



06 | Spring AOP常见错误（下）

2021-05-03 傅健

Spring编程常见错误50例

[进入课程 >](#)



讲述：傅健

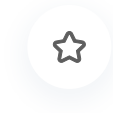
时长 15:55 大小 14.58M



你好，我是傅健。


上一节课，我们介绍了 Spring AOP 常遇到的几个问题，通过具体的源码解析，相信你对 Spring AOP 的基本原理已经有所了解了。不过，AOP 毕竟是 Spring 的核心功能之一，不可能规避那零散的两三个问题就一劳永逸了。所以这节课，我们继续聊聊 Spring AOP 中还会有哪些易错点。实际上，当一个系统采用的切面越来越多时，因为执行顺序而导致的问题便会逐步暴露出来，下面我们就重点看一下。

案例 1：错乱混合不同类型的增强




还是沿用上节课的宿舍管理系统开发场景。

这里我们先回顾下，你就不用去翻代码了。这个宿舍管理系统保护了一个电费充值模块，它包含了一个负责电费充值的类 `ElectricService`，还有一个充电方法 `charge()`：

 复制代码

```
1 @Service
2 public class ElectricService {
3     public void charge() throws Exception {
4         System.out.println("Electric charging ...");
5     }
6 }
```

为了在执行 `charge()` 之前，鉴定下调用者的权限，我们增加了针对于 `Electric` 的切面类 `AopConfig`，其中包含一个 `@Before` 增强。这里的增强没有做任何事情，仅仅是打印了一行日志，然后模拟执行权限校验功能（占用 1 秒钟）。

 复制代码

```
1 //省略 imports
2 @Aspect
3 @Service
4 @Slf4j
5 public class AspectService {
6     @Before("execution(* com.spring.puzzle.class6.example1.ElectricService.charg
7     public void checkAuthority(JoinPoint pjp) throws Throwable {
8         System.out.println("validating user authority");
9         Thread.sleep(1000);
10    }
11 }
```

执行后，我们得到以下 log，接着一切按照预期继续执行：

 复制代码

```
1 validating user authority
2 Electric charging ...
```

一段时间后，由于业务发展，`ElectricService` 中的 `charge()` 逻辑变得更加复杂了，我们需要仅仅针对 `ElectricService` 的 `charge()` 做性能统计。为了不影响原有的业务逻辑，我们在 `AopConfig` 中添加了另一个增强，代码更改后如下：

```
1 //省略 imports
2 @Aspect
3 @Service
4 public class AopConfig {
5     @Before("execution(* com.spring.puzzle.class6.example1.ElectricService.cha
6     public void checkAuthority(JoinPoint pjp) throws Throwable {
7         System.out.println("validating user authority");
8         Thread.sleep(1000);
9     }
10
11     @Around("execution(* com.spring.puzzle.class6.example1.ElectricService.cha
12     public void recordPerformance(ProceedingJoinPoint pjp) throws Throwable {
13         long start = System.currentTimeMillis();
14         pjp.proceed();
15         long end = System.currentTimeMillis();
16         System.out.println("charge method time cost: " + (end - start));
17     }
18 }
```

执行后得到日志如下：

```
validating user authority
Electric charging ...
charge method time cost 1022 (ms)
```

通过性能统计打印出的日志，我们可以得知 `charge()` 执行时间超过了 1 秒钟。然而，该方法仅打印了一行日志，它的执行不可能需要这么长时间。

因此我们很容易看出问题所在：当前 `ElectricService` 中 `charge()` 的执行时间，包含了权限验证的时间，即包含了通过 `@Around` 增强的 `checkAuthority()` 执行的所有时间。这并不符合我们的初衷，我们需要统计的仅仅是 `ElectricService.charge()` 的性能统计，它并不包含鉴权过程。

当然，这些都是从日志直接观察出的现象。实际上，这个问题出现的根本原因和 AOP 的执行顺序有关。针对这个案例而言，当同一个切面（Aspect）中同时包含多个不同类型的增强时（Around、Before、After、AfterReturning、AfterThrowing 等），它们的执行是有顺序的。那么顺序如何？我们不妨来解析下。

案例解析

其实一切都可以从源码中得到真相！在🔗第 04 课我们曾经提到过，Spring 初始化单例类的一般过程，基本都是 `getBean()->doGetBean()->getSingleton()`，如果发现 Bean 不存在，则调用 `createBean()->doCreateBean()` 进行实例化。

而如果我们的代码里使用了 Spring AOP，`doCreateBean()` 最终会返回一个代理对象。至于代理对象如何创建，大体流程我们在上一讲已经概述过了。如果你记忆力比较好的话，应该记得在代理对象的创建过程中，我们贴出过这样一段代码（参考 `AbstractAutoProxyCreator#createProxy`）：

[复制代码](#)

```
1  protected Object createProxy(Class<?> beanClass, @Nullable String beanName,
2      @Nullable Object[] specificInterceptors, TargetSource targetSource) {
3      //省略非关键代码
4      Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
5      proxyFactory.addAdvisors(advisors);
6      proxyFactory.setTargetSource(targetSource);
7      //省略非关键代码
8      return proxyFactory.getProxy(getProxyClassLoader());
9  }
```

其中 `advisors` 就是增强方法对象，它的顺序决定了面临多个增强时，到底先执行谁。而这个集合对象本身是由 `specificInterceptors` 构建出来的，而 `specificInterceptors` 又是由 `AbstractAdvisorAutoProxyCreator#getAdvicesAndAdvisorsForBean` 方法构建：

[复制代码](#)

```
1  @Override
2  @Nullable
3  protected Object[] getAdvicesAndAdvisorsForBean(
4      Class<?> beanClass, String beanName, @Nullable TargetSource targetSource
5      List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
6      if (advisors.isEmpty()) {
7          return DO_NOT_PROXY;
8      }
9      return advisors.toArray();
10 }
```

简单说，其实就是根据当前的 `beanClass`、`beanName` 等信息，结合所有候选的 `advisors`，最终找出匹配（Eligible）的 `Advisor`，为什么如此？毕竟 AOP 拦截点可能会

配置多个，而我们执行的方法不见得会被所有的拦截配置拦截。寻找匹配 Advisor 的逻辑参考 AbstractAdvisorAutoProxyCreator#findEligibleAdvisors:

[复制代码](#)

```
1  protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanNa
2      //寻找候选的 Advisor
3      List<Advisor> candidateAdvisors = findCandidateAdvisors();
4      //根据候选的 Advisor 和当前 bean 算出匹配的 Advisor
5      List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors
6      extendAdvisors(eligibleAdvisors);
7      if (!eligibleAdvisors.isEmpty()) {
8          //排序
9          eligibleAdvisors = sortAdvisors(eligibleAdvisors);
10     }
11     return eligibleAdvisors;
12 }
```

通过研读代码，最终 Advisors 的顺序是由两点决定：


1. candidateAdvisors 的顺序；
2. sortAdvisors 进行的排序。

这里我们可以重点看下对本案例起关键作用的 candidateAdvisors 排序。实际上，它的顺序是在 @Aspect 标记的 AopConfig Bean 构建时就决定了。具体而言，就是在初始化过程中会排序自己配置的 Advisors，并把排序结果存入了缓存（BeanFactoryAspectJAdvisorsBuilder#advisorsCache）。

后续 Bean 创建代理时，直接拿出这个排序好的候选 Advisors。候选 Advisors 排序发生在 Bean 构建这个结论时，我们也可以通过 AopConfig Bean 构建中的堆栈信息验证：

```
sort:1462, ArrayList (java.util)
getAdvisorMethods:157, ReflectiveAspectJAdvisorFactory (org.springframework.aop.aspectj.annotation)
getAdvisors:125, ReflectiveAspectJAdvisorFactory (org.springframework.aop.aspectj.annotation)
buildAspectJAdvisors:110, BeanFactoryAspectJAdvisorsBuilder (org.springframework.aop.aspectj.annotation)
findCandidateAdvisors:95, AnnotationAwareAspectJAutoProxyCreator (org.springframework.aop.aspectj.annotation)
shouldSkip:101, AspectJAwareAdvisorAutoProxyCreator (org.springframework.aop.aspectj.autoproxy)
postProcessBeforeInstantiation:251, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
applyBeanPostProcessorsBeforeInstantiation:1141, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
resolveBeforeInstantiation:1114, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
createBean:506, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
```


可以看到，排序是在 Bean 的构建中进行的，而最后排序执行的关键代码位于下面的方法中（参考 ReflectiveAspectJAdvisorFactory#getAdvisorMethods）：

 复制代码

```
1 private List<Method> getAdvisorMethods(Class<?> aspectClass) {
2     final List<Method> methods = new ArrayList<>();
3     ReflectionUtils.doWithMethods(aspectClass, method -> {
4         // Exclude pointcuts
5         if (AnnotationUtils.getAnnotation(method, Pointcut.class) == null) {
6             methods.add(method);
7         }
8     }, ReflectionUtils.USER_DECLARED_METHODS);
9     // 排序
10    methods.sort(METHOD_COMPARATOR);
11    return methods;
12 }
```

上述代码的重点是第九行 `methods.sort(METHOD_COMPARATOR)` 方法。

我们来查看 `METHOD_COMPARATOR` 的代码，会发现它是定义在 `ReflectiveAspectJAdvisorFactory` 类中的静态方法块，代码如下：

 复制代码

```
1 static {
2     Comparator<Method> adviceKindComparator = new ConvertingComparator<>(
3         new InstanceComparator<>(
4             Around.class, Before.class, After.class, AfterReturning.class,
5             (Converter<Method, Annotation>) method -> {
6                 AspectJAnnotation<?> annotation =
7                     AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod(method,
8                         return (annotation != null ? annotation.getAnnotation() : null);
9             });
10    Comparator<Method> methodNameComparator = new ConvertingComparator<>(Method
11        //合并上面两者比较器
12    METHOD_COMPARATOR = adviceKindComparator.thenComparing(methodNameComparator
13 }
```

`METHOD_COMPARATOR` 本质上是一个连续比较器，由 `adviceKindComparator` 和 `methodNameComparator` 这两个比较器通过 `thenComparing()` 连接而成。

通过这个案例，我们重点了解 `adviceKindComparator` 这个比较器，此对象通过实例化 `ConvertingComparator` 类而来，而 `ConvertingComparator` 类是 Spring 中较为经典的一个实现。顾名思义，先转化再比较，它构造参数接受以下这两个参数：

第一个参数是基准比较器，即在 `adviceKindComparator` 中最终要调用的比较器，在构造函数中赋值于 `this.comparator`;

第二个参数是一个 `lambda` 回调函数，用来将传递的参数转化为基准比较器需要的参数类型，在构造函数中赋值于 `this.converter`。

查看 `ConvertingComparator` 比较器核心方法 `compare` 如下：

[复制代码](#)

```
1 public int compare(S o1, S o2) {
2     T c1 = this.converter.convert(o1);
3     T c2 = this.converter.convert(o2);
4     return this.comparator.compare(c1, c2);
5 }
```

可知，这里是先调用从构造函数中获取到的 `lambda` 回调函数 `this.converter`，将需要比较的参数进行转化。我们可以从之前的代码中找出这个转化工作：

[复制代码](#)

```
1 (Converter<Method, Annotation>) method -> {
2     AspectJAnnotation<?> annotation =
3         AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod(method);
4     return (annotation != null ? annotation.getAnnotation() : null);
5 };
```

转化功能的代码逻辑较为简单，就是返回传入方法（`method`）上标记的增强注解（`Pointcut`, `Around`, `Before`, `After`, `AfterReturning` 以及 `AfterThrowing`）：

[复制代码](#)

```
1 private static final Class<?>[] ASPECTJ_ANNOTATION_CLASSES = new Class<?>[] {
2     Pointcut.class, Around.class, Before.class, After.class, AfterReturning.
3
4     protected static AspectJAnnotation<?> findAspectJAnnotationOnMethod(Method met
5         for (Class<?> clazz : ASPECTJ_ANNOTATION_CLASSES) {
6             AspectJAnnotation<?> foundAnnotation = findAnnotation(method, (Class<Ann
7                 if (foundAnnotation != null) {
8                     return foundAnnotation;
9                 }
10            }
11            return null;
12        }
```

经过转化后，我们获取到的待比较的数据其实就是注解了。而它们的排序依赖于 `ConvertingComparator` 的第一个参数，即最终会调用的基准比较器，以下是它的关键实现代码：

[复制代码](#)

```
1 new InstanceComparator<>(  
2     Around.class, Before.class, After.class, AfterReturning.class, AfterThro
```

最终我们要调用的基准比较器本质上就是一个 `InstanceComparator` 类，我们先重点注意下这几个增强注解的传递顺序。继续查看它的构造方法如下：

[复制代码](#)

```
1 public InstanceComparator(Class<?>... instanceOrder) {  
2     Assert.notNull(instanceOrder, "'instanceOrder' array must not be null");  
3     this.instanceOrder = instanceOrder;  
4 }
```

构造方法也是较为简单的，只是将传递进来的 `instanceOrder` 赋予了类成员变量，继续查看 `InstanceComparator` 比较器核心方法 `compare` 如下，也就是最终要调用的比较方法：

[复制代码](#)

```
1 public int compare(T o1, T o2) {  
2     int i1 = getOrder(o1);  
3     int i2 = getOrder(o2);  
4     return (i1 < i2 ? -1 : (i1 == i2 ? 0 : 1));  
5 }
```

一个典型的 `Comparator`，代码逻辑按照 `i1`、`i2` 的升序排列，即 `getOrder()` 返回的值越小，排序越靠前。

查看 `getOrder()` 的逻辑如下：


```
1 private int getOrder(@Nullable T object) {  
2     if (object != null) {  
3         for (int i = 0; i < this.instanceOrder.length; i++) {  
4             //instance 在 instanceOrder 中的“排号”  
5             if (this.instanceOrder[i].isInstance(object)) {  
6                 return i;  
7             }  
8         }  
9     }  
10    return this.instanceOrder.length;  
11 }
```

[复制代码](#)

返回当前传递的增强注解在 `this.instanceOrder` 中的序列值，序列值越小，则越靠前。而结合之前构造参数传递的顺序，我们很快就能判断出：最终的排序结果依次是 `Around.class`, `Before.class`, `After.class`, `AfterReturning.class`, `AfterThrowing.class`。

到此为止，答案也呼之欲出：`this.instanceOrder` 的排序，即为不同类型增强的优先级，**排序越靠前，优先级越高**。

结合之前的讨论，我们可以得出一个结论：同一个切面中，不同类型的增强方法被调用的顺序依次为 `Around.class`, `Before.class`, `After.class`, `AfterReturning.class`, `AfterThrowing.class`。


问题修正

从上述案例解析中，我们知道 `Around` 类型的增强被调用的优先级高于 `Before` 类型的增强，所以上述案例中性能统计所花费的时间，包含权限验证的时间，也在情理之中。

知道了原理，修正起来也就简单了。假设不允许我们去拆分类，我们可以按照下面的思路来修改：


1. 将 `ElectricService.charge()` 的业务逻辑全部移动到 `doCharge()`，在 `charge()` 中调用 `doCharge()`；
2. 性能统计只需要拦截 `doCharge()`；
3. 权限统计增强保持不变，依然拦截 `charge()`。

`ElectricService` 类代码更改如下：

 复制代码

```
1 @Service
2 public class ElectricService {
3
4     public void charge() {
5         doCharge();
6     }
7     public void doCharge() {
8         System.out.println("Electric charging ...");
9     }
10 }
```

切面代码更改如下：


 复制代码

```
1 //省略 imports
2 @Aspect
3 @Service
4 public class AopConfig {
5     @Before("execution(* com.spring.puzzle.class6.example1.ElectricService.cha
6     public void checkAuthority(JoinPoint pjp) throws Throwable {
7         System.out.println("validating user authority");
8         Thread.sleep(1000);
9     }
10
11     @Around("execution(* com.spring.puzzle.class6.example1.ElectricService.doC
12     public void recordPerformance(ProceedingJoinPoint pjp) throws Throwable {
13         long start = System.currentTimeMillis();
14         pjp.proceed();
15         long end = System.currentTimeMillis();
16         System.out.println("charge method time cost: " + (end - start));
17     }
18 }
```

案例 2：错乱混合同类型增强

那学到这里，你可能还有疑问，如果同一个切面里的多个增强方法其增强都一样，那调用顺序又如何呢？我们继续看下一个案例。

这里业务逻辑类 `ElectricService` 没有任何变化，仅包含一个 `charge()`：


 复制代码

```
1 import org.springframework.stereotype.Service;
```

```
2 @Service
3 public class ElectricService {
4     public void charge() {
5         System.out.println("Electric charging ...");
6     }
7 }
```

切面类 AspectService 包含两个方法，都是 Before 类型增强。

第一个方法 logBeforeMethod(), 目的是在 run() 执行之前希望能输入日志，表示当前方法被调用一次，方便后期统计。另一个方法 validateAuthority(), 目的是做权限验证，其作用是在调用此方法之前做权限验证，如果不符合权限限制要求，则直接抛出异常。这里为了方便演示，此方法将直接抛出异常：

 复制代码

```
1 //省略 imports
2 @Aspect
3 @Service
4 public class AopConfig {
5     @Before("execution(* com.spring.puzzle.class5.example2.ElectricService.charge
6     public void logBeforeMethod(JoinPoint pjp) throws Throwable {
7         System.out.println("step into ->" + pjp.getSignature());
8     }
9     @Before("execution(* com.spring.puzzle.class5.example2.ElectricService.charge
10    public void validateAuthority(JoinPoint pjp) throws Throwable {
11        throw new RuntimeException("authority check failed");
12    }
13 }
```

我们对代码的执行预期为：当鉴权失败时，由于 ElectricService.charge() 没有被调用，那么 run() 的调用日志也不应该被输出，即 logBeforeMethod() 不应该被调用，但事实总是出乎意料，执行结果如下：

```
step into ->void com.spring.puzzle.class6.example2.Electric.charge()
Exception in thread "main" java.lang.RuntimeException: authority check failed
```

虽然鉴权失败，抛出了异常且 ElectricService.charge() 没有被调用，但是 logBeforeMethod() 的调用日志却被输出了，这将导致后期针对于 ElectricService.charge() 的调用数据统计严重失真。

这里我们就需要搞清楚一个问题：当同一个切面包含多个同一种类型的多个增强，且修饰的都是同一个方法时，这多个增强的执行顺序是怎样的？

案例解析

我们继续从源代码中寻找真相！你应该还记得上述代码中，定义 `METHOD_COMPARATOR` 的静态代码块吧。

`METHOD_COMPARATOR` 本质是一个连续比较器，而上个案例中我们仅仅只看了第一个比较器，细心的你肯定发现了这里还有第二个比较器 `methodNameComparator`，任意两个比较器都可以通过其内置的 `thenComparing()` 连接形成一个连续比较器，从而可以让我们按照比较器的连接顺序依次比较：

[复制代码](#)

```
1 static {
2     //第一个比较器，用来按照增强类型排序
3     Comparator<Method> adviceKindComparator = new ConvertingComparator<>(
4         new InstanceComparator<>(
5             Around.class, Before.class, After.class, AfterReturning.class,
6             (Converter<Method, Annotation>) method -> {
7                 AspectJAnnotation<?> annotation =
8                     AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod(met
9                 return (annotation != null ? annotation.getAnnotation() : null);
10            })
11    //第二个比较器，用来按照方法名排序
12    Comparator<Method> methodNameComparator = new ConvertingComparator<>(Method
13    METHOD_COMPARATOR = adviceKindComparator.thenComparing(methodNameComparator
14 }
```

我们可以看到，在第 12 行代码中，第 2 个比较器 `methodNameComparator` 依然使用的是 `ConvertingComparator`，传递了方法名作为参数。我们基本可以猜测出该比较器是按照方法名进行排序的，这里可以进一步查看构造器方法及构造器调用的内部 `comparable()`：

[复制代码](#)

```
1 public ConvertingComparator(Converter<S, T> converter) {
2     this(Comparators.comparable(), converter);
3 }
4 // 省略非关键代码
5 public static <T> Comparator<T> comparable() {
6     return ComparableComparator.INSTANCE;
```

```
7 }
```

上述代码中的 ComparableComparator 实例其实极其简单，代码如下：

[复制代码](#)

```
1 public class ComparableComparator<T extends Comparable<T>> implements Comparat
2
3     public static final ComparableComparator INSTANCE = new ComparableComparato
4
5     @Override
6     public int compare(T o1, T o2) {
7         return o1.compareTo(o2);
8     }
9 }
```

答案和我们的猜测完全一致，methodNameComparator 最终调用了 String 类自身的 compareTo()，代码如下：

[复制代码](#)

```
1 public int compareTo(String anotherString) {
2     int len1 = value.length;
3     int len2 = anotherString.value.length;
4     int lim = Math.min(len1, len2);
5     char v1[] = value;
6     char v2[] = anotherString.value;
7
8     int k = 0;
9     while (k < lim) {
10         char c1 = v1[k];
11         char c2 = v2[k];
12         if (c1 != c2) {
13             return c1 - c2;
14         }
15         k++;
16     }
17     return len1 - len2;
18 }
```

到这，答案揭晓：如果两个方法名长度相同，则依次比较每一个字母的 ASCII 码，ASCII 码越小，排序越靠前；若长度不同，且短的方法名字符串是长的子集时，短的排序靠前。

问题修正

从上述分析我们得知，在同一个切面配置类中，针对同一个方法存在多个同类型增强时，其执行顺序仅和当前增强方法的名称有关，而不是由谁代码在先、谁代码在后来决定。了解了这点，我们就可以直接通过调整方法名的方式来修正程序：

[复制代码](#)

```
1 //省略 imports
2 @Aspect
3 @Service
4 public class AopConfig {
5     @Before("execution(* com.spring.puzzle.class6.example2.ElectricService.charg
6     public void logBeforeMethod(JoinPoint pjp) throws Throwable {
7         System.out.println("step into ->" + pjp.getSignature());
8     }
9     @Before("execution(* com.spring.puzzle.class6.example2.ElectricService.charg
10    public void checkAuthority(JoinPoint pjp) throws Throwable {
11        throw new RuntimeException("authority check failed");
12    }
13 }
```

我们可以将原来的 `validateAuthority()` 改为 `checkAuthority()`，这种情况下，**对增强 (Advisor) 的排序，其实最后就是在比较字符 l 和 字符 c**。显而易见，`checkAuthority()` 的排序会靠前，从而被优先执行，最终问题得以解决。

重点回顾

通过学习这两个案例，相信你对 Spring AOP 增强方法的执行顺序已经有了较为深入的理解。这里我来总结下关键点：

在同一个切面配置中，如果存在多个不同类型的增强，那么其执行优先级是按照增强类型的特定顺序排列，依次为增强类型为 `Around.class`, `Before.class`, `After.class`, `AfterReturning.class`, `AfterThrowing.class`;

在同一个切面配置中，如果存在多个相同类型的增强，那么其执行优先级是按照该增强的方法名排序，排序方式依次为比较方法名的每一个字母，直到发现第一个不相同且 ASCII 码较小的字母。

同时，这节课我们也拓展了一些比较器相关的知识：

任意两个比较器（Comparator）可以通过 `thenComparing()` 连接合成一个新的连续比较器；

比较器的比较规则有一个简单的方法可以帮助你理解，就是最终一定需要对象两两比较，而比较的过程一定也是比较这两个对象的同种属性。你只要抓住这两点：比较了什么属性以及比较的结果是什么就可以了，若比较结果为正数，则按照该属性的升序排列；若为负数，则按属性降序排列。

思考题

实际上，审阅上面两个案例的修正方案，你会发现它们虽然改动很小，但是都还不够优美。那么有没有稍微优美点的替代方案呢？如果有，你知道背后的原理及关键源码吗？顺便你也可以想想，我为什么没有用更优美的方案呢？

期待在留言区看到你的思考，我们下节课再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 05 | Spring AOP常见错误（上）

下一篇 07 | Spring事件常见错误

精选留言（1）

写留言



哦吼掉了

2021-05-06

思考题：切面实现Order接口或者增加@Ordered注解

AspectJAwareAdvisorAutoProxyCreator#sortAdvisors -->

AnnotationAwareOrderComparator.sort(advisors) AnnotationAwareOrderComparator

...

展开

