L’AOP in Spring con AspectJ – Parte II

Cosa imparerai

- Come implementare l’AOP in Spring con AspectJ



paolopreite

Aspect Oriented Programming in Spring con AspectJ

Vediamo come gestire i vari elementi che caratterizzano l'AOP in Spring.

- **AOP proxy**
- **Aspect**
- **Join Point**
- **Advice**
- **Pointcut**
- **Target object**

Aspect Oriented Programming in Spring con AspectJ

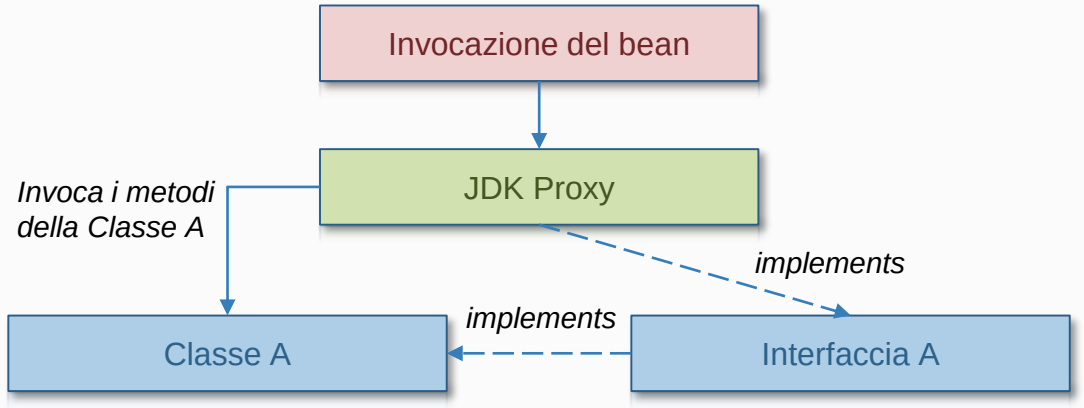
AOP proxy - Quale utilizzare?

Proxy JDK dinamici

L'impostazione predefinita prevede che per i proxy AOP vengano utilizzati i **proxy JDK dinamici**.

Il proxy JDK può generare proxy solo di interfacce.

Per questo, ogni oggetto deve implementare almeno un'interfaccia. L'oggetto creato dal proxy sarà un oggetto che implementa questa interfaccia.



Con il proxy JDK, tutte le decisioni su come gestire una particolare chiamata di metodo vengono gestite in runtime ogni volta che viene richiamato il metodo.

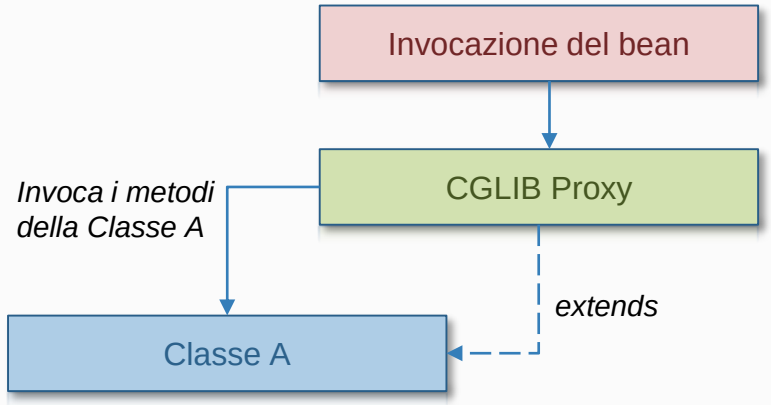
Aspect Oriented Programming in Spring con AspectJ

AOP proxy - Quale utilizzare?

Proxy CGLIB

Quando non è possibile creare un'interfaccia per la nostra classe, è possibile utilizzare un altro tipo di proy: **proxy CGLIB**.

Se un oggetto non implementa un'interfaccia, per impostazione predefinita viene utilizzato CGLIB.



CGLIB genera dinamicamente il bytecode per una nuova classe «on the fly» per ogni proxy, riutilizzando le classi già generate se possibile. Il tipo di proxy generato da CGLIB è una sottoclasse della classe di oggetti di destinazione.

NOTA: *Code Generation Library*, una libreria che consente di manipolare o creare classi dopo la fase di compilazione di un programma.

Aspect Oriented Programming in Spring con AspectJ

Aspect

Un *Aspect* è un POJO al quale cui aggiungiamo l'annotation **@Aspect**.

Un *Aspect* può avere metodi e attributi, come qualsiasi altra classe. Possono anche contenere *Pointcut* e *Advice*.

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

@Aspect
public class LogAsppect {

}
```

L'Aspect deve essere configurato come bean (via XML o annotation).

```
@Bean
public LogAspect getLogAspect() {
    return new LogAspect();
}
```

oppure

```
<bean class="it.test.aop.LogAspect" />
```

Aspect Oriented Programming in Spring con AspectJ

Advice

Un *Advice* è un metodo della classe *Aspect* che implementa l'operazione che un *Aspect* deve eseguire ogni volta che ad un *Join Point* si verifica una determinata condizione.

Un *Advice* può avere come argomento un oggetto di tipo **org.aspectj.lang.JoinPoint**, un'interfaccia che fornisce un accesso sia allo stato disponibile in un punto di unione sia alle informazioni su di esso.

```
@Aspect
public class LogAspect {

    @Before(value = "...")
    public void logBefore(JoinPoint jp) {
        System.out.println(jp.getSignature().getName());
    }
}
```

Il **value** contiene il *Pointcut*, ovvero l'espressione che definisce le regole con cui associamo l'esecuzione di un *Advice* ad un determinato *Join Point*.

Aspect Oriented Programming in Spring con AspectJ

Advice

Esistono diversi tipi di Advice:

Tipo di Advice	Annotation	Quando viene eseguito
BeforeAdvice	***@Before*** *org.aspectj.lang.annotation.Before*	prima venga eseguito un *Join Point*
AfterReturning	***@AfterReturning*** *org.aspectj.lang.annotation.AfterReturning*	dopo l'esecuzione del *Join Point* e se non sono state generate eccezioni
AfterThrowing	***@AfterThrowing*** *org.aspectj.lang.annotation.AfterThrowing*	dopo l'esecuzione del *Join Point* che ha generato l'eccezione
AfterAdvice	***@After*** *org.aspectj.lang.annotation.After*	dopo l'esecuzione del Join Point, indipendentemente dall'esito
AroundAdvice	***@Around*** *org.aspectj.lang.annotation.Around*	consente di prendere il controllo del *Join Point*, specificando anche quando e se eseguire il *Join Point*

Aspect Oriented Programming in Spring con AspectJ

L'Advice AfterReturning

L'attributo **returning** deve contenere il nome della variabile dell'advice a cui verrà assegnato il valore ritornato dal metodo invocato.

```
@AfterReturning(
        value = "this(it.test.service.ClienteService)",
        returning = "ret")
public void logAfterReturning(JoinPoint jp, Object ret) {
    System.out.println("SONO NEL logAfterReturning");
    System.out.println(jp.getSignature().getName());
    System.out.println(ret.toString());
}
```

L'Advice AfterThrowing

In questo Advice è possibile utilizzare l'attributo **throwing** per indicare di eseguire il metodo solo se è stata lanciata una determinata eccezione. Questa eccezione verrà assegnata alla variabile indicata nell'attributo *throwing*.

```
@AfterThrowing(
        value = "this(it.test.service.ClienteService)",
        throwing = "nex")
public void logAfterThrowing(JoinPoint jp, NumberFormatException nex) {
    System.out.println("SONO NEL logAfterReturning");
    System.out.println(jp.getSignature().getName());
}
```

Aspect Oriented Programming in Spring con AspectJ

Pointcut

Un *Pointcut* è un'espressione che definisce le regole con cui associamo l'esecuzione di un *Advice* ad un determinato *Join Point*.

In pratica indica che al verificarsi di un determinato *Join Point* (ad es. l'esecuzione del metodo *salvaOrdine()* della classe *OrderService*) venga applicato un determinato *Advice* (ad es. un metodo che implementa l'*Advice AfterReturning*).

```
@Aspect
public class LogAspect {

    @Before(value = "execution(* it.test.service.OrdineService.sayHello())")
    public void logBefore(JoinPoint jp) {
        System.out.println("SONO NEL logBefore");
        System.out.println(jp.getSignature().getName());
    }
}
```

Aspect Oriented Programming in Spring con AspectJ

Pointcut

L'espressione si può scrivere in diversi modi:

*Posso scrivere direttamente l'espressione oppure inserirla nell'attributo **value***

```
@Before("execution(* it.test.service.OrdineService.sayHello())")
public void logBefore(JoinPoint jp) {
    System.out.println("SONO NEL logBefore");
    System.out.println(jp.getSignature().getName());
}
```

```
@Aspect
public class LogAspect {

    @Before(value = "execution(* it.test.service.OrdineService.sayHello())")
    public void logBefore(JoinPoint jp) {
        System.out.println("SONO NEL logBefore");
        System.out.println(jp.getSignature().getName());
    }
}
```

```
@AfterReturning(
        pointcut = "this(it.test.service.ClienteService)",
        returning = "ret")
public void logAfterReturning(JoinPoint jp, Object ret) {
    System.out.println("SONO NEL logAfterReturning");
    System.out.println(jp.getSignature().getName());
    System.out.println(ret.toString());
}
```

*Per alcune annotation è possibile usare l'attributo **pointcut** al posto dell'attributo value…*

Aspect Oriented Programming in Spring con AspectJ

Pointcut
Spring AOP supporta i seguenti *AspectJ pointcut designators (PCD)* che è possibile utilizzare nelle espressioni:

- **execution**: utilizzato per invocare un advice quando si verifica un determinato join point (ad es. viene eseguito il metodo indicato nell'espressione).

```
@Before(value = "execution(* it.test.service.OrdineService.sayHello())")
public void logBefore(JoinPoint jp) {
    System.out.println("SONO NEL logBefore");
    System.out.println(jp.getSignature().getName());
}
```

Il metodo logBefore viene invocato prima dell'esecuzione del metodo sayHello() del bean OrdineService

- **within**: utilizzato per invocare un advice quando si verificano i join point di un certo tipo (ad es. viene eseguito un qualunque metodo dei bean che si trovano nel package specificato).

```
@After(value = "within(it.test.service..*)")
public void logAfter(JoinPoint jp) {
    System.out.println("SONO NEL logAfter");
    System.out.println(jp.getSignature().getName());
}
```

Il metodo logAfter viene invocato dopo l'esecuzione di tutti i metodi pubblici dei bean che si trovano nel package it.test.service

Aspect Oriented Programming in Spring con AspectJ

Pointcut

- **this**: utilizzato per invocare un advice quando il proxy bean è un'istanza del tipo specificato.

```
@Before(value = "this(it.test.service.ClienteService)")
public void logBeforeInstance(JoinPoint jp) {
    System.out.println("SONO NEL logBeforeInstance");
    System.out.println(jp.getSignature().getName());
}
```

Il metodo viene invocato prima dell'esecuzione di tutti i metodi pubblici del bean ClienteService

- **target**: utilizzato per invocare un advice quando il bean target è un'istanza del tipo specificato.

Aspect Oriented Programming in Spring con AspectJ

Pointcut

- **args**: utilizzato per invocare un advice quando il join point (in questo caso un metodo) contiene come argomento il tipo specificato.

```
@Before(value = "args(java.lang.Long)")
public void logAfterArgs(JoinPoint jp) {
    System.out.println("SONO NEL logAfterArgs");
    System.out.println(jp.getSignature().getName());
}
```

Il metodo viene invocato dopo l'esecuzione di tutti i metodi pubblici del bean ClienteService che hanno come argomento un Long

- **bean**: utilizzato per invocare un advice quando il bean ha il nome specificato oppure contiene la parola inserita:

```
@Bean(name = "ordineService")
public OrdineService getOrdineService() {
    return new OrdineService(getProdottoService());
}
```

```
@After(value = "bean(ordineService)")
public void logAfterBean(JoinPoint jp) {
    System.out.println("SONO NEL logAfterBean");
    System.out.println(jp.getSignature().getName());
}
```

```
@After(value = "bean(*Service)")
public void logAfterBean(JoinPoint jp) {
    System.out.println("SONO NEL logAfterBean");
    System.out.println(jp.getSignature().getName());
}
```

Aspect Oriented Programming in Spring con AspectJ

Pointcut

Esempi di Poincut sono disponibili nella documentazione ufficiale di Spring.

https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop-pointcuts-examples

Aspect Oriented Programming in Spring con AspectJ

È possibile combinare più espressioni?
Si, utilizzando gli operatori AND (&&) OR (||) e NOT(!).

```
@After(value = "args(java.lang.Long) && !this(it.test.service.ClienteService)")
public void logAfterMixed(JoinPoint jp) {
    System.out.println("SONO NEL logAfterMixed");
    System.out.println(jp.getSignature().getName());
}
```

Di cosa abbiamo parlato in questa lezione

- Come implementare l'AOP in Spring con AspectJ