LOOKUP METHOD INJECTION

SpringTutors January 10, 2016 Features 90 Views

Lookup Method Injection

In spring when non-singleton bean is get injected into singleton bean the result will be always singleton. Let's discuss it in details-

Let's see an example

```
class A
{
private B b;
public void setB(B b)
{
this.b=b;
}
}
```

```
class B
{
public void m1()
{
//some logic
}
}
```

Now class A wants the functionality of B then there are two ways to get it one is inheritance and another one is composition. Composition is recommended so attribute of type B is declared in class A. Now A should not create the object of B because it will get tightly coupled with B. So we should tell the spring about our classes through spring bean configuration file by declaring A and B as bean

<bean id="a" class="a"> <property name="b" ref="b"/> <bean id="b" class="B"/>

Now when we configure the bean as above the scope of bean a and bean b both are singleton by default so when we request for the bean a it will create only once and return to us.
In this case spring will go to the IOC container and looks for the bean definition in metadata of IOC container as it is there, then it will check the scope of bean a as it is singleton it goes to IOC container and check weather object for bean a is there or not as it is requested for first time it will not there then spring will perform circular dependency check and other dependency-check and starts creating the object of bean a. while creating spring will see that there is property attribute with reference b then it will go to the bean definition of b and checks the scope of bean as it is also singleton then it will go to the IOC container and checks weather object for the bean b is created or not as it is not there, then it will create the object of bean b and store it in IOC container and perform the injection and create the object of a and store it in IOC container.Now class A can use the functionality of B.

```
class A
{
private B b;
public void setB(B b)
{
this.b=b;
}
void m2()
{
b.m1();
}
}
```

Now let's see another example

```
class Circle
{
public final float pi=3.14;
 public float radius;
public float area()
{
return pi*radius*radius;
}
}
```

Now let's suppose there are multiple circles like C1,C2,C3. All circles have different radius.Now I want area of C1 whose radius is 5cm

```
class C1
{
private Circle circle1;
public void getArea()
{
circle1=new Circle();
circle1.radius=5;
float area=circle1.area();
System.olut.println(area);
}
}
```

Now one of my friend wants area of circle C2 whose radius is 10cm.

```
class C2 {
private Circle circle2;
public void getArea()
{
circle2=new Circle();
circle2.radius=10;
float area=circle2.area();
System.out.println(area)l;
}
}
```

Now in the above application we can see that we require different objects for different Circles like C1 and

C2 because radius of different circle will be different.Now for calculating the area of circle all user needs separate object.Now let's see if we permit only one object of Circle to all the user then what happens

Let's C1 uses that object like the following-

```
Circle circle=new Circle();
circle.radius=5;
circle.area();
```

Let's C2 also uses that object like the following-

```
Circle circle=new Circle();
circle.radius=10;
circle.area();
```

Now let's suppose when C1 starts executing after setting the value of radius as 5 with circle object thread switching happens and C2 get chance to execute. C2 set the value of radius as 10 and execute after C2 execution C1 again starts to execute from where it has left the execution means after setting the value of radius as 5 but it's value is overridden by C2 so when it execute it gives wrong results for him.It means data inconsistency will happen if we use only one object of a class which contain non-sharable attribute.For sharable attribute it is fine to use one object like in above application pi value is same for all the circle. It means it is common to all then it should be declared as attribute.

Now configure the above application as bean

```
<bean id="c1" class="C1">
<property name="circle" ref="circle"/>
</bean>
<bean id="circle" class="Circle" scope="prototype"/>
```

Now in the above configuration file we can see that scope of C1 is singleton by default and scope of Circle is prototype. We need multiple objects for the Circle so we have declare its scope as prototype.Now let's create the bean factory and calls the bean

```
BeanFactory factory=new XmlBeanFactory(new ClassPathResource("application-contect.xml"));
 C1 c=factory.getBean("c1",C1.class);
```

Now when we call c1 bean Spring goes to IOC container looks for the bean definition c1 as it is there it reads it and checks for the scope of the bean as it is singleton (by default) it again goes to IOC container and checks weather object of that bean is there or not. As we have requested for the first time it is not there then it comes to configuration file and checks for the circular dependency and dependency-check after passing that stage it starts creating the object of bean c1, while creating the object spring will read the property file attribute name and its reference as reference is of circle bean it looks for the bean definition of circle and checks the scope of circle while it is of prototype so it will not go to IOC container it will just create the object and inject it into c1. In this way object of bean c1 is created and stored into IOC

container.

Now when we request for the bean c1 again like-

```
C1 c1=factory.getBean("c1",C1.class);
```

This time after checking the scope of bean spring will goes to IOC container and checks weather object of c1 is there or not as it is there IOC container will just return it to us. It will not create the new object. It means c==c1 is true. Now in this case Circle is also behaving like singleton but actually its scope is prototype. By doing this we are forced to use same object of Circle which leads to data inconsistency. Now in this situation we should tell the spring to not perform the dependency injection.To solve this problem spring has given the concept called lookup-method.

Now let's see another example

```
public abstract class Radio
{
public void listen()
{
 IReciever reciver;
reciver=lookupReciver();
reciver.tune(399);
 System.out.println(reciver.hashCode());
}
public abstract IReciever lookupReciver();
}
```

```
public interface IReciever
{
public void tune(int frequency);
}
```

```
public class Reciver100Hz implements IReciever
{
public void tune(int frequency)
{
System.out.println("Reciver100Hz is tuned to frquency :"+frequency);
}
}
```

```
public class Reciver10Hz implements IReciever
{
public void tune(int frequency)
{
System.out.println("Reciver10Hz is tuned to frquency :"+frequency);
}
}
```

Now create configuration file

Application-context.xml

```
<bean id="radio" class="Radio">
<lookup-method name="lookupReciver" bean="reciver10Hz"/>
</bean>
<bean id="reciver10Hz" class="Reciver10Hz" scope="prototype"/>
<bean id="reciver100Hz" class="Reciver100Hz" scope="prototype"/>
</beans>
```

Now create the IOC container and call the bean in test class

```
public class Test
{
public static void main(String[] args)
{
BeanFactory factory=new XmlBeanFactory(new ClassPathResource("application-context.xml"));
Radio radio=(Radio)factory.getBean("radio");
Radio radio1=(Radio)factory.getBean("radio");
radio1.listen();
System.out.println(radio1.hashCode());
radio.listen();
System.out.println(radio.hashCode());
}
}
```

Now when factory.getBean("radio") is called spring will go to the IOC container and checks the bean Definition in metadata, as it is there it reads the bean definition while reading it encounter that Radio is an abstract class. But we have written an attribute called lookup-mehod in the radio bean configuration. Now spring will understand that I have to lookup for the method whose name is lookupReciver and goes to the bean called reciver10Hz. Create the object of bean reciver10Hz. Now created bean is given to the lookupReciver by creating the object of Radio.

Note:-IOC container will only create the object of abstract Class (bean) when we have configured lookup-method attribute.

Actually IOC container will do the following when it is asked for lookup-method injection.
First it will create a proxy class like following-

```
class Radio$Proxy extends Radio
{
public IReciver LookupReciver
{
IReciver reciver=factory.getBean("reciver10Hz",IReciver.class);
return reciver;
}
}
```

While overriding the LookupReciver method it will call the bean definition of those bean which are configured in configuration file bean="reciver10Hz" now object of bean reciver10Hz is created and stored in the reference reciver.

Now IOC will create the object of Radio$Proxy class and store it as radio in the IOC container and return its reference to us. Now when we call the listen method of Radio class with reference radio actually we are calling the Radio$Proxy class method listen. In this way Method injection is actually processed.