第一章：Spring AOP 总览

小马哥（mercyblitz）

Spring AOP 总览

1. 知识储备：基础、基础，还是基础！
2. AOP 引入：OOP 存在哪些局限性？
3. AOP 常见使用场景
4. AOP 概念：Aspect、Join Point 和 Advice 等术语应该如何理解？
5. Java AOP 设计模式：代理、判断和拦截器模式
6. Java AOP 代理模式（Proxy）：Java 静态代理和动态代理的区别是什么？
7. Java AOP 判断模式（Predicate）：如何筛选 Join Point？
8. Java AOP 拦截器模式（Interceptor）：拦截执行分别代表什么？
9. Spring AOP 功能概述：核心特性、编程模型和使用限制
10. Spring AOP 编程模型：注解驱动、XML 配置驱动和底层 API

Spring AOP 总览

11. Spring AOP 设计目标：Spring AOP 与 AOP 框架之间的关系是竞争还是互补？
12. Spring AOP Advice 类型：Spring AOP 丰富了哪些 AOP Advice 呢？
13. Spring AOP 代理实现：为什么 Spring Framework 选择三种不同 AOP 实现？
14. JDK 动态代理：为什么 Proxy.newProxyInstance 会生成新的字节码？
15. CGLIB 动态代理：为什么 Java 动态代理无法满足 AOP 的需要？
16. AspectJ 代理代理：为什么 Spring 推荐 AspectJ 注解？
17. AspectJ 基础：Aspect、Join Points、Pointcuts 和 Advice 语法和特性
18. AspectJ 注解驱动：注解能完全替代 AspectJ 语言吗？
19. 面试题精选

知识储备

- Java 基础
  - Java ClassLoading
  - Java 动态代理
  - Java 反射
  - 字节码框架：ASM、CGLIB

知识储备

- OOP 概念
  - 封装
  - 继承
  - 多态

知识储备

- GoF23 设计模式
  - 创建模式（Creational patterns）
    - 抽象工厂模式（Abstract factory）
    - 构建器模式（Builder）
    - 工厂方法模式（Factory method）
    - 原型模式（Prototype）
    - 单利模式（Singleton）

知识储备

- GoF23 设计模式
  - 结构模式（Structural patterns）
    - 适配器模式（Adapter）
    - 桥接模式（Bridge）
    - 组合模式（Composite）
    - 装饰器模式（Decorator）
    - 门面模式（Facade）
    - 享元模式（Flyweight）
    - 代理模式（Proxy）

知识储备

- GoF23 设计模式
  - 行为模式（Behavioral patterns）
    - 模板方法模式（Template Method）
    - 中继器模式（Mediator）
    - 责任链模式（Chain of Responsibility）
    - 观察者模式（Observer）
    - 策略模式（Strategy）
    - 命令模式（Command）
    - 状态模式（State）
    - 访问者模式（Visitor）
    - 解释器模式（Interpreter）、迭代器模式（Iterator）、备忘录模式（Memento）

知识储备

- Spring 核心基础
  - Spring IoC 容器
  - Spring Bean 生命周期（Bean Lifecycle）
  - Spring 配置元信息（Configuration Metadata）
  - Spring 事件（Events）
  - Spring 注解（Annotations）

AOP 引入

- Java OOP 存在哪些局限性？
  - 静态化语言：类结构一旦定义，不容易被修改
  - 侵入性扩展：通过继承和组合组织新的类结构

AOP 常见使用场景

- 日志场景
  - 诊断上下文，如：log4j 或 logback 中的 _x0008_MDC
  - 辅助信息，如：方法执行时间

AOP 常见使用场景

- 统计场景
  - 方法调用次数
  - 执行异常次数
  - 数据抽样
  - 数值累加

AOP 常见使用场景

- 安防场景
  - 熔断，如：Netflix Hystrix
  - 限流和降级：如：Alibaba Sentinel
  - 认证和授权，如：Spring Security
  - 监控，如：JMX

AOP 常见使用场景

- 性能场景
  - 缓存，如 Spring Cache
  - 超时控制

AOP 概念

- AOP 定义

  - AspectJ：Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common concerns.

  - Spring：Aspect-oriented Programming (AOP) complements Object-oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns (such as transaction management) that cut across multiple types and objects.

AOP 概念

- Aspect 概念
  - AspectJ：aspect are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations.
  - Spring：A modularization of a concern that cuts across multiple classes.

AOP 概念

- Join point 概念
  - AspectJ: A join point is a well-defined point in the program flow. A pointcut picks out certain join points and values at those points.
  - Spring: A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

AOP 概念

- Pointcut 概念
  - AspectJ：pointcuts pick out certain join points in the program flow.
  - Spring：A predicate that matches join points.

AOP 概念

- Advice 概念

  - AspectJ：So pointcuts pick out join points. But they don't do anything apart from picking out join points. To actually implement crosscutting behavior, we use advice. Advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points).

  - Spring：Action taken by an aspect at a particular join point. Different types of advice include "around", "before" and "after" advice. Many AOP frameworks, including Spring, model an advice as an interceptor and maintain a chain of interceptors around the join point.

AOP 概念

- Introduction 概念

  - AspectJ：Inter-type declarations in AspectJ are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes.

  - Spring：Declaring additional methods or fields on behalf of a type. Spring AOP lets you introduce new interfaces (and a corresponding implementation) to any advised object.

Java AOP 设计模式

- 代理模式：静态和动态代理
- 判断模式：类、方法、注解、参数、异常...
- 拦截模式：前置、后置、返回、异常

Java AOP 代理模式（Proxy）

- Java 静态代理
  - 常用 OOP 继承和组合相结合

- Java 动态代理
  - JDK 动态代理

  - 字节码提升，如 CGLIB

Java AOP 判断模式（Predicate）

- 判断来源
  - 类型（Class）
  - 方法（Method）
  - 注解（Annotation）
  - 参数（Parameter）
  - 异常（Exception）

Java AOP 拦截器模式（Interceptor）

- 拦截类型
  - 前置拦截（Before）
  - 后置拦截（After）
  - 异常拦截（Exception）

Spring AOP 功能概述

- 核心特性
  - 纯 Java 实现、无编译时特殊处理、不修改和控制 ClassLoader
  - 仅支持方法级别的 Join Points
  - 非完整 AOP 实现框架
  - Spring IoC 容器整合
  - AspectJ 注解驱动整合（非竞争关系）

Spring AOP 编程模型

- 注解驱动
  - 实现：Enable 模块驱动，@EnableAspectJAutoProxy
  - 注解：
    - 激活 AspectJ 自动代理：@EnableAspectJAutoProxy
    - Aspect ： @Aspect
    - Pointcut ：@Pointcut
    - Advice ：@Before、@AfterReturning、@AfterThrowing、@After、@Around
    - Introduction ：@DeclareParents

Spring AOP 编程模型

- XML 配置驱动
  - 实现：Spring Extensble XML Authoring
  - XML 元素
    - 激活 AspectJ 自动代理：<aop:aspectj-autoproxy/>
    - 配置：<aop:config/>
    - Aspect ： <aop:aspect/>
    - Pointcut ：<aop:pointcut/>
    - Advice ：<aop:around/>、<aop:before/>、<aop:after-returning/>、<aop:after-throwing/> 和 <aop:after/>
    - Introduction ：<aop:declare-parents/>
    - 代理 Scope ： <aop:scoped-proxy/>

Spring AOP 编程模型

- 底层 API
  - 实现：JDK 动态代理、CGLIB 以及 AspectJ
  - API：
    - 代理：AopProxy
    - 配置：ProxyConfig
    - Join Point：JoinPoint
    - Pointcut ：Pointcut
    - Advice ：Advice、BeforeAdvice、AfterAdvice、AfterReturningAdvice、ThrowsAdvice

Spring AOP 设计目标

- 整体目标

Spring AOP’s approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable). Rather, the aim is to provide a close integration between AOP implementation and Spring IoC, to help solve common problems in enterprise applications.

Spring AOP never strives to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks such as Spring AOP and full-blown frameworks such as AspectJ are valuable and that they are complementary, rather than in competition.Spring seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API. Spring AOP remains backward-compatible.

Spring AOP Advice 类型

- Advice 类型
  - 环绕（Around）
  - 前置（Before）
  - 后置（After）
    - 方法执行
    - finally 执行
  - 异常（Exception）

Spring AOP 代理实现

- JDK 动态代理实现 - 基于接口代理
- CGLIB 动态代理实现 - 基于类代理（字节码提升）
- AspectJ 适配实现

JDK 动态代理

- 为什么 Proxy.newProxyInstance 会生成新的字节码？

CGLIB 动态代理

- 为什么 Java 动态代理无法满足 AOP 的需要？

AspectJ 代理

- 为什么 Spring 推荐 AspectJ 注解?

AspectJ 基础

- AspectJ 语法
  - Aspect
  - Join Points
  - Pointcuts
  - Advice
  - Introduction

AspectJ 注解驱动

- AspectJ 注解
  - 激活 AspectJ 自动代理：@EnableAspectJAutoProxy
  - Aspect ： @Aspect
  - Pointcut ：@Pointcut
  - Advice ：@Before、@AfterReturning、@AfterThrowing、@After、@Around
  - Introduction ：@DeclareParents

面试题精选

- 问：Spring AOP 和 AspectJ AOP 存在哪些区别？

- 答：
  - AspectJ 是 _x0008_AOP 完整实现，Spring AOP 则是部分实现
  - Spring AOP 比 AspectJ 使用更简单
  - Spring AOP 整合 AspectJ 注解与 Spring IoC 容器
  - Spring AOP 仅支持基于代理模式的 AOP
  - Spring AOP 仅支持方法级别的 Pointcuts