

19 | Spring框架：IoC和AOP是扩展的核心

2020-04-25 朱晔

Java业务开发常见错误100例

[进入课程 >](#)



讲述：王少泽

时长 17:44 大小 16.24M



你好，我是朱晔。今天，我们来聊聊 Spring 框架中的 IoC 和 AOP，及其容易出错的地方。

熟悉 Java 的同学都知道，Spring 的家族庞大，常用的模块就有 Spring Data、Spring Security、Spring Boot、Spring Cloud 等。其实呢，Spring 体系虽然庞大，但都是围绕 Spring Core 展开的，而 Spring Core 中最核心的就是 IoC（控制反转）和 AOP（面向切面编程）。



概括地说，IoC 和 AOP 的初衷是解耦和扩展。理解这两个核心技术，就可以让你的代码变得更灵活、可随时替换，以及业务组件间更解耦。在接下来的两讲中，我会与你深入剖析几个案例，带你绕过业务中通过 Spring 实现 IoC 和 AOP 相关的坑。

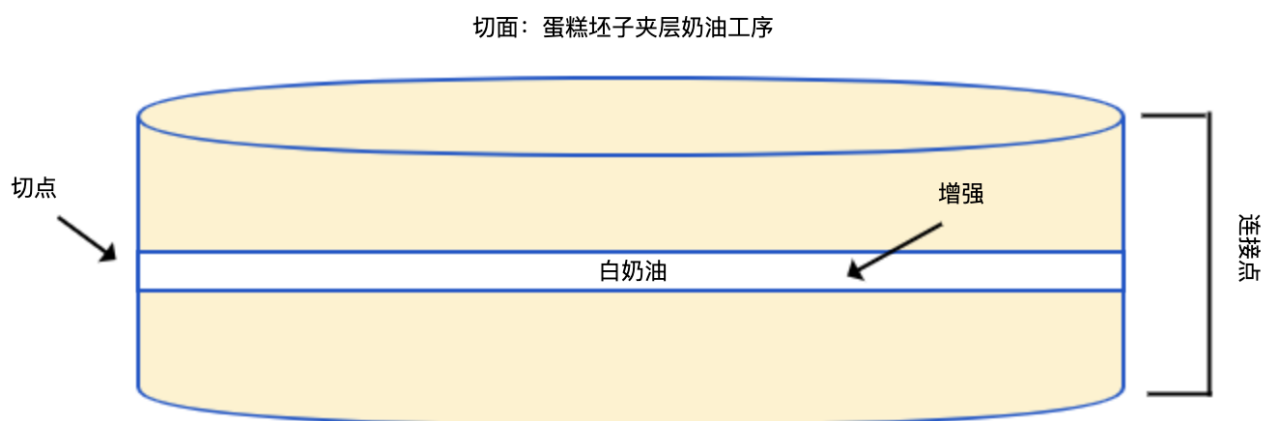
为了便于理解这两讲中的案例，我们先回顾下 IoC 和 AOP 的基础知识。

IoC，其实就是一种设计思想。使用 Spring 来实现 IoC，意味着将你设计好的对象交给 Spring 容器控制，而不是直接在对象内部控制。那，为什么要让容器来管理对象呢？或许你能想到的是，使用 IoC 方便、可以实现解耦。但在我看来，相比于这两个原因，更重要的是 IoC 带来了更多的可能性。

如果以容器为依托来管理所有的框架、业务对象，我们不仅可以无侵入地调整对象的关系，还可以无侵入地随时调整对象的属性，甚至是实现对象的替换。这就使得框架开发者在程序背后实现一些扩展不再是问题，带来的可能性是无限的。比如我们要监控的对象如果是 Bean，实现就会非常简单。所以，这套容器体系，不仅被 Spring Core 和 Spring Boot 大量依赖，还实现了一些外部框架和 Spring 的无缝整合。

AOP，体现了松耦合、高内聚的精髓，在切面集中实现横切关注点（缓存、权限、日志等），然后通过切点配置把代码注入合适的地方。切面、切点、增强、连接点，是 AOP 中非常重要的概念，也是我们这两讲会大量提及的。

为方便理解，我们把 Spring AOP 技术看作为蛋糕做奶油夹层的工序。如果我们希望找到一个合适的地方把奶油注入蛋糕胚子中，那应该如何指导工人完成操作呢？



首先，我们要提醒他，只能往蛋糕胚子里面加奶油，而不能上面或下面加奶油。这就是连接点（Join point），对于 Spring AOP 来说，连接点就是方法执行。

然后，我们要告诉他，在什么点切开蛋糕加奶油。比如，可以在蛋糕胚子中间加入一层奶油，在中间切一次；也可以在中间加两层奶油，在 1/3 和 2/3 的地方切两次。这就是

切点 (Pointcut) , Spring AOP 中默认使用 AspectJ 查询表达式, 通过在连接点运行查询表达式来匹配切入点。

接下来也是最重要的, 我们要告诉他, 切开蛋糕后要做什么, 也就是加入奶油。这就是增强 (Advice) , 也叫作通知, 定义了切入切点后增强的方式, 包括前、后、环绕等。Spring AOP 中, 把增强定义为拦截器。

最后, 我们要告诉他, 找到蛋糕胚子中要加奶油的地方并加入奶油。为蛋糕做奶油夹层的操作, 对 Spring AOP 来说就是切面 (Aspect) , 也叫作方面。切面 = 切点 + 增强。

好了, 理解了这几个核心概念, 我们就可以继续分析案例了。

我要首先说明的是, Spring 相关问题的复杂度比较复杂, 一方面是 Spring 提供的 IoC 和 AOP 本就灵活, 另一方面 Spring Boot 的自动装配、Spring Cloud 复杂的模块会让问题排查变得更复杂。因此, 今天这一讲, 我会带你先打好基础, 通过两个案例来重点聊聊 IoC 和 AOP; 然后, 我会在下一讲中与你分享 Spring 相关的坑。

单例的 Bean 如何注入 Prototype 的 Bean?

我们虽然知道 Spring 创建的 Bean 默认是单例的, 但当 Bean 遇到继承的时候, 可能会忽略这一点。为什么呢? 忽略这一点又会造成什么影响呢? 接下来, 我就和你分享一个由单例引起内存泄露的案例。

架构师一开始定义了这么一个 SayService 抽象类, 其中维护了一个类型是 ArrayList 的字段 data, 用于保存方法处理的中间数据。每次调用 say 方法都会往 data 加入新数据, 可以认为 SayService 是有状态, 如果 SayService 是单例的话必然会 OOM:

 复制代码


```
1 @Slf4j
2 public abstract class SayService {
3     List<String> data = new ArrayList<>();
4
5     public void say() {
6         data.add(IntStream.rangeClosed(1, 1000000)
7             .mapToObj(__ -> "a")
8             .collect(Collectors.joining("")) + UUID.randomUUID().toString());
9         log.info("I'm {} size:{}", this, data.size());
10    }
11 }
```

但实际开发的时候，开发同学没有过多思考就把 SayHello 和 SayBye 类加上了 @Service 注解，让它们成为了 Bean，也没有考虑到父类是有状态的：

 复制代码

```
1 @Service
2 @Slf4j
3 public class SayHello extends SayService {
4     @Override
5     public void say() {
6         super.say();
7         log.info("hello");
8     }
9 }
10
11 @Service
12 @Slf4j
13 public class SayBye extends SayService {
14     @Override
15     public void say() {
16         super.say();
17         log.info("bye");
18     }
19 }
```

许多开发同学认为，@Service 注解的意义在于，能通过 @Autowired 注解让 Spring 自动注入对象，就比如可以直接使用注入的 List 获取到 SayHello 和 SayBye，而没想过类的生命周期：

 复制代码

```
1 @Autowired
2 List<SayService> sayServiceList;
3
4 @GetMapping("test")
5 public void test() {
6     log.info("=====");
7     sayServiceList.forEach(SayService::say);
8 }
```

这一点非常容易忽略。开发基类的架构师将基类设计为有状态的，但并不知道子类是怎么使用基类的；而开发子类的同学，没多想就直接标记了 @Service，让类成为了 Bean，通

过 @Autowired 注解来注入这个服务。但这样设置后，有状态的基类就可能产生内存泄露或线程安全问题。


正确的方式是，在为类标记上 @Service 注解把类型交由容器管理前，首先评估一下类是否有状态，然后为 Bean 设置合适的 Scope。好在上线前，架构师发现了这个内存泄露问题，开发同学也做了修改，为 SayHello 和 SayBye 两个类都标记了 @Scope 注解，设置了 PROTOTYPE 的生命周期，也就是多例：

 复制代码

```
1 @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

但，上线后还是出现了内存泄漏，证明修改是无效的。

从日志可以看到，第一次调用和第二次调用的时候，SayBye 对象都是 4c0bfe9e，SayHello 也是一样的问题。从日志第 7 到 10 行还可以看到，第二次调用后 List 的元素个数变为了 2，说明父类 SayService 维护的 List 在不断增长，不断调用必然出现 OOM：

 复制代码

```
1 [15:01:09.349] [http-nio-45678-exec-1] [INFO ] [.s.d.BeanSingletonAndOrderCont
2 [15:01:09.401] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo1.SayService
3 [15:01:09.402] [http-nio-45678-exec-1] [INFO ] [t.commonmistakes.spring.demo1.
4 [15:01:09.469] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo1.SayService
5 [15:01:09.469] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo1.SayHello
6 [15:01:15.167] [http-nio-45678-exec-2] [INFO ] [.s.d.BeanSingletonAndOrderCont
7 [15:01:15.197] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.spring.demo1.SayService
8 [15:01:15.198] [http-nio-45678-exec-2] [INFO ] [t.commonmistakes.spring.demo1.
9 [15:01:15.224] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.spring.demo1.SayService
10 [15:01:15.224] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.spring.demo1.SayHello
```

这就引出了单例的 Bean 如何注入 Prototype 的 Bean 这个问题。Controller 标记了 @RestController 注解，而 @RestController 注解 = @Controller 注解 + @ResponseBody 注解，又因为 @Controller 标记了 @Component 元注解，所以 @RestController 注解其实也是一个 Spring Bean：

 复制代码

```
1 //@RestController注解=@Controller注解+@ResponseBody注解@Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
```

```

3  @Documented
4  @Controller
5  @ResponseBody
6  public @interface RestController {}
7
8  //@Controller又标记了@Component元注解
9  @Target({ElementType.TYPE})
10 @Retention(RetentionPolicy.RUNTIME)
11 @Documented
12 @Component
13 public @interface Controller {}

```

Bean 默认是单例的，所以单例的 Controller 注入的 Service 也是一次性创建的，即使 Service 本身标识了 prototype 的范围也没用。

修复方式是，让 Service 以代理方式注入。这样虽然 Controller 本身是单例的，但每次都能从代理获取 Service。这样一来，prototype 范围的配置才能真正生效：


 复制代码

```

1  @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE, proxyMode = ScopedProx

```

通过日志可以确认这种修复方式有效：

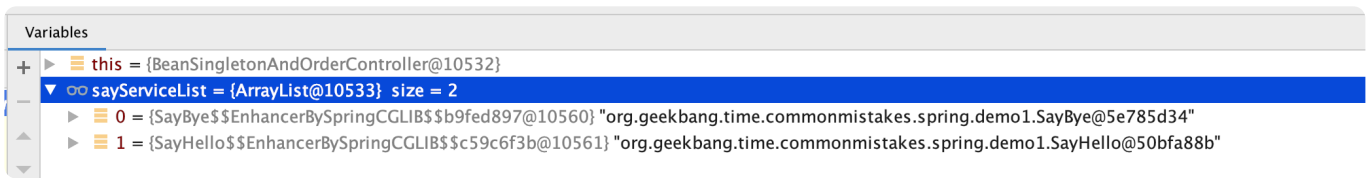
 复制代码

```

1  [15:08:42.649] [http-nio-45678-exec-1] [INFO ] [.s.d.BeanSingletonAndOrderCont
2  [15:08:42.747] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo1.SayService
3  [15:08:42.747] [http-nio-45678-exec-1] [INFO ] [t.commonmistakes.spring.demo1.
4  [15:08:42.871] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo1.SayService
5  [15:08:42.872] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo1.SayHello
6  [15:08:42.932] [http-nio-45678-exec-2] [INFO ] [.s.d.BeanSingletonAndOrderCont
7  [15:08:42.991] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.spring.demo1.SayService
8  [15:08:42.992] [http-nio-45678-exec-2] [INFO ] [t.commonmistakes.spring.demo1.
9  [15:08:43.046] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.spring.demo1.SayService
10 [15:08:43.046] [http-nio-45678-exec-2] [INFO ] [o.g.t.c.spring.demo1.SayHello

```

调试一下也可以发现，注入的 Service 都是 Spring 生成的代理类：



当然，如果不希望走代理的话还有一种方式是，每次直接从 `ApplicationContext` 中获取 Bean：

 复制代码

```
1 @Autowired
2 private ApplicationContext applicationContext;
3 @GetMapping("test2")
4 public void test2() {
5     applicationContext.getBeansOfType(SayService.class).values().forEach(SayService
6 }
```

如果细心的话，你可以发现另一个潜在的问题。这里 Spring 注入的 `SayService` 的 List，第一个元素是 `SayBye`，第二个元素是 `SayHello`。但，我们更希望的是先执行 Hello 再执行 Bye，所以注入一个 List Bean 时，需要进一步考虑 Bean 的顺序或者说优先级。

大多数情况下顺序并不是那么重要，但对于 AOP，顺序可能会引发致命问题。我们继续往下看这个问题吧。

监控切面因为顺序问题导致 Spring 事务失效

实现横切关注点，是 AOP 非常常见的一个应用。我曾看到过一个不错的 AOP 实践，通过 AOP 实现了一个整合日志记录、异常处理和方法耗时打点为一体的统一切面。但后来发现，使用了 AOP 切面后，这个应用的声明式事务处理居然都是无效的。你可以先回顾下 [第 6 讲](#)中提到的，Spring 事务失效的几种可能性。

现在我们来看下这个案例，分析下 AOP 实现的监控组件和事务失效有什么关系，以及通过 AOP 实现监控组件是否还有其他坑。

首先，定义一个自定义注解 `Metrics`，打上了该注解的方法可以实现各种监控功能：

 复制代码

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.METHOD, ElementType.TYPE})
```


```

3 public @interface Metrics {
4     /**
5      * 在方法成功执行后打点，记录方法的执行时间发送到指标系统，默认开启
6      *
7      * @return
8      */
9     boolean recordSuccessMetrics() default true;
10
11     /**
12      * 在方法成功失败后打点，记录方法的执行时间发送到指标系统，默认开启
13      *
14      * @return
15      */
16     boolean recordFailMetrics() default true;
17
18     /**
19      * 通过日志记录请求参数，默认开启
20      *
21      * @return
22      */
23     boolean logParameters() default true;
24
25     /**
26      * 通过日志记录方法返回值，默认开启
27      *
28      * @return
29      */
30     boolean logReturn() default true;
31
32     /**
33      * 出现异常后通过日志记录异常信息，默认开启
34      *
35      * @return
36      */
37     boolean logException() default true;
38
39     /**
40      * 出现异常后忽略异常返回默认值，默认关闭
41      *
42      * @return
43      */
44     boolean ignoreException() default false;
45 }

```

然后，实现一个切面完成 Metrics 注解提供的功能。这个切面可以实现标记了 @RestController 注解的 Web 控制器的自动切入，如果还需要对更多 Bean 进行切入的话，再自行标记 @Metrics 注解。

备注：这段代码有些长，里面还用到了一些小技巧，你需要仔细阅读代码中的注释。

 复制代码

```
1 @Aspect
2 @Component
3 @Slf4j
4 public class MetricsAspect {
5     //让Spring帮我们注入ObjectMapper，以方便通过JSON序列化来记录方法入参和出参
6
7     @Autowired
8     private ObjectMapper objectMapper;
9
10    //实现一个返回Java基本类型默认值的工具。其实，你也可以逐一写很多if-else判断类型，然后手
11    private static final Map<Class<?>, Object> DEFAULT_VALUES = Stream
12        .of(boolean.class, byte.class, char.class, double.class, float.class,
13            Integer.class, Long.class, Short.class, String.class)
14        .collect(toMap(clazz -> (Class<?>) clazz, clazz -> Array.get(Array
15            .of(clazz).toArray(), 0)));
16    public static <T> T getDefaultValue(Class<T> clazz) {
17        return (T) DEFAULT_VALUES.get(clazz);
18    }
19
20    //@annotation指示器实现对标记了Metrics注解的方法进行匹配
21    @Pointcut("within(@org.geekbang.time.commonmistakes.springpart1.aopmetrics.
22        *)")
23    public void withMetricsAnnotation() {}
24
25    //within指示器实现了匹配那些类型上标记了@RestController注解的方法
26    @Pointcut("within(@org.springframework.web.bind.annotation.RestController
27        *)")
28    public void controllerBean() {}
29
30    @Around("controllerBean() || withMetricsAnnotation()")
31    public Object metrics(ProceedingJoinPoint pjp) throws Throwable {
32        //通过连接点获取方法签名和方法上Metrics注解，并根据方法签名生成日志中要输出的方法定
33        MethodSignature signature = (MethodSignature) pjp.getSignature();
34        Metrics metrics = signature.getMethod().getAnnotation(Metrics.class);
35
36        String name = String.format("[%s] [%s]", signature.getDeclaringType().
37            getSimpleName(), signature.getMethod().getSimpleName());
38        //因为需要默认对所有@RestController标记的Web控制器实现@Metrics注解的功能，在这种
39        if (metrics == null) {
40            @Metrics
41            final class c {}
42            metrics = c.class.getAnnotation(Metrics.class);
43        }
44        //尝试从请求上下文（如果有的话）获得请求URL，以方便定位问题
45        RequestAttributes requestAttributes = RequestContextHolder.getRequestAttributes();
46        if (requestAttributes != null) {
47            HttpServletRequest request = ((ServletRequestAttributes) requestAttributes).getRequest();
48            if (request != null) {
49                name += String.format("[%s]", request.getRequestURL().toString());
50            }
51        }
52    }
53}
```

```

48         //实现的是入参的日志输出
49         if (metrics.logParameters())
50             log.info(String.format("【入参日志】调用 %s 的参数是: 【%s】", name, ob);
51         //实现连接点方法的执行, 以及成功失败的打点, 出现异常的时候还会记录日志
52         Object returnValue;
53         Instant start = Instant.now();
54         try {
55             returnValue = pjp.proceed();
56             if (metrics.recordSuccessMetrics())
57                 //在生产级代码中, 我们应考虑使用类似Micrometer的指标框架, 把打点信息记录:
58                 log.info(String.format("【成功打点】调用 %s 成功, 耗时: %d ms", nam
59         } catch (Exception ex) {
60             if (metrics.recordFailMetrics())
61                 log.info(String.format("【失败打点】调用 %s 失败, 耗时: %d ms", nam
62             if (metrics.logException())
63                 log.error(String.format("【异常日志】调用 %s 出现异常!", name), ex
64
65             //忽略异常的时候, 使用一开始定义的getDefaultValue方法, 来获取基本类型的默认值
66             if (metrics.ignoreException())
67                 returnValue = getDefaultValue(signature.getReturnType());
68             else
69                 throw ex;
70         }
71         //实现了返回值的日志输出
72         if (metrics.logReturn())
73             log.info(String.format("【出参日志】调用 %s 的返回是: 【%s】", name, re
74         return returnValue;
75     }
76 }

```

接下来, 分别定义最简单的 Controller、Service 和 Repository, 来测试 MetricsAspect 的功能。

其中, Service 中实现创建用户的时候做了事务处理, 当用户名包含 test 字样时会抛出异常, 导致事务回滚。同时, 我们为 Service 中的 createUser 标记了 @Metrics 注解。这样一来, 我们还可以手动为类或方法标记 @Metrics 注解, 实现 Controller 之外的其他组件的自动监控。

 复制代码

```

1  @Slf4j
2  @RestController //自动进行监控
3  @RequestMapping("metricstest")
4  public class MetricsController {
5      @Autowired
6      private UserService userService;


```

```

7     @GetMapping("transaction")
8     public int transaction(@RequestParam("name") String name) {
9         try {
10             userService.createUser(new UserEntity(name));
11         } catch (Exception ex) {
12             log.error("create user failed because {}", ex.getMessage());
13         }
14         return userService.getUserCount(name);
15     }
16 }
17
18 @Service
19 @Slf4j
20 public class UserService {
21     @Autowired
22     private UserRepository userRepository;
23     @Transactional
24     @Metrics //启用方法监控
25     public void createUser(UserEntity entity) {
26         userRepository.save(entity);
27         if (entity.getName().contains("test"))
28             throw new RuntimeException("invalid username!");
29     }
30
31     public int getUserCount(String name) {
32         return userRepository.findByName(name).size();
33     }
34 }
35
36 @Repository
37 public interface UserRepository extends JpaRepository<UserEntity, Long> {
38     List<UserEntity> findByName(String name);
39 }

```

使用用户名 “test” 测试一下注册功能：

 复制代码

```

1 [16:27:52.586] [http-nio-45678-exec-3] [INFO ] [o.g.t.c.spring.demo2.MetricsAs
2 [16:27:52.590] [http-nio-45678-exec-3] [INFO ] [o.g.t.c.spring.demo2.MetricsAs
3 [16:27:52.609] [http-nio-45678-exec-3] [INFO ] [o.g.t.c.spring.demo2.MetricsAs
4 [16:27:52.610] [http-nio-45678-exec-3] [ERROR] [o.g.t.c.spring.demo2.MetricsAs
5 java.lang.RuntimeException: invalid username!
6     at org.geekbang.time.commonmistakes.spring.demo2.UserService.createUser(User:
7     at org.geekbang.time.commonmistakes.spring.demo2.UserService$$FastClassBySpr
8 [16:27:52.614] [http-nio-45678-exec-3] [ERROR] [g.t.c.spring.demo2.MetricsCont
9 [16:27:52.617] [http-nio-45678-exec-3] [INFO ] [o.g.t.c.spring.demo2.MetricsAs
10 [16:27:52.618] [http-nio-45678-exec-3] [INFO ] [o.g.t.c.spring.demo2.MetricsAs

```

看起来这个切面很不错，日志中打出了整个调用的出入参、方法耗时：

第 1、8、9 和 10 行分别是 Controller 方法的入参日志、调用 Service 方法出错后记录的错误信息、成功执行的打点和出参日志。因为 Controller 方法内部进行了 try-catch 处理，所以其方法最终是成功执行的。出参日志中显示最后查询到的用户数量是 0，表示用户创建实际是失败的。


第 2、3 和 4~7 行分别是 Service 方法的入参日志、失败打点和异常日志。正是因为 Service 方法的异常抛到了 Controller，所以整个方法才能被 @Transactional 声明式事务回滚。在这里，MetricsAspect 捕获了异常又重新抛出，记录了异常的同时又不影响事务回滚。

一段时间后，开发同学觉得默认的 @Metrics 配置有点不合适，希望进行两个调整：

对于 Controller 的自动打点，不要自动记录入参和出参日志，否则日志量太大；


对于 Service 中的方法，最好可以自动捕获异常。

于是，他就为 MetricsController 手动加上了 @Metrics 注解，设置 logParameters 和 logReturn 为 false；然后为 Service 中的 createUser 方法的 @Metrics 注解，设置了 ignoreException 属性为 true：

 复制代码

```
1 @Metrics(logParameters = false, logReturn = false) //改动点1
2 public class MetricsController {
3
4     @Service
5     @Slf4j
6     public class UserService {
7         @Transactional
8         @Metrics(ignoreException = true) //改动点2
9         public void createUser(UserEntity entity) {
10             ...
11         }
12     }
13 }
```

代码上线后发现日志量并没有减少，更要命的是事务回滚失效了，从输出看到最后查询到了名为 test 的用户：

 复制代码

```
1 [17:01:16.549] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo2.MetricsAs|
2 [17:01:16.670] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo2.MetricsAs|
3 [17:01:16.885] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo2.MetricsAs|
4 [17:01:16.899] [http-nio-45678-exec-1] [ERROR] [o.g.t.c.spring.demo2.MetricsAs|
5 java.lang.RuntimeException: invalid username!
6     at org.geekbang.time.commonmistakes.spring.demo2.UserService.createUser(User:
7     at org.geekbang.time.commonmistakes.spring.demo2.UserService$$FastClassBySpr
8 [17:01:16.902] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo2.MetricsAs|
9 [17:01:17.466] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo2.MetricsAs|
10 [17:01:17.467] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo2.MetricsAs|
```

在介绍 [数据库事务](#) 时，我们分析了 Spring 通过 `TransactionAspectSupport` 类实现事务。在 `invokeWithinTransaction` 方法中设置断点可以发现，在执行 Service 的 `createUser` 方法时，`TransactionAspectSupport` 并没有捕获到异常，所以自然无法回滚事务。原因就是，**异常被 `MetricsAspect` 吃掉了**。


我们知道，切面本身是一个 Bean，Spring 对不同切面增强的执行顺序是由 Bean 优先级决定的，具体规则是：

入操作（Around（连接点执行前）、Before），切面优先级越高，越先执行。一个切面的入操作执行完，才轮到下一切面，所有切面入操作执行完，才开始执行连接点（方法）。

出操作（Around（连接点执行后）、After、AfterReturning、AfterThrowing），切面优先级越低，越先执行。一个切面的出操作执行完，才轮到下一切面，直到返回到调用点。

同一切面的 Around 比 After、Before 先执行。

对于 Bean 可以通过 `@Order` 注解来设置优先级，查看 `@Order` 注解和 `Ordered` 接口源码可以发现，默认情况下 Bean 的优先级为最低优先级，其值是 `Integer` 的最大值。其实，**值越大优先级反而越低，这点比较反直觉**：

 复制代码

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
3 @Documented
4 public @interface Order {
5
```

```

6     int value() default Ordered.LOWEST_PRECEDENCE;
7
8 }
9 public interface Ordered {
10     int HIGHEST_PRECEDENCE = Integer.MIN_VALUE;
11     int LOWEST_PRECEDENCE = Integer.MAX_VALUE;
12     int getOrder();
13 }

```

我们再通过一个例子，来理解下增强的执行顺序。新建一个 `TestAspectWithOrder10` 切面，通过 `@Order` 注解设置优先级为 10，在内部定义 `@Before`、`@After`、`@Around` 三类增强，三个增强的逻辑只是简单的日志输出，切点是 `TestController` 所有方法；然后再定义一个类似的 `TestAspectWithOrder20` 切面，设置优先级为 20：

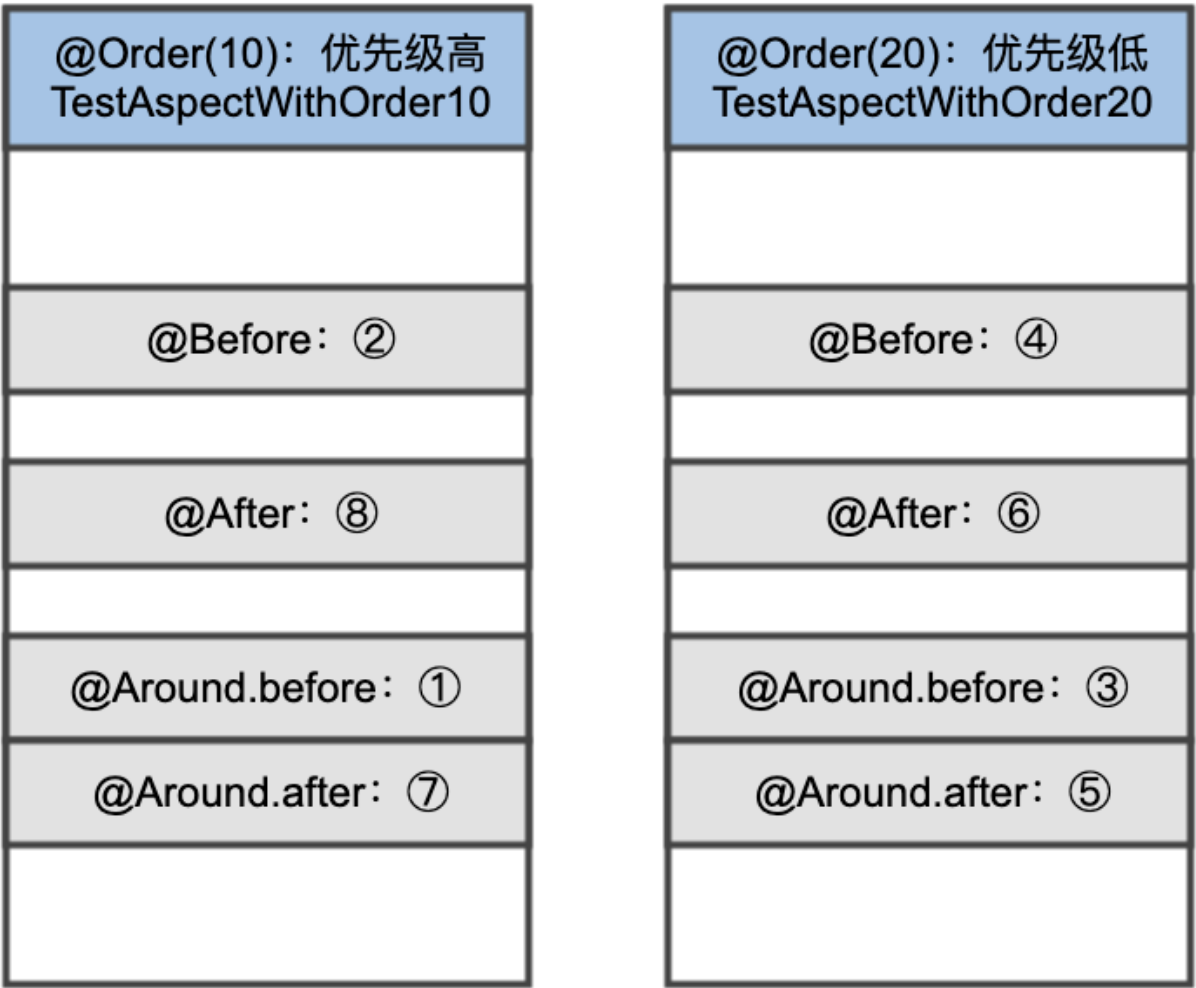
 复制代码

```

1  @Aspect
2  @Component
3  @Order(10)
4  @Slf4j
5  public class TestAspectWithOrder10 {
6      @Before("execution(* org.geekbang.time.commonmistakes.springpart1.aopmetric:
7          public void before(JoinPoint joinPoint) throws Throwable {
8          log.info("TestAspectWithOrder10 @Before");
9      }
10     @After("execution(* org.geekbang.time.commonmistakes.springpart1.aopmetric:
11         public void after(JoinPoint joinPoint) throws Throwable {
12         log.info("TestAspectWithOrder10 @After");
13     }
14     @Around("execution(* org.geekbang.time.commonmistakes.springpart1.aopmetric:
15         public Object around(ProceedingJoinPoint pjp) throws Throwable {
16         log.info("TestAspectWithOrder10 @Around before");
17         Object o = pjp.proceed();
18         log.info("TestAspectWithOrder10 @Around after");
19         return o;
20     }
21 }
22
23 @Aspect
24 @Component
25 @Order(20)
26 @Slf4j
27 public class TestAspectWithOrder20 {
28     ...
29 }

```

调用 TestController 的方法后，通过日志输出可以看到，增强执行顺序符合切面执行顺序的三个规则：



因为 Spring 的事务管理也是基于 AOP 的，默认情况下优先级最低也就是会先执行出操作，但是自定义切面 MetricsAspect 也同样是最低优先级，这个时候就可能出现异常：如果出操作先执行捕获了异常，那么 Spring 的事务处理就会因为无法捕获到异常导致无法回滚事务。

解决方式是，明确 MetricsAspect 的优先级，可以设置为最高优先级，也就是最先执行入操作最后执行出操作：

复制代码

```
1 //将MetricsAspect这个Bean的优先级设置为最高
2 @Order(Ordered.HIGHEST_PRECEDENCE)
3 public class MetricsAspect {
```



```
4     ...
5 }
```

此外，我们要知道切入的连接点是方法，注解定义在类上是无法直接从方法上获取到注解的。修复方式是，改为优先从方法获取，如果获取不到再从类获取，如果还是获取不到再使用默认的注解：

 复制代码

```
1 Metrics metrics = signature.getMethod().getAnnotation(Metrics.class);
2 if (metrics == null) {
3     metrics = signature.getMethod().getDeclaringClass().getAnnotation(Metrics.class);
4 }
```

经过这 2 处修改，事务终于又可以回滚了，并且 Controller 的监控日志也不再出现入参、出参信息。

我再总结下这个案例。利用反射 + 注解 + Spring AOP 实现统一的横切日志关注点时，我们遇到的 Spring 事务失效问题，是由自定义的切面执行顺序引起的。这也让我们认识到，因为 Spring 内部大量利用 IoC 和 AOP 实现了各种组件，当使用 IoC 和 AOP 时，一定要考虑是否会影响其他内部组件。

重点回顾

今天，我通过 2 个案例和你分享了 Spring IoC 和 AOP 的基本概念，以及三个比较容易出错的点。

第一，让 Spring 容器管理对象，要考虑对象默认的 Scope 单例是否适合，对于有状态的类型，单例可能产生内存泄露问题。

第二，如果要为单例的 Bean 注入 Prototype 的 Bean，绝不是仅仅修改 Scope 属性这么简单。由于单例的 Bean 在容器启动时就会完成一次性初始化。最简单的解决方案是，把 Prototype 的 Bean 设置为通过代理注入，也就是设置 proxyMode 属性为 TARGET_CLASS。

第三，如果一组相同类型的 Bean 是有顺序的，需要明确使用 @Order 注解来设置顺序。你可以再回顾下，两个不同优先级切面中 @Before、@After 和 @Around 三种增强的执行顺序，是什么样的。

最后我要说的是，文内第二个案例是一个完整的统一日志监控案例，继续修改就可以实现一个完善的、生产级的方法调用监控平台。这些修改主要是两方面：把日志打点，改为对接 Metrics 监控系统；把各种功能的监控开关，从注解属性获取改为通过配置系统实时获取。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

思考与讨论

1. 除了通过 @Autowired 注入 Bean 外，还可以使用 @Inject 或 @Resource 来注入 Bean。你知道这三种方式的区别是什么吗？
2. 当 Bean 产生循环依赖时，比如 BeanA 的构造方法依赖 BeanB 作为成员需要注入，BeanB 也依赖 BeanA，你觉得会出现什么问题呢？又有哪些解决方式呢？

在下一讲中，我会继续与你探讨 Spring 核心的其他问题。我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把今天的内容分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你
攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 18 | 当反射、注解和泛型遇到OOP时，会有哪些坑？

下一篇 20 | Spring框架：框架帮我们做了很多工作也带来了复杂度

精选留言 (7)

写留言



Darren 置顶

2020-04-26

一、注解区别

@Autowired

1、@Autowired是spring自带的注解，通过 'AutowiredAnnotationBeanPostProcessor' 类实现的依赖注入；

2、@Autowired是根据类型进行自动装配的，如果需要按名称进行装配，则需要配合...
展开

作者回复:



3



norman

2020-04-25

@Resource 和 @Autowired @Inject 三者区别：

1 @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

2 @Autowired默认是按照类型装配注入的，如果想按照名称来转配注入，则需要结合@Qualifier。这个注释是Spring特有的。...

展开

作者回复: ，也可以参考 <https://stackoverflow.com/questions/20450902/inject-and-resource-and-autowired-annotations> 这里的回复



3



Husiun

2020-04-25

问题2，循环依赖会抛出异常BeanCurrentlyInCreationException，官网的解决方案是由构造器注入改为setter注入

展开 ∨



👍 2



W

2020-04-25

MetricsAspect 这个类里面的小技巧学到了

展开 ∨



👍 1



左琪

2020-04-26

这里的代理类不是单例么，还是说会在增强逻辑里不断创建被代理类？

作者回复: 代理类会来判断是否需要创建新的对象



Joker

2020-04-25

老师，请教一下，那个sayservice里的data有啥用，那个单例是为了一种重复使用data对吧，那换成每次都生成一个新的bean，那个data还有效果吗。。

展开 ∨

作者回复: 只是为了模拟SayService是有状态



Demon.Lee

2020-04-25

连接点: 程序执行过程中能够应用通知的所有点；通知（增强）：即切面的工作，定义了What以及When；切点定义了Where，通知被应用的具体位置（哪些连接点）

----Spring实战（第4版）

展开 ∨

作者回复: 不错



