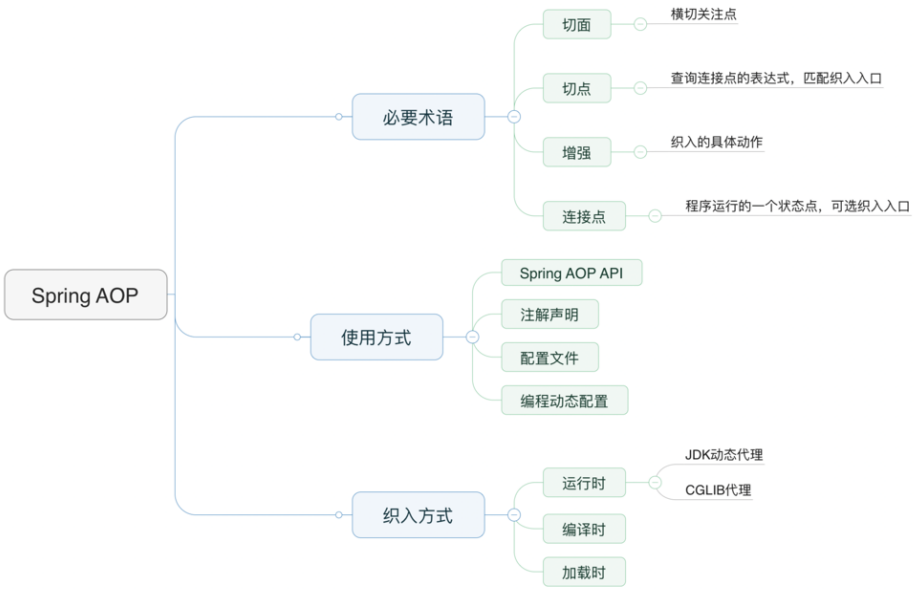
朱晔和你聊 Spring 系列 S1E6：容易犯错的 Spring AOP

标题有点标题党了，这里说的容易犯错不是 Spring AOP 的错，是指使用的时候容易犯错。本文会以一些例子来展开讨论 AOP 的使用以及使用过程中容易出错的点。

几句话说清楚 AOP



有关必要术语：

1. 切面：Aspect，有的地方也叫做方面。切面=切点+增强，表示我们在什么点切入蛋糕，切入蛋糕后我们以什么方式来增强这个点。
2. 切点：Pointcut，类似于查询表达式，通过在连接点运行查询表达式来寻找匹配切入点，Spring AOP 中默认使用 AspjectJ 查询表达式。
3. 增强：Advice，有的地方也叫做通知。定义了切入切点后增强的方式，增强方式有前、后、环绕等等。Spring AOP 中把增强定义为拦截器。

4. 连接点：Join point，蛋糕所有可以切入的点，对于 Spring AOP 连接点就是方法执行。

有关使用方式：

1. Spring AOP API：这种方式是 Spring AOP 实现的基石。最老的使用方式，在 Spring 1.2 中的时候用这种 API 的方式定义 AOP。
2. 注解声明：使用@AspectJ 的@Aspect、@Pointcut 等注解来定义 AOP。现在基本都使用这种方式来定义，也是官方推荐的方式。
3. 配置文件：相比注解声明方式，配置方式有两个缺点，一是定义和实现分离了，二是功能上会比注解声明弱，无法实现全部功能。好处么就是 XML 在灵活方面会强一点。
4. 编程动态配置：使用 AspectJProxyFactory 进行动态配置。可以作为注解方式静态配置的补充。

有关织入方式：

织入说通俗点就是怎么把增强代码注入到连接点，和被增强的代码融入到一起。

1. 运行时：Spring AOP 只支持这种方式。实现上有两种方式，一是 JDK 动态代理，通过反射实现，只支持对实现接口的类进行代理，二是 CGLIB 动态字节码注入方式实现代理，没有这个限制。Spring 3.2 之后的版本已经包含了 CGLIB，会根据需要选择合适的方式来使用。
2. 编译时：在编译的时候把增强代码注入进去，通过 AspjectJ 的 ajc 编译器实现。实现上有两种方式，一种是直接使用 ajc 编译所有代码，还有一种是 javac 编译后再进行后处理。
3. 加载时：在 JVM 加载类型的时候注入代码，也叫做 LTW。通过启动程序的时候通过 javaagent 代理默认的类加载器实现。

使用 Spring AOP 实现事务的坑

新建一个模块：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>me.josephzhu</groupId>
   <artifactId>spring101-aop</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>jar</packaging>

   <name>spring101-aop</name>
   <description></description>

   <parent>
      <groupId>me.josephzhu</groupId>
      <artifactId>spring101</artifactId>
      <version>0.0.1-SNAPSHOT</version>
   </parent>

   <dependencies>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-aop</artifactId>
      </dependency>
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>1.3.2</version>
        </dependency>
```

```
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
        </dependency>
      <dependency>
         <groupId>com.fasterxml.jackson.core</groupId>
         <artifactId>jackson-databind</artifactId>
         <version>2.9.7</version>
      </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

</project>
```

在这里我们引入了 jackson，以后我们会用来做 JSON 序列化。引入了 mybatis 启动器，以后我们会用 mybstis 做数据访问。引入了 h2 嵌入式数据库，方便本地测试使用。引入了 web 启动器，之后我们还会来测试一下对 web 项目的 Controller 进行注入。

先来定义一下我们的测试数据类：

```
package me.josephzhu.spring101aop;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@Data
@NoArgsConstructor
@AllArgsConstructor
```

```
public class MyBean {

    private Long id;

    private String name;

    private Integer age;

    private BigDecimal balance;

}
```

然后，我们在 resources 文件夹下创建 schema.sql 文件来初始化 h2 数据库：

```
CREATE TABLE PERSON(

ID BIGINT  PRIMARY KEY AUTO_INCREMENT,

NAME VARCHAR(255),

AGE SMALLINT,

BALANCE DECIMAL

);
```

还可以在 resources 文件夹下创建 data.sql 来初始化数据：

```
INSERT INTO PERSON (NAME, AGE, BALANCE) VALUES ('zhuye', 35, 1000);
```

这样程序启动后就会有一个 PERSON 表，表里有一条 ID 为 1 的记录。

通过启动器使用 Mybatis 非常简单，无需进行任何配置，建一个 Mapper 接口：

```
package me.josephzhu.spring101aop;


import org.apache.ibatis.annotations.Insert;

import org.apache.ibatis.annotations.Mapper;

import org.apache.ibatis.annotations.Select;


import java.util.List;


@Mapper

public interface DbMapper {

    @Select("SELECT COUNT(0) FROM PERSON")

    int personCount();


    @Insert("INSERT INTO PERSON (NAME, AGE, BALANCE) VALUES ('zhuye', 35, 1000)")
```

```
    void personInsertWithoutId();

    @Insert("INSERT INTO PERSON (ID, NAME, AGE, BALANCE) VALUES (1,'zhuye', 35, 1000)")
    void personInsertWithId();

    @Select("SELECT * FROM PERSON")
    List<MyBean> getPersonList();

}
```

这里我们定义了 4 个方法：

1. 查询表中记录数的方法
2. 查询表中所有数据的方法
3. 带 ID 字段插入数据的方法，由于程序启动的时候已经初始化了一条数据，如果这里我们再插入 ID 为 1 的记录显然会出错，用来之后测试事务使用
4. 不带 ID 字段插入数据的方法

为了我们可以观察到数据库连接是否被 Spring 纳入事务管理，我们在 application.properties 配置文件中设置 mybatis 的 Spring 事务日志级别为 DEBUG：

```
logging.level.org.mybatis.spring.transaction=DEBUG
```

现在我们来创建服务接口：

```
package me.josephzhu.spring101aop;

import java.time.Duration;
import java.util.List;

public interface MyService {
    void insertData(boolean success);
    List<MyBean> getData(MyBean myBean, int count, Duration delay);
}
```

定义了插入数据和查询数据的两个方法，下面是实现：

```java
package me.josephzhu.spring101aop;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.Duration;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

@Service
public class MyServiceImpl implements MyService {

    @Autowired
    private DbMapper dbMapper;

    @Transactional(rollbackFor = Exception.class)
    public void _insertData(boolean success){
        dbMapper.personInsertWithoutId();
        if(!success)
            dbMapper.personInsertWithId();
    }

    @Override
    public void insertData(boolean success) {
        try {
            _insertData(success);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("记录数: " + dbMapper.personCount());
    }
```

```
    @Override
    public List<MyBean> getData(MyBean myBean, int count, Duration delay) {
        try {
            Thread.sleep(delay.toMillis());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return IntStream.rangeClosed(1,count)
                .mapToObj(i->new MyBean((long)i,myBean.getName() + i, myBean.getAge(),
myBean.getBalance()))
                .collect(Collectors.toList());
    }
}
```

getData 方法我们就不细说了，只是实现了休眠然后根据传入的 myBean 作为模板组装了 count 条测试数据返回。我们来重点看一下 insertData 方法，这就是使用 Spring AOP 的一个坑了。看上去配置啥的都没问题，但是_insertData 是不能生效自动事务管理的。

我们知道 Spring AOP 使用代理目标对象方式实现 AOP，在从外部调用 insertData 方法的时候其实走的是代理，这个时候事务环绕可以生效，在方法内部我们通过 this 引用调用 _insertData 方法，虽然方法外部我们设置了 Transactional 注解，但是由于走的不是代理调用，Spring AOP 自然无法通过 AOP 增强为我们做事务管理。

我们来创建主程序测试一下：

```
package me.josephzhu.spring101aop;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.AdviceMode;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.transaction.annotation.EnableTransactionManagement;


import java.math.BigDecimal;
import java.time.Duration;


@SpringBootApplication
public class Spring101AopApplication implements CommandLineRunner {


   public static void main(String[] args) {
      SpringApplication.run(Spring101AopApplication.class, args);
   }


   @Autowired
   private MyService myService;


   @Override
   public void run(String... args) throws Exception {
      myService.insertData(true);
      myService.insertData(false);
      System.out.println(myService.getData(new MyBean(0L, "zhuye",35, new
BigDecimal("1000")),
            5,
            Duration.ofSeconds(1)));
   }
}
```

在 Runner 中，我们使用 true 和 false 调用了两次 insertData 方法。后面一次调用肯定会失败，因为_insert 方法中会进行重复 ID 的数据插入。运行程序后得到如下输出：

```
2018-10-07 09:11:44.605  INFO 19380 --- [           main] m.j.s.Spring101AopApplication            : Started
Spring101AopApplication in 3.072 seconds (JVM running for 3.74)
```

2018-10-07 09:11:44.621 DEBUG 19380 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@2126664214 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will not be managed by Spring

2018-10-07 09:11:44.626 DEBUG 19380 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@775174220 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will not be managed by Spring

记录数：2

2018-10-07 09:11:44.638 DEBUG 19380 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@2084486251 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will not be managed by Spring

2018-10-07 09:11:44.638 DEBUG 19380 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@26418585 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will not be managed by Spring

2018-10-07 09:11:44.642 INFO 19380 --- [main] o.s.b.f.xml.XmlBeanDefinitionReader : Loading XML bean definitions from class path resource [org/springframework/jdbc/support/sql-error-codes.xml]

org.springframework.dao.DuplicateKeyException:

Error updating database. Cause: org.h2.jdbc.JdbcSQLException: Unique index or primary key violation: "PRIMARY KEY ON PUBLIC.PERSON(ID)"; SQL statement:

INSERT INTO PERSON (ID, NAME, AGE, BALANCE) VALUES (1,'zhuye', 35, 1000) [23505-197]

2018-10-07 09:11:44.689 DEBUG 19380 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@529949842 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will not be managed by Spring

记录数：3

[MyBean(id=1, name=zhuye1, age=35, balance=1000), MyBean(id=2, name=zhuye2, age=35, balance=1000), MyBean(id=3, name=zhuye3, age=35, balance=1000), MyBean(id=4, name=zhuye4, age=35, balance=1000), MyBean(id=5, name=zhuye5, age=35, balance=1000)]

从日志的几处我们都可以得到结论，事务管理没有生效：

1. 我们可以看到有类似 Connection will not be managed by Spring 的提示，说明连接没有进入 Spring 的事务管理。
2. 程序启动的时候记录数为 1，第一次调用 insertData 方法后记录数为 2，第二次调用方法如果事务生效方法会回滚记录数会维持在 2，在输出中我们看到记录数最后是 3。

那么，如何解决这个问题呢，有三种方式：

1. 使用 AspjectJ 来实现 AOP，这种方式是直接修改代码的，不是走代理实现的，不会有这个问题，下面我们会详细说明一下这个过程。
2. 在代码中使用 AopContext.currentProxy()来获得当前的代理进行_insertData 方法调用。这种方式侵入太强，而且需要被代理类意识到自己是通过代理被访问，显然不是合适的方式。
3. 改造代码，使需要事务代理的方法直接调用，类似：

```
@Override
@Transactional(rollbackFor = Exception.class)
public void insertData(boolean success) {
    dbMapper.personInsertWithoutId();
    if(!success)
        dbMapper.personInsertWithId();
}
```

这里还容易犯错的地方是，这里不能对异常进行捕获，否则 Spring 事务代理无法捕获到异常也就无法实现回滚。

使用 AspectJ 静态织入进行改造

那么原来这段代码如何不改造实现事务呢？可以通过 AspjectJ 编译时静态织入实现。整个配置过程如下：

首先在 pom 中加入下面的配置：

```
<build>
      <sourceDirectory>${project.build.directory}/generated-
sources/delombok</sourceDirectory>
      <plugins>
     <plugin>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-maven-plugin</artifactId>
     </plugin>
         <plugin>
             <groupId>org.projectlombok</groupId>
             <artifactId>lombok-maven-plugin</artifactId>
             <version>1.18.0.0</version>
             <executions>
                 <execution>
                     <phase>generate-sources</phase>
                     <goals>
                         <goal>delombok</goal>
                     </goals>
                 </execution>
             </executions>
             <configuration>
                 <addOutputDirectory>false</addOutputDirectory>
                 <sourceDirectory>src/main/java</sourceDirectory>
             </configuration>
         </plugin>
         <plugin>
             <groupId>org.codehaus.mojo</groupId>
             <artifactId>aspectj-maven-plugin</artifactId>
             <version>1.10</version>
             <configuration>
                 <complianceLevel>1.8</complianceLevel>
```

```
                <source>1.8</source>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-aspects</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>test-compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

这里的一个坑是 ajc 编译器无法支持 lambok，我们需要先使用 lombok 的插件在生成源码阶段对 lombok 代码进行预处理，然后我们再通过 aspjectj 插件来编译代码。Pom 文件中还需要加入如下依赖：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
</dependency>
```

然后需要配置 Spring 来使用 ASPECTJ 的增强方式来做事务管理：

```
@EnableTransactionManagement(mode = AdviceMode.ASPECTJ)
public class Spring101AopApplication implements CommandLineRunner {
```

重新使用 maven 编译代码后可以看到，相关代码已经变了样：

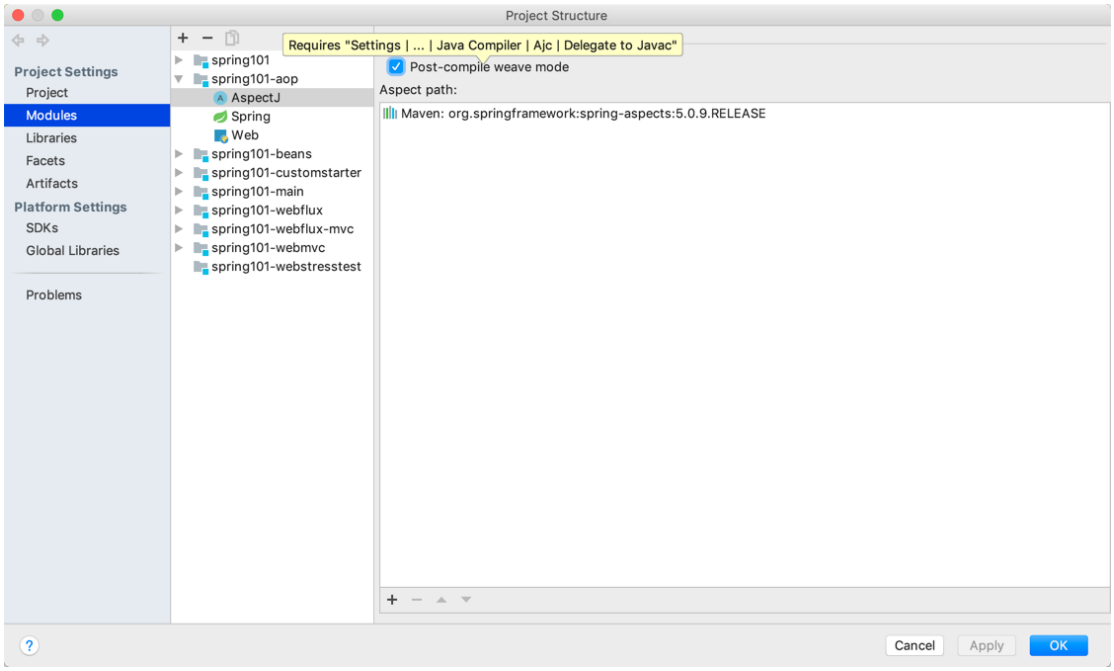
```
@Transactional(
    rollbackFor = {Exception.class}
)
public void _insertData(boolean success) {
    AnnotationTransactionAspect var10000 = AnnotationTransactionAspect.aspectOf();
    Object[] var3 = new Object[]{this, Conversions.booleanObject(success)};

var10000.ajc$around$org_springframework_transaction_aspectj_AbstractTransactionAspect$
1$2a73e96c(this, new MyServiceImpl$AjcClosure1(var3), ajc$tjp_0);
}

public void insertData(boolean success) {
    try {
        this._insertData(success);
    } catch (Exception var3) {
        var3.printStackTrace();
    }

    System.out.println("记录数: " + this.dbMapper.personCount());
}
```

运行程序可以看到如下日志：

2018-10-07 09:35:12.360 DEBUG 19459 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@1169317628 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will be managed by Spring

而且最后输出的结果是 2，说明第二次插入数据整体回滚了。

如果使用 IDEA 的话还可以配置先由 javac 编译再由 ajc 后处理，具体参见 IDEA 官网这里不详述。



使用 AOP 进行事务后处理

我们先使用刚才说的方法 3 改造一下代码，使得 Spring AOP 可以处理事务（Aspject AOP 功能虽然强大但是和 Spring 结合的不好，所以我们接下去的测试还是使用 Spring AOP），删除 aspjectj 相关依赖，在 IDEA 配置回 javac 编译器重新编译项目。本节中我们尝试建立第一个我们的切面：

```
package me.josephzhu.spring101aop;


import lombok.extern.slf4j.Slf4j;

import org.aspectj.lang.JoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import org.springframework.transaction.support.TransactionSynchronizationAdapter;

import org.springframework.transaction.support.TransactionSynchronizationManager;
```

```
@Aspect
@Component
@Slf4j
class TransactionalAspect extends TransactionSynchronizationAdapter {

    @Autowired
    private DbMapper dbMapper;

    private ThreadLocal<JoinPoint> joinPoint = new ThreadLocal<>();

    @Before("@within(org.springframework.transaction.annotation.Transactional) ||
@annotation(org.springframework.transaction.annotation.Transactional)")
    public void registerSynchronization(JoinPoint jp) {
        joinPoint.set(jp);
        TransactionSynchronizationManager.registerSynchronization(this);
    }

    @Override
    public void afterCompletion(int status) {
        log.info(String.format("【%s】【%s】事务提交 %s，目前记录数：%s",
                joinPoint.get().getSignature().getDeclaringType().toString(),
                joinPoint.get().getSignature().toLongString(),
                status == 0 ? "成功":"失败",
                dbMapper.personCount()));
        joinPoint.remove();
    }
}
```

在这里，我们的切点是所有标记了@Transactional 注解的类以及标记了@Transactional 注解的方法，我们的增强比较简单，在事务同步管理器注册一个回调方法，用于事务完成后进行额外的处理。这里的一个坑是 Spring 如何实例化切面。通过查文档或做实验可以得知，默认情况下 TranscationalAspect 是单例的，在多线程情况下，可能会有并发，保险起见我们使用 ThreadLocal 来存放。运行代码后可以看到如下输出：

2018-10-07 10:01:32.384 INFO 19599 --- [main] m.j.spring101aop.TransactionalAspect : 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】事务提交 成功，目前记录数：2

2018-10-07 10:01:32.385 DEBUG 19599 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@1430104337 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will be managed by Spring

2018-10-07 10:01:32.449 DEBUG 19599 --- [main] o.m.s.t.SpringManagedTransaction : JDBC Connection [HikariProxyConnection@1430104337 wrapping conn0: url=jdbc:h2:mem:testdb user=SA] will be managed by Spring

2018-10-07 10:01:32.449 INFO 19599 --- [main] m.j.spring101aop.TransactionalAspect : 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】事务提交 失败，目前记录数：2

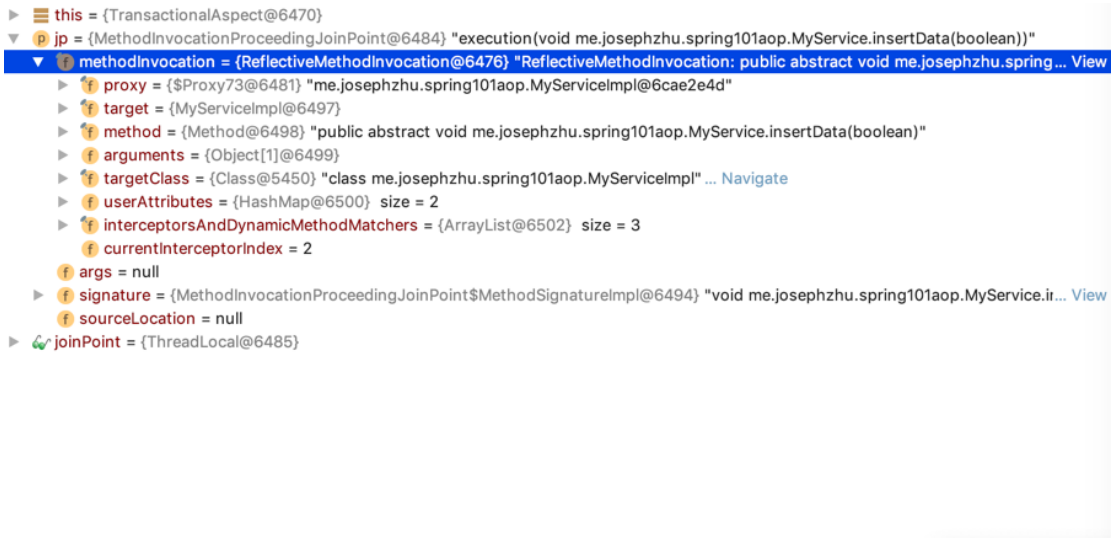
可以看到 Spring AOP 做了事务管理，我们两次事务提交第一次成功第二次失败，失败后记录数还是 2。这个功能还可以通过 Spring 的@TransactionalEventListener 注解实现，这里不详述。

切换 JDK 代理和 CGLIB 代理

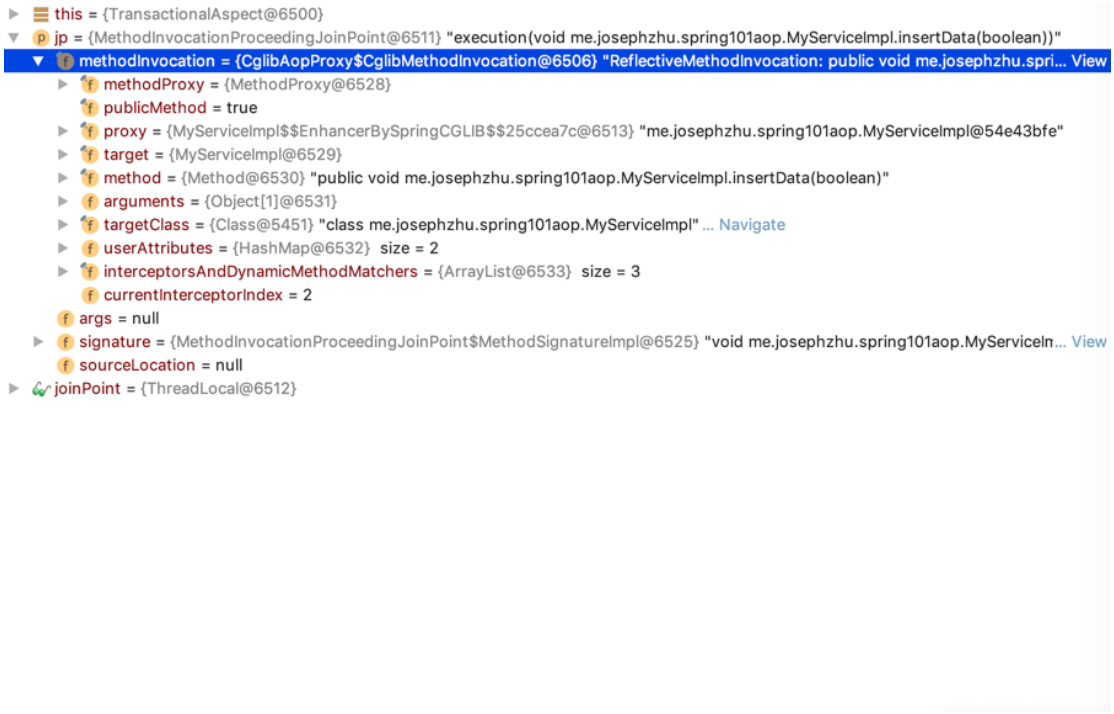
我们现在注入的是接口，我们知道对于这种情况 Spring AOP 应该使用的是 JDK 代理。但是 SpringBoot 默认开启了下面的属性来全局启用 CGLIB 代理：

```
spring.aop.proxy-target-class=true
```

我们尝试把这个属性设置成 false，然后在刚才的 TransationalAspect 中的增强方法设置断点，可以看到这是一个 ReflectiveMethodInvocation：



把配置改为 true 重新观察可以看到变为了 CglibMethodInvocation：



我们把开关改为 false，然后切换到注入实现，运行程序会得到如下错误提示，意思就是我我们走 JDK 代理的话不能注入实现，需要注入接口：

The bean 'myServiceImpl' could not be injected as a 'me.josephzhu.spring101aop.MyServiceImpl' because it is a JDK dynamic proxy that implements:

me.josephzhu.spring101aop.MyService

我们修改我们的 MyServiceImpl，去掉实现接口的代码和@Override 注解，使之成为一个普通的类，重新运行程序可以看到我们的代理方式自动降级为了 CGLIB 方式（虽然 spring.aop.proxy-target-class 参数我们现在设置的是 false）。

使用 AOP 无缝实现日志+异常+打点

现在我们来实现一个复杂点的切面的例子。我们知道，出错记录异常信息，对于方法调用记录打点信息（如果不知道什么是打点可以参看《朱晔的互联网架构实践心得 S1E4：简单好用的监控六兄弟》），甚至有的时候为了排查问题需要记录方法的入参和返回，这三个事情是我们经常需要做的和业务逻辑无关的事情，我们可以尝试使用 AOP 的方式一键切入这三个事情的实现，在业务代码无感知的情况下做好监控和打点。

首先实现我们的注解，通过这个注解我们可以细化控制一些功能：

```
package me.josephzhu.spring101aop;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Metrics {
    /**
     * 是否在成功执行方法后打点
     * @return
     */
    boolean recordSuccessMetrics() default true;

    /**
```

```
     * 是否在执行方法出错时打点
     * @return
     */
    boolean recordFailMetrics() default true;

    /**
     * 是否记录请求参数
     * @return
     */
    boolean logParameters() default true;

    /**
     * 是否记录返回值
     * @return
     */
    boolean logReturn() default true;

    /**
     * 是否记录异常
     * @return
     */
    boolean logException() default true;

    /**
     * 是否屏蔽异常返回默认值
     * @return
     */
    boolean ignoreException() default false;
}
```

下面我们就来实现这个切面：

```
package me.josephzhu.spring101aop;
```

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.web.context.request.RequestAttributes;
import org.springframework.web.context.request.RequestContextHolder;
import org.springframework.web.context.request.ServletRequestAttributes;

import javax.servlet.http.HttpServletRequest;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.time.Duration;
import java.time.Instant;

@Aspect
@Component
@Slf4j
public class MetricsAspect {
    private static ObjectMapper objectMapper = new ObjectMapper();

    @Around("@annotation(me.josephzhu.spring101aop.Metrics) ||
@within(org.springframework.stereotype.Controller)")
    public Object metrics(ProceedingJoinPoint pjp) throws Throwable {
        //1
        MethodSignature signature = (MethodSignature) pjp.getSignature();
        Metrics metrics;
        String name;
        if (signature.getDeclaringType().isInterface()) {
            Class implClass = pjp.getTarget().getClass();
```

```
            Method method = implClass.getMethod(signature.getName(),
signature.getParameterTypes());
            metrics = method.getDeclaredAnnotation(Metrics.class);
            name = String.format("【%s】【%s】", implClass.toString(),
method.toString());
        } else {
            metrics = signature.getMethod().getAnnotation(Metrics.class);
            name = String.format("【%s】【%s】", signature.getDeclaringType().toString(),
signature.toLongString());
        }
        //2
        if (metrics == null)
            metrics = new Metrics() {
                @Override
                public boolean logException() {
                    return true;
                }

                @Override
                public boolean logParameters() {
                    return true;
                }

                @Override
                public boolean logReturn() {
                    return true;
                }

                @Override
                public boolean recordFailMetrics() {
                    return true;
                }
```

```
                @Override
                public boolean recordSuccessMetrics() {
                    return true;
                }

                @Override
                public boolean ignoreException() {
                    return false;
                }

                @Override
                public Class<? extends Annotation> annotationType() {
                    return Metrics.class;
                }
            };
        RequestAttributes requestAttributes =
RequestContextHolder.getRequestAttributes();
        if (requestAttributes != null) {
            HttpServletRequest request = ((ServletRequestAttributes)
requestAttributes).getRequest();
            if (request != null)
                name += String.format("【%s】", request.getRequestURL().toString());
        }
        //3
        if (metrics.logParameters())
            log.info(String.format("【入参日志】调用 %s 的参数是：【%s】", name,
objectMapper.writeValueAsString(pjp.getArgs())));
        //4
        Object returnValue;
        Instant start = Instant.now();
        try {
            returnValue = pjp.proceed();
            if (metrics.recordSuccessMetrics())
```

```
                log.info(String.format("【成功打点】调用 %s 成功，耗时：%s", name,
Duration.between(Instant.now(), start).toString()));
        } catch (Exception ex) {
            if (metrics.recordFailMetrics())
                log.info(String.format("【失败打点】调用 %s 失败，耗时：%s", name,
Duration.between(Instant.now(), start).toString()));

            if (metrics.logException())
                log.error(String.format("【异常日志】调用 %s 出现异常！", name), ex);

            if (metrics.ignoreException())
                returnValue = getDefaultValue(signature.getReturnType().toString());
            else
                throw ex;
        }
        //5
        if (metrics.logReturn())
            log.info(String.format("【出参日志】调用 %s 的返回是：【%s】", name,
returnValue));
        return returnValue;
    }

    private static Object getDefaultValue(String clazz) {
        if (clazz.equals("boolean")) {
            return false;
        } else if (clazz.equals("char")) {
            return '\u0000';
        } else if (clazz.equals("byte")) {
            return 0;
        } else if (clazz.equals("short")) {
            return 0;
        } else if (clazz.equals("int")) {
            return 0;
```

```
        } else if (clazz.equals("long")) {
            return 0L;
        } else if (clazz.equals("flat")) {
            return 0.0F;
        } else if (clazz.equals("double")) {
            return 0.0D;
        } else {
            return null;
        }
    }

}
```

看上去代码量很多，其实实现比较简单：

1. 最关键的切点，我们在两个点切入，一是标记了 Metrics 注解的方法，二是标记了 Controller 的类（我们希望实现的目标是对于 Controller 所有方法默认都加上这个功能，因为这是对外的接口，比较重要）。所以在之后的代码中，我们还需要额外对 Web 程序做一些处理。
2. 对于@Around 我们的参数是 ProceedingJoinPoint 不是 JoinPoint，因为环绕增强允许我们执行方法调用。
3. 第一段代码，我们尝试获取当前方法的类名和方法名。这里有一个坑，如果连接点是接口的话，@Metrics 的定义需要从实现类（也就是代理的 Target）上获取。作为框架的开发者，我们需要考虑到各种使用方使用的情况，如果有遗留的话就会出现 BUG。
4. 第二段代码，是为 Web 项目准备的，如果我们希望默认为所有的 Controller 方法做日志异常打点处理的话，我们需要初始化一个@Metrics 注解出来，然后对于 Web 项目我们可以从上下文中获取到额外的一些信息来丰富我们的日志。
5. 第三段代码，实现的是入参的日志输出。
6. 第四段代码，实现的是连接点方法的执行，以及成功失败的打点，出现异常的时候还会记录日志。这里我们通过日志方式暂时替代了打点的实现，标准的实现是需要把信息提交到类似 Graphite 这样的时间序列数据库或对接 SpringBoot Actuator。另外，如果开启忽略异常的话，我们需要把结果替换为返回类型的默认值，并且吃掉异常。
7. 第五段代码，实现了返回值的日志输出。

最后，我们修改一下 MyServiceImpl 的实现，在 insertData 和 getData 两个方法上加入我们的@Metrics 注解。运行程序可以看到如下输出：

2018-10-07 10:47:00.813 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【入参日志】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】 的参数是：【[true]】

2018-10-07 10:47:00.864 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【成功打点】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】 成功，耗时：PT-0.048S

2018-10-07 10:47:00.864 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【出参日志】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】 的返回是：【null】

2018-10-07 10:47:00.927 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【入参日志】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】 的参数是：【[false]】

2018-10-07 10:47:01.084 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【失败打点】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】 失败，耗时：PT-0.156S

2018-10-07 10:47:01.102 ERROR 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【异常日志】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public void me.josephzhu.spring101aop.MyServiceImpl.insertData(boolean)】 出现异常！

2018-10-07 10:47:01.231 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【入参日志】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public java.util.List me.josephzhu.spring101aop.MyServiceImpl.getData(me.josephzhu.spring101aop.MyBean,int,java.time.Duration)】 的参数是：

【[{"id":0,"name":"zhuye","age":35,"balance":1000},5,{"seconds":1,"zero":false,"nano":0,"units":["SECONDS","NANOS"],"negative":false}]】

2018-10-07 10:47:02.237 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【成功打点】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public java.util.List me.josephzhu.spring101aop.MyServiceImpl.getData(me.josephzhu.spring101aop.MyBean,int,java.time.Duration)】 成功，耗时：PT-1.006S

2018-10-07 10:47:02.237 INFO 19737 --- [main] me.josephzhu.spring101aop.MetricsAspect : 【出参日志】调用 【class me.josephzhu.spring101aop.MyServiceImpl】【public java.util.List me.josephzhu.spring101aop.MyServiceImpl.getData(me.josephzhu.spring101aop.MyBean,int,java.time.Duration)】 的返回是：【[MyBean(id=1, name=zhuye1, age=35, balance=1000), MyBean(id=2, name=zhuye2, age=35, balance=1000), MyBean(id=3, name=zhuye3, age=35, balance=1000), MyBean(id=4, name=zhuye4, age=35, balance=1000), MyBean(id=5, name=zhuye5, age=35, balance=1000)]】

[MyBean(id=1, name=zhuye1, age=35, balance=1000), MyBean(id=2, name=zhuye2, age=35, balance=1000), MyBean(id=3, name=zhuye3, age=35, balance=1000), MyBean(id=4, name=zhuye4, age=35, balance=1000), MyBean(id=5, name=zhuye5, age=35, balance=1000)]

正确实现了参数日志、异常日志、成功失败打点（含耗时统计）等功能。

下面我们创建一个 Controller 来测试一下是否可以自动切入 Controller：

```
package me.josephzhu.spring101aop;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.List;

@Controller
public class MyController {

    @Autowired
    private DbMapper dbMapper;
```

```
    @ResponseBody
    @GetMapping("/data")
    public List<MyBean> getPersonList(){
        return dbMapper.getPersonList();
    }
}
```

运行程序打开浏览器访问 http://localhost:8080/data 后能看到如下输出：

2018-10-07 10:49:53.811 INFO 19737 --- [nio-8080-exec-1] me.josephzhu.spring101aop.MetricsAspect : 【入参日志】调用 【class me.josephzhu.spring101aop.MyController】 【public java.util.List me.josephzhu.spring101aop.MyController.getPersonList()】 【http://localhost:8080/data】 的参数是：【[]】

2018-10-07 10:49:53.819 INFO 19737 --- [nio-8080-exec-1] me.josephzhu.spring101aop.MetricsAspect : 【成功打点】调用 【class me.josephzhu.spring101aop.MyController】 【public java.util.List me.josephzhu.spring101aop.MyController.getPersonList()】 【http://localhost:8080/data】 成功，耗时：PT-0.008S

2018-10-07 10:49:53.819 INFO 19737 --- [nio-8080-exec-1] me.josephzhu.spring101aop.MetricsAspect : 【出参日志】调用 【class me.josephzhu.spring101aop.MyController】 【public java.util.List me.josephzhu.spring101aop.MyController.getPersonList()】 【http://localhost:8080/data】 的返回是：【[MyBean(id=1, name=zhuye, age=35, balance=1000), MyBean(id=2, name=zhuye, age=35, balance=1000)]】

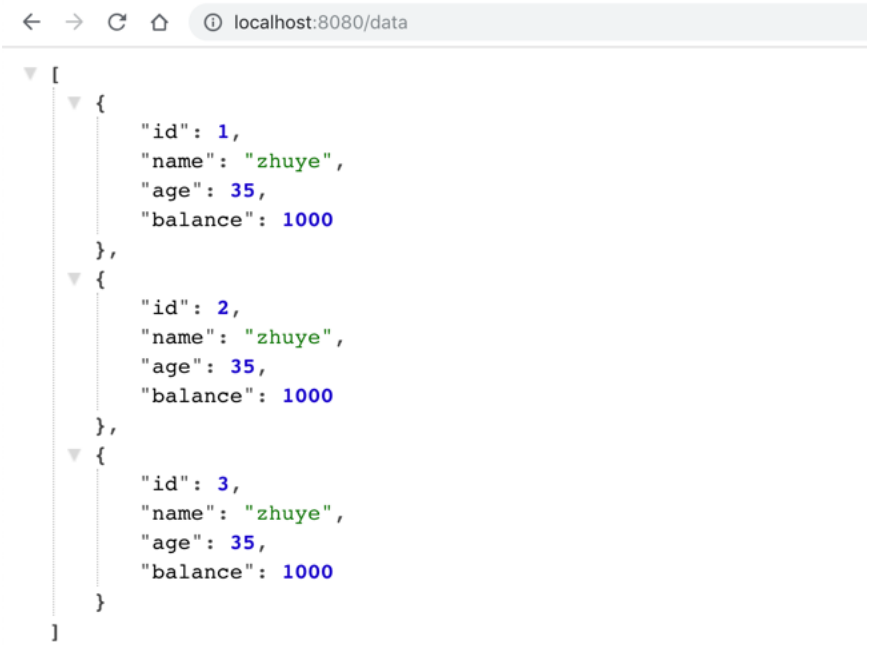
最后，我们再来踩一个坑。我们来测一下 ignoreException 吞掉异常的功能（默认为 false）：

```
@Transactional(rollbackFor = Exception.class)
@Override
```

```
@Metrics(ignoreException = true)
public void insertData(boolean success){
    dbMapper.personInsertWithoutId();
    if(!success)
        dbMapper.personInsertWithId();
}
```

这个功能会吞掉异常，在和 Transactional 事务管理结合时候会不会出问题呢？

开启这个配置后刷新页面可以看到数据库内有三条记录了，说明第二次的 insertData 方法执行没有成功回滚事务。这也是合情合理的，毕竟我们的 MetricsAspect 吃掉了异常。



怎么绕开这个问题呢？答案是我们需要手动控制一下我们的切面的执行优先级，我们希望这个切面优先级比 Spring 事务控制切面优先级低：

```
@Aspect
@Component
@Slf4j
@Order(1)
public class MetricsAspect {
```

再次运行程序可以看到事务正确回滚。

总结

本文我们通过一些例子覆盖了如下内容：

1. Spring AOP 的一些基本知识点。
2. Mybatis 和 H2 的简单配置使用。
3. 如何实现 Spring 事务管理。
4. 如何切换为 AspjectJ 进行 AOP。
5. 观察 JDK 代理和 CGLIB 代理。
6. 如何定义切面实现事务后处理和日志异常打点这种横切关注点。

在整个过程中，也踩了下面的坑，印证的本文的标题：

1. Spring AOP 代理不能作用于代理类内部 this 方法调用的坑。
2. Spring AOP 实例化切面默认单例的坑。
3. AJC 编译器无法支持 lambok 的坑。
4. 切面优先级顺序的坑。
5. 切面内部获取注解方式的坑。

老样子，本系列文章代码见我的 github：https://github.com/JosephZhu1983/Spring101。