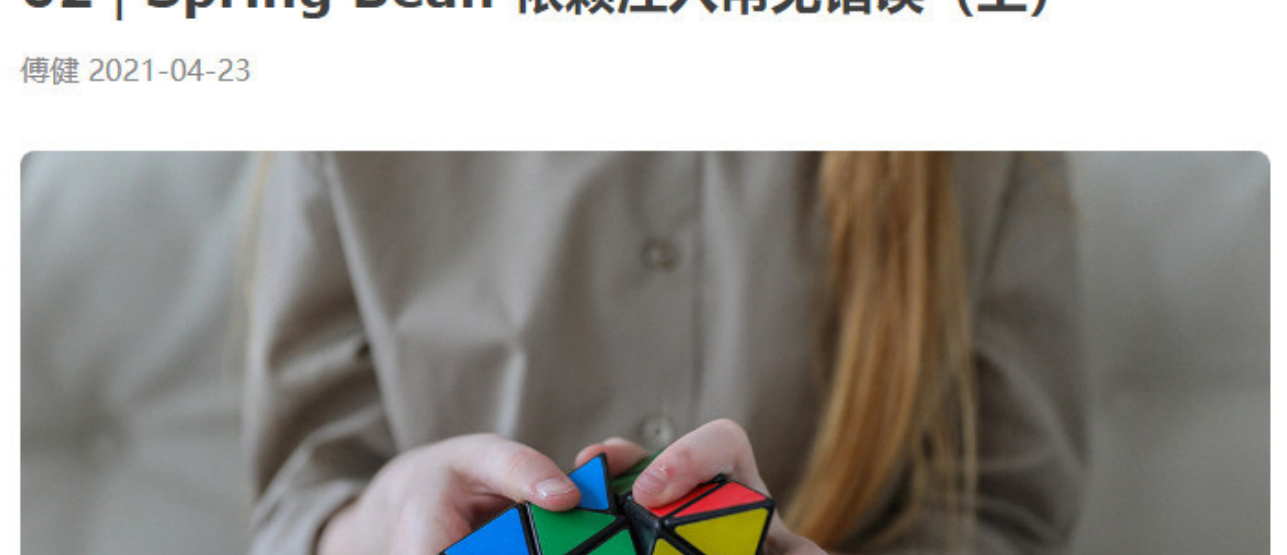
02 | Spring Bean 依赖注入常见错误（上）

傅健 2021-04-23



00:00 1.0x
讲述：傅健 大小：14.74M 时长：16:05

你好，我是傅健，这节课我们来聊聊 Spring @Autowired。

提及 Spring 的优势或特性，我们都会立马想起"**控制反转、依赖注入**"这八字真言。而 @Autowired 正是用来支持依赖注入的核心利器之一。表面上看，它仅仅是一个注解，在使用上不应该出错。但是，在实际使用中，我们仍然会出现各式各样的错误，而且都堪称经典。所以这节课我就带着你学习下这些经典错误及其背后的原因，以防患于未然。

案例 1：过多赠予，无所适从

在使用 @Autowired 时，不管你是菜鸟级还是专家级的 Spring 使用者，都应该制造或者遭遇过类似的错误：

> required a single bean, but 2 were found

顾名思义，我们仅需要一个 Bean，但实际却提供了 2 个（这里的"2"在实际错误中可能是其它大于 1 的任何数字）。

为了重现这个错误，我们可以先写一个案例来模拟下。假设我们在开发一个学籍管理系统案例，需要提供一个 API 根据学生的学号（ID）来移除学生，学生的信息维护肯定需要一个数据库来支撑，所以大体上可以实现如下：

复制代码
```
@RestController
@Slf4j
@Validated
public class StudentController {
    @Autowired
    DataService dataService;

    @RequestMapping(path = "students/{id}", method = RequestMethod.DELETE)
    public void deleteStudent(@PathVariable("id") @Range(min = 1,max = 100) in
        dataService.deleteStudent(id);
    };
}
```

其中 DataService 是一个接口，其实现依托于 Oracle，代码示意如下：

复制代码
```
public interface DataService {
    void deleteStudent(int id);
}

@Repository
@Slf4j
public class OracleDataService implements DataService{
    @Override
    public void deleteStudent(int id) {
        log.info("delete student info maintained by oracle");
    }
}
```

截止目前，运行并测试程序是毫无问题的。但是需求往往是源源不断的，某天我们可能接到节约成本的需求，希望把一些部分非核心的业务从 Oracle 迁移到社区版 Cassandra，所以我们自然会先添加上一个新的 DataService 实现，代码如下：

复制代码
```
@Repository
@Slf4j
public class CassandraDataService implements DataService{
    @Override
    public void deleteStudent(int id) {
        log.info("delete student info maintained by cassandra");
    }
}
```

实际上，当我们完成支持多个数据库的准备工作时，程序就已经无法启动了，报错如下：

很显然，上述报错信息正是我们这一小节讨论的错误，那么这个错误到底是怎么产生的呢？接下来我们具体分析下。

案例解析

要找到这个问题的根源，我们就需要对 @Autowired 实现的依赖注入的原理有一定的了解。首先，我们先来了解下 @Autowired 发生的位置和核心过程。

当一个 Bean 被构建时，核心包括两个基本步骤：

1. 执行 AbstractAutowireCapableBeanFactory#createBeanInstance 方法：通过构造器反射构造出这个 Bean，在此案例中相当于构建出 StudentController 的实例；
2. 执行 AbstractAutowireCapableBeanFactory#populate 方法：填充（即设置）这个 Bean，在本案例中，相当于设置 StudentController 实例中被 @Autowired 标记的 dataService 属性成员。

在步骤 2 中，"填充"过程的关键就是执行各种 BeanPostProcessor 处理器，关键代码如下：

复制代码
```
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable
      //省略非关键代码
      for (BeanPostProcessor bp : getBeanPostProcessors()) {
         if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanP
            PropertyValues pvsToUse = ibp.postProcessProperties(pvs, bw.getWra
           //省略非关键代码
         }
      }
   }
}
```

在上述代码执行过程中，因为 StudentController 含有标记为 Autowired 的成员属性 dataService，所以会使用到 AutowiredAnnotationBeanPostProcessor（BeanPostProcessor 中的一种）来完成"装配"过程：找出合适的 DataService 的 bean 并设置给 StudentController#dataService。如果深究这个装配过程，又可以细分为两个步骤：

1. 寻找出所有需要依赖注入的字段和方法，参考 AutowiredAnnotationBeanPostProcessor#postProcessProperties 中的代码行：

复制代码
```
InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(),
```

2. 根据依赖信息寻找出依赖并完成注入，以字段注入为例，参考 AutowiredFieldElement#inject 方法：

复制代码
```
1  @Override
2  protected void inject(Object bean, @Nullable String beanName, @Nullable Proper
3     Field field = (Field) this.member;
4     Object value;
5     //省略非关键代码
6        try {
7           DependencyDescriptor desc = new DependencyDescriptor(field, this.req
8          //寻找"依赖"，desc为"dataService"的DependencyDescriptor
9           value = beanFactory.resolveDependency(desc, beanName, autowiredBeanNa
10       }
11
12    }
13    //省略非关键代码
14    if (value != null) {
15       ReflectionUtils.makeAccessible(field);
16       //装配"依赖"
17       field.set(bean, value);
18    }
19 }
```

说到这里，我们基本了解了 @Autowired 过程发生的位置和过程。而且很明显，我们案例中的错误就发生在上述"寻找依赖"的过程中（上述代码的第 9 行），那么到底是怎么发生的呢？我们可以继续刨根问底。

为了更清晰地展示错误发生的位置，我们可以采用调试的视角展示其位置（即 DefaultListableBeanFactory#doResolveDependency 中代码片段），参考下图：

如上图所示，当我们根据 DataService 这个类型来找出依赖时，我们会找出 2 个依赖，分别为 CassandraDataService 和 OracleDataService。在这样的情况下，如果同时满足以下两个条件则会抛出本案例的错误：

1. 调用 determineAutowireCandidate 方法来选出优先级最高的依赖，但是发现并没有优先级可依据。具体选择过程可参考 DefaultListableBeanFactory#determineAutowireCandidate：

复制代码
```
protected String determineAutowireCandidate(Map<String, Object> candidates, De
   Class<?> requiredType = descriptor.getDependencyType();
   String primaryCandidate = determinePrimaryCandidate(candidates, requiredTyp
   if (primaryCandidate != null) {
      return primaryCandidate;
   }
   String priorityCandidate = determineHighestPriorityCandidate(candidates, re
   if (priorityCandidate != null) {
      return priorityCandidate;
   }
   // Fallback
   for (Map.Entry<String, Object> entry : candidates.entrySet()) {
      String candidateName = entry.getKey();
      Object beanInstance = entry.getValue();
      if ((beanInstance != null && this.resolvableDependencies.containsValue(b
            matchesBeanName(candidateName, descriptor.getDependencyName())) {
         return candidateName;
      }
   }
   return null;
}
```

如代码所示，优先级的决策是先根据 @Primary 来决策，其次是 @Priority 决策，最后是根据 Bean 名字的严格匹配来决策。如果这些帮助决策优先级的注解都没有被使用，名字也不精确匹配，则返回 null，告知无法决策出哪种最合适。

2. @Autowired 要求是必须注入的（即 required 保持默认值为 true），或者注解的属性类型并不是可以接受多个 Bean 的类型，例如数组、Map、集合。这点可以参考 DefaultListableBeanFactory#indicatesMultipleBeans 的实现：

复制代码
```
private boolean indicatesMultipleBeans(Class<?> type) {
   return (type.isArray() || (type.isInterface() &&
         (Collection.class.isAssignableFrom(type) || Map.class.isAssignableFro
}
```

对比上述两个条件和我们的案例，很明显，案例程序能满足这些条件，所以报错并不奇怪。而如果我们把这些条件想得简单点，或许更容易帮助我们去理解这个设计。就像我们遭遇多个无法比较优劣的选择，却必须选择其一时，与其偷偷地随便选择一种，还不如直接报错，起码可以避免更严重的问题发生。

问题修正

针对这个案例，有了源码的剖析，我们可以很快找到解决问题的方法：**打破上述两个条件中的任何一个即可，即让候选项具有优先级或压根可以不去选择。**不过需要你注意的是，不是每一种条件的打破都满足实际需求，例如我们可以通过使用标记 @Primary 的方式来让被标记的候选者有更高优先级，从而避免报错，但是它并不一定符合业务需求，这就好比我们本身需要两种数据库都能使用，而不是顾此失彼。

复制代码
```
@Repository
@Primary
@Slf4j
public class OracleDataService implements DataService{
    //省略非关键代码
}
```

现在，请你仔细研读上述的两个条件，要同时支持多种 DataService，且能在不同业务情景下精确匹配到要选择到的 DataService，我们可以使用下面的方式去修改：

仅供本群内部学习使用
防断更微信：699250

复制代码
```
@Autowired
DataService oracleDataService;
```

如代码所示，修改方式的精髓在于将属性名和 Bean 名字精确匹配，这样就可以让注入选择不犯难：需要 Oracle 时指定属性名为 oracleDataService，需要 Cassandra 时则指定属性名为 cassandraDataService。

案例 2：显式引用 Bean 时首字母忽略大小写

针对案例 1 的问题修正，实际上还存在另外一种常用的解决办法，即采用 @Qualifier 来显式指定引用的是那种服务，例如采用下面的方式：

复制代码
```
@Autowired()
@Qualifier("cassandraDataService")
DataService dataService;
```

这种方式之所以能解决问题，在于它能让寻找出的 Bean 只有一个（即精确匹配），所以压根不会出现后面的决策过程，可以参考 DefaultListableBeanFactory#doResolveDependency：

复制代码
```
1  @Nullable
2  public Object doResolveDependency(DependencyDescriptor descriptor, @Nullable S
3        @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeCo
4        //省略其他非关键代码
5        //寻找bean过程
6        Map<String, Object> matchingBeans = findAutowireCandidates(beanName, typ
7        if (matchingBeans.isEmpty()) {
8           if (isRequired(descriptor)) {
9              raiseNoMatchingBeanFound(type, descriptor.getResolvableType(), des
10          }
11          return null;
12       }
13       //省略其他非关键代码
14       if (matchingBeans.size() > 1) {
15          //省略多个bean的决策过程，即案例1重点介绍内容
16       }
17      //省略其他非关键代码
18 }
```

我们会使用 @Qualifier 指定的名称去匹配，最终只找到了唯一一个。

不过在使用 @Qualifier 时，我们有时候会犯另一个经典的小错误，就是我们可能会忽略 Bean 的名称首字母大小写。这里我们把校正后的案例稍稍变形如下：

复制代码
```
@Autowired
@Qualifier("CassandraDataService")
DataService dataService;
```

运行程序，我们会报错如下：

> Exception encountered during context initialization - cancelling refresh attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'studentController': Unsatisfied dependency expressed through field 'dataService'; nested exception is org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'com.spring.puzzle.class2.example2.DataService' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations: {@org.springframework.beans.factory.annotation.Autowired(required=true), @org.springframework.beans.factory.annotation.Qualifier(value=CassandraDataService)}

这里我们很容易得出一个结论：**对于 Bean 的名字，如果没有显式指明，就应该是类名，不过首字母应该小写。**但是这个轻松得出的结论成立么？

不妨再测试下，假设我们需要支持 SQLite 这种数据库，我们定义了一个命名为 SQLiteDataService 的实现，然后借鉴之前的经验，我们很容易使用下面的代码来引用这个实现：

复制代码
```
@Autowired
@Qualifier("sQLiteDataService")
DataService dataService;
```

满怀信心运行完上面的程序，依然会出现之前的错误，而如果改成 SQLiteDataService，则运行通过了。这和之前的结论又矛盾了。所以，显式引用 Bean 时，首字母到底是大写还是小写呢？

案例解析

对于这种错误的报错位置，其实我们正好在本案例的开头就贴出了（即第二段代码清单的第 9 行）：

复制代码
```
raiseNoMatchingBeanFound(type, descriptor.getResolvableType(), descriptor);
```

即当因为名称问题（例如引用 Bean 首字母搞错了）找不到 Bean 时，会直接抛出 NoSuchBeanDefinitionException。

在这里，我们真正需要关心的问题是：不显式设置名字的 Bean，其默认名称首字母到底是大写还是小写呢？

看案例的话，当我们启动基于 Spring Boot 的应用程序时，会自动扫描我们的 Package，以找出直接或间接标记了 @Component 的 Bean 的定义（即 BeanDefinition）。例如 CassandraDataService、SQLiteDataService 都被标记了 @Repository，而 Repository 本身被 @Component 标记，所以它们都是间接标记了 @Component。

一旦找出这些 Bean 的信息，就可以生成这些 Bean 的名字，然后组合成一个个 BeanDefinitionHolder 返回给上层。这个过程关键步骤可以查看下图的代码片段（ClassPathBeanDefinitionScanner#doScan）：

基本匹配我们前面描述的过程，其中方法调用 BeanNameGenerator#generateBeanName 即用来产生 Bean 的名字，它有两种实现方式。因为 DataService 的实现都是使用注解标记的，所以 Bean 名称的生成逻辑最终调用的其实是 AnnotationBeanNameGenerator#generateBeanName 这种实现方式，我们可以看下它的具体实现，代码如下：

复制代码
```
@Override
public String generateBeanName(BeanDefinition definition, BeanDefinitionRegist
   if (definition instanceof AnnotatedBeanDefinition) {
      String beanName = determineBeanNameFromAnnotation((AnnotatedBeanDefiniti
      if (StringUtils.hasText(beanName)) {
         // Explicit bean name found.
         return beanName;
      }
   }
   // Fallback: generate a unique default bean name.
   return buildDefaultBeanName(definition, registry);
}
```

大体流程只有两步：看 Bean 有没有显式指明名称，如果有则用显式名称，如果没有则产生一个默认名称。很明显，在我们的案例中，是没有给 Bean 指定名字的，所以产生的 Bean 的名称就是生成的默认名称，查看默认名的产生方法 buildDefaultBeanName，其实现如下：

复制代码
```
protected String buildDefaultBeanName(BeanDefinition definition) {
   String beanClassName = definition.getBeanClassName();
   Assert.state(beanClassName != null, "No bean class name set");
   String shortClassName = ClassUtils.getShortName(beanClassName);
   return Introspector.decapitalize(shortClassName);
}
```

首先，获取一个简短的 ClassName，然后调用 Introspector#decapitalize 方法，设置首字母大写或小写，具体参考下面的代码实现：

复制代码
```
public static String decapitalize(String name) {
    if (name == null || name.length() == 0) {
        return name;
    }
    if (name.length() > 1 && Character.isUpperCase(name.charAt(1)) &&
                    Character.isUpperCase(name.charAt(0))){
        return name;
    }
    char chars[] = name.toCharArray();
    chars[0] = Character.toLowerCase(chars[0]);
    return new String(chars);
}
```

到这，我们很轻松地明白了前面两个问题出现的原因：**如果一个类名是以两个大写字母开头的，则首字母不变，其它情况下默认首字母变成小写。**结合我们之前的案例，SQLiteDataService 的 Bean，其名称应该就是类名本身，而 CassandraDataService 的 Bean 名称则变成了首字母小写（cassandraDataService）。

问题修正

现在我们已经从源码级别了解了 Bean 名字产生的规则，就可以很轻松地修正案例中的两个错误了。以引用 CassandraDataService 类型的 Bean 的错误修正为例，可以采用下面这两种修改方式：

1. 引用处纠正首字母大小写问题：

复制代码
```
@Autowired
@Qualifier("cassandraDataService")
DataService dataService;
```

2. 定义处显式指定 Bean 名字，我们可以保持引用代码不变，而通过显式指明 CassandraDataService 的 Bean 名称为 CassandraDataService 来纠正这个问题。

复制代码
```
@Repository("CassandraDataService")
@Slf4j
public class CassandraDataService implements DataService {
  //省略实现
}
```

现在，我们的程序就可以精确匹配到要找的 Bean 了。比较一下这两种修改方法的话，如果你不太了解源码，不想纠结于首字母到底是大写还是小写，建议你用第二种方法去避免困扰。

案例 3：引用内部类的 Bean 遗忘类名

解决完案例 2，是不是就意味着我们能搞定所有 Bean 的显式引用，不再犯错了呢？天真了。我们可以沿用上面的案例，稍微再添加点别的需求，例如我们需要定义一个内部类来实现一种新的 DataService，代码如下：

复制代码
```
public class StudentController {
    @Repository
    public static class InnerClassDataService implements DataService{
        @Override
        public void deleteStudent(int id) {
          //空实现
        }
    }
    //省略其他非关键代码
 }
```

遇到这种情况，我们一般都会很自然地用下面的方式直接去显式引用这个 Bean：

复制代码
```
@Autowired
@Qualifier("innerClassDataService")
DataService innerClassDataService;
```

很明显，有了案例 2 的经验，我们上来就直接采用了**首字母小写**以避免案例 2 中的错误，但这样的代码是不是就没问题了呢？实际上，仍然会报错"找不到 Bean"，这是为什么？

案例解析

实际上，我们遭遇的情况是"如何引用内部类的 Bean"。解析案例 2 的时候，我曾经贴出了如何产生默认 Bean 名的方法（即 AnnotationBeanNameGenerator#buildDefaultBeanName），当时我们只关注了首字母是否小写的代码片段，而在最后变换首字母之前，有一行语句是对 class 名字的处理，代码如下：

> String shortClassName = ClassUtils.getShortName(beanClassName);

我们可以看下它的实现，参考 ClassUtils#getShortName 方法：

复制代码
```
public static String getShortName(String className) {
   Assert.hasLength(className, "Class name must not be empty");
   int lastDotIndex = className.lastIndexOf(PACKAGE_SEPARATOR);
   int nameEndIndex = className.indexOf(CGLIB_CLASS_SEPARATOR);
   if (nameEndIndex == -1) {
      nameEndIndex = className.length();
   }
   String shortName = className.substring(lastDotIndex + 1, nameEndIndex);
   shortName = shortName.replace(INNER_CLASS_SEPARATOR, PACKAGE_SEPARATOR);
   return shortName;
}
```

很明显，假设我们是一个内部类，例如下面的类名：

> com.spring.puzzle.class2.example3.StudentController.InnerClassDataSe rvice

在经过这个方法的处理后，我们得到的其实是下面这个名称：

> StudentController.InnerClassDataService

最后经过 Introspector.decapitalize 的首字母变换，最终获取的 Bean 名称如下：

> studentController.InnerClassDataService

所以我们在案例程序中，直接使用 innerClassDataService 自然找不到想要的 Bean。

问题修正

通过案例解析，我们很快就找到了这个内部类，Bean 的引用问题顺手就修正了，如下：

复制代码
```
@Autowired
@Qualifier("studentController.InnerClassDataService")
DataService innerClassDataService;
```

这个引用看起来有些许奇怪，但实际上是可以工作的，反而直接使用 innerClassDataService 来引用倒是真的不可行。

通过这个案例我们可以看出，**对源码的学习是否全面决定了我们以后犯错的可能性大小。**如果我们在学习案例 2 时，就对 class 名称的变化部分的源码进行了学习，那么这种错误是不容易犯的。不过有时候我们确实很难一上来就把学习开展的全面而深入，总是需要时间和错误去锤炼的。

重点回顾

看完这三个案例，我们会发现，这些错误的直接结果都是找不到合适的 Bean，但是原因却不尽相同。例如案例 1 是因为提供的 Bean 过多又无法决策选择谁；案例 2 和案例 3 是因为指定的名称不规范导致引用的 Bean 找不到。

实际上，这些错误在一些"聪明的"IDE 会被提示出来，但是它们在其它一些不太智能的主流 IDE 中并不能被告警出来。不过悲剧的是，即使聪明的 IDE 也存在误报的情况，所以**完全依赖 IDE 是不靠谱的**，毕竟这些错误都能编译过去。

另外，我们的案例都是一些简化的场景，很容易看出和发现问题，而真实的场景往往复杂得多。例如对于案例 1，我们的同种类型的实现，可能不是同时出现在自己的项目代码中，而是有部分实现出现在依赖的 Jar 库中。所以你一定要对案例背后的源码实现有一个扎实的了解，这样才能在复杂场景中去规避这些问题。

思考题

我们知道了通过 @Qualifier 可以引用想匹配的 Bean，也可以直接命名属性的名称为 Bean 的名称来引用，这两种方式如下：

复制代码
```
//方式1：属性命名为要装配的bean名称
@Autowired
DataService oracleDataService;

//方式2：使用@Qualifier直接引用
@Autowired
@Qualifier("oracleDataService")
DataService dataService;
```

那么对于案例 3 的内部类引用，你觉得可以使用第 1 种方式做到么？例如使用如下代码：

> @Autowired
> DataService studentController.InnerClassDataService;

期待在留言区看到你的答案，我们下节课见！

© 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。