**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.

Напомним основные понятия:

**Аспект** – это класс, содержащий реализацию сквозной функциональности.

**Точка соединения (joinpoint)** — это конкретное место в программе, где может быть выполнен код аспекта.

**Срез (pointcut)** — это выражение, которое определяет набор точек соединения, к которым будет применяться аспект.

**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.

Рассмотрим различные варианты weathing.

Преимущества weathing **во время компиляции(AspectJ)**:

• Более высокая производительность, так как аспекты уже встроены в скомпилированный код.

• Возможность применения аспектов ко всем классам, включая финальные классы и статические методы.

• Недостатки:

• Необходимость использования специального компилятора или плагина для интеграции с IDE.

• Усложнение процесса сборки проекта.

• Преимущества на **уровне загрузчика**:

• Позволяет применять аспекты без изменения процесса сборки.

• Можно управлять аспектами на уровне загрузки классов.

• Недостатки:

• Меньшая производительность по сравнению с компиляцией, так как внедрение происходит во время загрузки классов.

**Runtime Weaving:**

Spring AOP использует этот подход, который основан на динамическом создании прокси-объектов при помощи SGlib. В Spring AOP аспекты внедряются во время выполнения приложения, когда создаются прокси для целевых объектов.

• Преимущества:

• Простота настройки и интеграции с приложением.

• Не требует изменения процесса сборки или использования специальных инструментов.

• Поддержка конфигурации через XML или аннотации.

• Недостатки:

• Ограниченная поддержка (не может проксировать финальные классы и статические методы).

• Небольшая потеря производительности из-за создания прокси-объектов.

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

**@Before** — советы данного типа запускаются перед выполнением целевых методов.

**@After** — советы, которые выполняются после завершения выполнения целевых методов, как в обычных случаях, так и при бросании исключения.

**@AfterReturning** — данные советы выполняются только в том случае, когда целевой метод отрабатывает нормально, без ошибок.

**@AfterThrowing** — данный вид советов предназначен для тех случаев, когда целевой исполняемый метод, то есть точка соединения выдает исключение.

**@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.

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.**

Для своих проектов мы выбрали Spring Aop и проксирование CGlib. Проксировать финальные классы мы не планировали, и нам требовалось максимально простой вариант для наших задач.

Как и почему мы решили использовать аспекты в проме?

**3.1 Логирование выполнения методов**

В первую очередь у нас было желание разгрузить классы контроллеров от бесконечных log.info о вызовах метода. Код выглядел некрасиво и громоздко. 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 необходимую 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.

* 1. **Логирование исключений**

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

Как только в аннотированном методе выбрасывается исключение, аспект перехватывает все сигнатуры метода, записывает отформатированное сообщение в лог.

Примечательно, что если целевой метод вызывает приватный метод того же класса, и в нем выбрасывается исключение, перехват все равно происходит, и не требуется применять self injection.

Применение логирования исключений на уровне аспектов разгрузили наши классы, больше не требуется в сообщении указывать какие-либо параметры, все необходимые аргументы будут вписаны в аспекте, и получилось стандартизировать большинство сообщений, выведя их в Enum.

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

**3.3 Бенчмарк**

Следующий наш кейс применения аспектов – применение advice измеряющего время выполнения отдельных методов в целях тестирования и проверки гипотез.

Часто возникали вопросы, насколько быстро отработает та или иная реализация метода, и возникла потребность получить данные метрики.

Для того, чтобы аспект выполнялся только на определенных стендах и не был виден на проме, к аннотации добавили метаинформацию о запрещенных активных профилях.

Для обработки данного кейса мы применяли совет @Around – перехватывая метод, в аспекте мы замеряли длительность выполнения метода, также выводя результаты в лог.

Так как наша система не является высоконагруженной, применение других способов бенчмарка да данном этапе мы посчитали излишним, тем более методов, подвергающихся такой оценке не так уж и много.

**3.4 Перехват и подмена аргументов метода для проверки возможности исполнения**

В данный момент функция еще тестируется в тестовой среде, но рассказать о ней мне показалось интересным.

Периодически по разным запросам мы генерируем списки задач, которые распределяются по разным подразделениям. В зависимости от источника запроса требовалось генерировать задачи только для определенных подразделений. Список подразделений передавался в виде обычного листа.

Методы могли вызываться из разных мест и приходилось выполнять проверки в разных местах.

Было решено применить конвенцию именования методов, и внедрить аспект, который перехватывал метод, на основе передаваемых аргументов определял необходимые разрешения, и подменял в сигнатуре метода список разрешений. Логика определения разрешений размещалась в одном месте и легко поддавалась изменениям

Таким образом мы упростили тестирование, вынесли логику определения разрешений в сквозную функциональность.

**3.5 Применение аспектов для интеграции с внешними службами**

На текущем проекте данный аспект мы пока не применяем, но на предыдущем проекте были использованы аспекты для интеграции с внешними службами. Перехватывались определенные методы, используя advice @AfterReturning, как только пользователь вызывал один из методов, результат выполнения метода перехватывался, и информация из возвращаемого объекта передавалась во внешнюю службу.

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

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 должны 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.