**Spring AOP — все про него знают и слышали, но часто ли его используют на самом деле?**

1. **Введение в AOP**

Что же такое AOP?

АОП — аспектно-ориентированное программирование — это парадигма программирования, направленная внедрение сквозной-кросс функциональности в существующее приложение. Для этого к уже исходному коду добавляется дополнительного поведение, без изменений в изначальном коде.

Для тех, кто не знаком с АОП обозначим основные понятия, для тех, кто уже знаком с аспектами – напомних.

Итак, основные понятия АОП:

- аспект – это класс, содержащий реализацию сквозной функциональности. Аспект изменяет поведение остального кода, применяя совет(advice) в точках соединения(joinpoint), определенных некоторым срезом(poincut).

Точка соединения (joinpoint) — это конкретное место в программе, где может быть выполнен код аспекта. Это может быть вызов метода, создание объекта или выполнение блока кода.

Срез (pointcut) — это выражение, которое определяет набор точек соединения, к которым будет применяться аспект. С помощью срезов можно указать, какие точки соединения будут обрабатываться аспектом.

Совет(advice) — это код, который выполняется в определённые моменты времени, указанные в точке соединения.

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

**Цель (target)** — объект, к которому применяются советы.

**Вплетение (weaving)** — это процесс связывания аспектов с другими объектами для создания прокси-объектов. Это можно сделать во время компиляции, загрузки или во время выполнения. Тут надо еще покопать.

Для определения времени выполнения существуют следующие виды советов(Advice):

**Перед (Before)** — советы данного типа запускаются перед выполнением целевых методов. При использовании аспектов в виде классов мы используем аннотацию @Before, чтобы пометить тип совета перед логикой исполнения метода-совета. Отлично подходит для реализации логирования.

**После (After)** — советы, которые выполняются после завершения выполнения целевых методов, как в обычных случаях, так и при бросании исключения. Таким образом выброс исключения не изменит поведения аспекта, и логика, заложенная в аспект все равно будет выполнена. Метод совет помечается аннотацией @After.

**После возврата (After Returning)** — данные советы выполняются только в том случае, когда целевой метод отрабатывает нормально, без ошибок.  
Помечается аннотацией @AfterReturning.

После выброса (After Throwing) — данный вид советов предназначен для тех случаев, когда целевой исполняемый метод, то есть точка соединения выдает исключение. Мы можем использовать этот совет для некой обработки неудачного выполнения (к примеру, для отката всей транзакции или логирования с необходимым уровнем трассировки).  
Используется аннотация @AfterThrowing.

**Вокруг (Around)** — один из самых важных видов советов, который окружает метод, то есть — точку соединения, с помощью которого мы можем, к примеру, выбрать, выполнять данный метод точки соединения или нет, а также изменять возвращаемый объект.

Можно написать код совета, который будет выполняться до и после выполнения метода точки соединения.  
В обязанности around advice входит вызов метода точки соединения и возвращение значений, если метод что-то возвращает. То есть в этом совете можно попросту сымитировать работу целевого метода, не вызывая его, и в качестве результата вернуть что-то свое. Если данное поведение никак не задокументировано, это может вызывать большие проблемы при отладке.

Объявляется аннотацией @Around для создания советов, оборачивающих точку соединения.

1. Аспекты, применяемые нами каждый день.

2.1 Controller Advice - это специальный тип аспекта, который применяется ко всем контроллерам в приложении или к определённым контроллерам. Он позволяет обрабатывать общие задачи, такие как обработка исключений или настройка модели или возвращаемого ResponseEntity.

Используя @ControllerAdvice мы не вносим никакого кода в сам контроллер, однако теперь можем перехватывать ошибки, выброшенных методом контроллера, и возвращать клиенту стандартизированное сообщение.

* 1. Использование аннотации @Transactional в Springframework.

Наглядный пример использования аспектной парадигмы – аннотация @Transactional. Выполнение перехватыается при помощи AnnotationTransactionalAspect, который позволяет управлять и создавать транзакции на основе метаданных аннотации, и откатывать их, в случае возникновения не перехваченного исключения, таким образом отделяя данную логику от бизнес методов.

2.3. @PreAuthorize и @PostAuthorize:

• Эти аннотации используются в Spring Security для выражения условий доступа до или после выполнения метода, соответственно. Они также создают прокси для проверки условий безопасности. Применяется advice @Around.

Вскользь упомянем, что механика аспектов применяется в обработке аннотаций @Async, @Cacheable, @CacheEvict, @CachePut и др.

1. Как мы используем кастомные аспекты в проме.

Как и почему мы решили использовать аспекты в проме?

В первую очередь у нас было желание разгрузить классы контроллеров от бесконечных log.info о вызовах метода. Код замусоривался. Следующим нашим желанием было централизованно логировать выбрасываемые исключения, чтобы также не замусоривать код бесконечными log.error.

Было решено в качестве срезов использовать аннотации с RetentionPolicy Type и Method, чтобы гибко маркировать классы.

Используя advice @After и Before и кастомной аннотации мы можем логировать начало и конец выполнения требуемого метода, или всех методов класса. Благодаря JoinPoint мы можем вытащить всю неообходимую нам мета информацию – сигнатуру метода, имя целевого класса, исполняющего метод, имя прокси класса, имя метода. Эту информацию мы собираем в логи, и теперь наши контроллеры просты и чисты от мусора.

Второй срез, примененный нами, также являлся кастомной аннотацией-маркером, позволяющей выполнять advice @AfterThrowing. Этот аспект позволяет нам логировать исключения.

Аннтоации помимо выполнения функции среза, также осуществляют и функции документирования, оставляя информацию о том, что в исполняемом классе или методе присутствует скрытая логика.

В планах на будущее – применение advice @Around для создания аспекта-бенчмарка, для измерения выполнения отдельных методов в целях тестирования и проверки гипотез.

Также можно реализовать вызов логики интеграции с внешними службами через аспекты. Наша команда пока это не применяла, но в предыдущей компании были использованы аспекты для интеграции. Перехватывались определенные методы, как только пользователь вызывал один из них, информация передавалась во внешнюю службу.

Конечно, нагляднее было бы вызывать интеграцию непосредственно в коде исполняемого метода, но таким образом мы добивались отделения логики интеграции от бизнес-логики. Подобные аспекты требуют определенной конвенции именования методов в команде.

1. Подводные камни.

Что же с производительностью? Дополнительное проксирование безусловно накладывает некоторые издержки при сборке, но при применении данных аспектов они оказались на уровне погрешности. Но надо иметь ввиду, что в рамках одного микросервиса больших накладных расходов и не получится ввиду небольшого(относительно) числа логируемых классов. В рантайме просадок производительности также пока замечено не было.

Также не стоит забывать, что CGlib, который используется в Spring для проксирования классов не может работать с финальными классами, так как он создает прокси через механизм наследования, и при попытке скомпилировать код, где финальный класс пытается применить на себе точку среза, вы получите исключение BeanCreationException. Таким образом, вы дожны применять аспекты с осторожностью, если в вашем проекте много финальных классов.

Желательно(обязательно) документировать использование сквозной логики, во избежание проблем с отладкой.

1. Выводы

Итак, мы рассмотрели основные применения аспектов в повседневной жизни, которые сопровождают нас неявно, также мы обсудили, как создавать собственные аспекты и как их применять в своем коде.

Применением аспектов мы немного разгрузили наш код и избавились от излишнего дублирования. Использовать их или нет – решение всегда остается за командой, там было интересно пойти именно по этому пути, но возможно кого-то оттолкнет неявное поведение программы и сложности с проксированием.