蚂蚁金服金融科技产品手册

SOFAStack 微服务

产品版本： V2.30.0

文档版本： V20200403

蚂蚁金服金融科技文档

商标声明

及其他蚂蚁金服金融科技服务相关的商标均为蚂蚁金服金融科技所有。
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。蚂蚁金服金融科技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，并在蚂蚁金服金融科技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金服金融科技授权渠道下载、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。

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。使用微服务框架开发应用，在应用托管后启动应用，微服务会自动注册到服务注册中心，您可以在微服务控制台进行服务管理和治理的相关操作。微服务主要提供分布式应用常用解决方案，包含 RPC 服务、动态配置、限流熔断等。

服务注册中心

服务注册中心（SOFARegistry）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。SOFARPC 提供服务发布与订阅、服务调用、服务路由、服务限流、服务管控、服务链路跟踪等一系列稳定实用的功能。

应用依赖

提供对应用 RPC 发布订阅服务的实时分析结果，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。

动态配置

提供应用运行时动态修改配置的服务，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推送值，可实时查看接入动态配置的客户端应用节点的内存值。

限流熔断

提供对业务系统的限流服务，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。

1.2 产品优势

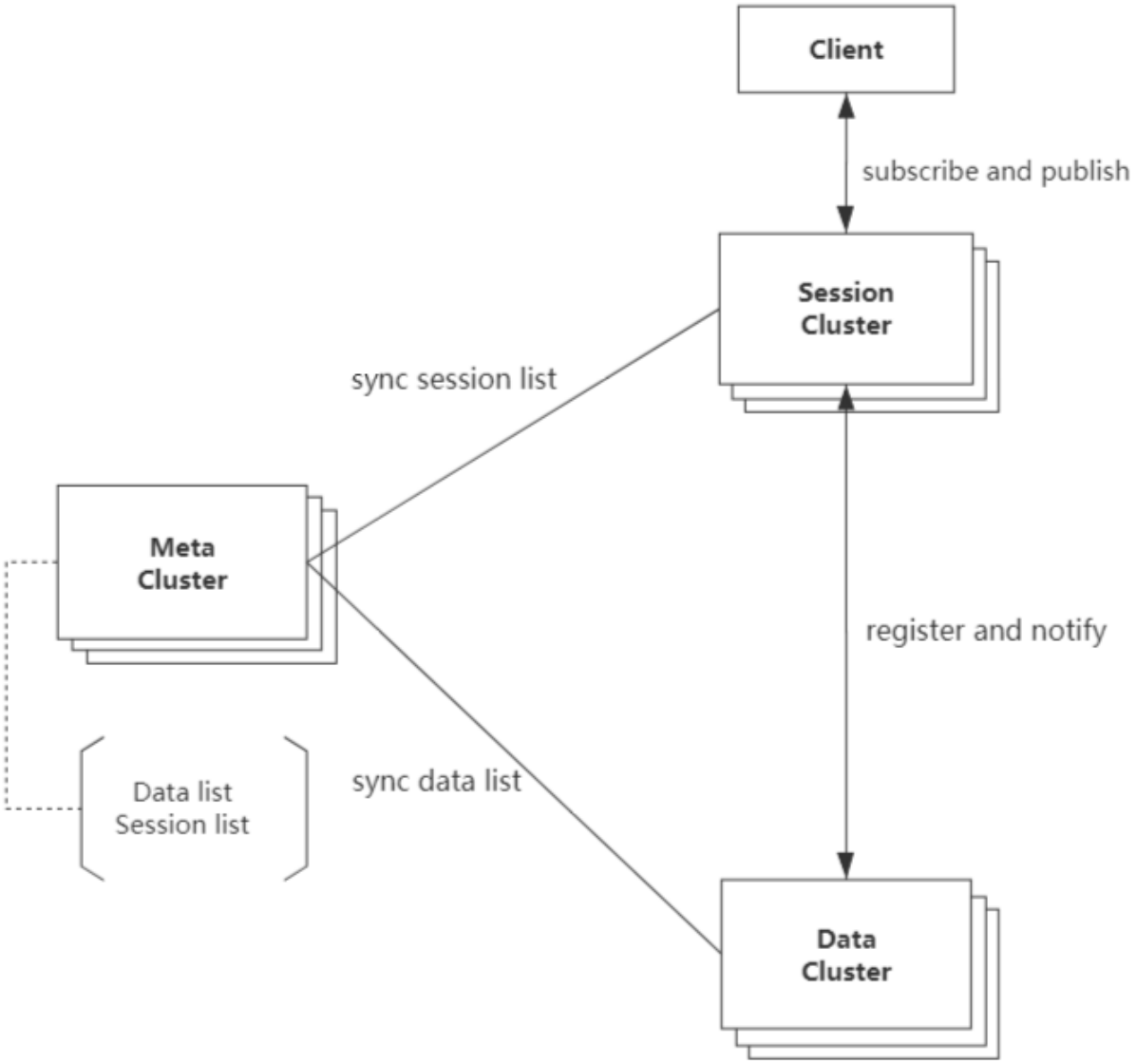
微服务产品提供金融级分布式架构的基础设施能力，包括 RPC 框架及服务治理、服务注册与发现、动态配置、定时任务、服务限流等，为传统单体应用架构深入拆分为分布式应用架构提供稳定可靠的基础设施能力，帮助企业级客户快速构建基于微服务架构的分布式应用，从而实现更灵活地响应业务变化，提高系统的可扩展性及性能。

微服务产品在蚂蚁金服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。

1.3 产品架构

SOFARegistry

SOFARegistry 服务注册中心分为四个角色：客户端（Client）、会话服务器（SessionServer）、数据服务器（DataServer）、元数据服务器（MetaServer），每个角色司职不同能力组合后共同提供对外服务能力，各部分关系和结构如下：

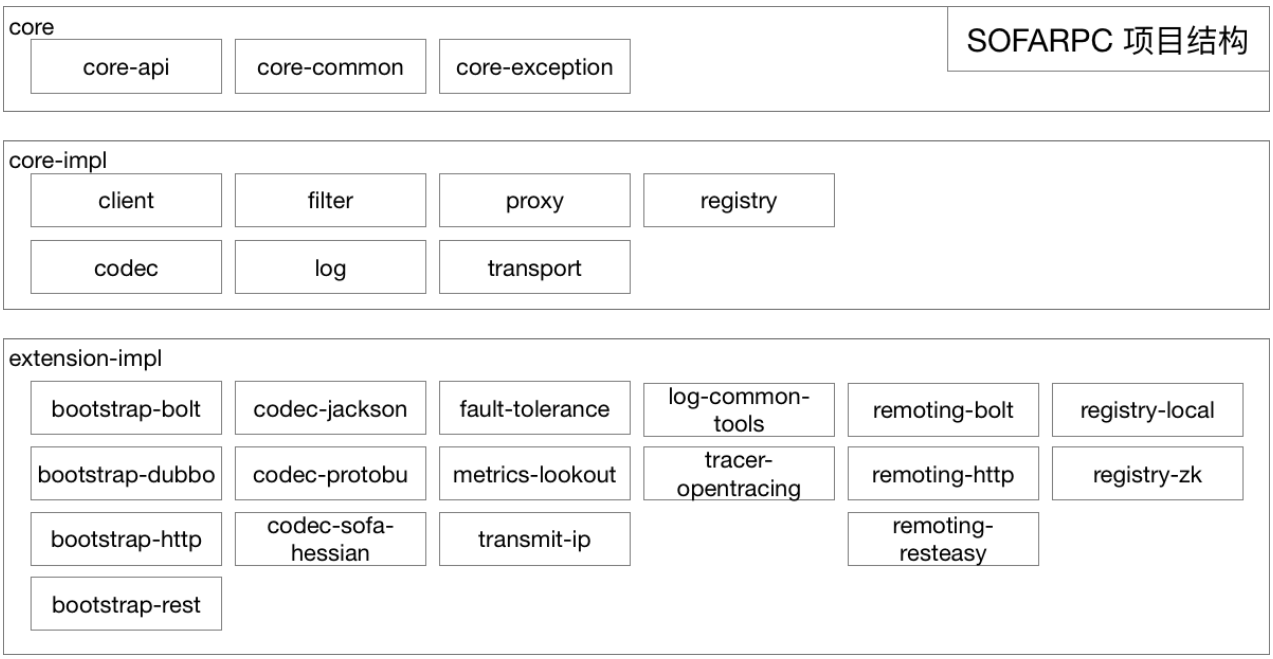


SOFARegistry 服务注册中心的主要组件有：

- Client：提供应用接入服务注册中心的基本 API 能力，应用系统通过依赖客户端 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 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量。
- 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

目前的 RPC 框架结构如下：



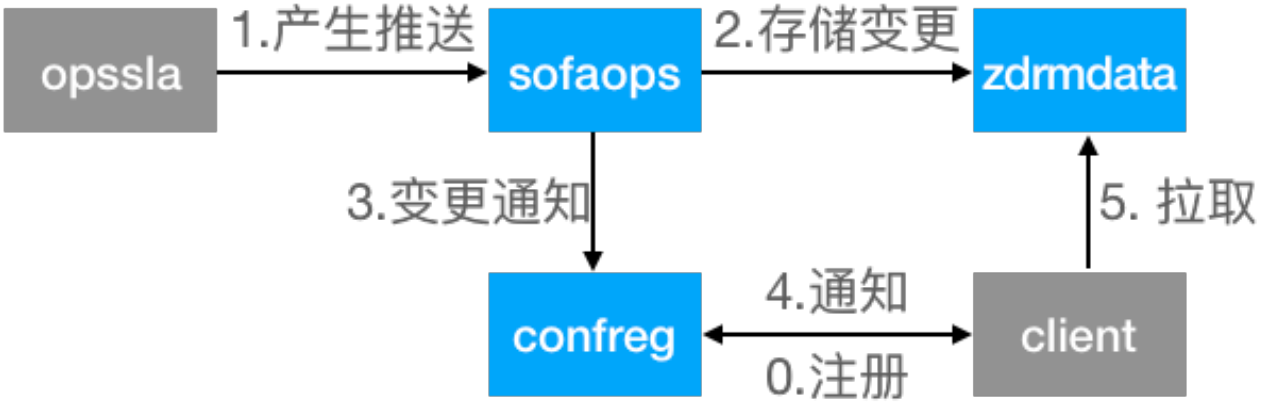
主要分为以下两部分：

- core 和 core-impl 是核心的功能，包含 API 和一些扩展机制。
- extension-impl 则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。

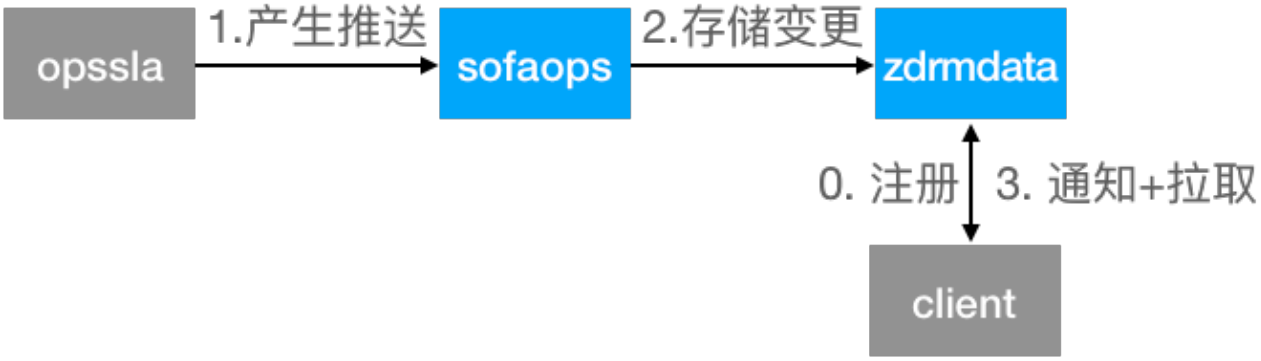
动态配置（DRM）

DRM3 架构设计

DRM3 的具体配置值并不通过服务注册中心推送，而是增加了一个数据存储系统 zdrmdata，通过它存入数据库中。只通过服务注册中心推送一个特定格式的简短指令，应用接收到该指令后，知道配置项发生了变更，向 zdrmdata 发起 HTTP 请求，读取真正的配置值。过程如下图所示。

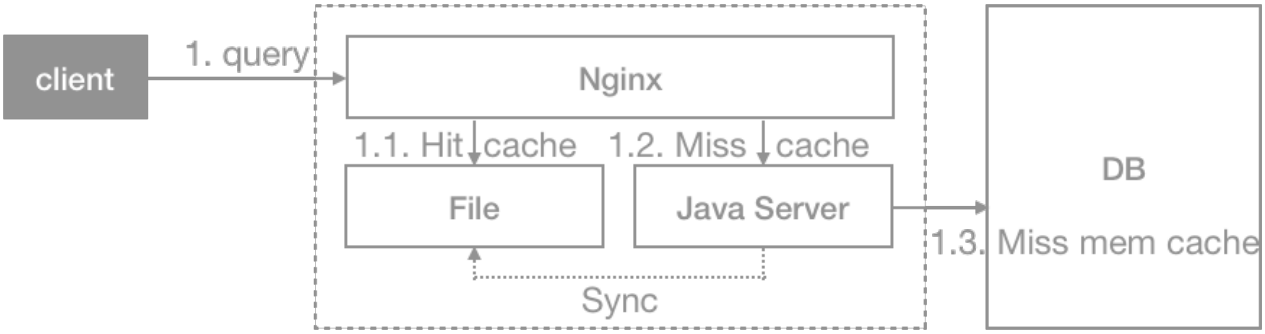


上图的整个交互路径适用与 DRM Client 3.7.0 以下，3.7.0 以上客户端简化了注册模型，不再依赖 Confreg 做推送，而是将整个逻辑在 DRM 的内部自包含掉，3.7.0 以上客户端的推送逻辑如下：



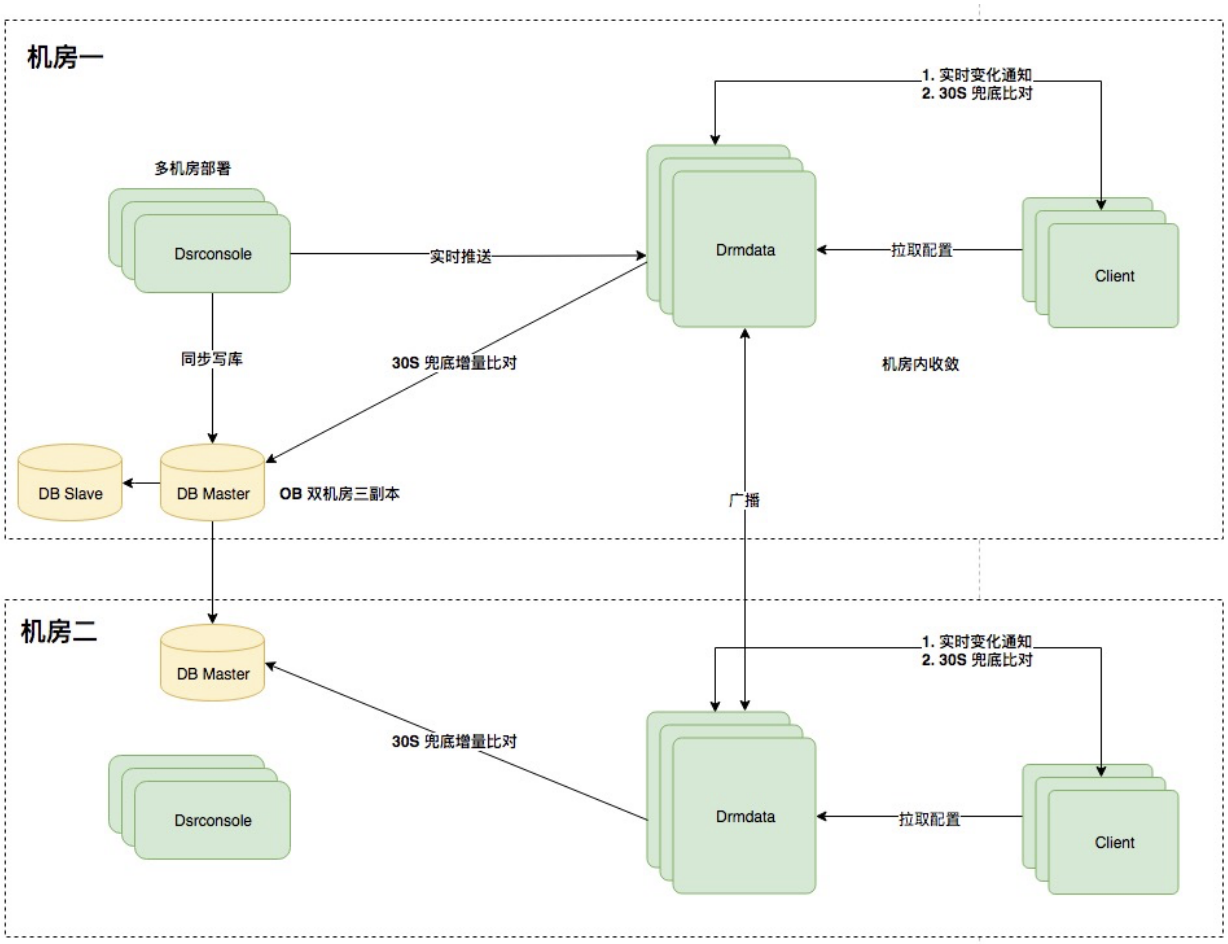
上图路径中的 sofaops 的保留是为了兼容旧客户端使用 Confreg 订阅，终态会去除此中间节点。每次推送的数据，都会有分配的自增版本号。客户端到 zdrmdata 取数据时，也会携带版本号参数，确保不会获取到过期的缓存数据。

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，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，zdrmdata 采用多级缓存提升读取效率，数据拉取时首先经过 zdrmdata 的前置 Nginx，Nginx 的共享内存中缓存有配置值的版本，当版本命中时请求将被转化为读取磁盘上持久化的配置值文件，当缓存未命中时，请求会透传到 Java Server，再经过 Java Server 的内存缓存过滤一次，未命中请求才可能请求到 DB。



DRM3 架构和推送流程

以下是 DRM3 的架构图：

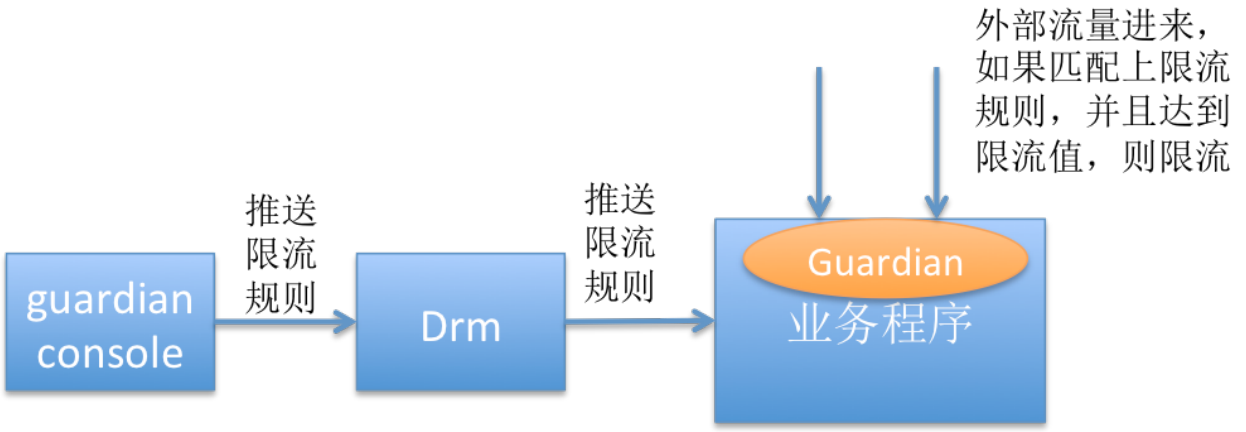


DRM3 完整的推送流程是：

1. 控制台页面上，点击 **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 version。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。

限流熔断（Guardian）

Guardian 的架构如下图所示：



Guardian 运作主要分成两条主线：

1. 控制台 GuardianConsole 推送限流规则：
   业务程序集成了 Guardian 之后，用户在控制台 GuardianConsole 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。
2. 匹配流量，判断是否需要限流：
   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，如果匹配上，并且达到了预设的限流值，则限流。

1.4 功能特性

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（默认自由协议）、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

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，抢占市场待机，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自研二进制通信协议，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，在蚂蚁金服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

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 1.7 及以上	JDK 版本支持 1.7 及以上

1.7 基础术语

SOFARegistry

中文	英文	释义
SOFARegistry	SOFARegistry	蚂蚁金服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。
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	MetaServever	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	DataCenter	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）
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。
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。

限流熔断

中文	英文	释义
运行模式	running mode	指限流熔断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 产品定价

有关 SOFAStack 微服务的收费、出账、欠费及退款相关的规则，详见 产品定价 > 微服务平台 。

3 快速入门

微服务提供分布式任务常用解决方案，支持在线配置、管理、监控 SOFA 应用。在 SOFA 应用接入不同的微服务组件后，您可以在微服务控制台页面完成对各组件功能的配置，包括 RPC 服务、动态配置、限流熔断等。

前置条件

在开始使用微服务之前，您必须配置好一个 SOFABoot 工程，并添加安全配置及引入相应的中间件 Maven 依赖。

操作步骤如下：

1. 创建并配置好一个 SOFABoot 工程。具体步骤，参见 创建 SOFABoot 工程 。

   打开全局配置文件 application.properties 配置文件，添加以下配置项：

   - com.alipay.instanceid：应用实例在工作空间中的唯一标识。您可以前往 **脚手架 > Step 2 > 实例标识** 获取。

     com.antcloud.antvip.endpoint：应用通过 AntVIP 指来获取各个组件的服务端地址。每个区域一个地址。您可以前往 **脚手架 > Step 2 > AntVIP** 获取。

     com.antcloud.mw.access 、com.antcloud.mw.secret 是访问中间件的身份验证密钥，可前往 RAM 控制台 获取。详见 创建 AccessKey。

3. 在 SOFABoot 工程的 pom.xml 文件中引入微服务模块相应的 Maven 依赖。
   - SOFARPC

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

   - 动态配置

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

- 限流熔断

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

关于中间件引入的更多信息，参见 引入 SOFA 中间件 。

本地开发

完成上述步骤后，您需要进行相应模块的本地开发与配置，并通过微服务控制台进行微服务的应用管理。

在本地开发环境运行 SOFABoot 应用，存在以下两个限制：

> 只能以 IP 直连的方式进行多个应用间的 SOFARPC、SOFAREST 间互相通信。
> **建议您直接下载 示例工程 体验 SOFARPC 与 SOFAREST，**示例工程的详细说明可参考 开始使用 SOFARPC 。
>
> 暂无法体验动态配置、限流熔断服务。因为本地开发环境没有本地注册中心服务，且由于网络问题无法连通阿里云 SOFAStack 服务集群。
>
> > **说明**：后续将会开放一个公网环境的 SOFAStack 体验 Region，到时即可在本地体验所有 SOFAStack 服务。

微服务各模块开发，具体操作可参见：

- SOFARPC： 开始使用 SOFARPC 。
- SOFAREST： 开始使用 SOFAREST 服务 。
- 动态配置： 开始使用动态配置 。
- 限流熔断： 开始使用限流熔断 。

4 SOFARegistry

4.1 概述

SOFARegistry 是蚂蚁金服开源的一个生产级、高时效、高可用的服务注册中心。SOFARegistry 最早源自于淘宝的 ConfigServer，十年来，随着蚂蚁金服的业务发展，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金服的自有业务，还随着蚂蚁金融科技服务众多合作伙伴，同时也兼容开源生态。SOFARegistry 采用 AP 架构，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。

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”。经受过蚂蚁金服生产环境海量节点数与服务数的考验。

高时效性

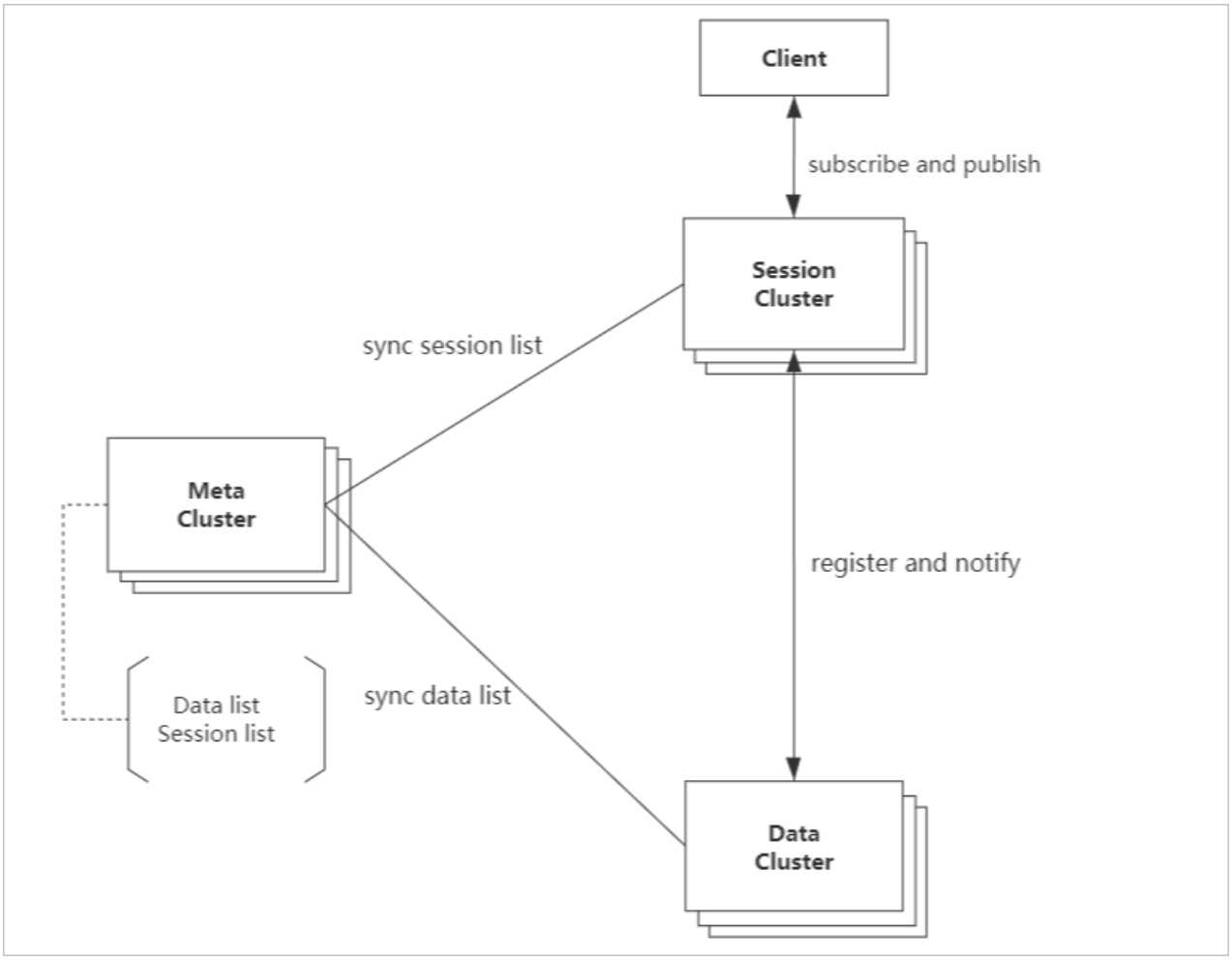
借助 SOFABolt 通信框架，实现基于TCP长连接的节点判活与推模式的变更推送，服务上下线通知时效性在秒级以内。

高可用性

不同于 Zookeeper、Consul、Etcd 等 CP 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。

架构

服务注册中心分为四个角色，客户端（Client）、会话服务器（SessionServer）、数据服务器（DataServer）、元数据服务器（MetaServer），每个角色司职不同能力组合后共同提供对外服务能力，各部分关系和结构如下：



Client

提供应用接入服务注册中心的基本 API 能力，应用系统通过依赖客户端 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 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量。

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。

4.2 基本原理

为什么需要分 meta、data、session 三个角色？

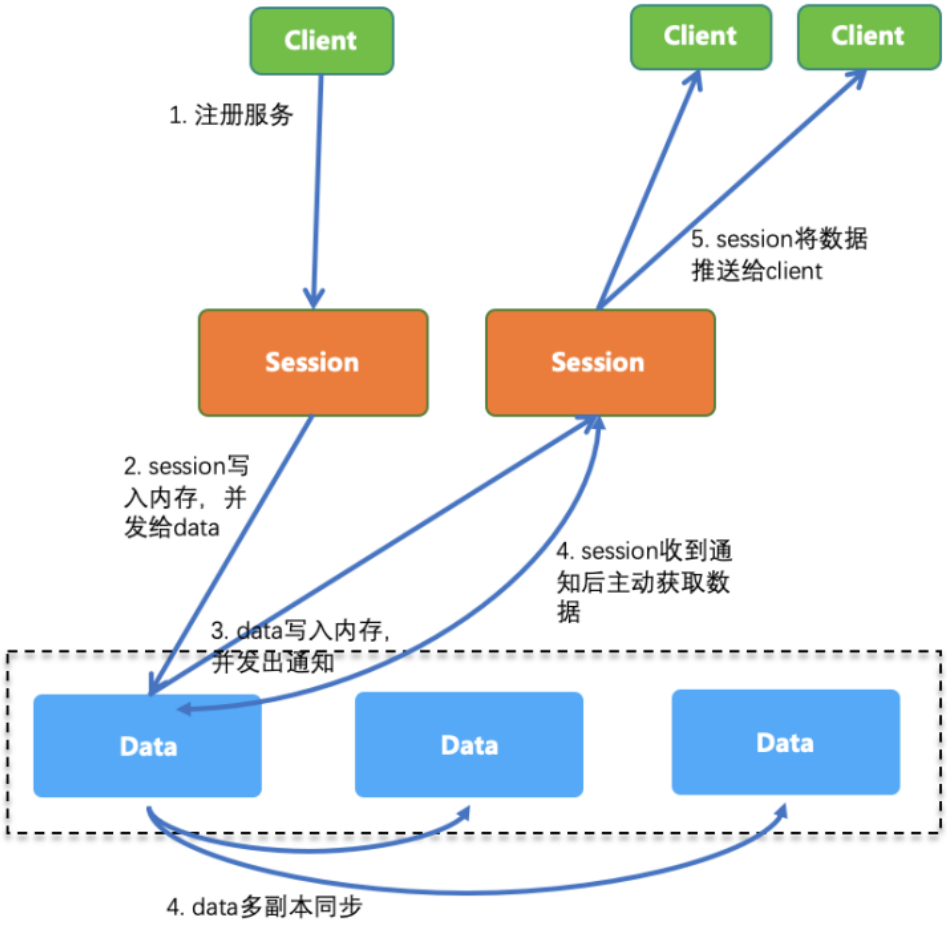
三个角色的职责分别如下：

- session：主要是跟客户端建连
- data 是用来存储数据的
- meta 是用来管理集群元数据的

三个角色的详细作用如下：

1. 首先 session 与 data 的分离是为了同时突破连接数和存储的瓶颈；
2. 其次引入 meta 是为了运维简单，因为角色的多样化让服务注册中心有状态，使用 meta 管理这些元数据，就无需引入第三方组件管理这些元数据；
3. 同时 meta 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。

一次服务注册的过程

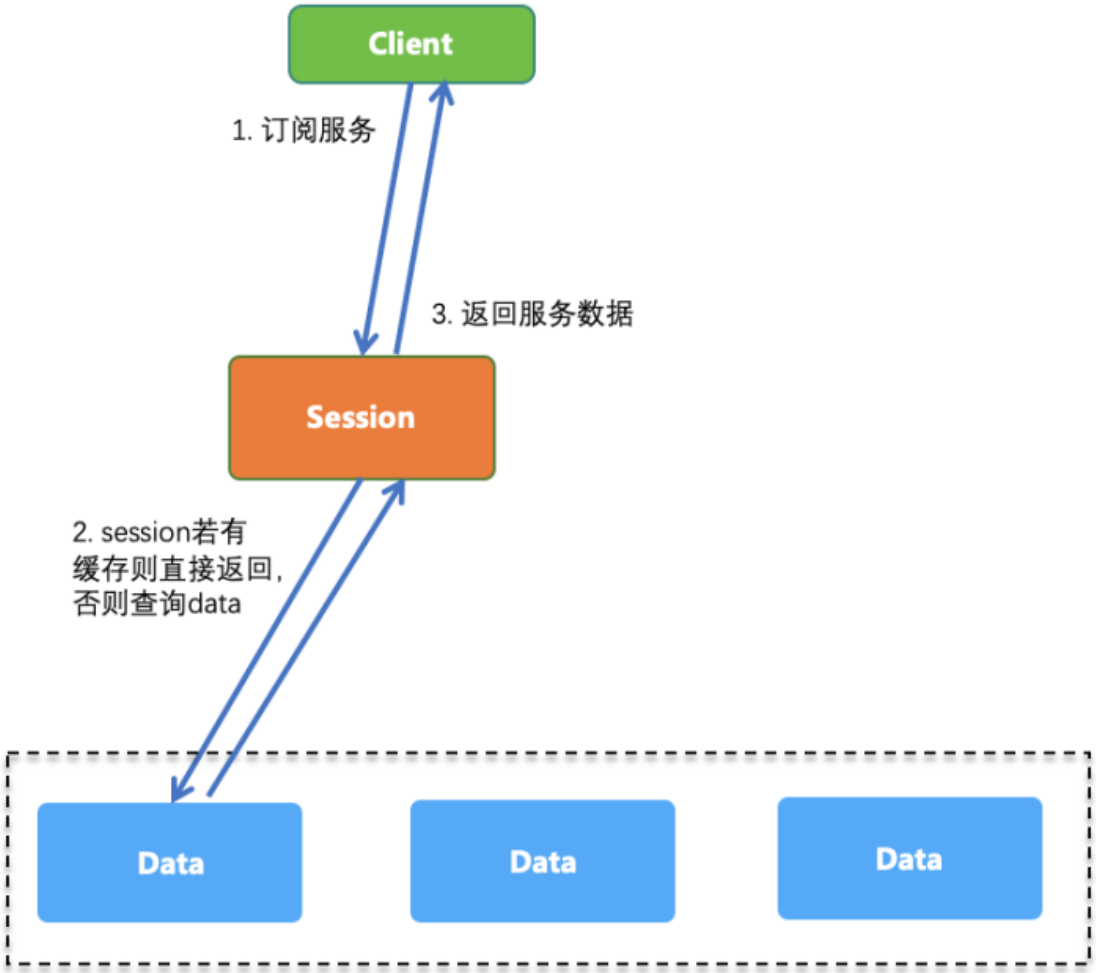


1. Client 发起服务注册数据 Publisher 给 SessionServer。
2. SessionServer 接收到 Publisher 数据后，首先写入内存（ Client 发送过来的 Publisher 数据

，SessionServer 都会存储到内存，用于后续可以跟 DataServer 做定期检查），然后将 Publisher 数据发送给 DataServer。

3. DataServer 接收到 Publisher 数据后，首先也是将数据写入内存（ DataServer 会以 dataInfoId 的维度汇总所有 SessionServer 的数据）。
4. 然后 DataServer 一方面需要将数据同步给副本，因为 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）；同时，SessionServer 将数据的变更事件通知给所有 SessionServer（事件内容是 id 和版本号信息: <dataInfoId> 和 <version>）。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 的数据，即获取具体的 Publisher 列表数据。
5. SessionServer 获取到 dataInfoId 的数据后，将数据推送给相应的 Client，Client 就接收到这一次服务注册之后的最新 Publisher 列表数据。

一次服务订阅的过程



1. Client 发起服务订阅请求 Subscriber 给 SessionServer，Subscriber 主要包含 dataInfoId，表示需要订阅 哪个 dataInfoId 的服务数据。

2. SessionServer 接收到 Subscriber 订阅请求后，首先写入内存（Client 发送过来的 Subscriber 数据，SessionServer 都会存储到内存，用于实现数据变更推送的功能），然后尝试从缓存里获取相应 dataInfoId 的数据，若无则向 DataServer 发起请求同步的获取一次 dataInfoId 数据。
3. SessionServer 将 dataInfoId 的数据(即 Publisher 列表) 发送给 Client。

4.3 SOFARPC 使用 SOFARegistry

概念

服务路由

RPC 最重要的是获取对端地址，SOFARPC 采用服务发布和引用模型，通过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，最后通过负载均衡算法为消费方选择可用地址进行远程通信。

服务注册中心

服务注册中心是 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。

软负载

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

指定调用地址

测试环境

服务注册中心让您在使用 SOFARPC 的时候，不用将地址硬编码在代码中，并且通过服务注册中心的服务发现的方案，实现负载均衡。但是，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，您可以使用 SOFARPC 的 test-url 功能。

使用 test-url，只需要在 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即可。

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

说明：

- test-url 仅是提供给线下测试环境使用的一种 RPC 路由机制。一旦配置了 test-url，软负载的逻辑将会失效，请求将会直接发送到 test-url 配置的服务地址。
- 使用该参数需要在 application.properties 中配置 run_mode=TEST。

线上环境

线上环境不推荐使用 run_mode=TEST 配置来使 test-url 生效，如果在线上环境中部分 reference 也有指定调用地址的需求，请使用以下配置：

```
<sofa:reference id="sampelService"interface="com.alipay.test.SampleService">
<sofa:binding.bolt>
<sofa:route target-url="target-url:12200"/>
</sofa:binding.bolt>
</sofa:reference>
```

说明：

- 如果配置了 target-url，软负载阶段将会失效。
- 如果同时配置了 run_mode=TEST & test-url 和 target-url，将会直接使用 test-url 的配置。

4.4 Spring Cloud 使用 SOFARegistry

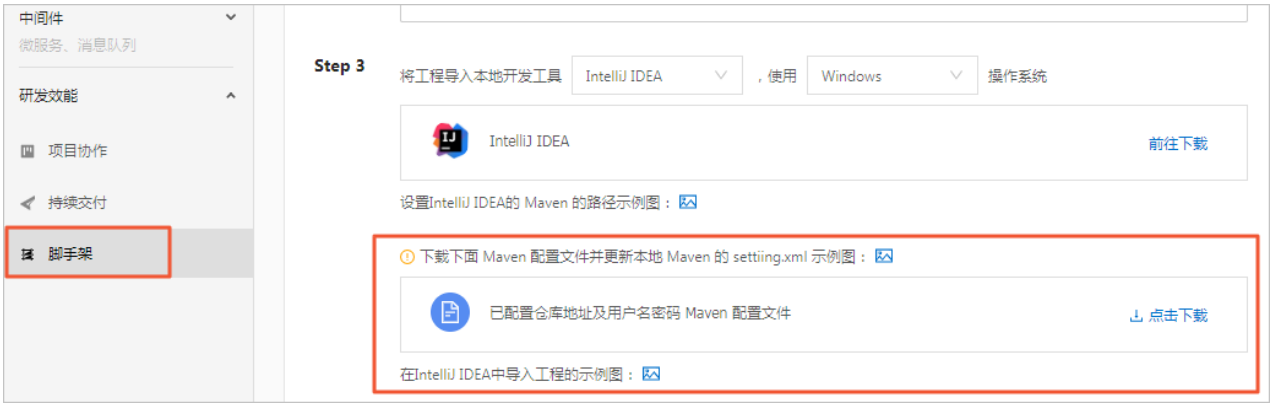
本教程介绍如何将改造本地 Spring Cloud 工程，将其接入 SOFA 服务注册中心。

前置条件

在进行开发前，您需要确认本地 Maven 配置文件 ~/.m2/settings.xml 中已配置 mvn.cloud.alipay.com 仓库的地址及用户名密码。配置后，工程才可以通过 Maven 获取 mvn.cloud.alipay.com 仓库里注册中心的 JAR 包。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>
```

您可以前往 **SOFAStack > 研发效能 > 脚手架 > Step 3** 控制台页面，直接下载已配置好的 settings.xml，并前往 Maven 安装目录 ~/.m2/ 覆盖原有配置文件。



操作步骤

开发应用

在 pom.xml 文件中，引入以下 SDK 依赖：

说明：需将 sofa-registry-cloud-all 版本号替换为该 SDK 最新版本号，详见 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>
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
<!-- 根据不同的 Spring Cloud 版本，将 x.x.x 替换为对应的 tracer 依赖版本号，详细版本信息参见下方表格 -->
<version>x.x.x</version>
<exclusions>
<exclusion>
<artifactId>config-common</artifactId>
<groupId>com.alipay.configserver</groupId>
</exclusion>
<exclusion>
<artifactId>fastjson</artifactId>
<groupId>com.alibaba</groupId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<artifactId>config-common</artifactId>
<groupId>com.alipay.configserver</groupId>
<version>4.3.2.alipay</version>
</dependency>
```

Spring Cloud 与 Tracer 依赖版本号对应表如下：

Spring Cloud 版本	Spring Boot 版本	Tracer 依赖版本
Greenwich.SRx	SpringBoot 2.x	3.0.4

Fichley.SRx		
Edgware.SRx	SpringBoot 1.x	2.3.5.JST
Dalston.SRx		
Camden.SRx		

通过以下任一方式，添加应用启动参数：

说明：如您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 文件指定参数:

```
sofa:
registry:
discovery:
instanceId: // 需要填写
antcloudVip: // 需要填写
accessKey: // 需要填写
secretKey: // 需要填写
```

- 指定 JVM 启动参数值：

```
-Dsofa.registry.discovery.instanceId= // 需要填写
-Dsofa.registry.discovery.antcloudVip= // 需要填写
-Dsofa.registry.discovery.accessKey= // 需要填写
-Dsofa.registry.discovery.secretKey= // 需要填写
```

- 指定系统环境变量：

```
SOFA_INSTANCE_ID= // 需要填写
SOFA_ANTVIP_ENDPOINT= // 需要填写
SOFA_SECRET_KEY= // 需要填写
SOFA_ACCESS_KEY= // 需要填写
```

参数说明：以上参数（instanceId，antcloudVip，accessKey 及 secretKey）是中间件的全局配置项，参数值均可在 **脚手架** 控制台获取。详见 引入 SOFA 中间件 > 中间件全局配置 。

发布部署应用

> **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。

1. 前往 **经典应用服务** 发布应用，详见 应用部署 。
2. 应用发布后，即可前往 **中间件 > 微服务平台 > 微服务 > 服务管控** 控制台页面查看发布的服务。

查看日志

如需查看应用相关日志，可前往 {user.dir}/logs/tracelog 目录或查看 {user.dir}/logs/spring.log。

4.5 Dubbo 使用 SOFARegistry

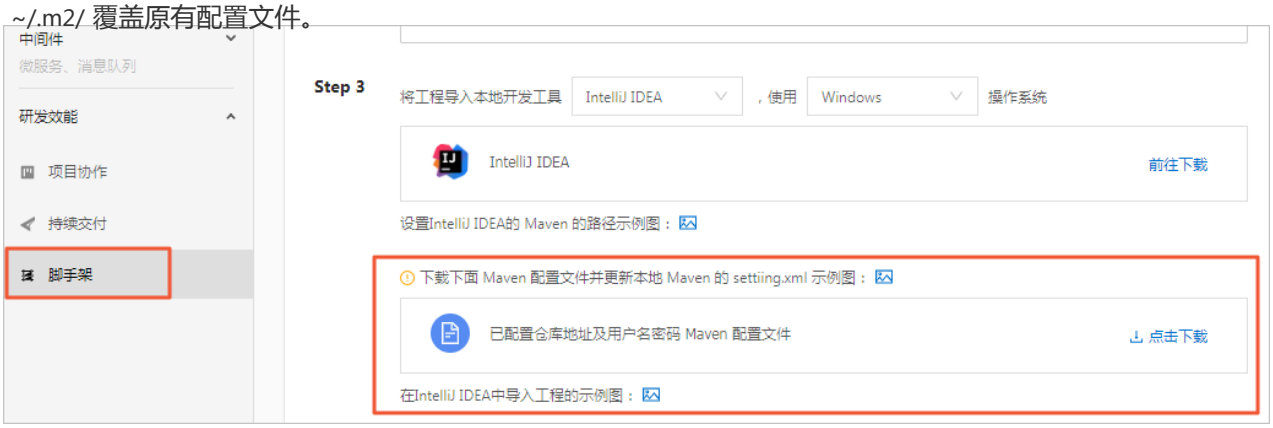
本教程介绍如何将改造本地 Dubbo 工程，将其接入 SOFA 服务注册中心。

前置条件

在进行开发前，您需要确认本地 Maven 配置文件 ~/.m2/settings.xml 中已配置 mvn.cloud.alipay.com 仓库的地址及用户名密码。配置后，工程才可以通过 Maven 获取 mvn.cloud.alipay.com 仓库里注册中心的 JAR 包。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>
```

您可以前往 **SOFAStack > 脚手架 > Step 3** 页面，直接下载已配置好的 settings.xml，并前往 Maven 安装目录 ~/.m2/ 覆盖原有配置文件。



操作步骤

引入依赖

微服务不仅支持使用 sofa-registry-dubbo-all 依赖接入 SOFA 服务注册中心，也支持使用 sofa-registry-cloud-all 依赖接入。

- 通过 sofa-registry-dubbo-all 依赖接入
- 通过 sofa-registry-cloud-all 依赖接入

通过 sofa-registry-dubbo-all 依赖接入

1. 引入注册中心依赖。在主 pom.xml 文件中添加以下依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-registry-dubbo-all</artifactId>
<version>1.0.0</version>
</dependency>
```

2. 配置注册中心寻址参数。在 application.properties 文件中配置以下参数：

```
spring.dubbo.registry.address=dsr://xxx:9003
spring.dubbo.registry.parameters[com.alipay.env]=shared
spring.dubbo.registry.parameters[com.alipay.instanceid]=分配instanceid
spring.dubbo.registry.parameters[com.antcloud.antvip.endpoint]=xxx
spring.dubbo.registry.parameters[com.antcloud.mw.access]=key
spring.dubbo.registry.parameters[com.antcloud.mw.secret]=value
```

参数说明：以上参数（instanceId，antvip.endpoint，access 及 secret）是中间件的全局配置项，参数值均可在 **脚手架** 控制台获取。详见 引入 SOFA 中间件 > 中间件全局配置 。

通过 sofa-registry-cloud-all 依赖接入

在 pom.xml 文件中，引入以下 SDK 依赖：

说明：需将 sofa-registry-cloud-all 版本号替换为该 SDK 最新版本号，详见 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>
<dependency>
```

```
<groupId>com.alipay.sofa</groupId>
<artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
<version>3.0.4</version>
<exclusions>
<exclusion>
<artifactId>config-common</artifactId>
<groupId>com.alipay.configserver</groupId>
</exclusion>
<exclusion>
<artifactId>fastjson</artifactId>
<groupId>com.alibaba</groupId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<artifactId>config-common</artifactId>
<groupId>com.alipay.configserver</groupId>
<version>4.3.2.alipay</version>
</dependency>
```

说明：如果已经使用了依赖 dubbo-sofa-registry，需将其替换成上述的 SDK。

2. 配置 Dubbo 的注册中心，使用 dsr: <dubbo:registry address="dsr://dsr"/>
3. 通过以下任一方式，添加应用启动参数：
    - 在 dubbo.properties 中配置如下：

```
com.alipay.instanceid= // 需要填写
com.antcloud.antvip.endpoint= // 需要填写
com.antcloud.mw.access= // 需要填写
com.antcloud.mw.secret= // 需要填写
```

    - 指定 JVM 启动参数值：

```
-Dcom.alipay.instanceid= // 需要填写
-Dcom.antcloud.antvip.endpoint= // 需要填写
-Dcom.antcloud.mw.access= // 需要填写
-Dcom.antcloud.mw.secret= // 需要填写
```

    - 指定系统环境变量：

```
SOFA_INSTANCE_ID= // 需要填写
SOFA_ANTVIP_ENDPOINT= // 需要填写
SOFA_ACCESS_KEY= // 需要填写
SOFA_SECRET_KEY= // 需要填写
```

    参数说明：以上参数（instanceId，antvip.endpoint，access 及 secret）是中间件的全局配置项，参数值可在 **脚手架** 控制台获取。详见 引入 SOFA 中间件 > 中间件全局配置 。

部署应用

在本地工程完成依赖引入及配置修改后，您需要将微服务发布至 **经典应用服务** 平台。详细发布步骤，参见 应用部署 。

应用发布后，即可前往 **中间件 > 微服务平台 > 微服务 > 服务管控** 控制台页面查看发布的服务。

查看日志

前往 {user.dir}/logs/tracelog/ 目录查看日志。

4.6 服务网格使用 SOFARegistry

使用服务网格，您需要在本地代码中，完成服务注册，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 服务网格连接 SOFA 服务注册中心 。

4.7 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，在微服务控制台无法找到该服务

RPC 服务端发布之后，在微服务控制台无法找到该服务

现象

如题

原因

应用服务器相对应的 IP，不在发布部署参数 rpc_enabled_ip_range 范围中，例如：

- 应用服务器的 IP 是172.19.202.232
- rpc_enabled_ip_range：10:11,172.16,192.168

解决方案

在应用实例的发布部署参数中，将 rpc_enabled_ip_range 更改为：10:11,172.19,192.168，然后重新发布。

5 SOFARPC

5.1 概述

SOFARPC 提供应用之间的点对点服务调用功能，具有高可伸缩、高容错的特性。

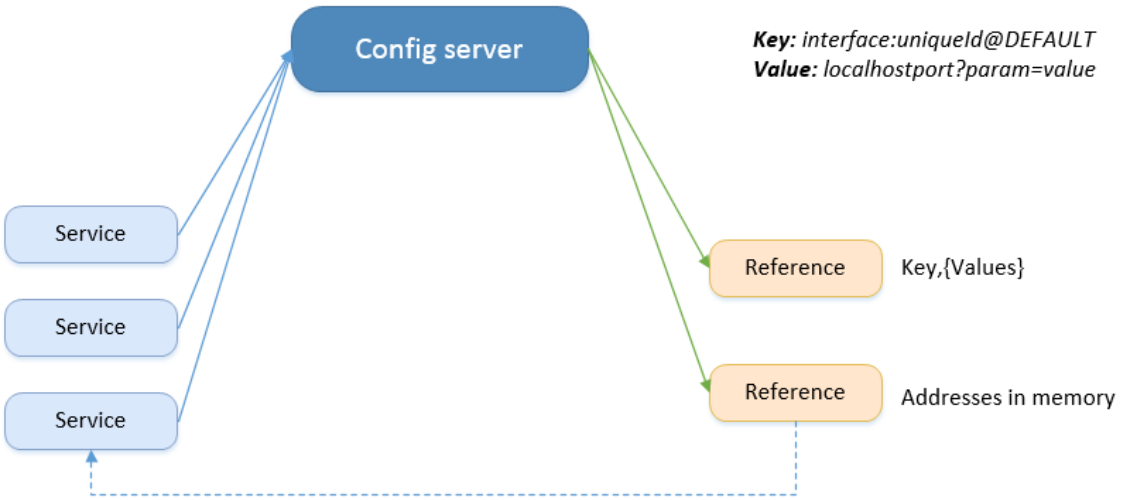
为保证高可用性，通常同一个应用或同一个服务提供方都会部署多份，以达到对等服务的目标。SOFARPC 提供软件负载的能力，它是对等服务调用的调度器，会帮助服务消费方在这些对等的服务提供方中合理地选择一个来执行相关的业务逻辑。

为保证应用的高容错性，需要服务消费方能够感知服务提供方的异常，并做出相应的处理，以减少应用出错后导致的服务调用抖动。在 SOFARPC 中，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。

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和服务提供方，对应 RPC 的调用端和被调用端，可以理解为调用客户端和调用服务端。对于 RPC 服务，服务提供方称之为 “服务(service)” ，而服务消费方称之为 “引用 (reference)” 。

服务发布、引用以及调用的简单流程图如下：



1. 当一个 SOFARPC 的应用启动时，如果发现当前应用需要发布 RPC 服务，那么 SOFARPC 会将该服务注册到配置中心，就是图中蓝色实线所示的过程。
2. 当引用这个服务的 SOFA 应用启动时，会从配置中心订阅对应服务的地址，当配置中心收到订阅请求后，会将发布方的地址列表推送给订阅方，就是图中绿色实线所示的过程。

3. 当引用服务的一方拿到地址以后，就可以调用服务了，就是图中蓝色虚线所示的过程。

5.2 开始使用 SOFARPC

本文将引导您如何通过示例工程快速上手 SOFARPC，实现服务发布、引用，以及应用部署和服务查询。

示例工程

点击此处下载 SOFARPC 示例工程。下载完成后，您可以通过 IDEA 或 Eclipse 将 rpc-demo 中的 2 个工程 myserver-app 和 myclient-app 分别打开。

示例工程服务验证

- 首先启动 myserver-app web 模块的 SOFABootWebSpringApplication 发布服务；
- 然后启动 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: Hello SOFARpc! 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。

其它本地编译方式，可参见 本地编译运行 。

示例工程说明

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。

示例工程配置

SOFARPC 的服务发布与引用配置均需要写到 Spring 的 XML 文件中。以 Web 工程为例，Spring XML 文件的默认路径示例如下：

endpoint 子模块为：

- src/main/resources/META-INF/myserver-app/myserver-app-endpoint.xml
- src/main/resources/META-INF/myclient-app/myclient-app-endpoint.xml

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 XML 配置发现服务。2 个工程通过相同接口实现关联。一个客户端，一个服务端，如果是本机，通过 XML 中配置

testurl 发现服务；如果是服务器上部署测试，走 DSR 底座发现服务。

- 启动 myserver-app web 模块的 SOFABootWebSpringApplication，发布服务。
- 启动 myclient-app web 模块的 SOFABootWebSpringApplication，引用服务。

服务发布

服务发布包含四个步骤，以 endpoint 子模块为例：

设计服务接口类 SampleService.java，代码如下：
接口路径为：com.alipay.samples.rpc.SampleService

```
/**
* 服务接口类
*/
public interface SampleService {
public String hello();
}
```

编写服务实现类 SampleServiceImpl.java， 代码如下：
接口实现路径为：com.alipay.samples.rpc.impl.SampleServiceImpl

```
public class SampleServiceImpl implements SampleService {
private int count = 0;

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

通过以下任一方式在 Spring XML 中配置服务发布的相关参数：

- 新增 XML，配置相关参数。
- 直接在已有的 XML 中新增以下配置。
  使用 SOFABoot 原型 Web 工程时，可以在src/main/resources/META-INF/myserver-app/myserver-app-endpoint.xml 中新增下述配置：

```
<!-- 声明服务的实现对象，以下类全名和接口全名，请根据自己的包名进行指定 -->
<bean id="sampleService"class="com.alipay.samples.rpc.impl.SampleServiceImpl"/>
<!-- RPC 服务发布 -->
<sofa:service ref="sampleService"interface="com.alipay.samples.rpc.SampleService">
<sofa:binding.bolt/>
</sofa:service>
```

运行 web 子模块中的 SOFABootWebSpringApplication，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，示例如下：

```
//*************** 注意 ******************//
//1. 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 myserver-app 的 端口号为 8083
//2. 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，所以需要注释掉这行代码
System.setProperty("server.port","8083");
```

重要：如需将 myserver-app 发布至云上环境，必须注释掉上述代码。

服务引用

通过在 XML 文件中配置参数和引用对象注入，即可实现服务引用。

配置参数

您可以通过以下任一方式配置参数：

- 新增 XML，配置相关参数。
- 直接在已有的 XML 中新增以下配置。
  使用 SOFABoot 原型 Web 工程时，可以在src/main/resources/META-INF/myserver-app/myclient-app-endpoint.xml 中新增配置：

```
<!--引用 SOFARPC 服务-->
<!--以下接口名请根据服务提供方的接口全名来指定-->
<sofa:reference id="sampleServiceRef"interface="com.alipay.samples.rpc.SampleService">
<sofa:binding.bolt>
<sofa:global-attrs address-wait-time="5000"/>
<!--使用直连的方式访问本地启动的 myserver-app, 发布到线上时需要删除此配置项-->
<sofa:route target-url="127.0.0.1:12201"/>
</sofa:binding.bolt>
</sofa:reference>
```

说明：target-url="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 中配置 rpc.tr.port=12201 进行对应。

引用对象注入

为了方便直观使用，我们将引用服务，放在 RPC 服务引用类，即 Web 子模块的 com.alipay.mytestsofa.SOFABootWebSpringApplication 中，示例如下：

```
@ImportResource({"classpath*:META-INF/myclient-app/*.xml"})
@org.springframework.boot.autoconfigure.SpringBootApplication
public class SOFABootWebSpringApplication {
private static final Logger logger = LoggerFactory.getLogger(SOFABootWebSpringApplication.class);

public static void main(String[] args) {
```

```
//*************** 注意 ******************//
//1. 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
//2. 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");
//3. 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(SOFABootWebSpringApplication.class);
ApplicationContext applicationContext = springApplication.run(args);

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

//2. 调用 SOFARpc 服务
final SampleService sampleService = (SampleService) applicationContext.getBean("sampleServiceRef");

new Thread(new Runnable() {
@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) {
//ignore
}
}

}

}
}).start();

}

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

本地测试中，如果客户端和服务端同时在本地启动，需要修改客户端的端口，防止端口冲突，示例如下：

- 在 myserver-app web 模块 src/main/resources/config/application.properties 文件中配置 rpc.tr.port=12201。

- 在 myclient-app web 模块 src/main/resources/config/application.properties 文件中配置 rpc.tr.port=12202。

说明：rpc.tr.port 是 TR 端口号，默认为 12200。

自定义工程

如您想要参考上述 示例工程 自定义创建 SOFABoot 工程，发布引用 SOFARPC 服务，您可以通过以下步骤完成。

1. 创建并配置好 2 个 SOFABoot Web 工程，分别作为客户端和服务端。创建 SOFABoot 工程的详细步骤，可参见 SOFABoot 快速入门 。

   说明：请使用 SOFABoot 3.x.x 版本。最新版本信息参见 SOFABoot 版本说明 。

2. 在全局配置文件 application.properties 中，添加以下配置信息：
   - com.alipay.instanceid：应用实例在工作空间中的唯一标识。您可以前往 **脚手架 > Step 2 > 实例标识** 获取。
   - com.antcloud.antvip.endpoint：应用通过 AntVIP 指来获取各个组件的服务端地址。每个区域一个地址。您可以前往 **脚手架 > Step 2 > AntVIP** 获取。

   - com.antcloud.mw.access 、 com.antcloud.mw.secret：访问中间件的身份验证密钥，可前往 RAM 控制台 获取。详见 创建 AccessKey。

3. 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>
```

4. 在上述创建的工程 endpoint 模块相同位置添加服务接口及实现，参考上文中的 示例工程配置 。
5. 在 2 个工程 endpoint 模块的 Spring XML 文件中，分别添加服务发布和引用配置信息。
6. 启动服务提供方 web 模块的 SOFABootWebSpringApplication，发布服务。
7. 启动服务引用方 web 模块的 SOFABootWebSpringApplication，引用服务。

部署应用

本地应用开发测试完成后，您需要将应用部署至金融分布式架构 SOFAStack 平台。发布部署的详细步骤，可参见 经典应用服务的快速入门 。

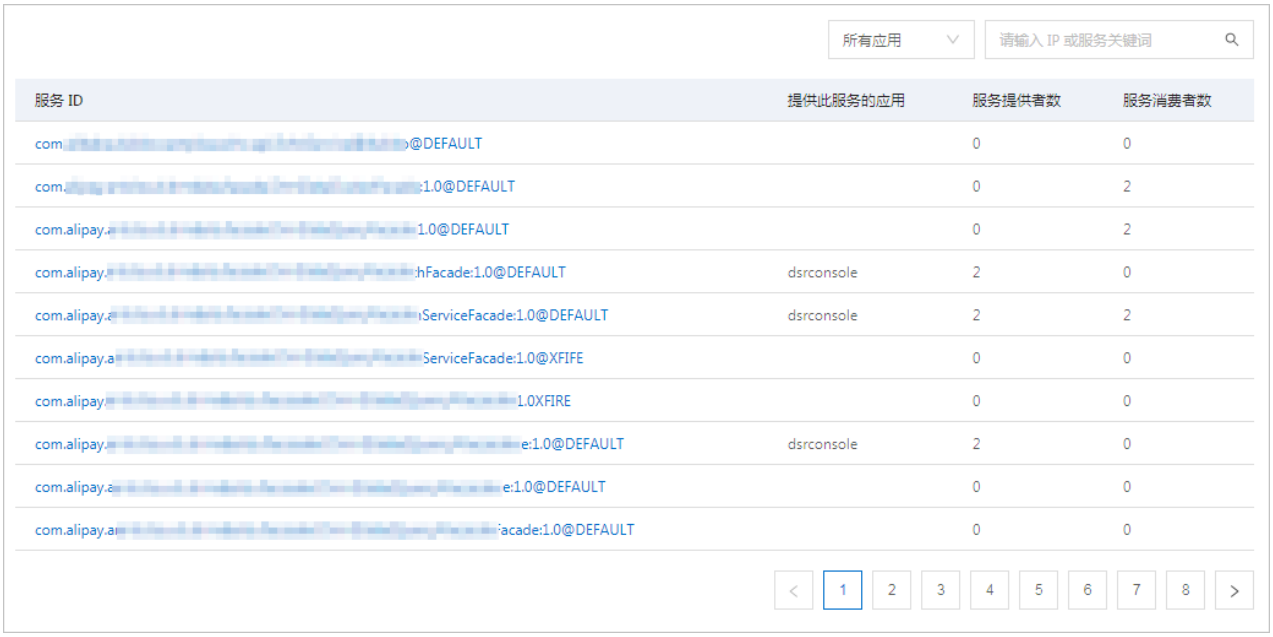
服务查询

应用在云上环境发布成功后，即可在微服务控制台查询到该服务。微服务控制台提供已注册服务的查询功能，可以查询已经注册的服务的提供者与消费者的具体信息。

登录微服务平台后，点击 **微服务** > **服务管控**。

查询实例下所有的服务

通过关键字可以模糊查询已经发布或订阅的服务。例如，通过 ElasticService 关键字可以搜索到服务类名或包名中含 ElasticService 的所有服务。无条件查询可以匹配该实例下所有的服务，也可以输入 IP 进行精确查询，如输入 11.165.199.35，可以查询这个 IP 节点发布或订阅的所有服务。



有关 RPC 服务管控的更多信息，请参见 服务管控 > 查看及管理 RPC 服务 。

5.3 开始使用 SOFAREST

SOFARPC 原生支持 REST 协议，同时支持 DSR（Direct Server Return）和负载均衡。

前置条件

在项目中引入下面的依赖。版本已由 SOFABoot 管控。

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

发布服务

服务接口定义

```
@Path("/webapi")
@Consumes("application/json;charset=UTF-8")
@Produces("application/json;charset=UTF-8")
public interface RestService {

@GET
@Path("/restService/{id}")
String sayRest(@PathParam("id") String string);
}
```

服务实现

```
public class RestServiceImpl implements RestService {
@Override
public String sayRest(String string) {
return"rest";
}
}
```

服务发布配置

在 rpc_server.xml 中添加服务发布的配置：

```
<bean id="restServiceImpl"class="com.alipay.sofa.rpc.samples.rest.RestServiceImpl"/>
<sofa:service ref="restServiceImpl"interface="com.alipay.sofa.rpc.samples.rest.RestService">
<sofa:binding.rest/>
</sofa:service>
```

服务发布的默认端口是 8341。可通过配置 com.alipay.sofa.rpc.rest.port 以修改 SOFAREST 默认端口。详见 RPC 应用参数配置 及 SOFABoot 系统配置参数 。

服务引用

在 rpc_client.xml 中添加对服务引用的配置：

```
<sofa:reference id="restServiceReference"interface="com.alipay.sofa.rpc.samples.rest.RestService">
<sofa:binding.rest/>
</sofa:reference>
```

服务调用

SOFARPC 支持对 RESTful 服务的直接调用。调用通过注册中心进行负载均衡的地址获取，示例代码如下：

```
RestService restService = (RestService) applicationContext.getBean("restServiceReference");
String result = restService.sayRest("rest");
```

您也可以直接通过 HttpClient 或者浏览器发起服务调用请求：

```
# curl http://127.0.0.1:8341/webapi/restService/1
rest%
```

5.4 服务发布与引用

5.4.1 发布 SOFARPC 服务

RPC 是日常开发中最常用的中间件，通过本教程，您将学习到如何利用一个 SOFABoot Core 工程发布一个 RPC 服务。

点击此链接 下载示例工程。SOFARPC 项目示例代码位于 middleware-v2/SOFA_Lite2_Share_RPC 文件夹下。下载完成后，需导入 IDE 工具，具体方法请参见 SOFABoot 快速入门 > 导入 IDE 工具 。

配置额外参数

您需要在 application.properties 中配置以下参数：

1. com.alipay.env
2. com.alipay.instanceid
3. com.antcloud.antvip.endpoint
4. com.antcloud.mw.access
5. com.antcloud.mw.secret

参数的具体含义见 引入 SOFA 中间件 > 添加中间件全局配置项 。

定义服务接口并提供实现

要发布一个 RPC 服务，我们首先要定义一个接口，示例如下：

```
// com.alipay.APPNAME.facade.SampleService
public interface SampleService {

String message();
}
```

我们对这个接口提供一个默认实现，示例如下：

```
// com.alipay.APPNAME.service.SampleServiceImpl
public class SampleServiceImpl implements SampleService {

@Override
public String message() {
return"Hello, Service SOFABoot";
}
}
```

同时，将提供的这个实现配置为一个 Java bean。

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

发布 RPC 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。

服务提供方定义服务 <sofa:service>，进行服务发布；服务消费方定义服务引用 <sofa:reference>，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。

interface

SOFA 服务以 Java 接口形式定义，最主要的属性就是 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。

ref

ref 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。

下面的代码样例中 ref 字段配置的值引用的就是我们之前配置的 sampleServiceBean。

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

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，那么您可以在发布服务的时候加上一个 unique-id 属性来进行区分。

```
<!-- 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>
```

本地运行

1. 在工程的根目录下执行 mvn clean install 命令，会在 target 目录下生成一个 APPNAME-service-1.0-SNAPSHOT-executable.jar 文件，这是一个可执行的 fat jar 文件。
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。
    - 在服务器上执行 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

详情参见 在云端运行 SOFABoot 应用 。

日志查看

查看 sofaboot-rpc-server 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：

```
2016-12-17 15:16:44,466 INFO main - sofa rpc run.mode = DEV
2016-12-17 15:16:49,479 INFO main - PID:42843 sofa rpc starting!
```

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

```
2016-12-17 15:17:07,764 INFO main RPC-REGISTRY - 发布 RPC 服务：服务名
[com.alipay.APPNAME.facade.SampleService:1.0@DEFAULT]
```

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，说明我们成功发布了一个 RPC 服务。

关于日志的详细信息，请参见 日志说明 。

5.4.2 引用 SOFARPC 服务

本文将引导您快速学习如何引用一个 RPC 服务。

点击此处 下载示例工程。项目示例代码位于 middleware-v2/SOFA_Lite2_Share_RPC 文件夹下。下载完成后，需导入 IDE 工具，具体方法请参见 SOFABoot 快速开始 > 导入 IDE 工具 。

配置额外参数

您需要在 application.properties 中配置以下参数：

1. com.alipay.env
2. com.alipay.instanceid
3. com.antcloud.antvip.endpoint

参数的具体含义参见 引入 SOFA 中间件 > 中间件全局配置项 。

引入接口定义依赖

要引用一个 RPC 服务，我们需要知道 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 ，我们还需要知道 unique-id ），这就要求服务提供方将自己所发布的接口所在的 JAR 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 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，两个应用名称不可完全一样。

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

注意：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，然后触发调用引用的服务，再调用到服务端，实际开发中，并不需要注入 Controller 这一步，请使用方注意。

本教程中，我们将这个 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，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 。
- 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.TomcatEmbeddedServletContainer -
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, Service SOFABoot
```

云端运行

详情参见 在云端运行 SOFABoot 应用 。

日志查看

查看 sofaboot-rpc-client 工程引用 RPC 服务启动日志：查看日志目录 logs/rpc/rpc-registry.log，内容参考如下：

```
2016-12-17 15:45:50,340 INFO main RPC-REGISTRY - 订阅 RPC 服务：服务名
[com.alipay.APPNAME.facade.SampleService:1.0@DEFAULT]
```

以上日志说明 RPC 服务引用成功，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 。

5.4.3 基本配置

5.4.3.1 使用 XML 配置

本文介绍如何在 XML 方式中发布和引用服务。XML 配置各元素意义如下：

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

使用示例如下：

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

一个服务也可以通过多种协议进行发布，如下：

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

服务引用示例：

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

服务引用也可以通过其他的协议，例如：

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

5.4.3.2 使用注解方式

除了常规的 XML 方式发布服务外，也支持在SOFABoot 环境下使用注解方式的发布与引用，同 XML 类似，我们有@SofaService 和 @SofaReference，同时对于多协议，存在@SofaServiceBinding 和 @SofaReferenceBinding 注解。

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(bindingType ="bolt") })
@Component
public class AnnotationServiceImpl implements AnnotationService {
@Override
public String sayAnnotation(String stirng) {
return stirng;
}
}
```

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 str) {

String result = annotationService.sayAnnotation(str);

return result;
}
}
```

5.4.3.3 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。SOFA 提供两种方式获取 ClientFactory：

- 实现 ClientFactoryAware 接口
- 使用 @SofaClientFactory 注解

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>
```

获取 ClientFactory

实现 ClientFactoryAware 接口

第一种获取 ClientFactory 的方式是实现 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"/>
```

这样，ClientFactoryBean 就可以获取到 clientFactory 对象来使用了。

使用 @SofaClientFactory 注解

第二种获取 ClientFactory 的方式是使用 @SofaClientFactory 注解，代码示例如下：

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

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

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

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

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

不过，这样只是获取到 ClientFactory 这个对象。如果要获取特定的客户端，如 ServiceFactory，还需要调用 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。只要是在 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);
```

重要：

- 通过动态客户端创建 SOFA Reference 返回的对象是一个非常重的对象，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。

5.4.4 特性使用

本文主要介绍服务发布和引用的一些特性。

同一服务发布多种协议

在 SOFARPC 中，可以将同一个服务发布成多个协议，让调用端可以使用不同的协议调用服务提供方。

如果使用 Java API，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);
```

如果使用 XML 的方式，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>
```

如果使用 Annotation 的方式，在 @SofaService 中增加多个 binding 即可：

```
@SofaService(
interfaceType = SampleService.class,
bindings = {
@SofaServiceBinding(bindingType ="rest"),
@SofaServiceBinding(bindingType ="bolt")
}
)
public class SampleServiceImpl implements SampleService {
// ...
}
```

同一服务注册多个注册中心

如果使用 API 的方式，构建多个 RegistryConfig 设置给 ProviderConfig 即可：

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

方法级参数设置

在 Java API 方式中，调用 MethodConfig 对象相应的 set 方法即可设置对应的参数，如下所示：

```
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
```

使用 XML 的方式，在对应的 binding 里面使用 <sofa:method> 标签即可设置对应的参数：

```
<sofa:reference
id="personReferenceBolt"interface="com.alipay.sofa.boot.examples.demo.rpc.bean.PersonService">
<sofa:binding.bolt>
<sofa:global-attrs timeout="3000"address-wait-time="2000"/> <!-- 调用超时；地址等待时间。 -->
<sofa:route target-url="127.0.0.1:22000"/> <!-- 直连地址 -->
<sofa:method name="sayName"timeout="3000"/> <!-- 方法级别配置 -->
</sofa:binding.bolt>
</sofa:reference>

<sofa:service ref="sampleFacadeImpl"interface="com.alipay.sofa.rpc.bean.SampleFacade">
<sofa:binding.bolt>
<sofa:global-attrs timeout="3000"/>
<sofa:method name="sayName"timeout="2000"/>
</sofa:binding.bolt>
</sofa:service>
```

目前 Annotation 的方式暂不支持设置方法级别的参数，将在后续版本中支持。

配置覆盖

SOFARPC 里面的某些配置在服务提供方可以设置，在服务调用方也可以设置，比如调用的超时的 timeout 属性，这些配置的优先级如下：

```
线程调用级别设置 >> 服务调用方方法级别设置 >> 服务调用方 Reference 级别设置 >> 服务提供方方法级别设置 >> 服务提供方 Service 级别设置
```

5.4.5 配置详参

5.4.5.1 配置说明

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
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
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/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斟酌。查看机器地址的命令：Windows 系统为 ipconfig；Mac/Linux 系统为 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。

性能日志打印阈值配置

详见 日志说明 。

5.4.5.2 RPC 发布订阅说明

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	名称	默认值	说明
id	ID	bean 名	
class	类	无	
ref	服务接口实现类		
interface	服务接口（唯一标识元素）		不管是普通调用和返回调用，这里都设置实际的接口类
unique-id	服务标签（唯一标识元素）		
filter	过滤器配置别名		多个用逗号隔开
registry	服务端注册中心		逗号分隔
timeout	服务端执行超时时间		
serialize-type	序列化协议	hessian2,pro	

		tobuf	
thread-pool-ref 服务端当前接口使用的线程池	无		
weight	服务静态权重		
warm-up-weight	服务预热权重		
warm-up-time	服务预热时间		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-wait-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	名称	默认值	备注
id	ID	自动生成	
jvm-first	是否优先本地	true	
interface	服务接口（唯一标识元素）		不管是普通调用和返回调用，这里都设置实际的接口类
unique-id	服务标签（唯一标识元素）		
type	调用方式	sync	callback,sync,future,oneway
filter	过滤器配置别名		List
registry	服务端注册中心		List
method	方法级配置		说明同上
serialize-type	序列化协议	hessian2	
target-url	直连地址		直连后register
generic-interface	泛化接口		
connect.timeout	建立连接超时时间	3000(cover 5000)	
connect.num	连接数	1	
idle.timeout	空闲超时时间		
idle.timeout.read	读空闲超时时间		

loadBalancer	负载均衡算法	random	
lazy	是否延迟建立长连接	false	
address-wait-time	等待地址获取时间	-1	取决于实现，可能不生效。
timeout	调用超时时间	3000(cover 5000)	
retries	失败后重试次数	0	跟集群模式有关，failover读取此参数。
callback-class	callback 回调类	无	callback 才可用
callback-ref	callback 回调类	无	callback 才可用

5.4.5.3 RPC 应用参数配置

在 SOFABoot 的使用场景下，RPC 框架在应用层面，提供一些配置参数，支持的应用级别的参数配置，如端口、线程池等信息，都是通过Spring Boot的@ConfigurationProperties 进行的绑定。绑定属性类是 com.alipay.sofa.rpc.boot.config.SofaBootRpcProperties，配置前缀如下。

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

那么在 application.properties 文件中，目前可以配置以下几个选项。其中使用者也可以根据自己的编码习惯，按照 Spring Boot 的规范，按照驼峰、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;
```

5.4.5.4 RPC 发布订阅配置

ProviderConfig

属性	名称	默认值	说明
id	ID	自动生成	
application	应用对象	空 ApplicationConfig	
interfaceId	服务接口（唯一标识元素）		不管是普通调用和返回调用，这里都设置实际的接口类
uniqueId	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		多个用逗号隔开
registry	服务端注册中心		List
methods	方法级配置		Map<String, MethodConfig>
serialization	序列化协议	hessian2	
register	是否注册	true	取决于实现，可能不生效。

subscribe	是否订阅	true	取决于实现，可能不生效。
proxy	代理类型	javassist	还有 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	服务延迟发布时间		服务延迟
weight	服务静态权重		
include	包含的方法		
exclude	不包含的方法		
dynamic	是否动态注册		
priority	服务优先级		
bootstrap	服务发布启动器	bolt	
executor	自定义线程池		
timeout	服务端执行超时时间		
concurrents	并发执行请求		接口下每方法的最大可并行执行请求数，配置 -1 关闭并发过滤器，等于 0 表示开启过滤但是不限制
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（jsr303）		
compress	是否启动压缩	false	
cache	是否启用结果缓存	false	
parameters	额外属性		Map<String, String>

ConsumerConfig

属性	名称	默认值	说明
id	ID	自动生成	
application	应用对象	空 ApplicationConfig	
interfaceId	服务接口（唯一标识元素）		不管是普通调用和返回调用，这里都设置实际的接口类
uniqueId	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
registry	服务端注册中心		List
methods	方法级配置		Map<String, MethodConfig>
serialization	序列化协议	hessian2	
register	是否注册	true	取决于实现，可能不生效。
subscribe	是否订阅	true	取决于实现，可能不生效。
proxy	代理类型	javassist	还有 JDK 动态代理
protocol	调用的协议	bolt	目前支持 bolt，rest，dubbo
directUrl	直连地址		直连后 register
generic	是否泛化调用	false	
connectTimeout	建立连接超时时间	3000(cover 5000)	
disconnectTimeout	断开连接等等超时时间	5000(cover 10000)	
cluster	集群模式	failover	
connectionHolder	连接管理器实现	all	
loadBalancer	负载均衡算法	random	
lazy	是否延迟建立长连接	false	
sticky	是否使用粘性连接	false	跳过负载均衡算法使用上一个地址
inJVM	是否转为JVM调用	true	JVM发现服务提供者，转为走本地
check	是否检查强依赖	false	无可用服务端启动失败
heartbeat	心跳间隔	30000	客户端给服务端发心跳间隔。取决于实现，可能不生效。
reconnect	重连间隔	10000	客户端重建端口长连接的间隔。取决于实现，可能不生效。
router	路由器配置别名		List
routerRef	路由器配置实例		List
bootstrap	服务引用启动器	bolt	
addressWait	等待地址获取时间	-1	取决于实现，可能不生效。
timeout	调用超时时间	3000(cover 5000)	
retries	失败后重试次数	0	跟集群模式有关，failover读取此参数。
invokeType	调用类型	sync	
onReturn	并发执行请求数		接口下每方法的最大可并行执行请求数， 配置-1关闭并发过滤器，等于0表示开启过滤但是不限制
cacheRef	结果缓存实现类		
mockRef	Mock 实现类		
cache	是否启用结果缓存	false	
mock	是否开启 Mock		
validation	是否开启参数验证		基于 JSR303

compress	是否启动压缩	false	
parameters	额外属性		Map<String, String>

MethodConfig

属性	名称	默认值	说明
name	方法名		
timeout	调用超时时间	null	
retries	失败后重试次数	null	
invokeType	调用类型	null	
validation	是否开启参数验证	null	基于JSR303
onReturn	返回时调用的SofaResponseCallback	null	用于实现Callback等
concurrent	并发执行请求数	null	接口下每方法的最大可并行执行请求数，配置 -1 关闭并发过滤器，等于 0 表示开启过滤但是不限制。
validation	是否开启参数验证	null	
compress	是否启动压缩	null	
parameters	额外属性		Map<String, String>

ServerConfig

id	Id	默认值	说明
protocol	协议	bolt	目前支持 bolt，rest，dubbo
host	主机	0.0.0.0	
port	端口	12200	默认端口 bolt:12200, rest:8341, h2c:12300, dubbo:20880
contextPath	上下文路径	/	
ioThreads	IO 线程池数	0	取决于实现，可能不生效。例如 bolt 默认 cpu*2。0 表示自动计算。
threadPoolType	业务线程池类型	cached	
coreThreads	业务线程池核心大小	80(override 20)	
maxThreads	业务线程池最大值	400(override 200)	
telnet	是否允许telnet	true	取决于实现，可能不生效。例如bolt不支持telnet。
queueType	业务线程池类型	normal	另外还有优先级队列等
queues	业务线程池队列	1000(override 0)	
aliveTime	业务线程池存活时间	300000(override 60000)	

preStartCore	Id	自动生成	
accepts	最大长连接数量	100000	取决于实现，可能不生效。
serialization	序列化协议	hesisan2	
virtualHost	虚拟主机地址		注册到注册中心时优先使用
virtualPort	虚拟主机端口		注册到注册中心时优先使用
epoll	使用	false	取决于实现，可能不生效。
daemon	是否守护端口	true	true的话随主线程退出而退出，false 的话则要主动退出
adaptivePort	是否调整端口	false	当端口被占用，自动 +1 进行适应
transport	传输层实现	bolt (cover netty4)	取决于实现，可能不生效
autoStart	是否自动启动端口	true	
stopTimeout	优雅关闭超时时间（毫秒）	10000(override 20000)	
boundHost	绑定的地址		默认取host值。
parameters	额外属性		Map<String, String>

RegistryConfig

属性	名称	默认值	说明
protocol	协议	zookeeper	目前支持 zookeeper 和 local
address	注册中心地址		和 index 属性必选一个
index	指定注册中心寻址服务的地址		和 address 属性必选一个
register	是否注册服务	true	
subscribe	是否订阅服务	true	
timeout	调用注册中心超时时间	10s	
connectTimeout	连接注册中心超时时间	20s	
file	local 协议时使用的本地文件位置	$HOME	

5.4.5.5 自动故障剔除

自动故障剔除会自动监控 RPC 调用的情况，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。

在 SOFABoot 中，只需要配置自动故障剔除的参数到 application.properties 即可。可以不完全配置，只配置自己关心的参数，其余参数会取默认值。需要注意的是 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。`

属性	描述	默认值
timeWindow	时间窗口大小：对统计信息计算的周期。	10s

leastWindowCount	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。	10次
leastWindowExceptionRateMultiple	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。	6倍
weightDegradeRate	降级比率：地址在进行权重降级时的降级比率。	1/20
weightRecoverRate	恢复比率：地址在进行权重恢复时的恢复比率。	2倍
degradeEffective	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。	false(关闭)
degradeLeastWeight	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。	1
degradeMaxIpCount	降级的最大ip数：同一个服务被降级的ip数不能超过该值。	2
regulationEffective	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摘除功能，否则完全不进入该功能的逻辑。	false(关闭)

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=6
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=ture
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
```

如上配置，打开了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 6 倍则对该节点进行权重降级，降级的比率为 0.5。权重最小降级到 1。如果单个节点的异常率低于了平均异常率的 6 倍则对该节点进行权重恢复，恢复的比率为 1.2。单个服务最多降级 2 个 IP。

5.4.6 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，并生成以下几个日志文件，包含：

- 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

有关 Tracer 日志，具体参见 日志格式 > SOFARPC 日志 。

5.5 通信协议

5.5.1 BOLT 协议

5.5.1.1 配置 BOLT 服务

BOLT 服务的名称来自于 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，详见 配置说明 。

发布及引用 BOLT 服务

SOFA 中的 RPC 是通过 Binding 模型来定义不同的通信协议的，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/> 。

- 要发布一个 BOLT 的服务，请在 <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="service1">
<sofa:binding.bolt/>
</sofa:service>
```

- 要引用一个 BOLT 的服务，请在 <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。BOLT 支持在服务提供方配置超时时间，XML 配置如下：

```
<!-- 服务方超时设置 -->
<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>
```

> **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。

消费方的超时时间配置和提供方类似，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>
```

如果没有配置 type，则默认 type 为 sync。如果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。

```
<!-- 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>
```

oneway 调用

如果是 oneway 调用方式，消费方不关心结果，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。

```
<!-- 超时设置无效 -->
<sofa:reference interface="com.alipay.test.SampleService2"id="sampleService">
<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"timeout="1000"/>
</sofa:binding.bolt>
</sofa:reference>
```

callback 调用

callback 是一种异步回调的方式，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。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.test.binding.tr.MyCallBackHandler"/>
</sofa:binding.bolt>
</sofa:reference>
```

上面的 callback 接口是通过配置 class 的方式来实现的，SOFA 也提供了通过引用一个 Bean 的方式来实现

callback。

```
<!-- 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="myCallBackHandlerBean"/>
</sofa:binding.bolt>
</sofa:reference>
```

需要注意，callback 调用方式的 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，配置和前面类似，只需要在 method 层面设置 type 为 future 即可。

```
<!-- 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	说明	默认值	取值范围
connect.timeout	INTEGER	消费方连接超时时间	1000 (ms)	正整数，以毫秒为单位（ms）。
connect.num	INTEGER	消费方连接数	-1	-1 代表不设置（默认为每个目标地址建立一个连接）。
idle.timeout	INTEGER	消费方最大空闲时间	-1	-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	消费方最大读空闲时间	-1	-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。
address-wait-time	INTEGER	reference 生成时，等待服务注册中心将地址推送到消费方的时间。	0 (ms)	最大值为 30000 ms，超过这个值的配置将调整为 30000 ms。

BOLT 服务完整配置

BOLT 配置标签主要有两种：global-attrs 和 method。如果 global-attrs 和 method 同时进行了配置，那么以 method 中的配置为准。可配置的参数如下：

- type：调用方式，有 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="service1">
<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="service1">
<sofa:binding.bolt>
<sofa:global-attrs timeout="5000"test-url="localhost:12200"address-wait-time="1000"
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>
```

5.5.1.2 BOLT 协议的基本使用

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 协议只需要在 <sofa:service> 标签下增加 <sofa:binding.bolt> 标签即可。

```
<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 协议的服务只需要设置 @SofaServiceBinding 的 bindingType 为 bolt 即可。

```
@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 协议的服务只需要往 ServiceParam 里面增加一个 BoltBindingParam 即可。

```
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())
.setServer(serverConfig) // 将 ServerConfig 设置给 ProviderConfig，表示这个服务发布的协议为 Bolt。
.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。不同的使用方式添加 Bolt Binding 的方式如下。

XML

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

```
<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 协议的服务只需要设置 @SofaReferenceBinding 的 bindingType 为 bolt 即可。

```
@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，只需要往 ReferenceParam 里面增加一个 BoltBindingParam 即可。

```
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，只需要设置 ConsumerConfig 的 Protocol 为 bolt 即可。

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

5.5.1.3 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

在同步的调用方式下，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，无需进行任何设置即可。

异步

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，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。如果需要将一个服务设置为异步的调用方式，在对应的使用方式下设置 type 属性即可。

XML 方式

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

```
<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 即可。

```
@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 属性即可。

```
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 裸 API，设置 ConsumerConfig 的 invokeType 属性即可。

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

获取调用结果

使用异步调用的方式，目前提供了两种方式来获取异步调用的结果。

直接获取结果

用户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);
```

其中第一个参数是获取结果的超时时间，第二个参数表示是否清除线程上下文中的结果。

获取 JDK 原生 Future

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

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

其中的第一个参数表示是否清除线程上下文中的结果。

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，用户使用 SOFARPC Bolt 协议的回调方式，首先需要实现这个接口，该接口提供了三个方法：

- 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，比如路由错误的时候，SOFARPC 会回调这个方法。

设置回调接口

实现回调接口之后，用户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两种方式，分别为 Callback Class 和 Callback Ref。Callback Class 的方式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 的方式则为用户直接提供回调类的实例。

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 属性即可。

```
<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 属性即可。

```
@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。

```
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，调用 setInvokeType 将类型设置成 callback，并且调用 setOnReturn 设置回调类即可。

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

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，还可以在调用级别设置回调接口，方式如下：

```
RpcInvokeContext.getContext().setResponseCallback(new SampleCallback());
```

单向

当客户端发送请求后不关心服务端返回的结果时，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要将调用方式设置为 oneway 即可，设置方式和将调用方式设置为 future 或者 callback 一样，这里不再重复讲述，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，所以所有的超时设置在单向的情况下都是无效的。

5.5.1.4 超时控制

使用 BOLT 协议进行通信的时候，SOFARPC 的超时时间默认为 3 秒，用户可以在引用服务的时候去设置超时时间，又分别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

如果需要在发布服务的时候在服务维度设置超时时间，设置对应的 timeout 参数到对应的值即可。

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

如果想要单独调整一个服务中某一个方法的超时时间，可以通过在方法维度上设置超时时间来实现。

对于某一个方法来说，优先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);
```

5.5.1.5 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) 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 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, GenericContext
context) throws GenericException;

}
```

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

Spring XML 配置：

```
<!-- 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 Object[]{"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.generic.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.sofa.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.alipay.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.alipay.sofa.rpc.test.generic.bean.People"}, new Object[] { genericObject }, People.class, genericContext);
}
```

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"
```

5.5.1.6 序列化协议

SOFARPC 可以在使用 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**。

默认的情况下，SOFARPC 使用 hessian2 作为序列化协议，如果需要将序列化协议设置成 protobuf，在发布服务的时候，需要做如下的设置。

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

相应的，在引用服务的时候，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>
```

目前，使用注解的方式尚不能支持设置序列化协议，这个将在后续的版本中支持。

5.5.1.7 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独立的业务线程池，和 SOFARPC 自身的业务线程池是隔离的。多个服务可以共用一个独立的线程池。

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，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="init">
<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，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 ="customThreadPool")})
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，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，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(providerConfig),
threadPool);
```

如上为 HelloService 服务设置了一个自定义线程池。

5.5.2 RESTful 协议

5.5.2.1 RESTful 协议的基本使用

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

在定义 RESTful 的服务接口的时候，需要采用 JAXRS 标准的注解在接口上加上元信息，比如下面的接口：

```
@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。

在定义好了接口之后，将接口的实现发布成一个服务。例如，通过 Annotation 的方式发布任务：

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

如果要通过其他的方式发布服务，请参考 Bolt 协议基本使用 。

通过浏览器访问服务

在发布服务之后，用户可以通过浏览器来直接访问服务，对于上面的服务，访问的地址如下：

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

SOFARPC 的 RESTful 服务的默认端口为 8341。

引用服务

除了通过浏览器访问 SOFARPC 发布的 RESTful 服务之外，用户也可以通过 SOFARPC 标准的服务引用的方式来引用服务，比如通过 Annotation 的方式：

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

如果要使用其他的方式引用服务，请参考 Bolt 协议基本使用 。

5.5.2.2 REST 跨域

对于 REST，我们内置了一个跨域 Filter 的支持。

SOFARPC API 使用

对于使用 SOFARPC API 的用户，可以在 ServerConfig 中添加一个参数表明。

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

XML 方式使用

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
```

5.5.2.3 REST 自定义 Filter

对于 REST，我们设计了一个 JAXRSProviderManager 管理器类。在服务端生效，生效时间为服务启动时。

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

对于用户自定义的 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，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
```

5.5.2.4 集成 SOFARPC RESTful 服务和 Swagger

从 rpc-sofa-boot-starter 6.0.1 版本开始，SOFARPC 提供了 RESTful 服务和 Swagger 的一键集成的能力。

在使用了 rpc-sofa-boot-starter 的情况下，如果想要开启 swagger 的能力，首先需要在 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>
```

然后在 application.properties 里面增加 com.alipay.sofa.rpc.restSwagger=true。

最后，访问 http://localhost:8341/swagger/openapi 就可以拿到 SOFARPC 的 RESTful 的 Swagger OpenAPI 内容。

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

首先，需要在应用中引入 Swagger 相关的依赖，由于 SOFARPC 的 RESTful 协议s使用的是 JAXRS 标准，因

此只需引入 Swagger 的 JAXRS 依赖即可。

```
<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>
```

上面引入 Guava 的 20.0 的版本是为了解决 Guava 的版本冲突。

增加一个 Swagger 的 RESTful 服务

为了能够让 Swagger 将 SOFARPC 的 RESTful 的服务通过 Swagger OpenAPI 暴露出去，可以反过来用 SOFARPC 的 RESTful 的服务提供 Swagger 的 OpenAPI 服务。首先，需要新建一个接口：

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

然后提供一个实现类，并且发布成 SOFARPC 的 RESTful 的服务：

```
@Service
@SofaService(bindings = {@SofaServiceBinding(bindingType ="rest")}, interfaceType = OpenApiService.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(Collectors.toSet()));

OpenApiContext oac = new JaxrsOpenApiContextBuilder()
```

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

这样，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

如果用户在另外一个端口中启动了一个 Swagger UI，并且希望通过 Swagger UI 来访问 http://localhost:8341/swagger/openapi 查看 API 定义，发起调用，那么可能需要解决访问跨域的问题，要解决 SOFARPC RESTful 服务访问跨域的问题，可以在应用启动前增加如下的代码：

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

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

这样，Swagger UI 就可以跨域访问应用提供的 Swagger OpenAPI 了。

5.5.3 Dubbo 协议

5.5.3.1 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。本文使用以注解的方式为例，其他的使用方式可以参考 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 即可。

```
@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 即可。

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

设置 Dubbo 服务的 Group

在 SOFARPC 的模型中，没有直接的一个字段叫做 Group，但是 SOFARPC 有一个 uniqueId 的模型，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 {
}
```

如下的代码，就是引用了一个 Group 为 groupDemo 的服务。

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

注意：目前 Dubbo 协议只支持 Zookeeper 作为服务注册中心。

5.5.4 H2C 协议

5.5.4.1 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 即可。本文使用以注解的方式来例，其他的使用方式可以参考 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 即可。

```
@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 即可。

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

5.5.5 HTTP 协议

5.5.5.1 HTTP 协议的基本使用

在 SOFARPC（非 SOFABoot 环境）中，当使用 HTTP 作为服务端协议的时候，支持JSON 作为序列化方式，作为一些基础的测试方式使用。

SOFARPC API 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();
```

服务引用

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());
```

5.6 进阶指南

5.6.1 直连调用

SOFARPC 支持指定地址进行调用的场景。用 Java API 的使用方式如下，设置直连地址即可。

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

用 XML 的使用方式如下：

```
<sofa:reference interface="com.alipay.sample.HelloService"id="helloService">
<sofa:binding.bolt>
<sofa:route target-url="127.0.0.1:12200"/>
</sofa:binding.bolt>
</sofa:reference>
```

用 Annotation 的使用方式如下：

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

5.6.2 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，通过该类，可以获得最后一次 Reference 以及当次 Service 的相关信息。

需要注意的是，RPC 上下文是存在 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，参见 RPC 日志格式 了解更多信息。

Reference

- traceId
- rpcId
- 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
- targetUrl：服务提供方的地址
- protocol：调用协议，如 TR
- invokeType：调用类型，如 sync、oneway 等
- routeRecord：路由寻址链路，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 的话，那么 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。详细的路由规则参见 RPC 路由 。
- costTime：调用耗时，单位为 ms。
- resultCode：结果码，00 - 成功；01 - 业务异常；02 - RPC 框架错误；03 - 超时失败；04 - 路由失败。

Service

- traceId
- rpcId
- 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

5.6.3 Node 跨语言调用

如果您需要通过 NodeJs 调用 SOFARPC 的，可以按照本文快速开始.

安装

可参考 文档 安装 SOFARPC Nodejs 实现版本。

执行以下命令即可安装。

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

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;

// 1. 创建 zk 注册中心客户端
const registry = new ZookeeperRegistry({
logger,
address: '127.0.0.1:2181', // 需要本地启动一个 zkServer
});

// 2. 创建 RPC Server 实例
const server = new RpcServer({
logger,
registry, // 传入注册中心客户端
port: 12200,
});

// 3. 添加服务
server.addService({
interfaceName: 'com.nodejs.test.TestService',
}, {
async plus(a, b) {
return a + b;
},
});

// 4. 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;

// 1. 创建 zk 注册中心客户端
const registry = new ZookeeperRegistry({
logger,
address: '127.0.0.1:2181',
});

async function invoke() {
// 2. 创建 RPC Client 实例
const client = new RpcClient({
logger,
registry,
});
// 3. 创建服务的 consumer
const consumer = client.createConsumer({
interfaceName: 'com.nodejs.test.TestService',
});
// 4. 等待 consumer ready（从注册中心订阅服务列表...）
await consumer.ready();

// 5. 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 = new RpcClient({
logger,
});
const consumer = client.createConsumer({
interfaceName: 'com.nodejs.test.TestService',
serverHost: '127.0.0.1:12200', // 直接指定服务地址
});
```

```
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 来定义接口

```
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;
Group group = 2;
}

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

enum Group {
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,
protocol, // 覆盖协议
registry,
codecType: 'protobuf', // 设置默认的序列化方式为 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);
```

5.6.4 负载均衡

SOFARPC 提供多种负载均衡算法，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	同样的方法级别的请求会路由到同样的节点。
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。

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。

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

Annotation 方式

Annotation 方式目前暂未提供设置某一个 Reference 的负载均衡算法的方式。将在后续的版本中提供。

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。

```
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。

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

5.6.5 自定义 Filter

自定义 Filter 类

继承 com.alipay.sofa.rpc.filter.Filter 类。

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

import com.alipay.sofa.rpc.core.exception.SofaRpcException;
import com.alipay.sofa.rpc.core.request.SofaRequest;
import com.alipay.sofa.rpc.core.response.SofaResponse;
import com.alipay.sofa.rpc.filter.Filter;
import com.alipay.sofa.rpc.filter.FilterInvoker;
import com.alipay.sofa.rpc.log.Logger;
import com.alipay.sofa.rpc.log.LoggerFactory;

public class CustomEchoFilter extends Filter {

/**
* Logger for CustomEchoFilter
**/
private static final Logger LOGGER = LoggerFactory.getLogger(CustomEchoFilter.class);

@Override
public boolean needToLoad(FilterInvoker invoker) {
// 判断一些条件，自己决定是否加载这个 Filter
return true;
}

@Override
public SofaResponse invoke(FilterInvoker invoker, SofaRequest request) throws SofaRpcException {
// 调用前打印请求
LOGGER.info("echo request : {}, {}", request.getInterfaceName() +"."+ request.getMethod(),
request.getMethodArgs());

// 继续调用
SofaResponse response = invoker.invoke(request);

// 调用后打印返回值
if (response == null) {
return response;
} else if (response.isError()) {
```

```
LOGGER.info("server rpc error: {}", response.getErrorMsg());
} else {
Object ret = response.getAppResponse();
if (ret instanceof Throwable) {
LOGGER.error("server biz error: {}", (Throwable) ret);
} else {
LOGGER.info("echo response : {}", response.getAppResponse());
}
}

return response;
}
}
```

配置对单个服务发布者（消费者）生效的 Filter

```
<!-- 配置一个自定义 Filter -->
<bean id="customEchoFilter"class="com.alipay.sofa.rpc.customfilter.CustomEchoFilter"/>
<property name="filed1"value="xxxx"/> <!-- 假如有一些自己的字段赋值 -->
</bean>

<bean id="xxServiceImpl"class="com.alipay.xxx.XXServiceImpl"/>

<!-- 配置这个服务发布者的 Filter，支持配置多个，彼此以逗号分开 -->
<sofa:service id="xxServiceExport"ref="xxServiceImpl"interface="com.alipay.xxx.XXService">
<sofa:binding.bolt>
<sofa:global-attrs filter="customEchoFilter"/> <!-- 设置到这里 -->
</sofa:binding.bolt>
</sofa:service>

<!-- 配置这个服务引用者的 Filter，支持配置多个，彼此以逗号分开 -->
<sofa:reference id="xxServiceRef"interface="com.alipay.xxx.XXService">
<sofa:binding.bolt>
<sofa:global-attrs filter="customEchoFilter"/> <!-- 设置到这里 -->
</sofa:binding.bolt>
</sofa:reference>
```

全局 Filter

全局 Filter 的 XML 配置方式和非全局 Filter 配置相似，示例如下：

```
<!-- 方式一：配置一个自定义 Filter -->
<bean id="customEchoFilter"class="com.alipay.sofa.rpc.customfilter.CustomEchoFilter">
<property name="filed1"value="xxxx"/> <!-- 假如有一些自己的字段赋值 -->
</bean>
<!-- 方式二：标记这个 Filter 为全局 Filter -->
<sofa:rpc-global-filter ref="customEchoFilter"/>

<!-- 方式三：标记这个 Filter 为全局 Filter -->
<sofa:rpc-global-filter class="com.alipay.sofa.rpc.customfilter.CustomEchoFilter"/>

<!-- 无需任何配置 -->
<sofa:service ... />
```

```
<!-- 无需任何配置 -->
<sofa:reference ... />
```

5.6.6 自定义路由寻址

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。

```
@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;
}
```

新建扩展文件 META-INF/services/sofa-rpc/com.alipay.sofa.rpc.client.Router，内容如下：

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

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。

5.6.7 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 默认即为 FailOver 模式）。重试只有在发生服务端的框架层面异常或者是超时异常才会发起。如果是业务抛出异常，是不会重试的。默认情况下 SOFARPC 不进行任何重试。

> **注意**：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.sofa.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：

```
@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：

```
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：

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

5.6.8 链路追踪

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

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 ）

- 日志打印时间
- TraceId
- SpanId
- Span 类型
- 当前 appName
- 协议类型
- 服务接口信息
- 方法名
- 当前线程名
- 调用类型

- 路由记录
- 目标ip
- 本机ip
- 返回码
- 请求序列化时间
- 响应反序列化时间
- 响应大小(单位Byte)
- 请求大小(单位Byte)
- 客户端连接耗时
- 调用总耗时
- 本地客户端端口
- 透传的 baggage 数据 (kv 格式)

样例：

{"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":""}

RPC 服务端摘要日志（ rpc-server-digest.log ）

- 日志打印时间
- TraceId
- SpanId
- Span 类型
- 服务接口信息
- 方法名
- 来源ip
- 来源 appName
- 协议
- 本应用 appName
- 当前线程名
- 返回码
- 服务端线程池等待时间
- 业务处理耗时
- 响应序列化时间
- 请求反序列化时间

- 响应大小(单位Byte)
- 请求大小(单位Byte)
- 透传的 baggage 数据 (kv 格式)

样例：


```
{"timestamp":"2018-05-20
17:00:53.312","tracerId":"1e27326d152680685303210018501 1","spanId":"0","span.kind":"server","service":"com.alipa
y.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"S
OFATracerRPC","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":"1e27326d152680698539410018558 9","spanId":"0","span.kind":"server","service":"com.alipa
y.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"S
OFATracerRPC","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":"1e27326d152680700049810018559 7","spanId":"0","span.kind":"server","service":"com.alipa
y.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"S
OFATracerRPC","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":"1e27326d152680704660610018563 5","spanId":"0","span.kind":"server","service":"com.alipa
y.sofa.tracer.examples.sofarpc.direct.DirectService:1.0","method":"sayDirect","remote.ip":"127.0.0.1","remote.app":"S
OFATracerRPC","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":""}
```


RPC 客户端统计日志（ rpc-client-stat.log ）

- 日志打印时间
- 日志关键key
- 方法信息
- 客户端 appName
- 服务接口信息
- 调用次数
- 总耗时
- 调用结果

样例：


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


RPC 服务端统计日志（ rpc-server-stat.log ）

- 日志打印时间
- 日志关键key
- 方法信息
- 客户端 appName
- 服务接口信息
- 调用次数
- 总耗时
- 调用结果

样例：

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

5.6.9 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，达到整个链路上的应用都可以操作该数据。

使用方式如下，可分别向链路的 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。

```
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 A > B > 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);
final RpcInvokeContext 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：

1. 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。
2. 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。

5.6.10 预热权重

集群中一台机器刚启动的一段时间（称之为“预热期”）内，如果请求过多可能会影响机器性能和正常业务。框架提供一种功能，将处于预热期的机器的请求转发到集群内其它机器，过了预热期之后再恢复正常。

也就是说，预热转发功能是指机器在启动完成后的“一段时间”内将其接收的请求转发至集群内的其它机器

，等过了这段时间后再正常接收请求。这段时间内接收的请求既可以全部转发至其它机器，也可以按照一定比例转发，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。

RPC 转发配置

配置的方式有两种：

直接转发到 IP

core_proxy_url = x.x.x.x

不论何时都直接转发到 x.x.x.x，和 core_proxy_url=address:x.x.x.x 有同样效果。

配置预热与权重

core_proxy_url=weightStarting:0.3,during:60,weightStarted:0.2,address:x.x.x.x,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，要区别不同集群可以通过配置此项区分。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，如下所示：
rpc_transmit_url_timeout_tr=8000。
单位为 ms，默认为 10000 ms。

5.6.11 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，使得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，使得客户端请求逐渐将流量分发到该节点。单机故障剔除功能解决了服务故障持续影响业务的问题，避免了雪崩效应。可以减少人工干预需要的较长的响应时间，提高系统可用率。

运行机制如下：

- 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。
- 当达到 IP 异常率大于服务平均异常率到一定比例时，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。

单机故障剔除的使用方式如下：

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

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

如上，该应用会在打开了单机故障剔除开关，每 20s 的时间窗口进行一次异常情况的计算，如果某个服务 + IP 的调用维度被判定为故障节点，则会进行将该服务 + IP 的权重降级为 0.5 倍。

更加详细的参数，请参考 单机故障剔除拆除 。

5.6.12 优雅关闭

优雅关闭包括两部分，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

作为服务端的时候，RPC 框架在关闭时，不应该直接暴力关闭。在 RPC 框架中

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

在静态初始化块中，添加了一个 ShutdownHook。

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

ShutdownHook 的作用是当发布平台/用户执行 kill pid 的时候，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。

```
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_DESTROY))) { // 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

作为客户端，实际上就是 Cluster 的关闭。关闭调用的服务这一步，可以查看下 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 transports 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() {
}
}
```

这里面也会逐步将正在调用的请求处理完成才会下线。

最佳实践

可以看到，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。

5.6.13 单元化配置（仅专有云）

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**：该功能仅适用于专有云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

1. 在本地项目的全局配置文件 application.properties 中，正确配置 antvip、instanceid、AK、SK，详见 引入 SOFA 中间件 > 添加全局配置项 。
2. 在部署的时候，还需要传入以下参数：

```
com.alipay.ldc.zone=LCD-Test-A-1
com.alipay.ldc.datacenter=LCD-Test-A
//表示开启严格模式的LDC
com.alipay.ldc.strictmode=true
zmode=true
//需要确保您的 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。
- 部署在 CZone，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)"}]
```

5.7 SOFARPC 开发

5.7.1 如何编译 SOFARPC 工程

> **前提条件**：安装 JDK7 及以上，Maven 3.2.5 及以上。

直接下载代码，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，如果每个子模块都会 install 和 deploy，仓库内会有较多无用记录。

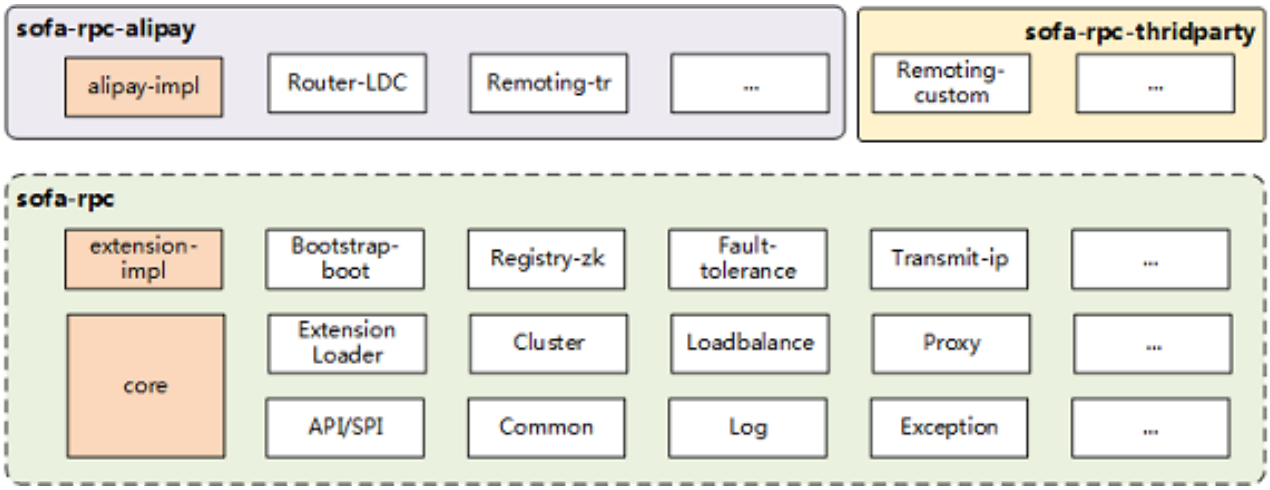
所以在设计 SOFARPC 工程结构的时候，我们决定各个子模块组件是不需要 install 和 deploy 到仓库里的，我们只会 install 和 deploy 一个sofa-rpc-all(all) 模块。

5.7.2 SOFARPC 工程架构介绍

架构图

SOFARPC 从下到上分为两层：

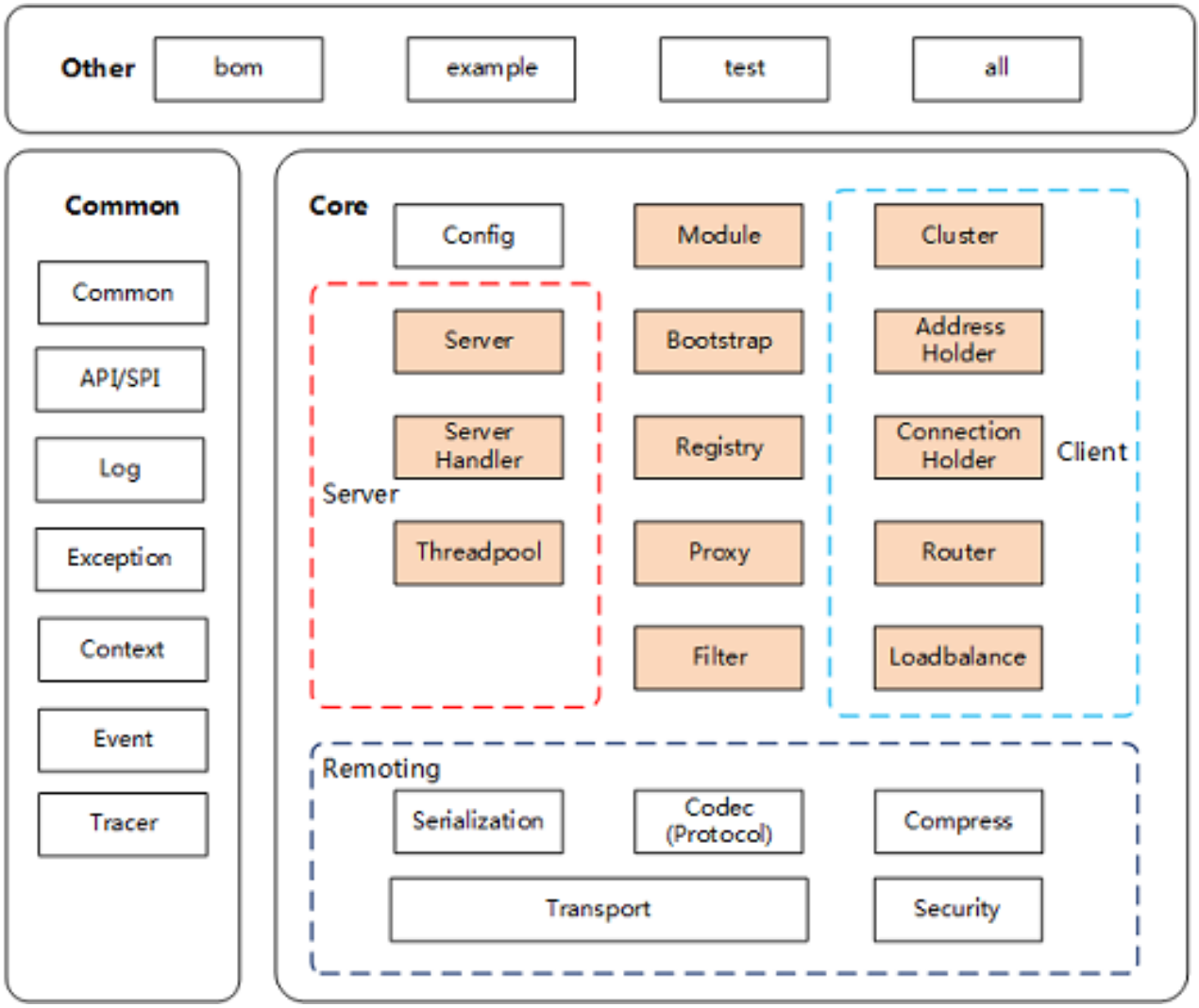
1. 核心层：包含了 RPC 的核心组件（例如各种接口、API、公共包）以及一些通用的实现（例如随机等负载均衡算法）。
2. 功能实现层：所有的功能实现层的用户都是平等的，都是基于扩展机制实现的。



模块划分

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。需要交叉依赖的全部已经抽象到 core 或者 common 模块中。

目前模块划分如下:



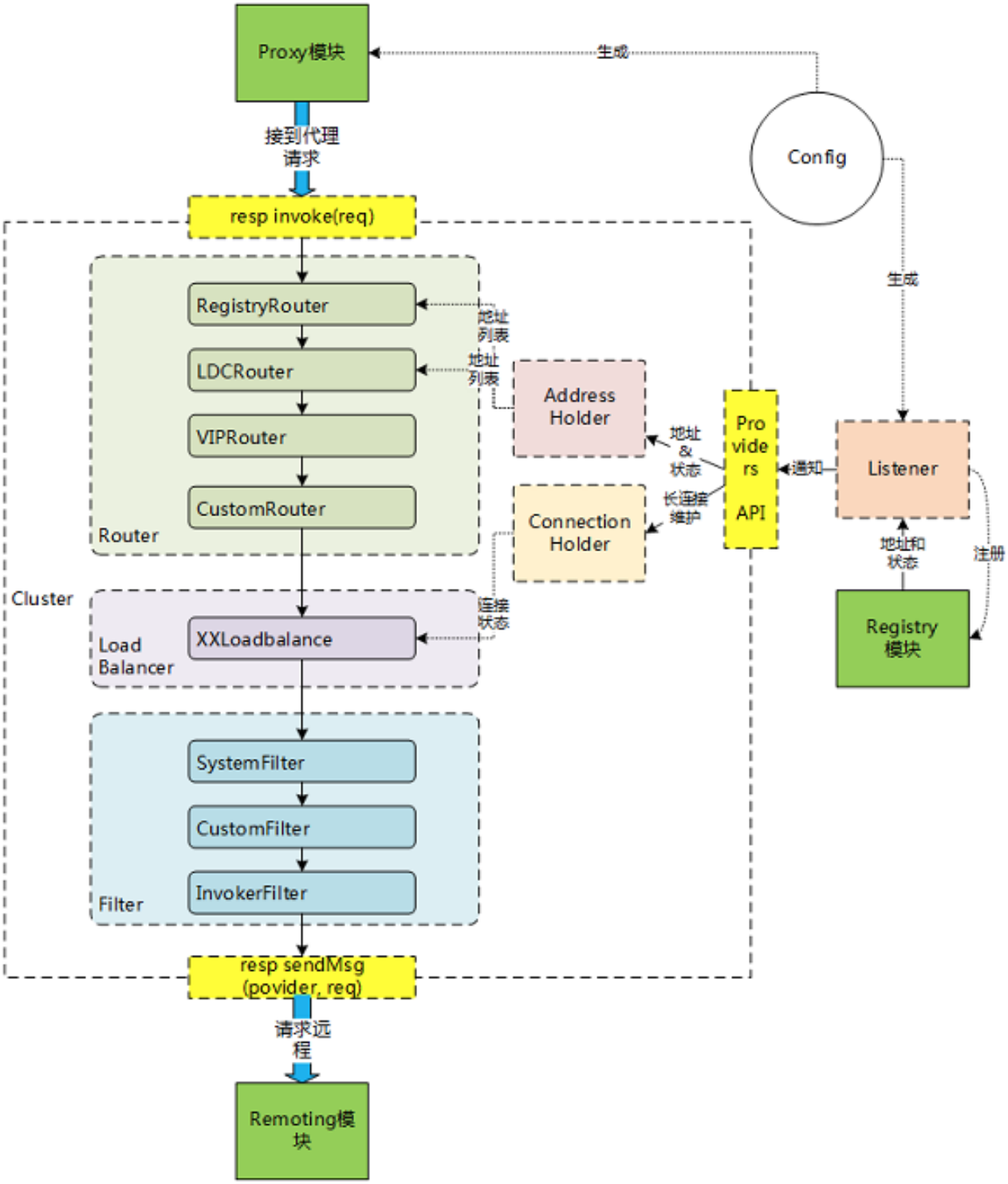
主要模块及其依赖如下：

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
bom		依赖管控模块	依赖版本管控	无
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	发送请求、接收响应、连接维护、路由、负载均衡、同	core

		块	步异步等	
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
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 等	core

5.7.3 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。



5.7.4 基础模型

消息

内部全部使用 SofaRequest 和 SofaResponse 进行传递。

如果需要转换为其它协议，那么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。虽然是扩展，但是由于日志的加载应该是最早的，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取的），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

5.7.5 单元测试与性能测试

单元测试

单元测试例子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。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。

如果还依赖了第三方服务端（例如 Zookeeper），请放到 test-intergrated-3rd 模块中。

性能测试

关闭了以下默认开启项目：-Dcontext.attachment.enable=false -Dserialize.blacklist.enable=false -Ddefault.tracer= -Dlogger.impl=com.alipay.sofa.rpc.log.SLF4JLoggerImpl -Dmultiple.classloader.enable=false -Devent.bus.enable=false

对 BOLT + Hessian 进行了压测。

服务端：4C8G 虚拟机，千 M 网络，jdk1.8.0_111；

客户端：50 个客户端并发请求。

协议	请求	响应	服务端	TPS	平均 RT(ms)
bolt+hessian	1K String	1K String	直接返回	10000	1.93
bolt+hessian	1K String	1K String	直接返回	20000	4.13
bolt+hessian	1K String	1K String	直接返回	30000	7.32
bolt+hessian	1K String	1K String	直接返回	40000	15.78

bolt+hessian	1K String	1K String	直接返回	50000（接近极限,错误率0.3%）	26.51

5.8 常见问题

本文列出 SOFARPC 使用过程中的一些常见问题。

- RPC 单次传输的数据量是否有限制？
- RPC-02306: Cannot get the service address of service
- 每次 RPC 调用都耗时很长，明显超时却不报超时异常

RPC 单次传输的数据量是否有限制？

本身没有限制，考虑性能，建议 4k 以内。如果超过，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

RPC-02306: Cannot get the service address of service

现象

dev 环境 rpc 调用出错，用户调用时，middleware_error.log 报错找不到服务地址

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 服务地址不符。

- 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

示例如下：

```
<sofa:reference id="[illegible]"
          interface="com.[illegible]"
          unique-id="[illegible]">
   <sofa:binding.bolt>
      <sofa:global-attrs timeout="5000" test-url="${[illegible]_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。

每次 RPC 调用都耗时很长，明显超时却不报超时异常

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，且不超时。
- 从业务日志来看，开始处理业务和结束业务之间确实花了 30 秒。

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。

6 服务管控

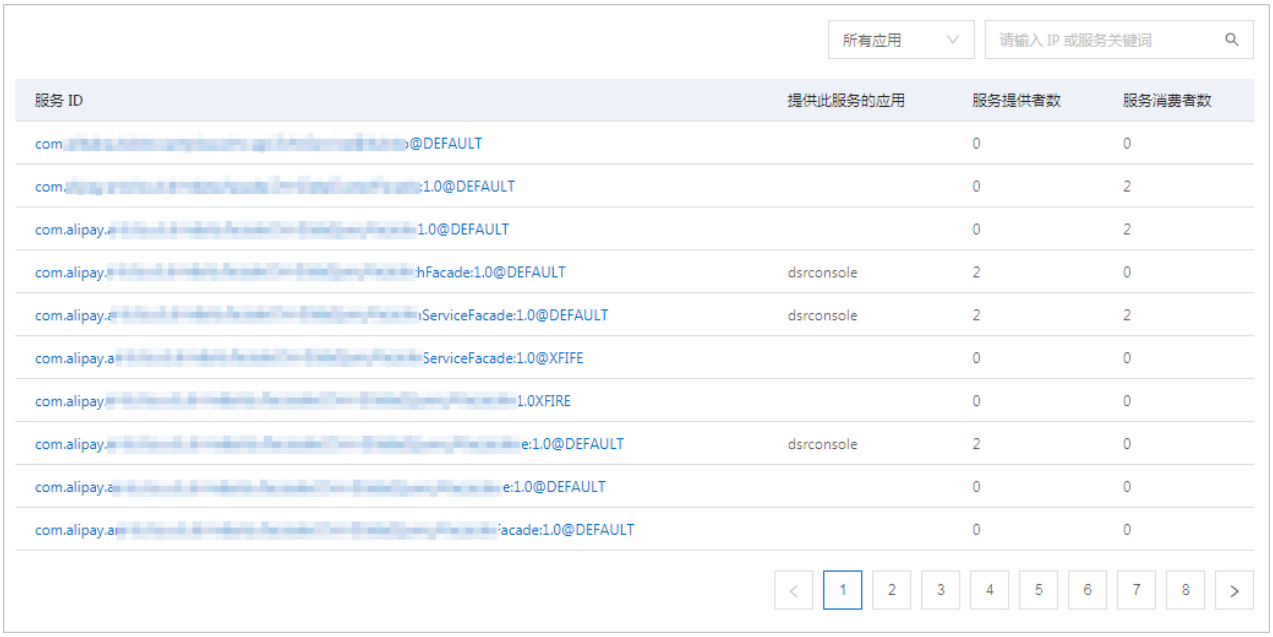
6.1 查看及管理 RPC 服务

您可以在 **服务管控** 页面查看并管理当前环境下的所有可用服务。

查看服务列表

在页面上方的搜索栏中输入您要查找的应用或服务名称，点击搜索。搜索结果提供以下信息：

- 服务 ID
- 提供该服务的应用名
- 服务提供者数量
- 服务消费者数量



查看服务详情

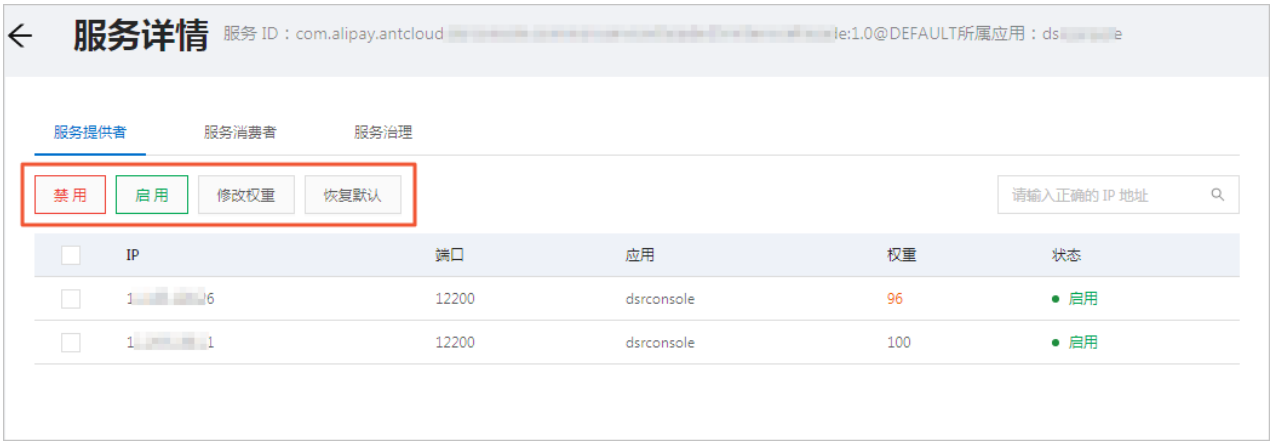
在服务查询列表中点击服务 ID，可以查看该服务的服务提供者和服务消费者列表。您还可以自由切换查看服务提供者列表与服务消费者列表。

- 服务提供者列表中包含服务提供者的 IP、端口、应用名、权重与状态信息。
- 服务消费者列表中包含服务消费者的 IP 与应用名。

管理应用服务

在搜索结果列表中，点击要查看的服务行，进入 **服务详情** 页面。您可以对该服务进行以下操作：

- 启用服务
- 禁用服务
- 修改权重
- 恢复默认值



6.2 查看应用依赖关系

您可以在 **应用依赖** 页面查看当前工作环境中的应用依赖关系图。

将鼠标悬浮在某一应用图标上可以查看以下信息：

- 与当前应用有依赖关系的所有相关应用，包括服务提供者与消费者。
- 该应用的节点数
- 该应用提供的服务数
- 该应用消费的服务数

要查看某一应用的依赖详情，点击该应用的图标或使用搜索栏快速筛选应用名称。

示例

下图展示了名为 sofa2-rpc-client 的应用的依赖详情示例。右侧的应用依赖关系图显示 sofa2-rpc-client 此时没有提供任何服务。而作为服务消费者，sofa2-rpc-client 使用了来自以下 3 个应用的共 5 个服务：

- sofa2-rpc-server
- dtxserver
- dts-sample-action

消费的服务 页签列出了消费的服务 ID 及所属应用。

6.3 访问控制

6.3.1 服务访问控制

通过创建访问控制规则，RPC 服务的提供者可以为特定的调用者添加或限制访问授权，灵活地调整服务订阅者调用本服务的权限。

前置条件

1. SOFABoot 版本必须是 3.1.1 或以上。
2. 在工程的 pom.xml 文件中，引入动态配置依赖：

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

3. 在 RPC 服务端和客户端的 application.properties 文件中，添加以下配置：

```
com.alipay.sofa.rpc.dynamic.alias=drm
```

4. 在引入 RPC jar 包的 pom.xml 文件中，引入 sofa-configuration-sdk：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-configuration-sdk</artifactId>
<version>0.1.1</version>
</dependency>
```

服务访问控制说明

白名单与黑名单

访问控制可通过设置 **白名单**（whitelist）或 **黑名单**（blacklist）完成。白名单与黑名单互斥，二者无法同时开启。每个服务的访问控制只能启用其中一种模式。

- **白名单模式**：只有符合白名单规则的服务调用者有访问权限。白名单外的所有请求都会被拒绝。
- **黑名单模式**：所有符合黑名单规则的服务调用者将被拒绝访问。黑名单外的所有请求都被允许。

规则说明

白名单与黑名单均由一条或多条规则构成。多条规则间用 **或**（OR）的关系连接。一旦名单被启用，访问请求只要满足其中任意一条已启用的规则，即被视为满足过滤条件。

说明：只有处于“已启用”状态的规则才被用来做访问请求的规则匹配。

规则构成

- **规则名**：由汉字、英文字母、数字、下划线组成。
- **状态**：已启用 或 已禁用。

- **匹配条件**：一条规则由一个或多个匹配条件构成，多个匹配条件间用 **与** 的关系连接。
- **操作**：编辑规则、删除规则。

匹配条件

规则匹配条件支持使用系统字段或自定义字段做匹配。

- **系统字段**：
  - 调用方应用名
  - 调用方 IP
  - 服务方应用名
  - 服务方服务名
  - 服务方方法名
- **自定义字段**：支持自定义字段名。
- **逻辑关系**（操作符）：
  - 等于
  - 不等于
  - 属于
  - 不属于
  - 正则：使用正则表达式匹配。
- **字段值**

6.3.2 创建规则

本文介绍如何快速为一个服务创建并启用访问控制规则。

操作步骤

1. 在微服务平台页面，选择 **微服务 > 服务管控**。
2. 在页面上方的搜索栏中，输入目标应用或服务名称，点击搜索图标。
3. 在搜索到的服务列表中，点击目标服务 ID。
4. 进入服务详情页后，选择 **服务治理** 页签。
5. 选择访问控制模式：
   - 启用白名单：只允许符合白名单规则的服务调用请求，拒绝白名单外的所有请求。
   - 启用黑名单：拒绝符合黑名单规则的服务调用请求，允许黑名单外的所有请求。
6. 点击 **添加规则**，输入 **规则名**，由汉字、英文字母、数字、下划线组成。
7. 编辑 **匹配条件**：
   - 选择 **字段类型** 与 **字段名**：
     - 系统字段：提供的字段名有 调用方应用名、调用方 IP、服务方应用名、服务方服务名、服务方方法名。

- 自定义字段：支持自定义字段名。

- 选择 **逻辑关系**：
  - 等于
  - 不等于
  - 属于
  - 不属于
  - 正则：使用正则表达式匹配
- 输入待匹配的 **字段值**。

8. 点击 **保存**。
9. 点击 **启用规则**。

6.3.3 管理规则

编辑规则

您可以根据业务需要修改现有的访问控制规则。操作步骤如下：

1. 在访问控制规则列表中，点击目标规则右上角的 **编辑规则** 按钮。
2. 修改规则内容，支持以下操作：
   - 修改规则名称
   - 修改现有匹配条件
   - 新增其他匹配条件
3. 点击 **保存**，完成规则修改。

删除规则

您可以将已禁用的规则从规则列表中删除。操作步骤如下：

1. 在规则列表中，点击目标规则右上角的 **删除** 按钮。
2. 弹出窗口中，点击 **确定**。

7 动态配置

7.1 概述

动态配置是一个配置管理框架，可以在分布式环境下、运行期动态管理应用集群配置参数。动态配置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，用户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。

- **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。

7.2 开始使用动态配置

说明：请使用 SOFABoot 2.3.0 及以上版本。
开始使用前，请确认您已经完成环境配置，详情见 前置条件 。

使用动态配置的步骤为：

1. 本地开发
2. 发布应用
3. 云端管控动态配置类

完整的工程示例参见 动态配置教程 。

本地开发

SOFABoot 2.3.0 起，所有中间件的 maven 坐标已统一规范，现有文档仅提供最新写法，升级兼容相关变更见 ： SOFABoot 发布说明 。

在 SOFABoot 工程中，仅需在 POM 中增加以下依赖。注意动态配置对 SOFABoot 父 POM 版本有要求，需要使用最新的 SOFABoot 版本，无需关注 ddcs-enterprise-sofa-boot-starter 的版本。

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，安全配置请参考 引入中间件 。

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

然后，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;
}
}
```

Spring 配置示例：

```
<bean id="dynamicConfig"class="com.antcloud.tutorial.configuration.tutorial.config.DynamicConfig"init-
method="init"/>
```

1. 首先，要提供一个普通的 Java 类，称之为配置类。该配置类它要符合 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。
2. 在配置类上加上 @DObject 注解，注意它的包名是 com.alipay.drm.client.api.annotation。@DObject 需

要提供属性 id、region、appName。

- 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 是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。

3. 在资源属性上加上 @DAttribute 注解。注意包名同样是 com.alipay.drm.client.api.annotation。
4. 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，可能想要改变 get、set 方法的内容，而不是简单的赋值、取值。这是可以的，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。
5. 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，例如打日志，可以提供参数 (String,Object)，无返回值的方法，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。

6. 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。

发布应用

SOFABoot 应用的发布参见 SOFABoot 快速开始 - 云端运行 。

云端管控动态配置类

应用发布完成后，您需要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，详情参见 新增动态配置 和 推送动态配置 。

7.3 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。

1. 在微服务平台页面，选择 **微服务 >动态配置**。
2. 点击 **新增配置类**。
3. 输入以下必填信息：
   - 所属域：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：配置类所属的应用名。
   - 类标识：必须与代码中配置类的注解 @DObject 中的字段保持一致。
   - 描述：自定义的描述信息。

4. 点击 **确定**，完成新增。
5. 点击 **新增属性**，输入属性名与描述。属性为 key-value 的形式。具体属性值在推送至服务器时定义。
6. 点击 **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。更多相关信息请参见 推送动态配置 。

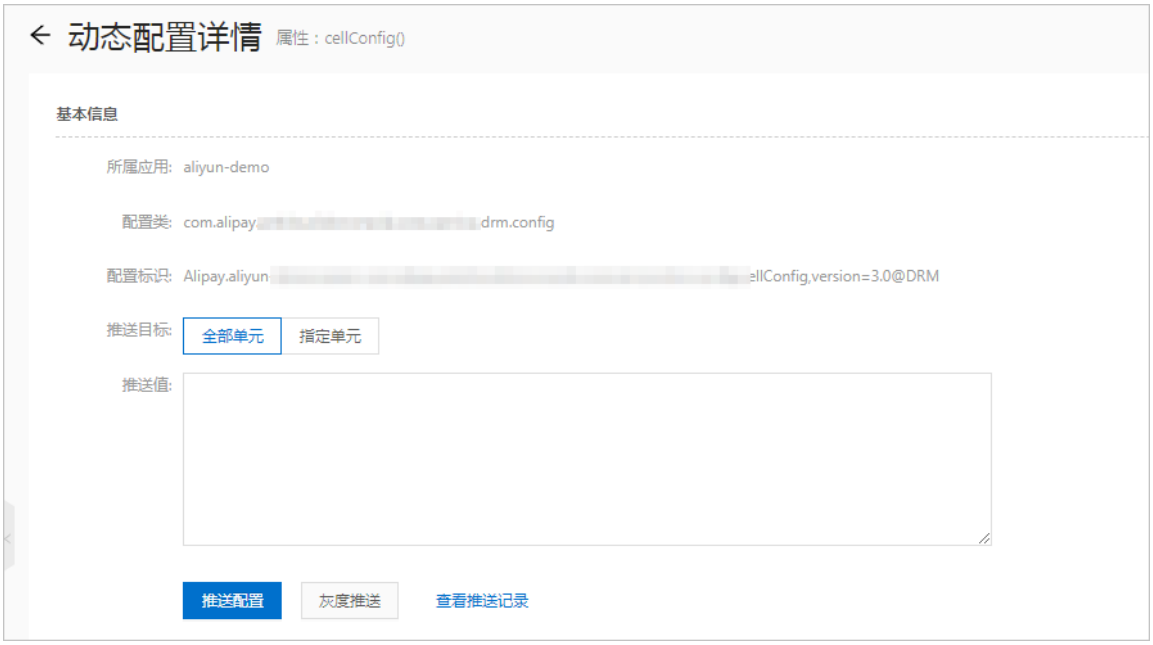
7.4 推送动态配置

您可以根据实际需求选择推送动态配置的方式：

- **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，并不保存入数据库。

操作步骤

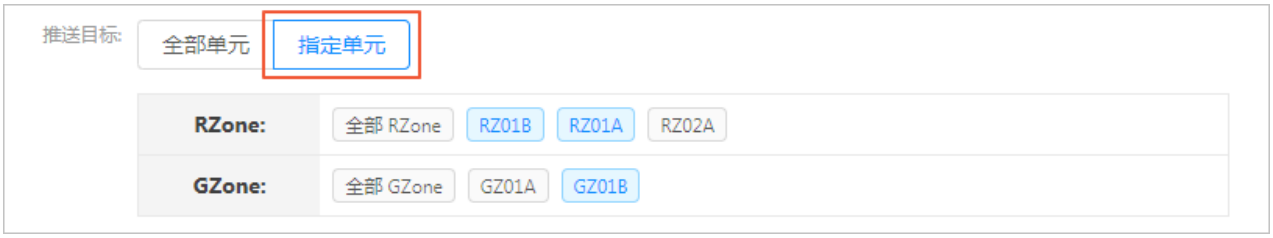
1. 前往 **微服务平台 > 微服务 > 动态配置** 页面，在列表中点击要推送的配置类前的加号，展开属性列表。
2. 点击要推送的属性名称，进入属性基本信息页面。
3. 选择推送目标：**全部单元** 或 **指定单元**。
4. 输入推送值。



5. 选择推送方式：
    - 直接推送：点击 **推送配置** > **确定**。
    - 灰度推送：
        - 点击 **灰度推送**。
        - 在弹窗中勾选要推送的机器 IP 地址。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。
        - 点击 **推送**。

单元化说明（仅专有云）

对于专有云开启了单元化能力的用户，推送目标支持指定具体 Zone 具体单元进行按单元化推送，如下图所示。



查看推送记录

点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：Success（成功）或 Failure（失败）。

点击 **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。

7.5 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，参见 开始使用动态配置 了解如何完全使用注解方式配置一个动态配置类。通过本文，您将了解如何覆盖注解配置，实现更灵活的动态配置类初始化。

本文包含以下内容：

- 覆盖注解配置的方法
- 属性注解高阶用法

覆盖注解配置的方法

动态配置客户端提供两种方式注册配置类：

```
# 1. 直接注册含有所有注解配置的配置类

DistributedResourceManager#register(Object resourceObject);

# 2. 注册配置类实例的同时，传入覆盖注解的配置项，Config 包含 @DObject 中的所有属性，Config 中配置的值会覆盖注解配置

DistributedResourceManager#register(Object resourceObject, Config config);
```

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。

7.6 导入导出元数据

动态配置提供元数据导入、导出，以方便用于数据迁移。操作步骤如下：

- 进入 **动态配置** 页面，点击 **更多 > 导出** 按钮，导出文件。

- 点击 **更多 > 导入** 按钮，可以导入任意一版产品导出的文件。

导入数据文件格式

数据导入功能主要用于跨环境数据迁移，使用 JSON 格式。数据文件每一行对应一个完整的配置类 JSON 结构。多个配置类的 JSON 数据以换行符分隔。

内容格式

```
{"region":"Alipay","appName":"testModel","name":"测试配置
","resourceId":"com.alipay.test","attributes":[{"attributeName":"age","name":"年龄
"},{"attributeName":"name","name":"名称"}]}
```

说明

- 文件格式可为 txt
- 每行一个配置类，配置类中可包含多个属性，多个配置类按换行符分割
- 参数说明
    - region：域，配置类的一个命名空间，默认值为Alipay
    - appName：配置类所属的应用名
    - resourceId：类标识
    - name：配置类的描述
    - attributes：属性
        - attributeName：属性名
        - name：属性的描述

示例

test.txt

```
{"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"}]}
```

7.7 教程

7.7.1 使用动态配置

在本教程中，您将通过一个示例了解动态配置的业务编码和日志排查。

前序课程

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。

- 配置搭建 SOFABoot 基础环境： 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： SOFABoot 编译运行 ；
- 如果需要在服务器上部署示例代码： SOFABoot 应用发布 。

下载示例代码

点击此链接 下载示例工程，动态配置项目示例代码位于 middleware-v2/dynamic-configuration-tutorial 文件夹下。

课程结构

1. 云端管控 ：示例资源的录入
2. 发布应用 ：示例代码下载，服务器部署运行
3. 推送资源 ：管控端资源属性值操作
4. 查看运行结果 ：结合日志确认操作结果
5. 代码解析

云端管控

参考 快速开始 云端管控动态配置类。

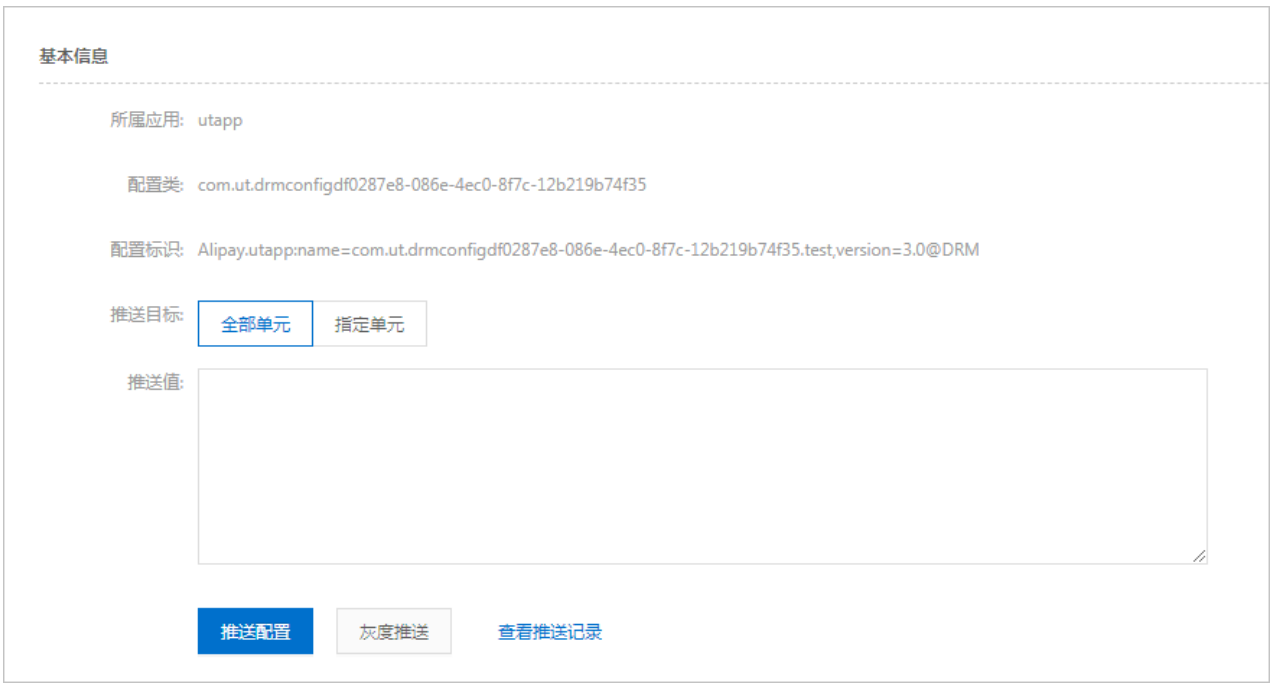
发布应用

参考 发布应用 把应用部署到云端。

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。

推送资源

可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。



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

通过网页端 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

查看 /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

如果客户端启动正常，在推送资源后，您可以查看动态配置的推送日志，即 /home/admin/drm/drm-monitor.log 。

- “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，称之为配置类，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；
- 正真的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”。

7.8 常见问题

本文汇总了动态配置（DRM）在应用中的一些常见问题及对应的解决方案。

- 发布部署卡在部署服务中，直到超时，导致发布部署失败

发布部署卡在部署服务中，直到超时，导致发布部署失败

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。

原因

在 DRM 中，RefreshCacheDRM.refreshCacheType = GEOHASH，业务代码在收到该项更新后，花费了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。

8 限流熔断

8.1 概述

限流熔断 (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。

限流目标与范围

- 支持对 RPC 接口和普通 Bean 的方法进行限流。

- 支持方法限流和方法参数条件限流，参数条件限流支持 MVEL 的完整能力。
- 支持多个方法合并限流。
- 支持对 Web 请求以及 Web 请求的参数条件限流。

功能特性

- 支持监控模式和拦截模式。
- 支持切换限流算法：QPS 计数算法和令牌桶算法。
- 支持多种限流条件：单位时间计数、堆内存使用量、CPU 负载、并发线程数。
- 支持多种限流后置处理，包括空处理（丢弃调用）、抛出异常等。

依赖

限流熔断的规则配置依赖于动态配置推送，所以接入限流熔断前必须先接入动态配置。

8.2 开始使用限流熔断

开始使用前，请确认您已经完成环境配置，详情见 前置条件。

使用限流熔断限流的流程为：

1. 开发编码并接入限流熔断客户端组件
2. 云端部署应用
3. 配置限流任务

开发编码并接入限流熔断客户端组件

引入 Maven 依赖

1. 接入动态配置客户端，详见 动态配置快速开始 。
2. 在工程 pom.xml 文件中引入限流熔断依赖：

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

配置 Spring Bean 和 AOP 拦截器

在上一步添加 guardian-sofa-boot-starter 后，应用已经可以对 SOFARPC 接口和 Spring MVC 请求进行限流。

如果还要对内部通过 Spring Bean 定义的方法限流，则需要在 Spring Bean 配置文件中添加配置 AOP 拦截器。示例如下：

```
<!-- 引入限流熔断中定义的 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>
```

云端部署应用

将开发完成的应用部署到金融科技平台上，以开始对平台上的应用进行限流配置。操作步骤请参考 SOFABoot 快速入门 > 发布部署 。

配置限流任务

应用部署完成后，您必须前往 **微服务平台 > 微服务 > 限流熔断** 进行限流任务的创建与配置，详情参见 新增限流规则 。

8.3 新增限流规则

您可以为每个应用设定多个限流规则，以应对大规模突发流量，增加系统稳定性。

1. 如果要限流的应用不在应用列表中：
    - 点击 **新增应用。**
    - 输入需要限流的应用名称。
    - 点击 **确定。**

      说明：如出现“数据库异常”错误，请检查输入的应用名称是否已经存在。

2. 选择要限流的应用，点击 **新建规则**，跳转至规则详情页面。
3. 配置以下规则信息，所有内容均为必填项。有关规则的详细说明，请参见 限流规则说明 。
    - 规则名称
    - 限流类型
    - 运行模式
    - 限流算法
    - 限流后置操作

- 限流条件模型及阈值
- 限流对象

4. 选择提交或灰度推送：
   - **提交**：确认保存并提交规则。
   - **灰度推送**：将更改后的规则推送到指定的几台机器上用以验证。更改的规则仅保存在被推送的服务器内存中，不写入数据库。 参见 使用灰度推送 。

若要将同一规则作用于多个应用，请参考 导入导出限流熔断配置 。

8.4 设置限流算法

限流熔断提供了两种限流算法：QPS 计数算法和令牌桶算法，详细介绍参见 限流算法介绍 。

选择限流算法

- 单机 QPS < 100 时，建议使用使用令牌桶算法。
- 单机 QPS > 100 时，可以选择 QPS 算法和令牌桶算法。
- 不能容忍单个周期放过的请求数超过限流值时，选择 QPS 算法。

令牌桶系数

表示存量桶容量与限流值之间的关系，默认值为 1.0，表示存量桶容量是 1.0 倍的限流值；推荐设置值为 0.6~1.5。

8.5 配置参数条件过滤

对于接口方法类的限流规则，如果需要指定限流的指定具体的接口及方法，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，若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：对于方法 foo(int a, int[] al)，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)

8.6 配置限流对象方法签名

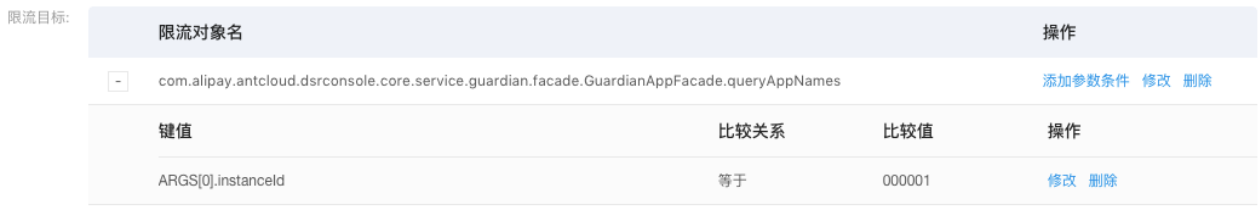
限流熔断可以对方法的参数进行过滤，可实现对某个特定的参数进行限流。

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
限流对象名	包括要限流的接口与方法名： • 接口名：支持 RPC 服务接口或配置了 Spring AOP 拦截器的 Bean。 • 方法名：支持带参数或不带参数的方法签名。有关方法重载的说明参见 方法配置说明 。
键值	限流参数及属性名称，用 MVEL 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

下图给出了一个参数配置的样例：



- 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 是限流熔断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。

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。例如请求：/queryAllNames?instanceId=00001&name=cloudinc，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。

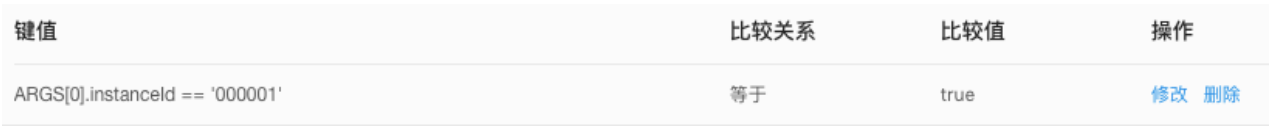
下图给出了针对请求 URL http://xxx.domain//webapi/guardian/history/search?instanceId=000001 限流对象配置样例：

其中，instanceId 是请求 URL 中的参数的键值，00001 是参数中的属性值。

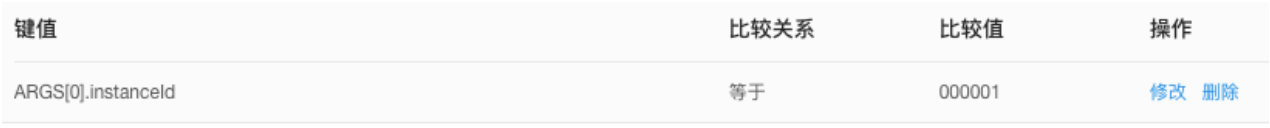
接口方法中的 MVEL 表达式

配置方式

- 方式一：左侧键值计算结果是 true/false，右侧比较值中也填写 true/false，例如：



- 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，例如：



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：

可使用 obj.field 的格式获取参数的属性值。属性必须有 public 的 getter 方法，或是本身是 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，或者 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/黑名单范围内的场景。限制：白名单/黑名单列表的元素，不能超过 100 个。

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。

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'。

8.7 使用灰度推送

在您编辑限流规则时，若不确定规则是否正确，可以使用灰度推送功能，将限流规则推送到限定的几台机器上用以测试。

使用步骤如下：

1. 在规则列表中选择要推送的规则，点击 **修改** 进入规则编辑页面。
2. 点击底部的 **灰度推送。**
3. 在弹窗中选择要推送的机器 IP 地址。只有该规则的订阅者机器 IP 会出现在列表中。您可以使用弹窗右上角的搜索栏进行快速筛选。
4. 点击 **推送**。
5. 在对应的服务器上验证限流结果。

说明：限流熔断使用动态配置的灰度推送功能。灰度推送的数据不会保存到数据库，只会保存在于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，仍使用更改前的规则。

8.8 导入导出限流规则

若要将同一规则作用于多个应用，您可以导入导出限流规则，进行规则迁移。

导出限流规则

限流规则的导出是以应用为维度进行的，在应用右侧的 **操作** 列中点击 **更多** > **导出** 即可导出限流规则，如下图

：

导出数据格式说明

导出的数据为 JSON 格式，格式化后如下：

```
[
{
"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.GuardianAppFacade.query",
"resourceType":"METHOD",
"ruleIds": [ ]
},
{
"baseName":"11.22",
"resourceType":"METHOD",
"ruleIds": [ ]
}
],
"resourceType":"METHOD",
```

```
"runMode":"CONTROL",
"trafficType":"ALL"
}
]
```

说明：

- actionConfig：后置处理动作
  - 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：限流计算阈值
  - 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
  - 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

导入限流规则

在应用右侧的 **操作** 列中点击 **更多** > **导入**，可通过导入上文所述的 JSON 文件来导入限流规则。

注意：

- 导入的限流规则均默认为不开启，如果需要启用，需要在界面上进行手动开启。
- 导入和导出均以应用为维度进行，导出的规则可以导入至任何一个其他应用中。
- 系统会根据规则名称过滤掉已存在的规则，所以在同一个应用中，同一个规则不会被重复导入。

8.9 参考

8.9.1 限流规则说明

限流规则的定义包括以下维度:

- 限流类型
- 运行模式

- 限流算法
- 限流后置操作
- 限流条件阈值
- 限流对象

限流类型

- **接口方法**：支持对某个具体的 RPC 接口或普通 Bean 的方法限流。要求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。要求在限流对象中配置请求 URI。

运行模式

- **拦截模式**：限流生效的模式，若匹配上规则，会将方法调用进行限制，调用配置的“限流后操作”。
- **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

- **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**

有关算法的详细说明参见 限流算法说明 。

限流后置操作

限流操作	适用于接口方法限流	适用于Web 页面限流	解释
空处理	Y	Y	不做任何处理，返回空值。
抛出异常	Y	N	异常信息为填写的输入框内容。
跳转到指定页面	N	Y	跳转到指定的页面地址。
页面json报文	N	Y	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"}
页面xml报文	N	Y	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>

限流条件阈值

- **条件模型**

条件模型	限流阈值	说明
单位时间内服务访问次数或 Web 页面访问次数	QPS 计数值	根据单位时间内的请求数进行限流。
堆内存使用量	最大堆内存使用量（单位为兆 MB）	根据当前堆内存使用量进行限流。
CPU 负载	100 * CPU 负载百分比	根据过去一分钟内的 CPU 平均负载进行限流。
并发线程数	最大并发线程数	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**：见上表。
- **流量类型**：
    - 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：仅对压测流量线路。

限流对象

- 对于接口方法，配置要限流的接口路径名称和方法签名。
- 对于 Web 页面方法，配置要限流的请求 URI。

更多详情，请参考 配置限流对象参数 。

8.9.2 限流算法介绍

常见的限流方式有：

- 通过限制单位时间段内调用量来限流（QPS 限流算法）
- 通过限制系统的并发调用程度来限流
- 使用漏桶（Leaky Bucket）算法来进行限流
- 使用令牌桶（Token Bucket）算法来进行限

限流熔断中主要使用了其中两种：**QPS 限流算法** 和 **令牌桶算法**。

QPS 限流算法

QPS 限流算法通过限制单位时间内允许放过的请求数来限流。

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。

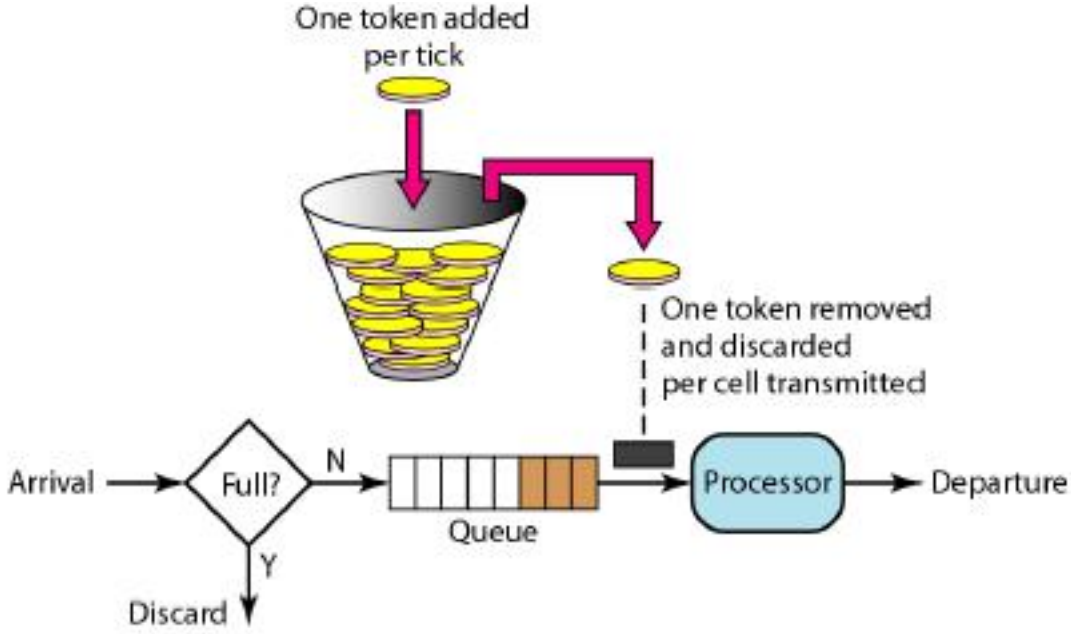
- 可以通过拉长限流周期来应对突发流量。如 1 秒限流 10 个，想要放过瞬间 20 个请求，可以把限流配置改成 3 秒限流 30 个。拉长限流周期会有一定风险，用户可以自主决定承担多少风险。

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里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：限流熔断是对单机进行限流，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。

8.9.3 限流日志

限流熔断的限流日志打印在 logs/guardian 中，该路径下存在多个日志文件，分别打印不同的日志内容。

- 限流熔断默认日志
- 限流熔断运行错误日志
- 限流熔断限流统计日志

限流熔断默认日志

限流熔断的默认日志是 guardian/guardian-default.log，主要打印推送下来的限流配置信息，日志内容没有固定格式.

样例：

```
2016-12-12 19:49:09,610 INFO Registring GuardianCodeWrapperInterceptor
2016-12-12 19:49:10,757 WARN receive message with key=[guardianConfig] and
value=[{"@type":"com.alipay.guardian.client.drm.GuardianConfig","engineConfigs":{"@type":"java.util.HashMap","LI
MIT":{"@type":"com.alipay.guardian.client.engine.limit.LimitEngineConfig","actionConfigMap":{"@type":"java.util.Ha
shMap",880:{"@type":"com.alipay.guardian.client.engine.limit.LimitActionConfig","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.guardiante
stsofalite.facade.GuardianTestTrServiceFacade.testLimitBasicCondition","id":379,"resourceType":"METHOD","ruleIds"
:[880]}],"ruleConfigMap":{"@type":"java.util.HashMap",880:{"@type":"com.alipay.guardian.client.engine.limit.LimitRul
eConfig","actionId":880,"calculationConfigs":[{"calculationType":"INVOKE_BY_TIME","maxAllow":10,"period":5000},{"
calculationType":"INVOKE_BY_TIME","maxAllow":10,"period":5000},{"calculationKey":"[0].booleanValue","calculation
Type":"INVOKE_BY_TIME_CATEGORY","period":5000,"tairCompareKey":"true>5,false>6"}],"enable":true,"extParamCo
nfigs":[],"id":880,"limitType":"GENERIC_LIMIT","paramConfigs":[{"checkMode":"BYVALUE","compare":"EQUALS","key"
:"[0].stringValue","value":"testStrMutilBasicParams"},{"checkMode":"BYVALUE","compare":"EQUALS","key":"[1].string
Value","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.alipay.guardian.client.engine.fuse.FuseEngineCo
nfig","actionConfigMap":{"@type":"java.util.HashMap"},"ruleConfigMap":{"@type":"java.util.HashMap"}}},"version":1}
]
2016-12-12 19:49:10,759 WARN after update with key=[guardianConfig]
2016-12-12 19:49:11,195 INFO Guardian Config version=1
2016-12-12 19:49:11,197 WARN rebuild Rules, GuardianFactory: class
```

```
com.alipay.guardian.client.limit.LimitGuardianFactory
```

限流熔断运行错误日志

限流熔断的运行时错误日志是 guardian/guardian-error.log，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。

限流熔断限流统计日志

限流熔断的限流统计日志是 guardian/guardian-limit-stat.log，日志内容的固定格式如下：

```
CONTROL/MONITOR,id,规则名称,统计间隔,开始时间,结束时间,统计类型,限流阈值,总请求数,放行数,限流数
```

- MONITOR：表示当前的限流模式是监控模式
- CONTROL：表示当前的限流模式是拦截模式
- 倒数第四位：限流规则阈值
- 倒数第三位：限流周期内的总请求数
- 倒数第二位：限流周期内的放行请求数
- 倒数第一位：表示当前限流周期内被限流的请求数

样例：

```
2016-11-21 00:00:02,001 INFO MONITOR, 43,规则名字,1000,2016-11-21T00:00:01,2016-11-
21T00:00:02INVOKE_BY_TIME,10,40,10,30
```

8.10 常见问题

本文汇总了限流熔断（Guardian）在应用中的一些常见问题及对应的解决方案。

- 在业务程序启动的时候，Guardian 没有加载

在业务程序启动的时候，Guardian 没有加载

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。

原因

业务程序打包时，没有引入 Guardian 的 Jar 包。

解决方案

解决客户的打包问题，确保引入 Guardian 的 Jar 包。

9 故障排查

9.1 错误 RPC-02306：无法获取 RPC 服务地址问题

问题现象

RPC 客户端调用服务时，收到如下错误：

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

排查思路

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，可以用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，建议首选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
```

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。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。

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。

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 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，这种情况多发生在业务系统自己通过定时任务或者在 bean 初始化完成后就开始调用服务，而此时客户端应用还没启动完成。此种情况可以通过配置如下 address-wait-time 解决。

配置项	描述	默认值
address-wait-time	reference 生成时，等待服务注册中心将地址推送到消费方的时间。address-wait-time 的最大值为 30000 ms，超过这个值的配置将调整为 30000 ms。	0 (ms)

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，查看以下参数是否配置相同，如配置不同，RPC 客户端无法感知 RPC 服务端。

- com.alipay.instanceid

- com.antcloud.antvip.endpoint

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。

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。

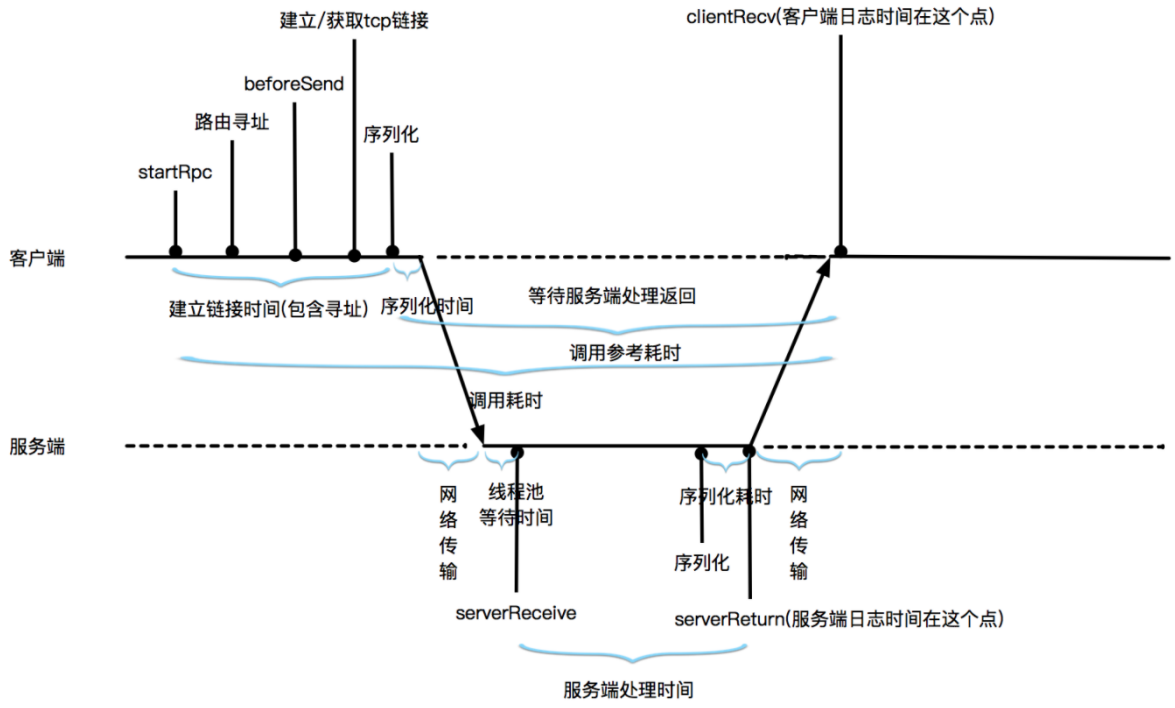
9.2 如何排查 RPC 超时问题

问题现象

若调用 RPC 服务超时，在客户端的 logs/tracelog/middleware_error.log 日志中，可看到如下异常信息：

```
2018-07-06 13:21:20.463,sofa2-rpc-client,707c27b915308544774611046466 3,0,main,timeout_error,rpc,invokeType=sync&uid=&protocol=bolt&targetApp=sofa2-rpc-server&targetIdc=&targetCity=&paramTypes=&methodName=message&serviceName=com.alipay.share.rpc.facade.SampleService:1.0&targetUrl=10.160.34.141:12200&targetZone=&,,com.alipay.sofa.rpc.core.exception.SofaTimeOutException: com.alipay.remoting.rpc.exception.InvokeTimeoutException: Rpc invocation timeout[responseCommand TIMEOUT]! the address is 10.160.34.141:12200
```

RPC 调用时序图



有关客户端和服务端各阶段的耗时信息，请参考 RPC Tracer 日志 。

排查思路

PRC 调用超时一般可以按照如下顺序逐步排查：

1. 服务本身确实超时 ，如业务代码处理时间过长。
2. 服务端 RPC 线程池耗尽 。
3. 发生垃圾回收 （ Garbage Collection，简称 GC ），导致线程停止。
4. 发生网络问题 。
5. 其他外部因素影响服务器性能，如定时任务、批处理，或者与宿主机上其他虚拟机、容器发生资源争抢。

服务本身超时

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 707c27b915308544774611046466 3，找到服务端处理对应请求的日志。

日志格式如下所示：

```
2018-07-06 13:21:22.441,sofa2-rpc-server,707c27b915308544774611046466 3,0,com.alipay.share.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：

- 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，定位问题。
- 服务端有数据库操作，如数据库连接耗时、慢 SQL 等。

服务端 RPC 线程池耗尽

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 RPC 线程池大小、队列长度配置说明 。

GC 问题

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

1. 用 kill -15 命令结束服务端进程。
2. 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**：**环境标识** 可在控制台中选择 **产品与服务** > **金融分布式架构** > **环境资源管理** > **环境** 中查看。

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。

网络延时抖动

排查是否由于网络问题导致RPC调用超时:

1.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。

2.在客户端和服务端同时部署 tcpdump 进行循环抓包，问题发生后分析网络包。

3.在客户端和服务端运行 ping 观察是否存在网络延时。

打印客户端RPC调用统计

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。实际使用时，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。

```
$ grep sofa2-rpc-server rpc-client-digest.log | awk -F, '{if(int($18)>3000)print $9,$10,$27}' |sort | uniq -c | sort -n
```

10 常见问题

本文汇总了 SOFAStack 微服务中各个功能组件的常见问题。

- SOFARegistry 常见问题
- SOFARPC 常见问题
- 动态配置常见问题
- 限流熔断常见问题

11 定时任务（仅专有云）

11.1 概述

> **重要**：目前，定时任务功能已在 SOFAStack 微服务公有云下线，仅在专有云暂时保留。如需定时任务服务，建议您选择 **任务调度** 产品，详见 任务调度文档 。

定时任务服务旨在为业务系统提供统一通用的任务调度服务，提供定时任务的管理监控平台，减轻业务系统开发和后续线上运维的工作量。

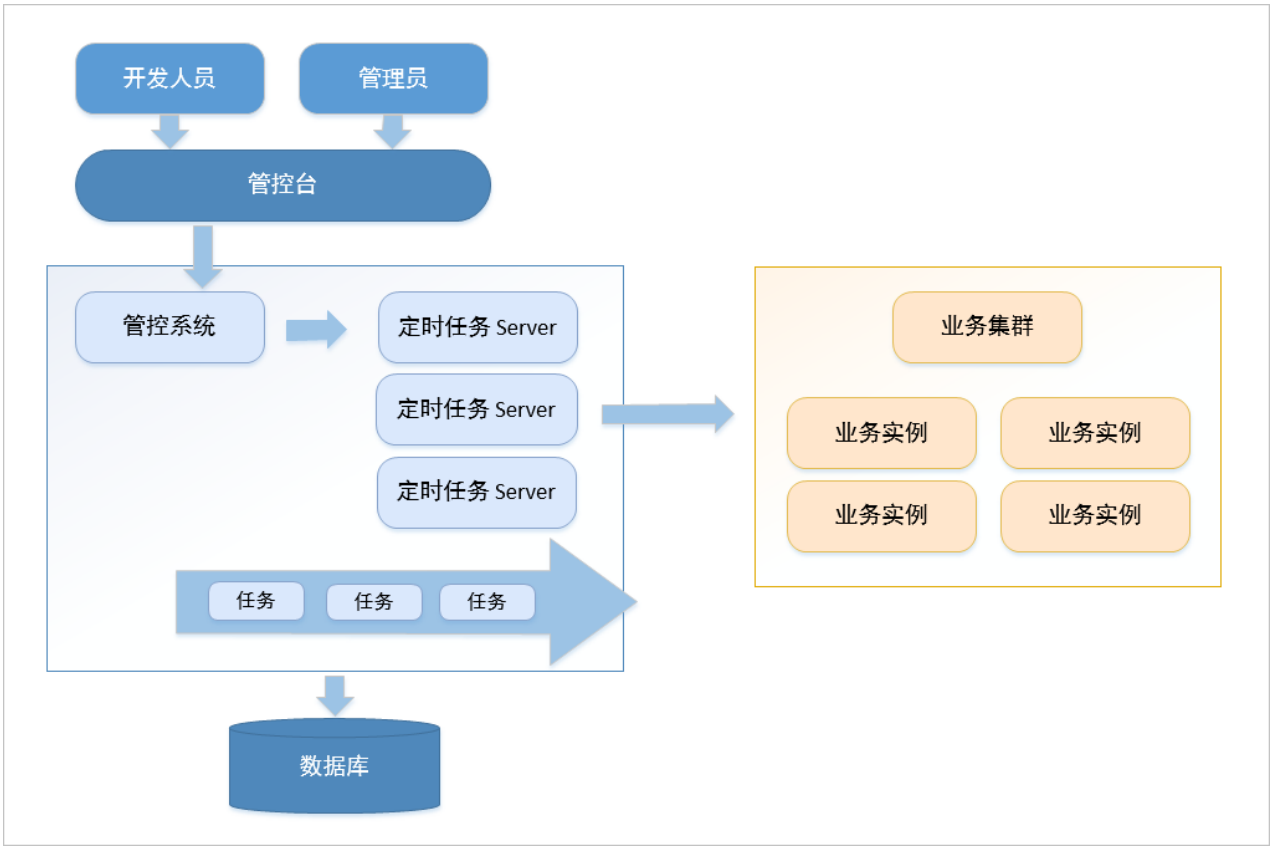
功能和目标：

- 提供统一的定时任务注册和管理监控平台；
- 提供集中的定时调度；
- 提供特殊时段（停机维护）的支持；
- 提供便捷的测试支持。

核心场景

定时任务服务端按照用户配置的定时任务信息，一到任务执行时间就向应用发起 RPC 请求。应用收到请求后，开始执行自己预设的任务逻辑。如果是回调类型的任务，执行完毕后回调服务端。

部署图



1. 开发人员和管理员在控制台界面上配置管理定时任务。
2. 调度系统会将任务元数据固化到数据库，按照配置参数定时调用客户端。
3. 业务集群系统接收 RPC 请求后，执行实际的业务逻辑，完成定时触发的效果。
4. 如果是回调类型的任务，执行完毕后回调服务端。

11.2 开始使用定时任务

定时任务组件依赖消息队列（Message Queue）。在使用定时任务之前，您必须先开通消息队列服务。

开始使用定时任务前，请确认您已经完成环境配置，详情见 前置条件 。

使用定时任务中间件实现定时任务功能的流程为：

1. 开发编码
2. 发布应用至云端
3. 云端配置定时任务

开发编码

Maven 依赖

在接收定时任务的模块的 pom.xml 中添加如下依赖：

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

任务处理器代码

```
public class SchedulerDemo implements ISimpleJobHandler {

ThreadPoolExecutor threadPoolExecutor;

@Override
public String getName() {
return"DEMOAPP_DEMOTASK";
}

@Override
public ThreadPoolExecutor getThreadPool() {
return threadPoolExecutor;
}

@Override
public ClientCommonResult handle(JobExecuteContext context) {
boolean success = true;

// 业务逻辑代码

if (success) {
return ClientCommonResult.buildSuccessResult();
} else {
return ClientCommonResult.buildFailResult("handle failed");
}
}

public void setThreadPoolExecutor(ThreadPoolExecutor threadPoolExecutor) {
this.threadPoolExecutor = threadPoolExecutor;
}

}
```

任务处理器实现 com.alipay.antschedulerclient.handler.ISimpleJobHandler 接口及接口内的所有方法：

- getName() 返回字符串，必须跟页面配置的任务名称一致；
- getThreadPool() 执行该任务所用的线程池，可以每个任务处理器注入独立的线程池，也可以多个任务处理器共享一个线程池，如果返回 null 则使用公共的默认线程池；
- handle(JobExecuteContext context) 编写任务执行逻辑，需返回 ClientCommonResult 表示任务执行结果。

Spring 配置

您可以通过以下两种方式将实现了 ISimpleJobHandler 接口的类 SchedulerDemo 声明为 Spring Bean：

- 在 Spring XML 中定义 <bean/>。示例如下：

```
<bean id="schedulerDemo"class="com.antcloud.tutorial.scheduler.SchedulerDemo"/>
```

- 使用注解驱动（annotation-driven）的方式声明 Bean。

application.properties

每个环境需要连接的服务端不一样，参见 RPC 引用服务 完成对应环境的 application.properties 参数调整：

- com.alipay.env
- com.alipay.instanceid
- com.antcloud.antvip.endpoint

发布应用

将应用发布至云端，SOFABoot 应用的发布参见 SOFABoot 快速入门 。

云端配置定时任务

完成应用发布后，您需要前往微服务控制台进行定时任务的创建与管理，参见 新增与管理定时任务 。

11.3 新增与管理定时任务

本文介绍如何在定时任务控制台新增与管理定时任务。

新增定时任务

在定时任务控制台中，点击页面上方的 **新增定时任务** 按钮，按提示输入任务信息，点击 **确定**。

任务基本信息

- **任务名称**：推荐命名格式为 APPNAME_FUNCTION，注意需要和实际代码保持一致。
- **应用名称**：需要和工程中配置的应用名称一致，任务执行时将按照应用名称发送请求。
- **CRON 表达式**：定义任务执行周期及时间的字符串。具体语法格式请参考 CRON 表达式详解 。微服务平台调度中心使用 Quartz 来实现定时执行，配置规则可参见 Quartz 官网。

说明：在开发联调初期，建议将 CRON 表达式设置为 0 0 0 * * ? 这种低频的形式，待开发完成后再调整为预期频率的自动执行。

高级设置

- **路由策略**：
  - **随机**：默认策略，每次触发都随机调用一个客户端，以达到负载均衡的目的。

    - **定向**：每次触发都固定调用一个客户端，以方便排查问题，但是不支持指定调用目标。
    - **轮询**：将连接到服务端的客户端 IP 地址排序，每次任务执行时按顺序选择一台客户端作为目标机器。
- **触发类型**：提供 ONEWAY 和 CALLBACK 两种类型。
    - ONEWAY 适用于频率较高的非重要任务，执行记录不写入数据库，页面不支持查看执行记录；
    - CALLBACK 适用于低频重要任务，每次的执行记录都写入数据库，必须回调成功才算执行成功，提供多种失败处理策略，任务触发间隔必须大于 5 分钟。
- **超时时间**：适用于触发类型是 CALLBACK 的任务，为必填项。单位为分或者秒。超过此时间未回调则认为执行失败。
- **失败处理策略**：适用于触发类型是 CALLBACK 的任务，为必填项。提供三种策略：
    - 不重试
    - 重试三次
    - 重试到下次触发
- **描述**：非必填项。定时任务的详细描述，例如业务含义、影响范围等。

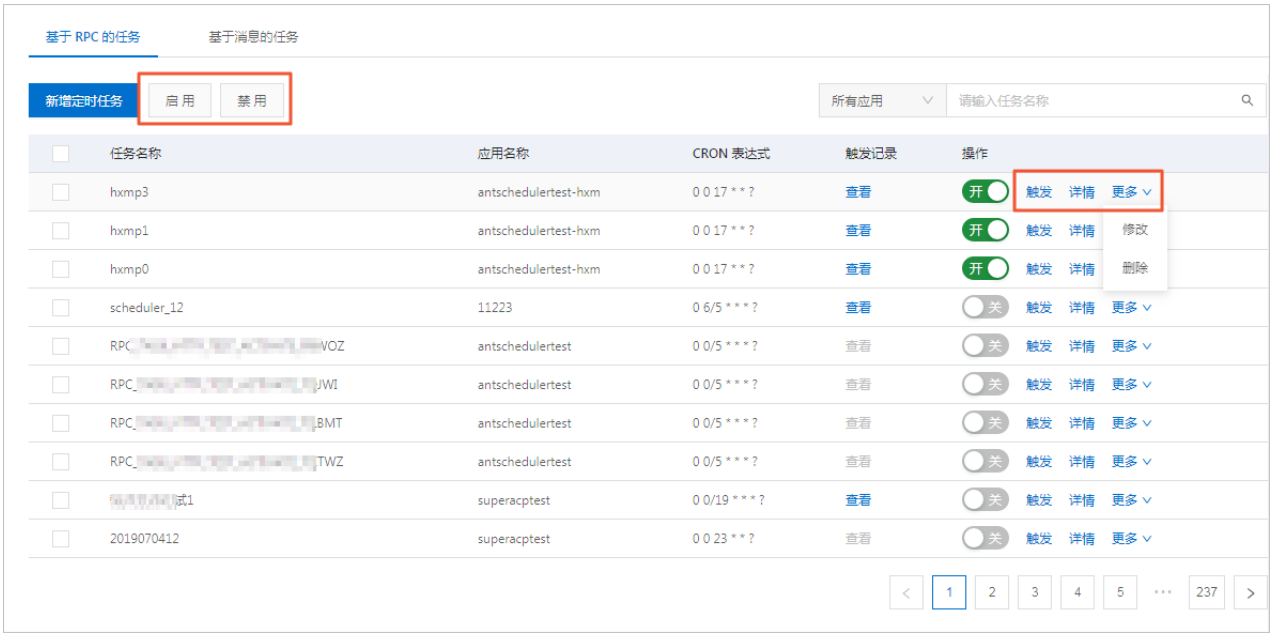
新增完成之后，定时任务就按照预期的频率开始定时执行了。

管理定时任务

控制台提供定时任务的查询管理功能，包括：任务开/关、手动触发、修改、删除。

在定时任务界面上，除了新增任务以外，您还可以完成如下操作：

- **开/关**：界面显示为 **开** 时，任务会自动执行；将开关滑动至 **关** 后，，任务停止自动执行。
- **触发**：每手动点击一次 **触发**，任务就会在后台执行一次。
- **编辑**：调整任务名称、CRON、系统。
- **删除**：删除某个定时任务。



11.4 基于消息的定时任务

11.4.1 概述

基于消息的定时任务仅对存量老用户开放，新用户推荐使用基于 RPC 的定时任务。

定时任务服务旨在为业务系统提供统一通用的任务调度服务，提供定时任务的管理监控平台，减轻业务系统开发和后续线上运维的工作量，并通过任务拆分和负载均衡等方案提升大数据量任务的性能。

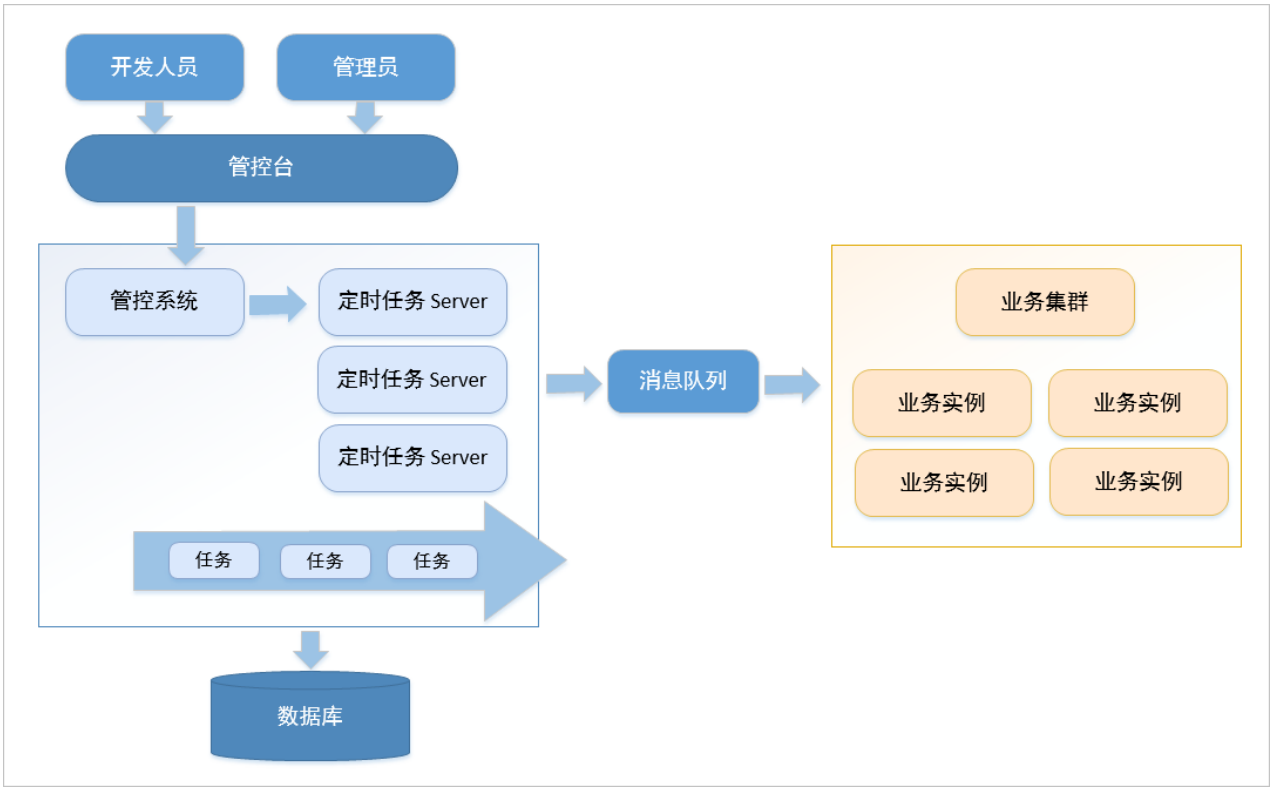
功能和目标：

- 提供统一的定时任务注册和管理监控平台
- 提供集中的定时调度、消息通知
- 通过任务拆分、调度数据托管提升大数据量任务的性能
- 提供特殊时段（停机维护）的支持
- 提供便捷的测试支持

核心场景

定时任务依赖于消息队列，定时任务服务端按照用户配置的定时任务信息，到了任务执行时间就向应用发送一个消息。应用接收到消息后，开始执行自己预设的任务逻辑。

部署图



1. 开发人员和管理员在界面上配置管理定时任务。
2. 调度系统将任务元数据固化到数据库，按照配置参数定时发送消息。
3. 业务集群系统接收经由消息队列发送的消息后，执行实际的业务逻辑，完成定时触发的效果。

11.4.2 基于消息的定时任务教程

通过本课程，您将学会如何使用定时任务，包括定时任务和消息队列相关的配置。本课程包括以下内容：

1. 编码 ：接入定时任务，参见 下载示例代码 ，阅读示例代码。
2. 管控台操作 ：完成本地编码后，需要在服务端管控台中提交相关的配置，涉及定时任务、消息类型、订阅关系配置。
3. 日志确认 ：在 SOFAStack 平台上部署示例应用后，您需要通过日志来确认是否已经开始正常运行。

前序课程

本课程的示例代码是基于 SOFABoot 开发的。学习本课程前，确保您对 SOFABoot 有一定程度的了解，并了解 **定时任务** 和 **消息队列** 的关系。

- 了解 SOFABoot 基础知识： SOFABoot > 快速开始
- 了解定时任务运行机制： 定时任务 > 概述
- 了解消息队列运行机制： 消息队列 > 概述

编码

下载代码

> **说明**：点击此处 下载示例工程。

项目示例代码位于 middleware-v2/schedulertutorial 文件夹下。

运行代码

定时任务的任务管控在云端，所以需要把应用部署到云端来运行。

配置调整

SOFABoot 中 application.properties 包含了工作空间相关的三个参数。在云上部署前，确认想要部署的目标工作空间及对应的三个参数，然后进行相应的调整，参考 SOFARPC 进阶指南 > 引用 SOFARPC 服务 。相关参数如下：

- com.alipay.env
- com.alipay.instanceid
- com.antcloud.antvip.endpoint

部署应用

示例应用是普通的 SOFABoot Core 工程，参考 SOFABoot > 在 SOFAStack 上运行 完成打包并将其部署到云端。

管控台操作

配置定时任务

参考 配置定时任务 完成配置。注意任务名称需要和代码中的 eventCode 保持一致，参考：schedulertutorial/schedulertutorial-service/src/main/resources/META-INF/schedulertutorial/schedulertutorial-service.xml。

配置消息类型和订阅关系

参考 消息队列 中的消息类型、订阅关系部分在消息队列控制台上新增消息主题、消费组以及订阅关系。

注意：

- 消息主题、消息事件码、消息订阅组需要和代码中保持一致，参考：schedulertutorial/schedulertutorial-service/src/main/resources/META-INF/schedulertutorial/schedulertutorial-service.xml。
- 配置消息主题、订阅关系后记得将其生效。

日志确认

在 SOFAStack 平台上通过网页端 SSH 登录至 schedulertutorial 的 ECS，消息队列传来的日志存储在 /home/admin/logs/tracelog/msg-sub-digest.log，格式如下。查看日志以确认示例应用是否开始正常运行。

```
2017-07-04 22:06:13.334,schedulertutorial,0ba5c70814991771740321854473 4,0.1.50bafee1,48502cafabf3ae16cbc29896f5cb4752,a1446577617326be85d2853a95c9b098,^8LFHHIA4ZPEX$TP_F_SC,EC_TASK_SCHEDULERTUTORIAL_DEMO,^000001$P-scheduler-prod,00,656B,37ms,dms-server-shared-1-1,0,msgWorkTP-1954168392-1-thread-1,,
```

11.4.3 新增与管理基于消息的定时任务

控制台提供定时任务的查询管理功能，包括：新增任务、任务开/关、手动触发、修改、删除。

新增任务

在 **定时任务 > 基于消息的任务** 界面，您可以输入任务名关键字搜索您的任务，也可以完成任务的新增。

调度任务主要有 **任务名称**、**应用名称**、**CRON 表达式** 三个关键项。

更多操作

定时任务新增之后，就会开始按照 CRON 表达式的频率执行。如果在开发联调阶段，不期望执行过于频繁，建议设置类似 0 0 0 * * ? 的低频表达式，或者将任务设置为 **关**。

在定时任务界面上，您可以完成如下的操作：

- **开/关**：界面显示为 **开** 时，任务会自动执行；将开关滑动至 **关** 后，，任务停止自动执行
- **触发**：每手动点击一次 **触发**，任务就会在后台执行一次
- **编辑**：调整任务名称、CRON、系统
- **删除**：删除某个定时任务

11.4.4 最佳实践

为让您能够更好地管理定时任务，保障开发环境和生产环境的稳定，本章节推荐一些编码规范。

任务名称推荐采用 EC_TASK_APPNAME_FUNCTION 格式

例如，系统名称为 scheduler，对应的一个定时任务功能为清理后台垃圾流水 truncate_flow_record，建议将任务名称命名为 EC_TASK_SCHEDULER_TRUNCATE_FLOW_RECORD。这种命名形式便于根据任务名称获取任务的具体业务功能。

开发环境 CRON 表达式推荐为 0 0 0 * * ?

0 0 0 * * ? 含义为每天的 0 点 0 分 0 秒触发一次。

由于开发环境的业务机器较少，如果在开发初期直接配置类似 * 0/1 * * * ? 的 CRON 表达式，即一分钟一次的高频率，一旦业务机器发生故障、重启或部署新代码，由于机器较少导致消息接收能力不足，从而导致大量的消息投递失败，而消息队列的高可靠特性会导致这些消息持续数天重试。

上述场景浪费机器资源，同时消息过多还会导致预期外的收费，影响开发环境的稳定。所以建议开发初期设置类似 0 0 0 * * ? 的低频任务，在联调时手动触发定时任务，在代码稳定后，再调整为预期的频率，进行稳定性测试。

11.5 CRON 表达式详解

CRON 表达式是一个字符串，以 5 或 6 个空格隔开，分为 6 或 7 个域，每一个域代表一个含义。CRON 有如下两种语法格式：

- 秒 分 小时 日期 月份 星期 年
- 秒 分 小时 日期 月份 星期

每个域允许的值

域	允许的数值	允许的特殊字符	备注
秒	0-59	- * /	
分	0-59	- * /	
小时	0-23	- * /	
日期	1-31	- * ? / L W C	
月份	1-12	JAN-DEC - * /	
星期	1-7	SUN-SAT - * ? / L C #	1 表示星期天，2 表示星期一，依次类推
年（可选）	留空，1970-2099	, - * /	自动生成，工具不显示该值

特殊字符的含义

字符	含义	示例
*	表示匹配域的任意值	例如：在分这个域使用 *，即表示每分钟都会触发事件。
?	表示匹配域的任意值，但只能用在日期和星期两个域，因为这两个域会相互影响。	例如：要在每月的 20 号触发调度，不管 20 号到底是星期几，则只能使用如下写法：13 13 15 20 * ?。其中，因为日期域已经指定了 20 号，最后一位星期域只能用 ?，不能使用 *。如果最后一位使用 *，则表示不管星期几都会触发，与日期域的 20 号相斥，此时表达式不正确。
-	表示起止范围	例如：在分这个域使用 5-20，表示从 5 分到 20 分钟每分钟触发一次。
/	表示起始时间开始触发，然后每隔固定时间触发一次	例如：在分这个域使用 5/20，则意味着 5 分钟触发一次，而 25，45 等分别触发一次。
,	表示列出枚举值	例如：在分这个域使用 5,20，则意味着在 5 和 20 分每分钟触发一次。
L	表示最后，只能出现在日和星期两个域	例如：在星期这个域使用 5L，意味着在最后的一个星期四触发。
W	表示有效工作日（周一到周五），只能出现在日这个域，系统将在离指定日期最近的有效工作日触发事件。	例如：在日这个域使用 5W，如果 5 号是星期六，则将在最近的工作日星期五，即 4 号触发。如果 5 号是星期天，则在 6 号（周一）触发；如果 5 号为工作日，则就在 5 号触发。另外，W 的最近寻找不会跨过月份。
L W	这两个字符可以连用，表示在某个月最后一个工作日，即最后一个星期五。	
#	表示每个月第几个星期几，只能出现在星期这个域	例如：在星期这个域使用 4#2，表示某月的第二个星期三，4 表示星期三，2 表示第二个。

示例

- */5 * * * * ? 每隔 5 秒执行一次
- 0 */1 * * * ? 每隔 1 分钟执行一次
- 0 0 2 1 * ? * 每月 1 日的凌晨 2 点执行一次
- 0 15 10 ? * MON-FRI 周一到周五每天上午 10：15 执行作业
- 0 15 10 ? 6L 2002-2006 2002 年至 2006 年的每个月的最后一个星期五上午 10:15 执行作业
- 0 0 23 * * ? 每天 23 点执行一次
- 0 0 1 * * ? 每天凌晨 1 点执行一次
- 0 0 1 1 * ? 每月 1 日凌晨 1 点执行一次
- 0 0 23 L * ? 每月最后一天 23 点执行一次
- 0 0 1 ? * L 每周星期天凌晨 1 点执行一次
- 0 26,29,33 * * * ? 在 26 分、29 分、33 分执行一次

- 0 0 0,13,18,21 * * ? 每天的 0 点、13 点、18 点、21 点都执行一次
- 0 0 10,14,16 * * ? 每天上午 10 点，下午 2 点，4 点执行一次
- 0 0/30 9-17 * * ? 朝九晚五工作时间内每半小时执行一次
- 0 0 12 ? * WED 每个星期三中午 12 点执行一次
- 0 0 12 * * ? 每天中午 12 点触发
- 0 15 10 ? * * 每天上午 10:15 触发
- 0 15 10 * * ? 每天上午 10:15 触发
- 0 15 10 * * ? * 每天上午 10:15 触发
- 0 15 10 * * ? 2005 2005 年的每天上午 10:15 触发
- 0 * 14 * * ? 每天下午 2 点到 2:59 期间的每 1 分钟触发
- 0 0/5 14 * * ? 每天下午 2 点到 2:55 期间的每 5 分钟触发
- 0 0/5 14,18 * * ? 每天下午 2 点到 2:55 期间和下午 6 点到 6:55 期间的每 5 分钟触发
- 0 0-5 14 * * ? 每天下午 2 点到 2:05 期间的每 1 分钟触发
- 0 10,44 14 ? 3 WED 每年三月的星期三的下午 2:10 和 2:44 触发
- 0 15 10 ? * MON-FRI 周一至周五的上午 10:15 触发
- 0 15 10 15 * ? 每月 15 日上午 10:15 触发
- 0 15 10 L * ? 每月最后一日的上午 10:15 触发
- 0 15 10 ? * 6L 每月的最后一个星期五上午 10:15 触发
- 0 15 10 ? * 6L 2002-2005 2002 年至 2005 年的每月的最后一个星期五上午 10:15 触发
- 0 15 10 ? * 6#3 每月的第三个星期五上午 10:15 触发


蚂蚁金服
ANT FINANCIAL
金融科技
FINANCIAL TECHNOLOGY