蚂蚁金服金融科技产品手册

微服务

产品版本： v2.19.7

文档版本： V20191031

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

微服务平台（Microservices，简称 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，您可以在微服务平台控制台进行服务管理和治理的相关操作。

微服务平台通过 SOFA 微服务（简称 SOFA MS）和 Service Mesh 微服务（简称 Service Mesh MS），提供了既支持 SOFA 框架又支持 Service Mesh 架构的微服务管理和治理能力。

- **SOFA 微服务**：提供了 SOFA 框架的微服务，包含 RPC 服务、定时任务、动态配置、限流熔断。
- **Service Mesh 微服务**：通过 Service Mesh 技术支持原生 Dubbo 、Spring Cloud、SOFA 框架，无侵入地提供了对 Dubbo、Spring Cloud、SOFA 应用的服务管理和治理能力。

SOFA 微服务

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。

定时任务

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，可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。

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。

Service Mesh 微服务

服务管控

支持查看应用服务详情，包括其基本信息、服务提供者以及消费者信息等。同时提供了精细化调配的服务路由与服务限流功能，保证应用高可用，保障业务持续运行。

SideCar 管理

支持查看及管理当前工作空间中的 SideCar 实例，提供了 SideCar 状态、Pod 状态、SideCar 注入时间等信息。

服务拓扑

可视化展示了不同应用服务之间的调用关系和依赖关系，以及各节点的实时监控信息，包括请求量、响应时间及错误率等

。

实时监控

提供了所有应用以及各项性能指标的总体统计数据等。支持实时监控应用服务的吞吐量、响应时间、RPS、状态，及时发现应用服务异常。

1.2 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。

分布式任务调度框架

提供可靠的自动化任务管理调度功能，实现集群管理调度和分布式部署。

多种任务类型
支持单任务，并发任务，顺序任务，并支持定时执行及手动执行。

多种任务触发方式

支持定时任务在客户端集群中随机触发、定向触发、及按指定 IP 触发，并支持多种失败重试策略。

图形化的集中式管理界面
简单易用的管理界面，并支持查看定时任务的执行记录，包括触发时间，执行是否成功，执行的客户端IP等。

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。

1.3 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。

2 SOFA MS

2.1 快速入门

微服务平台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，您可以通过微服务控制台完成对各组件功能的配置，包括：RPC 服务、定时任务调度服务、动态配置、限流熔断等。

前置条件

在开始使用中间件微服务之前，您必须完成以下准备：

- 配置好一个 SOFABoot 工程，参考 SOFABoot 快速开始 。

  说明：请使用 SOFABoot 2.3.0 及以上版本。最新版本信息参见 SOFABoot 发布说明 。

- 完成安全配置。为保障中间件的安全性，所有的调用均需要验证访问者的身份，详细信息参见 引入中间件 。
- 在 SOFABoot 工程的 pom.xml 文件中引入相应的 Maven 依赖，参见 Maven 依赖 。

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。具体操作参见：

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

Maven 依赖

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。

以下 Maven 依赖的版本管控均由 SOFABoot 统一实现，无须单独指定。

SOFARPC

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

定时任务

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

参见 开始使用限流熔断 。

2.2 开始使用 SOFARPC

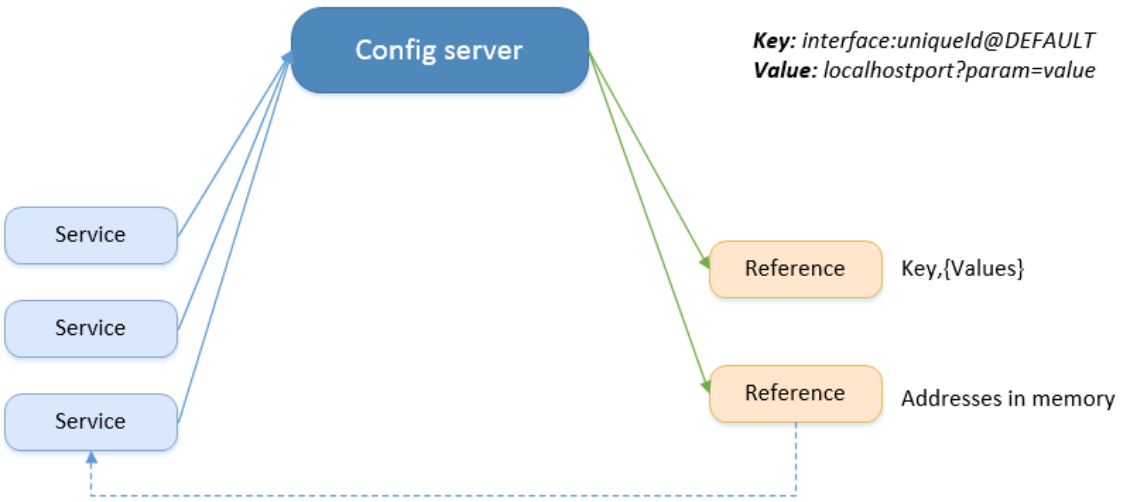
SOFARPC 提供应用之间的点对点服务调用功能，具有高可伸缩、高容错的特性。

- 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。

开始使用 SOFARPC

使用 SOFARPC 包括以下步骤：

1. 确认已完成系统环境配置，详情见 前置条件 。
2. 在本地 SOFABoot 工程中引入 SOFARPC 的 Maven 依赖。

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

3. 完成 RPC 服务的开发与配置，详见 服务发布与引用 。
4. 部署应用 。
5. 在微服务控制台中查询与管理已注册的 RPC 服务，详情见 服务查询 。

服务发布与引用

> **重要**：点击此处 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。

以下假设新建 SOFABoot 项目时，设置的 artifactId 为 appName，可以根据自己的实际情况新增阅读和设置。

SOFARPC 的服务发布与引用配置均需要写到 Spring 的 XML 文件中。Spring XML 文件的默认路径如下：

```
src/main/resources/META-INF/appName/xxx-xxx.xml
```

服务发布

服务发布包含四个步骤：

设计服务接口类服务接口类 SampleService.java 的 Java 代码如下：

```
/**
* 服务接口类
*/
public interface SampleService {

public String hello();

}
```

编写服务实现类服务实现类 SampleServiceImpl.java 的 Java 代码如下：

```
/**
* 实现服务接口：SampleService
*/
public class SampleServiceImpl implements SampleService{

@Override
public String hello() {
return"hello world";
}
}
```

在 Spring XML 中配置服务发布在 Spring XML 文件 rpc_server.xml 中配置服务发布的相关参数（也可以直接在已有的 XML 中新增以下配置，使用 SOFABoot 原型工程时，可以在 appName-endpoint.xml 进行新增）。

```
<?xml version="1.0"encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sofa="http://schema.alipay.com/sofa/schema/slite"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://schema.alipay.com/sofa/schema/slite http://schema.alipay.com/sofa/slite.xsd">

<!-- 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"/>

<!-- 发布 RPC 服务 -->
<sofa:service ref="sampleService"interface="com.alipay.samples.rpc.SampleService">
<sofa:binding.bolt/>
</sofa:service>
</beans>
```

说明：配置中的 <sofa:binding.bolt/> 在 SOFABoot 2.2.3 之前的版本中，默认为 <sofa:binding.tr/>，请根据自己的项目情况进行配置。

4. 启动服务直接运行项目中的 SOFABootWebSpringApplication 即可，框架会自动进行服务的发布。

服务引用

1. 在 Spring XML 中配置服务引用在 Spring XML 文件 rpc_client.xml 中配置服务引用的相关参数（也可以在原型工程中已有的 XML 中增加以下配置）。这里将以下内容放在 endpoint 这个子模块的 src/main/resources/META-INF/appName/rpc_client.xml 中：

```
<?xml version="1.0"encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sofa="http://schema.alipay.com/sofa/schema/slite"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://schema.alipay.com/sofa/schema/slite http://schema.alipay.com/sofa/slite.xsd">
<!-- 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>
<!-- 等待配置中心返回地址的时间，如果在 5000 ms 内有地址返回的话，等待过程会立马结束 -->
<sofa:global-attrs address-wait-time="5000"test-url="127.0.0.1:12200"/>
</sofa:binding.bolt>
</sofa:reference>
</beans>
```

说明：配置中的 <sofa:binding.bolt/> 在 SOFABoot 2.2.3 之前的版本中，默认为 <sofa:binding.tr/>，请根据自己的项目情况进行配置。

将服务引用对象注入业务代码为了方便直观使用，我们将引用服务，放在 RPC 服务引用类 SOFABootWebSpringApplication.java 的 Java 代码如下：

```
@ImportResource({"classpath*:META-INF/demo/*.xml"})
@org.springframework.boot.autoconfigure.SpringBootApplication
public class SOFABootWebSpringApplication {

// init the logger
private static final Logger logger = LoggerFactory.getLogger(SOFABootWebSpringApplication.class);

public static void main(String[] args) {
//下面这一行只有在本地同时启动客户端和服务端的时候需要，正式环境不可以写
System.setProperty("server.port","8081");
SpringApplication springApplication = new SpringApplication(SOFABootWebSpringApplication.class);
ApplicationContext applicationContext = springApplication.run(args);

if (logger.isInfoEnabled()) {
logger.info("application start");
}

SampleService sampleService = (SampleService) applicationContext.getBean("sampleServiceRef");

String resp = sampleService.hello();
if (logger.isInfoEnabled()) {
logger.info("the resp data is" + resp);
}

}
}
```

正常启动客户端即可调用服务，可在应用日志中看到日志，一般在 common-default.log 中。在测试中，为了方便，也可以将以上日志打印写成输出到控制台。

注意：

- 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，在 application.properties 修改以下配置即可。

```
rpc.tr.port=12201
```

rpc_tr_port 是 TR 端口号，详见 SOFARPC 进阶指南 > 配置说明 。

- 如果本机由于网络问题，无法连接本地服务注册中心。可以将 run.mode 设置为 TEST ，进行直连调用。

```
run.mode=TEST
```

run.mode 是 RPC 运行模式，详见 系统配置参数 。

部署应用

将应用发布在 SOFAStack 上，发布流程参见 发布部署应用 。

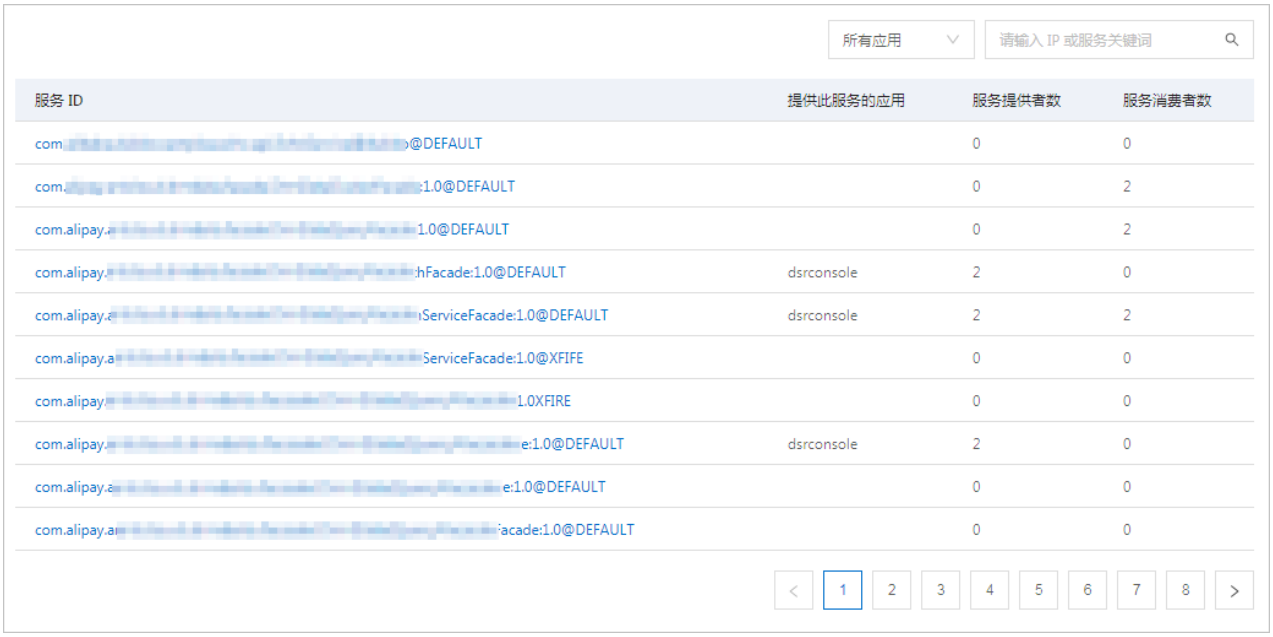
服务查询

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。

登录微服务控制台后，点击 **SOFA MS** > **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 。

2.3 开始使用 SOFAREST 服务

说明：本功能适用于 SOFABoot 2.4.0 及以上版本。

对于 SOFABoot 2.4.0 及以上版本，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 SOFARPC 进阶指南 > 配置说明 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%
```

2.4 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 SOFARPC 进阶指南 > 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/> 。

说明：在 SOFABoot 2.2.3 之前的版本中，配置中的 <sofa:binding.bolt/> 默认为 <sofa:binding.tr/>，请根据项目情况进行配置。

- 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 SOFARPC 进阶指南 > 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>
```

2.5 使用原生 API

原生 API 是 SOFARPC 底层核心的使用方式。大多数情况下，并不需要直接操作原生 API，在 SOFABoot 环境下，您可以参考 SOFARPC 进阶指南 直接使用 SOFARPC。但在非 SOFABoot 环境下，可能需要暂时使用原生 API 进行服务发布与引用。

定义接口

```
public interface SampleService {

String message();
}
```

引用依赖

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-rpc-enterprise-all</artifactId>
<version>${rpc.enterprise.version}</version>
</dependency>
```

具体版本信息，请使用技术支持团队推荐的 SOFABoot 版本中携带的 sofa-rpc-enterprise-all 版本号。

初始化配置

```
System.setProperty("com.alipay.env","shared");
System.setProperty("com.alipay.instanceid","xxx");
System.setProperty("com.antcloud.antvip.endpoint","xxx");
System.setProperty("com.antcloud.mw.access","xxx");
System.setProperty("com.antcloud.mw.secret","xxx");
```

其中，关于 com.alipay.instanceid，com.antcloud.antvip.endpoint， com.antcloud.mw.access 和 com.antcloud.mw.secret 参数的具体含义及对应值，参见 中间件全局配置项 。

设置当前应用信息

```
ApplicationConfig appConfiguration = new ApplicationConfig();
appConfiguration.setAppName("sofa2-rpc-client");
```

设置服务注册中心信息

```
RegistryConfig registryConfig = new RegistryConfig();
registryConfig.setProtocol("dsr");
registryConfig.setSubscribe(true);
registryConfig.setConnectTimeout(3000);
```

引用服务

```
ConsumerConfig<SampleService> consumerConfig = new ConsumerConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setProtocol(protocol)
.setApplication(appConfiguration);

consumerConfig.setRegistry(registryConfig)
.setConnectTimeout(3000)
.setTimeout(3000);

SampleService refer = consumerConfig.refer();

//进行调用
```

发布服务

```
ProviderConfig providerConfig = new ProviderConfig<SampleService>()
.setInterfaceId(SampleService.class.getName())
.setRef(new HelloServiceTimeOutImpl())
.setServer(serverConfig)
.setRegister(true)
.setApplication(providerAconfig)
.setRegistry(registryConfig);

providerConfig.export();
```

2.6 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：

- **对于 SOFABoot 2.3.0 及以上版本**

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

- **对于 SOFABoot 2.0 - 2.3.0 之间的版本**

```
<dependency>
<groupId>com.alipay.boot</groupId>
<artifactId>sofarpc-spring-boot-starter</artifactId>
</dependency>
```

- **对于 SOFABoot 1.x 的版本**

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-runtime-api</artifactId>
</dependency>
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-rpc-api</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。

TR 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);
```

TR 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。

2.7 使用服务路由与服务注册中心

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。

2.8 SOFARPC 进阶指南

本指南主要介绍以下内容：

- 发布 SOFARPC 服务
- 引用 SOFARPC 服务
- 调用上下文
- 泛化调用
- 预热转发
- 自定义 Filter
- 配置说明
- 日志说明
- 性能测试

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。

重要：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>
```

说明：在 SOFABoot 2.2.3 之前的版本中，配置中的 <sofa:binding.bolt/> 默认为 <sofa:binding.tr/>，请根据项目情况进行配置。

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/sofa-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 。

引用 SOFARPC 服务

通过本教程，您将快速学习到如何引用一个 RPC 服务。

> **重要**：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。

说明：在 SOFABoot 2.2.3 之前的版本中，配置中的 <sofa:binding.bolt/> 默认为 <sofa:binding.tr/>，请根据项目情况进行配置。

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/sofa-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 。

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

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"
```

预热转发

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。

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 ... />
```

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

> **说明：**对于 SOFABoot 2.4.0 以上的版本，RPC 框架线程池的配置项发生变更。标注 * 的配置项仅适用于 2.4.0 及以上的版本。

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：

对于 SOFABoot 2.4.0 及以上版本：

- 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

对于 SOFABoot 2.3.4 及更早的版本：

- rpc_min_pool_size_tr：最小线程数，默认 20
- rpc_max_pool_size_tr：最大线程数，默认 200
- rpc_pool_queue_size_tr：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

参见 性能埋点日志 。

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（SOFALite 1.0 以及 SOFABoot 均支持）
- 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 RPC Tracer 日志格式 。

性能测试

本文提供基于 SOFABoot 2.3.0 的 RPC 性能测试数据供参考。测试包括 BOLT 和 REST 协议的两种性能数据。

压测环境

服务端机器配置

- 内存：4G
- CPU：4-Core（Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz）
- 磁盘：普通机械硬盘100G（非 SSD）

压测方案

为了测试 RPC 框架的性能数据，服务端提供一个用于增删改查的接口，客户端进行真实调用。压测工具使用标准 JMH（Java Microbenchmark Harness）进行多轮测试。

```
// 服务端接口
public interface RpcUserService {

public boolean existUser(String email);

public boolean createUser(RpcUser user);

public RpcUser getUser(long id);

public Page<RpcUser> listUser(int pageNo);

}
```

测试结果

在服务端 CPU 达到 60-70%、内存占用 50% 的情况下，测试结果如下：

BOLT

> 在没有打开 Tracer 的情况下，不会统计每笔调用信息。此时，服务端平均达到 50000 tps，成功率 100%，其中 99% 的请求在 1.5 ms 内响应完成。
>
> 在已经打开 Tracer 的情况下，统计每笔调用信息并打印在磁盘中。此时，服务端平均达到 30000 tps，成功率 100%，其中 99% 的请求在 2.4 ms 内响应完成。

REST

在没有打开 Tracer 的情况下，不会统计每笔调用信息。此时，服务端平均达到 15000 tps，成功率 100%，其中 99% 的请求在 4 ms 内响应完成。

在已经打开 Tracer 的情况下，统计每笔调用信息并打印在磁盘中。此时，服务端平均达到 9000 tps，成功率 100%，其中 99% 的请求在 7 ms 内响应完成。

使用建议

基于以上测试结果，在选择通信协议时您可以考虑以下方案：

- 应用内部的调用使用 BOLT 通信协议。
- 当需要对外提供 REST/HTTP 服务时，可以将 REST 协议作为入口，并在内部转发给 BOLT 进行处理，以便进行更高效的通信。

建议您根据自己的实际场景来选择，或者进行相关的性能测试。

2.9 服务管控

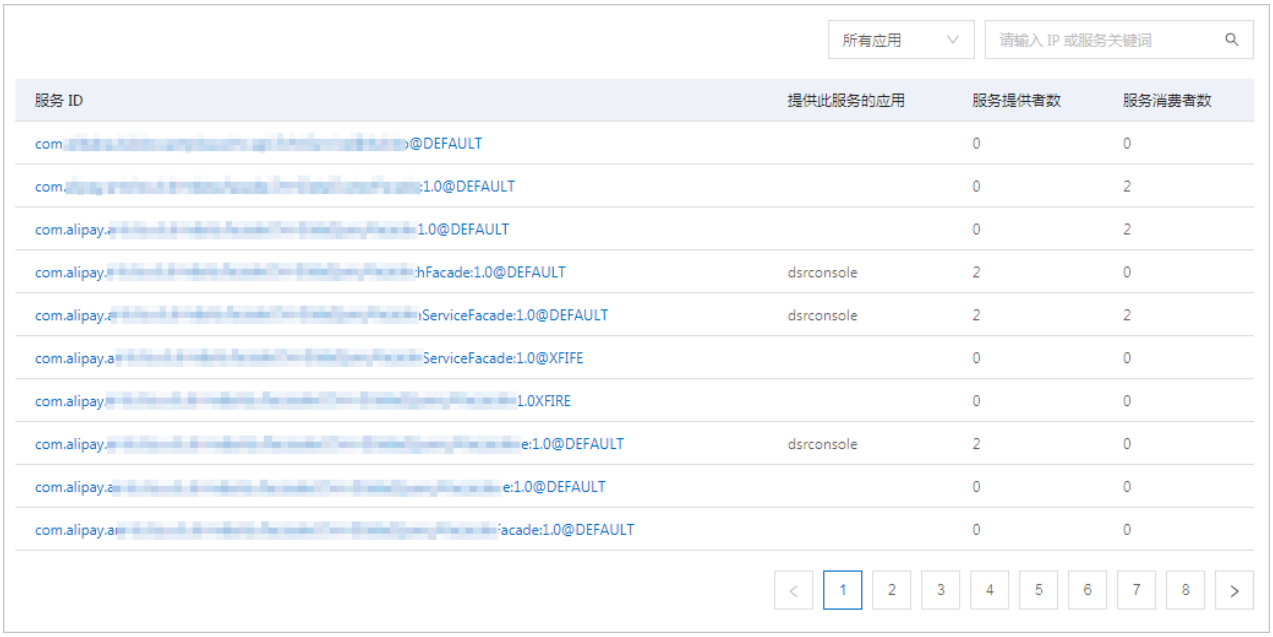
查看及管理 RPC 服务

您可以在 **微服务 > SOFA MS > 服务管控** 页面查看并管理当前工作空间下的所有可用 RPC 服务。

查看服务列表

在页面右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

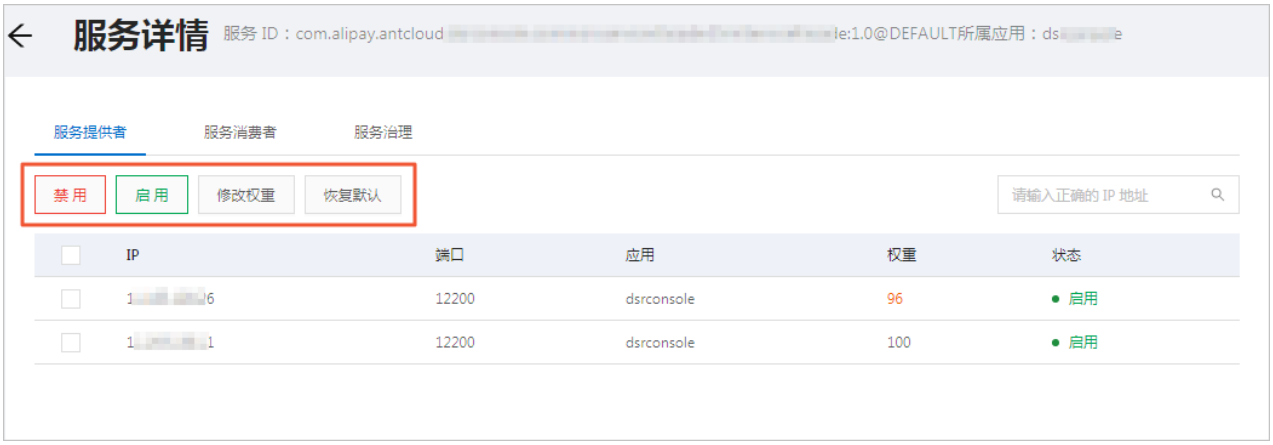
在服务查询列表中点击服务 ID，可以查看该服务的服务提供者和服务消费者列表。您还可以自由切换查看服务提供者列表与服务消费者列表。

- 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



查看应用依赖关系

您可以在 **微服务 > SOFA 微服务 > 应用依赖** 页面查看当前工作空间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。

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。

> **说明**：如需使用该功能，请先 提交工单 咨询技术支持团队。

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**

创建访问控制规则

1. 在微服务控制台页面，选择 **SOFA 微服务 > 服务管控**。
2. 在页面右上方的搜索栏中，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**。

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**。

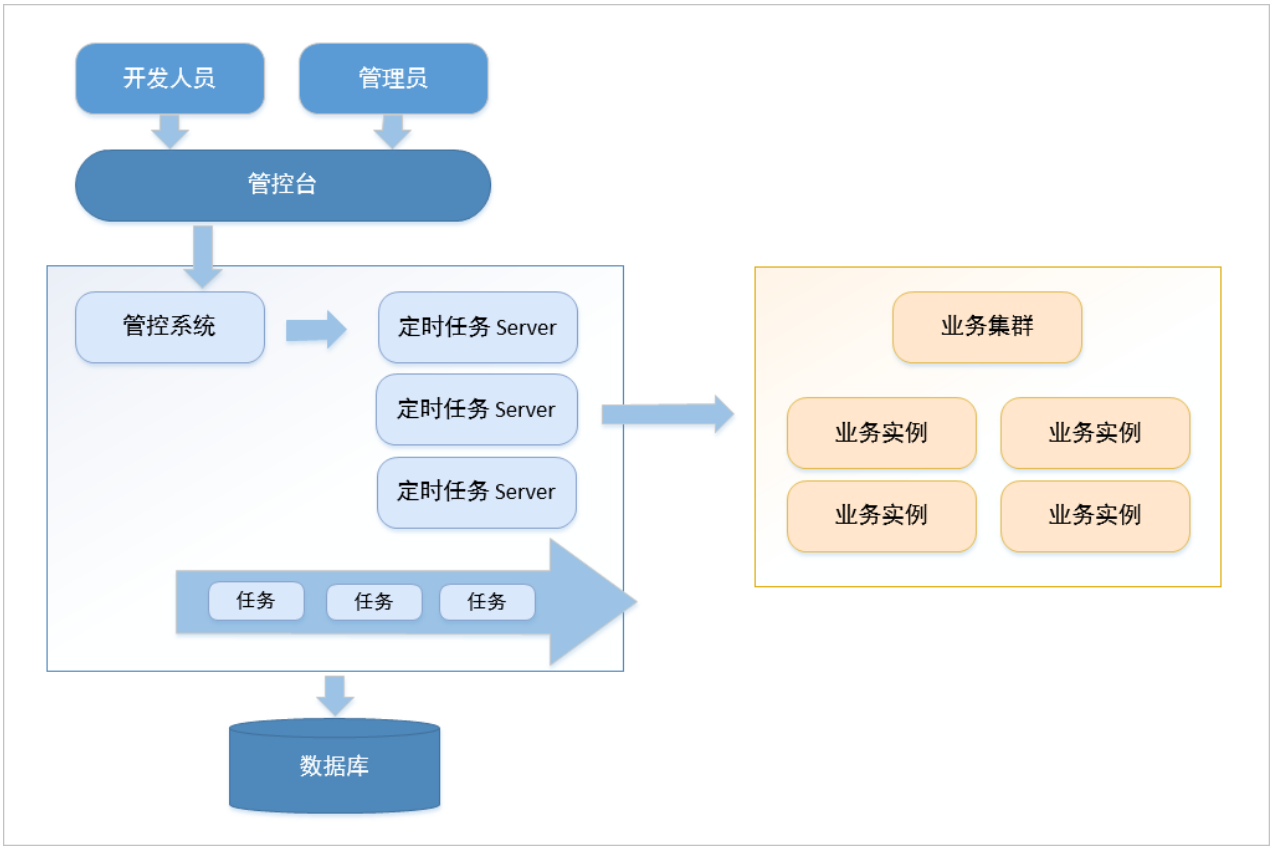
2.10 开始使用定时任务

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。

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

定时任务服务端按照用户配置的定时任务信息，在到达任务执行时间时向应用发起 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。



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。

开始使用定时任务

开始使用定时任务前，请确认您已经完成系统环境配置，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

完整的定时任务使用教程请参见 接入并配置定时任务 。

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 SOFARPC 进阶指南 > 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 快速开始 。

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。

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。

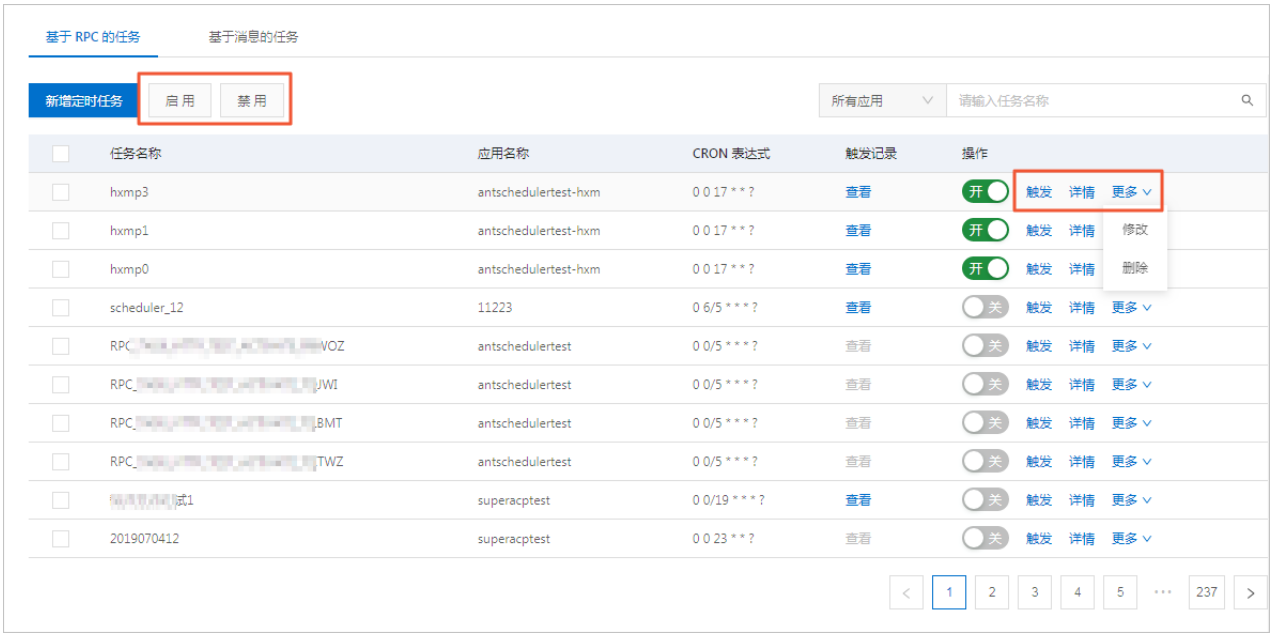
新增完成之后，定时任务就按照预期的频率开始定时执行了。

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。



注意事项

使用定时任务服务时，需要注意以下事项：

- 单台执行：正常情况下，一次任务触发只会有一台机器执行，不会全集群同时执行；
- 秒级触发：CRON 表达式只能精确到秒级，无法完成毫秒级的触发；
- 幂等性控制：使用消息完成交互，要考虑异常场景下消息重投的可能性，做好幂等控制；

2.11 基于消息的定时任务

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。

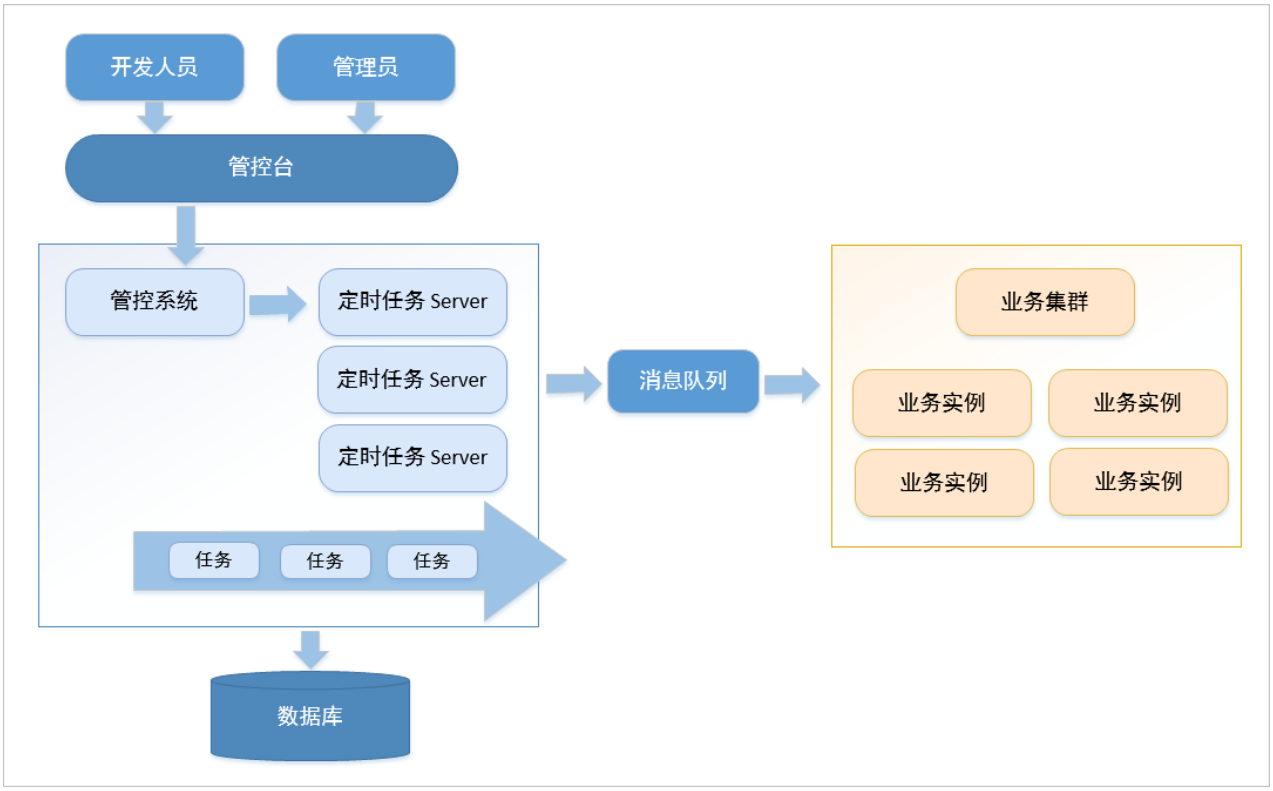
定时任务服务旨在为业务系统提供统一通用的任务调度服务，提供定时任务的管理监控平台，减轻业务系统开发和后续线上运维的工作量，并通过任务拆分和负载均衡等方案提升大数据量任务的性能。

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。

开始使用基于消息的定时任务

> 基于消息的定时任务依赖消息队列（Message Queue）。在使用基于消息的定时任务之前，您必须先开通消息队列服务。

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
3. 配置定时任务
4. 配置消息类型及订阅关系

前置条件

- 您已经准备好一个 SOFABoot 工程，推荐升级到最新版本。
- 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 。

开发编码

定时任务服务依赖消息队列实现定时任务功能。本地编码主要是完成订阅端代码（监听消息完成任务逻辑）。具体步骤介绍参见 消息队列 > 快速开始 。

消息订阅端代码

```
public class SchedulerDemo implements UniformEventMessageListener {

/** logger */
private static final Logger LOGGER = LoggerFactory.getLogger(SchedulerDemo.class);

@Override
public void onUniformEvent(UniformEvent uniformEvent, UniformEventContext uniformEventContext)
throws Exception {
final String topic = uniformEvent.getTopic();
final String eventCode = uniformEvent.getEventCode();

// 接收触发后的定时业务处理
LOGGER.info("[Receive an uniformEvent] topic {} eventcode {} eventId {} payload {}",
new Object[] { topic, eventCode, uniformEvent.getId()});
}
}
```

消息订阅端配置

一条典型的消息队列消息包括 TOPIC 和 EVENTCODE。定时任务发送的消息 TOPIC 统一为 TP_F_SC，开发时需要确定一个 EventCode（以 EC_ 开头），以及消息订阅组 Group（格式为：S_appname_service）。

```
<!-- consumer declaration, the id and group attribute are required and their value must be unique -->
<sofa:consumer id="uniformEventSubscriber"group="S_schedulertutorial_demo">
<sofa:listener ref="schedulerDemo"/>
<sofa:channels>
<!-- channel value is the involved topic -->
<sofa:channel value="TP_F_SC">
<!-- each event represents a subscription -->
<sofa:event eventType="direct"eventCode="EC_TASK_SCHEDULERTUTORIAL_DEMO"persistence="false"/>
</sofa:channel>
</sofa:channels>
<sofa:binding.msg_broker/>
</sofa:consumer>

<!-- messageListener listener bean declaration, implements
com.alipay.common.event.UniformEventMessageListener -->
<bean id="schedulerDemo"class="com.antcloud.tutorial.scheduler.SchedulerDemo"/>
```

消息超时

消息队列的消息超过 8 秒会重试，会存在部分定时任务执行时间超过 8 秒的情况。对于这种情况，需要使用线程池，启用异步线程执行任务处理，然后直接 return，告诉消息队列已经完成消息处理，避免超时重试。

示例如下:

```
<!-- 异步任务使用的线程池 -->
<bean id="threadPool"class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
<property name="corePoolSize"value="10"/>
<property name="maxPoolSize"value="120"/>
```

```
<property name="queueCapacity"value="200"/>
</bean>
<!-- 具体参数请根据业务特征自行配置 -->
//然后在您的消息监听器中启用新线程去处理任务:
threadPool.execute(new Runnable() {
public void run() {
//执行任务逻辑
}
});
```

发布应用

将应用发布至云端，SOFABoot 应用的发布参见 SOFABoot 快速开始 。

配置定时任务

配置入口

登录微服务控制台后，选择 **SOFA MS** > **定时任务** > **基于消息的任务**，即可开始使用定时任务服务。

配置任务

在定时任务界面中，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**。

- **任务名称**：命名格式为 EC_TASK_SYSTEM_FUNCTION，注意需要和实际代码中的配置保持一致。
- **系统名**：填写发布的应用名。
- **CRON 表达式**：调度中心使用 Quartz 来实现定时执行，配置规则可参见 Quartz 官网。

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，所以建议开发联调初期，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。

新增完成后，还可以对任务进行其他的操作：

- **开/关**：界面显示为 **开** 时，任务会自动执行，点击后状态变更为 **关**，任务停止自动执行。

- **触发**：手工在界面进行一次触发，任务就会在后台触发一次。
- **编辑**：调整任务名称、CRON、系统。
- **删除**：删除某个定时任务。

另外，由于定时任务的开/关、修改，涉及到在后台内存中的实际生效，每次生效执行完成需要耗时 30 秒 ~ 60 秒，在每次这类操作生效之前，下次这类操作无法开始执行。

application.properties

每个工作空间需要连接的服务端不一样，参见 SOFARPC 进阶指南 > RPC 引用服务 完成对应工作空间的 application.properties 参数调整：

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

配置消息类型及订阅关系

在云端控制台配置消息主题、消费组及订阅关系，参见 消息队列 > 快速开始 。

> **注意**：
>
> - 消息主题：对于来自定时任务的消息，统一填写：TP_F_SC。
> - 消息分类码：对应定时任务中配置的任务名称，需要和实际代码中的配置一致。
> - 消息消费组：需要和实际代码中的配置一致。

配置消息类型、订阅关系后记得将其生效。到了设定的任务执行时间后，任务就会被执行。

注意事项

使用定时任务服务时，需要注意以下事项：

- 组件依赖 MQ（Message Queue，消息队列）：需要先开通消息队列；
- 单台执行：正常情况下，一次任务触发只会有一台机器执行，不会全集群同时执行；
- 秒级触发：CRON 表达式只能精确到秒级，无法完成毫秒级的触发；
- 幂等性控制：使用消息完成交互，要考虑异常场景下消息重投的可能性，做好幂等控制；
- 超时：消息会超时重试 8 s，如果业务处理耗时很长，建议在 UniformEventMessageListener 中接受消息后直接返回成功，使用异步线程池，异步完成实际业务处理。

2.12 Cron 表达式说明

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

2.13 开始使用动态配置

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，用户只需要在每个工作空间开通中间件微服务，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。

动态配置快速入门

> **说明**：请使用 SOFABoot 2.3.0 及以上版本。

开始使用前，请确认您已经完成系统环境配置，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 SOFA 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。

有关注解使用方法，参见 使用注解标识配置类 。

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。

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. 在微服务控制台页面，选择 **SOFA MS >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。

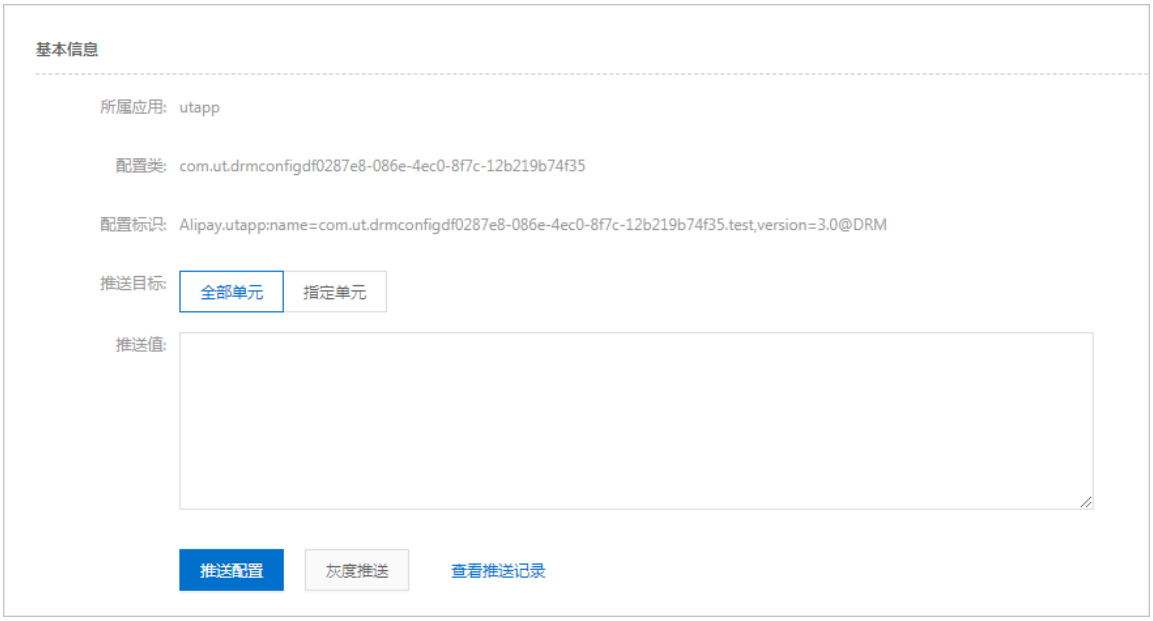
推送动态配置

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。

操作步骤如下：

1. 前往 **微服务** > **SOFA MS** > **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. 输入推送值。



4. 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**。

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。

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，参见上文 动态配置快速入门 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。

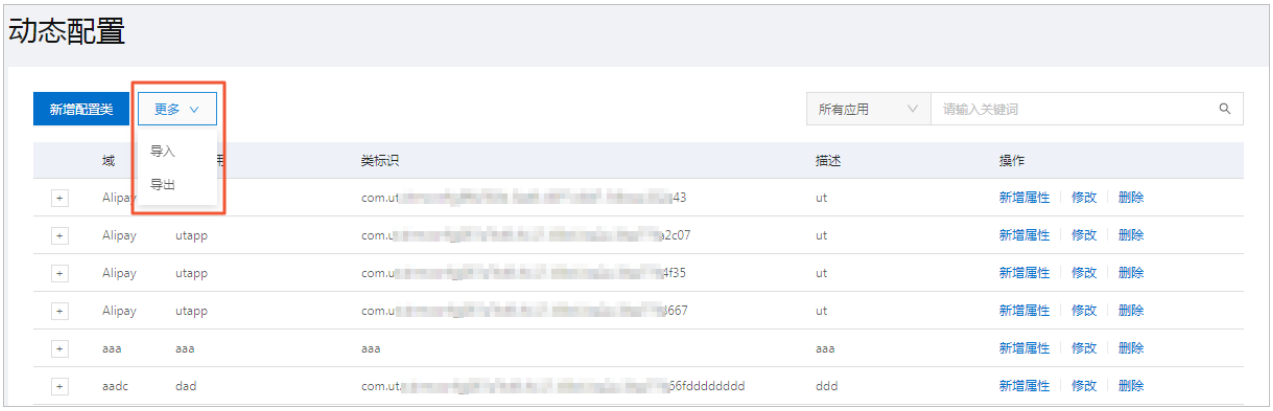
属性依赖等级有以下几种：

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。

导入导出动态配置元数据

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。

导入数据示例如下：

```
{"appName":"drmtestProject","attributes":[{"attributeName":"age","name":"年龄
"},{"attributeName":"name","name":"名称"},{"attributeName":"test","name":"dd"}],"name":"测试资源类
","region":"Alipay","resourceDomain":"Alipay.drmtestProject","resourceId":"com.alipay.share.drm.NormalDrmResour
ceTest"}
{"appName":"test","attributes":[{"attributeName":"ddd","name":"xxx"}],"name":"描述
","region":"domain","resourceDomain":"domain.test","resourceId":"classa"}
```

配置类 JSON 数据结构解析：

```
{
"region":"domain",
"appName":"test",
"resourceId":"classa",
"name":"描述",
"attributes": [
{
"attributeName":"ddd",
"name":"xxx"
}
]
}
```

2.14 开始使用限流熔断

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。

- 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。
- 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。

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。

限流熔断快速入门

开始使用前，请确认您已经完成系统环境配置，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
    - SOFABoot 2.4.0 及以上版本
    - SOFABoot 2.3.4 或更早的版本
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

SOFABoot 2.4.0 及以上版本

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>
```

说明：guardian-sofa-boot-starter 仅适用于 SOFABoot 2.4.0 及以上版本。最新版本信息请参见 SOFABoot 发布说明 。

SOFABoot 2.3.4 或更早的版本

1. 在工程主 pom.xml 的 dependencyManagement 中引入限流熔断依赖：

```
<dependency>
<groupId>com.alipay.guardian</groupId>
<artifactId>guardian-core</artifactId>
```

```
<version>1.3.8</version>
<exclusions>
<exclusion>
<groupId>org.apache</groupId>
<artifactId>common-logging</artifactId>
</exclusion>
<exclusion>
<groupId>com.alipay.sofa.common.log</groupId>
<artifactId>sofa-middleware-log</artifactId>
</exclusion>
</exclusions>
</dependency>
<dependency>
<groupId>com.alipay.guardian</groupId>
<artifactId>guardian-sofalite</artifactId>
<version>1.3.8</version>
</dependency>
<dependency>
<groupId>org.mvel</groupId>
<artifactId>mvel2</artifactId>
<version>2.3.2.Final</version>
</dependency>
<dependency>
<groupId>org.codehaus.groovy</groupId>
<artifactId>org.codehaus.groovy.all</artifactId>
<version>1.8.9</version>
</dependency>
<dependency>
<groupId>apache-collections</groupId>
<artifactId>commons-collections</artifactId>
<version>3.1</version>
</dependency>
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>guardian-sofa-boot-starter</artifactId>
</dependency>
```

2. 在 core-service 模块的 pom.xml 中引入以下内容：

```
<dependency>
<groupId>com.alipay.guardian</groupId>
<artifactId>guardian-core</artifactId>
</dependency>
<dependency>
<groupId>com.alipay.guardian</groupId>
<artifactId>guardian-sofalite</artifactId>
</dependency>
<dependency>
<groupId>org.mvel</groupId>
<artifactId>mvel2</artifactId>
</dependency>
<dependency>
<groupId>com.alipay.boot</groupId>
<artifactId>ddcs-spring-boot-starter</artifactId>
```

```
</dependency>
<dependency>
<groupId>org.codehaus.groovy</groupId>
<artifactId>org.codehaus.groovy.all</artifactId>
</dependency>
<dependency>
<groupId>apache-collections</groupId>
<artifactId>commons-collections</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 SOFAStack 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 快速开始 > 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 > SOFA MS > 限流熔断** 进行限流任务的创建与配置。操作步骤如下：

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
        - 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。
    - 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。

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。操作步骤如下：

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。

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。

查看限流日志

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
```

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)

2.15 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

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：

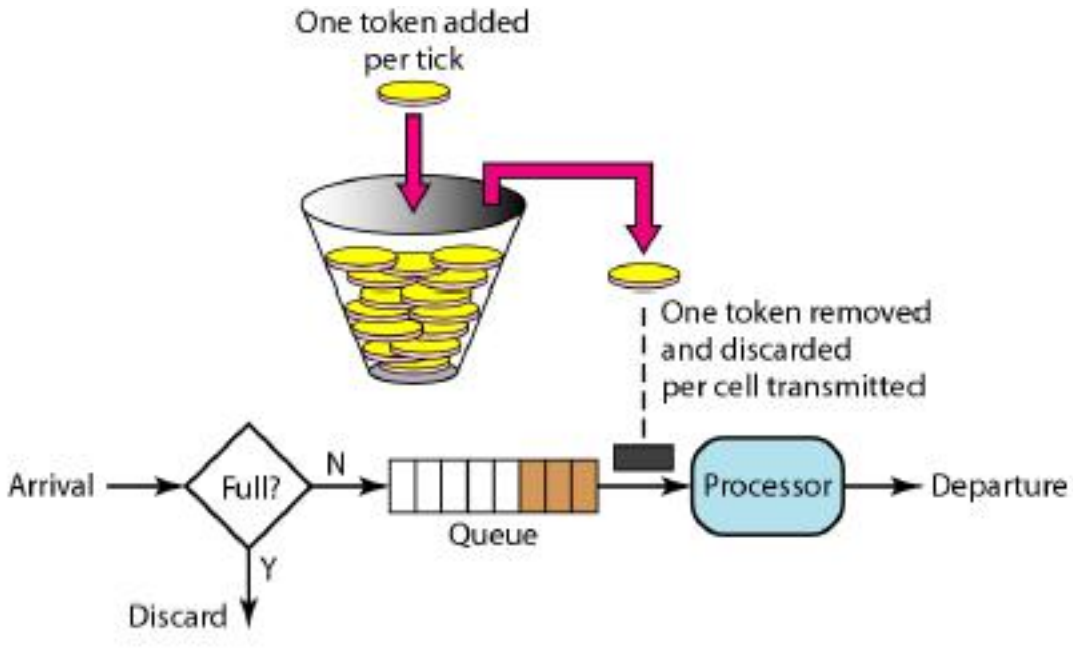
- 计算简单，是否限流只跟请求数相关，放过的请求数是可预知的（令牌桶算法放过的请求数还依赖于流量是否均匀）。比较符合用户直觉和预期。
- 可以通过拉长限流周期来应对突发流量。如 1 秒限流 10 个，想要放过瞬间 20 个请求，可以把限流配置改成 3 秒限流 30 个。拉长限流周期会有一定风险，用户可以自主决定承担多少风险。

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。

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

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。

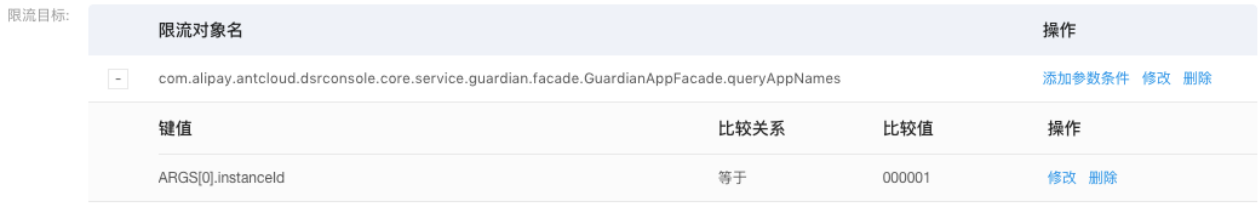
- 对于接口方法，配置要限流的接口路径名称和方法签名。
- 对于 Web 页面方法，配置要限流的请求 URI。

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。
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：

限流对象名			操作
/webapi/guardian/history/search			添加参数条件 修改 删除
键值	**比较关系**	**比较值**	**操作**
instanceId	等于	000001	修改 删除

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：

键值	比较关系	比较值	操作
ARGS[0].instanceId == '000001'	等于	true	修改 删除

- 方式二：左侧键值计算结果是个普通字符串，右侧比较值中也填写一个字符串，例如：

键值	比较关系	比较值	操作
ARGS[0].instanceId	等于	000001	修改 删除

上述两种方式的效果一样，推荐用第一种方式，第一种方式支持更多的运算符，例如：&&、||、>、<= 等，表达能力更丰富。

配置样例

使用 MVEL 表达式获取参数的属性值：

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'。

2.16 故障排查

SOFARPC

错误 RPC-02306：无法获取 RPC 服务地址问题

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 **脚手架** 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。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。默认值为 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
```

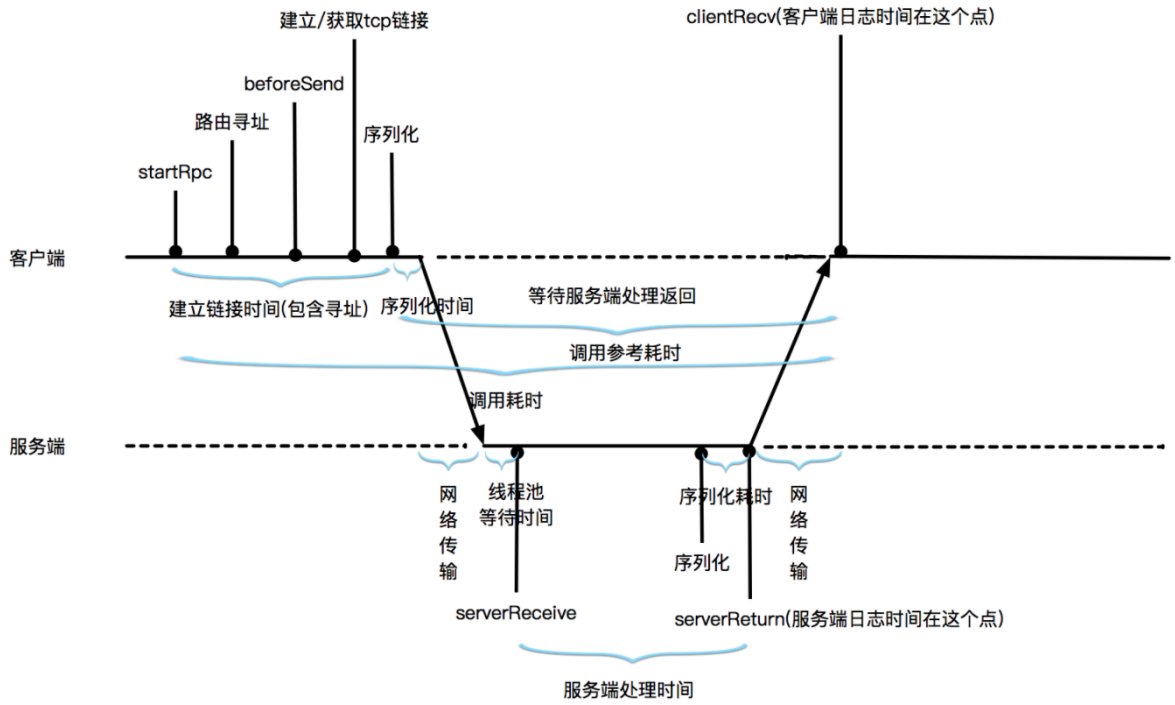
查看进程启动参数rpc_bind_network_interface 或 rpc_enabled_ip_range 是否绑定了正确的IP地址。

如何排查 RPC 超时问题

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,707c27b9153085447746110464663,0,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 SOFARPC 进阶指南 > 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=&{工作空间标识} -jar /home/admin/app-run/sofa2-rpcserver-service-1.0-SNAPSHOT-executable.jar &
```

说明：**工作空间标识** 可在控制台中选择 **金融分布式架构** > **资源管理** > **工作空间** 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
```

定时任务

日常研发时无法收到任务触发

建议按以下步骤依次排查问题：

1. 在界面上手动触发任务，确认是否提示 **成功**。
2. 查看调度界面上的 EventCode 是否与代码中的一致。
3. 查看消息队列界面上消息类型、订阅关系是否完成配置，状态是否为 **生效**。
4. 查看代码中的 GroupId/Topic/EventCode 是否与管控界面上的配置一致。
5. 查看服务器 netstat –an|grep 9529 是否连上消息中心。
6. 确认消息客户端日志 logs/tracelog/msg-sub-digest.log 是否有接收对应的 EventCode。
7. 如果 msg-sub-digest 有日志，业务日志未打印，则可能是业务处理出现异常，应从 UniformEventMessageListener 进行排查，查看是否有其他报错。

消息重复投递

问题描述：我们在界面上已经停止了调度任务，但是依然收到了触发消息。

问题分析：定时任务依赖消息队列来完成调度任务投递，消息队列为了确保高可靠，对于业务处理异常或者业务处理耗时太长，都会认为是客户端处理失败，会发起重试，一般包括以下场景：

- 消息接收端处理失败，直接抛出异常；
- 消息接收端处理耗时太长，超过 8s，服务端会认定客户端处理失败，发起重试；
- 调度界面任务触发频率很高，消息量太大，无法接收新的消息（所以建议开发工作空间下将定时任务配置为暂停状态，开发完成后，评估单机实际处理能力再设置一个可靠的 CRON）。

解决方案：查看 logs/msgbroker/common-default.log 或者 common-error.log 是否有异常报错，如果有报错，异常堆栈一般从业务代码中抛出，请结合业务代码确认异常原因进行修复。

3 Service Mesh MS

3.1 快速入门

本文将引导您快速体验 Service Mesh 微服务的微服务管理和治理能力。

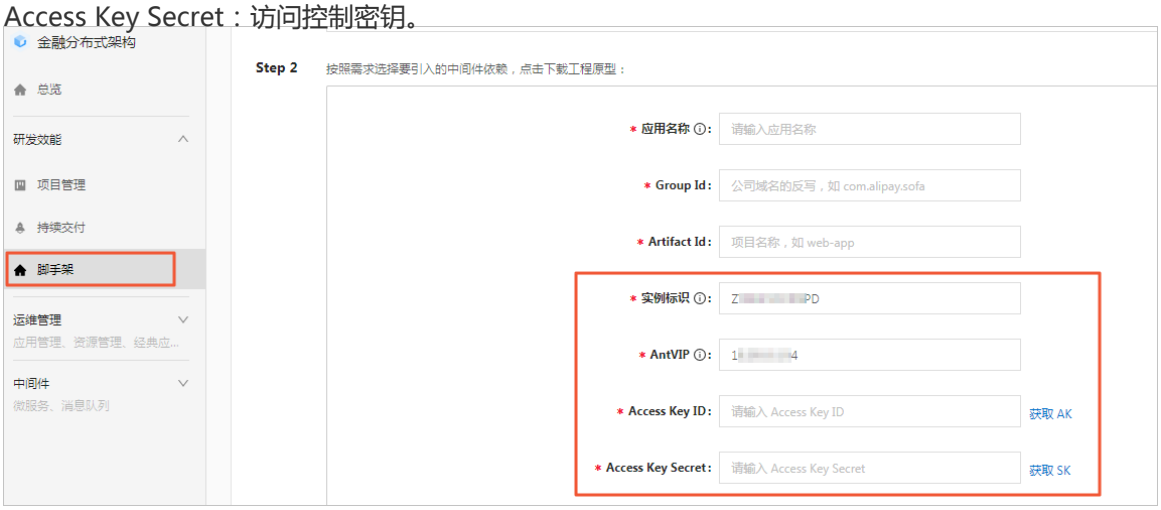
1. 获取中间件全局配置项
2. 准备镜像

3. 发布应用服务
4. 服务监控及治理

获取中间件全局配置项

使用 Service Mesh 微服务，您需要先获取中间件的全局配置信息，可以从 **脚手架 > Step 2** 获取。更多详情，参见 引入 SOFA 中间件 > 中间件全局配置 及 创建和管理访问控制密钥 。

- InstanceId：中间件实例唯一标识；
- AntVIP：中间件基于 AntVIP 寻址；
- Access Key ID：访问控制键值；
- Access Key Secret：访问控制密钥。



准备镜像

Service Mesh 微服务使用镜像方式进行发布。因此，您需要先将您的应用（目前支持 SOFA、Dubbo 和 SpringCloud）打包成镜像，而后进行发布。

此处，您可以直接使用提供的示例镜像，也可以自行改造已有的工程项目：

- 使用示例镜像（推荐）：已提前准备好示例镜像，您可以直接使用任一镜像，快速体验 Service Mesh 微服务。
- 改造已有的本地工程：您需要在本地代码中，完成服务注册，即将工程项目接入 SOFA 服务注册中心。

  说明：有关本地 SOFABoot 工程的开发与部署，可参考 开始使用 SOFARPC 。要使用 Service Mesh 微服务，只需对本地工程进行改造即可。如需获取更多代码详情，可点击此处 下载示例代码 。

示例镜像

类型	镜像地址	说明
SOFA	reg-cnsh-nf.cloud.alipay.com/library/sofahellomeshserver:1.0	接口：com.alipay.sofa.mesh.facade.HelloMeshFacade，方

Service	.1	法名：helloMesh。
SOFA Client	reg-cnsh-nf.cloud.alipay.com/library/sofahellomeshclient:1.0.1	
Dubbo Service	reg-cnsh-nf.cloud.alipay.com/library/dubboechoprovider:1.0.1	接口：com.alibaba.dubbo.samples.echo.api.EchoService，方法名：echo。
Dubbo Client	reg-cnsh-nf.cloud.alipay.com/library/dubboechoconsumer:1.0.1	
SpringCloud Service	reg-cnsh-nf.cloud.alipay.com/library/springcloudreservationservice:1.0.1	应用名：reservation-service，接口路径：/reservations/names
SpringCloud Client	reg-cnsh-nf.cloud.alipay.com/library/springcloudreservationclient:1.0.1	应用名：reservation-client

改造本地工程

Service Mesh 微服务需要将本地工程接入 SOFA 服务注册中心，所以需要您需要对已有的项目进行一些改造。

SOFABoot 服务

1. 升级 SOFABoot 依赖版本至 3.1.1 版本:

```
<parent>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofaboot-enterprise-dependencies</artifactId>
<version>3.1.1</version>
</parent>
```

2. 通过以下任一方式，添加应用启动参数：
   - 在 application.properties 中配置如下：

```
com.alipay.env=shared
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.env=shared
-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：

```
COM_ALIPAY_ENV=shared
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：以上参数值是中间件的全局配置项，可在 **脚手架 > Step 2** 获取，详见 引入 SOFA 中间件 > 中间件全局配置 。

Dubbo 服务

1. 在 pom.xml 文件中，引入以下 SDK 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-registry-cloud-all</artifactId>
<version>1.2.2</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：以上参数值是中间件的全局配置项，可在 **脚手架 > Step 2** 获取，详见 引入 SOFA 中间件 > 中间件全局配置 。

Spring Cloud 服务

1. 在 pom.xml 文件中，引入以下 SDK 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-registry-cloud-all</artifactId>
<version>1.2.2</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>
```

2. 注释掉或移除以下依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

3. 通过以下任一方式，添加应用启动参数：

- 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=
SOFA_ANTVIP_ENDPOINT=
SOFA_SECRET_KEY=
SOFA_ACCESS_KEY=
```

参数说明：以上参数值是中间件的全局配置项，详见 引入 SOFA 中间件 > 中间件全局配置 。

改造完成后，您需要将该应用打包成镜像，详见 制作 SOFABoot 应用的 Docker 镜像 。镜像成功构建后，需将其上传至容器应用服务镜像仓库，参见 镜像仓库 。

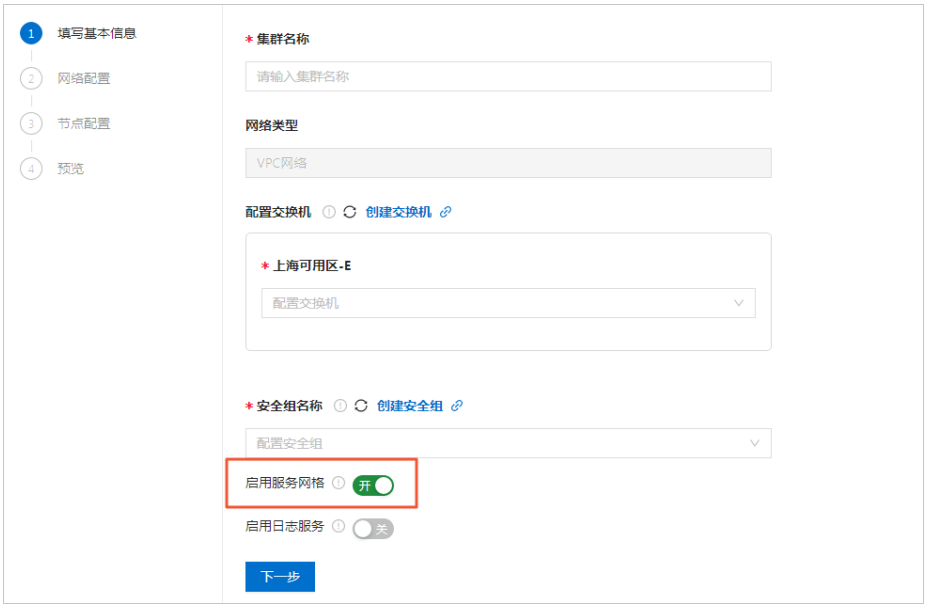
发布应用服务

发布应用

1. 进入应用管理控制台，选择 **应用列表**。
2. 在应用列表上方，点击 **创建应用**。
3. 在创建页面输入应用信息：
    - 应用名称：应用的名称。
    - 技术栈：Service Mesh 应用不依赖技术栈信息，可以选择 Spring Boot 或 SOFABoot。
4. 点击 **确定**，待应用状态变成 **创建完成**，即完成应用创建。

创建集群

1. 进入容器应用服务控制台，选择 **集群管理 > 集群详情**。
2. 在 **集群详情** 页，点击 **创建集群**。
3. 在 **创建集群** 页面，填写以下配置信息：
    - **基本信息**
        - 集群名称
        - 网络类型：默认仅支持虚拟私有网络（VPC）。
        - 配置交换机：创建一个新的交换机或选择当前可用区下的交换机来分配私网 IP。
        - 安全组名称：创建一个新的安全组或选择当前可用区的安全组。
        - 启用服务网格：默认关闭，必须 **启用服务网格**。
        - 启用日志服务：默认关闭。



- **网络配置**：保持默认配置即可。
- **节点配置**：
    - 可用区：从下拉列表中选择当前工作空间的可用区。
    - 部署单元：选择节点的部署单元。
    - 系列：选择节点对应的 ECS 系列。
    - 规格：选择节点对应的 ECS 规格。
    - 系统盘：支持 SSD 及 高效云盘。
    - 挂载数据盘：支持 SSD 及 高效云盘。
    - 实例数量：输入待创建的节点数量。
    - 实例名称前缀：指定节点名称的前缀，会已 -[n~(n+1)] 的格式自动为其添加序列号。
    - 设置 Root 密码：登录节点的密码，如遗忘可登录 ECS 控制台重置密码。
- **预览**：预览集群配置信息，确认无误。

4. 在 **集群配置概览** 页中点击 **创建**，等待集群创建完成。

创建并发布应用服务

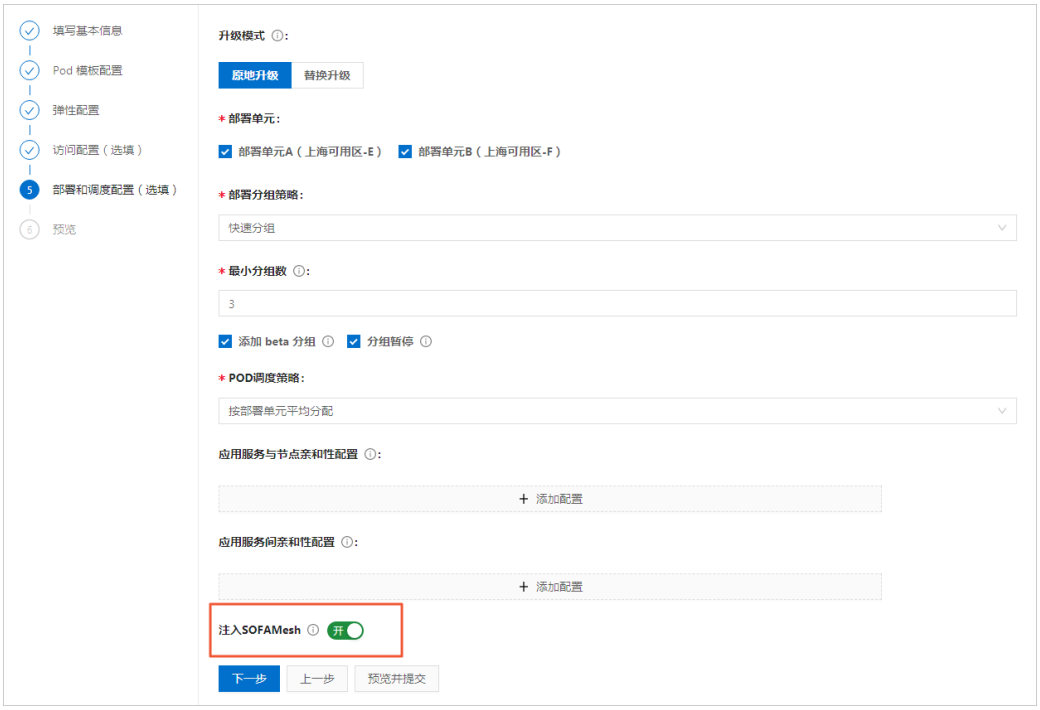
创建应用服务

1. 进入 **容器应用服务**，选择 **应用发布 > 应用服务**。
2. 进入应用服务列表页，点击 **创建**。
3. 在新打开页面，填写应用服务的基本信息：
    - 命名空间：必选，保持默认 default。
    - 应用服务名称：必填，自定义应用服务名
    - 所属应用：必选，选择刚刚在应用管理创建的应用

- 负责人：必选，默认为应用创建人
- 副本个数：必填，输入该应用服务需要部署的 Pod 数量

4. 点击 **下一步**，开始配置 Pod 模板：
   - 容器名称：容器名称
   - 镜像地址：输入镜像仓库地址
   - CPU 配置：容器使用的 CPU 的数量
   - 请求核数：能保证的最小核数
   - 最大核数：能使用的最大核数
   - 内存配置：容器使用的内存的数量
   - 请求内存：能保证的最小内存数量
   - 最大内存：能使用的最大内存数量
   - 环境变量配置：指定容器启动后会有哪些系统环境变量。**如果您使用的是上述提供的 示例镜像 ，必须在此配置中间件全局配置项**，如下图所示：

5. 点击 **下一步**，配置访问：可选择 集群内访问、VPC 内网访问、公网访问。
6. 点击 **下一步**，配置部署和调度：
   - 升级模式：原地升级、替换升级。
   - 部署单元：选择部署单元。
   - 部署分组策略：快速分组、每个 POD 一组、公分一组。
   - 最小分组数：部署时保证的最小分组数量。
   - POD 调度策略：按部署单元平均分配。
   - **注入 SOFAMesh**：必须开启此功能。



7. 点击 **预览并提交**，完成应用服务创建。

发布应用服务

1. 在容器应用服务控制台，选择 **应用发布 > 发布单**。
2. 在发布部署大盘，点击 **创建发布单**。
3. 在 **创建发布单** 页面，填写以下发布信息后：
    - 基本信息
        - 标题：发布单标题。
        - 部署单元：展示当前工作空间下的所有部署单元。
        - 应用服务发布列表：选择需要发布的应用服务。
    - 高级配置：设置应用服务之间的依赖关系。
4. 在 **预览** 页面，确认信息无误后，点击 **创建**。
5. 在 **发布单详情** 页面，点击 **整体发布** 即可开始发布。

服务监控及治理

应用服务发布完成后，您可以前往 **中间件 > 微服务 > Service Mesh MS** 查看服务详情 及 实时监控服务状态等。如发现服务异常，您可以通过添加限流规则或路由规则进行服务治理。

> **说明**：当前版本仅支持 SOFA 和 Dubbo 服务的限流和路由，暂不支持 Spring Cloud 服务的限流和路由。

添加服务限流规则

如您发现服务流量过载，您可以通过添加服务限流规则进行服务限流。

1. 在服务详情页，切换至 **服务限流** 页签，点击 **添加限流规则**。
2. 在右侧新窗口中，配置以下规则信息：
    - 规则名称：限流规则的名称
    - 应用：应用名
    - 服务：应用服务名称
    - 服务类型：应用服务的类型
    - 方法：方法名
    - 限流算法：目前仅支持扩展令牌桶算法。关于限流算法的更多信息，参见 限流规则说明 > 令牌桶算法 。
    - 令牌通系数：默认为 1
    - 限流阈值
        - 条件模式：目前仅支持 QPS，即根据单位时间内的请求数进行限流
        - 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：即 QPS 计数值，单位时间内的请求数
        - 流量类型：限流的流量类型，目前仅支持所有流量
3. 配置完成后，点击 **提交**，规则进入 **新建中** 状态。

如限流规则创建失败，您可以选择 **重试** 或 **返回修改** 配置信息后重新提交。

添加服务路由规则

如果发现有服务提供者状态异常，您可以通过配置服务路由规则，切光流量。

1. 在服务详情页，切换至 **服务路由** 页签，点击 **添加路由规则**。
2. 在右侧新窗口中，配置以下规则信息：
    - 规则名称：服务路由规则的名称
    - 应用：应用名。
    - 路由类型：目前仅支持按版本路由。
    - 分流开关：默认开启。分流开启后，如果路由目标没有机器，会主动调整到其他路由目标的可用机器上。
    - 版本权重：设置各版本权重，且需确保各版本权重和必须为 100%。
3. 配置完成后，点击 **提交**。
4. 返回路由规则列表，在 **状态** 列启用该规则。

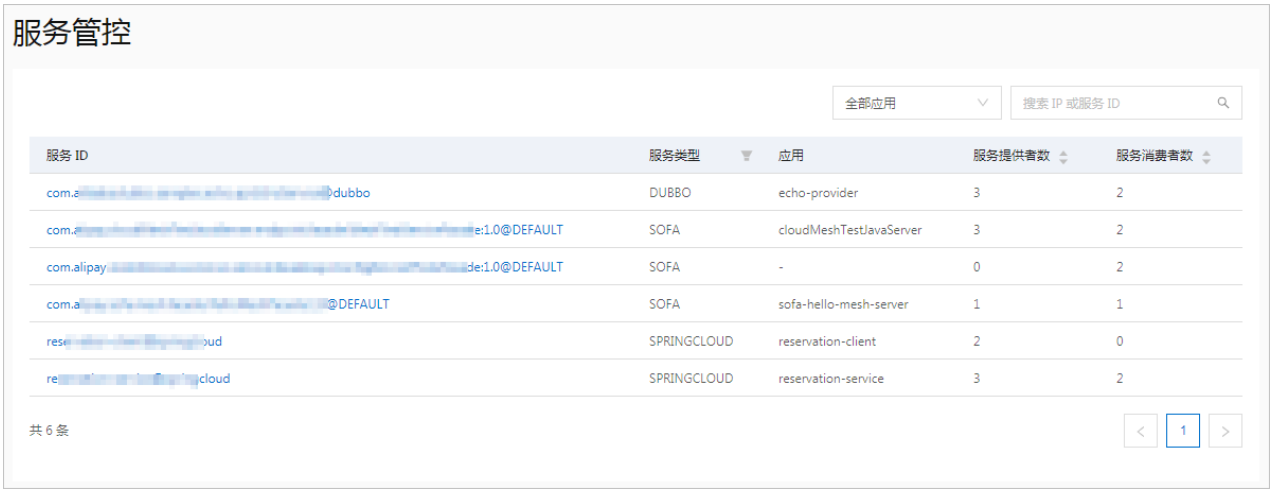
3.2 查看及管理应用服务

在 **微服务 > Service Mesh MS > 服务管控** 页面，您可以查看并管理当前工作空间下的所有启用了服务网格的应用服务。

查看服务列表

在 **服务管控** 页面，当前工作空间下所有在容器服务发布并启用了服务网格的应用服务以列表形式呈现。服务列表展现了以下信息：

- 服务 ID：服务的唯一标识
- 服务类型：服务的类型，包括 SOFA、Spring Cloud、Dubbo。
- 应用：提供该服务的应用名，一个服务可能会属于多个应用。
- 服务提供者数量
- 服务消费者数量



查看服务详情

在服务列表中，点击服务 ID，即可进入该服务的详情页，提供了以下信息：

- 服务提供者列表：服务提供者的 IP、协议、端口、应用版本、权重与状态信息。您可以在此处启用、禁用服务或修改服务权重，详见 管理应用服务 。
- 服务消费者列表：服务消费者的 IP 与应用名。
- 服务限流规则列表：服务限流规则名称、应用名、限流方法、拦截模式、修改时间与状态信息。详见 添加限流规则 。
- 服务路由规则列表：服务路由规则名、应用名、路由类型、修改时间与状态信息。详见 添加服务路由规则 。

管理应用服务

在 **服务详情 > 服务提供者** 页签，您可以对服务进行以下操作：

- 启用服务
- 禁用服务
- 修改权重

说明：目前仅支持对 SOFA 类型的应用服务进行启用禁用、权重修改操作。

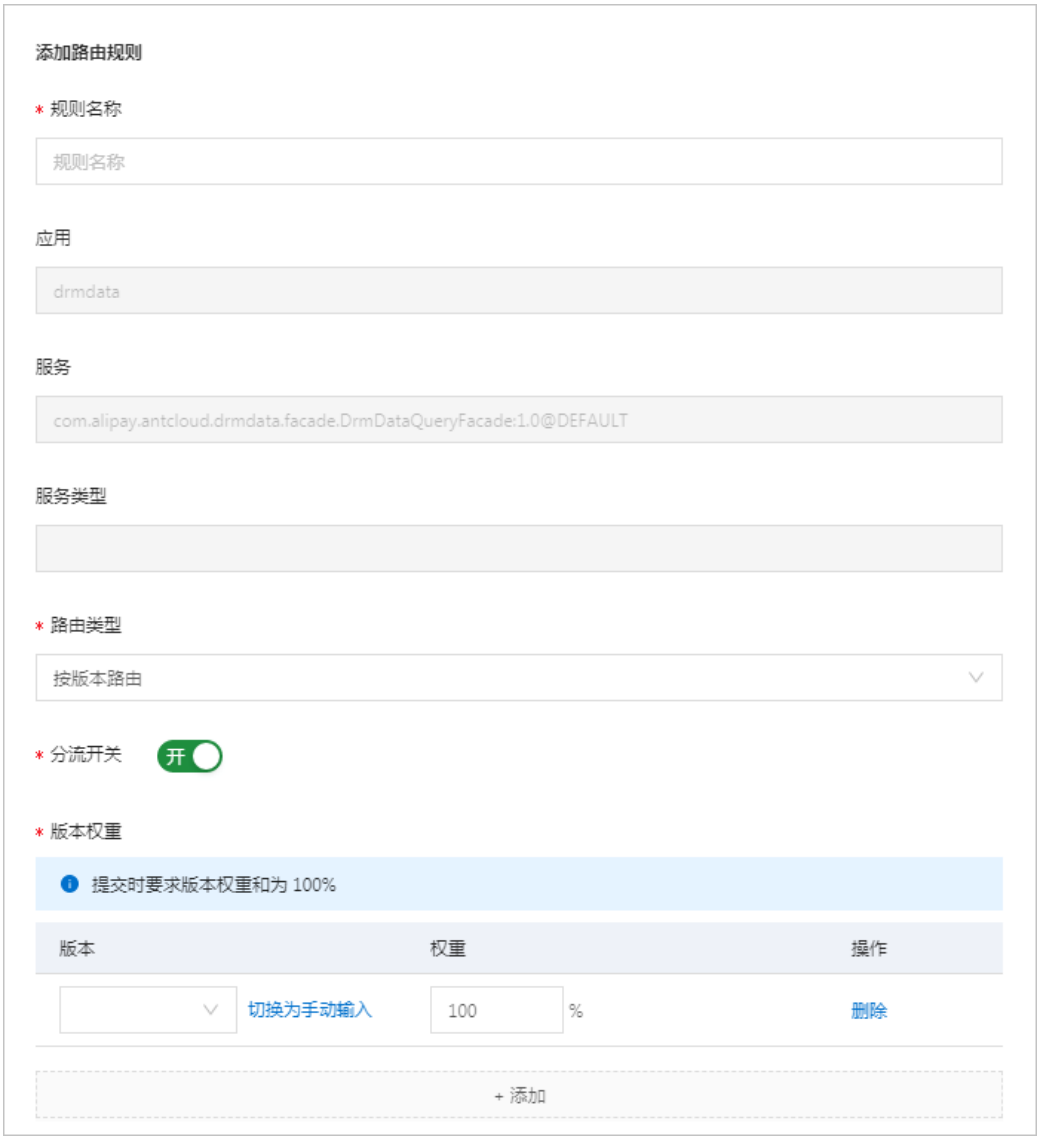
3.3 配置服务路由规则

如果发现有服务提供者状态异常，您可以在 **服务路由** 页面通过配置服务路由规则，切光流量。您也可以根据业务需要修改或删除现有的服务路由规则。

说明：当前版本仅支持 SOFA 和 Dubbo 服务的路由，暂不支持 Spring Cloud 服务的路由。

添加服务路由规则

1. 在微服务控制台页面，选择 **Service Mesh MS > 服务管控**，进入服务列表页。
2. 在服务列表，选择您想要配置路由的目标服务，点击其服务 ID，进入服务详情页。
3. 切换至 **服务路由** 页签，点击 **添加路由规则**。
4. 在右侧新窗口中，配置以下规则信息：
    - 规则名称：服务路由规则的名称。
    - 应用：应用名称。
    - 路由类型：目前仅支持按版本路由。
    - 分流开关：默认开启。打开分流意味着自适应调整错误流量路由转发规则，关闭分流意味着按照用户的路由规则真实处理。
    - 版本权重：设置各版本权重，且需确保各版本权重和必须为 100%。



5. 配置完成后，点击 **提交**。
6. 返回路由规则列表，在 **状态** 列启用该规则。

说明：启用规则会覆盖已启用规则，请谨慎操作。

修改服务路由规则

1. 在服务限路由规则列表，找到想要修改的路由规则。
2. 点击 **操作** 列的 **编辑** 按钮，即可修改限流配置。
3. 修改完成后，点击 **提交**。

删除服务路由规则

1. 在服务限路由规则列表，找到想要删除的路由规则。
2. 点击 **操作** 列的 **删除** 按钮。
3. 在确认窗口，选择 **确定**。

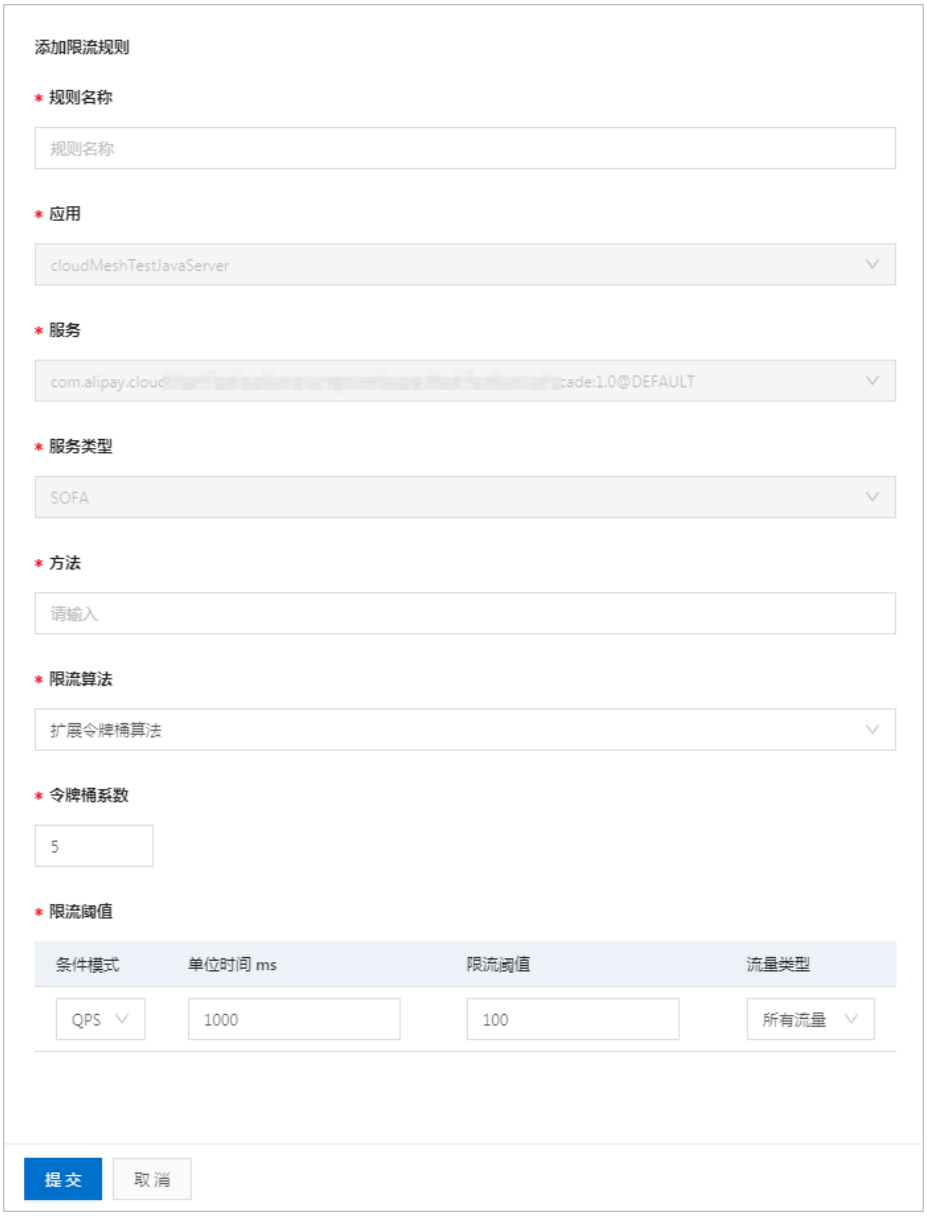
3.4 配置服务限流规则

对于流量过载的应用服务，您可以在 **服务限流** 页面通过添加限流规则，进行服务限流。您也可以根据需要修改或删除现有的服务限流规则。

> **说明**：当前版本仅支持 SOFA 和 Dubbo 服务的限流，暂不支持 Spring Cloud 服务的限流。

添加限流规则

1. 在微服务控制台页面，选择 **Service Mesh 微服务 > 服务管控**，进入服务列表页。
2. 在服务列表中，选择您想要限流的目标服务，点击其服务 ID，进入服务详情页。
3. 切换至 **服务限流** 页签，点击 **添加限流规则**。
4. 在右侧新窗口中，配置以下规则信息：
    - 规则名称：限流规则的名称
    - 应用：应用名
    - 服务：应用服务名称
    - 服务类型：应用服务的类型
    - 方法：方法名
    - 限流算法：目前仅支持扩展令牌桶算法。关于限流算法的更多信息，参见 限流规则说明 > 令牌桶算法 。
    - 令牌桶系数：默认为 1。
    - 限流阈值
        - 条件模式：目前仅支持 QPS，即根据单位时间内的请求数进行限流。
        - 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：即 QPS 计数值，单位时间内的请求数。
        - 流量类型：限流的流量类型，目前仅支持所有流量。



5. 配置完成后，点击 **提交**，规则进入 **新建中** 状态。

如限流规则创建失败，您可以选择 **重试** 或 **返回修改** 配置信息后重新提交。

编辑限流规则

1. 在服务限流规则列表，找到想要修改的限流规则。
2. 点击 **操作** 列的 **编辑** 按钮，即可修改限流配置。
3. 修改完成后，点击 **提交** 即可。

删除限流规则

对于已无用的限流规则，您可以对其进行删除操作。

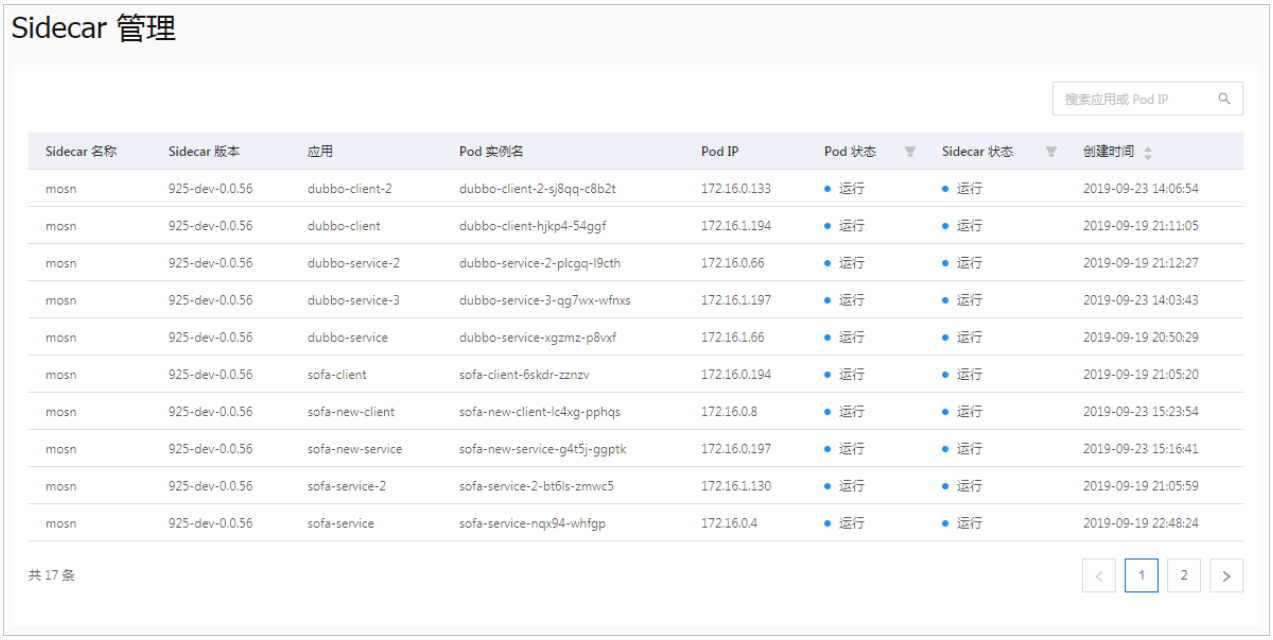
1. 在服务限流规则列表，找到想要删除的限流规则，点击右侧 **删除** 按钮。

2. 在确认窗口，点击 **确定**。

3.5 查看 SideCar 实例列表

在 **微服务 > Service Mesh MS > SideCar 管理** 页面，您可以查看到当前工作空间下所有的 SideCar 实例列表。列表提供了以下信息：

- **SideCar 名称**：SideCar 实例的名称。
- **SideCar 版本**：注入的 SideCar 版本。
- **应用**：应用名称。
- **Pod 实例名**：Pod 实例的名称。
- **Pod IP**：业务 Pod 的 IP 地址信息。
- **Pod 状态**：业务 Pod 的状态信息，包括 堵塞、运行、成功、失败、未知。
- **SideCar 状态**：SideCar 实例的状态，包括 运行、结束、等待、未知。
- **创建时间**：SideCar 实例创建时间。



3.6 查看服务拓扑关系

实际业务中，应用之间的关联与依赖非常复杂，需要通过全局视角检查具体的局部异常。您可以在 **服务拓扑** 页面查看应用在指定时间内的调用及其性能状况。如发现问题，可以通过 服务路由 和 服务限流 进行服务治理。

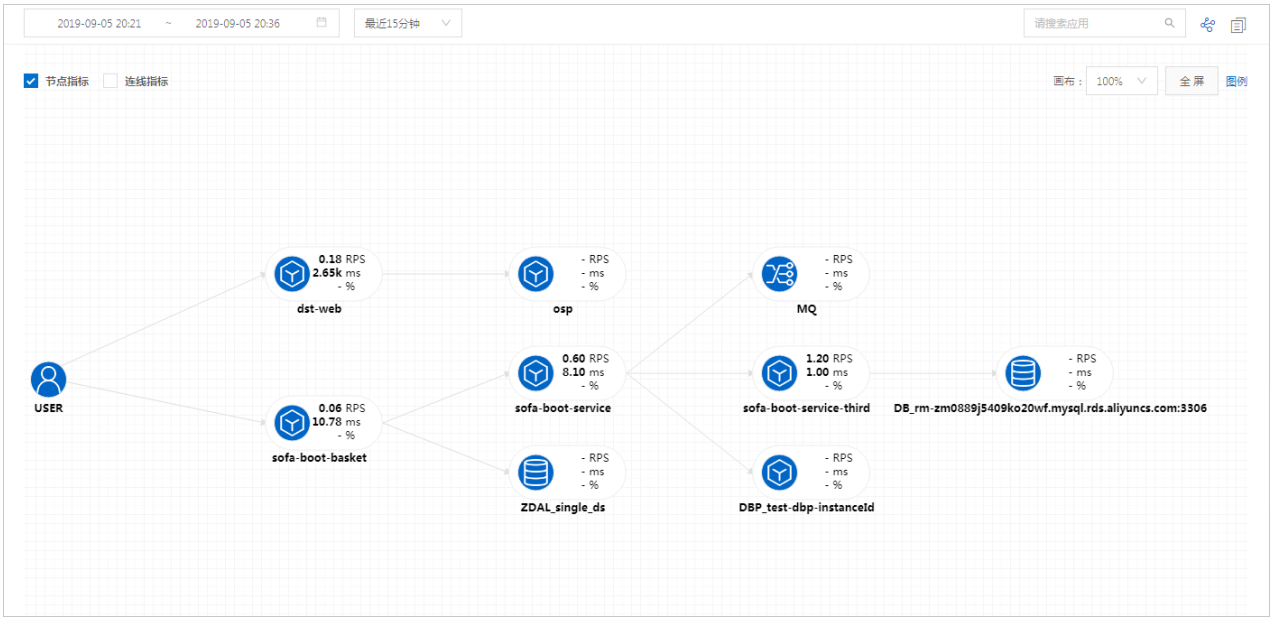
操作步骤

1. 在微服务控制台页面，选择 **Service Mesh MS > 服务拓扑**，打开服务拓扑图。
2. 在拓扑图的左上方，设置起止时间。默认范围为最近 15 分钟，最长时间间隔为 7 天。
3. 选择您想要查看的指标（节点指标 或 连线指标），拓扑图会相应显示或隐藏相关的指标信息。

服务拓扑图

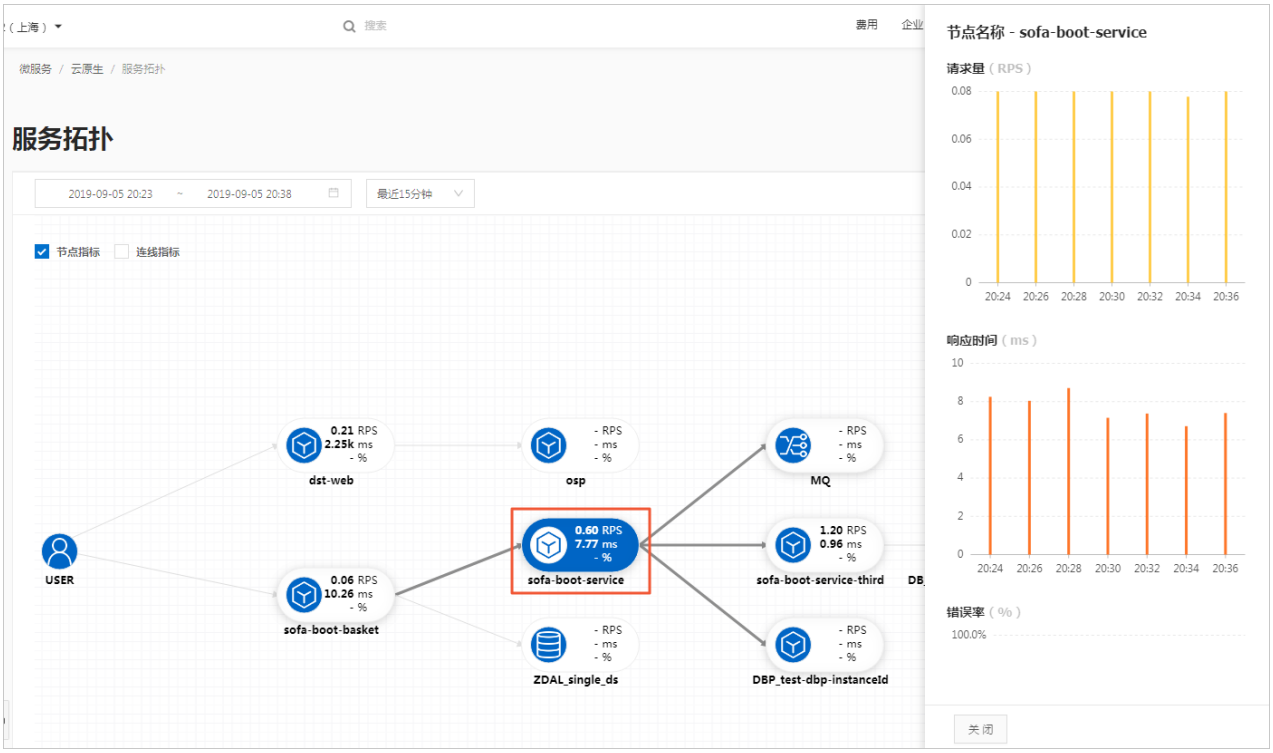
在服务拓扑图中，您可以获取以下信息：

- 应用服务的名称及版本号
- 应用服务间的调用关系
- 服务的请求量（RPS）
- 服务的响应时间（ms）
- 服务的错误率（%）



在服务拓扑图中，点击一个节点图标，即可查看该节点的详细信息。

- 节点上下游相关的拓扑连线。
- 应用节点信息：包括 请求量、响应时间 及 错误率。
- 数据库节点信息：包括 实例名称、IP、机房、QPS、错误率 及 响应时间。
- 缓存节点信息：包括 实例名称、IP、机房、QPS、命中率 及 响应时间。



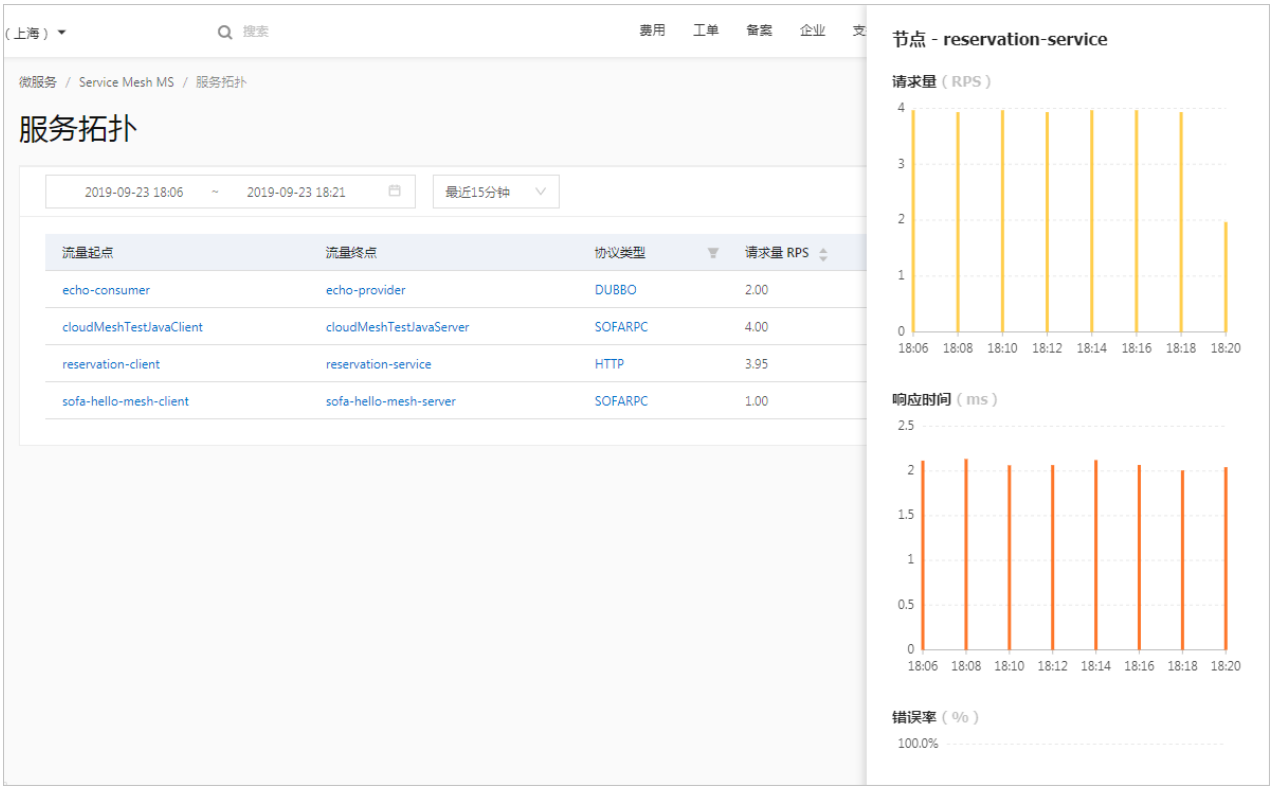
服务列表

您可以点击拓扑图右上角的列表图标，切换至列表视图，查看指定统计时段内的服务信息，包括流量起点、流量终点、协议类型、请求量 RPS、响应时间以及错误率。

流量起点	流量终点	协议类型	请求量 RPS	响应时间 ms	错误率
echo-consumer	echo-provider	DUBBO	2.00	2.2778	0.00%
cloudMeshTestJavaClient	cloudMeshTestJavaServer	SOFARPC	4.00	2.158	0.00%
reservation-client	reservation-service	HTTP	3.95	3.7504	0.00%
sofa-hello-mesh-client	sofa-hello-mesh-server	SOFARPC	1.00	2.0479	0.00%

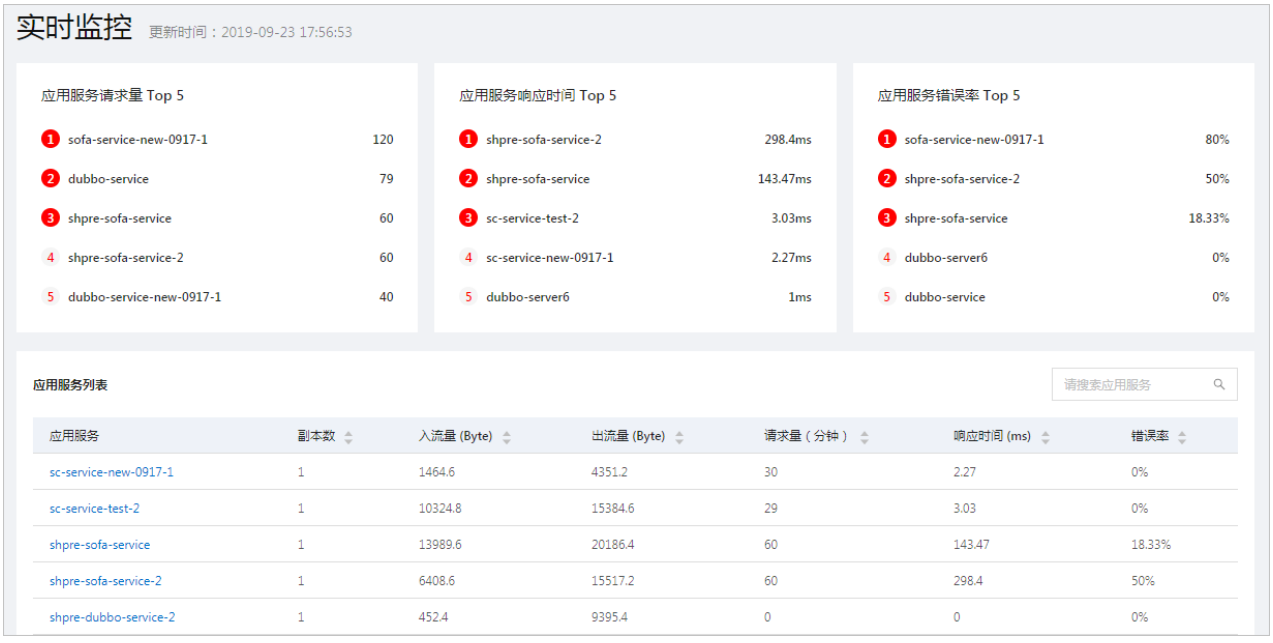
您还可以对应用进行如下操作：

- 点击任意流量起点应用或终点应用名称，即可查看该节点详情。
- 点击协议类型，即可查看流量概览信息。



3.7 实时监控

您可以在 **实时监控** 页面查看应用服务各项指标的总体统计数据，包括应用服务响应时间、错误率及请求量。



实时监控总览

应用服务 Top 5 排行榜

- **应用服务请求量 Top 5**：即请求量最多的 5 个应用，显示在默认统计时间窗口内请求量最多的 5 个应

用及其具体请求量数据。

- **应用服务响应时间 Top 5**：即响应时间最长的 5 个应用，显示在默认统计时间窗口内响应时间最长的 5 个应用及其具体响应时长。
- **应用服务错误率 Top 5**：即请求错误率最高的 5 个应用，显示在默认统计时间窗口内请求错误率最高的 5 个应用及其具体响应时长。

应用服务列表

列出当前环境下运行的所有应用及其监控概览信息。

- 应用服务：应用服务名称
- 副本数：副本数量
- 入流量：应用服务的入流量（byte）
- 出流量：应用服务的出流量（byte）
- 请求量：应用服务的请求量（分钟）
- 响应时间：应用服务的响应时间（ms）
- 错误率：应用服务的错误率

应用服务实时监控详情

在 **实时监控** 页面，您可以点击应用服务名称，进入该应用服务的监控详情页，查看该应用服务的各项监控指标数据，及时发现 Pod 异常状况或服务流量过载等，及时处理问题。

1. 在 **实时监控** 页面，点击您想要查看监控详情的应用服务名称，进入该应用服务的监控详情页。
2. 在页面顶部的筛选区域，您可以指定以下信息：
   - **应用版本**：选择应用版本。
   - **Pod**：选择 Pod。
   - **时间段**：选择统计窗口的起止时间，默认为最近 5 分钟。

3. 点击 **搜索**，页面显示当前指定条件内应用的监控信息，包括以下两部分。
   - **服务指标**：默认显示该应用所有服务的请求量、响应时间以及错误率趋势。您也可以选择查看指定服务的性能状况。
   - **系统指标**：显示了统计时间段内应用的 CPU 使用率、内存使用率、入流量以及出流量的趋势走向。

如发现某个应用服务异常，可通过添加限流规则或路由规则进行服务治理。详见 服务路由 及 服务限流 。

4 教程

4.1 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。

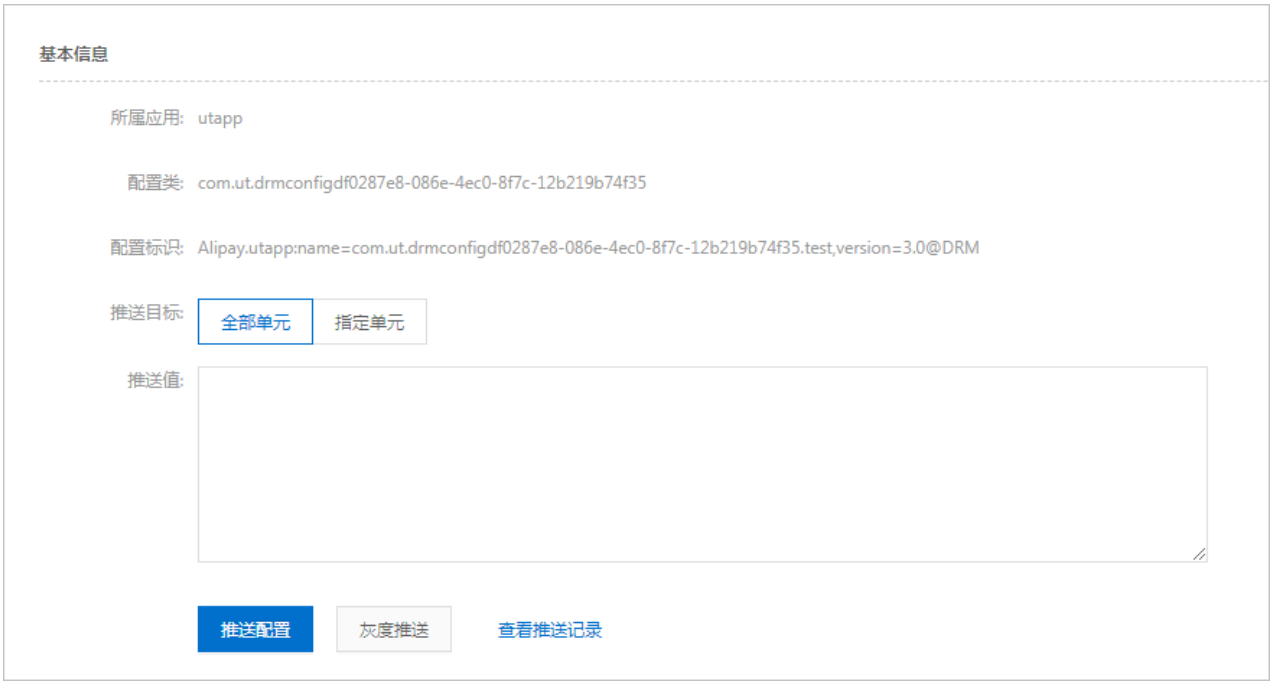
发布应用

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联此配置的客户端，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有关联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：

- 一个被称为配置类的普通 Java 类（即配置类），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” 。

4.2 Dubbo 连接 SOFA 注册中心

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。

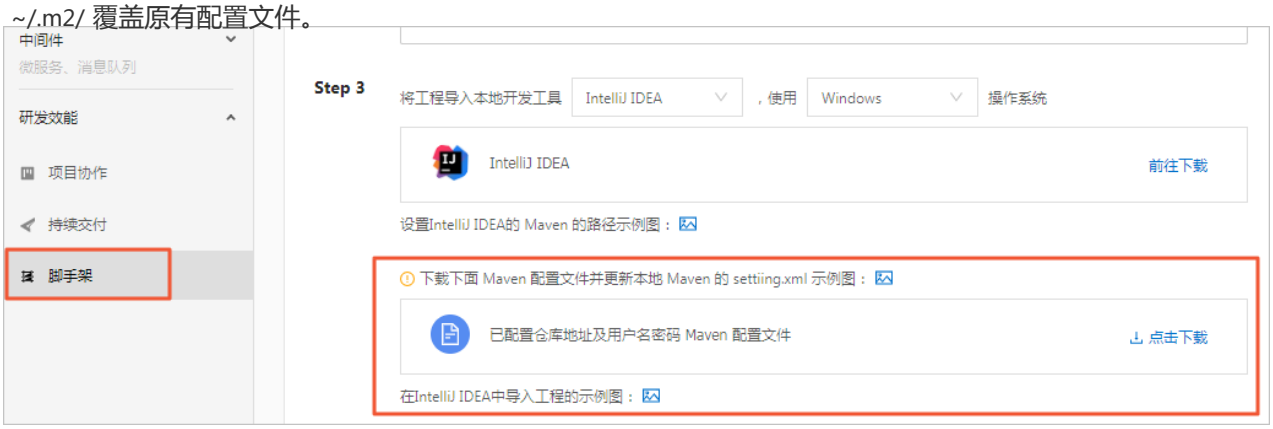
前置条件

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

SOFA 微服务与 Service Mesh 微服务模式下的 Dubbo 服务注册有所不同：

- **开发应用**：SOFA 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 Service Mesh 微服务的 sofa-registry-cloud-all。
- **发布应用**：SOFA 微服务要求在 **经典应用服务** 发布部署应用；而 Service Mesh 微服务要求在 **容器应用服务** 发布应用，且需要在集群层 **启用服务网格**，在应用层开启 **注入 SOFAMesh**。

SOFA 微服务

以下是引入 sofa-registry-dubbo-all 依赖的服务注册方法，如需使用 sofa-registry-cloud-all 方法，参见 Service Mesh 微服务 。

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 。

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

Service Mesh 微服务

1. 在 pom.xml 文件中，引入以下 SDK 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-registry-cloud-all</artifactId>
<version>1.2.2</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
```

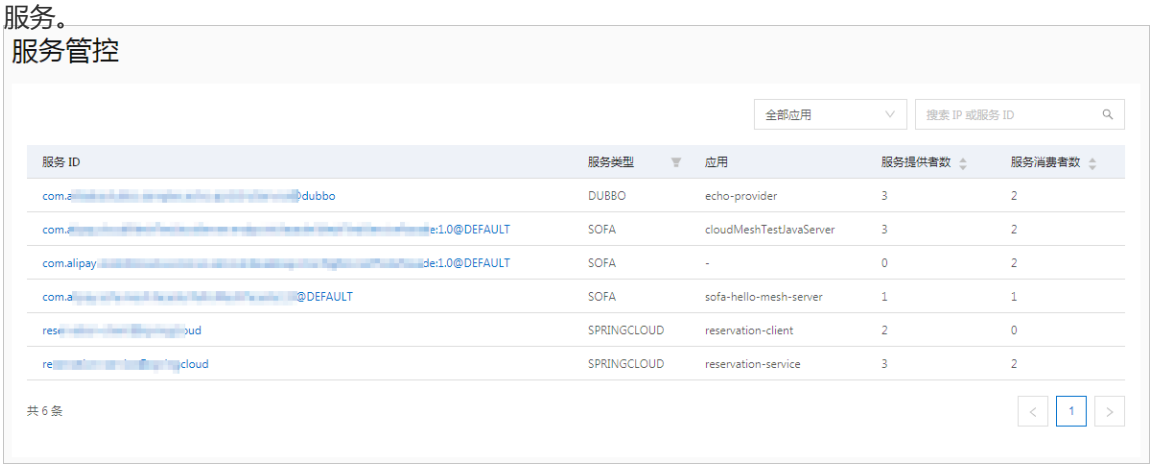
    - 指定系统环境变量：

```
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 。

4. 将改造完成的应用打包成镜像，详见 镜像构建 。
5. 前往 **容器应用服务** 发布应用，同时需要 **启用服务网格** 并 **注入 SOFAMesh**，详见 快速入门 > 发布应用服务 。
6. 应用发布后，即可前往 **中间件 > 微服务 > Service Mesh MS > 服务管控** 控制台，验证查看发布的服务。



查看日志

前往 {user.dir}/logs/registry/ 目录查看注册中心的日志。

相关链接

- 快速入门
- 服务路由与服务注册中心
- 创建 SOFABoot 工程

4.3 Spring Cloud 连接 SOFA 注册中心

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。

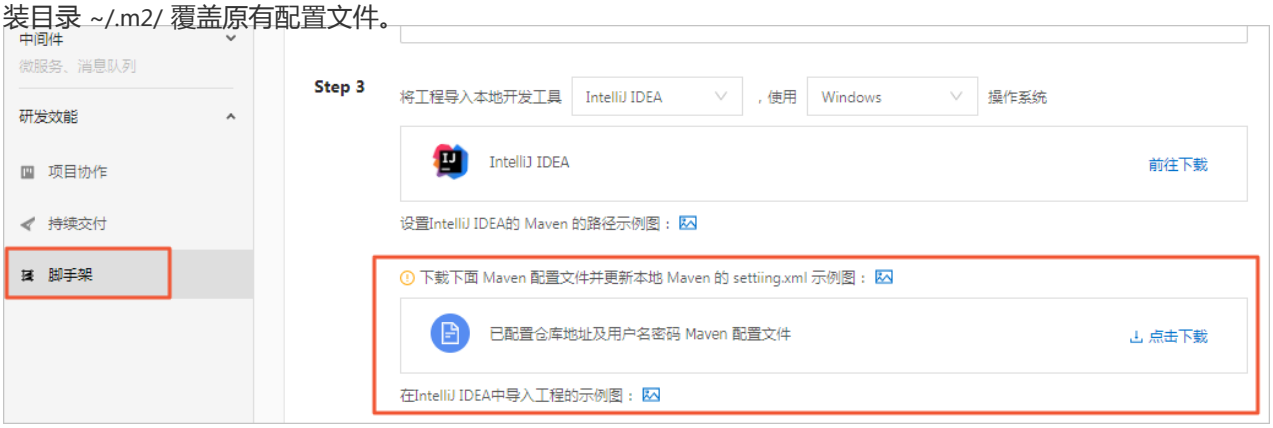
前置条件

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 > 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

SOFA 微服务与 Service Mesh 微服务模式下的 Spring Cloud 服务注册相同，两个模式主要区别如下：

- SOFA 微服务要求在 **经典应用服务** 发布部署应用。
- Service Mesh 微服务要求在 **容器应用服务** 发布应用，且需要在集群层 **启用服务网格**，在应用层开启 **注入 SOFAMesh**。

开发应用

1. 在 pom.xml 文件中，引入以下 SDK 依赖：

```
<dependency>
<groupId>com.alipay.sofa</groupId>
<artifactId>sofa-registry-cloud-all</artifactId>
<version>1.2.2</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>
```

2. 检查是否存在以下依赖，如存在，需将其移除：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

3. 通过以下任一方式，添加应用启动参数：

- 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 。

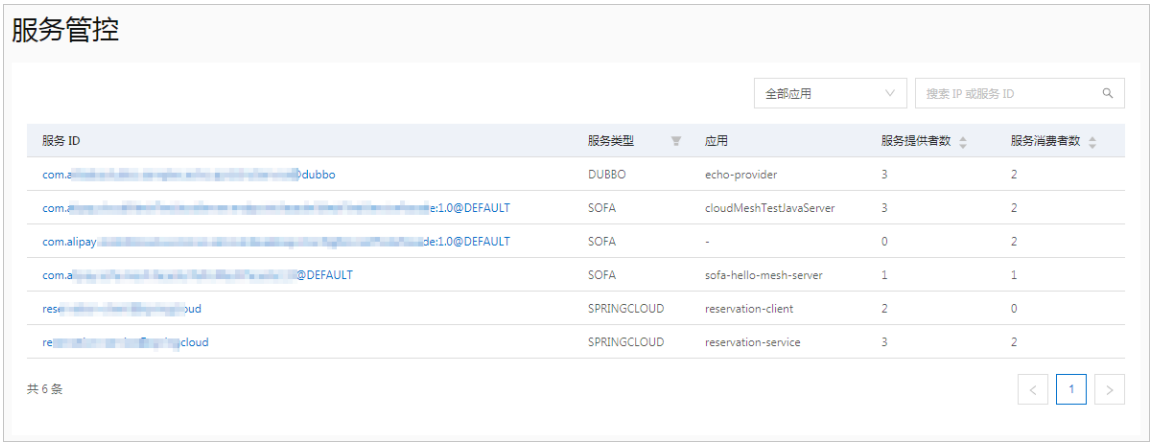
发布部署应用

SOFA 微服务

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 > SOFA MS > 服务管控** 控制台页面查看发布的服务。

Service Mesh 微服务

1. 将工程打包成镜像，详见 镜像构建 。
2. 前往 **容器应用服务** 发布应用，同时需要 **启用服务网格** 并 **注入 SOFAMesh**，详见 快速入门 > 发布应用服务 。
3. 应用发布后，即可前往 **中间件 > 微服务 > Service Mesh MS > 服务管控** 控制台，验证查看发布的服务。



查看日志

前往 {user.dir}/logs/registry/ 目录，即可查看注册中心的日志。

相关链接

- 快速入门
- 服务路由与服务注册中心
- 创建 SOFABoot 工程

4.4 接入并配置定时任务

通过本教程，您将学会如何使用定时任务，包括相关的配置。本课程包括以下内容：

1. 编码 ：接入定时任务，下载运行示例代码。
2. 控制台操作 ：完成本地编码后，需要在服务端管控台中提交相关的配置。
3. 日志确认 ：在 SOFAStack 上部署应用后，您需要通过日志来确认是否已经开始正常运行。

前序课程

本教程的示例代码是基于 SOFABoot 开发的。学习本教程前，确保您对 SOFABoot 有一定程度的了解，并了解定时任务运行机制。

- 了解 SOFABoot 基础知识： SOFABoot 快速入门 ；
- 了解定时任务运行机制： 开始使用定时任务 。

编码

下载示例代码

点击此处 下载示例工程，项目示例代码位于 middleware-v2/antscheduler-demo 文件夹下。

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

该示例应用是普通的 SOFABoot Core 工程，参考 本地编译运行 及 发布应用 完成编译打包并将其部署到云端。

控制台操作

配置定时任务

参考 定时任务快速入门 完成配置。注意任务名称需要和代码中的任务处理器的 getName() 的返回值保持一致。

日志确认

在 SOFAStack 上通过网页端 SSH 登录至 antscheduler-demo 的 ECS，任务处理相关的日志保存在 /home/admin/logs/scheduler/common-default.log。查看日志以确认应用是否开始正常运行。


蚂蚁金服
ANT FINANCIAL
金融科技
FINANCIAL TECHNOLOGY