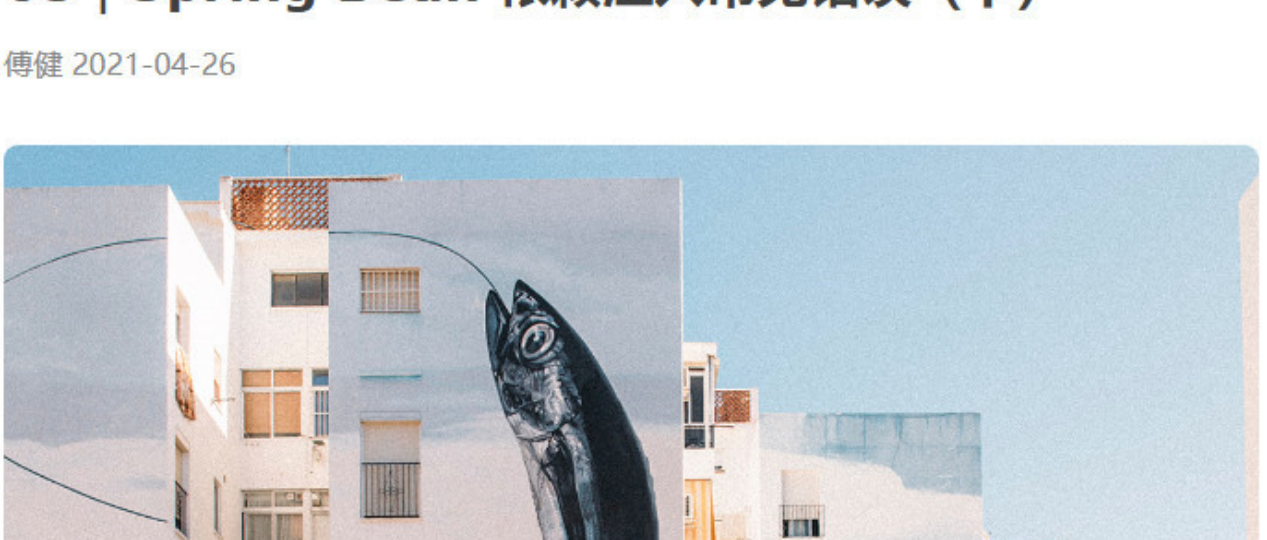
03 | Spring Bean 依赖注入常见错误（下）

傅健 2021-04-26



00:00 1.0x
讲述：傅健 大小：12.32M 时长：13:27

你好，我是傅健，这节课我们接着聊 Spring 的自动注入。

上一讲我们介绍了 3 个 Spring 编程中关于依赖注入的错误案例，这些错误都是比较常见的。如果你仔细分析的话，你会发现它们大多都是围绕着 @Autowired、@Qualifier 的使用而发生，而且自动注入的类型也都是普通对象类型。

那在实际应用中，我们也会使用 @Value 等不太常见的注解来完成自动注入，同时也存在注入到集合、数组等复杂类型的场景。这些情况下，我们也会遇到一些问题。所以这一讲我们不妨来梳理下。

案例 1：@Value 没有注入预期的值

在装配对象成员属性时，我们常常会使用 @Autowired 来装配。但是，有时候我们也使用 @Value 进行装配。不过这两种注解使用风格不同，使用 @Autowired 一般都不会设置属性值，而 @Value 必须指定一个字符串值，因为其定义做了要求，定义代码如下：

```java
public @interface Value {

   /**
    * The actual value expression &mdash; for example, <code>#{systemPropertie
    */
   String value();

}
```

另外在比较这两者的区别时，**我们一般都会因为 @Value 常用于 String 类型的装配而误以为 @Value 不能用于非内置对象的装配，实际上这是一个常见的误区**。例如，我们可以使用下面这种方式来 Autowired 一个属性成员：

```java
@Value("#{student}")
private Student student;
```

其中 student 这个 Bean 定义如下：

```java
@Bean
public Student student(){
    Student student = createStudent(1, "xie");
    return student;
}
```

当然，正如前面提及，我们使用 @Value 更多是用来装配 String，而且它支持多种强大的装配方式，典型的方式参考下面的示例：

```java
//注册正常字符串
@Value("我是字符串")
private String text;

//注入系统参数、环境变量或者配置文件中的值
@Value("${ip}")
private String ip

//注入其他Bean属性，其中student为bean的ID，name为其属性
@Value("#{student.name}")
private String name;
```

上面我给你简单介绍了 @Value 的强大功能，以及它和 @Autowired 的区别。那么在使用 @Value 时可能会遇到那些错误呢？这里分享一个最为典型的错误，即使用 @Value 可能会注入一个不是预期的值。

我们可以模拟一个场景，我们在配置文件 application.properties 配置了这样一个属性：

```
username=admin
password=pass
```

然后我们在一个 Bean 中，分别定义两个属性来引用它们：

```java
@RestController
@Slf4j
public class ValueTestController {
    @Value("${username}")
    private String username;
    @Value("${password}")
    private String password;

    @RequestMapping(path = "user", method = RequestMethod.GET)
    public String getUser(){
       return username + ":" + password;
    };
}
```

当我们去打印上述代码中的 username 和 password 时，我们会发现 password 正确返回了，但是 username 返回的并不是配置文件中指明的 admin，而是运行这段程序的计算机用户名。很明显，使用 @Value 装配的值没有完全符合我们的预期。

案例解析

通过分析运行结果，我们可以知道 @Value 的使用方式应该是没有错的，毕竟 password 这个字段装配上了，但是为什么 username 没有生效成正确的值？接下来我们就来具体解析下。

我们首先了解下对于 @Value，Spring 是如何根据 @Value 来查询"值"的。我们可以先通过方法 DefaultListableBeanFactory#doResolveDependency 来了解 @Value 的核心工作流程，代码如下：


```java
@Nullable
public Object doResolveDependency(DependencyDescriptor descriptor, @Nullable S
      @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeCo
    //省略其他非关键代码
    Class<?> type = descriptor.getDependencyType();
      //寻找@Value
      Object value = getAutowireCandidateResolver().getSuggestedValue(descript
      if (value != null) {
         if (value instanceof String) {
            //解析Value值
            String strVal = resolveEmbeddedValue((String) value);
            BeanDefinition bd = (beanName != null && containsBean(beanName) ?
                  getMergedBeanDefinition(beanName) : null);
            value = evaluateBeanDefinitionString(strVal, bd);
         }

         //转化Value解析的结果到装配的类型
         TypeConverter converter = (typeConverter != null ? typeConverter : ge
         try {
            return converter.convertIfNecessary(value, type, descriptor.getTyp
         }
         catch (UnsupportedOperationException ex) {
            //异常处理
         }
      }
    //省略其他非关键代码
  }
```

可以看到，@Value 的工作大体分为以下三个核心步骤。

1. 寻找 @Value

在这步中，主要是判断这个属性字段是否标记为 @Value，依据的方法参考 QualifierAnnotationAutowireCandidateResolver#findValue：

```java
@Nullable
protected Object findValue(Annotation[] annotationsToSearch) {
   if (annotationsToSearch.length > 0) {
      AnnotationAttributes attr = AnnotatedElementUtils.getMergedAnnotationAtt
            AnnotatedElementUtils.forAnnotations(annotationsToSearch), this.va
      //valueAnnotationType即为@Value
      if (attr != null) {
         return extractValue(attr);
      }
   }
   return null;
}
```

2. 解析 @Value 的字符串值

如果一个字段标记了 @Value，则可以拿到对应的字符串值，然后就可以根据字符串值去做解析，最终解析的结果可能是一个字符串，也可能是一个对象，这取决于字符串怎么写。

3. 将解析结果转化为要装配的对象的类型

当拿到第二步生成的结果后，我们会发现可能和我们要装配的类型不匹配。假设我们定义的是 UUID，而我们获取的结果是一个字符串，那么这个时候就会根据目标类型来寻找转化器执行转化，字符串到 UUID 的转化实际上发生在 UUIDEditor 中：

```java
public class UUIDEditor extends PropertyEditorSupport {

   @Override
   public void setAsText(String text) throws IllegalArgumentException
      if (StringUtils.hasText(text)) {
         //转化操作
         setValue(UUID.fromString(text.trim()));
      }
      else {
         setValue(null);
      }
   }
   //省略其他非关代码

}
```

通过对上面几个关键步骤的解析，我们大体了解了 @Value 的工作流程。结合我们的案例，很明显问题应该发生在第二步，即解析 Value 指定字符串过程，执行过程参考下面的关键代码行：

```java
String strVal = resolveEmbeddedValue((String) value);
```

这里其实是在解析嵌入的值，实际上就是"替换占位符"工作。具体而言，它采用的是 PropertySourcesPlaceholderConfigurer 根据 PropertySources 来替换。不过当使用 ${username} 来获取替换值时，其最终执行的查找并不是局限在 application.property 文件中的。通过调试，我们可以看到下面的这些"源"都是替换依据：

```
[ConfigurationPropertySourcesPropertySource {name='configurationProperties'},
StubPropertySource {name='servletConfigInitParams'}, ServletContextPropertySou
OriginTrackedMapPropertySource {name='applicationConfig: classpath:/applicatio
MapPropertySource {name='devtools'}]
```

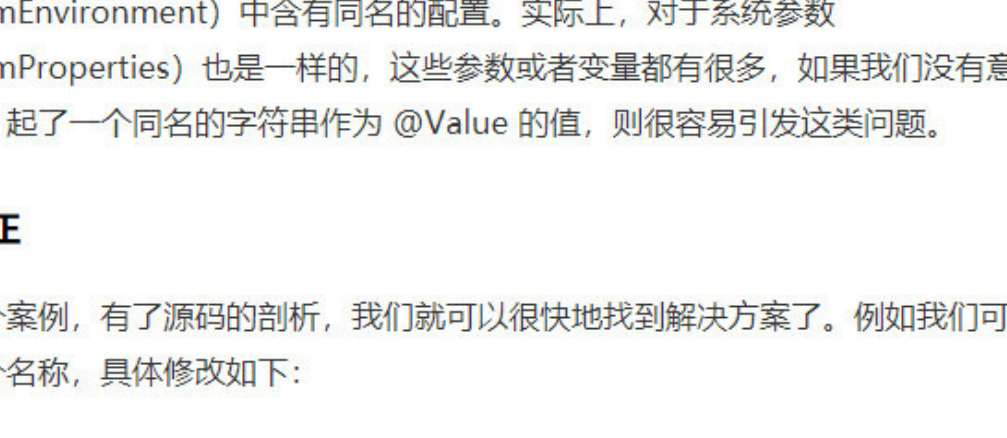
而具体的查找执行，我们可以通过下面的代码（PropertySourcesPropertyResolver#getProperty）来获取它的执行方式：

```java
@Nullable
protected <T> T getProperty(String key, Class<T> targetValueType, boolean reso
   if (this.propertySources != null) {
      for (PropertySource<?> propertySource : this.propertySources) {
         Object value = propertySource.getProperty(key);
         if (value != null) {
         //查到value即退出
         return convertValueIfNecessary(value, targetValueType);
         }
      }
   }

   return null;
}
```

从这可以看出，在解析 Value 字符串时，其实是有顺序的（查找的源是存在 CopyOnWriteArrayList 中，在启动时就被有序固定下来），一个一个"源"执行查找，在其中一个源找到后，就可以直接返回了。

如果我们查看 systemEnvironment 这个源，会发现刚好有一个 username 和我们是重合的，且值不是 pass。



所以，讲到这里，你应该知道问题所在了吧？这是一个误打误撞的例子，刚好系统环境变量（systemEnvironment）中含有同名的配置。实际上，对于系统参数（systemProperties）也是一样的，这些参数或者变量都有很多，如果我们没有意识到它的存在，起了一个同名的字符串作为 @Value 的值，则很容易引发这类问题。

问题修正

针对这个案例，有了源码的剖析，我们就可以很快地找到解决方案了。例如我们可以避免使用同一个名称，具体修改如下：

```
user.name=admin
user.password=pass
```

但是如果我们这么改的话，其实还是不行的。实际上，通过之前的调试方法，我们可以找到类似的原因，在 systemProperties 这个 PropertiesPropertySource 源中刚好存在 user.name，真是无巧不成书。所以命名时，我们一定要注意**不仅要避免和环境变量冲突，也要注意避免和系统变量等其他变量冲突**，这样才能从根本上解决这个问题。

通过这个案例，我们可以知道：Spring 给我们提供了很多好用的功能，但是这些功能交织到一起后，就有可能让我们误入一些坑，只有了解它的运行方式，我们才能迅速定位问题、解决问题。

案例 2：错乱的注入集合

前面我们介绍了很多自动注入的错误案例，但是这些案例都局限在单个类型的注入，对于集合类型的注入并无提及。实际上，**集合类型的自动注入是 Spring 提供的另外一个强大功能。**

假设我们存在这样一个需求：存在多个学生 Bean，我们需要找出来，并存储到一个 List 里面去。多个学生 Bean 的定义如下：

```java
@Bean
public Student student1(){
    return createStudent(1, "xie");
}

@Bean
public Student student2(){
    return createStudent(2, "fang");
}

private Student createStudent(int id, String name) {
    Student student = new Student();
    student.setId(id);
    student.setName(name);
    return student;
}
```

有了集合类型的自动注入后，我们就可以把零散的学生 Bean 收集起来了，代码示例如下：

```java
@RestController
@Slf4j
public class StudentController {

    private List<Student> students;

    public StudentController(List<Student> students){
        this.students = students;
    }

    @RequestMapping(path = "students", method = RequestMethod.GET)
    public String listStudents(){
       return students.toString();
    };

}
```

通过上述代码，我们就可以完成集合类型的注入工作，输出结果如下：

> [Student(id=1, name=xie), Student(id=2, name=fang)]

然而，业务总是复杂的，需求也是一直变动的。当我们持续增加一些 student 时，可能就不喜欢用这种方式来注入集合类型了，而是倾向于用下面的方式去完成注入工作：

```java
@Bean
public List<Student> students(){
    Student student3 = createStudent(3, "liu");
    Student student4 = createStudent(4, "fu");
    return Arrays.asList(student3, student4);
}
```

为了好记，这里我们不妨将上面这种方式命名为"直接装配方式"，而将之前的那种命名为"收集方式"。

实际上，如果这两种方式是非此即彼的存在，自然没有任何问题，都能玩转。但是如果我们不小心让这 2 种方式同时存在了，结果会怎样？

这时候很多人都会觉得 Spring 很强大，肯定会合并上面的结果，或者认为肯定是以直接装配结果为准。然而，当我们运行起程序，就会发现后面的注入方式根本没有生效。即依然返回的是前面定义的 2 个学生。为什么会出现这样的错误呢？

案例解析

要了解这个错误的根本原因，你就得先清楚这两种注入风格在 Spring 中是如何实现的。对于收集装配风格，Spring 使用的是 DefaultListableBeanFactory#resolveMultipleBeans 来完成装配工作，针对本案例关键的核心代码如下：

```java
private Object resolveMultipleBeans(DependencyDescriptor descriptor, @Nullable
      @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeCo
   final Class<?> type = descriptor.getDependencyType();
   if (descriptor instanceof StreamDependencyDescriptor) {
      //装配stream
      return stream;
   }
   else if (type.isArray()) {
      //装配数组
      return result;
   }
   else if (Collection.class.isAssignableFrom(type) && type.isInterface()) {
      //装配集合
      //获取集合的元素类型
      Class<?> elementType = descriptor.getResolvableType().asCollection().res
      if (elementType == null) {
         return null;
      }
      //根据元素类型查找所有的bean
      Map<String, Object> matchingBeans = findAutowireCandidates(beanName, ele
            new MultiElementDescriptor(descriptor));
      if (matchingBeans.isEmpty()) {
         return null;
      }
      if (autowiredBeanNames != null) {
         autowiredBeanNames.addAll(matchingBeans.keySet());
      }
      //转化查到的所有bean放置到集合并返回
      TypeConverter converter = (typeConverter != null ? typeConverter : getTy
      Object result = converter.convertIfNecessary(matchingBeans.values(), typ
      //省略非关键代码
      return result;
   }
   else if (Map.class == type) {
      //解析map
      return matchingBeans;
   }
   else {
      return null;
   }
}
```

到这，我们就不难概括出这种收集式集合装配方式的大体过程了。

1. 获取集合类型的元素类型

针对本案例，目标类型定义为 List<Student> students，所以元素类型为 Student，获取的具体方法参考代码行：

> Class<?> elementType = descriptor.getResolvableType().asCollection().resolveGeneric();

2. 根据元素类型，找出所有的 Bean

有了上面的元素类型，即可根据元素类型来找出所有的 Bean，关键代码行如下：

> Map<String, Object> matchingBeans = findAutowireCandidates(beanName, elementType, new MultiElementDescriptor(descriptor));

3. 将匹配的所有的 Bean 按目标类型进行转化

经过步骤 2，我们获取的所有的 Bean 都是以 java.util.LinkedHashMap.LinkedValues 形式存储的，和我们的目标类型大概率不同，所以最后一步需要做的是**按需转化**。在本案例中，我们就需要把它转化为 List，转化的关键代码如下：

> Object result = converter.convertIfNecessary(matchingBeans.values(), type);

如果我们继续深究执行细节，就可以知道最终是转化器 CollectionToCollectionConverter 来完成这个转化过程。

学习完收集方式的装配原理，我们再来看下直接装配方式的执行过程，实际上这步在前面的课程中我们就提到过（即 DefaultListableBeanFactory#findAutowireCandidates 方法执行），具体的执行过程这里就不多说了。

知道了执行过程，接下来无非就是根据目标类型直接寻找匹配的 Bean。在本案例中，就是将 Bean 名称为 students 的 List<Student> 装配给 StudentController#students 属性。

了解了这两种方式，我们再来思考这两种方式的关系：当同时满足这两种装配方式时，Spring 是如何处理的？这里我们可以参考方法 DefaultListableBeanFactory#doResolveDependency 的几行关键代码，代码如下：

```java
Object multipleBeans = resolveMultipleBeans(descriptor, beanName, autowiredBea
if (multipleBeans != null) {
   return multipleBeans;
}
Map<String, Object> matchingBeans = findAutowireCandidates(beanName, type, des
```

很明显，这两种装配集合的方式是**不能同存**的，结合本案例，当使用收集装配方式来装配时，能找到任何一个对应的 Bean，则返回，如果一个都没有找到，才会采用直接装配的方式。说到这里，你大概能理解为什么后期以 List 方式直接添加的 Student Bean 都不生效了吧。

问题修正

现在如何纠正这个问题就变得简单多了，就是你一定要下意识地避免这 2 种方式共存去装配集合，只用一个这个问题就迎刃而解了。例如，在这里，我们可以使用直接装配的方式去修正问题，代码如下：

```java
@Bean
public List<Student> students(){
    Student student1 = createStudent(1, "xie");
    Student student2 = createStudent(2, "fang");
    Student student3 = createStudent(3, "liu");
    Student student4 = createStudent(4, "fu");
    return Arrays.asList(student1, student2, student3, student4);
}
```

也可以使用收集方式来修正问题时，代码如下：

```java
    @Bean
    public Student student1(){
        return createStudent(1, "xie");
    }
    @Bean
    public Student student2(){
        return createStudent(2, "fang");
    }
    @Bean
    public Student student3(){
        return createStudent(3, "liu");
    }
    @Bean
    public Student student4(){
        return createStudent(4, "fu");
    }
```

总之，都是可以的。还有一点要注意：**在对于同一个集合对象的注入上，混合多种注入方式是不可取的，这样除了错乱，别无所得。**

重点回顾

今天我们又学习了关于 Spring 自动注入的两个典型案例。

通过案例 1 的学习，我们了解到 @Value 不仅可以用来注入 String 类型，也可以注入自定义对象类型。同时在注入 String 时，你一定要意识到它不仅仅可以用来引用配置文件里配置的值，也可能引用到环境变量、系统参数等。

而通过案例 2 的学习，我们了解到集合类型的注入支持两种常见的方式，即上文中我们命名的收集装配式和直接装配式。这两种方式共同装配一个属性时，后者就会失效。

综合上一讲的内容，我们一共分析了 5 个问题以及背后的原理，通过这些案例的分析，我们不难看出 Spring 的自动注入非常强大，围绕 @Autowired、@Qualifier、@Value 等内置注解，我们可以完成不同的注入目标和需求。不过这种强大，正如我在开篇词中提及的，它建立在很多隐性的规则之上。只有你把这些规则都烂熟于心了，才能很好地去规避问题。

思考题

在案例 2 中，我们初次运行程序获取的结果如下：

> [Student(id=1, name=xie), Student(id=2, name=fang)]

那么如何做到让学生 2 优先输出呢？

我们留言区见！