

## 20 | Spring框架：框架帮我们做了很多工作也带来了复杂度

2020-04-28 朱晔

Java业务开发常见错误100例

[进入课程 >](#)




讲述：王少泽

时长 22:40 大小 20.77M



你好，我是朱晔。今天，我们聊聊 Spring 框架给业务代码带来的复杂度，以及与之相关的坑。

在上一讲，通过 AOP 实现统一的监控组件的案例，我们看到了 IoC 和 AOP 配合使用的威力：当对象由 Spring 容器管理成为 Bean 之后，我们不但可以通过容器管理配置 Bean 的属性，还可以方便地对感兴趣的方法做 AOP。

不过，前提是对象必须是 Bean。你可能会觉得这个结论很明显，也很容易理解啊。但 ，上一讲提到的 Bean 默认是单例一样，理解起来简单，实践的时候却非常容易踩坑。其中原因，一方面是，理解 Spring 的体系结构和使用方式有一定曲线；另一方面是，Spring 多年发展堆积起来的内部结构非常复杂，这也是更重要的原因。

在我看来，Spring 框架内部的复杂度主要表现为三点：

第一，Spring 框架借助 IoC 和 AOP 的功能，实现了修改、拦截 Bean 的定义和实例的灵活性，因此真正执行的代码流程并不是串行的。

第二，Spring Boot 根据当前依赖情况实现了自动配置，虽然省去了手动配置的麻烦，但也因此多了一些黑盒、提升了复杂度。

第三，Spring Cloud 模块多版本也多，Spring Boot 1.x 和 2.x 的区别也很大。如果要对 Spring Cloud 或 Spring Boot 进行二次开发的话，考虑兼容性的成本会很高。

今天，我们就通过配置 AOP 切入 Spring Cloud Feign 组件失败、Spring Boot 程序的文件配置被覆盖这两个案例，感受一下 Spring 的复杂度。我希望这一讲的内容，能帮助你面对 Spring 这个复杂框架出现的问题时，可以非常自信地找到解决方案。

## Feign AOP 切不到的诡异案例

我曾遇到过这么一个案例：使用 Spring Cloud 做微服务调用，为方便统一处理 Feign，想到了用 AOP 实现，即使用 within 指示器匹配 feign.Client 接口的实现进行 AOP 切入。

代码如下，通过 @Before 注解在执行方法前打印日志，并在代码中定义了一个标记了 @FeignClient 注解的 Client 类，让其成为一个 Feign 接口：

 复制代码

```
1 //测试Feign
2 @FeignClient(name = "client")
3 public interface Client {
4     @GetMapping("/feignaop/server")
5     String api();
6 }
7
8 //AOP切入feign.Client的实现
9 @Aspect
10 @Slf4j
11 @Component
12 public class WrongAspect {
13     @Before("within(feign.Client+)")
14     public void before(JoinPoint pjp) {
15         log.info("within(feign.Client+) pjp {}", args:{}", pjp, pjp.getArgs());
16     }
17 }
18
```

```

19 //配置扫描Feign
20 @Configuration
21 @EnableFeignClients(basePackages = "org.geekbang.time.commonmistakes.spring.dei
22 public class Config {
23 }
24 //测试Feign
25 @FeignClient(name = "client")
26 public interface Client {
27     @GetMapping("/feignaop/server")
28     String api();
29 }
30
31 //AOP切入feign.Client的实现
32 @Aspect
33 @Slf4j
34 @Component
35 public class WrongAspect {
36     @Before("within(feign.Client+)")
37     public void before(JoinPoint pjp) {
38         log.info("within(feign.Client+) pjp {}, args:{}", pjp, pjp.getArgs());
39     }
40 }
41
42 //配置扫描Feign
43 @Configuration
44 @EnableFeignClients(basePackages = "org.geekbang.time.commonmistakes.spring.dei
45 public class Config {
46 }

```

通过 Feign 调用服务后可以看到日志中有输出，的确实现了 feign.Client 的切入，切入的是 execute 方法：

 复制代码

```

1 [15:48:32.850] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo4.WrongAspe
2
3 Binary data, feign.Request$Options@5c16561a]

```

一开始这个项目使用的是客户端的负载均衡，也就是让 Ribbon 来做负载均衡，代码没啥问题。后来因为后端服务通过 Nginx 实现服务端负载均衡，所以开发同学把 @FeignClient 的配置设置了 URL 属性，直接通过一个固定 URL 调用后端服务：

 复制代码

```

1 @FeignClient(name = "anotherClient",url = "http://localhost:45678")
2 public interface ClientWithUrl {

```

```
3     @GetMapping("/feignaop/server")
4     String api();
5 }
```

但这样配置后，之前的 AOP 切面竟然失效了，也就是 within(feign.Client+) 无法切入 ClientWithUrl 的调用了。

为了还原这个场景，我写了一段代码，定义两个方法分别通过 Client 和 ClientWithUrl 这两个 Feign 进行接口调用：

 复制代码

```
1 @Autowired
2 private Client client;
3
4 @Autowired
5 private ClientWithUrl clientWithUrl;
6
7 @GetMapping("client")
8 public String client() {
9     return client.api();
10 }
11
12 @GetMapping("clientWithUrl")
13 public String clientWithUrl() {
14     return clientWithUrl.api();
15 }
```

可以看到，调用 Client 后 AOP 有日志输出，调用 ClientWithUrl 后却没有：

 复制代码

```
1 [15:50:32.850] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo4.WrongAspe
2
3 Binary data, feign.Request$Options@5c16561
```

这就很费解了。难道为 Feign 指定了 URL，其实现就不是 feign.Clinet 了吗？

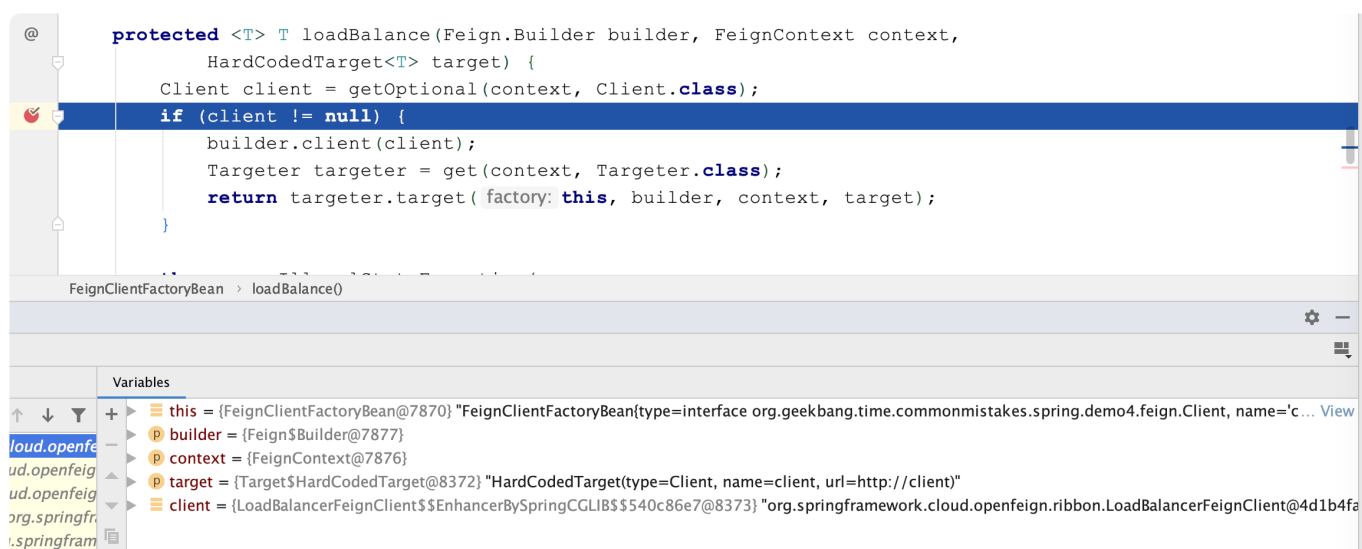
要明白原因，我们需要分析一下 FeignClient 的创建过程，也就是分析 FeignClientFactoryBean 类的 getTarget 方法。源码第 4 行有一个 if 判断，当 URL 没有

内容也就是为空或者不配置时调用 `loadBalance` 方法，在其内部通过 `FeignContext` 从容器获取 `feign.Client` 的实例：

 复制代码

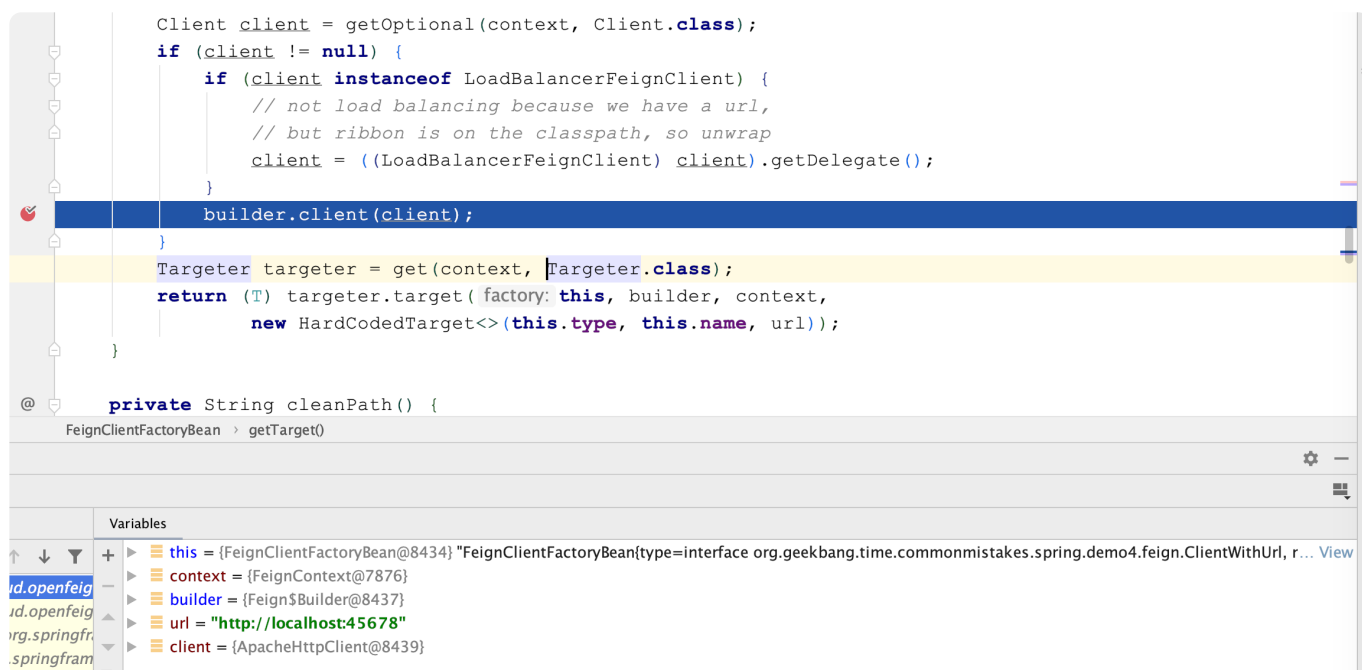
```
1 <T> T getTarget() {
2     FeignContext context = this.applicationContext.getBean(FeignContext.class);
3     Feign.Builder builder = feign(context);
4     if (!StringUtils.hasText(this.url)) {
5         ...
6         return (T) loadBalance(builder, context,
7             new HardCodedTarget<>(this.type, this.name, this.url));
8     }
9     ...
10    String url = this.url + cleanPath();
11    Client client = getOptional(context, Client.class);
12    if (client != null) {
13        if (client instanceof LoadBalancerFeignClient) {
14            // not load balancing because we have a url,
15            // but ribbon is on the classpath, so unwrap
16            client = ((LoadBalancerFeignClient) client).getDelegate();
17        }
18        builder.client(client);
19    }
20    ...
21 }
22 protected <T> T loadBalance(Feign.Builder builder, FeignContext context,
23     HardCodedTarget<T> target) {
24     Client client = getOptional(context, Client.class);
25     if (client != null) {
26         builder.client(client);
27         Targeter targeter = get(context, Targeter.class);
28         return targeter.target(this, builder, context, target);
29     }
30     ...
31 }
32 protected <T> T getOptional(FeignContext context, Class<T> type) {
33     return context.getInstance(this.contextId, type);
34 }
```

调试一下可以看到，`client` 是 `LoadBalancerFeignClient`，已经是经过代理增强的，明显是一个 `Bean`：



所以，没有指定 URL 的 @FeignClient 对应的 LoadBalanceFeignClient，是可以通过 feign.Client 切入的。

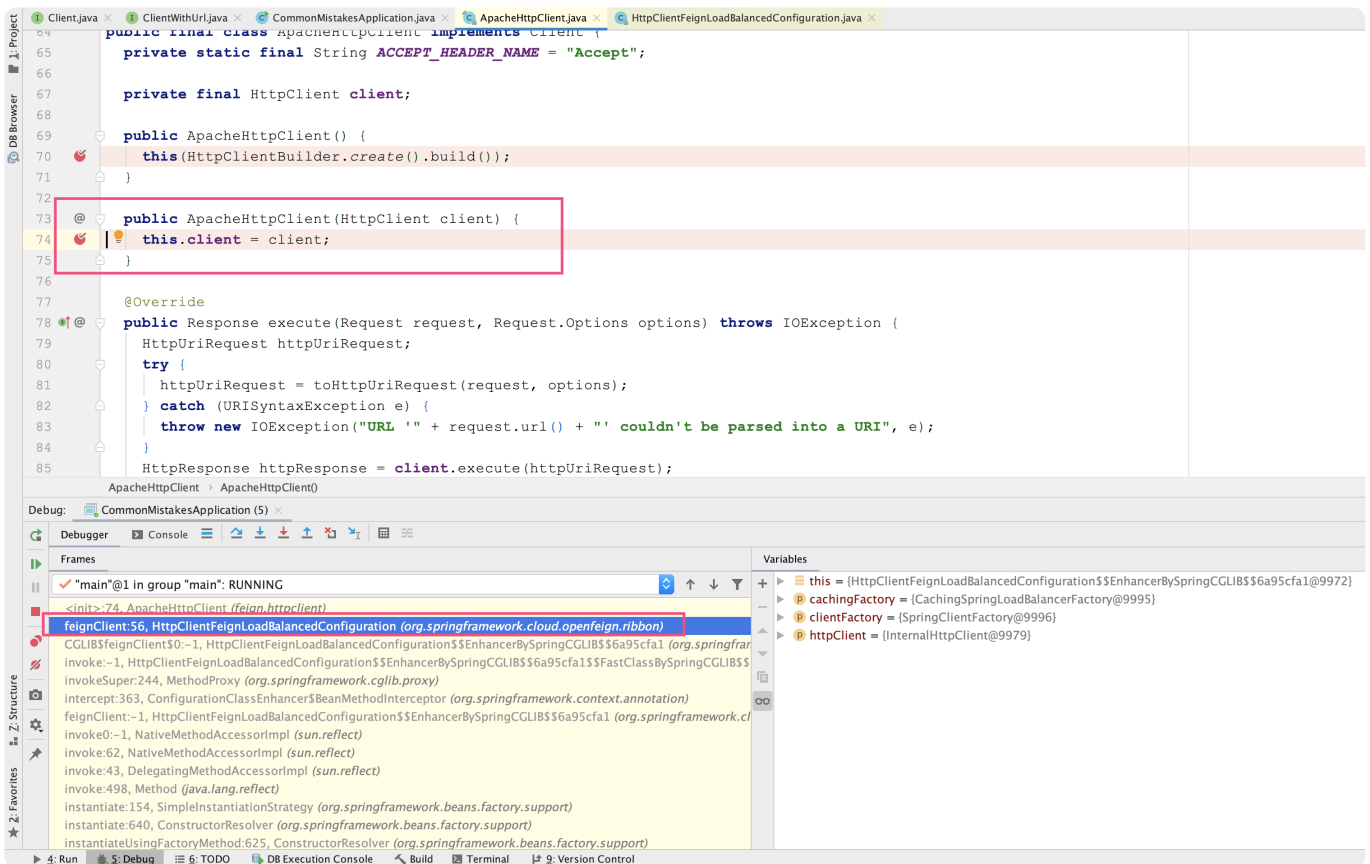
在我们上面贴出来的源码的 16 行可以看到，当 URL 不为空的时候，client 设置为了 LoadBalanceFeignClient 的 delegate 属性。其原因注释中有提到，因为有了 URL 就不需要客户端负载均衡了，但因为 Ribbon 在 classpath 中，所以需要从 LoadBalanceFeignClient 提取出真正的 Client。断点调试下可以看到，这时 client 是一个 ApacheHttpClient：



那么，这个 ApacheHttpClient 是从哪里来的呢？这里，我教你一个小技巧：如果你希望知道一个类是怎样调用栈初始化的，可以在构造方法中设置一个断点进行调试。这样，你就可以在 IDE 的栈窗口看到整个方法调用栈，然后点击每一个栈帧看到整个过程。



用这种方式，我们可以看到，是 HttpClientFeignLoadBalancedConfiguration 类实例化的 ApacheHttpClient：



进一步查看 HttpClientFeignLoadBalancedConfiguration 的源码可以发现，LoadBalancerFeignClient 这个 Bean 在实例化的时候，new 出来一个 ApacheHttpClient 作为 delegate 放到了 LoadBalancerFeignClient 中：

复制代码

```
1 @Bean
2 @ConditionalOnMissingBean(Client.class)
3 public Client feignClient(CachingSpringLoadBalancerFactory cachingFactory,
4     SpringClientFactory clientFactory, HttpClient httpClient) {
5     ApacheHttpClient delegate = new ApacheHttpClient(httpClient);
6     return new LoadBalancerFeignClient(delegate, cachingFactory, clientFactory)
7 }
8
9 public LoadBalancerFeignClient(Client delegate,
10     CachingSpringLoadBalancerFactory lbClientFactory,
11     SpringClientFactory clientFactory) {
12     this.delegate = delegate;
13     this.lbClientFactory = lbClientFactory;
14     this.clientFactory = clientFactory;
15 }
```

显然，ApacheHttpClient 是 new 出来的，并不是 Bean，而 LoadBalancerFeignClient 是一个 Bean。

有了这个信息，我们再来捋一下，为什么 within(feign.Client+) 无法切入设置过 URL 的 @FeignClient ClientWithUrl：

表达式声明的是切入 feign.Client 的实现类。

Spring 只能切入由自己管理的 Bean。

**虽然 LoadBalancerFeignClient 和 ApacheHttpClient 都是 feign.Client 接口的实现，但是 HttpClientFeignLoadBalancedConfiguration 的自动配置只是把前者定义为 Bean，后者是 new 出来的、作为 LoadBalancerFeignClient 的 delegate，不是 Bean。**

在定义了 FeignClient 的 URL 属性后，我们获取的是 LoadBalancerFeignClient 的 delegate，它不是 Bean。

因此，定义了 URL 的 FeignClient 采用 within(feign.Client+) 无法切入。

那，如何解决这个问题呢？有一位同学提出，修改一下切点表达式，通过 @FeignClient 注解来切：

 复制代码

```
1 @Before("@within(org.springframework.cloud.openfeign.FeignClient)")
2 public void before(JoinPoint pjp){
3     log.info("@within(org.springframework.cloud.openfeign.FeignClient) pjp {}"),
4 }
```

修改后通过日志看到，AOP 的确切成功了：

 复制代码

```
1 [15:53:39.093] [http-nio-45678-exec-3] [INFO ] [o.g.t.c.spring.demo4.Wrong2Aspi
```

但仔细一看就会发现，这次切入的是 ClientWithUrl 接口的 API 方法，而不是 client.Feign 接口的 execute 方法，显然不符合预期。



这位同学犯的错误是，没有弄清楚真正希望切的是什么对象。@FeignClient 注解标记在 Feign Client 接口上，所以切的是 Feign 定义的接口，也就是每一个实际的 API 接口。而通过 feign.Client 接口切的是客户端实现类，切到的是通用的、执行所有 Feign 调用的 execute 方法。

那么问题来了，ApacheHttpClient 不是 Bean 无法切入，切 Feign 接口本身又不符合要求。怎么办呢？

经过一番研究发现，ApacheHttpClient 其实有机会独立成为 Bean。查看 HttpClientFeignConfiguration 的源码可以发现，当没有 ILoadBalancer 类型的时候，自动装配会把 ApacheHttpClient 设置为 Bean。

这么做的原因很明确，如果我们不希望做客户端负载均衡的话，应该不会引用 Ribbon 组件的依赖，自然没有 LoadBalancerFeignClient，只有 ApacheHttpClient：

 复制代码


```
1 @Configuration
2 @ConditionalOnClass(ApacheHttpClient.class)
3 @ConditionalOnMissingClass("com.netflix.loadbalancer.ILoadBalancer")
4 @ConditionalOnMissingBean(CloseableHttpClient.class)
5 @ConditionalOnProperty(value = "feign.httpclient.enabled", matchIfMissing = true)
6 protected static class HttpClientFeignConfiguration {
7     @Bean
8     @ConditionalOnMissingBean(Client.class)
9     public Client feignClient(HttpClient httpClient) {
10         return new ApacheHttpClient(httpClient);
11     }
12 }
```

那，把 pom.xml 中的 ribbon 模块注释之后，是不是可以解决问题呢？

 复制代码

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
4 </dependency>
```

但，问题并没解决，启动出错误了：

 复制代码

```
1 Caused by: java.lang.IllegalArgumentException: Cannot subclass final class feign.  
2   at org.springframework.cglib.proxy.Enhancer.generateClass(Enhancer.java:657)  
3   at org.springframework.cglib.core.DefaultGeneratorStrategy.generate(DefaultG
```

这里，又涉及了 Spring 实现动态代理的两种方式：

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；


CGLIB 动态字节码注入方式，通过继承实现代理，没有这个限制。

**Spring Boot 2.x 默认使用 CGLIB 的方式，但通过继承实现代理有个问题是，无法继承 final 的类。因为，ApacheHttpClient 类就是定义为了 final：**

 复制代码


```
1 public final class ApacheHttpClient implements Client {
```

为解决这个问题，我们把配置参数 proxy-target-class 的值修改为 false，以切换到使用 JDK 动态代理的方式：

 复制代码

```
1 spring.aop.proxy-target-class=false
```

修改后执行 clientWithUrl 接口可以看到，通过 within(feign.Client+) 方式可以切入 feign.Client 子类了。以下日志显示了 @within 和 within 的两次切入：

 复制代码

```
1 [16:29:55.303] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo4.Wrong2Aspi  
2 [16:29:55.310] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.spring.demo4.WrongAspe  
3  
4  
5 Binary data, feign.Request$Options@387550b0]
```

这下我们就明白了，Spring Cloud 使用了自动装配来根据依赖装配组件，组件是否成为 Bean 决定了 AOP 是否可以切入，在尝试通过 AOP 切入 Spring Bean 的时候要注意。


加上上一讲的两个案例，我就把 IoC 和 AOP 相关的坑点和你说清楚了。除此之外，我们在业务开发时，还有一个绕不开的点是，Spring 程序的配置问题。接下来，我们就具体看看吧。

## Spring 程序配置的优先级问题

我们知道，通过配置文件 `application.properties`，可以实现 Spring Boot 应用程序的参数配置。但我们可能不知道的是，Spring 程序配置是有优先级的，即当两个不同的配置源包含相同的配置项时，其中一个配置项很可能会被覆盖掉。这，也是为什么我们会遇到些看似诡异的配置失效问题。

我们来通过一个实际案例，研究下配置源以及配置源的优先级问题。

对于 Spring Boot 应用程序，一般我们会通过设置 `management.server.port` 参数，来暴露独立的 actuator 管理端口。这样做更安全，也更方便监控系统统一监控程序是否健康。

 复制代码

```
1 management.server.port=45679
```

有一天程序重新发布后，监控系统显示程序离线。但排查下来发现，程序是正常工作的，只是 actuator 管理端口的端口号被改了，不是配置文件中定义的 45679 了。

后来发现，运维同学在服务器上定义了两个环境变量 `MANAGEMENT_SERVER_IP` 和 `MANAGEMENT_SERVER_PORT`，目的是方便监控 Agent 把监控数据上报到统一的管理服务上：

 复制代码

```
1 MANAGEMENT_SERVER_IP=192.168.0.2
2 MANAGEMENT_SERVER_PORT=12345
```

问题就是出在这里。`MANAGEMENT_SERVER_PORT` 覆盖了配置文件中的 `management.server.port`，修改了应用程序本身的端口。当然，监控系统也就无法通过老的管理端口访问到应用的 health 端口了。如下图所示，actuator 的端口号变成了 12345：

```
▼ {
  "status": "UP",
  ▼ "components": {
    ▼ "db": {
      "status": "UP",
      ▼ "details": {
        "database": "MySQL",
        "result": 1,
        "validationQuery": "/* ping */ SELECT 1"
      }
    }
  },
}
```

到这里坑还没完，为了方便用户登录，需要在页面上显示默认的管理员用户名，于是开发同学在配置文件中定义了一个 `user.name` 属性，并设置为 `defaultadminname`：

```
1 user.name=defaultadminname
```

📋 复制代码

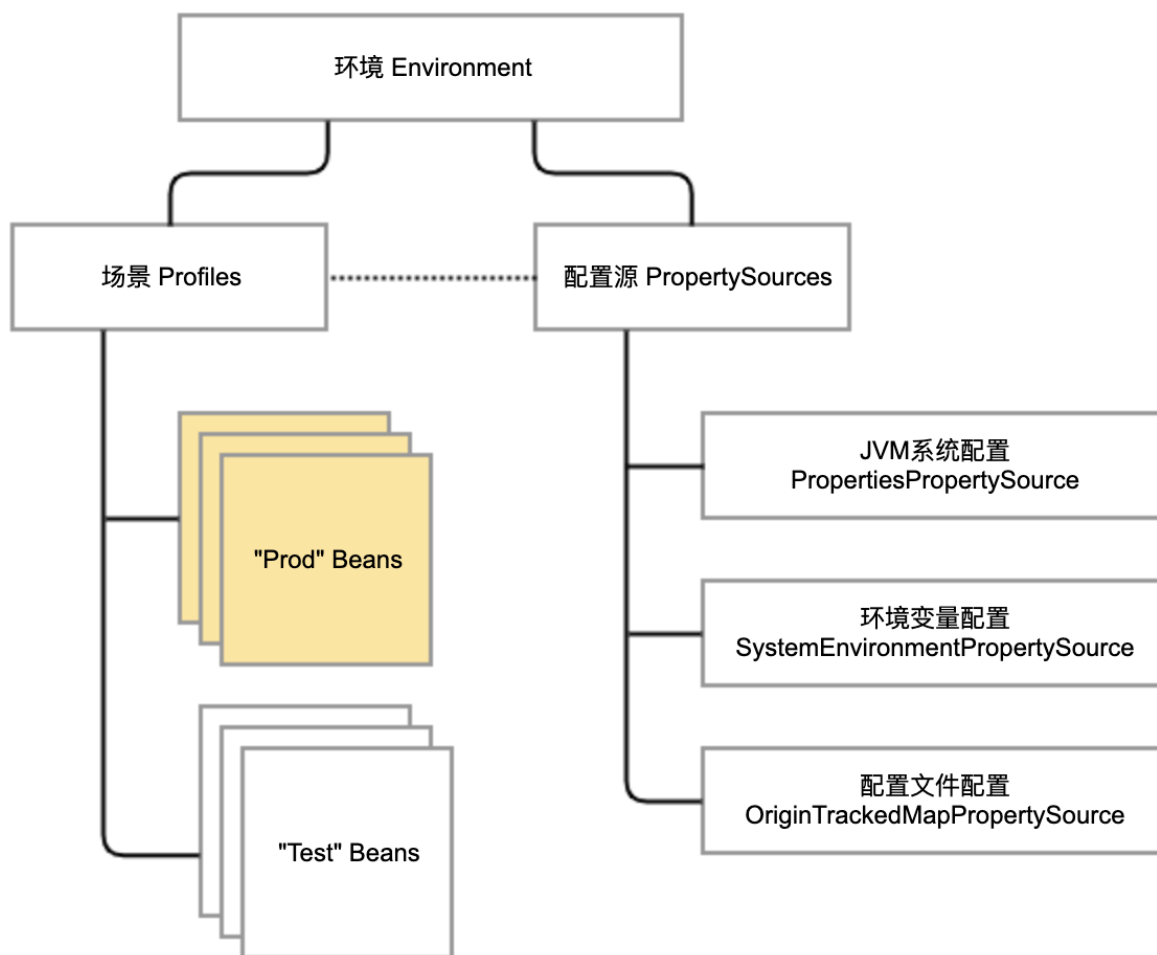
后来发现，程序读取出来的用户名根本就不是配置文件中定义的。这，又是咋回事？

带着这个问题，以及之前环境变量覆盖配置文件配置的问题，我们写段代码看看，从 Spring 中到底能读取到几个 `management.server.port` 和 `user.name` 配置项。

要想查询 Spring 中所有的配置，我们需要以环境 `Environment` 接口为入口。接下来，我就与你说说 Spring 通过环境 `Environment` 抽象出的 `Property` 和 `Profile`：

针对 `Property`，又抽象出各种 `PropertySource` 类代表配置源。一个环境下可能有多个配置源，每个配置源中有诸多配置项。在查询配置信息时，需要按照配置源优先级进行查询。

`Profile` 定义了场景的概念。通常，我们会定义类似 `dev`、`test`、`stage` 和 `prod` 等环境作为不同的 `Profile`，用于按照场景对 `Bean` 进行逻辑归属。同时，`Profile` 和配置文件也有关系，每个环境都有独立的配置文件，但我们只会激活某一个环境来生效特定环境的配置文件。



接下来，我们重点看看 Property 的查询过程。

对于非 Web 应用，Spring 对于 Environment 接口的实现是 StandardEnvironment 类。我们通过 Spring 注入 StandardEnvironment 后循环 getPropertySources 获得的 PropertySource，来查询所有的 PropertySource 中 key 是 user.name 或 management.server.port 的属性值；然后遍历 getPropertySources 方法，获得所有配置源并打印出来：

[复制代码](#)

```
1 @Autowired
2 private StandardEnvironment env;
3 @PostConstruct
4 public void init(){
5     Arrays.asList("user.name", "management.server.port").forEach(key -> {
6         env.getPropertySources().forEach(propertySource -> {
7             if (propertySource.containsProperty(key)) {
8                 log.info("{} -> {} 实际取值: {}", propertySource, proper
```

itjc8.com 搜集整理

```

9         }
10    });
11 });
12
13    System.out.println("配置优先级: ");
14    env.getPropertySources().stream().forEach(System.out::println);
15 }

```

我们研究下输出的日志：

 复制代码

```

1 2020-01-15 16:08:34.054 INFO 40123 --- [main] o.g.t.c.s.d.CommonMi:
2 2020-01-15 16:08:34.054 INFO 40123 --- [main] o.g.t.c.s.d.CommonMi:
3 2020-01-15 16:08:34.054 INFO 40123 --- [main] o.g.t.c.s.d.CommonMi:
4 2020-01-15 16:08:34.054 INFO 40123 --- [main] o.g.t.c.s.d.CommonMi:
5 2020-01-15 16:08:34.054 INFO 40123 --- [main] o.g.t.c.s.d.CommonMi:
6 2020-01-15 16:08:34.054 INFO 40123 --- [main] o.g.t.c.s.d.CommonMi:
7 配置优先级:
8 ConfigurationPropertySourcesPropertySource {name='configurationProperties'}
9 StubPropertySource {name='servletConfigInitParams'}
10 ServletContextPropertySource {name='servletContextInitParams'}
11 PropertiesPropertySource {name='systemProperties'}
12 OriginAwareSystemEnvironmentPropertySource {name='systemEnvironment'}
13 RandomValuePropertySource {name='random'}
14 OriginTrackedMapPropertySource {name='applicationConfig: [classpath:/applicati
15 MapPropertySource {name='springCloudClientHostInfo'}
16 MapPropertySource {name='defaultProperties'}

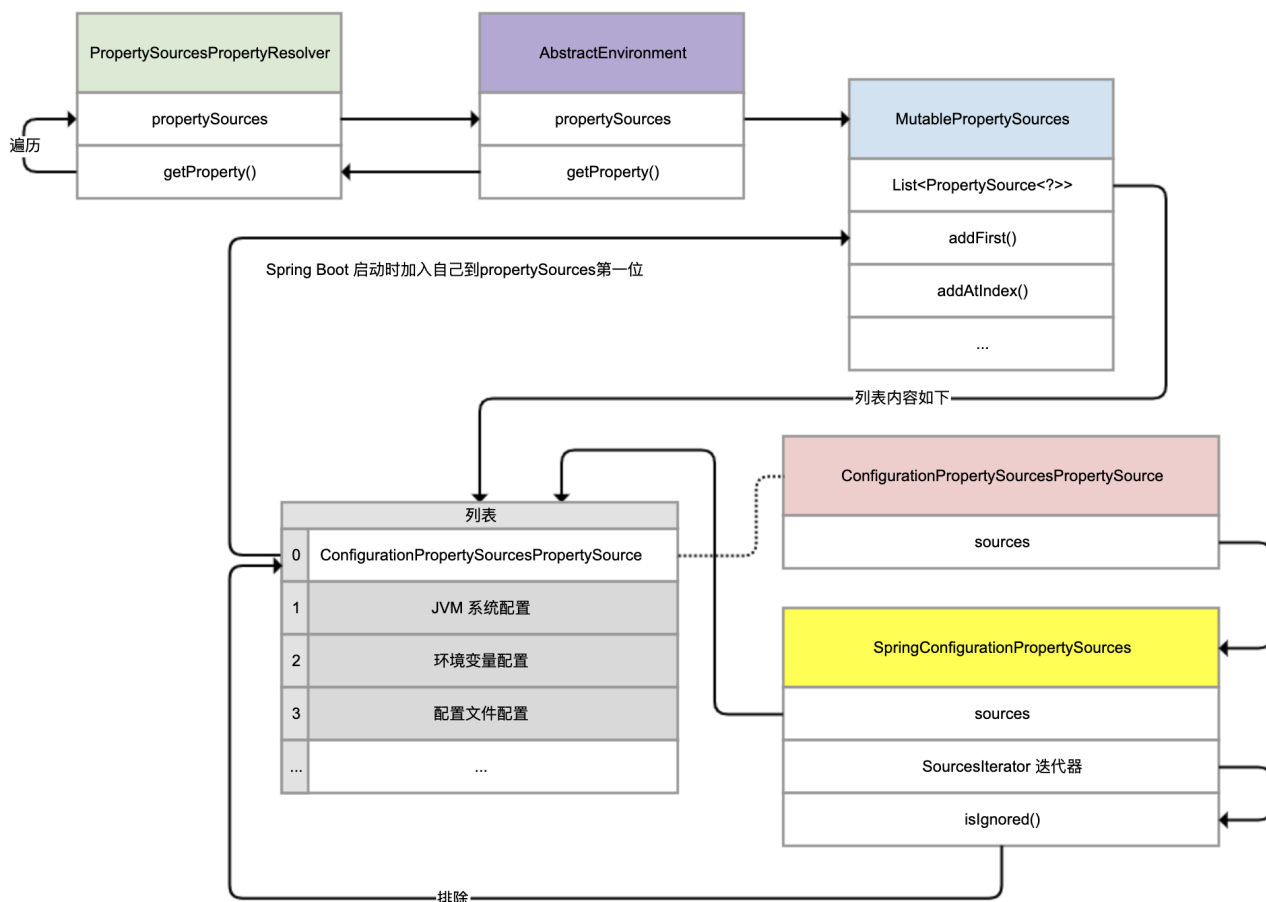
```

有三处定义了 user.name：第一个是 configurationProperties，值是 zhuye；第二个是 systemProperties，代表系统配置，值是 zhuye；第三个是 applicationConfig，也就是我们的配置文件，值是配置文件中定义的 defaultadminname。

同样地，也有三处定义了 management.server.port：第一个是 configurationProperties，值是 12345；第二个是 systemEnvironment 代表系统环境，值是 12345；第三个是 applicationConfig，也就是我们的配置文件，值是配置文件中定义的 45679。

第 7 到 16 行的输出显示，Spring 中有 9 个配置源，值得关注是 ConfigurationPropertySourcesPropertySource、PropertiesPropertySource、OriginAwareSystemEnvironmentPropertySource 和我们的配置文件。

那么，Spring 真的是按这个顺序查询配置吗？最前面的 configurationProperties，又是什么？为了回答这 2 个问题，我们需要分析下源码。我先说明下，下面源码分析的逻辑有些复杂，你可以结合着下面的整体流程图来理解：



Demo 中注入的 StandardEnvironment，继承的是 AbstractEnvironment（图中紫色类）。AbstractEnvironment 的源码如下：

[复制代码](#)

```
1 public abstract class AbstractEnvironment implements ConfigurableEnvironment {
2     private final MutablePropertySources propertySources = new MutablePropertySources();
3     private final ConfigurablePropertyResolver propertyResolver =
4         new PropertySourcesPropertyResolver(this.propertySources);
5
6     public String getProperty(String key) {
7         return this.propertyResolver.getProperty(key);
8     }
9 }
```

可以看到：



MutablePropertySources 类型的字段 propertySources，看起来代表了所有配置源；  
getProperty 方法，通过 PropertySourcesPropertyResolver 类进行查询配置；  
实例化 PropertySourcesPropertyResolver 的时候，传入了当前的  
MutablePropertySources。

接下来，我们继续分析 MutablePropertySources 和  
PropertySourcesPropertyResolver。先看看 MutablePropertySources 的源码（图中蓝色类）：

 复制代码

```
1 public class MutablePropertySources implements PropertySources {
2
3     private final List<PropertySource<?>> propertySourceList = new CopyOnWriteAr
4
5     public void addFirst(PropertySource<?> propertySource) {
6         removeIfPresent(propertySource);
7         this.propertySourceList.add(0, propertySource);
8     }
9     public void addLast(PropertySource<?> propertySource) {
10        removeIfPresent(propertySource);
11        this.propertySourceList.add(propertySource);
12    }
13    public void addBefore(String relativePropertySourceName, PropertySource<?> p
14        ...
15        int index = assertPresentAndGetIndex(relativePropertySourceName);
16        addAtIndex(index, propertySource);
17    }
18    public void addAfter(String relativePropertySourceName, PropertySource<?> |
19        ...
20        int index = assertPresentAndGetIndex(relativePropertySourceName);
21        addAtIndex(index + 1, propertySource);
22    }
23    private void addAtIndex(int index, PropertySource<?> propertySource) {
24        removeIfPresent(propertySource);
25        this.propertySourceList.add(index, propertySource);
26    }
27 }
```


可以发现：

propertySourceList 字段用来真正保存 PropertySource 的 List，且这个 List 是一个  
CopyOnWriteArrayList。

类中定义了 `addFirst`、`addLast`、`addBefore`、`addAfter` 等方法，来精确控制 `PropertySource` 加入 `propertySourceList` 的顺序。这也说明了顺序的重要性。

继续看下 `PropertySourcesPropertyResolver`（图中绿色类）的源码，找到真正查询配置的方法 `getProperty`。

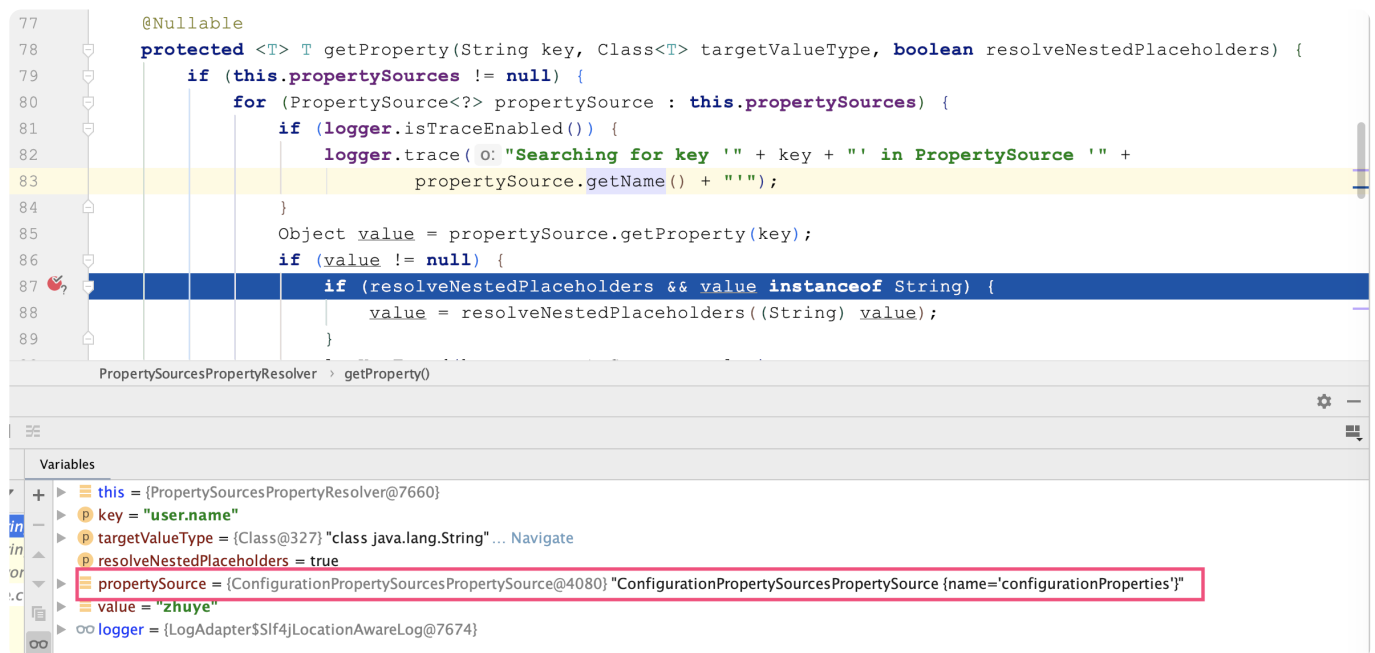
这里，我们重点看一下第 9 行代码：遍历的 `propertySources` 是 `PropertySourcesPropertyResolver` 构造方法传入的，再结合 `AbstractEnvironment` 的源码可以发现，这个 `propertySources` 正是 `AbstractEnvironment` 中的 `MutablePropertySources` 对象。遍历时，如果发现配置源中有对应的 `Key` 值，则使用这个值。因此，`MutablePropertySources` 中配置源的次序尤为重要。

 复制代码

```
1 public class PropertySourcesPropertyResolver extends AbstractPropertyResolver {
2     private final PropertySources propertySources;
3     public PropertySourcesPropertyResolver(@Nullable PropertySources propertySou
4         this.propertySources = propertySources;
5     }
6
7     protected <T> T getProperty(String key, Class<T> targetType, boolean re:
8         if (this.propertySources != null) {
9             for (PropertySource<?> propertySource : this.propertySources) {
10                 if (logger.isTraceEnabled()) {
11                     logger.trace("Searching for key '" + key + "' in PropertySource '" +
12                         propertySource.getName() + "'");
13                 }
14                 Object value = propertySource.getProperty(key);
15                 if (value != null) {
16                     if (resolveNestedPlaceholders && value instanceof String) {
17                         value = resolveNestedPlaceholders((String) value);
18                     }
19                     logKeyFound(key, propertySource, value);
20                     return convertValueIfNecessary(value, targetType);
21                 }
22             }
23         }
24         ...
25     }
26 }
```

回到之前的问题，在查询所有配置源的时候，我们注意到处在第一位的是 `ConfigurationPropertySourcesPropertySource`，这是什么呢？

其实，它不是一个实际存在的配置源，扮演的是一个代理的角色。但通过调试你会发现，我们获取的值竟然是由它提供并且返回的，且没有循环遍历后面的 PropertySource：



继续查看 ConfigurationPropertySourcesPropertySource（图中红色类）的源码可以发现，getProperty 方法其实是通过 findConfigurationProperty 方法查询配置的。如第 25 行代码所示，这其实还是在遍历所有的配置源：



```

23     return null;
24 }
25 for (ConfigurationPropertySource configurationPropertySource : getSource())
26     ConfigurationProperty configurationProperty = configurationPropertySource
27     if (configurationProperty != null) {
28         return configurationProperty;
29     }
30 }
31 return null;
32 }
33 }

```

调试可以发现，这个循环遍历（getSource() 的结果）的配置源，其实是 SpringConfigurationPropertySources（图中黄色类），其中包含的配置源列表就是之前看到的 9 个配置源，而第一个就是 ConfigurationPropertySourcesPropertySource。看到这里，我们的第一感觉是会不会产生死循环，它在遍历的时候怎么排除自己呢？

同时观察 configurationProperty 可以看到，这个 ConfigurationProperty 其实类似代理的角色，实际配置是从系统属性中获得的：

The screenshot shows a Java IDE with a method `findConfigurationProperty` and its evaluation results. The method is as follows:

```

59
60 @
61 private ConfigurationProperty findConfigurationProperty(ConfigurationPropertyName name) {
62     if (name == null) {
63         return null;
64     }
65     for (ConfigurationPropertySource configurationPropertySource : getSource()) {
66         ConfigurationProperty configurationProperty = configurationPropertySource.getConfigurationProperty(name);
67         if (configurationProperty != null) {
68             return configurationProperty;
69         }
70     }
71     return null;
72 }
73
74

```

The evaluation results for the expression `getSource()` are shown in a window titled "Evaluate". The result is a `SpringConfigurationPropertySources` object with a `propertySourceList` of size 9. The list contains the following elements:


- 0 = `ConfigurationPropertySourcesPropertySource@4080` "ConfigurationPropertySourcesPropertySource"
- 1 = `PropertySource$StubPropertySource@7745` "StubPropertySource {name=...}"
- 2 = `ServletContextPropertySource@7729` "ServletContextPropertySource {name=...}"
- 3 = `PropertiesPropertySource@7724` "PropertiesPropertySource {name='systemProperties'}"
- 4 = `SystemEnvironmentPropertySourceEnvironmentPostProcessor$OriginAwarePropertySource@7722` "OriginTrackedMapPropertySource {name='systemProperties'}"
- 5 = `RandomValuePropertySource@7716` "RandomValuePropertySource {name='randomValue'}"
- 6 = `OriginTrackedMapPropertySource@7722` "OriginTrackedMapPropertySource {name='systemProperties'}"
- 7 = `MapPropertySource@7725` "MapPropertySource {name='springCloudClientHostId'}"
- 8 = `MapPropertySource@7746` "MapPropertySource {name='defaultProperties'}"

The `ConfigurationPropertySourcesPropertySource` object is also shown in the Variables window. It has the following properties:

- `this` = `ConfigurationPropertySourcesPropertySource@4080` "ConfigurationPropertySourcesPropertySource"
- `name` = `ConfigurationPropertyName@7684` "user.name"
- `configurationPropertySource` = `SpringIterableConfigurationPropertySource@7685` "PropertiesPropertySource {name='systemProperties'}"
- `configurationProperty` = `ConfigurationProperty@7686` "ConfigurationProperty@b5390 name = user.name, value = 'zhuye', origin = 'user.name' from property source 'systemProperties'"
- `name` = `ConfigurationPropertyName@7697` "user.name"
- `value` = `"zhuye"`
- `origin` = `PropertySourceOrigin@7698` "user.name" from property source "systemProperties"

继续查看 SpringConfigurationPropertySources 可以发现，它返回的迭代器是内部类 SourcesIterator，在 fetchNext 方法获取下一个项时，通过 isIgnored 方法排除了

## ConfigurationPropertySourcesPropertySource (源码第 38 行) :

 复制代码

```
1 class SpringConfigurationPropertySources implements Iterable<ConfigurationProp
2
3     private final Iterable<PropertySource<?>> sources;
4     private final Map<PropertySource<?>, ConfigurationPropertySource> cache = new
5         ReferenceType.SOFT);
6
7     SpringConfigurationPropertySources(Iterable<PropertySource<?>> sources) {
8         Assert.notNull(sources, "Sources must not be null");
9         this.sources = sources;
10    }
11
12    @Override
13    public Iterator<ConfigurationPropertySource> iterator() {
14        return new SourcesIterator(this.sources.iterator(), this::adapt);
15    }
16
17    private static class SourcesIterator implements Iterator<ConfigurationProper
18
19        @Override
20        public boolean hasNext() {
21            return fetchNext() != null;
22        }
23
24        private ConfigurationPropertySource fetchNext() {
25            if (this.next == null) {
26                if (this.iterators.isEmpty()) {
27                    return null;
28                }
29                if (!this.iterators.peek().hasNext()) {
30                    this.iterators.pop();
31                    return fetchNext();
32                }
33                PropertySource<?> candidate = this.iterators.peek().next();
34                if (candidate.getSource() instanceof ConfigurableEnvironment) {
35                    push((ConfigurableEnvironment) candidate.getSource());
36                    return fetchNext();
37                }
38                if (isIgnored(candidate)) {
39                    return fetchNext();
40                }
41                this.next = this.adapter.apply(candidate);
42            }
43            return this.next;
44        }
45
46
47        private void push(ConfigurableEnvironment environment) {
```

```

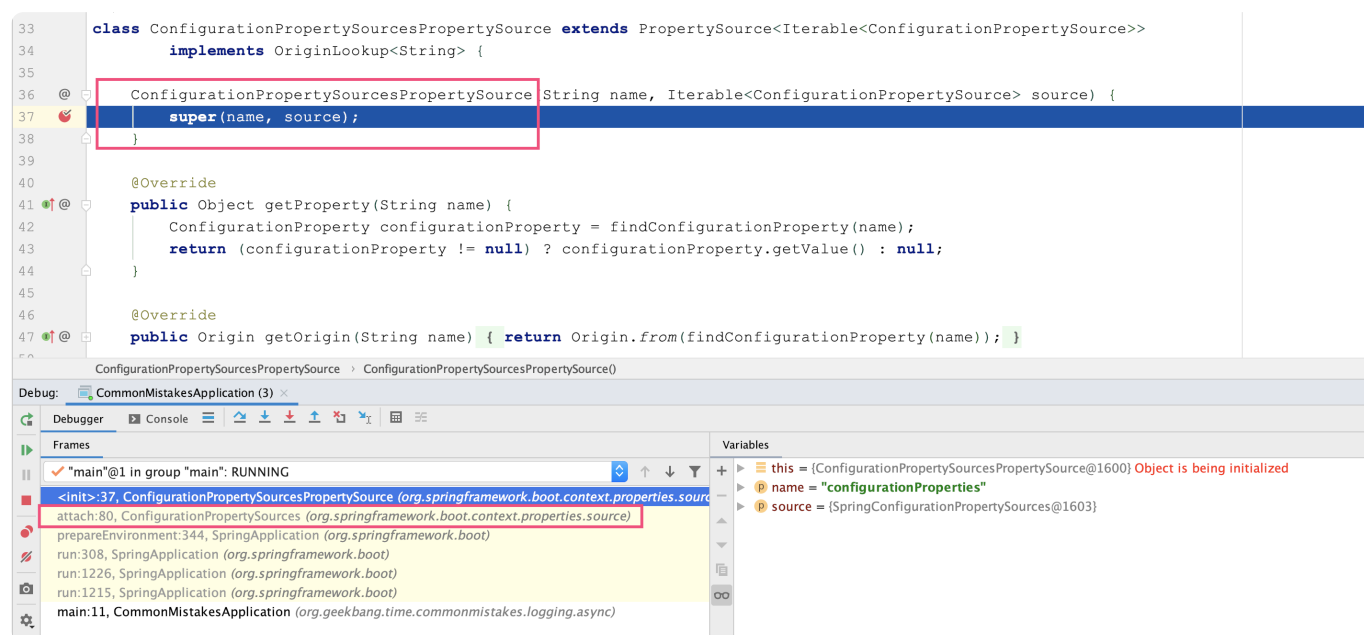
48         this.iterators.push(environment.getPropertySources().iterator());
49     }
50
51
52     private boolean isIgnored(PropertySource<?> candidate) {
53         return (candidate instanceof StubPropertySource
54             || candidate instanceof ConfigurationPropertySourcesPropertySource);
55     }
56 }
57 }

```

我们已经了解了 ConfigurationPropertySourcesPropertySource 是所有配置源中的第一个，实现了对 PropertySourcesPropertyResolver 中遍历逻辑的“劫持”，并且知道了其遍历逻辑。最后一个问题是，它如何让自己成为第一个配置源呢？

再次运用之前我们学到的那个小技巧，来查看实例化

ConfigurationPropertySourcesPropertySource 的地方：



可以看到，ConfigurationPropertySourcesPropertySource 类是由 ConfigurationPropertySources 的 attach 方法实例化的。查阅源码可以发现，这个方法的确从环境中获得了原始的 MutablePropertySources，把自己加入成为一个元素：

复制代码

```

1 public final class ConfigurationPropertySources {
2     public static void attach(Environment environment) {
3         MutablePropertySources sources = ((ConfigurableEnvironment) environment).g
4         PropertySource<?> attached = sources.get(ATTACHED_PROPERTY_SOURCE_NAME);

```

itjc8.com 搜集整理

```
5     if (attached == null) {
6         sources.addFirst(new ConfigurationPropertySourcesPropertySource(ATTACHED,
7             new SpringConfigurationPropertySources(sources)));
8     }
9 }
10 }
```

而这个 attach 方法，是 Spring 应用程序启动时准备环境的时候调用的。在 SpringApplication 的 run 方法中调用了 prepareEnvironment 方法，然后又调用了 ConfigurationPropertySources.attach 方法：

 复制代码

```
1 public class SpringApplication {
2
3     public ConfigurableApplicationContext run(String... args) {
4         ...
5         try {
6             ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
7             ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);
8             ...
9         }
10        private ConfigurableEnvironment prepareEnvironment(SpringApplicationRunListeners listeners,
11            ApplicationArguments applicationArguments) {
12            ...
13            ConfigurationPropertySources.attach(environment);
14            ...
15        }
16    }
```

看到这里你是否彻底理清楚 Spring 劫持 PropertySourcesPropertyResolver 的实现方式，以及配置源有优先级的原因了呢？如果你想知道 Spring 各种预定义的配置源的优先级，可以参考 [官方文档](#)。

## 重点回顾

今天，我用两个业务开发中的实际案例，带你进一步学习了 Spring 的 AOP 和配置优先级这两大知识点。现在，你应该也感受到 Spring 实现的复杂度了。

对于 AOP 切 Feign 的案例，我们在实现功能时走了一些弯路。Spring Cloud 会使用 Spring Boot 的特性，根据当前引入包的情况做各种自动装配。如果我们要扩展 Spring 的



组件，那么只有清晰了解 Spring 自动装配的运作方式，才能鉴别运行时对象在 Spring 容器中的情况，不能想当然认为代码中能看到的所有 Spring 的类都是 Bean。

对于配置优先级的案例，分析配置源优先级时，如果我们以为看到 PropertySourcesPropertyResolver 就看到了真相，后续进行扩展开发时就可能会踩坑。我们一定要注意，**分析 Spring 源码时，你看到的表象不一定是实际运行时的情况，还需要借助日志或调试工具来理清整个过程**。如果没有调试工具，你可以借助 [第 11 讲](#) 用到的 Arthas，来分析代码调用路径。

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. 除了我们这两讲用到 execution、within、@within、@annotation 四个指示器外，Spring AOP 还支持 this、target、args、@target、@args。你能说说后面五种指示器的作用吗？
2. Spring 的 Environment 中的 PropertySources 属性可以包含多个 PropertySource，越往前优先级越高。那，我们能否利用这个特点实现配置文件中属性值的自动赋值呢？比如，我们可以定义 %%MYSQL.URL%%、%%MYSQL.USERNAME%% 和 %%MYSQL.PASSWORD%%，分别代表数据库连接字符串、用户名和密码。在配置数据源时，我们只要设置其值为占位符，框架就可以自动根据当前应用程序名 application.name，统一把占位符替换为真实的数据库信息。这样，生产的数据库信息就不需要放在配置文件中了，会更安全。

关于 Spring Core、Spring Boot 和 Spring Cloud，你还遇到过其他坑吗？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。

© 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 19 | Spring框架：IoC和AOP是扩展的核心

下一篇 加餐1 | 带你吃透课程中Java 8的那些重要知识点（上）

### 精选留言 (3)

 写留言



**Darren**

2020-04-28

第一个问题：

切入点指示符

Spring AOP 支持10种切点指示符：execution、within、this、target、args、@target、@args、@within、@annotation、bean下面做下简记(没有写@Pointcut(),请注意)：

...

展开 ▾

作者回复：第二个问题，我给了我的实现：<https://github.com/JosephZhu1983/java-common-mistakes/tree/master/src/main/java/org/geekbang/time/commonmistakes/springpart2/custompropertysource>



4



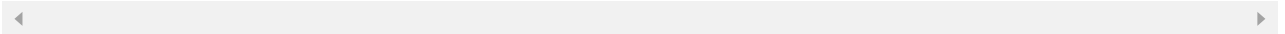
梦倚栏杆

2020-04-28

1. 配置优先级的问题，理解热加载的时候仔细的看过，知道有这么回事，老师举得这个例子：开发和运维都设置了配置，然后运维的把开发的覆盖了，这种问题如果遇到了怎么查呢？如果是同一个人统一管理还好说，不同的人谁知道谁设置了什么呢？
2. 切面不成功的内容，我卡在了切面上。切面怎么用还不太了解。

展开 ▾

作者回复: 可以参考我文中的例子，程序启动的时候检查一下所有配置项，发现有重复配置的话发出告警



👍 2



hellojd

2020-04-28

我也经常看框架源码，但缺失了老师的演示及溯源过程，学习到了。

作者回复: :)

