第四节: Spring与Dubbo整合原理与源码分析

课程内容

笔记更新地址：

整体架构和流程

@EnableDubbo

DubboConfigConfigurationRegistrar

- 流程
- DubboConfigBindingRegistrar
- DubboConfigBindingBeanPostProcessor
- DefaultDubboConfigBinder
- 总结

DubboComponentScanRegistrar

- ServiceAnnotationBeanPostProcessor
  - DubboClassPathBeanDefinitionScanner
  - ServiceBean
- ReferenceAnnotationBeanPostProcessor

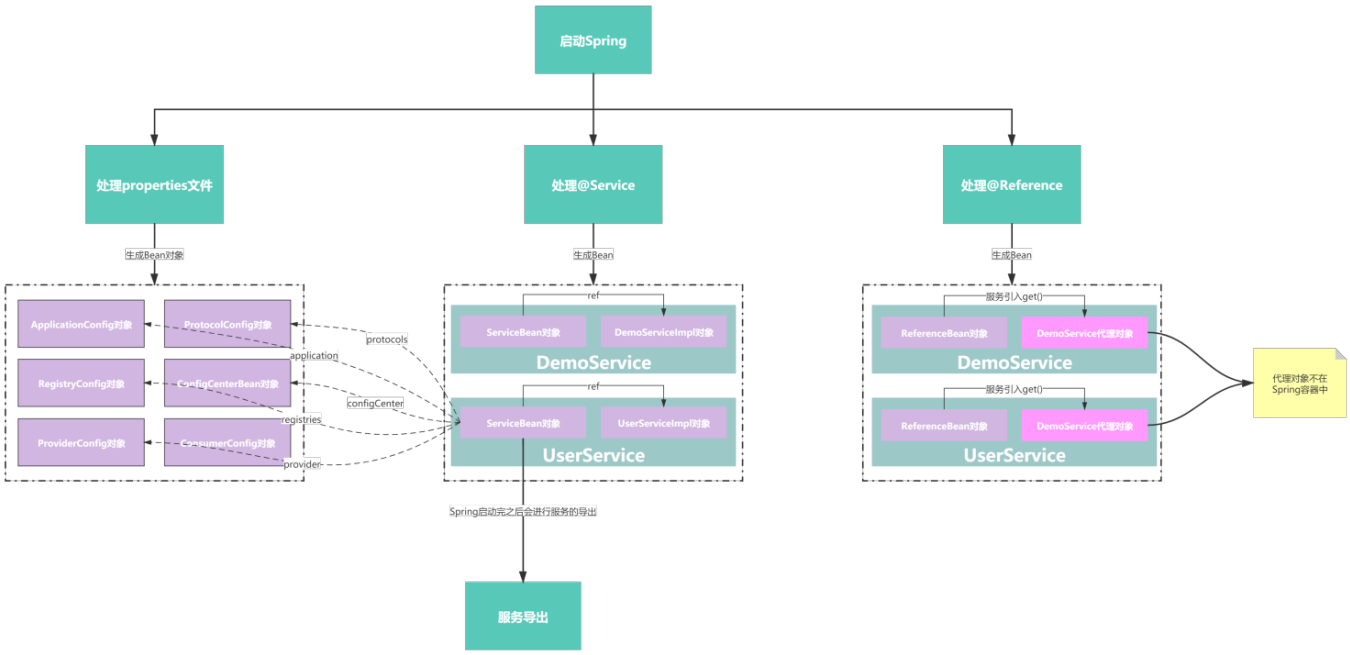
课程内容

1. Dubbo中propertie文件解析以及处理原理
2. Dubbo中@Service注解解析以及处理原理
3. Dubbo中@Reference注解解析以及处理原理

笔记更新地址：

https://www.yuque.com/books/share/f2394ae6-381b-4f44-819e-c231b39c1497（密码：kyys）
《Dubbo笔记》

整体架构和流程



应用启动类与配置

```
1  public class Application {
2      public static void main(String[] args) throws Exception {
3          AnnotationConfigApplicationContext context = new Annotati
   onConfigApplicationContext(ProviderConfiguration.class);
4          context.start();
5          System.in.read();
6      }
7
8      @Configuration
9      @EnableDubbo(scanBasePackages = "org.apache.dubbo.demo.provid
   er")
10     @PropertySource("classpath:/spring/dubbo-provider.properties"
   )
11     static class ProviderConfiguration {
12
13     }
14 }
```

应用配置类为ProviderConfiguration，在配置上有两个比较重要的注解

1. @PropertySource表示将dubbo-provider.properties中的配置项添加到Spring容器中，可以通过@Value的方式获取到配置项中的值

2. @EnableDubbo(scanBasePackages = "org.apache.dubbo.demo.provider")表示对指定包下的类进行扫描，扫描@Service与@Reference注解，并且进行处理

@EnableDubbo

在EnableDubbo注解上，有另外两个注解，也是研究Dubbo最重要的两个注解

1. @EnableDubboConfig
2. @DubboComponentScan

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@Import(DubboConfigConfigurationRegistrar.class)
public @interface EnableDubboConfig {
    boolean multiple() default true;
}
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(DubboComponentScanRegistrar.class)
public @interface DubboComponentScan {
    String[] value() default {};

    String[] basePackages() default {};

    Class<?>[] basePackageClasses() default {};

}
```

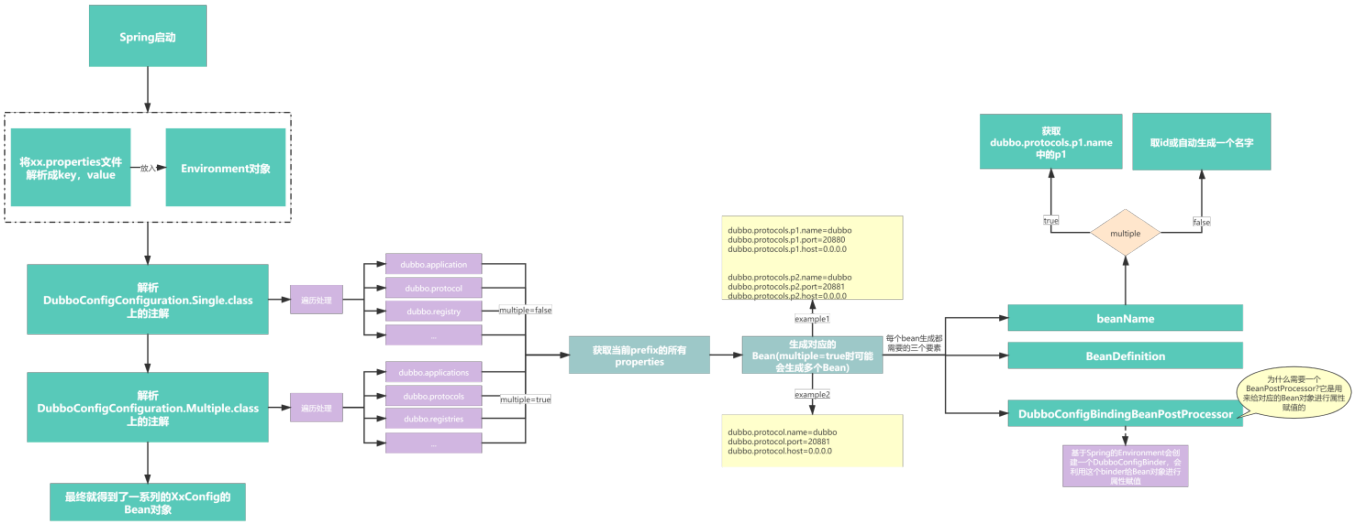
注意两个注解中对应的@Import注解所导入的类：

1. DubboConfigConfigurationRegistrar
2. DubboComponentScanRegistrar

Spring在启动时会解析这两个注解，并且执行对应的Registrar类中的registerBeanDefinitions方法（这是Spring中提供的扩展功能。）

DubboConfigConfigurationRegistrar

流程



Spring启动时，会调用DubboConfigConfigurationRegistrar的registerBeanDefinitions方法，该方法是利用Spring中的AnnotatedBeanDefinitionReader来读取：

1. DubboConfigConfiguration.Single.**class**
2. DubboConfigConfiguration.Multiple.**class**

这两个类上的注解。

```
@EnableDubboConfigBindings({
    @EnableDubboConfigBinding(prefix = "dubbo.application", type = ApplicationConfig.class),
    @EnableDubboConfigBinding(prefix = "dubbo.module", type = ModuleConfig.class),
    @EnableDubboConfigBinding(prefix = "dubbo.registry", type = RegistryConfig.class),
    @EnableDubboConfigBinding(prefix = "dubbo.protocol", type = ProtocolConfig.class),
    @EnableDubboConfigBinding(prefix = "dubbo.monitor", type = MonitorConfig.class),
    @EnableDubboConfigBinding(prefix = "dubbo.provider", type = ProviderConfig.class),
```

```
    @EnableDubboConfigBinding(prefix = "dubbo.consumer", type = C
onsumerConfig.class),
    @EnableDubboConfigBinding(prefix = "dubbo.config-center", typ
e = ConfigCenterBean.class),
    @EnableDubboConfigBinding(prefix = "dubbo.metadata-report", t
ype = MetadataReportConfig.class),
    @EnableDubboConfigBinding(prefix = "dubbo.metrics", type = Me
tricsConfig.class)
})
public static class Single {

}
```

```
@EnableDubboConfigBindings({
    @EnableDubboConfigBinding(prefix = "dubbo.applications", type
= ApplicationConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.modules", type = Mo
duleConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.registries", type =
RegistryConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.protocols", type =
ProtocolConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.monitors", type = M
onitorConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.providers", type =
ProviderConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.consumers", type =
ConsumerConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.config-centers", ty
pe = ConfigCenterBean.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.metadata-reports",
type = MetadataReportConfig.class, multiple = true),
    @EnableDubboConfigBinding(prefix = "dubbo.metricses", type =
MetricsConfig.class, multiple = true)
})
public static class Multiple {

```

```
}
```

这两个类主要用到的就是@EnableDubboConfigBindings注解：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(DubboConfigBindingsRegistrar.class)
public @interface EnableDubboConfigBindings {

    /**
     * The value of {@link EnableDubboConfigBindings}
     *
     * @return non-null
     */
    EnableDubboConfigBinding[] value();

}
```

@EnableDubboConfigBindings注解上也有一个@Import注解，导入的是DubboConfigBindingsRegistrar.**class**。该类会获取@EnableDubboConfigBindings注解中的value，也就是多个@EnableDubboConfigBinding注解，然后利用DubboConfigBindingRegistrar去处理这些@EnableDubboConfigBinding注解。

DubboConfigBindingRegistrar

此类中的主要方法是registerDubboConfigBeans()方法，主要功能就是获取用户所设置的properties文件中的内容，对Properties文件进行解析，根据Properties文件的每个配置项的前缀、参数名、参数值生成对应的BeanDefinition。

比如：

```
dubbo.application.name=dubbo-demo-provider1-application
dubbo.application.logger=log4j
```

前缀为"dubbo.application"的配置项，会生成一个ApplicationConfig类型的BeanDefinition，并且name和logger属性为对应的值。

再比如：

```
dubbo.protocols.p1.name=dubbo
dubbo.protocols.p1.port=20880
dubbo.protocols.p1.host=0.0.0.0

dubbo.protocols.p2.name=dubbo
dubbo.protocols.p2.port=20881
dubbo.protocols.p2.host=0.0.0.0
```

比如前缀为"dubbo.protocols"的配置项，会生成两个ProtocolConfig类型的BeanDefinition，两个BeanDefinition的beanName分别为p1和p2。

并且还会针对生成的每个BeanDefinition生成一个和它一对一绑定的BeanPostProcessor，类型为DubboConfigBindingBeanPostProcessor.**class**。

DubboConfigBindingBeanPostProcessor

DubboConfigBindingBeanPostProcessor是一个BeanPostProcessor，在Spring启动过程中，会针对所有的Bean对象进行后置加工，但是在DubboConfigBindingBeanPostProcessor中有如下判断：

```
if (this.beanName.equals(beanName) && bean instanceof AbstractConf
ig)
```

所以DubboConfigBindingBeanPostProcessor并不会处理Spring容器中的所有Bean，它只会处理上文由Dubbo所生成的Bean对象。

并且，在afterPropertiesSet()方法中，会先创建一个DefaultDubboConfigBinder。

DefaultDubboConfigBinder

当某个AbstractConfig类型的Bean，在经过DubboConfigBindingBeanPostProcessor处理时，此时Bean对象中的属性是没有值的，会利用DefaultDubboConfigBinder进行赋值。底层就是利用Spring中的DataBinder技术，结合properties文件对对应的属性进行赋值。

对应一个AbstractConfig类型（针对的其实是子类，比如ApplicationConfig、RegistryConfig）的Bean，每个类都有一些属性，而properties文件是一个key-value对，所以实际上DataBinder就是将属性名和properties文件中的key进行匹配，如果匹配成功，则把value赋值给属性。具体DataBinder技术是如何工作的，请自行学习（不难）。

举个例子：

```
dubbo.application.name=dubbo-demo-provider1-application
dubbo.application.logger=log4j
```

对于此配置，它对应ApplicationConfig对象（beanName是自动生成的），所以最终ApplicationConfig对象的name属性的值为“dubbo-demo-provider1-application”，logger属性的值为“log4j”。

对于

```
dubbo.protocols.p1.name=dubbo
dubbo.protocols.p1.port=20880
dubbo.protocols.p1.host=0.0.0.0
```

它对应ProtocolConfig对象（beanName为p1）,所以最终ProtocolConfig对象的name属性的值为“dubbo”，port属性的值为20880，host属性的值为“0.0.0.0”。

这样就完成了对properties文件的解析。

总结

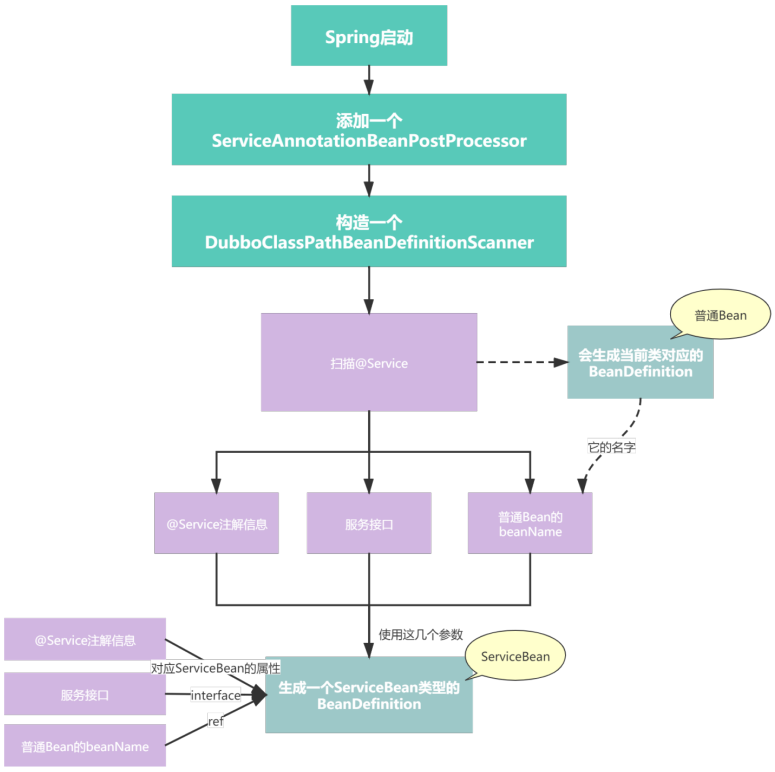
DubboConfigConfigurationRegistrar的主要作用就是对propties文件进行解析并根据不同的配置项项生成对应类型的Bean对象。

DubboComponentScanRegistrar

DubboConfigConfigurationRegistrar的作用是向Spring容器中注册两个Bean:

1. ServiceAnnotationBeanPostProcessor
2. ReferenceAnnotationBeanPostProcessor

ServiceAnnotationBeanPostProcessor



ServiceAnnotationBeanPostProcessor是一个BeanDefinitionRegistryPostProcessor，是用来注册BeanDefinition的。

它的主要作用是扫描Dubbo的@Service注解，一旦扫描到某个@Service注解就把它以及被它注解的类当做一个Dubbo服务，进行**服务导出**。

DubboClassPathBeanDefinitionScanner

DubboClassPathBeanDefinitionScanner是所Dubbo自定义的扫描器，继承了Spring中的ClassPathBeanDefinitionScanner了。

DubboClassPathBeanDefinitionScanner相对于ClassPathBeanDefinitionScanner并没有做太多的改变，只是把useDefaultFilters设置为了false，主要是因为Dubbo中的@Service注解是Dubbo自定义的，在这个注解上并没有用@Component注解（因为Dubbo不是一定要结合Spring才能用），所以为了能利用Spring的扫描逻辑，需要把useDefaultFilters设置为false。

没扫描到一个@Service注解，就会得到一个BeanDefinition，这个BeanDefinition的beanClass属性就是具体的服务实现类。

但，如果仅仅只是这样，这只是得到了一个Spring中的Bean，对于Dubbo来说此时得到的Bean是一个**服务**，并且，还需要解析@Service注解的配置信息，因为这些都是服务的参数信息，所以在扫描完了之后，

会针对所得到的每个BeanDefinition，都会**额外**的再生成一个**ServiceBean**类型的Bean对象。

ServiceBean

ServiceBean表示一个Dubbo服务，它有一些参数，比如：

1. **ref，表示服务的具体实现类**
2. **interface，表示服务的接口**
3. **parameters，表示服务的参数（@Service注解中所配置的信息）**
4. **application，表示服务所属的应用**
5. **protocols，表示服务所使用的协议**
6. **registries，表示服务所要注册的注册中心**

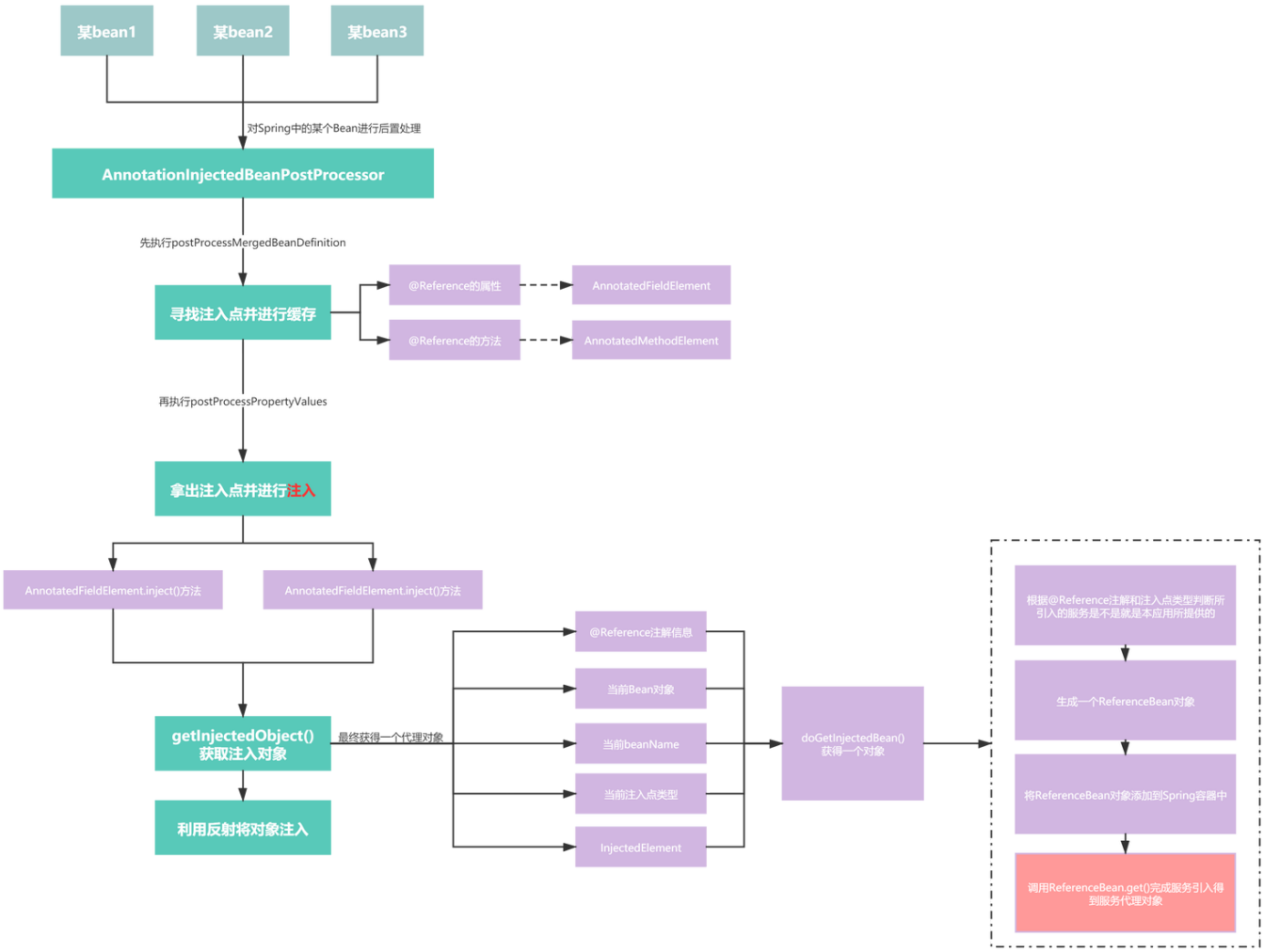
所以在扫描到一个@Service注解后，其实会得到两个Bean:

1. 一个就是服务实现类本身一个Bean对象
2. 一个就是对应的ServiceBean类型的一个Bean对象

并且需要注意的是，ServiceBean实现了ApplicationListener接口，所以当Spring启动完成后会触发onApplicationEvent()方法的调用，而在这个方法内会调用**export()**，这个方法就是**服务导出的入口方法**。

> 关于RuntimeBeanReference参考https://www.yuque.com/renyong-jmovm/ufz328/gbwvk7。

ReferenceAnnotationBeanPostProcessor



ReferenceAnnotationBeanPostProcessor是处理@Reference注解的。

ReferenceAnnotationBeanPostProcessor的父类是AnnotationInjectedBeanPostProcessor，是一个InstantiationAwareBeanPostProcessorAdapter，是一个BeanPostProcessor。

Spring在对Bean进行依赖注入时会调用AnnotationInjectedBeanPostProcessor的postProcessPropertyValues()方法来给某个Bean按照ReferenceAnnotationBeanPostProcessor的逻辑进行依赖注入。

在注入之前会查找注入点，被@Reference注解的属性或方法都是注入点。

针对某个Bean找到所有注入点之后，就会进行注入了，注入就是给属性或给set方法赋值，但是在赋值之前得先得到一个值，此时就会调用ReferenceAnnotationBeanPostProcessor的**doGetInjectedBean()**方法来得到一个对象，而这个对象的构造就比较复杂了，因为对于Dubbo来说，注入给某个属性的应该是当前这个**属性所对应的服务接口的代理对象**。

但是在生成这个代理对象之前，还要考虑问题：

1. 当前所需要引入的这个服务，是不是在本地就存在？不存在则要把按Dubbo的逻辑生成一个代理对象
2. 当前所需要引入的这个服务，是不是已经被引入过了（是不是已经生成过代理对象了），如果是应该是不用再重复去生成了。

首先如何判断当前所引入的服务是本地的一个服务（就是当前应用自己所提供的服务）。

我们前面提到，Dubbo通过@Service来提供一个服务，并且会生成两个Bean：

1. 一个服务实现类本身Bean
2. 一个ServiceBean类型的Bean，这个Bean的名字是这么生成的：

```
1 private String generateServiceBeanName(AnnotationAttributes servic
  eAnnotationAttributes, Class<?> interfaceClass) {
2     ServiceBeanNameBuilder builder = create(interfaceClass, enviro
  nment)
3                 .group(serviceAnnotationAttributes.getString("grou
  p"))
4                 .version(serviceAnnotationAttributes.getString("ve
  rsion"));
5     return builder.build();
6 }
```

是通过接口类型+group+version来作为ServiceBean类型Bean的名字的。

所以现在对于服务引入，也应该提前根据@Reference注解中的信息和属性接口类型去判断一下当前Spring容器中是否存在对应的ServiceBean对象，如果存在则直接取出ServiceBean对象的ref属性所对应的对象，作为要注入的结果。

然后如何判断当前所引入的这个服务是否已经被引入过了（是不是已经生成过代理对象了）。

这就需要在第一次引入某个服务后（生成代理对象后）进行缓存（记录一下）。Dubbo中是这么做的：

1. 首先根据@Reference注解的所有信息+属性接口类型生成一个**字符串**
2. 然后@Reference注解的所有信息+属性接口类型生成一个ReferenceBean对象（**ReferenceBean对象中的get方法可以得到一个Dubbo生成的代理对象，可以理解为服务引入的入口方法**）
3. 把字符串作为beanName，ReferenceBean对象作为bean注册到Spring容器中，同时也会放入**referenceBeanCache**中。

有了这些逻辑，@Reference注解服务引入的过程是这样的：

1. 得到当前所引入服务对应的ServiceBean的beanName（源码中叫referencedBeanName）

2. 根据@Reference注解的所有信息+属性接口类型得到一个referenceBeanName
3. 根据referenceBeanName从**referenceBeanCach**e获取对应的ReferenceBean，如果没有则创建一个ReferenceBean
4. 根据referencedBeanName（ServiceBean的beanName）判断Spring容器中是否存在该bean，如果存在则给ref属性所对应的bean取一个别名，别名为referenceBeanName。
    a. 如果Spring容器中不存在referencedBeanName对应的bean，则判断容器中是否存在referenceBeanName所对应的Bean，如果不存在则将创建出来的ReferenceBean注册到Spring容器中（**此处这么做就支持了可以通过@Autowired注解也可以使用服务了，ReferenceBean是一个FactoryBean**）
5. 如果referencedBeanName存在对应的Bean，则额外生成一个代理对象，代理对象的InvocationHandler会缓存在**localReferenceBeanInvocationHandlerCache中，这样如果引入的是同一个服务，并且这个服务在本地，**
6. 如果referencedBeanName不存在对应的Bean，则直接调用ReferenceBean的get()方法得到一个代理对象