







Spring AOP

Урок 8



Что будет на уроке сегодня

-  Spring AOP
-  Аспекты в Spring AOP
-  After, AfterReturning, AfterThrowing, Around Advice
-  Join Points
-  Introductions
-  Interceptors



Spring AOP

Spring AOP (Aspect-Oriented Programming) — это часть фреймворка Spring, позволяющая вводить аспекты в программный код, что упрощает модульность приложения путем разделения кросс-куттинговой функциональности, такой как логирование, безопасность и транзакции.





Spring AOP

Spring AOP помогает разработчикам писать более чистый и модульный код. Вы можете сосредоточиться на основной логике вашего приложения, не беспокоясь о дополнительных функциях, таких как логирование или безопасность.





Аспекты в Spring AOP

Аспект в Spring AOP — это модуль, который определяет “перекрестные” или “сквозные” задачи, такие как логирование, безопасность или транзакции.

Аспекты работают таким образом, что они “внедряются” или “вплетаются” в ваш код в определенные места, которые вы указываете.





Логирование методов

```
1 @Aspect
2 @Component
3 public class LoggingAspect {
4
5     @Before("execution(* com.example.service.*(..))")
6     public void logBeforeMethodCall(JoinPoint joinPoint) {
7         System.out.println("Метод " + joinPoint.getSignature().getName() + " был вызван");
8     }
9 }
```



Измерение времени выполнения

```
1 @Aspect
2 @Component
3 public class PerformanceAspect {
4
5     @Around("execution(* com.example.service.*(..))")
6     public Object measureMethodExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
7         long start = System.currentTimeMillis();
8
9         Object result = joinPoint.proceed();
10
11         long elapsedTime = System.currentTimeMillis() - start;
12         System.out.println("Метод " + joinPoint.getSignature().getName() + " выполнен за " + elapsedTime + " миллисекунд");
13
14         return result;
15     }
16 }
```



Изменение возвращаемого значения

```
1 @Aspect
2 @Component
3 public class ChangeReturnValueAspect {
4
5     @AfterReturning(pointcut = "execution(* com.example.service.getName(..))", returning = "result")
6     public void changeName(JoinPoint joinPoint, String result) {
7         result = "Измененное имя";
8     }
9 }
```




Компоненты аспектов. Advices

Before Advice

After Returning Advice

After Throwing Advice

After (or After Finally) Advice



Компоненты аспектов. **Introductions**

Introduction (или “Mixin”) позволяет добавлять новые методы или свойства в существующие бины.





Компоненты аспектов. Around-Advices

Around-Advice — объединяет в себе все остальные типы advices, так как позволяет вам вмешиваться в вызов метода до его выполнения, после него и даже изменять возвращаемое значение или кидать исключение вместо целевого метода.





Before Advice

```
1 @Component
2 @Aspect
3 public class LoggingAspect {
4 }
```

```
6 @Before("execution(* com.example.service.UserService.viewProtectedPage( .. ))")
7 public void logBeforeAccess() {
8     System.out.println("Попытка доступа к защищенной странице!");
9 }
```



After Advice

```
1 @Component
2 @Aspect
3 public class TaskCompletionAspect {
4 }
5
6 @After("execution(* com.example.service.TaskService.completeTask(..))")
7 public void logAfterTaskCompletion() {
8     System.out.println("Задание успешно завершено!");
9 }
```



AfterReturning Advice

```
1 @Component
2 @Aspect
3 public class MessageAspect {
4 }
5
6 @AfterReturning(pointcut = "execution(* com.example.service.MessageService.getUserMessage(..))", returning = "message")
7 public void logUserMessage(String message) {
8     System.out.println("Возвращаемое сообщение: " + message);
9 }
```



AfterThrowing Advice

```
1 @Component
2 @Aspect
3 public class PaymentAspect {
4 }
5
6 @AfterThrowing(pointcut = "execution(* com.example.service.PaymentService.processPayment(..))", throwing = "exception")
7 public void logPaymentError(Exception exception) {
8     System.out.println("Произошла ошибка при обработке платежа: " + exception.getMessage());
9 }
```



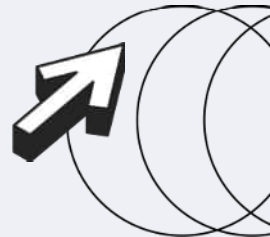
Around Advice

```
1 @Component
2 @Aspect
3 public class PerformanceAspect {
4 }
5
6 @Around("execution(* com.example.service.HeavyDutyService.performTask(..))")
7 public Object measureExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
8     long startTime = System.currentTimeMillis();
9
10    Object result = joinPoint.proceed(); // вызов целевого метода
11
12    long endTime = System.currentTimeMillis();
13
14    System.out.println("Метод " + joinPoint.getSignature() + " выполнялся " + (endTime - startTime) + " миллисекунд.");
15
16    return result; // возвращаем результат выполнения целевого метода
17 }
```




Join Point

Join Point в Spring AOP — это место в программе, где аспект может быть применен. Это может быть при вызове метода, при обработке исключения, при инициализации объекта и так далее.





Join Point

```
1 @Aspect
2 @Component
3 public class MyAspect {
4
5     @Before("execution(* com.example.service.MyService.*(..))")
6     public void beforeAnyMethodInMyService() {
7         System.out.println("Вызван метод в MyService!");
8     }
9 }
```



Join Point

```
1 @Aspect
2 @Component
3 public class MyAspect {
4
5     @Before("execution(* com.example.service.*(..))")
6     public void beforeAnyMethodInServicePackage() {
7         System.out.println("Вызван метод в пакете service!");
8     }
9 }
```



Join Point

```
1 @Aspect
2 @Component
3 public class MyAspect {
4
5     @Before("execution(* com.example.service.MyService.specificMethod(..))")
6     public void beforeSpecificMethod() {
7         System.out.println("Вызван конкретный метод specificMethod!");
8     }
9 }
```



Pointcut Expressions

1.

```
1 @Pointcut("execution(* *.*(..))")  
2 private void selectAllMethods() {}
```

2.

```
1 @Pointcut("execution(* *.set*(..))")  
2 private void selectAllSetters() {}
```

3.

```
1 @Pointcut("execution(* *.find*(String))")  
2 private void selectAllStringFinders() {}
```



Introductions

```
1 @Aspect
2 public class FlyerIntroduction {
3
4     @DeclareParents(value = "com.example.Car", defaultImpl = FlyingImpl.class)
5     public static Flyer flyer;
6 }
7
8 public class FlyingImpl implements Flyer {
9     @Override
10    public void fly() {
11        System.out.println("Car is now flying!");
12    }
13 }
```



Introductions

Introductions могут быть полезны в ряде сценариев:

1. Переиспользование кода
2. Постепенное внедрение новых возможностей
3. Работа с сторонними библиотеками



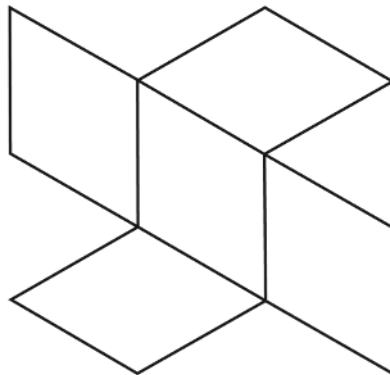
AfterThrowing

```
1 @Aspect
2 public class ErrorHandlingAspect {
3
4     @AfterThrowing(pointcut = "execution(* com.example.*.*(..))", throwing = "error")
5     public void handleAllErrors(Exception error) {
6         System.out.println("An error occurred: " + error.getMessage());
7         // Здесь можно, например, отправлять уведомление или логировать ошибку
8     }
9 }
```




Управление исключениями на уровне аспектов

1. Логирование.
2. Уведомления.
3. Трансформация исключений.
4. Откат транзакций.





Управление исключениями на уровне аспектов

```
1 @Aspect
2 public class ExceptionTransformingAspect {
3
4     @AfterThrowing(pointcut = "execution(* com.example.service.*.*(..))", throwing = "ex")
5     public void transformExceptions(Exception ex) throws CustomException {
6         if (ex instanceof SpecificException) {
7             throw new CustomException("Custom message");
8         }
9     }
10 }
```



Как управлять порядком advice в Spring AOP?

```
1 @Aspect
2 @Order(1)
3 public class SecurityAspect {
4     // код аспекта безопасности
5 }
6
7 @Aspect
8 @Order(2)
9 public class LoggingAspect {
10     // код аспекта логирования
11 }
```



JDK Dynamic Proxy vs. CGLIB proxy

```
7 public class SimpleService$$EnhancerBySpringCGLIB extends SimpleService {
8     @Override
9     public void doSomething() {
10         // логика аспекта (например, логирование)
11         super.doSomething(); // вызов оригинального метода
12     }
13 }
```



@Transactional аннотация и её настройка

```
1 @Service
2 public class BookService {
3
4     @Autowired
5     private BookRepository bookRepository;
6
7     @Transactional
8     public void addTwoBooks(Book book1, Book book2) {
9         bookRepository.save(book1);
10        bookRepository.save(book2);
11    }
12 }
```

```
1 @Transactional(rollbackFor = CustomException.class)
2 public void someTransactionalMethod() {
3     // ваш код
4 }
```



Interceptors

```
1 public class LoggingInterceptor extends HandlerInterceptorAdapter {
2
3     @Override
4     public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
5         System.out.println("Request URL: " + request.getRequestURL().toString());
6         return true;
7     }
8
9     @Override
10    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws
    Exception {
11        System.out.println("After handling the request");
12    }
13
14    @Override
15    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws
    Exception {
16        System.out.println("Request completed");
17    }
18 }
```



Interceptors

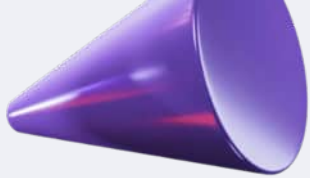
```
1 @Configuration
2 public class AppConfig implements WebMvcConfigurer {
3
4     @Override
5     public void addInterceptors(InterceptorRegistry registry) {
6         registry.addInterceptor(new LoggingInterceptor());
7     }
8 }
```



Как AOP влияет на разработку

1. Повторное использование кода и модульность .
2. Улучшение читаемости и поддержки кода
3. Гибкая настройка
4. Обработка исключений
5. Безупречные транзакции
6. Понимание проксирования





Спасибо за внимание

