# Lookup注解

#### 1. 引言

想象一下，如果你需要在一个Spring 的单例Bean中使用一个原型Bean，该如何做呢？首先按常规思路试一下。我们使用`@Autowired`注解进行字段注入，然后使用该实例属性，发现无论我们如何尝试，都是同一个实例，其使用的都是同一个实例。

那么我们再思考一下，既然这个原型Bean也是Spring 管理的，那么我们通过其容器`ApplicationContext`应该是能获取到的。按照这个思路来，那么首先应该先获取`ApplicationContext`这个容器，那么该如何获取呢。这里有两种方法，第一种是在我们的单例Bean中通过`@Autowired`直接注入，因为`ApplicationContext`本身也是Spring管理的特殊Bean，第二种我们让单例Bean实现`ApplicationContextAware`接口，这个是Spring的容器感知接口，实现了这个接口的类，就能感知到Spring的容器，也就是Spring会将`ApplicationContext`容器赋值给我们的单例Bean。

到此为止呢，我们已经获取了Spring的容器`ApplicationContext`，那么该如何从容器中获取原型Bean的实例呢，当然跟获取单例Bean的方式没什么不同，我们直接通过`ApplicationContext`的`getBean(Class)`尝试多次获取，然后比较其引用地址，果然发现获取到了不同的Bean，成功地拿到了原型Bean，接下来修改这个原型Bean的内容，进行相关的业务逻辑即可。

到目前为止呢，一切看起来都很正常，就是有点小小的耦合，我们一直跟Spring的容器耦合在一起，那么有没有更好的办法呢？一个小小的优化，我们可以将获取实例的方法封装起来，作为一个公共方法使用，现在看起来我们进了一步，不用在每个类中显式地注入容器了。

那么大家想想，我们可不可以把这个方法也封装起来，让Spring框架自己去实现呢？我们只要给一个方法一个标记，然后由框架帮我们自动完成。假设我们要实现这个功能，该如何做呢？首先我们需要知道我们要的类的类型是什么吧，然后我们要给他什么参数呢，当然是不需要了，因为我们`getBean(Class`的参数已经通过返回值确定了。那么具体该如何实现呢，首先想到的应该是Spring的AOP切面了吧，我们通过拦截这个标记注解`@Lookup`的方法，在后置拦截方法中，执行我们上面封装好的方法，然后将原型Bean的实例返回，到这一步也就实现了通过注解实现方法注入，从而获取到了原型Bean。

看到这儿大家应该感到很熟悉，我们都知道`Spring AOP`底层是通过JDK动态代理或CGLIB代理来实现的，而Spring `@Lookup`正是通过CGLIB实现的。

#### 2. 简介

`Lookup`在英文中是`查找、检查`的意思 **，** 发音为 `/lkp/` **，** `@Lookup`注解是Spring框架提供的一个注解，用于解决依赖注入中的方法注入问题。这是XML配置中`lookup-method`属性的注解版本，用于在运行时改变特定方法的行为，让它们返回Spring容器中的Bean实例。

#### 3. 基本信息

| 所属框架 | Spring Framework |
| --- | --- |
| 框架版本 | 6.1.4 |
| 所属子项目 | spring-beans |
| 全路径类名 | org.springframework.beans.factory.annotation.Lookup |
| 起始版本 | 4.1 |
| 注解行数 | 67 |

#### 4. 注解目的

`@Lookup`注解标记在方法上，指示容器需要重写这些方法。重写后的方法将会回调到Spring的`BeanFactory`，执行一个`getBean`调用来获取bean实例。这种技术主要用于获取原型作用域的bean或每次需要新实例时的场景。

#### 5. 核心属性

* **value**: 此属性可用于指定目标bean的名称。如果没有指定，容器将根据方法的返回类型解析目标bean。这意味着你可以通过返回类型或者bean名称来指定要查找的bean。

#### 6. 工作原理

* 当容器遇到标记有 `@Lookup`的方法时，它会在运行时通过CGLIB生成该方法所在类的子类，并在这个子类中重写或实现这些方法。重写或实现后的方法

将直接使用`BeanFactory`获取指定的bean。
* 目标bean的解析可以基于方法的返回类型（使用`getBean(Class)`）或建议的bean名称（使用`getBean(String)`），同时传递方法的参数给`getBean`调用，这些参数可以作为目标工厂方法的参数或构造函数的参数。
* 注入方法的签名为:

**<public**|protected **>** [abstract] theMethodName(no-arguments);

#### 7. 使用场景

* **动态返回原型bean**：在单例bean中需要动态获取原型作用域的bean时，可以使用 `@Lookup`注解。因为直接注入原型bean到单例bean中会导致原型bean只被创建一次，失去了原型作用域的意义。
* **动态解析bean**：在需要根据运行时条件动态选择不同bean实例时，`@Lookup`提供了一种灵活的方法来实现。

#### 8. 示例

##### 8.1. 通过`getBean(Class)` 多次获取Bean

有两个类，`AwareCommandManager`为单例Bean，`Command`为原型Bean,在一个循环中分别通过容器的`getBean(Class)`方法根据类的类型获取实例。示例如下：

```
@Component
public class AwareCommandManager implements
ApplicationContextAware {
    //省略代码
    ...
}
@Component
@Scope(”prototype”)
public class Command {
    //省略代码
    ...
}

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
```

```
AnnotationConfigApplicationContext(AppConfig.class);
        //1. 测试多次获取bean是否是同一个实例
        for (int i = 0; i < 3; i++) {
            AwareCommandManager awareCommandManager =
context.getBean(AwareCommandManager.class);
            Command command = context.getBean(Command.class);
            //单例
            System.out.println("awareCommandManager:" +
awareCommandManager);
            //每次获取的示例不一样
            System.out.println("command:" + command);
        }
    }
}
//输出结果:
awareCommandManager:org.study.lookup.AwareCommandManager@2e3
77400
command:org.study.lookup.Command@1757cd72
awareCommandManager:org.study.lookup.AwareCommandManager@2e3
77400
command:org.study.lookup.Command@445b295b
awareCommandManager:org.study.lookup.AwareCommandManager@2e3
77400
command:org.study.lookup.Command@49e5f737

```

可以看到对于单例Bean，多次获取的是同一个实例，对于原型Bean，每次获取的是一个新的实例。

##### 8.2. **对抽象方法使用**`@Lookup`

现在我们使用Spring 提供的`@Lookup`注解在抽象方法上使用，示例如下：

```
@Component
public abstract class AbstractCommandManager {

    public Object process(Object commandObject) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setCommandObject(commandObject);
        return command.execute();
```

```
    }

    // okay... but where is the implementation of this method?
    @Lookup
    protected abstract Command createCommand();
}

@Component
@Scope("prototype")
public class Command {
    private Map<String,String> commandState = new HashMap<>();
    public Object execute() {
        //输出commandState的内容
        System.out.println("当前的实例："+this);
        System.out.println(commandState.values().stream().findFirst().get());
        return null;
  }
}
public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        AbstractCommandManager abstractCommandManager =
context.getBean(AbstractCommandManager.class);
        for (int i = 0; i < 3; i++) {
            abstractCommandManager.process(Integer.valueOf(i));
        }
  }
}

//输出：
当前的实例：org.study.lookup.Command@37271612 commandState:0
当前的实例：org.study.lookup.Command@ed7f8b4 commandState:1
当前的实例：org.study.lookup.Command@4c309d4d commandState:2

```

可以看到添加了注解`@Lookup`的方法createCommand确实返回了不同的实例，Spring动态生成的子类会实现这个抽象方法。

##### 8.3. **对具体方法使用**`@Lookup`

具体方法的使用和抽象类是一样的，Spring动态生成的子类会覆盖原始类中定义的具体方法。

#### 9. 总结

**@Lookup**注解为Spring应用提供了一种强大的方法注入机制，特别是在需要根据每次调用动态返回不同bean实例的场景中。通过使用这个注解，开发者可以在保持应用组件解耦的同时，实现复杂的依赖注入模式。

#### 10. Spring创始人谈方法注入

作者：罗德·约翰逊(Rod Johnson)，时间： 2004年8月6日

几个月前，在我还没有博客的日子里，Cedric和Bob讨论了“Getter注入”。

基本概念是，IoC容器可以在部署时覆盖托管对象上的抽象或具体方法。容器正在注入一个方法，比如一个getter方法，而不是像在Setter注入中那样注入一个引用或原始值。事实上，我已经正在为Spring 1.1开发一个容器方法覆盖机制，此功能已经在Spring 1.1 RC1中发布。这是一个有趣的概念，绝对是完整IoC容器的一部分。然而，我认为这个概念更加通用，需要一个更通用的名称。同时，它只应在相当狭窄的场景范围内使用。

为什么你会想要这样做？Cedric的动机是setter方法是“无用”的，而且“在Java对象中拥有你永远不会调用的方法是一种设计上的坏味道。”在他看来，对象中真正重要的方法实际上是getter，它们通常返回在setter中保存的对象引用。因此，他提议由容器实现getter方法，并且废弃setter方法。在实践中，这意味着容器实际上将覆盖作为应用程序代码一部分定义的getter方法，否则无法使用它们。因此，容器最终将使用与CMP 2.x相似的机制来实现它（尽管希望任何相似之处到此为止）。

我并不真的认同“无用方法”的论点，因为setter方法将通过使用依赖注入的IoC容器调用，并且在没有任何容器的情况下在单元测试中调用。如果对象在容器外部使用，它们将被应用程序代码调用。此外，getter/setter组合是建立默认值的好方法，以防你选择不配置一个或多个setter：如果你需要它，setter就在那里。虽然我可以看到Cedric的动机，但这里有一个权衡：如果我们摆脱了所谓的无用setter，我们留下的是不完整的类。如果getter是抽象的，我们回到了需要测试抽象对象的CMP 2.x测试场景。如果getter是具体的，我们常规地编写将在运行时被覆盖的方法。现在，这确实是无用的代码，在我看来。（通常，我不是覆盖具体方法的粉丝，并且尽可能避免它。我想我第一次在UML参考手册中读到这个建议，这是很有道理的。）在“setter注入”中也涉及一种魔法元素。如果我可以拥有一个简单的POJO，没有任何花哨的容器子类化，我更喜欢它。正如Cedric自己在TSSS上个五月的一个小组讨论中非常恰当地说，“只有当科学失败时才使用魔法。”

我认为这个概念应该被重命名为方法注入，并且它的价值对于一些其他较不常见的场景而言要大得多。

我不会将其作为典型的使用依赖注入配置对象的Setter或构造器注入的替代品。Setter方法和构造器是普通Java构造，它们在容器中工作得很好，但并不依赖于容器。这是好事。由IoC容器提供的魔法方法创造了更多对容器的依赖，尽管当然，仍然可以在容器外部对对象进行子类化，而且它们仍然只是Java。

本质上我将方法注入视为在某些特定情况下替代子类化的一种方式，其中超类应该保持与容器依赖隔离，而容器比常规子类更容易实现所需的行为。所讨论的方法不需要是getter方法（如Setter注入的getter），尽管通常它将是一个返回某物的方法。

我看到容器实现的方法有三个主要情况：

它们可以将容器依赖从应用程序代码中移除。它们可以依赖于直到部署时才知道的基础设施。它们可以定制运行时环境下遗留代码的行为。然而，普通的子类化在这里也是有意义的。容器子类化也比常规子类化更具动态性。我们可以潜在地采取一个基类，并以不同的方式部署它，而不需要管理多个类的源代码。然而，因为它的魔法成分比常规子类化、策略接口或各种替代品更高，我觉得不应该过于急切地使用方法注入。

对我而言，方法注入的主要吸引力是作为一种摆脱我在使用Spring 1.0时有时不得不承担的容器依赖的方式，这适用于任何支持“非单例”或“原型”对象概念的容器。（即，一个容器可以根据配置，根据请求给你一个共享或新实例的IoC管理对象的选项。）我喜欢使用Spring，但我讨厌不得不为了配置而导入Spring API。

促使我实现这一点的具体用例是，当一个通过Spring配置的“单例”对象需要创建一个非单例对象的实例时——例如，一个单线程、单次使用的处理对象——却希望该对象使用依赖注入进行配置，而不仅仅是使用`new`。例如，设想`ThreadSafeService`需要创建一个通过依赖注入配置的`SingleShotHelper`实例。在Spring 1.0.x中，必须让`ThreadSafeService`实现`BeanFactoryAware`生命周期接口，保存`BeanFactory`引用，并且

每次需要创建助手时调用

`(SingleShotHelper) beanFactory.getBean(”singleShotHelper”)`

这样做很好，测试起来也不难（`BeanFactory`是一个简单的接口，所以很容易模拟），但这是一个Spring依赖，理想的情况是能够更接近一个完全非侵入式框架。类型转换也有点不雅观，尽管不是什么大问题。

我通常在大概10个类中就会遇到这种情况的一个案例。我有时会重构这个方法，像这样：

```
protected SingleShotHelper createSingleShotHelper() {
    return (SingleShotHelper) context.getBean(”singleShotHelper”);
}
```

我现在可以通过子类化来实现这一点并将Spring依赖从超类中移除，但这似乎有点过分。

这种方法是容器而不是应用程序开发者实现的理想候选者。它返回容器知道的对象；整个过程实际上可以在配置中比代码中表达得更简洁（考虑到保存`BeanFactory`引用所需的一点点代码）。

通过在Spring 1.1中引入的新方法注入功能，可以使用抽象（或具体）方法，如：

`protected abstract SingleShotHelper createSingleShotHelper();`

并告诉容器在部署时覆盖该方法以从同一个或父工厂返回

特定的bean，如下所示：

```
<lookup-method name=”createSingleShotHelper”
bean=”singleShotHelper” />
```

这些方法可以是protected或public。可以覆盖任意数量的方法。 元素可以像property或constructor-arg元素一样在bean定义元素内使用。

我认为方法注入最有说服力的案例是返回查找容器管理的命名对象的结果。（当然这不是Spring特有的：任何容器都可以实现这一点。）通常会查找一个非单例bean（用Spring的说法）。

这样，在应用程序代码中就没有对Spring或任何其他IoC容器的依赖。一个不需要导入Spring API的边界案例得以解决。正如我所说，这个功能是直接由我正在进行的一个客户项目中的需求所激发的，并且在实践中已被证明是有用的。

查找方法可以与Setter注入或构造器注入结合使用。它们不接受参数，因此方法重载不是问题。

实现使用CGLIB来子类化类。（仅当类路径上有CGLIB时可用，以避免使Spring核心容器依赖于CGLIB。）

Spring更进一步，允许你为覆盖的方法定义任意行为——不仅仅是bean查找。例如，你可能想这样做，以使用基于运行时基础设施的通用行为——比如使用Spring的`TransactionInterceptor`类进行事务回滚。（当然，通常应使用回滚规则来避免这种情况。）或者，对于通用覆盖行为可能有令人信服的案例——比如“如果有活跃的事务，则返回事务性数据源DS1，否则返回非事务性数据源DS2”。再次，如果我们可以将这种逻辑从应用程序代码中隐藏起来，那就是一个胜利。在这里，我们超出了纯粹“getters”的范围：我们可以覆盖方法来发布一个事件，例如，对于任意容器覆盖，通常有替代方案，如子类化类并以正常方式覆盖方法（科学，而不是魔法），或使用AOP。在示例中的bean查找的情况下，容器进行覆盖清楚地消除了对Spring API的依赖。它在XML中描述也更简单。对于更一般的情况，需要有一种方法来解决重载方法。

这已经比我计划的更长了——并且花了一段时间！——所以如果有人感兴趣，我将留待将来的帖子中讨论Spring 1.1的任意覆盖机制（包括它是如何解决重载方法的）。我对那些像Dion和Matt Raible这样的不知疲倦的博客作者产生了新的敬佩，他们似乎一天要博客三次。

注：”CMP 2.x”是指企业Java（Java EE或Jakarta EE）中的”Container-Managed Persistence”（容器管理持久性）版本2.x。CMP是一种允许EJB（Enterprise JavaBeans）容器自动处理数据库交互的技术，简化了开发者对数据库操作的代码编写。在CMP模型中，开发者不需要编写SQL语句或直接管理数据库连接；相反，这些职责都由EJB容器承担。