SOFAStack

微服务

使用指南

产品版本：AntStack Plus 1.9.0

文档版本：20221020

蚂蚁集团
ANT GROUP

法律声明

蚂蚁集团版权所有©2022，并保留一切权利。

未经蚂蚁集团事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

商标声明

蚂蚁集团 ANT GROUP及其他蚂蚁集团相关的商标均为蚂蚁集团所有。本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁集团保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁集团授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁集团授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

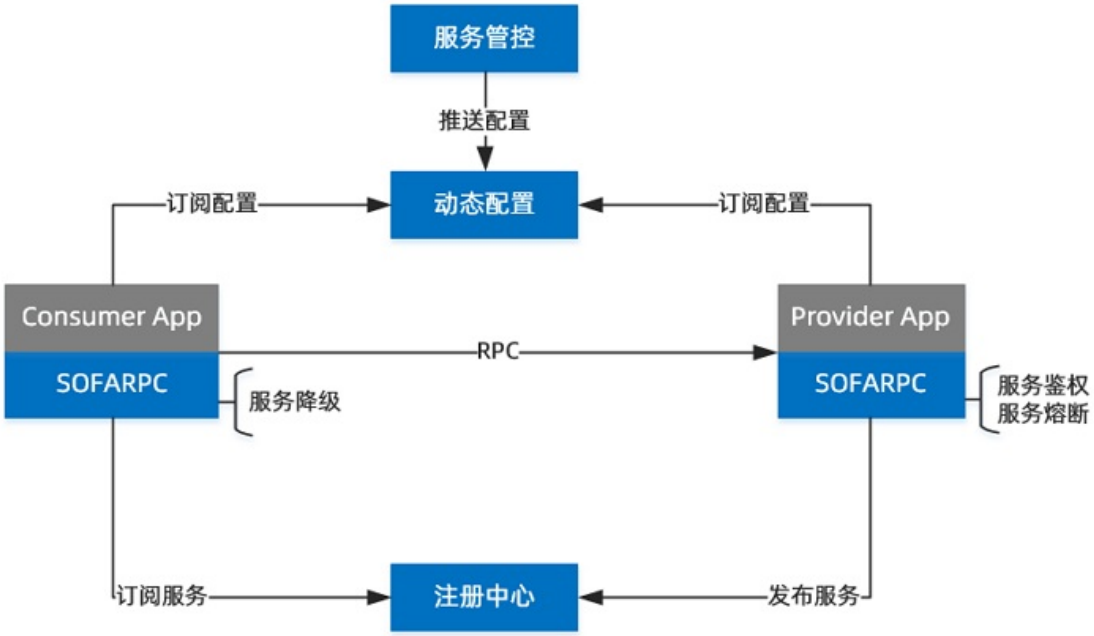
1.什么是微服务

1.1. 概述

微服务（SOFAStack Microservices，简称 SOFAStack MS）主要提供分布式应用常用解决方案。使用微服务框架可以进行服务的自动注册与发现，以及服务管理相关操作。

使用 SOFABoot 开发应用，通过 SOFAStack 控制台部署到云端后，微服务会自动注册到服务注册中心。您可以通过微服务控制台进行服务管控和治理的相关操作。

微服务主要通过 SOFARPC 实现服务的发布和引用，其它模块都围绕 SOFARPC 展开。产品架构如下：



服务注册

服务注册通过注册中心（SOFARegistry）实现。注册中心是蚂蚁中间件的底层组件，用于存储所有服务提供方的地址信息，以及所有服务消费方的订阅信息。它和服务消费方、服务提供方都建立长连接，动态感知服务发布地址变更，并通知消费方。

RPC 服务

提供对 SOFARPC 的支持。SOFARPC 是一个分布式服务框架，为应用提供高性能、透明化、点对点的远程服务调用方案，具有高可伸缩性、高容错性。

动态配置

动态配置（Distributed Resource Management，简称 DRM）可以实现在应用运行时，动态修改配置的功能。提供动态配置的简便接入方式与集中化管理平台，可在管理平台维护动态配置元数据，并可对配置值进行推送，还可以实时查看接入动态配置的客户端应用节点的内存值。

服务治理

提供对业务系统的限流、熔断、降级服务，从而保证业务系统不会被大量突发请求击垮，提高系统稳定性。

应用依赖

应用通过 RPC 发布、订阅服务时，应用依赖可以提供实时分析结果，可展示不同应用之间的服务调用关系，以及应用发布和订阅的服务信息。

1.2. 产品优势

微服务产品在蚂蚁集团内部已支撑数万个节点规模的分布式应用架构，具有高可用性、高可扩展性、高性能、高时效性、稳定可靠等核心优势，并提供丰富的功能来帮助用户简化分布式系统的管理，让业务开发人员可以专注于业务逻辑实现，提升研发效率。

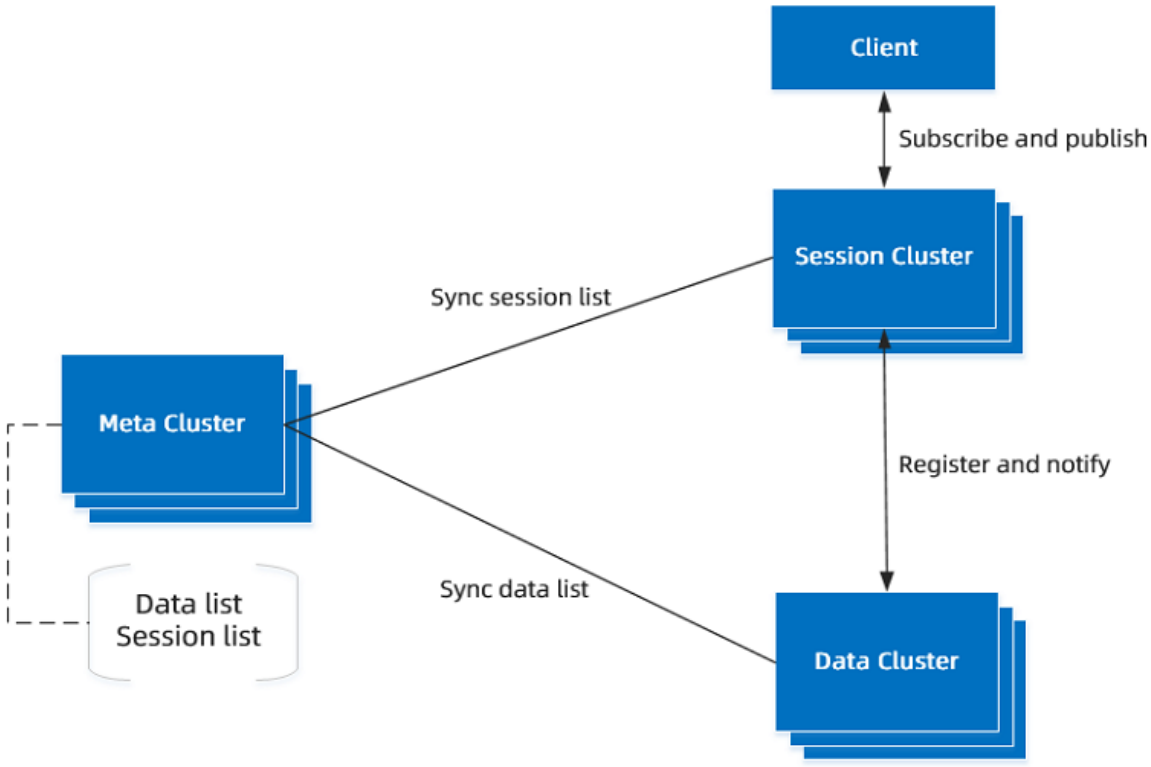
微服务产品提供金融级分布式架构的基础设施能力，包括 RPC 框架及服务治理、服务注册与发现、动态配置、定时任务、服务限流等，为传统单体应用架构深入拆分为分布式应用架构提供稳定可靠的基础设施能力，帮助企业级客户快速构建基于微服务架构的分布式应用，从而实现更灵活地响应业务变化，提高系统的可扩展性及性能。

1.3. 产品架构

微服务包含服务注册、服务治理、动态配置及 RPC 服务，本文详细介绍提供各服务的组件架构信息。

SOFARegistry

SOFARegistry 即服务注册中心，提供服务注册服务。产品架构如下：

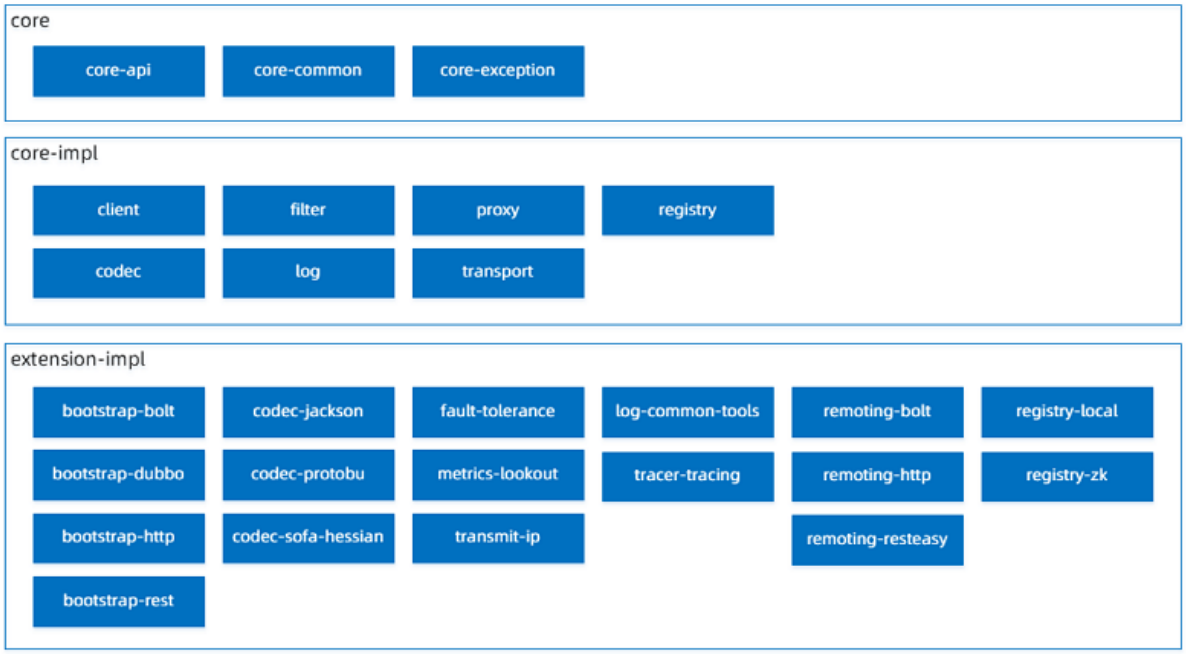


- **Client（客户端）**：提供应用接入服务注册中心的基本 API 能力，应用系统依赖客户端 JAR 包，通过编程方式调用服务注册中心的服务订阅和服务发布能力。
- **SessionServer（会话服务器）**：提供客户端接入能力，接受客户端的服务发布及服务订阅请求，并作为一个中间层将发布数据转发至 DataServer 存储。SessionServer 可无限扩展以支持海量客户端连接。
- **DataServer（数据服务器）**：负责存储客户端发布数据，按照数据 ID 使用自定义 slot 分配算法分片存储数据存储，支持多副本备份，保证数据高可用。DataServer 可无限扩展以支持海量数据。

- **MetaServer（元数据服务器）**：负责维护集群 SessionServer 和 DataServer 的一致列表，在节点变更时及时通知集群内其他节点。MetaServer 通过 SOFAJRaft 保证高可用和一致性。

SOFARPC

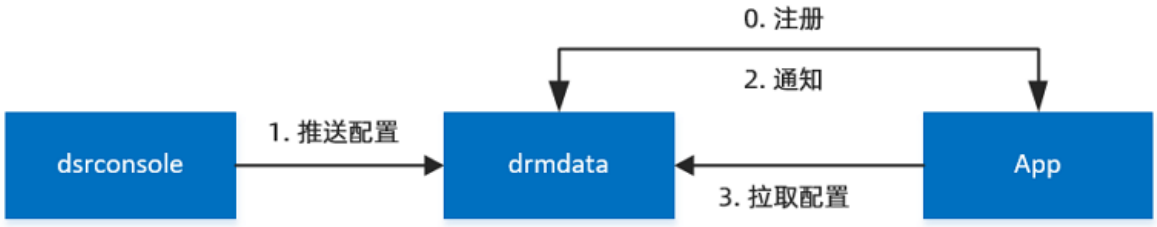
SOFARPC 是一个分布式服务框架，为应用提供高性能、透明化、点对点的远程服务调用方案。产品架构如下：



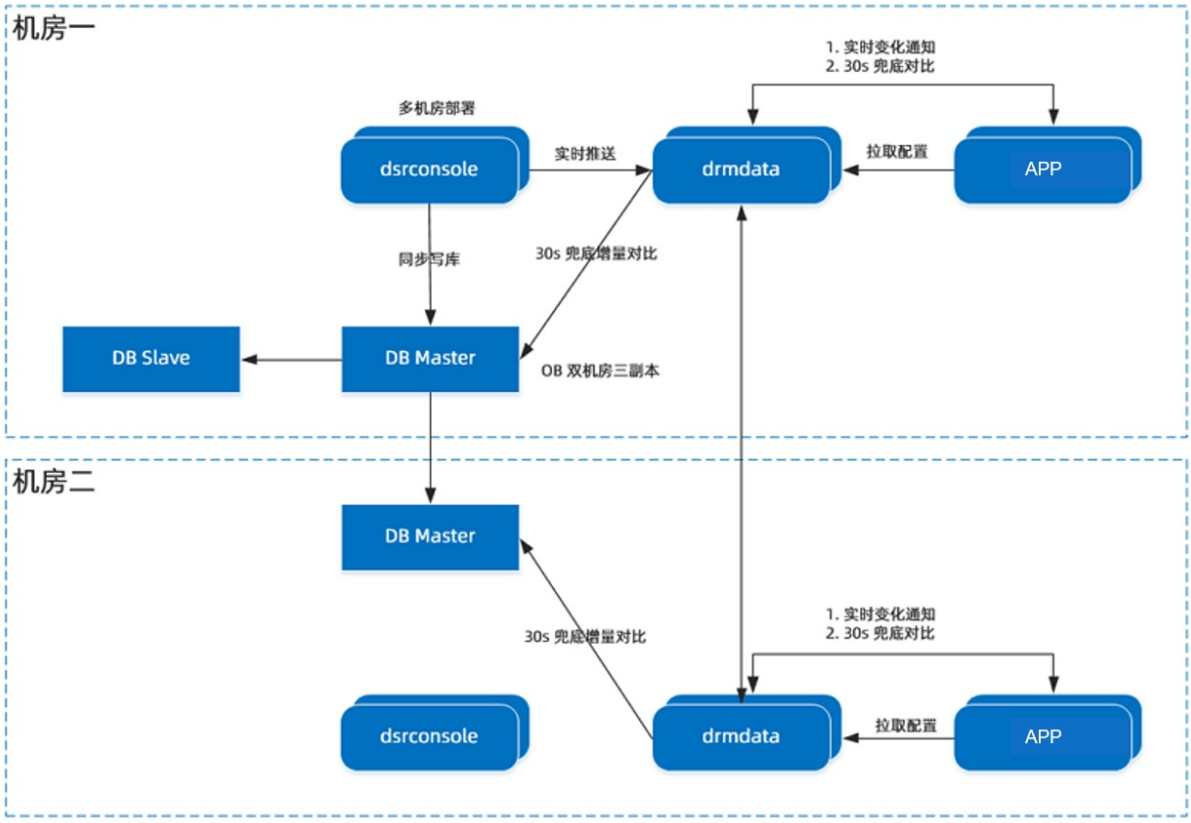
- **核心功能**：包括 core 和 core-impl 两部分，主要内容为 API 和一些扩展机制。
- **扩展及实现**：即 extension-impl 部分，主要包含了不同的实现和扩展，比如对 HTTP、REST、 metrics 以及其他注册中心的集成和扩展，例如 bootstrap 中对协议的支持，remoting 中对网络传输的支持，registry 中对注册中心的支持等。

DRM

DRM 的具体配置值并不通过服务注册中心推送，它由 dsrconsole 和 drmdata 两个组件组成。dsrconsole 负责实时推送时配置的持久化化，并同步给 drmdata。drmdata 则维护着与各个应用之间的注册链接，并在收到 dsrconsole 的推送通知时缓存配置并通知各个订阅应用来拉取配置。应用接收到该指令后，知道配置项发生了变更，向 drmdata 发起 HTTP 请求，读取真正的配置值，过程如下图所示：



DRM 的整套模型中，配置项的变更推送仅起到通知作用，推送的是版本号，而真实的配置值是依靠客户端接收到推送后主动发起的 HTTP 拉取。为了提高读取性能，drmdata 采用缓存提升读取效率，请求透传到 Java Server，经过 Java Server 的内存缓存过滤一次，未命中请求才可能请求到 DB。推送流程如下：

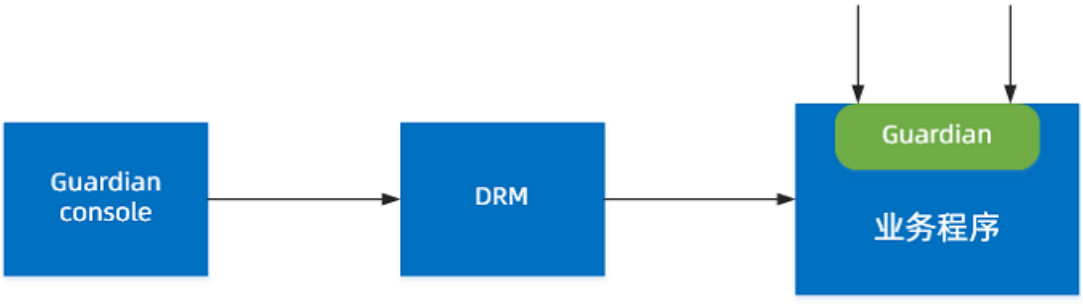


DRM 完整的推送流程是：

1. 在页面上，单击 **推送** 按钮。
2. dsrconsole 写最新的推送数据到 DB，DB 返回此 key 的最新版本号。
3. dsrconsole 通知推送消息到同机房的一台 drmdata 机器，消息内容包括：key、value、version。
4. drmdata 收到推送消息后，广播通知消息到所有的机房的 drmdata 机器。
5. 每台 drmdata 收到推送消息后，根据 key 查询所有 Client 建立的长连接信息，通过长连接发送给客户端 key 最新的版本号。
6. Client 收到推送消息后，拿着 drmdata 告知的最新版本号，随机调用同机房 drmdata 提供拉取配置的 HTTP 接口，拉取最新的配置。

Guardian

Guardian 主要用于服务限流，产品架构如下：

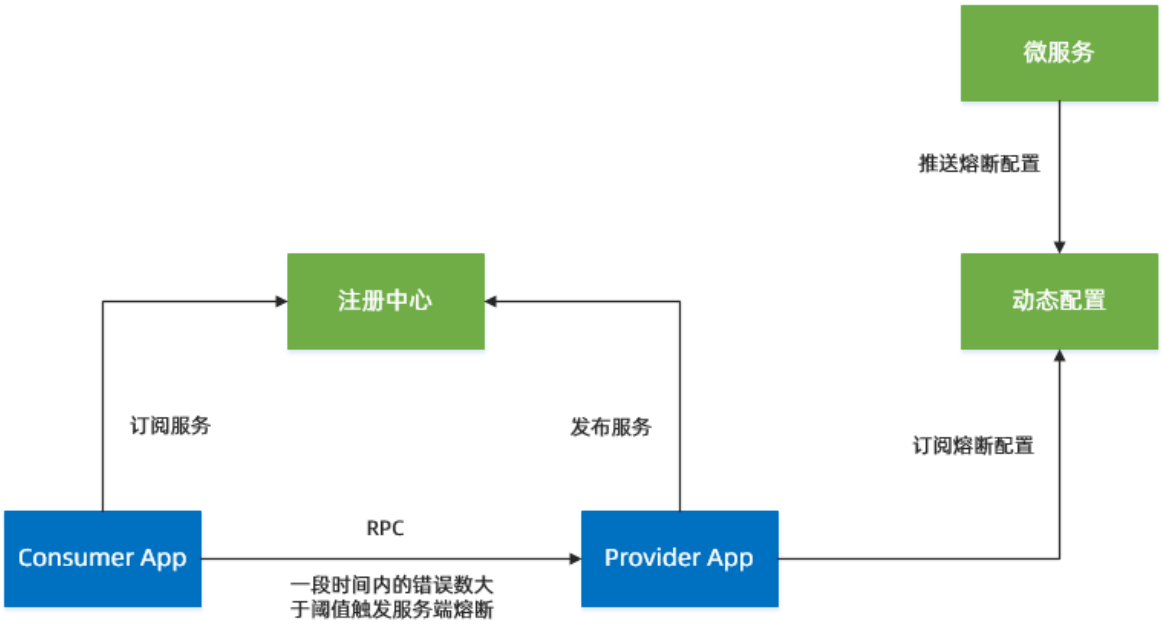


Guardian 运行主要分成为以下 4 个部分：

- **组件集成**：业务程序首先要集成 Guardian。
- **规则推送**：用户在 Guardian Console（控制台） 上编辑限流规则，并且把编辑好的规则推送到客端户，限流规则才会生效。
- **流量匹配**：当有流量进入到业务程序，首先会被 Guardian 组件拦截到，Guardian 会判断当前流量是否和限流规则匹配。
- **限流生效**：如果流量和限流规则匹配上，并且达到了预设的限流值，则限流。

服务熔断

服务熔断主要目的是当某个服务故障或者异常时，如果该服务触发熔断，可以防止其他调用方一直等待所导致的超时或者故障，从而防止雪崩。产品架构如下：



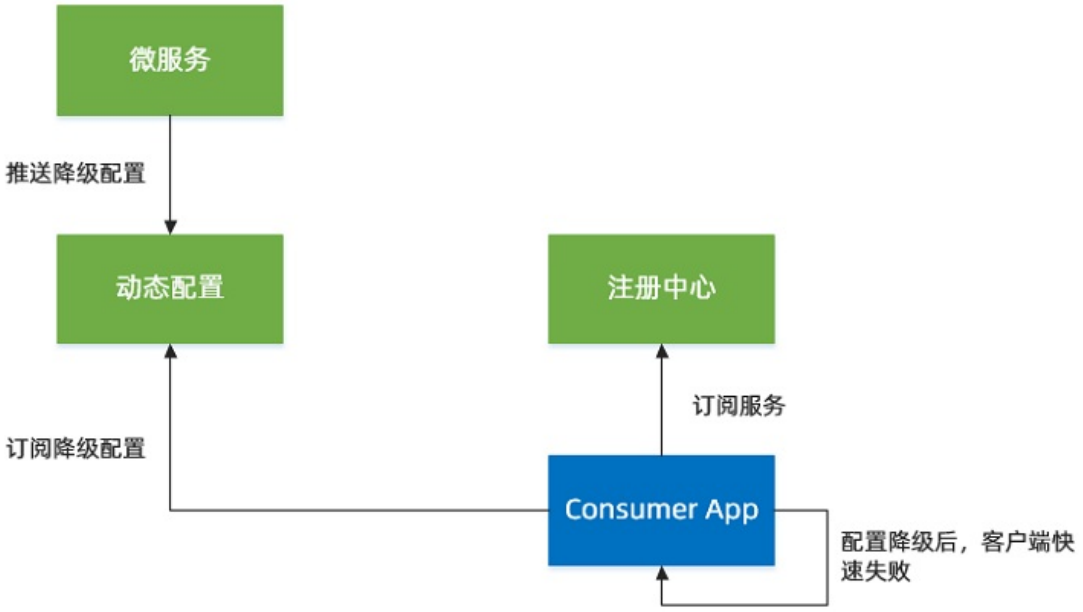
- **Provider App**：指服务提供端发布服务，并向注册中心注册。
- **Consumer App**：指服务使用端通过从注册中心拉取信息，来使用服务。
- **注册中心**：即 SOFARegistry，是 SOFA 中间件的底层组件，用于存储所有服务提供方的地址信息以及所有服务消费方的订阅信息；它和服务消费方、服务提供方都建立长连接，动态感知服务发布地址变更并通知消费方。

服务熔断主要分为以下 4 个部分：

- **订阅配置**：Provider App 首先要订阅熔断配置。
- **配置生效**：用户在微服务控制台编辑、推送熔断配置值，并通过动态配置（DRM）进行动态调整。配置推送后，Provider App 所订阅的配置即开始生效。
- **熔断触发**：Consumer App 通过 RPC 调用服务后，在规定时间内，如果错误数大于阈值，会触发服务端熔断配置。
- **熔断自动恢复**：当触发熔断后，RPC 会在一个时间窗口内快速失败。过了这个时间窗口后，系统会允许通过一个请求，去尝试探测服务是否恢复。如果恢复，则自动关闭熔断；如果没有恢复，则继续熔断，并等待下一个时间窗口。

服务降级

服务降级的主要目的是当服务器压力剧增的情况下，根据实际业务情况及流量，对某些不重要的服务，不处理或换种简单的方式处理，从而释放服务器资源以保证核心业务正常运作或高效运作。产品架构如下：



- **Consumer App**：指服务使用端通过从注册中心拉取信息，来使用服务。
- **注册中心**：即 SOFARegistry，是 SOFA 中间件的底层组件，用于存储所有服务提供方的地址信息以及所有服务消费方的订阅信息；它和服务消费方、服务提供方都建立长连接，动态感知服务发布地址变更并通知消费方。

服务降级主要分为以下3 个部分：

- **订阅配置**：Consumer App 首先要订阅降级配置。
- **配置生效**：用户在微服务控制台编辑、推送降级配置值，并通过动态配置进行动态调整。配置推送后，Consumer App 所订阅的配置即开始生效。
- **降级触发**：当 Consumer App 触发服务降级后，Consumer App 将无法使用服务，而且表现为快速失败。此时，需要业务方主动处理降级所带来的影响。

1.4. 功能特性

微服务有高性能分布式服务框架、微服务治理中心、高可靠的轻量级配置中心、多活数据中心等特性，本文主要介绍这些特性。

高性能分布式服务框架

提供高性能和透明化的 RPC 远程服务调用，具有高可伸缩性、高容错性的特点。

- 支持多协议、多序列化、多语言，包括 Bolt（默认协议）、Dubbo、RESTful、WebService、Protobuf、Hessian、JSON 等。
- 服务自动注册与发现支持服务自动注册与发现，无需配置地址即可实现分布式环境下的负载均衡，并支持多种路由策略及健康检查。
- 依赖管理视图提供对 RPC 发布订阅的实时结果，可展示不同应用之间的服务调用关系，以及应用发布和订阅的服务信息。

微服务治理中心

提供一系列的服务治理策略，保障服务高质量运行，最终达到对外承诺的服务质量等级协议。

- 服务高可用支持客户端限流、集群容错（失败重试）、服务熔断（故障剔除）、故障注入、服务降级等保障服务高可用。
- 服务安全支持 CRC 校验，调用加解密，黑白名单等保障服务的安全。
- 服务的监控支持 Metrics 2.0 规范的日志埋点，支持成功率、调用次数、耗时、异常次数等多维度监控信息。

高可靠的轻量级配置中心

提供应用运行时动态修改配置的服务，并提供图形化的集中化管理界面。

- 配置动态推送实时生效支持按全量 IP 地址及指定 IP 地址进行配置推送，无需重启应用，并支持推送回滚。
- 客户端信息管理可查看客户端列表信息，包括客户端的当前内存值及服务端的推送值。
- 推送记录管理支持在控制台查看动态配置的推送记录，并支持以文件的方式对配置进行批量导入及导出。

多活数据中心

支持同城双活、异地多活架构，具备异地容灾能力，保障系统的可用性。

- 支持多种维度系统扩展支持应用级、数据库级、机房级、地域级的快速扩展。
- 按机房进行服务发现和路由支持跨 IDC 的服务发现，并支持按机房进行路由。
- 按数据中心进行配置修改支持按数据中心进行配置的动态推送，不同的机房的配置可根据业务需求设置为不同的值。

1.5. 应用场景

SOFAStack 微服务具有高性能、高可靠、高可用的特点，适用于以下应用场景。

传统应用微服务改造

通过微服务产品将传统金融业务系统拆分为模块化、标准化、松耦合、可插拔、可扩展的微服务架构，可缩短产品面世周期，快速上架，抢占市场先机，不仅可确保客户服务的效率，也降低了运营成本。

- 开发简单：提供高性能微服务框架，轻松构建原生云应用，具备快速开发，持续交付和部署的能力。
- 管理简单：框架自带服务治理能力，使用门槛低，可轻松管理成千上万个服务实例，保障服务高质量运行。
- 接入门槛低：完全托管的 SaaS 服务，轻资产，且无需自己部署及运维，有效降低投入成本。

高并发业务快速扩展

通过微服务产品开发互联网金融业务可提高研发效率，更灵活地响应业务变化，快速迭代创新产品，并针对热点模块进行快速扩展来提高处理能力，轻松应对突发流量，同时提高用户体验，为更多小微客户提供个性化的金融产品和交易成本较低的便捷金融服务。

- 高性能：提供基于事件驱动的架构以及私有通信协议，轻松搭建低延迟、高吞吐的服务。
- 可扩展性强：支持无限水平扩展，无性能、容量瓶颈，在蚂蚁集团内部已支撑数万个节点规模的分布式应用架构。
- 可视化管理：在分布式系统中，面对爆发式增长的应用数量和服务器数量，提供图形化的集中式管理平台，简单易用，学习成本低。

多数据中心异地多活

通过微服务产品可快速构建高可扩展、高性能的金融级分布式核心系统，拥有弹性扩容和异地多活的能力，实现技术安全自主可控，突破业务发展瓶颈，并减少开发及运维成本。实现轻型银行，助力业务快速发展和持续创新。

- 异地多活：支持同城双活、异地多活架构，具备异地容灾能力。
- 弹性扩容：支持应用级、数据库级、机房级、地域级的快速扩展。
- 自主可控：基于支付宝的业务迭代衍生完全自主研发，产品拥有完全自主知识产权，自身开源开放，并兼容开源生态。

1.6. 使用限制

SOFAStack微服务是高性能分布式微服务框架，在使用时具有以下环境要求和限制。

限制项	限制范围	限制说明
微服务开发框架	Dubbo/SpringCloud/SOFA	服务注册中心 SDK 支持这三种框架的兼容
SOFA SDK 语言限制	Java	SOFA SDK 支持 Java 语言开发
JDK 版本	JDK 1.8 及以上	JDK 版本支持 1.8 及以上

1.7. 基础术语

本文根据模块对微服务涉及的基础术语进行说明。

SOFARegistry

中文	英文	释义
服务注册中心	SOFARegistry	蚂蚁金融科技开源的一款服务注册中心产品，基于“发布-订阅”模式实现服务发现功能。同时它并不假定总是用于服务发现，也可用于其他更一般的“发布-订阅”场景。
数据	Data	在服务发现场景下，特指服务提供者的网络地址及其它附加信息。其他场景下，也可以表示任意发布到 SOFARegistry 的信息。
单元	Zone	单元化架构关键概念，在服务发现场景下，单元是一组发布与订阅的集合，发布及订阅服务时需指定单元名，更多内容可参考异地多活单元化架构解决方案。

中文	英文	释义
发布者	Publisher	发布数据到 SOFARegistry 的节点。在服务发现场景下，服务提供者就是“服务提供者的网络地址及其它附加信息”的发布者。
订阅者	Subscriber	从 SOFARegistry 订阅数据的节点。在服务发现场景下，服务消费者就是“服务提供者的网络地址及其它附加信息”的订阅者。
数据标识	DataId	用来标识数据的字符串。在服务发现场景下，通常由服务接口名、协议、版本号等信息组成，作为服务的标识。
分组标识	GroupId	用于为数据归类的字符串，可以作为数据标识的命名空间，即只有 DataId、GroupId、InstanceId 都相同的服务，才属于同一服务。
实例 ID	InstanceId	实例 ID，可以作为数据标识的命名空间，即只有 DataId、GroupId、InstanceId 都相同的服务，才属于同一服务。
会话服务器	SessionServer	SOFARegistry 内部负责跟客户端建立 TCP 长连接、进行数据交互的一种服务器角色。
数据服务器	DataServer	SOFARegistry 内部负责数据存储的一种服务器角色。
元信息服务器	MetaServer	SOFARegistry 内部基于 Raft 协议，负责集群内一致性协调的一种服务器角色。
数据中心	Data Center	物理位置、供电、网络具备一定独立性的物理区域，通常作为高可用设计的重要考量粒度。一般可认为：同一数据中心内，网络质量较高、网络传输延时较低、同时遇到灾难的概率较大；不同数据中心间，网络质量较低、网络延时较高、同时遇到灾难的概率较小。

SOFARPC

中文	英文	释义
RPC	RPC	远程方法调用（Remote Procedure Call）。

中文	英文	释义
RPC 服务	RPC service	服务端提供接口的实现对象。
RPC 引用	RPC reference	客户端针对 RPC 服务创建的一个代理对象。
服务 ID	service ID	服务唯一标识，由接口全路径、版本、分组与通讯协议组成的唯一标识。
服务提供方	service provider	提供 RPC 服务的应用。
服务消费方	service consumer	使用 RPC 服务的应用。
服务注册中心	Service Registry	一个独立的应用集群，用来存储和维护所有在线的 RPC 应用地址列表。
服务参数	service parameters	服务提供者可被动态修改的参数，如权重、状态。
服务发现	Service Discovery	服务消费者获取服务提供者的网络地址的过程。

动态配置

中文	英文	释义
配置类	Configuration class	业务应用中的一个普通 Java 对象，按动态配置框架的编程 API 注册后，成为一个可被外界动态管理的资源，称为配置类。域、应用、类标识三者唯一标识一个配置类实例。
域	domian	配置类的一个命名空间，默认值为 Alipay，可通过编程注解修改。
所属应用	application	配置类所属的应用名。
类标识	class ID	代表配置类的一个字符串，跟应用代码中 @DObject 注解的 ID 字段一致，通常使用全类名。

中文	英文	释义
属性	attribute	配置类对象的具有公有读写方法的私有属性。一个配置类下可以有多个属性。一个配置类属性对应业务的一个配置项。
属性名	attribute name	代表属性的字符串，跟业务代码中的私有属性命名一致。
DataId	DataId	用于全局唯一标识一个属性的字符串，由域、应用、类标识、属性名四者按一定规则拼接而成。
drm-client	drm-client	动态配置框架的客户端 JAR 包。

服务治理

中文	英文	释义
运行模式	running mode	指限流 guardian 客户端对限流的处理方式，分为监控模式和拦截模式。
拦截模式	intercept mode	限流匹配上后，会实际拦截请求。
监控模式	monitor mode	限流匹配上后，不会实际拦截请求，只会打印限流记录日志。
限流后操作：空处理	post-throttling operation: null process	不做任何处理，直接返回。 对于接口方法，返回 null；对于 Web 页面，返回为空，并结束本次页面访问。

2.快速入门

微服务（SOFAStack MicroService）提供分布式应用常用解决方案，支持在线配置、管理、监控 SOFA 应用等。它主要是通过 SOFARPC 来实现服务的发布和引用，而服务注册、动态配置、限流熔断、服务降级等功能，都是服务于SOFARPC 的。快速入门以 SOFARPC 的实现为载体，带您体验微服务的整个流程。

操作步骤

1. 在本地实现 SOFARPC 服务。

   在本地使用 SOFABoot 框架实现 SOFARPC 服务，主要包括下述步骤：

   i. 搭建 SOFABoot 环境。

   具体操作，请参见 搭建环境。

   ii. 创建 SOFABoot Web 工程，分别作为服务发布方和引用方。

   您可以通过以下任一方式生成 2 个 SOFABoot Web 工程，分别作为服务发布方和引用方。

   - 创建 2 个 SOFABoot Web 工程。具体操作，请参见 创建工程。
   - 直接下载 SOFARPC Demo。更多详情，请参见 SOFARPC 快速入门。

   iii. 开发本地业务逻辑。

   如果需要引入微服务组件，请参考下述文档：

   - DRM 组件，请参见 动态配置快速入门。
   - Guardian 组件，请参见 服务限流快速入门。

   iv. 配置 `application.properties` 。

   本地测试时无需配置该项。在云端发布前，请务必完成下述属性配置。更多详情，请参见 引入 SOFA 中间件。

2. 接入微服务组件依赖。

   在 SOFABoot Web 工程中 `endpoint` 模块下的 pom.xml 文件中，引入下述依赖：

   - 动态配置依赖

   ```
   <dependency>
       <groupId>com.alipay.sofa</groupId>
       <artifactId>ddcs-enterprise-sofa-boot-starter</artifactId>
   </dependency>
   ```

   - 服务限流依赖

   ```
   <dependency>
       <groupId>com.alipay.sofa</groupId>
       <artifactId>guardian-sofa-boot-starter</artifactId>
   </dependency>
   ```

3. 应用打包和云端发布

   i. 打包本地应用。

   操作步骤，请参见 编译运行。

ii. 发布应用。

- 应用整体发布流程，请参见 技术栈与应用发布流程。
- 应用的详细发布步骤，请参见 经典应用服务快速入门。

4. 服务管控和治理。

您可以通过 SOFAStack 控制台 进行微服务的应用管理，包括 动态配置、应用依赖、服务限流、服务熔断、服务降级、故障注入、服务鉴权 等。

> 说明
>
> - 目前 SOFAStack 控制台通过 SOFARegistry 来实现服务的注册、发现和引用，暂不支持通过 VPN 的方式连注册中心。通过本地注册中心的方式，也无法体验 SOFAStack 控制台中微服务的服务管控和治理功能。
> - 在本地可以通过 IP 直连的方式来体验 SOFARPC 服务，但是，不能在本地体验微服务的服务管控和治理功能。
> - 后续将会开放一个公网环境的 SOFAStack 体验 Region，届时即可在本地体验所有 SOFAStack 服务，敬请期待。

3.SOFARPC

3.1. 概述

SOFARPC 提供应用之间的点对点服务调用功能。

产品特性

- **高可用**

  SOFARPC 提供软件负载均衡的能力，它是对等服务调用的调度器，会帮助服务消费方在这些对等的服务提供方中合理地选择一个来执行相关的业务逻辑。

- **高容错**

  一切服务调用的容错机制均由软负载和配置中心控制，这样可以在应用系统无感知的情况下，帮助服务消费方正确选择健康的服务提供方，保障全站的稳定性。

基本功能

主要为用户提供下述功能：

- **多种服务路由方式**：包括软负载、硬负载、直连等。
- **负载均衡**：支持随机策略、考虑长连接、权重等负载均衡策略。
- **多种调用方式**：支持同步、单向、回调、泛化等多种调用方式。
- **多种编程界面**：支持 XML、动态客户端、Standalone 模式等多种编程界面。
- **流量转发**：支持应用之间的流量转发。
- **链路追踪**：支持网格外部应用调用网格内部应用并形成一个完整的链路追踪信息
- **链路数据透传**：支持应用调用上下文中存放数据，达到整个链路上的应用都可以操作该数据。
- **故障剔除**：目前支持 bolt 协议。它会自动监控 RPC 调用的情况。

协议支持

SOFARPC 支持不同的通讯协议，目前主要包括：

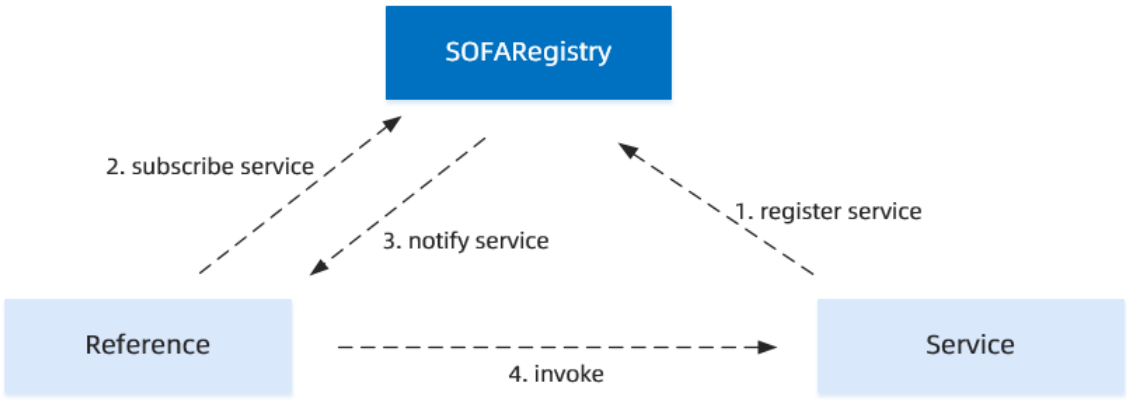
- **BOLT**：是蚂蚁金融科技服务集团开放的，基于 Netty 开发的网络通信框架。
- **RESTful**：基于 HTTP 一种设计框架。
- **Dubbo**：开源分布式服务框架
- **H2C**：开放的网络通信框架。

实现原理

SOFARPC 中的远程调用是通过服务模型来定义服务调用双方的。服务分为：

- **服务消费方**：对应 RPC 的调用端，可以理解为调用客户端，即 “引用 (Reference)” 。
- **服务提供方**：对应 RPC 的被调用端，可以理解为调用服务端。即 “服务 (Service)” 。

SOFARPC 实现原理示意图



上述原理说明如下：

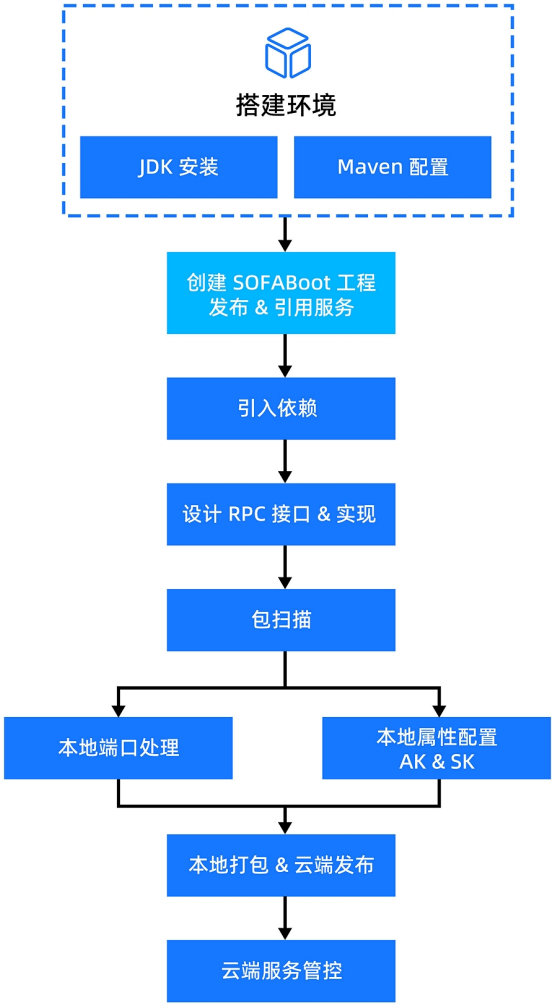
1. **register service**：当一个 SOFARPC 的应用启动的时候，如果发现当前应用需要发布 RPC 服务的话，那么 SOFARPC 会将这些服务注册到服务注册中心上，如上图中 Service 指向 Registry。
2. **subscribe service**：当引用这个服务的 SOFARPC 应用启动时，会从服务注册中心订阅到相应服务的元数据信息。
3. **notify address**：服务注册中心收到订阅请求后，会将发布方的元数据列表实时推送给服务引用方，如上图中 Registry 指向 Reference。
4. **invoke**：当服务引用方拿到地址以后，就可以从中选取地址发起调用了，如上图中 Reference 指向 Service。

3.2. SOFARPC 快速入门

微服务（SOFAStack MicroService）主要是通过 SOFARPC 来实现服务的发布和引用，微服务中的其它模块也都围绕 SOFARPC 展开。本文以微服务本地开发到云端发布的整体流程为框架，让您了解 SOFARPC 如何在本地实现、如何发布到云端、如何在云端进行服务管控。在云端发布前，示例工程也支持直连的方式，让您在本地体验 SOFARPC 的服务发布和服务引用。

本地工程开发

SOFARPC 工程开发流程图



准备工作

- 搭建 SOFABoot 环境。具体操作，请参见 搭建环境。

  > **重要**
  >
  > 暂时不支持非 SpringBoot、SOFABoot 应用接入。

- 通过以下任一方式生成 2 个 SOFABoot Web 工程，分别作为服务发布方和引用方。
  - 创建 2 个 SOFABoot Web 工程。具体操作，请参见 创建工程。
  - 直接下载 示例工程。下载后，请参考 版本说明 将工程根目录下 `pom.xml` 文件中的版本号修改为最新版本号。

本地开发流程

1. 引入依赖。

   按步骤创建的工程默认已经引入该依赖，请忽略此步骤；直接下载的示例工程需在 2 个本地 SOFABoot 工程 Web 模块 `pom.xml` 中引入 SOFARPC 的 Maven 依赖。

```
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>rpc-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

2. 编写业务逻辑。

   主要为服务发布和服务引用。本示例使用注解的方式配置 Bean，实现服务发布和引用。更多详情，请参见 使用注解方式。其它配置方式，请参见 使用 XML 配置 及 使用编程 API。

   - 服务发布的业务逻辑
     - 设计服务接口类

       本示例类名称为 `SampleService.java` ，接口路径为 `com.alipay.samples.rpc.SampleService` 。

       ```
       public interface SampleService{
          public String hello();
       }
       ```

     - 编写服务实现类

       本示例类名称为 `SampleServiceImpl.java` ，接口实现路径为 `com.alipay.samples.rpc.impl.SampleServiceImpl` 。

       ```
       @Service
       @SofaService(interfaceType =SampleService.class,bindings =@SofaServiceBinding(bindi
       ngType ="bolt"))
       public class SampleServiceImpl implements SampleService{
          private int count =0;

          @Override
          public String hello(){
             return "Hello SOFARpc! times = "+ count++;
          }
       }
       ```

   - 服务引用的业务逻辑

- 引用对象注入

本示例采用 `ReferenceHolder` 类对引用对象进行统一管理，示例如下：

```
@Component
public class ReferenceHolder {
    @SofaReference(interfaceType = SampleService.class, binding = @SofaReferenceBin
ding(bindingType = "bolt", directUrl = "127.0.0.1:12201"))
    private SampleService sampleService;

    public SampleService getSampleService() {
        return sampleService;
    }

    public void setSampleService(SampleService sampleService) {
        this.sampleService = sampleService;
    }
}
```

> **说明**
>
> `directUrl="127.0.0.1:12201"` ：注意端口号 12201 需要和 `myserver-app` Web 模块 `src/main/resources/config/application.properties` 中的配置 `rpc.tr.port=12201` 对应。

- 引用对象

为了方便直观使用，本示例将引用服务逻辑放在 `myclient-app` 工程 Web 模块的 `com.alipay.mytestsofa.SOFABootWebSpringApplication` 中，示例如下：

```
@SpringBootApplication
public class SOFABootWebSpringApplication{
    private static final Logger logger =LoggerFactory.getLogger(SOFABootWebSpringAp
plication.class);

    public static void main(String[] args){
        //*************** 注意 ******************//
        //本地同时启动 myserver-app 和 myclient-app 时，由于 tomcat 端口冲突问题，需要修
改 myclient-app 的 端口号为 8084。
        //将 myserver-app 和 myclient-app 发布到云上环境时，由于默认健康检查端口是 8080，
所以需要注释掉该行代码。
        System.setProperty("server.port","8084");
        //由于本地启动没有注册中心，所以使用本地直连的方式访问本地启动的 myserver-app，发布
到线上的时候需要注释掉该行代码。
        System.setProperty("run.mode","TEST");
        //*******************************//

        SpringApplication springApplication =new SpringApplication(SOFABootWebSprin
gApplication.class);
        ApplicationContext applicationContext = springApplication.run(args);
```

```
        if(logger.isInfoEnabled()){
            printMsg("SofaRpc Application (myclient-app) started on 8084 port.");
        }

        //调用 SOFARPC 服务。
        ReferenceHolder referenceHolder = applicationContext.getBean(ReferenceHolde
r.class);
        final SampleService sampleService = referenceHolder.getSampleService();

        new Thread(newRunnable(){
            @Override
            public void run(){
                while(true){
                    try{
                        String response = sampleService.hello();
                        printMsg("Response from myserver-app: "+ response);
                    }catch(Exception e){
                        e.printStackTrace();
                    }finally{
                        try{
                            TimeUnit.SECONDS.sleep(3);
                        }catch(InterruptedException e){
                        }
                    }

                }

            }
        }).start();
    }
    private static void printMsg(String msg){
        System.out.println(msg);
        if(logger.isInfoEnabled()){
            logger.info(msg);
        }
    }
}
```

3. 配置包扫描。

```
@SpringBootApplication(scanBasePackages ={"com.alipay.mytestsofa","com.alipay.samples.r
pc"})
public class SOFABootWebSpringApplication{
...
}
```

4. 配置本地运行的端口。

   i. 在 `myserver-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12201` 。

   `rpc.tr.port` 是 TR 端口号，默认为 12200； `TR = TaobaoRemoting` 是 RPC 使用的底层通信框架。云端发布时不需要该项配置。

ii. 运行 Web 子模块中的 `SOFABootWebSpringApplication` 。

运行后，框架会自动进行服务的发布。为了避免端口冲突，需要在该类中指定端口。

```
System.setProperty("server.port","8083");
```

> **重要**
>
> 将 `myserver-app` 发布至云上环境前，必须注释掉代码中对 8083 端口的配置。

iii. 在 `myclient-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12202` 。

`rpc.tr.port` 是 TR 端口号，默认为 12200。云端发布时不需要该项配置。

5. 配置 `application.properties` 。

i. 登录 SOFAStack 控制台。

ii. 在左侧导航栏选择 **研发效能 > 脚手架**，然后在 **Step 2** 中获取如下信息：

- **实例标识**：应用实例在工作空间中的唯一标识。

  在 `application.properties` 中对应的 key 为 `com.alipay.instanceid` 。

- **AntVIP**：区域的唯一标识，每个区域一个地址。您可以在 SOFAStack 控制台左侧导航栏，单击 **中间件总览**，然后在 **元数据信息** 区域获取该信息。

  在 `application.properties` 中对应的 key 为 `com.antcloud.antvip.endpoint` 。

- **Access Key ID**：用于标识用户。单击 **获取 AK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.access` 。

- **Access Key Secret**：用于验证用户。单击 **获取 SK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.secret` 。

iii. 配置运行模式和运行环境，示例如下：

```
run.mode=NORMAL
com.alipay.env=shared
```

iv. 将上述步骤获取的参数配置在 `application.properties` 文件中。

> **说明**
>
> 更多信息，请参见 引入 SOFA 中间件。

示例工程

下文以示例工程为例，对 SOFARPC 的实现原理进行说明。

示例概述

通过 IDEA 或 Eclipse 分别打开 rpc-demo 中的 `myserver-app` 和 `myclient-app` 工程。

示例工程关键信息

- groupId：工程组织的唯一标识，示例工程为 `com.alipay.mytestsofa` 。
- artifactId：工程的构件标识符，示例工程为 `myserver-app` 或 `myclient-app` 。
- version：版本号，默认为 `1.0-SNAPSHOT` 。
- package：应用包名，默认等同于 groupId，工程示例为 `com.alipay.mytestsofa` 。

示例工程 Bean 配置

Bean 的配置分为服务发布和服务引用 2 个类型：

- 服务发布示例

```
@SofaService(interfaceType =SampleService.class,bindings =@SofaServiceBinding(bindingType
="bolt"))
```

- 服务引用示例

```
@SofaReference(interfaceType =SampleService.class,binding =@SofaReferenceBinding(bindingT
ype ="bolt",directUrl ="127.0.0.1:12201"))
privateSampleService sampleService;
```

> **说明**
>
> 示例中，为了便于对所有待引用实例进行统一管理，创建了 `ReferenceHolder` 类，进行引用管理。

示例工程原理

- 在 2 个工程的 endpoint 模块中相同位置，提供相同的服务接口和实现，并通过注解发现服务。2 个工程通过相同接口实现关联。一个客户端，一个服务端，如果是本地工程，在引用时，通过配置 directUrl，以直连方式发现服务；如果是服务器上部署测试，则通过 DSR（Direct Server Return） 底座发现服务。
- 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 可发布服务。
- 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 可引用服务。

示例工程服务验证

1. 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 发布服务。
2. 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 引用服务。

   引用成功后， `myclient-app` 控制台将输出：

   ```
   Response from myserver-app:HelloSOFARpc! times = xx
   ```

   您也可通过日志来查看服务引用结果。在 Web 子模块路径 `src/main/resources/config/application.properties` 中，可设置日志路径。默认在 `logs/myweb-app/common-default.log` 中查看服务引用结果。

应用打包和云端发布

1. 打包本地应用。

   操作步骤，请参见 编译运行。
2. 发布应用。
   - 应用整体发布流程，请参见 技术栈与应用发布流程。
   - 应用的详细发布步骤，建议根据发布方式，参考下述文档：
     - 经典应用服务，请参见 经典应用服务快速入门。
     - 容器应用服务，请参见 容器应用服务快速入门。

服务管控

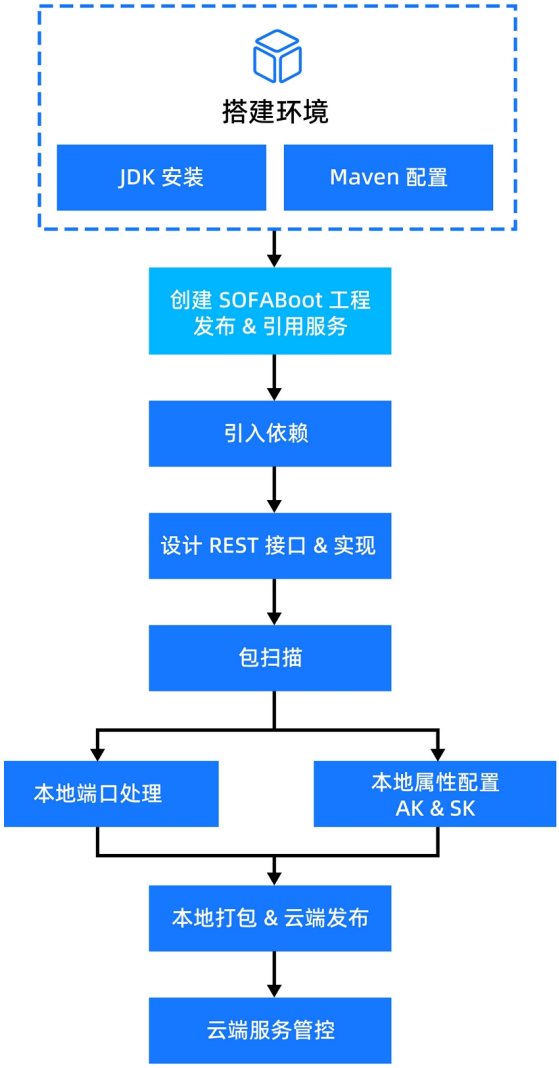
在 服务管控 页面查询并管控发布的 RPC 服务。更多详情，请参见 服务查看及管理。

3.3. REST 服务快速入门

微服务（SOFAStack MicroService）主要是通过 SOFARPC 来实现服务的发布和引用，而 SOFARPC 支持 REST 协议。本文以微服务本地开发到云端发布的整体流程为框架，让您了解如何在本地实现 SOFAREST 功能，以及如何将应用发布到云端，并在云端进行服务管控。本文主要讲述 SOFAREST 的实现原理，示例工程也支持直连的方式，让您在本地体验 SOFAREST 的服务发布和服务引用。

本地工程开发

SOFAREST 工程开发流程图



准备工作

- 搭建 SOFABoot 环境。具体操作，请参见 搭建环境。
- 通过以下任一方式生成 2 个 SOFABoot Web 工程，分别作为服务发布方和引用方。
  - 创建 2 个 SOFABoot Web 工程。具体操作，请参见 创建工程。
  - 直接下载 示例工程。下载后，请参考 版本说明 将工程根目录下 `pom.xml` 文件中的版本号修改为最新版本号。

本地开发流程

1. 引入依赖。

   按步骤创建的工程默认已经引入该依赖，请忽略此步骤；直接下载的示例工程需在 2 个本地 SOFABoot 工程 Web 模块 `pom.xml` 中引入 SOFARPC 的 Maven 依赖。

```
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>rpc-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

2. 编写业务逻辑。

   主要为服务发布和服务引用。本示例使用注解的方式配置 Bean，实现服务发布和引用。更多详情，请参见 使用注解方式。其它配置方式，请参见 使用 XML 配置 及 使用编程 API。

   - 服务发布的业务逻辑
     - 设计服务接口类

       本示例类名称为 `SampleRestFacade.java` ，接口路径为 `com.alipay.samples.rpc.SampleRestFacade` 。

```
@Path("/sofarest")  //注意该注解的继承性问题，实现类或方法中，该注解的缺失，可能会造成 SOFA REST 调用报 404 错误。
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
@Produces(MediaType.APPLICATION_JSON +";charset=UTF-8")
public interface SampleRestFacade{
    /**
    * http://localhost:8341/sofarest/hello
    */
    @GET
    @Path("/hello")
    public String hello();
}
```

     - 编写服务实现类

       本示例类名称为 `SampleRestFacadeImpl.java` ，接口实现路径为 `com.alipay.samples.rpc.impl.SampleRestFacadeImpl` 。

```
@Service
@SofaService(interfaceType =SampleRestFacade.class,bindings =@SofaServiceBinding(bindingType ="rest"))
public class SampleRestFacadeImpl implements SampleRestFacade{
      private int count =0;

      public SampleRestFacadeImpl(){
          System.out.println("print start");
      }

     @Override
     public String hello(){
          return "Hello SOFARest! times = "+ count++;
     }
}
```

   - 服务引用逻辑

- 引用对象注入

本示例采用 `ReferenceHolder` 类对引用对象进行统一管理，示例如下：

```
@Component
public class ReferenceHolder{
      @SofaReference(interfaceType =SampleRestFacade.class, binding =@SofaReference
Binding(bindingType ="rest",directUrl ="127.0.0.1:8341"))
      private SampleRestFacade sampleRestFacade;
      public SampleRestFacade getSampleRestFacade(){
          return sampleRestFacade;
      }
     public void setSampleRestFacade(SampleRestFacade sampleRestFacade){
         this.sampleRestFacade = sampleRestFacade;
     }
}
```

> **说明**
>
> bindingType 为 rest 协议，直连 URL 端口为 8341，即 `directUrl="127.0.0.1:8341"` 。

- 引用对象

为了方便直观使用，本示例将引用服务逻辑放在了 `myclient-app` 工程 Web 模块的 `com.alipay.mytestsofa.SOFABootWebSpringApplication` 中，示例如下：

```
@SpringBootApplication
public class SOFABootWebSpringApplication{
     private static final Logger logger =LoggerFactory.getLogger(SOFABootWebSpringA
pplication.class);

     public static void main(String[] args){
           //*************** 注意 ******************//
          //本地同时启动 myserver-app 和 myclient-app 时，由于 tomcat 端口冲突问题，需要
修改 myclient-app 的 端口号为 8084。
          //将 myserver-app 和 myclient-app 发布到云上环境时，由于默认健康检查端口是 808
0，所以需要注释掉该行代码。
         System.setProperty("server.port","8084");
         //由于本地启动没有注册中心，所以使用本地直连的方式访问本地启动的 myserver-app，发
布到线上的时候需要注释掉该行代码。
         System.setProperty("run.mode","TEST");
         //*********************************//

         SpringApplication springApplication =newSpringApplication(SOFABootWebSprin
gApplication.class);
         ApplicationContext applicationContext = springApplication.run(args);

          if(logger.isInfoEnabled()){
                printMsg("SofaRpc Application (myclient-app) started on 8084 port."
);
          }
```

```
          ReferenceHolder referenceHolder = applicationContext.getBean(ReferenceHol
der.class);
          //调用 SOFAREST 服务。
         final SampleRestFacade sampleRestFacade = referenceHolder.getSampleRestFac
ade();

          new Thread(newRunnable(){
             @Override
             public void run(){
                  while(true){
                          try{
                                  String response = sampleRestFacade.hello();
                                  printMsg("Response from myserver-app.rest: "+ re
sponse);
                          }catch(Exception e){
                                 e.printStackTrace();
                          }finally{
                                 try{
                                         TimeUnit.SECONDS.sleep(3);
                                  }catch(InterruptedException e){
                                      //ignore
                                 }
                            }
                      }
                 }
          }).start();
       }
     private static void printMsg(String msg){
          System.out.println(msg);
           if(logger.isInfoEnabled()){
                logger.info(msg);
            }
     }
}
```

3. 配置包扫描。

```
@SpringBootApplication(scanBasePackages ={"com.alipay.mytestsofa","com.alipay.samples.r
pc"})
public class SOFABootWebSpringApplication{
...
}
```

4. 配置本地运行的端口。

   i. 在 `myserver-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12201` 。

   `rpc.tr.port` 是 TR 端口号，默认为 12200； `TR = TaobaoRemoting` 是 RPC 使用的底层通信框架。云端发布时不需要该项配置。

ii. 运行 Web 子模块中的 `SOFABootWebSpringApplication` 。

运行后，框架会自动进行服务的发布。为了避免端口冲突，需要在该类中指定端口。

> 重要
>
> 将 `myserver-app` 发布至云上环境前，必须注释掉代码中对 8083 端口的配置。

iii. 在 `myclient-app` Web 模块 `application.properties` 文件中配置 `rpc.tr.port=12202` 。

`rpc.tr.port` 是 TR 端口号，默认为 12200。云端发布时不需要该项配置。

5. 配置 `application.properties` 。

i. 登录 SOFAStack 控制台。

ii. 在左侧导航栏选择 **研发效能 > 脚手架**，然后在 Step 2 中获取如下信息：

- **实例标识**：应用实例在工作空间中的唯一标识。

  在 `application.properties` 中对应的 key 为 `com.alipay.instanceid` 。

- **AntVIP**：区域的唯一标识，每个区域一个地址。应用通过 AntVIP 寻找所在区域环境的地址。不同环境的 AntVIP 地址值如下：

  - 杭州金区：`cn-hangzhou-fin-middleware-acvip-prod.cloud.alipaycs.net`
  - 上海非金：`cn-shanghai-middleware-acvip-prod.cloud.alipaycs.net`
  - 上海金区：`cn-shanghai-fin-sofastack-middleware-acvip-prod.cloud.alipaycs.net`
  - 杭州非金：`cn-hangzhou-middleware-acvip-prod.cloud.alipaycs.net`

  在 `application.properties` 中对应的 key 为 `com.antcloud.antvip.endpoint` 。

- **Access Key ID**：用于标识用户。单击 **获取 AK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.access` 。

- **Access Key Secret**：用于验证用户。单击 **获取 SK** 可前往 RAM 控制台获取。具体操作，请参见 创建 AccessKey。

  在 `application.properties` 中对应的 key 为 `com.antcloud.mw.secret` 。

iii. 配置运行模式和运行环境，示例如下：

```
run.mode=NORMAL
com.alipay.env=shared
```

iv. 将上述步骤获取的参数配置在 `application.properties` 文件中。

> 说明
>
> 更多信息，请参见 引入 SOFA 中间件。

示例工程

下文以示例工程为例，对 SOFARPC 的实现原理进行说明。

示例概述

通过 IDEA 或 Eclipse 分别打开 rpc-demo 中的 `myserver-app` 和 `myclient-app` 工程。

示例工程关键信息

- groupId: 工程组织的唯一标识，示例工程为 `com.alipay.mytestsofa` 。
- artifactId：工程的构件标识符，示例工程为 `myserver-app` 或 `myclient-app` 。
- version：版本号，默认为 `1.0-SNAPSHOT` 。
- package：应用包名，默认等同于 groupId，工程示例为 `com.alipay.mytestsofa` 。

示例工程 Bean 配置

Bean 的配置分为服务发布和服务引用 2 个类型：

- 服务发布的示例

```
@SofaService(interfaceType =SampleRestFacade.class,bindings =@SofaServiceBinding(bindingT
ype ="rest"))
```

- 服务引用的示例

```
@SofaReference(interfaceType =SampleRestFacade.class, binding =@SofaReferenceBinding(bind
ingType ="rest",directUrl ="127.0.0.1:8341"))
private SampleRestFacade sampleRestFacade;
```

> 说明
>
> 示例中，为了便于对所有待引用实例进行统一管理，创建了 `ReferenceHolder` 类，进行引用管理。

示例工程的 REST 实现

SOFAREST 的实现基于 SOFARPC，SOFARPC 的实现原理说明如下：

- 在 2 个工程的 endpoint 模块中相同位置，提供相同的服务接口和实现，并通过注解发现服务。2 个工程通过相同接口实现关联。一个客户端，一个服务端，如果是本地工程，在引用时，通过配置 directUrl，以直连方式发现服务；如果是服务器上部署测试，则通过 DSR（Direct Server Return） 底座发现服务。
- 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 可发布服务。

- 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 可引用服务。

示例工程服务验证

1. 启动 `myserver-app` Web 模块的 `SOFABootWebSpringApplication` 发布服务。
2. 启动 `myclient-app` Web 模块的 `SOFABootWebSpringApplication` 引用服务。

   引用成功后， `myclient-app` 控制台将输出：

   ```
   Response from myserver-app.rest:HelloSOFARest! times = xx
   ```

   本地浏览器访问 `http://localhost:8341/sofarest/hello` ，将输出访问 URL 链接那一刻，客户端的调用次数。

   ```
   HelloSOFARest! times = xx
   ```

云端发布后，可以在客户端命令行中输入 `curl http://{服务器 IP 地址}:8341/sofarest/hello` 命令进行验证。

您也可以通过日志来查看服务引用结果：默认在 `/logs/myclient-app/common-default.log` 中查看服务引用结果。您可以在 Web 子模块路径 `src/main/resources/config/application.properties` 中修改日志路径。

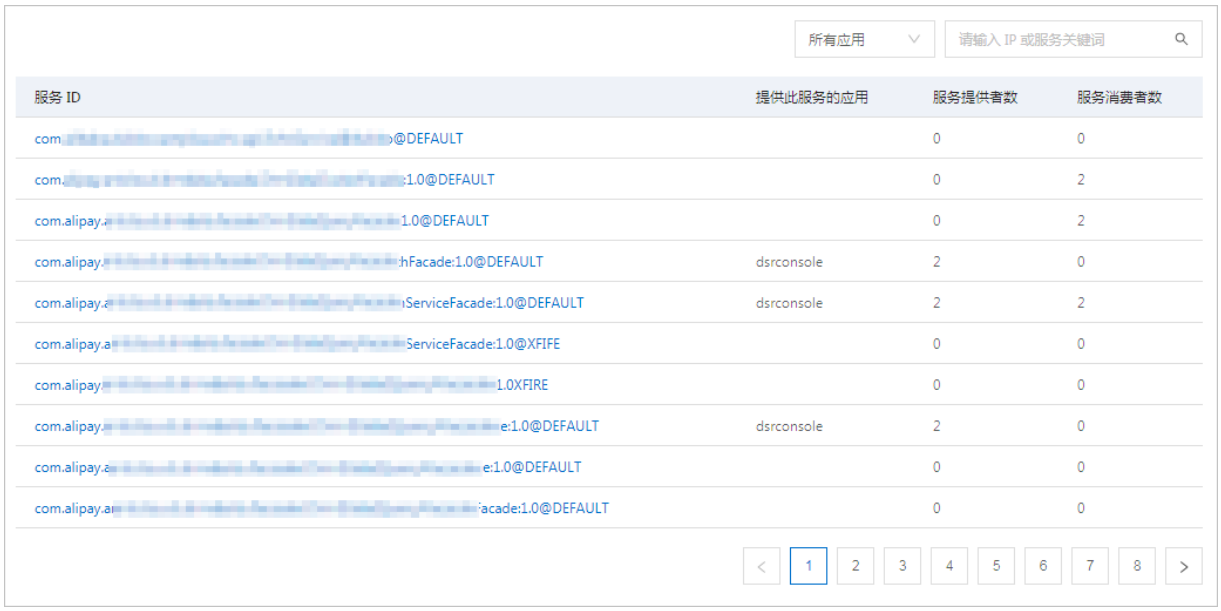
应用打包和云端发布

1. 打包本地应用。

   操作步骤，请参见 编译运行。
2. 发布应用。
   - 应用整体发布流程，请参见 技术栈与应用发布流程。
   - 应用的详细发布步骤，建议根据发布方式，参考下述文档：
     - 经典应用服务，请参见 经典应用服务快速入门。
     - 容器应用服务，请参见 容器应用服务快速入门。

服务管控

在 服务管控 页面查询并管控发布的 RPC 服务。更多详情，请参见 服务管控。



3.4. 服务发布与引用

3.4.1. 发布 SOFARPC 服务

RPC 是日常开发中最常用的中间件，本文主要说明如何发布一个 RPC 服务。

前提条件

- 已完成环境搭建。具体步骤，请参见 搭建环境。
- 已下载 示例工程。
- 已将工程导入 IDE 工具。具体操作，请参见 将 Web 工程导入 IDE。

发布 SOFARPC 服务

要发布一个 RPC 服务，主要步骤说明如下：

1. 在 `app/endpoint/` 下设计接口，示例如下：

```
// com.alipay.APPNAME.facade.SampleService
public interface SampleService{
    String message();
}
```

2. 提供一个接口的实现类，示例如下：

```
// com.alipay.APPNAME.service.SampleServiceImpl
public class SampleServiceImpl implements SampleService{
   @Override
   public String message(){
        return "Hello, Service SOFABootRPC!";
   }
}
```

3. 在接口所在模块中，通过 `resources/META-INF` 下的 xml 文件，将接口实现类配置为一个 Java bean。

```
<bean id="sampleServiceBean" class="com.alipay.APPNAME.service.SampleServiceImpl"/>
```

4. 在接口所在模块中，通过 `resources/META-INF` 下的 xml 文件，发布 RPC 服务。根据接口实现类的个数，可以分为下述 2 种：
   - 单接口单实现，示例如下：

```
<!-- 服务 -->
<sofa:service ref="sampleServiceBean" interface="com.alipay.APPNAME.facade.SampleServ
ice">
<sofa:binding.bolt/>
</sofa:service>
```

   - 单接口多实现，示例如下：

```
<!-- 服务一 -->
<sofa:service ref="sampleServiceBean1" interface="com.alipay.APPNAME.facade.SampleSer
vice"unique-id="service1">
<sofa:binding.bolt/>
</sofa:service>

<!-- 服务二 -->
<sofa:service ref="sampleServiceBean2" interface="com.alipay.APPNAME.facade.SampleSer
vice"unique-id="service2">
<sofa:binding.bolt/>
</sofa:service>
```

> **说明**
>
> - 服务提供方定义服务 `<sofa:service>`，并进行服务发布；服务消费方定义服务引用 `<sofa:reference>`，并进行服务引用。任何一个 SOFA 框架应用节点都可以同时发布服务和引用其它节点的服务。
> - RPC 服务提供方通过服务 `<sofa:service>` 来定义，主要属性有 `interface`、`unique-id` 和 `ref`。
>   - `interface`：该属性用于确定一个服务，属性值为：命名空间包名 + Java 接口名。
>   - `unique-id`：如果同一个接口有两个不同的实现，而这两个不同的实现都需要发布成 SOFA 的 RPC 服务，则可以在发布服务的时候加上一个 `unique-id` 属性来进行区分。
>   - `ref`：该属性用于指定服务实现所对应的 Spring Bean，通过 Bean ID 和服务实现类进行关联。

本地运行

1. 在工程的根目录下执行 `mvn clean install` 命令，在 `target` 目录下生成一个可执行的 `Fat JAR` 文件，例如：`APPNAME-service-1.0-SNAPSHOT-executable.jar`。
2. 通过以下任一方法执行 JAR 文件，如果没有错误日志输出，则表示 `sofaboot-rpc-server` 工程启动成功。
   - 在命令行中执行 `java -jar APPNAME-service-1.0-SNAPSHOT-executable.jar` 命令。
   - 在本地 IDE 中直接运行 `main` 函数。

云端运行

SOFARPC 在本地只能通过直连方式进行体验。只有在云端发布成功后，通过 SOFAStack 控制台，才能进行服务管控和治理。具体云端发布步骤，请参考下述信息：

- 对于应用的整体发布流程，请参见 技术栈与应用发布流程。
- 应用的详细发布步骤，建议根据发布方式，参考下述文档：
  - 经典应用服务，请参见 经典应用服务快速入门。
  - 容器应用服务，请参见 容器应用服务快速入门。

> 说明
>
> 云端发布时，您必须在 `application.properties` 中配置以下属性：
>
> - `run.mode=NORMAL`
> - `com.alipay.env=shared`
> - `com.alipay.instanceid=`
> - `com.antcloud.antvip.endpoint=`
> - `com.antcloud.mw.access=`
> - `com.antcloud.mw.secret=`
>
> 运行模式和运行环境的值为默认固定值。参数的具体配置，请参见 properties 配置项。

日志查看

1. 查看本地工程 RPC 发布服务启动日志 `logs/rpc/common-default.log` 。

   如出现类似以下内容，说明 RPC 服务端成功启动：

   ```
   2016-12-1715:16:44,466 INFO  main                                - sofa rpc run.mode = DEV
   2016-12-1715:16:49,479 INFO  main                                - PID:42843 sofa rpc star
   ting!
   ```

2. 查看日志 `logs/rpc/rpc-registry.log` ，内容参考如下：

   ```
   2016-12-1715:17:07,764 INFO  main                                RPC-REGISTRY -发布 RPC 服
   务：服务名[com.alipay.APPNAME.facade.SampleService:1.0@DEFAULT]
   ```

3. 查看错误日志 `logs/rpc/common-error.log` 。

   如果没有任何错误日志输出且应用启动正常，说明成功发布了一个 RPC 服务。

关于日志的更多详情，请参见 日志说明。

3.4.2. 引用 SOFARPC 服务

RPC 是日常开发中最常用的中间件，本文主要说明如何引用一个 RPC 服务。

前提条件

- 已完成环境搭建。具体步骤，请参见 搭建环境。
- 已下载 示例工程。
- 已将工程导入 IDE 工具。具体操作，请参见 将 Web 工程导入 IDE。

引入接口定义依赖

要引用一个 RPC 服务，您需要知道 RPC 服务的提供方所发布的接口是什么（如果发布的服务有 `unique-id`，您还需要知道 `unique-id`），这就要求服务提供方将发布的接口所在的 JAR 包及依赖信息传到 Maven 仓库，以便服务引用方能够引用服务提供方所发布的 RPC 服务。

- 如果是本地运行，需要在 `sofaboot-rpc-server` 工程目录运行 `mvn clean install`，将接口依赖的 JAR 安装到本地仓库。
- 若非本地运行，需要将接口依赖 JAR 上传到对应的 Maven 仓库。

获得 RPC 服务的发布接口后，在 `sofaboot-rpc-client` 工程下的主 pom 中添加所引用的 RPC 服务的接口依赖信息，示例如下：

```
<dependency>
    <groupId>com.alipay.APPNAME</groupId>
    <artifactId>APPNAME-facade</artifactId>
     <version>1.0-SNAPSHOT</version>
</dependency>
```

> **说明**
>
> 此处客户端和服务端的应用名称均命名为 APPNAME，仅供学习使用。在实际环境中，两个应用名称不能完全一样。

引用 RPC 服务

在配置文件 `META-INF/APPNAME/APPNAME-web.xml` 中，根据接口配置引用一个 RPC 服务：

```
<sofa:reference id="sampleRpcService" interface="com.alipay.APPNAME.facade.SampleService">
<sofa:binding.bolt/>
</sofa:reference>
```

此处的 RPC 引用也是一个 bean，其 bean id 为 `sampleRpcService`。

将引用的 RPC 服务注入 Controller

> **重要**
>
> 这一步是为了演示方便，实现用户通过浏览器或者其他方式访问一个 rest 接口，触发调用所引用的服务，并调用到服务端。实际开发中，并不需要注入 Controller 这一步。

本教程将这个 RPC 服务注入到了 `com.alipay.APPNAME.web.springrest.RpcTestController` 中，示例如下：

```
@RestController
@RequestMapping("/rpc")
public class RpcTestController{
    @Autowired
     private SampleService sampleRpcService;
     @RequestMapping("/hello")
     String rpcUniqueAndTimeout(){
          String rpcResult =this.sampleRpcService.message();
          return rpcResult;
     }
}
```

本地编译

`sofaboot-rpc-client` 是一个 SOFABoot Web 工程。依次执行以下命令以进行本地编译并启动 RPC Client：

1. 将客户端和服务端 `config/application.properties` 中的 `run.mode` 均配置为 `DEV`，即 `run.mode=DEV`。
2. 在工程根目录下执行 `mvn clean install` 命令，生成可执行文件 `target/APPNAME-web-1.0-SNAPSHOT-executable.jar`。
3. 在工程根目录下执行 `java -jar ./target/APPNAME-web-1.0-SNAPSHOT-executable.jar` 命令。
   - 如果控制台输出如下信息，则表示 Web 容器启动成功。

     ```
     16:11:13.625 INFO  org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedSer
     vletContainer-Tomcat started on port(s):8080(http)
     ```

   - 如果输出错误信息，请在解决问题后重试以上步骤。

测试 RPC 服务

1. 启动 RPC 服务端 `sofaboot-rpc-server` 及客户端 `sofaboot-rpc-client`。
2. 在浏览器中访问 `http://localhost:8080/rpc/hello` 来测试引用的 RPC 服务。

   当浏览器输出如下信息时，表示 RPC 服务发布和引用均成功。

   ```
   Hello,ServiceSOFABoot
   ```

云端运行

SOFARPC 在本地只能通过直连方式进行体验。只有在云端发布成功后，通过 SOFAStack 控制台，才能进行服务管控和治理。具体云端发布步骤，请参考下述信息：

- 对于应用的整体发布流程，请参见 技术栈与应用发布流程。
- 应用的详细发布步骤，建议根据发布方式，参考下述文档：
  - 经典应用服务，请参见 经典应用服务快速入门。

- 容器应用服务，请参见 容器应用服务快速入门。

> **说明**
>
> 云端发布时，您必须在 `application.properties` 中配置以下属性：
>
> - `run.mode=NORMAL`
> - `com.alipay.env=shared`
> - `com.alipay.instanceid=`
> - `com.antcloud.antvip.endpoint=`
> - `com.antcloud.mw.access=`
> - `com.antcloud.mw.secret=`
>
> 运行模式和运行环境的值为默认固定值。其他参数的具体配置，请参见 properties 配置项。

日志查看

查看 `sofaboot-rpc-client` 工程引用 RPC 服务启动日志 `logs/rpc/rpc-registry.log` ，出现类似如下日志时，说明 RPC 服务引用成功。

```
2016-12-1715:45:50,340 INFO  main                                RPC-REGISTRY -订阅 RPC 服务：服务名[com.alipay.APPNAME.facade.SampleService:1.0@DEFAULT]
```

如果发现引用 RPC 服务失败，请重点关注日志目录 `logs` 下的所有 `common-error.log` 。

有关日志的详细信息，请参考 日志说明。

3.5. 配置说明

3.5.1. 配置方式

SOFARPC 的服务发布和引用方式包括使用注解方式、使用 XML 配置方式和使用编程 API 方式。

使用注解方式

SOFABoot 环境支持使用注解方式，包括以下两种：

- 单协议注解：`@SofaService` 和 `@SofaReference` .
- 多协议注解：增加注解 `@SofaServiceBinding` 和 `@SofaReferenceBinding` 。

服务发布

如果要发布一个 RPC 服务，只需要在 Bean 上面打上 `@SofaService` 注解，指定接口和协议类型即可。

```
@SofaService(interfaceType =AnnotationService.class, bindings ={@SofaServiceBinding(binding
Type ="bolt")})
@Component
public class AnnotationServiceImpl implements AnnotationService{
     @Override
     public String sayAnnotation(String helloAnno){
          return helloAnno;
     }
}
```

服务引用

对于需要引用远程服务的 Bean，只需要在属性或者方法上打上 `Reference` 的注解即可，支持 Bolt、Dubbo、REST 协议。

```
@Component
public class AnnotationClientImpl{

     @SofaReference(interfaceType =AnnotationService.class, binding =@SofaReferenceBinding(
bindingType ="bolt"))
     private AnnotationService annotationService;

      public String sayClientAnnotation(String clientAnno){

             String result = annotationService.sayAnnotation(clientAnno);

             return result;
       }
}
```

使用 XML 配置

XML 配置中主要标签含义如下：

- `sofa:service` ：表示发布服务。
- `sofa:reference` ：表示引用服务。
- `sofa:binding` ：表示服务发布或引用的协议。

> **说明**
>
> XML 配置完整示例，请参见 示例工程。

服务发布示例

- 单协议发布

```
<bean id="personServiceImpl" class="com.alipay.sofa.boot.examples.demo.rpc.bean.PersonSer
viceImpl"/>
<sofa:service ref="personServiceImpl" interface="com.alipay.sofa.boot.examples.demo.rpc.b
ean.PersonService">
   <sofa:binding.bolt/>
</sofa:service>
```

- 多协议发布

```
<sofa:service ref="personServiceImpl" interface="com.alipay.sofa.boot.examples.demo.rpc.b
ean.PersonService">
    <sofa:binding.bolt/>
    <sofa:binding.rest/>
    <sofa:binding.dubbo/>
</sofa:service>
```

服务引用示例

- Bolt 协议引用

```
<sofa:reference id="personReferenceBolt" interface="com.alipay.sofa.boot.examples.demo.rp
c.bean.PersonService">
     <sofa:binding.bolt/>
</sofa:reference>
```

- REST 协议引用

```
<sofa:reference id="personReferenceRest" interface="com.alipay.sofa.boot.examples.demo.rp
c.bean.PersonService">
    <sofa:binding.rest/>
</sofa:reference>
```

使用编程 API

SOFA 提供一套机制去存放各种组件的编程 API，并提供一套统一的方法，让您可以获取到这些 API。组件编程 API 的存放与获取均通过 SOFA 的 ClientFactory 类进行，通过这个 ClientFactory 类，可以获取到对应组件的编程 API。

前提条件

使用 SOFA 组件编程相关的 API，请确保使用的模块里面已经添加了如下的依赖：

```
<dependency>
     <groupId>com.alipay.sofa</groupId>
     <artifactId>rpc-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

配置方式

SOFA 提供两种方式获取 `ClientFactory` ：

- 实现 ClientFactoryAware 接口

  代码示例如下：

```
public class ClientFactoryBean implements ClientFactoryAware{
      private ClientFactory clientFactory;

      @Override
      public void setClientFactory(ClientFactory clientFactory){
          this.clientFactory = clientFactory;
      }

     public ClientFactory getClientFactory(){
          return clientFactory;
     }
}
```

然后，将上面的 `ClientFactoryBean` 配置成一个 Spring Bean。

```
<bean id="clientFactoryBean" class="com.alipay.test.ClientFactoryBean"/>
```

完成后， `ClientFactoryBean` 就可以获取到 `clientFactory` 对象来使用了。

- 使用 @SofaClientFactory 注解

  代码示例如下：

```
public class ClientAnnotatedBean{
      @SofaClientFactory
      private ClientFactory clientFactory;

      public ClientFactory getClientFactory(){
           return clientFactory;
      }
}
```

  您需要在 `ClientFactory` 字段上加上 `@SofaClientFactory` 的注解，然后将 `ClientAnnotatedBean` 配置成一个 Spring Bean 。

```
<bean id="clientAnnotatedBean" class="com.alipay.test.ClientAnnotatedBean"/>
```

  完成后，SOFA 框架就会自动将 `ClientFactory` 的实例注入到被 `@SofaClientFactory` 注解的字段上。

  以上操作只是获取到 `ClientFactory` 这个对象，如果要获取特定的客户端（如 `ServiceFactory` ），您还需调用 `ClientFactory` 的 `getClient` 方法。SOFA 对 `@SofaClientFactory` 的注解进行了增强，可以直接通过 `@SofaClientFactory` 来获取具体的 Client，代码示例如下：

```
public class ClientAnnotatedBean{
    @SofaClientFactory
    private ServiceClient serviceClient;

    public ServiceClient getServiceClient(){
        return serviceClient;
    }
}
```

当 `@SofaClientFactory` 直接使用在具体的 Client 对象上时，此注解可以直接将对应的 Client 对象注入到被注解的字段上。在上述例子中，`@SofaClientFactory` 是直接注解在类型为 `ServiceClient` 的字段上，SOFA 框架会直接将 `ServiceClient` 对象注入到这个字段上。所有在 `ClientFactory` 中存在的对象，都可以通过此种方式来获得。

编程 API 示例

服务的发布与订阅不仅可以通过在 XML 中配置 Spring Bean 的方式在应用启动期静态加载，也可以采用编程 API 的方式在应用运行期动态执行，用法如下：

Bolt 服务发布

```
ServiceClient serviceClient = clientFactory.getClient(ServiceClient.class);

ServiceParam serviceParam = new ServiceParam();
serviceParam.setInstance(sampleService);
serviceParam.setInterfaceType(SampleService.class);
serviceParam.setUniqueId(uniqueId);

TrBindingParam trBinding = new TrBindingParam();
// 对应 global-attrs 标签。
trBinding.setClientTimeout(5000);

serviceParam.addBindingParam(trBinding);

serviceClient.service(serviceParam);
```

Bolt 服务引用

```
ReferenceClient referenceClient = clientFactory.getClient(ReferenceClient.class);
ReferenceParam<SampleService> referenceParam = new ReferenceParam<SampleService>();
referenceParam.setInterfaceType(SampleService.class);
referenceParam.setUniqueId(uniqueId);

TrBindingParam trBinding = new TrBindingParam();
// 对应 global-attrs 标签。
trBinding.setClientTimeout(8000);

// 对应 method 标签。
TrBindingMethodInfo trMethodInfo = new TrBindingMethodInfo();
trMethodInfo.setName("helloMethod");
trMethodInfo.setType("callback");
// 对象必须实现 com.alipay.sofa.rpc.api.callback.SofaResponseCallback 接口。
trMethodInfo.setCallbackHandler(callbackHandler);
trBinding.addMethodInfo(trMethodInfo);

referenceParam.setBindingParam(trBinding);

SampleService proxy = referenceClient.reference(referenceParam);
```

警告

通过动态客户端创建 SOFA Reference 返回的对象是一个非常重要的对象，在使用的时候不要频繁创建，自行做好缓存，否则可能存在内存溢出的风险。 因为编程 API 的类不能轻易变化，类名为了兼容以前的用法，保持 TR 写法，但实际其中走的是 Bolt 协议。

3.5.2. 应用维度配置

本文介绍 SOFARPC 可用的配置项和常见配置。

基础配置项说明

您可以根据自身环境的需求，在 `application.properties` 文件中添加以下配置项：

配置项	类型	说明	默认值
`sofa_runtime_local_mode`	BOOLEAN	本地优先调用开关。	false
`run_mode`	STRING	RPC 运行模式。	空
`rpc_tr_port`	INTEGER	TR 端口号。	12200

配置项	类型	说明	默认值
rpc_bind_network_interface	STRING	服务器绑定固定网卡。	空
rpc_enabled_ip_range	STRING	服务器绑定本地 IP 范围。	空
rpc_min_pool_size_tr	INTEGER	TR 服务器线程池最小线程数。	20
rpc_max_pool_size_tr	INTEGER	TR 服务器线程池最大线程数。	200
rpc_pool_queue_size_tr	INTEGER	TR 服务器线程池队列大小。	0
com.alipay.sofa.rpc.bolt.port	INTEGER	Bolt 端口号。	12200
com.alipay.sofa.rpc.bolt.thread.pool.core.size	INTEGER	Bolt 服务器线程池最小线程数。	20
com.alipay.sofa.rpc.bolt.thread.pool.max.size	INTEGER	Bolt 服务器线程池最大线程数。	200
com.alipay.sofa.rpc.bolt.thread.pool.queue.size	INTEGER	Bolt 服务器线程池队列大小。	0
com.alipay.sofa.rpc.rest.port	INTEGER	SOFAREST 端口号。	8341
rpc_transmit_url	STRING	预热与权重配置。	空

配置项	类型	说明	默认值
rpc_transmit_url_timeout_tr	INTEGER	预热调用超时时间，单位 ms。	0
rpc_profile_threshold_tr	INTEGER	RPC 服务处理性能日志打印阈值，单位 ms。	300

本地优先调用模式

当本地启动多个 SOFA 应用时，要使这几个应用能优先相互调用，而不需要经过软负载过程，只需要在 `application.properties` 中加入 `sofa_runtime_local_mode=true` 即可。

但是 `sofa_runtime_local_mode` 这个配置依然需要依赖于配置中心推送下来的地址。拿到服务提供方地址列表后，服务消费方会优先选择本地的 IP 地址进行服务调用。如果开发者所处的工作空间没有配置中心，则需要指定服务提供方地址进行调用，具体参见 路由与配置中心。

```
application.properties:

run_mode=TEST

<!--服务应用方配置-->
<sofa:reference ...>
     <sofa:binding.bolt>
         <global-attrs test-url="localhost:12200"/>
     </sofa:binding.bolt>
</sofa:reference>
```

IP 或网卡绑定

SOFARPC 发布服务地址的时候，只会选取本地的第一张网卡的 IP 发布到配置中心，如果有多张网卡（如在 SOFAStack 平台上，有内网 IP 和外网 IP），则需要设置 IP 选择策略。

SOFARPC 提供了两种方式选择 IP：

- `rpc_bind_network_interface`

  指定具体的网卡名进行选择，如：`rpc_bind_network_interface=eth0` 。

- `rpc_enabled_ip_range`

  指定 IP 范围进行绑定，格式：`IP_RANGE1:IP_RANGE2,IP_RANGE` 。例如，`rpc_enabled_ip_range=10.1:10.2,11` 表示 IP 范围在 `10.1.0.0~10.2.255.255` 和 `11.0.0.0~11.255.255.255` 内的才会选择。

> 说明
>
> SOFAStack 平台的内网地址均绑定在 eth0 网卡上，推荐直接使用 `rpc_bind_network_interface=eth0` 配置。如果应用运行在其它非 SOFAStack 平台上，请查看运行机器的内网地址自行修改。查看机器地址的命令如下：
>
> - Windows 系统：`ipconfig`
> - Mac 和 Linux 系统：`ifconfig`

TR 线程池配置

在 `application.properties` 文件中使用以下选项配置 TR 线程池信息：

- `com.alipay.sofa.rpc.bolt.thread.pool.core.size`：最小线程数，默认 20。
- `com.alipay.sofa.rpc.bolt.thread.pool.max.size`：最大线程数，默认 200。
- `com.alipay.sofa.rpc.bolt.thread.pool.queue.size`：队列大小，默认 0。

TR 采用了 JDK 中的线程池 `ThreadPoolExecutor`。当核心线程池扩张时，先涨到最小线程数大小。当并发请求达到最小线程数后，请求被放入线程池队列中。队列满了之后，线程池会扩张到最大线程数指定的大小。如果超过最大线程数则会抛出 `RejectionException` 异常。

3.5.3. 应用维度配置扩展

在 SOFABoot 的使用场景下，RPC 框架在应用层面提供一些配置参数，如端口、线程池等信息。

应用参数都是通过 `Spring Boot@ConfigurationProperties` 进行的绑定，绑定属性类为 `com.alipay.sofa.rpc.boot.config.SofaBootRpcProperties`，配置前缀如下：

```
static final String PREFIX ="com.alipay.sofa.rpc";
```

您可以根据需要，在 `application.properties` 文件中增加以下配置项：

> 说明
>
> 您也可以根据自己的编码习惯按照 Spring Boot 的规范，使用驼峰、中划线等进行书写。

```
# 单机故障剔除
com.alipay.sofa.rpc.aft.regulation.effective # 是否开启单机故障剔除功能。
com.alipay.sofa.rpc.aft.degrade.effective  # 是否开启降级。
com.alipay.sofa.rpc.aft.time.window # 时间窗口。
com.alipay.sofa.rpc.aft.least.window.count # 最小调用次数。
com.alipay.sofa.rpc.aft.least.window.exception.rate.multiple # 最小异常率。
com.alipay.sofa.rpc.aft.weight.degrade.rate # 降级速率。
```

```
com.alipay.sofa.rpc.aft.weight.recover.rate # 恢复速率。
com.alipay.sofa.rpc.aft.degrade.least.weight #降级最小权重。
com.alipay.sofa.rpc.aft.degrade.max.ip.count # 最大降级 IP。

# bolt
com.alipay.sofa.rpc.bolt.port # bolt 端口。
com.alipay.sofa.rpc.bolt.thread.pool.core.size # bolt 核心线程数。
com.alipay.sofa.rpc.bolt.thread.pool.max.size # bolt 最大线程数。
com.alipay.sofa.rpc.bolt.thread.pool.queue.size # bolt 线程池队列。
com.alipay.sofa.rpc.bolt.accepts.size # 服务端允许客户端建立的连接数。

# rest
com.alipay.sofa.rpc.rest.hostname # rest hostname。
com.alipay.sofa.rpc.rest.port # rest port。
com.alipay.sofa.rpc.rest.io.thread.size # rest io 线程数。
com.alipay.sofa.rpc.rest.context.path # rest context path。
com.alipay.sofa.rpc.rest.thread.pool.core.size # rest 核心线程数。
com.alipay.sofa.rpc.rest.thread.pool.max.size # rest 最大线程数。
com.alipay.sofa.rpc.rest.max.request.size # rest 最大请求大小。
com.alipay.sofa.rpc.rest.telnet # 是否允许 rest telnet
com.alipay.sofa.rpc.rest.daemon # 是否hold住端口，true的话随主线程退出而退出。

# dubbo
com.alipay.sofa.rpc.dubbo.port # dubbo port。
com.alipay.sofa.rpc.dubbo.io.thread.size # dubbo io 线程大小。
com.alipay.sofa.rpc.dubbo.thread.pool.max.size # dubbo 业务线程最大数。
com.alipay.sofa.rpc.dubbo.accepts.size # dubbo 服务端允许客户端建立的连接数。
com.alipay.sofa.rpc.dubbo.thread.pool.core.size #dubbo 核心线程数。
com.alipay.sofa.rpc.dubbo.thread.pool.queue.size #dubbo 最大线程数。

# registry
com.alipay.sofa.rpc.registry.address # 注册中心地址。
com.alipay.sofa.rpc.virtual.host # virtual host。
com.alipay.sofa.rpc.bound.host # 绑定 host。
com.alipay.sofa.rpc.virtual.port # virtual 端口。
com.alipay.sofa.rpc.enabled.ip.range # 多网卡 IP 范围。
com.alipay.sofa.rpc.bind.network.interface # 绑定网卡。

# h2c
com.alipay.sofa.rpc.h2c.port # h2c 端口。
com.alipay.sofa.rpc.h2c.thread.pool.core.size # h2c 核心线程数。
com.alipay.sofa.rpc.h2c.thread.pool.max.size # h2c 最大线程数。
com.alipay.sofa.rpc.h2c.thread.pool.queue.size # h2c 队列大小。
com.alipay.sofa.rpc.h2c.accepts.size # 服务端允许客户端建立的连接数。

# 扩展
com.alipay.sofa.rpc.lookout.collect.disable # 是否关闭 lookout。

# 代理
com.alipay.sofa.rpc.consumer.repeated.reference.limit # 允许客户端对同一个服务生成的引用代理数量，默认为3。

# 注册中心
<sofa:binding.bolt>
```

```
<sofa:global-attrs registry="mesh" />  # 指定服务的注册中心为 mesh。
</sofa:binding.bolt>
```

> **说明**
>
> 服务注册中心默认为 DSR，若您有特殊需求需要变更，请参见 注册中心路由。

3.5.4. 服务维度配置

本文介绍在 SOFABoot 环境下完整的 SOFARPC 服务发布与引用说明。

发布服务

```
<bean id="helloSyncServiceImpl" class="com.alipay.sofa.rpc.samples.invoke.HelloSyncServiceImpl"/>
<sofa:service ref="helloSyncServiceImpl" interface="com.alipay.sofa.rpc.samples.invoke.HelloSyncService"unique-id="">
<sofa:binding.bolt>
<sofa:global-attrs registry="" serialize-type="" filter="" timeout="3000" thread-pool-ref=""
warm-up-time="60000"
warm-up-weight="10" weight="100"/>
</sofa:binding.bolt>
<sofa:binding.rest>
</sofa:binding.rest>
</sofa:service>
```

属性说明：

属性	默认值	说明
id	Bean 名	ID
class	-	类
ref	-	服务接口实现类
interface	-	服务接口（唯一标识元素）
unique-id	-	服务标签（唯一标识元素）
filter	-	过滤器配置别名

属性	默认值	说明
registry	DSR	服务端注册中心。多个注册中心需使用逗号分隔。 支持的注册中心请参见 注册中心路由。
timeout	3000	服务端执行超时时间 单位：毫秒。
serialize-type	hessian2	序列化协议，取值为：hessian2、protobuf。
thread-pool-ref	-	自定义业务线程池，需要指向一个 `com.alipay.sofa.rpc.core.context.UserThreadPool` 对象。
weight	100	服务静态权重
warm-up-weight	10	服务提供者的预热权重，当客户端刚建立连接时，在指定的预热时间内会使用此权重。
warm-up-time	0	服务提供者的预热时间，当客户端刚建立连接时，在该时间内会使用预热权重。 单位：毫秒。

引用服务

```
<sofa:reference jvm-first="false" id="helloSyncServiceReference"
interface="com.alipay.sofa.rpc.samples.invoke.HelloSyncService" unique-id="">
<sofa:binding.bolt>
<sofa:global-attrs type="sync" timeout="3000" callback-ref="" callback-class="" address-wai
t-time="1000"
connect.num="1" check="false" connect.timeout="1000" filter="" generic-interface=""
idle.timeout="1000"
idle.timeout.read="1000" lazy="false" loadBalancer="" registry="" retries="1"
serialize-type=""/>
<sofa:route target-url="xxx:12200"/>
<sofa:method name="hello" callback-class="" callback-ref="" timeout="3000" type="sync"/>
</sofa:binding.bolt>
</sofa:reference>
```

属性说明：

属性	默认值	说明
id	自动生成	ID
jvm-first	true	是否优先使用本地 JVM 配置，取值：true、false。
interface	-	服务接口（唯一标识元素） 无论是普通调用还是返回调用，都必须设置实际的接口类。
unique-id	-	服务标签（唯一标识元素）
type	sync	调用方式。取值为：sync、oneway、callback、future。
filter	-	过滤器配置别名
registry	DSR	服务端注册中心。多个注册中心需使用逗号分隔。 支持的注册中心请参见 注册中心路由。

属性	默认值	说明
method	-	方法级配置，`method` 配置的属性优先级高于 `global-attrs` 中的配置。没有配置时，默认跟随全局配置。
serialize-type	hessian2	序列化协议，取值为：hessian2、protobuf。
target-url	-	使用直连调用时配置的直连地址。 配置 `target-url` 后，软负载阶段将会失效。详情请参见 直连调用。
generic-interface	-	泛化接口
connect.timeout	1000	消费方连接超时时间 单位：毫秒。
connect.num	-1	消费方连接数 -1 表示不设置（默认为每个目标地址建立一个连接）。
idle.timeout	-1	消费方最大空闲时间 -1 表示使用底层默认值（底层默认值为 0，表示永远不会读 idle）。该配置也是心跳的时间间隔，当请求 idle 时间超过配置时间后，发送心跳到服务方。

属性	默认值	说明
idle.timeout.read	-1	消费方最大读空闲时间 -1 表示使用底层默认值（底层默认值为 30）。如果 `idle.timeout > 0`，在 `idle.timeout` + `idle.timeout.read` 时间内，如果没有请求发生，将自动断开连接。
loadBalancer	random	负载均衡算法
lazy	false	是否延迟建立长连接，取值：true、false。
address-wait-time	0	reference 生成时，等待配置中心将地址推送到消费方的时间。 单位：毫秒。 最大值为 30000 毫秒，超过这个值时会被调整为 30000 毫秒。
timeout	3000(cover 5000)	调用超时时间
retries	0	当出现非业务异常时，框架层面自动重试的次数。需要服务提供者自己保证方法的幂等性。
callback-class	-	callback 回调类 callback 时才可用。
callback-ref	-	callback 回调类 callback 时才可用。

配置的优先级

SOFARPC 里面的某些配置，例如调用超时 `timeout` 属性可以在服务提供方设置，也可以在服务调用方设置。这些配置的优先级从高到低排列如下：

1. 线程调用级别设置。
2. 服务调用方。
   - 方法级别设置。
   - Reference 级别设置。
3. 服务提供方。
   - 方法级别设置。
   - Service 级别设置。

3.5.5. 获取环境参数

本文介绍获取环境变量的方法。

获取实例标识和 AntVIP 地址

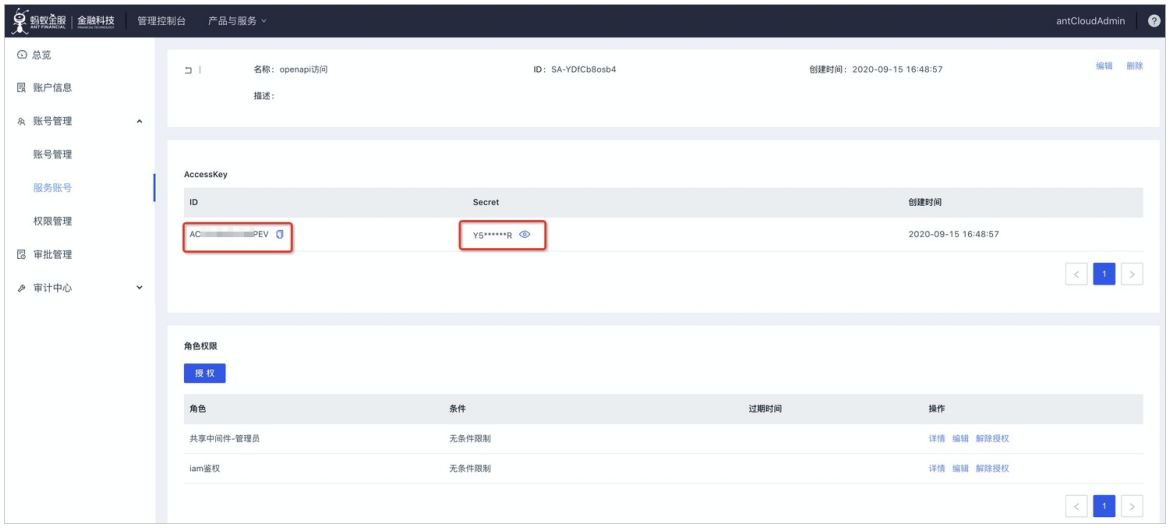
1. 登录中间件管理控制台。
2. 在 **金融级分布式中间件 SOFA** 控制台页面中，查看 **实例标识** 和 **AntVIP 地址**。

获取 AK（AccessKey ID）和 SK（AccessKey Secret）

1. 生成服务账号。
   i. 登录 IAM 控制台。
   ii. 在左侧导航栏，选择 **账号管理 > 服务账号**。
   iii. 单击 **新增服务账号**，并在弹出的对话框中输入服务账号的名称和描述。
   iv. 单击 **OK**。
2. 为账号授予权限。
   i. 单击刚刚创建的账号右侧的 **授权**。
   ii. 按照步骤给账号授予合适的中间件权限。

   您可以给账号授予中间件开发者或管理员权限，或根据自己需要创建合适的权限授予该服务账号。
3. 获取 AK 和 SK。

单击账号名称或右侧的 **详情**，进入服务账号详情页。在该页面的 **AccessKey** 区域查看 **ID** 和 **Secret** 信息。



3.6. 服务路由

3.6.1. 服务路由介绍

RPC 最重要的功能是获取对端地址。根据使用环境的不同，服务路由策略主要有软负载和直连两种。

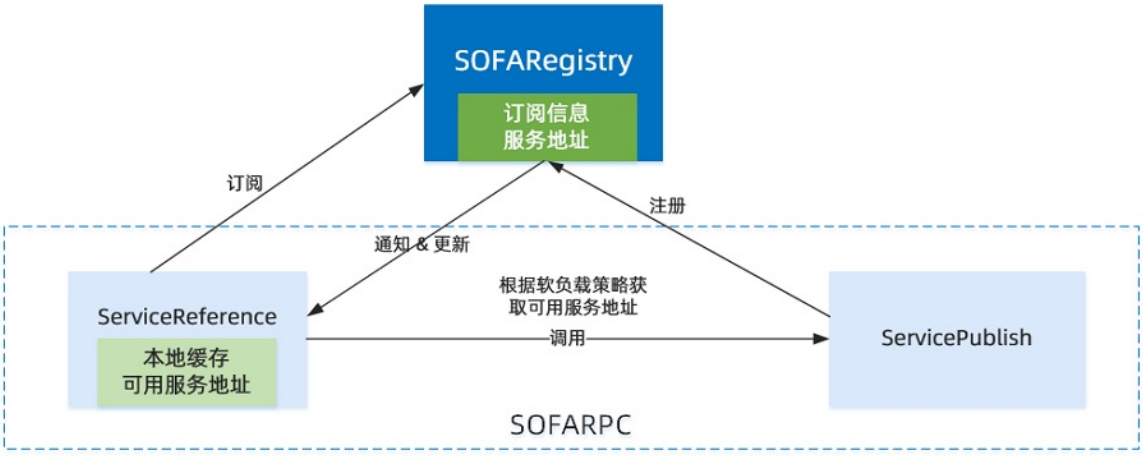
软负载

软负载即软件负载。当需要调用服务时，消费方根据软负载策略，从服务注册中心（SOFARegistry）推送到本地缓存的列表里选择一个地址，再调用该地址所提供的服务。

SOFARPC 采用服务发布（ServicePublish）和引用（ServiceReference）模型，通过服务注册中心（SOFARegistry）动态感知服务发布，并将服务地址列表推送给已经引用该服务的消费方，更新消费方本地缓存中的可用服务列表，最后通过软负载策略，为消费方选择可用地址进行远程通信。

通过路由策略，您在使用 SOFARPC 的时候，就不用将服务地址硬编码在服务注册中心的代码中。详情请参见 负载均衡。

软负载示意图



直连

在开发及测试环境下，开发者经常需要绕过注册中心，只测试指定服务提供方，这时候可能需要点对点直连来进行测试。该功能可以通过 SOFARPC 的 `test-url` 或 `target-url` 配置来实现。详情请参见 直连调用。

3.6.2. 直连调用

SOFARPC 支持指定地址进行调用的场景，此方式允许用户指定服务端地址。本文介绍直连调用的应用场景和配置方式。

注意事项

- 当直连配置生效时，注册中心软负载将会失效。
- 直连地址允许配置多个地址，多个地址之间使用英文逗号（,）或英文分号（;）分隔，多个地址间支持负载均衡。

应用场景

具体使用场景分为以下 2 种：

- 线上场景

  在 XML 文件中配置服务引用时，在标签 `sofa:binding.bolt` 里面加上一个 `route` 标签。`route` 标签中放入一个 `target-url` 属性，属性值设置为需要调用的地址。

```
<sofa:reference id="sampelService" interface="com.alipay.test.SampleService">
     <sofa:binding.bolt>
         <sofa:route target-url="${targetUrl}:port"/>
     </sofa:binding.bolt>
</sofa:reference>
```

  `${targetUrl}:port` 需修改为实际调用的地址和端口。更多配置方式，请参见 配置方式。

- 测试场景
  - 在 `application.properties` 中配置 `run_mode=TEST` 。
  - 在 XML 文件中配置服务引用时，在标签 `sofa:binding.bolt` 里面加上一个 `global-attrs` 标签，里面放入一个 `test-url` 的属性，属性值设置为需要调用的地址。

```
<sofa:reference id="sampleService" interface="com.alipay.test.SampleService">
     <sofa:binding.bolt>
         <sofa:global-attrs test-url="${targetUrl}:port"/>
     </sofa:binding.bolt>
</sofa:reference>
```

> 重要
>
> - `test-url` 仅适用于 XML 的配置方式。
> - `test-url` 必须配合 `run_mode=TEST` 才会生效，主要用于测试阶段。

配置方式

- API 方式

  通过 API 方式配置直连调用，地址格式为： `协议://IP:端口` 。配置示例如下：

```
ConsumerConfig<HelloService> consumer =new ConsumerConfig<HelloService>()
.setInterfaceId(HelloService.class.getName())
.setRegistry(registryConfig)
.setDirectUrl("bolt://127.0.0.1:12201");
```

- Annotation 方式

  通过 Annotation 方式配置直连调用，地址格式为： `IP:端口` 。配置示例如下：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", directUrl ="127.0.0.1:
12220"))
private SampleService sampleService;
```

- XML 方式

  通过 XML 方式配置直连调用，地址格式为： `IP:端口` 。配置示例如下：

```
<sofa:reference id="sampleService" interface="com.alipay.test.SampleService">
     <sofa:binding.bolt>
         <sofa:global-attrs target-url="127.0.0.1:12200"/>
     </sofa:binding.bolt>
</sofa:reference>
```

3.6.3. 注册中心路由

软负载的情况下，消费方会从注册中心做服务发现，默认选择蚂蚁的注册中心 DSR。您也可以手动指定服务的注册中心。

注册中心

目前支持的注册中心如下：

注册中心 alias		注册中心名称
	dsr	蚂蚁注册中心（企业版）

注册中心 alias		注册中心名称
普通注册中心	sofa	蚂蚁注册中心（开源版）
	consul	开源注册中心 Consul
	zookeeper	开源注册中心 ZooKeeper
	nacos	开源注册中心 Nacos
特殊注册中心	mesh	Mesh 模式，使用蚂蚁 MOSN
	gateway	网关模式，使用网关服务发现
	local	本地模式，本地调试使用
	multicast	广播模式

配置方式

- 为全局服务指定默认注册中心

  您可以在 `application.properties` 或启动参数中增加如下参数，为全局服务指定注册中心。

  - 默认使用 DSR 注册中心

    ```
    run.mode=normal
    ```

  - 默认使用本地模式

    ```
    run.mode=dev
    ```

  - 使用 Mesh 模式

    ```
    MOSN_ENABLE=true
    ```

- 为单个服务指定注册中心

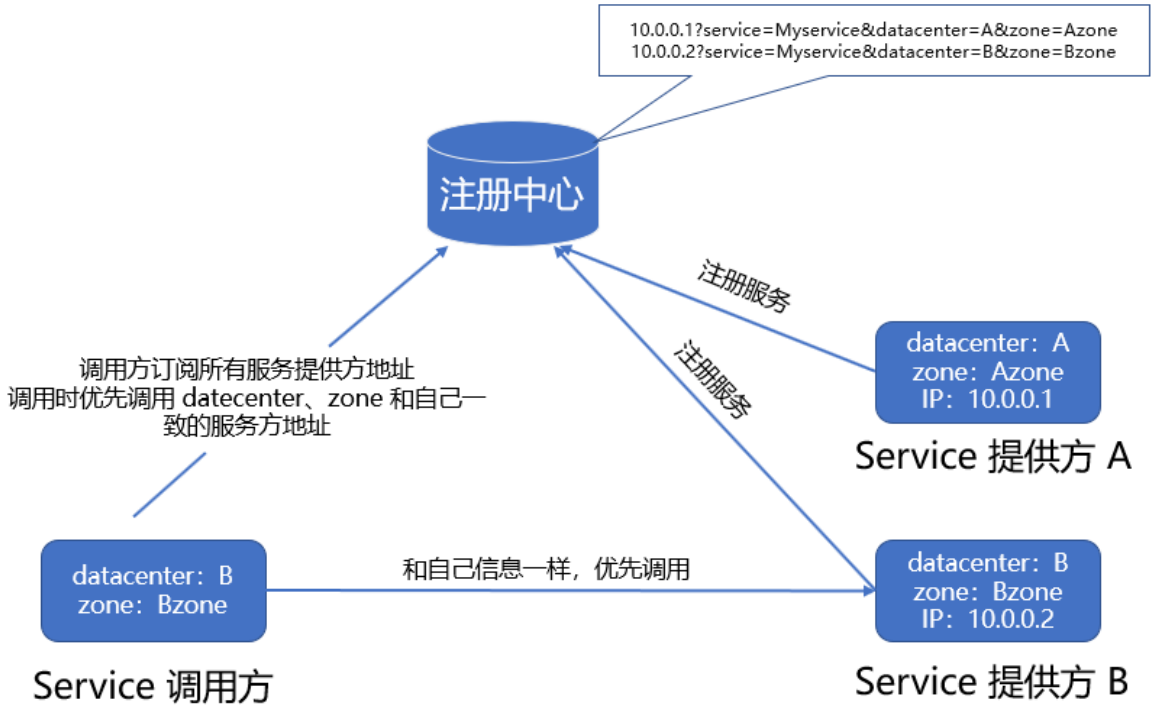
  您可以在发布和订阅服务的代码中加入 `<sofa:global-attrs registry="<注册中心 alias>" />` 字段指定注册中心，示例如下：

```
<bean id="helloSyncServiceImpl" class="com.alipay.sofa.rpc.samples.invoke.HelloSyncServic
eImpl"/>
<sofa:service ref="helloSyncServiceImpl" interface="com.alipay.sofa.rpc.samples.invoke.He
lloSyncService"unique-id="">
<sofa:binding.bolt>
<sofa:global-attrs registry="mesh" />
</sofa:binding.bolt>

</sofa:service>
```

3.6.4. 同机房路由收敛

当存在多个机房时，RPC 请求可能会跨机房调用。受限于网络等因素，跨机房调用增加调用耗时。为保证业务质量，建议您配置同机房路由收敛，让 RPC 请求优先调用同机房服务器提供的服务。

调用流程



> **说明**
>
> 以上数据结构为讲解原理进行了简化，并非真实的结构。

- datacenter：服务器所在的物理机房。
- zone：物理机房为了方便管理划分出来的逻辑单元，datacenter + zone 共同标识服务器位置信息。

调用流程如下：

1. 服务提供方 A 和 B 将自己的服务注册到注册中心。
2. 服务调用方会订阅注册中心所有服务提供方的地址信息。
3. 当调用方调用服务时，会优先调用 datacenter、zone 和自身配置一致的服务方地址。

4. 如果当前 zone 没有提供服务，则调用其他 zone 的服务。

配置方式

在服务启动参数中增加服务的位置信息：

```
-Dcom.alipay.ldc.zone=xxx
-Dcom.alipay.ldc.datacenter=xxx
```

同机房路由收敛默认开启，如果没有配置 datacenter、zone，则系统使用默认值，即所有服务的 datacenter、zone 一样，同机房路由收敛不会生效。

注意事项

- 同机房路由收敛会优先调本 zone 的服务，所以在部署时要平衡各个 zone 的服务数量，避免出现某个 zone 流量大，服务少的情况。
- datacenter + zone 标识唯一位置信息，但目前某些版本对 datacenter 不感知，所以不同 datacenter 内也要保持 zone 名不同。
- datacenter 和 zone 在阿里云上对应 datacenter 和 cell，为了方便维护，尽量不要自己命名。

3.6.5. 单元化配置

本文将引导您如何在本地客户端配置单元化，发布与引用 RPC 服务。

> **说明**
>
> 该功能仅适用于开启了单元化能力的用户。

升级依赖

您需要将 SOFABoot 版本升级到 3.3.0 或以上，详见 SOFABoot 版本说明。

```
<parent>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>sofaboot-enterprise-dependencies</artifactId>
    <version>3.3.0</version>
</parent>
```

应用参数配置

在本地项目的全局配置文件 `application.properties` 中，配置以下参数：

- 配置 AntVIP、instanceid、AccessKey ID、AccessKey Secret。

```
run.mode=NORMAL  //运行模式
com.alipay.env=shared  //环境标识
com.alipay.instanceid=${real value} //实例标识
com.antcloud.antvip.endpoint=${real value}  //AntVIP 地址
com.antcloud.mw.access=${real value}  // AccessKey ID
com.antcloud.mw.secret=${real value}  // AccessKey Secret
```

各参数获取方式，请参见 获取环境参数。

> **重要**
>
> 若您未启用权限控制，则不需要配置 AccessKey ID 和 AccessKey Secret 参数。

- 配置 Zone 信息，示例如下：

```
com.alipay.ldc.zone=LCD-Test-A-1
com.alipay.ldc.datacenter=LCD-Test-A
//开启严格模式的 LDC。
com.alipay.ldc.strictmode=true
zmode=true
//需要确保您配置的 zonemng-pool 能 ping 通。
zonemng_zone_url=http://zonemng-pool
```

服务发布

服务发布，如果您没有特殊要求，跟随部署的 Zone 进行发布即可。如在 RZone 部署，则服务就发布在 RZone；如在 CZone 部署，服务就发布在 CZone；如果两个 Zone 都部署，那么都发布。

默认情况下，服务全 Zone 发布，如果您需要指定在某些 Zone 发布，则配置如下 DRM 动态配置即可。

- 资源：`com.alipay.sofa.rpc.ldc.route.drm.config`
- 字段：`publishConfig`
- 内容：`[{"serviceName":"interface:1.0:uniqueId","cell":"RZ,CZ"}]`

比如，您当前的 Zone 是 RZ00A，则通过上面的 DRM 提前配置，您的服务会在 RZ00A 发布出来，而如果您的 Zone 名是 TZ00A，则不会发布。

配置好之后，进行应用重启，即可看到效果。目前暂不支持运行中动态变更该信息。一旦修改，必须重启生效。

服务引用

目前，一旦配置了 `com.alipay.ldc.strictmode=true`（严格模式），则路由的时候，要求入参的第一个参数必须为 `userId` 路由信息。RPC 框架计算出倒数第二、三位进行路由。如果不是，则会认为没有 LDC 逻辑，走本 Zone 优先。

如果在某些情况下，您认为以上的默认规则不足，也支持更高级的用法（支持方法维度）：

- 资源：`com.alipay.sofa.rpc.ldc.route.drm.config`
- 字段：`routeConfig`

如下所示，先匹配方法，然后匹配接口。会根据第一个参数的第二、三位进行计算，例如对于 `123`，则计算为 `23`。

```
[
    {
        "serviceName":"interface:1.0:uniqueId",
        "rule":"string.substring(args[0], 1, 3)"
    },
    {
        "serviceName":"interface:1.0:uniqueId:methodA",
        "rule":"string.substring(args[0], 1, 3)"
    }
]
```

如果是复杂对象，需要使用如下配置：

```
[
    {
        "serviceName":"interface:1.0:uniqueId",
        "rule":"string.substring(#args.[0].uid, 1, 3)"
    }
]
```

如是简单对象，也可以使用如下配置：

```
[
    {
        "serviceName":"interface:1.0:uniqueId",
        "rule":"string.substring(#args.[0], 1, 3)"
    }
]
```

3.7. 路由容错

3.7.1. 调用重试

SOFARPC 支持进行框架层面的重试策略，前提是集群模式为 FailOver（SOFARPC 默认为 FailOver 模式）。重试只有在服务端的框架层面异常或超时异常时才会发起，不会对业务抛出的异常进行重试。默认情况下 SOFARPC 不进行任何重试。

> **重要**
>
> 超时异常虽然可以重试，但是需要服务端保证业务的幂等性，否则可能会有风险。

XML 方式

如果使用 XML 方式订阅服务，可以设置 `sofa:global-attrs` 的 `retries` 参数来设置重试次数：

```
<sofa:reference jvm-first="false" id="retriesServiceReferenceBolt" interface="com.alipay.so
fa.rpc.samples.retries.RetriesService">
    <sofa:binding.bolt>
        <sofa:global-attrs retries="2"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Annotation 方式

如果是使用 Annotation 的方式，可以通过设置 `@SofaReferenceBinding` 注解的 `retries` 属性来设置重试次数：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", retries =2))
private SampleService sampleService;
```

Spring 环境下 API 方式

如果是在 Spring 环境下用 API 的方式，可以调用 `BoltBindingParam` 的 `setRetries` 方法来设置重试次数：

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setRetries(2);
```

非 Spring 环境下 API 方式

如果是在非 Spring 环境下直接使用 SOFARPC 的裸 API 的方式，可以通过调用 `ConsumerConfig` 的 `setRetries` 方法来设置重试次数：

```
ConsumerConfig<RetriesService> consumerConfig =new ConsumerConfig<RetriesService>();
consumerConfig.setRetries(2);
```

3.7.2. 预热转发

集群中某台机器刚启动的一段时间（称之为“预热期”）内，如果收到的请求过多，可能会影响机器性能和正常业务。RPC 框架可通过预热转发功能将处于预热期机器的请求转发给集群内其它机器，当机器过了预热期后再恢复正常。

您可以手动指定预热期内请求转发的比例，比如 80% 的请求转发出去，20% 自身系统处理。

与 RPC 压测转发不同的是，RPC 预热转发仅适用于应用启动后的一段时间，而压测转发则是长期生效。

注意事项

要使用预热转发功能时，您需要注意以下要求：

- 服务端在同一台主机上启动时，预热转发不生效。
- 先启动的服务还在预热中时，预热转发不生效。
- 先启动的服务没有配置预热转发时，后启动的服务不会转发到没配置预热转发的机器。

- 所有预热转发的机器，BOLT 端口必须一致。

RPC 转发配置

配置的方式有两种：

- 直接转发到 IP

  在 `application.properties` 增加如下参数：

  ```
  rpc.transmit.url = 10.**.**.2
  ```

  数据无论何时都直接转发到指定 IP，和 `rpc.transmit.url=address:10.**.**.2` 效果同样。

- 配置预热与权重

  在 `application.properties` 增加如下参数：

  ```
  rpc.transmit.url=weightStarting:0.3,during:60,weightStarted:0.2,address:10.**.**.2,uniqueId:core_unique
  ```

  - `weightStarting`：预热期内的转发权重或概率。RPC 框架内部会在集群中随机找一台机器以此权重转出或接收。
  - `during`：预热期的时间长度。单位为秒。
  - `weightStarted`：预热期过后的转发权重。将会一直生效。
  - `address`：预热期过后的转发地址。将会一直生效。
  - `uniqueId`：同 appName 多集群部署的情况下，要区别不同集群时可配置此项。指定一个自定义的系统变量，保证集群唯一即可。`core_unique` 是一个 `application.properties` 的配置，可以动态替换。

> 说明
>
> 在 `application.properties` 中可以配置转发请求超时时间，例如 `rpc_transmit_url_timeout_tr=8000`。单位为 ms，默认为 10000 ms。

3.7.3. 容灾恢复

集群中通常一个服务有多个服务提供者，其中部分服务提供者可能由于网络、配置、长时间 fullgc、线程池满、硬件故障等导致长连接还存活但是程序已经无法正常响应。单机故障剔除功能会将这部分异常的服务提供者进行降级，使客户端的请求更多地指向健康节点。当异常节点的表现正常后，单机故障剔除功能会对该节点进行恢复。解决了服务故障持续影响业务的问题，避免了雪崩效应，提高系统可用率。

功能原理

单机故障剔除会统计一个时间窗口内的调用次数和异常次数，并计算每个服务对应 IP 的异常率和该服务的平均异常率。当 IP 的异常率大于服务平均异常率，且达到一定比例时，单机故障剔除会对该服务 + IP 的维度进行权重降级。如果该服务 + IP 维度的权重并没有降为 0，那么当该服务 + IP 维度的调用情况正常时，则会对其进行权重恢复。整个计算和调控过程异步进行，不会阻塞调用。

使用方式

配置示例如下：

```
FaultToleranceConfig faultToleranceConfig =new FaultToleranceConfig();
faultToleranceConfig.setRegulationEffective(true);
faultToleranceConfig.setDegradeEffective(true);
faultToleranceConfig.setTimeWindow(20);
faultToleranceConfig.setWeightDegradeRate(0.5);

FaultToleranceConfigManager.putAppConfig("appName", faultToleranceConfig);
```

以上配置完成后，目标应用会在每 20s 的时间窗口进行一次异常情况计算，如果某个服务 + IP 的调用维度被判定为故障节点，则会将该服务 + IP 的权重降级为 0.5。更多信息，请参见 自动故障剔除。

3.7.4. 自动故障剔除

自动故障剔除功能会自动监控 RPC 调用的情况，当某个节点出现故障时，可对故障节点进行权重降级，并在节点恢复健康时进行权重恢复。目前支持 Bolt 协议。

配置方式

将自动故障剔除的参数配置到 SOFABoot 中的 `application.properties` 即可。参数说明如下：

参数	默认值	说明
com.alipay.sofa.rpc.aft.time.window	10s	时间窗口：用于设定统计信息计算的周期。
com.alipay.sofa.rpc.aft.least.window.count	10 次	时间窗口内最少调用数：只有在时间窗口内达到了该最低值的数据才会被加入到计算和调控中。
com.alipay.sofa.rpc.aft.least.window.exception.rate.multiple	6 倍	时间窗口内异常率与服务平均异常率的降级比值：在对统计信息进行计算的时候，会计算出该服务所有有效调用 IP 的平均异常率，如果某个 IP 的异常率大于等于了这个最低比值，则会被降级。
com.alipay.sofa.rpc.aft.weight.degrade.rate	1/20	降级比率：地址在进行权重降级时的降级比率。

参数	默认值	说明
com.alipay.sofa.rpc.aft.weight.recover.rate	2 倍	恢复比率：地址在进行权重恢复时的恢复比率。
com.alipay.sofa.rpc.aft.degrade.effective	false	降级开关：如果应用打开了这个开关，则会对符合降级的地址进行降级，否则只会进行日志打印。
com.alipay.sofa.rpc.aft.degrade.least.weight	1	降级最小权重：地址权重被降级后的值如果小于这个最小权重，则会以该最小权重作为降级后的值。
com.alipay.sofa.rpc.aft.degrade.max.ip.count	2	降级的最大 IP 数：同一个服务被降级的 IP 数不能超过该值。
com.alipay.sofa.rpc.aft.regulation.effective	false	全局开关：如果应用打开了这个开关，则会开启整个单点故障自动剔除功能，否则该功能不启用。

说明

- 每个参数都有默认值，您可以根据需要自行修改参数值。
- `com.alipay.sofa.rpc.aft.regulation.effective` 是该功能的全局开关，如果关闭则该功能不会运行，其他参数也都不生效。

配置示例

```
com.alipay.sofa.rpc.aft.time.window=20
com.alipay.sofa.rpc.aft.least.window.count=30
com.alipay.sofa.rpc.aft.least.window.exception.rate.multiple=1.4
com.alipay.sofa.rpc.aft.weight.degrade.rate=0.5
com.alipay.sofa.rpc.aft.weight.recover.rate=1.2
com.alipay.sofa.rpc.aft.degrade.effective=true
com.alipay.sofa.rpc.aft.degrade.least.weight=1
com.alipay.sofa.rpc.aft.degrade.max.ip.count=2
com.alipay.sofa.rpc.aft.regulation.effective=true
```

对示例配置，说明如下：

- 已打开自动故障剔除功能和降级开关，当节点出现故障时会被进行权重降级，在恢复时会被进行权重恢复。
- 每隔 20s 进行一次节点健康状态的度量，20s 内调用次数超过 30 次的节点才被作为计算数据。

- 如果单个节点的异常率超过了所有节点的平均异常率的 1.4 倍，则对该节点进行权重降级，降级的比率为 0.5。
- 如果单个节点的异常率低于平均异常率的 1.4 倍，则对该节点进行权重恢复，恢复的比率为 1.2。
- 权重最小降级到 1。
- 单个服务最多降级 2 个 IP。

3.8. 负载均衡

本文介绍 SOFARPC 支持的负载均衡算法以及如何设置负载均衡。

负载均衡算法

SOFARPC 目前支持以下五种算法：

类型	名称	描述
random	随机算法	默认负载均衡算法。
localPref	本地优先算法	优先发现是否本机发布了该服务，如果没有再采用随机算法。
roundRobin	轮询算法	方法级别的轮询，各个方法间各自轮询，互不影响。
consistentHash	一致性 Hash 算法	服务请求与处理请求的服务器之间可以一一对应。
weightRoundRobin	按权重负载均衡轮询算法	按照权重对节点进行轮询。性能较差，不推荐使用。

设置负载均衡

要使用某种特定的负载均衡算法，可以按照以下的方式进行设置：

- **XML 方式**

  如果使用 XML 的方式引用服务，可以通过设置 `sofa:global-attrs` 标签的 `loadBalancer` 属性来设置负载均衡。以下示例以设置负载均衡算法为 roundRobin 为例：

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
    <sofa:binding.bolt>
        <sofa:global-attrs loadBalancer="roundRobin"/>
    </sofa:binding.bolt>
</sofa:reference>
```

- **在 Spring 环境下 API 方式**

如果在 Spring 或者 Spring Boot 的环境下使用 API，可以通过调用 `BoltBindingParam` 的 `setLoadBalancer` 方法来设置。以下示例以设置负载均衡算法为 roundRobin 为例：

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setLoadBalancer("roundRobin");
```

- 非 Spring 环境下 API 方式

  如果在非 Spring 环境下直接使用 SOFARPC 提供的裸 API，可以通过调用 `ConsumerConfig` 的 `setLoadBalancer` 方法来设置。以下示例以设置负载均衡算法为 random 为例：

```
ConsumerConfig consumerConfig =new ConsumerConfig();
consumerConfig.setLoadbalancer("random");
```

3.9. 通信协议

3.9.1. 多协议发布

在 SOFARPC 中，同一个服务可以多协议发布，并按不同协议被客户端调用。主要调用方式有下述几种：

- Java API 方式

  可以按照如下的代码构建多个 `ServerConfig`，不同的 `ServerConfig` 设置不同的协议，然后将这些 `ServerConfig` 设置给 `ProviderConfig`。

```
List<ServerConfig> serverConfigs =new ArrayList<ServerConfig>();
serverConfigs.add(serverConfigA);
serverConfigs.add(serverConfigB);
providerConfig.setServer(serverConfigs);
```

- XML 方式

  直接在 `<sofa:service>` 标签中增加多个 `binding` 即可：

```
<sofa:service ref="sampleFacadeImpl" interface="com.alipay.sofa.rpc.bean.SampleFacade">
    <sofa:binding.bolt/>
    <sofa:binding.rest/>
    <sofa:binding.dubbo/>
</sofa:service>
```

- Annotation 方式

  在 `@SofaService` 中增加多个 `binding` 即可：

> 说明
>
> 如果使用 rest 协议，接口需要要加 path。

```
@SofaService(
      interfaceType =SampleService.class,
      bindings ={
@SofaServiceBinding(bindingType ="rest"),//使用 rest 协议，接口需要要加 path。
@SofaServiceBinding(bindingType ="bolt")
}
)
public class SampleServiceImpl implements SampleService{
// ...
}
```

3.9.2. Bolt 协议

3.9.2.1. Bolt 协议的基本用法

本文介绍如何发布和引用 Bolt 协议服务。

发布服务

使用 SOFARPC 发布一个 Bolt 协议的服务，只需要增加名称为 Bolt 的 Binding 即可，不同的使用方式添加 Bolt Binding 的方式如下：

- XML

  使用 XML 发布一个 Bolt 协议只需在 `<sofa:service>` 标签下增加 `<sofa:binding.bolt>` 标签。

```
<sofa:service ref="sampleService" interface="com.alipay.sofa.rpc.sample.SampleService">
    <sofa:binding.bolt/>
</sofa:service>
```

- Annotation

  使用 Annotation 发布一个 Bolt 协议的服务只需设置 `@SofaServiceBinding` 的 `bindingType` 为 `bolt`。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="bolt")})
public class SampleServiceImpl implements SampleService{
}
```

- Spring 环境下 API 方式

  在 Spring 或者 Spring Boot 环境下发布一个 Bolt 协议的服务只需往 `ServiceParam` 里面增加一个 `BoltBindingParam`。

```
ServiceParam serviceParam =new ServiceParam();
serviceParam.setInterfaceType(SampleService.class);// 设置服务接口。
serviceParam.setInstance(new SampleServiceImpl());// 设置服务接口的实现。

List<BindingParam> params =new ArrayList<BindingParam>();
BindingParam serviceBindingParam =new BoltBindingParam();
params.add(serviceBindingParam);
serviceParam.setBindingParams(params);
```

- 非 Spring 环境下的 API 方式

  在非 Spring 环境下使用 SOFARPC 的裸 API 提供 Bolt 协议的服务，只需要将 Protocol 为 Bolt 的 `ServerConfig` 设置给对应的 `ProviderConfig` 。

```
RegistryConfig registryConfig =new RegistryConfig()
.setProtocol("zookeeper")
.setAddress("127.0.0.1:2181");
// 新建一个协议为 Bolt 的 ServerConfig。
ServerConfig serverConfig =new ServerConfig()
.setPort(8803)
.setProtocol("bolt");
ProviderConfig<SampleService> providerConfig =new ProviderConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRef(new SampleServiceImpl())
// 将 ServerConfig 设置给 ProviderConfig，表示这个服务发布的协议为 Bolt。
.setServer(serverConfig)
.setRegistry(registryConfig);
providerConfig.export();
```

引用服务

使用 SOFARPC 引用一个 Bolt 服务，只需要增加名称为 Bolt 的 Binding 即可。不同发布方式中添加 Bolt Binding 的方法如下：

- XML

  使用 XML 引用一个 Bolt 协议的服务只需在 `<sofa:reference>` 标签下增加 `<sofa:binding.bolt>` 标签。

```
<sofa:reference id="sampleService" interface="com.alipay.sofa.rpc.sample.SampleService">
<sofa:binding.bolt/>
</sofa:reference>
```

- Annotation

  使用 Annotation 引用一个 Bolt 协议的服务只需设置 `@SofaReferenceBinding` 的 `bindingType` 为 `bolt` 。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt"))
private SampleService sampleService;
```

- Spring 环境下 API 方式

  在 Spring 或者 Spring Boot 环境下引用一个 Bolt 协议的服务，只需往 `ReferenceParam` 里面增加一个 `BoltBindingParam` 。

```
ReferenceClient referenceClient = clientFactory.getClient(ReferenceClient.class);
ReferenceParam<SampleService> referenceParam =new ReferenceParam<SampleService>();
referenceParam.setInterfaceType(SampleService.class);

BindingParam refBindingParam =new BoltBindingParam();
referenceParam.setBindingParam(refBindingParam);
```

- 非 Spring 环境下的 API 方式

  在非 Spring 环境下使用 SOFARPC 的裸 API 引用一个 Bolt 协议的服务，只需设置 `ConsumerConfig` 的 Protocol 为 `bolt` 。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt");
SampleService sampleService = consumerConfig.refer();
```

3.9.2.2. Bolt 协议的调用方式

SOFARPC 在 BOLT 协议下提供了多种调用方式满足不同的场景。

同步

在同步调用方式下，客户端发起调用后会等待服务端返回结果再进行后续的操作。这是 SOFARPC 的默认调用方式，无需进行任何设置。

异步

设置调用方式

异步调用的方式下，客户端发起调用后不会等到服务端的结果，而是继续执行后面的业务逻辑。服务端返回的结果会被 SOFARPC 缓存，当客户端需要结果的时候，再主动调用 API 获取。如果您需要将一个服务设置为异步的调用方式，可以在对应的使用方式下设置 `type` 属性。

- XML 方式

  在 XML 方式下，设置 `<sofa:global-attrs>` 标签的 `type` 属性为 `future` 。

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
     <sofa:binding.bolt>
         <sofa:global-attrs type="future"/>
     </sofa:binding.bolt>
</sofa:reference>
```

- Annotation 方式

在 Annotation 方式下，设置 `@SofaReferenceBinding` 的 `invokeType` 属性为 `future`。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", invokeType ="future"))
private SampleService sampleService;
```

- Spring 环境下 API 方式

  在 Spring 环境下使用 API，设置 `BoltBindingParam` 的 `type` 属性为 `future`。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setType("future");
```

- 在非 Spring 环境下 API 方式

  在非 Spring 环境下使用 SOFARPC 的裸 API，设置 `ConsumerConfig` 的 `invokeType` 属性为 `future`。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setInvokeType("future");
```

获取调用结果

目前提供两种方式来获取异步调用的结果：

- 直接获取结果

  您可以通过以下的方式来直接获取异步调用的结果：

```
String result =(String)SofaResponseFuture.getResponse(0,true);
```

  其中，`0` 表示获取结果的超时时间，您可以根据需要修改；`true` 表示清除线程上下文中的结果。若您不希望清除结果，可以修改为 `false`。

- 获取 JDK 原生 Future

  您可以通过以下的方式来获取 JDK 的原生的 Future 对象，再从任意地方去调用这个 Future 对象来获取结果。

```
Future future =SofaResponseFuture.getFuture(true);
```

  `true` 表示清除线程上下文中的结果。若您不希望清除，可以修改为 `false`。

回调

SOFARPC Bolt 协议的回调方式可以让 SOFARPC 在发起调用后不等待结果，在客户端收到服务端返回的结果后，自动回调用户实现的一个回调接口。

使用 SOFARPC Bolt 协议的回调方式，首先需要实现一个回调接口，并且在对应的配置中设置回调接口，再将调用方式设置为 `callback` 。

实现回调接口

SOFARPC 提供了一个回调的接口 `com.alipay.sofa.rpc.core.invoke.SofaResponseCallback` 。使用 SOFARPC Bolt 协议的回调方式时，需要先实现这个接口，该接口提供以下三个方法：

- `onAppResponse` ：当客户端接收到服务端的正常返回的时候，SOFARPC 会回调这个方法。
- `onAppException` ：当客户端接收到服务端的异常响应的时候，SOFARPC 会回调这个方法。
- `onSofaException` ：当 SOFARPC 本身出现一些错误时（例如路由错误），SOFARPC 会回调这个方法。

设置回调接口

实现回调接口之后，您需要将实现类设置到对应的服务引用配置中，并且将调用方式设置为 callback。SOFARPC 为设置回调接口提供了 Callback Class 和 Callback Ref 两种方式：

- Callback Class：直接设置回调的类名，SOFARPC 会通过调用回调类的默认构造函数的方式生成回调类的实例。
- Callback Ref：为用户直接提供回调类的实例。

通过不同使用方式设置回调接口的代码如下：

- **XML 方式**

  如果通过 XML 的方式引用服务，将 `<sofa:global-attrs>` 标签的 `type` 属性设置为 `callback` ，并且设置 `callback-ref` 或者 `callback-class` 属性。以 `callback-ref` 为例，代码如下：

  ```
  <bean id="sampleCallback" class="com.example.demo.SampleCallback"/>
  <sofa:reference interface="com.example.demo.SampleService" id="sampleService">
      <sofa:binding.bolt>
          <sofa:global-attrs type="callback" callback-ref="sampleCallback"/>
      </sofa:binding.bolt>
  </sofa:reference>
  ```

  在 XML 的方式下， `callback-ref` 的值需要是回调类的 Bean 名称。

- **Annotation 方式**

  如果通过 Annotation 的方式引用服务，设置 `@SofaReferenceBinding` 注解的 `invokeType` 属性为 `callback` ，并且设置 `callbackClass` 或者 `callbackRef` 属性。以 `callback-ref` 为例，代码如下：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt",
            invokeType ="callback",
            callbackRef ="sampleCallback"))
private SampleService sampleService;
```

在 Annotation 的方式下，`callbackRef` 属性的值需要是回调类的 Bean 名称。

- **Spring 环境 API 方式**

  如果在 Spring 或者 Spring Boot 的环境下使用 API 的方式，设置 `BoltBindingParam` 的 `type` 属性为 `callback`，并且设置 `callbackClass` 或者 `callbackRef` 属性即可。以 `callbackClass` 为例，代码如下：

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setType("callback");
boltBindingParam.setCallbackClass("com.example.demo.SampleCallback");
```

- **非 Spring 环境下 API 方式**

  如果在非 Spring 环境下使用 SOFARPC 的裸 API，需将 `setInvokeType` 类型设置成 `callback`，并且调用 `setOnReturn` 设置回调类。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setInvokeType("callback")
.setOnReturn(newSampleCallback());
```

在调用级别设置回调接口

除了在服务级别设置回调接口之外，您还可以在调用级别设置回调接口，方式如下：

```
RpcInvokeContext.getContext().setResponseCallback(newSampleCallback());
```

单向

当客户端发送请求后不关心服务端返回的结果时，您可以使用单向的调用方式。这种方式会在发起调用后立即返回 null，并且忽略服务端的返回结果。

使用单向的方式只需将调用方式设置为 `oneway` 即可，设置方式和将调用方式设置为 `future` 或者 `callback` 一样，您可以参考上面的文档中提供的设置方式。需要特别注意的是，由于单向的调用方式会立即返回，所有的超时设置在单向的情况下都是无效的。

3.9.2.3. 配置 Bolt 服务

Bolt 服务的名称源自 RPC 使用的底层通信框架 Bolt。相对于传统的 WebService，Bolt 支持更加复杂的对象，序列化后的对象更小，且提供了更为丰富的调用方式（sync、oneway、callback、future 等），支持更广泛的应用场景。

在 SOFA 中，Bolt 服务提供方使用的端口是 12200，详情请参见 配置说明。

发布及引用 Bolt 服务

SOFA 中的 RPC 通过 Binding 模型定义不同的通信协议，每接入一种新的协议，就会增加一种 Binding 模型。要在 SOFA 中添加 RPC 的 Bolt 协议的实现，就需要在 Binding 模型中增加一个 `<sofa:binding.bolt/>` 。

- 发布一个 Bolt 的服务需在 `<sofa:service>` 中添加 `<sofa:binding.bolt/>` ，示例如下：

```
<!-- 发布 Bolt 服务 -->
<sofa:service interface="com.alipay.test.SampleService" ref="sampleService" unique-id="se
rvice1">
     <sofa:binding.bolt/>
</sofa:service>
```

- 引用一个 Bolt 的服务需在 `<sofa:reference>` 中添加 `<sofa:binding.bolt/>` ，示例如下：

```
<!-- 引用 Bolt 服务 -->
<sofa:reference interface="com.alipay.test.SampleService" id="sampleService">
     <sofa:binding.bolt/>
</sofa:reference>
```

Bolt 服务完整配置

Bolt 配置标签主要有 `global-attrs` 和 `method` 两种，同时配置时，以 `method` 中的配置为准。可配置的参数如下：

- `type` ：调用方式。取值为 sync、oneway、callback、future，默认为 sync。
- `timeout` ：超时时间。默认为 3000 ms。
- `callback-class` ：调用方式为 callback 时的回调对象类。
- `callback-ref` ：调用方式为 callback 时的回调 Spring Bean 引用。

Bolt 服务发布完整配置

```
<sofa:service interface="com.alipay.test.SampleService" ref="sampleService" unique-id="serv
ice1">
    <sofa:binding.bolt>
        <sofa:global-attrs timeout="5000"/>
        <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
        <sofa:method name="service" timeout="3000"/>
    </sofa:binding.bolt>
</sofa:service>
```

Bolt 服务引用完整配置

```
<sofa:reference id="sampleService" interface="com.alipay.test.SampleService" unique-id="ser
vice1">
    <sofa:binding.bolt>
        <sofa:global-attrs timeout="5000" test-url="localhost:12200" address-wait-time="100
0"
            connect.timeout="1000" connect.num="-1" idle.timeout="-1" idle.timeout.read="-1
"/>
        <!-- method 配置的属性优先级高于 global-attrs 中的配置。 -->
        <!-- 此处方法名 futureMethod 和 callbackMethod 为示例，需要替换成实际方法名。 -->
        <sofa:method name="futureMethod" type="future" timeout="5000"/>
        <sofa:method name="callbackMethod" type="callback" timeout="3000"
            callback-class="com.alipay.test.TestCallback"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Bolt 服务提供方配置

Bolt 的底层服务提供方是一个 Java NIO Server (Non-blocking I/O Server)，SOFA 框架提供几个选项来调整 Bolt Server 的一些属性。详情请参见 配置说明。

Bolt 服务消费方配置

Bolt 引用调用方式

Bolt 提供多种调用方式，以满足各种业务场景的需求。目前，Bolt 提供的调用方式有以下几种：

调用方式	类型	说明
sync	同步	Bolt 默认的调用方式。
oneway	异步	消费方发送请求后直接返回，忽略提供方的处理结果。
callback	异步	消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。
future	异步	消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。

- sync 调用

  Bolt 默认的调用方式，支持在服务提供方和消费方配置超时时间：

- 服务提供方

  XML 配置如下：

```
<!-- 服务方超时配置 -->
<sofa:service interface="com.alipay.test.SampleService2" ref="sampleService2">
    <sofa:binding.bolt>
    <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
    <sofa:method name="service" timeout="5000"/>
    </sofa:binding.bolt>
</sofa:service>
```

- 服务消费方

  XML 配置如下：

```
<!-- 消费方超时配置 -->
<sofa:reference interface="com.alipay.test.SampleService2" id="sampleServiceSync">
    <sofa:binding.bolt>
         <!-- 超时全局配置 -->
          <sofa:global-attrs timeout="8000" test-url="127.0.0.1:12200"/>
          <!-- 如果配置了 method ，优先使用 method 配置。 -->
          <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
          <sofa:method name="service" type="sync" timeout="10000"/>
    </sofa:binding.bolt>
</sofa:reference>
```

  `type` 的默认取值为 `sync` 。如果您有多个方法需要配置超时时间，并且超时时间都相同，也可以设置全局的超时时间。XML 配置如下：

```
<!-- 消费方批量配置超时时间 -->
<sofa:reference interface="com.alipay.test.SampleService2" id="sampleServiceSync">
     <sofa:binding.bolt>
          <sofa:global-attrs timeout="5000" test-url="127.0.0.1:12200"/>
     </sofa:binding.bolt>
</sofa:reference>
```

  > **说明**
  >
  > 如果在消费方配置了超时时间，那么将以消费方的超时时间为准，提供方的超时时间将被覆盖；如果消费方没有配置，则以提供方的超时时间为准。超时时间优先级：`reference method > reference global-attrs > service method > service global-attrs` 。

- oneway 调用

  如果是 oneway 调用方式，消费方不关心结果，在发起调用后直接返回，框架会忽略提供方的处理结果。在 Bolt 中，在 XML 中针对 `method` 的配置如下：

```
<!-- 配置 oneway -->
<sofa:reference interface="com.alipay.test.SampleService2" id="sampleService2">
    <sofa:binding.bolt>
        <sofa:global-attrs test-url="127.0.0.1:12200"/>
        <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
        <sofa:method name="service" type="oneway"/>
    </sofa:binding.bolt>
</sofa:reference>
```

> **说明**
>
> 由于消费方是直接返回，不关心处理结果，所以在 oneway 方式下配置超时属性是无效的。

- callback 调用

  callback 是一种异步回调方式，消费方需要提供回调接口。在调用结束后，回调接口会被框架调用。callback 接口支持通过配置 `class` 或引用 Bean 的方式实现 callback:

  - 配置 class

    XML 配置如下：

    ```
    <!-- callback 调用配置 -->
    <sofa:reference interface="com.alipay.test.SampleService2" id="sampleService">
        <sofa:binding.bolt>
            <!-- 此处方法名 testCallback 为示例，需要替换成实际方法名。 -->
            <sofa:method name="testCallback" type="callback" callback-class="com.alipay.tes
    t.binding.tr.MyCallBackHandler"/>
        </sofa:binding.bolt>
    </sofa:reference>
    ```

  - 引用 Bean

    XML 配置如下：

    ```
    <!-- 使用 Bean 来实现 callback 接口 -->
    <bean id="myCallBackHandlerBean"class="com.alipay.test.binding.tr.MyCallBackHandler"/>
    <sofa:reference interface="com.alipay.test.SampleService2" id="sampleService">
        <sofa:binding.bolt>
            <!-- 此处方法名 testCallback 为示例，需要替换成实际方法名。 -->
            <sofa:method name="testCallback" type="callback" callback-ref="myCallBackHandle
    rBean"/>
        </sofa:binding.bolt>
    </sofa:reference>
    ```

需要注意，使用 callback 调用方式时，callback 接口必须实现 `com.alipay.sofa.rpc.api.callback.SofaResponseCallback` 。这个接口包含三个方法，说明如下：

```
public interface SofaResponseCallback {

  /**
   * 当服务提供方业务层正常返回结果，sofa-remoting 层将回调该方法。
   * @param appResponse response object。
   * @param methodName 调用服务对应的方法名。
   * @param request callback 对应的 request。
   `*/
  public void onAppResponse(Object appResponse, String methodName, RequestBase request);

  /**
   * 当服务提供方业务层抛出异常，sofa-remoting 层将回调该方法。
   * @param t 服务方业务层抛出的异常。
   * @param methodName 调用服务对应的方法名。
   * @param request callback 对应的 request。
   `*/
  public void onAppException(Throwable t, String methodName, RequestBase request);

  /**
   * 当 sofa-remoting 层出现异常时，回调该方法。
   * @param sofaException sofa-remoting 层异常。
   * @param methodName 调用服务对应的方法名。
   * @param request callback 对应的 request。
   `*/
  public void onSofaException(SofaRpcException sofaException, String methodName,
                              RequestBase request);

}
```

- future 调用

  future 也是一种异步的调用方式。消费方发起调用后，马上返回。当需要结果时，消费方需要主动去获取数据。使用 future 的方式调用，配置和其他方式类似，只需要在 `method` 层面设置 `type` 为 future 即可。XML 配置如下：

  ```
  <!-- Future 调用配置 -->
  <sofa:reference interface="com.alipay.test.SampleService" id="sampleServiceFuture">
      <sofa:binding.bolt>
          <sofa:global-attrs test-url="127.0.0.1:12200"/>
          <!-- 此处方法名 service 为示例，需要替换成实际方法名。 -->
          <sofa:method name="service" type="future"/>
      </sofa:binding.bolt>
  </sofa:reference>
  ```

  使用 future 调用，返回的结果保存在一个 `ThreadLocal` 线程变量里面，可以通过如下方式获取这个线程变量的值：

```
// Future 获取调用结果。
public void testFuture() throws SofaException, InterruptedException {
  sampleServiceFuture.service();
  Object result = SofaResponseFuture.getResponse(1000, true);

  Assert.assertEquals("Hello, world!", result);
}
```

要拿到 future 调用后的结果，只需要调用 `SofaResponseFuture` 的 `getResponse` 方法即可。

`getResponse` 方法的两个参数说明如下：

- 第一个参数是超时时间，含义是调用线程等待的最长时间，负数表示无等待时间限制，零表示立即返回，单位是毫秒。
- 第二个参数代表是否清除 `ThreadLocal` 变量的值，如果设为 true，则返回 response 之前，先清除 `ThreadLocal` 的值，避免内存泄漏。

Bolt 引用详细配置

除了各种调用方式之外，Bolt 还在消费方提供了各种配置选项，这些选项不太常用，列举如下：

配置项	类型	默认值	说明
`connect.timeout`	INTEGER	1000	消费方连接超时时间。 单位：ms
`connect.num`	INTEGER	-1	消费方连接数。 -1 代表不设置（默认为每个目标地址建立一个连接）。
`idle.timeout`	INTEGER	-1	消费方最大空闲时间。 -1 表示使用底层默认值（底层默认值为 0，表示永远不会有读 idle）。该配置也是心跳的时间间隔，当请求 idle 时间超过配置时间后，发送心跳到服务方。
`idle.timeout.read`	INTEGER	-1	消费方最大读空闲时间。 -1 表示使用底层默认值（底层默认值为 30）。如果 `idle.timeout` > 0，在 `idle.timeout` + `idle.timeout.read` 时间内，如果没有请求发生，那么该连接将会自动断开。

配置项	类型	默认值	说明
address-wait-time	INTEGER	0	reference 生成时，等待服务注册中心将地址推送到消费方的时间。 单位：ms 取值范围：[0, 30000] 取值超过 30000 时，默认调整为 30000 。

3.9.2.4. 超时控制

使用 BOLT 协议进行通信的时，SOFARPC 的超时时间默认为 3000 毫秒，您可以在引用服务时设置超时时间，也可以在服务以及方法的维度设置超时时间。SOFARPC 的超时时间的设置的单位都为毫秒。

服务维度

如果您需要在发布服务时，在服务维度设置超时时间，可以设置 `timeout` 参数。

XML 方式

如果使用 XML 的方式引用服务，设置 `<sofa:binding.bolt>` 标签下的 `<sofa:global-attrs>` 标签的 `timeout` 属性的值即可。

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
    <sofa:binding.bolt>
        <sofa:global-attrs timeout="2000"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Annotation 方式

如果使用 Annotation 引用服务，设置 `@SofaReferenceBinding` 的 `timeout` 属性的值即可。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="bolt", timeout =2000))
private SampleService sampleService;
```

Spring 环境 API 方式

如果在 Spring 或者 Spring Boot 的环境下引用服务，设置 `BoltBindingParam` 的 `timeout` 属性的值即可。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setTimeout(2000)
```

非 Spring 环境下 API 方式

如果在非 Spring 环境下直接使用 SOFARPC 的裸 API 引用服务，设置 `ConsumerConfig` 的 `timeout` 属性即可。

```
ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setTimeout(2000);
```

方法维度

如果您希望单独调整一个服务中某一个方法的超时时间，可以在方法维度上设置超时时间。对于某一个方法，优先生效方法维度的超时时间，如果没有设置，则使用服务维度的超时时间。

XML 方式

如果使用 XML 的方式引用一个服务，设置对应的 `<sofa:method>` 标签的 `timeout` 属性即可。

```
<sofa:reference interface="com.example.demo.SampleService" id="sampleService">
    <sofa:binding.bolt>
        <sofa:method name="hello" timeout="2000"/>
    </sofa:binding.bolt>
</sofa:reference>
```

Annotation 方式

目前暂未提供通过 Annotation 的方式来设置方法级别的超时时间。

Spring 环境 API 方式

如果在 Spring 或者 Spring Boot 的环境下引用服务，设置 `RpcBindingMethodInfo` 的 `timeout` 属性的值即可。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();

RpcBindingMethodInfo rpcBindingMethodInfo =new RpcBindingMethodInfo();
rpcBindingMethodInfo.setName("hello");
rpcBindingMethodInfo.setTimeout(2000);

List<RpcBindingMethodInfo> rpcBindingMethodInfos =new ArrayList<>();
rpcBindingMethodInfos.add(rpcBindingMethodInfo);

boltBindingParam.setMethodInfos(rpcBindingMethodInfos);
```

非 Spring 环境 API 方式

如果在非 Spring 环境下使用 SOFARPC 的裸 API 引用服务，设置 `MethodConfig` 的 `timeout` 属性即可。

```
MethodConfig methodConfig =new MethodConfig();
methodConfig.setName("hello");
methodConfig.setTimeout(2000);

List<MethodConfig> methodConfigs =new ArrayList<MethodConfig>();
methodConfigs.add(methodConfig);

ConsumerConfig<SampleService> consumerConfig =new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRegistry(registryConfig)
.setProtocol("bolt")
.setMethods(methodConfigs);
```

3.9.2.5. 泛化调用

在进行 RPC 调用时，应用无需依赖服务提供方的 JAR 包，只需要知道服务的接口名、方法名即可调用 RPC 服务。

泛化接口

```
public interface GenericService{

/**
   * 泛化调用仅支持方法参数为基本数据类型，或者方法参数类型在当前应用的 ClassLoader 中存在的情况。
   *
   * @param methodName 调用方法名。
   * @param args 调用参数列表。
   * @return 调用结果。
   * @throws com.alipay.sofa.rpc.core.exception.GenericException 调用异常。
   */
Object $invoke(String methodName,String[] argTypes,Object[] args) throws GenericException;

/**
  * 支持参数类型无法在类加载器加载情况的泛化调用，对于非 JDK 类会序列化为 GenericObject。
  *
  * @param methodName  调用方法名。
  * @param argTypes  参数类型。
  * @param args 方法参数，参数类型支持 GenericObject。
  * @return result GenericObject 类型。
  * @throws com.alipay.sofa.rpc.core.exception.GenericException
  */
Object $genericInvoke(String methodName,String[] argTypes,Object[] args)
throws GenericException;

/**
     * 支持参数类型无法在类加载器加载情况的泛化调用。
     *
     * @param methodName 调用方法名。
     * @param argTypes   参数类型。
     * @param args       方法参数,参数类型支持 GenericObject。
     * @param context    GenericContext
     * @return result GenericObject 类型。
     * @throws com.alipay.sofa.rpc.core.exception.GenericException
```

```
     */
Object $genericInvoke(String methodName,String[] argTypes,Object[] args,
GenericContext context)throws GenericException;

/**
  * 支持参数类型无法在类加载器加载情况的泛化调用，返回结果类型为 T。
  *
  * @param methodName  调用方法名。
  * @param argTypes  参数类型。
  * @param args 方法参数，参数类型支持 GenericObject。
  * @return result T 类型。
  * @throws com.alipay.sofa.rpc.core.exception.GenericException
  */
<T> T $genericInvoke(String methodName,String[] argTypes,Object[] args,Class<T> clazz) thro
ws GenericException;

/**
     * 支持参数类型无法在类加载器加载情况的泛化调用。
     *
     * @param methodName 调用方法名。
     * @param argTypes   参数类型。
     * @param args       方法参数，参数类型支持 GenericObject。
     * @param clazz      返回类型。
     * @param context    GenericContext
     * @return result T 类型。
     * @throws com.alipay.sofa.rpc.core.exception.GenericException
     */
<T> T $genericInvoke(String methodName,String[] argTypes,Object[] args,Class<T> clazz,Gener
icContext context)throwsGenericException;

}
```

`$invoke` 仅支持方法参数类型在当前应用的 `ClassLoader` 中存在的情况；`$genericInvoke` 支持方法参数类型在当前应用的 `ClassLoader` 中不存在的情况。

使用示例

- 使用 Spring XML 创建泛化调用代理对象

```
<!-- 引用 TR 服务 -->
<sofa:reference interface="com.alipay.sofa.rpc.api.GenericService" id="genericService">
    <sofa:binding.tr>
        <sofa:global-attrs generic-interface="com.alipay.test.SampleService"/>
    </sofa:binding.tr>
</sofa:reference>
```

Java 代码：

```
/**
    * Java Bean
    */
   public class People{
       private String name;
       private int    age;

       // getters and setters
   }


   /**
    * 服务方提供的接口。
    */
   interface SampleService{
       String hello(String arg);
       People hello(People people);
   }
```

```
 /**
     * 消费方测试类。
     */
    public class ConsumerClass{
        GenericService genericService;

        public void do(){
            // 1. $invoke 仅支持方法参数类型在当前应用的 ClassLoader 中存在的情况。
            genericService.$invoke("hello",new String[]{String.class.getName()},new Objec
t[]{"I'm an arg"});

            // 2. $genericInvoke 支持方法参数类型在当前应用的 ClassLoader 中不存在的情况。
            // 2.1 构造参数
            GenericObject genericObject =new GenericObject("com.alipay.sofa.rpc.test.gene
ric.bean.People");// 构造函数中指定全路径类名。
            genericObject.putField("name","Lilei");// 调用 putField，指定field值。
            genericObject.putField("age",15);

            // 2.2 进行调用，不指定返回类型，返回结果类型为 GenericObject。
            Object obj = genericService.$genericInvoke("hello",new String[]{"com.alipay.s
ofa.rpc.test.generic.bean.People"},new Object[]{ genericObject });
            Assert.assertTrue(obj.getClass()==GenericObject.class);

            // 2.3 进行调用，指定返回类型。
            People people = genericService.$genericInvoke("hello",new String[]{"com.alipa
y.sofa.rpc.test.generic.bean.People"},new Object[]{ genericObject },People.class);

            // 3. LDC 架构下的泛化调用使用。
            // 3.1 构造 GenericContext 对象。
            AlipayGenericContext genericContext =new AlipayGenericContext();
            genericContext.setUid("33");

            // 3.2 进行调用。
            People people = genericService.$genericInvoke("hello",new String[]{"com.alipa
y.sofa.rpc.test.generic.bean.People"},new Object[]{ genericObject },People.class, generic
Context);
        }
```

- 使用 API 创建泛化调用代理对象

  在非必要情况下，并不推荐项目中使用 API 方式创建泛化调用代理对象，建议您优先通过 Spring XML 进行创建。如果您无法使用 Spring XML 方式进行初始化，可考虑使用 API 方式进行创建。

  使用 API 方式创建泛化调用代理时，需要注意以下事项：

  - 泛化调用代理是比较重要的对象，不可放在交易中反复创建。

    RPC 框架会检查是否有重复创建的情况，当相同配置的服务引用超过 3 次时，会产生异常，影响正常使用。
  - 在应用启动过程中进行对象创建，通过异常中止、健康检查等方式确保项目启动时，代理对象已创建成功。
  - 在应用退出前销毁泛化调用代理，实现优雅停止。

  Java 代码：

```
@Component
public class GenericServiceDemo implements InitializingBean, DisposableBean {
    private ConsumerConfig<GenericService> consumerConfig;
    private GenericService genericService;

    public GenericService getGenericService() {
        return genericService;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        this.consumerConfig = new ConsumerConfig<>();
        RegistryConfig registryConfig = new RegistryConfig()
                .setProtocol("dsr");
        consumerConfig.setGeneric(true)
                .setInterfaceId("com.alipay.test.SampleService")
                .setRegistry(registryConfig)
                .setProtocol("bolt")     // 引用 Bolt 服务。
                .setTimeout(5000);       // 设置超时时间。
        this.genericService = consumerConfig.refer();
    }

    @Override
    public void destroy() throws Exception {
        if(this.consumerConfig != null) {
            consumerConfig.unRefer();   // 注销引用。
        }
    }
}
```

特殊说明

调用 `$genericInvoke(String methodName, String[] argTypes, Object[] args)` 接口，会将除以下包以外的其他类序列化为 `GenericObject` 。

```
"com.sun",
"java",
"javax",
"org.ietf",
"org.ogm",
"org.w3c",
"org.xml",
"sunw.io",
"sunw.util"
```

3.9.2.6. 序列化协议

SOFARPC 在使用 Bolt 通信协议的情况下，可以选择不同的序列化协议。目前支持 hessian2 和 protobuf。

默认情况下，SOFARPC 使用 hessian2 作为序列化协议。如果您需要将序列化协议设置成 protobuf，需要在发布服务时进行如下设置：

在 `<sofa:binding.bolt>` 标签内增加 `<sofa:global-attrs>` 标签，并且设置 `serialize-type` 属性为 `protobuf` 。

```
<sofa:service ref="sampleService" interface="com.alipay.sofarpc.demo.SampleService">
    <sofa:binding.bolt>
        <sofa:global-attrs serialize-type="protobuf"/>
    </sofa:binding.bolt>
</sofa:service>
```

在引用服务时，您也需要将序列化协议改成 `protobuf` ，设置方式和发布服务的时候类似。

```
<sofa:reference interface="com.alipay.sofarpc.demo.SampleService" id="sampleServiceRef" jvm
-first="false">
    <sofa:binding.bolt>
        <sofa:global-attrs serialize-type="protobuf"/>
    </sofa:binding.bolt>
</sofa:reference>
```

3.9.2.7. 自定义线程池

SOFARPC 支持自定义业务线程池，可以为指定服务设置一个与 SOFARPC 业务线程池隔离的独立业务线程池。多个服务可以共用一个独立的线程池。

> 说明
>
> SOFARPC 要求自定义线程池的类型必须是 `com.alipay.sofa.rpc.server.UserThreadPool` 。

XML 方式

如果采用 XML 的方式发布服务，您可以先设定一个 class 为 `com.alipay.sofa.rpc.server.UserThreadPool` 的线程池的 Bean，然后设置到 `<sofa:global-attrs>` 标签的 `thread-pool-ref` 属性中。

```
<bean id="helloService" class="com.alipay.sofa.rpc.quickstart.HelloService"/>

<!-- 自定义一个线程池 -->
<bean id="customExecutor" class="com.alipay.sofa.rpc.server.UserThreadPool" init-method="in
it">
    <property name="corePoolSize" value="10"/>
    <property name="maximumPoolSize" value="10"/>
    <property name="queueSize" value="0"/>
</bean>

<sofa:service ref="helloService" interface="XXXService">
    <sofa:binding.bolt>
       <!-- 将线程池设置给一个 Service -->
       <sofa:global-attrs thread-pool-ref="customExecutor"/>
    </sofa:binding.bolt>
</sofa:service>
```

Annotation 方式

如果是采用 Annotation 的方式发布服务，您可以通过设置 `@SofaServiceBinding` 的 `userThreadPool` 属性来设置自定义线程池的 Bean。

```
@SofaService(bindings ={@SofaServiceBinding(bindingType ="bolt", userThreadPool ="customThr
eadPool")})
public class SampleServiceImpl implements SampleService{
}
```

在 Spring 环境使用 API 方式

如果是在 Spring 环境下使用 API 的方式发布服务，您可以通过调用 `BoltBindingParam` 的 `setUserThreadPool` 方法来设置自定义线程池。

```
BoltBindingParam boltBindingParam =new BoltBindingParam();
boltBindingParam.setUserThreadPool(new UserThreadPool());
```

在非 Spring 环境下使用 API 方式

如果是在非 Spring 环境下使用 API 的方式，您可以通过如下的方式来设置自定义线程池。

```
UserThreadPool threadPool =new UserThreadPool();
threadPool.setCorePoolSize(10);
threadPool.setMaximumPoolSize(100);
threadPool.setKeepAliveTime(200);
threadPool.setPrestartAllCoreThreads(false);
threadPool.setAllowCoreThreadTimeOut(false);
threadPool.setQueueSize(200);

UserThreadPoolManager.registerUserThread(ConfigUniqueNameGenerator.getUniqueName(providerCo
nfig), threadPool);
```

3.9.3. RESTful 协议

3.9.3.1. RESTful 协议的基本用法

在 SOFARPC 中，使用不同的通信协议即使用不同的 Binding。如果需要使用 RESTful 协议，只要将 Binding 设置为 REST 即可。

发布服务

1. 定义服务接口。

   在定义 RESTful 服务接口时，需要采用 JAXRS 标准的注解在接口上加上元信息，示例如下：

   ```
   @Path("sample")
       public interface SampleService{
           @GET
           @Path("hello")
           String hello();
       }
   ```

   > 说明
   >
   > JAXRS 的标准的注解的使用方式可以参考 RESTEasy 的文档。

2. 发布服务。

   定义好接口之后，你可以将接口的实现发布成一个服务。以 Annotation 方式发布服务为例：

   ```
   @Service
   @SofaService(bindings ={@SofaServiceBinding(bindingType ="rest")})
   public class RestfulSampleServiceImpl implements SampleService{
       @Override
       public String hello(){
           return "Hello";
       }
   }
   ```

如果您要通过其他方式发布服务，请参考 BOLT 协议基本使用。

通过浏览器访问服务

服务发布后，您可以通过浏览器直接访问服务。以上文发布的服务为例，SOFARPC 的 RESTful 服务的默认端口为 8341，访问地址如下：

```
http://localhost:8341/sample/hello
```

引用服务

您也可以通过 SOFARPC 标准的服务引用的方式来引用服务。以 Annotation 方式引用服务为例：

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="rest"))
private SampleService sampleService;
```

如果您要通过其他方式引用服务，请参考 BOLT 协议基本使用。

3.9.3.2. REST 跨域

对于 REST，SOFARPC 内置了一个跨域 Filter，您可以以此实现跨域访问。

SOFARPC API 方式

对于使用 SOFARPC API 的用户，可以在 ServerConfig 中添加如下参数，实现跨域访问。

```
Map<String,String> parameters=new HashMap<String,String>()
parameters.put(RpcConstants.ALLOWED_ORIGINS,"abc.com,cdf.com");
serverConfig.setParameters(parameters);
```

XML 方式

对于 XML 方式，直接通过如下配置即可。

```
com.alipay.sofa.rpc.rest.allowed.origins=a.com,b.com
```

3.9.3.3. REST 自定义 Filter

对于 REST，SOFAStack 支持 JAXRSProviderManager 管理器类，您可以根据这个管理器类自定义 Filter。

JAXRSProviderManager 管理器类在服务端生效，生效时间为服务启动时。接口如下：

```
com.alipay.sofa.rpc.server.rest.RestServer#registerProvider
```

对于您自定义的 Filter 类，可以在初始化完成后，调用如下类进行注册。

```
com.alipay.sofa.rpc.config.JAXRSProviderManager#registerCustomProviderInstance
```

其中，自定义的 Filter 遵循 REST 的规范，需要实现如下接口：

```
javax.ws.rs.container.ContainerResponseFilter
或者
javax.ws.rs.container.ContainerRequestFilter
```

REST server 启动之后，对于裸 SOFARPC 的使用，您需要先注册，再启动服务。对于 SOFABoot 环境下的使用，也是类似的过程，具体的写法可以参考如下示例：

```
com.alipay.sofa.rpc.server.rest.TraceRequestFilter
com.alipay.sofa.rpc.server.rest.TraceResponseFilter
```

3.9.3.4. 集成 SOFARPC RESTful 服务和 Swagger

本文介绍如何集成SOFARPC RESTful 服务和 Swagger。

使用 rpc-sofa-boot-starter 6.0.1 以上版本

从 `rpc-sofa-boot-starter` 6.0.1 版本开始，SOFARPC 提供了 RESTful 服务和 Swagger 的一键集成能力。操作步骤如下：

1. 在 `pom.xml` 中增加 Swagger 的依赖。

```
<dependency>
    <groupId>io.swagger.core.v3</groupId>
    <artifactId>swagger-jaxrs2</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>20.0</version>
</dependency>
```

2. 在 `application.properties` 里面增加 `com.alipay.sofa.rpc.restSwagger=true` 。

3. 访问 `http://localhost:8341/swagger/openapi` 即可获取 SOFARPC 的 RESTful 的 Swagger OpenAPI 内容。

不使用 rpc-sofa-boot-starter 或版本低于 6.0.1

如果没有使用 `rpc-sofa-boot-starter` 或者 `rpc-sofa-boot-starter` 的版本低于 6.0.1，可以采用如下的方式集成 Swagger:

1. 在应用中引入 Swagger 相关的依赖。

   由于 SOFARPC 的 RESTful 协议使用的是 JAXRS 标准，因此只需引入 Swagger 的 JAXRS 依赖。依赖如下：

```
<dependency>
      <groupId>io.swagger.core.v3</groupId>
      <artifactId>swagger-jaxrs2</artifactId>
      <version>2.0.0</version>
</dependency>
<dependency>
     <groupId>com.google.guava</groupId>
     <artifactId>guava</artifactId>
     <version>20.0</version>
</dependency>
```

   > **说明**
   >
   > 引入 Guava 的 20.0 的版本是为了解决 Guava 的版本冲突。

2. 发布一个 Swagger 的 RESTful 服务。

   为了让 Swagger 能够将 SOFARPC 的 RESTful 的服务通过 Swagger OpenAPI 暴露出去，可以反向使用用 SOFARPC 的 RESTful 的服务提供 Swagger 的 OpenAPI 服务。

i. 新建一个接口。

```
@Path("swagger")
public interface OpenApiService{
    @GET
    @Path("openapi")
    @Produces("application/json")
    String openApi();
}
```

ii. 提供一个实现类，并发布成 SOFARPC 的 RESTful 的服务。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="rest")}, interfaceType =O
penApiService.class)
public class OpenApiServiceImpl implements OpenApiService,InitializingBean{
     private OpenAPI openAPI;

    @Override
    public String openApi(){
         return Json.pretty(openAPI);
    }

    @Override
    public void afterPropertiesSet(){
    List<Package> resources =new ArrayList<>();
    resources.add(this.getClass().getPackage());// 扫描当前类所在的 Package，也可以扫描
其他的 SOFARPC RESTful 服务接口所在的 Package
    if(!resources.isEmpty()){
         // init context
        try{
            SwaggerConfiguration oasConfig =new SwaggerConfiguration()
            .resourcePackages(resources.stream().map(Package::getName).collect(Coll
ectors.toSet()));

            OpenApiContext oac =new JaxrsOpenApiContextBuilder()
            .openApiConfiguration(oasConfig)
            .buildContext(true);
            openAPI = oac.read();
        }catch(OpenApiConfigurationException e){
           throw new RuntimeException(e.getMessage(), e);
       }
    }
  }
}
```

3. 应用启动后，访问 `http://localhost:8341/swagger/openapi` 即可得到当前的应用发布的所有的 RESTful 的服务的信息。

解决跨域问题

如果您在另外一个端口中启动了一个 Swagger UI，并且希望通过 Swagger UI 访问 `http://localhost:8341/swagger/openapi` 查看 API 定义，发起调用，可能需要解决访问跨域的问题。您可以在应用启动前增加如下代码：

```
import org.jboss.resteasy.plugins.interceptors.CorsFilter;

public static void main(String[] args){
    CorsFilter corsFilter =newCorsFilter();
    corsFilter.getAllowedOrigins().add("*");
    JAXRSProviderManager.registerCustomProviderInstance(corsFilter);
    SpringApplication.run(DemoApplication.class, args);
}
```

3.9.4. Dubbo 协议

3.9.4.1. Dubbo 协议的基本使用

在 SOFARPC 中，使用不同的通信协议只要设置使用不同的 Binding 即可。如果需要使用 Dubbo 协议，只要将 Binding 设置为 Dubbo 即可。

本文以注解的使用方式为例，其他使用方式可以参考 BOLT 协议基本使用。

发布服务

发布一个 Dubbo 的服务，只需要将 `@SofaServiceBinding` 的 `bindingType` 设置为 `dubbo` 。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="dubbo")})
public class SampleServiceImpl implements SampleService{
}
```

引用服务

引用一个 Dubbo 的服务，只需要将 `@SofaReferenceBinding` 的 `bindingType` 设置为 `dubbo` 。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="dubbo"), jvmFirst =false)
private SampleService sampleService;
```

设置 Dubbo 服务的 Group

在 SOFARPC 的模型中，没有直接的字段叫做 Group，但 SOFARPC 有一个 `uniqueId` 模型，可以直接映射到 Dubbo 的模型中的 Group。比如下面的代码，就是发布了一个 Group 为 `groupDemo` 的服务。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="dubbo")}, uniqueId ="groupDemo")
public class SampleServiceImpl implements SampleService{
}
```

如下代码引用了一个 Group 为 `groupDemo` 的服务。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="dubbo"), uniqueId ="groupDemo",
jvmFirst =false)
private SampleService sampleService;
```

> **重要**
>
> 目前 Dubbo 协议只支持 Zookeeper 作为服务注册中心。

3.9.5. H2C 协议

3.9.5.1. H2C 协议的基本使用

在 SOFARPC 中，使用不同的通信协议只要设置使用不同的 Binding 即可。如果需要使用 H2C 协议，只要将 Binding 设置为 H2C。

本文以注解的使用方式为例，其他使用方式可以参考 BOLT 协议基本使用。

发布服务

发布一个 H2C 的服务，只需要将 `@SofaServiceBinding` 的 `bindingType` 设置为 `h2c` 。

```
@Service
@SofaService(bindings ={@SofaServiceBinding(bindingType ="h2c")})
public class SampleServiceImpl implements SampleService{
}
```

引用服务

引用一个 H2C 的服务，只需要将 `@SofaReferenceBinding` 的 `bindingType` 设置为 `h2c` 。

```
@SofaReference(binding =@SofaReferenceBinding(bindingType ="h2c"), jvmFirst =false)
private SampleService sampleService;
```

3.9.6. HTTP 协议

3.9.6.1. HTTP 协议的基本使用

在 SOFARPC（非 SOFABoot 环境）中使用 HTTP 作为服务端协议时，支持 JSON 以序列化方式作为一些基础的测试方式使用。

发布服务

```
// 只有1个线程执行。
ServerConfig serverConfig = new ServerConfig()
.setStopTimeout(60000)
.setPort(12300)
.setProtocol(RpcConstants.PROTOCOL_TYPE_HTTP)
.setDaemon(true);

// 发布一个服务，每个请求要执行 1 秒。
ProviderConfig<HttpService> providerConfig = new ProviderConfig<HttpService>()
.setInterfaceId(HttpService.class.getName())
.setRef(new HttpServiceImpl())
.setApplication(new ApplicationConfig().setAppName("serverApp"))
.setServer(serverConfig)
.setUniqueId("uuu")
.setRegister(false);
providerConfig.export();
```

引用服务

因为是 HTTP + JSON，所以引用方可以直接通过 HttpClient 进行调用，以下为一段测试代码。

```
private ObjectMapper mapper =new ObjectMapper();
HttpClient httpclient =HttpClientBuilder.create().build();
// POST 正常请求。
String url ="http://127.0.0.1:12300/com.alipay.sofa.rpc.server.http.HttpService:uuu/object"
;
HttpPost httpPost =new HttpPost(url);
httpPost.setHeader(RemotingConstants.HEAD_SERIALIZE_TYPE,"json");
ExampleObj obj =new ExampleObj();
obj.setId(1);
obj.setName("xxx");
byte[] bytes = mapper.writeValueAsBytes(obj);
ByteArrayEntity entity =new ByteArrayEntity(bytes,
ContentType.create("application/json"));

httpPost.setEntity(entity);

HttpResponse httpResponse = httpclient.execute(httpPost);
Assert.assertEquals(200, httpResponse.getStatusLine().getStatusCode());
byte[] data =EntityUtils.toByteArray(httpResponse.getEntity());

ExampleObj result = mapper.readValue(data,ExampleObj.class);

Assert.assertEquals("xxxxx", result.getName());
```

3.10. 链路说明

3.10.1. 链路追踪

SOFARPC 集成了 SOFATracer 的功能，可以输出链路中的数据信息。默认开启。

默认为 `JSON` 数据格式，具体的字段含义解释如下：

RPC 客户端摘要日志（ rpc-client-digest.log）

日志示例如下：


```
{"timestamp":"2018-05-20 17:03:20.708","tracerId":"1e27326d152680700049810018559 7","spanId":"0","span.kind":"client","local.app":"SOFATracerRPC","protocol":"bolt","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","current.thread.name":"main","invoke.type":"sync","router.record":"DIRECT","remote.ip":"127.0.0.1:12200","local.client.ip":"127.0.0.1","result.code":"00","req.serialize.time":"33","resp.deserialize.time":"39","resp.size":"170","req.size":"582","client.conn.time":"0","client.elapse.time":"155","local.client.port":"59774","baggage":""}
```


参数	说明
timestamp	日志打印时间
tracerId	请求的 TracerId。
spanId	请求的 SpanId。
span.kind	Span 类型。
local.app	当前 App 名称。
protocol	协议类型。
service	服务接口信息。
method	方法名。
current.thread.name	当前线程名。
invoke.type	调用类型。
router.record	路由记录
remote.ip	目标 IP。

参数	说明
local.client.ip	本机 IP。
result.code	返回码。
req.serialize.time	请求序列化时间。
resp.deserialize.time	响应反序列化时间。
resp.size	响应大小，单位Byte。
req.size	请求大小，单位 Byte。
client.conn.time	客户端连接耗时。
client.elapse.time	客户端调用总耗时。
local.client.port	本地客户端端口。
baggage	透传的 baggage 数据 （KV 格式）。

RPC 服务端摘要日志（ rpc-server-digest.log）

日志示例如下：

{"timestamp":"2018-05-20 17:00:53.312","tracerId":"1e27326d1526806853032100185011","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"3","biz.impl.time":"0","resp.serialize.time":"4","req.deserialize.time":"38","resp.size":"170","req.size":"582","baggage":"",{"timestamp":"2018-05-20 17:03:05.646","tracerId":"1e27326d1526806985394100185589","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"2","biz.impl.time":"1","resp.serialize.time":"1","req.deserialize.time":"6","resp.size":"170","req.size":"582","baggage":"",{"timestamp":"2018-05-20 17:03:20.701","tracerId":"1e27326d1526807000498100185597","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"2","biz.impl.time":"0","resp.serialize.time":"1","req.deserialize.time":"4","resp.size":"170","req.size":"582","baggage":"",{"timestamp":"2018-05-20 17:04:19.966","tracerId":"1e27326d1526807046606100185635","spanId":"0","span.kind":"server","service":"com.alipay.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"SOFATracerRPC","protocol":"bolt","local.app":"SOFATracerRPC","current.thread.name":"SOFA-BOLT-BIZ-12200-5-T1","result.code":"00","server.pool.wait.time":"2","biz.impl.time":"0","resp.serialize.time":"1","req.deserialize.time":"4","resp.size":"170","req.size":"582","baggage":""}

参数	说明
timestamp	日志打印时间。
tracerId	请求的 TracerId。
spanId	请求的 SpanId。
span.kind	Span 类型。
server	服务接口信息。
method	方法名。
remote.ip	来源 IP。
remote.app	来源 App 名称。

参数	说明
protocol	协议类型。
local.app	本地 App 名称。
current.thread.name	当前线程名。
result.code	返回码。
server.pool.wait.time	服务端线程池等待时间。
biz.impl.time	业务处理耗时。
resp.serialize.time	响应序列化时间。
req.deserialize.time	请求反序列化时间。
resp.size	响应大小，单位 Byte。
req.size	请求大小，单位 Byte。
baggage	透传的 baggage 数据（KV 格式）。

RPC 客户端统计日志（ rpc-client-stat.log）

日志示例如下：

```
{"time":"2018-05-18 07:02:19.717","stat.key":{"method":"method","local.app":"client","servi
ce":"app.service:1.0"},"count":10,"total.cost.milliseconds":17,"success":"Y"}
```

参数	说明
time	日志打印时间。
stat.key	日志关键 key。

参数	说明
method	方法信息。
local.app	客户端 App 名称。
service	服务接口信息。
count	调用次数。
total.cost.milliseconds	总耗时。
success	调用结果。

RPC 服务端统计日志（ rpc-server-stat.log）

日志示例如下：

```
{"time":"2018-05-18 07:02:19.717","stat.key":{"method":"method","local.app":"client","service":"app.service:1.0"},"count":10,"total.cost.milliseconds":17,"success":"Y"}
```

参数	说明
time	日志打印时间。
stat.key	日志关键 key。
method	方法信息。
local.app	客户端 App 名称。
service	服务接口信息。
count	调用次数。
total.cost.milliseconds	总耗时。

参数	说明
success	调用结果。

3.10.2. 链路数据透传

链路数据透传功能支持应用向调用上下文中存放数据，使得整个链路上的应用都可以操作该数据。

您可以分别向链路的 request 和 response 中放入数据进行透传，并可获取到链路中相应的数据。使用方式如下：

```
RpcInvokeContext.getContext().putRequestBaggage("key_request","value_request");
RpcInvokeContext.getContext().putResponseBaggage("key_response","value_response");

String requestValue=RpcInvokeContext.getContext().getRequestBaggage("key_request");
String responseValue=RpcInvokeContext.getContext().getResponseBaggage("key_response");
```

使用示例

例如数据从 A 到 B，再到 C 的场景中，将 A 设置的请求隐式传参数据传递给 B 和 C。在返回的时候，将 C 和 B 的响应隐式传参数据传递给 A。

A 请求方设置：

```
// 调用前设置请求透传的值。
RpcInvokeContext context =RpcInvokeContext.getContext();
context.putRequestBaggage("reqBaggageB","a2bbb");
// 调用。
String result = service.hello();
// 拿到结果透传的值。
context.getResponseBaggage("respBaggageB");
```

B 业务代码：

```
public String hello(){
    // 拿到请求透传的值。
    RpcInvokeContext context =RpcInvokeContext.getContext();
    String reqBaggage = context.getRequestBaggage("reqBaggageB");
    doSomthing();
    // 结果透传一个值。
    context.putResponseBaggage("respBaggageB","b2aaa");
    return result;
}
```

如果中途自己启动了子线程，则需要设置子线程的上下文：

```
CountDownLatch latch =new CountDownLatch(1);
finalRpcInvokeContext parentContext =RpcInvokeContext.peekContext();
Thread thread =new Thread(new Runnable(){
    public void run(){
          try{
                RpcInvokeContext.setContext(parentContext);
                // 调一个远程服务。
                xxxService.sayHello();
                latch.countDown();
          }finally{
               RpcInvokeContext.removeContext();
          }
    }
},"new-thread");
thread.start();

// 此时拿不到返回值透传的数据的。
latch.await();//等待
// 此时返回结束，能拿到返回透传的值。
```

和 SOFATracer 的比较

SOFATracer 是蚂蚁开源的一个分布式链路追踪系统，RPC 目前已经集成 Tracer，默认开启。

RPC 和 Tracer 进行数据传递不同点如下：

- RPC 的数据透传更偏向业务使用，而且可以在全链路中进行双向传递。调用方可以传给服务方，服务方也可以传递信息给调用方。SOFATracer 更加偏向于中间件和业务无感知的数据的传递，只能进行单向传递。
- RPC 的透传可以选择性地不在全链路中透传，而 Tracer 中如果传递大量信息，会在整个链路中传递，可能对下游业务会有影响。

所以整体来看，两者各有利弊，在有一些和业务相关的透传数据的情况下，可以选择 RPC 的透传。

3.10.3. 调用上下文

RPC 上下文中存放了当前调用过程中的一些其他信息，如服务提供方应用名、IP。应用开发人员可以获取这些信息做一些业务上的操作。

RPC 提供获取单次调用上下文的工具类 `com.alipay.sofa.rpc.api.context.RpcContextManager`，您可以通过该类获得最后一次 Reference 以及当次 Service 的相关信息。

> 重要
>
> RPC 上下文是存放在 `ThreadLocal` 中的临时数据，切换线程或者清空 `ThreadLocal` 后数据都将丢失。

使用方式

```
// Reference Context
SampleService sampleService;
public void do(){
   sampleService.hello();
   // 参数为 true 代表清空上下文信息。
   RpcReferenceContext referenceContext =RpcContextManager.lastReferenceContext(true);
  // do something on referenceContext
}
```

```
// Service Context
public void doService(){
   // do sth
   ...
  // 参数为 true 代表清空上下文信息。
  RpcServiceContext serviceContext =RpcContextManager.currentServiceContext(true);
  // do something on serviceContext
  ...
}
```

上下文内容

RPC 上下文的信息均是从 Tracer 中获得，包含如下内容：

Reference

参数	说明
traceId	Trace ID 名称。
rpcId	RPC ID 名称。
interfaceName	服务接口。
methodName	服务方法。
uniqueId	服务的唯一标识。
serviceName	唯一的服务名。
isGeneric	是否为泛化调用。
targetAppName	服务提供方的应用名。

参数	说明
targetUrl	服务提供方的地址。
protocol	调用协议，例如 TR。
invokeType	调用类型，例如 sync、oneway 等。
routeRecord	路由寻址链路，例如 `TURL>CFS>RDM`，表示路由寻址路径是从 `test-url` 到软负载到随机寻址。如果上次请求的路由策略是 `test-url`，则 `routeRecord` 等于 `TURL>RDM`。 如要判断 `test-url` 或者软负载是否生效，请使用 `RpcReferenceContext.isTestUrlValid` 或者 `RpcReferenceContext.isConfigServerValid` 方法。详细的路由规则，请参见 RPC 路由。
costTime	调用耗时，单位为 ms。
resultCode	**结果码，取值如下：** • **00：表示成功。** • **01：表示业务异常。** • **02：表示 RPC 框架错误。** • **03：表示出现超时失败错误。** • **04：表示路由失败。**

Service

参数	说明
traceId	Trace ID 名称。
rpcId	RPC ID 名称。

参数	说明
methodName	服务方法。
serviceName	唯一的服务名。
callerAppName	服务消费方的应用名。
callerUrl	服务消费方的地址。

更多信息，请参见 SOFARPC 日志。

3.11. 高级功能

3.11.1. Node 跨语言调用

本文介绍如何通过 NodeJs 调用 SOFARPC服务。

安装 SOFARPC Nodejs

执行以下命令安装 SOFARPC Nodejs：

```
$ npm install sofa-rpc-node --save
```

更多信息请参见 GitHub。

代码示例

暴露 RPC 服务，并发布到注册中心

```
'use strict';

const{RpcServer}= require('sofa-rpc-node').server;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 创建 zk 注册中心客户端。
const registry =newZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',// 需要本地启动一个 zkServer
});

// 创建 RPC Server 实例。
const server =newRpcServer({
  logger,
  // 传入注册中心客户端。
  registry,
  port:12200,
});

// 添加服务。
server.addService({
  interfaceName:'com.nodejs.test.TestService',
},{
  async plus(a, b){
return a + b;
},
});

// 启动 Server 并发布服务。
server.start()
.then(()=>{
    server.publish();
});
```

调用 RPC 服务（从注册中心获取服务列表）

```
'use strict';

const{RpcClient}= require('sofa-rpc-node').client;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 创建 zk 注册中心客户端。
const registry =new ZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',
});

async function invoke(){
// 创建 RPC Client 实例。
const client =new RpcClient({
    logger,
    registry,
});
// 创建服务的 consumer。
const consumer = client.createConsumer({
    interfaceName:'com.nodejs.test.TestService',
});
// 等待 consumer ready（从注册中心订阅服务列表...）。
  await consumer.ready();

// 执行泛化调用。
const result = await consumer.invoke('plus',[1,2],{ responseTimeout:3000});
  console.log('1 + 2 = '+ result);
}

invoke().catch(console.error);
```

调用 RPC 服务（直连模式）

```
'use strict';

const{RpcClient}= require('sofa-rpc-node').client;
const logger = console;

async function invoke(){
// 不需要传入 registry 实例了。
const client =newRpcClient({
    logger,
});
const consumer = client.createConsumer({
    interfaceName:'com.nodejs.test.TestService',
    // 直接指定服务地址。
    serverHost:'127.0.0.1:12200',
});
  await consumer.ready();

const result = await consumer.invoke('plus',[1,2],{ responseTimeout:3000});
  console.log('1 + 2 = '+ result);
}

invoke().catch(console.error);
```

暴露和调用 protobuf 接口

接口定义

通过 `*.proto` 定义接口。

```
syntax ="proto3";

package com.alipay.sofa.rpc.test;

// 可选
option java_multiple_files =false;

service ProtoService{
  rpc echoObj (EchoRequest) returns (EchoResponse){}
}

message EchoRequest{
   string name =1;
   Groupgroup=2;
}

message EchoResponse{
  int32 code =1;
  string message =2;
}

enumGroup{
  A =0;
  B =1;
}
```

服务端代码

```
'use strict';

const antpb = require('antpb');
const protocol = require('sofa-bolt-node');
const{RpcServer}= require('sofa-rpc-node').server;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 传入 *.proto 文件存放的目录，加载接口定义。
const proto = antpb.loadAll('/path/proto');
// 将 proto 设置到协议中。
protocol.setOptions({ proto });

const registry =new ZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',
});

const server =new RpcServer({
  logger,
  // 覆盖协议。
  protocol,
  registry,
  // 设置默认的序列化方式为 protobuf。
  codecType:'protobuf',
  port:12200,
});

server.addService({
  interfaceName:'com.alipay.sofa.rpc.test.ProtoService',
},{
  async echoObj(req){
    req = req.toObject({ enums:String});
return{
      code:200,
      message:'hello '+ req.name +', you are in '+ req.group,
};
},
});
server.start()
.then(()=>{
    server.publish();
});
```

客户端代码

```
'use strict';

const antpb = require('antpb');
const protocol = require('sofa-bolt-node');
const{RpcClient}= require('sofa-rpc-node').client;
const{ZookeeperRegistry}= require('sofa-rpc-node').registry;
const logger = console;

// 传入 *.proto 文件存放的目录，加载接口定义。
const proto = antpb.loadAll('/path/proto');
// 将 proto 设置到协议中。
protocol.setOptions({ proto });

const registry =new ZookeeperRegistry({
  logger,
  address:'127.0.0.1:2181',
});

async function invoke(){
const client =new RpcClient({
    logger,
    protocol,
    registry,
});
const consumer = client.createConsumer({
    interfaceName:'com.alipay.sofa.rpc.test.ProtoService',
});
  await consumer.ready();

const result = await consumer.invoke('echoObj',[{
    name:'gxcsoccer',
    group:'B',
}],{ responseTimeout:3000});
  console.log(result);
}

invoke().catch(console.error);
```

3.11.2. 多注册中心注册

同一个服务可以注册多个注册中心，您可以通过以下方式实现：

XML 方式

1. 在 `application.properties` 中配置注册中心源信息。

   配置格式为：`com.alipay.sofa.rpc.registries.<自定义别名>=协议://地址`，支持的协议为：dsr、sofa、zookeeper、consul、gateway、mesh、multicast、local、nacos。

   配置示例如下：

```
com.alipay.sofa.rpc.registries.registryName1=dsr://10.0.0.1:9600
com.alipay.sofa.rpc.registries.registryName2=dsr://10.0.0.2:9600
```

2. 配置需要多注册中心的服务。

```
<sofa:service interface="com.alipay.sofa.facade.SampleService" ref="sampleService">
        <sofa:binding.bolt>
            <sofa:global-attrs registry="registryName1,registryName2"/>
        </sofa:binding.bolt>

    </sofa:service>
```

API 方式

1. 构建多个 `RegistryConfig`，并设置给 `ProviderConfig`。

   示例如下：

```
List<RegistryConfig> registryConfigs =new ArrayList<RegistryConfig>();
registryConfigs.add(registryA);
registryConfigs.add(registryB);
providerConfig.setRegistry(registryConfigs);
```

2. 调用 `MethodConfig` 对象相应的 `set` 方法设置对应的参数。

   示例如下：

```
MethodConfig methodConfigA =new MethodConfig();
MethodConfig methodConfigB =new MethodConfig();
List<MethodConfig> methodConfigs =new ArrayList<MethodConfig>();
methodConfigs.add(methodConfigA);
methodConfigs.add(methodConfigB);
providerConfig.setMethods(methodConfigs);     //服务端设置
consumerConfig.setMethods(methodConfigs);  //客户端设置
```

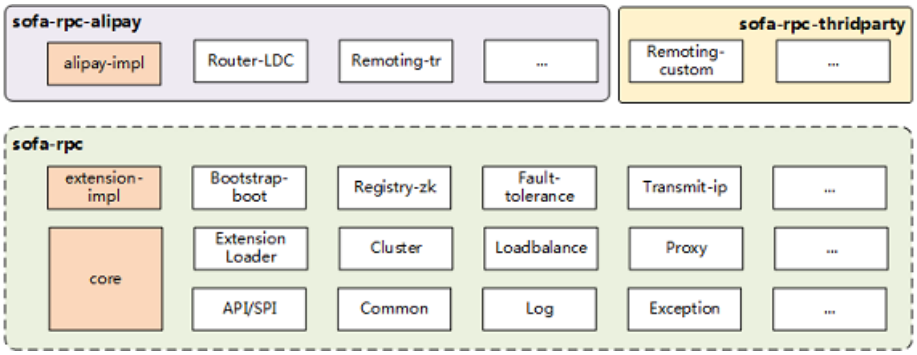
3.12. 自定义扩展

3.12.1. 架构模块介绍

本文介绍 SOFARPC 的架构模块。

工程架构

SOFARPC 架构如下所示：



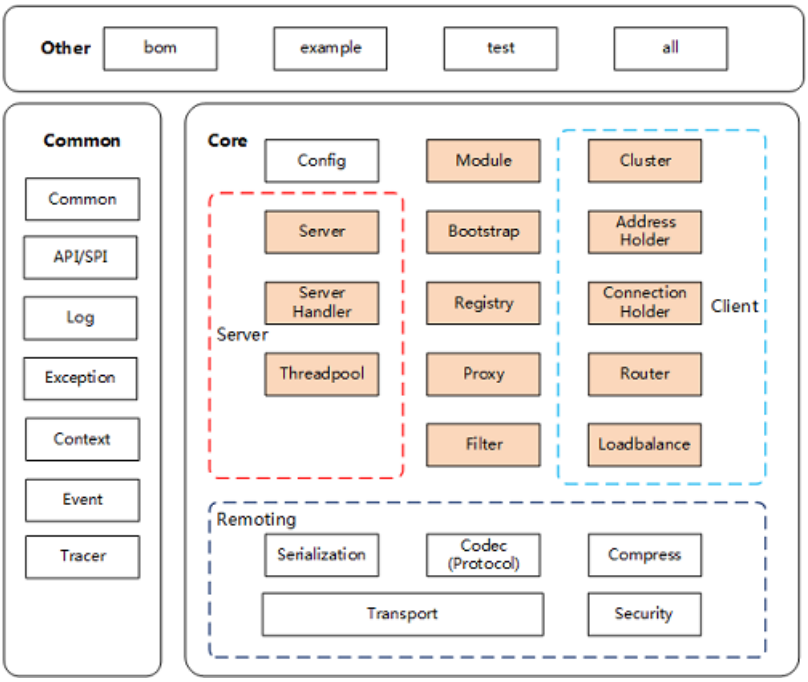
SOFARPC 从下到上分为两层：

- 核心层：包含了 RPC 的核心组件（例如各种接口、API、公共包）以及一些通用的实现（例如随机等负载均衡算法）。
- 功能实现层：所有的功能实现层的用户都是平等的，都是基于扩展机制实现的。

模块划分

SOFARPC 各个模块的实现类都只在自己模块中出现，一般不交叉依赖。需要交叉依赖的实现类已经全部抽象到 core 或者 common 模块中。

模块划分如下：



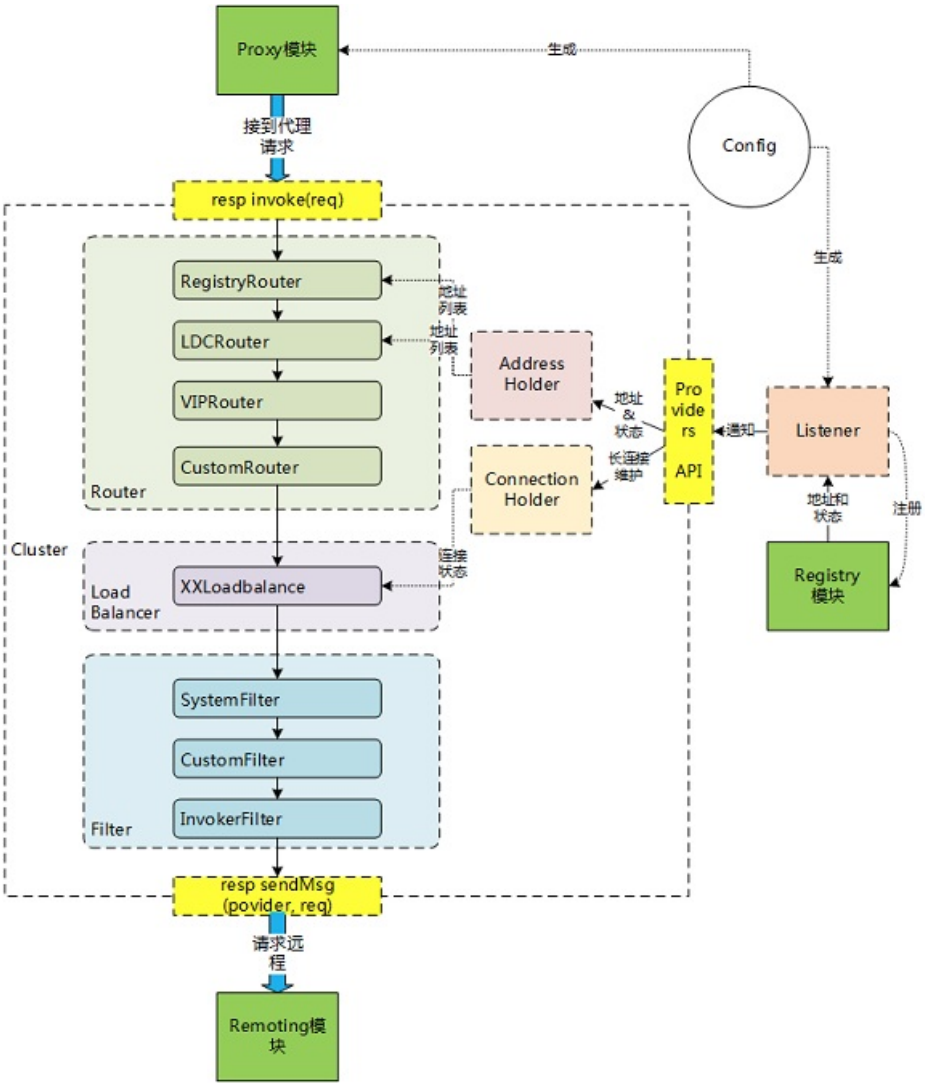
模块名	子模块名	中文名	说明	依赖
all	-	发布打包模块	需要打包的全部模块。	all
bom	-	依赖管控模块	依赖版本管控。	无

模块名	子模块名	中文名	说明	依赖
example	-	示例模块	-	all
test	-	测试模块	包含集成测试	all
core	api	API模块	各种基本流程接口、消息、上下文、扩展接口等。	common
core	common	公共模块	utils、数据结构。	exception
core	exception	异常模块	各种异常接口等。	common
bootstrap	-	启动实现模块	启动类，发布或者引用服务逻辑以及 registry 的操作。	core
proxy	-	代理实现模块	接口实现代理生成。	core
client	-	客户端实现模块	发送请求、接收响应、连接维护、路由、负载均衡、同步异步等。	core
server	-	服务端实现模块	启动监听、接收请求，发送响应、业务线程分发等。	core
filter	-	拦截器实现模块	服务端和客户端的各种拦截器实现。	core
codec	-	编解码实现模块	例如压缩，序列化等。	core
protocol	-	协议实现模块	协议的包装处理、协商。	core

模块名	子模块名	中文名	说明	依赖
transport	-	网络传输实现模块	TCP 连接的建立、数据分包粘包处理、请求响应对象分发等。	core
registry	-	注册中心实现模块	实现注册中心，例如 zk。	core

3.12.2. 客户端调用流程

客户端模块是一个较复杂的模块，这里包含了集群管理、路由、地址管理器、连接管理器、负载均衡器，还与代理、注册中心等模块交互。



3.12.3. 关键类介绍

本文介绍在使用 SOFARPC 进行开发时会涉及的关键类。

消息

内部消息全部使用 `SofaRequest` 和 `SofaResponse` 进行传递。如果您需要转换为其它协议，请在真正调用和收到请求的时候，转换为实际要传输的对象。

可以对 `SofaRequest` 和 `SofaResponse` 进行写入操作的模块如下：

- Invoker
- Filter
- ServerHandler
- Serialization

对消息体是只读的模块如下：

- Cluster

- Router
- LoadBalance

日志

日志的初始化也是基于扩展机制，但由于日志先加载，所以在 `rpc-config.json` 里有一个单独的 Key：

```
{
//日志实现。日志早于配置加载，所以不能适应Extension机制。
"logger.impl":"com.alipay.sofa.rpc.log.MiddlewareLoggerImpl"
}
```

配置项

使用者的 RPC 配置

用户的配置。例如端口配置（虽然已经开放对象中设置端口的字段，但是 SOFA 默认是从配置文件里读取）、线程池大小配置等。

- 通过 `SofaConfigs` 加载配置，调用 `ExternalConfigLoader` 读取外部属性。
- 通过 `SofaConfigs` 提供的 API 进行获取。
- 所有内部配置的 Key 都在 `SofaOptions` 类。
- 优先级：`System.property` > `sofa-config.properties`（每个应用一个）> `rpc-config.properties`。

RPC 框架配置

框架自身的配置。例如默认序列化、默认超时等。

- 通过 `RpcConfigs` 加载配置文件。
- 通过 `RpcConfigs` 其提供的 API 进行获取和监听数据变化。
- 所有内部配置的 Key 都在 `RpcOptions` 类。
- 优先级：`System.property` > `custom rpc-config.json`（可能存在多个自定义，会排序）> `rpc-config-default.json`。

常量

- 全局的基本常量在 `RpcConstants` 中，例如：
  - 调用方式：sync、oneway。
  - 协议：bolt、grpc。
  - 序列化：hessian、java、protobuf。
  - 上下文的 Key。

- 如果扩展实现自身的常量，请自行维护。

  例如 BOLT 协议的常量：

  - `SERIALIZE_CODE_HESSIAN = 1`
  - `PROTOCOL_TR = 13`

  例如 DSR 配置中心相关的常量 `_WEIGHT`、`_CONNECTTIMEOUT`，这种配置中心特有的 key。

地址

地址信息放到 `ProviderInfo` 类中。`ProviderInfo` 的值主要分为三部分：

- 字段：一般是一些必须项目，例如 IP、端口、状态等。
- 静态字段：例如应用名。
- 动态字段：例如预热权重等。

字段枚举维护在 `ProviderInfoAttrs` 类中。

3.12.4. 关键配置类介绍

本文介绍 RPC 的关键配置类，您可以基于这些配置类进行开发。

ProviderConfig

属性	默认值	说明
id	自动生成	ID
application	空 ApplicationConfig	设置应用对象。
interfaceId	-	设置服务接口（唯一标识元素）。 无论是普通调用还是返回调用，这里都设置实际的接口类。
uniqueId	-	设置服务标签（唯一标识元素）。
filterRef	-	过滤器配置实例。 List
filter	-	过滤器配置别名。 List，多个别名使用英文逗号（,）分隔。

属性	默认值	说明
registry	-	设置服务端注册中心。 List
methods	-	方法级配置，配置格式： `Map<String, MethodConfig>` 。
serialization	hessian2	设置序列化协议。
register	true	是否注册服务。 取决于实现，可能不生效。
subscribe	true	是否订阅服务。 取决于实现，可能不生效。
proxy	javassist	设置代理类型，支持 javassist 代理和 JDK 动态代理。
ref	-	服务接口实现类
server	-	服务端 List，可以一次发到多个服务端。
delay	0	设置服务延迟发布时间。 设置为 -1 代表 spring 加载完毕（通过 spring 才生效）。
weight	-	设置服务静态权重。
include	-	发布的方法列表
exclude	-	不发布的方法列表。
dynamic	true	是否为动态注册。 配置为 false 表示不主动发布，您需要到管理端进行上线操作。

属性	默认值	说明
priority	-	设置服务优先级。 数值越大，优先级越高。
bootstrap	bolt	设置服务发布启动器。
executor	-	设置自定义线程池。
timeout	-	设置服务端执行超时时间。 单位：毫秒 超时后不会打断执行线程，只是打印警告。
concurrents	-	设置接口下每个方法的最大可并行执行请求数，取值如下： • -1：关闭并发过滤器。 • 0：开启过滤，但不限制并发执行数量。
cacheRef	-	结果缓存实现类
mockRef	-	Mock 实现类
mock	-	是否开启 Mock。
validation	-	是否开启参数验证（基于 JSR303）。
compress	false	是否启动压缩。
cache	false	是否启用结果缓存。
parameters	-	额外属性，格式：`Map<String, String>`

ConsumerConfig

属性	默认值	说明
id	自动生成	ID
application	空 ApplicationConfig	设置应用对象。
interfaceId	-	设置服务接口（唯一标识元素）。 无论是普通调用还是返回调用，这里都设置实际的接口类。
uniqueId	-	设置服务标签（唯一标识元素）。
filterRef	-	设置过滤器配置实例。 List
filter	-	设置过滤器配置别名。 List，多个别名使用英文逗号（,）分隔。
registry	-	设置服务端注册中心。 List
methods	-	方法级配置，格式： `Map<String, MethodConfig>`
serialization	hessian2	设置序列化协议。
register	true	是否注册服务。 取决于实现，可能不生效。
subscribe	true	是否订阅服务。 取决于实现，可能不生效。
proxy	javassist	设置代理类型，支持 javassist 代理和 JDK 动态代理。
protocol	bolt	设置调用的协议，目前支持 bolt、rest、dubbo。

属性	默认值	说明
directUrl		设置直连地址，用于直连后 register。
generic	false	是否为泛化调用。
connectTimeout	3000(cover 5000)	设置建立连接超时时间。 单位：毫秒
disconnectTimeout	5000(cover 10000)	设置断开连接等待超时时间。 单位：毫秒
cluster	failover	设置集群模式。
connectionHolder	all	设置连接管理器实现。
loadBalancer	random	设置负载均衡算法。
lazy	false	是否延迟建立长连接。 第一次调用时创建。
sticky	false	是否使用粘性连接。 取值为 true 时，跳过负载均衡算法，使用上一个地址（一个连接断开后，才会选下一个地址）。
injVM	true	是否转为 JVM 调用。 配置为 true 时，由 JVM 发现服务提供者，使用本地配置。
check	false	设置是否检查强依赖。 配置为 true 时，若无可用服务端，会导致启动失败。
heartbeat	30000	设置客户端给服务端发送的心跳间隔。 取决于实现，可能不生效。
reconnect	10000	设置客户端重建端口长连接的间隔。 取决于实现，可能不生效。

属性	默认值	说明
router	-	设置路由器配置别名。 List
routerRef	-	设置路由器配置实例。 List
bootstrap	bolt	设置服务引用启动器。
addressWait	-1	设置等待地址获取时间。 取决于实现，可能不生效。
timeout	3000(cover 5000)	设置调用超时时间。 单位：毫秒
retries	0	设置失败后重试次数。 跟集群模式有关，failover 会读取此参数。
invokeType	sync	设置调用类型，取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
onReturn	-	接口下每个方法的最大可并行执行请求数，取值如下： • -1：关闭并发过滤器。 • 0：开启过滤，但不限制最大请求数。
cacheRef	-	结果缓存实现类。
mockRef	-	Mock 实现类。
cache	false	是否启用结果缓存。

属性	默认值	说明
mock	-	是否开启 Mock。
validation	-	是否开启参数验证（基于 JSR303）。
compress	false	是否启动压缩。
parameters	-	额外属性，格式：`Map<String, String>`

MethodConfig

属性	默认值	备注
name	-	设置方法名。 不能用于重载方法。
timeout	-	设置调用超时时间。 单位：毫秒
retries	-	设置失败后重试次数。
invokeType	-	设置调用类型，取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
validation	-	是否开启参数验证（基于JSR303）。
onReturn	-	返回时调用的 SofaResponseCallback，用于实现 Callback 等。

属性	默认值	备注
concurrent	-	接口下每个方法的最大可并行执行请求数，取值如下： • -1：关闭并发过滤器。 • 0：开启过滤，但不限制最大请求数。
validation	-	是否开启参数验证。
compress	-	是否启动压缩。
parameters	-	额外属性，格式：`Map<String, String>`。

ServerConfig

属性	默认值	备注
protocol	bolt	调用的协议，目前支持 bolt、rest、dubbo。
host	0.0.0.0	实际监听 IP，与网卡地址对应。
port	12200	协议的端口，默认端口如下： • bolt：12200 • rest：8341 • h2c：12300 • dubbo：20880
contextPath	-	设置上下文路径。
ioThreads	0	设置 IO 线程池数。 取决于实现，可能不生效。例如 Bolt 默认的默认线程数为 cpu 数量*2。0 表示自动计算。
threadPoolType	cached	设置业务线程池类型。
coreThreads	80(override 20)	设置业务线程池核心大小。

属性	默认值	备注
maxThreads	400(override 200)	设置业务线程池最大值。
telnet	true	是否允许 telnet。 取决于实现，可能不生效。例如 Bolt 不支持 telnet。
queueType	normal	设置业务线程池类型。 可用于实现优先级队列等。
queues	1000(override 0)	设置业务线程池队列。
aliveTime	300000(override 60000)	设置业务线程池存活时间。 单位：毫秒
preStartCore	-	是否初始化核心线程。
accepts	100000	设置最大长连接数量。 取决于实现，可能不生效。
serialization	hesisan2	设置序列化协议。
virtualHost	-	设置虚拟主机地址，注册到注册中心时优先使用该地址。
virtualPort	-	设置虚拟主机端口，注册到注册中心时优先使用该端口。
epoll	false	是否启动 epoll。 取决于实现，可能不生效。
daemon	true	是否守护端口，取值如下： • true：随主线程退出而退出。 • false：需要主动退出。
adaptivePort	false	是否调整端口。 设置为 true 时，若端口被占用，则会自动将端口号 +1 进行适应。

属性	默认值	备注
transport	bolt (cover netty4)	传输层实现 取决于实现，可能不生效。
autoStart	true	是否自动启动端口。
stopTimeout	10000(override 20000)	设置优雅关闭超时时间。 单位：毫秒
boundHost	-	绑定的地址，默认取 host 值。
parameters	-	额外属性，格式：`Map<String, String>`。

RegistryConfig

属性	默认值	备注
protocol	zookeeper	服务协议，目前支持 zookeeper 和 local。
address	-	指定注册中心地址。 address 和 index 属性必须配置一个。
index	-	指定注册中心寻址服务的地址。 address 和 index 属性必须配置一个。
register	true	是否注册服务。
subscribe	true	是否订阅服务。
timeout	10	调用注册中心超时时间 单位：秒
connectTimeout	20	连接注册中心超时时间 单位：秒

属性	默认值	备注
file	$HOME	local 协议时使用的本地文件位置，默认在 `$HOME` 下。

3.12.5. 自定义 Registry

> **说明**
>
> 自定义注册中心的实现类可参考 `com.alipay.sofa.rpc.registry.zk.ZookeeperRegistry`，实现的具体信息请参见 Registry-ZK。

自定义注册中心的流程如下：

实现自定义注册中心类

1. 自定义注册中心类。

   自定义注册中心类时，需继承如下虚拟类：

   ```
   package com.alipay.sofa.rpc.registry;

   @Extensible(singleton =false)
   public abstract class Registry implements Initializable,Destroyable{
       public abstract boolean start();  //启动客户端
       public abstract void register(ProviderConfig config);  //注册服务
       public abstract void unRegister(ProviderConfig config);  //取消注册，优雅关闭使用
       public abstract void batchUnRegister(List<ProviderConfig> configs); //批量取消注册
       public abstract List<ProviderGroup> subscribe(ConsumerConfig config); //订阅服务
       public abstract void unSubscribe(ConsumerConfig config); //取消订阅，优雅关闭使用
       public abstract void batchUnSubscribe(List<ConsumerConfig> configs); //批量取消订阅
   }
   ```

2. 添加注解。

   示例如下：

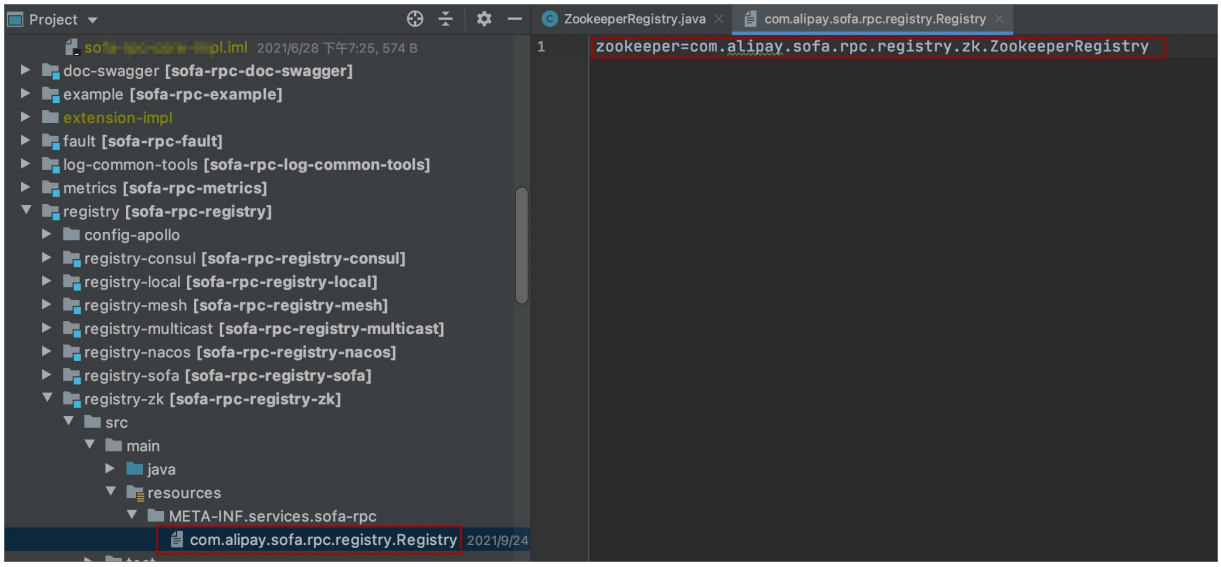
   ```
   @Extension("zookeeper")
   public class ZookeeperRegistry extends Registry {
   ```

   `@Extension("")` 用于指定注册中心别名，可自定义。您可以在业务指定注册中心时使用该别名。

将自定义注册中心配置到 RPC 架构

基于 SOFARPC 的 SPI 机制，您需要新建扩展文件 `META-INF/services/sofa-rpc/com.alipay.sofa.rpc.registry.Registry`，将自定义的注册中心配置到 RPC 架构。内容示例如下：

```
zookeeper=com.alipay.sofa.rpc.registry.zk.ZookeeperRegistry
```



在业务中使用自定义注册中心

1. 在 `application.properties` 中配置注册中心源信息。

   配置格式为：`com.alipay.sofa.rpc.registries.<注册中心用例名称>=协议://地址`，协议名称为上面自定义的注册中心名称，例如 `zookeeper`。示例如下：

   ```
   com.alipay.sofa.rpc.registries.myRegistry=zookeeper://10.0.0.1:9600  //配置注册中心地址。
   ```

2. 配置需要注册中心的服务。

   注册中心名称为上面自定义的注册中心用例名。示例如下：

   ```
   <sofa:service interface="com.alipay.sofa.facade.SampleService" ref="sampleService">
           <sofa:binding.bolt>
               <sofa:global-attrs registry="myRegistry"/>
           </sofa:binding.bolt>

       </sofa:service>
   ```

3.12.6. 自定义 Filter

SOFARPC 提供了一套良好的可扩展性机制，为各个模块提供 SPI（Service Provider Interface） 的能力。

SOFARPC 对请求与响应的过滤链（FilterChain）处理方式为：

- 通过多个过滤器 Filter 来进行具体的拦截处理。
- 允许用户自定义 Filter 扩展，且自定义 Filter 的执行顺序在内置 Filter 之后。

下文将就自定义 Filter 类及其生效进行说明：

自定义 Filter 类

自定义 Filter 类需继承 `com.alipay.sofa.rpc.filter.Filter` 类。示例如下：

▶ Details

自定义 Filter 类生效方式

自定义 Filter 类支持如下生效方式：

- API 方式

  该方式可以指定生效的 provider 或 consumer 。示例如下：

  ▶ Details

- @Extension 注解 + 扩展文件 + 编码注入

  处理步骤如下：

  i. 处理自定义 Filter 类。

  ▶ Details

  ii. 创建扩展文件。

  - 文件名：以 Filter 为后缀。例如：`com.alipay.sofa.rpc.filter.Filter` 。
  - 文件路径：为固定路径。位于 `META-INF/services/sofa-rpc/` 下。

  完整示例：`META-INF/services/sofa-rpc/com.alipay.sofa.rpc.filter.Filter` 。扩展文件内容如下：

  ```
  customer=com.alipay.sofa.rpc.custom.CustomFilter
  ```

  iii. 注入编码。

  ▶ Details

- @Extension 注解 + 扩展文件 + @AutoActive 注解

  处理步骤如下：

  i. 处理自定义 Filter 类。

  ▶ Details

  > **说明**
  >
  > 使用 2 个注解 `@Extension` 和 `@AutoActive` 处理 Filter 类可以免去编码注入。作用域为所有 provider 或 consumer。其中，参数 `providerSide` 表示是否生效于服务端；参数 `consumerSide` 表示是否生效于客户端。

  ii. 创建扩展文件。

  - 文件名：以 Filter 为后缀。例如 `com.alipay.sofa.rpc.filter.Filter` 。
  - 文件路径：固定路径，位于 `META-INF/services/sofa-rpc/` 下。

完整示例：`META-INF/services/sofa-rpc/com.alipay.sofa.rpc.filter.Filter`。扩展文件内容如下：

```
customer=com.alipay.sofa.rpc.custom.CustomFilter
```

3.12.7. 自定义 Router

SOFARPC 中对服务地址的选择也抽象为了一条处理链，由每一个 Router 进行处理。同 Filter 一样，SOFARPC 对 Router 提供了同样的扩展能力。本文介绍如何自定义 Router 类。

自定义 Router 类的流程如下：

实现自定义 Router 类

自定义 Router 类时，需继承 `com.alipay.sofa.rpc.client.Router` 类：

```
@Extension(value ="customerRouter")
@AutoActive(consumerSide =true)
public class CustomerRouter extends Router {

    @Override
    public void init(ConsumerBootstrap consumerBootstrap) {

    }
    @Override
    public boolean needToLoad(ConsumerBootstrap consumerBootstrap) {
        return true;
    }
    @Override
    public List<ProviderInfo> route(SofaRequest request, List<ProviderInfo> providerInfos)
{
        return providerInfos;
    }
}
```

RPC 框架支持通过 `@Extension` 注解来扩展 Router，并且可以覆盖重名的 Router。如需覆盖原有 Router，需将 `@Extension` 中 `override` 置为 `true`，并确保 `order` 属性值大于要覆盖的重名 Router。未配置 `order` 属性时，默认为 0。

如上自定义了一个 `CustomerRouter`，生效于所有消费者。其中：

- `init` 参数 `ConsumerBootstrap` 是引用服务的包装类，能够拿到 `ConsumerConfig`、代理类、服务地址池等对象。
- `needToLoad` 表示是否生效该 Router。`route` 方法即筛选地址的方法。

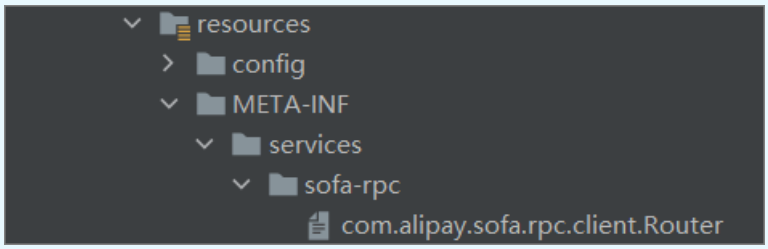
将自定义 Router 类配置到 RPC 架构

基于 SOFARPC 的 SPI 机制，您需要在类路径中新建扩展文件 `META-INF/services/sofa-rpc/com.alipay.sofa.rpc.client.Router`，将自定义的 Router 类配置到 RPC 架构。内容示例如下：

```
customerRouter=com.alipay.sofa.rpc.custom.CustomRouter
```

重要

`resource` 内的目录和文件名需要遵守如下固定规则：



3.12.8. 自定义 ShutdownHook

优雅关闭涉及两方面，一个是 RPC 框架作为客户端，一个是 RPC 框架作为服务端。

作为服务端

RPC 框架作为服务端，在关闭时，不应该直接暴力关闭。关闭方式如下：

1. 查看 RPC 框架：

   ```
   com.alipay.sofa.rpc.context.RpcRuntimeContext
   ```

2. 在静态初始化块中，添加一个 `ShutdownHook` 。

   ```
       // 增加 JVM 关闭事件。
       if(RpcConfigs.getOrDefaultValue(RpcOptions.JVM_SHUTDOWN_HOOK,true)){
           Runtime.getRuntime().addShutdownHook(newThread(newRunnable(){
               @Override
               public void run(){
                   if(LOGGER.isWarnEnabled()){
                       LOGGER.warn("SOFA RPC Framework catch JVM shutdown event, Run shutd
   own hook now.");
                   }
                   destroy(false);
               }
           },"SOFA-RPC-ShutdownHook"));
       }
   ```

   `ShutdownHook` 的作用是当发布平台或用户执行 `kill pid` 时，会先执行 `ShutdownHook` 中的逻辑。在销毁操作中，RPC 框架会先执行向注册中心取消服务注册、关闭服务端口等动作。示例如下：

```
private static void destroy(boolean active){
        RpcRunningState.setShuttingDown(true);
        for(Destroyable.DestroyHook destroyHook : DESTROY_HOOKS){
            destroyHook.preDestroy();
        }
        List<ProviderConfig> providerConfigs = new ArrayList<ProviderConfig>();
        for(ProviderBootstrap bootstrap : EXPORTED_PROVIDER_CONFIGS){
            providerConfigs.add(bootstrap.getProviderConfig());
        }
        // 先反注册服务端。
        List<Registry> registries =RegistryFactory.getRegistries();
        if(CommonUtils.isNotEmpty(registries) && CommonUtils.isNotEmpty(providerConfigs
)){
            for(Registry registry : registries){
                registry.batchUnRegister(providerConfigs);
            }
        }
        // 关闭启动的端口。
        ServerFactory.destroyAll();
        // 关闭发布的服务。
        for(ProviderBootstrap bootstrap : EXPORTED_PROVIDER_CONFIGS){
            bootstrap.unExport();
        }
        // 关闭调用的服务。
        for(ConsumerBootstrap bootstrap : REFERRED_CONSUMER_CONFIGS){
            ConsumerConfig config = bootstrap.getConsumerConfig();
                if(!CommonUtils.isFalse(config.getParameter(RpcConstants.HIDDEN_KEY_DES
TROY))){
                // 除非不让主动 unrefer。
                bootstrap.unRefer();
            }
        }
        // 关闭注册中心。
        RegistryFactory.destroyAll();
        // 关闭客户端的一些公共资源。
        ClientTransportFactory.closeAll();
        // 卸载模块。
        if(!RpcRunningState.isUnitTestMode()){
            ModuleFactory.uninstallModules();
        }
        // 卸载钩子。
        for(Destroyable.DestroyHook destroyHook : DESTROY_HOOKS){
            destroyHook.postDestroy();
        }
        // 清理缓存。
        RpcCacheManager.clearAll();
        RpcRunningState.setShuttingDown(false);
        if(LOGGER.isWarnEnabled()){
            LOGGER.warn("SOFA RPC Framework has been release all resources {}...",
                    active ?"actively ":"");
        }
 }
```

其中以 Bolt 为例，关闭端口并不是一个立刻执行的动作，而是会判断当前服务端上面的连接和队列的任务，先处理完队列中的任务再缓慢关闭。

```
    @Override
    public void destroy(){
        if(!started){
            return;
        }
        int stopTimeout = serverConfig.getStopTimeout();
        if(stopTimeout >0){// 需要等待结束时间
            AtomicInteger count = boltServerProcessor.processingCount;
            // 有正在执行的请求 或者 队列里有请求
            if(count.get()>0|| bizThreadPool.getQueue().size()>0){
                long start =RpcRuntimeContext.now();
                if(LOGGER.isInfoEnabled()){
                    LOGGER.info("There are {} call in processing and {} call in queue,
wait {} ms to end",
                                count, bizThreadPool.getQueue().size(), stopTimeout);
                }
                while((count.get()>0|| bizThreadPool.getQueue().size()>0)
                        && RpcRuntimeContext.now()- start < stopTimeout){
                    // 等待返回结果
                    try{
                        Thread.sleep(10);
                    }catch(InterruptedException ignore){
                    }
                }
            }
            // 关闭前检查已有请求?
        }
        // 关闭线程池
        bizThreadPool.shutdown();
        stop();
    }
```

作为客户端

RPC 框架作为客户端时，实际上就是 Cluster 的关闭。关闭调用的服务这一步，可以查看 `com.alipay.sofa.rpc.client.AbstractCluster` 。

```
   /**
     * 优雅关闭的钩子。
     */
    protected class GracefulDestroyHook implements DestroyHook{
        @Override
        public void preDestroy(){
            // 准备关闭连接。
            int count = countOfInvoke.get();
            final int timeout = consumerConfig.getDisconnectTimeout();// 等待结果超时时间。
            if(count >0){// 有正在调用的请求。
                long start =RpcRuntimeContext.now();
                if(LOGGER.isWarnEnabled()){
                    LOGGER.warn("There are {} outstanding call in client, will close transp
orts util return",
                                count);
                }
                while(countOfInvoke.get()>0 && RpcRuntimeContext.now()- start < timeout){//
等待返回结果。
                    try{
                        Thread.sleep(10);
                    }catch(InterruptedException ignore){
                    }
                }
            }
        }

        @Override
        public void postDestroy(){
        }
    }
```

客户端会逐步将正在调用的请求处理完成才会下线。

> **说明**
>
> 优雅关闭是需要和发布平台联动的。如果强制 kill，那么任何优雅关闭的方案都不会生效。后续会考虑在 SOFABoot 层面提供一个统一的 API，来给发布平台调用，而不是依赖 hook 的逻辑。

3.12.9. 单元测试与性能测试

本文主要介绍在进行 SOFARPC 开发时如何进行单元测试与性能测试。

单元测试

将单元测试的示例放到您自己开发的模块下。如果依赖了第三方服务端（例如 Zookeeper），请手动加入 profile，并放到 `test-intergrated-3rd` 模块中。参考 `registry-zookeeper` 模块代码。如果依赖了其它模块要集成测试，请放到 `test/test-intergrated` 模块中。

性能测试

1. 在启动参数中修改以下参数，以关闭相关项目。

```
-Dcontext.attachment.enable=false
-Dserialize.blacklist.enable=false
-Ddefault.tracer= false
-Dlogger.impl=com.alipay.sofa.rpc.log.SLF4JLoggerImpl
-Dmultiple.classloader.enable=false
-Devent.bus.enable=false
```

2. 对 `bolt+hessian` 进行压测。

   压测环境如下：

   - 服务端：4C8G 虚拟机、千兆网络、JDK1.8.0_111。
   - 客户端：50 个客户端并发请求。

   压测结果：

协议	请求	响应	服务端	TPS	平均 RT(ms)
bolt+hessian	1000 String	1000 String	直接返回	10000	1.93
bolt+hessian	1000 String	1000 String	直接返回	20000	4.13
bolt+hessian	1000 String	1000 String	直接返回	30000	7.32
bolt+hessian	1000 String	1000 String	直接返回	40000	15.78
bolt+hessian	1000 String	1000 String	直接返回	50000（接近极限，错误率 0.3%）	26.51

3.12.10. 如何编译 SOFARPC 工程（暂时下线）

本文介绍如何编译 SOFARPC 工程。

前提条件

- 安装 JDK7 或以上版本。
- 已安装 Maven 3.2.5 或以上版本。

操作说明

在 GitHub 中下载代码，然后执行如下命令：

```
cd sofa-rpc
mvn clean install
```

不能在子目录（即子模块）下进行编译。因为 SOFARPC 模块太多，如果每个子模块都进行安装和部署，仓库内会有较多无用记录。所以在设计 SOFARPC 工程结构的时候，各个子模块组件是不需要安装和部署到仓库里的，只会安装和部署一个 `sofa-rpc-all` (all) 模块。

3.13. 日志说明

当您使用 SOFARPC 启动应用程序以后，默认情况下，RPC 会创建 logs 目录，并生成相关日志文件。

日志列表

日志名称	说明
rpc/rpc-registry.log	服务地址订阅与接收日志。
rpc/tr-threadpool	服务连接池日志（SOFABoot 支持该日志）。
rpc/rpc-default.log	SOFARPC INFO、WARN 日志，无标准格式。
rpc/common-error.log	SOFARPC 错误日志，无标准格式。
rpc/rpc-remoting.log	网络层日志，无标准格式。
rpc/sofa-router.log	SOFARouter 相关日志，无标准格式。
rpc/rpc-remoting-serialization.log	网络层序列化日志，无标准格式。
tracelog/rpc-client-digest.log	SOFARPC 调用客户端摘要日志。
tracelog/rpc-server-digest.log	SOFARPC 调用服务端摘要日志。
tracelog/rpc-profile.log	SOFARPC 处理性能日志。
confreg/config.client.log	服务注册中心客户端日志。

SOFATracer 日志

SOFATracer 集成在 SOFARPC（5.4.0 及之后的版本）后，还会输出如下链路数据日志。

> 说明
>
> - 开源版的日志默认为 JSON 格式，企业版默认以逗号分隔。
> - 日志会不定期新增部分字段，新增字段会从日志尾部添加，不会影响原日志字段。若您实际打印的日志与本文中日志字段数不一致，请按顺序进行对比，新增字段可咨询售后技术支持。

RPC 客户端摘要日志

`rpc-client-digest.log` 是 RPC 客户端摘要日志，日志样例如下：

```
2021-09-27 16:43:59.096,myserver-app,1ecee1741632732******410896596,0,com.alipay.samples.rpc.SampleService:1.0,hello,bolt,,127.0.0.1,myclient-app,,,1ms,0ms,SOFA-SEV-BOLT-BIZ-12201-9-T20,00,,,0ms,,
```

对应 key 的说明如下：

key	说明
timestamp	日志打印时间
local.app	客户端应用名称
tracerId	TraceId
spanId	SpanId
service	服务接口信息
method	调用方法
span.kind	Span 类型
protocol	协议类型。取值：bolt、rest。

key	说明
invoke.type	调用类型。取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
remote.ip	目标 IP
remote.app	服务端应用名称
remote.zone	目标 zone
remote.idc	目标 IDC
remote.city	目标城市
user.id	用户 ID
result.code	结果码。取值如下： • 00：请求成功。 • 01：业务异常。 • 02：RPC 逻辑错误。 • 03：请求超时。 • 04：路由失败。
req.size	请求数据大小
resp.size	响应数据大小
client.elapse.time	调用总耗时，单位：ms。
client.conn.time	客户端连接耗时，单位：ms。

key	说明
req.serialize.time	请求序列化耗时，单位：ms。
outtime	超时参考耗时，单位：ms。
current.thread.name	当前线程名
route.record	路由记录，路由选择的过程记录。
elastic.id	弹性数据位
be.elastic	本次调用的服务是否需要弹性： • T：表示需要弹性。 • F：表示不需要。
elastic.service.name	转发调用的方法名。
local.client.ip	源 IP
local.client.port	本地客户端端口
local.zone	本地 zone
target.ip.in.one.physical	目标 IP 是否在当前物理机： • T：表示在同一物理机。 • F：表示不在同一物理机。
sys.baggage	系统透传的 baggage 数据
bus.baggage	业务透传的 baggage 数据
send.time	RPC 请求耗时
phase.time	各阶段耗时，单位：ms。

key	说明
special.time.mark	特殊时间点标记
router.forward	路由转发详情

RPC 服务端摘要日志

`rpc-server-digest.log` 是 RPC 服务端摘要日志，日志样例如下：

```
2014-06-19 17:14:35.006,client,0ad1348f140****2750021003,0.1,com.alipay.cloudenginetest.services.SofaApiWebReferenceLocalFalseTrService:1.0,service_method,TR,,10.**.**.143,client,,,12ms,0ms,HSFBizProcessor-4-thread-2,00,,F,1000ms,zue:l;ztg:r,mark=T&uid=a2&
```

对应 key 的说明如下：

key	说明
timestamp	日志打印时间
local.app	客户端应用名称
traceId	TraceId
spanId	SpanId
service	服务接口信息
method	调用方法
protocol	协议类型。取值：bolt、rest。
remote.ip	目标 IP
remote.app	服务端应用名称

key	说明
invoke.type	调用类型。取值如下： • sync：同步调用，Bolt 默认的调用方式。 • oneway：异步调用，消费方发送请求后直接返回，忽略提供方的处理结果。 • callback：异步调用，消费方提供一个回调接口，当提供方返回后，SOFA 框架会执行回调接口。 • future：异步调用，消费方发起调用后马上返回，当需要结果时，消费方需要主动去获取数据。
remote.ip	调用方 IP
remote.app	调用方应用名
remote.zone	调用方 zone
remote.idc	调用方 IDC
biz.impl.time	请求处理耗时，单位：ms。 不包含服务端响应序列化耗时和反序列化耗时。
resp.serialize.time	响应序列化耗时，单位：ms。
current.thread.name	当前线程名
result.code	返回码，取值如下： • 00：请求成功。 • 01：业务异常。 • 02：RPC 逻辑错误。
elastic.service.name	表明当前是转发调用，包含转发的服务名称和方法值。
be.elasticc	本次服务是否被转发： • T：表示已被转发。 • F：表示未被转发。

key	说明
server.pool.wait.time	服务端线程池等待时间，单位：ms。
sys.baggage	系统透传的 baggage 数据
bus.baggage	业务透传的 baggage 数据
server.send.time	RPC 请求转发耗时
req.size	请求数据大小
resp.size	响应数据大小
phase.time	各阶段耗时明细
special.time	特殊时间点标记
router.forward	路由转发详情

RPC 客户端统计日志

`rpc-client-stat.log` 是 RPC 客户端统计日志。其中，`stat.key` 即本段时间内的统计关键字集合。统计关键字集合唯一确定一组统计数据，包含 `local.app` 、`remote.app` 、`service.name` 和 `method.name` 字段。日志样例如下：

```
2014-06-19 17:14:02.186,client,client,com.alipay.cloudenginetest.services.SofaApiWebReferenceLocalFalseTrService:1.0,service_method,1,79,Y,T,RZ00B
```

对应 key 的说明如下：

key		说明
time		日志打印时间
	local.app	客户端应用名称

key		说明
stat.key	remote.app	服务端应用名称
	service.name	服务名
	method.name	方法名
count		调用次数
total.cost.milliseconds		请求总耗时
success		调用结果： • Y：调用成功。 • N：调用失败。
load.test.mark		判断当前是否为全链路压测： • T：表示当前为全链路压测。当前线程中能获获取到日志上下文，且上下文中有压测信息。 • F：表示当前非全链路压测。当前线程中不能获取到日志上下文，或上下文中没有压测信息。
remote.zone		目标 zone

RPC 服务端统计日志

`rpc-server-stat.log` 是 RPC 服务端统计日志，以 JSON 格式输出的数据。其中，`stat.key` 即本段时间内的统计关键字集合。统计关键字集合唯一确定一组统计数据，包含 `local.app`、`remote.app`、`service.name` 和 `method.name` 字段。日志样例如下：

```
2014-06-19 17:14:02.186,client,client,com.alipay.cloudenginetest.services.SofaApiWebReferenceLocalFalseTrService:1.0,service_method,1,7,Y,T,GZ00B
```

对应 key 的说明如下：

key		说明
time		日志打印时间
stat.key	local.app	客户端应用名称
	remote.app	服务端应用名称
	service.name	服务名
	method	方法名
count		调用次数
total.cost.milliseconds		请求总耗时
success		调用结果： • Y：调用成功。 • N：调用失败。
load.test.mark		判断当前是否为全链路压测： • T：表示当前为全链路压测。当前线程中能获获取到日志上下文，且上下文中有压测信息。 • F：表示当前非全链路压测。当前线程中不能获取到日志上下文，或上下文中没有压测信息。
remote.zone		目标 zone

4.服务治理

4.1. 查看服务

您可以通过服务管控功能查看和管理所有可用的微服务应用。

查看服务

1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 服务目录**。

   您可以在服务列表中查看所有已发布的微服务应用，包括服务 ID、提供服务的应用、服务提供者数量、服务消费者数量等。
3. 单击目标服务 ID 查看服务详情。

   您可以查看服务的基本信息、服务提供者和服务消费者列表：
   - 基本信息：包含服务 ID、类型、应用等信息。
   - 服务提供者：包含服务提供者的 IP、端口、应用名、权重、禁用和启用状态信息及恢复默认按钮。
   - 服务消费者：包含服务消费者的 IP 与应用名。

管理服务

您可以通过以下方式管理您的服务：

- 进行服务治理

  您可以在服务列表中，通过操作列对应用进行服务治理，例如服务限流、服务路由等。具体操作，请参见服务治理。
- 管理应用

  您可以在服务提供者列表，选择目标应用后，进行包括启用、禁用、修改权重、恢复默认在内的操作。
  - 启用：启用已禁用的服务提供者。
  - 禁用：禁用一个或多个服务提供者。
  - 修改权重：修改服务提供者的权重，取值为 0~100。
  - 恢复默认：将服务提供者恢复为默认状态（服务状态为启用、权重为 100）。

4.2. 应用依赖

您可以在应用依赖页面查询当前工作环境中的应用依赖关系。

操作步骤

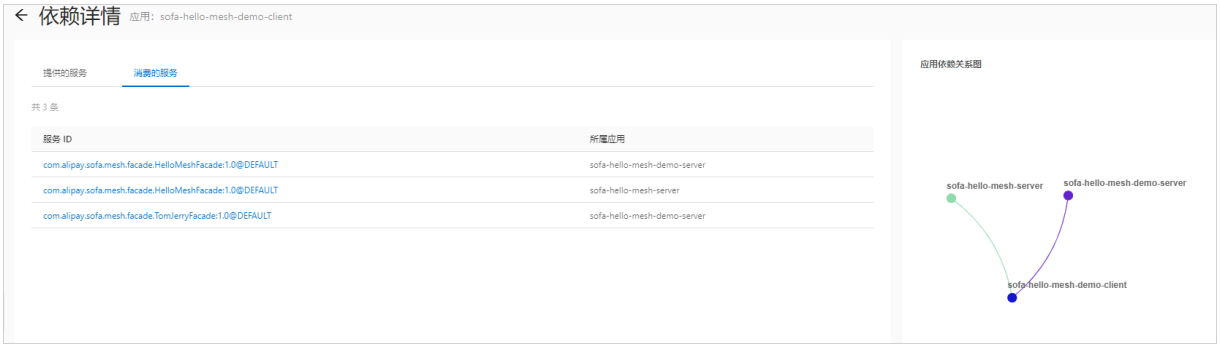
1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 服务目录**，然后单击 **应用依赖** 页签。

   在 **应用依赖** 页面，您可以执行以下操作：
   - 查看有依赖关系的所有应用，包括服务提供者和消费者。
   - 将鼠标悬浮在应用图标上，可以查看所选应用的依赖关系、节点数、提供的服务数、消费的服务数。
   - 单击应用图标可查看提供的服务和消费的服务详情。

     您可以在服务列表单击目标服务 ID，查看服务的详细信息。

依赖示例

消费的服务 页签列出了消费的服务 ID 及所属应用。示例如下：



示例说明：

- 应用名：`sofa-hello-mesh-demo-client`
- 依赖关系图：消费的服务共 3 个，来自于 2 个应用。
  - 2 个服务来自于 `sofa-hello-mesh-demo-server` 。
  - 1 个服务来自于 `sofa-hello-mesh-server` 。

4.3. 服务限流

4.3.1. 快速入门

本文从本地工程开发到应用的云端发布，再到配置限流，介绍服务限流的整理过程。

操作步骤

1. 本地工程开发。

   操作步骤请参见 本地实现 SOFARPC 服务。
2. 引入依赖和本地配置。
   - 引入依赖

在 SOFABoot Web 工程 `endpoint` 模块下的 `pom.xml` 文件中，引入 DRM 和 guardian 依赖，示例如下：

```
<dependency>
     <groupId>com.alipay.sofa</groupId>
     <artifactId>ddcs-enterprise-sofa-boot-starter</artifactId>
</dependency>

<dependency>
     <groupId>com.alipay.sofa</groupId>
     <artifactId>guardian-sofa-boot-starter</artifactId>
</dependency>
```

使用 `rest-enterprise-sofa-boot-starter` 和 `rpc-enterprise-sofa-boot-starter` 时，都需要引入上述依赖。引入 `rpc-enterprise-sofa-boot-starter` 时，如果需要使用限流功能，还需要引入下述依赖，否则限流不起作用。

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-autoconfigure</artifactId>
</dependency>
<dependency>
     <groupId>com.alipay.sofa</groupId>
     <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
</dependency>
```

定义接口时还需要自定义一个异常和一个异常处理器，用于限流成功后抛出异常，并且把异常处理结果返回给前端。

- 本地配置

  当您引入 `guardian-sofa-boot-starter` 依赖时，应用已经可以对 SOFARPC 接口和 Spring MVC 请求进行限流。如果您还需要对内部 Spring Bean 定义的方法进行限流，则需要在 Spring Bean 配置文件中添加配置 AOP 拦截器。示例如下：

```
<!-- 引入 guardian 中定义的 bean。 -->
<import resource="classpath:META-INF/spring/guardian-sofalite.xml"/>
<!-- 配置 AOP 拦截器。 -->
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="interceptorNames">
        <list>
            <value>guardianExtendInterceptor</value>
        </list>
    </property>
    <property name="beanNames">
        <list>
            <!-- 配置需要被拦截的 bean。 -->
            <value>*DAO</value>
        </list>
    </property>
    <!-- 如要使用 CGLIB 代理，取消下面这行的注释。 -->
    <!-- <property name="optimize" value="true" /> -->
</bean>
```

3. 云端发布。

   请参考下述信息，完成应用的云端发布：

   - 应用整体的发布流程，请参见 技术栈与应用发布流程。
   - 应用的详细发布步骤，请参见 经典应用服务快速入门。

4. 在 SOFAStack 控制台配置限流规则。

   配置步骤，请参见 控制台配置限流规则。

4.3.2. 管理限流规则

4.3.2.1. 添加限流规则

SOFAStack 微服务主要通过 SOFARPC 来实现服务的发布和引用，在 SOFAStack 微服务中的限流主要是针对 SOFARPC 框架。微服务的服务限流（Guardian）是一个限流组件，您可通过在业务系统中集成该组件，配置限流规则来提供限流服务，从而保证业务系统不会被大量突发请求击垮，提高系统稳定性。

前提条件

服务限流的规则配置依赖于动态配置推送，所以接入限流前必须先接入动态配置。详情请参见 新增动态配置。

操作步骤

1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 服务治理**。
3. 单击 **服务限流**，然后单击 **新增应用**。
4. 输入应用名称后，单击 **确定**。
5. 在应用列表单击目标应用右侧的 **添加规则**，然后配置规则参数。

区域	参数	说明
基本信息	规则名称	配置限流规则的名称，用于描述规则。
	限流类型	配置限流规则的方法，可选值： ○ **接口方法**：支持对某个具体的 RPC 接口或普通 Bean 的方法限流。 您需要在 **限流对象** 中配置接口路径名称和方法签名。 ○ **Web 页面**：对基于 Spring MVC 的 Web 请求进行限流。 您需要在 **限流对象** 中配置请求 URI。
	运行模式	配置限流规则的运行模式，可选值： ○ **拦截模式**：限流生效的模式，会根据配置的规则实际拦截请求。 ○ **监控模式**：仅打印限流记录日志，不实际产生限流效果。
	限流算法	配置限流规则的算法，可选值： ○ **QPS 计数算法**：通过限制单位时间段内允许的请求调用量进行限流。 ○ **令牌桶（Token Bucket）算法**：控制发送到网络上的数据的数目，并允许突发数据的发送。 有关算法的详细说明参见 限流算法说明。

区域	参数	说明

限流后置操作	限流后操作	配置限流后进行的后续操作，可选值：
		○ **空配置**：限流后不做任何处理，返回空值。
		○ **抛出异常**：限流后返回异常信息。异常信息为填写的输入框内容。
		仅当 **限流类型** 为 **接口方法** 时，支持配置。
		○ **跳转到指定页面**：限流后跳转到指定的页面地址。
		仅当 **限流类型** 为 **Web 页面** 时，支持配置。
		○ **页面 JSON 报文**：限流后直接将指定的 JSON 字符串在 HTML Response 中返回。
		默认返回内容为：
		`{success:false,error:"MAX_VISIT_LIMIT"}`
		仅当 **限流类型** 为 **Web 页面** 时，支持配置。
		○ **页面 XML 报文**：限流后直接将特定的 XML 字符串在 HTML Response 中返回。
		默认返回内容为：
		`<?xml version="1.0" encoding="GBK"?><alipay><is_success>F</is_success><error>MAX_VISIT_LIMIT</error></alipay>`
		仅当 **限流类型** 为 **Web 页面** 时，支持配置。

区域	参数	说明
限流阈值	限流阈值	配置限流阈值，配置项如下： ○ 条件模型：配置限速的条件。 可选值如下： ■ **单位时间内服务访问次数或 Web 页面访问次数**：根据单位时间内的请求数进行限流。 ■ **堆内存使用量**：根据当前堆内存使用量进行限流。 ■ **CPU 负载**：根据过去一分钟内的 CPU 平均负载进行限流。 ■ **并发线程数**：根据单台机器上并发的线程数进行限流。 ○ **单位时间**：打印限流日志的周期。对于单位时间内访问次数的限流条件，也表示统计周期。 单位为毫秒（ms）。最小值为 1000 ms。 ○ **限流阈值**：根据选择的 **条件模型** 配置限流阈值，超过阈值的流量会被限速。 ■ 选择 **单位时间内服务访问次数或 Web 页面访问次数** 时：配置最大允许的 QPS 数。 ■ 选择 **堆内存使用量** 时：配置最大堆内存使用量，单位为 MB。 ■ 选择 **CPU 负载** 时：配置 CPU 的最大负载，数值为 `100 * CPU 负载百分比` 。 ■ 选择 **并发线程数** 时：配置最大并发线程数。 ○ **流量类型**：配置限流规则针对的流量类型。 可选值如下： ■ **所有流量**：对正常流量和压测流量均限流。 ■ **正常流量**：仅对正常流量限流。 ■ **压测流量**：仅对压测流量限流。

区域	参数	说明
	限流对象	根据选择的 **限流类型** 配置限流的对象名。 ○ 选择 **接口方法** 时，配置以下参数： ■ 接口：配置限流的接口名称。支持 RPC 服务接口或配置了 Spring AOP 拦截器的 Bean。 ■ 方法：配置需要限流的方法。支持带参数或不带参数的方法签名。 ■ 参数关系：配置多个参数条件之间的逻辑关系。可选值为 **AND**（与）和 **OR**（或）。 ○ 选择 **Web 页面** 时，配置以下参数： ■ URI：需要限流的请求 URI，不包含域名和参数部分。 ■ 参数关系：配置多个参数条件之间的逻辑关系。可选值为 **AND**（与）和 **OR**（或）。 限流对象名配置完成后，您还需要添加参数条件。更多信息，请参见配置限流对象方法签名。

6. 配置完成后，根据需求选择推送方式：
   - **提交**：限速规则向所有机器推送限速规则。
   - **灰度推送**：单击后选择推送目标，限速规则仅推送给指定机器。

   > 重要
   > - 灰度推送的数据不会保存到数据库，只会保存在被推送到的服务器的内存中。
   > - 服务器重启后推送的规则被还原，仍使用灰度推送前的规则。

7. 在应用列表单击目标应用左侧的加号，然后将目标规则的状态修改为 **开启**。

   限速规则开启后，将根据推送配置将限速规则推送给所有机器或指定机器。您也可以根据需要让限速规则在指定机器上开启：
   - 限速规则状态为 **关闭** 时：单击 **灰度开启**，然后选择开启限速规则的机器。
   - 限速规则状态为 **开启** 时：单击 **灰度关闭**，然后选择关闭限速规则的机器。

4.3.2.2. 导入导出限流规则

若要将同一规则作用于多个应用，您可以通过导入、导出限流规则，进行规则迁移。

导出限流规则

1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 服务治理**。
3. 单击 **服务限流**，然后在应用列表选择目标应用右侧 **更多 > 导出**。

导出的文件为 JSON 格式，存放在浏览器默认的下载文件夹中。

导入限流规则

您可以将导出的限流规则导入到其他应用中，以快速生成限速规则。

1. 在 **服务限流** 页面的应用列表，选择目标应用右侧 **更多 > 导入**。
2. 单击 **浏览**，选择目标文件后，单击 **打开**。

限速规则文件格式

导出的限速规则文件为 JSON 格式，文件内容如下：

```
[
    {
        "actionConfig":{
            "actionType":"LIMIT_EXCEPTION",
            "responseContent":"sssssssss"
        },
        "calculationConfigs":[
            {
                "calculationType":"INVOKE_BY_TIME",
                "maxAllow":1,
                "period":1000
            }
        ],
        "desc":"GuardianApp.query",
        "enable":false,
        "globalLimit":false,
        "limitStrategy":"QpsLimiter",
        "limitType":"GENERIC_LIMIT",
        "maxBurstRatio":0,
        "resourceConfigs":[
            {
                "baseName":"com.alipay.antcloud.dsrconsole.core.service.guardian.facade.Gua
rdianAppFacade.query",
                "resourceType":"METHOD",
                "ruleIds":[

                ]
            },
            {
                "baseName":"11.22",
                "resourceType":"METHOD",
                "ruleIds":[

                ]
            }
        ],
        "resourceType":"METHOD",
        "runMode":"CONTROL",
        "trafficType":"ALL"
    }
]
```

参数说明如下：

参数	说明

参数	说明
actionConfig	后置处理动作，包括如下参数： • actionType：后置动作类型。 • responseContent：如果后置动作类型为限流异常，则此字段表示异常信息。
calculationConfigs	限流配置，包括如下参数： • calculationType：限流计算类型。 • maxAllow：限流阈值。 • period：限流计算周期。
desc	限流规则描述。
enable	是否开启限流规则，导出规则均默认为不开启。
limitStrategy	限流算法类型。
maxBurstRatio	令牌桶算法的存量桶系数。
resourceConfigs	限流对象，包含如下参数： • baseName：限流对象名，如接口名+方法名、Web 请求的 URI。 • resourceType：目标对象类型，如接口的方法、Web 请求。
runMode	运行模式，如拦截模式、监控模式。

更多信息，请参见 添加限流规则。

4.3.2.3. 修改和删除限流规则

您可以随时对已有限流规则进行修改。对于不再需要的限流规则，您可以选择删除。修改和删除操作实时生效。

修改限流规则

1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 服务治理**。
3. 单击 **服务限流**，然后单击目标应用左侧的加号（+）。
4. 单击目标限流规则右侧的 **修改**。

5. 根据您的需求修改限流规则后，单击 **提交并推送**。

删除限流规则

1. 在 **服务限流** 页签，单击目标应用左侧的加号（+）。
2. 单击目标限流规则右侧的 **删除**，然后单击 **确定**。

   如果限流规则的状态为 **开启**，您需要将状态修改为 **关闭** 后，再删除限流规则。

4.3.2.4. 限流算法选择

服务限流中主要使用了 QPS 限流算法和令牌桶算法两种限流算法，本文对这两种算法进行介绍。

QPS 限流算法

QPS 限流算法通过限制单位时间内允许通过的请求数来限流。

优点：

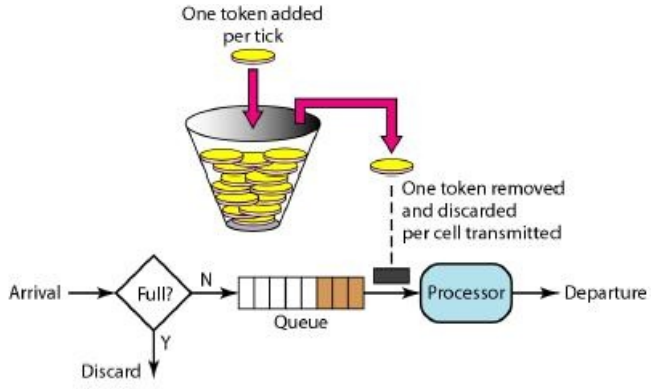
- 计算简单，是否限流只跟请求数相关，放过的请求数是可预知的（令牌桶算法放过的请求数还依赖于流量是否均匀），比较符合用户直觉和预期。
- 可以通过拉长限流周期来应对突发流量。如 1 秒限流 10 个，想要放过瞬间 20 个请求，可以把限流配置改成 3 秒限流 30 个。拉长限流周期会有一定风险，用户可以自主决定承担多少风险。

缺点：

- 没有很好的处理单位时间的边界。比如在前一秒的最后一毫秒和下一秒的第一毫秒都触发了最大的请求数，就看到在两毫秒内发生了两倍的 QPS。
- 放过的请求不均匀。突发流量时，请求总在限流周期的前一部分放过。如 10 秒限 100 个，高流量时放过的请求总是在限流周期的第一秒。

令牌桶算法

令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。



优点：

- 放过的流量比较均匀，有利于保护系统。
- 存量令牌能应对突发流量，很多时候，我们希望能放过脉冲流量。而对于持续的高流量，后面又能均匀地放过不超过限流值的请求数。

缺点：

- 存量令牌没有过期时间，突发流量时第一个周期会多放过一些请求，可解释性差。即在突发流量的第一个周期，默认最多会放过 2 倍限流值的请求数。
- 实际限流数难以预知，跟请求数和流量分布有关。

存量桶系数

令牌桶算法中，多余的令牌会放到桶里。这个桶的容量是有上限的，决定这个容量的就是存量桶系数，默认为 1.0，即默认存量桶的容量是 1.0 倍的限流值。推荐设置 0.6~1.5 之间。

存量桶系数的影响有两方面：

- 突发流量第一个周期放过的请求数。如存量桶系数等于 0.6，第一个周期最多放过 1.6 倍限流值的请求数。
- 影响误杀率。存量桶系数越大，越能容忍流量不均衡问题。

  误杀率：服务限流是对单机进行限流，线上场景经常会用单机限流模拟集群限流。由于机器之间的秒级流量不够均衡，所以很容易出现误限。例如两台服务器，总限流值 20，每台限流 10，某一秒两台服务器的流量分别是 5、15，这时其中一台就限流了 5 个请求。减小误杀率的两个办法：

  - 拉长限流周期。
  - 使用令牌桶算法，并且调出较好的存量桶系数。

如何选择限流算法

您可以会根据您的实际情况选择限流算法，建议如下：

- 当单机 QPS < 100 时，建议使用令牌桶算法。
- 当单机 QPS > 100 时，可以选择 QPS 限流算法和令牌桶算法。
- 若您不能容忍单个周期放过的请求数超过限流值时，请选择 QPS 限流算法。

4.3.2.5. 配置限流对象方法签名

服务限流可以对方法的参数进行过滤，以实现对某个特定的参数进行限流。

配置接口方法类型的限流对象

接口方法类型的限流对象的参数配置包括以下内容：

参数	说明

参数	说明
限流对象名	包括要限流的接口与方法名： • **接口**：支持 RPC 服务接口或配置了 Spring AOP 拦截器的 Bean。 • **方法**：支持带参数或不带参数的方法签名。详情请参见 配置方法签名。 例如上图中的配置表示限流对象为 `com.alipay.antcloud.dsrconsole.core.service.guardian.facade.GuardianAppFacade.queryAppNames`。方法的第一个参数中，`instanceId` 属性值为 `000001` 的请求。 `ARGS` 是服务限流内部定义的一个变量，表示方法的所有参数。 `ARGS[0]` 表示第一个参数。
键值	限流参数及属性名称，用 MEVL 表达式 表示，获取用于比较的键值。
比较关系	等于或不等于。
比较值	用于比较的属性值。

配置方法签名

对于接口方法类的限流规则，如果需要指定限流的具体接口及方法，您必须完成方法签名的配置。在配置方法名时，您可以根据实际情况选择是否在方法签名中添加参数。

- 方法不添加参数

  如果没有重载方法，或需要对所有重载方法限流，则不需要添加参数。例如限流对象接口中有以下几个同名方法：

  ```
  testBreakerScriptCondition(){}

  testBreakerScriptCondition(String name,Integer value){}

  testBreakerScriptCondition(int a,int[] al){}
  ```

  配置限流对象方法为 `testBreakerScriptCondition`，则对所有同名方法的总流量限流。

- 方法添加参数

  接口中有多个同名方法时，如果需要对某个具体方法限流，可以添加入参。添加参数时需要注意以下几点：

  - 不要使用形参。
  - 入参类型使用完整的类名。
  - 参数的逗号前后不要有空格。

- 支持基本类型和基本类型数组。

  例如方法 `foo(int a, int[] a1)` ，因为 `int[]` 的类型是 `[I` ，所以对应的方法配置为 `foo(int,[I)` ，其他基本类型的数组以此类推。

下面是添加参数的方法示例：

- `testBreakerScriptCondition(java.lang.String,java.lang.Integer)`
- `testBreakerScriptCondition(int,[I)`

接口方法中的 MVEL 表达式

配置方式

- 方式一：左侧键值计算结果是 true 或 false，右侧比较值中也填写 true 或 false。

  示例如下：

键值	比较关系	比较值	操作
ARGS[0].instanceId == '000001'	等于	true	修改 删除

- 方式二：左侧键值计算结果是个普通字符串，右侧比较值中也填写一个字符串。

  示例如下：

键值	比较关系	比较值	操作
ARGS[0].instanceId	等于	000001	修改 删除

上述两种方式的效果一样，推荐用第一种方式。第一种方式支持更多的运算符，例如 `&&`、`||`、`>`、`<=` 等，表达能力更丰富。

配置样例

- 使用 MVEL 表达式获取参数的属性值：

  您可以使用 `obj.field` 的格式获取参数的属性值，属性必须有 public 的 `getter` 方法，或本身是 public 的。若没有属性值，只有 public 的 `getter` 方法也可以。获取到的属性值可以和特定的值比较，例如 `ARGS[0].field == 'loull'` 。

- 使用 MVEL 表达式的基础运算符：
  - `!=` ，例如 `ARGS[0].id != 100` 。
  - `==` ，例如 `ARGS[1].uid == 'test'` 。
  - `>=` ，例如 `ARGS[0].number >= 200` 。
  - `>` ，例如 `ARGS[0].number > 100` 。

- `<=`，例如 `ARGS[0].number <= 101`。
- `<`，例如 `ARGS[0].number < 200`。
- `+ - * /`，例如 `ARGS[0].num1 + ARGS[1].num2 > ARGS[2].num3`。
- `&& ||`，例如 `ARGS[0].number > 100 && ARGS[0].number < 200`。

- 使用 MVEL 表达式获取 Date 类型：

  例如 `ARGS[0].getTime()<123123123`。

- 使用 MVEL 表达式获取 Enum 枚举值：

  例如 `AccountTypeEnum` 类型

  ```
  AccountTypeEnum type = AccountTypeEnum.CORPORATE_ACCOUNT;
  ```

  匹配名字属性可以配置为 `ARGS[0].name == 'CORPORATE_ACCOUNT'`。

- 使用 MVEL 表达式获取数组元素：

  例如 `ARGS[0][0]=='20170802000770000000022076255'` 表示第一个参数是数组，数组的第一个元素是 `20170802000770000000022076255`。

- 使用 MVEL 表达式操作集合：
  - List 类型

    例如 `ARGS[0].get(1)=='test2'`。

  - Map 类型

    例如

    ```
    Map<String,Object> dataMap = new HashMap<String, Object>();
    dataMap.put("testInteger",new Integer(20)); dataMap.put("testDouble",new Double(30));
    ```

    匹配表达式可以表示为 `ARGS[0].get('testDouble') == 30.0` 或 `ARGS[0].testDouble == 30.0`。

- 使用关键字 contains 做范围匹配：
  - 是否包含在集合内：`['aa', 'bb', 'Xin'].contains([0].last)` 或者 `[0].namelist contians ('Xi')`，其中 `[0].namelist` 是数组，不是字符串。
  - 是否包含在字符串内：`[0].last contains 'in'`，其中 `[0].last` 是字符串，判断是否包含 `'in'`。

- 使用关键字 IN 做范围匹配：

用于比较一个参数是否在一个白名单或黑名单范围内的场景。`IN` 表示在名单范围内，则匹配成功，`NOT_IN` 相反。例如：`ARGS[0].id IN 1,2,3,100` 。

> **重要**
>
> 白名单或黑名单列表的元素不能超过 100 个。

- 使用 MVEL 表达式执行参数的 public 方法：

  可以调用某个参数的 public 方法，用返回的结果和特定的值比较。例如 `[0].publicMethod == 'xxxx'` 。

配置 Web 请求中的限流对象

Web 类型的限流对象的参数配置包括以下内容：

参数	说明
限流对象名	Web 请求中的 URI，不包括域名和参数部分。
键值	Web 请求 URL 中的参数键值，不支持 MVEL 表达式。
比较关系	等于或不等于。
比较值	用于比较的属性值。

> **说明**
>
> Web 类型的限流参数键值不支持 MVEL 表达式。 Web 类型的限流参数键值不支持 ARGS 变量。例如 `/queryAllNames?instanceId=00001&name=cloudinc` 请求，参数之间没有顺序关系，所以对于 Web 请求的参数过滤不能使用 ARGS 变量。

下图给出了针对请求 URL `http://xxx.domain//webapi/guardian/history/search?instanceId=000001` 限流对象配置样例：

其中，`instanceId` 是请求 URL 中的参数的键值，`00001` 是参数中的属性值。

4.3.3. 限流日志

服务限流的限流日志打印在 `${user.name}/logs/guardian` 中，包括默认日志、运行错误日志和限流统计日志。

默认日志

服务限流的默认日志是打印在 `guardian/guardian-default.log` 中，主要打印推送过来的限流配置信息，日志内容没有固定格式。日志样例如下：

```
2016-12-1219:49:09,610 INFO  RegistringGuardianCodeWrapperInterceptor
2016-12-1219:49:10,757 WARN  receive message with key=[guardianConfig]and value=[{"@type":"
com.alipay.guardian.client.drm.GuardianConfig","engineConfigs":{"@type":"java.util.HashMap"
,"LIMIT":{"@type":"com.alipay.guardian.client.engine.limit.LimitEngineConfig","actionConfig
Map":{"@type":"java.util.HashMap",880:{"@type":"com.alipay.guardian.client.engine.limit.Lim
itActionConfig","actionType":"LIMIT_EXCEPTION","id":880,"responseContent":"限流配置-接口-多计
算模型-抛出异常"}},"globalConfig":{"enable":true,"runMode":"CONTROL"},"resourceConfigList":[{
"baseName":"com.alipay.guardiantestsofalite.facade.GuardianTestTrServiceFacade.testLimitBas
icCondition","id":379,"resourceType":"METHOD","ruleIds":[880]}],"ruleConfigMap":{"@type":"j
ava.util.HashMap",880:{"@type":"com.alipay.guardian.client.engine.limit.LimitRuleConfig","a
ctionId":880,"calculationConfigs":[{"calculationType":"INVOKE_BY_TIME","maxAllow":10,"perio
d":5000},{"calculationType":"INVOKE_BY_TIME","maxAllow":10,"period":5000},{"calculationKey"
:"[0].booleanValue","calculationType":"INVOKE_BY_TIME_CATEGORY","period":5000,"tairCompareK
ey":"true>5,false>6"}],"enable":true,"extParamConfigs":[],"id":880,"limitType":"GENERIC_LIM
IT","paramConfigs":[{"checkMode":"BYVALUE","compare":"EQUALS","key":"[0].stringValue","valu
e":"testStrMutilBasicParams"},{"checkMode":"BYVALUE","compare":"EQUALS","key":"[1].stringVa
lue","value":"MultileCalculations"}],"paramRelation":"AND","ruleBizId":"[tr]限流配置-接口-基
本参数多项-多个计算模型","runMode":"CONTROL","trafficType":"all"}}},"FUSE":{"@type":"com.alipa
y.guardian.client.engine.fuse.FuseEngineConfig","actionConfigMap":{"@type":"java.util.HashM
ap"},"ruleConfigMap":{"@type":"java.util.HashMap"}}},"version":1}]
2016-12-1219:49:10,759 WARN  after update with key=[guardianConfig]
2016-12-1219:49:11,195 INFO  GuardianConfig version=1
2016-12-1219:49:11,197 WARN  rebuild Rules,GuardianFactory:class com.alipay.guardian.client
.limit.LimitGuardianFactory
```

运行错误日志

服务限流的运行时错误日志打印在 `guardian/guardian-error.log` 中，主要打印一些错误信息，其中的错误堆栈信息需要重点关注，日志内容没有固定格式。

限流统计日志

服务限流的限流统计日志是 `guardian/guardian-limit-stat.log` ，日志内容为固定格式。日志样例如下：

```
2016-11-2100:00:02,001 INFO  MONITOR,43,test,1000,2016-11-21T00:00:01,2016-11-21T00:00:02,I
NVOKE_BY_TIME,10,40,10,30
```

示例值对应说明如下：

示例值	说明
2016-11-2100:00:02	日志打印时间。
001	请求耗时，单位：ms。
INFO	日志等级，取值为：INFO、DEBUG、ERROE、WARN。 默认为：INFO。
MONITOR	限流模式，取值如下： • MONITOR：表示当前的限流模式为监控模式。 • CONTROL：表示当前的限流模式是拦截模式。
43	限流规则 ID。
test	限流规则名称。
1000	统计间隔。
2016-11-21T00:00:01	本次统计的开始时间。
2016-11-21T00:00:02	本次统计的结束时间。
INVOKE_BY_TIME	统计类型。
10	限流规则阈值。
40	限流周期内的总请求数。
10	限流周期内的放行请求数。
30	表示当前限流周期内被限流的请求数。

4.4. 服务路由

当服务消费者面临多个服务提供者时，需要通过路由规则来确定具体的服务提供者。服务路由功能提供了灵活的路由功能，允许您定义多条服务路由规则，可以帮助您解决多个场景下的难题。

功能简介

服务路由多用于线下测试连调、蓝绿发布、灰度引流场景，将因版本升级造成的问题影响降到最低。

- 线下测试联调

  线下测试时，可能会缺少相应环境。您可以将测试应用注册到线上，然后开启路由规则，在本地进行测试。

- 蓝绿发布

  蓝绿发布时，您可以通过路由规则将流量切换到绿组，以测试新版本应用是否正常。

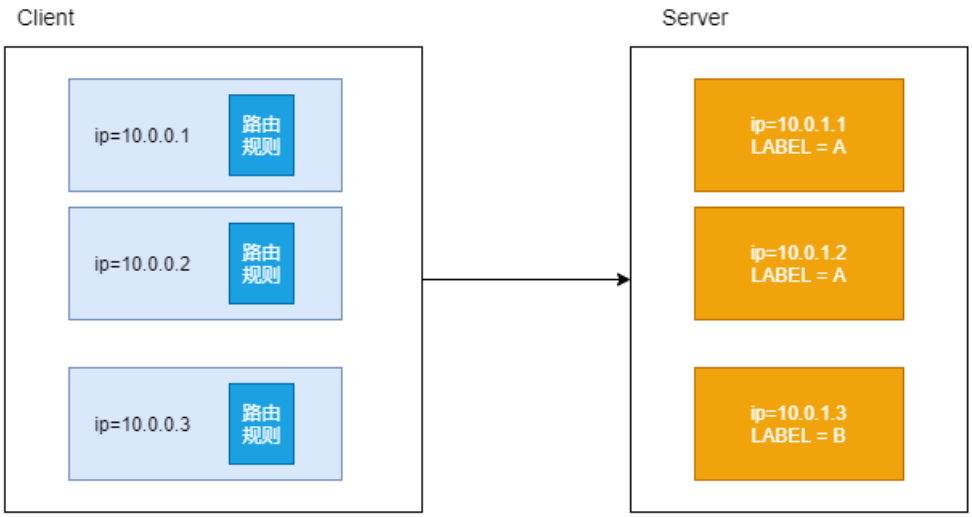
- 灰度引流

  灰度发布时，您可以通过路由规则将流量引导到灰度发布的应用版本上，测试灰度发布的版本是否正常。

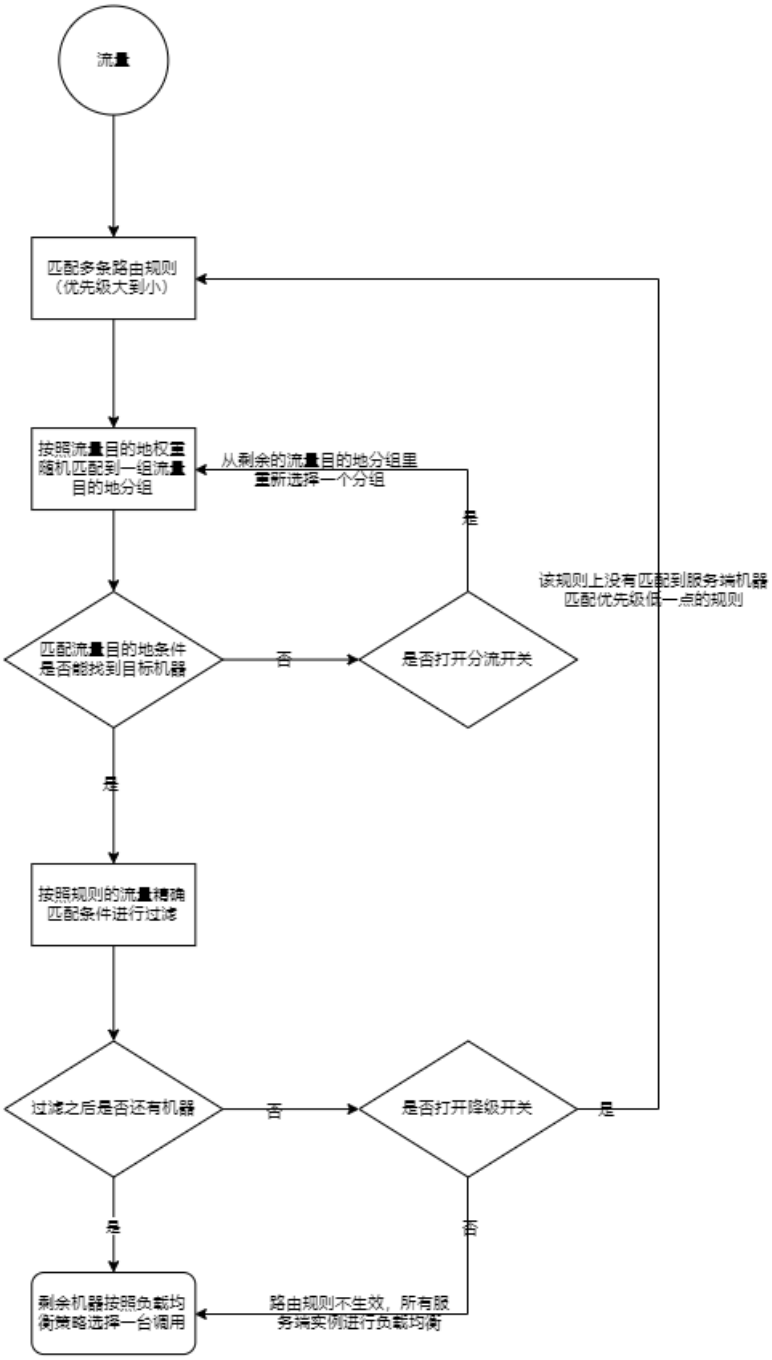
> **说明**
>
> 当前版本仅支持 SOFA 和 Dubbo 服务的路由，暂不支持 Spring Cloud 服务的路由。

配置的路由规则实际在服务消费方（即客户端）生效，通过界面上的配置来过滤服务提供方，目前主要有 RPC-Client（经典微服务生效）和 MOSN（服务网格生效）两种执行实体。

在部署服务提供方时，您可以在机器上加固定格式的标签，比如 LABEL=A。



路由规则生效流程如下图所示：



添加路由规则

1. 登录微服务控制台。
2. 在左侧菜单栏选择 **经典微服务 > 服务治理**，然后单击 **服务路由** 页签。
3. 单击 **添加路由组规则**，然后配置以下参数：

区域	参数	说明
	规则组名称	配置路由规则组名称。

区域	参数	说明
基本信息	服务方应用名	配置服务端的应用名称。
	服务	配置应用包含的服务。 单击 **切换输入模式** 可在手动填写与下拉选择之间切换。
规则属性	规则名称	配置路由规则名称。
	优先级	配置路由的优先级，数字越大优先级越高。 同时存在多条路由时，按照路由优先级大小进行匹配。
	规则开关	配置是否开启该条路由规则。默认开启。
	分流开关	当流量目的地分组没有服务实例时，是否转发到其他分组。 ○ 开启后，流量在高权重分组中未匹配到服务实例，会继续去匹配其他分组。 ○ 关闭后，流量不会匹配其他分组。
	降级开关	当整条规则都匹配不到服务实例时，是否继续匹配其他规则。 ○ 开启后，流量未在当前规则中匹配到服务实例时，会继续去匹配优先级更低的路由规则。 ○ 关闭后，流量不会匹配其他路由规则。

区域	参数	说明
	流量精确匹配（可选）	通过流量匹配条件来过滤需要执行路由规则的请求，满足这些条件的流量才会使用这条规则，不满足这条规则的就走降级开关逻辑，往低优先级的路由规则去匹配。不填表示匹配所有流量。 多个条件按照顺序执行，直到被某一个规则被拦截或全部通过。规则主要包括下述内容： ○ **字段**：包括系统字段、请求头。 ○ **字段名**：根据字段类型有不同的值。 ■ 系统字段：包括流量类型、调用方应用名、调用方 IP、服务方应用名。 ■ 请求头：请求头是指协议的请求头，比如 Dubbo 协议取的是 attachment，HTTP 协议取的是 Request Header。用户可以在应用系统中自定义请求头参数和值。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：字段名对应的值。
流量目的地	**权重**	设置当前流量目的地分组的权重，多个流量目的地的权重之和为 100%。 您可以单击最下方的 **添加** 按钮，配置多个流量目的地分组。
	服务实例分组条件	配置服务实例分组条件，多个条件之间是与的关系。 ○ **字段名**：可从下拉列表选择，或者自定义输入。 ○ **逻辑**：等于。 ○ **字段值**：配置字段名对应的值。 您可以单击下方的 **添加** 按钮，配置多个分组条件。

4. 单击 **提交**。
5. 在服务路由列表中，将刚创建的路由规则的状态改为 **开启**。

> **重要**
> 您可以添加多条路由规则，相同应用名称的路由规则会被合并。

编辑路由规则

您可以随时编辑已创建的路由规则，规则提交后实时生效。

1. 在 **服务路由** 页签，单击目标应用左侧的加号（+）。
2. 单击目标路由规则右侧的 **编辑**。

3. 按需求编辑路由规则后，单击 **提交**。

删除路由规则

您可以删除已创建的路由规则，删除规则会实时生效，请谨慎操作。

1. 在 **服务路由** 页签，单击目标应用左侧的加号（+）。
2. 单击目标路由规则右侧的 **删除**。
3. 单击 **确定**。

4.5. 服务熔断

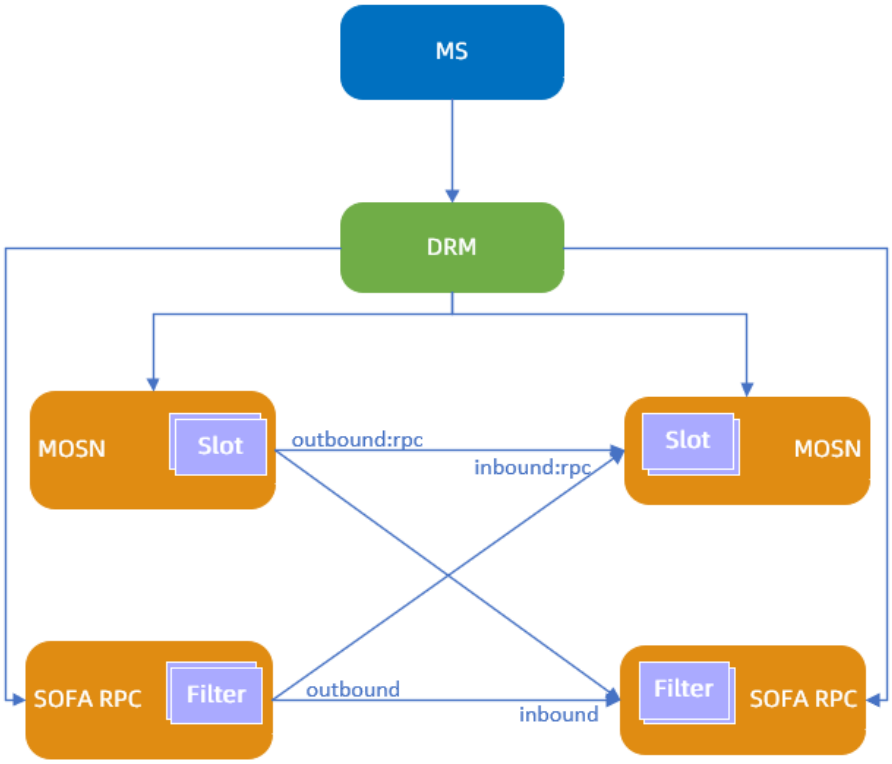
当服务中的服务端接口不稳定，出现频繁超时或错误时，可能会引起服务调用雪崩。您可以对应用开启服务熔断功能，使有故障的服务端及时返回错误，并释放系统资源，提高用户体验和系统性能。

功能简介

您可以通过下述操作让故障处于可控范围：

- 通过监控或者服务拓扑查看到某个服务延时较大、错误率较多后，进行服务治理。
- 选择异常服务后，进行自动熔断设置，可自定义熔断条件，例如：在某个时间段内（例如 10s）请求数达到某个值，且错误率或者延时达到某个值。

满足条件，则可以触发熔断。服务熔断后还可以持续测试该服务，进行自动熔断恢复。服务熔断的规则发布流程如下：



1. 通过微服务管控台 MS 下发熔断的动态配置到分布式配置中心 DRM。
2. DRM 将熔断的动态配置下发到 MOSN。
3. MOSN 根据平均 RT 和错误率触发熔断。

熔断可在客户端或服务端生效。

创建熔断规则

1. 登录微服务控制台。
2. 在左侧菜单栏选择 **经典微服务 > 服务治理**，然后单击 **服务熔断** 页签。
3. 单击 **添加熔断组规则**，然后配置以下参数：

参数	说明
规则名称	配置服务熔断规则的名称。
应用	配置检测的目标应用。 您可以手动填写或在下拉列表中选择。
熔断方向	配置服务熔断的方向，取值如下： ○ **客户端熔断**：表示熔断规则在客户端生效。在客户端侧统计调用服务端的请求指标，应用填写客户端的应用，服务和方法填写引用的服务端的服务和方法。 ○ **服务端熔断**：表示熔断规则在服务端生效，每个服务端单独统计请求指标，应用填写服务端的应用，服务和方法填写服务端的服务和方法。
熔断服务	配置熔断的服务。单击 **切换输入模式** 可在手动填写与下拉选择之间切换。
方法	配置需要熔断的服务的方法名。
熔断类型	配置熔断的类型，取值如下： ○ **错误比率**：熔断时间窗口内的请求错误率大于错误率阈值时，触发熔断。 ○ **平均 RT**：熔断时间窗口内的请求数阈值大于平均响应时间 RT（Response Time）的阈值时，触发熔断。
运行模式	配置熔断的运行模式，取值如下： ○ **拦截模式**：满足条件的请求会触发熔断规则。 ○ **观察者模式**：熔断规则不会被触发，只会在 MOSN 里打印日志。

熔断配置	配置触发熔断的各项参数： ○ **熔断指标窗口**：和请求数阈值搭配使用，表示熔断指标窗口内至少有请求数阈值的请求就会纳入熔断指标统计。单位为秒。 ○ **请求数阈值**：和熔断指标窗口搭配使用。单位为个。 ○ **错误率阈值**：请求数的平均错误率达到此阈值时触发熔断。熔断类型配置为 **错误比率** 时配置。 ○ **平均 RT 阈值**：请求响应时间超过此阈值时触发熔断。单位为毫秒。熔断类型配置为 **平均 RT** 时配置。 ○ **熔断时间段**：触发熔断后的熔断时长，熔断时长之内的请求会直接返回错误。当熔断超过该时间后会再次尝试调用服务端，如果还是失败则继续触发熔断。 ○ **RPC 超时时间**：经典微服务场景会使用。单位为毫秒。熔断类型配置为 **错误比率** 时配置。
流量精确匹配（可选）	配置之后，只有满足匹配条件的流量才会使用这份熔断规则。置空表示匹配所有流量。 您可以配置多条匹配条件，多个条件是与的关系，按顺序进行匹配。参数配置如下： ○ **字段**：可选择系统字段和请求头。 ○ **字段名**：根据字段类型有不同的值。 ■ 系统字段：包括流量类型、调用方应用名、调用方 IP、服务方应用名。 ■ 请求头：请求头是指协议的请求头，比如 Dubbo 协议取的是 attachment，HTTP 协议取的是 Request Header。用户可以在应用系统中自定义请求头参数和值。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：字段名对应的值。

4. 单击 **提交**，然后单击 **确定**。
5. 在熔断规则列表中，将刚刚创建的熔断规则的状态改为 **开启**。

编辑熔断规则

您可以随时编辑已创建的熔断规则，规则提交后会实时生效。

1. 在 **服务熔断** 页签，单击目标应用左侧的加号（+）。
2. 单击目标熔断规则右侧的 **编辑**。
3. 按需求编辑熔断规则后，单击 **提交**。

删除熔断规则

您可以删除已创建的熔断规则，删除操作会实时生效，请谨慎操作。

1. 在 **服务熔断** 页签，单击目标应用左侧的加号（+）。
2. 单击目标熔断规则右侧的 **删除**。

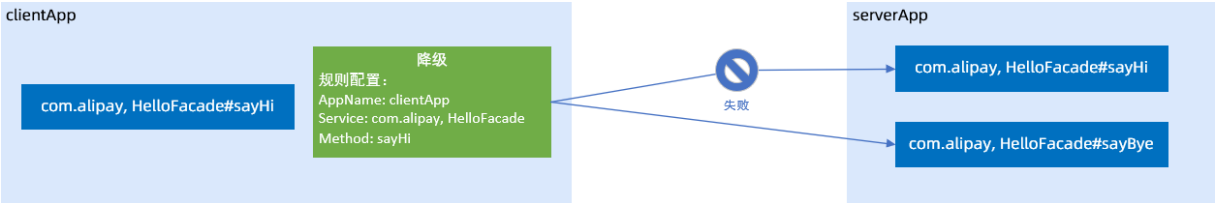
3. 单击 **确定**。

4.6. 服务降级

当服务器压力剧增的情况下，根据实际业务情况及流量，对某些不重要的服务，不处理或换种简单的方式处理，从而释放服务器资源以保证核心业务正常运作或高效运作。

降级原理

降级在客户端实现，当客户端在调用服务端时及时响应一个降级错误码，不会真正请求到服务端。目前支持服务级降级和方法级降级两种方式。实现流程如下：



配置服务降级规则

1. 登录微服务控制台。
2. 在左侧菜单栏选择 **经典微服务 > 服务治理**，然后单击 **服务降级** 页签。
3. 单击 **添加降级组规则**，然后配置以下参数：

配置项	说明
规则名称	配置降级规则的名称。
调用方应用	降级在客户端生效，这里填写或选择客户端应用。
运行模式	设置降级规则的运行模式，取值如下： ○ 拦截模式：降级规则生效。 ○ 观察者模式：降级规则不会生效，只会在 MOSN 中打印日志。
降级服务	选择或填写客户端引用的服务端服务名称。星号（*）表示所有服务。
降级方法	填写待降级的方法名。星号（*）表示所有方法。

4. 单击 **提交**，然后单击 **确定**。
5. 在降级规则列表中，将刚刚创建的规则状态修改为 **开**。

编辑服务降级规则

您可以随时编辑已创建的服务降级规则，规则提交后实时生效。

1. 在 **服务降级** 页签，单击目标应用左侧的加号（+）。

2. 单击目标服务降级规则右侧的 **编辑**。
3. 按需求编辑服务降级规则后，单击 **提交**。

删除服务降级规则

您可以删除已创建的服务降级规则，删除操作实时生效，请谨慎操作。

1. 在 **服务降级** 页签，单击目标应用左侧的加号（+）。
2. 单击目标服务降级规则右侧的 **删除**。
3. 单击 **确定**。

4.7. 故障注入

您可以通过故障注入功能向测试应用注入故障，检测应用面对异常时的处理情况。您可以根据检测的情况调整您的应用，以减少应用在正式使用时出现的异常问题。多用于测试环境。

功能简介

故障注入流程如下所示：

在微服务中，其实现方式为：

1. 管控台 MS 拼接故障注入规则，将其发送到 DRM。
2. MOSN 和 RPC 客户端订阅 DRM 的 Key 值。

配置故障注入规则

1. 登录微服务控制台。
2. 在左侧菜单栏选择 **服务网格** > **服务治理**，然后单击 **故障注入** 页签。
3. 单击 **添加注入组规则**，然后配置以下参数：

参数	说明
规则名称	设置故障注入规则的名称。
应用	选择或填写目标应用的名称。星号（*）表示所有应用。
注入方向	设置故障注入的方向，可选值为： ○ **服务端注入**：在应用的服务端注入故障。 ○ **客户端注入**：在应用的客户端注入故障。

参数	说明
服务	配置注入故障的服务。单击 **切换输入模式** 可在手动填写与下拉选择之间切换。
方法	配置故障注入的方法。星号（*）表示所有方法。
运行模式	配置故障注入规则的运行模式，取值如下： ○ **拦截模式**：满足条件的故障注入请求会被注入。 ○ **观察者模式**：满足条件的故障注入请求不会被注入，只会在 MOSN 里打印日志。
故障类型	故障注入支持注入错误或者超时等事件，方便服务的异常测试，用于模拟服务异常的情况。取值如下： ○ **中断异常**：注入运行时异常，中断请求并返回既定的错误状态码。 ○ **超时异常**：满足条件的请求，会增加响应时间，请求正常调用。
超时时间	故障类型为 **超时异常** 时，设置异常的超时时间。
异常比例	设置注入异常流量的比例。例如设置为 80，则只注入 80% 的异常流量。
流量精确匹配（可选）	设置流量的匹配条件，满足匹配条件的流量才会使用故障注入规则。置空此项时表示匹配所有流量。 您可以配置多条匹配条件，多个条件是与的关系，按顺序进行匹配。参数配置如下： ○ **字段**：可选择系统字段和请求头。 ○ **字段名**：根据字段类型有不同的值。 ■ **系统字段**：包括流量类型、调用方应用名、调用方 IP、服务方应用名。 ■ **请求头**：请求头是指协议的请求头。例如 Dubbo 协议取的是 attachment，HTTP 协议取的是 Request Header。用户可以在应用系统中自定义请求头参数和值。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：字段名对应的值。

4. 单击 **提交**，然后单击 **确定**。
5. 在故障注入规则列表中，将刚刚创建的故障注入规则的状态改为 **开启**。

> 说明
>
> 当多条故障注入规则针对同一个服务时，只会生效第一条。

编辑故障注入规则

您可以随时编辑已创建的故障注入规则，规则提交后实时生效。

1. 在 **故障注入** 页签，单击目标应用左侧的加号（+）。
2. 单击目标故障注入规则右侧的 **编辑**。
3. 按需求编辑故障注入规则后，单击 **提交**。

删除故障注入规则

您可以删除已创建的故障注入规则，删除操作实时生效，请谨慎操作。

1. 在 **故障注入** 页签，单击目标应用左侧的加号（+）。
2. 单击目标故障注入规则右侧的 **删除**。
3. 单击 **确定**。

4.8. 服务鉴权

服务提供者提供服务后，您可以通过服务鉴权功能对服务调用方进行鉴权。

注意事项

- 服务鉴权功能支持 SOFARPC、Dubbo、Spring Cloud 三种框架。其中，Dubbo 和 Spring Cloud 需要确保引入的 `sofa-registry-cloud-all` 依赖版本为 1.2.8 及以上版本，详情请参见 SDK 版本说明。
- 在使用容器应用服务发布应用时，应用名称必须与本地应用注册代码配置的 `spring.application.name` 一致。
- 请确保 SOFABoot 版本在 3.3.3 及以上。有关 SOFABoot 的版本信息，请参见 SOFABoot 版本说明。

添加鉴权规则

1. 登录微服务控制台。
2. 在左侧菜单栏选择 **经典微服务 > 服务治理**，然后单击 **服务鉴权** 页签。
3. 单击 **添加鉴权规则**，然后配置以下参数：

参数	说明
鉴权粒度	配置鉴权的粒度，可选值为： ○ **应用级**：对某个应用添加鉴权规则。 ○ **服务级**：对应用下的一个或多个服务添加鉴权规则。

参数	说明
规则名称	配置鉴权规则的名称。 仅支持中文、英文、数字、下划线（_），最多支持 255 个字符。
类型	配置鉴权类型，可选值为： ○ **白名单**：选择此项时，匹配条件的请求会被放通。 ○ **黑名单**：选择此项时，匹配条件的请求会被拒绝。
应用	填写待鉴权的应用。星号（*）表示所有应用。
服务	选择或填写应用下的一个或多个服务。仅在 **鉴权粒度** 选择 **服务级** 时配置。
运行模式	配置服务鉴权规则的运行方式，取值如下： ○ **拦截模式**：鉴权规则生效，则拒绝请求。 ○ **观察者模式**：鉴权规则生效时，不拒绝请求，只打印日志。
匹配条件	配置鉴权规则的匹配条件，符合条件的流量会被鉴权。 可配置多条匹配规则，各匹配规则之间是与的关系。参数配置如下： ○ **字段**：可选择系统字段和自定义字段。 ○ **字段名**：根据字段类型有不同的值。 ■ 系统字段：可选择调用方应用名、调用方 IP、服务方应用名、服务方方法名。 ■ 自定义字段：根据实际需求自行设置字段名。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：填入所选字段的值。

4. 单击 **提交**，然后单击 **确定**。
5. 在鉴权规则列表中，将刚刚创建的鉴权规则的状态改为 **开**。
6. 根据设置的类型打开白名单或黑名单的开关。

   规则类型为白名单时，打开白名单开关；反之，打开黑名单开关。否则，鉴权规则不生效。

编辑鉴权规则

您可以随时编辑已创建的鉴权规则，规则提交后实时生效。

1. 在 **服务鉴权** 页签，单击目标服务左侧的加号（+）。
2. 单击目标鉴权规则右侧的 **编辑**。

3. 按需求编辑鉴权规则后，单击 **提交**。

删除鉴权规则

您可以删除已创建的鉴权规则，删除操作实时生效，请谨慎操作。

1. 在 **服务鉴权** 页签，单击目标服务左侧的加号（+）。
2. 将目标鉴权规则的状态改为 **关**。
3. 单击目标鉴权规则右侧的 **删除**。然后单击 **确定**。

查看服务鉴权日志

您可以前往 `mosn-sidecar-container` 容器，服务鉴权日志打印在 `/home/admin/logs/mosn/rbac.log` 文件中。

4.9. 审计目录

审计目录功能用于查询用户的操作，并对操作进行溯源。操作审计的维度是实体 ID + 实体类型。

> **说明**
>
> 实体 ID 和实体类型相同的数据会保留最近 20 条，与按时间来保留数据的设计相比，该数据保留方式的优点在于：即使实体 ID 和实体类型已经创建了很长时间，您仍可以通过审计记录查询出数据。而如果以时间为保留标准，超过预定时间后，数据将会丢失。

应用场景

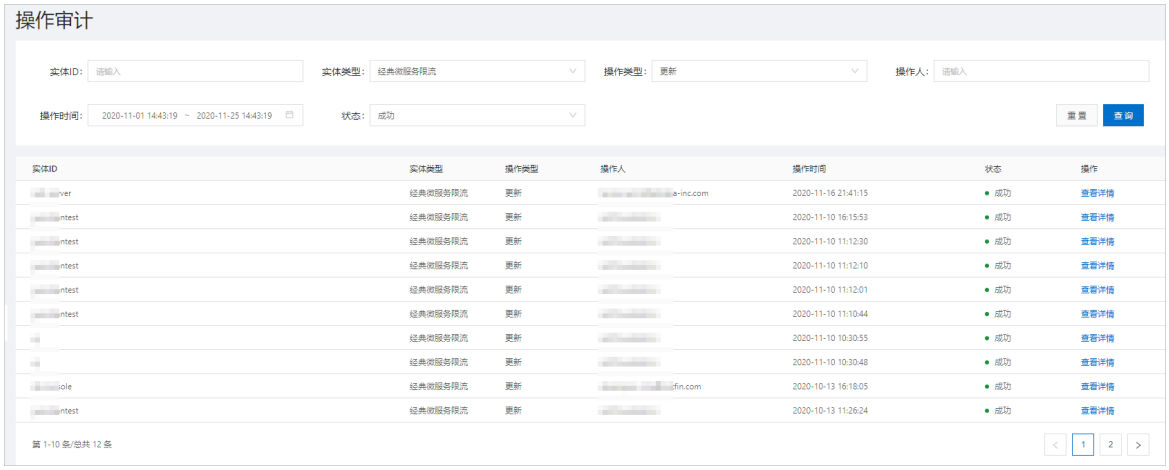
假设 `org.example.services.echoService.XyEchoService:1.0.0:default@crpc` 服务多次更新了路由规则，且部分路由规则失效了。您想查一下每次路由规则更新的内容，以及谁修改了失效的路由规则，可以通过下述步骤实现：

1. 根据实体 ID（服务级的实体 ID 是服务 ID；应用级的实体 ID 是应用名）和实体类型（服务路由）查询出最近的 20 条数据。
2. 单击审计记录的 **查看详情**，分析请求参数和响应参数。

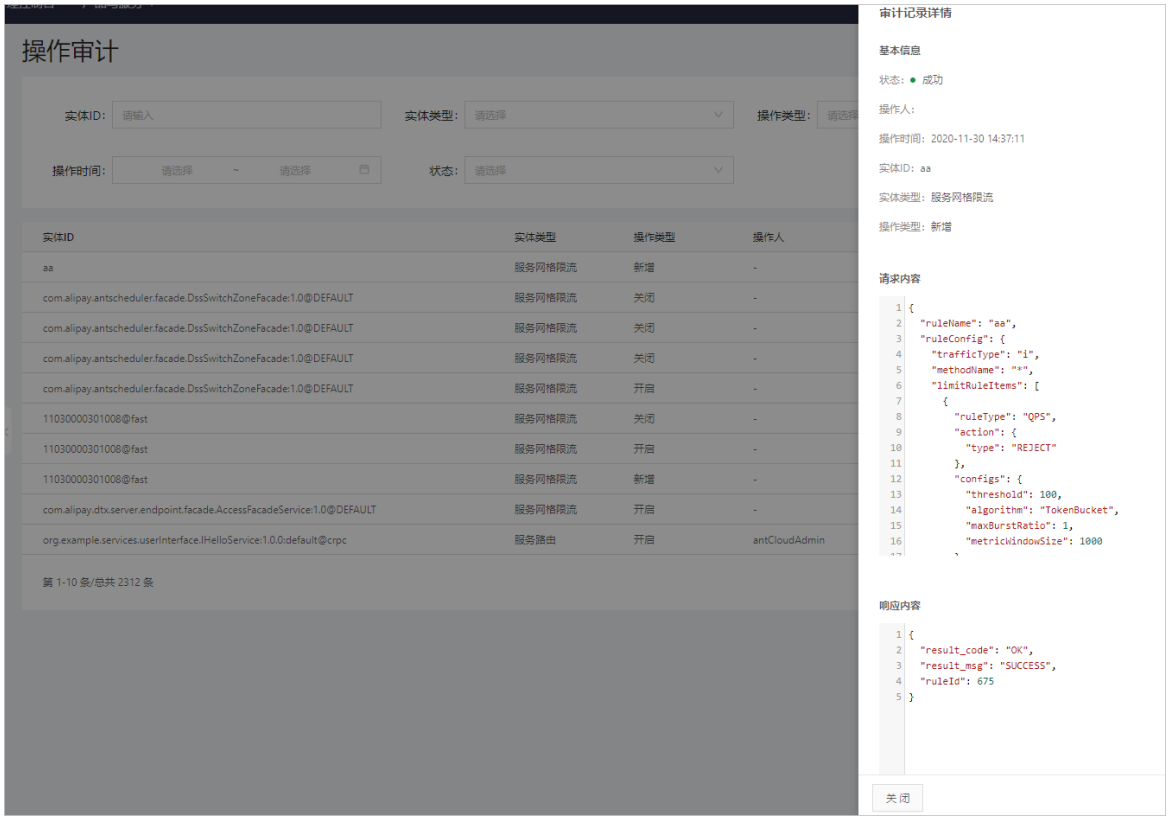
查询审计记录

1. 登录微服务控制台。
2. 在左侧菜单栏选择 **经典微服务 > 审计目录**。
3. 在操作记录列表中查看操作审计的基本信息。

基本信息包括实体 ID、实体类型、操作类型、操作人等。您也可以在搜索栏查询指定操作。



4. 单击目标操作右侧的 **查看详情**，查看操作的详细信息。



5.动态配置

5.1. 概述

动态配置（Distributed Resource Management，简称 DRM ）是一个分布式环境下，实时动态的配置管理框架。可以在应用没有重启的情况下，完成配置的动态更新。广泛用于业务参数配置、应急开关切换等场景。

动态配置是微服务下的模块之一，您只需要在每个环境开通中间件微服务，即可使用动态配置。用户实例之间的数据通过实例标识进行逻辑隔离，保证数据安全。

动态配置具有以下优点：

- 编程 API 简单

  面向注解和普通 JavaBean 编程，编程方式统一且简单。

- 实时性高

  秒级推送能力，集群实时配置变更。

- 一致性高

  除变更推送能力外，客户端还定时检查数据版本，一旦有数据不一致就触发主动拉数据。

5.2. 动态配置快速入门

动态配置（DRM）功能是基于 SOFABoot 工程实现的。本文介绍如何快速实现动态配置。

前提条件

- 已完成环境配置，配置详情请参见 搭建环境。
- 已下载 示例工程，并将示例工程主 `pom.xml` 文件的 SOFABoot 版本号改为最新版。SOFABoot 版本信息，请参见 版本说明。

配置步骤

1. 在本地开发应用。
   i. 在 SOFABoot 工程的待创建动态配置类的模块中，添加下述依赖：

   ```
   <dependency>
        <groupId>com.alipay.sofa</groupId>
        <artifactId>ddcs-enterprise-sofa-boot-starter</artifactId>
   </dependency>
   ```

   您无需关注该依赖版本。

   ii. 进行安全配置。

   为保障中间件的安全性，所有的调用均需要验证访问者的身份。配置方式，请参考 引入 SOFA 中间件。

   iii. 创建动态配置类。

   配置类代码示例：

```
@DObject(region ="AntCloud", appName ="dynamic-configuration-tutorial", id ="com.antcloud.tutorial.configuration.DynamicConfig")
    public class DynamicConfig{

        @DAttribute
        private String name;

        @DAttribute
        private int age;

        @DAttribute
        private boolean man;

        public void init(){
            DRMClient.getInstance().register(this);
        }

        public String getName(){
            return name;
        }

        public void setName(String name){
            this.name = name;
        }

        public int getAge(){
            return age;
        }

        public void setAge(int age){
            this.age = age;
        }

        public boolean isMan(){
            return man;
        }

        public void setMan(boolean man){
            this.man = man;
        }
    }
```

代码说明：

- 该配置类是一个普通的 Java 类，要符合 Java Bean 的规范，有若干私有属性，属性有对应的 `get` 和 `set` 方法。例如上面的 `name` 、 `age` 、 `man` 称为资源属性，资源属性只允许 `String` 和基本类型。

- 在配置类上加上 `@DObject` 注解，它的包名是 `com.alipay.drm.client.api.annotation` 。`@DObject` 需要提供 `id` 、 `region` 、 `appName` 属性。
  - `id` ：是全站唯一的字符串，一般用全类名来保证唯一，如不设值，则默认为全类名。
  - `region` ：用于区分不同组织的域，如可为每个子公司设定独立域。
  - `appName` ：是应用名。
- 在资源属性上加上 `@DAttribute` 注解，包名同样是 `com.alipay.drm.client.api.annotation` 。
- 动态配置框架将通过反射的方式调用 `get` 、 `set` 方法，从而读写资源属性。在特殊应用场景下，允许改变 `get` 、 `set` 方法的内容，但是不可以修改方法的形式（方法名、参数、返回值）。因为系统启动时动态配置框架会检查该属性是否符合 Java Bean 的规范，如果不符合，会跳过注册这个属性。
- 提供两个可选的注解 `@BeforeUpdate` ， `@AfterUpdate` 。如果需要在每个属性更新前或更新后执行统一的操作，例如打印日志，可以提供参数 `(String,Object)` 和无返回值的方法，并打上相应注解。

  > **说明**
  >
  > 这两个方法被调用时，传入的参数都是属性名和本次 set 方法的入参，并不是对应的私有属性更新前和更新后的值。这两个方法只适合用来执行打日志等次要任务，真正的业务逻辑要放在 set 方法中。

- 调用 `register` 方法注册到动态配置客户端后，即可享受服务端动态修改数据后的秒级推送能力。

iv. 进行 xml 配置。

示例如下：

```
<bean id="dynamicConfig" class="com.antcloud.tutorial.configuration.tutorial.config
.DynamicConfig" init-method="init"/>
```

2. 启动应用。

   启动应用，分为 2 种情形：

   - 本地启动：启动本地开发所完成的 SOFABoot 工程。
   - 云端启动：将本地工程打包后发布到云端启动，详情请参见 云端运行。

3. 配置云端动态配置。

   前往微服务控制台进行动态配置项的创建、管理与推送操作，实现在不重启应用的情况下，完成配置的动态更新。详情参见 新增动态配置 和 推送动态配置。

5.3. 新增动态配置

动态配置属性以键值对的形式定义，隶属于某一动态配置类。配置类与属性的关系可类比 Java 中的类与属性的关系。

配置步骤

1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 动态配置**。
3. 单击 **新建配置**，然后配置以下参数：

   您可以通过以下两种方式创建配置：

   - 自定义新建

     单击 **自定义新建**，然后配置以下参数：

参数	说明
域	填写配置类的命名空间。 只能包含大小写字母、数字或者下划线（_）。
所属应用	填写配置类所属的应用名。 ■ 只能以英文字母开头。 ■ 可以包含大小写字母、数字、下划线（_）、短划线（-）。
类标识	代表配置类的一个字符串，跟应用代码中 `@DObject` 注解的 ID 字段一致，通常使用全类名。 ■ 只能以英文字母开头。 ■ 可以包含大小写字母、数字、下划线（_）、短划线（-）、英文句号（.）。
描述（可选）	填写配置类的描述信息。

   - 从模板新建

     单击 **从模板新建**，然后根据模板要求填写相关参数。

4. 单击 **确定**。
5. 单击刚创建的配置类后侧的 **新增属性**，然后填写 **属性名** 与 **描述**。

   属性为 key-value 的形式。具体属性值在推送至服务器时定义。

6. 单击 **确定**。

   属性配置完成后，您可以选择将属性值直接发布到目标服务器上或通过灰度推送进行测试。更多信息，请参见 推送动态配置。

5.4. 推送动态配置

属性配置完成后，您可以根据实际需求推送动态配置。

目前支持以下两种方式推送动态配置：

- 直接推送：立即将配置发布至所有订阅服务器。建议在验证配置无误后再进行此操作。
- 灰度推送：仅将配置推送到几台服务器进行测试验证，并不保存数据到数据库。

操作步骤

1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 动态配置**。
3. 单击目标动态配置左侧的加号（+），然后单击目标属性名称。
4. 配置 **推送值** 后，单击需要的推送方式。
   - 直接推送

     单击 **推送配置**，然后单击 **确定**。动态配置将立即将发布至所有订阅服务器。
   - 灰度推送

     a. 单击 **灰度推送**。

     b. 选中需要推送的 IP 后，单击 **推送**。

     只有已经订阅该配置的服务器 IP 地址会显示在弹窗列表中。您可以使用右上角的搜索栏来快速筛选要查找的服务器 IP 地址。灰度推送时，您的配置仅推送到目标服务器进行测试验证，数据不保存到数据库。

查看推送记录

单击属性基本信息页面的 **查看推送记录** 链接以查看推送记录列表。

每条推送记录包括以下信息：

- 推送时间
- 操作者
- 推送值：本次操作推送的属性值。
- 推送结果：成功或失败。

单击 **复用** 将关闭推送记录窗口并将当前推送记录的推送值填充到推送值文本框中，方便您重新进行推送。

5.5. 使用注解标识配置类

动态配置的主要编程方式为使用注解标识配置类信息。本文介绍如何覆盖注解配置，实现更灵活的动态配置类初始化。

> **说明**
>
> 如何完全使用注解方式配置一个动态配置类，请参见 开始使用动态配置。

覆盖注解配置方式

动态配置客户端提供两种方式注册配置类：

- 直接注册含有所有注解配置的配置类

```
DistributedResourceManager#register(Object resourceObject);
```

- 注册配置类实例的同时，传入覆盖注解的配置项

```
DistributedResourceManager#register(Object resourceObject, Config config);
```

Config 包含 `@DObject` 中的所有属性，在 Config 中配置的值会覆盖注解配置。

属性注解高阶用法

动态配置默认用法是当服务端推送配置后，客户端启动时会默认同步加载服务端配置值。如果您希望服务端配置值仅在运行期生效，或者不希望客户端在启动期同步拉取配置值，可通过 `@DAttribute` 中的 `DependencyLevel` 来定义此属性的依赖等级。

属性依赖等级有以下几种：

依赖等级	依赖描述
NONE	无依赖，启动期不加载服务端值。启动此级别后，客户端仅会接收在运行期间服务端产生的配置推送。
ASYNC	异步更新，启动期异步加载服务端值，不关注加载结果。
WEAK	弱依赖，启动期同步加载服务端推送值。当服务端不可用时不影响应用正常启动；服务端可用后，客户端会依靠心跳检测重新拉取到服务端值。
STRONG	强依赖，启动期同步加载服务端值。如服务端未设置值，则使用代码初始化值。如从服务端获取数据请求异常或客户端设值异常时，均会抛出异常，应用启动失败。
EAGER	最强依赖，启动期必须拉取到服务端值。如服务端未推送过值则抛异常，应用启动失败。

5.6. 导出和导入动态配置

动态配置提供配置快速导出和导入功能，您可以通过导出功能将动态配置备份到本地，也可通过导入功能快速恢复动态配置。支持在不同环境中导出和导入动态配置。

导出动态配置

1. 登录微服务控制台。
2. 在左侧导航栏，选择 **经典微服务 > 动态配置**。
3. 在 **动态配置** 页面，选择 **更多 > 导出**。

动态配置文件为 JSON 格式，存放在浏览器默认下载文件夹中。

导入动态配置

1. 在 **动态配置** 页面，选择 **更多 > 导入**。
2. 单击 **浏览**，选中目标文件后单击 **打开**。

   支持导入 JSON 和 TXT 格式的动态配置文件。

动态配置文件格式

动态配置文件每一行对应一个完整的配置类 JSON 结构，配置类中可包含多个属性，多个配置类的 JSON 数据以换行符分隔。内容格式如下：

```
{
    "region":"Alipay",
    "appName":"testModel",
    "name":"测试配置",
    "resourceId":"com.alipay.test",
    "attributes":[
        {
            "attributeName":"age",
            "name":"年龄"
        },
        {
            "attributeName":"name",
            "name":"名称"
        }
    ]
}
```

参数说明如下：

参数	说明
region	配置类的命名空间。
appName	配置类所属的应用名。
resourceId	代表配置类的一个字符串，跟应用代码中 `@DObject` 注解的 ID 字段一致，通常使用全类名。
name	配置类的描述。
attributes	配置类的属性： • attributeName：属性名。 • name：属性描述。

配置文件示例：

```
[
    {
        "region":"Alipay",
        "appName":"testModel1",
        "name":"配置类描述",
        "resourceId":"com.alipay.test",
        "attributes":[
            {
                "attributeName":"age",
                "name":"属性描述"
            },
            {
                "attributeName":"name",
                "name":"属性描述"
            }
        ]
    },
    {
        "region":"Alipay",
        "appName":"testModel2",
        "name":"配置类描述",
        "resourceId":"com.alipay.test",
        "attributes":[
            {
                "attributeName":"age",
                "name":"属性描述"
            },
            {
                "attributeName":"name",
                "name":"属性描述"
            }
        ]
    }
]
```

5.7. 教程示例：使用动态配置

本文通过一个示例介绍动态配置的业务编码和日志排查。

准备工作

动态配置的示例是基于 SOFABoot 开发的。学习本课程前，确保您对 SOFABoot 有一定程度的了解。

- 已搭建 SOFABoot 基础环境。操作步骤，请参见 SOFABoot 环境搭建。
- 如果您在线下有联调环境，想在本地编译调试，需要了解 SOFABoot 项目如何编译运行。详情请参见 SOFABoot 编译运行。
- 如果您需要在服务器上部署示例代码，请参见 SOFABoot 应用发布。
- 下载 动态配置的示例工程。

步骤 1：云端管控

录入示例代码，配置云端管控动态配置类。详情请参见 动态配置快速入门。

步骤 2：发布应用

操作步骤，请参见 发布应用。

应用成功发布之后，您可以在动态配置控制台的资源查询页面查看运行代码的 ECS 和这些 ECS 内存中的属性值。

步骤 3：推送资源

在动态配置属性详情页输入需修改的值，然后推送配置。此配置会被及时推送至关心此配置的客户端，并修改客户端对应的动态配置属性值。操作步骤，请参见 推送动态配置。

推送配置后，可实时查看对此配置感兴趣的客户端与客户端中此属性的内存值。

查看运行结果

您可以通过 SSH 工具登录到应用的 ECS，查看客户端日志和业务日志：

动态配置客户端启动日志

日志目录：`/home/admin/drm/drm-boot.log`。

- 如果没有异常日志，表明动态配置客户端启动正常。
- 根据资源的 ID 进行查找，如示例中的资源 ID 为 `com.antcloud.tutorial.configuration.DynamicConfig`，就可以通过 `grep com.antcloud.tutorial.configuration.DynamicConfig drm-boot.log` 看到这个资源注册的相关日志。

动态配置推送日志

如果客户端启动正常，在推送资源后，您可以在动态配置的推送日志 `/home/admin/drm/drm-monitor.log` 中查找如下信息：

- “`Receive update command from zdrmdata server`”表示从服务端收到了更新资源的命令。
- “`Query data from zdrmdata`”表示客户端到服务端查询并更新了最新的推送值。

业务日志

在代码中，我们还通过 Logger 在 `set` 和 `before/update` 中打印了自己的业务日志，根据 log4j 中的路径，业务日志会打印在 `/home/admin/logs/service/default.log` 中。

代码解析

`DynamicConfig` 是一个动态配置类，具有以下特点：

- 是一个普通的 Java 类，符合 Java Bean 的规范。

- 它的属性均具有 `get/set` 方法（没有会导致资源注册失败），例如代码中的 `str`，称为资源属性。资源属性只允许 `String` 和基本类型。
- 资源类上加上注解 `@DObject`（包名为 `com.alipay.drm.client.api.annotation`），属性需要加上 `@DAttribute` 注解。
- `@BeforeUpdate` 和 `@AfterUpdate` 在每个属性更新前或更新后执行统一的操作，例如打印日志。这两个方法是可选的，只适合做一些非业务主流程的逻辑。
- 业务逻辑需要放在属性的 `set` 方法中执行。
- 注意如果业务逻辑执行耗时很长，最好能够异步处理，避免超时后动态客户端报错 “ `interrupt` ”。

6.SOFARegistry

6.1. 概述

SOFARegistry 是蚂蚁集团开源的一个生产级、高时效、高可用的服务注册中心，采用 AP 架构（CAP 理论中的 AP，强调可用性），支持秒级时效性推送，同时采用分层架构支持无限水平扩展。

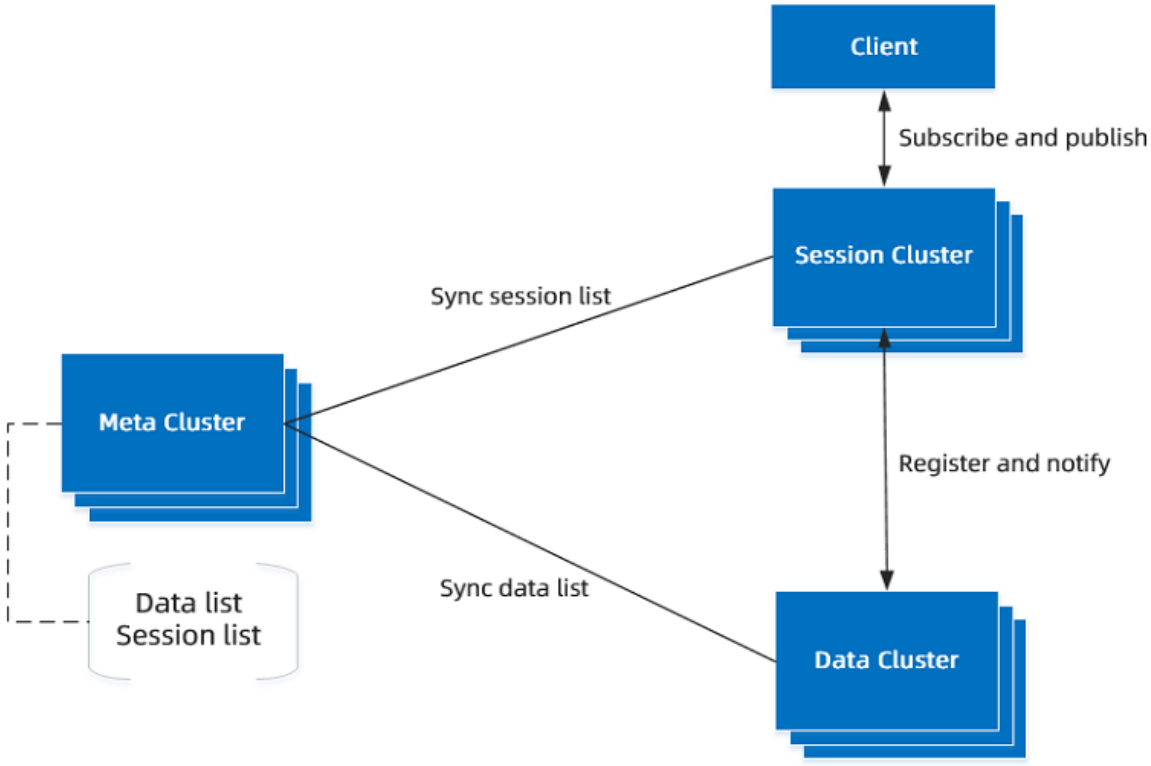
SOFARegistry 最早源自于淘宝的 ConfigServer，十年来，随着蚂蚁集团的业务发展，注册中心架构已经演进至第五代。目前 SOFARegistry 不仅全面服务于蚂蚁集团的自有业务，还随着蚂蚁集团服务众多合作伙伴，同时也兼容开源生态。

产品特点

- 高可扩展性：采用分层架构、数据分片存储等方式，突破单机性能与容量瓶颈，接近理论上的“无限水平扩展”。经受过蚂蚁集团生产环境海量节点数与服务数的考验。
- 高时效性：借助 SOFABolt 通信框架，实现基于 TCP 长连接的节点判活与推模式的变更推送，服务上下线通知时效性在秒级以内。
- 高可用性：不同于 Zookeeper、Consul、etcd 等 CP（CAP 理论中的 CP，强调一致性）架构注册中心产品，SOFARegistry 针对服务发现的业务特点采用 AP 架构，最大限度地保证网络分区故障下注册中心的可用性。通过集群多副本等方式，应对自身节点故障。

产品架构

产品架构如下：



> 说明
>
> 图中 Session Cluster、Data Cluster、Meta Cluster 都是集群，表示 SessionServer、DataServer、MetaServer 可无限扩展。

- **Client（客户端）**：提供应用接入服务注册中心的基本 API 能力，应用系统依赖客户端 JAR 包，通过编程方式调用服务注册中心的服务订阅和服务发布能力。
- **SessionServer（会话服务器）**：提供客户端接入能力，接受客户端的服务发布及服务订阅请求，并作为一个中间层将发布数据转发至 DataServer 存储。SessionServer 可无限扩展以支持海量客户端连接。
- **DataServer（数据服务器）**：负责存储客户端发布数据，数据存储按照数据 ID 进行一致性 hash 分片存储，支持多副本备份，保证数据高可用。DataServer 可无限扩展以支持海量数据。
- **MetaServer（元数据服务器）**：负责维护集群 SessionServer 和 DataServer 的一致列表，在节点变更时及时通知集群内其他节点。MetaServer 通过 SOFAJRaft 保证高可用和一致性。

6.2. 基本原理

SOFARegistry 即服务注册中心。下面详细介绍 SOFARegistry 的原理。

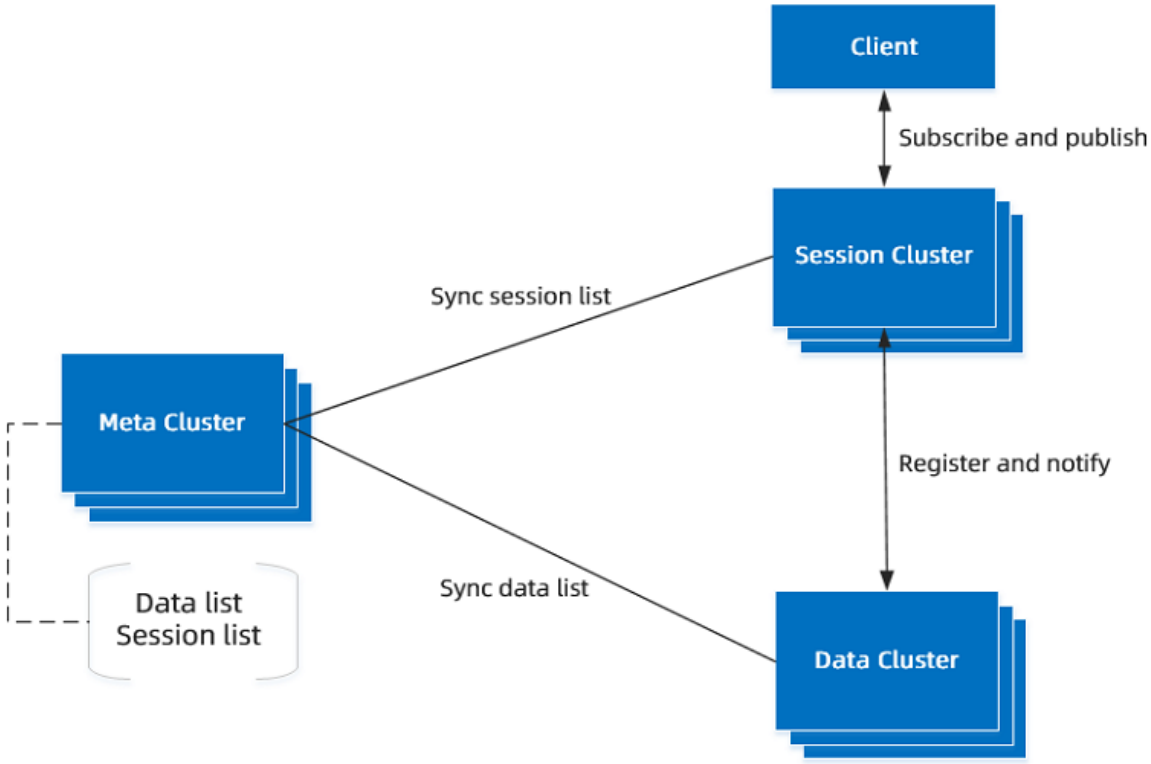
SOFARegistry 组成

SOFARegistry 即服务注册中心。其包含的 4 个组件及其职责为：

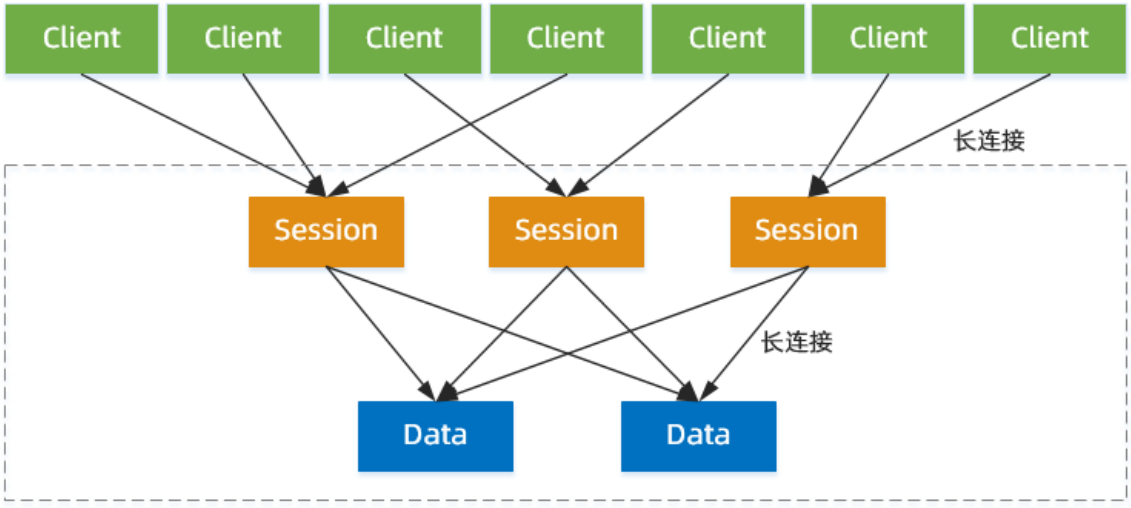
- 客户端（Client）：提供应用接入服务注册中心的基本 API 能力，可以是订阅方，也可以是发布方。
- 会话服务器（SessionServer）：主要是跟客户端建立连接，并作为一个中间层将发布数据转发至 DataServer 存储。
- 数据服务器（DataServer）：负责存储客服端发布的数据。支持多副本存储，保证高可用，并支持海量客户端。
- 元数据服务器（MetaServer）：主要维护集群的一致列表，保证高可用和一致性。

> 说明
>
> 引入这三个角色的目的为：
>
> - SessionServer 与 DataServer 的分离是为了同时突破连接数和存储的瓶颈。
> - 引入 MetaServer 是为了运维简单化。因为角色的多样化让服务注册中心有状态，使用 MetaServer 管理这些元数据，就无需引入第三方组件管理这些元数据，MetaServer 能够感知到 session 和 data 的状态，而无需 session 内部或者 data 内部自感知，让架构更加清晰。

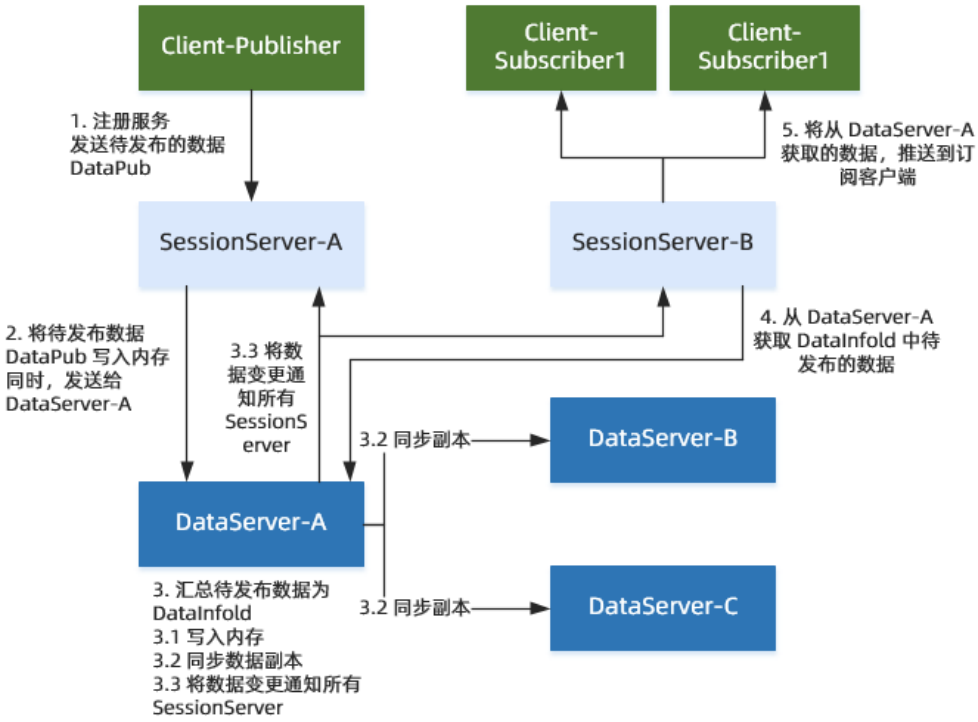
SOFARegistry 架构图



SOFARegistry 交互图



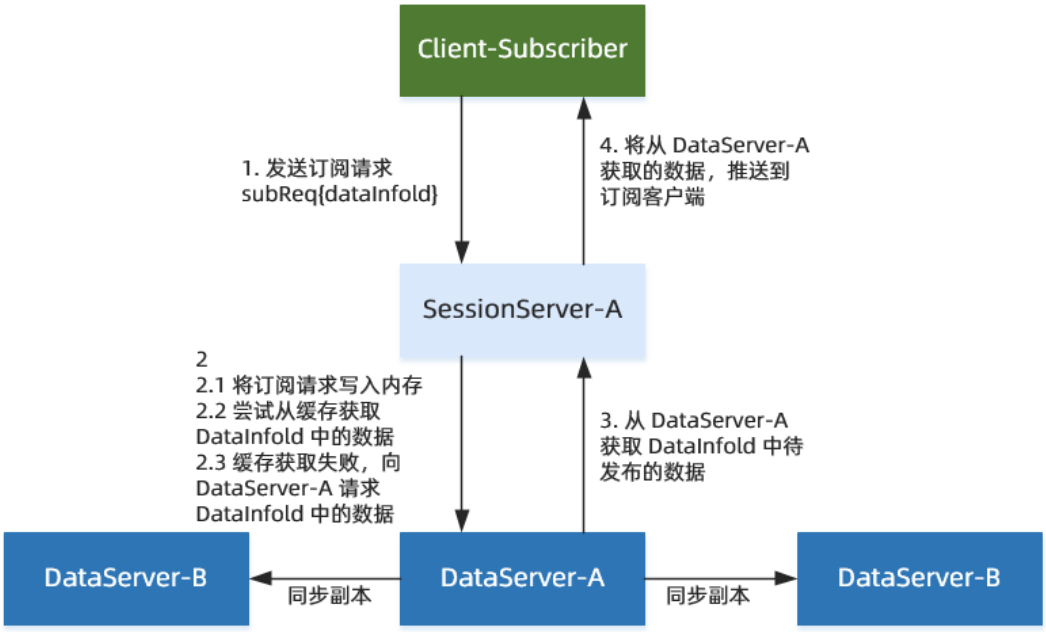
单次服务注册过程



单次服务注册的主要步骤为：

1. Client 发布端向 SessionServer 发送待发布数据 dataPub。
2. SessionServer 接收到 dataPub 数据后，进行下述操作：
   i. 首先写入内存：用于后续可以跟 DataServer 做定期检查。
   ii. 然后将数据 dataPub 发送给 DataServer。
3. DataServer 接收到 dataPub 数据后，进行下述操作：
   i. 将数据写入内存：DataServer 将从 SessionServer 收到的所有待发布数据汇总为 dataInfoId。
   ii. 将数据同步给副本：DataServer 在一致性 hash 分片的基础上，对每个分片保存了多个副本（默认是 3 个副本）。
   iii. 将数据变更事件通知给所有 SessionServer，事件内容是：
      - id：`<dataInfoId>`
      - 版本号信息：`<version>`
4. SessionServer 接收到变更事件通知后，对比 SessionServer 内存中存储的 dataInfoId 的 version，发现比 DataServer 发过来的小，所以主动向 DataServer 获取 dataInfoId 中待发布的数据，即获取具体的 dataPub 列表。
5. SessionServer 获取到 dataInfoId 中待发布的数据后，将数据推送给相应的 Client 订阅端，Client 订阅端就接收到这一次服务注册之后的最新待发布的 dataPub 列表数据。

单次服务订阅过程



单次服务订阅的主要步骤为：

1. Client 订阅端向 SessionServer 发起订阅请求 subReq。subReq 主要包含 dataInfoId，表示需要订阅哪个 dataInfoId 的数据。
2. SessionServer 接收到订阅请求 subReq 后，进行下述操作：
   i. 首先将订阅请求 subReq 写入内存：发送过来的 subReq 数据，SessionServer 都会存储到内存，用于实现数据变更推送的功能。
   ii. 然后尝试从缓存里获取相应 dataInfoId 中的待发布数据。根据缓存中是否有此数据，做出下一步行动：
      - 若无，则向 DataServer 发起请求，获取相应数据。
      - 若有，则直接返回相应数据。
3. SessionServer 从 DataServer 中获取到 dataInfoId 中的待发布数据。
4. SessionServer 将获取到的数据发送给 Client 订阅端。

6.3. 版本说明

SOFARegistry 版本及发布说明如下：

版本	发布时间	发布说明

版本	发布时间	发布说明
2.8.0	2021-07-14	**更新** 适配云游 1.4.0 版本。云游 1.4.0 版本对于 volume 挂载使用的是 PV，`/home/admin/logs` 目录权限被修改为 root 权限，需要应用在启动时，进行权限适配。 **修复** 修复服务端安全问题。
2.7.0	2021-04-27	**新增** 在 Antstack Plus 模式下，MetaServer 的 IP 会发生变化。为了支持双机房部署，增加 MetaServer 申请 SLB IP。 对于单机房也增加 `meta-slb` 的配置，为以后扩容为双机房做准备。
2.6.0	2021-04-06	**更新** MetaServer 启动时，增加 30s 睡眠时间，避免获取其他机器的域名失败。
2.4.0	2020-04-22	**新增** 支持 ARM 架构。 **说明** 该版本只适用于 ARM 机房。
2.3.4	2020-10-13	**修复** • 修复同一个 Client 超时重 pub 会导致 publisher 丢失的问题。 • 修复 SessionServer 内存在连接脏数据时不停打印错误日志的问题。 • 修复 RPC Provider 改名后 DSRConsole 新旧 appName 都存在的问题。 • 修复使用 OpenAPI 下线服务后，服务又在 DSRConsole 中出现的问题。

版本	发布时间	发布说明
2.3.2	2020-06-05	**修复** 修复 hessian 和 Java 安全问题。
2.3.0	2020-02-07	**新增** 新增多机房数据打通的能力，支持在 metaNodes 参数中配置多个机房的 MetaServer 地址列表。
2.2.2	2020-06-05	**修复** 修复 hessian 和 Java 安全问题。
2.2.1	2020-02-06	**修复** 修复 MetaServer 内存升高，导致系统内存占用 90% 的问题。
2.2.0		**新增** 合并 2.1.2 版本 MetaServer 内存持续增长问题。 **更新** • 删除 noguardregistry 一次性任务。 • 一次性任务兼容专有云和网商融合版本（自动发现 meta_server 和 registrymeta）。
2.1.2	2019-12-23	**修复** 修复 MetaServer 内存持续增长问题。
2.1.1	2019-12-19	**修复** • 修复 IDC 维度，订阅数据被清空问题。 • 修复在 scope.global 下 GR 隔离失效问题。 • 修复网商 registrysession 同步给 DSRConsole 的问题：syncPub 和 syncSub 的 TaskDispatcher 队列太小，导致 QueueOverflows。

版本	发布时间	发布说明
2.1.0	2019-10-16	**新增** 支持网商独占模式功能： • 增加同步 DSRConsole 独立开关，控制同步 DSRConsole。 • 增加默认 instanceid 定制能力，用于同一网商环境的默认 instanceid。 **更新** 数据同步功能性能增强。
1.14.1	2019-08-12	**更新** 升级一次性任务 metapush 和 metapushclose 的基础镜像，减少镜像体积。
1.14.0	2019-07-16	**更新** 更改内存计算规则脚本： • 操作系统不支持 `free -m` 时，将获取内存的脚本修改为 `containerinfo --totalmemory` 方式。 • 修改非必要的 cd 目录解压缩包脚本。
1.13.0	2019-06-26	**更新** metapush 和 metapushclose 升级基础镜像到 CentOS7。
1.12.0	2019-06-10	**更新** MetaServer、DataServer、SessionServer 的基础镜像升级到 CentOS7。 **修复** • 注册中心服务下线 API 导致的序列化问题。 • 修复 SyncClientsHeartbeatTask 在定期任务里做耗时操作的问题。 • 修复 watcher 进行 unregistry 操作时获取数据问题。
1.11.1	2019-05-28	**更新** 适配 AKE 2.0，将 MetaServer、DataServer、SessionServer 的自动重启策略改为“是”。
		新增 • 新增以下日志：

版本	发布时间	发布说明
1.11.0	2019-05-24	○ 增加推送数量日志。 ○ 每分钟打印 healthcheck 健康状态日志。 ○ 新增最近一分钟推空的数量日志。 ○ MetaServer 定期打印 nodeList。 ○ SessionServer 日志新增连接数信息。 ○ 每分钟打印推送状态日志。 • 新增以下监控： ○ MetaServer ■ 健康状态：不健康状态持续 5 分钟就报警。 ■ 推送状态：推送关闭超过 10 分钟就报警。 ■ metaNodeList、dataNodeList和 sessionNodeList：列表为空超过 10 分钟就报警。 ○ DataServer 健康状态：不健康状态持续 5 分钟就报警。 ○ SessionServer ■ 健康状态：不健康状态持续 5 分钟就报警。 ■ 推送状态：推送关闭超过 10 分钟就报警。 ■ 连接数监控：连接数连续 60 分钟为 0 就报警。 **更新** • JRaft 升级到 1.2.5 版本。 • 修改 DataServer 之间的重连保证。 • 修改 SOFA.CONFIG 类型数据启动期获取不到数据的默认处理方式。 • 删除同步 DSRConsole 失败重试记录的无效记录。 • SessionServer 的 console 日志改为1 分钟打印一次。 • 删除“node ipAddress:10.**.**.74 cannot be found on config list!”报错日志。 • 云游解决方案修改以下内容： ○ MetaServer 添加后置任务：关闭推送。 ○ 将 logs 目录单独挂载。 ○ 将 DataServer 和 SessionServer 配置在不同的物理机上。 ○ 修改 MetaServer 启动内存参数。 再次降低 xmx（2.3GB -> 1.8GB），并提取为云游参数，后续可针对不同环境修改配置。

版本	发布时间	发布说明
		◦ 云游增加环境变量：`REGISTRY_NODE_TYPE`，代表具体的角色，取值为 `META`、`DATA`、`SESSION`。镜像在启动后会根据该取值决定启动哪个角色。 **修复** • 修复 DataServer 从 working 状态变回 init 状态的 Bug。 • 修复 DataServer 无法达到 working 问题。 • 修复 DataServer 作为服务端，存储其他 DataServer 链接信息错误问题。 • DataServer 扩容或重启之前没有 working 时，将 clientoff 和其他节点同步写入数据进行延后处理。 • 修复 DataServer 定时重连 MetaServer 逻辑错误导致所有 DataServer 连接 MetaServer 失效，最后导致 SessionServer 无法接受新的 pub 请求问题。 • 修复 MetaServer 节点在启动初期有几率注册自身节点失败的问题。
1.8.4	2019-03-07	**新增** • 新增 AnyTunnelSLB。 • 在云游发布过程中，增加 HTTP 健康检测，确保启动成功。 **修复** 修复 MetaServer 关闭前自动打开推送的问题。
1.8.1	2019-03-01	**修复** 修复 SessionServer 在某些情况下存在脏数据引起 pub 被 discard 的问题。
1.8.0	2019-01-29	**更新** • 升级基础镜像。 • 降低 MetaServer 堆内存（2.3GB -> 2.2GB）。 • 同步 DSRConsole 主机名。
1.7.1	2019-01-09	**更新** • 延迟 session off 时间。 • MetaServer JVM 参数微调。 **修复** • 修复 session list 不断变更的 Bug。 • 修复 data unpub npe 的 Bug。

版本	发布时间	发布说明
1.5.3	2018-12-07	**新增** • MetaServer、DataServer、SessionServer 节点的健康检查提供RESTFul 接口查询健康情况。 • MetaServer 节点增加 update peer 接口。 **更新** 升级 JRaft 版本到 1.1.0。 **修复** 修复容器脚本 supervisor 参数配置导致多次启动的 BUG。
1.5.2	2018-12-04	**修复** 修复堆内存物理内存减少 500MB 再进行计算的问题。
1.5.1	2020-10-16	**更新** 修改 JVM 启动参数。
1.5.0	2018-11-23	**新增** 增加测试应用 registrytest 镜像。
1.4.0	2018-10-11	**新增** DataServer 增加 REST 接口，用户可以通过 REST 接口进行数据访问查询。 **修复** 修复 DataServer 部分数同步问题，增加线程池分配。

6.4. SOFARPC 使用 SOFARegistry

SOFARegistry，即服务注册中心，是 SOFA 中间件的底层组件。

主要存储的信息包括：

- 服务提供方的地址信息
- 服务消费方的订阅信息

主要工作机制为：

- 和服务消费方、服务提供方都建立长连接。
- 动态感知服务发布地址变更并通知消费方。

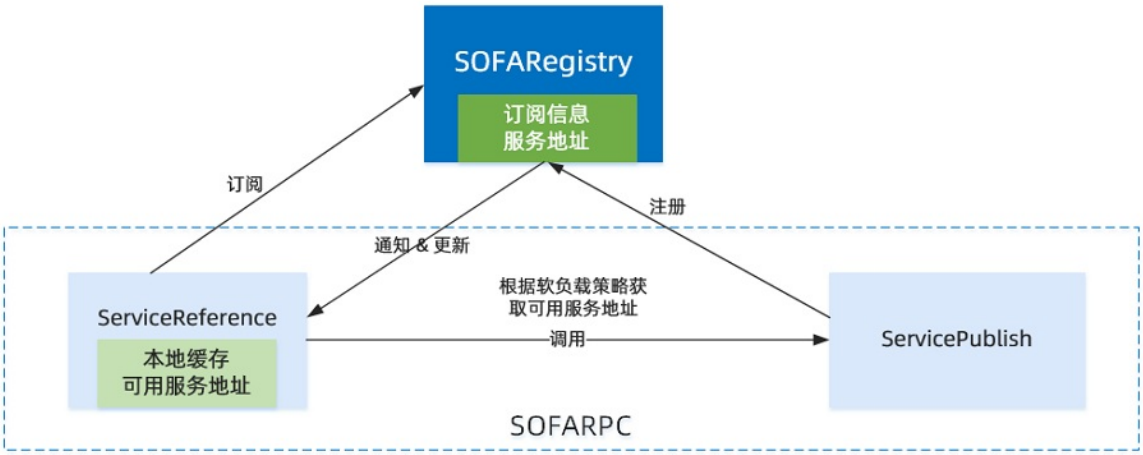
SOFARPC 在采用软负载路由策略时，需要使用 SOFARegistry（服务注册中心）。

软负载即软件负载，当需要调用服务时，消费方根据软负载策略，从 SOFARegistry 推送到本地缓存的列表里，选择一个地址，再调用该地址所提供的服务。

SOFARPC 采用服务发布（ServicePublish）和引用（ServiceReference）模型，通过 SOFARegistry（服务注册中心）动态感知服务发布并将服务地址列表推送给已经引用该服务的消费方，更新消费方本地缓存中的可用服务列表，最后通过软负载策略，为消费方选择可用地址进行远程通信。

在使用 SOFARPC 的时候，使用服务注册中心，可以将地址配置在配置文件中。

SOFARPC 使用 SOFARegistry 示意图



6.5. Spring Cloud 使用 SOFARegistry

本文介绍如何改造本地 Spring Cloud 工程，将其接入 SOFA 服务注册中心 SOFARegistry。

前置条件

在进行开发前，首先需要完成本地 Maven settings 中对 JAR 包仓库的配置，工程才可以通过 Maven 获取仓库里注册中心的 JAR 包。配置说明如下：

- 配置路径：
  - Windows 系统：`C:\Users\userName_XXX\.m2\settings.xml`
  - Mac or Linux 系统：`/Users/userName_XXX/.m2/settings.xml`

  > **重要**
  >
  > Linux 和 Mac OS 系统上，`.m2` 目录可能被隐藏。Mac OS 可以通过 `Command + Shift + .` 进行查看；Linux 可以通过 `Ctrl + H` 进行查看。

- 配置内容：

  i. 配置 `mvn.cloud.alipay.com` 仓库的地址及用户名密码。

  配置示例如下：

```
<servers>
    <server>
        <id>nexus-server@public</id>
        <username>${username}</username>
        <password>${password}</password>
    </server>
    <server>
        <id>nexus-server@public-snapshots</id>
        <username>${username}</username>
        <password>${password}</password>
    </server>
    <server>
        <id>mirror-all</id>
        <username>${username}</username>
        <password>${password}</password>
    </server>
</servers>
```

ii. 更新开发环境 mirror 配置（可选，本地不能正常下载 JAR 包时进行配置）。

如果本地 maven `settings.xml`（默认路径 `~/.m2/settings.xml`）配置 mirror 为 `*`，需要排除 `alipay-cloud-server@public` 仓库托管，示例如下：

```
<mirror>
    <id>nexus-aliyun</id>
    <!-- <mirrorOf>*</mirrorOf> -->
   <mirrorOf>*,!alipay-cloud-server@public</mirrorOf>
    <name>Nexus aliyun</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

`mirrorOf` 的配置 `*` 改成 `*,!alipay-cloud-server@public`，代表不需要托管仓库 `alipay-cloud-server@public`。

您也可以前往 脚手架控制台 下载已配置好的 `settings.xml`，并前往上述配置路径，覆盖原有 `settings.xml` 文件。

操作步骤

1. 在本地开发应用。

   i. 添加仓库地址。

   为保证 SDK 和对应 tracer 包能下载到，请在项目根 `pom.xml` 添加仓库地址：

   ```
   <repositories>
     <repository>
       <id>alipay-cloud-server@public</id>
       <url>http://mvn.cloud.alipay.com/nexus/content/groups/open</url>
       <snapshots>
         <enabled>false</enabled>
       </snapshots>
     </repository>
   </repositories>
   ```

   ii. 在工程根目录下的 `pom.xml` 中，引入 `sofa-registry-cloud-all` SDK 依赖。

   版本信息，请参见 SDK 版本说明。

   ```
   <dependency>
       <groupId>com.alipay.sofa</groupId>
       <artifactId>sofa-registry-cloud-all</artifactId>
       <!-- 替换 x.x.x 为该 SDK 最新版本号 -->
       <version>x.x.x</version>
   </dependency>
   ```

   > **说明**
   >
   > `sofa-registry-cloud-all` SDK 1.2.8 版本同时兼容 Spring Boot 1.x 和 Spring Boot 2.x。

   iii. 根据您的 Spring Cloud 版本信息，引入对应的 Tracer 依赖。

- Camden、Dalston 和 Edgware 版本（对应 Spring Boot 1.x 版本）

```
<!-- 支持服务调用tracer日志记录能力 -->
<!-- for Spring Boot 1.X -->
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
<version>2.3.7.JST.1</version>
<exclusions>
<exclusion>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
</exclusion>
<exclusion>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-dst-plugin</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
<version>1.0.17</version>
</dependency>
```

- Finchley、Greenwich 版本（对应 Spring Boot 2.x 版本）

```
<!-- 接入tracer -->
<!-- for Spring Boot 2.X -->
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
<version>3.2.3.JST.1</version>
<exclusions>
<exclusion>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
</exclusion>
<exclusion>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-dst-plugin</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
<version>1.0.17</version>
</dependency>
```

iv. 通过以下任一方式，添加应用启动参数。

> **重要**
>
> 如您即将使用的应用服务发布平台不是 SOFAStack 自提供的容器应用服务或经典应用服务，则无需添加以下参数配置。

- 在应用 yml 文件中指定参数：

```
sofa:
registry:
  discovery:
        instanceId:          //当前登录账号的实例 ID。
        antcloudVip:       //当前地域的 AntVIP 地址值。
        accessKey:           //当前登录账号的 AccessKey ID。
        secretKey:           //当前登录账号的 AccessKey Secret。
```

- 指定 JVM 启动参数值：

```
-Dcom.alipay.instanceid=                  //当前登录账号的实例 ID。
-Dcom.antcloud.antvip.endpoint=    //当前地域的 AntVIP 地址值。
-Dcom.antcloud.mw.access=             //当前登录账号的 AccessKey ID。
-Dcom.antcloud.mw.secret=             //当前登录账号的 AccessKey Secret。
```

- 指定系统环境变量：

```
SOFA_INSTANCE_ID=                    //当前登录账号的实例 ID。
SOFA_ANTVIP_ENDPOINT=          //当前地域的 AntVIP 地址值。
SOFA_SECRET_KEY=                     //当前登录账号的 AccessKey ID。
SOFA_ACCESS_KEY=                    //当前登录账号的 AccessKey Secret。
```

> **说明**
>
> 以上参数值是中间件的全局配置项，可在 脚手架控制台 获取。详情请参见 配置说明。

v. 编写业务代码。

2. 发布应用。

i. 登录 SOFAStack 控制台。

ii. 在左侧导航栏选择 **运维管理 > 经典应用服务 > 应用发布 > 发布单**，然后创建并执行发布单。

具体操作，请参见 新建并执行发布单。应用发布后，您可以在左侧导航栏选择 **中间件 > 微服务平台 > 微服务 > 服务管控**，然后在 **服务管控** 页面查看发布的服务。

> **说明**
>
> 如您需要在其他发布平台（非 SOFAStack 容器应用服务或经典应用服务）发布部署应用，请参考对应发布平台的帮助文档。

查看日志

您可以在 `/home/admin/logs/registry` 中查看注册中心的日志。

> **说明**
>
> 如果您使用的是旧版客户端运行模式，请在 `/home/admin/logs/confreg` 中查看。

6.6. Dubbo 使用 SOFARegistry

本文介绍如何改造本地 Dubbo 工程，将其接入 SOFA 服务注册中心 SOFARegistry。

前置条件

在进行开发前，首先需要完成本地 Maven settings 中对 JAR 包仓库的配置，工程才可以通过 Maven 获取仓库里注册中心的 JAR 包。配置说明如下：

- 配置路径：
  - Windows 系统：`C:\Users\userName_XXX\.m2\settings.xml`

- Mac or Linux 系统：`/Users/userName_XXX/.m2/settings.xml`

> **重要**
>
> Linux 和 Mac OS 系统上，`.m2` 目录可能被隐藏。Mac OS 可以通过 `Command + Shift + .` 进行查看；Linux 可以通过 `Ctrl + H` 进行查看。

- 配置内容：

i. 配置 `mvn.cloud.alipay.com` 仓库的地址及用户名密码。

配置示例如下：

```
<servers>
    <server>
        <id>nexus-server@public</id>
        <username>${username}</username>
        <password>${password}</password>
    </server>
    <server>
        <id>nexus-server@public-snapshots</id>
        <username>${username}</username>
        <password>${password}</password>
    </server>
    <server>
        <id>mirror-all</id>
        <username>${username}</username>
        <password>${password}</password>
    </server>
</servers>
```

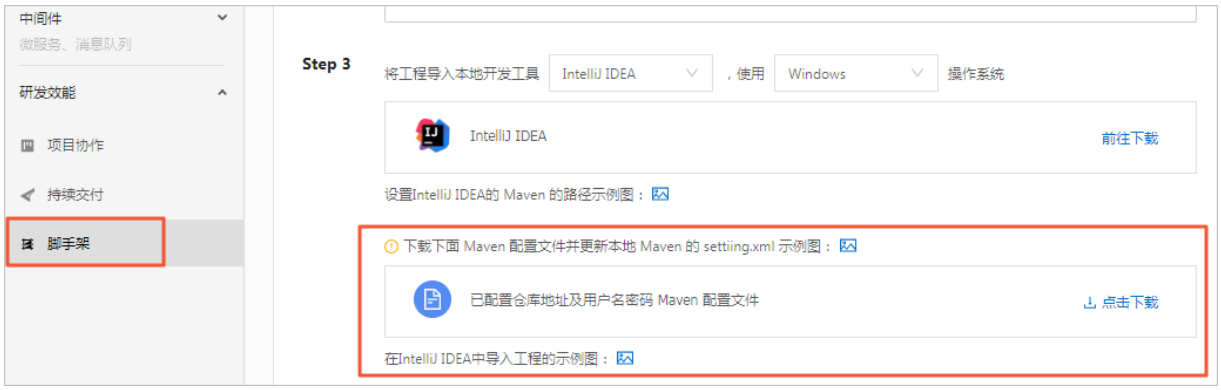
ii. 更新开发环境 mirror 配置（可选，本地不能正常下载 JAR 包时进行配置）。

如果本地 maven `settings.xml`（默认路径 `~/.m2/settings.xml`）配置 mirror 为 `*`，需要排除 `alipay-cloud-server@public` 仓库托管，示例如下：

```
<mirror>
    <id>nexus-aliyun</id>
    <!-- <mirrorOf>*</mirrorOf> -->
   <mirrorOf>*,!alipay-cloud-server@public</mirrorOf>
    <name>Nexus aliyun</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

`mirrorOf` 的配置 `*` 改成 `*,!alipay-cloud-server@public`，代表不需要托管仓库 `alipay-cloud-server@public` 。

您也可以前往 脚手架控制台 下载已配置好的 `settings.xml`，并前往上述配置路径，覆盖原有 `settings.xml` 文件。



操作步骤

1. 在本地开发应用。

   i. 添加仓库地址。

   为保证 SDK 和对应 tracer 包能下载到，请在项目根 `pom.xml` 添加仓库地址：

```
<repositories>
  <repository>
    <id>alipay-cloud-server@public</id>
    <url>http://mvn.cloud.alipay.com/nexus/content/groups/open</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

   ii. 在工程根目录下的 `pom.xml` 中，引入 `sofa-registry-cloud-all` SDK 依赖。

   版本信息，请参见 SDK 版本说明。

```
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>sofa-registry-cloud-all</artifactId>
    <!-- 替换 x.x.x 为该 SDK 最新版本号 -->
    <version>x.x.x</version>
</dependency>
```

> 说明
>
> - `sofa-registry-cloud-all` SDK 1.2.8 版本兼容目前 Dubbo 应用的所有版本。
> - 如果是通过 `sofa-registry-dubbo-all` 引入的 SDK 依赖，可直接升级为 `sofa-registry-cloud-all` 依赖。

iii. 引入 Tracer 依赖。

根据 Dubbo 应用框架版本的不同，Tracer 提供下述 3 种接入方式，请根据实际业务需求选择合适方式。

- Spring Boot 1.x

```
<!-- for Spring Boot 1.X -->
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
<version>2.3.7.JST.1</version>
<exclusions>
<exclusion>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
</exclusion>
<exclusion>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-dst-plugin</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
<version>1.0.17</version>
</dependency>
```

- Spring Boot 2.x

```
<!-- for Spring Boot 2.X -->
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
<version>3.2.3.JST.1</version>
<exclusions>
<exclusion>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
</exclusion>
<exclusion>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-dst-plugin</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>com.alipay.sofa.common</groupId>
<artifactId>sofa-common-tools</artifactId>
<version>1.0.17</version>
</dependency>
```

- 非 Spring Boot 类型的 Dubbo 应用。
  a. 引入上文 Spring boot 2.x 相同依赖。
  b. 在代码中， Main 启动入口第一行加入以下开关：

```
SofaTracerConfiguration.setProperty(SofaTracerConfiguration.JSON_FORMAT_OUTPUT,"false");
```

iv. 配置 Dubbo 的注册中心。

使用 dsr: `<dubbo:registry address="dsr://dsr"/>` 。

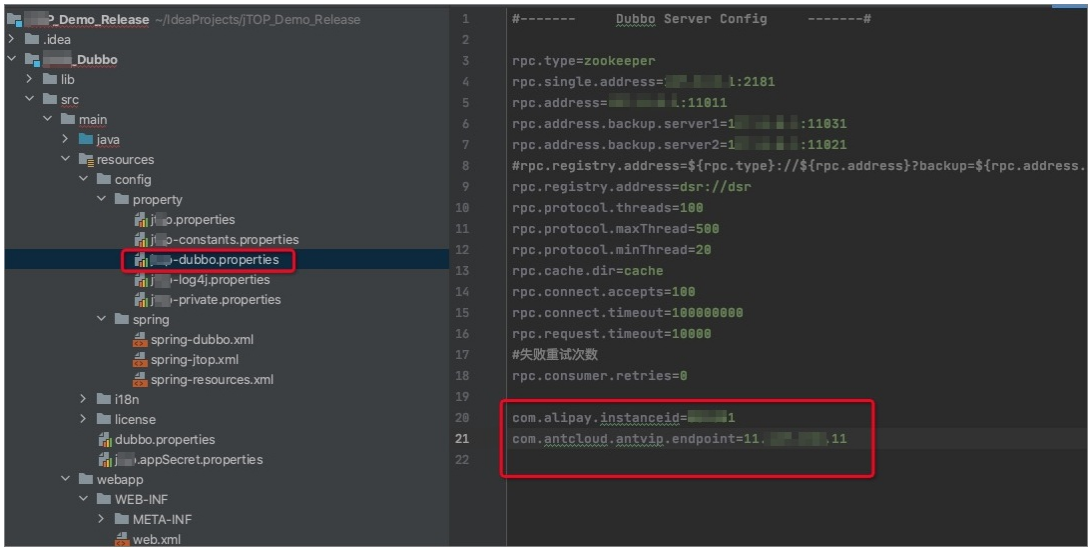
v. 添加应用启动参数。

您可以通过以下任一方式添加：

- 在 `dubbo.properties` 中配置下述属性：

> **说明**
>
> 默认路径为 `classpath` 根目录下的 `dubbo.properties`，您也可以根据需求使用 JVM 参数 `-Ddubbo.properties.file=xxx.properties` 手动指定路径。

```
com.alipay.instanceid=                    //当前登录账号的实例 ID。
com.antcloud.antvip.endpoint=     //当前地域的 AntVIP 地址值。
com.antcloud.mw.access=                //当前登录账号的 AccessKey ID。
com.antcloud.mw.secret=                 //当前登录账号的 AccessKey Secret。
```



- 指定 JVM 启动参数值：

```
-Dcom.alipay.instanceid=                    //当前登录账号的实例 ID。
-Dcom.antcloud.antvip.endpoint=     //当前地域的 AntVIP 地址值。
-Dcom.antcloud.mw.access=                //当前登录账号的 AccessKey ID。
-Dcom.antcloud.mw.secret=                 //当前登录账号的 AccessKey Secret。
```

- 指定系统环境变量：

```
SOFA_INSTANCE_ID=                    //当前登录账号的实例 ID。
SOFA_ANTVIP_ENDPOINT=          //当前地域的 AntVIP 地址值。
SOFA_SECRET_KEY=                    //当前登录账号的 AccessKey ID。
SOFA_ACCESS_KEY=                    //当前登录账号的 AccessKey Secret。
```

> **说明**
>
> 以上参数值是中间件的全局配置项，可在 脚手架控制台 获取。详情请参见 配置说明。

2. 发布应用。

i. 登录 SOFAStack 控制台。

ii. 在左侧导航栏选择 **运维管理 > 经典应用服务 > 应用发布 > 发布单**，然后创建并执行发布单。

具体操作，请参见 新建并执行发布单。应用发布后，您可以在左侧导航栏选择 **中间件 > 微服务平台 > 微服务 > 服务管控**，然后在 **服务管控** 页面查看发布的服务。

查看日志

您可以在 `/home/admin/logs/registry` 中查看注册中心的日志。

> **说明**
>
> 如果您使用的是旧版客户端运行模式，请在 `/home/admin/logs/confreg` 中查看。

6.7. 服务网格使用 SOFARegistry

使用服务网格时，您需要在本地代码中完成服务注册，并将本地的工程项目接入 SOFA 服务注册中心（即 SOFARegistry）。

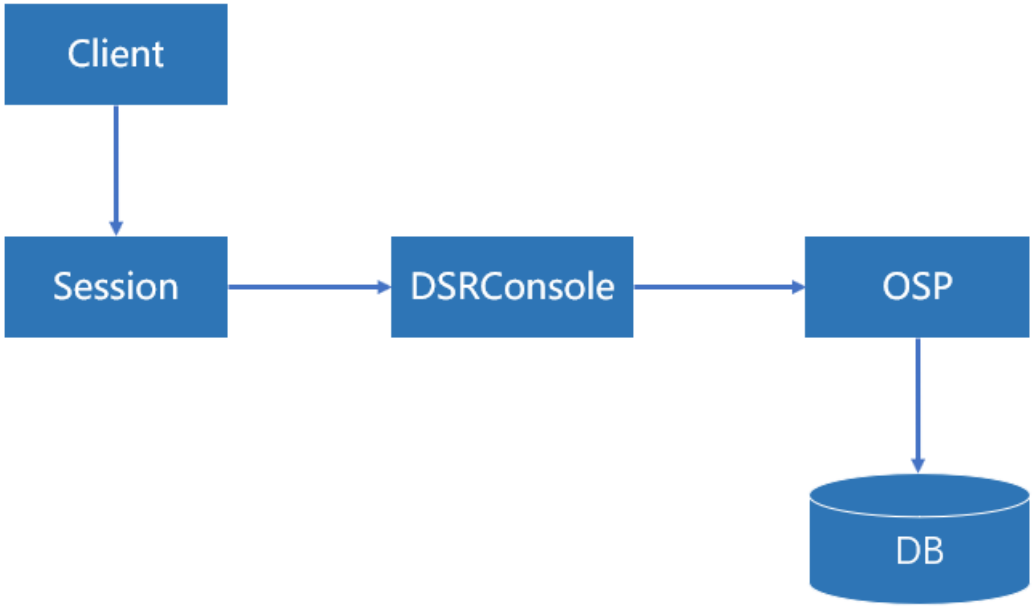
详细的操作步骤，请参见 连接 SOFA 服务注册中心。

6.8. 注册中心鉴权

注册中心鉴权在专有云环境默认关闭，本文介绍如何开启鉴权。

调用链路

注册中心鉴权的调用链路如下图所示：



注意事项

- 开启鉴权后，会对非 000001 的所有客户生效。您在开启鉴权之前，需注意鉴权对生产环境可能产生的影响，如果出现问题，需尽快回滚。
- 开启鉴权后，客户端和注册中心 Session 服务器的时差绝对值不能超过 6 分钟。Session 服务器会进行时间校对，对于时间戳超过 6 分钟的请求，会被认为是无效请求，导致鉴权失败。
- 鉴权通过后，客户端会产生如下日志：

```
m=HmacSHA256, !Timestamp=1640762700000}}, RegisterResponse{success=true, registId='8ee3c2dc-5249-44a7-b68b-73d968201879', version=0, refused=false, message='ConfiguratorRegister register success!'}
```

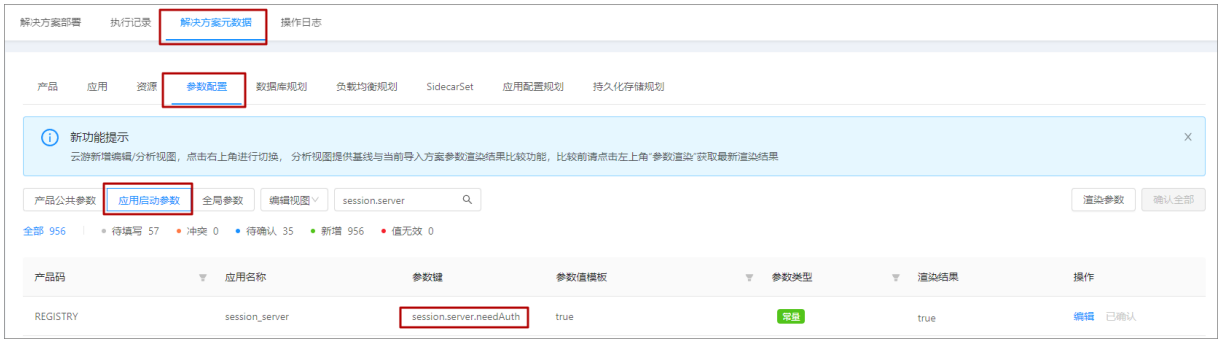
如果鉴权失败，则会出现 refuse=true 的日志。

服务端开启鉴权

注册中心鉴权在专有云环境默认关闭，如果您想开启鉴权，需配置以下组件参数，开启鉴权。

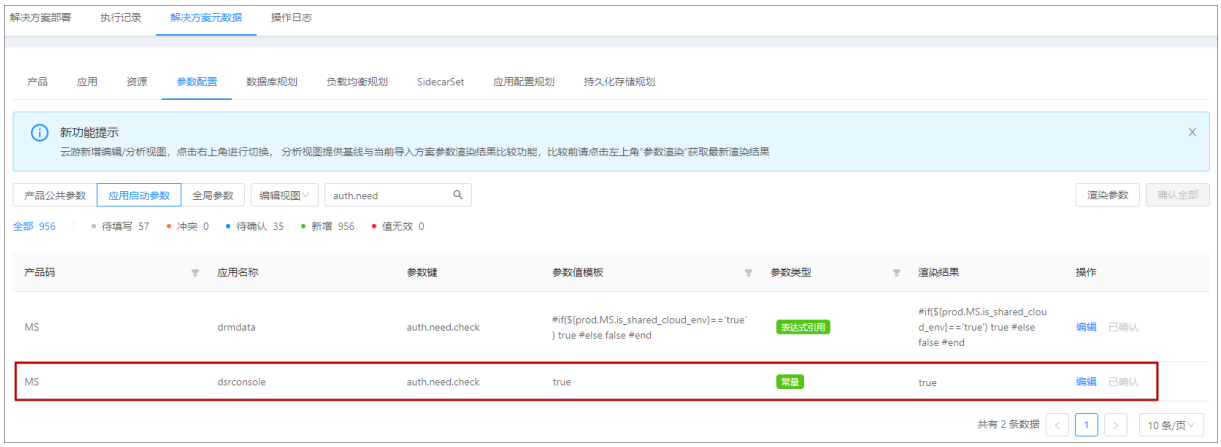
开启 Session 服务器的鉴权功能

您需要在云游 Local 发布注册中心产品前，将启动参数中的 `session.server.needAuth` 的参数值修改为 `true` 。



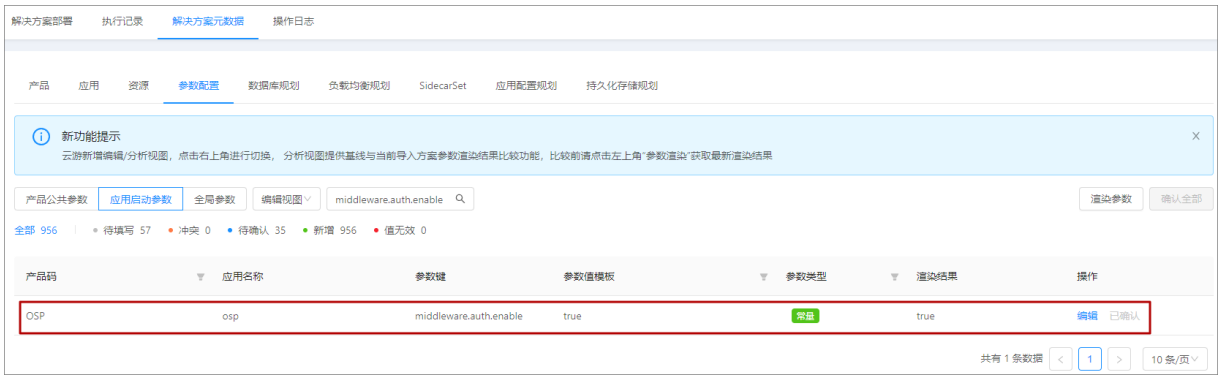
开启 DSRConsole 的鉴权功能

您需要在云游 Local 发布微服务产品前，将启动参数中的 `auth.need.check` 的参数值修改为 `true` 。



开启 OSP 的鉴权功能

您需要在云游 Local 发布 OSP 产品前，将启动参数中的 `middleware.auth.enable` 的参数值修改为 `true` 。



客户端配置认证参数

您可以通过以下任意方式，添加应用启动参数。

- 在应用 YML 文件中指定参数

```
sofa:
registry:
  discovery:
        instanceId:          //当前登录账号的实例 ID。
        antcloudVip:         //当前地域的 AntVIP 地址值。
        accessKey:           //当前登录账号的 AccessKey ID。
        secretKey:           //当前登录账号的 AccessKey Secret。
```

- 指定 JVM 启动参数值

```
-Dcom.alipay.instanceid=                  //当前登录账号的实例 ID。
-Dcom.antcloud.antvip.endpoint=           //当前地域的 AntVIP 地址值。
-Dcom.antcloud.mw.access=                 //当前登录账号的 AccessKey ID。
-Dcom.antcloud.mw.secret=                 //当前登录账号的 AccessKey Secret。
```

- 指定系统环境变量

```
SOFA_INSTANCE_ID=                    //当前登录账号的实例 ID。
SOFA_ANTVIP_ENDPOINT=                //当前地域的 AntVIP 地址值。
SOFA_SECRET_KEY=                     //当前登录账号的 AccessKey ID。
SOFA_ACCESS_KEY=                     //当前登录账号的 AccessKey Secret。
```

获取 AccessKey ID 和 AccessKey Secret

1. 在 OSP 上创建账号。

   在 OSP 服务器上运行以下命令创建账号：

   ```
   curl -H "Content-Type: application/json" -XPOST
   'http://127.0.0.1:8341/api/accesskey/create' \
   -d '{
       "name": "test",
       "instanceId": "WTTEOCEHCTYV"
   }'
   ```

2. 获取账号的 AccessKey ID 和 AccessKey Secret。

   您可以查看 OSP 的 `t_access_key` 表，获取已有账户的 AccessKey ID 和 AccessKey Secret。

6.9. 问题排查

6.9.1. 常见问题

本文汇总了 SOFARegistry 使用过程中的一些常见问题及对应的解决方案。

RPC 服务端发布之后，为什么在微服务控制台无法找到该服务。

问题原因

应用服务器的 IP 地址，不在发布部署参数 `rpc_enabled_ip_range` 范围内。例如：应用服务器的 IP 是 172.19.**.**，而 `rpc_enabled_ip_range` 配置的参数为 `10:11,172.16,192.168` 。

解决方案

修改应用实例的发布部署参数 `rpc_enabled_ip_range` ，将应用服务器的 IP 地址包含在 `rpc_enabled_ip_range` 的配置范围内。如上述示例中，将 `10:11,172.16,192.168` 修改为 `10:11,172.19,192.168` ，然后重新发布应用。

服务消费者调用服务提供者的策略是什么？

默认是随机调用，如有更精细的需求，请参见 负载均衡。

在 Registry 宕机后，如何避免因为服务提供者不健康而出现调用失败的问题？

无论 SOFARegistry 是否宕机，SOFARPC 框架对服务提供者调用失败的情况，做了两方面的容错处理：

- 调用重试：您可以配置调用重试，当发生非业务错误导致的请求失败时（例如网络超时等），会尝试重试（随机调用一个服务提供者重试）。配置方式请参见 调用重试。

> **重要**
>
> 打开重试机制时，需要业务方保证幂等。

- 单机剔除：通过重试基本能保证调用成功，但会增加相应延迟。若您想要进一步提高调用成功率，可以配置单机剔除。当一台服务提供者长时间无法提供服务，可以降低它的权重。配置方式请参见 自动故障剔除。

服务消费者缓存保存时间是多久？

消费者不重启，会一直保存。

是否只有 Registry 扩缩容才会涉及 ACVIP 配置变更？

只有注册中心的 SessionServer 扩缩容，才需要变更 ACVIP 的配置。ACVIP上已经有的数据不会自动变更，您可以通过 OSP 刷新已经有的数据，或进入 cloudinc 手动配置。

如何确认微服务与注册中心是否连接正常？

您可以在项目服务器上通过 `ps -ef | grep 9600` 命令检查 9600 端口，如果端口为监听状态，则表示连接正常。

服务下线的 OpenAPI 调用失败，出现“your product does not have any configured rest url”报错

这个问题是 Registry 1.11.0 的版本问题，已在 1.12.0 及以上版本修复。若您遇到此问题，建议升级版本。

6.9.2. ACVIP 问题排查

ACVIP 部署问题

前置条件

解决 ACVIP 部署问题时，请先按照以下内容确认您的环境：

- 数据库时间和 ACVIP 机器时间需要一致，否则无法部署完成。
  - 查看机器时间

    命令如下：

    ```
    date -R
    ```

  - 查看数据库时间

    a. 获取 DB 连接。

    ```
    env | grep data
    ```

    b. 连接数据库。

    ```
    mysql -h<hostname> -u<username> -p<passwd> -P <port> -D <DB 链接>
    ```

    c. 查看数据库时间。

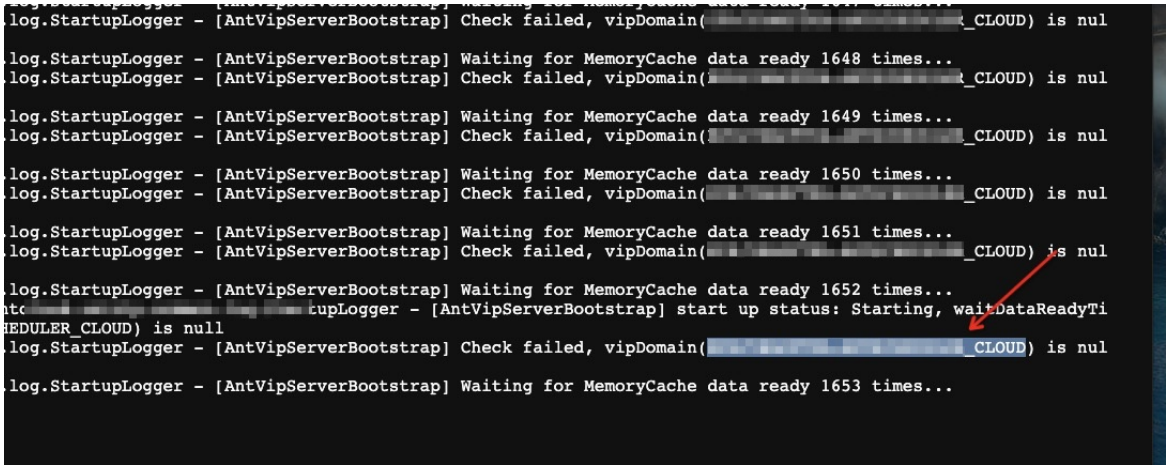
    ```
    select now()
    ```

- 数据库版本不能为 MySQL 8 。
- `/etc/hosts` 文件中需要有 IPv4 地址和 hostname 的映射配置。

ACVIP 部署不到终态

排查步骤如下：

1. 查看启动日志 `/home/admin/logs/acvip/Startup.log` 。



   图中日志说明 domain 加载失败，需要确认发布失败的原因。

2. 查询 `antvip_vip_domain` 。

   查询命令如下：

```
select * from antvip_vip_domain
```

```
                domain_type: NORMAL
*************************** 4. row ***************************
                         id: 4
                       name: [redacted]_CLOUD
                        app: NULL
                        env: PROD
                    station: MAIN_SITE
                       zone: NULL
          protect_threshold: 50
                     enable: 1
          health_check_type: TCP
health_check_default_port: 9600
       health_check_timeout: 2000
      health_check_interval: 5000
         health_check_raise: 1
          health_check_fall: 3
       health_check_payload:
                    version: 13
                  gmt_ceate: 2020-11-13 11:00:58
                gmt_modified: 2021-02-24 15:05:44
                      owner: osp
        health_check_enable: 1
                domain_type: NORMAL
*************************** 5. row ***************************
                         id: 5
```

查找是否有步骤 1 中加载失败的域名，且 enable 是否为 1，为 1 则继续下面的步骤。

3. 查询是否存在有效节点。

   查询命令如下：

   ```
   select * from antvip_real_node where domain_id=4
   ```

```
MySQL [cloudinc]> select * from antvip_real_node where domain_id=4;
+-------+-------------+---------------+---------------+------+------+--------+-----------------+--------+---------------------+---------------------+-----------+-------------+
| id    | domain_name | ip            | fqdn          | env  | zone | weight | health_check_port | enable | gmt_create          | gmt_modified        | domain_id | data_center |
+-------+-------------+---------------+---------------+------+------+--------+-----------------+--------+---------------------+---------------------+-----------+-------------+
| 13001 | NULL        | 10.[redacted].111   | DSR_HTTP_CLOUD | PROD | ENV  |      1 |            9600 |      0 | 2020-12-25 17:36:38 | 2021-02-24 15:05:44 |         4 | NULL        |
| 13002 | NULL        | 10.[redacted].40    | DSR_HTTP_CLOUD | PROD | ENV  |      1 |            9600 |      0 | 2020-12-25 17:36:49 | 2021-02-24 15:05:44 |         4 | NULL        |
| 13005 | NULL        | 10.[redacted]2.216 | DSR_HTTP_CLOUD | PROD | ENV  |      1 |            9600 |      1 | 2021-02-24 14:58:15 | 2021-02-24 15:05:16 |         4 | NULL        |
| 13006 | NULL        | 10.[redacted]2.217 | DSR_HTTP_CLOUD | PROD | ENV  |      1 |            9600 |      1 | 2021-02-24 15:05:34 | 2021-02-24 15:05:34 |         4 | NULL        |
+-------+-------------+---------------+---------------+------+------+--------+-----------------+--------+---------------------+---------------------+-----------+-------------+
4 rows in set (0.00 sec)
```

确认节点是否存活，如果有不正常的节点，进行处理。如果都正常，继续进行下面的步骤。因为 ACVIP 有多个节点时，会有数据分组，这个 domain 可能是其他的 ACVIP 在进行加载。

4. 查询其他 ACVIP 节点。

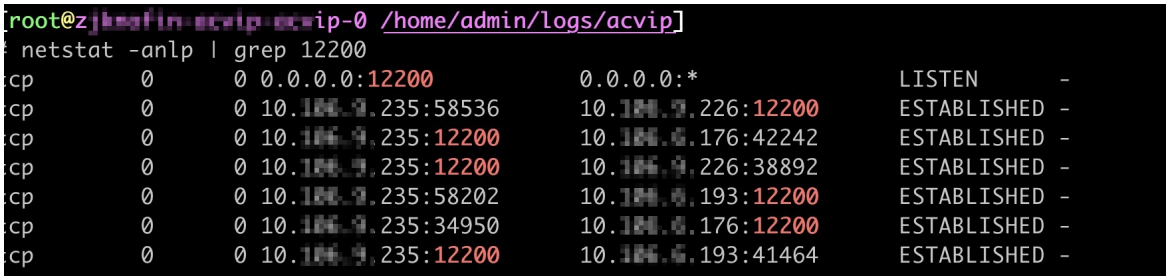
   查询命令如下：

   ```
   select * from antvip_vip_server
   ```

心跳时间（last_heartbeat）和当前时间相近的是活跃节点，需要测试当前服务器和这些服务器是否建立连接。

5. 确认网络连通性。
   命令如下：

```
netstat -anlp | grep 12200
```



查看 ACVIP 的 12200 端口是否有正常的通讯，有则表示网络正常，建议联系售后技术支持进行排查；如果没有通讯，则说明网络出现故障，建议检查网络问题。

使用问题

获取不到域名信息如何处理？

使用 `curl` 命令尝试获取域名信息，获取命令如下：

```
curl -i -XPOST {acvip地址}:9003/antcloud/antvip/instances/get -d '{"vipDomainName2ChecksumMap":{"000001-DSR_CLOUD":"N"}}'
```

长连接 Long Polling 命令示例如下：

```
curl -i -XPOST localhost:9003/antcloud/antvip/instances/get -d '{"vipDomainName2ChecksumMap":{"000001-DSR_CLOUD":"N"},"allowPolling":true}'
```

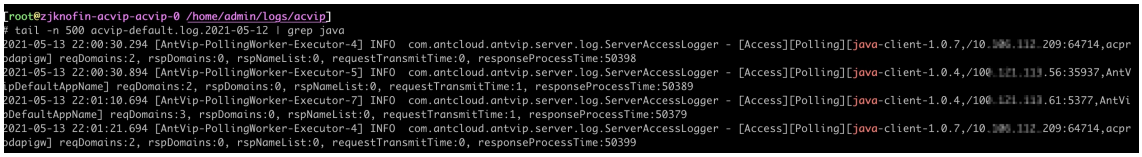
- 查询到结果，说明 ACVIP 服务正常，请检查本地 ACVIP 配置。
  i. 执行 `cd /home/admin/conf/acvip-java-client-cache/domains/` 命令，查看本地缓存中是否有目标域名信息，并查看获取的时间信息。
  ii. 如果获取信息失败，查看 `api-stat.log` 是否有报错。

```
cd /home/admin/logs/acvip-java-client
cat api-stat.log
```

iii. 根据查询的报错信息进行处理。

您可以查询指定时段的 JAVA 客户端拉取记录，命令如下：

```
tail -n 500 acvip-default.log.2021-05-12 | grep java
```



日志格式如下：

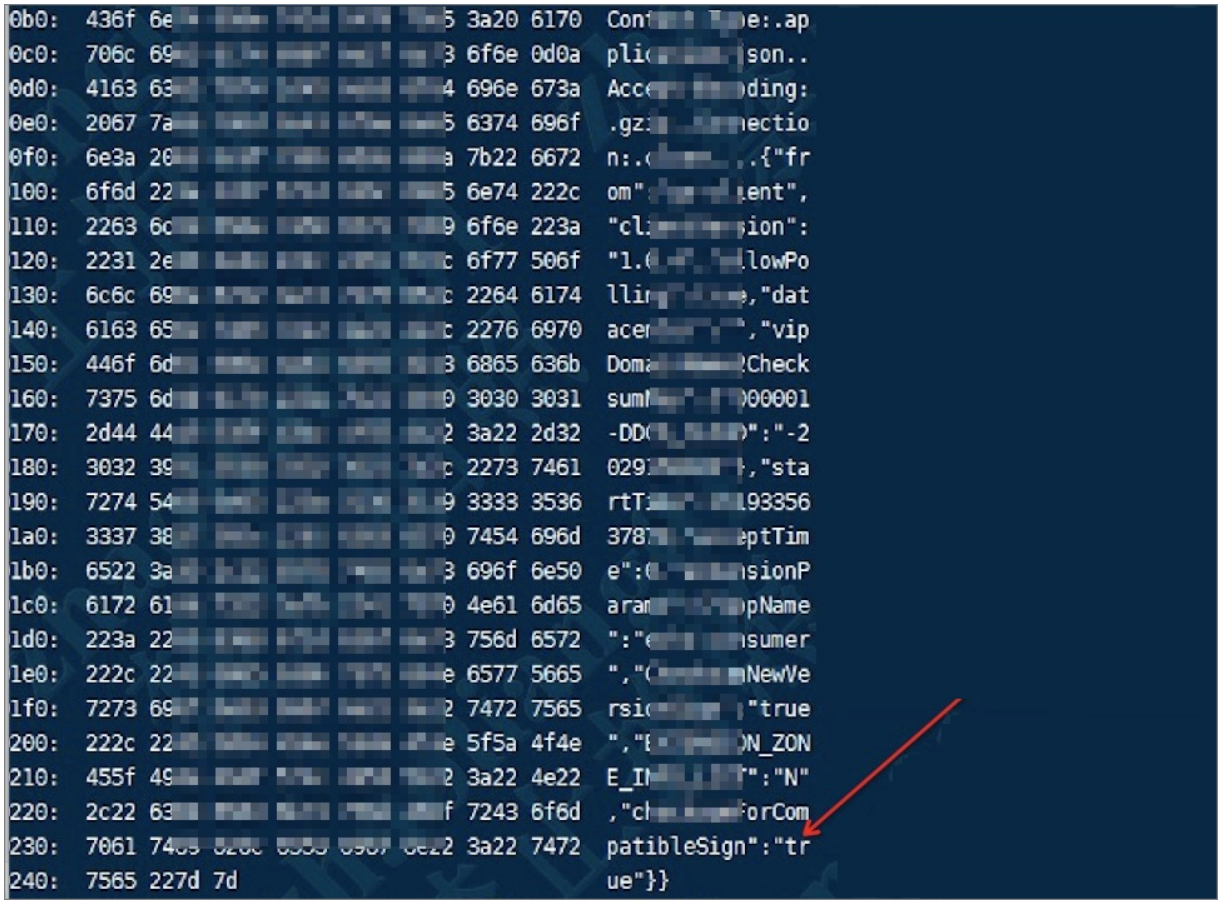
```
    this.info(
        msgFormat: "[Access][Polling][%s,%s,%s] reqDomains:%s, rspDomains:%s, rspNameList:%s, requestTransmitTime:%s, responseProcessTime:%s",
        from, address, appName, reqDomains, rspDomains, rspNameList, transmitTime, processTime);
}

/**
```

- 如果查询到的结果是空，需要到 ACVIP 服务端增加域名配置。
- 如果网络报错，确认 ACVIP 服务器地址配置是否正确。

如何查看客户端的请求内容？

您可以在 ACVIP 上执行如下命令查看：

```
tcpdump port 9003 and host <对端 IP> -nnvvXS
```



6.9.3. 注册中心问题排查

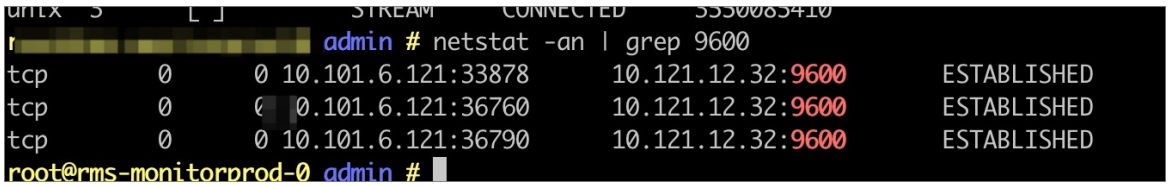
部署

1. 删除 MetaServer `/home/admin/raftData/` 路径下面的数据，按照顺序重新发布 MetaServer-->DataServer-->SessionServer。
2. SessionServer 发布后，Pod 的节点 IP 会发生变化，需要检查 ACVIP 上的配置是否正确。

使用

1. 查看服务器是否连接到注册中心。

   在有问题的机器上，执行 `netstat -anlp | grep 9600` 命令。



2. 未连接到注册中心，查看 ACVIP 配置。

   执行 `cd /home/admin/conf/acvip-java-client-cache/domains/` 命令，然后找到 `*-DSR_CLOUD.json` 和 `*-DSR_HTTP_CLOUD.json` 文件，查看里面的配置，是否拉到注册中心的信息。

3. 服务器未拉到注册中心，查看 ACVIP 配置信息。

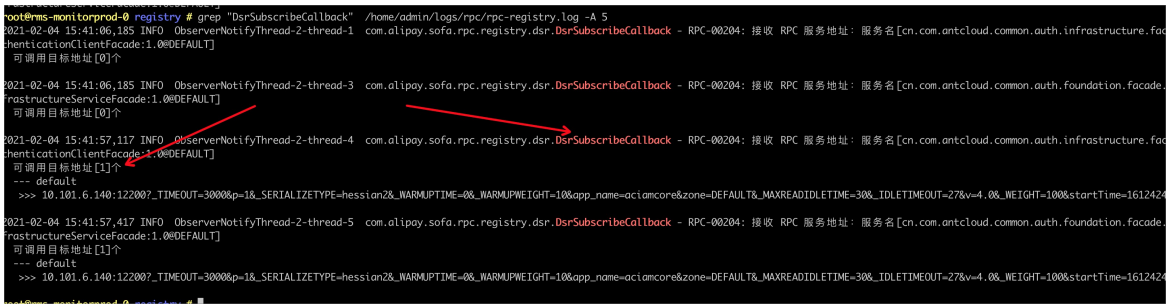
   执行 `cd /home/admin/logs/acvip-java-client` 命令查看 `api-stat.log` 文件是否有报错。

4. 链接到注册中心之后，查看拉取信息。

   sub 端执行命令如下：

   ```
   grep "DsrSubscribeCallback"  /home/admin/logs/rpc/rpc-registry.log
   ```

   出现类似下图信息，说明订阅成功。



   confreg 端执行 `grep 接口名 /home/admin/logs/rpc/rpc-registry.log -A 5` 命令。

```
--
2021-03-02 19:20:41,754 INFO  worker-3-thread-6                RPC-REGISTRY - Re
ceive RPC service addresses: service[com.alipay.ap.prodcenter.common.service.fac
ade.query.api.contract.ArrangementQueryFacade:1.0@DEFAULT]
  usable target addresses[1]
  >>> 10.1.86.116:12200?_TIMEOUT=3000&p=1&_SERIALIZETYPE=4&_WARMUPTIME=0&_WARMUP
WEIGHT=10&app_name=approdcenter&zone=CELLA&_MAXREADIDLETIME=30&_IDLETIMEOUT=27&v
=4.0&_WEIGHT=100&startTime=1611911393956
```

5. 业务重启后失败，注册中心没有报错，查看 SessionServer 是否关闭推送。

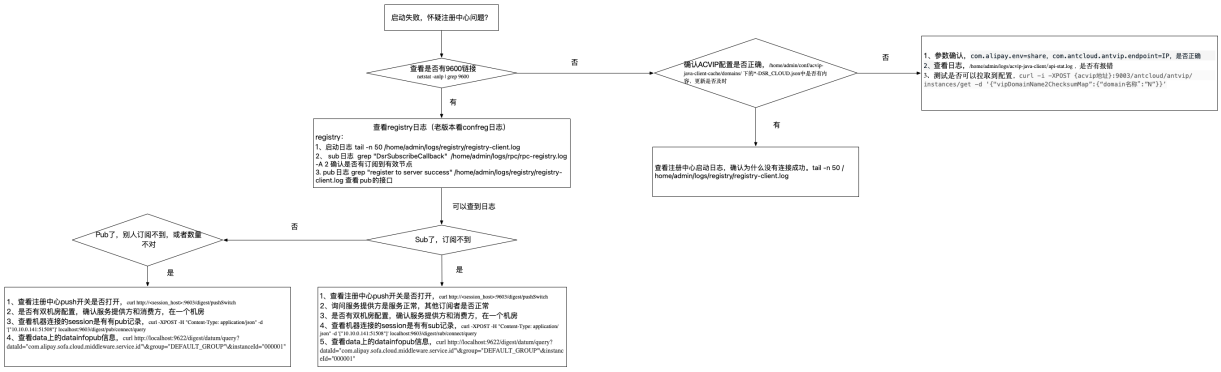
   执行命令如下：

   ```
   curl http://localhost:9603/digest/pushSwitch
   ```

6.9.4. 应用启动不成功-注册中心排查思路

排查流程

应用启动不成功，怀疑注册中心有问题时，请按照如下流程排查：



1. 使用以下命令查看是否有 9600 端口的连接。

```
netstat -anlp | grep 9600
```

   - 如果没有 9600 端口的连接，需确认 ACVIP 配置是否正确。

     确认方法为：查看 `/home/admin/conf/acvipjava-client-cache/domains/` 下的 `*-DSR_CLOUD.json` 文件是否有内容并查看内容是否更新及时。

     - 有内容，且内容更新及时：

       查看注册中心启动日志，找连接失败原因。查看日志命令如下：

       ```
       tail -n 50 /home/admin/logs/registry/registry-client.log
       ```

     - 无内容或内容更新不及时：

       a. 检查 `com.alipay.env=share,com.antcloud.antvip.endpoint=IP` 配置是否正确。

       b. 查看 `/home/admin/logs/acvip-java-client/api-stat.log` 日志是否有报错。

       c. 测试是否可以拉取到配置，测试命令如下：

       ```
       curl -i -XPOST {acvip9003}:□□/antcloud/antvip/
       instances/get -d '{"vipDomainName2ChecksumMap”:{“domain 名称:”N”}}’
       ```

   - 有 9600 连接执行步骤 2。

2. 查看 registry 日志（老版本查看 confreg 日志）。

   查看 registry 日志方法如下：

   - 执行 `tail -n 50 /home/admin/logs/registry/registry-client.log` 命令查看启动日志。
   - 执行 `grep "DsrSubscribeCallback" /home/admin/logs/rpc/rpc-registry.log -A 2` 命令查看 sub 日志，确认是否订阅到有效节点。

     - 订阅不到服务：

       a. 执行 `curl http://:9603/digest/pushSwitch` 命令查看注册中心 push 开关是否打开。

       b. 询问服务提供方的服务是否正常，其他订阅者是否正常。

       c. 是否有双机房配置，确认服务提供方和消费方在同一机房。

       d. 查看机器连接的 session 是否有 sub 记录，查看命令如下：

       ```
       curl -XPOST -H "Content-Type: application/
       json" -d '["10.10.0.141:51508"]' localhost:9603/digest/sub/connect/query
       ```

       e. 查看 data 上的 datainfopub 信息，查看命令如下：

       ```
       curl http://localhost:9622/digest/datum/query?
       dataId="com.alipay.sofa.cloud.middleware.service.id"\&group="DEFAULT_GROUP"\&in
       stanceId
       ```

     - 能订阅到服务，查看 pub 日志。

- 执行 `grep "register to server success" /home/admin/logs/registry/registryclient.log` 命令查看 pub 日志。

  若 pub 端有日志，其他服务订阅不到，或者数量不对。请按照如下步骤排查：

  a. 执行 `curl http://:9603/digest/pushSwitch` 命令查看注册中心 push 开关是否打开。

  b. 是否有双机房配置，确认服务提供方和消费方在同一机房。

  c. 查看机器连接的 session 是否有 pub 记录，查看命令如下：

  ```
  curl -XPOST -H "Content-Type: application/json" -d '["10.10.0.141:51508"]' localh
  ost:9603/digest/pub/connect/query
  ```

  d. 查看 data 上的 datainfopub 信息，查看命令如下：

  ```
  curl http://localhost:9622/digest/datum/query?
  dataId="com.alipay.sofa.cloud.middleware.service.id"\&group="DEFAULT_GROUP"\&inst
  anceId="000001"
  ```

7.权限说明

微服务控制台使用中间件产品的用户角色定义，包括以下三种角色：

- 共享中间件-管理员
- 共享中间件-开发者
- 共享中间件-观察者

> 说明
>
> 专有云环境若需开启鉴权，请联系技术支持工程师。开启后，您可前往 IAM 控制台获取 AK（AccessKey ID）和 SK（AccessKey Secret）信息。获取 AK 和 SK 操作，请参见 获取 AK 和 SK。

具体权限设置请参见下表。若要修改您的用户角色，请联系空间管理员处理。

模块	操作	共享中间件-管理员	共享中间件-开发者	共享中间件-观察者
服务管控	启用服务提供者	√	×	×
	禁用服务提供者	√	×	×
	修改服务提供者权重	√	×	×
	恢复服务提供者配置	√	×	×
	查询服务	√	√	√
	浏览数据	√	√	√
动态配置	创建配置类	√	√	×
	修改配置类	√	√	×
	删除配置类	√	√	×
	创建属性	√	√	×

模块	操作	共享中间件-管理员	共享中间件-开发者	共享中间件-观察者
	修改属性	√	√	×
	修改属性	√	√	×
	通过文件导入配置类	√	√	×
	通过文件导出配置类	√	√	×
	推送配置值	√	×	×
	灰度推动	√	×	×
	查询配置类	√	√	√
	浏览数据	√	√	√
	查看推送记录	√	√	√
服务治理	新增应用	√	√	×
	新增规则	√	√	×
	全局开关	√	√	×
	删除应用	√	√	×
	删除规则	√	√	×
	修改规则	√	√	×
	单个规则启用禁用	√	√	×

模块	操作	共享中间件-管理员	共享中间件-开发者	共享中间件-观察者
	通过文件导入治理规则	√	√	×
	通过文件导出治理规则	√	√	×
	查询应用	√	√	√
	查看推送记录	√	√	√
	浏览数据	√	√	√

8.中间件细粒度权限管控方案

目前中间件只有 middleware_root、middleware_master、middleware_developer、middleware_observer 四种角色，权限管控粒度比较粗，本文介绍如何通过 IAM 实现中间件更灵活、可配的权限管控能力。

当前鉴权流程

1. 各个中间件控制台通过引入 OSP 的 JAR 包，引入了 `UserInfoFetcherJaxrsFilter`，并通过 RPC 调用 OSP 的 `UserFacade`。`UserFacade` 会通过调用 IAM 的 WebAPI 获取用户信息以及用户和用户下的所有租户的角色，最后会把信息写入 `resteasy` 的 `context` 上下文。
2. 在引入 OSP 的 JAR 包的同时，也会引入一个 `OSPRoleBasedSecurityFilter`，该 Filter 通过一个对应的 Feature 去提取各个中间件控制台代码上打的注解（RolesAllowed），来提取调用该方法需要具有的角色，再调用上面写入的 `context` 的 `hasAnyRoles` 进行鉴权。
3. `hasAnyRoles` 会调用 OSP 的 `UserFacade` 的 `judge` 方法，`judge` 方法会调用 IAM 的 OpenAPI 进行鉴权。

   OpenAPI 的请求参数里除了必带的 `tenant` 参数，剩余参数中：

   - `action` 用于添加角色。
   - `conditions` 在 mock 掉 apconsole 的情况下传入 instanceId；不 mock 的情况下传入 workspace。

细粒度管控方案

IAM 除了支持 condition 这种自定义条件，还支持通过 resource 进行资源级别的鉴权。方法如下：

- 各个中间件控制台代码，以 dsrconsole 为例，需要在 IAM 控制台配置多个角色，角色里包含若干个 `action`，`action` 可以定位为 OpenAPI 的API 名称（例如 `ms.ddcs.attribute.add`）。
- 在接口上增加注解：

```
@RolesAllowed(MS.DDCS_DEVELOPER)
String getConfig();
```

```
@MiddlewareAuth(MS_DDCS_ATTRIBUTE_ADD, method=POST)
String addAttribute(String attribute);
```

  以上注解是 IAM 会用到的 action，并搭配类似 HTTP 的 Method。

- 将 OSP 的 SDK 升级到 1.2.19 版本。

以上操作完成后，您可以参考以下配置示例将中间件产品接入 IAM。

配置示例

以 dsrconsole 接入 IAM 鉴权为例，操作如下；

1. 在云游中发布微服务前，修改以下应用启动参数：

- `osp.canUseJudge=true`（必选）：鉴权是否远程调用 OSP 接口。false 为不使用；true 为使用。
- `osp.auth.fine.grain=true`（必选）：是否使用 OSP 细粒度鉴权。true 为启用；false为不启用。

  默认不启用，使用旧的共享中间件管理员、共享中间件观察者、共享中间件开发者鉴权。
- `osp.auth.strict.mode=false`（可选）：是否使用 OSP 细粒度鉴权严格模式。true 为使用；false为不使用。

  默认不启用，严格模式下，查看的请求也会报 403 错误。

2. 在 IAM 控制台录入权限。
   i. 下载 MS 操作点。
   ii. 登录 IAM 控制台。
   iii. 选择租户为 ROC。

   iv. 在左侧导航栏选择 **系统管理 > 权限元数据**。
   v. 单击 **权限录入**，然后将步骤 1 下载的文件上传。
   vi. 单击 **提交**。

9.常见问题

9.1. DRM 常见问题

本文汇介绍动态配置（DRM）在使用过程中遇到的常见问题及解决方法。

如何利用 DRM 动态设置日志等级？

基于 SOFABoot 开发的程序，对于日志有如下默认配置：

- application.properties

```
# log level for current application with groupid com.hula.sofa
 logging.level.com.hula.sofa=INFO
```

- logback-spring.xml

```
<springProperty scope="context" name="logging.level" source="logging.level.com.hula.sofa"/>
```

您可以通过 DRM 来动态更新日志等级，步骤如下：

1. 在 SOFABoot 工程中定义对应的动态配置类。

   示例如下：

```
        import com.alipay.drm.client.DRMClient;
        import com.alipay.drm.client.api.annotation.DAttribute;
        import com.alipay.drm.client.api.annotation.DObject;
        import com.alipay.drm.client.api.model.DependencyLevel;
        import org.springframework.beans.factory.annotation.Autowired;
        import org.springframework.beans.factory.annotation.Value;
        import org.springframework.boot.logging.LogLevel;
        import org.springframework.boot.logging.LoggingSystem;
        @DObject(region="hula", appName ="dynamic-log-level", id ="com.hula.sofa.config
.DynamicConfig")
        public class DynamicConfig{
            // 取 application.properties 配置的值作为默认值。
            @DAttribute
            @Value("${logging.level.com.hula.sofa}")
            private String loglevel;
            @Autowired
            LoggingSystem loggingSystem;
            public void init(){
                DRMClient.getInstance().register(this);
            }
            public String getLoglevel(){
                 return loglevel;
            }
            public void setLoglevel(String loglevel){
                this.loglevel = loglevel;
                LogLevel level =LogLevel.valueOf(loglevel.toUpperCase());
                // 使用 springboot的LoggingSystem 设置目标 logger 的日志等级。
                loggingSystem.setLogLevel("com.hula.sofa", level);
            }
        }
```

2. 在 SOFA 微服务平台上新增如下动态配置。

   操作步骤，请参见 新增动态配置。

> 说明
>
> 在客户端值处，推送成功的情况下不会显示新的值。

重启服务后，为何属性获取的是动态配置推送过的值，而不是默认值？

动态配置的默认用法是：当服务端推送配置后启动，客户端会默认同步加载服务端配置值。如果您不想使用该默认同步加载，需设置 `@DAttribute(dependency = DependencyLevel.NONE` 。

更多依赖级别，说明如下：

依赖等级	依赖描述
NONE	无依赖，启动期不加载服务端值。启动此级别后，客户端仅会接收服务端在运行期间产生的配置推送。
ASYNC	异步更新，启动期异步加载服务端值，不关注加载结果。
WEAK	弱依赖，启动期同步加载服务端推送值。 • 当服务端不可用时不影响应用正常启动。 • 当服务端可用后，客户端会依靠心跳检测重新拉取到服务端值。
STRONG	强依赖，启动期同步加载服务端值。 • 如服务端未设置值，则使用代码初始化值。 • 如从服务端获取数据请求异常，或客户端设值异常时，均会抛出异常，应用启动失败。
EAGER	最强依赖，启动期必须拉取到服务端值。如服务端未推送过值则抛异常，应用启动失败。

如何给属性赋初始值？

您可以通过以下方式为属性赋值：

- 直接在定义式赋值。例如：`private String loglevel = "info"` 。
- 从 `application.properties` 中获取，示例如下：

```
@DAttribute private String loglevel ="info";   //初始值为 info
```

或

```
@Value("${logging.level.com.hula.sofa}")
@DAttributeprivate String loglevel;
```

重要

请勿在 init 方法里赋值，否则客户端重启后会覆盖推送值，并变回默认值，造成后续推送无法生效。示例如下：

```
public void init(){
        DRMClient.getInstance().register(this);
        setLoglevel("info");
}
```

发布部署卡在部署服务中，直到超时，导致发布部署失败

问题现象：

发布部署卡在部署服务中，直到 8 分钟后超时，导致发布部署失败。

问题原因：

在 DRM 中设置了 `RefreshCacheDRM.refreshCacheType = GEOHASH`，业务代码在收到该项更新后，需花费十几分钟处理业务逻辑。

解决方案：

- 临时方案：设置 `RefreshCacheDRM.refreshCacheType = null`，这样暂时不会触发业务处理逻辑。
- 长期方案：需要优化您的业务代码，在收到 DRM 的属性更新后，使用异步线程，延迟处理该业务，并及时反馈更新成功的信号给 DRM。

启动后拉取不到配置

问题现象：

应用启动后拉取不到配置，查看日志出现如下信息：

- `logs/drm/drm-boot.log`

```
2019-06-26 16:28:15,893 INFO Query inital drm value was empty. app = sofa-client, attribute = message
```

- `logs/drm/drm-monitor.log`

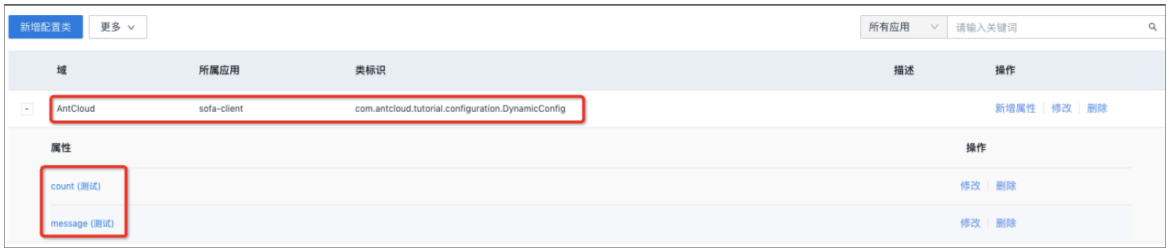
```
2019-06-26 16:28:15,893 INFO Query data from zdrmdata: version = 0, dataId = AntCloud.sofa-client:name=com.antcloud.tutorial.configuration.DynamicConfig.message,version=3.0@DRM, value =
```

解决方案：

1. 检查应用是否与动态配置服务端连接。

```
netstat -nalp |grep 9880
```

2. 检查应用中配置的域名、应用名、类标识和属性是否与控制台配置一致。



配置推送不生效

问题现象：

推送配置后，应用的属性没有变化。

解决方案：

在 `logs/drm/common-error.log` 查看是否有错误信息：

```
2019-06-26 17:17:30,655 ERROR ATTRIBUTE_GET_SET_THREAD - Update drm attribute failed! app = sofa
```

确认推送的配置值是否能够正确转化成应用代码中定义的属性类型，例如应用端代码中定义的属性类型为整型 int，但是推送的值为字符串，如"test"。

9.2. 服务限流常见问题

本文汇总梳理了限流功能使用过程中遇到的常见问题。

为什么服务限流配置后没有效果？

排查步骤如下：

1. 确认 SOFA 的版本大于 3.3.0，如果低于这个版本，请升级版本。

   SOFA 版本说明，请参见 版本说明。

2. 接入动态配置客户端和服务限流。

```
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>ddcs-enterprise-sofa-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>com.alipay.sofa</groupId>
    <artifactId>guardian-sofa-boot-starter</artifactId>
</dependency>
```

3. 确认客户使用的是 REST 协议的限流方式，步骤如下：

i. 检查代码的编写，验证 Facade 接口、实现类和 XML 配置。接口定义如下：

```
@Path("abs-teside/TestIde")
@Consumes({ "application/json;charset=UTF-8" })
@Produces({ "application/json;charset=UTF-8" })
public interface Absteside_TestIde {
    @GET
    String cloudService();
}
```

ii. XML 中配置实现类的 bean，同时发布 REST 协议的服务。

```
<!--对rest访问限流进行配置，还需要在下面aop的Bean中配置一个value值-->
<bean id="Absteside_TestIde" class="facade.impl.Absteside_TestIdeImpl"/>
<sofa:service ref="Absteside_TestIde" interface="facade.Absteside_TestIde">
    <sofa:binding.rest/>
</sofa:service>

<!--下面是固定内容，只需要将上面的beanId配置到这里-->
<import resource="classpath:META-INF/spring/guardian-sofalite.xml"/>
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="interceptorNames">
        <list>
            <value>guardianExtendInterceptor</value>
        </list>
    </property>
    <property name="beanNames">
        <list>
            <!-- 配置需要被拦截的 bean -->
            <value>Absteside_TestIde</value>
        </list>
    </property>
    <!-- 如要使用 CGLIB 代理，取消下面这行的注释 -->
    <!--<property name="optimize" value="true" />-->
</bean>
```

4. 将应用部署到 CAFE 进行测试，步骤如下：

i. 部署完成，通过查看 Guardian 的 `guardian-default.log` 日志文件，确认 Guardian 组件已经成功注册。

```
sh-4.2# more guardian-default.log
2020-04-27 17:49:48,259 INFO  register GuardianDrm success, appName=duardian-service
```

ii. 在 SOFA 控制台配置限流条件。

配置的应用名称必须和 `applicatioin.properties` 中的 `spring.application.name` 值一致。

iii. 配置完成以后，重新打开服务限流的开关。

iv. 通过查看 Guardian 的 `guardian-default.log` 日志文件确认应用程序成功打印了配置信息的日志，如下所示：

```
sh-4.2# more guardian-default.log
2020-04-27 17:49:48,259 INFO  register GuardianDrm success, appName=duardian-service
2020-04-27 17:52:25,356 WARN  receive message with key=[guardianConfig] and value=[{"@type":"com.alipay.guardian.client.drm.GuardianConfig","engineConfigs":
{"@type":"java.util.HashMap","LIMIT":{"@type":"com.alipay.guardian.client.engine.limit.LimitEngineConfig","actionConfigMap":{"@type":"java.util.HashMap"},"g
lobalConfig":{"enable":false,"runMode":"CONTROL"},"ruleConfigMap":{"@type":"java.util.HashMap"}}},"version":1}]
2020-04-27 17:52:25,357 WARN  after update with key=[guardianConfig]
2020-04-27 17:52:25,392 INFO  Guardian Config version=1
2020-04-27 17:52:25,394 WARN  rebuild Rules, GuardianFactory: class com.alipay.guardian.client.sofalite.limit.LimitGuardianFactory
2020-04-27 17:52:25,394 WARN  After rebuildRules, refresh guardianMethodInterceptor.methodMathedMap!
2020-04-27 17:52:29,316 WARN  receive message with key=[guardianConfig] and value=[{"@type":"com.alipay.guardian.client.drm.GuardianConfig","engineConfigs":
{"@type":"java.util.HashMap","LIMIT":{"@type":"com.alipay.guardian.client.engine.limit.LimitEngineConfig","actionConfigMap":{"@type":"java.util.HashMap"},"g
lobalConfig":{"enable":true,"runMode":"CONTROL"},"ruleConfigMap":{"@type":"java.util.HashMap",141:{"@type":"com.alipay.guardian.client.engine.limit.LimitRul
eConfig","actionConfig":{"actionType":"LIMIT_EXCEPTION","responseContent":"too much request for rest"},"calculationConfigs":[{"calculationType":"INVOKE_BY_T
":141,"limitStrategy":"QpsLimiter","limitType":"GENERIC_LIMIT","maxBurstRatio":0.0,"resourceConfigs":[{"baseName":"facade.SampleRestFacade.testParam","metho
d":"testParam","resourceType":"METHOD","ruleIds":[]}],"resourceType":"METHOD","ruleBizId":"TestBeanByrest","runMode":"CONTROL"}}}},"version":1}]
2020-04-27 17:52:29,316 WARN  after update with key=[guardianConfig]
2020-04-27 17:52:29,369 INFO  Guardian Config version=1
```

v. 确认以上信息无误后，在浏览器进行访问，验证效果。可通过 `guardian-limit-stat.log` 查看成功通过的请求。

```
sh-4.2# more guardian-limit-stat.log
2020-04-28 10:14:30,004 INFO  CONTROL, 141,TestBeanByrest_ALL,10000,2020-04-28T10:14:20,2020-04-28T10:14:30,INVOKE_BY_TIME,2,13,2,11
2020-04-28 10:14:40,004 INFO  CONTROL, 141,TestBeanByrest_ALL,10000,2020-04-28T10:14:30,2020-04-28T10:14:40,INVOKE_BY_TIME,2,4,2,2
2020-04-28 10:16:10,004 INFO  CONTROL, 141,TestBeanByrest_ALL,10000,2020-04-28T10:16:00,2020-04-28T10:16:10,INVOKE_BY_TIME,2,3,2,1
2020-04-28 10:16:20,004 INFO  CONTROL, 141,TestBeanByrest_ALL,10000,2020-04-28T10:16:10,2020-04-28T10:16:20,INVOKE_BY_TIME,2,21,2,19
2020-04-28 10:16:30,004 INFO  CONTROL, 141,TestBeanByrest_ALL,10000,2020-04-28T10:16:20,2020-04-28T10:16:30,INVOKE_BY_TIME,2,2,2,0
```

为什么在配置的 logs 文件夹下没有限流日志文件目录？

您需要在启动类中添加自定义路径，限流的日志文件将在设置的目录生成。示例如下：

```
System.setProperty("user.home","/tmp/localfile")
```

为什么业务程序启动时，Guardian 没有加载？

问题现象：

在业务程序启动的时候，Guardian 没有加载，日志目录也没有生成。

问题原因：

业务程序打包时，没有引入 Guardian 的 JAR 包。

解决方案：

在进行打包业务程序时，确保引入 Guardian 的 JAR 包。

9.3. RPC 常见问题

本文汇总梳理了 RPC 使用过程中遇到的常见问题及排查思路。

使用 RPC 客户端调用服务时报错

调用服务时报“RPC-02306: 没有获得服务 [{0}] 的调用地址，请检查服务是否已经推送”错误

排查思路如下：

1. 检查服务地址是否推送。

   登录客户端，查看 `/home/admin/logs/rpc/sofa-registry.log` 日志，您可以通过服务接口名过滤日志找到最后一次推送记录。如果发现服务端地址没有推送到客户端，建议首先排查服务是否注册成功。例如，以下日志中有可调用目标地址[0]个的记录，则说明 `com.alipay.share.rpc.facade.SampleService` 的服务端地址没有推送到客户端：

   ```
   RPC-REGISTRY - RPC-00204:接收 RPC 服务地址：服务名[com.alipay.share.rpc.facade.SampleService:1.0@DEFAULT]
   可调用目标地址[0]个
   ```

2. 检查客户端启动时是否收到 RPC Config 推送。

   查看 `/home/admin/logs/rpc/rpc-registry.log` 日志，确定最近一次 RPC 客户端的启动时间。您可以根据客户端上次启动时间和服务接口名过滤日志，检查对应的接口是否有 `Receive Rpc Config info` 的记录。如果没有也会导致后续无法调用服务，可以考虑重启客户端。

3. 检查服务是否注册成功登录 SOFA 应用中心。

   查看服务注册情况，或登录服务端查看 `/home/admin/logs/confreg/config.client.log` 日志。如果有服务发布相关的错误，可根据日志信息进一步排查。

4. 检查服务调用是否早于地址推送时间。

   如果客户端日志 `sofa-registry.log` 中显示服务地址已经推送，但是 RPC-02306 错误发生的时间在服务地址推送之前，这种情况多发生在调用服务时，客户端应用还没有完成启动。问题原因多为业务系统自己通过定时任务调用服务，或者在 bean 初始化完成后就开始调用服务导致的报错，可通过配置 `address-wait-time` 来解决。

5. 检查 RPC 服务端和客户端应用配置信息是否匹配。

   分别打开服务端和客户端应用的配置文件 `application.properties`，查看 `com.alipay.instanceid` 和 `com.antcloud.antvip.endpoint` 参数是否配置相同。如配置不同，RPC 客户端将无法感知 RPC 服务端。

6. 检查服务注册中心连接。

   运行以下命令以检查客户端和服务端与服务注册中心的连接情况：

   ```
   netstat -a |grep 9600
   ```

   9600 端口是服务注册中心的监听端口，客户端和服务端与 9600 端口建立长连接，向服务注册中心发布和订阅服务。如果客户端或者服务端与 9600 端口的连接断开，则需要重启应用恢复，并进一步排查端口异常断开的原因。

7. 检查 RPC 服务端地址绑定。

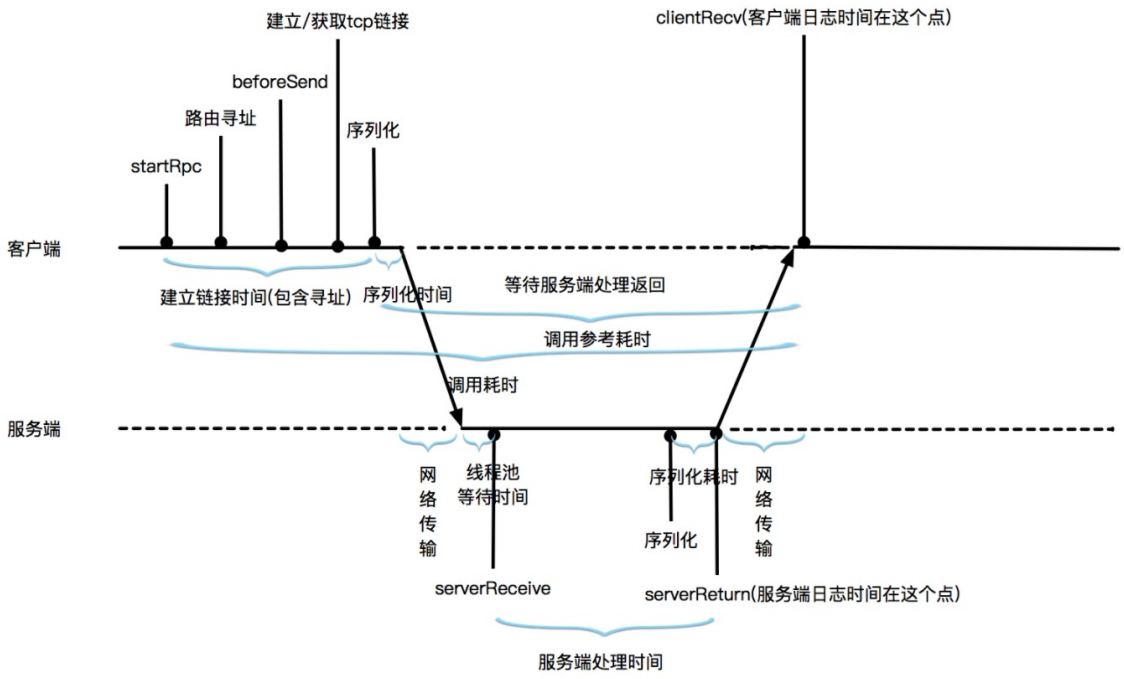
   登录 RPC 服务端，运行以下命令：

   ```
   ps -ef|grep java
   ```

   查看进程启动参数 `rpc_bind_network_interface` 或 `rpc_enabled_ip_range` 是否绑定了正确的 IP 地址。

错误日志中出现“Rpc invocation timeout[responseCommand TIMEOUT]”报错

RPC 调用时序图如下：



> **说明**
>
> 有关客户端和服务端各阶段的耗时信息，请参考 链路追踪。

若您调用 RPC 服务超时，在客户端的 `logs/tracelog/middleware_error.log` 日志中，看到如下异常信息：

```
2018-07-0613:21:20.463,sofa2-rpc-client,707c27b9153085447746110464663,0,main,timeout_error,
rpc,invokeType=sync&uid=&protocol=bolt&targetApp=sofa2-rpc-server&targetIdc=&targetCity=&pa
ramTypes=&methodName=message&serviceName=com.alipay.share.rpc.facade.SampleService:1.0&targ
etUrl=10.160.34.141:12200&targetZone=&,,com.alipay.sofa.rpc.core.exception.SofaTimeOutExcep
tion: com.alipay.remoting.rpc.exception.InvokeTimeoutException:Rpc invocation timeout[respo
nseCommand TIMEOUT]! the address is10.160.34.141:12200
```

您可以通过如下步骤进行排查：

1. 查看是否因服务本身问题导致的超时，如业务代码处理时间过长。

   默认情况下，RPC 的超时时间为 3 秒。要确定某个请求的实际处理时间，您可登录服务端查看 `logs/tracelog/rpc-server-digest.log` 日志。根据客户端超时日志中的 traceID，如 `707c27b9153085447746110464663`，找到服务端处理对应请求的日志。日志格式如下所示：

   ```
   2018-07-0613:21:22.441,sofa2-rpc-server,707c27b9153085447746110464663,0,com.alipay.shar
   e.rpc.facade.SampleService:1.0,message,bolt,,10.160.33.96,sofa2-rpc-client,,,4001ms,0ms
   ,SofaBizProcessor-12200-0-T46,02,,,1ms,,
   ```

   上述日志中服务端业务代码处理时间为 4001 毫秒。

   由于 RPC 调用默认的超时时间是 3 秒，如果日志中的耗时大于 3 秒或者非常接近 3 秒，建议首先从服务端本身排查，可能原因如下：

   - 服务端业务代码执行慢。
   - 服务端本身有外网服务调用，或者服务端又调用了其他 RPC 服务（client > RPC Server A > RPC Server B），此种情况需要分别排查 A 和 B，确定问题。
   - 服务端有数据库操作，如数据库连接耗时、慢 SQL 等。

   如因服务端本身原因导致超时，建议调整代码。

2. 查看是否因服务端 RPC 线程池耗尽导致的超时。

   登录服务端查看 `rpc/tr-threadpool` 日志。如果发生 RPC 线程池队列阻塞，先确认是否发生超时的时间段有业务请求高峰，或者用 `jstack` 查看业务线程是否有等待或者死锁情况，导致 RPC 线程耗尽。更多信息，请参见 应用维度配置扩展。

3. 查看是否因 GC 问题（Garbage Collection，简称 GC），导致线程停止。

   某些 GC 类型会触发“stop the world”问题，会将所有线程挂起。若要排查是否是 GC 导致的超时问题，可以通过以下方法开启 GC 日志。

   - 方法一：

     在 `config/java_opts` 文件中加入以下启动参数，并重新打包发布。

     ```
     -verbose:gc -XX:+PrintGCDetails-XX:+PrintGCDateStamps-Xloggc:/home/admin/logs/gc.log
     ```

   - 方法二：

     a. 用 `kill -15` 命令结束服务端进程。

b. 手动启动 RPC 服务。

运行 `su admin` 进入 admin 用户，用如下 `nohup` 形式启动 RPC 服务：

```
$ nohup java -verbose:gc -XX:+PrintGCDetails-XX:+PrintGCDateStamps-Xloggc:/home/admin/logs/gc.log -Drpc_bind_network_interface=eth0 -Dspring.profiles.active=&{环境标识}-jar /home/admin/app-run/sofa2-rpcserver-service-1.0-SNAPSHOT-executable.jar &
```

> **说明**
>
> **工作空间标识** 可登录 SOFAStack 控制台，然后在左侧导航栏选择 **资源管理** > **工作空间** 查看。

c. 等待下次 RPC 超时发生后，查看 `gc.log` 验证超时的时间段是否有耗时较长的 GC，尤其是 Full GC。

4. 查看是否因网络延时抖动导致的超时。

   您可以通过以下步骤排查：

   i. 在客户端和服务端运行 `tsar -i 1` 查看问题发生的时间点是否有网络重传。

   ii. 在客户端和服务端同时部署 `tcpdump` 进行循环抓包，当问题发生后分析网络包。

   iii. 在客户端和服务端运行 `ping` 观察是否存在网络延时。

5. 确认是否因其他外部因素影响服务器性能，如任务调度、批处理，或者与宿主机上其他虚拟机、容器发生资源争抢。

如何打印客户端 RPC 调用统计？

可以参考以下示例语句打印调用 `sofa2-rpc-server` 的应用超过 3 秒的请求总数、服务端IP、服务应用和客户端IP：

```
$ grep sofa2-rpc-server rpc-client-digest.log | awk -F,'{if(int($18)>3000)print $9,$10,$27}'|sort | uniq -c | sort -n
```

实际使用时，请将 `sofa2-rpc-server` 替换成对应的服务端应用名称，并根据日志中处理时长所对应列的具体位置调整 `$18` 数值。打印信息也可以根据需要调整。

为什么 SOFABoot 应用已经启动，但服务没有发布成功？

您可以根据以下几个情况进行排查：

- 应用非正常启动

  通常可以查看 `health-check` 日志。如果有 error 日志，可以根据相关信息进行排查，常见的故障信息包括：

  - redis 没有正确配置。

- 一个服务在本地开启了多个实例。
- Bolt 服务没有启动，并发现端口占用等。

- ACVIP 问题

  专有云环境，要确保 `instanceid` 、 `endpoint` 、 `access` 、 `secret` 参数配置正确。其中 `access` 、 `secret` 两个参数在确定接入 IAM 才需写入。

  如果上述参数配置错误，则会在 `logs/registry/registry-client.log` 中出现 `Vip(null) endpoint might be wrong` 的错误。该错误表明当前 ACVIP 配置有误或 ACVIP 出现网络故障，此时需要联系运维人员检查 ACVIP 和前端负载均衡器的网络连通性。

- 注册中心问题

  如果应用已经启动，但服务没有发布成功，则按下述步骤排查：

  i. 查看注册中心内有没有服务被注册， 如果没有，则排除注册中心的故障。

  ii. 查看是否是 ACVIP 的问题。如果排除后，服务还有问题，按下述步骤排查：

  a. 观察应用容器是否有类似 `/Users/xxx/conf/acvip-java-client-cache/domains/0000X-DSR_HTTP.json` 这样的文件。

  如果有，可进入该文件查看其内容，一般都是缓存到本地的 DSR 注册中心地址，可自行检查是否有异常。例如：健康检查不通过，IP 没有获取正确等。

  b. 通过命令判断当前注册中心是否正常，示例如下：

  ```
  curl -i -XPOST {antvip}:9003/antcloud/antvip/instances/get -d
  '{"vipDomainName2ChecksumMap":{"000001-DSR_CLOUD":"N"}}'
  ```

  。如果不正常，请检查注册中心是否配置正确。

- 服务提供方的运行模式

  云上发布时，未修改 `run.mode=DEV` 参数。在 DEV 模式下，将只注册到本地，而不会注册到注册中心里。

如何将 Dubbo 内部项目迁移到 SOFABoot 上?

问题描述：

- 如何将 Dubbo 内部项目迁移到 SOFABoot 上?
- 如果第三方需要保有 Dubbo，系统要如何设计?

解决方案：

系统改造过程中，并不能确保所有关联系统一次性改造完成，会面临需要和历史系统兼容的场景。例如一个服务被改造成 SOFA Bolt 服务后，发现还有调用方依然是依赖 Dubbo 的。那么，一个简单的兼容办法为：这个服务同时暴露 BOLT 和 Dubbo 服务。

在 SOFABoot 中暴露 Dubbo 服务，步骤如下：

1. 加入 Dubbo 的 starter 依赖。

   示例如下：

```
<dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>0.1.1</version>
</dependency>
<!-- Dubbo -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>dubbo</artifactId>
    <version>2.6.4</version>
</dependency>
<!-- Spring Context Extras -->
<dependency>
    <groupId>com.alibaba.spring</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>1.0.2</version>
</dependency>
```

2. 配置 `application.properties` 。

   示例如下：

```
################ common configuration ##############
spring.application.name=bank-dubbo-provider
logging.level.com.dubbo.example=INFO
logging.path=./logs
################ dubbo configuration ##############
demo.service.version =1.0.0
dubbo.application.id = bank-dubbo-provider
dubbo.application.name = bank-dubbo-provider
################ sofa configuration ##############
run.mode=DEV
com.alipay.sofa.rpc.bolt-port=12201
# shared middleware
com.alipay.env=shared
com.alipay.instanceid=IPYJUBMB231N
com.antcloud.antvip.endpoint=100.103.1.174
com.antcloud.mw.access=uPxHLxsMmstcQCNWEh
com.antcloud.mw.secret=TyMlUB9uGRMzcc2pG0dMv6xzUXCMA1WI
```

3. 添加 Dubbo 服务发布。

   示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sofa="http://schema.alipay.com/sofa/schema/slite"
    xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xs
    http://schema.alipay.com/sofa/schema/slite http://schema.alipay.com/sofa/slite.xsd
http://www.springframework.org/schema/context http://www.springframework.org/schema/con
text/spring-context.xsd http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/sc
hema/dubbo/dubbo.xsd">
    <!-- dubbo zookeeper configuration -->
    <dubbo:registry address="zookeeper://127.0.0.1:2181"/>
    <dubbo:protocol name="dubbo" port="20880"/>
    <!-- bean define -->
    <bean id="dubboService" class="com.dubbo.example.service.DubboServiceImpl"/>
    <!-- sofa service -->
    <sofa:service interface="com.dubbo.example.facade.DubboService" ref="dubboService"
unique-id="sofaDubboService">
        <sofa:binding.bolt/>
    </sofa:service>
    <!-- dubbo service -->
    <dubbo:service interface="com.dubbo.example.facade.DubboService" ref="dubboService"
version="1.0.0"/>
</beans>
```

> **说明**
>
> 服务需要引入 Dubbo 的 schema，这样基于 Dubbo 的定义才会被显示，并且 Dubbo 的注册中心是 zk，也需要配置。

4. 更新 main 函数，开启 Dubbo。

   示例如下：

```
@ImportResource({"classpath*:META-INF/bank-dubbo-provider/*.xml"})
@org.springframework.boot.autoconfigure.SpringBootApplication
@EnableDubbo
public class SOFABootSpringApplication{
     private static final Logger logger =LoggerFactory.getLogger(SOFABootSpringApplicat
ion.class);
     public static void main(String[] args){
          SpringApplication springApplication =new SpringApplication(SOFABootSpringAppl
ication.class);
          ApplicationContext applicationContext = springApplication.run(args);
     }
}
```

RPC 调用时报“02306，服务无法向注册中心发布”错误。

问题现象：

调用时错误日志如下：

```
-9987-exec-3] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet] in context wi

Exception: RPC-02306: 没有获得服务[com.ali.test.api.facade.PaymentService:1.0:2323qw]的调用地址，请检查服务是否已经推送
uster.noAvailableProviderException(AbstractCluster.java:444) ~[sofa-rpc-all-5.6.3.jar:5.6.3]
uster.select(AbstractCluster.java:377) ~[sofa-rpc-all-5.6.3.jar:5.6.3]
```

查看 RPC 注册中心日志，却未发现注册中心，示例如下：

```
com.alipay.sofa.rpc.registry.local.LocalRegistryHelper - Load backup file failure cause by can't found file: D:\tmp
com.alipay.sofa.rpc.registry.local.LocalRegistryHelper - Load backup file failure cause by can't found file: D:\tmp
```

发布服务无效，查看健康检查日志，发现问题如下：

```
20-04-02 15:27:16,146 ERROR main                                - Error occurred while doing ReadinessCh
m.alipay.sofa.rpc.core.exception.SofaRpcRuntimeException: Failed to start bolt server!
  at com.alipay.sofa.rpc.server.bolt.BoltServer.start(BoltServer.java:132)
```

故障原因：

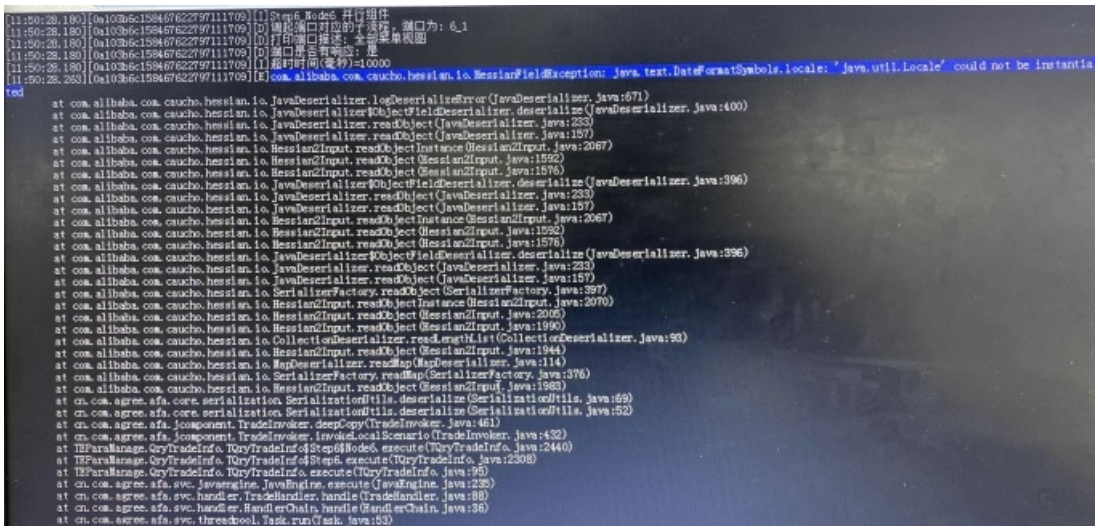
Bolt 端口 12200 被占用。

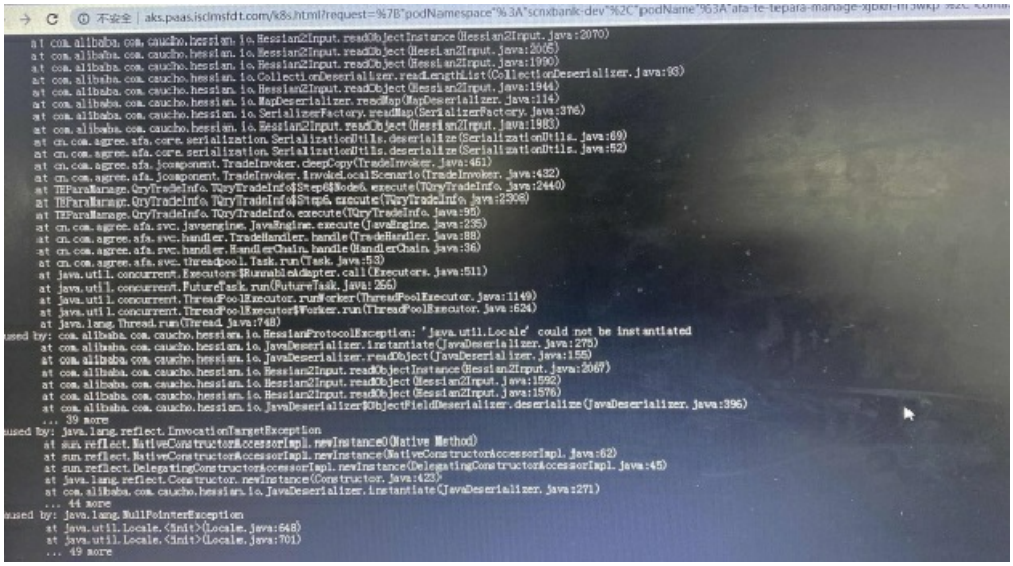
解决方案：

更换端口或者把本地占用端口的服务关闭。

进行 RPC 调用时出现“com.alibaba.com.caucho.hessian.io.HessianFieldException:'xxxx' could not be instantiated”报错

问题分析：





通过对上面的 2 个日志进行分析，结合日志的提示，可以得出下述结论：

- 进行 RPC 调用的时候，参数序列化失败。
- `java.util.Local` 这个类不能实例化，原因是 Local 这个类在 init 的时候，出现了空指针的异常。

解决方案：

修改序列化方法。

如何解决应用启动失败，服务无法注册的问题？

问题描述：

应用依赖了一些 SOFA 组件，但在本地环境中仅希望测试 RPC，如何处理下述问题：

- 应用启动失败
- 服务无法注册

解决方案：

健康检查机制会在项目启动的时候对所有组件进行探活，如果此时引用了 DDCS（Distributed Dynamic Configuration Service）或者其它组件，会出现应用启动正常，但 RPC 无法在本地注册的现象。此时，如果业务暂时没有用到这些组件，可以在 healthcheck 时略过这些检查。本质原因是这些组件会到 antvip 中寻找组件地址，而此时应用并不在云上，所以会失败。具体操作如下：

在 `application.properties` 中添加下述配置，以略过所有组件的健康检查。

```
com.alipay.sofa.healthcheck.skip.component=true
```

> **重要**
>
> 此方案只建议在测试中使用，线上环境一定要打开健康检查。

SOFARest 接口在上传文件时，文件超过 10 MB 时会报错，如何处理？

问题现象：

SOFARest 接口在上传文件时，文件超过 10 MB 时会报错，报错信息如下：

```
ERROR org.jboss.resteasy.core.ExceptionHandler- failed to execute
javax.ws.rs.NotFoundException:Couldnot find resource for full path: http://unknown/bad-requ
est
    at org.jboss.resteasy.core.registry.ClassNode.match(ClassNode.java:73)
    at org.jboss.resteasy.core.registry.RootClassNode.match(RootClassNode.java:48)
```

排查步骤：

1. 在日志里将该报错打开，将 Netty 中 HTTP 相关类的日志改成 debug 模式。

   示例如下：

   ```
   logging.level.io.netty.handler.codec.http.HttpObjectAggregator=DEBUG
   ```

2. 获取具体报错的原因，例如 `Failed to send a 413 Request Entity Too Large` 。

3. 在 `application.properties` 文件里设置相应参数。

   ```
   com.alipay.sofa.rpc.RestMaxRequestSize=104857600
   ```

   假如应用的内存容量默认比较小，比如 1 GB 或者 2 GB，而 Netty 的 REST 所请求的 Payload 是放在 DirectMemory 里的，且该 DirectMemory 有个最大值，默认是系统 JVM 初始化时所申请内存大小。如果上传大文件时发生了 OOM 或者 DirectMemory 内存溢出的错误，需要进行下述处理：

   i. 确定当前系统的内存大小是否能够承受所传大文件。

   ii. 确认运行时内存或者 DirectMemory 的最大值。

   iii. 如果可以优化，请选择分段上传。

客户端调用远端服务时有大量的超时，但服务端的响应正常，且耗时很少，该如何排查？

您可以排查以下几个方面：

- 线程池发生了阻塞

  该问题一般在多级链路调用时才会发生。比如：A 调用 B 再调用 C，B 作为服务端的线程发生了阻塞，则需要查看 B 的 `tr-threadpool.log` 日志。

- GC 处理遇到严重故障

  框架遇到 GC 的故障很少，但也不排除在某些特定的场景会发生的概率。一般查看 `logs/stdout.log` 日志文件，并据此查看 CMS-remark、YG、ParNew 等指标，它们都标识着 STW，从而会导致 JVM 停顿。

- 硬件磁盘 IO 故障

  一般通过 `tsar -I 1` 查看下一分钟内的 IO 请求次数。在某些场景下，如果磁盘 IO 较高会影响到整个系统的性能。

- 网络故障

  网络问题一般都很难定位，这里介绍一个较为常见的网络设备故障问题，即防火墙会剔除不活跃（90s）链接或 LVS 故障切流量剔除链接时，均不会向客户端 Socket 发 RST 包，这样会导致客户端存在脏 Socket。

  某台机器请求一个具体 IP 的服务，该服务流量不大，所以请求频率很低，几十分钟甚至几小时一次。当超时时，会超时一次即断开连接，或连续超时 n 次后链接才断开。超时一次，应该是防火墙断开的链接；超时 n 次，则是 LVS 断开的链接。

  为什么超时 n 次之后才断开连接？因为 Socket 已经是脏链接，在写数据的时候，数据仅写到了 TCP Buffer 中，没有真正写出去，而这个时候 OS 并不会给上层使用者一个中断，仅当 TCP Buffer 写满之后，才会出现 `org.apache.mina.common.WriteTimeoutException`。TCP Buffer 在服务器上一般是 64 KB。

  一般这种问题，可以配置心跳来排除故障，或结合故障剔除功能来排查。

SOFA 客户端调用耗时较长的服务时，需要注意什么？

问题描述：

- SOFA 客户端调用耗时较长的服务时，需要注意什么？
- 如果设置了超长时间，没有生效，怎么排查错误？

排查思路：

服务发起方如果发现对方是一个耗时较长的服务，则需要配置一个比较合理的超时时间，否则，要判断该接口是不是需要一个 oneway 方式去执行。如果必须等待结果，且触发后发现，无论如何配置，超时时间都无法生效，则需检查防火墙或负载均衡器是否在上游配置了连接超时控制。

RPC 本地调用，修改注册中心路径的方式有哪些？

目前只能是设置环境变量方式：`System.setProperty("user.home","本地目录")`。企业版会把注册中心的路径配置包装起来，SOFA 在启动的时候会默认读这个 `user.home`。在配置的时候需要多添加两个字符（/），因为框架会省掉前两个字符，从第三个字符开始读取，例如：`user.home=//c://hulu`。

> **重要**
>
> 这个问题，只会在本地开发的时候会遇到。云上开发不需要关心注册中心。 企业版是通过 antvip 来获取一个健康的注册中心的地址，然后会构建 dsr://ip:port，同时，企业版将这个构建过程包装在了框架里。

有没有 Resteasy 的 key-value 方法请求示例？

示例如下：

- SpringMVC 的 Controller 请求方法：
  - URL：`http://localhost:8080/test?str=aaa`
  - 代码配置，示例如下：

```
@GetMapping("/test")
public String testParam(@RequestParam("str") String str){
    return str;
}
```

- Resteasy 的 GET 请求：
  - 类型一：
    - URL：`http://localhost:8341/webapi/users/test/xiaoming`
    - 代码配置，示例如下：

```
@Path("/webapi/users")
public interface SampleRestFacade{
    @GET
    @Path("/test/{userName}")
    public RestSampleFacadeResp<DemoUserModel> user(@PathParam("userName")String user
Name)throws CommonException;
}
```

  - 类型二（key-value）
    - URL：`http://localhost:8341/webapi/users/test?userName=xiaoming`
    - 代码配置，示例如下：

```
@Path("/webapi/users")
public interface SampleRestFacade{
      @GET
      @Path("/test")
      public RestSampleFacadeResp<DemoUserModel> userInfo(@QueryParam("userName") Str
ing userName) throws CommonException;
}
```

- `@FormParam`：将表单中的字段映射到方法调用上，此类方式提交方式一般为 Post。

RPC Tracer 日志格式说明

日志格式请参见 SOFARPC 日志。

有没有泛化调用的示例？

SOFARPC 在框架层面提供了通用的接口方法和类型：

- 服务的接口名：通过服务定义设置。
- 方法名和参数列表：通过 `$invoke` 或 `$genericInvoke` 传入。
- 自定义类型：使用 `GenericObject` 。

其中几个特别需要注意事项为：

- `$invoke` 方法：只用于参数类型，可以被当前应用的类加载器加载，如果只有基础类型，则可以使用此方法。
- `$genericInvoke` 结合 `GenericObject` ，当参数类型无法被当前应用的类加载器加载时，使用该方法。
- `argTypes` 必须传递接口声明的参数类型，不可使用子类类型。
- 调用 `$genericInvoke` 接口时，会将除以下包以外的其他类序列化为 `GenericObject` ：

  ```
  "com.sun","java","javax","org.ietf","org.ogm","org.w3c","org.xml","sunw.io","sunw.util"
  ```

- `GerericContext` 暂时只用于单元化场景。
- `GenericObject` 、 `fields` 的 value 也可以是一个 `GenericObject` 。

SOFARPC 的泛化调用，示例如下：

- 服务方：

- 服务、接口和类型定义：

```
// 服务定义
<sofa:reference interface="com.alipay.sofa.rpc.api.GenericService" id="xxxGenericServic
e">
    <sofa:binding.tr>
        <sofa:global-attrs generic-interface="目标服务接口的fullname"/>
    </sofa:binding.tr>
</sofa:reference>

    // 接口方法定义
    public interface GenericService{
        Object $invoke(String methodName,String[] argTypes,Object[] args) throws Generi
cException;
        Object $genericInvoke(String methodName,String[] argTypes,Object[] args) throws
GenericException;
        Object $genericInvoke(String methodName,String[] argTypes,Object[] args,Generic
Context context) throws GenericException;
        <T> T $genericInvoke(String methodName,String[] argTypes,Object[] args,Class<T>
clazz) throws GenericException;
        <T> T $genericInvoke(String methodName,String[] argTypes,Object[] args,Class<T>
clazz,GenericContext context) throws GenericException;
    }
    // 类型定义
    public final class GenericObject implements Serializable{
        private String type;
        private Map<String,Object> fields =new HashMap<String,Object>();
    }
```

- 服务方的接口、自定义类型和发布泛化调用的配置：

```
public interface PeopleService{
    String hello();
    String hello(String arg);
    People hello(People people);
    String[] hello(String[] args);
    People[] hello(People[] peoples);
}
public class People{
    private String name;
    private int age;
   //getter和setter方法
}
<!--发布泛化接口的配置-->
<bean id="genericService" class="com.aliyun.gts.financial.product.demo.rpc.server.servi
ce.PeopleServiceImpl"/>
<sofa:service ref="genericService" interface="com.aliyun.gts.financial.product.demo.ser
vice.facade.PeopleService">
     <sofa:binding.bolt/>
</sofa:service>
```

- 客户端：

  i. 定义泛化的服务，并设置正确的目标服务接口。

```
<!--调用泛化接口的配置-->
<sofa:reference interface="com.alipay.sofa.rpc.api.GenericService" id="genericFacade"
>
     <sofa:binding.bolt>
          <sofa:global-attrs generic-interface="com.aliyun.gts.financial.product.demo
.service.facade.PeopleService"/>
     </sofa:binding.bolt>
</sofa:reference>
```

重要

reference 里的 interface 需要填写框架定义的 GenericService 接口。 global-attrs 里的 generic-interface 才是填写真正的目标服务接口。 reference 里的 interface 都是 GenericService，如果要泛化调用多个不同的服务接口，可通过 reference 的 id 来区分。

ii. 通过 GenericService 的方法来调用目标方法。

```
    @Controller
    public class TestController{
        private String peoplePath ="com.aliyun.gts.financial.product.demo.rpc.bean.Pe
ople";
        private static final Logger logger =LoggerFactory.getLogger(TestController.cl
ass);
        /**
         * 默认ByName注入
         */
        @Autowired
        private GenericService genericFacade;
        /**
         * 无参场景使用$invoke
         * $invoke方法只用于参数类型可以被当前应用的类加载器加载，如果只有基础类型可以使用此方
法
         * 泛化调用 String hello()方法
         */
        @GetMapping("/test/invokeWithoutArgs")
        @ResponseBody
        @Produces("application/json;charset=UTF-8")
        public void invokeWithoutArgs(){
            String result =(String) genericFacade.$invoke("hello",
                    newString[]{},
            new Object[]{});
            if(logger.isInfoEnabled()){
                logger.info("Generic invoke result: {}", result);
            }
        }
        /**
         * $invoke调用，有参数
         * 泛化调用 String hello(String arg);
         */
        @GetMapping("/test/invokeBasicTypeMethod")
        @ResponseBody
        @Produces("application/json;charset=UTF-8")
        public void invokeBasicTypeMethod(){
```

```
        String result =(String) genericFacade.$invoke(
                "hello",
                new String[]{String.class.getName()},
        new Object[]{"BasicType"});
        if(logger.isInfoEnabled()){
            logger.info("Generic invoke result: {}", result);
        }
    }
    /**
     * $genericInvoke调用，用于参数类型无法被当前应用的类加载器加载的场景
     * 泛化调用 People hello(People people);
     */
    @GetMapping("/test/invokeCustomTypeMethod")
    @ResponseBody
    @Produces("application/json;charset=UTF-8")
    public void invokeCustomTypeMethod(){
        // 构造函数中指定全路径类名
        GenericObject genericPeopleObject =new GenericObject(peoplePath);
        // 调用putField，指定field值
        genericPeopleObject.putField("name","Lilei");
        genericPeopleObject.putField("age",15);
        Object result = genericFacade.$genericInvoke(
                "hello",
                newString[]{peoplePath},
        new Object[]{genericPeopleObject});
        // 返回的类型还是GenericObject类型
        if(logger.isInfoEnabled()){
            logger.info("Type of result: {}", result.getClass().getName());
        }
    }
    /**
     * $genericInvoke调用，参数为数组
     * 泛化调用 String[] hello(String[] args);
     */
    @GetMapping("/test/invokeBasicArrayTypeMethod")
    @ResponseBody
    @Produces("application/json;charset=UTF-8")
    public void invokeBasicArrayTypeMethod(){
        String[] results =(String[]) genericFacade.$genericInvoke(
                "hello",
                new String[]{new String[]{}.getClass().getName()},
        new Object[]{new String[]{"BasicArrayType"}});
        // 返回的类型还是GenericObject类型
        if(logger.isInfoEnabled()){
            for(String result : results){
                logger.info("Generic invoke result: {}", result);
            }
        }
    }
    /**
     * $genericInvoke调用，自定义类型数组
     * People[] hello(People[] peoples);
     */
    @GetMapping("/test/invokeCustomArrayTypeMethod")
```

```
        @ResponseBody
        @Produces("application/json;charset=UTF-8")
        public void invokeCustomArrayTypeMethod(){
            GenericObject genericObject =new GenericObject(peoplePath);
            // 调用 putField，指定field值
            genericObject.putField("name","HanMeimei");
            genericObject.putField("age",14);
            // 服务端反射，class.forName对于数组类型的格式有特定要求
            String genericObjArrayType ="[L"+ peoplePath +";";
            GenericObject[] genericObjArray =new GenericObject[]{genericObject};
            GenericArray resultArray =(GenericArray) genericFacade.$genericInvoke("he
llo",
                    new String[]{genericObjArrayType},
            new Object[]{genericObjArray});
            for(Object result : resultArray.getObjects()){
                logger.info(result.toString());
            }
        }
    }
```

如何使用泛化调用？

目前提供了两种方法：

- `$invoke`：仅支持方法参数类型在当前应用的 ClassLoader 中存在的情况。
- `$genericInvoke`：支持方法参数类型在当前应用的 ClassLoader 中不存在的情况。

具体使用，示例如下：

- 服务方：
  - 服务引用，示例如下：

```
<!-- 引用 BOLT 服务 -->
<sofa:referenceinterface="com.alipay.sofa.rpc.api.GenericService"id="genericService">
    <sofa:binding.bolt>
        <sofa:global-attrsgeneric-interface="com.alipay.test.SampleService"/>
    </sofa:binding.bolt>
</sofa:reference>
```

  - 服务端服务定义，示例如下：

```
/*** Java Bean*/
public class People{
    private String name;
    private int    age;
    // getters and setters
}
/** * 服务方提供的接口 */
interface SampleService{
    String hello(String arg);
    People hello(People people);
}
```

- 客户方：
  - 泛化调用，示例如下：

```
    /** * 消费方测试类。 */
    public class ConsumerClass{
        GenericService genericService;
        public void do(){
            // $invoke 仅支持方法参数类型在当前应用的 ClassLoader 中存在的情况。
            genericService.$invoke("hello",new String[]{String.class.getName()},new Obj
ect[]{"I'm an arg"});

            // $genericInvoke 支持方法参数类型在当前应用的 ClassLoader 中不存在的情况。

            //构造参数。
            GenericObject genericObject =new GenericObject("com.alipay.sofa.rpc.test.ge
neric.bean.People");//构造函数中指定全路径类名
            genericObject.putField("name","Lilei");// 调用 putField，指定 field 值。
            genericObject.putField("age",15);
            // 进行调用，不指定返回类型，返回结果类型为 GenericObject。
            Object obj = genericService.$genericInvoke("hello",new String[]{"com.alipay
.sofa.rpc.test.generic.bean.People"},new bject[]{ genericObject });
            Assert.assertTrue(obj.getClass()==GenericObject.class);
            // 进行调用，指定返回类型。
            People people = genericService.$genericInvoke("hello",new String[]{"com.ali
pay.sofa.rpc.test.generic.bean.People"},new Object[]{ genericObject },People.class);

            // LDC 架构下的泛化调用使用。
            // 构造 GenericContext 对象。
            AlipayGenericContext genericContext =new AlipayGenericContext();
            genericContext.setUid("33");
            // 进行调用。
            People people = genericService.$genericInvoke("hello",new String[]{"com.ali
pay.sofa.rpc.test.generic.bean.People"},new Object[]{ genericObject },People.class, gen
ericContext);
}
```

> **重要**
>
> 调用 `$genericInvoke(String methodName, String[] argTypes, Object[] args)` 接口，会将除 `com.sun`、`java`、`javax`、`org.ietf`、`org.ogm`、`org.w3c`、`org.xml`、`sunw.io`、`sunw.util` 包以外的其他类序列化为 GenericObject。

泛化调用的使用场景有哪些？

泛化调用提供了让客户端，在不需要依赖服务端接口的情况下，也能发起调用的能力。在 Bolt 通信协议下使用 Hessian2 作为序列化协议，是目前 SOFARPC 的泛化调用仅支持的方式。

泛化调用的常见场景：

在开发中遇到一些第三方应用不想要依赖我们自己开发的依赖接口 JAR，但也想通过某种方式发起调用，或者更进一步，做一个非依赖 JAR 的简单微服务网关。

如何针对 RPC 请求做一些定制化处理，比如白名单过滤？

可通过过滤器的方式进行 RPC 接口过滤，比如 IP 黑白名单的过滤、Token 的验证等。

白名单过滤的实现步骤如下：

1. 继承 SOFA 的 Filter 抽象类，实现里面的 invoke 方法：

```
    @Component
    public class WhiteIpFilter extends Filter{
        @Value("${security.firewall.whiteIps}")
        private String whiteIpList;

        @Override
        public boolean needToLoad(FilterInvoker invoker){
               return true;
        }
        @Override
        public SofaResponse invoke(FilterInvoker invoker,SofaRequest request) throws So
faRpcException{
            RpcInternalContext context =RpcInternalContext.getContext();
            InetSocketAddress remoteAddress = context.getRemoteAddress();
            finalString remoteIp = remoteAddress.getHostString();
            if(whiteIpList.contains(remoteIp)){
                return invoker.invoke(request);
            }else{
                SofaResponse sofaResponse =new SofaResponse();
                sofaResponse.setErrorMsg("非法IP: "+ remoteIp +" 访问，请联系管理员。");
                return sofaResponse;
            }
        }
    }
```

2. 在需要白名单验证的接口上配置过滤器：

```
<sofa:service interface="cloud.provider.facade.CallerService" ref="callerService">
   <sofa:binding.bolt>
        <sofa:global-attrs filter="whiteIpFilter"/>
   </sofa:binding.bolt>
   <sofa:binding.rest/>
</sofa:service>
```

是否有文件上传下载的示例代码？

SOFARPC 的 REST 协议底层使用的是 Resteasy，可以实现文件上传下载。

主要步骤如下：

1. 声明 Facade 接口：

```
public interface FileServiceFacade{
   @GET
   @Path("/files/{fileName}")
   @Produces("text/plain")
   Response downloadFile(@PathParam("fileName")String fileName) throws Exception;
   @POST
   @Path("/files")
   @Consumes("multipart/form-data")
   Response uploadFile(MultipartFormDataInput input) throws IOException;
}
```

2. 实现下载方法：

```
@Ocerride
    public Response downloadFile(String fileName)throws Exception  {
          // 指定一个存放文件的目录。
          final String dir="/destdir/"
          // 判断文件请求是否为空。
          if(fileName == null  || fileName.isEmpty( )) {
                ResponseBuilder  response = Response.status(Status.BAD_REQUEST);
                return response.build( );
}
    // 文件名为 utf-8。
    String utfFlieName = URLDecoder.decode(fileName, "utf-8");
    File file = new File(dir + utfFilename);
   // 判断文件是否存在。
    if (!file.exists( )) {
         ResponseBuilder response = Response.ok((object)file);
         // 设置请求头。
         response.header("Content-Disposition",  "attachment; filename=" + utfFileName)
;
         // 下载响应。
         return response.builde( );
}
```

3. 实现上传方法：

```
@Override
    public Response uploadFile(MultipartFormDataInput input)throws IOException{
        final String UPLOAD_FILE_PATH ="/Users/yuanshaopeng/Desktop/temp/";
        Map<String,List<InputPart>> uploadForm = input.getFormDataMap();
        // httpclient
        // Get file name
        //String fileName = uploadForm.get("fileName").get(0).getBodyAsString();
        // Get file data to save
        //List<InputPart> inputParts = uploadForm.get("attachment");
        // http mode
        List<InputPart> inputParts = uploadForm.get("uploadedFile");
        String fileName ="";
        for(InputPart inputPart : inputParts){
            try{
                @SuppressWarnings("unused")
                MultivaluedMap<String,String> header = inputPart.getHeaders();
                fileName = getFileName(header);
                byte[] bytes =IOUtils.toByteArray(inputPart.getBody(InputStream.class,n
ull));
                log.info("上传文件大小："+ bytes.length);
                File desFile =new File(UPLOAD_FILE_PATH + fileName);
                FileUtils.writeByteArrayToFile(desFile, bytes);
                System.out.println("Success !!!!!");
            }catch(Exception e){
                e.printStackTrace();
            }
        }
        return Response.status(200).entity("Upload file name : "+ fileName).build();
    }

    private String getFileName(MultivaluedMap<String,String> header){
        String[] contentDisposition = header.getFirst("Content-Disposition").split(";")
;
        for(String filename : contentDisposition){
            if((filename.trim().startsWith("filename"))){
                String[] name = filename.split("=");
                String finalFileName = name[1].trim().replaceAll("\"","");
                return finalFileName;
            }
        }
        return "unknown";
    }
}
```

如何实现一个不依赖 Facade 接口的服务网关？

一般实现思路为：

1. 通过通用的 RESTful 协议进行接入。
2. 后端根据泛化接口对传入数据进行解析。
3. 通过代理的过滤器指定路由配置。

具体可以参考 spring-cloud-gateway 的过滤器设计和 SOFABolt 协议的泛化设计。可能需要下述扩展：

- 动态路由
- 注册中心联动
- 缓存设计
- 自动代码生成等

如何实现 RPC 请求携带数据进行传递？

> **重要**
>
> 默认该功能是关闭的，开启后会影响性能，请尽量避免使用。

实现步骤如下：

1. 开启配置。

   在 `resource` 目录下添加 `rpc-config.json` 文件。

   ```
   {"invoke.baggage.enable":true}
   ```

2. 代码实现。

   示例如下：

   ```
   System.out.println(RpcInvokeContext.isBaggageEnable());
   RpcInvokeContext context =RpcInvokeContext.getContext();
   context.putRequestBaggage("hellod","lolo");
   ```

   > **说明**
   >
   > `RpcInvokeContext` 是一个 RPC 执行上下文，在这个上下文中，我们可以向 `RequestBaggege` 里添加数据，且数据必须是字符串类型。

如何设置 SOFARPC 服务暴露的端口？

在 SOFARPC 服务中，协议默认端口的约定为：

- BOLT 协议：12200
- REST 协议：8341

因此，您可以在 `/resources/config/application.properties` 里设置参数来实现服务暴露的端口，示例如下：

```
com.alipay.sofa.rpc.bolt.port=12202
com.alipay.sofa.rpc.rest.port=8765
```

如果一个服务有多个实现，服务在发布和引用的时候该如何处理？

一个服务如果有多个实现，可以在 RPC 暴露服务和引用服务的地方配置一个 `unique-id` 来作为它的唯一标识。示例如下：

```
<!-- 服务一 -->
<sofa:service ref="sampleServiceBean1" interface="com.alipay.APPNAME.facade.SampleService"
unique-id="service1">
     <sofa:binding.bolt/>
</sofa:service>
<!-- 服务二 -->
<sofa:service ref="sampleServiceBean2" interface="com.alipay.APPNAME.facade.SampleService"
unique-id="service2">
    <sofa:binding.bolt/>
</sofa:service>
```

RPC 服务超时控制，有哪些参数可以配置？

RPC服务在发布和引用时都有超时控制的配置，方法也可以做超时控制，其超时时间的优先级为：引用服务的方法超时 > 引用服务的全局超时时间 > 服务发布者的方法超时 > 服务发布者的全局超时时间。示例如下：

```
<sofa:binding.bolt>
     <sofa:global-attrs timeout="5000"/>
     <sofa:method name="message" type="future" timeout="25000"/>
</sofa:binding.bolt>
```

当接口以 RESTful 协议暴露时，需要注意哪些事项？

需要注意下述事项：

- 首先是需要明确中间件使用的版本，目前推荐 sofaboot-enterprise 的版本是 3.4.*，在使用该版本时，如果要使用 RESTful 接口进行开发时，只需要引入 RPC 对应的 starter 包依赖即可，无需再引入 REST 的 starter 依赖，否则会启动两个 RESTful 服务端，导致 8341 端口被占用。
- 在使用 REST 接口开发时，需要在接口上暴露 Path 和 WS 相关的注解，否则服务不知道具体的请求是什么。具体的 WS 注解使用请参见 WS 注解使用。

一个服务如何同时暴露 RESTful 和 Bolt 两种协议？

发布服务时直接同时声明两种协议的 `binding` 即可，如下所示：

```
<bean id="demoServiceImpl" class="com.alipay.sofa.samples.rpc.DemoServiceImpl"/>
<sofa:service ref="demoServiceImpl" interface="com.alipay.sofa.samples.rpc.DemoService">
     <sofa:binding.rest/>
     <sofa:binding.bolt/>
</sofa:service>
```

dev 环境 RPC 调用出现“RPC-02306: Can not get the service address of service”报错

问题现象：

dev 环境 RPC 调用出错，日志 `middleware_error.log` 中出现找不到服务地址 URL 的报错。

问题原因：

属性配置中打开了直连开关，而代码中配置的 URL 与实际 RPC 服务地址不一致。

- `application-dev.properties` 中配置 `run.mode=test` 。
- 代码中配置 `test-url="${servicename_tr_service_ur}"` 。
- `servicename_tr_service_ur` 指向的地址与实际的 RPC 服务地址不符。
- RPC provider 与 consumer 工程的 SOFABoot 版本不一致。

示例如下：

```
<sofa:reference id="[redacted]"
          interface="com[redacted]"
          unique-id="[redacted]">
   <sofa:binding.bolt>
      <sofa:global-attrs timeout="5000" test-url="${[redacted]_tr_service_url}" address-wait-
time="1000"
                  connect.timeout="1000" connect.num="-1" idle.timeout="-1"
idle.timeout.read="-1"/>
   </sofa:binding.bolt>
</sofa:reference>
```

解决方案：

在 `application-dev.properties` 中注释掉 `run.mode=test` ，或者将 RPC provider 与 consumer 工程的 SOFABoot 版本升级至同一版本。详情请参见 SOFABoot 版本说明。

每次 RPC 调用都耗时很长，明显超时却不报超时异常

问题现象：

SOFA RPC 使用 REST 接口触发 RPC 的泛化调用，每次触发都需要 30 秒的时间，且不报超时异常。从业务日志中可以看出，开始处理业务和结束业务之间确实花了 30 秒。

问题原因：

可能由于 DNS 配置错误，导致超时。

解决方案：

在 `/etc/hosts` 中添加 IP 与主机名的映射，尝试解决该问题。

RPC 注册不成功

问题现象

RPC 注册不成功。

问题原因

`application.properties` 文件中的 `run.mode` 设置成了 `dev` 。

解决方案

删除 `run.mode` 配置或者将其设置成 `normal` 。

RPC 应用启动时出现“Can't find BindingConverter of type binding.tr”报错

问题现象：

RPC 应用启动时出现如下报错：

```
Causedby: org.springframework.beans.factory.BeanCreationException:Error creating bean with name 'secretFacade':Invocation of init method failed; nested exception is com.alipay.sofa.runtime.api.ServiceRuntimeException:Can't find BindingConverter of type binding.tr
```

问题原因：

`rpc-enterprise-sofa-boot-starter` 被注释掉了，而这个 JAR 包提供了如下 binding：

- `rpc-enterprise-sofa-boot/3.2.2/rpc-enterprise-sofa-boot-3.2.2.jar!/com/alipay/boot/sofarpc/converter/TrBindingConverter.class`
- `rpc-sofa-boot/3.2.2/rpc-sofa-boot-3.2.2.jar!/com/alipay/sofa/rpc/boot/runtime/converter/BoltBindingConverter.class`

解决方案：

引入 `rpc-enterprise-sofa-boot-starter` JAR 包。

Tr 接口找不到服务地址

问题原因：

仅客户端迁移至了共享中间件，服务端并未迁移。

解决方案：

将服务端迁移到共享中间件。

RPC 出现重复发布的报错

问题现象：

具体报错信息如下：

```
java.util.concurrent.ExecutionException: com.alipay.sofa.rpc.core.exception.SofaRpcRuntimeE
xception: RPC-010010014: KEY 为 [bolt://com.aliyun.fsi.insurance.aboss.rule.facade.AbossRul
eFacade:] 的 Consumer config 重复发布超过了 [3] 次.也许这是一个错误的配置导致的,请检查.
  at java.util.concurrent.FutureTask.report(FutureTask.java:122)
  at java.util.concurrent.FutureTask.get(FutureTask.java:192)
  at SingelFormula.main(SingelFormula.java:47)
Caused by: com.alipay.sofa.rpc.core.exception.SofaRpcRuntimeException: RPC-010010014: KEY
为 [bolt://com.aliyun.fsi.insurance.aboss.rule.facade.AbossRuleFacade:] 的 Consumer config
重复发布超过了 [3] 次.也许这是一个错误的配置导致的,请检查.
  at com.alipay.sofa.rpc.bootstrap.DefaultConsumerBootstrap.refer(DefaultConsumerBootstrap.
java:136)
  at com.alipay.sofa.rpc.config.ConsumerConfig.refer(ConsumerConfig.java:926)
  at SingelFormula$RuleSingle.call(SingelFormula.java:87)
  at SingelFormula$RuleSingle.call(SingelFormula.java:62)
  at java.util.concurrent.FutureTask.run(FutureTask.java:266)
  at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
  at java.util.concurrent.FutureTask.run(FutureTask.java:266)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
  at java.lang.Thread.run(Thread.java:748)
```

解决方案：

出现这种问题，一般是重复多次注册服务导致，建议您检查服务的服务注册，删除多余的注册信息。

RPC 单次传输的数据量是否有限制

RPC 单次传输的数据量本身没有限制，但基于性能考虑，建议设置为 4 KB 以内。如果数据量超过限制，在高并发场景下，可能出现一些 overflow 的问题，报错关键字为 `maybe write overflow` 。建议通过以下系统参数计算实际需要的大小：

参数	默认值
`-Dbolt.netty.buffer.low.watermark`	32 × 1024
`-Dbolt.netty.buffer.high.watermark`	64 × 1024

限流支持哪些客户端场景？

目前主要支持以下几种客户端：

- Spring MVC
  - 代码侵入：无
  - 限流方法：Web URL
- SOFA RPC Bolt
  - 代码侵入：无

- 限流方法：接口方法

- 普通 Spring Bean
  - 代码侵入：结合 AOP
  - 限流方法：接口方法

不支持以下客户端：

- SOFA RPC REST
  - 代码侵入：无
  - 限流方法：接口方法 、Web URL
- SOFA REST（RESTEASY）
  - 代码侵入：无
  - 限流方法：接口方法、Web URL

  对于SOFA REST，目前只能通过 AOP 的方式去拦截 REST 对应的 Bean 来实现限流，步骤如下：

  i. 设计要拦截的接口方法，示例如下：

```
@Path(URLConstants.REST_API_PEFFIX +"/users")
@Consumes(RestConstants.DEFAULT_CONTENT_TYPE)
@Produces(RestConstants.DEFAULT_CONTENT_TYPE)
public interface SampleRestFacade{
    @GET
    @Path("/{userName}")
    public RestSampleFacadeResp<DemoUserModel> userInfo(@PathParam("userName") String
userName)throws CommonException;
}
```

  ii. 定义 bean，示例如下：

```
<bean id="sampleRestFacadeRest" class="com.hula.sofa.demos.guardian.endpoint.impl.Sam
pleRestFacadeRestImpl"/>
```

  iii. 配置 AOP，示例如下：

```
<import resource="classpath:META-INF/spring/guardian-sofalite.xml"/>
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
      <property name="interceptorNames">
            <list>
                <value>guardianExtendInterceptor</value>
            </list>
      </property>
     <property name="beanNames">
            <list>
                 <!-- 配置需要被拦截的 bean -->
                 <value>sampleRestFacadeRest</value>
            </list>
     </property>
    <!-- 如要使用 CGLIB 代理，取消下面这行的注释 -->
    <!-- <property name="optimize" value="true" /> -->
</bean>
```

iv. 配置限流，示例如下：

SOFARPC 在 DEV 模式下，生成的缓存地址文件在什么位置？

DEV 模式不会向注册中心进行注册，只会在本地生成一个临时文件。您可以在 `/logs/rpc/rpc-registry.log` 中查找 `Write backup file to` 字段，确认临时文件的名称。此文件只会在服务端生成。

如何检查项目已经注册成功或启动成功？

您可以通过以下方式进行检查：

- 在项目服务器上，通过 `ps -ef | grep java` 命令查看 Java 进程。如果进程存在，则表示项目已启动；反之，则没有启动。
- 在项目服务器上，执行以下命令，查看返回结果：
  - RPC 版本为 3.0 及以上

    ```
    curl http://localhost:8080/actuator/readiness
    ```

    8080 为项目端口，您需要替换为实际的项目端口。
  - RPC 版本为 3.0 以下

    ```
    curl http://localhost:8080/health/readiness
    ```

  如果返回的结果中没有 `down` 字段，则表示项目已启动；反之，则没有启动。
- 在项目服务器上，通过 `ps -ef | grep 9600` 命令检查 9600 端口是否正常。如果端口存在，则表示注册中心已连接；反正，则表示没有正常连接。

RPC 调用超时

排查步骤如下：

1. 检查日志文件 `rpc-server-digest.log` 。

   确认出现调用超时问题的时间段内，服务提供方处理时间是否正常。

2. 检查异常日志的下一行的是否存在相同 traceid 的日志，确认是否存在其他调用。
   - 如果存在，则排查同一个 traceid 下的其他调用是否有问题。
   - 如果不存在，则建议您优化自身业务的功能。