GeekBrains

Разработка интернет-магазина на Spring Framework

Spring Expression Language. AOP

Spring Expression Language (SpEL), AOP.

Оглавление

Spring Expression Language (SpEL)

SpEL — мощный язык выражений, который позволяет манипулировать графом объекта[1] во время выполнения программы (**runtime**). Например, можно обратиться к свойству инстанса объекта или вложенного объекта, вызвать метод, инстанцировать объект определенного типа, то есть динамически выполнить Java-выражение в пределах корневого объекта или связанных с ним. Синтаксис SpEL похож на Unified EL, но предоставляет дополнительные возможности: вызов методов и базовую функциональность строковой шаблонизации. Несмотря на существование аналогичных языков: OGNL, MVEL и JBoss EL, — SpEL был создан для предоставления сообществу Spring единого хорошо поддерживаемого языка выражений для всех продуктов. SpEL нужен для эффективного использования Thymeleaf.

Применение

SpEL используется:

1. В .xml-файлах конфигурации Spring:

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() *
100.0 }"/>
</bean>
```

2. В шаблонизаторах — например, Thymeleaf:

```
<p th:text="${param.q[0]}" th:unless="${param.q == null}/>
```

3. В коде Java:

```
parser.parseExpression("'Hello World'.concat('!!!')").getValue();
```

```
@Service
public class GoogleGeoCoderService {
    @Value("${google.key}")
    private final String googleApiKey;
```

Функциональность SpEL

- литералы;
- булевы операторы и операторы сравнения;

[1] Граф объекта — это совокупность инстанса объекта со всеми иными экземплярами объектов, на которые он имеет ссылки. Если диаграмма классов описывает отношения между типами, например тип «Заказ» компонует в себе тип «Заказчика», то граф объекта описывает связь между инстансами объектов: заказ включает в себя инстанс заказчика (который включает инстанс «электронная почта»), инстанс дата_заказа, агрегированные инстансы строк заказа и так далее

- регулярные выражения;
- операции с классами;
- доступ к свойствам, массивам, спискам, картам;
- вызов методов;
- реляционные операторы;
- присвоение;
- вызов конструкторов;
- ссылки на бины;
- операции с массивами;
- встроенные списки;
- встроенные карты;
- тернарный оператор;
- переменные;
- определенные пользователем проекции;
- проекция коллекции;
- выбор коллекции;
- шаблонные выражения.

Подключение

SpEL входит в состав модуля:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-expression</artifactId>
</dependency>
```

А модуль входит как зависимость сюда:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
</dependency>
```

Поэтому мы автоматически получаем функционал SpEL в любом Spring-проекте. Spring Boot также включает ее автоматически в недрах стартера **spring-boot-starter**.

Вычисление выражений с использованием Spring

Простейшие примеры

Основные классы и интерфейсы для использования SpEL расположены в пакете **org.springframework.expression**.

За разбор строки выражения отвечает интерфейс **ExpressionParser**, а **Expression** — за вычисление ранее определенного выражения (очень похоже на **Regexp**, паттерн «Интерпретатор»).

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'");
String message = (String) exp.getValue();
```

В этом примере строка выражения представляет собой просто строковый литерал, заключенный в одинарные кавычки.

Есть два исключения, которые могут быть выброшены при вызове **parser.parseExpression** и **exp.getValue** соответственно: **ParseException** и **EvaluationException**.

Метод **getValue** может принимать в качестве аргумента класс, к которому будет приведено вычисленное значение выражения.

SpEL поддерживает вызов метода, доступ к свойствам и вызов конструктора. Например:

```
parser.parseExpression("'Hello World'.concat('!!!')").getValue();
// → Hello World!!!
```

```
int length =
parser.parseExpression("new String('777').length()").getValue(int.class);
// → length = 3
```

Язык выражений SpEL старается следовать синтаксису Java, но, как мы увидим позднее, имеет и отличия, отражающие его целевое применение.

Обратите внимание на использование дженерика **public <T> T getValue(Class<T> desiredResultType)** во втором примере. Это устраняет необходимость приводить значения выражения к желаемому типу результата. Исключение **EvaluationException** будет выброшено, если значение не может быть присвоено типу **T** или преобразовано с использованием зарегистрированного конвертера типов.

Пример обращения к свойствам класса:

```
// классический POJO, стандартные геттеры и сеттеры опущены
public class User {
    private String name;

    public User(String name) {
        this.name = name;
    }
    ...
}
```

```
        User user = new User("John");
        parser.parseExpression("'name").getValue(user);    //  → John
```

И даже так:

```
        parser.parseExpression("name == 'John'").
               getValue(user, Boolean.class);    // → true
```

Интерфейс EvaluationContext

Интерфейс **EvaluationContext** применяется при вычислении выражения для поиска свойств, методов, полей и для преобразования типов. Встроенная реализация **StandardEvaluationContext** использует отражение для управления объектом, при этом она применяет кеширование для повышения производительности.

```
class Simple {
    public List<Boolean> booleanList = new ArrayList<>();
}

Simple simple = new Simple();
simple.booleanList.add(true);

StandardEvaluationContext simpleContext = new
                                          StandardEvaluationContext(simple);
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false");

Boolean b = simple.booleanList.get(0);    // → b = false
```

Здесь **false** передается как строка, но SpEL понимает, что требуется преобразование к **boolean**. Кроме того, видно, что SpEL адресует элементы упорядоченного списка как массив.

Можно настроить парсер выражений SpEL с помощью объекта конфигурации парсера (**org.springframework.expression.spel.SpelParserConfiguration**). Он управляет поведением некоторых компонентов выражения. Например, если выполняется индексирование в массив или коллекцию и элемент с указанным индексом имеет значение **null**, можно автоматически создать элемент. Это полезно при использовании выражений, составленных из цепочки ссылок на свойства.

При работе с выражениями обычно подразумевается большая гибкость в процессе вычисления значений в ущерб производительности. Чтобы ее увеличить, используется компилятор выражений. Он на лету генерирует настоящий Java-класс, поведение которого соответствует вычисляемому

выражению. Поскольку само выражение нетипизируемо, компилятор исходит из результатов его интерпретации и компиляции. Если взять, к примеру, выражение **someArray[0].someProperty.someOtherProperty < 0.1**, которое предполагает доступ к элементу массива, разыменование свойства и числовую операцию, показатели производительности могут существенно отличаться.

По умолчанию компилятор не включен. Чтобы его включить, есть два способа: аргумент в конструкторе **SpelParserConfiguration** или системное свойство **spring.expression.compiler.mode**. Существуют 3 режима компиляции выражений:

- **OFF** — компилятор отключен. Режим по умолчанию.
- **IMMEDIATE** — компиляция выражения происходит как можно скорее, обычно после первой интерпретации.
- **MIXED** — сначала выражение вычисляется интерпретатором, но после вычислений происходит переключение на режим компиляции, в котором выражение компилируется.

Поддержка выражений для определения бинов

Выражения SpEL могут использоваться в XML-конфигурации или в составе аннотаций при определении бинов. В обоих случаях синтаксис для определения выражения имеет такой вид:

```
#{<строка выражения>}
```

Примеры использования в XML-конфигурации

Инициализируем свойство результатом вычисления выражения — случайного числа:

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
      <property name="randomNumber" value="#{T(java.lang.Math).random() * 100.0
}"/>
</bean>
```

Ссылаемся на другой бин, определенный ранее:

```
<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">
    <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>
</bean>
```

Используем предопределенную переменную **systemProperties**:

```
<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
       <property name="defaultLocale" value="#{ systemProperties['user.region']
}"/>
</bean>
```

Annotation-based конфигурация // @Value

```
    public class SomeClass {
        // значение поля
        @Value("#{ systemProperties['user.region'] }")
        private String defaultLocale;

        // эквивалент, но в сеттере
        @Value("#{ systemProperties['user.region'] }")
        public void setDefaultLocale(String defaultLocale) {
            this.defaultLocale = defaultLocale;
        }

        // в методе
        @Autowired
        public void configure(@Value("#{ systemProperties['user.region'] }")
                              String defaultLocale) {
            this.defaultLocale = defaultLocale;
        }
    }
```

Прочие языковые конструкции SpEL

Встроенные списки

Списки могут быть представлены в выражении с использованием {...} нотации:

```
List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);
List listOfLists =
    (List) parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);
```

Ассоциативные массивы // Maps

Описываются с использованием нотации **{key:value}**:

```
Map map = (Map)
parser.parseExpression("{name:'Nikola',dob:'10-July-1856'}").getValue(context);
Map mapOfMaps = (Map) parser.parseExpression(
  "{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").
   getValue(context);
```

Операторы сравнения

Примеры выражений:

```
boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);
boolean trueValue = parser.parseExpression
                              ("2 > -5.0").getValue(Boolean.class);
boolean trueValue = parser.parseExpression
                              ("'black' < 'block'").getValue(Boolean.class);
```

Особый случай **null** — любое число всегда больше **null**:

```
parser.parseExpression("0 > null").getValue(Boolean.class);     // → true
parser.parseExpression("0 < null").getValue(Boolean.class);     // -> false
```

Примитивные типы немедленно помещаются в Boxed-тип, поэтому:

```
        parser.parseExpression("1 instanceof T(int)").
               getValue(Boolean.class);                    // →  false
        parser.parseExpression("1 instanceof T(Integer)").
               getValue(Boolean.class);                    // →  true
```

Регулярные выражения доступны «из коробки»:

```
parser.parseExpression(" '5.00' matches '\^-?\\d+(\\.\\d{2})?$'").
                                    getValue(Boolean.class);   // → true
```

Переменные

На переменные можно ссылаться в выражении с использованием синтаксиса **#variableName**. Переменные инициируются с помощью метода **StandardEvaluationContext.setVariable(...)**.

```
       User user = new User("Пупкин") ;
        StandardEvaluationContext context = new
                                  StandardEvaluationContext(user);
        context.setVariable("newName", "Батарейкин");
        parser.parseExpression("name = #newName").getValue(context);

        System.out.println(user.getName());     // --> "Батарейкин"
```

#this and #root

Переменная **#this** всегда относится к текущему объекту вычисления. Переменная **#root** всегда определена и относится к корневому контекстному объекту.

Получить список простых чисел, превышающих 10, из предварительно сформированного списка:

```
List<Integer> primes = new ArrayList<Integer>();
primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

List<Integer> primesGreaterThanTen =
               (List<Integer>) parser.parseExpression("#primes.?[#this>10]").
               getValue(context);
```

Расширение функциями

Функциональность SpEL можно расширить, зарегистрировав определенные пользователем функции, которые можно вызвать в строке выражения. Они регистрируются с помощью метода **registerFunction(String name, Method m)**:

```
public abstract class StringUtils {
    public static String reverseString(String input) {
        StringBuilder backwards = new StringBuilder();
        for (int i = 0; i < input.length(); i++)
            backwards.append(input.charAt(input.length() - 1 - i));
        }
        return backwards.toString();
    }
}

StandardEvaluationContext context = new StandardEvaluationContext();
context.registerFunction("reverseString",
StringUtils.class.getDeclaredMethod("reverseString",
                                    new Class[] { String.class }));
parser.parseExpression("#reverseString('hello')").
                                getValue(context, String.class);   // → olleh
```

Тернарный оператор

```
parser.parseExpression("true ? 'trueExp' :
                        'falseExp'").getValue(String.class);   // → trueExp
```

Элвис-оператор ?:

Более короткая запись проверки на **null**:

```
String displayName = name != null ? name : "Unknown";
parser.parseExpression("name?:'Unknown'").getValue(String.class);
```

Может использоваться, чтобы применить значения по умолчанию:

```
@Value("#{systemProperties['ssh.port'] ?: 22}")
```

Поиск в коллекции

Следующий код выберет из исходной коллекции все **Inventor**, обладающие свойством **Nationality == 'Serbian'**:

```
class Inventor {
    String name;
    String nationality;
    ...
}
class Society {
    List<Inventors> members;
    …
}

StandardEvaluationContext context = new StandardEvaluationContext(society);

List<Inventor> list = (List<Inventor>) parser
        .parseExpression("Members.?[Nationality ==
                          'Serbian']").getValue(context);
```

Еще доступен выбор первого **^[…]** и последнего **$[…]** объектов.

Проецирование коллекций

Предположим, что у нас есть перечень изобретателей, но нам нужен список их национальностей:

```
List nationalities = (List)parser.parseExpression("Members.![Nationality]");
```

Шаблоны выражений

Шаблоны выражений позволяют смешивать обычный текст с одним или несколькими блоками выражений. Каждый блок разделен символами префикса и суффикса. Общепринято использовать **#{...}** в качестве разделителей. Например:

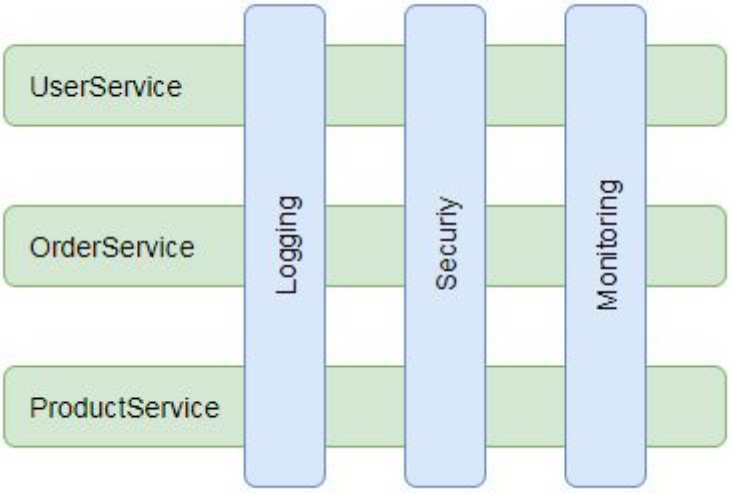
```
String randomPhrase = parser.parseExpression(
        "random number is #{T(java.lang.Math).random()}",
        new TemplateParserContext()).getValue(String.class);
                                        // → "random number is 0.7038186..."

public class TemplateParserContext implements ParserContext {
    public String getExpressionPrefix() {
        return "#{";
    }

    public String getExpressionSuffix() {
        return "}";
    }

    public boolean isTemplate() {
        return true;
    }
}
```

Аспектно-ориентированное программирование (AOP)

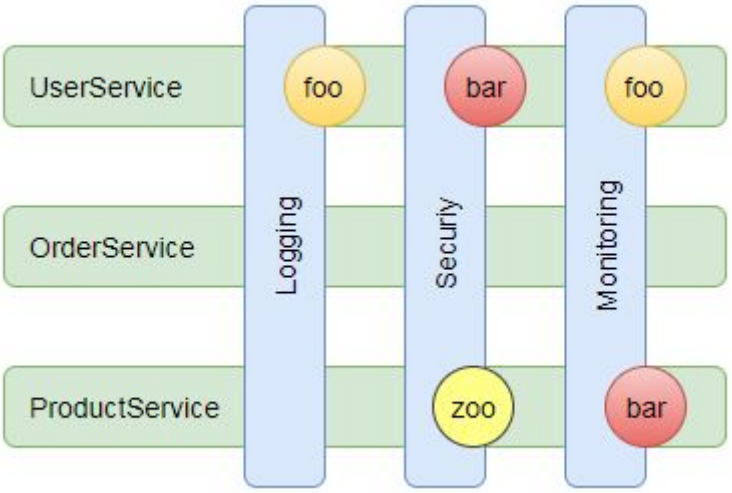
Аспектно-ориентированное программирование (АОП) **дополняет** объектно-ориентированное (ООП), предоставляя еще один способ мышления о структуре программы. Ключевым элементом модульности в ООП является класс, тогда как в АОП это аспект. АОП **не является** частью ООП, а предлагает новые возможности и позволяет элегантно решать проблемы, возникающие в узких местах классического ООП.

Рассмотрим пример: есть набор не связанных сервисов (например, **UserService**, **OrderService**, **ProductService**).

Надо реализовать функциональность, не имеющую ничего общего с этим набором сервисов, но оказывающую воздействие на них одинаково или, возможно, по-разному — в зависимости от конкретного сервиса. Например, логирование, управление доступом или мониторинг.



В общем виде это действие является суперпозицией сервисов и добавляемой функциональности. Причем информацией о том, на каком пересечении применить то или иное действие или не применять его вовсе, не обладают ни сервисы, ни добавляемая функциональность.



Таким образом в системе появляется третий компонент, который:

- «знает», где и какое именно действие применить;
- «знает» эти действия, то есть содержит так или иначе их код;
- элементы, над которыми он стоит, могут и не догадываться о его существовании.

Этот компонент и является аспектом в АОП.

Основные понятия

Аспект (Aspect) — модуль или класс, реализующий сквозную функциональность. Аспект изменяет поведение остального кода, применяя «совет» в «точках соединения», определенных некоторым «срезом».

Срез (Pointcut) — набор «точек соединения». Срез определяет, подходит ли данная точка соединения к данному «совету».

Точка соединения (Join-Point) — точка в выполняемой программе, где следует применить «совет».

Совет (Advice) — средство оформления кода, которое должно быть вызвано из точки соединения. Совет может быть выполнен до, после или вместо точки соединения.

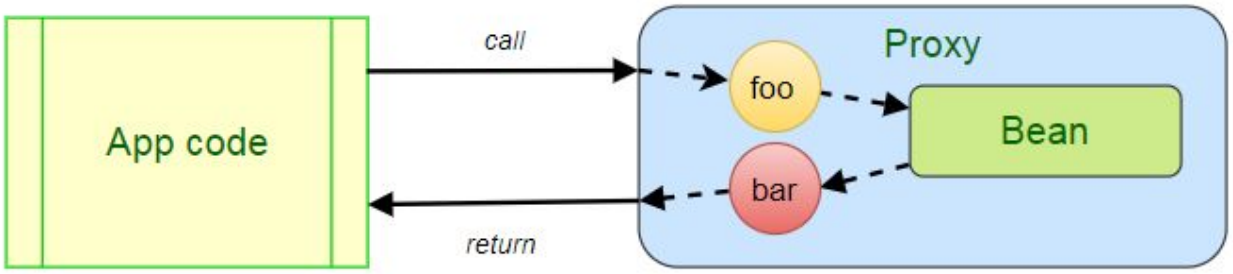
Внедрение (introduction) — изменение структуры класса и/или изменение иерархии наследования для добавления функциональности аспекта в инородный код.

Таким образом, **Аспект,** обладая информацией (получаемой от **советов**), разрывает выполнение программного кода аспектируемых классов в **точке соединения** и **внедряет** туда известную ему функциональность (программный код).

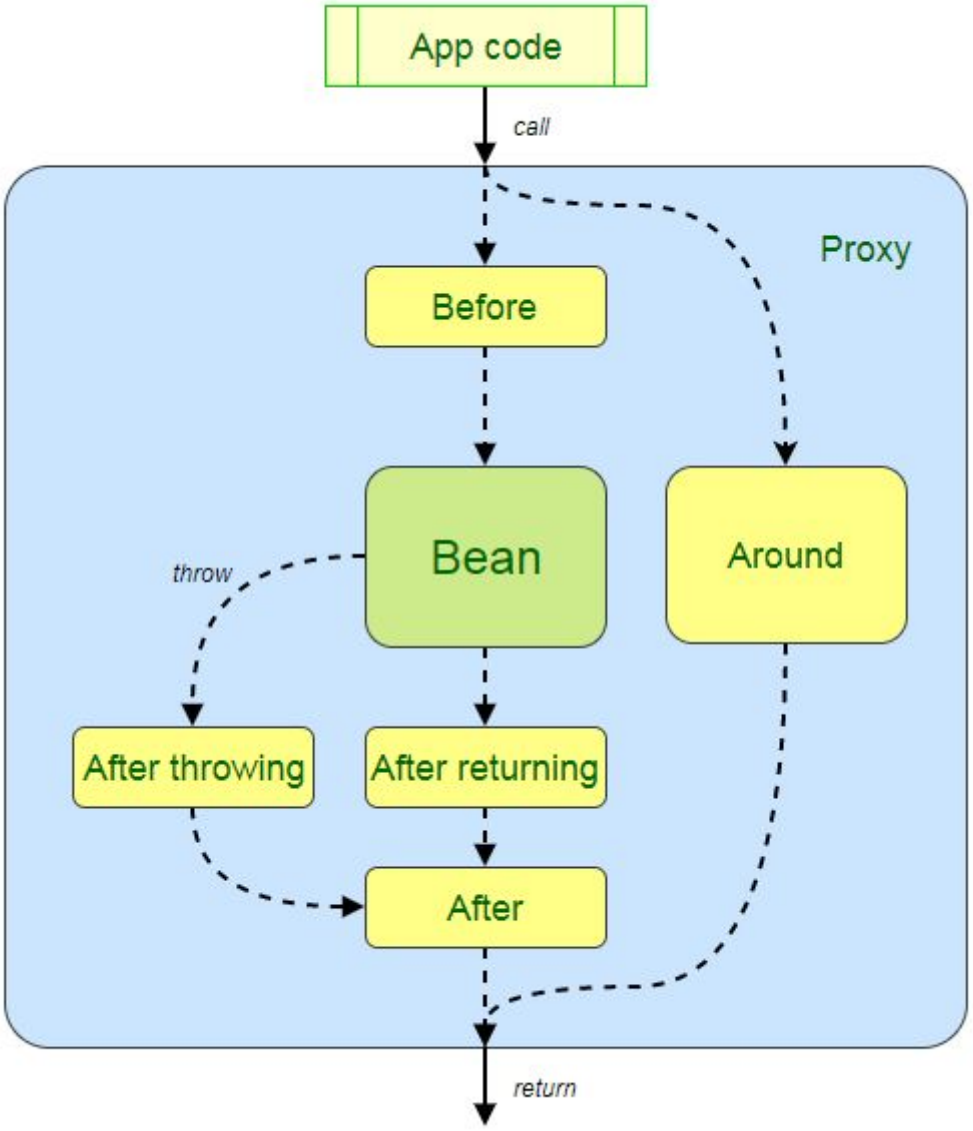
Spring AOP

«Из коробки» Java не поддерживает АОП. Для этого существуют сторонние библиотеки, например AspectJ. Spring применяет собственную реализацию АОП — Spring AOP, которая использует стиль описания AspectJ, но не реализует все ее возможности.

Для внедрения кода Spring AOP применяет проксирование бинов, находящихся в IoC-контейнере. Это избавляет как от использования специальных загрузчиков классов (**classloaders**), так и от посткомпиляции или работы со специальным компилятором, что в целом соответствует «легковесной» направленности Spring.



Фреймворк создает прокси-классы для тех бинов, на которые ссылаются определенные конфигурацией Pointcut'ы. Ограничение такого подхода — невозможность создать аспект для другого аспекта.



Spring AOP поддерживает советы (advice) по выполнению кода до метода бина, после возврата из него, после выброса исключения, после последних двух и вместо выполнения метода бина.

Пример

Рассмотрим тестовый пример. Создадим приложение, которое логирует обращения ко всем собственным сервисам.

Spring AOP подключается добавлением зависимости:

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>LATEST</version>
</dependency>
```

Для вариации **Spring Boot** это добавляется стартером:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Spring AOP поддерживает как конфигурирование, основанное на аннотациях, так и XML-конфигурирование.

Включение автопроксирования бинов:

```
@Configuration
@EnableAspectJAutoProxy
```

Или аналогичный способ:

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

Сервис:

```
package com.example.aop.service;

@Service
public class SomeService {
    private String name = SomeService.class.getCanonicalName(); // что-то
полезное
    public String getName() {
        return name;
    }
}
```

Обратите внимание на пакет **com.example.aop.services**. В дальнейшем мы будем ссылаться на него в определении среза (pointcut) аспекта.

Аспект:

```
@Aspect
@Component
public class LogAspect {
    // определяем срез по всем методам бинов из пакета com.example.aop.service
    @Pointcut("execution(* com.example.aop.service..*.*(..))")
    private void getName() {
    }

    // определяем совет (Advice) "ПЕРЕД" выполнением кода бина (класса)
    @Before("getName()")
    public void logBefore(JoinPoint joinPoint) {
        // выводим в консоль информацию о текущей точке соединения
        System.out.println(joinPoint);
    }
}
```

При обращении к методам сервисов, например, в контроллере:

```
@RestController
public class SomeController {
    @RequestMapping("/")
    public String getGreeting() {
        return someService.getName();
    }
    // ...
}
```

На консоль будут выдаваться сообщения, определенные в методе аспекта **logBefore(JoinPoint joinPoint)**:

```
execution(String com.example.aop.service.SomeService.getName())
```

Обобщим материал.

1. Есть сервисы, которые не знают о существовании сторонней функциональности (в данном случае — логирования), их код не подвергался дополнительным модификациям (относительно типичного сервиса).
2. Имеется сторонняя функциональность (логирование), которая готова работать с любым объектом.
3. Определенный нами аспект, используя возможности библиотеки Spring AOP, меняет поведение исходных объектов (сервисов), заставляя выполняться при определенных нами условиях отдельный код, не пересекающийся с кодом исходных объектов.

Следует отметить, что функциональность Spring Security основана на применении АОП.

Например, аннотация метода **@PreAuthorize(...)** гарантирует, что только пользователь, обладающий определенными правами, имеет доступ к аннотируемому методу. Spring Security будет использовать во время запуска (AOP) pointcut для запуска совета **(advice) Before** на методах, помеченных данной аннотацией. И будет выбрасывать **AccessDeniedException**, если ограничения безопасности, указанные в параметрах аннотации, не будут выполнены.

Практическое задание

1. Доработать шаблоны предыдущего урока, используя SpEL-выражения.
2. * Реализовать в приложении логирование работы пользователей с корзиной через Spring AOP.

Дополнительные материалы

1. Руководство по Spring. АОП в Spring Framework.
2. Spring изнутри. Этапы инициализации контекста.

3. Защита приложений с помощью АОП.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальная документация.
2. Aspect Oriented Programming with Spring.
3. Аспектно-ориентированное программирование (АОП) — Википедия.