注册配置治理

实践教程

腾讯云

文档目录

实践教程
Spring Cloud 实践教程
Spring Cloud 应用接入
Spring Cloud 应用接入 ZooKeeper-CVM 部署场景

最近更新时间：2025-04-17 16:00:54

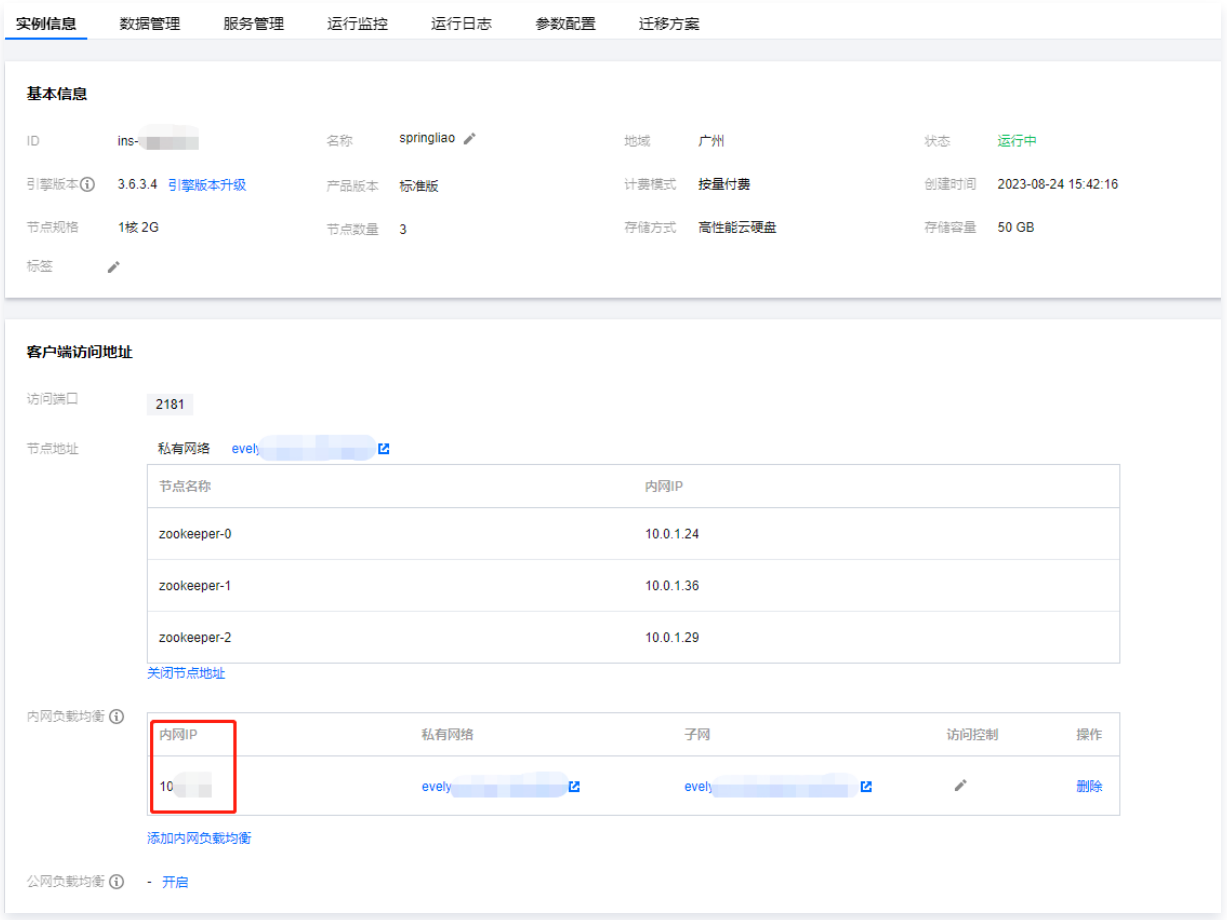
操作场景

本文以一对 Demo 示例（包含一个 provider 应用和一个 consumer 应用）介绍如何将通过 CVM 部署的 Spring Cloud 应用接入微服务引擎托管的 ZooKeeper 注册中心，并实现简单的服务访问。帮助您快速了解如何使用 ZooKeeper 注册中心。

前提条件

- 已创建 ZooKeeper 注册中心，请参见 引擎管理 。
- 已 购买云服务器 CVM，且云服务器所在的私有网络 VPC 与 ZooKeeper 注册中心所在的 VPC 相同。
- 下载 Github 的 Demo 源码 到本地并解压。
- 本地编译构建打包机器环境已安装了Java JDK、Maven，并且能够访问 Maven 中央库。

操作步骤

1. 登录 TSF 控制台，左侧导航栏选择 **Zookeeper**，进入引擎实例列表页。
2. 单击已创建的引擎实例的“ID”，进入基本信息页面。
3. 在页面上方选择**实例信息**页签，可以获取 ZooKeeper 注册中心实例访问 IP。
4. 

5. 进入下载好 Demo 源码目录。
6. 打包 Demo 源码成 jar 包。在 `tse-simple-demo-main` 源码根目录下，打开终端窗口，执行 mvn clean package 命令，对项目进行打包编译。编译成功后，可以在如下目录看到生成如下表所示的2个ZooKeeper Jar 包。

软件包所在目录	软件包名称	说明
\tse-simple-demo-main\tse-zookeeper-provider-demo\target	tse-zookeeper-provider-demo-1.0-SNAPSHOT.jar	服务生产者
\tse-simple-demo-main\tse-zookeeper-consumer-demo\target	tse-zookeeper-consumer-demo-1.0-SNAPSHOT.jar	服务消费者

7. 将编译好的 jar 包上传至云服务器，详细操作请参见 如何将本地文件拷贝到云服务器 。
8. 登录云服务器，进入到刚刚上传 jar 文件所在的目录，可看到文件已上传到云服务器。

```
[root@VM-49-34-centos ~]# cd /home
[root@VM-49-34-centos home]# ll
total 79696
-rw------- 1 root root     2418 Aug 22 15:23 nohup.out
-rw-r--r-- 1 root root 37912576 Aug 22 15:27 tse-zookeeper-consumer-demo-1.0-SNAPSHOT.jar
-rw-r--r-- 1 root root 43689706 Aug 22 15:15 tse-zookeeper-provider-demo-1.0-SNAPSHOT.jar
```

9. 执行如下命令指定注册中心地址参数并运行该应用。

```
nohup java -Dspring.cloud.zookeeper.connect-string=[Zookeeper注册中心实例访问IP:2181] -jar [jar包名称] &
```

10. 运行成功后，登录登录 TSF 控制台，左侧导航栏选择 Zookeeper，进入引擎实例列表页。点击目标实例后进入注册中心实例的服务管理页面，若出现以下页面，则证明服务注册成功。

11. 登录云服务器，执行如下命令，调用 consumer 接口访问 provider 服务。

```
curl localhost:19001/ping/test
```

返回结果如下：

```
[root@VM-49-34-centos home]# curl localhost:19001/ping/test
provider echo: test[root@VM-49-34-centos home]#
```

注意事项

Spring Cloud 应用接入 Zookeeper 注册中心，配置文件格式需如下所示：

```
spring:
  cloud:
    zookeeper:
      connect-string: [zookeeper注册中心IP:2181]
      discovery:
        register: true
        enabled: true
        prefer-ip-address: true
```

Spring Cloud 应用接入 ZooKeeper-TKE 部署场景

最近更新时间：2025-04-17 16:00:54

操作场景

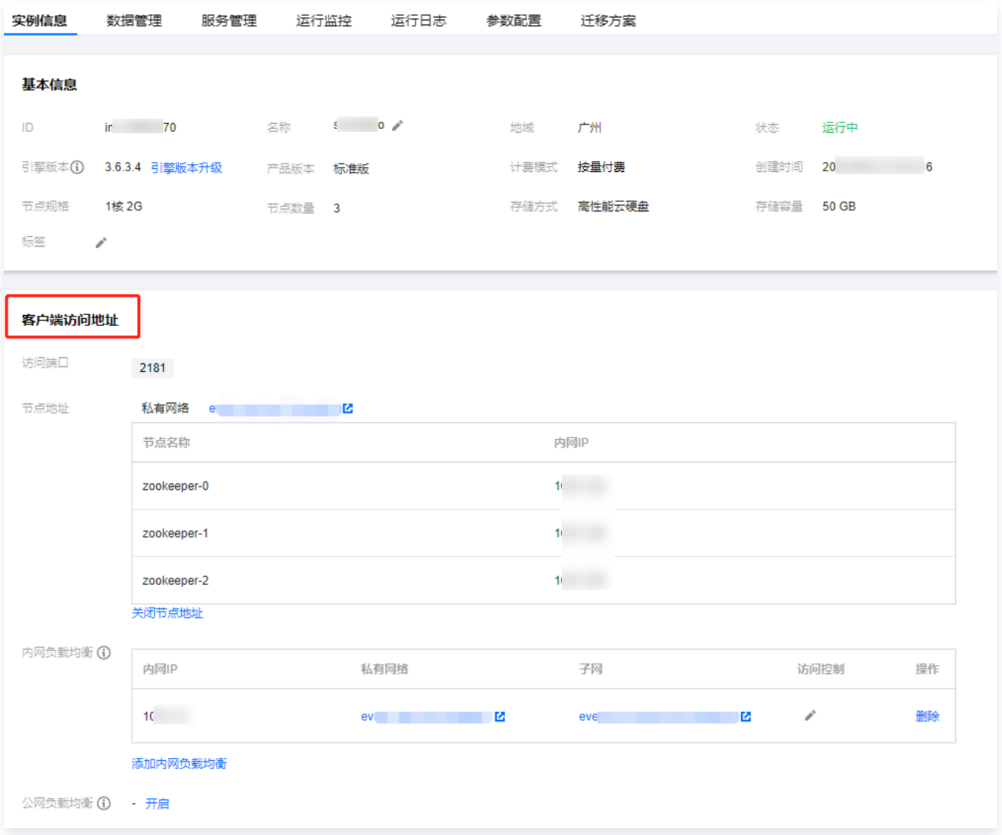
本文以一对 Demo 示例（包含一个 provider 应用和一个 consumer 应用）介绍如何将通过 TKE 部署的 Spring Cloud 应用接入微服务引擎托管的 ZooKeeper 注册中心，并实现简单的服务访问。帮助您快速了解如何使用 ZooKeeper 注册中心。

前提条件

- 已创建ZooKeeper 注册中心，请参见 引擎管理 。
- 本地编译构建打包机器环境已安装了 Java JDK、Maven，并且能够访问 Maven 中央库。

操作步骤

1. 创建 TKE 容器集群。
   登录 TKE 控制台，新建一个标准集群，容器集群所在的私有网络 VPC 需要与已创建好的 ZooKeeper 引擎所在的私有网络保持一致。具体操作参见 快速创建一个标准集群 。
2. 获取 ZooKeeper 注册中心实例访问 IP。
3. 登录 TSF 控制台，单击已创建好的 ZooKeeper 引擎实例的“ID”，进入基本信息页面，在访问管理页签可以获取 ZooKeeper 注册中心实例访问 IP。



4. 下载 Github 的 Demo 源码 到本地并解压。
5. 打包 Demo 源码成 jar 包。
   在 `tse-simple-demo-main` 源码根目录下，打开终端窗口，执行 `mvn clean package` 命令，对项目进行打包编译。编译成功后，可以在如下目录看到生成如下表所示的2个 ZooKeeper Jar 包。

软件包所在目录	软件包名称	说明

\tse-simple-demo-main\tse-zookeeper-provider-demo\target	tse-zookeeper-provider-demo-1.0-SNAPSHOT.jar	服务生产者
\tse-simple-demo-main\tse-zookeeper-consumer-demo\target	tse-zookeeper-consumer-demo-1.0-SNAPSHOT.jar	服务消费者

6. 制作 provider 和 consumer 应用容器镜像并上传至镜像仓库。

   6.1 编写 dockerfile 并生成镜像，dockerfile 内容参见：

```
FROM openjdk:8

ADD ./[jar包名称.jar] /root/app.jar

ENTRYPOINT  [ "sh", "-c", "java $JAVA_OPTS -jar /root/app.jar"]
```

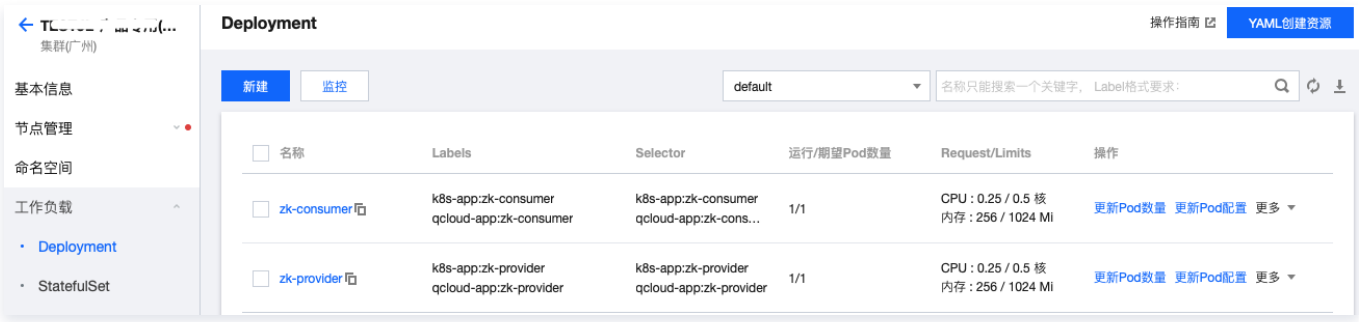
   6.2 参见 镜像仓库快速入门 上传 Spring Cloud 应用镜像至 TKE 镜像仓库

7. 在 TKE 容器集群中创建工作负载并选择对应镜像文件

   7.1 登录 TKE 控制台，找到已创建好的 TKE 容器集群，单击集群 ID，进入集群的**工作负载** > **Deployment**页面，创建工作负载并选择对应镜像文件，详细操作参见 Deployment 管理 。

   - 镜像：选择已上传的 Spring Cloud 应用镜像。
   - 镜像版本：选择已上传的 Spring Cloud 应用镜像版本。
   - 环境变量：新增环境变量 `JAVA_OPTS` 并指定为 `-Dspring.cloud.zookeeper.connect-string=[TSE Zookeeper注册中心实例访问IP:2181]`

7.2 Deployment 信息填写完成后，单击**创建 Deployment**，出现如下页面时，代表 Deployment 创建成功。



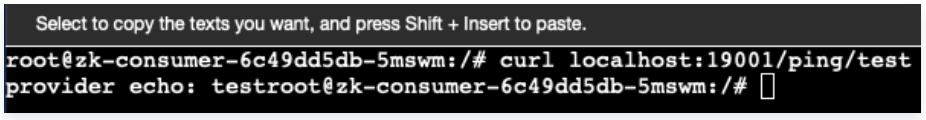
8. 验证服务注册成功。
9. 登录 TSF 控制台，在左侧导航栏选择 **Zookeeper**，单击目标实例的 ID，进入基本信息页面。在页面上方选择**服务管理**页签，若出现以下页面，则证明服务注册成功。

10. 验证服务调用。

登录 Consumer 服务所在的 Pod，执行 curl 命令调用 Consumer 接口访问 Provider 服务。


```
curl localhost:19001/ping/test
```

访问结果如下：



注意事项

Spring Cloud 应用接入 Zookeeper 注册中心，配置文件格式需如下所示：

```
spring:
  cloud:
    zookeeper:
      connect-string: [zookeeper注册中心IP:2181]
      discovery:
        register: true
        enabled: true
        prefer-ip-address: true
```

Spring Cloud 应用接入 Nacos-CVM 部署场景

最近更新时间：2025-04-17 16:00:54

操作场景

本文以一对 Demo 示例（包含一个 provider 应用和一个 consumer 应用）介绍如何将通过 CVM 部署的 Spring Cloud 应用接入微服务引擎托管的 Nacos 注册中心，并实现简单的服务访问。帮助您快速了解如何使用 TSF Nacos 注册中心。

前提条件

- 已创建 Nacos 注册中心，请参见 引擎管理 。
- 已 购买云服务器 CVM，且云服务器所在的私有网络 VPC 与 Nacos 注册中心所在的 VPC 相同。
- 下载 Github 的 Demo 源码 到本地并解压。
- 本地编译构建打包机器环境已安装了Java JDK、Maven，并且能够访问 Maven 中央库。

操作步骤

1. 登录 TSF 控制台，在左侧导航栏选择 **Nacos**，进入引擎实例列表页。
2. 单击已创建好的 Nacos 引擎实例的“ID”，进入基本信息页面.
3. 在页面上方选择"基本信息"页签，可以获取 Nacos 注册中心实例访问 IP。

4. 进入下载好的 Demo 源码目录。
5. 打包 Demo 源码成 jar 包。在 `tse-simple-demo-main` 源码根目录下，打开终端窗口，执行 `mvn clean package` 命令，对项目进行打包编译。编译成功后，可以在如下目录看到生成如下表所示的2个 Nacos Jar 包。

软件包所在目录	软件包名称	说明
tse-nacos-spring-cloud-provider-demo\target	tse-nacos-spring-cloud-provider-demo-2.0.1.RELEASE.jar	服务生产者
tse-nacos-spring-cloud-consumer-demo\target	tse-nacos-spring-cloud-consumer-demo-2.0.1.RELEASE.jar	服务消费者

6. 将编译好的 jar 包上传至云服务器，详细操作请参见 如何将本地文件拷贝到云服务器 。
7. 登录云服务器，进入到刚刚上传 jar 文件所在的目录，可看到文件已上传到云服务器。

```
[root@VM-49-34-centos ~]# cd /home
[root@VM-49-34-centos home]# ll
total 177416
-rw------- 1 root root 46809659 Aug 22 17:48 nacos-consumer-demo.jar
-rw------- 1 root root 46809410 Aug 22 17:50 nacos-provider-demo.jar
```

8. 执行如下命令指定注册中心地址参数并运行该应用。

```
nohup java -Dspring.cloud.nacos.discovery.server-addr=[TSE Zookeeper注册中心实例访问IP:8848] -jar
[jar包名称] &
```

9. 登录 TSF 控制台，在左侧导航栏选择 **nacos**，单击目标实例的 ID，进入基本信息页面。
10. 在页面上方选择**访问控制**页签，找到控制台访问的公网地址和登录用户名密码，通过 Web 访问 Nacos 原生控制台。
11. 

12. 在 Nacos 原生控制台页面，选择**服务管理 > 服务列表**，可以看到注册成功的服务。

13. 登录云服务器，执行如下命令，调用 consumer 接口访问 provider 服务。

```
curl localhost:8080/echo/str
```

运行结果如下：

```
[root@VM-49-34-centos home]# curl localhost:8080/echo/str
Hello Nacos Provider str[root@VM-49-34-centos home]#
```

Spring Cloud 应用接入 Nacos-TKE 部署场景

最近更新时间：2025-04-17 16:00:54

操作场景

本文以一对 Demo 示例（包含一个 provider 应用和一个 consumer 应用）介绍如何将通过 TKE 部署的 Spring Cloud 应用接入微服务引擎托管的 Nacos 注册中心，并实现简单的服务访问。帮助您快速了解如何使用 Nacos 注册中心。

前提条件

- 已创建 Nacos 注册中心，请参见 引擎管理 。
- 本地编译构建打包机器环境已安装了 Java JDK、Maven，并且能够访问 Maven 中央库。

操作步骤

1. 创建 TKE 容器集群。
   登录 TKE 控制台，新建一个标准集群，容器集群所在的私有网络 VPC 需要与已创建好的 Nacos 引擎所在的私有网络保持一致。具体操作参见 快速创建一个标准集群 。
2. 获取 Nacos 注册中心实例访问 IP。
3. 登录 TSF 控制台，在左侧导航栏选择Nacos，进入引擎实例列表页。单击已创建好的 Nacos 引擎实例的“ID”，进入基本信息页面，在访问控制页签可以获取 Nacos 注册中心实例访问 IP。

4. 下载 Github 的 Demo 源码 到本地并解压。
5. 打包 Demo 源码成 jar 包。
   在 `tse-simple-demo-main` 源码根目录下，打开终端窗口，执行 `mvn clean package` 命令，对项目进行打包编译。编译成功后，可以在如下目录看到生成如下表所示的2个 Nacos Jar 包。

软件包所在目录	软件包名称	说明
tse-nacos-spring-cloud-provider-demo\target	tse-nacos-spring-cloud-provider-demo-2.0.1.RELEASE.jar	服务生产者
tse-nacos-spring-cloud-consumer-demo\target	tse-nacos-spring-cloud-consumer-demo-2.0.1.RELEASE.jar	服务消费者

6. 制作 provider 和 consumer 应用容器镜像并上传至镜像仓库。
   6.1 编写 dockerfile 并生成镜像，dockerfile 内容参见：

```
FROM openjdk:8

ADD ./[jar包名称.jar] /root/app.jar

ENTRYPOINT  [ "sh", "-c", "java $JAVA_OPTS -jar /root/app.jar"]
```

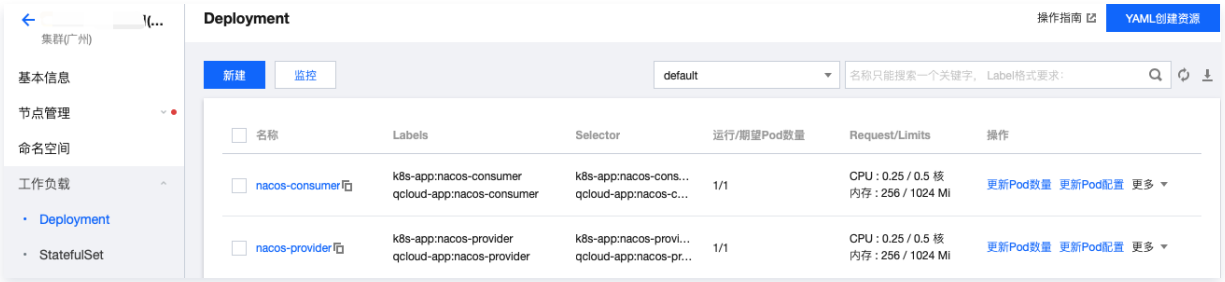
   6.2 参见 镜像仓库快速入门 上传 Spring Cloud 应用镜像至 TKE 镜像仓库。

7. 在 TKE 容器集群中创建工作负载并选择对应镜像文件。

7.1 登录 TKE 控制台，找到已创建好的 TKE 容器集群，单击集群 ID，进入集群的**工作负载** > **Deployment**页面，创建工作负载并选择对应镜像文件，详细操作参见 Deployment 管理 。

- 镜像：选择已上传的 Spring Cloud 应用镜像。
- 镜像版本：选择已上传的 Spring Cloud 应用镜像版本。
- 环境变量：新增环境变量 `JAVA_OPTS` 并指定为
  `-Dspring.cloud.nacos.discovery.server-addr=[Nacos 注册中心实例访问 IP:8848]`

7.2 Deployment 信息填写完成后，单击**创建 Deployment**，出现如下页面时，代表 Deployment 创建成功。



8. 验证服务注册成功。

8.1 登录 TSF 控制台，在左侧导航栏选择Nacos，进入引擎实例列表页。单击目标实例的 ID，进入基本信息页面。

8.2 在页面上方选择**访问控制**页签，找到控制台访问的公网地址和登录用户名密码，通过 web 访问 Nacos 原生控制台。

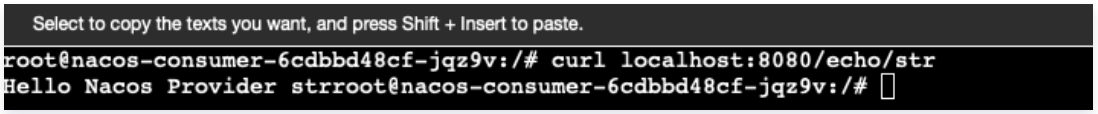
8.3 在 Nacos 原生控制台页面，选择**服务管理 > 服务列表**，可以看到注册成功的服务。

9. 验证服务调用。

登录 Consumer 服务所在的 Pod，执行 curl 命令调用 Consumer 接口访问 Provider 服务。


```
curl localhost:8080/echo/str
```

访问结果如下：

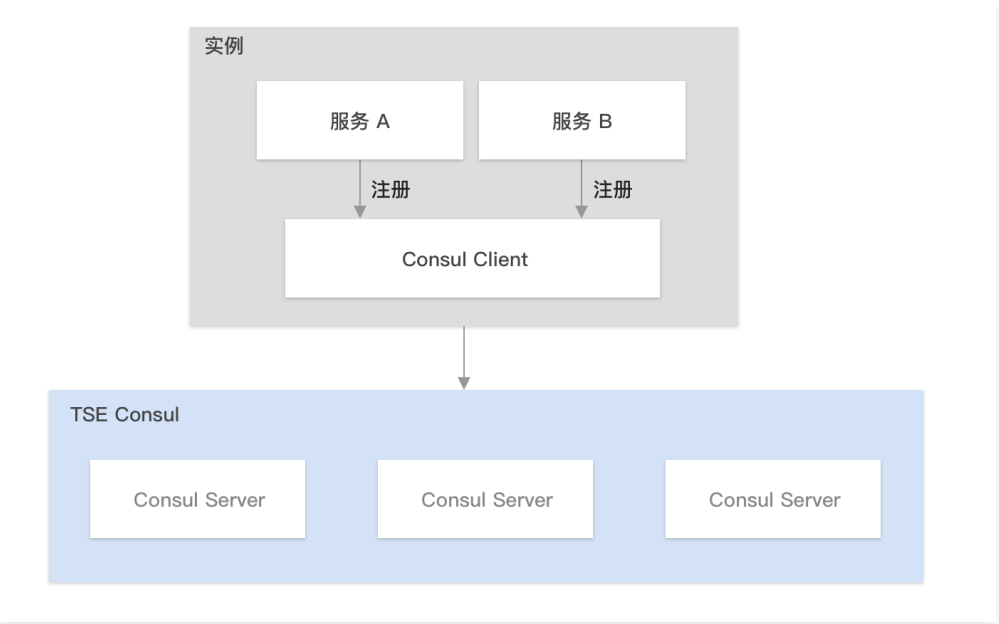


Spring Cloud 应用接入 Consul-CVM 部署

最近更新时间：2025-08-18 14:56:31

操作场景

本文以一对 Demo 示例（包含一个 provider 应用和一个 consumer 应用）介绍如何将通过 CVM 部署的 Spring Cloud 应用接入微服务引擎托管的 Consul 注册中心，并实现简单的服务访问。帮助您快速了解如何使用 Consul 注册中心。



前提条件

- 已创建 Consul 注册中心，请参见 引擎管理 。
- 已 购买云服务器 CVM，且云服务器所在的私有网络 VPC 与 Consul 注册中心所在的 VPC 相同。
- 下载 Github 的 Demo 源码 到本地并解压。
- 本地编译构建打包机器环境已安装了Java JDK、Maven，并且能够访问 Maven 中央库。

步骤1：准备配置

1. 下载 Consul，并将下载好的安装包上传至云服务器 CVM。详细操作请参见 如何将本地文件拷贝到云服务器 。
2. 登录云服务器，执行如下命令创建相关资源。

   2.1 执行如下命令创建数据文件夹。

   ```
   mkdir -p /data/consul/data
   ```

   2.2 执行如下命令创建日志文件夹

   ```
   mkdir -p /data/consul/log
   ```

   2.3 执行如下命令创建配置文件。

   ```
   vim /data/consul/config/config.json
   ```

   输入 `i` ，写入如下配置文件内容后，单击 `esc` ，输入 `:wq` 后回车，完成配置文件创建。

   ```
   {
   ```

```
  "datacenter": "dc1",
  "data_dir": "/data/consul/data/",
  "node_name": "consul-agent-node",
  "server": false,
  "bind_addr": "{{ GetInterfaceIP \"eth0\" }}",
  "client_addr": "127.0.0.1",
  "log_json": true,
  "log_level": "info",
  "log_rotate_max_files": 10,
  "log_rotate_duration": "24h",
  "log_file": "/data/consul/log/",
  "retry_join": [
      "[TSE Consul Node1 Address]",
      "[TSE Consul Node2 Address]",
      "[TSE Consul Node3 Address]"
  ]
}
```

注意

Consul Agent 支持本地模式和局域网模式两种部署模式：

- 本地模式：本地模式请指定 client_addr 为127.0.0.1，需要在每台 CVM 实例部署 Agent。
- 局域网模式：局域网模式请指定 client_addr 为0.0.0.0，只需指定几台 CVM 实例部署 Agent，部署后同一局域网内的 Agent 即可互相连通。

3. 进入 Consul 安装包所在的目录，执行如下命令解压 Consul 安装包。

```
unzip consul_1.8.6_linux_amd64.zip
```

4. 执行如下命令启动 Consul Agent。

```
./consul agent --config-dir=/data/consul/config
```

步骤2：应用接入

1. 下载 Github 的 Demo 源码 到本地并解压。
2. 打包 demo 源码成 jar 包。在 `tse-simple-demo-main` 源码根目录下，打开终端窗口，执行 `mvn clean package` 命令，对项目进行打包编译。编译成功后，可以在如下目录看到生成如下表所示的2个 Consul Jar 包。

软件包所在目录	软件包名称	说明
\tse-simple-demo-main\tse-consul-provider-demo\target	tse-consul-provider-demo-1.0-SNAPSHOT.jar	服务生产者
\tse-simple-demo-main\tse-consul-consumer-demo\target	tse-consul-consumer-demo-1.0-SNAPSHOT.jar	服务消费者

3. 将编译好的 jar 包上传至云服务器，详细操作请参见 如何将本地文件拷贝到云服务器 。
4. 登录云服务器，进入到刚刚上传 jar 文件所在的目录，可看到文件已上传到云服务器。

```
-rw-r--r-- 1 root root  43300221 Aug 23 15:10 tse-consul-consumer-demo-1.0-SNAPSHOT.jar
-rw-r--r-- 1 root root  43300061 Aug 23 15:09 tse-consul-provider-demo-1.0-SNAPSHOT.jar
```

5. 执行如下命令指定注册中心地址参数并运行该应用。

说明

> `[TSE Consul Client Agent IP]` 为 步骤1：准备配置 中的 json 文件中的 `client_addr` 参数值。

```
nohup java -Dspring.cloud.consul.host=[TSE Consul Client Agent IP] -jar [jar包名称] &
```

步骤3：验证服务注册成功

1. 登录 TSF 控制台，在左侧导航栏选择 Consul，进入引擎实例列表页。
2. 单击目标实例的 ID，进入基本信息页面。
3. 在页面上方选择**服务管理**页签，出现如下页面代表服务注册成功。单击服务名称或者操作栏的**查看实例列表**可以查看服务的详细信息，如ID、地址、端口等信息。

4. 登录云服务器，执行如下命令，调用 consumer 接口访问 provider 服务。

```
curl http://localhost:18084/ping-provider/str
```

返回结果如下：

```
[root@VM-49-34-centos home]# curl http://localhost:18084/ping-provider/str
request param: str, response from consul-demo-provider[root@VM-49-34-centos home]#
```

解决 Spring Cloud 下测试环境路由问题

最近更新时间：2024-01-22 16:19:31

前言

Spring Cloud Tencent 新增了 spring-cloud-tencent-plugin-starts 模块，在此模块下实现不同业务场景的解决方案。现阶段我们主要聚焦在精细化流量治理能力场景化方案上，并按照开发流程拆分为三个阶段：

1. 开发测试阶段的多测试环境场景。
2. 发布阶段的金丝雀发布、蓝绿发布、全链路灰度等。
3. 生产运行阶段的单元化、AB 测试等。
   本文档展示开发测试阶段的多测试环境场景实战，详细介绍 Spring Cloud Tencent 实现多测试环境场景的方案。

基础知识

测试环境路由

在实际的开发过程中，一个微服务架构系统下的不同微服务可能是由多个团队进行开发与维护的，每个团队只需关注所属的一个或多个微服务，而各个团队维护的微服务之间可能存在相互调用关系。如果一个团队在开发其所属的微服务，调试的时候需要验证完整的微服务调用链路。此时需要依赖其他团队的微服务。在部署开发联调环境时，会遇到以下问题：

- 如果所有团队都使用同一套开发联调环境，那么一个团队的测试微服务实例无法正常运行时，会影响其他依赖该微服务的应用也无法正常运行。
- 如果每个团队有单独的一套开发联调环境，那么每个团队不仅需要维护自己环境的微服务应用，还需要维护其他团队环境的自身所属微服务应用，效率大大降低。同时，每个团队都需要部署完整的一套微服务架构应用，成本也随着团队数的增加而大大上升。

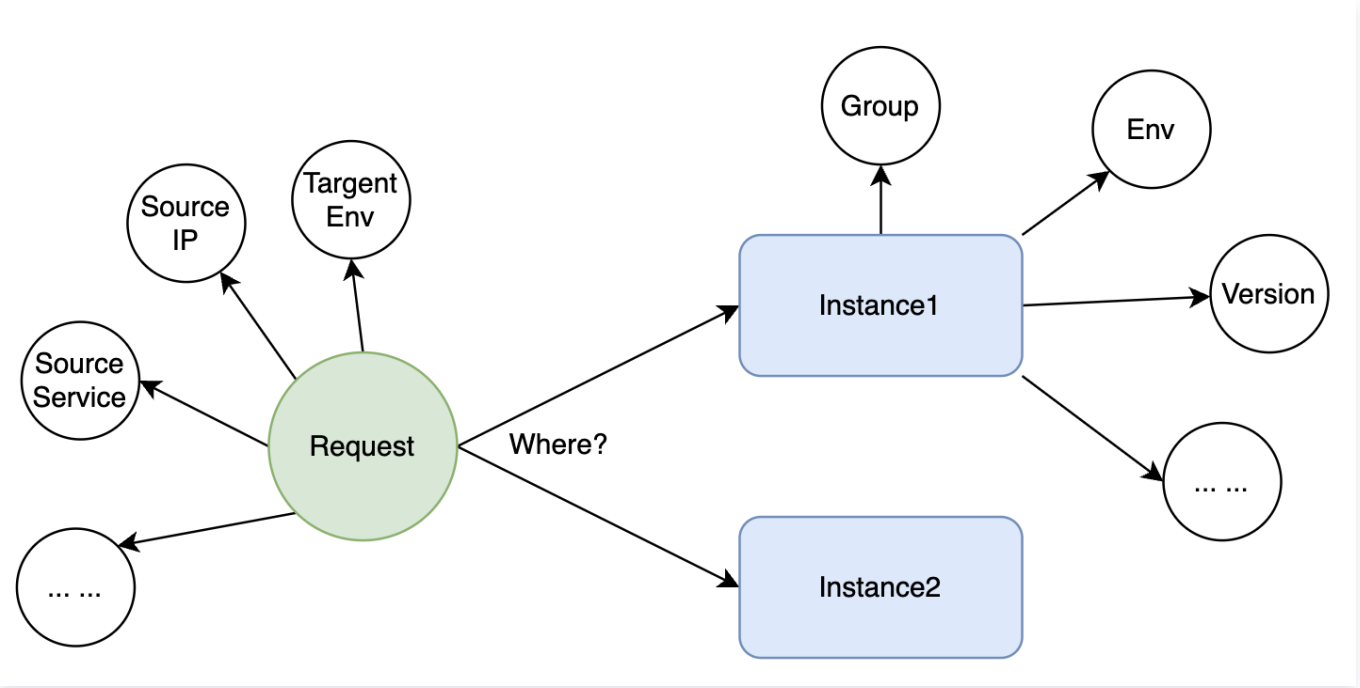
此时可以使用测试环境路由的架构来帮助部署一套运维简单且成本较低的开发联调环境。测试环境路由是一种基于服务路由的环境治理策略，核心是维护一个稳定的基线环境作为基础环境，测试环境仅需要部署需要变更的微服务。多测试环境有两个基础概念，如下所示：

1. 基线环境（Baseline Environment）：完整稳定的基础环境，是作为同类型下其他环境流量通路的一个兜底可用环境，用户应该尽量保证基线环境的完整性、稳定性。
2. 测试环境（Feature Environment）：一种临时环境，仅可能为开发/测试环境类型，测试环境不需要部署全链路完整的服务，而是仅部署本次有变更的服务，其他服务通过服务路由的方式复用基线环境服务资源。

部署完成多测试环境后，开发者可以通过一定的路由规则方式，将测试请求打到不同的测试环境，如果测试环境没有相应的微服务处理链路上的请求，那么会降级到基线环境处理。因此，开发者需要将开发新测试的微服务部署到对应的测试环境，而不需要更新或不属于开发者管理的微服务则复用基线环境的服务，完成对应测试环境的测试。
虽然测试环境路由是一个相对成熟的开发测试环境解决方案，但是能够开箱即用的生产开发框架却不多，往往需要开发者二次开发相应的功能。因此需要一个相对完善的解决方案来帮助实现测试环境路由，简化开发难度并提升开发效率。

服务路由

服务路由抽象出最简化的模型如下图所示，解决的是 “哪些请求转发到哪些实例” 的问题。细化来看，包含三个问题：1. 如何精确标识请求？2. 如何精确标识实例？3. 如何转发？



在流量的微观世界里，统一通过标签（属性）来标识一个实体，例如请求有来源调用服务、目标环境标签等，服务实例则有版本号、实例分组、环境分组等标签。服务路由则是将满足标签匹配条件的请求转发到满足匹配条件的服务实例。所以服务路由的模型可拆解出如下的专业术语：

- 服务实例染色（为服务实例设置标签信息）
- 流量染色（为请求设置标签信息）
- 服务路由（根据路由策略，把请求转发到目标实例）

服务实例标签如何传递到调用方

服务实例注册到注册中心时，会带上标签信息。服务调用方从注册中心获取到服务实例信息就包含了实例的标签信息。

标签全链路透传

有一类请求标签数据需要在业务响应链路上一直传递，例如全链路追踪里的 TraceId 、测试环境路由的 FeatureEnv 标签等。

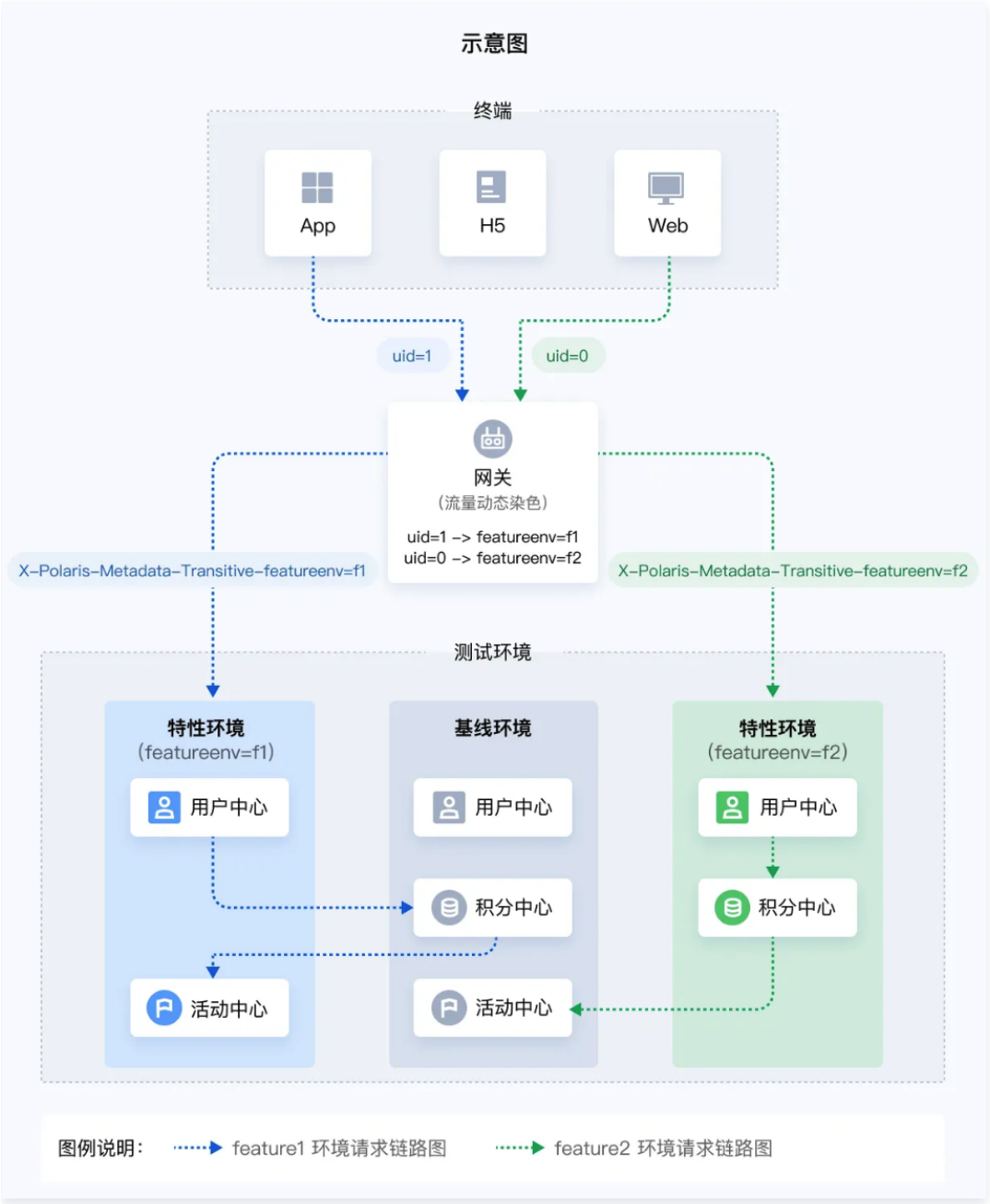
服务路由和负载均衡的区别

服务路由和负载均衡都是解决选择服务实例的问题。区别在于服务路由是从全量的服务实例中挑选出一批满足路由规则的服务实例，而负载均衡则是从路由匹配之后的服务实例列表中挑选出一个适合处理请求的实例。

测试环境路由实现原理

方案总览

测试环境路由的样例实现以下图为例，一共有两个测试环境以及一个基线环境。流量从端到端会依次经过以下组件：App > 网关 > 用户中心 > 积分中心 > 活动中心。



根据上一节服务路由章节所述，为了达到测试环境路由的能力，开发工作需要做三件事情：

1. 服务实例染色（标识实例属于哪个测试环境）
2. 流量染色（标识请求应该被转发到哪个测试环境）
3. 服务路由
   a. 网关根据请求的目标测试环境标签转发到对应的目标测试环境的用户中心。
   b. 服务调用时，优先转发到同测试环境下的目标服务实例，如果同测试环境下没有服务实例则转发到基线环境。

下文将会详细介绍服务实例染色、流量染色和服务路由这三部分的原理。

服务实例染色

在多测试环境的场景中，需要对每个测试环境部署的实例进行区分，因此需要在实例上打 `<featureenv=测试环境名>` 的标签。Spring Cloud Tencent 一共支持三种服务实例染色方式。

方式一：配置文件

在 Spring Boot 的 application.yml 配置文件里配置以下内容即可实现染色：

```
spring:
  cloud:
    tencent:
      metadata:
        content:
          idc: shanghai
          env: f1
```

Spring Cloud Tencent 应用在启动时，读取配置文件并解析出 idc=shanghai 和 env=f1 标签信息。
如果以上配置文件放在项目源码里，要实现不同的实例具有不同的标签值则需要打不同包。可以通过以下两种方式实现同一个运行包设置不同的标签值：

1. 通过 -D 启动参数覆盖，例如 -Dspring.cloud.tencent.metadata.content.idc=guangzhou。
2. 通过 Spring Boot 标准方式，把 application.yml 外挂本地磁盘上。

方式二：环境变量

环境变量在容器场景下非常方便，Spring Cloud Tencent 约定了前缀为 SCT_METADATA_CONTENT_ 的环境变量为实例的标签信息，例如：

- SCT_METADATA_CONTENT_IDC=shanghai
- SCT_METADATA_CONTENT_ENV=f1
  Spring Cloud Tencent 应用在启动时，自动会读取环境变量并解析出 IDC=shanghai 和 ENV=f1 标签信息。

方式三：自定义实现 SPI

前面两种方式为 Spring Cloud Tencent 内置的方式，但是不一定符合每个生产项目的规范，因此 Spring Cloud Tencent 还提供了一种允许开发者自定义标签 Provider 的方式。例如以下两种实践场景：

1. 把实例标签放到机器上的某一个配置文件里，例如 /etc/metadata。
2. 应用启动时，调用公司的 CMDB 接口获取元信息。
   这种场景下，只要实现 InstanceMetadataProvider SPI 扩展即可。

流量染色

流量染色即为每个请求打上目标测试环境标签，路由转发时根据请求标签匹配目标服务实例。而流量染色可以分为以下几种方式：

方式一：静态染色

2.2 小节介绍了可以为服务实例设置一系列的标签信息，例如 idc=shanghai、env=f1 等。在有些场景下，期望所有经过当前实例的请求都带上当前实例的标签信息。例如经过 env=f1 的实例的请求都携带 env=f1 的标签信息。
服务实例染色有三种方式，对应的定义哪些标签需要作为请求标签透传到链路上也有三种方式，核心思想就是定义需要全链路传递的标签键值对的键列表。

1. 通过配置文件的 spring.cloud.tencent.metadata.content.transitive=["idc", "env"] 配置项指定。
2. 通过 SCT_METADATA_CONTENT_TRANSITIVE=IDC,ENV 环境变量指定。
3. 通过实现 InstanceMetadataProvider#getTransitiveMetadataKeys() 方法指定。

方式二：动态染色

静态染色是把服务实例的某些标签作为请求标签，服务实例标签相对静态，应用启动后初始化一次之后就不再变更。但是在实际的应用场景下，不同的请求往往需要设置不同的标签信息。此时则需要通过动态染色的能力。
为请求动态染色也非常简单，只需增加以 X-Polaris-Metadata-Transitive- 为前缀的 HTTP 请求头即可，例如：X-Polaris-Metadata-Transitive-featureenv=f1。这样 featureenv=f1 就能够作为请求标签在链路上透传。

方式三：网关流量染色

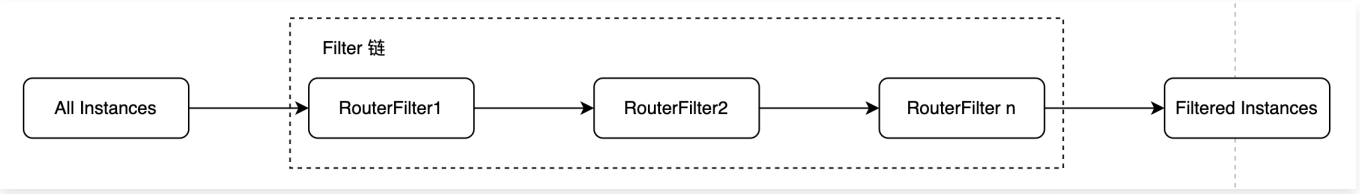
网关常常作为流量的入口或者中转站。经过网关的请求，可以根据某些染色规则为请求增加标签信息。例如满足请求参数 uid=1000 请求打上 featureenv=f1 标签。
网关流量染色是非常实用的能力，在 Spring Cloud Tencent 里实现了非常灵活基于染色规则的 Spring Cloud Gateway 染色插件。例如以下染色规则可以实现为 uid=1000 的请求打上 featureenv=f1 标签、uid=1001 的请求打上 featureenv=f2 标签。更详细的染色规则，可以参考文档。

```
{
    "rules":[
        {
            "conditions":[
                {
                    "key":"${http.query.uid}",
                    "values":["1000"],
                    "operation":"EQUALS"
                }
            ],
            "labels":[
                {
                    "key":"featureenv",
                    "value":"f1"
                }
            ]
        },
        {
            "conditions":[
                {
                    "key":"${http.query.uid}",
                    "values":["1001"],
                    "operation":"EQUALS"
                }
            ],
            "labels":[
                {
                    "key":"featureenv",
                    "value":"f2"
                }
            ]
        }
    ]
}
```

同时 Spring Cloud Tencent 也预留了 TrafficStainer SPI，用户可以实现自定义流量染色插件。

Spring Cloud Tencent 路由功能原理

北极星提供了非常完善的服务治理能力，上层的服务框架基于北极星原生 SDK 就能快速实现强大的服务治理能力。Spring Cloud Tencent 就是在北极星的基础上实现了服务路由能力。
北极星服务路由实现原理并不复杂，如下图所示，从注册中心获取到所有实例信息，再经过一系列的 RouterFilter 插件过滤出满足条件的实例集合。



在多测试环境场景下主要用到了 MetadataRouter（元数据路由）插件，此插件核心能力是**根据请求的标签完全匹配服务实例的标签**。
例如请求有两个标签 key1=value1和 key2=value2，MetadataRouter 则会筛选出所有实例中包含同时满足 key1=value1 和 key2=value2 的服务实例。在多测试环境场景下，Spring Cloud Tencent 缺省使用 featureenv 标签，通过 featureenv 标签筛选出属于同一个测试环境的服务实例。

Spring Cloud Tencent 服务路由原理

Spring Cloud Tencent 实现路由核心分成两个部分：

1. 扩展 RestTemplate 、 Feign、SCG 获取请求的标签信息并塞到 RouterContext（路由信息上下文）里。
2. 扩展 Spring Cloud 负载均衡组件（Hoxton 版本之前为 Ribbon，2020版本之后为 Spring Cloud LoadBalancer），在扩展的实现里调用北极星的服务路由 API 实现服务实例过滤。
   扩展部分逻辑较为复杂，感兴趣的读者可以参考 spring-cloud-starter-tencent-polaris-router 模块源码。

测试环境路由用户操作指引

在上一节中详细介绍了测试环境路由的实现原理，这一节则详细介绍站在用户的视角需要操作的内容。
通过 Spring Cloud Tencent 实现流量的测试环境路由非常简单，核心包含三步：
1. 服务增加测试环境路由插件依赖
2. 部署的实例打上环境标签
3. 为请求流量打上环境标签
   完成以上三个步骤即可。

步骤1：添加测试环境路由插件依赖

Spring Cloud Tencent 中的 spring-cloud-tencent-featureenv-plugin 模块闭环了测试环境路由全部能力，所有服务只需要添加该依赖即可引入测试环境路由能力。

步骤2：服务实例打上环境标签

spring-cloud-tencent-featureenv-plugin 默认以 featureenv 标签作为匹配标签，用户也可以通过系统内置的 system-feature-env-router-label=custom_feature_env_key 标签来指定测试环境路由使用的标签键。以下三种方式以默认的 featureenv 作为示例。

方式一：配置文件

在服务实例的配置文件中添加配置，如在 bootstrap.yml添加如下所示即可：

```
spring:
  cloud:
    tencent:
      metadata:
        content:
          featureenv: f1   # f1 替换为测试环境名称
```

方式二：环境变量

在服务实例所在的操作系统中添加环境变量也可进行打标，例如：SCT_METADATA_CONTENT_featureenv=f1 。

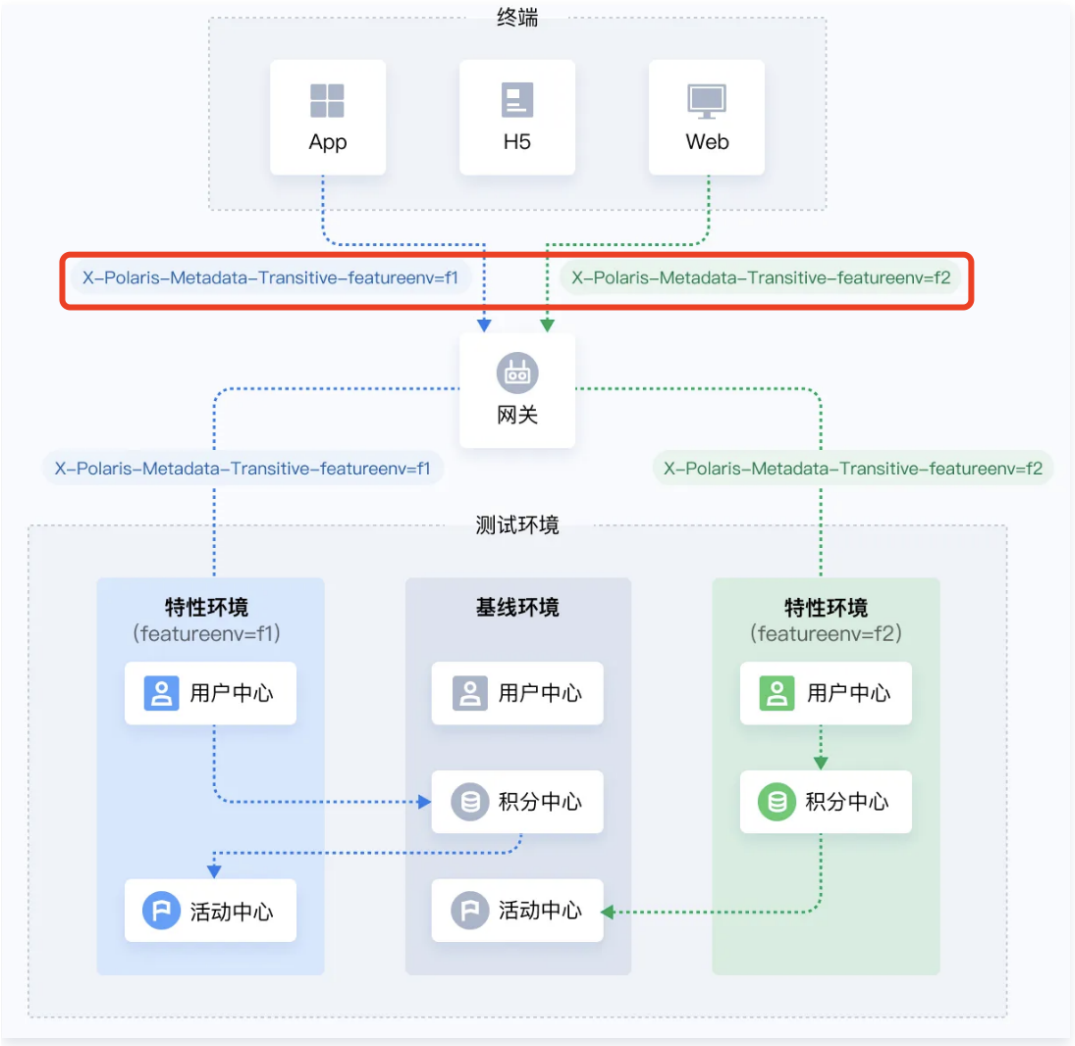
方式三：SPI 方式

自定义实现 InstanceMetadataProvider#getMetadata() 方法的返回值里包含 featureenv 即可。
基线环境标签值
注意，基线环境部署的服务实例不需要设置 featureenv 标签，表明其不属于任何测试环境，才可在请求没有匹配到对应测试环境的时候，匹配到基线环境。

流量染色

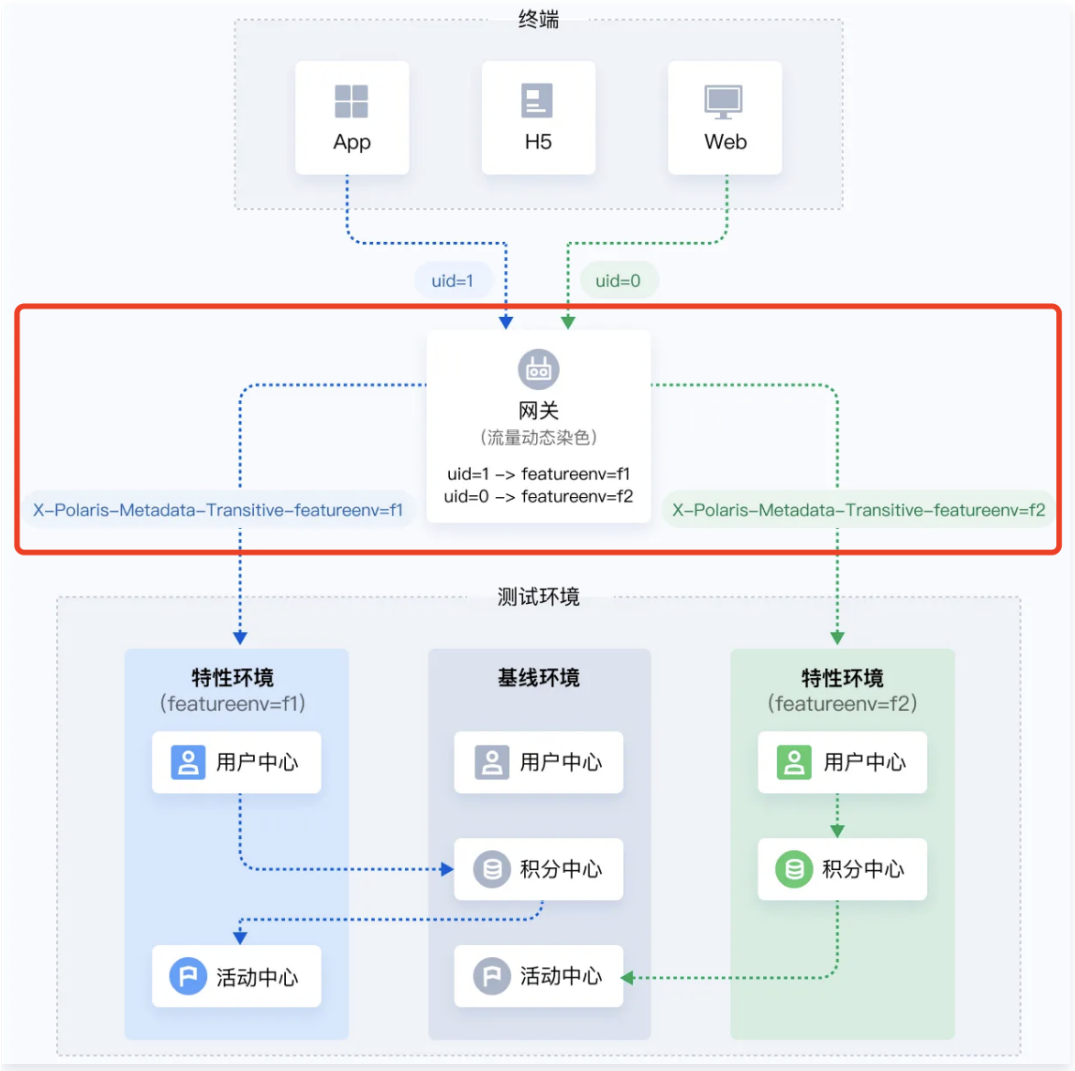
方式一：客户端染色（推荐）

如下图所示，在客户端发出的 HTTP 请求里，新增 X-Polaris-Metadata-Transitive-featureenv=f1 请求头即可实现染色。该方式是让开发者在请求创建的时候根据业务逻辑进行流量染色。



方式二：网关动态染色（推荐）

动态染色是开发者配置一定的染色规则，让流量经过网关时自动染色，使用起来相当方便。例如把 uid=1 用户的请求都转发到 f1 环境，把 uid=0 用户的请求都转发到 f2 环境。只需要配置一条染色规则即可实现。



Spring Cloud Tencent 通过实现 Spring Cloud Gateway 的 GlobalFilter 来实现流量染色插件，开发者只需要添加 spring-cloud-tencent-gateway-plugin 依赖，并在配置文件中打开染色插件开关（spring.cloud.tencent.plugin.scg.staining.enabled=true）即可引入流量染色能力。

方式三：网关静态染色

往请求中加入固定的 Header 是网关最常见的插件，如下图所示。可以在每个环境部署一个网关，所有经过网关的请求都增加 X-Polaris-Metadata-Transitive-featureenv=f1 请求头即可。此种方式需要每个环境部署网关，成本高，所以使用频率相对较低。

完成以上操作步骤即可实现测试环境路由，您可运行 Spring Cloud Tencent 下 polaris-router-featureenv-example 完整体验。

总结

测试环境路由在微服务架构系统的开发阶段是非常实用的功能，能够大大降低测试环境的维护成本、资源成本，同时能够极大的提高研发效率。通过操作指引的章节可以看出通过 Spring Cloud Tencent 实现测试环境路由非常简单的，只需要部署的服务实例增加相应的环境标签以及在请求头中增加一个标签即可。

业界常见的测试环境路由实现方案往往需要下发路由规则给链路上的服务，从而实现路由能力。但是通过北极星的元数据路由能力，整个方案里无需下发任何路由规则，只需要在实例设置相应的标签信息即可，操作成本非常低。

如果项目刚好使用 Spring Cloud Gateway 作为网关，那么集成 Spring Cloud Tencent 里的网关染色插件能够进一步降低流量染色成本，客户端无需做任何事情，只需要配置网关染色规则即可实现流量染色。

目前 Spring Cloud Tencent 主要实现了微服务之间调用流量的测试环境路由能力，不涉及消息队列、任务调度的测试环境路由能力。

Nacos 实践教程
Nacos 多活容灾

最近更新时间：2023-05-23 17:25:26

操作场景

当您在生产环境中已经使用了自建的 Nacos 集群，并希望在腾讯云中部署灾备集群时；或者对集群的高可用、稳定性以及网络上的跨地域延迟有要求时，可以参考本指引配置多活容灾与就近访问方案，以提供跨云、跨 IDC 机房之间的应用访问，或者腾讯云内的跨集群之间的应用访问。

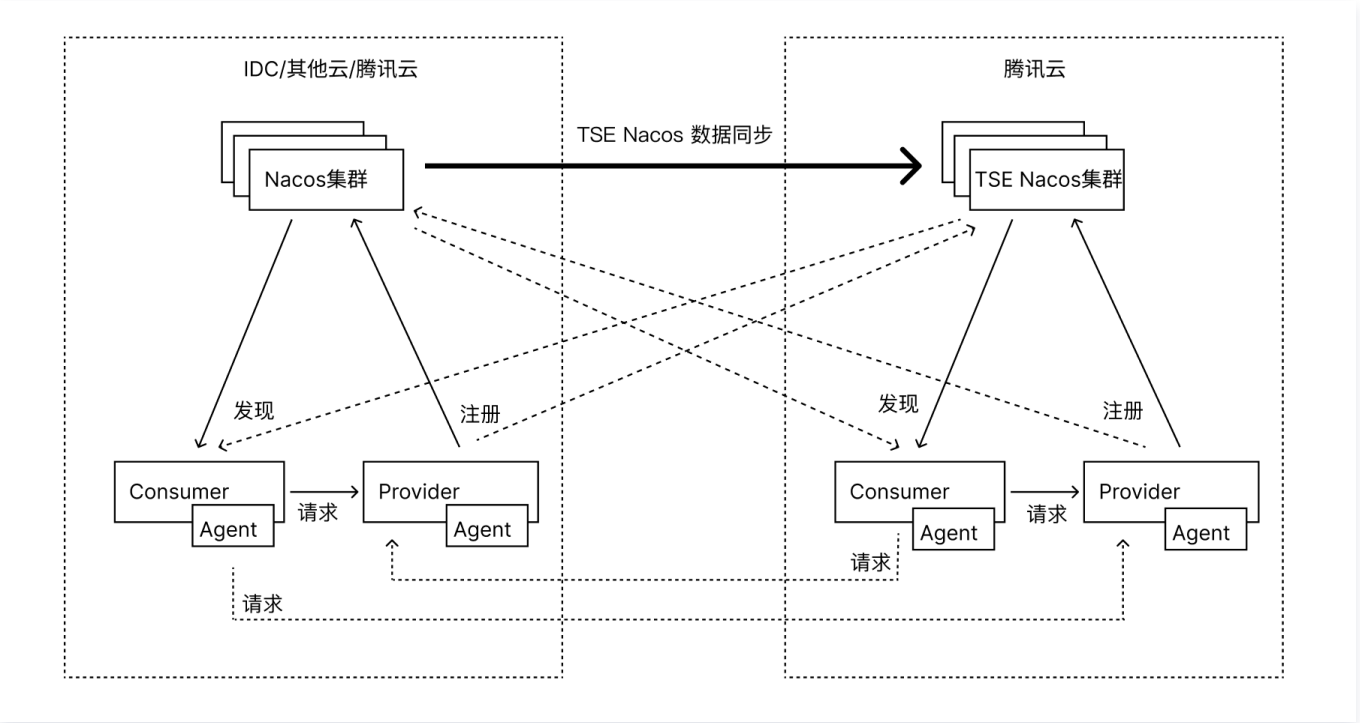
操作原理

Nacos 的核心功能为服务注册发现与配置管理，故 Nacos 多活容灾方案需要包括数据同步与服务注册发现同步：

- 数据同步：Nacos 的配置数据、用户信息、命名空间数据、访问控制等信息在数据库中持久化存储，因此首先需要使用 TSE Nacos 的数据同步功能实现源数据库到 TSE Nacos 数据库的全量迁移与实时增量同步，以保证每个 Nacos 集群都会有全量配置数据。
- 服务注册发现同步：采用 Java agent 的形式，在不改变业务代码的情况下，通过字节码增强的方式，实现服务的双注册和发现，以此提供服务同时在不同集群间路由的能力。

例如：当您在自建 IDC 机房和腾讯云分别部署了一整套应用和 Nacos 集群，可以通过本方案配置多活容灾与就近访问。即 IDC 机房内的 consumer 应用调用 provider 应用时，优先访问 IDC 内的 provider 应用，如果找不到该应用，则访问腾讯云中的 provider 应用。

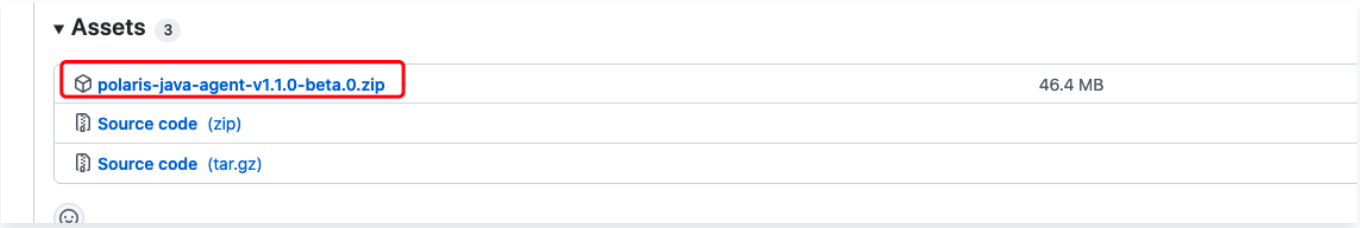
Nacos 多活部署架构图



前提条件

1. 登录 TSE 控制台，新建一个 nacos 实例，并获取客户端访问地址。详细操作请参见 引擎管理 。

2. 访问 Github 地址，下载 Polaris Java Agent（支持 Nacos 双注册双发现）。



操作步骤

步骤1：开启数据同步

通过 TSE Nacos 的配置数据同步能力，将源 Nacos 数据库导入与同步至 TSE Nacos 集群。请参考 Nacos 数据同步 。

步骤2：配置自建集群中的服务双注册发现

1. 将提前下载好的 polaris java agent zip 包上传到您的服务所部署的路径下并解压，确认该位置可以被正常访问。
2. 将原本注册在自建集群中的服务依次重新部署并双注册至 TSE Nacos 集群，部署时需要添加如下参数：

```
java
-javaagent:/*/polaris-java-agent-v*/polaris-agent-core-bootstrap.jar
-Dplugins.enable=nacos-all-plugin
-Dnacos.cluster.name=*
-Dother.nacos.server.addr=**.**.**.**
-Drouter.nearby.level=nacos_cluster -jar xx.jar
```

> **说明**
> 四处 `*` 分别代表 polaris java agent 所在的路径、版本号、集群名称、另一个 Nacos 集群的客户端访问地址。

Java Agent 参数配置

polaris java agent 提供以下配置项，所有的配置项通过系统变量（-D参数）的方式进行配置。

配置项	描述	必填	可选值	默认值
plugins.enable	选择需要加载的插件	是	nacos-all-plugin	nacos-all-plugin
nacos.cluster.name	Nacos 集群名称/标签，唯一标识即可	是	唯一标识即可	无

other.nacos.server.addr	另一个 Nacos 集群的访问地址	是	无	无
router.nearby.level	就近路由级别	否	null：不开启就近路由，采用客户端配置的路由策略，默认为轮询策略。 nacos_cluster：开启集群级别的就近路由，优先路由至相同 nacos.cluster.name 下的服务。	null

3. 部署成功后，在 TSE Nacos 原生控制台的服务管理页面可以看到注册的服务。Nacos 原生控制台的访问方式请参见 访问控制 。此时服务在自建 Nacos 集群和 TSE Nacos 集群中均进行了注册。
4. 观察自建的 Nacos 集群和 TSE 的 Nacos 集群，依次验证下注册、发现、反注册，看是否均符合预期。待所有服务均重新部署完毕后，在自建 Nacos 集群和 TSE Nacos 集群的控制台均能看到所有服务以及其下的实例信息。

步骤3：配置 TSE Nacos 集群中的服务双注册发现

重复 步骤2 中的操作，将原本注册至 TSE Nacos 集群中的服务依次重新部署，并双注册至自建 Nacos 集群。请注意将 Java Agent 参数配置项中的 `nacos.cluster.name` 与 `other.nacos.server.addr` 修改为合适的值。
完成以上步骤后，您已经成功配置了自建 Nacos 集群与 TSE Nacos 集群之间的多活方案，建议您保持 TSE Nacos 数据同步任务的持续运行，并保持在自建 Nacos 集群更新配置数据，以保证集群之间的数据一致。

Nacos 客户端本地缓存及推空保护开启

最近更新时间：2025-04-17 16:00:54

操作场景

- 本地缓存：Nacos Server 因网络抖动、主动升级或意外宕机导致短暂失联（例如秒级到分钟级不可用）。客户端需在服务端不可用期间继续工作，避免因服务发现中断导致业务调用失败。
- 推空保护：运维人员误操作（例如删除服务）、Nacos Server 异常（例如数据损坏）或恶意攻击导致服务端推送空服务列表。客户端需避免信任异常推送，防止服务列表被清空。

操作原理

客户端本地缓存

缓存机制

Nacos 客户端有两种缓存机制，两种缓存机制默认开启，并且不支持关闭。

- 内存缓存。
- 文件缓存：默认保存在 {user.home}/nacos/naming/{namespace}/failover 。

生效逻辑

客户端本地缓存生效逻辑：

- 启动时，先从服务端查询数据，如果服务端没有响应，从文件缓存读取数据。
- 运行中，先从内存缓存查询数据，如果内存缓存没有，从服务端查询数据，并且更新内存缓存和文件缓存。
- 缓存更新机制：运行中会有异常任务监听服务端变更，更新内存缓存和文件缓存。

保证启动时文件缓存有效的操作方法

不同的应用部署方式，如何保证启动时文件缓存有效?

- CVM 部署：不要清理缓存文件
- 容器部署：POD 挂载 volume，并挂载目录 {user.home}/nacos，可确保 POD 重启后，容灾文件依旧存在。

测试结果

推空保护

Nacos 的推空保护是一种客户端容错机制，旨在防止服务端因异常情况（例如误操作、数据损坏）向客户端推送空服务列表，导致客户端误删本地缓存并引发服务调用中断。其核心原理是通过客户端逻辑拦截异常推送，结合本地缓存实现服务列表的“故障隔离”。

操作步骤

客户端本地缓存

本地缓存默认生效

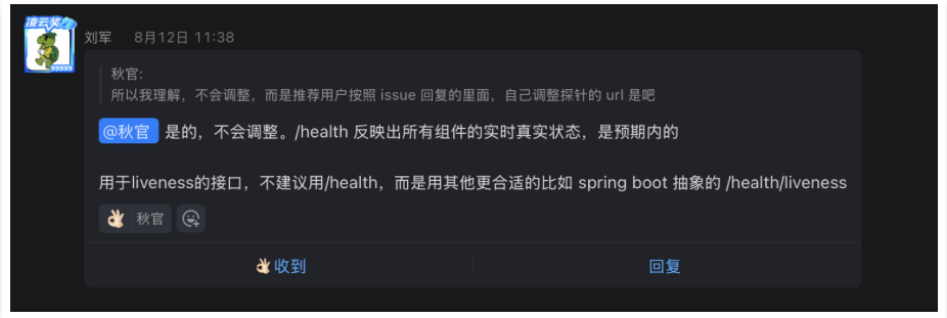
客户端本地缓存默认开启，并且不支持关闭。

本地缓存不生效场景

- **场景1**：Spring Boot 开启健康检查接口，k8s 存活探针配置 Spring Boot 健康检查接口。可能导致 k8s Pod 重启，缓存失效。
- **场景2**：通过 Nacos OpenAPI 访问 Nacos 服务端，例如：APISIX 服务发现插件使用 Lua 脚本调用 Nacos OpenAPI。

场景1实践教程

- 在 Spring Boot 2.3.x 之前，Actuator 提供 /actuator/health 接口。
  - /actuator/health 接口：反映 Spring Boot 依赖组件的实时监控状态。
  - Spring Boot 没有建议在 k8s readiness 和 liveness 探针中配置 /actuator/health 接口。
- 从 Spring Boot 2.3.x 版本开始，Actuator 针对 k8s 场景新增两个接口。
  - /actuator/health/readiness 接口：用于配置 readiness 探针。
  - /actuator/health/liveness 接口：用于配置 liveness 探针。
- **其他说明：**
  - spring cloud alibaba 开源项目相关 issue： https://github.com/alibaba/spring-cloud-alibaba/issues/3535
  - spring cloud alibaba 开源项目成员相关建议：



场景2实践教程

调用 OpenAPI 访问 Nacos 服务端时，未使用 Nacos client SDK，且缺少本地缓存机制。建议如下：

- 在查询数据后立即使用，可参考 Nacos client sdk 实现本地缓存，先更新内存缓存，再更新文件缓存。
- 将数据写入其他组件，如果从服务端查询数据失败或者数据为空，不要对其他组件的数据进行更新。

推空保护

建议您开启 Nacos 客户端的推空保护开关（Nacos Java Client 1.4.1及以上版本支持客户端推空保护功能）。

1. 使用 Nacos-client

```
Properties properties = new Properties();
properties.put(PropertyKeyConst.SERVER_ADDR, "${Nacos 客户端负载均衡 IP}:8848");
properties.put(PropertyKeyConst.NAMING_PUSH_EMPTY_PROTECTION, "true");
NamingService naming = NamingFactory.createNamingService(properties);
```

2. 使用 spring-cloud-alibaba

```
spring.cloud.nacos.discovery.namingPushEmptyProtection=true
```

3. 使用 Dubbo

```
dubbo.registry.address=nacos://${Nacos 客户端负载均衡 IP}:8848?namingPushEmptyProtection=true
```