Dubbo和spring的整合和原理剖析

1、dubbo单独使用

dubbo是可以不跟spring整合单独使用的，我们通过案例看一下dubbo单独使用的情况。

1.1、案例

生产者服务暴露

```
public class ProviderApi {
    public static void main(String[] args) throws IOException {
        UserServiceImpl userService = new UserServiceImpl();

        // 1、应用信息
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("dubbo_provider");

        //2、注册信息
        RegistryConfig registry = new RegistryConfig();
        registry.setAddress("zookeeper://127.0.0.1:2181");

        //3、协议信息
        ProtocolConfig protocolConfig = new ProtocolConfig();
        protocolConfig.setName("dubbo");
        protocolConfig.setPort(20880);
        protocolConfig.setThreads(200);

        //服务发布
        ServiceConfig<UserService> serviceConfig = new ServiceConfig<>();
        serviceConfig.setApplication(applicationConfig);
        serviceConfig.setRegistry(registry);
        serviceConfig.setProtocol(protocolConfig);

        serviceConfig.setInterface(UserService.class);
        serviceConfig.setRef(userService);
//        serviceConfig.setVersion("1.0.0");

        //服务发布
        serviceConfig.export();

        System.in.read();
    }
}
```

消费者引用服务

```
@Test
public void refService() {
    // 1、应用信息
    ApplicationConfig applicationConfig = new ApplicationConfig();
    applicationConfig.setName("dubbo_consumer");

    //2、注册信息
    RegistryConfig registry = new RegistryConfig();
    registry.setAddress("zookeeper://127.0.0.1:2181");

    //引用API
    ReferenceConfig<UserService> referenceConfig = new ReferenceConfig<>();
    referenceConfig.setApplication(applicationConfig);
    referenceConfig.setRegistry(registry);
    referenceConfig.setInterface(UserService.class);

    //服务引用  。这个引用过程非常重，如果想用api方式去引用服务，这个对象需要缓存
    UserService userService = referenceConfig.get();
    System.out.println(userService.queryUser("wuya"));
}
```

1.2、弊端

通过代码案例我可以看到，用dubbo api的方式发布和引用服务还是比较麻烦的需要开发者熟悉api并且要写大量的代码。

2、dubbo跟spring整合

随着spring越来越流行，很多现在的框架都需要跟spring整合，所以我们更有必要精通spring源码了，下面看一下dubbo跟spring整合的案例分析。

2.1、xml的方式跟spring整合

很多老项目现在依然是采用的xml的方式配置dubbo应用，但是xml方式已经不是现在的主流了，但是这种方式我们依然要掌握的，因为纯注解的方式的底层逻辑依然是采用的xml方式的底层逻辑，掌握xml方式的逻辑对我们掌握spring还是很有帮助的。

2.1.1、案例

生产者

applicationProvider.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:dubbo="http://code.alibabatech.com/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://code.alibabatech.com/schema/dubbo
```

```
        http://code.alibabatech.com/schema/dubbo/dubbo.xsd ">

    <dubbo:application name="dubbo_provider"/>
    <!-- 其实就是类似于一个全局变量 -->
    <dubbo:provider timeout="5000"/>
    <dubbo:registry address="zookeeper://127.0.0.1:2181" check="false"/>
    <!-- 如果协议是dubbo，这个就是netty服务端绑定的端口  默认：20880 -->
    <dubbo:protocol name="dubbo" port="29015"/>
    <bean id="userServiceImpl" class="cn.enjoy.service.UserServiceImpl"/>
    <dubbo:service interface="cn.enjoy.service.UserService" ref="userServiceImpl"
timeout="2000">
<!--        <dubbo:method name=""/>-->
    </dubbo:service>
    <!--<dubbo:reference interface=""-->
</beans>
```

生产者启动

```
public class XmlProvider {
    public static void main(String[] args) throws InterruptedException {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationProvider.xml");
        System.out.println("dubbo service started");
        new CountDownLatch(1).await();
    }
}
```

消费者

applicationConsumer.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:dubbo="http://code.alibabatech.com/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://code.alibabatech.com/schema/dubbo
        http://code.alibabatech.com/schema/dubbo/dubbo.xsd ">

    <dubbo:application name="dubbo_consumer"/>
    <!-- 其实就是类似于一个全局变量 -->
    <dubbo:consumer check="false"/>
    <dubbo:registry address="zookeeper://127.0.0.1:2181" check="false"/>

    <dubbo:reference interface="cn.enjoy.service.UserService" check="false"
id="userServiceImpl">
        <dubbo:method name="queryUser" timeout="9000"/>
        <dubbo:method name="doKill" cache="lru"/>
    </dubbo:reference>
</beans>
```

2.1.2、优势

相较于用dubbo api的方式发布和引用服务，基于xml的方式发布和引用服务就相对简单很多，只要通过简单的xml配置就可以完成服务的发布与引用。

2.1.3、源码分析

基于xml的方式跟spring的整合，首先我们必须要知道spring的xml解析流程，只有知道这点才能清楚dubbo的自定义标签是如何解析的。

2.1.3.1、xml解析流程

核心源码在refresh()方法里面，如图：

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        //为容器初始化做准备，重要程度：0
        // Prepare this context for refreshing.
        prepareRefresh();

        /*
          重要程度：5
        * 1、创建BeanFactory对象
        * 2、xml解析
        *   传统标签解析：bean、import等
        *   自定义标签解析 如：<context:component-scan base-package="com.xiangxue.jack"/>
        *   自定义标签解析流程：
        *       a、根据当前解析标签的头信息找到对应的namespaceUri
        *       b、加载spring所有jar中的spring.handlers文件。并建立映射关系
        *       c、根据namespaceUri从映射关系中找到对应的实现了NamespaceHandler接口的类
        *       d、调用类的init方法，init方法是注册了各种自定义标签的解析类
        *       e、根据namespaceUri找到对应的解析类，然后调用paser方法完成标签解析
        *
        * 3、把解析出来的xml标签封装成BeanDefinition对象
        * */
        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
```

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

该方法主要进行xml解析工作，流程如下：

1、创建XmlBeanDefinitionReader对象

```
@Override
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws BeansException, IOException {
    // Create a new XmlBeanDefinitionReader for the given BeanFactory.
    XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
```

2、通过Reader对象加载配置文件

```
 */
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws IOException {
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        for (String configLocation : configLocations) {
            reader.loadBeanDefinitions(configLocation);
        }
    }
}
```

3、根据加载的配置文件把配置文件封装成document对象

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
        throws BeanDefinitionStoreException {

    try {
        //把inputSource 封装成Document文件对象，这是jdk的API
        Document doc = doLoadDocument(inputSource, resource);

        //主要看这个方法，根据解析出来的document对象，拿到里面的标签元素封装成BeanDefinition
        int count = registerBeanDefinitions(doc, resource);
        if (logger.isDebugEnabled()) {
            logger.debug( o: "Loaded " + count + " bean definitions from " + resource);
        }
        return count;
    }
    catch (BeanDefinitionStoreException ex) {
        throw ex;
    }
```

4、创建BeanDefinitionDocumentReader对象，DocumentReader负责对document对象解析

```
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {
    //又来一记委托模式，BeanDefinitionDocumentReader委托这个类进行document的解析
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
    int countBefore = getRegistry().getBeanDefinitionCount();

    //主要看这个方法，createReaderContext(resource) XmlReaderContext上下文，封装了XmlBeanDefinitionReader对象
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

5、默认标签解析流程

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                    //默认标签解析
                    parseDefaultElement(ele, delegate);
                }
                else {
                    //自定义标签解析
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

6、自定义标签解析流程

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                    //默认标签解析
                    parseDefaultElement(ele, delegate);
                }
                else {
                    //自定义标签解析
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

7、最终解析的标签封装成BeanDefinition并缓存到容器中

8、Xml流程图

2.1.3.2、自定义标签解析

1、获取自定义标签的namespace命令空间

例如：http://code.alibabatech.com/schema/dubbo

源码中通过，String namespaceUri = getNamespaceURI(ele); 原始获取到原始的uri，拿dubbo:service标签为例

通过标签头就可以获取到标签对应的uri，xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"

2、根据命令空间获取NamespaceHandler对象

NamespaceUri和NamespaceHandler之间会建立一个映射，spring会从所有的spring jar包中扫描spring.handlers文件，建立映射关系。

核心逻辑如下：

```
NamespaceHandler handler =
this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri);

Map<String, Object> handlerMappings = getHandlerMappings();
Object handlerOrClassName = handlerMappings.get(namespaceUri);
```

3、反射获取NamespaceHandler实例

```
NamespaceHandler namespaceHandler = (NamespaceHandler)
BeanUtils.instantiateClass(handlerClass);
```

4、调用init方法

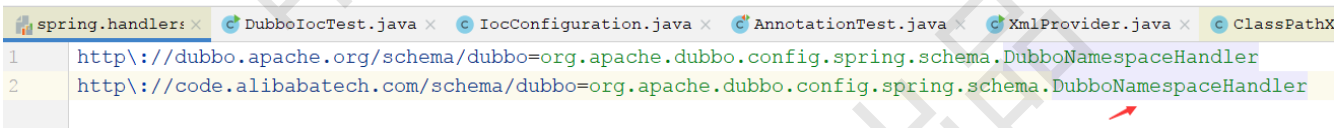
```
namespaceHandler.init();
```

```
//调用处理类的init方法，在init方法中完成标签元素解析类的注册
namespaceHandler.init();
```

5、调用parse方法

```
handler.parse(ele, new ParserContext(this.readerContext, this, containingBd))
```

dubbo的jar中spring.handlers文件



2.1.3.3、dubbo中标签的解析

前面我们已经熟悉了xml解析和自定义标签解析了，那么接下来就来描述清楚dubbo中的标签的解析流程和需要完成的工作。

以dubbo:service标签的解析为例，看看这个标签解析过程中是如何完成服务的发布的。

标签如下：

```
<dubbo:service interface="cn.enjoy.service.UserService" ref="userServiceImpl"
timeout="2000">
</dubbo:service>
```

1、直接从dubbo jar包中的spring.handlers找到namespaceHandler实例。

spring.handlers | DubboIocTest.java | IocConfiguration.java | AnnotationTest.java | XmlProvider.java | ClassPathXmlAppl

```
1  http\://dubbo.apache.org/schema/dubbo=org.apache.dubbo.config.spring.schema.DubboNamespaceHandler
2  http\://code.alibabatech.com/schema/dubbo=org.apache.dubbo.config.spring.schema.DubboNamespaceHandler
```

2、看init方法确定标签和解析类的映射关系

```
@Override
public void init() {
    registerBeanDefinitionParser( elementName: "application", new DubboBeanDefinitionParser(ApplicationConfig.class));
    registerBeanDefinitionParser( elementName: "module", new DubboBeanDefinitionParser(ModuleConfig.class));
    registerBeanDefinitionParser( elementName: "registry", new DubboBeanDefinitionParser(RegistryConfig.class));
    registerBeanDefinitionParser( elementName: "config-center", new DubboBeanDefinitionParser(ConfigCenterBean.class));
    registerBeanDefinitionParser( elementName: "metadata-report", new DubboBeanDefinitionParser(MetadataReportConfig.class));
    registerBeanDefinitionParser( elementName: "monitor", new DubboBeanDefinitionParser(MonitorConfig.class));
    registerBeanDefinitionParser( elementName: "metrics", new DubboBeanDefinitionParser(MetricsConfig.class));
    registerBeanDefinitionParser( elementName: "ssl", new DubboBeanDefinitionParser(SslConfig.class));
    registerBeanDefinitionParser( elementName: "provider", new DubboBeanDefinitionParser(ProviderConfig.class));
    registerBeanDefinitionParser( elementName: "consumer", new DubboBeanDefinitionParser(ConsumerConfig.class));
    registerBeanDefinitionParser( elementName: "protocol", new DubboBeanDefinitionParser(ProtocolConfig.class));
    registerBeanDefinitionParser( elementName: "service", new DubboBeanDefinitionParser(ServiceBean.class));
    registerBeanDefinitionParser( elementName: "reference", new DubboBeanDefinitionParser(ReferenceBean.class));
    registerBeanDefinitionParser( elementName: "annotation", new AnnotationBeanDefinitionParser());
}
```

3、DubboNamespaceHandler的parse方法逻辑

核心看一下registerCommonBeans(registry);方法逻辑

```
@Override
public BeanDefinition parse(Element element, ParserContext parserContext) {
    BeanDefinitionRegistry registry = parserContext.getRegistry();
    registerAnnotationConfigProcessors(registry);
    /**
     * @since 2.7.8
     * issue : https://github.com/apache/dubbo/issues/6275
     */
    registerCommonBeans(registry);
    BeanDefinition beanDefinition = super.parse(element, parserContext);
    setSource(beanDefinition);
    return beanDefinition;
}
```

在该方法中核心把两个类变成了BeanDefinition对象，让其给spring容器实例化对象。

```
registerInfrastructureBean(registry, ReferenceAnnotationBeanPostProcessor.BEAN_NAME,
                ReferenceAnnotationBeanPostProcessor.class);

registerInfrastructureBean(registry, DubboBootstrapApplicationListener.BEAN_NAME,
        DubboBootstrapApplicationListener.class);
```

ReferenceAnnotationBeanPostProcessor完成了@DubboReference属性的依赖注入。

DubboBootstrapApplicationListener完成了ServiceBean，ReferenceBean在spring容器启动完成以后，完成了服务的发布和引用功能。

4、DubboBeanDefinitionParser中的parse方法逻辑

该方法的逻辑其实就是把各配置类，例如ServiceBean，把配置类变成BeanDefinition并且交给spring容器实例化，这样ServiceBean就会被spring实例化。

5、ServiceBean中的afterPropertiesSet方法

前面讲过，ServiceBean会被spring容器实例化，由于实现了InitializingBean接口，所以当实例化ServiceBean的时候就会调到afterPropertiesSet方法。

```
@Override
public void afterPropertiesSet() throws Exception {
    if (StringUtils.isEmpty(getPath())) {
        if (StringUtils.isNotEmpty(getInterface())) {
            setPath(getInterface());
        }
    }
    //register service bean and set bootstrap
    DubboBootstrap.getInstance().service(this);
}
```

其实在该方法中的核心逻辑就是**往ConfigManager中添加ServiceBean的实例**，这样当spring容器完成启动后，dubbo的事件监听类**DubboBootstrapApplicationListener**就会根据ConfigManager中的ServiceBean实例来完成服务的发布。

6、当spring容器完成启动后

当spring容器完成启动后会发布一个**ContextRefreshedEvent**事件，代码如下，在spring的refresh核心方法中有一个finishRefresh();方法会发布该事件。

前面讲过，由于**DubboBootstrapApplicationListener**已经通过**registerCommonBeans(registry);**方法完成了spring的实例化，所以该监听类就会捕获到spring容器启动完成后的**ContextRefreshedEvent**事件。代码如下：

```
@Override
public void onApplicationEvent(ApplicationEvent event) {
    if (isOriginalEventSource(event)) {
        if (event instanceof DubboAnnotationInitedEvent) {
            // This event will be notified at AbstractApplicationContext.registerListeners(),
            // init dubbo config beans before spring singleton beans
            initDubboConfigBeans();
        } else if (event instanceof ApplicationContextEvent) {
            this.onApplicationContextEvent((ApplicationContextEvent) event);
        }
    }
}
```

捕获事件后就会调用**onApplicationContextEvent**方法，最终会调到DubboBootstrap的start方法。

```
public synchronized DubboBootstrap start() {
    // avoid re-entry start method multiple times in same thread
    if (isCurrentlyInStart){
        return this;
```

```
    }

    isCurrentlyInStart = true;
    try {
        if (started.compareAndSet(false, true)) {
            startup.set(false);
            shutdown.set(false);
            awaited.set(false);

            //这里会完成注册中心和元数据中心的初始化
            initialize();

            if (logger.isInfoEnabled()) {
                logger.info(NAME + " is starting...");
            }
            //这里会发布服务，根据ConfigManager中的配置类发布服务
            doStart();

            if (logger.isInfoEnabled()) {
                logger.info(NAME + " has started.");
            }
        } else {
            if (logger.isInfoEnabled()) {
                logger.info(NAME + " is started, export/refer new services.");
            }

            doStart();

            if (logger.isInfoEnabled()) {
                logger.info(NAME + " finish export/refer new services.");
            }
        }
        return this;
    } finally {
        isCurrentlyInStart = false;
    }
}
```

```
private void doStart() {
    // 1. export Dubbo Services
    exportServices();

    // If register consumer instance or has exported services
    if (isRegisterConsumerInstance() || hasExportedServices()) {
        // 2. export MetadataService
        exportMetadataService();
        // 3. Register the local ServiceInstance if required
        registerServiceInstance();
    }

    //这里完成了dubbo服务的引用，也是会从ConfigManager中获取到ReferenceBean实例
    referServices();
```

```
    // wait async export / refer finish if needed
    awaitFinish();

    if (isExportBackground() || isReferBackground()) {
        new Thread(() -> {
            while (!asyncExportFinish || !asyncReferFinish) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    logger.error(NAME + " waiting async export / refer occurred and error.",
e);
                }
            }
            onStarted();
        }).start();
    } else {
        onStarted();
    }
}
```

```
private void exportServices() {
    for (ServiceConfigBase sc : configManager.getServices()) {
        // TODO, compatible with ServiceConfig.export()
        ServiceConfig<?> serviceConfig = (ServiceConfig<?>) sc;
        serviceConfig.setBootstrap(this);
        if (!serviceConfig.isRefreshed()) {
            serviceConfig.refresh();
        }
        if (sc.isExported()) {
            continue;
        }
        if (sc.shouldExportAsync()) {
            ExecutorService executor = executorRepository.getServiceExportExecutor();
            CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
                try {
                    if (!sc.isExported()) {
                        sc.export();
                        exportedServices.add(sc);
                    }
                } catch (Throwable t) {
                    logger.error("export async catch error : " + t.getMessage(), t);
                }
            }, executor);

            asyncExportingFutures.add(future);
        } else {
            if (!sc.isExported()) {
                //最终在这里会调到ServiceBean中的export方法完成服务的发布
                sc.export();
                exportedServices.add(sc);
            }
        }
```

```
    }
}
```

```
//export方法就是服务发布的核心方法，调用到这个方法就可以发布服务
public synchronized void export() {
    if (this.shouldExport() && !this.exported) {
        this.init();

        // check bootstrap state
        if (!bootstrap.isInitialized()) {
            throw new IllegalStateException("DubboBootstrap is not initialized");
        }

        if (!this.isRefreshed()) {
            this.refresh();
        }

        if (!shouldExport()) {
            return;
        }

        if (shouldDelay()) {
            DELAY_EXPORT_EXECUTOR.schedule(this::doExport, getDelay(),
TimeUnit.MILLISECONDS);
        } else {
            doExport();
        }

        if (this.bootstrap.getTakeoverMode() == BootstrapTakeoverMode.AUTO) {
            this.bootstrap.start();
        }
    }
}
```

至此，dubbo的xml方式跟spring整合就分析完成。

2.2、注解方式跟spring整合

注解方式已经是现在的主流了，例如springboot现在都是零xml配置了，只要通过少量的配置就可以完成框架的引用。

2.2.1、案例

生产者

配置类

```
@Configuration
//作用 扫描 @DubboService注解  @DubboReference
@EnableDubbo(scanBasePackages = "cn.enjoy")
@PropertySource("classpath:/dubbo-provider.properties")
public class ProviderConfiguration {
}
```

dubbo-provider.properties配置

```
dubbo.application.name=dubbo_provider
dubbo.registry.address=zookeeper://${zookeeper.address:127.0.0.1}:2181
dubbo.protocol.name=dubbo
dubbo.protocol.port=20880

dubbo.config-center.address=zookeeper://${zookeeper.address:127.0.0.1}:2181
```

服务暴露

```
@DubboService
public class UserServiceImpl implements UserService {
    @Override
    public String queryUser(String s) {

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(s);
        System.out.println("==========provider==========" + s);
        return "OK--" + s;
    }

    @Override
    public void doKill(String s) {
        System.out.println("==========provider==========" + s);
    }
}
```

启动

```
public class AnnotationProvider {
    public static void main(String[] args) throws InterruptedException {
        ZKTools.generateDubboProperties();
        new AnnotationConfigApplicationContext(ProviderConfiguration.class);
        System.out.println("dubbo service started.");
        new CountDownLatch(1).await();
    }
}
```

2.2.2、优势

可以看到，只需要在需要暴露的服务上面加上一个@DubboService注解就可以完成服务的暴露了，配置量非常少，但是这种方式也有一定的侵入性。

2.2.3、源码分析

2.2.3.1、@EnableDubbo注解

@EnableDubbo注解可以理解为引入dubbo功能，其实在这里它起到的作用就是两个，

1、扫描类上面的@DubboService注解

2、扫描类中属性或者方法上面的@DubboReference注解

那为什么@EnableDubbo会被spring扫描到呢?

要回答这个问题，首先我们得看看@EnableDubbo的结构。

```
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@EnableDubboConfig
@DubboComponentScan
public @interface EnableDubbo {
```

```
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(DubboComponentScanRegistrar.class)
public @interface DubboComponentScan {
```

可以看到，在@DubboComponentScan注解中有一个@Import注解，实际上spring能扫描到的就是这个@Import注解，通过扫描到@Import注解从而把import进来的类变成BeanDefinition交给spring实例化。

具体spring如何扫描的，请看下面步骤：

1、通过spring上下文对象的实例化把ConfigurationClassPostProcessor变成BeanDefinition

```
public AnnotationConfigApplicationContext() {
    this.reader = new AnnotatedBeanDefinitionReader( registry: this);
    this.scanner = new ClassPathBeanDefinitionScanner( registry: this);
}
```

```
 */
public AnnotatedBeanDefinitionReader(BeanDefinitionRegistry registry, Environment environment) {
    Assert.notNull(registry, message: "BeanDefinitionRegistry must not be null");
    Assert.notNull(environment, message: "Environment must not be null");
    this.registry = registry;
    this.conditionEvaluator = new ConditionEvaluator(registry, environment, resourceLoader: null);
    AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
}
 */
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<>( initialCapacity: 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));
    }
```

2、ConfigurationClassPostProcessor对@Import的扫描

由于ConfigurationClassPostProcessor类是一个BeanDefinitionRegistryPostProcessor类型的，所以在spring容器中它会优先被实例化，实例化的地方在refresh核心方法的：

```
        /*
         * BeanDefinitionRegistryPostProcessor
         * BeanFactoryPostProcessor
         * 完成对这两个接口的调用
         * */
        // Invoke factory processors registered as beans in the context.
        invokeBeanFactoryPostProcessors(beanFactory);

        /*
```

所以当ConfigurationClassPostProcessor实例化的时候就调用到postProcessBeanDefinitionRegistry方法，方法逻辑如下：

```
@Override
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
   int registryId = System.identityHashCode(registry);
   if (this.registriesPostProcessed.contains(registryId)) {
      throw new IllegalStateException(
            "postProcessBeanDefinitionRegistry already called on this post-processor against
" + registry);
   }
   if (this.factoriesPostProcessed.contains(registryId)) {
      throw new IllegalStateException(
            "postProcessBeanFactory already called on this post-processor against " +
```

```
registry);
   }
   this.registriesPostProcessed.add(registryId);

   //核心逻辑，重点看，重要程度5
   processConfigBeanDefinitions(registry);
}
```

```
public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {
   List<BeanDefinitionHolder> configCandidates = new ArrayList<>();
   //获取所有的beanNames
   String[] candidateNames = registry.getBeanDefinitionNames();

   for (String beanName : candidateNames) {
      BeanDefinition beanDef = registry.getBeanDefinition(beanName);
      //如果有该标识就不再处理
      if (beanDef.getAttribute(ConfigurationClassUtils.CONFIGURATION_CLASS_ATTRIBUTE) !=
null) {
         if (logger.isDebugEnabled()) {
            logger.debug("Bean definition has already been processed as a configuration
class: " + beanDef);
         }
      }
      //判断是否是候选的需要处理的BeanDefinition，如果是则放入容器configCandidates
      else if (ConfigurationClassUtils.checkConfigurationClassCandidate(beanDef,
this.metadataReaderFactory)) {
         configCandidates.add(new BeanDefinitionHolder(beanDef, beanName));
      }
   }

   // Return immediately if no @Configuration classes were found
   //如果容器为空，则直接返回
   if (configCandidates.isEmpty()) {
      return;
   }

   // Sort by previously determined @Order value, if applicable
   //对需要处理的所有beanDefinition排序
   configCandidates.sort((bd1, bd2) -> {
      int i1 = ConfigurationClassUtils.getOrder(bd1.getBeanDefinition());
      int i2 = ConfigurationClassUtils.getOrder(bd2.getBeanDefinition());
      return Integer.compare(i1, i2);
   });

   // Detect any custom bean name generation strategy supplied through the enclosing
application context
   SingletonBeanRegistry sbr = null;
   if (registry instanceof SingletonBeanRegistry) {
      sbr = (SingletonBeanRegistry) registry;
      if (!this.localBeanNameGeneratorSet) {
         BeanNameGenerator generator = (BeanNameGenerator) sbr.getSingleton(
               AnnotationConfigUtils.CONFIGURATION_BEAN_NAME_GENERATOR);
         if (generator != null) {
```

```
            this.componentScanBeanNameGenerator = generator;
            this.importBeanNameGenerator = generator;
         }
      }
   }

   if (this.environment == null) {
      this.environment = new StandardEnvironment();
   }

   //候选BeanDefinition的解析器
   // Parse each @Configuration class
   ConfigurationClassParser parser = new ConfigurationClassParser(
         this.metadataReaderFactory, this.problemReporter, this.environment,
         this.resourceLoader, this.componentScanBeanNameGenerator, registry);

   Set<BeanDefinitionHolder> candidates = new LinkedHashSet<>(configCandidates);
   Set<ConfigurationClass> alreadyParsed = new HashSet<>(configCandidates.size());
   do {
      //解析核心流程，重点看，重要程度5
      //其实就是把类上面的特殊注解解析出来最终封装成beanDefinition
      parser.parse(candidates);
      parser.validate();

      Set<ConfigurationClass> configClasses = new LinkedHashSet<>
(parser.getConfigurationClasses());
      configClasses.removeAll(alreadyParsed);

      // Read the model and create bean definitions based on its content
      if (this.reader == null) {
         this.reader = new ConfigurationClassBeanDefinitionReader(
               registry, this.sourceExtractor, this.resourceLoader, this.environment,
               this.importBeanNameGenerator, parser.getImportRegistry());
      }
      //@Bean @Import 内部类 @ImportedResource ImportBeanDefinitionRegistrar具体处理逻辑
      this.reader.loadBeanDefinitions(configClasses);
      //已经解析完成了的类
      alreadyParsed.addAll(configClasses);

      candidates.clear();
      //比较差异又走一遍解析流程
      if (registry.getBeanDefinitionCount() > candidateNames.length) {
         String[] newCandidateNames = registry.getBeanDefinitionNames();
         Set<String> oldCandidateNames = new HashSet<>(Arrays.asList(candidateNames));
         Set<String> alreadyParsedClasses = new HashSet<>();
         for (ConfigurationClass configurationClass : alreadyParsed) {
            alreadyParsedClasses.add(configurationClass.getMetadata().getClassName());
         }
         for (String candidateName : newCandidateNames) {
            if (!oldCandidateNames.contains(candidateName)) {
               BeanDefinition bd = registry.getBeanDefinition(candidateName);
               if (ConfigurationClassUtils.checkConfigurationClassCandidate(bd,
this.metadataReaderFactory) &&
```

```
                    !alreadyParsedClasses.contains(bd.getBeanClassName())) {
                candidates.add(new BeanDefinitionHolder(bd, candidateName));
            }
         }
      }
      candidateNames = newCandidateNames;
   }
}
while (!candidates.isEmpty());

// Register the ImportRegistry as a bean in order to support ImportAware @Configuration
classes
if (sbr != null && !sbr.containsSingleton(IMPORT_REGISTRY_BEAN_NAME)) {
   sbr.registerSingleton(IMPORT_REGISTRY_BEAN_NAME, parser.getImportRegistry());
}

if (this.metadataReaderFactory instanceof CachingMetadataReaderFactory) {
   // Clear cache in externally provided MetadataReaderFactory; this is a no-op
   // for a shared cache since it'll be cleared by the ApplicationContext.
   ((CachingMetadataReaderFactory) this.metadataReaderFactory).clearCache();
}
}
```

就是在上面方面里面的this.reader.loadBeanDefinitions(configClasses);这行代码对所有BeanDefinition中对应类上如果有@Import注解进行了解析处理，这样spring就能够扫描的@EnableDubbo注解了。

2.2.3.2、DubboComponentScanRegistrar

DubboComponentScanRegistrar类是通过@EnableDubbo上面的@DubboComponentScan注解import进来的，它是一个ImportBeanDefinitionRegistrar类型的，所有当被引入进来以后就会被spring调用到它的registerBeanDefinitions方法，代码如下：

```
@Override
public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
BeanDefinitionRegistry registry) {

    // @since 2.7.6 Register the common beans
    registerCommonBeans(registry);

    Set<String> packagesToScan = getPackagesToScan(importingClassMetadata);

    registerServiceAnnotationPostProcessor(packagesToScan, registry);
}
```

至于为什么spring能调到这个方法，起调用逻辑还是在this.reader.loadBeanDefinitions(configClasses);这行代码里面，具体调用代码：

```
private void loadBeanDefinitionsFromRegistrars(Map<ImportBeanDefinitionRegistrar,
AnnotationMetadata> registrars) {
   registrars.forEach((registrar, metadata) ->
         registrar.registerBeanDefinitions(metadata, this.registry,
this.importBeanNameGenerator));
}
```

我们在来看看registerBeanDefinitions方法做了些什么， registerCommonBeans(registry);方法前面我们分析过，注册了两个比较重要的类

1、ReferenceAnnotationBeanPostProcessor

2、DubboBootstrapApplicationListener

registerServiceAnnotationPostProcessor(packagesToScan, registry);这行代码里面又注册了一个比较重要的类，ServiceAnnotationPostProcessor

所以该方法其实核心就是注册了三个比较重要的类给了spring容器：

1、ReferenceAnnotationBeanPostProcessor

2、DubboBootstrapApplicationListener

3、ServiceAnnotationPostProcessor

DubboBootstrapApplicationListener这个类前面我们分析过，它就是在spring容器启动完成后通过spring发布一个spring容器启动完成的事件，然后该类捕获到事件，通过捕获事件来完成服务的发布和引用的，这里就不再赘述了。现在主要分析ServiceAnnotationPostProcessor和ReferenceAnnotationBeanPostProcessor

2.2.3.3、ServiceAnnotationPostProcessor

首先这个ServiceAnnotationPostProcessor类是干什么的呢?

这个类的作用就是用来扫描类上面的@DubboService注解的，就只有这个功能。

该类的类型是BeanDefinitionRegistryPostProcessor类型的。所以它会被spring调用到postProcessBeanDefinitionRegistry方法，调用逻辑如下：

```
@Override
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws
BeansException {
    this.registry = registry;

    Set<String> resolvedPackagesToScan = resolvePackagesToScan(packagesToScan);

    if (!CollectionUtils.isEmpty(resolvedPackagesToScan)) {
        //这里是扫描的核心逻辑
        scanServiceBeans(resolvedPackagesToScan, registry);
    } else {
        if (logger.isWarnEnabled()) {
            logger.warn("packagesToScan is empty , ServiceBean registry will be ignored!");
        }
    }
```

```
}
```

```
private void scanServiceBeans(Set<String> packagesToScan, BeanDefinitionRegistry registry) {

    //定义一个扫描器，这个dubbo扫描器其实就是继承了spring的ClassPathBeanDefinitionScanner扫描器
    DubboClassPathBeanDefinitionScanner scanner =
            new DubboClassPathBeanDefinitionScanner(registry, environment, resourceLoader);

    BeanNameGenerator beanNameGenerator = resolveBeanNameGenerator(registry);
    scanner.setBeanNameGenerator(beanNameGenerator);

    //对扫描器添加需要扫描的注解类型，这里的注解类型有三种，阿帕奇的@Service，@DubboService，阿里巴巴的
@Service
    for (Class<? extends Annotation> annotationType : serviceAnnotationTypes) {
        scanner.addIncludeFilter(new AnnotationTypeFilter(annotationType));
    }

    ScanExcludeFilter scanExcludeFilter = new ScanExcludeFilter();
    scanner.addExcludeFilter(scanExcludeFilter);

    for (String packageToScan : packagesToScan) {

        // avoid duplicated scans
        if (servicePackagesHolder.isPackageScanned(packageToScan)) {
            if (logger.isInfoEnabled()) {
                logger.info("Ignore package who has already bean scanned: " +
packageToScan);
            }
            continue;
        }

        //这里就是核心的扫描代码
        //扫描的核心流程，其实就是递归找文件的过程，如果找的是文件夹则递归找，如果是文件则根据完整限定名反射
出反射对象，根据反射对象判断类上是否有DubboService注解，如果又则把该类变成BeanDefinition
        // Registers @Service Bean first
        scanner.scan(packageToScan);

        // Finds all BeanDefinitionHolders of @Service whether @ComponentScan scans or not.
        Set<BeanDefinitionHolder> beanDefinitionHolders =
            //这行代码就是根据前面的扫描找到的BeanDefinition集合，获取这个集合
                findServiceBeanDefinitionHolders(scanner, packageToScan, registry,
beanNameGenerator);

        if (!CollectionUtils.isEmpty(beanDefinitionHolders)) {
            if (logger.isInfoEnabled()) {
                List<String> serviceClasses = new ArrayList<>(beanDefinitionHolders.size());
                for (BeanDefinitionHolder beanDefinitionHolder : beanDefinitionHolders) {

serviceClasses.add(beanDefinitionHolder.getBeanDefinition().getBeanClassName());
                }
                logger.info("Found " + beanDefinitionHolders.size() + " classes annotated by
Dubbo @Service under package [" + packageToScan + "]: " + serviceClasses);
```

```
            }

            for (BeanDefinitionHolder beanDefinitionHolder : beanDefinitionHolders) {
                //这行代码很重要，因为要完成服务暴露那么就必须调用到ServiceBean中的export方法
                //这行代码就是根据有注解的类的BeanDefinition来创建根该类对应的ServiceBean的
BeanDefinition对象
                processScannedBeanDefinition(beanDefinitionHolder, registry, scanner);

servicePackagesHolder.addScannedClass(beanDefinitionHolder.getBeanDefinition().getBeanClassN
ame());
            }
        } else {
            if (logger.isWarnEnabled()) {
                logger.warn("No class annotated by Dubbo @Service was found under package ["
                        + packageToScan + "], ignore re-scanned classes: " +
scanExcludeFilter.getExcludedCount());
            }
        }

        servicePackagesHolder.addScannedPackage(packageToScan);
    }
}
```

```
private void processScannedBeanDefinition(BeanDefinitionHolder beanDefinitionHolder,
BeanDefinitionRegistry registry,
                                          DubboClassPathBeanDefinitionScanner scanner) {

    Class<?> beanClass = resolveClass(beanDefinitionHolder);

    Annotation service = findServiceAnnotation(beanClass);

    // The attributes of @Service annotation
    Map<String, Object> serviceAnnotationAttributes = AnnotationUtils.getAttributes(service,
true);

    String serviceInterface = resolveInterfaceName(serviceAnnotationAttributes, beanClass);

    String annotatedServiceBeanName = beanDefinitionHolder.getBeanName();

    // ServiceBean Bean name
    String beanName = generateServiceBeanName(serviceAnnotationAttributes,
serviceInterface);

    AbstractBeanDefinition serviceBeanDefinition =
        //这里就是创建ServiceBean的BeanDefinition
            buildServiceBeanDefinition(serviceAnnotationAttributes, serviceInterface,
annotatedServiceBeanName);

    registerServiceBeanDefinition(beanName, serviceBeanDefinition, serviceInterface);

}
```

OK,从上面的分析，我们知道，扫描到有@DubboService注解的类以后，实际上会创建跟该类对应的ServiceBean的实例，也就是有一个@DubboService注解的类就会对应一个ServiceBean的实例在spring容器中。那么又回到了上面xml分析的流程了，ServiceBean实例化走到afterPropertiesSet方法，然后spring容器启动完成走到dubbo的监听器完成服务发布了。

2.2.3.4、ReferenceAnnotationBeanPostProcessor

ReferenceAnnotationBeanPostProcessor的作用就是完成@DubboReference属性或者方法的依赖注入，其依赖注入的流程也是完全依赖spring的，所以我们必须要掌握spring的依赖注入流程，其实spring的依赖注入核心就两点；

2.2.3.4.1、spring的依赖注入

1、注解的收集

2、实例的注入

以@Autowired注解的属性来分析spring的依赖注入，例如：

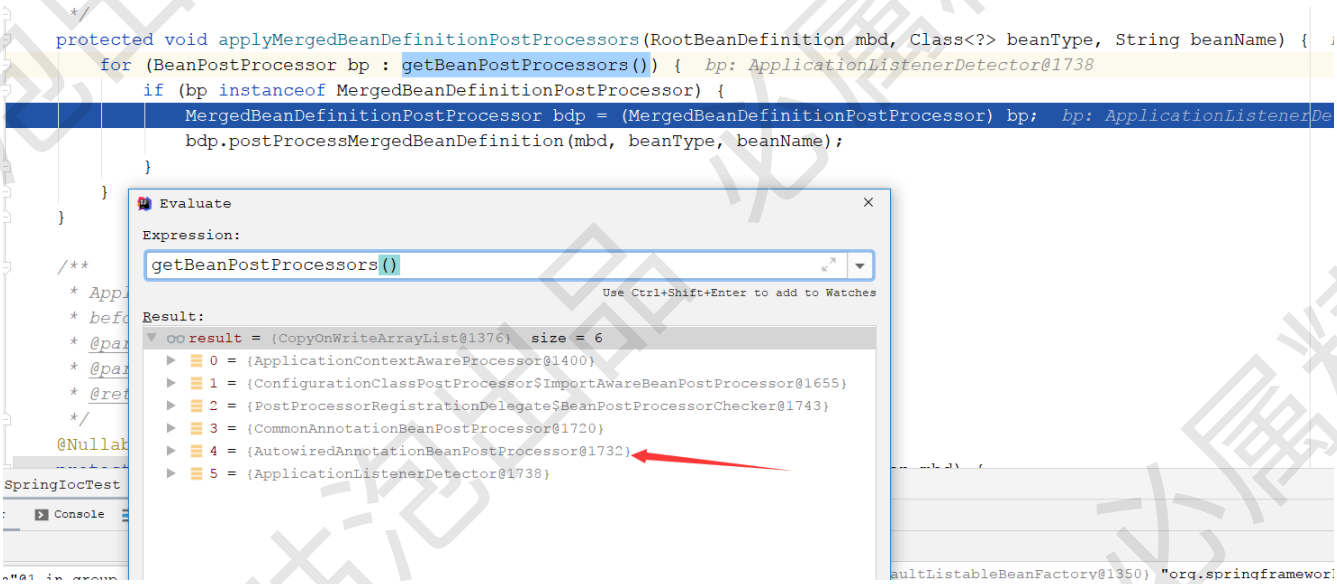
```
@Component
public class SpringTest {

    @Autowired
    private UserService userService;

    private UserService userService1;

    private UserService userService2;

    private UserService userService3;
    private UserService userService4;


    public UserService getUserService() {
        return userService;
    }

    @PostConstruct
    public void init() {
        System.out.println(userService);
    }
}
```

1、@Autowired注解收集

注解收集的核心流程

```
        // Allow post-processors to modify the merged bean definition.
        synchronized (mbd.postProcessingLock) {
            if (!mbd.postProcessed) {
                try {
                    //CommonAnnotationBeanPostProcessor  支持了@PostConstruct，@PreDestroy,@Resou
                    //AutowiredAnnotationBeanPostProcessor 支持 @Autowired,@Value注解
                    //BeanPostProcessor接口的典型运用，这里要理解这个接口
                    //对类中注解的装配过程
                    //重要程度5，必须看
                    applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                            "Post-processing of merged bean definition failed", ex);
                }
                mbd.postProcessed = true;
            }
        }
```

对应的接口类型MergedBeanDefinitionPostProcessor，这个类型的BeanPostProcessor的埋点就是用来处理注解收集这个功能点的，只有关系该功能点的类才会对该接口的方法进行实现，其他不关心的可以不管MergedBeanDefinitionPostProcessor该接口的方法实现，也就是方法可以直接是一个空方法。



AutowiredAnnotationBeanPostProceessor类就是用来对@Autowired注解进行支持的。该类的注册是在spring上下文对象的构造函数的完成注册的。

```
@Override
public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?>
beanType, String beanName) {
   //注解收集核心逻辑
   InjectionMetadata metadata = findAutowiringMetadata(beanName, beanType, null);
   metadata.checkConfigMembers(beanDefinition);
}
```

```
private InjectionMetadata findAutowiringMetadata(String beanName, Class<?> clazz, @Nullable
PropertyValues pvs) {
   // Fall back to class name as cache key, for backwards compatibility with custom callers.
```

```
   String cacheKey = (StringUtils.hasLength(beanName) ? beanName : clazz.getName());
   // Quick check on the concurrent map first, with minimal locking.
    //先从缓存拿结果
   InjectionMetadata metadata = this.injectionMetadataCache.get(cacheKey);
   if (InjectionMetadata.needsRefresh(metadata, clazz)) {
      synchronized (this.injectionMetadataCache) {
         metadata = this.injectionMetadataCache.get(cacheKey);
         if (InjectionMetadata.needsRefresh(metadata, clazz)) {
            if (metadata != null) {
               metadata.clear(pvs);
            }
            //主要看这个方法
             //收集的核心逻辑
            metadata = buildAutowiringMetadata(clazz);
            this.injectionMetadataCache.put(cacheKey, metadata);
         }
      }
   }
   return metadata;
}
```

```
//其实这里就是拿到类上所有的field，然后判断field上是否有@Autowired注解，如果有则封装成对象
//寻找field上面的@Autowired注解并封装成对象
ReflectionUtils.doWithLocalFields(targetClass, field -> {
   MergedAnnotation<?> ann = findAutowiredAnnotation(field);
   if (ann != null) {
      if (Modifier.isStatic(field.getModifiers())) {
         if (logger.isInfoEnabled()) {
            logger.info("Autowired annotation is not supported on static fields: " + field);
         }
         return;
      }
      boolean required = determineRequiredStatus(ann);
      currElements.add(new AutowiredFieldElement(field, required));
   }
});
```

注解的收集就完成了

2、@Autowired的依赖注入

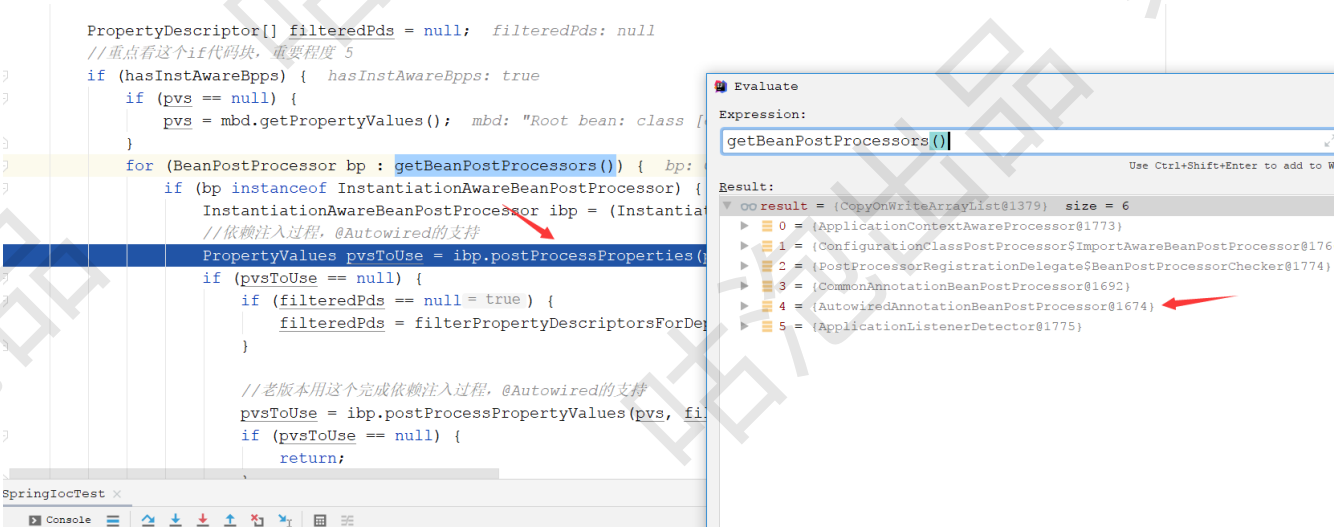
依赖注入的核心方法：

```
//ioc di，依赖注入的核心方法，该方法必须看，重要程度：5
populateBean(beanName, mbd, instanceWrapper);
```

```
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper
bw) {
   if (bw == null) {
      if (mbd.hasPropertyValues()) {
```

```
            throw new BeanCreationException(
                  mbd.getResourceDescription(), beanName, "Cannot apply property values to null
instance");
         }
         else {
            // Skip property population phase for null instance.
            return;
         }
      }

      // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
      // state of the bean before properties are set. This can be used, for example,
      // to support styles of field injection.
      //这里很有意思，写接口可以让所有类都不能依赖注入
      if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
         for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
               InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor)
bp;
               if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                  return;
               }
            }
         }
      }

      PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

      int resolvedAutowireMode = mbd.getResolvedAutowireMode();
      if (resolvedAutowireMode == AUTOWIRE_BY_NAME || resolvedAutowireMode == AUTOWIRE_BY_TYPE)
{
         MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
         // Add property values based on autowire by name if applicable.
         if (resolvedAutowireMode == AUTOWIRE_BY_NAME) {
            autowireByName(beanName, mbd, bw, newPvs);
         }
         // Add property values based on autowire by type if applicable.
         if (resolvedAutowireMode == AUTOWIRE_BY_TYPE) {
            autowireByType(beanName, mbd, bw, newPvs);
         }
         pvs = newPvs;
      }

      boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
      boolean needsDepCheck = (mbd.getDependencyCheck() !=
AbstractBeanDefinition.DEPENDENCY_CHECK_NONE);

      PropertyDescriptor[] filteredPds = null;
      //重点看这个if代码块，重要程度 5
      if (hasInstAwareBpps) {
         if (pvs == null) {
            pvs = mbd.getPropertyValues();
         }
```

```
      for (BeanPostProcessor bp : getBeanPostProcessors()) {
         if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor)
bp;
            //依赖注入过程，@Autowired的支持。。这里完成了依赖注入
            PropertyValues pvsToUse = ibp.postProcessProperties(pvs,
bw.getWrappedInstance(), beanName);
            if (pvsToUse == null) {
               if (filteredPds == null) {
                  filteredPds = filterPropertyDescriptorsForDependencyCheck(bw,
mbd.allowCaching);
               }

               //老版本用这个完成依赖注入过程，@Autowired的支持
               pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(), beanName);
               if (pvsToUse == null) {
                  return;
               }
            }
            pvs = pvsToUse;
         }
      }
   }
   if (needsDepCheck) {
      if (filteredPds == null) {
         filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
      }
      checkDependencies(beanName, mbd, filteredPds, pvs);
   }

   //这个方法很鸡肋了，建议不看，是老版本用<property name="username" value="Jack"/>
   //标签做依赖注入的代码实现，复杂且无用
   if (pvs != null) {
      applyPropertyValues(beanName, mbd, bw, pvs);
   }
}
```



这里又会调用到AutowiredAnnotationBeanPostProceessor类的方法进行依赖注入。

```
public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String
beanName) {
   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;
}
```

通过前面的收集结果，我们知道了哪些属性获取方法有注解，那么在这里我们只要根据有注解的属性进行依赖注入就可以了，这里就不再赘述了。

2.2.3.4.2、dubbo的依赖注入

其实dubbo的依赖注入流程跟spring是一模一样的，也是借助BeanPostProcessor类型的接口来实现，只是这个类是**ReferenceAnnotationBeanPostProcessor**

OK，我们来一个示例：

```
public class Ioc {
    @DubboReference
    private UserService userService;

    @PostConstruct
    public void init() {
        System.out.println(userService);
    }
}
```

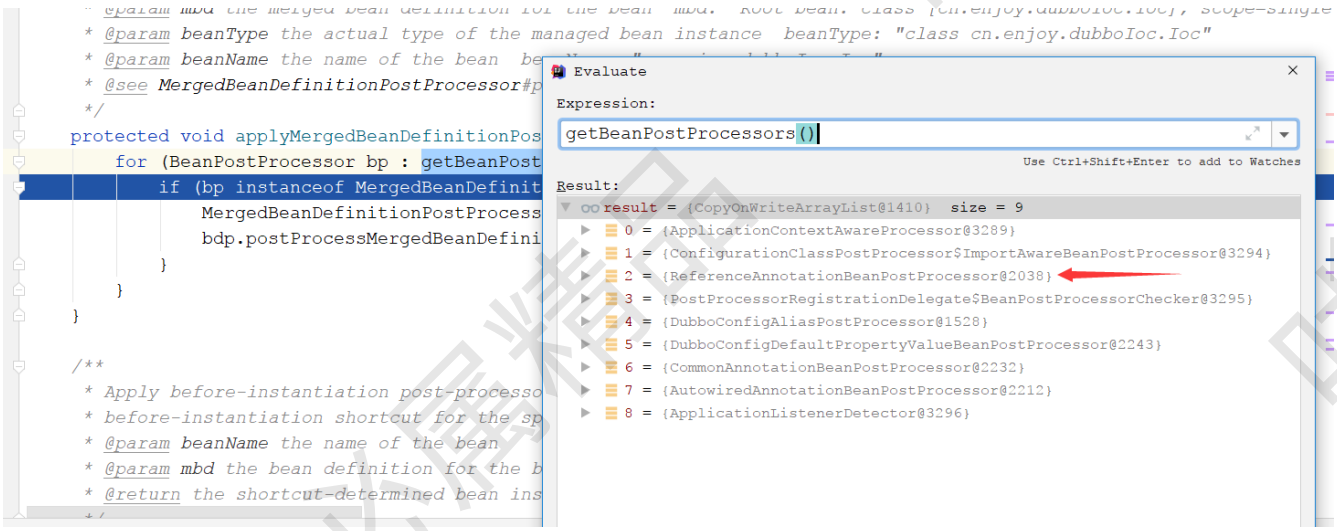
也是两个步骤，只是这两个步骤是由ReferenceAnnotationBeanPostProcessor它来完成的。

1、@DubboReference注解收集

2、实例的依赖注入

1、@DubboReference注解收集

收集的逻辑跟spring是一模一样的，如图：



会走到ReferenceAnnotationBeanPostProcessor类中进行注解收集；

```
@Override
public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?>
beanType, String beanName) {
    if (beanType != null) {
        if (isReferenceBean(beanDefinition)) {
            //mark property value as optional
            List<PropertyValue> propertyValues =
beanDefinition.getPropertyValues().getPropertyValueList();
            for (PropertyValue propertyValue : propertyValues) {
                propertyValue.setOptional(true);
            }
        } else if (isAnnotatedReferenceBean(beanDefinition)) {
            // extract beanClass from java-config bean method generic return type:
ReferenceBean<DemoService>
            //Class beanClass = getBeanFactory().getType(beanName);
        } else {
            //收集有@DubboReference注解的属性或者方法并封装成对象
            AnnotatedInjectionMetadata metadata = findInjectionMetadata(beanName, beanType,
null);
            metadata.checkConfigMembers(beanDefinition);
            try {
                //这里是对每一个有@DubboReference注解的属性或者方法创建一个ReferenceBean的
BeanDefinition对象，因为要完成服务的引用，必须要有@ReferenceBean的实例
                prepareInjection(metadata);
            } catch (Exception e) {
                throw new IllegalStateException("Prepare dubbo reference injection element
failed", e);
            }
        }
    }
}
```

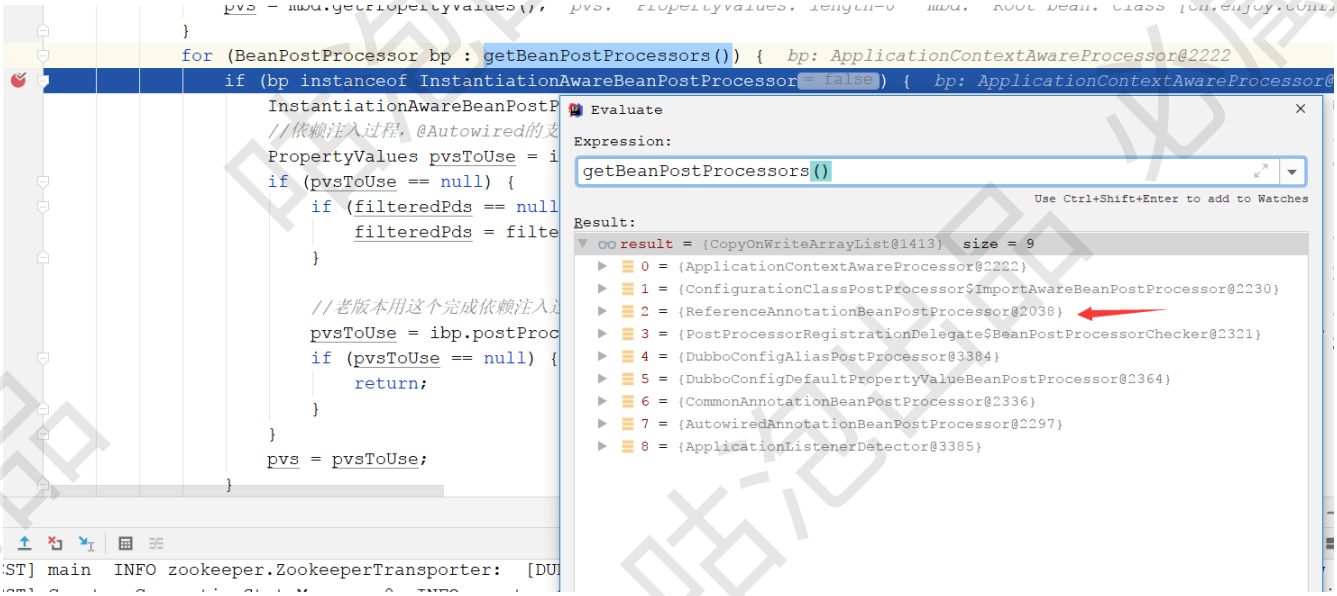
prepareInjection(metadata);的核心代码贴一下：

```
RootBeanDefinition beanDefinition = new RootBeanDefinition();
//这里就会创建ReferenceBean对象
beanDefinition.setBeanClassName(ReferenceBean.class.getName());
beanDefinition.getPropertyValues().add(ReferenceAttributes.ID, referenceBeanName);

// set attribute instead of property values
beanDefinition.setAttribute(Constants.REFERENCE_PROPS, attributes);
beanDefinition.setAttribute(ReferenceAttributes.INTERFACE_CLASS, interfaceClass);
beanDefinition.setAttribute(ReferenceAttributes.INTERFACE_NAME, interfaceName);

// create decorated definition for reference bean, Avoid being instantiated when getting the
beanType of ReferenceBean
// see org.springframework.beans.factory.support.AbstractBeanFactory#getTypeForFactoryBean()
GenericBeanDefinition targetDefinition = new GenericBeanDefinition();
targetDefinition.setBeanClass(interfaceClass);
String id = (String) beanDefinition.getPropertyValues().get(ReferenceAttributes.ID);
beanDefinition.setDecoratedDefinition(new BeanDefinitionHolder(targetDefinition,
id+"_decorated"));

// signal object type since Spring 5.2
beanDefinition.setAttribute(Constants.OBJECT_TYPE_ATTRIBUTE, interfaceClass);

beanDefinitionRegistry.registerBeanDefinition(referenceBeanName, beanDefinition);
```

总结一下，收集注解会收集@DubboReference注解的属性或方法，然后还会创建跟@DubboReference对应的ReferenceBean对象，因为要完成代理的生成和服务的引用。

2、实例的依赖注入

@DubboReference属性的依赖注入，流程跟spring也是一样的，处理类是ReferenceAnnotationBeanPostProcessor



会走到ReferenceAnnotationBeanPostProcessor类中的postProcessPropertyValues方法完成属性依赖注入，

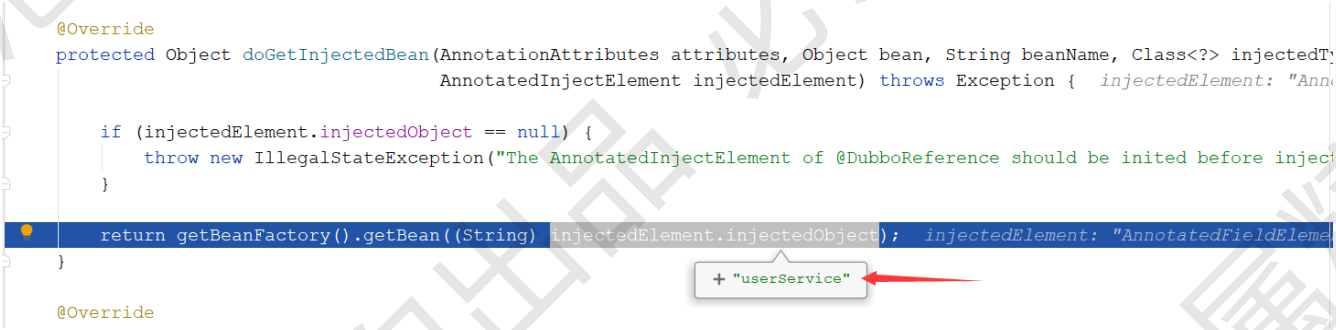
```
public PropertyValues postProcessPropertyValues(
        PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws
BeansException {

    try {
        //这里走缓存拿手机的结果
        AnnotatedInjectionMetadata metadata = findInjectionMetadata(beanName,
bean.getClass(), pvs);
        //再次确定有没有生成Referencebean的BeanDefinition
        prepareInjection(metadata);
        //依赖注入
        metadata.inject(bean, beanName, pvs);
    } catch (BeansException ex) {
        throw ex;
    } catch (Throwable ex) {
        throw new BeanCreationException(beanName, "Injection of @" +
getAnnotationType().getSimpleName()
                + " dependencies is failed", ex);
    }
    return pvs;
}
```

```
protected void inject(Object bean, String beanName, PropertyValues pvs) throws Throwable {

    //获取代理对象的核心方法
    Object injectedObject = getInjectedObject(attributes, bean, beanName, getInjectedType(),
this);

    //属性的方式注入
    if (member instanceof Field) {
        Field field = (Field) member;
        ReflectionUtils.makeAccessible(field);
        field.set(bean, injectedObject);
    } else if (member instanceof Method) {
        Method method = (Method) member;
        ReflectionUtils.makeAccessible(method);
        method.invoke(bean, injectedObject);
    }
}
```

```
    protected Object getInjectedObject(AnnotationAttributes attributes, Object bean, String
beanName, Class<?> injectedType,
                                       AnnotatedInjectElement injectedElement) throws
Exception {

//        String cacheKey = buildInjectedObjectCacheKey(attributes, bean, beanName,
injectedType, injectedElement);
//
//        Object injectedObject = injectedObjectsCache.get(cacheKey);
//
//        if (injectedObject == null) {
//            injectedObject = doGetInjectedBean(attributes, bean, beanName, injectedType,
```

```
injectedElement);
//            // Customized inject-object if necessary
//            injectedObjectsCache.put(cacheKey, injectedObject);
//        }
//        return injectedObject;

        return doGetInjectedBean(attributes, bean, beanName, injectedType, injectedElement);
    }
```

```
@Override
protected Object doGetInjectedBean(AnnotationAttributes attributes, Object bean, String
beanName, Class<?> injectedType,
                                   AnnotatedInjectElement injectedElement) throws Exception
{

    if (injectedElement.injectedObject == null) {
        throw new IllegalStateException("The AnnotatedInjectElement of @DubboReference
should be inited before injection");
    }

    //getBean获取实例，id是 UserService
    return getBeanFactory().getBean((String) injectedElement.injectedObject);
}
```



前面创建了ReferenceBean对象，其中有一个ReferenceBean对象的id就是userService.所有这里getBean会获取到ReferenceBean的实例或者其FactoryBean的getObject返回的实例，其实ReferenceBean是有实现FactoryBean接口的，所有这里会返回getObject返回的实例，我们看看ReferenceBean。

```
public class ReferenceBean<T> implements FactoryBean<T>,
        ApplicationContextAware, BeanClassLoaderAware, BeanNameAware, InitializingBean,
DisposableBean

            @Override
    public T getObject() {
        if (lazyProxy == null) {
            createLazyProxy();
        }
        return (T) lazyProxy;
    }
```

```
private void createLazyProxy() {

    //set proxy interfaces
    //see also:
org.apache.dubbo.rpc.proxy.AbstractProxyFactory.getProxy(org.apache.dubbo.rpc.Invoker<T>,
boolean)
    //很明显这里会用spring的代理工厂生成代理对象
    ProxyFactory proxyFactory = new ProxyFactory();
    //定义哦了TargetSource类型实例，spring中会有该类调用其getTarget方法拿到目标对象，其实这里就会生成
Dubbo的代理
    proxyFactory.setTargetSource(new DubboReferenceLazyInitTargetSource());
    proxyFactory.addInterface(interfaceClass);
    Class<?>[] internalInterfaces = AbstractProxyFactory.getInternalInterfaces();
    for (Class<?> anInterface : internalInterfaces) {
        proxyFactory.addInterface(anInterface);
    }
    if (!StringUtils.isEquals(interfaceClass.getName(), interfaceName)) {
        //add service interface
        try {
            Class<?> serviceInterface = ClassUtils.forName(interfaceName, beanClassLoader);
            proxyFactory.addInterface(serviceInterface);
        } catch (ClassNotFoundException e) {
            // generic call maybe without service interface class locally
        }
    }

    //返回spring的代理
    this.lazyProxy = proxyFactory.getProxy(this.beanClassLoader);
}
```

```
private class DubboReferenceLazyInitTargetSource extends AbstractLazyCreationTargetSource {

    @Override
    protected Object createObject() throws Exception {
        return getCallProxy();
    }

    @Override
    public synchronized Class<?> getTargetClass() {
        return getInterfaceClass();
    }
}

//父类的getTarget方法会被spring的advice调用到，又会回调子类的createObject方法，模板设计模式
@Override
public synchronized Object getTarget() throws Exception {
    if (this.lazyTarget == null) {
        logger.debug("Initializing lazy target object");
        this.lazyTarget = createObject();
    }
    return this.lazyTarget;
}
```

```
private Object getCallProxy() throws Exception {
    if (referenceConfig == null) {
        throw new IllegalStateException("ReferenceBean is not ready yet, please make sure to
call reference interface method after dubbo is started.");
    }
    //get reference proxy
    return referenceConfig.get();
}
```

所以从这个流程看，@DubboReference依赖注入的对象其实就一个spring的代理对象，然后用这个代理对象调用的时候，最终会调到TargetSource对象的getTarget方法，由这个方法会调到referenceConfig.get()方法生成dubbo的代理对象，referenceConfig.get()这个方法也是引用dubbo服务的核心方法。