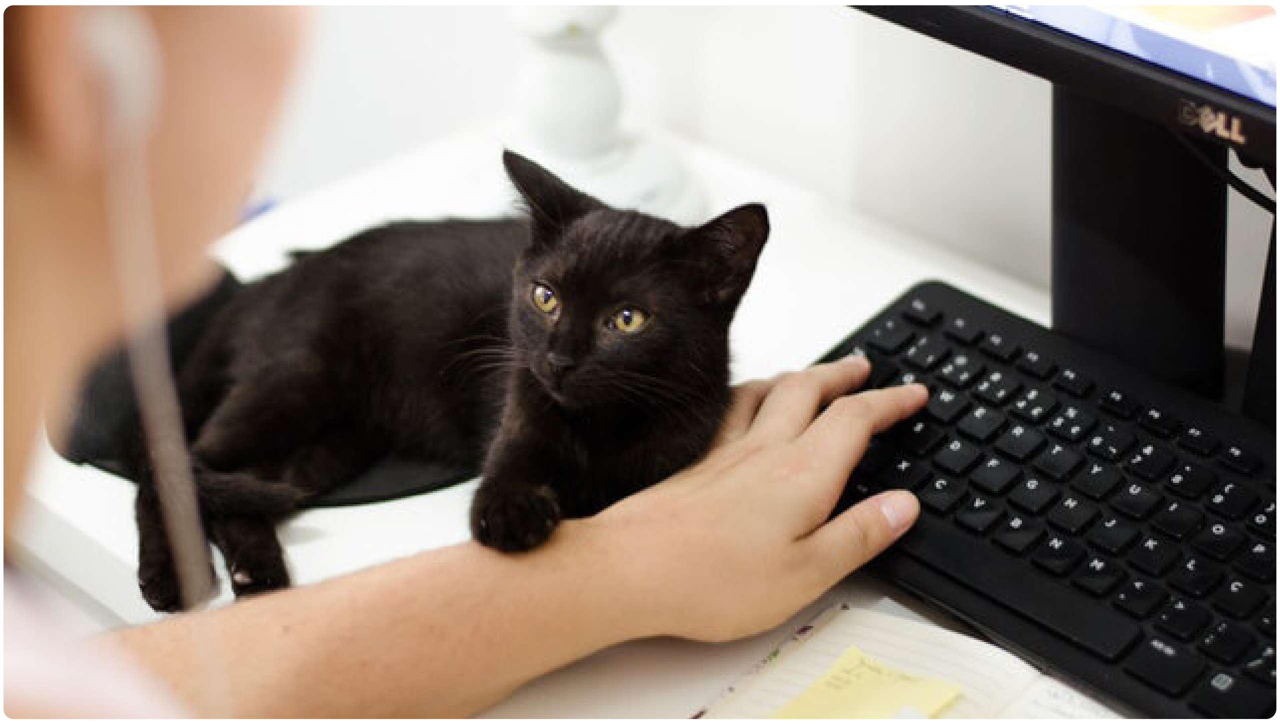
20 面试官，请不要再问我@Resource和@Autowire注解的区别了

更新时间：2020-08-04 17:54:54



谁和我一样用功，谁就会和我一样成功。——莫扎特

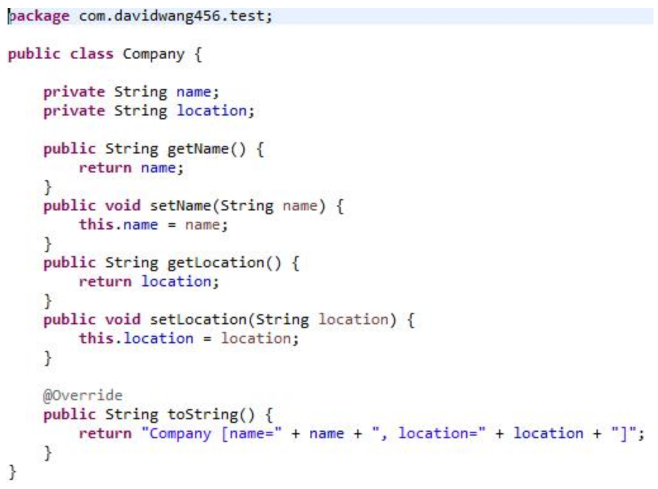
背景

我们平时在写业务代码时会使用一些注解，注解给我们的业务提供了很多便利，但开发者往往不知道这些注解背后的故事，正如本文要讲到的 @Resource 注解一样，如果理解了实现原理，对我们开发由莫大的好处：一是可以更好的利用这些注解及它们的使用场景，不会出现使用错误；二是排查问题更有目标性，提升开发效率；三是面试时再也不担心面试官问我 @Resource 和 @Autowire 的区别了。

本专栏的另一篇文章：《@Autowired 是如何工作的？——Spring 注解源码深度揭秘》介绍了 Autowired 的实现机制，本篇就重点介绍 @Resource 注解的使用及实现原理。

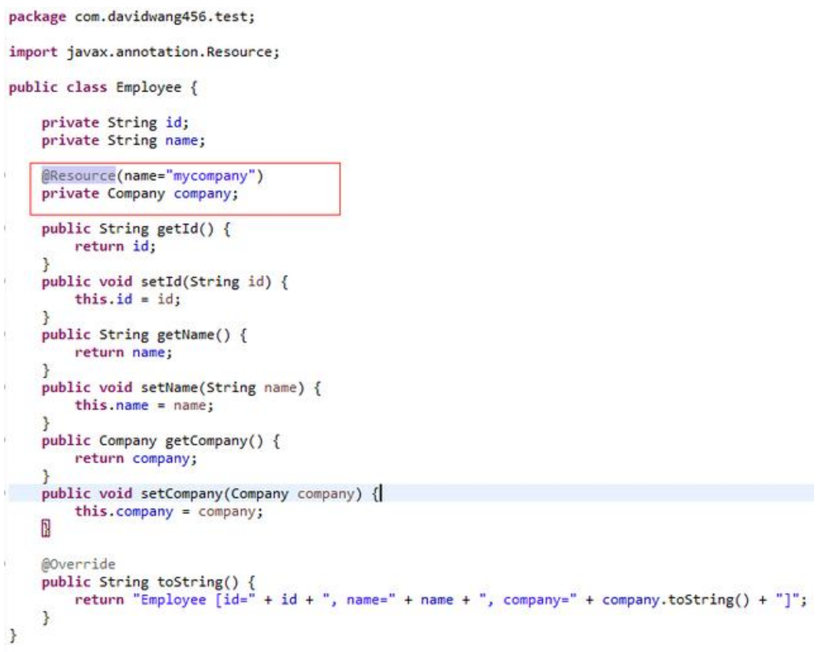
实例

用 @Resource 注解导入的 POJO 类：



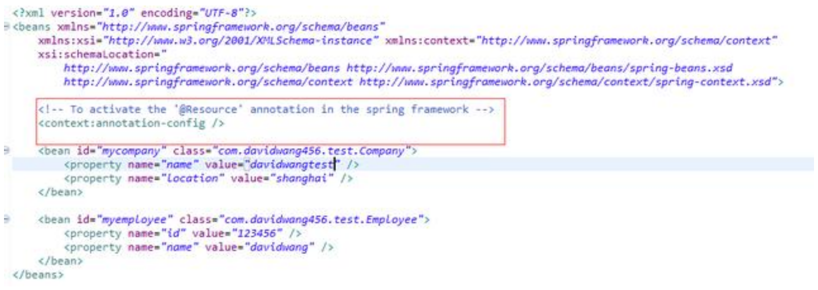
```java
package com.davidwang456.test;

public class Company {

    private String name;
    private String location;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getLocation() {
        return location;
    }
    public void setLocation(String location) {
        this.location = location;
    }

    @Override
    public String toString() {
        return "Company [name=" + name + ", location=" + location + "]";
    }
}
```

导入 Resource 注解类：



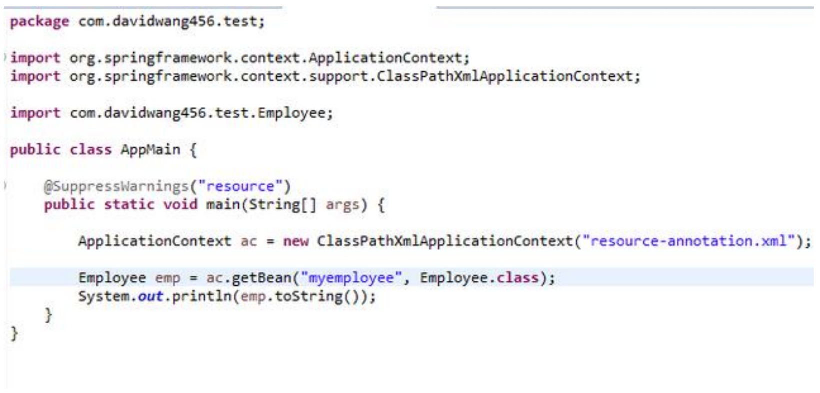
```java
package com.davidwang456.test;

import javax.annotation.Resource;

public class Employee {

    private String id;
    private String name;

    @Resource(name="mycompany")
    private Company company;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Company getCompany() {
        return company;
    }
    public void setCompany(Company company) {
        this.company = company;
    }

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", company=" + company.toString() + "]";
    }
}
```

配置文件中使 @Resource 注解生效：



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

    <!-- To activate the '@Resource' annotation in the spring framework -->
    <context:annotation-config />

    <bean id="mycompany" class="com.davidwang456.test.Company">
        <property name="name" value="davidwangtest" />
        <property name="location" value="shanghai" />
    </bean>

    <bean id="myemployee" class="com.davidwang456.test.Employee">
        <property name="id" value="123456" />
        <property name="name" value="davidwang" />
    </bean>
</beans>
```

测试类：



```
package com.davidwang456.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.davidwang456.test.Employee;

public class AppMain {

    @SuppressWarnings("resource")
    public static void main(String[] args) {

        ApplicationContext ac = new ClassPathXmlApplicationContext("resource-annotation.xml");

        Employee emp = ac.getBean("myemployee", Employee.class);
        System.out.println(emp.toString());
    }
}
```

运行结果：

```
Employee [id=123456, name=davidwang, company=Company [name=davidwangtest, location=shanghai]]
```

原理

配置文件中使用了 annotation-config 来开启 @Resource 注解，那么怎么实现的呢？

```
<!-- To activate the '@Resource' annotation in the spring framework -->
<context:annotation-config />
```

1. xml 配置文件解析处理器 ContextNamespaceHandler

ContextNamespaceHandler 定义了 XML 配置文件中元素的解析器，这些元素分别是：

`<context:property-placeholder/>` 元素的解析器 PropertyPlaceholderBeanDefinitionParser；

`<context:property-override>` 元素的解析器 PropertyOverrideBeanDefinitionParser；

`<context:annotation-config>` 元素的解析器 AnnotationConfigBeanDefinitionParser；

`<context:component-scan/>` 元素的解析器 ComponentScanBeanDefinitionParser；

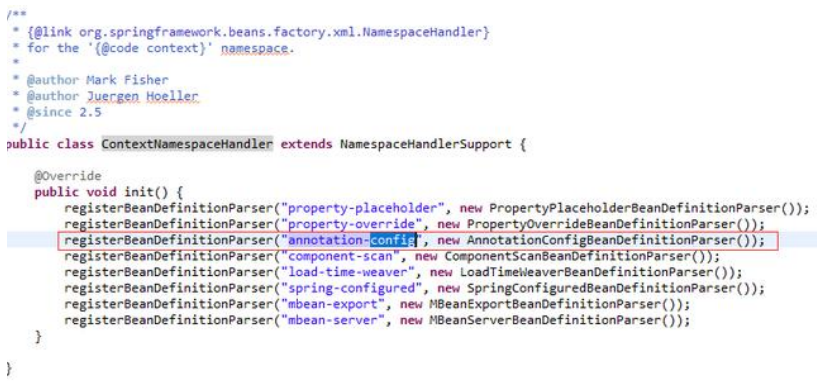
`<context:load-time-weaver/>` 元素的解析器 LoadTimeWeaverBeanDefinitionParser；

`<context:spring-configured/>` 元素的解析器 SpringConfiguredBeanDefinitionParser；

`<context:mbean-export/>` 元素的解析器 MBeanExportBeanDefinitionParser；

`<context:mbean-server/>` 元素的解析器 MBeanServerBeanDefinitionParser。

上面每个元素的用法如果想深入了解，需要参照相关资料。本节仅讨论 `<context:annotation-config>` 元素的作用。



```
/**
 * {@link org.springframework.beans.factory.xml.NamespaceHandler}
 * for the '{@code context}' namespace.
 *
 * @author Mark Fisher
 * @author Juergen Hoeller
 * @since 2.5
 */
public class ContextNamespaceHandler extends NamespaceHandlerSupport {

    @Override
    public void init() {
        registerBeanDefinitionParser("property-placeholder", new PropertyPlaceholderBeanDefinitionParser());
        registerBeanDefinitionParser("property-override", new PropertyOverrideBeanDefinitionParser());
        registerBeanDefinitionParser("annotation-config", new AnnotationConfigBeanDefinitionParser());
        registerBeanDefinitionParser("component-scan", new ComponentScanBeanDefinitionParser());
        registerBeanDefinitionParser("load-time-weaver", new LoadTimeWeaverBeanDefinitionParser());
        registerBeanDefinitionParser("spring-configured", new SpringConfiguredBeanDefinitionParser());
        registerBeanDefinitionParser("mbean-export", new MBeanExportBeanDefinitionParser());
        registerBeanDefinitionParser("mbean-server", new MBeanServerBeanDefinitionParser());
    }

}
```

<context:annotation-config/>的作用是向 Spring 容器注册各种 BeanPostProcessor，是为了让系统能够识别相应的注解。常用的 BeanPostProcessor 有以下几种：

ConfigurationClassPostProcessor：支持 **@Import** 注解；

AutowiredAnnotationBeanPostProcessor：支持 **@Autowired**， **@Value**，**@Inject** 等；

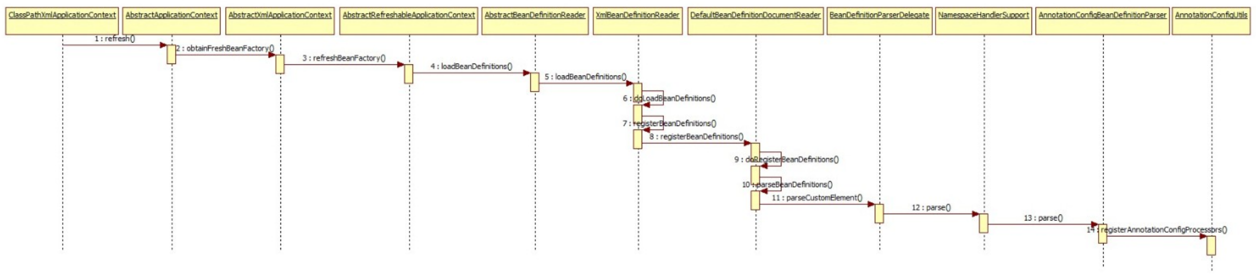
CommonAnnotationBeanPostProcessor：支持 JSR-250 注解，其中包括 **@PostConstruct**，**@PreDestroy** 和 **@Resource** 注解；

PersistenceAnnotationBeanPostProcessor：支持 JPA 注解；

2. @Resource 注解实现

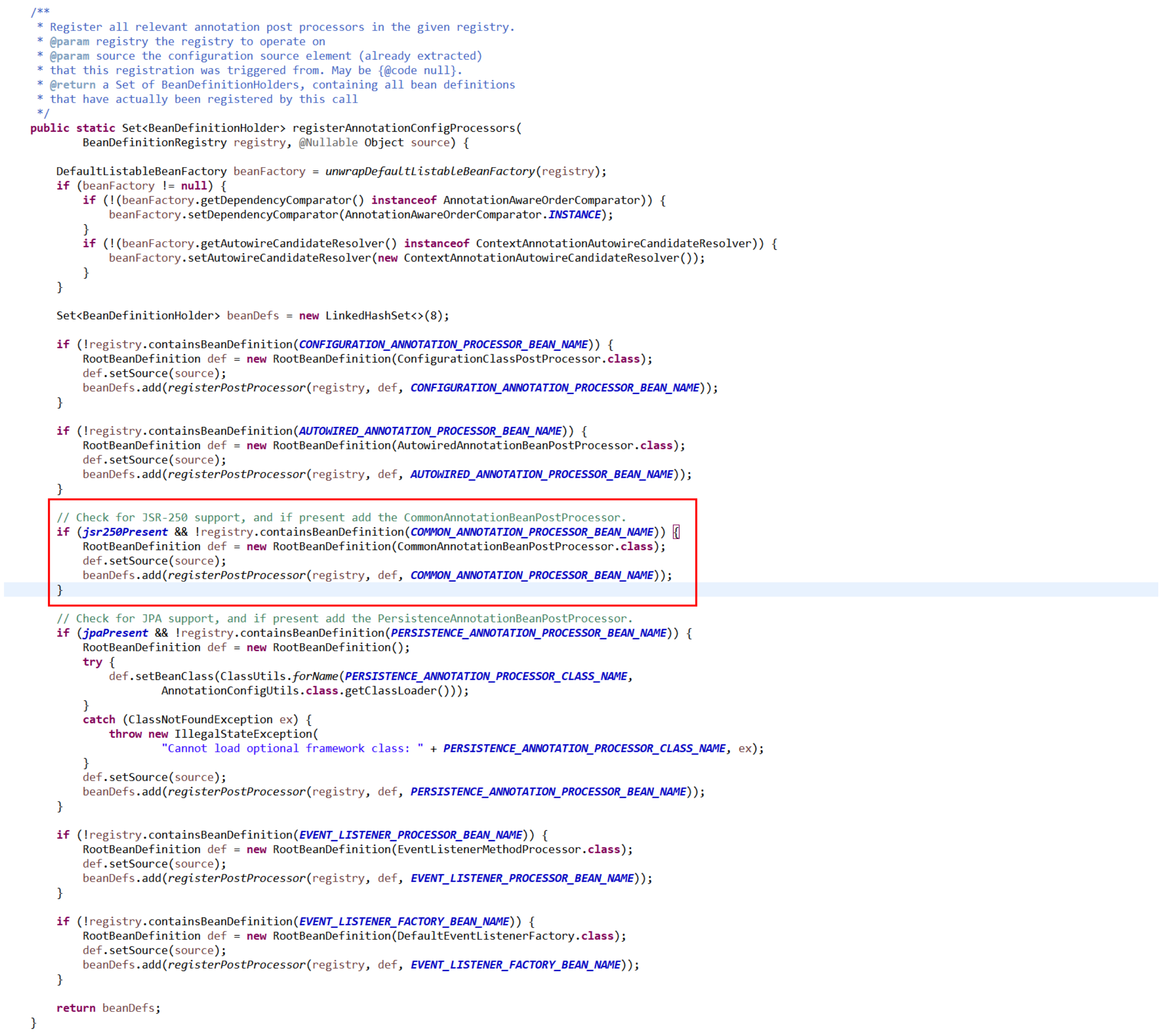
由 上面的描述，可以知道 CommonAnnotationBeanPostProcessor 实现了 **@Resource** 注解。

CommonAnnotationBeanPostProcessor 注册的过程如下图所示：


ClassPathXmlApplicationContext
AbstractApplicationContext
AbstractXmlApplicationContext
AbstractRefreshableApplicationContext
AbstractBeanDefinitionReader
XmlBeanDefinitionReader
DefaultBeanDefinitionDocumentReader
BeanDefinitionParserDelegate
NamespaceHandlerSupport
AnnotationConfigBeanDefinitionParser
AnnotationConfigUtils
1 : refresh()
2 : obtainFreshBeanFactory()
3 : refreshBeanFactory()
4 : loadBeanDefinitions()
5 : loadBeanDefinitions()
6 : doLoadBeanDefinitions()
7 : registerBeanDefinitions()
8 : registerBeanDefinitions()
9 : doRegisterBeanDefinitions()
10 : parseBeanDefinitions()
11 : parseCustomElement()
12 : parse()
13 : parse()
14 : registerAnnotationConfigProcessors()

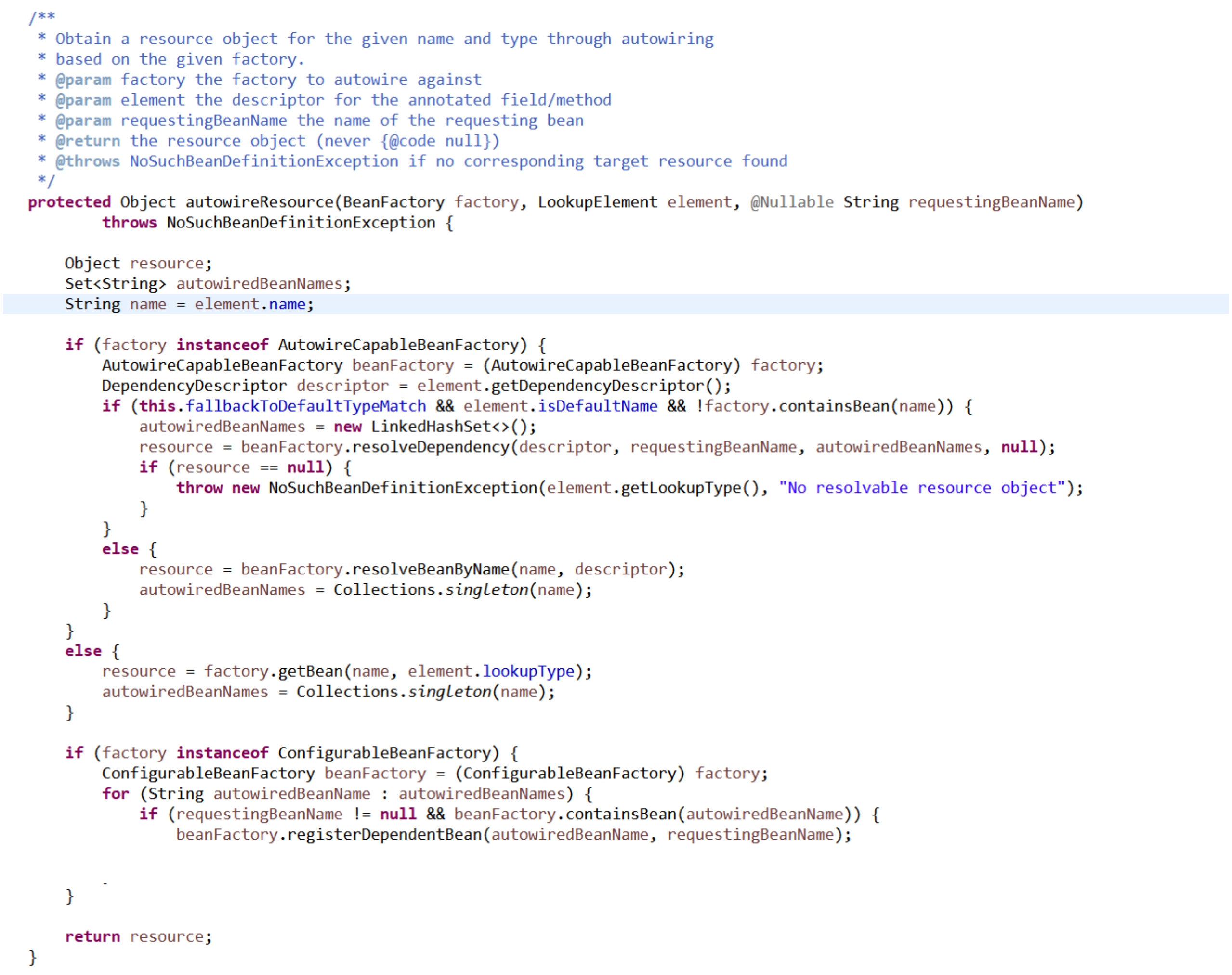
下面从源码看看链路的主要步骤：

<context:property-placeholder/>元素的解析器 AnnotationConfigBeanDefinitionParser 注册了 CommonAnnotationBeanPostProcessor，其代码如下：



```java
/**
 * 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, @Nullable 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<>(8);

    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));
    }

    // 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));
    }

    if (!registry.containsBeanDefinition(EVENT_LISTENER_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(EventListenerMethodProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_PROCESSOR_BEAN_NAME));
    }

    if (!registry.containsBeanDefinition(EVENT_LISTENER_FACTORY_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(DefaultEventListenerFactory.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_FACTORY_BEAN_NAME));
    }

    return beanDefs;
}
```

CommonAnnotationBeanPostProcessor 实现 @Resource 注解， 默认按 byName 自动注入。 既不指定 name 属性，也不指定 type 属性，则自动按 byName 方式进行查找。如果没有找到符合的 bean，则回退为一个原始类型进行进行查找，如果找到就注入。



```java
/**
 * Obtain a resource object for the given name and type through autowiring
 * based on the given factory.
 * @param factory the factory to autowire against
 * @param element the descriptor for the annotated field/method
 * @param requestingBeanName the name of the requesting bean
 * @return the resource object (never {@code null})
 * @throws NoSuchBeanDefinitionException if no corresponding target resource found
 */
protected Object autowireResource(BeanFactory factory, LookupElement element, @Nullable String requestingBeanName)
        throws NoSuchBeanDefinitionException {

    Object resource;
    Set<String> autowiredBeanNames;
    String name = element.name;

    if (factory instanceof AutowireCapableBeanFactory) {
        AutowireCapableBeanFactory beanFactory = (AutowireCapableBeanFactory) factory;
        DependencyDescriptor descriptor = element.getDependencyDescriptor();
        if (this.fallbackToDefaultTypeMatch && element.isDefaultName && !factory.containsBean(name)) {
            autowiredBeanNames = new LinkedHashSet<>();
            resource = beanFactory.resolveDependency(descriptor, requestingBeanName, autowiredBeanNames, null);
            if (resource == null) {
                throw new NoSuchBeanDefinitionException(element.getLookupType(), "No resolvable resource object");
            }
        }
        else {
            resource = beanFactory.resolveBeanByName(name, descriptor);
            autowiredBeanNames = Collections.singleton(name);
        }
    }
    else {
        resource = factory.getBean(name, element.lookupType);
        autowiredBeanNames = Collections.singleton(name);
    }

    if (factory instanceof ConfigurableBeanFactory) {
        ConfigurableBeanFactory beanFactory = (ConfigurableBeanFactory) factory;
        for (String autowiredBeanName : autowiredBeanNames) {
            if (requestingBeanName != null && beanFactory.containsBean(autowiredBeanName)) {
                beanFactory.registerDependentBean(autowiredBeanName, requestingBeanName);


    }

    return resource;
}
```

总结

@Autowired、@Inject、@Value 和 **@Resource** 注释由 **Spring BeanPostProcessor** 实现处理。 这意味着你不能在自己的 BeanPostProcessor 或 BeanFactoryPostProcessor 类型（如果有的话）中应用这些注释。要使用这些注解，必须使用 XML 或 Spring 注解 @Bean 的方法显式地声明这些 **BeanPostProcessor**。

关于**@Autowired 和 @Resource 的应用场景，官方给出的解释如下：**

> That said, if you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if it is capable of selecting by bean name among type-matching candidates. Instead, use the JSR-250 @Resource annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process. @Autowired has rather different semantics: After selecting candidate beans by type, the specified String qualifier value is considered within those type-selected candidates only (for example, matching an account qualifier against beans marked with the same qualifier label).

> For beans that are themselves defined as a collection, Map, or array type, @Resource is a fine solution, referring to the specific collection or array bean by unique name. That said, as of 4.3, collection, you can match Map, and array types through Spring’s @Autowired type matching algorithm as well, as long as the element type information is preserved in @Bean return type signatures or collection inheritance hierarchies. In this case, you can use qualifier values to select among same-typed collections, as outlined in the previous paragraph.
>
> As of 4.3, @Autowired also considers self references for injection (that is, references back to the bean that is currently injected). Note that self injection is a fallback. Regular dependencies on other components always have precedence. In that sense, self references do not participate in regular candidate selection and are therefore in particular never primary. On the contrary, they always end up as lowest precedence. In practice, you should use self references as a last resort only (for example, for calling other methods on the same instance through the bean’s transactional proxy). Consider factoring out the effected methods to a separate delegate bean in such a scenario. Alternatively, you can use @Resource, which may obtain a proxy back to the current bean by its unique name.
>
> @Autowired applies to fields, constructors, and multi-argument methods, allowing for narrowing through qualifier annotations at the parameter level. In contrast, @Resource is supported only for fields and bean property setter methods with a single argument. As a consequence, you should stick with qualifiers if your injection target is a constructor or a multi-argument method.

简单的说就是：

@Resource 默认按 **byName** 自动注入。 既不指定 name 属性，也不指定 type 属性，则自动按 byName 方式进行查找。如果没有找到符合的 Bean，则回退为一个原始类型进行进行查找，如果找到就注入。 只是指定了 @Resource 注解的 name，则按 name 后的名字去 Bean 元素里查找有与之相等的 name 属性的 bean。 只指定 @Resource 注解的 type 属性，则从上下文中找到类型匹配的唯一 Bean 进行装配，找不到或者找到多个，都会抛出异常。

@Autowired 默认先按 **byType** 进行匹配，如果发现找到多个 Bean，则又按照 byName 方式进行匹配，如果还有多个，则报出异常。

}

19 Spring MVC应用拓展Resource应用于文件下载

21 Resource总结