethodInvocation and it will reflect in target class object because AOP will call the target method by passing argument by reference.
   Ex-
   args[0]=(Integer)args[0]+1;
   args[1]=(Integer)args[1]+1;
   2.Programmer can control the execution of target class method. Programmer can skeep the executing of the target class method just by not calling the methodInvocation.proceed();
   3.Programmer can modify the result which is returned by the target class method.
   Ex-
   ret=(Integer)ret+1;

Use case-2

Around Advice can be used to fine tune the performance by using Cache.

Let's Understand the Internal control flow of around advice

In weaving class we have written

```
ProxyFactory pf=new ProxyFactory();
Pf.setTarget(new Calculator);
Pf.addAdvice(new LoggingAdvice);
Calculator proxy=(Calculator)pf.getProxy();
proxy.add(10,20);
```

When pf.getProxy() is called at run time internally AOP will create the Proxy class of which we have set as

Target like following-

```
Class Calculator$proxy extends Calculator
{
Public int add(int a,int b)
{
}
}
```

Aop will create the proxy class Calculator$proxy by extending from original target class (Calculator). It will do for Calculator because initially we have set the target class object as a calculator. Now Calculator$proxy is the child class of Calculator class so Calculator can hold the reference of its child class that's why we have type cast the proxy to Calculator like following in weaving-

Calculator proxy=(Calculatror)pf.getProxy().

Now Aop will override the add(–)method in Proxy like following-

```
Class Calculator$proxy extends Calculator
private Object target;
MethodInvocation methodInvocation=new MethodInvocation();
{
public int add(int a,int b)
{
methodInvocation.setThis(target); //Calculator.class
methodInvocation.setArguments(new Object[] {a,b});
methodInvocation.setAdviceChain(adviceChain);
methodInvocation.setMethod(Thread.currentmethod.getStackTrace);
Object ret=methodInvocation.invoke(methodInvocation);
}
}
```

```
Class MethodInvocation
{
private Object this;
private Object[] arguments;
private Object args ;
private Object adviceChain;
private Object executingAdvice;
public Object proceed()
{
Return method.invoke(this,args);
}
//Setter
//Getter
}
```

Now when we call the proxy.add(10,20) instead of going to target class method it will goes to LoggingAdvice invoke() method in which logging logics are written. It happens because after weaving we get the proxy and when we call the proxy.add(10,20) control goes to proxy class add(–) method. AOP will create the proxy class at run time by extending from Target class after weaving. add(–) method of proxy class is overridden in which invoke method of AroundAdvice is called by passing all the details of target method.

So when in invoke() method we call method invocation.proceed() it will goes to target class method and target class will return the value to the invoke method. After finishing the invoke() method execution it will return the value to the proxy class add method which will finally return the value to the callee.

Before Advice

Before Advice apply the aspects before the joinpoint. It means before executing the target class method it will apply the aspects on joinpoint.

Usecase-1 –Auditing

Let’s see an example-

Target class-

```
Class LoanAprover
{
public boolean approve()
{
return true;
}
}
```

Now we can see that approve() method is the joinpoint in target class which contains the primary business. Now we can understand that approving of loan cannot be done by any one so we need auditing which will keep the records of the user who access the approve() and who try to access it.

Now we can see Auditing logic is crosscutting logic which is to be used before the execution of approve() method so we need Before Advice.

MethodBeforeAdvice (Provided by Spring AOP API)

```
Class AuditAdvice implements MethodBeforeAdvice
{
public void before(Method method,Object[] args,Object target)
{
String methodName=method.getName();
System.out.println(“User1 has access the ”+methodName);
}
}
```

Write Weaving class

```
Class Test
{
public static void main(String[] args)
{
ProxyFactory pf=new Proxyfactory();
pf.setTarget(new LoanApprover());
pf.addAdvice(new AuditAdvice());
LoanApprover proxy=(LoanApprover)pf.getProxy();
proxy.approve();
}
}
```

Now when we call proxy.approve(), before calling the target class approve() method control will goes to MethodBeforeAdvice before(Method method,Object[] args,Object target) method. We can see the details of target class method by using "this" we can perform some auditing operation. Once the before operation finished control automatically goes to target approve() method which will execute and returns the value to caller. It will not return to before(—) because it will not called from there. Target class method will be called automatically by Aop after finishing the before(—) operation.

AfterReturnning Advice

In this advice method will executes after executing the target class method successfully but before returning the value to the caller.

Target Class

```
Class KeyGenerator {
Public int  generateKey(int len)
{
if(len<=8)
{
return 0;
}
return 1;
}
```

Advice Class

```
Class VerifyKeyAdvice implements AfterReturningAdvice
public void afterReturning(Object ret, Method method, Object[] args,Object target) throws Throwable
{
int r=Integer.parseInt(ret.toString());
if(r<=0)
{
throw new Exception("weak key");
}
}
```

Weaving Class

```
public class Test {
public static void main(String[] args) {
ProxyFactory pf=new ProxyFactory();
pf.setTarget(new KeyGenerator());
pf.addAdvice(new VerifyKeyAdvice());
KeyGenerator proxy=(KeyGenerator) pf.getProxy();
int value=proxy.generateKey(8);
System.out.println(value);
}
}
```

In this case control comes to the afterReturning (—-) when target class execute normally. If target class throw any exception then control does not comes to afterReturning(–). After executing afterReturning (—-) method of VerifyAdvice class return value will automatically goes to callee. In this case we can abrupt the execution of target class by throwing an exception in afterReturning (–) method.

Throws Advice

This advice method will be invoked only when the target class method throws an exception.

Target Class

```
package com.ta.bean;
public class Thrower {
public int throwing(int i)
{
 if(i<=0)
 {
 throw new IllegalArgumentException("In valid Input");
 }
 else
 {
 return 1;
 }
}
}
```

Throws Advice Class

```
package com.ta.bean;
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;
public class LoggingThrowsAdvice implements ThrowsAdvice {
public void afterThrowing(IllegalArgumentException iae)
{
System.out.println("Please provide positive value");
System.out.println( new IllegalArgumentException("invalid i"));
}
public void afterThrowing(Method method,Object[] args,Object target,IllegalArgumentException iae)
{
System.out.println("Please provide positive value");
System.out.println( new IllegalArgumentException("invalid value of i"));
 throw new IllegalArgumentException("invalid i="+args[0]+" Please provide positive value to "+method.getName());
}
}
```

Weaving Class

```
package com.ta.test;
import org.springframework.aop.framework.ProxyFactory;
import com.ta.bean.LoggingThrowsAdvice;
import com.ta.bean.Thrower;
public class TATest {
 public static void main(String[] args) {
 ProxyFactory pf=new ProxyFactory();
 pf.setTarget(new Thrower());
 pf.addAdvice(new LoggingThrowsAdvice());
 Thrower proxy=(Thrower) pf.getProxy();
System.out.println(proxy.throwing(-1));
 }
}
```

In case of throws advice control will comes to this advice method only when target class method throws exception.When target class throws exception then control comes to the throws advice method in which we can see the exception thrown by target class method and represent that exception in end-user understanding format. The ain purpose of the throws advice is to centralized the error handling mechanism.