Spring AOP

When we are developing a software the utmost importance of the developer is to write the business logic , so he/she should only focus on business logic, not anything else but in real life should we follow that pattern?

The Answer is no, probably you will wonder why I am telling this let's take an example then it will be clear why I want to mean.

Suppose we have a method name ***transferAmount()***  it’s sole responsibility is transact a certain amount from one Account to another.

But when we write above method we write it in following way

Create a Transaction so that if the operation fails we can rollback.

Log the Steps for debugging purpose.

Write business logic.

Handle error condition properly.

Although the responsibility of this method only is transferred amount from One Account to another but it does additional work so clearly a violation of Single Responsibility Principle.

***Spring address this problem and incorporate AOP(Aspect Oriented Programming) in it.***

As we can see to write any method we have to follow the steps mentioned above, while step 3 is the main work and other steps are additional but those are equally important , most important it is span across all modules (Business delegates,Service ,Dao), we call this Cross cutting Concerns, It would be nice if we can separate them from the actual business logic and can manage it centrally so we don’t have to write it repetitively for all methods. AOP does exactly that.

**What is AOP?**

Aspect Oriented Programming address ***cross cutting concerns*** which span across multiple points of an application and define them in an aspect which is the unit of modularity in AOP. Where Dependency Injection helps to decouple Object dependency from each,AOP helps decouple cross-cutting concerns from the business logic and dynamically apply the method, AOP can think of triggers in programming languages, Based on a condition it applies on the certain area of your application and performs the cross cutting jobs. Some good example of Cross cutting concerns are logging, auditing, declarative transactions, security, and caching etc.

Like OOPS to understand AOP we have to familiar with some AOP terminologies

**AOP Key Terms**

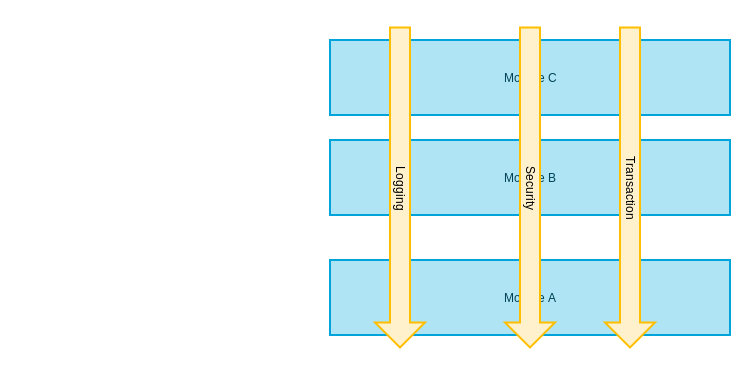
|  |  |
| --- | --- |
| **Key Term** | **Description** |
| **Advice** | Can be thought as **What action to be taken** (cross cutting concerns) before or after or around the method execution. |
| **Join point** | Can be thought of all **possible combination** in an application where AOP aspect. Can be applied. |
| **Pointcut** | Can be thought of **Where an advice should be executed.** This is the subset of join points as it is not necessary to plug in advice in every possible situation. |
| **Aspect** | Can be thought as a **combination of what and where ,it clubbing advice and pointcut** ,by seeing an Aspect we can say where what action it would be taken. |
| **Weaving** | Weaving can be thought of **linking aspects with Object to create an proxy object.** This can be done at compile time, load time, or at runtime. |
| **Target object** | Can be thought as the **Proxy Object which contains business logic and Cross cutting concerns together**. Spring will create it on the fly using Cglib or java Assists. |
| **Introduction** | The **Introduction allows add new methods or attributes** to a class on the fly using Cglib or Java Assists. |

In Spring an Advice can be triggered before an method, after a method or around a method. Spring AOP is not powerful like AspectJ but it will suffice our needs.

**Types of Advice**

|  |  |
| --- | --- |
| **Advice Type** | **Description** |
| **before** | Apply advice or cross cutting concern before a method. |
| **after** | Apply advice or cross cutting concern after a method in spite of outcome may be the method return successfully or throw an exception but advice always be executed. |
| **after-returning** | Apply advice or cross cutting concern after a successful method return. |
| **after-throwing** | Apply advice or cross cutting concern after a method throwing an exception. |
| **around** | Apply advice or cross cutting concern before and after a method. Here you have a privilege to take decision you should call actual method or not. |

**AOP diagram:**



Next chapter we will see how to create advice and apply it.