微服务平台 TSF

开发指南

腾讯云

文档目录

开发指南
应用开发概述

最近更新时间：2024-11-05 16:13:52

您可以直接将已有的应用部署到 TSF，但有些功能需要在代码中添加相应的依赖和配置，您也可以重新开发应用，TSF 为您提供应用联调功能和相关工具，帮助您提升应用开发的效率。
新业务开发一般分为技术栈选型，依赖构建，应用开发，测试联调和应用打包几个步骤：

序号	步骤	说明	参见文档
1	技术栈选型	根据您的业务需求和应用场景选择合适的开发语言、开发框架和部署资源类型。	技术栈选型
2	依赖构建	构建依赖，在 pom.xml 文件中定义工程需要的依赖包。	Java 应用开发
3	应用开发	根据您选择的应用类型，参见文档指引开发应用。	Java 应用开发
4	测试联调	应用开发完成后，需要进行测试联调，测试应用的可行性，并且实现本地应用和云端应用的相互调用。	测试联调
5	应用打包	应用开发测试完成后，需要将应用工程进行打包，部署到 TSF 中。	应用打包

TSF 支持 Spring Cloud 原生应用、普通应用和多协议多语言 Mesh 应用，您可以根据业务场景需要开发应用，并部署到 TSF 上。

功能		原生应用		普通应用	Mesh 应用
适用场景		存量业务应用开源 Spring Cloud 零代码改造		新业务全新技术框架选型	适配不同协议（Dubbo、HTTP、gRPC）不同语言接入（PHP、Java、Python）
注册发现		✓		✓	✓
服务治理	服务鉴权	✓	自定义标签需要结合 Mesh 标签实现	tsf-sdk	Mesh 流量劫持
	服务限流	✓	自定义标签需要结合 Mesh 标签实现	tsf-sdk	Mesh 流量劫持
	服务熔断	✓	自定义标签需要结合 Mesh 标签实现	tsf-sdk	Mesh 流量劫持
	服务路由	✓	自定义标签需要结合 Mesh 标签实现	tsf-sdk	Mesh 流量劫持
调用链		• 业务应用 Spring Cloud Sleuth、Zipkin 组件能够接入 TSF 调用链支持服务间调用链不支持方法级调用链 • 业务应用 SkyWalking 能够对接用户自建的 SkyWalking 服务端		tsf-sdk	支持服务间调用链串联
日志服务		✓		✓	✓
配置管理	实时配置（分布式配置）	不支持		tsf-sdk	不支持
	文件配置	支持		支持	支持

增强能力	服务优雅下线	✓	✓	✓
	全链路灰度	结合微服务网关 + Mesh 标签	结合微服务网关 + SDK	结合微服务网关 + Mesh 标签
	单元化	不支持	结合微服务网关 + SDK	不支持

技术栈选型

最近更新时间：2022-07-21 17:03:45

开发框架选择建议

TSF 支持的应用类型有 Java 应用、Go 应用和多语言（如 Python，C++ 等）应用，您可以根据您的业务需求和应用场景选择合适的开发语言、开发框架和部署资源类型，具体的选择建议如下：

支持功能对比

功能		原生应用		普通应用	Mesh 应用
适用场景		存量业务应用开源 Spring Cloud 零代码改造		新业务全新技术框架选型	适配不同协议（Dubbo、HTTP、gRPC）不同语言接入（PHP、Java、Python）
注册发现		✓		✓	✓
服务治理	服务鉴权	✓	自定义标签需要结合 Mesh 标签实现	✓	✓
	服务限流	✓	自定义标签需要结合 Mesh 标签实现	✓	✓
	服务熔断	✓	自定义标签需要结合	✓	✓

			Mesh 标签实现		
	服务路由	✓	自定义标签需要结合 Mesh 标签实现	✓	✓
调用链		spring cloud sleuth、zipkin		✓	✓
日志服务		✓		✓	✓
配置管理	实时配置（分布式配置）	–		✓	–
	文件配置	✓		✓	✓
增强能力	服务优雅下线	✓		✓	✓
	全链路灰度	✓		✓	✓
	单元化	–		✓	–

Java 应用开发
下载 Maven

最近更新时间：2024-01-23 16:36:01

在正式开发应用前，请确保您的机器上已经安装了 Java 和 Maven。

安装 Java

步骤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）。

步骤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。

安装 Maven

步骤1：下载 安装 Maven

参见 Maven 下载。

步骤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
```

步骤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。

Maven 配置 TSF 私服地址

步骤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：
您也可以下载 setting.xml 样例文件（鼠标右键另存为链接）。

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <!-- localRepository
   | The path to the local repository maven will use to store artifacts.
   |
   | Default: ${user.home}/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
  -->

  <pluginGroups></pluginGroups>
  <proxies></proxies>
  <servers></servers>
  <mirrors></mirrors>

  <profiles>
      <profile>
        <id>nexus</id>
        <repositories>
            <repository>
                <id>central</id>
                <url>https://repo1.maven.org/maven2</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </repository>
        </repositories>
        <pluginRepositories>
            <pluginRepository>
                <id>central</id>
                <url>https://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>https://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>https://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>

</settings>
```

步骤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。

```
➜ ~ mvn help:effective-settings | grep 'qcloud-repo'
      <id>qcloud-repo</id>
    <activeProfile>qcloud-repo</activeProfile>
```

> **说明**
> 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。

后续步骤

1. 完成以上步骤后，您可以参见 应用开发 来开发您的应用。
2. 完成应用开发后，您可以参见 应用打包 下载 TSF SDK 并将您的应用工程打包，部署到 TSF 平台。

Spring Cloud（TSF 依赖）
Spring Cloud 概述

最近更新时间：2024-11-07 10:21:33

版本配套关系说明

TSF 目前支持 Spring Cloud Edgware、Spring Cloud Finchley、Spring Cloud Greenwich、Spring Cloud Hoxton、Spring Cloud 2020 、Spring Cloud 2021 及 SpringCloud 2022。Spring Cloud 、Spring Boot 及 TSF SDK 版本之间的关系如下表所示。

Spring Cloud	Spring Boot
2022	3.1.x 3.0.x
2021	2.7.x 2.6.x
2020	2.5.x
Hoxton	2.3.x
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x

⚠ 注意：

- 2020年5月19日起，TSF 主要支持 Greenwich 和 Finchley 以及更新版本的功能迭代更新，Edgware 版本主要进行缺陷修复（ 社区 Edgware 版本 于2019年8月停止更新）。
- TSF 独占注册配置中心仅支持 SpringCloud 2022 版本。

长期维护 SDK 版本

TSF 长期维护 LTS (Long Term Support) 版本，SDK 的第三位版本号会根据缺陷修复递增。详情参见 SDK 版本更新日志 。

SDK 版本号	新增特性
1.46.x （1.40.x）	• 支持微服务网关可扩展性。支持使用 TSF 网关 SDK 的同时，自定义网关路由策略、支持 websocket、支持跨域等原生网关能力 • Oauth 插件支持第三方鉴权地址为微服务 API 的能力 • 支持原生网关使用熔断治理的能力 • 支持服务监听触发回调 • 支持查看下发配置
1.29.x	• 补齐 Spring Cloud Gateway 网关的服务治理能力 • 微服务网关支持托管外部 API • 支持云上 Spring Cloud 应用平滑迁移 TSF • 调用链支持 CMQ、PostgreSQL
1.23.x	• 新增 Spring Cloud Gateway 微服务网关，链路追踪和调用监控 • 微服务网关路径重写配置和微信小程序登录插件 • 调用链支持 RocketMQ

1.21.x	• 支持服务熔断 • 支持服务容错 • 支持全链路灰度发布
1.18.x	• 支持微服务网关（zuul1 版）SDK，基于此 SDK 二次研发，无缝集成 TSF 平台服务治理能力 • 增加 swagger-ui 依赖包 • 调用链支持 MySQL JDBC、Redis、MongoDB、CMQ、Kafka • 支持全局命名空间 • 新增自定义日志配置需要的 Converter 和 Layout 类，支持用户使用自定义 logback\log4j\log4j2 日志配置
1.12.x	• 支持服务限流 • 支持服务路由 • 支持服务鉴权 • 支持分布式配置 • 支持调用链

SDK 版本使用说明

说明：

- Spring Cloud Edgware、Spring Cloud Finchley、Spring Cloud Greenwich、Spring Cloud Hoxton、Spring Cloud 2020 、 Spring Cloud 2021 可支持在 JDK 8、JDK 11、JDK 17（包括 KONA JDK8，KONA JDK11，KONA JDK17）环境下运行。
- Spring Cloud 2022 可支持在 JDK 17（包括 KONA JDK17）环境下运行。

公有云 TSF

对于 TSF 公有云的用户，建议使用 TSF LTS 版本。如果您希望使用最新的产品能力，可以使用最新的版本。

私有化 TSF

对于 TSF 私有云的用户，SDK 版本号需要和 TSF 平台版本**保持一致**，SDK 的缺陷会在第三位版本号上体现，例如用户使用 TSF 1.12.4 版本，推荐使用的 SDK 版本为 1.12.x。

TSF 私有化平台版本	Edgware	Finchley	Greenwich	Hoxton	2020	2021
1.46.x	-	1.40.x-Finchley-RELEASE 1.46.x-Finchley-RELEASE	1.40.x-Greenwich-RELEASE 1.46.x-Greenwich-RELEASE	1.40.x-Hoxton-Higher-RELEASE 1.46.x-Hoxton-Higher-RELEASE	1.40.x-SpringCloud2020-RELEASE 1.46.x-SpringCloud2020-RELEASE	1.40.x-SpringCloud2021-RELEASE 1.46.x-SpringCloud2021-RELEASE
1.29.x	-	1.29.x-Finchley-RELEASE	1.29.x-Greenwich-RELEASE	1.29.x-Hoxton-Higher-RELEASE	-	-
1.23.x	-	1.23.x-Finchley-RELEASE	1.23.x-Greenwich-RELEASE	-	-	-

1.21.x	-	1.21.x-Finchley-RELEASE	1.21.x-Greenwich-RELEASE	-	-	-
1.18.x	-	1.18.x-Finchley-RELEASE	1.18.x-Greenwich-RELEASE	-	-	-
1.12.x	1.12.x-Edgware-RELEASE	1.12.x-Finchley-RELEASE	-	-	-	-

Demo 工程概述

最近更新时间：2024-01-19 10:39:01

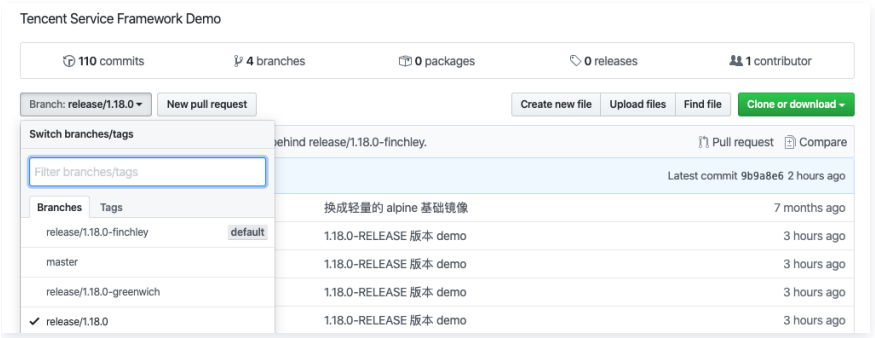
准备工作

在开始开发前，请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

下载 Demo

Demo 下载地址 。

- release/<版本号>：对应 Spring Cloud Edgware 系列的 Demo
- release/<版本号>-finchley：对应 Spring Cloud Finchley 系列的 Demo
- release/<版本号>-greenwich：对应 Spring Cloud Greenwich 系列的 Demo
- release/<版本号>-hoxton-higher：对应 Spring Cloud Hoxton Higher 系列的 Demo
- release/<版本号>-springcloud2020：对应 Spring Cloud 2020 系列的 Demo
- release/<版本号>-springcloud2021：对应 Spring Cloud 2021 系列的 Demo



Demo 目录介绍

`tsf-simple-demo` 的工程目录如下：

工程名称	工程说明
consumer-demo	TSF 微服务治理服务消费者
provider-demo	TSF 微服务治理服务提供者
msgw-demo	基于 TSF Spring Cloud MS Gateway 网关示例
opensource-zuul-demo	基于开源 Zuul 的微服务网关示例
opensource-scg-demo	基于开源 Spring Cloud Gateway 的微服务网关示例
rocketmq-producer	支持 RocketMQ 消息队列调用链的消息生产者示例
rocketmq-consumer	支持 RocketMQ 消息队列调用链的消息消费者示例
kafka-demo	支持 Kafka 调用链的示例，包含了消息消费者和生产者
mongodb-demo	支持 MongoDB 调用链的微服务示例
mysql-demo	支持 MySQL 调用链的微服务示例
redis-demo	支持 Redis 调用链的微服务示例
task-schedule-demo	TCT 分布式任务调度示例

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><!-- 调整为 SDK 长期维护（LTS）版本号 --></version>
            </parent>


            <groupId>com.tencent.tsf</groupId>
            <artifactId>tsf-demo</artifactId>
            <version><!-- 调整为 SDK 长期维护（LTS）版本号 --></version>
            <packaging>pom</packaging>



            <modules>
                <module>provider-demo</module>
                <module>consumer-demo</module>
                <module>opensource-zuul-demo</module>
                <module>rocketmq-demo</module>
                <module>mysql-demo</module>
                <module>redis-demo</module>
                <module>mongodb-demo</module>
                <module>kafka-demo</module>
                <module>msgw-demo</module>
                <module>opensource-scg-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 描述了不同微服务 demo 共同的 TSF 依赖。

```
<parent>
```

```
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-dependencies</artifactId>
        <version><!-- 调整为 SDK 长期维护（LTS）版本号 --></version>
</parent>
```

说明
SDK 长期维护（LTS）版本号请参见 Spring Cloud 概述 > SDK 版本说明 。

依赖构建及注解使用

注意
如果您使用的是 1.15.0-Edgware-RELEASE/1.15.0-Finchley-RELEASE 及之前的版本，使用方法参见 Spring Cloud SDK 历史版本使用方法 。

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-starter</artifactId>
    <version><!-- 调整为 SDK 长期维护（LTS）版本号 --></version>
</dependency>
```

`spring-cloud-tsf-starter` 中包含了服务注册发现、服务路由、服务鉴权、服务限流、服务熔断、服务容错、服务监控、分布式配置、调用链功能。

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

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

注意
TSF 各部分功能对应的注解如下，不推荐只单独用部分的注解，建议添加 @EnableTsf 注解可以开启 TSF 全部功能，

- @EnableTsfUnit // 单元化
- @EnableTsfAuth // 服务鉴权
- @EnableTsfRoute // 服务路由
- @EnableTsfRateLimit // 服务限流
- @EnableTsfSleuth // 调用链
- @EnableTsfMonitor // 服务监控
- @EnableTsfCircuitBreaker // 服务熔断
- @EnableTsfFaultTolerance // 服务容错
- @EnableTsfLane //泳道

服务注册与发现

最近更新时间：2025-07-01 14:39:41

操作场景

本文介绍在本地开发 Spring Cloud 微服务示例应用并注册到 TSF 服务注册发现中心，或者将已经接入 Eureka 服务注册与发现的应用迁移到 TSF 服务注册发现中心的操作方法。

前提条件

在开始开发前，请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

本地开发应用

创建 tsf-demo 工程，文件结构如下：

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

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

步骤1：创建服务提供者

在本地创建服务提供者应用工程，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` 中引入需要依赖的内容：

```
<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-starter</artifactId>
    </dependency>
</dependencies>
```

> **注意**
> Finchley 版本 SDK 无须添加 monitor 依赖包，具体请参见 服务监控 。

3. 开启服务注册发现

添加服务提供端的代码。

```
// 省略部分 import
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.tsf.annotation.EnableTsf;

@SpringBootApplication
@EnableFeignClients // 使用Feign微服务调用时请启用
@EnableTsf
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
```

> **注意**
> 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。

步骤2：创建服务消费者

本小节中，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. 改 pom 依赖

在 `pom.xml` 中引入需要依赖的内容：

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

```
</parent>

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

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

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` 。

```
// 省略部分 import
import org.springframework.cloud.netflix.feign.EnableFeignClients;
import org.springframework.tsf.annotation.EnableTsf;
import org.springframework.web.client.AsyncRestTemplate;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableFeignClients // 使用 Feign 微服务调用时请启用
@EnableTsf
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) throws InterruptedException {
        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
```

> **注意**
> 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。

步骤3：部署应用

参见 如何打 FatJar 包 将应用工程打包，并打包好的 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。部署相关操作可参见 容器托管应用 或 虚拟机托管应用 。

从 Eureka 迁移应用

已经接入 Eureka 服务注册与发现中心的应用，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。
2. 在单个 Spring Cloud 应用的 `pom.xml` 中，将 `spring-cloud-starter-eureka` 替换成 `spring-cloud-tsf-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 依赖的版本控制参见 Demo 或之前 pom 说明-->
```

3. 修改代码中的 Eureka 的相关注解。

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

延迟注册

> **说明：**
> 当前仅 1.46.21-Hoxton-Higher-RELEASE 及以上的 Hoxton 版本支持。

添加应用配置即可使用：

```
tsf:
  discovery:
    delay-register: true # 延迟注册开关，默认关闭
    delay-register-interval: 30000 # 延迟注册间隔，默认30000，单位：毫秒
```

配置管理

最近更新时间：2025-03-10 18:08:32

操作场景

该任务指导您通过轻量级服务注册中心或者腾讯微服务平台控制台下发动态配置。

前提条件

开始实践分布式配置功能前，请确保已参见 下载 Maven 下载安装了 Java 和 Maven。

添加依赖

向工程中添加 `spring-cloud-tsf-starter` 依赖并开启 `@EnableTsf` 注解，详情请参见 Demo工程概述 文档。

> **注意：**
> 如果您使用的是 1.15.0-Edgware-RELEASE/1.15.0-Finchley-RELEASE 及之前的版本，使用方法参见 Spring Cloud SDK 历史版本使用方法 。

修改配置

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
- `@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` 。使用 `@RefreshScope注解` 开启 refresh 机制。

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

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

    @Value("${test.demo.prop:default}")
    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 `RedisConfig` 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);
```

```
    }
}
```

轻量级服务注册中心下发动态配置

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：

前提条件

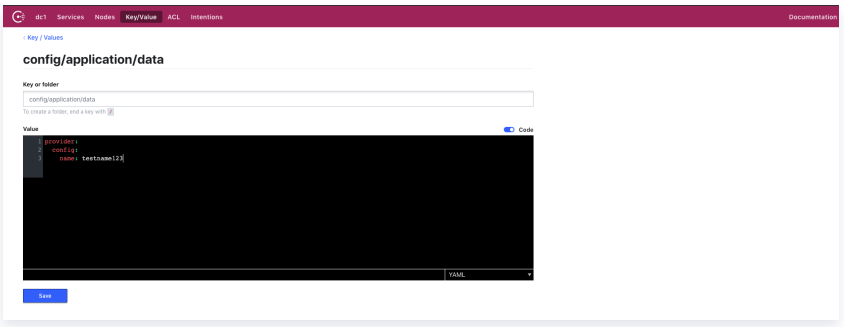
1. 已完成轻量级服务注册中心的安装和启动（详情请参见 轻量级服务注册中心 ）。
2. 服务已连接到轻量级服务注册中心。

操作步骤

1. 浏览器访问 **consul 控制台**（http://127.0.0.1:8500 ）。

> **说明：**
> 只有成功启动了 consul 后，才能访问 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` （适用于原生 Spring Cloud ）
   - 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。

腾讯微服务平台控制台下发动态配置

用户可以通过腾讯微服务平台控制台来下发动态配置。

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。参见 日志配置项 。

操作步骤

关于如何通过控制台创建及下发更新的配置，请参见 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
```

调用链
调用链快速入门

最近更新时间：2025-03-14 14:53:32

功能说明

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，无需其他配置。

微服务对下游组件访问的链路支持

微服务对消息队列组件（Kafka、CMQ、RocketMQ）、网关组件（微服务网关、API 网关）、数据库（Redis、MySQL、PostgreSQL、MongoDB）和 httpclient (RestTemplate、Feign)的访问操作会产生跟踪日志， TSF 会对该日志进行采集、分析、统计，这些组件的调用会展现在 TSF 平台的链路追踪中。具体使用说明如下：

- Redis 链路使用说明
- MySQL 链路使用说明
- MongoDB 链路使用说明
- 消息队列 CMQ 链路使用说明
- Kafka 链路追踪使用说明
- RocketMQ 链路追踪使用说明

> **说明：**
> 非微服务组件的链路支持功能依赖于1.14.0版本及以上版本的 SDK，详情请参见 SDK 更新日志 。

前提条件

在开始开发前，请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

依赖构建及注解使用

1. 向工程中添加 `spring-cloud-tsf-starter` 依赖，在 pom.xml 文件中添加以下代码：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-starter</artifactId>
    <version><!-- 调整为 SDK 长期维护（LTS）版本号 --></version>
</dependency>
```

`spring-cloud-tsf-starter` 中包含了服务注册发现、服务路由、服务鉴权、服务限流、服务熔断、服务容错、服务监控、分布式配置、调用链功能。

2. 向 Application 类中添加注解 `@EnableTsf`：

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

> **注意：**
> 如果您使用的是 1.15.0-Edgware-RELEASE/1.15.0-Finchley-RELEASE 及之前的版本，使用方法参见 Spring Cloud SDK 历史版本使用方法 。

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)
}
```

注入和获取 trace 信息

TSF SDK 1.23 及之后版本，支持注入和获取 traceId、spanId、tag 信息。

> **说明**
> - traceId、spanId 格式不建议自定义，可能有未知错误。
> - TSF 不保证注入后调用链的正确性。

Edgware 版本 SDK 使用方式

```
private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
@Autowired
private Tracer tracer;

public void addSpan() throw Exception{
    //获取 traceId、spanId、tag
        Span oldSpan = tracer.getCurrentSpan();
        String traceId = oldSpan.getTraceId();
        String spanId = oldSpan.getSpanId();
        Map<String, String> tags = oldSpan.tags();
        LOG.info("traceId: {}, spanId: {}", traceId, spanId);
        for (String key : tags.keySet()) {
            LOG.info("key: {}, value: {}", key, tags.get(key));
        }

        //设置 tag
        tracer.addTag("http.method","get");
        Map<String, String> tags1 = tracer.getCurrentSpan().tags();
```

```
        for (String key : tags1.keySet()) {
            LOG.info("key: {}, value: {}", key, tags1.get(key));
        }

        //设置 traceId、spanId（必须先新建 span）
        Span.SpanBuilder spanBuilder = Span.builder();
        Span span =
spanBuilder.name("mockName").traceId(UUID.randomUUID().toString()).spanId(UUID.randomUUID().toStrin
g()).tag("key", "value").build();

        tracer.continueSpan(span);

        Span newSpan = tracer.getCurrentSpan();
        String newTraceId = newSpan.getTraceId();
        String newSpanId = newSpan.getSpanId();

        LOG.info("traceId: {}, spanId: {}", newTraceId, newSpanId);
 }
```

Finchley、Greenwich 版本 SDK 使用方式

```
    private static final Logger LOG =
LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
    @Autowired
    private Tracing tracing;

   public void addSpan() throw Exception{
      TraceContext traceContext = tracing.tracer().currentSpan().context();

        TraceContext context =
traceContext.toBuilder().traceId(2035873338086630653L).spanId(-2035873338086630653L).build();

        CurrentTraceContext currentTraceContext = tracing.currentTraceContext();
        currentTraceContext.newScope(context);

        Span spanCur = tracing.tracer().currentSpan();


        long traceId1 = spanCur.context().traceId();
        long spanId1 = spanCur.context().spanId();
        LOG.info("traceId1:{},spanId1:{}",traceId1, spanId1);


        //添加tag
        spanCur.tag("qx-test1","value");
        spanCur.tag("qx-test2","value");
        spanCur.tag("qx-test1","value");

        //获取tag
        Field state1 = spanCur.getClass().getDeclaredField("state");
        state1.setAccessible(true);
        MutableSpan mutableSpan1 = (MutableSpan)state1.get(spanCur);
        Field tags1 = mutableSpan1.getClass().getDeclaredField("tags");
        tags1.setAccessible(true);
        ArrayList<String> list1 = (ArrayList<String>) tags1.get(mutableSpan1);
        String key1 =null, value1 =null;
```

```
        for (int i = 0, length = list1.size(); i < length; i += 2) {
            key1 =  list1.get(i);
            value1 = list1.get(i + 1);

            LOG.info("tags key1: {}, value1: {}", key1, value1);
        }

        //获取 traceid、spanid
        TraceContext traceContext1 = spanCur.context();
        long traceId2 = traceContext1.traceId();
        long spanId2 = traceContext1.spanId();
        LOG.info("traceId1:{},spanId1:{}",traceId2, spanId2);
    }
```

异步线程注入调用链信息（以 Finchley、Greenwich 版 SDK 为例）

```
private static final Logger LOG = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());

@Autowired
private Tracing tracing;

public void setTraceContext() {
    LOG.info("setTraceContext start");

    // 获取调用链上下文信息
    Span spanCur = tracing.tracer().currentSpan();
    TraceContext traceContextCur = spanCur.context();
    long traceIdCur = traceContextCur.traceId();                    // traceId 是 64 位二进制的 long
类型的非零随机数
    long spanIdCur = traceContextCur.spanId();
    String traceIdStringCur = traceContextCur.traceIdString();      // traceIdString 是 16 位十六进制
的字符串，用于向下游传递
    String spanIdStringCur = HexCodec.toLowerHex(spanIdCur);
    LOG.info("traceId: {}, spanId: {}, traceIdString: {}, spanIdString: {}", traceIdCur, spanIdCur,
traceIdStringCur, spanIdStringCur);

    ExecutorService executorService = Executors.newCachedThreadPool();
    executorService.execute(() -> {
        // 异步线程，调用链上下文信息为空
        LOG.info("new thread, trace context is empty");

        // 注入调用链上下文信息
        TraceContext traceContext =
TraceContext.newBuilder().traceId(traceIdCur).spanId(spanIdCur).sampled(true).build();
        CurrentTraceContext currentTraceContext = tracing.currentTraceContext();
        currentTraceContext.newScope(traceContext);

        // 注入后获取调用链上下文信息
        Span spanThread = tracing.tracer().currentSpan();
        TraceContext traceContextThread = spanThread.context();
        long traceIdThread = traceContextThread.traceId();
        long spanIdThread = traceContextThread.spanId();
        String traceIdStringThread = traceContextThread.traceIdString();
        String spanIdStringThread = HexCodec.toLowerHex(spanIdThread);
        LOG.info("in thread, traceId: {}, spanId: {}, traceIdString: {}, spanIdString: {}",
traceIdThread, spanIdThread, traceIdStringThread, spanIdStringThread);
```

```
    });

    executorService.shutdown();
}
```

调整采样率

在 application.yml 或者 bootstrap.yml 添加如下配置：

```
tsf:
  sleuth:
    samplerRate: 0.1
```

说明：
samplerRate 默认为1，采样率为100%；samplerRate 设置为0.1时，采样率为10%，以此类推。

Redis 链路追踪

最近更新时间：2022-06-09 16:38:10

添加依赖

考虑到 Redis 库的多样性，以及 spring-data-redis 库的易用性，目前只对 `spring-boot-starter-data-redis` 进行支持，在引用 spring-boot-starter-data-redis 时不要指定版本，只需要整个工程依赖 parent pom 即可：

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-dependencies</artifactId>
    <version>tsf 的版本号（1.14以后开始支持 Redis）</version>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
     </dependency>
</parent>
```

`spring-boot-starter-data-redis` 的版本为 parent pom 文件管理的 Redis Starter 的版本。在代码中具体使用时，引入 RedisTemplate，然后使用其方法即可。不建议直接引用 Jedis 和 Lettuce 相关的依赖，spring-boot-starter-data-redis 会自动引用相关的依赖，并做适配。
如果通过其他方式引入 Redis 客户端（例如直接 new Jedis），则将无法在 TSF 的链路中查看到相应的信息。

16.1-Finchley-RELEASE 版本 Redis 调用链使用方式

lettuce 方式

使用 lettuce 方式，pom 依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

不需要额外依赖，自动依赖了 lettuce 方面的 jar 包，可以直接使用。通过该配置，也具有链路追踪的能力。

jedis 方式

使用 jedis 方式，pom 依赖如下：

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
</dependency>
<!-- redis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

通过以上配置，也能使用 SDK 的链路追踪能力。

MySQL 链路追踪

最近更新时间：2022-06-09 16:38:49

TSF 支持实现 JDBC 规范的 MySQL 驱动器和各类连接池（如 Tomcat-JDBC、DBCP、Hikari、Druid），在使用时需引入 SpringBoot 相关依赖和 MySQL 驱动依赖：

```
<!-- spring-boot-starter-jdbc -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <version>RELEASE</version>
</dependency>
<!-- mysql -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>RELEASE</version>
</dependency>
```

引入依赖后，根据需要添加相关数据库连接池依赖或直接使用 SpringBoot 默认选项。

MongoDB 链路追踪

最近更新时间：2023-09-28 15:36:52

考虑到 spring-data-mongodb 库的易用性，TSF 目前只对 `spring-boot-starter-data-mongodb` 进行支持，在引用 spring-boot-starter-data-mongodb 时不要指定版本，只需要整个工程依赖 parent pom 即可，示例如下：

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-dependencies</artifactId>
    <version>tsf的版本号（1.14以后开始支持 MongoDB）</version>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
     </dependency>
</parent>
```

`spring-boot-starter-data-mongodb` 的版本即是 parent pom 文件管理的 mongodb starter 的版本。在代码中具体使用时，引入 MongoTemplate，然后使用其方法即可。

- 如果需要制定 MongoDB 连接的 URI，需要满足以下格式：

  ```
  mongodb://host[:port1][/[database][?options]] #暂时只支持单节点 MongonDB
  ```

  也可以不用填写，即默认 host 是本机，默认端口27017。
- 如果通过其他方式引入 MongoDB 客户端，例如直接 `new MongoClient(host,port)`，则在 TSF 的链路中将无法查看到相应的信息。

Kafka 链路追踪

最近更新时间：2022-06-21 14:50:45

Kafka 的链路追踪能力通过 Spring Boot 的自动配置方式实现。

1.16.0-Edgware 版本说明

实现方式是向 bean 容器中各加入了一个 producerFactory 和 consumerFactory 类的 bean。他们各自分别重写了 createKafkaProducer 方法和 createKafkaConsumer 方法。使用该 producerFactory/consumerFactory 便具有了链路追踪的能力。建议使用 Spring 默认的 KafkaTemplate，无须自定义。
1.16.0-Edgware 版使用的 spring-kafka 版本必须在**1.3.0**以上，因为从这个版本开始 kafka-client 中增加了请求头，这才能方便的实现链路追踪。

```
<!--支持 kafka 使用调用链-->
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
            <version>1.3.0.RELEASE</version>
        </dependency>
```

参见配置

```
#============== kafka ===================
# 指定 kafka 代理地址，可以多个
spring.kafka.bootstrap-servers=

#=============== provider  =======================
spring.kafka.producer.retries=0
# 每次批量发送消息的数量
spring.kafka.producer.batch-size=16384
spring.kafka.producer.buffer-memory=33554432

# 指定消息 key 和消息体的编解码方式
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer

#=============== consumer  =======================
# 指定默认消费者 Group ID
spring.kafka.consumer.group-id=test-consumer-group

spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.enable-auto-commit=true
spring.kafka.consumer.auto-commit-interval=100

# 指定消息 key 和消息体的编解码方式
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

其他更详细配置、使用方法详见 Demo 示例 。

1.16.0-Finchley 版本说明

实现方式是向 Spring 容器中各增加了一个对切面，分别作用于 ProducerFactory.createProducer 方法和 ConsumerFactory.createConsumer 方法的，使得通过他们返回的 KafkaProducer 和 KafkaConsumer 具有链路追踪的能力。

使用 1.16.0-Finchley 版本无须考虑 spring-kafka 版本问题，无须指定它的版本，因为底层依赖的 Finchley 版本的 spring cloud 会自动为它配置高于 1.3.0.RELEASE 版本的 spring-kafka。

```
        <!--支持kafka使用调用链-->
        <dependency>
            <groupId>org.springframework.kafka</groupId>
            <artifactId>spring-kafka</artifactId>
        </dependency>
```

参见配置

```
  server:
    servlet:
      context-path: /
    port: xxxxx
  spring:
    application:
      name: xxxxxx
    kafka:
      bootstrap-servers: xxxxxx
      #生产者的配置，大部分我们可以使用默认的，这里列出几个比较重要的属性
      producer:
        #每批次发送消息的数量
        batch-size: 16
        #设置大于0的值将使客户端重新发送任何数据，一旦这些数据发送失败。注意，这些重试与客户端接收到发送错误时的重试没有什么不同。允许重试将潜在的改变数据的顺序，如果这两个消息记录都是发送到同一个partition，则第一个消息失败第二个发送成功，则第二条消息会比第一条消息出现要早。
        retries: 0
        #producer可以用来缓存数据的内存大小。如果数据产生速度大于向broker发送的速度，producer会阻塞或者抛出异常，以“block.on.buffer.full”来表明。这项设置将和producer能够使用的总内存相关，但并不是一个硬性的限制，因为不是producer使用的所有内存都是用于缓存。一些额外的内存会用于压缩（如果引入压缩机制），同样还有一些用于维护请求。
        buffer-memory: 33554432
        #key序列化方式
        key-serializer: org.apache.kafka.common.serialization.StringSerializer
        value-serializer: org.apache.kafka.common.serialization.StringSerializer
      #消费者的配置
      consumer:
        #Kafka中没有初始偏移或如果当前偏移在服务器上不再存在时,默认区最新 ，有三个选项 **latest, earliest, none**
        auto-offset-reset: latest
        #是否开启自动提交
        enable-auto-commit: true
        #自动提交的时间间隔
        auto-commit-interval: 100
        #key的解码方式
        key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        #value的解码方式
        value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        #在/usr/local/etc/kafka/consumer.properties中有配置
        group-id: test-consumer-group
  tsf:
    swagger:
      enabled: false
  logging:
    file: /tsf-demo-logs/${spring.application.name}/root.log
```

其他更详细配置、使用方法详见 Demo 示例 。

Kafka 批量消费消息场景

- Kafka 消费单条消息时，接收的调用链信息在同个线程中自动传递。
- Kafka 批量消费消息时，需要手动将调用链信息注入，从而继续传递调用链信息，方法如下：

```
@Autowired
private KafkaTracing kafkaTracing;

kafkaTracing.joinBatchReceiveSpan(consumerRecord);
```

> **说明**
> 当前仅以下 SDK 版本及其之后的 SDK 版本支持：1.29.4-H、1.32.0-G、1.29.1-G、1.32.1-F、1.29.7-F。

RocketMQ 链路追踪

最近更新时间：2023-05-23 16:11:35

TSF 从1.23.0版本开始支持 RocketMQ 使用联路追踪能力。

链路追踪原理

利用 springboot 提供自动配置原理，加入一个能插手 bean DefaultMQProducer、和 bean DefaultMQPushConsumer 创建过程的自动配置类。使用代理来增强 DefaultMQProducer/DefaultMQPushConsumer，在调用相应方法时加入 tracing 的逻辑，在方法结束时将 tracing 数据搜集起来，统一存储。最后分析展示给前端页面。

引入依赖及配置

```
    <!-- TSF 启动器 -->
    <dependency>
        <groupId>com.tencent.tsf</groupId>
        <artifactId>spring-cloud-tsf-starter</artifactId>
    </dependency>
    <!--RocketMQ-->
    <dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-client</artifactId>
        <version>4.4.0</version>
    </dependency>
apache:
  rocketmq:
    consumer:
      pushConsumer: myConsumer
    producer:
      producerGroup: myProducer
    name-server: xxx.xxx.xx.xx #rocketMq ip
    topic: long-topic
```

生产者示例

向 Spring 容器中加入 bean DefaultMQProducer：

```
@Configuration
public class ProducerConfiguration {
    private static final Logger logger = LoggerFactory.getLogger(ProducerConfiguration.class);
    @Value("${apache.rocketmq.name-server}")
    private String namesrvAddr;
    @Value("${apache.rocketmq.producer.producerGroup}")
    private String producerGroup;
    @Bean
    public DefaultMQProducer defaultMQProducer() throws MQClientException {
        DefaultMQProducer producer = new DefaultMQProducer(producerGroup);
        producer.setNamesrvAddr(namesrvAddr);
        producer.setVipChannelEnabled(false);
        producer.setRetryTimesWhenSendAsyncFailed(0);
        producer.start();
        logger.info("rocketmq producer is starting");
        return producer;
    }
```

```
}
```

引用 DefaultMQProducer，发送消息：

```
@Service
public class RocketmqService {
    private static  final Logger logger = LoggerFactory.getLogger(RocketmqService.class);
    @Autowired
    private DefaultMQProducer defaultMQProducer;
    // 使用 application.properties 里定义的 topic 属性
    @Value("${apache.rocketmq.topic}")
    private String springTopic;
    private AtomicLong id =new AtomicLong(0);
    @Scheduled(fixedDelayString = "${consumer.auto.test.interval:5000}")
    public String prepareSend() throws InterruptedException, RemotingException, MQClientException,
MQBrokerException {
       return send();
    }
    public String send() throws InterruptedException, RemotingException, MQClientException,
MQBrokerException {
        String sentText = "rocketmq message: msg id="+id.addAndGet(1);
        Message message = new Message(springTopic,"push", sentText.getBytes());
        message.putUserProperty("traceID","1234567");
        SendResult sendResult = defaultMQProducer.send(message);
        logger.info("消息id:"+id.get()+","+"发送状态:"+sendResult.getSendStatus());
        return sendResult.getMsgId();
    }
}
```

消费者示例

```
@Configuration
public class ConsumerConfiguration {
    private static final Logger logger = LoggerFactory.getLogger(ConsumerConfiguration.class);
    @Value("${apache.rocketmq.name-server}")
    private String namesrvAddr;
    @Value("${apache.rocketmq.topic}")
    private String springTopic;
    @Value("${apache.rocketmq.consumer.pushConsumer}")
    private String consumerGroup;
    @Bean
    public DefaultMQPushConsumer pushConsumer() {
        DefaultMQPushConsumer consumer = new DefaultMQPushConsumer(consumerGroup);
        consumer.setNamesrvAddr(namesrvAddr);
        try {
            // 订阅PushTopic下Tag为push的消息，都订阅消息
            consumer.subscribe(springTopic, "push");
            // 程序第一次启动从消息队列头获取数据
            consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_FIRST_OFFSET);
            //可以修改每次消费消息的数量，默认设置是每次消费一条
            consumer.setConsumeMessageBatchMaxSize(1);
            //在此监听中消费信息，并返回消费的状态信息
            consumer.registerMessageListener((MessageListenerConcurrently) (msgs, context) -> {
                // 会把不同的消息分别放置到不同的队列中
                for (Message msg : msgs) {
                    System.out.println("接收到了消息: " + new String(msg.getBody()));
```

```
                    logger.info("接收到了消息: " + new String(msg.getBody()));
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            });
            consumer.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return consumer;
    }
}
```

外部组件接入 TSF 调用链

最近更新时间：2024-01-19 10:39:01

操作场景

TSF SDK 对一些常用的消息队列组件、数据库组件等有埋点操作以支持全链路跟踪，但在客户实际场景下可能存在 TSF SDK 无法埋点的场景，例如流量经过 API 网关或客户未接入 TSF 组件时无法持续跟踪请求。
本文将介绍如何通过传递 HTTP header 使外部组件可在 TSF 依赖拓扑图和调用链查询中显示。

- 当前仅以下 SDK 版本及其之后的 SDK 版本支持：1.29.5-Hoxton、1.29.12-Finchley、1.29.2-Greenwich。
- 当前支持在依赖拓扑图中展示特定图标的外部组件包括：API 网关。

前提条件

在开始本文的实践前，您需要先了解 TSF 的以下功能。

- 调用链快速入门
- 服务依赖拓扑
- 调用链查询

操作步骤

API 网关（apigw）接入

调用链路为如下几种：

- consumer > apigw > provider，其中 consumer 与 provider 均部署在 TSF 上。TSF 依赖拓扑图将显示三个节点串联的链路，以及展示完整的两跳请求的调用详情。
- 外部请求 > apigw > provider，其中 provider 部署在 TSF 上。TSF 依赖拓扑将显示 apigw 和 provider 两个节点的链路，以及 apigw > provider 一跳请求的调用详情。
- consumer > apigw > msgw，其中 consumer 与 msgw 均部署在 TSF 上，msgw 为微服务网关。TSF 依赖拓扑图将显示三个节点串联的链路，以及展示完整的两跳请求的调用详情。
- 外部请求 > apigw > msgw，其中 msgw 部署在 TSF 上。TSF 依赖拓扑将显示 apigw 和 msgw 两个节点的链路，以及 apigw > msgw 一跳请求的调用详情。

consumer 侧注入
consumer 侧需要注入 apigw 的组件类型。因为通过 RestTemplate 或 Feign 调用 apigw，TSF sdk 无法得知下游服务的组件类型。若不注入，在 TSF 依赖拓扑图中，链路将会在 apigw 处中断。
consumer 侧还需要注入 apigw 的服务 ID 作为唯一标志。

```
TsfTracingContext tsfTracingContext = TsfTracingContextHolder.get();
tsfTracingContext.setClientTag("remoteComponent", "api");
tsfTracingContext.setServerServiceName("service-mygpdlyb");  // 设置为 apigw 的 service id
```

apigw 注入（注：该步骤由 apigw 实现）
apigw 需要透传 consumer 传递来的 trace 上下文信息。如果为第一跳可忽略。TSF 调用链功能使用基于 zipkin，因此需要透传如下信息。

HTTP Header	必选	含义	示例	说明
X-B3-TraceId	是	调用链 trace id。64 bit 长度。	471a1b50aa4e9852	透传 consumer 的值，请勿修改。若为第一跳，可为空。
X-B3-ParentSpanId	是	调用链 parent span id。64 bit 长度。	471a1b50aa4e9852	64 bit 长度。透传 consumer 的值，请勿修改。若为第一跳，可为空。

X-B3-SpanId	是	调用链 span id。64 bit 长度。	d3cc868d837f1985	64 bit 长度。透传 consumer 的值，请勿修改。若为第一跳，可为空。
X-B3-Sampled	是	调用链是否被采样。设置为 1，表示被采样。	1	透传 consumer 的值，请勿修改。若为第一跳，可为空。
b3	否	调用链单个 header。格式为b3={TraceId}-{SpanId}-{SamplingState}-{ParentSpanId}。id 均为 64 bit 长度	471a1b50aa4e9852-d3cc868d837f1985-1-471a1b50aa4e9852	brave 5.3+ 版本支持单个 b3 header。若有该 header，需要透传

apigw 接受 consumer 的请求，这一跳的 span 数据需要如下信息。如果为第一跳可忽略。

HTTP Header	必选	含义	示例	说明
X-Tsf-Server-Timestamp	是	apigw 接受请求以及返回响应的时间戳。格式为 sr-ss，时间精度均为微妙	1652358271377000-1652358281000000	注意精度为微妙，用单横线连接 server receive 和 server send的时间戳
X-Tsf-Server-Service-Name	是	apigw 的服务名。将显示在 TSF 依赖拓扑图和调用链详情中	service-mygpdlyb	apigw 服务 ID
X-Tsf-Server-Service-Interface	否	apigw 的接口名，将显示在 TSF 调用链详情中	/apigw-echo-provider/{param}	默认值为 external
X-Tsf-Server-Ip-Port	否	apigw 接受请求机器的 ip 和 port	127.0.0.1:8009	默认值为部署在 TSF 上的下游服务的 ip 和 port
X-Tsf-Server-Remote-Ip-Port	否	apigw 上游节点 consumer 的 ip 和 port	172.16.0.9:18083	默认值为部署在 TSF 上的下游服务的 HTTP 请求的 remoteAddress
X-Tsf-Server-Local-Component	是	apigw 的节点类型	apigw	必填且唯一取值

apigw 向 provider msgw 发送请求，这一跳的 span 数据需要如下信息。

HTTP Header	必选	含义	示例	说明
X-Tsf-Client-Timestamp	是	apigw 发送请求时间戳。格式为 cs，时间精度为微妙	1652358271377000	注意精度为微妙，cs 表示 client send 的时间戳
X-Tsf-Client-Service-Name	是	apigw 的服务名。将显示在 TSF 依赖拓扑图和调用链详情中	service-mygpdlyb	apigw 服务 ID
X-Tsf-Client-Service-Interface	否	apigw 的接口名，将显示在 TSF 调用链详情中	/apigw-echo-provider/{param}	默认值为 external
X-Tsf-Client-Ip-Port	否	apigw 发出请求机器的 ip 和 port	127.0.0.1:8009	默认值为空
X-Tsf-Client-Local-Component	是	apigw 的节点类型	apigw	必填且唯一取值

外部组件接入

调用链路为如下几种：

- consumer > 任意多跳的外部组件 > provider，其中 consumer 与 provider 均部署在 TSF 上。TSF 依赖拓扑图将显示三个节点串联的链路，以及展示完整的两跳请求的调用详情。
- 外部请求 >任意多跳的外部组件 > provider，其中 provider 部署在 TSF 上。TSF 依赖拓扑将显示外部组件和 provider 两个节点的链路，以及外部组件 > provider 一跳请求的调用详情。
- consumer > 任意多跳的外部组件 > msgw，其中 consumer 与 msgw 均部署在 TSF 上，msgw 为微服务网关。TSF 依赖拓扑图将显示三个节点串联的链路，以及展示完整的两跳请求的调用详情。
- 外部请求 > 任意多跳的外部组件 -> msgw，其中 msgw 部署在 TSF 上。TSF 依赖拓扑将显示外部组件和 msgw 两个节点的链路，以及外部组件 > msgw 一跳请求的调用详情。

consumer 侧注入

consumer 侧需要注入外部组件的组件类型，默认为 ms，表示微服务组件，无须注入。

```
TsfTracingContext tsfTracingContext = TsfTracingContextHolder.get();
tsfTracingContext.setClientTag("remoteComponent", "ms");
```

consumer 侧需要注入外部组件的微服务名。注入的微服务名需要与后续传递的 HTTP header 中微服务名保持一致。

```
tsfTracingContext.setServerServiceName("my-service");
```

外部组件注入

apigw 需要透传 consumer 传递来的 trace 上下文信息。如果为第一跳，可忽略。TSF 调用链功能使用基于 zipkin，因此需要透传如下信息。

HTTP Header	必选	含义	示例	说明
X-B3-TraceId	是	调用链 trace id。64 bit 长度。	471a1b50aa4e9852	透传 consumer 的值，请勿修改。若为第一跳，可为空。
X-B3-ParentSpanId	是	调用链 parent span id。64 bit 长度。	471a1b50aa4e9852	64 bit 长度。透传 consumer 的值，请勿修改。若为第一跳，可为空。
X-B3-SpanId	是	调用链 span id。64 bit 长度。	d3cc868d837f1985	64 bit 长度。透传 consumer 的值，请勿修改。若为第一跳，可为空。
X-B3-Sampled	是	调用链是否被采样。设置为 1，表示被采样。	1	透传 consumer 的值，请勿修改。若为第一跳，可为空。
b3	否	调用链单个 header。格式为b3={TraceId}-{SpanId}-{SamplingState}-{ParentSpanId}。id 均为 64 bit 长度	471a1b50aa4e9852-d3cc868d837f1985-1-471a1b50aa4e9852	brave 5.3+ 版本支持单个 b3 header。若有该 header，需要透传

外部组件接受 consumer 的请求，这一跳的 span 数据需要如下信息。如果为第一跳可忽略。

HTTP Header	必选	含义	示例	说明
X-Tsf-Server-Timestamp	是	外部组件接受请求以及返回响应的时间戳。格式为 sr-ss，时间精度均为微妙	1652358271377000-1652358281000000	注意精度为微妙，用单横线连接 server receive 和 server send的时间戳
X-Tsf-Server-Service-Name	否	外部组件的服务名。将显示在 TSF 依赖拓扑图和调用链详情中	external	需要与 X-Tsf-Client-Service-Name 保持一致，否则无法在依赖拓扑图中串联。默认值为 external

X-Tsf-Server-Service-Interface	否	外部组件的接口名	external	默认值为 external
X-Tsf-Server-Ip-Port	否	外部组件的 ip 和 port	127.0.0.1:8009	默认值为部署在 TSF 上的下游服务的 ip 和 port
X-Tsf-Server-Remote-Ip-Port	否	外部组件上游节点 consumer 的 ip 和 port	172.16.0.9:18083	默认值为部署在 TSF 上的下游服务的 HTTP 请求的 remoteAddress
X-Tsf-Server-Local-Component	否	外部组件的节点类型	ms	需要与在 consumer 侧注入的节点类型、X-Tsf-Client-Local-Component 保持一致。默认值为 ms

外部组件向 provider 或 msgw 发送请求，这一跳的 span 数据需要如下信息。

HTTP Header	必选	含义	示例	说明
X-Tsf-Client-Timestamp	是	外部组件发送请求时间戳。格式为 cs，时间精度为微妙	1652358271377000	注意精度为微妙，cs 表示 client send 的时间戳
X-Tsf-Client-Service-Name	否	外部组件的服务名。将显示在 TSF 依赖拓扑图和调用链详情中	external	默认值为 external
X-Tsf-Client-Service-Interface	否	外部组件的接口名，将显示在 TSF 调用链详情中	external	默认值为 external
X-Tsf-Client-Ip-Port	否	外部组件发出请求机器的 ip 和 port	127.0.0.1:8009	默认值为空
X-Tsf-Client-Local-Component	否	外部组件的组件类型	ms	默认值为 ms

说明与注意

注入失败不会影响业务的正常运行

ApacheHttpClient 链路追踪

最近更新时间：2022-07-27 08:51:45

操作场景

TSF SDK 对一些常用的消息队列组件、数据库组件等有埋点操作以支持全链路跟踪，但在客户实际场景下可能存在 直接使用 ApacheHttpClient 的场景。
本文将介绍如何使用 ApacheHttpClient 注入调用链。

> **说明**
> 当前仅以下 SDK 版本及其之后的 SDK 版本支持：1.29.5-Hoxton、1.29.12-Finchley、1.29.2-Greenwich。

前提条件

在开始本文的实践前，您需要先了解 TSF 的以下功能。

- 调用链快速入门
- 服务依赖拓扑
- 调用链查询

操作步骤

有两种使用方式

直接使用 @Autowired 注入的 HttpClientBuilder 创建 client。

```
@Autowired
private HttpClientBuilder httpClientBuilder;

public void call(String str) {

    CloseableHttpClient client = httpClientBuilder.build();
    HttpGet get = new HttpGet("http://localhost:18081/echo/" + str);
    HttpResponse response = client.execute(get);
}
```

创建 HttpClientBuilder 的时候传入 httpTracing。

```
@Autowired
private HttpTracing httpTracing;

public void call(String str) {

    HttpClientBuilder httpClientBuilder = TracingHttpClientBuilder.create(httpTracing);

    CloseableHttpClient client = httpClientBuilder.build();
    HttpGet get = new HttpGet("http://localhost:18081/echo/" + str);
    HttpResponse response = client.execute(get);
}
```

说明与注意

注入失败不会影响业务的正常运行

服务治理

最近更新时间：2024-11-05 16:13:52

操作场景

服务治理功能包含服务鉴权、服务限流、服务路由和服务熔断，该任务指导您在应用开发过程中配置服务治理功能。

前提条件

- 在开始开发前，请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

> **说明：**
> - 若要使用服务熔断功能，请确保 SDK 版本高于1.19。
> - 不支持创建子线程进行调用后的服务治理。

- 已参见 服务治理 文档熟悉了服务治理功能。

添加依赖

向工程中添加 `spring-cloud-tsf-starter` 依赖并开启 `@EnableTsf` 注解。

> **注意：**
> 如果您使用的是 1.15.0-Edgware-RELEASE/1.15.0-Finchley-RELEASE 及之前的版本，使用方法参见 Spring Cloud SDK 历史版本使用方法 。

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-starter</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

`spring-cloud-tsf-starter` 中包含了服务注册发现、服务路由、服务鉴权、服务限流、服务熔断、服务容错、服务监控、分布式配置、调用链功能。

2. 向 Application 类中添加注解 `@EnableTsf`：

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

服务鉴权

您需要在服务主调方和被调方都添加依赖项和注解的开启。此时您已经对服务开启了鉴权功能，任何到达服务的请求都会被鉴权，鉴权不通过时会返回 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。

1. 控制台上配置鉴权规则，参见 服务鉴权基本操作 。
2. 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);
    }
```

服务限流

在应用工程中添加依赖和注解后，您可以直接在控制台上配置限流规则，参见 服务限流原理及使用 。
当您开启服务限流功能后，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。

服务路由

在应用工程中添加依赖和注解后，您可以直接在控制台上配置路由规则，参见 服务路由使用方法 。

服务熔断

TSF 摒弃了已经不再继续维护的 Hystrix 断路器，采用官方推荐的 Resilience4J 作为底层实现。相比较原有单一的接口级别熔断，我们在此之上扩展成为：实例、API、服务级别熔断。同时，除了错误比率，TSF 也支持超时比率熔断，允许用户根据业务自行选择熔断配置。请配合 TSF 其他功能一起使用。

1. 关闭 Hystrix。
   使用 TSF 熔断功能需要将 Hystrix 关闭（默认是关闭的，如果之前有打开还请关闭）。相关的容错功能请参见 服务容错 。

> **注意**
> 如果您已经使用了 feign 的 fallback 或者 fallbackFactory 功能，此处将会不再生效。如果您需要继续使用该功能或需要使用更丰富的容错功能，请参见 服务容错 步骤3进行配置。

```
feign:
```

```
hystrix:
  enabled: false
```

2. 进行熔断配置。

   目前支持两种方式的熔断配置：

   - 方式一：在线动态配置下发。具体配置方法请参见 服务熔断使用方法 。
   - 方式二：本地静态配置（适合本地联调使用，如果线上使用会被在线配置覆盖，请谨慎使用），配置在 yaml 配置文件中。

```
tsf_namespace_id: default_namespace

tsf:
 circuit-breaker:
     # 可以配置多条规则
     rules:
         # 需要熔断的目标微服务名
         - targetServiceName: provider-demo
             # 熔断级别 API/SERVICE/INSTANCE
         - isolationLevel: API
             # 目标熔断服务的namespaceId，如果本地联调，需与tsf_namespace_id保持一致
         - targetNamespaceId: "default_namespace"
             # SERVICE和INSTANCE级别，只允许配置一个策略
             # API级别可以针对不同的API配置多个策略，也可以多个API配置一个策略
         - strategyList:
                     # 滑动窗口大小
                     - slidingWindowSize: 10
                     # 最小熔断请求数
                     - minimumNumberOfCalls: 10
                     # 熔断错误比例
                     - failureRateThreshold: 60
                     # 打开到半开状态的时间
                     - waitDurationInOpenState: 5
                     # 最大熔断实例个数百分比
                     # 只在INSTANCE级别生效
                     - maxEjectionPercent: 51
                     # 慢请求阈值，单位为ms
                     - slowCallDurationThreshold: 60000
                     # 慢请求熔断比例
                     - slowCallRateThreshold: 50
                     # 该策略作用的API，可以同时作用于多个API
                     - apiList:
                         - method: GET
                           path: "/echo/{param}"
                         - method: GET
                           path: "/echo2/{str}"
```

参数传递

最近更新时间：2023-11-29 15:33:21

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，例如一些业务场景：

- `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，例如 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><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

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见 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 custom.user=12345678 来检索调用链。注意需要添加“custom.”前缀查询。调用链标签的具体使用方法参见 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。

API 注册

最近更新时间：2024-09-04 14:29:31

操作场景

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。

前提条件

请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

添加依赖

向工程中添加 `spring-cloud-tsf-starter` 依赖并开启 `@EnableTsf` 注解。

> ⚠ **注意**
> 如果您使用的是 1.15.0-Edgware-RELEASE/1.15.0-Finchley-RELEASE 及之前的版本，使用方法参见 Spring Cloud SDK 历史版本使用方法 。

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-starter</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

`spring-cloud-tsf-starter` 中包含了服务注册发现、服务路由、服务鉴权、服务限流、服务熔断、服务容错、服务监控、分布式配置、调用链功能。

2. 向 Application 类中添加注解 `@EnableTsf`：

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

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 API 注册功能
tsf.swagger.basePackage	String	否	ApplicationMainClass 所在包路径	注册 API 的扫描包路径。推荐将 ApplicationMainClass 写在外层 Package

tsf.swagger.excludePath	String	否	（空）	排除扫描的包路径
tsf.swagger.group	String	否	default	swagger docket 分组
tsf.swagger.doc.auto-startup	Boolean	否	true	是否开启 Swagger 相关暴露接口。如果设置为 false，则不暴露相关接口，以下路径将会屏蔽并返回403： • /v2/api-docs • /swagger-ui.html • /swagger-resources/ • /webjars/springfox-swagger-ui/

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 = "notes") // 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;
    }
}
```

说明

依赖 spring-cloud-tsf-starter 后，将同时为您开启查看 API 文档能力，您可以通过 `ip:port/swagger-ui.html` 页面进行 API 查看。若访问不成功，建议更新到最新版本的 SDK。

如果您不需要这个能力，可以在依赖中进行排除，示例如下：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-starter</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>springfox-swagger-ui</artifactId>
            <groupId>io.springfox</groupId>
        </exclusion>
    </exclusions>
```

```
</dependency>
```

排除后，不影响在 TSF 服务治理 > 接口列表中查询 API 的能力，仅仅不支持通过 `ip:port/swagger-ui.html` 查看。

服务监控

最近更新时间：2022-06-09 16:44:31

操作场景

该任务指导您使用服务监控功能。

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

向工程中添加 `spring-cloud-tsf-starter` 依赖并开启 `@EnableTsf` 注解，详情请参见 Demo 工程概述 文档。

> **注意**
> 如果您使用的是 1.15.0-Edgware-RELEASE/1.15.0-Finchley-RELEASE 及之前的版本，使用方法参见 Spring Cloud SDK 历史版本使用方法 。

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><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

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

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

服务容错

最近更新时间：2024-01-19 10:39:01

操作场景

TSF 摒弃了已经不再继续维护的 Hystrix 作为容错模块，在原有单一的 fallback 上新增了 failfast、failover 和 forking。请配合 TSF 其他功能一起使用。

前提条件

请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

> **说明**
> 同时请确保 SDK 版本高于1.19。

操作步骤

> **说明**
> 步骤1 和 步骤2 与其他模块一样，已经使用过其他模块的可直接跳至 步骤3 。

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-starter</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

2. 向 Application 类中添加注解 @EnableTsf：

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

3. 使用 Feign 上的 fallback 以及 fallbackFactory 功能。 TSF 容错兼容 feign 的容错功能，如果需要使用 feign 的如下降级功能，则需要关闭 Hystrix 开关。

```
@FeignClient(name = "circuit-breaker-mock-service", fallbackFactory =
HystrixClientFallbackFactory.class)
@FeignClient(name = "circuit-breaker-mock-service", fallback = FeignClientFallback.class)
```

关闭 Hystrix 开关（默认是关闭的，如果之前使用了该功能，可以删除该配置或者关闭）：

```
feign:
  hystrix:
    enabled: false
```

打开 TSF 开关：

```
feign:
  tsf:
    enabled: true
```

4. 在代码中使用容错功能。

```
// 下面省略了无关的代码
@TsfFaultTolerance(strategy = TsfFaultToleranceStragety.FAIL_OVER, parallelism = 2,
fallbackMethod = "doWorkFallback")
public void doWork() throws InterruptedException {
    String response = providerDemoService.echo("1234");
    LOG.info("consumer-demo auto test, response: [" + response + "]");
}
public void doWorkFallback() {
    System.out.println("fallback");
}
// fallbackMethod可以加也可以不加，用户可以自行选择
@TsfFaultTolerance(strategy = TsfFaultToleranceStragety.FAIL_FAST)
public void doWork2() throws InterruptedException {
    String response = providerDemoService.echo2("1234");
    LOG.info("consumer-demo auto test, response: [" + response + "]");
}
```

可以看到，TSF 的容错使用起来非常方便，您只需在需要保护的方法上增加一个注解即可（需要该 Bean 被 Spring 所管理）。

容错策略说明

failfast、failover 和 forking 容错策略是可选择的配置：

容错策略	含义	支持配置
failfast	直接失败，对于没有幂等性的下游服务推荐 failfast	无
failover	请求错了之后会重试	重试次数 maxAttempts
forking	同时发送多个请求，需要用户配置并行度，例如同时发出两个请求，哪个先返回，就把这个结果返回回去。如果第一个请求是异常，则会等另一个请求，如果全部都异常，则返回异常。	并行度 parallelism

三种容错策略均支持 fallback 方法，要求入参类型和返回值类型与原方法相同。您可以选择是否添加某一策略。
除了如上所述的基本功能外，TSF 还提供了选择容错异常的能力：

```
// 下面省略了无关的代码
@TsfFaultTolerance(strategy = TsfFaultToleranceStragety.FAIL_OVER, parallelism = 2,
ignoreExceptions = {FeignException.class}, raisedExceptions = {RuntimeException.class,
InterruptedException.class}, fallbackMethod = "doWorkFallback")
public void doWork() throws InterruptedException {
    String response = providerDemoService.echo("1234");
    LOG.info("consumer-demo auto test, response: [" + response + "]");
}

public void doWorkFallback() {
```

```
    System.out.println("fallback");
}
```

- 用户如果设置了 ignoreExceptions，且当前异常是其中一个的子类的话，则不执行容错逻辑。
- 如果用户没有设置 ignoreExceptions，或当前异常不是 ignoreExceptions 的子类且满足如下条件则执行容错逻辑：
- 用户未设置 raisedExceptions，则执行容错逻辑。
- 用户设定了 raisedExceptions，且当前异常为用户设置的 raisedExceptions 其中一个的子类，则执行容错逻辑。

开启预热功能

最近更新时间：2022-10-21 15:07:33

普通应用预热原理

扫描 @FeignClient 注解，过滤前缀带 default. 的 FeignClient，并适配当服务名为变量的情况，使用 SpringClientFactory 进行预热。

> **说明**
> 当前支持预热功能的 SDK。

使用方法

预热开关默认关闭。预热开关 key 为：

```
tsf.feign.eager-load.enabled
```

> **说明**
> value 为 true 表示打开预热，为其他值时表示关闭。

示例

启动预热，在配置文件添加以下内容：

```
tsf:
  feign:
    eager-load:
      enabled: true
```

服务监听触发回调

最近更新时间：2022-06-10 11:43:45

操作场景

服务监听允许程序监听特定服务的上下线情况，从而触发对应的业务逻辑。

前提条件

请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

> **说明**
> 若要使用服务监听触发回调功能，请确保 SDK 版本高于1.32。

添加依赖

向工程中添加 `spring-cloud-tsf-starter` 依赖并开启 `@EnableTsf` 注解。

> **注意**
> 如果您使用的是 1.15.0-Edgware-RELEASE/1.15.0-Finchley-RELEASE 及之前的版本，使用方法参见 Spring Cloud SDK 历史版本使用方法 。

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-starter</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

`spring-cloud-tsf-starter` 中包含了服务注册发现、服务路由、服务鉴权、服务限流、服务熔断、服务容错、服务监控、分布式配置、调用链功能。

2. 向 Application 类中添加注解 `@EnableTsf`：

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

使用说明

监听 bean 需要使用 `@ConsulServiceChangeListener` 或 `@LocalServiceChangeListener` 注解，并且实现 `ConsulServiceChangeCallback` 接口。callback 接口有三个入参：

- currentServices：表示当前在线的服务实例列表。
- addServices：表示和上一次变更相比，新增的实例列表。
- deleteServices：表示和上一次变更相比，删除的实例列表。

实现原理

初始化 spring bean 时，当检测到该 bean 有使用 `@ConsulServiceChangeListener` 或 `@LocalServiceChangeListener` 注解，并且有实现 `ConsulServiceChangeCallback` 接口时，开启一个定时任务，与注册中心 consul 保持长轮询，当监听服务有实例上线或下线时，会立刻触发相关事件并回调对应 bean 的 callback 方法。

> **说明**
> 每个 bean 会启动一个额外线程执行，线程池的默认大小为10，如果监听服务的数量较多或 callback 里的业务逻辑较为复杂时可能会导致线程不够用，此时可以通过 spring.cloud.consul.discovery.callbackPoolSize 参数进行调整。

相关注解说明及使用示例

- 使用 `@ConsulServiceChangeListener` 注解：

```
/**
 * @ConsulServiceChangeListener 监听当前命名空间或全局命名空间的任意服务
 * serviceName: 监听服务的服务名
 * global: 监听服务是否全局命名空间，默认 false。注：如果当前服务位于全局命名空间，global 属性无论 true 还是
false，都是监听全局命名空间的服务
 */
@Component
@ConsulServiceChangeListener(serviceName = "provider-demo", global = false)
public class ProviderDemoChangeCallback implements ConsulServiceChangeCallback {

    private static final Logger log = LoggerFactory.getLogger(ProviderDemoChangeCallback.class);

    @Override
    public void callback(List<HealthService> currentServices, List<HealthService> addServices,
List<HealthService> deleteServices) {
        log.info("current size:{}, add size:{}, delete list:{}", currentServices.size(),
addServices.size(), deleteServices.size());
        log.info("current list:{}, add list:{}, delete list:{}", currentServices, addServices,
deleteServices);
        for (HealthService healthService: currentServices) {
            // 可以从 meta 信息中获取节点对应的 TSF 信息，如部署组id、应用id、包版本
            Map<String, String> meta = healthService.getService().getMeta();
                        String groupId = meta.get("TSF_GROUP_ID");
            String applicationId = meta.get("TSF_APPLICATION_ID");
            String progVersion = meta.get("TSF_PROG_VERSION");
        }
    }
}
```

- 使用 `@LocalServiceChangeListener` 注解：

```
/**
 * @LocalServiceChangeListener 监听当前服务(spring.application.name)
 * excludeLocalInstance: 回调的 currentServices 是否包含当前实例，默认为 false
 */
@Component
@LocalServiceChangeListener(excludeLocalInstance = false)
public class LocalServiceChangeCallback implements ConsulServiceChangeCallback {

    private static final Logger log = LoggerFactory.getLogger(LocalServiceChangeCallback.class);

    @Override
```

```
    public void callback(List<HealthService> currentServices, List<HealthService> addServices,
List<HealthService> deleteServices) {
        log.info("current size:{}, add size:{}, delete list:{}", currentServices.size(),
addServices.size(), deleteServices.size());
        log.info("current list:{}, add list:{}, delete list:{}", currentServices, addServices,
deleteServices);
    }
 }
```

HTTP2 应用开发

最近更新时间：2023-05-24 10:40:16

操作场景

本文以 provider-demo 和 consumer-demo 两个工程为例为您介绍 TSF 应用配置 http2 的操作方法。

前提条件

- 安装1.8或以上版本 JDK
- 下载 TSF Demo 工程（springboot 版本2.0+，tomcat 版本8.5+）

> **说明**
> - 推荐使用 1.29.0-Finchley-RELEASE。
> - 本文以 springboot-2.0.9.RELEASE 和 tomcat-8.5.56 为例。

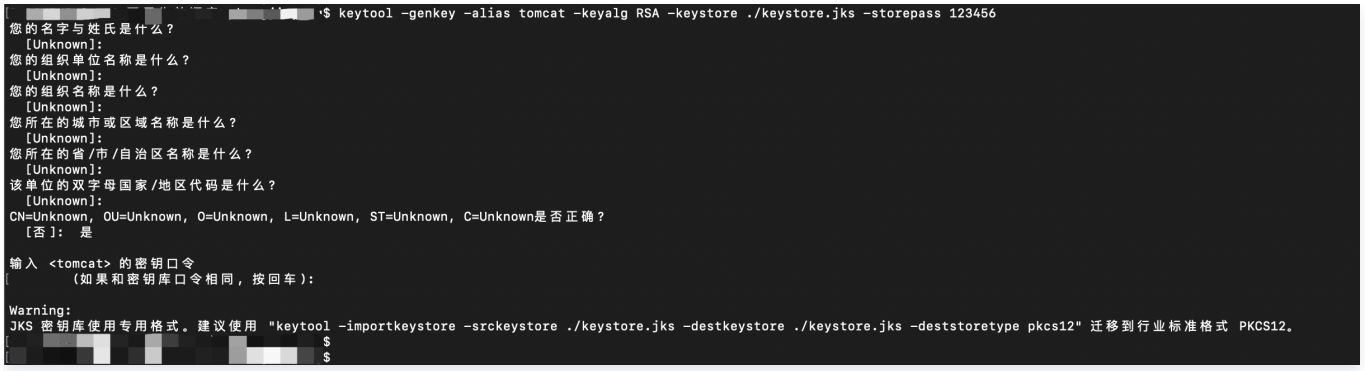
操作步骤

步骤1：制作 SSL 证书

通过 JDK 自带的 keytool 执行以下命令：

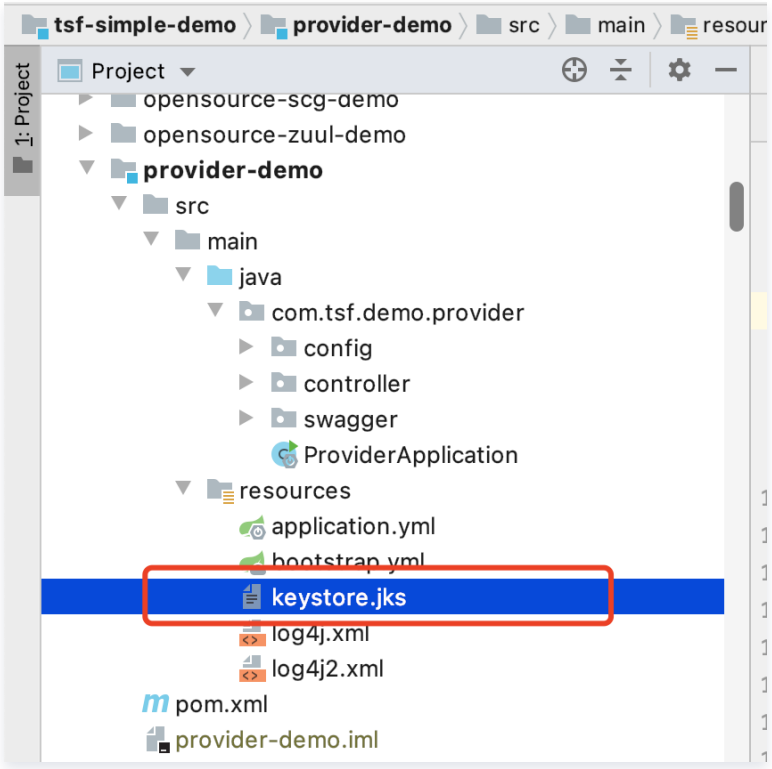
```
keytool -genkey -alias tomcat -keyalg RSA -keystore ./keystore.jks -storepass 123456
```

执行成功后当前目录下会生成 keystore.jks 证书文件。

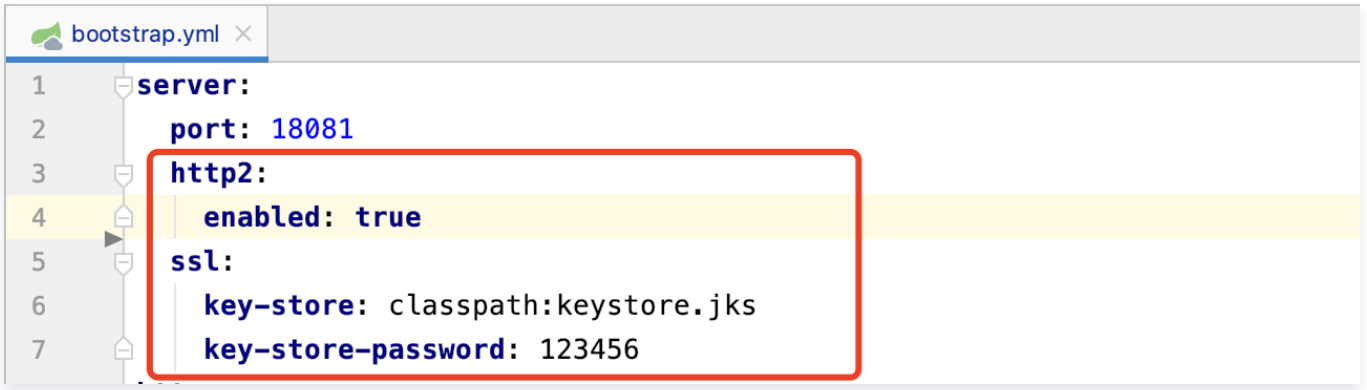


步骤2：改造 provider-demo 工程

1. 复制 jks 证书文件到 provider-demo 工程的 resources 目录下。



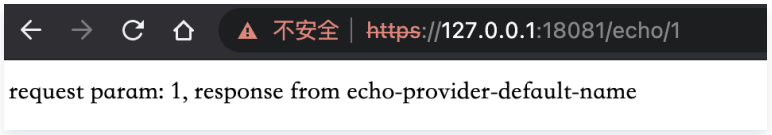
2. 修改 spring 配置文件，在 bootstrap.yaml 文件中增加如下配置。



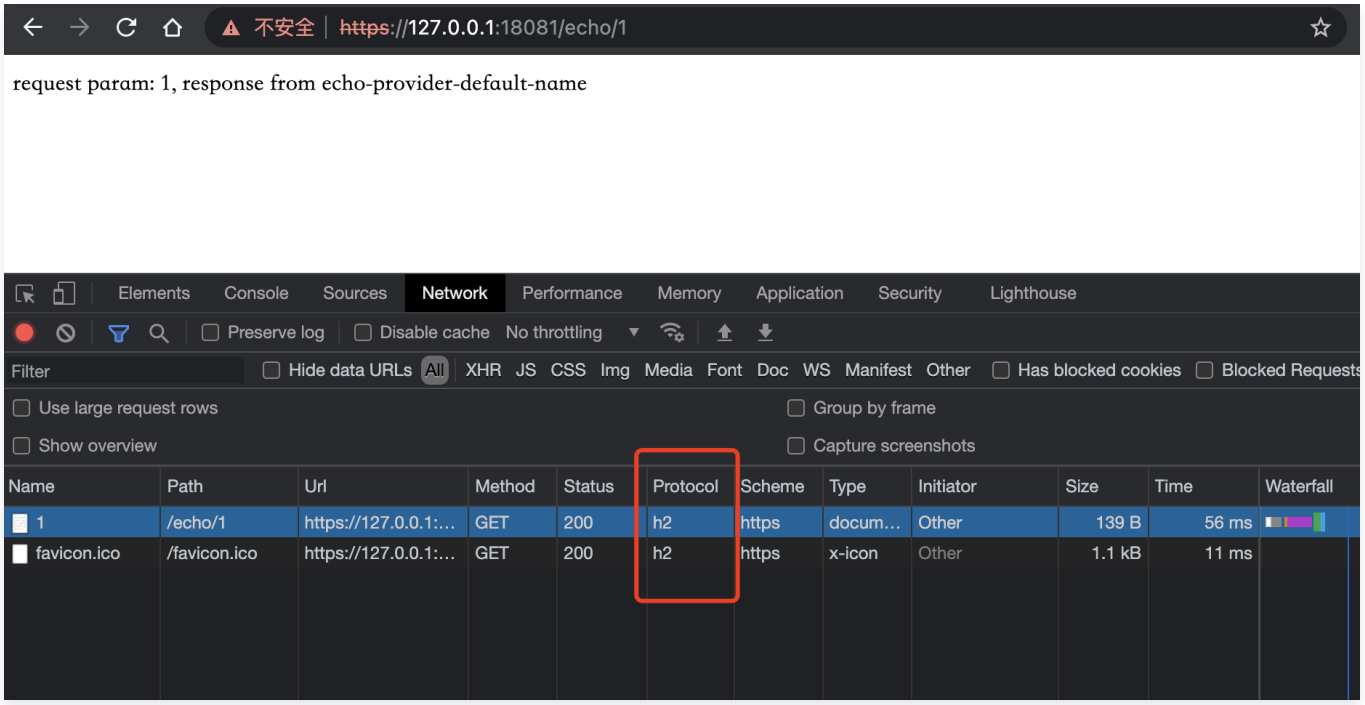

```
server.http2.enabled=true
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password: 123456
```

3. 启动 provider-demo，浏览器访问 `https://127.0.0.1:18081/echo/1` 。
Chrome 可能会提示“您的连接不是私密连接”。

只需单击任意空白处，键入“thisisunsafe”即可。



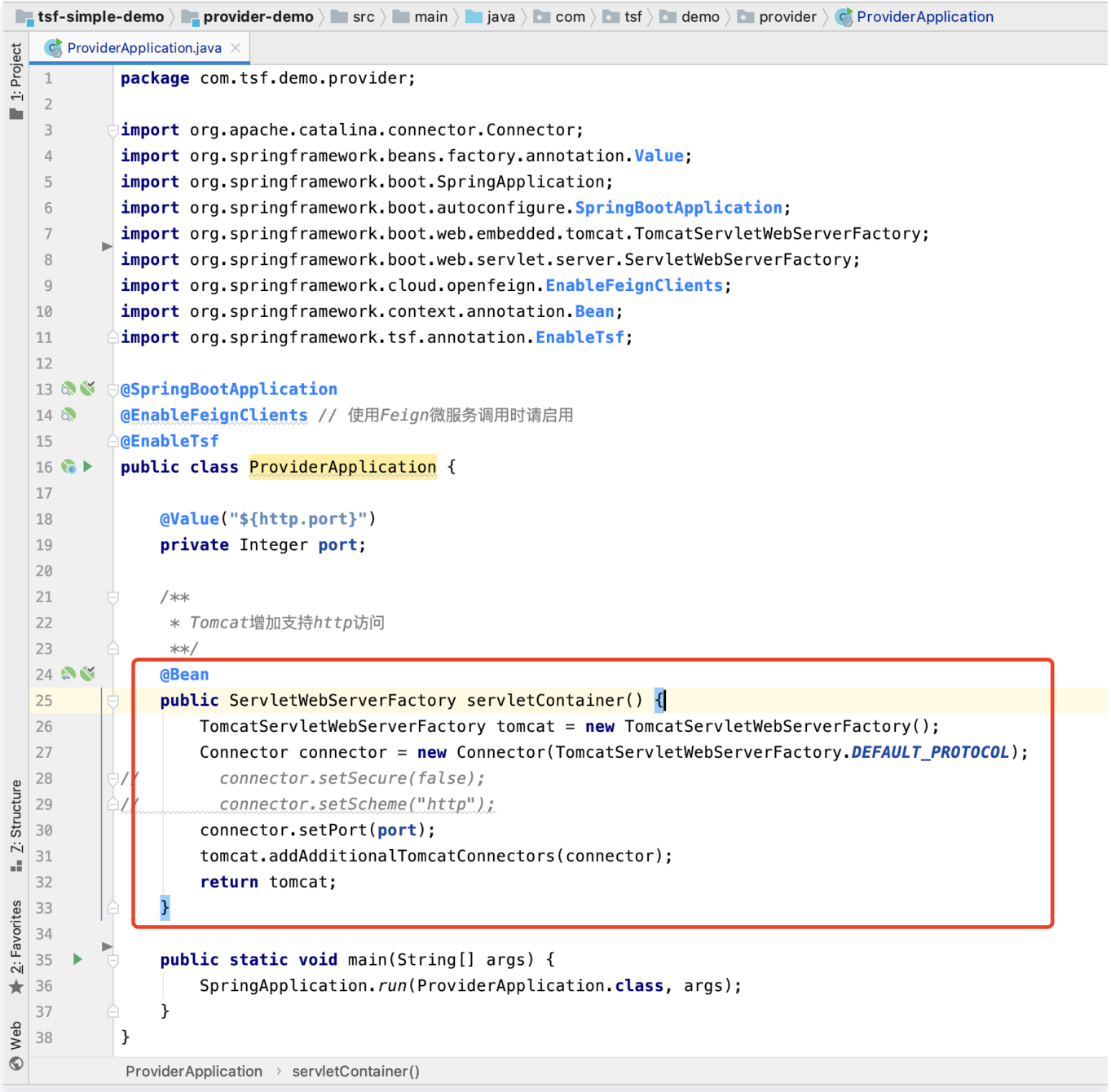
打开 Console，Protocol 的值为 h2 表示配置成功。



参见资料：

- Spring 配置 http2 文档：howto-configure-http2
- Tomcat 版本差异文档：whichversion

4. 开启 http 访问端口（可选）。
   由于此时只能通过 https 方式访问，可加入 Tomcat 配置开放新端口来支持 http 访问。
   4.1 在启动类中增加 Bean。




```
1  package com.tsf.demo.provider;
2
3  import org.apache.catalina.connector.Connector;
4  import org.springframework.beans.factory.annotation.Value;
5  import org.springframework.boot.SpringApplication;
6  import org.springframework.boot.autoconfigure.SpringBootApplication;
7  import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
8  import org.springframework.boot.web.servlet.server.ServletWebServerFactory;
9  import org.springframework.cloud.openfeign.EnableFeignClients;
10 import org.springframework.context.annotation.Bean;
11 import org.springframework.tsf.annotation.EnableTsf;
12
13 @SpringBootApplication
14 @EnableFeignClients // 使用Feign微服务调用时请启用
15 @EnableTsf
16 public class ProviderApplication {
17
18     @Value("${http.port}")
19     private Integer port;
20
21     /**
22      * Tomcat增加支持http访问
23      **/
24     @Bean
25     public ServletWebServerFactory servletContainer() {
26         TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory();
27         Connector connector = new Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
28 //          connector.setSecure(false);
29 //          connector.setScheme("http");
30         connector.setPort(port);
31         tomcat.addAdditionalTomcatConnectors(connector);
32         return tomcat;
33     }
34
35     public static void main(String[] args) {
36         SpringApplication.run(ProviderApplication.class, args);
37     }
38 }
```

```
package com.tsf.demo.provider;

import org.apache.catalina.connector.Connector;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.servlet.server.ServletWebServerFactory;
import org.springframework.cloud.openfeign.EnableFeignClients;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.tsf.annotation.EnableTsf;

@SpringBootApplication
@EnableFeignClients // 使用Feign微服务调用时请启用
@EnableTsf
public class ProviderApplication {

    @Value("${http.port}")
    private Integer port;

    /**
     * Tomcat增加支持http访问
     **/
    @Bean
    public ServletWebServerFactory servletContainer() {
        TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory();
        Connector connector = new
Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
//        connector.setSecure(false);
//        connector.setScheme("http");
        connector.setPort(port);
        tomcat.addAdditionalTomcatConnectors(connector);
        return tomcat;
    }

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

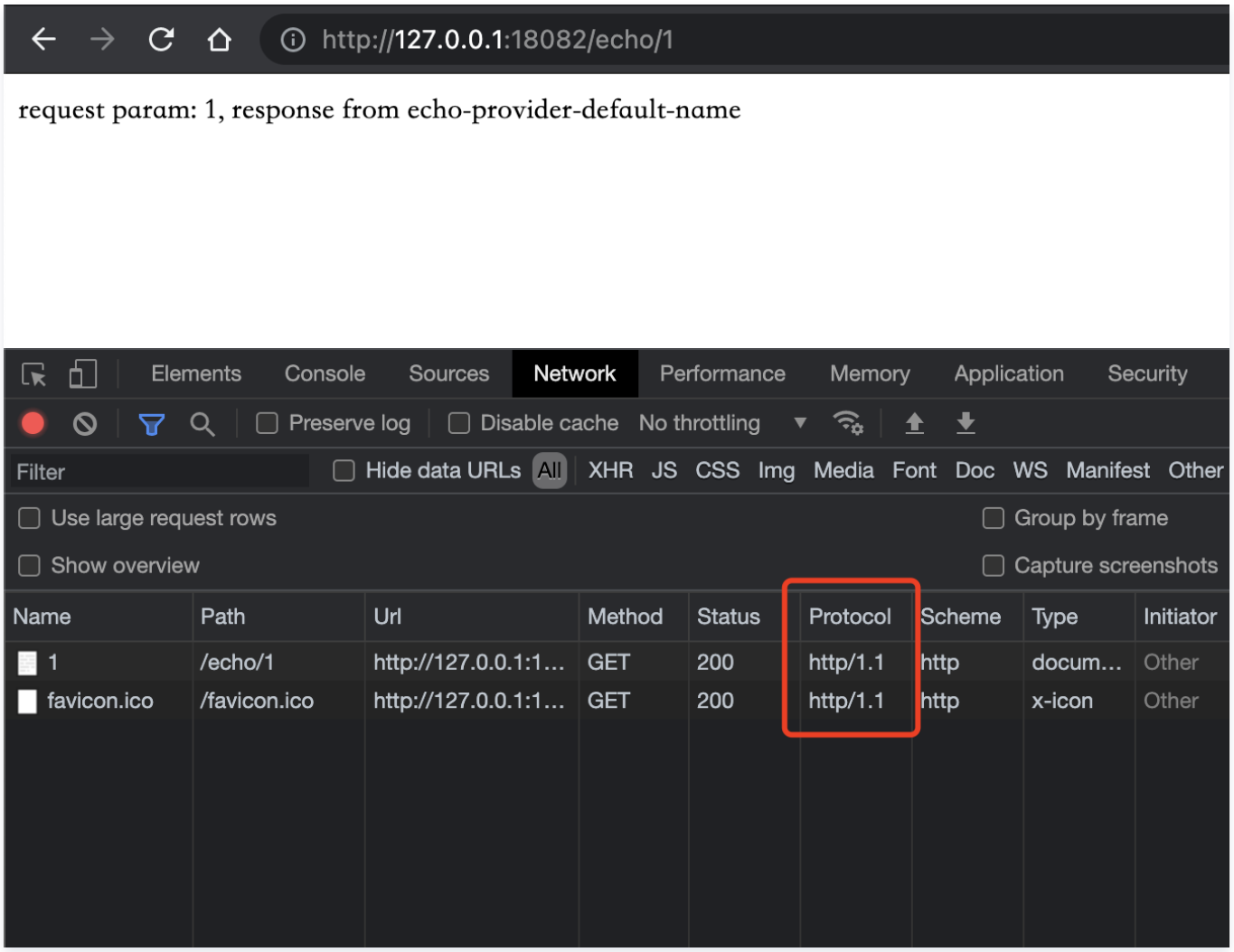
5. 在 Bootstrap.yml 配置文件中增加自定义配置。

bootstrap.yml

```
server:
  port: 18081
  http2:
    enabled: true
  ssl:
    key-store: classpath:keystore.jks
    key-store-password: 123456

http:
  port: 18082
```

```
http.port=18082
```

6. 重启 provider-demo，浏览器访问 `http://127.0.0.1:18082/echo/1` 。



此时 provider-demo 完成支持 https 和 http（通过不同端口访问），其中 https 访问时使用 http2 协议。

参见资料：Spring 配置 Tomcat 代码示例：SampleTomcatTwoConnectorsApplication.java 。

步骤3：改造 consumer-demo 工程

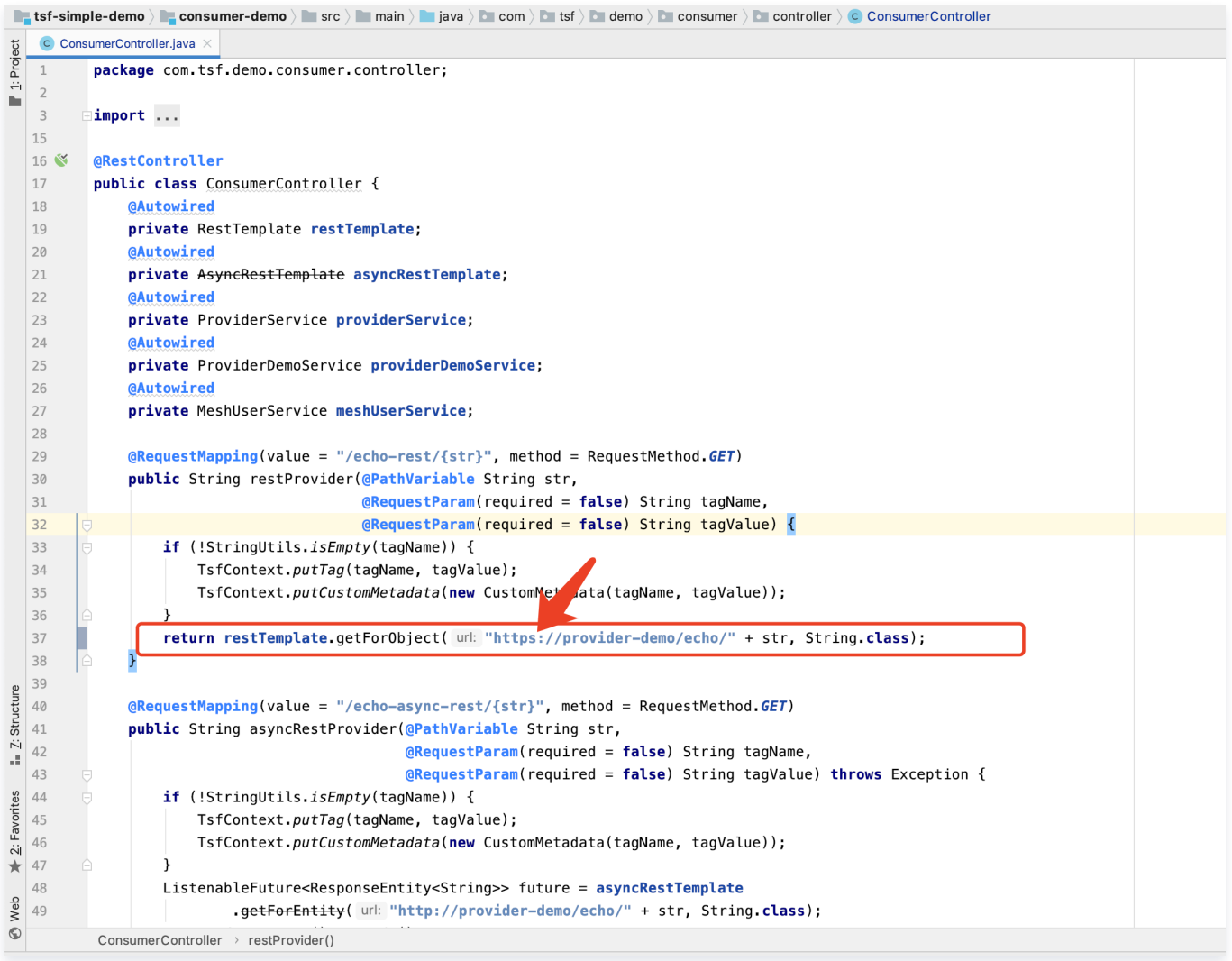
1. proxy 中的 @FeignClient 注解需指定 https 方式访问。

ProviderDemoService.java

```java
package com.tsf.demo.consumer.proxy;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

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

    @RequestMapping(value = "/echoTracer/{str}", method = RequestMethod.GET)
    String echoTracer(@PathVariable("str") String str);

    @RequestMapping(value = "/echo/error/{str}", method = RequestMethod.GET)
    String echoError(@PathVariable("str") String str);

    @RequestMapping(value = "/echo/slow/{str}", method = RequestMethod.GET)
    String echoSlow(@PathVariable("str") String str);
}
```

```
@FeignClient(name = "https://provider-demo")
```

2. RestTemplate 同理。



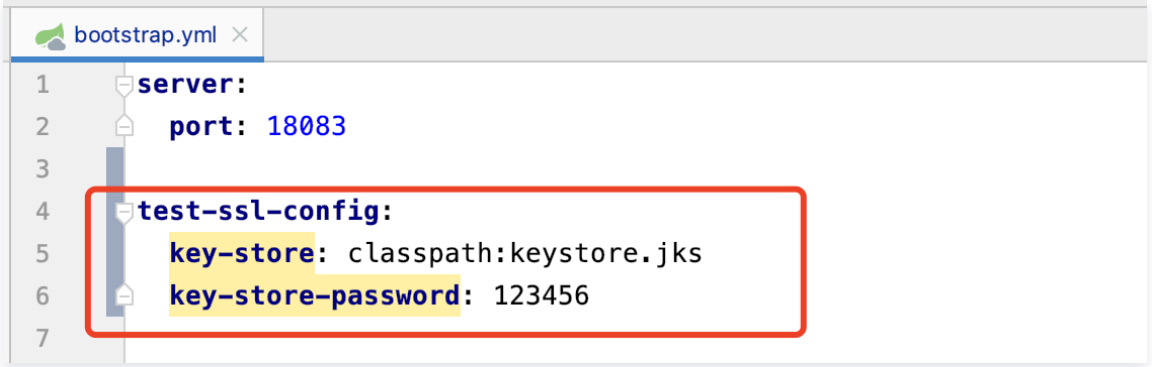

```java
package com.tsf.demo.consumer.controller;

import ...

@RestController
public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private AsyncRestTemplate asyncRestTemplate;
    @Autowired
    private ProviderService providerService;
    @Autowired
    private ProviderDemoService providerDemoService;
    @Autowired
    private MeshUserService meshUserService;

    @RequestMapping(value = "/echo-rest/{str}", method = RequestMethod.GET)
    public String restProvider(@PathVariable String str,
                               @RequestParam(required = false) String tagName,
                               @RequestParam(required = false) String tagValue) {
        if (!StringUtils.isEmpty(tagName)) {
            TsfContext.putTag(tagName, tagValue);
            TsfContext.putCustomMetadata(new CustomMetadata(tagName, tagValue));
        }
        return restTemplate.getForObject( url: "https://provider-demo/echo/" + str, String.class);
    }

    @RequestMapping(value = "/echo-async-rest/{str}", method = RequestMethod.GET)
    public String asyncRestProvider(@PathVariable String str,
                                    @RequestParam(required = false) String tagName,
                                    @RequestParam(required = false) String tagValue) throws Exception {
        if (!StringUtils.isEmpty(tagName)) {
            TsfContext.putTag(tagName, tagValue);
            TsfContext.putCustomMetadata(new CustomMetadata(tagName, tagValue));
        }
        ListenableFuture<ResponseEntity<String>> future = asyncRestTemplate
                .getForEntity( url: "http://provider-demo/echo/" + str, String.class);
```

```
restTemplate.getForObject("https://provider-demo/echo/" + str, String.class);
```

3. 通过注入不同的 bean 选择是否使用SSL证书认证（以下步骤二选一）：

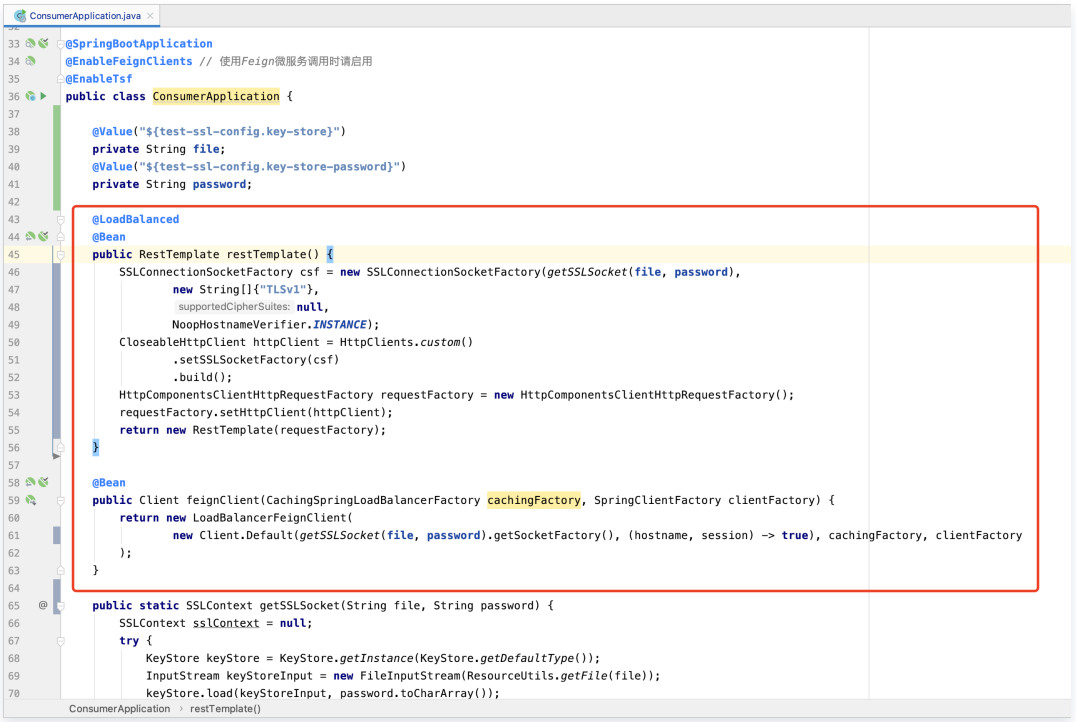
使用 SSL 证书认证访问方式

1. 复制 jks 证书文件到 consumer-demo 工程的 resources 目录。
2. spring 配置文件中增加自定义配置。



```
test-ssl-config.key-store=classpath:keystore.jks
test-ssl-config.key-store-password: 123456
```

3. 修改 bean（restTemplate、feignClient）




```
package com.tsf.demo.consumer;

import feign.Client;
import org.apache.http.conn.ssl.NoopHostnameVerifier;
import org.apache.http.conn.ssl.SSLConnectionSocketFactory;
import org.apache.http.conn.ssl.TrustSelfSignedStrategy;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.ssl.SSLContexts;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.netflix.ribbon.SpringClientFactory;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.cloud.openfeign.ribbon.CachingSpringLoadBalancerFactory;
import org.springframework.cloud.openfeign.ribbon.LoadBalancerFeignClient;
import org.springframework.context.annotation.Bean;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.tsf.annotation.EnableTsf;
import org.springframework.util.ResourceUtils;
import org.springframework.web.client.AsyncRestTemplate;
import org.springframework.web.client.RestTemplate;

import javax.net.ssl.*;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
```

```
import java.security.*;
import java.security.cert.CertificateException;

@SpringBootApplication
@EnableFeignClients // 使用Feign微服务调用时请启用
@EnableTsf
public class ConsumerApplication {

    @Value("${test-ssl-config.key-store}")
    private String file;
    @Value("${test-ssl-config.key-store-password}")
    private String password;

    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() {
        SSLConnectionSocketFactory csf = new
SSLConnectionSocketFactory(getSSLSocket(file, password),
                new String[]{"TLSv1"},
                null,
                NoopHostnameVerifier.INSTANCE);
        CloseableHttpClient httpClient = HttpClients.custom()
                .setSSLSocketFactory(csf)
                .build();
        HttpComponentsClientHttpRequestFactory requestFactory = new
HttpComponentsClientHttpRequestFactory();
        requestFactory.setHttpClient(httpClient);
        return new RestTemplate(requestFactory);
    }

    @Bean
    public Client feignClient(CachingSpringLoadBalancerFactory cachingFactory,
SpringClientFactory clientFactory) {
        return new LoadBalancerFeignClient(
            new Client.Default(getSSLSocket(file, password).getSocketFactory(),
(hostname, session) -> true), cachingFactory, clientFactory
        );
    }

    public static SSLContext getSSLSocket(String file, String password) {
        SSLContext sslContext = null;
        try {
            KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
            InputStream keyStoreInput = new
FileInputStream(ResourceUtils.getFile(file));
            keyStore.load(keyStoreInput, password.toCharArray());

            KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
            InputStream trustStoreInput = new
FileInputStream(ResourceUtils.getFile(file));
            trustStore.load(trustStoreInput, null);
            sslContext = SSLContexts.custom().loadKeyMaterial(keyStore,
password.toCharArray())
                    .loadTrustMaterial(trustStore, new
TrustSelfSignedStrategy()).build();
        } catch (NoSuchAlgorithmException | KeyManagementException | KeyStoreException
| CertificateException | IOException | UnrecoverableKeyException e) {
```

```
                e.printStackTrace();
            }
            return sslContext;
        }

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

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

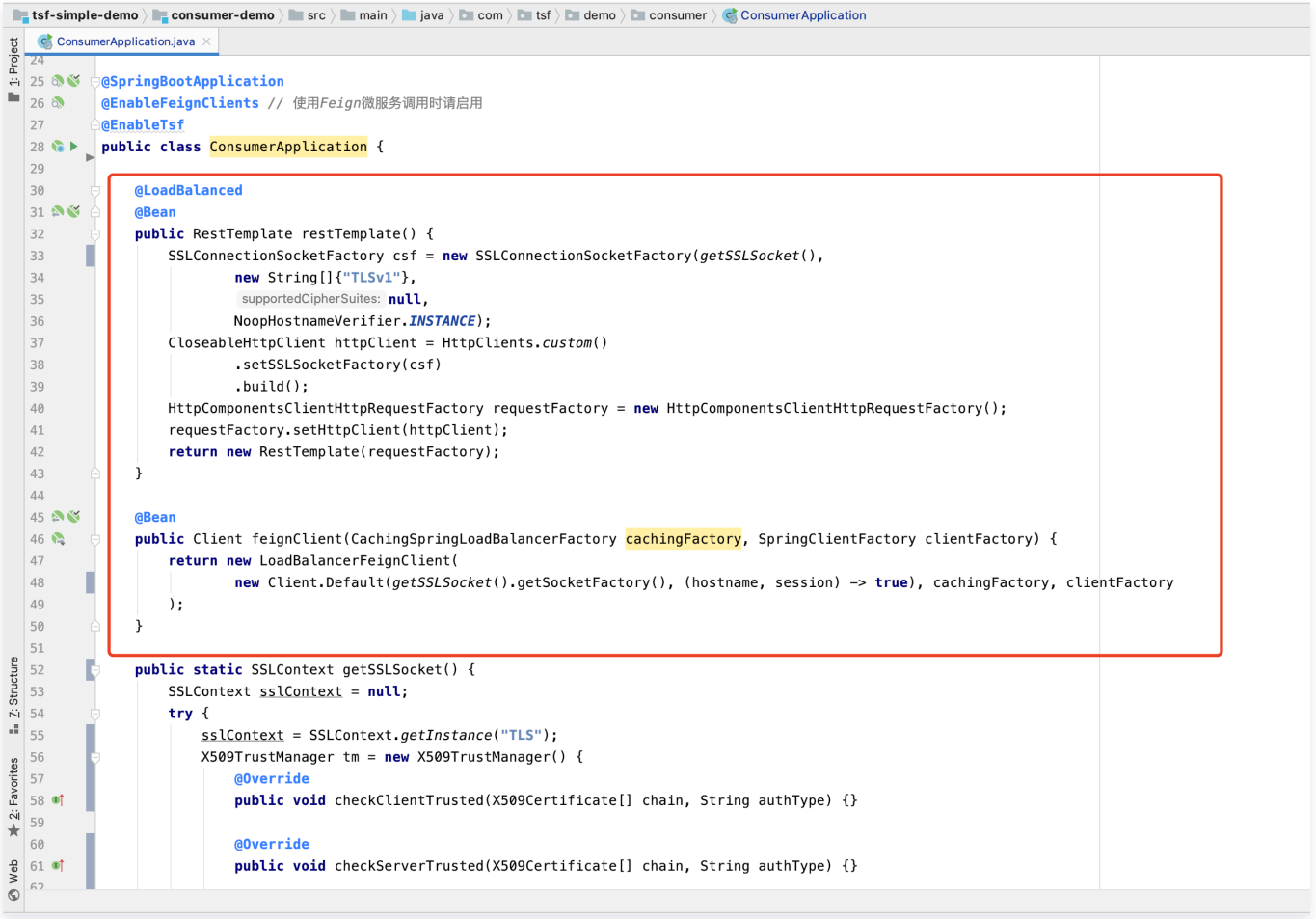
忽略 SSL 认证方式

只需要修改 bean（restTemplate、feignClient）

ConsumerApplication.java

```
        return new RestTemplate(requestFactory);
    }

    @Bean
    public Client feignClient(CachingSpringLoadBalancerFactory cachingFactory, SpringClientFactory clientFactory) {
        return new LoadBalancerFeignClient(
                new Client.Default(getSSLSocket(file, password).getSocketFactory(), (hostname, session) -> true), cachingFactory, clientFactory
        );
    }

    public static SSLContext getSSLSocket(String file, String password) {
        SSLContext sslContext = null;
        try {
            KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
            InputStream keyStoreInput = new FileInputStream(ResourceUtils.getFile(file));
            keyStore.load(keyStoreInput, password.toCharArray());

            KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
            InputStream trustStoreInput = new FileInputStream(ResourceUtils.getFile(file));
            trustStore.load(trustStoreInput,  password: null);
            sslContext = SSLContexts.custom().loadKeyMaterial(keyStore, password.toCharArray())
                    .loadTrustMaterial(trustStore, new TrustSelfSignedStrategy()).build();
        } catch (NoSuchAlgorithmException | KeyManagementException | KeyStoreException | CertificateException | IOException | UnrecoverableKeyException e) {
            e.printStackTrace();
        }
        return sslContext;
    }

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

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

ConsumerApplication › restTemplate()



```java
@SpringBootApplication
@EnableFeignClients // 使用Feign微服务调用时请启用
@EnableTsf
public class ConsumerApplication {

    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() {
        SSLConnectionSocketFactory csf = new SSLConnectionSocketFactory(getSSLSocket(),
                new String[]{"TLSv1"},
                supportedCipherSuites: null,
                NoopHostnameVerifier.INSTANCE);
        CloseableHttpClient httpClient = HttpClients.custom()
                .setSSLSocketFactory(csf)
                .build();
        HttpComponentsClientHttpRequestFactory requestFactory = new HttpComponentsClientHttpRequestFactory();
        requestFactory.setHttpClient(httpClient);
        return new RestTemplate(requestFactory);
    }

    @Bean
    public Client feignClient(CachingSpringLoadBalancerFactory cachingFactory, SpringClientFactory clientFactory) {
        return new LoadBalancerFeignClient(
                new Client.Default(getSSLSocket().getSocketFactory(), (hostname, session) -> true), cachingFactory, clientFactory
        );
    }

    public static SSLContext getSSLSocket() {
        SSLContext sslContext = null;
        try {
            sslContext = SSLContext.getInstance("TLS");
            X509TrustManager tm = new X509TrustManager() {
                @Override
                public void checkClientTrusted(X509Certificate[] chain, String authType) {}

                @Override
                public void checkServerTrusted(X509Certificate[] chain, String authType) {}
```

说明

与第一种方式的差异在 getSSLSocket() 方法。

```java
package com.tsf.demo.consumer;

import feign.Client;
import org.apache.http.conn.ssl.NoopHostnameVerifier;
import org.apache.http.conn.ssl.SSLConnectionSocketFactory;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.netflix.ribbon.SpringClientFactory;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.cloud.openfeign.ribbon.CachingSpringLoadBalancerFactory;
import org.springframework.cloud.openfeign.ribbon.LoadBalancerFeignClient;
import org.springframework.context.annotation.Bean;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.tsf.annotation.EnableTsf;
import org.springframework.web.client.AsyncRestTemplate;
import org.springframework.web.client.RestTemplate;

import javax.net.ssl.*;
import java.security.*;
```

```
import java.security.cert.X509Certificate;

@SpringBootApplication
@EnableFeignClients // 使用Feign微服务调用时请启用
@EnableTsf
public class ConsumerApplication {

        @LoadBalanced
        @Bean
        public RestTemplate restTemplate() {
                SSLConnectionSocketFactory csf = new SSLConnectionSocketFactory(getSSLSocket(),
                                new String[]{"TLSv1"},
                                null,
                                NoopHostnameVerifier.INSTANCE);
                CloseableHttpClient httpClient = HttpClients.custom()
                                .setSSLSocketFactory(csf)
                                .build();
                HttpComponentsClientHttpRequestFactory requestFactory = new
HttpComponentsClientHttpRequestFactory();
                requestFactory.setHttpClient(httpClient);
                return new RestTemplate(requestFactory);
        }

        @Bean
        public Client feignClient(CachingSpringLoadBalancerFactory cachingFactory,
SpringClientFactory clientFactory) {
                return new LoadBalancerFeignClient(
                        new Client.Default(getSSLSocket().getSocketFactory(), (hostname,
session) -> true), cachingFactory, clientFactory
                );
        }

        public static SSLContext getSSLSocket() {
                SSLContext sslContext = null;
                try {
                        sslContext = SSLContext.getInstance("TLS");
                        X509TrustManager tm = new X509TrustManager() {
                                @Override
                                public void checkClientTrusted(X509Certificate[] chain, String
authType) {}

                                @Override
                                public void checkServerTrusted(X509Certificate[] chain, String
authType) {}

                                @Override
                                public X509Certificate[] getAcceptedIssuers() {
                                        return null;
                                }
                        };
                        sslContext.init(null, new TrustManager[]{tm}, null);
                } catch (NoSuchAlgorithmException | KeyManagementException e) {
                        e.printStackTrace();
                }
                return sslContext;
        }
```

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

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

4. 验证结果。

4.1 启动 consumer-demo，浏览器访问 `http://127.0.0.1:18083/echo-feign/123` 。

4.2 浏览器访问 `http://127.0.0.1:18083/echo-rest/123` 。

本地使用 Agent 进行无侵入应用开发

最近更新时间：2024-05-06 17:24:22

操作场景

本文主要介绍 Spring Cloud 开源项目如何在本地使用 Agent 进行无侵入应用开发。

前提条件

请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

> **说明**
> 同时请确保 SpringCloud SDK 版本高于2020。

操作步骤

如果对 Agent 有疑问，可以先参见 Agent 插件功能说明 。

> **说明**
> 步骤1 和 步骤2 与其他模块一样，已经使用过其他模块的可直接跳至 步骤3 。

1. **在本地确保依赖了 2020 相关的 SDK 依赖。**
   在 pom.xml 中添加以下代码：

```
<properties>
    <spring-boot-dependencies.version>2.4.6</spring-boot-dependencies.version>
    <spring-cloud-dependencies.version>2020.0.2</spring-cloud-dependencies.version>
    <consul.version>1.4.2</consul.version>
    <feign-reactor.version>3.1.1</feign-reactor.version>
</properties>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${spring-boot-dependencies.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud-dependencies.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>

        <dependency>
            <groupId>com.ecwid.consul</groupId>
            <artifactId>consul-api</artifactId>
            <version>${consul.version}</version>
        </dependency>
    </dependencies>
```

```
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-bootstrap</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>2.9.2</version>
    </dependency>

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger-ui</artifactId>
        <version>2.9.2</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-consul-discovery</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-consul-config</artifactId>
    </dependency>

    <dependency>
        <groupId>io.projectreactor.netty</groupId>
```

```
        <artifactId>reactor-netty</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
    </dependency>

    <dependency>
        <groupId>com.playtika.reactivefeign</groupId>
        <artifactId>feign-reactor-core</artifactId>
        <version>${feign-reactor.version}</version>
    </dependency>

    <dependency>
        <groupId>com.playtika.reactivefeign</groupId>
        <artifactId>feign-reactor-webclient</artifactId>
        <version>${feign-reactor.version}</version>
    </dependency>

    <dependency>
        <groupId>com.playtika.reactivefeign</groupId>
        <artifactId>feign-reactor-cloud</artifactId>
        <version>${feign-reactor.version}</version>
    </dependency>

    <dependency>
        <groupId>com.playtika.reactivefeign</groupId>
        <artifactId>feign-reactor-spring-configuration</artifactId>
        <version>${feign-reactor.version}</version>
    </dependency>

    <dependency>
        <groupId>com.squareup.okhttp3</groupId>
        <artifactId>okhttp</artifactId>
        <version>3.14.9</version>
    </dependency>

</dependencies>
```

2. 向 Application 类中添加相关开源启动注解。

```
// 下面省略了无关的代码
@SpringBootApplication
@EnableSwagger2
@EnableDiscoveryClient
public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
}
```

3. 在启动应用时使用服务 Agent。

下载 service-agent-release.tar，并解压。在 IDE 中启动，通过 VM options 配置启动参数 `-javaagent:service-agent-release/femas-agent/femas-agent.jar`，通过 main 方法直接启动。

4. 在启动应用时使用可观测 Agent。

下载 ot-agent-release.tar，并解压。在 IDE 中启动，通过 VM options 配置启动参数

```
-javaagent:ot-agent-release/opentelemetry-javaagent.jar -Dotel.javaagent.extensions=ot-agent-release/femas-trace-opentelemetry.jar
```

，通过 main 方法直接启动。

> ⚠ 注意
>
> 如果想要多个 Agent 一起使用，可以使用命令如：
>
> ```
> -javaagent:service-agent-release/femas-agent/femas-agent.jar -javaagent:ot-agent-release/opentelemetry-javaagent.jar -Dotel.javaagent.extensions=ot-agent-release/femas-trace-opentelemetry.jar
> ```

Spring Cloud 2020 & 2021 SDK 使用说明

最近更新时间：2025-06-25 10:26:11

使用说明及限制

1. 添加 TSF 依赖
   - 通用：在 `pom.xml` 中使用 spring-cloud-tsf-dependencies 做依赖管理（以下两种方式中选一种即可）
     - 修改 parent 依赖

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-dependencies</artifactId>
    <version><!-- 调整为 SDK 长期维护（LTS）版本号 --></version>
</parent>
```

     - dependencyManagement 中添加（适用于项目有统一的 parent 依赖）

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.tencent.tsf</groupId>
      <artifactId>spring-cloud-tsf-dependencies</artifactId>
      <version><!-- 调整为 SDK 长期维护（LTS）版本号 --></version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

> **SDK 版本号请参考：**
> - 2020版本
> - 2021版本

   - 普通微服务：向工程中添加 `spring-cloud-tsf-starter` 依赖。

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

`spring-cloud-tsf-starter` 中包含了服务注册发现、服务路由、服务鉴权、服务限流、服务熔断、服务容错、服务监控、分布式配置功能。

   - 网关：向工程中添加 `spring-cloud-tsf-msgw-scg` 依赖。

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

> **注意**

> TSF Spring Cloud 2020 开始不再支持 zuul1 网关。

2. 向 Application 类中添加注解 `@EnableTsf` 。

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

> **注意**
> Spring Cloud Finchley/Greenwich/Hoxton 支持 @EnableTsfAuth 等注解的单独使用，仅开启部分功能，Spring Cloud 2020 不再支持，添加依赖后即支持全部 SDK 功能。

3. 配置日志 pattern。Spring Cloud 2020（Spring Boot 2.4）开始，日志格式有所变化。如果需要用日志配置项中的 Spring Boot 格式采集日志，需要对 pattern 进行以下设置。如果采集格式使用自定义 logback 格式/自定义 log4j2 格式或其他自定义规则格式，也需要参考该 pattern 进行对应的调整。

   3.1 无 xml 日志配置文件，输出格式使用 logging.pattern.level 配置进行设置：

```
logging:
  pattern:
    level: "%-5level [${spring.application.name},%mdc{trace_id},%mdc{span_id},]"
```

   3.2 输出格式使用 logback.xml 配置。logback.xml 中的 pattern 需要进行以下配置，其中 %X{trace_id} 表示调用链 trace id 的占位符，%X{span_id} 表示调用链 span id 的占位符。

```
%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level [${spring.application.name},%X{trace_id},%X{span_id},]
${PID} --- [%thread] %logger : %msg%n
```

   3.3 输出格式使用 log4j2.xml 配置。log4j2.xml 中的 pattern 需要进行以下配置，其中 %X{trace_id} 表示调用链 trace id 的占位符，%X{span_id} 表示调用链 span id 的占位符。

```
%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level
[${spring:spring.application.name},%X{trace_id},%X{span_id},] %pid --- [%thread] %logger :
%msg%n
```

   3.4 输出格式使用其他方式：参考以上 logback 和 log4j2 配置调整 pattern。

4. 配置可观测 Agent。Spring Cloud 2020（Spring Boot 2.4）开始，TSF 采用可观测 Agent 生成调用链、监控信息。需要进行以下配置：
   - 虚拟机：部署时展开高级选项，勾选“Agent 配置”中的“可观测 Agent”。

- 容器（通过 TSF 控制台制作的镜像）部署时展开高级选项，勾选“Agent 配置”中的“可观测 Agent”。

- 容器（手动制作的镜像）：Dockerfile 中添加 ot-agent-release.tar ，并通过 VM options 配置启动参数：

```
-javaagent:ot-agent-release/opentelemetry-javaagent.jar -Dotel.javaagent.extensions=ot-agent-release/femas-trace-opentelemetry.jar -Dotel.traces.exporter=none
```

Spring Cloud（SpringCloud Tencent）服务注册与发现

最近更新时间：2024-12-10 16:08:02

操作场景

本文介绍在本地开发 Java 应用，通过 Spring Cloud Tencent 的方式接入 TSF 独占注册配置中心，并实现服务注册与发现。

前提条件

1. 在开始开发前，请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven。
2. 已创建 TSF 独占注册配置中心实例（TSE 北极星），操作步骤详细参见 引擎管理 。

操作步骤

步骤1：引入服务注册与发现的依赖

1. 引入 spring cloud tencent 依赖

修改应用根目录下的 pom.xml，添加 `dependencyManagement`：

```
<dependencyManagement>
    <dependencies>
        <!-- Spring Cloud Tencent TSF Dependencies -->
        <dependency>
            <groupId>com.tencent.cloud</groupId>
            <artifactId>spring-cloud-tencent-dependencies</artifactId>
            <version>${revision}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>

        <!-- Spring Cloud Dependencies -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring.cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

注意：

springcloud tencent 和 springcloud 版本如下：

```
<revision>2.0.0.0-2022.0.5-RC4</revision>
<spring.cloud.version>2022.0.5</spring.cloud.version>
```

2. 引入 spring cloud tencent tsf starter

```
<dependency>
    <groupId>com.tencent.cloud</groupId>
    <artifactId>spring-cloud-starter-tencent-all</artifactId>
</dependency>
```

步骤2：部署应用

参见 如何打 FatJar 包 将应用工程打包，并打包好的 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。部署相关操作可参见 容器托管应用 或 虚拟机托管应用 。

步骤3： 服务调用

在 Spring Cloud 中可通过 **RestTemplate** 或者 **Feign** 发起服务调用。

- **RestTemplate**

  只需要在实例化 `RestTemplate` 的地方加上 `@LoadBalanced` 注解即可，示例代码如下：

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

- **Feign**

  通过 `Feign` 框架调用，按照标准的 `Feign` 方式即可。

配置管理

最近更新时间：2024-12-10 16:08:02

操作场景

本文介绍在本地开发 Java 应用，通过 Spring Cloud Tencent 的方式接入 TSF 独占注册配置中心，使用配置管理功能。

前提条件

1. 在开始开发前，请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven。
2. 已创建 TSF 独占注册配置中心实例（TSE 北极星），操作步骤详细可参见 引擎管理 。

操作步骤

步骤1：引入服务注册与发现的依赖

1. 引入 spring cloud tencent 依赖

修改应用根目录下的 pom.xml，添加 `dependencyManagement`：

```
    <dependencyManagement>
        <dependencies>
            <!-- Spring Cloud Tencent TSF Dependencies -->
            <dependency>
                <groupId>com.tencent.cloud</groupId>
                <artifactId>spring-cloud-tencent-dependencies</artifactId>
                <version>${revision}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>

            <!-- Spring Cloud Dependencies -->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring.cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

> ⚠ **注意：**
> springcloud tencent 和 springcloud 版本如下：
>
> ```
> <revision>2.0.0.0-2022.0.5-RC4</revision>
> <spring.cloud.version>2022.0.5</spring.cloud.version>
> ```

2. 引入 spring cloud tencent tsf starter

```
<dependency>
    <groupId>com.tencent.cloud</groupId>
    <artifactId>spring-cloud-starter-tencent-all</artifactId>
```

```
</dependency>
```

步骤2：在代码中使用配置

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
- `@ConfigurationProperties` 则更适用于有较多配置的情况

1. 通过 `@Value` 注入

```
@Value("${timeout:1000}")
private int timeout;
```

2. 通过 `@ConfigurationProperties` 注入

```
@Component
@ConfigurationProperties(prefix = "teacher")
@RefreshScope //如果使用反射模式，则不需要加这个注解
public class Person {

    private String name;

    private int age;

    String getName() {
        return name;
    }

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

    int getAge() {
        return age;
    }

    void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" + "name='" + name + '\'' + ", age=" + age + '}';
    }

}.
```

步骤3：TSF 控制台下发动态配置

用户可以通过 TSF 控制台来下发动态配置。

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。参见 日志配置项 。

操作步骤

关于如何通过控制台创建及下发更新的配置，请参见 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
```

Spring Cloud（原生依赖）
原生应用概述

最近更新时间：2022-06-09 16:49:26

TSF 支持原生 Spring Cloud 应用无侵入接入，无需改造即可直接接入 TSF，享受服务注册与发现、服务治理、应用监控和调用链跟踪等功能。

优势说明

Spring Cloud 原生应用迁移至 TSF 具备如下优势：

- 无需修改应用代码
- 无需引入额外 SDK（除使用调用链功能外）
- 无需进行额外配置

实现原理

应用到注册中心的所有请求如注册、发现等，都会被代理到我们自己的注册中心从而完成服务的注册和发现。当前的服务治理功能是基于 TSF Mesh 来实现的，服务调用会经过 TSF Mesh 的 sidecar，在 sidecar 中实现负载均衡、服务路由等治理功能。

- 用户在使用 TSF 的服务限流/熔断规则功能前，如果已通过 Hystrix/Sentinel 等组件配置了服务限流熔断规则，可能会与此处新建的规则产生冲突，请确保只开启了一种。如果确定要启用 TSF 的服务治理功能，需要关闭已经自建的规则。请参见 关闭服务熔断和限流规则 。
- 使用原生应用时，如果服务已经在使用调用链后端（如 Skywalking、Jaeger 等），用户可以选择继续使用自己搭建的调用链后端，也可以改用 TSF 提供的调用链功能。请参见 使用调用链功能 。

功能说明

原生应用目前仅支持 Spring Cloud 应用和 Eureka/Consul 两种注册中心和 Nacos 注册配置中心，后续可能支持更多语言和框架，以及更多注册中心。
原生应用不支持全局命名空间特性。全局命名空间说明请参见 命名空间管理 。

Spring Cloud 功能	开源实现	原生应用	说明
注册发现	Netflix Eureka Consul Discovery	兼容	-
负载均衡	Netflix Ribbon Spring Cloud loadbalancer	兼容	-
服务调用	Feign RestTemplate	兼容	自定义标签，需在 HTTP 请求头添加 tsf-mesh-tag: KEY=VALUE
服务限流	-	支持	自定义标签，需在 HTTP 请求头添加 tsf-mesh-tag: KEY=VALUE
服务熔断	hystrix	支持	-
服务鉴权	-	支持	自定义标签，需在 HTTP 请求头添加 tsf-mesh-tag: KEY=VALUE
配置管理	Config Server Consul Config	不支持	-
链路追踪	Spring Cloud Sleuth zipkin	兼容	-
微服务网关	Spring Cloud Gateway Netflix Zuul	兼容	-

Demo 工程概述

最近更新时间：2024-09-04 14:29:31

准备工作

开发前，请确保：

- 已参见 下载 Maven 安装 Java 和 Maven。
- 已安装 Docker（容器部署场景）。

下载 Demo

下载 Demo 地址： spring cloud demo 。其中 README.md 中介绍了打包 jar 包和 Docker 镜像的方法。

Demo 目录结构

```
.
├── pom.xml            # parent pom.xml
├── consul-consumer    # 使用 consul 的 consumer
├── consul-provider    # 使用 consul 的 provider
├── eureka-consumer    # 使用 eureka 的 consumer
├── eureka-provider    # 使用 eureka 的 provider
├── eureka-server      # eureka server
├── gateway            # 使用 Spring Cloud Gateway 的网关应用
└── zuul1              # 使用 Zuul1 的网关应用
```

从 pom.xml 可以看出，只使用了开源的原生组件。

调用说明

Demo 提供的服务和监听端口为：

- consul-consumer:8001
- consul-provider:8002
- eureka-consumer:8003
- eureka-provider:8004

可以访问 consumer 来调用 provider，例如：

```
curl consul-consumer:8001/ping-provider # 会调用 consul-provider:8002/ping

curl eureka-consumer:8003/ping-provider # 会调用 eureka-provider:8004/ping
```

关闭服务熔断和限流规则

最近更新时间：2022-06-09 16:51:14

操作场景

用户在 TSF 控制台新建服务限流/熔断规则时，如果已通过 Hystrix/Sentinel 等组件配置了服务限流熔断规则，可能会与此处新建的规则产生冲突。如果确定要启用 TSF 的服务治理功能，需要关闭已经自建的规则。

> ⚠ 注意
> 某些配置可能会关闭其他功能，请注意使用，做好验证。

操作步骤

限流

组件	关闭方法
Resilience	不支持通过 property 配置，需自行修改代码来关闭。
Sentinel	不支持通过 property 配置，需自行修改代码来关闭。

熔断

Hystrix/Spring Cloud Hystrix

可以通过 property hystrix.command.default.circuitBreaker.enabled 关闭，修改 application.yml：

```
hystrix:
 command:
   default:
     circuitBreaker:
       enabled: false
```

如果用了 Feign，此方式也可以关闭其中的熔断功能。

Resilience

不支持配置，只能通过 transitionToDisabledState() 或自行修改代码来关闭。
`transitionToDisabledState` 示例如下：

```
public class ProviderServiceResilience {
    private final static String cbName = "default";

    private final CircuitBreakerRegistry cbRegistry;

    public ProviderServiceResilience() {
        cbRegistry = CircuitBreakerRegistry.ofDefaults();
    }

    public String run() {
        try {
            cbRegistry.circuitBreaker(cbName).executeCallable(...);
        } catch (Exception e) {
        }
        return "resilience fallback\n";
    }
```

```
    public void disable() {
        cbRegistry.circuitBreaker(cbName).transitionToDisabledState();
    }
}
```

Spring Cloud Circuit Breaker – Resilience

> ⚠ **注意**
> 此方法会关闭其他 Resilience 功能，如 TimeLimiter，请谨慎使用。

可以通过 property `spring.cloud.circuitbreaker.resilience4j.enabled` 关闭，修改 application.yml：

```
spring:
  cloud:
    circuitbreaker:
      resilience4j:
        enabled: false
```

Sentinel

需要自行修改代码来关闭。

Spring Cloud Circuit Breaker – Sentinel

可以通过 property `spring.cloud.circuitbreaker.sentinel.enabled` 关闭，修改 application.yml：

```
spring:
  cloud:
    circuitbreaker:
      sentinel:
        enabled: false
```

使用调用链

最近更新时间：2022-06-09 16:52:06

操作场景

使用原生应用时，如果服务已经在使用调用链后端（如 Skywalking、Jaeger），用户可以选择继续使用自己搭建的调用链后端，也可以改用 TSF 提供的调用链功能。

操作步骤

使用自己搭建的调用链后端

只需要保证 TSF 部署的应用到调用链后端的网络是通的，然后正常部署原生应用即可。目前已测试通过 Skywalking、Zipkin、Jaeger、Pinpoint。
要使用 Skywalking/Pinpoint 这类需要 Java Agent 的后端，如果在虚拟机部署，需要先自行把相关文件安装在虚拟机上，然后做好路径配置。如果部署容器，也是类似。

使用 TSF 的调用链功能

需要在应用中传递调用链信息，目前支持 Zipkin 的 B3 Propagation，所以只要是和 Zipkin B3 兼容的 SDK 均可，例如 Spring Cloud Sleuth。
使用 Spring Cloud Sleuth 来传递调用链信息
在 pom.xml 添加 `spring-cloud-starter-sleuth` 依赖：

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

通过 Feign/RestTemplate 等调用服务时，会传递如下 HTTP Header：

```
X-B3-SpanId: 381cea277131b627
X-B3-ParentSpanId: 7fba7505d61e4db2
X-B3-Sampled: 0
X-B3-TraceId: 7fba7505d61e4db2
```

容器托管应用

最近更新时间：2023-09-27 15:23:24

操作场景

本文以一个 consumer 应用和一个 provider 应用为例介绍通过容器部署方式将 Spring Cloud 原生应用部署到 TSF 的操作方法。

前提条件

1. 已参见 下载 Maven 安装 Java 和 Maven。
2. 下载 TSF 提供的 Demo (参见 Demo 工程概述)。
3. 解压 Demo 压缩包，按 README.md 文件说明执行命令 `make docker-build` 制作容器镜像。
4. 在 TSF 控制台上已创建容器集群并导入云主机，详情参见 集群管理 。

操作步骤

步骤1：创建并部署原生应用

1. 创建应用

1. 登录 TSF 控制台 。
2. 在左侧导航栏，单击**应用管理**，进入应用列表页。
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。
   - **部署方式**：选择 **容器部署**。
   - **业务类型**：选择 **业务应用**。
   - **开发语言**：选择 **JAVA**。
   - **开发框架**：选择 **SpringCloud**。
   - **应用类型**：选择 **原生应用**。
4. 单击**提交**，完成应用创建。

2. 将镜像推送到仓库

1. 在左侧导航栏，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. 在左侧导航栏，单击**应用管理** > **ID/应用名** > **镜像**，单击**使用指引**，根据指引中的命令将 Demo 应用的镜像（参见 前提条件 中的第2步）推送到镜像仓库中（详情操作参见 镜像管理 ）。

3. 部署应用

1. 在应用列表中，单击在 步骤1：新建应用 中创建的应用的 “ID”。
2. 在部署组页面，单击**新建部署组**，设置部署组相关信息。
   - **组名**：填写部署组名。
   - **集群**：选择提前创建好的集群。
   - **命名空间**：选择集群关联的默认命名空间。
   - **日志配置项**：用于采集应用的业务日志数据，此处可选择**无**。
   - **日志投递**：用于日志转储，此处可选择**无**。
3. 单击**保存&下一步**，进入部署应用页面。
4. 设置部署相关信息。
   - **选择镜像**：选择 步骤2：将镜像推送到仓库 中推送到镜像仓库的镜像版本。
   - **启动参数**（选填）：设置 Java 应用的启动参数。
   - **资源配置**：应用容器的 CPU 和内存限制使用默认值即可，实例数设置为1。

- 访问配置
  - 网络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。
  - 端口映射中选择 TCP 协议，容器端口和服务端口设置为8002。

5. 单击**提交**，完成应用部署，部署应用后部署组状态变为运行中。

步骤2：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，选择正确的地域和命名空间，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，查看具体信息。

步骤3：验证服务调用

使用与之前相同的流程部署一组 consumer 和 provider（如 consul-consumer 和 consul-provider）。

1. 请求 consumer 来调用 provider

有3种方式可以从公网请求 consumer：

- **云主机 IP + NodePort**：如果部署组在部署时，网络访问方式选择了**主机端口访问**，可以通过**云主机 IP + NodePort**来访问 `consumer` 服务的 `/ping-provider` 接口。其中 `云主机 IP` 为集群中任一云主机的 IP，`NodePort` 可以在部署组的基本信息页面被查看。

```
curl <云主机 IP>:<NodePort>/ping-provider
```

- **负载均衡 IP + 服务端口**：如果部署组在部署时，网络访问方式选择了**提供公网访问方式**，可以通过**负载均衡 IP + 服务端口**来访问 `consumer` 服务的 `/ping-provider` 接口。
- **API 网关**：用户可以通过在 API 网关配置微服务 API 来调用 user 服务的接口。关于如何配置微服务 API 网关，可参见文档 API 网关作为请求入口。

也可以在容器内通过服务名和服务端口请求 consumer。先通过 kubectl 等方式进入 容器，然后调用：

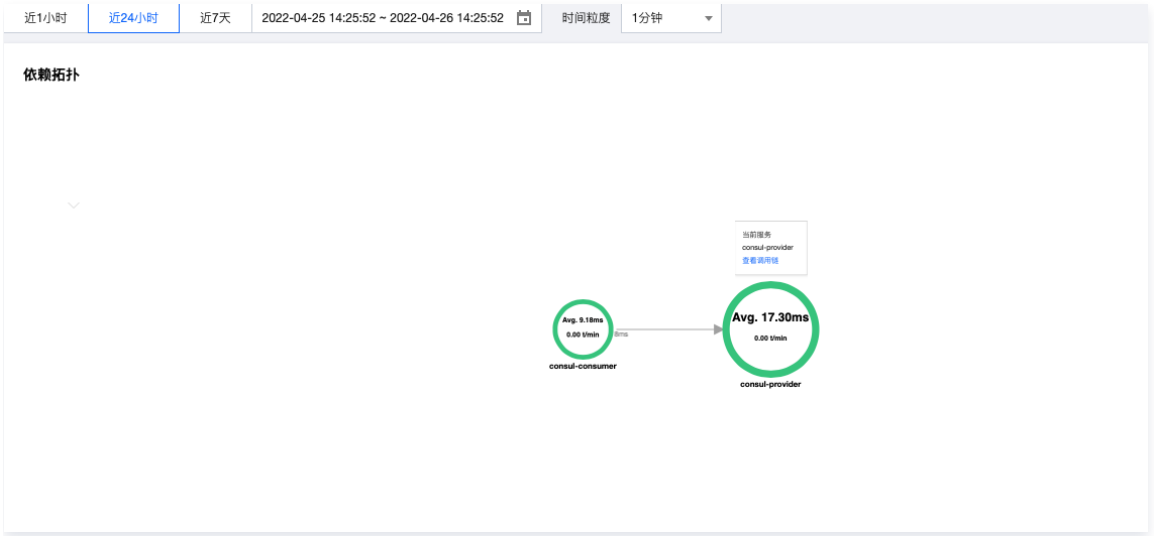
```
wget -O- consul-consumer:8001/ping-provider
wget -O- eureka-consumer:8003/ping-provider
```

2. 在控制台验证服务之间是否调用

可以验证服务是否成功被注册，同时验证服务之间是否成功地进行了调用。

在**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 provider 被 consumer 请求成功。

在**依赖拓扑**界面：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。



虚拟机托管应用

最近更新时间：2024-01-19 10:39:01

操作场景

本文以一个 consumer 应用和一个 provider 应用为例介绍通过虚拟机部署方式将 Spring Cloud 原生应用部署到 TSF 的操作方法。

前提条件

1. 已参见 下载 Maven 安装 Java 和 Maven。
2. 下载 TSF 提供的 Demo (参见 Demo 工程概述)。
3. 解压 Demo 压缩包，按 README.md 文件说明执行命令 `make build` 制作应用程序包。
4. 在 TSF 控制台上已创建虚拟机集群并导入云主机，详情参见 集群管理 。

操作步骤

步骤1：创建并部署原生应用

1. 创建应用

1. 登录 TSF 控制台 。
2. 在左侧导航栏单击**应用管理**，进入应用管理列表页。
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**。
   - **业务类型**：选择 **业务应用**。
   - **开发语言**：选择 **JAVA**。
   - **开发框架**：选择 **SpringCloud**。
   - **应用类型**：选择 **原生应用**。
4. 单击**提交**，完成应用创建，在弹出的窗口中选择**确认**，前往上传程序包并部署应用。

2. 上传程序包

1. 上传程序包页面，单击**上传程序包**，选择程序包（如 `consul-provider/target/consul-provider-*.jar` ），填写程序包相关信息。
2. 单击**提交**，完成上传。

3. 创建部署组

1. 在应用列表中，单击在 步骤1：新建应用 中创建的应用的 “ID”。
2. 在部署组页面，单击**新建部署组**，设置部署组相关信息。
   - **组名**：部署组的名称，不超过60个字符。
   - **集群**：选择提前创建好的集群。
   - **命名空间**：选择集群关联的默认命名空间。
   - **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。此处可选择**无**。
   - **日志投递**：用于日志转储，此处可选择**无**。
3. 单击**保存&下一步**，从关联集群的可用云主机列表勾选用于部署的云主机。
4. 单击**部署应用**，设置部署信息。
   - 软件仓库：选择**默认仓库**。
   - 程序包类型：选择 **jar**。
   - JDK 版本：选择 JDK 版本。
   - 程序包类型：选择已上传的程序包。

- 启动参数：选填。
- 更新方式：选择立即更新。
- 健康检查：可选。详情参见 健康检查 。
- 描述：可选。

5. 单击**完成**，部署应用后部署组状态变为运行中。

步骤2：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，查看具体信息。

步骤3：验证服务调用

使用与之前相同的流程部署一组 consumer 和 provider（如 consul-consumer 和 consul-provider）。

1. 登录服务器验证服务间调用

为了验证 consumer 服务能通过服务名来调用 provider 服务，需要用户通过以下方式来请求 provider 服务的调用（以 consul 为例）：

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

```
curl localhost:8001/ping-provider
```

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

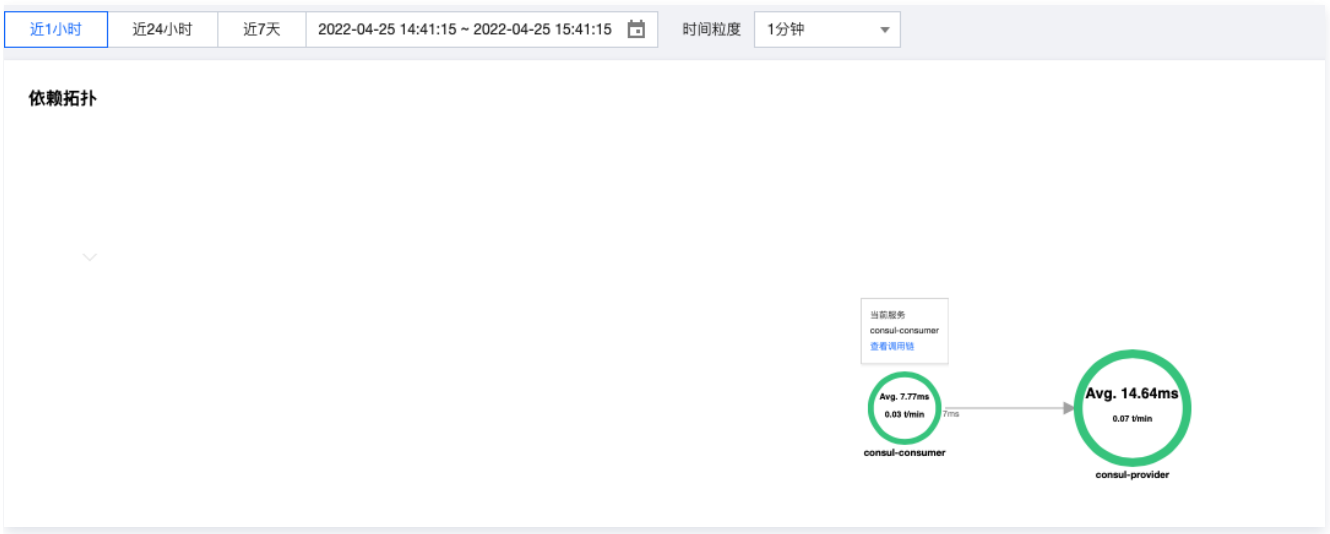
```
curl consul-provider:8002/ping
```

- **API 网关**：用户可以通过在 API 网关配置微服务 API 来调用 consumer 服务的接口。关于如何配置微服务 API 网关，请参见文档 API 网关作为请求入口 。

2. 在控制台验证服务之间是否调用

可以验证服务是否成功被注册，同时验证服务之间是否成功地进行了调用。
在**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 provider 被 consumer 请求成功。

在**依赖拓扑**界面：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。


近1小时
近24小时
近7天
2022-04-25 14:41:15 ~ 2022-04-25 15:41:15
时间粒度
1分钟
依赖拓扑
当前服务
consul-consumer
查看调用链
Avg. 7.77ms
0.03 t/min
7ms
consul-consumer
Avg. 14.64ms
0.07 t/min
consul-provider

Dubbo 应用接入
Demo 工程概述

最近更新时间：2023-07-26 14:51:01

获取 Demo

Demo 工程下载地址：Demo 下载。

> **说明**
> - 本 Demo 工程仅是示例，现存的 Dubbo 应用接入 TSF 可以直接参见 Dubbo 应用接入TSF 。
> - 如果从零开始新开发的微服务系统推荐采用 Spring Cloud 框架，详情参见 Spring Cloud TSF 应用开发 。

工程目录

`tsf-dubbo-xxx-demo` 的工程目录如下：

```
|- consumer-dubbo2
|- provider-dubbo2
|- middleman-dubbo2
|- pom.xml
```

其中 consumer 表示服务消费者， middleman 和 provider 都表示服务提供者，pom.xml 中定义了工程需要的依赖包。

Dubbo 应用接入 TSF

最近更新时间：2024-01-19 10:55:51

通过 Dubbo 的扩展机制，将 TSF 的大部分能力适配至 Dubbo 上，允许用户只修改几行依赖和配置即可体验完整的服务治理和监控体验。

> **说明**
> Apache Dubbo 接入 TSF 需要平台组件支持，私有云 TSF 支持的版本请提交工单咨询。

功能支持

- 服务注册与发现： 支持 Dubbo 的接口级注册发现，同时扩展了应用级注册发现。
  - 应用级的服务名为 dubbo.application.name。但如果与 Spring Cloud 结合使用，并且与 spring.application.name 相同，为了区分服务，Dubbo 应用级服务名自动加下 `-dubbo` 后缀。当程序有暴露 Dubbo 接口时，才会进行应用级的服务注册。
  - 接口级的服务名为： `providers:<接口限定名>:<Version信息>:<Group信息>` 。
- 配置管理：如需使用，建议结合 spring-cloud-tsf-starter 一起使用。如果是纯 Dubbo2 应用，仅支持通过内部接口获取配置及监听。
- 服务治理：支持服务鉴权、服务路由、服务限流、服务熔断。
  - TSF 服务熔断需要配置在主调方，由于仅 dubbo client 时可以不注册服务。可以结合 spring-cloud-tsf-starter 使用并配置在 Spring Cloud 对应的服务上。或者在控制台上手动创建 dubbo.application.name 对应的服务，并在上面配置熔断规则。
  - Dubbo 第一跳没有对应的服务，因此第一跳时，治理规则自定义标签中无法使用上游服务名进行限制。第二跳时，上游服务名为上一跳的目标 Dubbo 服务。
  - Dubbo 治理规则系统标签不支持 API 以及 HTTP method。
  - Dubbo 默认优先选择应用级服务发现，此时服务鉴权、服务路由、服务限流规则需要配置在应用级对应的服务上。
- 服务容错：不单独支持 Dubbo 服务容错，如果需要使用可结合 spring-cloud-tsf-starter。
- 参数传递
  - 与 Spring Cloud 结合使用时，可以参见 Spring Cloud TSF 参数传递的用法 。
  - Dubbo 单独使用时，可使用 `com.tencent.tsf.util.TsfTagUtils` 的对应方法。
- 全链路灰度发布：支持全链路中包含 Dubbo 服务也有 Spring Cloud 服务（仅支持 Spring Cloud 2020 及以上的版本）。
- 调用链、服务监控：通过可观测 Agent（opentelemetry）支持大部分功能，但调用链中的 API 信息不支持跳转，不支持 API 级别的监控信息。
- API 注册：Dubbo 不支持。
- 与 Spring Cloud 框架结合：仅支持 Spring Cloud 2020 及以上的版本一起使用 。

接入操作步骤

步骤1：安装 Maven 环境

参见 下载Maven指引 下载安装 Java 和 Maven。

步骤2：配置注册中心

- 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**）： 下载地址
  XML 格式：

```
<dubbo:registry address="tsf://127.0.0.1:8500">
</dubbo:registry>
```

  YAML 格式：

```
dubbo:
  registry:
    address: tsf://127.0.0.1:8500
```

- 注册中心协议名为 tsf。

> **说明**
> 新接入客户推荐使用注册中心协议 tsf，接口级注册时，服务名附带完整的信息。

- "注册中心地址:端口" 可以填写 127.0.0.1:8500，在 TSF 控制台部署时，SDK 会替换为正确的地址。
- 默认的 register-mode 为 all，即双注册（既包括应用级，也包括接口级）。
  xml 格式：

```
<dubbo:application name="provider-demo-dubbo" logger="slf4j" register-mode="instance"/>
```

yaml 格式：

```
dubbo:
  application:
    name: provider-demo-dubbo
    logger: slf4j
    register-mode: instance
```

步骤3：添加依赖

根据 Dubbo 单独使用还是和 Spring Cloud 一起使用，所用的依赖及其版本有一定区别。

- 单独使用 Dubbo（可能与 Spring Boot 一起使用），在 pom .xml 文件中增加插件依赖：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>femas-adaptor-tsf-apache-dubbo2</artifactId>
    <version>1.0.9-RELEASE</version>
</dependency>
```

- Dubbo 与 Spring Cloud 一起使用（Spring Cloud 只支持 2020 及以上版本）
  - Spring Cloud 2021 版本：
    - 父依赖：父依赖使用 spring-cloud-tsf-dependencies。

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-dependencies</artifactId>
    <version>1.40.4-SpringCloud2021-RELEASE</version>
</parent>
```

    - 如果必须要用业务的父依赖，则可以在子项目的 dependencyManagement 里添加。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.tencent.tsf</groupId>
      <artifactId>spring-cloud-tsf-dependencies</artifactId>
      <version>1.40.4-SpringCloud2021-RELEASE</version>
```

```
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- tsf dubbo 依赖：不需要指定版本号，如果手动指定了，可能与 spring cloud tsf 的相关依赖产生冲突，造成异常。

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>femas-adaptor-tsf-apache-dubbo2</artifactId>
</dependency>
```

- Spring Cloud 2020 版本：
  - 父依赖：父依赖使用 spring-cloud-tsf-dependencies。

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-dependencies</artifactId>
    <version>1.40.7-SpringCloud2020-RELEASE</version>
</parent>
```

  - 如果必须要用业务的父依赖，则可以在子项目的 dependencyManagement 里添加。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.tencent.tsf</groupId>
      <artifactId>spring-cloud-tsf-dependencies</artifactId>
      <version>1.40.7-SpringCloud2020-RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

  - tsf dubbo 依赖：不需要指定版本号，如果手动指定了，可能与 spring cloud tsf 的相关依赖产生冲突，造成异常。

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>femas-adaptor-tsf-apache-dubbo2</artifactId>
</dependency>
```

步骤4：配置日志格式

Spring Cloud 2020（Spring Boot 2.4）开始，默认的日志格式有所变化。如果需要用日志配置项中的 Spring Boot 格式采集日志，需要对 pattern 进行以下设置：

```
logging:
  pattern:
    level: "%-5level [${spring.application.name},%mdc{trace_id},%mdc{span_id},]"
```

步骤5：打包 FatJar

和 Spring Boot 或 Spring Cloud 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），在 pom.xml 所在目录下执行命令 `mvn clean package` 。

> **说明**
> 如果是单纯的 Dubbo 应用，可以使用 Maven 的 FatJar 插件。

步骤6：使用可观测 Agent

可观测 Agent 在云上部署时才需要使用。本地使用也无法采集数据。
使用方法：下载 ot-agent-release.tar，并解压。容器部署时需要手动将 Agent 物料放到镜像内。启动 jar 时附带 -javaagent:ot-agent-release/opentelemetry-javaagent.jar -Dotel.traces.exporter=none -Dotel.javaagent.extensions=ot-agent-release/femas-trace-opentelemetry.jar`，通过 main 方法直接启动。其中 ot-agent-release/opentelemetry-javaagent.jar 和 ot-agent-release/femas-trace-opentelemetry.jar 是对应的具体路径。例如：

```
java ${JAVA_OPTS} -Xshare:off -javaagent:/opt/ot-agent-release/opentelemetry-javaagent.jar -
Dotel.traces.exporter=none -Dotel.javaagent.extensions=/opt/ot-agent-release/femas-trace-
opentelemetry.jar -jar provider-demo.jar
```

如果使用的是 jdk17，还需要额外加以下参数

```
--add-opens java.base/java.lang=ALL-UNNAMED --add-opens java.base/java.lang.reflect=ALL-UNNAMED --add-opens
java.base/java.math=ALL-UNNAMED --add-opens java.base/sun.net.util=ALL-UNNAMED
```

```
java ${JAVA_OPTS} -Xshare:off -javaagent:/opt/ot-agent-release/opentelemetry-javaagent.jar -
Dotel.traces.exporter=none -Dotel.javaagent.extensions=/opt/ot-agent-release/femas-trace-
opentelemetry.jar --add-opens java.base/java.lang=ALL-UNNAMED --add-opens
java.base/java.lang.reflect=ALL-UNNAMED --add-opens java.base/java.math=ALL-UNNAMED --add-opens
java.base/sun.net.util=ALL-UNNAMED -jar provider-demo.jar
```

Dubbo reference 配置建议

Dubbo 客户端需要使用 reference 对象调用 Dubbo 服务端，reference 的 check 属性默认为 true。即启动时会检查服务端的实例状态，如果不存在在线实例，则会启动失败。建议设置为 false，避免对服务启动顺序的依赖。timeout 属性默认为 1000（单位ms），初始化建立连接，可观测 Agent 对相关 class 的增强，都是在第一次调用的时候进行。如果业务程序也有一些初始化而导致慢些，较容易超时，建议根据实际情况调整，建议调整在 5000ms 以上。

- xml 格式示例：

```
<dubbo:reference id="echoDubboService" interface="com.tsf.demo.dubbo.EchoDubboService"
check="false"  timeout="5000"/>
```

- Java 注解示例：

```
@DubboReference(timeout = 5000, check = false)
EchoDubboService echoDubboService;
```

Dubbo 应用接入 Mesh

最近更新时间：2022-06-10 11:45:21

操作场景

Dubbo 作为一款传统基于 SDK 的 Java RPC 框架，通过引入 jar 包的方式实现服务间远程方法调用、服务注册发现及服务治理。Dubbo 开发的应用在不修改任何业务代码的前提下，可以通过简单的 Mesh 封包部署到 TSF 平台，可透明实现服务注册发现和无侵入的服务治理能力。
本文档以 Dubbo Mesh Demo 为例介绍 Dubbo 应用在 TSF 平台接入 Mesh 的操作方案及相关注意事项。

前提条件

已下载 Dubbo Mesh Demo 。

操作步骤

步骤1：安装 Maven 环境

参见 下载 Maven 下载安装 Java 和 Maven。

步骤2：配置 spec.yaml 服务注册文件

Dubbo Mesh Demo 中提供了三个 Dubbo 应用，以 greet 应用为例，其 spec.yaml 配置为：

```
apiVersion: v1
kind: Application
spec:
  services:
    - name: org.apache.dubbo.samples.api.GreetingService # 需要注册的 Dubbo 服务名，必须跟提供的
interface 名保持一致
      ports:
        - targetPort: 20881 # 该 Dubbo 服务监听的端口
          protocol: dubbo # 协议指定为 Dubbo
      healthCheck:
        path:
```

> **说明**
> 部署到 TSF 平台后，Mesh 的 sidecar 解析 Dubbo 应用的 spec.yaml，根据服务名、监听端口、协议等信息自动注册服务到注册中心。

屏蔽 greet 应用原注册中心（可选）：

```
<dubbo:registry address="N/A" subscribe="false" register="false"/>
```

> **说明**
> 此处也可保留原注册中心，实现 Dubbo 服务双注册。

步骤3：编译打包

Dubbo Mesh Demo 中每个应用都提供了 `build.sh` 脚本用于构建部署在 TSF 上的 Mesh 程序包，执行 `./build.sh` 将在当前目录下生成对应的 `tar.gz` 包，该包可直接部署在 TSF 上，以 greet 应用为例，生成的程序包为 `dubbo-greet.tar.gz` ，文件目录包括：

- dubbo-samples-greetservice-1.0.jar: Maven 构建的 FATJAR 包
- start.sh
- stop.sh
- cmdline

- spec.yaml

`build.sh` 脚本及程序包中文件详细内容请 下载 Dubbo Mesh Demo 查看。

Dubbo 兼容说明

- TSF Mesh 为 Dubbo 应用提供服务间通信、服务注册发现、负载均衡、服务路由、服务熔断、服务鉴权和服务限流等治理能力。
- Dubbo 应用可以保留原有的注册方式。
- Dubbo 应用可以继续使用原有的 filter 机制。

Dubbo3 接入 TSF

最近更新时间：2024-01-19 10:55:51

通过 Dubbo 的扩展机制，将 TSF 的大部分能力适配至 Dubbo 上，允许用户只修改几行依赖和配置即可体验完整的服务治理和监控体验。

> **说明**
> Apache Dubbo3 接入 TSF 需要平台组件支持，私有云 TSF 支持的版本请提交工单咨询。

功能支持

- 服务注册与发现： 支持 Dubbo3 的接口级注册发现和应用级注册发现
  - 应用级的服务名为 dubbo.application.name。但如果与 Spring Cloud 结合使用，并且与 spring.application.name 相同，为了区分服务，Dubbo 应用级服务名自动加下 `-dubbo` 后缀。Dubbo3 机制下，当程序有暴露 Dubbo 接口时，才会进行应用级的服务注册。
  - 接口级的服务名为 `providers:<接口全限定名>:<Version信息>:<Group信息>` 。
- 配置管理：如需使用，建议结合 spring-cloud-tsf-starter 一起使用。如果是纯 Dubbo3 应用，仅支持通过内部接口获取配置及监听。
- 服务治理：支持服务鉴权、服务路由、服务限流、服务熔断。
  - TSF 服务熔断需要配置在主调方，由于仅 dubbo client 时可以不注册服务。可以结合 spring-cloud-tsf-starter 使用并配置在 Spring Cloud 对应的服务上。或者在控制台上手动创建 dubbo.application.name 对应的服务，并在上面配置熔断规则。
  - Dubbo 第一跳没有对应的服务，因此第一跳时，治理规则自定义标签中无法使用上游服务名进行限制。第二跳时，上游服务名为上一跳的目标 Dubbo 服务。
  - Dubbo3 治理规则系统标签不支持 API 以及 HTTP method。
  - Dubbo3 默认优先选择应用级服务发现，此时服务鉴权、服务路由、服务限流规则需要配置在应用级对应的服务上。
- 服务容错：不单独支持 Dubbo3 服务容错，如果需要使用可结合 spring-cloud-tsf-starter。
- 参数传递
  - 与 Spring Cloud 结合使用时，可以参见 Spring Cloud TSF 参数传递的用法 。
  - Dubbo3 单独使用时，可使用 `com.tencent.tsf.util.TsfTagUtils` 的对应方法。
- 全链路灰度发布：支持。并且支持全链路中包含 Dubbo3 服务也有 Spring Cloud 服务（仅支持 Spring Cloud 2020 及以上的版本）。
- 调用链、服务监控：通过可观测 Agent（opentelemetry）支持大部分功能，但调用链中的 API 信息不支持跳转，不支持 API 级别的监控信息。
- API 注册：Dubbo3 不支持。
- 与 Spring Cloud 框架结合：仅支持 Spring Cloud 2020 及以上的版本一起使用 。

接入操作步骤

步骤1：安装 Maven 环境

参见 下载 Maven 下载安装 Java 和 Maven。

步骤2：配置注册中心

- 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**）： 下载地址->>
  xml 格式：

```
<dubbo:registry address="tsf://127.0.0.1:8500">
</dubbo:registry>
```

  yaml 格式：

```
dubbo:
  registry:
    address: tsf://127.0.0.1:8500
```

- 注册中心协议名为 tsf。

> **说明**
> 新接入客户推荐使用注册中心协议 tsf，接口级注册时，服务名附带完整的信息。存量使用 TSF Atom Dubbo 接入的客户，如果使用的协议是 tsfconsul，升级到 TSF Dubbo3 时也需要保持，否则服务发现不兼容。

- "注册中心地址:端口" 可以填写 127.0.0.1:8500，在 TSF 控制台部署时，SDK 会替换为正确的地址。
- 默认的 register-mode 为 all，即双注册（既包括应用级，也包括接口级），如果都是 Dubbo3 的应用，建议 register-mode 改为 instance，仅注册应用级服务。这样服务列表的服务数比较少，比较干净，也可以降低注册中心的压力。如果有存量的 TSF Atom Dubbo2.7 的，可以全部迁到 TSF Dubbo3 后建议将 register-mode 改为 instance。
  xml 格式：

```
<dubbo:application name="provider-demo-dubbo" logger="slf4j" register-mode="instance"/>
```

  yaml 格式：

```
dubbo:
  application:
    name: provider-demo-dubbo
    logger: slf4j
    register-mode: instance
```

步骤3：添加依赖

根据 Dubbo3 单独使用还是和 Spring Cloud 一起使用，所用的依赖及其版本有一定区别。

- 单独使用 Dubbo（可能与 Spring Boot 一起使用），在 pom .xml 文件中增加插件依赖：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>femas-adaptor-tsf-apache-dubbo3.1</artifactId>
      <!-- 调整为 tsf dubbo3 最新版本号，1.0.10-RELEASE 开始，将 femas-adaptor-tsf-apache-dubbo3 细分为为 femas-adaptor-tsf-apache-dubbo3.0 和 femas-adaptor-tsf-apache-dubbo3.1.1  -->
    <version>1.0.10-RELEASE</version>
</dependency>
```

- Dubbo3 与 Spring Cloud 一起使用（Spring Cloud 只支持 2020 及以上版本）
  - Spring Cloud 2021 版本：
    - 父依赖：父依赖使用 spring-cloud-tsf-dependencies。

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-dependencies</artifactId>
    <version>1.40.4-SpringCloud2021-RELEASE</version>
</parent>
```

    - 如果必须要用业务的父依赖，则可以在子项目的 dependencyManagement 里添加。

```
<dependencyManagement>
```

```
  <dependencies>
    <dependency>
      <groupId>com.tencent.tsf</groupId>
      <artifactId>spring-cloud-tsf-dependencies</artifactId>
      <version>1.40.4-SpringCloud2021-RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- tsf dubbo 依赖：不需要指定版本号，如果手动指定了，可能与 spring cloud tsf 的相关依赖产生冲突，造成异常。

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>femas-adaptor-tsf-apache-dubbo3.1</artifactId>
</dependency>
```

- Spring Cloud 2020 版本：
  - 父依赖：父依赖使用 spring-cloud-tsf-dependencies。

```
<parent>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-dependencies</artifactId>
    <version>1.40.7-SpringCloud2020-RELEASE</version>
</parent>
```

  - 如果必须要用业务的父依赖，则可以在子项目的 dependencyManagement 里添加。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.tencent.tsf</groupId>
      <artifactId>spring-cloud-tsf-dependencies</artifactId>
      <version>1.40.7-SpringCloud2020-RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

  - tsf dubbo 依赖：不需要指定版本号，如果手动指定了，可能与 spring cloud tsf 的相关依赖产生冲突，造成异常。

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>femas-adaptor-tsf-apache-dubbo3.1</artifactId>
</dependency>
```

步骤4：配置日志格式

Spring Cloud 2020（Spring Boot 2.4）开始，默认的日志格式有所变化。如果需要用日志配置项中的 Spring Boot 格式采集日志，需要对 pattern 进行以下设置：

```
logging:
```

```
  pattern:
    level: "%-5level [${spring.application.name},%mdc{trace_id},%mdc{span_id},]"
```

步骤5：打包 FatJar

和 Spring Boot 或 Spring Cloud 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），在 pom.xml 所在目录下执行命令 `mvn clean package` 。

> **说明**
> 如果是单纯的 Dubbo 应用，可以使用 Maven 的 FatJar 插件。

步骤：使用可观测 Agent

可观测 Agent 在云上部署时才需要使用。本地使用也无法采集数据。
使用方法：下载 ot-agent-release.tar，并解压。容器部署时需要手动将 Agent 物料放到镜像内。启动 jar 时附带 -javaagent:ot-agent-release/opentelemetry-javaagent.jar -Dotel.traces.exporter=none -Dotel.javaagent.extensions=ot-agent-release/femas-trace-opentelemetry.jar`，通过 main 方法直接启动。其中 ot-agent-release/opentelemetry-javaagent.jar 和 ot-agent-release/femas-trace-opentelemetry.jar 是对应的具体路径。例如：

```
java ${JAVA_OPTS} -Xshare:off -javaagent:/opt/ot-agent-release/opentelemetry-javaagent.jar -
Dotel.traces.exporter=none -Dotel.javaagent.extensions=/opt/ot-agent-release/femas-trace-
opentelemetry.jar -jar provider-demo.jar
```

如果使用的是 jdk17，还需要额外加以下参数

```
--add-opens java.base/java.lang=ALL-UNNAMED --add-opens java.base/java.lang.reflect=ALL-UNNAMED --add-opens
java.base/java.math=ALL-UNNAMED --add-opens java.base/sun.net.util=ALL-UNNAMED
```

```
java ${JAVA_OPTS} -Xshare:off -javaagent:/opt/ot-agent-release/opentelemetry-javaagent.jar -
Dotel.traces.exporter=none -Dotel.javaagent.extensions=/opt/ot-agent-release/femas-trace-
opentelemetry.jar --add-opens java.base/java.lang=ALL-UNNAMED --add-opens
java.base/java.lang.reflect=ALL-UNNAMED --add-opens java.base/java.math=ALL-UNNAMED --add-opens
java.base/sun.net.util=ALL-UNNAMED -jar provider-demo.jar
```

与 TSF Atom Dubbo 2.7 的兼容性

- 适用于 consumer 为 TSF Atom Dubbo 2.7 、provider 为 TSF Dubbo3 或 consumer 为 TSF Dubbo3、provider 为 TSF Atom Dubbo 2.7 的场景。
- 注册中心协议需要保持一致，否则服务发现会失败。
- 混用场景仅兼容服务注册与发现。治理能力需要 consumer 和 provider 都升级到 TSF Dubbo3 才能完整支持。

Dubbo reference 配置建议

Dubbo 客户端需要使用 reference 对象调用 Dubbo 服务端，reference 的 check 属性默认为 true。即启动时会检查服务端的实例状态，如果不存在在线实例，则会启动失败。建议设置为 false，避免对服务启动顺序的依赖。timeout 属性默认为 1000（单位ms），初始化建立连接，可观测 Agent 对相关 class 的增强，都是在第一次调用的时候进行。如果业务程序也有一些初始化而导致慢些，较容易超时，建议根据实际情况调整，最好 5000ms 以上。

- xml 格式示例：

```
<dubbo:reference id="echoDubboService" interface="com.tsf.demo.dubbo.EchoDubboService"
check="false"  timeout="5000"/>
```

- Java 注解示例：

```
@DubboReference(timeout = 5000, check = false)
EchoDubboService echoDubboService;
```

Mesh 应用开发
TSF Mesh 概述

最近更新时间：2023-06-05 17:50:01

Mesh 微服务平台（Tencent Service Mesh Framework，以下简称 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 集成在腾讯微服务平台 TSF 中，支持跨语言微服务与 Spring Cloud、Dubbo 框架微服务互通，支持对等的服务治理能力。
以下视频将为您介绍 TSF Mesh 的基本架构和优势：
观看视频

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，代码无需改造。

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。

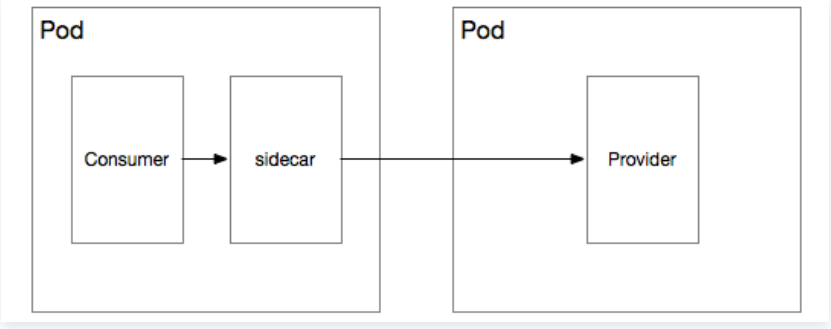
使用场景

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

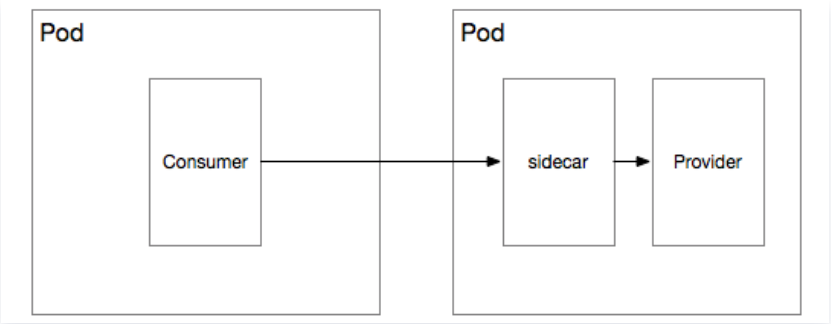
- 服务提供者使用 TSF-Spring Cloud 框架实现，注册到服务注册中心；
- 服务消费者作为 Mesh 应用部署，由 Sidecar 注册到服务注册中心。



场景2：仅服务提供者作为 Mesh 应用部署

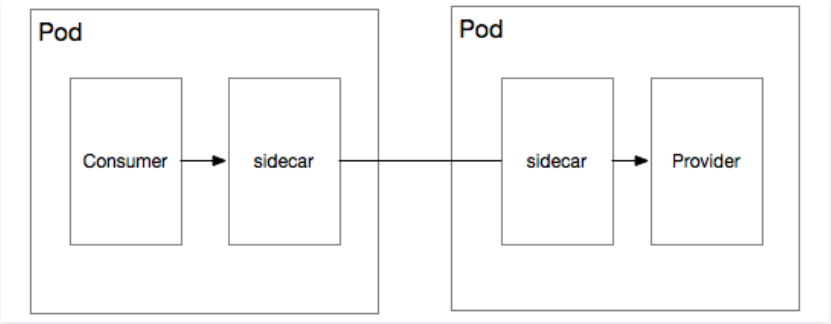
- 服务提供者作为 Mesh 应用部署，由 Sidecar 注册到服务注册中心；

- 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。



Demo 工程概述

最近更新时间：2024-01-19 10:55:51

下载 Demo

Demo 语言	下载地址	说明
Python Demo（vm）	tsf_python_vm_demo	-
Python Demo（docker）	tsf_python_docker_demo	-
.NET Demo（vm & docker）	tsf_mesh_demo_dotnet	其中 REAME.md 介绍了程序包和镜像两种构建方式
Java Demo（vm）	tsf_mesh_vm_java_demo	-
Java Demo (docker)	tsf_mesh_docker_java_demo	-
PHP Demo（vm）	tsf_php_vm_demo	其中 README.md 介绍了健康检查的端口和路径
Dubbo Demo（vm）	tsf_mesh_dubbo_vm_demo	-

注意

Mesh 的部署和使用与语言无关，具体可参见 Python 使用方式进行改造。

调用说明

下文以 Python Demo 为例进行介绍。Python Demo 提供了3个应用，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`。

注意

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))
```

Spring Cloud 应用和 Mesh 应用调用打通 tracing

Spring Cloud 应用和 Mesh 应用相互调用时，如果要打通 tracing，需要在请求中传递调用链相关的 header（参见 上文）。

Demo 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，具体解释请参见 Mesh 开发使用指引。
- **apis 目录**：存放 API 定义的目录，具体解释请参见 Mesh 开发使用指引。

其中 start.sh、stop.sh、cmdline 的编写方法请参见 程序包格式说明。

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。

> **注意**
> 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。**

Mesh 应用开发

最近更新时间：2025-05-26 10:45:51

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，如果不使用服务名调用，流量不会经过 sidecar 直接传递到被调服务，被调服务无法识别主调服务的服务名。
- 端口使用80或者业务真实的监听端口。
- 其他代码不做修改。

> **说明：**
> 以下代码片段可参见 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。

> **说明：**
> 当前支持在控制台上选择**使用本地 spec.yaml**和**控制台配置**两种方式描述服务信息，推荐在控制台上直接设置。具体操作参见 Mesh 应用部署（容器篇） 或者 Mesh 应用部署（虚拟机篇）。

若选择使用本地 spec.yaml 文件，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 # 目前支持 HTTP、HTTP2 和 gRPC
    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，否则，健康检查失败会导致此服务实例变为离线状态，其它服务将无法调用该服务实例；如果不提供此健康检查接口，sidecar 会通过 TCP 的方式探测 targetPort 是否连通来判断此服务实例是否健康。
- `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。

服务间调用方式

使用服务名调用

在 user 服务所在实例上执行 curl 命令，通过使用 `shop` 服务名进行访问。

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

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，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: Payment Required
        '403':
           description: Forbidden
  /api/v6/user/account/query:
```

```
    get:
      responses:
        '200':
           description: OK
        '401':
           description: Unauthorized
        '402':
           description: Payment Required
        '403':
           description: Forbidden
  /health:
    get:
      responses:
        '200':
           description: OK
        '401':
           description: Unauthorized
        '402':
           description: Payment Required
        '403':
           description: Forbidden
```

设置自定义标签（可选）

Mesh 支持通过 HTTP Header 设置自定义标签，以 Python 应用为例说明如何设置自定义标签。

- **对于调用链和全链路灰度发布中的自定义标签。**

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

- **对于服务鉴权、服务路由和服务限流中的自定义标签。**

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

> **说明：**
> 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']
```

具体使用方法参见 Mesh Demo 介绍 。

全链路灰度发布 Header 传递（可选）

要实现全链路灰度发布功能，还需要在请求中额外带上 `tsf-system-tags` 的 header，传递方式和**调用链 Header 传递**类似。

Mesh 运维接口说明

最近更新时间：2022-06-09 17:08:21

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，代理侧提供了大量的 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。**

容器托管应用

最近更新时间：2023-09-27 15:23:24

操作场景

本文以容器环境为例介绍通过 TSF 控制台创建并部署 Mesh 应用，查看服务是否注册成功并验证服务调用的操作方法。

准备工作

1. 下载 TSF 提供的 Python Mesh Demo（docker）。
2. 解压 Demo 压缩包，分别进入二级目录（如 `demo-mesh-promotion` 目录），执行 `docker build` 命令制作容器镜像。
3. 在 TSF 控制台上已创建容器集群并导入云主机，详情参见 集群管理。
4. 用户的开发机上已安装 `docker` 环境（用于推送镜像到镜像仓库）。

操作步骤

步骤1：创建并部署 Mesh 应用

1. 创建应用

1. 登录 TSF 控制台。
2. 在左侧导航栏，单击**应用管理**，进入应用列表页。
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**。
   - **业务类型**：选择 **业务应用**。
   - **开发语言**：选择 **其他语言**。
   - **开发框架**：选择 **其他框架**。
   - **应用类型**：选择 **Mesh应用**。
   - **服务配置**：可以选择**使用本地 Spec.yaml**（默认方式）或者**控制台配置**。
     若选择**控制台配置**，需要填写以下信息：

   - 服务名：最长60个字符，只能包含字母、数字和分隔符（“-”），且不能以分隔符开头或结尾。
   - 监听端口：只有1个（接口使用数组类型）。协议：HTTP/HTTP2/gRPC/Dubbo；端口范围 1~65535。
   - 心跳检查接口：校验逻辑，以 / 开头。不超过200个字符。

> **注意**
> 控制台配置时，应用镜像的 Dockerfile 运行命令前需加上这样一条命令 `touch /opt/tsf/app_config/running` 以告知 Sidecar 容器应用容器已启动，例如 Demo 中 Dockerfile 的启动命令可以这样写：

```
ENTRYPOINT ["bash", "-c", "touch /opt/tsf/app_config/running &&
/root/app/shopService/start.sh"]
```

- 标签：选择已有标签或者前往标签管理创建。

4. 单击**提交**，完成应用创建。

2. 将镜像推送到仓库

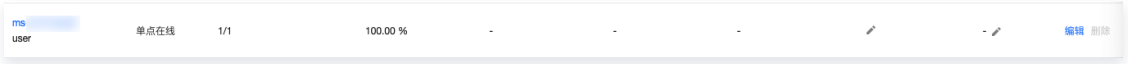
1. 在左侧导航栏，单击**镜像仓库**，进入镜像列表页。首次使用时，您需要设置镜像仓库密码（该密码与腾讯云官网账号密码独立）。
2. 在镜像列表页，单击**应用管理** > **ID/应用名** > **镜像**，单击**使用指引**，根据指引中的命令将 Python demo 应用的镜像（参见 准备工作 中的第2步）推送到镜像仓库中（详情参见 镜像管理 ）。

3. 部署应用

1. 在应用列表中，单击在 步骤1：创建应用 中创建的应用的 “ID”。
2. 在部署组页面，单击**新建部署组**，设置部署组相关信息。
   - **组名**：填写 部署组信息。
   - **集群**：选择提前创建好的集群。
   - **命名空间**：选择集群关联的默认命名空间。
   - **日志配置项**：用于采集应用的业务日志数据，此处可选择**无**。
   - **日志投递**：用于日志转储，此处可选择**无**。
3. 单击**保存&下一步**，进入部署应用页面。
4. 设置部署相关信息。
   - **选择镜像**：选择 步骤2：将镜像推送到仓库 中推送到镜像仓库的镜像版本。
   - **启动参数**（选填）：设置 Java 应用的启动参数。
   - 环境变量（选填）：此处可不填写。
   - **资源配置**：应用容器的 CPU 和内存限制使用默认值即可，实例数设置为1。
   - 访问配置：
     - 网络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。
     - 端口映射中选择 TCP 协议，容器端口和服务端口填写相同的数值（user：8089，shop：8090，promotion：8091）
5. 单击**提交**，完成应用部署，部署应用后部署组状态变为运行中。

步骤2：查看服务是否注册成功

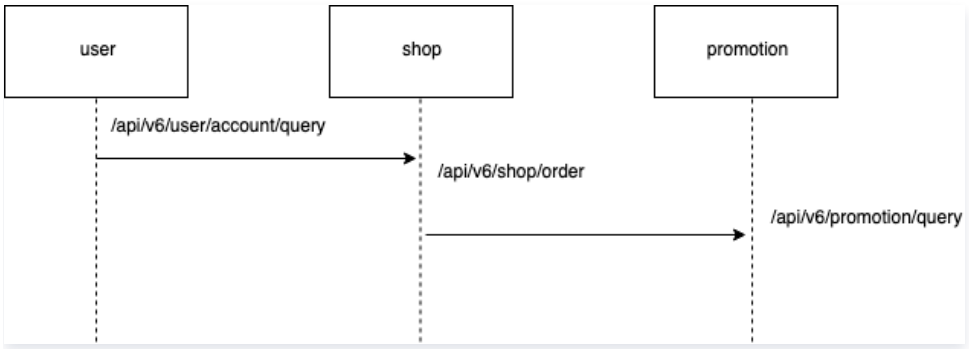
1. 在左侧导航栏，单击 **服务治理**，进入服务列表页，查看服务是否注册成功。如果注册成功，服务显示在线状态。



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。

步骤3：验证服务调用

使用同样的步骤部署 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，网络访问方式选择了**提供公网访问方式**，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。
- **云主机 IP + NodePort**：如果部署组在部署时，网络访问方式选择了**主机端口访问**，可以通过**云主机 IP + NodePort**来访问 `user` 服务的 `/api/v6/user/account/query` 接口。其中 **云主机** `IP` 为集群中任一云主机的内网 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所在 VPC 的机器，然后执行如下命令：

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

- **API 网关**：用户可以通过在 API 网关配置微服务 API 来调用 user 服务的接口。关于如何配置微服务 API 网关，可参见文档 API 网关作为请求入口 。

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。**

常见问题

- 应用部署成功后，服务列表中为何没有出现服务?
- 服务实例显示离线状态如何解决?
- 如何联系后台运维人员?

虚拟机托管应用

最近更新时间：2023-09-27 15:23:24

操作场景

本文以虚拟机应用为例，指导您通过 TSF 控制台，创建并部署 Mesh 应用、查看服务是否注册成功、验证服务调用。

准备工作

1. 下载 TSF 提供的 Python Mesh Demo（vm）。
2. 解压 Demo 压缩包，解压出三个压缩包分别为 `promotionService.tar.gz`、`shopService.tar.gz`、`userService.tar.gz`（不需要再解压）。
3. 在 TSF 控制台上已创虚拟机集群并导入云主机，参见 集群管理。
4. 主机上已安装应用运行的环境（如 Python 应用的相关依赖等，TSF 对相关依赖的版本没有限制）。

操作步骤

步骤1：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 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 **虚拟机部署**。
- 业务类型：选择 **业务应用**。
- 开发语言：选择 **其他语言**。
- 开发框架：选择 **其他框架**。
- 应用类型：选择 **Mesh 应用**。
- 服务配置：可以选择**使用本地Spec.yaml**（默认方式）或者**控制台配置**。
  若选择**控制台配置**，需要填写以下信息：

  - 服务名：最长60个字符，只能包含字母、数字和分隔符（“-”），且不能以分隔符开头或结尾。
  - 监听端口：只有1个（接口使用数组类型）。协议：HTTP/HTTP2/gRPC/Dubbo；端口范围 1~65535。
  - 心跳检查接口：校验逻辑，以 / 开头。不超过200个字符。
- 标签：选择已有标签或者前往标签管理创建。

1.4 单击**提交**，完成应用创建。

2. 上传程序包

1.1 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。

1.2 在应用服务详情页单击**程序包管理**标签页，单击**上传程序包**，选择程序包（如 `promotionService.tar.gz` ），填写程序包相关信息。

1.3 单击**提交**，完成上传。

3. 部署应用

1.1 在应用列表中，单击在 步骤1：新建应用 中创建的应用的 “ID”。

1.2 在新建部署组页面，填写部署组信息。

- **组名**：部署组的名称，不超过60个字符。
- **集群**：选择提前创建好的集群。
- **命名空间**：选择集群关联的默认命名空间。
- **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。此处可选择**无**。参见 日志服务 。
- **日志投递**：用于日志转储，此处可选择**无**。关于日志投递的详情说明可参见 日志投递 。
- **标签**：用于分类管理资源，可不选。详情参见 标签 。
- **备注**：选填，可留空。

1.3 单击**保存&下一步**，从关联集群的可用云主机列表勾选用于部署的云主机。

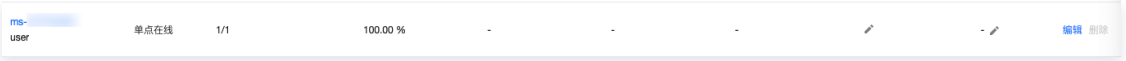
1.4 单击**部署应用**，设置部署信息。

- 软件仓库：选择**默认仓库**。
- 程序包类型：选择 **zip/tar.gz**。
- JDK 版本：选择 JDK 版本。
- 程序包类型：选择程序包名称为 `userService.tar.gz` 的程序包。
- 启动参数：选填。
- 更新方式：选择立即更新。
- 健康检查：可选。详情参见 健康检查 。
- 描述：可选。

1.5 单击**完成**，应用部署成功后，部署组中**已启动/总机器数**的数值发生变化。

步骤2：查看服务是否注册成功

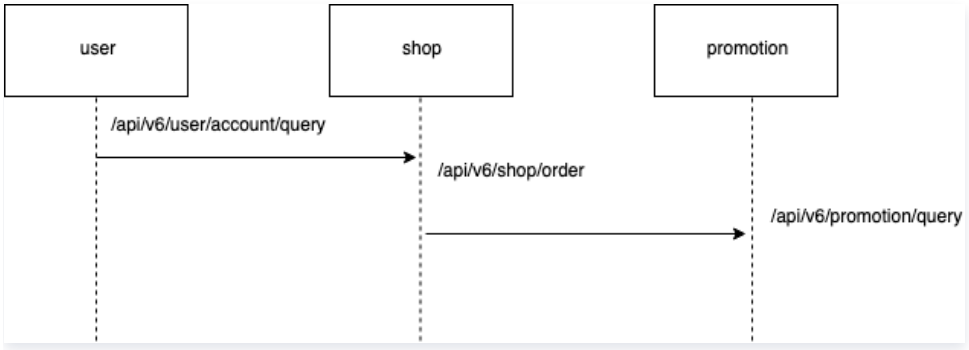
1.1 在左侧导航栏单击 服务治理，进入服务列表页，查看服务是否注册成功。如果成功，服务显示在线状态。



1.2 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。

步骤3：验证服务调用

使用同样的步骤部署 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：



用户可以登录虚拟机集群 VPC 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）。

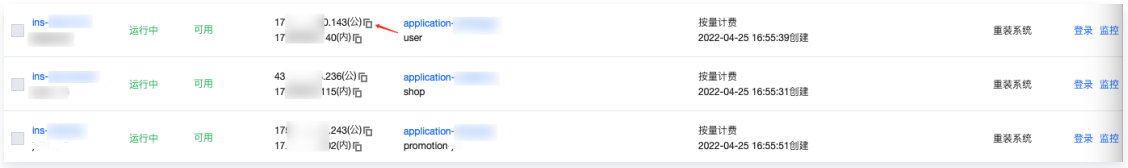
```
curl shop:<shop端口>/api/v6/shop/order
```

- **API 网关**：用户可以通过在 API 网关配置微服务 API 来调用 user 服务的接口。关于如何配置微服务 API 网关，请参见文档 API 网关作为请求入口。

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。

常见问题

- 应用部署成功后，服务列表中为何没有出现服务？
- 服务实例显示离线状态如何解决？
- 如何联系后台运维人员？

查看 SideCar 监控信息

最近更新时间：2022-06-09 17:11:48

操作场景

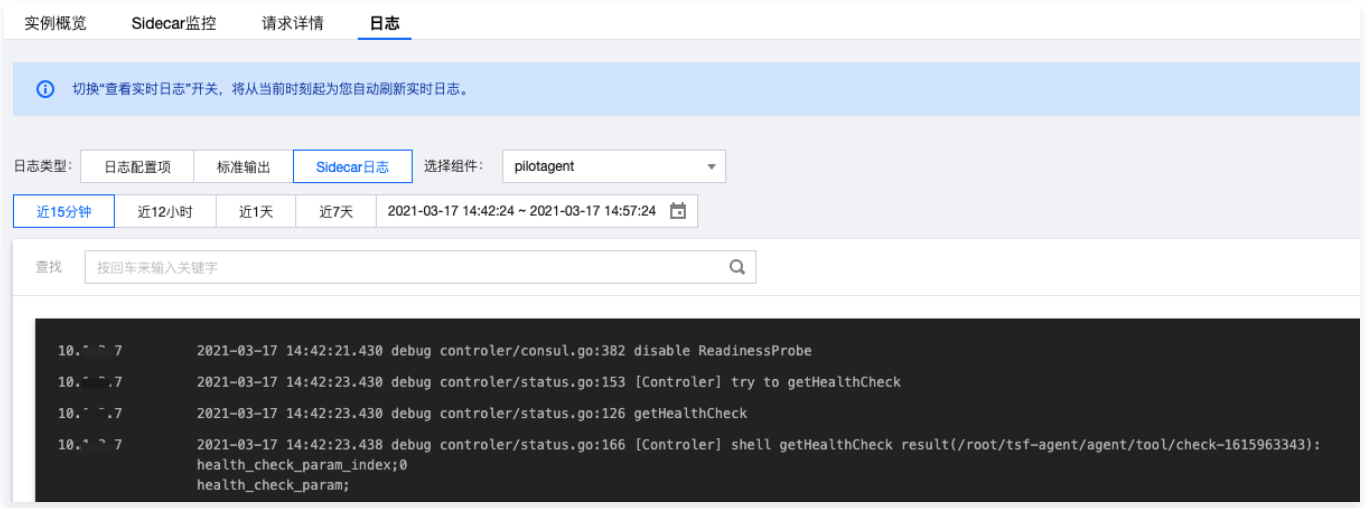
该任务指导您通过 TSF 控制台查看 SideCar 监控信息，包含 SideCar 运行状态、SideCar 监控数据和 SideCar 日志。

> **说明**
> 以下信息仅针对容器和虚拟机部署的 Mesh 应用生效。

操作步骤

1. 登录 TSF 控制台 。
2. 在左侧导航栏选择**部署组**，单击"部署组 ID "进入部署组详情，默认在**服务实例列表**页，可查看 SideCar 状态。

3. 选择您感兴趣的实例，将鼠标放置在 SideCar 状态上，单击**查看日志**，跳转至 SideCar 日志 tab 页。
4. 在 SideCar 日志页面，选择日志类型、组件和时间，可查看日志信息。



5. 选择**SideCar监控**，查询 SideCar 的版本与组件状态和基本信息。

配置 Sidecar 过滤器

最近更新时间：2024-01-29 14:39:41

操作场景

Service Mesh 应用支持在 Sidecar 上运行用户自定义的 Lua 脚本的能力，用户可以自定义一些业务逻辑放在 Sidecar 上执行，来实现一些自定义的业务逻辑（例如在请求头中添加参数、设置服务路由规则等）。配置 Sidecar 过滤器可以大大加强 Mesh 应用的灵活性。

操作步骤

新建过滤器

1. 登录 TSF 控制台 。
2. 在左侧导航栏选择**应用管理**，单击目标应用的 ID，进入应用详情页。
3. 访问 Sidecar过滤器页面，新建过滤器。
   - 填写过滤器名称以及备注。
   - 填写过滤器作用位置。
     - 作为服务端是指：该脚本将在部署组运行启动的微服务作为服务端时生效。
     - 作为客户端是指：该脚本将在部署组运行启动的微服务作为客户端的时候生效。当选择过滤器位置为客户端时，需要填写被调用的微服务的名称。仅当调用被调服务的时候，过滤器会生效。
   - 填写脚本内容。脚本需要严格按照 Lua 脚本的格式进行书写。

> **说明**
> - Lua 脚本最大为65535字节。
> - 已经下线的过滤器才能删除。
> - 编辑后的 Lua 脚本，需要再次发布之后才能生效。
>
> 以下为微服务作为客户端时，向请求头中添加两个字段并返回请求体大小的示例：

```
function envoy_on_request(request_handle)
    request_handle:headers():add("foo", "bar")
end
function envoy_on_response(response_handle)
    body_size = response_handle:body():length()
    response_handle:headers():add("response-body-size", tostring(body_size))
end
```

> **说明**
> 以下为微服务作为服务端时，设置请求响应延迟的示例，该脚本从 HTTP 请求头 header 中读取 x-delay-percent 和 x-delay-duration 字段，然后根据这些字段的值决定是否注入延迟。如果随机数小于 x-delay-percent，则注入延迟（单位：毫秒）。

```
function envoy_on_request(request_handle)
    --自定义部分，定义延迟概率和延迟时间的请求头
    local delay_percent = tonumber(request_handle:headers():get("x-delay-percent"))
    local delay_duration = tonumber(request_handle:headers():get("x-delay-duration"))

    if delay_percent and delay_duration then
        local random_value = math.random(100)
        if random_value < delay_percent then
            request_handle:logInfo("Injecting delay for " .. delay_duration .. " ms")
```

```
            os.execute("sleep " .. tostring(delay_duration / 1000))
        end
    end
end

function envoy_on_response(response_handle)
    --自定义部分
end
```

有关 Lua 脚本更多的书写原则，可以参见 Envoy 使用文档 。

4. 单击**提交**，完成创建。

发布过滤器

创建好的过滤器需要发布才能生效。当前 TSF 支持将同一个过滤器发布到多个部署组上，或将某一个过滤器下线。过滤器也可以发布在离线状态的部署组上，当部署组启动时会将过滤器加载。

在Sidecar过滤器列表页面，单击操作列的**发布**，勾选要发布的部署组，单击**确认**，完成发布。

更新服务配置

最近更新时间：2022-06-09 17:12:47

操作场景

当 Mesh 应用的服务配置选择**控制台配置**方式时，可以在应用基本信息页中更新服务配置；如果是使用 spec.yaml，则不显示服务配置信息。该任务指导您在 TSF 控制台更新 Mesh 应用服务配置。

操作步骤

1. 登录 TSF 控制台 。
2. 在左侧导航栏选择**应用管理**。
3. 选择目标应用，单击应用的“ID/应用名”，进入应用**基本信息**页。
4. 在**服务配置**模块，选择**编辑**，修改服务配置信息。

5. 单击**提交**后，在弹出框中选择**确认**去重启部署组。

> **说明**
> 更新服务配置后，需要重启部署组才会生效。

6. 在部署组列表页，在部署组状态栏单击**重启**弹出重启操作对话框。

> **说明**

- 更新服务配置后，对于**运行中**状态的部署组，如果部署组的服务配置和应用的服务配置信息不一致可重启部署组进行更新。
- 当 Mesh 应用（且使用虚拟机部署）的部署组内存在实例的 agent 版本较低时，不支持**服务配置**特性，请升级 agent 版本，参见 应用管理相关 。

7. 在重启应用对话框中，单击**提交**后，可看到部署组的运行状态变为正常。

微服务网关
微服务网关开发

最近更新时间：2024-11-05 16:13:52

操作场景

微服务网关作为后台架构的入口，提供路由转发、API 管理、访问过滤器等作用，是微服务架构中的重要组件。TSF 中的微服务网关基于 Spring Cloud 中的 Zuul 和 Spring Cloud Gateway 实现，提供了符合微服务体系的灵活可自定义的网关功能。

前提条件

- 在开始开发前，请确保您已经参见 下载 Maven 下载安装了 Java 和 Maven，并且配置了 TSF 私服地址。

> **说明：**
> - 从1.22.0版本开始，TSF 微服务网关 SDK（TSF-MSGW）提供基于 Zuul 和 SCG (Spring Cloud Gateway) 两种类型的实现。
> - 从1.22.0以前版本进行升级的用户，需要注意 POM 依赖发生了调整。

- 已参见 微服务网关 文档熟悉了微服务网关的功能。

Zuul

快速上手

使用微服务网关功能前，您需要在 `pom.xml` 中添加网关依赖项，同时在代码中使用网关开关注解。您也可以参见官方 TSF Demo 中 **msgw-demo** 模块来开发。

1.22.0-`Series`-RELEASE 以及之后版本

1. 向工程中添加依赖，在 `pom.xml` 文件中添加以下代码。

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-msgw-zuul</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

2. 向 Application 类中添加注解 `@EnableZuulProxy` 和 `SpringBootApplication`：

```
// 下面省略了无关的代码
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
@EnableZuulProxy
@SpringBootApplication
public class Application {
        public static void main(String[] args) {
                SpringApplication.run(Application.class, args);
        }
}
```

3. 创建分组，导入 API。具体请参见 微服务网关分组管理 。

1.22.0-Edgware-RELEASE/1.22.0-Finchley-RELEASE 之前的版本

1. 向工程中添加依赖，在 `pom.xml` 文件中添加以下代码。

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-gateway</artifactId>
    <version><!-- 1.22.0 之前版本 --></version>
</dependency>
```

2. 向 Application 类中添加注解 `@EnableTsfGateway` 。

```
// 下面省略了无关的代码
import com.tencent.tsf.gateway.core.annotation.EnableTsfGateway;
@EnableTsfGateway
public class Application {
        public static void main(String[] args) {
                SpringApplication.run(Application.class, args);
        }
}
```

3. 创建分组，导入 API。具体请参见 微服务网关分组管理 。

集成 TSF 其它功能

您在使用微服务网关功能的同时，还可以集成 TSF 其它功能，包括分布式配置、服务监控、服务治理等，您需要在 `pom.xml` 中添加其对应依赖项，同时在代码中启用注解，具体参见其 功能接入文档 。以下示例为集成所有功能：

1. 向工程中增加依赖。在 `pom.xml` 中**按顺序**添加以下代码：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-msgw-zuul</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
<!--TSF 其它 SDK 依赖，添加到 msgw-zuul 依赖的后面-->
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-starter</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

2. 向 Application 类中增加注解 `@EnableTsf` ：

```
// 下面省略了无关的代码
import com.tencent.tsf.gateway.core.annotation.EnableTsfGateway;
import org.springframework.tsf.annotation.EnableTsf;
@EnableZuulProxy
@SpringBootApplication
@EnableTsf
public class Application {
        public static void main(String[] args) {
                SpringApplication.run(Application.class, args);
        }
}
```

自定义网关过滤器

msgw-zuul SDK 提供通过继承 `TsfGatewayZuulFilter`（或原生 `ZuulFilter`）的方式自定义 Filter，同时需要在类上添加 `@TsfGatewayFilter` 注解（或其它生成Bean方式）。目前 msgw-zuul 基于 zuul1 实现，其 Filter 功能和原生 `ZuulFilter` 保持一致。

```
import static org.springframework.cloud.netflix.zuul.filters.support.FilterConstants.PRE_TYPE;

import com.netflix.zuul.exception.ZuulException;
import com.tencent.tsf.gateway.core.annotation.TsfGatewayFilter;
import com.tencent.tsf.gateway.zuul1.filter.TsfGatewayZuulFilter;

@TsfGatewayFilter
public class TestFilter extends TsfGatewayZuulFilter {

    @Override
    public String filterType() {
        return PRE_TYPE;
    }

    @Override
    public int filterOrder() {
        return 100;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() throws ZuulException {
        System.out.println("hello world");
        return null;
    }
}
```

SCG（Spring Cloud Gateway）

快速上手

使用微服务网关功能前，您需要在 `pom.xml` 中添加网关依赖项，同时在代码中使用网关开关注解。或参见官方 TSF Demo 中 msgw-demo 模块来编写。

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-msgw-scg</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
```

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

```
// 下面省略了无关的代码
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
```

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

3. 创建分组，导入 API。具体请参见 微服务网关分组管理 。

集成 TSF 其它功能

您在使用微服务网关功能的同时，还可以集成 TSF 其它功能，包括分布式配置、监控、服务治理等，您需要在 `pom.xml` 中添加其对应依赖项，同时在代码中启用注解，具体参见其 功能接入文档 。以下示例为集成所有功能：

1. 向工程中增加依赖（scg 在1.26.0才支持服务治理功能，1.26.0以前的 SDK 版本添加 spring-cloud-tsf-starter 会报错）。在 `pom.xml` 中按顺序添加以下代码：

```
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-msgw-scg</artifactId>
    <version><!-- 调整为 SDK 最新版本号 --></version>
</dependency>
<!--TSF 其它 SDK 依赖，添加到 msgw-scg 依赖的后面-->
<dependency>
    <groupId>com.tencent.tsf</groupId>
    <artifactId>spring-cloud-tsf-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
        <exclusion>
            <groupId>com.tencent.tsf</groupId>
            <artifactId>spring-cloud-tsf-swagger</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

2. 向 Application 类中增加注解 `@EnableTsf`：

```
// 下面省略了无关的代码
import org.springframework.tsf.annotation.EnableTsf;

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

自定义网关过滤器

msgw-scg SDK 提供通过继承 `AbstractTsfGlobalFilter` 的方式，或通过实现原生 `GlobalFilter` 接口的方式自定义 Filter。同时需要在类上添加 `@TsfGatewayFilter` 注解（或其它生成 Bean 方式）。`AbstractTsfGlobalFilter` 提供了和编写 ZuulFilter 类似的体验：

```
// 下面省略了无关的代码
@TsfGatewayFilter
public class TestFilter extends AbstractTsfGlobalFilter {
```

```
    @Override
    public int getOrder() {
        return 100;
    }

    @Override
    public boolean shouldFilter(ServerWebExchange exchange, GatewayFilterChain chain) {
        return true;
    }

    @Override
    public Mono<Void> doFilter(ServerWebExchange exchange, GatewayFilterChain chain) {
        System.out.println("hello world");
        return null;
    }
}
```

网关配置自定义路由模式

最近更新时间：2024-10-28 10:39:32

功能说明

- 1.46 版本 SDK 之前，TSF 微服务网关 SDK 只支持分组路由模式，即请求路径需要携带网关分组等信息，并且不支持自定义路由规则。如果客户原本使用的是开源 Zuul 或 SCG，迁移改造量大。
- 从 1.46 版本 SDK 开始，TSF 微服务网关 SDK 支持两种路由模式：分组路由模式和自定义路由模式。
  - **分组路由模式**：TSF 托管微服务API。该模式下，TSF 会校验网关分组、请求 URL 请求路径等是否合法。支持 TSF 服务治理、支持使用网关分组相关能力（网关插件、API 限流、API 超时等）。不支持自定义路由规则 。
  - **自定义路由模式**：TSF 不托管微服务 API。路由方式由配置文件定义。该模式下，TSF 不会校验网关分组和 URL 请求路径等是否合法。支持 TSF 服务治理，不支持网关分组的相关能力（包括网关插件、API 限流、API 超时等）

功能	分组路由模式	自定义路由模式
TSF 服务治理	支持	支持
网关分组的相关能力（包括网关插件、TSF API 限流、API 超时等）	支持	不支持
自定义路由规则	不支持	支持

前提条件

开始实践配置微服务 SDK 支持网关自定义路由功能之前，请确保您已完成 微服务网关开发 。

操作步骤

您可以通过以下两种方式配置网关自定义路由功能。

方式一：在微服务网关的请求 Header 头设置

在微服务网关的请求 Header 头里设置标识某个请求是自定义路由 API 的请求。

- 请求的 Header 头里设置 **TSF-Opensource-Mode: true**：表示本次网关请求是自定义路由 API 的请求，走自定义路由模式的逻辑，否则依然走分组路由模式（默认都是分组路由模式）。
- 请求的 Header 头里设置 **TSF-NamespaceId: namespace-xxxxx**：表示本次网关请求路由到命名空间 ID 为 namespace-xxxxx 的下游服务，如果不设置则默认请求到与网关相同命名空间的下游服务。

> **说明**
> - 只有全局命名空间的网关才可以路由到任意一个命名空间的服务，普通命名空间的网关只能路由到与网关相同普通命名空间的服务。
> - TSF-NamespaceId 填写的是命名空间 ID，而非命名空间名称。
> - 跨命名空间访问时，需要配置对应自定义路由。

方式二：在微服务网关应用的配置文件设置

- 请求的Header 头里设置 **TSF-NamespaceId: namespace-xxxxx**：表示本次网关请求路由到命名空间 ID 为 namespace-xxxxx 的下游服务，如果不设置则默认请求到与网关相同命名空间的下游服务。
- 在微服务网关应用的配置文件设置 **tsf.gateway.opensource-mode: true**。

```
tsf:
  gateway:
```

```
    opensource-mode: true
```

说明：

- TSF-NamespaceId 填写的是命名空间 ID，而非命名空间名称。
- 跨命名空间访问时，需要配置对应自定义路由。
- tsf.gateway.opensource-mode 默认为 false，表示默认是分组路由模式，如果设置为 true 表示开启自定义路由模式。
- 开启自定义路由模式以后，所有非 TSF 网关分组前缀的请求都会被当作是自定义路由 API 的请求，例如某个 TSF 网关分组名是 /test，那么只有 `http://ip:port/test/xxx` 的请求才会走原先分组路由模式的逻辑，其他请求（例如 `http://ip:port/abc/xxx` ）走自定义路由 API 请求的逻辑。

自定义路由规则配置

自定义路由模式下，默认请求路径为{Host:port}/{下游服务名}/{下游API路径}。例如： localhost:8080/provider-demo/echo/test，其中 provider-demo 为下游服务名，/echo/aaa为下游 API 路径。
如果需要额外的路由规则，或跨命名空间访问，可以通过以下配置进行设置：

1. SCG网关配置

配置名称	配置说明
spring.cloud.gateway.routes[].id	路由的 ID ，不允许重复
spring.cloud.gateway.routes[].uri	转发地址，可以是服务名，也可以是静态地址
spring.cloud.gateway.routes[].predicates	SCG 断言，用于匹配路由
spring.cloud.gateway.routes[].filters	SCG 过滤器

配置示例：

```
spring:
  cloud:
    gateway:
      routes:
        - id: provider-demo
          uri: lb://provider-demo        # 走服务发现，跨命名空间配置参考这里
          predicates:
            - Path=/provider-demo/**     # /provider-demo/ 前缀的请求转发走服务发现转发到 provider-demo
实例上
          filters:
            - StripPrefix=1              # 标识在转发之前应删除此路由的前缀
        - id: business-server
          uri: http://localhost:9999     # 静态地址，不走服务发现
          predicates:
            - Path=/business/**          # /business/ 前缀的请求直接转发到 http://localhost:9999
          filters:
            - StripPrefix=1
        - id: websocket
          uri: ws://127.0.0.1:1234       # websocket 转发，只支持配置静态地址
          predicates:
            - Path=/websocket
          filters:
```

```
            - StripPrefix=1
```

2. Zuul 网关配置

配置名称	配置说明
zuul.routes.{路由key}.id	{路由key} 为替换为实际路由名称。路由的 ID（默认情况下与其 map 的 key 相同）。不允许重复。
zuul.routes.{路由key}.path	{路由key} 为替换为实际路由名称。路由的 pattern（样式），例如 /foo/**。
zuul.routes.{路由key}.serviceId	{路由key} 为替换为实际路由名称。映射到此路由的服务 ID（如果有）。您可以指定物理 URL 或服务，但不能两者都指定。
zuul.routes.{路由key}.url	{路由key} 为替换为实际路由名称。完整的物理 URL 以映射到路线。一种替代方法是使用服务 ID 和服务发现来查找物理地址。
zuul.routes.{路由key}.stripPrefix	{路由key} 为替换为实际路由名称。默认为 true，用于标识是否在转发之前是否应删除此路由的前缀

配置示例：

```
zuul:
  routes:
    api-v1:
      path: /provider-demo/**          # /provider-demo/ 前缀的请求转发走服务发现转发到 provider-demo 实例上，跨命名空间配置参考这里
      serviceId: provider-demo         # 走服务发现
      stripPrefix: true
    api-v2:
      path: /v2/**
      url: http://127.0.0.1:18081
      stripPrefix: true
```

密钥对鉴权使用说明

最近更新时间：2022-06-10 09:47:33

HTTP 请求头

HTTP 请求头中各参数说明如下：

名称	位置	是否必选	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	授权的 SecretID，用于加签。开启密钥对鉴权时需要。
x-mg-alg	请求/响应	否	加密算法。开启密钥对鉴权时需要。 0：hmac_md5 1：hmac_sha_1 2：hmac_sha_256 3：hmac_sha_512
x-mg-sign	请求/响应	否	签名值。开启密钥对鉴权时需要。
x-mg-nonce	请求/响应	否	随机数。开启密钥对鉴权时需要。
x-mg-code	响应	是	响应码。

签名算法

算法列表

- Hmac_MD5
- HMAC_SHA_1
- HMAC_SHA_256
- HMAC_SHA_512

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/DHVXXXXgeROMrrQ==
secretKey：+t9tTMTXXXXUcE+RKOleg==
x-mg-noce：D7pAR5fqXXXXx1yacuVzdO
digetValue：D7pAR5fqXXXXx1yacuVzdOhKhATL/DHVXXXXgeROMrrQ==+t9tTMTXXXXXUcE+RKOleg==
signValue：FE6nLWwYBDXXXXU2CVtFndUBoyg=

校验规则

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

代码实现

Java 代码：

```
/**
```

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

    public static void main(String[] args) {
        String secretId = "hKhATL/DHVXXXXgeROMrrQ==";
        String secretKey = "+t9tTMTXXXXUcE+RKOleg==";
        String nonce = "D7pAR5fqXXXXx1yacuVzdO";
        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 和 SecetKey
"""
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','o
','p','q','r','s','t','u','v','w','x','y','z','A','B','C','D','E','F','G','H','I','J','K','L','M','
N','O','P','Q','R','S','T','U','V','W','X','Y','Z']
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
```

使用说明

生成签名

使用上面 Python 代码 生成签名，文件命名为：gen_sign.py。

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

{secretId} 替换成密钥 ID，{secretKey} 替换成密钥 KEY，示例如下：

```
python gen_sign.py hKhATL/DHVXXXXgeROMrrQ==  +t9tTMTXXXXUcE+RKOleg==
```

输出内容：

```
generate local sign: FE6nLWwYBDXXXXU2CVtFndUBoyg=
=== http request headers as followed ===
x-mg-nonce: D7pAR5fqXXXXx1yacuVzdO
x-mg-secretid: hKhATL/DHVXXXXgeROMrrQ==
x-mg-traceid: 8a237eba-71b2-11e9-acee-5254001d2da0
x-mg-alg: 1
x-mg-sign: FE6nLWwYBDXXXXU2CVtFndUBoyg=
```

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}
```

- 测试鉴权的 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**
> { } 中的参数请根据实际内容替换。

配置 HTTPS

最近更新时间：2022-06-10 09:47:51

操作场景

本文档指导您通过 TSF 的微服务网关配置 HTTPS 访问。

操作步骤

步骤1：准备 SSL 证书

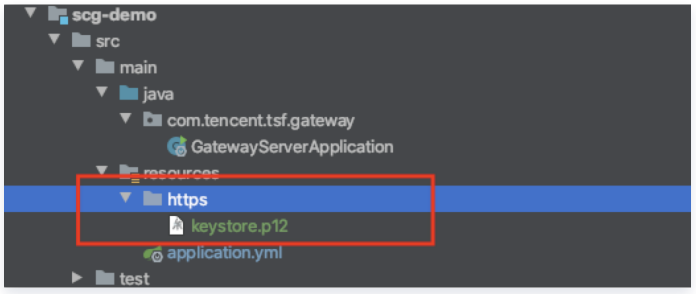
您可以通过以下任一种方式准备一个 SSL 证书：

- 方式一：购买（通过证书授权机构购买）
- 方式二：自己生成（通过 keytool 或 openssl 工具生成）

步骤2：配置证书信息

> **说明**
> 如果网关类型为 Envoy 网关，证书需要放在部署 Envoy 网关应用的时候配置的路径下。

1. 将 SSL 证书放到 resources下 的 https 文件夹（可自定义名称）中：



2. 在配置文件 application.yml 中设置证书的基本信息。如下：

```
# ssl证书相关配置

server:
  ssl:
    #启用ssl
    enabled: true
    #证书地址
    key-store: classpath:https/keystore.p12
    #证书密码
    key-store-password: 123456
    #证书类型
    key-store-type: JKS
```

> **说明**
> SpringCloudZuul 与 SpringCloudGateway 配置方式一致，是因为 SpringBoot 服务使用 SSL-HTTPS 的通用方式。

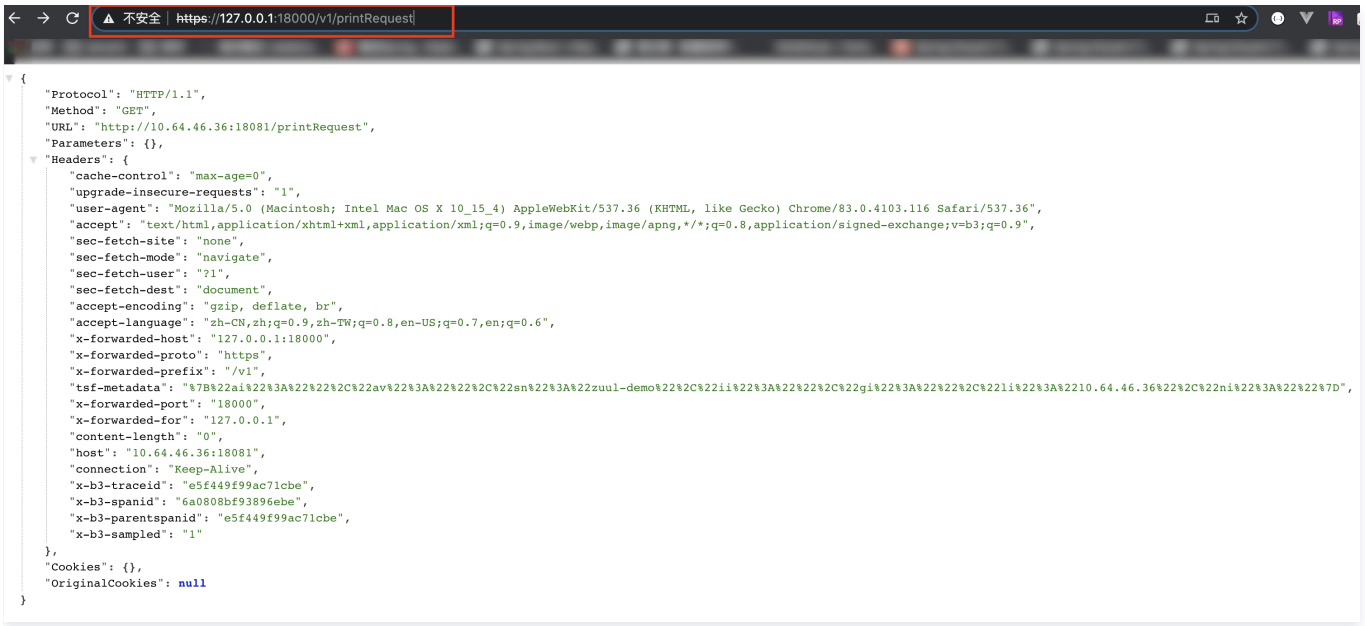
步骤3：访问验证

zuul application.yml 配置如下：

```
server:
```

```
  port: 18000
  ssl:
    enabled: true
    key-store: classpath:https/keystore.p12
    key-store-password: 123456
    key-alias: tomcathttps
    key-store-type: JKS
spring:
  application:
    name: zuul-demo
  cloud:
    consul:
      host: 127.0.0.1
      port: 8500
      discovery:
        heartbeat:
          enabled: true
          ttl-value: 5
          ttl-unit: s
        preferIpAddress: true
ribbon:
  ReadTimeout: 10000
  ConnectTimeout: 5000
zuul:
  max:
    host:
      connections: 500
  host:
    socket-timeout-millis: 10000
    connect-timeout-millis: 5000
  routes:
    api-v1:
      path: /v1/**
      serviceId: provider-demo
      stripPrefix: true
    api-v2:
      path: /v2/**
      url: http://127.0.0.1:18081
      stripPrefix: true
    api-v3:
      path: /*/v3/**
      serviceId: provider-demo
      stripPrefix: true
  ignored-patterns: /**/ignore/**
hystrix:
  command:
    default:
      execution:
        timeout:
          enabled: true
        isolation:
          thread:
            timeoutInMilliseconds: 30000
```

HTTPS 访问网关成功的显示如下：

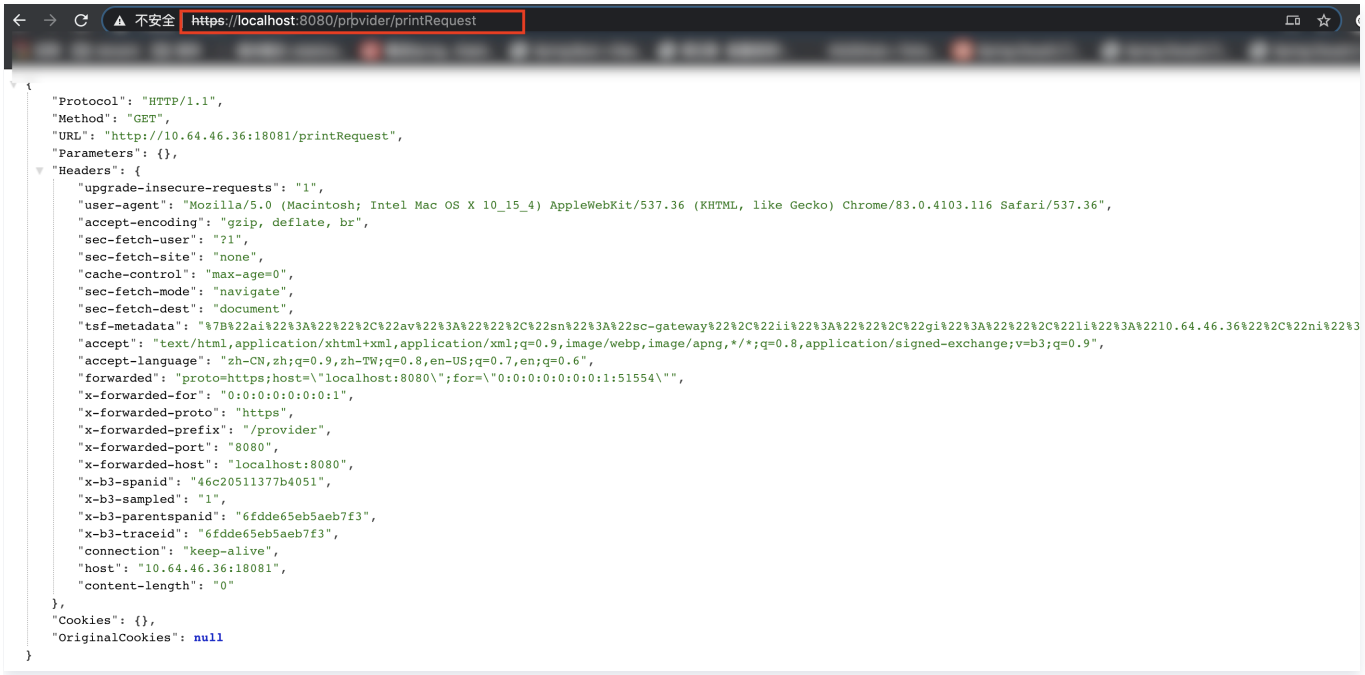


scg application.yml 配置如下：

```yaml
server:
  port: 8080
  ssl:
    enabled: true
    key-store: classpath:https/keystore.p12
    key-store-password: 123456
    key-alias: tomcathttps
    key-store-type: JKS
  error:
    include-exception: true
spring:
  application:
    name: sc-gateway
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lower-case-service-id: false
      httpclient:
        connectTimeout: 10
        responseTimeout: 20s
      routes:
        - id: api-v1
          uri: lb://PROVIDER-DEMO
          order: 0
          predicates:
           - Path=/provider/**
          filters:
           - StripPrefix=1
#    consul:
#        host: 127.0.0.1
#        port: 8500
#        enabled: true
```

```
#        scheme: HTTP
logging:
  file: /var/logs/${spring.application.name}/${spring.application.name}.log
  level:
    root: INFO
    com.tencent.tsf: INFO
```

HTTPS 访问网关成功的显示如下：



常见问题

问题描述：当使用自签名证书时，Chrome 会拦截请求，并展示 ERR_CERT_INVALID，旧版本可以选择跳过，继续访问，但是新版本 Chrome 不允许继续，且提示：您的连接不是私密连接攻击者可能会试图从 XX.XX.XX.XX 窃取您的信息（例如：密码、通讯内容或信用卡信息）。
解决方案：在 Chrome 该页面上，输入 thisisunsafe，即可进行访问，用来验证 HTTPS 配置是否成功。线上环境不推荐使用自签名证书。

开启网关预热功能

最近更新时间：2022-10-21 15:07:53

网关预热原理

用户点击发布分组的时候，会将 API 信息同步到 Consul。网关感知到 Consul 变动，会重新拉取 Consul 的值，并将 Service 重新加载到内存，达到预加载的效果。

使用方法

预热开关默认关闭，预热开关 key 为：

```
tsf.gateway.ribbon.eager-load.enabled
```

> **说明**
> value 为 true 表示打开预热，为其他值时表示关闭。

示例

启动预热，在网关配置文件添加以下内容：

```
tsf:
  gateway:
    ribbon:
      eager-load:
        enabled: true
```

测试联调
轻量级服务注册中心

最近更新时间：2022-06-10 09:49:09

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，例如某台测试机器。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`。

> **说明**
> 用户可通过执行 `netstat -apn|grep LISTEN` 命令查看端口占用信息。

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，例如 /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; sudo ./start.sh local_ip
```

其中 `local_ip` 填写本机 IP。例如，Linux 机器上的 IP 为 192.168.1.10，那么执行的命令是：`sudo ./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。

本地开发联调

最近更新时间：2024-12-24 21:02:52

本文介绍了三种本地开发联调的使用场景：

- 本地普通服务之间调用
- 本地微服务网关和普通服务之间的调用
- 本地服务调用云端服务

本地普通服务之间调用

操作场景

开发者通过搭建本地轻量级注册中心，将本地服务注册到轻量级注册中心上，服务之间通过服务名来进行调用。

操作步骤

步骤1：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 。

步骤2：启动应用

本地启动应用可以通过 IDE 和 FatJar 两种方式。您可以单击以下页签，查看两种方式的操作步骤：

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，即可在 target 目录下找到打包好的 FatJar 文件。
3. 通过 Java 命令启动：

```
java -Dtsf_instance_id=1111 -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。

> **注意**
> - Instance_id 是服务实例唯一标识，需要保证唯一性。
> - 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)`。

步骤3：验证

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`

本地微服务网关和普通服务之间的调用

操作场景

开发者通过搭建本地轻量级注册中心，将本地服务注册到轻量级注册中心上，服务之间通过服务名来进行调用。

操作步骤

步骤1：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 。

步骤2：本地启动微服务网关应用

本地启动应用可以通过 IDE 和 FatJar 两种方式。您可以单击以下页签，查看两种方式的操作步骤：

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_token=c -Dtsf_app_id=d -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 -Dtsf_token=c -Dtsf_app_id=d`，取值可为任意值。

> **注意：**
> TSF 1.29.5-Finchley-RELEASE、1.29.1-Greenwich-RELEASE、1.29.3-Hoxton-Higher-RELEASE 之前的版本必须需要设置以上参数才能在本地环境正常启动，否则会报错。

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，即可在 target 目录下找到打包好的 FatJar 文件。
3. 通过 Java 命令启动

```
java -jar -Dtsf_consul_ip=127.0.0.1 -Dtsf_consul_port=8500 -Dtsf_application_id=a -
Dtsf_group_id=b -Dtsf_token=c -Dtsf_app_id=d -Dtsf.swagger.enabled=false msgw-scg-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、tsf_group_id、tsf_token 和 tsf_app_id 可以填任意值。

> **注意**
> - Instance_id 是服务实例唯一标识，需要保证唯一性。
> - 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)` 。

步骤3：配置分组和分组 API

将 localtest.zip 包中的 group.json 和 api.json 复制到 `/$user.home/tsf/gateway/` 下，其中 `$user.home` 指的是用户目录，例如：/root（Linux），或 C:\Users\<用户名>（Windows），或/Users/<用户名>（MacOs）。其中 group.json 是部署组下的分组信息，api.json 是部署组下的分组 API 信息，默认要调用的下游微服务名是 provider-demo。

步骤4：验证

`http://{msgw-scg-ip}:{msgw-scg-port}/test_group1/test_namespace/provider-demo/echo/1234`
请求路径是：`http://网关ip:网关port/网关分组context/命名空间名称/微服务名/API路径`
如果本地网关调用的微服务名不是 provider-demo，则想要修改分组 API 配置中的微服务名，即需要修改 api.json 里的 serviceName 属性的值（修改下图中红框标出的内容）：

```
{
    "gatewayName":"test-group",
    "gatewayGroupId":"group-yrlkqnly",
    "reversion":80,
    "updatedTime":"2021-03-29 11:56:52",
    "result":[
        {
            "apiId":"api-c5zjvdbq",
            "groupId":"grp-60khzyb4",
            "path":"/echo/{param}",
            "method":"GET",
            "serviceName":"provider-demo",
            "namespaceId":"namespace-v6m6pp5a",
            "namespaceName":"test_namespace",
            "usableStatus":"enabled",
            "pathMapping":"/echo/{param}",
            "apiType":"ms"
        },
        {
            "apiId":"api-lc5cxghw",
            "groupId":"grp-60khzyb4",
            "path":"/echo/slow/{param}",
            "method":"GET",
            "serviceName":"provider-demo",
            "namespaceId":"namespace-v6m6pp5a",
            "namespaceName":"test_namespace",
            "usableStatus":"enabled",
            "pathMapping":"/echo/slow/{param}",
            "apiType":"ms"
        }
    ]
}
```


group.json 内容解释：

其中 gatewayName 是部署组名称，gatewayGroupId 是部署组ID，groupId 是网关分组ID，groupName 是网关分组名称，groupContext 是网关分组 context。

```
{
    "gatewayName":"test-group",            //部署组名称
    "gatewayGroupId":"group-yrlkqnly",     //部署组ID
    "reversion":24,
    "updatedTime":"2021-03-29 11:56:52",
    "result":[
        {
            "groupId":"grp-60khzyb4",      //网关分组ID
            "groupName":"group1",          //网关分组名称
            "groupContext":"/test_group1", //网关分组 context
            "authMode":"none",
            "groupType":"ms"
        },
        {
            "groupId":"grp-sh1b4f4w",
            "groupName":"group2",
            "groupContext":"/test_group2",
```

```
            "authMode":"none",
            "groupType":"ms"
        }
    ]
}
```

api.json 内容解释：
其中 gatewayName 是部署组名称，gatewayGroupId 是部署组ID，groupId 是网关分组ID（与 group.json 中的 groupId 保持一致时会认为该 API 属于该网关分组），apiId 是网关分组 API 的 ID，serviceName 是网关下游的微服务名（也就是下游微服务的名称），path 是网关分组 API 的路径（也就是下游微服务的 API）

```
{
    "gatewayName":"test-group",                 //部署组名称
    "gatewayGroupId":"group-yrlkqnly",          //部署组ID
    "reversion":80,
    "updatedTime":"2021-03-29 11:56:52",
    "result":[
        {
            "apiId":"api-c5zjvdbq",             //网关分组 API 的 ID
            "groupId":"grp-60khzyb4",           //网关分组ID（与 group.json 中的 groupId 保持一致时会认
为该 API 属于该网关分组）
            "path":"/echo/{param}",             //网关分组 API 的路径（也就是下游微服务的 API）
            "method":"GET",
            "serviceName":"provider-demo",      //网关下游的微服务名（也就是下游微服务的名称）
            "namespaceId":"namespace-v6m6pp5a",
            "namespaceName":"test_namespace",
            "usableStatus":"enabled",
            "pathMapping":"/echo/{param}",
            "apiType":"ms"
        },
        {
            "apiId":"api-lc5cxghw",
            "groupId":"grp-60khzyb4",
            "path":"/echo/slow/{param}",
            "method":"GET",
            "serviceName":"provider-demo",
            "namespaceId":"namespace-v6m6pp5a",
            "namespaceName":"test_namespace",
            "usableStatus":"enabled",
            "pathMapping":"/echo/slow/{param}",
            "apiType":"ms"
        }
    ]
}
```

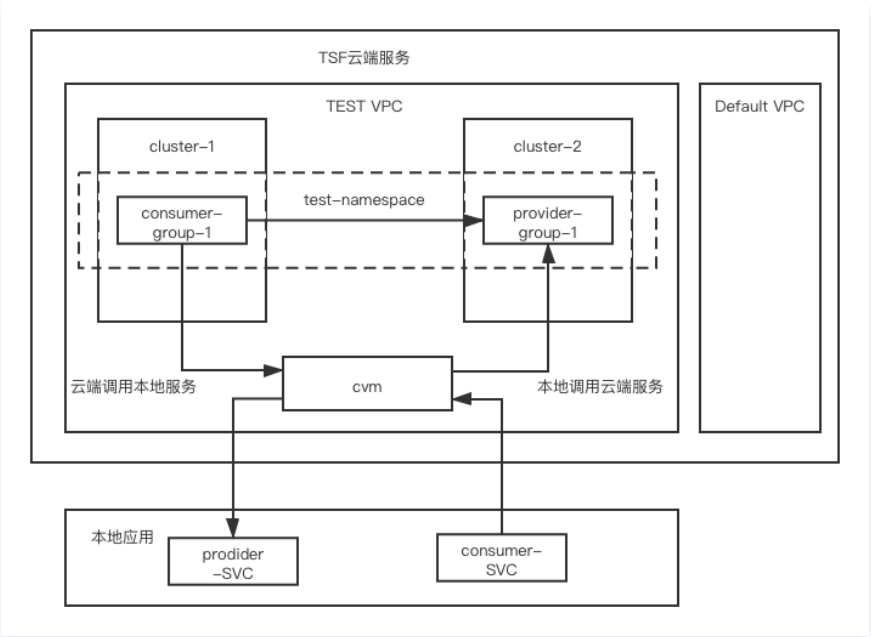
端云联调

最近更新时间：2023-12-06 17:02:12

操作场景

为满足企业开发者用户本地与云端联调测试需求，TSF提供端云联调开发者工具，通过在本地服务配置启动插件及参数即可实现本地应用和部署在云端的TSF应用相互调用测试联调能力，无需搭建VPN，帮助用户快速提升开发效率。

- **本地应用消费云端服务：**本地新增应用调用云端联调测试环境服务，提供云端服务测试联调能力，提升开发效率。
- **云端服务调用本地实例：**提供方服务已经部署至云端，需要对提供方服务进行更新，则可在本地启动提供方实例，通过指定路由规则联调测试本地服务进行功能验证。



前提条件

创建流量代理

1. 您需在腾讯云购买一台与云端联调服务同地域的 CVM 资源，将其加入到云端联调服务的 VPC 中，并绑定一个公网 IP。
2. 如果您需要云端调用本地测试服务，需要在上述已购买的 CVM 资源的 SSH 服务进行如下配置：

   2.1 编辑 sshd_config 配置

   ```
   sudo vim /etc/ssh/sshd_config
   ```

   2.2 添加如下配置信息允许远程主机连接本地的转发端口，并保存：

   ```
   GatewayPorts clientspecified
   ```

   2.3 重启 SSH 服务

   Debian/Ubuntu linux：

   ```
   sudo systemctl restart ssh
   ```

   CentOS / RHEL / Fedora / Redhat Linux

```
sudo systemctl restart sshd
```

使用限制

- 作为跳板机的 CVM 必须和联调服务集群处于同一个私有网络 VPC。
- 支持 Spring Cloud 架构的普通应用、Mesh 应用和微服务网关。
- 不支持分布式任务调度、分布式事务、日志、监控、分布式链路追踪的云端联调能力。
- 云端服务调用本地服务的方式，仅支持 1.29 及以下版本的 Spring Cloud SDK，1.40及以上版本的 Spring Cloud SDK 暂不支持。

操作步骤

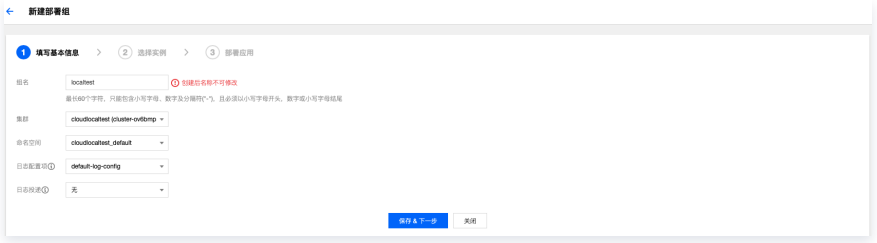
步骤1：创建应用

如果本地测试服务未在云端部署则需为其创建应用并创建部署组获取联调配置，如果本地测试服务已在云端部署仅为本地开发测试实例则仅需在已有的云端应用下创建部署组获取联调配置，您可以使用 TSF 官网提供的 Demo（单击下载）进行验证。

步骤2：创建部署组获取联调配置

1. 创建部署组

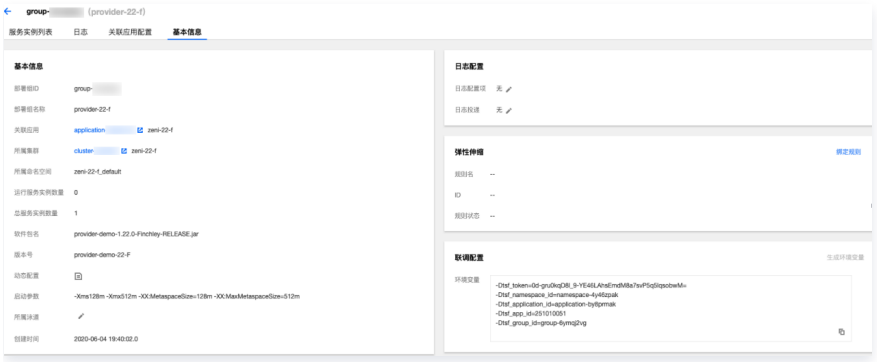
   创建部署组关联云端测试服务的集群及命名空间，确保其能相互访问，无需关联实例及部署应用单击保存。



2. 获取联调配置

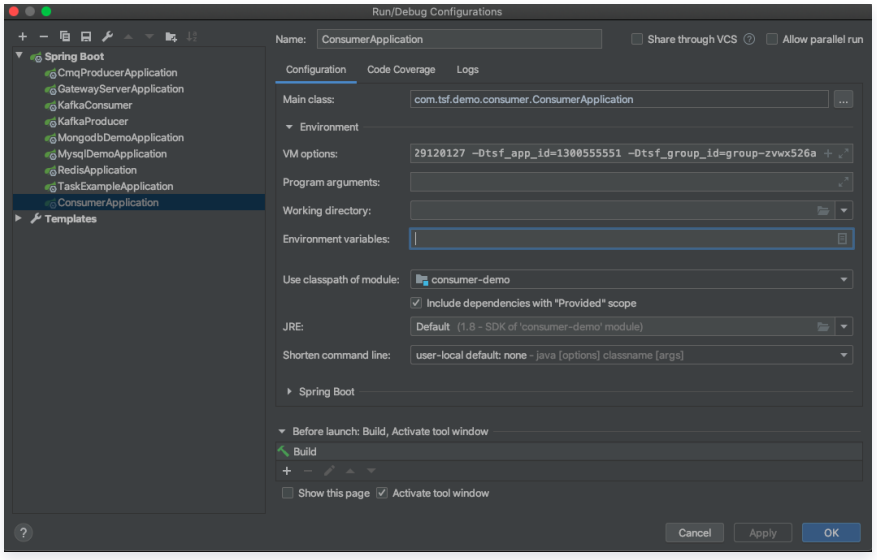
单击部署组 ID，查看基本信息获取联调配置，具体参数如下所示：

参数	描述
tsf_token	TSF 服务注册 token 认证信息
tsf_namespace_id	TSF 命名空间 ID
tsf_applicaton_id	TSF 应用 ID
tsf_appid_id	腾讯云账号 ID 信息
tsf_group_id	TSF 部署组 ID



步骤3：本地环境配置

1. 下载本地联调 agent 程序包（单击下载）。
2. 打开本地 IDE 环境，配置 JVM 启动参数。



JVM 启动参数配置如下：

```
-javaagent:"agent绝对路径=ssh_host=cvm公网ip&&user=用户名&&pass=密码&&local_port=本地服务启动端口
&&remote_ip=cvm内网ip"
-Dtsf_token=TSF服务注册token认证信息
-Dtsf_namespace_id=TSF命名空间ID
-Dtsf_application_id=TSF应用ID
-Dtsf_app_id=腾讯云 账户ID信息
-Dtsf_group_id=TSF部署组ID
```

示例：

```
-
javaagent:"/Users/root/agent.jar=ssh_host=139.136.79.195&&user=root&&pass=123456tt&&local_port=8
080&&remote_ip=172.30.0.93"
-Dtsf_token=rakxSMEnGjXtAvSe9I2BlvqOootl_WAw9YzIiaJ-RD4=
-Dtsf_namespace_id=namespace-py5lr6v4
-Dtsf_application_id=application-nygxjma2
-Dtsf_app_id=1300555551
-Dtsf_group_id=group-jy9zr8ag
```

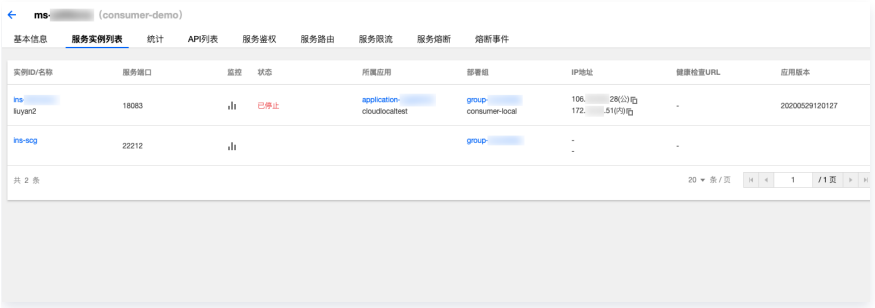
配置详情描述如下：

参数	描述	必填
javaagent	JavaAgent 动态代理 jar 包路径	必填
ssh_host	SSH 服务器地址	必填
ssh_port	SSH 服务监听端口	必填
user	SSH 服务登录用户名	必填
pass	SSH 服务登录密码	选填，pass 和 key 必须要有一个不为空
key	SSH 服务登录私钥路径	选填，pass 和 key 必须要有一个不为空
local_port	本地启动服务端口	选填，如果本地调试服务只是正向调用云端服务则可以不填写，如果需要被云端服务访问则需要填写
remote_ip	CVM 内网 IP	选填，如果本地调试服务只是正向调用云端服务则可以不填写，如果需要被云端服务访问则需要填写
tsf_token	TSF 服务注册 token 认证信息	必填
tsf_namespace_id	TSF 命名空间 ID	必填
tsf_applicaton_id	TSF 应用 ID	必填
tsf_appid_id	腾讯云账号 ID 信息	必填
tsf_group_id	TSF 部署组 ID	必填
tsf_instance_id	注册实例 ID	选填 ，如果有多个同类型的本地服务需要注册，为了避免实例 ID 相同可以特殊指定

3. 启动本地服务。

步骤4：联调测试验证

本地服务启动后会在 TSF 服务治理对应的服务详情中显示服务实例列表信息，包含实例信息、服务端口、部署组信息。



在本地通过联调测试远程服务，显示 response 信息则验证成功。

```
tools % curl http://127.0.0.1:18083/echo-rest/test
request param: test, response from echo-provider-default-name%
```

应用打包
制作应用程序包

最近更新时间：2024-01-19 10:55:52

应用开发联调完成后，需要将应用工程进行打包，部署到 TSF 中。

> **说明**
> 当部署 Spring Cloud 类型的微服务时，当前只有 jar 包部署的微服务支持完整的服务注册发现、完整的服务监控、调用链和服务治理能力。

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

- jar： FatJar 格式的程序包，FatJar 是一种可执行的 Jar 包（Executable Jar）。FatJar 和普通的 Jar 不同在于它包含了依赖的 Jar 包。用户可以参见 如何打 FatJar 包 。
- war： war 格式的程序包，在部署 war 包时，TSF 会自动安装 Tomcat 环境。示例 demo 下载 ，示例 demo 部署成功后，使用 `http://<IP>:8080/provider-demo/echo/test` 访问，其中 IP 可以是实例的外网 IP。
- 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
war	云服务器上的 agent 会使用 `java -jar` 命令启动程序。
jar	云服务器上的 agent 会使用 `java -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。

如何打 FatJar 包

1. 在工程的 pom.xml 文件中添加插件：

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

2. 添加完插件后，在 pom.xml 所在目录，使用 maven 命令 `mvn clean package` 即可下载 TSF SDK 并将应用工程进行打包，在 target 目录下找到打包好的 FatJar 文件。

> **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。

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 -jar 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。

```
ubuntu@VM-1-12-ubuntu:~$ ps aux | grep 'java -Xms128m -Xmx512m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m -jar provider-demo-0.0.1-SNAPSHOT.jar'
root      9824  0.5  5.6 4083536 452936 ?      Sl   Nov15  59:20 java -Xms128m -Xmx512m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m -jar provider-demo-0.0.1-SNAPSHOT.jar
ubuntu   14258  0.0  0.0  13228   936 pts/0    S+   15:17   0:00 grep --color=auto java -Xms128m -Xmx512m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m -jar provider-demo-0.0.1-SNAPSHOT.jar
```

> **注意**
> 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**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。

后续操作

应用打包完成后，您可以将应用程序包上传至 TSF 控制台进行应用部署，详情请参见 虚拟机托管应用 。

制作容器镜像

最近更新时间：2024-05-29 10:39:21

本文介绍通过 Spring Cloud 应用和 Mesh 应用制作容器镜像的操作方法。
关于 JDK 版本，推荐使用 Tencent KonaJDK，请下载 KonaJDK 安装文件（ 下载地址 ）。

准备构建材料

Spring Cloud 应用构建材料

1. 简化版本
简化版本的 Dockerfile 不包含文件配置和 JVM 监控功能，仅需要用户替换掉 Dockerfile 中 Spring Cloud 应用 jar 包名称，您也可以先试用 TSF 提供的 Spring Cloud 应用 Demo JAR 包（ 下载地址 ）。

> **注意**
> 在 Spring Cloud 应用 JAR 包同级目录下编写 Dockerfile。

KonaJDK

```
FROM centos:7
RUN echo "ip_resolve=4" >> /etc/yum.conf
#安装 KonaJDK
ADD ./java-8-konajdk.rpm /java-8-konajdk.rpm
RUN yum update -y && yum install -y java-8-konajdk.rpm

# 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/

# 下面的 jar 包可替换为您的 Spring Cloud 应用 jar包，注意这个 jar 包要和您的 dockerfile 位于同一级目录
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。
# 考虑到容器场景对于内存的要求，建议添加-Xshare:off选项关闭CDS功能
CMD ["sh", "-ec", "exec java ${JAVA_OPTS} -Xshare:off -jar ${jar}"]
```

OpenJDK

```
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/

# 下面的 jar 包可替换为您的 Spring Cloud 应用 jar包，注意这个 jar 包要和您的 dockerfile 位于同一级目录
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}"]
```

2. 使用 JVM 监控功能

如果您希望使用 JVM 监控 功能，则需要在 Dockerfile 中增加 JVM 监控组件 `TencentCloudJvmMonitor-1.3.1` （ 下载地址 ），JVM 监控包相关版本更新列表请查看文档，然后在 CMD 命令中启动该组件。

> ⚠ **注意**
> 将 Spring Cloud 应用 JAR 包和 JVM 监控组件放在同级目录下，并在该目录下编写 Dockerfile。

KonaJDK

```
FROM centos:7
RUN echo "ip_resolve=4" >> /etc/yum.conf
#安装 KonaJDK
ADD ./java-8-konajdk.rpm /java-8-konajdk.rpm
RUN yum update -y && yum install -y java-8-konajdk.rpm

# 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/

# 下面的 jar 包可替换为您的 Spring Cloud 应用 jar 包，注意这个 jar 包要和您的 dockerfile 位于同一级目录
ENV jar provider-demo-0.0.1-SNAPSHOT.jar
COPY ${jar} ${workdir}
WORKDIR ${workdir}

# JVM 监控组件要和您的 Dockerfile 位于同一级目录，并创建 JVM 监控数据采集目录
ENV agentjar TencentCloudJvmMonitor-1.3.1-RELEASE.jar
# 若容器的基础版本为 非 gnu-libc 版本，如 Alpine，请添加如下语句
# RUN ln -sf /lib/libc.musl-x86_64.so.1 /lib/ld-linux-x86-64.so.2
COPY ${agentjar} ${workdir}

RUN mkdir -p /data/tsf_apm/monitor/jvm-metrics/

# 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。
# 使用 JVM监控功能需要加上 gclog 和 javaagent 的配置，否则将无法提供 jvm 监控能力
# 考虑到容器场景对于内存的要求，建议添加-Xshare:off选项关闭CDS功能
CMD ["sh", "-ec", "exec java -Xloggc:/data/tsf_apm/monitor/jvm-metrics/gclog.log -
XX:+PrintGCDateStamps -XX:+PrintGCDetails -verbose:gc -XX:+UseGCLogFileRotation -
```

```
XX:NumberOfGCLogFiles=8 -XX:GCLogFileSize=50M -
javaagent:${workdir}/${agentjar}=hascontroller=true ${JAVA_OPTS} -Xshare:off -jar ${jar}"]
```

OpenJDK

```
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/

# 下面的 jar 包可替换为您的 Spring Cloud 应用 jar 包，注意这个 jar 包要和您的 dockerfile 位于同一级目录
ENV jar provider-demo-0.0.1-SNAPSHOT.jar
COPY ${jar} ${workdir}
WORKDIR ${workdir}

# JVM 监控组件要和您的 Dockerfile 位于同一级目录，并创建 JVM 监控数据采集目录
ENV agentjar TencentCloudJvmMonitor-1.3.1-RELEASE.jar
# 若容器的基础版本为 非 gnu-libc 版本，如 Alpine，请添加如下语句
# RUN ln -sf /lib/libc.musl-x86_64.so.1 /lib/ld-linux-x86-64.so.2
COPY ${agentjar} ${workdir}

RUN mkdir -p /data/tsf_apm/monitor/jvm-metrics/

# 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。
# 使用 JVM监控功能需要加上 gclog 和 javaagent 的配置，否则将无法提供 jvm 监控能力
CMD ["sh", "-ec", "exec java -Xloggc:/data/tsf_apm/monitor/jvm-metrics/gclog.log -
XX:+PrintGCDateStamps -XX:+PrintGCDetails -verbose:gc -XX:+UseGCLogFileRotation -
XX:NumberOfGCLogFiles=8 -XX:GCLogFileSize=50M -
javaagent:${workdir}/${agentjar}=hascontroller=true ${JAVA_OPTS} -jar ${jar}"]
```

3. 使用文件配置

如果您希望使用 TSF 文件配置 功能，则需要在 Dockerfile 中增加文件配置组件 `tsf-consul-template-docker.tar.gz` （ 下载地址 ），然后在 CMD 启动命令中启动该组件。

KonaJDK

```
FROM centos:7
RUN echo "ip_resolve=4" >> /etc/yum.conf
#安装 KonaJDK
ADD ./java-8-konajdk.rpm /java-8-konajdk.rpm
RUN yum update -y && yum install -y java-8-konajdk.rpm

# 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/

# 下面的 jar 包可替换为您的 Spring Cloud 应用 jar包，注意这个 jar 包要和您的 dockerfile 位于同一级目录
ENV jar provider-demo-0.0.1-SNAPSHOT.jar
COPY ${jar} ${workdir}
WORKDIR ${workdir}

# tsf-consul-template-docker 用于文件配置功能，如不需要可注释掉该行
ADD tsf-consul-template-docker.tar.gz /root/

# 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。
# 考虑到容器场景对于内存的要求，建议添加-Xshare:off选项关闭CDS功能
CMD ["sh", "-ec", "sh /root/tsf-consul-template-docker/script/start.sh; exec java ${JAVA_OPTS}
-Xshare:off -jar ${jar}"]
```

私有化版本使用建议：

私有化的 TSF 要支持 stdout 日志，需要在启动命令中将 stdout 及 stderr 重定向到一个文件中。将上文的 `CMD` 一行替换成：

```
RUN mkdir -p /data/tsf_std/stdout/logs
CMD ["sh", "-ec", "exec java ${JAVA_OPTS} -Xshare:off -jar ${jar} 2>&1 >
/data/tsf_std/stdout/logs/sys_log.log"]
```

OpenJDK

```
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/

# 下面的 jar 包可替换为您的 Spring Cloud 应用 jar包，注意这个 jar 包要和您的 dockerfile 位于同一级目录
ENV jar provider-demo-0.0.1-SNAPSHOT.jar
COPY ${jar} ${workdir}
WORKDIR ${workdir}

# tsf-consul-template-docker 用于文件配置功能，如不需要可注释掉该行
ADD tsf-consul-template-docker.tar.gz /root/

# 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", "sh /root/tsf-consul-template-docker/script/start.sh; exec java ${JAVA_OPTS}
-jar ${jar}"]
```

私有化版本使用建议：

私有化的 **TSF 1.12 及之前版本**要支持 stdout 日志，需要在启动命令中将 stdout 及 stderr 重定向到一个文件中。将上文的 `CMD` 一行替换成：

```
RUN mkdir -p /data/tsf_std/stdout/logs
```

```
CMD ["sh", "-ec", "exec java ${JAVA_OPTS} -jar ${jar} 2>&1 >
/data/tsf_std/stdout/logs/sys_log.log"]
```

Mesh 应用构建材料

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

KonaJDK

```
FROM centos:7
# 添加 KonaJDK 安装包
ADD ./java-8-konajdk.rpm /java-8-konajdk.rpm
# 安装 KonaJDK
RUN yum update -y && yum install -y /java-8-konajdk.rpm

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"]
```

OpenJDK

```
FROM centos:7
RUN yum update -y && yum install -y java-1.8.0-openjdk

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见 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，具体使用指引参见 文件配置 > 前提条件 。

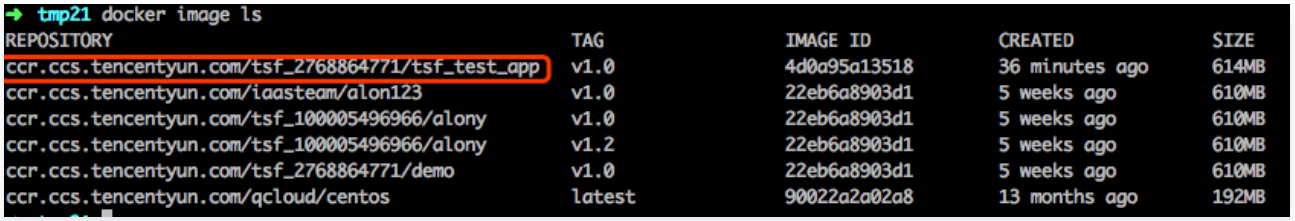
构建镜像

1. 在 `Dockerfile` 所在目录执行 `build` 命令：

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

其中 `<主账号 ID>` 对应用户腾讯云的**主账号 ID**（注意不是当前登录账号 ID，主账号 ID 可以在腾讯云个人信息页面获取。），`<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。

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



```
➜ tmp21 docker image ls
REPOSITORY                                          TAG       IMAGE ID       CREATED          SIZE
ccr.ccs.tencentyun.com/tsf_2768864771/tsf_test_app  v1.0      4d0a95a13518   36 minutes ago   614MB
ccr.ccs.tencentyun.com/iaasteam/alon123             v1.0      22eb6a8903d1   5 weeks ago      610MB
ccr.ccs.tencentyun.com/tsf_100005496966/alony       v1.0      22eb6a8903d1   5 weeks ago      610MB
ccr.ccs.tencentyun.com/tsf_100005496966/alony       v1.2      22eb6a8903d1   5 weeks ago      610MB
ccr.ccs.tencentyun.com/tsf_2768864771/demo          v1.0      22eb6a8903d1   5 weeks ago      610MB
ccr.ccs.tencentyun.com/qcloud/centos                latest    90022a2a02a8   13 months ago    192MB
```

后续操作

镜像制作完成后，您可以将镜像推送至镜像仓库进行应用部署。推送镜像请参见 镜像管理，部署应用请参见 容器托管应用。

YAML 格式介绍

最近更新时间：2024-01-19 10:55:52

YAML 专门用来写配置文件的语言。

语法规则

YAML 的基本语法规则如下：

- 大小写敏感。
- 使用缩进表示层级关系。
- 缩进时**不允许**使用 Tab 键，只允许使用空格。
- 缩进的空格数目不重要，只要相同层级的元素左侧对齐即可。

数据结构

YAML 支持三种数据结构：对象、数组和纯量。

- 对象：键值对的集合，又称为映射（mapping）/ 哈希（hashes）/ 字典（dictionary）。
- 数组：一组按次序排列的值，又称为序列（sequence）/ 列表（list）。
- 纯量（scalars）：单个的、不可再分的值。

对象

简单对象

```
foo: whatever
bar: stuff
```

复杂对象

```
foo: whatever
bar:
 -
   fruit: apple
   name: steve
   sport: baseball
 - more
 -
   python: rocks
   perl: papers
   ruby: scissorses
```

转换为 JavaScript 代码后：

```
{ foo: 'whatever',
  bar:
   [ { fruit: 'apple', name: 'steve', sport: 'baseball' },
     'more',
     { python: 'rocks', perl: 'papers', ruby: 'scissorses' } ] }
```

数组

```
- Cat
- Dog
```

```
- Goldfish
```

纯量

纯量是最基本的、不可再分的值。

```
- 字符串
- 布尔值
- 整数
- 浮点数
- Null
- 时间
- 日期
```

字符串

字符串是比较复杂的类型，举例说明：

```
str: 这是一行字符串
```

如果字符串之中包含空格或特殊字符，需要放在引号之中。

```
str: '内容: 字符串'
```

单引号和双引号都可以使用，双引号不会对特殊字符转义。

```
s1: '内容\n字符串' # 会对 \n 字符转义
s2: "内容\n字符串" # 不会对 \n 字符转义
```

多行字符串可以使用 `|` 保留换行符，也可以使用 `>` 折叠换行。

```
this: |
  Foo
  Bar
that: >
  Foo
  Bar
```

转换为 JavaScript 代码：

```
{ this: 'Foo\nBar\n', that: 'Foo Bar\n' }
```

工具

- YAML 的格式校验工具
- YAML 和 Properties 格式互转工具

参见

- Yaml Cookbook：提供了很多典型的 YAML 用例
- YAML Syntax - Ansible：写了一些 YAML 的常见陷阱