18_源码剖析：@AutoWired 注解依赖注入的原理？

1、开篇

上节课介绍了 Spring IOC 依赖注入的两种方式，分别是根据 Bean 名称注入和根据 Bean 类型注入。在上节课代码基础上增加了 TestUtil 类，同时修改了 XML 文件，在增加的新 bean 节点中使用了 autowire 属性（byName/byType）完成自动装配的功能。本节课会介绍通过源码剖析的方式了解@AutoWired 注解依赖注入的原理。包括以下内容：

- @Autowire 的定义
- @Autowire 的工作原理

2、@Autowire 的定义

通常来说我们会使用@Autowired 进行依赖注入的操作，如图 1 所示通过@Autowired 注解的源码定义看出，它被标注可以作用于：构造函数(Constructor)、方法(Method)、参数(Parameter)、字段(Field)、注释(Annotation)，也就是针对这些描述都可以通过@Autowired 的方式实现自动的依赖注入。

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {

    /**
     * Declares whether the annotated dependency is required.
     * <p>Defaults to {@code true}.
     */
    boolean required() default true;

}
```

图 1 Autowired 定义

3、@Autowire 的工作原理

@Autowired 注解是由 AutowiredAnnotationBeanPostProcessor 实现的，如图 2 所示，查看该类的源码会发现它实现了 MergedBeanDefinitionPostProcessor 接口。

```
public class AutowiredAnnotationBeanPostProcessor extends InstantiationAwareBeanPostProcessorAdapter
        implements MergedBeanDefinitionPostProcessor, PriorityOrdered, BeanFactoryAware {
```

图 2 AutowiredAnnotationBeanPostProcessor 实现了

MergedBeanDefinitionPostProcessor

如图 3 所示，同时 AutowiredAnnotationBeanPostProcessor 实现了 MergedBeanDefinitionPostProcessor 中的 postProcessMergedBeanDefinition 方法，@Autowired 注解正是通过这个方法实现注入类型的预解析，将需要依赖注入的属性信息封装到 InjectionMetadata 类中。

```
@Override
public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?> beanType, String beanName) {
    if (beanType != null) {
        InjectionMetadata metadata = findAutowiringMetadata(beanName, beanType,  pvs: null);
        metadata.checkConfigMembers(beanDefinition);
    }
}
```

图 3 postProcessMergedBeanDefinition 方法

如图 4 所示，InjectionMetadata 类中包含了需要注入的元素(injectedElements)及元素要注入的目标类(tagetClass)。

```
public class InjectionMetadata {

    private static final Log logger = LogFactory.getLog(InjectionMetadata.class);

    private final Class<?> targetClass;

    private final Collection<InjectedElement> injectedElements;

    private volatile Set<InjectedElement> checkedElements;
```

图 4 InjectionMetadata

既然 AutowiredAnnotationBeanPostProcessor 是实现 Autowire 的主力类，那么如何执行它实现依赖注入的呢？Spring 容器在启动的时候会执行 AbstractApplicationContext 类的 refresh 方法，如图 5 所示，refresh 方法中 registerBeanPostProcessors(beanFactory)完成了对 AutowiredAnnotationBeanPostProcessor 的注册，当执行 finishBeanFactoryInitialization(beanFactory)方法对非延迟初始化的单例 bean 进

行初始化时，会执行到 AbstractAutowireCapableBeanFactory 类的 doCreateBean 方法。

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
            finishRefresh();
        }
```

图 5 refresh 方法

如图 6 所示，doCreateBean 方法中会去调用 applyMergedBeanDefinitionPostProcessors 方法。

```
// Allow post-processors to modify the merged bean definition.
synchronized (mbd.postProcessingLock) {
    if (!mbd.postProcessed) {
        try {
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Post-processing of merged bean definition failed", ex);
        }
        mbd.postProcessed = true;
    }
}
```

图 6 applyMergedBeanDefinitionPostProcessors 方法

如图 7 所示，查看 postProcessMergedBeanDefinition 方法的具体实现类，会发现调用的是 AutowiredAnnotationBeanPostProcessor 类的 postProcessMergedBeanDefinition 方法，这个方法中完成了对注入元素注解的预解析。

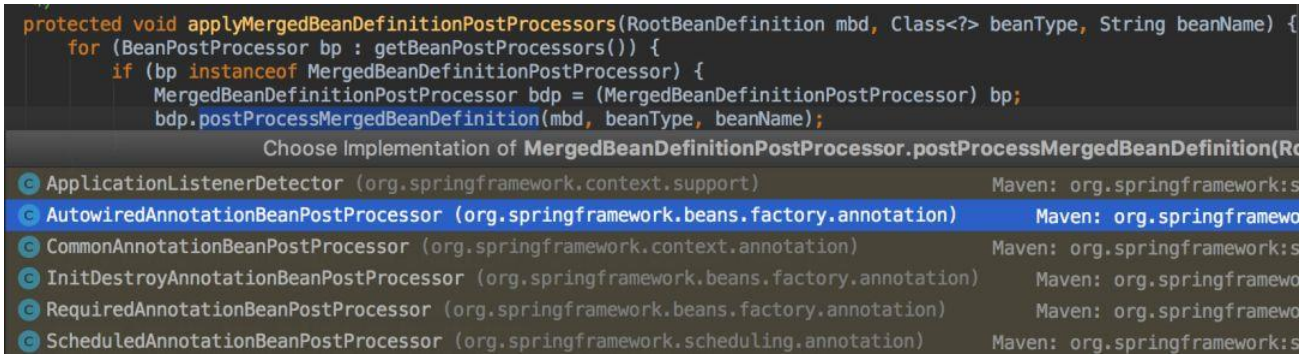

图 7 postProcessMergedBeanDefinition 方法

如图 8 所示，还是把眼光移回 doCreateBean 方法中，其中会执行 populateBean 方法实现对属性的注入。

```
// Initialize the bean instance.
Object exposedObject = bean;
try {
    populateBean(beanName, mbd, instanceWrapper);
    if (exposedObject != null) {
        exposedObject = initializeBean(beanName, exposedObject, mbd);
    }
}
```

图 8 doCreateBean 方法中调用 populateBean 方法

如图 9 所示，populateBean 方法中会遍历所有注册过的 BeanPostProcessor 接口实现类的实例，如果实例属于 InstantiationAwareBeanPostProcessor 类型的，则执行实例类的 postProcessPropertyValues 方法。

```
if (hasInstAwareBpps || needsDepCheck) {
    PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
    if (hasInstAwareBpps) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);
                if (pvs == null) {
                    return;
                }
            }
        }
    }
    if (needsDepCheck) {
        checkDependencies(beanName, mbd, filteredPds, pvs);
    }
}

applyPropertyValues(beanName, mbd, bw, pvs);
```

图 9 populateBean 方法内部

如图 10 所示，进入到 postProcessPropertyValues 方法内部。由于 AutowiredAnnotationBeanPostProcessor 继承了 InstantiationAwareBeanPostProcessorAdapter，而 AutowiredAnnotationBeanPostProcessor 间接实现了 InstantiationAwareBeanPostProcessor 接口，所以这里会执行到 AutowiredAnnotationBeanPostProcessor 类的 postProcessPropertyValues 方法。其中会调用 metadata.inject(bean, beanName, pvs)代码进行注入操作。

```
@Override
public PropertyValues postProcessPropertyValues(
        PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeanCreationException {

    InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(), pvs);
    try {
        metadata.inject(bean, beanName, pvs);
    }
    catch (BeanCreationException ex) {
        throw ex;
    }
    catch (Throwable ex) {
        throw new BeanCreationException(beanName, "Injection of autowired dependencies failed", ex);
    }
    return pvs;
}
```

图 10 postProcessPropertyValues 方法

如图 11 所示在调用 metadata 中的 inject 方法时，会遍历所有的元素针对每个元素执行 inject 方法。InjectedElement 有两个子类，分别是 AutowiredFieldElement 和 AutowiredMethodElement。AutowiredFieldElement 用于对标注在属性上的注入，AutowiredMethodElement 用于对标注在方法上的注入。

```
public void inject(Object target, String beanName, PropertyValues pvs) throws Throwable {
    Collection<InjectedElement> elementsToIterate =
            (this.checkedElements != null ? this.checkedElements : this.injectedElements);
    if (!elementsToIterate.isEmpty()) {
        boolean debug = logger.isDebugEnabled();
        for (InjectedElement element : elementsToIterate) {
            if (debug) {
                logger.debug( o: "Processing injected element of bean '" + beanName + "': " + element);
            }
            element.inject(target, beanName, pvs);
        }
    }
}
```

图 11 inject 方法

上面的两种方式的注入过程都差不多，根据需要注入的元素的描述信息，按类型或名称查找需要的依赖值，如果依赖没有实例化先实例化依赖，然后使用反射进行赋值。图 12 是 field 注入的场景，图 13 是方法注入的场景。

```
if (value != null) {
    ReflectionUtils.makeAccessible(field);
    field.set(bean, value);
}
```

图 12 field 注入

```
if (arguments != null) {
    try {
        ReflectionUtils.makeAccessible(method);
        method.invoke(bean, arguments);
    }
    catch (InvocationTargetException ex){
        throw ex.getTargetException();
    }
}
```

图 13 method 注入

至此将@AutoWired 注入的过程和源代码带大家过了一遍。

4、总结

这节课介绍了@AutoWired 可以对构造函数(Constructor)、方法(Method)、参数(Parameter)、字段(Field)、注释(Annotation)进行自动依赖注入，同时通过源码分析将这一过程带大家走了一遍。下节课会介绍@Resource 和@AutoWired 的区别。下期见，拜拜。