

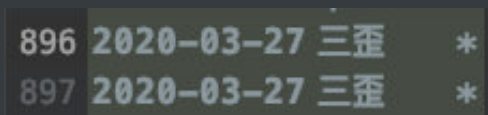
点赞再看，养成习惯，微信搜索【三太子敖丙】关注这个互联网苟且偷生的工具人。

本文 **GitHub** <https://github.com/JavaFamily> 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

背景

某天运营反馈，点了一次保存，但是后台出现了3条数据，我当时就想，不应该啊，这代码我几万年没动了，我当时就叫他先别操作了，保留一下现场，我去排查一下。

我看了下新增的代码，直接右键查看作者



896	2020-03-27	三歪	*
897	2020-03-27	三歪	*

没想到三歪做过改动，我就去问三歪，XX模块的新增代码你是不是动过？

他沉默了很久没说话，然后抓起桌子上用剩下来的纸擦了擦鬓角留下的汗水，咽了一下口水说，是的我改过，我把之前dubbo的xml配置方式改成了注解的方式。

怎么了？现在出BUG了？

你呀你，下次这种改动跟我说一下，我估计是dubbo源码的bug吧，不要慌，让我去看看什么问题。

正文

其实dubbo配置的方式有很多种，大家用的最多的就是xml配置的方式，如果不需要重试次数，我们会加上重试次数为0，因为他默认是有多次的。

```
<dubbo:reference id="testService"
interface="heidea.trade.service.sdk.interfice.TestService" retries="0"/>
```

或者使用注解的方式

```
@Reference(retries =0)
```

其实我已经大概知道是什么原因了，但是为了证实自己的猜想，于是开启了接下来的debug之旅~~~

注：dubbo版本：2.6.2

首先是在采用@Reference注解条件下：

```
@Reference(retries = 0, timeout = 1)
```

采用@Reference注解配置重试次数

首先是都找到了dubbo重试的代码位置（启动dubbo项目，到调用接口时，F5进入方法，会跳转到InvokerInvocationHandler中的invoke方法中，继续跟踪进入MockClusterInvoker中的invoke方法，然后进入AbstractClusterInvoker中的invoke方法中，这里主要是拿到配置的负载均衡策略，后面会到FailoverClusterInvoker的doInvoke方法中）。

重点来了，这里会获取配置的retries值，可以看到上面配置的是0，但是取出来居然是null，如图：

```
public int getMethodParameter(String method, String key, int defaultValue) {  
    String methodKey = method + "." + key; methodKey: "getAge.retries"  
    Number n = getNumbers().get(methodKey); n: null methodKey: "getAge.retries"  
    if (n != null) {  
        return n.intValue(); n: null  
    }  
    String value = getMethodParameter(method, key); value: null method: "getAge"  
    if (value == null || value.length() == 0) { value: null  
        return defaultValue;  
    }  
    int i = Integer.parseInt(value);  
    getNumbers().put(methodKey, i);  
    return i;  
}
```

所以会返回defaultValue，加上本身调用的那一次，计算之后就会为3，如图：

```
public Result doInvoke(Invocation invocation, final List<Invoker<T>>  
    List<Invoker<T>> copyInvokers = invokers; copyInvokers: size = 1  
    checkInvokers(copyInvokers, invocation); copyInvokers: size = 1  
    int len = getUrl().getMethodParameter(invocation.getMethodName(),  
    if (len <= 0) { len: 3  
        len = 1;  
    }  
    // retry loop.  
}
```

所以可以发现，采用@Reference注解的形式配置retries为0时，dubbo重试次数为2次（3中包含本身调用的那次）。

后面是采用 dubbo:reference 标签的方式：

```
<dubbo:reference id="testService" interface="com.dubbo.service.TestService"  
    protocol="dubbo" check="false" retries="0" timeout="3000"/>
```

方式如上，在获取属性时，可以看到获得的值为0，和注解形式配置的一致，如图：

```

public int getMethodParameter(String method, String key, int defaultValue) {
    String methodKey = method + "." + key;
    Number n = getNumbers().get(methodKey);
    if (n != null) {
        return n.intValue();
    }
    String value = getMethodParameter(method, key);
    if (value == null || value.length() == 0) {
        return + "0" defaultValue;
    }
}

```

加上本身调用的那一次，计算之后就会为1，如图：

```

public int getMethodParameter(String method, String key, int defaultValue) {
    String methodKey = method + "." + key;
    Number n = getNumbers().get(methodKey);
    if (n != null) {
        return n.intValue();
    }
    String value = getMethodParameter(method, key);
    if (value == null || value.length() == 0) {
        return + "0" defaultValue;
    }
}

```

所以可以发现，采用 dubbo:reference 标签形式配置retries为0时，dubbo重试次数为0（1为本身调用的那次）。

原因分析

首先是@Reference注解形式：

dubbo会把每个接口先解析为ReferenceBean，加上ReferenceBean实现了FactoryBean接口，所以在注入的时候，会调用getObject方法，生成代理对象。

但是这不是关键，因为到这一步时，所有的属性都已经加载完成，所以需要找到dubbo解析注解中属性的代码位置。

dubbo会使用自定义驱动器ReferenceAnnotationBeanPostProcessor来注入属性，而具体执行注入的代码位置是在ReferenceAnnotationBeanPostProcessor类的postProcessPropertyValues方法中调用inject方法执行的。

重点来了，因为采用标签时，是采用@Autowired注解注入，所以是采用spring原生方式注入，而在采用@Reference注解时，注入时会走到dubbo自己的ReferenceAnnotationBeanPostProcessor中私有内部类ReferenceFieldElement的inject方法中，然后调用buildReferenceBean创建ReferenceBean。

离原因越来越近了，在该方法中可以看到beanBuilder中的retries值还是0，说明到这一步还没有被解析为null，如图：

```

ReferenceBeanBuilder beanBuilder = ReferenceBeanBuilder
    .create(reference, classLoader, applicationContext)
    .interfaceClass(referenceClass); referenceClass:
referenceBean = beanBuilder.build(); referenceBean: null
beanBuilder
24 = {LinkedHashMap$Entry@7486} "check" -> true
25 = {LinkedHashMap$Entry@7487} "stubevent" -> "false"
26 = {LinkedHashMap$Entry@7488} "version" ->
27 = {LinkedHashMap$Entry@7489} "sent" -> "false"
28 = {LinkedHashMap$Entry@7490} "actives" -> "0"
29 = {LinkedHashMap$Entry@7491} "generic" -> "false"
30 = {LinkedHashMap$Entry@7492} "url" ->
31 = {LinkedHashMap$Entry@7493} "filter" ->
32 = {LinkedHashMap$Entry@7494} "retries" -> "0"
33 = {LinkedHashMap$Entry@7495} "proxy" ->

```

继续往下走，调用build方法中的configureBean时，在第一步preConfigureBean中方法，在该方法中会创建AnnotationPropertyValuesAdapter对象，在该对象构造方法中会调用adapt方法，然后走到AnnotationUtils中的getAttributes方法中，有一个关键方法nullSafeEquals，该方法会传入当前属性值和默认值。

如果相等，则会忽略掉该属性，然后将符合条件的属性放入actualAttributes这个map中，而我们的retries属性是0，和默认值一致，所以map中不会保存retries属性的值，只有timeout属性，因此出现了后面获取的值为null。

注解方式debug告一段落。

```

return actualAttributes; actualAttributes: size = 1
actualAttributes
actualAttributes = {LinkedHashMap@9252} size = 1
  0 = {LinkedHashMap$Entry@9380} "timeout" -> "1"
    key = "timeout"
    value = {Integer@9362} 1

```

后面是dubbo:reference标签形式：

上面说到了，标签形式走到inject时，会和注解形式有所不同，采用该标签时，dubbo会使用自定义的名称空间解析器去解析，很容易理解，spring也不知道它自定义标签里面那些玩意儿是什么意思，所以dubbo会继承spring的。

NamespaceHandlerSupport，采用自定义的DubboNamespaceHandler解析器来解析的标签，如下图：

```
public class DubboNamespaceHandler extends NamespaceHandlerSupport {  
  
    static {  
        Version.checkDuplicate(DubboNamespaceHandler.class);  
    }  
  
    @Override  
    public void init() {  
        registerBeanDefinitionParser(elementName: "application", new DubboBeanDefinitionParser(ApplicationConfig.class, required: true));  
        registerBeanDefinitionParser(elementName: "module", new DubboBeanDefinitionParser(ModuleConfig.class, required: true));  
        registerBeanDefinitionParser(elementName: "registry", new DubboBeanDefinitionParser(RegistryConfig.class, required: true));  
        registerBeanDefinitionParser(elementName: "monitor", new DubboBeanDefinitionParser(MonitorConfig.class, required: true));  
        registerBeanDefinitionParser(elementName: "provider", new DubboBeanDefinitionParser(ProviderConfig.class, required: true));  
        registerBeanDefinitionParser(elementName: "consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, required: true));  
        registerBeanDefinitionParser(elementName: "protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, required: true));  
        registerBeanDefinitionParser(elementName: "service", new DubboBeanDefinitionParser(ServiceBean.class, required: true));  
        registerBeanDefinitionParser(elementName: "reference", new DubboBeanDefinitionParser(ReferenceBean.class, required: false));  
        registerBeanDefinitionParser(elementName: "annotation", new AnnotationBeanDefinitionParser());  
    }  
}
```

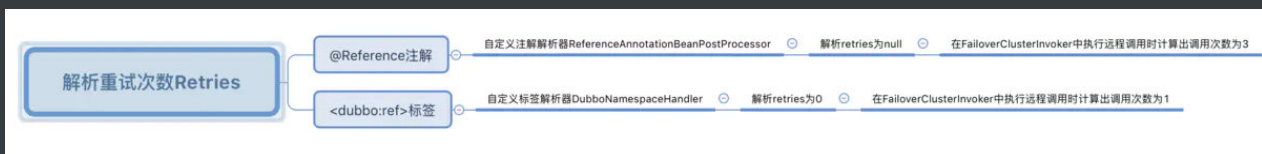
然后调用该类中的parse方法进行解析，而解析retries的地方就是获取class（此时的class就是上图绿色标明的ReferenceBean的class，其父类中有好多好多set方法，其中就包含setRetries方法）中所有的方法，过滤出set开头的方法，然后切割出属性名，放入属性池中，可以看到此处解析出的值为0，并不为null，如下图：

```
131     for (Method setter : beanClass.getMethods()) {  
132         String name = setter.getName();  
133         if (name.length() > 3 && name.startsWith("set")  
134             && Modifier.isPublic(setter.getModifiers())  
135             && setter.getParameterTypes().length == 1) {  
136             Class<?> type = setter.getParameterTypes()[0];  
137             String property = StringUtils.camelToSplitName(camelName: name.substring(3, 4).toL  
138                 props.add(property);
```

```
153     if ("parameters".equals(property)) {  
154         parameters = parseParameters(element.getChildNodes(), beanDefinition);  
155     } else if ("methods".equals(property)) {  
156         parseMethods(id, element.getChildNodes(), beanDefinition, parserContext);  
157     } else if ("arguments".equals(property)) {  
158         parseArguments(id, element.getChildNodes(), beanDefinition, parserContext);  
159     } else {  
160         String value = element.getAttribute(property);  
161         if (value != null) {  
162             value = value.trim();  
163             if (value.length() > 0) {  
164                 if ("registry".equals(property) && RegistryConfig.NO_AVAILABLE.equalsIgn  
165                     registryConfig registryConfig = new RegistryConfig();  
166                     registryConfig.setAddress(RegistryConfig.NO_AVAILABLE);  
167                     beanDefinition.getPropertyValues().addPropertyValue(property, regist  
  
DubboBeanDefinitionParser > parse()
```

小结

画个简单图：



结论

- 采用注解形式：不配置retries或者配置为0，都会重试两次，只有配置为 -1 或更小，才会不执行重试。

- 采用标签形式：不配置retries会重试两次，配置为0或更小都不会重试。

所以建议大家不需要重试时可以设置为-1，比如增删改操作的接口，否则需要保证幂等性。需要重试则设置为1或更大，其实这应该算dubbo的一个bug吧？（我觉得是。。）

到这里就结束了，而上面说到的调用getObject方法就是后续服务发现以及和服务端建立长连接并返回代理对象了。

数据出现3条是因为我定义了接口超时的时间比较短，但是我们的新增涉及文件的操作，流程时间比较长，但是线程还是在的，所以dubbo重试了三次，三次也都是成功的了。

我后面把文件操作改成异步，然后主流程是同步的时间就缩短了很多。

补充：2.7.3版本已修复，就是在注解情况下，nullSafeEquals方法中的默认值和后面保持一致了，都是2，所以为0时也能保存到map中。

我是敖丙，一个在互联网苟且偷生的工具人。

你知道的越多，你不知道的越多，人才们的【三连】就是丙丙创作的最大动力，我们下期见！

注：如果本篇博客有任何错误和建议，欢迎人才们留言，你快说句话啊！

文章持续更新，可以微信搜索「三太子敖丙」第一时间阅读，回复【资料】【面试】【简历】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** <https://github.com/JavaFamily> 已经收录，有大厂面试完整考点，欢迎Star。

