目 录

1.产品简介

腾讯微服务平台 （Tencent Service Framework） 是一个围绕着应用和微服务的 PaaS 平台，提供应用全生命周期管理、数据化运营、立体化监控和服务治理等功能。TSF 拥抱 Spring Cloud 、Service Mesh 微服务框架，帮助企业客户解决传统集中式架构转型的困难，打造大规模高可用的分布式系统架构，实现业务、产品的快速落地。

TSF 以腾讯云中间件团队多款成熟的分布式产品为核心基础组件，提供秒级推送的分布式配置服务、链路追踪等高可用稳定性组件。此外，TSF 与腾讯云 API 网关和消息队列打通，让企业轻松构建大型分布式系统。

服务注册发现

TSF 服务注册发现包括三个角色，服务提供者，服务调用者和服务注册中心。服务提供者和服务调用者将地址信息注册到服务注册中心，并从服务注册中心获取所有注册服务的实例列表，服务调用者使用服务提供者的实例地址进行调用。

应用生命周期管理

TSF 提供从创建应用到运行应用的全程管理，功能包括创建、删除、部署、回滚、扩容、下线、启动和停止应用。TSF 提供部署组来实现应用的版本控制功能。TSF 将每次操作记录下来，用户可以在应用的变更记录页面中查看和搜索变更记录。

分布式配置管理

配置管理包括应用配置、全局配置和文件配置。用户可以通过控制台进行分布式配置版本管理，发布配置到部署组或者命名空间范围内的实例。

细粒度服务治理

TSF 提供服务级和 API 级别的服务治理能力。支持控制台上进行配置服务路由、服务限流、服务鉴权规则。在 TSF 控制台上，用户可以通过配置、权重标签的形式进行细粒度的流量控制，实现灰度发布、就近路由、部分账号内测、流量限制、访问权限控制等功能。

数据化运营

TSF 提供全面的监控和分布式调用链分析工具，帮助用户把握应用上线后的运行状况。

- 监控包括应用监控，应用监控的指标包括应用的 QPS, 请求时间和请求出错率等。
- 分布式调用链分析包括调用链查询和调用链详情。用户可以根据时间范围和服务名等条件来查询一组调用链。调用链详情显示了请求经过每个服务的层次关系和耗时情况等信息。

TSF 提供日志分析能力，自动获取用户的业务日志并支持在 TSF 控制台上进行日志查看、日志检索，支持日志关键词告警功能，并提供日志与调用链联动排查线上问题。

分布式事务

TSF 集成了分布式事务能力，支持 TCC 模式分布式事务管理功能。对于跨数据库、跨服务的分布式场景，用户可以在控制台上查看事务运行情况并进行超时事务处理。保证事务的一致性。

优势项	TSF 服务治理平台	自建服务治理平台
服务注册中心	平台提供高可用的注册中心集群，金融级别容灾	自行搭建和维护
调用链能力	符合国人习惯的交互方式，与日志服务联动	使用开源组件，英文界面
应用部署	成熟、灵活的部署方案，支持灰度发布等高级能力	自行开发部署模块
日志服务	提供从采集、呈现、解析、检索一站式能力，帮助开发者快速定位目标日志	基于 ELK 等开源组件搭建
配置管理	分布式配置，支持从环境、版本、应用三个维度进行管理，配置动态推送和历史推送回溯能力	自行搭建和维护
服务治理	支持细粒度的服务路由、限流、鉴权功能，控制台进行可视化配置	自行搭建和维护
分布式事务	提供分布式事务解决方案，经腾讯内部多产品实践验证	使用开源或自行开发
微服务 API 网关	腾讯云 API 网关提供微服务网关能力，支持配置鉴权、限流等策略	基于 Zuul 等组件开发
与腾讯云服务整合	与腾讯云服务深度整合	基于云 API 开发
售后服务	提供稳定及时的售后服务，强大的技术、运维团队支持	企业运维团队支持

构建分布式服务系统

单体应用转变为分布式系统后，实现系统间的可靠调用是关键问题之一，涉及到路由管理，序列化协议等技术细节。

TSF 提供了 RESTful 调用方式和自研的高性能 RPC 框架，能够构建高可用、高性能的分布式系统，TSF 系统地考虑了分布式服务发现、路由管理、安全、负载均衡等细节问题。同时 TSF 将在未来打通消息队列、API Gateway 等服务，满足用户多样化的需求。

应用发布和管理

相对于传统的应用发布需要运维人员登录到每一台服务器进行发布和部署，TSF 针对分布式系统的应用发布和管理，提供了简单易用的可视化控制台。用户通过控制台可以发布应用，包括创建、部署、启动应用，也支持查看应用的部署状态。除此之外，用户可以通过控制台管理应用，包括回滚应用、扩容、缩容和删除应用。

数据化运营

通过对日志埋点的收集和分析，可以得到一次请求在各个服务间的调用链关系，有助于梳理应用的请求入口与服务的调用来源、依赖关系。当遇到请求耗时较长的情况，可以通过调用链分析调用瓶颈，快速定位异常。

2.Spring Cloud 开发手册

获取 Demo

Demo 下载 >>

工程目录

`tsf-simple-demo` 的工程目录如下：

```
|- consumer-demo
|- provider-demo
|- pom.xml
```

其中 consumer-demo 表示服务消费者， provider-demo 表示服务提供者。 pom.xml 中定义了工程需要的依赖包（以下以基于 Spring Cloud Finchley 版本 SDK 举例说明）：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-dependencies</artifactId>
        <version><!-- 指定版本号 --></version>
    </parent>

    <groupId>com.tsf.demo</groupId>
    <artifactId>tsf-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <modules>
        <module>provider-demo</module>
        <module>consumer-demo</module>
    </modules>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

其中 parent 描述了 `provider-demo` 和 `consumer-demo` 共同的 TSF 依赖。

```
<parent>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-dependencies</artifactId>
        <version><!-- 指定版本号 --></version>
</parent>
```

准备工作

开始实践服务注册发现功能前，请确保已完成了 SDK 下载。

实现服务注册和发现

通过一个简单的示例说明如何实践服务的注册和发现。创建 tsf-demo 工程，文件结构如下：

```
|- consumer-demo
|- provider-demo
|- pom.xml
```

其中 `pom.xml` 文件参考Demo 工程概述中的 `pom.xml` 内容。

创建服务提供者

此服务提供者提供一个简单的 echo 服务，并将自身注册到服务注册中心。

1. 创建 provider 工程

创建一个 Spring Cloud 工程，命名为 `provider-demo` 。

2. 修改 pom 依赖

在 `pom.xml` 中引入需要的依赖内容：

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>tsf-demo</artifactId>
    <version><!-- 关联 parent version 属性--></version>
</parent>

<artifactId>provider-demo</artifactId>
<packaging>jar</packaging>
<name>provider-demo</name>

<dependencies>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-consul-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-consul-config</artifactId>
    </dependency>
    <!-- 使用分布式配置自动刷新功能，需要显示添加actuator的依赖包 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-sleuth</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-auth</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-route</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-swagger</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-ratelimit</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-encrypt</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-monitor</artifactId>
    </dependency>
</dependencies>
```

3. 开启服务注册发现

添加服务提供端的代码，其中 `@EnableDiscoveryClient` 注解表明此应用需开启服务注册与发现功能。

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
}
```

4. 提供 echo 服务

创建一个 `EchoController` ，提供简单的 `echo` 服务。

```
@RestController
public class EchoController {
    @RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
    public String echo(@PathVariable String string) {
        return string;
    }
}
```

5. 修改配置

在 `resource` 目录下的 `application.yml` 文件中配置应用名与监听端口号。

```
server:
  port: 18081
spring:
  application:
    name: provider-demo
```

> !运行在 TSF 平台上的应用无须配置服务注册中心地址，SDK 会通过环境变量自动获取注册中心地址。

创建服务消费者

本示例中，我们将创建一个服务消费者，消费者通过 `RestTemplate` 、 `AsyncRestTemplate` 、 `FeignClient` 这三个客户端去调用服务提供者。

1. 创建 consumer 工程

创建一个 Spring Cloud 工程，命名为 `consumer-demo`

2. 修改 pom 依赖

在 `pom.xml` 中引入需要的依赖内容：

```
<parent>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>tsf-demo</artifactId>
        <version>1.12.1-Edgware-RELEASE</version>
</parent>

<artifactId>consumer-demo</artifactId>
<packaging>jar</packaging>
<name>consumer-demo</name>

<dependencies>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-consul-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-consul-config</artifactId>
    </dependency>
    <!-- 使用分布式配置自动刷新功能，需要显示添加actuator的依赖包 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-feign</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-sleuth</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-auth</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-route</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-swagger</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-ratelimit</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-encrypt</artifactId>
    </dependency>
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-monitor</artifactId>
    </dependency>
</dependencies>
```

因为此处需要演示 `FeignClient` 的使用，所以与 `provider-demo` 相比， `pom.xml` 文件中的依赖增加了一个 `spring-cloud-starter-feign` 。

3. 开启服务注册发现

与服务提供者 `provider-demo` 相比，除了开启服务与注册外，还需要添加两项配置才能使用 `RestTemplate` 、 `AsyncRestTemplate` 、 `FeignClient` 这三个客户端：

- 添加 `@LoadBalanced` 注解将 `RestTemplate` 与 `AsyncRestTemplate` 与服务发现结合。
- 使用 `@EnableFeignClients` 注解激活 `FeignClients` 。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class ConsumerApplication {
   @LoadBalanced
   @Bean
   public RestTemplate restTemplate() {
       return new RestTemplate();
   }

   @LoadBalanced
   @Bean
   public AsyncRestTemplate asyncRestTemplate(){
       return new AsyncRestTemplate();
   }

   public static void main(String[] args) {
       SpringApplication.run(ConsumerApplication.class, args);
   }
}
```

4. 设置调用信息

在使用 `EchoService` 的 `FeignClient` 之前，还需要完善它的配置。配置服务名以及方法对应的 HTTP 请求，服务名为 `provider-demo` 工程中配置的服务名 `provider-demo` ，代码如下：

```
@FeignClient(name = "provider-demo")
public interface EchoService {
    @RequestMapping(value = "/echo/{str}", method = RequestMethod.GET)
    String echo(@PathVariable("str") String str);
}
```

5. 创建 Controller

创建一个 `Controller` 供调用测试。

- `/echo-rest/*` 验证通过 RestTemplate 去调用服务提供者。
- `/echo-async-rest/*` 验证通过 AsyncRestTemplate 去调用服务提供者。
- `/echo-feign/*` 验证通过 FeignClient 去调用服务提供者。

```
@RestController
public class Controller {
   @Autowired
   private RestTemplate restTemplate;
   @Autowired
   private AsyncRestTemplate asyncRestTemplate;
   @Autowired
   private  EchoService echoService;
   @RequestMapping(value = "/echo-rest/{str}", method = RequestMethod.GET)
   public String rest(@PathVariable String str) {
       return restTemplate.getForObject("http://provider-demo/echo/" + str, String.class);
   }
   @RequestMapping(value = "/echo-async-rest/{str}", method = RequestMethod.GET)
   public String asyncRest(@PathVariable String str) throws Exception{
       ListenableFuture<ResponseEntity<String>> future = asyncRestTemplate.
               getForEntity("http://provider-demo/echo/"+str, String.class);
       return future.get().getBody();
   }
   @RequestMapping(value = "/echo-feign/{str}", method = RequestMethod.GET)
   public String feign(@PathVariable String str) {
       return echoService.echo(str);
   }
}
```

6. 修改配置

```
server:
  port: 18083
spring:
  application:
    name: consumer-demo
```

> !运行在 TSF 平台上的应用无须配置服务注册中心地址，SDK 会通过环境变量自动获取注册中心地址。

本地开发调试

启动轻量级注册中心

本地开发调试时，需要使用轻量级注册中心，轻量级注册中心包含了 TSF 服务注册发现服务端的轻量版，详请参见 轻量级服务注册中心。

启动应用

本地启动应用可以通过 IDE 和 FatJar 两种方式。

IDE 中启动

在 IDE 中启动，通过 VM options 配置启动参数 `-Dtsf_consul_ip=127.0.0.1 -Dtsf_consul_port=8500 -Dtsf_application_id=a -Dtsf_group_id=b -Dtsf.swagger.enabled=false`，通过 main 方法直接启动。其中 IP 和 port 取值为轻量级服务注册中心的地址，使用了分布式配置功能的模块，需要设置 `-Dtsf_application_id=a -Dtsf_group_id=b`， 取值可为任意值。

FatJar 启动

1. 添加 FatJar 打包方式

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

2 打包 FatJar 文件

添加完插件后，在工程的主目录下，使用 Maven 命令 `mvn clean package` 进行打包，及可在 target 目录下找到打包好的 FatJar 文件。

3. 通过 Java 命令启动

```
java -Dtsf_consul_ip=127.0.0.1 -Dtsf_consul_port=8500 -Dtsf.swagger.enabled=false -jar
provider-demo-0.0.1-SNAPSHOT.jar
```

其中 `127.0.0.1` 和8500为轻量级服务注册中心地址，在本地调试时 tsf_application_id 和 tsf_group_id 可以填任意值。

> !由于轻量级服务注册中心（原生的 consul）的 metadata 只能支持512个字节，因此需要关闭 TSF 的 API 上报能力 `-Dtsf.swagger.enabled=false`，如果没有这个启动参数，在本地运行 Demo 时将会报错，错误信息中包含 `Value is too long (limit 512 characters)`。

演示

启动服务，分别进行调用，观察调用结果。

`http://{consumer-demo-ip}:{consumer-demo-port}/echo-rest/test?user=test-tsf`

`http://{consumer-demo-ip}:{consumer-demo-port}/echo-async-rest/test?user=test-tsf`

`http://{consumer-demo-ip}:{consumer-demo-port}/test?user=test-tsf`

TSF 中部署应用

将打包好的 FatJar 程序包上传到 TSF 控制台，进行部署操作，无需关心额外配置。部署相关操作可参考 虚拟机应用部署组 或 容器应用部署组。

从 Eureka 迁移

已经接入 Eureka 服务注册与发现的应用，只需要修改 `pom.xml` 依赖，就可以将服务接入 TSF 服务注册发现中心。

1. 在工程根目录的 pom.xml 中增加 `spring-cloud-tsf-dependencies` 的 parent。参考上文中的 Demo 工程。
2. 在单个 Spring Cloud 应用的 `pom.xml` 中，将 `spring-cloud-starter-eureka` 替换成 `spring-cloud-starter-consul-discovery`。 替换前：

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

   替换后：

```
<dependency>
 <groupId>com.tencent.tsf</groupId>
 <artifactId>spring-cloud-tsf-consul-discovery</artifactId>
</dependency>
<!-- consul SDK 依赖的版本控制参考 Demo 或之前 pom 说明-->
```

3. 修改代码中的 Eureka 的相关注解。

```
@EnableEurekaClient   => @EnableDiscoveryClient
```

异常检查

通过 TSF 平台部署改造后的应用，发现应用无法启动，可以在工程中检查 `spring-cloud-starter-consul-discovery` 的版本，如果发现版本号中没有 `TSF-RELEASE` ，说明没有正确依赖 TSF 相关依赖，可参考 Demo 工程中 `pom.xml` 文件。

轻量级服务注册中心给开发者提供在开发、调试、测试的过程中的服务发现、注册和查询功能。

在一个公司内部，通常只需要在一台机器上安装轻量级服务注册中心。具体安装和使用的步骤请参见下文。

下载轻量级服务注册中心

推荐您找一台专门的机器启动轻量服务注册中心，比如某台测试机器。根据是否涉及到多个微服务联调测试，分为单机调试和多微服务联调两种场景进行说明。

本地使用轻量服务注册中心

如果 **不涉及** 到多个微服务联调场景，可以通过本地机器启动一个 Consul 作为轻量服务注册中心。单机调试支持 Windows 和 Linux / macOS 操作系统，多操作系统版本 Consul 下载地址。

确保机器以下的端口是空闲的：`8300`，`8301`，`8302`，`8500`，`8600`。

> 注： 用户可通过执行如下命令查看端口占用信息 `netstat -apn|grep LISTEN`

启动轻量级服务注册中心

本地使用轻量服务注册中心场景下支持 Windows 和 Linux / macOS 操作系统。 进入解压目录，启动服务注册中心。

- Windows 操作系统：

```
.\consul.exe agent -dev
```

- Linux / macOS 操作系统：

```
./consul agent -dev
```

验证服务注册中心启动

- 查看 8301 和 8500 的端口是否被监听；
- 通过浏览器查看服务注册中心页面（`http://127.0.0.1:8500`）。

多微服务联调环境的轻量服务注册中心

在多个微服务联调场景下，可以找一台可以被微服务访问的机器来部署轻量服务注册中心。目前本场景下仅支持 Linux 系统的 Consul，Linux 系统 Consul 下载地址。

启动轻量级服务注册中心

1. 将 consul 二进制文件放到任意一个目录，比如 /data/。
2. 将 start.sh 也放到同一个目录。下载脚本 start.sh。
3. 执行如下命令：

```
chmod +x start.sh; ./start.sh local_ip
```

其中 `local_ip` 填写本机 IP。比如，linux 机器上的 IP 为 192.168.1.10，那么执行的命令是：`./start.sh 192.168.1.10`。

验证服务注册中心启动

```
./consul members -http-addr=127.0.0.1:8500
curl http://127.0.0.1:8500/v1/catalog/services
```

如果有正常输出，代表 Consul 已经正常启动。

准备工作

开始实践分布式配置功能前，请确保已完成了 SDK 下载。 使用分布式配置功能涉及到以下部分：

- pom.xml 添加配置依赖项
- 在代码中引用配置
- 轻量级服务注册中心下发动态配置
- 通过 TSF 平台下发动态配置

一、依赖项

添加 pom.xml 依赖：

```
<dependency>
  <groupId>com.tencent.tsf</groupId>
  <artifactId>spring-cloud-tsf-consul-config</artifactId>
  <version><!-- 指定版本号 --></version>
</dependency>
 <!-- 使用分布式配置自动刷新功能，需要显示添加actuator的依赖包-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

二、代码引用配置

用户可通过两种方式更新代码中的配置信息：使用配置类 `@ConfigurationProperties` 和 `@Value` 注解。 `@Value` 比较适用于配置比较少的场景，而 `@ConfigurationProperties` 则更适用于有较多配置的情况。 用户也可以动态更新应用配置文件（如 application.yml）中的配置，如动态更改 redis 的地址或者鉴权功能开关等。

使用配置类 @ConfigurationProperties

在 `provider-demo` 的 ProviderNameConfig 类中，有一个字符串类型的变量 `name` 。其中：

- 使用 `@ConfigurationProperties` 注解来标明这个类是一个配置类。
- 使用 `@RefreshScope注解` 开启 refresh 机制。

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.stereotype.Component;

@Component
@RefreshScope
@ConfigurationProperties(prefix="provider.config")
public class ProviderNameConfig {
    private String name = "echo-provider-default-name";

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

使用 @Value 注解

在启动类 `ProviderApplication` 中，使用 `@Value` 注解来标识一个配置变量。下面的示例中 `${test.demo.prop:default}` 中 `test.demo.prop` 是在动态配置下发中使用的 key，value 默认是 `default` 。

```
// 其他 import
import org.springframework.web.bind.annotation.RestController;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;

public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }

    @Value("${test.demo.prop:defalut}")
    private String propFromValue;

    @RequestMapping("/hello/") public String index() {
        String result = "propFromValue: " + propFromValue + "\n";
        return result;
    }
}
```

配置更新触发回调

配置更新触发回调功能允许程序在不重启的情况下动态修改业务逻辑。当配置更新时，触发配置回调方法的调用。配置更新触发回调功能的使用场景包括：

- 程序使用一个防刷开关配置，当开关为启用状态时，启动防刷逻辑，当开关为关闭状态时，停用防刷逻辑。

- 程序使用一个 `ReidsConfig` 的动态配置类，包含 redis 的 host 和 port，当更新这个配置时，更新 Redis 实例。
- 配置更新后发出通知消息，通知本地或者远程的其他模块执行变更逻辑

TSF 分布式配置支持使用 `@ConfigChangeListener` 注解且实现 `ConfigChangeCallback` 接口。
在 `SimpleConfigurationListener` 类中，设置 `callback` 回调方法。支持场景包括：

- 支持按照配置项前缀模糊匹配方式。
- 支持配置项精确匹配方式。
- 支持回调方法同步 sync 或者异步 async 方式执行。

```
// prefix:  配置信息前缀，前缀匹配
// async: 回调事件是否异步执行，默认 false
// value:  精确匹配配置项
// 监听前缀为 io 的配置项变更
@ConfigChangeListener(prefix = "io",async = true)
public class SimpleConfigurationListener implements ConfigChangeCallback {
    @Autowired
    private SimpleService simpleService;
    @Override
    public void callback(ConfigProperty lastConfigItem, ConfigProperty currentConfigItem) {
        simpleService.saySomething("receive Consul Config Change Event >>>> ");
        simpleService.saySomething("Last config: [" + currentConfigItem.getKey()  + ":"
+lastConfigItem.getValue() + "]");
        simpleService.saySomething("Current config: [" + currentConfigItem.getKey()  + ":"
+currentConfigItem.getValue() + "]");
    }
}
@Component
public class SimpleService {
    public void saySomething(String words){
        System.out.println(words);
    }
}
```

三、轻量级服务注册中心下发动态配置

前提条件：

1. 已完成轻量级服务注册中心的安装和启动（参考 《轻量级服务注册中心》）。
2. 服务已连接到轻量级服务注册中心。

用户可以使用轻量级服务注册中心下发动态配置，下面以 provider-demo 为例说明更新的具体步骤：

1. 浏览器访问 consul 控制台。
2. 单击菜单栏【Key/Value】。
3. 单击【Create】，在配置新建界面填写 key 和 value。
    - key 是 /config/application/data 或 /config/application/{spring.profiles.active}/data

    - value 是配置的内容，截图中使用 provider-demo 中的自定义配置 provider.config.name=testname123。
4. 单击【Save】，观察 stdout 中是否出现 `[TSF SDK] Configuration Change Listener: key: {}, old value: null, new value: testname123`，如果出现说明配置生效。

四、通过 TSF 平台下发动态配置

用户可以通过 TSF 平台来下发动态配置，前提条件：

- 已经在 TSF 平台上部署了 `provider-demo` 和 `consumer-demo` 应用。
- 部署 `provider-demo` 的部署组的日志配置项的日志路径中包含了 `/tsf-demo-logs/provider-demo/root.log`，以确保打印的日志被采集后，可以通过控制台查看应用的日志。

关于如何通过控制台创建及下发更新的配置，请参考 《应用配置》。

如果希望修改 `ProviderNameConfig` 类中的 `providerName` 的值，创建配置时，配置内容填写：

```
provider:
  config:
    name: testname123
```

将配置发布到已部署 `provider-demo` 的部署组上，检查打印的日志中是否 name 的值已更新。如果已更新，说明更新的配置生效。

```
provider-demo -- provider config name: testname123
```

准备工作

开始实践服务鉴权功能前，请确保已完成了 SDK 下载 ，并阅读 《服务鉴权概述》。

快速上手

这里演示如何快速实践服务鉴权功能。假如现在有两个微服务 provider 和 consumer，想实现 consumer 调用 provider 时，provider 对请求做鉴权。

1. 向 provider 和 consumer 工程都添加依赖。在 `pom.xml` 中添加以下代码：

```
<dependency>
 <groupId>com.tencent.tsf</groupId>
 <artifactId>spring-cloud-tsf-auth</artifactId>
 <version><!-- 指定版本号 --></version>
</dependency>
```

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

1. 向 Application 类中添加注解 `@EnableTsfAuth`：

```
// 下面省略了无关的代码
import org.springframework.tsf.auth.annotation.EnableTsfAuth;
@SpringBootApplication
@EnableTsfAuth
public class ProviderApplication {
 public static void main(String[] args) {
     SpringApplication.run(ProviderApplication.class, args);
}
}
```

此时您已经对 provider 微服务开启了鉴权功能，任何到达 provider 的请求都会被鉴权，鉴权不通过时会返回 HTTP 403 Forbidden。

TSF 提供了两种类型的鉴权能力，一种根据调用方服务名鉴权，一种根据调用方设置的 tag 鉴权。在控制台上可以配置相应的规则。如果在控制台上对 provider 启用了鉴权功能，并且配置了至少一条规则，那么调用 provider 的微服务（如 comsumer）也需要引入 `spring-cloud-tsf-auth` 的依赖并且加上 `@EnableTsfAuth` 注解，否则到 provider 的请求会被返回 HTTP 403 Forbidden。

如果请求双方想使用基本 tag 的鉴权规则，那么：

- 对于 provider 而言，需要在控制台上设置 tag 鉴权规则。
- 对于 consumer 而言，需要在业务代码中设置 tag 的内容。

控制台上配置鉴权规则，参考 《服务鉴权基本操作》 。

在 consumer 中设置 tag，使用 `org.springframework.tsf.core` 包中的 `TsfContext` 类。设置 Tag 的方法签名如下：

```
/**
 * 设置多个 tag。如果有某个 tag 之前已经被设置过，那么它的值会被覆盖。
 */
public static void putTags(Map<String, String> tagMap, Tag.ControlFlag... flags) {}

/**
 * 设置单个 tag。如果该 key 之前已经被设置过，那么它的值会被覆盖。
 */
public static void putTag(String key, String value, Tag.ControlFlag... flags) {}
```

其中 `flags` 决定 tag 的使用场景，如果您没有特殊需要，不传即可：

```
public enum ControlFlag {
    TRANSITIVE,     // 表示标签要传递下去，默认不启用。
    NOT_IN_AUTH,    // 表示标签不被使用在服务鉴权，默认是被使用的。
    NOT_IN_ROUTE,   // 表示标签不被使用在服务路由，默认是被使用的。
    NOT_IN_SLEUTH   // 表示标签不被使用在调用链，默认是被使用的。
}
```

TSF 提供的 Demo `consumer-demo/src/main/java/com/tsf/demo/consumer/Controller.java` 中提供了一个设置 tag 的例子：

```
@RequestMapping(value = "/echo-rest/{str}", method = RequestMethod.GET)
public String rest(@PathVariable String str, @RequestParam String user) {
    TsfContext.putTag("user", user);
    return restTemplate.getForObject("http://provider-demo/echo/" + str, String.class);
}
```

准备工作

开始实践服务限流功能前，请确保已完成了 SDK 下载。

快速上手

如果用户要对某个服务开启限流的能力，即对调用它的请求做限流，可以按下面的步骤打开限流开关。

1. 向工程中添加依赖。在 `pom.xml` 中添加以下代码：

```
<dependency>
 <groupId>com.tencent.tsf</groupId>
 <artifactId>spring-cloud-tsf-ratelimit</artifactId>
 <version><!-- 指定版本号 --></version>
</dependency>
```

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

1. 向 Application 类中添加注解 `@EnableTsfRateLimit`：

```
// 下面省略了无关的代码
import org.springframework.tsf.ratelimit.annotation.EnableTsfRateLimit;
@SpringBootApplication
@EnableTsfRateLimit
public class ProviderApplication {
 public static void main(String[] args) {
     SpringApplication.run(ProviderApplication.class, args);
 }
}
```

此时您已经对服务开启了限流功能，任何到达的请求都会被限流模块处理。如果该服务上的配额已经消耗完，会对请求返回 HTTP 429 Too Many Requests；否则会正常放行。

准备工作

开始实践服务路由功能前，请确保已完成了 SDK 下载。

快速上手

使用服务路由功能前，您需要在 `pom.xml` 中添加路由依赖项，同时在代码中使用路由开关注解。

1. 向工程中添加依赖。在 `pom.xml` 中添加以下代码：

```
<dependency>
 <groupId>com.tencent.tsf</groupId>
 <artifactId>spring-cloud-tsf-route</artifactId>
 <version><!-- 指定版本号 --></version>
</dependency>
```

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

2. 向 Application 类中添加注解 `@EnableTsfRoute`：

```
// 下面省略了无关的代码
import org.springframework.tsf.ratelimit.annotation.EnableTsfRateLimit;
@SpringBootApplication
@EnableTsfRoute
public class ProviderApplication {
 public static void main(String[] args) {
     SpringApplication.run(ProviderApplication.class, args);
 }
}
```

元数据类别

TSF 提供两种类型的元数据供开发者在代码中进行设置：

类型	特点
标签信息（Tags）	可设置传递性，仅支持 key-value 数据结构，key 和 value 均为字符串类型。
辅助信息（CustomMetadata）	仅供展示，不支持筛选，不具备传递性。

场景说明：

- **标签信息** ：用于信息分类，使用场景包括：
    - 服务鉴权：被调方通过标签来决定是否提供服务。
    - 服务路由：通过标签来判断应该访问什么服务，可用于实现金丝雀发布等。
    - 调用链：可用于调用链的筛选和附带业务信息。
- **辅助信息** （CustomMetadata）：仅供展示，不支持筛选，不具备传递性。

元数据的传递性

以调用关系 A ≥ B ≥ C，说明传递性的概念：

- 可传递（Transitive）：需要传递的标签，在整条链路都传递，即用户在 A 设置的标签，会传递到 B 再传递到 C。
- 不可传递：不需要传递的标签，即用户在 A 设置的标签，会传递到 B，但是不会传递到 C。

标签信息支持允许用户设置是否支持传递，**辅助信息**不支持传递。 不同的标签可以设置不同传递性，比如一些业务场景：

- `userid` 标签是需要传递的：
    - 可以作为整条调用链上的服务的 上下文信息。
    - 可以实现按 uin 区分的服务路由，比如 A、B、C 三个服务同时做滚动发布，那么可以让一批 uin 都走新版本的 A、B、C 服务，其他用户走老版本。
- `level=高级会员` 这种标签，很可能就不需要在调用间传递下去。

使用元数据

依赖项

在 `pom.xml` 中添加依赖项：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-core</artifactId>
    <version><!-- 指定版本号 --></version>
</dependency>
```

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

接口

```
public enum ControlFlag {
    TRANSITIVE     // 表示标签要传递下去，默认不启用
    NOT_IN_AUTH    // 表示标签不被使用在服务鉴权，默认是被使用的
    NOT_IN_ROUTE   // 表示标签不被使用在服务路由，默认是被使用的
    NOT_IN_SLEUTH  // 表示标签不被使用在调用链，默认是被使用的
}

public class TsfContext {
    /**
     * 设置多个 Tag。如果有某个 Tag 之前已经被设置过，那么它的值会被覆盖。
     */
    public static void putTags(Map<String, String> tagMap, TagControlFlag... flags) {}

    /**
     * 设置 Tag。如果该 key 之前已经被设置过，那么它的值会被覆盖。
     */
    public static void putTag(String key, String value, TagControlFlag... flags) {}
}
```

场景1：设置鉴权 Tag

TSF 提供的 Demo `consumer-demo/src/main/java/com/tsf/demo/consumer/Controller.java` 中设置了键为 user，请求参数作为值的 Tag。Tag 鉴权的具体使用方法可参考 《服务鉴权》。

```
@RequestMapping(value = "/echo-rest/{str}", method = RequestMethod.GET)
public String rest(@PathVariable String str, @RequestParam String user) {
    TsfContext.putTag("user", user);
    return restTemplate.getForObject("http://provider-demo/echo/" + str, String.class);
}
```

场景2：设置调用链 Tag

设置 `user=12345678` 的标签，用户可以在控制台 **调用链查询界面** 通过标签 user=12345678 来检索调用链。调用链标签的具体使用方法参考 《调用链》。

```
TsfContext.putTag("user", "12345678", TRANSITIVE);
```

Tag 的长度限制

用户传递到下流的 Tag （包含从上流带过来的有传递性的 Tag），数量上限为16个。Key 的长度上限为 UTF-8 编码后32字节，value 的长度上限为 UTF-8 编码后128字节。

为了节约用户的开发成本和提升使用效率，TSF 提供了 Spring Cloud 全链路跟踪的组件。用户只需要在代码中配置组件依赖，即可直接使用 TSF 的全链路跟踪功能，无需关心日志采集、分析、存储等过程。 用户仅需要安装了对应的依赖包及添加依赖项即可，无须其他配置。

准备工作

开始实践调用链功能前，请确保已完成了 SDK 下载。

依赖项

添加 pom.xml 依赖：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-sleuth</artifactId>
    <version><!-- 指定版本号 --></version>
</dependency>
```

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

标签与自定义元数据

调用链支持用户在代码中设置标签（Tag） 和自定义元数据（CustomMetada），分别用于调用链的筛选和附带业务信息。

接口定义：

```
public class TsfContext {
    /**
     * 设置多个 Tag。如果有某个 Tag 之前已经被设置过，那么它的值会被覆盖。
     */
    public static void putTags(Map<String, String> tagMap, TagControlFlag... flags) {}

    /**
     * 设置 Tag。如果该 key 之前已经被设置过，那么它的值会被覆盖。
     */
    public static void putTag(String key, String value, TagControlFlag... flags) {}

    /**
     * 设置 CustomMetadata。
     */
    public static void putCustomMetadata(Object customMetadata)
}
```

TSF 框架在微服务注册时，会自动收集并注册微服务提供的 API 接口，用户可通过 TSF 控制台实时掌握当前微服务提供的 API 情况。API 注册功能基于 OpenApi Specification 3.0 规范注册 API 元数据信息。 用户在查看 API 接口的同时，可查看到 API 出入参数据结构信息。

准备工作

开始实践 API 注册功能前，请确保已完成了 SDK 下载。

添加依赖

在 pom.xml 中添加以下代码：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-swagger</artifactId>
    <version><!-- 指定版本号  --></version>
    <scope>compile</scope>
</dependency>
```

添加依赖包后，TSF API 注册功能即生效。

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

配置选项

API 注册功能基于 Swagger 原生规范实现，提供多个配置以适配 Swagger 不同配置应用场景，可用配置如下表所示：

配置项	类型	必填	默认值	说明
tsf.swagger.enabled	Boolean	否	true	是否开启 TSF A 册功能
tsf.swagger.basePackage	String	否	ApplicationMainClass 所在包路径	注册 API 的扫描 径。 推荐将 ApplicationMain 写在外层 Packa
tsf.swagger.excludePath	String	否	（空）	排除扫描的包路
tsf.swagger.group	String	否	default	swagger docke

代码和示例

- SDK 会自动扫描 API 的 path 和 出入参。
- 如果需要上报 API 的描述，需要 `import io.swagger.annotations.ApiOperation;` ，同时在 API 上加上注解 `@ApiOperation(value = "url路径值",notes = "对api资源的描述")` 。如果不关注

API 描述，可以不设置 @ApiOperation。

```
package com.tsf.demo.provider.controller;
// 省略掉部分 import
import io.swagger.annotations.ApiOperation;
import com.tsf.demo.provider.config.ProviderNameConfig;

@RestController
public class ProviderController {
    private static final Logger LOG = LoggerFactory.getLogger(ProviderController.class);

    @Autowired
    private ProviderNameConfig providerNameConfig;
    @ApiOperation(value= "/echo/{param}", notes = "示例描述") // notes 对应 API 描述
    @RequestMapping(value = "/echo/{param}", method = RequestMethod.GET)
    public String echo(@PathVariable String param) {
        LOG.info("provider-demo -- request param: [" + param + "]");
        String result = "request param: " + param + ", response from " +
providerNameConfig.getName();
        LOG.info("provider-demo -- provider config name: [" + providerNameConfig.getName() +
']');
        LOG.info("provider-demo -- response info: [" + result + "]");
        return result;
    }
}
```

前提条件

开始实践服务监控功能前，请确保已完成了 SDK 下载。

操作步骤

Edgware 版本 SDK 相关设置

向工程中添加依赖。在 `pom.xml` 中添加以下代码：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-monitor</artifactId>
    <version><!-- 指定版本号 --></version>
</dependency>
```

> **注意** 仅支持 Spring Cloud SDK 更新日志中的版本号

Finchley 版本 SDK 相关设置

1. 向工程中添加依赖。在 `pom.xml` 中添加以下代码，**依赖的是 `spring-cloud-tsf-sleuth` 而不是 `spring-cloud-tsf-monitor`** 。

   ```
   <dependency>
       <groupId>com.tencent.tsf</groupId>
       <artifactId>spring-cloud-tsf-sleuth</artifactId>
       <version><!-- 指定版本号 --></version>
   </dependency>
   ```

   > **注意** 仅支持 中的版本号

2. 向 Application 类中添加注解 `@EnableTsfMonitor` ：

   ```
   // 下面省略了无关的代码
   import org.springframework.tsf.ratelimit.annotation.EnableTsfMonitor;
   @SpringBootApplication
   @EnableTsfMonitor
   public class ProviderApplication {
       public static void main(String[] args) {
           SpringApplication.run(ProviderApplication.class, args);
       }
   }
   ```

开发前准备

在执行安装脚本之前，请确保您的机器上已经安装了 Java 和 Maven。

1. 安装 Java

1.1 检查 Java 安装

打开终端，执行如下命令：

```
java -version
```

如果输出 Java 版本号，说明 Java 安装成功；如果没有安装 Java，请 下载安装 Java 软件开发套件（JDK）。

1.2 设置 Java 环境

设置 `JAVA_HOME` 环境变量，并指向您机器上的 Java 安装目录。 以 Java 1.6.0_21 版本为例，操作系统的输出如下：

操作系统	输出
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA_HOME=/usr/local/java-current
Mac OSX	export JAVA_HOME=/Library/Java/Home

将 Java 编译器地址添加到系统路径中。

操作系统	输出
Windows	将字符串“;C:\Program Files\Java\jdk1.6.0_21\bin”添加到系统变量“Path”的末尾
Linux	export PATH=PATH:JAVA_HOME/bin/
Mac OSX	not required

使用上面提到的 **java -version** 命令验证 Java 安装。

2. 安装 Maven

2.1 下载 安装 Maven

参考 Maven 下载。

2.2 设置 MAVEN_HOME 和 PATH 环境变量

- Windows 系统下

```
新建系统变量   MAVEN_HOME  变量值：E:\Maven\apache-maven-3.3.9
编辑系统变量  Path         添加变量值：  ;%MAVEN_HOME%\bin
```

- Linux、macOS 系统下

```
export MAVEN_HOME=/usr/local/maven/apache-maven-3.3.9
export PATH=$MAVEN_HOME/bin:$PATH
```

2.3 验证 Maven 安装

当 Maven 安装完成后， 通过执行如下命令验证 Maven 是否安装成功。

```
mvn --version
```

若出现正常的版本号信息后，说明 Maven 安装成功。

3. Maven 配置 TSF 私服地址

3.1 添加私服配置

找到 Maven 所使用的配置文件，一般在 `~/.m2/settings.xml` 中，在 settings.xml 中加入如下配置：

```
<profiles>
     <profile>
       <id>nexus</id>
       <repositories>
           <repository>
               <id>central</id>
               <url>http://repo1.maven.org/maven2</url>
               <releases>
                   <enabled>true</enabled>
               </releases>
               <snapshots>
                   <enabled>true</enabled>
               </snapshots>
           </repository>
       </repositories>
       <pluginRepositories>
           <pluginRepository>
               <id>central</id>
               <url>http://repo1.maven.org/maven2</url>
               <releases>
                   <enabled>true</enabled>
               </releases>
               <snapshots>
                   <enabled>true</enabled>
               </snapshots>
           </pluginRepository>
       </pluginRepositories>
   </profile>
   <profile>
       <id>qcloud-repo</id>
       <repositories>
           <repository>
               <id>qcloud-central</id>
               <name>qcloud mirror central</name>
               <url>http://mirrors.cloud.tencent.com/nexus/repository/maven-public/</url>
               <snapshots>
                   <enabled>true</enabled>
               </snapshots>
               <releases>
                   <enabled>true</enabled>
               </releases>
           </repository>
           </repositories>
       <pluginRepositories>
           <pluginRepository>
               <id>qcloud-plugin-central</id>
               <url>http://mirrors.cloud.tencent.com/nexus/repository/maven-public/</url>
               <snapshots>
                   <enabled>true</enabled>
               </snapshots>
               <releases>
                   <enabled>true</enabled>
               </releases>
           </pluginRepository>
       </pluginRepositories>
   </profile>
 </profiles>
```

```
 <activeProfiles>
   <activeProfile>nexus</activeProfile>
   <activeProfile>qcloud-repo</activeProfile>
</activeProfiles>
```

setting.xml 样例文件下载>> （鼠标右键另存为链接）

3.2 验证配置是否成功

在命令行执行如下命令 `mvn help:effective-settings` 。

- 查看执行结果，没有错误表明 setting.xml 格式正确。
- profiles 中包含 qcloud-repo ，则表明 qcloud-repo 私服已经加入到 profiles 中；activeProfiles 中包含 qcloud-repo，则表明 qcloud-repo 私服已经激活成功。可以通过 `mvn help:effective-settings | grep 'qcloud-repo'` 命令检查。

  > **注意**：执行正确的 Maven 命令后，如果无法下载 qcloud 相关依赖包，请重启 IDE，或者检查 IDE Maven 相关配置。

安装 SDK

通过 Maven 获取 TSF SDK。在 《Demo 工程概述》 中， `pom.xml` 所在目录执行 `mvn clean package` 即可下载 TSF SDK。

> **注意**：如果无法下载相关依赖，请检查网络是否有防火墙限制。

Spring Cloud Finchley

基于 Spring Cloud Finchley 版本 SDK，支持 spring boot 2.0.x。

1.12.4-Finchley-RELEASE（2019-08-15）

Bug 修复

修复 TSF SDK 依赖的 scheduler 和业务自身的 scheduler 相互影响的问题。 修复 spring-cloud-tsf-route 包路由不生效的问题。 修复 spring-cloud-tsf-ratelimit 包限流不准确问题。

版本建议

支持向后兼容，建议全量升级。

1.12.3-Finchley-RELEASE（2019-05-17）

Bug 修复

修复 Finchley 版本服务调用监控问题。

版本建议

支持向后兼容，建议全量升级。

1.12.2-Finchley-RELEASE（2019-04-22）

Bug 修复

- 修复 Finchley 版本 TSF Route 启动问题。
- 修复 Finchley 版本 Feign HttpClient 调用链问题。

版本建议

支持向后兼容，建议全量升级。

1.12.1-Finchley-RELEASE（2019-03-25）

Bug 修复

修复配置回调功能未生效问题。

版本建议

支持向后兼容，建议全量升级。

1.12.0-Finchley-RELEASE（2019-03-13）

新特性

- 支持自动重注册，服务鉴权/路由/限流策略本地缓存。
- 服务路由支持基于可用区和地域就近访问策略。

优化

- 升级分布式配置监听，精确并减小监听范围，处理更新为空的场景，避免大范围 key 刷新事件。
- 优化分部署配置回调触发逻辑。

Bug 修复

- spring-cloud-commons 升级到1.3.1解决 RetryTemplate 会导致 LoadBalanceInterceptor thread unsafe 问题。
- 修复启用 hystrix 时配置会导致 tsf-route 与 feignbuilder 冲突的问题。

版本建议

支持向后兼容，建议全量升级。

Spring Cloud Edgware

基于 Spring Cloud Edgware 版本 SDK，支持 springboot 1.5.x。

1.12.4-Edgware-RELEASE（2019-08-15）

Bug 修复

修复 TSF SDK 依赖的 scheduler 和业务自身的 scheduler 相互影响的问题。 修复 spring-cloud-tsf-ratelimit 包限流不准确问题。

版本建议

支持向后兼容，建议全量升级。

1.12.2-Edgware-RELEASE（2019-04-22）

Bug 修复

修复 Edgware 版本自定义 tag 问题。

版本建议

支持向后兼容，建议全量升级。

1.12.1-Edgware-RELEASE（2019-03-25）

Bug 修复

修复配置回调功能未生效问题。

版本建议

支持向后兼容，建议全量升级。

1.12.0-Edgware-RELEASE（2019-03-13）

新特性

- 支持自动重注册，服务鉴权/路由/限流策略本地缓存。
- 服务路由支持基于可用区和地域就近访问策略。

优化

- 升级分布式配置监听，精确并减小监听范围，处理更新为空的场景，避免大范围 key 刷新事件。
- 优化分部署配置回调触发逻辑。

Bug 修复

- spring-cloud-commons 升级到1.3.1解决 RetryTemplate 会导致 LoadBalanceInterceptor thread unsafe 问题。
- 修复启用 hystrix 时配置会导致 tsf-route 与 feignbuilder 冲突的问题。

版本建议

支持向后兼容，建议全量升级。

1.10.0-RELEASE（2018-11-12）

新特性

- 服务路由（srping-cloud-tsf-route）：支持服务下单个 API 请求级别的路由。
- 服务限流（spring-cloud-tsf-ratelimite）：支持服务下单个 API 请求级别的限流。
- 日志输出（spring-cloud-tsf-logger）：支持默认日志输出。
- API 注册（spring-cloud-tsf-swagger）：支持服务下 API 信息自动注册，查看 API 出入参请求结构。

Bug 修复

解决 RestTemplate Bean 冲突问题。

升级建议

- 支持向后兼容。
- 新功能建议全量升级。

1.1.1-RELEASE（2018-08-26）

新特性

- 服务限流 （spring-cloud-tsf-rate-limit）：支持针对所有请求、单个服务的请求进行流量控制。
- 服务路由（spring-cloud-tsf-route）：支持基于部署组、系统标签、自定义标签的路由设置。
- 服务鉴权 （spring-cloud-tsf-auth）：支持基于服务名和标签的鉴权设置。
- 配置加密 （spring-cloud-tsf-encrypt）：支持配置加密功能。
- 应用状况监控（spring-cloud-tsf-metrics）：支持查看 Spring Boot 应用的 JVM 内存分布、线程、HTTP Traces、环境变量。
- 调用链 （spring-cloud-tsf-sleuth）：支持在调用链上设置标签和自定义 Metada。

Bug 修复

调用链 SDK 问题修复。

升级建议

全部建议升级。

3. TSF Mesh 开发手册

TSF Service Mesh（以下简称为 TSF Mesh）是一个基础设施层，用于处理服务间的通信。TSF Mesh 是由一系列轻量级的网络代理组成，这些代理（又称 Sidecar）与应用程序部署在一起，而应用程序不感知 Sidecar 的存在。 TSF Mesh 是处于 TCP/IP 之上的一个抽象层。TCP 解决了网络端点间字节传输问题，TSF Mesh 解决服务节点间请求的路由问题。

TSF Mesh 优势

TSF Mesh 具有如下优势：

- 多编程语言应用兼容。
- 业务代码零侵入，代码无须改造。

实现原理

TSF Mesh 可以代理使用云服务器或者容器部署的应用。下面以容器为例说明 TSF Mesh 的实现原理。Sidecar 是 L7 层代理，和服务运行在同一个 Pod 中，与 Pod 共享网络，其中 Sidecar 与服务的关系如下：

- Sidecar 代理服务向注册中心注册服务相关信息，以便其他服务发现自身。
- Sidecar 作为 Pod 内服务的 HTTP 代理，可以自动发现其他服务。

使用场景

TSF Mesh 主要有三种使用场景：

- 仅服务消费者作为 Mesh 应用部署。
- 仅服务提供者作为 Mesh 应用部署。
- 服务消费者和服务提供者均作为 Mesh 应用部署。

场景 1：仅服务消费者作为 Mesh 应用部署

- 服务提供者使用 TSF-Spring Cloud 框架实现，注册到服务注册中心；
- 服务消费者作为 Mesh 应用部署，由 Sidecar 注册到服务注册中心。

场景 2：仅服务提供者作为 Mesh 应用部署

- 服务提供者作为 Mesh 应用部署，由 Sidecar 注册到服务注册中心；
- 服务消费者使用 TSF-Spring Cloud 框架实现，注册到服务注册中心。

场景 3：服务消费者和服务提供者均作为 Mesh 应用部署

- 服务提供者作为 Mesh 应用部署，由 Sidecar 注册到服务注册中心；
- 服务消费者作为 Mesh 应用部署，由 Sidecar 注册到服务注册中心。

虚拟机部署 Demo： tsf_python_vm_demo 容器部署 Demo： tsf_python_docker_demo

Demo 提供了3个 Python 应用，对应的服务名和应用监听端口为：

- user （8089）
- shop （8090）
- promotion （8091）

3 个应用之间的调用关系是：`user -> shop -> promotion`，相互访问时可以用默认的80或者业务的真实端口（对应 Demo 中的 sidecarPort），如 shop 监听8090，user 访问 shop 可以用`shop:80/api/v6/shop/items`或者`shop:8090/api/v6/shop/items`。

> !Mesh 的调用链通过头传递实现。如果用户想要串联整个服务调用关系，需要在访问其他服务时，带上父调用的9个相关调用链头，具体示例如下：

```
// 9个调用链相关的头，具体说明见
(https://www.envoyproxy.io/docs/envoy/v1.8.0/configuration/http_conn_man/headers.html?
highlight=tracing)
traceHeaders = ['x-request-id',
                'x-trace-service',
                'x-ot-span-context',
                'x-client-trace-id',
                'x-b3-traceid',
                'x-b3-spanid',
                'x-b3-parentspanid',
                'x-b3-sampled',
                'x-b3-flags']

// 填充调用链相关的头
def build_trace_headers(handler):
    retHeaders = {}
    for header in traceHeaders:
        if handler.headers.has_key(header):
            header_value = handler.headers.get(header)
            retHeaders[header] = header_value
    return retHeaders

// 访问 shop 服务的端口，使用默认的80，或者 shop 的真实端口8090
sidecarPort = 80
def do_GET(self):
    // 调用shop服务时填充父调用的调用链相关头
    if self.path == '/api/v6/user/create':
        print "headers are %s" % self.headers.keys()
        logger.info("headers are %s" % self.headers.keys())
        headers = common.build_trace_headers(self)
        if common.sendAndVerify("shop", sidecarPort, "/api/v6/shop/items", headers):
            self.send_response(200)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
            msg = {"result":{"userId":"1234", "userName":"vincent"}}
            self.wfile.write(json.dumps(msg))
        else:
            self.send_response(500)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
            msg = {"exception":"Error invoke %s" % "/api/v6/shop/items"}
            self.wfile.write(json.dumps(msg))
```

工程目录

虚拟机工程目录

以 tsf_python_vm_demo 中的 userService 为例说明虚拟机应用工程目录。

- **userService.py 和 common.py**：Python 应用程序
- **start.sh**：启动脚本
- **stop.sh**：停止脚本
- **cmdline**：检查进程是否存活的文件

- **spec.yaml**：服务描述文件，具体解释请参考 《Mesh 开发使用指引》
- **apis 目录**：存放 API 定义的目录， 具体解释请参考《[Mesh 开发使用指引》

其中 star.sh、stop.sh、cmdline 的编写方法请参考 《上传程序包要求》。

容器应用工程目录

以 tsf_python_docker_demo 中的 demo-mesh-user 为例说明容器应用工程目录。

> !您需要在容器启动后通过用户程序的启动脚本拷贝目录，不可以在 Dockerfile 中提前拷贝。

- **Dockerfile**：使用 userService 目录中 start.sh 脚本来启动 Python 应用。
- **userService**目录：基本结构类似 tsf_python_vm_demo 中 userService 目录，但是没有 stop.sh 和 cmdline 文件。
- **start.sh**：应用的启动脚本，user demo 的启动脚本如下：

```
#! /bin/bash
cp /root/app/userService/spec.yaml /opt/tsf/app_config/
cp -r /root/app/userService/apis /opt/tsf/app_config/
cd /root/app/userService/
python ./userService.py 80 1>./logs/user.log 2>&1
```

脚本说明：

- 应用工作目录为 `/root/app/userService/` ，应用日志目录为 `/root/app/userService/logs/user.log` 。
- 第2行：创建 `/opt/tsf/app_config/apis` 目录，并将 `spec.yaml` 文件拷贝到 `/opt/tsf/app_config/` 中。
- 第3行：将 `apis` 目录拷贝到 `/opt/tsf/app_config/` 中。
- 第5行：启动 user 应用。

开发说明

本文将以 Python 应用为例说明如何改造代码来接入 TSF。您不需要修改 Python 服务代码，只需要修改服务间调用的 host。

- 将原来的 IP:Port 替换为服务名。
- 端口使用80或者业务真实的监听端口。
- 其他代码不做修改。

> ?以下代码片段可参考 Demo 工程内 userService.py。

改造前：

```
sidecarPort = 80
if common.sendAndVerify("127.0.0.1", sidecarPort, "/api/v6/shop/items", headers):
            self.send_response(200)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
            msg = {"result":{"userId":"1234", "userName":"vincent"}}
            self.wfile.write(json.dumps(msg))
        else:
            self.send_response(500)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
            msg = {"exception":"Error invoke %s" % "/api/v6/shop/items"}
            self.wfile.write(json.dumps(msg))
```

改造后：

```
sidecarPort = 80
if common.sendAndVerify("shop", sidecarPort, "/api/v6/shop/items", headers):
            self.send_response(200)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
            msg = {"result":{"userId":"1234", "userName":"vincent"}}
            self.wfile.write(json.dumps(msg))
        else:
            self.send_response(500)
            self.send_header('Content-type', 'application/json')
            self.end_headers()
            msg = {"exception":"Error invoke %s" % "/api/v6/shop/items"}
            self.wfile.write(json.dumps(msg))
```

可以看到，代码行中除了**访问方式发生变化（`127.0.0.1` 变为 `shop`）**外，其他都不需要改动。

服务定义和注册（必选）

如果是虚拟机部署，需要在应用程序所在目录中设置创建 `spec.yaml` 文件；**如果是容器部署**，需要在应用启动时，在 `/opt/tsf/app_config` 下写入 `spec.yaml` 文件，该文件用于描述服务信息。Sidecar 会通过服务描述文件将服务注册到服务注册中心。 spec.yaml 格式如下：

```
apiVersion: v1
kind: Application
spec:
  services:
  - name: user # 服务名
    ports:
    - targetPort: 8091 # 服务监听端口
      protocol: http # 目前仅支持 http
    healthCheck:
      path: /health # 健康检查 URL
```

> 注意：
>
> - healthCheck 是健康检查的接口，请确认本地调用 `curl -i -H 'Host: local-service' {ip}:{Port}/health` 能返回200。
> - `Host: local-service` 是代理加的 header，业务如果对 Host 有检查（如 nginx 配置的 server_name），则需将 local-service 加到白名单。

API 定义和上报（可选）

TSF 支持 Mesh 应用 API 上报功能，用于 API 级别的服务治理，如路由、鉴权和限流等，不需要可以跳过。 **如果是虚拟机部署**，需要在应用程序所在目录中创建 `apis` 目录；**如果是容器部署**，需要在 `/opt/tsf/app_config` 下创建 `apis` 目录，该目录放置服务的 API 定义。 一个服务对应一个 yaml 文件，文件名即服务名，如 petstore 服务对应 petstore.yaml 配置。API 遵循 OPENAPI 3.0 规范。配置文件遵循 样例参考。user.yml 的 API 定义如下：

```
openapi: 3.0.0
info:
  version: "1.0.0"
  title: user service
paths:
  /api/v6/user/create:
    get:
      responses:
        '200':
           description: OK
        '401':
           description: Unauthorized
        '402':
           description: Forbidden
        '403':
           description: Not Found
  /api/v6/user/account/query:
    get:
      responses:
        '200':
           description: OK
        '401':
           description: Unauthorized
        '402':
           description: Forbidden
        '403':
           description: Not Found
  /health:
    get:
      responses:
        '200':
           description: OK
        '401':
           description: Unauthorized
        '402':
           description: Forbidden
        '403':
           description: Not Found
```

设置自定义标签 （可选）

Mesh 支持通过 HTTP Header 设置自定义标签（标签可用于服务治理，参考服务治理相关文档）。 以 Python 应用为例说明如何设置自定义标签。

```
>>> import requests
>>> url = 'https://api.github.com/some/endpoint'
>>> headers = {'custom-key': 'custom-value'}
>>> r = requests.get(url, headers=headers)
```

> ?以上示例已经在依赖机器上安装了 requests 库。

调用链 Header 传递（可选）

要实现 Mesh 应用调用链和服务依赖拓扑功能，需要在请求中带上 9 个相关 header。

```
// 9个调用链相关的头，具体说明见
(https://www.envoyproxy.io/docs/envoy/v1.8.0/configuration/http_conn_man/headers.html?
highlight=tracing)
traceHeaders = ['x-request-id',
                'x-trace-service',
                'x-ot-span-context',
                'x-client-trace-id',
                'x-b3-traceid',
                'x-b3-spanid',
                'x-b3-parentspanid',
                'x-b3-sampled',
                'x-b3-flags']
```

具体使用方法参考文档 《Mesh Demo 介绍》。

准备工作

1. 下载 TSF 提供的 Demo（该步骤预计耗时1min）。
2. 在 TSF 控制台上已创建容器集群并添加节点，详情参考 《集群》（对于未创建容器集群和添加节点的用户，该步骤预计耗时10min）。
3. 用户的开发机上已安装 `docker` 环境（用于推送镜像到镜像仓库。对于本地无 `docker` 环境的用户，该步骤预计耗时20min）。

一、创建并部署 Mesh 应用

1. 创建应用

1.1 登录 TSF 控制台。 1.2 在左侧导航，单击【应用管理】，进入应用列表页。 1.3 在应用列表页，单击【新建应用】，并填写以下信息：

- 应用名：填写应用名称
- 部署方式：选择**容器部署**
- 应用类型：选择 **Mesh 应用**

1.4 单击【提交】按钮，完成应用创建。

2. 将镜像推送到仓库

2.1 在左侧导航，单击【镜像仓库】，进入镜像列表页。首次使用时，您需要设置镜像仓库密码（该密码与腾讯云官网账号密码独立）。 2.2 在镜像列表页，单击【应用管理】 > 【ID/应用名】 > 【镜像】，单击【使用指引】按钮，根据指引中的命令将 Python demo 应用的镜像推送到镜像仓库中（详请参见 《镜像仓库使用指引》）。

3. 创建部署组

在 TSF 平台上，使用部署组来部署应用，用户需要先创建部署组，再执行部署应用操作。 3.1 在应用详情页内，单击【部署组】，进入部署组列表页。 3.2 在部署组列表页，单击【新建部署组】按钮，并填写以下信息：

- 部署组名：填写部署组名称
- 集群：选择应用将部署的集群
- 命名空间：选择命名空间属性
- 实例资源保留：0.5 核，512 MiB
- 实例数：1

3.3 设置访问设置、更新方式、日志配置项。

- 网络访问方式：NodePort

- 端口协议：协议选择 TCP，容器端口和服务端口都填写 8089
- 更新方式：快速更新
- 日志配置项：选择无

3.4 单击【提交】按钮。

4. 部署应用

4.1 在部署组列表页，单击部署组右侧的【部署应用】。 4.2 配置部署相关信息，选择在 **步骤2** 中推送到仓库的镜像版本。 4.3 单击【提交】按钮，直到部署组状态变为运行中，则表示应用部署成功。

二、查看服务是否注册成功

1. 在左侧导航栏，单击【服务治理】，进入服务列表页，查看服务是否注册成功。如果注册成功，服务显示「在线」状态。
2. 在服务列表页，单击服务 ID，进入服务详情页。单击【API 列表】标签页，可以查看上报的 API 定义。

三、验证服务调用

使用同样的步骤一和步骤二部署 user、shop 和 promotion 三个应用。user、shop、promotion 三个服务的接口间调用关系如下：

1. 触发 user 服务调用 shop 和 promotion 服务

为了验证 user 服务能通过服务名来调用 shop 服务，需要用户通过以下三种方式来触发 user 服务的接口调用：

- **负载均衡 IP + 服务端口**：如果部署组在部署时，选择了公网访问方式，可以通过**负载均衡 IP + 服务端口**来访问 `user` 服务的 `/api/v6/user/account/query` 接口。
- **节点IP + NodePort**： 如果部署组在部署时，选择了 NodePort 访问方式，可以通过**节点IP + NodePort**来访问 `user` 服务的 `/api/v6/user/account/query` 接口。其中 `节点IP` 为集群中任一节点的内网IP， `NodePort` 可以在部署组的基本信息页面被查看。用户首先登录到集群的工作节点，然后执行如下命令：

```
curl -XGET <节点IP>:<NodePort>/api/v6/user/account/query
```

2. 在控制台验证服务之间是否调用

可以通过以下两种方式验证服务是否成功被 Sidecar 代理注册到注册中心，同时验证服务之间是否成功地进行了调用。

- **服务治理**界面：选择集群和命名空间后，如果服务列表中的服务状态为**在线**或**单点在线**，表示服务被代理注册成功。如果服务提供者的请求量大于0，表示服务提供者被服务消费者请求成功。
- **依赖拓扑**界面：选择集群和命名空间后，调整时间范围，使其覆盖服务运行期间的时间范围，正常情况下，将出现服务之间相互依赖的界面。

准备工作

1. 下载 TSF 提供的 Demo（该步骤预计耗时1min）。
2. 在 TSF 控制台上已创建容器集群并添加节点，参考 《集群》。对于未创建容器集群和添加节点的用户（该步骤预计耗时10min）。
3. 主机上已安装应用运行的环境（如 Python 应用的相关依赖等，该步骤预计耗时根据运行环境的复杂度有所不同）。

一、创建并部署 Mesh 应用

1. 创建应用

1.1 登录 TSF 控制台。 1.2 在左侧导航栏单击 应用管理，进入应用管理列表页。 1.3 单击【新建应用】，并填写以下信息：

- 应用名：填写应用名称
- 部署方式：选择**虚拟机部署**
- 应用类型：选择 **Mesh 应用**

1.4 单击【提交】按钮，完成应用创建。

2. 上传程序包

2.1 在左侧导航栏单击 应用管理，选择某一应用的【ID/应用名】，进入应用服务详情页。 2.2 在应用服务详情页单击【程序包管理】标签页。 2.3 在标签页单击【上传程序包】按钮，选择程序包，填写程序包相关信息。 2.4 单击【提交】按钮，完成上传。

3. 创建部署组

可参考 《虚拟机应用部署组》中关于**创建部署组**的流程描述，此处不再赘述。

4. 部署应用

可参考 《虚拟机应用部署组》中关于**部署应用**的流程描述，此处不再赘述。

二、查看服务是否注册成功

1. 在左侧导航栏单击 服务治理，进入服务列表页，查看服务是否注册成功。如果成功，服务显示在线状态。
2. 在服务列表页单击服务 ID，进入服务详情页。单击【API 列表】标签页，可以查看上报的 API 定义。

三、验证服务调用

使用同样的步骤一和步骤二部署 user、shop 和 promotion 三个应用。user、shop、promotion 三个服务的接口间调用关系如下：

用户可以登录容器集群下的任一机器，然后通过 `curl` 命令验证 user 服务是否健康，以及触发 user 服务调用 shop 和 promotion 服务。

1. 登录服务器验证服务间调用

为了验证 user 服务能通过服务名来调用 shop 服务，需要用户通过以下几种方式来触发 user 服务的接口调用：

- 登录 user 所在云服务器，执行如下 `curl` 命令调用 user 服务接口。

```
curl localhost:<user端口>/api/v6/user/account/query
```

- 登录 user 所在云服务器，执行如下 `curl` 命令调用 shop 服务接口（注意使用服务名来调用）。

```
curl shop:<shop端口>/api/v6/shop/order
```

2. 在控制台验证服务之间是否调用

可以通过以下两种方式验证服务是否成功被 Sidecar 代理注册到注册中心，同时服务之间是否成功地进行了调用。

- **服务治理**界面：选择集群和命名空间后，如果服务列表中的服务状态为**在线**或**单点在线**，表示服务被代理注册成功。如果服务提供者的请求量大于0，表示服务提供者被服务消费者请求成功。
- **依赖拓扑**界面：选择集群和命名空间后，调整时间范围，使其覆盖服务运行期间的时间范围，正常情况下，将出现服务之间相互依赖的界面。

Service Mesh 微服务架构的核心是在用户的服务侧同机（虚拟机）或同 Pod（容器）部署一个网络代理来让用户实现无侵入的接入微服务。因为代理是以本地额外的服务实现的，本地应用无感知，为了快速定位出现的问题，代理测提供了大量的 HTTP 运维接口。

代理的组件

本地代理分为以下三个组件：

- **pilot-agent** 管理整个代理侧环境，如 iptables 规则的更新、本地应用和服务的配置管理，同时负责 mesh-dns 和 envoy 组件的生命周期管理。
- **Mesh-DNS** 托管本地的 DNS 请求，将 Mesh 结构内的流量导入本地代理。
- **Envoy** 负责服务发现和路由，HTTP 请求的负载均衡等，负责处理流入本地服务和从本地服务流出的所有流量。

iptables 规则

为了将流入和流出的流量导入到本地代理中，TSF Mesh 使用了 iptables 作流量转发，一般规则如下（虚拟机环境在本地，容器环境在 sidecar 容器中）：

iptables -t nat -L -n

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
ISTIO_INBOUND  tcp  --  0.0.0.0/0            0.0.0.0/0

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ISTIO_OUTPUT  all  --  0.0.0.0/0           0.0.0.0/0
DNS_OUTPUT  all  --  0.0.0.0/0            0.0.0.0/0

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain DNS_OUTPUT (1 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0            owner UID match 1000
DNAT       udp  --  0.0.0.0/0            0.0.0.0/0            udp dpt:53 to:127.0.0.1:55354
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:53 to:127.0.0.1:55354

Chain ISTIO_INBOUND (1 references)
target     prot opt source               destination
ISTIO_REDIRECT  tcp  --  0.0.0.0/0            9.77.7.28            tcp dpt:8089

Chain ISTIO_OUTPUT (1 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0            owner UID match 1000
DNAT       tcp  --  0.0.0.0/0            {特定ip}        to:9.77.7.28:15001

Chain ISTIO_REDIRECT (1 references)
target     prot opt source               destination
REDIRECT   tcp  --  0.0.0.0/0            0.0.0.0/0            redir ports 15001
```

DNS_OUTPUT

- 托管本地的 DNS 请求到 mesh-dns 进程。
- 第一条 RETURN 规则非常重要，不再托管从 mesh-dns 转发出来的 DNS 请求。
- 后面两条是托管本地的53端口（即 DNS 原生请求端口）的流量。

ISTIO_INBOUND

- 托管访问本地服务的流量。
- 可以看到托管的流量非常细粒度，只托管注册到 Mesh 框架的 `9.77.7.28:8089` 的流量。

ISTIO_OUTPUT

- 托管从本地流出的流量。
- 第一条 RETURN 规则非常重要，不再托管从代理转发出来的请求。
- 第二条是将访问{特定 IP} - 从 DNS 中获得的流量导入到本地的代理中（即引用的 ISTIO_REDIRECT 规则）。

本地接口

如果在本地调用 `netstat -lntp | grep 127.0.0.1`，会出现以下三个 listen 服务，分别是各个本地组件提供的运维接口的 IP 和 port。

```
tcp        0      0 127.0.0.1:15020         0.0.0.0:*               LISTEN      5168/pilot-
agent
tcp        0      0 127.0.0.1:15021         0.0.0.0:*               LISTEN      5261/mesh-dns
tcp        0      0 127.0.0.1:15000         0.0.0.0:*               LISTEN      5266/envoy
```

pilot-agent 运维接口

curl 127.0.0.1:15020/help

```
admin commands are:
    GET /health: print out the health info for data-plane
    GET /config_dump/{component}: print out the configuration of the component, component can
be pilot-agent/envoy/mesh-dns
    GET /help: print out list of admin commands
    GET /config/agent: print out the pilot-agent configuration
    GET /config/services: print out the services info
    GET /config/global: print out the global mesh configuration
    GET /config/envoy: print out the envoy startup configuration
    GET /version: print out the pilot-agent version
    GET /version/{component}: print out the version of the component, component can be pilot-
agent/envoy/mesh-dns
    GET /latest_error: print out the latest error info
    GET /status: print out the status, result could be <INIT>, <CONFIG_READY>, <FLOW_TAKEOVER>,
<SERVICE_REGISTERED>
    GET /epoch/{component}: print out the component restart epoch, component can be envoy/mesh-
dns
    POST /hot_restart/{component}: hot restart the component process, component can be
envoy/mesh-dns
    PUT /update: update the component
    PUT /logging/{scope}/{level}: dynamic change logging level, scope can be
healthz/admin/ads/default/model, level can be debug/info/warn/error/none
```

其中主要的接口如下：

- /status：查看本地各个组件的状态。
- /health：查看本地各个组件的健康信息。
- /version：查看本地代理服务的版本。

Mesh-DNS 运维接口

curl 127.0.0.1:15021/help

```
admin commands are:
    GET /health: print out the health info for mesh-dns
    GET /config_dump: print out all of the mesh-dns runtime configs
    GET /latest_error: print out the latest error info
    GET /help: print out list of admin commands
    GET /version: print out version info
    GET /status: print out status info
    PUT /logging/{scope}/{level}: dynamic change logging level, scope can be
admin/default/healthz/model, level can be error/none/debug/info/warn
```

其中主要的接口如下： /config_dump：查看本地托管的 DNS 信息。

Envoy 运维接口

curl 127.0.0.1:15000/help

```
admin commands are:
  /: Admin home page
  /certs: print certs on machine
  /clusters: upstream cluster status
  /config_dump: dump current Envoy configs (experimental)
  /cpuprofiler: enable/disable the CPU profiler
  /healthcheck/fail: cause the server to fail health checks
  /healthcheck/ok: cause the server to pass health checks
  /help: print out list of admin commands
  /hot_restart_version: print the hot restart compatibility version
  /listeners: print listener addresses
  /logging: query/change logging levels
  /quitquitquit: exit the server
  /reset_counters: reset all counters to zero
  /runtime: print runtime values
  /runtime_modify: modify runtime values
  /server_info: print server version/status information
  /stats: print server stats
  /stats/prometheus: print server stats in prometheus format
```

其中主要的接口如下：

- /clusters：查看各个部署组下的实例信息和健康信息，例如查看本地服务节点的健康信息 `curl -s 127.0.0.1:15000/clusters | grep "in#"` 。
- /config_dump：将服务路由信息打印出来，一般调用 `curl -s 127.0.0.1:15000/config_dump -o 1.json` ，打开1.json即可查看路由信息。

4. Dubbo 应用接入

获取 Demo

- 基于 Alibaba Dubbo 版本 SDK 的 Demo 下载
- 基于 Apache Dubbo 版本 SDK 的 Demo 下载

工程目录

`tsf-dubbo-xxx-demo` 的工程目录如下：

```
|- consumer-demo
|- provider-demo
|- pom.xml
```

其中 consumer-demo 表示服务消费者， provider-demo 表示服务提供者，pom.xml 中定义了工程需要的依赖包（以下以基于 Alibaba Dubbo 版本 SDK 举例说明）：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
    </parent>

    <artifactId>provider-demo</artifactId>
    <version>1.1.6-alibaba-RELEASE</version>
    <packaging>jar</packaging>
    <name>provider-demo</name>

    <properties>
        <start-class>com.tsf.demo.provider.ProviderApplication</start-class>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.6</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.tencent.tsf</groupId>
            <artifactId>tsf-dubbo</artifactId>
            <version>1.1.6-alibaba-RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

关于 Maven 环境安装，请参考 Spring Cloud SDK 下载时的 Maven 配置。

操作场景

TSF 为其他应用提供服务注册中心，Dubbo 应用可通过依赖 jar 包的方式接入该项服务。本文档介绍 Dubbo 应用从接入TSF 到部署应用的操作方法及相关注意事项。

操作步骤

1. Maven 环境安装

详细操作请参考 Maven 安装。

2. 注册中心配置

Dubbo 官网 Demo：

```
<dubbo:registry address="multicast://224.5.6.7:1234"/>
```

TSF Demo（**注册中心地址使用注册中心IP和端口替换**）：

```
<dubbo:registry address="tsfconsul://注册中心地址:端口"/>
```

> ?协议名为 tsfconsul。

3. 添加依赖

根据业务使用的对应的 TSF Dubbo 版本 SDK 如下：

- TSF Alibaba Dubbo 版本的 SDK：

```
<dependency>
<groupId>com.tencent.tsf</groupId>
<artifactId>tsf-dubbo</artifactId>
<!-- 修改为对应的版本号 -->
<version>1.1.6-alibaba-RELEASE</version>
</dependency>
```

- TSF Apache Dubbo 版本的 SDK：

```
<dependency>
<groupId>com.tencent.tsf</groupId>
<artifactId>tsf-dubbo</artifactId>
<!-- 修改为对应的版本号 -->
<version>1.1.6-apache-RELEASE</version>
</dependency>
```

> ?当当网的 Dubbox 的部分功能可能不支持。

4. 打包 FATJAR

和 Spring Boot 结合的时候，您可以通过 **spring-boot-maven-plugin** 构建一个包含所有依赖的 jar 包（FatJar），执行命令 `mvn clean package`。 详情请参考 《Dubbo Demo 工程概述》。

Dubbo 兼容说明

- TSF 提供服务注册中心，Dubbo 应用通过依赖 jar 包的方式使用。
- TSF 支持 Dubbo 应用的 Dubbo 服务注册、Dubbo 服务调用、调用链、监控。
- Dubbo 应用的其他能力（如 filter 机制），可以继续使用。
- TSF 平台提供的服务限流、鉴权、路由功能目前只支持基于 Spring Cloud 和 Service Mesh 框架开发的应用。

常见错误

部分包对版本有要求，如果发生**包冲突**，请尝试主动依赖以下版本：

```
 <dependency>
  <groupId>com.ecwid.consul</groupId>
  <artifactId>consul-api</artifactId>
  <version>1.4.2</version>
</dependency>

<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave</artifactId>
  <version>5.4.3</version>
</dependency>s
```

5. 快速入门

步骤1：创建容器集群

新建容器集群

操作步骤参考手册中资源中心《集群》说明文档。

集群初始化及导入云主机

操作步骤参考手册中资源中心《集群》说明文档。

步骤2：创建容器应用

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【应用管理】。
3. 在应用列表上方，单击【新建应用】。
4. 设置应用信息后，单击【提交】按钮。
    - 部署方式：选择容器部署
    - 应用类型：选择普通应用

步骤3：创建部署组

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【部署组】进入部署组详情页。
3. 选择部署组所属集群和所属命名空间。
4. 在部署组列表上方，单击【新建部署组】按钮。
5. 设置部署组相关信息。
    - **部署组名称**：部署组的名称，不超过60个字符。
    - **关联应用**：选择部署组关联的应用。
    - **实例资源保留**：分配给单个实例使用的 CPU 核数和内存资源 Request 值。
    - **实例数量**：一个实例由相关的一个容器构成，可单击 + 和 - 控制实例数量。实例数和实例资源限制的乘积不能超过集群剩余的可用资源。
    - **网络访问方式**： 网络的访问方式决定了部署组内应用的网络属性，不同访问方式的应用可以提供不同网络能力。参阅 《容器部署组访问方式》。
        - **NodePort**：访问方式不绑定外网负载均衡，在集群内所有主机自动分配 NodePort 端口，可通过**集群内任意主机IP + NodePort**访问该服务。
        - **集群内访问**：访问方式不绑定外网负载均衡，该服务只能在集群内部访问。
    - **端口映射**：容器端口与服务端口的映射关系。
        - 协议：TCP、UDP。
        - 容器端口：容器内应用程序监听的端口。
        - 服务端口：集群外通过负载均衡域名或 IP+服务端口访问服务；集群内通过服务名+服务端口访问服务。建议和容器端口保持一致。

- **更新方式**：选择部署的更新方式。
  - 快速更新：直接关闭所有实例，启动相同数量的新实例。
  - 滚动更新：对实例进行逐个更新，这种方式可以让您不中断业务实现对服务的更新。
- **日志配置项**：选择日志配置项，用于采集应用的业务日志数据。

步骤4：创建镜像和推送镜像到仓库

1. 在应用列表中，单击在 **步骤2** 中创建的应用 ID/应用名。
2. 单击**镜像**标签页，TSF 会针对每个容器应用创建一个名为 `tsf_<账号ID>/<应用名>` 的镜像仓库。
3. 使用 Dockerfile 创建镜像，参考 《制作容器镜像》。

   > 注意：
   >
   > - 制作镜像操作前，请确保执行命令的机器上已安装 `docker` 。
   > - 如果用户需要输入两次密码，首次为 sudo 密码，第二次为镜像仓库登录密码。

4. 单击【使用指引】按钮，获取登录镜像仓库的命令。

   ```
   sudo docker login --username=<账号 ID>  <仓库服务器地址>
   ```

5. 在使用指引中，获取给镜像打 tag 的命令。

   ```
   sudo docker tag [ImageId] 》<仓库服务器地址>/tsf_<账号ID>/<应用名>:[tag]
   ```

6. 在使用指引中，获取推送镜像到仓库的命令。其中，**tag** 和步骤 3 相同。

   ```
   sudo docker push <仓库服务器地址>/tsf_<账号ID>/<应用名>:[tag]
   ```

7. 推送完成后，在镜像版本列表可查看镜像仓库中的镜像。

步骤5：部署应用

1. 在部署组页面中，单击目标部署组右侧操作栏的【部署应用】按钮。
2. 设置部署相关信息。
   - **选择镜像**：选择要部署的镜像。
   - **单实例资源保留**：分配给单个实例使用的 CPU 核数和内存资源 Request 值。
   - **实例数量**：一个实例由相关的一个容器构成，可单击 + 和 - 控制实例数量。实例数和实例资源限制的乘积不能超过集群剩余的可用资源。
   - **启动参数**（选填）：设置 Java 应用的启动参数。
3. 单击【提交】按钮。

步骤1：创建虚拟机集群

新建虚拟机集群

操作步骤参考手册中资源中心《集群》说明文档。

将虚拟机导入集群

操作步骤参考手册中资源中心《集群》说明文档。

步骤2：创建虚拟机应用

1. 在左侧导航栏，单击【应用管理】，进入应用列表。
2. 在应用列表上方单击【新建应用】。
3. 设置应用信息后，单击【提交】按钮。
    - 应用类型：虚拟机应用将部署在云服务器上
    - 部署方式：选择虚拟机部署

步骤3：创建部署组

1. 在 应用列表页，单击新建应用的 ID，进入应用详情页=>部署组标签页。
2. 在应用详情页=>部署组标签页的上方，单击【新建部署组】按钮。
3. 设置部署组相关信息。
    - 部署组名称：部署组的名称，不超过60个字符。
    - 集群：选择希望建立部署组的集群。可以选择上文建立好的集群。
    - 命名空间：选择希望建立部署组的命名空间。可以选择上文中建立好的命名空间。
    - 日志配置项：应用的日志配置项用于指定 TSF 采集应用的日志路径。参考使用文档日志服务章节《日志配置项》。
4. 单击【提交】按钮。

步骤4：在部署组中添加实例

1. 在部署组列表的右侧操作栏，单击【添加实例】。
2. 选择要添加进部署组的云服务器，单击【提交】。
3. 在部署组的详情页，实例列表页面中显示出刚刚添加的云服务器。

步骤5：上传程序包

1. 在左侧导航栏，单击【应用管理】，进入应用管理列表页。
2. 在应用管理列表页 ，单击目标应用的**ID/应用名**，进入应用详情页。
3. 在应用详情页的上方，单击**程序包管理**标签页，单击【上传程序包】按钮。

4. 在**上传程序包**对话框中填写相关参数。
    - 上传程序包：单击【选择文件】，选择编译为 jar 格式的程序包
    - 程序包版本：填写版本号，或单击【用时间戳作为版本号】
    - 备注：填写备注
5. 单击【提交】按钮，程序包上传成功后出现在程序包列表中。

步骤6：部署应用

1. 在部署组列表页的右侧操作栏，单击【部署应用】。
2. 选择程序包后，单击【提交】按钮。
3. 应用部署成功后，部署组中的**已启动/总机器数**数值发生变化。

6. 资源中心

集群是指云资源管理的集合，包含了运行应用的云主机等资源。集群包括虚拟机集群和容器集群两种类型。

集群操作支持新建集群、删除集群、导入云主机、移出云主机等。

新建集群

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击 【集群】。
3. 在集群列表页的左上方，单击【新建集群】。
4. 设置集群的基本信息。
    - **集群名称**：集群名称，不超过60个字符。
    - **集群类型**：选择集群类型
        - 虚拟机集群：虚拟机集群中的主机用于部署虚拟机应用。
        - 容器集群：容器集群中的主机用于部署容器应用。
    - **集群描述**：集群的描述，不超过200个字符。

导入云主机

1. 在云主机列表中，单击操作列的【安装 Agent】。
2. 复制脚本（下图脚本为图片，**不能复制**，用户需要在控制台上对应弹框中单击【复制】）。
3. 登录云主机，粘贴脚本并执行。下图为脚本执行成功后终端最后几条打印日志的截图。
4. 脚本执行成功后，回到云主机列表，稍等片刻，云主机的可用状态会变为"可用"。

移出云主机

> **注意**：只有当节点上没有运行应用时，才可以将节点从集群中移出。停止应用需要到部署组的实例列表页面中操作。

1. 在集群列表页中，单击目标集群的**ID/名称**，进入集群的节点列表页面。
2. 在集群节点列表页面，选择需要移出的云主机，单击操作列的【删除】。
3. 在弹出的提示框中，单击【确定】将云主机移出节点。

删除集群

> **注意**：集群在删除期间，无法对外提供服务，请提前做好准备，以免造成影响。

1. 在集群列表页中，单击目标集群操作列的【删除】。
2. 在弹出的提示框中，单击【确定】删除集群。

TSF 容器功能依赖于底层容器平台。

新建集群

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击 【集群】。
3. 在集群列表页的左上方，单击【新建集群】。
4. 设置集群的基本信息。
5. **集群名称**：集群名称，不超过60个字符。
6. **集群类型**：选择集群类型
    - 虚拟机集群：虚拟机集群中的主机用于部署虚拟机应用。
    - 容器集群：容器集群中的主机用于部署容器应用。
    - **集群描述**：集群的描述，不超过200个字符。

TSF 容器功能使用（灵雀云）

TSF 上新建集群

打开 TSF 控制台，新增一个容器集群，把集群 ID（`cls-xxxxxx`）和集群名记下来。

灵雀上新建集群

这个步骤仅部署控制节点，而 不部署计算节点。计算节点的部署，需要后续在 TSF 上点「导入云主机」进行。

准备至少两台机器用于搭建 K8S 集群。任何加入灵雀集群的机器，无论是 master 还是 node 节点，都 必须 做这些准备工作：

- 需要保证 `/etc/hosts` 中有 `127.0.0.1 localhost` 行。不能有多行 `127.0.0.1` 或者 `::1`；也不要带多个域名，如 `127.0.0.1 localhost localhost.localdomain`
- 主机名改成不带大写、下划线等。用全小写加英文字母。用 `hostname` 命令去修改，而 不是 直接修改 `/etc/hostname` 文件

去灵雀控制台创建集群页：http://{控制台地址}/console/admin/cluster/create-v4 。

集群名称填 TSF 集群 ID，显示名称填 TSF 集群名，集群类型选单点。控制节点选上述集群的任一台机器，用户名、密码、端口填 SSH 的。网络模式选 Flannel Vxlan，CIDR 问一下灵雀同学应该怎么配。

填完点确定，复制好弹出的脚本到机器上运行。如果运行过程中提示无 `sshpass` 命令，可以用系统包管理工具安装之（如 `yum`）。

TSF 上导入云主机

在灵雀上部署完集群的控制节点后，在 TSF 控制台上可以看到集群状态变为运行中。此时可以点击「导入云主机」导入机器。注意：计算节点也需要按上一节所说，修改 `/etc/hosts` 及主机名。导入的机器对应为灵雀 k8s 集群的计算节点。

灵雀上部署监控服务

请灵雀同学在刚新建的集群上部署「监控服务」。切记每个集群都需要搭建单独的监控服务组件，不能混用。TSF 需要使用其接口计算集群剩余资源。

灵雀上新建项目并把集群加入项目

由于灵雀 2.3 版本暂不支持将集群加入已创建的项目中，因此需要为每个集群新建项目。

访问灵雀控制台（http://{控制台地址}/console/admin/projects），新建一个项目，「名称」为 TSF 集群 ID，「显示名称」为 TSF 集群名，「集群」处把刚才建的集群加进去。项目配额全部不限制。

验证

到这里就做完配置了，看看能否正常用 TSF 平台的功能。此后功能验证可参考腾讯云 TSF 官网文档进行。

TSF 容器功能使用（Gaiastack）

TSF 上新建集群

打开 TSF 控制台，新增一个容器集群

集群初始化

不同于公有云上 TSF 集群初始化是在创建集群时在后台自动完成，独立版 TSF 集群初始化需要手动添加 master 节点。Gaia 容器集群需要 2 个 master 节点。

在集群列表页中，单击目标集群的 **ID/集群名** 。在云主机列表标签页，单击左上方的【集群初始化】。在集群初始化对话框中填写 Master 节点登录信息：

- IP : 云主机的 IP 地址，一般填写内网 IP
- SSH 端口：默认为 22
- 密码：root 用户密码

输入完节点登录信息后，单击【检测】按钮，检测填写信息是否正确，如果不正确需要重新填写。

单击【提交】，集群状态变为初始化。集群从初始化状态变为运行中状态需要大概十几分钟。

导入云主机

在集群详情页中单击云主机列表上方的【导入云主机】，在导入云主机对话框中填写云主机的登录信息，操作步骤和集群初始化时基本一致。

单击【提交】，等待几分钟后，云主机状态从准备中变为运行中，说明主机导入成功。

验证

到这里就做完配置了，看看能否正常用 TSF 平台的功能。此后功能验证可参考腾讯云 TSF 官网文档进行。

命名空间（Namespace）是对一组资源和对象的抽象集合。例如可以将开发环境、联调环境和测试环境的服务分别放到不同的命名空间中。 在网络连通性的前提下，同一命名空间内的服务可以相互发现和相互调用。

命名空间分类

命名空间有两种类型：

- 默认命名空间：默认规则是 `<cluster-name>_default`。
- 自定义命名空间：用户可以在集群中按照需要创建命名空间。

跨集群访问

在网络连通性的前提下，同一命名空间内的服务可以相互发现和相互调用。如果要实现**跨集群的服务访问**，有两个前提：

1. 集群内实例网络互通
2. 集群关联相同的命名空间。**多个集群通过命名空间名称（而不是命名空间 ID）作为关联的 key**。用户可以在命名空间一级页面中看到关联的集群信息。

例如：有2个集群分别是 cluster-1 和 cluster-2，两个集群内实例网络互通，并且都关联了命名空间 test-namespace。要实现跨集群访问，需要确保在创建部署组 provider-group 和 consumer-group 时，使用相同的命名空间 test-namespace。

创建命名空间

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【集群】。
3. 在集群列表页，单击集群 ID，进入集群详情页。
4. 在集群详情页，单击【命名空间】。
5. 在命名空间列表左上方，单击【新建命名空间】按钮。
6. 填写命名空间的名称和描述，并单击【提交】按钮。

删除命名空间

> 注意：删除命名空间将销毁命名空间下的所有资源，销毁后所有数据将被清除且不可恢复，清除前将请提前备份数据。

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【集群】。
3. 在集群列表页，单击集群 ID，进入集群详情页。
4. 在集群详情页，单击【命名空间】。
5. 在命名空间列表的操作列，单击【删除】。
6. 单击【确认】按钮，解除集群和命名空间的绑定关系。

7. 应用中心

TSF 应用分为两种类型：

- **虚拟机应用**：通过程序包部署在云服务器上的应用。
- **容器应用**：通过镜像部署在 Docker 容器中的应用。

虚拟机应用和容器应用的对比如下：

应用类型	虚拟机应用	容器应用
应用托管方式	一台云服务器部署一个应用	使用 Docker 部署应用，一台云服务器可以部署多个应用
使用场景	传统部署场景	对容器运行环境需要定制和希望提升资源利用率的场景
集群类型	虚拟机集群	容器集群
部署方式	JAR 包、zip 压缩包、tar.gz 压缩包	镜像
应用举例	Spring Boot、 Dubbo	Spring Boot、 Dubbo、MySQL、WordPress

操作场景

应用管理包括创建应用和删除应用，您可以按照以下步骤进行操作。

操作步骤

创建应用

1. 登录 TSF 控制台。
2. 在左侧导航栏，单击【应用管理】。
3. 在应用列表左上方，单击【新建应用】。
4. 设置应用基本信息：
    - 应用类型：普通应用或者 Mesh 应用
    - 部署方式：选择虚拟机部署或者容器部署组

删除应用

1. 在应用列表的操作列，单击【删除】。
2. 在确认弹框中，单击【确认】完成删除。

> 注意：
>
> - 删除应用会同时删除应用关联的程序包或者镜像。
> - 当应用下有部署组时，无法执行删除操作，需要先删除所有部署组后才能删除应用。

部署组概述

部署组是执行应用批量部署的逻辑概念。一个部署组内包括多个应用实例，每个应用实例上运行相同的应用程序。

部署组的操作包括：创建部署组、删除部署组、部署应用、启动应用、下线应用、应用扩容等。

部署组类型

按照部署组关联的应用类型，部署组分为虚拟机应用部署组和容器应用部署组。参阅 《应用概述》。

虚拟机应用部署组：使用云服务器来部署应用。参阅 《虚拟机应用部署组》。

容器应用部署组：使用 Docker 容器来部署应用。Docker 应用部署时，将在云服务器上创建多个 Docker 容器实例。参阅 《容器应用部署组》。

类型	虚拟机应用部署组	容器应用部署组
实例类型	云服务器	Kubernetes Pod
部署方式	JAR 包、zip 包、.tar.gz 压缩包	镜像
操作	部署应用、停止应用、启动应用、下线实例、扩容	部署应用、停止应用、启动应用、扩缩容

虚拟机部署组的操作包括两种：

- 基本操作：部署组的创建和删除。
- 应用生命周期管理相关操作：添加实例（扩容）、下线实例（缩容）、部署应用、启动应用、停止应用。

基础操作

创建部署组

1. 登录 TSF 控制台。
2. 单击左侧导航栏中的 **部署组**。
3. 在页面顶部选择集群。
4. 单击部署组列表上方的【新建部署组】按钮。
5. 设置部署组相关信息。
    - **部署组名称**：部署组的名称，不超过 60 个字符。
    - **集群**：选择集群。
    - **命名空间**：选择命名空间。
    - **关联应用**：关联应用字段决定了后续程序包来源和应用配置来源。
    - **日志配置项**：指定部署组内实例的业务日志采集规则。如果配置为"无"，将不采集业务日志。
6. 选择实例。选择云主机并导入。
7. 部署应用。选择程序包版本并配置启动参数。
8. 完成。

删除部署组

1. 下线部署组内所有实例，才能执行部署组的删除操作。
2. 单击部署组列表页右侧的【删除】。

应用生命周期管理相关

功能	说明
添加实例	将云服务器添加到部署组中，如果部署组此时已经关联了程序包，将执行部署命令。
下线实例	停止云服务器上的应用，然后将实例从部署组中移除。
部署应用	将应用部署到云服务器上，并执行启动命令。
启动应用	当应用处于停止状态时可以启动应用。
停止应用	当应用处于运行中状态时可以停止应用。

添加实例

1. 单击部署组列表右侧的【添加实例】。
2. 选择要添加进部署组的云服务器，单击【提交】。
3. 在部署组的实例列表页面中显示出刚刚添加的云服务器。

下线实例

1. 单击目标部署组。
2. 单击实例列表右侧的【下线实例】按钮。
3. 在弹出的确认页面中，单击【提交】。

部署应用

1. 单击部署组列表页右侧的【部署应用】。
2. 选择程序包后，设置启动参数（选填）后，单击【提交】按钮。
3. 应用部署成功后，部署组中的 **已启动/总机器数** 数值发生变化。

启动应用

1. 单击部署组列表页右侧的【更多】中的【启动应用】。
2. 在弹出的确认页面中，单击【确认】。

停止应用

1. 单击目标部署组。
2. 单击实例列表右侧的【停止应用】按钮。
3. 在弹出的确认页面中，单击【确认】。

操作场景

该任务指导您通过 TSF 控制台对容器应用部署组进行操作，包括：创建部署组、删除部署组、部署应用、启动应用、应用扩缩容等。

操作步骤

创建部署组

1. 登录 TSF 控制台。
2. 单击左侧导航栏中的 【部署组】。
3. 选择部署组所属集群和所属命名空间。
4. 单击部署组列表上方的【新建部署组】。
5. 设置部署组相关信息。
    - **部署组名称**：部署组的名称，不超过60个字符。
    - **命名空间**：选择命名空间。
    - **关联应用**：选择部署组关联的应用。关联应用字段决定了后续镜像来源和应用配置来源。
    - **实例资源限制**：分配给单个实例使用的 CPU 核数和内存资源 Request 值。
    - **实例数量**：一个实例由相关的一个容器构成，可单击 + 和 - 控制实例数量。实例数和实例资源限制的乘积不能超过集群剩余的可用资源。
    - **网络访问方式**：网络的访问方式决定了部署组内应用的网络属性，不同访问方式的应用可以提供不同网络能力。参阅 《容器部署组访问方式》。
        - NodePort：访问方式不绑定外网负载均衡，在集群内所有主机节点自动分配 NodePort 端口，可通过 **集群内任意主机IP + NodePort** 访问该服务。
        - 公网：访问方式自动绑定外网负载均衡，可通过外网负载均衡地址访问该服务。
        - 集群内访问：访问方式不绑定外网负载均衡，该服务只能在集群内部访问。
    - **端口映射**：容器端口与服务端口的映射关系。
        - 协议：TCP、UDP。
        - 容器端口：容器内应用程序监听的端口。
        - 服务端口：集群外通过负载均衡域名或IP+服务端口访问服务；集群内通过服务名+服务端口访问服务。建议和容器端口保持一致。
    - **更新方式**：选择部署的更新方式。
        - 快速更新：直接关闭所有实例，启动相同数量的新实例。
        - 滚动更新：对实例进行逐个更新，这种方式可以让您不中断业务实现对服务的更新。
    - **日志配置项**：选择日志配置项，用于采集应用的业务日志数据。

部署应用

前提条件：已经将镜像推送到应用的镜像仓库中，参考 《制作容器镜像》和 《镜像仓库》。

部署流程：

1. 单击部署组列表页右侧的【部署应用】。
2. 设置部署信息。
    - **选择镜像**：选择要部署的镜像。
    - **实例资源保留**：分配给单个实例使用的 CPU 核数和内存资源 Request 值。
    - **实例数量**：一个实例由相关的一个容器构成，可单击 + 和 - 控制实例数量。实例数和实例资源限制的乘积不能超过集群剩余的可用资源。
    - **JVM 启动参数**（选填，仅适用普通应用）：设置 Java 应用的启动参数。参数会通过 `JAVA_OPTS` 环境变量带到容器运行环境中，参考 《制作容器镜像》 中的使用方式。

应用扩缩

1. 单击部署组右侧的【应用扩缩】。
2. 选择扩缩的实例数量后，单击【提交】按钮。

删除部署组

1. 单击部署组列表页右侧的【删除】。
2. 弹出提示页面，显示部署组内的实例信息，单击【确定】删除部署组。

实例资源限制说明

新建容器部署组时可以设置实例数量和每个实例资源限制（**这里设置的都是 Request 值**）。实例的资源限制包括两个指标：CPU 和内存大小。 TSF 会根据 Request 值，自动设置容器实例的 CPU Limit 和 内存 Limit，两者的数量关系如下：

```
Limit = Request * 2
```

当 `实例数量 * 资源限制数量 > 集群剩余的资源` 时，会提示 "资源不足，请导入节点" 的提示语。此时用户需要去集群页面导入云服务器以扩充资源。例如实例数量为2，实例资源限制是 `CPU=0.5核，内存=1GB` ，而集群剩余资源为 `CPU=0.8核，内存=2GB` 时， 由于 CPU 核数资源不够，会提示资源不足的错误。

Java 应用的最大堆内存和容器内存大小关系

Java 应用通常需要设置 JVM 启动参数，包括最大堆内存（-Xmx）的设置。建议 JVM 最大堆内存和容器实例内存资源大小的关系符合以下关系：

```
JVM 最大堆内存 <= 容器内存限制 * 2 - 256MB
```

例如，在部署应用的弹框中设置实例资源限制为512MB，启动参数中设置 `-Xmx` 为768MB（512MB * 2-256MB）。

关于如何使用启动参数，请参考 《制作容器镜像》中 Spring Cloud 应用构建材料中 Dockerfile 启动命令中的 `${JAVA_OPTS}` 。

程序包管理包括上传程序包和删除程序包。

上传程序包

> 为了更好地使用本产品，请使用较新版本的 Chrome 浏览器。

1. 登录 TSF 控制台。
2. 在控制台页，单击左侧导航栏 【应用管理】。
3. 在应用管理列表页，单击目标应用的**ID/应用名**。
4. 在目标应用详情页，单击**程序包管理**标签页，单击【上传程序包】按钮。
5. 在**上传程序包**对话框中，填写相关参数。
    - 上传程序包：单击【选择文件】，选择编译为 fatjar 格式的程序包
    - 程序包版本：填写版本号
    - 备注：填写备注
6. 单击【提交】按钮，程序包上传成功后出现在程序包列表中。

删除程序包

1. 登录 TSF 控制台。
2. 在控制台页，单击左侧导航栏 【应用管理】。
3. 在应用管理列表页，单击目标应用的**ID/应用名**。
4. 在目标应用详情页，单击**程序包管理**标签页，单击程序包列表右侧的【删除】按钮。
5. 单击【确认】按钮。

目前使用云服务器部署的应用支持的程序包格式包括 jar 、tar.gz 和 zip。

- jar： FatJar 格式的程序包，用户可以参考 如何打 FatJar 包。
- tar.gz 、zip ： 压缩包中**必须**包含三个文件，**确保文件名正确**：
    - start.sh：启动脚本
    - stop.sh：停止脚本
    - cmdline：用于检查应用进程是否存在，**没有** `.sh` 后缀

文件类型	启动方式
jar	云服务器上的 agent 会使用 `jar -jar` 命令启动程序。
tar.gz	云服务器上的 agent 会解压压缩包，使用解压目录下的 `start.sh` 脚本启动应用程序。
zip	云服务器上的 agent 会解压压缩包，使用解压目录下的 `start.sh` 脚本启动应用程序。

start.sh / stop.sh / cmdline 说明

以一个 Python 应用的压缩包为示例，解压后的文件目录如下：

- promotionService.py
- start.sh
- stop.sh
- cmdline

start.sh 启动脚本内容如下：

```
#! /bin/bash

already_run=`ps -ef|grep "python promotion"|grep -v grep|wc -l`
if [ ${already_run} -ne 0 ];then
    echo "promotionService already Running!!!! Stop it first"
    exit -1
fi

nohup python promotionService.py 8093 &
```

stop.sh 停止脚本内容如下：

```
#!/bin/bash

pid=`ps -ef|grep "python promotion"|grep -v grep|awk '{print $2}'`
kill -SIGTERM $pid
echo "process ${pid} killed"
```

cmdline 检测进程脚本，agent 通过 `ps -ef | grep 'cmdline 内容'` 来检测进程是否存在，示例如下：

```
python promotion
```

cmdline 更多说明

如果启动应用是 Java 应用，启动脚本中通过 `java -java xxx.jar` 来启动应用。在 cmdline 文件中使用完整的 Java 启动命令。例如启动脚本中包含如下启动命令：

```
java -Xms128m -Xmx512m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m -jar consumer-demo-0.0.1-SNAPSHOT.jar
```

那么在 cmdline 中内容为：

```
java -Xms128m -Xmx512m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m -jar consumer-demo-0.0.1-SNAPSHOT.jar
```

当应用启动后，agent 会在服务器上执行 `ps -ef | grep 'cmdline 内容'` 来检查进程是否存在。

> !如果没有 cmdline 文件或者 cmdline 文件内容不正确，在控制台上部署组的状态会显示为"已停止"，即使此时服务器上的应用已经运行起来（但是 TSF agent 无法获取应用进程状态）。

MacOS 系统压缩软件说明

对于 MacOS 系统的用户，使用系统自带压缩软件时，会在压缩包里面生成 `__MACOSX` 的临时目录，从而导致 agent 无法找到启停脚本。用户可以 下载 keka 压缩软件，选择 zip 压缩格式，勾选【排除 Mac 资源文件文件】选项。将文件拖拽到 keka 界面上进行压缩，这种方式生成的压缩包没有 `__MACOSX` 的临时目录。

镜像仓库用于存放用户上传的镜像。

镜像仓库使用指引

TSF 中镜像仓库和容器应用一一对应，TSF 会针对每个容器应用创建一个名为 `tsf_<账号ID>/<应用名>` 的镜像仓库。创建容器应用后，在容器应用的详情页中找到【镜像】标签页。单击【使用指引】按钮，镜像仓库的使用指引。

使用指引中显示了查看登录镜像仓库、拉取镜像和推送镜像到仓库的命令行。

在执行这些操作前，请确保执行命令的机器上已安装 `docker` ，同时可以访问镜像仓库。 使用 `sudo` 允许系统管理员让普通用户执行 `docker` 命令。

登录镜像仓库

```
sudo docker login --username=<账号ID>  <镜像仓库地址>
```

> 如果用户需要输入两次密码，首次为 sudo 密码，第二次为镜像仓库登录密码。

拉取镜像

```
sudo docker pull [NAME]:[tag]
```

其中 `[NAME]` 表示镜像名称， `[tag]` 表示镜像标签，例如：

```
sudo docker pull <镜像仓库地址>/tsf_123456/tsf_demo:v1.0
```

推送镜像到仓库

1. 登录镜像仓库

```
sudo docker login --username=<账号ID>  <镜像仓库地址>
```

2. 给镜像打 tag

```
sudo docker tag [ImageId] [TARGET_IMAGE]:[tag]
```

其中 `[ImageId]` 表示源镜像 ID，可以通过 `docker image ls` 命令查看。 `[TARGET_IMAGE]` 表示目标镜像名称，直接复制控制台上的名称即可。 `[tag]` 表示镜像标签。例如：

```
sudo docker tag 0e5574283393 <镜像仓库地址>/tsf_123456/tsf_demo:v1.0
```

3. 将镜像推送到仓库

```
docker push [TARGET_IMAGE]:[tag]
```

其中 `[TARGET_IMAGE]` 和 `[tag]` 和第 2 步保持一致。例如：

```
docker push <镜像仓库地址>/tsf_123456/tsf_demo:v1.0
```

操作场景

该任务指导您通过 Spring Cloud 和 Mesh 两种方式制作容器镜像。

操作步骤

准备构建材料

Spring Cloud 应用构建材料

1. 下载使用了 TSF SDK 的 Spring Cloud 应用 Demo JAR 包，参考 《Demo 工程概述》。
2. 在该 JAR 包同级目录下，编写 Dockerfile 文件：

```
FROM centos:7
RUN echo "ip_resolve=4" >> /etc/yum.conf
RUN yum update -y && yum install -y java-1.8.0-openjdk
# 设置时区。这对于日志、调用链等功能能否在 TSF 控制台被检索到非常重要。
RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
RUN echo "Asia/Shanghai" > /etc/timezone
ENV workdir /app/
ENV jar provider-demo-0.0.1-SNAPSHOT.jar
COPY ${jar} ${workdir}
WORKDIR ${workdir}
# JAVA_OPTS 环境变量的值为部署组的 JVM 启动参数，在运行时 bash 替换。使用 exec 以使 Java 程序
可以接收 SIGTERM 信号。
CMD ["sh", "-ec", "exec java ${JAVA_OPTS} -jar ${jar}"]
```

对于1.12.0及之前版本的 TSF，要支持 stdout 日志，需要在启动命令中使用 start.sh 启动。start.sh 下载>> 以下代码片段为使用 start.sh 启动的示例。

```
COPY start.sh /data/tsf/
CMD ["sh", "-c", "cd /data/tsf; sh start.sh provider-demo-0.0.1-SNAPSHOT.jar /data/tsf"]
```

Mesh 应用构建材料

1. 下载 Mesh 应用 Demo 包：userService.tar.gz。
2. 在该 tar.gz 包同级目录下，编写 Dockerfile 文件：

```
FROM centos:7
RUN mkdir /root/app/
# 其中 userService.tar.gz 是 Mesh 应用压缩包
ADD userService.tar.gz /root/app/
# 设置时区。这对于日志、调用链等功能能否在 TSF 控制台被检索到非常重要。
RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
RUN echo "Asia/Shanghai" > /etc/timezone
ENTRYPOINT ["bash","/root/app/userService/start.sh"]
```

Mesh 应用压缩包解压后的文件目录结构及文件规范参考 《Mesh Demo 介绍》。

使用文件配置功能

如果容器应用需要使用 TSF 文件配置功能，需要修改 Dockerfile，具体使用指引参考 《文件配置》中的前提条件说明。

构建镜像

1. 在 `Dockerfile` 所在目录执行 `build` 命令：

   ```
   docker build . -t ccr.ccs.tencentyun.com/tsf_<账号 ID>/<应用名>:[tag]
   ```

   其中 `<账号 ID>` 表示用户腾讯云的账号 ID， `<应用名>` 表示控制台上的应用名。 `tag` 为镜像的 tag，用户可自定义。帐号 ID 及应用名可以在 TSF 控制台的应用详情页中获得。示例如下：

   ```
   docker build . -t ccr.ccs.tencentyun.com/tsf_12345678/tsf_test_app:v1.0
   ```

2. 命令执行完成后，通过 `docker image ls` 查看创建的镜像。

概述

弹性伸缩含义：根据预先设定的弹性伸缩规则，动态增加或者减少部署组的实例数。

弹性伸缩规则：由规则名、扩容活动、缩容活动、冷却时间等参数构成的规则，用来描述弹性扩缩容的触发条件、实例数量变化和限制。

弹性伸缩指标：

- CPU 利用率：在指定时间范围内，部署组内所有实例 CPU 利用率的平均值。
- 内存利用率：在指定时间范围内，部署组内所有实例内存利用率的平均值。
- 请求 QPS：在指定时间范围内，部署组内所有实例请求 QPS 的平均值。
- 响应时间：在指定时间范围内，部署组内所有实例响应时间的平均值。

冷却时间：设置冷却时间，可以确保在上一扩（缩）容活动生效前弹性伸缩不会启动或终止其他实例。弹性伸缩会等待冷却时间完成，然后再继续扩（缩）容活动。建议设置冷却时间大于持续时间。

新建规则

1. 在 TSF 控制台左侧导航栏，单击【弹性伸缩】。
2. 在弹性伸缩页面左上方，单击【新建】按钮。
3. 在新建弹性伸缩规则中，填写弹性伸缩规则内容。
    - **名称**
    - **扩容活动** a. 触发条件：由指标、阈值、持续时间构成。多条触发条件为逻辑或（OR）的关系，满足任一条件。 b. 增加实例数：每次部署组的指标达到了触发条件后，增加的实例数量。 c. 最大实例数：部署组的实例数上限。
    - **缩容活动** a. 触发条件：由指标、阈值、持续时间构成。多条触发条件为逻辑与 AND 的关系，必须同时满足。 b. 减少实例数：每次部署组的指标达到了触。发条件后，减少的实例数量。 c. 最小实例数：部署组的实例数下限
    - **冷却时间**：建议设置冷却时间大于持续时间，如持续时间设置1分钟，冷却时间设置5分钟。

关联部署组

创建弹性伸缩规则后，将规则关联到部署组上。

1. 在弹性伸缩规则列表的操作列，单击【关联部署组】。
2. 在关联部署组页面，选择应用，然后选择部署组。
3. 在关联部署组页面左下方，选择是否立刻开启规则。如果选择开启，则规则会在部署组上立刻生效，否则将不生效。用户可以后续在规则详情页的【关联部署组】 tab 页中修改启用状态。

解除规则和部署组的关联

1. 单击规则名称进入详情页，单击【关联部署组】标签页。
2. 在关联部署组列表右侧操作栏，单击【删除】按钮，解除规则和部署组的关联。

删除规则

前提条件：已解除规则和部署组的关联。

1. 在弹性伸缩列表的操作栏，单击【删除】按钮。
2. 在弹出的对话框中，单击【确认】按钮。

8. 配置管理

配置功能概述

- 版本管理
- 动态更新
- 配置回滚
- 配置发布历史
- 配置模板

配置类型

配置类型	功能说明	适用应用类型	作用对象
应用配置	动态更新 Spring Cloud 或者 Dubbo 应用内的配置	Spring Cloud 或 Dubbo 应用	部署组
全局配置	动态更新 Spring Cloud 或者 Dubbo 应用内的配置	Spring Cloud 或 Dubbo 应用	命名空间
文件配置	将文件配置发布到实例指定路径，发布成功后触发回调	任何应用类型	部署组

应用配置、全局配置、本地配置优先级

应用配置和全局配置属于 TSF 平台上的配置（下面称为 `远程配置`），本地配置是应用程序在代码工程中创建的配置（如 `application.yml` 和 `bootstrap.yml`）。应用配置和全局配置的根本区别在于 **配置发布的范围**，应用配置发布的范围是部署组维度，全局配置发布的范围是命名空间维度。

```
优先级：应用配置 > 全局配置 > 本地配置
```

当用户通过 TSF 控制台发布 **远程配置**，微服务应用会按照配置的 key 来进行合并操作。举例来说微服务应用本地 `application.yml` 配置文件的内容中包括：

```
# application.yml
username: test_user1
feature.status: false
feature.color: red
```

TSF 平台上 `远程配置` 的内容是：

```
# TSF 应用配置或者全局配置
username: test_user2
feature.status: true
```

当 `远程配置` 的发布范围包含了上面的服务实例，微服务应用会将远程配置和本地配置按照 `key` 来进行合并，最终生成的配置是：

```
# 远程配置与本地配置合并结果
username: test_user2
feature.status: true
feature.color: red
```

多份应用配置发布到同一个部署组

TSF 支持多份应用配置发布到同一个部署组，多份配置会根据发布时间的先后顺序以 `key` 来进行合并。举例来说，应用 A 有两个应用配置项 `config-1` ， `config-2` 。

`config-1` 的配置内容：

```
# config-1
username: test_user1
feature.status: false
```

`config-2` 的配置内容：

```
# config-2
username: test_user2
feature.color: red
```

`config-1` 和 `config-2` 先后发布到部署组 `group` ，会按照 `key` 来进行合并。

```
# config-1 与 config-2 合并结果
username: test_user2
feature.status: false
feature.color: red
```

操作场景

应用配置功能有两个入口，一个入口是在单个应用的应用详情页内，另一个入口是在**配置管理**模块的**应用配置**。应用配置功能**仅针对 Spring Cloud 应用生效**，应用配置支持如下功能：

- 创建配置项：一个配置项管理多个版本的配置。
- 生成新版本：基于历史版本生成新版本。
- 发布配置：支持发布配置到部署组。
- 发布情况：查看配置项的发布到哪些部署组。
- 回滚：回滚到上一个版本的配置。

前提条件

在使用控制台的应用配置功能前，请确保已经按照 《分布式配置》配置了相关依赖项。

操作步骤

创建配置

1. 登录 TSF 控制台。
2. 在左侧导航栏，单击【配置管理】>【应用配置】。
3. 在应用配置页面顶部，选择目标应用。
4. 在配置列表标签页，单击【新建】。
5. 填写配置内容。
    - 配置项名：填写配置名。
    - 配置内容：按照 YAML 格式。YAML 格式规范参考 YAML 格式介绍。
    - 版本号：填写初始版本号。
    - 版本描述：填写初始版本的描述。
6. 单击【完成】。

> **注意**：单个应用配置版本的大小不能超过65535个字节，如果应用的配置超过了该上限值，可以分成多个应用配置项发布到同一个部署组，多个配置会合并成一份配置。

生成新版本配置

1. 在配置列表页，单击目标配置名称，进入详情页。
2. 在配置版本标签页，单击某个配置版本旁的【生成新版本】按钮。
3. 填写变更的新版本的配置内容和版本号。

> **注意**：新版本配置的版本号不能与原版本相同。

4. 单击【完成】。

发布配置

用户可以可以发布配置项的某个版本到部署组上。

1. 在配置列表页，单击目标配置名称，进入详情页。
2. 在配置版本标签页，单击某个配置版本旁的【发布】。
3. 选择配置发布的目标部署组，填写发布描述。
4. 单击【提交】。

配置合并逻辑说明

按照配置**下发时间**来排序执行合并（merge）。不同名的配置项中如果存在相同 key 会进行合并。合并规则：按照配置下发时间排序，离当前时间近的优先级较高。举例如下：

```
1. 创建配置项 config-abc，配置内容是 custom-key: value-1，发布时间 15:00:00
2. 创建配置项 config-bcd，配置内容是 custom-key: value-2，  发布时间15:00:01
```

最终在实例上生效的配置：`custom-key: value-2`

查看配置发布历史

用户可以在【配置发布历史】页面查看应用下部署组的配置发布记录。

1. 在左侧导航栏，单击【配置管理】>【应用配置】>【配置发布历史】。
2. 页面顶部选择【关联应用】，查看该应用下部署组的配置发布记录。

回滚配置

回滚配置会将部署组的配置回滚到**上一次**发布的版本。

1. 单击【配置发布历史】标签页。
2. 在配置发布历史列表中，单击右侧的【回滚】。
3. 填写回滚说明，单击【提交】。

操作场景

全局配置功能用于动态更新应用代码中的配置。全局配置可以保证配置内容在某个集群或者命名空间中全局生效。 全局配置包括管理配置和发布配置两部分。管理配置包括创建配置、生成新版本配置和删除配置。配置可以发布到命名空间下的所有应用。

前提条件

在使用全局配置功能之前，请确保已经按照 《分布式配置》开发文档添加了代码注释。

操作步骤

创建配置

1. 登录 TSF 控制台。
2. 在左侧导航栏，单击【配置管理】>【全局配置】。
3. 在全局配置页面左上角，单击【配置列表】。
4. 在配置列表页，单击【新建】，进入配置界面。
5. 填写配置内容。配置可以按照 YAML 方式进行编辑。YAML 格式规范参考 YAML 格式介绍。
6. 单击【提交】，完成创建。

> **注意**：单个全局配置版本的大小不能超过65535个字节，如果实际使用的配置超过了该上限值，可以分成多个全局配置项发布到同一个命名空间，多个配置会合并成一份配置。

生成新版本配置

1. 单击配置列表右侧的【生成新版本】。
2. 填写变更的配置内容和版本号。
3. 单击【提交】。

> **注意**：新版本配置的版本号不能与原版本相同。

删除配置

1. 单击配置项名称，进入配置详情页。
2. 删除每个配置版本，删除最后一个配置版本后，配置项将被删除。

> **注意**：对于已经发布的配置，需要在【发布情况】页面中先删除配置，然后再删除配置版本，避免配置被误删除。

3. 在弹框中，单击【确认】。

发布配置

1. 在配置列表中，单击操作列的【发布】。
2. 选择配置发布的目标集群和命名空间，填写发布描述。
3. 单击【提交】。

配置合并逻辑说明

按照配置**下发时间**排序执行合并（merge）。不同名的配置项中如果存在相同 key 会进行合并。合并规则：按照配置下发时间排序，离当前时间近的优先级较高。举例如下：

```
1. 创建配置项 config-abc，配置内容是 custom-key: value-1，发布时间 15:00:00
2. 创建配置项 config-bcd，配置内容是 custom-key: value-2，  发布时间15:00:01
```

最终在实例上生效的配置：`custom-key: value-2`

查看配置发布历史

1. 在全局配置页面中，单击【配置发布历史】。
2. 查看全局配置的发布历史。

配置回滚

1. 在全局配置页面中，单击【配置发布历史】。
2. 单击配置发布历史列表右侧的【回滚】。
3. 填写回滚说明，单击【提交】。

配置模板功能是为了方便用户保存常用的配置信息，也提供了 Ribbon, Hystrix, Zuul 等 Spring Cloud 组件的配置模板。用户可以基于已有的配置模板进行修改。

用户可以基于配置模板来创建 《应用配置》 或者 《全局配置》。

在使用配置模板功能之前，请确保已经按照《分布式配置》开发文档添加了代码注释。

新建配置模板

1. 登录 TSF 控制台。
2. 单击左侧导航栏 **配置管理**，再选择 **配置模板**。
3. 单击【新建模板】。
4. 填写配置模板信息。
    - 模板名：填写模板名。
    - 类型：Ribbon、Hystrix、Zuul 或自定义。
    - 配置内容：根据不同的类型，会自动生成对应的配置内容。用户可以进一步修改配置内容。
    - 描述：填写描述信息。
5. 单击【提交】按钮。

使用配置模板

用户可以使用配置模板来创建应用配置或者全局配置，下面以全局配置举例。

1. 登录 TSF 控制台。
2. 单击左侧导航栏 **配置管理**，再选择 **全局配置**。
3. 单击【导入配置模板】。
4. 选择要导入的配置模板。
5. 在新建配置页面中，补充全局配置的其他信息。

文件配置功能支持用户通过控制台将配置下发到服务器的指定目录。应用程序通过读取该目录下的配置文件实现特殊的业务逻辑。

文件配置支持如下功能：

- 创建文件配置项：一个文件配置项管理多个版本的配置。
- 生成新版本：基于历史版本生成新版本。
- 发布配置：支持发布配置到部署组。
- 发布情况：查看配置项的发布到哪些部署组。
- 回滚：回滚到上一个版本的配置。

使用场景

定时检查配置是否更新

- 应用程序中包含了读取指定目录配置文件的逻辑，例如定时去检查配置文件是否更新（通过文件 md5 是否变化等方式检查），如果更新了会执行特定逻辑。
- 在控制台上创建文件配置，下发到部署组。

动态替换 PHP 文件

通过控制台发布一个 PHP 文件到指定目录，来达到动态替换服务器上 PHP 文件的目的。

前提条件

能否使用文件配置功能，依赖于应用部署的环境是否满足以下条件：

- **对于使用虚拟机部署的应用**：只有2018年11月20号之后导入到集群的云主机上会具有满足应用配置功能的环境。
- **对于容器部署的应用**：该功能需要用户修改 Dockerfile。以下示例在 《制作容器镜像》文档的基础上做修改：
  - 需要将 `tsf-consul-template-docker.tar.gz` （下载地址） 添加到到 `/root/` 目录下：

    ```
    ADD tsf-consul-template-docker.tar.gz /root/
    ```

  - 启动脚本中，需要执行 `/root/tsf-consul-template-docker/script` 目录下的 `start.sh` 脚本：

    ```
    CMD ["sh", "-ec", "sh /root/tsf-consul-template-docker/script/start.sh; exec java ${JAVA_OPTS} -jar ${jar} 2>&1"]
    ```

控制台基本操作

创建文件配置

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【配置管理】>【文件配置】。
3. 在文件配置页面，选择配置列表标签页，单击【新建】按钮。
4. 填写文件配置信息
    - 配置名称
    - 关联应用
    - 文件保存编码
    - 配置内容：支持上传文件或者控制台编辑
    - 配置文件名称：下发到服务器的配置文件的文件名称
    - 版本号
    - 版本描述
    - 配置下发路径：配置下发到服务器的路径
    - 后置脚本（选填）：配置下发到服务器后执行的命令（**不需要** 包含 #! /bin/bash)

生成新版本

1. 在配置列表页，单击配置名称进入详情页。
2. 单击某个配置版本旁的【生成新版本】。
3. 填写变更的新版本的配置内容和版本号。
4. 单击【完成】，生成新版本。

发布配置

1. 在配置列表页，单击配置名称进入详情页。
2. 单击某个配置版本旁的【发布】。
3. 选择配置发布的目标部署组，填写发布描述。
4. 单击【提交】，完成发布。

查看配置发布历史

1. 在文件配置页，单击**配置发布历史**标签页。
2. 选择目标部署组，单击【查询】，查看配置发布历史。

配置回滚

1. 在文件配置页，单击**配置发布历史**标签页。
2. 单击配置发布历史列表右侧的【回滚】。
3. 填写回滚说明，单击【提交】，完成回滚。

配置加密功能提供了对配置值加密的存储全套解决方案。

通过增强源生SDK能力，同时兼容本地文件配置和分布式配置的配置值加密。

一、准备工作

1. 确保使用最新的 TSF SDK。
2. 按照《分布式配置》开发文档添加了代码注释。
3. 下载 SDK 加密工具。
4. 准备需要加密的相关信息（此处为举例，用户使用时请调整）
    - 密码明文（plaintext）：TX_PwDemO_1hblsqT
    - 密钥（encpty password）：encryptPassword

二、SDK 加密工具

1. 找到加密工具包 (spring-cloud-tsf-encrypt-1.1.1-RELEASE.jar)。
2. 执行以下命令对配置明文密码进行加密：

```
D:\repo\com\tencent\tsf\spring-cloud-tsf-encrypt\1.1.1-RELEASE>java -jar spring-cloud-
tsf-encrypt-1.1.1-RELEASE.jar encrypt TX_PwDemO_1hblsqT encryptPassword
```

输出结果：

```
[encrypt] result:
3M7wGw2XtFc5Y+rxOgNBLrm2spUtgodjIxa+7F3XcAo=
```

用例：

```
D:\repo\com\tencent\tsf\spring-cloud-tsf-encrypt\1.1.1-RELEASE>java -jar spring-cloud-
tsf-encrypt-1.1.1-RELEASE.jar
At least 3 arguments required. Usage: [operation] [content] [password]
[operation]: Choose one from [encrypt | decrypt].
[content]: Plaintext when encrypt or ciphertext when decrypt.
[password]: Encrypt or decrypt password.
```

3. 执行以下命令对密文密码进行解密：

```
D:\repo\com\tencent\tsf\spring-cloud-tsf-encrypt\1.1.1-RELEASE>java -jar spring-cloud-
tsf-encrypt-1.1.1-RELEASE.jar decrypt 3M7wGw2XtFc5Y+rxOgNBLrm2spUtgodjIxa+7F3XcAo=
encryptPassword
```

输出结果：

```
[decrypt] result:
TX_PwDemO_1hblsqT
```

用例：

```
D:\repo\com\tencent\tsf\spring-cloud-tsf-encrypt\1.1.1-RELEASE>java -jar spring-cloud-
tsf-encrypt-1.1.1-RELEASE.jar
At least 3 arguments required. Usage: [operation] [content] [password]
[operation]: Choose one from [encrypt | decrypt].
[content]: Plaintext when encrypt or ciphertext when decrypt.
[password]: Encrypt or decrypt password.
```

三、配置项填写方式

> **注意：**本地配置和线上配置同时支持（需要符合spring-config源生规范）。

本地 YAML

配置在 application.yml或application-*.yml

```
tsf:
  inventory:
    password:
      encrypt1: ENC(3M7wGw2XtFc5Y+rxOgNBLrm2spUtgodjIxa+7F3XcAo=)
```

配置中心 YAML

配置在全局配置/应用配置，并发布

```
tsf:
  inventory:
    password:
      encrypt2: ENC(3M7wGw2XtFc5Y+rxOgNBLrm2spUtgodjIxa+7F3XcAo=)
```

本地 Properties

配置在application.properties或application-*.properties

```
tsf.inventory.password.encrypt3=ENC(3M7wGw2XtFc5Y+rxOgNBLrm2spUtgodjIxa+7F3XcAo=)
```

四、业务应用使用

环境变量（推荐）

在系统环境变量中配置密钥(password)：此时密钥泄露的风险最小。

```
tsf_config_encrypt_password=encryptPassword
```

JVM 参数（不推荐）

也可以在JVM参数中配置密钥(password)：

```
-Dtsf_config_encrypt_password=encryptPassword
```

启动参数（不推荐）

也可以在应用启动参数中配置密钥(password)：

```
--tsf_config_encrypt_password=encryptPassword
```

Java 测试代码

Java代 码按照常规配置使用。

配置类：

```
@ConfigurationProperties("tsf.inventory.password")
@Component
@RefreshScope
public class PasswordConfiguration {

    private String encrypt1;
    private String encrypt2;
    private String encrypt3;

    @Value("${tsf.inventory.password.encrypt1}")
    private String encrypt4;
    @Value("${tsf.inventory.password.encrypt2}")
    private String encrypt5;
    @Value("${tsf.inventory.password.encrypt3}")
    private String encrypt6;

    // getters and setters
}
```

测试类：

```
@RestController
public class TestController {

    @Autowired
    private PasswordConfiguration pwConfig;
    /**
     * 显示明文密码
     *
     * @return 明文密码
     */
    @RequestMapping("/inventory/password")
    public String showPassword() {
        String content = "TX_PwDemO_1hblsqT";
        StringBuffer sb = new StringBuffer("Test Config Encrypt/Decrypt:\n");
        // 内存读取
        sb.append(String.format("[%s]\t内存读取*.yml文件配置: %s\n",
                content.equals(

SpringCloudTsfApplication.ctx.getEnvironment().getProperty("tsf.inventory.password.encrypt1")),

SpringCloudTsfApplication.ctx.getEnvironment().getProperty("tsf.inventory.password.encrypt1")));

        sb.append(String.format("[%s]\t内存读取consul配置: %s\n",
                content.equals(

SpringCloudTsfApplication.ctx.getEnvironment().getProperty("tsf.inventory.password.encrypt2")),

SpringCloudTsfApplication.ctx.getEnvironment().getProperty("tsf.inventory.password.encrypt2")));

        sb.append(String.format("[%s]\t内存读取*.properties文件配置: %s\n",
                content.equals(

SpringCloudTsfApplication.ctx.getEnvironment().getProperty("tsf.inventory.password.encrypt3")),

SpringCloudTsfApplication.ctx.getEnvironment().getProperty("tsf.inventory.password.encrypt3")));

        // Bean读取
        sb.append(String.format("[%s]\tBean读取*.yml文件配置: %s\n",
content.equals(pwConfig.getEncrypt1()),
                pwConfig.getEncrypt1()));
        sb.append(String.format("[%s]\tBean读取consul配置: %s\n",
content.equals(pwConfig.getEncrypt2()),
                pwConfig.getEncrypt2()));
        sb.append(String.format("[%s]\tBean读取*.properties文件配置: %s\n",
content.equals(pwConfig.getEncrypt3()),
                pwConfig.getEncrypt3()));
        // @Value读取
        sb.append(String.format("[%s]\t@Value读取*.yml文件配置: %s\n",
content.equals(pwConfig.getEncrypt4()),
                pwConfig.getEncrypt4()));
        sb.append(String.format("[%s]\t@Value读取consul配置: %s\n",
content.equals(pwConfig.getEncrypt5()),
                pwConfig.getEncrypt5()));
        sb.append(String.format("[%s]\t@Value读取*.properties文件配置: %s\n",
content.equals(pwConfig.getEncrypt5()),
                pwConfig.getEncrypt5()));
```

```
        return sb.toString();
    }
}
```

输出结果如下：

```
Test Config Encrypt/Decrypt:
[true]    内存读取*.yml文件配置: TX_PwDemO_1hblsqT
[true]    内存读取consul配置: TX_PwDemO_1hblsqT
[true]    内存读取*.properties文件配置: TX_PwDemO_1hblsqT
[true]    Bean读取*.yml文件配置: TX_PwDemO_1hblsqT
[true]    Bean读取consul配置: TX_PwDemO_1hblsqT
[true]    Bean读取*.properties文件配置: TX_PwDemO_1hblsqT
[true]    @Value读取*.yml文件配置: TX_PwDemO_1hblsqT
[true]    @Value读取consul配置: TX_PwDemO_1hblsqT
[true]    @Value读取*.properties文件配置: TX_PwDemO_1hblsqT**
```

9. 服务治理

服务是微服务平台管理的基本单元。服务管理包括服务的生命周期管理和服务治理两部分。 服务基本操作包括创建服务和删除服务。

创建服务

1. 登录 TSF 控制台。
2. 单击左侧导航栏的 **服务治理**，选择集群和命名空间。
3. 单击服务列表页的【新建】。
4. 设置服务的基本信息后，单击【提交】按钮。
    - **服务名称**：要创建的服务的名称，不超过 60 个字符。服务名称由小写字母、数字和 - 组成，且由小写字母开头，小写字母或数字结尾。
    - **服务描述**：服务的描述信息。

删除服务

1. 当服务的状态为【离线】时，才能删除服务。单击服务列表右侧【删除】。
2. 在弹框中单击【确认】按钮。

> **注意：** 只有当服务运行的实例数为0时，可删除服务。

服务监控

在 TSF 控制台服务治理页面可以看到线上服务的请求数、请求成功率、平均耗时等监控数据。数据统计周期都是 24 小时。

- **请求数**：对一个服务，统计其作为服务提供者，被所有消费他的服务消费者发起调用的 24 小时内总调用数。
- **请求成功率**：对于一个服务，统计 24 小时内其作为服务提供者，成功向消费他的所有服务消费者返回请求的总数比上服务请求总数。
- **平均耗时**：对于一个服务，统计 24 小时内其作为服务提供者，统计消费者从发起调用到调用返回到服务提供者的耗时平均值。

标签说明

TSF 引入**标签**概念以区分不同的请求来源，TSF 标签包括系统标签和业务自定义标签。

- **系统标签** 每一个 TSF 上运行的服务都已经被预先设置好了某些标签，如发起请求的服务消费方所在的部署组、IP、服务发起方的版本号等。
- **业务自定义标签** 在实际的使用中，如果系统自带标签不能保证用户使用的场景，用户可以自定义标签内容。对于 Spring Cloud 应用，TSF 提供了用户配置自定义标签的 SDK，参考开发手册 《参数传递》；对于 Mesh 应用，用户需要在 header 中设置标签，参考《Mesh 开发使用指引》中关于设置自定义标签的说明。

> !这里的标签和腾讯云的标签产品不是同一个概念。腾讯云的标签产品是一种划分资源的方式，而 TSF 服务治理中的标签是为了区分不同的请求来源。

标签表达式

用户在控制台创建服务治理规则时，可以选择通过设置**标签表达式**区分请求来源。多个标签表达式之间是**逻辑与（AND）**的关系。例如两条标签表达式分别是：

- 系统标签主调服务名等于 consumer-demo
- 自定义标签 userid 等于123456

只有当一条请求是 consumer-demo 发出，**且**带有 userid 是123456的自定义标签时才满足上面2个标签表达式。

一条标签表达式中，逻辑关系与值的个数对应如下：

逻辑关系	值个数
包含（IN）	多个
不包含（ NOT IN）	多个
等于（==）	一个
不等于（!=）	一个
正则表达式（regex）	一个

在服务的详情页中会显示出服务提供的 API 列表。API 列表显示服务对外提供的 API。 单击 API 进入详情页，可以查看到 API 的详细信息。

API 详情按照【应用名/版本号】划分显示了 API 的详细信息，包括：路径、方法、描述、入参、出参。其中 Models 表示参数中的复杂类型。

服务鉴权是处理微服务之间相互访问权限问题的解决方案。配置中心下发鉴权规则到服务，当请求到来时，服务根据鉴权规则判断鉴权结果，如果鉴权通过，则继续处理请求，否则返回鉴权失败的 HTTP 状态码 403（Forbidden）。

鉴权原理

鉴权流程如下：

服务鉴权功能支持白名单和黑名单两种鉴权方式。

- **白名单**：当请求匹配**任意一条**鉴权规则时，允许调用；否则拒绝调用。
- **黑名单**：当请求匹配**任意一条**鉴权规则时，拒绝调用；否则允许调用。

多个鉴权规则

一个服务可能有多个鉴权规则，多个鉴权规则之间是逻辑或（OR）的关系，只要请求满足任意一条鉴权规则，就相当于匹配成功。

示例说明

示例1

需求：服务 provider-demo 只允许来自 consumer-demo 服务**且**带有 user=foo 的自定义标签的请求调用。 **解决方案**：要满足上面的鉴权需求，用户可以在 provider-demo 的鉴权页面，设置鉴权方式为白名单，鉴权规则如下图（注意最后要将生效状态改为生效）： **结论**：要满足 `逻辑与 AND` （既满足条件 A ，又满足条件 B）时，需要使用标签表达式。

示例2

需求：服务 provider-demo 只允许来自 consumer-demo 服务**或**带有 user=foo 的自定义标签的请求调用。 **解决方案**：要满足上面的鉴权需求，用户可以在 provider-demo 的鉴权页面，设置鉴权方式白名单，创建2条鉴权规则，如下图： **结论**：要满足 `逻辑与 OR` （满足条件 A 或 条件 B）时，需要使用多条鉴权规则。

说明

白名单鉴权方式示例： 鉴权规则内容是 username 等于 foo，当请求中带有 username=foo 的 tag 时，因为匹配规则，服务允许调用；当请求中带有 username=bar 的 tag 时，因为不匹配规则，服务拒绝调用。

黑名单鉴权方式示例：鉴权规则内容是 username 等于 foo，当请求中带有 username=foo 的 tag 时，因为匹配规则，服务**拒绝**调用；当请求中带有 username=bar 的 tag 时，因为不匹配规则，服务允许调用。

使用鉴权功能时，用户需要先在客户端配置依赖项，然后在 TSF 控制台设置鉴权规则。

1. 配置依赖项

- 对于 Spring Cloud 应用，请参考开发者手册中的 《服务鉴权》。
- 对于 Mesh 应用，无须额外配置。

2. 设置鉴权规则

1. 登录 TSF 控制台。
2. 在左侧导航栏，单击【服务治理】。
3. 在服务列表，单击服务名，进入服务详情页。
4. 在服务详情的【服务鉴权】标签页，选择鉴权方式：
    - 不启用：关闭鉴权功能。
    - 白名单：匹配任意一条规则的请求，允许调用。
    - 黑名单：匹配任意一条规则的请求，拒绝调用。
5. 选择白名单或黑名单，单击【新建鉴权规则】按钮。
6. 在新建页面中填写规则信息，并选择规则的【生效状态】。

3. 切换鉴权方式（可选）

用户可以通过控制台，从一种鉴权模式切换到另外一种鉴权模式。

- 白名单切换到黑名单（或黑名单切换到白名单）：保留鉴权规则，但鉴权逻辑逆转。如果用户希望切换后，使用新的规则，则需要删除原有的，再创建新的规则。
- 白名单（或黑名单）切换到不启用：关闭鉴权功能。

4. 检查鉴权效果

以产品 Demo 为例说明如何验证鉴权功能。consumer-demo 中已包含鉴权依赖 jar 包，因此这里只需要说明在控制台上创建鉴权规则用来限制特定 API 的调用。

consumer-demo 中提供了三个 API `/echo-rest/{str}` 、 `/echo-async-rest/{str}` 、 `/echo-feign/{str}` 。在控制台上设置鉴权方式为**白名单**，鉴权规则的标签表达式： 创建好规则后，登录机器，使用 curl 命令来验证鉴权是否生效。

`curl IP:PORT/echo-rest/hello?user=test` 预期：正常返回

`curl IP:PORT/echo-async-rest/hello?user=test` 预期：返回鉴权失败

`curl IP:PORT/echo-feign/hello?user=test` 预期：返回鉴权失败

限制说明

等于、不等于、包含、不包含属于严格匹配，正则表达式属于模糊匹配。因此当系统标签是被调方 API PATH 时，目前仅支持使用**正则表达式**的逻辑关系来匹配带参数的 API 请求。 例如标签的逻辑关系是正则表达式，值填写 `/echo/.*`，可以匹配带参数的请求 /echo/test123 （其中test123是参数）；当标签的逻辑关系是等于、不等于、包含、不包含关系，值是 `/echo/{param}` 时，不能匹配带参数的请求 /echo/test123 （其中 test123 是参数）。

服务限流主要是保护服务节点或者数据节点，防止瞬时流量过大造成服务和数据崩溃，导致服务不可用。当资源成为瓶颈时，服务框架需要对请求做限流，启动流控保护机制。

限流原理

在服务提供者端配置限流依赖项，在 TSF 控制台配置限流规则。此时服务消费者去调用服务提供者时，所有的访问请求都会通过限流模块进行计算，若服务消费者调用量在一定时间内超过了预设阈值，则会触发限流策略，进行限流处理。 TSF 限流方案采用了动态配额分配制，限流中控根据实例的历史流量记录，动态计算预测下一时刻该实例的流量，若所有实例的流量预测值都小于额定平均值（总配额/在线实例数），则以该平均值作为所有实例分配的配额；否则按预测流量的比例分配，且保证一个最小值。

限流使用场景

- 全局限流：服务设置一个最大的 QPS（每秒请求数） 阈值 T，当服务实际接收到的每秒请求数超过 T 时，触发限流。
- 基于标签限流：服务设置【基于标签】的限流规则。
    - 示例1：设置自定义标签 `username=foo` 的 QPS 阈值1000次/秒，当服务实际接收到的请求中包含了标签 `username=foo` 的 QPS 超过1000次/秒 时，触发限流。
    - 示例2：设置自定义标签 " username 包含 foo, bar " 的 QPS 阈值1000次/秒，当服务实际接收到的请求中包含了标签 `username=foo` 或 `username=bar` 的 QPS 超过1000次/秒 时，触发限流。

使用限流功能

要使用限流功能，用户需要在客户端配置依赖项，然后在 TSF 控制台设置限流规则。

1. 配置依赖项

对于 Spring Cloud 应用，参考开发手册中的《服务限流》。对于 Mesh 应用，如果希望使用基于标签的限流，需要在代码中设置标签。

2. 新建限流规则

前提条件：服务列表上有 "在线" 状态的微服务。 2.1 登录 TSF 控制台。 2.2 在左侧导航栏，单击【服务治理】。 2.3 在服务列表页，单击服务名，进入服务详情页。 2.4 选择**服务限流**标签页，单击【新建限流规则】按钮。 2.5 填写限流规则信息。 全局限流： 基于标签限流：

- **规则名**：填写规则名
- **限流粒度**：
    - 全局限流：不区分限流来源，统计所有请求
    - 基于标签限流：根据标签规则设置限流

- **单位时间**：正整数，单位：秒
- **请求数**：正整数，单位：次
- **生效状态**：是否立即启用限流规则
- **描述**：填写描述信息

2.6 单击【提交】按钮。

3. 启动限流规则

在限流规则列表中，可以修改规则的【生效状态】。 多条限流规则都是生效状态时，只要服务接收到的请求满足**任意一条**限流规则，就会触发限流逻辑。

4. 触发限流

假设服务提供了 `/echo` API，可以通过不断执行 `curl /echo` 来模拟限流场景。 **示例**：在 Demo 中 consumer-demo 服务提供了 `/echo-feign/{str}` API ，那么针对 consumer-demo 服务新建限流规则，限流粒度为全局限流，单位时间 2 秒，请求数 5 次。启用限流规则。 下载脚本 tsf_ratelimit.sh ，登录可以访问到 consumer-demo 的机器（consumer-demo 所在机器也可以），执行 `./tsf_ratelimit.sh <IP>:<Port>` ，其中 IP 是 consumer-demo 所在机器 IP， Port 为服务监听端口 18083。脚本的作用是**每 2 秒触发 10 次**调用。由于调用的频率大于限流规则，正常情况下，会收到 HTTP 429 (Too Many Requests) 的状态码。

5. 查看限流效果

如果请求数达到了限流阈值，任何到达的请求都会限流模块处理。如果该服务上的配额已经消耗完，会对请求返回 HTTP 429 （Too Many Requests）；否则会正常放行。用户可以在限流规则列表下方的**请求数-时间**图中查看到被限制的请求数或者**被限制请求率-时间**图中查看到被限制请求率（计算公式 `被限制请求率 = 被限制的请求数 / 请求数` ）随时间的变化。

服务路由概述

用户在使用 TSF 运行自己的业务时，由于业务的复杂程度，常常需要部署数目庞大的服务运行在现网环境中。这些服务运行在属性不同的实例上、部署在不同的地域中，用户经常需要根据符合自己特定要求的属性选择服务的提供者，对服务间流量的分配起到掌控的作用。 同时，在微服务的场景下，用户研发新版本上线的迭代周期越来越快，稳定敏捷的上线新版本需要微服务框架能够支持灰度发布、金丝雀发布、滚动发布等发布方式。通过服务路由功能，用户可以配置流量分配权重，设置某些权重的流量被分配到某个版本号中，为灰度发布等上线模式提供了无需终止服务的底层能力支持。为了保证满足客户的定制化需求，TSF 支持用户定制自己的路由标签，并支持选择不同的逻辑形式配置标签值，定向分配流量。 总而言之，服务路由功能的主要作用是将调用流量按照自己的需求进行分配。

服务路由原理

要实现服务路由需要完成两部分操作：

- 在控制台上，给**服务端（服务提供者）**设置路由规则。
- **客户端（服务消费者）**获取路由规则， 根据规则来分发请求。

以 `user -> shop -> promotion` 为例说明服务路由的原理，三个服务特点如下：

- user：Spring Cloud 应用，使用路由 SDK。
- shop：Mesh 应用，有两个版本 v1 和 v2.0-beta。
- promotion：Spring Cloud 应用，有两个版本 v1 和 v2.0-beta。

服务调用和路由情况如下图所示。用户需要在控制台创建如下路由规则：

- shop 服务详情页中配置路由规则：90%的流量分配到 v1 版本，10%的流量分配到 v2 版本。
- promotion 服务详情页中配置路由规则：服务名等于 shop 且版本号为 v1 的流量 100% 分配到 v1 版本，服务名等于 shop 且版本号为 v2 的流量100%分配到 v2 版本。

前提条件

要使用路由功能，用户需要在客户端配置依赖项，然后在 TSF 控制台设置路由规则。

使用服务路由

配置依赖项

Spring Cloud 应用请参考开发手册中的 《服务路由》。对于 Mesh 应用，如果希望使用基于自定义标签的路由，需要在代码中设置标签，关于如何设置标签参考 《Mesh 开发使用指引》。

新建路由规则

1. 登录 TSF 控制台。
2. 在左侧菜单中，选择【服务治理】。
3. 在服务列表中，选择需要配置服务路由规则的服务，单击服务名称，进入服务详情页。
4. 选择服务路由选项，单击【新建路由规则】。
5. 新建路由规则
    i. 路由规则名称
    ii. 填写规则
        - 流量来源配置：设置系统标签和自定义标签表达式。
        - 流量目的地：支持部署组和版本号两种目的地类型，确保权重加总为100。
    iii. 单击【提交】。

    > ?最多新建10条路由规则。

启用路由规则

1. 登录 TSF 控制台，单击左侧导航栏的【服务治理】，进入服务详情页面。
2. 选择服务路由标签。
3. 单击【生效状态】的切换按钮，当按钮为蓝色表明已经生效。
4. 配置生效后，可以在列表项的下面流量分配图中查看流量分配情况，用户可以选择时间段，查看部署组上流量分配情况。 24小时内的流量分配情况如下： 应用路由规则后10分钟之内的流量分配曲线如下：
5. 在流量分配图下方的流量分配表中，可以查看近五分钟内的平均每分钟请求数比例。

容错保护

开启容错保护后，会实现兜底策略。例如服务设置了如下路由规则：

- 10%的流量分配到 v1.0 版本。
- 40%的流量分配到 v1.1 版本。

- 50%的流量请求分配到 v1.2 版本。

假设场景： v1.0 版本的实例全部不可用

- 如果不开启容错保护，仍然会有10%的请求分发到 v1.0 版本的实例上，此时请求会失败。
- 如果开启容错保护，SDK 发现 v1.0 版本的实例不可用时，会这 10% 的请求随机分配给 v1.1 和 v1.2 版本实例，最终结果是 v1.1 和 v1.2 收到的请求比例**约等于**45%和55% 。

使用说明

- 填写路由规则需要在服务提供方进行配置，例如 A 服务调用 B 服务，需要在 B 服务上配置服务路由规则。
- 对于 Spring Cloud 服务，配置路由规则后，若配置的目标部署组无法运行，流量将按照原有默认的轮询方式分配到其他部署组上。
- 对于 Spring Cloud 服务，当服务提示未绑定应用时，需要在服务详情页单击编辑，绑定服务，才能开始配置路由规则。**服务绑定应用操作，一经绑定，不能修改。**
- Spring Cloud 服务调用其他服务的场景时，要使服务路由生效，需要确保 Spring Cloud 服务使用了 SDK 并添加开启路由注解，详情请参考开发手册中 《服务路由》。
- 对于 Mesh 应用，配置路由规则后，若配置的目标部署组无法运行，则路由规则配置失败，请求无法发送。

其他操作

编辑路由规则

1. 登录 TSF 控制台，单击左侧导航栏的【服务治理】，进入服务详情页面。
2. 选择服务路由标签。
3. 在服务路由页面已经提交的规则页面单击【编辑】。 在编辑页面，仅仅支持编辑规则的详情，不支持编辑规则类型。
4. 单击【提交】，完成路由编辑。

> ?在生效状态的路由规则可以编辑，编辑之后立即生效。

删除路由规则

1. 登录 TSF 控制台，单击左侧导航栏的【服务治理】，进入服务详情页面。
2. 选择服务路由标签。
3. 在服务路由页面已经提交的规则页面单击【删除】。

> ?在生效状态的服务路由规则不能被删除，只能先停用，再删除。

灰度发布

- 使用目的：当用户需要上线新的功能时，希望使用灰度发布的手段在小范围内进行新版本发布测试。
- 使用方法：用户可以将新的程序包上传到原有的应用中。用户选择按照权重的方式配置路由规则，填写权重大小，并选择目标版本版本号，便可以实现使用部分流量进行灰度发布的能力。生效中的权重可以被编辑，实时生效，间接实现了滚动发布的功能。

同地机房优先

- 使用目的：当企业规模较大时，单个机房的容量已经不能满足业务需求，业务常常出现跨机房部署的情况。然而由于异地跨机房调用出现的网络延迟问题，需要能够保证服务消费方能优先调用本地的服务消费方，这就需要采用服务路由的方式。
- 使用方法：用户选择系统自带标签路由选项，配置系统自带标签为发起方ip，在正则表达式中填写服务消费方的ip字段规则。对于服务提供方，用户可以将ip地址相近的实例归属在同一个部署组上，作为目标部署组，实现优先调用同地机房。

部分帐号内测

- 使用目的：希望配置某些使用者使用的版本为新的内测版本。
- 使用方法：用户可以配置自定义标签为用户id，设置id值的正则表达式计算方式，保证服务消费方发起的请求带有以上条件的流量分配到服务提供方的某个版本号上，实现帐号内测功能。

其他实践

- 在实际的使用中，用户也可以通过服务路由功能，实现优先保护重要服务的运行质量、前后端分离、读写分离等等功能。

10. 微服务网关

微服务网关作为后台微服务的入口，接收外界请求并将请求转发到后端的微服务上。微服务网关封装了系统内部架构，按照系统开发者的意愿开放微服务的API供外界使用者进行调用。微服务网关起到的作用主要有：

- 请求转发作用。请求转发是微服务网关的核心能力。
- 统一的服务治理能力。系统开发者可以在微服务网关上配置服务的鉴权、限流、熔断、负载均衡、服务路由等等能力，统一对后端微服务进行管理，不需要将重复功能冗余配置。
- 过滤器的配置。用户可以配置第三方鉴权、前后端参数映射、标签管理等等能力，甚至可以在网关上自定义过滤器，方便灵活的实现业务逻辑。

TSF提供了基于 Spring Cloud 组件 Zuul 和 Spring Cloud Gateway 来实现的微服务网关能力，支持配置请求的转发、网关层的限流、密钥对鉴权、分组管理、查看日志等等能力，支持使用虚拟机或者容器部署微服务网关。

TSF 提供的微服务网关模块已经与TSF注册中心打通，并且支持客户根据自己的业务诉求和微服务部署形式（容器部署或者虚拟机部署）灵活的配置微服务网关的机器资源。

使用微服务网关之前，需要先协调机器资源，部署微服务网关。用户根据自己的需求，可以部署多个微服务网关，方便不同环境、不同团队使用。

微服务网关的部署

新建微服务网关

1. 登陆腾讯微服务平台TSF，点击组件中心，选择微服务网关 - 网关配置
2. 点击新建网关

使用虚拟机部署微服务网关

1. 填写微服务网关名称，名称不能超过60字符。
2. 选择类型为 “虚拟机部署”。
3. 选择虚拟机集群。
4. 选择虚拟机集群中可用的云主机，云主机当前没有被放在其他部署组中。
5. 选择应用：这里支持两种 Zuul 1 和 Spring Cloud Gateway。
6. 用户可以配置启动参数。需要注意的是，虚拟机部署的微服务网关默认选择 8080 端口，如果需要修改启动参数，需要在启动参数中添加 -Dserver.port = 需要添加的端口。
7. 选择授权可以访问的命名空间。当选中某个命名空间后，微服务网关可以将对应命名空间下的微服务的API导入到网关的路由列表中，没有被网关选中的命名空间无法被网关直接访问。
8. 选择完成并启动或完成。

使用容器部署微服务网关

当用户选择容器部署微服务网关时，需要填写部署网关的CPU、内存限制以及服务监听端口。其他填写内容与虚拟机部署的微服务网关保持一致。

微服务网关的程序包以及镜像已经默认内置，如果希望替换微服务网关需要部署的包或者镜像版本，需要联系管理员在 TSF 运营平台上传包或者镜像。路径：运营平台 - 资源运营 - 应用服务器管理 - 组建列表，在组件 msgw 中进行更新。

微服务网关扩缩容

当微服务网关部署完成后，用户可以根据业务量对微服务网关进行资源的扩缩容。

当用户选择虚拟机部署时，点击操作中的应用扩容和下线实例可以对网关的实例进行增减。或者点击进入网关服务实例列表对实例进行增删。

当用户选择容器部署时，点击操作可以调整容器的节点个数，此时请注意集群中的资源是否足够网关增加资源。

查看网关运行情况

有三种方式查看网关的运行情况

1. 用户可以在网关配置列表中查看网关状态是否是正常还是异常，并查看在线节点数。
2. 用户点击进入网关详情页面，查看服务实例列表，可以查看实例的监控数值以及实例运行状态。对容器部署的微服务网关，还可以查看容器节点的重启次数和运行时间。
3. 用户点击“日志”页面，可以查看网关的启动日志。

删除网关

当某个网关被废弃时，需要将网关删除，首先需要删除网关下所有分组。当网关下所有分组被删除后，网关才可以被删除。

当用户已经依据文档 “微服务网关的部署与配置”将网关成功部署之后，可以开始通过微服务网关管理API。

微服务网关的分组

分组是微服务网关管理API的维度，同一个分组下的API使用相同的鉴权方法，使用相同的一个或者多个密钥进行访问鉴权。每一个分组有一个固定的context 作为访问路径中的path参数。

新建分组

1. 点击组建中心 - 微服务网关 - 分组管理页面
2. 在页面上方选择分组所属的网关
3. 点击新建分组
4. 填写分组名称，最长60个字符
5. 填写访问context。context是用户访问网关管理的某一个API的路径的路径参数。以“/”开头，不能为空，但可以只填写一个 “/”
6. 选择鉴权类型：密钥对鉴权或免鉴权。当选择密钥对鉴权时，请求参数中不带正确的secret id和签名的访问会被拒绝。

删除分组

点击删除按钮可以删除分组，但分组下有API的分组不能被删除。

导入API

1. 点击分组详情
2. 点击API列表，选择导入API
3. 选择API。当列表中的API为空时，可以点击命名空间和微服务的漏斗按钮对命名空间和微服务进行筛选
4. 点击提交

发布分组

当且仅当分组被发布时，才能通过微服务网关访问微服务API。导入API后，需要点击分组列表，将分组发布。若某一个分组不想被访问，需要将分组下线。

访问微服务API

当分组发布后，使用什么样的路径能够访问微服务API呢?

分组下的某个API的访问路径为:

网关的域名或网关的ip+port/分组context/微服务API所在的微服务命名空间名称/微服务名称/API路径

举例：当分组context为 sell 时，所在命名空间为 test-env，微服务名称为consumer，API路径为 /echo/{test}

则访问路径为 域名/sell/test-env/consumer/echo/{test}

点击API路径后面的复制按钮，会自动复制 从context以后的路径。

粘贴后的内容为/sell/test/provider-demo/v1/user/delete/user

微服务网关域名

在用户公有云或私有化使用的场景下，需要用户将部署网关的节点的ip配置在 DNS 或 LB 上，以获取统一的访问网关的统一域名或统一VIP。

若用户没有 DNS 或 LB，也可以通过部署网关的节点的ip和端口来访问网关的某一个节点。对虚拟机部署的网关，可以访问虚拟机节点的内网或外网ip：端口。默认端口为8080。对容器部署的网关，可以访问容器节点所在的机器ip+nodeport端口。但使用这种方式，只能访问网关的某一个节点进行测试，没有起到节点之间负载均衡的作用。

微服务网关的密钥对鉴权

微服务网关支持用户使用密钥对对访问进行鉴权。

使用方法如下：

1. 点击 组件中心 - 微服务网关 - 分组管理 - 访问信息页面
2. 确认微服务网关的鉴权类型为 “密钥对鉴权”
3. 新建密钥
4. 新建后的密钥对会展示在控制台上，可以通过复制按钮进行复制
5. 当密钥需要禁用和更换时，点击 “禁用”和“更换”按钮。

有关如何使用TSF 的 SDK计算签名，请参考 SDK使用文档。

微服务网关的限流

用户可以针对网关下的每一个API配置限流规则设置最大请求次数。

使用方法：

1. 组件中心 - 微服务网关 - 分组管理 - API列表页面
2. 选择 API 进行 “编辑限流规则”
3. 填写最大请求次数

当配置了限流规则后，可以在配置API限流规则的微服务的服务限流列表中，也找到限流规则。

HTTP 请求头

名称	位置	必填	说明	
x-mg-traceid	请求/响应	是	请求响应ID，用于跟踪异常请求调用	
x-mg-secretid	请求	是	开启秘钥对鉴权才需要,授权的SecretID，用于加签	
x-mg-alg	请求/响应	否	开启秘钥对鉴权才需要。 加密算法。 0: hmac_md5 1：hmac_sha_1 2：hmac_sha_256 3：hmac_sha_512	
x-mg-sign	请求/响应	否	开启秘钥对鉴权才需要,签名值	
x-mg-nonce	请求/响应	否	开启秘钥对鉴权才需要,随机数	
x-mg-code	响应	是	响应码	

签名算法

签名算法列表

- Hmac_MD5
- HMAC_SHA_1
- HMAC_SHA_256
- HMAC_SHA_512

签名生成规则

- digetValue = (x-mg-noce)+secretId+secretKey
- signValue = Base64String(签名算法(secretKey, digetValue),"utf-8")

案例说明

签名算法： hmac*md5 secretId: hKhATL/DHV4XZtgeROMrrQ== secretKey: +t9tTMTX4C9HUcE+RKOleg== x-mg-noce: helloworld digetValue =hKhATL/DHV4XZtgeROMrrQ==helloworld+t9tTMTX4C9HUcE+RKOleg==* ***_signValue*** =Base64String(hmac_md5(secretKey,digetValue),"utf-8") = pGIhXf0sZDzTpYVMbKg8xA==

校验规则

请求头中的 x-mg-sign 与服务端根据签名生成规则计算的 serviceSign 进行比对，判断是否通过鉴权。

签名 Java 代码

```
 /**
     * 生成签名
     * @param nonce 随机字符串
     * @param secretId 秘钥ID
     * @param appSecret 秘钥值
     * @param algType 签名算法 {@link AlgType}
     * @return
     */
    public static String generate(String nonce,String secretId, String appSecret, AlgType
algType) {
        StringBuilder signStr = new StringBuilder();
        signStr.append(nonce);
        signStr.append(secretId);
        signStr.append(appSecret);
        String digestValue = signStr.toString();
        byte[] serverSignBytes;
        switch (algType) {
            case HMAC_MD5:
                serverSignBytes = HmacUtils.hmacMd5(appSecret,digestValue);
                break;
            case HMAC_SHA_1:
                serverSignBytes = HmacUtils.hmacSha1(appSecret,digestValue);
                break;
            case HMAC_SHA_256:
                serverSignBytes = HmacUtils.hmacSha256(appSecret,digestValue);
                break;
            case HMAC_SHA_512:
                serverSignBytes = HmacUtils.hmacSha512(appSecret,digestValue);
                break;
             default:
                 throw new UnsupportedException("不支持的鉴权算法: " + algType);
        }
        String serverSign = Base64.encodeBase64String(serverSignBytes);
        if (logger.isDebugEnabled()) {
            logger.debug("签名明文: {}, 签名密文: {}", signStr.toString(), serverSign);
        }
        return serverSign;
    }


    public static void main(String[] args) {
        String secretId = "hKhATL/DHV4XZtgeROMrrQ==";
        String secretKey = "+t9tTMTX4C9HUcE+RKOleg==";
        String nonce = "D7pAR5fqQJhZx1yacuVzdO";
        AlgType algType = AlgType.HMAC_SHA_1;
        System.out.println(SignUtil.generate(nonce,secretId,secretKey,algType));
    }
```

Python 代码

```
# -*- coding: utf-8 -*-
"""
     使用SHA1算法生成签名，入参为secretId SecetKey
"""
# Created by kysonli on 2019/05/09

import sys
from hashlib import sha1
import string
import random
import hmac
import base64
import uuid
seed=
['1','2','3','4','5','6','7','8','9','0','a','b','c','d','e','f','g','h','i','j','k','l','m','n','

nonce=string.join(random.sample(seed,22)).replace(" ","")
signstr=nonce+sys.argv[1]+sys.argv[2]
local_sign_seed = hmac.new(sys.argv[2],signstr , sha1).digest()
sign = base64.b64encode(local_sign_seed)
print ""
print "generate local sign: " +sign
print ""
print "=== http request headers as followed === "
print "x-mg-nonce: "+nonce
print "x-mg-secretid: "+sys.argv[1]
print "x-mg-traceid: "+str(uuid.uuid1())
print "x-mg-alg: 1"
print "x-mg-sign: "+sign
```

物料说明

物料名称	物料说明	备注
msgw-zuul1-1.0.0-SNAPSHOT.jar	zuul1版本网关JAR包	用于部署zuul1版本网关
msgw-scg-1.0.0-SNAPSHOT.jar	spring cloud gateway版本网关JAR包	用于部署spring cloud gateway版本网关
consumer-demo-1.12.0-Edgware-RELEASE.jar	微服务JAR包	用于部署consumer-demo服务
provider-demo-1.12.0-Edgware-RELEASE.jar	微服务JAR包	用于部署provider-demo服务
gen_sign.py	签名生成工具	用于生成鉴权sign,及请求头信息
PostMan脚本样例.postman_collection.json	PostMan请求样例	API鉴权和无鉴权样例，导入postman

使用说明

gen_sign.py

```
python gen_sign.py {secretId} {secretKey}
```

- {secretId} 替换成秘钥ID
- {secretKey} 替换成秘钥KEY
- Example

  ```
  python gen_sign.py hKhATL/DHV4XZtgeROMrrQ==  +t9tTMTX4C9HUcE+RKOleg==
  ```

- 输出内容:

  ```
  generate local sign: FE6nLWwYBDYPmJU2CVtFndUBoyg=
  === http request headers as followed ===
  x-mg-nonce: D7pAR5fqQJhZx1yacuVzdO
  x-mg-secretid: hKhATL/DHV4XZtgeROMrrQ==
  x-mg-traceid: 8a237eba-71b2-11e9-acee-5254001d2da0
  x-mg-alg: 1
  x-mg-sign: FE6nLWwYBDYPmJU2CVtFndUBoyg=
  ```

curl 方式测试

- 测试无鉴权的API

  ```
  curl -X {GET}  -H 'x-mg-traceid:71b2-11e9-acee-5254001-8a237eba'  http://{gatewayIp}:{gatewayPort}/{groupContext}/{namespaceName}/{serviceName}/{apiPath}
  ```

- 测试鉴权API

  ```
  curl -X {POST} -H 'content-type: application/json;charset=utf-8' -H 'x-mg-secretid:{secretid}' -H 'x-mg-sign: {sign}' -H 'x-mg-nonce: {nonce}'  -H 'x-mg-alg:1'  -H 'x-mg-traceid:71b2-11e9-acee-5254001-8a237eba' -d '{具体的报文}' http://{gatewayIp}:{gatewayPort}/{groupContext}/{namespaceName}/{serviceName}/{apiPath}
  ```

  说明：*{var}中的参数请按需替换*

1. 新建分组后，无法导入api

可能原因

- 没有授权命名空间。需要在创建网关的时候授权能够网关可以访问的微服务所在的命名空间。
- 导入API时，是否在筛选框中选择到了对应的微服务名称和命名空间。
- 如果以上两步都已经进行了检查，需要跳转到服务治理页面，查看服务详情对应服务下是否有展示想要导入的API。如没有，且该微服务在线，请检查 Spring Cloud 应用是否依赖了 swagger - tsf 的依赖，如果是 mesh 应用，是否已经将该API写在 yaml 文件中。

2. 导入API 后，通过微服务网关访问该API访问不通

- 检查分组是否已经发布。
- 检查是否分组是密钥对鉴权的访问权限，如果是，需要带密钥对鉴权的对应请求头。
- 检查似乎否按照给定的访问路径发起了访问 ip：端口/context/命名空间/微服务名称/API路径

11. 数据化运营

服务依赖拓扑包含了查询服务之间相互依赖调用的拓扑关系，查询特定集群特定命名空间下服务之间调用的统计结果等功能。

查询拓扑关系

1. 登录 TSF 控制台。
2. 在左侧导航栏选择【运维中心】>【服务依赖拓扑】，进入服务依赖拓扑界面。
3. 在页面顶部数据中心位置，选择需要查看的服务所属命名空间。
4. 下方按钮选择需要依赖拓扑的时间，近30分钟、近10分钟、近5分钟以及选择特定时间段（**特定时间段的时间跨度最长为32天**）。
5. 选择之后将在下方空白处出现对应的服务依赖调用关系。
    - 灰色的圆圈表示主动调用的服务，箭头表示发出调用。
    - 绿色的圆圈表示成功调用，黄色表示调用失败。
    - 绿色和黄色组成的圆圈，绿色所占的比例是成功调用的比例，黄色为失败的比例。
    - 圆圈中的数字表示平均请求耗时（单位：ms)和请求频率（单位：次/分钟）。
    - 服务间带箭头曲线上的数字表示两个服务间调用的平均耗时。

    在选中时间范围内，consumer-demo 调用了 provider-demo 服务，调用成功比例为100% （绿色部分）。其中平均每次调用耗时1.91ms，请求频率为每分钟59.8次。

查询依赖详情

鼠标放置到图上特定位置可以显示调用依赖详情。

- 放置在服务间的依赖线条上，弹出面板显示被调用者和调用次数统计，单击弹出框上的“查看调用链”可以进入到调用链查询界面。
- 放置在服务圈内（白色底），可以展示被调用次数，单击弹出框上的“查看调用链”可以进入到调用链查询界面。

查看监控

> ?要使用该功能，需要更新到1.12.0版本之后的 agent 和 SDK (spring cloud)，否则无法看到调用概览。

单击依赖详情对话框中【查看图表】，侧边弹出半屏的调用概览。调用概览的时间范围和调用链查询的时间范围一致。监控概览页面中包括4部分信息：

- 健康概览：显示按分钟统计的正常请求和失败请求数量。
- 延时概览：显示请求耗时在不同时间段的分配比例。
- 状态码概览：显示不同请求状态码的分配比例。
- 并发概览：显示每分钟统计的连接并发数。

调用链查询

调用链查询用来查询和定位具体某一次调用的情况。使用者可以通过具体的服务、接口定位、IP等查询具体的调用过程，包括调用过程所需要的时间和运行情况。

操作步骤

1. 登录 TSF 控制台。
2. 在左侧导航栏中选择【调用链查询】。
3. 在调用链查询中，设置查询条件，单击【查询】。
    - **时间范围**：支持特定和自定义时间范围选择。特定时间范围包括：1分钟前、10分钟前和30分钟前。
    - **调用服务/调用接口**：单击下拉框，在下拉框中选择服务，可以输入关键字进行搜索。
    - **被调服务/被调接口**：单击下拉框，在下拉框中选择服务，可以输入关键字进行搜索。
    - **仅查询出错的调用链**：勾选后，可以查询系统中的出错业务。
    - **耗时大于**：设置耗时的阈值，可以查询系统中的慢业务。
    - **客户端 IP**：客户端 IP 地址。
    - **服务端 IP**：服务端 IP 地址。
    - **标签**：用户在代码中设置的标签，参考 《参数传递》中设置调用链 Tag。
4. 根据查询结果，可以单击【TraceID】进入具体慢业务或出错业务，查看调用链详情。

调用链详情

您可以根据 TraceID 查询调用链的详细信息。调用链详情是为了定位在分布式链路调用过程中每个环节的耗时和异常（不包含本地方法调用情况，本地方法调用建议使用业务 log 的方式记录）。 通过调用链通常为了解决以下问题：

- 定位耗时较长的服务
- 不合理的调用逻辑（如一次请求多次调用某服务，建议改为批量调用接口）

操作步骤

1. 登录 TSF 控制台。
2. 在左侧导航栏中选择【调用链查询】。
3. 在 TraceID 查询中，在搜索框中输入目标 TraceID，单击【查询】。 如果搜索有结果，页面将显示调用链每个环节的耗时和状态。
4. 将鼠标移动到每个环节的时间条上并单击，会弹出 Span 的详细信息。Span 包含三部分信息：
    - 基本信息：显示 Span 名、Span ID、状态和阶段耗时信息。

- 标签：显示系统和业务自定的标签
- 自定义 Metadata：显示用户在代码中设置的自定义元数据信息。参考开发手册中《调用链》，了解如何设置 Metadata。

调用链与业务日志联动

目前要实现调用链与业务日志联动的前提条件是部署组关联的日志配置项的日志格式必须是 Spring Boot 日志格式。 在调用链详情页内，单击日志的 icon，可以查看与这条调用链相关的业务日志。

12. 日志服务

日志服务为用户提供一站式日志服务，从日志采集、日志存储到日志内容搜索，帮助用户轻松定位业务问题。用户通过指定部署组的日志配置项来指定日志采集规则，TSF Agent 根据日志配置项采集指定路径下的文件日志，并上传日志到日志存储模块。用户可以通过 TSF 控制台查看部署组实时日志，并根据关键词来检索日志。

下图是用户在 TSF 平台上使用日志服务的流程图。

日志服务原理

日志采集

通过 Agent 来采集日志，无须手动安装。

日志索引与查询

- 实时索引：采集的日志数据建立索引。
- 查询灵活：支持全文检索、关键词检索（短语或者分词）。

您可以通过以下四种方式，在控制台上查看应用程序实时打印的日志。

查看标准输出（stdout）日志

假设用户已经完成了通过部署组部署应用的操作，以下为查看日志的步骤：

1. 在 TSF 控制台，单击左侧导航【部署组】。
2. 在部署组操作栏中，单击【查看日志】。

使用 Spring Boot 默认日志格式

1. 在 TSF 控制台【日志配置项】界面，创建日志配置项 `consumer-demo-log`，选择日志格式 **Spring Boot**，采集路径为 `/var/root.log`。
2. 应用程序打印日志到指定目录，在配置文件（如 application.yml）中设置打印文件日志的路径，和步骤1中日志配置项的日志采集路径保持一致。

```
logging.file=/var/root.log
```

3. 程序打包后，在控制台上新建部署组，选择日志配置项 `consumer-demo-log` 。
4. 部署应用。
5. 在部署组操作栏中，单击【查看日志】。

使用自定义 logback 日志格式

1. 在 TSF 控制台【日志配置】界面创建日志配置项 `log-config`，选择日志格式为**自定义 logback**，设置解析规则和日志路径如下图：
2. 应用程序打印日志到指定目录，在配置文件 `logback.xml` 中配置日志的 Pattern。

```
<configuration scan="true" scanPeriod="60 seconds" debug="false">
    <property name="log.path" value="/var/root.log" />
    <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} %contextName [%thread] %-5level
%logger{36} - %msg%n</pattern>
        </encoder>
    </appender>
</configuration>
```

3. 程序打包后，在控制台上新建部署组，选择日志配置项 `log-config` 。
4. 部署应用。
5. 单击部署组操作栏中【查看日志】，选择日志配置项 `log-config` 查看日志信息。

应用程序不配置日志路径

如果用户不想配置日志的打印路径，但是仍然希望将日志采集后做检索，可以使用 TSF 提供的默认日志打印和采集功能。该功能要求工程满足以下条件：

- 该工程为 Spring Cloud 应用。
- 工程的配置文件中**不要设置** `logging.file` 参数（如果用户设置了该参数，默认日志功能不会生效）。
- 在 pom.xml 中依赖 TSF 的 logger 依赖，该依赖会打印日志到默认日志路径。

```
<dependency>
            <groupId>com.tencent.tsf</groupId>
            <artifactId>spring-cloud-tsf-logger</artifactId>
</dependency>
```

当工程满足上面三个条件后，在 TSF 上的操作步骤如下：

1. 在控制台上新建部署组时，选择日志配置项为 `default-log-config`，该日志配置项会通知 TSF agent 去采集默认日志路径下的日志。
2. 部署应用。
3. 在部署组操作栏中，单击【查看日志】，选择日志配置项 `default-log-config` 查看日志信息。

日志配置项用于指定采集日志的规则，包括日志的采集路径和日志解析格式（功能待发布）。用户可以在 TSF 控制台上创建日志配置项，然后将配置项发布到部署组上。同一个部署组可以关联多个日志配置项。

创建日志配置项

1. 登录 TSF 控制台。
2. 单击左侧导航栏 **日志服务>日志配置**。
3. 单击【新建配置】，在弹出框中填写日志配置项信息。
    - 名称：日志配置名称，不超过 60 个字符。
    - 日志格式：目前支持 Spring Boot、自定义 Logback、自定义 Log4j 和 无解析规则。
        - **Spring Boot**：如果应用程序使用默认的 Spring Boot 日志，则选择 Spring Boot 日志格式。
        - **自定义 Logback**：如果应用程序使用 logback 日志配置，设置日志格式为 logback，然后设置解析规则（对应 logback 中的 pattern）。参考 Logback 中关于 Pattern 的介绍。
        - **自定义 Log4j**：如果应用程序使用 log4j 日志配置，设置日志格式为 log4j，然后设置解析规则（对应 log4j 中的 PatternLayout）。需要进一步填写解析规则，可以参考 Log4j 中关于 PatternLayout 的介绍。
        - **无解析规则**：指一行日志内容为一条完整的日志。日志服务在采集的时候，将使用换行符 `\n` 来作为一条日志日志的结束符。为了统一结构化管理，每条日志都会存在一个默认的键值 `__CONTENT__`，但日志数据本身不再进行日志结构化处理，也不会提取日志字段，日志属性的时间项由日志采集的时间决定。
    - 日志路径：可填写一个或者多个日志路径。目前支持**绝对路径具体到日志文件**，如/data/log/2017.log，文件名称允许使用数字、字母、横杠(-)、下划线(_)、通配符(*)和小数点(.)。日志文件必须具有文件后缀，文件名和目录支持通配符，如 `/data/log/*.log` 或 `/var/log/*/*.log`。
    - 描述：日志配置项的描述信息。
4. 单击【提交】按钮。

默认日志配置项

TSF 提供默认日志配置项，格式是 Spring Boot。日志路径包括三部分，相对于应用启动路径：

- `./*.log`
- `./log/*.log`
- `./logs/*.log`

用户可以在微服务应用的配置文件中配置 `logging.file` 为上述任意一个路径，然后在创建部署组时，选择默认日志配置项 `default-log-config`。

> !默认日志配置项仅适用于虚拟机部署的应用。

部署组关联日志配置项

部署组关联日志配置项有两个入口：

- 用户可以在创建应用时将日志配置项关联到应用。
- 在日志配置项列表，右侧操作栏中单击【发布配置】。

创建部署组时选择日志配置项

1. 登录 TSF 控制台。
2. 单击左侧导航栏 **部署组**。
3. 单击【新建部署组】，在弹出框中选择关联的日志配置。
4. 单击【提交】按钮。

日志配置项列表，发布配置

1. 登录 TSF 控制台。
2. 单击左侧导航栏 **日志配置项**。
3. 在日志配置项列表，右侧操作栏中单击【发布配置】。
4. 选择要绑定的部署组。
5. 单击【提交】按钮。

删除日志配置项

当日志配置项没有被其他部署组关联时，才可以删除日志配置项。

1. 登录 TSF 控制台。
2. 单击左侧导航栏 **日志配置**。
3. 单击日志配置项列表右侧的 【删除】按钮。
4. 在删除确认框中单击【确认】。

操作场景

日志告警功能允许您通过配置业务日志中的关键词，设置关键词出现频率的告警。您需要在 TSF 控制台上配置需要告警的关键词和监控对象，并在云监控界面上配置告警通知人。

前提条件

要触发日志告警，您需要先完成以下操作：

1. 已采集日志数据（参考日志服务 《快速入门》）
    - 部署组已经关联了日志配置项
    - 程序打印的日志的路径和日志配置的日志路径项保持一致
2. 日志中有告警统计规则中的关键词

操作步骤

新建告警策略

1. 登录 TSF 控制台。
2. 在左侧导航栏，单击【监控告警】。
3. 选择【新增告警策略】。在新增策略界面，填写新建策略内容。
    - 策略名称：20 字以内。
    - 策略类型：选择TSF 日志告警或日志指标告警。
    - 告警对象：选择在已经在 TSF 控制台上配置的告警统计对象（关键词和需要对这个关键词进行统计的部署组）。此处支持多选。
    - 选择告警策略。

      ```
      - 当策略类型选择“TSF日志告警”时，需填写告警关键词，选择指标告警并选择告警策略。
      ```

        - 当策略类型选择“服务指标告警”时，需选择指标告警或事件告警或两者同时选择，并对告警指标进行配置。
    - 告警接收组：添加告警人员。
    - 告警接收渠道：可以通过邮件或短信方式配置告警通知渠道。
4. 单击【完成】。当被监控的对象发生告警时，告警接受组的用户即可在配置的邮件或短信上收到监控信息。

查看告警策略详情

1. 登录腾讯云【云监控控制台】。
2. 在【告警配置】>【告警策略】中，查看当前配置的告警策略列表。
3. 在策略列表中，单击目标策略的策略名称，即可查看策略详情。 在这个页面上，用户可以修改当前的告警触发条件。**当某一条策略被修改时，该策略导致的告警条目在告警列表**

中告警状态将展示为“数据不足”。

4. 在管理告警策略页面底部，您可以填写告警回调信息，填写公网可访问到的 URL 作为回调接口地址（域名或 IP[:端口][/path]），云监控将及时把告警信息推送到该地址。

查看告警列表

1. 登录 云监控控制台。
2. 在左侧导航栏中，选择【告警历史】，进入告警列表页。 在此页面上，可以看到近期的告警信息。其中告警状态表明的是曾经发生的告警在当前的状态是否依然能够触发告警。当显示为 “已恢复” 时，表明此时告警情况已经被修复。

编辑告警对象

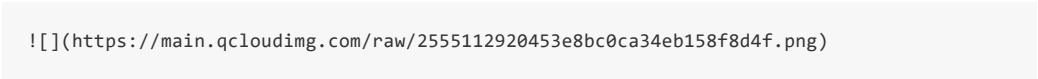
告警对象，即需要配置告警的关键词和部署组，需要在 TSF 控制台 上进行编辑。 在【日志服务】>【日志告警】>【日志告警列表】中，可以对告警对象进行编辑和删除。

TSF 会根据部署组的日志配置项来采集业务日志。例如日志配置项的采集路径是 `/tsf-demo-logs/tsf-inventory/*.log`，TSF 会采集该路径下的文件日志。

TSF 默认提供 stdout 标准输出日志的查看，无须额外设置。

查看实时日志

1. 登录 TSF 控制台，单击左侧导航栏【部署组】。
2. 用户可以单击部署组列表上的 **日志** 图标，查看实时日志。切换日志源查看日志配置项的业务日志或者 stdout 标准输出日志。【日志配置项】查看日志配置项的采集路径下的业务日志。【stdout (标准输出) 】查看实例标准输出日志。
3. 用户也可以单击部署组 ID 进入详情页中，单击 **日志** 标签页查看实时日志。

```

```

操作场景

当部署组关联了日志配置项后，TSF 会对采集的日志数据实时建立索引。用户可以在 TSF 控制台中，通过日志检索功能使用关键词检索出关键日志信息。

操作步骤

1. 登录 TSF 控制台，单击左侧导航栏【日志服务】>【日志检索】。
2. 选择搜索的时间范围。
3. 选择排序方式，默认是【按关键词匹配度排序】。
4. 选择日志数据源
    - 日志配置项：日志配置项决定了日志格式、日志采集路径等规则；
    - 部署组：从关联了日志配置项的部署组中进行选择；
    - 实例：当选择单个部署组时，可以将数据源范围精细化到部署组内某个实例。
5. 输入日志关键词，支持输入多个关键词。检索结果中将包含所有关键词。
6. 单击【查询】。
7. 单击【查看日志上下文】，查看该条日志的上下文信息。

说明：

- 时间范围选择组件的结束时间为进入日志检索页面的时间，而不是检索时刻的时间。如果希望更新结束时间为当前时间，需要刷新页面。
- 日志检索不支持模糊搜索，因此检索时必须是完整的词汇。示例日志如下：

  ```
  2018-10-13 14:26:35.465 INFO [provider-demo,,,] 16595 --- [main]
  trationDelegate$BeanPostProcessorChecker : Bean 'sleuthAdvisorConfig' of type
  [org.springframework.cloud.tsf.sleuth.annotation.SleuthAdvisorConfig] is not eligible
  for getting processed by all BeanPostProcessors (for example: not eligible for auto-
  proxying)
  ```

  如果检索关键词 `auto-proxying` ，则可以检索出结果；如果检索关键词是 `auto` ，则无法检索出结果。

13. 交付中心

为提高用户开发微服务代码的效率， TSF 提供服务编排功能。服务编排可以实现快速生成基于 TSF 框架的微服务工程源码，帮助业务设计人员快速构建可运行的微服务程序。

TSF 引入了服务编排模板的概念用于保存工程信息，方便后续可以基于模板修改工程参数和下载工程源码。

新建服务编排模板

1. 登录 TSF 控制台。
2. 单击左侧导航 **服务编排**。
3. 单击 【新建模板】按钮。
4. 填写工程配置和 POM 配置信息。
    - **工程名**：Spring Boot 工程名，字母开头，支持大小写字母、数字组成，不超过 24 个字符长度。
    - **包路径**：package 路径，小写字母开头，支持小写字母、数字或小数点组成，不超过 60 个字符长度。
    - **GroupID**： pom.xml 文件中的 groupId。
    - **Artifact ID**： pom.xml 文件中的 artifactId。
    - **Name**：pom.xml 文件中的 name。
    - **Version**：pom.xml 文件中的 version。
    - **Description**：选填，pom.xml 文件中的 description。
5. 填写服务基本信息：可填写多个
    - **服务名**：对应工程文件中的 `spring.application.name` 。
    - **端口**：服务监听端口，仅支持 0-65535。
    - **Controller 类名前缀**：可填写多个，将会生成多个带有前缀的 Controller 类。
6. 调用方式：选择 Feign、RestTemplate 或 AsyncRestTemplate。
7. Controller 调用关系（仅服务个数多余 1 时可配置）：可填写多个，将在生成的工程中显示服务之间以 `/echo` 接口的调用逻辑代码。最多支持配置 5 个。
    - 主调服务名：选择主调服务的名称
    - 主调 Controller 类：选择主调服务的 Controller 类。
    - 被调服务名：选择被调服务的名称，不能与主调服务名称重复。
    - 被调 Controller 类：选择被调服务的 Controller 类。
8. 单击【保存并下载】，保存模板，并下载工程的 zip 文件。或者单击【保存】按钮，仅保存模板，不执行下载操作。

> **注意：** 界面将会根据 Controller 调用关系实时生成桑基图，以便用户更直观地看到服务相互间的依赖关系。

修改服务编排模板

1. 单击服务编排模板列表上目标模板的名称。

2. 可修改模板的参数信息，各参数字段含义参考 新建操作。

删除服务编排模板

1. 单击服务编排模板列表上目标模板右侧的【删除】按钮。
2. 在弹框中单击【确认】按钮。

项目整体结构

命名	描述
tsf-projectName	TSF 总工程
projectName-parent	工程父依赖
projectName-common	TSF 公共基础组件（Jar 包插件，非微服务）
projectName-xxx	微服务模块

包命名规范

基础包路径(packageBase)：{orgnization}.{工程名}.{模块名}.{子模块名}

模块内结构：

包	存放内容
packageBase	模型
packageBase.constant	常量，枚举等
packageBase.controller	控制器
packageBase.service	服务接口
packageBase.service.impl	服务实现
packageBase.proxy	远程接口或代理
packageBase.dao	数据层操作
packageBase.exception	异常

项目部署准备

1. 根据《SDK 下载》下载 TSF 相关依赖 (SDK)，并安装到本地 Maven 仓库中。
2. 修改工程的 pom.xml ，检查并修改 `projectName-parent` 工程的 pom.xml 的 **依赖项** 和 **版本号** 与步骤 1 中下载的依赖项相同。
3. 参考官网文档 《搭建本地轻量级服务注册中心》进行本地联调。

项目访问路径

被调服务开放地址：ip:port/被调服务controller前缀/{str} 主调服务访问地址：ip:port/主调服务controller前缀/{str}

生成工程的 SDK 版本说明

目前服务编排功能使用的 TSF SDK 可能和最新的 SDK 不一致，如果您希望使用最新的 SDK，可以参考开发手册 《SDK 下载》将最新依赖包安装到本地，并更新工程的 `pom.xml` 文件。

14. 分布式事务

TSF 提供金融级别高可用分布式事务能力，保证大规模的分布式场景下的业务一致性。TSF 框架下的分布式事务基于 TCC（Try、Confirm 和 Cancel 的简称）模式，支持跨数据库、跨服务的使用场景。为金融、制造业、互联网等行业客户保驾护航。

TCC 模式

TCC 模式是一种补偿性分布式事务，包含 Try、Confirm、Cancel 三种操作。其中 Try 阶段预留业务资源，Confirm 阶段确认执行业务操作，Cancel 阶段取消执行业务操作。 TCC 模式解决了跨服务操作的原子性问题，对数据库的操作为一阶段提交，性能较好。因此，TCC 模式是现今被广泛应用的一种分布式事务模式。

配置 TCC 事务

1. 添加依赖

通过 Maven 引入依赖，在项目的 pom 文件添加下述配置项：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>tcc-transaction-core</artifactId>
    <version>1.10.1.TSF-RELEASE</version>
</dependency>
```

2. 启用 TCC 事务

在 Spring Cloud 的启动类加入注解 EnableTcc：

```
@SpringBootApplication
@EnableTcc
public class App {
  public static void main(String[] args) {
      SpringApplication.run(App.class);
  }
}
```

3. 添加注解

您可以在事务的入口函数上添加 @TsfTcc 注解。建议将注解加在接口定义的方法上。 TsfTcc注解有如下属性：

- serviceName：事务所属的服务名，必选。
- type：事务类型，TransactionType.ROOT表示主事务，TransactionType.BRANCH表示子事务，默认值为Root，可选。
- timeout_ms：事务的超时时间，属性可选，默认为60秒，可选。
- confirmMethodName：confirm方法名，主事务可选，子事务必选。
- cancelMethodName：cancel方法名，主事务可选，子事务必选。
- autoRetry：事务超时后，是否由服务器托管继续自动重试。

定义主事务

主事务函数只需要定义一个事务入口函数即可。type 选择 Root类型；函数抛出 Throwable 异常，返回值没有特殊要求；建议业务将所需的参数都封装成一个对象，只使用这一个对象作为入参即可（所有的参数必须实现 Serializable 接口），一个主事务函数定义的样例如下：

```
public class MainTransaction {

    @TsfTcc(serviceName = "myTcc", type = TransactionType.ROOT, timeout_ms = 60000)
    public String beginTcc(MyParams params) throws Throwable {
        //这里写业务逻辑，调用子事务函数
    }
}
```

定义子事务

一个完整的子事务需要定义3个方法：**Try**、**Confirm**、**Cancel**。 其中**只有 Try 方法需要加 TsfTcc 注解**，在 TsfTcc 标签中指定对应 Confirm 和 Cancel 方法名即可。

子事务函数定义约束如下：

- Try、Confirm、Cancel 的前两个参数必须为 String txId 和 long branchId(在主事务调用子事务的时候，这两个参数分别为 null 和0即可)，其中 txId 为主事务 ID，全局唯一；branchId为子事务Id，用于区分子事务的父子关系和子事务之间的调用顺序
- Try 函数返回 Throwable 异常；Try 函数的返回值为 void，TCC 通过异常返回失败结果。
- Confirm 和 Cancel 函数返回值为 Boolean 类型，操作成功返回 true，操作失败返回 false。

示例：子事务函数定义

```
public interface SubService {

    @TsfTcc(serviceName = "subService", type = TransactionType.BRANCH, confirmMethodName =
"subConfirm", cancelMethodName = "subCancel")
    public void subTry(String txId,long branchId,MyParams params) throws Throwable;

    public boolean subConfirm(String txId,long branchId,MyParams params) throws Throwable;

    public boolean subCancel(String txId,long branchId,MyParams params) throws Throwable;

}
```

示例：主事务与子事务结合

```
public class MainTransaction {

    SubService subService;

    @TsfTcc(serviceName = "myTcc", type = TransactionType.ROOT, timeout_ms = 60000)
    public String beginTcc(MyParams params) throws Throwable {
        subService.subTry(null,0,params);
    }
}
```

检查事务调用

启动服务，从外部调用主事务函数，调用完成之后查看业务日志，一次成功的事务调用日志如下：

```
14:21:30.470 INFO [main] executeLog - begin transaction: 775025cf-d3e6-3fe8-8c8f-31d44fcde46d
14:21:30.649 INFO [main] executeLog - transaction: 775025cf-d3e6-3fe8-8c8f-31d44fcde46d branch
index 1 succ,retry times: 0
14:21:30.652 INFO [main] executeLog - confirm 775025cf-d3e6-3fe8-8c8f-31d44fcde46d branch
index: 1, times: 1
14:21:30.653 INFO [main] executeLog - commit 775025cf-d3e6-3fe8-8c8f-31d44fcde46d branch index:
1 succ
14:21:30.653 INFO [main] executeLog - transaction:775025cf-d3e6-3fe8-8c8f-31d44fcde46d confirm
succ
```

子事务访问外部服务

在 SpringCloud 中，您可以通过集成 FeignClient 访问外部服务，再次以上述的子事务为例，假如子事务需要调用一个代金券服务的相关接口，则配置如下：

```
@FeignClient(value = "couponService")
@RequestMapping(value = "/api/v6/data/couponService")
public interface SubService {

    @TsfTcc(serviceName = "subService", type = TransactionType.BRANCH, confirmMethodName =
"subConfirm", cancelMethodName = "subCancel")
    @RequestMapping(value = "/try", method = RequestMethod.POST)
    public void subTry(String txId,long branchId,MyParams params) throws Throwable;

    @RequestMapping(value = "/confirm", method = RequestMethod.POST)
    public boolean subConfirm(String txId,long branchId,MyParams params) throws Throwable;

    @RequestMapping(value = "/cancel", method = RequestMethod.POST)
    public boolean subCancel(String txId,long branchId,MyParams params) throws Throwable;
}
```

在接口上增加了如下注解：

- `@FeignClient(value = "couponService")`，value 的值为外部服务在服务注册中心注册的服务名。
- `@RequestMapping(value = "/api/v6/data/couponService")`，value 的值为外部服务的根 URL。

在方法上增加了如下注解：

- `@RequestMapping(value = "/try", method = RequestMethod.POST)`，value 的值为外部服务的 API 的 URL 和请求方式。

由于分布式事务框架本身会针对异常的调用做重试，业务方在调用事务相关接口（包含主事务和子事务）的时候请关闭请求重试策略。这里以 Spring Cloud 标准的 FeignClient 和 Ribbon 为例，指导如何关闭请求重试策略。

- FeignClient 默认是不会进行重试的，如果业务自定义了 FeignClient 上面的 retryer，取消自定义配置即可。
- Ribbon 默认的配置是，当请求失败时，会选择另外一个可用的服务实例进行重试，因此需要手动配置 Ribbon 重试策略。
    1. 为项目创建一个 Spring boot 的标准配置文件 application.properties，如果项目已经有 Spring boot 的 properties 文件，可以不额外创建，直接添加内容即可。
    2. 指定关闭某个服务的自动重试策略。在配置文件中添加配置项：

        ```
        couponService.ribbon.MaxAutoRetriesNextServer=0
        couponService.ribbon.MaxAutoRetries=0
        ```

        上述配置表示当调用服务名为 couponService 的服务时，关闭 Ribbon 的自动重试，服务名根据业务实际情况进行填写。

分布式事务的超时和重试机制

用户可以在主事务的注释中定义超时时间，默认超时时间为60秒，此时的超时时间为整个事务的超时时间。当事务超时后，事务管理器对事务进行接管，以每5秒一次的重试频率自动触发重试，重试频率逐渐增长，直到600秒。用户可以在控制台上控制中断这一重试过程，也可以手动触发重试。7天后，重试自动终止。

对于未超时的事务，Try 阶段自动重试3次，3次重试不成功自动触发 Cancel 进行回滚。对于 Confirm 或者 Cancel 阶段始终不能执行成功的情况，会重试直到超时。超时后由事务管理器接管。

查询事务信息

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【事务管理】。
3. 在事务管理界面，选择需要查看的事务时间段、事务状态，并填写关键词。关键词可以选择事务相关的服务名、方法名称。
4. 单击【查询】按钮，可查看事务 ID、起始节点、超时时长等相关信息。
5. 在查询结果列表中，单击主事务 ID，可以查看主事务下的子事务运行状态，包括子事务的方法名、服务名、节点、状态、起止时间等信息。 每一次子事务发起的请求都会记录在子事务列表中，包含子事务的确认、取消、重试等。当子事务发生重试时，您可以查看子事务重试原因（由主事务触发、由事务管理器启动发起或者由用户在控制台上手动触发）。
6. 在子事务列表中，单击操作列的【查看事务参数】，可以查看每条子事务的参数内容。

处理已超时事务

当主事务的处理时间超过了 SDK 中设置的超时时限后，系统会认为事务已处于超时状态。针对超时事务，TSF 提供了两种重试渠道：自动重试和手动重试。

自动重试

当事务超时时，框架自动重试，重试时间间隔从5秒/次倍速递增直到600秒/次，当超时时间超过7天后，会自动停止重试。

手动重试

用户可以在控制台上，通过鼠标点击的方式触发重试。

1. 登录 TSF控制台。
2. 在左侧导航栏中，单击【事务管理】。
3. 批量选择需要进行手动重试或需要停止/触发自动重试的事务，单击【手动触发重试】。

业务场景

本样例是常见的一个线上的代金券/现金协同购物场景。在进行购物的时候，消费用户可以通过使用代金券来抵消一部分的现金费用：用户在消费的时候出示一张2元的代金券，在购买价值20元的物品的时候，只需要从微信钱包中支付18元即可。 在整个购物事务场景中，假设涉及三个的不同子服务：代金券服务、微信钱包服务以及商家账务服务。各个子服务部署在不同的节点上，使用不同的数据库。本样例展示了如何使用 TCC 来完成一次跨服务/跨数据库的分布式事务。

样例模块说明

下载样例 >> 下载 Demo 样例工程之后，进行解压，项目结构如下：

```
tcc-transaction-sample
    |__sample-tcc-consumer
    |__sample-tcc-couponService
    |__sample-tcc-mainService
    |__sample-tcc-transferService
    |__sample-tcc-walletService
    |__tcc-test-simple
    |__init_database.sql
```

lib：分布式事务需要依赖的 jar 包。 **sample-tcc-consumer**：消费用户 Client，发起购物事务。 **sample-tcc-couponService**：代金券子服务，处理代金券使用流程。 **sample-tcc-walletService**：微信钱包子服务，处理消费用户钱包消费流程。 **sample-tcc-transferService**：商家账务子服务，转账给对应商家。 **sample-tcc-mainService**：购物服务，购物事务入口，协调执行3个子服务。 **tcc-test-simple**：完整的简易 Demo，只需要一台服务器就能运行，无需安装 MySQL。 **init_database.sql**：初始化样例数据库。

公有云构建方法

1. 下载 Demo 压缩包并解压，在 Demo 的根目录下执行以下命令。

   ```
   mvn clean package
   ```

   等待 maven 打包成功之后，在以下目录找到 jar 包，用于公有云部署。

   - sample-tcc-couponService/target
   - sample-tcc-mainService/target
   - sample-tcc-transferService/target
   - sample-tcc-walletService/target
   - tcc-test-simple/target

2. 在公有云上申请机器，并按照 couponService、transferService、walletService、mainService 的顺序部署应用。 如果您只需简单试用，可以只部署 tcc-test-simple，跳过以下3、4、5步骤，执行步骤6。

3. 在 mainService 上安装 MySQL 和 ubuntu 服务器。

```
sudo apt-get install mysql-server
sudo vim /etc/mysql/mysql.conf.d/mysqld.cnf //修改bind-address为0.0.0.0
service mysql restart
```

4. 配置 MySQL 远程登录权限，并初始化数据库。

```
mysql -u root -p
grant all privileges  on *.* to root@'%' identified by "root";
create database demo;
use demo;
source init_database.sql  //需要拷贝到机器上去
```

5. 配置数据库地址。 由于完整版本 Demo 需要连接 MySQL 数据库，因此您需要对每个服务配置数据库的地址。您可以使用 TSF 的 《应用配置》 功能进行配置，每个服务的配置文件是各个服务的 resources 中的 application.yml，例如 couponService 的配置文件如下。只需要按照实际修改 MySQL 的 url 即可。

```
spring:
application:
 name: "couponService"
datasource:
 url: jdbc:mysql://127.0.0.1:3306/demo
 username: root
 password: root
server:
tomcat:
 basedir: servlet
port: 8082
address: 0.0.0.0
```

6. 在安装了 mainService 的机器上，通过 HTTP 请求 mainService。如果 mainService 所在的实例有公网 IP，也可以直接通过公网 IP 访问。

```
curl -X POST \
http://localhost:8083/buy \
-H 'Cache-Control: no-cache' \
-H 'Content-Type: application/json' \
-d '{
"couponId" : "9",
"userId" : "1",
"merchantId" : "2",
"money" : "20",
"couponValue" : "5"}'
//tcc-test-simple命令
curl -X POST \
http://localhost:8083/buy \
-H 'Cache-Control: no-cache' \
-H 'Content-Type: application/json' \
-d '{
"from":"a",
"to":"b",
"money":1
}'
```

15. 最佳实践

当用户需要上线新的功能时，希望使用灰度发布的手段在小范围内进行新版本发布测试。 TSF 支持通过部署组和服务路由来实现灰度发布。

场景说明

consumer 调用 provider ， consumer 有两个版本 v1 和 v2-beta，其中 v2-beta 是测试版本。consumer 首先将90%的请求分配给 provider v1 版本，剩下的10%分配给 v2-beta 版本。如果发现 v2-beta 版本运行正常，则增加该版本的流量比例。

前提条件

- 已经下载基于 TSF Spring Cloud SDK 或者 Mesh 编写的代码程序包。
- 已经创建了集群和命名空间，集群中导入2个云服务器。
- 已经创建了 consumer 和 provider 应用（虚拟机部署方式），同时已经创建并部署了 consumer 部署组。

操作步骤

一、provider 创建2个部署组

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【应用管理】，进入应用列表页，选择 provider 应用，进入应用详情页。
3. 单击顶部【程序包管理】，上传 v1 和 v2-beta 的程序包。
4. 单击顶部【部署组】，在部署组列表页面创建两个部署组：
    - 部署组 provider-group-1，添加1个实例，实例规格为1核1G。
    - 部署组 provider-group-2，添加1个实例，实例规格为1核1G。
5. provider-group-1 部署程序包 v1。
6. provider-group-2 部署程序包 v2-beta。

二、配置服务路由规则及初始路由权重 90:10

1. 在左侧导航栏中，单击【服务治理】，进入服务列表页，单击 provider 服务，进入服务详情页。
2. 单击顶部【服务路由】， 新建路由规则，流量来源为 `主调服务名等于consumer-demo` ，流量目的地如下图所示，设置 provider-group-1的权重为90，provider-group-2 的权重为10。
3. 在服务路由规则列表中，单击生效状态列的图标启动该规则，稍等几分钟，刷新页面观察列表下方流量比例变化情况，如果发现 provider-group-1 和 provider-group-2 的流量比例接近 90:10，说明路由规则生效。

三、修改路由权重为 50:50

当 v2-beta 版本服务已经正常运行一段时间后，逐步增加 v2-beta 版本的流量比例，并减少 v1 版本的流量比例。

1. 在服务路由规则列表中，单击【编辑】。
2. 修改流量目的地中 provider-group-1 和 provider-group-2 的流量比例分别为50和50。
3. 稍等几分钟，刷新页面观察列表下方流量比例变化情况，如果 provider-group-1 和 provider-group-2 的流量比例接近 50:50，说明路由规则生效。

四、修改路由权重为 10:90

逐步增加 v2-beta 版本的流量比例，并减少 v1 版本的流量比例。操作方法参考 步骤三。

五、修改路由权重为 0:100

1. 修改路由规则，将 provider-group-1 的权重设置为100。
2. 稍等几分钟，刷新页面观察列表下方流量比例变化情况，如果只有 provider-group-1 有流量，说明路由规则生效。
3. 此时可以停止部署组 provider-group-2 。

使用说明

在真实使用场景中，有以下几点需要考虑：

- 评估单个 provider 服务实例可以处理多少请求，并根据流量分配来规划部署组的实例数量。
- 如果流量比例变化后，可能需要调整部署组的实例数量来满足新的流量比例。
- 可以通过设置**弹性伸缩规则**来支持动态调整实例数量。

功能概述

命名空间、集群有如下特点：不同集群可关联同一命名空间，服务支持跨集群访问，详细说明请参考 《命名空间》。

以 consumer 服务调用 provider 服务为例说明服务跨可用区访问的原理。在上方示意图中有两个集群，分别位于广州一区和广州二区。集群1分别部署了 consumer 和 provider 微服务，集群2部署了 provider 微服务。两个可用区的服务都会注册到同一个注册中心集群。

- 当**开启就近路由**时，广州一区的 consumer 会优先调用同一可用区的 provider。
- 当**开启就近路由**时，当广州一区的 provider 不可用时，consumer 会跨可用区调用广州二区的 provider。
- 当**关闭就近路由**时，广州一区的 consumer 会从两个可用区 provider 中随机选择实例进行调用。

就近路由原理

微服务的路由模块会将可用区作为一种系统标签，当**开启就近路由**时，服务消费者会将请求流量**全部**路由到相同可用区标签的服务提供者，只有当同一可用区服务提供者不可用时，才会进行跨可用区的服务调用；当关闭就近路由时，服务消费者会随机选择不同可用区的服务提供者实例进行调用。

关于服务路由的更多说明参考 《服务路由基本原理》 和《服务路由使用方法》。

最佳实践

跨可用区容灾场景下的就近路由

场景：当**开启就近路由**时，广州一区的 consumer 会优先调用同一可用区的 provider ，只有当广州一区的 provider 不可用时，consumer 才会跨可用区调用广州二区的 provider。

步骤一：准备资源

1. 登录 TSF 控制台。
2. 在左侧导航栏中，单击【集群】，进入集群列表页，单击【新建集群】按钮。
3. 新建虚拟机集群 cluster1，【所在可用区】选择广州一区，设置其他属性。
4. 新建虚拟机集群 cluster2，【所在可用区】选择广州二区，设置其他属性。
5. 集群 cluster1 和集群 cluster2 分别导入所在可用区的云服务器。
6. 在集群 cluster1 的集群详情页 >【命名空间】标签页，单击【新建】按钮，新建命名空间 dev-ns。
7. 在集群 cluster2 的集群详情页 >【命名空间】标签页，单击【新建】按钮，新建命名空间 dev-ns。

8. 确认命名空间 dev-ns 的就近路由开关默认是开启的。

步骤二：部署应用

1. 在左侧导航栏中，单击【应用管理】，进入应用管理页，单击【新建应用】按钮。
2. 新建应用 consumer-app 和 provider-app。
3. 在应用详情页 >【程序包管理】页面上传 Demo 程序包，Demo 获取请参考 《Demo 工程概述》。
4. 进入 conumser-app 应用详情页 >【部署组】标签页。 新建部署组 consumer-group， 属于集群 cluster1，命名空间 dev-ns。添加实例，部署 consumer-demo 程序包。
5. 在 provider-app 应用详情页 >【部署组】标签页
    - 新建部署组 provider-group-03，属于集群 cluster1，命名空间 dev-ns，添加实例，部署 provider-demo 程序包。
    - 新建部署组 provider-group-04，属于集群 cluster2，命名空间 dev-ns，添加实例，部署 provider-demo 程序包。

关于部署的详细操作参考 《虚拟机应用部署组》。

步骤三：验证就近路由

consumer 和 provider 部署成功后，观察服务依赖拓扑图是否出现，如果出现说明服务间调用正常。

1. 在控制台左侧导航栏，单击【服务治理】。
2. 单击 provider-demo 进入微服务详情页，单击【服务路由】标签页的流量详情中可以观察来自 consumer 的请求是否全部路由到部署组 provider-group-03 中，如果请求全部路由到部署组 provider-group-03 证明就近路由功能生效。

步骤四：验证容灾

1. 停止部署组 provider-group-03。
2. 观察 provider-demo 服务路由页面的流量详情中可以观察到流量分配到部署组 provider-group-04，证明容灾能力生效。

步骤五：关闭就近路由

1. 启动部署组 provider-group-03，观察到流量分配回 provider-group-03。
2. 在命名空间页面关闭就近路由开关，观察到流量均匀分配到2个部署组。

就近路由与路由规则

开启就近路由后，服务消费者会按照路由规则优先调用同一可用区的服务提供者，只有当同一可用区的服务提供者不可用时，会按照路由规则进行跨可用区调用。

场景：集群1和集群2都部署了 provider 的两个版本 v1 和 v2；路由规则是90%的流量分配到 v1，10%的流量分配到 v2；命名空间开启了就近路由。

- 当集群1中 provider 的 v1 和 v2 版本实例都正常时，consumer 会按照路由规则调用同一可用区的 provider。
- 当集群1中 provider 的 v1 实例不正常， v2 版本实例正常时，consumer 会按照路由规则将90%请求发送给可用区二的 v1 ，10%请求发送给可用区一的 v2。
- 当集群1中 provider 的 v1 和 v2 实例都不正常，consumer 会按照路由规则将所有90%请求发送给可用区二的 v1，10%请求发送给可用区二的 v2。

16. 运营端

16. 运营端

运营端功能包括以下几个部分：

- 基础模块管理
- 可用区管理
- 应用服务器管理
- 参数配置
- 自定义图标

一、基础模块管理

模块列表

TSF 模块列表。模块支持模块配置、新增节点、管理节点、部署和查看日志功能。

功能	说明
模块配置	配置 VIP、VPort信息，需要用户准备好 VIP
新增节点	添加模块部署节点
管理节点	查看、移除、新增模块节点
部署	在节点上先卸载脚本，再执行安装脚本
查看日志	查看部署日志

机器列表

该页面用于管理部署 TSF 本身的机器。

功能	说明
添加机器	填写机器的基本信息，IP、用户名、端口和密码
删除机器	只有当机器没有和模块关联时才能删除机器

二、可用区管理

基础可用区

基础可用区用于划分用于部署 TSF 本身的业务机器。

用途：在**基础模块管理>机器列表**页面，添加机器时会选择基础可用区。

业务地域

业务地域表示业务服务器所在地域。类似公有云上广州、北京、上海一级地域概念。

用途：在 TSF 租户端集群列表页面，新建集群时选择地域。

业务可用区

业务可用区是比地域小一级的概念，类似广州一区、广州二区。

用途：在 TSF 租户端集群列表页面，新建集群时选择业务可用区。当命名空间设置为就近路由时，同一个业务可用区内的服务会优先调用。

三、应用服务器管理

应用服务器列表

在 TSF 租户端上使用的**虚拟机应用服务器**目前会展示在应用服务器列表中。

字段	含义
服务器ID	公有云/TCE版本中是instance-id，在独立版中是机器IP
集群ID	应用服务器所在集群的ID
部署组ID	应用服务器所在部署组的ID
IP	机器的IP
类型	虚拟机、容器，目前只有虚拟机
应用类型	普通应用、Mesh应用
Agent 状态	正常、异常
MasterID	Master服务器ID
应用部署组信息	机器上 Java 和 Python 的版本号

应用服务器支持两种操作：

- 更新组件
- 查看日志

组件列表

以下组件是运行在应用服务器上的 TSF 组件

组件类型	含义
tsf-agent	与应用部署、启动、停止等任务相关
Pilot-agent	与 service mesh 相关
Mesh-dns	与 service mesh 相关
envoy	与 service mesh 相关
istioctl	与 service mesh 相关
filebeat	与日志采集，调用链数据采集相关

以下是公共包，会在 TSF 租户端微服务功能中使用

公共包	含义
msgw_zuu1	微服务网关 zuul1
msgw_zuu2	微服务网关 zuul2
msgw_scg	微服务网关 spring cloud gateway

单击组件列表的组件名称，进入组件详情页，支持包上传和删除操作。

更新组件

1. 单击应用服务器操作 **更新组件**
2. 选择对应的组件和版本
3. 单击**提交**按钮。

四、参数配置

参数配置页面提供用户修改三种类型参数：

- 容器平台相关参数
- 告警相关参数
- 腾讯云相关参数

容器平台参数设置

TSF 支持standard、alauda、alauda2、ccs 四种类型的容器参数配置。

- 对于 TCE 上的 TSF，**选择 ccs** 。
- 对于使用 TSF 标准 API 对接的容器平台，选择 standard。
- 对于对接灵雀云 alauda，根据灵雀云版本1.x和2.x，分别选择 alauda和aluada2。

各参数含义参考控制台上参数的含义说明。

告警参数设置

告警参数设置用于配置告警回调接口。

腾讯云参数设置

主要涉及 elasticsearch模块、ctsdb 模块、监控 barad 模块的参数，可根据参数含义和示例填写。

五、自定义图标

修改菜单栏标题

修改租户端菜单栏标题时，需要上传一个 json 文件，json格式是：

```
{
  "tsfTitle": 自定义名称
}
```

修改平台登录图标

修改平台登录图标时，需要上传一个像素为 300*100，格式为 PNG 的图片。

修改浏览器导航栏图标

修改浏览器导航栏图标时，需要上传一个像素为 32*32，格式为 ico 的图片。

修改租户端左上角图标

修改租户端左上角图标时，需要上传一个像素为 32*32，格式为 PNG 的图片。

17. 账号管理及安全

企业开发运维中台账号系统

基本概念

租户：租户是一个资源、人员的组合。在私有云场景下，租户解决的是不同的部门、分公司使用相同的系统或者组建下，确保各个不同的部门、分公司间数据的隔离性。在不同的租户下，使用相同的TSF组件，但集群等资源却相互隔离。每一个租户下都有不同的人员对资源进行管理、维护，上线不同的业务。

项目：在同一个租户下，常常出现不同的团队需要关注IT架构中不同模块的情况，如有些用户需要关注消息队列相关的信息，有些用户需要关注API网关相关的信息。需要对不同的产品、产品下不同的资源进行隔离。在TSF的使用上，不同的项目关注的应用不同，部署组也不相同，不同的项目下也对应不同的人员。切换项目后，原有项目下的应用、部署组不可见，保证资源的隔离和数据的安全。

运营端：运营端关注的是 TSF 自身模块运转的情况，方便管理员定位 TSF 自身异常。

租户端：租户端关注的是用户的业务模块，不包含 TSF 自身平台的运行情况。

租户、项目对资源的隔离情况

	组件（MQ、API网管、TSF等）	TSF集群、命名空间	TSF部署组、应用、分布式配置	TSF上其他服务、规则、资源	人员
租户	同一组件不同租户不共享信息	不同租户下集群、命名空间、虚拟机资源不同	同一租户下共用	同一租户下共用	同一账号可以开通多个租户权限
项目	同一组件不同租户不共享信息	同一项目下集群、命名空间、虚拟机资源共用	不同项目下部署组、应用进行隔离	同一租户下共用	同一账号可以添加到多个项目中

在此说明，当用户仅仅使用 TSF 产品时，同一个租户下共用集群、命名空间，无需切换项目就可以看到全量的集群、命名空间以及相关的节点信息。用户可以将应用部署在不同的项目下，实现应用的隔离，也就是说，部署某一个项目下的应用的开发人员，不能看到其他项目下的应用、部署组，也不能看到不同项目下的应用的弹性伸缩、告警规则、应用配置模块信息。但同一个租户下的不同应用，即使在不同项目中也可以相互调用。如果某项目下的开发人员需要对其他项目下的应用、部署组、配置进行增删改查，需要申请加入其他项目中，并在控制台上的右上角项目标签处进行项目切换。

为了方便用户对线上业务进行全局的运维管理，在同一个租户下，用户默认可以查看项目下所有部署组的日志、查看所有服务构成的服务依赖拓扑图、查询调用链，用户可以针对全局的服务进行服务治理。并使用分布式事务功能。以上提到的功能只需要用户加入租户下的任意一个项目就可以看到全量信息。

人员角色

在腾讯分布式框架中，存在4种不同的角色：平台管理员、租户管理员、项目管理员以及开发运维人员。每种角色拥有不同的使用权限。每一个账号都可以属于多个租户、一个租户下的多个项目。

不同角色对TSF平台的使用权限

角色	TSF运营端	TSF租户端	租户端集群、命名空间	租户端部署组、应用	租户端配置、规则
平台管理员	有权限	需创建用户开通权限			
租户管理员	无权限	有权限	支持增删改	支持所有项目下应用、部署组增删改查	支持增删改查
项目管理员	无权限	有权限	支持查看	支持所属项目下应用、部署组增删改查	支持增删改查
开发运维人员	无权限	有权限	支持查看	支持所属项目下应用、部署组增删改查	支持增删改查

不同角色对权限体系使用权限

角色	租户管理	项目管理	人员管理
平台管理员	支持新建、编辑、查看租户，管理租户下用户	支持新建、编辑、查看项目，管理项目下用户	支持增、删、改、冻结账号，修改账号密码
租户管理员	支持查看租户，管理租户下用户	支持新建、编辑、查看项目，管理项目下用户	支持新建、编辑查看人员账号信息
项目管理员	支持查看租户信息	支持查看项目，管理项目下用户	支持查看人员账号信息
开发运维人员	支持查看租户信息	支持查看项目	无权限

说明

基本设定：

- 同一个账号可以在多个租户下有登陆查看权限，同一个账号可以有多个角色。
- 同一个租户、项目的管理员只有一名。

- 平台管理员可以通过为自己开放租户、项目权限的方式查看租户端使用情况。
- 租户管理员自动包含租户下所有项目权限。
- 同一个应用、部署组只能属于一个项目，需要对某个应用、部署组进行操作需要首先被添加到对应项目中。
- TSF平台贯彻devops思想，不对开发人员、运维人员进行权限区分。

使用指导

当用户使用 TSF 产品时，常常出于管理、安全考虑进行账户、权限管理。

当用户业务处于不同部门、不同团队，服务之间很少发生互相调用，不希望共享集群、命名空间、以及机器资源，可以将这些不同的部门、团队的业务隔离在不同的租户下。此时，不同租户下的服务如果需要调用，只能通过外网 API 网关的形式进行。每一个租户设置一位管理员，管理员将使用 TSF 的人员添加到所管理的租户中。同一个租户只能设置一位管理员。当出现同一个开发人员需要涉及到两个不同租户的业务中，可以将这个开发人员的账号添加到两个不同的租户下，登陆时进行切换。

当用户的业务需要相互调用，或者希望不同线上服务共用同一套集群、命名空间、机器资源，建议将这些业务放在同一个租户下。同一个租户下共用集群、命名空间、节点资源。同一个租户下相同命名空间的下的服务可以互相调用。在某一个租户下，只有租户管理员能够新建集群、新建命名空间、导入删除节点，其他人员对这些资源仅仅拥有查询权限。

在同一个租户下，我们通过“项目”这个概念进一步细化了管理粒度，对应用、部署组和部分配置进行了隔离。在不同的项目中建立的应用、部署组不会展示在其他项目的部署组、应用列表中，应用或者部署组相关的应用配置、告警设置、弹性伸缩规则等也不会展示在其他项目中。项目的分离，适合于相同业务组下不同模块、不同侧重点下的资源管理。如果用户不希望对应用进行分割，我们在每一个租户下默认添加了一个默认项目。

贯彻 DevOps 理念，在 TSF 中并没有对开发人员和运维人员进行权限的划分。任何项目下，都可以看到线上所有租户下服务全量列表，以及服务的服务治理配置信息。用户的全局配置信息也所有人可见，当需要对某些配置设置权限时，可以采取配置加密功能进行加密。对于运维过程中的服务运行情况、日志、调用链，都没有进行项目划分。

用户可以设置研发小组的组长为项目管理员。项目管理员可以将开发运维人员拉入项目，对人员进行管理。然而在控制台权限上，项目管理员与开发运维人员相同，都不能进行集群、命名空间的增删改操作，其他操作均有权限。

运营端是关注部署的 TSF 服务本身的管理运营，运营端初始账号密码默认为qloudAdmin。登陆后默认看到 TSF 支撑环境运行情况和权限管理模块，建议使用者为基础运维人员或总管理员。当使用人员希望登陆到 TSF 租户端时，需要将运营端qloudAdmin添加进某个租户、某个项目中。

账号系统使用流程

租户相关操作

新建租户

1. 点击租户管理标签
2. 点击新建租户
3. 填写租户名称及备注

修改租户信息

1. 点击租户管理页面
2. 点击需要修改的租户
3. 在基本信息页面修改租户名称及备注

设置租户管理员

1. 点击租户管理标签
2. 选择需要设置管理员的租户
3. 点击人员信息
4. 点击编辑管理员
5. 在已经添加在TSF人员列表的人员中选择租户管理员

管理租户成员

1. 点击租户管理标签
2. 选择需要设置管理员的租户
3. 点击人员信息
4. 点击导入成员，在已经导入TSF人员的列表中选择需要导入的成员
5. 点击删除，可以将该成员从租户中删除，该成员账号密码依然生效，但不能登陆到本租户中
6. 需要注意的是，删除租户管理员时，不能再成员列表中删除，需要更换新管理员

项目相关操作

新建项目

1. 点击项目管理标签
2. 点击新建项目
3. 填写项目名称、项目所属租户、备注
4. 点击提交
5. 如果不需要项目隔离资源，TSF会自动创建默认项目

修改项目信息

1. 点击项目管理页面
2. 点击需要修改的项目
3. 在基本信息页面修改项目名称及备注

设置项目管理员

1. 点击项目管理标签
2. 选择需要设置管理员的项目
3. 点击人员信息
4. 点击编辑管理员
5. 在已经添加在改租户下面的人员列表中选择需要被添加的项目管理员

管理项目成员

1. 点击项目管理标签
2. 选择需要设置管理员的项目
3. 点击人员信息
4. 点击导入成员，在已经导入项目所在租户的租户人员的列表中选择需要导入的成员
5. 点击删除，可以将该成员从项目中删除，该成员账号密码依然生效，但不能登陆到本项目中
6. 需要注意的是，删除项目管理员时，不能再成员列表中删除，需要更换新管理员

账号信息与安全相关操作

查看账号信息

1. 点击账号信息与安全标签
2. 查看账号信息，包含登陆账号、id、所在租户、角色等信息

修改账号信息

1. 点击账号信息与安全标签
2. 点击所在租户右侧的编辑
3. 修改用户本人的用户名称、电话、邮箱
4. 点击登陆密码右侧的编辑按钮，修改登陆密码，修改后需要重新登陆
5. 平台管理员信息写在后台数据库，不能通过控制台修改

人员管理相关操作

新建账号

1. 新建帐号前，需要先创建租户和项目
2. 点击人员管理标签
3. 点击新建账号
4. 选择希望账号持有人所在的租户和项目
5. 填写账号、名称、手机号、邮箱和初始密码
6. 点击提交

锁定、删除账号，重置密码

1. 点击人员管理标签
2. 选择希望锁定、删除、重置密码的账号
3. 点击对应账号右侧的操作按钮
4. 锁定账号后，账号不能登陆，但账号权限不被删除，解锁后权限还原；删除账号后，账号权限信息全部删除

编辑账号

1. 点击人员管理标签
2. 点击希望编辑的账号
3. 在账号详情页中所在租户页面点击编辑，修改账号名称、手机号、邮箱
4. 同一个账号在不同租户下运行使用不同名称、手机号、邮箱

常见操作流程

第一次使用 TSF

1. 使用qcloudAdmin作为账号密码登陆 TSF 平台
2. 在租户管理中新建一个租户
3. 在人员管理中新建人员，选择刚刚建立好的租户和默认项目
4. 添加多个人员进入 TSF 平台，填写初始账号密码
5. 在新建好的租户下面设置某个人为管理员

6. 使用刚刚创建的账号密码进入 TSF 平台，开始租户端 TSF 使用。

18. 常见问题

虚拟机集群内云主机的可用状态为何显示不可用?

虚拟机集群中的云服务器显示"不可用"状态是因为 agent 无法连接 TSF 后台服务器导致的，因此需要检查 agent 的可用状态。

1. 首先检查云服务器的状态是否为"运行中"。如果云服务器的状态不是运行中，请确保云服务器为开机运行状态。
2. 登录云服务器，执行 `ps aux | grep 'agent'` 命令查看是否有包含 `tsf-agent` 命名的进程。如果没有，请查看步骤3。
3. 切换到 `/root` 目录下，查看是否有 `tsf_agent` 目录。
    - 如果没有，需要将主机从集群中删除后，再重新导入。
    - 如果有，进入 `/root/tsf_agent/ops` 目录下，先执行 `uninstall.sh` ，再执行 `install.sh` 命令。然后通过 ps 命令检查进程。

TSF 是否支持在同一台服务器上安装多个应用?

在 TSF 中，应用的部署有两种类型：

- 云服务器独占实例：在一台云服务器上，仅部署单独一个应用。
- 容器实例：TSF 使用 Docker 容器在一台独立的云服务器上创建多个 Docker 实例，允许在每一个 Docker 实例上部署一个应用。

TSF 应用实例状态为什么显示 Agent 异常?

TSF Agent 会定期上报心跳数据给 TSF 管理模块，如果 Agent 停止上报状态，则某段时间后该机器将会被判定为未知状态。通常而言，该问题是由于 Agent 停止导致。

用户可尝试在云服务器界面，重启该云服务器。

容器部署组执行部署操作时提示内存（或 CPU）不足?

请检查该部署组所在集群和命名空间中的节点的内存（或 CPU）的使用情况，确保在执行部署操作时填写的内存（或 CPU）数值小于剩余内存（或 CPU）资源 。

用户可以在集群的节点列表页面中找到已分配 CPU 和已分配内存信息。

如何排查应用是否部署失败?

1. 在应用详情页，单击部署组操作列的【查看日志】查看 stdout 日志，通过日志初步定位是否是业务程序本身问题。如果没有日志信息，进行步骤2。
2. 单击【变更记录】，查看本次部署任务的 taskid。
3. 登录虚拟机或容器，查看本次任务的日志信息 `/root/tsf-agent/agent/task/<taskid>` ，其中 taskid 是步骤2中的任务 ID。

程序包无法上传如何解决?

当发现程序包无法上传时，请检查浏览器是否设置了代理。用户可尝试换一个浏览器或者切换网络重新进行上传。

在工程的配置文件中，是否需要填写服务注册中心地址？

- 对于本地开发调试的应用，在启动 Java 应用的启动参数中需要填写轻量级服务注册中心 consul 的 IP 和 Port。配置文件（如 application.yml）中则无需填写。
- 对于通过 TSF 平台部署的应用，既不需要在启动参数中设置注册中心的地址，也不需要在配置文件中设置注册中心地址。SDK 会通过环境变量获取注册中心的地址。

应用部署成功后，服务列表中为何没有出现服务?

请检查部署压缩包中是否包含了 `spec.yaml` ，且 `spec.yaml` 的格式是否正确。 如果不存在 `spec.yaml` 或者格式不对，TSF 无法将服务注册到服务注册中心。详情参考 《TSF Mesh 开发指引》。

服务实例显示离线状态如何解决?

在 Mesh 环境下，TSF Sidecar 会定期通过调用服务的健康检查接口获取服务健康状态，并将健康状态上报到服务注册中心。由于某些原因，比如用户健康检查接口信息配置错误、端口配置错误、或者服务实例出现访问失败，则会导致服务不健康。 您可以通过以下步骤进行排查：

1. 查看服务配置信息

- 服务配置信息错误 查看应用的软件包，获取服务配置信息（spec.yaml），检查服务名是否为期望暴露的服务名、端口号是否为服务真实监听的端口号、健康检查接口是否存在、检查健康接口格式是否正确（不含 ip:port，类似 `/health` 是符合的）。
- 服务配置文件格式错误 将 spec.yaml 内容，拷贝到 yamllint 中，校验 yaml 格式是否正确。如果格式正确，则继续检查字段名称，是否与下面示例的格式一致。

```
apiVersion: v1
kind: Application
metadata:
name: service1
namespace: nsTester
spec:
services:
  - name: user
    healthCheck:
      path: /health
    ports:
      - targetPort: 8089
        protocol: http
```

- 服务配置没有挂载到正确的位置
    - 在容器环境下，排查业务是否在容器的启动脚本中，把 spec.yaml 和 apis 目录(可选)，拷贝到挂载路径 `/opt/tsf/app_config` 下面。
    - 在虚拟机环境下，排查业务程序包根目录下面，是否存在 spec.yaml 以及 apis 目录（可选），如果没有，则需要修改。
    - 通用排查方法：
        1. 检查 pilot-agent 加载配置文件的绝对目录。 输入： `curl 127.0.0.1:15020/config/agent|grep appConfigDir` ， 输出： `appConfigDir: /data/demo/shopService` 代表 pilot-agent 会从 `/data/demo/shopService` 中加载 spec.yaml。

2. 确认 appConfigDir 指向的目录存在 spec.yaml。

2. 查看健康检查返回

登录应用所在的容器或者虚拟机，通过调用本地的健康检查路径，查看返回码是否为200，如果不是200，证明服务存在健康问题。详情参考 《TSF Mesh 开发指引》。

```
curl -i -H 'Host: local-service' {ip}:{Port}/<healthCheck_path>
```

3. 通过运维接口查看 sidecar 相关信息

登录应用所在的容器或者虚拟机，调用 pilot-agent 的运维接口，查看 sidecar 容器的相关状态信息。

- 查看接口列表 调用 `GET 127.0.0.1:15020/help` 接口查看接口列表：

```
[root@TENCENT64 ~/pilot-agent/op]# curl 127.0.0.1:15020/help
admin commands are:
GET /health: print out the health info for data-plane
GET /config_dump/{component}: print out the configuration of the component, component
can be pilot-agent/envoy/mesh-dns
GET /help: print out list of admin commands
GET /config/agent: print out the pilot-agent configuration
GET /config/services: print out the services info
GET /config/global: print out the global mesh configuration
GET /config/envoy: print out the envoy startup configuration
GET /version: print out the pilot-agent version
GET /version/{component}: print out the version of the component, component can be
pilot-agent/envoy/mesh-dns
GET /latest_error: print out the latest error info
GET /status: print out the status, result could be <INIT>, <CONFIG_READY>,
<FLOW_TAKEOVER>, <SERVICE_REGISTERED>
GET /epoch/{component}: print out the component restart epoch, component can be
envoy/mesh-dns
POST /hot_restart/{component}: hot restart the component process, component can be
envoy/mesh-dns
PUT /update: Update the component
PUT /logging/{scope}/{level}: dynamic change logging level, scope can be
healthz/admin/ads/default/model, level can be info/warn/error/none/debug
```

- 查看版本号 调用 `GET 127.0.0.1:15020/version` 接口查看版本号：

```
[root@TENCENT64 ~/pilot-agent/op]# curl 127.0.0.1:15020/version
1.0.13.release-20190225_103608
```

- 查看 sidecar 健康状态 调用 `GET 127.0.0.1:15020/health` 接口查看 sidecar 健康状态：

```
[root@TENCENT64 ~/pilot-agent/op]# curl 127.0.0.1:15020/health
{"envoy":{"status":"UP"},"mesh-dns":{"status":"UP"},"status":"UP"}
```

如果出现部件不健康的组件（status 不为 UP），或者接口调用失败，则需要联系后台运维人员处理。

4. 调用 clusters 接口

登录应用所在的容器或者虚拟机，调用 envoy 的 clusters 接口，查看本地 cluster（格式为 in#port#serviceName）是否存在或者健康状态是否 healthy。 输入：

```
curl http://127.0.0.1:15000/clusters
```

输出：

```
in#8080#reporttimeb::default_priority::max_connections::1024
in#8080#reporttimeb::default_priority::max_pending_requests::1024
in#8080#reporttimeb::default_priority::max_requests::1024
in#8080#reporttimeb::default_priority::max_retries::3
in#8080#reporttimeb::high_priority::max_connections::1024
in#8080#reporttimeb::high_priority::max_pending_requests::1024
in#8080#reporttimeb::high_priority::max_requests::1024
in#8080#reporttimeb::high_priority::max_retries::3
in#8080#reporttimeb::added_via_api::true
in#8080#reporttimeb::9.77.7.132:8080::cx_active::0
in#8080#reporttimeb::9.77.7.132:8080::cx_connect_fail::0
in#8080#reporttimeb::9.77.7.132:8080::cx_total::4
in#8080#reporttimeb::9.77.7.132:8080::rq_active::0
in#8080#reporttimeb::9.77.7.132:8080::rq_error::0
in#8080#reporttimeb::9.77.7.132:8080::rq_success::4
in#8080#reporttimeb::9.77.7.132:8080::rq_timeout::0
in#8080#reporttimeb::9.77.7.132:8080::rq_total::4
in#8080#reporttimeb::9.77.7.132:8080::health_flags::healthy
in#8080#reporttimeb::9.77.7.132:8080::weight::1
in#8080#reporttimeb::9.77.7.132:8080::region::
in#8080#reporttimeb::9.77.7.132:8080::zone::
in#8080#reporttimeb::9.77.7.132:8080::sub_zone::
in#8080#reporttimeb::9.77.7.132:8080::canary::false
in#8080#reporttimeb::9.77.7.132:8080::success_rate::-1
```

- 如果 cluster 不存在，则需要联系后台运维人员处理。
- 如果状态不为 healthy，则执行后续步骤。

5. 调用 config_dump 接口

登录应用所在的容器或者虚拟机，调用 envoy 的 clusters 接口，查看本地的路由配置信息：

```
curl http://127.0.0.1:15000/config_dump -o config.json
```

通过 vi 打开 config.json 文件，并查找 in#8080#reporttimeb（本地 cluster，格式为 in#port#serviceName），查看配置中的服务地址(address)、端口(port_value）是否正确。 健康检查信息 health_checks、熔断配置 circuit_breakers 是否正确。如果不正确，且确认服务没有被熔断，则需要联系后台运维人员处理。

```
"dynamic_active_clusters": [
    {
     "version_info": "2018-10-17T11:31:50+08:00",
     "cluster": {
      "name": "BlackHoleCluster",
      "connect_timeout": "1s"
     },
     "last_updated": "2018-10-16T19:31:41.792Z"
    },
    {
     "version_info": "2018-10-17T11:31:50+08:00",
     "cluster": {
      "name": "in#8080#reporttimeb",
      "connect_timeout": "5s",
      "hosts": [
       {
        "socket_address": {
         "address": "9.77.7.132",
         "port_value": 8080
        }
       }
      ],
      "health_checks": [
       {
        "timeout": "5s",
        "interval": "10s",
        "unhealthy_threshold": 2,
        "healthy_threshold": 2,
        "http_health_check": {
         "path": "/health"
        }
       }
      ]
```

如何联系后台运维人员?

您需要提供以下信息，方便运维人员快速定位问题。

- 问题：B 服务注册失败，已初步定位，原因在于下发的配置与服务配置不相符。
- 服务节点信息：A 调用 B 服务，B 服务192.168.3.1，A 服务192.168.2.1。
- 服务类型信息：A 服务为 springcloud，B 服务为 Mesh。

为什么无法显示 stdout 日志?

对于使用容器部署的应用，在下述两种情况下 TSF 会采集 stdout 日志：

- **业务容器启动服务进程时不重定向 stdout** 此时可以登录上容器所在的机器，通过 `kubectl logs` 或者 `docker logs` 观察程序自身有无输出 stdout
- **业务容器启动服务进程时将 stdout 重定向到** `/data/tsf_std/stdout/logs/sys_log.log` 此时可以登录上容器所在的机器，通过 `kubectl exec` 或者 `docker exec` 进入到容器中，观察上述 `sys_log.log` 文件有无内容。

为什么无法显示自定义的文件日志?

1. 检查日志配置项中的日志采集规则
    - 确认日志配置项中的**日志路径**和应用程序在配置文件中的日志路径相同。
    - 确认日志配置项中的**日志解析格式**和应用程序在配置文件中的日志解析格式相同。例如如果日志配置项为默认日志配置项（Spring Boot 日志格式），而应用程序实际使用 logback 和自定义的日志 pattern，那么 TSF 无法采集日志。用户需要创建一个 logback 格式的日志配置项，然后关联到部署组。
2. 在日志配置项的详情页中，查看日志配置项是否已经发布到对应的部署组中。
3. 日志配置项修改后，虚拟机部署的应用会自动使用修改后的日志配置项来采集日志，容器部署的应用需要重启部署组（先停止部署组，再启动部署组）后才会使用修改后的日志配置项来采集日志。
4. 如果容器镜像的时区不对，则无法检索到日志，需要在 Dockerfile 中调整时区。

```
# GMT+8 for CentOS
RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
RUN echo "Asia/Shanghai" > /etc/timezone
```

为什么实时日志一直加载不出来?

1. 请确认日志配置项已正确配置。
2. 实时日志会有15s时延，即显示从15秒前开始打印的日志。如果一直在加载中，可以确认下最近15s有没有日志打印，如果没有则会一直到有新的日志打印才会显示。

默认容器是 UTC 时区，和宿主机的时区不一致如何解决?

保证容器的时区和宿主机一致，避免调用链日志时间收集等有偏差。如果基础镜像时区是没调整过的，那么在编写 `Dockerfile` 时再调整时区，例如基础镜像是 centos，那么微服务 `Dockerfile` 增加以下配置，并重新 build 即可：

```
#GMT+8
RUN /bin/cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
RUN echo "Asia/Shanghai" > /etc/timezone
```

如何保证全局事务的正确执行?

TCC 以主事务函数作为入口，协调控制多个跨库/跨服务的子事务的执行流程，可以保证各个子事务之间的事务特性（一次性 Confirm 或者 Cancel 所有的子事务）。 由于 TCC 不能够保证每个子事务内部的事务性质，因此在使用 TCC 时，建议在每个子事务内部自己执行本地事务，以保证全局事务的正确执行。

业务需要处理什么类型的异常?

主事务函数中主要需要捕获两个异常：

- **TransactionCancelledException**：表示事务失败cancelled。
- **TransactionTimeoutException**：表示事务超时。

运行时业务抛出的异常会被包装在以上两个异常中，您可以通过 `getCause()` 方法获取业务真正的运行时异常：

```
public class MainTransaction {

    SubService subService;

    @TsfTcc(serviceName = "myTcc", type = TransactionType.ROOT, timeout_ms = 60000)
    public String beginTcc(MyParams params) throws Throwable {
        try{
            subService.subTry(null,0,params);
        } catch (TransactionCancelledException e) {
            //获取业务运行时真正的异常
            Throwable t = e.getCause()
            return "request canceled.";
        } catch (TransactionTimeoutException e) {
            //获取业务运行时真正的异常
            Throwable t = e.getCause()
            return "request timeout.";
        } catch (Throwable e) {
            //打印业务异常日志
            return "request exception.";
        }
    }
}
```

业务什么时候会进行回滚?

当前业务只有在子事务 Try 失败时会进行 cancel 操作，**当进入 confirm 阶段之后，confirm 失败只会继续重试 confirm，直到超时为止。**

Netty 版本冲突时如何处理?

1.10.0.TSF-RELEASE 版本的 SDK 依赖4.1.X 版本的 Netty，用户如果使用 TSF 的 SDK 会引入如下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

用户可以通过添加 exclusions 标签，避免引入4.0.x 版本的 Netty：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
    <exclusions>
        <exclusion>
            <groupId>io.netty</groupId>
            <artifactId>netty-codec-http</artifactId>
        </exclusion>
        <exclusion>
            <groupId>io.netty</groupId>
            <artifactId>netty-transport-native-epoll</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

19. 词汇表

资源管理相关

专有名词	对应英文	解释
集群	Cluster	集群是指云资源管理的集合，包含了运行应用的云主机等资源。集群中的云资源可以是容器、可以是虚拟机等。在同一个集群中，资源之间可以互相通信。
命名空间	Namespace	命名空间是对一组资源和对象的抽象集合。命名空间起到的作用是将同一个集群下的资源进行隔离，使用相同的账号创建不同的环境。在同一个集群中可以有多个命名空间。集群中的资源只能属于一个命名空间。在使用中，用户可以将一个集群隔离为开发环境、测试环境等等。
部署组	Deployment Group	部署组是执行应用批量部署的逻辑概念。一个部署组内包括多个实例，每个实例上运行相同的应用程序。同一个部署组上可以运行一个或者多个应用，但其中的每个实例应用相同。

应用生命周期管理相关

专有名词	对应英文	解释
应用	Application	应用是可部署的软件实体，包含一个或一组容器或进程。
TSF Agent	TSF Agent	TSF Agent 是 TSF 中部署在应用机器上的 Daemon 程序，在运行中承担接收和执行扩容等任务、上报机器运行状态、上报应用监控数据等功能；同时也是 TSF 控制台与应用程序之间通信的桥梁。
TSF 应用生命周期管理	TSF Application Lifecycle Management	应用是 TSF 管理的基本单位，TSF 提供了完整的应用生命周期管理能力，可以完成从部署到运行过程的全程管理，包括应用创建、部署、启动、回滚，扩容缩容和停止下线等操作。

服务框架相关

专有名词	对应英文	解释
微服务	Microservice	微服务是一些协同工作的小而自治的服务，具有弹性扩展、简化部署、可组合等优点。一个微服务既可以是服务提供者，也可以是服务消费者。
Spring Boot	Spring Boot	Spring Boot 是一种用于简化 Spring 应用的初始搭建以及开发过程的框架，该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板式的配置。
Spring Cloud	Spring Cloud	Spring Cloud 是一系列框架的集合，利用 Spring Boot 开发简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 Spring Boot 的开发风格做到一键启动和部署。

数据化运营与监控相关

专有名词	对应英文	解释
IaaS 基础监控	IaaS Basic Monitoring	TSF 监控功能之一，能够针对应用的运行状态，对机器的 CPU、内存、负载、网络和磁盘等基础指标进行详细的监控。
应用监控	Application Monitoring	TSF 的监控功能之一，监控应用的QPS, 请求时间和出错情况等指标。
TSF 分布式跟踪系统	TSF Distributed Tracing System	TSF 调用链跟踪能够分析每一次请求的调用路径，帮助用户了解调用链各阶段耗时情况和状态，从而精准发现系统的瓶颈和隐患。

分布式工具相关

专有名词	对应英文	解释
TSF 分布式配置管理	TSF Distributed Configuration Management	TSF 分布式配置管理实现了将分布式系统的配置信息在 TSF 控制台上进行集中管理，可以实时新增、修改、删除配置，并将更新的配置推送到全局或者应用内部。

事务管理相关

专有名词	对应英文	解释
事务	Transaction	事务是指作为单个逻辑工作单元执行的一系列操作。事务的执行有一致性，同一个事务只能同时被操作或不被操作。
主事务	Main Transaction	主事务是事务的发起者。同一个事务只能有一个主事务。
子事务	Sub Transaction	子事务是同一个主事务下的分支。
事务管理器	Transaction Coordinator	事务管理器是一个协调分布式事务、管理事务执行状态的独立服务。