# 前言

Dubbo是阿里巴巴开源的一款分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案(主要是这两个功能)。Dubbo采用Spring配置方式集成，这样避免了代码侵入

1. 概述
   1. 什么是RPC

我们先来看看什么是RPC，RPC全称为Remote Procedure Call，翻译过来为“远程过程调用”， 也就是说假设有两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数**/**方法，由于两个应用不在同一个内存空间内运行，因此不能直接调用，需要通过网络来表达调用的语义和传达调用的数据，这就是所谓的RPC

* 1. 什么是SOA

SOA是面向服务的架构，即把系统分离成不同的服务，服务和服务之间使用接口来进行数据交互，最终达到整合系统的目的。

举一个简单的例子：

程序员A负责考勤模块的开发，程序员B负责薪资模块的开发

B：“A，我要用考勤的数据，你考勤模块做好了吗，数据录了吗？”

A：“早就做好了，考勤表叫\*\*\*，\*\*字段代表员工id，\*\*字段代表员工出勤天数，\*\*字段代表..”

这是我们传统开发中日常出现的场景，模块之间的数据交互通过数据库进行，A模块存入考勤数据，B模块读取考勤数据并处理，模块与模块之间耦合。

那么SOA是怎么处理的呢？

考勤作为单独模块，成为一个考勤服务，发布了一个考勤数据接口（WebServices），向外提供查询考勤数据的接口

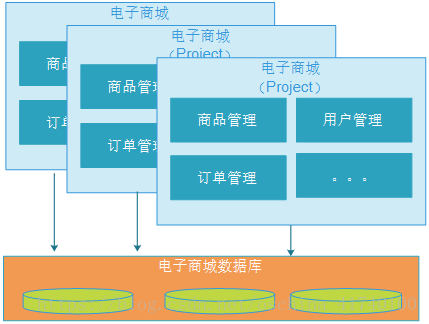
B需要使用考勤数据，只需要调用考勤服务的接口即可，这样就实现了模块分离，模块间要进行数据交互，通过接口来完成，这就是SOA

那么SOA有哪些好处呢？

1. 模块间解耦，B只需要调用A的接口就能得到想要的数据，并不关心A是如何获取数据的，这样一来即使更改数据库的表结构，只需要重写A的接口即可(试想一下如果多个模块需要A的接口，如果没有SOA，那么多个模块都得重写接口)
2. 拆分系统更为方便，在用户量越来越大的情况下，数据库服务器和应用服务器的压力也越来越大，解决方法是拆分系统为几个子模块，子模块分别使用不同的数据库服务器和应用服务器(分布式)
3. 可扩展性强，当功能被严格划分后，新的功能有时可以通过原有功能的拆分重组实现，就像搭积木一样，把积木拆了就可能重新搭出新的积木模型
   1. 为什么需要dubbo

网站的架构一般要经历如下的发展过程

* **单一应用架构**



**特点：**

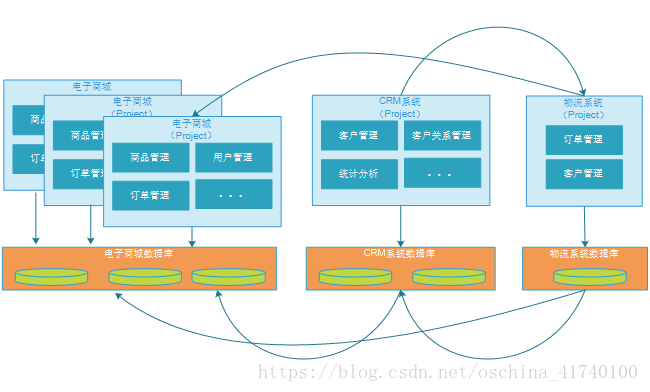
* + 当网站流量很小时，只需一个应用，将所有功能都集成在一个项目中，打成一个war包部署到服务器(可能是多个服务器通过负载均衡技术进行集群部署)，以减少部署节点和成本。
  + 应用和数据库分开部署到应用服务器和数据库服务器上
  + 此时，用于简化增删改查工作量的**数据访问ORM框架(如Mybaits)**是关键。
  + 可以通过部署应用集群和数据库集群来提高系统的性能。

**优点：**

* + 项目架构简单，前期开发成本低，周期短，小型项目的首选。

**缺点：**

* + 功能集成在一个项目，耦合度高，不利于开发和扩展。
  + 系统性能扩展只能通过增加集群结点，成本高、有瓶颈。
  + 必须采用单一技术栈进行开发
* **垂直应用架构**



**特点：**

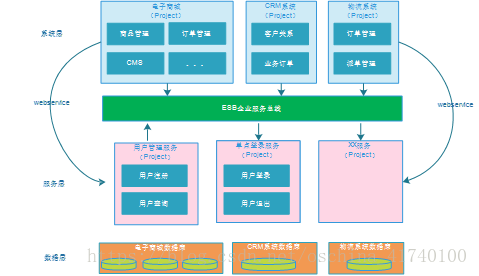
* + 当访问量逐渐增大，单一应用增加机器带来的加速度越来越小(一般来说流量洪峰过去后，大量部署的服务器就会出现资源浪费)，将大应用垂直拆成互不相干的几个单体应用，以提升效率。如上图将商城项目中的用户关系管理(CRM)和物流系统拆分出来
  + 此时，用于加速前端页面开发的**Web框架(MVC)**是关键
  + 项目与项目之间的存在数据冗余，耦合性较大，比如上图中三个项目都存在客户信息。

**优点：**

* + 项目架构简单，前期开发成本低，周期短，小型项目的首选。
  + 通过垂直拆分，对原项目进行了一定程度上的解耦
  + 不同项目可以采用不同技术栈开发

**缺点：**

* + 耦合度仍然过高，不利于开发和扩展。
  + 系统性能扩展只能通过增加集群结点，成本高、有瓶颈。
* **分布式服务架构**



**特点：**

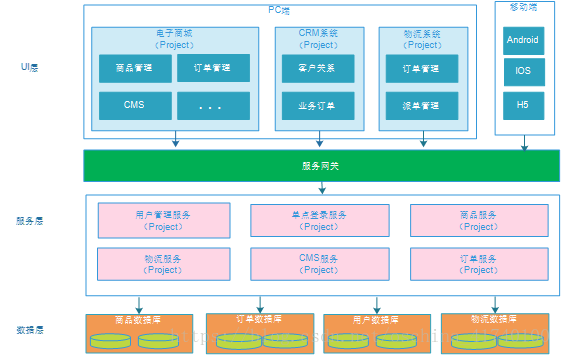
* + 当垂直拆分出来的应用越来越多，应用之间交互不可避免，这个时候将多个应用中的核心业务功能抽取出来，形成一个个公用的服务，逐渐组成稳定的服务中心，给上层应用提供服务，使上层应用能更快速的响应多变的市场需求，上层应用直接调用公共服务，提高复用率。
  + 各项目（系统）与服务之间采用webservice、rpc等方式进行通信。
  + ESB企业服务总线(绿色)作为项目与服务之间通信的桥梁。
  + 此时，用于提高业务复用及整合的  **分布式服务框架(RPC)**  是关键。

**优点：**

* + 将重复的功能抽取为服务，提高开发效率，提高系统的可重用性、可维护性。
  + 可以针对不同服务的特点制定集群及优化方案。
  + 采用ESB企业服务总线减少接口耦合

**缺点：**

* + 系统与服务的界限模糊，不利于开发及维护。
  + 虽然使用了ESB，但是服务的接口协议不固定，种类繁多，不利于系统维护。
  + 抽取的服务的粒度过大，系统与服务之间耦合性高。
* **微服务架构**



* + 当服务越来越多，各个服务占有的资源越来越难以把控，比如一些压力小的服务，可能占据了大量硬件资源，我们需要基于访问流量实时监控和管理各个服务
  + 将系统服务层完全独立出来，并将服务层抽取为一个一个的微服务。这些服务的创建仅限于一个特定的业务功能，如用户管理、用户角色、电子商务车、搜索引擎、社交媒体登录等。此外，它们是完全独立的，也就是说它们可以写入不同的编程语言并使用不同的数据库。
  + 微服务遵循单一原则，即单个服务功能完整，职责单一。
  + 微服务之间采用RESTful等轻量协议传输。
  + 相比于SOA，微服务抽取出来的粒度更细

<http://www.cnblogs.com/pflee/p/4507579.html>

* 1. dubbo的运行流程

在了解如何使用Dubbo之前，我们首先了解一下Dubbo中的几个角色：

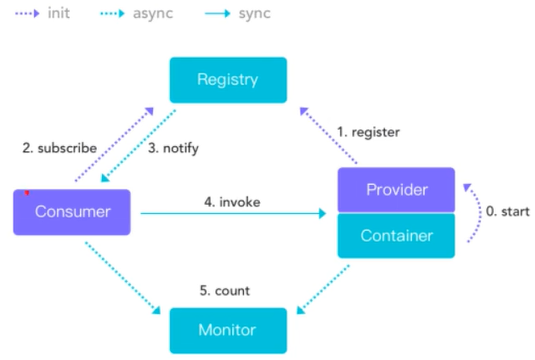
**运行容器（Container）：**负责启动，加载，运行服务提供者(provider)，可以是tomcat等

**服务器注册中心（Register）：**服务注册的中心，服务提供者需要向注册中心注册自己提供的服务

**服务提供方（Provider）：**服务提供方暴露服务，将服务提供给消费方使用

**服务消费方（Consumer）：**调用远程服务的服务消费方

**统计监控中心（Monitor）：**统计服务的调用次调和调用时间的监控中心



**紫色代表初始化操作，蓝色虚线代表异步请求，蓝色实现代表同步请求**

0、服务容器负责启动，加载，运行服务提供者。

1、服务提供者在启动时，向注册中心注册自己提供的服务。

2、服务消费者在启动时，向注册中心订阅自己所需的服务。

3、注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

4、服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用(因为该服务可能是集群部署)，如果调用失败，再选另一台调用。

5、服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

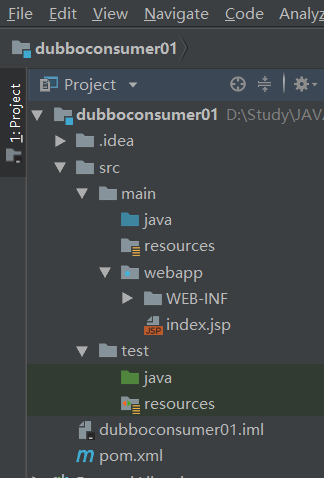
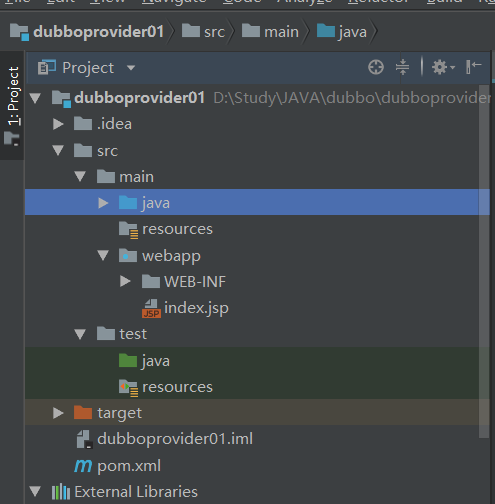
1. Dubbo的使用
   1. HelloWorld
      1. 不使用注册中心

在开发和测试环境下，为了方便起见，我们常常绕过注册中心(生产环境中不允许)，直接连接到服务提供者，这就是无注册中心的点对点直连，我们先从这里开始

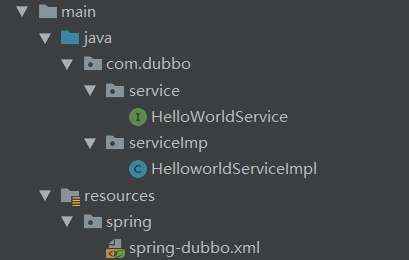
首先我们需要加载dubbo的jar包，由于dubbo和spring无缝集成(也可以和spring boot集成)，我们首先需要导入spring(或者spring boot)的jar包，然后导入dubbo的jar包

<dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>dubbo</artifactId>  
 <version>${dubboVersion}</version>  
 <!--dubbo默认会再加载一个2.5版本的spring进来，这里阻止该行为防止spring版本冲突-->  
 <exclusions>  
 <exclusion>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>

接下来，我们创建两个工程——服务提供者项目和服务消费者项目(简单的demo，各创建一个就好)，如图创建了一个provider01和consumer01，这里我们假设provider01为consumer01提供服务：



接下来，我们在服务提供者provider01中写一个服务类



HelloWorldService.java

public interface HelloWorldService {  
 String say(String name);  
}

HelloworldServiceImpl.java

public class HelloworldServiceImpl implements HelloWorldService {  
 public String say(String name) {  
 return name + ",hello world!";  
 }  
}

spring-dubbo.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"  
 xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"  
 xmlns:tx="http://www.springframework.org/schema/tx"  
 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  
 http://code.alibabatech.com/schema/dubbo  
 http://code.alibabatech.com/schema/dubbo/dubbo.xsd">  
 <!--指定服务提供者的名称-->  
 <dubbo:application name="dubbo-provider01"></dubbo:application>  
 <!--扫描并将com.dubbo.serviceImpl类注册为bean-->  
 <bean id="helloWorldService" class="com.dubbo.serviceImp.HelloworldServiceImpl"></bean>  
 <!--配置服务提供者的服务，register="N/A"是绕过注册中心的配置-->  
 <dubbo:service interface="com.dubbo.service.HelloWorldService" ref="helloWorldService" register="false"></dubbo:service>  
</beans>

然后我们给web.xml添加监听器，保证项目一启动就加载spring配置文件

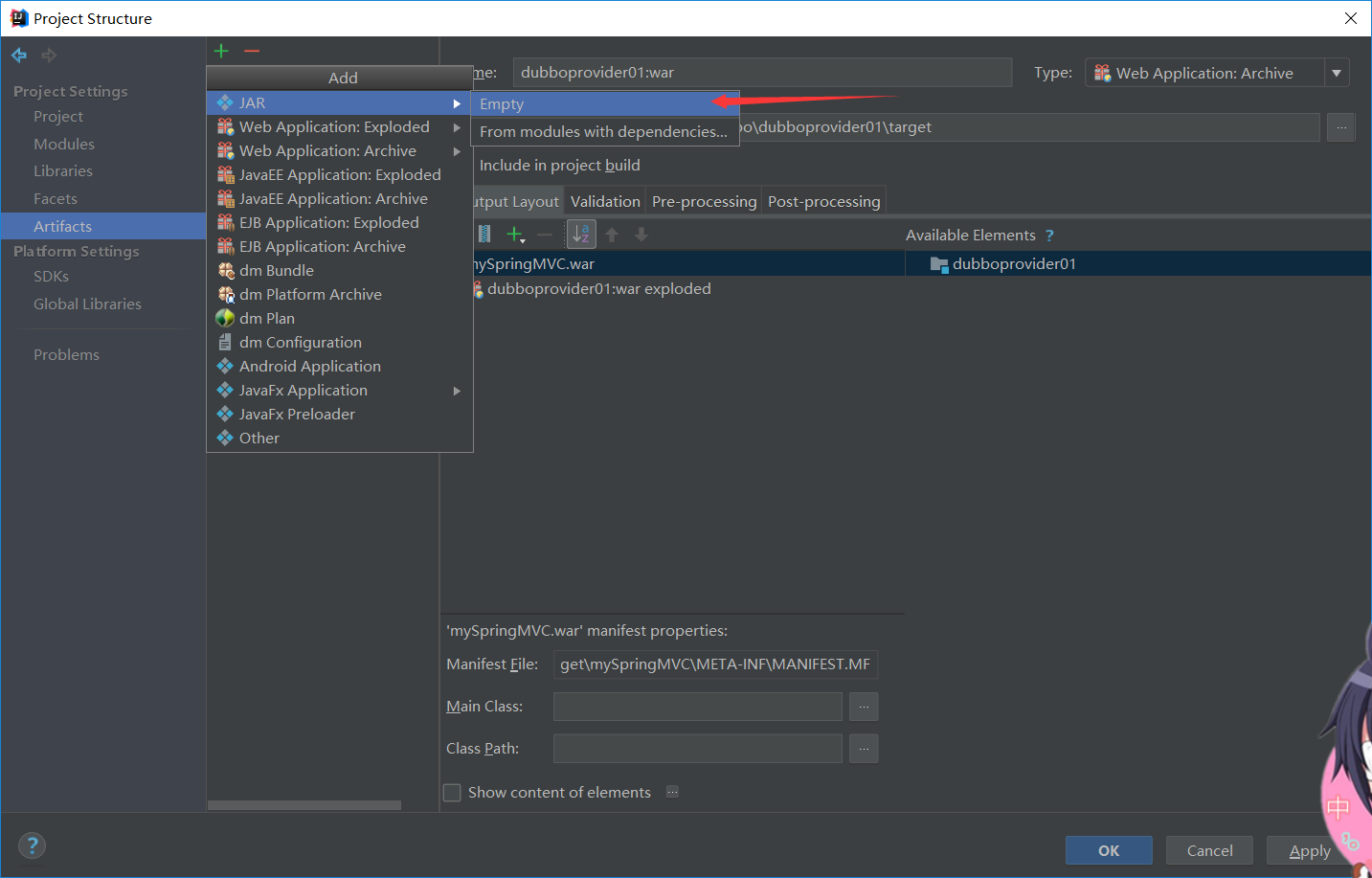
<!--加载spring配置文件-->  
<display-name>Archetype Created Web Application</display-name>  
<context-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath\*:spring/spring-\*.xml</param-value>  
</context-param>  
<listener>  
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>

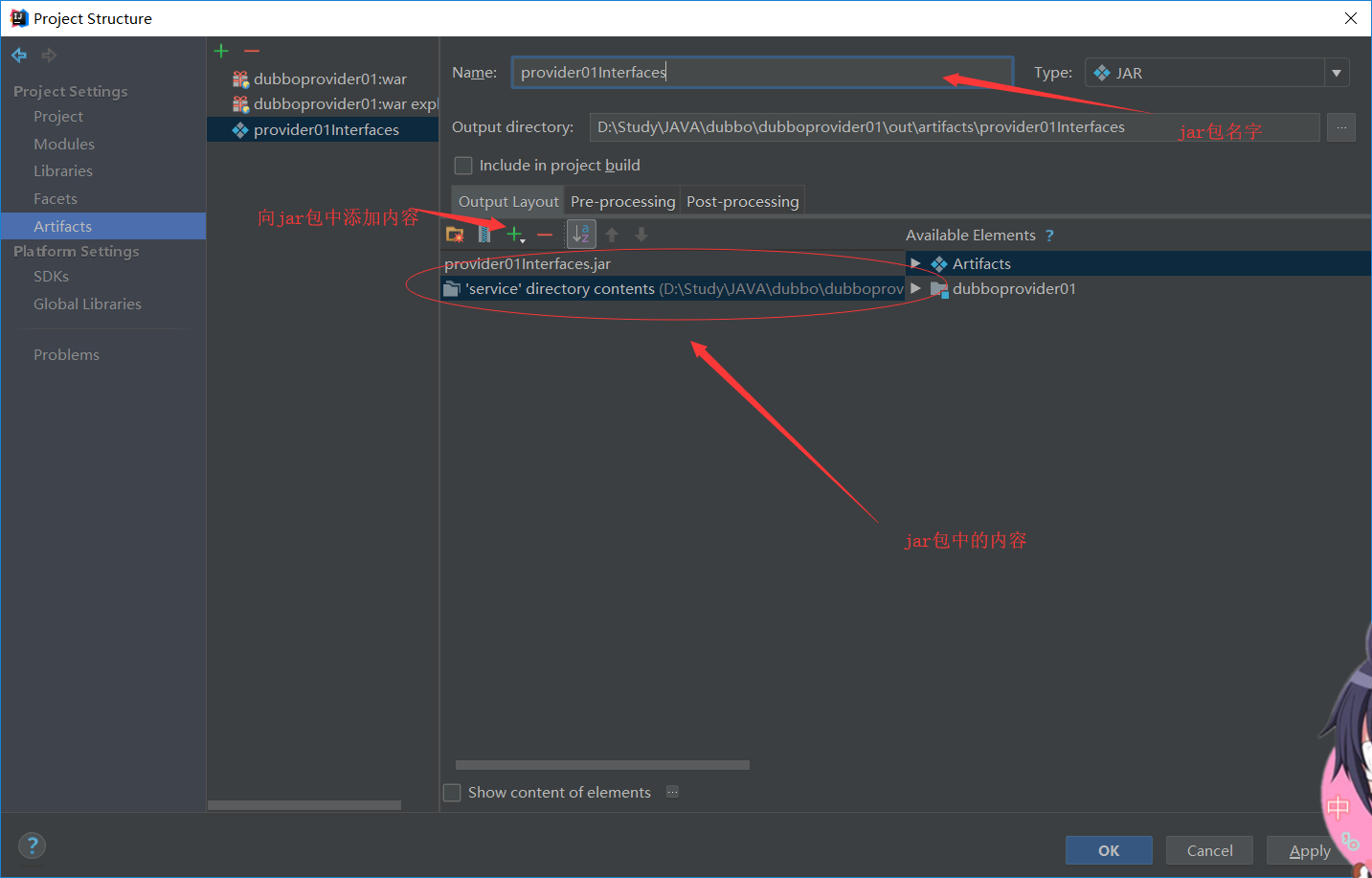
到这里，服务提供者就完成了，接下来是服务消费者

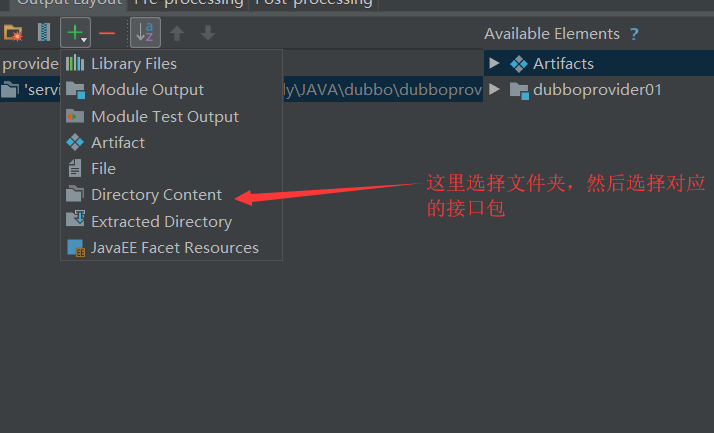
首先和上面一样，我们需要导入dubbo的jar包到消费者项目中去，同时给web.xml设置好spring配置文件自动加载的监听器

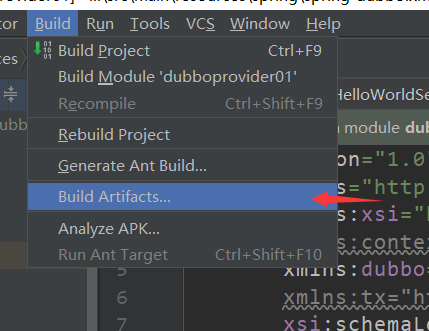
然后，我们需要把服务提供者的服务接口包打成jar包给服务消费者使用。注意，不需要服务实现类，只需要interface即可，也就是com.dubbo.service里面的接口

接下来演示idea打包过程，首先选择File-project Structure-Artifacts









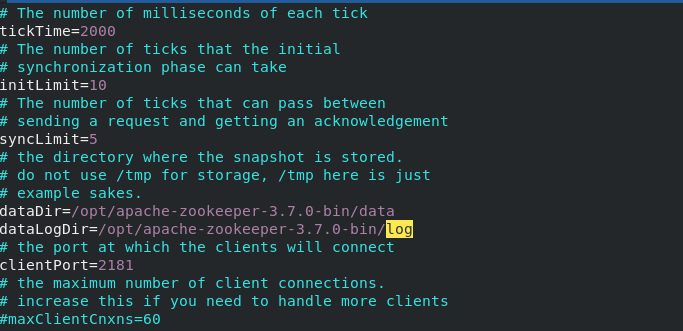
打包之后的jar包存放的地址在第二张图中红字”jar包名字”那一行

mvn install:install-file -Dfile=D:\Study\JAVA\dubbo\dubboconsumer01\out\artifacts\provider01Interface\provider01Interface.jar -DgroupId=com.dubbo -DartifactId=dubbo-provider01 -Dversion=1.0-SNAPSHOT -Dpackaging=jar

将打好的包注册到本地库中通过maven加载到消费者项目中去

* + 1. 使用zookeeper作为注册中心
       1. 安装启动zookeeper

1. 首先下载zookeeper包apache-zookeeper-3.7.0-bin.tar
2. 然后通过tar –zxf apache-zookeeper-3.7.0-bin.tar解压缩
3. 将conf/zoo\_sample.cfg复制为zoo.cfg，并修改zk的数据存储位置
4. 在/apache-zookeeper-3.7.0-bin/下创建data目录，用于存放zookeeper数据：mkdir data
5. 同样操作创建日志文件夹
6. 打开zoo.cfg，修改日志和data目录地址为刚才创建的地址



单机部署的话，配置文件写到这里就结束了，但zookeeper一般是集群部署

如果是集群的话，还需要在zoo.cfg中增加集群配置：

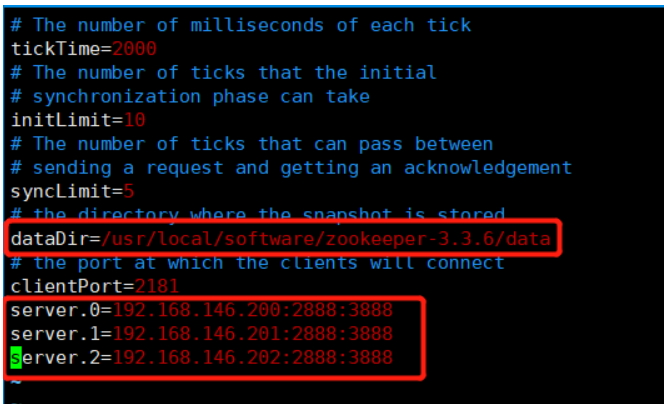
**server.A=B:C:D**

**A：其中 A 是一个数字，表示这个是服务器的编号，服务器编号配置在myid文件中，这个下面会讲到如何配置；**

**B：是集群的各个服务器的 ip 地址；**

**C：Zookeeper服务器之间的通信端口；**

**D：Leader选举的端口。**



1. 如果是集群部署的话还需要在dataDir配置的目录下创建myid文件

然后在该文件添加上一步 server 配置的对应 A 数字。

　　比如我们上面的配置：

server.0=192.168.146.200:2888:3888

server.1=192.168.146.201:2888:3888

server.2=192.168.146.202:2888:3888

那么就必须在 192.168.146.200 机器的的 /usr/local/software/zookeeper-3.3.6/data 目录下创建 myid 文件，然后在该文件中写上 0 即可。

同理，201、202也需要创建对应的myid文件

* 1. 负载均衡

Dubbo 提供了多种均衡策略，默认的负载均衡策略为 random 随机调用

可以在provider端的@Service中配置负载均衡策略，eg：



也可以在consumer端的@Reference中配置，如果两端都配置，则以consumer端为准



常用的负载均衡策略有：

* 1. 随机(Random LoadBalance)
  2. 轮询(RoundRobin LoadBalance)
  3. 最少活跃调用数(LeastActive LoadBalance)
  4. 一致性Hash(ConsistentHash LoadBalance)

在dubbo中，所有的负载均衡实现类均继承自 AbstractLoadBalance，该类实现了 LoadBalance 接口方法，并封装了一些公共的逻辑。所以在分析负载均衡实现之前，先来看一下 AbstractLoadBalance 的逻辑。首先来看一下负载均衡的入口方法 select，如下：

@Override  
public <T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) {  
 if (CollectionUtils.*isEmpty*(invokers)) {  
 return null;  
 }

// 如果 invokers 列表中仅有一个 Invoker，直接返回即可，无需进行负载均衡  
 if (invokers.size() == 1) {  
 return invokers.get(0);  
 }

// 调用 doSelect 方法进行负载均衡，该方法为抽象方法，由子类实现  
 return doSelect(invokers, url, invocation);  
}  
  
protected abstract <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation);  
  
*/\*\** \* 服务提供者权重计算逻辑 *\** ***@param*** *invoker the invoker  
 \** ***@param*** *invocation the invocation of this invoker  
 \** ***@return*** *weight  
 \*/*int getWeight(Invoker<?> invoker, Invocation invocation) {  
 int weight = 0;  
 URL url = invoker.getUrl();  
 // Multiple registry scenario, load balance among multiple registries.  
 if (url.getServiceInterface().equals("org.apache.dubbo.registry.RegistryService")) {  
 weight = url.getParameter(*REGISTRY\_KEY* + "." + *WEIGHT\_KEY*, *DEFAULT\_WEIGHT*);  
 } else {

// 从服务提供者的 url 中获取 weight 配置值  
 weight = url.getMethodParameter(invocation.getMethodName(), *WEIGHT\_KEY*, *DEFAULT\_WEIGHT*);  
 if (weight > 0) {

// 获取服务提供者启动时间戳  
 long timestamp = invoker.getUrl().getParameter(*TIMESTAMP\_KEY*, 0L);  
 if (timestamp > 0L) {

// 计算服务提供者运行时长  
 long uptime = System.currentTimeMillis() - timestamp;  
 if (uptime < 0) {  
 return 1;  
 }

// 获取服务预热时间，默认为10分钟

//服务预热就是服务刚启动的时候需要一段磨合期，这段期间，服务运行状态没有达到最佳。

因此，如果这时候一下子将服务流量提升到正常水平，就有可能导致大量请求超时或者瞬间将系统压垮。

所以，服务刚启动时，需要慢慢增加流量，直到一段时间之后达到阈值上限，让其运行状态达到最佳  
 int warmup = invoker.getUrl().getParameter(*WARMUP\_KEY*, *DEFAULT\_WARMUP*);

// 如果服务运行时间小于预热时间，则重新计算服务权重，即降权  
 if (uptime > 0 && uptime < warmup) {

// 重新计算服务权重  
 weight = *calculateWarmupWeight*((int)uptime, warmup, weight);  
 }  
 }  
 }  
 }

// 返回weight权重  
 return Math.max(weight, 0);  
}

//当服务运行时长小于服务预热时间时，对服务进行降权，避免让服务在启动之初就处于高负载状态

static int calculateWarmupWeight(int uptime, int warmup, int weight) {

// 计算权重，下面代码逻辑上形似于 (uptime / warmup) \* weight。

// 随着服务运行时间 uptime 增大，权重计算值 ww 会慢慢接近配置值 weight  
 int ww = (int) ( uptime / ((float) warmup / weight));  
 return ww < 1 ? 1 : (Math.min(ww, weight));  
}

* + 1. 加权随机(Random LoadBalance)

根据权重随机选择（对加权随机算法的实现）。这是Dubbo默认采用的一种负载均衡策略。

优点：简单容易实现

缺点：当调用次数比较少时，Random 产生的随机数可能会比较集中，此时多数请求会落到同一台服务器上。这个缺点并不是很严重，多数情况下可以忽略。RandomLoadBalance 是一个简单，高效的负载均衡实现，因此 Dubbo 选择它作为缺省实现。

源码如下：

public class RandomLoadBalance extends AbstractLoadBalance {  
  
 public static final String *NAME* = "random";  
  
 */\*\*  
 \* Select one invoker between a list using a random criteria  
 \** ***@param*** *invokers List of possible invokers  
 \** ***@param*** *url URL  
 \** ***@param*** *invocation Invocation  
 \** ***@param*** <*T*>  
 *\** ***@return*** *The selected invoker  
 \*/* @Override  
 protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  
 // 服务提供者的数量  
 int length = invokers.size();  
 // 是否每一个provider都有相同的权重?true代表都相同  
 boolean sameWeight = true;  
 // 每个provider的权重  
 int[] weights = new int[length];  
 // 第一个provider的权重  
 int firstWeight = getWeight(invokers.get(0), invocation);  
 weights[0] = firstWeight;  
 // 总权重   
 int totalWeight = firstWeight;  
 for (int i = 1; i < length; i++) {  
 int weight = getWeight(invokers.get(i), invocation);  
 weights[i] = weight;  
 // 累加到总权重中  
 totalWeight += weight;  
 if (sameWeight && weight != firstWeight) {

// 如果存在不相同的权重，则将其置为false  
 sameWeight = false;  
 }  
 }

//如果存在provider的权重不相同，则获取随机数，并计算随机数落在哪个区间上  
 if (totalWeight > 0 && !sameWeight) {  
 // 获取一个[0, totalWeight)之间的随机数  
 int offset = ThreadLocalRandom.current().nextInt(totalWeight);  
 // 循环让 offset 数减去服务提供者权重值，当 offset 小于0时，返回相应的 Invoker。

// 还是用上面的例子进行说明，servers = [A, B, C]，weights = [5, 3, 2]，offset = 7。

// 第一次循环，offset - 5 = 2 > 0，说明 offset 肯定不会落在服务器 A 对应的区间上。

// 第二次循环，offset - 3 = -1 < 0，表明 offset 落在服务器 B 对应的区间上  
 for (int i = 0; i < length; i++) {  
 offset -= weights[i];  
 if (offset < 0) {  
 return invokers.get(i);  
 }  
 }  
 }  
 // 如果所有服务提供者权重值相同，此时直接随机返回一个即可  
 return invokers.get(ThreadLocalRandom.current().nextInt(length));  
 }  
  
}

* + 1. 加权轮询(RoundRobin LoadBalance)

轮询就是把请求依次分配给每个服务提供者。加权轮询就是在轮询的基础上，让更多的请求落到权重更大的服务提供者上

* + 1. 最少活跃调用数(LeastActive LoadBalance)

初始状态下所有服务提供者provider的活跃数均为 0（每个服务提供者的中特定方法都对应一个活跃数，也就是说服务提供者中的每一个方法的活跃数都是互相独立的），每收到一个请求后，对应的服务提供者的活跃数 +1，当这个请求处理完之后，活跃数 -1

在这种策略中，Dubbo 就认为谁的活跃数越少，谁的处理速度就越快，性能也越好，这样的话，dubbo就优先把请求给活跃数少的服务提供者处理。

如果有多个服务提供者的活跃数相等怎么办？很简单，那就再走一遍加权随机 RandomLoadBalance

源码如下：

public class LeastActiveLoadBalance extends AbstractLoadBalance {  
  
 public static final String *NAME* = "leastactive";  
  
 @Override  
 protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  
 // 服务提供者总数  
 int length = invokers.size();  
 // 最小的活跃数

int leastActive = -1;  
 // 具有相同“最小活跃数”的服务者提供者数量  
 int leastCount = 0;  
 // leastIndexs 用于记录具有相同“最小活跃数”的 Invoker 在 invokers 列表中的索引下标  
 int[] leastIndexes = new int[length];  
 // the weight of every invokers  
 int[] weights = new int[length];  
 // 总权重  
 int totalWeight = 0;  
 // 第一个最小活跃数的 Invoker 权重值，用于与其他具有相同最小活跃数的 Invoker 的权重进行对比，

// 以检测是否所有具有相同最小活跃数的 Invoker 的权重均相等  
 int firstWeight = 0;  
 // 是否每一个provider都有相同的权重?true代表都相同  
 boolean sameWeight = true;  
  
 for (int i = 0; i < length; i++) {  
 Invoker<T> invoker = invokers.get(i);  
 // 获取 Invoker 对应的活跃数  
 int active = RpcStatus.*getStatus*(invoker.getUrl(), invocation.getMethodName()).getActive();  
 // 获取权重. 默认值100（考虑过预热后的权重）

int afterWarmup = getWeight(invoker, invocation);  
 // 保存权重

weights[i] = afterWarmup;  
 // 通过一次次循环找到最小活跃数   
 if (leastActive == -1 || active < leastActive) {  
 // 发现更小的活跃数，重新初始化各项记录数据

leastActive = active;  
 // 因为发现了更小的活跃数，因此重设相同活跃数的invoker统计数量

leastCount = 1;  
 // 记录索引下标  
 leastIndexes[0] = i;  
 //重设总权重  
 totalWeight = afterWarmup;  
 // Record the weight the first least active invoker  
 firstWeight = afterWarmup;  
 // Each invoke has the same weight (only one invoker here)  
 sameWeight = true;  
 } else if (active == leastActive) {  
 //如果当前 Invoker 的活跃数 active 与最小活跃数 leastActive 相同  
 //将相同活跃数的invoker的下标记录下来

leastIndexes[leastCount++] = i;  
 //累加总权重数  
 totalWeight += afterWarmup;  
 // 判断是否每一个invoker的权重都相同  
 if (sameWeight && i > 0 && afterWarmup != firstWeight) {  
 sameWeight = false;  
 }  
 }  
 }

// 当只有一个 Invoker 具有最小活跃数，此时直接返回该 Invoker 即可  
 if (leastCount == 1) {  
 return invokers.get(leastIndexes[0]);  
 }

if (!sameWeight && totalWeight > 0) {  
 //如果有多个 Invoker 具有相同的最小活跃数，但他们的权重不同

//重新走一遍加权随机的逻辑  
 int offsetWeight = ThreadLocalRandom.current().nextInt(totalWeight);  
 for (int i = 0; i < leastCount; i++) {  
 int leastIndex = leastIndexes[i];  
 offsetWeight -= weights[leastIndex];  
 if (offsetWeight < 0) {  
 return invokers.get(leastIndex);  
 }  
 }  
 }

//如果权重相同或权重为0时，随机返回一个 Invoker  
 return invokers.get(leastIndexes[ThreadLocalRandom.current().nextInt(leastCount)]);  
 }  
}

* + 1. 一致性Hash(ConsistentHash LoadBalance)

在该策略下，具体是哪个服务提供者provider处理请求是由请求的参数决定的，也就是说相同参数的请求总是发到同一个服务提供者。

* 1. 服务超时

在provider和consumer中都可以配置服务超时

如果在服务端和消费端只在其中一方配置了timeout，那么没有歧义，都表示消费端调用服务的超时时间，消费端如果超过时间还没有收到响应结果，则消费端会抛超时异常，但服务端不会抛异常，服务端在执行服务后，会检查执行该服务的时间，如果超过timeout，则会打印一个超时日志。服务会正常的执行完。

如果在服务端和消费端各配了一个timeout，那就比较复杂了，假设

1. 服务执行为5s

2. 消费端timeout=3s

3. 服务端timeout=6s

那么消费端调用服务时，消费端会收到超时异常（因为消费端超时了），服务端一切正常（服务端没有超时）

* 1. 集群容错

当provider集群调用失败时，dubbo提供了对应的容错方案，默认容错方案为failover重试

dubbo提供的容错方案有以下几种：

1. Failover失败转移
2. Failfast快速失败
3. Failsafe失败安全
4. Failback失败自动恢复
5. Forking并行调用
6. Broadcast广播调用
   * 1. Failover失败转移（默认）

失败转移，当出现失败，重试其它服务器，通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次，即最多调用3次)。retries=2是默认配置

**该策略只适用于幂等操作**。对于非幂等操作，当服务处理超时或者网络问题导致服务响应超时，服务会被重复调用两次，这样一来重试会导致接口被多次调用，出现多次业务逻辑处理，会产生重复数据等问题。

* + 1. Failfast快速失败

快速失败，只发起一次调用，失败立即报错。

通常用于非幂等性的写操作，比如新增记录

* + 1. Failsafe失败安全

失败安全，出现异常时，直接忽略，以保障调用方(consumer)的正常运行。

通常用于写入日志等重要性较低的，与业务无关的操作

* + 1. Failback失败自动恢复

失败自动恢复，后台记录失败请求，异步定时重发。

通常用于消息通知操作，这类操作中，consumer不需要知道provider的执行结果，因此可以不需要同步等待服务返回，出现异常时可以异步重试

* + 1. Forking并行调用

并行调用多个服务器，取最先返回的结果。通常用于实时性要求较高的读操作，但比较浪费服务资源。可通过 forks="2" 来设置最大并行数

* + 1. Broadcast广播调用

广播调用所有提供者，逐个调用，任意一台报错则报错。

通常用于通知所有服务提供者更新缓存或日志等本地资源信息。

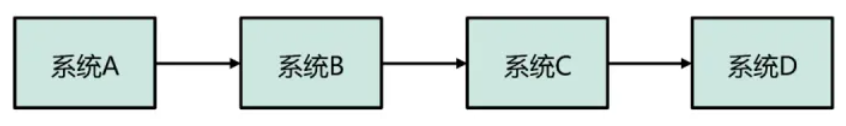
在广播调用中，可以通过 broadcast.fail.percent 配置节点调用失败的比例，当达到这个比例后，BroadcastClusterInvoker 将不再调用其他节点，直接抛出异常。 broadcast.fail.percent 取值在 0～100 范围内。默认情况下当全部调用失败后，才会抛出异常。 broadcast.fail.percent 只是控制的当失败后是否继续调用其他节点，并不改变结果(任意一台报错则报错)。broadcast.fail.percent 参数 在 dubbo2.7.10 及以上版本生效

eg：

@reference(cluster = "broadcast", parameters = {"broadcast.fail.percent", "20"})

* 1. 服务降级
     1. 什么是服务雪崩

假设存在A、B、C、D四个系统，系统间存在如下调用链路：



在正常情况下系统之间调用快速且正常，系统运行平稳。但是此时用户访问系统A的流量激增，这些流量在瞬间透传到B、C、D三个系统。B、C系统服务器节点较多抗住了这些流量，但是D系统服务器节点较少，没有抗住这些流量，导致D系统的资源逐渐耗尽，只能提供慢服务，最终结果是响应用户时延很长。

D服务出现大量超时后，会导致其上游服务陆续阻塞住，从而耗尽线程资源变得不可用，最后整个调用链的服务都不可用，这种情况被称为服务雪崩

* + 1. 什么是服务降级

当某个非关键服务出现错误时，或者系统压力过大时，可以通过降级功能来临时屏蔽这个服务

降级分类：自动降级和人工降级

功能分类：读服务降级和写服务降级

常见场景：

1. 人工降级，对一些非核心服务进行人工降级，在大促之前通过降级开关主动关闭那些推荐内容、评价等对主流程没有影响的功能
2. 故障降级，比如调用的远程服务挂了，网络故障或者RPC 服务返回异常。那么可以直接降级
3. 限流降级，在秒杀这种流量比较集中并且流量特别大的情况下，因为突发访问量特别大可能会导致系统挂掉，这个时候可以采用限流来限制访问量。当达到阀值时，后续的请求被降级
   * 1. Dubbo的降级策略

dubbo提供了3种服务降级策略：

* 1. 强制降级
  2. 异常降级
  3. 自定义降级

配置方式有几种：

* 1. 在dubbo控制台中控制
  2. 在xml配置文件中配置，如下面的例子
  3. 使用java代码，判断满足降级条件后，动态写入注册中心，eg：

RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();

Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));

registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&mock=force:return+null"));

* + - 1. 强制降级
         1. 屏蔽服务

<dubbo:reference id="helloService" mock="force:return 1" interface="com.java.front.demo.provider.HelloService" />

<dubbo:reference id="helloService" mock="force:return null" interface="com.java.front.demo.provider.HelloService" />

表示消费方对该服务的方法调用都直接返回 1/null，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。

* + - * 1. 服务容错

<dubbo:reference id="helloService" mock="fail:return 1" interface="com.java.front.demo.provider.HelloService" />

<dubbo:reference id="helloService" mock="fail:return null" interface="com.java.front.demo.provider.HelloService" />

表示消费方对该服务的方法调用在失败后，再返回 1/null值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。

* + - 1. 异常降级

<dubbo:reference id="helloService" mock="throw com.java.front.BizException" interface="com.java.front.dubbo.demo.provider.HelloService" />

* + - 1. 自定义降级

package com.java.front.dubbo.demo.consumer;

import com.java.front.demo.provider.HelloService;  
  
public class HelloServiceMock implements HelloService {  
  
 @Override  
 public String sayHello(String name) throws Exception {  
 return "mock";  
 }  
}

<dubbo:reference id="helloService" mock="com.java.front.dubbo.demo.consumer.HelloServiceMock" interface="com.java.front.demo.provider.HelloService" />

* 1. 本地存根

使用dubbo后，消费者往往只需要调用接口，而接口的业务实现都放在provider端，有时候，我们也希望对消费者的接口进行增强，类似AOP，可以用来做 ThreadLocal 缓存，提前验证参数，调用失败后伪造容错数据等等

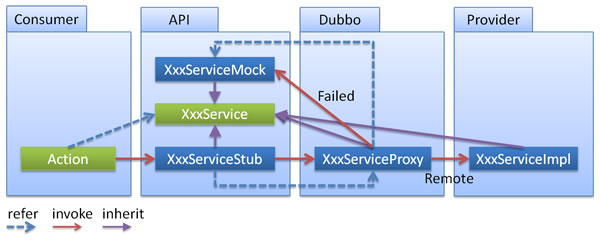
我们可以通过指定stub属性来实现本地存根



stub=”true”，默认会使用**当前包名+接口名+Stub**拼接出一个类名，然后该类也必须实现×××Service接口，当然也可以直接指定



调用关系如下，蓝色虚线是引用，红线是调用，紫线是继承



* 1. 异步调用

我们平常大部分都是使用 Dubbo 的同步调用，即调用 Dubbo 请求之后，调用线程将会阻塞，直到服务提供者返回结果。而Dubbo 异步调用就不会阻塞调用线程，那么在服务提供者返回结果这段时间，我们就可以执行其他业务逻辑

* + 1. RpcContext+修改xml

第一种方式是通过RpcContext实现异步调用

* 1. 我们需要增加一个 dubbo:method将指定方法配置成异步调用。

<dubbo:reference id="asyncService" interface="org.apache.dubbo.samples.governance.api.AsyncService">  
 <dubbo:method name="sayHello" async="true" />  
</dubbo:reference>

* 1. 服务引用xml配置完成之后，此时如果直接调用这个方法，将会立即返回 null，内部将会异步执行服务端调用逻辑

// 此调用会立即返回null  
String world = asyncService.sayHello("world");

* 1. 我们可以借助RpcContext异步获取provider方法的执行结果。。

CompletableFuture<String> helloFuture = RpcContext.*getContext*().getCompletableFuture();

* 1. CompletableFuture 是 JDK1.8 之后提供的异步任务增强类，我们可以直接调用其 get 方法(get会阻塞当前线程直到获取返回结果或服务超时，如果不想阻塞，可以添加回调)直接获取返回结果，也可以给他添加一个回调函数

// 为Future添加回调  
helloFuture.whenComplete((retValue, exception) -> {  
 if (exception == null) {  
 System.*out*.println(retValue);  
 } else {  
 exception.printStackTrace();  
 }  
});

* + 1. 直接通过RpcContext

第一种方式还需要修改Dubbo配置，如果不希望修改xml，我们可以直接借助RpcContext的asyncCall()方法直接完成异步调用

CompletableFuture<String> future = RpcContext.getContext().asyncCall(  
 () -> {  
 asyncService.sayHello("oneway call request1");  
 }  
 );  
future.get();

当然，这里的future对象依然是CompletableFuture对象，我们依然可以选择添加回调或者调用get()阻塞等待

* + 1. 服务提供者端定义

第三种方式不需要借助RpcContext，而是在provider端定义，其开发流程和普通dubbo接口一样，只不过返回值需要变一下

* 1. 定义Provider接口，**注意接口的返回类型是 CompletableFuture<String>**

public interface AsyncService {  
 CompletableFuture<String> sayHello(String name);  
}

* 1. 服务提供者Provider端需要使用 CompletableFuture 完成业务逻辑

public class AsyncServiceImpl implements AsyncService {  
 private static Logger *logger* = LoggerFactory.getLogger(AsyncServiceImpl.class);  
  
 @Override  
 public CompletableFuture<String> sayHello(String name) {  
 return CompletableFuture.supplyAsync(() -> {  
 try {  
 Thread.*sleep*(10000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return "async response from provider.";  
 });  
 }  
  
}

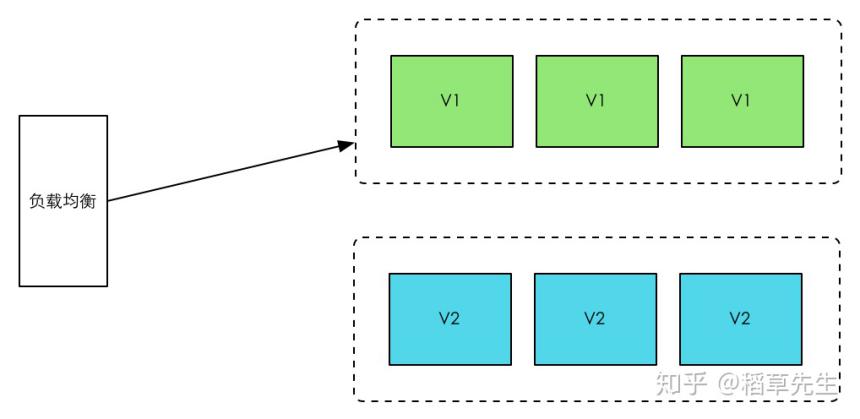
* 1. 消费端Consumer就不再需要借助RpcContext，可以直接调用

// 调用直接返回CompletableFuture  
CompletableFuture<String> future = asyncService.sayHello("async call request");  
// 增加回调  
future.whenComplete((v, t) -> {  
 if (t != null) {  
 t.printStackTrace();  
 } else {  
 System.out.println("Response: " + v);  
 }  
});  
// 早于结果输出  
System.out.println("Executed before response return.")

基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。

* 1. 蓝绿发布、滚动发布、灰度发布
     1. 蓝绿发布

所谓蓝绿部署，是指同时运行两个版本的应用，如上图所示，蓝绿部署的时候，并不停止掉老版本，而是直接部署一套新版本，等新版本运行起来后，再将流量切换到新版本上。但是蓝绿部署要求在升级过程中，同时运行两套程序，对硬件的要求就是日常所需的二倍，比如日常运行时，需要10台服务器支撑业务，那么使用蓝绿部署，你就需要购置二十台服务器。

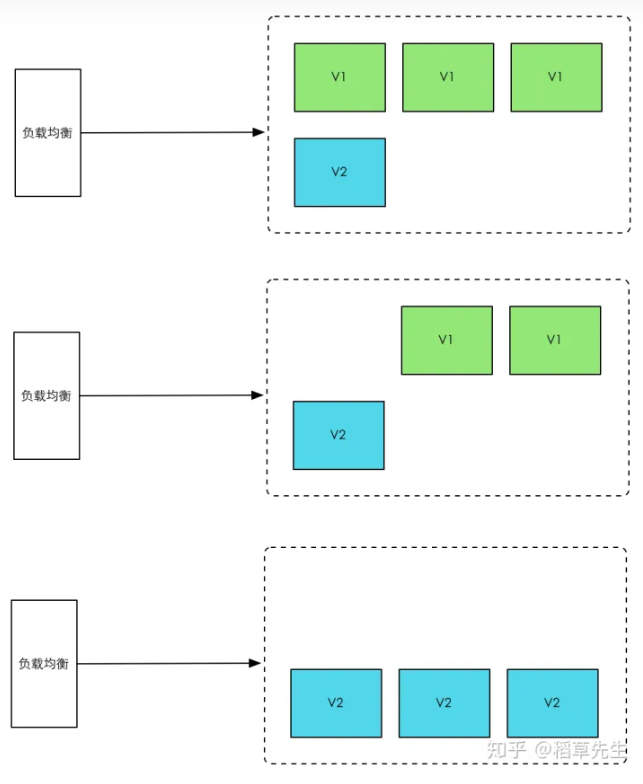


* + 1. 滚动发布

所谓滚动升级，就是在升级过程中，并不一下子启动所有新版本，是先启动一台新版本，再停止一台老版本，然后再启动一台新版本，再停止一台老版本，直到升级完成，这样的话，如果日常需要10台服务器，那么升级过程中也就只需要11台就行了。

但是滚动升级有一个问题，在开始滚动升级后，流量会直接流向已经启动起来的新版本，但是这个时候，新版本是不一定可用的，比如需要进一步的测试才能确认。那么在滚动升级期间，整个系统就处于非常不稳定的状态，如果发现了问题，也比较难以确定是新版本还是老版本造成的问题。

为了解决这个问题，我们需要为滚动升级实现流量控制能力。

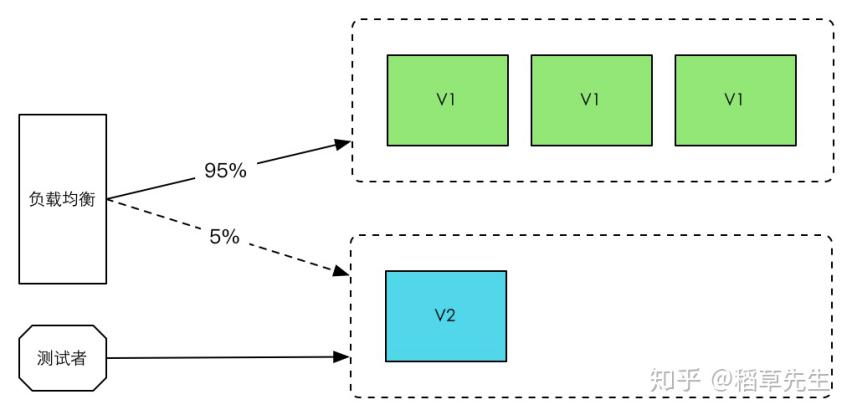


* + 1. 灰度发布

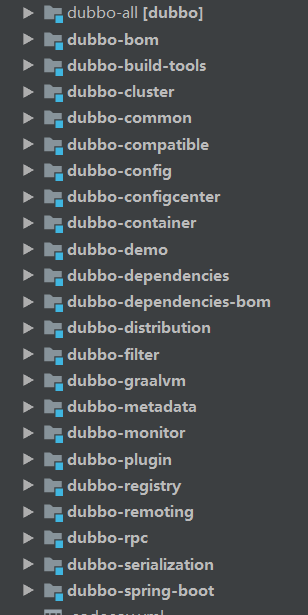
在灰度发布开始后，先启动一个新版本应用，但是并不直接将流量切过来，而是测试人员对新版本进行线上测试，启动的这个新版本应用，就是我们的金丝雀。如果没有问题，那么可以将少量的用户流量导入到新版本上，然后再对新版本做运行状态观察，收集各种运行时数据，如果此时对新旧版本做各种数据对比，就是所谓的A/B测试。

当确认新版本运行良好后，再逐步将更多的流量导入到新版本上，在此期间，还可以不断地调整新旧两个版本的运行的服务器副本数量，以使得新版本能够承受越来越大的流量压力。直到将100%的流量都切换到新版本上，最后关闭剩下的老版本服务，完成灰度发布。

如果在灰度发布过程中（灰度期）发现了新版本有问题，就应该立即将流量切回老版本上，这样，就会将负面影响控制在最小范围内。



1. Dubbo详解
   1. dubbo 的文件结构



**Dubbo**的主要模块及功能如下：

**dubbo-cluster：**dubbo负责管理集群调用的模块。提供了负载均衡、容错、路由等一系列功能

**dubbo-common：**dubbo的公共模块，有很多公共类和工具实现，比如url等

**dubbo-config：**负责解析dubbo配置的模块。dubbo提供了四种配置方式，包括XML配置、属性配置、API配置、注解配置，配置模块就是实现了这四种配置的功能

**dubbo-configcenter：** dubbo动态配置模块，主要是负责接入动态配置中心

**dubbo-metadata：** dubbo元数据模块

**dubbo-monitor：**dubbo监控模块，主要负责统计服务调用次数，调用时间以及实时调用链跟踪

**dubbo-register：**dubbo负责与多种开源注册中心进行交互的模块，提供注册中心能力

**dubbo-remoting：**dubbo远程通信模块

**dubbo-rpc：**dubbo远程调用协议的抽象模块，包括rmi，hession，dubbo，http，redis等

**dubbo-serialization：**该模块中封装了各类序列化框架的支持实现

* 1. 实现一个RPC框架需要哪些东西

RPC框架主要是面向服务提供者和服务消费者的，我们分别从这两个角度来考虑

1. 对于服务消费者
2. **注册中心：**消费者需要知道有哪些接口可以被调用，以及到哪台服务器上去调用，因此需要一个注册中心，相当于服务目录。推荐使用Zookeeper，当然Nacos和redis也是可以的
3. **代理类：**想要调用这些接口，消费者需要一个代理类来屏蔽底层调用细节
4. **负载均衡策略：**一般一个服务是有多个服务提供者的，因此RPC框架还需要支持多种负载均衡策略
5. **容错机制：**网络调用是不可靠的，当rpc调用失败时，rpc框架需要提供对应的容错方案
6. **网络传输：**既然要调用远程的方法，那就要发送网络请求来传递目标类和方法的信息以及方法的参数等数据到服务提供者处，这里可以直接使用socket，也可以使用NIO，当然最好的选择还是Netty
7. **序列化和反序列化：**因为网络传输的数据必须是二进制的，而 Java 对象没办法直接在网络中传输，因此需要将其序列化为二进制的数据。
8. 对于服务提供者
9. **服务暴露：**provider需要向注册中心注册自己，暴露自己所能提供的服务
10. 反序列化完的请求应该扔到线程池里面做处理，某个线程接受到这个请求之后找到对应的实现调用，然后再将结果原路返回

除此之外还需要:

1. **监控中心：**面对众多的服务，精细化的监控和方便的运维是必不可少的

典型的RPC调用，一般分为以下几个步骤：

（1）服务提供者Provider启动后主动向服务注册中心注册机器ip、端口以及提供的服务列表；

（2）服务消费者Consumer启动时向服务注册中心获取服务提供方地址列表，在本地缓存一份；

（3）服务消费者Consumer通过本地调用的方式调用服务，调用模块收到请求后通过负载均衡策略选取合适的远程服务地址；

（4）协议模块负责将方法、入参等信息序列化(编码)成能够进行网络传输的消息体，并将消息通过网络发送给服务端；

（5）服务端Provider收到消息后进行解码(反序列化操作)。

（6）根据解码结果调用本地的服务进行相关处理；

（7）服务端Provider将处理返回的结果进行序列化（编码），并将结果通过网络发送至服务消费者Consumer；

（8）服务消费者Consumer收到消息后进行解码最终得到结果；

不同的 RPC 框架实现中步骤 1、2、3的顺序可能有些不同，但总体来说都是差不多的

* 1. dubbo中的url

Dubbo采用 URL 的方式来作为约定的参数类型，被称为公共契约。Dubbo 用 URL 作为配置总线，贯穿整个体系，几乎所有的模块之间的交互都是通过url进行的

我们知道，一个rpc框架的调用信息中至少包含以下信息：

1. 调用的是哪个类或接口
2. 调用的是哪个方法，方法名和方法参数类型（考虑方法重载）
3. 调用方法的入参

而dubbo的URL 具体的参数如下：

1. **protocol：**指的是 dubbo 中的各种协议，如：dubbo thrift http
2. **username/password**：用户名/密码
3. **host/port：**主机/端口
4. **path：**接口的名称
5. **parameters**：参数键值对

eg：

dubbo:*//192.168.1.6:20880/moe.cnkirito.sample.HelloService?timeout=3000*

描述一个 dubbo 协议的服务

zookeeper:*//127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=demo-consumer&dubbo=2.0.2&interface=org.apache.dubbo.registry.RegistryService&pid=1214&qos.port=33333&timestamp=1545721981946*

描述一个 zookeeper 注册中心

consumer:*//30.5.120.217/org.apache.dubbo.demo.DemoService?application=demo-consumer&category=consumers&check=false&dubbo=2.0.2&interface=org.apache.dubbo.demo.DemoService&methods=sayHello&pid=1209&qos.port=33333&side=consumer&timestamp=1545721827784*

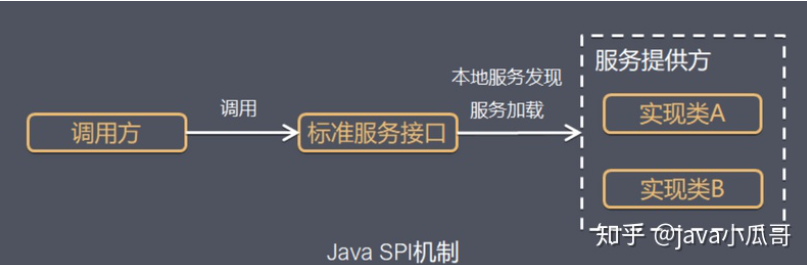
描述一个消费者

* 1. dubbo的SPI服务发现机制
     1. jdk的SPI机制
        1. 原理

SPI的全名为Service Provider Interface，面向对象的设计里面，模块之间基于接口编程，而不是对实现类进行硬编码，这样做也是为了模块设计的可拔插原则。为了在模块装配的时候不在程序里指明是哪个实现，就需要一种服务发现的机制，jdk的spi就是为某个接口寻找服务实现。

jdk提供了服务实现查找的工具类：java.util.ServiceLoader，它会去加载META-INF/service/目录下的配置文件。

相比于接口interface，SPI主要是提供给框架使用，实现框架扩展和组件替换，而interface主要作用于类，使类具有多态的特性



当服务的提供者提供了一种接口的实现之后，需要在classpath下的META-INF/services/目录里创建一个以服务接口命名的文件，这个文件里的内容就是这个接口的具体的实现类名。当其他的程序需要这个服务的时候，就可以通过查找这个jar包（一般都是以jar包做依赖）的META-INF/services/中的配置文件，配置文件中有接口的具体实现类名，可以根据这个类名进行加载实例化，就可以使用该服务了。

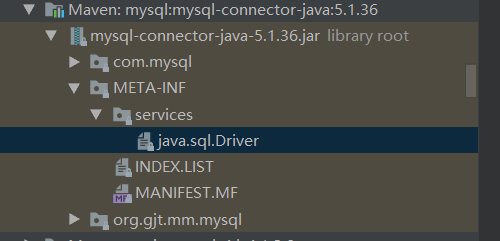
**一个SPI接口对应一个扩展点，一个扩展点可以有多个实现**

比如JDBC:

1.首先在Java中定义了接口java.sql.Driver，并没有具体的实现，具体的实现都是由不同厂商提供。

2.在MySQL的jar包mysql-connector-java-6.0.6.jar中，可以找到META-INF/services目录，该目录下会有一个名字为java.sql.Driver的文件，文件内容是com.mysql.cj.jdbc.Driver，这里面的内容就是针对Java中定义的接口的实现。

3.同样在PostgreSQL的jar包PostgreSQL-42.0.0.jar中，也可以找到同样的配置文件，文件内容是org.postgresql.Driver，这是PostgreSQL对Java的java.sql.Driver的实现。





* + - 1. 示例实现

为了模拟一个完整的jdk SPI发现过程，我们一共需要创建4个项目，这里以jdbc为例进行模拟。

com.test.jdbc 定义SPI的接口

com.test.mysql mysql实现

com.test.orcale oracle实现

com.test.demo 使用jdbc的任一实现模拟访问数据库

* + - * 1. 创建项目

我们首先创建com.test.jdbc项目，然后将其mvn install发布到maven仓库中

然后创建com.test.mysql和com.test.orcale，这pom中引入com.test.jdbc。这里需要注意，在pom.xml文件中还需要加上：

<build>

<resources>  
 <resource>  
 <directory>src/main/resources</directory>  
 <targetPath>META-INF/</targetPath>  
 </resource>  
 </resources>  
</build>

这是为了保证resources目录下的内容在打jar包时，会被复制到META-INF目录中

完整的pom如下：

<?xml version="1.0" encoding="UTF-8"?>  
  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.test</groupId>  
 <artifactId>mysql</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <dependencies>  
 <dependency>  
 <groupId>com.test</groupId>  
 <artifactId>jdbc</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <resources>  
 <resource>  
 <directory>src/main/resources</directory>  
 <targetPath>META-INF/</targetPath>  
 </resource>  
 </resources>  
 </build>  
</project>

接着创建com.test.demo，引入com.test.jdbc和com.test.oracle

* + - * 1. 编写com.test.jdbc代码

在com.test.jdbc中，我们使用两个类

package com.test.jdbc;  
  
*/\*\*  
 \* Desc :模拟jdbc驱动  
 \*/*public interface JdbcDriver {  
 //连接数据库  
 boolean connectDataBase();  
 //执行sql  
 boolean exec(String sql);  
}

package com.test.jdbc;  
  
import java.util.ServiceLoader;  
  
*/\*\*  
 \* Desc :服务发现过程，供com.test.demo使用  
 \*/*public class Sql {  
 public void execSql(){  
 ServiceLoader<JdbcDriver> jdbcDriverLoader = ServiceLoader.*load*(JdbcDriver.class);  
 for (JdbcDriver jDriver:jdbcDriverLoader) {  
 jDriver.connectDataBase();  
 jDriver.exec("select \* from table");  
 }  
 }  
}

* + - * 1. 编写实现jar代码

package com.test.mysql;  
  
import com.test.jdbc.JdbcDriver;  
  
*/\*\*  
 \* Desc :mysql驱动实现  
 \*/*public class MysqlDriver implements JdbcDriver {  
  
 @Override  
 public boolean connectDataBase() {  
 System.*out*.println("连接mysql成功");  
 return true;  
 }  
  
 @Override  
 public boolean exec(String sql) {  
 System.*out*.println("mysql执行了" + sql);  
 return true;  
 }  
}

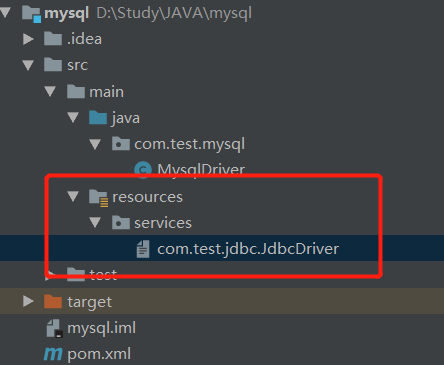
oracle的就省略了

* + - * 1. 在resources/services下创建服务发现配置文件

因为前面我们加了pom配置，所以这里我们只需要把配置文件放到resource目录下，打包时会将resources的文件自动复制到META-INF/下

这里需要注意，配置文件名必须是SPI接口全限定名,这里是com.test.jdbc.JdbcDriver,里面的内容必须是其对应的服务实现类的全限定名，这里是com.test.mysql.MysqlDriver

配置文件必须放在services目录下，如图：

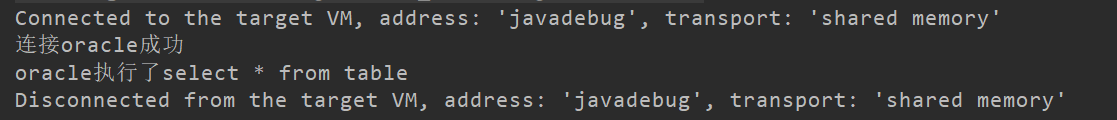


* + - * 1. 使用

com.test.demo必须同时引入SPI接口项目com.test.jdbc和SPI实现项目com.test.mysql或com.test.oracle

public class App {  
 public static void main( String[] args ) {  
 Sql sql = new Sql();  
 sql.execSql();  
 }  
}

执行结果：



* + 1. dubbo的SPI机制

JDK标准的SPI只能通过遍历来查找扩展点和实例化，有可能导致一次性加载所有的扩展点，如果不是所有的扩展点都被用到，就会导致资源的浪费。例如com.alibaba.dubbo.rpc.Protocol接口有InjvmProtocol、DubboProtocol、RmiProtocol、HttpProtocol、HessianProtocol等实现，如果只是用到其中一个实现，却要加载全部实现，会导致资源的浪费。

dubbo把配置文件中扩展实现的格式修改为键值对格式，例如META-INF/dubbo/com.xxx.Protocol里的com.foo.XxxProtocol格式改为了xxx = com.foo.XxxProtocol这种以键值对的形式，这样做的目的是为了让我们更容易的定位到问题：比如由于第三方库不存在，无法初始化，导致无法加载扩展名（“A”），当用户配置使用A时，dubbo就会报无法加载扩展名的错误，而不是报哪些扩展名的实现加载失败以及错误原因，这是因为原来的配置格式没有记录扩展名的id，导致dubbo无法抛出较为精准的异常，这会加大排查问题的难度。所以改成key-value的形式来进行配置。

dubbo的SPI机制增加了对IOC、AOP的支持，一个扩展点可以直接通过setter注入到其他扩展点

总结：

**1.JDK 标准的 SPI 会一次性实例化一个扩展点(一个spi接口对应一个扩展点)所有实现，如果有扩展实现初始化很耗时，但如果没用上也会加载，会很浪费资源，dubbo实现了扩展类的延迟加载。**

**2.增加了对扩展点 IoC 和 AOP 的支持，一个扩展点可以直接 setter 注入其它扩展点。**

**3.扩展点文件中的配置方式改为键值对方式，提高异常精度，减少排查问题的难度**

* + - 1. @SPI、@Adaptive、@Activate注解

首先我们了解几个概念

**1.扩展点(Extension Point)**：是一个Java的接口

**2.扩展名(Extension name)**：配置文件中键值对的key

**2.扩展(Extension)：**扩展点的实现类。一个扩展点可能对应多个扩展

**3.自适应扩展实例(Extension Adaptive Instance)**：在调用扩展点的接口方法时，会根据实际的参数来决定要使用哪个扩展类。通过@Adaptive注解指定

* + - * 1. @SPI注解

注解在接口上。标识了接口是一个扩展点，属性 value 用来指定扩展点的键值对的key名称，这个，eg

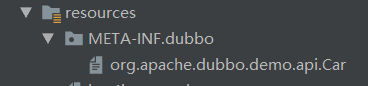
@SPI("Mercedes")  
public interface Car {  
 //汽车发动  
 void run();  
}

public class MercedesCar implements Car{  
  
 @Override  
 public void run() {  
 System.*out*.println("奔驰车发动");  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 ExtensionLoader<Car> loader = ExtensionLoader.*getExtensionLoader*(Car.class);

//获取默认扩展类的实例对象  
 Car car = loader.getDefaultExtension();  
 car.run();  
 }  
}

在resources目录下创建对应SPI配置文件



Mercedes=org.apache.dubbo.demo.api.MercedesCar  
BMW=org.apache.dubbo.demo.api.BMWCar

运行结果：



* + - * 1. @Adaptive注解

通过@SPI只能加载@SPI注解value值默认指定的扩展类，如果希望在运行时根据传入的实际参数来决定使用哪种扩展，就需要@Adaptive注解配合使用

自适应扩展点，运行时根据实际的参数来决定使用哪一种扩展，注解在class和方法上。

有时，我们往往不希望在框架启动阶段加载对应的扩展类，而是希望在运行过程中，通过传入的参数(往往是外部传入)来决定加载哪一种扩展类，那么这就产生了矛盾，拓展类未被加载，那么拓展类的方法就无法被调用（静态方法除外）。拓展方法未被调用，拓展就无法被加载

dubbo提供了@Adaptive注解帮助我们解决运行时根据参数自适应选择扩展类的问题，该注解一般使用在方法上，很少用在类上:

1. **当用在方法上时：**Dubbo会为该接口自动生成一个$子类（代理类），并且按照一定的格式重写该方法，而其余没有标注@Adaptive注解的方法将会默认抛出异常

**注意：被@Adaptive注解标记的方法必须携带Url参数（可以不直接作方法参数，即可以包裹在pojo对象里，只要pojo对象有URL类型的属性就会被检测到）**

1. **当用在类上时：**在代表这个类是人工实现的（Dubbo 不会为该类生成代理类），当它被注释到类上，意味着扩展的加载逻辑由人工编码完成，不需要框架为其生成$子类（代理类）。另外需要注意的是，如果标记在类上，则使用该类作为扩展点的自适应扩展实例(也就是说不需要URL参数，可以查看源码的[ExtensionLoader#loadClass](#loadClass方法)方法)。dubbo中只有两个类是这样的：AdaptiveCompiler、AdaptiveExtensionFactory等。另外，@Adaptive注解不能同时标记同一扩展点的两个不同的扩展实现类，否则加载时会抛异常

下面我们重点介绍当@Adaptive被用在方法上的场景：

@Adaptive注解分为有参数和无参数两种使用方式：

**1.有参数的@Adaptive注解**：@Adaptive注解可以接收一个String[] value的数组，数组中的字符串是key值，代码中要根据key值来获取对应的Value值，进而加载相应的extension实例。比如new String[]{“key1”,”key2”}，表示会先在URL中寻找key1的值，如果找到则使用此值加载extension，如果key1没有，则寻找key2的值，如果key2也没有，则使用@SPI注解的默认值，如果SPI注解没有默认值，则报错

@SPI  
public interface MyCar {  
 //汽车发动  
 @Adaptive({"A\_name","B\_name","name"})  
 void run(URL url);  
  
 //汽车开门  
 void openDoor(URL url);  
}

public static void main(String[] args) {  
 ExtensionLoader<MyCar> loader = ExtensionLoader.*getExtensionLoader*(MyCar.class);  
 MyCar car = loader.getAdaptiveExtension();  
 */\*\*  
 \* 有参数使用方式  
 \*/* //有符合@Adaptive的参数，B\_name优先级高于name，因此是宝马车发动  
 car.run(URL.*valueOf*("http://localhost:9999/xxx?name=Mercedes&B\_name=BMW"));  
 //没有符合@Adaptive的参数，使用@SPI注解指定的默认值，因为@SPI注解没有指定默认值，在这里会报错  
 car.run(URL.*valueOf*("http://localhost:9999/xxx"));

//调用没有被@Adaptive标记的方法会抛出异常

car.openDoor(url);  
}

结果：



**2.无参的@Adaptive注解**：以’.’分隔，例如org.apache.dubbo.xxx.YyyInvokerWrapper接口名会变成yyy.invoker.wrapper，然后以此名称作为key到URL寻找，如果仍没有找到，使用@SPI注解默认值，如果仍然没有则抛出IllegalStateException异常。

eg:

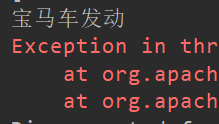
@SPI("Mercedes")  
public interface MyCar {  
 //汽车发动  
 @Adaptive  
 void run(URL url);  
  
 //汽车开门  
 void openDoor(URL url);  
}

public static void main(String[] args) {  
 ExtensionLoader<MyCar> loader = ExtensionLoader.*getExtensionLoader*(MyCar.class);  
 MyCar car = loader.getAdaptiveExtension();  
 */\*\*  
 \* 无参数使用方式  
 \*/* //没有符合@Adaptive的参数，将扩展点接口名按单词拆分，用.分隔得到扩展点key，根据key去查找URL参数列表  
 URL url = URL.*valueOf*("http://localhost:9999/xxx?my.car=BMW");

//没有符合@Adaptive的参数，将扩展点接口名按单词拆分，用.分隔得到扩展点key，根据key去查找URL参数列表，如果URL中没有找到对应参数，则使用默认值

URL url = URL.*valueOf*("http://localhost:9999/xxx);  
 car.run(url);  
 //调用没有被@Adaptive标记的方法会抛出异常  
 car.openDoor(url);  
}

运行结果



**需要注意的是，URL不一定直接作为方法参数，也可以包裹在pojo对象中，然后将pojo对象作为参数传递，但是对应的pojo类型必须提供符合要求的返回url的方法**

这里直接贴上源码里对于**返回URL方法**的要求吧:

**（1）方法以get开头，且名字长度大于3**

**（2）必须是public方法**

**（3）不能是静态方法**

**（4）方法必须是无参的**

**（5）方法的返回值必须是URL类型**

if ((name.startsWith("get") || name.length() > 3)  
 && Modifier.*isPublic*(m.getModifiers())  
 && !Modifier.*isStatic*(m.getModifiers())  
 && m.getParameterTypes().length == 0  
 && m.getReturnType() == URL.class)

了解了源码的要求，我们甚至可以做一些骚操作，如：

@SPI("Mercedes")  
public interface MyCar {  
  **//汽车发动(URL包裹在Ssr对象中)**  
 @Adaptive  
 void run(Ssr surl);  
  
 //汽车开门  
 void openDoor(URL url);  
}

public class Ssr {  
 */\*\*  
 \* 我是不是很骚？**\*/* public URL get我就是乱写的方法名() {  
 return URL.*valueOf*("http://localhost:9999/xxx?my.car=Mercedes");  
 }  
}

另外还要注意的是，不论是**@Adaptive注解**有没有加参数，只要是最终生成的extenssion的key值为protocol，dubbo spi都会特殊处理，让我们看一下源码：

if (!"protocol".equals(value[i])) {  
 if (hasInvocation) {  
 getNameCode = String.*format*("url.getMethodParameter(methodName, \"%s\", \"%s\")", value[i], defaultExtName);  
 } else {  
 getNameCode = String.*format*("url.getParameter(\"%s\", \"%s\")", value[i], defaultExtName);  
 }  
} else {  
 getNameCode = String.*format*("( url.getProtocol() == null ? \"%s\" : url.getProtocol() )", defaultExtName);  
}

因此对于Protocol扩展点接口（或者通过其他方式导致key最后为protocol的扩展点接口）的SPI自适应扩展，都是通过根据URL的网络协议来的，而其他接口则是根据URL路径中的parameter来的，如：

@SPI("Mercedes")  
public interface MyCar {  
 //通过adaptive注解加参数的方式使得扩展点的key名为protocol  
 @Adaptive("protocol")  
 void run(Ssr url);  
  
 //汽车开门  
 void openDoor(URL url);  
}

* + - * 1. getAdaptiveExtension()

看完getExtension()，接下来看一下getAdaptiveExtension()，该方法用于获取自适应扩展实现对象，主要实现[@Adaptive注解](#Adaptive注解)的功能

getAdaptiveExtension()

-> createAdaptiveExtension() #如果无缓存则创建

-> getAdaptiveExtensionClass().newInstance() #获取AdaptiveExtensionClass

-> getExtensionClasses() # 加载当前扩展所有实现，看是否有实现被标注为@Adaptive

-> createAdaptiveExtensionClass() #如果没有实现被标注为@Adaptive，则动态创建一个Adaptive实现类

-> new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate() #动态生成实现类java代码

-> compiler.compile(code, classLoader) #动态编译java代码，加载类并实例化

-> injectExtension(instance)

public T getAdaptiveExtension() {

**// 首先判断是否已经有缓存的自适应扩展实例对象**  
 Object instance = cachedAdaptiveInstance.get();  
 if (instance == null) {  
 if (createAdaptiveInstanceError != null) {  
 throw new IllegalStateException("Failed to create adaptive instance: " +  
 createAdaptiveInstanceError.toString(),  
 createAdaptiveInstanceError);  
 }  
 **// 双重检查防止并发**  
 synchronized (cachedAdaptiveInstance) {  
 instance = cachedAdaptiveInstance.get();  
 if (instance == null) {  
 try {

**//重**要**，没有缓存的自适应扩展实例对象，创建新的AdaptiveExtension实例**  
 instance = createAdaptiveExtension();  
 cachedAdaptiveInstance.set(instance);  
 } catch (Throwable t) {  
 createAdaptiveInstanceError = t;  
 throw new IllegalStateException("Failed to create adaptive instance: " + t.toString(), t);  
 }  
 }  
 }  
 }  
 return (T) instance;  
}

createAdaptiveExtension()

该方法关系到@Adaptive注解的实现机制，源码如下:

private T createAdaptiveExtension() {  
 try {

**//重**要**， 使用getAdaptiveExtensionClass().newInstance()创建自适应扩展实例对象，然后用**[**injectExtension()**](#dubbo的ioc源码实现)**为其注入依赖**  
 return injectExtension((T) getAdaptiveExtensionClass().newInstance());  
 } catch (Exception e) {  
 throw new IllegalStateException("Can't create adaptive extension " + type + ", cause: " + e.getMessage(), e);  
 }  
}

private Class<?> getAdaptiveExtensionClass() {

**//重**要**，加载当前Extension的所有实现,如果有@Adaptive标记在该扩展点的某个扩展类上，则会将该类的class赋值为cachedAdaptiveClass属性**

getExtensionClasses();  
 if (cachedAdaptiveClass != null) {  
 return cachedAdaptiveClass;  
 }

**//重**要**，如果没有找到@Adaptive类型实现，即该扩展点没有@Adaptive注解标记在扩展类上，则动态生成代码，创建一个自适应扩展类的class对象，并将其放到cachedAdaptiveClass中缓存起来**  
 return cachedAdaptiveClass = createAdaptiveExtensionClass();  
}

getExtensionClasses()

private Map<String, Class<?>> getExtensionClasses() {

**//cachedClasses的key是扩展名，value是扩展类型的class对象**  
 Map<String, Class<?>> classes = cachedClasses.get();  
 if (classes == null) {  
 synchronized (cachedClasses) {  
 classes = cachedClasses.get();  
 if (classes == null) {

**//重**要**，如果缓存cachedClasses为空，则加载cachedClasses**  
 classes = loadExtensionClasses();  
 cachedClasses.set(classes);  
 }  
 }  
 }  
 return classes;  
}

private Map<String, Class<?>> loadExtensionClasses() {

**//重**要**，主要是获取@SPI注解中指定的默认扩展名，其实就是给**cachedDefaultName**赋值，源码参见**[**cacheDefaultExtensionName**](#cacheDefaultExtensionName)  
 cacheDefaultExtensionName();  
  
 Map<String, Class<?>> extensionClasses = new HashMap<>();  
 **//重**要**，加载指定文件夹下的配置文件**  
 loadDirectory(extensionClasses, *DUBBO\_INTERNAL\_DIRECTORY*, type.getName(), true);  
 loadDirectory(extensionClasses, *DUBBO\_INTERNAL\_DIRECTORY*, type.getName().replace("org.apache", "com.alibaba"), true);  
 loadDirectory(extensionClasses, *DUBBO\_DIRECTORY*, type.getName());  
 loadDirectory(extensionClasses, *DUBBO\_DIRECTORY*, type.getName().replace("org.apache", "com.alibaba"));  
 loadDirectory(extensionClasses, *SERVICES\_DIRECTORY*, type.getName());  
 loadDirectory(extensionClasses, *SERVICES\_DIRECTORY*, type.getName().replace("org.apache", "com.alibaba"));  
 return extensionClasses;  
}

loadClass()

loadDirectory()方法会调用loadResource ()，而loadResource()会调用loadClass()，下面我们来看一下loadClass()

private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL resourceURL, Class<?> clazz, String name) throws NoSuchMethodException {  
 if (!type.isAssignableFrom(clazz)) {  
 throw new IllegalStateException("Error occurred when loading extension class (interface: " +  
 type + ", class line: " + clazz.getName() + "), class "  
 + clazz.getName() + " is not subtype of interface.");  
 }

**//检测目标类上是否有 Adaptive 注解，在dubbo中只有AdaptiveCompiler、AdaptiveExtensionFactory两个类上携带了@Adaptive注解**  
 if (clazz.isAnnotationPresent(Adaptive.class)) {

**//重**要**，如果当前类被@Adaptive注解标记，则将该类的class对象缓存到cachedAdaptiveClass中去，如果cachedAdaptiveClass非空，并且不等于当前Class对象，则抛出异常，因此@Adaptive不能标记同一扩展点的多个扩展类，否则报错**  
 cacheAdaptiveClass(clazz);  
 } else if (isWrapperClass(clazz)) {

**//重**要**，判断当前class是否为wrapper类，即看该类有没有拷贝构造函数**

**//如果有，那么就需要把该class缓存到cachedWrapperClasses中去，等dubbo的aop执行到的时候，会进行自动包装**  
 cacheWrapperClass(clazz);  
 } else {**//处理扩展实现类**

**// 扩展实现类必须有无参构造函数，否则抛出NoSuchMethod异常**  
 clazz.getConstructor();  
 if (StringUtils.*isEmpty*(name)) {

**//如果name为空，SPI中配置的xxx = com.foo.XxxProtocol，xxx没有填写，那么：**

**1.检查扩展类是否有@Extension注解，如果有的话，以@Extension注解的value值作为扩展名**

**2.如果没有注解，则使用小写的类名(去掉扩展点接口名)作为 name，如DubboProtocl的name为dubbo**  
 name = findAnnotationName(clazz);  
 if (name.length() == 0) {  
 throw new IllegalStateException("No such extension name for the class " + clazz.getName() + " in the config " + resourceURL);  
 }  
 }  
 **//重**要**，如果name以,逗号分割的话，那么就说明有很多个**  
 String[] names = *NAME\_SEPARATOR*.split(name);  
 if (ArrayUtils.*isNotEmpty*(names)) {

**//重**要**，将包含@Activate注解的实现类缓存到*cachedActivates*集合中**  
 cacheActivateClass(clazz, names[0]);  
 for (String n : names) {

**// 在cachedNames集合中缓存实现类->扩展名的映射**  
 cacheName(clazz, n);

**// 在cachedClasses集合中缓存扩展名->实现类的映射，**  
 saveInExtensionClass(extensionClasses, clazz, n);  
 }  
 }  
 }  
}

createAdaptiveExtensionClass()

private Class<?> createAdaptiveExtensionClass() {

**//重**要**，为扩展点接口自动生成实现类代码字符串**  
 String code = new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate();  
 ClassLoader classLoader = *findClassLoader*();

**//获取dubbo编译器，AdaptiveCompiler是dubbo中被@Adaptvie标记的两个类中的一个**  
 org.apache.dubbo.common.compiler.Compiler compiler = ExtensionLoader.*getExtensionLoader*(org.apache.dubbo.common.compiler.Compiler.class).getAdaptiveExtension();

**//编译代码生成Class对象**  
 return compiler.compile(code, classLoader);  
}

$子类代码

动态生成后的代码如下：这里以Protocol#export()自适应扩展点为例：

public class Protocol$Adaptive implements Protocol {  
 public void destroy() {  
 throw new UnsupportedOperationException("The method public abstract void Protocol.destroy() of interface Protocol is not adaptive method!");  
 }  
  
 public int getDefaultPort() {  
 throw new UnsupportedOperationException("The method public abstract int Protocol.getDefaultPort() of interface Protocol is not adaptive method!");  
 }  
  
 public Exporter export(Invoker arg0) throws RpcException {  
 if (arg0 == null) throw new IllegalArgumentException("Invoker argument == null");  
 if (arg0.getUrl() == null)  
 throw new IllegalArgumentException("Invoker argument getUrl() == null");  
 URL url = arg0.getUrl();  
 String extName = (url.getProtocol() == null ? "dubbo" : url.getProtocol());  
 if (extName == null)  
 throw new IllegalStateException("Failed to get extension (Protocol) name from url (" + url.toString() + ") use keys([protocol])");

**//重**要**，可以看出，getAdaptiveExtension()其实就是从url中获取到extName，然后再调用getExtension()方法**  
 Protocol extension = (Protocol) ExtensionLoader.*getExtensionLoader*(Protocol.class).getExtension(extName);  
 return extension.export(arg0);  
 }  
  
 public java.util.List getServers() {  
 throw new UnsupportedOperationException("The method public default java.util.List Protocol.getServers() of interface Protocol is not adaptive method!");  
 }  
  
 public Invoker refer(java.lang.Class arg0, URL arg1) throws RpcException {  
 if (arg1 == null) throw new IllegalArgumentException("url == null");  
 URL url = arg1;  
 String extName = (url.getProtocol() == null ? "dubbo" : url.getProtocol());  
 if (extName == null)  
 throw new IllegalStateException("Failed to get extension (Protocol) name from url (" + url.toString() + ") use keys([protocol])");  
 Protocol extension = (Protocol) ExtensionLoader.*getExtensionLoader*(Protocol.class).getExtension(extName);  
 return extension.refer(arg0, arg1);  
 }  
}

你可能会有些疑惑,为什么extName取的是**url.getProtocol()**？这里请注意，其实是dubbo spi为Protocol扩展点开了小灶，这里可以看一下**AdaptiveClassCodeGenerator#generate()=》generateMethod()=》generateMethodContent()=》generateExtNameAssignment()**

源码如下：

private String generateExtNameAssignment(String[] value, boolean hasInvocation) {  
 // *TODO: refactor it* String getNameCode = null;  
 for (int i = value.length - 1; i >= 0; --i) {  
 if (i == value.length - 1) {  
 if (null != defaultExtName) {

if (!"protocol".equals(value[i])) {

**//重**要**，如果扩展名的key不是protocol，调用url.getParameter()获取扩展名**  
 if (hasInvocation) {  
 getNameCode = String.format("url.getMethodParameter(methodName, \"%s\", \"%s\")", value[i], defaultExtName);  
 } else {  
 getNameCode = String.format("url.getParameter(\"%s\", \"%s\")", value[i], defaultExtName);  
 }  
 } else {

**//重**要**，如果扩展名的key是protocol，调用url.getProtocol()获取扩展名**  
 getNameCode = String.format("( url.getProtocol() == null ? \"%s\" : url.getProtocol() )", defaultExtName);  
 }  
 } else {  
 if (!"protocol".equals(value[i])) {  
 if (hasInvocation) {  
 getNameCode = String.format("url.getMethodParameter(methodName, \"%s\", \"%s\")", value[i], defaultExtName);  
 } else {  
 getNameCode = String.*format*("url.getParameter(\"%s\")", value[i]);  
 }  
 } else {  
 getNameCode = "url.getProtocol()";  
 }  
 }  
 } else {  
 if (!"protocol".equals(value[i])) {  
 if (hasInvocation) {  
 getNameCode = String.format("url.getMethodParameter(methodName, \"%s\", \"%s\")", value[i], defaultExtName);  
 } else {  
 getNameCode = String.*format*("url.getParameter(\"%s\", %s)", value[i], getNameCode);  
 }  
 } else {  
 getNameCode = String.*format*("url.getProtocol() == null ? (%s) : url.getProtocol()", getNameCode);  
 }  
 }  
 }  
 return String.format(CODE\_EXT\_NAME\_ASSIGNMENT, getNameCode);  
}

由此可见dubbo对于扩展名的key为protocol的扩展，都会取url.getProtocol()而不是url.getParameter()

* + - * 1. @Activate注解

扩展点自动激活，主要使用在有多个扩展点实现、需要同时根据不同条件被激活的场景中。注解在class和方法上。@Adaptive每次只能返回一个扩展点实现类，如果希望返回多个扩展点实现类，可以使用@Activate注解。说白了就是根据指定的条件过滤出我们想要的一个或多个扩展实例

@Documented  
@Retention(RetentionPolicy.*RUNTIME*)  
@Target({ElementType.*TYPE*, ElementType.*METHOD*})  
public @interface Activate {  
 */\*\*  
 \* 分组(筛选条件)，如果getActivateExtension()的入参值是注解的group成员之一，则getActivateExtension()生成的list中就会有@Activate标记的扩展类  
 \*/* String[] group() default {};  
 */\*\*  
 \* value值(筛选条件)，如当url为https://localhost:20880/path?ext=name1,name2,-default,-name3"，且key为"ext"的时候，则key所对应的value值为name1,name2,-default,-name3*

*\*需要注意的是，如果url中key指定的数组的元素值中包含了”-default”字符串的话，****表示该扩展点下的所有默认的@Activate类都不进行激活，只激活指定的****。若value数组的元素值中包含了”-name3”的值，则表示value值为name3的@Activate注解所标记的类也不进行激活。*

*\* 如果同时出现group和value，先用group筛一遍，然后对group后的结果集用value再筛一遍  
 \*/* String[] value() default {};  
 */\*\*  
 \* 填写扩展点列表，表示哪些扩展点要在本扩展点之前激活  
 \*/* @Deprecated  
 String[] before() default {};  
 */\*\*  
 \* 表示哪些扩展点需要在本扩展点之后激活  
 \*/* @Deprecated  
 String[] after() default {};  
 */\*\*  
 \* 生成的list结果集中的各扩展类，按order大小升序排序  
 \*/* int order() default 0;  
}

eg：

ExtensionLoader<Filter> loader = ExtensionLoader.*getExtensionLoader*(Filter.class);  
Map<String, String> map = new HashMap<>();  
map.put("MyFilter", "cache,context,-echo");  
URL url = new URL("dubbo", "127.0.0.1", 21880,map);  
// key指定为空字符串，即没有给出明确的key  
List<Filter> filterList = loader.getActivateExtension(url, "MyFilter");  
for (Filter filter : filterList) {  
 System.*out*.println(filter.getClass().getName());  
}

**1.作用在类上 ：** 指定该类作为spi扩展的激活时机，一般是通过group和value属性去指定，如果getActivateExtension()的入参满足该扩展类指定的条件，那么该方法返回的扩展实例list中就会有该扩展类。

**2.作用在方法上 ：**

具体实现源码解析可以参见[getActivateExtension()方法](#Activate注解的实现)

* + - 1. 扩展点自动包装(类似AOP代理模式)

自动包装扩展点的 Wrapper 类。ExtensionLoader.getExtensionLoader() 在加载扩展点时，如果加载到的扩展点在META-INF的配置文件中的某个扩展Extension有拷贝构造函数，则判定该扩展Extension为扩展点的Wrapper 类

需要注意：自动包装需要同时创建被代理类和代理类两个扩展Extension，然后将其一起声明到META-INF配置文件中，dubbo会帮我们自动代理

如下例中，扩展点Driver存在两个扩展Extension，分别是OldDriver和OldDriverWrapper，而OldDriverWrapper中存在拷贝构造函数，然后两个扩展又被同时声明到META-INF配置文件中，那么Dubbo就会使用扩展点自动包装。如果此时我们通过ExtensionLoader.getExtensionLoader(Driver.class).getExtension("Old") 方法获取到的就是OldDriverWrapper代理对象，而不是OldDriver对象

@SPI  
public interface Driver {  
 void drive();  
}

public class OldDriver implements Driver{  
  
 @Override  
 public void drive() {  
 System.*out*.println("老司机开车");  
 }  
}

public class OldDriverWrapper implements Driver{  
 private Driver driver;

// 存在拷贝构造函数，将会创建一个OldDriver.class类型的扩展类，由OldDriverWrapper代理

// 最终ExtenssionLoader.getExtenssion获取到的不是OldDriver

// 而是其包装代理对象OldDriverWrapper  
 public OldDriverWrapper(Driver driver){  
 this.driver = driver;  
 }  
 @Override  
 public void drive() {  
 System.*out*.println("老司机发动汽车");  
 driver.drive();  
 System.*out*.println("老司机停车");  
 }  
}

young=org.apache.dubbo.demo.api.YoungDriver

//需要注意的是，如果希望OldDriver被OldDriverWrapper自动包装代理，则需要将包装代理类也声明进来

old=org.apache.dubbo.demo.api.OldDriver  
oldWrapper=org.apache.dubbo.demo.api.OldDriverWrapper

通过扩展点自动包装，我们可以实现类似AOP的增强功能

* + - 1. 扩展点自动注入(类似IOC自动注入)

加载扩展点时，自动注入依赖的扩展点。比如加载扩展点A时，扩展点A实现类的成员变量如果为其它扩展点类型B，ExtensionLoader 在会自动为A的扩展实现类注入B的扩展实现类。ExtensionLoader 通过扫描扩展点实现类的所有 setter 方法来判定其成员。

这个特性就需要B的方法中存在@Adaptive接口修饰的方法，被注入的B是B接口的自适应扩展类，eg:

@SPI("old")  
public interface Driver {  
 @Adaptive  
 void drive(URL url);  
}

public class OldDriver implements Driver{  
 @Override  
 public void drive(URL url) {  
 System.*out*.println("老司机开车");  
 }  
}

@SPI("Mercedes")  
public interface MyCar {  
 //汽车发动  
 void run();  
}

public class MercedesCar implements MyCar{  
 private Driver driver;  
 //set后面的名字可以随意取，注入依赖  
 public void setDependency(Driver driver) {  
 this.driver = driver;  
 }  
 @Override  
 public void run() {  
 System.*out*.println("奔驰车发动");

//没有符合@Adaptive的参数，使用@SPI注解指定的默认值  
 driver.drive(URL.*valueOf*("http://localhost:9999/xxx"));  
 }  
}

实现原理详见[dubbo的ioc源码](#dubbo的ioc源码实现)

* + - 1. SPI源码详解——ExtensionLoader类

ExtensionLoader类是dubbo中用于实现SPI机制的核心类，包括上述的@SPI注解、@Adaptive注解、@Activate注解，IOC、AOP等功能，都是通过该类实现的

每一个ExtensionLoader实例仅负责加载特定SPI扩展的实现。因此想要获取某个扩展的实现，首先要获取到该扩展对应的ExtensionLoader实例

下面我们来看看ExtensionLoader的内部结构

* + - * 1. 重要类变量

private static final String *SERVICES\_DIRECTORY* = "META-INF/services/";  
private static final String *DUBBO\_DIRECTORY* = "META-INF/dubbo/";  
private static final String *DUBBO\_INTERNAL\_DIRECTORY* = *DUBBO\_DIRECTORY* + "internal/";

**dubbo支持三个目录下的配置文件搜寻：**

1. **"META-INF/services/"**是dubbo为了兼容jdk的SPI扩展机制思想而设存在的
2. **"META-INF/dubbo/"**是为了给用户自定义的扩展实现配置文件存放
3. **"META-INF/dubbo/internal/"**是dubbo内部提供的扩展的配置文件路径

private static final ConcurrentMap<Class<?>, ExtensionLoader<?>> *EXTENSION\_LOADERS* = new ConcurrentHashMap<>();  
private static final ConcurrentMap<Class<?>, Object> *EXTENSION\_INSTANCES* = new ConcurrentHashMap<>();

1. **EXTENSION\_LOADERS**静态变量存储的是SPI接口类的Class对象和ExtensionLoader对象的映射关系。Dubble会为每一个SPI接口都会创建一个ExtensionLoader对象，并保存到EXTENSION\_LOADERS变量中。
2. **EXTENSION\_INSTANCES**静态变量存储的是每个SPI接口的多个实现类Class和对应实例之间的关系。因此java进程中通常对应接口实现类对象是单例的。

**//key是扩展类型的Class对象，value是扩展名称**

private final ConcurrentMap<Class<?>, String> cachedNames = new ConcurrentHashMap<>();

**//key是扩展名称，value是扩展类型的Class对象**

private final Holder<Map<String, Class<?>>> cachedClasses = new Holder<>();

**//key是扩展名称，value是标记在该扩展类上的@Activate注解**

private final Map<String, Object> cachedActivates = new ConcurrentHashMap<>();

**//key是扩展名称，value是扩展类的实例对象instance，扩展类的实例对象都是单例的，因此一个扩展名称对应一个Class，也对应一个instance**

private final ConcurrentMap<String, Holder<Object>> cachedInstances = new ConcurrentHashMap<>();

**//自适应扩展类的实例缓存，一个ExtensionLoader只会有一个自适应扩展类，之所以用Holder包裹，是为了双重检查并发加锁时使用的，避免对null加锁，所以生成一个Holder包裹住它**

private final Holder<Object> cachedAdaptiveInstance = new Holder<>();

**//当前ExtensionLoader的自适应扩展类的Class对象，只有类被@Adaptive注解标记，cachedAdaptiveClass才会被赋值**

private volatile Class<?> cachedAdaptiveClass = null;

**//@SPI注解默认的扩展名**

private String cachedDefaultName;

**//AOP用到的，包装类Class对象集合**

private Set<Class<?>> cachedWrapperClasses;

* + - * 1. getExtensionLoader()

我们先来看看最常用的getExtensionLoader()方法

public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {

**// 判断接口类型是否符合要求，不能为空，必须是接口，必须有@SPI修饰**  
 if (type == null) {  
 throw new IllegalArgumentException("Extension type == null");  
 }  
 if (!type.isInterface()) {  
 throw new IllegalArgumentException("Extension type (" + type + ") is not an interface!");  
 }  
 if (!*withExtensionAnnotation*(type)) {  
 throw new IllegalArgumentException("Extension type (" + type +  
 ") is not an extension, because it is NOT annotated with @" + SPI.class.getSimpleName() + "!");  
 }  
 **// 先从静态缓存中获取对应的ExtensionLoader实例**  
 ExtensionLoader<T> loader = (ExtensionLoader<T>) *EXTENSION\_LOADERS*.get(type);  
 if (loader == null) {

**//重**要，**new关键字调用了ExtensionLoader的私有构造函数**  
 *EXTENSION\_LOADERS*.putIfAbsent(type, new ExtensionLoader<T>(type));  
 loader = (ExtensionLoader<T>) *EXTENSION\_LOADERS*.get(type);  
 }  
 return loader;  
}

该方法接收一个Class类型的参数，用于根据Class类型从静态缓存**EXTENSION\_LOADERS**中获取扩展加载器ExtensionLoader

如果缓存中没有，那么就通过new调用私有构造函数创建一个，下面我们看一下ExtensionLoader类的**私有构造函数**

private final ExtensionFactory objectFactory;

private ExtensionLoader(Class<?> type) {  
 this.type = type;  
 objectFactory = (type == ExtensionFactory.class ? null : ExtensionLoader.*getExtensionLoader*(ExtensionFactory.class).getAdaptiveExtension());  
}

从这个私有构造函数我们可以看出：

1. **每个接口的Class会对应一个唯一的ExtensionLoader对象，对应关系存放在EXTENSION\_LOADERS中**
2. **如果扩展类型是ExtensionFactory,那么则设置objectFactory为null**
3. **AdaptiveExtensionFactory是dubbo中仅有的两个被@Adaptive注释在类上的扩展之一，因此其扩展点的getAdaptiveExtesion()将直接返回AdaptiveExtensionFactory扩展**
4. **由于SPI扩展类的实例对象都是单例的，因此不同Class对应的ExtensionLoader持有的objectFactory都是同一个，也就是说所有的ExtensionLoader持有的objectFactory其实都是同一个对象，即单例的AdaptiveExtensionFactory对象**

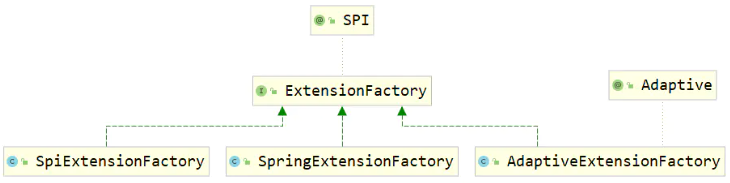
下面我们来看看什么是ExtensionFactory

ExtensionFactory

ExtensionFactory是一个扩展点的工厂类，主要用于加载扩展的实现

@SPI  
public interface ExtensionFactory {  
 */\*\*  
 \* Get extension.  
 \*  
 \** ***@param*** *type 接口类型.  
 \** ***@param*** *name 接口的扩展点名称.  
 \** ***@return*** *扩展点实例对象  
 \*/* <T> T getExtension(Class<T> type, String name);  
}

下面是该接口的继承结构



从上图可以看出 ExtensionFactory接口有三个实现类，具体功能实现在SpiExtensionFactory和SpringExtensionFactory中，而AdaptiveExtensionFactory作为默认实现，主要作用是用于管理协调另外两个实现。AdaptiveExtensionFactory会遍历当前系统中所有的ExtensionFactory实现来获取指定的扩展实现，获取到扩展实现或遍历完所有的ExtensionFactory实现

AdaptiveExtensionFactory

标注了@Adaptive注解，这是dubbo中仅有的两个在类上标注@Adaptive注解的类之一，也就是说dubbo框架的SPI机制不会为该类创建$子类作为代理类，该类的扩展加载逻辑由该类自己实现

另外由于@Adaptive注解被标注在类上(而没有标注在接口的方法上)，那么ExtensionFactory默认的自适应扩展实例为AdaptiveExtensionFactory，也就是说

ExtensionLoader.*getExtensionLoader*(ExtensionFactory.class).getAdaptiveExtension()

上述代码的返回值为AdaptiveExtensionFactory类的实例

ps：如果标记在接口的方法上，则需要通过该方法参数传入的URL参数来确定使用哪个扩展，而ExtensionFactory并未标记在方法上，而是直接标记在类上，这样就不需要URL来确定采用哪个扩展了

*/\*\*  
 \* AdaptiveExtensionFactory  
 \** **由于本实现类上有@Adaptive注解，因此它才是ExtensionFactory的自适应扩展实例** *\* 其本身包含ExtensionFactory的所有实现类  
 \* 在获取接口实例时，就遍历其他的ExtensionFactory实例。调用他们的getExtension方法   
 \*/*@Adaptive  
public class AdaptiveExtensionFactory implements ExtensionFactory {  
  
 **//存放SpiExtensionFactory ,SpringExtensionFactory实例**private final List<ExtensionFactory> factories;  
  
 */\*\*  
 \* 在构造方法中就加载所有的ExtensionFactory的实例  
 \*/* public AdaptiveExtensionFactory() {  
 ExtensionLoader<ExtensionFactory> loader = ExtensionLoader.*getExtensionLoader*(ExtensionFactory.class);  
 List<ExtensionFactory> list = new ArrayList<ExtensionFactory>();  
  **/\*重要  
 \* ExtensionLoader.getSupportedExtensions()返回所有扩展名的TreeSet集合（其实就是spi和spring），  
 \* 里面会对ExtensionFactory进行排序，默认排序会使SpiExtensionFactory实例排在前面  
 \* 这样就会优先从Dubbo的SPI容器中获取扩展点，如果获取不到再从SpringExtensionFactory容器中获取   
 \* 执行完毕后list中将存放SpiExtensionFactory和SpringExtensionFactory  
 \*/**  
 for (String name : loader.getSupportedExtensions()) {  
 list.add(loader.getExtension(name));  
 }  
 //并使用不可变的list存到内存中  
 factories = Collections.unmodifiableList(list);  
 }  
  
 @Override  
 public <T> T getExtension(Class<T> type, String name) {  
  **/\*  
 \* 重**要**，该方法用于直接从SPI容器或spring容器中，根据扩展名获取扩展实例（请注意这个不是构造函数中调用的getExtension方法）  
 \* 依次遍历各个ExtensionFactory实现的getExtension方法，一旦获取到Extension即返回  
 \* 如果遍历完所有的ExtensionFactory实现均无法找到Extension,则返回null  
 \* 因此获取扩展点实例的方法，实际是调用SpiExtensionFactory,SpringExtensionFactory等的getExtension  
 \*/**  
 for (ExtensionFactory factory : factories) {  
 T extension = factory.getExtension(type, name);  
 if (extension != null) {  
 return extension;  
 }  
 }  
 return null;  
 }  
}

SpiExtensionFactory

*/\*\*  
 \* SpiExtensionFactory  
 \* Dubbo SPI扩展点工厂类,主要功能是从Dubbo容器中获取SPI接口的默认的扩展点  
 \*/*public class SpiExtensionFactory implements ExtensionFactory {  
  
 @Override  
 public <T> T getExtension(Class<T> type, String name) {  
  **//1要求type必须是一个接口，并且有@SPI注解。这是dubbo中SPI接口的标准配置**  
 if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {  
  **//2调用ExtensionLoader的静态方法获取type接口的ExtensionLoader实例**  
 ExtensionLoader<T> loader = ExtensionLoader.*getExtensionLoader*(type);  
  **//3 loader.getSupportedExtensions()返回的是扩展点名称的TreeSet集合**  
 if (!loader.getSupportedExtensions().isEmpty()) {  
  **//4 重**要**，使用ExtensionLoader#getAdaptiveExtension()获取默认的实现类**  
 return loader.getAdaptiveExtension();  
 }  
 }  
  **//不满足就直接返回null**  
 return null;  
 }  
  
}

从SpiExtensionFactory 的实现可以看出，底层还是使用的ExtensionLoader#getAdaptiveExtension() 。而参数name在整个过程没有被使用。这里获取SPI接口实例的关键是接口type

SpringExtensionFactory

这是Dubbo整合Spring框架时，获取spring的bean容器中的实例的工厂类

*/\*\*  
 \* SpringExtensionFactory  
 \* dubbo与spring容器的整合  
 \* 通过SpringExtensionFactory可以获取到spring容器中的扩展类  
 \*/*public class SpringExtensionFactory implements ExtensionFactory {  
 private static final Logger *logger* = LoggerFactory.getLogger(SpringExtensionFactory.class);  
  
 */\*\*  
 \* 自动去重的set集合保存spring的上下文对象  
 \*/* private static final Set<ApplicationContext> *CONTEXTS* = new ConcurrentHashSet<ApplicationContext>();  
  
 public static void addApplicationContext(ApplicationContext context) {  
 *CONTEXTS*.add(context);  
 if (context instanceof ConfigurableApplicationContext) {  
 ((ConfigurableApplicationContext) context).registerShutdownHook();  
 }  
 }  
  
 public static void removeApplicationContext(ApplicationContext context) {  
 *CONTEXTS*.remove(context);  
 }  
  
 public static Set<ApplicationContext> getContexts() {  
 return *CONTEXTS*;  
 }  
  
 // currently for test purpose  
 public static void clearContexts() {  
 *CONTEXTS*.clear();  
 }  
  
 */\*\*  
 \* 从spring容器中获取指定class类型和名称的对象  
 \** ***@param*** *type object type. 扩展点类型  
 \** ***@param*** *name object name. 扩展点名称  
 \** ***@param*** <*T*> *扩展点class  
 \** ***@return*** *扩展点  
 \*/* @Override  
 @SuppressWarnings("unchecked")  
 public <T> T getExtension(Class<T> type, String name) {  
 **/\*  
 \* 如果扩展类时一个接口，并且接口上由@SPI注解，就返回null。  
 \* 意思是：SPI接口的扩展点实现应该从SpiExtensionFactory中获取，而不是通过本类获取  
 \*/**  
 if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {  
 return null;  
 }  
  
  **//遍历Spring的上下文对象ApplicationContext**  
 for (ApplicationContext context : *CONTEXTS*) {  
  **//重**要**，通过接口类型和接口实现的名称从上下文中获取接口的实例对象，如果有多个实现，默认获取第一个**  
 T bean = BeanFactoryUtils.getOptionalBean(context, name, type);  
 if (bean != null) {  
 return bean;  
 }  
 }  
 **//Spring容器中没有找到就返回null**   
 return null;  
 }  
}

由实现逻辑可以看出，就是在spring容器的上下文对象作为静态变量存入SpringExtensionFactory中，获取接口实例时就遍历spring的上下文，从中根据接口类型和实现类的名称去查找，有就返回，没有就返回null。

* + - * 1. getExtension()

下面来看一下getExtension()的源码，该方法主要实现[@SPI注解](#Spi注解)功能

getExtension(name)

-> getDefaultExtension() #如果name为字符串true，则返回默认扩展

-> createExtension(name) #如果无缓存则创建

-> getExtensionClasses().get(name) #获取name对应的扩展类型

-> 实例化扩展类

-> injectExtension(instance) # 扩展点ioc注入

-> instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance)) #循环遍历所有wrapper实现，实例化wrapper并进行扩展点注入，即dubbo的aop

*/\*\*  
 \* 寻找指定名称的扩展，如果没找到就会抛出IllegalStateException异常  
 \*/*

public T getExtension(String name) {  
 if (StringUtils.*isEmpty*(name)) {  
 throw new IllegalArgumentException("Extension name == null");  
 }

**//重**要**，如果name为字符串true，那么就去找默认的扩展实例，即@SPI接口中指定的扩展**  
 if ("true".equals(name)) {  
 return getDefaultExtension();  
 }

**// Holder，顾名思义，用于持有目标对象**

**// getOrCreateHolder()方法中封装了查找cachedInstances缓存的逻辑**  
 final Holder<Object> holder = getOrCreateHolder(name);  
 Object instance = holder.get();

**// 双重检查防止并发问题**  
 if (instance == null) {  
 synchronized (holder) {  
 instance = holder.get();  
 if (instance == null) {

**//重**要**，根据扩展名从SPI配置文件中查找对应的扩展实现类对象**  
 instance = createExtension(name);  
 holder.set(instance);  
 }  
 }  
 }  
 return (T) instance;  
}

ps：我们可以看到，上面的代码是使用了一个Holder对象来持有扩展实现类对象，那为什么要在外面包裹一层Hoder呢？这是因为后面是双重检查，如果holder变量不采用Holder类包裹，而是直接用扩展实现类对象instance，那么holder变量就可能出现为null的情况，而对一个null加synchronized锁是有问题，因此需要Holder进行包裹

getDefaultExtension()

我们可以看到，这里主要是通过getExtension()方法去加载扩展名为cachedDefaultName的扩展实例

*/\*\*  
 \* 返回默认扩展，如果没有配置默认扩展，则返回null  
 \*/*

public T getDefaultExtension() {

**//重**要**，加载cachedClasses，如果cachedClasses还没有，那就通过loadExtensionClasses()方法去加载它，cachedClasses是一个Hashmap，它的key是扩展名称，value是扩展类型Class对象,详见**[**getExtensionClass()**](#getExtensionClasses)  
 getExtensionClasses();  
 if (StringUtils.*isBlank*(cachedDefaultName) || "true".equals(cachedDefaultName)) {  
 return null;  
 }  
 return getExtension(cachedDefaultName);  
}

那么cachedDefaultName从哪儿来的呢？很明显，它是@SPI注解指定的value值，源码如下：

private void cacheDefaultExtensionName() {  
 final SPI defaultAnnotation = type.getAnnotation(SPI.class);  
 if (defaultAnnotation == null) {  
 return;  
 }  
  
 String value = defaultAnnotation.value();  
 if ((value = value.trim()).length() > 0) {  
 String[] names = *NAME\_SEPARATOR*.split(value);  
 if (names.length > 1) {  
 throw new IllegalStateException("More than 1 default extension name on extension " + type.getName()  
 + ": " + Arrays.*toString*(names));  
 }  
 if (names.length == 1) {  
 cachedDefaultName = names[0];  
 }  
 }  
}

createExtension()

private T createExtension(String name) {

**//重**要**，尝试从*cachedClasses*缓存中获取扩展类的Class对象，*cachedClasses*缓存是一个HashMap，key是扩展名，value是扩展类的Class对象**  
 Class<?> clazz = getExtensionClasses().get(name);  
 if (clazz == null) {  
 throw findException(name);  
 }  
 try {

**//重**要**，尝试从*EXTENSION\_INSTANCES*缓存中获取扩展类的实例对象，*EXTENSION\_INSTANCES*缓存是一个HashMap，key是扩展类的Class对象，value是扩展类的实例对象(扩展类的实例对象都是单例的，因此一个Class对应一个instance)**  
 T instance = (T) *EXTENSION\_INSTANCES*.get(clazz);  
 if (instance == null) {  
 *EXTENSION\_INSTANCES*.putIfAbsent(clazz, clazz.newInstance());  
 instance = (T) *EXTENSION\_INSTANCES*.get(clazz);  
 }

**//重**要**，属性注入(即**[**dubbo的ioc**](#dubbo的ioc)**依赖注入)**  
 injectExtension(instance);  
 Set<Class<?>> wrapperClasses = cachedWrapperClasses;

**//重**要**，如果有自动包装扩展类（即**[**dubbo的AOP**](#dubbo的aop)**自动代理实现）**  
 if (CollectionUtils.*isNotEmpty*(wrapperClasses)) {  
 for (Class<?> wrapperClass : wrapperClasses) {

**//重**要**，循环调用拷贝构造函数对instance进行层层包裹**  
 instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance));  
 }  
 }

**//初始化(包裹后的)instance**  
 initExtension(instance);  
 return instance;  
 } catch (Throwable t) {  
 throw new IllegalStateException("Extension instance (name: " + name + ", class: " +  
 type + ") couldn't be instantiated: " + t.getMessage(), t);  
 }  
}

injectExtension()——dubbo的ioc

private T injectExtension(T instance) {  
 if (objectFactory == null) {  
 return instance;  
 }  
  
 try {  
 for (Method method : instance.getClass().getMethods()) {

**// 当下列三个条件都成立时，isSetter()返回true**

**1. public**

**2. 名称以 "set"开头**

**3．只有一个入参**

if (!isSetter(method)) {  
 continue;  
 }  
 **//如果setter方法上存在*@DisableInject*注解，则不注入**if (method.getAnnotation(DisableInject.class) != null) {  
 continue;  
 }

**//获取setter方法的入参类型**  
 Class<?> pt = method.getParameterTypes()[0];

**//如果setter方法的入参类型是8种基本类型，则不注入**  
 if (ReflectUtils.*isPrimitives*(pt)) {  
 continue;  
 }  
  
 try {

**//从setter方法名中获取要注入的属性名(驼峰转小写)**  
 String property = getSetterProperty(method);

**//重**要**，使用AdaptiveExtensionFactory加载并实例化扩展实现类(先在SpiExtensionFactory中找，然后再去SpringExtensionFactory中找，都没有就返回null)**  
 Object object = objectFactory.getExtension(pt, property);  
 if (object != null) {

**//重**要**，通过setter方法注入instance中**  
 method.invoke(instance, object);  
 }  
 } catch (Exception e) {  
 *logger*.error("Failed to inject via method " + method.getName()  
 + " of interface " + type.getName() + ": " + e.getMessage(), e);  
 }  
  
 }  
 } catch (Exception e) {  
 *logger*.error(e.getMessage(), e);  
 }  
 return instance;  
}

* + - * 1. getActivateExtension()

该方法主要实现了[@Activate注解](#Activate注解)机制

getActivateExtension方法主要获取当前扩展的所有可自动激活的实现，该方法虽然有4个重载函数，但是基本最后都是调用了同一个重载，代码如下：

public List<T> getActivateExtension(URL url, String[] values, String group) {  
 List<T> exts = new ArrayList<>();  
 List<String> names = values == null ? new ArrayList<>(0) : Arrays.*asList*(values);

**// 如果 values数组中不包含字符串"-default"，则遍历 缓存中所有的 默认自动激活扩展**

**// 如果存在某个value值为"-default"，则移除所有 默认自动激活扩展**

if (!names.contains(*REMOVE\_VALUE\_PREFIX* + *DEFAULT\_KEY*)) {

**// 触发*cachedActivates*等缓存字段的加载，将标记了@Activate注解的实现类缓存到*cachedActivates*集合中，详见**[**getExtensionClasses()**](#getExtensionClasses)  
 getExtensionClasses();  
 for (Map.Entry<String, Object> entry : cachedActivates.entrySet()) {

**//扩展名**  
 String name = entry.getKey();

**//标记在该扩展类上的@Activate注解**  
 Object activate = entry.getValue();  
  
 String[] activateGroup, activateValue;  
  
 if (activate instanceof Activate) {  
 activateGroup = ((Activate) activate).group();  
 activateValue = ((Activate) activate).value();  
 } else if (activate instanceof com.alibaba.dubbo.common.extension.Activate) {

**//兼容老版本的注解**  
 activateGroup = ((com.alibaba.dubbo.common.extension.Activate) activate).group();  
 activateValue = ((com.alibaba.dubbo.common.extension.Activate) activate).value();  
 } else {

**//没指定value和group，则跳过该扩展类**  
 continue;  
 }

**//检查入参的group和注解上的group是否匹配**  
 if (isMatchGroup(group, activateGroup)

**//检查入参的values数组元素中是否有注解上的name值**

**//本循环只激活默认的扩展实现（下面会专门处理要获取的扩展点)，如果名称出现在values配置中的，那么就不是默认激活的扩展实现，因此需要去除**  
 && !names.contains(name)

**//”-name”这种的也不需要激活**  
 && !names.contains(*REMOVE\_VALUE\_PREFIX* + name)  
 && isActive(activateValue, url)) {  
 exts.add(getExtension(name));  
 }  
 }  
 exts.sort(ActivateComparator.*COMPARATOR*);  
 }  
 List<T> usrs = new ArrayList<>();

**//前面只获取默认激活的，这里获取values指定激活的**  
 for (int i = 0; i < names.size(); i++) {  
 String name = names.get(i);

**// 通过"-"开头配置的扩展 为 明确指定不激活的扩展实现，直接就忽略了**  
 if (!name.startsWith(*REMOVE\_VALUE\_PREFIX*)  
 && !names.contains(*REMOVE\_VALUE\_PREFIX* + name)) {  
 if (*DEFAULT\_KEY*.equals(name)) {  
 if (!usrs.isEmpty()) {

**// 按照顺序，将自定义的扩展addAll到默认扩展集合的最前面**

**// 例如，<dubbo:service filter="demo,default,demo2" /> ，则 DemoFilter 就会放在默认的过滤器前面。demo2就会放到默认过滤器的后面**  
 exts.addAll(0, usrs);  
 usrs.clear();  
 }  
 } else {  
 usrs.add(getExtension(name));  
 }  
 }  
 }  
 if (!usrs.isEmpty()) {

**// 按照顺序，将自定义的扩展添加到默认扩展集合后面**  
 exts.addAll(usrs);  
 }  
 return exts;  
}

* 1. provider的服务暴露过程

服务暴露是指服务提供者provider向注册中心register注册服务，将服务实现类以服务接口的形式提供出去，以便服务消费者从注册中心查阅并调用服务

服务暴露分为本地暴露和远程暴露，一个service服务不仅会暴露到注册中心中一份，也会暴露一份到当前所在机器的jvm中。为什么呢？因为也有可能当前机器中其他的服务会使用到这个demoService服务，同一台机器上就不必走注册中心，肯定直接走jvm会更快

当然我们这里主要看远程暴露

* + 1. 服务暴露的入口
       1. xml和注解入口
          1. <dubbo:service>入口

如果使用xml配置的<dubbo:service>来声明一个dubbo服务，dubbo会使用xml命名空间处理器DubboNamespaceHandler来解析xml配置，最终将<dubbo:service>标签解析为一个serviceBean来处理

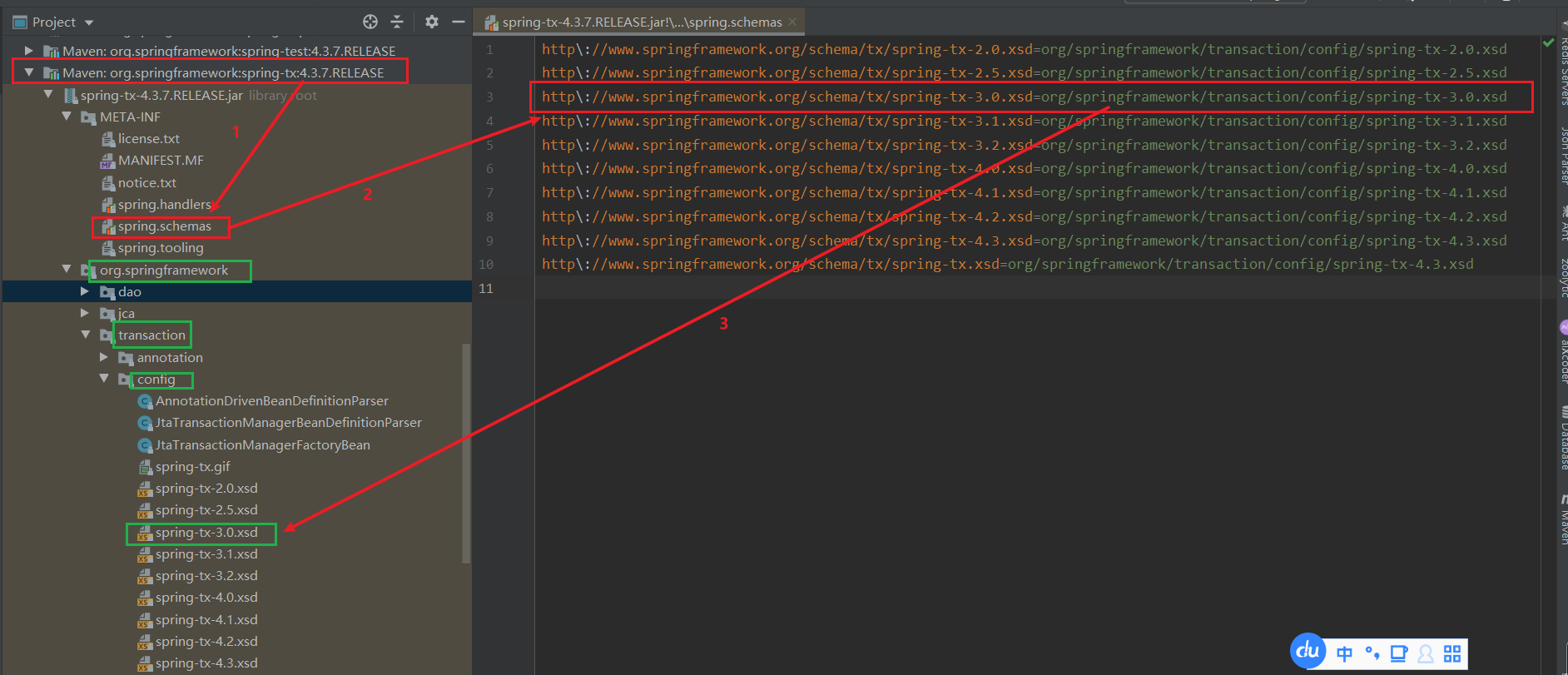
xmlns机制

接下来我们拿spring的<tx:advice>标签举个例子来讲解一下xmlns机制，其实dubbo也是一样的原理

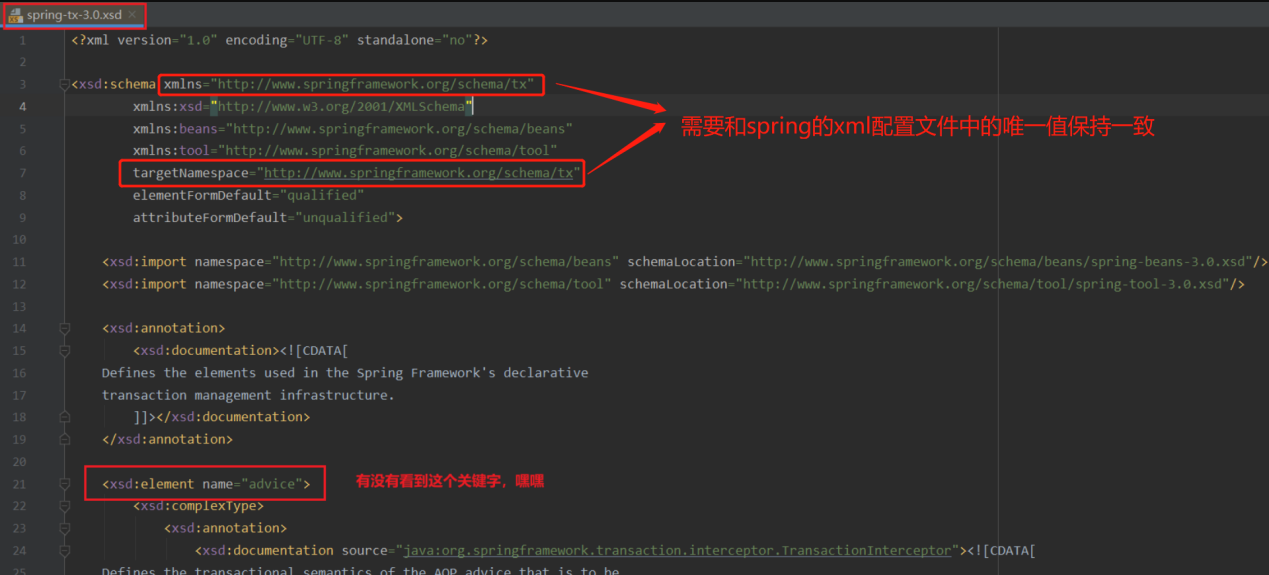
xmlns全称是xml namespace，翻译一下就是xml命名空间，那么它有什么用？其实他类似一个键值对（例如下图的tx ---> www.springframework.org/schema/tx）, 键tx表示xml标签的前缀，值(www.springframework.org/schema/tx)就是一个唯一标识，但是这个唯一标识还需要和xsi:schemaLocation中的一个xsd文件对应起来(即[www.springframework.org/schema/tx](http://www.springframework.org/schema/tx) --->www.springframework.org/schema/tx/spring-tx-3.0.xsd)，如下图所示

每一个xsd文件中就是描述了当前命名空间中指定的标签，规范了各个属性



其实根据这里的xml配置文件中的头文件信息，spring就会去寻找[www.springframework.org/schema/tx/spring-tx-3.0.xsd](http://www.springframework.org/schema/tx/spring-tx-3.0.xsd)文件，其实在spring-tx.jar中META-INF/spring.schemas中可以找到本地该文件的地址，如果没有，才会联网去spring官方地址那里去下载，如下图

接着点开路径org/springframework/transaction/config/spring-tx-3.0.xsd，如下图：

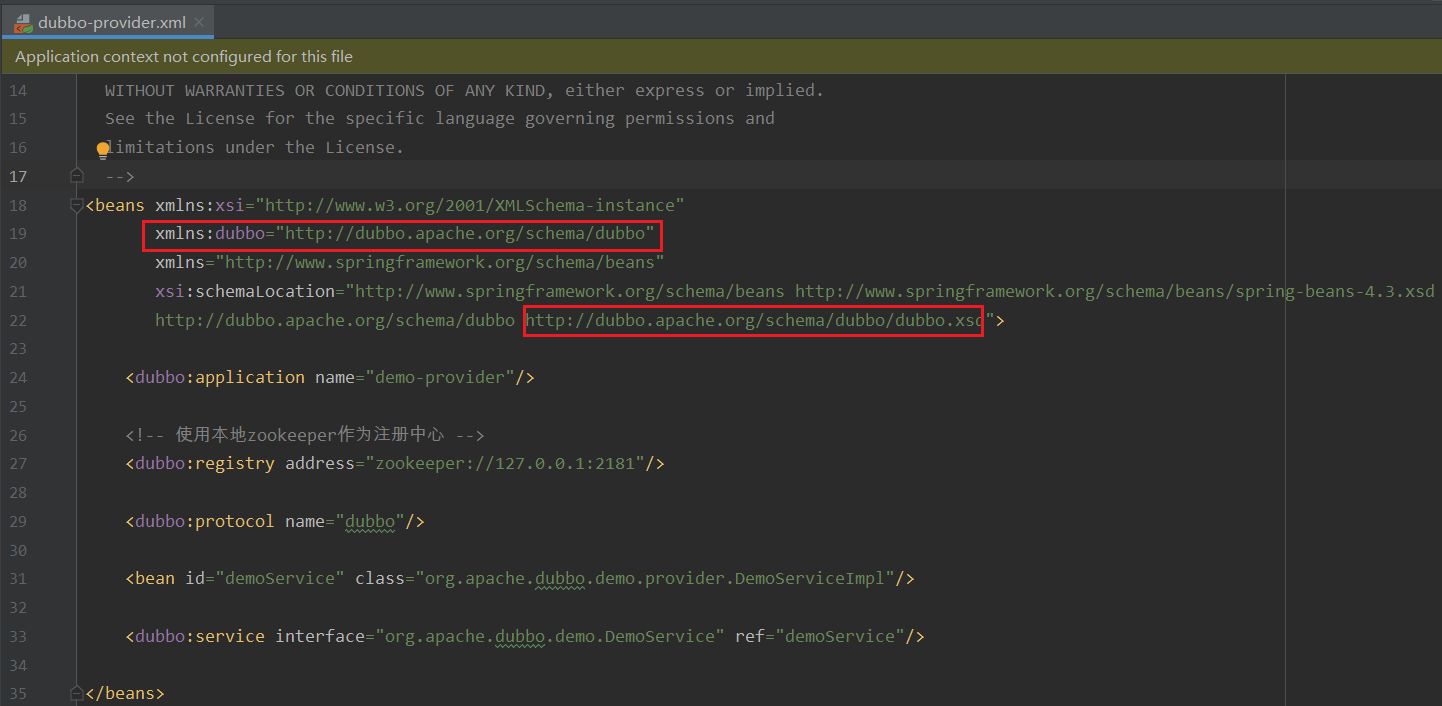


从上图中我们可以看到<tx:advice>标签中各个属性值，以及规定的子标签，但需要注意xmlns和targetNamespqce属性需要和xml配置文件中的唯一标识保持一致

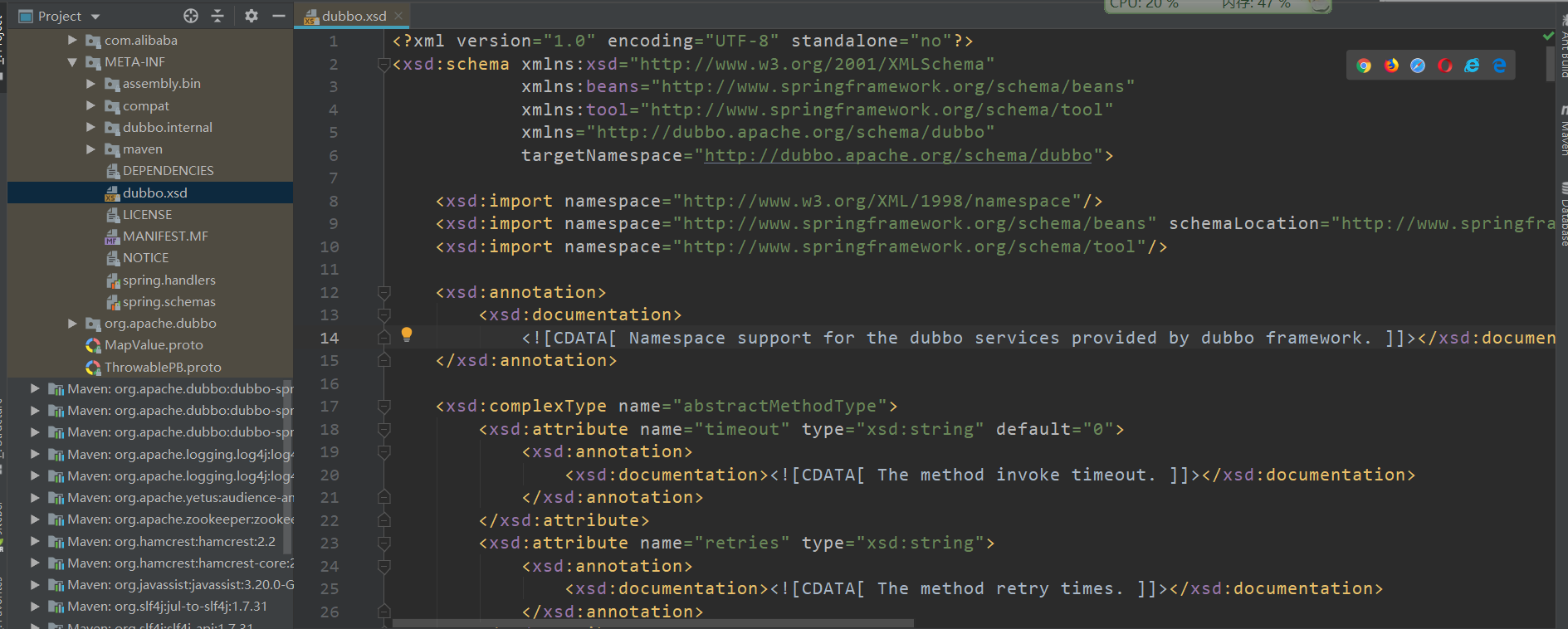
解析dubbo标签

回到dubbo

我们在dubbo的xml文件中会加入这么一段spring配置文件的头文件：



这就意味着我们需要去找dubbo.xsd文件，我们很容易就找到了xsd文件



那么如何根据这个xsd的规则去解析xml文件呢？这就要用到NamespaceHandler和BeanDefinitionParser，NamespaceHandler会根据schema和节点名找到某个BeanDefinitionParser，然后由 BeanDefinitionParser完成具体的解析工作。

spring.handlers文件是NamespaceHandler和XSD文件的连接器，即NamespaceHandler通过spring.handlers配置文件找到XSD文件

让我们来看一下spring.handlers文件的内容：

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

很明显，这里是使用DubboNameSpaceHanlder来解析的，其源码如下：

public class DubboNamespaceHandler extends NamespaceHandlerSupport implements ConfigurableSourceBeanMetadataElement {  
 static {  
 Version.*checkDuplicate*(DubboNamespaceHandler.class);  
 }  
 @Override  
 public void init() {  
 registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(ApplicationConfig.class, true));  
 registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class, true));  
 registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(RegistryConfig.class, true));  
 registerBeanDefinitionParser("config-center", new DubboBeanDefinitionParser(ConfigCenterBean.class, true));  
 registerBeanDefinitionParser("metadata-report", new DubboBeanDefinitionParser(MetadataReportConfig.class, true));  
 registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class, true));  
 registerBeanDefinitionParser("metrics", new DubboBeanDefinitionParser(MetricsConfig.class, true));  
 registerBeanDefinitionParser("ssl", new DubboBeanDefinitionParser(SslConfig.class, true));  
 registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(ProviderConfig.class, true));  
 registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, true));  
 registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, true));

//重要，解析<dubbo:service/>标签 用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心  
  **registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));**  
 registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class, true));  
 registerBeanDefinitionParser("annotation", new AnnotationBeanDefinitionParser());  
 }

//当执行refresh()的obtainFreshBeanFactory()加载BeanDefinition时，会找到DubboNamespaceHandler触发parse方法

@Override

public BeanDefinition parse(Element element, ParserContext parserContext) {

BeanDefinitionRegistry registry = parserContext.getRegistry();

registerAnnotationConfigProcessors(registry);

//重要，注册[DubboBootstrapApplicationListener](#DubboBootstrapApplicationListener监听器原理)监听器，即将其作为一个BeanDefinition注册到spring的BeanDefinitionMap中，该监听器负责监听容器的刷新和关闭，当spring容器刷新时，会进行服务暴露和引用，容器关闭时执行销毁

registerApplicationListeners(registry);

//重要，找到init()方法中注册的DubboBeanDefinitionParser，用它来解析对应的元素标签，将其解析为一个BeanDefinition<ServiceBean>对象

BeanDefinition beanDefinition = super.parse(element, parserContext);

setSource(beanDefinition);

return beanDefinition;

}

private void registerApplicationListeners(BeanDefinitionRegistry registry) {

registerBeans(registry, DubboLifecycleComponentApplicationListener.class);

registerBeans(registry, DubboBootstrapApplicationListener.class);

}

}

对于xml入口，这里只是将<dubbo:service>标签解析，并封装成为一个BeanDefinition<ServiceBean>对象。在spring的机制中，后续才会真正的根据BeanDefinition去实例化出ServiceBean对象

在spring的解析过程中·，会调用DubboNamespaceHandler#parse()，具体的调用链是:

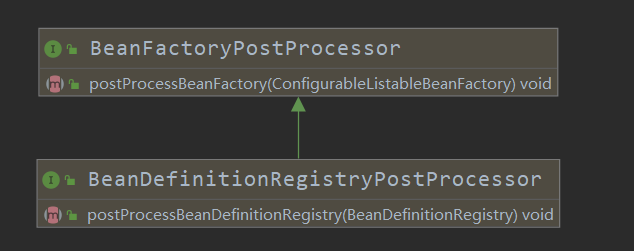
spring调用refresh() 🡪 obtainFreshBeanFactory() 🡪 refreshBeanFactory() 🡪 loadBeanDefinitions() 🡪 XmlBeanDefinitionReader#loadBeanDefinitions() 🡪 doLoadBeanDefinitions() 🡪 registerBeanDefinitions 🡪 documentReader.registerBeanDefinitions() 🡪 doRegisterBeanDefinitions() 🡪 parseBeanDefinitions() 🡪 delegate.parseCustomElement() 🡪 BeanDefinitionParserDelegate#parseCustomElement()在这里获取NameSpaceHandler 🡪 最后调用handler.parse()

因为这里的解析不是本文的重点，因此我们只需要知道spring会在refresh()的obtainFreshBeanFactory()中调用DubboNamespaceHandler#parse()即可

* + - * 1. @Service注解入口

对于@Service标注的服务类，Dubbo主要是通ServiceAnnotationBeanPostProcessor来实现服务类加载的，ServiceAnnotationBeanPostProcessor类实现了BeanDefinitionRegistryPostProcessor接口，其方法postProcessBeanDefinitionRegistry()会在当前容器默认的BeanDefinition注册完毕后调用。Dubbo就是在该方法中扫描@Service标注的类，然后构建一个ServiceBean对象注入到Spring容器中

这里介绍一下BeanDefinitionRegistryPostProcessor接口，这是spring留下的扩展点，在执行spring的refresh()方法时，会执行invokeBeanFactoryPostProcessors()，该方法主要作用是调用并执行所有的已定义实现BeanFactoryPostProcessor接口的类的postProcessBeanFactory()方法，这样可以对beanFactory和beanDefinition进行后续处理，而BeanFactoryPostProcessor接口有一个子接口，就是我们的BeanDefinitionRegistryPostProcessor接口，接口继承关系如下图



refresh()的invokeBeanFactoryPostProcessors()方法会:

1. 先执行子接口BeanDefinitionRegistryPostProcessor的实现类的postProcessBeanDefinitionRegistry()方法，然后再执行父接口BeanFactoryPostProcessor的实现类的postProcessBeanFactory()方法

2. 在满足第一条的前提下，优先执行实现了PriorityOrdered接口的实现类，然后再执行实现了Ordered接口的实现类，最后是常规的实现类

通过BeanDefinitionRegistryPostProcessor接口，我们可以在bean实例化之前，向BeanDefinitionMap中添加新的BeanDefinition

接着我们再看看ServiceAnnotationBeanPostProcessor类的源码：

public class ServiceAnnotationBeanPostProcessor implements BeanDefinitionRegistryPostProcessor, EnvironmentAware,  
 ResourceLoaderAware, BeanClassLoaderAware {  
 ……  
 @Override  
 public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {  
 //重要，注册了一个DubboBootstrapApplicationListener监听，负责监听容器的刷新和关闭，当spring容器刷新时，会进行服务暴露和引用，容器关闭时执行销毁  
 *registerBeans*(registry, DubboBootstrapApplicationListener.class);

// packagesToScan中保存了指定的要扫描的dubbo类路径，这里主要是对这些路径中的占位符进行处理，以将其替换为相应的值  
 Set<String> resolvedPackagesToScan = resolvePackagesToScan(packagesToScan);  
 if (!CollectionUtils.*isEmpty*(resolvedPackagesToScan)) {

//重要，注册被dubbo @Service注解标注的类，主要是将被标记的类注册成ServiceBean类型的BeanDefinition到BeanDefinitionMap中去  
 registerServiceBeans(resolvedPackagesToScan, registry);  
 } else {  
 if (logger.isWarnEnabled()) {  
 logger.warn("packagesToScan is empty , ServiceBean registry will be ignored!");  
 }  
 }  
 }

}

* + - 1. ServiceBean

可以看到无论是xml入口还是@Service入口，最终都是注册了一个ServiceBean类型的BeanDefinition到BeanDefinitionMap中。一个ServiceBean将包裹一个业务上的ServiceImpl的bean

再往下，2.7.5之前的dubbo和2.7.5之后的dubbo就不同了

* + - * 1. dubbo2.7.5之前的ServiceBean

我们先来看一下2.7.5之前的ServiceBean源码

public class ServiceBean<T> extends ServiceConfig<T> implements InitializingBean, DisposableBean,  
 ApplicationContextAware, ApplicationListener<ContextRefreshedEvent>, BeanNameAware,  
 ApplicationEventPublisherAware {

……

@Override

public void onApplicationEvent(ContextRefreshedEvent event) {  
 if (isDelay() && !isExported() && !isUnexported()) {  
 if (logger.isInfoEnabled()) {  
 logger.info("The service ready on spring started. service: " + getInterface());  
 }  
 export();  
 }

}

}

我们可以看到ServiceBean实现了InitializingBean和ApplicationListener接口。

* 1. 那么当spring容器执行refresh()方法时，首先会读取dubbo服务的bean配置，并将其加载为BeanDefinition<ServiceBean>对象
  2. 然后进行实例化，在spring执行createBeanInstance()方法创建了一个ServiceBean对象之后；
  3. spring接着会执行populateBean()设置bean的属性值；
  4. 然后就是对ServiceBean进行初始化操作，会判断是否实现了InitializingBean接口，实现了的话，就会执行afterPropertiesSet方法先初始化一下；
  5. 然后会判断ServcieBean有没有初始化方法，即配置文件中<bean>标签指定的init-method方法
  6. 随后执行finishRefresh()，里面会执行publishEvent()，随后spring会获取到所有的ApplicationListener。若有executor支持进行异步派发，否则直接同步执行InvokeListener()方法。该方法会去执行每个bean的onApplicationEvent()方法
  7. 至此，ServiceBean就会去执行onApplicationEvent()方法，执行export()方法实际的去开始暴露自己代表的服务，这就是服务暴露的启动点
     + - 1. dubbo2.7.5及之后的ServiceBean

我们先来看一下2.7.5之后的ServiceBean源码

public class ServiceBean<T> extends ServiceConfig<T> implements InitializingBean, DisposableBean,  
 ApplicationContextAware, BeanNameAware,  
 ApplicationEventPublisherAware {

……

@Override

public void afterPropertiesSet() throws Exception {  
 if (StringUtils.*isEmpty*(getPath())) {  
 if (StringUtils.*isNotEmpty*(beanName) && StringUtils.*isNotEmpty*(getInterface())

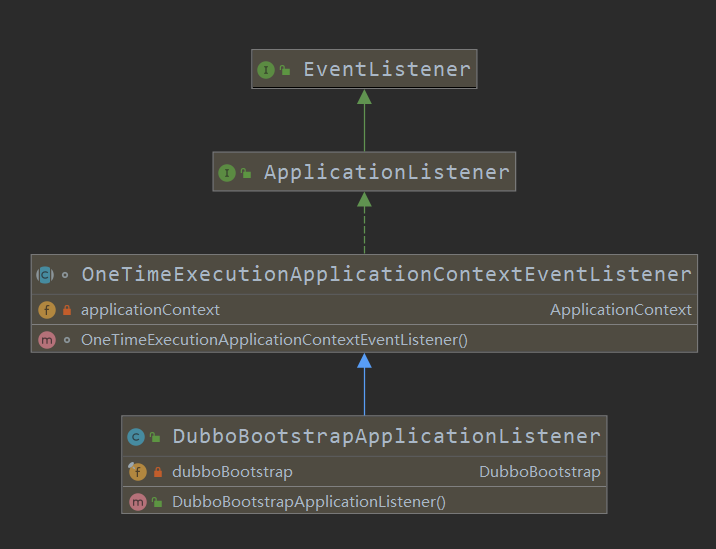
&& beanName.startsWith(getInterface())) {  
 setPath(beanName);  
 }  
 }  
 }

}

很显然，2.7.5以后已经去掉了ApplicationListener接口，那么dubbo要如何执行服务暴露呢？

* + - 1. DubboBootstrapApplicationListener

<dubbo:\*/>标签是自定义命名空间,解析就交给DubboNamespaceHandler处理,当refresh()的obtainFreshBeanFactory()方法解析dubbo命名空间下的节点时，会触发DubboNamespaceHandler#parse()方法，该方法会注册一个[DubboBootstrapApplicationListener](#注册DubboBootstrapApplicationListener监听器源码)类型的监听器，类图如下：



我们可以看到，DubboBootstrapApplicationListener又是一个ApplicationListener，这意味着在收到spring容器事件时，会触发该类的onApplicationContextEvent()方法，下面我们来看一下它的源码

public class DubboBootstrapApplicationListener extends OneTimeExecutionApplicationContextEventListener implements Ordered {

//dubbo启动引导器  
 private final DubboBootstrap dubboBootstrap;  
 public DubboBootstrapApplicationListener() {  
 this.dubboBootstrap = DubboBootstrap.*getInstance*();  
 }

//当接收到spring容器事件时，先执行父类OneTimeExecutionApplicationContextEventListener的onApplicationEvent()方法，然后该方法将调用onApplicationContextEvent()方法  
 @Override  
 public void onApplicationContextEvent(ApplicationContextEvent event) {  
 if (event instanceof ContextRefreshedEvent) {

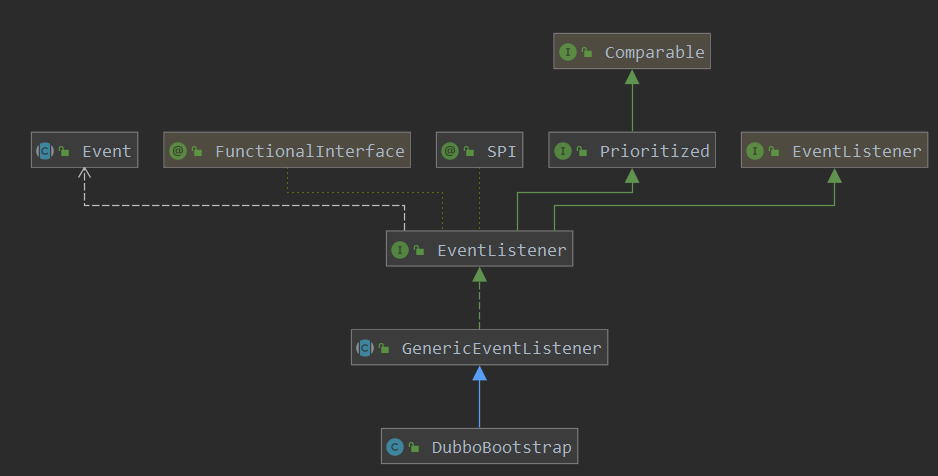
// dubboBootstrap.start()  
 onContextRefreshedEvent((ContextRefreshedEvent) event);  
 } else if (event instanceof ContextClosedEvent) {

// dubboBootstrap.stop()  
 onContextClosedEvent((ContextClosedEvent) event);  
 }  
 }  
 private void onContextRefreshedEvent(ContextRefreshedEvent event) {  
 dubboBootstrap.start();  
 }  
  
 private void onContextClosedEvent(ContextClosedEvent event) {  
 dubboBootstrap.stop();  
 }  
  
 @Override  
 public int getOrder() {  
 return *LOWEST\_PRECEDENCE*;  
 }  
}

也就是说：

1. 当spring容器刷新时，将自动执行dubboBootstrap.start()
2. 当spring容器销毁时，将自动执行dubboBootstrap.stop()
   * + 1. DubboBootstrap

接下来我们看一下DubboBootstrap类，其类图如下



其部分源码如下：

public DubboBootstrap start() {  
 if (started.compareAndSet(false, true)) {  
 initialize();  
 if (logger.isInfoEnabled()) {  
 logger.info(*NAME* + " is starting...");  
 }  
 // 重要，暴露dubbo服务  
 exportServices();  
  
 // Not only provider register  
 if (!isOnlyRegisterProvider() || hasExportedServices()) {  
 // 2. export MetadataService  
 exportMetadataService();  
 //3. Register the local ServiceInstance if required  
 registerServiceInstance();  
 }  
 // 重要，执行consumer的服务引用流程  
 referServices();  
  
 if (logger.isInfoEnabled()) {  
 logger.info(*NAME* + " has started.");  
 }  
 }  
 return this;  
}

我们可以看到，DubboBootstrap只是一个普通的监听器EventListener，但是里面有一个start()方法，这里会调用exportServices()方法，然后就会去调用ServiceConfig的export()方法

exportServices()方法源码如下：

private void exportServices() {  
//configManager.getServices() 获取的就是在provider.xml中配置的dubbo:service对应bean,类型是ServiceConfig

configManager.getServices().forEach(sc -> {  
 ServiceConfig serviceConfig = (ServiceConfig) sc;  
 serviceConfig.setBootstrap(this);  
 // 如果exportAsync为true就异步export,否则同步发布,默认为false  
 if (exportAsync) {  
 ExecutorService executor = executorRepository.getServiceExporterExecutor();  
 Future<?> future = executor.submit(() -> {  
 sc.export();  
 });  
 asyncExportingFutures.add(future);  
 } else {  
 sc.export();  
 exportedServices.add(sc);  
 }  
 });  
}

可以看到，无论是同步暴露还是异步暴露，最终还是调用了export()方法

* + 1. export()

export()方法是负责dubbo服务暴露的核心方法，同样的，export()不同dubbo版本也会有区别，但是基本都是进行相关的检查，然后选择立刻暴露或者延迟暴露

下面是两个版本的源码，我们可以看到，真正负责暴露的还是doExport()方法

dubbo 2.6.5源码如下：

public synchronized void export() {  
 if (provider != null) {  
 if (export == null) {  
 export = provider.getExport();  
 }  
 if (delay == null) {  
 delay = provider.getDelay();  
 }  
 }  
 if (export != null && !export) {  
 return;  
 }  
  
 if (delay != null && delay > 0) {  
 *delayExportExecutor*.schedule(new Runnable() {  
 @Override  
 public void run() {

//重要，执行服务暴露  
 doExport();  
 }  
 }, delay, TimeUnit.*MILLISECONDS*);  
 } else {

//重要，执行服务暴露  
 doExport();  
 }  
}

dubbo 2.7.5源码如下：

public synchronized void export() {  
 if (!shouldExport()) {  
 return;  
 }  
  
 if (bootstrap == null) {  
 bootstrap = DubboBootstrap.*getInstance*();  
 bootstrap.init();  
 }  
  
 checkAndUpdateSubConfigs();  
  
 //init serviceMetadata  
 serviceMetadata.setVersion(version);  
 serviceMetadata.setGroup(group);  
 serviceMetadata.setDefaultGroup(group);  
 serviceMetadata.setServiceType(getInterfaceClass());  
 serviceMetadata.setServiceInterfaceName(getInterface());  
 serviceMetadata.setTarget(getRef());  
  
 if (shouldDelay()) {  
 *DELAY\_EXPORT\_EXECUTOR*.schedule(this::doExport, getDelay(), TimeUnit.*MILLISECONDS*);  
 } else {

//重要，执行服务暴露  
 doExport();  
 }  
}

可以看出doExport才是真正执行暴露的方法

* + 1. doExport()

doExport()在执行完一堆检查后会执行doExportUrls()

protected synchronized void doExport() {  
 if (unexported) {  
 throw new IllegalStateException("The service " + interfaceClass.getName() + " has already unexported!");  
 }

//如果已经暴露，就不再重复暴露  
 if (exported) {  
 return;  
 }  
 exported = true;  
  
 if (StringUtils.*isEmpty*(path)) {  
 path = interfaceName;  
 }

//重要，执行服务暴露  
 doExportUrls();  
  
 // dispatch a ServiceConfigExportedEvent since 2.7.4  
 dispatch(new ServiceConfigExportedEvent(this));  
}

* + - 1. doExportUrls()

doExportUrls()方法是整个export()核心逻辑的开始

private void doExportUrls() {  
 ServiceRepository repository = ApplicationModel.*getServiceRepository*();  
 ServiceDescriptor serviceDescriptor = repository.registerService(getInterfaceClass());  
 repository.registerProvider(  
 getUniqueServiceName(),  
 ref,  
 serviceDescriptor,  
 this,  
 serviceMetadata  
 );  
 //重要，获取服务注册中心的URL地址信息

//Dubbo 支持多注册中心，因此一个服务可以同时在多个注册中心注册。所以此处是List  
 List<URL> registryURLs = ConfigValidationUtils.*loadRegistries*(this, true);  
 //Dubbo 不仅支持多注册中心，还支持多个协议暴露，因此这里遍历需要暴露的每个协议  
 for (ProtocolConfig protocolConfig : protocols) {

//构建服务的path的key  
 String pathKey = URL.*buildKey*(getContextPath(protocolConfig)  
 .map(p -> p + "/" + path)  
 .orElse(path), group, version);  
 // In case user specified path, register service one more time to map it to path.  
 repository.registerService(pathKey, interfaceClass);  
 // TODO, uncomment this line once service key is unified  
 serviceMetadata.setServiceKey(pathKey);

//重要，看方法名就能知道，该方法是针对一个协议，在一个或多个注册中心进行注册的情况，来export这个服务的  
 doExportUrlsFor1Protocol(protocolConfig, registryURLs);  
 }  
}

这里可以看出dubbo支持多注册中心和多协议，一个服务如果有多个协议那么就都需要暴露，比如同时支持 dubbo 协议和 hessian 协议，那么需要将这个服务用两种协议分别向多个注册中心（如果有多个的话）暴露注册。

* + - 1. doExportUrlsFor1Protocol ()

因为doExportUrlsFor1Protocol()方法比较长，因此只贴出来一部分：

private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs) {

//获取协议名称，没有则默认使用dubbo  
 String name = protocolConfig.getName();  
 if (StringUtils.*isEmpty*(name)) {  
 name = *DUBBO*;  
 }  
 //将一些参数填充到map中去  
 Map<String, String> map = new HashMap<String, String>();  
 map.put(*SIDE\_KEY*, *PROVIDER\_SIDE*);  
 ………此处省略N行代码  
  
 // export service

//构建出URL  
 String host = findConfigedHosts(protocolConfig, registryURLs, map);  
 Integer port = findConfigedPorts(protocolConfig, name, map);  
 URL url = new URL(name, host, port, getContextPath(protocolConfig).map(p -> p + "/" + path).orElse(path), map);  
  
 // You can customize Co9nfigurator to append extra parameters  
 if (ExtensionLoader.*getExtensionLoader*(ConfiguratorFactory.class)  
 .hasExtension(url.getProtocol())) {  
 url = ExtensionLoader.*getExtensionLoader*(ConfiguratorFactory.class)  
 .getExtension(url.getProtocol()).getConfigurator(url).configure(url);  
 }  
  
 String scope = url.getParameter(*SCOPE\_KEY*);  
 //当URL中的scope参数为”none”时，不进行暴露  
 if (!*SCOPE\_NONE*.equalsIgnoreCase(scope)) {  
  
 // export to local if the config is not remote (export to remote only when config is remote)

//当URL中的scope参数不是”remote”时，进行本地暴露  
 if (!*SCOPE\_REMOTE*.equalsIgnoreCase(scope)) {

//重要，执行本地服务暴露  
 exportLocal(url);  
 }  
 // export to remote if the config is not local (export to local only when config is local)

//重要，当URL中的scope参数不是”local”时，进行远程暴露  
 if (!*SCOPE\_LOCAL*.equalsIgnoreCase(scope)) {  
 if (CollectionUtils.*isNotEmpty*(registryURLs)) {

//遍历多个注册中心的地址(如果有多个的话)，一个一个地进行远程暴露  
 for (URL registryURL : registryURLs) {  
 //if protocol is only injvm ,not register  
 if (*LOCAL\_PROTOCOL*.equalsIgnoreCase(url.getProtocol())) {  
 continue;  
 }  
 url = url.addParameterIfAbsent(*DYNAMIC\_KEY*, registryURL.getParameter(*DYNAMIC\_KEY*));  
 //如果有监控中心，则加载dubbo的监控中心，即将监控中心的monitorUrl配置添加到url中

URL monitorUrl = ConfigValidationUtils.*loadMonitor*(this, registryURL);  
 if (monitorUrl != null) {  
 url = url.addParameterAndEncoded(*MONITOR\_KEY*, monitorUrl.toFullString());  
 }  
 ………此处省略N行代码  
 //重要，这里将需要暴露的服务对象ref封装成Invoker对象，这里用的是registryURL而不是url  
 Invoker<?> invoker = *PROXY\_FACTORY*.getInvoker(ref, (Class) interfaceClass, registryURL.addParameterAndEncoded(*EXPORT\_KEY*, url.toFullString()));

//生成Invoker的包装类，该包装类也是一个Invoker，只是在invoker的基础上扩展了ServiceConfig

//这里其实是通过SPI机制加载了JavassistProxyFactory来调用getInvoker()方法  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);

//重要，暴露服务的invoker到指定的注册中心，并生成exporter对象。因为生成invoker用的是registerUrl，因此这里其实是通过SPI加载了RegistryProtocol来调用export()方法

//一般来说，dubbo的spi加载具体哪个扩展类是根据URL路径中的parameter来确定的，但我们知道dubbo的SPI会对Protocol扩展点做特殊处理，对于Protocol扩展点接口的SPI自适应扩展，都是通过根据URL的网络协议来确定的

Exporter<?> exporter = *protocol*.export(wrapperInvoker);

//收集exporter，将这个Exporter实例存入ServiceConfig中的exporters属性中  
 exporters.add(exporter);  
 }  
 } else {

//没有服务注册中心，就采用直连的方式暴露服务，不需要配置监听   
 if (*logger*.isInfoEnabled()) {  
 *logger*.info("Export dubbo service " + interfaceClass.getName() + " to url " + url);  
 }

//注意生成invoker用的参数用的是url而不是registryURL

Invoker<?> invoker = *PROXY\_FACTORY*.getInvoker(ref, (Class) interfaceClass, url);  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);  
 //直接将invoker暴露到url，并生成exporter对象。因为没有注册中心，因此用的是url而不是registryURL，这样一来，加载的protocol扩展类就是具体url的协议对应的扩展类，如默认就是DubboProtocol

Exporter<?> exporter = *protocol*.export(wrapperInvoker);  
 exporters.add(exporter);  
 }

………此处省略N行代码

}  
 }  
 this.urls.add(url);  
}

如上所示，doExportUrlsFor1Protocol首先是将一堆属性放到map中，然后使用map生成url，接下来是分为远程暴露和本地暴露两步

先判断scope属性的暴露策略，如果是本地暴露，则调用exportLocal()，如果是远程暴露，则进行远程暴露

* + - * 1. 本地暴露exportLocal()

本地暴露就是把服务暴露到当前jvm中，它会走injvm协议，会重新创建一个injvm协议的URL代替原来的URL进行暴露

执行了本地暴露以后，如果当前的jvm中B服务要使用同一个jvm中的A服务时，就不用去注册中心获取相关信息，避免了网络间的通信；

*/\*\*  
 \* always export injvm  
 \*/*private void exportLocal(URL url) {

//重新创建一个injvm的url  
 URL local = URLBuilder.*from*(url)  
 .setProtocol(*LOCAL\_PROTOCOL*)  
 .setHost(*LOCALHOST\_VALUE*)  
 .setPort(0)  
 .build();

//Protocol 的 export 方法是标注了 @ Adaptive 注解的，因此会生成扩展代理类，然后代理类会根据 Invoker 里面的 URL 参数得知具体的协议，然后通过 Dubbo SPI 机制选择对应的实现类进行 export，而这个方法实际就会调用 InjvmProtocol # export() 方法  
 Exporter<?> exporter = *protocol*.export(  
 *PROXY\_FACTORY*.getInvoker(ref, (Class) interfaceClass, local));  
 exporters.add(exporter);  
 *logger*.info("Export dubbo service " + interfaceClass.getName() + " to local registry url : " + local);  
}

通过***PROXY\_FACTORY***的getInvoker()方法生成Invoker对象，然后再通过export方法将其生成对应的exporter对象。

这里其实会进行两次SPI服务扩展：

1. 第一次是proxyFactory动态调用扩展类javassistProxyFactory的getInvoker()方法获取Invoker对象
2. 第二次是Protocol动态调用扩展类injvmProtocol的export()方法获取export()对象

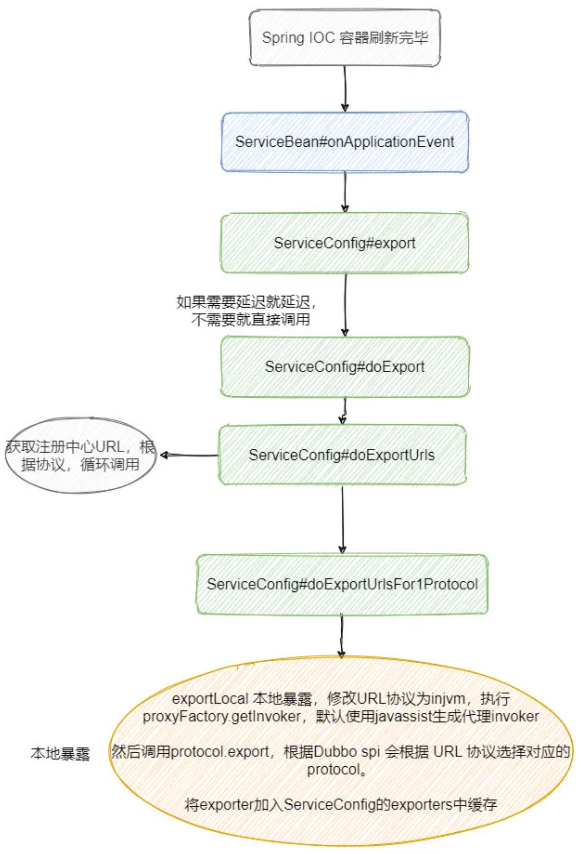
public class InjvmProtocol extends AbstractProtocol implements Protocol {

…………

@Override  
 public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {  
 return new InjvmExporter<T>(invoker, invoker.getUrl().getServiceKey(), exporterMap);  
 }

}

我们可以看到dubbo是使用javassist进行动态代理，至于 dubbo 为什么用 javassist 而不用 jdk 动态代理是因为 javassist 快，虽然ASM动态代理会更快，但是并没有比javassist快出一个数量级，反而使用难度会增加不少，因此最终选定javassist动态代理。



* + - * 1. 远程暴露

接下来是远程暴露，如果scope属性不为local，就将进行远程暴露，我们再看一下doExportUrlsFor1Protocol()方法的源码

//执行远程暴露

if (!*SCOPE\_LOCAL*.equalsIgnoreCase(scope)) {  
 if (CollectionUtils.*isNotEmpty*(registryURLs)) {

//遍历多个注册中心的地址(如果有多个的话)，一个一个地进行远程暴露  
 for (URL registryURL : registryURLs) {  
 //if protocol is only injvm ,not register  
 if (*LOCAL\_PROTOCOL*.equalsIgnoreCase(url.getProtocol())) {  
 continue;  
 }  
 url = url.addParameterIfAbsent(*DYNAMIC\_KEY*, registryURL.getParameter(*DYNAMIC\_KEY*));  
 //如果有监控中心，则加载dubbo的监控中心，即将监控中心的monitorUrl配置添加到url中

URL monitorUrl = ConfigValidationUtils.*loadMonitor*(this, registryURL);  
 if (monitorUrl != null) {  
 url = url.addParameterAndEncoded(*MONITOR\_KEY*, monitorUrl.toFullString());  
 }  
 ………此处省略N行代码  
 //重要，这里将需要暴露的服务对象ref封装成Invoker对象，这里用的是registryURL而不是url  
 Invoker<?> invoker = *PROXY\_FACTORY*.getInvoker(ref, (Class) interfaceClass, registryURL.addParameterAndEncoded(*EXPORT\_KEY*, url.toFullString()));

//生成Invoker的包装类，该包装类也是一个Invoker，只是在invoker的基础上扩展了ServiceConfig

//这里其实是通过SPI机制加载了JavassistProxyFactory来调用getInvoker()方法  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);

//重要，暴露服务的invoker到指定的注册中心，并生成exporter对象。因为生成invoker用的是registerUrl，因此这里其实是通过SPI加载了RegistryProtocol来调用export()方法

//一般来说，dubbo的spi加载具体哪个扩展类是根据URL路径中的parameter来确定的，但我们知道dubbo的SPI会对Protocol扩展点做特殊处理，对于Protocol扩展点接口的SPI自适应扩展，都是通过根据URL的网络协议来确定的

Exporter<?> exporter = *protocol*.export(wrapperInvoker);

//收集exporter，将这个Exporter实例存入ServiceConfig中的exporters属性中  
 exporters.add(exporter);  
 }  
 } else {

//没有服务注册中心，就采用直连的方式暴露服务，不需要配置监听   
 if (*logger*.isInfoEnabled()) {  
 *logger*.info("Export dubbo service " + interfaceClass.getName() + " to url " + url);  
 }

//注意生成invoker用的参数用的是url而不是registryURL

Invoker<?> invoker = *PROXY\_FACTORY*.getInvoker(ref, (Class) interfaceClass, url);  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);  
 //直接将invoker暴露到url，并生成exporter对象。因为没有注册中心，因此用的是url而不是registryURL，这样一来，加载的protocol扩展类就是具体url的协议对应的扩展类，如默认就是DubboProtocol

Exporter<?> exporter = *protocol*.export(wrapperInvoker);  
 exporters.add(exporter);  
 }  
 ………此处省略N行代码

}

从源码中我们可以看到

1. 首先判断是否有注册中心；
2. 如果有注册中心，那么遍历多个注册中心，为每个注册中心暴露一次，具体做法是：拼接生成注册中心的registerUrl；如果有监控中心，同时也会生成一个monitorUrl；接下来将实现类ref包装成invoker对象，再将invoker对象包装成export对象

我们来看一下**registryURL.addParameterAndEncoded(*EXPORT\_KEY*, url.toFullString())**生成的registerUrl长什么样子：

registry://127.0.0.1:2181/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&dubbo=2.0.2&export=dubbo://192.168.1.17:20880/com.alibaba.dubbo.demo.DemoService....

我们可以看到registerUrl将走register协议，它将我们原来dubbo协议的url拼接到参数export后面去了，所以我们可以得知，dubbo会先通过 registry 协议找到RegistryProtocol 进行 export，并且在RegistryProtocol#export()内部还会根据registryURL的export 参数的参数值（即providerUrl）再进行一次protocol.export()暴露，这样就会再次触发SPI加载机制，执行 DubboProtocol 的 export() 方法。

1. 如果没有注册中心，就采用直连的方式直接暴露了，但不会生成monitorUrl，并且invoker的url使用的是url而不是registerURL，这样一来会直接使用DubboProtocol(按url中指定的具体协议来，默认是dubbo)扩展类进行exoport

由此可见，不管有没有注册中心，dubbo执行远程暴露，最后都必将执行类似于protocol.export(new Invoker(url))这样的代码

下面我们将具体分析一下这个过程

有注册中心

有注册中心的逻辑如下

………此处省略N行代码  
 //重要，这里将需要暴露的服务对象ref封装成Invoker对象，这里用的是registryURL而不是url  
 Invoker<?> invoker = *PROXY\_FACTORY*.getInvoker(ref, (Class) interfaceClass, registryURL.addParameterAndEncoded(*EXPORT\_KEY*, url.toFullString()));

//生成Invoker的包装类，该包装类也是一个Invoker，只是在invoker的基础上扩展了ServiceConfig

//这里其实是通过SPI机制加载了JavassistProxyFactory来调用getInvoker()方法  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);

//重要，暴露服务的invoker到指定的注册中心，并生成exporter对象。因为生成invoker用的是registerUrl，因此这里其实是通过SPI加载了RegistryProtocol来调用export()方法

//一般来说，dubbo的spi加载具体哪个扩展类是根据URL路径中的parameter来确定的，但我们知道dubbo的SPI会对Protocol扩展点做特殊处理，对于Protocol扩展点接口的SPI自适应扩展，都是通过根据URL的网络协议来确定的

Exporter<?> exporter = *protocol*.export(wrapperInvoker);

………此处省略N行代码

我们可以看到，invoker中的url是registryURL，那么***protocol*.export(wrapperInvoker)**将会执行RegistryProtocol#export()  
 下面是RegistryProtocol#export()的源码：

@Override  
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {  
 URL registryUrl = getRegistryUrl(originInvoker);  
 //重要，获取invoker的registryUrl中，参数export的参数值，也就是我们前面放到*EXPORT\_KEY*中的参数值，即url，它是服务提供者的url，eg：dubbo://192.168.1.17:20880/com.alibaba.dubbo.demo.DemoService....  
 URL providerUrl = getProviderUrl(originInvoker);  
  
 final URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);  
 final OverrideListener overrideSubscribeListener = new OverrideListener(overrideSubscribeUrl, originInvoker);  
 overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);  
  
 providerUrl = overrideUrlWithConfig(providerUrl, overrideSubscribeListener);  
 //重要，开启默认的netty服务端，暴露服务生成exporter，即根据providerUrl上的指定协议暴露出 exporter   
 final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker, providerUrl);  
  
 // url to registry  
 final Registry registry = getRegistry(originInvoker);  
 final URL registeredProviderUrl = getUrlToRegistry(providerUrl, registryUrl);  
 // decide if we need to delay publish  
 boolean register = providerUrl.getParameter(*REGISTER\_KEY*, true);  
 if (register) {

//如果需要注册，向注册中心注册服务，一般是使用zookeeper作为注册中心  
 register(registryUrl, registeredProviderUrl);  
 }  
  
 // Deprecated! Subscribe to override rules in 2.6.x or before.  
 registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);  
  
 exporter.setRegisterUrl(registeredProviderUrl);  
 exporter.setSubscribeUrl(overrideSubscribeUrl);  
 //Ensure that a new exporter instance is returned every time export  
 return new DestroyableExporter<>(exporter);  
}

doLocalExport()方法

源码如下：

private <T> ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker, URL providerUrl) {  
 String key = getCacheKey(originInvoker);  
  
 return (ExporterChangeableWrapper<T>) bounds.computeIfAbsent(key, s -> {

//重要，生成一个invoker对象

//这里传进来的providreUrl是类似这样的协议dubbo://192.168.1.17:20880/com.alibaba.dubbo.demo.DemoService....  
 Invoker<?> invokerDelegate = new InvokerDelegate<>(originInvoker, providerUrl);

//重要，根据providerUrl中指定的协议进行相应的暴露，如dubbo协议  
 return new ExporterChangeableWrapper<>((Exporter<T>) protocol.export(invokerDelegate), originInvoker);  
 });  
}

dubboProtocol#export()

这里我们假设provoderUrl使用的是我们最常见的dubbo协议

那么dubboProtocol#export()的源码如下

@Override  
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {  
 URL url = invoker.getUrl();  
  
 // 这里的key是类似com.alibaba.dubbo.demo.DemoService:20880形式的  
 String key = *serviceKey*(url);

//invoker就是在这里被转化成export的  
 DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);

//将export和key关联，存放到缓存中  
 exporterMap.put(key, exporter);  
  
 //export an stub service for dispatching event  
 Boolean isStubSupportEvent = url.getParameter(*STUB\_EVENT\_KEY*, *DEFAULT\_STUB\_EVENT*);  
 Boolean isCallbackservice = url.getParameter(*IS\_CALLBACK\_SERVICE*, false);  
 if (isStubSupportEvent && !isCallbackservice) {  
 String stubServiceMethods = url.getParameter(*STUB\_EVENT\_METHODS\_KEY*);  
 if (stubServiceMethods == null || stubServiceMethods.length() == 0) {  
 if (logger.isWarnEnabled()) {  
 logger.warn(new IllegalStateException("consumer [" + url.getParameter(*INTERFACE\_KEY*) +  
 "], has set stubproxy support event ,but no stub methods founded."));  
 }  
  
 } else {  
 stubServiceMethodsMap.put(url.getServiceKey(), stubServiceMethods);  
 }  
 }  
 //重要，开启netty服务  
 openServer(url);  
 optimizeSerialization(url);  
  
 return exporter;  
}

openServer()

openServer负责创建netty服务端，dubbo中默认netty监听端口是20880，所以平常我们如果本地20880端口占用了，那么启动dubbo服务就会失败的

private void openServer(URL url) {  
 //获取服务提供者的ip+端口，端口默认是20880，并将其作为服务器实例的key，用于标识当前的服务器实例

String key = url.getAddress();  
 //client can export a service which's only for server to invoke  
 boolean isServer = url.getParameter(*IS\_SERVER\_KEY*, true);  
 if (isServer) {

//如果服务已经开启，那么server不为null，否则说明服务是第一次开启   
 ProtocolServer server = serverMap.get(key);  
 if (server == null) {  
 synchronized (this) {  
 server = serverMap.get(key);  
 if (server == null) {

//重要，第一次开启服务，加锁创建一个Server放到serverMap缓存中  
 serverMap.put(key, createServer(url));  
 }  
 }  
 } else {  
 // server supports reset, use together with override  
 server.reset(url);  
 }  
 }  
}

createServerr()

创建netty服务端

private ProtocolServer createServer(URL url) {  
 url = URLBuilder.*from*(url)  
 //服务器关闭时发送readonly事件，它是默认开启的  
 .addParameterIfAbsent(*CHANNEL\_READONLYEVENT\_SENT\_KEY*, Boolean.*TRUE*.toString())  
 // 开启心跳，这也是默认开启的  
 .addParameterIfAbsent(*HEARTBEAT\_KEY*, String.*valueOf*(*DEFAULT\_HEARTBEAT*))  
 .addParameter(*CODEC\_KEY*, DubboCodec.*NAME*)  
 .build();  
 String str = url.getParameter(*SERVER\_KEY*, *DEFAULT\_REMOTING\_SERVER*);  
  
 if (str != null && str.length() > 0 && !ExtensionLoader.*getExtensionLoader*(Transporter.class).hasExtension(str)) {  
 throw new RpcException("Unsupported server type: " + str + ", url: " + url);  
 }  
  
 ExchangeServer server;  
 try {

//重要，创建了server并完成了监听。requestHandler负责处理请求  
 server = Exchangers.*bind*(url, requestHandler);  
 } catch (RemotingException e) {  
 throw new RpcException("Fail to start server(url: " + url + ") " + e.getMessage(), e);  
 }  
  
 str = url.getParameter(*CLIENT\_KEY*);  
 if (str != null && str.length() > 0) {  
 Set<String> supportedTypes = ExtensionLoader.*getExtensionLoader*(Transporter.class).getSupportedExtensions();  
 if (!supportedTypes.contains(str)) {  
 throw new RpcException("Unsupported client type: " + str);  
 }  
 }  
  
 return new DubboProtocolServer(server);  
}

public static ExchangeServer bind(URL url, ExchangeHandler handler) throws RemotingException {  
 if (url == null) {  
 throw new IllegalArgumentException("url == null");  
 }  
 if (handler == null) {  
 throw new IllegalArgumentException("handler == null");  
 }  
 url = url.addParameterIfAbsent(Constants.*CODEC\_KEY*, "exchange");

//重要  
 return *getExchanger*(url).bind(url, handler);  
}

public static Exchanger getExchanger(URL url) {  
 String type = url.getParameter(Constants.*EXCHANGER\_KEY*, Constants.*DEFAULT\_EXCHANGER*);  
 //重要

return *getExchanger*(type);  
}  
  
public static Exchanger getExchanger(String type) {

//重要，通过spi获取Exchanger，这里是HeaderExchanger  
 return ExtensionLoader.*getExtensionLoader*(Exchanger.class).getExtension(type);  
}

接下来看HeaderExchanger#bind()，该方法获取NettyTransporter，调用bind方法，然后就是创建NettyServer实例

public class HeaderExchanger implements Exchanger {  
 public static final String *NAME* = "header";  
 @Override  
 public ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException {  
 return new HeaderExchangeClient(Transporters.*connect*(url, new DecodeHandler(new HeaderExchangeHandler(handler))), true);  
 }  
  
 @Override  
 public ExchangeServer bind(URL url, ExchangeHandler handler) throws RemotingException {  
 return new HeaderExchangeServer(Transporters.*bind*(url, new DecodeHandler(new HeaderExchangeHandler(handler))));  
 }  
}

无注册中心

下面是直连的逻辑

//没有服务注册中心，就采用直连的方式暴露服务，不需要配置监听   
 if (*logger*.isInfoEnabled()) {  
 *logger*.info("Export dubbo service " + interfaceClass.getName() + " to url " + url);  
 }

//注意生成invoker用的参数用的是url而不是registryURL

Invoker<?> invoker = *PROXY\_FACTORY*.getInvoker(ref, (Class) interfaceClass, url);  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);  
 //直接将invoker暴露到url，并生成exporter对象。因为没有注册中心，因此用的是url而不是registryURL，这样一来，加载的protocol扩展类就是具体url的协议对应的扩展类，如默认就是DubboProtocol

Exporter<?> exporter = *protocol*.export(wrapperInvoker);  
 exporters.add(exporter);  
 }  
 ………此处省略N行代码

我们可以看到，直连的invoker中，其使用的url是变量url，那么***protocol*.export(wrapperInvoker)**将根据变量url中的协议，执行指定协议对应的扩展类的export()方法，默认是dubbo协议，假设这里使用的是默认的dubbo协议，那么将执行DubboProtocol#export()，接下来的执行流程和上面一样

过滤器

看起来好像流程结束了？但Filter 到现在还没出现呢？它到底在哪里执行的呢？

我们知道，dubbo SPI支持AOP特性，如果加载的扩展点在META-INF的配置文件中的某个扩展Extension有拷贝构造函数，则判定该扩展Extension为扩展点的Wrapper类，在执行***protocol*.export(wrapperInvoker)** 的时候，由于***protocol***是：

private static final Protocol *PROTOCOL* = ExtensionLoader.*getExtensionLoader*(Protocol.class).getAdaptiveExtension();

可以看出，最终$子类在执行export()的时候,其实执行的是:

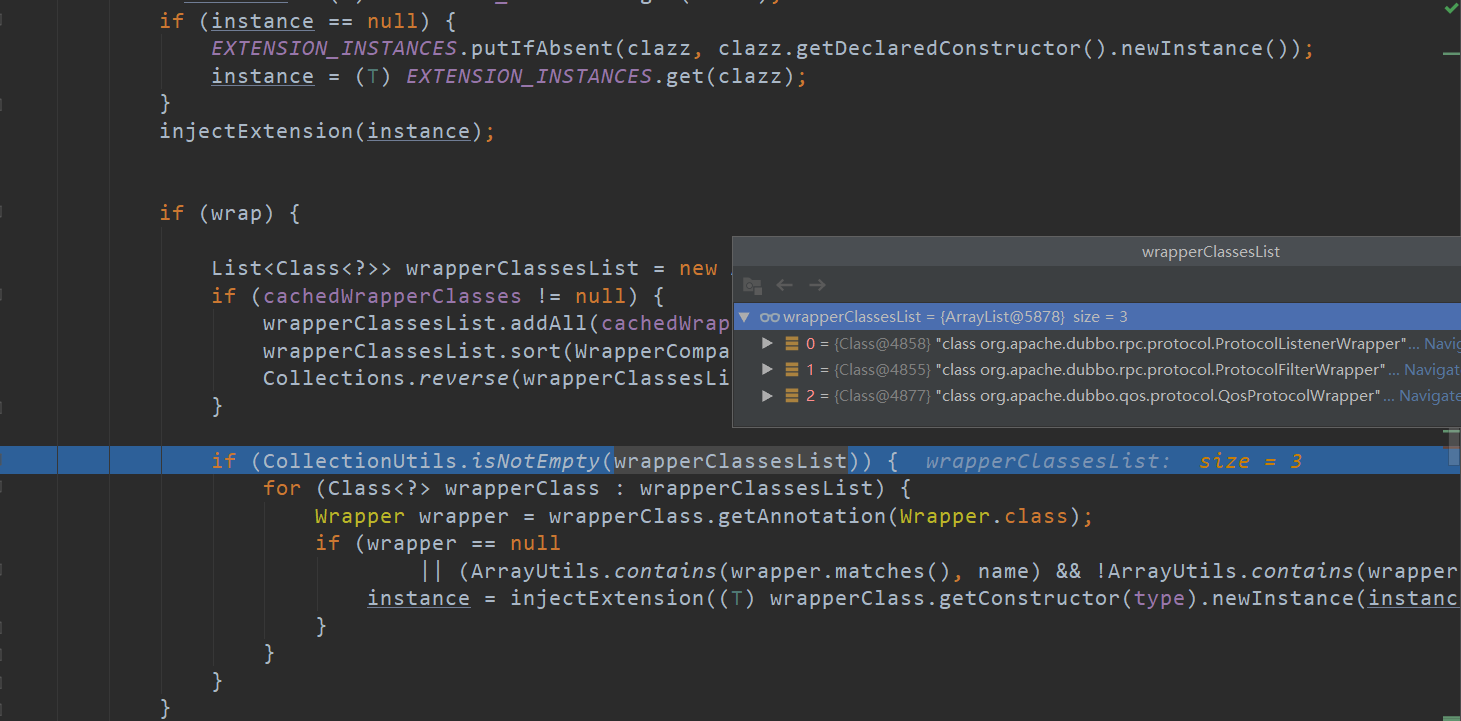
Protocol extension = (Protocol) ExtensionLoader.*getExtensionLoader*(Protocol.class).getExtension(extName);

如果有注册中心，那么这里的extName就是registry，如果是直连，那么这里的extName就是具体的协议，如dubbo

详见[$子类源码](#doller子类源码)

那么调用getExtension()就会触发protocol的AOP，最后我们会发现，RegistryProtocol(或者是DubboProtocol)外面其实包裹了三个wrapper类，这三个类和RegistryProtocol一样被声明在/META-INF/dubbo/internal/org.apache.dubbo.rpc.protocol文件内

下面是ExtensionLoader执行createExtension(“registry”)的执行截图



因此，RegistryProtocol外面会依次包裹上ProtocolListenerWapper、ProtocolFilterWrapper和QosProtocolWrapper,形成这种形式:



而在 ProtocolFilterWrapper 的 export() 里面就会把 invoker 组装上各种 Filter,调用protocol.export()的时候，会先从最外层QosProtocolWrapper的export()开始向内层调用

ProtocolFilterWrapper#export()

我们直接看ProtocolFilterWrapper#export()的源码

@Override  
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {  
 if (UrlUtils.*isRegistry*(invoker.getUrl())) {

//如果是Registry协议,就直接调用更内层的ProtocolListenerWrapper$export()  
 return protocol.export(invoker);  
 }

//如果不是Registry协议,就构建filter调用链  
 return protocol.export(*buildInvokerChain*(invoker, *SERVICE\_FILTER\_KEY*, CommonConstants.*PROVIDER*));  
}

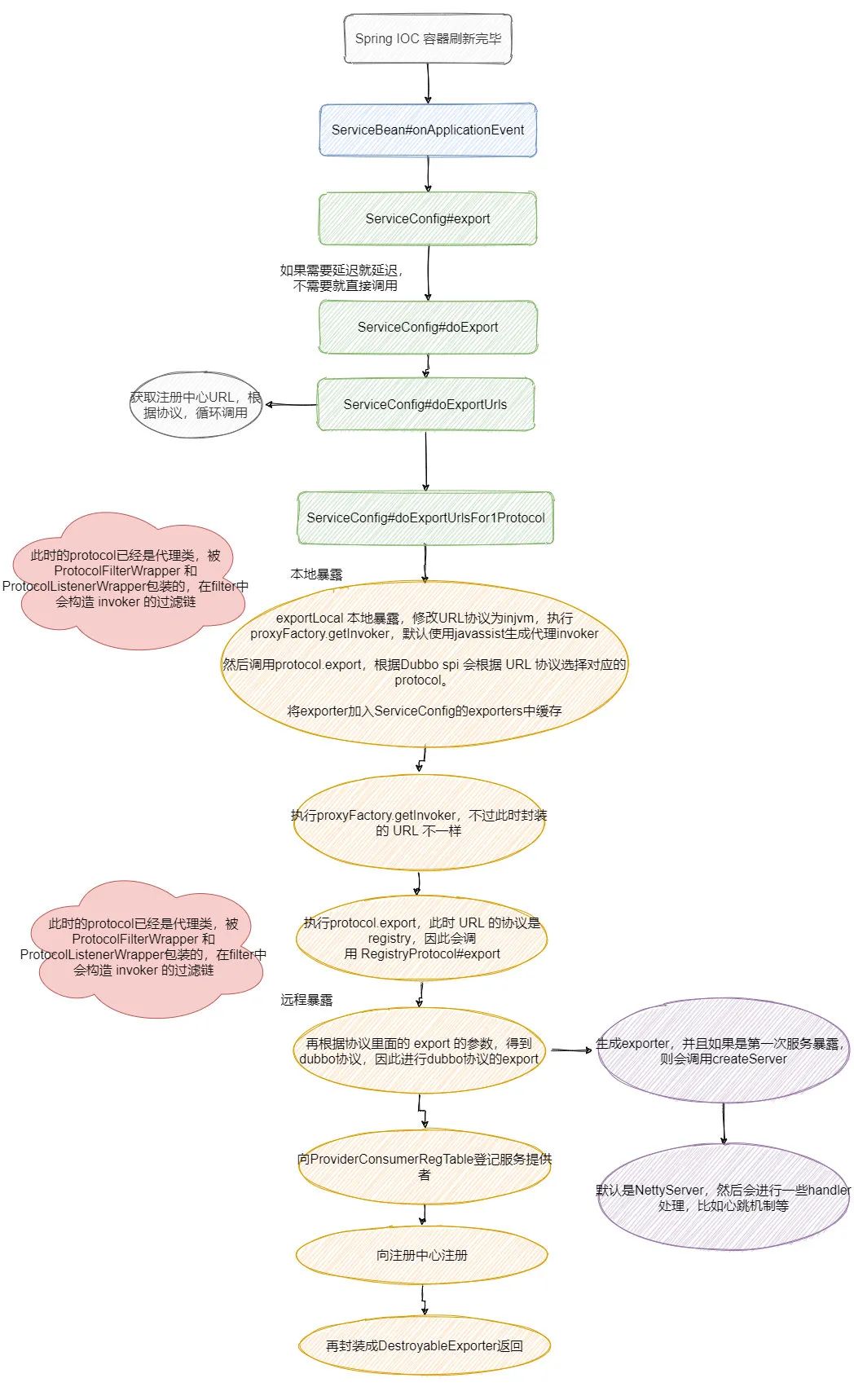
ProtocolFilterWrapper#buildInvokerChain

private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key, String group) {  
 Invoker<T> last = invoker;

//重要，获取@ActivateExtension标记的group为provider的所有Filter扩展类，一共是8个  
 List<Filter> filters = ExtensionLoader.*getExtensionLoader*(Filter.class).getActivateExtension(invoker.getUrl(), key, group);  
  
 if (!filters.isEmpty()) {  
 for (int i = filters.size() - 1; i >= 0; i--) {  
 final Filter filter = filters.get(i);  
 last = new FilterNode<T>(invoker, last, filter);  
 }  
 }  
  
 return last;  
}

* + 1. 服务暴露的流程总结

该总结是基于2.6.5版本的，不过核心流程基本一致



* 1. consumer的服务引用过程

服务暴露是provider将自己暴露出来，注册到注册中心的过程，而服务引用是向注册中心订阅服务，并生成一个代理对象来调用provider提供的服务的过程

服务引用其实有两种方式，一种是直连，一种是通过注册中心，比较常用的是第二种

* + 1. 服务引用的入口
       1. 懒汉式加载和饿汉式加载

服务引用和服务暴露不同，服务引用需要同时做两件事：

1. **首先要用<dubbo:reference>标签来声明需要引用这个服务**
2. **通过@Reference将声明的服务注入到bean中**

另外，dubbo服务引用存在懒汉式和饿汉式两种引用方式，Dubbo默认采用懒汉式加载。

1. **饿汉式加载是在ReferenceBean创建的过程，即在spring创建ReferenceBean的过程中，在initializeBean()方法初始化bean的时候，执行ReferenceBean#afterpropertiesSet()，进行加载。源码中的具体入口参见**[**服务引用之饿汉式加载入口**](#服务引用之饿汉式加载入口)
2. **懒汉式加载是在@Reference注解对ServiceImpl的bean注入依赖的时候，即在spring创建ServiceImpl的bean的过程中，执行populateBean()将ReferenceBean依赖注入ServiceImplBean的时候，进行加载。源码中的具体入口参见[服务引用之懒汉式加载入口](#服务引用之懒汉式加载入口)**

这两者的加载时机有何区别呢？我们知道，服务引用的过程中一般涉及至少两个bean，一个是ServiceImpl的bean对象，即服务消费者的service对象，以下简称ServiceImplBean。另一个是我们希望注入的依赖，由于是该依赖是来源于其他机器上的服务，因此该依赖只能是一个代理对象，然后该代理对象被ReferenceBean<>包裹。因此懒汉式加载则是发生在创建ServiceImplBean的过程中，而饿汉式加载发生在创建ReferenceBean的过程中的

Dubbo默认使用懒汉式加载，如果希望使用饿汉式服务引用，可以将<dubbo:reference>标签的init属性设为true，或者将@Reference注解的init属性设为true

* + - 1. ReferenceBean

我们先来看一下ReferenceBean源码：

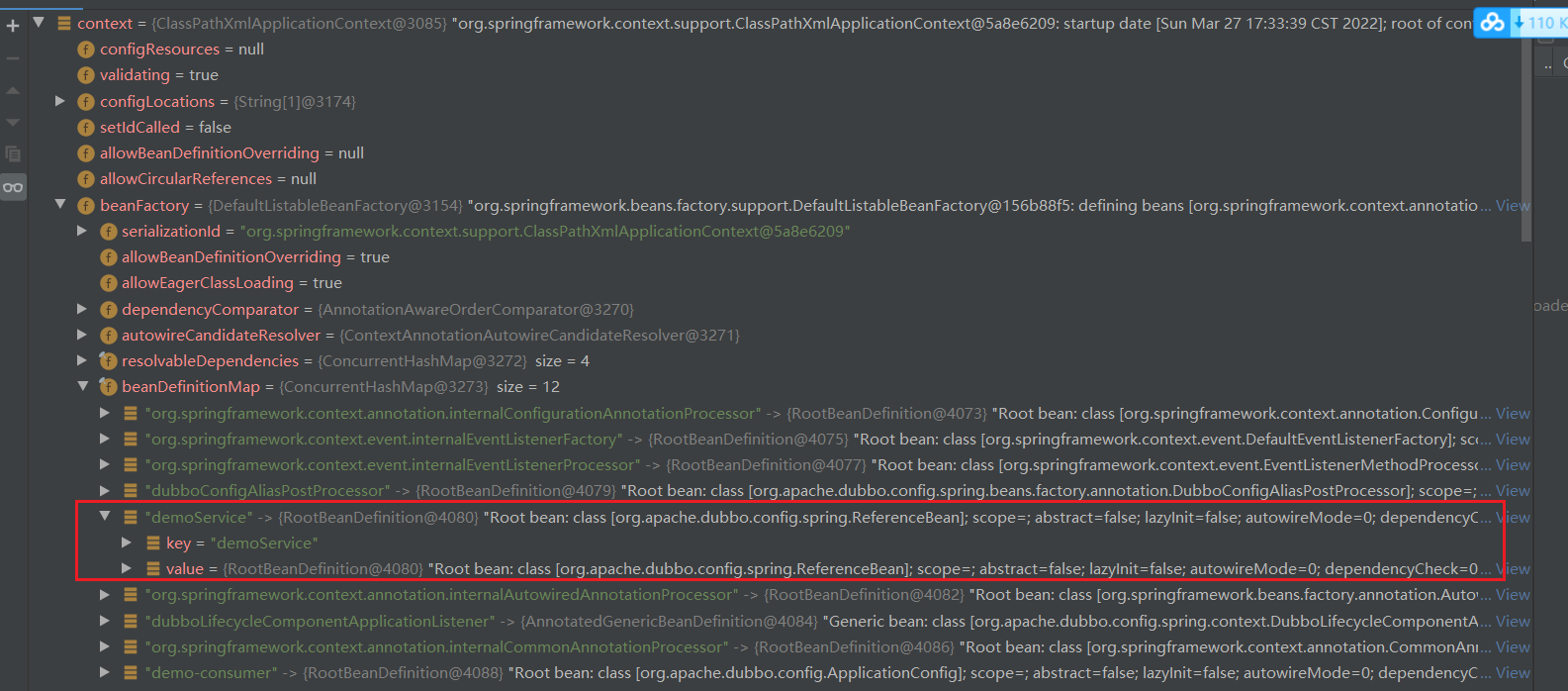
//重要，ReferenceBean实现了FactoryBean和InitializingBean

public class ReferenceBean<T> extends ReferenceConfig<T> implements FactoryBean,  
 ApplicationContextAware, InitializingBean, DisposableBean {  
 private static final long *serialVersionUID* = 213195494150089726L;  
 private transient ApplicationContext applicationContext;  
  
 public ReferenceBean() {  
 super();  
 }  
 public ReferenceBean(Reference reference) {  
 super(reference);  
 }  
  
 @Override  
 public void setApplicationContext(ApplicationContext applicationContext) {  
 this.applicationContext = applicationContext;  
 SpringExtensionFactory.*addApplicationContext*(applicationContext);  
 }  
  
 @Override  
 public Object getObject() {  
 return get();  
 }  
  
 @Override  
 public Class<?> getObjectType() {  
 return getInterfaceClass();  
 }  
  
 @Override  
 @Parameter(excluded = true)  
 public boolean isSingleton() {  
 return true;  
 }  
  
 */\*\*  
 \* Initializes there Dubbo's Config Beans before @Reference bean autowiring  
 \*/* private void prepareDubboConfigBeans() {  
 *beansOfTypeIncludingAncestors*(applicationContext, ApplicationConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, ModuleConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, RegistryConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, ProtocolConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, MonitorConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, ProviderConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, ConsumerConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, ConfigCenterBean.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, MetadataReportConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, MetricsConfig.class);  
 *beansOfTypeIncludingAncestors*(applicationContext, SslConfig.class);  
 }  
  
 @Override  
 @SuppressWarnings({"unchecked"})  
 public void afterPropertiesSet() throws Exception {  
  
 // Initializes Dubbo's Config Beans before @Reference bean autowiring  
 prepareDubboConfigBeans();  
  
 // lazy init by default.  
 if (init == null) {  
 init = false;  
 }  
  
 // eager init if necessary.  
 if (shouldInit()) {

//饿汉式加载就是在这里触发服务引用的  
 getObject();  
 }  
 }  
  
 @Override  
 public void destroy() {  
 // do nothing  
 }  
}

我们可以看到ReferenceBean实现了**FactoryBean接口和InitializingBean接口**，也就是说ReferenceBean是一个FactoryBean

在spring的ioc初始化的过程中，会解析我们配置在xml文件中<dubbo:reference id="xxx" interface="xxx"/>，解析出来的对象放在ioc容器中，如下图：



那么在spring创建demoService这个bean的时候，首先会判断它是不是FactoryBean，很显然，它是一个FactoryBean，因此其服务引入的起点是ReferenceBean的getObject()方法

* + 1. getObject()

ReferenceBean#getObject()方法是FactoryBean接口中的方法，用于获取一个bean对象，那么我们注入ServiceBeanImpl中的代理对象就是通过该方法获取的

@Override  
public Object getObject() {  
 return get();  
}

public synchronized T get() {  
 if (destroyed) {  
 throw new IllegalStateException("The invoker of ReferenceConfig(" + url + ") has already destroyed!");  
 }

//重要，ref是我们RefernceBean<>包裹的Provider的代理对象，如果没有，那么就需要通过init()初始化一个

//为了避免多线程重复初始化，该方法加了synchronized关键字  
 if (ref == null) {  
 init();  
 }  
 return ref;  
}

* + 1. init()

ReferenceConfig#init()方法，负责在服务还未引用的时候执行服务引入

public synchronized void init() {  
 if (initialized) {  
 return;  
 }  
  
 if (bootstrap == null) {  
 bootstrap = DubboBootstrap.*getInstance*();  
 bootstrap.init();  
 }  
  
 checkAndUpdateSubConfigs();  
  
 //init serivceMetadata  
 serviceMetadata.setVersion(version);  
 serviceMetadata.setGroup(group);  
 serviceMetadata.setDefaultGroup(group);  
 serviceMetadata.setServiceType(getActualInterface());  
 serviceMetadata.setServiceInterfaceName(interfaceName);  
 // TODO, uncomment this line once service key is unified  
 serviceMetadata.setServiceKey(URL.*buildKey*(interfaceName, group, version));  
  
 checkStubAndLocal(interfaceClass);  
 ConfigValidationUtils.*checkMock*(interfaceClass, this);  
  
 Map<String, String> map = new HashMap<String, String>();  
 map.put(*SIDE\_KEY*, *CONSUMER\_SIDE*);  
  
 ReferenceConfigBase.*appendRuntimeParameters*(map);  
 if (!ProtocolUtils.*isGeneric*(generic)) {  
 String revision = Version.*getVersion*(interfaceClass, version);  
 if (revision != null && revision.length() > 0) {  
 map.put(*REVISION\_KEY*, revision);  
 }  
  
 String[] methods = Wrapper.*getWrapper*(interfaceClass).getMethodNames();  
 if (methods.length == 0) {  
 *logger*.warn("No method found in service interface " + interfaceClass.getName());  
 map.put(*METHODS\_KEY*, *ANY\_VALUE*);  
 } else {  
 map.put(*METHODS\_KEY*, StringUtils.*join*(new HashSet<String>(Arrays.*asList*(methods)), *COMMA\_SEPARATOR*));  
 }  
 }  
 map.put(*INTERFACE\_KEY*, interfaceName);  
 AbstractConfig.*appendParameters*(map, getMetrics());  
 AbstractConfig.*appendParameters*(map, getApplication());  
 AbstractConfig.*appendParameters*(map, getModule());  
 // remove 'default.' prefix for configs from ConsumerConfig  
 // appendParameters(map, consumer, Constants.DEFAULT\_KEY);  
 AbstractConfig.*appendParameters*(map, consumer);  
 AbstractConfig.*appendParameters*(map, this);  
 Map<String, Object> attributes = null;  
 if (CollectionUtils.*isNotEmpty*(getMethods())) {  
 attributes = new HashMap<>();  
 for (MethodConfig methodConfig : getMethods()) {  
 AbstractConfig.*appendParameters*(map, methodConfig, methodConfig.getName());  
 String retryKey = methodConfig.getName() + ".retry";  
 if (map.containsKey(retryKey)) {  
 String retryValue = map.remove(retryKey);  
 if ("false".equals(retryValue)) {  
 map.put(methodConfig.getName() + ".retries", "0");  
 }  
 }  
 ConsumerModel.AsyncMethodInfo asyncMethodInfo = AbstractConfig.*convertMethodConfig2AsyncInfo*(methodConfig);  
 if (asyncMethodInfo != null) {  
// consumerModel.getMethodModel(methodConfig.getName()).addAttribute(ASYNC\_KEY, asyncMethodInfo);  
 attributes.put(methodConfig.getName(), asyncMethodInfo);  
 }  
 }  
 }  
  
 String hostToRegistry = ConfigUtils.*getSystemProperty*(*DUBBO\_IP\_TO\_REGISTRY*);  
 if (StringUtils.*isEmpty*(hostToRegistry)) {  
 hostToRegistry = NetUtils.*getLocalHost*();  
 } else if (*isInvalidLocalHost*(hostToRegistry)) {  
 throw new IllegalArgumentException("Specified invalid registry ip from property:" + *DUBBO\_IP\_TO\_REGISTRY* + ", value:" + hostToRegistry);  
 }  
 map.put(*REGISTER\_IP\_KEY*, hostToRegistry);  
  
 serviceMetadata.getAttachments().putAll(map);  
  
 ServiceRepository repository = ApplicationModel.*getServiceRepository*();  
 ServiceDescriptor serviceDescriptor = repository.registerService(interfaceClass);  
 repository.registerConsumer(  
 serviceMetadata.getServiceKey(),  
 attributes,  
 serviceDescriptor,  
 this,  
 null,  
 serviceMetadata);

//前面一大堆都是负责设置参数到map中  
 //重要，根据map创建代理对象  
 ref = createProxy(map);  
  
 serviceMetadata.setTarget(ref);  
 serviceMetadata.addAttribute(*PROXY\_CLASS\_REF*, ref);  
 repository.lookupReferredService(serviceMetadata.getServiceKey()).setProxyObject(ref);  
  
 initialized = true;  
  
 // dispatch a ReferenceConfigInitializedEvent since 2.7.4  
 dispatch(new ReferenceConfigInitializedEvent(this, invoker));  
}

init()方法先组装了参数，并用组装出的map创建代理对象

* + - 1. createProxy()

ReferenceConfig#createProxy()方法，负责使用map中的参数创建代理对象

private T createProxy(Map<String, String> map) {  
//重要，判断是否为本地引用

if (shouldJvmRefer(map)) {

//重要，如果是本地引用，那么这个场景就很简单了，直接构建一个InJvm协议的url   
 URL url = new URL(*LOCAL\_PROTOCOL*, *LOCALHOST\_VALUE*, 0, interfaceClass.getName()).addParameters(map);  
 //重要，那么根据构建的url调用InjvmProtocol#refer()方法，去之前的exporterMap中获取invoker对象

invoker = *REF\_PROTOCOL*.refer(interfaceClass, url);  
 if (*logger*.isInfoEnabled()) {  
 *logger*.info("Using injvm service " + interfaceClass.getName());  
 }  
 } else {

//重要，如果是远程引用

urls.clear();

//重要，先判断有没有指定url，如果指定了url，那么可能是点对点直连的url，也可能是注册中心的url  
 if (url != null && url.length() > 0) {

//注册中心可能是多个，用分割符切分一下  
 String[] us = *SEMICOLON\_SPLIT\_PATTERN*.split(url);  
 if (us != null && us.length > 0) {  
 for (String u : us) {  
 URL url = URL.*valueOf*(u);  
 if (StringUtils.*isEmpty*(url.getPath())) {  
 url = url.setPath(interfaceName);  
 }  
 if (UrlUtils.*isRegistry*(url)) {

//重要，如果url是注册中心，将map参数作为refer的属性值附在url后面，并将url加入urls  
 urls.add(url.addParameterAndEncoded(*REFER\_KEY*, StringUtils.*toQueryString*(map)));  
 } else {

//重要，如果是点对点直连，合并url，并将url加入urls  
 urls.add(ClusterUtils.*mergeUrl*(url, map));  
 }  
 }  
 }  
 } else { // assemble URL from register center's configuration  
 //重要，如果使用者没有指定url，那么肯定就是走注册中心了，这也是最常见的情况  
 if (!*LOCAL\_PROTOCOL*.equalsIgnoreCase(getProtocol())) {  
 checkRegistry();

//重要，获取服务注册中心的URL地址信息

//Dubbo 支持多注册中心。所以此处是List  
 List<URL> us = ConfigValidationUtils.*loadRegistries*(this, false);  
 if (CollectionUtils.*isNotEmpty*(us)) {  
 for (URL u : us) {  
 URL monitorUrl = ConfigValidationUtils.*loadMonitor*(this, u);  
 if (monitorUrl != null) {

//添加监控中心  
 map.put(*MONITOR\_KEY*, URL.*encode*(monitorUrl.toFullString()));  
 }

//重要，将map参数作为refer的属性值附在注册中心的url后面，并将url加入urls  
 urls.add(u.addParameterAndEncoded(*REFER\_KEY*, StringUtils.*toQueryString*(map)));  
 }  
 }  
 if (urls.isEmpty()) {  
 throw new IllegalStateException("No such any registry to reference " + interfaceName + " on the consumer " + NetUtils.*getLocalHost*() + " use dubbo version " + Version.*getVersion*() + ", please config <dubbo:registry address=\"...\" /> to your spring config.");  
 }  
 }  
 }  
 //下面的代码主要是为了合并多个invoker  
 if (urls.size() == 1) {

//重要，如果只有一个url，那么只会有一个invoker，这时候的URL可能是这样子：

// registry://username@:password@127.0.0.1:20880/com.alibaba.dubbo.registry.RegistryService?refer=.(经过编码的key，value键值队集合).&protocol=remote&owner=lvyanfeng&...

//那么就将直接调用RegistryProtocol#refer()方法生成invoker  
 invoker = *REF\_PROTOCOL*.refer(interfaceClass, urls.get(0));  
 } else {

//重要，如果是多注册中心  
 List<Invoker<?>> invokers = new ArrayList<Invoker<?>>();  
 URL registryURL = null;  
 for (URL url : urls) {

//重要，对url们挨个调用refer()方法  
 invokers.add(*REF\_PROTOCOL*.refer(interfaceClass, url));  
 if (UrlUtils.*isRegistry*(url)) {  
 registryURL = url; // use last registry url  
 }  
 }

if (registryURL != null) { // registry url is available  
 //如果registryURL不为空，则说明urls中至少存在一个以上的注册中心，这导致同一个服务可能出现多个invoker，必须处理

// for multi-subscription scenario, use 'zone-aware' policy by default  
 URL u = registryURL.addParameterIfAbsent(*CLUSTER\_KEY*, ZoneAwareCluster.*NAME*);  
 // The invoker wrap relation would be like: ZoneAwareClusterInvoker(StaticDirectory) -> FailoverClusterInvoker(RegistryDirectory, routing happens here) -> Invoker

//通过Cluster对多个Invoker进行合并，只暴露出一个invoker便于调用  
 invoker = *CLUSTER*.join(new StaticDirectory(u, invokers));  
 } else { // not a registry url, must be direct invoke.

//如果registryURL是空，代表urls中没有注册中心  
 invoker = *CLUSTER*.join(new StaticDirectory(invokers));  
 }  
 }  
 }  
  
 if (shouldCheck() && !invoker.isAvailable()) {  
 throw new IllegalStateException("Failed to check the status of the service "  
 + interfaceName  
 + ". No provider available for the service "  
 + (group == null ? "" : group + "/")  
 + interfaceName +  
 (version == null ? "" : ":" + version)  
 + " from the url "  
 + invoker.getUrl()  
 + " to the consumer "  
 + NetUtils.*getLocalHost*() + " use dubbo version " + Version.*getVersion*());  
 }  
 if (*logger*.isInfoEnabled()) {  
 *logger*.info("Refer dubbo service " + interfaceClass.getName() + " from url " + invoker.getUrl());  
 }  
 */\*\*  
 \** ***@since*** *2.7.0  
 \* ServiceData Store  
 \*/* String metadata = map.get(*METADATA\_KEY*);  
 WritableMetadataService metadataService = WritableMetadataService.*getExtension*(metadata == null ? *DEFAULT\_METADATA\_STORAGE\_TYPE* : metadata);  
 if (metadataService != null) {  
 URL consumerURL = new URL(*CONSUMER\_PROTOCOL*, map.remove(*REGISTER\_IP\_KEY*), 0, map.get(*INTERFACE\_KEY*), map);  
 metadataService.publishServiceDefinition(consumerURL);  
 }  
 //重要，使用invoker创建代理  
 return (T) *PROXY\_FACTORY*.getProxy(invoker);  
}

* + - * 1. shouldJvmRefer()

ReferenceConfig#shouldJvmRefer()，用于判断当前引用的服务是否为本地服务

protected boolean shouldJvmRefer(Map<String, String> map) {  
 //先用map临时构建一个url

URL tmpUrl = new URL("temp", "localhost", 0, map);  
 boolean isJvmRefer;

//先看看ReferenceConfig有没有指定使用InJvm协议  
 if (isInjvm() == null) {  
 //如果ReferenceConfig没有指定，再看有没有url，如果有url则认定不是Injvm  
 if (url != null && url.length() > 0) {  
 isJvmRefer = false;  
 } else {  
 //最后再用map构建的tmpUrl看一看是不是Injvm  
 isJvmRefer = InjvmProtocol.*getInjvmProtocol*().isInjvmRefer(tmpUrl);  
 }  
 } else {  
 isJvmRefer = isInjvm();  
 }  
 return isJvmRefer;  
}

* + - * 1. InjvmProtocol#refer()本地引用

该方法在AbstractProtocol#refer()中，

@Override  
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {

// 最终由InjvmProtocol.protocolBindingRefer实现  
 return new AsyncToSyncInvoker<>(protocolBindingRefer(type, url));  
}

我们再来看看InjvmProtocol#protocolBindingRefer()方法

@Override  
public <T> Invoker<T> protocolBindingRefer(Class<T> serviceType, URL url) throws RpcException {

// 这个exporterMap很重要，就是之前我们injvm服务暴露时，用于存放暴露后的exporter的地方，这里直接把这个map当做构造参数传入  
 return new InjvmInvoker<T>(serviceType, url, url.getServiceKey(), exporterMap);  
}

我们可以看到，InjvmProtocol#refer()主要是生成一个InjvmInvoker对象

* + - * 1. RegistryProtocol#refer()从注册中心引用

如果是远程引用并且urls中有注册中心，那么就会调用RegistryProtocol#refer()方法

@Override  
@SuppressWarnings("unchecked")  
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {

//获取zookeeperRegistry注册中心对象  
 url = getRegistryUrl(url);  
 Registry registry = registryFactory.getRegistry(url);

//判断当前要引用的服务是否为注册中心的RegistryService服务，如果是，那么跳过注册中心  
 if (RegistryService.class.equals(type)) {  
 return proxyFactory.getInvoker((T) registry, type, url);  
 }  
  
 // group="a,b" or group="\*"  
 Map<String, String> qs = StringUtils.*parseQueryString*(url.getParameterAndDecoded(*REFER\_KEY*));  
 String group = qs.get(*GROUP\_KEY*);  
 if (group != null && group.length() > 0) {  
 if ((*COMMA\_SPLIT\_PATTERN*.split(group)).length > 1 || "\*".equals(group)) {

//重要，通过doRefer()从注册中心进行远程引用  
 return doRefer(getMergeableCluster(), registry, type, url);  
 }  
 }  
 return doRefer(cluster, registry, type, url);  
}

可以看到，真正执行refer操作的是doRefer()方法

doRefer()

我们来看一下RegistryProtocol#doRefer()的源码

private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {

//创建一个Directory对象，并塞入注册中心registry对象和protocol对象

RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);  
 directory.setRegistry(registry);  
 directory.setProtocol(protocol);  
 // all attributes of REFER\_KEY  
 Map<String, String> parameters = new HashMap<String, String>(directory.getUrl().getParameters());

//生成服务消费者的url  
 URL subscribeUrl = new URL(*CONSUMER\_PROTOCOL*, parameters.remove(*REGISTER\_IP\_KEY*), 0, type.getName(), parameters);  
 if (!*ANY\_VALUE*.equals(url.getServiceInterface()) && url.getParameter(*REGISTER\_KEY*, true)) {  
 directory.setRegisteredConsumerUrl(getRegisteredConsumerUrl(subscribeUrl, url));

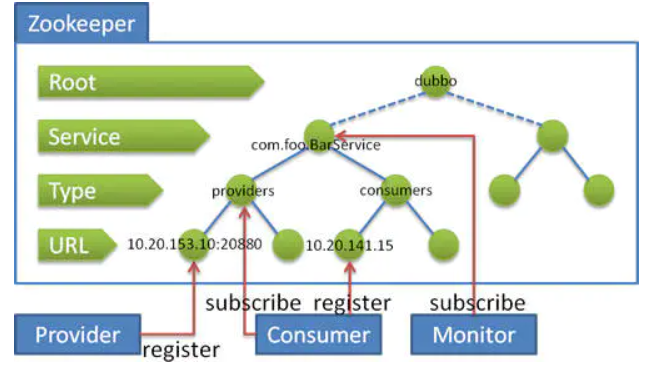
//重要，如果register参数设置为true的话，就会将consumer也注册到注册中心，即将自身这个consumer的信息注册到注册中心去。说白了就是去zookeeper的consumer节点下创建新节点  
 registry.register(directory.getRegisteredConsumerUrl());  
 }  
 directory.buildRouterChain(subscribeUrl);

//重要，这里通过RegistryDirectory订阅服务，不过本质上还是通过registry来订阅。即向注册中心订阅了 providers 节点、 configurators 节点 和 routers 节点  
 directory.subscribe(subscribeUrl.addParameter(*CATEGORY\_KEY*, *PROVIDERS\_CATEGORY* + "," + *CONFIGURATORS\_CATEGORY* + "," + *ROUTERS\_CATEGORY*));  
 //重要，服务接口可能是集群部署，然后都注册到同一个注册中心中，我们需要将获取到的这个接口的多个服务提供者，聚合成一个invoker  
 Invoker invoker = cluster.join(directory);  
 return invoker;  
}

首先生成一个RegistryDirectory ，RegistryDirectory 其实很形象，它的中文意思是目录，形象地表现出这个目录下可能有多个服务提供者。RegistryDirectory中被塞入了注册中心实例对象registry，同时RegistryDirectory自身也实现了NotifyListener 接口，因此注册中心的监听其实是靠这家伙来处理的。

接着向注册中心注册consumer自身的信息，然后再向注册中心订阅 providers / configurators / routers 节点，如果这些被订阅节点发生变化，则推送通知给consumer节点，比如：

作为consumer我们监听了/dubbo/com.foo.BaseService/providers目录，今后如果providers下面的信息有变更，比如说IP地址变更了，那么zk会首先在Watcher（监听器）中触发到变更信息，这样Dubbo就能借助于zk的推送变更消息来作出相应的响应来。如下图：



最后由于一个服务可能有多个提供者，因此通过cluster包装一下，得到invoker对象

notify()

上面我们说过，RegistryDirectory实现了NotifyListener 接口，一旦接收到zookeeper的消息推送，将触发NotifyListener 接口的notify()方法

RegistryDirectory#notify() -> refreshOverrideAndInvoker() -> refreshInvoker() -> toInvokers() ->

invoker = new InvokerDelegate<>(protocol.refer(serviceType, url), url, providerUrl)

这里的protocol将根据具体的协议来，通常默认是dubbo协议

AbstractProtocol#refer()

DubboProtocol的refer()实际上是AbstractProtocol#refer()

@Override  
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {  
 return new AsyncToSyncInvoker<>(protocolBindingRefer(type, url));  
}  
  
protected abstract <T> Invoker<T> protocolBindingRefer(Class<T> type, URL url) throws RpcException;

DubboProtocl# protocolBindingRefer()

@Override  
public <T> Invoker<T> protocolBindingRefer(Class<T> serviceType, URL url) throws RpcException {  
 optimizeSerialization(url);  
  
 //重要，创建invoker，重点看getClients()

DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url), invokers);  
 invokers.add(invoker);  
  
 return invoker;  
}

DubboProtocl# getClients()

private ExchangeClient[] getClients(URL url) {  
 // whether to share connection  
 boolean useShareConnect = false;  
 //获取连接数，默认为0表示未设置，未设置则使用共享连接，否则每个服务一个连接  
 int connections = url.getParameter(*CONNECTIONS\_KEY*, 0);  
 List<ReferenceCountExchangeClient> shareClients = null;  
 // if not configured, connection is shared, otherwise, one connection for one service  
 if (connections == 0) {  
 useShareConnect = true;  
 */\*\*  
 \* The xml configuration should have a higher priority than properties.  
 \*/* String shareConnectionsStr = url.getParameter(*SHARE\_CONNECTIONS\_KEY*, (String) null);  
 connections = Integer.*parseInt*(StringUtils.*isBlank*(shareConnectionsStr) ? ConfigUtils.*getProperty*(*SHARE\_CONNECTIONS\_KEY*,*DEFAULT\_SHARE\_CONNECTIONS*) : shareConnectionsStr);  
 shareClients = getSharedClient(url, connections);  
 }  
  
 ExchangeClient[] clients = new ExchangeClient[connections];  
 for (int i = 0; i < clients.length; i++) {  
 if (useShareConnect) {

//得到共享客户端

clients[i] = shareClients.get(i);  
 } else {

//重要，初始化新客户端

clients[i] = initClient(url);  
 }  
 }

//使用返回clients构建出invoker

return clients;  
}

DubboProtocl# initClient()

private ExchangeClient initClient(URL url) {

//获取客户端类型，默认为Netty  
 String str = url.getParameter(*CLIENT\_KEY*, url.getParameter(*SERVER\_KEY*, *DEFAULT\_REMOTING\_CLIENT*));  
 url = url.addParameter(*CODEC\_KEY*, DubboCodec.*NAME*);  
 //默认开启心跳  
 url = url.addParameterIfAbsent(*HEARTBEAT\_KEY*, String.*valueOf*(*DEFAULT\_HEARTBEAT*));  
 //BIO不允许使用，因为它存在严重的性能问题.  
 if (str != null && str.length() > 0 && !ExtensionLoader.*getExtensionLoader*(Transporter.class).hasExtension(str)) {  
 throw new RpcException("Unsupported client type: " + str + "," +  
 " supported client type is " + StringUtils.*join*(ExtensionLoader.*getExtensionLoader*(Transporter.class).getSupportedExtensions(), " "));  
 }  
 ExchangeClient client;  
 try {  
 // connection should be lazy  
 if (url.getParameter(*LAZY\_CONNECT\_KEY*, false)) {  
 client = new LazyConnectExchangeClient(url, requestHandler);  
 } else {

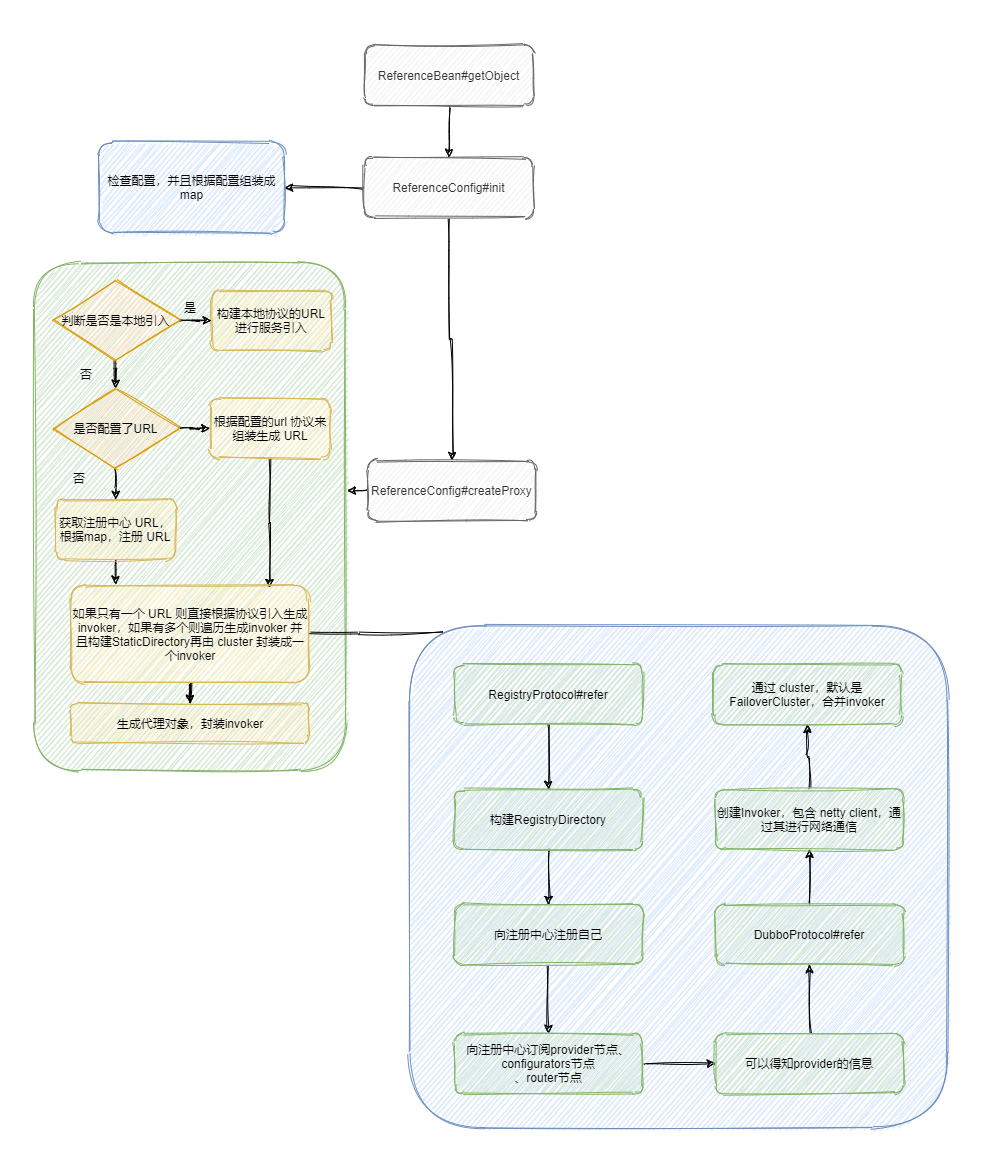
//重要，远程连接，属性client是HeaderExchangeClient类型的，里面封装了NettyClient  
 client = Exchangers.*connect*(url, requestHandler);  
 }  
 } catch (RemotingException e) {  
 throw new RpcException("Fail to create remoting client for service(" + url + "): " + e.getMessage(), e);  
 }  
 return client;  
}

* + - * 1. ProxyFactory #getProxy()

在getProxy()方法里面，其实就是判断当前服务接口是不是一个泛化接口（可以去了解一下dubbo的泛化，就是另外一种调用接口的方式而已，可以不提供服务接口），如果是的话，就做特殊处理，否则就把代理对象直接返回

public <T> T getProxy(Invoker<T> invoker) throws RpcException {  
 T proxy = proxyFactory.getProxy(invoker);  
 if (GenericService.class != invoker.getInterface()) {  
 URL url = invoker.getUrl();  
 String stub = url.getParameter(*STUB\_KEY*, url.getParameter(*LOCAL\_KEY*));  
 if (ConfigUtils.*isNotEmpty*(stub)) {  
 Class<?> serviceType = invoker.getInterface();  
 if (ConfigUtils.*isDefault*(stub)) {  
 if (url.hasParameter(*STUB\_KEY*)) {  
 stub = serviceType.getName() + "Stub";  
 } else {  
 stub = serviceType.getName() + "Local";  
 }  
 }  
 try {  
 Class<?> stubClass = ReflectUtils.*forName*(stub);  
 if (!serviceType.isAssignableFrom(stubClass)) {  
 throw new IllegalStateException("The stub implementation class " + stubClass.getName() + " not implement interface " + serviceType.getName());  
 }  
 try {  
 Constructor<?> constructor = ReflectUtils.*findConstructor*(stubClass, serviceType);  
 proxy = (T) constructor.newInstance(new Object[]{proxy});  
 //export stub service  
 URLBuilder urlBuilder = URLBuilder.*from*(url);  
 if (url.getParameter(*STUB\_EVENT\_KEY*, *DEFAULT\_STUB\_EVENT*)) {  
 urlBuilder.addParameter(*STUB\_EVENT\_METHODS\_KEY*, StringUtils.*join*(Wrapper.*getWrapper*(proxy.getClass()).getDeclaredMethodNames(), ","));  
 urlBuilder.addParameter(*IS\_SERVER\_KEY*, Boolean.*FALSE*.toString());  
 try {  
 export(proxy, (Class) invoker.getInterface(), urlBuilder.build());  
 } catch (Exception e) {  
 *LOGGER*.error("export a stub service error.", e);  
 }  
 }  
 } catch (NoSuchMethodException e) {  
 throw new IllegalStateException("No such constructor \"public " + stubClass.getSimpleName() + "(" + serviceType.getName() + ")\" in stub implementation class " + stubClass.getName(), e);  
 }  
 } catch (Throwable t) {  
 *LOGGER*.error("Failed to create stub implementation class " + stub + " in consumer " + NetUtils.*getLocalHost*() + " use dubbo version " + Version.*getVersion*() + ", cause: " + t.getMessage(), t);  
 // ignore  
 }  
 }  
 }  
 return proxy;  
}

* + 1. 服务引用总结

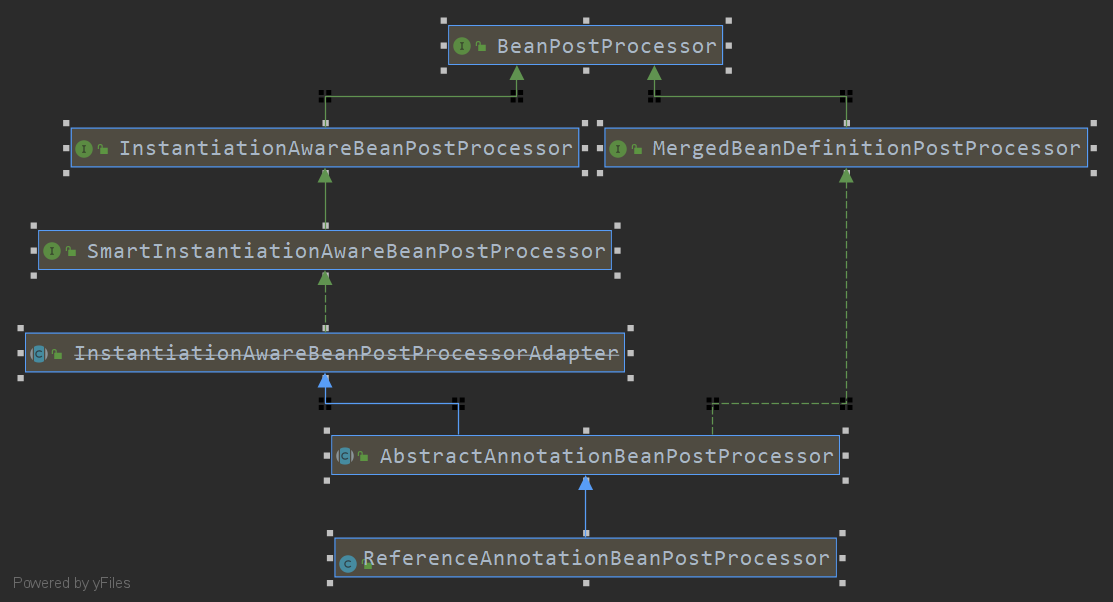


* + 1. @Reference注解（注入dubbo服务依赖）

通过对服务引入过程源码的学习，我们已经了解到服务提供者的代理是怎么来的了，接下来Dubbo将通过@Reference注解将服务提供者的代理注入bean中。在dubbo依赖的注入过程中，会发生懒汉式服务引用

负责扫描注解并对@Reference标记的字段执行注入操作的类是ReferenceAnnotationBeanPostProcessor，它的主要加载逻辑在父类AbstractAnnotationBeanPostProcessor中完成，而这个父类他有一个”爷爷”类辈的接口，叫SmartInstantiationAwareBeanPostProcessor

是不是很眼熟？没错，它在我们AOP源码中也出现过，我们先不管他，后面再看它。我们先重点关注MergedBeanDefinitionPostProcessor接口，我们来看一下他们的类继承图：



* + - * 1. MergedBeanDefinitionPostProcessor#postProcessMergedBeanDefinition()接口的调用时机

MergedBeanDefinitionPostProcessor在spring的doCreateBean()中被调用，那让我们回顾一下doCreateBean()的源码：

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)  
 throws BeanCreationException {  
  
 // Instantiate the bean.  
 BeanWrapper instanceWrapper = null;  
 if (mbd.isSingleton()) {

//如果是单例，那么就从FactoryBean缓存中清除当前bean的缓存，防止重复创建  
 instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);  
 }  
 if (instanceWrapper == null) {

//（重点），实例化bean，开辟内存空间，创建bean对象  
 instanceWrapper = createBeanInstance(beanName, mbd, args);  
 }  
 final Object bean = instanceWrapper.getWrappedInstance();  
 Class<?> beanType = instanceWrapper.getWrappedClass();  
 if (beanType != NullBean.class) {  
 mbd.resolvedTargetType = beanType;  
 }  
  
 // Allow post-processors to modify the merged bean definition.  
 synchronized (mbd.postProcessingLock) {  
 if (!mbd.postProcessed) {  
 try {

**//注意看这里，在这里spring调用了MergedBeanDefinitionPostProcessor接口的postProcessMergedBeanDefinition()方法**  
 applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
 "Post-processing of merged bean definition failed", ex);  
 }  
 mbd.postProcessed = true;  
 }  
 }  
  
 // 判断当前bean是否需要提前曝光，如果是单例、允许循环依赖、bean正在创建，则需要提前曝光  
 boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&  
 isSingletonCurrentlyInCreation(beanName));  
 if (earlySingletonExposure) {  
 if (logger.isDebugEnabled()) {  
 logger.debug("Eagerly caching bean '" + beanName +  
 "' to allow for resolving potential circular references");  
 }

// （重点），在bean初始化完成之前，添加三级缓存，同时移除二级缓存  
 addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));  
 }  
  
 // Initialize the bean instance.  
 Object exposedObject = bean;  
 try {

//（重点），填充bean的属性，如果存在循环依赖，则会递归创建所依赖的bean   
 populateBean(beanName, mbd, instanceWrapper);

//（重点）执行初始化逻辑   
 exposedObject = initializeBean(beanName, exposedObject, mbd);  
 }  
 catch (Throwable ex) {  
 if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {  
 throw (BeanCreationException) ex;  
 }  
 else {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);  
 }  
 }  
  
 if (earlySingletonExposure) {  
 Object earlySingletonReference = getSingleton(beanName, false);  
 if (earlySingletonReference != null) {  
 if (exposedObject == bean) {  
 exposedObject = earlySingletonReference;  
 }  
 else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {  
 String[] dependentBeans = getDependentBeans(beanName);  
 Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);  
 for (String dependentBean : dependentBeans) {  
 if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {  
 actualDependentBeans.add(dependentBean);  
 }  
 }  
 if (!actualDependentBeans.isEmpty()) {  
 throw new BeanCurrentlyInCreationException(beanName,  
 "Bean with name '" + beanName + "' has been injected into other beans [" +  
 StringUtils.*collectionToCommaDelimitedString*(actualDependentBeans) +  
 "] in its raw version as part of a circular reference, but has eventually been " +  
 "wrapped. This means that said other beans do not use the final version of the " +  
 "bean. This is often the result of over-eager type matching - consider using " +  
 "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");  
 }  
 }  
 }  
 }  
  
 // Register bean as disposable.  
 try {  
 registerDisposableBeanIfNecessary(beanName, bean, mbd);  
 }  
 catch (BeanDefinitionValidationException ex) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);  
 }  
 return exposedObject;  
}

从上述源码中我们可以看到，doCreateBean()做了以下几件事：

1. createBeanInstance()实例化bean
2. applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);即调用MergedBeanDefinitionPostProcessor#postProcessMergedBeanDefinition()方法
3. addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean)) 在实例化以后，填充属性之前，将工厂添加到三级缓存，同时移除二级缓存
4. populateBean()填充bean属性
5. initializeBean()执行初始化逻辑

让我们来看看doCreateBean()的第(2)步applyMergedBeanDefinitionPostProcessors()方法的源码：

protected void applyMergedBeanDefinitionPostProcessors(RootBeanDefinition mbd, Class<?> beanType, String beanName) {  
 for (MergedBeanDefinitionPostProcessor processor : getBeanPostProcessorCache().mergedDefinition) {  
 processor.postProcessMergedBeanDefinition(mbd, beanType, beanName);  
 }  
}

可以看到，该方法主要是获取所有的MergedBeanDefinitionPostProcessor接口的实现，并调用该接口的postProcessMergedBeanDefinition()方法

也就是说每个bean创建的过程中都会调用postProcessMergedBeanDefinition()方法，执行：

processor.postProcessMergedBeanDefinition(mbd, beanType, beanName);

就是这里的processor就是MergedBeanDefinitionPostProcessor接口的实现类

* + - * 1. AbstractAnnotationBeanPostProcessor#postProcessMergedBeanDefinition()扫描注解标记的属性和方法

再回到MergedBeanDefinitionPostProcessor接口的具体实现类AbstractAnnotationBeanPostProcessor中来，我们看一看它是怎么实现postProcessMergedBeanDefinition()方法的，源码如下：

public abstract class AbstractAnnotationBeanPostProcessor extends  
 InstantiationAwareBeanPostProcessorAdapter implements MergedBeanDefinitionPostProcessor, PriorityOrdered,  
 BeanFactoryAware, BeanClassLoaderAware, EnvironmentAware, DisposableBean {

//重要，当前的BeanPostProcessor需要扫描的注解，支持多个

private final Class<? extends Annotation>[] annotationTypes;

public AbstractAnnotationBeanPostProcessor(Class<? extends Annotation>... annotationTypes) {

//重要，构造方法，annotationTypes是需要扫描的注解类型，支持多个注解的同时扫描

//其子类ReferenceAnnotationBeanPostProcessor通过构造函数传参来指定扫描@Reference注解，子类构造函数如下

// public ReferenceAnnotationBeanPostProcessor() {

// super(Reference.class, com.alibaba.dubbo.config.annotation.Reference.class);

// }  
 Assert.*notEmpty*(annotationTypes, "The argument of annotations' types must not empty");  
 this.annotationTypes = annotationTypes;  
}

...此处省略N行代码

@Override  
public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?> beanType, String beanName) {  
 if (beanType != null) {

//查找当前bean中所有被指定注解标记的字段和方法，并将其包装为InjectionMetadata  
 InjectionMetadata metadata = findInjectionMetadata(beanName, beanType, null);  
 metadata.checkConfigMembers(beanDefinition);  
 }  
}

...此处省略N行代码

}

我们可以看到，上述代码扫描并解析了当前正在创建的bean中的，所有被@Reference注解标记的方法和属性，并通过findInjectionMetadata()将这些被标记的方法和属性包装成AnnotatedInjectionMetadata对象。在findInjectionMetadata()方法内，将解析后的AnnotatedInjectionMetadata以map的形式缓存到AbstractAnnotationBeanPostProcessor类的injectionMetadataCache属性中，其中map的key是beanName，value是AnnotatedInjectionMetadata对象，如下图：



* + - * 1. AbstractAutowireCapableBeanFactory#populateBean()

接下来doCreateBean()继续执行，执行到populateBean()填充bean属性时，会执行属性注入操作，我们回顾一下spring容器启动流程的populateBean()源码

protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {

//bw是上一步createBeanInstance实例化的对象，对他进行判空   
 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;  
 }  
 }  
  
 // 给InstantiationAwareBeanPostProcessor最后一次机会，在属性注入前修改bean的属性值  
 boolean continueWithPropertyPopulation = true;  
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

// 执行InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation()方法  
 if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {  
 continueWithPropertyPopulation = false;  
 break;  
 }  
 }  
 }  
 }  
  
 if (!continueWithPropertyPopulation) {  
 return;  
 }

//获取要注入的属性列表  
 PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

//根据bean的依赖注入方式，遍历所有属性，依次对属性进行注入  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_NAME* ||  
 mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_TYPE*) {  
 MutablePropertyValues newPvs = new MutablePropertyValues(pvs);  
  
 // Add property values based on autowire by name if applicable.  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_NAME*) {

//（重点）根据bean的属性名进行依次注入  
 autowireByName(beanName, mbd, bw, newPvs);  
 }  
  
 // Add property values based on autowire by type if applicable.  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_TYPE*) {

//（重点）根据bean的类型进行依次注入  
 autowireByType(beanName, mbd, bw, newPvs);  
 }  
  
 pvs = newPvs;  
 }

//判断容器是否注册了InstantiationAwareBeanPostProcessors  
 boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

//是否进行了依赖检查  
 boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.*DEPENDENCY\_CHECK\_NONE*);  
  
 if (hasInstAwareBpps || needsDepCheck) {  
 if (pvs == null) {  
 pvs = mbd.getPropertyValues();  
 }  
 PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);  
 if (hasInstAwareBpps) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

**// 重点！！！！！！**

**//执行InstantiationAwareBeanPostProcessor的postProcessPropertyValues()方法**  
 pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  
 if (pvs == null) {  
 return;  
 }  
 }  
 }  
 }  
 if (needsDepCheck) {

//检查depends-on依赖的bean是否已经完成初始化  
 checkDependencies(beanName, mbd, filteredPds, pvs);  
 }  
 }  
  
 if (pvs != null) {

//（重点），填充bean属性,即将pvs中的所有属性填充到bw中  
 applyPropertyValues(beanName, mbd, bw, pvs);  
 }  
}

还记得ReferenceAnnotationBeanPostProcessor类有一个”祖先”接口叫SmartInstantiationAwareBeanPostProcessor吗？该接口在aop源码分析中出现过，它有一个InstantiationAwareBeanPostProcessor类型的父接口

在populateBean()中，将会查找所有的InstantiationAwareBeanPostProcessor接口的实现类并调用其postProcessPropertyValues()方法

* + - * 1. AbstractAnnotationBeanPostProcessor#postProcessPropertyValues()

那我们看看在AbstractAnnotationBeanPostProcessor类中是怎么实现该方法的

@Override  
public PropertyValues postProcessPropertyValues(  
 PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeanCreationException {  
 //（重点），因为之前在MergedBeanDefinitionPostProcessor#postProcessMergedBeanDefinition()已经调用过findInjectionMetadata()了，所以这个方法内部会判断缓存中有没有，有的话直接从缓存中用beanName取出来  
 InjectionMetadata metadata = findInjectionMetadata(beanName, bean.getClass(), pvs);  
 try {

//（重点），对metadata元数据中解析出来的，被@Reference标记的字段和方法，执行注入操作。也就是说对bean中被标记的方法和属性注入依赖  
 metadata.inject(bean, beanName, pvs);  
 } catch (BeanCreationException ex) {  
 throw ex;  
 } catch (Throwable ex) {  
 throw new BeanCreationException(beanName, "Injection of @" + getAnnotationType().getSimpleName()  
 + " dependencies is failed", ex);  
 }  
 return pvs;  
}

injectionMetadata#inject()

来看一下InjectionMetadata#inject()方法的源码

public void inject(Object target, @Nullable String beanName, @Nullable PropertyValues pvs) throws Throwable {  
 Collection<InjectedElement> checkedElements = this.checkedElements;  
 Collection<InjectedElement> elementsToIterate =  
 (checkedElements != null ? checkedElements : this.injectedElements);  
 if (!elementsToIterate.isEmpty()) {  
 for (InjectedElement element : elementsToIterate) {

//（重点），执行InjectedElement#inject()  
 element.inject(target, beanName, pvs);  
 }  
 }  
}

inject()

最后我们发现，它最后会执行AbstractAnnotationBeanPostProcessor#AnnotatedFieldElement#inject()，源码如下：

@Override  
protected void inject(Object bean, String beanName, PropertyValues pvs) throws Throwable {  
 Class<?> injectedType = field.getType();  
 //（重点），执行getInjectedObject()获取将要被注入的依赖对象  
 Object injectedObject = getInjectedObject(attributes, bean, beanName, injectedType, this);  
 ReflectionUtils.*makeAccessible*(field);  
 //（重点），执行依赖注入  
 field.set(bean, injectedObject);  
}

getInjectedObject()

AbstractAnnotationBeanPostProcessor#getInjectedObject()方法负责获取将要被@Reference注解注入的bean·

protected Object getInjectedObject(AnnotationAttributes attributes, Object bean, String beanName, Class<?> injectedType,  
 InjectionMetadata.InjectedElement injectedElement) throws Exception {  
 //得到需要被注入的对象的key  
 String cacheKey = buildInjectedObjectCacheKey(attributes, bean, beanName, injectedType, injectedElement);

//尝试从缓存中获取

Object injectedObject = injectedObjectsCache.get(cacheKey);  
  
 if (injectedObject == null) {

//（重点），缓存中没有的话就创建一个并放入缓存  
 injectedObject = doGetInjectedBean(attributes, bean, beanName, injectedType, injectedElement);  
 //放入缓存  
 injectedObjectsCache.putIfAbsent(cacheKey, injectedObject);  
 }  
 return injectedObject;  
}

injectedObjectsCache用于缓存即将被注入的bean，如果没有的话，就通过doGetInjectedBean()创建一个，并将其放入injectedObjectsCache缓存中

doGetInjectedBean()

可以看到，最终getInjectedObject()将会执行ReferenceAnnotationBeanPostProcessor#doGetInjectedBean()方法，在方法中我们看到，最终是生成了一个ReferenceBean，并将这个ReferenceBean注册到spring的单例池**singletonObjects**中

@Override  
protected Object doGetInjectedBean(AnnotationAttributes attributes, Object bean, String beanName, Class<?> injectedType,  
 InjectionMetadata.InjectedElement injectedElement) throws Exception {  
 */\*\*  
 \* The name of bean that annotated Dubbo's {****@link*** *Service @Service} in local Spring {****@link*** *ApplicationContext}  
 \*/* String referencedBeanName = buildReferencedBeanName(attributes, injectedType);  
 */\*\*  
 \* The name of bean that is declared by {****@link*** *Reference @Reference} annotation injection  
 \*/* String referenceBeanName = getReferenceBeanName(attributes, injectedType);  
 //构建referenceBean  
 ReferenceBean referenceBean = buildReferenceBeanIfAbsent(referenceBeanName, attributes, injectedType);  
 //将这个ReferenceBean注册到spring容器的单例池singletonObjects中

//这里有两种场景

//第一种是对应的服务存在于我们的本地，也就是说spring中已经有这个bean了，那么只需要给它注册一个别名就可以了

//第二种是对应的服务不在我们本地。如果不存在就表示对应的服务存在于远程，那么就先将封装好的ReferenceBean对象注册到容器中，等到下面构建代理类时，再触发referenceBean.get()进行服务引用  
 registerReferenceBean(referencedBeanName, referenceBean, attributes, injectedType);  
  
 cacheInjectedReferenceBean(referenceBean, injectedElement);  
 // 创建一个代理对象，最后被注入到service中的就是这个代理对象

// 内部会调用referenceBean.get();触发服务引用，懒汉式加载就是在这里触发服务引用的  
 return getOrCreateProxy(referencedBeanName, referenceBeanName, referenceBean, injectedType);  
}

getOrCreateProxy()

可以看到，该方法会调用referenceBean.get()，触发服务引用

private Object getOrCreateProxy(String referencedBeanName, String referenceBeanName, ReferenceBean referenceBean, Class<?> serviceInterfaceType) {

//如果本地已经存在该bean，那么只构建一个ReferenceBean的代理  
 if (existsServiceBean(referencedBeanName)) { // If the local @Service Bean exists, build a proxy of ReferenceBean  
 return *newProxyInstance*(getClassLoader(), new Class[]{serviceInterfaceType},  
 wrapInvocationHandler(referenceBeanName, referenceBean));  
 } else { // ReferenceBean should be initialized and get immediatelyreturn referenceBean.get();  
 }  
}

* 1. dubbo服务调用过程
     1. 服务端需要的信息

我们先思考一下，在服务调用的过程中，客户端需要告诉服务提供者哪些信息：

1. 接口名
2. 方法名
3. 参数类型
4. 参数值

知道了以上几个参数以后，服务提供者就可已定位所需要调用的方法，并进行调用，随后将调用结果组装并返回给客户端

* + 1. 协议
       1. 粘包和拆包

除了需要提供以上信息外，客户端需要和服务提供者约定一个共同的协议，否则就是鸡同鸭讲，只有约定好协议后，以上信息才能被正确地传输并解析出来

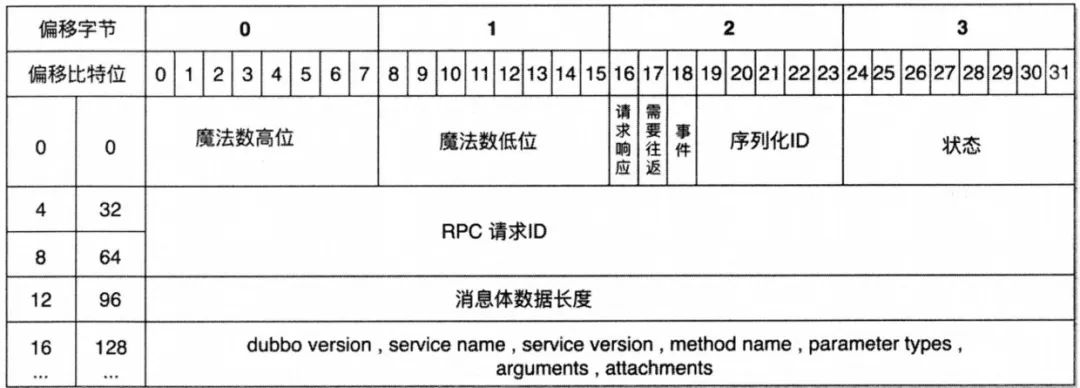
计算机网络的应用层根据拆包方式不同，主要分为3种类型的协议：

1. **固定长度：**即协议的包长度是固定的，比如100个字节为一个包单元，没有使用完的部分用空位填充。该方案很少使用
2. **优点：**效率高，读固定长度后立即解析，然后再继续读下一段
3. **缺点：**死板，长度固定，每次发消息不能超过该长度，并且没用完的部分还要填白，说白了就是定长了浪费，定短了不够用，在RPC场景种不适用
4. **特殊字符隔断：**就是定义一个特殊字符，用该字符作为包结束的标志，但是对于消息内容中带有特殊标志位的需要进行转义处理，如FTP协议
5. **优点：**长度自由，节约空间
6. **缺点：**如果传输的数据中出现了该特殊字符，需要对该字符做转义处理
7. **Header+body：**头部header固定长度，里面是一些属性，包括body的长度。Body长度不固定，具体长度在header中定义
8. **优点：**长度自由
9. **缺点：**header头如果过大会浪费空间

Dubbo除了支持dubbo协议外，还支持多种其他协议，这里我们只简单介绍一下默认使用的dubbo协议

Dubbo协议就采用了header+body的方式，同时也有特殊字符0xdabb，用于解决TCP网络粘包问题

* + - 1. Dubbo协议



上图是dubbo协议的协议头，我们可以看到，共16个字节，即128个bit位，如下图：



16字节协议头后面就是协议体，协议体中包括协议版本、接口名字、接口版本、方法名字等等

* + - 1. 序列化

网络数据传输是以字节流形式传输的，而我们的对象是多维的，字节流是一维的，因此在进行网络传输前，我们必须将对象进行序列化处理成字节流，传输完毕后再将字节流进行反序列化处理

Dubbo支持多种序列化，这些序列化主要分为两大类：

1. **字符型**
2. **二进制流**
   * + - 1. 字符型序列化

字符型的代表就是 XML、JSON

优点：调试方便，它是对人友好的，我们一看就能知道那个字段对应的哪个参数。

缺点：传输的效率低，有很多冗余的东西，比如 JSON 的括号，对于网络传输来说传输的时间变长，占用的带宽变大。

* + - * 1. 二进制流序列化

最典型的二进制流序列化就是dubbo默认使用的hessian2

优点：对机器友好的，它的数据更加的紧凑，所以占用的字节数更小，传输更快。

缺点：就是调试很难，肉眼是无法识别的，必须借用特殊的工具转换