# 方法注入

一般来说，我们的应用中大多数的 Bean 都是 singleton 的。singleton 依赖 singleton，或者 prototype 依赖 prototype 都很好解决，直接设置属性依赖就可以了。

但是，如果是 singleton 依赖 prototype 呢？这个时候不能用属性依赖，因为如果用属性依赖的话，我们每次其实拿到的还是第一次初始化时候的 bean。

一种解决方案就是不要用属性依赖，每次获取依赖的 bean 的时候从 BeanFactory 中取。这个也是大家最常用的方式了吧。怎么取，我就不介绍了，大部分 Spring 项目大家都会定义那么个工具类的。

另一种解决方案就是这里要介绍的通过使用 Lookup method。

## lookup-method

我们来看一下 Spring Reference 中提供的一个例子：

|  |
| --- |
| package fiona.apple;  // no more Spring imports!  public abstract class CommandManager {  public Object process(Object commandState) {  // grab a new instance of the appropriate Command interface  Command command = createCommand();  // set the state on the (hopefully brand new) Command instance  command.setState(commandState);  return command.execute();  }  // okay... but where is the implementation of this method?  protected abstract Command createCommand();  } |

xml 配置 <lookup-method />：

|  |
| --- |
| <!-- a stateful bean deployed as a prototype (non-singleton) -->  <bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">  <!-- inject dependencies here as required -->  </bean>  <!-- commandProcessor uses statefulCommandHelper -->  <bean id="commandManager" class="fiona.apple.CommandManager">  <lookup-method name="createCommand" bean="myCommand"/>  </bean> |

Spring 采用 CGLIB 生成字节码的方式来生成一个子类。我们定义的类不能定义为 final class，抽象方法上也不能加 final。

lookup-method 上的配置也可以采用注解来完成，这样就可以不用配置 <lookup-method /> 了，其他不变：

|  |
| --- |
| public abstract class CommandManager {  public Object process(Object commandState) {  MyCommand command = createCommand();  command.setState(commandState);  return command.execute();  }  @Lookup("myCommand")  protected abstract Command createCommand();  } |

注意，既然用了注解，要配置注解扫描：<context:component-scan base-package="com.javadoop" />

甚至，我们可以像下面这样：

|  |
| --- |
| public abstract class CommandManager {  public Object process(Object commandState) {  MyCommand command = createCommand();  command.setState(commandState);  return command.execute();  }  @Lookup  protected abstract MyCommand createCommand();  } |
| 上面的返回值用了 MyCommand，当然，如果 Command 只有一个实现类，那返回值也可以写 Command。 |

## replaced-method

replaced-method用于替换掉 bean 中的一些方法。

|  |
| --- |
| public class MyValueCalculator {  public String computeValue(String input) {  // some real code...  }  // some other methods...  } |

方法法覆写，注意要实现 MethodReplacer 接口：

|  |
| --- |
| public class ReplacementComputeValue  implements org.springframework.beans.factory.support.MethodReplacer {  public Object reimplement(Object o, Method m, Object[] args)  throws Throwable {  // get the input value, work with it, and return a computed result  String input = (String) args[0];  ...  return ...;  }  } |

配置也很简单：

|  |
| --- |
| <bean id="myValueCalculator" class="x.y.z.MyValueCalculator">  <!-- 定义 computeValue 这个方法要被替换掉 -->  <replaced-method name="computeValue" replacer="replacementComputeValue">  <arg-type>String</arg-type>  </replaced-method>  </bean>  <bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/> |
| arg-type 明显不是必须的，除非存在方法重载，这样必须通过参数类型列表来判断这里要覆盖哪个方法。 |