20_Spring IOC 依赖来源有哪些？

1、开篇

前面两节课介绍了 Spring IOC 的依赖查找和依赖注入的两种方式，实际上针对这两种方式都对应了依赖来源。今天针对 Spring IOC 的依赖来源展开介绍，包括如下内容：

- 自定义 Bean 作为依赖来源
- 容器内建 Bean 对象作为依赖来源
- 容器内建依赖作为依赖来源

2、自定义 Bean 作为依赖来源

这部分的依赖来源最容易理解，如图 1 所示在前面的课程也用到了这个例子，TestUtil 中引用了 User 的 Bean 这里的 TestUtil 可以在 XML 中进行定义。

```
public class TestUtil {

    private User user;
    public User getUser() {
        return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
}
```

图 1 自定义 Bean

如图 2 所示 TestUtil 在 XML 文件中进行了定义 bean 的 id 为“testUtil”。

```xml
<?xml version="1.0"encoding="UTF-8"?>
<beans xmlns="//www.springframework.org/schema/beans"
       xmlns:xsi="//www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="//www.springframework.org/schema/beans
        //www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="user" class="org.spring.ioc.overview.dependency.domain.User">
        <property name="id" value="1"/>
        <property name="name" value="小明"/>
    </bean>

    <bean id="testUtil" class="org.spring.ioc.overview.dependency.domain.TestUtil" autowire="byType"/>


</beans>
```

图 2 XML 中定义的 Bean

这类就属于用户自定义的 Bean，通常会在 XML 文件中进行定义。可以通过 beanFactory.getBean()的方式查找依赖。

如图 3 所示，直接通过 BeanFactory 中的 getBean 方法就可以获取 Bean 的实例。

```java
public static void main(String[] args) {
    BeanFactory beanFactory = new ClassPathXmlApplicationContext("classpath:/META-INF/dependency-lookup.xml");
    TestUtil testUtil = (TestUtil) beanFactory.getBean("testUtil");
    User user = testUtil.getUser();
    System.out.println(user);
}
```

图 3 获取自定义 Bean 实例

3、容器内建 Bean 对象作为依赖来源

容器内建的 Bean 对象并不是我们创建的而是 Spring 内部实例化产生的，说白了就是 Spring 自己需要使用从而创建的 Bean。同样也可以通过 BeanFactory.getBean()的方式进行依赖查找。

如图 4 所示，通过 BeanFactory 类中的 getBean 方法可以查询 Environment Bean 对象的依赖。

```java
public static void main(String[] args) {
    BeanFactory beanFactory = new ClassPathXmlApplicationContext("classpath:/META-INF/dependency-lookup.xml");
    Environment environment = beanFactory.getBean(Environment.class);
    System.out.println("获取 Environment 类型的 Bean: " + environment);
}
```

图 4 查找 Environment Bean 对象的依赖

和 Environment 对象相似的 Spring 内建 Bean 对象还有很多，如图 5 所示，这里列举了对应的内建对象和使用场景。

- Environment 对象，在外部化配置和 Profiles 场景使用。
- Java.util.Properties 对象，被用到 Java 系统属性场景。
- Java.util.Map 对象，可以使用到操作系统环境变量的场景。
- MessageSource 对象，用作国际化文案。
- LifecycleProcessor 对象，用来处理 Lifecycle Bean 处理器。
- ApplicationEventMulticaster 对象，用来处理 Spring 时间广播器。

Bean 名称	Bean 实例	使用场景
environment	Environment 对象	外部化配置以及 Profiles
systemProperties	java.util.Properties 对象	Java 系统属性
systemEnvironment	java.util.Map 对象	操作系统环境变量
messageSource	MessageSource 对象	国际化文案
lifecycleProcessor	LifecycleProcessor 对象	Lifecycle Bean 处理器
applicationEventMulticaster	ApplicationEventMulticaster 对 象	Spring 事件广播器

图 5 Spring 内建 Bean 对象

4、容器内建依赖作为依赖来源

容器内建依赖也称作内建非 Bean 对象，为什么这么说呢？因为这类对象并不是 Spring 的 Bean，比如说 BeanFactory 就是这类对象，但是这类对象是 Spring 启动必须的组件，也就是基础组件。基于以上原因容器内建依赖是不能通过 BeanFactory.getBean()的方式依赖查找出来的。

如图 6 所示，这里 testUtil 的 getBeanFactory 与 beanFactory 进行比较结果就为“false”，证明不是同一个 beanFactory。

```
BeanFactory beanFactory = new ClassPathXmlApplicationContext("classpath:/META-INF/dependency-lookup.xml");
TestUtil testUtil = (TestUtil) beanFactory.getBean("testUtil");
System.out.println(testUtil.getBeanFactory() == beanFactory);
```

图 6 两个 beanFactory 比较

顺着上面的思路继续，如图 7 所示，如果通过 BeanFactory 的 getBean 方法获取 BeanFactory 的实例会报错，也证明了 BeanFactory 不是一个内建 Bean 对象，而是内建的依赖。

```
System.out.println(beanFactory.getBean(BeanFactory.class));
```

图 7 通过 getBean 获取 BeanFactory.class

对于容器内建依赖具有一下特征：无生命周期管理，无法实现延迟初始化 Bean，无法通过依赖查找。此类对象包括：BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext 从名字上看都是系统级别的对象。如图 8 所示，如果一定要对这些变量进行注册，可以使用 ConfigurableListableBeanFactory 中的 registerResolvableDependency 方法。

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    // Tell the internal bean factory to use the context's class loader etc.
    beanFactory.setBeanClassLoader(getClassLoader());
    beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
    beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar( resourceLoader: this, getEnvironment()));

    // Configure the bean factory with context callbacks.
    beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
    beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
    beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
    beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
    beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
    beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
    beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

    // BeanFactory interface not registered as resolvable type in a plain factory.
    // MessageSource registered (and found for autowiring) as a bean.
    beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
    beanFactory.registerResolvableDependency(ResourceLoader.class, autowiredValue: this);
    beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, autowiredValue: this);
    beanFactory.registerResolvableDependency(ApplicationContext.class, autowiredValue: this);
```

图 8 注册容器内建依赖

5、总结

本节课延续了 Spring IOC 的依赖查找和依赖注入的话题，聊了 Spring IOC 的依赖来源，分别是自定义 Bean、容器内建 Bean、容器内建依赖。下节课会继续

Spring IOC 的基本原理，内容会延伸到 JDK 动态代理和 CGLib 动态代理两种模式。下期见，拜拜。