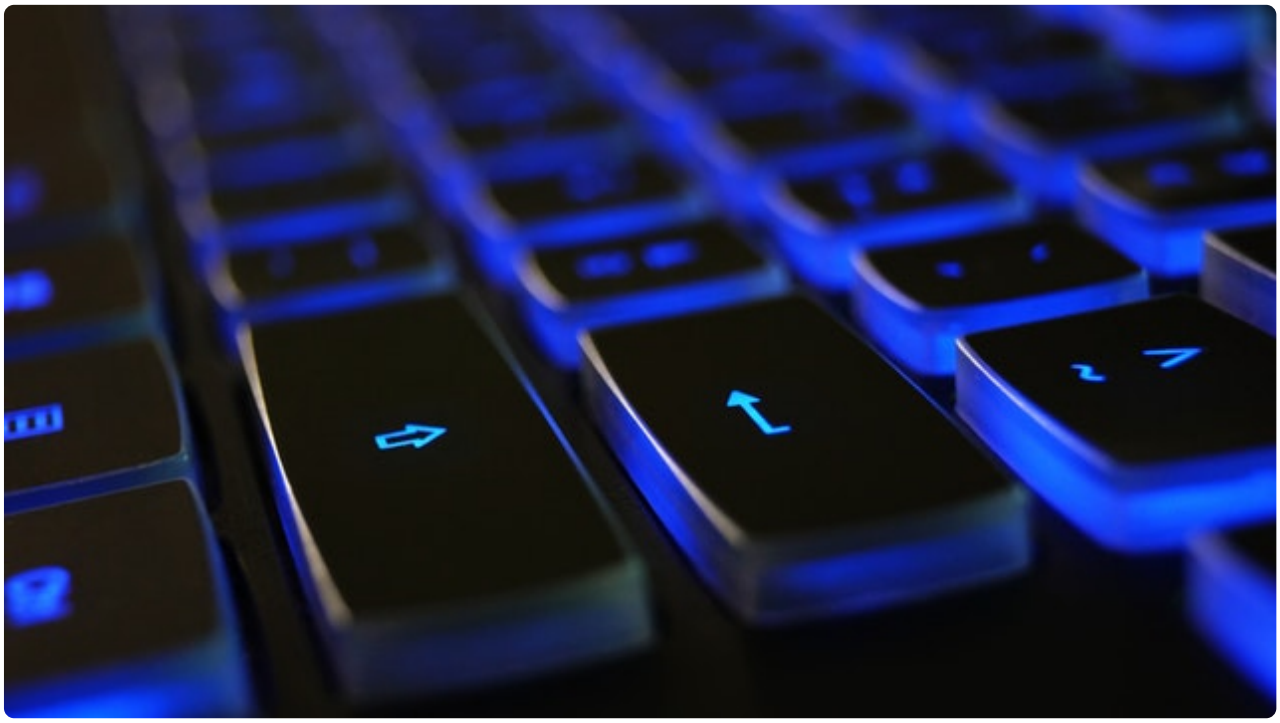
12 @Autowired是如何工作的？--Spring注解源码深度揭秘

更新时间：2020-08-04 13:40:15



> 人生的价值，并不是用时间，而是用深度去衡量的。——列夫·托尔斯泰

背景

注解可以减少代码的开发量，Spring 提供了丰富的注解功能。我们可能会被问到，Spring 的注解到底是怎么触发的呢？今天以 Spring 最常使用的一个注解 Autowired 来跟踪代码，进行 debug。

Autowired 的定义及作用

作用：Marks a constructor, field, setter method or config method as to be autowired by Spring's dependency injection facilities.

定义：

```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD, ElementType.METHOD, 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;
}
```

@Target

@Target 起到什么作用呢？我们来看 jdk 的说明：

> Indicates the kinds of program element to which an annotation type is applicable. If a Target meta-annotation is not present on an annotation type declaration, the declared type may be used on any program element. If such a meta-annotation is present, the compiler will enforce the specified usage restriction.

@Target 限定注解可应用于的程序元素的类型。如果目标元注解没有出现在注解类型声明中，则声明的类型可以在任何程序元素上使用。如果存在这样的元注释，编译器将强制执行指定的使用限制。

其次，@Target 定义：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
 ElementType[] value();
}
```

注解表明适用于哪种程序元素，程序元素有以下几种：

```
public enum ElementType {
 /** Class, interface (including annotation type), or enum declaration */
 TYPE,
 /** Field declaration (includes enum constants) */
 FIELD,
 /** Method declaration */
 METHOD,
 /** Parameter declaration */
 PARAMETER,
 /** Constructor declaration */
 CONSTRUCTOR,
 /** Local variable declaration */
 LOCAL_VARIABLE,
 /** Annotation type declaration */
 ANNOTATION_TYPE,
 /** Package declaration */
 PACKAGE
}
```

最后，Target 使用方式：

```
@Target(value={ANNOTATION_TYPE})
```

@Retention

官方说明：

> Indicates how long annotations with the annotated type are to be retained. If no Retention annotation is present on an annotation type declaration, the retention policy defaults to RetentionPolicy.CLASS

简单的说就是作用域。

然后我们看其定义：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
 RetentionPolicy value();
}
```

RetentionPolicy 作用域定义：

```
public enum RetentionPolicy {
 /**
 * Annotations are to be discarded by the compiler.
 */
 SOURCE,
 /**
 * Annotations are to be recorded in the class file by the compiler
 * but need not be retained by the VM at run time. This is the default
 * behavior.
 */
 CLASS,
 /**
 * Annotations are to be recorded in the class file by the compiler and
 * retained by the VM at run time, so they may be read reflectively.
 *
 * @see java.lang.reflect.AnnotatedElement
 */
 RUNTIME
}
```

最后，使用方式：

```
@Retention(value=RetentionPolicy )
```

@Documented

官方说明：

> Indicates that annotations with a type are to be documented by javadoc and similar tools by default. This type should be used to annotate the declarations of types whose annotations affect the use of annotated elements by their clients. If a type declaration is annotated with Documented, its annotations become part of the public API of the annotated elements.

简单的说，可以成为 API 的一部分。

定义:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Documented {
}
```

使用方式：

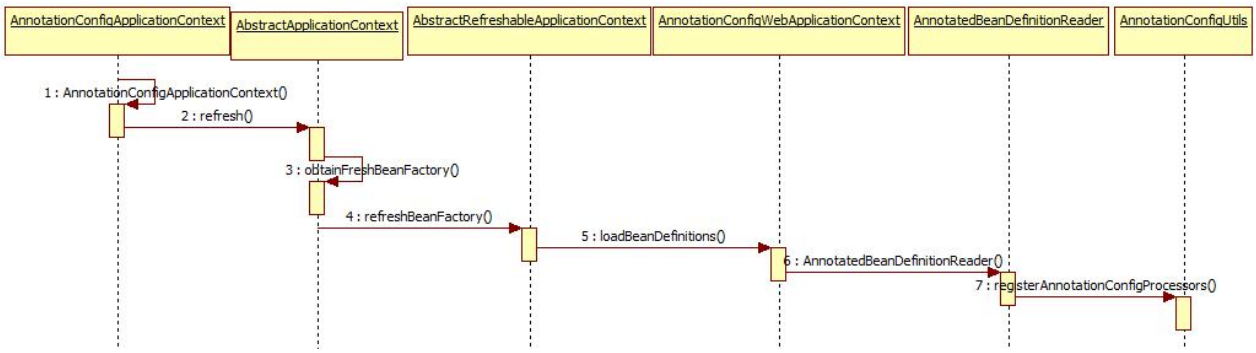
@Documented

注解实现类 AutowiredAnnotationBeanPostProcessor 的注册

它的注册方式有两种：

一种使用 xml 配置文件方式，即 <context:annotation-config/>，它通过 AnnotationConfigBeanDefinitionParser 解析器来将 AutowiredAnnotationBeanPostProcessor 注册到容器中，后面讨论@Resource 注解会详细论述。

一种是使用 AnnotationConfigWebApplicationContext 容器来启动应用，通过 AnnotatedBeanDefinitionReader 将 AutowiredAnnotationBeanPostProcessor 注册到容器中。它的注册过程如下：



注册的注解相关处理器代码如下：

```
/**
* Register all relevant annotation post processors in the given registry.
* @param registry the registry to operate on
* @param source the configuration source element (already extracted)
* that this registration was triggered from. May be {@code null}.
* @return a Set of BeanDefinitionHolders, containing all bean definitions
* that have actually been registered by this call
*/
public static Set<BeanDefinitionHolder> registerAnnotationConfigProcessors(
BeanDefinitionRegistry registry, Object source) {
    DefaultListableBeanFactory beanFactory = unwrapDefaultListableBeanFactory(registry);
    if (beanFactory != null) {
     if (!(beanFactory.getDependencyComparator() instanceof AnnotationAwareOrderComparator)) {
     beanFactory.setDependencyComparator(AnnotationAwareOrderComparator.INSTANCE);
     }
     if (!(beanFactory.getAutowireCandidateResolver() instanceof  ContextAnnotationAutowireCandidateResolver)) {
     beanFactory.setAutowireCandidateResolver(new ContextAnnotationAutowireCandidateResolver());
     }
    }
    Set<BeanDefinitionHolder> beanDefs = new LinkedHashSet<BeanDefinitionHolder>(4);
    if (!registry.containsBeanDefinition(CONFIGURATION_ANNOTATION_PROCESSOR_BEAN_NAME)) {
     RootBeanDefinition def = new RootBeanDefinition(ConfigurationClassPostProcessor.class);
     def.setSource(source);
     beanDefs.add(registerPostProcessor(registry, def, CONFIGURATION_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    if (!registry.containsBeanDefinition(AUTOWIRED_ANNOTATION_PROCESSOR_BEAN_NAME)) {
     RootBeanDefinition def = new RootBeanDefinition(AutowiredAnnotationBeanPostProcessor.class);
     def.setSource(source);
     beanDefs.add(registerPostProcessor(registry, def, AUTOWIRED_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    if (!registry.containsBeanDefinition(REQUIRED_ANNOTATION_PROCESSOR_BEAN_NAME)) {
     RootBeanDefinition def = new RootBeanDefinition(RequiredAnnotationBeanPostProcessor.class);
     def.setSource(source);
     beanDefs.add(registerPostProcessor(registry, def, REQUIRED_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    // Check for JSR-250 support, and if present add the CommonAnnotationBeanPostProcessor.
    if (jsr250Present && !registry.containsBeanDefinition(COMMON_ANNOTATION_PROCESSOR_BEAN_NAME)) {
     RootBeanDefinition def = new RootBeanDefinition(CommonAnnotationBeanPostProcessor.class);
     def.setSource(source);
     beanDefs.add(registerPostProcessor(registry, def, COMMON_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    // Check for JPA support, and if present add the PersistenceAnnotationBeanPostProcessor.
    if (jpaPresent && !registry.containsBeanDefinition(PERSISTENCE_ANNOTATION_PROCESSOR_BEAN_NAME)) {
     RootBeanDefinition def = new RootBeanDefinition();
        try {
          def.setBeanClass(ClassUtils.forName(PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME,
          AnnotationConfigUtils.class.getClassLoader()));
        }
     catch (ClassNotFoundException ex) {
     throw new IllegalStateException(
 "Cannot load optional framework class: " + PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME, ex);
     }
     def.setSource(source);
     beanDefs.add(registerPostProcessor(registry, def, PERSISTENCE_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    return beanDefs;
}
```

通过代码，我们可以看到注册的注解处理器有：

org.springframework.context.annotation.internalConfigurationAnnotationProcessor 对应于 ConfigurationClassPostProcessor；

org.springframework.context.annotation.internalAutowiredAnnotationProcessor 对应于 AutowiredAnnotationBeanPostProcessor；

org.springframework.context.annotation.internalRequiredAnnotationProcessor 对应于 RequiredAnnotationBeanPostProcessor；

org.springframework.context.annotation.internalCommonAnnotationProcessor 对应于 CommonAnnotationBeanPostProcessor；

org.springframework.context.annotation.internalPersistenceAnnotationProcessor 对应于 PersistenceAnnotationBeanPostProcessor；

注解实现类 **AutowiredAnnotationBeanPostProcessor** 使用

@Autowired 注解，定义了两种主要的注入类型：

1. 基于构造方法注入

其调用链如下：

AbstractAutowireCapableBeanFactory#createBean()

AbstractAutowireCapableBeanFactory#doCreateBean()

AbstractAutowireCapableBeanFactory#createBeanInstance()

AbstractAutowireCapableBeanFactory#determineConstructorsFromBeanPostProcessors()

AutowiredAnnotationBeanPostProcessor#determineConstructorsFromBeanPostProcessors()

2. 基于成员变量自动注入

其调用链如下：

AbstractAutowireCapableBeanFactory#createBean()

AbstractAutowireCapableBeanFactory#doCreateBean()

AbstractAutowireCapableBeanFactory#populateBean()

AutowiredAnnotationBeanPostProcessor#postProcessPropertyValues()

InjectionMetadata#inject()

AutowiredFieldElement#inject()/AutowiredMethodElement#inject()

小结

Spring 框架从创建伊始就致力于为复杂问题提供强大的、非侵入性的解决方案。开始时 Spring 使用 XML 配置文件数量引入定制命名空间功能。从 Spring 2.5 推出了一整套注解，作为基于 XML 的配置的可选方案。注解可用于 Spring 管理对象的自动发现、依赖注入、生命周期方法、Web 层配置和单元/集成测试。

Spring 注解确实提高了开发效率，但一直以来，很多开发者对 Spring 注解的工作原理都处于一知半解的状态，使用注解过程中碰到问题，也是通过搜索或者多次尝试的方式来验证，为了更好的工作，还是要把 how Spring annotation works 这件事做起来。

}

11 spring IoC容器中事件event消息的发送和接收内部工作原理揭秘

13 我的人生我做主揭秘Bean的生命周期