JAVA WEB

Unit 7: Spring Boot AoP



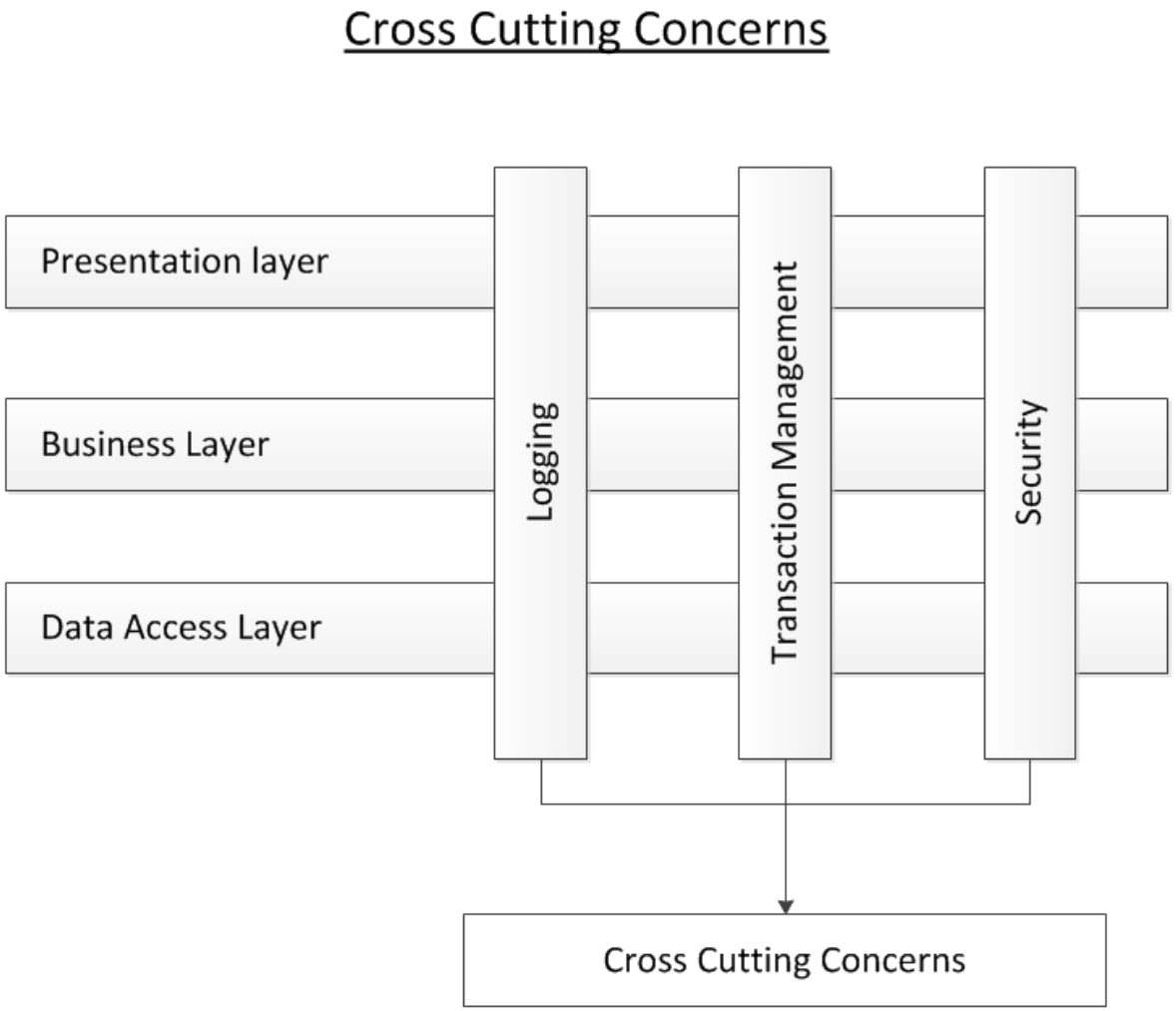
Mục lục

AOP (Aspect Oriented Programming)

AOP (Aspect Oriented Programming) là 1 kỹ thuật lập trình bổ sung cho lập trình hướng đối tượng (OOP), AoP tạo ra 1 cách suy nghĩ khác của lập trình cấu trúc. Đối tượng của OOP là class, còn đối tượng của AOP là *aspect*.

Có một số *aspect*-khía cạnh cần được quan tâm trong quá trình phát triển ứng dụng: Các khía cạnh này có thể trải rộng trên các lớp khác nhau của ứng dụng. Ví dụ: ghi nhật ký (log), quản lý giao dịch, xử lý ngoại lệ, giám sát hiệu suất, v.v. Những khía cạnh này chính là mối quan tâm xuyên suốt (***cross cutting concerns***).

Cross Cutting Concerns



Ví dụ - Cần ghi cho một số method đã có

Giả sử chúng ta có một số methods **đã** hoàn chỉnh, chúng ta muốn chèn log khi method đó được gọi. Theo các logic thông thường thì chúng ta sẽ phải **SỬA CHƯƠNG TRÌNH**:

1) Hoặc là phải sửa code trong method

2) Hoặc **tìm tất cả** những chỗ nào method được gọi, insert log vào trước

=> Mất nhiều thời gian (code+test), trộn lẫn code chức năng + ghi log;

Spring Framework giúp chúng ta cài đặt các khía cạnh này bằng cách cung cấp Spring AoP. Nói một cách đơn giản, Spring AOP sẽ chiếm quyền điều khiển việc thực thi chương trình và đưa vào các tính năng bổ sung gồm: trước, sau hoặc xung quanh việc thực thi một phương thức.

Như vậy Spring **AoP giúp chúng ta chèn log mà không sửa chương trình.**

=> tách biệt code chức năng và phần code ghi log;

AOP (Aspect Oriented Programming)

Chúng ta xem xét ví dụ sau về khía cạnh ghi nhật ký, gồm các yêu cầu:

1. Ghi thông tin nhật ký trước khi thực thi phương thức;
2. Ghi thông tin nhật ký sau khi thực thi phương thức;
3. Ghi lại thời gian mà phương thức cần để hoàn tất quá trình thực thi.

- Ở đây: Ghi nhật ký là một khía cạnh- *Aspect*.

- Việc thực thi phương thức được gọi là Điểm nối - ***Join point***.

- Đoạn code ghi lại nhật ký và thời gian thực hiện phương thức được gọi là các *advices* (chức năng bổ sung).

- Danh sách các methods mà hành vi này yêu cầu được gọi là **Point Cuts**.

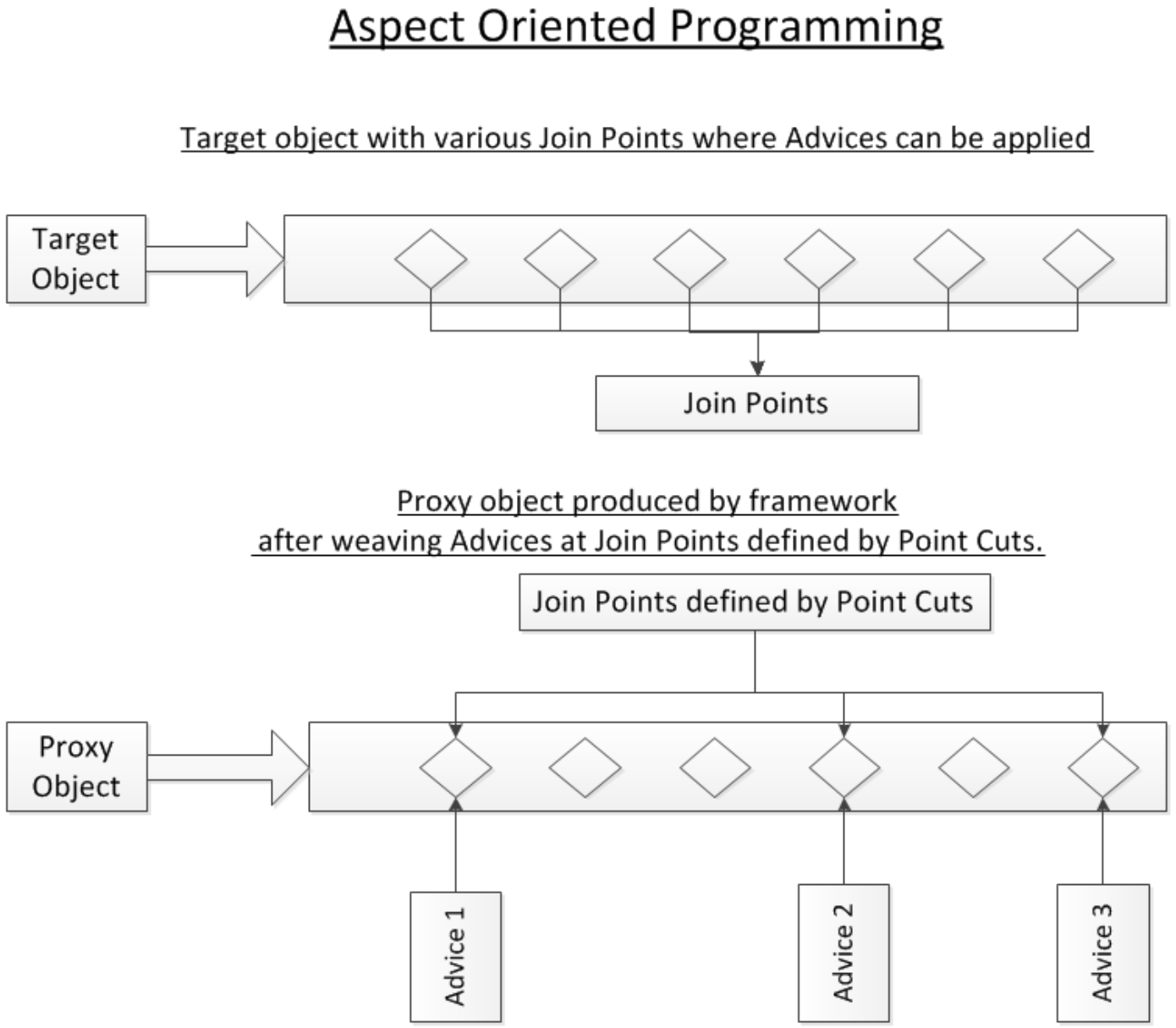
- Và cuối cùng, đối tượng java áp dụng khía cạnh này được gọi là Mục tiêu- *Targets*.

Có một số loại *advices sau đây:*

1. *Before advice* –Thực hiện trước phương thức (method execution)
2. *After returning advice* – Thực hiện nếu phương thực thực thi bình thường (Run if the join point executes normally)
3. *After throwing advice* – Thực hiện nếu phương thức ném ra ngoại lệ.
4. *Around advice* – Thực thi xung quang - run around the join point (method execution)

Như vậy: 1. Ghi thông tin nhật ký trước khi thực thi phương thức (sử dụng Before advice); 2. Ghi thông tin nhật ký sau khi thực thi phương thức (sử dụng After advice); 3. Ghi lại thời gian mà phương thức cần để hoàn tất quá trình thực thi (sử dụng Around advice)

Các thành phần trong AoP



- *Targets*: đối tượng java áp dụng khía cạnh
- Thực thi phương thức của target điểm nối - ***Join point***.
- Danh sách các methods của target mà chức năng bổ sung đc áp dung gọi là **Point Cuts**.
- Đoạn code ghi lại nhật ký và thời gian thực hiện phương thức được gọi là các *advices* (chức năng bổ sung).

Thư viện AoP

Pom.xml

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
        <version>2.4.5</version>
 </dependency>
```

Spring AoP Annotation

Các annotations:

- **@Aspect** khai báo khí cạnh.
- **@Pointcut** khai báo điểm cắt.

- **@Before** thực hiện trước khi gọi phương thức.
- **@After** thực hiện sau khi gọi phương thức và trước khi trả về kết quả.
- **@AfterReturning** thực hiện sau khi gọi phương thức và trước khi trả về kết quả nhưng có thể lấy kết quả trong advice.
- **@Around** thực hiện trước và sau khi gọi phương thức.
- **@AfterThrowing**

Ví dụ

Pointcut: Điểm cắt dùng để khai báo rằng Aspect đó sẽ được gọi khi nào.

Ví dụ: điểm cắt **("execution(* mta.aopDemo.dao..(..))")** áp dụng với tất cả các phương thức trong tất cả các lớp có trong package mta.aopDemo.dao.

Advice: Thiết lập công việc cần phải làm gì khi xảy ra điểm cắt đó, ví dụ:

```
@Before
public void before(JoinPoint joinPoint){
        logger.info(" before called " + joinPoint.toString())
}
```

Ví dụ - Các method đã có (cần chèn log)

Target:

```
@Repository
public class ClassDAO {
    private final Logger logger    = LoggerFactory.getLogger(ClassDAO.class);
    public String callDaoSuccess(){
        logger.info("callDaoSuccess is called");
        return "dao1";
    }
    public String callDaoThrowException(){
        logger.info("DAO is called");
        throw new RuntimeException("");
    }
    public String callMethodTrackTime(){
        logger.info("callDaoSuccess is called");
        return "dao1";
    }
}
```

Định nghĩa các Aspect

```
@Aspect
@Configuration
public class TestServiceAspect {
    private Logger logger = LoggerFactory.getLogger(TestServiceAspect.class);

    @Before("execution(* mta.aopDemo.dao.ClassDAO.callDaoSuccess(..))") //Point Cut
    //Advice
    public void before(JoinPoint joinPoint) {
        logger.info("Aspect: before called " + joinPoint.toString());
        }
    @After("execution(* mta.aopDemo.dao.*.*(..))") //Point Cut
    public void after(JoinPoint joinPoint) {
        logger.info("Advice:  after called " + joinPoint.toString());
        }
    @AfterReturning("execution(* mta.aopDemo.dao.*.*(..))")//Point Cut
    public void afterReturning(JoinPoint joinPoint) {
        logger.info("Advice:  afterReturning called " + joinPoint.toString()); }
    @AfterThrowing("execution(* mta.aopDemo.dao.*.*(..))")//Point Cut
    public void afterThrowing(JoinPoint joinPoint) {
        logger.info("Advice:  afterThrowing called " + joinPoint.toString()); }
    @Around("execution(* mta.aopDemo.dao.*.*(..))")//Point Cut
    public void around(ProceedingJoinPoint joinPoint) throws Throwable {
        Long startTime = System.currentTimeMillis();
        logger.info("Advice: Start Time Taken by {} is {}", joinPoint, startTime);
        joinPoint.proceed();
        Long timeTaken = System.currentTimeMillis() - startTime;
        logger.info("Advice: Time Taken by {} is {}", joinPoint, timeTaken);  }
}
```

```java
//DemoSpringAopApplicationTests.java

@SpringBootTest
class DemoSpringAopApplicationTests {
    @Autowired
    ClassService classService;
    @Test
    void callTestService() {
       classService.callDaoSuccess();
//         try {
//             testService.callDaoFailed();
//         } catch (Exception ex) {
//
//         }

   //     testService.callDaoTrackTime();
```

```java
@Service
public class ClassService {
    private Logger logger = LoggerFactory.getLogger(ClassService.class);
    @Autowired
    ClassDAO classDAO;
    public String callDaoSuccess(){
        logger.info("Test Service callDaoSuccess ");
        return classDAO.callDaoSuccess();
    }
    public String callDaoFailed(){
        logger.info("Test Service callDaoFailed");
        return classDAO.callDaoThrowException();
    }
    public String callDaoTrackTime(){
        logger.info("Test Service callDaoTrackTime");
        return classDAO.callMethodTrackTime();
    }
```

```java
@Aspect
@Configuration
public class TestServiceAspect {
    private Logger logger = LoggerFactory.getLogger(TestServiceAspect.class);

    @Before("execution(* mta.aopDemo.dao.ClassDAO.callDaoSuccess(..))") //Point Cut
    //Advice
    public void before(JoinPoint joinPoint) {
        logger.info("Aspect: before called " + joinPoint.toString());
        }
    @After("execution(* mta.aopDemo.dao.*.*(..))") //Point Cut
    public void after(JoinPoint joinPoint) {
        logger.info("Advice:  after called " + joinPoint.toString());
        }
    @AfterReturning("execution(* mta.aopDemo.dao.*.*(..))")//Point Cut
    public void afterReturning(JoinPoint joinPoint) {
        logger.info("Advice:  afterReturning called " + joinPoint.toString()); }
    @AfterThrowing("execution(* mta.aopDemo.dao.*.*(..))")//Point Cut
    public void afterThrowing(JoinPoint joinPoint) {
        logger.info("Advice:  afterThrowing called " + joinPoint.toString()); }
    @Around("execution(* mta.aopDemo.dao.*.*(..))")//Point Cut
    public void around(ProceedingJoinPoint joinPoint) throws Throwable {
        Long startTime = System.currentTimeMillis();
        logger.info("Advice: Start Time Taken by {} is {}", joinPoint, startTime);
        joinPoint.proceed();
        Long timeTaken = System.currentTimeMillis() - startTime;
        logger.info("Advice: Time Taken by {} is {}", joinPoint, timeTaken);  }
}
```

```java
@Repository
public class ClassDAO {
    private final Logger logger    = LoggerFactory.getLogger(ClassDAO.class);
    public String callDaoSuccess(){
        logger.info("callDaoSuccess is called");
        return "dao1";
    }
    public String callDaoThrowException(){
        logger.info("DAO is called");
        throw new RuntimeException("");
    }
    public String callMethodTrackTime(){
        logger.info("callDaoSuccess is called");
        return "dao1";
    }
}
```

Project: Chapter_SpringAoP