Spring AOP

Przegląd i wprowadzaniema

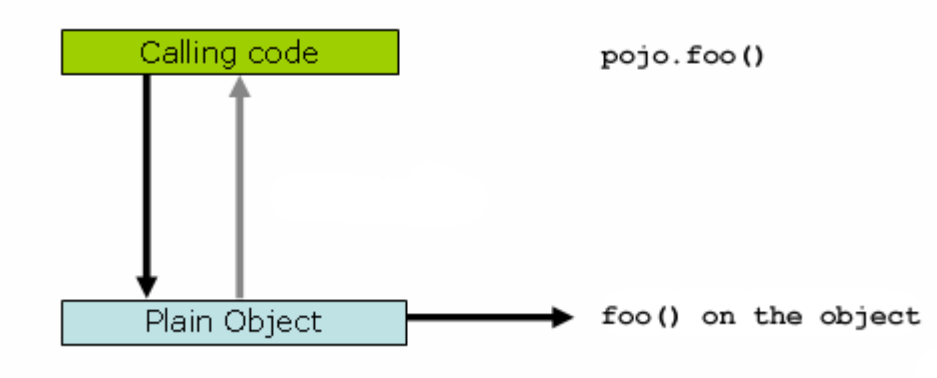
1. Wstęp

Aspect Oriented Programing jest jedną ze składowych Spring Framework. Zastosowano tu podejście bardzo zbliżone do AspectJ. W Spring AOP podobnie jak w AspectJ wyróżniamy:
Aspekty (ang. Aspect) – podstawowa jednostka konsolidująca pewne zachowanie (np. Llgowanie).
Punkty złączeń (ang. Join point) – punkty w których obiekty łączą się z aspektami. W Spring AOP to zawsze wywołania metod.
Rady (ang. Advice) – akcje które są wykonywane w ramach pojedynczego punktu złączeń. Podobie jak w AspectJ rady to
around – aspekt wywołuje otaczaną metodę w środku swojego kodu. Może decydować czy do wywoływania dojdzie (wywołanie metody proceed).
before – przd wykonaniem metody
after returing – po powrocie z metody
after throwing – po wyrzuceniu wyjątku
after - wywoływana zawsze niezależnie od sposobu zakończenia metody
Rady są traktowane jako „przerywacze" które są chainowane dookoła punktu złączeń.
Introductions – Spring AOP pozwala za pomocą aspektów wprowadzanie nowych interfejsów do obiektów.
Target Obejcts – obiekt może być celem jednego lub więcej aspektów. Obiekty są zawsze opakowywane w proxy
AOP proxy – obiekty które są tworzone po tkaniu to zawsze proxy (w Spring AOP to proxy dynamiczna z JDK lub CGLIB proxy)
Tkanie – proces łączenia klas z aspektami. Podobnie jak AspectJ Spring AOP pozwala na tkanie:
w czasie ładowania
w czasie kompilacji
lub w runtime
Spring AOP jest zaimplementowany w czystej Javie oraz nie potrzebuje kontrolować hierarchię classloaderów – co daje możliwość używania go w każdym kontenerze aplikacyjnym czy środowisku uruchmieniowym. Radami można otaczać tylko metody, lecz w obecnym stylu programowania w Javwie każde liczące się pole w klasie ma setter i getter więc nie jest to duże ograniczenie.
Głównym założeniem Spring AOP jest integracaja ze mechanizmem IoC dostarczanym przez Springa. Spring AOP nie ma być kompletną implementacją AOP.
W Springu Aspecty są przeważnie deklarowane przy pomocy normalnej definicji beanów, istnieje jednak możliwość zastosowania autoproxing'u o czym będzie później).
Spring AOP jest stworzony do rozwiązywania powszechnych problemów a aplikacjach J2EE.

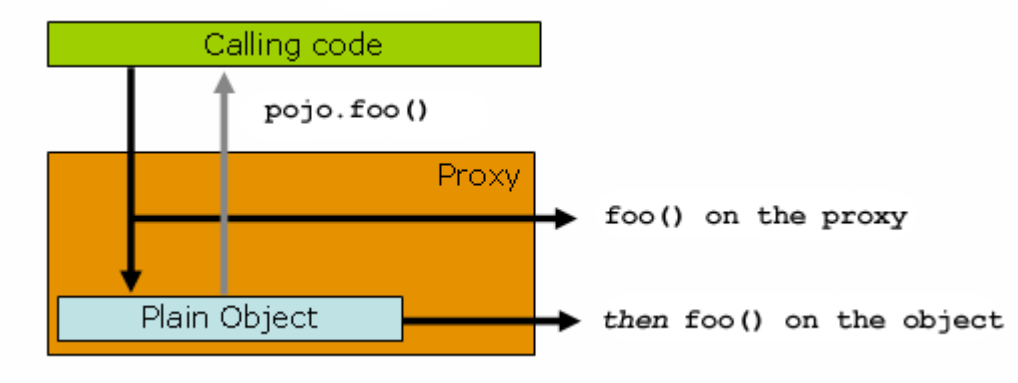
2. AOP Proxies

Spring AOP jest proxy-based, tz że tkanie aspektów opiera się na opakowywaniu obiektów w proxy, które opakowuje wywołania metod, lub dodaje nowe interfejsy.

Normalne wywołanie metody:



Wywołanie metody w proxy:



Spring AOP domyślnie wykorzystuje proxy dynamiczne JS2E. Niestety nie nadają się one do opakowywania klas, dlatego w takich przypadkach Sping AOP wykorzystuje CGLIB proxies. Można oczywiście wymusić stosowanie CGLIB.

3. Praca z Spring AOP – adnotacje @AspecjJ

Jednym ze sposobów pracy z Spring AOP są adnotacje @AspectJ. Spring korzysta tylko z adnotacji więc środowisko jest dalej „czystym" środowiskiem Javy. Nie jest wymagany specjalny classloader, modyfikacje bytecode'u czy specjalna kompilacja. Spring gdy zidentyfikuje że bean jest oznaczony adnotacjami @AspectJ – automatyczne dodaje do niego odpowiednie aspekty.

Żeby włączyć obsługe adnotacji w konfiguracji dodajemy linijkę:

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

lub gdy kożystamy z DTD:

```
<bean
class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAuto
ProxyCreator" />
```

Oczywiscie musimy dodać do classpath'u `aspectjweaver.jar` oraz `aspectjrt.jar`.

3.1 Deklaracja aspektu

Po napisaniu klasy i oznaczeniu jej adnotacją @Aspect zostanie automatycznie wykryty przez Springa. Musimy dodać standardową deklaracje beanów dla aspektu:

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
   <!-- configure properties of aspect here as normal -->
</bean>
```

Przykładowa klasa:

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

@Aspect
public class NotVeryUsefulAspect {

}
```

Aspekty są normalnymi klasami. W Spring AOP aspekty **nie mogą być celem rad innych aspektów**, co jest sporym ograniczeniem.

3.2 Punkty złączeń

Do deklaracji punktów złączeń wykorzystujemy adnotacje @Pointcut, którą oznaczamy żądaną metodę (w Spring AOP punktami złączeń mogą być tylko metody).

Przykład:

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}
```

Pełna specyfikacja @Pointcut jest dostępna pod adresem http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

Do specyfikacji celu punktu złączeń możemy stosować operatory *, &&, ||, !, przykład:

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

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

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

4. Rady

4.1 Before

Metodą najmniejszego zaskoczenia korzystamy z @Before

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

@Aspect
public class BeforeExample {
```

```
  @Before("execution(* com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

4.2 After

Adnotacja @AfterXX gdzie XX to

- returnig – po zwróceniu wartości bez wyjątku
- throwing – po rzuceniu wyjątku przez metodę
- finally – zawsze po wykonaniu metody

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

@Aspect
public class AfterXXExample {

  @AfterXX(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    returning="retVal")
  public void doXX(Object retVal) {
    // ...
  }

}
```

@AfterThrowing – przechwytywanie wyjątku:

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

@Aspect
public class AfterThrowingExample {

  @AfterThrowing(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    throwing="ex")
  public void doRecoveryActions(DataAccessException ex) {
    // ...
  }

}
```

Nazwy parametru metody oraz wartość parametru ex muszą być jednakowe!

4.3 Around

Spring zaleca korzystania z najmniejszej możliwego „działa” - kiedy wystarczy before korzystaj z before!

Metoda adnotowana jako @Around powinna przyjmować parametr typu `ProceedingJoinPoint`. Sgdy chcemy wywołać metodę wywołujemy proced() na `ProceedingJoinPoint`. Proceed przyjmuje Object[] - jako listę parametrów do wywołania.

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

```
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
   public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable
{
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
  }

}
```

4.4 Dostęp do Punktu Złączenia

Każda metoda może mieć dostęp do aktualnego punktu złączeń. Musi tylko jako pierwszy parametr przyjmować typ `ProceedingJoinPoint` (patrz around).

4.5 Parametry rad

Do rad można przekazywać parametry. Przykład:

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
```

```
        "args(account,..)")
public void validateAccount(Account account) {
  // ...
}
```

4.6 Wprowadzenia (ang. Introductions)

Spring AOP pozawala na zastosowanie aspektów do implementowania interfejsów – tz możemy oznaczyć obiekt danym aspektem który doda implementacje interfejsu do obiektu.

Przykład:

```
@Aspect
```

```
public class UsageTracking {

  @DeclareParents(value="com.xzy.myapp.service.*+",
                  defaultImpl=DefaultUsageTracked.class)
  public static UsageTracked mixin;
```

```
  @Before("com.xyz.myapp.SystemArchitecture.businessService() &&" +
          "this(usageTracked)")
  public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
  }

}
```

Duży przykład:

@Aspect

```
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
      this.order = order;
   }

   @Around("com.xyz.myapp.SystemArchitecture.businessService()")
     public  Object  doConcurrentOperation(ProceedingJoinPoint  pjp)  throws
Throwable {
      int numAttempts = 0;
      PessimisticLockingFailureException lockFailureException;
      do {
         numAttempts++;
         try {
            return pjp.proceed();
         }
         catch(PessimisticLockingFailureException ex) {
            lockFailureException = ex;
         }
      }
      while(numAttempts <= this.maxRetries);
      throw lockFailureException;
   }

}
```

5. Schema-based definiowanie aspektów

Aspekty mogą generowane także kożystając z plików XML. Do pracy z nimi potrzebujemy zaimortować odpowiednią shema XML'ową:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:aop="http://www.springframework.org/schema/aop"
          xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
```

Wszytkie taki związane z aop zyją w przestrzeni nazw „aop” w tagu <aop:config>

5.1 Deklarowanie aspektu

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
  ...
</bean>
```

5.2 Deklarowanie punktu złączeń

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

Korzystanie z && czy || nie pasuje do XML dlatego możemy kożystać z and, or oraz not.

5.3 Rady

Składnia i semantyka rad jest analogiczna do sposobu deklarowania z adnotacjami @AspectJ

Przykłady:

before:

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

after returing:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

after throwing

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      throwing="dataAccessEx"
      method="doRecoveryActions"/>

    ...

</aop:aspect>
```

after

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
      pointcut-ref="dataAccessOperation"
      method="doReleaseLock"/>

    ...

</aop:aspect>
```

around:

```
<aop:aspect id="aroundExample" ref="aBean">

```

```
    <aop:around
      pointcut-ref="businessService"
      method="doBasicProfiling"/>

    ...

</aop:aspect>
```

oraz aspekt

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
}
```

6. Który styl wybrać?

Pierwszym kryterium jest wersja Javy z którą pracujemy. Jeżeli nie korzystasz z Javy 1.5+ to musisz zdecydować się na XML.

XML jest lepszy gdy konfiguracja aspektów może się często zmieniać, tj. Chcesz zmieniać konfigurację aspektów bez rekompilowania kodu, lub gdy zamierzasz wykorzystywać aspekty do np. profilowania. W XML oddzielasz konfigurację aspektów od ich kodu co może mieć swoje dobre i złe strony.

XML łamie zasadę DRY (poprzez rozdzielenie konfiguracji od implementacji, korzystając @AspectJ wszystkie informację trzymamy w jednym miejscu). XML jest też bardziej ograniczony (w XML aspekty muszą być singletonami, oraz nie ma możliwości łączenia nazwanych punktów złączeń).

Jeżeli uznamy że Spring AOP nie jest wystarczający do naszych potrzeb, przy przejściu korzystając z @AspectJ nie musimy przepisywać deklaracji aspektów.

Ogólnie zalecane jest korzystanie z adnotacji.

7. AspectJ w Springu

Spring przychodzi z małą biblioteką AspectJ którą możemy włączyć do naszegoprojektu.

7.1 Configurable

Spring pozwala przy wykorzystaniu AspectJ na konigurwanie isteniejących obiektów na podstawie protoypu.

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class Account {
   // ...
}
```

oraz w konfiguracji

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
  <property name="fundsTransferService" ref="fundsTransferService"/>
</bean>
```

spowoduje że do każdej instancji klasy Account zostanie wstrzyknięty prametr fundsTransferService wskazujący na beana „fundsTransferService”.

Spring daje tam tranzakcyjność w aplikacji, która oparta jest na AspectJ (celem tego było umożliwienie korzystania z niej poza IoC)

7. 2 Tkanie AspectJ w Springu

W tym miejscu zostaje złamana zasada Spring AOP – nie jest to już „czysta” Java. Żeby korzystać z tkania AspectJ musimy ustawić javaAgenta na tkcza z AspectJ

```
-javaagent:path/to/spring-agent.jar
```

Przykład:

```
package foo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
```

```
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    public void methodsToBeProfiled(){}
}
```

Musimy także stworzyć META-INF/aop.xml

```
<!DOCTYPE aspectj PUBLIC
                                                    "-//AspectJ//DTD//EN"
"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>

    <weaver>

        <!-- only weave classes in our application-specific packages -->
        <include within="foo.*"/>

    </weaver>

    <aspects>

        <!-- weave in just this aspect -->
        <aspect name="foo.ProfilingAspect"/>

    </aspects>
```

Teraz konfiguracja Springa:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <!-- a service object; we will be profiling it's methods -->
    <bean id="entitlementCalculationService"
          class="foo.StubEntitlementCalculationService"/>

    <!-- this switches on the load-time weaving -->
    <context:load-time-weaver/>

</beans>
```

teraz możemy uruchomić naszą klasę

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {

                                    ApplicationContext     ctx     =     new
ClassPathXmlApplicationContext("beans.xml", Main.class);

        EntitlementCalculationService entitlementCalculationService
                                        =    (EntitlementCalculationService)
ctx.getBean("entitlementCalculationService");

        // the profiling aspect is 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

oraz polecenie

```
java -javaagent:C:/projects/foo/lib/global/spring-agent.jar foo.Main
```

6. Dynamiczne tworzenie Proxies

7. Spring + AspectJ

8. Przykłady

Linki

1. specyfikacja http://static.springsource.org/spring/docs/2.5.5/reference/aop.html
2. opis API http://static.springsource.org/spring/docs/2.0.8/reference/aop-api.html