# Apache Dubbo3
# 源码深入解析

源码视角全面剖析 Apache Dubbo3 工作原理

掌握大厂面试必备技能包


**宋小生**

Apache Dubbo Committer

现就职于平安壹钱包

负责中间件相关研发工作 热衷开源软件源码解析

乘风者计划专家博主



阿里云开发者社区 ALIBABA CLOUD DEVELOPER COMMUNITY

乘风者计划 THE WIND RIDERS PROGRAM

DUBBO

THE APACHE SOFTWARE FOUNDATION


扫码关注 Apache Dubbo

阿里云开发者“藏经阁”
海量电子手册免费下载

# 目录

# 关于本书

本书是由 Apache Dubbo Committer 宋小生所著，作者对于 Apache Dubbo3 源码的探究，一开始是为了将公司内 Dubbo2 顺利升级至 Dubbo3 版本，在源码探究的过程中发现 Apache Dubbo3 有许多优秀的设计思想值得学习，于是就打算通过 Debug 源码的方式将整个过程编写成博客的形式分享给其他对 Dubbo3 感兴趣的用户又或者是升级 Dubbo3 的用户，在整个过程中发现 Apache Dubbo3 在快速的成长，还有许多要完善的地方，于是就积极的参与了 Dubbo3 的一些建设，现将 Dubbo3 源码系列整理成一个电子书的形式让更多对源码感兴趣的小伙伴深入了解底层源码知识，由于一开始没想到能著作成书，因此个别章节之间会显得缺乏连贯性，尽管我们在编写成书的过程中做了一些增补，仍不可避免的会有类似的一些问题，希望得到大家的谅解和指正，相信阅读这本电子书的同学也是可以自己动手 Debug 找到自己需要的源码知识的。

本书源码主要基于 Dubbo3 v3.0.8 版本，自写作期间 Dubbo3 一直处于快速演进中，目前已经演进到了 3.2.x 版本，书中个别内容可能与最新代码实现有一定差异。

如果您对本书中的内容有建议或者意见、欢迎提出专业方面的修改建议。您可以直接在 GitHub 上以 issue 或者 PR 的形式提出。

https://github.com/songxiaosheng/post

**联系作者**

宋小生，Apache Dubbo Committer，负责中间件相关研发工作，热衷于开源软件源码解析。

欢迎关注作者的公众号《中间件源码》，如果您有任何意见、建议，或者想与作者交流，都可以直接后台留言。当然升级 Dubbo3 过程中遇到的一些困难也可以一起交流。

# Apache Dubbo 概念与架构

## 一、 Dubbo 简介

Apache Dubbo 是一款易用、高性能的 WEB 和 RPC 框架，同时为构建企业级微服务提供服务发现、流量治理、可观测、认证鉴权等能力、工具与最佳实践。

使用 Dubbo 开发的微服务原生具备相互之间的远程地址发现与通信能力，利用 Dubbo 提供的丰富服务治理特性，可以实现诸如服务发现、负载均衡、流量调度等服务治理诉求。Dubbo 被设计为高度可扩展，用户可以方便的实现流量拦截、选址的各种定制逻辑。

在云原生时代，Dubbo 相继衍生出了 Dubbo3、Proxyless Mesh 等架构与解决方案，在易用性、超大规模微服务实践、云原生基础设施适配、安全性等几大方向上进行了全面升级。

### 1. Dubbo 的开源故事

Apache Dubbo 最初是为了解决阿里巴巴内部的微服务架构问题而设计并开发的，在十多年的时间里，它在阿里巴巴公司内部的很多业务系统的到了非常广泛的应用。最早在 2008 年，阿里巴巴就将 Dubbo 捐献到开源社区，它很快成为了国内开源服务框架选型的事实标准框架，得到了业界更广泛的应用。在 2017 年，Dubbo 被正式捐献 Apache 软件基金会并成为 Apache 顶级项目，开始了一段新的征程。

Dubbo 被证实能很好的满足企业的大规模微服务实践，并且能有效降低微服务建设的开发与管理成本，不论是阿里巴巴还是工商银行、中国平安、携程、海尔等社区用户，它们都通过多年的大规模生产环境流量对 Dubbo 的稳定性与性能进行了充分验证。

后来 Dubbo 在很多大企业内部衍生出了独立版本，比如在阿里巴巴内部就基于 Dubbo3 衍生出了 HSF3，HSF 见证了阿里巴巴以电商业务为守的微服务系统的快速发展。自云原生概念推广以来，各大厂商都开始拥抱开源标准实现，阿里巴巴将其

内部 HSF 系统与开源社区 Dubbo 相融合，与社区一同推出了云原生时代的 Dubbo3 架构，截止 2022 年双十一结束，Dubbo3 已经在阿里巴巴内部全面取代 HSF 系统，包括电商核心、阿里云等一些核心系统已经全面运行在 Dubbo3 之上。

## 2. 为什么需要 Dubbo，它能做什么？

按照微服务架构的定义，采用它的组织能够很好的提高业务迭代效率与系统稳定性，但前提是要先能保证微服务按照期望的方式运行，要做到这一点需要解决服务拆分与定义、数据通信、地址发现、流量管理、数据一致性、系统容错能力等一系列问题。

Dubbo 可以帮助解决如下微服务实践问题：

- **微服务编程范式和工具**

Dubbo 支持基于 IDL 或语言特定方式的服务定义，提供多种形式的服务调用形式（如同步、异步、流式等）

- **高性能的 RPC 通信**

Dubbo 帮助解决微服务组件之间的通信问题，提供了基于 HTTP、HTTP/2、TCP 等的多种高性能通信协议实现，并支持序列化协议扩展，在实现上解决网络连接管理、数据传输等基础问题。

- **微服务监控与治理**

Dubbo 官方提供的服务发现、动态配置、负载均衡、流量路由等基础组件可以很好的帮助解决微服务基础实践的问题。除此之外，您还可以用 Admin 控制台监控微服务状态，通过周边生态完成限流降级、数据一致性、链路追踪等能力。

- **部署在多种环境**

Dubbo 服务可以直接部署在容器、Kubernetes、Service Mesh 等多种架构下。

- **活跃的社区**

Dubbo 项目托管在 Apache 社区，有来自国际、国内的活跃贡献者维护着超 10 个生态项目，贡献者包括来自海外、阿里巴巴、工商银行、携程、蚂蚁、腾讯等知名企业技术专家，确保 Dubbo 及时解决项目缺陷、需求及安全漏洞，跟进业界最新技术发展趋势。

- **庞大的用户群体**

Dubbo3 已在阿里巴巴成功取代 HSF 框架实现全面落地，成为阿里集团面向云原生时代的统一服务框架，庞大的用户群体是 Dubbo 保持稳定性、需求来源、先进性的基础。

## 3. Dubbo 不是什么？

- **不是应用开发框架的替代者**

Dubbo 设计为让开发者以主流的应用开发框架的开发模式工作，它不是各个语言应用开发框架的替代者，如它不是 Spring/Spring Boot 的竞争者，当你使用 Spring 时，Dubbo 可以无缝的与 Spring & Spring Boot 集成在一起。

- **不仅仅只是一款 RPC 框架**

Dubbo 提供了内置 RPC 通信协议实现，但它不仅仅是一款 RPC 框架。首先，它不绑定某一个具体的 RPC 协议，开发者可以在基于 Dubbo 开发的微服务体系中使用多种通信协议；其次，出了 RPC 通信之外，Dubbo 提供了丰富的服务治理能力与生态。

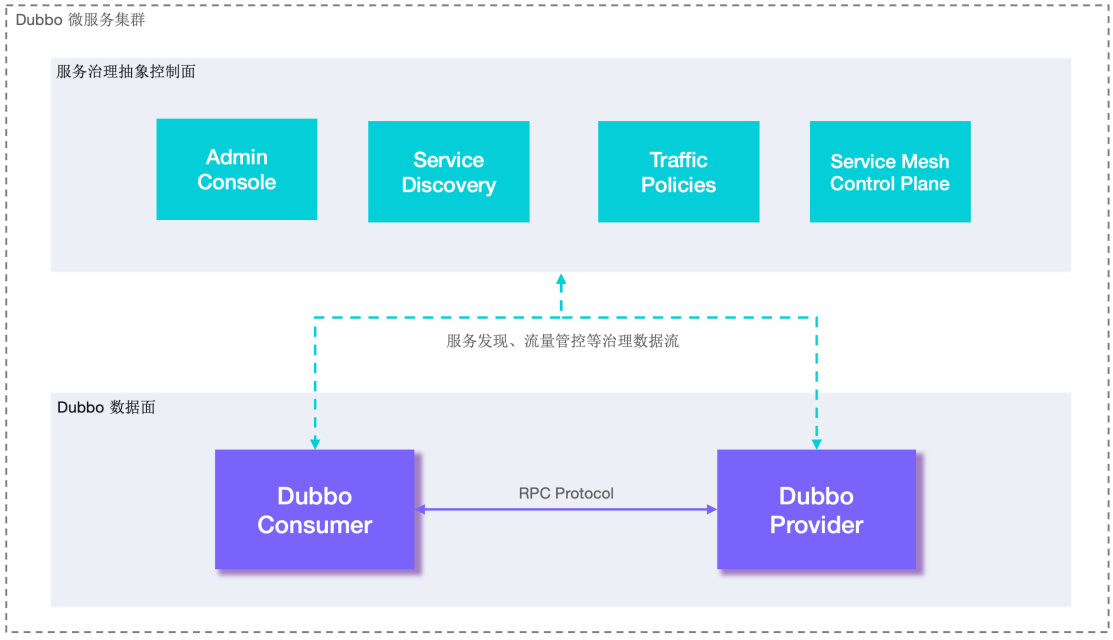
- **不是 gRPC 协议的替代品**

Dubbo 支持基于 gRPC 作为底层通信协议，在 Dubbo 模式下使用 gRPC 可以带来更好的开发体验，享有统一的编程模型和更低的服务治理接入成本

- **不只有 Java 语言实现**

自 Dubbo3 开始，Dubbo 提供了 Dubbo、Golang、Rust、Node.js 等多语言实现，未来会有更多的语言实现。

## 二、 核心架构




以上是 Dubbo 的工作原理图，从抽象架构上分为两层：服务治理抽象控制面和 Dubbo 数据面。

- **服务治理控制面**

服务治理控制面不是特指如注册中心类的单个具体组件，而是对 Dubbo 治理体系的抽象表达。控制面包含协调服务发现的注册中心、流量管控策略、Dubbo Admin 控制台等，如果采用了 Service Mesh 架构则还包含 Istio 等服务网格控制面。

- **Dubbo 数据面**

数据面代表集群部署的所有 Dubbo 进程，进程之间通过 RPC 协议实现数据交换，Dubbo 定义了微服务应用开发与调用规范并负责完成数据传输的编解码工作。

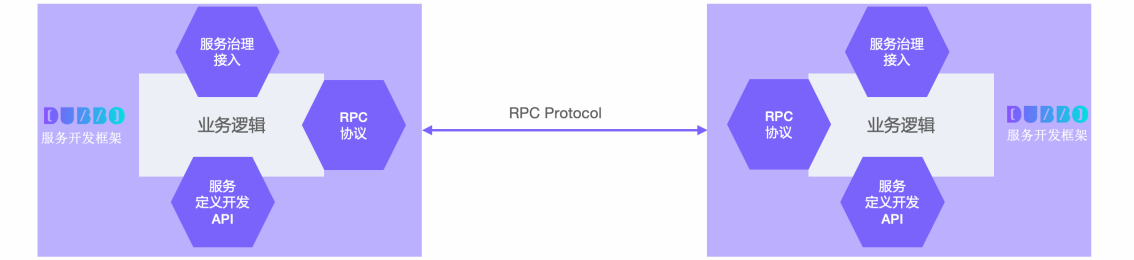
  - 服务消费者（Dubbo Consumer），发起业务调用或 RPC 通信的 Dubbo 进程。
  - 服务提供者（Dubbo Provider），接收业务调用或 RPC 通信的 Dubbo 进程。

## 1. Dubbo 数据面

从数据面视角，Dubbo 帮助解决了微服务实践中的以下问题：

- Dubbo 作为服务开发框架约束了微服务定义、开发与调用的规范，定义了服务治理流程及适配模式。

- Dubbo 作为 RPC 通信协议实现解决服务间数据传输的编解码问题。

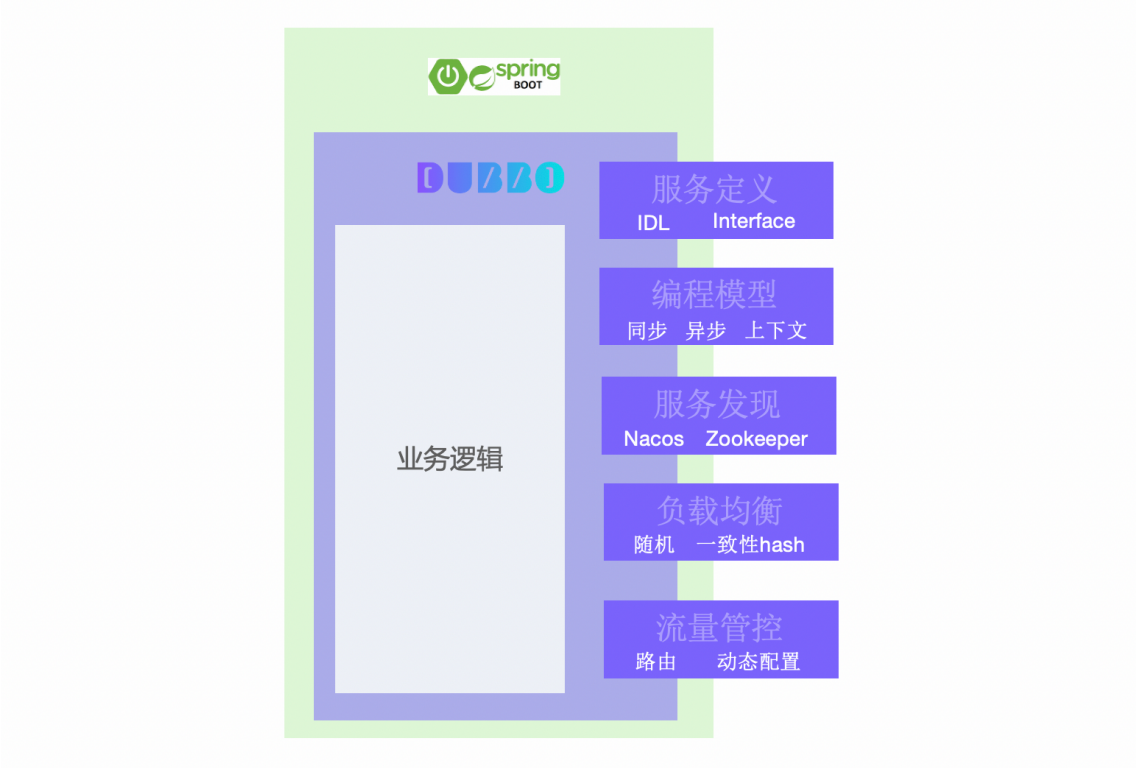


## 2. 服务开发框架

微服务的目标是构建足够小的、自包含的、独立演进的、可以随时部署运行的分布式应用程序，几乎每个语言都有类似的应用开发框架来帮助开发者快速构建此类微服务应用，比如 Java 微服务体系的 Spring Boot，它帮 Java 微服务开发者以最少的配置、最轻量的方式快速开发、打包、部署与运行应用。

微服务的分布式特性，使得应用间的依赖、网络交互、数据传输变得更频繁，因此不同的应用需要定义、暴露或调用 RPC 服务，那么这些 RPC 服务如何定义、如何与应用开发框架结合、服务调用行为如何控制？

这就是 Dubbo 服务开发框架的含义，Dubbo 在微服务应用开发框架之上抽象了一套 RPC 服务定义、暴露、调用与治理的编程范式，比如 Dubbo Java 作为服务开发框架，当运行在 Spring 体系时就是构建在 Spring Boot 应用开发框架之上的微服务开发框架，并在此之上抽象了一套 RPC 服务定义、暴露、调用与治理的编程范式。




Dubbo 作为服务开发框架包含的具体内容如下：

- **RPC 服务定义、开发范式**

比如 Dubbo 支持通过 IDL 定义服务，也支持编程语言特有的服务开发定义方式，如通过 Java Interface 定义服务。

- **RPC 服务发布与调用 API**

Dubbo 支持同步、异步、Reactive Streaming 等服务调用编程模式，还支持请求上下文 API、设置超时时间等。

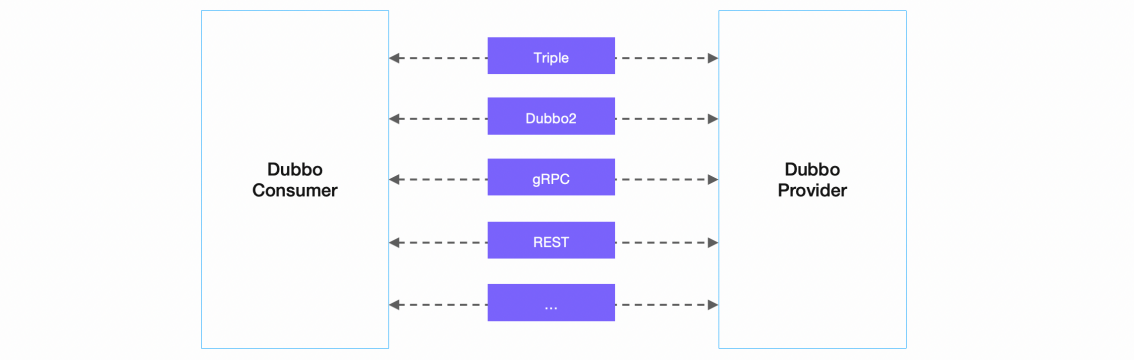
- **服务治理策略、流程与适配方式等**

作为服务框架数据面，Dubbo 定义了服务地址发现、负载均衡策略、基于规则的流量路由、Metrics 指标采集等服务治理抽象，并适配到特定的产品实现。

想了解如何使用 Dubbo 微服务框架进行业务编码？可以参考 Dubbo 官网关于多语言 SDK 的详细介绍。

## 3. 通信协议

Dubbo 从设计上不绑定任何一款特定通信协议，HTTP/2、REST、gRPC、JsonRPC、Thrift、Hessian2 等几乎所有主流的通信协议，Dubbo 框架都可以提供支持。这样的 Protocol 设计模式给构建微服务带来了最大的灵活性，开发者可以根据需要如性能、通用型等选择不同的通信协议，不再需要任何的代理来实现协议转换，甚至你还可以通过 Dubbo 实现不同协议间的迁移。



Dubbo Protocol 被设计支持扩展，您可以将内部私有协议适配到 Dubbo 框架上，进而将私有协议接入 Dubbo 体系，以享用 Dubbo 的开发体验与服务治理能力。比如 Dubbo3 的典型用户阿里巴巴，就是通过扩展支持 HSF 协议实现了内部 HSF 框架到 Dubbo3 框架的整体迁移。

Dubbo 还支持多协议暴露，您可以在单个端口上暴露多个协议，Dubbo Server 能够自动识别并确保请求被正确处理，也可以将同一个 RPC 服务发布在不同的端口（协议），为不同技术栈的调用方服务。

Dubbo 提供了两款内置高性能 Dubbo2、Triple（兼容 gRPC）协议实现，以满足部分微服务用户对高性能通信的诉求，两者最开始都设计和诞生于阿里巴巴内部的高性能通信业务场景。

- Dubbo2 协议是在 TCP 传输层协议之上设计的二进制通信协议。
- Triple 则是基于 HTTP/2 之上构建的支持流式模式的通信协议，并且 Triple 完全兼容 gRPC 但实现上做了更多的符合 Dubbo 框架特点的优化。

总的来说，Dubbo 对通信协议的支持具有以下特点：

- 不绑定通信协议
- 提供高性能通信协议实现
- 支持流式通信模型
- 不绑定序列化协议
- 支持单个服务的多协议暴露
- 支持单端口多协议发布
- 支持一个应用内多个服务使用不同通信协议

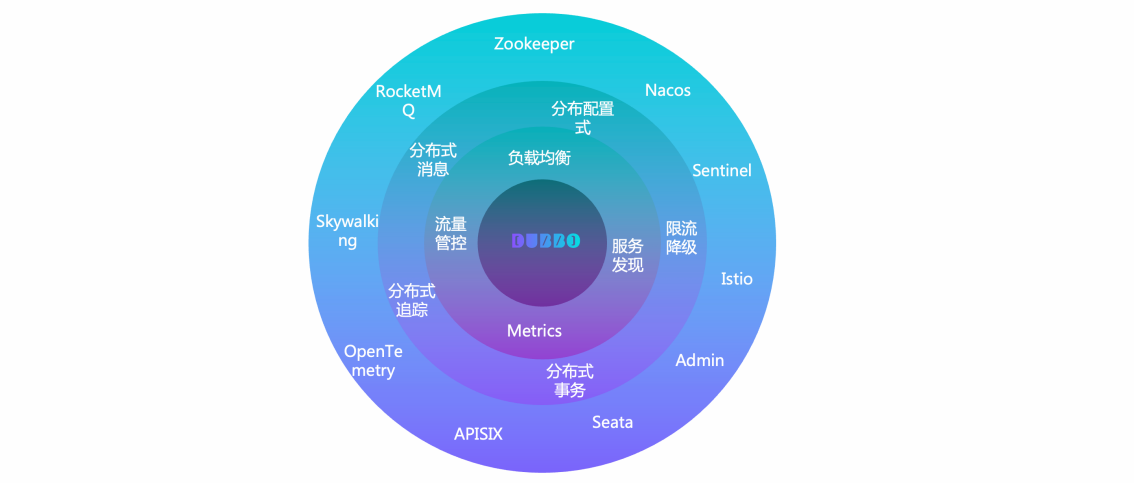
## 4. Dubbo 服务治理

服务开发框架解决了开发与通信的问题，但在微服务集群环境下，我们仍需要解决无状态服务节点动态变化、外部化配置、日志跟踪、可观测性、流量管理、高可用性、数据一致性等一系列问题，我们将这些问题统称为服务治理。

Dubbo 抽象了一套微服务治理模式并发布了对应的官方实现，服务治理可帮助简化微服务开发与运维，让开发者更专注在微服务业务本身。

### 1) 服务治理抽象

以下展示了 Dubbo 核心的服务治理功能定义。




- **地址发现**

Dubbo 服务发现具备高性能、支持大规模集群、服务级元数据配置等优势，默认提供 Nacos、Zookeeper、Consul 等多种注册中心适配，与 Spring Cloud、Kubernetes Service 模型打通，支持自定义扩展。

- **负载均衡**

Dubbo 默认提供加权随机、加权轮询、最少活跃请求数优先、最短响应时间优先、一致性哈希和自适应负载等策略

- **流量路由**

Dubbo 支持通过一系列流量规则控制服务调用的流量分布与行为，基于这些规则可以实现基于权重的比例流量分发、灰度验证、金丝雀发布、按请求参数的路由、同区域优先、超时配置、重试、限流降级等能力。

- **链路追踪**

Dubbo 官方通过适配 OpenTelemetry 提供了对 Tracing 全链路追踪支持，用户可以接入支持 OpenTelemetry 标准的产品如 Skywalking、Zipkin 等。另外，很多社区如 Skywalking、Zipkin 等在官方也提供了对 Dubbo 的适配。

- **可观测性**

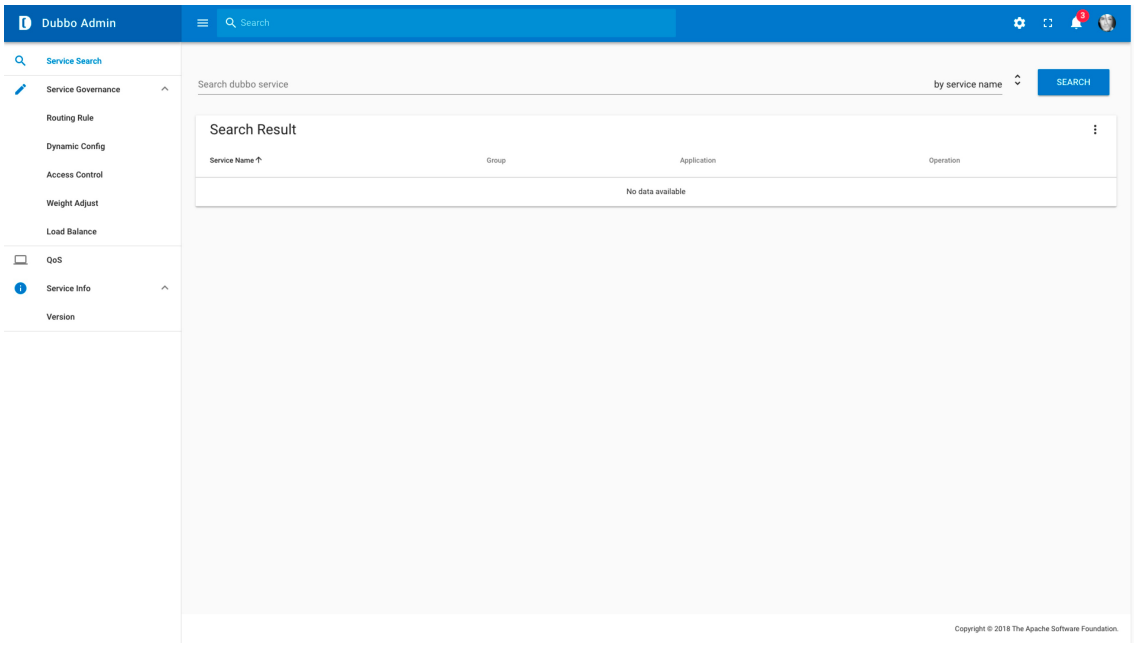
Dubbo 实例通过 Prometheus 等上报 QPS、RT、请求次数、成功率、异常次数等多维度的可观测指标帮助了解服务运行状态，通过接入 Grafana、Admin 控制台帮助实现数据指标可视化展示。

Dubbo 服务治理生态还提供了对 API 网关、限流降级、数据一致性、认证鉴权等场景的适配支持。

### 2) Dubbo Admin

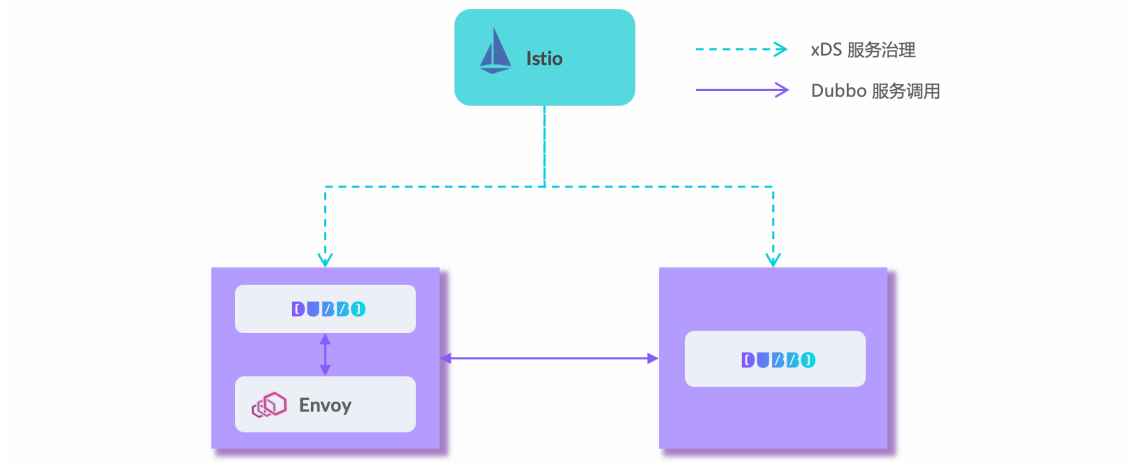
Admin 控制台提供了 Dubbo 集群的可视化视图，通过 Admin 你可以完成集群的几乎所有管控工作。

- 查询服务、应用或机器状态
- 创建项目、服务测试、文档管理等
- 查看集群实时流量、定位异常问题等
- 流量比例分发、参数路由等流量管控规则下发




### 3) 服务网格

将 Dubbo 接入 Istio 等服务网格治理体系。




## 三、 Dubbo 核心特点

### 1. 快速易用

无论你是计划采用微服务架构开发一套全新的业务系统，还是准备将已有业务从单体架构迁移到微服务架构，Dubbo 框架都可以帮助到你。Dubbo 让微服务开发变

得非常容易，它允许你选择多种编程语言、使用任意通信协议，并且它还提供了一系列针对微服务场景的开发、测试工具帮助提升研发效率。

### 1) 多语言 SDK

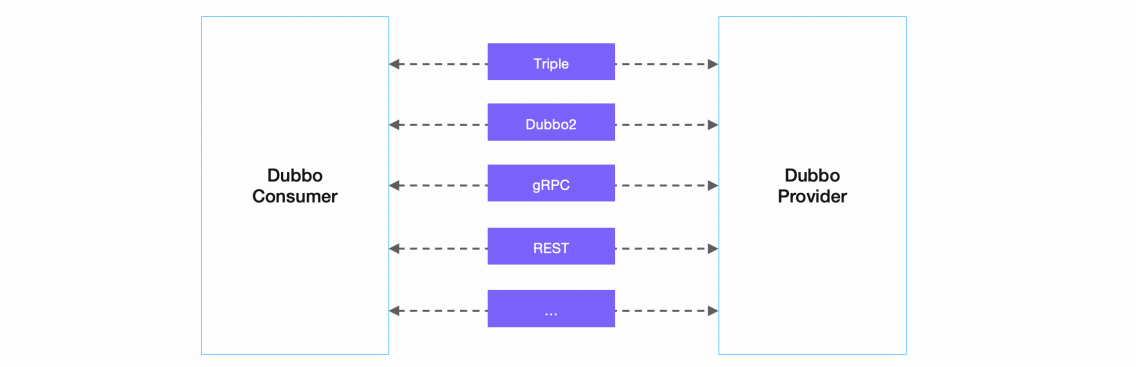
Dubbo 提供几乎所有主流语言的 SDK 实现，定义了一套统一的微服务开发范式。Dubbo 与每种语言体系的主流应用开发框架做了适配，总体编程方式、配置符合大多数开发者已有编程习惯。

比如在 Java 语言体系下，你可以使用 `dubbo-spring-boot-starter` 来开发符合 Spring、Spring Boot 模式的微服务应用，开发 Dubbo 应用只是为 Spring Bean 添加几个注解、完善 application.properties 配置文件。


### 2) 任意通信协议

Dubbo 微服务间远程通信实现细节，支持 HTTP、HTTP/2、gRPC、TCP 等所有主流通信协议。与普通 RPC 框架不同，Dubbo 不是某个单一 RPC 协议的实现，它通过上层的 RPC 抽象可以将任意 RPC 协议接入 Dubbo 的开发、治理体系。

多协议支持让用户选型，多协议迁移、互通等变得更灵活。

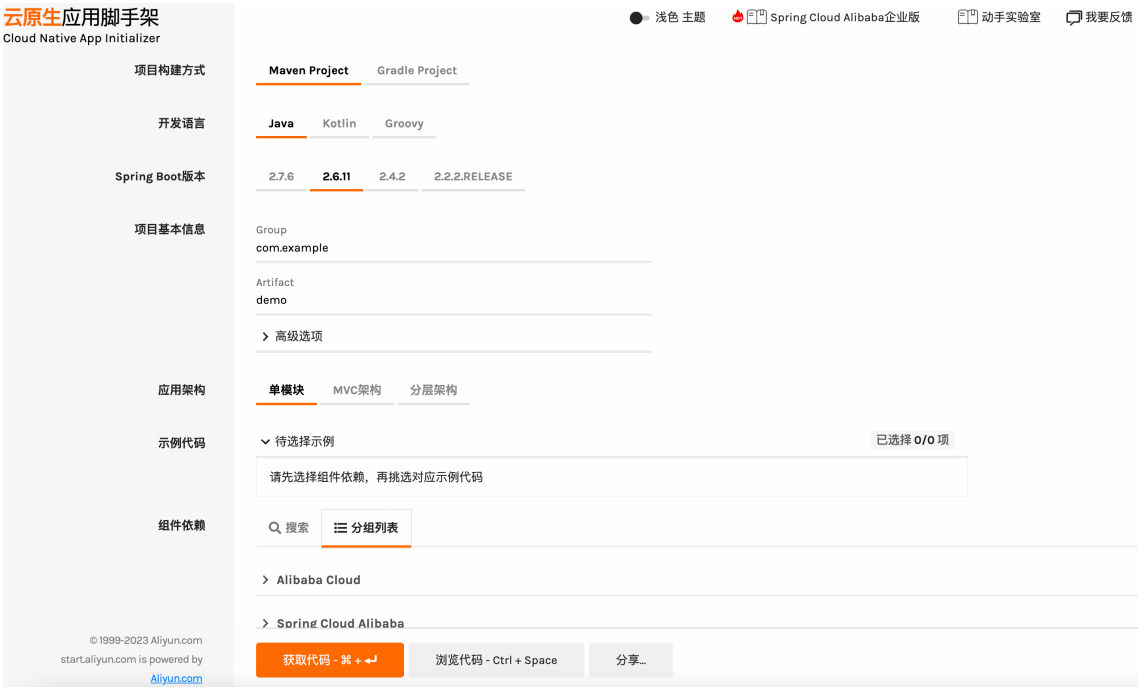



### 3) 加速微服务开发

- **项目脚手架**

项目脚手架 Dubbo 项目创建、依赖管理更容易。

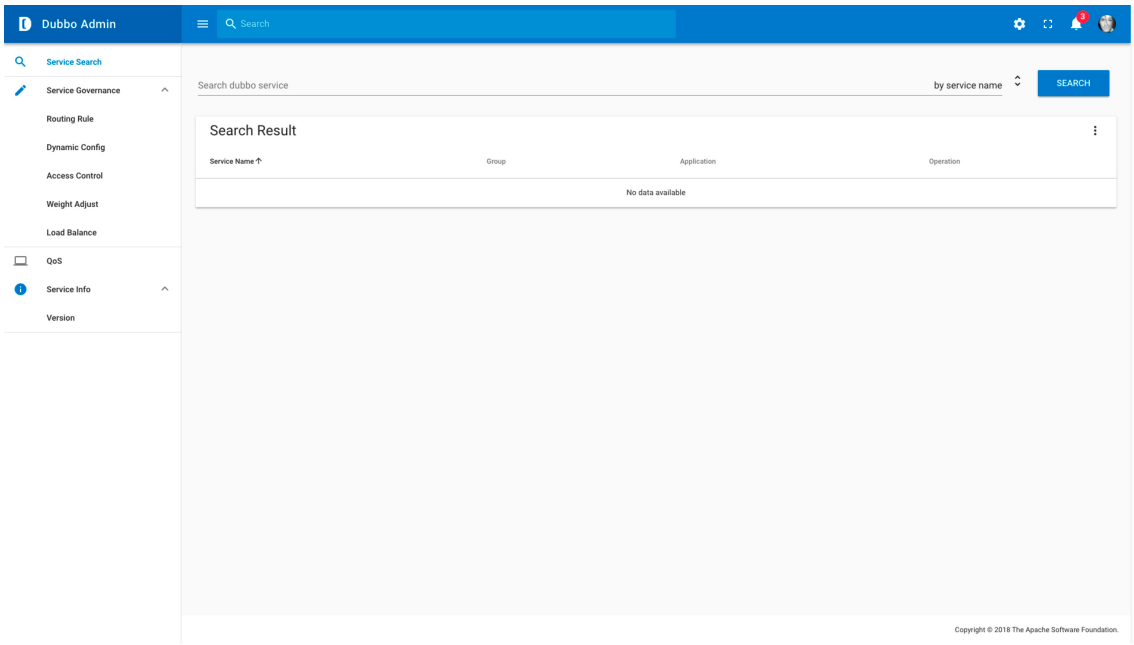
比如通过如下可视化界面，勾选 Dubbo 版本、Zookeeper 注册中心以及必要的微服务生态选项后，一个完整的 Dubbo 项目模板就可以自动生成，接下来基于脚手架项目添加业务逻辑就可以了。



- **开发测试**

相比于单体应用，微服务分布式的特性会让不同组织之间的研发协同变得困难，这时我们需要有效的配套工具，用来提升整体的微服务研发效率。

Dubbo 从内核设计和实现阶段就考虑了如何解决开发、测试与运维问题，配合官方提供的生态工具，可以实现服务测试、服务 Mock、文档管理、单机运维等能力，并通过 Dubbo Admin 控制台将所有操作都可视化的展现出来。




## 2. 超高性能

Dubbo 被设计用于解决阿里巴巴超大规模的电商微服务集群实践，并在各个行业头部企业经过多年的十万、百万规模的微服务实践检验，因此，Dubbo 在通信性能、稳定性方面具有无可比拟的优势，非常适合构建近乎无限水平伸缩的微服务集群，这也是 Dubbo 从实践层面优于业界很多同类的产品的巨大优势。

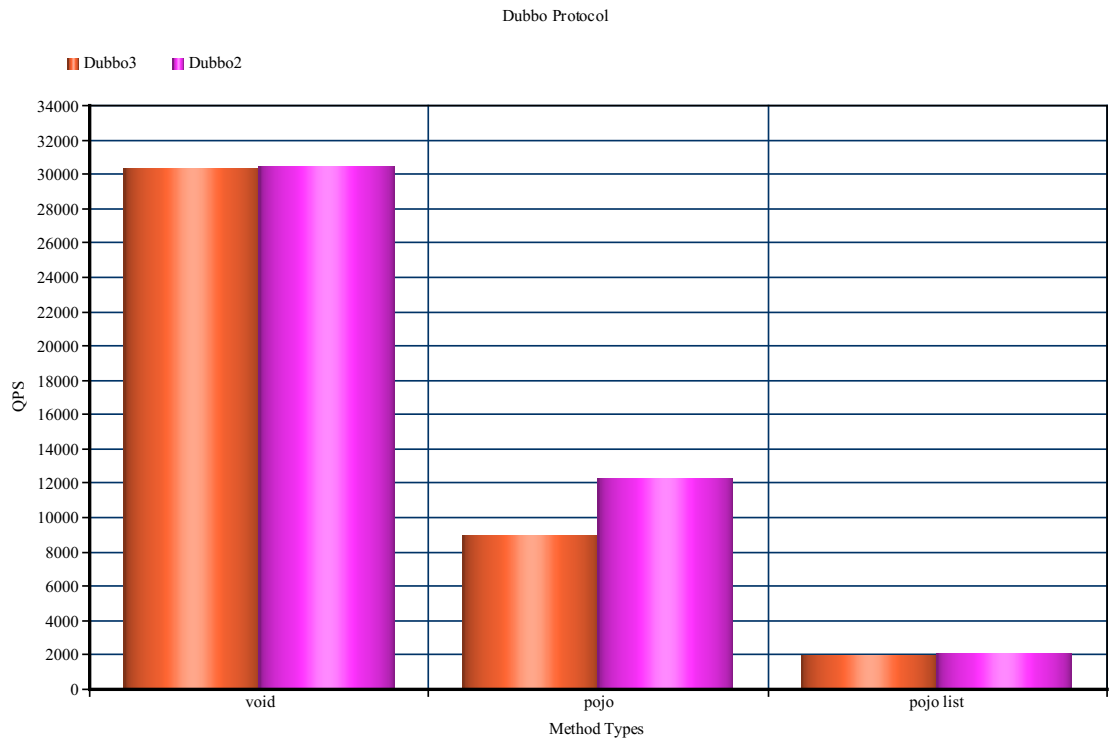
### 1) 高性能数据传输

Dubbo 内置支持 Dubbo2、Triple 两款高性能通信协议。其中

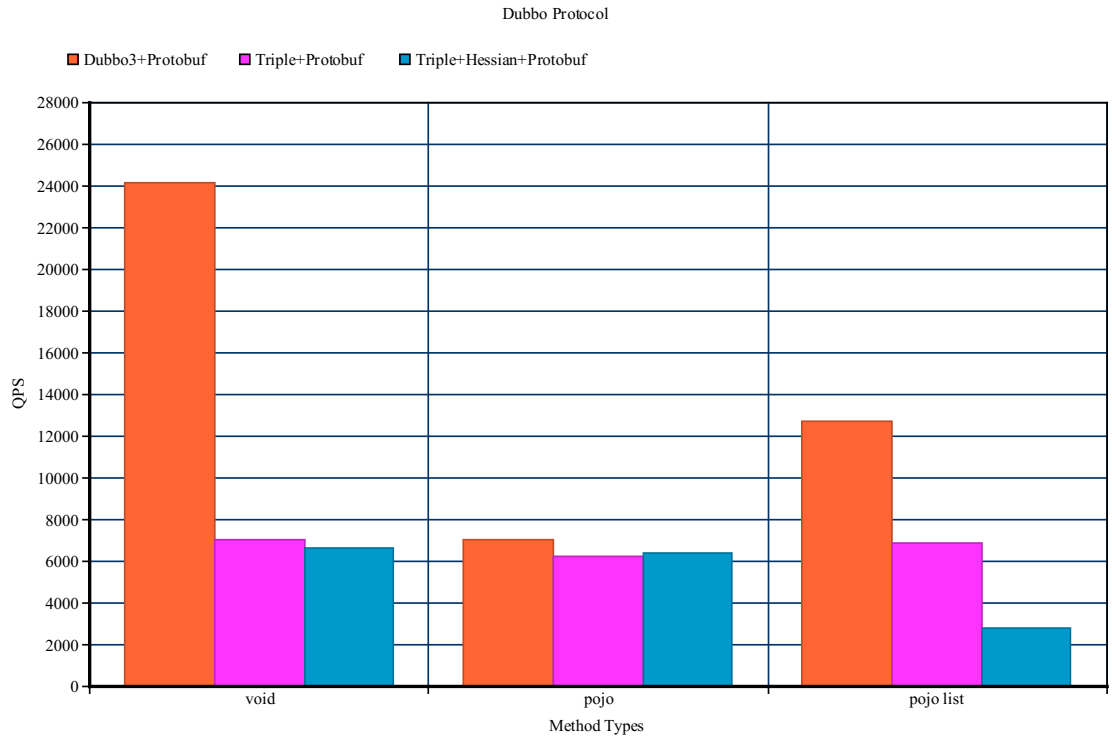
- Dubbo2 是基于 TCP 传输协议之上构建的二进制私有 RPC 通信协议，是一款非常简单、紧凑、高效的通信协议。

- Triple 是基于 HTTP/2 的新一代 RPC 通信协议，在网关穿透性、通用性以及 Streaming 通信上具备优势，Triple 完全兼容 gRPC 协议。

Dubbo2 & Triple benchmark 性能指标：




Dubbo 协议在不同版本的实现对比



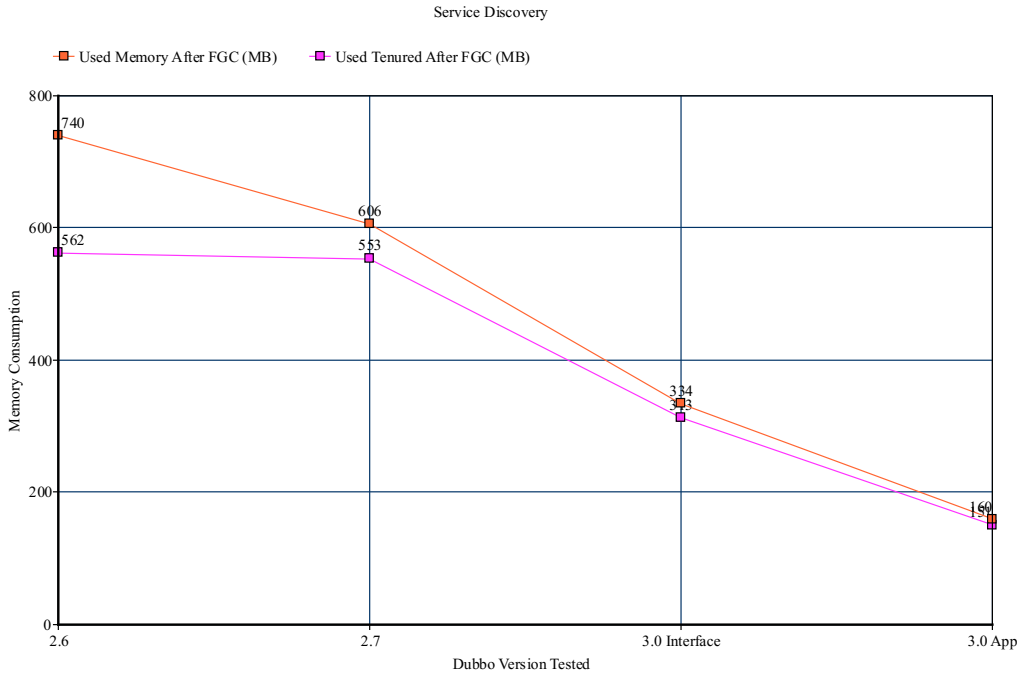

Triple vs Dubbo

### 2) 构建可伸缩的微服务集群

业务增长带来了集群规模的快速增长，而集群规模的增长会对服务治理架构带来挑战：

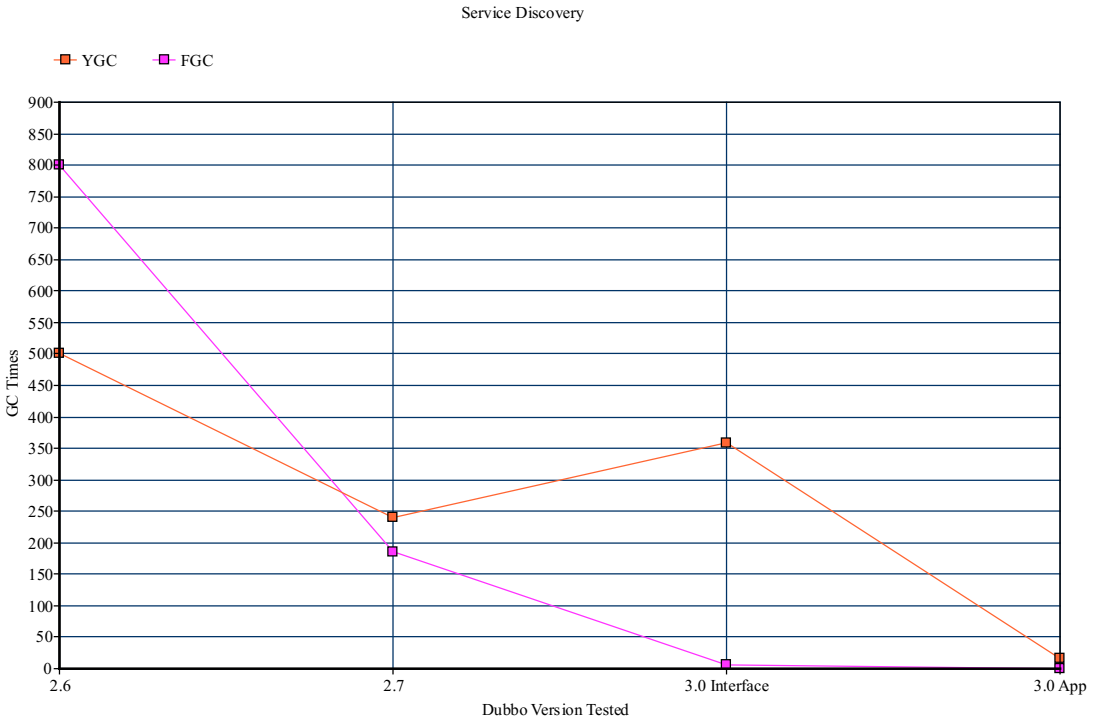
- 注册中心的存储容量瓶颈
- 节点动态变化带来的地址推送与解析效率下降
- 消费端存储大量网络地址的资源开销
- 复杂的网络链接管理
- 高峰期的流量无损上下线
- 异常节点的自动节点管理

以上内容直接关系到微服务集群的稳定性，因此很容易成为影响集群和业务增长的瓶颈，集群规模越大，问题带来的影响面也就被进一步放大。很多开发者可能会想只有几个应用而已，当前不需要并不关心集群规模，但作为技术架构选型的关键因素之一，我们还是要充分考虑微服务集群未来的可伸缩性。并且基于对业界大量微服务架构和框架实现的调研，一些产品的性能瓶颈点可能很快就会到来（部分产品所能高效支持的瓶颈节点规模阈值都是比较低的，比如几十个应用、数百个节点）。




服务发现模型内存占用变化

- Dubbo3 接口级服务发现模型，常驻内存较 2.x 版本下降约 50%
- Dubbo3 应用级服务发现模型，常驻内存较 2.x 版本下降约 75%




服务发现模型 GC 变化

- Dubbo3 接口级服务发现模型，YGC 次数 2.x 版本大幅下降，从数百次下降到十几次。
- Dubbo3 应用级服务发现模型，FGC 次数 2.x 版本大幅下降，从数百次下降到零次。

Dubbo 的优势在于近乎无限水平扩容的集群规模，在阿里巴巴双十一场景万亿次调用的实践检验，通过本书后续内容了解 Dubbo 构建生产可用的、可伸缩的大规模微服务集群背后的原理。

## 3. 服务治理

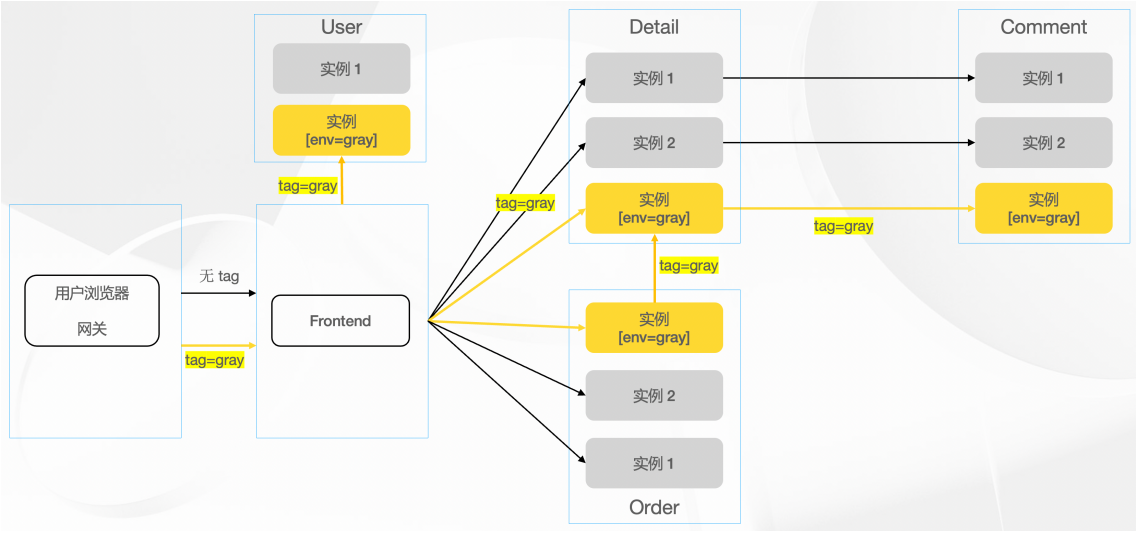
### 1) 流量管控

在地址发现和负载均衡机制之外，Dubbo 丰富的流量管控规则可以控制服务间的流量走向和 API 调用，基于这些规则可以实现在运行期动态的调整服务行为如超时时间、重试次数、限流参数等，通过控制流量分布可以实现 A/B 测试、金丝雀发布、多版本按比例流量分配、条件匹配路由、黑白名单等，提高系统稳定性。

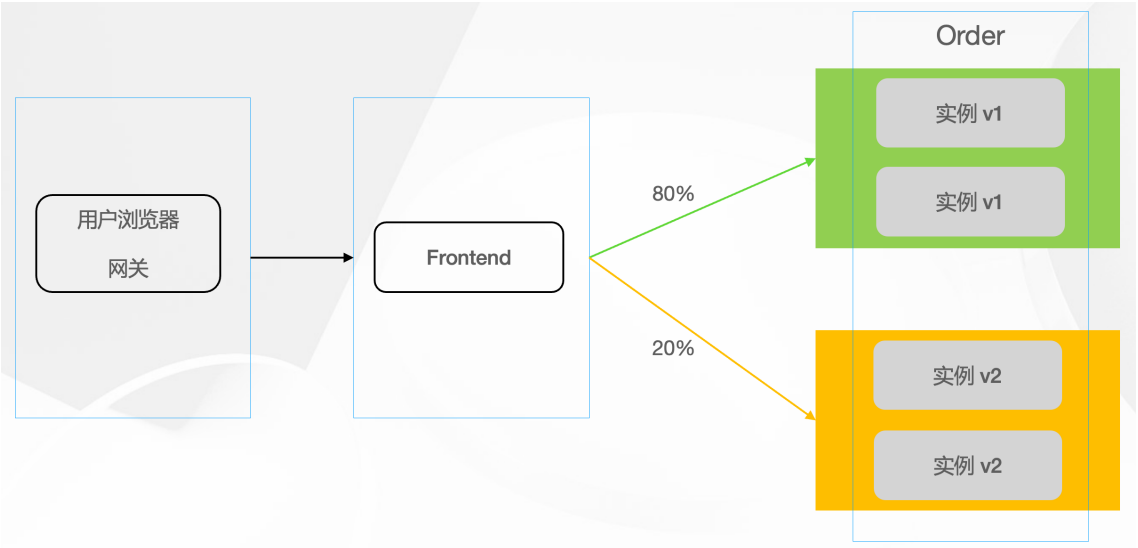
**Dubbo 流量管控能解决哪些问题？**

**场景一：**搭建多套独立的逻辑测试环境。

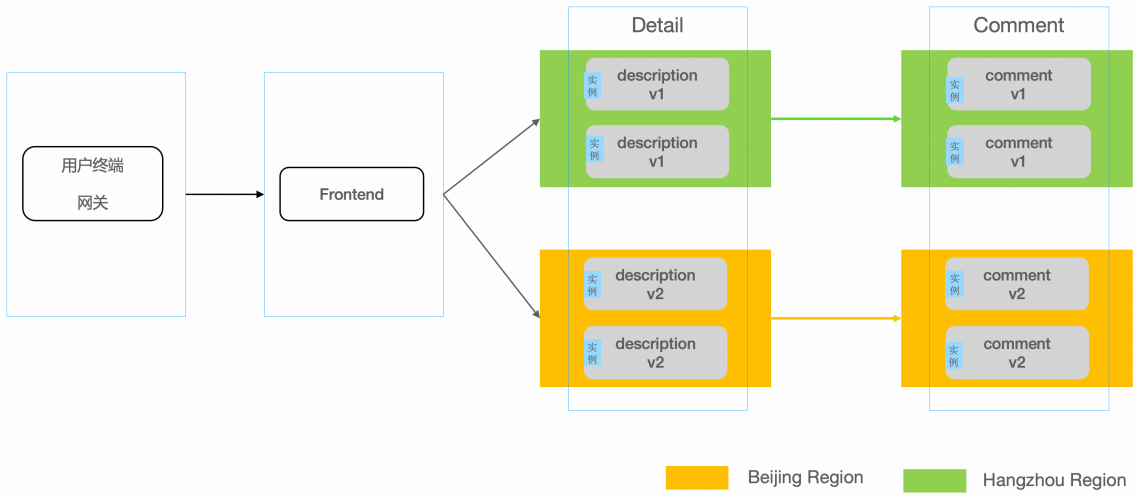
**场景二：**搭建一套完全隔离的线上灰度环境用来部署新版本服务。




**场景三：**金丝雀发布




**场景四：**同区域优先。当应用部署在多个不同机房/区域的时候，优先调用同机房/区域的服务提供者，避免了跨区域带来的网络延时，从而减少了调用的响应时间。



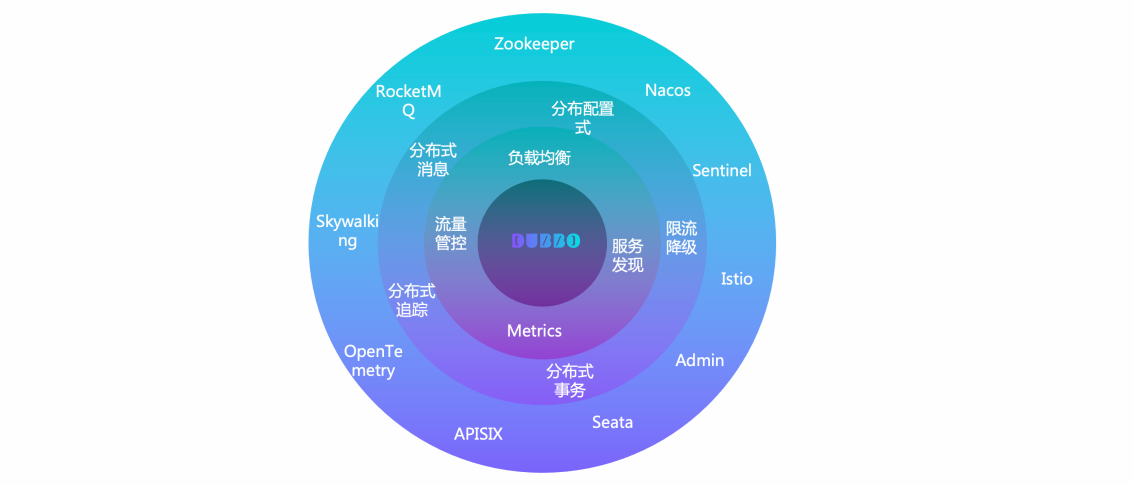

除了以上几个典型场景，我们还可以基于 Dubbo 支持的流量管控规则实现微服务场景中更丰富的流量管控，如：

- 动态调整超时时间
- 服务重试
- 访问日志
- 同区域优先
- 灰度环境隔离
- 参数路由
- 按权重比例分流
- 金丝雀发布
- 服务降级
- 实例临时拉黑
- 指定机器导流

可以在 Dubbo 官网【流量管理任务】中了解以上实践场景细节。背后的规则定义与工作原理在文档中也有相应解释。

### 2) 微服务生态

围绕 Dubbo 我们构建了完善的微服务治理生态，对于绝大多数服务治理需求，通过简单几行配置即可开启。对于官方尚未适配的组件或者用户内部系统，也可以通过 Dubbo 扩展机制轻松适配。




### 3) 可视化控制台

Dubbo Admin 是 Dubbo 官方提供的可视化 Web 交互控制台，基于 Admin 你可以实时监测集群流量、服务部署状态、排查诊断问题。

### 4) 安全体系

Dubbo 支持基于 TLS 的 HTTP、HTTP/2、TCP 数据传输通道，并且提供认证、鉴权策略，让开发者实现更细粒度的资源访问控制。

### 5) 服务网格

基于 Dubbo 开发的服务可以透明的接入 Istio 等服务网格体系，Dubbo 支持基于 Envoy 的流量拦截方式，也支持更加轻量的 Proxyless Mesh 部署模式。

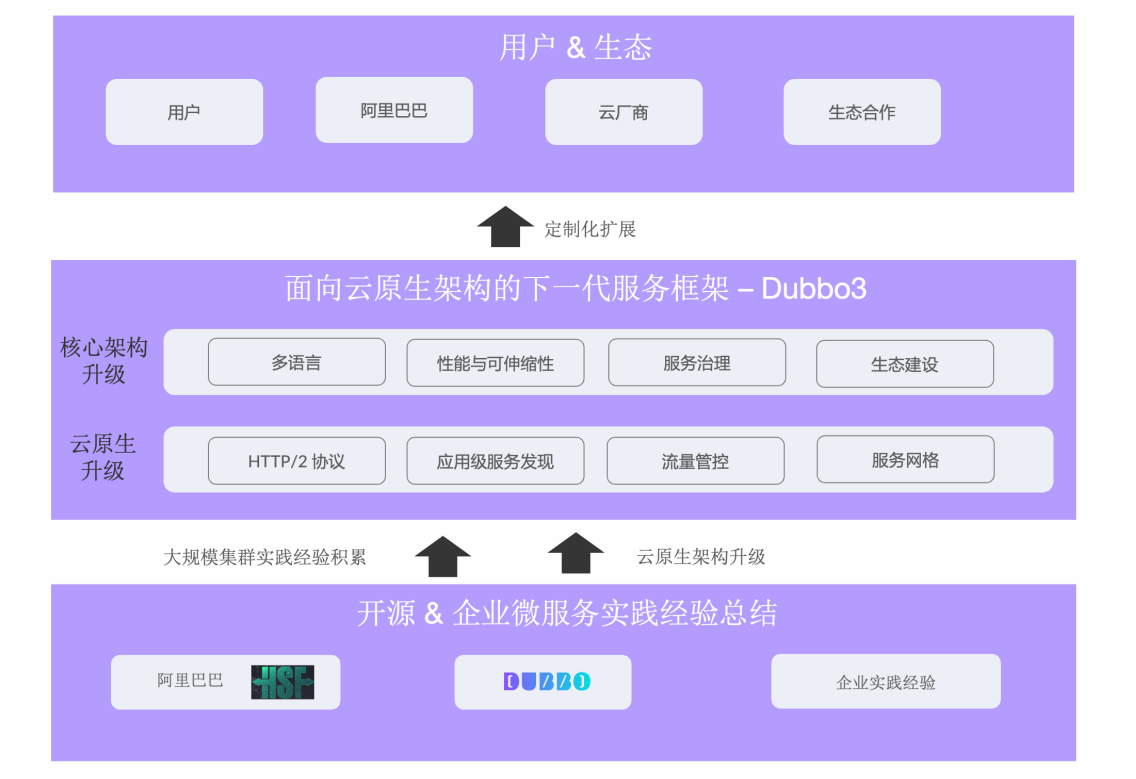
## 4. 生产环境检验

Apache Dubbo 是一款有着数以万计企业用户的国际化开源项目，经过了多年大规模集群生产环境的检验，影响了数百万开发者，带动了大量微服务开源生态发展。Dubbo 从企业实践中孵化并走向开源，又迅速在开源社区获得了成功，大量的生产实践用户是 Dubbo 长期保持先进性、稳定性和活跃度的核心驱动力。

### 1) Dubbo 在阿里巴巴的应用

Dubbo 设计用于解决阿里巴巴内部复杂的电商微服务集群的开发和治理问题，在 2020 年，阿里巴巴与 Apache Dubbo 社区共同合作，基于 Dubbo2&HSF2 发布了面向云原生架构的下一代服务框架——Dubbo3，目前，Dubbo3 已经完全取代 HSF、Dubbo2 成为阿里巴巴内部统一的服务框架，成功的跑在了数十万应用、数百万节点的双十一集群之上。

Dubbo3 吸取了 HSF2 框架所有大规模微服务集群的治理经验，解决了 Dubbo2 架构设计上长期积累的一些缺陷，同时增加了一系列面向云原生架构的新特性。




- 阿里巴巴结合 HSF 框架的大规模集群实践经验，基于 Apache Dubbo、开源社区需求等推出了面向云原生架构的全新服务框架——Dubbo3，Dubbo3 在完全兼容之前 API 模式的情况下，完成了彻底的云原生架构升级。

- Dubbo 的高度可扩展能力是其广泛适用的重要前提，阿里巴巴基于 Dubbo3 内核维护了一套内部特有的适配插件体系以实现平滑升级，这包括注册中心扩展、路由组件扩展、监控组件扩展等。

- 几乎所有主流云厂商、主流微服务开源社区都提供了 Dubbo 适配或托管服务。

**2) 更多案例**

据 Wanted、Who's、Using Dubbo 统计，Dubbo 已知部分典型用户包括：

网联清算、银联商务、中国人寿、中国平安、中国银行、人民银行、工商银行、招商证券、平安保险、中国人寿、阿里巴巴、滴滴出行、携程网、小米、斗鱼直播、瓜子二手车、金蝶、亚信科技、中国电信、文思海辉、中科软、科大讯飞、恒生电子、红星凯美龙、海尔、新东方、软通动力、中远海运、昆明航空、中通快递、顺丰科技、普华永道等。

## 四、 与 gRPC、Spring Cloud、Istio 的关系

很多开发者经常会问到 Apache Dubbo 与 Spring Cloud、gRPC 以及一些 Service Mesh 项目如 Istio 的关系，要解释清楚它们的关系并不困难，你只需要跟随这篇文章和 Dubbo 文档做一些更深入的了解，但总的来说，它们之间有些能力是重合的，但在一些场景你可以把它们放在一起使用。

虽然这是一篇 Dubbo 维护者写的文档，我们仍会尽力将 Dubbo 与其他组件之间的联系与差异客观、透明的展现出来，但考虑到每个人对不同产品的熟悉程度不一，这里的个别表述可能并不完全准确，希望能给开发者带来帮助。

### 1. Dubbo 与 Spring Cloud

从下图我们可以看出，Dubbo 和 Spring Cloud 有很多相似之处，它们都在整个架构图的相同位置并提供一些相似的功能。


- **Dubbo 和 Spring Cloud 都侧重在对分布式系统中常见问题模式的抽象**（如服务发现、负载均衡、动态配置等），同时对每一个问题都提供了配套组件实现，形成了一套微服务整体解决方案，让使用 Dubbo 及 Spring Cloud 的用户在开发微服务应用时可以专注在业务逻辑开发上。

- **Dubbo 和 Spring Cloud 都完全兼容 Spring 体系的应用开发模式**，Dubbo 对 Spring 应用开发框架、Spring Boot 微服务框架都做了很好的适配，由于 Spring Cloud 出自 Spring 体系，在这一点上自然更不必多说。

虽然两者有很多相似之处，但由于它们在诞生背景与架构设计上的巨大差异，两者在性能、适用的微服务集群规模、生产稳定性保障、服务治理等方面都有很大差异。

Spring Cloud 的优势在于：

- 同样都支持 Sprig 开发体系的情况下，Spring Cloud 得到更多的原生支持。

- 对一些常用的微服务模式做了抽象如服务发现、动态配置、异步消息等，同时包括一些批处理任务、定时任务、持久化数据访问等领域也有涉猎。

- 基于 HTTP 的通信模式，加上相对比较完善的入门文档和演示 demo 和 starters，让开发者在第一感觉上更易于上手。

Spring Cloud 的问题有：

- 只提供抽象模式的定义不提供官方稳定实现，开发者只能寻求类似 Netflix、Alibaba、Azure 等不同厂商的实现套件，而每个厂商支持的完善度、稳定性、活跃度各异。

- 有微服务全家桶却不是能拿来就用的全家桶，demo 上手容易，但落地推广与长期使用的成本非常高。

- 欠缺服务治理能力，尤其是流量管控方面如负载均衡、流量路由方便能力都比较弱。

- 编程模型与通信协议绑定 HTTP，在性能、与其他 RPC 体系互通上存在障碍。

- 总体架构与实现只适用于小规模微服务集群实践，当集群规模增长后就会遇到地址推送效率、内存占用等各种瓶颈的问题，但此时迁移到其他体系却很难实现。

- 很多微服务实践场景的问题需要用户独自解决，比如优雅停机、启动预热、服务测试，再比如双注册、双订阅、延迟注册、服务按分组隔离、集群容错等。

而以上这些点，都是 Dubbo 的优势所在：

- 完全支持 Spring & Spring Boot 开发模式，同时在服务发现、动态配置等基础模式上提供与 Spring Cloud 对等的能力。

- 是企业级微服务实践方案的整体输出，Dubbo 考虑到了企业微服务实践中会遇到的各种问题如优雅上下线、多注册中心、流量管理等，因此其在生产环境的长期维护成本更低。

- 在通信协议和编码上选择更灵活，包括 rpc 通信层协议如 HTTP、HTTP/2（Triple、gRPC）、TCP 二进制协议、rest 等，序列化编码协议 Protobuf、JSON、Hessian2 等，支持单端口多协议。

- Dubbo 从设计上突出服务服务治理能力，如权重动态调整、标签路由、条件路由等，支持 Proxyless 等多种模式接入 Service Mesh 体系。

- 高性能的 RPC 协议编码与实现。

- Dubbo 是在超大规模微服务集群实践场景下开发的框架，可以做到百万实例规模的集群水平扩容，应对集群增长带来的各种问题。

- Dubbo 提供 Java 外的多语言实现，使得构建多语言异构的微服务体系成为可能。

如果您的目标是构建企业级应用，并期待在未来的持久维护中能够更省心、更稳定，我们建议你能更深入的了解 Dubbo 的使用和其提供的能力。

**备注**：Dubbo 在入门资料上的欠缺是对比 Spring Cloud 的一个劣势，这体现在依赖配置管理、文档、demo 示例完善度上，当前整个社区在重点投入这一部分的建设，期望能降低用户在第一天体验和学习 Dubbo 时的门槛，不让开发者因为缺乏文档而错失 Dubbo 这样一款优秀的产品。

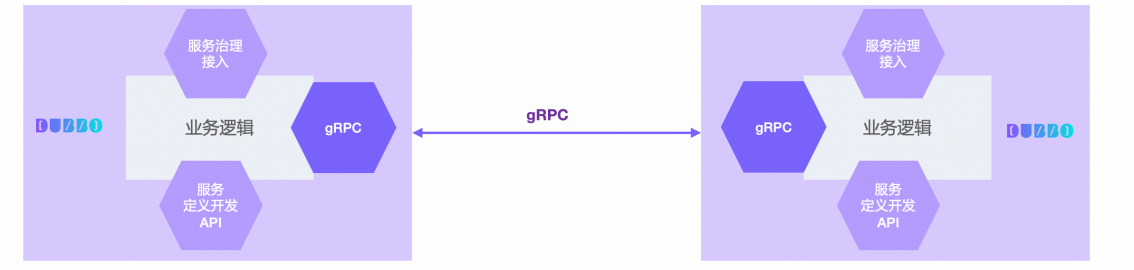
## 2. Dubbo 与 gRPC

Dubbo 与 gRPC 最大的差异在于两者的定位上：

- gRPC 定位为一款 RPC 框架，Google 推出它的核心目标是定义云原生时代的 rpc 通信规范与标准实现。

- Dubbo 定位是一款微服务开发框架，它侧重解决微服务实践从服务定义、开发、通信到治理的问题，因此 Dubbo 同时提供了 RPC 通信、与应用开发框架的适配、服务治理等能力。

Dubbo 不绑定特定的通信协议，即 Dubbo 服务间可通过多种 RPC 协议通信并支持灵活切换。因此，你可以在 Dubbo 开发的微服务中选用 gRPC 通信，Dubbo 完全兼容 gRPC，并将 gRPC 设计为内置原生支持的协议之一。




如果您看中基于 HTTP/2 的通信协议、基于 Protobuf 的服务定义，并基于此决定选型 gRPC 作为微服务开发框架，那很有可能您会在未来的微服务业务开发中遇到障碍，这主要源于 gRPC 没有为开发者提供以下能力：

- 缺乏与业务应用框架集成的开发模式，用户需要基于 gRPC 底层的 RPC API 定义、发布或调用微服务，中间可能还有与业务应用开发框架整合的问题。

- 缺乏微服务周边生态扩展与适配，如服务发现、限流降级、链路追踪等没有多少可供选择的官方实现，且扩展起来非常困难。

- 缺乏服务治理能力，作为一款 rpc 框架，缺乏对服务治理能力的抽象。

因此，gRPC 更适合作为底层的通信协议规范或编解码包，而 Dubbo 则可用作微服务整体解决方案。对于 gRPC 协议，我们推荐的使用模式 Dubbo+gRPC 的组合，这个时候，gRPC 只是隐藏在底层的一个通信协议，不被微服务开发者感知，开发者基于 Dubbo 提供的 API 和配置开发服务，并基于 dubbo 的服务治理能力治理服务，在未来，开发者还能使用 Dubbo 生态还开源的 IDL 配套工具管理服务定义与发布。

如果我们忽略 gRPC 在应用开发框架侧的空白，只考虑如何给 gRPC 带来服务治理能力，则另一种可以采用的模式就是在 Service Mesh 架构下使用 gRPC，这就引出了我们下一小节要讨论的内容：Dubbo 与 Service Mesh 架构的关系。

## 3. Dubbo 与 Istio

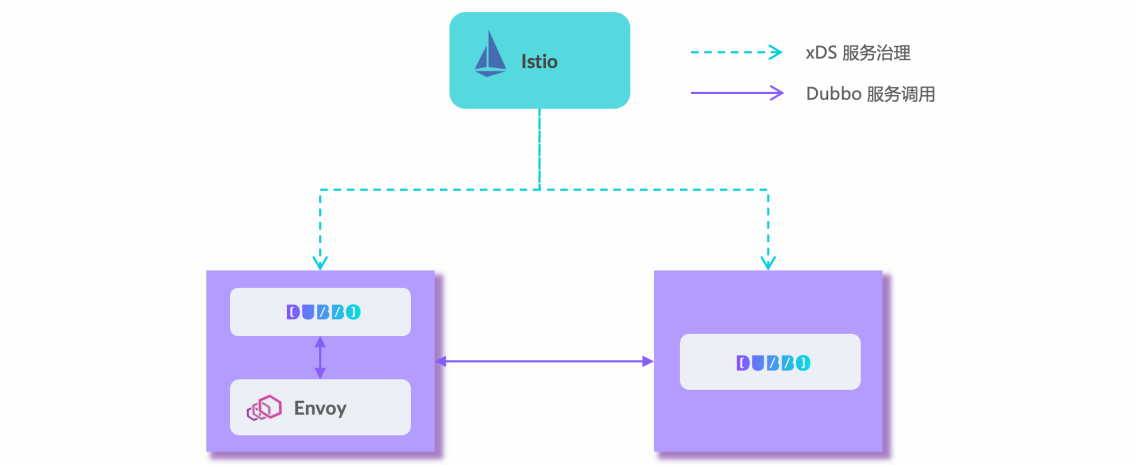
Service Mesh 是近年来在云原生背景下诞生的一种微服务架构，在 Kubernetes 体系下，让微服务开发中的更多能力如流量拦截、服务治理等下沉并成为基础设施，让微服务开发、升级更轻量。Istio 是 Service Mesh 的开源代表实现，它从部署架构上分为数据面与控制面，从这一点上与 Dubbo 总体架构是基本一致的，Istio 带来的主要变化在于：

- **数据面**，Istio 通过引入 Sidecar 实现了对服务流量的透明拦截，Sidecar 通常是与 Dubbo 等开发的传统微服务组件部署在一起。

- **控制面**，将之前抽象的服务治理中心聚合为一个具有统一实现的具体组件，并实现了与底层基础设施如 Kubernetes 无缝适配。

Dubbo 已经实现了对 Istio 体系的全面接入，可以用 Istio 控制面治理 Dubbo 服务，而在数据面部署架构上，针对 Sidecar 引入的复杂性与性能问题，Dubbo 还支持无代理的 Proxyless 模式。

除此之外，Dubbo Mesh 体系还解决了 Istio 架构落地过程中的很多问题，包括提供更灵活的数据面部署架构、更低的迁移成本等。




从数据面的视角，Dubbo 支持如下两种开发和部署模式，可以通过 Istio、Consul、Linkerd 等控制面组件实现对数据面服务的治理。

- **Proxy 模式**，Dubbo 与 Envoy 一起部署，Dubbo 作为编程框架&协议通信组件存在，流量管控由 Envoy 与 Istio 控制面交互实现。

- **Proxyless 模式**，Dubbo 进程保持独立部署，Dubbo 通过标准 xDS 协议直接接入 Istio 等控制面组件。

从控制面视角，Dubbo 可接入原生 Istio 标准控制面和规则体系，而对于一些 Dubbo 老版本用户，Dubbo Mesh 提供了平滑迁移方案，具体请查看 Dubbo Mesh 服务网格。

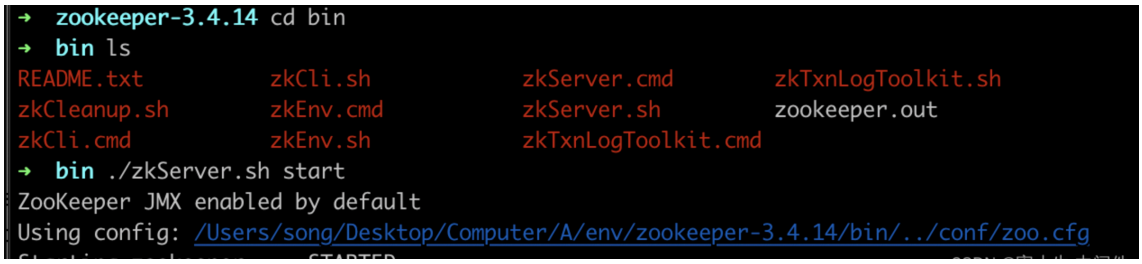
# 提供者服务发布过程

## 一、 从一个服务提供者的 Demo 说起

为了更方便了解原理，我们先来编写一个 Demo，从例子中来看源码实现。

### 1. 启动 Zookeeper

为了 Demo 可以正常启动，需要我们先在本地启动一个 Zookeeper 如下图所示：




### 2. 服务提供者

接下来给大家贴一下示例源码，这个源码来源于 Dubbo 源码目录的 dubbo-demo/dubbo-demo-api 目录下面的 dubbo-demo-api-provider 子项目，这里我做了删减，方便看核心代码。

首先我们定义一个服务接口如下所示：

```
import java.util.concurrent.CompletableFuture;
public interface DemoService {
    /**
     * 同步处理的服务方法
     * @param name
     * @return
     */
    String sayHello(String name);

    /**
     * 用于异步处理的服务方法
     * @param name
     * @return
     */
    default CompletableFuture<String> sayHelloAsync(String name) {
        return CompletableFuture.completedFuture(sayHello(name));
    }
}
```

服务实现类如下：

```
import org.apache.dubbo.rpc.RpcContext;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.concurrent.CompletableFuture;

public class DemoServiceImpl implements DemoService {
    private static final Logger logger = LoggerFactory.getLogger(DemoServiceImpl.class);

    @Override
    public String sayHello(String name) {
        logger.info("Hello " + name + ", request from consumer: " +
RpcContext.getServiceContext().getRemoteAddress());
        return "Hello " + name + ", response from provider: " +
RpcContext.getServiceContext().getLocalAddress();
    }

    @Override
    public CompletableFuture<String> sayHelloAsync(String name) {
        return null;
    }

}
```

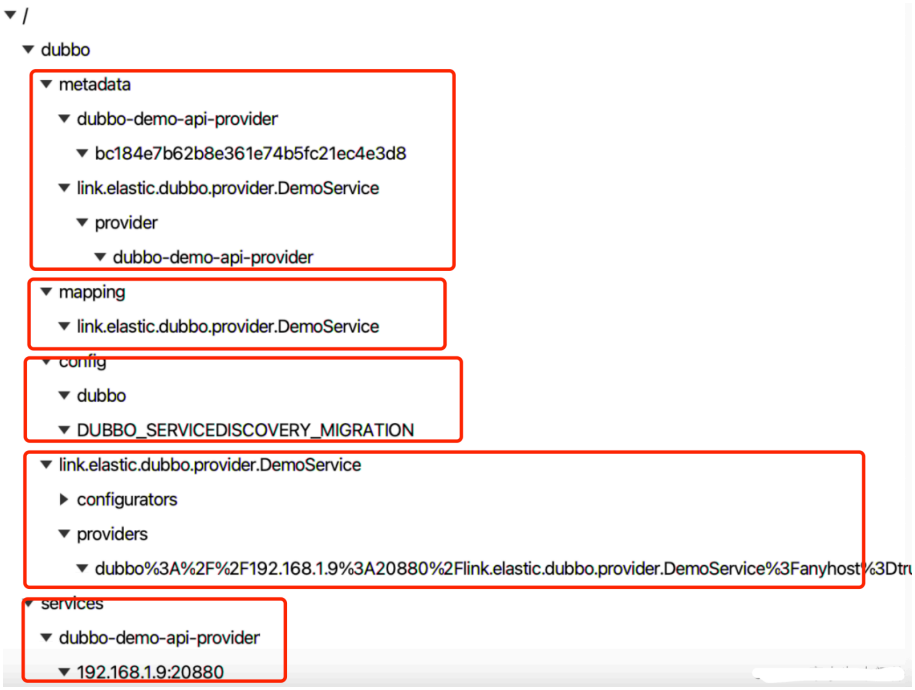
## 3. 启用服务

有了服务接口之后我们来启用服务，启用服务的源码如下：

```
import org.apache.dubbo.common.constants.CommonConstants;
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.MetadataReportConfig;
import org.apache.dubbo.config.ProtocolConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.ServiceConfig;
import org.apache.dubbo.config.bootstrap.DubboBootstrap;
import org.apache.dubbo.demo.DemoService;

public class Application {
    public static void main(String[] args) throws Exception {
            startWithBootstrap();
    }
    private static void startWithBootstrap() {
        ServiceConfig<DemoServiceImpl> service = new ServiceConfig<>();
        service.setInterface(DemoService.class);
        service.setRef(new DemoServiceImpl());
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .service(service)
            .start()
            .await();
    }
}
```

## 4. 启用服务后写入 Zookeeper 的节点数据

启动服务，这个时候我们打开 Zookeeper 图形化客户端来看看这个服务在 Zookeeper 上面写入来哪些数据，如下图：



写入 Zookeper 上的节点用于服务在分布式场景下的协调，这些节点是比较重要的。

如果了解过 Dubbo 的同学，应该会知道 Dubbo 在低版本的时候会向注册中心中写入服务接口，具体路径在上面的 dubbo 目录下，然后在/dubbo/服务接口/路径下写入如下信息：

- 服务提供者配置信息 URL 形式
- 服务消费者的配置信息 URL 形式
- 服务路由信息
- 配置信息

上面这个图就是 Dubbo3 的注册信息了，后面我们也会围绕细节来说明下，这里可以看下新增了：

- /dubbo/metadata 元数据信息
- /dubbo/mapping 服务和应用的映射信息
- /dubbo/config 注册中心配置
- /services 目录应用信息

在这里可以大致了解下，在后面会有更详细的源码解析这个示例代码.通过透析代码来看透 Dubbo3 服务注册原理，服务提供原理。

## 二、 ServiceConfig 与服务发布过程

### 1. 示例源码回顾

```
public class Application {
    public static void main(String[] args) throws Exception {
            startWithBootstrap();
    }
    private static void startWithBootstrap() {
        ServiceConfig<DemoServiceImpl> service = new ServiceConfig<>();
        service.setInterface(DemoService.class);
        service.setRef(new DemoServiceImpl());
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .service(service)
            .start()
            .await();
    }
}
```

上面这几行代码虽然看似简单,仅仅几行的启动,但是完全掌握也得下一翻大功夫,接下来我们重点看启动代码中的第一行，创建一个服务配置对象：

```
ServiceConfig<DemoServiceImpl> service = new ServiceConfig<>();
```

## 2. 了解一下服务配置的建模

下面是一个简单的 UML 继承关系图，当然这个图很是简单的，这里仅仅列出了当前服务提供者的相关服务配置继承关系，服务提供者独有的配置标注颜色为蓝色，一些可能与服务引用配置所共有的父类型我们用红色背景，当然这里为了简便起见不会提起服务引用相关的配置类型,这里列举了如下服务提供者类型,他们各司其职：

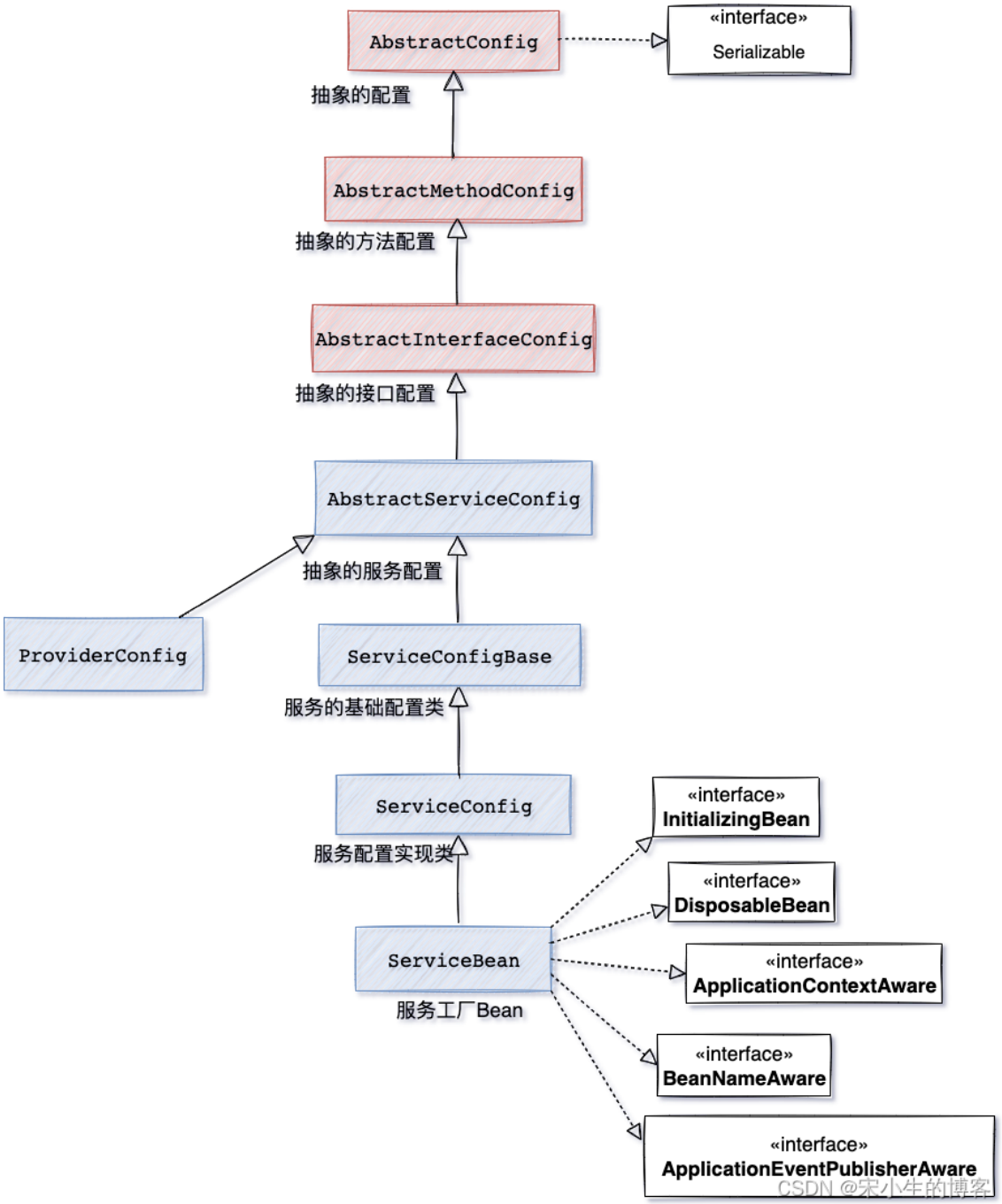



图 2.1 服务引用类继承关系 UML

## AbstractConfig

- **抽象的配置类型**，也是最顶层的服务配置类型，封装着解析配置的实用方法和公共方法，比如服务 id 的设置，服务标签名字的处理，服务参数的添加，属性的提取等等。

### AbstractMethodConfig

- **抽象的方法配置**，同样这个类型也是见名知意，服务方法的相关配置处理，这个类型主要用于对服务方法的一些配置信息建模比如服务方法的调用超时时间，重试次数，最大并发调用数，负载均衡策略，是否异步调用，是否确认异步发送等等配置信息。

### AbstractInterfaceConfig

- **抽象的接口配置**，与前面介绍的方法配置类似，这个类型是对服务接口的建模，主要的配置信息有暴漏服务的接口名字，服务接口的版本号，客户/提供方将引用的远程服务分组，服务元数据，服务接口的本地 impl 类名，服务监控配置，对于生成动态代理的策略，可以选择两种策略：jdk 和 javassist，容错类型等等配置。

### AbstractServiceConfig

- **抽象的服务配置**，这个就与我们的服务提供者有了具体的关系了，主要记录了一些服务提供者的公共配置，如服务版本，服务分组，服务延迟注册毫秒数，是否暴漏服务，服务权重，是否为动态服务，服务协议类型，是否注册等等。

### ServiceConfigBase

- **服务的基础配置类**，这个类型仍旧是个抽象的类型提取了一些基础的配置：导出服务的接口类，服务名称，接口实现的引用类型，提供者配置，是否是通用服务 GenericService。

### ServiceConfig

- **服务配置实现类**，上面的类型都是抽象类型不能做为具体存在的事物，这个类型是我们出现的第一个服务配置实现类型，服务配置实现类已经从父类型中继承了这么多的属性，这里主要为实现服务提供了一些配置如服务的协议配置，服务的代理工厂 JavassistProxyFactory 是将生成导出服务代理的 ProxyFactory

实现，是其默认实现，服务提供者模型，是否导出服务，导出的服务列表，服务监听器等等。

### ServiceBean

- **服务工厂 Bean**，这个主要是 Spring 模块来简化配置的一个服务工厂 Bean 这里就先不详细介绍 Spring 相关的配置。

## 3. ServiceConfig 构造器的初始化调用链

有了上面的类型继承关系我们就比较好分析了，接下来我们开始创建服务配置对象如下代码所示：

```
ServiceConfig<DemoServiceImpl> service = new ServiceConfig<>();
```

根据 Java 基础的构造器知识，在每个构造器的第一行都会有个 super 方法来调用父类的构造器，当前这个 super 方法我们可以不写但是 Java 编译器底层还是会为我们默认加上这么一行 super()代码来调用父类构造器的。

对于上面我提到的这几个构造器根据代码被调用的先后顺序，这里重点说几个重要的，这里我仍旧按代码执行的先后顺序来说：

### 1） 父类型 AbstractMethodConfig 构造器的初始化

根据 super 调用链这里先来看 AbstractMethodConfig 抽象方法配置

```
public AbstractMethodConfig() {
    super(ApplicationModel.defaultModel().getDefaultModule());
 }
```

在这个构造器中只有个 super 方法用来调用父类型的构造器，但是在调用之前会先使用代码 ApplicationModel.defaultModel().getDefaultModule()创建一个模块模型对象 ModuleModel

关于模型对象的细节我们会在下个章节来说，这里我们继续来看调用链。

### 2) 最顶层类型 AbstractConfig 构造器的初始化

AbstractConfig 的构造器初始化一共有两个，第一个步骤就是创建一个应用程序模型对象 ApplicationModel，刚刚我们在 AbstractMethodConfig 的构造器中了解到使用这个代码 ApplicationModel.defaultModel().getDefaultModule()创建了个模块模型对象 ModuleModel，具体他们细节我们下一章细说，了解了子类型 AbstractMethodConfig 的构造器是带参数的那我们就直接看第二个构造器。

```
public AbstractConfig() {
        this(ApplicationModel.defaultModel());
    }
```

将会调用第二个构造器初始化域模型

```
public AbstractConfig(ScopeModel scopeModel) {
        this.setScopeModel(scopeModel);
    }
```

当前类型设置 ScopeModel 类型对象

```
  public final void setScopeModel(ScopeModel scopeModel) {
     //第一次初始化的当前成员变量是空的可以设置变量
       if (this.scopeModel != scopeModel) {
         //检查参数是否合法
           checkScopeModel(scopeModel);
           //初始化对象
           ScopeModel oldScopeModel = this.scopeModel;
           this.scopeModel = scopeModel;
           // reinitialize spi extension and change referenced config's scope model
           //被子类重写的方法,根据多态会调用具体子类型的这个方法我们下面来看
           //子类应该重写此方法以初始化其SPI扩展并更改引用的配置的范围模型。
           this.postProcessAfterScopeModelChanged(oldScopeModel, this.scopeModel);
       }
   }
```

检查 ScopeModel 参数是否合法，合法的参数是不能为空并且必须是 ApplicationModel 类型或者子类型

```
   protected void checkScopeModel(ScopeModel scopeModel) {
        if (scopeModel == null) {
            throw new IllegalArgumentException("scopeModel cannot be null");
        }
        if (!(scopeModel instanceof ApplicationModel)) {
            throw new IllegalArgumentException("Invalid scope model, expect to be a
ApplicationModel but got: " + scopeModel);
        }
    }
```

a) 重写的 postProcessAfterScopeModelChanged 调用逻辑

当 ScopeModel 模型对象发生了改变，上面调用了 postProcessAfterScopeModelChanged 方法来通知模型对象改变的时候要执行的操作，根据多态重写的逻辑我们从实现类的 postProcessAfterScopeModelChanged 来看，在下面的调用链路中部分父类型并未实现 postProcessAfterScopeModelChanged 方法我们就直接忽略了。

第一个被调用到的是 ServiceConfig 类型的 postProcessAfterScopeModelChanged 方法。

```
  @Override
    protected void postProcessAfterScopeModelChanged(ScopeModel oldScopeModel, ScopeModel
newScopeModel) {
        super.postProcessAfterScopeModelChanged(oldScopeModel, newScopeModel);
        //初始化当前协议对象,通过扩展机制获取协议Protocol类型的对象
        protocolSPI = this.getExtensionLoader(Protocol.class).getAdaptiveExtension();
        //初始化当前代理工厂对象,通过扩展机制获取ProxyFactory类型的对象
        proxyFactory = this.getExtensionLoader(ProxyFactory.class).getAdaptiveExtension();
    }
```

第二个被调用到的方法为 ServiceConfigBase 的 postProcessAfterScopeModelChanged 方法。

```
@Override
    protected void postProcessAfterScopeModelChanged(ScopeModel oldScopeModel, ScopeModel
newScopeModel) {
        super.postProcessAfterScopeModelChanged(oldScopeModel, newScopeModel);
        //当服务提供者配置对象不为空时候为服务提供者对象设置域模型,这里服务提供者对象仍旧为空,这个一般用在兼
容Dubbo低版本
        if (this.provider != null && this.provider.getScopeModel() != scopeModel) {
            this.provider.setScopeModel(scopeModel);
        }
    }
```

第三个被调用到的是 AbstractInterfaceConfig 类型的 postProcessAfterScopeModelChanged 方法。

```
@Override
    protected void postProcessAfterScopeModelChanged(ScopeModel
oldScopeModel, ScopeModel newScopeModel) {
        super.postProcessAfterScopeModelChanged(oldScopeModel,
newScopeModel);
        // remove this config from old ConfigManager
//        if (oldScopeModel != null && oldScopeModel instanceof
ModuleModel) {
//
((ModuleModel)oldScopeModel).getConfigManager().removeConfig(this);
//        }

        // change referenced config's scope model
        //获取应用程序模型对象
        ApplicationModel applicationModel =
ScopeModelUtil.getApplicationModel(scopeModel);
        //为配置中心对象设置 ApplicationModel 类型对象(当前阶段配置中心配置对象为
空)
        if (this.configCenter != null &&
this.configCenter.getScopeModel() != applicationModel) {
            this.configCenter.setScopeModel(applicationModel);
        }
       //为元数据配置对象设置 ApplicationModel 类型对象(当前阶段数据配置配置对象为
空)
        if (this.metadataReportConfig != null &&
this.metadataReportConfig.getScopeModel() != applicationModel) {
            this.metadataReportConfig.setScopeModel(applicationModel);
        }
        //为 MonitorConfig 服务监控配置对象设置 ApplicationModel 类型对象(当前阶
段数据配置配置对象为空)
```

```
        if (this.monitor != null && this.monitor.getScopeModel() !=
applicationModel) {
            this.monitor.setScopeModel(applicationModel);
        }
        //这个 if 判断和上面的上面是重复的估计是写代码人加班加的太久了,没注意看
        if (this.metadataReportConfig != null &&
this.metadataReportConfig.getScopeModel() != applicationModel) {
            this.metadataReportConfig.setScopeModel(applicationModel);
        }
        //如果注册中心配置列表不为空则为每个注册中心配置设置一个 ApplicationModel
类型对象(当前注册中心对象都为空)
        if (CollectionUtils.isNotEmpty(this.registries)) {
            this.registries.forEach(registryConfig -> {
                if (registryConfig.getScopeModel() != applicationModel) {
                    registryConfig.setScopeModel(applicationModel);
                }
            });
        }
    }
```

最后被调用到的是最顶层父类型 AbstractConfig 的 postProcessAfterScopeModelChanged 方法。

这个方法什么也没干只是在父类型创建的模版方法让子类型来重写用的。

```
protected void postProcessAfterScopeModelChanged(ScopeModel oldScopeModel, ScopeModel
newScopeModel) {
        // remove this config from old ConfigManager
//        if (oldScopeModel != null && oldScopeModel instanceof ApplicationModel) {
//
((ApplicationModel)oldScopeModel).getApplicationConfigManager().removeConfig(this);
//        }
    }
```

### 3) ServiceConfigBase 构造器的初始化

```
public ServiceConfigBase() {
    //服务元数据对象创建
        serviceMetadata = new ServiceMetadata();
        //为服务元数据对象
        serviceMetadata.addAttribute("ORIGIN_CONFIG", this);
    }
```

注意，ServiceMetadata 这个类目前在 Dubbo 中没有用法。与服务级别相关的数据，例如名称、版本、业务服务的类加载器、安全信息等，还带有用于扩展的 AttributeMap。

服务配置对象的创建过程就这样结束了，当然有一些细节会放到后面来写。

上面主要顺序是按照代码执行的顺序来写的部分地方可能稍微做了调整，如果有条件的同学一定要自己进行 DEBUG 了解下细节。

关于服务配置官网提供了 xml 的配置信息这里我拷贝过来，可以做为参考：

当然这个配置不是最新的比如服务配置的标签配置 tag，warmup 预热时间单位毫秒，暂时还没有说明。

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 作用 | 描述 | 兼容性 |
|---|---|---|---|---|---|---|---|
| interface | | class | 必填 | | 服务发现 | 服务接口名 | 1.0.0 以上版本 |
| ref | | object | 必填 | | 服务发现 | 服务对象实现引用 | 1.0.0 以上版本 |
| version | version | string | 可选 | 0.0.0 | 服务发现 | 服务版本，建议使用两位数字版本，如：1.0，通常在接口不兼容时版本号才需要升级 | 1.0.0 以上版本 |
| group | group | string | 可选 | | 服务发现 | 服务分组，当一个接口有多个实现，可以用分组区分 | 1.0.7 以上版本 |
| path | <path> | string | 可选 | 缺省为接口名 | 服务发现 | 服务路径（注意：1.0 不支持自定义路径，总是使用接口名，如果有 1.0 调 2.0，配置服务路径可能不兼容） | 1.0.12 以上版本 |
| delay | delay | int | 可选 | 0 | 性能调优 | 延迟注册服务时间（毫秒），设为-1 时，表示延迟到 Spring 容器初始化完成时暴露服务 | 1.0.14 以上版本 |
| timeout | timeout | int | 可选 | 1000 | 性能调优 | 远程服务调用超时时间（毫秒） | 2.0.0 以上版本 |
| retries | retries | int | 可选 | 2 | 性能调优 | 远程服务调用重试次数，不包括第一次调用，不需要重试请设为 0 | 2.0.0 以上版本 |
| connections | connections | int | 可选 | 100 | 性能调优 | 对每个提供者的最大连接数，rmi、http、hessian 等短连接协议表示限制连接数，dubbo 等长连接协表示建立的长连接个数 | 2.0.0 以上版本 |
| loadbalance | loadbalance | string | 可选 | random | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮询，最少活跃调用 | 2.0.0 以上版本 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| async | async | boolean | 可选 | false | 性能调优 | 是否缺省异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程 | 2.0.0 以上版本 |
| local | local | class/boolean | 可选 | false | 服务治理 | 设为 true，表示使用缺省代理类名，即：接口名 +Local 后缀，已废弃，请使用 stub | 2.0.0 以上版本 |
| stub | stub | class/boolean | 可选 | false | 服务治理 | 设为 true，表示使用缺省代理类名，即：接口名 + Stub 后缀，服务接口客户端本地代理类名，用于在客户端执行本地逻辑，如本地缓存等，该本地代理类的构造函数必须允许传入远程代理对象，构造函数如： public XxxServiceStub（XxxService xxxService） | 2.0.0 以上版本 |
| mock | mock | class/boolean | 可选 | false | 服务治理 | 设为 true，表示使用缺省 Mock 类名，即：接口名 +Mock 后缀，服务接口调用失败 Mock 实现类，该 Mock 类必须有一个无参构造函数，与 Local 的区别在于，Local 总是被执行，而 Mock 只在出现非业务异常（比如超时，网络异常等）时执行，Local 在远程调用之前执行，Mock 在远程调用后执行。 | 2.0.0 以上版本 |
| token | token | string/boolean | 可选 | false | 服务治理 | 令牌验证，为空表示不开启，如果为 true，表示随机生成动态令牌，否则使用静态令牌，令牌的作用是防止消费者绕过注册中心直接访问，保证注册中心的授权功能有效，如果使用点对点调用，需关闭令牌功能 | 2.0.0 以上版本 |
| registry | | string | 可选 | 缺省向所有 registry 注册 | 配置关联 | 向指定注册中心注册，在多个注册中心时使用，值为 dubbo:registry 的 id 属性，多个注册中心 ID 用逗号分隔，如果不想将该服务注册到任何 registry，可将值设为 N/A | 2.0.0 以上版本 |
| provider | | string | 可选 | 缺省使用第一个 provider 配置 | 配置关联 | 指定 provider，值为 dubbo:provider 的 id 属性 | 2.0.0 以上版本 |
| deprecated | deprecated | boolean | 可选 | false | 服务治理 | 服务是否过时，如果设为 true，消费方引用时将打印服务过时警告 error 日志 | 2.0.5 以上版本 |
| dynamic | dynamic | boolean | 可选 | true | 服务治理 | 服务是否动态注册，如果设为 false，注册后将显示后 disable 状态，需人工启用，并且服务提供者停止时，也不会自动取消册，需人工禁用。 2.0.5 以上版本 | |
| accesslog | accesslog | string/boolean | 可选 | false | 服务治理 | 设为 true，将向 logger 中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件 | 2.0.5 以上版本 |
| owner | owner | string | 可选 | | 服务治理 | 服务负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5 以上版本 |
| document | document | string | 可选 | | 服务治理 | 服务文档 URL | 2.0.5 以上版本 |
| weight | weight | int | 可选 | | 性能调优 | 服务权重 | 2.0.5 以上版本 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| executes | executes | int | 可选 | 0 | 性能调优 | 服务提供者每服务每方法最大可并行执行请求数 | 2.0.5 以上版本 |
| proxy | proxy | string | 可选 | javassist | 性能调优 | 生成动态代理方式，可选：jdk/javassist | 2.0.5 以上版本 |
| cluster | cluster | string | 可选 | failover | 性能调优 | 集群方式，可选：failover/failfast/failsafe/failback/forking | 2.0.5 以上版本 |
| filter | service.filter | string | 可选 | default | 性能调优 | 服务提供方远程调用过程拦截器名称，多个名称用逗号分隔 | 2.0.5 以上版本 |
| listener | exporter.listener | string | 可选 | default | 性能调优 | 服务提供方导出服务监听器名称，多个名称用逗号分隔 | |
| protocol | | string | 可选 | | 配置关联 | 使用指定的协议暴露服务，在多协议时使用，值为 dubbo:protocol 的 id 属性，多个协议 ID 用逗号分隔 | 2.0.5 以上版本 |
| layer | layer | string | 可选 | 服务治理 | 服务提供者所在的分层。如：biz、dao、intl:web、china:acton。 | 2.0.7 以上版本 | |
| register | register | boolean | 可选 | true | 服务治理 | 该协议的服务是否注册到注册中心 | 2.0.8 以上版本 |

# 模块化与多实例

## 一、 框架、应用、模块领域模型及 Model 对象的初始化

在上一章中我们详细看了服务配置 ServiceConfig 类型的初始化，不过我们跳过了 AbstractMethodConfig 的构造器中创建模块模型对象的过程，那这一章我们就来看下模块模型对象的初始化过程：

```
public AbstractMethodConfig() {
        super(ApplicationModel.defaultModel().getDefaultModule());
    }
```

那为什么会在 Dubbo3 的新版本中加入这个域模型呢，主要有如下原因。之前 dubbo 都是只有一个作用域的，通过静态类属性共享增加域模型是为了：

- 让 Dubbo 支持多应用的部署，这块一些大企业有诉求。
- 从架构设计上，解决静态属性资源共享、清理的问题。
- 分层模型将应用的管理和服务的管理分开。

可能比较抽象，可以具体点来看。Dubbo3 中在启动时候需要启动配置中心、元数据中心，这个配置中心和元数据中心可以归应用模型来管理。Dubbo 作为 RPC 框架又需要启动服务和引用服务，服务级别的管理就交给了这个模块模型来管理。分层次的管理方便我们理解和处理逻辑，父子级别的模型又方便了数据传递。

了解过 JVM 类加载机制的同学应该就比较清楚 JVM 类加载过程中的数据访问模型。子类加载器先交给父类加载器查找，找不到再从子类加载器中查找。Dubbo 的分层模型类似这样一种机制，这一章先来简单了解下，后面用到时候具体细说。

### 1. 模型对象的关系

为了不增加复杂性，我们这里仅仅列出模型对象类型类型之间的继承关系如下所示：

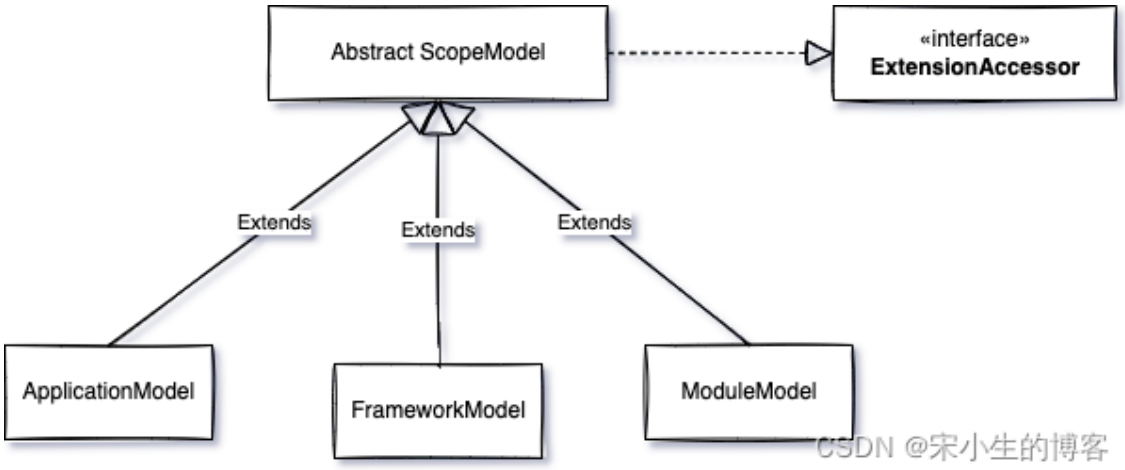



图 3.1 模型对象的继承关系

模型对象一共有 4 个，公共的属性和操作放在了域模型类型中，下面我们来详细说下这几个模型类型：

- ExtensionAccessor 扩展的统一访问器
- 用于获取扩展加载管理器 ExtensionDirector 对象
- 获取扩展对象 ExtensionLoader
- 根据扩展名字获取具体扩展对象
- 获取自适应扩展对象
- 获取默认扩展对象
- ScopeModel 模型对象的公共抽象父类型
- 内部 id 用于表示模型树的层次结构
- 公共模型名称，可以被用户设置
- 描述信息
- 类加载器管理
- 父模型管理 parent
- 当前模型的所属域 ExtensionScope 有：FRAMEWORK（框架），APPLICATION（应用），MODULE（模块），SELF（自给自足，为每个作用域创建一个实例，用于特殊的 SPI 扩展，如 ExtensionInjector）
- 具体的扩展加载程序管理器对象的管理：ExtensionDirector
- 域 Bean 工厂管理，一个内部共享的 Bean 工厂 ScopeBeanFactory
- 等等

- FrameworkModel dubbo 框架模型，可与多个应用程序共享
- FrameworkModel 实例对象集合，allInstances
- 所有 ApplicationModel 实例对象集合，applicationModels
- 发布的 ApplicationModel 实例对象集合 pubApplicationModels
- 框架的服务存储库 FrameworkServiceRepository 类型对象（数据存储在内存中）
- 内部的应用程序模型对象 internalApplicationModel
- ApplicationModel 表示正在使用 Dubbo 的应用程序，并存储基本元数据信息，以便在 RPC 调用过程中使用

  ApplicationModel 包括许多关于发布服务的 ProviderModel 和许多关于订阅服务的 Consumer Model。

  - ExtensionLoader、DubboBootstrap 和这个类目前被设计为单例或静态（本身完全静态或使用一些静态字段）。因此，从它们返回的实例属于流程范围。如果想在一个进程中支持多个 dubbo 服务器，可能需要重构这三个类。
  - 所有 ModuleModel 实例对象集合 moduleModels
  - 发布的 ModuleModel 实例对象集合 pubModuleModels
  - 环境信息 Environment 实例对象 environment
  - 配置管理 ConfigManager 实例对象 configManager
  - 服务存储库 ServiceRepository 实例对象 serviceRepository
  - 应用程序部署器 ApplicationDeployer 实例对象 deployer
  - 所属框架 FrameworkModel 实例对象 frameworkModel
  - 内部的模块模型 ModuleModel 实例对象 internalModule
  - 默认的模块模型 ModuleModel 实例对象 defaultModule

- ModuleModel 服务模块的模型

  - 所属应用程序模型 ApplicationModel 实例对象 applicationModel
  - 模块环境信息 ModuleEnvironment 实例对象 moduleEnvironment
  - 模块服务存储库 ModuleServiceRepository 实例对象 serviceRepository
  - 模块的服务配置管理 ModuleConfigManager 实例对象 moduleConfigManager
  - 模块部署器 ModuleDeployer 实例对象 deployer 用于导出和引用服务

了解了这几个模型对象的关系我们可以了解到这几个模型对象的管理层级从框架到应用程序，然后到模块的管理（FrameworkModel->ApplicationModel->ModuleModel），他们主要用来针对框架，应用程序，模块的存储，发布管理，配置管理。

看来 Dubbo3 针对应用服务治理与运维这一块也是在努力尝试。

### 1) AbstractMethodConfig 配置对象中获取模型对象的调用

模块模型（ModuleModel）参数对象的创建。

这个 AbstractMethodConfig 构造器在初始化的时候调调用了这么一行代码做为参数向父类型传递对象。

```
ApplicationModel.defaultModel().getDefaultModule()
```

默认情况下使用 ApplicationModel 的静态方法获取默认的模型对象和默认的模块对象。

ApplicationModel（应用程序领域模型）类型中获取默认模型对象的方法：

```
public static ApplicationModel defaultModel() {
       // should get from default FrameworkModel, avoid out of sync
       return FrameworkModel.defaultModel().defaultApplication();
   }
```

这里可以看到要想获取应用程序模型必须先通过框架领域模型来获取层级也是框架领域模型到应用程序领域模型。

### 2) 使用双重校验锁获取框架模型对象

FrameworkModel（框架模型）的默认模型获取工厂方法 defaultModel()

```
/**
* 在源码的注释上有这么一句话:在销毁默认的 FrameworkModel 时， FrameworkModel.defaultModel()
*或ApplicationModel.defaultModel() 将返回一个损坏的模型
*可能会导致不可预知的问题。建议：尽量避免使用默认模型。
*/
 public static FrameworkModel defaultModel() {
     //双重校验锁的形式创建单例对象
      FrameworkModel instance = defaultInstance;
      if (instance == null) {
          synchronized (globalLock) {
            //重置默认框架模型
              resetDefaultFrameworkModel();
              if (defaultInstance == null) {
                  defaultInstance = new FrameworkModel();
              }
              instance = defaultInstance;
          }
      }
      Assert.notNull(instance, "Default FrameworkModel is null");
      return instance;
  }
```

### 3) 刷新重置默认框架模型对象

FrameworkModel 中的重置默认框架模型 resetDefaultFrameworkModel

```
  private static void resetDefaultFrameworkModel() {
      //全局悲观锁，同一个时刻只能有一个线程执行重置操作
      synchronized (globalLock) {
        //defaultInstance为当前成员变量FrameworkModel类型代表当前默认的FrameworkModel类型的实例对
象
          if (defaultInstance != null && !defaultInstance.isDestroyed()) {
              return;
          }
          FrameworkModel oldDefaultFrameworkModel = defaultInstance;
          //存在实例模型列表则直接从内存缓存中查后续不需要创建了
          if (allInstances.size() > 0) {
            //当前存在的有FrameworkModel框架实例多个列表则取第一个为默认的
              defaultInstance = allInstances.get(0);
          } else {
              defaultInstance = null;
          }
          if (oldDefaultFrameworkModel != defaultInstance) {
              if (LOGGER.isInfoEnabled()) {
                  LOGGER.info("Reset global default framework from " +
safeGetModelDesc(oldDefaultFrameworkModel) + " to " + safeGetModelDesc(defaultInstance));
              }
          }
      }
  }
```

上面单例做了很多的初始化操作，这里开始调用构造器来创建框架模型对象，如下代码：

## 2. 创建 FrameworkModel 对象

FrameworkModel()构造器

```
public FrameworkModel() {
    //调用父类型ScopeModel传递参数，这个构造器的第一个参数为空代表这是一个顶层的域模型，第二个代表了这个是框架FRAMEWORK域，第三个false不是内部域
        super(null, ExtensionScope.FRAMEWORK, false);
        //内部id用于表示模型树的层次结构，如层次结构：
        //FrameworkModel（索引=1）->ApplicationModel（索引=2）->ModuleModel（索引=1，第一个用户模块）
        //这个index变量是static类型的为静态全局变量默认值从1开始，如果有多个框架模型对象则internalId编号从1开始依次递增
        this.setInternalId(String.valueOf(index.getAndIncrement()));
        // register FrameworkModel instance early
        //将当前新创建的框架实例对象添加到容器中
        synchronized (globalLock) {
          //将当前框架模型实例添加到所有框架模型缓存对象中
            allInstances.add(this);
            //如上面代码所示重置默认的框架模型对象，这里将会是缓存实例列表的第一个，新增了一个刷新默认实例对象
            resetDefaultFrameworkModel();
        }
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info(getDesc() + " is created");
        }
        //初始化框架模型领域对象
        initialize();
    }
```

ExtensionScope.FRAMEWORK

### 1) 初始化 FrameworkModel

FrameworkModel 框架模型的初始化方法 initialize()

```
@Override
    protected void initialize() {
      //这里初始化之前先调用下父类型ScopeModel的初始化方法我们在下面来看
        super.initialize();
    //使用TypeDefinitionBuilder的静态方法initBuilders来初始化类型构建器TypeBuilder类型集合
        TypeDefinitionBuilder.initBuilders(this);
    //框架服务存储仓库对象，可以用于快速查询服务提供者信息
        serviceRepository = new FrameworkServiceRepository(this);
    //获取ScopeModelInitializer类型(域模型初始化器)的扩展加载器ExtensionLoader，每个扩展类型都会创建一个扩展加载器缓存起来
        ExtensionLoader<ScopeModelInitializer> initializerExtensionLoader = this.getExtensionLoader(ScopeModelInitializer.class);
        //获取ScopeModelInitializer类型的支持的扩展集合，这里当前版本存在8个扩展类型实现
        Set<ScopeModelInitializer> initializers = initializerExtensionLoader.getSupportedExtensionInstances();
        //遍历这些扩展实现调用他们的initializeFrameworkModel方法来传递FrameworkModel类型对象，细节我们待会再详细说下
        for (ScopeModelInitializer initializer : initializers) {
            initializer.initializeFrameworkModel(this);
        }
    //创建一个内部的ApplicationModel类型，细节下面说
        internalApplicationModel = new ApplicationModel(this, true);
        //创建ApplicationConfig类型对象同时传递应用程序模型对象internalApplicationModel
        //获取ConfigManager类型对象，然后设置添加当前应用配置对象
       internalApplicationModel.getApplicationConfigManager().setApplication(
            new ApplicationConfig(internalApplicationModel, CommonConstants.DUBBO_INTERNAL_APPLICATION));
            //设置公开的模块名字为常量DUBBO_INTERNAL_APPLICATION
        internalApplicationModel.setModelName(CommonConstants.DUBBO_INTERNAL_APPLICATION);
    }
```

继续上面代码的调用链路，我们来看 FrameworkModel 的 super.initialize(); 方法调用父类型 ScopeModel 的 initialize() 方法。

### 2) 初始化 ScopeModel

ScopeModel 类型的初始化方法 initialize()：

```
protected void initialize() {
    //初始化ExtensionDirector是一个作用域扩展加载程序管理器。
    //ExtensionDirector支持多个级别，子级可以继承父级的扩展实例。
    //查找和创建扩展实例的方法类似于Java classloader。
        this.extensionDirector = new ExtensionDirector(parent != null ?
parent.getExtensionDirector() : null, scope, this);
        //这个参考了Spring的生命周期回调思想，添加一个扩展初始化的前后调用的处理器，在扩展初始化之前或之后
调用的后处理器，参数类型为ExtensionPostProcessor
        this.extensionDirector.addExtensionPostProcessor(new
ScopeModelAwareExtensionProcessor(this));
        //创建一个内部共享的域工厂对象，用于注册Bean，创建Bean，获取Bean，初始化Bean等
        this.beanFactory = new ScopeBeanFactory(parent != null ? parent.getBeanFactory() :
null, extensionDirector);
        //使用数据结构链表，创建销毁监听器容器，一般用于关闭进程，重置应用程序对象等操作时候调用
        this.destroyListeners = new LinkedList<>();
        //使用ConcurrentHashMap属性集合
        this.attributes = new ConcurrentHashMap<>();
        //使用ConcurrentHashSet存储当前域下的类加载器
        this.classLoaders = new ConcurrentHashSet<>();

        // Add Framework's ClassLoader by default
        //将当前类的加载器存入加载器集合classLoaders中
        ClassLoader dubboClassLoader = ScopeModel.class.getClassLoader();
        if (dubboClassLoader != null) {
            this.addClassLoader(dubboClassLoader);
        }
    }
```

### 3) 初始类型定义构建器

TypeDefinitionBuilder 的初始化类型构造器方法 initBuilders

```
public static void initBuilders(FrameworkModel model) {
        Set<TypeBuilder> tbs =
model.getExtensionLoader(TypeBuilder.class).getSupportedExtensionInstances();
        BUILDERS = new ArrayList<>(tbs);
    }
```

#### a) 服务存储仓库对象的创建

FrameworkServiceRepository 对象的初始化

```
public FrameworkServiceRepository(FrameworkModel frameworkModel) {
        this.frameworkModel = frameworkModel;
    }
```

### 4) 域模型初始化器的获取与初始化回调

域模型初始化器的获取与初始化（ScopeModelInitializer 类型和 initializeFrameworkModel 方法）。

加载到的 ScopeModelInitializer 类型的 SPI 扩展实现。

```
ExtensionLoader<ScopeModelInitializer> initializerExtensionLoader =
this.getExtensionLoader(ScopeModelInitializer.class);
    //获取ScopeModelInitializer类型的支持的扩展集合，这里当前版本存在8个扩展类型实现
    Set<ScopeModelInitializer> initializers =
initializerExtensionLoader.getSupportedExtensionInstances();
    //遍历这些扩展实现调用他们的initializeFrameworkModel方法来传递FrameworkModel类型对象，细节我们待会
再详细说下
    for (ScopeModelInitializer initializer : initializers) {
        initializer.initializeFrameworkModel(this);
    }
```

通过 Debug 查到域模型初始化器的 SPI 扩展类型有如下 8 个：


这里我随机找两个说一下吧。容错域模型初始化器:ClusterScopeModelInitializer 的 initializeFrameworkModel 方法。

```
public class ClusterScopeModelInitializer implements ScopeModelInitializer {
    @Override
    public void initializeFrameworkModel(FrameworkModel frameworkModel) {
        ScopeBeanFactory beanFactory = frameworkModel.getBeanFactory();
        beanFactory.registerBean(RouterSnapshotSwitcher.class);
    }
```

```
public class CommonScopeModelInitializer implements ScopeModelInitializer {
    @Override
    public void initializeFrameworkModel(FrameworkModel frameworkModel) {
        ScopeBeanFactory beanFactory = frameworkModel.getBeanFactory();
        beanFactory.registerBean(FrameworkExecutorRepository.class);
    }
```

```
public class ConfigScopeModelInitializer implements ScopeModelInitializer {

    @Override
    public void initializeFrameworkModel(FrameworkModel frameworkModel) {
        frameworkModel.addDestroyListener(new FrameworkModelCleaner());
    }
```

### 5) 将内部应用配置对象创建与添加至应用模型中

创建 ApplicationConfig 对象让后将其添加至应用模型中。

内部应用程序模型，这里为应用配置管理器设置一个应用配置对象，将这个应用配置的模块名字配置名字设置为 DUBBO_INTERNAL_APPLICATION，应用配置记录着我们常见的应用配置信息，如下面表格所示：

```
 //获取ConfigManager类型对象，然后设置添加当前应用配置对象
   internalApplicationModel.getApplicationConfigManager().setApplication(
        new ApplicationConfig(internalApplicationModel,
CommonConstants.DUBBO_INTERNAL_APPLICATION));
        //设置公开的模块名字为常量DUBBO_INTERNAL_APPLICATION
    internalApplicationModel.setModelName(CommonConstants.DUBBO_INTERNAL_APPLICATION);
```

来自官网目前版本的配置解释。官网当前的配置描述知道到了元数据类型，后面我再补充几个。

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 作用 | 描述 | 兼容性 |
|---|---|---|---|---|---|---|---|
| name | application | string | 必填 | | 服务治理 | 当前应用名称，用于注册中心计算应用间依赖关系，注意：消费者和提供者应用名不要一样，此参数不是匹配条件，你当前项目叫什么名字就填什么，和提供者消费者角色无关，比如：kylin 应用调用了 morgan 应用的服务，则 kylin 项目配成 kylin，morgan 项目配成 | 1.0.16 以上版本 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | morgan，可能 kylin 也提供其它服务给别人使用，但 kylin 项目永远配成 kylin，这样注册中心将显示 kylin 依赖于 morgan | |
| version | application.version | string | 可选 | | 服务治理 | 当前应用的版本 | 2.2.0 以上版本 |
| owner | owner | string | 可选 | | 服务治理 | 应用负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5 以上版本 |
| organization | organization | string | 可选 | | 服务治理 | 组织名称（BU 或部门），用于注册中心区分服务来源，此配置项建议不要使用 autoconfig，直接写死在配置中，比如 china,intl,itu,crm,asc,dw,aliexpress 等 | 2.0.0 以上版本 |
| architecture | architecture | string | 可选 | | 服务治理 | 用于服务分层对应的架构。如，intl、china。不同的架构使用不同的分层。 | 2.0.7 以上版本 |
| environment | environment | string | 可选 | | 服务治理 | 应用环境，如：develop/test/product，不同环境使用不同的缺省值，以及作为只用于开发测试功能的限制条件 | 2.0.0 以上版本 |
| compiler | compiler | string | 可选 | javassist | 性能优化 | Java 字节码编译器，用于动态类的生成，可选：jdk 或 javassist | 2.1.0 以上版本 |
| logger | logger | string | 可选 | slf4j | 性能优化 | 日志输出方式，可选：slf4j,jcl,log4j,log4j2,jdk | 2.2.0 以上版本 |
| metadata-type | metadata-type | String | 可选 | local | 服务治理 | metadata 传递方式，是以 Provider 视角而言的，Consumer 侧配置无效，可选值有：remote - Provider 把 metadata 放到远端注册中心，Consumer 从注册中心获取 local - Provider 把 metadata 放在本地，Consumer 从 Provider 处直接获取 | 2.7.6 以上版本 |

当前在 Dubbo3.0.7 中还有一些的配置我下面列举下：

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 作用 | 描述 | 兼容性 |
|---|---|---|---|---|---|---|---|
| register-consumer | register-consumer | boolean | 可选 | false | 服务治理 | 是否注册使用者实例，默认为 false。 | |
| register-mode | register-mode | string | 可选 | all | 服务治理 | 将 interface/instance/all 地址注册到注册中心，默认为 all。 | |
| enable-empty-protection | enable-empty-protection | boolean | 可选 | true | 服务治理 | 在空地址通知上启用空保护，默认为 true | |
| protocol | protocol | string | 可选 | dubbo | 服务治理 | 此应用程序的首选协议（名称） | |

## 3. 创建 ApplicationModel 对象

ApplicationModel 对象的初始化调用在前面 3.2.4 FrameworkModel 框架模型的初始化方法 initialize()章节中，我们看到了代码 ApplicationModel 对象的初始化调用如下代码，这里我们来详细说一下。

```
 internalApplicationModel = new ApplicationModel(this, true);
        internalApplicationModel.getApplicationConfigManager().setApplication(
            new ApplicationConfig(internalApplicationModel,
CommonConstants.DUBBO_INTERNAL_APPLICATION));
        internalApplicationModel.setModelName(CommonConstants.DUBBO_INTERNAL_APPLICATION);
```

### 1） ApplicationModel 的构造器

ApplicationModel（FrameworkModel frameworkModel，boolean isInternal）刚刚 3.2.9 那个地方我们看到了使用代码 new ApplicationModel（this，true）来创建对象这里我们详细看下代码细节：

```
public ApplicationModel(FrameworkModel frameworkModel, boolean isInternal) {
    //调用父类型ScopeModel传递参数，这个构造器的传递没与前面看到的FrameworkModel构造器的中的调用参数有些不同第一个参数我们为frameworkModel代表父域模型，第二个参数标记域为应用程序级别APPLICATION，第三个参数我们传递的为true代表为内部域
        super(frameworkModel, ExtensionScope.APPLICATION, isInternal);
        Assert.notNull(frameworkModel, "FrameworkModel can not be null");
        //应用程序域成员变量记录frameworkModel对象
        this.frameworkModel = frameworkModel;
        //frameworkModel对象添加当前应用程序域对象
        frameworkModel.addApplication(this);
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info(getDesc() + " is created");
        }
        //初始化应用程序
        initialize();
    }
```

a） 将 ApplicationModel 添加至 FrameworkModel 容器中

FrameworkModel 的添加应用程序方法 addApplication:

```
void addApplication(ApplicationModel applicationModel) {
        // can not add new application if it's destroying
        //检查FrameworkModel对象是否已经被标记为销毁状态，如果已经被销毁了则抛出异常无需执行逻辑
        checkDestroyed();
        synchronized (instLock) {
          //如果还未添加过当前参数传递应用模型
            if (!this.applicationModels.contains(applicationModel)) {
              //为当前应用模型生成内部id
                applicationModel.setInternalId(buildInternalId(getInternalId(),
appIndex.getAndIncrement()));
                //添加到成员变量集合applicationModels中
                this.applicationModels.add(applicationModel);
                //如果非内部的则也向公开应用模型集合pubApplicationModels中添加一下
                if (!applicationModel.isInternal()) {
                    this.pubApplicationModels.add(applicationModel);
                }
                resetDefaultAppModel();
            }
        }
    }
```

内部 id 生成算法 buildInternalId 方法代码如下。

看代码胜过，文字解释

```
 protected String buildInternalId(String parentInternalId, long childIndex) {
        // FrameworkModel    1
        // ApplicationModel  1.1
        // ModuleModel       1.1.1
        if (StringUtils.hasText(parentInternalId)) {
            return parentInternalId + "." + childIndex;
        } else {
            return "" + childIndex;
        }
    }
```

**重置默认的应用模型对象**

FrameworkModel 重置默认的应用模型对象 resetDefaultAppModel()方法与默认框架模型设置方式类似取集合的第一个，这里应用模型需要使用公开的应用模型的第一个做为默认应用模型。

代码如下所示：

```
private void resetDefaultAppModel() {
        synchronized (instLock) {
            if (this.defaultAppModel != null && !this.defaultAppModel.isDestroyed()) {
                return;
            }
            //取第一个公开的应用模型做为默认应用模型
            ApplicationModel oldDefaultAppModel = this.defaultAppModel;
            if (pubApplicationModels.size() > 0) {
                this.defaultAppModel = pubApplicationModels.get(0);
            } else {
                this.defaultAppModel = null;
            }
            if (defaultInstance == this && oldDefaultAppModel != this.defaultAppModel) {
                if (LOGGER.isInfoEnabled()) {
                    LOGGER.info("Reset global default application from " +
safeGetModelDesc(oldDefaultAppModel) + " to " + safeGetModelDesc(this.defaultAppModel));
                }
            }
        }
    }
```

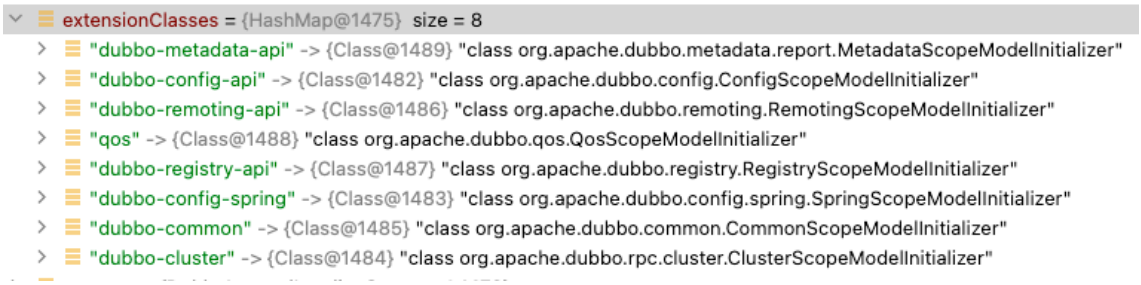
## 2) 初始化 ApplicationModel

ApplicationModel 的初始化 initialize()方法。在前面 3.2.10 ApplicationModel 的构造器 ApplicationModel(FrameworkModel frameworkModel, boolean isInternal)中的最后一行开始初始化应用模型我们还未详细说明下面可以来看下。

```
@Override
    protected void initialize() {
    //这个是调用域模型来初始化基础信息如扩展访问器等，可以参考 3.2.5 ScopeModel类型的初始化方法
initialize()章节
        super.initialize();
        //创建一个内部的模块模型对象
        internalModule = new ModuleModel(this, true);
        //创建一个独立服务存储对象
        this.serviceRepository = new ServiceRepository(this);
    //获取应用程序初始化监听器ApplicationInitListener扩展
        ExtensionLoader<ApplicationInitListener> extensionLoader =
this.getExtensionLoader(ApplicationInitListener.class);
        //如果存在应用程序初始化监听器扩展则执行这个初始化方法，在当前的版本还未看到有具体的扩展实现类型
        Set<String> listenerNames = extensionLoader.getSupportedExtensions();
        for (String listenerName : listenerNames) {
            extensionLoader.getExtension(listenerName).init();
        }
    //初始化扩展(这个是应用程序生命周期的方法调用，这里调用初始化方法
        initApplicationExts();

    //获取域模型初始化器扩展对象列表，然后执行初始化方法
        ExtensionLoader<ScopeModelInitializer> initializerExtensionLoader =
this.getExtensionLoader(ScopeModelInitializer.class);
        Set<ScopeModelInitializer> initializers =
initializerExtensionLoader.getSupportedExtensionInstances();
        for (ScopeModelInitializer initializer : initializers) {
            initializer.initializeApplicationModel(this);
        }
    }
```

### 3) initApplicationExts()初始化应用程序扩展方法

```
    private void initApplicationExts() {
    //这个扩展实现一共有两个可以看下面那个图扩展类型为ConfigManager和Environment
         Set<ApplicationExt> exts =
this.getExtensionLoader(ApplicationExt.class).getSupportedExtensionInstances();
         for (ApplicationExt ext : exts) {
             ext.initialize();
         }
     }
```




### a) ConfigManager 类型的 initialize 方法

先简单说下 ConfigManager 的作用，无锁配置管理器(通过 ConcurrentHashMap)，用于快速读取操作。写入操作锁带有配置类型的子配置映射，用于安全检查和添加新配置。

其实 ConfigManager 实现类中并没有这个初始化方法 initialize，不过 ConfigManager 的父类型 AbstractConfigManager 中是有 initialize 方法的，如下所示。

AbstractConfigManager 的初始化方法 initialize

```
@Override
    public void initialize() throws IllegalStateException {
         //乐观锁判断是否初始化过
        if (!initialized.compareAndSet(false, true)) {
            return;
        }
        //从模块环境中获取组合配置，目前 Environment 中有 6 种重要的配置，我们后面详细说
```

```
        CompositeConfiguration configuration =
scopeModel.getModelEnvironment().getConfiguration();

        // dubbo.config.mode 获取配置模式，配置模式对应枚举类型 ConfigMode，目前
有这么几个 STRICT，OVERRIDE，OVERRIDE_ALL，OVERRIDE_IF_ABSENT，IGNORE，这个
配置决定了属性覆盖的顺序，当有同一个配置 key 多次出现时候，以最新配置为准，还是以最老
的那个配置为准，还是配置重复则抛出异常，默认值为严格模式 STRICT 重复则抛出异常
        String configModeStr = (String)
configuration.getProperty(ConfigKeys.DUBBO_CONFIG_MODE);
        try {
            if (StringUtils.hasText(configModeStr)) {
                this.configMode =
ConfigMode.valueOf(configModeStr.toUpperCase());
            }
        } catch (Exception e) {
            String msg = "Illegal '" + ConfigKeys.DUBBO_CONFIG_MODE + "'
config value [" + configModeStr + "], available values " +
Arrays.toString(ConfigMode.values());
            logger.error(msg, e);
            throw new IllegalArgumentException(msg, e);
        }

        // dubbo.config.ignore-duplicated-interface
        //忽略重复的接口（服务/引用）配置。默认值为 false
        String ignoreDuplicatedInterfaceStr = (String) configuration
            .getProperty(ConfigKeys.DUBBO_CONFIG_IGNORE_DUPLICATED_INTER
FACE);
        if (ignoreDuplicatedInterfaceStr != null) {
            this.ignoreDuplicatedInterface =
Boolean.parseBoolean(ignoreDuplicatedInterfaceStr);
        }

        // print 打印配置信息
        Map<String, Object> map = new LinkedHashMap<>();
        map.put(ConfigKeys.DUBBO_CONFIG_MODE, configMode);
        map.put(ConfigKeys.DUBBO_CONFIG_IGNORE_DUPLICATED_INTERFACE,
this.ignoreDuplicatedInterface);
        logger.info("Config settings: " + map);
    }
```

### b) Environment 类型的 initialize 方法

这是一个与环境配置有关系的类型，我们先来简单了解下它的初始化方法，后期再详细说明：

Environment 类型的 initialize 方法

```
@Override
    public void initialize() throws IllegalStateException {
        //乐观锁判断是否进行过初始化
        if (initialized.compareAndSet(false, true)) {
         //PropertiesConfiguration 从系统属性和 dubbo.properties 中获取配置
            this.propertiesConfiguration = new
PropertiesConfiguration(scopeModel);
            //SystemConfiguration 获取的是 JVM 参数 启动命令中-D 指定的
            this.systemConfiguration = new SystemConfiguration();
            //EnvironmentConfiguration 是从环境变量中获取的配置
            this.environmentConfiguration = new
EnvironmentConfiguration();
            //外部的 Global 配置 config-center global/default config
            this.externalConfiguration = new
InmemoryConfiguration("ExternalConfig");
            //外部的应用配置如:config-center 中的应用配置
            this.appExternalConfiguration = new
InmemoryConfiguration("AppExternalConfig");
            //本地应用配置 ， 如 Spring
Environment/PropertySources/application.properties
            this.appConfiguration = new
InmemoryConfiguration("AppConfig");
                          //服务迁移配置加载 dubbo2 升级 dubbo3 的一些配置
            loadMigrationRule();
        }
    }
         //服务迁移配置加载 JVM  > env >  代码路径 dubbo-migration.yaml
    private void loadMigrationRule() {
     //文件路径配置的 key dubbo.migration.file
     // JVM 参数中获取
        String path =
System.getProperty(CommonConstants.DUBBO_MIGRATION_KEY);
        if (StringUtils.isEmpty(path)) {
        //env 环境变量中获取
            path = System.getenv(CommonConstants.DUBBO_MIGRATION_KEY);
```

```
            if (StringUtils.isEmpty(path)) {
            //类路径下获取文件 dubbo-migration.yaml
                path = CommonConstants.DEFAULT_DUBBO_MIGRATION_FILE;
            }
        }
        this.localMigrationRule =
ConfigUtils.loadMigrationRule(scopeModel.getClassLoaders(), path);
    }
```

ConfigUtils 中读取迁移规则配置文件 loadMigrationRule

这个我们不细说了，贴一下代码感兴趣可以了解下，这个代码主要是读取文件到内存字符串：

```
public static String loadMigrationRule(Set<ClassLoader> classLoaders,
String fileName) {
        String rawRule = "";
        if (checkFileNameExist(fileName)) {
            try {
                try (FileInputStream input = new
FileInputStream(fileName)) {
                    return readString(input);
                }
            } catch (Throwable e) {
                logger.warn("Failed to load " + fileName + " file from "
+ fileName + "(ignore this file): " + e.getMessage(), e);
            }
        }

        try {
            List<ClassLoader> classLoadersToLoad = new LinkedList<>();
            classLoadersToLoad.add(ClassUtils.getClassLoader());
            classLoadersToLoad.addAll(classLoaders);
            for (Set<URL> urls :
ClassLoaderResourceLoader.loadResources(fileName,
classLoadersToLoad).values()) {
                for (URL url : urls) {
                    InputStream is = url.openStream();
                    if (is != null) {
                        return readString(is);
                    }
                }
            }
```

```
        } catch (Throwable e) {
            logger.warn("Failed to load " + fileName + " file from " +
fileName + "(ignore this file): " + e.getMessage(), e);
        }
        return rawRule;
    }

    private static String readString(InputStream is) {
        StringBuilder stringBuilder = new StringBuilder();
        char[] buffer = new char[10];
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(is))) {
            int n;
            while ((n = reader.read(buffer)) != -1) {
                if (n < 10) {
                    buffer = Arrays.copyOf(buffer, n);
                }
                stringBuilder.append(String.valueOf(buffer));
                buffer = new char[10];
            }
        } catch (IOException e) {
            logger.error("Read migration file error.", e);
        }

        return stringBuilder.toString();
    }

    /**
     * check if the fileName can be found in filesystem
     *
     * @param fileName
     * @return
     */
    private static boolean checkFileNameExist(String fileName) {
        File file = new File(fileName);
        return file.exists();
    }
```

## 4. 创建 ModuleModel 对象

前面 ApplicationModel 对象初始化的时候创建了 ModuleModel 如下代码：

```
internalModule = new ModuleModel(this, true);
```

这里我们来看下这个它所对应的构造器：

```
public ModuleModel(ApplicationModel applicationModel, boolean isInternal) {
    //调用ScopeModel传递3个参数父模型，模型域为模块域，是否为内部模型参数为true
        super(applicationModel, ExtensionScope.MODULE, isInternal);
        Assert.notNull(applicationModel, "ApplicationModel can not be null");
        //初始化成员变量applicationModel
        this.applicationModel = applicationModel;
        //将模块模型添加至应用模型中
        applicationModel.addModule(this, isInternal);
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info(getDesc() + " is created");
        }
    //初始化模块模型
        initialize();
        Assert.notNull(serviceRepository, "ModuleServiceRepository can not be null");
        Assert.notNull(moduleConfigManager, "ModuleConfigManager can not be null");
        Assert.assertTrue(moduleConfigManager.isInitialized(), "ModuleConfigManager can not
be initialized");

        // notify application check state
        //获取应用程序发布对象，通知检查状态
        ApplicationDeployer applicationDeployer = applicationModel.getDeployer();
        if (applicationDeployer != null) {
            applicationDeployer.notifyModuleChanged(this, DeployState.PENDING);
        }
    }
```

### 1) 将模块模型添加至应用模型中

如上面代码所示调用如下代码将模块模型添加到应用模型中：

```
applicationModel.addModule(this, isInternal);
```

这里我们来看下添加过程

```
void addModule(ModuleModel moduleModel, boolean isInternal) {
    //加锁
        synchronized (moduleLock) {
          //不存在则添加
            if (!this.moduleModels.contains(moduleModel)) {
              //检查应用模型是否已销毁
                checkDestroyed();
                //添加至模块模型成员变量中
                this.moduleModels.add(moduleModel);
                //设置模块模型内部id，这个内部id生成过程与上面将应用模型添加到框架模型中的方式是一致的
                //可以参考 3.3.2 将ApplicationModel添加至FrameworkModel容器中
                moduleModel.setInternalId(buildInternalId(getInternalId(),
moduleIndex.getAndIncrement()));
                //如果不是内部模型则添加到公开模块模型中
                if (!isInternal) {
                    pubModuleModels.add(moduleModel);
                }
            }
        }
    }
```

### 2) 初始化模块模型

前面 ModuleModel 构造器中通过 initialize()方法来进行初始化操作如下代码：

```
@Override
    protected void initialize() {
      //调用域模型ScopeModel的初始化，可以参考 3.2.5 ScopeModel类型的初始化方法initialize()章节
        super.initialize();
        //创建模块服务存储库对象
        this.serviceRepository = new ModuleServiceRepository(this);
        //创建模块配置管理对象
        this.moduleConfigManager = new ModuleConfigManager(this);
        //初始化模块配置管理对象
        this.moduleConfigManager.initialize();

    //初始化模块配置扩展
        initModuleExt();

    //初始化域模型扩展
        ExtensionLoader<ScopeModelInitializer> initializerExtensionLoader =
this.getExtensionLoader(ScopeModelInitializer.class);
        Set<ScopeModelInitializer> initializers =
initializerExtensionLoader.getSupportedExtensionInstances();
        for (ScopeModelInitializer initializer : initializers) {
            initializer.initializeModuleModel(this);
        }
    }
```

### a) 模块服务存储库的创建

ModuleServiceRepository 是模块模型中用来存储服务的通过如下代码调用

```
//创建模块服务存储库对象
        this.serviceRepository = new ModuleServiceRepository(this);
```

这里我们来看下模块服务存储库的构造器代码：

```
 public ModuleServiceRepository(ModuleModel moduleModel) {
    //初始化模块模型
        this.moduleModel = moduleModel;
        //
        frameworkServiceRepository =
ScopeModelUtil.getFrameworkModel(moduleModel).getServiceRepository();
    }
```

ModuleServiceRepository 存储库中使用框架存储库 frameworkServiceRepository 来间接存储。

这里我们看下怎么通过模块模型获取框架服务存储库 frameworkServiceRepository：通过代码

```
ScopeModelUtil.getFrameworkModel(moduleModel).getServiceRepository()
```

ScopeModelUtil 工具类获取 getFrameworkModel 代码如下：

```
public static FrameworkModel getFrameworkModel(ScopeModel scopeModel) {
        if (scopeModel == null) {
            return FrameworkModel.defaultModel();
        }
        //通过成员变量获取(构造器初始化的时候将FrameworkModel赋值给了ApplicationModel的成员变量
        if (scopeModel instanceof ApplicationModel) {
          //直接获取
            return ((ApplicationModel) scopeModel).getFrameworkModel();
        } else if (scopeModel instanceof ModuleModel) {
            ModuleModel moduleModel = (ModuleModel) scopeModel;
            //间接通过ApplicationModel获取，不越级获取
            return moduleModel.getApplicationModel().getFrameworkModel();
        } else if (scopeModel instanceof FrameworkModel) {
            return (FrameworkModel) scopeModel;
        } else {
            throw new IllegalArgumentException("Unable to get FrameworkModel from " +
scopeModel);
        }
    }
```

### b) 模块配置管理器对象的创建与初始化

```
 //创建模块配置管理对象
        this.moduleConfigManager = new ModuleConfigManager(this);
        //初始化模块配置管理对象
        this.moduleConfigManager.initialize();
```

ModuleConfigManager 的构造器代码如下：

```
public ModuleConfigManager(ModuleModel moduleModel) {
  //向抽象的配置管理器AbstractConfigManager传递参数
  //模块模型参数，模块支持的配置类型集合
        super(moduleModel, Arrays.asList(ModuleConfig.class, ServiceConfigBase.class,
ReferenceConfigBase.class, ProviderConfig.class, ConsumerConfig.class));
        //获取应用程序配置管理器
        applicationConfigManager =
moduleModel.getApplicationModel().getApplicationConfigManager();
    }
```

ModuleConfigManager 类型的初始化方法代码如下：

```
@Override
    public void initialize() throws IllegalStateException {
        if (!initialized.compareAndSet(false, true)) {
            return;
        }
```

```
        //获取组合配置对象
        CompositeConfiguration configuration =
scopeModel.getModelEnvironment().getConfiguration();

        // dubbo.config.mode
        //3.3.4.1 提到过这里再重复一次 dubbo.config.mode 获取配置模式，配置模式
对应枚举类型 ConfigMode，目前有这么几个 STRICT，OVERRIDE，OVERRIDE_ALL，
OVERRIDE_IF_ABSENT，IGNORE，这个配置决定了属性覆盖的顺序，当有同一个配置 key 多
次出现时候，以最新配置为准，还是以最老的那个配置为准，还是配置重复则抛出异常，默认值
为严格模式 STRICT 重复则抛出异常
        String configModeStr = (String)
configuration.getProperty(ConfigKeys.DUBBO_CONFIG_MODE);
        try {
            if (StringUtils.hasText(configModeStr)) {
                this.configMode =
ConfigMode.valueOf(configModeStr.toUpperCase());
            }
        } catch (Exception e) {
            String msg = "Illegal '" + ConfigKeys.DUBBO_CONFIG_MODE + "'
config value [" + configModeStr + "], available values " +
Arrays.toString(ConfigMode.values());
            logger.error(msg, e);
            throw new IllegalArgumentException(msg, e);
        }

        // dubbo.config.ignore-duplicated-interface
        //忽略重复的接口（服务/引用）配置。默认值为 false
        String ignoreDuplicatedInterfaceStr = (String) configuration
            .getProperty(ConfigKeys.DUBBO_CONFIG_IGNORE_DUPLICATED_INTER
FACE);
        if (ignoreDuplicatedInterfaceStr != null) {
            this.ignoreDuplicatedInterface =
Boolean.parseBoolean(ignoreDuplicatedInterfaceStr);
        }

        // print 打印配置信息到控制台
        Map<String, Object> map = new LinkedHashMap<>();
        map.put(ConfigKeys.DUBBO_CONFIG_MODE, configMode);
        map.put(ConfigKeys.DUBBO_CONFIG_IGNORE_DUPLICATED_INTERFACE,
this.ignoreDuplicatedInterface);
        logger.info("Config settings: " + map);
    }
```

### c) 模块配置扩展的初始化

```
initModuleExt();
```

```
private void initModuleExt() {
//目前这里的扩展只支持有一个类型ModuleEnvironment
        Set<ModuleExt> exts =
this.getExtensionLoader(ModuleExt.class).getSupportedExtensionInstances();
        for (ModuleExt ext : exts) {
            ext.initialize();
        }
    }
```

ModuleEnvironment 的初始化

```
  @Override
    public void initialize() throws IllegalStateException {
        this.orderedPropertiesConfiguration = new
OrderedPropertiesConfiguration(moduleModel);
    }
```

OrderedPropertiesConfiguration 对象的创建

```
public OrderedPropertiesConfiguration(ModuleModel moduleModel) {
        this.moduleModel = moduleModel;
        refresh();
    }

    public void refresh() {
        properties = new Properties();
        //有序的配置提供器扩展获取
        ExtensionLoader<OrderedPropertiesProvider> propertiesProviderExtensionLoader =
moduleModel.getExtensionLoader(OrderedPropertiesProvider.class);
        Set<String> propertiesProviderNames =
propertiesProviderExtensionLoader.getSupportedExtensions();
        if (CollectionUtils.isEmpty(propertiesProviderNames)) {
            return;
        }
        List<OrderedPropertiesProvider> orderedPropertiesProviders = new ArrayList<>();
        for (String propertiesProviderName : propertiesProviderNames) {

orderedPropertiesProviders.add(propertiesProviderExtensionLoader.getExtension(propertiesProv
iderName));
        }

        //order the propertiesProvider according the priority descending
        //根据优先级进行排序，值越小优先级越高
        orderedPropertiesProviders.sort((a, b) -> b.priority() - a.priority());


        //override the properties. 目前没看到有具体的扩展实现
        for (OrderedPropertiesProvider orderedPropertiesProvider :
orderedPropertiesProviders) {
            properties.putAll(orderedPropertiesProvider.initProperties());
        }

    }
```

# SPI 扩展机制

## 一、 Dubbo 的扩展机制概述

### 1. 回顾我们前面使用到扩展场景

在上一章中我们初始化应用模型对象的时候，了解到有几个地方用到了扩展机制来创建对象，这一章我们会详细来讲一下这个扩展对象的加载过程，这里我们先来回顾下哪些地方用到了扩展机制：

```
// 使用扩展机制获取 TypeBuilder
Set<TypeBuilder> tbs =
model.getExtensionLoader(TypeBuilder.class).getSupportedExtensionInsta
nces();

//获取域模型初始化器 ScopeModelInitializer 扩展对象,执行初始化方法
ExtensionLoader<ScopeModelInitializer> initializerExtensionLoader =
this.getExtensionLoader(ScopeModelInitializer.class);
 Set<ScopeModelInitializer> initializers =
initializerExtensionLoader.getSupportedExtensionInstances();

// OrderedPropertiesConfiguration 中获取有序配置提供器对象
ExtensionLoader<OrderedPropertiesProvider>
propertiesProviderExtensionLoader =
moduleModel.getExtensionLoader(OrderedPropertiesProvider.class);

// ApplicationModel 中获取配置管理器对象
 configManager = (ConfigManager)
this.getExtensionLoader(ApplicationExt.class)
              .getExtension(ConfigManager.NAME);

//ModuleModel 中获取模块扩展对象
Set<ModuleExt> exts =
this.getExtensionLoader(ModuleExt.class).getSupportedExtensionInstance
s();

// ApplicationModel 中获 Environment 对象
environment = (Environment)
this.getExtensionLoader(ApplicationExt.class)
              .getExtension(Environment.NAME);
```

```
// ApplicationModel 中获取应用初始化监听器 ApplicationInitListener 扩展对象
ExtensionLoader<ApplicationInitListener> extensionLoader =
this.getExtensionLoader(ApplicationInitListener.class);
Set<String> listenerNames = extensionLoader.getSupportedExtensions();

//ScopeModel 中创建扩展访问器:
this.extensionDirector = new ExtensionDirector(parent != null ?
parent.getExtensionDirector() : null, scope, this);
```

有了以上的应用场景我们可以来看下扩展机制了。

## 2. 为什么要用到扩展机制?

为什么要用到扩展这个想必每个编程人员都比较了解，一个好的程序是要遵循一定的设计规范比如设计模式中的开闭原则，英文全称是 Open Closed Principle，简写为 OCP，对扩展开放、对修改关闭:

- **对扩展开放**: 指的是我们系统中的模块、类、方法对它们的提供者（开发者）应该是开放的，提供者可以对系统进行扩展（新增）新的功能。

- **对修改关闭**: 指的是系统中的模块、类、方法对它们的使用者（调用者）应该是关闭的。使用者使用这些功能时，不会因为提供方新增了功能而导致使用者也进行相应修改。

我们再来了解下 Dubbo 的一些基本特性。

下面这句话是我摘自官网的:

*Apache Dubbo 是一款微服务开发框架，它提供了 RPC 通信与微服务治理两大关键能力。这意味着，使用 Dubbo 开发的微服务，将具备相互之间的远程发现与通信能力，同时利用 Dubbo 提供的丰富服务治理能力，可以实现诸如服务发现、负载均衡、流量调度等服务治理诉求。同时 Dubbo 是高度可扩展的，用户几乎可以在任意功能点去定制自己的实现，以改变框架的默认行为来满足自己的业务需求。*

*Dubbo3 基于 Dubbo2 演进而来，在保持原有核心功能特性的同时，Dubbo3 在易用性、超大规模微服务实践、云原生基础设施适配、安全设计等几大方向上进行了全面升级。以下文档都将基于 Dubbo3 展开。*

- **对修改关闭的地方**：对于 Apache Dubbo 来说不变的是 RPC 调用流程，微服务治理这些抽象的概念，我们可以用摘自官网的下面几个图表示。

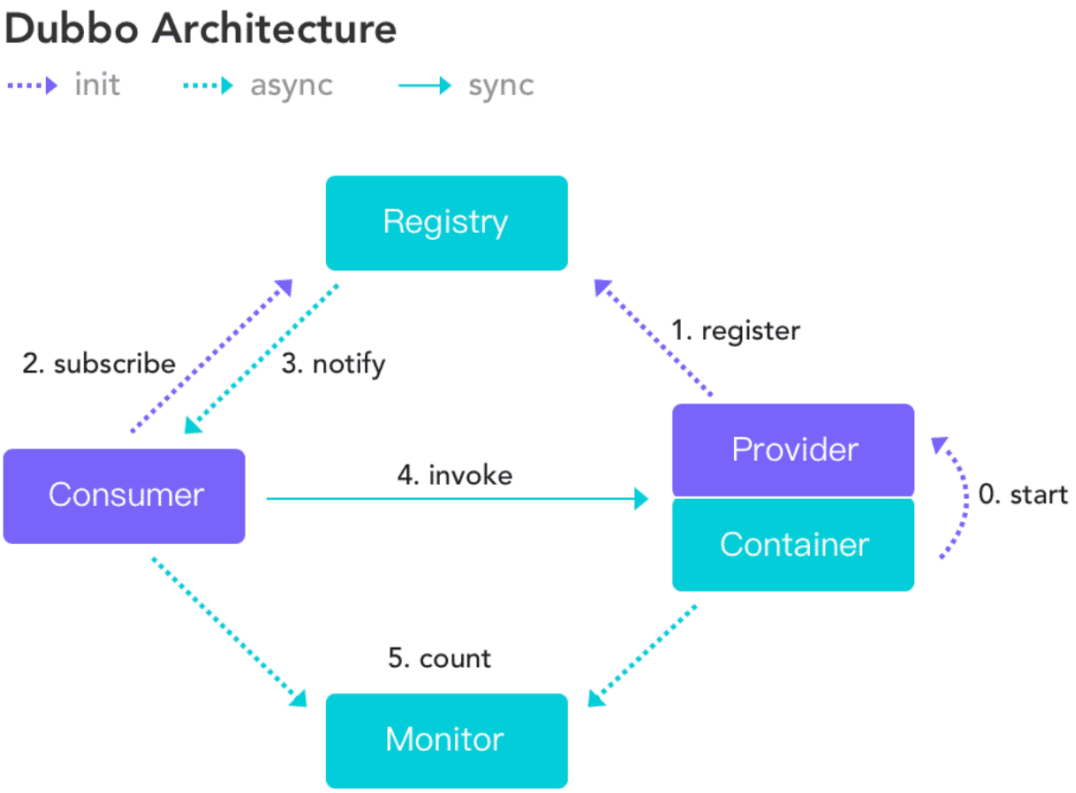



图 4.1 Dubbo 架构图

再来看一个调用链路的架构图

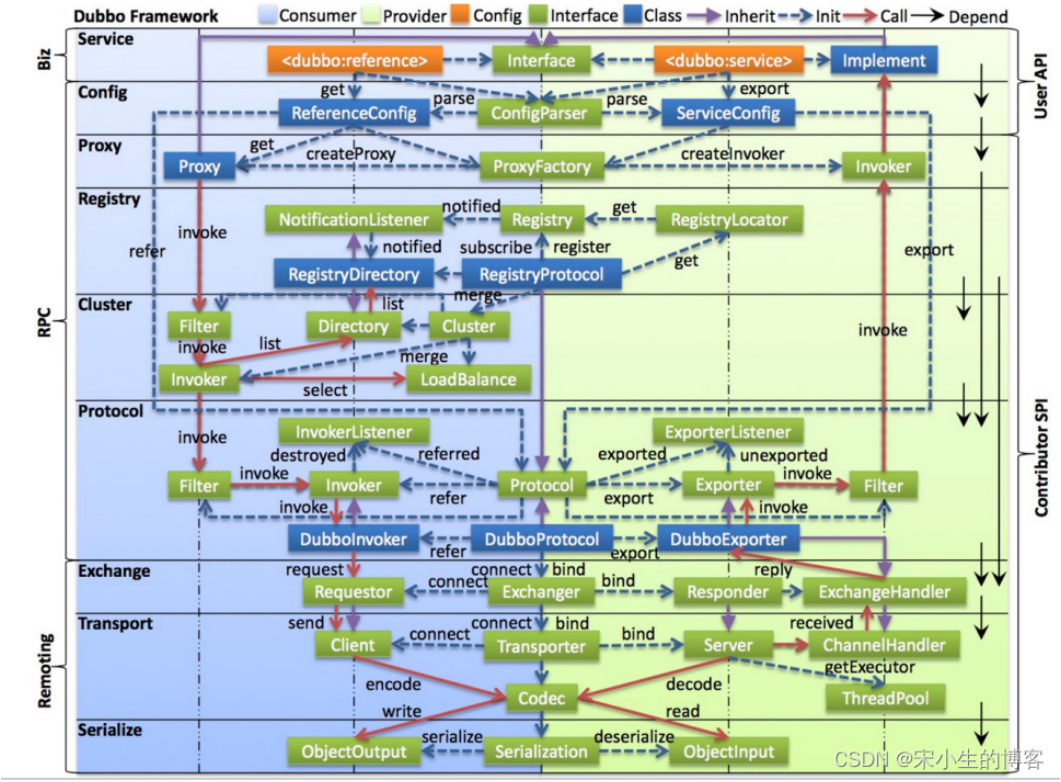



图 4.2 Dubbo RPC 调用链路

上面两个图整体来看都是 Dubbo 不变的地方涉及到服务的 RPC 调用和服务治理的一些概念与流程，但是对于每个环节又可以使用各种方式实现，比如序列化机制可以是 Json，Java 序列化，Hession2 或者 Protobuf 等等，网络传输层可以是 netty 实现的 tcp 通信，也可以使用 http 协议，那 Dubbo 又是如何封装不变部分扩展这种可变部分呢？那就是接下来要说的微内核机制，这个我们待会说。

- **对扩展开放**：对于 Apache Dubbo 来说变化的是 RPC 调用流程和微服务治理这些抽象的概念的具体实现，每个点应该用什么技术实现，又是用什么场景，这个可以用如下图来表示下：

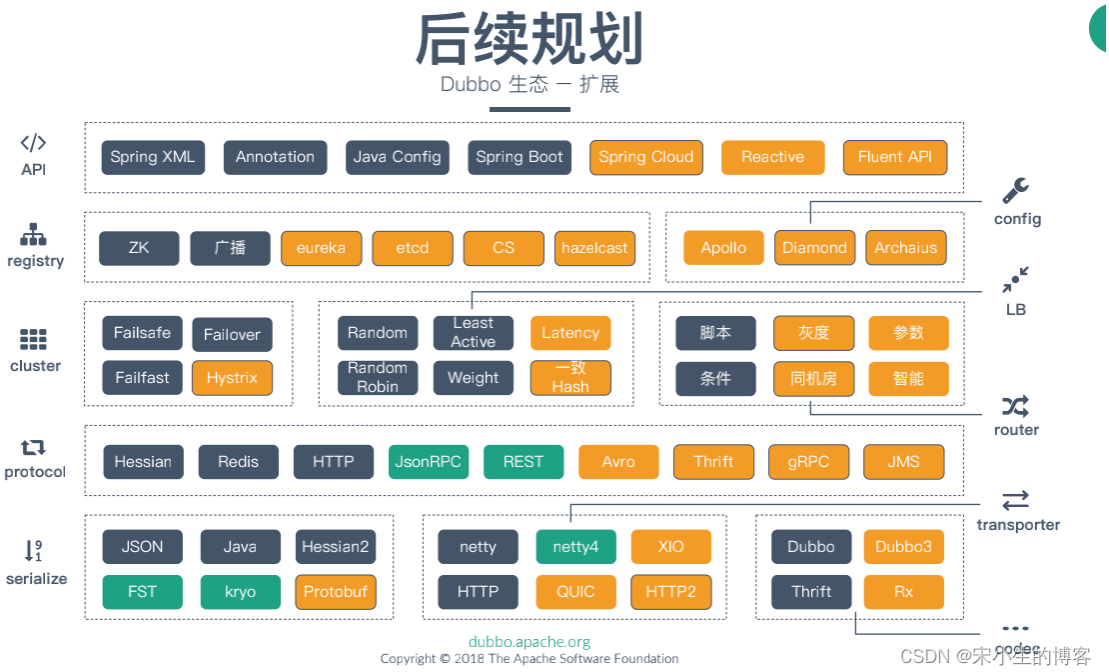



图 4.3 Dubbo 的扩展生态

看到这里应该各位就明白了，我们写程序是为了业务，而针对不同的业务需求很多场景下我们是需要使用不同的实现来满足的，Dubbo 使用微内核的架构，将具体的实现开放出来，让使用者可以根据自己的需求来选择，定制.Dubbo 开放了很多的扩展点供大家扩展，可想而知使用 Dubbo 的灵活性是非常高的。

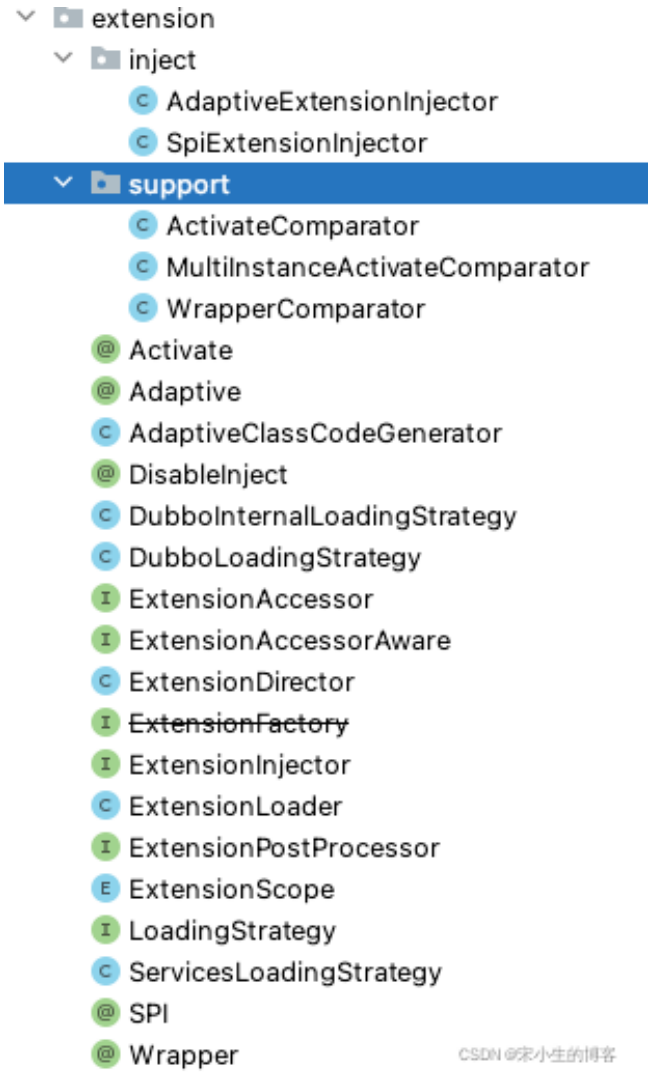
- **微内核架构**：微内核架构由两大架构模块组成：核心系统与插件模块，设计一个微内核体系关键工作全部集中于核心系统怎么构建。

- **核心系统**：负责和具体业务功能无关的通用功能，例如模块加载、模块间通信等，这个其实对应着 Dubbo 的 SPI 机制。

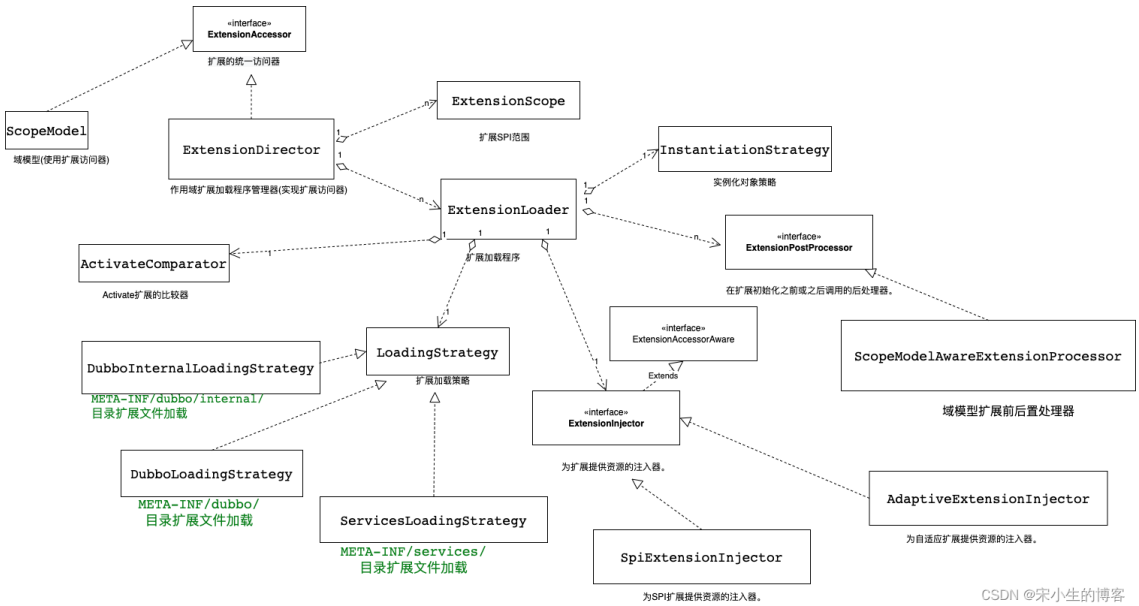
- **插件模块**：负责实现具体的业务逻辑，Dubbo，SPI 接口与实现。

## 3. Dubbo 的扩展机制包含了哪些重要的组成部分?

前面我们说了为什么要使用扩展机制，这里我们来看下具体实现。先将扩展包里面的代码截个图认识认识各类型的单词




顺便我们先简单看下类结构图，后续再详细看每个类型的解释



为了后续看具体的扩展加载流程我们先看下以上类型的解释说明：

- **ExtensionAccessor**
  扩展的统一访问器。

- **ExtensionDirector**
  ExtensionDirector 是一个作用域扩展加载程序管理器。ExtensionDirector 支持多个级别，子级可以继承父级的扩展实例。查找和创建扩展实例的方法类似于 Java classloader。

- **ExtensionScope**

  - 扩展 SPI 域，目前有 FRAMEWORK，APPLICATION，MODULE，SELF
  - FRAMEWORK：扩展实例在框架内使用，与所有应用程序和模块共享。
    框架范围 SPI 扩展只能获取 FrameworkModel，无法获取 ApplicationModel 和 ModuleModel。考虑：一些 SPI 需要在框架内的应用程序之间共享数据无状态 SPI 在框架内是安全共享的。

- **APPLICATION**
  扩展实例在一个应用程序中使用，与应用程序的所有模块共享，不同的应用程序创建不同的扩展实例。

  - MODULE 扩展实例在一个模块中使用，不同的模块创建不同的扩展实例。模块范围 SPI 扩展可以获得 FrameworkModel、ApplicationModel 和 ModuleModel。考虑：隔离应用程序内部不同模块中的扩展数据。
  - SELF 自给自足，为每个作用域创建一个实例，用于特殊的 SPI 扩展，如 ExtensionInjector。

- **ExtensionLoader**

  - ApplicationModel、DubboBootstrap 和这个类目前被设计为单例或静态（本身完全静态或使用一些静态字段）。因此，从它们返回的实例属于

process 或 classloader 范围。如果想在一个进程中支持多个 dubbo 服务器，可能需要重构这三个类。

- 加载 dubbo 扩展
- 自动注入依赖项扩展
- 包装器中的自动包装扩展
- 默认扩展是一个自适应实例
- JDK 自带 SPI 参考地址点击查看
- @SPI 服务扩展接口详细内容看后面
- @Adaptive 自适应扩展点注解详细内容看后面
- @Activate 自动激活扩展点注解详细内容看后面

- **ExtensionPostProcessor**
  在扩展初始化之前或之后调用的后处理器。

- **LoadingStrategy**
  扩展加载策略，目前有 3 个扩展加载策略分别从不同文件目录加载扩展。

- **DubboInternalLoadingStrategy**
  Dubbo 内置的扩展加载策略，将加载文件目录为 META-INF/dubbo/internal/的扩展。

- **DubboLoadingStrategy**
  Dubbo 普通的扩展加载策略，将加载目录为 META-INF/dubbo/的扩展。

- **ServicesLoadingStrategy**
  JAVA SPI 加载策略，将加载目录为 META-INF/services/的扩展。

- **Wrapper 注解**

- **SPI 注解**

- **ExtensionInjector 接口**
  为 SPI 扩展提供资源的注入器。

- **ExtensionAccessorAware**
  SPI 扩展可以实现这个感知接口，以获得适当的 xtensionAccessor 实例。

- **DisableInject 注解**

- **AdaptiveClassCodeGenerator**
  自适应类的代码生成器。

- **Adaptive 注解**
  为 ExtensionLoader 注入依赖扩展实例提供有用信息。

- **Activate 注解**
  Activate。此注解对于使用给定条件自动激活某些扩展非常有用，例如：@Activate 可用于在有多个实现时加载某些筛选器扩展。

  - group()指定组条件。框架 SPI 定义了有效的组值。
  - value()指定 URL 条件中的参数键。
  - SPI 提供程序可以调用 ExtensionLoader。
  - getActivateExtension（URL、String、String）方法以查找具有给定条件的所有已激活扩展。

- **ActivateComparator**
  Activate 扩展的排序器。

- **MultiInstanceActivateComparator**

- **WrapperComparator**

- **AdaptiveExtensionInjector**

- **SpiExtensionInjector**

## 4. 扩展加载创建之前的调用过程

### 1) 扩展的调用代码示例

了解了这么多与扩展相关的概念，接下来我们就来从前面的代码调用中找几个例子来看下扩展的调用过程。

代码来源于 FrameworkModel 对象的初始化 initialize()中的如下代码调用：

```
TypeDefinitionBuilder.initBuilders(this);
```

TypeDefinitionBuilder 中初始化类型构建器代码如下：

```
  public static void initBuilders(FrameworkModel model) {
        Set<TypeBuilder> tbs =
model.getExtensionLoader(TypeBuilder.class).getSupportedExtensionInstances();
        BUILDERS = new ArrayList<>(tbs);
    }
```

### 2) Dubbo 的分层模型获取扩展加载器对象

以上扩展调用的时候对于扩展加载器对象的获取代码如下所示，我们来看下它的调用链路。

```
model.getExtensionLoader(TypeBuilder.class)
```

getExtensionLoader 方法来源于 FrameworkModel 类型的父类型 ScopeModel 的实现的接口 ExtensionAccessor 中的默认方法（JDK8 默认方法）。

ExtensionAccessor 接口中的 getExtensionLoader 方法如下代码：

```
default <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
        return this.getExtensionDirector().getExtensionLoader(type);
    }
```

获取扩展加载器之前需要先获取扩展访问器。这里的链路先梳理下：

模型对象（FrameworkModel）--->扩展访问器（ExtensionAccessor）--->作用域扩展加载程序管理器（ExtensionDirector）--->

这个 getExtensionDirector()方法来源于 FrameworkModel 的抽象父类型 ScopeModel 中的 getExtensionDirector()如下代码：

```
@Override
    public ExtensionDirector getExtensionDirector() {
        return extensionDirector;
    }
```

这里直接返回了 extensionDirector，不知道介绍到这里记得这个扩展加载程序管理器 extensionDirector 对象的由来不，在上个章节《3-框架，应用程序，模块领域模型 Model 对象的初始化》中 3.2.2 初始化 ScopeModel 的章节中的 ScopeModel 类型的初始化方法 initialize()方法中我们提到过这个对象的创建，具体代码如下所示（这个代码比较简单）：

```
this.extensionDirector = new ExtensionDirector(parent != null ?
parent.getExtensionDirector() : null, scope, this);
```

我们继续前面 getExtensionLoader（type）方法调用逻辑，前面我们知道了这个扩展访问器的对象是 ExtensionDirector，接下来我们看下 ExtensionDirector 中获取扩展加载器的代码（如下所示）。

在详细介绍扩展加载器对象获取之前我们先来看下当前我们要加载的扩展类型的源码，后续会用到。

我们要加载的扩展类型 TypeBuilder 接口

```
@SPI(scope = ExtensionScope.FRAMEWORK)
public interface TypeBuilder extends Prioritized {

    /**
     * Whether the build accept the class passed in.
     */
    boolean accept(Class<?> clazz);

    /**
     * Build type definition with the type or class.
     */
    TypeDefinition build(Type type, Class<?> clazz, Map<String, TypeDefinition> typeCache);

}
```

ExtensionDirector 类型中获取扩展加载器的代码。这个代码非常有意思其实就是前面说到的域模型架构的数据访问架构类似于 JVM 类加载器访问加载类的情况，但是这个顺序可能有所不同，Dubbo 的扩展加载器是如何访问的呢？遵循以下顺序：

- 先从缓存中查询扩展加载器。
- 如果前面没找到则查询扩展类型的 scope 所属域，如果是当前域扩展则从直接创建扩展加载器。
- 如果前面没找到就从父扩展访问器中查询，查询这个扩展是否数据父扩展域。
- 前面都没找到就尝试创建。

```
@Override
    public <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
         //如果扩展加载器已经被销毁则抛出异常
        checkDestroyed();
        //这里参数类型传的是 TypeBuilder.class 不为空
        if (type == null) {
            throw new IllegalArgumentException("Extension type ==
null");
        }
        //扩展类型不为接口也要抛出异常,这个 TypeBuilder.class 具体类型代码往上看,
这个类型是一个接口
        if (!type.isInterface()) {
            throw new IllegalArgumentException("Extension type (" + type
+ ") is not an interface!");
        }
        //这个判断逻辑是判断这个扩展接口是有有@SPI 注解,TypeBuilder 是有的
        if (!withExtensionAnnotation(type)) {
            throw new IllegalArgumentException("Extension type (" + type
+
```

```
                ") is not an extension, because it is NOT annotated with
@" + SPI.class.getSimpleName() + "!");
        }

        // 1. find in local cache
        //被加载的扩展类型对应的扩展加载器会放到 extensionLoadersMap 这个
ConcurrentHashMap 类型的集合中方便缓存
        ExtensionLoader<T> loader = (ExtensionLoader<T>)
extensionLoadersMap.get(type);
                //查询扩展所属域,这个类型的扩展域是框架级别的
ExtensionScope.FRAMEWORK
       //extensionScopeMap 为 ConcurrentHashMap 类型的扩展域缓存集合
        ExtensionScope scope = extensionScopeMap.get(type);
        if (scope == null) {
            SPI annotation = type.getAnnotation(SPI.class);
            scope = annotation.scope();
            extensionScopeMap.put(type, scope);
        }
                //首次访问的时候当前类型的扩展加载器类型肯定是空的,会走如下两个逻
辑中的其中一个进行创建扩展加载器
                //1)如果 扩展域为 SELF 自给自足，为每个作用域创建一个实例，用于特
殊的 SPI 扩展，如{@link ExtensionInjector}
        if (loader == null && scope == ExtensionScope.SELF) {
            // create an instance in self scope
            loader = createExtensionLoader0(type);
        }

        // 2. find in parent
        //3) 从父扩展加载器中查询当前扩展加载器是否存在,这里 parent 是空的先不考虑
        if (loader == null) {
            if (this.parent != null) {
                loader = this.parent.getExtensionLoader(type);
            }
        }

        // 3. create it
        //4) 这个是我们本次会走的逻辑,大部分是会走这个逻辑来创建扩展加载器对象的
        if (loader == null) {
            loader = createExtensionLoader(type);
        }

        return loader;
    }
```

前面提到的 withExtensionAnnotation 判断代码如下：

```
private static boolean withExtensionAnnotation(Class<?> type) {
        return type.isAnnotationPresent(SPI.class);
    }
```

ExtensionDirector 类型的 createExtensionLoader 方法

```
private <T> ExtensionLoader<T> createExtensionLoader(Class<T> type) {
        ExtensionLoader<T> loader = null;
        //当前类型注解的scope与当前扩展访问器ExtensionDirector的scope是否一致,不一致则抛出异常
        //当前类型ExtensionDirector的scope是在构造器中传递的,在Model对象初始化的时候创建的本类型
        if (isScopeMatched(type)) {
            // if scope is matched, just create it
            loader = createExtensionLoader0(type);
        } else {
            // if scope is not matched, ignore it
        }
        return loader;
    }
```

ExtensionDirector 类型的 createExtensionLoader0 方法

```
 private <T> ExtensionLoader<T> createExtensionLoader0(Class<T> type) {
    //检查当前扩展访问器是否被销毁掉了
        checkDestroyed();
        ExtensionLoader<T> loader;
        //为当前扩展类型创建一个扩展访问器并缓存到,当前成员变量extensionLoadersMap中
        extensionLoadersMap.putIfAbsent(type, new ExtensionLoader<T>(type, this,
scopeModel));
        loader = (ExtensionLoader<T>) extensionLoadersMap.get(type);
        return loader;
    }
```

### 3) 扩展加载器对象 ExtensionLoader 的构造器

扩展加载器相对来说是比较复杂的实现内容比较多，用到哪里我们说下哪里，这里先来看 ExtensionLoader 的构造器代码如下所示：

```
 ExtensionLoader(Class<?> type, ExtensionDirector extensionDirector, ScopeModel scopeModel)
{
    //当前扩展加载器,需要加载的扩展的类型
        this.type = type;
        //创建扩展加载器的扩展访问器对象
        this.extensionDirector = extensionDirector;
        //从扩展访问器中获取扩展执行前后的回调器
        this.extensionPostProcessors = extensionDirector.getExtensionPostProcessors();
        //创建实例化对象的策略对象
        initInstantiationStrategy();
        //如果当前扩展类型为扩展注入器类型则设置当前注入器变量为空,否则的话获取一个扩展注入器扩展对象
        this.injector = (type == ExtensionInjector.class ? null :
extensionDirector.getExtensionLoader(ExtensionInjector.class)
            .getAdaptiveExtension());
         //创建Activate注解的排序器
        this.activateComparator = new ActivateComparator(extensionDirector);
        //为扩展加载器下的域模型对象赋值
        this.scopeModel = scopeModel;
    }
```

先来看创建实例化对象的策略对象代码 initInstantiationStrategy();

```
private void initInstantiationStrategy() {
        for (ExtensionPostProcessor extensionPostProcessor : extensionPostProcessors) {
              //ScopeModelAwareExtensionProcessor在域模型对象时候为扩展访问器添加了这个域模型扩展处理
器对象ScopeModelAwareExtensionProcessor,这个类型实现了ScopeModelAccessor域模型访问器可以用来获取域模
型对象
            if (extensionPostProcessor instanceof ScopeModelAccessor) {

                instantiationStrategy = new InstantiationStrategy((ScopeModelAccessor)
extensionPostProcessor);
                break;
            }
        }
        if (instantiationStrategy == null) {
            instantiationStrategy = new InstantiationStrategy();
        }
    }
```

再来看 ExtensionInjector 扩展对象的获取

```
//1)这里有个type为空的判断,普通的扩展类型肯定不是ExtensionInjector类型 这里必定会为每个非扩展注入
ExtensionInjector类型创建一个ExtensionInjector类型的扩展对象,
//2) 这里代码会走extensionDirector.getExtensionLoader(ExtensionInjector.class)这一步进去之后的代
码刚刚看过就不再看了,这个代码会创建一个为ExtensionInjector扩展对象的加载器对象ExtensionLoader
//3) getAdaptiveExtension() 这个方法就是通过扩展加载器获取具体的扩展对象的方法我们会详细说
 this.injector = (type == ExtensionInjector.class ? null :
extensionDirector.getExtensionLoader(ExtensionInjector.class)
            .getAdaptiveExtension());
```

## 二、 自适应扩展对象的创建及 getAdaptiveExtension 方法

自适应扩展又称为动态扩展，可以在运行时生成扩展对象 ExtensionLoader 中的 getAdaptiveExtension()方法,这个方法也是我们看到的第一个获取扩展对象的方法。这个方法可以帮助我们通过 SPI 机制从扩展文件中找到需要的扩展类型并创建它的对象。

自适应扩展：如果对设计模式比较了解的可能会联想到适配器模式，自适应扩展其实就是适配器模式的思路，自适应扩展有两种策略：

- 一种是我们自己实现自适应扩展：然后使用@Adaptive 修饰这个时候适配器的逻辑由我们自己实现，当扩展加载器去查找具体的扩展的时候可以通过找到我们这个对应的适配器扩展，然后适配器扩展帮忙去查询真正的扩展，这个比如我们下面要举的扩展注入器的例子，具体扩展通过扩展注入器适配器，注入器适配器来查询具体的注入器扩展实现来帮忙查找扩展。

- 还有一种方式是我们未实现这个自适应扩展，Dubbo 在运行时通过字节码动态代理的方式在运行时生成一个适配器，使用这个适配器映射到具体的扩展。第二种情况往往用在比如 Protocol、Cluster、LoadBalance 等。有时，有些拓展并不想在框架启动阶段被加载，而是希望在拓展方法被调用时，根据运行时参数进行加载。（如果还不了解可以考虑看下@Adaptive 注解加载方法上面的时候扩展是如何加载的）

```
public T getAdaptiveExtension() {
                //检查当前扩展加载器是否已经被销毁
        checkDestroyed();
        //从自适应扩展缓存中查询扩展对象如果存在就直接返回,这个自适应扩展类型只会有一个扩展实现类型如果是多个的话根据是否可以覆盖参数决定扩展实现类是否可以相互覆盖
        Object instance = cachedAdaptiveInstance.get();
        //这个 if 判断不太优雅 容易多层嵌套,上面 instance 不为空就可以直接返回了
        if (instance == null) {
         //创建异常则抛出异常直接返回(多线程场景下可能第一个线程异常了第二个线程进来之后走到这里)
            if (createAdaptiveInstanceError != null) {
                throw new IllegalStateException("Failed to create adaptive instance: " +
```

```
                createAdaptiveInstanceError.toString(),
                createAdaptiveInstanceError);
        }
                    //加锁排队 (单例模式创建对象的思想 双重校验锁)
        synchronized (cachedAdaptiveInstance) {
              //加锁的时候对象都是空的,进来之后先判断下防止重复创建
            instance = cachedAdaptiveInstance.get();
            //只有第一个进来锁的对象为空开始创建扩展对象
            if (instance == null) {
                try {
                        //根据 SPI 机制获取类型,创建对象
                    instance = createAdaptiveExtension();
                    //存入缓存
                    cachedAdaptiveInstance.set(instance);
                } catch (Throwable t) {
                    createAdaptiveInstanceError = t;
                    throw new IllegalStateException("Failed to create
adaptive instance: " + t.toString(), t);
                }
            }
        }

        return (T) instance;
    }
```

前面使用单例思想来调用创建自适应扩展对象的方法，下面就让我们深入探究下创建自适应扩展对象的整个过程 createAdaptiveExtension()；方法。

### 1. 创建扩展对象的生命周期方法：注意这个后续会详细解析这个声明周期方法的细节

createAdaptiveExtension()

我们先来看 ExtensionLoader 类型中的 createAdaptiveExtension();方法，这个方法包含了扩展对象创建初始化的整个生命周期，如下代码所示：

```
private T createAdaptiveExtension() {
        try {
          //获取扩展类型实现类，创建扩展对象
            T instance = (T) getAdaptiveExtensionClass().newInstance();
            //注入扩展对象之前的回调方法
            instance = postProcessBeforeInitialization(instance, null);
            //注入扩展对象
            instance = injectExtension(instance);
            //注入扩展对象之后的回调方法
            instance = postProcessAfterInitialization(instance, null);
            //初始化扩展对象的属性,如果当前扩展实例的类型实现了Lifecycle则调用当前扩展对象的生命周期回调方法initialize()(来自Lifecycle接口)
            //参考样例第一个instance为ExtensionInjector的自适应扩展对象类型为AdaptiveExtensionInjector,自适应扩展注入器(适配器)用来查询具体支持的扩展注入器比如scope,spi,spring注入器
            initExtension(instance);
            return instance;
        } catch (Exception e) {
            throw new IllegalStateException("Can't create adaptive extension " + type + ", cause: " + e.getMessage(), e);
        }
    }
```

## 2. SPI 机制获取扩展对象实现类型 getAdaptiveExtensionClass()

这个方法可以帮助我们了解具体的 Dubbo SPI 机制如果找到扩展类型的实现类，会寻找哪些文件，扩展文件的优先级又是什么，对我们自己写扩展方法很有帮助，接下来我们就来看下它的源码：

```
 private Class<?> getAdaptiveExtensionClass() {
    //获取扩展类型,将扩展类型存入成员变量cachedClasses中进行缓存
        getExtensionClasses();
        //在上个方法的详细解析中的最后一步loadClass方法中如果扩展类型存在Adaptive注解将会将扩展类型赋值给cachedAdaptiveClass,否则的话会把扩展类型都缓存起来存储在扩展集合extensionClasses中
        if (cachedAdaptiveClass != null) {
            return cachedAdaptiveClass;
        }
        //扩展实现类型没有一个这个自适应注解Adaptive时候会走到这里
        //刚刚我们扫描到了扩展类型然后将其存入cachedClasses集合中了 接下来我们看下如何创建扩展类型
        return cachedAdaptiveClass = createAdaptiveExtensionClass();
    }
```

继续看获取扩展类型的方法 getExtensionClasses()

```
private Map<String, Class<?>> getExtensionClasses() {
    //缓存中查询扩展类型是否存在
        Map<String, Class<?>> classes = cachedClasses.get();
        if (classes == null) {
          //单例模式双重校验锁判断
            synchronized (cachedClasses) {
                classes = cachedClasses.get();
                if (classes == null) {
                  //加载扩展类型
                    classes = loadExtensionClasses();
                    //将我们扫描到的扩展类型存入成员变量cachedClasses中
                    cachedClasses.set(classes);
                }
            }
        }
        return classes;
    }
```

### 1) 使用不同的的策略加载加载不同目录下的扩展

加载扩展类型的方法 loadExtensionClasses()

```
  private Map<String, Class<?>> loadExtensionClasses() {
      //检查扩展加载器是否被销毁
        checkDestroyed();
        //缓存默认的扩展名到成员变量cachedDefaultName中
        cacheDefaultExtensionName();
    //加载到的扩展集合
        Map<String, Class<?>> extensionClasses = new HashMap<>();
    //扩展策略,在4.3章节中我们介绍了这个类型的UML与说明
    //LoadingStrategy扩展加载策略,目前有3个扩展加载策略
    //DubboInternalLoadingStrategy:Dubbo内置的扩展加载策略,将加载文件目录为META-
INF/dubbo/internal/的扩展
    //DubboLoadingStrategy:Dubbo普通的扩展加载策略,将加载目录为META-INF/dubbo/的扩展
    //ServicesLoadingStrategy:JAVA SPI加载策略 ,将加载目录为META-INF/services/的扩展
    //扩展策略集合对象在什么时候初始化的呢在成员变量初始化的时候就创建了集合对象,这个可以看方法
loadLoadingStrategies() 通过Java的 SPI加载策略
        for (LoadingStrategy strategy : strategies) {
          //根据策略从指定文件目录中加载扩展类型
            loadDirectory(extensionClasses, strategy, type.getName());

            // compatible with old ExtensionFactory
            //如果当前要加载的扩展类型是扩展注入类型则扫描下ExtensionFactory类型的扩展
            if (this.type == ExtensionInjector.class) {
              //这个方法和上面那个方法是一样的就不详细说了 扫描文件 找到扩展类型
                loadDirectory(extensionClasses, strategy, ExtensionFactory.class.getName());
            }
        }
    //通过loadDirectory扫描 扫描到了ExtensionInjector类型的扩展实现类有3个 我们将会得到这样一个集合例
子:
    //"spring" ->  "class org.apache.dubbo.config.spring.extension.SpringExtensionInjector"
    //"scopeBean" ->  "class org.apache.dubbo.common.beans.ScopeBeanExtensionInjector"
    //"spi" ->  "class org.apache.dubbo.common.extension.inject.SpiExtensionInjector"
        return extensionClasses;
    }
```

从文件中加载扩展实现 loadDirectory 方法

```
 private void loadDirectory(Map<String, Class<?>> extensionClasses, LoadingStrategy
strategy, String type) {
    //加载并根据策略的参数来加载
        loadDirectory(extensionClasses, strategy.directory(), type,
strategy.preferExtensionClassLoader(),
            strategy.overridden(), strategy.includedPackages(), strategy.excludedPackages(),
strategy.onlyExtensionClassLoaderPackages());
            //下面两行就是要兼容alibaba的扩展包了
        String oldType = type.replace("org.apache", "com.alibaba");
        loadDirectory(extensionClasses, strategy.directory(), oldType,
strategy.preferExtensionClassLoader(),
            strategy.overridden(), strategy.includedPackagesInCompatibleType(),
strategy.excludedPackages(), strategy.onlyExtensionClassLoaderPackages());
    }
```

带扩展策略参数的 loadDirectory 方法

关于扩展策略的参数列表我这里列个表格方便大家来看

| 扩展类型 | Dir（目录） | extensionLoaderClassLoaderFirst（优先扩展类型的类加载器） | Overridden（是否允许覆盖同名扩展） | includedPackages（明确包含的扩展包） | excludedPackages（明确排除的扩展包） | onlyExtensionClassLoaderPackages（限制应该从 Dubbo 的类加载器加载的类） |
|---|---|---|---|---|---|---|
| DubboInternalLoadingStrategy | META-INF/dubbo/internal/ | false | false | null | null | [] |
| DubboLoadingStrategy | META-INF/dubbo/ | false | true | null | null | [] |
| ServicesLoadingStrategy | META-INF/services/ | false | true | null | null | [] |

```
/**
     * 不同的扩展策略传递了不同的参数,但是扩展的加载流程是相同的,这里我们可以参考上
面表格
     * @param extensionClasses
     * @param dir
     * @param type 这里我们参考的示例这个值为
org.apache.dubbo.common.extension.ExtensionInjector
     * @param extensionLoaderClassLoaderFirst
     * @param overridden false
     * @param includedPackages
     * @param excludedPackages
     * @param onlyExtensionClassLoaderPackages
     */
```

```
private void loadDirectory(Map<String, Class<?>> extensionClasses,
String dir, String type,
                           boolean extensionLoaderClassLoaderFirst,
boolean overridden, String[] includedPackages,
                           String[] excludedPackages, String[]
onlyExtensionClassLoaderPackages) {
       //扩展目录 + 扩展类型全路径 比如: META-
INF/dubbo/internal/org.apache.dubbo.common.extension.ExtensionInjector
       String fileName = dir + type;
       try {
          List<ClassLoader> classLoadersToLoad = new LinkedList<>();

          // try to load from ExtensionLoader's ClassLoader first
          //是否优先使用扩展加载器的 类加载器
          if (extensionLoaderClassLoaderFirst) {
             ClassLoader extensionLoaderClassLoader =
ExtensionLoader.class.getClassLoader();
             if (ClassLoader.getSystemClassLoader() !=
extensionLoaderClassLoader) {
                classLoadersToLoad.add(extensionLoaderClassLoader);
             }
          }

          // load from scope model
          //获取域模型对象的类型加载器 ,这个域模型对象在初始化的时候会将自己的类加
载器放入集合中可以参考《3.2.2 初始化 ScopeModel》章节
          Set<ClassLoader> classLoaders =
scopeModel.getClassLoaders();

                        //没有可用的类加载器则从使用
          if (CollectionUtils.isEmpty(classLoaders)) {
          //从用于加载类的搜索路径中查找指定名称的所有资源。
             Enumeration<java.net.URL> resources =
ClassLoader.getSystemResources(fileName);
             if (resources != null) {
                while (resources.hasMoreElements()) {
                   loadResource(extensionClasses, null,
resources.nextElement(), overridden, includedPackages,
excludedPackages, onlyExtensionClassLoaderPackages);
                }
             }
          } else {
             classLoadersToLoad.addAll(classLoaders);
          }
```

```
                        //使用类加载资源加载器(ClassLoaderResourceLoader)来
加载具体的资源
          Map<ClassLoader, Set<java.net.URL>> resources =
ClassLoaderResourceLoader.loadResources(fileName, classLoadersToLoad);
         //遍历从所有资源文件中读取到资源 url 地址,key 为类加载器,值为扩展文件 url
如夏所示

//jar:file:/Users/song/.m2/repository/org/apache/dubbo/dubbo/3.0.7/dub
bo-3.0.7.jar!/META-
INF/dubbo/internal/org.apache.dubbo.common.extension.ExtensionInjector
          resources.forEach(((classLoader, urls) -> {
           //从文件中加载完资源之后开始根据类加载器和 url 加载具体的扩展类型,最后
将扩展存放进 extensionClasses 集合
              loadFromClass(extensionClasses, overridden, urls,
classLoader, includedPackages, excludedPackages,
onlyExtensionClassLoaderPackages);
          }));
       } catch (Throwable t) {
          logger.error("Exception occurred when loading extension
class (interface: " +
              type + ", description file: " + fileName + ").", t);
       }
    }
```

### 2) 借助类加载器的 getResources 方法遍历所有文件进行扩展文件的查询

查找扩展类型对应的扩展文件的 url 方法:ClassLoaderResourceLoader 类型的 loadResources 源码:

```
public static Map<ClassLoader, Set<URL>> loadResources(String fileName, List<ClassLoader>
classLoaders) {
    //
        Map<ClassLoader, Set<URL>> resources = new ConcurrentHashMap<>();
        //不同的类加载器之间使用不同的线程异步的方式进行扫描
        CountDownLatch countDownLatch = new CountDownLatch(classLoaders.size());
        for (ClassLoader classLoader : classLoaders) {
          //多线程扫描,这个是个newCachedThreadPool的类型的线程池
            GlobalResourcesRepository.getGlobalExecutorService().submit(() -> {
              //
                resources.put(classLoader, loadResources(fileName, classLoader));
                countDownLatch.countDown();
            });
        }
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return Collections.unmodifiableMap(new LinkedHashMap<>(resources));
    }
```

加载具体类加载器中的资源文件的 loadResources 方法

```
public static Set<URL> loadResources(String fileName, ClassLoader
currentClassLoader) {
       Map<ClassLoader, Map<String, Set<URL>>> classLoaderCache;
       //第一次进来类加载器资源缓存是空的
       if (classLoaderResourcesCache == null || (classLoaderCache =
classLoaderResourcesCache.get()) == null) {
        //类对象锁
           synchronized (ClassLoaderResourceLoader.class) {
               if (classLoaderResourcesCache == null ||
(classLoaderCache = classLoaderResourcesCache.get()) == null) {
                   classLoaderCache = new ConcurrentHashMap<>();
                   //创建一个类资源映射 url 的软引用缓存对象
                   //软引用(soft references)，用于帮助垃圾收集器管理内存使用和消
除潜在的内存泄漏。当内存快要不足的时候，GC 会迅速的把所有的软引用清除掉，释放内存空间
                   classLoaderResourcesCache = new
SoftReference<>(classLoaderCache);
               }
           }
       }
       //第一次进来时候类加载器 url 映射缓存是空的,给类加载器缓存对象新增一个值,key
是类加载器,值是 map 类型用来存储文件名对应的 url 集合
       if (!classLoaderCache.containsKey(currentClassLoader)) {
           classLoaderCache.putIfAbsent(currentClassLoader, new
ConcurrentHashMap<>());
       }
```

```
        Map<String, Set<URL>> urlCache =
classLoaderCache.get(currentClassLoader);
        //缓存中没有就从文件里面找
        if (!urlCache.containsKey(fileName)) {
            Set<URL> set = new LinkedHashSet<>();
            Enumeration<URL> urls;
            try {
                  //getResources 这个方法是这样的:加载当前类加载器以及父类加载器所
在路径的资源文件,将遇到的所有资源文件全部返回！这个可以理解为使用双亲委派模型中的类
加载器 加载各个位置的资源文件
                urls = currentClassLoader.getResources(fileName);
                //native 配置 是否为本地镜像(k 可以参考官方文
档:https://dubbo.apache.org/zh/docs/references/graalvm/support-graalvm/
                boolean isNative = NativeUtils.isNative();
                if (urls != null) {
                //遍历找到的对应扩展的文件 url 将其加入集合
                    while (urls.hasMoreElements()) {
                        URL url = urls.nextElement();
                        if (isNative) {
                            //In native mode, the address of each URL is
the same instead of different paths, so it is necessary to set the ref
to make it different
                            //动态修改 jdk 底层 url 对象的 ref 变量为可访问,让我们在
用反射时访问私有变量
                            setRef(url);
                        }
                        set.add(url);
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            //存入缓存
            urlCache.put(fileName, set);
        }
        //返回结果
        return urlCache.get(fileName);
    }
```

### 3) 使用找到的扩展资源 url 加载具体扩展类型到内存

ExtensionLoader 类型中的 loadFromClass 方法遍历 url 开始加载扩展类型

```
 private void loadFromClass(Map<String, Class<?>> extensionClasses, boolean overridden,
Set<java.net.URL> urls, ClassLoader classLoader,
                           String[] includedPackages, String[] excludedPackages,
String[] onlyExtensionClassLoaderPackages) {
        if (CollectionUtils.isNotEmpty(urls)) {
            for (java.net.URL url : urls) {
                loadResource(extensionClasses, classLoader, url, overridden,
includedPackages, excludedPackages, onlyExtensionClassLoaderPackages);
            }
        }
    }
```

ExtensionLoader 类型中的 loadResource 方法 使用 IO 流读取扩展文件的内容

读取内容之前我这里先贴一下我们参考的扩展注入类型的文件中的内容如下所示：

```
adaptive=org.apache.dubbo.common.extension.inject.AdaptiveExtensionInjector
spi=org.apache.dubbo.common.extension.inject.SpiExtensionInjector
scopeBean=org.apache.dubbo.common.beans.ScopeBeanExtensionInjector
```

扩展中的文件都是一行一行的，并且扩展名字和扩展类型之间使用等号隔开=
了解了文件内容之后，应该下面的代码大致思路就知道了，我们可以详细看下。

```
private void loadResource(Map<String, Class<?>> extensionClasses,
ClassLoader classLoader,
                          java.net.URL resourceURL, boolean overridden,
String[] includedPackages, String[] excludedPackages, String[]
onlyExtensionClassLoaderPackages) {
       try {
          //这里固定了文件的格式为 utf8
           try (BufferedReader reader = new BufferedReader(new
InputStreamReader(resourceURL.openStream(), StandardCharsets.UTF_8)))
{
               String line;
               String clazz;
               //按行读取 例如读取到的内
容:spring=org.apache.dubbo.config.spring.extension.SpringExtensionInjec
tor
               while ((line = reader.readLine()) != null) {
               //不知道为何会有这么一行代码删除#之后的字符串
                   final int ci = line.indexOf('#');
                   if (ci >= 0) {
```

```
                    line = line.substring(0, ci);
                }
                line = line.trim();
                //
                if (line.length() > 0) {
                    try {
                        String name = null;
                        //扩展文件可能如上面我贴的那样 名字和类型等号隔开,也可能是无类型的,例如扩展加载策略使用的是 JDK 自带的方式 services 内容中只包含具体的扩展类型
                        int i = line.indexOf('=');
                        if (i > 0) {
                            name = line.substring(0, i).trim();
                            clazz = line.substring(i + 1).trim();
                        } else {
                            clazz = line;
                        }
                        //isExcluded 是否为加载策略要排除的配置,参数这里为空代表全部类型不排除
                        //isIncluded 是否为加载策略包含的类型,参数这里为空代表全部文件皆可包含
                        //onlyExtensionClassLoaderPackages 参数是否只有扩展类的类加载器可以加载扩展,其他扩展类型的类加载器不能加载扩展 这里结果为 false 不排除任何类加载器
                        if (StringUtils.isNotEmpty(clazz)
&& !isExcluded(clazz, excludedPackages) && isIncluded(clazz,
includedPackages)
                            && !isExcludedByClassLoader(clazz,
classLoader, onlyExtensionClassLoaderPackages)) {
                            //根据类全路径加载类到内存
                            loadClass(extensionClasses, resourceURL,
Class.forName(clazz, true, classLoader), name, overridden);
                        }
                    } catch (Throwable t) {
                        IllegalStateException e = new
IllegalStateException("Failed to load extension class (interface: " +
type +
                            ", class line: " + line + ") in " +
resourceURL + ", cause: " + t.getMessage(), t);
                        exceptions.put(line, e);
                    }
                }
            }
        }
    } catch (Throwable t) {
```

```
            logger.error("Exception occurred when loading extension
class (interface: " +
                    type + ", class file: " + resourceURL + ") in " +
resourceURL, t);
        }
    }
```

ExtensionLoader 类型中的 loadClass 方法加载具体的类到内存

```
private void loadClass(Map<String, Class<?>> extensionClasses,
java.net.URL resourceURL, Class<?> clazz, String name,
                       boolean overridden) throws NoSuchMethodException
{
          //当前 clazz 是否为 type 的子类型
          //这里第一次访问到的 type 是 ExtensionInjector,clazz 是
SpringExtensionInjector 父子类型关系满足情况
        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.");
        }
        //扩展子类型是否存在这个注解@Adaptive
        if (clazz.isAnnotationPresent(Adaptive.class)) {
            cacheAdaptiveClass(clazz, overridden);
        } else if (isWrapperClass(clazz)) {
        //扩展子类型构造器中是否有这个类型的接口（这个可以想象下我们了解的 Java IO
流中的类型使用到的装饰器模式 构造器传个类型）
            cacheWrapperClass(clazz);
        } else {
        //无自适应注解,也没有构造器是扩展类型参数 ,这个 name 我们在扩展文件中找到了
就是等号前面那个
            if (StringUtils.isEmpty(name)) {
            //低版本中可以使用@Extension 扩展注解来标注扩展类型,这里获取注解有两个
渠道:
            //先查询@Extension 注解是否存在如果存在则取 value 值,如果不存在
@Extension 注解则获取当前类型的名字
                name = findAnnotationName(clazz);
                if (name.length() == 0) {
                    throw new IllegalStateException("No such extension
name for the class " + clazz.getName() + " in the config " +
resourceURL);
                }
            }
```

```
                //获取扩展名字数组,扩展名字可能为逗号隔开的
            String[] names = NAME_SEPARATOR.split(name);
            if (ArrayUtils.isNotEmpty(names)) {
              //@Activate 注解修饰的扩展
                cacheActivateClass(clazz, names[0]);
                for (String n : names) {
                  //cachedNames 缓存集合缓存当前扩展类型的扩展名字
                    cacheName(clazz, n);
                    //将扩展类型加入结果集合 extensionClasses 中,不允许覆盖的话出现同同名字扩展将抛出异常
                    saveInExtensionClass(extensionClasses, clazz, n, overridden);
                }
            }
        }
    }
```

### ExtensionLoader 类型中 cacheAdaptiveClass

Adaptive 机制，即扩展类的自适应机制。即其可以指定想要加载的扩展名，也可以不指定。若不指定，则直接加载默认的扩展类。即其会自动匹配，做到自适应。其是通过@Adaptive 注解实现的。

自适应注解修饰的扩展同一个扩展名字只能有一个扩展实现类型，扩展策略中提供的参数 overridden 是否允许覆盖扩展覆盖。

```
private void cacheAdaptiveClass(Class<?> clazz, boolean overridden) {
        if (cachedAdaptiveClass == null || overridden) {
           //成员变量存储这个自适应扩展类型
            cachedAdaptiveClass = clazz;
        } else if (!cachedAdaptiveClass.equals(clazz)) {
            throw new IllegalStateException("More than 1 adaptive class found: "
                + cachedAdaptiveClass.getName()
                + ", " + clazz.getName());
        }
    }
```

### ExtensionLoader 类型中 cacheWrapperClass

Wrapper 机制，即扩展类的包装机制。就是对扩展类中的 SPI 接口方法进行增强，进行包装，是 AOP 思想的体现，是 Wrapper 设计模式的应用。一个 SPI 可以包含多个 Wrapper。这个也是可以同一个类型多个。

```
private void cacheWrapperClass(Class<?> clazz) {
        if (cachedWrapperClasses == null) {
            cachedWrapperClasses = new ConcurrentHashSet<>();
        }
        //缓存这个Wrapper类型的扩展
        cachedWrapperClasses.add(clazz);
    }
```

### ExtensionLoader 类型中 cacheActivateClass

Activate 用于激活扩展类的。这个扩展类型可以出现多个比如过滤器可以同一个扩展名字多个过滤器实现，所以不需要有 override 判断。

Activate 机制，即扩展类的激活机制。通过指定的条件来激活当前的扩展类。其是通过@Activate 注解实现的。

```
private void cacheActivateClass(Class<?> clazz, String name) {
        Activate activate = clazz.getAnnotation(Activate.class);
        if (activate != null) {
          //缓存Activate类型的扩展
            cachedActivates.put(name, activate);
        } else {
            // support com.alibaba.dubbo.common.extension.Activate
            com.alibaba.dubbo.common.extension.Activate oldActivate =
clazz.getAnnotation(com.alibaba.dubbo.common.extension.Activate.class);
            if (oldActivate != null) {
                cachedActivates.put(name, oldActivate);
            }
        }
    }
```

### ExtensionLoader 类型中的 saveInExtensionClass 方法

上面扩展对象加载了这么多最终的目的就是将这个扩展类型存放进结果集合 extensionClasses 中，扩展策略中提供的参数 overridden 是否允许覆盖扩展覆盖。

```
private void saveInExtensionClass(Map<String, Class<?>> extensionClasses, Class<?> clazz,
String name, boolean overridden) {
        Class<?> c = extensionClasses.get(name);
        if (c == null || overridden) {
        //上面扩展对象加载了这么多最终的目的就是将这个扩展类型存放进结果集合中
            extensionClasses.put(name, clazz);
        } else if (c != clazz) {
            // duplicate implementation is unacceptable
            unacceptableExceptions.add(name);
            String duplicateMsg = "Duplicate extension " + type.getName() + " name " + name
+ " on " + c.getName() + " and " + clazz.getName();
            logger.error(duplicateMsg);
            throw new IllegalStateException(duplicateMsg);
        }
    }
```

## 3. 自适应扩展代理对象的代码生成与编译

### 1) 自适应扩展对象的创建

Dubbo 的自适应扩展机制中如果自己生成了自适应扩展的代理类。

Dubbo 的自适应扩展为了做什么：在运行时动态调用扩展方法。以及怎么做的：生成扩展代理类。比如：代理类中根据 URL 获取扩展名，使用 SPI 加载扩展类，并调用同名方法，返回执行结果。

看了上一个章节，我们了解到了 Dubbo 是如何通过扫描目录来查询扩展实现类的这一次我们看下扩展类我们找到了之后，如果这个扩展类型未加上这个@Adaptive 注解那么是如何创建这个类型的，接下来看 createAdaptiveExtensionClass 方法，这个方法是借助字节码工具来动态生成所需要的扩展类型的包装类型的代码，这个代码在编译时我们可能看不到，但是在 Debug 的时候，我们还是可以看到这个对象名字的，但是往往 Debug 的时候又进不到具体的代码位置，这里可以注意下。

当扩展点的方法被@Adaptive 修饰时，在 Dubbo 初始化扩展点时会自动生成和编译一个动态的 Adaptive 类。

下面我们可以以 interface org.apache.dubbo.rpc.Protocol 这个协议扩展类型来看，协议扩展类型目前没有一个是带有自适应注解的

```
private Class<?> createAdaptiveExtensionClass() {
        // Adaptive Classes' ClassLoader should be the same with Real SPI interface classes'
ClassLoader
        //获取加载器
        ClassLoader classLoader = type.getClassLoader();
        try {
          // //native配置 是否为本地镜像(可以参考官方文
档:https://dubbo.apache.org/zh/docs/references/graalvm/support-graalv
            if (NativeUtils.isNative()) {
                return classLoader.loadClass(type.getName() + "$Adaptive");
            }
        } catch (Throwable ignore) {

        }
        //创建一个代码生成器,来生成代码 详细内容我们就下一章来看
        String code = new AdaptiveClassCodeGenerator(type, cachedDefaultName).generate();
        //获取编译器
        org.apache.dubbo.common.compiler.Compiler compiler =
extensionDirector.getExtensionLoader(
    org.apache.dubbo.common.compiler.Compiler.class).getAdaptiveExtension();
            //生成的代码进行编译
        return compiler.compile(type, code, classLoader);
    }
```

## 4. 为扩展对象的 set 方法注入自适应扩展对象

在 4.4.5 小节中我们已经讲解了获取扩展类型实现类，创建扩展对象。

```
T instance = (T) getAdaptiveExtensionClass().newInstance();
```

接下来就让我们来看下为扩展对象的 set 方法注入自适应的扩展对象。

调用方法代码如下：

```
   //注入扩展对象之前的回调方法
injectExtension(instance);
```

ExtensionLoader 类型的 injectExtension 方法具体代码如下：

```
private T injectExtension(T instance) {
                //如果注入器为空则直接返回当前对象
       if (injector == null) {
```

```
            return instance;
        }

        try {
         //获取当前对象的当前类的所有方法
            for (Method method : instance.getClass().getMethods()) {
                 //是否为 set 方法 不是的话则跳过,在这里合法的 set 方法满足 3 个条
件:
                 //set 开头,参数只有一个,public 修饰
                if (!isSetter(method)) {
                    continue;
                }
                /**
                 * Check {@link DisableInject} to see if we need auto
injection for this property
                 */
                 //方法上面是否有注解 DisableInject 修饰,这种情况也直接跳过
                if (method.isAnnotationPresent(DisableInject.class)) {
                    continue;
                }
                //方法的参数如果是原生类型也跳过
                Class<?> pt = method.getParameterTypes()[0];
                if (ReflectUtils.isPrimitives(pt)) {
                    continue;
                }
                try {
                  //获取 set 方法对应的成员变量如 setProtocol 属性为 protocol
                    String property = getSetterProperty(method);
                    //根据参数类型如 Protocol 和属性名字如 protocol 获取应该注入的
对象
                    Object object = injector.getInstance(pt, property);
                    if (object != null) {
                           //执行对应对象和对应参数的这个方法
                        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) 获取注入对象

这里我们主要来看下如何通过注入器找到需要注入的那个对象，调用代码如下：

```
Object object = injector.getInstance(pt, property);
```

在前面看注入器扩展对象的获取的时候是会获取到 ExtensionInjector 扩展的一个自适应扩展注入器实现类型，AdaptiveExtensionInjector，这个地方对应的 getInstance 也是这个扩展里面的，我们来看下它的方法：

```
@Override
    public <T> T getInstance(Class<T> type, String name) {
      //遍历所有的扩展注入器
        for (ExtensionInjector injector : injectors) {
          //遍历所有的扩展注入器,如果可以获取到扩展对象则直接返回
            T extension = injector.getInstance(type, name);
            if (extension != null) {
                return extension;
            }
        }
        return null;
    }
```

可以看到上面代码按扩展注入器顺序来遍历的第一个找到的对象就直接返回了，这个 AdaptiveExtensionInjector 在初始化的时候会获取所有的 ExtensionInjector 的扩展，非自适应的，它本身自适应的扩展，这里会获取非自适应的扩展列表一共有 3 个按顺序为：

- ScopeBeanExtensionInjector
- SpiExtensionInjector
- SpringExtensionInjector

接下来我们详细看下每种扩展注入器加载扩展对象的策略。

### 2) 域模型中的 Bean 扩展注入器 ScopeBeanExtensionInjector

#### ScopeBeanExtensionInjector 的 getInstance 方法

每个域模型都会有个 ScopeBeanFactory 类型的对象用于存储共享对象，并且域模型之间按照层级子类型的 Bean 工厂可以从父域的 Bean 工厂中查询对象。

```
@Override
    public <T> T getInstance(Class<T> type, String name) {
        return beanFactory.getBean(name, type);
    }
```

#### ScopeBeanFactory 的 getBean 方法

先从当前域空间查询对象，如果找不到对应类型的扩展对象则从父域工厂查询扩展对象。

```
public <T> T getBean(String name, Class<T> type) {
    //当前域下注册的扩展对象
        T bean = getBeanInternal(name, type);
        if (bean == null && parent != null) {
          //父域中查找扩展对象
            return parent.getBean(name, type);
        }
        return bean;
    }
```

#### ScopeBeanFactory 的 getBeanInternal 方法

从当前域下找注册的参数类型的对象

```
private <T> T getBeanInternal(String name, Class<T> type) {
       checkDestroyed();
       // All classes are derived from java.lang.Object, cannot filter
bean by it
       if (type == Object.class) {
           return null;
       }
       List<BeanInfo> candidates = null;
       BeanInfo firstCandidate = null;
```

```
        //遍历列表查询
        for (BeanInfo beanInfo : registeredBeanInfos) {
            // if required bean type is same class/superclass/interface
of the registered bean
            if (type.isAssignableFrom(beanInfo.instance.getClass())) {
                if (StringUtils.isEquals(beanInfo.name, name)) {
                    return (T) beanInfo.instance;
                } else {
                    // optimize for only one matched bean
                    if (firstCandidate == null) {
                        firstCandidate = beanInfo;
                    } else {
                        if (candidates == null) {
                            candidates = new ArrayList<>();
                            candidates.add(firstCandidate);
                        }
                        candidates.add(beanInfo);
                    }
                }
            }
        }

        // if bean name not matched and only single candidate
        if (candidates != null) {
            if (candidates.size() == 1) {
                return (T) candidates.get(0).instance;
            } else if (candidates.size() > 1) {
                List<String> candidateBeanNames =
candidates.stream().map(beanInfo ->
beanInfo.name).collect(Collectors.toList());
                throw new ScopeBeanException("expected single matching
bean but found " + candidates.size() + " candidates for type [" +
type.getName() + "]: " + candidateBeanNames);
            }
        } else if (firstCandidate != null) {
            return (T) firstCandidate.instance;
        }
        return null;
    }
```

### 3) SPI 扩展机制注入器 SpiExtensionInjector

SPI 是 Dubbo 自行实现的一套扩展机制，我们来看下它是如何查找扩展对象的

```
@Override
    public <T> T getInstance(Class<T> type, String name) {
      //如果是一个标准的被@SPI注解修饰的扩展接口则满足条件
        if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {
        //使用扩展访问器来获取对应类型的扩展加载器
            ExtensionLoader<T> loader = extensionAccessor.getExtensionLoader(type);
            if (loader == null) {
                return null;
            }
            //使用对应类型的扩展加载器来加载自适应扩展 这个加载的扩展可以参考4.4.6小节
            if (!loader.getSupportedExtensions().isEmpty()) {
                return loader.getAdaptiveExtension();
            }
        }
        return null;
    }
```

### 4) Spring 扩展注入器

### SpringExtensionInjector

Spring 扩展注入器主要是用来从 Spring 容器中查询当前类型的 Bean 是否存在的，如下代码直接看代码吧。

```
@Override
    @SuppressWarnings("unchecked")
    public <T> T getInstance(Class<T> type, String name) {

        if (context == null) {
            // ignore if spring context is not bound
            return null;
        }

        //check @SPI annotation ,类型需要满足SPI机制 @SPI修饰的接口
        if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {
            return null;
        }
    //从Spring容器中查询Bean
        T bean = getOptionalBean(context, name, type);
        if (bean != null) {
            return bean;
        }

        //logger.warn("No spring extension (bean) named:" + name + ", try to find an
extension (bean) of type " + type.getName());
        return null;
    }
```

```
private <T> T getOptionalBean(ListableBeanFactory beanFactory, String name, Class<T> type) {
    //要搜索的扩展名字为空就根据类型搜索
        if (StringUtils.isEmpty(name)) {
        //返回与给定类型（包括子类）匹配的bean的名称，对于FactoryBeans
            String[] beanNamesForType = beanFactory.getBeanNamesForType(type, true, false);
            if (beanNamesForType != null) {
                if (beanNamesForType.length == 1) {
                //返回指定bean的实例，该实例可以是共享的，也可以是独立的。
                //根据Bean Name和类型 查询具体的扩展对象
                    return beanFactory.getBean(beanNamesForType[0], type);
                } else if (beanNamesForType.length > 1) {
                    throw new IllegalStateException("Expect single but found " +
beanNamesForType.length + " beans in spring context: " +
                        Arrays.toString(beanNamesForType));
                }
            }
        } else {
          //扩展名字不为空则直接通过名字搜索Bean
            if (beanFactory.containsBean(name)) {
                return beanFactory.getBean(name, type);
            }
        }
        return null;
    }
```

# 三、 普通扩展对象的创建与 Wrapper 机制

## 1. 普通扩展对象的加载与创建

这里我们要分析的是 ExtensionLoader 类型的 getExtension（String name）方法，有了前面自适应扩展的铺垫，这里就更容易来看了 getExtension 是根据扩展名字获取具体扩展的通用方法，我们来根据某个类型来获取扩展的时候就是走的这里，比如在这个博客开头的介绍：

### ApplicationModel 中获取配置管理器对象

```
configManager = (ConfigManager) this.getExtensionLoader(ApplicationExt.class)
                .getExtension(ConfigManager.NAME);
```

### 1） getExtension 方法源码

先来看下 getExtension 方法的源码，根据扩展名字查询扩展对象

```
public T getExtension(String name) {
    //这里并不能看到什么,只多传了个参数wrap为true调用另外一个重载的方法
        T extension = getExtension(name, true);
        if (extension == null) {
            throw new IllegalArgumentException("Not find extension: " + name);
        }
        return extension;
    }
```

```
public T getExtension(String name, boolean wrap) {
    //检查扩展加载器是否已被销毁
        checkDestroyed();
        if (StringUtils.isEmpty(name)) {
            throw new IllegalArgumentException("Extension name == null");
        }
        //扩展名字为true则加载默认扩展
        if ("true".equals(name)) {
            return getDefaultExtension();
        }
        //非wrap类型则将缓存的扩展名字key加上_origin后缀
        //wrap是aop机制 俗称切面,这个origin在aop里面可以称为切点,下面的wrap扩展可以称为增强通知的类型,普
通扩展和wrap扩展的扩展名字是一样的
        String cacheKey = name;
        if (!wrap) {
            cacheKey += "_origin";
        }
        //从cachedInstances缓存中查询
        final Holder<Object> holder = getOrCreateHolder(cacheKey);
        Object instance = holder.get();
        //缓存中不存在则创建扩展对象 双重校验锁
        if (instance == null) {
            synchronized (holder) {
              //双重校验锁的方式
                instance = holder.get();
                if (instance == null) {
                  //创建扩展对象
                    instance = createExtension(name, wrap);
                    holder.set(instance);
                }
            }
        }
        return (T) instance;
    }
```

我们先来看一下默认扩展的加载代码：

```
public T getDefaultExtension() {
    //加载扩展类型对应的所有扩展SPI实现类型,在加载所有扩展实现类型的时候会缓存这个扩展的默认实现类型,将对象缓存在cachedDefaultName中
        getExtensionClasses();
        if (StringUtils.isBlank(cachedDefaultName) || "true".equals(cachedDefaultName)) {
            return null;
        }
        //再回到加载扩展的方法
        return getExtension(cachedDefaultName);
    }
```

创建扩展对象方法这个和自适应扩展的创建扩展类似 createExtension。

具体过程如下:

- 加载扩展类型：getExtensionClasses()
- 创建扩展对象：createExtensionInstance（clazz）
- 注入自适应扩展：injectExtension（instance）
- wrap 处理

```
private T createExtension(String name, boolean wrap) {
                //扩展的创建的第一步扫描所有 jar 中的扩展实现,这里扫描完之后获取对应扩展名字的扩展实现类型的 Class 对象
        Class<?> clazz = getExtensionClasses().get(name);
        //出现异常了 转换下异常信息 再抛出
        if (clazz == null || unacceptableExceptions.contains(name)) {
            throw findException(name);
        }
        try {
         //当前扩展对象是否已经创建过了则直接从缓存中获取
            T instance = (T) extensionInstances.get(clazz);
            if (instance == null) {
                  //第一次获取缓存中肯定没有则创建扩展对象然后缓存起来
                  //createExtensionInstance 这个是与自适应扩展对象创建对象的不同之处
                extensionInstances.putIfAbsent(clazz, createExtensionInstance(clazz));
                instance = (T) extensionInstances.get(clazz);
                instance = postProcessBeforeInitialization(instance, name);
                //注入扩展自适应方法,这个方法前面讲自适应扩展时候说了,注入自适应扩展方法的自适应扩展对象
                injectExtension(instance);
```

```
                instance = postProcessAfterInitialization(instance,
name);
            }
                        //是否开启了 wrap
                        //Dubbo 通过 Wrapper 实现 AOP 的方法
            if (wrap) {
            //这个可以参考下 Dubbo 扩展的加载
                List<Class<?>> wrapperClassesList = new ArrayList<>();
                //wrap 类型排序 这个 wrap 类型是如何来的呢,在前面扫描扩展类型的时候
如果当前扩展类型不是 Adaptive 注解修饰的,并且当前类型 type 有个构造器参数是 type 自
身的也是前面加载扩展类型时候说的装饰器模式 可以参考 DubboProtocol 的构造器
                if (cachedWrapperClasses != null) {
                    wrapperClassesList.addAll(cachedWrapperClasses);
                    //根据 Wrapper 注解的 order 值来进行排序值越小越在列表的前面
                    wrapperClassesList.sort(WrapperComparator.COMPARATOR);
                    //反转之后值越大就会在列表的前面
                    Collections.reverse(wrapperClassesList);
                }
                                //从缓存中查到了 wrapper 扩展则遍历这些 wrapp
扩展进行筛选
                if (CollectionUtils.isNotEmpty(wrapperClassesList)) {
                    for (Class<?> wrapperClass : wrapperClassesList) {
                        Wrapper wrapper =
wrapperClass.getAnnotation(Wrapper.class);
                        //需要包装的扩展名。当此数组为空时，默认值为匹配
                        //看下当前扩展是否匹配这个 wrap,如何判断呢?
                        //wrapper 注解不存在或者 matches 匹配,或者 mismatches 不包
含当前扩展
                        //如果匹配到了当前扩展对象是需要进行 wrapp 的就为当前扩展创
建当前 wrapper 扩展对象进行包装
                        boolean match = (wrapper == null) ||
                            ((ArrayUtils.isEmpty(wrapper.matches()) ||
ArrayUtils.contains(wrapper.matches(), name)) &&
                                !ArrayUtils.contains(wrapper.mismatches(),
name));
                                //这是扩展类型是匹配 wrapp 的则开始注入
                        if (match) {
                        //匹配到了就创建所有的 wrapper 类型的对象同时构造器参数设置
为当前类型
                            instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
                            instance =
postProcessAfterInitialization(instance, name);
                        }
                    }
```

```
                }
            }

            // Warning: After an instance of Lifecycle is wrapped by
cachedWrapperClasses, it may not still be Lifecycle instance, this
application may not invoke the lifecycle.initialize hook.
            //初始化扩展,如果当前扩展是 Lifecycle 类型则调用初始化方法
            initExtension(instance);
            return instance;
        } catch (Throwable t) {
            throw new IllegalStateException("Extension instance (name: "
+ name + ", class: " +
                type + ") couldn't be instantiated: " + t.getMessage(),
t);
        }
    }
```

### 2) 创建扩展对象

前面加载扩展类型在自适应扩展的时候已经说过了这里就不重复了，这里我们来看下扩展对象的创建过程：createExtensionInstance（clazz）

前面看自适应扩展对象创建的时候自适应扩展对象仅仅是使用反射 newInstance 了一个扩展对象，而普通的扩展类型创建对象的过程就相对复杂一点，接下来我们来看下。

#### ExtensionLoader 的 createExtensionInstance 方法

```
private Object createExtensionInstance(Class<?> type) throws ReflectiveOperationException {
    //在ExtensionLoader构造器中,有个initInstantiationStrategy()方法中new了一个初始化策略
InstantiationStrategy类型对象
        return instantiationStrategy.instantiate(type);
    }
```

InstantiationStrategy 的实例化对象方法 instantiate

```
public <T> T instantiate(Class<T> type) throws
ReflectiveOperationException {
```

```
        // should not use default constructor directly, maybe also has
another constructor matched scope model arguments
        // 1. try to get default constructor
        Constructor<T> defaultConstructor = null;
        try {
         //反射获取对应类型的无参构造器
            defaultConstructor = type.getConstructor();
        } catch (NoSuchMethodException e) {
            // ignore no default constructor
        }

        // 2. use matched constructor if found
        List<Constructor> matchedConstructors = new ArrayList<>();
        //获取所有构造器
        Constructor<?>[] declaredConstructors = type.getConstructors();
        //遍历构造器列表,
        for (Constructor<?> constructor : declaredConstructors) {
        //如果存在构造器则构造器参数类型是否为 ScopeModel 类型,如果为 ScopeModel
则为匹配的构造器 说明我们扩展类型在这个版本如果想要让这个构造器生效必须参数类型为
ScopeModel
            if (isMatched(constructor)) {
                matchedConstructors.add(constructor);
            }
        }
        // remove default constructor from matchedConstructors
        if (defaultConstructor != null) {
            matchedConstructors.remove(defaultConstructor);
        }

        // match order:
        // 1. the only matched constructor with parameters
        // 2. default constructor if absent

        Constructor targetConstructor;
        //匹配的参数 ScopeModel 的构造器太多了就抛出异常
        if (matchedConstructors.size() > 1) {
            throw new IllegalArgumentException("Expect only one but
found " +
                matchedConstructors.size() + " matched constructors for
type: " + type.getName() +
                ", matched constructors: " + matchedConstructors);
        } else if (matchedConstructors.size() == 1) {
        //一个参数一般为一个参数类型 ScopeModel 的构造器
            targetConstructor = matchedConstructors.get(0);
        } else if (defaultConstructor != null) {
```

```
        //如果没有自定义构造器则使用空参数构造器
            targetConstructor = defaultConstructor;
        } else {
        //一个构造器也没匹配上也要报错
            throw new IllegalArgumentException("None matched constructor
was found for type: " + type.getName());
        }

        // create instance with arguments
        //反射获取构造器参数的参数类型列表
        Class[] parameterTypes = targetConstructor.getParameterTypes();
        //如果存在参数则为参数设置值
        Object[] args = new Object[parameterTypes.length];
        for (int i = 0; i < parameterTypes.length; i++) {
        //借助 scopeModelAccessor 工具获取参数类型,这个参数类型为当前的域模型对象
            args[i] = getArgumentValueForType(parameterTypes[i]);
        }
        //创建扩展对象
        return (T) targetConstructor.newInstance(args);
    }
```

## 2. wrap 机制

### 1) Wrapper 机制说明

Dubbo 通过 Wrapper 实现 AOP 的方法

Wrapper 机制，即扩展点自动包装。Wrapper 类同样实现了扩展点接口，但是 Wrapper 不是扩展点的真正实现。它的用途主要是用于从 ExtensionLoader 返回扩展点时，包装在真正的扩展点实现外。即从 ExtensionLoader 中返回的实际上是 Wrapper 类的实例，Wrapper 持有了实际的扩展点实现类。

扩展点的 Wrapper 类可以有多个，也可以根据需要新增。

通过 Wrapper 类可以把所有扩展点公共逻辑移至 Wrapper 中。新加的 Wrapper 在所有的扩展点上添加了逻辑，有些类似 AOP，即 Wrapper 代理了扩展点。

### Wrapper 的规范

Wrapper 机制不是通过注解实现的，而是通过一套 Wrapper 规范实现的。

Wrapper 类在定义时需要遵循如下规范。

- 该类要实现 SPI 接口
- 该类中要有 SPI 接口的引用
- 该类中必须含有一个含参的构造方法且参数只能有一个类型为 SPI 接口
- 在接口实现方法中要调用 SPI 接口引用对象的相应方法
- 该类名称以 Wrapper 结尾

比如如下几个扩展类型

```
class org.apache.dubbo.rpc.protocol.ProtocolListenerWrapper
class org.apache.dubbo.qos.protocol.QosProtocolWrapper
class org.apache.dubbo.rpc.protocol.ProtocolListenerWrapper
class org.apache.dubbo.qos.protocol.QosProtocolWrapper
```

回顾下 Wrapper 扩展类型的扫描于对象的创建。

### 2) Wrapper 类型的扫描

Wrapper 类型的扫描代码如下。

来自 4.5.2.3 小节 ExtensionLoader 类型中的 loadClass 方法

```
//扩展子类型是否存在这个注解@Adaptive
        if (clazz.isAnnotationPresent(Adaptive.class)) {
            cacheAdaptiveClass(clazz, overridden);
        } else if (isWrapperClass(clazz)) {
        //扩展子类型构造器中是否有这个类型的接口 (这个可以想象下我们了解的Java IO流中的类型使用到的装饰器模
式 构造器传个类型)
            cacheWrapperClass(clazz);
        } else {
```

isWrapperClass 方法通过判断构造器类型是否为当前类型来判断是否为 Wrapper 类型

```
private boolean isWrapperClass(Class<?> clazz) {
       try {
           clazz.getConstructor(type);
           return true;
       } catch (NoSuchMethodException e) {
           return false;
       }
   }
```

### 3) Wrapper 类型的创建

这个可以看下 4.6.1getExtension 方法源码的获取扩展对象时候查询扩展对象是否有对应的 Wrapper 类型的扩展为其创建 Wrapper 扩展对象，如下代码。

```
//Dubbo 通过 Wrapper 实现 AOP 的方法
           if (wrap) {
           //这个可以参考下 Dubbo 扩展的加载
               List<Class<?>> wrapperClassesList = new ArrayList<>();
               //wrap 类型排序 这个 wrap 类型是如何来的呢,在前面扫描扩展类型的时候如果当前扩展类型不是 Adaptive 注解修饰的,并且当前类型 type 有个构造器参数是 type 自身的也是前面加载扩展类型时候说的装饰器模式 可以参考 DubboProtocol 的构造器
               if (cachedWrapperClasses != null) {
                   wrapperClassesList.addAll(cachedWrapperClasses);
                   //根据 Wrapper 注解的 order 值来进行排序值越小越在列表的前面
                   wrapperClassesList.sort(WrapperComparator.COMPARATOR);
                   //反转之后值越大就会在列表的前面
                   Collections.reverse(wrapperClassesList);
               }
                                   //从缓存中查到了 wrapper 扩展则遍历这些 wrapp 扩展进行筛选
               if (CollectionUtils.isNotEmpty(wrapperClassesList)) {
                   for (Class<?> wrapperClass : wrapperClassesList) {
                       Wrapper wrapper = wrapperClass.getAnnotation(Wrapper.class);
                       //需要包装的扩展名。当此数组为空时，默认值为匹配
                       //看下当前扩展是否匹配这个 wrap,如何判断呢?
                       //wrapper 注解不存在或者 matches 匹配,或者 mismatches 不包含当前扩展
                       //如果匹配到了当前扩展对象是需要进行 wrapp 的就为当前扩展创建当前 wrapper 扩展对象进行包装
```

```
                    boolean match = (wrapper == null) ||
                        ((ArrayUtils.isEmpty(wrapper.matches()) ||
ArrayUtils.contains(wrapper.matches(), name)) &&
                            !ArrayUtils.contains(wrapper.mismatches(),
name));
                            //这是扩展类型是匹配 wrapp 的则开始注入
                    if (match) {
                    //匹配到了就创建所有的 wrapper 类型的对象同时构造器参数设置
为当前类型
                        instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
                        instance =
postProcessAfterInitialization(instance, name);
                    }
                }
            }
        }
```

主要来看下什么情况下才为当前扩展类型创建 Wrapper 包装类型：

- wrapper 注解不存在（前面判断过 Wrapper 类型是构造器满足条件的）
- 存在 Wrapper 注解
- matches 匹配
  或者 mismatches 不包含当前扩展

如果匹配到了当前扩展对象是需要进行 wrapp 的就为当前扩展创建当前 wrapper 扩展对象进行包装。

## 四、 自动激活扩展 Activate 注解

### 1. Activate 扩展的说明

此注解对于使用给定条件自动激活某些扩展非常有用，例如：@Activate 可用于在有多个实现时加载某些筛选器扩展。

- group()指定组条件。框架 SPI 定义了有效的组值。
- value()指定 URL 条件中的参数键。

SPI 提供程序可以调用 ExtensionLoader。getActivateExtension(URL、String、String)方法以查找具有给定条件的所有已激活扩展。

比如后面我们会说到的过滤器扩展对象的获取，如下通过调用 getActivateExtension 方法的代码：

```
 List<Filter> filters;
 filters = ScopeModelUtil.getExtensionLoader(Filter.class,
moduleModels.get(0)).getActivateExtension(url, key, group);
```

## 2. 获取自动激活扩展的源码

前面我们看了激活扩展是通过调用 getActivateExtension 方法来获取对象的，那接下来就来看下这个方法做了什么操作：

```
/**
* @param url 服务的url
* @param key  用于获取扩展点名称的url参数键 比如监听器:exporter.listener,过滤器:params-
filter,telnet处理器:telnet
*/
 public List<T> getActivateExtension(URL url, String key) {
        return getActivateExtension(url, key, null);
    }
```

继续调用重载的方法

```
/**
     *
     *
     * @param url    服务的url
     * @param key    用于获取扩展点名称的url参数键 比如监听器:exporter.listener,过滤器:params-
filter,telnet处理器:telnet
     * @param group group 用于筛选的分组,比如过滤器中使用此参数来区分消费者使用这个过滤器还是提供者使用这
个过滤器他们的group参数分表为consumer,provider
     * @return 已激活的扩展列表。
     */
 public List<T> getActivateExtension(URL url, String key, String group) {
    //从参数中获取url指定的值
        String value = url.getParameter(key);
        //调用下个重载的方法
        return getActivateExtension(url, StringUtils.isEmpty(value) ? null :
COMMA_SPLIT_PATTERN.split(value), group);
    }
```

上面的重载方法都是用来转换参数的，下面这个方法才是真正的逻辑

```
/**
     * 获取激活扩展.
     *
     * @param url    服务的 url
     * @param values 这个 value 是扩展点的名字 当指定了时候会使用指定的名字的扩展
     * @param group  group 用于筛选的分组,比如过滤器中使用此参数来区分消费者使用
这个过滤器还是提供者使用这个过滤器他们的 group 参数分表为 consumer,provider
     * @return 获取激活扩展.
     */
    public List<T> getActivateExtension(URL url, String[] values,
String group) {
        //检查扩展加载器是否被销毁了
        checkDestroyed();
        // solve the bug of using @SPI's wrapper method to report a null
pointer exception.
        //创建个有序的 Map 集合,用来对扩展进行排序
        Map<Class<?>, T> activateExtensionsMap = new
TreeMap<>(activateComparator);
        //初始化扩展名字,指定了扩展名字 values 不为空
        List<String> names = values == null ? new ArrayList<>(0) :
asList(values);
        //扩展名字使用 Set 集合进行去重
        Set<String> namesSet = new HashSet<>(names);
        //参数常量是 -default  扩展名字是否不包含默认的
        if (!namesSet.contains(REMOVE_VALUE_PREFIX + DEFAULT_KEY)) {
        //第一次进来肯定是没有缓存对象双重校验锁检查下
            if (cachedActivateGroups.size() == 0) {
                synchronized (cachedActivateGroups) {
                    // cache all extensions
                    if (cachedActivateGroups.size() == 0) {
                            //加载扩展类型对应的扩展类,这个具体细节参考源码或者
《[Dubbo3.0.7 源码解析系列]-5-Dubbo 的 SPI 扩展机制与自适应扩展对象的创建与扩展文
件的扫描源码解析》章节
                        getExtensionClasses();
                        for (Map.Entry<String, Object> entry :
cachedActivates.entrySet()) {
                            String name = entry.getKey();
                            Object activate = entry.getValue();

                            String[] activateGroup, activateValue;
```

```
                                                        //遍历所有的
activates 列表获取 group()和 value()值
                        if (activate instanceof Activate) {
                            activateGroup = ((Activate)
activate).group();
                            activateValue = ((Activate)
activate).value();
                        } else if (activate instanceof
com.alibaba.dubbo.common.extension.Activate) {
                            activateGroup =
((com.alibaba.dubbo.common.extension.Activate) activate).group();
                            activateValue =
((com.alibaba.dubbo.common.extension.Activate) activate).value();
                        } else {
                            continue;
                        }
                        //缓存分组值
                        cachedActivateGroups.put(name, new
HashSet<>(Arrays.asList(activateGroup)));
                        String[][] keyPairs = new
String[activateValue.length][];
                        //遍历指定的激活扩展的扩展名字列表
                        for (int i = 0; i < activateValue.length; i++)
{
                            if (activateValue[i].contains(":")) {
                                keyPairs[i] = new String[2];
                                String[] arr =
activateValue[i].split(":");
                                keyPairs[i][0] = arr[0];
                                keyPairs[i][1] = arr[1];
                            } else {
                                keyPairs[i] = new String[1];
                                keyPairs[i][0] = activateValue[i];
                            }
                        }
                        //缓存指定扩展信息
                        cachedActivateValues.put(name, keyPairs);
                    }
                }
            }
        }

        // traverse all cached extensions
        //遍历所有激活的扩展名字和扩展分组集合
        cachedActivateGroups.forEach((name, activateGroup) -> {
```

```
                //筛选当前扩展的扩展分组与激活扩展的扩展分组是否可以匹配
            if (isMatchGroup(group, activateGroup)
                //不能是指定的扩展名字
                && !namesSet.contains(name)
                //也不能是带有 -指定扩展名字
                && !namesSet.contains(REMOVE_VALUE_PREFIX + name)
                //如果在 Active 注解中配置了 value 则当指定的键出现在 URL 的参数
中时，激活当前扩展名。
                //如果未配置 value 属性则默认都是匹配的(cachedActivateValues
中不存在对应扩展名字的缓存的时候默认为 true)
                && isActive(cachedActivateValues.get(name), url)) {
                                        //缓存激活的扩展类型映射的扩展名字
                activateExtensionsMap.put(getExtensionClass(name),
getExtension(name));
            }
        });
    }

    if (namesSet.contains(DEFAULT_KEY)) {
        // will affect order
        // `ext1,default,ext2` means ext1 will happens before all of
the default extensions while ext2 will after them
        ArrayList<T> extensionsResult = new
ArrayList<>(activateExtensionsMap.size() + names.size());
        for (int i = 0; i < names.size(); i++) {
            String name = names.get(i);
            if (!name.startsWith(REMOVE_VALUE_PREFIX)
                && !namesSet.contains(REMOVE_VALUE_PREFIX + name)) {
                if (!DEFAULT_KEY.equals(name)) {
                    if (containsExtension(name)) {
                        extensionsResult.add(getExtension(name));
                    }
                } else {

extensionsResult.addAll(activateExtensionsMap.values());
                }
            }
        }
        return extensionsResult;
    } else {
        // add extensions, will be sorted by its order
        for (int i = 0; i < names.size(); i++) {
            String name = names.get(i);
            if (!name.startsWith(REMOVE_VALUE_PREFIX)
                && !namesSet.contains(REMOVE_VALUE_PREFIX + name)) {
```

```
                if (!DEFAULT_KEY.equals(name)) {
                    if (containsExtension(name)) {

activateExtensionsMap.put(getExtensionClass(name),
getExtension(name));
                    }
                }
            }
        }
        return new ArrayList<>(activateExtensionsMap.values());
    }
}
```

再来回顾下扫描扩展类型的时候，与激活扩展的相关扫描代码。

与激活注解关键的代码位置在这里 ExtensionLoader 的 loadClass 方法中
我来贴下 loadClass 方法核心的代码。

```
if (clazz.isAnnotationPresent(Adaptive.class)) {
            cacheAdaptiveClass(clazz, overridden);
        } else if (isWrapperClass(clazz)) {
            cacheWrapperClass(clazz);
        } else {
            if (StringUtils.isEmpty(name)) {
                name = findAnnotationName(clazz);
                if (name.length() == 0) {
                    throw new IllegalStateException("No such extension name for the class "
+ clazz.getName() + " in the config " + resourceURL);
                }
            }

            String[] names = NAME_SEPARATOR.split(name);
            if (ArrayUtils.isNotEmpty(names)) {
            //位置在这里其他地方就不标记注释了,前面判断了如果不是Adaptive也不是Wrapper类型则我们可以来判
断是否为Activate 类型如果是的话调用cacheActivateClass方法将扩展缓存进cachedActivates缓存中
                cacheActivateClass(clazz, names[0]);
                for (String n : names) {
                    cacheName(clazz, n);
                    saveInExtensionClass(extensionClasses, clazz, n, overridden);
                }
            }
        }
```

```
private void cacheActivateClass(Class<?> clazz, String name) {
        Activate activate = clazz.getAnnotation(Activate.class);
        if (activate != null) {
          //注解存在则加入激活注解缓存
            cachedActivates.put(name, activate);
        } else {
            // support com.alibaba.dubbo.common.extension.Activate
            com.alibaba.dubbo.common.extension.Activate oldActivate =
clazz.getAnnotation(com.alibaba.dubbo.common.extension.Activate.class);
            if (oldActivate != null) {
                cachedActivates.put(name, oldActivate);
            }
        }
    }
```

# Bootstrap 启动过程

## 一、 Dubbo 配置体系及工作原理

以下是一个 Dubbo 属性配置的例子：

```
  ## application.properties

  # Spring boot application
  spring.application.name=dubbo-externalized-configuration-provider-
sample

  # Base packages to scan Dubbo Component:
@com.alibaba.dubbo.config.annotation.Service
  dubbo.scan.base-packages=com.alibaba.boot.dubbo.demo.provider.service

  # Dubbo Application
  ## The default value of dubbo.application.name is
${spring.application.name}
  ## dubbo.application.name=${spring.application.name}

  # Dubbo Protocol
  dubbo.protocol.name=dubbo
  dubbo.protocol.port=12345

  ## Dubbo Registry
  dubbo.registry.address=N/A

  ## service default version
  dubbo.provider.version=1.0.0
```

接下来，我们就围绕这个示例，分别从配置格式、配置来源、加载流程三个方面对 Dubbo 配置的工作原理进行分析。

## 1. 配置格式

目前 Dubbo 支持的所有配置都是.properties 格式的，包括-D、Externalized Configuration 等，.properties 中的所有配置项遵循一种 path-based 的配置格式。

在 Spring 应用中也可以将属性配置放到 application.yml 中，其树层次结构的方式可读性更好一些。

```
# 应用级配置（无 id）
dubbo.{config-type}.{config-item}={config-item-value}

# 实例级配置（指定 id 或 name）
dubbo.{config-type}s.{config-id}.{config-item}={config-item-value}
dubbo.{config-type}s.{config-name}.{config-item}={config-item-value}

# 服务接口配置
dubbo.service.{interface-name}.{config-item}={config-item-value}
dubbo.reference.{interface-name}.{config-item}={config-item-value}

# 方法配置
dubbo.service.{interface-name}.{method-name}.{config-item}={config-
item-value}
dubbo.reference.{interface-name}.{method-name}.{config-item}={config-
item-value}

# 方法 argument 配置
dubbo.reference.{interface-name}.{method-name}.{argument-
index}.{config-item}={config-item-value}
```

### 1) 应用级配置（无 id）

应用级配置的格式为：配置类型单数前缀，无 id/name。

```
# 应用级配置（无 id）
dubbo.{config-type}.{config-item}={config-item-value}
```

类似 application、monitor、metrics 等都属于应用级别组件，因此仅允许配置单个实例；而 protocol、registry 等允许配置多个的组件，在仅需要进行单例配置时，可采用此节描述的格式。常见示例如下：

```
dubbo.application.name=demo-provider
dubbo.application.qos-enable=false

dubbo.registry.address=zookeeper://127.0.0.1:2181

dubbo.protocol.name=dubbo
dubbo.protocol.port=-1
```

**2) 实例级配置（指定 id 或 name）**

针对某个实例的属性配置需要指定 id 或者 name，其前缀格式为：配置类型复数前缀+id/name。适用于 protocol、registry 等支持多例配置的组件。

```
# 实例级配置（指定 id 或 name）
dubbo.{config-type}s.{config-id}.{config-item}={config-item-value}
dubbo.{config-type}s.{config-name}.{config-item}={config-item-value}
```

- 如果不存在该 id 或者 name 的实例，则框架会基于这里列出来的属性创建配置组件实例。

- 如果已存在相同 id 或 name 的实例，则框架会将这里的列出的属性作为已有实例配置的补充，详细请参考属性覆盖。

- 具体的配置复数形式请参考单复数配置对照表。

### 3) 配置示例

```
dubbo.registries.unit1.address=zookeeper://127.0.0.1:2181
dubbo.registries.unit2.address=zookeeper://127.0.0.1:2182

dubbo.protocols.dubbo.name=dubbo
dubbo.protocols.dubbo.port=20880

dubbo.protocols.hessian.name=hessian
dubbo.protocols.hessian.port=8089
```

服务接口配置

```
dubbo.service.org.apache.dubbo.samples.api.DemoService.timeout=5000
dubbo.reference.org.apache.dubbo.samples.api.DemoService.timeout=6000
```

**方法配置**

方法配置格式:

```
# 方法配置
dubbo.service.{interface-name}.{method-name}.{config-item}={config-
item-value}
dubbo.reference.{interface-name}.{method-name}.{config-item}={config-
item-value}

# 方法 argument 配置
dubbo.reference.{interface-name}.{method-name}.{argument-
index}.{config-item}={config-item-value}
```

方法配置示例:

```
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.time
out=7000
```

```
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.onin
voke=notifyService.onInvoke
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.onre
turn=notifyService.onReturn
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.onth
row=notifyService.onThrow
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.0.ca
llback=true
```

等价于 XML 配置：

```
<dubbo:reference
interface="org.apache.dubbo.samples.api.DemoService" >
   <dubbo:method name="sayHello" timeout="7000"
oninvoke="notifyService.onInvoke"
               onreturn="notifyService.onReturn"
onthrow="notifyService.onThrow">
       <dubbo:argument index="0" callback="true" />
   </dubbo:method>
</dubbo:reference>
```

#### 4) 参数配置

parameters 参数为 map 对象，支持 xxx.parameters=[{key:value},{key:value}]方式进行配置。

```
dubbo.application.parameters=[{item1:value1},{item2:value2}]
dubbo.reference.org.apache.dubbo.samples.api.DemoService.parameters=[{
item3:value3}]
```

#### 5) 传输层配置

triple 协议采用 Http2 做底层通信协议，允许使用者自定义 Http2 的 6 个 settings 参数

配置格式如下：

```
# 通知对端 header 压缩索引表的上限个数
dubbo.rpc.tri.header-table-size=4096

# 启用服务端推送功能
dubbo.rpc.tri.enable-push=false

# 通知对端允许的最大并发流数
dubbo.rpc.tri.max-concurrent-streams=2147483647

# 声明发送端的窗口大小
dubbo.rpc.tri.initial-window-size=1048576

# 设置帧的最大字节数
dubbo.rpc.tri.max-frame-size=32768

# 通知对端 header 未压缩的最大字节数
dubbo.rpc.tri.max-header-list-size=8192
```

等价于 yml 配置：

```
dubbo:
  rpc:
    tri:
      header-table-size: 4096
      enable-push: false
      max-concurrent-streams: 2147483647
      initial-window-size: 1048576
      max-frame-size: 32768
      max-header-list-size: 8192
```

#### 6) 属性与 XML 配置映射规则

可以将 xml 的 tag 名和属性名组合起来，用'.'分隔。每行一个属性。

- `dubbo.application.name=foo` 相当于 `<dubbo:application name="foo" />`
- `dubbo.registry.address=10.20.153.10:9090` 相当于 `<dubbo:registry address="10.20.153.10:9090" />`

如果在 xml 配置中有超过一个的 tag，那么你可以使用'id'进行区分。如果你不指定 id，它将作用于所有 tag。

- `dubbo.protocols.rmi.port=1099` 相当于 `<dubbo:protocol id="rmi" name="rmi" port="1099" />`
- `dubbo.registries.china.address=10.20.153.10:9090` 相当于 `<dubbo:registry id="china" address="10.20.153.10:9090" />`

### 7) 配置项单复数对照表

复数配置的命名与普通单词变复数的规则相同：

- 字母 y 结尾时，去掉 y，改为 ies
- 字母 s 结尾时，加 es
- 其它加 s

| Config Type | 单数配置 | 复数配置 |
| --- | --- | --- |
| application | dubbo.application.xxx=xxx | dubbo.applications.{id}.xxx=xxx<br>dubbo.applications.{name}.xxx=xxx |
| protocol | dubbo.protocol.xxx=xxx | dubbo.protocols.{id}.xxx=xxx<br>dubbo.protocols.{name}.xxx=xxx |
| module | dubbo.module.xxx=xxx | dubbo.modules.{id}.xxx=xxx<br>dubbo.modules.{name}.xxx=xxx |
| registry | dubbo.registry.xxx=xxx | dubbo.registries.{id}.xxx=xxx |
| monitor | dubbo.monitor.xxx=xxx | dubbo.monitors.{id}.xxx=xxx |
| config-center | dubbo.config-center.xxx=xxx | dubbo.config-centers.{id}.xxx=xxx |
| metadata-report | dubbo.metadata-report.xxx=xxx | dubbo.metadata-reports.{id}.xxx=xxx |
| ssl | dubbo.ssl.xxx=xxx | dubbo.ssls.{id}.xxx=xxx |
| metrics | dubbo.metrics.xxx=xxx | dubbo.metricses.{id}.xxx=xxx |
| provider | dubbo.provider.xxx=xxx | dubbo.providers.{id}.xxx=xxx |
| consumer | dubbo.consumer.xxx=xxx | dubbo.consumers.{id}.xxx=xxx |
| service | dubbo.service.{interfaceName}.xxx=xxx | 无 |
| reference | dubbo.reference.{interfaceName}.xxx=xxx | 无 |
| method | dubbo.service.{interfaceName}.{methodName}.xxx=xxx<br>dubbo.reference.{interfaceName}.{methodName}.xxx=xxx | 无 |

| | | |
|---|---|---|
| argument | dubbo.service.{interfaceName}.{methodName}.{arg-index}.xxx=xxx | 无 |

## 2. 配置来源

**Dubbo 默认支持 6 种配置来源：**

- JVM System Properties，JVM-D 参数。
- System environment，JVM 进程的环境变量。
- Externalized Configuration，外部化配置，从配置中心读取。
- Application Configuration，应用的属性配置，从 Spring 应用的 Environment 中提取“dubbo”打头的属性集。
- API/XML/注解等编程接口采集的配置可以被理解成配置来源的一种，是直接面向用户编程的配置采集方式。
- 从 classpath 读取配置文件 dubbo.properties。

**关于 dubbo.properties 属性：**

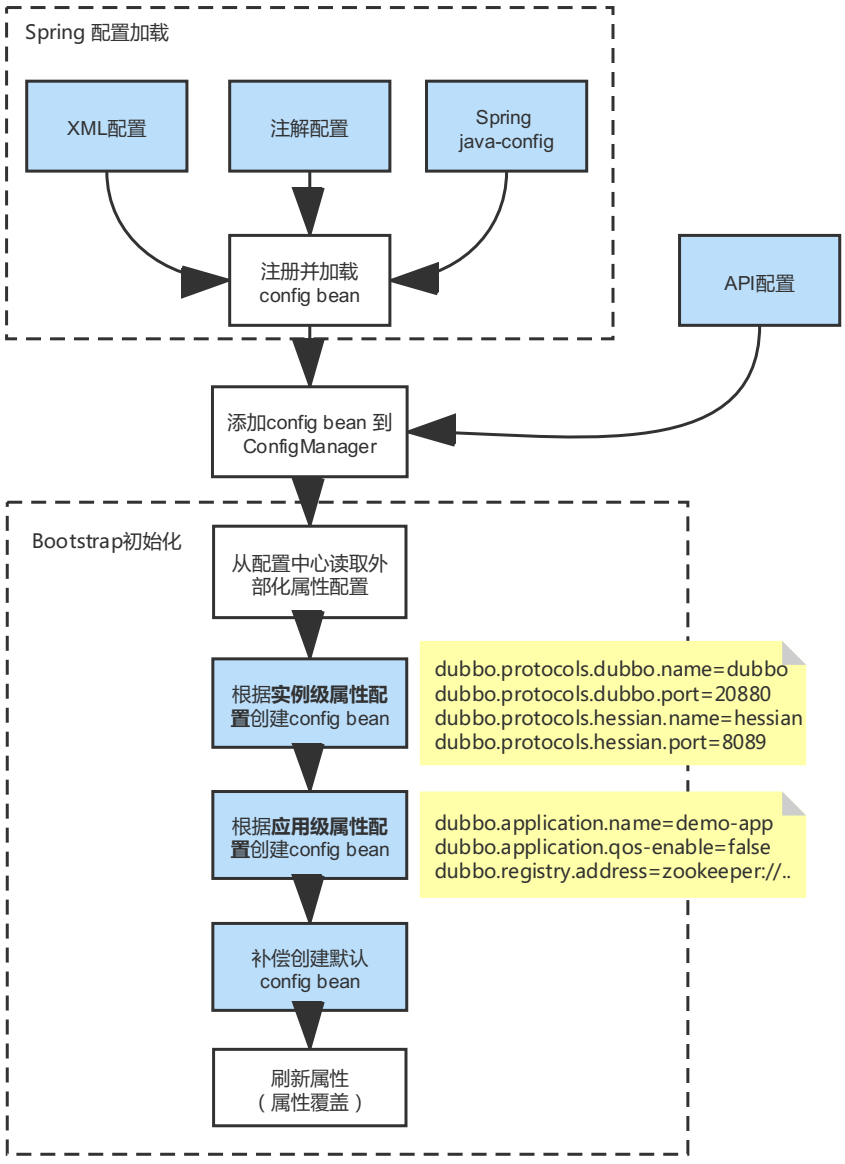
- 如果在 classpath 下有超过一个 dubbo.properties 文件，比如，两个 jar 包都各自包含了 dubbo.properties，dubbo 将随机选择一个加载，并且打印错误日志。
- Dubbo 可以自动加载 classpath 根目录下的 dubbo.properties，但是你同样可以使用 JVM 参数来指定路径：-Ddubbo.properties.file=xxx.properties。

### 1) 覆盖关系

如果通过多种配置来源指定了相同的配置项，则会出现配置项的互相覆盖，具体覆盖关系和优先级请参考下一小节。

## 3. 配置加载流程

### 1) 处理流程




Dubbo 配置加载大概分为两个阶段：

- **第一阶段为 DubboBootstrap 初始化之前**，在 Spring context 启动时解析处理 XML 配置/注解配置/Java-config 或者是执行 API 配置代码，创建 config bean 并且加入到 ConfigManager 中。

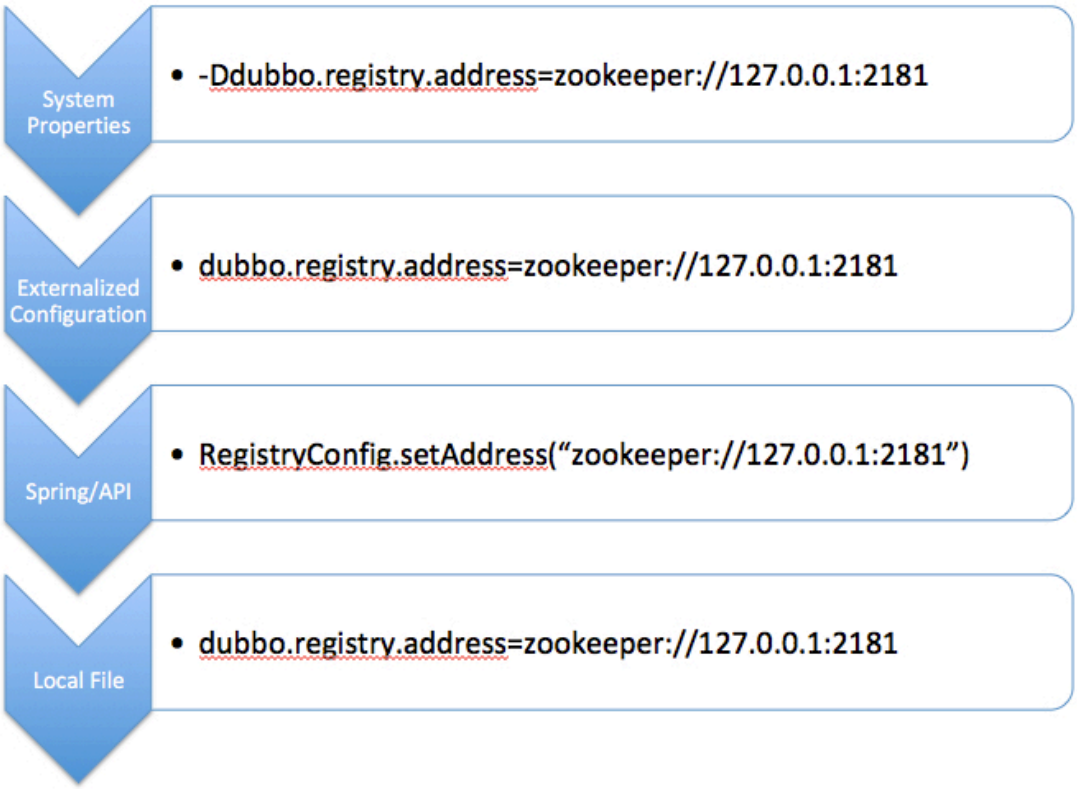
- **第二阶段为 DubboBootstrap 初始化过程**，从配置中心读取外部配置，依次处理实例级属性配置和应用级属性配置，最后刷新所有配置实例的属性，也就是属性覆盖。

### 2) 属性覆盖

发生属性覆盖可能有两种情况，并且二者可能是会同时发生的：

- 不同配置源配置了相同的配置项。
- 相同配置源，但在不同层次指定了相同的配置项。

### a) 不同配置源



**b) 相同配置源**

属性覆盖是指用配置的属性值覆盖 config bean 实例的属性，类似 Spring PropertyOverrideConfigurer 的作用。

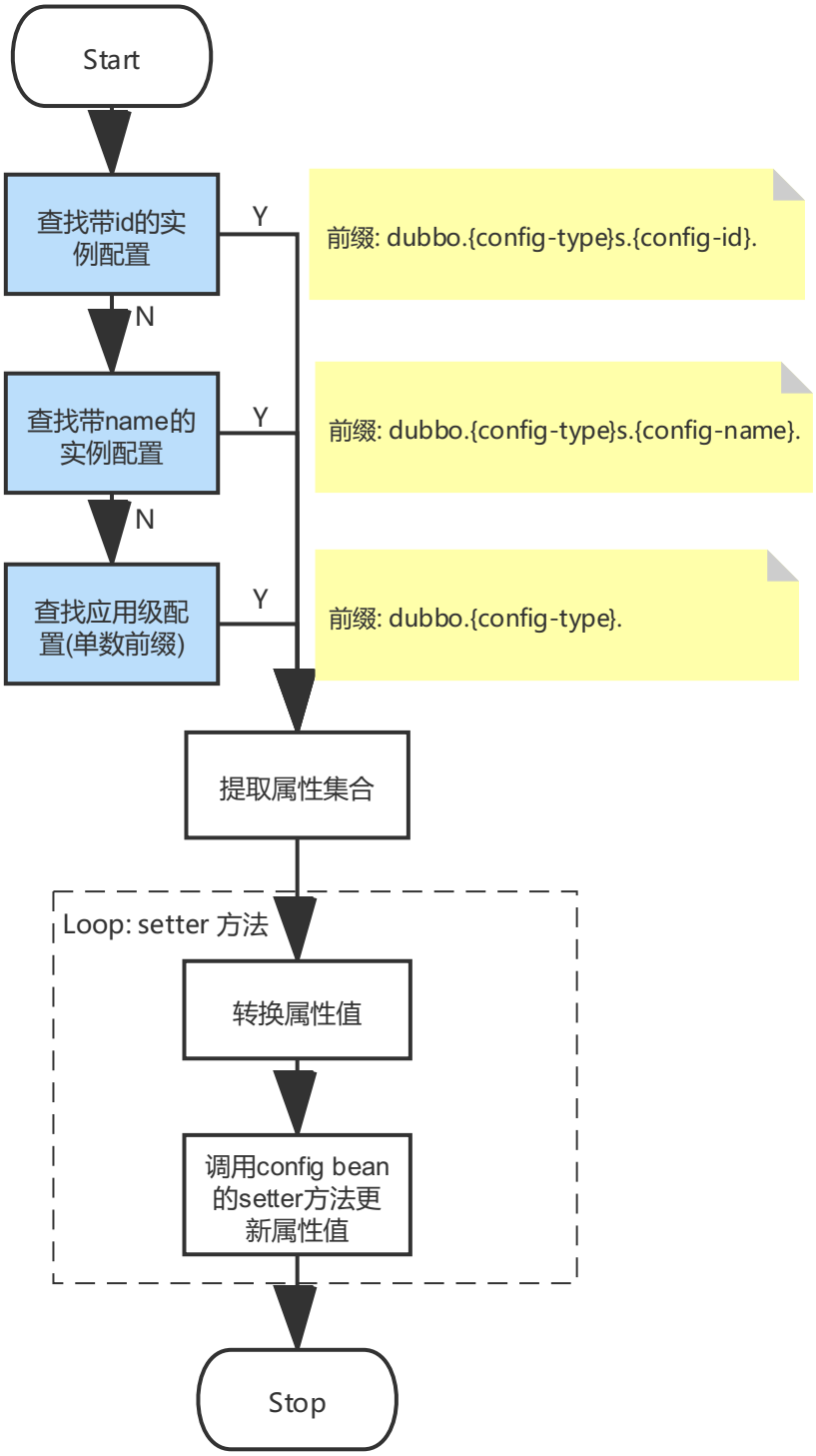
Property resource configurer that overrides bean property values in an application context definition. It pushes values from a properties file into bean definitions.
Configuration lines are expected to be of the following form:
beanName.property=value

但与 PropertyOverrideConfigurer 的不同之处是，Dubbo 的属性覆盖有多个匹配格式，优先级从高到低依次是：

```
#1. 指定 id 的实例级配置
dubbo.{config-type}s.{config-id}.{config-item}={config-item-value}

#2. 指定 name 的实例级配置
dubbo.{config-type}s.{config-name}.{config-item}={config-item-value}

#3. 应用级配置（单数配置）
dubbo.{config-type}.{config-item}={config-item-value}
```

**属性覆盖处理流程**：

按照优先级从高到低依次查找，如果找到此前缀开头的属性，则选定使用这个前缀提取属性，忽略后面的配置。




### 3) 外部化配置

外部化配置目的之一是实现配置的集中式管理，这部分业界已经有很多成熟的专业配置系统如 Apollo，Nacos 等，Dubbo 所做的主要是保证能配合这些系统正常工作。

外部化配置和其他本地配置在内容和格式上并无区别，可以简单理解为dubbo.properties 的外部化存储，配置中心更适合将一些公共配置如注册中心、元数据中心配置等抽取以便做集中管理。

```
# 将注册中心地址、元数据中心地址等配置集中管理，可以做到统一环境、减少开发侧感知。
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.registry.simplified=true

dubbo.metadata-report.address=zookeeper://127.0.0.1:2181

dubbo.protocol.name=dubbo
dubbo.protocol.port=20880

dubbo.application.qos.port=33333
```

- **优先级**

  外部化配置默认较本地配置有更高的优先级，因此这里配置的内容会覆盖本地配置值，关于各配置形式间的覆盖关系有单独一章说明。

- **作用域**

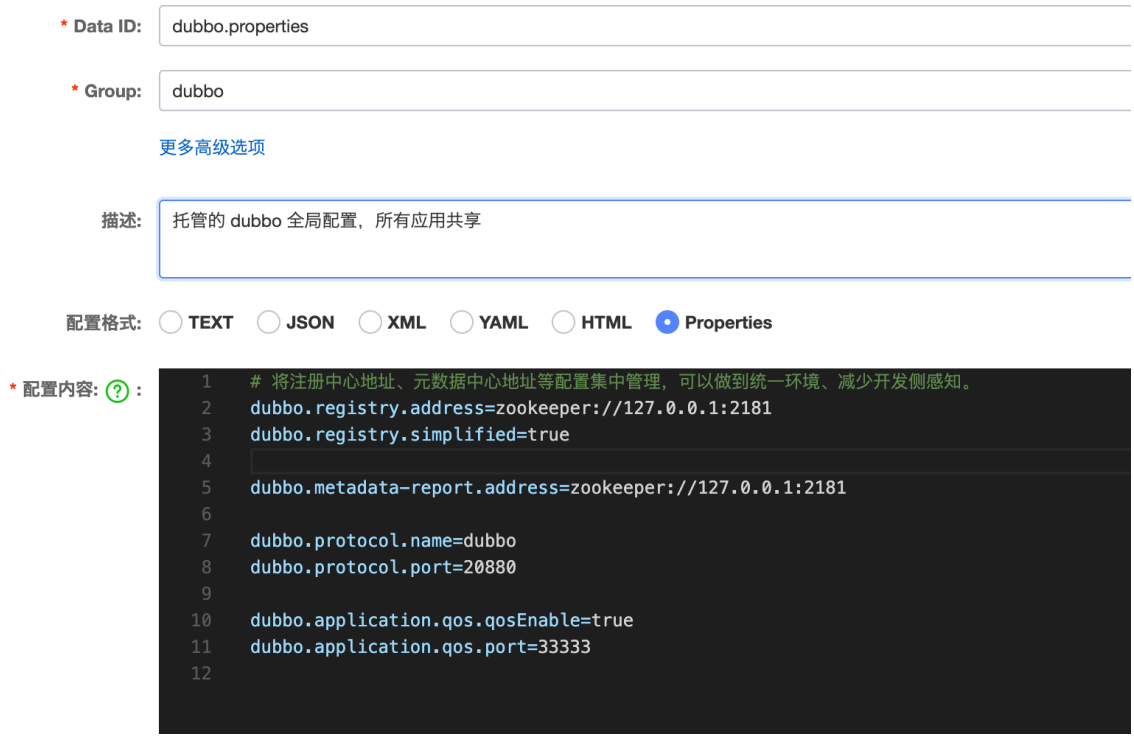
  外部化配置有全局和应用两个级别，全局配置是所有应用共享的，应用级配置是由每个应用自己维护且只对自身可见的。当前已支持的扩展实现有Zookeeper、Apollo、Nacos。

### a) 外部化配置使用方式

增加 config-center 配置

```
<dubbo:config-center address="zookeeper://127.0.0.1:2181"/>
```

在相应的配置中心（zookeeper、Nacos 等）增加全局配置项，如下以 Nacos 为例：




开启外部化配置后，registry、metadata-report、protocol、qos 等全局范围的配置理论上都不再需要在应用中配置，应用开发侧专注业务服务配置，一些全局共享的全局配置转而由运维人员统一配置在远端配置中心。这样能做到的效果就是，应用只需要关心：

- 服务暴露、订阅配置
- 配置中心地址

  当部署到不同的环境时，其他配置就能自动的被从对应的配置中心读取到。

举例来说，每个应用中 Dubbo 相关的配置只有以下内容可能就足够了，其余的都托管给相应环境下的配置中心：

```
dubbo
  application
    name: demo
  config-center
    address: nacos://127.0.0.1:8848
```

### b) 自行加载外部化配置

所谓 Dubbo 对配置中心的支持，本质上就是把.properties 从远程拉取到本地，然后和本地的配置做一次融合。理论上只要 Dubbo 框架能拿到需要的配置就可以正常的启动，它并不关心这些配置是自己加载到的还是应用直接塞给它的，所以 Dubbo 还提供了以下 API，让用户将自己组织好的配置塞给 Dubbo 框架（配置加载的过程是用户要完成的），这样 Dubbo 框架就不再直接和 Apollo 或 Zookeeper 做读取配置交互。

```
// 应用自行加载配置
Map<String, String> dubboConfigurations = new HashMap<>();
dubboConfigurations.put("dubbo.registry.address",
"zookeeper://127.0.0.1:2181");
dubboConfigurations.put("dubbo.registry.simplified", "true");

//将组织好的配置塞给 Dubbo 框架
ConfigCenterConfig configCenter = new ConfigCenterConfig();
configCenter.setExternalConfig(dubboConfigurations);
```

## 二、 DubboBootstrap 之借助双重校验锁的单例模式进行对象的初始化

### 1. 启动器简介

在说启动器之前先把视野拉回第一章《从一个服务提供者的 Demo 说起》我们的 Demo 代码，下面只贴一下核心代码。

```
public class Application {
    public static void main(String[] args) throws Exception {
            startWithBootstrap();
    }
    private static void startWithBootstrap() {
      //前面的文章都在说这个服务配置对象的创建,中间又说了分层域模型,扩展加载机制
        ServiceConfig<DemoServiceImpl> service = new ServiceConfig<>();
        //为服务配置下服务接口和服务实现,下面两行用来初始化对象就不详细说了
        service.setInterface(DemoService.class);
        service.setRef(new DemoServiceImpl());
        //这一个篇章主要说这里:
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .service(service)
            .start()
            .await();
    }
}
```

Dubbo3 往云原生的方向走自然要针对云原生应用的应用启动，应用运行，应用发布等信息做一些建模，这个 DubboBootstrap 就是用来启动 Dubbo 服务的。类似于 Netty 的 Bootstrap 类型和 ServerBootstrap 启动器。

## 2. 双重校验锁的单例模式创建启动器对象的

Dubbo 的 bootstrap 类为啥要用单例模式。

通过调用静态方法 getInstance()获取单例实例。之所以设计为单例，是因为 Dubbo 中的一些类（如 ExtensionLoader）只为每个进程设计一个实例。

下面就来直接看代码吧，代码胜千言。

对象的调用代码如下：

```
DubboBootstrap bootstrap = DubboBootstrap.getInstance();
```

DubboBootstrap 获取对象的 getInstance()方法：

```
public static DubboBootstrap getInstance() {
   //双重校验锁第一次判断空
       if (instance == null) {
         //为空都进行排队
           synchronized (DubboBootstrap.class) {
           //双重校验锁第二次判断空 上面为空的都排队了这里得判断下
               if (instance == null) {
                 //调用重载方法获取对象
                   instance = DubboBootstrap.getInstance(ApplicationModel.defaultModel());
               }
           }
       }
       return instance;
   }
```

DubboBootstrap 获取对象重载的 getInstance（ApplicationModel applicationModel）方法：

*computeIfAbsent()方法对 hashMap 中指定 key 的值进行重新计算，如果不存在这个 key，则添加到 hashMap 中。*

instanceMap 设计为 Map<ApplicationModel, DubboBootstrap>类型 Key，意味着可以为多个应用程序模型创建不同的启动器，启动多个服务。

```
 public static DubboBootstrap getInstance(ApplicationModel applicationModel) {
       return instanceMap.computeIfAbsent(applicationModel, _k -> new
DubboBootstrap(applicationModel));
    }
```

## 3. DubboBootstrap 的构造器代码

构造器代码是逻辑比较复杂的地方，我们先来看下代码。

```
private DubboBootstrap(ApplicationModel applicationModel) {
                //存储应用程序启动模型
       this.applicationModel = applicationModel;
       //获取配置管理器 ConfigManager:  配置管理器的扩展类型 ApplicationExt ，
扩展名字 config
       configManager = applicationModel.getApplicationConfigManager();
       //获取环境信息 Environment: 环境信息的扩展类型为 ApplicationExt,扩展名字
为 environment
       environment = applicationModel.getModelEnvironment();
```

```
                //执行器存储仓库(线程池)ExecutorRepository: 扩展类型为
ExecutorRepository,默认扩展扩展名字为 default
        executorRepository =
applicationModel.getExtensionLoader(ExecutorRepository.class).getDefau
ltExtension();
        //初始化并启动应用程序实例
ApplicationDeployer,DefaultApplicationDeployer 类型
        applicationDeployer = applicationModel.getDeployer();
        // listen deploy events
        //为发布器 设置生命周期回调
        applicationDeployer.addDeployListener(new
DeployListenerAdapter<ApplicationModel>() {
            @Override
            public void onStarted(ApplicationModel scopeModel) {
                notifyStarted(applicationModel);
            }

            @Override
            public void onStopped(ApplicationModel scopeModel) {
                notifyStopped(applicationModel);
            }

            @Override
            public void onFailure(ApplicationModel scopeModel, Throwable
cause) {
                notifyStopped(applicationModel);
            }
        });
        //将启动器对象注册到应用程序模型 applicationModel 的 Bean 工厂中
        // register DubboBootstrap bean
        applicationModel.getBeanFactory().registerBean(this);
    }
```

# 三、 DubboBootstrap 之添加应用程序的配置信息 ApplicationConfig

## 1. 简介

先贴个代码用来参考：

```
DubboBootstrap bootstrap = DubboBootstrap.getInstance();
bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .service(service)
            .start()
            .await();
```

上个博客我们说了启动器对象的创建，启动器对象在启动之前是要初始化一些配置信息的，这里我们来看这一行代码：

```
bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
```

## 2. 应用程序 ApplicationConfig 的配置信息

ApplicationConfig 的构造器比较简单就是为他的成员变量 name 赋值来标识这个应用程序的名字。

下面我们直接参考下官网的配置表格：

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 作用 | 描述 | 兼容性 |
|---|---|---|---|---|---|---|---|
| name | application | string | 必填 | | 服务治理 | 当前应用名称，用于注册中心计算应用间依赖关系，注意：消费者和提供者应用名不要一样，此参数不是匹配条件，你当前项目叫什么名字就填什么，和提供者消费者角色无关，比如：kylin 应用调用了 morgan 应用的服务，则 kylin 项目配成 kylin，morgan 项目配成 morgan，可能 kylin 也提供其它服务给别人使用，但 kylin 项目永远配成 kylin，这样注册中心将显示 kylin 依赖于 morgan | 1.0.16 以上版本 |
| version | application.version | string | 可选 | | 服务治理 | 当前应用的版本 | 2.2.0 以上版本 |
| owner | owner | string | 可选 | | 服务治理 | 应用负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5 以上版本 |
| organization | organization | string | 可选 | | 服务治理 | 组织名称（BU 或部门），用于注册中心区分服务来源，此配置项建议不要使用 autoconfig，直接写死在配置中，比如 china,intl,itu,crm,asc,dw,aliexpress 等 | 2.0.0 以上版本 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| architecture | architecture | string | 可选 | | 服务治理 | 用于服务分层对应的架构。如，intl、china。不同的架构使用不同的分层。 | 2.0.7 以上版本 |
| environment | environment | string | 可选 | | 服务治理 | 应用环境，如：develop/test/product，不同环境使用不同的缺省值，以及作为只用于开发测试功能的限制条件 | 2.0.0 以上版本 |
| compiler | compiler | string | 可选 | javassist | 性能优化 | Java 字节码编译器，用于动态类的生成，可选：jdk 或 javassist | 2.1.0 以上版本 |
| logger | logger | string | 可选 | slf4j | 性能优化 | 日志输出方式，可选：slf4j,jcl,log4j,log4j2,jdk | 2.2.0 以上版本 |
| metadata-type | metadata-type | String | 可选 | local | 服务治理 | metadata 传递方式，是以 Provider 视角而言的，Consumer 侧配置无效，可选值有： remote - Provider 把 metadata 放到远端注册中心，Consumer 从注册中心获取 local - Provider 把 metadata 放在本地，Consumer 从 Provider 处直接获取 | 2.7.6 以上版本 |

官网的配置很详细了上面有一些属性是值得注意的比如这个 name，compiler，logger，metadata-type 我们可能要多看下默认值是什么，方便我们在使用过程中遇到问题的排查。

常用的属性参考官网的表格已经足够了，不过上面的属性不是列举了所有的属性，后续应该官方文档回更新。

我这里把缺失的一些属性列举出来：

| 变量 | 类型 | 说明 |
|---|---|---|
| name | application | string |
| registries | List<RegistryConfig> | 应用级注册中心列表 |
| registryIds | String | 注册中心 id 列表 |
| monitor | MonitorConfig | 应用级监控配置 |
| dumpDirectory | String | 保存线程转储的目录 |
| qosEnable | Boolean | 是否启用 qos |
| qosHost | String | 要侦听的 qos 主机地址 |
| qosPort | Integer | 要侦听的 qos 端口 |
| qosAcceptForeignIp | Boolean | qos 是否接收外部 IP |
| parameters | Map<String, String> | 自定义参数 |
| shutwait | String | 应用程序关闭时间 |
| hostname | String | 主机名 |
| registerConsumer | Boolean | 用于控制是否将实例注册到注册表。仅当实例是纯消费者时才设置为“false”。 |
| repository | String | 没找到哪里用了 |
| enableFileCache | Boolean | 是否开启本地文件缓存 |
| protocol | String | 此应用程序的首选协议（名称）适用于难以确定哪个是首选协议的地方 |
| metadataServiceProtocol | String | 用于点对点的元数据传输的协议 |

| | | |
|---|---|---|
| metadataServicePort | Integer | 元数据服务端口号，用于服务发现 |
| livenessProbe | String | Liveness 存活探针 用于设置 qos 中探测器的扩展 |
| readinessProbe | String | Readiness 就绪探针 |
| startupProbe | String | Startup 启动探针 |
| registerMode | String | 注册模式,实例级,接口集,所有 |
| enableEmptyProtection | Boolean | 接收到的空 url 地址列表和空保护被禁用，将清除当前可用地址 |

这里我们先来简单了解下这个实体类型的基本配置，直接看配置可能不太好理解，后面我们讲到每个配置的时候可以回来参考一下。

**应用程序配置对象添加到启动器中的配置管理器中**

了解了配置信息再回过头来看下这个配置信息如何存放到启动器里面的。

我们的 Demo 调用代码如下：

```
DubboBootstrap bootstrap = DubboBootstrap.getInstance();
bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
```

DubboBootstrap 的 application 方法设置一个应用程序配置 ApplicationConfig 对象。

```
public DubboBootstrap application(ApplicationConfig applicationConfig) {
   //将启动器构造器中初始化的默认应用程序模型对象传递给配置对象
       applicationConfig.setScopeModel(applicationModel);
       //将配置信息添加到配置管理器中
       configManager.setApplication(applicationConfig);
       return this;
   }
```

ConfigManager 配置管理器的 setApplication 方法

```
@DisableInject
  public void setApplication(ApplicationConfig application) {
       addConfig(application);
   }
```

ConfigManager 配置管理器的 addConfig 方法

```
public final <T extends AbstractConfig> T addConfig(AbstractConfig
config) {

        if (config == null) {
            return null;
        }
        // ignore MethodConfig
        //检查当前配置管理器支持管理的配置对象
        //目前支持的配置有
ApplicationConfig,MonitorConfig,MetricsConfig,SslConfig,

//ProtocolConfig,RegistryConfig,ConfigCenterConfig,MetadataReportConfi
g
        if (!isSupportConfigType(config.getClass())) {
            throw new IllegalArgumentException("Unsupported config type:
" + config);
        }

        if (config.getScopeModel() != scopeModel) {
            config.setScopeModel(scopeModel);
        }

                //缓存中是否存在
        Map<String, AbstractConfig> configsMap =
configsCache.computeIfAbsent(getTagName(config.getClass()), type ->
new ConcurrentHashMap<>());

        // fast check duplicated equivalent config before write lock
        //不是服务级配置则直接从缓存中读取到配置之后直接返回
        if (!(config instanceof ReferenceConfigBase || config instanceof
ServiceConfigBase)) {
            for (AbstractConfig value : configsMap.values()) {
                if (value.equals(config)) {
                    return (T) value;
                }
            }
        }

        // lock by config type
        //添加配置
        synchronized (configsMap) {
            return (T) addIfAbsent(config, configsMap);
        }
```

```
    }
```

ConfigManager 配置管理器的 addIfAbsent 方法：

```
private <C extends AbstractConfig> C addIfAbsent(C config, Map<String,
C> configsMap)
        throws IllegalStateException {
                //配置信息为空直接返回
        if (config == null || configsMap == null) {
            return config;
        }

        // find by value
        //根据配置规则判断,配置存在则返回
        Optional<C> prevConfig = findDuplicatedConfig(configsMap,
config);
        if (prevConfig.isPresent()) {
            return prevConfig.get();
        }

                //生成配置 key
        String key = config.getId();
        if (key == null) {
            do {
                // generate key if id is not set
                key = generateConfigId(config);
            } while (configsMap.containsKey(key));
        }

                //不相同的配置 key 重复则抛出异常
        C existedConfig = configsMap.get(key);
        if (existedConfig != null && !isEquals(existedConfig, config)) {
            String type = config.getClass().getSimpleName();
            logger.warn(String.format("Duplicate %s found, there already
has one default %s or more than two %ss have the same id, " +
                    "you can try to give each %s a different id, override
previous config with later config. id: %s, prev: %s, later: %s",
                type, type, type, type, key, existedConfig, config));
        }

        // override existed config if any
        //将配置对象存入 configsMap 对象中,configsMap 来源于 configsCache
```

```
        configsMap.put(key, config);
        return config;
    }
```

# 四、 DubboBootstrap 之添加注册中心配置信息 RegistryConfig

## 1. 简介

先贴个代码用来参考：

```
DubboBootstrap bootstrap = DubboBootstrap.getInstance();
bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .service(service)
            .start()
            .await();
```

上个博客我们说了启动器 ApplicationConfig 对象的创建，启动器对象在启动之前是要初始化一些配置信息的，这里我们来看这一行代码注册中心配置信息。

```
registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
```

## 2. 注册中心的配置相关

下面的配置来源于官网

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 作用 | 描述 | 兼容性 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| id | | string | 可选 | | 配置关联 | 注册中心引用 BeanId，可以在 <dubbo:service registry=""> 或 <dubbo:reference registry="">中引用此 ID | 1.0.16 以上版本 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| address | host:port | string | 必填 | | 服务发现 | 注册中心服务器地址，如果地址没有端口缺省为 9090，同一集群内的多个地址用逗号分隔，如：ip:port,ip:port，不同集群的注册中心，请配置多个 dubbo:registry 标签 | 1.0.16 以上版本 |
| protocol | <protocol> | string | 可选 | dubbo | 服务发现 | 注册中心地址协议，支持 dubbo, multicast，zookeeper，redis, consul（2.7.1），sofa（2.7.2），etcd（2.7.2），nacos（2.7.2）等协议 | 2.0.0 以上版本 |
| port | <port> | int | 可选 | 9090 | 服务发现 | 注册中心缺省端口，当 address 没有带端口时使用此端口做为缺省值 | 2.0.0 以上版本 |
| username | <username> | string | 可选 | | 服务治理 | 登录注册中心用户名，如果注册中心不需要验证可不填 | 2.0.0 以上版本 |
| password | <password> | string | 可选 | | 服务治理 | 登录注册中心密码，如果注册中心不需要验证可不填 | 2.0.0 以上版本 |
| transport | registry.transporter | string | 可选 | netty | 性能调优 | 网络传输方式，可选 mina,netty | 2.0.0 以上版本 |
| timeout | registry.timeout | int | 可选 | 5000 | 性能调优 | 注册中心请求超时时间（毫秒） | 2.0.0 以上版本 |
| session | registry.session | int | 可选 | 60000 | 性能调优 | 注册中心会话超时时间（毫秒），用于检测提供者非正常断线后的脏数据，比如用心跳检测的实现，此时间就是心跳间隔，不同注册中心实现不一样。 | 2.1.0 以上版本 |
| file | registry.file | string | 可选 | | 服务治理 | 使用文件缓存注册中心地址列表及服务提供者列表，应用重启时将基于此文件恢复，注意：两个注册中心不能使用同一文件存储 | 2.0.0 以上版本 |
| wait | registry.wait | int | 可选 | 0 | 性能调优 | 停止时等待通知完成时间（毫秒） | 2.0.0 以上版本 |
| check | check | boolean | 可选 | true | 服务治理 | 注册中心不存在时，是否报错 | 2.0.0 以上版本 |
| register | register | boolean | 可选 | true | 服务治理 | 是否向此注册中心注册服务，如果设为 false，将只订阅，不注册 | 2.0.5 以上版本 |
| subscribe | subscribe | boolean | 可选 | true | 服务治理 | 是否向此注册中心订阅服务，如果设为 false，将只注册，不订阅 | 2.0.5 以上版本 |
| dynamic | dynamic | boolean | 可选 | true | 服务治理 | 服务是否动态注册，如果设为 false，注册后将显示为 disable 状态，需人工启用，并且服务提供者停止时，也不会自动取消注册，需人工禁用。 | 2.0.5 以上版本 |
| group | group | string | 可选 | dubbo | 服务治理 | 服务注册分组，跨组的服务不会相互影响，也无法相互调用，适用于环境隔离。 | 2.0.5 以上版本 |
| simplified | simplified | boolean | 可选 | false | 服务治理 | 注册到注册中心的 URL 是否采用精简模式的（与低版本兼容） | 2.7.0 以上版本 |
| extra-keys | extraKeys | string | 可选 | | 服务治理 | 在 simplified=true 时，extraKeys 允许你在默认参数外将额外的 key 放到 URL 中，格式：“interface,key1,key2”。 | 2.7.0 以上版本 |

同样官网提供的参数里面并未包含所有的属性 下面我就将其余的属性列举一下方便学习参考：

| 变量 | 类型 | 说明 |
| --- | --- | --- |
| server | String | |
| client | String | |
| cluster | String | 影响流量在注册中心之间的分布，在订阅多个注册中心时很有用，可用选项：1。区域感知，特定类型的流量总是根据流量的来源进入一个注册表。 |
| zone | String | 注册表所属的区域，通常用于隔离流量 |
| parameters | Map<String, String> | 自定义参数 |
| useAsConfigCenter | Boolean | 该地址是否用作配置中心 |
| useAsMetadataCenter | Boolean | 该地址是否用作远程元数据中心 |
| accepts | String | 此注册表接受的 rpc 协议列表，例如“dubbo，rest” |
| preferred | Boolean | 如果设置为 true，则始终首先使用此注册表，这在订阅多个注册表时非常有用 |
| weight | Integer | 影响注册中心之间的流量分布，当订阅多个注册中心仅在未指定首选注册中心时才生效时，此功能非常有用。 |
| registerMode | String | 注册模式:实例级,接口级,所有 |
| enableEmptyProtection | Boolean | 收到的空 url 地址列表和空保护被禁用，将清除当前可用地址 |

## 3. 注册中心配置对象创建与添加

前面例子中调用的代码

```
.registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
```

首先我们要来看的是 RegistryConfig 类型的构造器

```
public RegistryConfig(String address) {
        setAddress(address);
 }
```

继续看 setAddress 方法

```
public void setAddress(String address) {
    //保存地址
        this.address = address;
        //下面是支持将参数在url地址后面 比如用户名,密码,协议,端口,这几个参数提前做解析放入成员变量中
        if (address != null) {
            try {
              //地址转Dubbo的URL对象 这个URL是Dubbo自行实现的URL封装信息的类型
                URL url = URL.valueOf(address);

                // Refactor since 2.7.8
                //值不存在时候更新属性,非常巧妙的代码 重构了多个if判断
                //第一个参数值不存在则调用第二个方法,第二个方法的参数为第三方方法
                updatePropertyIfAbsent(this::getUsername, this::setUsername,
url.getUsername());
                updatePropertyIfAbsent(this::getPassword, this::setPassword,
url.getPassword());
                updatePropertyIfAbsent(this::getProtocol, this::setProtocol,
url.getProtocol());
                updatePropertyIfAbsent(this::getPort, this::setPort, url.getPort());

        //移除掉url中的backup自定义参数 (备份的注册中心地址)
                Map<String, String> params = url.getParameters();
                if (CollectionUtils.isNotEmptyMap(params)) {
                    params.remove(BACKUP_KEY);
                }
                //将自定义参数存储到成员变量中
                updateParameters(params);
            } catch (Exception ignored) {
            }
        }
    }
```

然后再回过头来看 DubboBootstrap 的 registry 方法:

```
  public DubboBootstrap registry(RegistryConfig registryConfig) {
        //将applicationModel对象设置给注册中心配置对象
        registryConfig.setScopeModel(applicationModel);
        //将注册中心配置对象添加到配置管理器中
        configManager.addRegistry(registryConfig);
        return this;
    }
```

直接来看配置管理器 configManager 的添加注册中心配置 addRegistry 方法:

```
public void addRegistry(RegistryConfig registryConfig) {
    addConfig(registryConfig);
}
```

configManager 的 addConfig 方法:

```
public final <T extends AbstractConfig> T addConfig(AbstractConfig
config) {
       if (config == null) {
           return null;
       }
       // ignore MethodConfig
       //检查当前配置管理器支持管理的配置对象
       //目前支持的配置有
ApplicationConfig,MonitorConfig,MetricsConfig,SslConfig,

//ProtocolConfig,RegistryConfig,ConfigCenterConfig,MetadataReportConfi
g
     if (!isSupportConfigType(config.getClass())) {
           throw new IllegalArgumentException("Unsupported config type:
" + config);
       }

       if (config.getScopeModel() != scopeModel) {
           config.setScopeModel(scopeModel);
       }
                //缓存中是否存在
       Map<String, AbstractConfig> configsMap =
configsCache.computeIfAbsent(getTagName(config.getClass()), type ->
new ConcurrentHashMap<>());
                //不是服务级接口配置则直接从缓存中读取到配置之后直接返回
       // fast check duplicated equivalent config before write lock
       if (!(config instanceof ReferenceConfigBase || config instanceof
ServiceConfigBase)) {
           for (AbstractConfig value : configsMap.values()) {
               if (value.equals(config)) {
                   return (T) value;
               }
           }
       }

       // lock by config type
       //添加配置
       synchronized (configsMap) {
           return (T) addIfAbsent(config, configsMap);
       }
   }
```

ConfigManager 配置管理器的 addIfAbsent 方法：

```
private <C extends AbstractConfig> C addIfAbsent(C config, Map<String,
C> configsMap)
       throws IllegalStateException {
                //配置信息为空直接返回
       if (config == null || configsMap == null) {
           return config;
       }

       // find by value
       //根据配置规则判断,配置存在则返回
       Optional<C> prevConfig = findDuplicatedConfig(configsMap,
config);
       if (prevConfig.isPresent()) {
           return prevConfig.get();
       }

                //生成配置 key
       String key = config.getId();
       if (key == null) {
           do {
               // generate key if id is not set
               key = generateConfigId(config);
           } while (configsMap.containsKey(key));
       }

                //不相同的配置 key 重复则抛出异常
       C existedConfig = configsMap.get(key);
       if (existedConfig != null && !isEquals(existedConfig, config)) {
           String type = config.getClass().getSimpleName();
           logger.warn(String.format("Duplicate %s found, there already
has one default %s or more than two %ss have the same id, " +
                   "you can try to give each %s a different id, override
previous config with later config. id: %s, prev: %s, later: %s",
               type, type, type, type, key, existedConfig, config));
       }

       // override existed config if any
       //将配置对象存入 configsMap 对象中,configsMap 来源于 configsCache
       configsMap.put(key, config);
       return config;
```

```
    }
```

# 五、 DubboBootstrap 之添加协议配置信息 ProtocolConfig

## 1. 简介

先贴个代码用来参考：

```
DubboBootstrap bootstrap = DubboBootstrap.getInstance();
bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .service(service)
            .start()
            .await();
```

上个博客我们说了 RegistryConfig 对象的创建，启动器对象在启动之前是要初始化一些配置信息的，这里我们来看这一行代码协议配置信息：

```
.protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
```

## 2. 协议的配置相关

下面的配置来源于官网。

服务提供者协议配置。对应的配置类：org.apache.dubbo.config.ProtocolConfig。同时，如果需要支持多协议，可以声明多个<dubbo:protocol>标签，并在<dubbo:service>中通过 protocol 属性指定使用的协议。

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 作用 | 描述 | 兼容性 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| id | | string | 可选 | dubbo | 配置关联 | 协议 BeanId，可以在<dubbo:service protocol=""> 中引用此 ID，如果 ID 不填，缺 | 2.0.5 以上版本 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 省和 name 属性值一样，重复则在 name 后加序号。 | |
| name | <protocol> | string | 必填 | dubbo | 性能调优 | 协议名称 | 2.0.5 以上版本 |
| port | <port> | int | 可选 | dubbo 协议缺省端口为 20880，rmi 协议缺省端口为 1099，http 和 hessian 协议缺省端口为 80；如果没有配置 port，则自动采用默认端口，如果配置为-1，则会分配一个没有被占用的端口。Dubbo 2.4.0+，分配的端口在协议缺省端口的基础上增长，确保端口段可控。 | 服务发现 | 服务端口 | 2.0.5 以上版本 |
| host | <host> | string | 可选 | 自动查找本机 IP | 服务发现 | -服务主机名，多网卡选择或指定 VIP 及域名时使用，为空则自动查找本机 IP，-建议不要配置，让 Dubbo 自动获取本机 IP | 2.0.5 以上版本 |
| threadpool | threadpool | string | 可选 | fixed | 性能调优 | 线程池类型，可选：fixed/cached | 2.0.5 以上版本 |
| threads | threads | int | 可选 | 200 | 性能调优 | 服务线程池大小（固定大小） | 2.0.5 以上版本 |
| iothreads | threads | int | 可选 | cpu 个数+1 | 性能调优 | io 线程池大小（固定大小） | 2.0.5 以上版本 |
| accepts | accepts | int | 可选 | 0 | 性能调优 | 服务提供方最大可接受连接数 | 2.0.5 以上版本 |
| payload | payload | int | 可选 | 8388608(=8M) | 性能调优 | 请求及响应数据包大小限制，单位：字节 | 2.0.5 以上版本 |
| codec | codec | string | 可选 | dubbo | 性能调优 | 协议编码方式 | 2.0.5 以上版本 |
| serialization | serialization | string | 可选 | dubbo 协议缺省为 hessian2，rmi 协议缺省为 java，http 协议缺省为 json | 性能调优 | 协议序列化方式，当协议支持多种序列化方式时使用，比如：dubbo 协议的 dubbo,hessian2,java,compactedjava，以及 http 协议的 json 等 | 2.0.5 以上版本 |
| accesslog | accesslog | string/boolean | 可选 | | 服务治理 | 设为 true，将向 logger 中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件 | 2.0.5 以上版本 |
| path | <path> | string | 可选 | | 服务发现 | 提供者上下文路径，为服务 path 的前缀 | 2.0.5 以上版本 |
| transporter | transporter | string | 可选 | dubbo 协议缺省为 netty | 性能调优 | 协议的服务端和客户端实现类型，比如：dubbo 协议的 mina,netty 等，可以分拆为 server 和 client 配置 | 2.0.5 以上版本 |
| server | server | string | 可选 | dubbo 协议缺省为 netty，http 协议缺省为 servlet | 性能调优 | 协议的服务器端实现类型，比如：dubbo 协议的 mina,netty 等，http 协议的 jetty,servlet 等 | 2.0.5 以上版本 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| client | client | string | 可选 | dubbo 协议缺省为 netty | 性能调优 | 协议的客户端实现类型，比如：dubbo 协议的 mina,netty 等 | 2.0.5 以上版本 |
| dispatcher | dispatcher | string | 可选 | dubbo 协议缺省为 all | 性能调优 | 协议的消息派发方式，用于指定线程模型，比如：dubbo 协议的 all, direct, message, execution, connection 等 | 2.1.0 以上版本 |
| queues | queues | int | 可选 | 0 | 性能调优 | 线程池队列大小，当线程池满时，排队等待执行的队列大小，建议不要设置，当线程池满时应立即失败，重试其它服务提供机器，而不是排队，除非有特殊需求。 | 2.0.5 以上版本 |
| charset | charset | string | 可选 | UTF-8 | 性能调优 | 序列化编码 | 2.0.5 以上版本 |
| buffer | buffer | int | 可选 | 8192 | 性能调优 | 网络读写缓冲区大小 | 2.0.5 以上版本 |
| heartbeat | heartbeat | int | 可选 | 0 | 性能调优 | 心跳间隔，对于长连接，当物理层断开时，比如拔网线，TCP 的 FIN 消息来不及发送，对方收不到断开事件，此时需要心跳来帮助检查连接是否已断开 | 2.0.10 以上版本 |
| telnet | telnet | string | 可选 | | 服务治理 | 所支持的 telnet 命令，多个命令用逗号分隔 | 2.0.5 以上版本 |
| register | register | boolean | 可选 | true | 服务治理 | 该协议的服务是否注册到注册中心 | 2.0.8 以上版本 |
| contextpath | contextpath | String | 可选 | 缺省为空串 | 服务治理 | | 2.0.6 以上版本 |

同样官网提供的参数里面并未包含所有的属性 下面我就将其余的属性列举一下方便学习参考：

| 变量 | 类型 | 说明 |
|---|---|---|
| threadname | String | 线程池名称 |
| corethreads | Integer | 线程池核心线程大小 |
| alive | Integer | 线程池keepAliveTime，默认单位时间单位。毫秒 |
| exchanger | String | 交换器配置信息如何交换 |
| prompt | String | 命令行提示符 |
| status | String | 状态检查 |
| sslEnabled | Boolean | ssl是否启用 |

## 3. 协议配置对象创建与添加

前面例子中调用的代码。

```
.protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
```

这里我们配置了协议类型为 dubbo 端口为-1 则会分配一个没有被占用的端口

继续看下 DubboBootstrap 的 protocol 方法

```
public DubboBootstrap protocol(ProtocolConfig protocolConfig) {
    //配置信息转List
        return protocols(singletonList(protocolConfig));
    }
```

继续看 protocols 方法，这个代码与前面两个博客中看到的向配置管理器添加配置对象的逻辑是一样的，这里就不说了可以看前面的博客《Dubbo 启动器 DubboBootstrap 添加应用程序的配置信息 ApplicationConfig》。

```
   public DubboBootstrap protocols(List<ProtocolConfig> protocolConfigs) {
        if (CollectionUtils.isEmpty(protocolConfigs)) {
            return this;
        }
        for (ProtocolConfig protocolConfig : protocolConfigs) {
            protocolConfig.setScopeModel(applicationModel);
            configManager.addProtocol(protocolConfig);
        }
        return this;
    }
```

# 六、 全局视野再看服务端整体启动过程

## 1. 启动方法简介

在说启动方法之前先把视野拉回第一章《从一个服务提供者的 Demo 说起》我们的 Demo 代码，下面只贴一下核心代码：

```
public class Application {
    public static void main(String[] args) throws Exception {
            startWithBootstrap();
    }
    private static void startWithBootstrap() {
      //前面的文章都在说这个服务配置对象的创建,中间又说了分层域模型,扩展加载机制
        ServiceConfig<DemoServiceImpl> service = new ServiceConfig<>();
        //为服务配置下服务接口和服务实现,下面两行用来初始化对象就不详细说了
        service.setInterface(DemoService.class);
        service.setRef(new DemoServiceImpl());
        //这一个篇章主要说这里:
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        //初始化应用配置
        bootstrap.application(new ApplicationConfig("dubbo-demo-api-provider"))
        //初始化注册中心配置
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            //初始化协议配置
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            //初始化服务配置
            .service(service)
            //启动
            .start()
            .await();
    }
}
```

前面我们介绍了 Dubbo 启动器 DubboBootstrap 类型对象的创建，又介绍了为 DubboBootstrap 启动器初始化各种配置信息，这一个博客就开始到了分析启动方法的位置了，Dubbo 启动器借助 Deployer 发布器来启动和发布服务，发布器的启动过程包含了启动配置中心，加载配置，启动元数据中心，启动服务等操作都是比较重要又比较复杂的过程，这里我们先来看下启动过程的生命周期来为后面的内容做好铺垫。

## 2. 启动器启动方法的调用逻辑 start()

这里我们就直接来看 DubboBootstrap 的 start()方法：

```
public DubboBootstrap start() {
   //调用重载的方法进行启动参数代表是否等待启动结束
       this.start(true);
       return this;
   }
```

我们再来看重载的 start 方法：

```
public DubboBootstrap start(boolean wait) {
    //这个发布器是在ApplicationModel对象创建之后初始化的时候进行初始化的具体类型为
DefaultApplicationDeployer
        Future future = applicationDeployer.start();

        if (wait) {
            try {
              //等待异步启动的结果
                future.get();
            } catch (Exception e) {
              //启动失败则抛出一个异常
                throw new IllegalStateException("await dubbo application start finish
failure", e);
            }
        }
        return this;
    }
```

## 3. 应用程序发布器 DefaultApplicationDeployer 的启动方法

发布器是帮助我们发布服务和引用服务的，在 Dubbo3 中不论是服务提供者还是服务消费者如果想要启动服务都需要走这个启动方法的逻辑，所以务必重视。

我们直接来看 DefaultApplicationDeployer 的 start()代码：

```
@Override
    public Future start() {
         //启动锁，防止重复启动
        synchronized (startLock) {
            //发布器,状态已经设置为停止或者失败了就直接抛出异常
            if (isStopping() || isStopped() || isFailed()) {
                throw new IllegalStateException(getIdentifier() + " is
stopping or stopped, can not start again");
            }

            try {
                // maybe call start again after add new module, check if
any new module
                //可能在添加新模块后再次调用 start，检查是否有任何新模块
                //这里遍历当前应用程序下的所有模块如果某个模块是 PENDING 状态则这里
hasPendingModule 的值为 true
                boolean hasPendingModule = hasPendingModule();
                                    //发布器状态正在启动中
                if (isStarting()) {
```

```
                // currently, is starting, maybe both start by module
and application
                // if it has new modules, start them
                //存在挂器的模块
                if (hasPendingModule) {
                       //启动模块
                    startModules();
                }
                // if it is starting, reuse previous startFuture
                //模块异步启动中
                return startFuture;
            }

            // if is started and no new module, just return
            //如果已启动且没有新模块，直接返回
            if (isStarted() && !hasPendingModule) {
                return CompletableFuture.completedFuture(false);
            }

            // pending -> starting : first start app
            // started -> starting : re-start app
            //启动状态切换，将启动状态切换到 STARTING（pending 和 started 状态
无需切换）
            onStarting();
                                //核心初始化逻辑，这里主要做一些应用级别启动比
如配置中心，元数据中心
            initialize();
                                //启动模块（我们的服务提供和服务引用是在这个模
块级别的）
            doStart();
        } catch (Throwable e) {
            onFailed(getIdentifier() + " start failure", e);
            throw e;
        }

        return startFuture;
    }
}
```

这个启动方法逻辑不多，主要三个方法我们重点来看：

- onStarting()这个是启动之前的状态切换

- initialize()应用的初始化逻辑，比如配置中心，元数据中心的初始化
- doStart()启动模块比如启动我们的服务提供和服务引用的

继续看后面的细节吧，代码胜千言。

## 4. 应用程序发布器对应用级别的初始化逻辑

这个我们先来看 DefaultApplicationDeployer 的初始化方法 initialize()：

```
@Override
    public void initialize() {
         //状态判断 如果已经初始化过了就直接返回
        if (initialized) {
            return;
        }
        // Ensure that the initialization is completed when concurrent
calls
        //启动锁，确保在并发调用时完成初始化
        synchronized (startLock) {
         //双重校验锁 如果已经初始化过了就直接返回
            if (initialized) {
                return;
            }
            // register shutdown hook
            //注册关闭钩子，这个逻辑基本每个中间件应用都必须要要做的事情了，正常关闭
应用回收资源，一般没这个逻辑情况下容易出现一些异常，让我们开发人员很疑惑，而这个逻辑
往往并不好处理的干净。
            registerShutdownHook();

                        //启动配置中心，感觉 Dubbo3 耦合了这个玩意
            startConfigCenter();

                        //加载配置，一般配置信息当前机器的来源：环境变量，JVM 启
动参数，配置文字
            loadApplicationConfigs();

                        //初始化模块发布器 （发布服务提供和服务引用使用）
            initModuleDeployers();

            // @since 2.7.8
```

```
            //启动元数据中心
            startMetadataCenter();

                        //初始化完成
            initialized = true;

            if (logger.isInfoEnabled()) {
                logger.info(getIdentifier() + " has been initialized!");
            }
        }
    }
```

这个是个生命周期整体概览的方法，将具体逻辑拆分到各个子方法中，是代码重构的一种策略，上面注释也很清楚了就不细说了，上面每个方法在后面会有单独的博客来分析。

## 5. 应用下模块的启动（服务的发布与引用）

我们回过头来详细看 DefaultApplicationDeployer 的 doStart()代码：

```
private void doStart() {
    // 启动模块
        startModules();
```

DefaultApplicationDeployer 的 startModules()方法

```
private void startModules() {
        // ensure init and start internal module first
        //确保初始化并首先启动内部模块,Dubbo3中将模块分为内部和外部，内部是核心代码已经提供的一些服务比如元数据服务，外部是我们自己写的服务
        prepareInternalModule();

        // filter and start pending modules, ignore new module during starting, throw exception of module start
        //启动所有的模块 （启动所有的服务）
        for (ModuleModel moduleModel : new ArrayList<>(applicationModel.getModuleModels())) {
          //这个状态默认就是PENDING的
            if (moduleModel.getDeployer().isPending()) {
              //模块启动器，发布服务
                moduleModel.getDeployer().start();
            }
        }
    }
```

这个模块的启动其实就是用来启动服务的，先启动内部服务，再启动外部服务。

内部服务有个元数据服务 Dubbo3 中每个服务都可以对外提供服务的元数据信息，来简化服务配置，不论是内部服务还是外部服务调用的代码逻辑都是模块发布器 ModuleDeployer 的 start()方法，接下来我们详细看下模块发布器的生命周期函数。

## 6. 模块发布器发布服务的过程

前面我们说到了所有的服务都是经过模块发布器，ModuleDeployer 的 start()方法来启动的，那我们接下来就来看看这个模块发布器的启动方法。

ModuleDeployer 的 start()方法代码：

```
@Override
    public synchronized Future start() throws IllegalStateException {
        //模块发布器已经停止或者启动失败则直接抛出异常返回
        if (isStopping() || isStopped() || isFailed()) {
            throw new IllegalStateException(getIdentifier() + " is
stopping or stopped, can not start again");
        }

        try {
        //启动中或者已经启动了则直接返回一个 Future 对象
            if (isStarting() || isStarted()) {
                return startFuture;
            }
                        //切换模块启动状态为 STARTING
            onModuleStarting();

            // initialize
            //如果应用未初始化则初始化（非正常逻辑）
            applicationDeployer.initialize();
            //模块发布器进行初始化
            initialize();

            // export services
            //暴露服务
```

```
            exportServices();

            // prepare application instance
            // exclude internal module to avoid wait itself
            if (moduleModel !=
moduleModel.getApplicationModel().getInternalModule()) {
                applicationDeployer.prepareInternalModule();
            }

            // refer services
                        //引用服务
            referServices();

            // if no async export/refer services, just set started
            //非异步启动则直接切换状态为 STARTED
            if (asyncExportingFutures.isEmpty() &&
asyncReferringFutures.isEmpty()) {
                onModuleStarted();
            } else {
            //如果是异步的则等待服务发布和服务引用异步回调
                frameworkExecutorRepository.getSharedExecutor().submit(()
-> {
                    try {
                        // wait for export finish
                        waitExportFinish();
                        // wait for refer finish
                        waitReferFinish();
                    } catch (Throwable e) {
                        logger.warn("wait for export/refer services
occurred an exception", e);
                    } finally {
                            //异步回调完成 所有服务都启动了，再切换状态
                        onModuleStarted();
                    }
                });
            }
        } catch (Throwable e) {
            onModuleFailed(getIdentifier() + " start failed: " + e, e);
            throw e;
        }
        return startFuture;
    }
```

好了整体的服务启动生命周期就如上代码，后续我们再详细来看每个细节。

## 7. 发布器简介

前面主要说了应用和模块的发布器的启动和初始化，下面简单了解下它们的关系，如下所示。



可以发布器主要包含：

- 应用的发布器 ApplicationDeployer 用于初始化并启动应用程序实例
- 模块发布器 ModuleDeployer 模块（服务）的导出/引用服务

两种发布器有各自的接口，他们都继承了抽象的发布器 AbstractDeployer 封装了一些公共的操作比如状态切换，状态查询的逻辑。

另外我们再来看下发布过程的状态枚举 DeployState 如下：

```
public enum DeployState {
    /**
     * Unknown state
     */
    UNKNOWN,

    /**
     * Pending, wait for start
     */
    PENDING,
```

```
    /**
     * Starting
     */
    STARTING,

    /**
     * Started
     */
    STARTED,

    /**
     * Stopping
     */
    STOPPING,

    /**
     * Stopped
     */
    STOPPED,

    /**
     * Failed
     */
    FAILED
}
```

Dubbo 这一块后续可以优化以下，这里的状态切换全部耦合在一起了，可以考虑使用状态机将状态与行为解耦。

# 核心服务治理组件解析

## 一、 服务治理之注册、配置和元数据中心

### 1. 配置中心简介

百度了一段不错的文字来介绍配置中心，我看了下肯定比我写的好多了，那我就直接拷贝过来一起看。

对于传统的单体应用而言，常使用配置文件来管理所有配置，比如 SpringBoot 的 application.yml 文件，但是在微服务架构中全部手动修改的话很麻烦而且不易维护。微服务的配置管理一般有以下需求：

- **集中配置管理**。一个微服务架构中可能有成百上千个微服务，所以集中配置管理是很重要的。
- **不同环境不同配置**。比如数据源配置在不同环境（开发，生产，测试）中是不同的。
- **运行期间可动态调整**。例如，可根据各个微服务的负载情况，动态调整数据源连接池大小等。
- **配置修改后可自动更新**。如配置内容发生变化，微服务可以自动更新配置。

综上所述对于微服务架构而言，一套统一的，通用的管理配置机制是不可缺少的主要组成部分。常见的做法就是通过配置服务器进行管理。

不过对于来看这个文章的小伙伴应该大部分对配置中心都会比较了解，分布式配置中心实现简单一点就是借助 Zookeeper 来协助存储，变更推送，不过为了实现各种不同的业务需求，市面上已经有很多很可靠的配置中心可用了，比如我从其他地方拷贝过来的图（虽然不是最新的但是可以供大家参考下）。

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | 对比性 | 重要性 | Spring Cloud Config | netflix archaius | ctrip apollo | disconf |
| 2 | 功能特性 | | | | | |
| 3 | 静态配置管理 | 高 | 基于file | 无 | 支持 | 支持 |
| 4 | 动态配置管理 | 高 | 支持 | 支持 | 支持 | 支持 |
| 5 | 统一管理 | 高 | 无， 需要git、数据库等 | 无 | 支持 | 支持 |
| 6 | 多维度管理 | 中 | 无， 需要git、数据库等 | 无 | 支持 | 支持 |
| 7 | 变更管理 | 高 | 无， 需要git、数据库等 | 无 | 无 | 无 |
| 8 | 本地配置缓存 | 高 | 无 | 无 | 支持 | 支持 |
| 9 | 配置更新策略 | 中 | 无 | 无 | 无 | 无 |
| 10 | 配置锁 | 中 | 支持 | 不支持 | 不支持 | 不支持 |
| 11 | 配置校验 | 中 | 无 | 无 | 无 | 无 |
| 12 | 配置生效时间 | 高 | 重启生效、手动刷新 | 手动刷新生效 | 实时 | 实时 |
| 13 | 配置更新推送 | 高 | 需要手动触发 | 需要手动触发 | 支持 | 支持 |
| 14 | 配置定时拉取 | 高 | 无 | 无 | 支持 | 配置更新目前依赖事件驱动，<br>client 重启或者Server |
| 15 | 用户权限管理 | 中 | 无， 需要git、数据库等 | 无 | 支持 | 支持 |
| 16 | 授权、审核、审计 | 中 | 无， 需要git、数据库等 | 无 | 界面直接提供发布历史和回滚按钮 | 操作记录有落数据库，但无查询接口 |
| 17 | 配置版本管理 | 高 | git | 无 | 支持 | 操作记录有落数据库，但无查询接口 |
| 18 | 配置合规检测 | 高 | 不支持 | 不支持 | 支持(还需完善) | |
| 19 | 实例配置监控 | 高 | 需要结合spring admin | 不支持 | 支持 | 支持，可以查看每个配置在那台机器上加载 |
| 20 | 灰度发布 | 中 | 不支持 | 不支持 | 支持 | 不支持部分更新 |
| 21 | 告警通知 | 中 | 不支持 | 不支持 | 支持邮件方式警告 | 支持邮件方式警告 |
| 22 | 统计报表 | 中 | 不支持 | 不支持 | 不支持 | 不支持 |
| 23 | 依赖关系 | 高 | 不支持 | 不支持 | 不支持 | 不支持 |
| 24 | 技术路线 | | | | | |
| 25 | 支持Spring Boot | 高 | 原生支持 | 低 | 支持 | 与spring boot 无关 |
| 26 | 支持Spring Config | 高 | 原生支持 | 低 | 支持 | 与spring boot 无关 |
| 27 | 客户端支持 | 低 | java | java | java、.net | java |
| 28 | 业务系统入侵性 | 高 | 入侵性弱 | 入侵性弱 | 入侵性弱 | 入侵性弱， 支持注解和xml |
| 29 | 可依赖组建 | 高 | | | | |
| 30 | 可用性 | | | | | |
| 31 | 单点故障(SPOF) | 高 | 支持HA部署 | 支持HA部署 | 支持HA部署 | 支出HA部署， 高可用由ZK提供 |
| 32 | 多数据中心部署 | 高 | 支持 | 支持 | 支持 | 支持 |
| 33 | 配置获取性能 | 高 | unknow | unknow | unknow | unknow |
| 34 | 易用性 | | | | | |
| 35 | 配置界面 | 中 | 无， 需要git、数据库等操 | 无 | 统一界面(ng编写) | 统一界面 |

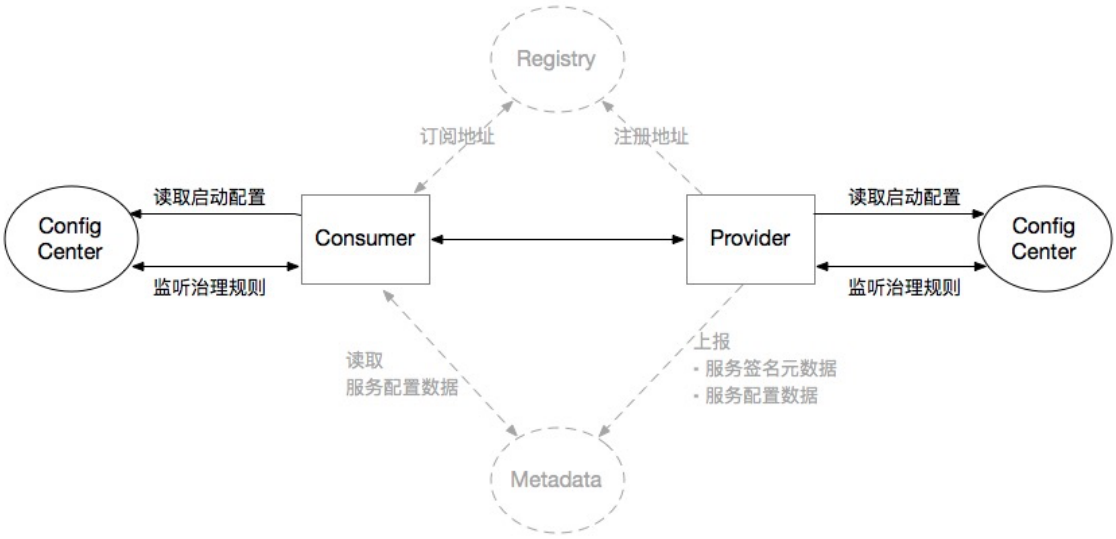


每个配置中心都有自己的实现，如果对配置中心感兴趣的小伙伴可以自行去对应开源项目官网查看，我们这里来看 Dubbo 对配置中心的支持。

**多配置中心**：Dubbo 支持多配置中心，来保证其中一个配置中心集群出现不可用时能够切换到另一个配置中心集群，保证能够正常从配置中心获取全局的配置、路由规则等信息。这也能够满足配置中心在部署上适应各类高可用的部署架构模式。

做中间件可能考虑更多的的不仅仅是性能，还要过多的考虑高可用，高可用怎么做呢，其实就是失效转移，主备切换，降级，降级再降级这些理论的运用，多多考虑某一个服务挂了怎么办，Dubbo 的多配置中心支持增加了复杂性，不过降低了服务不可用的风险，有一定的人手的公司还是值得做的。

关于 Dubbo 的配置中心这里我来贴个官网的图：




关于官网的介绍可以自行去官网看详细内容：部署架构（注册中心、配置中心、元数据中心）。

## 2. 启动配置中心

在上一个博客中说到了《全局视野来看 Dubbo3.0.8 的服务启动生命周期》Dubbo 应用的启动过程 DefaultApplicationDeployer 的 initialize()方法的全生命周期，在初始化方法中通过调用 startConfigCenter();方法来启动配置中心的加载。后面就来详细看下。

DefaultApplicationDeployer 类型的 startConfigCenter()代码如下

```
private void startConfigCenter() {

        // load application config
        //加载应用程序配置 （配置可能有多个地方可以配置需要遵循 Dubbo 约定的优先级进行设置，也可能是多应用，多注册中心这样的配置）

configManager.loadConfigsOfTypeFromProps(ApplicationConfig.class);

        // try set model name
        if (StringUtils.isBlank(applicationModel.getModelName())) {
        //设置一下模块名字和模块描述（我们再 Debug 里面经常会看到这个描述信息 toString 直接返回了 Dubbo 为我们改造的对象信息）
```

```
applicationModel.setModelName(applicationModel.tryGetApplicationName()
);
        }

        // load config centers
        //加载配置中心配置
        //配置可能有多个地方可以配置需要遵循 Dubbo 约定的优先级进行设置，也可能是多
应用，多注册中心这样的配置）

configManager.loadConfigsOfTypeFromProps(ConfigCenterConfig.class);
                //出于兼容性目的，如果没有明确指定配置中心，并且 registryConfig
的 UseAConfigCenter 为 null 或 true，请使用 registry 作为默认配置中心
        useRegistryAsConfigCenterIfNecessary();

        // check Config Center
        //配置管理器中获取配置中心
        Collection<ConfigCenterConfig> configCenters =
configManager.getConfigCenters();
        //配置中心配置不为空则刷新配置中心配置将其放入配置管理器中
        //下面开始刷新配置中心配置,如果配置中心配置为空则执行空刷新
        if (CollectionUtils.isEmpty(configCenters)) {
        //配置中心不存在的配置刷新
            ConfigCenterConfig configCenterConfig = new
ConfigCenterConfig();
            configCenterConfig.setScopeModel(applicationModel);
            configCenterConfig.refresh();
            //验证配置

ConfigValidationUtils.validateConfigCenterConfig(configCenterConfig);
            if (configCenterConfig.isValid()) {
            //配置合法则将配置放入配置管理器中
                configManager.addConfigCenter(configCenterConfig);
                configCenters = configManager.getConfigCenters();
            }
        } else {
        //一个或者多个配置中心配置存在的情况下的配置刷新
            for (ConfigCenterConfig configCenterConfig : configCenters)
{
                configCenterConfig.refresh();
                //验证配置

ConfigValidationUtils.validateConfigCenterConfig(configCenterConfig);
```

```
            }
        }

                //配置中心配置不为空则将配置中心配置添加到 environment 中
        if (CollectionUtils.isNotEmpty(configCenters)) {
        //多配置中心本地动态配置对象创建 CompositeDynamicConfiguration
            CompositeDynamicConfiguration compositeDynamicConfiguration
= new CompositeDynamicConfiguration();
            //获取配置中心的相关配置
            for (ConfigCenterConfig configCenter : configCenters) {
                // Pass config from ConfigCenterBean to environment
                //将配置中心的外部化配置,更新到环境里面

environment.updateExternalConfigMap(configCenter.getExternalConfigurat
ion());
                //将配置中心的应用配置,添加到环境里面

environment.updateAppExternalConfigMap(configCenter.getAppExternalConf
iguration());

                // Fetch config from remote config center
                //从配置中心拉取配置添加到组合配置中

compositeDynamicConfiguration.addConfiguration(prepareEnvironment(conf
igCenter));
            }
            //将配置中心中的动态配置信息 设置到 environment 的动态配置属性中

environment.setDynamicConfiguration(compositeDynamicConfiguration);
        }
    }
```

### 1) 配置管理器加载配置

前面我们看到了配置管理器会从系统属性中加载配置这里我们来详细看下，配置往往是我们使用者比较关注的内容，

```
configManager.loadConfigsOfTypeFromProps(ApplicationConfig.class);
```

配置管理器加载配置代码：

来自 ConfigManager 的父类型 AbstractConfigManager 中

```
public <T extends AbstractConfig> List<T>
loadConfigsOfTypeFromProps(Class<T> cls) {
       List<T> tmpConfigs = new ArrayList<>();
       //获取属性配置 dubbo properties in classpath
       //这个配置信息回头说
       PropertiesConfiguration properties =
environment.getPropertiesConfiguration();

       // load multiple configs with id
       //多注册中心配置 id 查询

      /*
      搜索属性并提取指定类型的配置 ID。
      例如如下配置
      # 配置信息 properties
      dubbo.registries.registry1.address=xxx
      dubbo.registries.registry2.port=xxx

      # 提取配置的 id extract
      Set configIds = getConfigIds(RegistryConfig.class)

      # 提取的配置 id 结果 result
      configIds: ["registry1", "registry2"]
      */
       Set<String> configIds = this.getConfigIdsFromProps(cls);
       configIds.forEach(id -> {
        //遍历这些配置 id 判断配置缓存(configsCache 成员变量)中是否已经存在当前配
置
          if (!this.getConfig(cls, id).isPresent()) {
              T config;
              try {
                //创建配置对象 为配置对象初始化配置 id
                  config = createConfig(cls, scopeModel);
                  config.setId(id);
              } catch (Exception e) {
                  throw new IllegalStateException("create config
instance failed, id: " + id + ", type:" + cls.getSimpleName());
              }
```

```
                String key = null;
                boolean addDefaultNameConfig = false;
                try {
                    // add default name config (same as id), e.g.
dubbo.protocols.rest.port=1234
                    key = DUBBO + "." +
AbstractConfig.getPluralTagName(cls) + "." + id + ".name";
                    if (properties.getProperty(key) == null) {
                        properties.setProperty(key, id);
                        addDefaultNameConfig = true;
                    }
                                    //刷新配置信息 好理解点就是 Dubbo 配置属性重写
                    config.refresh();
                    //将当前配置信息添加到配置缓存中 configsCache 成员变量
                    this.addConfig(config);
                    tmpConfigs.add(config);
                } catch (Exception e) {
                    logger.error("load config failed, id: " + id + ",
type:" + cls.getSimpleName(), e);
                    throw new IllegalStateException("load config failed,
id: " + id + ", type:" + cls.getSimpleName());
                } finally {
                    if (addDefaultNameConfig && key != null) {
                        properties.remove(key);
                    }
                }
            }
        });

        // If none config of the type, try load single config
        //如果没有该类型的配置，请尝试加载单个配置
        if (this.getConfigs(cls).isEmpty()) {
            // load single config
            List<Map<String, String>> configurationMaps =
environment.getConfigurationMaps();
            if (ConfigurationUtils.hasSubProperties(configurationMaps,
AbstractConfig.getTypePrefix(cls))) {
                T config;
                try {
                    config = createConfig(cls, scopeModel);
                    config.refresh();
                } catch (Exception e) {
```

```
                throw new IllegalStateException("create default config
instance failed, type:" + cls.getSimpleName());
            }

            this.addConfig(config);
            tmpConfigs.add(config);
        }
    }

    return tmpConfigs;
}
```

### 2) 默认使用注册中心地址为配置中心

出于兼容性目的，如果没有明确指定配置中心，并且 registryConfig 的 UseAConfigCenter 为 null 或 true，请使用 registry 作为默认配置中心调用方法 useRegistryAsConfigCenterIfNecessary()来处理逻辑。

我们来看下代码：

```
private void useRegistryAsConfigCenterIfNecessary() {
    // we use the loading status of DynamicConfiguration to decide
whether ConfigCenter has been initiated.
    //我们使用 DynamicConfiguration 的加载状态来决定是否已启动
ConfigCenter。配置中心配置加载完成之后会初始化动态配置
defaultDynamicConfiguration
    if (environment.getDynamicConfiguration().isPresent()) {
        return;
    }
            //从配置缓存中查询是否存在 config-center 相关配置 ,如果已经存在
配置了就无需使用注册中心的配置地址直接返回
    if
(CollectionUtils.isNotEmpty(configManager.getConfigCenters())) {
        return;
    }

    // load registry
    //加载注册中心相关配置
    configManager.loadConfigsOfTypeFromProps(RegistryConfig.class);
```

```
                //查询是否有注册中心设置了默认配置 isDefault 设置为 true 的注册中心则为默认注册中心列表,如果没有注册中心设置为默认注册中心,则获取所有未设置默认配置的注册中心列表
        List<RegistryConfig> defaultRegistries = configManager.getDefaultRegistries();
        //存在注册中心
        if (defaultRegistries.size() > 0) {
            defaultRegistries
                .stream()
                //判断当前注册中心是否可以作为配置中心
                .filter(this::isUsedRegistryAsConfigCenter)
                //将注册中心配置映射转换为配置中心
                .map(this::registryAsConfigCenter)
                //遍历配置中心流
                .forEach(configCenter -> {
                    if (configManager.getConfigCenter(configCenter.getId()).isPresent()) {
                        return;
                    }
                    //配置管理器中添加配置中心,方便后去读取配置中心的配置信息
                    configManager.addConfigCenter(configCenter);
                    logger.info("use registry as config-center: " + configCenter);

                });
        }
    }
```

### a) 如何判断当前注册中心是否可以为配置中心

isUsedRegistryAsConfigCenter

```
private boolean isUsedRegistryAsCenter(RegistryConfig registryConfig,
Supplier<Boolean> usedRegistryAsCenter,
                                       String centerType,
                                       Class<?> extensionClass) {
        final boolean supported;
                        //这个 useAsConfigCenter 参数是来自注册中心的配置 如果配置了这个值则以这个值为准,如果配置了 false 则这个注册中心不能做为配置中心
        Boolean configuredValue = usedRegistryAsCenter.get();
        if (configuredValue != null) { // If configured, take its value.
```

```
            supported = configuredValue.booleanValue();
        } else {                        // Or check the extension existence
         //这个逻辑的话是判断下注册中心的协议是否满足要求,我们例子代码中使用的是
zookeeper
            String protocol = registryConfig.getProtocol();
            //这个扩展是否支持的逻辑判断是这样的扫描扩展类 看一下当前扩展类型是否有
对应协议的扩展 比如在扩展文件里面这样配置过后是支持的 protocol=xxxImpl
            //动态配置的扩展类型为:interface
org.apache.dubbo.common.config.configcenter.DynamicConfigurationFactor
y
            //zookeeper 协议肯定是支持的因为 zookeeper 协议实现了这个动态配置工
厂 ,这个扩展类型为 ZookeeperDynamicConfigurationFactory
            //代码位置在 dubbo-configcenter-zookeeper 包中的
org.apache.dubbo.common.config.configcenter.DynamicConfigurationFactor
y 扩展配置中内容为
zookeeper=org.apache.dubbo.configcenter.support.zookeeper.ZookeeperDyn
amicConfigurationFactory
            supported = supportsExtension(extensionClass, protocol);
            //配置中心走注册中心会打印一条日志
            if (logger.isInfoEnabled()) {
                logger.info(format("No value is configured in the
registry, the %s extension[name : %s] %s as the %s center"
                        , extensionClass.getSimpleName(), protocol,
supported ? "supports" : "does not support", centerType));
            }
        }

                //配置中心走注册中心会打印一条日志
        if (logger.isInfoEnabled()) {
            logger.info(format("The registry[%s] will be %s as the %s
center", registryConfig,
                supported ? "used" : "not used", centerType));
        }
        return supported;
    }
```

这个扩展是否支持的逻辑判断是这样的扫描扩展类，看一下当前扩展类型是否有对应协议的扩展，比如在扩展文件里面这样配置过后是支持的 protocol=xxxImpl 配置中心的动态配置的扩展类型为 org.apache.dubbo.common.config.configcenter.DynamicConfigurationFactory

zookeeper 协议肯定是支持的因为 zookeeper 协议实现了这个动态配置工厂，这个扩展类型为 ZookeeperDynamicConfigurationFactory 代码位置在 dubbo-configcenter-zookeeper 包中的 org.apache.dubbo.common.config.configcenter.DynamicConfigurationFactory 扩展配置中内容为：

```
zookeeper=org.apache.dubbo.configcenter.support.zookeeper.ZookeeperDynamicConfigurationFactory
```

### b) 注册中心配置转配置中心配置

这个逻辑是 registryAsConfigCenter 方法，我来贴一下代码：

```
private ConfigCenterConfig registryAsConfigCenter(RegistryConfig registryConfig) {
                //注册中心协议获取这里例子中的是 zookeeper 协议
        String protocol = registryConfig.getProtocol();
        //注册中心端口 2181
        Integer port = registryConfig.getPort();
        //在 Dubbo 中配置信息 很多情况下都以 URL 形式表示,这里转换后的地址为 zookeeper://127.0.0.1:2181
        URL url = URL.valueOf(registryConfig.getAddress(), registryConfig.getScopeModel());
        //生成当前配置中心的 id 封装之后的内容为:
        //config-center-zookeeper-127.0.0.1-2181
        String id = "config-center-" + protocol + "-" + url.getHost() + "-" + port;
        //配置中心配置对象创建
        ConfigCenterConfig cc = new ConfigCenterConfig();
        //config-center-zookeeper-127.0.0.1-2181
        cc.setId(id);
        cc.setScopeModel(applicationModel);
        if (cc.getParameters() == null) {
            cc.setParameters(new HashMap<>());
        }
        if (CollectionUtils.isNotEmptyMap(registryConfig.getParameters())) {
            cc.getParameters().putAll(registryConfig.getParameters()); // copy the parameters
```

```
        }
        cc.getParameters().put(CLIENT_KEY, registryConfig.getClient());
        //zookeeper
        cc.setProtocol(protocol);
        //2181
        cc.setPort(port);
        if (StringUtils.isNotEmpty(registryConfig.getGroup())) {
            cc.setGroup(registryConfig.getGroup());
        }
        //这个方法转换地址是修复 bug 用的可以看 bug
https://github.com/apache/dubbo/issues/6476
        cc.setAddress(getRegistryCompatibleAddress(registryConfig));
        //注册中心分组做为配置中心命名空间 这里为 null
        cc.setNamespace(registryConfig.getGroup());
        //zk 认证信息
        cc.setUsername(registryConfig.getUsername());
         //zk 认证信息
        cc.setPassword(registryConfig.getPassword());
        if (registryConfig.getTimeout() != null) {
            cc.setTimeout(registryConfig.getTimeout().longValue());
        }
        //这个属性注释中已经建议了已经弃用了默认就是 false 了
        //如果配置中心被赋予最高优先级，它将覆盖所有其他配置，
        cc.setHighestPriority(false);
        return cc;
    }
```

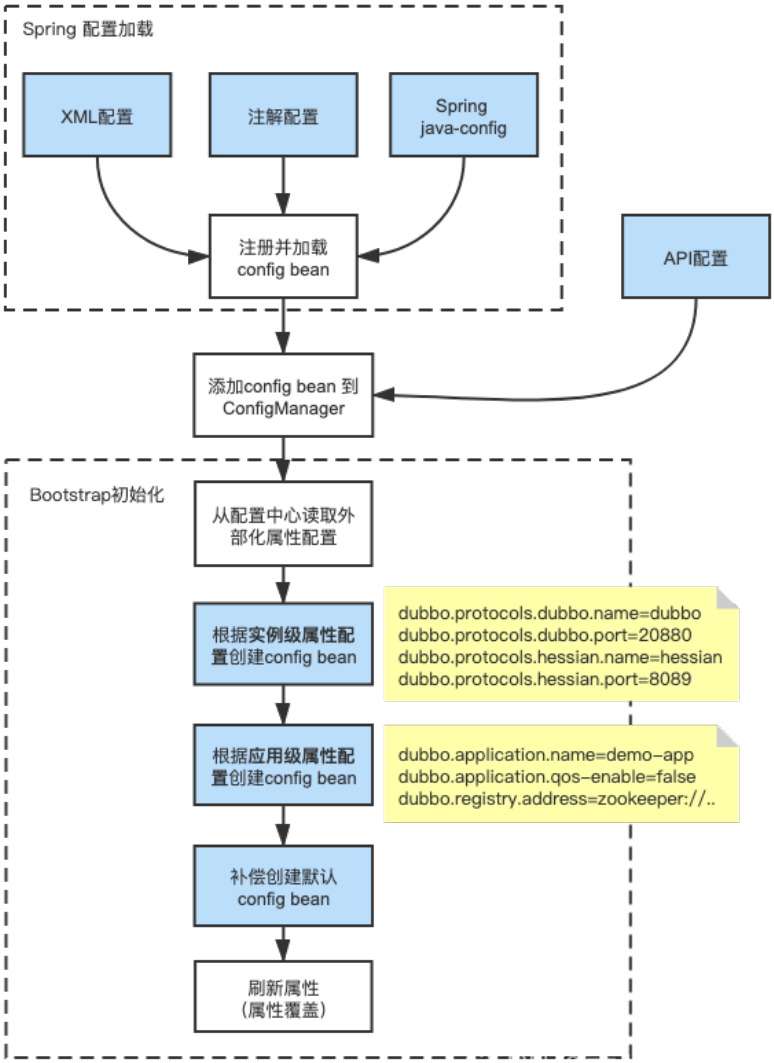
## 3. 配置刷新逻辑

来自 AbstractConfig 类型的 refresh()方法

```
public void refresh() {
        refreshed.set(true);
        try {
            // check and init before do refresh
            //刷新之前执行的逻辑 这里并做什么逻辑
            preProcessRefresh();

                                //获取当前域模型的环境信息对象
            Environment environment =
getScopeModel().getModelEnvironment();
```

```
            List<Map<String, String>> configurationMaps =
environment.getConfigurationMaps();

            // Search props starts with PREFIX in order
            String preferredPrefix = null;
            for (String prefix : getPrefixes()) {
                if
(ConfigurationUtils.hasSubProperties(configurationMaps, prefix)) {
                    preferredPrefix = prefix;
                    break;
                }
            }
            if (preferredPrefix == null) {
                preferredPrefix = getPrefixes().get(0);
            }
            // Extract sub props (which key was starts with
preferredPrefix)
            Collection<Map<String, String>> instanceConfigMaps =
environment.getConfigurationMaps(this, preferredPrefix);
            Map<String, String> subProperties =
ConfigurationUtils.getSubProperties(instanceConfigMaps,
preferredPrefix);
            InmemoryConfiguration subPropsConfiguration = new
InmemoryConfiguration(subProperties);

            if (logger.isDebugEnabled()) {
                String idOrName = "";
                if (StringUtils.hasText(this.getId())) {
                    idOrName = "[id=" + this.getId() + "]";
                } else {
                    String name = ReflectUtils.getProperty(this,
"getName");
                    if (StringUtils.hasText(name)) {
                        idOrName = "[name=" + name + "]";
                    }
                }
                logger.debug("Refreshing " +
this.getClass().getSimpleName() + idOrName +
                    " with prefix [" + preferredPrefix +
                    "], extracted props: " + subProperties);
            }
```

```
            assignProperties(this, environment, subProperties,
subPropsConfiguration);

            // process extra refresh of subclass, e.g. refresh method
configs
            processExtraRefresh(preferredPrefix, subPropsConfiguration);

        } catch (Exception e) {
            logger.error("Failed to override field value of config bean:
" + this, e);
            throw new IllegalStateException("Failed to override field
value of config bean: " + this, e);
        }

        postProcessRefresh();
    }
```




## 4. 配置中心配置大全

ConfigCenterConfig 类型

下面配置信息来自官网

dubbo:config-center 配置

配置中心。对应的配置类： `org.apache.dubbo.config.ConfigCenterConfig`

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 描述 | 兼容性 |
|---|---|---|---|---|---|---|
| protocol | config.protocol | string | 可选 | zookeeper | 使用哪个配置中心：apollo、zookeeper、nacos 等。 以 zookeeper 为例 1. 指定 protocol，则 address 可以简化为 127.0.0.1:2181； 2. 不指定 protocol ， 则 address 取值为 zookeeper://127.0.0.1:2181 | 2.7.0+ |
| address | config.address | string | 必填 | | 配置中心地址。 取值参见 protocol 说明 | 2.7.0+ |
| highest-priority | config.highestPriority | boolean | 可选 | true | 来自配置中心的配置项具有最高优先级，即会覆盖本地配置项。 | 2.7.0+ |
| namespace | config.namespace | string | 可选 | dubbo | 通常用于多租户隔离，实际含义视具体配置中心而不同。 如： zookeeper - 环境隔离，默认值 dubbo； apollo - 区分不同领域的配置集合，默认使用 dubbo 和 application | 2.7.0+ |
| cluster | config.cluster | string | 可选 | | 含义视所选定的配置中心而不同。 如 Apollo 中用来区分不同的配置集群 | 2.7.0+ |
| group | config.group | string | 可选 | dubbo | 含义视所选定的配置中心而不同。 nacos - 隔离不同配置集 zookeeper - 隔离不同配置集 | 2.7.0+ |
| check | config.check | boolean | 可选 | true | 当配置中心连接失败时，是否终止应用启动。 | 2.7.0+ |
| config-file | config.configFile | string | 可选 | dubbo.properties | 全局级配置文件所映射到的 key zookeeper - 默认路径 /dubbo/config/dubbo/dubbo.properties apollo - dubbo namespace 中的 dubbo.properties 键 | 2.7.0+ |
| timeout | config.timeout | integer | | 3000ms | 获取配置的超时时间 | 2.7.0+ |
| username | | string | | | 如果配置中心需要做校验，用户名 Apollo 暂未启用 | 2.7.0+ |
| password | | string | | | 如果配置中心需要做校验，密码 Apollo 暂未启用 | 2.7.0+ |
| parameters | | Map<string, string> | | | 扩展参数，用来支持不同配置中心的定制化配置参数 | 2.7.0+ |
| include-spring-env | | boolean | 可选 | false | 使用 Spring 框架时支持，为 true 时，会自动从 Spring Environment 中读取配置。 默认依次读取 key 为 dubbo.properties 的配置 key 为 dubbo.properties 的 PropertySource | 2.7.0+ |

# 二、 配置加载全解析

## 1. 回到启动器的初始化过程

在应用程序启动的时候会调用发布器的启动方法，然后调用初始化方法，在发布器 DefaultApplicationDeployer 中的初始化方法 initialize()如下：

```
@Override
    public void initialize() {
        if (initialized) {
            return;
        }
        // Ensure that the initialization is completed when concurrent calls
        synchronized (startLock) {
            if (initialized) {
                return;
            }
            // register shutdown hook
            registerShutdownHook();

            startConfigCenter();

            loadApplicationConfigs();

            initModuleDeployers();

            // @since 2.7.8
            startMetadataCenter();

            initialized = true;

            if (logger.isInfoEnabled()) {
                logger.info(getIdentifier() + " has been initialized!");
            }
        }
    }
```

初始化过程中会先启动配置中心配置信息处理，然后调用加载初始化应用程序配置方法 loadApplicationConfigs();进行配置加载。

Dubbo 框架的配置项比较繁多，为了更好地管理各种配置，将其按照用途划分为不同的组件，最终所有配置项都会汇聚到 URL 中，传递给后续处理模块。

**常用配置组件如下**

- Application：Dubbo 应用配置
- registry：注册中心
- protocol：服务提供者 RPC 协议
- config-center：配置中心

- metadata-report：元数据中心
- service：服务提供者配置
- reference：远程服务引用配置
- provider：service 的默认配置或分组配置
- consumer：reference 的默认配置或分组配置
- module：模块配置
- monitor：监控配置
- metrics：指标配置
- ssl：SSL/TLS 配置

配置还有几个比较重要的点：

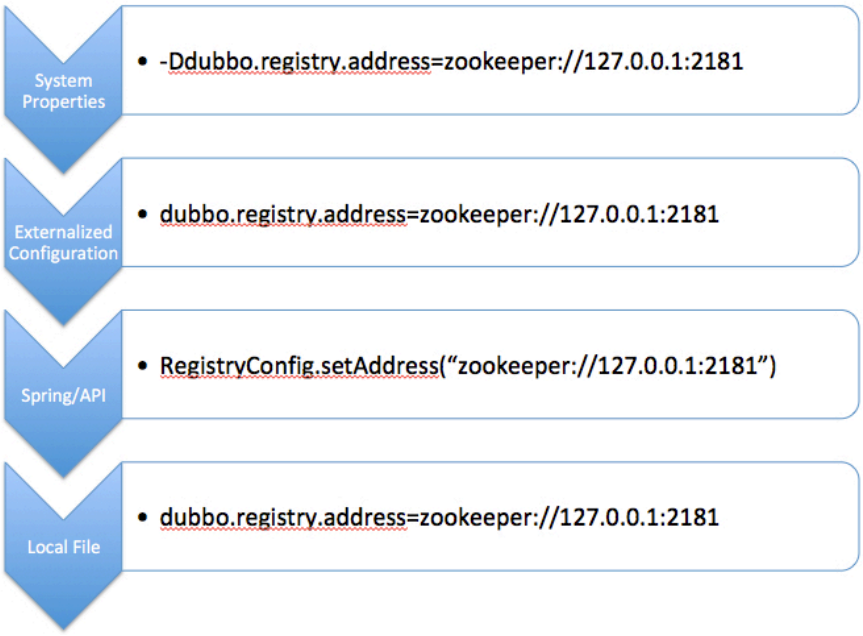
**配置来源**

从 Dubbo 支持的配置来源说起，默认有 6 种配置来源：

- JVM System Properties，JVM-D 参数
- System environment，JVM 进程的环境变量
- Externalized Configuration，外部化配置，从配置中心读取
- Application Configuration，应用的属性配置，从 Spring 应用的 Environment 中提取“dubbo”打头的属性集
- API/XML/注解等编程接口采集的配置可以被理解成配置来源的一种，是直接面向用户编程的配置采集方式
- 从 classpath 读取配置文件 dubbo.properties

**覆盖关系**

下图展示了配置覆盖关系的优先级，从上到下优先级依次降低：




**配置方式**

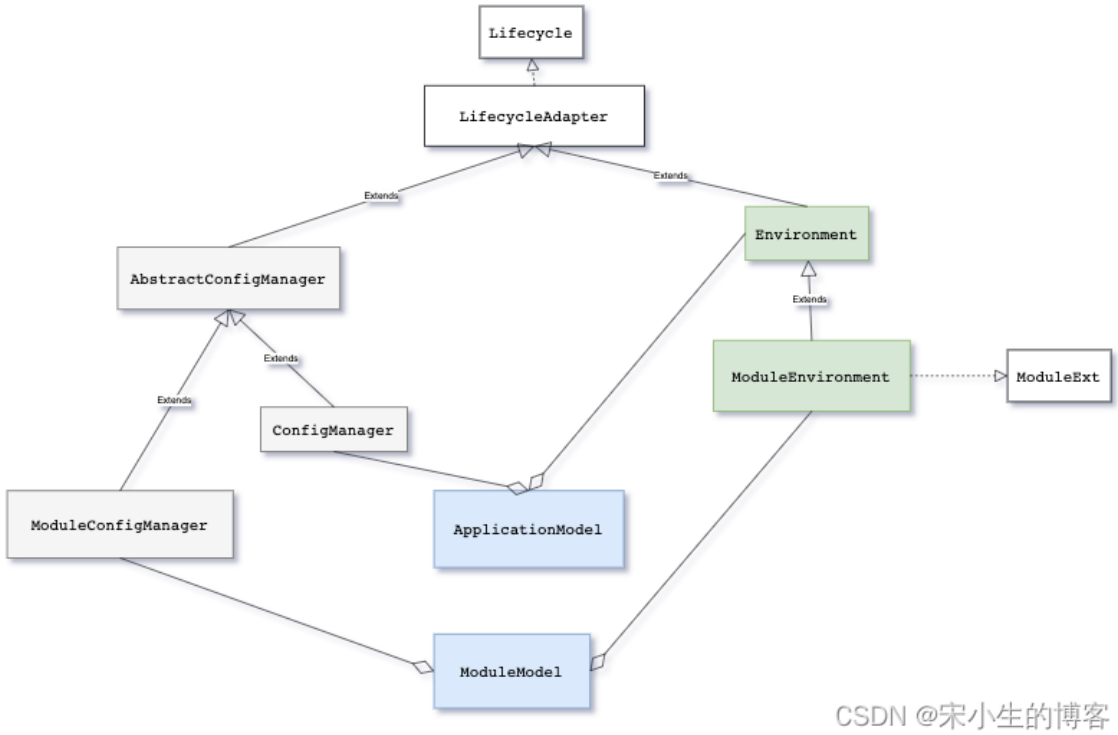
- Java API 配置
- XML 配置
- Annotation 配置
- 属性配置

配置虽然非常多，但是我们掌握一下配置加载的原理，再了解下官网的文档说明路径应该基础的配置搞定是没问题的，更深入的配置很多参数还是需要了解下源码的。

## 2. 配置信息的初始化回顾

前面我们在讲 ModuleModel 对象的创建的时候 ModuleModel 模型中包含了一个成员变量为 ModuleEnvironment 代表当前的模块环境和 ModuleConfigManager 配置管理器，而 ModuleModel 模型对象的父模型对象 ApplicationModel 中包含了一个成员变量 Environment 环境和 ConfigManager 配置管理器。

在回顾调用过程之前我们先看下模型，配置管理器和环境与配置之间的关系如下图：




在 ModuleConfigManager 对象初始化方法 initialize()中创建了模块配置管理器：ModuleConfigManager

如下代码所示：

```
@Override
  protected void initialize() {
      super.initialize();
      this.serviceRepository = new ModuleServiceRepository(this);
      this.moduleConfigManager = new ModuleConfigManager(this);
      this.moduleConfigManager.initialize();
```

ModuleEnvironment 环境信息对象也会在配置管理器创建的时候被调用到。

如下代码所示：

```
@Override
  public ModuleEnvironment getModelEnvironment() {
      if (moduleEnvironment == null) {
          moduleEnvironment = (ModuleEnvironment) this.getExtensionLoader(ModuleExt.class)
              .getExtension(ModuleEnvironment.NAME);
      }
      return moduleEnvironment;
  }
```

在扩展对象 ExtensionLoader 进行对象 ModuleEnvironment 创建之后会对对象进行初始化调用 initExtension（instance）方法初始化的时候调用如下代码。

ExtensionLoader 中的初始化方法如下：

```
private void initExtension(T instance) {
      if (instance instanceof Lifecycle) {
          Lifecycle lifecycle = (Lifecycle) instance;
          lifecycle.initialize();
      }
  }
```

## 3. 属性加载

### 1) Environment 中属性的初始化方法

这个初始化方法对应 ModuleEnvironment 的父类型 Environment 中的初始化方法如下：initialize()

```
@Override
    public void initialize() throws IllegalStateException {
        if (initialized.compareAndSet(false, true)) {
          //加载在JVM或者环境变量指定的dubbo.properties配置文件 配置的key为dubbo.properties.file ，
如果未指定则查找类路径下的dubbo.properties
            this.propertiesConfiguration = new PropertiesConfiguration(scopeModel);
            //系统JVM参数的配置无需我们来加载到内存,系统已经加载好了放到了System中,我们只需
System.getProperty(key)来调用
            this.systemConfiguration = new SystemConfiguration();
            //系统环境变量的配置,无需我们来加载到内存,系统已经加载好了放到了System中,我们只需
System.getenv(key)来获取就可以
            this.environmentConfiguration = new EnvironmentConfiguration();
            //从远程配置中心的全局配置获取对应配置
            this.externalConfiguration = new InmemoryConfiguration("ExternalConfig");
            //从远程配置中心的应用配置获取对应配置
            this.appExternalConfiguration = new InmemoryConfiguration("AppExternalConfig");
            //应用内的配置比如:  Spring Environment/PropertySources/application.properties
            this.appConfiguration = new InmemoryConfiguration("AppConfig");
      //加载迁移配置,用户在JVM参数或者环境变量中指定的dubbo.migration.file,如果用户未指定测尝试加载类路
径下的dubbo-migration.yaml
            loadMigrationRule();
        }
    }
```

## 2) 属性变量说明

前面我们已经基本上介绍了各个属性的含义下面用一个表格列举一下方便查看：

| 属性变量名 | 属性类型 | 说明 |
| --- | --- | --- |
| propertiesConfiguration | PropertiesConfiguration | dubbo.properties 文件中的属性 |
| systemConfiguration | SystemConfiguration | JVM 参数 启动进程时指定的（-D）配置 |
| environmentConfiguration | EnvironmentConfiguration | 环境变量中的配置 |
| externalConfiguration | InmemoryConfiguration | 外部配置全局配置 例如配置中心中 config-center global/default config |
| appExternalConfiguration | InmemoryConfiguration | 外部的应用配置 例如配置中心中执行的当前应用的配置 config-center app config |
| appConfiguration | InmemoryConfiguration | 来自应用中的配置例如:Spring Environment/PropertySources/application.properties |
| globalConfiguration | CompositeConfiguration | 前面 6 个配置属性放到一起就是这个 |
| globalConfigurationMaps | List<Map<String, String>> | 最前面的 6 个属性转换为 map 放到一起就是这个可以理解为将全局配置 globalConfiguration 转换成了列表 这个列表顺序在这里是 :SystemConfiguration -> EnvironmentConfiguration -> AppExternalConfiguration -> ExternalConfiguration -> AppConfiguration -> AbstractConfig -> PropertiesConfiguration |
| defaultDynamicGlobalConfiguration | CompositeConfiguration | 这个也是一个组合配置将 defaultDynamicConfiguration 动态配置（来自配置中心的配置）和全局配置添加到了自己的配置列表中，列表顺序为 defaultDynamicConfiguration -> globalConfiguration |
| localMigrationRule | String | ,用户在 JVM 参数或者环境变量中指定的 dubbo.migration.file,如果用户未指定测尝试加载类路径下的 dubbo-migration.yaml |

关于每个配置信息这里还是来了解下细节，方便大家了解原理。

### 3) dubbo.properties 配置文件加载解析原理

如前面所示：

```
//加载在JVM或者环境变量指定的dubbo.properties配置文件 配置的key为dubbo.properties.file ,如果未指定则查找类路径下的dubbo.properties
            this.propertiesConfiguration = new PropertiesConfiguration(scopeModel);
```

下面就直接提构造器的 PropertiesConfiguration 代码了：

```
public PropertiesConfiguration(ScopeModel scopeModel) {
        this.scopeModel = scopeModel;
        refresh();
    }

    public void refresh() {
      //配置获取的过程是借助工具类ConfigUtils来获取的
        properties = ConfigUtils.getProperties(scopeModel.getClassLoaders());
    }
```

继续看 ConfigUtils 的 getProperties 方法：

```
public static Properties getProperties(Set<ClassLoader> classLoaders) {
    //这个配置的KEY是dubbo.properties.file System.getProperty是从JVM参数中获取配置的 一般情况下我们在启动Java进程的时候会指定Dubbo配置文件 如配置：
    //-Ddubbo.properties.file=/dubbo.properties
        String path = System.getProperty(CommonConstants.DUBBO_PROPERTIES_KEY);

        if (StringUtils.isEmpty(path)) {
        //优先级最高的JVM参数拿不到数据则从 环境变量中获取,这个配置key也是dubbo.properties.file System.getenv是从环境变量中获取数据
        //例如我们在环境变量中配置 dubbo.properties.file=/dubbo.properties
            path = System.getenv(CommonConstants.DUBBO_PROPERTIES_KEY);
            if (StringUtils.isEmpty(path)) {
              //如果在JVM参数和环境变量都拿不到这个配置文件的路径我们就用默认的吧
              //默认的路径是类路径下的资源文件 这个路径是：dubbo.properties
                path = CommonConstants.DEFAULT_DUBBO_PROPERTIES;
            }
        }
        return ConfigUtils.loadProperties(classLoaders, path, false, true);
    }
```

路径获取之后加载详细的配置内容。

ConfigUtils 的 loadProperties 代码如下：

```
ConfigUtils.loadProperties(classLoaders, path, false, true);
```

代码如下：

```
public static Properties loadProperties(Set<ClassLoader> classLoaders,
String fileName, boolean allowMultiFile, boolean optional) {
       Properties properties = new Properties();
       // add scene judgement in windows environment Fix 2557
       //检查文件是否存在 直接加载配置文件如果加载到了配置文件则直接返回
       if (checkFileNameExist(fileName)) {
           try {
               FileInputStream input = new FileInputStream(fileName);
               try {
                   properties.load(input);
               } finally {
                   input.close();
               }
           } catch (Throwable e) {
               logger.warn("Failed to load " + fileName + " file from "
+ fileName + "(ignore this file): " + e.getMessage(), e);
           }
           return properties;
       }

                   //为什么会有下面的逻辑呢,如果仅仅使用上面的加载方式只能加载到本系
统下的配置文件,无法加载封装在 jar 中的根路径的配置
       Set<java.net.URL> set = null;
       try {
           List<ClassLoader> classLoadersToLoad = new LinkedList<>();
           classLoadersToLoad.add(ClassUtils.getClassLoader());
           classLoadersToLoad.addAll(classLoaders);
           //这个方法 loadResources 在扩展加载的时候说过
           set = ClassLoaderResourceLoader.loadResources(fileName,
classLoadersToLoad).values().stream().reduce(new LinkedHashSet<>(),
(a, i) -> {
               a.addAll(i);
```

```
                return a;
            });
        } catch (Throwable t) {
            logger.warn("Fail to load " + fileName + " file: " +
t.getMessage(), t);
        }

        if (CollectionUtils.isEmpty(set)) {
            if (!optional) {
                logger.warn("No " + fileName + " found on the class
path.");
            }
            return properties;
        }

        if (!allowMultiFile) {
            if (set.size() > 1) {
                String errMsg = String.format("only 1 %s file is
expected, but %d dubbo.properties files found on class path: %s",
                        fileName, set.size(), set);
                logger.warn(errMsg);
            }

            // fall back to use method getResourceAsStream
            try {

properties.load(ClassUtils.getClassLoader().getResourceAsStream(fileNa
me));
            } catch (Throwable e) {
                logger.warn("Failed to load " + fileName + " file from "
+ fileName + "(ignore this file): " + e.getMessage(), e);
            }
            return properties;
        }

        logger.info("load " + fileName + " properties file from " +
set);

        for (java.net.URL url : set) {
            try {
                Properties p = new Properties();
                InputStream input = url.openStream();
                if (input != null) {
```

```
                try {
                    p.load(input);
                    properties.putAll(p);
                } finally {
                    try {
                        input.close();
                    } catch (Throwable t) {
                    }
                }
            }
        } catch (Throwable e) {
            logger.warn("Fail to load " + fileName + " file from " +
url + "(ignore this file): " + e.getMessage(), e);
        }
    }

    return properties;
}
```

完整的配置加载流程这里用简单的话描述下：

- **项目内配置查询**

路径查询

- 从 JVM 参数中获取配置的 dubbo.properties.file 配置文件路径
- 如果前面未获取到路径则从环境变量参数中获取配置的 dubbo.properties.file 配置文件路径
- 如果前面未获取到路径则使用默认路径 dubbo.propertie

配置加载

- 将路径转为 FileInputStream 然后使用 Properties 加载
- 依赖中的配置扫描查询
- 使用类加载器扫描所有资源 URL
- url 转 InputStream 如 url.openStream()然后使用 Properties 加载

- **依赖中的配置扫描查询**
  - 使用类加载器扫描所有资源 URL
  - url 转 InputStream 如 url.openStream()然后使用 Properties 加载

#### 4) 加载 JVM 参数的配置

这里我们继续看 SystemConfiguration 配置的加载。这个直接看下代码就可以了。

这个类型仅仅是使用 System.getProperty 来获取 JVM 配置即可。

```
public class SystemConfiguration implements Configuration {

    @Override
    public Object getInternalProperty(String key) {
        return System.getProperty(key);
    }

    public Map<String, String> getProperties() {
        return (Map) System.getProperties();
    }
}
```

#### 5) 加载环境变量参数的配置

这里我们来看 EnvironmentConfiguration，这里我们直接来看代码：

```
public class EnvironmentConfiguration implements Configuration {

    @Override
    public Object getInternalProperty(String key) {
        String value = System.getenv(key);
        if (StringUtils.isEmpty(value)) {
            value = System.getenv(StringUtils.toOSStyleKey(key));
        }
        return value;
    }

    public Map<String, String> getProperties() {
        return System.getenv();
    }
}
```

### 6) 内存配置的封装:InmemoryConfiguration

这里我们看下 InmemoryConfiguration 的设计，这个直接看代码吧内部使用了一个 LinkedHashMap 来存储配置。

```
public class InmemoryConfiguration implements Configuration {

    private String name;

    // stores the configuration key-value pairs
    private Map<String, String> store = new LinkedHashMap<>();

    public InmemoryConfiguration() {
    }

    public InmemoryConfiguration(String name) {
        this.name = name;
    }

    public InmemoryConfiguration(Map<String, String> properties) {
        this.setProperties(properties);
    }

    @Override
    public Object getInternalProperty(String key) {
        return store.get(key);
    }

    /**
     * Add one property into the store, the previous value will be
replaced if the key exists
     */
    public void addProperty(String key, String value) {
        store.put(key, value);
    }

    /**
     * Add a set of properties into the store
     */
    public void addProperties(Map<String, String> properties) {
        if (properties != null) {
```

```
            this.store.putAll(properties);
        }
    }

    /**
     * set store
     */
    public void setProperties(Map<String, String> properties) {
        if (properties != null) {
            this.store = properties;
        }
    }

    public Map<String, String> getProperties() {
        return store;
    }

}
```

### 7) Dubbo 迁移新版本的配置文件加载 dubbo-migration.yaml

关于配置迁移文件的用法可以看下这个 Dubbo 官方的地址迁移规则说明。

这个配置文件的文件名字为：dubbo-migration.yaml

这个和 14.3.4 加载 JVM 参数配置的过程是相似的细节可以看 14.3.4 节。

```
  private void loadMigrationRule() {
    //JVM参数的dubbo.migration.file配置
        String path = System.getProperty(CommonConstants.DUBBO_MIGRATION_KEY);
        if (StringUtils.isEmpty(path)) {
        //环境变量的dubbo.migration.file配置
            path = System.getenv(CommonConstants.DUBBO_MIGRATION_KEY);
            if (StringUtils.isEmpty(path)) {
            //默认的迁移配置文件 dubbo-migration.yaml
                path = CommonConstants.DEFAULT_DUBBO_MIGRATION_FILE;
            }
        }
        this.localMigrationRule =
ConfigUtils.loadMigrationRule(scopeModel.getClassLoaders(), path);
    }
```

## 4. 初始化加载应用配置

加载配置涉及到了配置优先级的处理，下面来看加载配置代码loadApplicationConfigs()方法。

```
private void loadApplicationConfigs() {
    //发布器还是不处理配置加载的逻辑还是交给配置管理器
        configManager.loadConfigs();
    }
```

配置管理器加载配置：

```
@Override
    public void loadConfigs() {
        // application config has load before starting config center
        // load dubbo.applications.xxx
        //加载应用配置
        loadConfigsOfTypeFromProps(ApplicationConfig.class);

        // load dubbo.monitors.xxx
        //加载监控配置
        loadConfigsOfTypeFromProps(MonitorConfig.class);

        // load dubbo.metrics.xxx
        //加载指标监控配置
        loadConfigsOfTypeFromProps(MetricsConfig.class);

        // load multiple config types:
        // load dubbo.protocols.xxx
        //加载协议配置
        loadConfigsOfTypeFromProps(ProtocolConfig.class);

        // load dubbo.registries.xxx
        loadConfigsOfTypeFromProps(RegistryConfig.class);

        // load dubbo.metadata-report.xxx
        //加载元数据配置
        loadConfigsOfTypeFromProps(MetadataReportConfig.class);

        // config centers has bean loaded before starting config center
        //loadConfigsOfTypeFromProps(ConfigCenterConfig.class);
```

```
                //刷新配置
        refreshAll();

                //检查配置
        checkConfigs();

        // set model name
        if (StringUtils.isBlank(applicationModel.getModelName())) {

applicationModel.setModelName(applicationModel.getApplicationName());
        }
    }
```

## 三、 元数据中心源码解析

### 1. 简介

关于元数据中心的概念对于大部分用户来说是比较陌生的，配置中心的话我们还好理解，对于元数据中心是什么，我们来看下我从官网拷贝过来的一段文字。

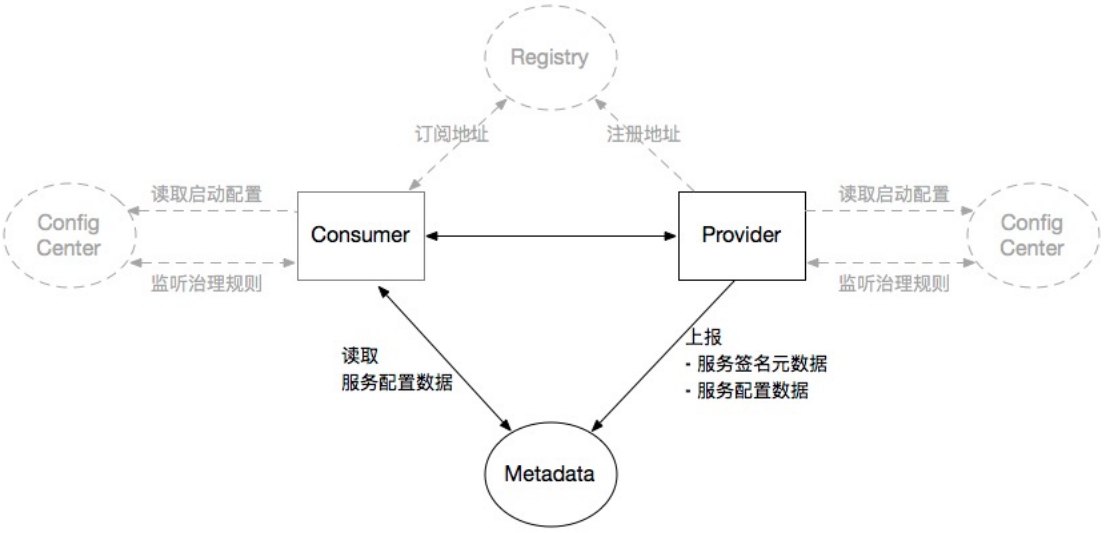
元数据中心在 2.7.x 版本开始支持，随着应用级别的服务注册和服务发现在 Dubbo 中落地,元数据中心也变的越来越重要。在以下几种情况下会需要部署元数据中心：

- 对于一个原先采用老版本 Dubbo 搭建的应用服务，在迁移到 Dubbo 3 时，Dubbo 3 会需要一个元数据中心来维护 RPC 服务与应用的映射关系（即接口与应用的映射关系），因为如果采用了应用级别的服务发现和服务注册，在注册中心中将采用“应用——实例列表”结构的数据组织形式，不再是以往的“接口——实例列表”结构的数据组织形式，而以往用接口级别的服务注册和服务发现的应用服务在迁移到应用级别时，得不到接口与应用之间的对应关系，从而无法从注册中心得到实例列表信息，所以 Dubbo 为了兼容这种场景，在 Provider 端启动时，会往元数据中心存储接口与应用的映射关系。

- 为了让注册中心更加聚焦与地址的发现和推送能力，减轻注册中心的负担，元数据中心承载了所有的服务元数据、大量接口/方法级别配置信息等，无论是接口粒度还是应用粒度的服务发现和注册，元数据中心都起到了重要的作用。

- 如果有以上两种需求，都可以选择部署元数据中心，并通过 Dubbo 的配置来集成该元数据中心。

元数据中心并不依赖于注册中心和配置中心，用户可以自由选择是否集成和部署元数据中心，如下图所示：




该图中不配备配置中心，意味着可以不需要全局管理配置的能力。该图中不配备注册中心，意味着可能采用了 Dubbo mesh 的方案，也可能不需要进行服务注册，仅仅接收直连模式的服务调用。

官网参考文章地址：

- 部署架构（注册中心 配置中心 元数据中心）
- 元数据参考手册

综上所述可以用几句话概括下：

- 元数据中心来维护 RPC 服务与应用的映射关系（即接口与应用的映射关系）来兼容接口与应用之间的对应关系。

- 让注册中心更加聚焦与地址的发现和推送能力。

注册中心的启动是在 DefaultApplicationDeployer 中的初始化方法 initialize()中:如下所示，这里只看下 startMetadataCenter();方法即可。

```
@Override
    public void initialize() {
        if (initialized) {
            return;
        }
        // Ensure that the initialization is completed when concurrent calls
        synchronized (startLock) {
            if (initialized) {
                return;
            }
            // register shutdown hook
            registerShutdownHook();

            startConfigCenter();

            loadApplicationConfigs();

            initModuleDeployers();

            // @since 2.7.8
            startMetadataCenter();

            initialized = true;

            if (logger.isInfoEnabled()) {
                logger.info(getIdentifier() + " has been initialized!");
            }
        }
    }
```

## 2. 深入探究元数据中心的启动过程

### 1) 启动元数据中心的代码全貌

关于元数据中心我们看下 startMetadataCenter()方法来大致了解下整个流程。

```
private void startMetadataCenter() {
                //如果未配置元数据中心的地址等配置则使用注册中心的地址等配置做为元数据中心的配置
        useRegistryAsMetadataCenterIfNecessary();
                //获取应用的配置信息
```

```
        ApplicationConfig applicationConfig = getApplication();
                //元数据配置类型 元数据类型， local 或 remote,, 如果选择远程，
则需要进一步指定元数据中心
        String metadataType = applicationConfig.getMetadataType();
        // FIXME, multiple metadata config support.
        //查询元数据中心的地址等配置
        Collection<MetadataReportConfig> metadataReportConfigs =
configManager.getMetadataConfigs();

        if (CollectionUtils.isEmpty(metadataReportConfigs)) {
        //这个就是判断 如果选择远程，则需要进一步指定元数据中心 否则就抛出来异常
            if (REMOTE_METADATA_STORAGE_TYPE.equals(metadataType)) {
                throw new IllegalStateException("No MetadataConfig found,
Metadata Center address is required when 'metadata=remote' is
enabled.");
            }
            return;
        }
                //MetadataReport 实例的存储库对象获取
        MetadataReportInstance metadataReportInstance =
applicationModel.getBeanFactory().getBean(MetadataReportInstance.class
);
        List<MetadataReportConfig> validMetadataReportConfigs = new
ArrayList<>(metadataReportConfigs.size());
        for (MetadataReportConfig metadataReportConfig :
metadataReportConfigs) {

ConfigValidationUtils.validateMetadataConfig(metadataReportConfig);
            validMetadataReportConfigs.add(metadataReportConfig);
        }
        //初始化元数据
        metadataReportInstance.init(validMetadataReportConfigs);
        //MetadataReport 实例的存储库对象初始化失败则抛出异常
        if (!metadataReportInstance.inited()) {
            throw new IllegalStateException(String.format("%s
MetadataConfigs found, but none of them is valid.",
metadataReportConfigs.size()));
        }
    }
```

### 2) 元数据中心未配置则使用注册中心配置

前面在说配置中心的时候有说过配置中心如果未配置会使用注册中心的地址等信息作为默认配置，这里元数据做了类似的操作，如代码：DefaultApplicationDeployer类型的 useRegistryAsMetadataCenterIfNecessary()方法。

```
private void useRegistryAsMetadataCenterIfNecessary() {
                //配置缓存中查询元数据配置
        Collection<MetadataReportConfig> metadataConfigs =
configManager.getMetadataConfigs();
                //配置存在则直接返回
        if (CollectionUtils.isNotEmpty(metadataConfigs)) {
            return;
        }
                ////查询是否有注册中心设置了默认配置 isDefault 设置为 true 的注
册中心则为默认注册中心列表,如果没有注册中心设置为默认注册中心,则获取所有未设置默认配
置的注册中心列表
        List<RegistryConfig> defaultRegistries =
configManager.getDefaultRegistries();
        if (defaultRegistries.size() > 0) {
        //多注册中心遍历
            defaultRegistries
                .stream()
                //筛选符合条件的注册中心 (筛选逻辑就是查看是否有对应协议的扩展支持)
                .filter(this::isUsedRegistryAsMetadataCenter)
                //注册中心配置映射为元数据中心  映射就是获取需要的配置
                .map(this::registryAsMetadataCenter)
                //将元数据中心配置存储在配置缓存中方便后续使用
                .forEach(metadataReportConfig -> {

                    if (metadataReportConfig.getId() == null) {
                        Collection<MetadataReportConfig>
metadataReportConfigs = configManager.getMetadataConfigs();
                        if
(CollectionUtils.isNotEmpty(metadataReportConfigs)) {
                            for (MetadataReportConfig existedConfig :
metadataReportConfigs) {
                                if (existedConfig.getId() == null &&
existedConfig.getAddress().equals(metadataReportConfig.getAddress()))
{
                                    return;
```

```
                            }
                        }
                    }

configManager.addMetadataReport(metadataReportConfig);
                } else {
                    Optional<MetadataReportConfig> configOptional =
configManager.getConfig(MetadataReportConfig.class,
metadataReportConfig.getId());
                    if (configOptional.isPresent()) {
                        return;
                    }

configManager.addMetadataReport(metadataReportConfig);
                }
                logger.info("use registry as metadata-center: " +
metadataReportConfig);
            });
        }
    }
```

这个代码有些细节就不细说了，我们概括下顺序梳理下思路：

- 配置缓存中查询元数据配置，配置存在则直接返回
- 查询所有可用的默认注册中心列表
    - 多注册中心遍历
    - 选符合条件的注册中心（筛选逻辑就是查看是否有对应协议的扩展支持）
    - 注册中心配置 RegistryConfig 映射转换为元数据中心配置类型 MetadataReportConfig 映射就是获取需要的配置
    - 将元数据中心配置存储在配置缓存中方便后续使用

元数据的配置可以参考官网：元数据参考手册

这里主要看下可配置项有哪些，对应类型为 MetadataReportConfig 在官网暂时未找到合适的文档，这里整理下属性列表方便后续配置说明查看：

| 配置变量 | 类型 | 说明 |
| --- | --- | --- |
| id | String | 配置 id |
| protocol | String | 元数据协议 |
| address | String | 元数据中心地址 |
| port | Integer | 元数据中心端口 |
| username | String | 元数据中心认证用户名 |
| password | String | 元数据中心认证密码 |
| timeout | Integer | 元数据中心的请求超时（毫秒） |
| group | String | 该组将元数据保存在中。它与注册表相同 |
| parameters | Map<String, String> | 自定义参数 |
| retryTimes | Integer | 重试次数 |
| retryPeriod | Integer | 重试间隔 |
| cycleReport | Boolean | 默认情况下， 是否每天重复存储完整的元数据 |
| syncReport | Boolean | Sync or Async report. |
| cluster | Boolean | 需要群集支持，默认为 false |
| registry | String | 注册表配置 id |
| file | String | 元数据报告文件存储位置 |
| check | Boolean | 连接到元数据中心时要应用的失败策略 |

### 3) 元数据中心的初始化逻辑

a) 元数据中心的初始化调用逻辑

主要看这一行比较重要的逻辑：

```
//初始化元数据
metadataReportInstance.init(validMetadataReportConfigs);
```

在了解这一行逻辑之前我们先来看下元数据相关联的类型：

MetadataReportInstance 中的初始化方法 init

```
public void init(List<MetadataReportConfig> metadataReportConfigs) {
                //CAS 判断是否有初始化过
        if (!init.compareAndSet(false, true)) {
            return;
        }
                //元数据类型配置如果未配置则默认为 local
        this.metadataType =
applicationModel.getApplicationConfigManager().getApplicationOrElseThr
ow().getMetadataType();
```

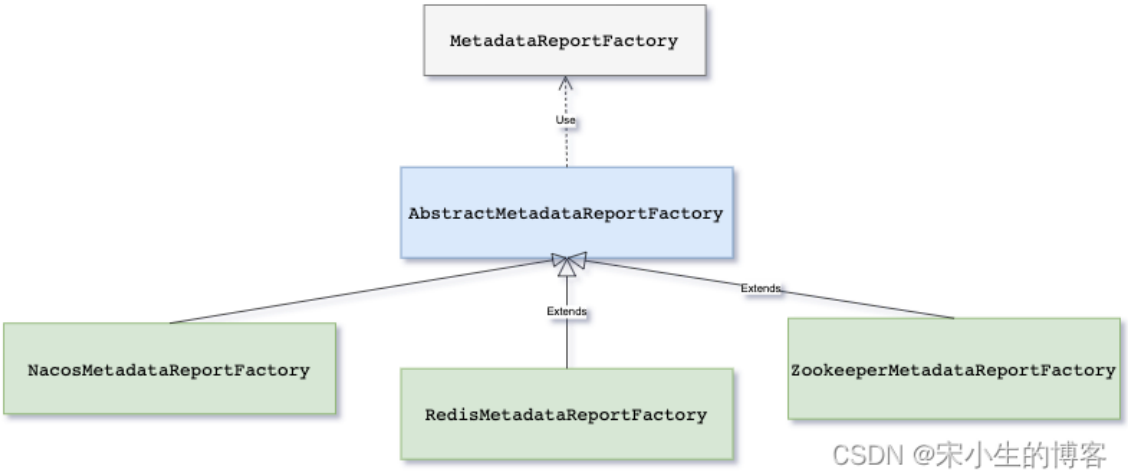
```
        if (metadataType == null) {
            this.metadataType = DEFAULT_METADATA_STORAGE_TYPE;
        }
                //获取 MetadataReportFactory 工厂类型
        MetadataReportFactory metadataReportFactory =
applicationModel.getExtensionLoader(MetadataReportFactory.class).getAd
aptiveExtension();
         //多元数据中心初始化
        for (MetadataReportConfig metadataReportConfig :
metadataReportConfigs) {
            init(metadataReportConfig, metadataReportFactory);
        }
    }

    private void init(MetadataReportConfig config,
MetadataReportFactory metadataReportFactory) {
    //配置转 url
        URL url = config.toUrl();
        if (METADATA_REPORT_KEY.equals(url.getProtocol())) {
            String protocol = url.getParameter(METADATA_REPORT_KEY,
DEFAULT_DIRECTORY);
            url = URLBuilder.from(url)
                    .setProtocol(protocol)
                    .setScopeModel(config.getScopeModel())
                    .removeParameter(METADATA_REPORT_KEY)
                    .build();
        }
        url = url.addParameterIfAbsent(APPLICATION_KEY,
applicationModel.getCurrentConfig().getName());
        String relatedRegistryId = isEmpty(config.getRegistry()) ?
(isEmpty(config.getId()) ? DEFAULT_KEY : config.getId()) :
config.getRegistry();
                //从元数据工厂中获取元数据
        MetadataReport metadataReport =
metadataReportFactory.getMetadataReport(url);
        //缓存元数据到内存
        if (metadataReport != null) {
            metadataReports.put(relatedRegistryId, metadataReport);
        }
    }
```

关于元数据的初始化我们主要看两个位置：

- 一个是元数据工厂对象的创建与初始化 MetadataReportFactory
- 一个是元数据对象的创建与初始化 MetadataReport

b) 元数据工厂对象 MetadataReportFactory

关于元数据工厂类型 MetadataReportFactory，元数据工厂，用于创建与管理元数据对象，相关类型如下：




我们这里主要以为 Zookeeper 扩展的元数据工厂 ZookeeperMetadataReportFactory 类型为例子。

实现类型逻辑不复杂，这里就直接贴代码看看：

```
public class ZookeeperMetadataReportFactory extends AbstractMetadataReportFactory {
  //与Zookeeper交互的传输器
    private ZookeeperTransporter zookeeperTransporter;
  //应用程序模型
    private ApplicationModel applicationModel;

    public ZookeeperMetadataReportFactory(ApplicationModel applicationModel) {
        this.applicationModel = applicationModel;
        this.zookeeperTransporter = ZookeeperTransporter.getExtension(applicationModel);
    }

    @DisableInject
    public void setZookeeperTransporter(ZookeeperTransporter zookeeperTransporter) {
        this.zookeeperTransporter = zookeeperTransporter;
    }

    @Override
    public MetadataReport createMetadataReport(URL url) {
        return new ZookeeperMetadataReport(url, zookeeperTransporter);
    }

}
```

元数据工厂的实现比较简单

- 继承抽象的元数据工厂 AbstractMetadataReportFactory
- 实现工厂方法 createMetadataReport 来创建一个元数据操作类型

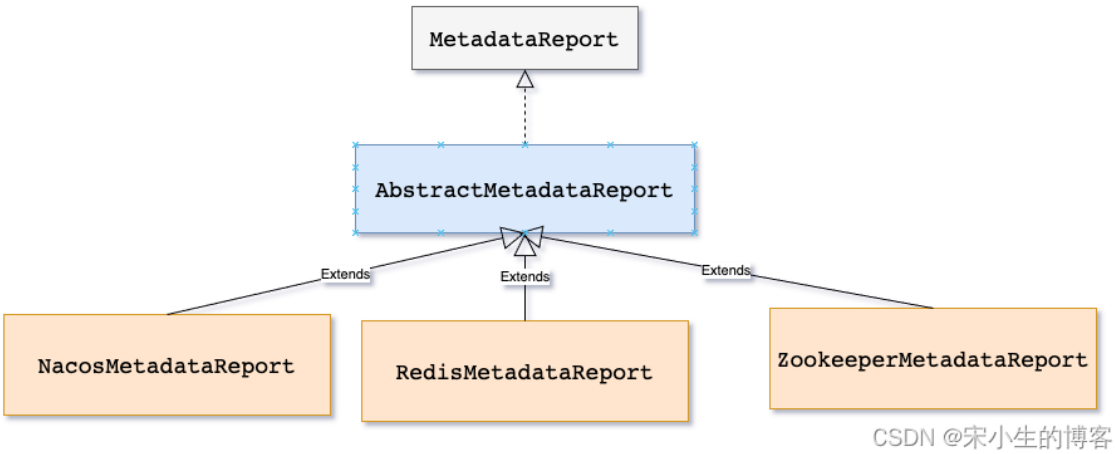
如果我们想要实现一个元数据工厂扩展可以参考 Zookeeper 的这个方式

c) 元数据操作对象 MetadataReport 的创建与初始化

前面的从元数据工厂中获取元数据操作对象的逻辑处理代码如下：

```
//从元数据工厂中获取元数据 ,url对象可以理解为配置
 MetadataReport metadataReport = metadataReportFactory.getMetadataReport(url);
```

关于元数据对象，用于元数据信息的增删改查等逻辑的操作与元数据信息的缓存




我们这里还是以 Zookeeper 的实现 ZookeeperMetadataReportFactory 类型做为参考。

我们先来看这个逻辑

```
//从元数据工厂中获取元数据 ,url对象可以理解为配置
 MetadataReport metadataReport = metadataReportFactory.getMetadataReport(url);
```

ZookeeperMetadataReportFactory 的父类型 AbstractMetadataReportFactory 中的 getMetadataReport 方法如下：

```
@Override
    public MetadataReport getMetadataReport(URL url) {
    //url 值参考例子 zookeeper://127.0.0.1:2181?application=dubbo-demo-
api-provider&client=&port=2181&protocol=zookeeper
                //如果存在 export 则移除
        url = url.setPath(MetadataReport.class.getName())
            .removeParameters(EXPORT_KEY, REFER_KEY);
          //生成元数据缓存 key 元数据维度 地址+名字
          //如:
zookeeper://127.0.0.1:2181/org.apache.dubbo.metadata.report.MetadataRe
port
        String key = url.toServiceString();
                //缓存中查询 查到则直接返回
        MetadataReport metadataReport = serviceStoreMap.get(key);
        if (metadataReport != null) {
            return metadataReport;
```

```
        }

        // Lock the metadata access process to ensure a single instance
of the metadata instance
        //存在写操作 加个锁
        lock.lock();
        try {
         //双重校验锁在查一下
            metadataReport = serviceStoreMap.get(key);
            if (metadataReport != null) {
                return metadataReport;
            }
            //check 参数 查元数据报错是否抛出异常
            boolean check = url.getParameter(CHECK_KEY, true) &&
url.getPort() != 0;
            try {
                  //关键模版方法 调用扩展实现的具体业务(创建元数据操作对象)
                metadataReport = createMetadataReport(url);
            } catch (Exception e) {
                if (!check) {
                    logger.warn("The metadata reporter failed to
initialize", e);
                } else {
                    throw e;
                }
            }
                  //check 逻辑检查
            if (check && metadataReport == null) {
                throw new IllegalStateException("Can not create metadata
Report " + url);
            }
            //缓存对象
            if (metadataReport != null) {
                serviceStoreMap.put(key, metadataReport);
            }
            //返回
            return metadataReport;
        } finally {
            // Release the lock
            lock.unlock();
        }
    }
```

上面这个抽象类 AbstractMetadataReportFactory 中的获取元数据操作对象的模版方法 getMetadataReport(URL url),用了双重校验锁的逻辑来创建对象缓存对象,又用了模版方法设计模式,来让抽象类做通用的逻辑,让实现类型去做扩展,虽然代码写的太长了些整体还是用了不少的设计思想。

我们直接看这个代码:

```
metadataReport = createMetadataReport(url);
```

这个创建元数据操作对象的代码实际上走的是实现类型的逻辑。

来自工厂 Bean ZookeeperMetadataReportFactory 的工厂方法如下所示:

```
@Override
    public MetadataReport createMetadataReport(URL url) {
        return new ZookeeperMetadataReport(url, zookeeperTransporter);
    }
```

创建了元数据操作对象,这里我们继续看下元数据操作对象 ZookeeperMetadataReport 创建做了哪些逻辑。

来自 ZookeeperMetadataReport 的构造器:

```
public ZookeeperMetadataReport(URL url, ZookeeperTransporter zookeeperTransporter) {
    //url即配置 配置传递给抽象类 做一些公共的逻辑
    //url参考:zookeeper://127.0.0.1:2181/org.apache.dubbo.metadata.report.MetadataReport?
application=dubbo-demo-api-provider&client=&port=2181&protocol=zookeeper
        super(url);
        if (url.isAnyHost()) {
            throw new IllegalStateException("registry address == null");
        }
        String group = url.getGroup(DEFAULT_ROOT);
        if (!group.startsWith(PATH_SEPARATOR)) {
            group = PATH_SEPARATOR + group;
        }
        this.root = group;
        //连接Zookeeper
        zkClient = zookeeperTransporter.connect(url);
    }
```

核心的公共的操作逻辑封装在父类 AbstractMetadataReport 里面，我们来看前面 super 调用的构造器逻辑。

如下所示：

```
public AbstractMetadataReport(URL reportServerURL) {
 //设置 url
如:zookeeper://127.0.0.1:2181/org.apache.dubbo.metadata.report.MetadataReport?application=dubbo-demo-api-provider&client=&port=2181&protocol=zookeeper
       setUrl(reportServerURL);
       // Start file save timer
       //缓存的文件名字
       //格式为: 用户目录+/.dubbo/dubbo-metadata- + 应用程序名字 application + url 地址(IP+端口) + 后缀.cache 如下所示
       ///Users/song/.dubbo/dubbo-metadata-dubbo-demo-api-provider-127.0.0.1-2181.cache
       String defaultFilename = System.getProperty(USER_HOME) + DUBBO_METADATA +
           reportServerURL.getApplication() + "-" +
           replace(reportServerURL.getAddress(), ":", "-") + CACHE;
           //如果用户配置了缓存文件名字则以用户配置为准 file
       String filename = reportServerURL.getParameter(FILE_KEY, defaultFilename);
       File file = null;
       //文件名字不为空
       if (ConfigUtils.isNotEmpty(filename)) {
           file = new File(filename);
           //文件和父目录不存在则创建文件目录
           if (!file.exists() && file.getParentFile() != null && !file.getParentFile().exists()) {
               if (!file.getParentFile().mkdirs()) {
                   throw new IllegalArgumentException("Invalid service store file " + file + ", cause: Failed to create directory " + file.getParentFile() + "!");
               }
           }
           // if this file exists, firstly delete it.
           //还未初始化则已存在的历史文件删除掉
           if (!initialized.getAndSet(true) && file.exists()) {
               file.delete();
```

```
            }
        }
        //赋值给成员变量后续继续可以用
        this.file = file;
        //文件存在则直接加载文件中的内容
        loadProperties();
        //sync-report 配置的值为同步配置还异步配置,true 是同步配置,默认为 false 为
异步配置
        syncReport = reportServerURL.getParameter(SYNC_REPORT_KEY,
false);
        //重试属性与逻辑也封装了一个类型 创建对象
        //retry-times 重试次数配置 默认为 100 次
        //retry-period 重试间隔配置 默认为 3000
        metadataReportRetry = new
MetadataReportRetry(reportServerURL.getParameter(RETRY_TIMES_KEY,
DEFAULT_METADATA_REPORT_RETRY_TIMES),
            reportServerURL.getParameter(RETRY_PERIOD_KEY,
DEFAULT_METADATA_REPORT_RETRY_PERIOD));

        // cycle report the data switch
        //是否定期从元数据中心同步配置
        //cycle-report 配置默认为 true
        if (reportServerURL.getParameter(CYCLE_REPORT_KEY,
DEFAULT_METADATA_REPORT_CYCLE_REPORT)) {
        //开启重试定时器 24 个小时间隔从元数据中心同步一次
            reportTimerScheduler =
Executors.newSingleThreadScheduledExecutor(new
NamedThreadFactory("DubboMetadataReportTimer", true));
            reportTimerScheduler.scheduleAtFixedRate(this::publishAll,
calculateStartTime(), ONE_DAY_IN_MILLISECONDS, TimeUnit.MILLISECONDS);
        }

        this.reportMetadata =
reportServerURL.getParameter(REPORT_METADATA_KEY, false);
        this.reportDefinition =
reportServerURL.getParameter(REPORT_DEFINITION_KEY, true);
    }
```

d) 内存中元数据自动同步到 Zookeeper 和本地文件

这里来总结下元数据操作的初始化逻辑：

- 首次初始化清理历史元数据文件如：Users/song/.dubbo/dubbo-metadata-dubbo-demo-api-provider-127.0.0.1-2181.cache

- 如果非首次进来则直接加载缓存在本地的缓存文件，赋值给 properties 成员变量

- 初始化同步配置是否异步（默认为 false），sync-report 配置的值为同步配置还异步配置，true 是同步配置，默认为 false 为异步配置

- 初始化重试属性

- 是否定期从元数据中心同步配置初始化，默认为 true 24 小时自动同步一次

关于元数据同步可以看 AbstractMetadataReport 类型的 publishAll 方法：

```
  reportTimerScheduler = Executors.newSingleThreadScheduledExecutor(new
NamedThreadFactory("DubboMetadataReportTimer", true));
            reportTimerScheduler.scheduleAtFixedRate(this::publishAll, calculateStartTime(),
ONE_DAY_IN_MILLISECONDS, TimeUnit.MILLISECONDS);
```

这里有个方法叫做 calculateStartTime 这个代码是随机时间的 between 2:00 am to 6:00 am，the time is random，2 点到 6 点之间启动，低峰期启动自动同步。

返回值：

AbstractMetadataReport 类型的

```
  void publishAll() {
        logger.info("start to publish all metadata.");
        this.doHandleMetadataCollection(allMetadataReports);
    }
```

AbstractMetadataReport 类型的 doHandleMetadataCollection

```
private boolean doHandleMetadataCollection(Map<MetadataIdentifier, Object> metadataMap) {
        if (metadataMap.isEmpty()) {
            return true;
        }
        Iterator<Map.Entry<MetadataIdentifier, Object>> iterable =
metadataMap.entrySet().iterator();
        while (iterable.hasNext()) {
            Map.Entry<MetadataIdentifier, Object> item = iterable.next();
            if (PROVIDER_SIDE.equals(item.getKey().getSide())) {
              //提供端的元数据则存储提供端元数据
                this.storeProviderMetadata(item.getKey(), (FullServiceDefinition)
item.getValue());
            } else if (CONSUMER_SIDE.equals(item.getKey().getSide())) {
            //消费端的元数据则存储提供端元数据
                this.storeConsumerMetadata(item.getKey(), (Map) item.getValue());
            }

        }
        return false;
    }
```

提供端元数据的存储：

AbstractMetadataReport 类型的 storeProviderMetadata

```
 @Override
    public void storeProviderMetadata(MetadataIdentifier providerMetadataIdentifier,
ServiceDefinition serviceDefinition) {
        if (syncReport) {
            storeProviderMetadataTask(providerMetadataIdentifier, serviceDefinition);
        } else {
            reportCacheExecutor.execute(() ->
storeProviderMetadataTask(providerMetadataIdentifier, serviceDefinition));
        }
    }
```

AbstractMetadataReport 类型的 storeProviderMetadataTask

具体同步代码：storeProviderMetadataTask

```
private void storeProviderMetadataTask(MetadataIdentifier providerMetadataIdentifier,
ServiceDefinition serviceDefinition) {
        try {
            if (logger.isInfoEnabled()) {
                logger.info("store provider metadata. Identifier : " +
providerMetadataIdentifier + "; definition: " + serviceDefinition);
            }
            allMetadataReports.put(providerMetadataIdentifier, serviceDefinition);
            failedReports.remove(providerMetadataIdentifier);
            Gson gson = new Gson();
            String data = gson.toJson(serviceDefinition);
             //内存中的元数据同步到元数据中心
            doStoreProviderMetadata(providerMetadataIdentifier, data);
            //内存中的元数据同步到本地文件
            saveProperties(providerMetadataIdentifier, data, true, !syncReport);
        } catch (Exception e) {
            // retry again. If failed again, throw exception.
            failedReports.put(providerMetadataIdentifier, serviceDefinition);
            metadataReportRetry.startRetryTask();
            logger.error("Failed to put provider metadata " + providerMetadataIdentifier + "
in  " + serviceDefinition + ", cause: " + e.getMessage(), e);
        }
    }
```

上面代码我们主要看本地内存中的元数据同步到元数据中心和存本地的两个点：

```
//内存中的元数据同步到元数据中心
doStoreProviderMetadata(providerMetadataIdentifier, data);
//内存中的元数据同步到本地文件
saveProperties(providerMetadataIdentifier, data, true,
```

//内存中的元数据同步到元数据中心

这个方法会调用当前子类重写的具体存储逻辑，这里我们以ZookeeperMetadataReport 的 doStoreProviderMetadata 举例：

```
   private void storeMetadata(MetadataIdentifier metadataIdentifier, String v) {
      //使用zkClient创建一个节点数据为参数v v是前面说的服务定义数据
        zkClient.create(getNodePath(metadataIdentifier), v, false);
    }
```

这里参数我们举个例子，提供者的元数据内容如下。

节点路径为：

- /dubbo/metadata/link.elastic.dubbo.entity.DemoService/provider/dubbo-demo-api-provider
- 格式：/dubbo/metadata 前缀
- 服务提供者接口
- 提供者类型 provider
- 应用名

具体的元数据内容如下。

比较详细的记录了应用信息，服务接口信息和服务接口对应的方法信息


```
{
  "parameters": {
    "side": "provider",
    "interface": "link.elastic.dubbo.entity.DemoService",
    "pid": "38680",
    "application": "dubbo-demo-api-provider",
    "dubbo": "2.0.2",
    "release": "3.0.8",
    "anyhost": "true",
    "bind.ip": "192.168.1.9",
    "methods": "sayHello,sayHelloAsync",
    "background": "false",
    "deprecated": "false",
    "dynamic": "true",
    "service-name-mapping": "true",
    "generic": "false",
    "bind.port": "20880",
    "timestamp": "1653097653865"
  },
  "canonicalName": "link.elastic.dubbo.entity.DemoService",
  "codeSource":
"file:/Users/song/Desktop/Computer/A/code/gitee/weaving-a-net/weaving-test/dubbo-test/target/classes/",
  "methods": [
    {
      "name": "sayHello",
```

```
      "parameterTypes": [
        "java.lang.String"
      ],
      "returnType": "java.lang.String",
      "annotations": [

      ]
    },
    {
      "name": "sayHelloAsync",
      "parameterTypes": [
        "java.lang.String"
      ],
      "returnType": "java.util.concurrent.CompletableFuture",
      "annotations": [

      ]
    }
  ],
  "types": [
    {
      "type": "java.util.concurrent.CompletableFuture",
      "properties": {
        "result": "java.lang.Object",
        "stack": "java.util.concurrent.CompletableFuture.Completion"
      }
    },
    {
      "type": "java.lang.Object"
    },
    {
      "type": "java.lang.String"
    },
    {
      "type": "java.util.concurrent.CompletableFuture.Completion",
      "properties": {
        "next": "java.util.concurrent.CompletableFuture.Completion",
        "status": "int"
      }
    },
    {
      "type": "int"
    }
```

```
  ],
  "annotations": [

  ]
}
```

本地缓存文件的写入，可以看下如下代码。

AbstractMetadataReport 类型的 saveProperties 方法

```
 private void saveProperties(MetadataIdentifier metadataIdentifier, String value, boolean
add, boolean sync) {
        if (file == null) {
            return;
        }

        try {
            if (add) {

properties.setProperty(metadataIdentifier.getUniqueKey(KeyTypeEnum.UNIQUE_KEY), value);
            } else {
                properties.remove(metadataIdentifier.getUniqueKey(KeyTypeEnum.UNIQUE_KEY));
            }
            long version = lastCacheChanged.incrementAndGet();
            if (sync) {
              //获取最新修改版本持久化到磁盘
                new SaveProperties(version).run();
            } else {
                reportCacheExecutor.execute(new SaveProperties(version));
            }

        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }
    }
```

主要看如下代码：

```
  new SaveProperties(version).run();
```

SaveProperties 类型代码如下：

```
    private class SaveProperties implements Runnable {
          private long version;

          private SaveProperties(long version) {
              this.version = version;
          }

          @Override
          public void run() {
              doSaveProperties(version);
          }
      }
```

继续看 doSaveProperties 方法：

```
private void doSaveProperties(long version) {
                //不是最新的就不要持久化了
        if (version < lastCacheChanged.get()) {
            return;
        }
        if (file == null) {
            return;
        }
        // Save
        try {
         //创建本地文件锁：
         //路径为：
         ///Users/song/.dubbo/dubbo-metadata-dubbo-demo-api-provider-
127.0.0.1-2181.cache.lock
            File lockfile = new File(file.getAbsolutePath() + ".lock");
            //锁文件不存在则创建锁文件
            if (!lockfile.exists()) {
                lockfile.createNewFile();
            }
            //随机访问文件工具类对象创建 读写权限
            try (RandomAccessFile raf = new RandomAccessFile(lockfile,
"rw");
            //文件文件 Channel
            //返回与此文件关联的唯一 FileChannel 对象。
                FileChannel channel = raf.getChannel()) {
                //FileChannel 中的 lock()与 tryLock()方法都是尝试去获取在某一文
件上的独有锁（以下简称独有锁），可以实现进程间操作的互斥。区别在于 lock()会阻塞
（blocking）方法的执行，tryLock()则不会。
                FileLock lock = channel.tryLock();
                //如果多个线程同时进来未获取锁的则抛出异常
```

```
            if (lock == null) {
                throw new IOException("Can not lock the metadataReport
cache file " + file.getAbsolutePath() + ", ignore and retry later,
maybe multi java process use the file, please config:
dubbo.metadata.file=xxx.properties");
            }
            // Save
            try {
              //文件不存在则创建本地元数据缓存文件
              ///Users/song/.dubbo/dubbo-metadata-dubbo-demo-api-
provider-127.0.0.1-2181.cache
                if (!file.exists()) {
                    file.createNewFile();
                }

                Properties tmpProperties;
                if (!syncReport) {
                    // When syncReport = false, properties.setProperty
and properties.store are called from the same
                    // thread(reportCacheExecutor), so deep copy is
not required
                    tmpProperties = properties;
                } else {
                    // Using store method and setProperty method of
the this.properties will cause lock contention
                    // under multi-threading, so deep copy a new
container
                    //异步存储会导致锁争用 使用此的 store 方法和 setProperty 方
法。属性将导致多线程下的锁争用，因此深度复制新容器
                    tmpProperties = new Properties();
                    Set<Map.Entry<Object, Object>> entries =
properties.entrySet();
                    for (Map.Entry<Object, Object> entry : entries) {
                        tmpProperties.setProperty((String)
entry.getKey(), (String) entry.getValue());
                    }
                }

                try (FileOutputStream outputFile = new
FileOutputStream(file)) {
                //Properties 类型自带的方法:
                //将此属性表中的属性列表（键和元素对）以适合使用 load（Reader）
方法的格式写入输出字符流。
```

```
                    tmpProperties.store(outputFile, "Dubbo
metadataReport Cache");
                }
            } finally {
                lock.release();
            }
        }
    } catch (Throwable e) {
        if (version < lastCacheChanged.get()) {
            return;
        } else {
            reportCacheExecutor.execute(new
SaveProperties(lastCacheChanged.incrementAndGet()));
        }
        //这个代码太诡异了如果是 lock 失败也会打印异常给人非常疑惑的感觉 后续会
修复
        logger.warn("Failed to save service store file, cause: " +
e.getMessage(), e);
    }
}
```

写入文件的内容大致如下

```
}
link.elastic.dubbo.entity.DemoService:::provider:dubbo-demo-api-
provider -> {
  "parameters": {
    "side": "provider",
    "interface": "link.elastic.dubbo.entity.DemoService",
    "pid": "41457",
    "application": "dubbo-demo-api-provider",
    "dubbo": "2.0.2",
    "release": "3.0.8",
    "anyhost": "true",
    "bind.ip": "192.168.1.9",
    "methods": "sayHello,sayHelloAsync",
    "background": "false",
    "deprecated": "false",
    "dynamic": "true",
    "service-name-mapping": "true",
    "generic": "false",
    "bind.port": "20880",
```

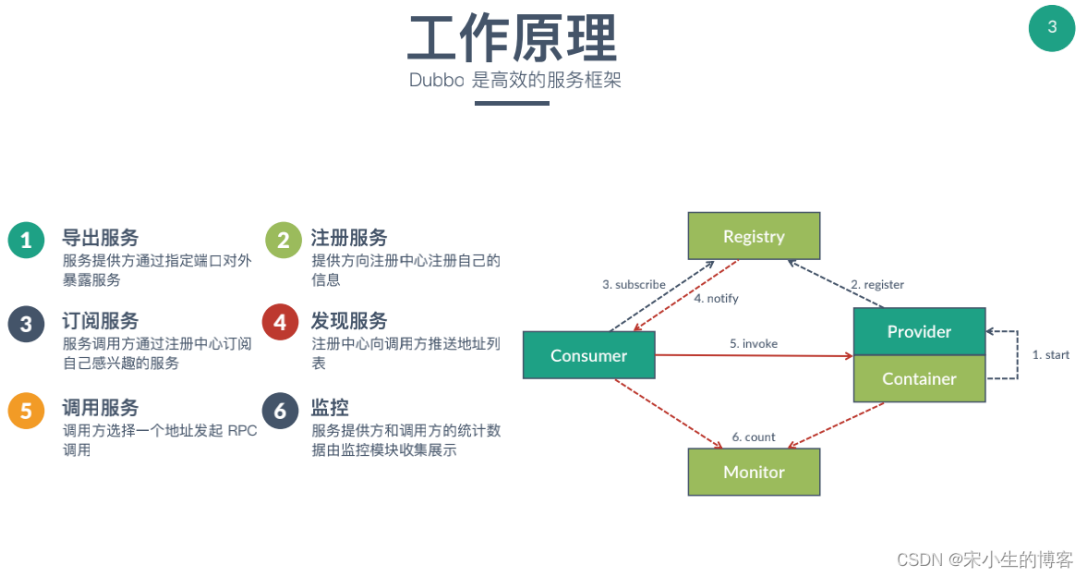
```
    "timestamp": "1653100253548"
  },
  "canonicalName": "link.elastic.dubbo.entity.DemoService",
  "codeSource":
"file:/Users/song/Desktop/Computer/A/code/gitee/weaving-a-net/weaving-
test/dubbo-test/target/classes/",
  "methods": [
    {
      "name": "sayHelloAsync",
      "parameterTypes": [
        "java.lang.String"
      ],
      "returnType": "java.util.concurrent.CompletableFuture",
      "annotations": [

      ]
    },
    {
      "name": "sayHello",
      "parameterTypes": [
        "java.lang.String"
      ],
      "returnType": "java.lang.String",
      "annotations": [

      ]
    }
  ],
  "types": [
    {
      "type": "java.util.concurrent.CompletableFuture",
      "properties": {
        "result": "java.lang.Object",
        "stack": "java.util.concurrent.CompletableFuture.Completion"
      }
    },
    {
      "type": "java.lang.Object"
    },
    {
      "type": "java.lang.String"
    },
    {
```

```
      "type": "java.util.concurrent.CompletableFuture.Completion",
      "properties": {
        "next": "java.util.concurrent.CompletableFuture.Completion",
        "status": "int"
      }
    },
    {
      "type": "int"
    }
  ],
  "annotations": [

  ]
}
```

## 四、 模块发布器发布服务解析

### 1. 简介

Dubbo 做为服务治理框架，比较核心的就是服务相关的概念，这里我先贴个找到的关于 Dubbo 工作原理的架构图：




CSDN @宋小生的博客

如果按完整服务启动与订阅的顺序我们可以归结为以下 6 点：

- **导出服务（提供者）**

服务提供方通过指定端口对外暴露服务

- **注册服务（提供者）**

提供方向注册中心注册自己的信息

- **（服务发现）-订阅服务（消费者）**

服务调用方通过注册中心订阅自己感兴趣的服务

- **（服务发现）-服务推送（消费者）**

注册中心向调用方推送地址列表

- **调用服务（消费者调用提供者）**

调用方选择一个地址发起 RPC 调用

- **监控服务**

服务提供方和调用方的统计数据由监控模块收集展示

上面的完整的服务启动订阅与调用流程不仅仅适用于 Dubbo 同样也适用于其他服务治理与发现的模型，一般服务发现与服务调用的思路就是这样的，我们将以上内容扩展，暴漏服务可以使用 http，tcp，udp 等各种协议，注册服务可以注册到 Redis，Dns，Etcd，Zookeeper 等注册中心中，订阅服务可以主动去注册中心查询服务列表，服务发现可以让注册中心将服务数据动态推送给消费者。

Dubbo 其实就是基于这种简单的服务模型来扩展出各种功能的支持，来满足服务治理的各种场景，了解了这里可能各位同学就想着自行开发一个简单的微服务框架了。

回到主题，从以上的服务完整发布调用流程可以看到，所有的功能都是由导出服务（提供者）开始的，只有提供者先提供了服务才可以有真正的服务让消费者调用。

之前的博客内容链接：《全局视野来看 Dubbo3.0.8 的服务启动生命周期》我们了解了 DefaultModuleDeployer 模块器启动的流程，其中在 start 代码的模版方法中开始了导出服务的功能，这里我们来详细看下服务发布的全过程：

入口代码：DefaultModuleDeployer 的发布服务方法

```
private void exportServices() {
   //从配置管缓存中查询缓存的所有的服务配置然后逐个服务发布
       for (ServiceConfigBase sc : configManager.getServices()) {
           exportServiceInternal(sc);
       }
   }
```

## 2. 导出服务的入口

入口代码：DefaultModuleDeployer 的发布服务方法

```
private void exportServices() {
   //从配置管缓存中查询缓存的所有的服务配置然后逐个服务发布
       for (ServiceConfigBase sc : configManager.getServices()) {
           exportServiceInternal(sc);
       }
   }
```

主要流程为遍历初始化的服务配置列表然后逐个服务开始到处内部导出服务代码。

exportServiceInternal 方法

```
private void exportServiceInternal(ServiceConfigBase sc) {

       ServiceConfig<?> serviceConfig = (ServiceConfig<?>) sc;
       //服务配置刷新 配置优先级覆盖
       if (!serviceConfig.isRefreshed()) {
           serviceConfig.refresh();
       }
       //服务已经导出过了就直接返回
       if (sc.isExported()) {
           return;
       }
       //是否异步方式导出 全局配置或者服务级其中一个配置了异步则异步处理
       if (exportAsync || sc.shouldExportAsync()) {
       //异步其实就是使用线程来导出服务 serviceExportExecutor
           ExecutorService executor =
executorRepository.getServiceExportExecutor();
```

```
            CompletableFuture<Void> future =
CompletableFuture.runAsync(() -> {
                try {
                    if (!sc.isExported()) {
                        sc.export();
                        exportedServices.add(sc);
                    }
                } catch (Throwable t) {
                    logger.error(getIdentifier() + " export async catch
error : " + t.getMessage(), t);
                }
            }, executor);

            asyncExportingFutures.add(future);
        } else {
         //同步导出服务
            if (!sc.isExported()) {
                sc.export();
                exportedServices.add(sc);
            }
        }
    }
```

这个逻辑里面做了一些基本的操作，可以直接看注释然后调用 ServiceConfig 的 export 的来导出服务，继续往后看服务配置的导出服务方法。

### 3. 服务配置导出服务模板方法

核心的服务导出代码是在服务配置中来做的 ServiceConfig 的 export()方法
ServiceConfig 的 export()方法代码如下：

```
@Override
    public void export() {
         //已经导出过服务直接放那会
        if (this.exported) {
            return;
        }

        // ensure start module, compatible with old api usage
        //确保模块启动了(基本的初始化操作执行了)
```

```
        getScopeModel().getDeployer().start();
                //悲观锁
        synchronized (this) {
         //双重校验
            if (this.exported) {
                return;
            }
                        //配置是否刷新 前面初始化时候已经刷新过配置
            if (!this.isRefreshed()) {
                this.refresh();
            }
            //服务导出配置配置为 false 则不导出
            if (this.shouldExport()) {
                  //服务发布前初始化一下元数据对象
                this.init();

                if (shouldDelay()) {
                  //配置了服务的延迟发布配置则走延迟发布逻辑
                    doDelayExport();
                } else {
                  //导出服务
                    doExport();
                }
            }
        }
    }
```

### 1) 服务配置导出服务前的初始化方法

ServiceConfig 导出服务之前的初始化方法 init

```
public void init() {
        if (this.initialized.compareAndSet(false, true)) {
          //加载服务监听器 这里暂时没有服务监听器扩展
            // load ServiceListeners from extension
            ExtensionLoader<ServiceListener> extensionLoader =
this.getExtensionLoader(ServiceListener.class);
            this.serviceListeners.addAll(extensionLoader.getSupportedExtensionInstances());
        }
        //服务提供者配置传递给元数据配置对象 一个服务提供者配置会有一个元数据配置，服务配置
        initServiceMetadata(provider);
        //元数据
        serviceMetadata.setServiceType(getInterfaceClass());

        serviceMetadata.setTarget(getRef());
        //元数据的key格式为 group/服务接口:版本号
        serviceMetadata.generateServiceKey();
    }
```

## 4. 服务配置导出服务模板方法 2

ServiceConfig 导出服务核心逻辑

```
protected synchronized void doExport() {
    //取消发布
        if (unexported) {
            throw new IllegalStateException("The service " + interfaceClass.getName() + "
has already unexported!");
        }
        //已经发布
        if (exported) {
            return;
        }
    //服务路径 为空则设置为接口名，本例子中为link.elastic.dubbo.entity.DemoService
        if (StringUtils.isEmpty(path)) {
            path = interfaceName;
        }
        //导出URL
        doExportUrls();
        //
        exported();
    }
```

### 1) 导出服务的 URL 配置逻辑

ServiceConfig 导出 URL 核心逻辑

```
private void doExportUrls() {
        //模块服务存储库
        ModuleServiceRepository repository =
getScopeModel().getServiceRepository();
        ServiceDescriptor serviceDescriptor;
        //ref 为服务实现类型 这里对应我们例子的 DemoServiceImpl
        final boolean serverService = ref instanceof ServerService;
        if(serverService){
            serviceDescriptor=((ServerService)
ref).getServiceDescriptor();
            repository.registerService(serviceDescriptor);
        }else{
         //我们代码走这个逻辑 注册服务 这个注册不是向注册中心注册 这个是解析服务接口
将服务方法等描述信息存放在了服务存储 ModuleServiceRepository 类型对象的成员变量
services 中
            serviceDescriptor =
repository.registerService(getInterfaceClass());
        }
        //提供者领域模型， 提供者领域模型 封装了一些提供者需要的就基本属性同时内部解
析封装方法信息 ProviderMethodModel 列表 ， 服务标识符 格式 group/服务接:版本号
        providerModel = new ProviderModel(getUniqueServiceName(),
         //服务实现类 DemoServiceImpl
            ref,
            //服务描述符 描述符里面包含了服务接口的方法信息，不过服务接口通过反射也
可以拿到方法信息
            serviceDescriptor,
            //服务配置
            this,
            //当前所处模型
            getScopeModel(),
            //当前服务接口的元数据对象
            serviceMetadata);

                //模块服务存储库存储提供者模型对象 ModuleServiceRepository
        repository.registerProvider(providerModel);
                //获取配置的注册中心列表 ， 同时将注册中心配置转 URL （在 Dubbo 中
URL 就是配置信息的一种形式）
                //这里会获取到两个  由 dubbo.application.register-mode 双注
册配置决定
                //注册中心
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?ap
plication=dubbo-demo-api-
```

```
provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timesta
mp=1653703292768
         //service-discovery-
registry://8.131.79.126:2181/org.apache.dubbo.registry.RegistryService
?application=dubbo-demo-api-
provider&dubbo=2.0.2&pid=10275&registry=zookeeper&release=3.0.8&timest
amp=1653704425920
         //参数 dubbo 是 dubbo 协议的版本不是 Dubbo 版本 Dubbo RPC protocol
version, for compatibility, it must not be between 2.0.10 ~ 2.6.2
          //这里后面详细说下 服务双注册  dubbo.application.register-mode
       List<URL> registryURLs =
ConfigValidationUtils.loadRegistries(this, true);

       for (ProtocolConfig protocolConfig : protocols) {
          String pathKey = URL.buildKey(getContextPath(protocolConfig)
                .map(p -> p + "/" + path)
                .orElse(path), group, version);
          // stub service will use generated service name
          if(!serverService) {
             // In case user specified path, register service one more
time to map it to path.
              //模块服务存储库 ModuleServiceRepository 存储服务接口信息
            repository.registerService(pathKey, interfaceClass);
          }
          //导出根据协议导出配置到注册中心
          doExportUrlsFor1Protocol(protocolConfig, registryURLs);
       }
    }
```

### 2) 应用级和接口级服务注册地址获取

这里主要看下注册中心的获取，这里涉及到服务的双注册配置

```
List<URL> registryURLs = ConfigValidationUtils.loadRegistries(this, true);
```

关于 loadRegistries 方法的详情我们就不看了主要看 loadRegistries 方法中调用的 genCompatibleRegistries 添加服务发现注册中心

```
/**
   * @param scopeModel 域模型
   * @param registryList 配置的注册中心列表 例如:
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?ap
plication=dubbo-demo-api-
provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timesta
mp=1653703292768
   *
   *  @param provider 是否为服务提供者 这里 Demo 为 true
   */
private static List<URL> genCompatibleRegistries(ScopeModel
scopeModel, List<URL> registryList, boolean provider) {
       List<URL> result = new ArrayList<>(registryList.size());
       //遍历所有的注册中心 为每个注册中心增加兼容的服务发现注册中心地址配置
       registryList.forEach(registryURL -> {
          //是否为提供者
           if (provider) {
               // for registries enabled service discovery,
automatically register interface compatible addresses.

               String registerMode;
               //注册协议配置了 service-discovery-registry 走这个逻辑
               //前面这个逻辑是直接接给 result 结果中添加应用级注册，如果是 all 配
置则增加接口级注册信息
               if
(SERVICE_REGISTRY_PROTOCOL.equals(registryURL.getProtocol())) {
                   registerMode =
registryURL.getParameter(REGISTER_MODE_KEY,
ConfigurationUtils.getCachedDynamicProperty(scopeModel,
DUBBO_REGISTER_MODE_DEFAULT_KEY, DEFAULT_REGISTER_MODE_INSTANCE));
                   if (!isValidRegisterMode(registerMode)) {
                       registerMode = DEFAULT_REGISTER_MODE_INSTANCE;
                   }
                   //这里配置的就是应用级配置 则先添加应用级地址，再根据条件判断是
否添加接口级注册中心地址
                   result.add(registryURL);
                   if
(DEFAULT_REGISTER_MODE_ALL.equalsIgnoreCase(registerMode)
                       && registryNotExists(registryURL, registryList,
REGISTRY_PROTOCOL)) {
                       URL interfaceCompatibleRegistryURL =
URLBuilder.from(registryURL)
                           .setProtocol(REGISTRY_PROTOCOL)
```

```
                        .removeParameter(REGISTRY_TYPE_KEY)
                        .build();
                    result.add(interfaceCompatibleRegistryURL);
                }
            } else {
                //正常情况下我们的配置会走这个逻辑
                // 获取服务注册的注册模式 配置为 dubbo.application.register-
mode 默认值为 all 既注册接口数据又注册应用级信息
                registerMode =
registryURL.getParameter(REGISTER_MODE_KEY,
ConfigurationUtils.getCachedDynamicProperty(scopeModel,
DUBBO_REGISTER_MODE_DEFAULT_KEY, DEFAULT_REGISTER_MODE_ALL));

                if (!isValidRegisterMode(registerMode)) {
                    registerMode = DEFAULT_REGISTER_MODE_INTERFACE;
                }
                 //根据逻辑条件判断是否添加应用级注册中心地址
                if
((DEFAULT_REGISTER_MODE_INSTANCE.equalsIgnoreCase(registerMode) ||
DEFAULT_REGISTER_MODE_ALL.equalsIgnoreCase(registerMode))
                    && registryNotExists(registryURL, registryList,
SERVICE_REGISTRY_PROTOCOL)) {
                    URL serviceDiscoveryRegistryURL =
URLBuilder.from(registryURL)
                        .setProtocol(SERVICE_REGISTRY_PROTOCOL)
                        .removeParameter(REGISTRY_TYPE_KEY)
                        .build();
                    result.add(serviceDiscoveryRegistryURL);
                }
                                        //根据逻辑条件判断是否添加接口级注册中
心地址
                if
(DEFAULT_REGISTER_MODE_INTERFACE.equalsIgnoreCase(registerMode) ||
DEFAULT_REGISTER_MODE_ALL.equalsIgnoreCase(registerMode)) {
                    result.add(registryURL);
                }
            }

            FrameworkStatusReportService reportService =
ScopeModelUtil.getApplicationModel(scopeModel).getBeanFactory().getBea
n(FrameworkStatusReportService.class);
```

```
reportService.reportRegistrationStatus(reportService.createRegistratio
nReport(registerMode));
            } else {
                result.add(registryURL);
            }
        });

        return result;
    }
```

这个方法是根据服务注册模式来判断使用接口级注册地址还是应用级注册地址分别如下所示。

配置信息：dubbo.application.register-mode

配置值：

- **interface**

接口级注册

- **instance**

应用级注册

- **all**

接口级别和应用级都注册

接口级注册地址：

```
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-
api-provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timestamp=1653703292768
```

应用级注册地址：

```
service-discovery-registry://8.131.79.126:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
provider&dubbo=2.0.2&pid=10275&registry=zookeeper&release=3.0.8&timestamp=1653704425920
```

## 5. 导出服务配置到本地和注册中心

```
doExportUrlsFor1Protocol(protocolConfig, registryURLs);
```

protocolConfig 为：dubbo 协议的配置<dubbo:protocol port="-1" name="dubbo" />

registryURLs 目前有两个，应用级注册地址和接口级注册地址。

```
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-
api-provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timestamp=1653703292768
```

```
service-discovery-registry://8.131.79.126:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
provider&dubbo=2.0.2&pid=10275&registry=zookeeper&release=3.0.8&timestamp=1653704425920
```

### 1) 导出服务配置的 doExportUrlsFor1Protocol 方法

```
    private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL>
registryURLs) {
        //生成协议配置具体可见下图中的元数据配置中的attachments
        Map<String, String> map = buildAttributes(protocolConfig);

        // remove null key and null value
        //移除空值 简化配置
        map.keySet().removeIf(key -> key == null || map.get(key) == null);
        // init serviceMetadata attachments
        //协议配置放到元数据对象中
        serviceMetadata.getAttachments().putAll(map);

    //协议配置 + 默认协议配置转URL类型的配置存储
        URL url = buildUrl(protocolConfig, map);
    //导出url
        exportUrl(url, registryURLs);
    }
```


## 2) 导出服务配置模板方法

继续看导出服务的模板方法，分为本地导出和注册中心导出//参数 url 为协议配置 url 可以参考：

```
dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?
anyhost=true&application=dubbo-demo-api-
provider&background=false&bind.ip=192.168.1.9&bind.port=20880&deprecated=false&dubbo=2.0.2&d
ynamic=true&generic=false&interface=link.elastic.dubbo.entity.DemoService&methods=sayHello,s
ayHelloAsync&pid=10953&release=3.0.8&side=provider&timestamp=1653705630518
```

```
private void exportUrl(URL url, List<URL> registryURLs) {
        String scope = url.getParameter(SCOPE_KEY);
        // don't export when none is configured
        if (!SCOPE_NONE.equalsIgnoreCase(scope)) {

            // export to local if the config is not remote (export to remote only when
config is remote)
            //未明确指定远程导出 则开启本地导出
            if (!SCOPE_REMOTE.equalsIgnoreCase(scope)) {
                exportLocal(url);
            }
      //未明确指定本地导出 则开启远程导出
            // export to remote if the config is not local (export to local only when config
is local)
            if (!SCOPE_LOCAL.equalsIgnoreCase(scope)) {
                url = exportRemote(url, registryURLs);
                if (!isGeneric(generic) && !getScopeModel().isInternal()) {
                    MetadataUtils.publishServiceDefinition(url,
providerModel.getServiceModel(), getApplicationModel());
                }
            }
        }
        this.urls.add(url);
    }
```

## 6. 导出服务到本地

本地调用使用了 injvm 协议，是一个伪协议，它不开启端口，不发起远程调用，只在 JVM 内直接关联，但执行 Dubbo 的 Filter 链。

直接通过代码来看吧。

```
 private void exportLocal(URL url) {
       //协议转为injvm 代表本地导出 host为127.0.0.1
        URL local = URLBuilder.from(url)
                .setProtocol(LOCAL_PROTOCOL)
                .setHost(LOCALHOST_VALUE)
                .setPort(0)
                .build();
        local = local.setScopeModel(getScopeModel())
            .setServiceModel(providerModel);
        doExportUrl(local, false);
        logger.info("Export dubbo service " + interfaceClass.getName() + " to local registry
url : " + local);
    }
```

### 1) doExportUrl 方法

```
private void doExportUrl(URL url, boolean withMetaData) {
    //这里是由adaptor扩展类型处理过的 我们直接看默认的类型javassist 对应JavassistProxyFactory代理工厂
获取调用对象 （
        Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass, url);
        if (withMetaData) {
            invoker = new DelegateProviderMetaDataInvoker(invoker, this);
        }
        Exporter<?> exporter = protocolSPI.export(invoker);
        exporters.add(exporter);
    }
```

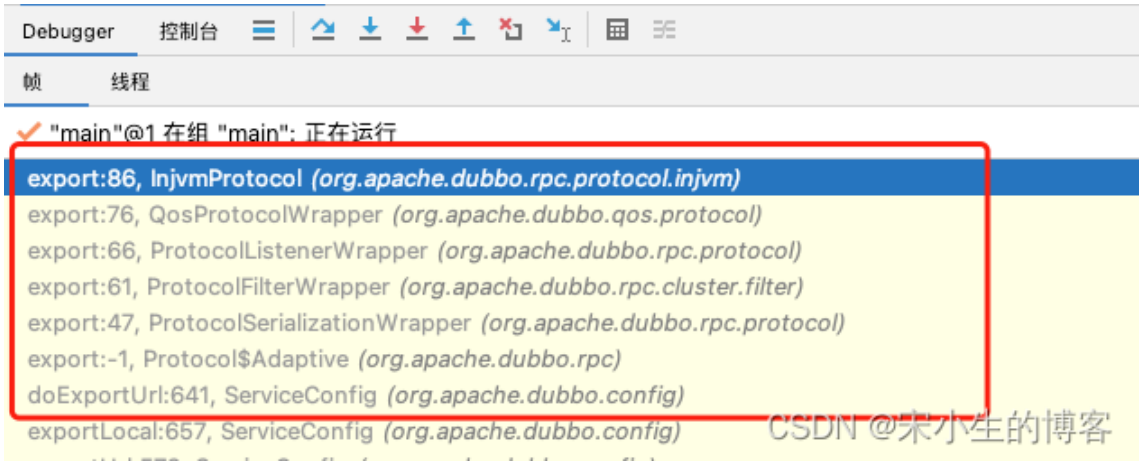
### 2) JavassistProxyFactory 类型的 getInvoker 方法

```
@Override
    public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
        try {
            // TODO Wrapper cannot handle this scenario correctly: the classname contains
'$'
            // 创建实际服务提供者的代理类型，代理类型后缀为DubboWrap在这里类型为
link.elastic.dubbo.entity.DemoServiceImplDubboWrap0
            final Wrapper wrapper =
Wrapper.getWrapper(proxy.getClass().getName().indexOf('$') < 0 ? proxy.getClass() : type);
            //创建一个匿名内部类对象 继承自AbstractProxyInvoker的Invoker对象
            return new AbstractProxyInvoker<T>(proxy, type, url) {
                @Override
                protected Object doInvoke(T proxy, String methodName,
                                          Class<?>[] parameterTypes,
                                          Object[] arguments) throws Throwable {
                    return wrapper.invokeMethod(proxy, methodName, parameterTypes,
arguments);
                }
            };
        } catch (Throwable fromJavassist) {
            // try fall back to JDK proxy factory
            ...
            }
        }
    }
```

### 3) 使用协议导出调用对象 export

```
Exporter<?> exporter = protocolSPI.export(invoker);
```

这个使用了 Adaptor 扩展和 Wrapper 机制 Debug 起来不太方便这里贴一下调用堆栈。



### a) 协议序列化机制 ProtocolSerializationWrapper

```
    @Override
     public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
         //这里主要逻辑是将服务提供者url添加到服务存储仓库中

 getFrameworkModel(invoker.getUrl().getScopeModel()).getServiceRepository().registerProvider
Url(invoker.getUrl());
         return protocol.export(invoker);
     }
```

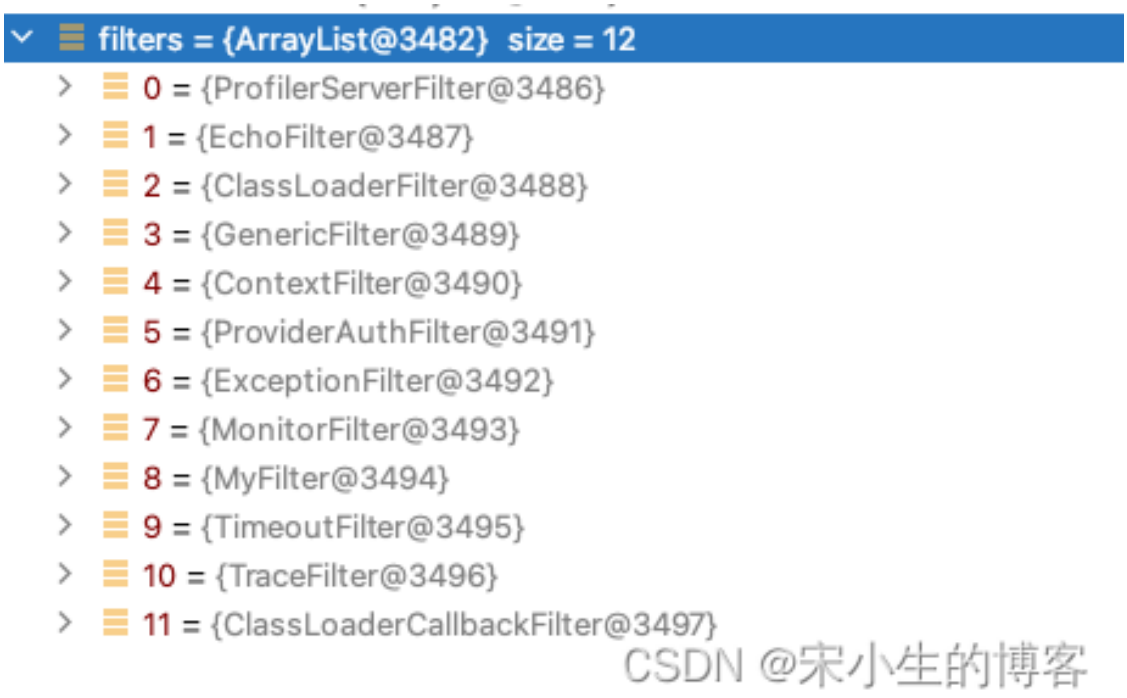
### b) 协议过滤器 Wrapper ProtocolFilterWrapper

```
  @Override
     public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
         //注册中心的协议导出直接执行
         if (UrlUtils.isRegistry(invoker.getUrl())) {
             return protocol.export(invoker);
         }
         //过滤器调用链FilterChainBuilder的扩展对象查询
         FilterChainBuilder builder = getFilterChainBuilder(invoker.getUrl());
         //这里分为2步 生成过滤器调用链 然后使用链表中的节点调用  这里值查询provider类型的过滤器
         return protocol.export(builder.buildInvokerChain(invoker, SERVICE_FILTER_KEY,
CommonConstants.PROVIDER));
     }
```

过滤器调用链的生成,对用 DefaultFilterChainBuilder 类型的 buildInvokerChain 方法

```
@Override
    public <T> Invoker<T> buildInvokerChain(final Invoker<T>
originalInvoker, String key, String group) {
         //originalInvoker 代表真正的服务调用器
        Invoker<T> last = originalInvoker;
        URL url = originalInvoker.getUrl();
        List<ModuleModel> moduleModels = getModuleModelsFromUrl(url);
        List<Filter> filters;
        if (moduleModels != null && moduleModels.size() == 1) {
        //类型 Filter key 为 service.filter 分组为 provider 所有提供者过滤器拉取
            filters = ScopeModelUtil.getExtensionLoader(Filter.class,
moduleModels.get(0)).getActivateExtension(url, key, group);
        } else if (moduleModels != null && moduleModels.size() > 1) {
            filters = new ArrayList<>();
```

```
            List<ExtensionDirector> directors = new ArrayList<>();
            for (ModuleModel moduleModel : moduleModels) {
                List<Filter> tempFilters =
ScopeModelUtil.getExtensionLoader(Filter.class,
moduleModel).getActivateExtension(url, key, group);
                filters.addAll(tempFilters);
                directors.add(moduleModel.getExtensionDirector());
            }
            filters = sortingAndDeduplication(filters, directors);

        } else {
            filters = ScopeModelUtil.getExtensionLoader(Filter.class,
null).getActivateExtension(url, key, group);
        }

                    //倒序拼接，将过滤器的调用对象添加到链表中  最后倒序遍历之后 last
节点指向了调用链路链表头节点的对象
        if (!CollectionUtils.isEmpty(filters)) {
            for (int i = filters.size() - 1; i >= 0; i--) {
                final Filter filter = filters.get(i);
                final Invoker<T> next = last;
                //每个 invoker 对象中都有 originalInvoker 对象
                last = new CopyOfFilterChainNode<>(originalInvoker, next,
filter);
            }
            return new CallbackRegistrationInvoker<>(last, filters);
        }

        return last;
    }
```




### c) 协议监听器 Wrapper ProtocolListenerWrapper

```
@Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        //注册中心地址则直接导出
        if (UrlUtils.isRegistry(invoker.getUrl())) {
            return protocol.export(invoker);
        }
        // 先导出对象 再创建过滤器包装对象 执行监听器逻辑
        return new ListenerExporterWrapper<T>(protocol.export(invoker),

Collections.unmodifiableList(ScopeModelUtil.getExtensionLoader(ExporterListener.class,
invoker.getUrl().getScopeModel())
                        .getActivateExtension(invoker.getUrl(), EXPORTER_LISTENER_KEY)));
    }
```

### d) QOS 的协议 Wrapper QosProtocolWrapper

```
@Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
         //注册中心导出的时候开启QOS 默认端口22222
        if (UrlUtils.isRegistry(invoker.getUrl())) {
            startQosServer(invoker.getUrl());
            return protocol.export(invoker);
        }
        return protocol.export(invoker);
    }
```

### e) InjvmProtocol 的导出方法

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

## 7. 导出服务到注册中心

16.5.2 导出服务配置模板方法中我们看到了服务导出会导出到本地和远程，接下来就看下导出到远程的方法 exportRemote

参数 url：

```
dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?
anyhost=true&application=dubbo-demo-api-
provider&background=false&bind.ip=192.168.1.9&bind.port=20880&deprecated=false&dubbo=2.0.2&d
ynamic=true&generic=false&interface=link.elastic.dubbo.entity.DemoService&methods=sayHello,s
ayHelloAsync&pid=12865&release=3.0.8&side=provider&timestamp=1653708351378
```

参数 registryURLs 目前有两个，应用级注册地址和接口级注册地址：

```
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-
api-provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timestamp=1653703292768
```

```
service-discovery-registry://8.131.79.126:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
provider&dubbo=2.0.2&pid=10275&registry=zookeeper&release=3.0.8&timestamp=1653704425920
```

```
private URL exportRemote(URL url, List<URL> registryURLs) {

       if (CollectionUtils.isNotEmpty(registryURLs)) {
        //遍历所有注册地址与注册模式 逐个注册
           for (URL registryURL : registryURLs) {
             //为协议 URL 添加应用级注册 service-discovery-registry 参数
service-name-mapping 为 true
               if
(SERVICE_REGISTRY_PROTOCOL.equals(registryURL.getProtocol())) {
```

```
                url =
url.addParameterIfAbsent(SERVICE_NAME_MAPPING_KEY, "true");
            }

            //if protocol is only injvm ,not register
            if (LOCAL_PROTOCOL.equalsIgnoreCase(url.getProtocol())) {
                continue;
            }
                                //为协议 url 添加动态配置 dynamic
            url = url.addParameterIfAbsent(DYNAMIC_KEY,
registryURL.getParameter(DYNAMIC_KEY));
            //监控配置暂时为 null
            URL monitorUrl = ConfigValidationUtils.loadMonitor(this,
registryURL);
            if (monitorUrl != null) {
                url = url.putAttribute(MONITOR_KEY, monitorUrl);
            }

            // For providers, this is used to enable custom proxy to
generate invoker
            String proxy = url.getParameter(PROXY_KEY);
            if (StringUtils.isNotEmpty(proxy)) {
                registryURL = registryURL.addParameter(PROXY_KEY,
proxy);
            }

                                //开始注册服务了 打印个认知 提示下
            if (logger.isInfoEnabled()) {
                if (url.getParameter(REGISTER_KEY, true)) {
                    logger.info("Register dubbo service " +
interfaceClass.getName() + " url " + url + " to registry " +
registryURL.getAddress());
                } else {
                    logger.info("Export dubbo service " +
interfaceClass.getName() + " to url " + url);
                }
            }

            doExportUrl(registryURL.putAttribute(EXPORT_KEY, url),
true);
        }

    } else {
```

```
            if (logger.isInfoEnabled()) {
                logger.info("Export dubbo service " +
interfaceClass.getName() + " to url " + url);
            }

            doExportUrl(url, true);
        }


        return url;
    }
```

### 1) doExportUrl 方法

与 16.6.1 doExportUrl 方法导出本地协议是一样的逻辑，我们来看看点不同地方

```
private void doExportUrl(URL url, boolean withMetaData) {
        Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass, url);
        if (withMetaData) {
          //远程服务导出逐个值为true 元数据invoker包装一下
            invoker = new DelegateProviderMetaDataInvoker(invoker, this);
        }
        Exporter<?> exporter = protocolSPI.export(invoker);
        exporters.add(exporter);
    }
```

与本地导出 ProtocolFilterWrapper 的不同之处

服务发现 service-discovery-registry 的导出 UrlUtils.isRegistry(invoker.getUrl())判断结果为 true 会走这个逻辑

```
 @Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        //注册中心的协议导出直接执行
        // 服务发现service-discovery-registry的导出会走这个逻辑
        if (UrlUtils.isRegistry(invoker.getUrl())) {
            return protocol.export(invoker);
        }
        //过滤器调用链FilterChainBuilder的扩展对象查询
        FilterChainBuilder builder = getFilterChainBuilder(invoker.getUrl());
        //这里分为2步 生成过滤器调用链 然后使用链表中的节点调用  这里值查询provider类型的过滤器
        return protocol.export(builder.buildInvokerChain(invoker, SERVICE_FILTER_KEY,
CommonConstants.PROVIDER));
    }
```

与协议监听器 Wrapper ProtocolListenerWrapper 的不同之处

服务发现 service-discovery-registry 的导出 UrlUtils.isRegistry(invoker.getUrl()判断结果为 true 会走这个逻辑

```
@Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        //注册中心地址则直接导出
         // 服务发现service-discovery-registry的导出会走这个逻辑
        if (UrlUtils.isRegistry(invoker.getUrl())) {
            return protocol.export(invoker);
        }
        // 先导出对象 再创建过滤器包装对象 执行监听器逻辑
        return new ListenerExporterWrapper<T>(protocol.export(invoker),

Collections.unmodifiableList(ScopeModelUtil.getExtensionLoader(ExporterListener.class,
invoker.getUrl().getScopeModel())
                        .getActivateExtension(invoker.getUrl(), EXPORTER_LISTENER_KEY)));
    }
```

与 16.6.3.4 QOS 的协议 Wrapper QosProtocolWrapper 不同之处

服务发现 service-discovery-registry 的导出 UrlUtils.isRegistry(invoker.getUrl()判断结果为 true 会走这个逻辑

```
@Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
         //注册中心导出的时候开启QOS 默认端口22222
        if (UrlUtils.isRegistry(invoker.getUrl())) {
            startQosServer(invoker.getUrl());
            return protocol.export(invoker);
        }
        return protocol.export(invoker);
    }
```

QOS 的 Server 的启动方法 start

```
private void startQosServer(URL url) {
        try {
            if (!hasStarted.compareAndSet(false, true)) {
                return;
            }

            boolean qosEnable = url.getParameter(QOS_ENABLE, true);
            if (!qosEnable) {
                logger.info("qos won't be started because it is disabled. " +
                        "Please check dubbo.application.qos.enable is configured either in
system property, " +
                        "dubbo.properties or XML/spring-boot configuration.");
                return;
            }

            String host = url.getParameter(QOS_HOST);
            int port = url.getParameter(QOS_PORT, QosConstants.DEFAULT_PORT);
            boolean acceptForeignIp =
Boolean.parseBoolean(url.getParameter(ACCEPT_FOREIGN_IP, "false"));
            Server server = frameworkModel.getBeanFactory().getBean(Server.class);
            server.setHost(host);
            server.setPort(port);
            server.setAcceptForeignIp(acceptForeignIp);
            server.start();

        } catch (Throwable throwable) {
            logger.warn("Fail to start qos server: ", throwable);
        }
```

QOS 的 Server 的启动方法 start

```
public void start() throws Throwable {
        if (!started.compareAndSet(false, true)) {
            return;
        }
        //1 个主线程
        boss = new NioEventLoopGroup(1, new DefaultThreadFactory("qos-
boss", true));
        //0 个从线程
        worker = new NioEventLoopGroup(0, new DefaultThreadFactory("qos-
worker", true));
        //服务端启动器，和参数设置
        ServerBootstrap serverBootstrap = new ServerBootstrap();
        serverBootstrap.group(boss, worker);
        serverBootstrap.channel(NioServerSocketChannel.class);
        serverBootstrap.option(ChannelOption.SO_REUSEADDR, true);
        serverBootstrap.childOption(ChannelOption.TCP_NODELAY, true);
        serverBootstrap.childHandler(new ChannelInitializer<Channel>() {
```

```
            @Override
            protected void initChannel(Channel ch) throws Exception {
                ch.pipeline().addLast(new
QosProcessHandler(frameworkModel, welcome, acceptForeignIp));
            }
        });
        try {
            if (StringUtils.isBlank(host)) {
                serverBootstrap.bind(port).sync();
            } else {
                serverBootstrap.bind(host, port).sync();
            }

            logger.info("qos-server bind localhost:" + port);
        } catch (Throwable throwable) {
            logger.error("qos-server can not bind localhost:" + port,
throwable);
            throw throwable;
        }
    }
```

QOS 处理器为 QosProcessHandler 关于 QosProcessHandler 的细节这里先不说

最后一个不同的地方调用链路走的这个 RegistryProtocol

### 2) 通过注册协议导出服务与注册服务的流程

RegistryProtocol 的导出方法，这个方法非常重要也是服务注册的核心代码，先概括下包含了哪些步骤：

- 覆盖配置
- 导出协议端口开启 TCP 服务
- 注册到注册中心
- 通知服务启动了

```
@Override
    public <T> Exporter<T> export(final Invoker<T> originInvoker)
throws RpcException {
```

```
      //service-discovery-
registry://8.131.79.126:2181/org.apache.dubbo.registry.RegistryService
?application=dubbo-demo-api-
provider&dubbo=2.0.2&pid=14256&registry=zookeeper&release=3.0.8&timest
amp=1653710477057
       URL registryUrl = getRegistryUrl(originInvoker);
       // url to export locally

//dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?anyh
ost=true&application=dubbo-demo-api-
provider&background=false&bind.ip=192.168.1.9&bind.port=20880&deprecat
ed=false&dubbo=2.0.2&dynamic=true&generic=false&interface=link.elastic
.dubbo.entity.DemoService&methods=sayHello,sayHelloAsync&pid=14256&rel
ease=3.0.8&service-name-
mapping=true&side=provider&timestamp=1653710479073
       URL providerUrl = getProviderUrl(originInvoker);

       // Subscribe the override data
       // FIXME When the provider subscribes, it will affect the
scene : a certain JVM exposes the service and call
       //  the same service. Because the subscribed is cached key with
the name of the service, it causes the
       //  subscription information to cover.

//provider://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?a
nyhost=true&application=dubbo-demo-api-
provider&background=false&bind.ip=192.168.1.9&bind.port=20880&category
=configurators&check=false&deprecated=false&dubbo=2.0.2&dynamic=true&g
eneric=false&interface=link.elastic.dubbo.entity.DemoService&methods=s
ayHello,sayHelloAsync&pid=14256&release=3.0.8&service-name-
mapping=true&side=provider&timestamp=1653710479073

       final URL overrideSubscribeUrl =
getSubscribedOverrideUrl(providerUrl);
       //override 配置
       final OverrideListener overrideSubscribeListener = new
OverrideListener(overrideSubscribeUrl, originInvoker);
       Map<URL, NotifyListener> overrideListeners =
getProviderConfigurationListener(providerUrl).getOverrideListeners();
       overrideListeners.put(registryUrl, overrideSubscribeListener);

       providerUrl = overrideUrlWithConfig(providerUrl,
overrideSubscribeListener);
```

```
        //export invoker
        final ExporterChangeableWrapper<T> exporter =
doLocalExport(originInvoker, providerUrl);

        // url to registry
        //通过 URL 获取 注册中心 Registry 操作对象
        final Registry registry = getRegistry(registryUrl);
        //需要向注册中心注册地址转换

//dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?anyh
ost=true&application=dubbo-demo-api-
provider&background=false&deprecated=false&dubbo=2.0.2&dynamic=true&ge
neric=false&interface=link.elastic.dubbo.entity.DemoService&methods=sa
yHello,sayHelloAsync&pid=14656&release=3.0.8&service-name-
mapping=true&side=provider&timestamp=1653711086189
        final URL registeredProviderUrl = getUrlToRegistry(providerUrl,
registryUrl);

        // decide if we need to delay publish (provider itself and
registry should both need to register)
        boolean register = providerUrl.getParameter(REGISTER_KEY, true)
&& registryUrl.getParameter(REGISTER_KEY, true);
        //是否向注册中心注册
        if (register) {
            register(registry, registeredProviderUrl);
        }

        // register stated url on provider model
        registerStatedUrl(registryUrl, registeredProviderUrl, register);


        exporter.setRegisterUrl(registeredProviderUrl);
        exporter.setSubscribeUrl(overrideSubscribeUrl);

        if (!registry.isServiceDiscovery()) {
            // Deprecated! Subscribe to override rules in 2.6.x or
before.
            registry.subscribe(overrideSubscribeUrl,
overrideSubscribeListener);
        }
                //内置监听器通知 这个不是通知消费者的
        notifyExport(exporter);
```

```
        //Ensure that a new exporter instance is returned every time
export
        return new DestroyableExporter<>(exporter);
    }
```

## 8. doLocalExport 本地导出协议开启端口

前面已经看过了本地协议 JVM 协议的服务导出和注册中心配置的导出，这里可以直接看一些关键代码：

```
  private <T> ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker, URL
providerUrl) {
        String key = getCacheKey(originInvoker);

        return (ExporterChangeableWrapper<T>) bounds.computeIfAbsent(key, s -> {
            Invoker<?> invokerDelegate = new InvokerDelegate<>(originInvoker, providerUrl);
            //代码中用的这个protoco对象是dubbo自动生成的适配器对象protocol$Adaptive 适配器对象会根据当
前协议的参数来查询具体的协议扩展对象
            return new ExporterChangeableWrapper<>((Exporter<T>)
protocol.export(invokerDelegate), originInvoker);
        });
    }
```

上面这个 protocol$Adaptive 协议的 export 导出方法与之前的一样也会经历下面几个过程，具体细节可以参考 JVM 协议的导出：

- ProtocolSerializationWrapper
- ProtocolFilterWrapper
- ProtocolListenerWrapper
- QosProtocolWrapper
- 唯一不同的是我们这里对应的协议扩展类型为 DubboProtocol、接下来来看下 DubboProtocol 的导出服务 export 方法实现：

```
@Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws
RpcException {
        checkDestroyed();
        //服务提供者的 url 参考例子
dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?anyhos
t=true&application=dubbo-demo-api-
```

```
provider&background=false&bind.ip=192.168.1.9&bind.port=20880&deprecat
ed=false&dubbo=2.0.2&dynamic=true&generic=false&interface=link.elastic
.dubbo.entity.DemoService&methods=sayHello,sayHelloAsync&pid=6043&rele
ase=3.0.8&service-name-
mapping=true&side=provider&timestamp=1654224285437
        URL url = invoker.getUrl();

        // export service.
        //生成服务的 key 参考：link.elastic.dubbo.entity.DemoService:20880
        String key = serviceKey(url);
        //创建导出服务用的导出器 DubboExporter
        DubboExporter<T> exporter = new DubboExporter<T>(invoker, key,
exporterMap);

        //export a stub service for dispatching event
        //stub 配置校验
        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."));
                }

            }
        }
                  //创建服务开启服务端口
        openServer(url);
        //
        optimizeSerialization(url);

        return exporter;
    }
```

### 开启服务端口

这里就到了 RPC 协议的 TCP 通信模块了，对应 DubboProtocol 的 openServer(url); 方法

```
private void openServer(URL url) {
        checkDestroyed();
        // find server. 地址作为key这里是192.168.1.9:20880
        String key = url.getAddress();
        // client can export a service which only for server to invoke
        //默认提供者开启服务，消费者是不能开启服务的
        boolean isServer = url.getParameter(IS_SERVER_KEY, true);
        if (isServer) {
            //协议服务器 下面一个双重校验锁检查，如果为空则创建服务
            ProtocolServer server = serverMap.get(key);
            if (server == null) {
                synchronized (this) {
                    server = serverMap.get(key);
                    if (server == null) {
                        serverMap.put(key, createServer(url));
                    }else {
                        server.reset(url);
                    }
                }
            } else {
                // server supports reset, use together with override
                server.reset(url);
            }
        }
    }
```

为当前地址创建协议服务对应方法如下：DubboProtocol 的 createServer 方法

```
private ProtocolServer createServer(URL url) {
                //下面将 url 增加了心跳参数最终如下
dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?anyhost=true&application=dubbo-demo-api-provider&background=false&bind.ip=192.168.1.9&bind.port=20880&channel.readonly.sent=true&codec=dubbo&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&heartbeat=60000&interface=link.elastic.dubbo.entity.DemoService&methods=sayHello,sayHelloAsync&pid=6700&release=3.0.8&service-name-mapping=true&side=provider&timestamp=1654225251112
        url = URLBuilder.from(url)
                // send readonly event when server closes, it's enabled by default
                .addParameterIfAbsent(CHANNEL_READONLYEVENT_SENT_KEY, Boolean.TRUE.toString())
```

```
                // enable heartbeat by default
                .addParameterIfAbsent(HEARTBEAT_KEY,
String.valueOf(DEFAULT_HEARTBEAT))
                .addParameter(CODEC_KEY, DubboCodec.NAME)
                .build();
        //这里服务端使用的网络库这里是默认值 netty
        String str = url.getParameter(SERVER_KEY,
DEFAULT_REMOTING_SERVER);

        if (StringUtils.isNotEmpty(str)
&& !url.getOrDefaultFrameworkModel().getExtensionLoader(Transporter.cl
ass).hasExtension(str)) {
            throw new RpcException("Unsupported server type: " + str +
", url: " + url);
        }
                //dubbo 交换器层对象创建
        ExchangeServer server;
        try {
         //这个方法会绑定端口，关于交换器与传输网络层到后面统一说
         //这里通过绑定 url 和请求处理器来创建交换器对象
            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 (StringUtils.isNotEmpty(str)) {
            Set<String> supportedTypes =
url.getOrDefaultFrameworkModel().getExtensionLoader(Transporter.class)
.getSupportedExtensions();
            if (!supportedTypes.contains(str)) {
                throw new RpcException("Unsupported client type: " +
str);
            }
        }

        DubboProtocolServer protocolServer = new
DubboProtocolServer(server);
        //关闭等待时长默认为 10 秒
        loadServerProperties(protocolServer);
        return protocolServer;
    }
```

### 9. 向注册中心注册服务 register

这个细节在下个博客中说涉及到 Dubbo3 的双注册。

## 五、 注册中心双注册原理

### 1. 简介

上个博客《15-Dubbo 的三大中心之元数据中心源码解析》导出服务端的时候多次提到了元数据中心，注册信息的注册。

Dubbo3 出来时间不太长，对于现在的用户来说大部分使用的仍旧是 Dubbo2.x，

Dubbo3 比较有特色也是会直接使用到的功能就是应用级服务发现：

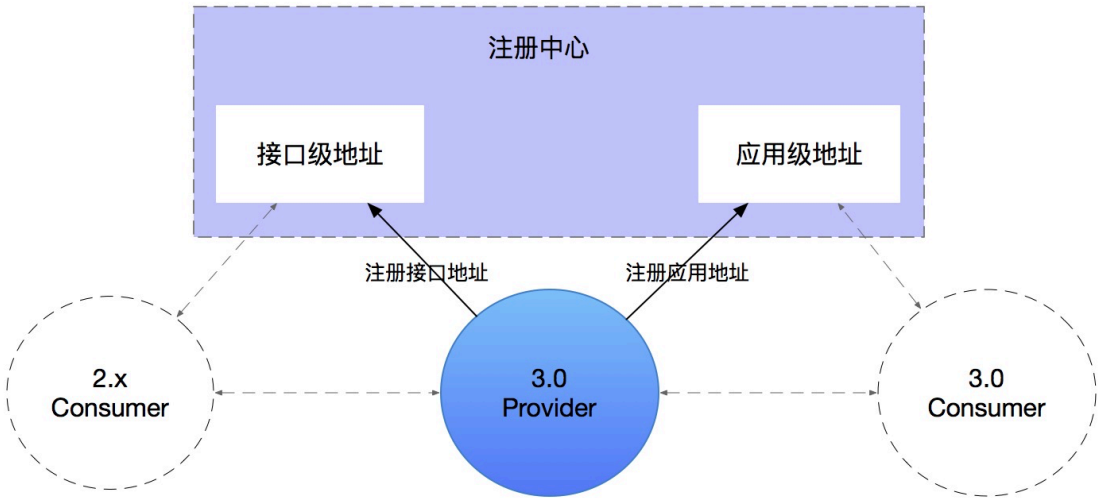
- **应用级服务发现**

从服务/接口粒度到应用粒度的升级，使得 Dubbo 在集群可伸缩性、连接异构微服务体系上更具优势。应用粒度能以更低的资源消耗支持超百万实例规模集群程；实现与 Spring Cloud、Kubernetes Service 等异构微服务体系的互联互通。

对于直接使用 Dubbo3 的用户还好，可以仅仅开启应用级注册，但是对于 Dubbo2.x 的用户升级到 Dubbo3 的用户来说前期都是要开启双注册来慢慢迁移的，既注册传统的接口信息到注册中心，又注册应用信息到注册中心，同时注册应用与接口关系的元数据信息。

关于双注册与服务迁移的过程的使用可以参考官网：应用级地址发现迁移指南

关于官网提供者双注册的图我这里贴一下，方便了解：




## 2. 双注册配置的读取

### 1) 注册中心地址作为元数据中心

这个配置的解析过程在前面的内容介绍元数据中心的时候很详细的说了。

对应代码位于：DefaultApplicationDeployer 类型的 startMetadataCenter()方法

```
private void startMetadataCenter() {
    //如果未配置元数据中心的地址等配置则使用注册中心的地址等配置做为元数据中心的配置
        useRegistryAsMetadataCenterIfNecessary();
        //...省略掉其他代码防止受到干扰
    }
```

具体逻辑是这个方法：useRegistryAsMetadataCenterIfNecessary

```
private void useRegistryAsMetadataCenterIfNecessary() {
    //配置缓存中查询元数据配置
        Collection<MetadataReportConfig> metadataConfigs =
configManager.getMetadataConfigs();

        //...省略掉空判断
    //查询是否有注册中心设置了默认配置isDefault 设置为true的注册中心则为默认注册中心列表,如果没有注册中心
设置为默认注册中心,则获取所有未设置默认配置的注册中心列表
        List<RegistryConfig> defaultRegistries = configManager.getDefaultRegistries();
        if (defaultRegistries.size() > 0) {
        //多注册中心遍历
            defaultRegistries
                .stream()
                //筛选符合条件的注册中心 (筛选逻辑就是查看是否有对应协议的扩展支持)
                .filter(this::isUsedRegistryAsMetadataCenter)
                //注册中心配置映射为元数据中心  映射就是获取需要的配置
                .map(this::registryAsMetadataCenter)
                //将元数据中心配置存储在配置缓存中方便后续使用
                .forEach(metadataReportConfig -> {
                  //...省略掉具体的逻辑
                });
        }
    }
```

关于元数据中心地址的获取，主要经过如下逻辑：

- **查询：**所有可用的默认注册中心列表
- **遍历：**多注册中心遍历
- **筛选：**选符合条件的注册中心(筛选逻辑就是查看是否有对应协议的扩展支持)
- **转化：**注册中心配置 RegistryConfig 映射转换为元数据中心配置类型 MetadataReportConfig

MetadataReportConfig 映射就是获取需要的配置。

最后会把查询到的元数据中心配置存储在配置缓存中方便后续使用。

### 2) 双注册模式配置

双注册配置类型是这个

```
dubbo.application.register-mode=all
```

默认值为 all 代表应用级注册和接口级注册，当前在完全迁移到应用级注册之后可以将服务直接迁移到应用级配置上去。

配置值解释：

- all 双注册
- instance 应用级注册
- interface 接口级注册

后面的代码如果想要看更详细的代码可以看文章《16-模块发布器发布服务全过程》。

关于这个配置的使用我们详细来看下，在 Dubbo 服务注册时候会先通过此配置查询需要注册服务地址，具体代码位于 ServiceConfig 的 doExportUrls()方法中：

```
private void doExportUrls() {
      //省略掉前面的代码...
        List<URL> registryURLs = ConfigValidationUtils.loadRegistries(this, true);
      //省略掉后面的代码...
    }
```

然后就是具体注册中心地址的获取过程我们看下：ConfigValidationUtils 的加载注册中心地址方法 loadRegistries

```
 public static List<URL> loadRegistries(AbstractInterfaceConfig interfaceConfig, boolean
provider) {
        // check && override if necessary
           //省略掉前面的代码...
           //这里会获取到一个接口配置注册地址例如：
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-
api-provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timestamp=1653703292768
       List<RegistryConfig> registries = interfaceConfig.getRegistries();
           //省略掉中间的的代码...
        return genCompatibleRegistries(interfaceConfig.getScopeModel(), registryList,
provider);
    }
```

ConfigValidationUtils 的双注册地址的获取 genCompatibleRegistries 方法.
前面代码获取到了一个注册中心地址列表例如：

```
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-api-provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timestamp=1653703292768
```

下面可以看下如果根据配置来转换为应用级注册地址+接口级注册地址

```
private static List<URL> genCompatibleRegistries(ScopeModel scopeModel, List<URL> registryList, boolean provider) {
        List<URL> result = new ArrayList<>(registryList.size());
        registryList.forEach(registryURL -> {
            if (provider) {
                // for registries enabled service discovery, automatically register interface compatible addresses.
                String registerMode;
                if (SERVICE_REGISTRY_PROTOCOL.equals(registryURL.getProtocol())) {
                    //为了更好理解这里简化掉服务发现注册地址配置的逻辑判断过程仅仅看当前例子提供的值走的逻辑
                } else {
                    //双注册模式配置查询 对应参数为 dubbo.application.register-mode 默认值为 all
                    registerMode = registryURL.getParameter(REGISTER_MODE_KEY, ConfigurationUtils.getCachedDynamicProperty(scopeModel, DUBBO_REGISTER_MODE_DEFAULT_KEY, DEFAULT_REGISTER_MODE_ALL));
                    //如果用户配置了一个错误的注册模式配置则只走接口级配置 这里默认值为 interface
                    if (!isValidRegisterMode(registerMode)) {
                        registerMode = DEFAULT_REGISTER_MODE_INTERFACE;
                    }
                    //这个逻辑是满足应用级注册就添加一个应用级注册的地址
                    if ((DEFAULT_REGISTER_MODE_INSTANCE.equalsIgnoreCase(registerMode) || DEFAULT_REGISTER_MODE_ALL.equalsIgnoreCase(registerMode))
                        && registryNotExists(registryURL, registryList, SERVICE_REGISTRY_PROTOCOL)) {
                        URL serviceDiscoveryRegistryURL = URLBuilder.from(registryURL)
                            .setProtocol(SERVICE_REGISTRY_PROTOCOL)
                            .removeParameter(REGISTRY_TYPE_KEY)
                            .build();
                        result.add(serviceDiscoveryRegistryURL);
                    }
```

```
                                                    //这个逻辑是满足接口级注册配置就添加一
个接口级注册地址
                    if
(DEFAULT_REGISTER_MODE_INTERFACE.equalsIgnoreCase(registerMode) ||
DEFAULT_REGISTER_MODE_ALL.equalsIgnoreCase(registerMode)) {
                        result.add(registryURL);
                    }
                }
                  //省略掉若干代码和括号

        return result;
    }
```

可以看到这里简化的配置比较容易理解了

- 双注册模式配置查询，对应参数为 dubbo.application.register-mode ，默认值为 all。
- 如果用户配置了一个错误的注册模式配置则只走接口级配置 这里默认值为 interface。
- 满足应用级注册就添加一个应用级注册的地址。
- 满足接口级注册配置就添加一个接口级注册地址。

这个方法是根据服务注册模式来判断使用接口级注册地址还是应用级注册地址分别如下所示。

配置信息：dubbo.application.register-mode

配置值：

- **interface**
  接口级注册

- **instance**

  应用级注册

- **all**

  接口级别和应用级都注册

最终的注册地址配置如下，接口级注册地址：

```
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-api-provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timestamp=1653703292768
```

应用级注册地址：

```
service-discovery-registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-api-provider&dubbo=2.0.2&pid=10275&registry=zookeeper&release=3.0.8&timestamp=1653704425920
```

## 3. 双注册服务数据的注册

### 1) 双注册代码逻辑调用简介

前面说了这个注册服务的配置地址会由 Dubbo 内部进行判断如果判断是 all 的话会自动将一个配置的注册地址转变为两个一个是传统的接口级注册，一个是应用级注册使用的配置地址。

然后我们先看注册中心，注册服务数据的源码。

如果想要查看源码细节可以在 RegistryProtocol 类型的 export（final Invoker<T> originInvoker）方法的如下代码位置打断点。

RegistryProtocol 的 export 方法的注册中心注册数据代码如下：

```
        // url to registry 注册服务对外的接口
                //如果 url 为 service-discovery-registry 发现则这个实现类型为
ServiceDiscoveryRegistry
        final Registry registry = getRegistry(registryUrl);
                //服务发现的提供者 url:
dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?anyhos
t=true&application=dubbo-demo-api-
provider&background=false&deprecated=false&dubbo=2.0.2&dynamic=true&ge
neric=false&interface=link.elastic.dubbo.entity.DemoService&methods=sa
yHello,sayHelloAsync&pid=19559&release=3.0.8&service-name-
mapping=true&side=provider&timestamp=1654938441023
        final URL registeredProviderUrl = getUrlToRegistry(providerUrl,
registryUrl);

        // decide if we need to delay publish (provider itself and
registry should both need to register)
        //register 参数是否 注册数据到注册中心
        boolean register = providerUrl.getParameter(REGISTER_KEY, true)
&& registryUrl.getParameter(REGISTER_KEY, true);
        if (register) {
                        //这里有两种情况 接口级注册会将接口级服务提供者数据直接注
册到 Zookeper 上面，服务发现（应用级注册）这里仅仅会将注册数据转换为服务元数据等后面
来发布元数据
            register(registry, registeredProviderUrl);
        }

        // register stated url on provider model
        //向提供者模型注册提供者配置 ProviderModel
        registerStatedUrl(registryUrl, registeredProviderUrl, register);


        exporter.setRegisterUrl(registeredProviderUrl);
        exporter.setSubscribeUrl(overrideSubscribeUrl);

        if (!registry.isServiceDiscovery()) {
            // Deprecated! Subscribe to override rules in 2.6.x or
before.
            registry.subscribe(overrideSubscribeUrl,
overrideSubscribeListener);
        }
```

在上个博客中我们整体说了下服务注册时候的一个流程，关于数据向注册中心的注册细节这里可以详细看下。

### 2) 注册中心领域对象的初始化

前面的代码使用 url 来获取注册中心操作对象如下调用代码：

```
// url to registry 注册服务对外的接口
final Registry registry = getRegistry(registryUrl);
```

对应代码如下：

```
 protected Registry getRegistry(final URL registryUrl) {
        //这里分为两步先获取注册中心工厂对象
        RegistryFactory registryFactory =
ScopeModelUtil.getExtensionLoader(RegistryFactory.class,
registryUrl.getScopeModel()).getAdaptiveExtension();
        //使用注册中心工厂对象获取注册中心操作对象
        return registryFactory.getRegistry(registryUrl);
    }
```

关于参数 URL 有两个在前面已经说过，url 信息如下：

接口级注册地址：

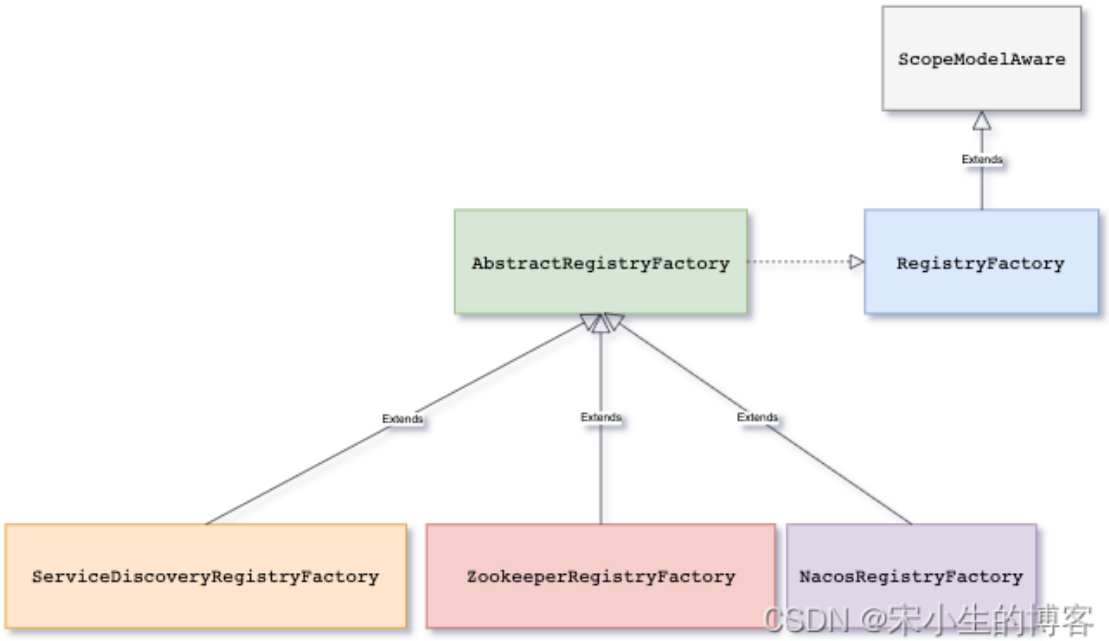
```
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-
api-provider&dubbo=2.0.2&pid=9008&registry=zookeeper&release=3.0.8&timestamp=1653703292768
```

应用级注册地址：

```
service-discovery-registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
provider&dubbo=2.0.2&pid=10275&registry=zookeeper&release=3.0.8&timestamp=1653704425920
```

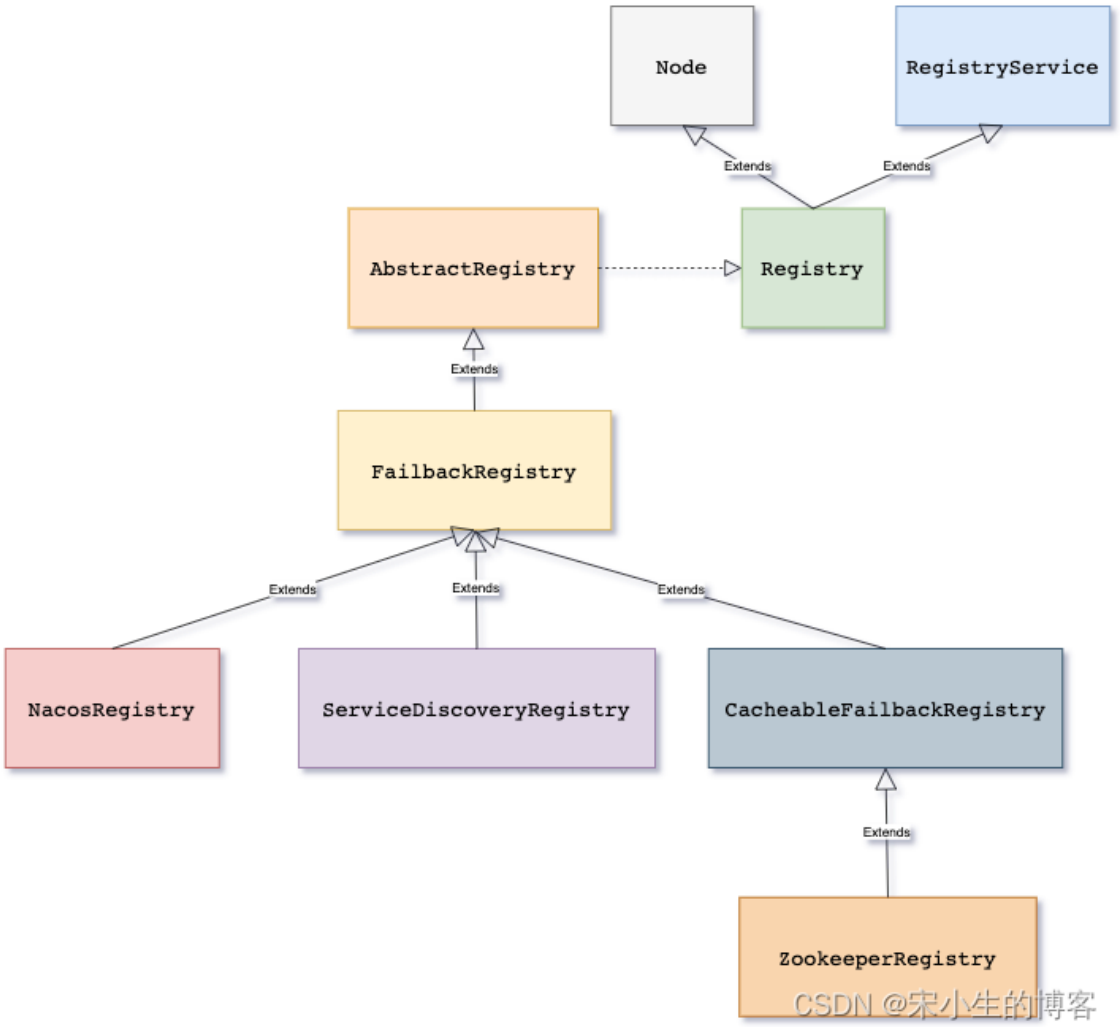
注册中心工厂对象与注册中心操作对象的获取与执行我们通过 Debug 来看比较麻烦，这里涉及到很多扩展机制动态生成的代码我们无法看到，这里我直接来贴一下比较关键的一些类型，以 Zookeeper 注册中心来举例子：

先来看下注册工厂相关的类型：




- RegistryFactory 注册中心对象获取
- AbstractRegistryFactory模板类型封装注册中心对象获取的基本逻辑，比如缓存和基本的逻辑判断
- ServiceDiscoveryRegistryFactory 用于创建服务发现注册中心工厂对象 用于创建 ServiceDiscoveryRegistry 对象
- ZookeeperRegistryFactory 用于创建 ZookeeperRegistry 类型对象
- NacosRegistryFactory Nacos 注册中心工厂对象，用于创建 NacosRegistry

接下来看封装了注册中心操作逻辑的注册中心领域对象：




- Node 节点信息开放接口，比如节点 url 的获取，销毁
- RegistryService 注册服务接口，比如注册，订阅，查询等操作
- Registry 注册中心接口，是否服务发现查询，注册，取消注册方法
- AbstractRegistry 注册中心逻辑抽象模板类型，封装了注册，订阅，通知的基本逻辑，和本地缓存注册中心信息的基本逻辑
- FailbackRegistry 封装了失败重试的逻辑
- NacosRegistry 封装了以 nacos 作为注册中心的基本逻辑

- ServiceDiscoveryRegistry 应用级服务发现注册中心逻辑，现在不需要这种网桥实现，协议可以直接与服务发现交互。ServiceDiscoveryRegistry 是一种非常特殊的注册表实现，用于以兼容的方式将旧的接口级服务发现模型与 3.0 中引入的新服务发现模型连接起来。它完全符合注册表 SPI 的扩展规范，但与 zookeeper 和 Nacos 的具体实现不同，因为它不与任何真正的第三方注册表交互，而只与过程中 ServiceDiscovery 的相关组件交互。

  简而言之，它架起了旧接口模型和新服务发现模型之间的桥梁：register()方法主要通过与 MetadataService 交互，将接口级数据聚合到 MetadataInfo 中 subscribe()触发应用程序级服务发现模型的整个订阅过程。

  根据 ServiceNameMapping 将接口映射到应用程序。-启动新的服务发现侦听器(InstanceListener)，并使 NotifierListener 成为 InstanceListener 的一部分。

- CacheableFailbackRegistry 提供了一些本地内存缓存的逻辑，对注册中心有用，注册中心的 sdk 将原始字符串作为提供程序实例返回，例如 zookeeper 和 etcd

- ZookeeperRegistry Zookeeper 作为注册中心的基本操作逻辑封装

了解了这几个领域对象这里我们回到代码逻辑，这里直接看将会执行的一些核心逻辑：

```
 protected Registry getRegistry(final URL registryUrl) {
        //这里分为两步先获取注册中心工厂对象
        RegistryFactory registryFactory =
ScopeModelUtil.getExtensionLoader(RegistryFactory.class,
registryUrl.getScopeModel()).getAdaptiveExtension();
        //使用注册中心工厂对象获取注册中心操作对象
        return registryFactory.getRegistry(registryUrl);
    }
```

前面注册中心工厂不论那种协议的地址信息获取到的都是一个 RegistryFactory$Adaptive 类型（由扩展机制的字节码工具自动生成的代码）

如果 getRegistry 参数为应用级注册地址。如下所示将获取到的类型为 ServiceDiscoveryRegistryFactory 逻辑来获取注册中心，(这个逻辑是@Adaptive 注解产生的了逻辑具体原理可以看扩展机制中@Adaptive 的实现）。

```
service-discovery-registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
provider&dubbo=2.0.2&pid=10275&registry=zookeeper&release=3.0.8&timestamp=1653704425920
```

getRegistry 方法优先走的逻辑是这里：AbstractRegistryFactory 模板类型中的 getRegistry 方法

```
@Override
    public Registry getRegistry(URL url) {
        if (registryManager == null) {
            throw new IllegalStateException("Unable to fetch
RegistryManager from ApplicationModel BeanFactory. " +
                "Please check if `setApplicationModel` has been
override.");
        }
                  //销毁状态直接返回
        Registry defaultNopRegistry =
registryManager.getDefaultNopRegistryIfDestroyed();
        if (null != defaultNopRegistry) {
            return defaultNopRegistry;
        }

        url = URLBuilder.from(url)
            .setPath(RegistryService.class.getName())
            .addParameter(INTERFACE_KEY,
RegistryService.class.getName())
            .removeParameter(TIMESTAMP_KEY)
            .removeAttribute(EXPORT_KEY)
            .removeAttribute(REFER_KEY)
            .build();
          //这个 key 为 service-discovery-
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService
        String key = createRegistryCacheKey(url);
        Registry registry = null;
        //check 配置 是否检查注册中心连通 默认为 true
        boolean check = url.getParameter(CHECK_KEY, true) &&
url.getPort() != 0;
```

```
        // Lock the registry access process to ensure a single instance
of the registry
        //给写操作加锁方式并发写问题
        registryManager.getRegistryLock().lock();
        try {
             //锁内检查是否销毁的逻辑
            // double check
            // fix https://github.com/apache/dubbo/issues/7265.
            defaultNopRegistry =
registryManager.getDefaultNopRegistryIfDestroyed();
            if (null != defaultNopRegistry) {
                return defaultNopRegistry;
            }
            //锁内检查是否缓存中存在存在则直接返回
            registry = registryManager.getRegistry(key);
            if (registry != null) {
                return registry;
            }
            //create registry by spi/ioc
            //使用 url 创建注册中心操作的逻辑
            registry = createRegistry(url);
        } catch (Exception e) {
           //check 配置检查
            if (check) {
                throw new RuntimeException("Can not create registry " +
url, e);
            } else {
                LOGGER.warn("Failed to obtain or create registry ", e);
            }
        } finally {
            // Release the lock
            registryManager.getRegistryLock().unlock();
        }

        if (check && registry == null) {
            throw new IllegalStateException("Can not create registry " +
url);
        }
                //缓存逻辑
        if (registry != null) {
            registryManager.putRegistry(key, registry);
        }
        return registry;
```

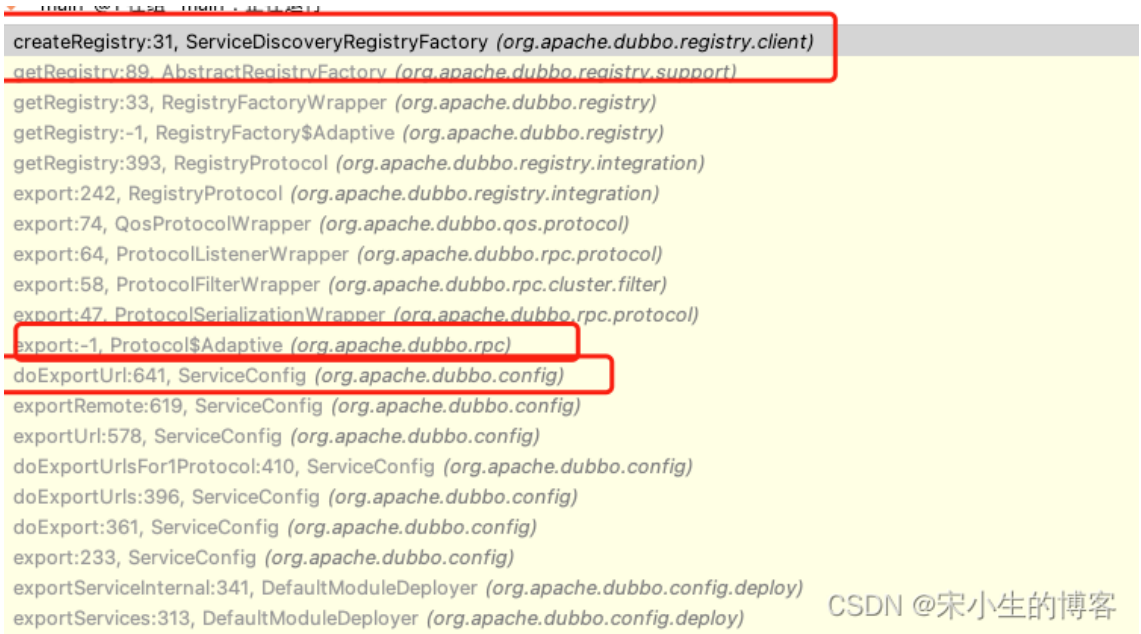
```
    }
```

逻辑其实吧比较简单，概括下上面的逻辑：

- 销毁逻辑判断
- 缓存获取，存在则直接返回
- 根据注册中心 url 配置，创建注册中心操作对象
- 注册中心连接失败的 check 配置逻辑处理
- 将注册中心操作对象存入缓存

上面比较重要的逻辑是 createRegistry 这个

整个调用过程我给大家看下 Debug 的详情，这里很多逻辑由扩展机制产生的这里直接看下逻辑调用栈，有几个需要关注的地方我圈了起来：




我们继续看服务发现的注册中心工厂对象的获取，代码如下：

ServiceDiscoveryRegistryFactory 类型的 createRegistry 方法

```
@Override
   protected Registry createRegistry(URL url) {
   //判断url是否是这个前缀：service-discovery-registry
       if (UrlUtils.hasServiceDiscoveryRegistryProtocol(url)) {
          //切换下协议：将服务发现协议切换为配置的注册中心协议这里是zookeeper如下：
          //zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
provider&dubbo=2.0.2&interface=org.apache.dubbo.registry.RegistryService&pid=39884&release=3
.0.8
           String protocol = url.getParameter(REGISTRY_KEY, DEFAULT_REGISTRY);
           url = url.setProtocol(protocol).removeParameter(REGISTRY_KEY);
       }
       //创建服务发现注册中心对象对象
       return new ServiceDiscoveryRegistry(url, applicationModel);
   }
```

通过以上代码可以看到其实最终创建的是一个 ServiceDiscoveryRegistry 注册中心对象，这个 url 协议被转换为了对应注册中心的协议，也就是说双注册会有两个协议一个是原先的接口级注册注册中心对象（这个还未说到）和这里对应注册中心协议的服务发现注册中心对象 ServiceDiscoveryRegistry

### 3) ServiceDiscoveryRegistry

ServiceDiscoveryRegistry 服务发现注册中心对象的初始化过程：

a) ServiceDiscoveryRegistry 的构造器

```
   public ServiceDiscoveryRegistry(URL registryURL, ApplicationModel applicationModel) {
       super(registryURL);
       //根据url创建一个服务发现对象类型为ServiceDiscovery
       this.serviceDiscovery = createServiceDiscovery(registryURL);
       //这个类型为是serviceNameMapping类型是MetadataServiceNameMapping类型
       this.serviceNameMapping = (AbstractServiceNameMapping)
ServiceNameMapping.getDefaultExtension(registryURL.getScopeModel());
       super.applicationModel = applicationModel;
   }
```

ServiceDiscoveryRegistry 中创建服务发现对象 createServiceDiscovery 方法

```
protected ServiceDiscovery createServiceDiscovery(URL registryURL) {
       return getServiceDiscovery(registryURL.addParameter(INTERFACE_KEY,
ServiceDiscovery.class.getName())
           .removeParameter(REGISTRY_TYPE_KEY));
   }
```

ServiceDiscoveryRegistry 中创建服务发现对象 getServiceDiscovery 方法

```
private ServiceDiscovery getServiceDiscovery(URL registryURL) {
//服务发现工厂对象的获取这里是ServiceDiscoveryFactory类型,
        ServiceDiscoveryFactory factory = getExtension(registryURL);
        //服务发现工厂对象获取服务发现对象
        return factory.getServiceDiscovery(registryURL);
    }
```

ServiceDiscoveryFactory 和 ServiceDiscovery 类型可以往后看

b) 父类型 FailbackRegistry 的构造器

```
   public FailbackRegistry(URL url) {
        super(url);
         //重试间隔配置retry.period ，默认为5秒
        this.retryPeriod = url.getParameter(REGISTRY_RETRY_PERIOD_KEY,
DEFAULT_REGISTRY_RETRY_PERIOD);

        // since the retry task will not be very much. 128 ticks is enough.
        //因为重试任务不会太多。128个刻度就足够了。Dubbo封装的时间轮用于高效率的重试，这个在Kafka也自定义
实现了后续可以单独来看看
        retryTimer = new HashedWheelTimer(new NamedThreadFactory("DubboRegistryRetryTimer",
true), retryPeriod, TimeUnit.MILLISECONDS, 128);
    }
```

c) AbstractRegistry 的构造器

参数 url 如下所示：

```
zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-
api-
provider&dubbo=2.0.2&interface=org.apache.dubbo.registry.RegistryService&pid=39884&release=3
.0.8
```

```
public AbstractRegistry(URL url) {
        setUrl(url);
        registryManager =
url.getOrDefaultApplicationModel().getBeanFactory().getBean(RegistryMa
nager.class);
        //是否本地缓存默认为 true
```

```
        localCacheEnabled =
url.getParameter(REGISTRY_LOCAL_FILE_CACHE_ENABLED, true);
        registryCacheExecutor =
url.getOrDefaultFrameworkModel().getBeanFactory()
            .getBean(FrameworkExecutorRepository.class).getSharedExecuto
r();
        if (localCacheEnabled) {
            // Start file save timer 是否同步缓存默认为 false
            syncSaveFile = url.getParameter(REGISTRY_FILESAVE_SYNC_KEY,
false);
             //默认缓存的文件路径与文件名字为：/Users/song/.dubbo/dubbo-
registry-dubbo-demo-api-provider-127.0.0.1-2181.cache
            String defaultFilename = System.getProperty(USER_HOME) +
DUBBO_REGISTRY +
                url.getApplication() + "-" +
url.getAddress().replaceAll(":", "-") + CACHE;
              //未指定缓存的文件名字则用默认的文件名字
            String filename = url.getParameter(FILE_KEY,
defaultFilename);
            File file = null;
            //父目录创建，保证目录存在
            if (ConfigUtils.isNotEmpty(filename)) {
                file = new File(filename);
                if (!file.exists() && file.getParentFile() != null
&& !file.getParentFile().exists()) {
                    if (!file.getParentFile().mkdirs()) {
                        throw new IllegalArgumentException("Invalid
registry cache file " + file + ", cause: Failed to create directory "
+ file.getParentFile() + "!");
                    }
                }
            }
            this.file = file;
            // When starting the subscription center,
            // we need to read the local cache file for future Registry
fault tolerance processing.
            //加载本地磁盘文件
            loadProperties();
            //变更推送
            notify(url.getBackupUrls());
        }
    }
```

### 4) 将服务提供者数据转换到本地内存的元数据信息中

在前面我们看到了 RegistryProtocol 中调用 register 来注册服务提供者的数据到注册的中心，接下来详细看下实现原理。

下面参数为 ServiceDiscoveryRegistry 为情况下举例子：ServiceDiscoveryRegistry 类型的 register 方法与 ZookeeperRegister 注册不一样传统的接口级注册在这个方法里面就将服务数据注册到注册中心了，服务发现的数据注册分为了两步，这里仅仅将数据封装到内存中如下。

url 例子为：

```
dubbo://192.168.1.9:20880/link.elastic.dubbo.entity.DemoService?
anyhost=true&application=dubbo-demo-api-
provider&background=false&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=
link.elastic.dubbo.entity.DemoService&methods=sayHello,sayHelloAsync&pid=19559&release=3.0.8
&service-name-mapping=true&side=provider&timestamp=1654938441023
```

RegistryProtocol 中的 register 方法：

```
private void register(Registry registry, URL registeredProviderUrl) {
       registry.register(registeredProviderUrl);
   }
```

上面这个代码会优先走 ListenerRegistryWrapper 的一些逻辑来执行 register 方法来触发一些监听器的逻辑，我们直接跳到 ServiceDiscoveryRegistry 中的 register 方法来看 ServiceDiscoveryRegistry 的 register 方法。

```
@Override
    public final void register(URL url) {
       //逻辑判断比如只有side为提供者时候才能注册
        if (!shouldRegister(url)) { // Should Not Register
            return;
        }
        doRegister(url);
    }
```

ServiceDiscoveryRegistry 的 doRegister 方法：

```
    @Override
    public void doRegister(URL url) {
        // fixme, add registry-cluster is not necessary anymore
        url = addRegistryClusterKey(url);
        serviceDiscovery.register(url);
    }
```

AbstractServiceDiscovery 的 register 方法：

```
@Override
    public void register(URL url) {
      //metadaInfo类型为MetadataInfo类型，用来操作元数据的
        metadataInfo.addService(url);
    }
```

MetadataInfo 类型的 addService 方法：

```
public synchronized void addService(URL url) {
        // fixme, pass in application mode context during initialization of MetadataInfo.
        //元数据参数过滤器扩展获取:MetadataParamsFilter
        if (this.loader == null) {
            this.loader =
url.getOrDefaultApplicationModel().getExtensionLoader(MetadataParamsFilter.class);
        }
        //元数据参数过滤器获取
        List<MetadataParamsFilter> filters = loader.getActivateExtension(url, "params-
filter");
        // generate service level metadata
        //生成服务级别的元数据
        ServiceInfo serviceInfo = new ServiceInfo(url, filters);
        this.services.put(serviceInfo.getMatchKey(), serviceInfo);
        // extract common instance level params
        extractInstanceParams(url, filters);

        if (exportedServiceURLs == null) {
            exportedServiceURLs = new ConcurrentSkipListMap<>();
        }
        addURL(exportedServiceURLs, url);
        updated = true;
    }
```

### 5) 接口级服务提供者配置的注册

前面我们通过服务发现的的 url 进行了举例子，其实在 RegistryProtocol 协议的 export 方法中还会注册接口级信息。

例如如下关键代码：

当 registryUrl 参数不是服务发现协议 service-discovery-registry 配置而是 zookeeper 如下时候获取到的扩展类型将是与 Zookeeper 相关的扩展对象

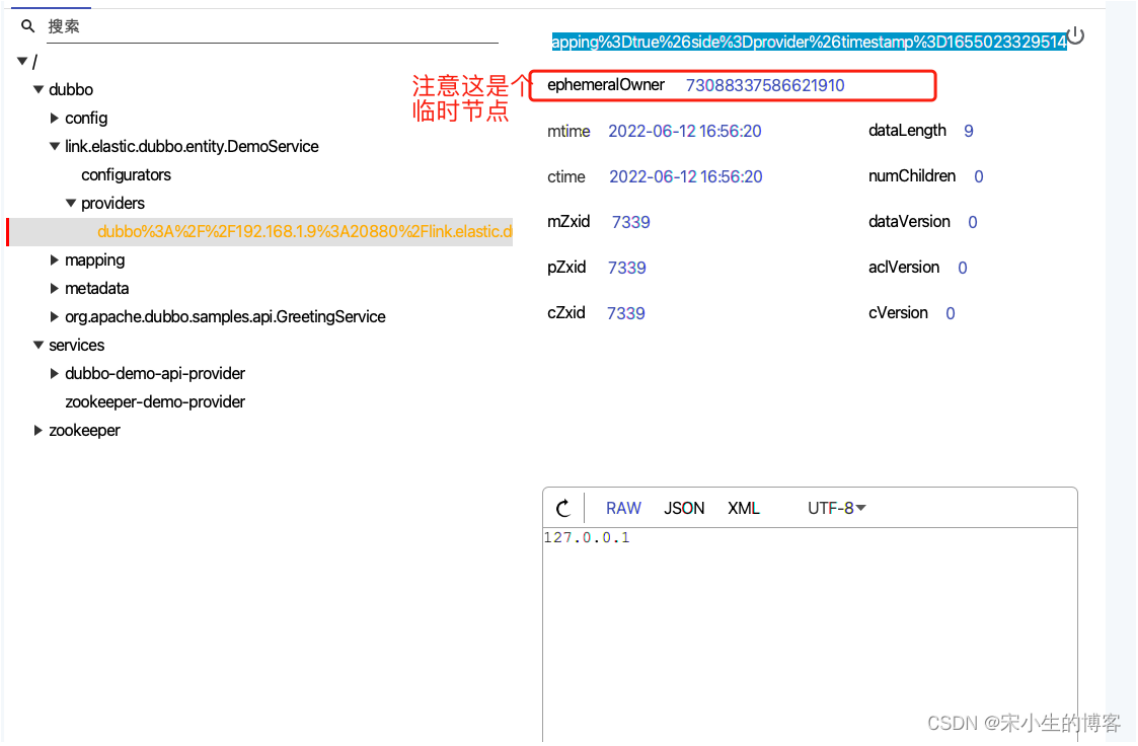
```
zookeeper://8.131.79.126:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-api-provider&dubbo=2.0.2&pid=29386&release=3.0.8&timestamp=1655023329438
```

RegistryProtocol 协议的 export 方法中接口级数据注册的核心代码如下。

如下代码的操作类型可以看注释

```
// url to registry 这里registry对象的类型为ZookeeperRegistry
        final Registry registry = getRegistry(registryUrl);

        final URL registeredProviderUrl = getUrlToRegistry(providerUrl, registryUrl);
        // decide if we need to delay publish (provider itself and registry should both need to register)
        boolean register = providerUrl.getParameter(REGISTER_KEY, true) && registryUrl.getParameter(REGISTER_KEY, true);
        //这一个方法里面会将提供者的url配置写入zookeeper的provider节点下面
        if (register) {
            register(registry, registeredProviderUrl);
        }
```

如上代码是获取 Zookeeper 操作对象和向 Zookeeper 中写入服务提供者信息的代码，关于与 Zookeeper 连接和注册数据本地缓存的代码可以看 ZookeeperRegistry 类型和它的几个父类型比如：CacheableFailbackRegistry 类型，关于接口级数据的注册可以看 register 方法，这个就不详细说了，下面我贴一下接口级数据注册的 Zookeeper 信息可以了解下就行：

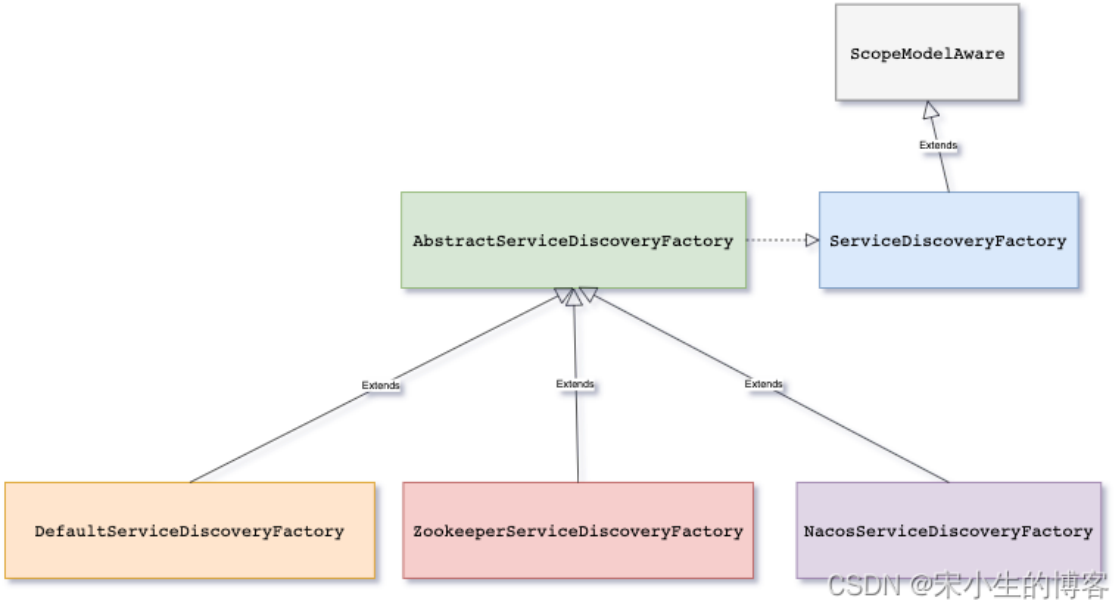


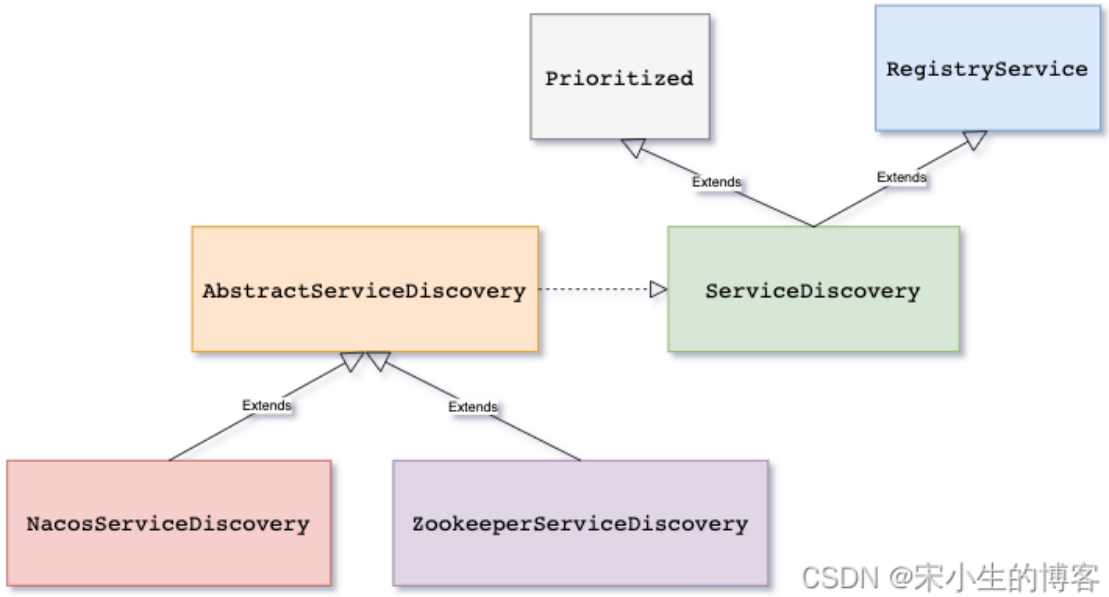

接口信息如下，上面我们需要注意的是这个 url 配置为临时节点，当与 Zookeeper 断开连接或者 Session 超时的时候这个信息会被移除：

```
/dubbo/link.elastic.dubbo.entity.DemoService/providers/dubbo%3A%2F%2F192.168.1.9%3A20880%2Fl
ink.elastic.dubbo.entity.DemoService%3Fanyhost%3Dtrue%26application%3Ddubbo-demo-api-
provider%26background%3Dfalse%26deprecated%3Dfalse%26dubbo%3D2.0.2%26dynamic%3Dtrue%26generi
c%3Dfalse%26interface%3Dlink.elastic.dubbo.entity.DemoService%26methods%3DsayHello%2CsayHell
oAsync%26pid%3D29386%26release%3D3.0.8%26service-name-
mapping%3Dtrue%26side%3Dprovider%26timestamp%3D1655023329514
```

## 4. 应用级服务发现功能的实现 ServiceDiscovery

在说这个实现之前我们先看看相关类型，这个服务发现相关的类型与注册中心相关的类型有点类似，服务发现工厂类型：




刚刚在 ServiceDiscoveryRegistry 中创建服务发现对象 getServiceDiscovery 方法看到了两个类型一个是服务发现工厂类型 ServiceDiscoveryFactory，一个是服务发现类型 ServiceDiscovery

```
private ServiceDiscovery getServiceDiscovery(URL registryURL) {
//服务发现工厂对象的获取这里是ServiceDiscoveryFactory类型，这里对应ZookeeperServiceDiscoveryFactory
        ServiceDiscoveryFactory factory = getExtension(registryURL);
        //服务发现工厂对象获取服务发现对象
        return factory.getServiceDiscovery(registryURL);
    }
```

AbstractServiceDiscoveryFactory 类型的 getServiceDiscovery 方法

```
    @Override
     public ServiceDiscovery getServiceDiscovery(URL registryURL) {
     //这个key是 zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.client.ServiceDiscovery
      //一个地址需要创建一个服务发现对象
         String key = registryURL.toServiceStringWithoutResolving();
         return discoveries.computeIfAbsent(key, k -> createDiscovery(registryURL));
     }
```

createDiscovery 方法对应 ZookeeperServiceDiscoveryFactory 类型中的 createDiscovery 方法，如下代码所示：

```
@Override
    protected ServiceDiscovery createDiscovery(URL registryURL) {
        return new ZookeeperServiceDiscovery(applicationModel, registryURL);
    }
```

### 1) ZookeeperServiceDiscovery

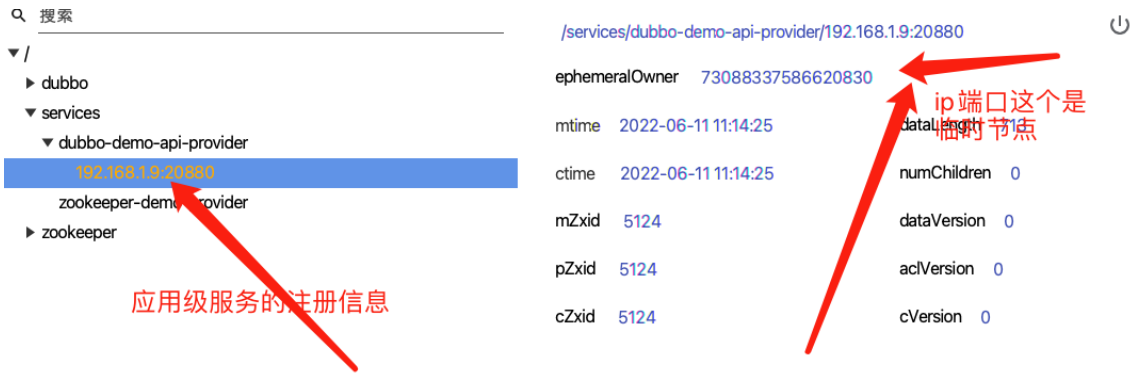
ZookeeperServiceDiscovery 的构造器

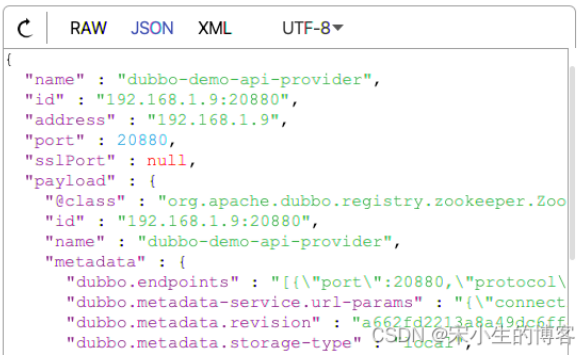
```
  public ZookeeperServiceDiscovery(ApplicationModel applicationModel, URL registryURL) {
        //先调用父类AbstractServiceDiscovery 模板类构造器
        super(applicationModel, registryURL);
        try {
          //创建 创建CuratorFramework 类型对象用于操作Zookeeper
            this.curatorFramework = buildCuratorFramework(registryURL);
            //获取应用级服务发现的根路径 值为/services 这个可以在Zookeeper上面看到
            this.rootPath = ROOT_PATH.getParameterValue(registryURL);
             //创建服务发现对象 实现类型为ServiceDiscoveryImpl 这个实现来源于Curator框架中的
discovery模块
            this.serviceDiscovery = buildServiceDiscovery(curatorFramework, rootPath);
            //启动服务发现
            this.serviceDiscovery.start();
        } catch (Exception e) {
            throw new IllegalStateException("Create zookeeper service discovery failed.",
e);
        }
    }
```

这个方法比较重要是应用级服务发现的实现，这里主要关注下 serviceDiscovery 类型的创建与启动，这个应用级服务发现的实现其实是 Dubbo 使用了 Curator 来做的，Dubbo 只是在这里封装了一些方法来进行调用 Curator 的实现。

- 关于 Curator 的官方文档可以看 curator 官网

- 关于 Zookeeper 上面注册服务应用级服务注册信息可以看如下图所示（后面会具体讲到数据注册时的调用）




我这个服务提供者注册的应用数据如下：

```
{
  "name" : "dubbo-demo-api-provider",
  "id" : "192.168.1.9:20880",
  "address" : "192.168.1.9",
  "port" : 20880,
  "sslPort" : null,
  "payload" : {
    "@class" : "org.apache.dubbo.registry.zookeeper.ZookeeperInstance",
    "id" : "192.168.1.9:20880",
    "name" : "dubbo-demo-api-provider",
    "metadata" : {
      "dubbo.endpoints" : "[{\"port\":20880,\"protocol\":\"dubbo\"}]",
      "dubbo.metadata-service.url-params" : "
{\"connections\":\"1\",\"version\":\"1.0.0\",\"dubbo\":\"2.0.2\",\"release\":\"3.0.8\",\"sid
e\":\"provider\",\"port\":\"20880\",\"protocol\":\"dubbo\"}",
      "dubbo.metadata.revision" : "a662fd2213a8a49dc6ff43a4c2ae7b9e",
      "dubbo.metadata.storage-type" : "local",
      "timestamp" : "1654916298616"
    }
  },
  "registrationTimeUTC" : 1654917265499,
  "serviceType" : "DYNAMIC",
  "uriSpec" : null
}
```


如果感兴趣的话可以看更详细的 curator 服务发现文档 curator-x-discovery

### 2) AbstractServiceDiscovery 的构造器

```
  public AbstractServiceDiscovery(ApplicationModel applicationModel, URL registryURL) {
        //调用重载的构造器
        this(applicationModel.getApplicationName(), registryURL);
        this.applicationModel = applicationModel;
        MetadataReportInstance metadataReportInstance =
applicationModel.getBeanFactory().getBean(MetadataReportInstance.class);
        metadataType = metadataReportInstance.getMetadataType();
        this.metadataReport =
metadataReportInstance.getMetadataReport(registryURL.getParameter(REGISTRY_CLUSTER_KEY));
//        if (REMOTE_METADATA_STORAGE_TYPE.equals(metadataReportInstance.getMetadataType()))
{
//            this.metadataReport =
metadataReportInstance.getMetadataReport(registryURL.getParameter(REGISTRY_CLUSTER_KEY));
//        } else {
//            this.metadataReport = metadataReportInstance.getNopMetadataReport();
//        }
    }
```

重载的构造器

```
    public AbstractServiceDiscovery(String serviceName, URL registryURL) {
        this.applicationModel = ApplicationModel.defaultModel();
        //这个url参考：zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
provider&dubbo=2.0.2&interface=org.apache.dubbo.registry.client.ServiceDiscovery&pid=4570&re
lease=3.0.8
        this.registryURL = registryURL;
        //这个serviceName参考dubbo-demo-api-provider
        this.serviceName = serviceName;
        //MetadataInfo 用来封装元数据信息
        this.metadataInfo = new MetadataInfo(serviceName);
        //这个是元数据缓存信息管理的类型 缓存文件使用LRU策略  感兴趣的可以详细看看
        //对应缓存路径为：/Users/song/.dubbo/.metadata.zookeeper127.0.0.1%003a2181.dubbo.cache
        this.metaCacheManager = new MetaCacheManager(getCacheNameSuffix(),
            applicationModel.getFrameworkModel().getBeanFactory()

.getBean(FrameworkExecutorRepository.class).getCacheRefreshingScheduledExecutor());
    }
```

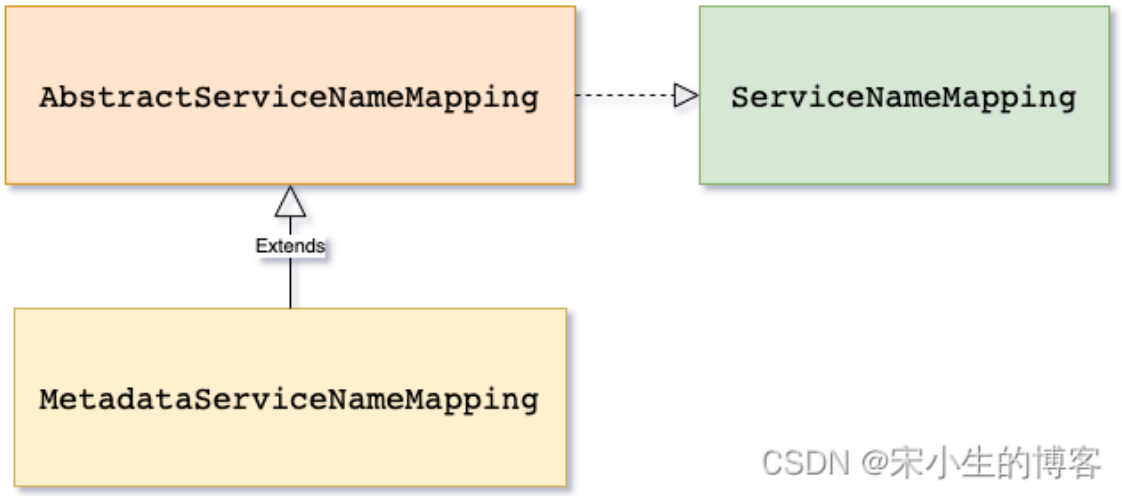
## 5. 服务映射类型 AbstractServiceNameMapping

服务映射主要是通过服务名字来反查应用信息的应用名字如下图所示


这里我们来看下服务映射相关的类型主要通过如下代码来获取扩展对象：

```
this.serviceNameMapping = (AbstractServiceNameMapping)
ServiceNameMapping.getDefaultExtension(registryURL.getScopeModel());
```

对应类型如下：




最终获取的扩展实现类型为：MetadataServiceNameMapping
构造器如下：

```
   public MetadataServiceNameMapping(ApplicationModel applicationModel) {
        super(applicationModel);
        metadataReportInstance =
applicationModel.getBeanFactory().getBean(MetadataReportInstance.class);
    }
```

服务映射元数据父类型 AbstractServiceNameMapping 如下：

```
 public AbstractServiceNameMapping(ApplicationModel applicationModel) {
        this.applicationModel = applicationModel;
        //LRU缓存保存服务映射数据
        this.mappingCacheManager = new MappingCacheManager("",
            applicationModel.getFrameworkModel().getBeanFactory()

.getBean(FrameworkExecutorRepository.class).getCacheRefreshingScheduledExecutor());
    }
```

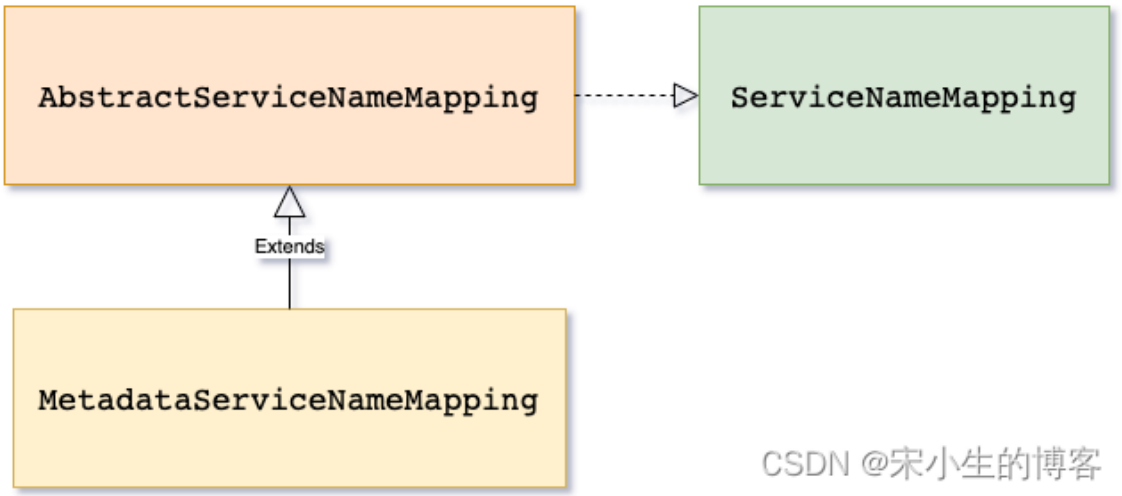
## 6. 双注册元数据信息发布到注册中心

服务映射主要是通过服务名字来反查应用信息的应用名字如下图所示


这里我们来看下服务映射相关的类型主要通过如下代码来获取扩展对象：

```
this.serviceNameMapping = (AbstractServiceNameMapping)
ServiceNameMapping.getDefaultExtension(registryURL.getScopeModel());
```

对应类型如下：




最终获取的扩展实现类型为：MetadataServiceNameMapping

构造器如下：

```
   public MetadataServiceNameMapping(ApplicationModel applicationModel) {
        super(applicationModel);
        metadataReportInstance =
applicationModel.getBeanFactory().getBean(MetadataReportInstance.class);
    }
```

服务映射元数据父类型 AbstractServiceNameMapping 如下：

```
 public AbstractServiceNameMapping(ApplicationModel applicationModel) {
        this.applicationModel = applicationModel;
        //LRU缓存保存服务映射数据
        this.mappingCacheManager = new MappingCacheManager("",
            applicationModel.getFrameworkModel().getBeanFactory()

.getBean(FrameworkExecutorRepository.class).getCacheRefreshingScheduledExecutor());
    }
```

## 7. 双注册元数据信息发布到注册中心

### 1) 回顾简介

前面注册数据的时候并没有把服务配置的元数据直接注册在注册中心而是需要在导出服务之后在 ServiceConfig 中来发布元数据，这个就需要我们回到 ServiceConfig 的 exportUrl 方法来看了如下所示：

```
private void exportUrl(URL url, List<URL> registryURLs) {
        String scope = url.getParameter(SCOPE_KEY);
        // don't export when none is configured
        ...省略到若干代码
            if (!SCOPE_LOCAL.equalsIgnoreCase(scope)) {
                url = exportRemote(url, registryURLs);
                if (!isGeneric(generic) && !getScopeModel().isInternal()) {
                    MetadataUtils.publishServiceDefinition(url,
providerModel.getServiceModel(), getApplicationModel());
                }
            }
        }
        this.urls.add(url);
    }
```

### 2) 元数据服务定义数据的发布

在 exportRemote 之后单独调用发布元数据的方法来发布，通过调用元数据工具类来发布元数据信息接下来我们详细看下。

MetadataUtils 类型的 publishServiceDefinition 方法：

```
public static void publishServiceDefinition(URL url, ServiceDescriptor
serviceDescriptor, ApplicationModel applicationModel) {
                //查询是否存在元数据存储对象 对应接口 MetadataReport 这里对应实
现类 ZookeeperMetadataReport
       if (getMetadataReports(applicationModel).size() == 0) {
           String msg = "Remote Metadata Report Server is not provided
or unavailable, will stop registering service definition to remote
center!";
           logger.warn(msg);
       }

       try {
           String side = url.getSide();
            //服务提供者走这个逻辑
           if (PROVIDER_SIDE.equalsIgnoreCase(side)) {
               String serviceKey = url.getServiceKey();
               //获取当前服务元数据信息
               FullServiceDefinition serviceDefinition =
serviceDescriptor.getFullServiceDefinition(serviceKey);

               if (StringUtils.isNotEmpty(serviceKey) &&
serviceDefinition != null) {
                   serviceDefinition.setParameters(url.getParameters());
                   for (Map.Entry<String, MetadataReport> entry :
getMetadataReports(applicationModel).entrySet()) {
                       MetadataReport metadataReport = entry.getValue();
                       if (!metadataReport.shouldReportDefinition()) {
                           logger.info("Report of service definition is
disabled for " + entry.getKey());
                           continue;
                       }
                       //存储服务提供者的元数据  metadataReport 类型为
ZookeeperMetadataReport 方法来源于父类模板方法： AbstractMetadataReport 类型
的 storeProviderMetadata 模板方法
```

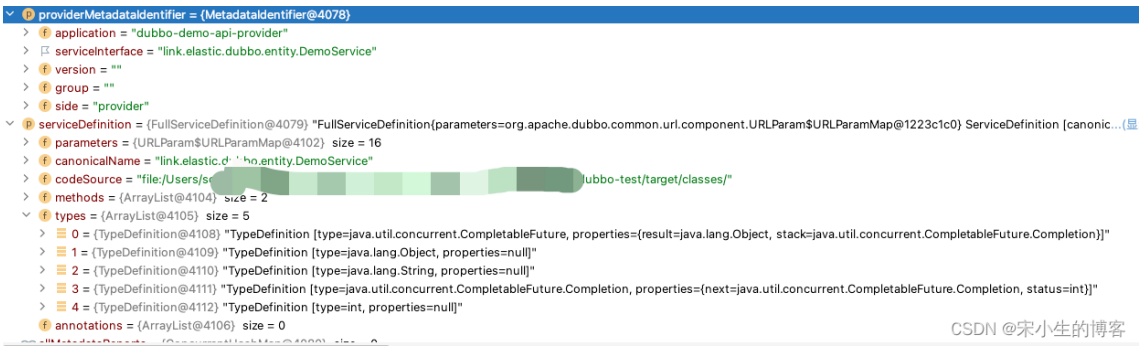
```
                    metadataReport.storeProviderMetadata(
                        new MetadataIdentifier(
                            url.getServiceInterface(),
                            url.getVersion() == null ? "" :
url.getVersion(),
                            url.getGroup() == null ? "" :
url.getGroup(),
                            PROVIDER_SIDE,
                            applicationModel.getApplicationName())
                        , serviceDefinition);
                }
            }
        } else {
          //服务消费者走这个逻辑
            for (Map.Entry<String, MetadataReport> entry :
getMetadataReports(applicationModel).entrySet()) {
                MetadataReport metadataReport = entry.getValue();
                if (!metadataReport.shouldReportDefinition()) {
                    logger.info("Report of service definition is
disabled for " + entry.getKey());
                    continue;
                }
                metadataReport.storeConsumerMetadata(
                    new MetadataIdentifier(
                        url.getServiceInterface(),
                        url.getVersion() == null ? "" :
url.getVersion(),
                        url.getGroup() == null ? "" : url.getGroup(),
                        CONSUMER_SIDE,
                        applicationModel.getApplicationName()),
                    url.getParameters());
            }
        }
    } catch (Exception e) {
        //ignore error
        logger.error("publish service definition metadata error.",
e);
    }
}
```

AbstractMetadataReport 的 storeProviderMetadata 方法如下所示：

```
    @Override
    public void storeProviderMetadata(MetadataIdentifier providerMetadataIdentifier,
ServiceDefinition serviceDefinition) {
        //是否同步配置对应sync-report 默认为异步
        if (syncReport) {
            storeProviderMetadataTask(providerMetadataIdentifier, serviceDefinition);
        } else {
            reportCacheExecutor.execute(() ->
storeProviderMetadataTask(providerMetadataIdentifier, serviceDefinition));
        }
    }
```

AbstractMetadataReport 的存储元数据方法 storeProviderMetadataTask

```
private void storeProviderMetadataTask(MetadataIdentifier providerMetadataIdentifier,
ServiceDefinition serviceDefinition) {
        try {
            if (logger.isInfoEnabled()) {
                logger.info("store provider metadata. Identifier : " +
providerMetadataIdentifier + "; definition: " + serviceDefinition);
            }
            allMetadataReports.put(providerMetadataIdentifier, serviceDefinition);
            failedReports.remove(providerMetadataIdentifier);
            Gson gson = new Gson();
            String data = gson.toJson(serviceDefinition);
            doStoreProviderMetadata(providerMetadataIdentifier, data);
            saveProperties(providerMetadataIdentifier, data, true, !syncReport);
        } catch (Exception e) {
            // retry again. If failed again, throw exception.
            failedReports.put(providerMetadataIdentifier, serviceDefinition);
            metadataReportRetry.startRetryTask();
            logger.error("Failed to put provider metadata " + providerMetadataIdentifier + "
in  " + serviceDefinition + ", cause: " + e.getMessage(), e);
        }
    }
```





元数据信息如下：可以分为两类，应用元数据，服务元数据

```
{
        "parameters": {
                "side": "provider",
                "interface": "link.elastic.dubbo.entity.DemoService",
                "pid": "22099",
                "application": "dubbo-demo-api-provider",
                "dubbo": "2.0.2",
                "release": "3.0.8",
                "anyhost": "true",
                "bind.ip": "192.168.1.9",
                "methods": "sayHello,sayHelloAsync",
                "background": "false",
                "deprecated": "false",
                "dynamic": "true",
                "service-name-mapping": "true",
                "generic": "false",
                "bind.port": "20880",
                "timestamp": "1654942353902"
        },
        "canonicalName": "link.elastic.dubbo.entity.DemoService",
        "codeSource": "file:/Users/song/Desktop/dubbo-
test/target/classes/",
        "methods": [{
                "name": "sayHelloAsync",
                "parameterTypes": ["java.lang.String"],
                "returnType": "java.util.concurrent.CompletableFuture",
                "annotations": []
        }, {
                "name": "sayHello",
                "parameterTypes": ["java.lang.String"],
                "returnType": "java.lang.String",
                "annotations": []
        }],
        "types": [{
                "type": "java.util.concurrent.CompletableFuture",
                "properties": {
                        "result": "java.lang.Object",
                        "stack":
"java.util.concurrent.CompletableFuture.Completion"
                }
        }, {
                "type": "java.lang.Object"
        }, {
```

```
                "type": "java.lang.String"
        }, {
                "type":
"java.util.concurrent.CompletableFuture.Completion",
                "properties": {
                        "next":
"java.util.concurrent.CompletableFuture.Completion",
                        "status": "int"
                }
        }, {
                "type": "int"
        }],
        "annotations": []
}
```

Zookeeper 扩展类型 ZookeeperMetadataReport 实现的存储方法如下所示 doStoreProviderMetadata：

如果我们自己实现一套元数据就可以重写这个方法来进行元数据的额存储

ZookeeperMetadataReport 的 doStoreProviderMetadata

```
    @Override
    protected void doStoreProviderMetadata(MetadataIdentifier providerMetadataIdentifier,
String serviceDefinitions) {
        storeMetadata(providerMetadataIdentifier, serviceDefinitions);
    }
```

ZookeeperMetadataReport 的 storeMetadata

```
 private void storeMetadata(MetadataIdentifier metadataIdentifier, String v) {
    //参数false为非临时节点，这个元数据为持久节点，这个细节就暂时不看了就是将刚刚的json元数据存储到对应路径
上面：路径为：/dubbo/metadata/link.elastic.dubbo.entity.DemoService/provider/dubbo-demo-api-
provider
        zkClient.create(getNodePath(metadataIdentifier), v, false);
    }
```

# 六、 服务发现 MetadataService 导出过程

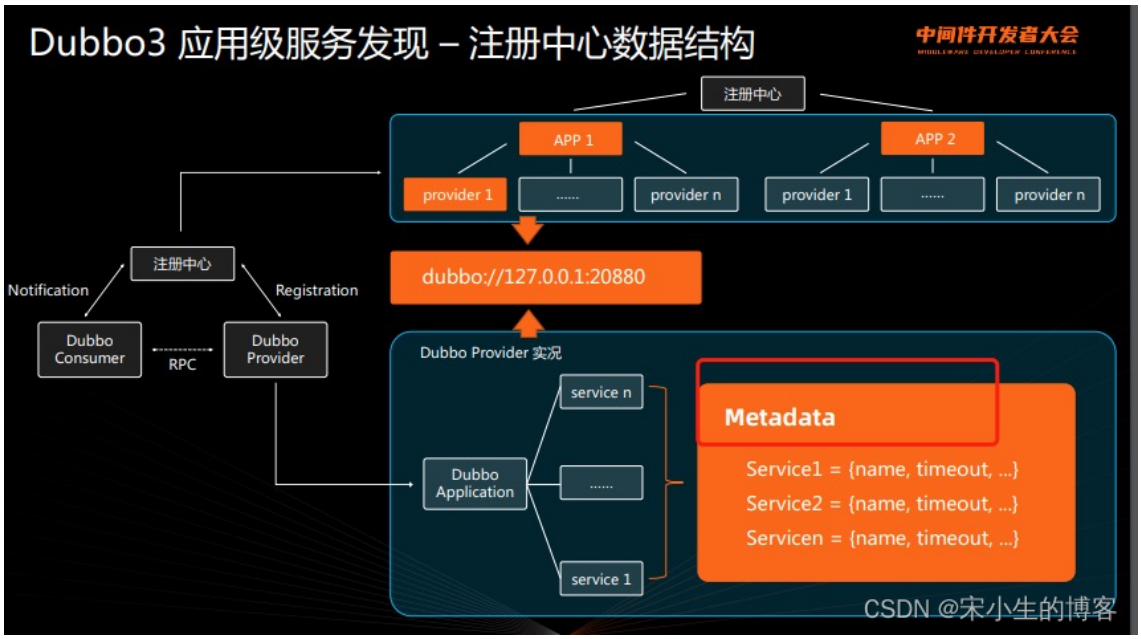
## 1. 简介

MetadataService

此服务用于公开 Dubbo 进程内的元数据信息。典型用途包括：

- 使用者查询提供者的元数据信息，以列出接口和每个接口的配置。
- 控制台（dubbo admin）查询特定进程的元数据，或聚合所有进程的数据。在 Dubbo2.x 的时候，所有的服务数据都是以接口的形式注册在注册中心。

Dubbo3 将部分数据抽象为元数据的形式来将数据存放在元数据中心，然后元数据由服务提供者提供给消费者而不是再由注册中心进行推送，如下图所示：

引入 MetadataService 元数据服务服务的好处

- 由中心化推送转向点对点拉取（Consumer - Proroder）
- 易于扩展更多的参数
- 更多的数据量
- 对外暴露更多的治理数据

## 2. MetadataService 的导出过程

了解元数据的到处过程，这个就要继续前面博客往后的代码了前面博客说了一个服务发布之后的服务信息的双注册数据，这里继续看下导出服务之后的代码。

先来简单回顾下模块发布的启动生命周期方法。

DefaultModuleDeployer 类型的 start 方法

```
@Override
   public synchronized Future start() throws IllegalStateException {
      ...

       try {
          ...
           onModuleStarting();

           // initialize
           applicationDeployer.initialize();
           initialize();

           // export services
           exportServices();

           // prepare application instance
           // exclude internal module to avoid wait itself
           if (moduleModel != moduleModel.getApplicationModel().getInternalModule()) {
               applicationDeployer.prepareInternalModule();
           }

           // refer services
           referServices();

           // if no async export/refer services, just set started
           if (asyncExportingFutures.isEmpty() && asyncReferringFutures.isEmpty()) {
               onModuleStarted();
           } else {
      ....
       return startFuture;
   }
```

前面的博客我们已经说了服务提供者导出服务的方法如下：

```
// export services
exportServices();
```

在导出服务之后如果代码中配置了引用服务的代码将会执行引用服务的功能，调用代码如下

```
referServices();
```

不过我们样例代码并没有介绍引用服务的功能，这里先不说，等服务提供者完全启动成功之后我们再来看消费者的逻辑。

接下来我们要看的是模块启动成功之后的方法 onModuleStarted();，在这个方法中会去发布服务元数据信息。

## 3. 模块启动成功时候的逻辑 onModuleStarted();

这里我们直接先看代码再来分析下逻辑。

DefaultModuleDeployer 类型的 onModuleStarted 方法如下所示

```
private void onModuleStarted() {
       try {
         //状态判断是否为启动中如果是则将状态设置为STARTED
           if (isStarting()) {
             //先修改状态
               setStarted();
               logger.info(getIdentifier() + " has started.");
               //状态修改成功之后开始通知应用程序发布器模块发布器启动成功了
               applicationDeployer.notifyModuleChanged(moduleModel, DeployState.STARTED);
           }
       } finally {
           // complete module start future after application state changed
           completeStartFuture(true);
       }
   }
```

应用程序发布器处理启动成功的逻辑。

DefaultApplicationDeployer 类型的 notifyModuleChanged 方法

```
@Override
   public void notifyModuleChanged(ModuleModel moduleModel, DeployState state) {
       //根据所有模块的状态来判断应用发布器的状态
       checkState(moduleModel, state);

       // notify module state changed or module changed
       //通知所有模块状态更新
       synchronized (stateLock) {
           stateLock.notifyAll();
       }
   }
```

应用发布器模型 DefaultApplicationDeployer 检查状态方法 checkState 代码如下

```
@Override
    public void checkState(ModuleModel moduleModel, DeployState
moduleState) {
       //存在写操作 先加个锁
        synchronized (stateLock) {
         //非内部模块，并且模块的状态是发布成功了
           if (!moduleModel.isInternal() && moduleState ==
DeployState.STARTED) {

               prepareApplicationInstance();
           }
           //应用下所有模块状态进行汇总计算
           DeployState newState = calculateState();
           switch (newState) {
               case STARTED:
                   onStarted();
                   break;
               case STARTING:
                   onStarting();
                   break;
               case STOPPING:
                   onStopping();
                   break;
               case STOPPED:
                   onStopped();
```

```
                break;
            case FAILED:
                Throwable error = null;
                ModuleModel errorModule = null;
                for (ModuleModel module :
applicationModel.getModuleModels()) {
                    ModuleDeployer deployer = module.getDeployer();
                    if (deployer.isFailed() && deployer.getError() !=
null) {
                        error = deployer.getError();
                        errorModule = module;
                        break;
                    }
                }
                onFailed(getIdentifier() + " found failed module: " +
errorModule.getDesc(), error);
                break;
            case PENDING:
                // cannot change to pending from other state
                // setPending();
                break;
        }
    }
}
```

## 4. 准备发布元数据信息和应用实例信息

前面有个代码调用比较重要：

```
prepareApplicationInstance()
```

DefaultApplicationDeployer 类型的 prepareApplicationInstance 方法如下所示

```
@Override
   public void prepareApplicationInstance() {
    //已经注册过应用实例数据了 直接返回 （下面CAS逻辑判断了）
       if (hasPreparedApplicationInstance.get()) {
           return;
       }
   //注册开关控制默认为true
   //通过将registerConsumer默认设置为“false”来关闭纯使用者进程实例的注册。
       if (isRegisterConsumerInstance()) {
           exportMetadataService();
           if (hasPreparedApplicationInstance.compareAndSet(false, true)) {
               // register the local ServiceInstance if required
               registerServiceInstance();
           }
       }
   }
```

### 1） 导出元数据服务方法 exportMetadataService

这里我们就先直接来贴一下代码。

DefaultApplicationDeployer 类型的 exportMetadataService 方法如下所示：

```
 private void exportMetadataService() {
        if (!isStarting()) {
            return;
        }
        //这里监听器我们主要关注的类型是ExporterDeployListener类型
        for (DeployListener<ApplicationModel> listener : listeners) {
            try {
                if (listener instanceof ApplicationDeployListener) {
                // 回调监听器的模块启动成功方法
                    ((ApplicationDeployListener)
listener).onModuleStarted(applicationModel);
                }
            } catch (Throwable e) {
                logger.error(getIdentifier() + " an exception occurred when handle starting
event", e);
            }
        }
    }
```

前面我们主要关注 ExporterDeployListener 类型的监听器的回调方法，这里我贴一下代码。

ExporterDeployListener 类型的 onModuleStarted 方法如下：

```
@Override
    public synchronized void onModuleStarted(ApplicationModel applicationModel) {
        // start metadata service exporter
      //MetadataServiceDelegation类型为实现提供远程RPC服务以方便元数据信息的查询功能的类型。
        MetadataServiceDelegation metadataService =
applicationModel.getBeanFactory().getOrRegisterBean(MetadataServiceDelegation.class);
        if (metadataServiceExporter == null) {
            metadataServiceExporter = new
ConfigurableMetadataServiceExporter(applicationModel, metadataService);
            // fixme, let's disable local metadata service export at this moment
            //默认我们是没有配置这个元数据类型的这里元数据类型默认为local 条件是不是remote则开始导出，在
前面的博客<<Dubbo启动器DubboBootstrap添加应用程序的配置信息ApplicationConfig>> 中有提到这个配置下面我
再说下
            if (!REMOTE_METADATA_STORAGE_TYPE.equals(getMetadataType(applicationModel))) {
                metadataServiceExporter.export();
            }
        }
    }
```

在前面的博客<<Dubbo 启动器 DubboBootstrap 添加应用程序的配置信息 ApplicationConfig>>中有提到这个配置下面我再说下 metadata-type，metadata 传递方式，是以 Provider 视角而言的，Consumer 侧配置无效，可选值有：

- remote-Provider 把 metadata 放到远端注册中心，Consumer 从注册中心获取。

- local-Provider 把 metadata 放在本地，Consumer 从 Provider 处直接获取。

可以看到默认的 local 配置元数据信息的获取是由消费者从提供者拉的，那提供者怎么拉取对应服务的元数据信息那就要要用到这个博客说到的 MetadataService 服务，传递方式为 remote 的方式其实就要依赖注册中心了相对来说增加了注册中心的压力。

### 2) 可配置元数据服务的导出 ConfigurableMetadataServiceExporter 的 export

前面了解了导出服务的调用链路，这里详细看下 ConfigurableMetadataServiceExporter 的 export 过程源码如下所示：

```
public synchronized ConfigurableMetadataServiceExporter export() {
    //元数据服务配置已经存在或者已经导出或者不可导出情况下是无需导出的
        if (serviceConfig == null || !isExported()) {
          //创建服务配置
            this.serviceConfig = buildServiceConfig();
            // export
            //导出服务 ,导出服务的具体过程这里就不再说了可以看上一个博客，这个导出服务的过程会绑定端口
            serviceConfig.export();
            metadataService.setMetadataURL(serviceConfig.getExportedUrls().get(0));
            if (logger.isInfoEnabled()) {
                logger.info("The MetadataService exports urls : " +
serviceConfig.getExportedUrls());
            }
        } else {
            if (logger.isWarnEnabled()) {
                logger.warn("The MetadataService has been exported : " +
serviceConfig.getExportedUrls());
            }
        }

        return this;
    }
```

### 3) 元数据服务配置对象的创建

前面我们看到了构建元数据服务对象的代码调用 ServiceConfig<MetadataService>，接下来我们详细看下构建源码如下所示 ConfigurableMetadataServiceExporter 类型的 buildServiceConfig 构建元数据服务配置对象方法如下：

```
private ServiceConfig<MetadataService> buildServiceConfig() {
//1 获取当前的应用配置 然后初始化应用配置
        ApplicationConfig applicationConfig = getApplicationConfig();
        //创建服务配置对象
        ServiceConfig<MetadataService> serviceConfig = new ServiceConfig<>();
        //设置域模型
        serviceConfig.setScopeModel(applicationModel.getInternalModule());
        serviceConfig.setApplication(applicationConfig);

       //2 创建注册中心配置对象 然后并初始化
        RegistryConfig registryConfig = new RegistryConfig("N/A");
        registryConfig.setId("internal-metadata-registry");

    //3 创建服务配置对象，并初始化
        serviceConfig.setRegistry(registryConfig);
        serviceConfig.setRegister(false);
        //4 生成协议配置 ， 这里会配置一下元数据使用的服务端口号默认使用其他服务的端口20880
        serviceConfig.setProtocol(generateMetadataProtocol());
        serviceConfig.setInterface(MetadataService.class);
        serviceConfig.setDelay(0);
       //这里也是需要注意的地方服务引用的类型为MetadataServiceDelegation
        serviceConfig.setRef(metadataService);
        serviceConfig.setGroup(applicationConfig.getName());
        serviceConfig.setVersion(MetadataService.VERSION);
        //5 生成方法配置 这里目前提供的服务方法为getAndListenInstanceMetadata方法 后续可以看下这个方法
的视线
        serviceConfig.setMethods(generateMethodConfig());
        serviceConfig.setConnections(1); // separate connection
        serviceConfig.setExecutes(100); // max tasks running at the same time

        return serviceConfig;
    }
```

这个服务配置对象的创建非常像我们第一个博客提到的服务配置过程，不过这个元数据服务对象有几个比较特殊的配置。

- 注册中心的配置 register 设置为了 false 则为不向注册中心注册具体的服务配置信息。
- 对每个提供者的最大连接数 connections 为 1。
- 服务提供者每服务每方法最大可并行执行请求数 executes 为 100。

在使用过程中可以知道上面这几个配置值。

## 5. 应用级数据注册 registerServiceInstance()

在前面导出元数据服务之后也会调用一行代码来注册应用级数据来保证应用上线。

主要涉及到的代码为 DefaultApplicationDeployer 类型中的 registerServiceInstance 方法如下所示：

```
 private void registerServiceInstance() {
        try {
         //标记变量设置为true
            registered = true;
            ServiceInstanceMetadataUtils.registerMetadataAndInstance(applicationModel);
        } catch (Exception e) {
            logger.error("Register instance error", e);
        }
        if (registered) {
            // scheduled task for updating Metadata and ServiceInstance
            asyncMetadataFuture =
frameworkExecutorRepository.getSharedScheduledExecutor().scheduleWithFixedDelay(() -> {

                // ignore refresh metadata on stopping
                if (applicationModel.isDestroyed()) {
                    return;
                }
                try {
                    if (!applicationModel.isDestroyed() && registered) {

ServiceInstanceMetadataUtils.refreshMetadataAndInstance(applicationModel);
                    }
                } catch (Exception e) {
                    if (!applicationModel.isDestroyed()) {
                        logger.error("Refresh instance and metadata error", e);
                    }
                }
            }, 0, ConfigurationUtils.get(applicationModel, METADATA_PUBLISH_DELAY_KEY,
DEFAULT_METADATA_PUBLISH_DELAY), TimeUnit.MILLISECONDS);
        }
    }
```

这个方法先将应用元数据注册到注册中心，然后开始开启定时器每隔 30 秒同步一次元数据向注册中心。

### 1) 服务实例元数据工具类注册服务发现的元数据信息

前面通过调用类型 ServiceInstanceMetadataUtils 工具类的 registerMetadataAndInstance 方法来进行服务实例数据和元数据的注册这里我们详细看下代码如下所示：

```
    public static void registerMetadataAndInstance(ApplicationModel applicationModel) {
        LOGGER.info("Start registering instance address to registry.");
        RegistryManager registryManager =
applicationModel.getBeanFactory().getBean(RegistryManager.class);
        // register service instance
//注意这里服务发现的类型只有ServiceDiscoveryRegistry类型的注册协议才满足
registryManager.getServiceDiscoveries().forEach(ServiceDiscovery::register);
    }
```

### 2) AbstractServiceDiscovery 中的服务发现数据注册的模版方法

AbstractServiceDiscovery 类型的注册方法 register()方法这个是一个模版方法，真正执行的注册逻辑封装在了 doRegister 方法中由扩展的服务发现子类来完成。

```
  @Override
    public synchronized void register() throws RuntimeException {
       //第一步创建应用的实例信息等待下面注册到注册中心
        this.serviceInstance = createServiceInstance(this.metadataInfo);
        if (!isValidInstance(this.serviceInstance)) {
            logger.warn("No valid instance found, stop registering instance address to
registry.");
            return;
        }

    //是否需要更新
        boolean revisionUpdated = calOrUpdateInstanceRevision(this.serviceInstance);
        if (revisionUpdated) {
            reportMetadata(this.metadataInfo);
            //应用的实例信息注册到注册中心之上 ， 这个
            doRegister(this.serviceInstance);
        }
    }
```

### 3) 应用级实例对象创建

可以看到在 AbstractServiceDiscovery 服务发现的第一步创建应用的实例信息等待下面注册到注册中心。

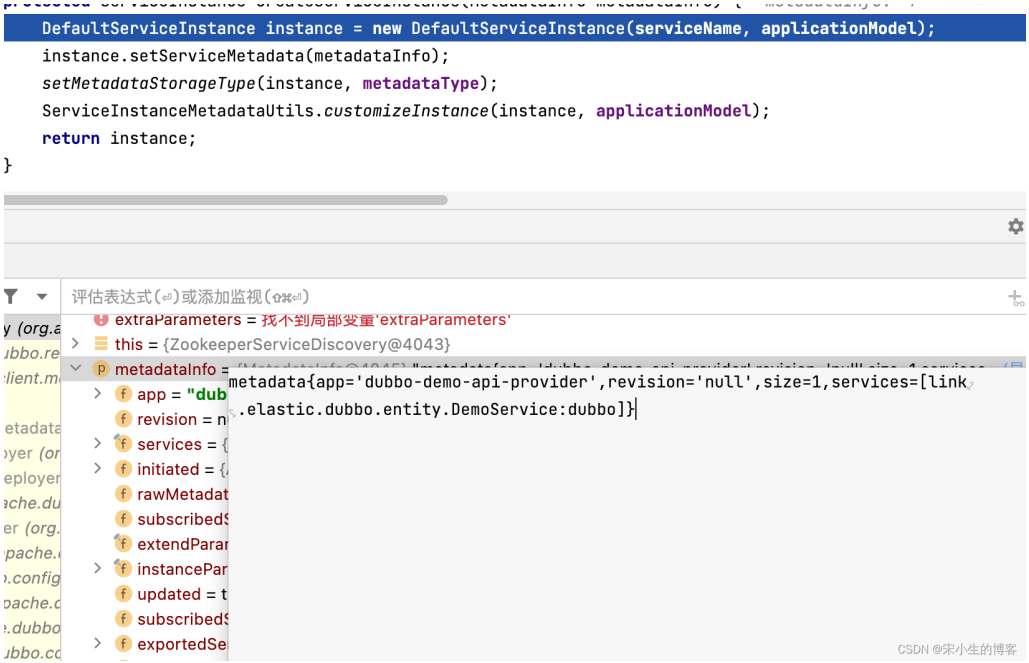
```
this.serviceInstance = createServiceInstance(this.metadataInfo);
```

最终创建的 serviceInstance 类型为 ServiceInstance 这个是 Dubbo 封装的一个接口，具体实现类型为 DefaultServiceInstance，我们可以看下应用级的元数据有哪些。

```
    protected ServiceInstance createServiceInstance(MetadataInfo metadataInfo) {
    //这里的服务名字为：dubbo-demo-api-provider
        DefaultServiceInstance instance = new DefaultServiceInstance(serviceName,
applicationModel);
        //应用服务的元数据 ， 可以看下面debug的数据信息
        instance.setServiceMetadata(metadataInfo);
        //metadataType的值为local 这个方法是将元数据类型存储到英勇的元数据对象中 对应内容为
dubbo.metadata.storage-type:local
        setMetadataStorageType(instance, metadataType);
        // 这个是自定义元数据数据 我们也可以通过实现扩展ServiceInstanceCustomizer来自定义一些元数据
        ServiceInstanceMetadataUtils.customizeInstance(instance, applicationModel);
        return instance;
    }
```

这个方法的主要目的就是将应用的元数据信息都封装到 ServiceInstance 类型中，不过额外提供了一个扩展性比较好的方法可以自定义元数据信息。

前面的 metadataInfo 对象的信息如下图所示：



自定义元数据类型 Dubbo 官方提供了一个默认的实现类型为：ServiceInstanceMetadataCustomizer

最终封装好的元数据信息如下所示：

```
DefaultServiceInstance{
serviceName='dubbo-demo-api-provider',
host='192.168.1.169',
port=20880,
enabled=true,
healthy=true,
 metadata={
  dubbo.metadata-service.url-params={"connections":"1",
  "version":"1.0.0",
  "dubbo":"2.0.2",
  "release":"3.0.9",
  "side":"provider",
  "port":"20880",
  "protocol":"dubbo"
  },
  dubbo.endpoints=[
  {"port":20880,"protocol":"dubbo"}],
  dubbo.metadata.storage-type=local,
  timestamp=1656227493387}}
```

### 4) 应用级实例数据配置变更的的版本号获取

前面创建元应用的实例信息后开始创建版本号来判断是否需要更新，对应 AbstractServiceDiscovery 类型的 calOrUpdateInstanceRevision

```
  protected boolean calOrUpdateInstanceRevision(ServiceInstance instance) {
      //获取元数据版本号对应字段dubbo.metadata.revision
        String existingInstanceRevision = getExportedServicesRevision(instance);
        //获取实例的服务元数据信息：metadata{app='dubbo-demo-api-
provider',revision='null',size=1,services=[link.elastic.dubbo.entity.DemoService:dubbo]}
        MetadataInfo metadataInfo = instance.getServiceMetadata();
        //必须在不同线程之间同步计算此实例的状态，如同一实例的修订和修改。此方法的使用仅限于某些点，例如在注
册期间。始终尝试使用此选项。改为getRevision（）。
        String newRevision = metadataInfo.calAndGetRevision();
        //版本号发生了变更（元数据发生了变更）版本号是md5元数据信息计算出来HASH验证
        if (!newRevision.equals(existingInstanceRevision)) {
            //版本号添加到dubbo.metadata.revision字段中
            instance.getMetadata().put(EXPORTED_SERVICES_REVISION_PROPERTY_NAME,
metadataInfo.getRevision());
            return true;
        }
        return false;
    }
```

a) 元数据版本号的计算与 HASH 校验 calAndGetRevision

这个方法其实比较重要，决定了什么时候会更新元数据，Dubbo 使用了一种 Hash 验证的方式将元数据转 MD5 值与之前的存在的版本号（也是元数据转 MD5 得到的），如果数据发生了变更则 MD5 值会发生变化，以此来更新元数据，不过发生了 MD5 冲突的话就会导致配置不更新这个冲突的概率非常小。

好了直接来看代码吧。

MetadataInfo 类型的 calAndGetRevision 方法：

```
public synchronized String calAndGetRevision() {
        if (revision != null && !updated) {
            return revision;
        }

        updated = false;
    //应用下没有服务则使用一个空的版本号
        if (CollectionUtils.isEmptyMap(services)) {
            this.revision = EMPTY_REVISION;
        } else {
            StringBuilder sb = new StringBuilder();
            //app是应用名
            sb.append(app);
            for (Map.Entry<String, ServiceInfo> entry : new TreeMap<>(services).entrySet())
{
                sb.append(entry.getValue().toDescString());
            }
            String tempRevision = RevisionResolver.calRevision(sb.toString());
            if (!StringUtils.isEquals(this.revision, tempRevision)) {
              //元数据重新注册的话我们可以看看这个日志metadata revision change
                if (logger.isInfoEnabled()) {
                    logger.info(String.format("metadata revision changed: %s -> %s, app: %s,
services: %d", this.revision, tempRevision, this.app, this.services.size()));
                }
                this.revision = tempRevision;
                this.rawMetadataInfo = JsonUtils.getJson().toJson(this);
            }
        }
        return revision;
    }
```

RevisionResolver 类型的 Md5 运算计算版本号

```
md5Utils.getMd5(metadata);
```

### 5) reportMetadata

回到 18.5.2 AbstractServiceDiscovery 中的模版方法 register，这里我们来看下 reportMetadata 方法，不过这个方法目前并不会走到，因为我们默认的配置元数据是 local 不会直接把应用的元数据注册在元数据中心。

```
  protected void reportMetadata(MetadataInfo metadataInfo) {
        if (metadataReport != null) {
        //订阅元数据的标识符
            SubscriberMetadataIdentifier identifier = new
SubscriberMetadataIdentifier(serviceName, metadataInfo.getRevision());
            //是否远程发布元数据，这里我们是本地注册这个就不会在元数据中心发布这个元数据信息
            if ((DEFAULT_METADATA_STORAGE_TYPE.equals(metadataType) &&
metadataReport.shouldReportMetadata()) || REMOTE_METADATA_STORAGE_TYPE.equals(metadataType))
{
                metadataReport.publishAppMetadata(identifier, metadataInfo);
            }
        }
    }
```

### 6) 扩展的注册中心来注册应用级服务发现数据 doRegister 方法

前面我们说了 AbstractServiceDiscovery 中的模版方法 register，在 register 会调用一个 doRegister 方法来注册应用级数据，这个方法是需要扩展注册中心的服务发现来自行实现的，我们这里以官方实现的 Zookeeper 服务发现模型为例。

ZookeeperServiceDiscovery 中的 doRegister 方法

```
 @Override
    public void doRegister(ServiceInstance serviceInstance) {
        try {
        //Dubbo实现的ServiceInstance类型对象转 Curator的ServiceInstance
            serviceDiscovery.registerService(build(serviceInstance));
        } catch (Exception e) {
            throw new RpcException(REGISTRY_EXCEPTION, "Failed register instance " +
serviceInstance.toString(), e);
        }
    }
```

前面我们介绍了 ZookeeperServiceDiscovery 发现的构造器连接注册中心，这里来看下服务注册，应用级实例数据注册一共分为两步：

- **第一步是：** Dubbo 实现的 ServiceInstance 类型对象转 Curator 的 ServiceInstance。

- **第二步是：** 执行 registerService 方法将数据注册到注册中心。

先来看第一步：Dubbo 实现的 ServiceInstance 类型对象转 Curator 的 ServiceInstance，关于 Curator 的服务发现原理可以参考官网的文章博客。

**什么是发现服务？**

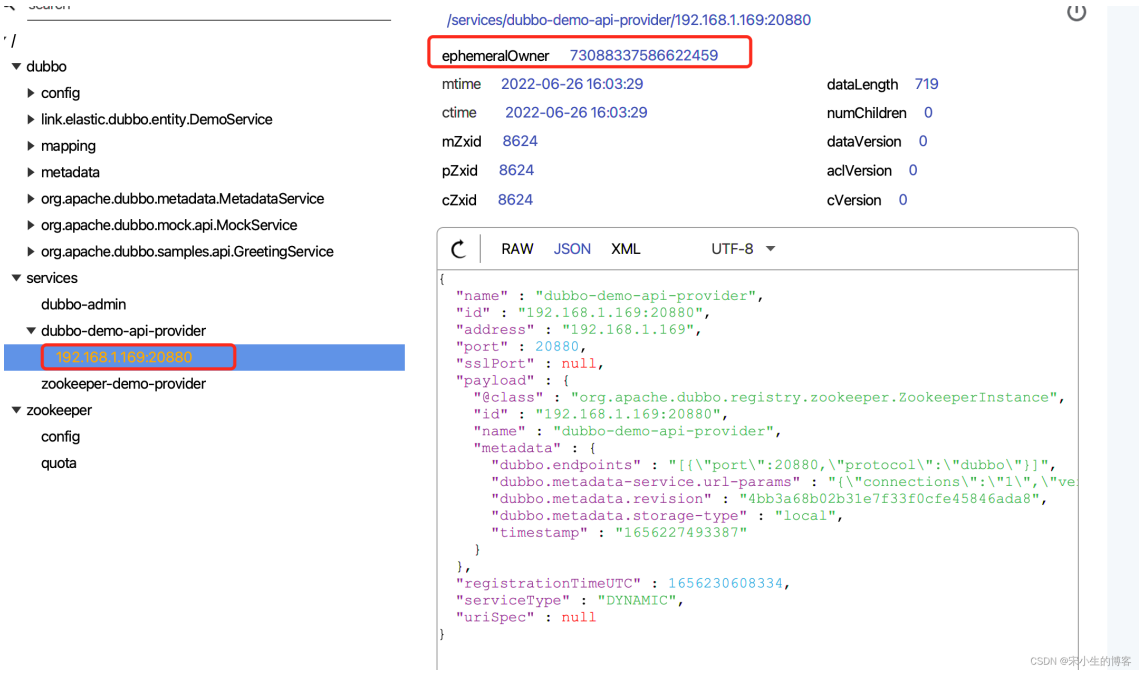
在 SOA/分布式系统中，服务需要找到彼此。即，Web 服务可能需要找到缓存服务等。DNS 可以用于此，但对于不断变化的服务来说，它远不够灵活。服务发现系统提供了一种机制：

- 注册其可用性的服务
- 定位特定服务的单个实例
- 在服务实例更改时通知

服务实例由类表示：ServiceInstance。ServiceInstances 具有名称、id、地址、端口和/或 ssl 端口，以及可选的有效负载（用户定义）。ServiceInstances 通过以下方式序列化并存储在 ZooKeeper 中：

```
base path
       |_______ service A name
                     |__________ instance 1 id --> (serialized ServiceInstance)
                     |__________ instance 2 id --> (serialized ServiceInstance)
                     |__________ ...
       |_______ service B name
                     |__________ instance 1 id --> (serialized ServiceInstance)
                     |__________ instance 2 id --> (serialized ServiceInstance)
                     |__________ ...
       |_______ ...
```

这个应用最终注册应用级服务数据如下：




这里需要注意的是这个，应用的 IP+端口的服务元数据信息是临时节点，build 方法内容对应着上图的 JSON 数据，可以看菜 build 方法封装的过程：

```
public static org.apache.curator.x.discovery.ServiceInstance<ZookeeperInstance>
build(ServiceInstance serviceInstance) {
        ServiceInstanceBuilder builder;

        String serviceName = serviceInstance.getServiceName();
        String host = serviceInstance.getHost();
        int port = serviceInstance.getPort();
        Map<String, String> metadata = serviceInstance.getSortedMetadata();
        String id = generateId(host, port);
        //ZookeeperInstance是Dubbo封装的用于存放payload数据 包含服务id，服务名字和元数据
        ZookeeperInstance zookeeperInstance = new ZookeeperInstance(id, serviceName,
metadata);
        try {
            builder = builder()
                .id(id)
                .name(serviceName)
                .address(host)
                .port(port)
                .payload(zookeeperInstance);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return builder.build();
    }
```

在《18.5 应用级数据注册 registerServiceInstance()》小节中介绍了应用元数据信息的注册调用代码，其实后面还有个 update 的逻辑定期 30 秒同步元数据到元数据中心，这里就不详细介绍了。

# 服务引用启动过程

## 一、 从一个服务消费者的 Demo 说起

为了更方便了解原理，我们先来编写一个 Demo，从例子中来看源码实现：前面说了提供者现在已经有服务注册上去了，那接下来我们编写一个消费者的例子来进行服务发现与服务 RPC 调用。

### 1. 启动 Zookeeper

为了 Demo 可以正常启动，需要我们先在本地启动一个 Zookeeper 如下图所示：

```
➜  zookeeper-3.4.14 cd bin
➜  bin ls
README.txt          zkCli.sh            zkServer.cmd        zkTxnLogToolkit.sh
zkCleanup.sh        zkEnv.cmd           zkServer.sh         zookeeper.out
zkCli.cmd           zkEnv.sh            zkTxnLogToolkit.cmd
➜  bin ./zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /Users/song/Desktop/Computer/A/env/zookeeper-3.4.14/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED                                  CSDN @宋小生-中间件
```

### 2. 服务消费者

接下来给大家贴一下示例源码，这个源码来源于 Dubbo 源码目录的 dubbo-demo/dubbo-demo-api 目录下面的 dubbo-demo-api-consumer 子项目，这里我做了删减，方便看核心代码。

首先我们定义一个服务接口如下所示：

```
import java.util.concurrent.CompletableFuture;
public interface DemoService {
    /**
     * 同步处理的服务方法
     * @param name
     * @return
     */
    String sayHello(String name);
```

```
    /**
     * 用于异步处理的服务方法
     * @param name
     * @return
     */
    default CompletableFuture<String> sayHelloAsync(String name) {
        return CompletableFuture.completedFuture(sayHello(name));
    }
}
```

服务实现类如下：

```
import org.apache.dubbo.rpc.RpcContext;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.concurrent.CompletableFuture;

public class DemoServiceImpl implements DemoService {
    private static final Logger logger =
LoggerFactory.getLogger(DemoServiceImpl.class);

    @Override
    public String sayHello(String name) {
        logger.info("Hello " + name + ", request from consumer: " +
RpcContext.getServiceContext().getRemoteAddress());
        return "Hello " + name + ", response from provider: " +
RpcContext.getServiceContext().getLocalAddress();
    }

    @Override
    public CompletableFuture<String> sayHelloAsync(String name) {
        return null;
    }

}
```

## 3. 启用服务消费者

有了服务接口之后我们来启用服务，启用服务的源码如下。

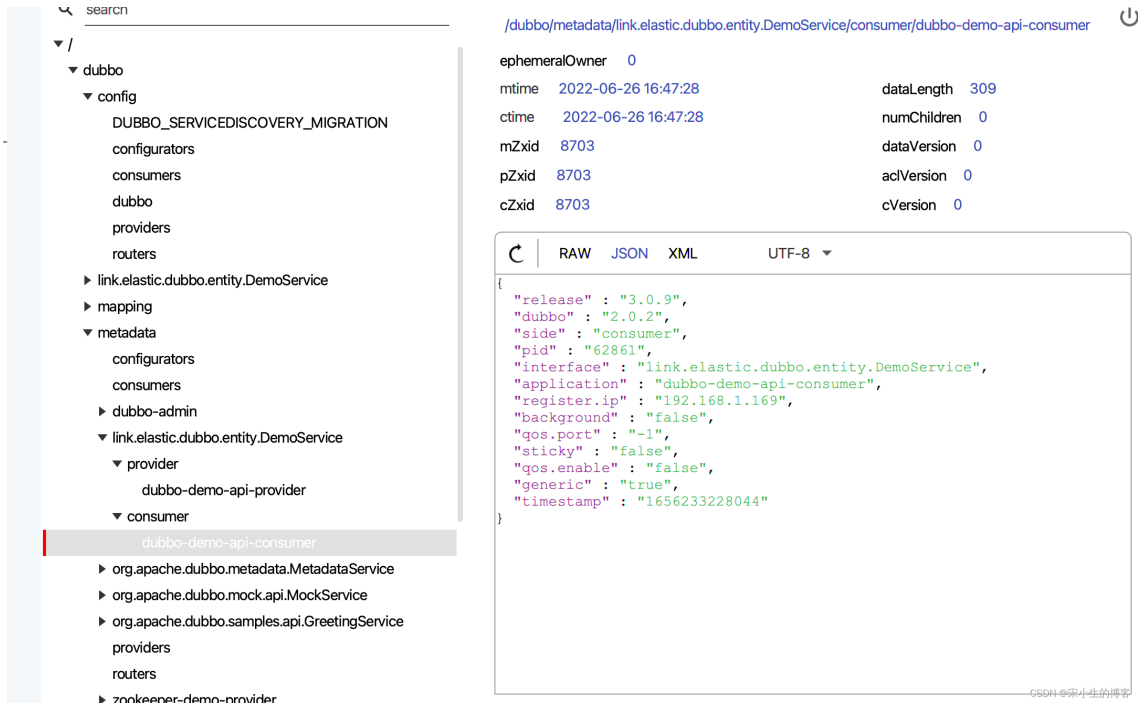
这里如果要启动消费者，主要要修改 QOS 端口这里我已经配置可以直接复用

```
package link.elastic.dubbo.consumer;

import link.elastic.dubbo.entity.DemoService;
import org.apache.dubbo.common.constants.CommonConstants;
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.MetadataReportConfig;
import org.apache.dubbo.config.ProtocolConfig;
import org.apache.dubbo.config.ReferenceConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.bootstrap.DubboBootstrap;
import org.apache.dubbo.rpc.service.GenericService;

public class ConsumerApplication {
    public static void main(String[] args) {
            runWithBootstrap();
    }
    private static void runWithBootstrap() {
        ReferenceConfig<DemoService> reference = new
ReferenceConfig<>();
        reference.setInterface(DemoService.class);
        reference.setGeneric("true");
        reference.setProtocol("");

        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        ApplicationConfig applicationConfig = new
ApplicationConfig("dubbo-demo-api-consumer");
        applicationConfig.setQosEnable(false);
        applicationConfig.setQosPort(-1);
        bootstrap.application(applicationConfig)
            .registry(new
RegistryConfig("zookeeper://8.131.79.126:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .reference(reference)
            .start();

        DemoService demoService = bootstrap.getCache().get(reference);
        String message = demoService.sayHello("dubbo");
        System.out.println(message);
```

```
        // generic invoke
        GenericService genericService = (GenericService) demoService;
        Object genericInvokeResult = genericService.$invoke("sayHello",
new String[]{String.class.getName()},
            new Object[]{"dubbo generic invoke"});
        System.out.println(genericInvokeResult);
    }
}
```

## 4. 启用服务后写入 Zookeeper 的节点数据

启动服务，这个时候我们打开 Zookeeper 图形化客户端来看看这个服务在 Zookeeper 上面写入来哪些数据如下图。




写入 Zookeper 上的节点用于服务在分布式场景下的协调，这些节点是比较重要的。

如果了解过 Dubbo 的同学，应该会知道 Dubbo 在低版本的时候会向注册中心中写入服务接口，具体路径在上面的 dubbo 目录下，然后在/dubbo/服务接口/路径下写入如下信息：

- 服务提供者配置信息 URL 形式
- 服务消费者的配置信息 URL 形式
- 服务路由信息
- 配置信息

上面这个图就是 Dubbo3 的注册信息了，后面我们也会围绕细节来说明下，这里可以看下新增了：

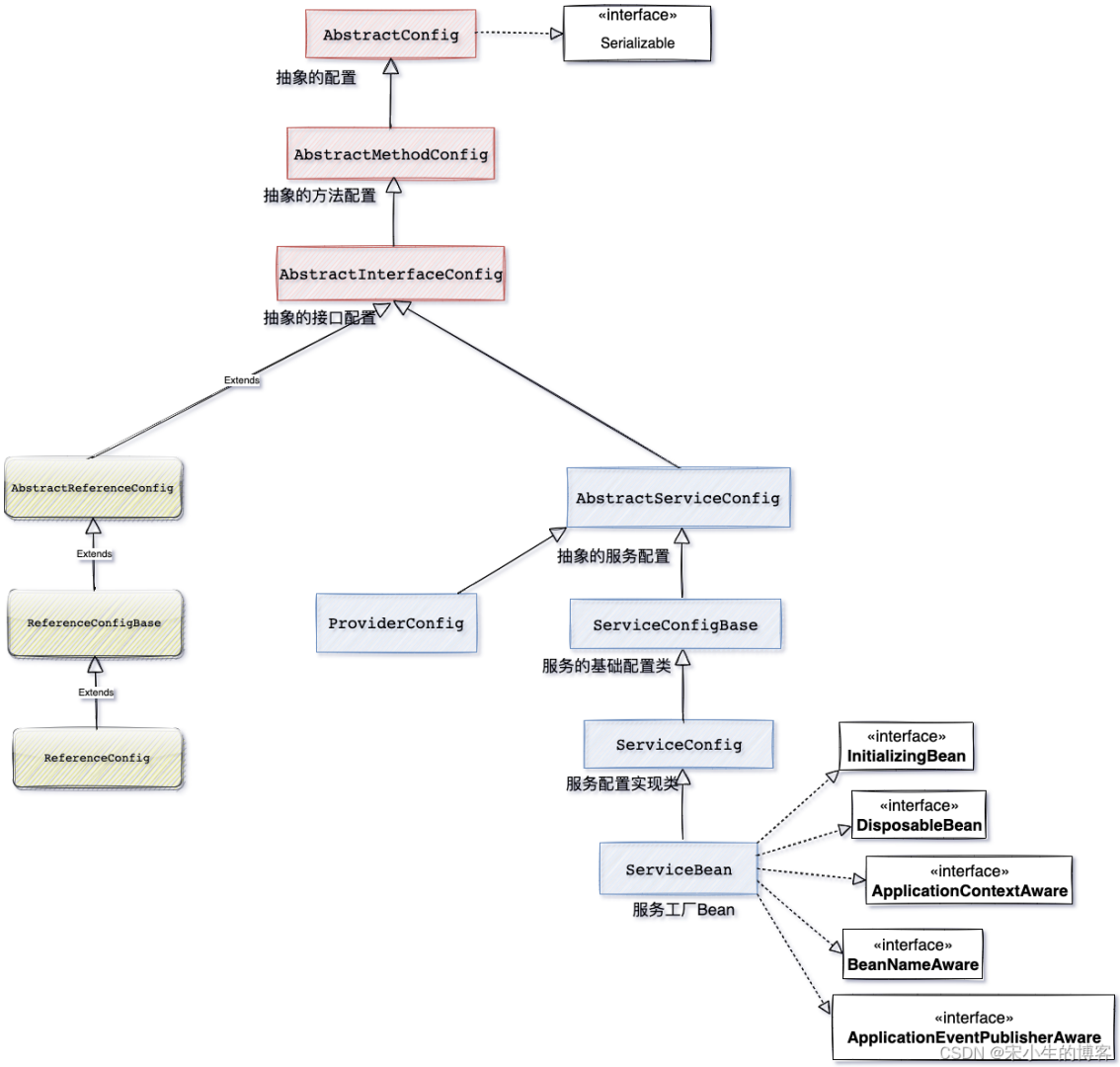
- /dubbo/metadata 元数据信息
- /dubbo/mapping 服务和应用的映射信息
- /dubbo/config 注册中心配置
- /services 目录应用信息

在这里可以大致了解下，在后面会有更详细的源码解析这个示例代码。通过透析代码来看透 Dubbo3 服务注册原理，服务提供原理。

## 二、 ReferenceConfig 设计

### 1. 简介

前面简单介绍了一下消费者的例子，消费者创建的第一步就是先进行消费者信息的配置对应类型为 ReferenceConfig，这里详细来看 ReferenceConfig 包含哪些信息？先简单了解下消费者配置的类型关系如下图所示：引用配置与服务配置类型都是通过继承接口配置来扩展的，在分析生产者的时候详细介绍过服务相关的配置，这里来详细看消费者引用者的相关配置信息。




前面例子说了消费者配置对象的创建主要是通过如下代码：

```
ReferenceConfig<DemoService> reference = new ReferenceConfig<>();
```

这个配置类型的对象创建过程并没有太多的逻辑这里主要来说下各种配置信息：服务消费者引用服务配置。对应的配置类：org.apache.dubbo.config.ReferenceConfig

| 属性 | 对应 URL 参数 | 类型 | 是否必填 | 缺省值 | 作用 | 描述 | 兼容性 |
|---|---|---|---|---|---|---|---|
| Id | | string | 必填 | | 配置关联 | 服务引用 BeanId | 1.0.0 以上版本 |
| interface | | class | 必填 | | 服务发现 | 服务接口名 | 1.0.0 以上版本 |
| version | version | string | 可选 | | 服务发现 | 服务版本，与服务提供者的版本一致 | 1.0.0 以上版本 |
| group | group | string | 可选 | | 服务发现 | 服务分组，当一个接口有多个实现，可以用分组区分，必需和服务提供方一致 | 1.0.7 以上版本 |
| timeout | timeout | long | 可选 | 缺省使用 dubbo:consumer 的 timeout | 性能调优 | 服务方法调用超时时间（毫秒） | 1.0.5 以上版本 |
| retries | retries | int | 可选 | 缺省使用 dubbo:consumer 的 retries | 性能调优 | 远程服务调用重试次数，不包括第一次调用，不需要重试请设为 0 | 2.0.0 以上版本 |
| connections | connections | int | 可选 | 缺省使用 dubbo:consumer 的 connections | 性能调优 | 对每个提供者的最大连接数，rmi、http、hessian 等短连接协议表示限制连接数，dubbo 等长连接协表示建立的长连接个数 | 2.0.0 以上版本 |
| loadbalance | loadbalance | string | 可选 | 缺省使用 dubbo:consumer 的 loadbalance | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮询，最少活跃调用 | 2.0.0 以上版本 |
| async | async | boolean | 可选 | 缺省使用 dubbo:consumer 的 async | 性能调优 | 是否异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程 | 2.0.0 以上版本 |
| generic | generic | boolean | 可选 | 缺省使用 dubbo:consumer 的 generic | 服务治理 | 是否缺省泛化接口，如果为泛化接口，将返回 GenericService | 2.0.0 以上版本 |
| check | check | boolean | 可选 | 缺省使用 dubbo:consumer 的 check | 服务治理 | 启动时检查提供者是否存在，true 报错，false 忽略 | 2.0.0 以上版本 |
| url | url | string | 可选 | | 服务治理 | 点对点直连服务提供者地址，将绕过注册中心 | 1.0.6 以上版本 |
| stub | stub | class/boolean | 可选 | | 服务治理 | 服务接口客户端本地代理类名，用于在客户端执行本地逻辑，如本地缓存等，该本地代理类的构造函数必须允许传入远程代理对象，构造函数如：public XxxServiceLocal（XxxService xxxService） | 2.0.0 以上版本 |
| mock | mock | class/boolean | 可选 | | 服务治理 | 服务接口调用失败 Mock 实现类名，该 Mock 类必须有一个无参构造函数，与 Local 的区别在于，Local 总是被执行，而 Mock 只在出现非业务异常（比如超时，网络异常等）时执行，Local 在远程调用之前执行，Mock 在远程调用后执行。 | Dubbo1.0.13 及其以上版本支持 |
| cache | cache | string/boolean | 可选 | | 服务治理 | 以调用参数为 key，缓存返回结果，可选：lru, threadlocal, jcache 等 | Dubbo2.1.0 及其以上版本支持 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| validation | validation | boolean | 可选 | | 服务治理 | 是否启用 JSR303 标准注解验证，如果启用，将对方法参数上的注解进行校验 | Dubbo2.1.0 及其以上版本支持 |
| proxy | proxy | boolean | 可选 | javassist | 性能调优 | 选择动态代理实现策略，可选：javassist, jdk | 2.0.2 以上版本 |
| client | client | string | 可选 | | 性能调优 | 客户端传输类型设置，如 Dubbo 协议的 netty 或 mina。 | Dubbo2.0.0 以上版本支持 |
| registry | | string | 可选 | 缺省将从所有注册中心获服务列表后合并结果 | 配置关联 | 从指定注册中心注册获取服务列表，在多个注册中心时使用，值为 dubbo:registry 的 id 属性，多个注册中心 ID 用逗号分隔 | 2.0.0 以上版本 |
| owner | owner | string | 可选 | | 服务治理 | 调用服务负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5 以上版本 |
| actives | actives | int | 可选 | 0 | 性能调优 | 每服务消费者每服务每方法最大并发调用数 | 2.0.5 以上版本 |
| uster | cluster | string | 可选 | failover | 性能调优 | 集群方式，可选：failover/failfast/failsafe/failback/forking | 2.0.5 以上版本 |
| filter | reference.filter | string | 可选 | default | 性能调优 | 服务消费方远程调用过程拦截器名称，多个名称用逗号分隔 | 2.0.5 以上版本 |
| listener | invoker.listener | string | 可选 | default | 性能调优 | 服务消费方引用服务监听器名称，多个名称用逗号分隔 | 2.0.5 以上版本 |
| layer | layer | string | 可选 | | 服务治理 | 服务调用者所在的分层。如：biz、dao、intl:web、china:acton。 | 2.0.7 以上版本 |
| init | init | boolean | 可选 | false | 性能调优 | 是否在 afterPropertiesSet()时饥饿初始化引用，否则等到有人注入或引用该实例时再初始化。 | 2.0.10 以上版本 |
| protocol | protocol | string | 可选 | | 服务治理 | 只调用指定协议的服务提供方，其它协议忽略。 | |

# 三、 消费者服务引用流程

## 1. 简介

前面我们通过 Demo 说了一个服务引用配置的创建。另外也在前面的文章说了服务提供者的启动完整过程，不过在说服务提供者启动的过程中并未提到服务消费者是如何发现服务，如果调用服务的，这里先就不再说关于服务消费者启动的一个细节了，直接来看前面未提到的服务消费者是如何引用到服务提供者提供的服务的。

先来回顾下样例代码：

```
public class ConsumerApplication {
    public static void main(String[] args) {
            runWithBootstrap();
    }
    private static void runWithBootstrap() {
        ReferenceConfig<DemoService> reference = new ReferenceConfig<>();
        reference.setInterface(DemoService.class);
        reference.setGeneric("true");
        reference.setProtocol("");

        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        ApplicationConfig applicationConfig = new ApplicationConfig("dubbo-demo-api-
consumer");
        applicationConfig.setQosEnable(false);
        applicationConfig.setQosPort(-1);
        bootstrap.application(applicationConfig)
            .registry(new RegistryConfig("zookeeper://8.131.79.126:2181"))
            .protocol(new ProtocolConfig(CommonConstants.DUBBO, -1))
            .reference(reference)
            .start();

        DemoService demoService = bootstrap.getCache().get(reference);
        String message = demoService.sayHello("dubbo");
        System.out.println(message);

        // generic invoke
        GenericService genericService = (GenericService) demoService;
        Object genericInvokeResult = genericService.$invoke("sayHello", new String[]
{String.class.getName()},
            new Object[]{"dubbo generic invoke"});
        System.out.println(genericInvokeResult);
    }
}
```

这段代码我们前面详细说了服务引用的配置 ReferenceConfig 和 Dubbo 启动器启动应用的过程 DubboBootstrap，后面我们直接定位到消费者引用服务的代码位置来看。

## 2. 入口代码

### 1) DefaultModuleDeployer 的 start 方法

第一个要关注的就是模块发布器 DefaultModuleDeployer 的 start 方法，这个 start 方法包含了 Dubbo 应用启动的过程。

DefaultModuleDeployer 的 start 方法

```
public synchronized Future start() throws IllegalStateException {
          ...省略掉若干代码

            onModuleStarting();

            // initialize
            applicationDeployer.initialize();
            initialize();

            // export services
            exportServices();

            // prepare application instance
            // exclude internal module to avoid wait itself
            if (moduleModel != moduleModel.getApplicationModel().getInternalModule()) {
                applicationDeployer.prepareInternalModule();
            }

            // refer services
            referServices();

            ...省略掉若干代码
        return startFuture;
    }
```

这个方法大部分代码已经省略，也不会详细去说了，感兴趣的可以看之前讲到的博客，这里主要来看引用服务方法 referServices

### 2) DefaultModuleDeployer 的 referServices 方法

下面就要来看消费者应用如何引用的服务的入口了，这个方法主要从大的方面做了一些服务引用生命周期的代码，看懂了这个方法我们就可以不依赖 Dubbo 负载的启动逻辑可以单独调用 ReferenceConfigBase 类型的对应方法来刷新，启动，销毁引用的服务了这里先来看下代码再详细介绍内容。

DefaultModuleDeployer 的 referServices 方法

```
private void referServices() {
        //这个是获取配置的所有的 ReferenceConfigBase 类型对象
       configManager.getReferences().forEach(rc -> {
           try {
               ReferenceConfig<?> referenceConfig = (ReferenceConfig<?>)
rc;

               if (!referenceConfig.isRefreshed()) {
                   //刷新引用配置
```

```
                referenceConfig.refresh();
            }

            if (rc.shouldInit()) {
                if (referAsync || rc.shouldReferAsync()) {
                    ExecutorService executor =
executorRepository.getServiceReferExecutor();
                    CompletableFuture<Void> future =
CompletableFuture.runAsync(() -> {
                        try {
                            //间接的通过缓存对象来引用服务配置
                            referenceCache.get(rc);
                        } catch (Throwable t) {
                            logger.error(getIdentifier() + " refer async
catch error : " + t.getMessage(), t);
                        }
                    }, executor);

                    asyncReferringFutures.add(future);
                } else {
                    //间接的通过缓存对象来引用服务配置
                    referenceCache.get(rc);
                }
            }
        } catch (Throwable t) {
            logger.error(getIdentifier() + " refer catch error.");
            //出现异常销毁引用配置
            referenceCache.destroy(rc);
            throw t;
        }
    });
  }
```

在这个代码中我们核心需要关心的就是SimpleReferenceCache类型的get方法了，在获取服务对象之外包装了一层缓存。

如果出现了异常则执行 referenceCache 的 destroy 方法进行销毁引用配置。

## 3. 开始引用服务

### 1) SimpleReferenceCache 是什么？

一个用于缓存引用 ReferenceConfigBase 的 util 工具类。

ReferenceConfigBase 是一个重对象，对于频繁创建 ReferenceConfigBase 的框架来说，有必要缓存这些对象。

如果需要使用复杂的策略，可以实现并使用自己的 ReferenceConfigBase 缓存。

这个 Cache 是引用服务的开始如果我们想在代码中自定义一些服务引用的逻辑，可以直接创建 SimpleReferenceCache 类型对象然后调用其 get 方法进行引用服务。那这个缓存对象是和缓存与引用服务的可以继续往下看。

### 2) 引用服务之前的缓存处理逻辑？

关于逻辑的处理，看代码有时候比文字更清晰明了，这里可以直接来看 SimpleReferenceCache 类型的 get 方法。

```
@Override
    @SuppressWarnings("unchecked")
    public <T> T get(ReferenceConfigBase<T> rc) {
         //这个生成的 key 规则是这样的 服务分组/服务接口:版本号  详细的代码就不看了
         //例如:  group/link.elastic.dubbo.entity.DemoService:1.0
        String key = generator.generateKey(rc);
        //服务类型 如果是泛化调用则这个类型为 GenericService
        Class<?> type = rc.getInterfaceClass();

        //服务是否为单例的这里默认值都为空，为单例模式
        boolean singleton = rc.getSingleton() == null ||
rc.getSingleton();
        T proxy = null;
        // Check existing proxy of the same 'key' and 'type' first.
        if (singleton) {
            //一般为单例的 这个方法是从缓存中获取
            proxy = get(key, (Class<T>) type);
```

```
        } else {
            //非单例容易造成内存泄露，无法从缓存中获取
            logger.warn("Using non-singleton ReferenceConfig and
ReferenceCache at the same time may cause memory leak. " +
                "Call ReferenceConfig#get() directly for non-singleton
ReferenceConfig instead of using
ReferenceCache#get(ReferenceConfig)");
        }
        //前面是从缓存中拿，如果缓存中获取不到则开始引用服务
        if (proxy == null) {
            //获取或者创建值，为引用类型 referencesOfType 对象（类型为
Map<Class<?>, List<ReferenceConfigBase<?>>>）缓存对象生成值（值不存咋时候会
生成一个）
            List<ReferenceConfigBase<?>> referencesOfType =
referenceTypeMap.computeIfAbsent(type, _t ->
Collections.synchronizedList(new ArrayList<>()));
            //每次走到这里都会添加一个 ReferenceConfigBase 引用配置对象（单例的从
缓存中拿到就可以直接返回了）
            referencesOfType.add(rc);

            //与前面一样 前面是类型映射，这里是 key 映射
            List<ReferenceConfigBase<?>> referenceConfigList =
referenceKeyMap.computeIfAbsent(key, _k ->
Collections.synchronizedList(new ArrayList<>()));
            referenceConfigList.add(rc);
            //开始引用服务
            proxy = rc.get();
        }

        return proxy;
    }
```

可以看到这个逻辑使用了享元模式（其实就是先查缓存，缓存不存在则创建对象存入缓存）来进行引用对象的管理这样一个过程，这里一共有两个缓存对象 referencesOfType 和 referenceConfigList
key 分别为引用类型和引用的服务的 key，值是引用服务的基础配置对象列表 List<ReferenceConfigBase<?>>

后面可以详细看下如果借助 ReferenceConfigBase 类型对象来进行具体类型的引用。

## 4. 初始化引用服务的过程

### 1) 初始化引用服务的调用入口

引用服务的逻辑其实是相对复杂一点的，包含了服务发现，引用对象的创建等等，接下来就让我们详细看下。

ReferenceConfig 类型的 get 方法

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

        //ref类型为 transient volatile T ref;
        if (ref == null) {
            // ensure start module, compatible with old api usage
            //这个前面已经调用了模块发布器启动过了，这里有这么一行代码是有一定作用的，如果使用方直接调用了
ReferenceConfigBase的get方法或者缓存对象SimpleReferenceCache类型的对象的get方法来引用服务端的时候就会
造成很多配置没有初始化下面执行逻辑的时候出现问题，这个代码其实就是启动模块进行一些基础配置的初始化操作 比如元
数据中心默认配置选择，注册中心默认配置选择这些都是比较重要的
            getScopeModel().getDeployer().start();

            synchronized (this) {
                if (ref == null) {
                    init();
                }
            }
        }

        return ref;
    }
```

这里有一段代码是：getScopeModel().getDeployer().start();

这个前面已经调用了模块发布器启动过了，这里有这么一行代码是有一定作用的，如果使用方直接调用了 ReferenceConfigBase 的 get 方法或者缓存对象 SimpleReferenceCache 类型的对象的 get 方法来引用服务端的时候就会造成很多配置没有初始化下面执行逻辑的时候出现问题，这个代码其实就是启动模块进行一

些基础配置的初始化操作 比如元数据中心默认配置选择，注册中心默认配置选择这些都是比较重要的。

另外可以看到的是这里使用了双重校验锁来保证单例对象的创建，发现 Dubbo 种大量的使用了双重校验锁的逻辑。

### 2) 初始化引用服务

这个就直接看代码了这，初始化过程相对复杂一点，我们一点点来看

ReferenceConfig 类型 init()方法

```
protected synchronized void init() {
     //初始化标记变量保证只初始化一次，这里又是加锁🔐又是加标记变量的
       if (initialized) {
           return;
       }
       initialized = true;
       //刷新配置
       if (!this.isRefreshed()) {
           this.refresh();
       }

       // init serviceMetadata
       //初始化 ServiceMetadata 类型对象 serviceMetadata 为其设置服务基本属性比如版本号，分组，服务接口名
       initServiceMetadata(consumer);

     //继续初始化元数据信息 服务接口类型和 key
       serviceMetadata.setServiceType(getServiceInterfaceClass());
       // TODO, uncomment this line once service key is unified
       serviceMetadata.setServiceKey(URL.buildKey(interfaceName, group, version));

       //配置转 Map 类型
       Map<String, String> referenceParameters = appendConfig();
       // init service-application mapping
       //来自本地存储和 url 参数的初始化映射。 参数转 URL 配置初始化 Dubbo 中喜欢用 url 作为配置的一种处理方式
```

```
        initServiceAppsMapping(referenceParameters);
         //本地内存模块服务存储库
        ModuleServiceRepository repository =
getScopeModel().getServiceRepository();
        //ServiceModel 和 ServiceMetadata 在某种程度上是相互重复的。我们将来应该
合并它们。
        ServiceDescriptor serviceDescriptor;
        if (CommonConstants.NATIVE_STUB.equals(getProxy())) {
            serviceDescriptor =
StubSuppliers.getServiceDescriptor(interfaceName);
            repository.registerService(serviceDescriptor);
        } else {
            //本地存储库注册服务接口类型
            serviceDescriptor =
repository.registerService(interfaceClass);
        }
        //消费者模型对象
        consumerModel = new
ConsumerModel(serviceMetadata.getServiceKey(), proxy,
serviceDescriptor, this,
            getScopeModel(), serviceMetadata, createAsyncMethodInfo());
         //本地存储库注册消费者模型对象
        repository.registerConsumer(consumerModel);

        //与前面代码一样基础初始化服务元数据对象为其设置附加参数
        serviceMetadata.getAttachments().putAll(referenceParameters);
        //创建服务的代理对象 ！！！ 核心代码在这里
        ref = createProxy(referenceParameters);

        //为服务元数据对象设置代理对象
        serviceMetadata.setTarget(ref);
        serviceMetadata.addAttribute(PROXY_CLASS_REF, ref);

        consumerModel.setProxyObject(ref);
        consumerModel.initMethodModels();

        //检查 invoker 对象初始结果
        checkInvokerAvailable();
    }
```

## 5. ReferenceConfig 创建服务引用代理对象的原理

### 1) 代理对象的创建过程

这里就要继续看 ReferenceConfig 类型的创建代理方法 createProxy 了。

直接贴一下源码：

```
private T createProxy(Map<String, String> referenceParameters) {
    //本地引用 这里为 false
       if (shouldJvmRefer(referenceParameters)) {
           createInvokerForLocal(referenceParameters);
       } else {
           urls.clear();
           if (StringUtils.isNotEmpty(url)) {
               //url 存在则为点对点引用
               // user specified URL, could be peer-to-peer address, or
register center's address.
               parseUrl(referenceParameters);
           } else {
               // if protocols not in jvm checkRegistry
               //这里不是 local 协议默认这里为空
               if (!LOCAL_PROTOCOL.equalsIgnoreCase(getProtocol())) {
                   //从注册表中获取 URL 并将其聚合。这个其实就是初始化一下注册中心的
url 配置
                   aggregateUrlFromRegistry(referenceParameters);
               }
           }
           //这个代码非常重要 创建远程引用，创建远程引用调用器
           createInvokerForRemote();
       }

       if (logger.isInfoEnabled()) {
           logger.info("Referred dubbo service: [" +
referenceParameters.get(INTERFACE_KEY) + "]." +

(Boolean.parseBoolean(referenceParameters.get(GENERIC_KEY)) ?
                   " it's GenericService reference" : " it's not
GenericService reference"));
       }
```

```
        URL consumerUrl = new ServiceConfigURL(CONSUMER_PROTOCOL,
referenceParameters.get(REGISTER_IP_KEY), 0,
                referenceParameters.get(INTERFACE_KEY),
referenceParameters);
        consumerUrl = consumerUrl.setScopeModel(getScopeModel());
        consumerUrl = consumerUrl.setServiceModel(consumerModel);
        MetadataUtils.publishServiceDefinition(consumerUrl,
consumerModel.getServiceModel(), getApplicationModel());

        // create service proxy
        return (T) proxyFactory.getProxy(invoker,
ProtocolUtils.isGeneric(generic));
    }
```

### 2) 创建远程引用，创建远程引用调用器

ReferenceConfig 类型的 createInvokerForRemote 方法

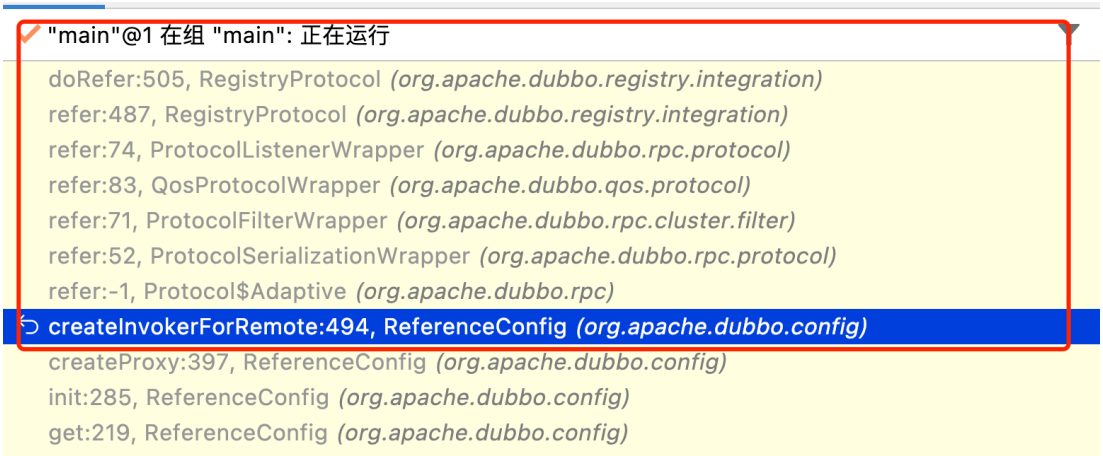
```
private void createInvokerForRemote() {
    //这个 url 为注册协议如
registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?ap
plication=dubbo-demo-api-
consumer&dubbo=2.0.2&pid=6204&qos.enable=false&qos.port=-
1&registry=zookeeper&release=3.0.9&timestamp=1657439419495
        if (urls.size() == 1) {
            URL curUrl = urls.get(0);
            //这个 SPI 对象是由字节码动态生成的自适应对象 Protocol$Adaptie 直接看
看不到源码，后续可以解析一个字节码生成的类型，这里后续来调用链路即可
            invoker = protocolSPI.refer(interfaceClass, curUrl);
            if (!UrlUtils.isRegistry(curUrl)) {
                List<Invoker<?>> invokers = new ArrayList<>();
                invokers.add(invoker);
                invoker = Cluster.getCluster(scopeModel,
Cluster.DEFAULT).join(new StaticDirectory(curUrl, invokers), true);
            }
        } else {
            List<Invoker<?>> invokers = new ArrayList<>();
            URL registryUrl = null;
            for (URL url : urls) {
```

```
                // For multi-registry scenarios, it is not checked
whether each referInvoker is available.
                // Because this invoker may become available later.
                invokers.add(protocolSPI.refer(interfaceClass, url));

                if (UrlUtils.isRegistry(url)) {
                    // use last registry url
                    registryUrl = url;
                }
            }

            if (registryUrl != null) {
                // registry url is available
                // for multi-subscription scenario, use 'zone-aware'
policy by default
                String cluster = registryUrl.getParameter(CLUSTER_KEY,
ZoneAwareCluster.NAME);
                // The invoker wrap sequence would be:
ZoneAwareClusterInvoker(StaticDirectory) -> FailoverClusterInvoker
                // (RegistryDirectory, routing happens here) -> Invoker
                invoker = Cluster.getCluster(registryUrl.getScopeModel(),
cluster, false).join(new StaticDirectory(registryUrl, invokers),
false);
            } else {
                // not a registry url, must be direct invoke.
                if (CollectionUtils.isEmpty(invokers)) {
                    throw new IllegalArgumentException("invokers ==
null");
                }
                URL curUrl = invokers.get(0).getUrl();
                String cluster = curUrl.getParameter(CLUSTER_KEY,
Cluster.DEFAULT);
                invoker = Cluster.getCluster(scopeModel,
cluster).join(new StaticDirectory(curUrl, invokers), true);
            }
        }
    }
```

### 3) Invoker 对象创建的全过程

为了更好理解 Protocol$Adaptie 内部的引用执行过程这里我把 Debug 的链路截图了过来，按照固定的顺序先执行 AOP 的逻辑再执行具体的逻辑：

- Protocol$Adaptie 的 refer 方法
- ProtocolSerializationWrapper AOP 类型的协议序列化器 refer 方法
- ProtocolFilterWrapper AOP 类型的协议过滤器的 refer 方法
- QosProtocolWrapper AOP 类型的 QOS 协议包装器的 refer 方法
- ProtocolListenerWrapper APO 类型监听器包装器的 refer 方法
- RegistryProtocol 注册协议的 refer 方法（会添加容错逻辑）
- RegistryProtocol 注册协议的 doRefer 方法（调用方法创建 Invoker 对象）




这里我们不再详细说这个引用链的具体过程直接定位到 RegistryProtocol 中创建 Invoker 类型的地方。

先来看 RegistryProtocol 类型的 refer 方法，如下代码所示：

RegistryProtocol 类型的 refer 方法

```
@Override
    @SuppressWarnings("unchecked")
    public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
        //这个url已经被转换为具体的注册中心协议类型了
        //zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?
application=dubbo-demo-api-
consumer&dubbo=2.0.2&pid=7944&qos.enable=false&qos.port=-1&release=3.0.9&timestamp=165744067
3100
        url = getRegistryUrl(url);
        //获取用于操作Zookeeper的Registry类型
        Registry registry = getRegistry(url);
        if (RegistryService.class.equals(type)) {
            return proxyFactory.getInvoker((T) registry, type, url);
        }

        // group="a,b" or group="*"
        Map<String, String> qs = (Map<String, String>) url.getAttribute(REFER_KEY);
        String group = qs.get(GROUP_KEY);
        if (StringUtils.isNotEmpty(group)) {
            if ((COMMA_SPLIT_PATTERN.split(group)).length > 1 || "*".equals(group)) {
                return doRefer(Cluster.getCluster(url.getScopeModel(),
MergeableCluster.NAME), registry, type, url, qs);
            }
        }
        //降级容错的逻辑处理对象 类型为Cluster 实际类型为MockClusterWrapper 内部包装的是
FailoverCluster
        //后续调用服务失败时候会先失效转移再降级
        Cluster cluster = Cluster.getCluster(url.getScopeModel(), qs.get(CLUSTER_KEY));
        //这里才是具体的Invoker对象的创建
        return doRefer(cluster, registry, type, url, qs);
    }
```

RegistryProtocol 类型的 doRefer 方法创建 Invoker 对象，直接来看代码了。

```
 protected <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL
url, Map<String, String> parameters) {
        Map<String, Object> consumerAttribute = new HashMap<>(url.getAttributes());
        consumerAttribute.remove(REFER_KEY);
        String p = isEmpty(parameters.get(PROTOCOL_KEY)) ? CONSUMER :
parameters.get(PROTOCOL_KEY);
        URL consumerUrl = new ServiceConfigURL (
            p,
            null,
            null,
            parameters.get(REGISTER_IP_KEY),
            0, getPath(parameters, type),
            parameters,
            consumerAttribute
        );
        url = url.putAttribute(CONSUMER_URL_KEY, consumerUrl);
        //重点看这一行 带迁移性质的Invoker对象
        ClusterInvoker<T> migrationInvoker = getMigrationInvoker(this, cluster, registry,
type, url, consumerUrl);
        //这一行回来执行迁移规则创建应用级优先的服务发现Invoker对象
        return interceptInvoker(migrationInvoker, url, consumerUrl);
    }
```

这里代码比较重要的其实只有两行 getMigrationInvoker 和 interceptInvoker 方法，比较核心也是 Dubbo3 比较重要的消费者启动逻辑基本都在这个方法里面 interceptInvoker，这个方法执行了消费者应用级发现和接口级发现迁移的逻辑，会自动帮忙决策一个 Invoker 类型对象，不过这个逻辑这里先简单看下，后续单独整个文章来说。

这里我们先来看 ClusterInvoker 对象的创建，下面先看代码。

RegistryProtocol 类型的 getMigrationInvoker 方法

```
 protected <T> ClusterInvoker<T> getMigrationInvoker(RegistryProtocol registryProtocol,
Cluster cluster, Registry registry, Class<T> type, URL url, URL consumerUrl) {
        return new ServiceDiscoveryMigrationInvoker<T>(registryProtocol, cluster, registry,
type, url, consumerUrl);
    }
```

详细的逻辑这里就不再看了，我们继续看 RegistryProtocol 类型的 interceptInvoker 方法。

具体代码如下：

RegistryProtocol 类型的 interceptInvoker 方法

```
 protected <T> Invoker<T> interceptInvoker(ClusterInvoker<T> invoker, URL url, URL
consumerUrl) {
    //获取激活的注册协议监听器扩展里面registry.protocol.listener，这里激活的类型为
MigrationRuleListener
        List<RegistryProtocolListener> listeners = findRegistryProtocolListeners(url);
        if (CollectionUtils.isEmpty(listeners)) {
            return invoker;
        }

        for (RegistryProtocolListener listener : listeners) {
            //这里执行MigrationRuleListener类型的onRefer方法
            listener.onRefer(this, invoker, consumerUrl, url);
        }
        return invoker;
    }
```

该方法尝试加载所有 RegistryProtocolListener 定义，这些定义通过与定义的交互来控制调用器的行为，然后使用这些侦听器更改 MigrationInvoker 的状态和行为。

当前可用的监听器是 MigrationRuleListener，用于通过动态变化的规则控制迁移行为。

可以看到核心的逻辑集中在了这个位置 MigrationRuleListener 类型的 onRefer 方法，这个这里就不深入往下说了，后续会有个文章专门来看 Dubbo2 迁移 Dubbo3 时候处理的逻辑。

Invoker 对象的创建完成其实就代表了服务引用执行完成，不过这里核心的协议并没有来说。

# 应用级服务发现源码解析

## 一、 应用级服务发现原理简介

```
title: "Dubbo3 应用级服务发现设计"
linkTitle: "Dubbo3 应用级服务发现设计"
date: 2023-01-30
author: Jun Liu
description: >
    当前版本的Dubbo Admin包含了之前版本中的绝大部分功能，包括服务治理，服务查询等，同时支持了Dubbo2.7中服务治理的新特性
```

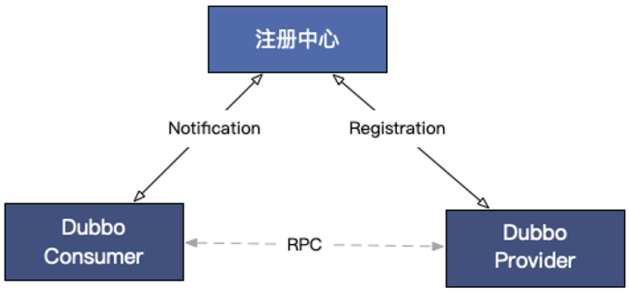
### 1. Objective

- 显著降低服务发现过程的资源消耗，包括提升注册中心容量上限、降低消费端地址解析资源占用等，使得 Dubbo3 框架能够支持更大规模集群的服务治理，实现无限水平扩容。

- 适配底层基础设施服务发现模型，如 Kubernetes、Service Mesh 等。

### 2. Background

**Dubbo 接口级服务发现 – 基本原理**



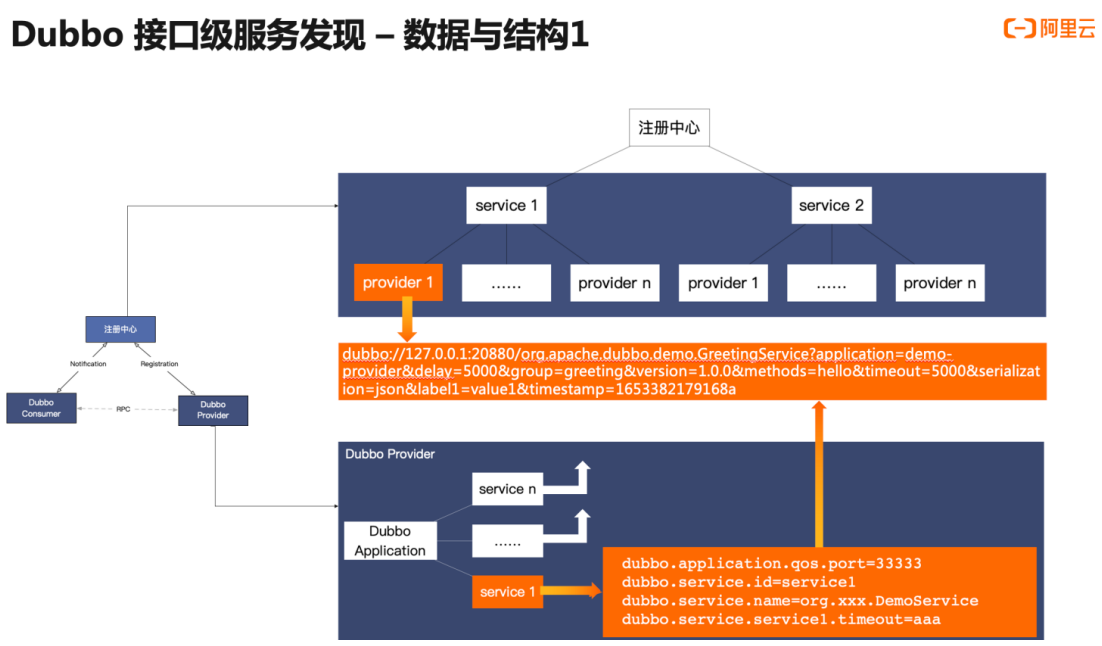

Dubbo 的地址发现是通过借助注册中心组件协调 Provider 与 Consumer 实例地址的过程。

- Provider 实例通过特定 key 向注册中心注册本机可访问地址
- 注册中心通过 key 将 Provider 实例地址聚合
- Consumer 通过订阅特定 key 实时从注册中心接收地址变更

我们从 Dubbo 最经典的工作原理图说起，Dubbo 从设计之初就内置了服务地址发现的能力，Provider 注册地址到注册中心，Consumer 通过订阅实时获取注册中心的地址更新，在收到地址列表后，consumer 基于特定的负载均衡策略发起对 provider 的 RPC 调用。

在这个过程中：

- 每个 Provider 通过特定的 key 向注册中心注册本机可访问地址。
- 注册中心通过这个 key 对 provider 实例地址进行聚合。
- Consumer 通过同样的 key 从注册中心订阅，以便及时收到聚合后的地址列表。




这里，我们对接口级地址发现的内部数据结构进行详细分析。

首先，看右下角 provider 实例内部的数据与行为。Provider 部署的应用中通常会有多个 Service，也就是 Dubbo2 中的服务，每个 service 都可能会有其独有的配置，我们所讲的 service 服务发布的过程，其实就是基于这个服务配置生成地址 URL 的过程，生成的地址数据如图所示；同样的，其他服务也都会生成地址。

然后，看一下注册中心的地址数据存储结构，注册中心以 service 服务名为数据划分依据，将一个服务下的所有地址数据都作为子节点进行聚合，子节点的内容就是实际可访问的 ip 地址，也就是我们 Dubbo 中 URL，格式就是刚才 provider 实例生成的。


这里把 URL 地址数据划分成了几份：

- 首先是实例可访问地址，主要信息包含 ip port，是消费端将基于这条数据生成 tcp 网络链接，作为后续 RPC 数据的传输载体。

- 其次是 RPC 元数据，元数据用于定义和描述一次 RPC 请求，一方面表明这条地址数据是与某条具体的 RPC 服务有关的，它的版本号、分组以及方法相关信息，另一方面表明。

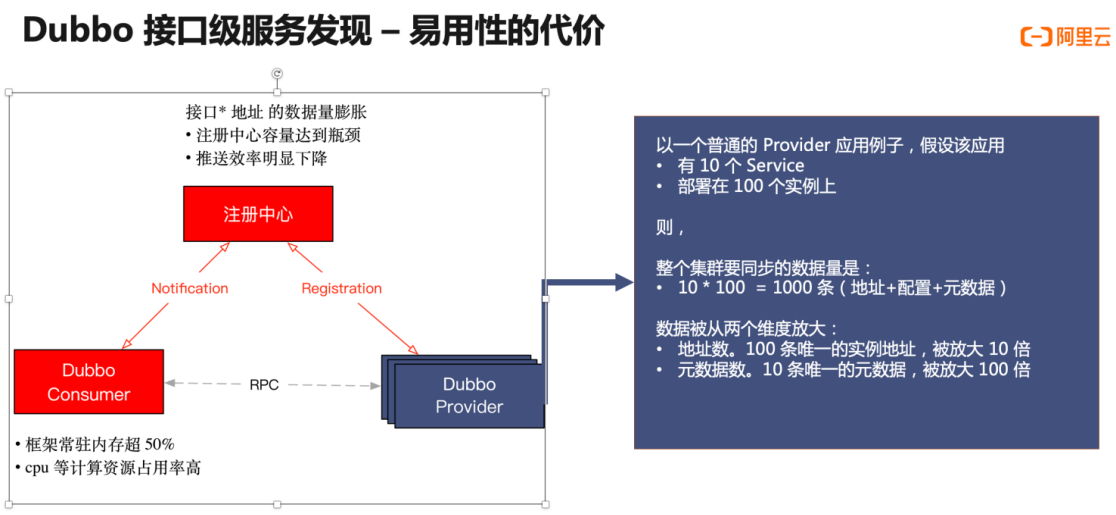
- 下一部分是 RPC 配置数据，部分配置用于控制 RPC 调用的行为，还有一部分配置用于同步 Provider 进程实例的状态，典型的如超时时间、数据编码的序列化方式等。

- 最后一部分是自定义的元数据，这部分内容区别于以上框架预定义的各项配置，给了用户更大的灵活性，用户可任意扩展并添加自定义元数据，以进一步丰富实例状态。

结合以上两页对于 Dubbo2 接口级地址模型的分析，以及最开始的 Dubbo 基本原理图，我们可以得出这么几条结论：

- 地址发现聚合的 key 就是 RPC 粒度的服务。
- 注册中心同步的数据不止包含地址，还包含了各种元数据以及配置。
- 得益于 1 与 2，Dubbo 实现了支持应用、RPC 服务、方法粒度的服务治理能力。

这就是一直以来 Dubbo2 在易用性、服务治理功能性、可扩展性上强于很多服务框架的真正原因。




一个事物总是有其两面性，Dubbo2 地址模型带来易用性和强大功能的同时，也给整个架构的水平可扩展性带来了一些限制。这个问题在普通规模的微服务集群下是完全感知不到的，而随着集群规模的增长，当整个集群内应用、机器达到一定数量时，整个集群内的各个组件才开始遇到规模瓶颈。在总结包括阿里巴巴、工商银行等多个典型的用户在生产环境特点后，我们总结出以下两点突出问题（如图中红色所示）：

- 首先，注册中心集群容量达到上限阈值。由于所有的 URL 地址数据都被发送到注册中心，注册中心的存储容量达到上限，推送效率也随之下降。

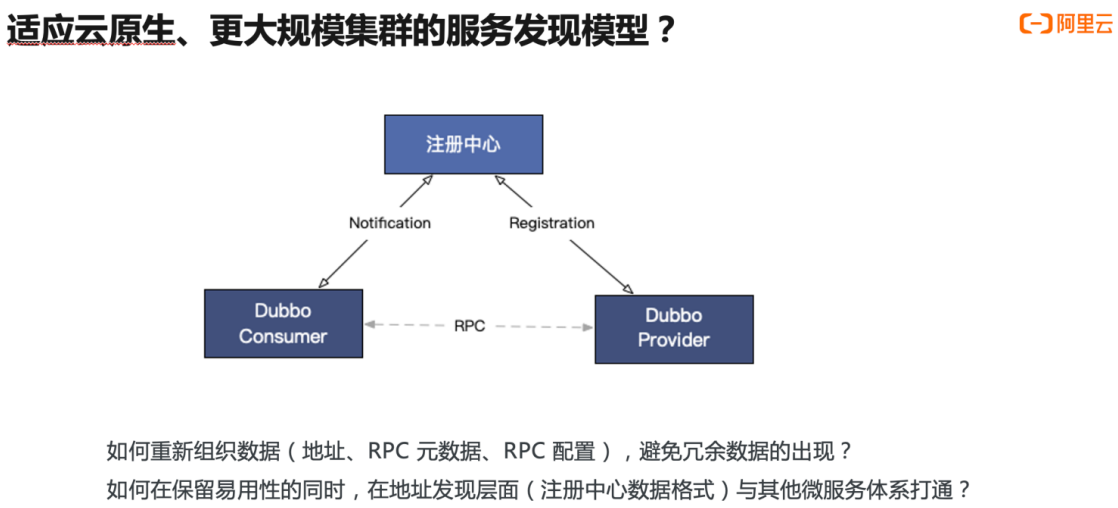
- 而在消费端这一侧，Dubbo2 框架常驻内存已超 40%，每次地址推送带来的 cpu 等资源消耗率也非常高，影响正常的业务调用。

为什么会出现这个问题？我们以一个具体 provider 示例进行展开，来尝试说明为何应用在接口级地址模型下容易遇到容量问题。

青蓝色部分，假设这里有一个普通的 Dubbo Provider 应用，该应用内部定义有 10 个 RPC Service，应用被部署在 100 个机器实例上。这个应用在集群中产生的数据量将会是“Service 数*机器实例数”，也就是 10*100=1000 条。数据被从两个维度放大：

- 从地址角度。100 条唯一的实例地址，被放大 10 倍。
- 从服务角度。10 条唯一的服务元数据，被放大 100 倍。

## 3. Proposal

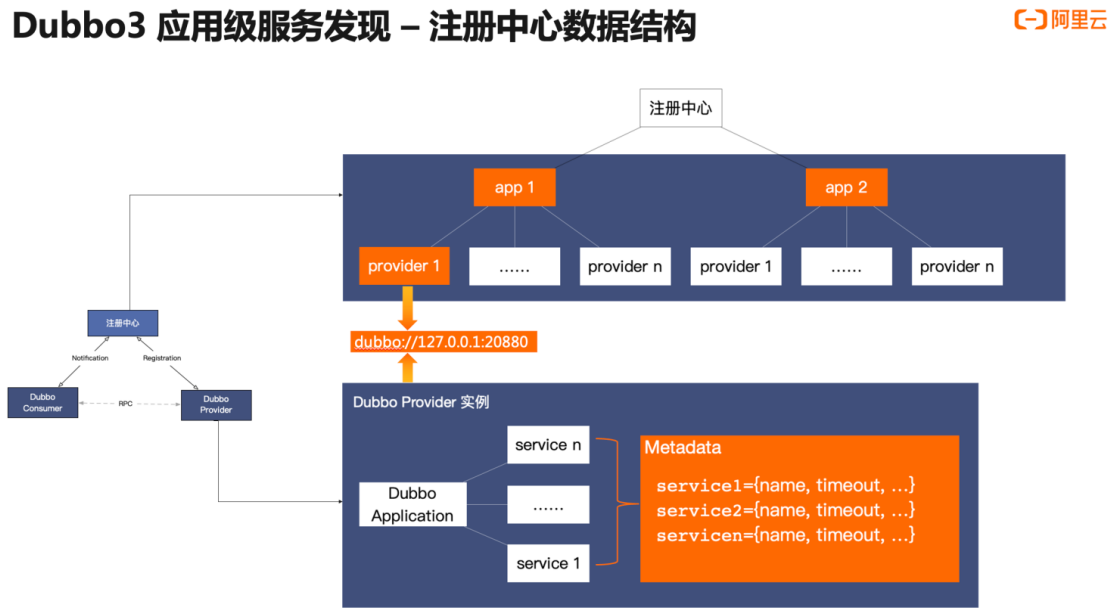



面对这个问题，在 Dubbo3 架构下，我们不得不重新思考两个问题：

- 如何在保留易用性、功能性的同时，重新组织 URL 地址数据，避免冗余数据的出现，让 Dubbo3 能支撑更大规模集群水平扩容？

- 如何在地址发现层面与其他的微服务体系如 Kubernetes、Spring Cloud 打通？


Dubbo3 的应用级服务发现方案设计本质上就是围绕以上两个问题展开。其基本思路是：地址发现链路上的聚合元素也就是我们之前提到的 Key 由服务调整为应用，这也是其名称叫做应用级服务发现的由来；另外，通过注册中心同步的数据内容上做了大幅精简，只保留最核心的 ip、port 地址数据。



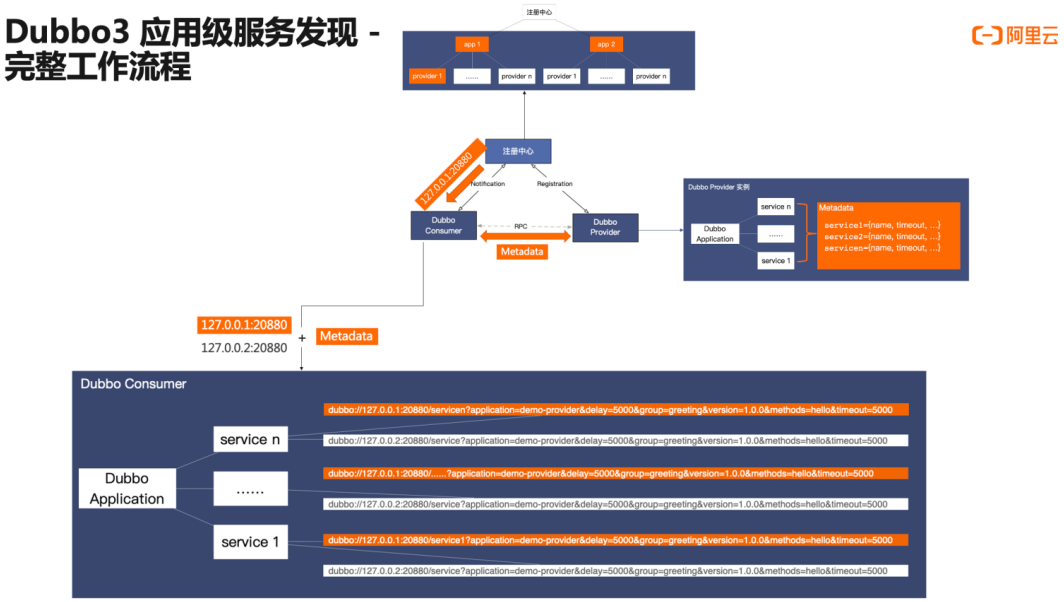

这是升级之后应用级地址发现的内部数据结构进行详细分析。

对比之前接口级的地址发现模型，我们主要关注橙色部分的变化。首先，在 provider 实例这一侧，相比于之前每个 RPC Service 注册一条地址数据，一个 provider 实例

只会注册一条地址到注册中心;而在注册中心这一侧,地址以应用名为粒度做聚合,应用名节点下是精简过后的 provider 实例地址。


应用级服务发现的上述调整，同时实现了地址单条数据大小和总数量的下降，但同时也带来了新的挑战：我们之前 Dubbo2 强调的易用性和功能性的基础损失了，因为元数据的传输被精简掉了，如何精细的控制单个服务的行为变得无法实现。

针对这个问题，Dubbo3 的解法是引入一个内置的 MetadataService 元数据服务，由中心化推送转为 Consumer 到 Provider 的点对点拉取，在这个模式下，元数据传输的数据量将不在是一个问题，因此可以在元数据中扩展出更多的参数、暴露更多的治理数据。



这里我们个重点看消费端 Consumer 的地址订阅行为，消费端从分两步读取地址数据，首先是从注册中心收到精简后的地址，随后通过调用 MetadataService 元数据服务，读取对端的元数据信息。在收到这两部分数据之后，消费端会完成地址数据的聚合，最终在运行态还原出类似 Dubbo2 的 URL 地址格式。因此从最终结果而言，应用级地址模型同时兼顾了地址传输层面的性能与运行层面的功能性。

以上就是的应用级服务发现背景、工作原理部分的所有内容，接下来我们看一下饿了么升级到 Dubbo3 尤其是应用级服务发现的过程。

## 二、 Dubbo3 消费者自动感应决策应用级服务发现原理

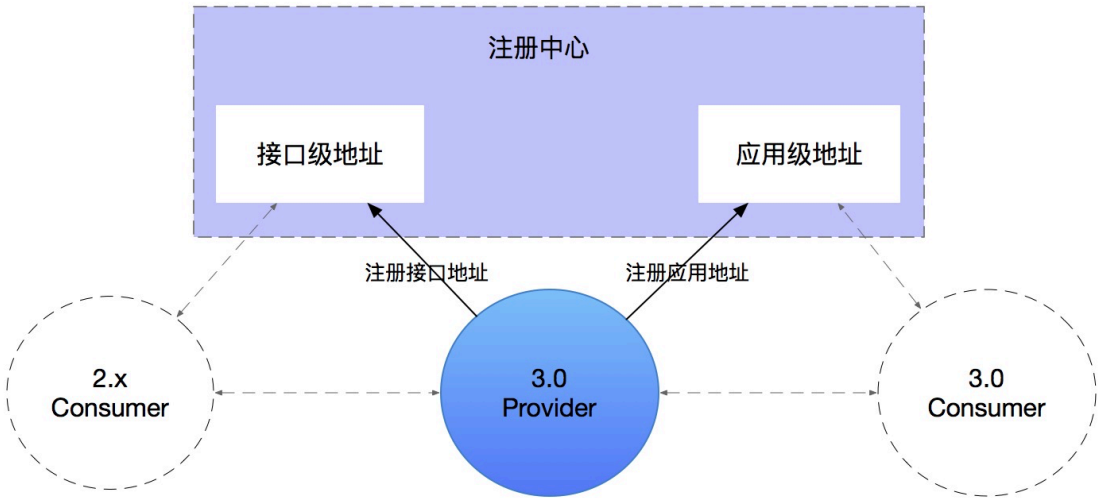
### 1. 简介

这里要说的内容对 Dubbo2 迁移到 Dubbo3 的应用比较有帮助，消费者应用级服务发现做了一些自动决策的逻辑来决定当前消费者是应用级发现还是接口级服务发现，这里与前面说的提供者双注册的原理是对等的，提供者默认同时进行应用级注册和接口级注册，消费者对提供者注册的数据来决定使用应用级发现或者接口级发现。这些都是默认的行为，当然对于消费者来说还可以自定义其他的迁移规则，具体的需要我们详细来看逻辑。

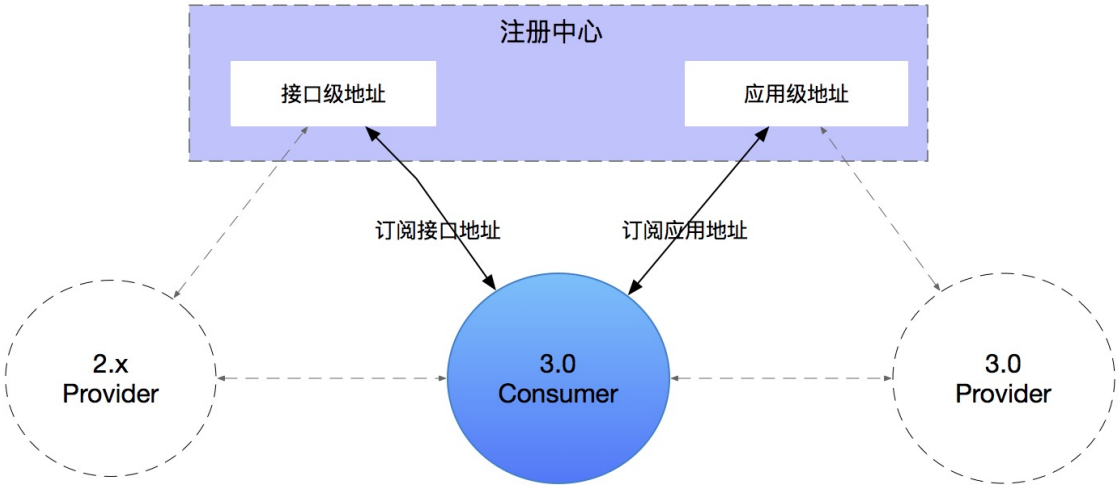
如果说对于迁移过程比较感兴趣可以直接去官网看文档相对来说还是比较清晰：点击此处查看

这里再借官网的图来用用，迁移过程主要如下所示。

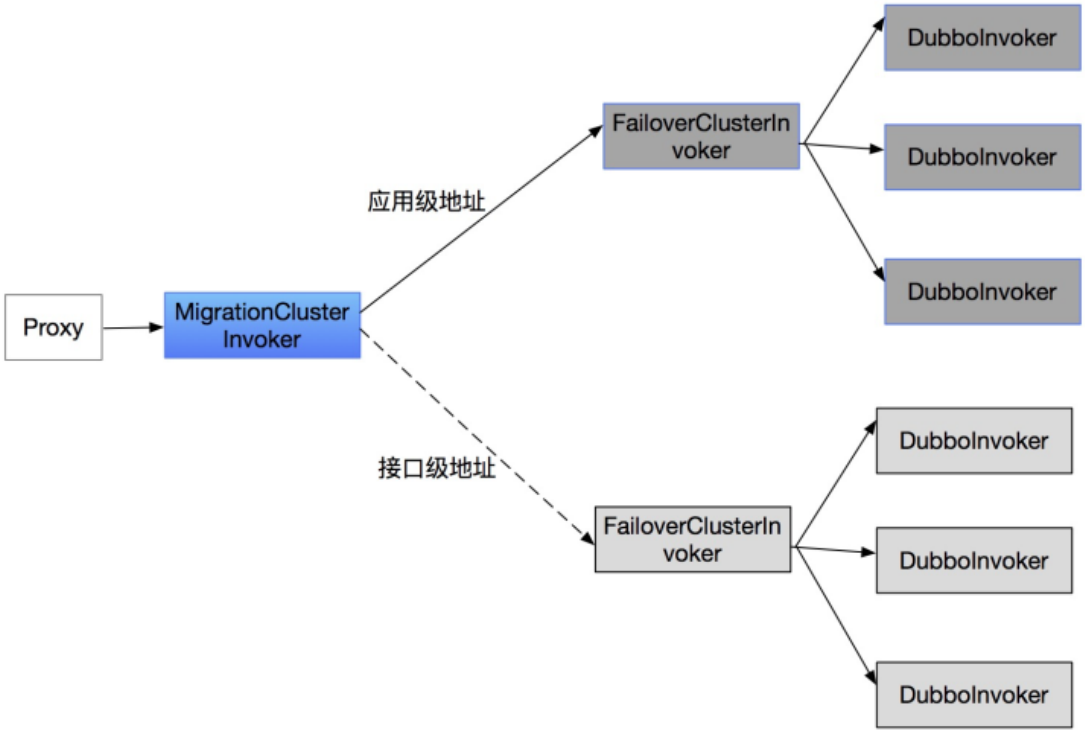
第一个图是提供者双注册的图：




第二个图是消费者订阅决策的图：




第三个图就是精确到消费者订阅的代码层的逻辑了，消费者服务间调用通过一个 Invoker 类型对象来进行对象，如下图所示消费者代理对象通过创建一个迁移容错的调用器对象来对应用级或者接口级订阅进行适配如下所示：




第二个图和第三个图是重点要关注的这一个文章的内容主要就是说这里的逻辑。

关于代码位置如果不知道是如何调用到这一块逻辑的可以查看博文《Dubbo3 消费者引用服务入口》。

这里直接将代码位置定位到：RegistryProtocol 类型的 interceptInvoker 方法中，如下所示：

RegistryProtocol 类型的 interceptInvoker 方法

```
 protected <T> Invoker<T> interceptInvoker(ClusterInvoker<T> invoker, URL url, URL
consumerUrl) {
  //目前存在的扩展类型为RegistryProtocolListener监听器的实现类型MigrationRuleListener
        List<RegistryProtocolListener> listeners = findRegistryProtocolListeners(url);
        if (CollectionUtils.isEmpty(listeners)) {
            return invoker;
        }

        for (RegistryProtocolListener listener : listeners) {
            listener.onRefer(this, invoker, consumerUrl, url);
        }
        return invoker;
    }
```

该方法尝试加载所有 RegistryProtocolListener 定义，这些定义通过与定义的交互来控制调用器的行为，然后使用这些侦听器更改 MigrationInvoker 的状态和行为。当前可用的监听器是 MigrationRuleListener，用于通过动态变化的规则控制迁移行为。

## 2. MigrationRuleListener 类型的 onRefer 方法

直接来看代码：

```
@Override
    public void onRefer(RegistryProtocol registryProtocol, ClusterInvoker<?> invoker, URL
consumerUrl, URL registryURL) {
       //创建一个对应invoker对象的MigrationRuleHandler类型对象 然后将其存放在缓存
Map<MigrationInvoker, MigrationRuleHandler>类型对象handles中
        MigrationRuleHandler<?> migrationRuleHandler =
handlers.computeIfAbsent((MigrationInvoker<?>) invoker, _key -> {
            ((MigrationInvoker<?>) invoker).setMigrationRuleListener(this);
            return new MigrationRuleHandler<>((MigrationInvoker<?>) invoker, consumerUrl);
        });

      //迁移规则执行 rule是封装了迁移的配置规则的信息对应类型MigrationRule类型，在初始化对象的时候进行了
配置初始化
        migrationRuleHandler.doMigrate(rule);
    }
```

关于这个 igrationRule 的文可以直接看官方的文档比较详细：地址迁移规则说明

这个迁移规则是为了更细粒度的迁移决策。

相关配置可以参考下面这个样例：

```
key：消费者应用名（必填）
step：状态名（必填）
threshold：决策阈值（默认1.0）
proportion：灰度比例（默认100）
delay：延迟决策时间（默认0）
force：强制切换（默认 false）
interfaces：接口粒度配置（可选）
  - serviceKey：接口名（接口 + ：+ 版本号）（必填）
    threshold：决策阈值
    proportion：灰度比例
    delay：延迟决策时间
    force：强制切换
    step：状态名（必填）
  - serviceKey：接口名（接口 + ：+ 版本号）
    step：状态名
applications：应用粒度配置（可选）
  - serviceKey：应用名（消费的上游应用名）（必填）
    threshold：决策阈值
    proportion：灰度比例
    delay：延迟决策时间
    force：强制切换
    step：状态名（必填）
```

不过为了简单起见暂时先不详细说这个配置细节，我们继续往下看。

## 3. 迁移规则处理器执行迁移规则 MigrationRuleHandler 类型的 doMigrate 方法

### 1) 迁移规则的模版方法

MigrationRuleHandler 类型的 doMigrate 方法代码如下：

```
public synchronized void doMigrate(MigrationRule rule) {
        //默认情况下这个类型是MigrationInvoker
        if (migrationInvoker instanceof ServiceDiscoveryMigrationInvoker) {
            refreshInvoker(MigrationStep.FORCE_APPLICATION, 1.0f, rule);
            return;
        }

      //迁移步骤，MigrationStep 一共有3种枚举情况：FORCE_INTERFACE, APPLICATION_FIRST,
FORCE_APPLICATION
        // initial step : APPLICATION_FIRST
        MigrationStep step = MigrationStep.APPLICATION_FIRST;
        float threshold = -1f;

        try {
            //获取配置的类型 默认走APPLICATION_FIRST
            step = rule.getStep(consumerURL);
            //threshold：决策阈值（默认-1.0）计算与获取
            threshold = rule.getThreshold(consumerURL);
        } catch (Exception e) {
            logger.error("Failed to get step and threshold info from rule: " + rule, e);
        }
        //刷洗调用器对象 来进行决策服务发现模式
        if (refreshInvoker(step, threshold, rule)) {
            // refresh success, update rule
            setMigrationRule(rule);
        }
    }
```

### 2） 服务发现调用器对象的选择（决策服务发现策略）

这里就是关键代码了：通过迁移配置和当前提供者注册信息来决定创建什么类型的调用器对象（Invoker）来为后续服务调用做准备。

MigrationRuleHandler 的 refreshInvoker，注意默认情况下这个 step 参数为 APPLICATION_FIRST

```
private boolean refreshInvoker(MigrationStep step, Float threshold,
MigrationRule newRule) {
       if (step == null || threshold == null) {
           throw new IllegalStateException("Step or threshold of
migration rule cannot be null");
       }
       MigrationStep originStep = currentStep;

       if ((currentStep == null || currentStep != step)
|| !currentThreshold.equals(threshold)) {
           boolean success = true;
```

```
            switch (step) {
                case APPLICATION_FIRST:
                    //默认和配置了应用级优先的服务发现则走这里

migrationInvoker.migrateToApplicationFirstInvoker(newRule);
                    break;
                case FORCE_APPLICATION:
                //配置了应用级服务发现则走这里
                    success =
migrationInvoker.migrateToForceApplicationInvoker(newRule);
                    break;
                case FORCE_INTERFACE:
                //配置了接口级服务发现则走这里
                default:
                    success =
migrationInvoker.migrateToForceInterfaceInvoker(newRule);
            }

            if (success) {
                setCurrentStepAndThreshold(step, threshold);
                logger.info("Succeed Migrated to " + step + " mode.
Service Name: " + consumerURL.getDisplayServiceKey());
                report(step, originStep, "true");
            } else {
                // migrate failed, do not save new step and rule
                logger.warn("Migrate to " + step + " mode failed.
Probably not satisfy the threshold you set "
                        + threshold + ". Please try re-publish
configuration if you still after check.");
                report(step, originStep, "false");
            }

            return success;
        }
        // ignore if step is same with previous, will continue override
rule for MigrationInvoker
        return true;
    }
```

可以看到这个代码做了判断的逻辑分别对应了 Dubbo3 消费者迁移的一个状态逻辑。

三种状态分别如下枚举类型。当前共存在三种状态：

- FORCE_INTERFACE（强制接口级）
- APPLICATION_FIRST（应用级优先）
- FORCE_APPLICATION（强制应用级）

通过代码我们可以看到默认情况下都会走 APPLICATION_FIRST（应用级优先）的策略，这里我们也重点来说 APPLICATION_FIRST（应用级优先）来看下 Dubbo3 是如何决策使用接口级还是应用级发现模型来兼容迁移的服务的。

### 3) 应用级优先的服务发现规则逻辑

这个规则就是智能选择应用级还是接口级的代码了，对应类型为 MigrationInvoker 的 migrateToApplicationFirstInvoker 方法，接下来我们详细看下。

MigrationInvoker 类型的 migrateToApplicationFirstInvoker 方法：

```
@Override
    public void migrateToApplicationFirstInvoker(MigrationRule newRule) {
        CountDownLatch latch = new CountDownLatch(0);
        //刷新接口级服务发现Invoker
        refreshInterfaceInvoker(latch);
        //刷新应用级服务发现Invoker类型对象
        refreshServiceDiscoveryInvoker(latch);

        // directly calculate preferred invoker, will not wait until address notify
        // calculation will re-occurred when address notify later
        //计算当前使用应用级还是接口级服务发现的Invoker对象
        calcPreferredInvoker(newRule);
    }
```

### 4) 刷新接口级服务发现 Invoker

MigrationInvoker 类型的 refreshInterfaceInvoker 方法

```
protected void refreshInterfaceInvoker(CountDownLatch latch) {
        clearListener(invoker);
        if (needRefresh(invoker)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Re-subscribing interface addresses for interface " +
type.getName());
            }

            if (invoker != null) {
                invoker.destroy();
            }
            invoker = registryProtocol.getInvoker(cluster, registry, type, url);
        }
        setListener(invoker, () -> {
            latch.countDown();
            if (reportService.hasReporter()) {
                reportService.reportConsumptionStatus(
                    reportService.createConsumptionReport(consumerUrl.getServiceInterface(),
consumerUrl.getVersion(), consumerUrl.getGroup(), "interface"));
            }
            if (step == APPLICATION_FIRST) {
                calcPreferredInvoker(rule);
            }
        });
    }
```

### 5) 刷新应用级服务发现 Invoker 类型对象

MigrationInvoker 类型的 refreshServiceDiscoveryInvoker 方法

```
protected void refreshServiceDiscoveryInvoker(CountDownLatch latch) {
        clearListener(serviceDiscoveryInvoker);
        if (needRefresh(serviceDiscoveryInvoker)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Re-subscribing instance addresses, current interface " +
type.getName());
            }

            if (serviceDiscoveryInvoker != null) {
                serviceDiscoveryInvoker.destroy();
            }
            serviceDiscoveryInvoker = registryProtocol.getServiceDiscoveryInvoker(cluster,
registry, type, url);
        }
        setListener(serviceDiscoveryInvoker, () -> {
            latch.countDown();
            if (reportService.hasReporter()) {
                reportService.reportConsumptionStatus(
                    reportService.createConsumptionReport(consumerUrl.getServiceInterface(),
consumerUrl.getVersion(), consumerUrl.getGroup(), "app"));
            }
            if (step == APPLICATION_FIRST) {
                calcPreferredInvoker(rule);
            }
        });
    }
```

### 6) 计算当前使用应用级还是接口级服务发现的 Invoker 对象

MigrationInvoker 类型的的 calcPreferredInvoker 方法

```
private synchronized void calcPreferredInvoker(MigrationRule migrationRule) {
        if (serviceDiscoveryInvoker == null || invoker == null) {
            return;
        }
        Set<MigrationAddressComparator> detectors =
ScopeModelUtil.getApplicationModel(consumerUrl == null ? null : consumerUrl.getScopeModel())

.getExtensionLoader(MigrationAddressComparator.class).getSupportedExtensionInstances();
        if (CollectionUtils.isNotEmpty(detectors)) {
            // pick preferred invoker
            // the real invoker choice in invocation will be affected by promotion
            if (detectors.stream().allMatch(comparator ->
comparator.shouldMigrate(serviceDiscoveryInvoker, invoker, migrationRule))) {
                this.currentAvailableInvoker = serviceDiscoveryInvoker;
            } else {
                this.currentAvailableInvoker = invoker;
            }
        }
    }
```

currentAvailableInvoker 是后期服务调用使用的 Invoker 对象。

## 三、 接口级地址订阅逻辑

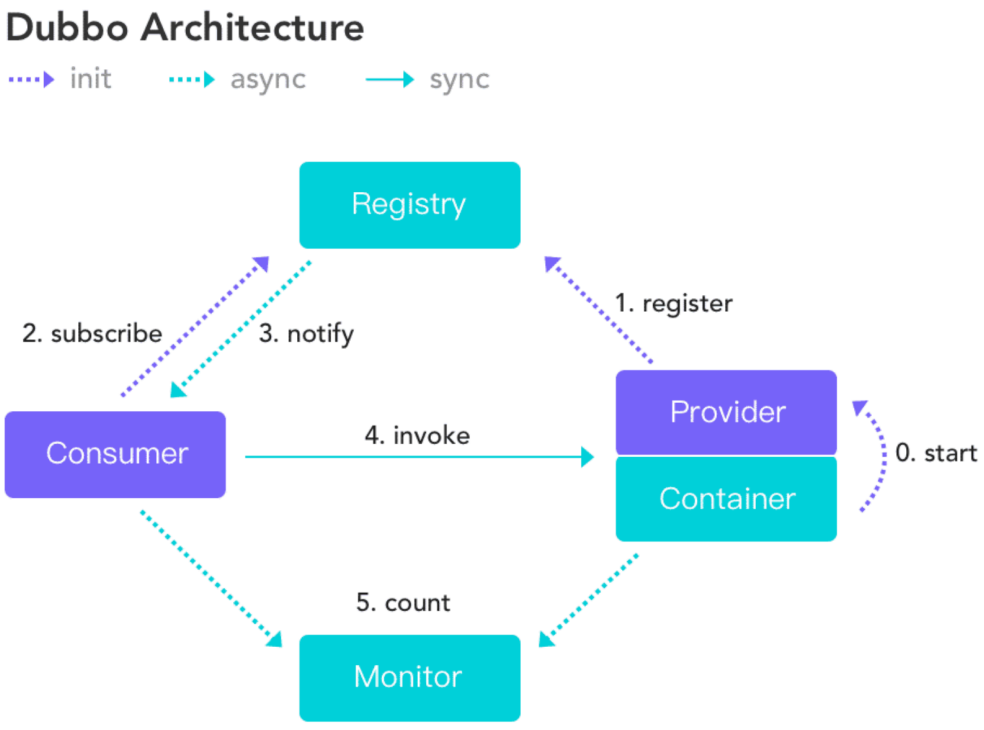
```
title: 23-【Dubbo3.0.8源码解析系列】消费者进行接口级服务发现订阅的逻辑
date: 2022-07-10 20:25:23
author: songxiaosheng
img: https://dubbo.apache.org/imgs/architecture.png
top: false
hide: false
cover: false
# coverImg: /images/1.jpg
password: 8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
toc: true
mathjax: false
summary: 23-Dubbo3消费者自动感应决策应用级服务发现原理
categories: Dubbo3源码
tags:
  - Dubbo
  - Dubbo3
  -  Dubbo3源码解析
```

## 1. 简介

前面说了《Dubbo3 消费者自动感应决策应用级服务发现原理》简单的说了下消费者是如何通过创建代理对象来进行应用级或者接口级的一个决策，只提到了迁移规则的使用与最终的 currentAvailableInvoker 对象的创建，并没有说详细的细节，这里我们来详细看看细节：

服务发现，即消费端自动发现服务地址列表的能力，是微服务框架需要具备的关键能力，借助于自动化的服务发现，微服务之间可以在无需感知对端部署位置与 IP 地址的情况下实现通信。

可以通过以下图的消费者与注册中心的交互逻辑来帮助理解。




消费者主动向注册中心订阅自己的信息，另外也会去注册中心拉取到提供者的信息来进行查询和订阅。

回到代码可以直接将位置定位到 MigrationInvoker 类型的 migrateToApplicationFirstInvoker 方法如下所示：

```
    @Override
    public void migrateToApplicationFirstInvoker(MigrationRule newRule) {
        CountDownLatch latch = new CountDownLatch(0);
        refreshInterfaceInvoker(latch);
        refreshServiceDiscoveryInvoker(latch);

        // directly calculate preferred invoker, will not wait until address notify
        // calculation will re-occurred when address notify later
        calcPreferredInvoker(newRule);
    }
```

这里我们主要说的逻辑为消费者进行接口级服务发现订阅的逻辑，所以我们主要看代码 refreshInterfaceInvoker 这个方法调用即可。

## 2. 关于 refreshInterfaceInvoker 方法

这里我们直接来看代码，MigrationInvoker 类型中的 refreshInterfaceInvoker 方法。

```
protected void refreshInterfaceInvoker(CountDownLatch latch) {
      //如果invoker对象存在则清理invoker对象的InvokersChangedListener
        clearListener(invoker);
         //invoker对象为空或者已经被销毁了则执行invoker对象创建的逻辑
        if (needRefresh(invoker)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Re-subscribing interface addresses for interface " +
type.getName());
            }

            if (invoker != null) {
                invoker.destroy();
            }
            //关键代码获取调用器invoker对象
            invoker = registryProtocol.getInvoker(cluster, registry, type, url);
        }
        setListener(invoker, () -> {
            latch.countDown();
            if (reportService.hasReporter()) {
                reportService.reportConsumptionStatus(
                    reportService.createConsumptionReport(consumerUrl.getServiceInterface(),
consumerUrl.getVersion(), consumerUrl.getGroup(), "interface"));
            }
            if (step == APPLICATION_FIRST) {
                calcPreferredInvoker(rule);
            }
        });
    }
```

## 3. 从注册中心协议中获取调用器对象 ClusterInvoker

前面我们可以看到当 invoker 对象不存在的时候会通过调用如下代码进行获取 Invoker 对象。

```
    //关键代码获取调用器invoker对象
     //这个registryProtocol类型为InterfaceCompatibleRegistryProtocol
    invoker = registryProtocol.getInvoker(cluster, registry, type, url);
```

### 1) InterfaceCompatibleRegistryProtocol 的 getInvoker 方法

接下来我们详细看 InterfaceCompatibleRegistryProtocol 类型的 getInvoker 方法如下所示：

```
@Override
    public <T> ClusterInvoker<T> getInvoker(Cluster cluster, Registry registry, Class<T>
type, URL url) {
        //创建注册中心目录（Directory这个英文单词翻译的结果为目录）
        DynamicDirectory<T> directory = new RegistryDirectory<>(type, url);
        //创建Invoker对象
        return doCreateInvoker(directory, cluster, registry, type);
    }
```

## 4. 服务目录 Directory

Directory 即服务目录，服务目录中存储了一些和服务提供者有关的信息，通过服务目录，服务消费者可获取到服务提供者的信息。在继续看源码之前我们下来分析下 Directory 之间的关系。

### 1) RegistryDirectory 对象的初始化

RegistryDirectory 注册中心目录的构造器，RegistryDirectory 类型的构造器源码：

```
public RegistryDirectory(Class<T> serviceType, URL url) {
        super(serviceType, url);
        moduleModel = getModuleModel(url.getScopeModel());
        //这里对应类型为：ConsumerConfigurationListener
        consumerConfigurationListener = getConsumerConfigurationListener(moduleModel);
    }
```

RegistryDirectory 类型的父类型 DynamicDirectory。

### 2) RegistryDirectory 对象的初始化

DynamicDirectory 的构造器。

```
public DynamicDirectory(Class<T> serviceType, URL url) {
        super(url, true);

        ModuleModel moduleModel = url.getOrDefaultModuleModel();
        //容错适配器，Cluster$Adaptive 默认的容错机制是失效转移 failover
        this.cluster = moduleModel.getExtensionLoader(Cluster.class).getAdaptiveExtension();
        //路由工厂适配器RouterFactory$Adaptive
        this.routerFactory =
moduleModel.getExtensionLoader(RouterFactory.class).getAdaptiveExtension();

        if (serviceType == null) {
            throw new IllegalArgumentException("service type is null.");
        }

        if (StringUtils.isEmpty(url.getServiceKey())) {
            throw new IllegalArgumentException("registry serviceKey is null.");
        }

        this.shouldRegister = !ANY_VALUE.equals(url.getServiceInterface()) &&
url.getParameter(REGISTER_KEY, true);
        this.shouldSimplified = url.getParameter(SIMPLIFIED_KEY, false);
        //这里对应我们的例子中的服务类型 为：link.elastic.dubbo.entity.DemoService
        this.serviceType = serviceType;
        //服务没有分组和版本 默认的key是服务信息 ： link.elastic.dubbo.entity.DemoService
        this.serviceKey = super.getConsumerUrl().getServiceKey();

        this.directoryUrl = consumerUrl;
        //分组信息查询
        String group = directoryUrl.getGroup("");
        this.multiGroup = group != null && (ANY_VALUE.equals(group) || group.contains(","));
        //服务目录信息为空是否快速失败 默认为true
        this.shouldFailFast =
Boolean.parseBoolean(ConfigurationUtils.getProperty(moduleModel,
Constants.SHOULD_FAIL_FAST_KEY, "true"));
    }
```

### 3) AbstractDirectory 抽象服务目录的构造器

先看前面调用道德构造器 isUrlFromRegistry 这个参数值为 true

```
public AbstractDirectory(URL url, boolean isUrlFromRegistry) {
      this(url, null, isUrlFromRegistry);
   }
```

AbstractDirectory 重载的构造器

```
public AbstractDirectory(URL url, RouterChain<T> routerChain, boolean
isUrlFromRegistry) {
       if (url == null) {
           throw new IllegalArgumentException("url == null");
       }

       this.url =
url.removeAttribute(REFER_KEY).removeAttribute(MONITOR_KEY);

       Map<String, String> queryMap;
      //注册中心中引用 URL 的关键字名称 这个查询到的是服务引用的一些配置信息
       Object referParams = url.getAttribute(REFER_KEY);
       if (referParams instanceof Map) {
           queryMap = (Map<String, String>) referParams;
           //查询
           this.consumerUrl = (URL) url.getAttribute(CONSUMER_URL_KEY);
       } else {
           queryMap =
StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
       }

       // remove some local only parameters
       ApplicationModel applicationModel =
url.getOrDefaultApplicationModel();
       this.queryMap =
applicationModel.getBeanFactory().getBean(ClusterUtils.class).mergeLoc
alParams(queryMap);

       if (consumerUrl == null) {
```

```
            String host = isNotEmpty(queryMap.get(REGISTER_IP_KEY)) ?
queryMap.get(REGISTER_IP_KEY) : this.url.getHost();
            String path = isNotEmpty(queryMap.get(PATH_KEY)) ?
queryMap.get(PATH_KEY) : queryMap.get(INTERFACE_KEY);
            String consumedProtocol =
isNotEmpty(queryMap.get(PROTOCOL_KEY)) ? queryMap.get(PROTOCOL_KEY) :
CONSUMER;

            URL consumerUrlFrom = this.url
                .setHost(host)
                .setPort(0)
                .setProtocol(consumedProtocol)
                .setPath(path);
            if (isUrlFromRegistry) {
                // reserve parameters if url is already a consumer url
                consumerUrlFrom = consumerUrlFrom.clearParameters();
            }
            this.consumerUrl = consumerUrlFrom.addParameters(queryMap);
        }
        //用于检查连接的线程池 核心线程数为 CPU 核心数，线程的名字为：Dubbo-
framework-connectivity-scheduler 分析线程时候看到这个名字要知道它的用处
        this.connectivityExecutor =
applicationModel.getFrameworkModel().getBeanFactory()
            .getBean(FrameworkExecutorRepository.class).getConnectivityS
cheduledExecutor();
        //获取全局配置，全局配置就是配置信息
        Configuration configuration =
ConfigurationUtils.getGlobalConfiguration(url.getOrDefaultModuleModel(
));
        //选择尝试重新连接的每个重新连接任务的最大调用程序数。 默认为 10
        //从 invokersToReconnect 中选取调用程序限制最大重新连接任务 TryCount，防
止此任务长时间挂起所有 ConnectionExecutor
        this.reconnectTaskTryCount =
configuration.getInt(RECONNECT_TASK_TRY_COUNT,
DEFAULT_RECONNECT_TASK_TRY_COUNT);
        //重连线程池两次触发重连任务的间隔时间，默认 1000 毫秒 重新连接任务的时间段
（如果需要）。（单位：毫秒）
        this.reconnectTaskPeriod =
configuration.getInt(RECONNECT_TASK_PERIOD,
DEFAULT_RECONNECT_TASK_PERIOD);
        //路由调用链
        setRouterChain(routerChain);
    }
```

## 5. 调用对象 ClusterInvoker 的创建过程 doCreateInvoker

前面介绍了创建服务目录对象，服务目录对象仅仅创建了服务目录的对象没有进行一些逻辑性操作比如服务查询，接下来我们继续看看如何借助服务目录工具来发现服务然后创建调用器 RegistryProtocol 的 doCreateInvoker 方法：

```
protected <T> ClusterInvoker<T> doCreateInvoker(DynamicDirectory<T> directory, Cluster
cluster, Registry registry, Class<T> type) {
        //初始化服务目录 为其设置 当前类型的的注册中心和协议
        directory.setRegistry(registry);
        directory.setProtocol(protocol);
        // all attributes of REFER_KEY
        Map<String, String> parameters = new HashMap<>
(directory.getConsumerUrl().getParameters());
        //消费者配置转ServiceConfigURL
        URL urlToRegistry = new ServiceConfigURL(
            parameters.get(PROTOCOL_KEY) == null ? CONSUMER : parameters.get(PROTOCOL_KEY),
            parameters.remove(REGISTER_IP_KEY),
            0,
            getPath(parameters, type),
            parameters
        );
        urlToRegistry =
urlToRegistry.setScopeModel(directory.getConsumerUrl().getScopeModel());
        urlToRegistry =
urlToRegistry.setServiceModel(directory.getConsumerUrl().getServiceModel());
        if (directory.isShouldRegister()) {
            directory.setRegisteredConsumerUrl(urlToRegistry);
            //这一行代码是将服务消费者的配置信息注册到注册中心的逻辑
            registry.register(directory.getRegisteredConsumerUrl());
        }
        //这一行代码是用来创建路由链的
        directory.buildRouterChain(urlToRegistry);
        //服务发现并订阅的逻辑
        directory.subscribe(toSubscribeUrl(urlToRegistry));
        //cluster类型为 MockClusterWrapper 包装了 FailoverCluster

        //这个是处理调用链路的 最前面的调用是容错然后回加上失效转移，过滤器负载均衡等等invoker执行的时候按
顺序执行
        return (ClusterInvoker<T>) cluster.join(directory, true);
    }
```

ListenerRegistryWrapper 的 register 方法

```
@Override
    public void register(URL url) {
        try {
            //这个registry类型为ZookeeperRegistry
            if (registry != null) {
                registry.register(url);
            }
        } finally {
            if (CollectionUtils.isNotEmpty(listeners) && !UrlUtils.isConsumer(url)) {
                RuntimeException exception = null;
                for (RegistryServiceListener listener : listeners) {
                    if (listener != null) {
                        try {
                            listener.onRegister(url, registry);
                        } catch (RuntimeException t) {
                            logger.error(t.getMessage(), t);
                            exception = t;
                        }
                    }
                }
                if (exception != null) {
                    throw exception;
                }
            }
        }
    }
```

ZookeeperRegistry 类型的父类型 FailbackRegistry 的 register 方法

```
@Override
    public void register(URL url) {
        if (!acceptable(url)) {
            logger.info("URL " + url + " will not be registered to
Registry. Registry " + this.getUrl() + " does not accept service of
this protocol type.");
            return;
        }
        super.register(url);
        removeFailedRegistered(url);
        removeFailedUnregistered(url);
        try {
            // Sending a registration request to the server side
            doRegister(url);
        } catch (Exception e) {
            Throwable t = e;

            // If the startup detection is opened, the Exception is
thrown directly.
```

```
            boolean check = getUrl().getParameter(Constants.CHECK_KEY,
true)
                    && url.getParameter(Constants.CHECK_KEY, true)
                    && (url.getPort() != 0);
            boolean skipFailback = t instanceof
SkipFailbackWrapperException;
            if (check || skipFailback) {
                if (skipFailback) {
                    t = t.getCause();
                }
                throw new IllegalStateException("Failed to register " +
url + " to registry " + getUrl().getAddress() + ", cause: " +
t.getMessage(), t);
            } else {
                logger.error("Failed to register " + url + ", waiting for
retry, cause: " + t.getMessage(), t);
            }

            // Record a failed registration request to a failed list,
retry regularly
            addFailedRegistered(url);
        }
    }
```

## AbstractRegistry 类型的 register 方法

```
    @Override
    public void register(URL url) {
        if (url == null) {
            throw new IllegalArgumentException("register url == null");
        }
        if (url.getPort() != 0) {
            if (logger.isInfoEnabled()) {
                logger.info("Register: " + url);
            }
        }
        registered.add(url);
    }
```

ZookeeperRegistry 的 doRegister 方法

```
    @Override
    public void doRegister(URL url) {
        try {
            checkDestroyed();
            //写入消费者路径 /dubbo/服务接口/consumers/消费者配置url  第二个参数是否为临时节点默认是的，如果动态配置为false就会是持久节点了
            zkClient.create(toUrlPath(url), url.getParameter(DYNAMIC_KEY, true));
        } catch (Throwable e) {
            throw new RpcException("Failed to register " + url + " to zookeeper " + getUrl() + ", cause: " + e.getMessage(), e);
        }
    }
```

前面的 zkClient 会写入一个消费者的路径如下所示：

```
/dubbo/link.elastic.dubbo.entity.DemoService/consumers/consumer%3A%2F%2F192.168.1.169%2Flink.elastic.dubbo.entity.DemoService%3Fapplication%3Ddubbo-demo-api-consumer%26background%3Dfalse%26category%3Dconsumers%26check%3Dfalse%26dubbo%3D2.0.2%26interface%3Dlink.elastic.dubbo.entity.DemoService%26methods%3DsayHello%2CsayHelloAsync%26pid%3D52237%26qos.enable%3Dfalse%26qos.port%3D-1%26release%3D3.0.10%26side%3Dconsumer%26sticky%3Dfalse%26timestamp%3D1659862505044
```

RegistryDirectory 类型的 subscribe 方法

```
  @Override
    public void subscribe(URL url) {
        super.subscribe(url);
        if (moduleModel.getModelEnvironment().getConfiguration().convert(Boolean.class, org.apache.dubbo.registry.Constants.ENABLE_CONFIGURATION_LISTEN, true)) {
            consumerConfigurationListener.addNotifyListener(this);
            referenceConfigurationListener = new ReferenceConfigurationListener(moduleModel, this, url);
        }
    }
```

DynamicDirectory 类型的 subscribe 方法

```
public void subscribe(URL url) {
        setSubscribeUrl(url);
        registry.subscribe(url, this);
    }
```

ListenerRegistryWrapper 类型的 subscribe 方法

```
@Override
    public void subscribe(URL url, NotifyListener listener) {
        try {
            if (registry != null) {
                registry.subscribe(url, listener);
            }
        } finally {
            if (CollectionUtils.isNotEmpty(listeners)) {
                RuntimeException exception = null;
                for (RegistryServiceListener registryListener : listeners) {
                    if (registryListener != null) {
                        try {
                            registryListener.onSubscribe(url, registry);
                        } catch (RuntimeException t) {
                            logger.error(t.getMessage(), t);
                            exception = t;
                        }
                    }
                }
                if (exception != null) {
                    throw exception;
                }
            }
        }
    }
```

FailbackRegistry 类型的 subscribe 方法

```
@Override
    public void subscribe(URL url, NotifyListener listener) {
        super.subscribe(url, listener);
        removeFailedSubscribed(url, listener);
        try {
            // Sending a subscription request to the server side
            doSubscribe(url, listener);
        } catch (Exception e) {
            Throwable t = e;

            List<URL> urls = getCacheUrls(url);
            if (CollectionUtils.isNotEmpty(urls)) {
                notify(url, listener, urls);
                logger.error("Failed to subscribe " + url + ", Using
cached list: " + urls + " from cache file: " +
getCacheFile().getName() + ", cause: " + t.getMessage(), t);
            } else {
```

```
            // If the startup detection is opened, the Exception is
thrown directly.
            boolean check =
getUrl().getParameter(Constants.CHECK_KEY, true)
                    && url.getParameter(Constants.CHECK_KEY, true);
            boolean skipFailback = t instanceof
SkipFailbackWrapperException;
            if (check || skipFailback) {
                if (skipFailback) {
                    t = t.getCause();
                }
                throw new IllegalStateException("Failed to subscribe "
+ url + ", cause: " + t.getMessage(), t);
            } else {
                logger.error("Failed to subscribe " + url + ", waiting
for retry, cause: " + t.getMessage(), t);
            }
        }

        // Record a failed registration request to a failed list,
retry regularly
        addFailedSubscribed(url, listener);
    }
}
```

## AbstractRegistry 类型的 subscribe 方法

```
    @Override
    public void subscribe(URL url, NotifyListener listener) {
        if (url == null) {
            throw new IllegalArgumentException("subscribe url == null");
        }
        if (listener == null) {
            throw new IllegalArgumentException("subscribe listener == null");
        }
        if (logger.isInfoEnabled()) {
            logger.info("Subscribe: " + url);
        }
        // ConcurrentMap<URL, Set<NotifyListener>> subscribed集合用来记录消费者对应的通知监听器
        Set<NotifyListener> listeners = subscribed.computeIfAbsent(url, n -> new
ConcurrentHashSet<>());
        listeners.add(listener);
    }
```

ZookeeperRegistry 的 doSubscribe 方法：

```
@Override
   public void doSubscribe(final URL url, final NotifyListener
listener) {
       try {
           checkDestroyed();
           if (ANY_VALUE.equals(url.getServiceInterface())) {
              String root = toRootPath();
              boolean check = url.getParameter(CHECK_KEY, false);
               ConcurrentMap<NotifyListener, ChildListener> listeners =
zkListeners.computeIfAbsent(url, k -> new ConcurrentHashMap<>());
               ChildListener zkListener =
listeners.computeIfAbsent(listener, k -> (parentPath, currentChildren)
-> {
                   for (String child : currentChildren) {
                       child = URL.decode(child);
                       if (!anyServices.contains(child)) {
                           anyServices.add(child);

subscribe(url.setPath(child).addParameters(INTERFACE_KEY, child,
                                   Constants.CHECK_KEY, String.valueOf(check)),
k);
                       }
                   }
               });
               zkClient.create(root, false);
               List<String> services = zkClient.addChildListener(root,
zkListener);
               if (CollectionUtils.isNotEmpty(services)) {
                   for (String service : services) {
                       service = URL.decode(service);
                       anyServices.add(service);

subscribe(url.setPath(service).addParameters(INTERFACE_KEY, service,
                           Constants.CHECK_KEY, String.valueOf(check)),
listener);
                   }
               }
           } else {
               CountDownLatch latch = new CountDownLatch(1);
               try {
```

```
                List<URL> urls = new ArrayList<>();
                //接口级默认的路径有 3 个我们暂时需要关注的：
                // 提供者：对应
dubbo/link.elastic.dubbo.entity.DemoService/providers
                //配置：
dubbo/link.elastic.dubbo.entity.DemoService/configurators
                //路由：
/dubbo/link.elastic.dubbo.entity.DemoService/routers
                for (String path : toCategoriesPath(url)) {
                    ConcurrentMap<NotifyListener, ChildListener>
listeners = zkListeners.computeIfAbsent(url, k -> new
ConcurrentHashMap<>());
                    //这里有个监听器 RegistryChildListenerImpl
                    ChildListener zkListener =
listeners.computeIfAbsent(listener, k -> new
RegistryChildListenerImpl(url, k, latch));
                    if (zkListener instanceof
RegistryChildListenerImpl) {
                        ((RegistryChildListenerImpl)
zkListener).setLatch(latch);
                    }
                    //创建非临时节点，不存在时候会创建 比如
/dubbo/link.elastic.dubbo.entity.DemoService/providers

///dubbo/link.elastic.dubbo.entity.DemoService/configurators
                    zkClient.create(path, false);
                    //服务目录创建完毕之后创建一个监听器用来监听子目录，同时要返
回一个 path 目录的子子节点，比如 providers 下面的提供者节点列表，如果有多个可以返回
多个，下面以 1 个的情况举例子

//dubbo%3A%2F%2F192.168.1.169%3A20880%2Flink.elastic.dubbo.entity.Demo
Service%3Fanyhost%3Dtrue%26application%3Ddubbo-demo-api-
provider%26background%3Dfalse%26deprecated%3Dfalse%26dubbo%3D2.0.2%26d
ynamic%3Dtrue%26generic%3Dfalse%26interface%3Dlink.elastic.dubbo.entit
y.DemoService%26methods%3DsayHello%2CsayHelloAsync%26pid%3D51534%26rel
ease%3D3.0.10%26service-name-
mapping%3Dtrue%26side%3Dprovider%26timestamp%3D1659860685159
                    List<String> children =
zkClient.addChildListener(path, zkListener);
                    if (children != null) {
                        //urls 存储的是当前服务对应服务配置的路径比如提供者
                        urls.addAll(toUrlsWithEmpty(url, path,
children));
```

```
                    }
                }
                //通知方法（服务订阅）
                notify(url, listener, urls);
            } finally {
                // tells the listener to run only after the sync
notification of main thread finishes.
                latch.countDown();
            }
        }
    } catch (Throwable e) {
        throw new RpcException("Failed to subscribe " + url + " to
zookeeper " + getUrl() + ", cause: " + e.getMessage(), e);
    }
}
```

前面这里查询到的 urls 默认一共有 3 种这里我只有一个提供者无配置，无路由数据将会获取到如下的 url 信息。

提供者

```
dubbo://192.168.1.169:20880/link.elastic.dubbo.entity.DemoService?
anyhost=true&application=dubbo-demo-api-
provider&background=false&category=providers,configurators,routers&check=false&deprecated=fa
lse&dubbo=2.0.2&dynamic=true&generic=false&interface=link.elastic.dubbo.entity.DemoService&m
ethods=sayHello,sayHelloAsync&pid=52816&qos.enable=false&qos.port=-1&release=3.0.10&service-
name-mapping=true&side=provider&sticky=false
```

配置信息

```
empty://192.168.1.169/link.elastic.dubbo.entity.DemoService?application=dubbo-demo-api-
consumer&background=false&category=configurators&dubbo=2.0.2&interface=link.elastic.dubbo.en
tity.DemoService&methods=sayHello,sayHelloAsync&pid=52816&qos.enable=false&qos.port=-1&relea
se=3.0.10&side=consumer&sticky=false&timestamp=1659863678563
```

路由信息

```
empty://192.168.1.169/link.elastic.dubbo.entity.DemoService?application=dubbo-demo-api-
consumer&background=false&category=routers&dubbo=2.0.2&interface=link.elastic.dubbo.entity.D
emoService&methods=sayHello,sayHelloAsync&pid=52816&qos.enable=false&qos.port=-1&release=3.0
.10&side=consumer&sticky=false&timestamp=1659863678563
```

观察一下可以看到如果无配置或者无路由的时候会有 empty 协议的 url

FailbackRegistry 类型的 notify 方法：

```
@Override
   protected void notify(URL url, NotifyListener listener, List<URL> urls) {
       if (url == null) {
           throw new IllegalArgumentException("notify url == null");
       }
       if (listener == null) {
           throw new IllegalArgumentException("notify listener == null");
       }
       try {
           doNotify(url, listener, urls);
       } catch (Exception t) {
           // Record a failed registration request to a failed list
           logger.error("Failed to notify addresses for subscribe " + url + ", cause: " +
t.getMessage(), t);
       }
   }
```

FailbackRegistry 类型的 doNotify 方法

```
protected void doNotify(URL url, NotifyListener listener, List<URL> urls) {
      super.notify(url, listener, urls);
   }
```

AbstractRegistry 类型的 notify 服务通知模版方法

```
protected void notify(URL url, NotifyListener listener, List<URL>
urls) {
       if (url == null) {
           throw new IllegalArgumentException("notify url == null");
       }
       if (listener == null) {
           throw new IllegalArgumentException("notify listener ==
null");
       }
       if ((CollectionUtils.isEmpty(urls))
&& !ANY_VALUE.equals(url.getServiceInterface())) {
           logger.warn("Ignore empty notify urls for subscribe url " +
url);
```

```
            return;
        }
        if (logger.isInfoEnabled()) {
            logger.info("Notify urls for subscribe url " + url + ", url
size: " + urls.size());
        }
        // keep every provider's category.
        Map<String, List<URL>> result = new HashMap<>();
        //这个例子中的 urls 会有 3 个可以看前面的说的
        for (URL u : urls) {
            //这里重点关注 isMatch 方法
            //这个 match 方法判断了消费者和提供者的分区+服务接口是否一致 或者其中一
个配置为泛化配置：*
            if (UrlUtils.isMatch(url, u)) {
                //获取当前分类
                String category = u.getCategory(DEFAULT_CATEGORY);
                List<URL> categoryList = result.computeIfAbsent(category,
k -> new ArrayList<>());
                categoryList.add(u);
            }
        }
        if (result.size() == 0) {
            return;
        }
        Map<String, List<URL>> categoryNotified =
notified.computeIfAbsent(url, u -> new ConcurrentHashMap<>());

        //下面这个循环很重要会将所有的注册中心注册的数据推给 notify 方法包含配置，路
由，服务提供者等
        for (Map.Entry<String, List<URL>> entry : result.entrySet()) {
            String category = entry.getKey();
            List<URL> categoryList = entry.getValue();
            categoryNotified.put(category, categoryList);
            //RegistryDirectory 类型的 notify
            listener.notify(categoryList);
            // We will update our cache file after each notification.
            // When our Registry has a subscribed failure due to network
jitter, we can return at least the existing cache URL.
            if (localCacheEnabled) {
                //缓存当前服务信息：
consumer://192.168.1.169/link.elastic.dubbo.entity.DemoService?applica
tion=dubbo-demo-api-
consumer&background=false&category=providers,configurators,routers&dub
```

```
bo=2.0.2&interface=link.elastic.dubbo.entity.DemoService&methods=sayHe
llo,sayHelloAsync&pid=36046&qos.enable=false&qos.port=-
1&release=3.0.10&side=consumer&sticky=false&timestamp=1660987214249
                saveProperties(url);
            }
        }
    }
```

接收服务

RegistryDirectory 类型的 notify 方法

```
@Override
    public synchronized void notify(List<URL> urls) {
        if (isDestroyed()) {
            return;
        }

        Map<String, List<URL>> categoryUrls = urls.stream()
            .filter(Objects::nonNull)
            .filter(this::isValidCategory)
            .filter(this::isNotCompatibleFor26x)
            .collect(Collectors.groupingBy(this::judgeCategory));

        List<URL> configuratorURLs =
categoryUrls.getOrDefault(CONFIGURATORS_CATEGORY,
Collections.emptyList());
        this.configurators =
Configurator.toConfigurators(configuratorURLs).orElse(this.configurato
rs);

        List<URL> routerURLs =
categoryUrls.getOrDefault(ROUTERS_CATEGORY, Collections.emptyList());
        toRouters(routerURLs).ifPresent(this::addRouters);

        // providers
        List<URL> providerURLs =
categoryUrls.getOrDefault(PROVIDERS_CATEGORY,
Collections.emptyList());

        // 3.x added for extend URL address
```

```
        ExtensionLoader<AddressListener> addressListenerExtensionLoader
=
getUrl().getOrDefaultModuleModel().getExtensionLoader(AddressListener.
class);
        List<AddressListener> supportedListeners =
addressListenerExtensionLoader.getActivateExtension(getUrl(),
(String[]) null);
        if (supportedListeners != null && !supportedListeners.isEmpty())
{
            for (AddressListener addressListener : supportedListeners) {
                providerURLs = addressListener.notify(providerURLs,
getConsumerUrl(), this);
            }
        }
        refreshOverrideAndInvoker(providerURLs);
    }
```

刷新服务通知的方法有以下几种：

RegistryDirectory 类型的 refreshOverrideAndInvoker 方法

```
private synchronized void refreshOverrideAndInvoker(List<URL> urls) {
        // mock zookeeper://xxx?mock=return null
        refreshInvoker(urls);
    }
```

RegistryDirectory 类型的 refreshInvoker 方法

```
private void refreshInvoker(List<URL> invokerUrls) {
        Assert.notNull(invokerUrls, "invokerUrls should not be null");

        if (invokerUrls.size() == 1
            && invokerUrls.get(0) != null
            && EMPTY_PROTOCOL.equals(invokerUrls.get(0).getProtocol()))
{
            this.forbidden = true; // Forbid to access
            routerChain.setInvokers(BitList.emptyList());
            destroyAllInvokers(); // Close all invokers
        } else {
            this.forbidden = false; // Allow to access
```

```
            if (invokerUrls == Collections.<URL>emptyList()) {
                invokerUrls = new ArrayList<>();
            }
            //使用本地引用以避免 NPE。destroyAllInvokers () 将
cachedInvokerUrls 设置为空。
            // use local reference to avoid NPE as
this.cachedInvokerUrls will be set null by destroyAllInvokers().
            Set<URL> localCachedInvokerUrls = this.cachedInvokerUrls;
            if (invokerUrls.isEmpty() && localCachedInvokerUrls != null)
{
                logger.warn("Service" + serviceKey + " received empty
address list with no EMPTY protocol set, trigger empty protection.");
                invokerUrls.addAll(localCachedInvokerUrls);
            } else {
                localCachedInvokerUrls = new HashSet<>();
                //缓存的调用器 URL，便于比较
                localCachedInvokerUrls.addAll(invokerUrls);//Cached
invoker urls, convenient for comparison
                this.cachedInvokerUrls = localCachedInvokerUrls;
            }
            if (invokerUrls.isEmpty()) {
                return;
            }

            ////使用本地引用以避免 NPE。urlInvokerMap 将在 destroyAllInvokers
 () 处同时设置为 null。
            // use local reference to avoid NPE as this.urlInvokerMap
will be set null concurrently at destroyAllInvokers().
            Map<URL, Invoker<T>> localUrlInvokerMap =
this.urlInvokerMap;
            //无法使用本地引用，因为 oldUrlInvokerMap 的映射可能会在 toInvokers
 () 处直接删除。
            // can't use local reference as oldUrlInvokerMap's mappings
might be removed directly at toInvokers().
            Map<URL, Invoker<T>> oldUrlInvokerMap = null;
            if (localUrlInvokerMap != null) {
                // the initial capacity should be set greater than the
maximum number of entries divided by the load factor to avoid
resizing.
                oldUrlInvokerMap = new LinkedHashMap<>(Math.round(1 +
localUrlInvokerMap.size() / DEFAULT_HASHMAP_LOAD_FACTOR));
                localUrlInvokerMap.forEach(oldUrlInvokerMap::put);
            }
```

```
            //将 URL 转换为 Invoker  这里会做一些协议的指定过滤操作
            Map<URL, Invoker<T>> newUrlInvokerMap =
toInvokers(oldUrlInvokerMap, invokerUrls);// Translate url list to
Invoker map

            /**
             * If the calculation is wrong, it is not processed.
             *
             * 1. The protocol configured by the client is inconsistent
with the protocol of the server.
             *    eg: consumer protocol = dubbo, provider only has other
protocol services(rest).
             * 2. The registration center is not robust and pushes
illegal specification data.
             *
             */
            if (CollectionUtils.isEmptyMap(newUrlInvokerMap)) {
                logger.error(new IllegalStateException("urls to invokers
error .invokerUrls.size :" + invokerUrls.size() + ", invoker.size :0.
urls :" + invokerUrls
                        .toString()));
                return;
            }

            List<Invoker<T>> newInvokers =
Collections.unmodifiableList(new
ArrayList<>(newUrlInvokerMap.values()));
            this.setInvokers(multiGroup ? new
BitList<>(toMergeInvokerList(newInvokers)) : new
BitList<>(newInvokers));
            // pre-route and build cache
            routerChain.setInvokers(this.getInvokers());
            this.urlInvokerMap = newUrlInvokerMap;

            try {
                destroyUnusedInvokers(oldUrlInvokerMap,
newUrlInvokerMap); // Close the unused Invoker
            } catch (Exception e) {
                logger.warn("destroyUnusedInvokers error. ", e);
            }

            // notify invokers refreshed
            this.invokersChanged();
```

```
        }
    }
```

将 invokerURL 列表转换为调用器映射。转换规则如下：

- 如果 URL 已转换为 invoker，它将不再被重新引用并直接从缓存中获取，请注意，URL 中的任何参数更改都将被重新引用。

- 如果传入调用器列表不为空，则表示它是最新的调用器列表。

- 如果传入 invokerUrl 的列表为空，则意味着该规则只是一个覆盖规则或路由规则，需要重新对比以决定是否重新引用。

RegistryDirectory 类型的 toInvokers 协议过滤，配置合并都会在这里面去做

将 url 转换为调用程序，如果 url 已被引用，则不会重新引用。将放入 newUrlInvokeMap 的项将从 OldUrlInvoceMap 中删除。

```
private Map<URL, Invoker<T>> toInvokers(Map<URL, Invoker<T>>
oldUrlInvokerMap, List<URL> urls) {
        Map<URL, Invoker<T>> newUrlInvokerMap = new
ConcurrentHashMap<>(urls == null ? 1 : (int) (urls.size() / 0.75f +
1));
        if (urls == null || urls.isEmpty()) {
            return newUrlInvokerMap;
        }
        //这个配置怎么来呢，这个配置是就是要过滤的协议，如果我们指定了当前接口的协议
比如 dubbo.reference.<interface>.protocol 这样的指定协议配置就可以在这里过滤出
合法的协议
        String queryProtocols = this.queryMap.get(PROTOCOL_KEY);
        //遍历所有提供者列表进行转换
        for (URL providerUrl : urls) {
            // If protocol is configured at the reference side, only the
matching protocol is selected
            if (queryProtocols != null && queryProtocols.length() > 0) {
                boolean accept = false;
                String[] acceptProtocols = queryProtocols.split(",");
                for (String acceptProtocol : acceptProtocols) {
```

```
                if (providerUrl.getProtocol().equals(acceptProtocol))
{
                    accept = true;
                    break;
                }
            }
            if (!accept) {
                continue;
            }
        }
        //空协议 直接跳过
        if (EMPTY_PROTOCOL.equals(providerUrl.getProtocol())) {
            continue;
        }
        //查询扩展是否存在，这个要注意如果是自定义扩展或者像 webservice 这样的
扩展一般需要额外引入扩展包才可以
        if
(!getUrl().getOrDefaultFrameworkModel().getExtensionLoader(Protocol.cl
ass).hasExtension(providerUrl.getProtocol())) {
            logger.error(new IllegalStateException("Unsupported
protocol " + providerUrl.getProtocol() +
                " in notified url: " + providerUrl + " from registry "
+ getUrl().getAddress() +
                " to consumer " + NetUtils.getLocalHost() + ",
supported protocol: " +

getUrl().getOrDefaultFrameworkModel().getExtensionLoader(Protocol.clas
s).getSupportedExtensions()));
            continue;
        }
        //这个合并很重要决定了一个服务的配置优先级
        //合并 url 参数。顺序是：覆盖 override 协议配置（比如禁用服务有时候就这
样做）> -D(JVM 参数) > 消费者 > 提供者
        URL url = mergeUrl(providerUrl);

        // Cache key is url that does not merge with consumer side
parameters, regardless of how the consumer combines parameters, if the
server url changes, then refer again
        Invoker<T> invoker = oldUrlInvokerMap == null ? null :
oldUrlInvokerMap.remove(url);
        if (invoker == null) { // Not in the cache, refer again
            try {
                boolean enabled = true;
```

```
                if (url.hasParameter(DISABLED_KEY)) {
                    enabled = !url.getParameter(DISABLED_KEY, false);
                } else {
                    enabled = url.getParameter(ENABLED_KEY, true);
                }
                if (enabled) {
                    //消费者引用服务提供者
                    invoker = protocol.refer(serviceType, url);
                }
            } catch (Throwable t) {
                logger.error("Failed to refer invoker for interface:"
+ serviceType + ",url:(" + url + ")" + t.getMessage(), t);
            }
            if (invoker != null) { // Put new invoker in cache
                newUrlInvokerMap.put(url, invoker);
            }
        } else {
            newUrlInvokerMap.put(url, invoker);
        }
    }
    return newUrlInvokerMap;
}
```

上面代码有一个很重要的一行,是消费者引用服务提供者的代码如下所示：这行代码将会执行服务引用逻辑：

```
if (enabled) {
      //消费者引用服务提供者
      invoker = protocol.refer(serviceType, url);
  }
```

前面这个代码会经过协议调用链(wrapper 机制的 aop)逻辑,默认使用的是 Dubbo 协议，这里我们就不详细展开了直接定位到 DubboProtocol 来看。

DubboProtocol 类型的 refer 方法：

```
@Override
    public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
        checkDestroyed();
        return protocolBindingRefer(type, url);
    }
```

DubboProtocol 类型的 protocolBindingRefer 方法：

```
@Override
    public <T> Invoker<T> protocolBindingRefer(Class<T> serviceType, URL url) throws
RpcException {
        checkDestroyed();
        //optimizer配置  序列化优化配置
        optimizeSerialization(url);

        // create rpc invoker
        //DubboInvoker调用器对象创建 这里要重点关注的方法是getClients.
        DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url),
invokers);
        invokers.add(invoker);

        return invoker;
    }
```

注意这里分了两步：

- 获取网络客户端
- 封装为 DubboInvoker

DubboProtocol 类型的 getClients 方法：

```
    private ExchangeClient[] getClients(URL url) {
        // whether to share connection

        boolean useShareConnect = false;

        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) ?
ConfigurationUtils.getProperty(url.getOrDefaultApplicationModel(), 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) {
                clients[i] = shareClients.get(i);

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

        return clients;
    }
```

DubboProtocol 类型的 getSharedClient 方法：

```
@SuppressWarnings("unchecked")
   private List<ReferenceCountExchangeClient> getSharedClient(URL url,
int connectNum) {
       String key = url.getAddress();

       Object clients = referenceClientMap.get(key);
       if (clients instanceof List) {
           List<ReferenceCountExchangeClient> typedClients =
(List<ReferenceCountExchangeClient>) clients;
           if (checkClientCanUse(typedClients)) {
               batchClientRefIncr(typedClients);
               return typedClients;
           }
       }
```

```
        List<ReferenceCountExchangeClient> typedClients = null;

        synchronized (referenceClientMap) {
            for (; ; ) {
                clients = referenceClientMap.get(key);

                if (clients instanceof List) {
                    typedClients = (List<ReferenceCountExchangeClient>)
clients;
                    if (checkClientCanUse(typedClients)) {
                        batchClientRefIncr(typedClients);
                        return typedClients;
                    } else {
                        referenceClientMap.put(key, PENDING_OBJECT);
                        break;
                    }
                } else if (clients == PENDING_OBJECT) {
                    try {
                        referenceClientMap.wait();
                    } catch (InterruptedException ignored) {
                    }
                } else {
                    referenceClientMap.put(key, PENDING_OBJECT);
                    break;
                }
            }
        }

        try {
            // connectNum must be greater than or equal to 1
            //长连接数量 默认为 1
            connectNum = Math.max(connectNum, 1);

            // If the clients is empty, then the first initialization is
            if (CollectionUtils.isEmpty(typedClients)) {
                //！！！！主要看这一行 构建客户端
                typedClients = buildReferenceCountExchangeClientList(url,
connectNum);
            } else {
                for (int i = 0; i < typedClients.size(); i++) {
                    ReferenceCountExchangeClient
referenceCountExchangeClient = typedClients.get(i);
```

```
                // If there is a client in the list that is no longer
available, create a new one to replace him.
                if (referenceCountExchangeClient == null ||
referenceCountExchangeClient.isClosed()) {
                    typedClients.set(i,
buildReferenceCountExchangeClient(url));
                    continue;
                }
                referenceCountExchangeClient.incrementAndGetCount();
            }
        }
    } finally {
        synchronized (referenceClientMap) {
            if (typedClients == null) {
                referenceClientMap.remove(key);
            } else {

                //这里 key 位 IP:端口 值为当前客户端 一个 IP:端口的服务提供者会
缓存一个连接列表
                referenceClientMap.put(key, typedClients);
            }
            referenceClientMap.notifyAll();
        }
    }
    return typedClients;

}
```

DubboProtocol 类型的 buildReferenceCountExchangeClientList 方法：创建网络交换器客户端

```
 private List<ReferenceCountExchangeClient> buildReferenceCountExchangeClientList(URL url,
int connectNum) {
        List<ReferenceCountExchangeClient> clients = new ArrayList<>();

        for (int i = 0; i < connectNum; i++) {
            clients.add(buildReferenceCountExchangeClient(url));
        }

        return clients;
    }
```

DubboProtocol 类型的 buildReferenceCountExchangeClient 方法

```
 private ReferenceCountExchangeClient buildReferenceCountExchangeClient(URL url) {
     //初始化客户端对象
        ExchangeClient exchangeClient = initClient(url);
        //包装交换器客户端对象
        ReferenceCountExchangeClient client = new
ReferenceCountExchangeClient(exchangeClient, DubboCodec.NAME);
        // read configs
        int shutdownTimeout =
ConfigurationUtils.getServerShutdownTimeout(url.getScopeModel());
        client.setShutdownWaitTime(shutdownTimeout);
        return client;
    }
```

DubboProtocol 类型的 initClient 方法：

```
  /**
     * Create new connection
     *
     * @param url
     */
    private ExchangeClient initClient(URL url) {

        /**
         * Instance of url is InstanceAddressURL, so addParameter
actually adds parameters into ServiceInstance,
         * which means params are shared among different services. Since
client is shared among services this is currently not a problem.
         */
        String str = url.getParameter(CLIENT_KEY,
url.getParameter(SERVER_KEY, DEFAULT_REMOTING_CLIENT));

        // BIO is not allowed since it has severe performance issue.
        if (StringUtils.isNotEmpty(str)
&& !url.getOrDefaultFrameworkModel().getExtensionLoader(Transporter.cl
ass).hasExtension(str)) {
            throw new RpcException("Unsupported client type: " + str +
"," +
                    " supported client type is " +
StringUtils.join(url.getOrDefaultFrameworkModel().getExtensionLoader(T
ransporter.class).getSupportedExtensions(), " "));
        }

        ExchangeClient client;
```

```
        try {
            // Replace InstanceAddressURL with ServiceConfigURL.
            url = new ServiceConfigURL(DubboCodec.NAME,
url.getUsername(), url.getPassword(), url.getHost(), url.getPort(),
url.getPath(),  url.getAllParameters());
            url = url.addParameter(CODEC_KEY, DubboCodec.NAME);
            // enable heartbeat by default
            url = url.addParameterIfAbsent(HEARTBEAT_KEY,
String.valueOf(DEFAULT_HEARTBEAT));

            // connection should be lazy
            if (url.getParameter(LAZY_CONNECT_KEY, false)) {
                client = new LazyConnectExchangeClient(url,
requestHandler);
            } else {
                client = Exchangers.connect(url, requestHandler);
            }

        } catch (RemotingException e) {
            throw new RpcException("Fail to create remoting client for
service(" + url + "): " + e.getMessage(), e);
        }

        return client;
    }
```

Exchangers 类型的 connect 方法：

```
 public static ExchangeClient connect(URL url, ExchangeHandler handler) throws
RemotingException {
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }
        if (handler == null) {
            throw new IllegalArgumentException("handler == null");
        }
        //交换器扩展对象获取默认为HeaderExchanger类型
        return getExchanger(url).connect(url, handler);
    }
```

HeaderExchanger 类型的 connect 方法：

```
  @Override
    public ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException
{
        return new HeaderExchangeClient(Transporters.connect(url, new DecodeHandler(new
HeaderExchangeHandler(handler))), true);
    }
```

HeaderExchangeHandler 对象创建

DecodeHandler 处理器对象创建

Transporters 的 connect 方法

```
 public static Client connect(URL url, ChannelHandler... handlers) throws RemotingException
{
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }
        ChannelHandler handler;
        if (handlers == null || handlers.length == 0) {
            handler = new ChannelHandlerAdapter();
        } else if (handlers.length == 1) {
            handler = handlers[0];
        } else {
            handler = new ChannelHandlerDispatcher(handlers);
        }
        //默认的传输器是netty4
        return getTransporter(url).connect(url, handler);
    }
```

org.apache.dubbo.remoting.transport.netty4.NettyTransporter 的 connect 方法：

```
@Override
    public Client connect(URL url, ChannelHandler handler) throws RemotingException {
        return new NettyClient(url, handler);
    }
```

org.apache.dubbo.remoting.transport.netty4.NettyClient 对象创建

```
  public NettyClient(final URL url, final ChannelHandler handler) throws RemotingException {
        // you can customize name and type of client thread pool by THREAD_NAME_KEY and
THREADPOOL_KEY in CommonConstants.
        // the handler will be wrapped: MultiMessageHandler->HeartbeatHandler->handler
        super(url, wrapChannelHandler(url, handler));
    }
```

### HeartbeatHandler MultiMessageHandler 创建

### AbstractClient 构造器

```
public AbstractClient(URL url, ChannelHandler handler) throws
RemotingException {
        super(url, handler);
        // set default needReconnect true when channel is not connected
        needReconnect = url.getParameter(Constants.SEND_RECONNECT_KEY,
true);
        //初始化线程池 默认为 FixedThreadPool
        initExecutor(url);

        try {
            //启动 netty 的核心代码初始化 Bootstrap
            //默认走的是 NioSocketChannel
            //初始化默认连接超时时间为 3 秒
            doOpen();
        } catch (Throwable t) {
            close();
            throw new RemotingException(url.toInetSocketAddress(), null,
                "Failed to start " + getClass().getSimpleName() + " " +
NetUtils.getLocalAddress()
                    + " connect to the server " + getRemoteAddress() + ",
cause: " + t.getMessage(), t);
        }

        try {
            // connect.
            //前面是初始化 netty 的客户端启动类 Bootstrap 这里是执行连接的代码:
bootstrap.connect(getConnectAddress());
            //等待 3 秒连接失败则抛出异常
            connect();
            if (logger.isInfoEnabled()) {
```

```
            logger.info("Start " + getClass().getSimpleName() + " " +
NetUtils.getLocalAddress() + " connect to the server " +
getRemoteAddress());
         }
      } catch (RemotingException t) {
         // If lazy connect client fails to establish a connection,
the client instance will still be created,
         // and the reconnection will be initiated by ReconnectTask,
so there is no need to throw an exception
         if (url.getParameter(LAZY_CONNECT_KEY, false)) {
            logger.warn("Failed to start " +
getClass().getSimpleName() + " " + NetUtils.getLocalAddress() +
                  " connect to the server " + getRemoteAddress() +
                  " (the connection request is initiated by lazy connect
client, ignore and retry later!), cause: " +
                  t.getMessage(), t);
            return;
         }
         //默认必须连接，无法连接则抛出异常
         if (url.getParameter(Constants.CHECK_KEY, true)) {
            close();
            throw t;
         } else {
            logger.warn("Failed to start " +
getClass().getSimpleName() + " " + NetUtils.getLocalAddress()
                  + " connect to the server " + getRemoteAddress() + "
(check == false, ignore and retry later!), cause: " + t.getMessage(),
t);
         }
      } catch (Throwable t) {
         close();
         throw new RemotingException(url.toInetSocketAddress(), null,
            "Failed to start " + getClass().getSimpleName() + " " +
NetUtils.getLocalAddress()
                  + " connect to the server " + getRemoteAddress() + ",
cause: " + t.getMessage(), t);
      }
   }
```

### AbstractEndpoint 构造器

```
 public AbstractEndpoint(URL url, ChannelHandler handler) {
        super(url, handler);
        //DubboCodec是实现RPC调用的Request和Response对象的编码和解码类，RPC调用实现的核心传输也就是这
两个类对象。
        //这里是封装了DubboCodec类型的DubboCountCodec
        this.codec = getChannelCodec(url);
        //默认连接超时时间为3000毫秒
        this.connectTimeout = url.getPositiveParameter(Constants.CONNECT_TIMEOUT_KEY,
Constants.DEFAULT_CONNECT_TIMEOUT);
    }
```

### AbstractPeer 构造器创建

```
public AbstractPeer(URL url, ChannelHandler handler) {
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }
        if (handler == null) {
            throw new IllegalArgumentException("handler == null");
        }
        this.url = url;
        this.handler = handler;
    }
```

# 四、 应用级服务发现源码解析

```
title: 23-【Dubbo3.0.8源码解析系列】Dubbo应用级服务发现
date: 2022-07-10 20:25:23
author: songxiaosheng
img: https://dubbo.apache.org/imgs/architecture.png
top: false
hide: false
cover: false
# coverImg: /images/1.jpg
password: 8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
toc: true
mathjax: false
summary: 24-Dubbo应用级服务发现
categories: Dubbo3源码
tags:
  - Dubbo
  - Dubbo3
  -  Dubbo3源码解析
```

## 1. 简介

这里我们要看的是 MigrationInvoker 类型的 refreshServiceDiscoveryInvoker 方法：根据应用级服务发现，创建应用级的服务调用器，这里有很多逻辑和接口级服务发现类似，不过与接口级调用的 invoker 对象不同的是应用级的粒度会比较大一些这一步不会去注意去订阅各个服务接口，只会订阅服务提供者的应用。

默认情况下如果没有配置强制走接口级或者应用级的服务配置，接口级逻辑和应用级服务订阅都会走，这里我们可以直接来看代码吧：

MigrationInvoker 类型的 refreshServiceDiscoveryInvoker 方法

## 2. 刷新服务发现调用器 Invoker

下面这个入口代码和接口级的 Invoker 对象创建类似，唯一不同的是接口级 Invoker 对象的创建调用的是 registryProtocol 对象（InterfaceCompatibleRegistryProtocol 类型）的 getInvoker 方法，而这里调用了 getServiceDiscoveryInvoker

MigrationInvoker 类型的 refreshServiceDiscoveryInvoker 方法代码如下：

```
  protected void refreshServiceDiscoveryInvoker(CountDownLatch latch) {
        clearListener(serviceDiscoveryInvoker);
        if (needRefresh(serviceDiscoveryInvoker)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Re-subscribing instance addresses, current interface " +
type.getName());
            }

            if (serviceDiscoveryInvoker != null) {
                serviceDiscoveryInvoker.destroy();
            }
            //registryProtocol类型为：InterfaceCompatibleRegistryProtocol
            serviceDiscoveryInvoker = registryProtocol.getServiceDiscoveryInvoker(cluster,
registry, type, url);
        }
        setListener(serviceDiscoveryInvoker, () -> {
            latch.countDown();
            if (reportService.hasReporter()) {
                reportService.reportConsumptionStatus(
                    reportService.createConsumptionReport(consumerUrl.getServiceInterface(),
consumerUrl.getVersion(), consumerUrl.getGroup(), "app"));
            }
            if (step == APPLICATION_FIRST) {
                calcPreferredInvoker(rule);
            }
        });
    }
```

InterfaceCompatibleRegistryProtocol 类型的 getServiceDiscoveryInvoker 方法

```
  public <T> ClusterInvoker<T> getServiceDiscoveryInvoker(Cluster cluster, Registry
registry, Class<T> type, URL url) {
        //注册中心Registry类型对象获取 ，
        //这里获取到的是ListenerRegistryWrapper 类型，其中包装了ServiceDiscoveryRegistry类型
        registry = getRegistry(super.getRegistryUrl(url));
        //服务发现注册目录对象创建  这里具体逻辑就不说了
        DynamicDirectory<T> directory = new ServiceDiscoveryRegistryDirectory<>(type, url);
        //开始创建invoker
        return doCreateInvoker(directory, cluster, registry, type);
    }
```

调用 InterfaceCompatibleRegistryProtocol 类型的父类型 RegistryProtocol 的 doCreateInvoker 方法。

下面这个代码接口级服务发现逻辑我中我们已经说过了代码是一样的 RegistryProtocol 的 doCreateInvoker 方法。

大部分逻辑是一样的，唯一有两个对象是不同的：

- **应用级**：registry 类型服务发现的类型 ServiceDiscoveryRegistry 接口级为：ZookeeperRegistry 类型。

- **应用级**：ServiceDiscoveryRegistryDirectory 接口级为：RegistryDirectory 类型。

```
protected <T> ClusterInvoker<T> doCreateInvoker(DynamicDirectory<T> directory, Cluster
cluster, Registry registry, Class<T> type) {
        directory.setRegistry(registry);
        directory.setProtocol(protocol);
        // all attributes of REFER_KEY
        Map<String, String> parameters = new HashMap<>
(directory.getConsumerUrl().getParameters());
        URL urlToRegistry = new ServiceConfigURL(
            parameters.get(PROTOCOL_KEY) == null ? CONSUMER : parameters.get(PROTOCOL_KEY),
            parameters.remove(REGISTER_IP_KEY),
            0,
            getPath(parameters, type),
            parameters
        );
        urlToRegistry =
urlToRegistry.setScopeModel(directory.getConsumerUrl().getScopeModel());
        urlToRegistry =
urlToRegistry.setServiceModel(directory.getConsumerUrl().getServiceModel());
        if (directory.isShouldRegister()) {
            directory.setRegisteredConsumerUrl(urlToRegistry);
            //这一行逻辑会走到ServiceDiscoveryRegistry  不过应用级消费者是无需注册服务数据的
            registry.register(directory.getRegisteredConsumerUrl());
        }
        directory.buildRouterChain(urlToRegistry);

        //发起订阅
        directory.subscribe(toSubscribeUrl(urlToRegistry));

        return (ClusterInvoker<T>) cluster.join(directory, true);
    }
```

ServiceDiscoveryRegistry 类型的 subscribe 方法：

```
  public void subscribe(URL url) {
        if (moduleModel.getModelEnvironment().getConfiguration().convert(Boolean.class,
Constants.ENABLE_CONFIGURATION_LISTEN, true)) {
            enableConfigurationListen = true;
            //为ConsumerConfigurationListener类型中的listeners列表添加监听器：  监听器类型为
ServiceDiscoveryRegistryDirectory
            getConsumerConfigurationListener(moduleModel).addNotifyListener(this);
            referenceConfigurationListener = new
ReferenceConfigurationListener(this.moduleModel, this, url);
        } else {
            enableConfigurationListen = false;
        }
        //调用父类类型DynamicDirectory的订阅方法subscribe 开始开启订阅逻辑 这个逻辑与接口级的逻辑是一样
的
        super.subscribe(url);
    }
```

```
//调用父类类型DynamicDirectory的订阅方法subscribe 开始开启订阅逻辑 这个逻辑与接口级的逻辑是一样的
    super.subscribe(url);
```

为了理解起来更直观我重复贴一下代码来重新看一遍：

DynamicDirectory 的订阅方法 subscribe 方法如下：

```
    public void subscribe(URL url) {
        setSubscribeUrl(url);
        //这里先走ListenerRegistryWrapper的subscribe逻辑再走包装的ServiceDiscoveryRegistry类型的
逻辑
        registry.subscribe(url, this);
    }
```

接下来的 subscribe 会先走 ListenerRegistryWrapper 的 subscribe 逻辑再走包装的 ServiceDiscoveryRegistry 类型的逻辑。

接下来直接看下核心的一些代码：

ListenerRegistryWrapper 类型的 subscribe 方法

```
public void subscribe(URL url, NotifyListener listener) {
        try {
            if (registry != null) {
                registry.subscribe(url, listener);
            }
        } finally {
            if (CollectionUtils.isNotEmpty(listeners)) {
                RuntimeException exception = null;
                for (RegistryServiceListener registryListener : listeners) {
                    if (registryListener != null) {
                        try {
                            registryListener.onSubscribe(url, registry);
                        } catch (RuntimeException t) {
                            logger.error(t.getMessage(), t);
                            exception = t;
                        }
                    }
                }
                if (exception != null) {
                    throw exception;
                }
            }
        }
    }
```

ServiceDiscoveryRegistry 类型的 subscribe 方法：

```
public final void subscribe(URL url, NotifyListener listener) {
       //前面是否注册shouldRegister为false这里是是否订阅shouldSubscribe方法结果为true
       if (!shouldSubscribe(url)) { // Should Not Subscribe
           return;
       }
       //执行订阅逻辑
       doSubscribe(url, listener);
   }
```

从这里开始要进入应用级服务订阅的路基了继续往下看。

ServiceDiscoveryRegistry 类型的 doSubscribe 方法：

```
public void doSubscribe(URL url, NotifyListener listener) {
       url = addRegistryClusterKey(url);
       //服务发现类型为 ZookeeperServiceDiscovery
       serviceDiscovery.subscribe(url, listener);

       boolean check = url.getParameter(CHECK_KEY, false);

       String key = ServiceNameMapping.buildMappingKey(url);
       //应用级服务发现悲观锁先加上一把
       Lock mappingLock = serviceNameMapping.getMappingLock(key);
       try {
           mappingLock.lock();
           Set<String> subscribedServices =
serviceNameMapping.getCachedMapping(url);
           try {
               MappingListener mappingListener = new
DefaultMappingListener(url, subscribedServices, listener);
               //注意注意这行代码超级重要 当前是服务接口要找到服务的应用名字 将会查
询映射信息对应节点：
               ///dubbo/mapping/link.elastic.dubbo.entity.DemoService
               //这里最终获取到的应用服务提供者名字集合为 dubbo-demo-api-
provider
               subscribedServices =
serviceNameMapping.getAndListen(this.getUrl(), url, mappingListener);
               mappingListeners.put(url.getProtocolServiceKey(),
mappingListener);
           } catch (Exception e) {
```

```
            logger.warn("Cannot find app mapping for service " +
url.getServiceInterface() + ", will not migrate.", e);
        }

        if (CollectionUtils.isEmpty(subscribedServices)) {
            logger.info("No interface-apps mapping found in local
cache, stop subscribing, will automatically wait for mapping listener
callback: " + url);
//              if (check) {
//                  throw new IllegalStateException("Should has at least
one way to know which services this interface belongs to, subscription
url: " + url);
//              }
            return;
        }
        //执行订阅 url 的逻辑
        subscribeURLs(url, listener, subscribedServices);
    } finally {
        mappingLock.unlock();
    }
}
```

接下来看

服务发现类型为 ZookeeperServiceDiscovery 的 subscribe 方法，不过这里封装在父类型里面了，调用其父类型 AbstractServiceDiscovery 的 subscribe 方法。

先看父类型 AbstractServiceDiscovery 的 subscribe 方法

```
public void subscribe(URL url, NotifyListener listener) {
  //这个metadataInfo类型为MetadataInfo
      metadataInfo.addSubscribedURL(url);
  }
```

MetadataInfo 类型的 addSubscribedURL 方法：

```
public synchronized void addSubscribedURL(URL url) {
       if (subscribedServiceURLs == null) {
           subscribedServiceURLs = new ConcurrentSkipListMap<>();
       }
       //下面将其url添加到subscribedServiceURLs成员变量里面就结束了
       addURL(subscribedServiceURLs, url);
   }
```

前面服务发现的 subscribe 并没有做什么逻辑性质的操作仅仅是将 url 放到了成员变量里面，接下来继续看 ServiceDiscoveryRegistry 类型的 doSubscribe 方法：

通过服务接口信息查询应用名字对应注册中心路径为：dubbo/mapping/link.elastic.dubbo.entity.DemoService

对应代码 MetadataServiceNameMapping 类型的 getAndListen 方法：

下面这个代码比较长，我们重点看一行代码

```
mappingServices = (new AsyncMappingTask(listener, subscribedURL, false)).call();
```

```
public Set<String> getAndListen(URL registryURL, URL subscribedURL,
MappingListener listener) {
       String key = ServiceNameMapping.buildMappingKey(subscribedURL);
       // use previously cached services.
       Set<String> mappingServices = this.getCachedMapping(key);

       // Asynchronously register listener in case previous cache does
not exist or cache expired.
       if (CollectionUtils.isEmpty(mappingServices)) {
           try {
               logger.info("Local cache mapping is empty");
               //重点看这个同步调用获取注册中心的路径信息 call 方法在父类型
AbstractServiceNameMapping 中
               mappingServices = (new AsyncMappingTask(listener,
subscribedURL, false)).call();
           } catch (Exception e) {
               // ignore
           }
           if (CollectionUtils.isEmpty(mappingServices)) {
```

```
            String registryServices =
registryURL.getParameter(SUBSCRIBED_SERVICE_NAMES_KEY);
            if (StringUtils.isNotEmpty(registryServices)) {
                logger.info(subscribedURL.getServiceInterface() + "
mapping to " + registryServices + " instructed by registry subscribed-
services.");
                mappingServices = parseServices(registryServices);
            }
        }
        if (CollectionUtils.isNotEmpty(mappingServices)) {
            this.putCachedMapping(key, mappingServices);
        }
    } else {
        ExecutorService executorService =
applicationModel.getFrameworkModel().getBeanFactory()
            .getBean(FrameworkExecutorRepository.class).getMappingRef
reshingExecutor();
        executorService.submit(new AsyncMappingTask(listener,
subscribedURL, true));
    }

    return mappingServices;
}
```

接下来看 MetadataServiceNameMapping 类型的父类型 AbstractServiceNameMapping 的 call 方法。

同样道理这里主要关注 getAndListen 方法即可

```
public Set<String> call() throws Exception {
        synchronized (mappingListeners) {
            Set<String> mappedServices = emptySet();
            try {
                //这个缓存的 key 与服务接口和分组有关这里我没配置分组那就只有接口
了 key 为 link.elastic.dubbo.entity.DemoService
                String mappingKey =
ServiceNameMapping.buildMappingKey(subscribedURL);
                if (listener != null) {
                    //这里获取到的应用名字为：dubbo-demo-api-provider
                    mappedServices =
toTreeSet(getAndListen(subscribedURL, listener));
```

```
                    Set<MappingListener> listeners =
mappingListeners.computeIfAbsent(mappingKey, _k -> new HashSet<>());
                    listeners.add(listener);
                    if (CollectionUtils.isNotEmpty(mappedServices)) {
                        if (notifyAtFirstTime) {
                            // guarantee at-least-once notification no
matter what kind of underlying meta server is used.
                            // listener notification will also cause
updating of mapping cache.
                            listener.onEvent(new
MappingChangedEvent(mappingKey, mappedServices));
                        }
                    }
                } else {
                    mappedServices = get(subscribedURL);
                    if (CollectionUtils.isNotEmpty(mappedServices)) {

AbstractServiceNameMapping.this.putCachedMapping(mappingKey,
mappedServices);
                    }
                }
            } catch (Exception e) {
                logger.error("Failed getting mapping info from remote
center. ", e);
            }
            return mappedServices;
        }
    }
}
```

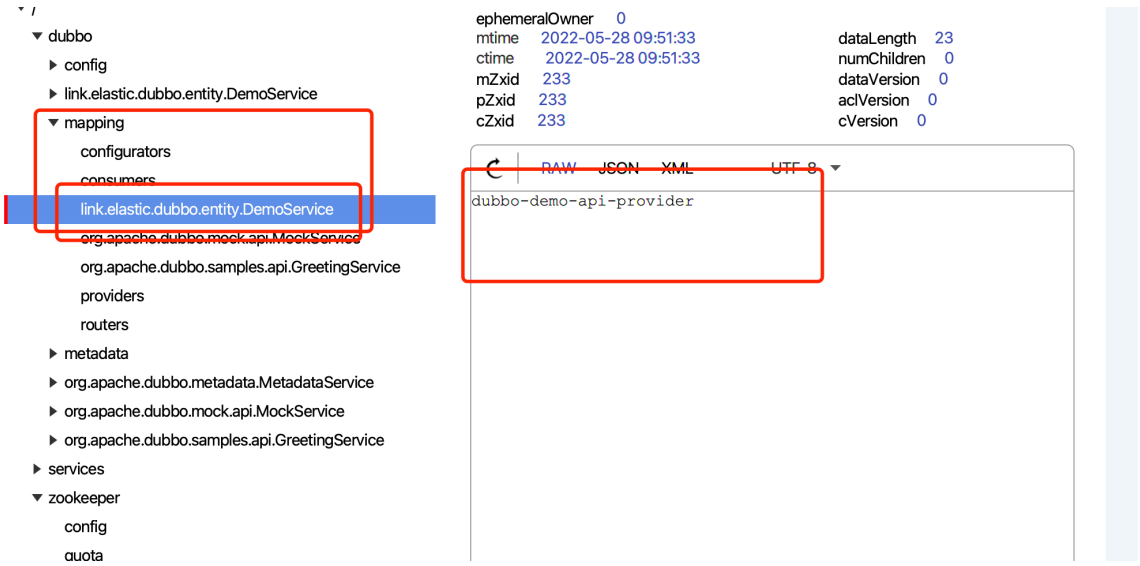
然后看 AbstractServiceNameMapping 的 getAndListen 方法

```
public Set<String> getAndListen(URL url, MappingListener mappingListener) {
        String serviceInterface = url.getServiceInterface();
        // randomly pick one metadata report is ok for it's guaranteed all metadata report
will have the same mapping data.
        String registryCluster = getRegistryCluster(url);
        MetadataReport metadataReport =
metadataReportInstance.getMetadataReport(registryCluster);
        if (metadataReport == null) {
            return Collections.emptySet();
        }
        return metadataReport.getServiceAppMapping(serviceInterface, mappingListener, url);
    }
```

ZookeeperMetadataReport 类型的 getServiceAppMapping 方法：

```
 public Set<String> getServiceAppMapping(String serviceKey, MappingListener listener, URL
url) {
        String path = buildPathKey(DEFAULT_MAPPING_GROUP, serviceKey);
        MappingDataListener mappingDataListener = casListenerMap.computeIfAbsent(path, _k ->
{
            MappingDataListener newMappingListener = new MappingDataListener(serviceKey,
path);
            zkClient.addDataListener(path, newMappingListener);
            return newMappingListener;
        });
        mappingDataListener.addListener(listener);
        //这个拼装后的路径为：/dubbo/mapping/link.elastic.dubbo.entity.DemoService
        //这里获取到的应用名字为：dubbo-demo-api-provider
        return getAppNames(zkClient.getContent(path));
    }
```

这里贴个图展示下映射数据：




有了应用名字开始订阅服务提供者 ServiceDiscoveryRegistry 类型的 subscribeURLs 方法。

代码如下所示：

```
protected void subscribeURLs(URL url, NotifyListener listener,
Set<String> serviceNames) {
       serviceNames = toTreeSet(serviceNames);
```

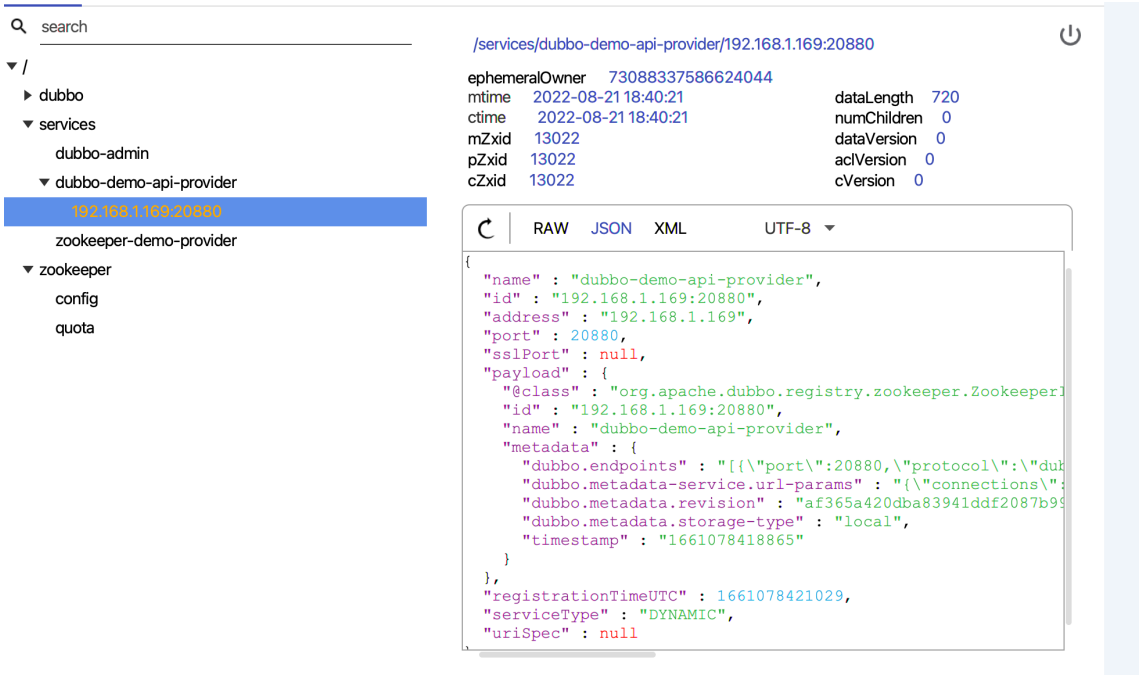
```
        String serviceNamesKey = toStringKeys(serviceNames);
        String protocolServiceKey = url.getProtocolServiceKey();
        logger.info(String.format("Trying to subscribe from apps %s for
service key %s, ", serviceNamesKey, protocolServiceKey));

        // register ServiceInstancesChangedListener
        Lock appSubscriptionLock = getAppSubscription(serviceNamesKey);
        try {
            //再来一把 url 订阅的悲观锁
            appSubscriptionLock.lock();
            ServiceInstancesChangedListener
serviceInstancesChangedListener =
serviceListeners.get(serviceNamesKey);
            if (serviceInstancesChangedListener == null) {
                serviceInstancesChangedListener =
serviceDiscovery.createListener(serviceNames);
                serviceInstancesChangedListener.setUrl(url);
                //这个应用名字为：dubbo-demo-api-provider
                for (String serviceName : serviceNames) {
                    //这个代码调用的是 curator 框架的方法 通过应用名字节点查询节点
下面的所有服务提供者的应用信息待会截图看
                    List<ServiceInstance> serviceInstances =
serviceDiscovery.getInstances(serviceName);
                    if (CollectionUtils.isNotEmpty(serviceInstances)) {
                        //发现了存在服务提供者则触发监听器开始进行应用发现通知
                        serviceInstancesChangedListener.onEvent(new
ServiceInstancesChangedEvent(serviceName, serviceInstances));
                    }
                }
                serviceListeners.put(serviceNamesKey,
serviceInstancesChangedListener);
            }

            if (!serviceInstancesChangedListener.isDestroyed()) {
                serviceInstancesChangedListener.setUrl(url);

listener.addServiceListener(serviceInstancesChangedListener);

serviceInstancesChangedListener.addListenerAndNotify(protocolServiceKe
y, listener);

serviceDiscovery.addServiceInstancesChangedListener(serviceInstancesCh
angedListener);
```

```
            } else {
                logger.info(String.format("Listener of %s has been
destroyed by another thread.", serviceNamesKey));
                serviceListeners.remove(serviceNamesKey);
            }
        } finally {
            appSubscriptionLock.unlock();
        }
    }
```

查询到的应用信息：

```
DefaultServiceInstance{
    serviceName='dubbo-demo-api-provider',
    host='192.168.1.169', port=20880,
    enabled=true, healthy=true,
    metadata={
        dubbo.endpoints=[{"port":20880,"protocol":"dubbo"}],
        dubbo.metadata-service.url-params=
{"connections":"1","version":"1.0.0","dubbo":"2.0.2","release":"3.0.10","side":"provider","p
ort":"20880","protocol":"dubbo"},
        dubbo.metadata.revision=af365a420dba83941ddf2087b998a1d2,
        dubbo.metadata.storage-type=local, timestamp=1661078418865}}
```

这里展示图注册中心的数据：



ServiceInstancesChangedListener 类型的时间通知方法 onEvent

```
public void onEvent(ServiceInstancesChangedEvent event) {
       if (destroyed.get() || !accept(event) || isRetryAndExpired(event)) {
           return;
       }
       doOnEvent(event);
   }
```

继续看 ServiceInstancesChangedListener 类型的时间通知方法的 doOnEvent

```
private synchronized void doOnEvent(ServiceInstancesChangedEvent
event) {
       if (destroyed.get() || !accept(event) ||
isRetryAndExpired(event)) {
           return;
       }
       //刷新内存数据
       refreshInstance(event);

       if (logger.isDebugEnabled()) {
           logger.debug(event.getServiceInstances().toString());
       }

       Map<String, List<ServiceInstance>> revisionToInstances = new
HashMap<>();
       Map<String, Map<String, Set<String>>> localServiceToRevisions =
new HashMap<>();

       // grouping all instances of this app(service name) by revision
         //刷新内存数据
       for (Map.Entry<String, List<ServiceInstance>> entry :
allInstances.entrySet()) {
           List<ServiceInstance> instances = entry.getValue();
           for (ServiceInstance instance : instances) {
               String revision = getExportedServicesRevision(instance);
               if (revision == null || EMPTY_REVISION.equals(revision))
{
                   if (logger.isDebugEnabled()) {
                       logger.debug("Find instance without valid service
metadata: " + instance.getAddress());
                   }
```

```
                continue;
            }
            List<ServiceInstance> subInstances =
revisionToInstances.computeIfAbsent(revision, r -> new
LinkedList<>());
            subInstances.add(instance);
        }
    }

    // get MetadataInfo with revision
    //重点看这里
    for (Map.Entry<String, List<ServiceInstance>> entry :
revisionToInstances.entrySet()) {
        String revision = entry.getKey();
        List<ServiceInstance> subInstances = entry.getValue();
        //这里对应 ZookeeperServiceDiscovery 类型
        MetadataInfo metadata =
serviceDiscovery.getRemoteMetadata(revision, subInstances);
        //解析元数据 最终结果存在 localServiceToRevisions 变量中 key 为协议
值为服务接口与服务元数据信息
        parseMetadata(revision, metadata, localServiceToRevisions);
        // update metadata into each instance, in case new instance
created.
        //为每个实例更新其元数据信息
        for (ServiceInstance tmpInstance : subInstances) {
            MetadataInfo originMetadata =
tmpInstance.getServiceMetadata();
            if (originMetadata == null
|| !Objects.equals(originMetadata.getRevision(),
metadata.getRevision())) {
                tmpInstance.setServiceMetadata(metadata);
            }
        }
    }

    int emptyNum = hasEmptyMetadata(revisionToInstances);
    if (emptyNum != 0) {// retry every 10 seconds
        hasEmptyMetadata = true;
        if (retryPermission.tryAcquire()) {
            if (retryFuture != null && !retryFuture.isDone()) {
                // cancel last retryFuture because only one
retryFuture will be canceled at destroy().
                retryFuture.cancel(true);
```

```
            }
            retryFuture = scheduler.schedule(new
AddressRefreshRetryTask(retryPermission, event.getServiceName()),
10_000L, TimeUnit.MILLISECONDS);
            logger.warn("Address refresh try task submitted");
         }
         // return if all metadata is empty, this notification will
not take effect.
         if (emptyNum == revisionToInstances.size()) {
            logger.error("Address refresh failed because of Metadata
Server failure, wait for retry or new address refresh event.");
            return;
         }
      }
      hasEmptyMetadata = false;

      Map<String, Map<Set<String>, Object>> protocolRevisionsToUrls =
new HashMap<>();
      Map<String, Object> newServiceUrls = new HashMap<>();
      for (Map.Entry<String, Map<String, Set<String>>> entry :
localServiceToRevisions.entrySet()) {
         String protocol = entry.getKey();
         entry.getValue().forEach((protocolServiceKey, revisions) ->
{
            Map<Set<String>, Object> revisionsToUrls =
protocolRevisionsToUrls.computeIfAbsent(protocol, k -> new
HashMap<>());
            Object urls = revisionsToUrls.get(revisions);
            if (urls == null) {
               urls = getServiceUrlsCache(revisionToInstances,
revisions, protocol);
               revisionsToUrls.put(revisions, urls);
            }

            newServiceUrls.put(protocolServiceKey, urls);
         });
      }

      this.serviceUrls = newServiceUrls;
      this.notifyAddressChanged();
   }
```

ZookeeperServiceDiscovery 类型的 getRemoteMetadata 方法

```
public MetadataInfo getRemoteMetadata(String revision,
List<ServiceInstance> instances) {
       MetadataInfo metadata = metaCacheManager.get(revision);
       //缓存的元数据为空将从元数据中心远端获取
       if (metadata != null && metadata != MetadataInfo.EMPTY) {
           metadata.init();
           // metadata loaded from cache
           if (logger.isDebugEnabled()) {
               logger.debug("MetadataInfo for revision=" + revision + ",
" + metadata);
           }
           return metadata;
       }

      //这里将从元数据中心获取数据
       synchronized (metaCacheManager) {
           // try to load metadata from remote.
           int triedTimes = 0;
           //失败则重试 3 次
           while (triedTimes < 3) {
               //根据版本号查询元数据 发起一次 RPC 请求
               metadata = MetadataUtils.getRemoteMetadata(revision,
instances, metadataReport);

               if (metadata != MetadataInfo.EMPTY) {// succeeded
                //前面 RPC 请求元数据成功接下来开始初始化
                   metadata.init();
                   break;
               } else {// failed
                   if (triedTimes > 0) {
                       if (logger.isDebugEnabled()) {
                           logger.debug("Retry the " + triedTimes + "
times to get metadata for revision=" + revision);
                       }
                   }
                   triedTimes++;
                   try {
                       Thread.sleep(1000);
                   } catch (InterruptedException e) {
                   }
```

```
            }
        }

        if (metadata == MetadataInfo.EMPTY) {
            logger.error("Failed to get metadata for revision after 3
retries, revision=" + revision);
        } else {
            //缓存查询到的元数据到元数据缓存管理器中
            metaCacheManager.put(revision, metadata);
        }
    }
    return metadata;
}
```

MetadataUtils 类型的 getRemoteMetadata 方法：

```
public static MetadataInfo getRemoteMetadata(String revision,
List<ServiceInstance> instances, MetadataReport metadataReport) {
    //随机轮训一台主机查询它的应用实例信息
    ServiceInstance instance = selectInstance(instances);
    //元数据提供者存储的位置默认为本地存储 local 消费者从提供者那里拿，
    //remote - Provider 把 metadata 放到远端注册中心，Consumer 从注册中心
获取；
    //local - Provider 把 metadata 放在本地，Consumer 从 Provider 处直接
获取；
    //这个配置可以看链接：https://dubbo.apache.org/zh/docs3-v2/java-
sdk/reference-manual/config/properties/
    String metadataType =
ServiceInstanceMetadataUtils.getMetadataStorageType(instance);
    MetadataInfo metadataInfo;
    try {
        if (logger.isDebugEnabled()) {
            logger.debug("Instance " + instance.getAddress() + " is
using metadata type " + metadataType);
        }

        //remote - Provider 把 metadata 放到远端注册中心，Consumer 从注册
中心获取；
        //local - Provider 把 metadata 放在本地，Consumer 从 Provider 处
直接获取；
        if (REMOTE_METADATA_STORAGE_TYPE.equals(metadataType)) {
            //这里走的是 remote 配置
```

```
            metadataInfo = MetadataUtils.getMetadata(revision,
instance, metadataReport);
         } else {
            // change the instance used to communicate to avoid all
requests route to the same instance
            ProxyHolder proxyHolder = null;
            try {
               //手动调用服务提供者内置的 MetadataService Dubbo 服务
               proxyHolder = MetadataUtils.referProxy(instance);
               //发起 RPC 调用 调用提供者的提供的元数据请求 RPC 接口
               metadataInfo =
proxyHolder.getProxy().getMetadataInfo(ServiceInstanceMetadataUtils.ge
tExportedServicesRevision(instance));
            } finally {
               MetadataUtils.destroyProxy(proxyHolder);
            }
         }
      } catch (Exception e) {
         logger.error("Failed to get app metadata for revision " +
revision + " for type " + metadataType + " from instance " +
instance.getAddress(), e);
         metadataInfo = null;
      }

      if (metadataInfo == null) {
         metadataInfo = MetadataInfo.EMPTY;
      }
      return metadataInfo;
   }
```

## MetadataUtils 类型的 referProxy

```
public static ProxyHolder referProxy(ServiceInstance instance) {
      MetadataServiceURLBuilder builder;
      ExtensionLoader<MetadataServiceURLBuilder> loader =
instance.getApplicationModel()
         .getExtensionLoader(MetadataServiceURLBuilder.class);

      Map<String, String> metadata = instance.getMetadata();
      // METADATA_SERVICE_URLS_PROPERTY_NAME is a unique key exists
only on instances of spring-cloud-alibaba.
```

```
        String dubboUrlsForJson =
metadata.get(METADATA_SERVICE_URLS_PROPERTY_NAME);
        //
        if (metadata.isEmpty() || StringUtils.isEmpty(dubboUrlsForJson))
{
            builder =
loader.getExtension(StandardMetadataServiceURLBuilder.NAME);
        } else {
            builder =
loader.getExtension(SpringCloudMetadataServiceURLBuilder.NAME);
        }
        //默认的 builder 类型为 StandardMetadataServiceURLBuilder 将远数据对象
转 url 配置
        //例如:
dubbo://192.168.1.169:20880/org.apache.dubbo.metadata.MetadataService?
connections=1&corethreads=2&dubbo=2.0.2&group=dubbo-demo-api-
provider&port=20880&protocol=dubbo&release=3.0.10&retries=0&side=provi
der&threadpool=cached&threads=100&timeout=5000&version=1.0.0
        List<URL> urls = builder.build(instance);
        if (CollectionUtils.isEmpty(urls)) {
            throw new IllegalStateException("Introspection service
discovery mode is enabled "
                + instance + ", but no metadata service can build from
it.");
        }

        URL url = urls.get(0);

        // Simply rely on the first metadata url, as stated in
MetadataServiceURLBuilder.
        ApplicationModel applicationModel =
instance.getApplicationModel();
        ModuleModel internalModel =
applicationModel.getInternalModule();
        ConsumerModel consumerModel =
applicationModel.getInternalModule().registerInternalConsumer(Metadata
Service.class, url);

        Protocol protocol =
applicationModel.getExtensionLoader(Protocol.class).getAdaptiveExtensi
on();

        url.setServiceModel(consumerModel);
```

```
        //!!!! 重点看这一行与普通的服务一样这里默认也是使用 DubboProtocol 来引用元数据服务
        Invoker<MetadataService> invoker = protocol.refer(MetadataService.class, url);

        ProxyFactory proxyFactory = applicationModel.getExtensionLoader(ProxyFactory.class).getAdaptiveExtension();

        //为其将要调用的 invoker 生成对应代理对象
        MetadataService metadataService = proxyFactory.getProxy(invoker);

        consumerModel.getServiceMetadata().setTarget(metadataService);
        consumerModel.getServiceMetadata().addAttribute(PROXY_CLASS_REF, metadataService);
        consumerModel.setProxyObject(metadataService);
        consumerModel.initMethodModels();

        return new ProxyHolder(consumerModel, metadataService, internalModel);
    }
```

metadata 的 init 方法初始化从提供者那里获取到的元数据

```
 public void init() {
        if (!initiated.compareAndSet(false, true)) {
            return;
        }
        if (CollectionUtils.isNotEmptyMap(services)) {
            //遍历所有的服务信息然后初始化 ServiceInfo
            services.forEach((_k, serviceInfo) -> {
                serviceInfo.init();
                // create duplicate serviceKey(without protocol)->serviceInfo mapping to support metadata search when protocol is not specified on consumer side.
                if (subscribedServices == null) {
                    subscribedServices = new HashMap<>();
                }
                Set<ServiceInfo> serviceInfos = subscribedServices.computeIfAbsent(serviceInfo.getServiceKey(), _key -> new HashSet<>());
                serviceInfos.add(serviceInfo);
            });
        }
    }
```

ServiceInfo 类型的 init 方法

```
protected void init() {
            //初始化matchKey变量 格式为: service + group + version + protocol
            buildMatchKey();
            //初始化服务keyserviceKey 格式为: service + group + version
            buildServiceKey(name, group, version);
            // init method params
            //初始化与方法匹配的参数
            this.methodParams = URLParam.initMethodParameters(params);
            // Actually, consumer params is empty after deserialized on the consumer side,
so no need to initialize.
            // Check how InstanceAddressURL operates on consumer url for more detail.
//            this.consumerMethodParams = URLParam.initMethodParameters(consumerParams);
            // no need to init numbers for it's only for cache purpose
        }
```

ServiceDiscoveryRegistryDirectory 接收订阅到的服务的通知方法:

```
 public synchronized void notify(List<URL> instanceUrls) {
        if (isDestroyed()) {
            return;
        }
        // Set the context of the address notification thread.
        RpcServiceContext.getServiceContext().setConsumerUrl(getConsumerUrl());

        //  3.x added for extend URL address
        ExtensionLoader<AddressListener> addressListenerExtensionLoader =
getUrl().getOrDefaultModuleModel().getExtensionLoader(AddressListener.class);
        List<AddressListener> supportedListeners =
addressListenerExtensionLoader.getActivateExtension(getUrl(), (String[]) null);
        if (supportedListeners != null && !supportedListeners.isEmpty()) {
            for (AddressListener addressListener : supportedListeners) {
                instanceUrls = addressListener.notify(instanceUrls, getConsumerUrl(), this);
            }
        }

        refreshOverrideAndInvoker(instanceUrls);
    }
```

```
  private synchronized void refreshOverrideAndInvoker(List<URL> instanceUrls) {
        // mock zookeeper://xxx?mock=return null
        refreshInvoker(instanceUrls);
    }
```

```
private void refreshInvoker(List<URL> invokerUrls) {
       Assert.notNull(invokerUrls, "invokerUrls should not be null, use
EMPTY url to clear current addresses.");
       this.originalUrls = invokerUrls;
```

```
        if (invokerUrls.size() == 1 &&
EMPTY_PROTOCOL.equals(invokerUrls.get(0).getProtocol())) {
            logger.warn("Received url with EMPTY protocol, will clear
all available addresses.");
            this.forbidden = true; // Forbid to access
            routerChain.setInvokers(BitList.emptyList());
            destroyAllInvokers(); // Close all invokers
        } else {
            this.forbidden = false; // Allow accessing
            if (CollectionUtils.isEmpty(invokerUrls)) {
                logger.warn("Received empty url list, will ignore for
protection purpose.");
                return;
            }

            // use local reference to avoid NPE as this.urlInvokerMap
will be set null concurrently at destroyAllInvokers().
            Map<String, Invoker<T>> localUrlInvokerMap =
this.urlInvokerMap;
            // can't use local reference as oldUrlInvokerMap's mappings
might be removed directly at toInvokers().
            Map<String, Invoker<T>> oldUrlInvokerMap = null;
            if (localUrlInvokerMap != null) {
                // the initial capacity should be set greater than the
maximum number of entries divided by the load factor to avoid
resizing.
                oldUrlInvokerMap = new LinkedHashMap<>(Math.round(1 +
localUrlInvokerMap.size() / DEFAULT_HASHMAP_LOAD_FACTOR));
                localUrlInvokerMap.forEach(oldUrlInvokerMap::put);
            }
            //主要这一行做一些协议的过滤与实例禁用数据的过滤得到最终结果需要的调用器
            Map<String, Invoker<T>> newUrlInvokerMap =
toInvokers(oldUrlInvokerMap, invokerUrls);// Translate url list to
Invoker map
            logger.info("Refreshed invoker size " +
newUrlInvokerMap.size());

            if (CollectionUtils.isEmptyMap(newUrlInvokerMap)) {
                logger.error(new IllegalStateException("Cannot create
invokers from url address list (total " + invokerUrls.size() + ")"));
                return;
            }
```

```
            List<Invoker<T>> newInvokers =
Collections.unmodifiableList(new
ArrayList<>(newUrlInvokerMap.values()));
            this.setInvokers(multiGroup ? new
BitList<>(toMergeInvokerList(newInvokers)) : new
BitList<>(newInvokers));
            // pre-route and build cache
            routerChain.setInvokers(this.getInvokers());
            this.urlInvokerMap = newUrlInvokerMap;

            if (oldUrlInvokerMap != null) {
                try {
                    destroyUnusedInvokers(oldUrlInvokerMap,
newUrlInvokerMap); // Close the unused Invoker
                } catch (Exception e) {
                    logger.warn("destroyUnusedInvokers error. ", e);
                }
            }
        }

        // notify invokers refreshed
        this.invokersChanged();
    }
```

```
private Map<String, Invoker<T>> toInvokers(Map<String, Invoker<T>>
oldUrlInvokerMap, List<URL> urls) {
        Map<String, Invoker<T>> newUrlInvokerMap = new
ConcurrentHashMap<>(urls == null ? 1 : (int) (urls.size() / 0.75f +
1));
        if (urls == null || urls.isEmpty()) {
            return newUrlInvokerMap;
        }
        for (URL url : urls) {
            InstanceAddressURL instanceAddressURL = (InstanceAddressURL)
url;
            if (EMPTY_PROTOCOL.equals(instanceAddressURL.getProtocol()))
{
                continue;
            }
            if
(!getUrl().getOrDefaultFrameworkModel().getExtensionLoader(Protocol.cl
ass).hasExtension(instanceAddressURL.getProtocol())) {
```

```
                logger.error(new IllegalStateException("Unsupported
protocol " + instanceAddressURL.getProtocol() +
                    " in notified url: " + instanceAddressURL + " from
registry " + getUrl().getAddress() +
                    " to consumer " + NetUtils.getLocalHost() + ",
supported protocol: " +

getUrl().getOrDefaultFrameworkModel().getExtensionLoader(Protocol.clas
s).getSupportedExtensions()));
                continue;
            }


instanceAddressURL.setProviderFirstParams(providerFirstParams);

            // Override provider urls if needed
            if (enableConfigurationListen) {
                instanceAddressURL =
overrideWithConfigurator(instanceAddressURL);
            }

            Invoker<T> invoker = oldUrlInvokerMap == null ? null :
oldUrlInvokerMap.get(instanceAddressURL.getAddress());
            if (invoker == null || urlChanged(invoker,
instanceAddressURL)) { // Not in the cache, refer again
                try {
                    boolean enabled = true;
                    if (instanceAddressURL.hasParameter(DISABLED_KEY)) {
                        enabled
= !instanceAddressURL.getParameter(DISABLED_KEY, false);
                    } else {
                        enabled =
instanceAddressURL.getParameter(ENABLED_KEY, true);
                    }
                    if (enabled) {
                        invoker = protocol.refer(serviceType,
instanceAddressURL);
                    }
                } catch (Throwable t) {
                    logger.error("Failed to refer invoker for interface:"
+ serviceType + ",url:(" + instanceAddressURL + ")" + t.getMessage(),
t);
                }
```

```
                if (invoker != null) { // Put new invoker in cache
                    newUrlInvokerMap.put(instanceAddressURL.getAddress(),
invoker);
                }
            } else {
                newUrlInvokerMap.put(instanceAddressURL.getAddress(),
invoker);
                oldUrlInvokerMap.remove(instanceAddressURL.getAddress(),
invoker);
            }
        }
        return newUrlInvokerMap;
    }
```

# RPC 调用过程解析

## 一、 集群容错与过滤器

```
title: 25-【Dubbo3.0.8源码解析系列】为调用器对象增加容错和过滤器功能
date: 2022-07-10 20:25:23
author: songxiaosheng
img: https://dubbo.apache.org/imgs/architecture.png
top: false
hide: false
cover: false
# coverImg: /images/1.jpg
password: 8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
toc: true
mathjax: false
summary: 25-为调用器对象增加容错和过滤器功能
categories: Dubbo3源码
tags:
  - Dubbo
  - Dubbo3
  -  Dubbo3源码解析
```

将思路拉回到 RegistryProtocol 的创建 Invoker 对象的 doCreateInvoker 代码

```
 protected <T> ClusterInvoker<T> doCreateInvoker(DynamicDirectory<T> directory, Cluster
cluster, Registry registry, Class<T> type) {
        directory.setRegistry(registry);
        directory.setProtocol(protocol);
        // all attributes of REFER_KEY
        Map<String, String> parameters = new HashMap<>
(directory.getConsumerUrl().getParameters());
        URL urlToRegistry = new ServiceConfigURL(
            parameters.get(PROTOCOL_KEY) == null ? CONSUMER : parameters.get(PROTOCOL_KEY),
            parameters.remove(REGISTER_IP_KEY),
            0,
            getPath(parameters, type),
            parameters
        );
        urlToRegistry =
urlToRegistry.setScopeModel(directory.getConsumerUrl().getScopeModel());
        urlToRegistry =
urlToRegistry.setServiceModel(directory.getConsumerUrl().getServiceModel());
        if (directory.isShouldRegister()) {
            directory.setRegisteredConsumerUrl(urlToRegistry);
            registry.register(directory.getRegisteredConsumerUrl());
        }
        directory.buildRouterChain(urlToRegistry);
        directory.subscribe(toSubscribeUrl(urlToRegistry));

        return (ClusterInvoker<T>) cluster.join(directory, true);
    }
```

不论是接口级还是应用级注册都会调用代码

```
return (ClusterInvoker<T>) cluster.join(directory, true);
```

我们详细看下：MockClusterWrapper 类型的 join 方法

buildFilterChain 参数为 true

```
  public <T> Invoker<T> join(Directory<T> directory, boolean buildFilterChain) throws
RpcException {
        return new MockClusterInvoker<T>(directory,
                this.cluster.join(directory, buildFilterChain));
    }
```

默认的 cluster 类型为 FailoverCluster，我们看 FailoverCluster 的 join 方法

FailoverCluster 没有实现 join 方法需要先调用它的父类型 AbstractCluster 的 join 方法如下：

```
   public <T> Invoker<T> join(Directory<T> directory, boolean buildFilterChain) throws
RpcException {
        //带有过滤器将走上面这个逻辑
        if (buildFilterChain) {
            return buildClusterInterceptors(doJoin(directory));
        } else {
            return doJoin(directory);
        }
    }
```

//过滤器参数为 true 将走上面这个逻辑 先创建 Invoker 对象再创建过滤器

```
        if (buildFilterChain) {
            return buildClusterInterceptors(doJoin(directory));
```

doJoin 方法将有默认类 AbstractCluster 的 doJoin 抽象方法调用具体方法 FailoverCluster 的 doJoin 方法如下。

FailoverCluster 的 doJoin 方法

```
public <T> AbstractClusterInvoker<T> doJoin(Directory<T> directory) throws RpcException {
       return new FailoverClusterInvoker<>(directory);
   }
```

可以看到这里调用链路创建了一个失效转移的 Invoker 对象 FailoverClusterInvoker

FailoverClusterInvoker 的构造器

```
 public FailoverClusterInvoker(Directory<T> directory) {
       super(directory);
   }
```

父调用器 AbstractClusterInvoker 的构造器

```
 public AbstractClusterInvoker(Directory<T> directory) {
       this(directory, directory.getUrl());

}
```

重载的构造器如下：

```
   public AbstractClusterInvoker(Directory<T> directory, URL url) {
       if (directory == null) {
           throw new IllegalArgumentException("service directory == null");
       }

       this.directory = directory;
       //sticky: invoker.isAvailable() should always be checked before using when
availablecheck is true.
       this.availableCheck = url.getParameter(CLUSTER_AVAILABLE_CHECK_KEY,
DEFAULT_CLUSTER_AVAILABLE_CHECK);
       Configuration configuration =
ConfigurationUtils.getGlobalConfiguration(url.getOrDefaultModuleModel());
       this.reselectCount = configuration.getInt(RESELECT_COUNT, DEFAULT_RESELECT_COUNT);
       this.enableConnectivityValidation =
configuration.getBoolean(ENABLE_CONNECTIVITY_VALIDATION, true);
   }
```

初始化过滤器链路的代码 AbstractCluster 类型的 buildClusterInterceptors 方法如下：

```
  private <T> Invoker<T> buildClusterInterceptors(AbstractClusterInvoker<T> clusterInvoker)
{
//        AbstractClusterInvoker<T> last = clusterInvoker;
        AbstractClusterInvoker<T> last = buildInterceptorInvoker(new ClusterFilterInvoker<>
(clusterInvoker));

        if
(Boolean.parseBoolean(ConfigurationUtils.getProperty(clusterInvoker.getDirectory().getConsum
erUrl().getScopeModel(), CLUSTER_INTERCEPTOR_COMPATIBLE_KEY, "false"))) {
            return build27xCompatibleClusterInterceptors(clusterInvoker, last);
        }
        return last;
    }
```

ClusterFilterInvoker 构造器

```
  public ClusterFilterInvoker(AbstractClusterInvoker<T> invoker) {
            //过滤器构造链DefaultFilterChainBuilder
            List<FilterChainBuilder> builders =
ScopeModelUtil.getApplicationModel(invoker.getUrl().getScopeModel()).getExtensionLoader(Filt
erChainBuilder.class).getActivateExtensions();
            if (CollectionUtils.isEmpty(builders)) {
                filterInvoker = invoker;
            } else {
                ClusterInvoker<T> tmpInvoker = invoker;
                for (FilterChainBuilder builder : builders) {
                    tmpInvoker = builder.buildClusterInvokerChain(tmpInvoker,
REFERENCE_FILTER_KEY, CommonConstants.CONSUMER);
                }
                filterInvoker = tmpInvoker;
            }
        }
```

DefaultFilterChainBuilder 类型的 buildClusterInvokerChain 构造过滤器链路

```java
public <T> ClusterInvoker<T> buildClusterInvokerChain(final
ClusterInvoker<T> originalInvoker, String key, String group) {
    ClusterInvoker<T> last = originalInvoker;
    URL url = originalInvoker.getUrl();
    List<ModuleModel> moduleModels = getModuleModelsFromUrl(url);
    List<ClusterFilter> filters;
    if (moduleModels != null && moduleModels.size() == 1) {
        //通过扩展查询匹配的消费者过滤器列表这里可以查询 4 个
```

```
        filters = ScopeModelUtil.getExtensionLoader(ClusterFilter.class,
moduleModels.get(0)).getActivateExtension(url, key, group);
    } else if (moduleModels != null && moduleModels.size() > 1) {
        filters = new ArrayList<>();
        List<ExtensionDirector> directors = new ArrayList<>();
        for (ModuleModel moduleModel : moduleModels) {
            List<ClusterFilter> tempFilters =
ScopeModelUtil.getExtensionLoader(ClusterFilter.class,
moduleModel).getActivateExtension(url, key, group);
            filters.addAll(tempFilters);
            directors.add(moduleModel.getExtensionDirector());
        }
        filters = sortingAndDeduplication(filters, directors);

    } else {
        filters = ScopeModelUtil.getExtensionLoader(ClusterFilter.class,
null).getActivateExtension(url, key, group);
    }
    //过滤器不为空则拼接到调用链表之中
    if (!CollectionUtils.isEmpty(filters)) {
        for (int i = filters.size() - 1; i >= 0; i--) {
            final ClusterFilter filter = filters.get(i);
            final Invoker<T> next = last;
            last = new CopyOfClusterFilterChainNode<>(originalInvoker,
next, filter);
        }
        return new ClusterCallbackRegistrationInvoker<>(originalInvoker,
last, filters);
    }

    return last;
}
```

默认为4个过滤器：

- ConsumerContextFilter
- FutureFilter
- MonitorClusterFilter
- RouterSnapshotFilter

# 二、 RPC 调用过程解析

```
title: 26-【Dubbo3.0.8源码解析系列】RPC调用过程
date: 2022-07-10 20:25:23
author: songxiaosheng
img: https://dubbo.apache.org/imgs/architecture.png
top: false
hide: false
cover: false
# coverImg: /images/1.jpg
password: 8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92
toc: true
mathjax: false
summary: 26-RPC调用过程
categories: Dubbo3源码
tags:
  - Dubbo
  - Dubbo3
  -  Dubbo3源码解析
```

这个就要回到消费者的调用代码了主要代码如下。

下面是我写的消费者样例代码中的一块代码：

```
//我们这个DemoService是个接口实际的对象是由Dubbo内部通过动态代理的方式创建的一个对象类型为
DemoServiceDubboProxy1
 DemoService demoService = bootstrap.getCache().get(reference);
 //消费者调用提供者的代码
        String message = demoService.sayHello("dubbo");
```

DemoServiceDubboProxy1 中的 sayHello 代理方法我们看不到这里直接看代理方法调用的方法：如下 InvokerInvocationHandler 类型的 invoke

参数：

- proxy DemoServiceDubboProxy1
- method sayHello 方法对应的 Java 反射元数据 Method
- args 这里 args 为参数，我们只有一个参数值为 dubbo

```
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        if (method.getDeclaringClass() == Object.class) {
            return method.invoke(invoker, args);
```

```
        }
        //方法名字为 sayHello
        String methodName = method.getName();
        //参数类型只有一个为 java.lang.String
        Class<?>[] parameterTypes = method.getParameterTypes();
        if (parameterTypes.length == 0) {
            if ("toString".equals(methodName)) {
                return invoker.toString();
            } else if ("$destroy".equals(methodName)) {
                invoker.destroy();
                return null;
            } else if ("hashCode".equals(methodName)) {
                return invoker.hashCode();
            }
        } else if (parameterTypes.length == 1 &&
"equals".equals(methodName)) {
            return invoker.equals(args[0]);
        }
        //这个 RpcInvocation 比较重要主要是封装调用方法的一些元数据信息
        RpcInvocation rpcInvocation = new RpcInvocation(serviceModel,
method.getName(), invoker.getInterface().getName(),
protocolServiceKey, method.getParameterTypes(), args);

        if (serviceModel instanceof ConsumerModel) {
            rpcInvocation.put(Constants.CONSUMER_MODEL, serviceModel);
            rpcInvocation.put(Constants.METHOD_MODEL, ((ConsumerModel)
serviceModel).getMethodModel(method));
        }
        //invoker 为调用器我们继续看
        return InvocationUtil.invoke(invoker, rpcInvocation);
    }
```

继续看方法 `InvocationUtil.invoke(invoker, rpcInvocation)`;

InvocationUtil 的 invoke 方法

```
public static Object invoke(Invoker<?> invoker, RpcInvocation
rpcInvocation) throws Throwable {
    //url 例子:
consumer://192.168.1.169/link.elastic.dubbo.entity.DemoService?applica
tion=dubbo-demo-api-
```

```
consumer&background=false&dubbo=2.0.2&interface=link.elastic.dubbo.ent
ity.DemoService&methods=sayHello,sayHelloAsync&pid=99368&qos.enable=fa
lse&qos.port=-
1&register.ip=192.168.1.169&release=3.0.10&side=consumer&sticky=false&
timestamp=1661595732778
       URL url = invoker.getUrl();
       //这里无分组值为：link.elastic.dubbo.entity.DemoService
       String serviceKey = url.getServiceKey();
       rpcInvocation.setTargetServiceUniqueName(serviceKey);

       // invoker.getUrl() returns consumer url.
       RpcServiceContext.getServiceContext().setConsumerUrl(url);

       //默认前后开启性能分析
       if (ProfilerSwitch.isEnableSimpleProfiler()) {
          //创建一个 InternalThreadLocal<ProfilerEntry> bizProfiler 线程本
地对象来存储性能信息
          //首次进入这个性能实体为空
          ProfilerEntry parentProfiler = Profiler.getBizProfiler();
          ProfilerEntry bizProfiler;

          //首次为空走下面逻辑创建个性能分析实体
          //这里如果是第二个调用 invoker 方法则将性能数据串起来前面的放到 parent
ProfilerEntry 内部用链表结构实现一个性能链路
          if (parentProfiler != null) {
             bizProfiler = Profiler.enter(parentProfiler,
                "Receive request. Client invoke begin. ServiceKey: " +
serviceKey + " MethodName:" + rpcInvocation.getMethodName());
          } else {
             bizProfiler = Profiler.start("Receive request. Client
invoke begin. ServiceKey: " + serviceKey + " " + "MethodName:" +
rpcInvocation.getMethodName());
          }
          rpcInvocation.put(Profiler.PROFILER_KEY, bizProfiler);
          try {
             //第一个 invoker 类型为 MigrationInvoker
             return invoker.invoke(rpcInvocation).recreate();
          } finally {
             Profiler.release(bizProfiler);
             int timeout;
             Object timeoutKey =
rpcInvocation.getObjectAttachmentWithoutConvert(TIMEOUT_KEY);
             if (timeoutKey instanceof Integer) {
```

```
                timeout = (Integer) timeoutKey;
            } else {
                timeout =
url.getMethodPositiveParameter(rpcInvocation.getMethodName(),
                    TIMEOUT_KEY,
                    DEFAULT_TIMEOUT);
            }
            long usage = bizProfiler.getEndTime() -
bizProfiler.getStartTime();
            if ((usage / (1000_000L *
ProfilerSwitch.getWarnPercent())) > timeout) {
                StringBuilder attachment = new StringBuilder();
                rpcInvocation.foreachAttachment((entry) -> {

attachment.append(entry.getKey()).append("=").append(entry.getValue())
.append(";\n");
                });

                logger.warn(String.format(
                    "[Dubbo-Consumer] execute service %s#%s
cost %d.%06d ms, this invocation almost (maybe already) timeout.
Timeout: %dms\n" + "invocation context:\n%s" + "thread info: \n%s",
                    rpcInvocation.getProtocolServiceKey(),
                    rpcInvocation.getMethodName(),
                    usage / 1000_000,
                    usage % 1000_000,
                    timeout,
                    attachment,
                    Profiler.buildDetail(bizProfiler)));
            }
        }
    }
    return invoker.invoke(rpcInvocation).recreate();
}
```

MigrationInvoker 类型的

其实前面我们已经说过是如何决策应用级还是接口级的调用默认走应用级下面直接看应用级相关的 invoker 链路

```
 public Result invoke(Invocation invocation) throws RpcException {
        if (currentAvailableInvoker != null) {
            if (step == APPLICATION_FIRST) {
                // call ratio calculation based on random value
                if (promotion < 100 && ThreadLocalRandom.current().nextDouble(100) >
promotion) {
                    // fall back to interface mode
                    return invoker.invoke(invocation);
                }
                // check if invoker available for each time

                return decideInvoker().invoke(invocation);
            }
            return currentAvailableInvoker.invoke(invocation);
        }

        switch (step) {
            case APPLICATION_FIRST:
                currentAvailableInvoker = decideInvoker();
                break;
            case FORCE_APPLICATION:
                currentAvailableInvoker = serviceDiscoveryInvoker;
                break;
            case FORCE_INTERFACE:
            default:
                currentAvailableInvoker = invoker;
        }

        return currentAvailableInvoker.invoke(invocation);
    }
```

接下来要走的 Invoker 逻辑是带有容错逻辑的 MockClusterInvoker 的 invoker

MockClusterInvoker 类型的 invoker

```
MockClusterInvoker 类型的 invoker
    ```java
    public Result invoke(Invocation invocation) throws RpcException {
        Result result;
        //判断是否开启服务容错逻辑 mock 默认是没有开启的
        String value =
getUrl().getMethodParameter(invocation.getMethodName(), MOCK_KEY,
Boolean.FALSE.toString()).trim();
        if (ConfigUtils.isEmpty(value)) {
            //no mock
            result = this.invoker.invoke(invocation);
        } else if (value.startsWith(FORCE_KEY)) {
            if (logger.isWarnEnabled()) {
```
```

```
            logger.warn("force-mock: " + invocation.getMethodName() +
" force-mock enabled , url : " + getUrl());
         }
         //force:direct mock
         result = doMockInvoke(invocation, null);
      } else {
         //fail-mock
         try {
            result = this.invoker.invoke(invocation);
  //fix:#4585
         if (result.getException() != null && result.getException()
instanceof RpcException) {
            RpcException rpcException = (RpcException)
result.getException();
            if (rpcException.isBiz()) {
               throw rpcException;
            } else {
               result = doMockInvoke(invocation, rpcException);
            }
         }

      } catch (RpcException e) {
         if (e.isBiz()) {
            throw e;
         }

         if (logger.isWarnEnabled()) {
            logger.warn("fail-mock: " + invocation.getMethodName() +
" fail-mock enabled , url : " + getUrl(), e);
         }
         result = doMockInvoke(invocation, e);
      }
   }
   return result;
}
````
```

接下来要走的逻辑是默认类 AbstractCluster 的 invoke 方法

```
````java
 public Result invoke(Invocation invocation) throws RpcException {
      return filterInvoker.invoke(invocation);
```

```
    }
```

接下来要走的逻辑是带有回调通知的链路，CallbackRegistrationInvoker 类型的 invoke 方法

```
```java
 public Result invoke(Invocation invocation) throws RpcException {
          Result asyncResult = filterInvoker.invoke(invocation);
          asyncResult.whenCompleteWithContext((r, t) -> {
              for (int i = filters.size() - 1; i >= 0; i--) {
                  FILTER filter = filters.get(i);
                  try {

InvocationProfilerUtils.releaseDetailProfiler(invocation);
                      if (filter instanceof ListenableFilter) {
                          ListenableFilter listenableFilter =
((ListenableFilter) filter);
                          Filter.Listener listener =
listenableFilter.listener(invocation);
                          try {
                              if (listener != null) {
                                  if (t == null) {
                                      listener.onResponse(r, filterInvoker,
invocation);
                                  } else {
                                      listener.onError(t, filterInvoker,
invocation);
                                  }
                              }
                          } finally {
                              listenableFilter.removeListener(invocation);
                          }
                      } else if (filter instanceof FILTER.Listener) {
                          FILTER.Listener listener = (FILTER.Listener)
filter;
                          if (t == null) {
                              listener.onResponse(r, filterInvoker,
invocation);
                          } else {
                              listener.onError(t, filterInvoker,
invocation);
                          }
```
```

```
                    }
                } catch (Throwable filterThrowable) {
                    LOGGER.error(String.format("Exception occurred
while executing the %s filter named %s.", i,
filter.getClass().getSimpleName()));
                    if (LOGGER.isDebugEnabled()) {
                        LOGGER.debug(String.format("Whole filter list
is: %s", filters.stream().map(tmpFilter ->
tmpFilter.getClass().getSimpleName()).collect(Collectors.toList())));
                    }
                    throw filterThrowable;
                }
            }
        });

        return asyncResult;
    }
```

接下来要走的是过滤器的链路，CopyOfFilterChainNode 类型的 invoke 方法

```
```java
  public Result invoke(Invocation invocation) throws RpcException {
      Result asyncResult;
      try {
          InvocationProfilerUtils.enterDetailProfiler(invocation, ()
-> "Filter " + filter.getClass().getName() + " invoke.");
          asyncResult = filter.invoke(nextNode, invocation);
      } catch (Exception e) {
          InvocationProfilerUtils.releaseDetailProfiler(invocation);
          if (filter instanceof ListenableFilter) {
              ListenableFilter listenableFilter = ((ListenableFilter)
filter);
              try {
                  Filter.Listener listener =
listenableFilter.listener(invocation);
                  if (listener != null) {
                      listener.onError(e, originalInvoker, invocation);
                  }
              } finally {
                  listenableFilter.removeListener(invocation);
              }
          } else if (filter instanceof FILTER.Listener) {
```
```

```
                FILTER.Listener listener = (FILTER.Listener) filter;
                listener.onError(e, originalInvoker, invocation);
            }
            throw e;
        } finally {

        }
        return asyncResult;
    }
    ````

    然后走第一个过滤器 ConsumerContextFilter
    ```java
    public Result invoke(Invoker<?> invoker, Invocation invocation)
throws RpcException {
    RpcContext.RestoreServiceContext originServiceContext =
RpcContext.storeServiceContext();
    try {
        RpcContext.getServiceContext()
            .setInvoker(invoker)
            .setInvocation(invocation)
            .setLocalAddress(NetUtils.getLocalHost(), 0);

        RpcContext context = RpcContext.getClientAttachment();
        context.setAttachment(REMOTE_APPLICATION_KEY,
invoker.getUrl().getApplication());
        if (invocation instanceof RpcInvocation) {
            ((RpcInvocation) invocation).setInvoker(invoker);
        }

        if (CollectionUtils.isNotEmpty(supportedSelectors)) {
            for (PenetrateAttachmentSelector supportedSelector :
supportedSelectors) {
                Map<String, Object> selected =
supportedSelector.select();
                if (CollectionUtils.isNotEmptyMap(selected)) {
                    ((RpcInvocation)
invocation).addObjectAttachments(selected);
                }
            }
        } else {
```

```
          ((RpcInvocation)
invocation).addObjectAttachments(RpcContext.getServerAttachment().getO
bjectAttachments());
       }

       Map<String, Object> contextAttachments =
RpcContext.getClientAttachment().getObjectAttachments();
       if (CollectionUtils.isNotEmptyMap(contextAttachments)) {
          /**
           * invocation.addAttachmentsIfAbsent(context){@link
RpcInvocation#addAttachmentsIfAbsent(Map)}should not be used here,
           * because the {@link RpcContext#setAttachment(String,
String)} is passed in the Filter when the call is triggered
           * by the built-in retry mechanism of the Dubbo. The
attachment to update RpcContext will no longer work, which is
           * a mistake in most cases (for example, through Filter to
RpcContext output traceId and spanId and other information).
           */
          ((RpcInvocation)
invocation).addObjectAttachments(contextAttachments);
       }

       // pass default timeout set by end user (ReferenceConfig)
       Object countDown =
context.getObjectAttachment(TIME_COUNTDOWN_KEY);
       if (countDown != null) {
          TimeoutCountDown timeoutCountDown = (TimeoutCountDown)
countDown;
          if (timeoutCountDown.isExpired()) {
             return AsyncRpcResult.newDefaultAsyncResult(new
RpcException(RpcException.TIMEOUT_TERMINATE,
                "No time left for making the following call: " +
invocation.getServiceName() + "."
                   + invocation.getMethodName() + ", terminate
directly."), invocation);
          }
       }

       RpcContext.removeServerContext();
       return invoker.invoke(invocation);
   } finally {
       RpcContext.restoreServiceContext(originServiceContext);
   }
```

```
}
```

继续走过滤器逻辑 FutureFilter

```
public Result invoke(final Invoker<?> invoker, final Invocation
invocation) throws RpcException {
    fireInvokeCallback(invoker, invocation);
    // need to configure if there's return value before the invocation
in order to help invoker to judge if it's
    // necessary to return future.
    return invoker.invoke(invocation);
}
```

过滤器 MonitorFilter

```java
 public Result invoke(Invoker<?> invoker, Invocation invocation) throws
RpcException {
       if (invoker.getUrl().hasAttribute(MONITOR_KEY)) {
          invocation.put(MONITOR_FILTER_START_TIME,
System.currentTimeMillis());
          invocation.put(MONITOR_REMOTE_HOST_STORE,
RpcContext.getServiceContext().getRemoteHost());
          // count up
          getConcurrent(invoker, invocation).incrementAndGet();
       }
       // proceed invocation chain
       return invoker.invoke(invocation);
    }
```

RouterSnapshotFilter 过滤器

```
 public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        if (!switcher.isEnable()) {
            return invoker.invoke(invocation);
        }

        if (!logger.isInfoEnabled()) {
            return invoker.invoke(invocation);
        }

        if (!switcher.isEnable(invocation.getServiceModel().getServiceKey())) {
            return invoker.invoke(invocation);
        }

        RpcContext.getServiceContext().setNeedPrintRouterSnapshot(true);
        return invoker.invoke(invocation);
    }
```

容错模版类 AbstractClusterInvoker

```
public Result invoke(final Invocation invocation) throws RpcException {
        checkWhetherDestroyed();

        // binding attachments into invocation.
//        Map<String, Object> contextAttachments =
RpcContext.getClientAttachment().getObjectAttachments();
//        if (contextAttachments != null && contextAttachments.size() != 0) {
//            ((RpcInvocation) invocation).addObjectAttachmentsIfAbsent(contextAttachments);
//        }

        InvocationProfilerUtils.enterDetailProfiler(invocation, () -> "Router route.");
        //从服务目录里面根据路由规则动态查询invoke服务提供者的调用器
        List<Invoker<T>> invokers = list(invocation);
        InvocationProfilerUtils.releaseDetailProfiler(invocation);
        //获取负载均衡策略默认为随机RandomLoadBalance
        LoadBalance loadbalance = initLoadBalance(invokers, invocation);
        RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);

        InvocationProfilerUtils.enterDetailProfiler(invocation, () -> "Cluster " +
this.getClass().getName() + " invoke.");
        try {
            return doInvoke(invocation, invokers, loadbalance);
        } finally {
            InvocationProfilerUtils.releaseDetailProfiler(invocation);
        }
    }
```

执行具有失效转移功能的 FailoverClusterInvoker 类型的 doInvoke 方法

```
```java
public Result doInvoke(Invocation invocation, final List<Invoker<T>>
invokers, LoadBalance loadbalance) throws RpcException {
    List<Invoker<T>> copyInvokers = invokers;
```
```

```
    checkInvokers(copyInvokers, invocation);
    String methodName = RpcUtils.getMethodName(invocation);
    //重试次数计算默认为 3
    int len = calculateInvokeTimes(methodName);
    // retry loop.
    RpcException le = null; // last exception.
    List<Invoker<T>> invoked = new
ArrayList<Invoker<T>>(copyInvokers.size()); // invoked invokers.
    Set<String> providers = new HashSet<String>(len);
    for (int i = 0; i < len; i++) {
        //Reselect before retry to avoid a change of candidate
`invokers`.
        //NOTE: if `invokers` changed, then `invoked` also lose
accuracy.
        if (i > 0) {
            checkWhetherDestroyed();
            copyInvokers = list(invocation);
            // check again
            checkInvokers(copyInvokers, invocation);
        }
        Invoker<T> invoker = select(loadbalance, invocation,
copyInvokers, invoked);
        invoked.add(invoker);
        RpcContext.getServiceContext().setInvokers((List) invoked);
        boolean success = false;
        try {
            Result result = invokeWithContext(invoker, invocation);
            if (le != null && logger.isWarnEnabled()) {
                logger.warn("Although retry the method " + methodName
                        + " in the service " + getInterface().getName()
                        + " was successful by the provider " +
invoker.getUrl().getAddress()
                        + ", but there have been failed providers " +
providers
                        + " (" + providers.size() + "/" +
copyInvokers.size()
                        + ") from the registry " +
directory.getUrl().getAddress()
                        + " on the consumer " + NetUtils.getLocalHost()
                        + " using the dubbo version " +
Version.getVersion() + ". Last error is: "
                        + le.getMessage(), le);
            }
```

```
            success = true;
            return result;
        } catch (RpcException e) {
            if (e.isBiz()) { // biz exception.
                throw e;
            }
            le = e;
        } catch (Throwable e) {
            le = new RpcException(e.getMessage(), e);
        } finally {
            if (!success) {
                providers.add(invoker.getUrl().getAddress());
            }
        }
    }
    throw new RpcException(le.getCode(), "Failed to invoke the method "
            + methodName + " in the service " + getInterface().getName()
            + ". Tried " + len + " times of the providers " + providers
            + " (" + providers.size() + "/" + copyInvokers.size()
            + ") from the registry " + directory.getUrl().getAddress()
            + " on the consumer " + NetUtils.getLocalHost() + " using
the dubbo version "
            + Version.getVersion() + ". Last error is: "
            + le.getMessage(), le.getCause() != null ? le.getCause() :
le);
}
```

AbstractClusterInvoker 类型的 select，负载均衡算法执行：

```
```java
protected Invoker<T> select(LoadBalance loadbalance, Invocation
invocation,
                            List<Invoker<T>> invokers, List<Invoker<T>>
selected) throws RpcException {

        if (CollectionUtils.isEmpty(invokers)) {
            return null;
        }
        String methodName = invocation == null ?
StringUtils.EMPTY_STRING : invocation.getMethodName();

        boolean sticky = invokers.get(0).getUrl()
```
```

```
          .getMethodParameter(methodName, CLUSTER_STICKY_KEY,
DEFAULT_CLUSTER_STICKY);

       //ignore overloaded method
       if (stickyInvoker != null && !invokers.contains(stickyInvoker))
{
          stickyInvoker = null;
       }
       //ignore concurrency problem
       if (sticky && stickyInvoker != null && (selected == null
|| !selected.contains(stickyInvoker))) {
          if (availableCheck && stickyInvoker.isAvailable()) {
             return stickyInvoker;
          }
       }

       Invoker<T> invoker = doSelect(loadbalance, invocation, invokers,
selected);

       if (sticky) {
          stickyInvoker = invoker;
       }

       return invoker;
    }

    private Invoker<T> doSelect(LoadBalance loadbalance, Invocation
invocation,
                                List<Invoker<T>> invokers, List<Invoker<T>>
selected) throws RpcException {

       if (CollectionUtils.isEmpty(invokers)) {
          return null;
       }
       if (invokers.size() == 1) {
          Invoker<T> tInvoker = invokers.get(0);
          checkShouldInvalidateInvoker(tInvoker);
          return tInvoker;
       }
       Invoker<T> invoker = loadbalance.select(invokers, getUrl(),
invocation);
```

```
        //If the `invoker` is in the  `selected` or invoker is
unavailable && availablecheck is true, reselect.
        boolean isSelected = selected != null &&
selected.contains(invoker);
        boolean isUnavailable = availableCheck && !invoker.isAvailable()
&& getUrl() != null;

        if (isUnavailable) {
            invalidateInvoker(invoker);
        }
        if (isSelected || isUnavailable) {
            try {
                Invoker<T> rInvoker = reselect(loadbalance, invocation,
invokers, selected, availableCheck);
                if (rInvoker != null) {
                    invoker = rInvoker;
                } else {
                    //Check the index of current selected invoker, if it's
not the last one, choose the one at index+1.
                    int index = invokers.indexOf(invoker);
                    try {
                        //Avoid collision
                        invoker = invokers.get((index + 1) %
invokers.size());
                    } catch (Exception e) {
                        logger.warn(e.getMessage() + " may because
invokers list dynamic change, ignore.", e);
                    }
                }
            } catch (Throwable t) {
                logger.error("cluster reselect fail reason is :" +
t.getMessage() + " if can not solve, you can set
cluster.availablecheck=false in url", t);
            }
        }

        return invoker;
    }
```

AbstractClusterInvoker 类型的调用，invokeWithContext 调用方法

```
ListenerInvokerWrapper的invoke方法
```

```
 public Result invoke(Invocation invocation) throws RpcException {
       return invoker.invoke(invocation);
    }
```

## AbstractInvoker 的 invoke

```java
public Result invoke(Invocation inv) throws RpcException {
       // if invoker is destroyed due to address refresh from registry,
let's allow the current invoke to proceed
       if (isDestroyed()) {
          logger.warn("Invoker for service " + this + " on consumer "
+ NetUtils.getLocalHost() + " is destroyed, "
             + ", dubbo version is " + Version.getVersion() + ", this
invoker should not be used any longer");
       }
```

```
    RpcInvocation invocation = (RpcInvocation) inv;

    // prepare rpc invocation 准备调用初始化一些变量
    prepareInvocation(invocation);

    // do invoke rpc invocation and return async result 执行调用
    AsyncRpcResult asyncResult = doInvokeAndReturn(invocation);

    // wait rpc result if sync
    waitForResultIfSync(asyncResult, invocation);

    return asyncResult;
 }
```

## AbstractClusterInvoker 类型的 doInvokeAndReturn

```java
private AsyncRpcResult doInvokeAndReturn(RpcInvocation invocation) {
       AsyncRpcResult asyncResult;
       try {
          asyncResult = (AsyncRpcResult) doInvoke(invocation);
       } catch (InvocationTargetException e) {
          Throwable te = e.getTargetException();
          if (te != null) {
```

```
                // if biz exception
                if (te instanceof RpcException) {
                    ((RpcException)
te).setCode(RpcException.BIZ_EXCEPTION);
                }
                asyncResult = AsyncRpcResult.newDefaultAsyncResult(null,
te, invocation);
            } else {
                asyncResult = AsyncRpcResult.newDefaultAsyncResult(null,
e, invocation);
            }
        } catch (RpcException e) {
            // if biz exception
            if (e.isBiz()) {
                asyncResult = AsyncRpcResult.newDefaultAsyncResult(null,
e, invocation);
            } else {
                throw e;
            }
        } catch (Throwable e) {
            asyncResult = AsyncRpcResult.newDefaultAsyncResult(null, e,
invocation);
        }

        if (setFutureWhenSync || invocation.getInvokeMode() !=
InvokeMode.SYNC) {
            // set server context
            RpcContext.getServiceContext().setFuture(new
FutureAdapter<>(asyncResult.getResponseFuture()));
        }

        return asyncResult;
    }
```

## DubboInvoker 类型的 doInvoke 方法

```
protected Result doInvoke(final Invocation invocation) throws
Throwable {
        RpcInvocation inv = (RpcInvocation) invocation;
        final String methodName = RpcUtils.getMethodName(invocation);
        inv.setAttachment(PATH_KEY, getUrl().getPath());
        inv.setAttachment(VERSION_KEY, version);
```

```
        ExchangeClient currentClient;
        if (clients.length == 1) {
            currentClient = clients[0];
        } else {
            currentClient = clients[index.getAndIncrement() %
clients.length];
        }
        try {
            boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
            int timeout = calculateTimeout(invocation, methodName);
            invocation.setAttachment(TIMEOUT_KEY, timeout);
            if (isOneway) {
                boolean isSent = getUrl().getMethodParameter(methodName,
Constants.SENT_KEY, false);
                currentClient.send(inv, isSent);
                return AsyncRpcResult.newDefaultAsyncResult(invocation);
            } else {
                //这里主要看默认的逻辑同步调用 默认的同步调用模式线程池为
ThreadlessExecutor 无线程线程池
                ExecutorService executor = getCallbackExecutor(getUrl(),
inv);
                CompletableFuture<AppResponse> appResponseFuture =
                        currentClient.request(inv, timeout,
executor).thenApply(obj -> (AppResponse) obj);
                // save for 2.6.x compatibility, for example, TraceFilter
in Zipkin uses com.alibaba.xxx.FutureAdapter

FutureContext.getContext().setCompatibleFuture(appResponseFuture);
                AsyncRpcResult result = new
AsyncRpcResult(appResponseFuture, inv);
                result.setExecutor(executor);
                return result;
            }
        } catch (TimeoutException e) {
            throw new RpcException(RpcException.TIMEOUT_EXCEPTION,
"Invoke remote method timeout. method: " + invocation.getMethodName()
+ ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);
        } catch (RemotingException e) {
            throw new RpcException(RpcException.NETWORK_EXCEPTION,
"Failed to invoke remote method: " + invocation.getMethodName() + ",
provider: " + getUrl() + ", cause: " + e.getMessage(), e);
        }
```

```
    }
```

## ReferenceCountExchangeClient 的 request

```
    public CompletableFuture<Object> request(Object request, int timeout, ExecutorService
executor) throws RemotingException {
        return client.request(request, timeout, executor);
    }
```

## HeaderExchangeClient 的 request

```java
 public CompletableFuture<Object> request(Object request, int timeout,
ExecutorService executor) throws RemotingException {
   return channel.request(request, timeout, executor);
}
```

## HeaderExchangeChannel 的 request 方法

选择语言

```
    public CompletableFuture<Object> request(Object request, int timeout, ExecutorService
executor) throws RemotingException {
        if (closed) {
            throw new RemotingException(this.getLocalAddress(), null, "Failed to send
request " + request + ", cause: The channel " + this + " is closed!");
        }
        // create request.
        Request req = new Request();
        req.setVersion(Version.getProtocolVersion());
        req.setTwoWay(true);
        req.setData(request);
        DefaultFuture future = DefaultFuture.newFuture(channel, req, timeout, executor);
        try {
            channel.send(req);
        } catch (RemotingException e) {
            future.cancel();
            throw e;
        }
        return future;
    }
```

## AbstractPeer 类型的 send

```
 public void send(Object message) throws RemotingException {
        send(message, url.getParameter(Constants.SENT_KEY, false));
    }


    public void send(Object message, boolean sent) throws RemotingException {
        if (needReconnect && !isConnected()) {
            connect();
        }
        Channel channel = getChannel();
        //TODO Can the value returned by getChannel() be null? need improvement.
        if (channel == null || !channel.isConnected()) {
            throw new RemotingException(this, "message can not send, because channel is
closed . url:" + getUrl());
        }
        channel.send(message, sent);
    }
```

## Dubbo netty4 包下的 NettyChannel 的 send

```
    public void send(Object message, boolean sent) throws RemotingException {
        // whether the channel is closed
        super.send(message, sent);

        boolean success = true;
        int timeout = 0;
        try {
            ChannelFuture future = channel.writeAndFlush(message);
            if (sent) {
                // wait timeout ms
                timeout = getUrl().getPositiveParameter(TIMEOUT_KEY, DEFAULT_TIMEOUT);
                success = future.await(timeout);
            }
            Throwable cause = future.cause();
            if (cause != null) {
                throw cause;
            }
        } catch (Throwable e) {
            removeChannelIfDisconnected(channel);
            throw new RemotingException(this, "Failed to send message " +
PayloadDropper.getRequestWithoutData(message) + " to " + getRemoteAddress() + ", cause: " +
e.getMessage(), e);
        }
        if (!success) {
            throw new RemotingException(this, "Failed to send message " +
PayloadDropper.getRequestWithoutData(message) + " to " + getRemoteAddress()
                    + "in timeout(" + timeout + "ms) limit");
        }
    }
```

## AbstractChannel 的 send

```
    public void send(Object message, boolean sent) throws RemotingException {
        if (isClosed()) {
            throw new RemotingException(this, "Failed to send message "
                    + (message == null ? "" : message.getClass().getName()) + ":" +
PayloadDropper.getRequestWithoutData(message)
                    + ", cause: Channel closed. channel: " + getLocalAddress() + " -> " +
getRemoteAddress());
        }
    }
```

netty 包下的 AbstractChannel 的 writeAndFlush 方法

```
public ChannelFuture writeAndFlush(Object msg) {
        return pipeline.writeAndFlush(msg);
    }
```

DefaultChannelPipeline 类型的 writeAndFlush

```
   public final ChannelFuture writeAndFlush(Object msg) {
        return tail.writeAndFlush(msg);
    }
```

AbstractChannelHandlerContext 类型的 writeAndFlush

```
 public ChannelFuture writeAndFlush(Object msg) {
        return writeAndFlush(msg, newPromise());
    }
 public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {
        write(msg, true, promise);
        return promise;
    }
```

AbstractChannelHandlerContext 类型的 write 方法

```
private void write(Object msg, boolean flush, ChannelPromise promise)
{
        ObjectUtil.checkNotNull(msg, "msg");
        try {
            if (isNotValidPromise(promise, true)) {
                ReferenceCountUtil.release(msg);
```

```
                // cancelled
                return;
            }
        } catch (RuntimeException e) {
            ReferenceCountUtil.release(msg);
            throw e;
        }

        final AbstractChannelHandlerContext next =
findContextOutbound(flush ?
                (MASK_WRITE | MASK_FLUSH) : MASK_WRITE);
        final Object m = pipeline.touch(msg, next);
        //executor 位 NioEventLoop 类型对象
        EventExecutor executor = next.executor();
        if (executor.inEventLoop()) {
            if (flush) {
                next.invokeWriteAndFlush(m, promise);
            } else {
                next.invokeWrite(m, promise);
            }
        } else {
            //生成 WriteTask
            final WriteTask task = WriteTask.newInstance(next, m,
promise, flush);
            //事件执行器执行任务 WriteTask
            if (!safeExecute(executor, task, promise, m, !flush)) {
                // We failed to submit the WriteTask. We need to cancel
it so we decrement the pending bytes
                // and put it back in the Recycler for re-use later.
                //
                // See https://github.com/netty/netty/issues/8343.
                task.cancel();
            }
        }
    }

    private static boolean safeExecute(EventExecutor executor,
Runnable runnable,
            ChannelPromise promise, Object msg, boolean lazy) {
        try {
            if (lazy && executor instanceof AbstractEventExecutor) {
                ((AbstractEventExecutor) executor).lazyExecute(runnable);
            } else {
```

```
            executor.execute(runnable);
        }
        return true;
    } catch (Throwable cause) {
        try {
            if (msg != null) {
                ReferenceCountUtil.release(msg);
            }
        } finally {
            promise.setFailure(cause);
        }
        return false;
    }
}
```

## SingleThreadEventExecutor 类型的 execute

```
public void execute(Runnable task) {
    ObjectUtil.checkNotNull(task, "task");
    execute(task, !(task instanceof LazyRunnable) &&
wakesUpForTask(task));
}

private void execute(Runnable task, boolean immediate) {
    //当前线程是否处于事件循环之中
    boolean inEventLoop = inEventLoop();
    //将当前任务添加到 SingleThreadEventExecutor 类型的
MpscUnboundedArrayQueue taskQueue 队列中
    addTask(task);
    if (!inEventLoop) {
        //是否需要开启线程
        startThread();
        if (isShutdown()) {
            boolean reject = false;
            try {
                if (removeTask(task)) {
                    reject = true;
                }
            } catch (UnsupportedOperationException e) {
                // The task queue does not support removal so the best
thing we can do is to just move on and
```

```
                // hope we will be able to pick-up the task before its
completely terminated.
                // In worst case we will log on termination.
            }
            if (reject) {
                reject();
            }
        }
    }

    if (!addTaskWakesUp && immediate) {
        wakeup(inEventLoop);
    }
}
```

SingleThreadEventExecutor 类型的 wakeup

```
protected void wakeup(boolean inEventLoop) {
    if (!inEventLoop && nextWakeupNanos.getAndSet(AWAKE) != AWAKE) {
        selector.wakeup();
    }
}
```

SelectedSelectionKeySetSelector 类型的 wakeup

```
public Selector wakeup() {
    return delegate.wakeup();
}
```

NioEventLoop 的 wakeup

```
protected void wakeup(boolean inEventLoop) {
    if (!inEventLoop && nextWakeupNanos.getAndSet(AWAKE) != AWAKE) {
        selector.wakeup();
    }
}
```

我用的 mac 电脑走的是 KQueueSelectorImpl 逻辑

```
public Selector wakeup() {
      synchronized (interruptLock) {
          if (!interruptTriggered) {
              try {
                  IOUtil.write1(fd1, (byte)0);
              } catch (IOException ioe) {
                  throw new InternalError(ioe);
              }
              interruptTriggered = true;
          }
      }
      return this;
  }
```