课程内容：

1. Dubbo3.0为什么要改成应用级注册
2. Dubbo3.0如何进行服务应用级注册
3. Dubbo3.0如何进行服务接口级注册
4. Dubbo3.0如何进行服务接口级引入
5. Dubbo3.0如何进行服务应用级引入
6. MigrationInvoker的生成及作用

有道云链接：https://note.youdao.com/s/Sp2JnMvm

Dubbo3.0源码：https://gitee.com/archguide/dubbo-dubbo-3.0.7
服务导出与引入原理流程图：https://www.processon.com/view/link/62c441e80791293dccaebded

不管是服务导出还是服务引入，都发生在应用启动过程中，比如当我们在启动类上加上@EnableDubbo时，该注解上有一个@DubboComponentScan注解，@DubboComponentScan注解Import了一个DubboComponentScanRegistrar，DubboComponentScanRegistrar中会调用DubboSpringInitializer.initialize()，该方法中会注册一个DubboDeployApplicationListener，而DubboDeployApplicationListener会监听Spring容器启动完成事件ContextRefreshedEvent，一旦接收到这个事件后，就会开始Dubbo的启动流程，就会执行DefaultModuleDeployer的start()进行**服务导出**与**服务引入**。

额外先提一下，在启动过程中，在做完**服务导出**与**服务引入**后，还会做几件非常重要的事情：

1. 导出一个应用元数据服务（就是一个MetadataService服务，这个服务也会注册到注册中心，后面会分析它有什么用），或者将应用元数据注册到元数据中心
2. 生成当前应用的实例信息对象ServiceInstance，比如应用名、实例ip、实例port，并将实例信息注册到注册中心，也就是应用级注册

这两个步骤的作用是什么，后面会细讲。

服务导出

当我们在**某个接口的实现类**上加上**@DubboService**后，就表示定义了一个Dubbo服务，应用启动时Dubbo只要扫描到了@DubboService，就会解析对应的类，得到服务相关的配置信息，比如：

1. 服务的类型，也就是接口，接口名就是服务名
2. 服务的具体实现类，也就是当前类
3. 服务的version、timeout等信息，就是@DubboService中所定义的各种配置

解析完服务的配置信息后，就会把这些配置信息封装成为一个ServiceConfig对象，并调用其export()进行服务导出，此时一个ServiceConfig对象就表示一个Dubbo服务。

而所谓的服务导出，主要就是完成三件事情：

1. 确定服务的最终参数配置
2. 按不同协议启动对应的Server（**服务暴露**）
3. 将服务注册到注册中心（**服务注册**）

确定服务参数

一个Dubbo服务，除开服务的名字，也就是接口名，还会有很多其他的属性，比如超时时间、版本号、服务所属应用名、所支持的协议及绑定的端口等众多信息。

但是，通常这些信息并不会全部在**@DubboService**中进行定义，比如，一个Dubbo服务肯定是属于某个应用的，而一个应用下可以有多个Dubbo服务，所以我们可以在应用级别定义一些通用的配置，比如协议。

我们在application.yml中定义：

```
dubbo:
  application:
    name: dubbo-springboot-demo-provider
  protocol:
    name: tri
    port: 20880
```

表示当前应用下所有的Dubbo服务都支持通过tri协议进行访问，并且访问端口为20880，所以在进行**某个服务**的服务导出时，就需要将应用中的这些配置信息合并到当前服务的配置信息中。

另外，除开可以通过**@DubboService**来配置服务，我们也可以在配置中心对服务进行配置，比如在配置中心中配置：

```
dubbo.service.org.apache.dubbo.samples.api.DemoService.timeout=5000
```

表示当前服务的超时时间为5s。

所以，在服务导出时，也需要从配置中心获取当前服务的配置，如果在@DubboService中也定义了timeout，那么就用配置中心的覆盖掉，配置中心的配置优先级更高。

最终确定出服务的各种参数。

这块内容和Dubbo2.7一致，想详细了解的可以看第4期Dubbo课程的笔记或视频。

服务注册

当确定好了最终的服务配置后，Dubbo就会根据这些配置信息生成对应的**服务URL**，比如：

```
tri://192.168.65.221:20880/org.apache.dubbo.springboot.demo.DemoService?
application=dubbo-springboot-demo-provider&timeout=3000
```

这个URL就表示了一个Dubbo服务，服务消费者只要能获得到这个服务URL，就知道了关于这个Dubbo服务的全部信息，包括服务名、支持的协议、ip、port、各种配置。

确定了服务URL之后，服务注册要做的事情就是把这个服务URL存到注册中心（比如Zookeeper）中去，说的再简单一点，就是把这个字符串存到Zookeeper中去，这个步骤其实是非常简单的，实现这个功能的源码在RegistryProtocol中的export()方法中，最终服务URL存在了Zookeeper的**/dubbo/接口名/providers**目录下。

但是服务注册并不仅仅就这么简单，既然上面的这个URL表示一个服务，并且还包括了服务的一些配置信息，那这些配置信息如果改变了呢？比如利用Dubbo管理台中的动态配置功能（注意，并不是配置中心）来修改服务配置，动态配置可以应用运行过程中动态的修改服务的配置，并实时生效。

如果利用动态配置功能修改了服务的参数，那此时就要重新生成服务URL并重新注册到注册中心，这样服务消费者就能及时的获取到服务配置信息。

而对于服务提供者而言，在服务注册过程中，还需要能监听到动态配置的变化，一旦发生了变化，就根据最新的配置重新生成服务URL，并重新注册到中心。

应用级注册

在Dubbo3.0之前，Dubbo是接口级注册，服务注册就是把接口名以及服务配置信息注册到注册中心中，注册中心存储的数据格式大概为：

```
接口名1：tri://192.168.65.221:20880/接口名1?application=应用名
接口名2：tri://192.168.65.221:20880/接口名2?application=应用名
接口名3：tri://192.168.65.221:20880/接口名3?application=应用名
```

key是接口名，value就是服务URL，上面的内容就表示现在有一个应用，该应用下有3个接口，应用实例部署在192.168.65.221，此时，如果给该应用增加一个实例，实例ip为192.168.65.222，那么新的

实例也需要进行服务注册，会向注册中心新增3条数据：

```
接口名1：tri://192.168.65.221:20880/接口名1?application=应用名
接口名2：tri://192.168.65.221:20880/接口名2?application=应用名
接口名3：tri://192.168.65.221:20880/接口名3?application=应用名

接口名1：tri://192.168.65.222:20880/接口名1?application=应用名
接口名2：tri://192.168.65.222:20880/接口名2?application=应用名
接口名3：tri://192.168.65.222:20880/接口名3?application=应用名
```

可以发现，如果一个应用中有3个Dubbo服务，那么每增加一个实例，就会向注册中心增加3条记录，那如果一个应用中有10个Dubbo服务，那么每增加一个实例，就会向注册中心增加10条记录，注册中心的压力会随着应用实例的增加而剧烈增加。

反过来，如果一个应用有3个Dubbo服务，5个实例，那么注册中心就有15条记录，此时增加一个Dubbo服务，那么注册中心就会新增5条记录，注册中心的压力也会剧烈增加。

注册中心的数据越多，数据就变化的越频繁，比如修改服务的timeout，那么对于注册中心和应用都需要消耗资源用来处理数据变化。

所以为了降低注册中心的压力，Dubbo3.0支持了应用级注册，同时也兼容接口级注册，用户可以逐步迁移成应用级注册，而一旦采用应用级注册，最终注册中心的数据存储就变成为：

```
应用名：192.168.65.221:20880
应用名：192.168.65.222:20880
```

表示在注册中心中，只记录应用所对应的实例信息（IP+绑定的端口），这样只有一个应用的实例增加了，那么注册中心的数据才会增加，而不关心一个应用中到底有多少个Dubbo服务。

这样带来的好处就是，注册中心存储的数据变少了，注册中心中数据的变化频率变小了（那服务的配置如果发生了改变怎么办呢？后面会讲），并且使用应用级注册，使得 Dubbo3 能实现与异构微服务体系如Spring Cloud、Kubernetes Service等在地址发现层面更容易互通， 为连通 Dubbo与其他微服务体系提供可行方案。

应用级注册带来了好处，但是对于Dubbo来说又出现了一些新的问题，比如：原本，服务消费者可以直接从注册中心就知道某个Dubbo服务的所有服务提供者以及相关的协议、ip、port、配置等信息，那现在注册中心上只有ip、port，那对于服务消费者而言：**服务消费者怎么知道现在它要用的某个Dubbo服务，也就是某个接口对应的应用是哪个呢？**

对于这个问题，在进行服务导出的过程中，会在Zookeeper中存一个映射关系，在服务导出的最后一步，在ServiceConfig的exported()方法中，会保存这个映射关系：

```
1 接口名：应用名
```

这个映射关系存在Zookeeper的**/dubbo/mapping目录**下，存了这个信息后，消费者就能根据接口名找到所对应的应用名了。

消费者知道了要使用的Dubbo服务在哪个应用，那也就能从注册中心中根据应用名查到应用的所有实例信息（ip+port），也就是可以发送方法调用请求了，但是在真正发送请求之前，还得知道服务的配置信息，对于消费者而言，它得知道当前要调用的这个Dubbo服务支持什么协议、timeout是多少，**那服务的配置信息从哪里获取呢?**

之前的服务配置信息是直接从注册中心就可以获取到的，就是服务URL后面，但是现在不行了，现在需要从服务提供者的元数据服务获取，前面提到过，在应用启动过程中会进行服务导出和服务引入，然后就会暴露一个**应用元数据服务**，其实这个应用元数据服务就是一个Dubbo服务（Dubbo框架内置的，自己实现的），消费者可以调用这个服务来获取某个应用中所提供的所有Dubbo服务以及服务配置信息，这样也就能知道服务的配置信息了。

后面分析服务引入时，会进一步分析具体细节。

现在我们知道了应用注册的好处，以及相关问题的解决方式，那么我们来看它到底是如何实现的。

首先，我们可以通过配置**dubbo.application.register-mode**来控制：

1. instance：表示只进行应用级注册
2. interface：表示只进行接口级注册
3. all：表示应用级注册和接口级注册都进行，默认

不管是什么注册，都需要存数据到注册中心，而Dubbo3的源码实现中会根据所配置的注册中心生成两个URL（不是服务URL，可以理解为注册中心URL，用来访问注册中心的）：

1. service-discovery-registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-springboot-demo-provider&dubbo=2.0.2&pid=13072&qos.enable=false&registry=zookeeper&timestamp=1651755501660
2. registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-springboot-demo-provider&dubbo=2.0.2&pid=13072&qos.enable=false&registry=zookeeper&timestamp=1651755501660

这两个URL只有schema不一样，一个是service-discovery-registry，一个是registry，而registry是Dubbo3之前就存在的，也就代表接口级服务注册，而service-discovery-registry就表示应用级服务

注册。

在服务注册相关的源码中，当调用RegistryProtocol的export()方法处理registry://时，会利用ZookeeperRegistry把服务URL注册到Zookeeper中去，这个我们能理解，这就是接口级注册。

而类似，当调用RegistryProtocol的export()方法处理service-discovery-registry://时，会利用ServiceDiscoveryRegistry来进行相关逻辑的处理，那是不是就是在这里把应用信息注册到注册中心去呢？并没有这么简单。

1. 首先，不可能每导出一个服务就进行一次应用注册，太浪费了，应用注册只要做一次就行了
2. 另外，如果一个应用支持了多个端口，那么应用注册时只要挑选其中一个端口作为实例端口就可以了（该端口只要能接收到数据就行）
3. 前面提到，应用启动过程中要暴露应用元数据服务，所以在此处也还是要收集当前所暴露的服务配置信息，以提供给应用元数据服务

所以ServiceDiscoveryRegistry在注册一个服务URL时，并不会往注册中心存数据，而只是把服务URL存到到一个MetadataInfo对象中，MetadataInfo对象中就保存了当前应用中所有的Dubbo服务信息（服务名、支持的协议、绑定的端口、timeout等）

前面提到过，在应用启动的最后，才会进行应用级注册，而应用级注册就是当前的应用实例上相关的信息存入注册中心，包括：

1. 应用的名字
2. 获取应用元数据的方式
3. 当前实例的ip和port
4. 当前实例支持哪些协议以及对应的port

比如：


```
{
    "name":"dubbo-springboot-demo-provider",
    "id":"192.168.65.221:20882",
    "address":"192.168.65.221",
    "port":20882,
    "sslPort":null,
    "payload":{
        "@class":"org.apache.dubbo.registry.zookeeper.ZookeeperInstance",
        "id":"192.168.65.221:20882",
        "name":"dubbo-springboot-demo-provider",
```

```
        "metadata":{
            "dubbo.endpoints":"[{\"port\":20882,\"protocol\":\"dubbo\"},{\"port\":50051,\"protocol\":\"tri\"}]",
            "dubbo.metadata-service.url-params":"{\"connections\":\"1\",\"version\":\"1.0.0\",\"dubbo\":\"2.0.2\",\"side\":\"provider\",\"port\":\"20882\",\"protocol\":\"dubbo\"}",
            "dubbo.metadata.revision":"65d5c7b814616ab10d32860b54781686",
            "dubbo.metadata.storage-type":"local"
        }
    },
    "registrationTimeUTC":1654585977352,
    "serviceType":"DYNAMIC",
    "uriSpec":null
}
```

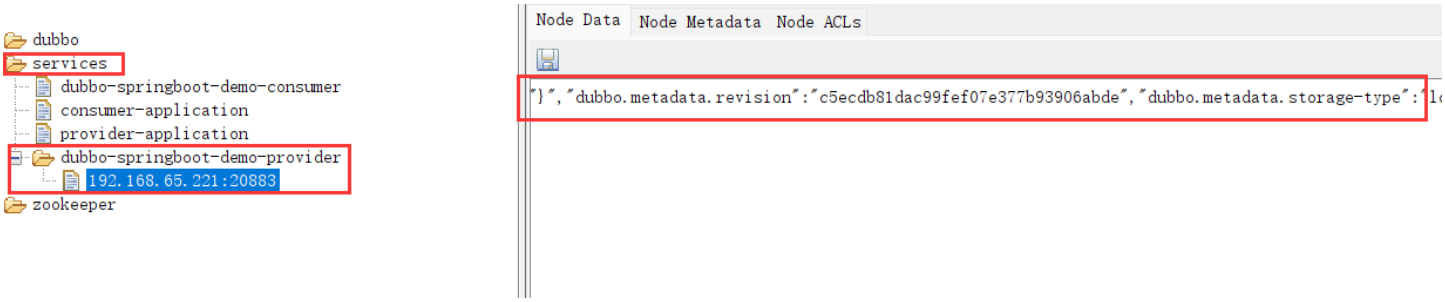
一个实例上可能支持多个协议以及多个端口，**那如何确定实例的ip和端口呢?**
答案是：获取MetadataInfo对象中保存的所有服务URL，优先取dubbo协议对应ip和port，没有dubbo协议则所有服务URL中的第一个URL的ip和port。

另外一个协议一般只会对应一个端口，但是如何就是对应了多个，比如：

```yaml
dubbo:
  application:
    name: dubbo-springboot-demo-provider
  protocols:
    p1:
      name: dubbo
      port: 20881
    p2:
      name: dubbo
      port: 20882
    p3:
      name: tri
      port: 50051
```

如果是这样，最终存入endpoint中的会保证一个协议只对应一个端口，另外那个将被忽略，最终服务消费者在进行服务引入时将会用到这个endpoint信息。

确定好实例信息后之后，就进行最终的应用注册了，就把实例信息存入注册中心的**/services/应用名目录**下：



可以看出services节点下存的是应用名，应用名的节点下存的是实例ip和实例port，而ip和port这个节点中的内容就是实例的一些基本信息。

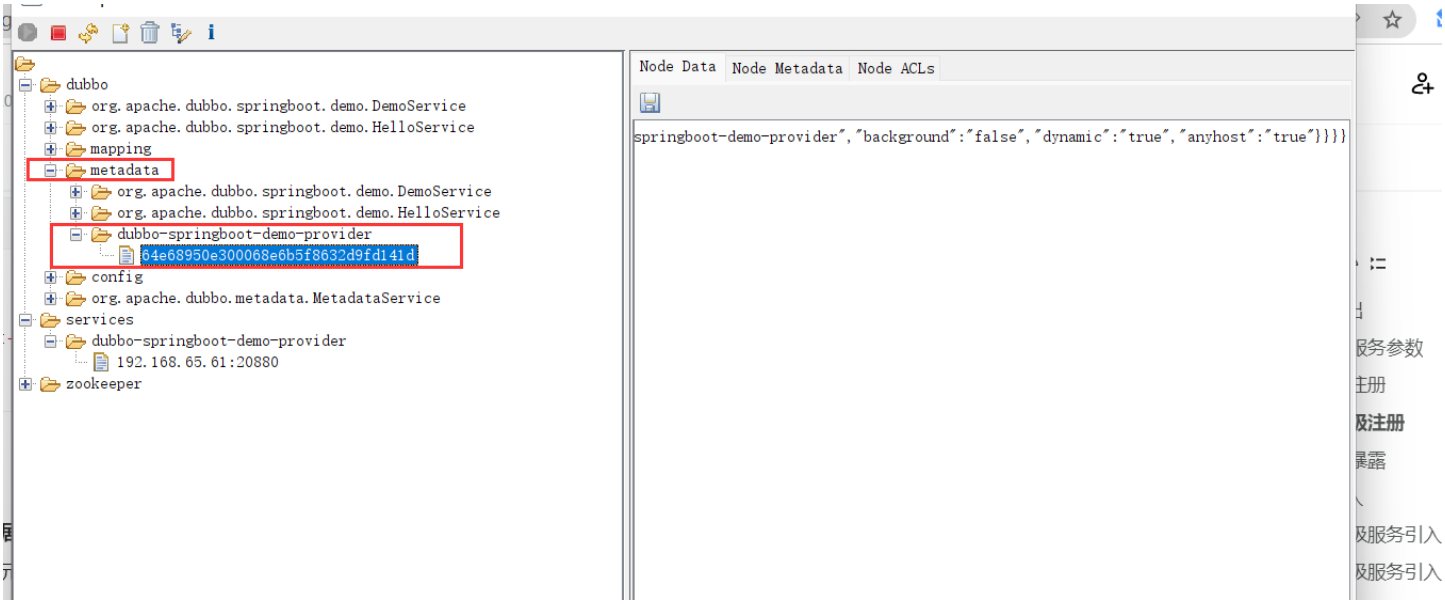
额外，我们可以配置dubbo.metadata.storage-type，默认时local，可以通过配置改为remote：

```
dubbo:
  application:
    name: dubbo-springboot-demo-provider
    metadata-type: remote
```

这个配置其实跟应用元数据服务有关系：

1. 如果为local，那就会启用**应用元数据服务**，最终服务消费者就会调用元数据服务获取到应用元数据信息
2. 如果为remote，那就不会暴露应用元数据服务，那么服务消费者从**元数据中心**获取应用元数据呢？

在Dubbo2.7中就有了元数据中心，它其实就是用来减轻注册中心的压力的，Dubbo会把服务信息完整的存一份到元数据中心，元数据中心也可以用Zookeeper来实现，在暴露完元数据服务之后，在注册实例信息到注册中心之前，就会把MetadataInfo存入元数据中心，比如：



节点内容为：

```
{
        "app": "dubbo-springboot-demo-provider",
        "revision": "64e68950e300068e6b5f8632d9fd141d",
        "services": {
                "org.apache.dubbo.springboot.demo.HelloService:tri": {
                        "name": "org.apache.dubbo.springboot.demo.HelloService",
                        "protocol": "tri",
                        "path": "org.apache.dubbo.springboot.demo.HelloService",
                        "params": {
                                "side": "provider",
                                "release": "",
                                "methods": "sayHello",
                                "deprecated": "false",
                                "dubbo": "2.0.2",
                                "interface": "org.apache.dubbo.springboot.demo.HelloService",
                                "service-name-mapping": "true",
                                "generic": "false",
                                "metadata-type": "remote",
                                "application": "dubbo-springboot-demo-provider",
                                "background": "false",
                                "dynamic": "true",
                                "anyhost": "true"
                        }
                },
```

```
                "org.apache.dubbo.springboot.demo.DemoService:tri": {
                        "name": "org.apache.dubbo.springboot.demo.DemoService",
                        "protocol": "tri",
                        "path": "org.apache.dubbo.springboot.demo.DemoService",
                        "params": {
                                "side": "provider",
                                "release": "",
                                "methods": "sayHelloStream,sayHello,sayHelloServerStream",
                                "deprecated": "false",
                                "dubbo": "2.0.2",
                                "interface": "org.apache.dubbo.springboot.demo.DemoService",
                                "service-name-mapping": "true",
                                "generic": "false",
                                "metadata-type": "remote",
                                "application": "dubbo-springboot-demo-provider",
                                "background": "false",
                                "dynamic": "true",
                                "anyhost": "true"
                        }
                }
        }
}
```


这里面就记录了当前实例上提供了哪些服务以及对应的协议，注意并没有保存对应的端口……，所以后面服务消费者得利用实例信息中的endpoint，因为endpoint中记录了协议对应的端口....

其实元数据中心和元数据服务提供的功能是一样的，都可以用来获取某个实例的MetadataInfo，上面中的UUID表示实例编号，只不过元数据中心是**集中**式的，元数据服务式**分散**在各个提供者实例中的，如果整个微服务集群压力不大，那么效果差不多，如果微服务集群压力大，那么元数据中心的压力就大，此时单个元数据服务就更适合，所以默认也是采用的元数据服务。

至此，应用级服务注册的原理就分析完了，总结一下：

1. 在导出某个Dubbo服务URL时，会把服务URL存入MetadataInfo中
2. 导出完某个Dubbo服务后，就会把**服务接口名:应用名**存入元数据中心（可以用Zookeeper实现）
3. 导出所有服务后，完成服务引入后
4. 判断要不要启动元数据服务，如果要就进行导出，固定使用Dubbo协议
5. 将MetadataInfo存入元数据中心
6. 确定当前实例信息（应用名、ip、port、endpoint）

7. 将实例信息存入注册中心，完成应用注册

服务暴露

服务暴露就是根据不同的协议启动不同的Server，比如dubbo和tri协议启动的都是Netty，像Dubbo2.7中的http协议启动的就是Tomcat，这块在服务调用的时候再来分析（dubbo协议在上期讲了，下节课主要讲triple协议）

服务引入

```
@DubboReference
private DemoService demoService;
```

我们需要利用@DubboReference注解来引入某一个Dubbo服务，应用在启动过程中，进行完服务导出之后，就会进行服务引入，属性的类型就是一个Dubbo服务接口，而服务引入最终要做到的就是给这个属性赋值一个接口代理对象。

在Dubbo2.7中，只有接口级服务注册，服务消费者会利用接口名从注册中心找到该服务接口所有的服务URL，服务消费者会根据每个服务URL的protocol、ip、port生成对应的Invoker对象，比如生成TripleInvoker、DubboInvoker等，调用这些Invoker的invoke()方法就会发送数据到对应的ip、port，生成好所有的Invoker对象之后，就会把这些Invoker对象进行封装并生成一个服务接口的代理对象，代理对象调用某个方法时，会把所调用的方法信息生成一个Invocation对象，并最终通过某一个Invoker的invoke()方法把Invocation对象发送出去，所以代理对象中的Invoker对象是关键，服务引入最核心的就是要生成这些Invoker对象。

Invoker是非常核心的一个概念，也有非常多种类，比如：

1. TripleInvoker：表示利用tri协议把Invocation对象发送出去
2. DubboInvoker：表示利用dubbo协议把Invocation对象发送出去
3. ClusterInvoker：有负载均衡功能
4. MigrationInvoker：迁移功能，后面分析，Dubbo3.0新增的

像TripleInvoker和DubboInvoker对应的就是具体服务提供者，包含了服务提供者的ip地址和端口，并且会负责跟对应的ip和port建立Socket连接，后续就可以基于这个Socket连接并按协议格式发送Invocation对象。

比如现在引入了DemoService这个服务，那如果该服务支持：

1. 一个tri协议，绑定的端口为20881

2. 一个tri协议，绑定的端口为20882
3. 一个dubbo协议，绑定的端口为20883

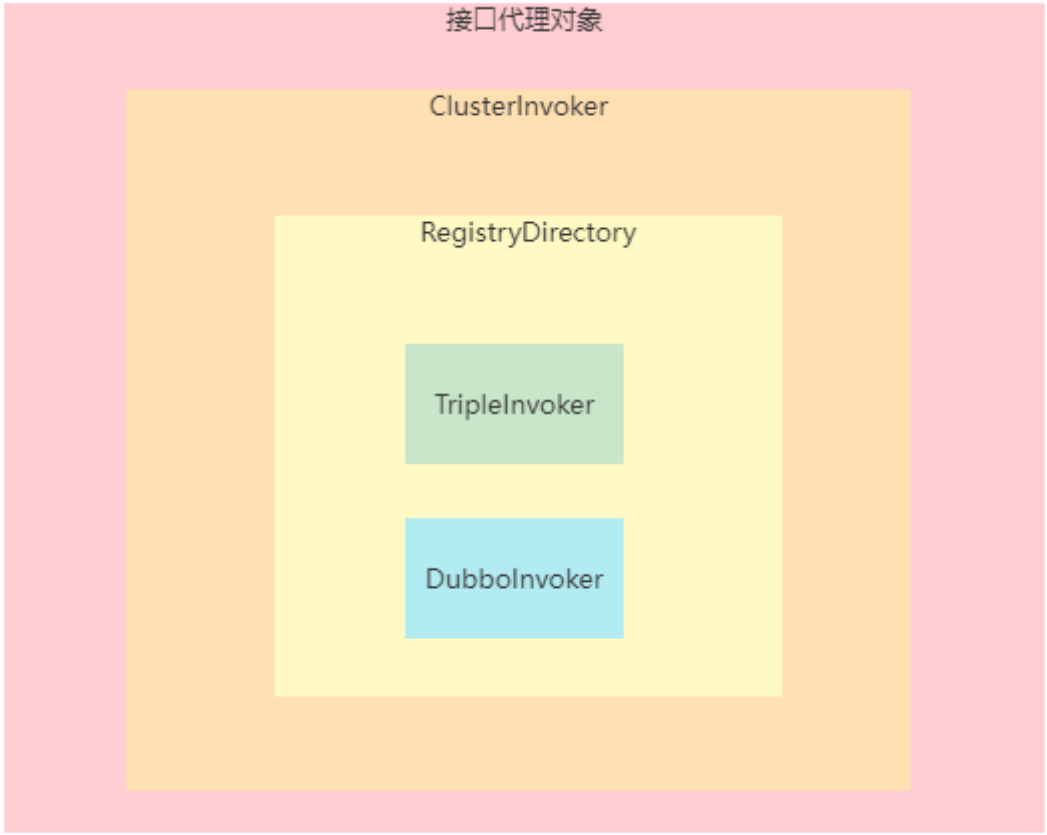
那么在服务消费端这边，就会生成两个TripleInvoker和一个DubboInvoker，代理对象执行方法时就会进行负载均衡选择其中一个Invoker进行调用。

接口级服务引入

在讲服务导出时，Dubbo3.0默认情况下即会进行接口级注册，也会进行应用级注册，目的就是为了兼容服务消费者应用用的还是Dubbo2.7，用Dubbo2.7就只能老老实实的进行**接口级服务引入**。

接口级服务引入核心就是要找到当前所引入的服务有哪些**服务URL**，然后根据每个服务URL生成对应的Invoker，流程为：

1. 首先，根据当前引入的服务接口生成一个**RegistryDirectory对象**，表示**动态服务目录**，用来查询并缓存服务提供者信息。
2. RegistryDirectory对象会根据**服务接口名**去注册中心，比如Zookeeper中的**/dubbo/服务接口名/providers/**节点下查找所有的服务URL
3. 根据每个服务URL生成对应的Invoker对象，并把Invoker对象存在RegistryDirectory对象的**invokers**属性中
4. RegistryDirectory对象也会监听**/dubbo/服务接口名/providers/**节点的数据变化，一旦发生了变化就要进行相应的改变
5. 最后将RegistryDirectory对象生成一个ClusterInvoker对象，到时候调用ClusterInvoker对象的invoke()方法就会进行负载均衡选出某一个Invoker进行调用



应用级服务引入

在Dubbo中，应用级服务引入，并不是指引入某个应用，这里和SpringCloud是有区别的，在SpringCloud中，服务消费者只要从注册中心找到**要调用的应用的所有实例地址**就可以了，但是在Dubbo中找到应用的实例地址还远远不够，因为在Dubbo中，我们是直接使用的接口，所以在Dubbo中就算是应用级服务引入，最终还是得找到服务接口有哪些服务提供者。

所以，对于服务消费者而言，不管是使用接口级服务引入，还是应用级服务引入，最终的结果应该得是一样的，也就是某个服务接口的提供者Invoker是一样的，不可能使用应用级服务引入得到的Invoker多一个或少一个，但是！！！，目前会有情况不一致，就是一个协议有多个端口时，比如在服务提供者应用这边支持：

```
dubbo:
  application:
    name: dubbo-springboot-demo-provider
  protocols:
    p1:
      name: dubbo
      port: 20881
    p2:
```

```
      name: tri
      port: 20882
    p3:
      name: tri
      port: 50051
```

那么在消费端进行服务引入时，比如引入DemoService时，接口级服务引入会生成3个Invoker（2个TripleInvoker，1个DubboInvoker），而应用级服务引入只会生成2个Invoker（1个TripleInvoker，1个DubboInvoker），原因就是在进行应用级注册时是按照一个协议对应一个port存的。

那既然接口级服务引入和应用级服务引入最终的结果差不多，有同学可能就不理解了，那应用级服务引入有什么好处呢？要知道应用级服务引入和应用级服务注册是对应，服务提供者应用如果只做**应用级注册**，那么对应的服务消费者就只能进行应用级服务引入，好处就是前面所说的，减轻了注册中心的压力等，那么带来的影响就是服务消费者端**寻找服务URL的逻辑更复杂了**。

只要找到了当前引入服务对应的服务URL，然后生成对应的Invoker，并最终生成一个ClusterInvoker。

在进行应用级服务引入时：

1. 首先，根据当前引入的服务接口生成一个**ServiceDiscoveryRegistryDirectory对象**，表示**动态服务目录**，用来查询并缓存服务提供者信息。
2. 根据接口名去获取**/dubbo/mapping/服务接口名**节点的内容，拿到的就是该接口所对应的**应用名**
3. 有了应用名之后，再去获取**/services/应用名**节点下的实例信息
4. 依次遍历每个实例，每个实例都有一个编号revision
    a. 根据metadata-type进行判断
        i. 如果是local：则调用实例上的**元数据服务**获取应用元数据（MetadataInfo）
        ii. 如果是remote：则根据应用名从**元数据中心**获取应用元数据（MetadataInfo）
    a. 获取到应用元数据之后就进行缓存，key为revision，MetadataInfo对象为value
    b. 这里为什么要去每个实例上获取应用的元数据信息呢？因为有可能不一样，虽然是同一个应用，但是在运行不同的实例的时候，可以指定不同的参数，比如不同的协议，不同的端口，虽然在生产上基本不会这么做，但是Dubbo还是支持了这种情况
1. 根据从所有实例上获取到的MetadataInfo以及endpoint信息，就能知道所有实例上所有的服务URL（**注意：一个接口+一个协议+一个实例：对应一个服务URL**）
2. 拿到了这些服务URL之后，就根据当前引入服务的信息进行过滤，会根据引入服务的接口名+协议名，消费者可以在@DubboReference中指定协议，表示只使用这个协议调用当前服务，如果没有指定协议，那么就会去获取tri、dubbo、rest这三个协议对应的服务URL（Dubbo3.0默认只支持这三个协议）
3. 这样，经过过滤之后，就得到了当前所引入的服务对应的服务URL了

4. 根据每个服务URL生成对应的Invoker对象，并把Invoker对象存在ServiceDiscoveryRegistryDirectory对象的**invokers**属性中
5. 最后将ServiceDiscoveryRegistryDirectory对象生成一个ClusterInvoker对象，到时候调用ClusterInvoker对象的invoke()方法就会进行负载均衡选出某一个Invoker进行调用

MigrationInvoker的生成

上面分析了接口级服务引入和应用级服务引入，最终都是得到某个服务对应的服务提供者Invoker，那最终进行服务调用时，到底该怎么选择呢？

所以在Dubbo3.0中，可以配置：

```
# dubbo.application.service-discovery.migration 仅支持通过 -D 以及 全局配置中心 两种方式进行配置。
dubbo.application.service-discovery.migration=APPLICATION_FIRST

# 可选值
# FORCE_INTERFACE，强制使用接口级服务引入
# FORCE_APPLICATION，强制使用应用级服务引入
# APPLICATION_FIRST，智能选择是接口级还是应用级，默认就是这个
```

对于前两种强制的方式，没什么特殊，就是上面走上面分析的两个过程，没有额外的逻辑，那对于APPLICATION_FIRST就需要有额外的逻辑了，也就是Dubbo要判断，当前所引入的这个服务，应该走接口级还是应用级，这该如何判断呢？

事实上，在进行某个服务的服务引入时，会统一利用InterfaceCompatibleRegistryProtocol的refer来生成一个MigrationInvoker对象，在MigrationInvoker中有三个属性：

```
private volatile ClusterInvoker<T> invoker;  // 用来记录接口级ClusterInvoker
private volatile ClusterInvoker<T> serviceDiscoveryInvoker; // 用来记录应用级的ClusterInvoker
private volatile ClusterInvoker<T> currentAvailableInvoker; // 用来记录当前使用的ClusterInvoker，要么是接口级，要么应用级
```

一开始构造出来的MigrationInvoker对象中三个属性都为空，接下来会利用MigrationRuleListener来处理MigrationInvoker对象，也就是给这三个属性赋值。

在MigrationRuleListener的构造方法中，会从配置中心读取DUBBO_SERVICEDISCOVERY_MIGRATION组下面的"当前应用名+.migration"的配置项，配置项为yml格式，对应的对象为MigrationRule，也就是可以配置具体的迁移规则，比如某个接口或某个应用的MigrationStep（FORCE_INTERFACE、APPLICATION_FIRST、FORCE_APPLICATION），还可以配置threshold，表示一个阈值，比如配置为2，表示应用级Invoker数量是接口级Invoker数量的两倍时才使用应用级Invoker，不然就使用接口级数量，可以参考：https://dubbo.apache.org/zh/docs/advanced/migration-invoker/

如果没有配置迁移规则，则会看当前应用中是否配置了migration.step，如果没有，那就从全局配置中心读取dubbo.application.service-discovery.migration来获取MigrationStep，如果也没有配置，那MigrationStep默认为APPLICATION_FIRST

如果没有配置迁移规则，则会看当前应用中是否配置了migration.threshold，如果没有配，则threshold默认为-1。

在应用中可以这么配置：

```
dubbo:
  application:
    name: dubbo-springboot-demo-consumer
    parameters:
      migration.step: FORCE_APPLICATION
      migration.threshold: 2
```

确定了step和threshold之后，就要真正开始给MigrationInvoker对象中的三个属性赋值了，先根据step调用不同的方法

```
switch (step) {
    case APPLICATION_FIRST:
        // 先进行接口级服务引入得到对应的ClusterInvoker，并赋值给invoker属性
        // 再进行应用级服务引入得到对应的ClusterInvoker，并赋值给serviceDiscoveryInvoker属性
        // 再根据两者的数量判断到底用哪个，并且把确定的ClusterInvoker赋值给currentAvailableInvoker属性
        migrationInvoker.migrateToApplicationFirstInvoker(newRule);
        break;
    case FORCE_APPLICATION:
        // 只进行应用级服务引入得到对应的ClusterInvoker，并赋值给serviceDiscoveryInvoker和currentAvailableInvoker属性
        success = migrationInvoker.migrateToForceApplicationInvoker(newRule);
```

```
            break;
        case FORCE_INTERFACE:
        default:
            // 只进行接口级服务引入得到对应的ClusterInvoker，并赋值给invoker和currentAvailableInvoker属性
            success = migrationInvoker.migrateToForceInterfaceInvoker(newRule);
    }
```

具体接口级服务引入和应用级服务引入是如何生成ClusterInvoker，前面已经分析过了，我们这里只需要分析当step为APPLICATION_FIRST时，是如何确定最终要使用的ClusterInvoker的。

得到了接口级ClusterInvoker和应用级ClusterInvoker之后，就会利用DefaultMigrationAddressComparator来进行判断：

1. 如果应用级ClusterInvoker中没有具体的Invoker，那就表示只能用接口级Invoker
2. 如果接口级ClusterInvoker中没有具体的Invoker，那就表示只能用应用级Invoker
3. 如果应用级ClusterInvoker和接口级ClusterInvoker中都有具体的Invoker，则获取对应的Invoker个数
4. 如果在迁移规则和应用参数中都没有配置threshold，那就读取全局配置中心的**dubbo.application.migration.threshold**参数，如果也没有配置，则threshold默认为0（不是-1了）
5. 用**应用级Invoker数量 / 接口级Invoker数量**，得到的结果如果**大于等于threshold**，那就用应用级ClusterInvoker，否则用接口级ClusterInvoker

threshold默认为0，那就表示在既有应用级Invoker又有接口级Invoker的情况下，就一定会用应用级Invoker，两个正数相除，结果肯定为正数，当然你自己可以控制threshold，如果既有既有应用级Invoker又有接口级Invoker的情况下，你想在应用级Invoker的个数大于接口级Invoker的个数时采用应用级Invoker，那就可以把threshold设置为1，表示个数相等，或者个数相除之后的结果大于1时用应用级Invoker，否者用接口级Invoker

这样MigrationInvoker对象中的三个数据就能确定好值了，和在最终的接口代理对象执行某个方法时，就会调用MigrationInvoker对象的invoke，在这个invoke方法中会直接执行currentAvailableInvoker对应的invoker的invoker方法，从而进入到了接口级ClusterInvoker或应用级ClusterInvoker中，从而进行负载均衡，选择出具体的DubboInvoer或TripleInvoker，完成真正的服务调用。

具体服务调用是怎么做的，下节课继续。