本节课内容

spring AOP 常见面试题目

Aop 是什么

与 OOP 对比，面向切面，传统的 OOP 开发中的代码逻辑是自上而下的，而这些过程会产生一些横切性问题，这些横切性的问题和我们的主业务逻辑关系不大，这些横切性问题不会影响到主逻辑实现的，但是会散落到代码的各个部分，难以维护。AOP 是处理一些横切性问题，AOP 的编程思想就是把这些问题和主业务逻辑分开，达到与主业务逻辑解耦的目的。使代码的重用性和开发效率更高。

aop 的应用场景

1. 日志记录
2. 权限验证
3. 效率检查
4. 事务管理
5. exception

springAop 的底层技术

	JDK 动态代理	CGLIB 代理
编译时期的织入还是运行时期的织入?	运行时期织入	运行时期织入
初始化时期织入还是获取对象时期织入?	初始化时期织入	初始化时期织入

springAop 和 AspectJ 的关系

Aop 是一种概念

springAop、AspectJ 都是 Aop 的实现，SpringAop 有自己的语法，但是语法复杂，所以 SpringAop 借助了 AspectJ 的注解，但是底层实现还是自己的

spring AOP 提供两种编程风格

@AspectJ support ---------->利用 aspectj 的注解

Schema-based AOP support ---------->xml aop:config 命名空间

证明:spring,通过源 码分析了,我们可以知道 spring 底层使用的是 JDK 或者 CGLIB 来完成的代理,并且在官网上 spring 给出了 aspectj 的文档,和 springAOP 是不同的

spring Aop 的概念

aspect:一定要给 spring 去管理　抽象　aspectj->类

pointcut:切点表示连接点的集合　---------->　　表

　(我的理解：PointCut 是 JoinPoint 的谓语，这是一个动作，主要是告诉通知连接点在哪里，切点表达式决定 JoinPoint 的数量)

Joinpoint:连接点　　目标对象中的方法 ---------->　　记录

　(我的理解：JoinPoint 是要关注和增强的方法，也就是我们要作用的点)

Weaving :把代理逻辑加入到目标对象上的过程叫做织入

target 目标对象 原始对象

aop Proxy 代理对象　包含了原始对象的代码和增加后的代码的那个对象

advice:通知　　(位置 + logic)

advice 通知类型:

Before 连接点执行之前，但是无法阻止连接点的正常执行，除非该段执行抛出异常

After　连接点正常执行之后，执行过程中正常执行返回退出，非异常退出

After throwing　执行抛出异常的时候

After (finally)　无论连接点是正常退出还是异常退出，都会执行

Around advice: 围绕连接点执行，例如方法调用。这是最有用的切面方式。around 通知可以在方法调用之前和之后执行自定义行为。它还负责选择是继续加入点还是通过返回自己的返回值或抛出异常来快速建议的方法执行。

Proceedingjoinpoint 和 JoinPoint 的区别:

Proceedingjoinpoint 继承了 JoinPoint,proceed()这个是 aop 代理链执行的方法。并扩充实现了 proceed()方法，用于继续执行连接点。JoinPoint 仅能获取相关参数，无法执行连接点。

JoinPoint 的方法

1.java.lang.Object[] getArgs()：获取连接点方法运行时的入参列表；

2.Signature getSignature() ：获取连接点的方法签名对象；

3.java.lang.Object getTarget() ：获取连接点所在的目标对象；

4.java.lang.Object getThis() ：获取代理对象本身；

proceed()有重载,有个带参数的方法,可以修改目标方法的的参数

Introductions

perthis

使用方式如下：

@Aspect("perthis(this(com.chenss.dao.IndexDaoImpl))")

要求：

1. AspectJ 对象的注入类型为 prototype

2. 目标对象也必须是 prototype 的

原因为：只有目标对象是原型模式的，每次 getBean 得到的对象才是不一样的，由此针对每个对象就会产生新的切面对象，才能产生不同的切面结果。

springAop 支持 AspectJ

1、启用@AspectJ 支持

使用 Java Configuration 启用@AspectJ 支持

要使用 Java @Configuration 启用@AspectJ 支持，请添加

@EnableAspectJAutoProxy 注释

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

使用 XML 配置启用@AspectJ 支持

要使用基于 xml 的配置启用@AspectJ 支持，可以使用 aop:aspectj-autoproxy 元素

<aop:aspectj-autoproxy/>

2、声明一个 Aspect

申明一个@Aspect 注释类，并且定义成一个 bean 交给 Spring 管理。

@Component

@Aspect

public class UserAspect {

}

3、申明一个 pointCut

切入点表达式由@Pointcut 注释表示。切入点声明由两部分组成:一个签名包含名称和任何参数，以及一个切入点表达式，该表达式确定我们对哪个方法执行感兴趣。

@Pointcut("execution(* transfer(..))")// 切入点表达式

private void anyOldTransfer() {}// 切入点签名

切入点确定感兴趣的 join points（连接点），从而使我们能够控制何时执行通知。Spring AOP 只支持 Spring bean 的方法执行 join points（连接点），所以您可以将切入点看作是匹配 Spring bean 上方法的执行。

/**

```
 * 申明 Aspect，并且交给 spring 容器管理
 */
@Component
@Aspect
public class UserAspect {
    /**
     * 申明切入点，匹配 UserDao 所有方法调用
     * execution 匹配方法执行连接点
     * within:将匹配限制为特定类型中的连接点
     * args：参数
     * target：目标对象
     * this：代理对象
     */
    @Pointcut("execution(* com.yao.dao.UserDao.*(..))")
    public void pintCut(){
        System.out.println("point cut");
    }
```

4、申明一个 Advice 通知

advice 通知与 pointcut 切入点表达式相关联，并在切入点匹配的方法执行@Before 之前、@After 之后或前后运行。

```
/**
 * 申明 Aspect，并且交给 spring 容器管理
```

```
 */
@Component
@Aspect
public class UserAspect {
    /**
     * 申明切入点，匹配 UserDao 所有方法调用
     * execution 匹配方法执行连接点
     * within:将匹配限制为特定类型中的连接点
     * args：参数
     * target：目标对象
     * this：代理对象
     */
    @Pointcut("execution(* com.yao.dao.UserDao.*(..))")
    public void pintCut(){
        System.out.println("point cut");
    }
    /**
     * 申明 before 通知,在 pintCut 切入点前执行
     * 通知与切入点表达式相关联，
     * 并在切入点匹配的方法执行之前、之后或前后运行。
     * 切入点表达式可以是对指定切入点的简单引用，也可以是在适当位置声明的切入点表达式。
```

```
     */
    @Before("com.yao.aop.UserAspect.pintCut()")
    public void beforeAdvice(){
        System.out.println("before");
    }
}
```

各种连接点 joinPoint 的意义:

1. execution

用于匹配方法执行 join points 连接点，最小粒度方法，在 aop 中主要使用。

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern) throws-pattern?)

这里问号表示当前项可以有也可以没有，其中各项的语义如下

modifiers-pattern：方法的可见性，如 public，protected；

ret-type-pattern：方法的返回值类型，如 int，void 等；

declaring-type-pattern：方法所在类的全路径名，如 com.spring.Aspect；

name-pattern：方法名类型，如 buisinessService()；

param-pattern：方法的参数类型，如 java.lang.String；

throws-pattern：方法抛出的异常类型，如 java.lang.Exception；

example:

@Pointcut("execution(* com.chenss.dao.*.*(..))")//匹配 com.chenss.dao 包下的任意接口和类的任意方法

@Pointcut("execution(public * com.chenss.dao.*.*(..))")//匹配 com.chenss.dao 包下的任意接口和类的 public 方法

@Pointcut("execution(public * com.chenss.dao.*.*())")//匹配 com.chenss.dao 包下的任意接口和类的 public 无方法参数的方法

@Pointcut("execution(* com.chenss.dao.*.*(java.lang.String, ..))")//匹配 com.chenss.dao 包下的任意接口和类的第一个参数为 String 类型的方法

@Pointcut("execution(* com.chenss.dao.*.*(java.lang.String))")//匹配 com.chenss.dao 包下的任意接口和类的只有一个参数，且参数为 String 类型的方法

@Pointcut("execution(* com.chenss.dao.*.*(java.lang.String))")//匹配 com.chenss.dao 包下的任意接口和类的只有一个参数，且参数为 String 类型的方法

@Pointcut("execution(public * *(..))")//匹配任意的 public 方法

@Pointcut("execution(* te*(..))")//匹配任意的以 te 开头的方法

@Pointcut("execution(* com.chenss.dao.IndexDao.*(..))")//匹配 com.chenss.dao.IndexDao 接口中任意的方法

@Pointcut("execution(* com.chenss.dao..*.*(..))")//匹配 com.chenss.dao 包及其子包中任意的方法

关于这个表达式的详细写法,可以脑补也可以参考官网很容易的,可以作为一个看 spring 官网文档的入门,打破你害怕看官方文档的心理,其实你会发觉官方文档也是很容易的

https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#aop-pointcuts-examples

由于 Spring 切面粒度最小是达到方法级别，而 execution 表达式可以用于明确指定方法返回类型，类名，方法名和参数名等与方法相关的信息，并且在 Spring 中，大部分需要使用 AOP 的业务场景也只需要达到方法级别即可，因而 execution 表达式的使用是最为广泛的

2. within

™表达式的最小粒度为类

// ------------

// within 与 execution 相比，粒度更大，仅能实现到包和接口、类级别。而 execution 可以精确到方法的返回值，参数个数、修饰符、参数类型等

@Pointcut("within(com.chenss.dao.*)")//匹配 com.chenss.dao 包中的任意方法

@Pointcut("within(com.chenss.dao..*)")//匹配 com.chenss.dao 包及其子包中的任意方法

3. args

args 表达式的作用是匹配指定参数类型和指定参数数量的方法,与包名和类名无关

/**

* args 同 execution 不同的地方在于：

* args 匹配的是运行时传递给方法的参数类型

* execution(* *(java.io.Serializable))匹配的是方法在声明时指定的方法参数类型。

*/

@Pointcut("args(java.io.Serializable)")//匹配运行时传递的参数类型为指定类型的、且参数个数和顺序匹配

@Pointcut("@args(com.chenss.anno.Chenss)")//接受一个参数，并且传递的参数的运行时类型具有@Classified

4. this JDK 代理时，指向接口和代理类 proxy，cglib 代理时 指向接口和子类(不使用 proxy)
5. target 指向接口和子类

/**

* 此处需要注意的是，如果配置设置 proxyTargetClass=false，或默认为 false，则是用 JDK 代理，否则使用的是 CGLIB 代理

* JDK 代理的实现方式是基于接口实现，代理类继承 Proxy，实现接口。

* 而 CGLIB 继承被代理的类来实现。

* 所以使用 target 会保证目标不变，关联对象不会受到这个设置的影响。

* 但是使用 this 对象时，会根据该选项的设置，判断是否能找到对象。

*/

@Pointcut("target(com.chenss.dao.IndexDaoImpl)")//目标对象，也就是被代理的对象。限制目标对象为 com.chenss.dao.IndexDaoImpl 类

@Pointcut("this(com.chenss.dao.IndexDaoImpl)")//当前对象，也就是代理对象，代理对象时通过代理目标对象的方式获取新的对象，与原值并非一个

@Pointcut("@target(com.chenss.anno.Chenss)")//具有@Chenss 的目标对象中的任意方法

@Pointcut("@within(com.chenss.anno.Chenss)")//等同于@targ

这个比较难.....

proxy 模式里面有两个重要的术语

proxy Class

target Class

CGLIB 和 JDK 有区别　　JDK 是基于接口　　cglib 是基于继承所有 this 可以在 cglib 作用

6.　@annotation

这个很简单.....

作用方法级别

上述所有表达式都有@ 比如@Target(里面是一个注解类 xx,表示所有加了 xx 注解的类,和包名无关)

注意:上述所有的表达式可以混合使用,|| && !

@Pointcut("@annotation(com.chenss.anno.Chenss)")//匹配带有

com.chenss.anno.Chenss 注解的方法

1. bean

@Pointcut("bean(dao1)")//名称为 dao1 的 bean 上的任意方法

@Pointcut("bean(dao*)")

Spring AOP XML 实现方式的注意事项:

1. 在 aop:config 中定义切面逻辑，允许重复出现，重复多次，以最后出现的逻辑为准，但是次数以出现的次数为准
2. aop:aspect ID 重复不影响正常运行，依然能够有正确结果
3. aop:pointcut ID 重复会出现覆盖，以最后出现的为准。不同 aop:aspect 内出现的 pointcut 配置，可以相互引用

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

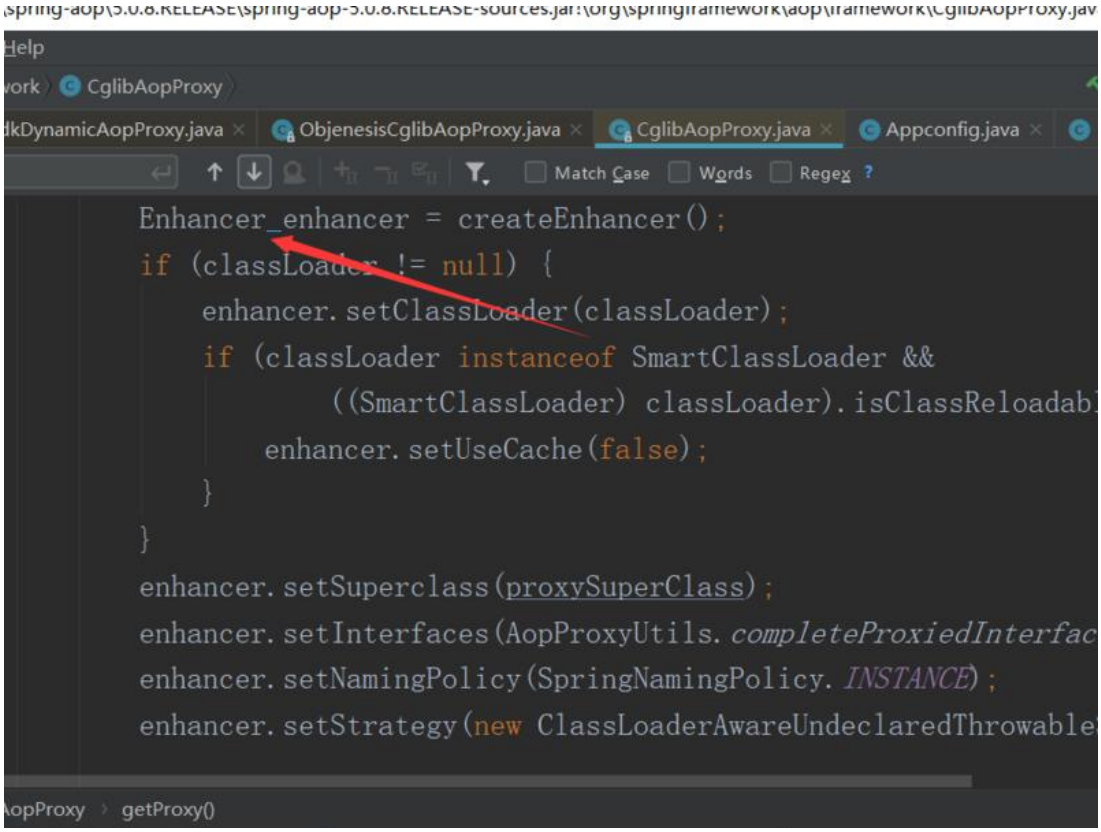
http://www.springframework.org/schema/beans/spring-beans.xsd

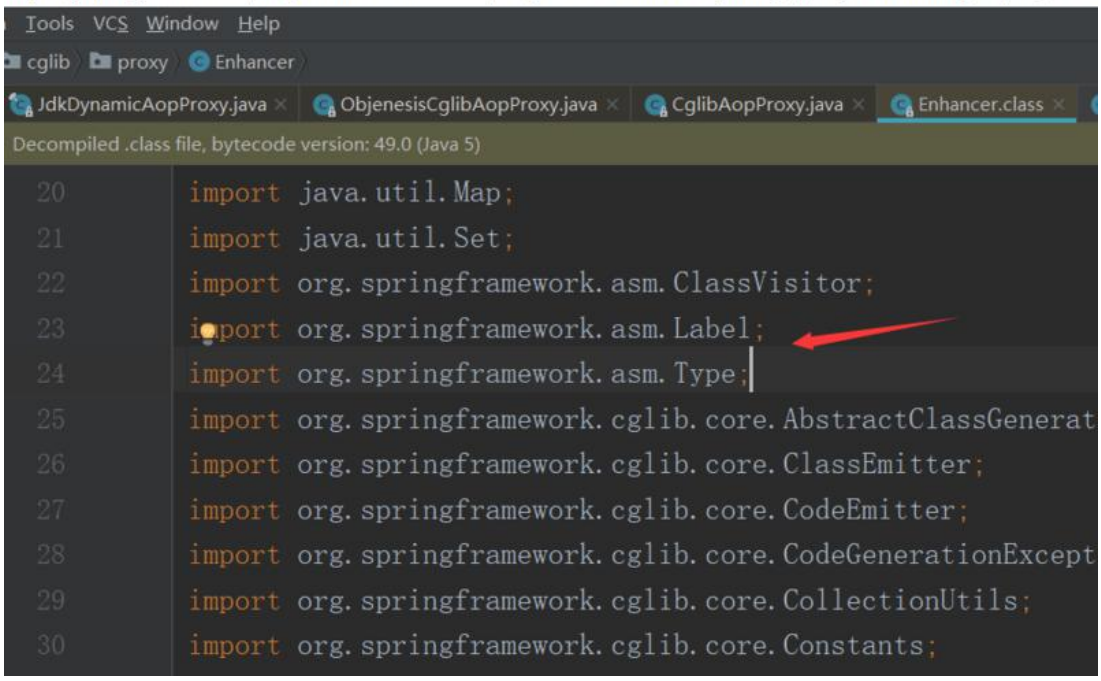
```
                                http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

                                http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">
    <!-- 定义开始进行注解扫描 -->
    <context:component-scan base-
package="com.chenss"></context:component-scan>

    <!-- 定义 AspectJ 对象使用的逻辑类，类中提供切面之后执行的逻辑方法 -->
    <bean id="aspectAop" class="com.chenss.aspectj.Aspect"></bean>
    <bean id="aspectAop2" class="com.chenss.aspectj.Aspect2"></bean>

    <bean id="indexDao" class="com.chenss.entity.IndexDao"></bean>

    <!--在 Config 中定义切面逻辑，允许重复出现，重复多次，以最后出现的逻辑为
准，但是次数以出现的次数为准-->
    <aop:config>
        <!-- aop:aspect ID 重复不影响正常运行，依然能够有正确结果 -->
        <!-- aop:pointcut ID 重复会出现覆盖，以最后出现的为准。不同
aop:aspect 内出现的 pointcut 配置，可以相互引用 -->
```

```
        <aop:aspect id="aspect" ref="aspectAop">
            <aop:pointcut id="aspectCut" expression="execution(*
com.chenss.entity.*.*())"/>
            <aop:before method="before" pointcut-
ref="aspectCut"></aop:before>
        fffffff
            <aop:pointcut id="aspectNameCut" expression="execution(*
com.chenss.entity.*.*(java.lang.String, ..))"/>
            <aop:before method="before2" pointcut-
ref="aspectNameCut"></aop:before>
        </aop:aspect>
    </aop:config>
</beans>
```

spring AOP 的源码分析

cglib


Help
CglibAopProxy
JdkDynamicAopProxy.java
ObjenesisCglibAopProxy.java
CglibAopProxy.java
Appconfig.java
Match Case
Words
Regex
Enhancer enhancer = createEnhancer();
if (classLoader != null) {
enhancer.setClassLoader(classLoader);
if (classLoader instanceof SmartClassLoader &&
((SmartClassLoader) classLoader).isClassReloadabl
enhancer.setUseCache(false);
}
}
enhancer.setSuperclass(proxySuperClass);
enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfac
enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
enhancer.setStrategy(new ClassLoaderAwareUndeclaredThrowable
getProxy()


Tools
VCS
Window
Help
cglib
proxy
Enhancer
JdkDynamicAopProxy.java
ObjenesisCglibAopProxy.java
CglibAopProxy.java
Enhancer.class
Decompiled .class file, bytecode version: 49.0 (Java 5)
import java.util.Map;
import java.util.Set;
import org.springframework.asm.ClassVisitor;
import org.springframework.asm.Label;
import org.springframework.asm.Type;
import org.springframework.cglib.core.AbstractClassGenerat
import org.springframework.cglib.core.ClassEmitter;
import org.springframework.cglib.core.CodeEmitter;
import org.springframework.cglib.core.CodeGenerationExcept
import org.springframework.cglib.core.CollectionUtils;
import org.springframework.cglib.core.Constants;

cglib 封装了 ASM 这个开源框架,对字节码操作,完成对代理类的创建

主要通过集成目标对象,然后完成重写,再操作字节码

具体看参考 ASM 的语法

JDK

在 Proxy 这个类当中首先实例化一个对象 ProxyClassFactory,然后在 get 方法中调用了 apply 方法,完成对代理类的创建

```
     * Generate a proxy class.  Must call the checkProxyAccess me
     * to perform permission checks before calling this.
     */
    private static Class<?> getProxyClass0(ClassLoader loader,
                                           Class<?>... interfaces
        if (interfaces.length > 65535) {
            throw new IllegalArgumentException("interface limit e
        }

        // If the proxy class defined by the given loader impleme
        // the given interfaces exists, this will simply return t
        // otherwise, it will create the proxy class via the Prox
        return proxyClassCache.get(loader, interfaces);
    }
```

其中最重要的两个方法

generateProxyClass 通过反射收集字段和属性然后生成字节

defineClass0 jvm 内部完成对上述字节的 load

```
/*
 * Generate the specified proxy class.
 */
byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
    proxyName, interfaces, accessFlags);
try {
    return defineClass0(loader, proxyName,
                        proxyClassFile,  off: 0, proxyClassFile.lengt
} catch (ClassFormatError e) {
    /*
     * A ClassFormatError here means that (barring bugs in the
     * proxy class generation code) there was some other
     * invalid aspect of the arguments supplied to the proxy
     * class creation (such as virtual machine limitations
     * exceeded).
     */
    throw new IllegalArgumentException(e.toString());
}
```

总结:cglib 是通过继承来操作子类的字节码生成代理类,JDK 是通过接口,然后利用 java 反射完成对类的动态创建,严格意义上来说 cglib 的效率高于 JDK 的反射,但是这种效率取决于代码功力,其实可以忽略不计,毕竟 JDK 是 JVM 的亲儿子.....

spring5 新特性

1 使用 lambda 表达式定义 bean

2 日志 spring4 的日志是用 jcl,原生的 JCL,底层通过循环去加载具·体的日志实现技术,所以有先后顺序,spring5 利用的是 spring-jcl,其实就是 spring 自己改了 JCL 的代码具体参考视频当中讲的两者的区别

新特性还有其他,但是这两个比较重要,由于时间问题,其他的特性可以去网上找到相应资料,但是这两个应付面试绝对可以了,其他的特性噱头居多,实用性可能不是很大.