ALIBABA CLOUD

阿里云

Serverless 应用引擎

应用开发

文档版本：20220712

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.使用Spring Cloud开发应用

1.1. Spring Cloud开发概述

SAE支持原生Spring Cloud微服务框架，在该框架下开发的应用只需添加服务依赖和修改注册中心配置，便可获取SAE企业级的应用托管、应用治理、监控报警和应用诊断等能力，实现零代码工作量的应用迁移。

为什么使用Spring Cloud

Spring Cloud提供了简化应用开发的一系列标准和规范。该标准和规范包含服务发现、负载均衡、熔断、配置管理、消息事件驱动、消息总线等，同时Spring Cloud在该规范的基础上，提供了服务网关、全链路跟踪、安全、分布式任务调度和分布式任务协调等功能的实现机制。

目前业界流行的Spring Cloud组件包括Spring Cloud Netflix、Spring Cloud Consul和Spring Cloud Alibaba等。

如果您已经使用Spring Cloud Netflix、Spring Cloud Consul等Spring Cloud组件开发应用，该应用可以直接部署到SAE上，获得应用托管能力。不需要修改任何代码，便可直接使用SAE所提供的高级监控功能，以及实现全链路跟踪、监控报警和应用诊断等监控功能。

如果您的Spring Cloud应用还想使用SAE中更多的服务治理相关功能，那么需要将您的Spring Cloud组件替换为Spring Cloud Alibaba中的组件或增加Spring Cloud Alibaba组件。

兼容性说明

SAE目前支持Spring Cloud Greenwich、Spring Cloud Finchley和Spring Cloud Edgware三个版本。Spring Cloud、Spring Boot和Spring Cloud Alibaba及各组件的版本对应关系请参见版本配套关系说明。

Spring Cloud功能、开源实现及SAE兼容性如下表所示。

Spring Cloud功能		开源实现	SAE兼容性
通用功能	服务注册与发现	• Netflix Eureka • Consul Discovery	兼容且提供替换组件
	负载均衡	Netflix Ribbon	兼容
	服务调用	• Feign • RestTemplate	兼容
配置管理		• Config Server • Consul Config	兼容且提供替换组件
服务网关		• Spring Cloud Gateway • Netflix Zuul	兼容
链路跟踪		Spring Cloud Sleuth	兼容且提供替换组件

Spring Cloud功能	开源实现	SAE兼容性
消息驱动Spring Cloud Stream	• RabbitMQ binder • Kafka binder	兼容且提供替换组件
消息总线Spring Cloud Bus	• RabbitMQ • Kafka	兼容且提供替换组件
安全	Spring Cloud Security	兼容
分布式任务调度	Spring Cloud Task	兼容
分布式协调	Spring Cloud Cluster	兼容

版本配套关系说明

Spring Cloud、Spring Boot和Spring Cloud Alibaba及SAE提供的正式商用组件的版本配套关系如下表所示。

Spring Cloud	Spring Boot	Spring Cloud Alibaba
ANS	ACM	SchedulerX
Greenwich	2.1.x	2.1.1.RELEASE
Finchley	2.0.x	2.0.1.RELEASE
Edgware	1.5.x	1.5.1.RELEASE
Hoxton	2.2.x	2.2.x

说明 Spring Cloud Alibaba Nacos Discovery和Spring Cloud Alibaba Nacos Config分别是ANS和ACM对应的开源组件。

1.2. 使用Spring Cloud开发微服务应用并部署至SAE

本文以包含服务提供者和服务消费者的Spring Cloud应用为例，让您快速体验如何在本地开发、调试Spring Cloud应用并部署到SAE中，实现应用的服务注册与发现，以及消费者对提供者的调用。

背景信息

- 如果您对Spring Cloud很陌生，仅了解Spring和Maven基础知识，那么阅读本文后，您将掌握如何通过Spring Cloud Alibaba Nacos Discovery实现Spring Cloud应用的服务注册与发现，以及实现消费者对提供者的调用。
- 如果您熟悉Spring Cloud中的Eureka、Consul和ZooKeeper等服务注册组件，但未使用过Spring Cloud Alibaba的服务注册组件Nacos Discovery，那么您仅需将服务注册组件的服务依赖关系和服务配置替换成Spring Cloud Alibaba Nacos Discovery，无需修改任何代码。

Spring Cloud Alibaba Nacos Discovery同样实现了Spring Cloud Registry的标准接口与规范，与您之前使用Spring Cloud接入服务注册与发现的方式基本一致。

- 如果您熟悉如何使用开源版本的Spring Cloud Alibaba Nacos Discovery实现Spring Cloud应用的服务注册与发现，那么您可以将应用直接部署到SAE，即可使用到SAE提供的商业版服务注册与发现的能力。更多信息，请参见应用部署概述。

为什么使用SAE服务注册中心

SAE服务注册中心提供了开源Nacos Server的商用版本，使用开源版本 `Spring Cloud Alibaba Nacos Discovery` 开发的应用可以直接使用SAE提供的商业版服务注册中心。

SAE服务注册中心与Nacos、Eureka和Consul相比，具有以下优势：

- 共享组件，节省了部署、运维Nacos、Eureka或Consul的成本。
- 在服务注册和发现的调用中都进行了链路加密，保护您的服务，无需再担心服务被未授权的应用发现。
- SAE服务注册中心与SAE其他组件紧密结合，为您提供一整套的微服务解决方案，包括环境隔离、灰度发布等。

您在SAE部署应用时，SAE服务注册中心以高优先级自动设置Nacos Server服务端地址和服务端口，以及命名空间、AccessKey、Context-path等信息，无需进行任何额外的配置。

当您的微服务应用较多时，注册中心按推荐程度由高到低依次排序如下：

- 商业版的服务注册中心（MSE）
- 自建服务注册中心
- SAE内置服务注册中心

如果您选择商业版的服务注册中心，即使用MSE的Nacos作为服务注册中心，具体操作，请参见使用MSE的Nacos注册中心。

如果您选择使用自建Nacos作为服务注册中心，具体操作，请参见自建Nacos服务注册中心。您需要确认以下内容：

- 请确保SAE的网络与自建Nacos的网络互通。
- 在部署应用时建议使用镜像方式或者JAR包方式，并配置启动参数 `-Dnacos.use.endpoint.parsing.rule=false` 和 `-Dnacos.use.cloud.namespace.parsing=false` 。

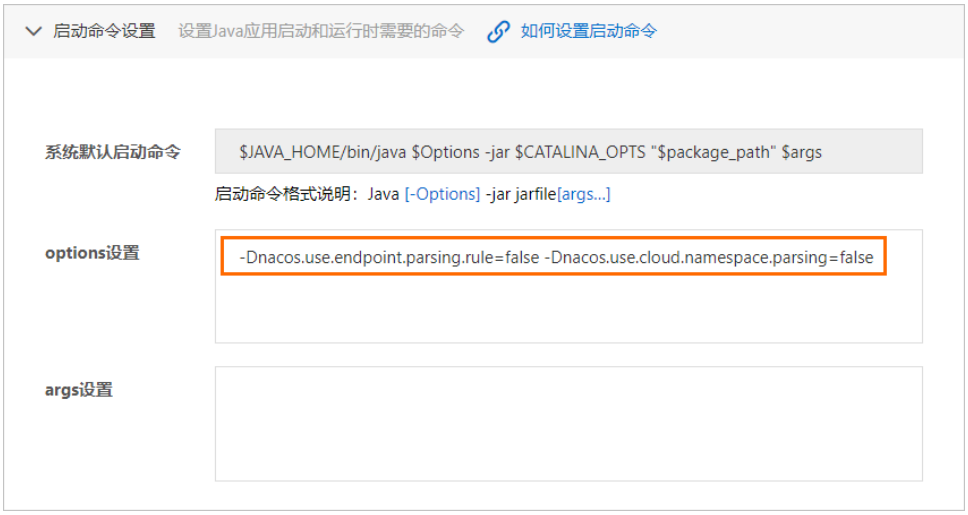
  > **注意** 启动参数需要放在 `-jar` 之前，否则可能会导致无法使用非SAE自带的注册中心。

  - 如果采用镜像方式，请将 `-Dnacos.use.endpoint.parsing.rule=false` 和 `-Dnacos.use.cloud.namespace.parsing=false` 配置在镜像文件的程序启动命令中。关于Docker镜像制作方法，请参见制作Java镜像。

    示例代码如下：

    ```
    RUN echo 'eval exec java -Dnacos.use.endpoint.parsing.rule=false -Dnacos.use.cloud.namespace.parsing=false -jar $CATALINA_OPTS /home/admin/app/hello-edas-0.0.1-SNAPSHOT.jar'> /home/admin/start.sh && chmod +x /home/admin/start.sh
    ```

- 如果采用JAR包方式，请在控制台**启动命令设置**区域的**options设置**文本框输入 `-Dnacos.use.endpoint.parsing.rule=false -Dnacos.use.cloud.namespace.parsing=false` 。图示为**OpenJDK 8**运行环境下的Java应用。具体操作，请参见设置启动命令。



准备工作

在开始开发前，请确保您已经完成以下工作。

- 下载Maven并设置环境变量。
- 下载最新版本的Nacos Server。
- 按以下步骤启动Nacos Server。
  i. 解压下载的Nacos Server压缩包。
  ii. 进入 `nacos/bin` 目录，启动Nacos Server。
    - Linux/Unix/Mac系统：执行命令 `sh startup.sh -m standalone` 。
    - Windows系统：双击执行 `startup.cmd` 文件。

创建服务提供者

在本地创建服务提供者应用工程，添加依赖，开启服务注册与发现功能，并将注册中心指定为Nacos Server。

1. 创建命名为 `nacos-service-provider` 的Maven工程。
2. 在 `pom.xml` 文件中添加依赖。

   具体示例，请参见nacos-service-provider。本文以Spring Boot 2.1.4.RELEASE和Spring Cloud Greenwich.SR1为例，依赖如下：

   > **说明** 不支持Spring Boot 2.4及以上版本。支持Spring Cloud Alibaba2.2.6.RELEASE版本（1.4.2客户端版本）。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>2.1.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

示例中使用的版本为Spring Cloud Greenwich，对应Spring Cloud Alibaba版本为2.1.1.RELEASE。

- 如果使用Spring Cloud Finchley版本，对应Spring Cloud Alibaba版本为2.0.1.RELEASE。
- 如果使用Spring Cloud Edgware版本，对应Spring Cloud Alibaba版本为1.5.1.RELEASE。

> **说明** Spring Cloud Edgware版本的生命周期已结束，不推荐使用该版本开发应用。

3. 在 `src\main\java` 下创建Package `com.aliware.edas` 。
4. 在Package `com.aliware.edas` 中创建服务提供者的启动类 `ProviderApplication` ，并添加以下代码。

   其中 `@EnableDiscoveryClient` 注解表明此应用需开启服务注册与发现功能。

```
package com.aliware.edas;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
@SpringBootApplication
@EnableDiscoveryClient
public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
}
```

5. 在Package `com.aliware.edas` 中创建 `EchoController` 。

   `EchoController` 中，指定URL mapping为 `/echo/{string}` ，指定HTTP方法为GET，从URL路径中获取方法参数，并回显收到的参数。

```
package com.aliware.edas;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class EchoController {
    @RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
    public String echo(@PathVariable String string) {
        return string;
    }
}
```

6. 在 `src\main\resources` 路径下创建文件 `application.properties` ，在 `application.properties` 中添加如下配置，指定Nacos Server的地址。

```
spring.application.name=service-provider
server.port=18081
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
```

   其中 `127.0.0.1` 为Nacos Server的地址。如果您的Nacos Server部署在另外一台机器，则需要修改成对应的IP地址。如果有其它需求，可以在 `application.properties` 文件中增加配置。更多信息，请参见配置项参考。

7. 验证结果。

   i. 执行 `nacos-service-provider` 中 `ProviderApplication` 的 `main` 函数，启动应用。

   ii. 登录本地启动的Nacos Server控制台 `http://127.0.0.1:8848/nacos` 。

   本地Nacos控制台的默认用户名和密码同为*nacos*。

   iii. 在左侧导航栏选择**服务管理 > 服务列表**。

   可以看到服务列表中已经包含了 `service-provider` ，且在**详情**中可以查询该服务的详情。

创建服务消费者

本节除介绍服务注册的功能，还将介绍Nacos服务与RestTemplate和FeignClient两个客户端如何配合使用。

1. 创建命名为 `nacos-service-consumer` 的Maven工程。

2. 在 `pom.xml` 中添加依赖。

   具体示例，请参见nacos-service-consumer。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>2.1.0.RELEASE</version>
    </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>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

3. 在 `src\main\java` 下创建Package `com.aliware.edas` 。
4. 在Package `com.aliware.edas` 中配置RestTemplate和FeignClient。

i. 在Package `com.aliware.edas` 中创建一个接口类 `EchoService` ，添加 `@FeignClient` 注解，并配置对应的HTTP URL地址及HTTP方法。

```
package com.aliware.edas;
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 = "service-provider")
public interface EchoService {
    @RequestMapping(value = "/echo/{str}", method = RequestMethod.GET)
    String echo(@PathVariable("str") String str);
}
```

ii. 在Package `com.aliware.edas` 中创建启动类 `ConsumerApplication` 并添加相关配置。

- 使用 `@EnableDiscoveryClient` 注解启用服务注册与发现。
- 使用 `@EnableFeignClients` 注解激活FeignClient。
- 添加 `@LoadBalanced` 注解将RestTemplate与服务发现集成。

```
package com.aliware.edas;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class ConsumerApplication {
    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
}
```

5. 在Package `com.aliware.edas` 中创建类 `TestController` 以演示和验证服务发现功能。

```
package com.aliware.edas;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;
@RestController
public class TestController {
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private EchoService echoService;
    @RequestMapping(value = "/echo-rest/{str}", method = RequestMethod.GET)
    public String rest(@PathVariable String str) {
        return restTemplate.getForObject("http://service-provider/echo/" + str,
                String.class);
    }
    @RequestMapping(value = "/echo-feign/{str}", method = RequestMethod.GET)
    public String feign(@PathVariable String str) {
        return echoService.echo(str);
    }
}
```

6. 在 `src\main\resources` 路径下创建文件 `application.properties`，在 `application.properties` 中添加以下配置，指定Nacos Server的地址。

```
spring.application.name=service-consumer
server.port=18082
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
```

其中 `127.0.0.1` 为Nacos Server的地址。如果您的Nacos Server部署在另外一台机器，则需要修改成对应的IP地址。如果有其它需求，可以在 `application.properties` 文件中增加配置。更多信息，请参见配置项参考。

7. 验证结果。
   i. 执行 `nacos-service-consumer` 中 `ConsumerApplication` 的 `main` 函数，启动应用。
   ii. 登录本地启动的Nacos Server控制台 `http://127.0.0.1:8848/nacos` 。

   本地Nacos控制台的默认用户名和密码同为*nacos*。
   iii. 在左侧导航栏中选择**服务管理 > 服务列表**，可以看到服务列表中已经包含了 `service-consumer` ，且在详情中可以查询该服务的详情。

本地测试

在本地测试消费者对提供者的服务调用结果。

- Linux/Unix/Mac系统：运行以下命令。

```
curl http://127.0.0.1:18082/echo-rest/rest-rest
curl http://127.0.0.1:18082/echo-feign/feign-rest
```

- Windows系统：在浏览器中输入*http://127.0.0.1:18082/echo-rest/rest-rest*和*http://127.0.0.1:18082/echo-feign/feign-rest*。

将应用部署到SAE

在本地完成应用的开发和测试后，便可将应用打包并部署到SAE。具体步骤，请参见部署应用到SAE。

> 注意
> - SAE暂不支持创建空应用，因此第一次部署需在控制台完成。
> - 如果使用JAR包部署，在应用部署配置时选择**应用运行环境**为**标准Java应用运行环境**。
> - 如果使用WAR包部署，在应用部署配置时**应用运行环境**为pache-tomcat-XXX。

当您将应用部署到SAE时，SAE服务注册中心会以更高优先级去设置Nacos Server服务端地址和服务端口，以及命名空间、AccessKey、Context-path信息。您无需进行任何额外的配置，原有的配置内容可以选择保留或删除。

结果验证

1. 为部署到SAE的应用绑定SLB。具体步骤，请参见绑定公网SLB。
2. 在浏览器输入配置好的公网访问地址，并在应用首页发起调用请求。
3. 登录SAE控制台，进入消费者应用详情页面，在左侧导航栏选择**应用监控 > 应用总览**，查看服务调用数据总览。
   如果能够监测到调用数据，则说明服务调用成功。

配置项参考

配置项	Key	默认值	说明
服务端地址	spring.cloud.nacos.discovery.server-addr	无	Nacos Server启动监听的IP地址和端口。
服务名	spring.cloud.nacos.discovery.service	${spring.application.name}	给当前的服务命名。
网卡名	spring.cloud.nacos.discovery.network-interface	无	当IP未配置时，注册的IP为此网卡所对应的IP地址。如果网卡名也未配置，则默认取第一块网卡的地址。
注册的IP地址	spring.cloud.nacos.discovery.ip	无	优先级最高。
注册的端口	spring.cloud.nacos.discovery.port	-1	默认情况下不用配置，系统会自动探测。
命名空间	spring.cloud.nacos.discovery.namespace	无	常用场景之一是隔离不同环境的资源，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

配置项	Key	默认值	说明
Metadata	spring.cloud.nacos.discovery.metadata	无	使用Map格式配置，您可以根据自己的需求自定义一些和服务相关的元数据信息。
集群	spring.cloud.nacos.discovery.cluster-name	DEFAULT	配置成Nacos集群名称。
接入点	spring.cloud.nacos.discovery.endpoint	UTF-8	地域的某个服务的入口域名。通过此域名可以动态地获取服务端地址，此配置在部署到EDAS时无需填写。
是否集成Ribbon	ribbon.nacos.enabled	true	如果没有明确需求，不需要修改。

更多关于Spring Cloud Alibaba Nacos Discovery的信息，请参见开源版本的Nacos Discovery。

更多信息

- 更多关于Spring Cloud Alibaba Nacos Discovery的信息，请参见Spring Cloud Alibaba Nacos Discovery。
- 如果您在本地开发了依赖Eureka、Consul、ZooKeeper等组件实现的服务注册与发现的Spring Cloud应用，那么修改该应用的依赖配置并部署至SAE的相关操作请参见将Spring Cloud应用托管到SAE。

1.3. 实现负载均衡

Spring Cloud的负载均衡是通过Ribbon组件完成的，Ribbon提供了客户端侧的软件负载均衡算法。Spring Cloud中的RestTemplate和Feign客户端底层的负载均衡是通过Ribbon实现的，本章介绍如何在您的应用中实现RestTemplate和Feign的负载均衡用法。

背景信息

Nacos集成了Ribbon的功能，NacosServerList实现了Ribbon提供的com.netflix.loadbalancer.ServerList接口。这个接口是通用的，其它类似的服务发现组件，Eureka、Consul、ZooKeeper也都实现了对应的ServerList接口，例如DomainExtractingServerList、ConsulServerList和ZookeeperServerList等。

实现该接口相当于接入了通用的Spring Cloud负载均衡规范。从Eureka、Consul、ZooKeeper等服务发现方案切换到Spring Cloud Alibaba的服务发现方案，无论是RestTemplate、FeignClient，以及已过时的AsyncRestTemplate，都无需修改任何代码，即可实现负载均衡。

> **说明** SAE兼容Hystrix，但是前提是客户端需要引入Hystrix的依赖。支持fallback属性，也可以通过Sentinel实现。

在本地开发应用时，可以使用Alibaba Cloud Toolkit插件实现本地应用和部署在SAE中的应用的相互调用，即端云互联，而无需搭建VPN，帮助您提升开发效率。更多信息，请参见使用Cloud Toolkit实现端云互联（IntelliJ IDEA）。

您可以按照本文的内容实现应用的负载均衡，也可以直接下载应用Demo：service-provider和service-consumer。

RestTemplate和Feign的实现方式有所不同，下面将分别介绍如何在您的应用中实现RestTemplate和Feign的负载均衡用法。

RestTemplate

RestTemplate是Spring提供的用于访问REST服务的客户端，提供了多种便捷访问远程HTTP服务的方法，能够大大提高客户端的编写效率。

您需要在您的应用中按照下面的示例修改代码，以便使用RestTemplate的负载均衡。

```
public class MyApp {
    // 注入刚刚使用@LoadBalanced注解修饰构造的RestTemplate。
    // 该注解相当于给RestTemplate加上了一个拦截器：LoadBalancerInterceptor。
    // LoadBalancerInterceptor内部会使用LoadBalancerClient接口的实现类RibbonLoadBalancerClient
完成负载均衡。
    @Autowired
    private RestTemplate restTemplate;
    @LoadBalanced // 使用@LoadBalanced注解修改构造的RestTemplate，使其拥有一个负载均衡功能。
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
    // 使用RestTemplate调用服务，内部会使用负载均衡调用服务。
    public void doSomething() {
        Foo foo = restTemplate.getForObject("http://service-provider/query", Foo.class);
        doWithFoo(foo);
    }
    ...
}
```

Feign

Feign是一个Java实现的HTTP客户端，用于简化RESTful调用。

配合 @EnableFeignClients和 @FeignClient完成负载均衡请求。

1. 使用 @EnableFeignClients开启Feign功能。

   ```
   @SpringBootApplication
   @EnableFeignClients // 开启Feign功能。
   public class MyApplication {
     ...
   }
   ```

2. 使用 @FeignClient构造FeignClient。

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

3. 注入EchoService并完成echo方法的调用。

   调用echo方法相当于发起了一个HTTP请求。

```
public class MyService {
@Autowired // 注入刚刚使用@FeignClient注解修饰构造的EchoService。
    private EchoService echoService;
    public void doSomething() {
    // 相当于发起了一个http://service-provider/echo/test请求。
    echoService.echo("test");
        }
    ...
}
```

结果验证

`service-consumer` 和多个 `serveice-provider` 启动后，访问 `service-consumer` 提供的URL确认是否实现了负载均衡。

- RestTemplate

  多次访问 `/echo-rest/rest-test` 查看是否转发到不同的实例。

- Feign

  多次访问 `/echo-feign/feign-test` 查看是否转发到不同的实例。

1.4. 实现配置管理

本文以Nacos配置管理的Demo应用为例，介绍如何在本地开发、调试Spring Cloud应用，使用Spring Cloud Alibaba Nacos Config实现配置管理，并通过SAE进行配置管理与推送。

前提条件

在开发前，确保您已完成以下工作：

- 下载Maven并设置环境变量。具体信息，请参见设置环境变量。
- 下载最新版本的Nacos Server。
- 启动Nacos Server。
  i. 解压下载的Nacos Server压缩包。
  ii. 进入 `nacos/bin` 目录，启动Nacos Server。
    - Linux、Unix和macOS系统：执行命令 `sh startup.sh -m standalone` 。
    - Windows系统：双击执行 `startup.cmd` 文件。
- 在本地Nacos Server控制台新建配置。
  i. 登录本地Nacos Server控制台（用户名和密码均默认为 `nacos` ）。
  ii. 在左侧导航栏单击**配置列表**，在**配置列表**页面右上角单击＋图标。
  iii. 在**新建配置**页面输入以下参数，单击并**发布**。
    - **Data ID**：*nacos-config-example.properties*
    - **Group**：*DEFAULT_GROUP*
    - **配置内容**：*test.name=nacos-config-test*

背景信息

在开发Spring Cloud应用时，您可以在本地使用Nacos实现应用的配置管理。由于SAE集成了Nacos的应用配置管理ACM的开源版本，在应用部署到SAE后，您可以通过SAE对应用进行配置的管理和推送。

本文以Spring Cloud应用开发过程为例，使用Spring Cloud Alibaba Nacos Config实现配置管理。您可以下载该应用示例的Demo进行操作。

> **说明** Spring Cloud Alibaba Nacos Config对Nacos与Spring Cloud的框架进行了整合，支持Spring Cloud的配置注入规范。

使用Nacos Config实现配置管理

1. 创建一个Maven工程，命名为 `nacos-config-example` 。
2. 在 `pom.xml` 文件中添加依赖。

   以Spring Boot 2.1.4.RELEASE和Spring Cloud Greenwich.SR1为例。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
        <version>2.1.1.RELEASE</version>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

示例中使用的版本为Spring Cloud Greenwich，对应Spring Cloud Alibaba版本为2.1.1.RELEASE。

- 如果使用Spring Cloud Finchley版本，对应Spring Cloud Alibaba版本为2.0.1.RELEASE。
- 如果使用Spring Cloud Edgware版本，对应Spring Cloud Alibaba版本为1.5.1.RELEASE。

> **说明** Spring Cloud Edgware版本的生命周期已结束，不推荐使用这个版本开发应用。

3. 在 `src\main\java` 下创建Package `com.aliware.edas` 。
4. 在Package `com.aliware.edas` 中创建 `nacos-config-example` 的启动类 `NacosConfigExampleApplication` 。

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class NacosConfigExampleApplication {
    public static void main(String[] args) {
        SpringApplication.run(NacosConfigExampleApplication.class, args);
    }
}
```

5. 在Package `com.aliware.edas` 中创建一个简单的Controller `EchoController` ，自动注入一个属性 `userName` ，且通过 `@Value` 注解指定从配置中取Key为 `test.name` 的值。

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RefreshScope
public class EchoController {
    @Value("${test.name}")
    private String userName;
    @RequestMapping(value = "/")
    public String echo() {
        return userName;
    }
}
```

6. 在 `src\main\resources` 路径下创建配置文件 `bootstrap.properties` ，在 `bootstrap.properties` 中添加如下配置，指定Nacos Server的地址。

   其中 `127.0.0.1:8848` 为Nacos Server的地址， `18081` 为服务端口。

   如果您的Nacos Server部署在另外一台机器，则需要修改成对应的IP和端口。如果有其它需求，可以参照配置项参考在 `bootstrap.properties` 文件中增加配置。

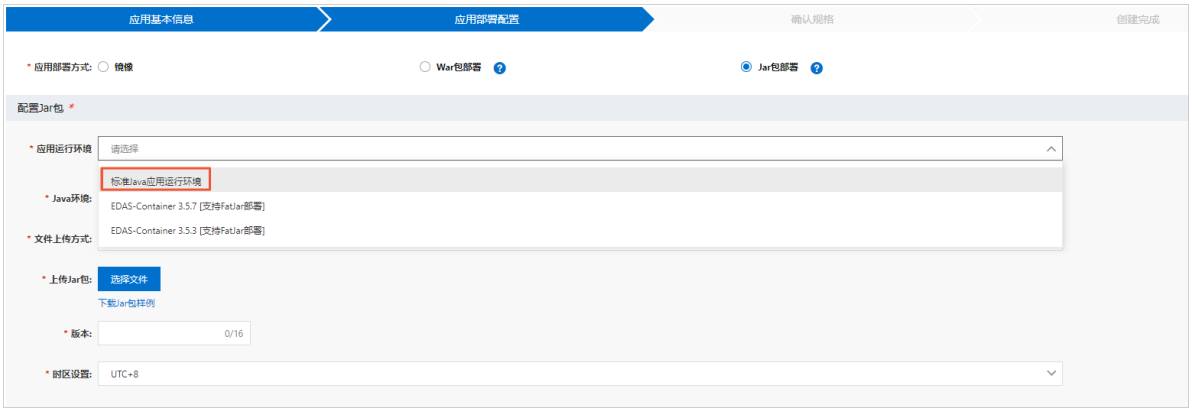
```
spring.application.name=nacos-config-example
server.port=18081
spring.cloud.nacos.config.server-addr=127.0.0.1:8848
```

7. 执行 `NacosConfigExampleApplication` 中的main函数，启动应用。

部署到SAE

在本地完成应用的开发和测试后，您可以将应用程序部署到SAE。具体操作，请参见应用部署概述。

- SAE配置管理中心提供了正式商用版本的Nacos Server。当您将应用部署到SAE的时候，SAE会通过优先级更高的方式去设置Nacos Server服务端地址和服务端口，以及namespace、access-key、secret-key和context-path信息。您无需进行任何额外的配置，原有的配置内容可以选择保留或删除。
- 您初次在SAE控制台进行部署时，如果使用JAR包部署，在创建应用时**应用运行环境**须选**标准Java应用运行环境**。



在部署应用前，请使用SAE的配置管理功能，创建与本地Nacos Server中相同的应用配置。

1. 登录SAE控制台。
2. 在左侧导航栏选择**配置管理>配置列表**。
3. 在**配置列表**页面选择**地域**和**命名空间**，单击**创建配置**。
4. 在**创建配置**页面，输入以下参数，配置完成后单击**创建**。

参数	说明
Data ID	配置ID，以*nacos-config-example.properties*为例。 采用类似 `package.class` （如 `com.taobao.tc.refund.log.level` ）的命名规则保证全局唯一性。建议根据配置的业务含义定义*class*部分。仅允许使用英文小写字符和以下4种特殊字符：英文句号（.）、冒号（:）、短划线（-）和下划线（_）。不超过236个字符。
Group	配置分组（命名空间），以*DEFAULT_GROUP*为例。 建议填写 **产品名:模块名** （如 `ACM:Test` ）保证唯一性。后续可以根据Group进行鉴权。仅允许使用英文字符和以下4种特殊字符：英文句号（.）、冒号（:）、短划线（-）和下划线（_）。不超过128个字符。

参数	说明
（可选）数据加密	配置数据是否加密。如果您的配置中包含敏感数据，建议您使用加密存储功能，降低配置泄漏风险。 注意 使用前必须开通密钥管理服务，并授权ACM使用密钥管理服务进行加解密。因为ACM数据加密功能依赖密钥管理服务，为其配置加密。加密配置的Data ID均以cipher-开头，具体操作，请参见创建和使用加密配置。
配置格式	选择配置格式，以TEXT为例。 SAE会根据您选择的格式进行数据校验。默认选择TEXT，支持TEXT、JSON、XML、YAML HTML和Properties格式。
配置内容	输入配置的内容，以*test.name=nacos-config-test*为例。 建议不超过10 KB，最大不超过100 KB。
（可选）配置描述	配置描述信息。便于理解配置含义，不超过128个字符。
（可选）更多配置	
（可选）应用	配置归属应用名。仅允许使用英文小写字符和以下4种特殊字符：英文句号（.）、冒号（:）、短划线（-）和下划线（_）。不超过128个字符。
（可选）标签	配置标签。方便您根据自己的维度管理配置，最多支持5个标签，每个标签不超过64个字符。

结果验证

部署完成后，您可以通过查看日志确认应用是否启动成功。

1. 执行命令`curl http://<应用实例 IP>:<服务端口>`，例如`curl http://192.168.0.34:8080`查看是否返回配置内容`nacos-config-test`。
2. 在SAE控制台将原有配置内容修改为`nacos-config-test2`，再执行命令`curl http://<应用实例 IP>:<服务端口>`，例如`curl http://192.168.0.34:8080`，查看是否返回变更后的配置内容`nacos-config-test2`。

配置项参考

如果有其它需求，可以参照下表在`bootstrap.properties`文件中增加配置。

配置项	key	默认值	说明

配置项	key	默认值	说明
服务端地址	spring.cloud.nacos.config.server-addr	无	无
DataId前缀	spring.cloud.nacos.config.prefix	${spring.application.name}	Data ID的前缀
Group	spring.cloud.nacos.config.group	DEFAULT_GROUP	分组
Data ID后缀及内容文件格式	spring.cloud.nacos.config.file-extension	properties	Data ID的后缀，同时也是配置内容的文件格式，默认是properties，也支持YAML和YML格式。
配置内容的编码方式	spring.cloud.nacos.config.encode	UTF-8	配置的编码
获取配置的超时时间	spring.cloud.nacos.config.timeout	3000	单位为ms
配置的命名空间	spring.cloud.nacos.config.namespace		常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源隔离等。
相对路径	spring.cloud.nacos.config.context-path		服务端API的相对路径
接入点	spring.cloud.nacos.config.endpoint	UTF-8	地域的某个服务的入口域名，通过此域名可以动态地获取服务端地址。
是否开启监听和自动刷新	spring.cloud.nacos.config.refresh.enabled	true	默认为true，不需要修改。

更多配置项，请参考开源版本的Spring Cloud Alibaba Nacos Config文档。

1.5. 搭建服务网关

本文介绍如何基于Spring Cloud Gateway和Spring Cloud Netflix Zuul使用Nacos搭建应用的服务网关。

为什么使用SAE服务注册中心

SAE服务注册中心提供了开源Nacos Server的商用版本，使用开源版本 `Spring Cloud Alibaba Nacos Discovery` 开发的应用可以直接使用SAE提供的商业版服务注册中心。

SAE服务注册中心与Nacos、Eureka和Consul相比，具有以下优势：

- 共享组件，节省了部署、运维Nacos、Eureka或Consul的成本。
- 在服务注册和发现的调用中都进行了链路加密，保护您的服务，无需再担心服务被未授权的应用发现。
- SAE服务注册中心与SAE其他组件紧密结合，为您提供一整套的微服务解决方案，包括环境隔离、灰度发布等。

您在SAE部署应用时，SAE服务注册中心以高优先级自动设置Nacos Server服务端地址和服务端口，以及命名空间、AccessKey等信息，无需进行任何额外的配置。

基于Spring Cloud Gateway搭建服务网关

介绍如何使用Nacos基于Spring Cloud Gateway从零搭建应用的服务网关。

1. 创建服务网关。
   i. 创建命名为 `spring-cloud-gateway-nacos` 的Maven工程。
   ii. 在 `pom.xml` 文件中添加Spring Boot和Spring Cloud的依赖。

   以Spring Boot 2.1.4.RELEASE和Spring Cloud Greenwich.SR1版本为例。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>2.1.1.RELEASE</version>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

iii. 开发服务网关启动类 `GatewayApplication` 。

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

iv. 在 `application.yaml` 中添加如下配置，将注册中心指定为Nacos Server的地址。

其中 `127.0.0.1:8848` 为Nacos Server的地址。如果您的Nacos Server部署在另外一台机器，则需要修改成对应的地址。

其中routes配置了Gateway的路由转发策略，这里我们配置将所有前缀为 `/provider1/` 的请求都路由到服务名为 `service-provider` 的后端服务中。

```
server:
 port: 18012
spring:
 application:
        name: spring-cloud-gateway-nacos
 cloud:
        gateway: # config the routes for gateway
          routes:
          - id: service-provider          #将/provider1/开头的请求转发到provider1
            uri: lb://service-provider
            predicates:
            - Path=/provider1/**
            filters:
            - StripPrefix=1               #表明前缀/provider1需要截取掉
        nacos:
          discovery:
            server-addr=127.0.0.1:8848
```

v. 执行启动类 `GatewayApplication` 中的main函数，启动Gateway。

vi. 登录本地启动的Nacos Server控制台http://127.0.0.1:8848/nacos（本地Nacos控制台的默认用户名和密码同为nacos），在左侧导航栏中选择**服务管理 > 服务列表**，可以看到服务列表中已经包含了spring-cloud-gateway-nacos，且在详情中可以查询该服务的详情。表明网关已经启动并注册成功，接下来我们将通过创建一个下游服务来验证网关的请求转发功能。

2. 创建服务提供者。

创建一个服务提供者的应用，请参见将Spring Cloud应用托管到SAE。

服务提供者示例：

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication, args);
    }
    @RestController
    public class EchoController {
        @RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
        public String echo(@PathVariable String string) {
            return string;
        }
    }
}
```

3. 结果验证。
   - 本地验证。

     本地启动开发好的服务网关和服务提供者，通过访问Spring Cloud Gateway将请求转发给后端服务，可以看到调用成功的结果。

     ```
     ➜  spring-cloud-gateway-nacos curl http://127.0.0.1:15012/provider1/echo/123456
     123456%
     ```

   - 在SAE中验证。

     SAE服务注册中心提供了正式商用版本Nacos。当您将应用部署到SAE的时候，SAE会直接替换本地Nacos Server的地址和服务端口，以及namespace、access-key、secret-key、context-path信息。您无需进行任何额外的配置，原有的配置内容可以选择保留或删除。

基于Zuul搭建服务网关

介绍如何基于Zuul使用Nacos作为服务注册中心从零搭建应用的服务网关。

1. 创建服务网关。
   i. 创建命名为 `spring-cloud-zuul-nacos` 的Maven工程。
   ii. 在 `pom.xml` 文件中添加Spring Boot、Spring Cloud和Spring Cloud Alibaba的依赖。

   请添加Spring Boot 2.1.4.RELEASE、Spring Cloud Greenwich.SR1和Spring Cloud Alibaba 0.9.0版本依赖。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>2.1.1.RELEASE</version>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

iii. 开发服务网关启动类 `ZuulApplication` 。

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

iv. 在 `application.properties` 中添加如下配置，将注册中心指定为Nacos Server的地址。

其中 `127.0.0.1:8848` 为Nacos Server的地址。如果您的Nacos Server部署在另外一台机器，则需要修改成对应的地址。

其中routes配置了Zuul的路由转发策略，这里我们配置将所有前缀为 `/provider1/` 的请求都路由到服务名为 `service-provider` 的后端服务中。

```
spring.application.name=spring-cloud-zuul-nacos
server.port=18022
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
zuul.routes.opensource-provider1.path=/provider1/**
zuul.routes.opensource-provider1.serviceId=service-provider
```

v. 执行spring-cloud-zuul-nacos中的main函数 `ZuulApplication` ，启动服务。

vi. 登录本地启动的Nacos Server控制台http://127.0.0.1:8848/nacos （本地Nacos控制台的默认用户名和密码同为nacos），在左侧导航栏中选择服务管理 > 服务列表，可以看到服务列表中已经包含了spring-cloud-zuul-nacos，且在详情中可以查询该服务的详情。表明网关已经启动并注册成功，接下来我们将通过创建一个下游服务来验证网关的请求转发功能。

2. 创建服务提供者。

如何快速创建一个服务提供者，请参见将Spring Cloud应用托管到SAE。

服务提供者启动类示例：

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication, args);
    }
    @RestController
    public class EchoController {
        @RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
        public String echo(@PathVariable String string) {
            return string;
        }
    }
}
```

3. 结果验证。

- 本地验证。

本地启动开发好的服务网关Zuul和服务提供者，通过访问Spring Cloud Netflix Zuul将请求转发给后端服务，可以看到调用成功的结果。

```
➜  spring-cloud-gateway-nacos curl http://127.0.0.1:18022/provider1/echo/123456
123456%
➜  spring-cloud-gateway-nacos
```

- 在SAE中验证。

您将应用部署到SAE，并验证。具体操作，请参见将Spring Cloud应用托管到SAE。

SAE服务注册中心提供了正式商用版本Nacos。当您将应用部署到SAE的时候，SAE会直接替换本地Nacos Server的地址和服务端口，以及namespace、access-key、secret-key、context-path信息。您无需进行任何额外的配置，原有的配置内容可以选择保留或删除。

版本说明

- 示例中使用的Spring Cloud版本为Greenwich，对应的Spring Cloud Alibaba版本为2.1.1.RELEASE。
- Spring Cloud Finchley对应的Spring Cloud Alibaba版本为2.0.1.RELEASE。
- Spring Cloud Edgware对应的Spring Cloud Alibaba版本为1.5.1.RELEASE。

> **说明** Spring Cloud Edgware版本的生命周期已结束，不推荐使用这个版本开发应用。

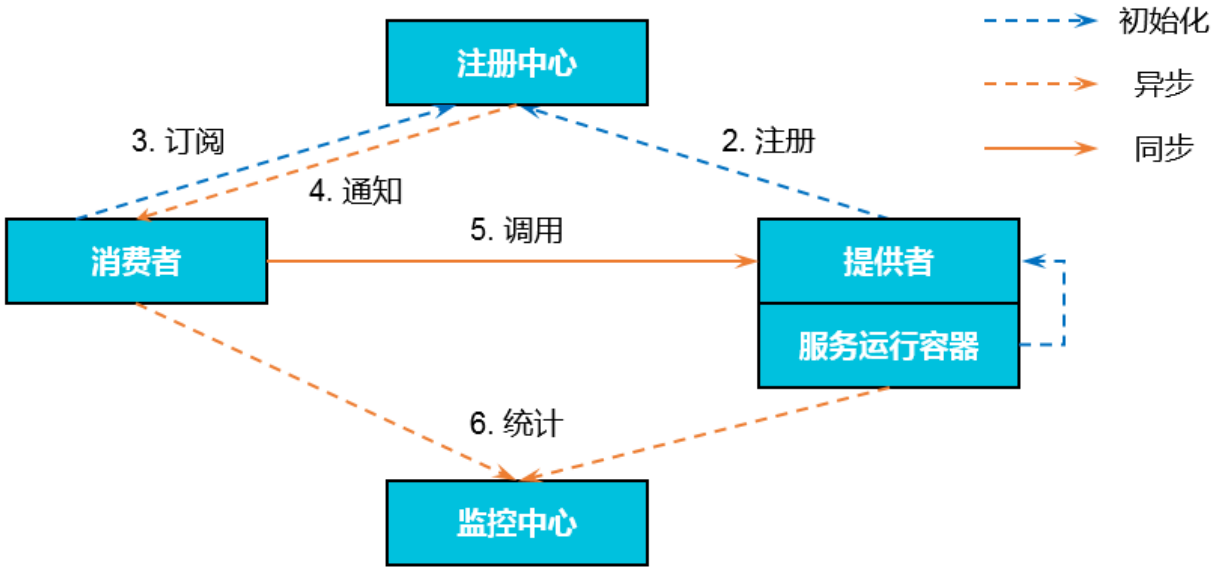
2.使用Dubbo开发应用

2.1. Dubbo开发概述

SAE支持原生Dubbo微服务框架，在该框架下开发的微服务只需添加依赖和修改配置，便可获得SAE企业级的微服务应用托管、微服务治理、监控报警和应用诊断等能力，实现零代码量应用迁移。

Dubbo的架构

Dubbo的架构如下图所示。



1. 服务运行容器负责启动，加载，运行提供者服务。
2. 提供者在启动时，需要向注册中心进行注册。
3. 消费者在启动时，需要向注册中心订阅所需的服务。
4. 注册中心返回提供者地址列表给消费者。如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 消费者从提供者地址列表中，基于软负载均衡算法，选择某个提供者进行调用。如果调用失败，则重新调用其他提供者。
6. 消费者和提供者在内存中存储累计调用次数和调用时间，定时（每分钟）发送统计数据至监控中心。

2.2. 将Dubbo应用托管到SAE

本文以包含服务提供者（本文简称Provider）和服务消费者（本文简称Consumer）的Dubbo微服务应用为例，介绍如何在本地通过XML配置的方式，开发Dubbo微服务示例应用，并部署到Serverless应用引擎SAE（Serverless App Engine）。

为什么托管到SAE

将Dubbo应用托管到SAE，您仅需关注Dubbo应用自身的逻辑，无需再关注注册中心和配置中心搭建和维护，托管后还可以使用SAE提供的弹性伸缩、一键启停和监控等功能，有效降低开发和运维成本。

当您的微服务应用较多时，注册中心按推荐程度由高到低依次排序如下：

- 商业版的服务注册中心（MSE）

- 自建服务注册中心
- SAE内置服务注册中心

如果您选择商业版的服务注册中心，即使用MSE的Nacos作为服务注册中心，具体操作，请参见使用MSE的Nacos注册中心。

如果您选择使用自建Nacos作为服务注册中心，具体操作，请参见自建Nacos服务注册中心。您需要确认以下内容：

- 请确保SAE的网络与自建Nacos的网络互通。
- 在部署应用时建议使用镜像方式或者JAR包方式，并配置启动参数 `-Dnacos.use.endpoint.parsing.rule=false` 和 `-Dnacos.use.cloud.namespace.parsing=false` 。

  > 注意 启动参数需要放在 `-jar` 之前，否则可能会导致无法使用非SAE自带的注册中心。

  - 如果采用镜像方式，请将 `-Dnacos.use.endpoint.parsing.rule=false` 和 `-Dnacos.use.cloud.namespace.parsing=false` 配置在镜像文件的程序启动命令中。关于Docker镜像制作方法，请参见制作Java镜像。

    示例代码如下：

    ```
    RUN echo 'eval exec java -Dnacos.use.endpoint.parsing.rule=false -Dnacos.use.cloud.namespace.parsing=false -jar $CATALINA_OPTS /home/admin/app/hello-edas-0.0.1-SNAPSHOT.jar'> /home/admin/start.sh && chmod +x /home/admin/start.sh
    ```

  - 如果采用JAR包方式，请在控制台**启动命令设置**区域的**options设置**文本框输入 `-Dnacos.use.endpoint.parsing.rule=false -Dnacos.use.cloud.namespace.parsing=false` 。图示为**OpenJDK 8**运行环境下的Java应用。具体操作，请参见设置启动命令。

准备工作

- 下载Maven并设置环境变量。
- 启动Nacos Server。
  i. 下载并解压Nacos Server。
  ii. 进入*nacos/bin*目录，启动Nacos Server。
  - Linux、Unix、macOS系统：执行命令 `sh startup.sh -m standalone` 。

- Windows系统：执行命令 `startup.cmd -m standalone` 。

> **说明** `standalone` 表示单机模式运行，非集群模式。*startup.cmd*文件默认以集群模式启动，因此您在使用Windows系统时，如果直接双击执行*startup.cmd*文件会导致启动失败，此时需要在*startup.cmd*文件内设置 `MODE="standalone"` 。更多信息，请参见Nacos快速开始。

版本说明

SAE支持托管2.5.x，2.6.x，2.7.x版本的Dubbo。推荐使用2.7.x，可以获得更强大的服务治理能力。本文将会以2.7.3版本为例，介绍如何将Dubbo应用托管到SAE。

步骤一：创建服务提供者

在本地创建一个提供者应用工程，添加依赖、配置服务注册与发现，并将注册中心指定为Nacos。

1. 创建Maven项目并引入依赖。
   i. 使用IDE（如IntelliJ IDEA或Eclipse）创建一个Maven项目。
   ii. 在 `pom.xml` 文件中添加dubbo、dubbo-registry-nacos和nacos-client依赖。

```
<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-registry-nacos</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>1.1.1</version>
    </dependency>
</dependencies>
```

2. 开发Dubbo服务提供者。

   Dubbo中服务都是以接口形式提供。
   i. 在*src/main/java*路径下创建一个package `com.alibaba.edas` 。
   ii. 在 `com.alibaba.edas` 下创建一个接口（interface） `IHelloService` ，里面包含一个 `SayHello` 方法。

```
package com.alibaba.edas;
public interface IHelloService {
    String sayHello(String str);
}
```

iii. 在 `com.alibaba.edas` 下创建一个类 `IHelloServiceImpl`，实现此接口。

```
package com.alibaba.edas;
public class IHelloServiceImpl implements IHelloService {
    public String sayHello(String str) {
        return "hello " + str;
    }
}
```

3. 配置Dubbo服务。

i. 在*src/main/resources*路径下创建 `provider.xml` 文件并打开。

ii. 在 `provider.xml` 中，添加Spring相关的XML Namespace（xmlns）和XML Schema Instance（xmlns:xsi），以及Dubbo相关的Namespace（xmlns:dubbo）和Scheme Instance（xsi:schemaLocation）。

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
xmlns="http://www.springframework.org/schema/beans"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springfr
amework.org/schema/beans/spring-beans-4.3.xsd
http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd
">
```

iii. 在 `provider.xml` 中将接口和实现类暴露成Dubbo服务。

```
<dubbo:application name="demo-provider"/>
<dubbo:protocol name="dubbo" port="28082"/>
<dubbo:service interface="com.alibaba.edas.IHelloService" ref="helloService"/>
<bean id="helloService" class="com.alibaba.edas.IHelloServiceImpl"/>
```

iv. 在 `provider.xml` 中将注册中心指定为本地启动的Nacos Server。

```
<dubbo:registry address="nacos://127.0.0.1:8848" />
```

- `127.0.0.1` 为Nacos Server的地址。如果您的Nacos Server部署在另外一台机器，则需要修改成对应的IP地址。当将应用部署到SAE后，无需做任何修改，注册中心会替换成SAE上的注册中心的地址。
- `8848` 为Nacos Server的端口号，不可修改。

4. 启动服务。

i. 在 `com.alibaba.edas` 中创建类Provider，并按下面的代码在Provider的main函数中加载Spring Context，将配置好的Dubbo服务暴露。

```
package com.alibaba.edas;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Provider {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext
(new String[] {"provider.xml"});
        context.start();
        System.in.read();
    }
}
```

ii. 执行Provider的main函数，启动服务。

5. 登录Nacos控制台 `http://127.0.0.1:8848` ，在左侧导航栏中单击**服务列表**，查看提供者列表。
可以看到服务提供者里已经包含了 `com.alibaba.edas.IHelloService` ，且可以查询该服务的**服务分组**和**提供者IP**。

步骤二：创建服务消费者

在本地创建一个消费者应用工程，添加依赖、订阅服务的配置。

1. 创建Maven项目并引入依赖。
   i. 使用IDE（如IntelliJ IDEA或Eclipse）创建一个Maven项目。
   ii. 在 `pom.xml` 文件中添加dubbo、dubbo-registry-nacos和nacos-client依赖。

```
<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-registry-nacos</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>1.1.1</version>
    </dependency>
</dependencies>
```

2. 开发Dubbo服务提供者。
   Dubbo中服务都是以接口形式提供。
   i. 在*src/main/java*路径下创建Package，命名为 `com.alibaba.edas` 。
   ii. 在 `com.alibaba.edas` 下创建一个接口（interface） `IHelloService` ，里面包含一个 `SayHello` 方法。

> **说明** 通常是在一个单独的模块中定义接口，服务提供者和服务消费者都通过Maven依赖来引用此模块。本文档为了简便，服务提供者和服务消费者分别创建两个完全一模一样的接口，实际使用中不推荐这样使用。

```
package com.alibaba.edas;
public interface IHelloService {
    String sayHello(String str);
}
```

3. 配置Dubbo服务。
   i. 在*src/main/resources*路径下创建 `consumer.xml` 文件并打开。

ii. 在 `consumer.xml` 中，添加Spring相关的XML Namespace（xmlns）和XML Schema Instance（xmlns:xsi），以及Dubbo相关的Namespace（xmlns:dubbo）和Scheme Instance（xsi:schemaLocation）。

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
xmlns="http://www.springframework.org/schema/beans"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springfr
amework.org/schema/beans/spring-beans-4.3.xsd
http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd
">
```

iii. 在 `consumer.xml` 中添加如下配置，订阅Dubbo服务。

```
  <dubbo:application name="demo-consumer"/>
  <dubbo:registry address="nacos://127.0.0.1:8848"/>
  <dubbo:reference id="helloService" interface="com.alibaba.edas.IHelloService"/>
```

4. 启动并验证服务。

i. 在 `com.alibaba.edas` 下创建类Consumer，并按下面的代码在Consumer的main函数中加载Spring Context，订阅并消费Dubbo服务。

```
    package com.alibaba.edas;
    import org.springframework.context.support.ClassPathXmlApplicationContext;
    import java.util.concurrent.TimeUnit;
    public class Consumer {
        public static void main(String[] args) throws Exception {
            ClassPathXmlApplicationContext context = new ClassPathXmlApplicationCon
text(new String[] {"consumer.xml"});
            context.start();
            while (true) {
                try {
                    TimeUnit.SECONDS.sleep(5);
                    IHelloService demoService = (IHelloService)context.getBean("hel
loService");
                    String result = demoService.sayHello("world");
                    System.out.println(result);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

ii. 执行Consumer的main函数，启动服务。

5. 验证结果。

启动后，可以看到控制台不断地输出 `hello world`，表明服务消费成功。

登录Nacos控制台 `http://127.0.0.1:8848`，在左侧导航栏中单击**服务列表**，再在**服务列表**页面选择**调用者列表**。

可以看到包含了 `com.alibaba.edas.IHelloService`，且可以查看该服务的**服务分组**和**调用者IP**。

步骤三：部署到SAE

1. 分别在Provider和Consumer的pom.xml文件中添加如下配置，配置完成后执行**mvn clean package**将本地程序编译为可执行的JAR包。
   - Provider

```
<build>
 <plugins>
       <plugin>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-maven-plugin</artifactId>
           <executions>
               <execution>
                   <goals>
                       <goal>repackage</goal>
                   </goals>
                   <configuration>
                       <classifier>spring-boot</classifier>
                       <mainClass>com.alibaba.sae.Provider</mainClass>
                   </configuration>
               </execution>
           </executions>
       </plugin>
 </plugins>
</build>
```

   - Consumer

```
<build>
 <plugins>
       <plugin>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-maven-plugin</artifactId>
           <executions>
               <execution>
                   <goals>
                       <goal>repackage</goal>
                   </goals>
                   <configuration>
                       <classifier>spring-boot</classifier>
                       <mainClass>com.alibaba.sae.Consumer</mainClass>
                   </configuration>
               </execution>
           </executions>
       </plugin>
 </plugins>
</build>
```

2. 将编译好的Provider和Consumer应用包部署至SAE。具体操作，请参见在SAE控制台使用JAR文件部署微服务应用。

2.3. 使用Spring Boot开发Dubbo应用

除了可以使用传统的XML配置方式开发Dubbo应用，还可以使用Spring Boot开发Dubbo应用，特别对于Java技术薄弱和Maven经验少，且又不熟悉Dubbo框架的开发者更为适合。本文以全新开发过程，向您展示如何使用Spring Boot开发Dubbo应用，并使用SAE服务注册中心实现服务注册与发现。

前提条件

- 下载Maven并设置环境变量。
- 下载最新版本的Nacos Server。
- 启动Nacos Server。
  i. 解压下载的Nacos Server压缩包
  ii. 进入 `nacos/bin` 目录，启动Nacos Server。
    - Linux/Unix/Mac系统：执行命令 `sh startup.sh -m standalone` 。
    - Windows系统：双击执行 `startup.cmd` 文件。

为什么使用Spring Boot开发Dubbo应用

Spring Boot简化了微服务应用的配置和部署，同时Nacos又同时提供了服务注册发现和配置管理功能，两者结合的方式能够帮助您快速搭建基于Spring的Dubbo服务，相比xml的开发方式，大幅提升开发效率。

全新场景使用Spring Boot开发Dubbo应用有两种主要的方式：

- 使用xml开发。
- 使用Spring Boot的注解方式开发。

使用xml方式请参考将Dubbo应用托管到SAE。文本档介绍如何使用Spring Boot的注解方式开发Dubbo服务。

视频教程

本视频仅介绍使用Spring Boot开发Dubbo应用，部署部分请参见在SAE控制台部署应用。

示例工程

您可以按照本文的逐步搭建工程，也可以选择直接下载本文对应的示例工程，或者使用Git来clone： `git clone https://github.com/aliyun/alibabacloud-microservice-demo.git`

该项目包含了众多了示例工程，本文对应的示例工程位于 `alibabacloud-microservice-demo/microservice-doc-demo/dubbo-samples-spring-boot` 。

创建服务提供者

1. 创建命名为 `spring-boot-dubbo-provider` 的Maven工程。
2. 在 `pom.xml` 文件中添加所需的依赖。

   这里以Spring Boot 2.0.6.RELEASE为例。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.0.6.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>1.1.1</version>
    </dependency>
</dependencies>
```

3. 开发Dubbo服务提供者。

   Dubbo中服务都是以接口的形式提供的。

   i. 在 `src/main/java` 路径下创建一个package `com.alibaba.edas.boot` 。

   ii. 在 `com.alibaba.edas.boot` 下创建一个接口（interface） `IHelloService` ，里面包含一个 `SayHello` 方法。

   ```
   package com.alibaba.edas.boot;
   public interface IHelloService {
   String sayHello(String str);
   }
   ```

iii. 在 `com.alibaba.edas.boot` 下创建一个类 `IHelloServiceImpl` ，实现此接口。

```
package com.alibaba.edas.boot;
import com.alibaba.dubbo.config.annotation.Service;
@Service
public class IHelloServiceImpl implements IHelloService {
public String sayHello(String name) {
  return "Hello, " + name + " (from Dubbo with Spring Boot)";
 }
}
```

> **说明** 这里的Service注解是Dubbo提供的一个注解类，类的全名称为：com.alibaba.dubbo.config.annotation.Service。

4. 配置Dubbo服务。

i. 在 `src/main/resources` 路径下创建 `application.properties` 或 `application.yaml` 文件并打开。

ii. 在 `application.properties` 或 `application.yaml` 中添加如下配置。

```
# Base packages to scan Dubbo Components (e.g @Service , @Reference)
dubbo.scan.basePackages=com.alibaba.edas.boot
dubbo.application.name=dubbo-provider-demo
dubbo.registry.address=nacos://127.0.0.1:8848
```

> **说明**
>
> - 以上三个配置没有默认值，必须要给出具体的配置。
> - `dubbo.scan.basePackages` 的值是开发的代码中含有 `com.alibaba.dubbo.config.annotation.Service` 和 `com.alibaba.dubbo.config.annotation.Reference` 注解所在的包。多个包之间用逗号隔开。
> - `dubbo.registry.address` 的值前缀必须以nacos://开头，后面的IP地址和端口指的是Nacos Server的地址。代码示例中为本地地址，如果您将Nacos Server部署在其它机器上，请修改为实际的IP地址。

5. 开发并启动Spring Boot入口类 `DubboProvider` 。

```
package com.alibaba.edas.boot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DubboProvider {
    public static void main(String[] args) {
        SpringApplication.run(DubboProvider.class, args);
    }
}
```

6. 登录Nacos控制台 `http://127.0.0.1:8848` ，在左侧导航栏中单击**服务列表**，查看提供者列表。

可以看到服务提供者里已经包含了 `com.alibaba.edas.boot.IHelloService` ，且可以查询该服务的服务分组和提供者IP。

创建服务消费者

1. 创建一个Maven工程，命名为 `spring-boot-dubbo-consumer` 。
2. 在 `pom.xml` 文件中添加相关依赖。

   这里以Spring Boot 2.0.6.RELEASE为例。

```
    <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.0.6.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>1.1.1</version>
    </dependency>
</dependencies>
```

   如果您需要选择使用Spring Boot 1.x的版本，请使用Spring Boot 1.5.x版本，对应的com.alibaba.boot:dubbo-spring-boot-starter版本为0.1.0。

   > **说明** Spring Boot 1.x版本的生命周期即将在2019年08月结束，推荐使用新版本开发您的应用。

3. 开发Dubbo消费者。

   i. 在 `src/main/java` 路径下创建package `com.alibaba.edas.boot` 。

ii. 在 `com.alibaba.edas.boot` 下创建一个接口（interface） `IHelloService` ，里面包含一个 `SayHello` 方法。

```
package com.alibaba.edas.boot;
public interface IHelloService {
 String sayHello(String str);
}
```

4. 开发Dubbo服务调用。

例如需要在Controller中调用一次远程Dubbo服务，开发的代码如下所示。

```
package com.alibaba.edas.boot;
import com.alibaba.dubbo.config.annotation.Reference;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
    public class DemoConsumerController {
        @Reference
        private IHelloService demoService;
        @RequestMapping("/sayHello/{name}")
        public String sayHello(@PathVariable String name) {
            return demoService.sayHello(name);
        }
    }
```

> **说明** 这里的Reference注解是com.alibaba.dubbo.config.annotation.Reference。

5. 在 `application.properties/application.yaml` 配置文件中新增以下配置。

```
dubbo.application.name=dubbo-consumer-demo
dubbo.registry.address=nacos://127.0.0.1:8848
```

> **说明**
>
> - 以上两个配置没有默认值，必须要给出具体的配置。
> - `dubbo.registry.address` 的值前缀必须以 `nacos://` 开头，后面的IP地址和端口为Nacos Server的地址。代码示例中为本地地址，如果您将Nacos Server部署在其它机器上，请修改为实际的IP地址。

6. 开发并启动Spring Boot入口类 `DubboConsumer` 。

```
package com.alibaba.edas.boot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DubboConsumer {
    public static void main(String[] args) {
        SpringApplication.run(DubboConsumer.class, args);
    }
}
```

7. 登录Nacos控制台 `http://127.0.0.1:8848` ，在左侧导航栏中单击**服务列表**，再在服务列表页面查看调用者服务。

   可以看到包含了 `com.alibaba.edas.boot.IHelloService` ，且可以查看该服务的服务分组和调用者IP。

结果验证

```
`curl http://localhost:8080/sayHello/SAE`
`Hello, SAE (from Dubbo with Spring Boot)`
```

部署到SAE

本地使用Nacos作为注册中心的应用，可以直接部署到SAE中，无需做任何修改，注册中心会被自动替换为SAE上的注册中心。

您可以根据实际需求选择部署途径（控制台或工具），详情请参见应用部署概述。

使用控制台部署前，请参见如下操作将应用程序编译为可运行的JAR包、WAR包。

1. 在 `pom.xml` 文件中添加以下打包插件的配置。
   - Provider

```
<build>
 <plugins>
       <plugin>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-maven-plugin</artifactId>
           <executions>
               <execution>
                   <goals>
                       <goal>repackage</goal>
                   </goals>
                   <configuration>
                       <classifier>spring-boot</classifier>
                       <mainClass>com.alibaba.edas.boot.DubboProvider</mainClass>
                   </configuration>
               </execution>
           </executions>
       </plugin>
 </plugins>
</build>
```

   - Consumer

```
<build>
 <plugins>
       <plugin>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-maven-plugin</artifactId>
           <executions>
               <execution>
                   <goals>
                       <goal>repackage</goal>
                   </goals>
                   <configuration>
                       <classifier>spring-boot</classifier>
                       <mainClass>com.alibaba.edas.boot.DubboConsumer</mainClass>
                   </configuration>
               </execution>
           </executions>
       </plugin>
 </plugins>
</build>
```

2. 执行**mvn clean package**将本地的程序打成JAR包。

3.使用HSF开发应用

3.1. HSF概述

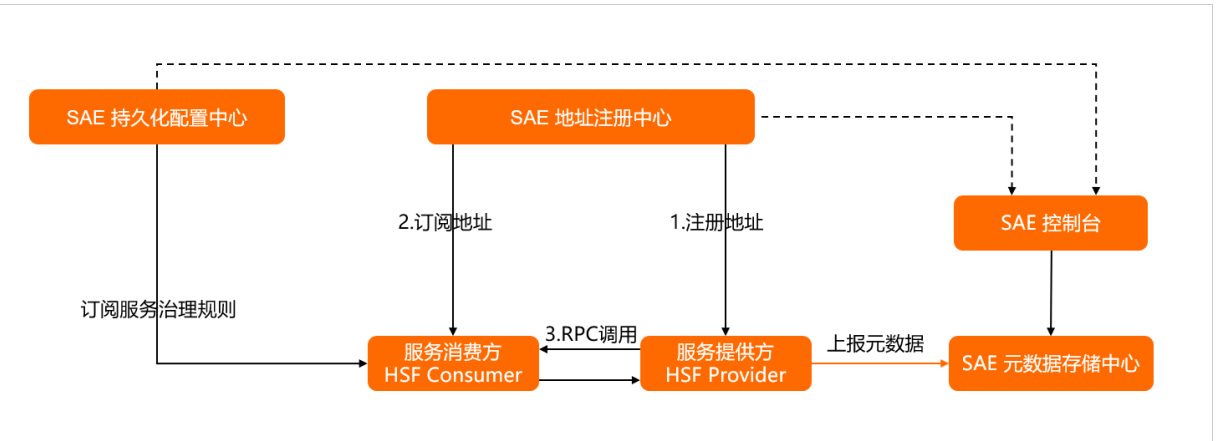
高速服务框架HSF（High-speed Service Framework），是在阿里巴巴广泛使用的分布式RPC服务框架。

概述

HSF连通不同的业务系统，解耦系统间的实现依赖。HSF从分布式应用的层面，统一了服务的发布与调用方式，从而帮助用户更加方便、快速地开发分布式应用，以及提供或使用公共功能模块。HSF为用户屏蔽了分布式领域中的各种复杂技术细节，如远程通讯、序列化实现、性能损耗、同步与异步调用方式的实现等。

HSF架构

HSF作为一个纯客户端架构的RPC框架，没有服务端集群，所有HSF服务调用均是通过服务消费方（Consumer）与服务提供方（Provider）点对点进行。为了实现整套分布式服务体系，HSF还需要依赖以下外部系统。



- 服务提供方

  服务提供方绑定了12200端口，用于接受请求并提供服务，同时将地址信息发布到地址注册中心。

- 服务消费方

  服务消费者通过地址注册中心订阅服务，根据订阅到的地址信息发起调用，地址注册中心不参与调用。

- SAE地址注册中心

  HSF依赖注册中心进行服务发现，如果没有注册中心，HSF只能完成简单的点对点调用。

  服务提供端无法将服务信息对外暴露，服务消费端可能已经明确了待调用的服务，但是无法获取该服务。因此注册中心是服务信息的中介，为服务提供了注册与发现的功能。

- SAE持久化配置中心

  持久化的配置中心用于存储HSF服务的各种治理规则，HSF客户端在启动的过程中向持久化配置中心订阅服务治理规则，如路由规则、归组规则、权重规则等，从而根据规则对调用过程的选址逻辑进行干预。

- SAE元数据存储中心

  元数据指HSF服务对应的方法列表以及参数结构等信息。元数据对HSF的调用过程不会产生影响，因此元数据存储中心是可选的。由于服务运维的便捷性，HSF客户端在启动时会将元数据上报到元数据存储中心，方便服务运维。

- SAE控制台

  SAE控制台打通了服务地址注册中心、持久化配置中心、元数据存储中心等，为用户提供了服务运维功能，包括服务查询、服务治理规则管理等，提高HSF服务研发的效率、运维的便捷性。

功能

HSF作为分布式RPC服务框架，支持多种服务的调用方式。

> **说明** Dubbo 3.0实现了和HSF框架的技术统一。在EDAS中，可以便捷的将HSF应用升级为Dubbo 3.0应用。升级之后，HSF应用可沿用原有开发方式，还可以使用EDAS为Dubbo应用提供的更加完善的服务治理功能。更多信息，请参见将HSF应用升级为Dubbo 3.0应用。

- 同步调用

  HSF客户端默认以同步调用的方式消费服务，客户端代码需要同步等待返回结果。

- 异步调用

  对于服务调用的客户端，并非所有HSF服务都需要同步等待返回结果。HSF提供异步调用，帮助客户端无需同步阻塞在HSF调用上。HSF的异步调用，有Future调用和Callback调用2种。

  - Future调用

    客户端在需要获取调用的返回结果时，通过HSFResponseFuture.getResponse（int timeout）主动获取结果。

  - Callback调用

    Callback调用利用HSF内部提供的回调机制，在指定HSF服务消费完毕拿到返回结果时，HSF框架会回调用户实现的 `HSFResponseCallback` 接口，客户端通过回调通知的方式获取结果。

- 泛化调用

  对于一般的HSF调用来说，HSF客户端需要依赖服务的二方包，通过依赖二方包中的API进行编程调用，获取返回结果。但是泛化调用不需要依赖服务的二方包，可以发起HSF调用、获取返回结果。在平台型的产品中，泛化调用的方式可以有效减少平台型产品的二方包依赖，实现系统的轻量级运行。

- 调用链路Filter扩展

  HSF内部设计了调用过滤器，能够主动发现用户的调用过滤器扩展点，将其集成到HSF调用链路中，便于扩展方对HSF的请求进行扩展处理。

应用开发方式

使用HSF框架开发应用有Ali-Tomcat和Pandora Boot两种方式。

- Ali-Tomcat：依赖Ali-Tomcat和Pandora，提供了完整的HSF功能，包括服务注册与发现、隐式传参、异步调用、泛化调用和调用链路Filter扩展。应用程序须以WAR包方式部署。
- Pandora Boot：依赖Pandora，提供了比较完整的HSF功能，包括服务注册与发现、异步调用。应用程序编译为可运行的JAR包并部署即可。

3.2. 启动轻量级配置及注册中心

开发者可以在本地使用轻量级配置及注册中心实现应用的注册、发现和配置管理，完成应用的开发和测试。在将应用部署到SAE后，这些功能仍然可以正常使用。本文介绍如何下载、启动和验证轻量级配置及注册中心。

前提条件

在使用轻量级配置及注册中心前，请完成以下工作：

- 下载1.8及以上版本的JDK，并设置环境变量 `JAVA_HOME` 。
- 确认8080、8848和9600端口未被使用。

> **说明** 由于轻量级配置及注册中心将占用8080、8848和9600端口，因此建议使用专门的机器安装并启动轻量级配置及注册中心。如果在本机上使用，请将应用的端口修改为其它端口。

步骤一：下载轻量级配置及注册中心

- Windows：
  i. 下载轻量级配置及注册中心压缩包
  ii. 在本地解压压缩包。
- Unix：
  i. 执行命令 `wget http://edas.oss-cn-hangzhou.aliyuncs.com/edas-res/edas-lightweight-server-1.0.0.tar.gz` 下载轻量级配置及注册中心压缩包。
  ii. 执行命令 `tar -zvxf edas-lightweight-server-1.0.0.tar.gz` 解压压缩包。

> **注意** 轻量级配置及注册中心仅用于本地开发、测试，请勿用于生产环境。如果需要暴露到公网，请控制好IP访问策略。

步骤二：启动轻量级配置及注册中心

1. 进入目录*edas-lightweight\bin*。
2. 启动轻量级配置及注册中心，并查看启动结果。
   - Windows：双击*startup.bat*。

   - Unix：执行*sh startup.sh*。

```
1. /usr/java/jdk1.8.0/bin/java -server -Xms1g -Xmx1g -Xmn512m -XX:MetaspaceSize=128m
-XX:MaxMetaspaceSize=320m -XX:-OmitStackTraceInFastThrow -XX:+HeapDumpOnOutOfMemoryEr
ror -XX:HeapDumpPath=/root/service/edas-lightweight/logs/java_heapdump.hprof -XX:-Use
LargePages -Djava.ext.dirs=/usr/java/jdk1.8.0/jre/lib/ext:/usr/java/jdk1.8.0/lib/ext:
/root/service/edas-lightweight/plugins/cmdb:/root/service/edas-lightweight/plugins/my
sql -Xloggc:/root/service/edas-lightweight/logs/nacos_gc.log -verbose:gc -XX:+PrintGC
Details -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation -XX:N
umberOfGCLogFiles=10 -XX:GCLogFileSize=100M -Dnacos.home=/root/service/edas-lightweig
ht -Dnacos.standalone=true -jar /root/service/edas-lightweight/target/edas-lightweigh
t.jar --spring.config.location=classpath:/,classpath :/config/,file:./,file:./config/
,file:/root/service/edas-lightweight/conf/ --logging.config=/root/service/edas-lightw
eight/conf/nacos-logback.xml --server.max-http-header-size=524288
2. edas lightweight is starting with standalone
3. edas lightweight is starting, you can check the /root/edas-lightweight/logs/start.
out
```

> **说明** 如果需要调整启动的JVM参数，根据您本地的环境在启动脚本中设置合适的JVM参数。

3. （可选）检查轻量级配置及注册中心的启动监听端口。

   正常启动后，会在本节点上监听以下三个端口：

   - 8848：用来支持基于Nacos应用的配置管理及服务注册。
   - 9600：HSF或Dubbo的服务注册及订阅。
   - 8080：支持服务注册以及配置管理。

   Linux和macOS操作系统环境，可使用 `netstat -an | grep -E "8080|8848|9600" | grep -i listen` 命令查看上述三个端口是否处在监听状态（Linux操作系统还可以使用 `netstat -nltp | grep -E "8080|8848|9600"` 来检查这三个端口的打开情况以及是否是轻量级配置及注册中心的进程打开的）。

步骤三：在本地开发环境中配置hosts

在需要使用轻量级配置及注册中心开发、测试应用的机器上配置轻量级配置及注册中心的hosts，即在DNS（hosts文件）中将 `jmenv.tbsite.net` 域名指向启动了轻量级配置及注册中心的机器IP。

1. 打开hosts文件。
   - Windows操作系统：C:\Windows\System32\drivers\etc\hosts
   - Unix操作系统：/etc/hosts
2. 添加轻量级配置及注册中心配置。
   - 如果在IP为192.168.1.100的机器上启动了轻量级配置及注册中心，则需要在hosts文件里加入配置：`192.168.1.100 jmenv.tbsite.net` 。
   - 如果在本地启动轻量级配置及注册中心，则在hosts文件中配置将上面的IP改为 `127.0.0.1 jmenv.tbsite.net` 。

结果验证

轻量级配置及注册中心的验证包含两部分。

- 验证轻量级配置及注册中心可用性。

  轻量级配置及注册中心可以在本机或独立机器上启动，所以访问会有两种方式。

- 本机

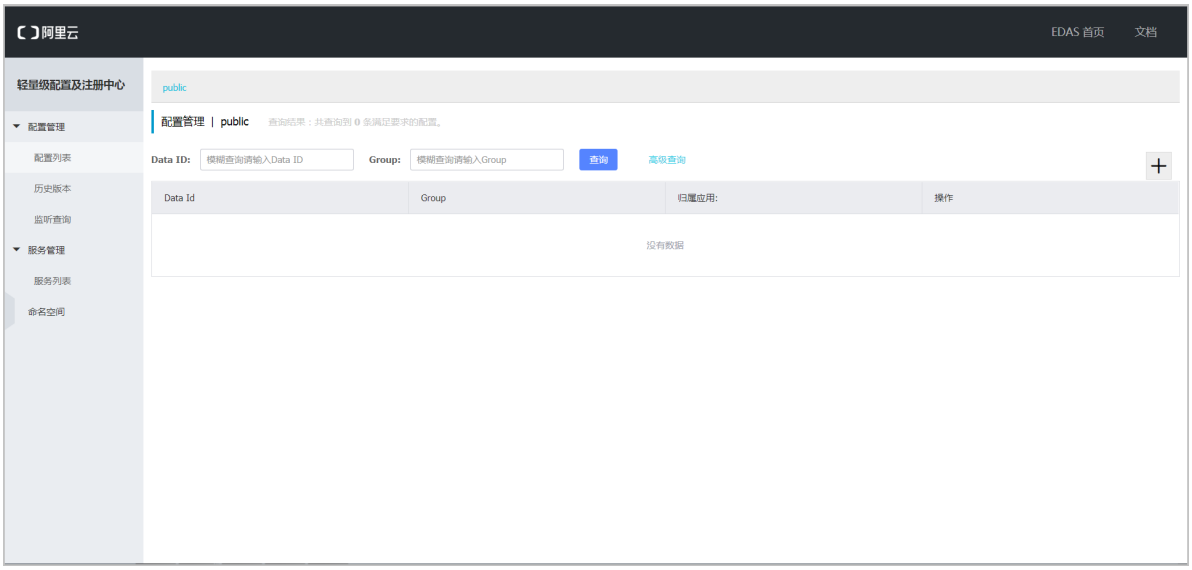
  在浏览器中输入轻量级配置及注册中心地址 `http://127.0.0.1:8080` 并回车。

- 独立机器

  在浏览器中输入轻量级配置及注册中心地址 `http://机器IP地址:8080` 并回车。

> **说明** 绑定hosts之后，可以直接访问轻量级配置及注册中心域名 + 端口 `jmenv.tbsite.net:8080` 。

轻量级配置及注册中心首页如下图所示。



如果首页不能正常显示，可以查看安装目录下的启动日志文件 `logs/start.log` 定位启动失败的原因，并修复。

- 验证功能可用性。

  轻量级配置及注册中心提供了服务注册、发现、配置管理和命名空间功能（仅适用于原有Nacos用户）。

  有些用户之前使用了轻量级配置中心或Nacos，有些用户初次使用轻量级配置及注册中心，所以验证分为原有用户和新用户两种场景。

  - 原有用户在下载、启动轻量级配置及注册中心之后，可以根据业务逻辑直接验证功能可用性。
  - 新用户在下载、启动轻量级配置及注册中心之后，还需要在应用中增加、修改配置，建议参考具体功能的应用开发文档验证功能可用性。

3.3. 将HSF应用托管到SAE

本文介绍如何将HSF应用托管到SAE。

背景信息

使用HSF框架开发应用有以下方式：

- Ali-Tomcat：依赖Ali-Tomcat和Pandora，提供了完整的HSF功能，包括服务注册与发现、隐式传参、异步调用、泛化调用和调用链路Filter扩展。应用程序须以WAR包方式部署。
- Pandora Boot：依赖Pandora，提供了比较完整的HSF功能，包括服务注册与发现、异步调用。应用程序

编译为可运行的JAR包并部署即可。

操作步骤

1. 开发应用。
   - 方法一：下载Demo工程。

     microservice-doc-demo项目内包含了众多示例工程，本文以其中通过Ali-Tomcat方式开发的HSF应用为例。具体信息，请参见hsf-ali-tomcat。
   - 方法二：本地构建工程。

     具体操作，请参见以下文档：
     - 使用Ali-Tomcat开发应用
     - 使用Pandora Boot开发应用
2. 本地开发调试。

   具体操作，请参见启动轻量级配置及注册中心。
3. 将应用打包成WAR包或JAR包，并部署到SAE。

   具体操作，请参见以下文档：
   - 在SAE控制台使用WAR包部署Java Web应用
   - 在SAE控制台使用JAR文件部署微服务应用

3.4. 使用Ali-Tomcat开发应用

3.4.1. Ali-Tomcat概述

开发HSF应用需要依赖Ali-Tomcat和Pandora。

Ali-Tomcat是SAE中的服务运行时可依赖的一个容器，它主要集成了服务的发布、订阅、调用链追踪等一系列的核心功能。无论是开发环境还是运行时，您均可将应用程序发布在该容器中。

Pandora是一个轻量级的隔离容器，也就是taobao-hsf.sar。它用来隔离应用和中间件的依赖，也用来隔离中间件之间的依赖。SAE的Pandora中集成了服务发现、配置推送和调用链跟踪等各种中间件功能产品插件。您可以利用该插件对SAE应用进行服务监控、治理、跟踪、分析等全方位运维管理。

> **注意** 在SAE中，只有WAR包格式的HSF应用才需要使用Ali-Tomcat。

3.4.2. 安装及开发环境配置

本文介绍如何安装Ali-Tomcat和Pandora，以及如何配置Eclipse和IntelliJ IDEA的开发环境。

安装Ali-Tomcat和Pandora

Ali-Tomcat和Pandora为SAE中的服务运行时所依赖的容器，集成了服务的发布、订阅、调用链追踪等一系列核心功能，应用程序须发布在该容器中运行。

> **注意** 请使用JDK 1.7及以上版本。

1. 下载Ali-Tomcat-8或Ali-Tomcat-7，保存并解压至相应的目录（如：d:\work\tomcat\）。

2. 下载Pandora容器，保存并解压至Ali-Tomcat的deploy目录（d:\work\tomcat\deploy）下。
3. 查看Pandora容器的目录结构。
   - Linux系统中，在相应路径下执行**tree -L 2 deploy/** 命令查看目录结构。

```
d:\work\tomcat >  tree -L 2 deploy/
            deploy/
                └── taobao-hsf.sar
                    ├── META-INF
                    ├── lib
                    ├── log.properties
                    ├── plugins
                    ├── sharedlib
                    └── version.properties
```

   - Windows中，直接进入相应路径进行查看。

   如果您在安装和使用Ali-Tomcat和Pandora过程中遇到问题，请参见Ali-Tomcat问题和Pandora问题。

配置Eclipse开发环境

配置Eclipse需要下载Tomcat4E插件，并存放在安装Ali-Tomcat和Pandora的保存路径中，完成配置后可以直接在Eclipse中发布、调试本地代码。

1. 下载Tomcat4E插件，压缩包内容如下图所示。

2. 打开Eclipse，在菜单栏中选择**Help > Install New Software**。
3. 在Install对话框中**Work with**区域右侧单击**Add**，且在弹出的**Add Repository**对话框中单击**Local**，并在弹出的对话框中选中已下载并解压的Tomcat4E插件的目录（d:\work\tomcat4e\），单击**OK**。
4. 返回Install对话框，单击**Select All**，并单击**Next**。

   后续步骤，请按界面提示操作。安装完成后，请重启Eclipse，使Tomcant4E插件生效。
5. 重启Eclipse后，在Eclipse菜单中选择**Run As > Run Configurations**。
6. 选择左侧导航选项中的**AliTomcat Webapp**，单击上方的**New launch configuration**图标。
7. 在弹出的界面中，选择**AliTomcat**页签，并在**taobao-hsf.sar Location**区域单击**Browse**，选择本地的Pandora路径，如：*d:\work\tomcat\deploy\taobao-hsf.sar*。
8. 单击**Apply**或**Run**，完成设置。

一个工程只需配置一次，下次可直接启动。

9. 查看工程运行的打印信息，如果出现下图Pandora Container的相关信息，即说明Eclipse开发环境配置成功。

```
********************************************************************************************
**                                                                                        **
**                                    Pandora Container                                   **
**                                                                                        **
**      Pandora Host:      [redacted]                                                      **
**      Pandora Version:   2.1.4                                                           **
**      SAR Version:       edas.sar.V3.3.4.dev                                             **
**      Package Time:      2017-11-20 09:58:18                                             **
**                                                                                        **
**      Plug-in Modules: 11                                                               **
**                                                                                        **
**         edas-assist ..................................... 1.6                          **
**         pandora-qos-service ............................. edas215                      **
**         spas-sdk-client ................................. 1.2.4-SNAPSHOT               **
**         eagleeye-core ................................... 1.6.0.2-SNAPSHOT             **
**         vipserver-client ................................ 4.6.8-SNAPSHOT               **
**         diamond-client .................................. acm-3.8.5                    **
**         spas-sdk-service ................................ 1.2.4.nodc-SNAPSHOT          **
**         config-client ................................... 2.0.1-edas-SNAPSHOT          **
**         unitrouter ...................................... 1.0.11                       **
**         sentinel-plugin ................................. 2.12.2_edas                  **
**         hsf ............................................. 2.2.4.2                      **
**                                                                                        **
**      [WARNING] All these plug-in modules will override maven pom.xml dependencies.     **
**      More: http://gitlab.alibaba-inc.com/middleware-container/pandora/wikis/home       **
**                                                                                        **
********************************************************************************************
```

配置IntelliJ IDEA开发环境

> **注意** 目前仅支持IDEA商业版，社区版暂不支持。

1. 运行IntelliJ IDEA。
2. 在菜单栏中选择**Run > Edit Configurations**。
3. 在**Run/Debug Configurations**页面左侧的导航栏中选择**Defaults > Tomcat Server > Local**。
4. 配置AliTomcat。
   i. 在右侧页面单击**Server**页签，并在**Application Server**区域单击**Configure**。
   ii. 在**Application Server**页面右上角单击**+**，并在**Tomcat Server**对话框中设置**Tomcat Home**和**Tomcat base directory** 路径，且单击**OK**。

   将Tomcat Home的路径设置为本地解压后的Ali-Tomcat路径，Tomcat base directory可以自动使用该路径，无需再设置。
5. 在**Application Server**区域的下拉菜单中，选择刚刚配置好的Ali-Tomcat。
6. 在**VM Options**区域的文本框中，设置JVM启动参数指向Pandora的路径及设置本地轻量级配置及注册中心的vipserver端口。

   例如：-Dpandora.location=d:\work\tomcat\deploy\taobao-hsf.sar -Dvipserver.server.port=8080

   将*d:\work\tomcat\deploy\taobao-hsf.sar*替换为在本地安装Pandora的实际路径。 -Dvipserver.server.port=8080 该参数用于指定应用连接本地轻量级配置及注册中心的vipserver端口，在本地开发环境使用新版本的轻量级配置及注册中心时，能有效提升本地应用启动速度。

当应用部署到EDAS时，不需要添加上面这个JVM属性参数。

7. 单击Apply或OK 完成配置。

3.4.3. 开发HSF应用（SDK）

介绍如何使用SDK快速开发HSF应用，完成服务注册与发现。

前提条件

在开发应用前，您已经完成以下工作：

- 配置EDASSAE的私服地址和轻量级配置及注册中心
- 启动轻量级配置及注册中心

下载Demo工程

您可以按照本文的步骤一步步搭建工程，也可以直接下载本文对应的示例工程，或者使用Git下载：**git clone https://github.com/aliyun/alibabacloud-microservice-demo.git**。

该项目包含了众多示例工程，本文对应的示例工程位于*alibabacloud-microservice-demo/microservice-doc-demo/hsf-ali-tomcat*，包含*itemcenter-api*，*itemcenter*和*detail*三个Maven工程文件夹。

- *itemcenter-api*：提供接口定义
- *itemcenter*：服务提供者
- *detail*：消费者服务

> **说明** 请使用JDK 1.7及以上版本。

定义服务接口

HSF服务基于接口实现，当接口定义好之后，生产者将使用该接口实现具体的服务，消费者也基于此接口去订阅服务。

在Demo的*itemcenter-api*工程中，定义了一个服务接口 `com.alibaba.edas.carshop.itemcenter.ItemService`。

```
public interface ItemService {
    public Item getItemById(long id);
    public Item getItemByName(String name);
}
```

该服务接口将提供两个方法：getItemById与getItemByName。

开发服务提供者

服务提供者将实现服务接口以提供具体服务。同时，如果使用了Spring框架，还需要在xml文件中配置服务属性。

> **说明** Demo工程中的itemcenter文件夹为服务提供者的示例代码。

1. 实现服务接口。

   请参考*ItemServiceImpl.java*文件中的示例代码构建服务接口。

```
public class ItemServiceImpl implements ItemService {
    @Override
    public Item getItemById( long id ) {
        Item car = new Item();
        car.setItemId( 1l );
        car.setItemName( "Mercedes Benz" );
        return car;
    }
    @Override
    public Item getItemByName( String name ) {
        Item car = new Item();
        car.setItemId( 1l );
        car.setItemName( "Mercedes Benz" );
        return car;
    }
}
```

2. 服务提供者配置。

   实现服务接口中实现了com.alibaba.edas.carshop.itemcenter.ItemService，并在两个方法中返回了Item对象。代码开发完成之后，除了在web.xml中进行必要的常规配置，您还需要增加相应的Maven依赖，同时在Spring配置文件使用 `<hsf />` 标签注册并发布该服务。

   i. 在*pom.xml*中添加Maven依赖。

   ```
   <dependencies>
       <!-- 添加servlet的依赖 -->
       <dependency>
           <groupId>javax.servlet</groupId>
           <artifactId>servlet-api</artifactId>
           <version>2.5</version>
           <scope>provided</scope>
       </dependency>
       <!-- 添加服务接口的依赖 -->
       <dependency>
           <groupId>com.alibaba.edas.carshop</groupId>
           <artifactId>itemcenter-api</artifactId>
           <version>1.0.0-SNAPSHOT</version>
       </dependency>
       <!-- 添加Spring的依赖 -->
       <dependency>
           <groupId>org.springframework</groupId>
           <artifactId>spring-web</artifactId>
           <version>2.5.6(及其以上版本)</version>
       </dependency>
       <!-- 添加edas-sdk的依赖 -->
       <dependency>
             <groupId>com.alibaba.edas</groupId>
              <artifactId>edas-sdk</artifactId>
             <version>1.8.1</version>
       </dependency>
   </dependencies>
   ```

   ii. 在*hsf-provider-beans.xml*文件中增加Spring关于HSF服务的配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:hsf="http://www.taobao.com/hsf"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.taobao.com/hsf
    http://www.taobao.com/hsf/hsf.xsd" default-autowire="byName">
    <!-- 定义该服务的具体实现 -->
    <bean id="itemService" class="com.alibaba.edas.carshop.itemcenter.ItemService
Impl" />
    <!-- 用hsf:provider标签表明提供一个服务生产者 -->
    <hsf:provider id="itemServiceProvider"
        <!-- 用interface属性说明该服务为此类的一个实现 -->
        interface="com.alibaba.edas.carshop.itemcenter.ItemService"
        <!-- 此服务具体实现的Spring对象 -->
        ref="itemService"
        <!-- 发布该服务的版本号，可任意指定，默认为1.0.0 -->
        version="1.0.0"
    </hsf:provider>
</beans>
```

上面的示例为基本配置，您也可以根据您的实际需求，参考下面的生产者服务属性列表，增加其它配置。

属性	描述
interface	必须配置，类型为 [String]，为服务对外提供的接口。
version	可选配置，类型为 [String]，含义为服务的版本，默认为1.0.0。
clientTimeout	该配置对接口中的所有方法生效，但是如果客户端通过methodSpecials属性对某方法配置了超时时间，则该方法的超时时间以客户端配置为准。其他方法不受影响，还是以服务端配置为准。
serializeType	可选配置，类型为 `[String(hessian|java)]`，含义为序列化类型，默认为hessian。
corePoolSize	单独针对这个服务设置核心线程池，从公用线程池中划分出来。
maxPoolSize	单独针对这个服务设置线程池，从公用线程池中划分出来。
enableTXC	开启分布式事务GTS。
ref	必须配置，类型为 [ref]，为需要发布为HSF服务的Spring Bean ID。

属性	描述
methodSpecials	可选配置，用于为方法单独配置超时时间（单位ms），这样接口中的方法可以采用不同的超时时间。该配置优先级高于上面的clientTimeout的超时配置，低于客户端的methodSpecials配置。

服务创建及发布存在以下限制：

名称	示例	限制大小	是否可调整
{服务名}:{版本号}	com.alibaba.edas.testcase.api.TestCase:1.0.0	最大192字节	否
组名	HSF	最大32字节	否
单个Pandora应用实例发布的服务数	N/A	最大800个	可在应用基本信息页面单击**应用设置**部分右侧的设置，在下拉列表中选择**JVM**，在弹出的**应用设置**对话框中进入**自定义 > 自定义参数**，`-DCC.pubCountMax=1200`属性参数（该参数值可根据应用实际发布的服务数调整）。

服务提供者属性配置示例：

```
<bean id="impl" class="com.taobao.edas.service.impl.SimpleServiceImpl" />
<hsf:provider
        id="simpleService"
        interface="com.taobao.edas.service.SimpleService"
        ref="impl"
        version="1.0.1"
        clientTimeout="3000"
        enableTXC="true"
        serializeType="hessian">
        <hsf:methodSpecials>
            <hsf:methodSpecial name="sum" timeout="2000" />
        </hsf:methodSpecials>
</hsf:provider>
```

开发服务消费者

消费者订阅服务从代码编写的角度分为两个部分。

- Spring的配置文件使用标签`<hsf:consumer/>`定义好一个Bean。
- 在使用的时候从Spring的context中将Bean取出来。

> **说明** Demo工程中的detail文件夹为消费者服务的示例代码。

与生产者相同，消费者的服务属性配置分为Maven依赖配置与Spring的配置。

1. 配置服务属性。
   i. 在*pom.xml*文件中添加Maven依赖。
   ii. 在*hsf-consumer-beans.xml*文件中添加Spring关于HSF服务的配置。

   增加消费者的定义，HSF框架将根据该配置文件去服务中心订阅所需的服务。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:hsf="http://www.taobao.com/hsf"
        xmlns="http://www.springframework.org/schema/beans"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.taobao.com/hsf
        http://www.taobao.com/hsf/hsf.xsd" default-autowire="byName">
        <!-- 消费一个服务示例 -->
        <hsf:consumer
            <!-- Bean ID，在代码中可根据此ID进行注入从而获取consumer对象  -->
            id="item"
            <!-- 服务名，与服务提供者的相应配置对应，HSF将根据interface + version查询并订
阅所需服务  -->
            interface="com.alibaba.edas.carshop.itemcenter.ItemService"
            <!-- 版本号，与服务提供者的相应配置对应，HSF将根据interface + version查询并订
阅所需服务  -->
            version="1.0.0">
        </hsf:consumer>
</beans>
```

2. 服务消费者配置。

   请参考*StartListener.java*文件中的示例进行。

```
public class StartListener implements ServletContextListener{
    @Override
    public void contextInitialized( ServletContextEvent sce ) {
        ApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext( s
ce.getServletContext() );
        // 根据Spring配置中的Bean ID“item” 获取订阅到的服务final ItemService itemService =
( ItemService ) ctx.getBean( "item" );
        ……
        // 调用服务ItemService的getItemById方法System.out.println( itemService.getItemByI
d( 1111 ) );
        // 调用服务ItemService的getItemByName方法System.out.println( itemService.getItemB
yName( "myname is le" ) );
        ……
    }
}
```

上面的示例中为基本配置，您也可以根据您的实际需求，参考下面的服务属性列表，增加其它配置。

属性	描述
interface	必须配置，类型为 [String]，为需要调用的服务的接口。

属性	描述
version	可选配置，类型为 [String]，为需要调用的服务的版本，默认为1.0.0。
methodSpecials	可选配置，为方法单独配置超时时间（单位ms）。这样接口中的方法可以采用不同的超时时间，该配置优先级高于服务端的超时配置。
target	主要用于单元测试环境和开发环境中，手动地指定服务提供端的地址。如果不想通过此方式，而是通过配置中心推送的目标服务地址信息来指定服务端地址，可以在消费者端指定 -Dhsf.run.mode=0。
connectionNum	可选配置，为支持设置连接到server连接数，默认为1。在小数据传输，要求低延迟的情况下设置多一些，会提升TPS。
clientTimeout	客户端统一设置接口中所有方法的超时时间（单位ms）。超时时间设置优先级由高到低是：客户端methodSpecials，客户端接口级别，服务端methodSpecials，服务端接口级别 。
asyncallMethods	可选配置，类型为 [List]，设置调用此服务时需要采用异步调用的方法名列表以及异步调用的方式。默认为空集合，即所有方法都采用同步调用。
maxWaitTimeForCsAddress	配置该参数，目的是当服务进行订阅时，会在该参数指定时间内，阻塞线程等待地址推送，避免调用该服务时因为地址为空而出现地址找不到的情况。若超过该参数指定时间，地址还是没有推送，线程将不再等待，继续初始化后续内容。注意，在应用初始化时，需要调用某个服务时才使用该参数。如果不需要调用其它服务，请勿使用该参数，会延长启动时间。

消费者服务属性配置示例

```
<hsf:consumer
      id="service"
      interface="com.taobao.edas.service.SimpleService"
      version="1.1.0"
      clientTimeout="3000"
      target="10.1.6.57:12200?_TIMEOUT=1000"
      maxWaitTimeForCsAddress="5000">
      <hsf:methodSpecials>
           <hsf:methodSpecial name="sum" timeout="2000" ></hsf:methodSpecial>
      </hsf:methodSpecials>
</hsf:consumer>
```

本地运行服务

完成代码、接口开发和服务配置后，在Eclipse或IDEA中，可直接以Ali-Tomcat运行该服务（具体请参见安装及开发环境配置）。

在开发环境配置时，有一些额外JVM启动参数来改变HSF的行为，具体如下：

属性	描述
-Dhsf.server.port	指定HSF的启动服务绑定端口，默认值为12200。
-Dhsf.serializer	指定HSF的序列化方式，默认值为hessian。
-Dhsf.server.max.poolsize	指定HSF的服务端最大线程池大小，默认值为720。
-Dhsf.server.min.poolsize	指定HSF的服务端最小线程池大小。默认值为50。
-DHSF_SERVER_PUB_HOST	指定对外暴露的IP，如果不配置，使用 -Dhsf.server.ip的值。
-DHSF_SERVER_PUB_PORT	指定对外暴露的端口，该端口必须在本机被监听，并对外开放了访问授权，默认使用 -Dhsf.server.port的配置，如果 -Dhsf.server.port没有配置，默认使用12200。

本地查询HSF服务

在开发调试的过程中，如果您的服务是通过轻量级注册配置中心进行服务注册与发现，就可以通过EDAS控制台查询某个应用提供或调用的服务。

假设您在一台IP为192.168.1.100的机器上启动了EDAS配置中心。

- 进入http://192.168.1.100:8080/
- 在左侧菜单栏单击**服务列表**，输入服务名、服务组名或者IP地址进行搜索，查看对应的服务提供者以及服务调用者。

> **说明** 配置中心启动之后默认选择第一块网卡地址做为服务发现的地址，如果开发者所在的机器有多块网卡的情况，可设置启动脚本中的SERVER_IP变量进行显式的地址绑定。

常见查询案例

- 提供者列表页
  - 在搜索框中输入IP地址，单击**搜索**，即可查询该IP地址的物理机所提供的服务。
  - 在搜索框中输入服务名或服务分组，即可查询提供该服务的IP地址。
- 调用者列表页
  - 在搜索框中输入IP地址，单击**搜索**，即可查询该IP地址的物理机所调用的服务。
  - 在搜索框中输入服务名或服务分组，即可查询调用该服务的IP地址。

部署到SAE

本地使用轻量级配置及注册中心的应用可以直接部署到SAE中，无需做任何修改，注册中心会被自动替换为SAE上的注册中心。

正常打包出可供EDAS-Container运行的WAR包，需要添加如下的Maven打包插件

1. 在*pom.xml*文件中添加以下打包插件的配置。

```
<build>
    <finalName>itemcenter</finalName>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
        </plugin>
    </plugins>
</build>
```

2. 执行**mvn clean package**将本地的程序打成WAR包。

应用运行时环境需要选择EDAS-Container。

具体部署操作请参见应用部署概述。

3.5. 使用Pandora Boot开发应用

3.5.1. Pandora Boot概述

Pandora Boot是在Pandora的基础之上，发展出的更轻量地使用Pandora的一种方式。

- Pandora Boot基于Pandora和Fat Jar技术，可以直接在IDE里启动Pandora环境，大大提高您的开发调试效率。
- Pandora Boot与Spring Boot AutoConfigure深度集成，让您同时可以享受Spring Boot框架带来的便利。

基于Pandora Boot来开发部署在SAE上的应用，适用于需要使用HSF的Spring Boot用户以及已经使用过Pandora Boot的用户。

3.5.2. 配置SAE的私服地址和轻量级配置及注册中心

使用Pandora Boot开发HSF应用前，需要先配置SAE的私服地址和轻量级配置及注册中心。

背景信息

- 目前Spring Cloud for Aliware的第三方包只发布在SAE的私服中，所以需要在Maven中配置SAE的私服地址。
- 本地开发调试时，需要启动轻量级配置注册中心。轻量级配置及注册中心包含了服务发现和配置管理功能。

在Maven中配置SAE的私服地址

> **说明** Maven要求3.x及后续版本。在Maven配置文件*settings.xml*中加入SAE私服地址。

1. 请参考如下示例，在Maven所使用的配置文件（一般为*~/.m2/settings.xml*）中添加SAE的私服配置。

```
<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>edas.oss.repo</id>
        <repositories>
            <repository>
                <id>edas-oss-central</id>
                <name>taobao mirror central</name>
                <url>https://edas-public.oss-cn-hangzhou.aliyuncs.com/repository</url>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
                <releases>
                    <enabled>true</enabled>
                </releases>
            </repository>
            </repositories>
        <pluginRepositories>
            <pluginRepository>
                <id>edas-oss-plugin-central</id>
                <url>https://edas-public.oss-cn-hangzhou.aliyuncs.com/repository</url>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
                <releases>
                    <enabled>true</enabled>
                </releases>
            </pluginRepository>
        </pluginRepositories>
    </profile>
</profiles>
<activeProfiles>
    <activeProfile>nexus</activeProfile>
    <activeProfile>edas.oss.repo</activeProfile>
</activeProfiles>
```

```
</activeProfiles>
```

2. 在命令行执行命令**mvn help:effective-settings**，验证配置是否成功。
   - 无报错，表明*setting.xml*文件格式没问题。
   - profiles中包含edas.oss.repo这个profile，表明私服已经配置到profiles中。
   - activeProfiles中包含edas.oss.repo属性，表明edas.oss.repo私服已激活。

> **说明** 如果在命令行执行Maven打包命令无问题，IDE仍无法下载依赖，请关闭IDE重新打开再尝试，或自行查找IDE配置Maven的相关资料。

配置轻量级配置及注册中心

具体操作步骤请参见启动轻量级配置及注册中心。

3.5.3. 开发RESTful应用（不推荐）

在HSF框架中开发RESTful应用，并实现服务注册与发现。由于SAE已经支持原生Spring Cloud框架的应用，新用户不推荐使用该开发方式。

背景信息

原生Spring Cloud框架下的服务开发请参考将Spring Cloud应用托管到SAE。

服务注册与发现

通过一个简单的示例详细介绍如何在本地开发RESTful应用并实现注册与发现。

Demo源码下载：

- sc-vip-server
- sc-vip-client

1. 创建服务提供者。

   此服务提供者提供了一个简单的echo服务，并将自身注册到服务发现中心。

   i. 创建命名为*sc-vip-server*的RESTful应用工程。

   ii. 在*pom.xml*中添加需要的依赖。

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-vipclient</artifactId>
            <version>1.3</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-pandora</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Dalston.SR4</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

如果您的工程不想将parent设置为 `spring-boot-starter-parent` ，也可以通过添加 `dependencyManagement` 并设置 `scope=import` 来达到依赖管理的效果。

```
<dependencyManagement>
            <dependencies>
                <dependency>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-dependencies</artifactId>
                    <version>1.5.8.RELEASE</version>
                    <type>pom</type>
                    <scope>import</scope>
                </dependency>
            </dependencies>
</dependencyManagement>
```

iii. 创建服务提供者应用，其中@EnableDiscoveryClient 注解表明此应用需开启服务注册与发现功能。

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

iv. 创建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;
        }
    }
```

v. 在*resources*下的*application.properties*文件中配置应用名与监听端口号。

```
spring.application.name=service-provider
server.port=18081
```

2. 创建服务消费者。

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

i. 创建名为*sc-vip-client*的RESTful应用工程。

ii. 在*pom.xml*中引入需要的依赖。

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
    </parent>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-vipclient</artifactId>
            <version>1.3</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-pandora</artifactId>
            <version>1.3</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-feign</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Dalston.SR4</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

本示例需要通过FeignClient进行演示，与sc-vip-server（服务提供者）相比，*pom.xml*文件中的依赖增加了 `spring-cloud-starter-feign` 。

iii. 与sc-vip-server相比，除了开启服务与注册外，还需要添加下面两项配置才能使用RestTemplate、AsyncRestTemplate、FeignClient这三个客户端。

- 添加@LoadBalanced注解将RestTemplate和AsyncRestTemplate与服务发现结合。
- 使用@EnableFeignClients注解激活FeignClient。

```
@SpringBootApplication
    @EnableDiscoveryClient
    @EnableFeignClients
    public class ConsumerApplication {
        @LoadBalanced
        @Bean
        public RestTemplate restTemplate() {
            return new RestTemplate();
        }
        @LoadBalanced
        @Bean
        public AsyncRestTemplate asyncRestTemplate(){
            return new AsyncRestTemplate();
        }
        public static void main(String[] args) {
            PandoraBootstrap.run(args);
            SpringApplication.run(ConsumerApplication.class, args);
            PandoraBootstrap.markStartupAndWait();
        }
    }
```

iv. 在使用EchoService的FeignClient之前，还需要配置服务名以及方法对应的HTTP请求。在sc-vip-server工程中配置服务名service-provider。

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

v. 创建一个Controller用于调用测试。

```
@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://service-provider/echo/" + str, Stri
ng.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://service-provider/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);
   }
}
```

代码说明如下：

- `/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。

vi. 配置应用名以及监听端口号。

```
spring.application.name=service-consumer
server.port=18082
```

3. 本地开发调试。

i. 启动轻量级配置中心。

本地开发调试时，需要使用轻量级配置中心，轻量级配置中心包含了SAE服务注册发现服务端的轻量版，详情请参见启动轻量级配置及注册中心。

ii. 启动应用。

- IDE中启动

在IDE中启动，通过VM options配置启动参数 `-Dvipserver.server.port=8080` （注意：该参数仅在本地开发且使用轻量级配置中心时需要添加，当应用部署到SAE时，须移除此参数，否则会使应用无法正常发布或订阅），通过main方法直接启动。

轻量级配置中心与应用部署在不同的机器上，需要绑定hosts，详情请参见启动轻量级配置及注册中心。

- FatJar启动

  a. 添加FatJar打包插件。

  使用Maven将pandora-boot工程打包成FatJar， 需要在pom.xml中添加如下插件。为避免与其他打包插件发生冲突，请勿在build的plugin中添加其他FatJar插件。

```
<build>
    <plugin>
        <groupId>com.taobao.pandora</groupId>
            <artifactId>pandora-boot-maven-plugin</artifactId>
            <version>2.1.9.1</version>
            <executions>
               <execution>
                  <phase>package</phase>
                  <goals>
                      <goal>repackage</goal>
                  </goals>
               </execution>
             </executions>
    </plugin>
</build>
```

  b. 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。

  c. 通过Java命令启动应用。

```
java -Dvipserver.server.port=8080 -Dpandora.location=/Users/{$username}/.m2/r
epository/com/taobao/pandora/taobao-hsf.sar/dev-SNAPSHOT/taobao-hsf.sar-dev-S
NAPSHOT.jar  -jar sc-vip-server-0.0.1-SNAPSHOT.jar
```

  > **说明** `-Dpandora.location` 指定的路径必须是全路径，且必须放在sc-vip-server-0.0.1-SNAPSHOT.jar之前。

  启动服务，分别进行调用，可以看到调用都成功了。

```
➜  ~ curl http://localhost:18082/echo-rest/rest-test
rest-test%
➜  ~ curl http://localhost:18082/echo-async-rest/async-rest-test
async-rest-test%
➜  ~ curl http://localhost:18082/echo-feign/feign-test
feign-test%
```

4. 常见问题处理。

   - AsyncRestTemplate无法接入服务发现。

     AsyncRestTemplate接入服务发现的时间比较晚，需要在Dalston之后的版本才能使用，相关内容，请参见pull request。

   - FatJar打包插件冲突。

     为避免与其他打包插件发生冲突，请勿在build的plugin中添加其他FatJar插件。

   - 打包时可不可以不排除taobao-hsf.sar?

可以，但是不建议这么做。

通过修改pandora-boot-maven-plugin插件，把excludeSar设置为false，打包时就会自动包含taobao-hsf.sar。

```
<plugin>
    <groupId>com.taobao.pandora</groupId>
    <artifactId>pandora-boot-maven-plugin</artifactId>
    <version>2.1.9.1</version>
    <configuration>
    <excludeSar>false</excludeSar>
    </configuration>
       <executions>
           <execution>
              <phase>package</phase>
              <goals>
                  <goal>repackage</goal>
              </goals>
           </execution>
    </executions>
</plugin>
```

这样打包后可以在不配置Pandora地址的情况下启动。

```
java -jar  -Dvipserver.server.port=8080 sc-vip-server-0.0.1-SNAPSHOT.jar
```

请在将应用部署到SAE前恢复到默认排除SAR包的配置。

3.5.4. 将Dubbo应用迁移到HSF（不推荐）

您可以通过添加Maven依赖、添加或修改Maven打包插件和修改配置，将使用Dubbo开发的应用迁移到HSF。不过，由于SAE已经支持原生Dubbo框架的应用，所以，新用户不建议使用此方式。

背景信息

原生Dubbo框架下的应用开发请参见使用Spring Boot开发Dubbo应用，您也可以直接下载Dubbo转换为HSF的示例代码。

添加Maven依赖

在应用工程的*pom.xml*中，增加 `spring-cloud-starter-pandora` 的依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-pandora</artifactId>
    <version>1.3</version>
</dependency>
```

添加或修改Maven打包插件

在应用工程的pom.xml中，添加或修改Maven的打包插件。

> **说明** 为避免与其他打包插件发生冲突，请勿在build的plugin中添加其他FatJar插件。

```
<build>
    <plugins>
        <plugin>
            <groupId>com.taobao.pandora</groupId>
            <artifactId>pandora-boot-maven-plugin</artifactId>
            <version>2.1.9.1</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

修改配置

在Spring Boot的启动类中，添加两行加载Pandora的代码：

```
import com.taobao.pandora.boot.PandoraBootstrap;
    import org.springframework.boot.SpringApplication;
    import org.springframework.boot.autoconfigure.SpringBootApplication;
    @SpringBootApplication
    public class ServerApplication {
        public static void main(String[] args) {
            PandoraBootstrap.run(args);
            SpringApplication.run(ServerApplication.class, args);
            PandoraBootstrap.markStartupAndWait();
        }
    }
```

3.6. 容器版本说明

本文列出了容器的版本发布记录，您可以视情况决定是否升级容器。

版本号	发布时间	构建包序号	Pandora版本	修改内容
3.6.3	2020-09-30	88、89	3.6.3	• 升级所有使用到logback、groovy-all这两个包的插件，将logback相关包的版本升级至1.2.3、groovy-all包的版本升级至2.4.20。 • 升级所有插件中commons-collections包的版本到3.2.2。 • 升级ons-client插件版本至1.8.7.1-EagleEye。

版本号	发布时间	构建包序号	Pandora版本	修改内容
3.6.2	2020-08-04	86、87	3.6.2	• 更新hsf插件，增加hsf-feature-warmup功能包，支持hsf服务上线预热功能。 • 升级ons-client插件，升级rocketmq到4.8.6.5。
3.6.1	2020-07-09	84、85	3.6.1	• 升级ons-client插件，升级rocketmq到4.8.6.5。 • 升级eagleeye插件，关闭场景链路模块导出类，防止与ARMS模块冲突。
3.6.0	2020-05-30	81、82、83	3.6.0	• 升级Pandora容器版本，修复JarURLConnection打开资源可能失败的问题。 • 升级所有使用到fastjson包的插件，将fastjson包升级到noneautotype的版本。 • 升级hsf插件， 使hsf服务支持同AZ优先路由。 • 移除unitrouter、monitor这两个插件。
3.5.9	2020-04-01	60 、80	3.5.9	• 升级edas.sar.V3.5.8里面所有用到fastjson的插件中的fastjson包的版本到安全版本。 • 升级ons-client插件版本到1.8.4.3-EagleEye。升级rocketmq。修复topAddr问题。
3.5.8	2020-01-10	59、79	3.5.8	升级config-client和hsf插件，修复HSF应用OOME时ConfigClientWorker线程退出以及spas鉴权失败导致注册服务被删除的问题。
3.5.7	2019-11-08	58、78	3.5.7	• 更新HSF插件，增加熔断功能，同时修复QoS Bug。 • 更新 `tddl-driver` 插件，移除GTS事务对PS Cache参数的检查。
3.5.6	2019-09-12	57、77	3.5.6	• 更新config-client插件，修复多租户场景未读缓存的问题。 • 更新HSF插件，修复pandora qos命令不能执行、HSF订阅服务数多的情况下可能遇到服务地址找不到的问题。 • 升级所有用到fastjson的插件到sec06安全版本。
3.5.5	2019-08-15	56、76	3.5.5	升级HSF插件中Dubbo依赖的hessian-lite。

版本号	发布时间	构建包序号	Pandora版本	修改内容
3.5.4	2019-07-18	55、75	3.5.4	• 更新HSF插件， 修复RPCContext不能清除的问题。 • 更新tddl-driver插件， 对于prepareCall操作遗漏了TXC_XID HINT信息的拼装处理。 • 更新metrics插件，修复BinAppender对象内存使用较多的问题。 • 升级所有使用到Fastjson的插件中Fastjson包版本至最新的1.2.58。
3.5.3	2019-03-13	54、74	3.5.3	• 升级了HSF与EagleEye插件版本，支持全链路灰度与HSF灰度流量控制。 • 升级了ONS-CLIENT插件版本到1.8.0-EagleEye。
3.5.2	2019-01-26	53、73	3.5.2	• 升级HSF插件到支持关闭服务发布或订阅到DEFAULT_TENANT租户的版本。 • 升级Ali-Tomcat版本到7.0.92。 • 重新添加ONS-CLIENT插件并升级版本至1.7.9-EagleEye。 • 更新其它Pandora插件版本。
3.5.1	2018-11-28	52、72	3.5.0	Docker镜像的JDK升级到JDK 1.8.0_191。
3.5.0	2018-09-10	51、71	3.5.0	• 升级eagleeye-core到1.7.4.8版本，修复Web应用URL请求中的中文参数值在应用中获取出现乱码的问题。 • 升级HSF到2.2.6.7-edas版本，修复了通过Pandora QoS命令无法看到HSF服务列表的问题。 • 去掉了ons-client插件（该插件中用到的JAR包跟应用的JAR包可能会引起冲突）。
3.4.7	2018-08-01	50、70	3.4.7	升级ONS到1.7.8-EagleEye版本，修复MQ Trace功能引起类冲突的问题。

版本号	发布时间	构建包序号	Pandora版本	修改内容

3.4.6	2018-07-05	49、69	3.4.6	• 升级HSF到2.2.6.1版本。
				  ◦ 支持CSB功能需求。
				  ◦ 修复某些场景下序列化出错的问题。
				  ◦ 修复强依赖VIP的问题。
				  ◦ 支持Spring Boot下Dubbo的健康检查。
				• 升级config-client到1.9.6版本，支持动态调整最大注册数。
				• 升级Sentinel到2.12.12-edas版本，支持Spring Boot 2。

3.4.5	2018-06-14	48、68	3.4.5	ACM升级到3.8.10版本，修复了在多租户下使用原生接口监听不生效的问题。
3.4.4	2018-05-18	47、67	3.4.4	• HSF服务端异步处理时+本地调用时，timeout值为0导致超时异常。 • SAE在使用Dubbo时，RpcContext缺少对端等IP属性 。 • 支持Dubbo的service标签在AOP场景使用。 • HSF泛化调用时，如果Bool值在Map里面，则不支持。
3.4.3	2018-04-24	46、66	3.4.3	• Diamond升级到3.8.8。 • 修复不断打印证书找不到问题，增加安全能力。 • EDAS-Assist升级到2.0。 • 优化端口可用性检测逻辑，Fastjson版本升级成1.2.48。
3.4.1	2018-03-15	44、64	3.4.1	• 升级hsf-plugin，支持dubboX。 • 升级diamond-client和configcenter-client。 • 升级edas-assit，取消在用户已经显示设定端口值时的端口检查。
3.4.0	2018-03-07	43、63	3.4.0	• 升级edas-assist，动态设置CSP端口及解决检查可用端口慢问题。 • 新增ConfigCenter租户版本。 • 升级configclient，实现内外客户端统一，支持CS2.0、CS3.0服务端。

版本号	发布时间	构建包序号	Pandora版本	修改内容
3.3.9	2018-01-17	42	3.3.9	• 升级HSF版本，解决了ZooKeeper阻塞的问题。 • 升级Sentinel新增系统保护（引入控制台推送规则才能生效）。 • 修改默认错误连接地址，适配国际化。
3.3.6	2017-12-20	41	3.3.6	• 重复抛出同一个异常时，会重复地增加头信息，导致异常提示信息特别长。 • EagleEye解析成UNKnown，影响了调用链解析，以及应用拓扑图的显示。
3.3.5	2017-12-20	40	3.3.4	HSF 2.2支持ACM。
3.3.4	2017-11-30	39	3.3.4	• 升级Diamond到最新版本，兼容ACM。 • 修复HSF泛化、Unit依赖、多个ZK地址解析异常、InetAddress序列化等问题，并支持白名单规则设置。 • 修复自定义限流降级页面需要等待30s左右才能生效的问题。 • EagleEye支持健康检查，附带alimetric、tomcat monitor。 • 升级ons-client，MQ新增了客户端设置消息缓存大小的配置。
3.3.3	2017-10-18	38	3.3.3	• 新增自动注册应用的功能，默认关闭。 • 解决HSF文件句柄占用问题。 • Sentinel支持HSF2.2，4，增强Pandora QoS命令。
3.3.2	2017-10-18	36	3.3.2	• 修复HSF持有hsf.lock句柄问题。 • 增加Redis埋点。升级tddl-driver，线上做全链路压测使用。 • 增强Pandora QoS命令。
3.3.1	2017-07-13	34	3.2.2	单独升级tddl-driver，线上做全链路压测使用。

> 说明
>
> - 版本号：为选择“EDAS Container”（Pandora容器）的应用容器版本。
> - 构建包序号：为“EDAS Container”（Pandora容器）的构建序号，与应用部署API中的“buildPackId”参数值对应。
> - Pandora版本：为“EDAS Container”（Pandora容器）真实的SAR包版本，与tabao-hsf.sar/version.properties文件中SAR属性值对应。单击该列对应的Pandora版本号数值可下载该版本的Pandora归档包。

SDK版本说明

SDK版本	EDAS Container版本要求	描述
1.8.2	3.5.0及以上版本	增强了自定义HSF Filter能力。您可以在filter自定义RPCResult的内容，并且通过HSFResponse返回给应用，例如filter内部特殊业务异常逻辑处理的场景。

3.7. 将应用从HSF架构迁移到Dubbo（Ali-Tomcat）

本文介绍如何使用Ali-Tomcat将使用HSF框架开发的应用迁移为Dubbo框架开发的应用。

迁移方案

迁移的最终目标是从HSF+SAE注册中心迁移到Dubbo+Nacos。目前有两种方案：

- 两步迁移
  i. 将HSF+SAE注册中心迁移到Dubbo+SAE注册中心。
  ii. 将SAE注册中心迁移到Nacos。

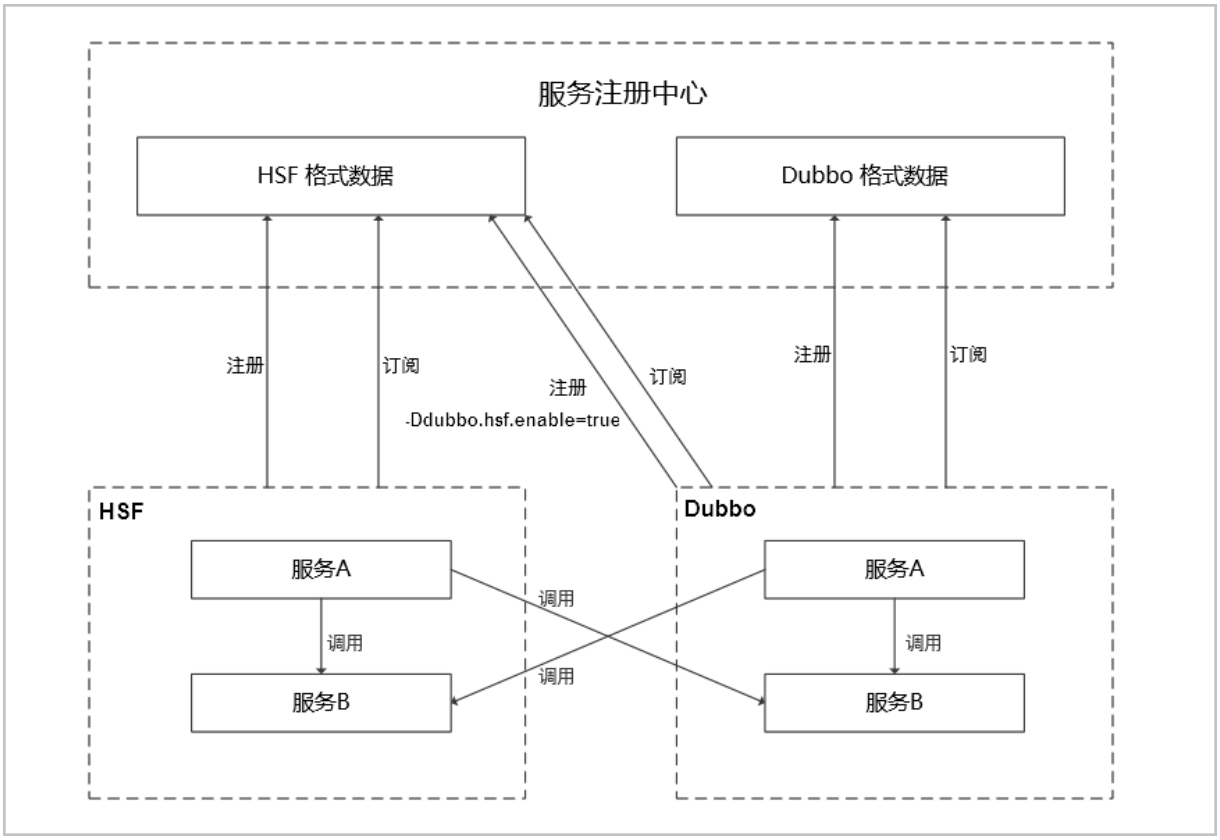
  优势是相对比较稳定，适合小步迭代。缺点是需要应用发布两次。
- 直接迁移

  将HSF+SAE注册中心直接迁移到Dubbo+Nacos。

  目前HSF尚不支持Nacos，需要额外开发。

如果应用想快速迁移到Dubbo并上线，建议采用第一种方案，从稳定性角度考虑也推荐第一种方案。下文将介绍如何进行两步迁移。

迁移架构图



Dubbo服务在服务注册的时候，同时注册成HSF和Dubbo的格式，保证HSF的服务消费者也发现Dubbo服务。Dubbo服务消费者在订阅的时候，同时订阅HSF和Dubbo格式的数据，保证Dubbo的消费者也能发现HSF的服务。

前提条件

迁移过程中需要依赖以下组件：

- 启动轻量级配置及注册中心
- Dubbo 2.7.3
- EDAS-container V3.5.5
- edas-dubbo-extension 2.0.6

假设HSF和Dubbo服务都继承自同一个接口，将这个接口单独剥离出来，命名为edas-demo-interface，只包含一个接口声明，目录如下：

```
├── pom.xml
├── src
│   └── main
│       └── java
│           └── com
│               └── alibaba
│                   └── edas
│                       └── DemoService.java
```

迁移服务提供者

假设待迁移的HSF应用为edas-hsf-demo-provider-war，主要包含以下文件：

- `pom.xml`：应用的模块之间依赖关系的配置文件。
- `DemoServiceImpl.java`：DemoService的实现。
- `hsf-provider-beans.xml`：HSF的Spring Bean声明文件。
- `web.xml`：用于WAR包部署的描述符。

`edas-hsf-demo-provider-war`的目录结构如下：

```
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── alibaba
│   │   │           └── edas
│   │   │               └── hsf
│   │   │                   └── provider
│   │   │                       └── DemoServiceImpl.java
│   │   ├── resources
│   │   │   └── hsf-provider-beans.xml
│   │   └── webapp
│   │       └── WEB-INF
│   │           └── web.xml
```

1. 在pom.xml中增加Dubbo相关依赖。

   HSF依赖的Spring版本建议使用4.x版本或其以上版本。

   i. 删除HSF的客户端依赖。

   ```
   <dependency>
               <groupId>com.alibaba.edas</groupId>
               <artifactId>edas-sdk</artifactId>
               <version>1.5.4</version>
   </dependency>
   ```

ii. 增加Dubbo相关依赖。

- `edas-dubbo-extension`：主要解决的是将Dubbo服务注册到SAE注册中心，以及以HSF格式方式对Dubbo服务进行注册和订阅。完整的*pom.xml*文件请参考示例代码。
- `dubbo`：标准的Dubbo依赖。

```
<dependency>
            <groupId>com.alibaba.edas</groupId>
            <artifactId>edas-dubbo-extension</artifactId>
            <version>2.0.5</version>
        </dependency>
        <dependency>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo</artifactId>
            <version>2.7.3</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
        </dependency>
```

2. 将hsf-provider-beans.xml修改为dubbo-provider-beans.xml。

   *hsf-provider-beans.xml*文件配置如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:hsf="http://www.taobao.com/hsf"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.taobao.com/hsf
       http://www.taobao.com/hsf/hsf.xsd" default-autowire="byName">
    <bean id="itemService" class="com.alibaba.edas.hsf.provider.DemoServiceImpl" />
    <!-- 提供一个服务示例 -->
    <hsf:provider id="demoService" interface="com.alibaba.edas.DemoService"
        ref="itemService" version="1.0.0">
    </hsf:provider>
</beans>
```

   需要修改为`dubbo-provider-beans.xml`：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin
gframework.org/schema/beans/spring-beans-4.3.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.
xsd">
    <dubbo:application name="edas-dubbo-demo-provider"/>
    <dubbo:registry id="edas" address="edas://127.0.0.1:8080">
        <!-- This means Dubbo services will be registered as HSF format, so that hsf co
nsumer can discover it. -->
        <dubbo:parameter key="hsf.enable" value="true"/>
    </dubbo:registry>
    <bean id="demoService" class="com.alibaba.edas.dubbo.provider.DemoServiceImpl"/>
    <dubbo:service interface="com.alibaba.edas.DemoService" ref="demoService" group="HS
F" version="1.0.0"/>
</beans>
```

说明

- Dubbo的注册中心需要配置为 `edas://127.0.0.1:8080` 。必须要以edas前缀开头，后续的IP和端口可以维持开发态，部署的时候SAE会自动替换成线上的地址。
- 需要增加 `<dubbo:parameter key="hsf.enable" value="true"/>` ，表示Dubbo服务在注册的时候会同时注册成HSF格式和Dubbo格式，确保HSF客户端可以发现该服务。
- 配置 `<dubbo:service>` 标签的时候，需要显示指定group和version，默认的group为 `HSF` ，版本号为 `1.0.0` ，否则HSF客户端无法正常调用。

3. 在web.xml文件中将hsf-provider-beans.xml替换为dubbo-provider-beans.xml。

   只需要将 `hsf-provider-beans.xml` 替换为 `dubbo-provider-beans.xml` 。

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
    <display-name>Archetype Created Web Application</display-name>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:dubbo-provider-beans.xml</param-value>
    </context-param>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
</web-app>
```

4. 本地验证。

   本地验证包括两部分：验证服务注册和验证服务消费者调用。

i. 验证服务注册。

a. 为了使HSF应用能够注册到本地的注册中心，需要修改host文件（例如*/etc/host*），添加如下条目到host文件中。

```
127.0.0.1     jmenv.tbsite.net
```

b. 下载轻量级配置及注册中心，并解压后进入bin目录，执行 `./startup.sh` 命令启动轻量级配置中心。

c. 执行 `mvn clean package` 命令将edas-hsf-demo-provider-war编译为WAR格式的程序包。

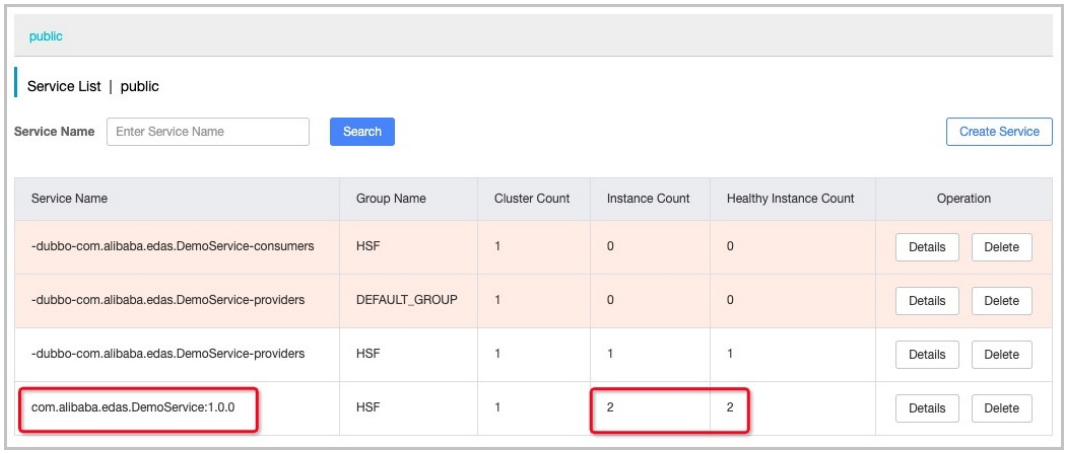
编译完成后文件存放在应用工程的*target*目录下。

d. 将*edas-hsf-demo-provider.war*部署到Ali-Tomcat下，将*edas-dubbo-demo-provider.war*部署到Apache Tomcat下。

> **说明** 启动两个Tomcat的端口可能会冲突，请在Tomcat的*conf/server.xml*文件中搜索8005和8080端口，将其修改为互不冲突的端口。

e. 访问轻量级配置及注册中心（http://127.0.0.1:8080/#/serviceManagement），查看com.alibaba.edas.DemoService:1.0.0服务。

如果服务已经注册，而且实例数为2，说明Dubbo和HSF的服务已经被注册成同一个HSF格式的服务了。



ii. 验证服务消费者调用。

请按照以下步骤验证迁移后的服务提供者是否能被消费者正常调用。

a. 准备测试的HSF服务消费者，如edas-hsf-demo-consumer-war，目录如下：

```
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   └── com
│   │   │       └── alibaba
│   │   │           └── edas
│   │   │               └── hsf
│   │   │                   └── consumer
│   │   │                       └── IndexServlet.java
│   │   ├── resources
│   │   │   └── hsf-consumer-beans.xml
│   │   └── webapp
│   │       └── WEB-INF
│   │           └── web.xml
```

它提供了一个Servlet，当接受到HTTP请求的时候，发起HSF调用。

```
public class IndexServlet extends HttpServlet {
    private DemoService demoService;
    @Override
    public void init() {
        WebApplicationContext wac = WebApplicationContextUtils.getRequiredWebAp
plicationContext(getServletContext());
        this.demoService = (DemoService) wac.getBean("demoService");
    }
    @Override
    public void doGet( HttpServletRequest req, HttpServletResponse resp ) {
        String result = demoService.sayHello("hsf");
        System.out.println("Received: " + result);
    }
}
```

b. 执行 `mvn clean package` 命令，将*edas-hsf-demo-consumer-war*编译为*edas-hsf-demo-consumer.war*，并将其部署到另外一个Ali-Tomcat中，注意避免端口冲突。

c. 登录轻量级配置及注册中心控制台。

如果发现HSF服务消费者没有数据在控制台里面展现，属于正常。

d. 启动Ali-Tomcat后，访问如下的URL：

```
curl http://localhost:8280/edas-hsf-demo-consumer/index.htm
```

e. 观察*edas-hsf-demo-consumer.war*所对应的Ali-Tomcat的标准输出。

如果输出类似如下内容，那么表示HSF客户端同时调用到了Dubbo和HSF提供的服务。

```
Received: Hello hsf, response from hsf provider: /192.168.XX.XX:62385
Received: Hello hsf, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello hsf, response from hsf provider: /192.168.XX.XX:62385
Received: Hello hsf, response from hsf provider: /192.168.XX.XX:62385
Received: Hello hsf, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello hsf, response from hsf provider: /192.168.XX.XX:62385
Received: Hello hsf, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello hsf, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello hsf, response from hsf provider: /192.168.XX.XX:62385
Received: Hello hsf, response from hsf provider: /192.168.XX.XX:62385
Received: Hello hsf, response from hsf provider: /192.168.XX.XX:62385
```

迁移服务消费者

基于*edas-hsf-demo-consumer-war*进行迁移，迁移为*edas-dubbo-demo-consumer-war*。

1. 在pom.xml中增加Dubbo相关依赖。

   迁移服务消费者操作和服务提供者迁移的相同，主要添加dubbo、dubbo-edas-extension依赖，删除edas-sdk依赖。具体操作请参见“迁移服务提供者”的在pom.xml中增加Dubbo相关依赖步骤。

2. 将*hsf-comsumer-beans.xml*修改为*dubbo-consumer-beans.xml*。

   hsf-consumer-beans.xml文件配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:hsf="http://www.taobao.com/hsf"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.taobao.com/hsf
       http://www.taobao.com/hsf/hsf.xsd" default-autowire="byName">
    <!-- 消费一个服务示例 -->
    <hsf:consumer id="demoService" interface="com.alibaba.edas.DemoService" version="1.
0.0">
    </hsf:consumer>
</beans>
```

   修改为*dubbo-consumer-beans.xml*。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin
gframework.org/schema/beans/spring-beans-4.3.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.
xsd">
    <dubbo:application name="edas-dubbo-demo-consumer"/>
    <dubbo:registry id="edas" address="edas://127.0.0.1:8080">
        <!-- This means Dubbo consumer will subscribe HSF services -->
        <dubbo:parameter key="hsf.enable" value="true"/>
    </dubbo:registry>
    <dubbo:reference id="demoService" interface="com.alibaba.edas.DemoService" group="H
SF" version="1.0.0" check="false"/>
</beans>
```

说明

- Dubbo的注册中心地址须以为 `edas://` 开头。
- 配置 `<dubbo:service>` 标签，需要指定 `group` 和 `version` ，且 `group` 和 `version` 须与服务提供者保持一致。默认 `group` 为HSF， `version` 为1.0.0。
- 添加 `check="false"` 配置，表示服务消费者应用在启动时如果没有服务端地址，那么该应用不会立刻失败。
- 增加 `<dubbo:parameter key="hsf.enable" value="true"/>` 配置，表示服务消费者订阅了服务提供者的数据。

3. 在web.xml文件中将hsf-consumer-beans.xml替换为dubbo-comsumer-beans.xml。
4. 本地验证。

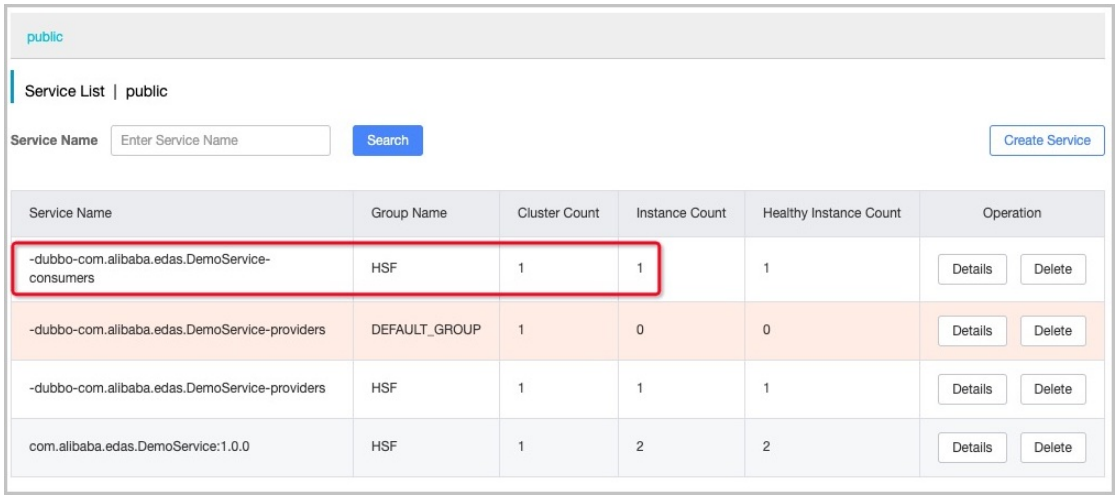
   本地验证包括两部分：验证服务是否注册到轻量级配置及注册中心和验证能否正常调用HSF和Dubbo服务。

   i. 将上述工程打包为*edas-dubbo-demo-consumer.war*，部署到Apache Tomcat下。

   注意 请避免端口冲突。

ii. 登录轻量级配置及注册中心控制台，检查Dubbo服务消费者注册是否注册成功。

在控制台上如果显示注册成功的消费者服务，那么表示服务注册成功。



iii. 访问*http://localhost:8280/edas-dubbo-demo-consumer/index.htm*。

```
curl http://localhost:8280/edas-dubbo-demo-consumer/index.htm
```

iv. 观察客户端Apache Tomcat的标准输出。

如果输出类似如下内容，那么表示Dubbo服务消费者消费了HSF和Dubbo的服务。

```
Received: Hello dubbo, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello dubbo, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello dubbo, response from hsf provider: /192.168.XX.XX:12202
Received: Hello dubbo, response from hsf provider: /192.168.XX.XX:12202
Received: Hello dubbo, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello dubbo, response from hsf provider: /192.168.XX.XX:12202
Received: Hello dubbo, response from hsf provider: /192.168.XX.XX:12202
Received: Hello dubbo, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello dubbo, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello dubbo, response from hsf provider: /192.168.XX.XX:12202
Received: Hello dubbo, response from hsf provider: /192.168.XX.XX:12202
Received: Hello dubbo, response from dubbo provider: 192.168.XX.XX:20880
Received: Hello dubbo, response from dubbo provider: 192.168.XX.XX:20880
```

将应用部署到SAE并验证

在SAE中创建4个应用：

- edas-dubbo-demo-consumer：迁移后的服务消费者应用，运行环境为Apache Tomcat 7.0.91。
- edas-dubbo-demo-provider：迁移后的服务提供者应用，运行环境为Apache Tomcat 7.0.91。
- edas-hsf-demo-consumer：迁移前的服务消费者应用，运行环境EDAS-Container v3.5.4。
- edas-hsf-demo-provider：迁移前的服务提供者应用，运行环境为EDAS-Container v3.5.4。

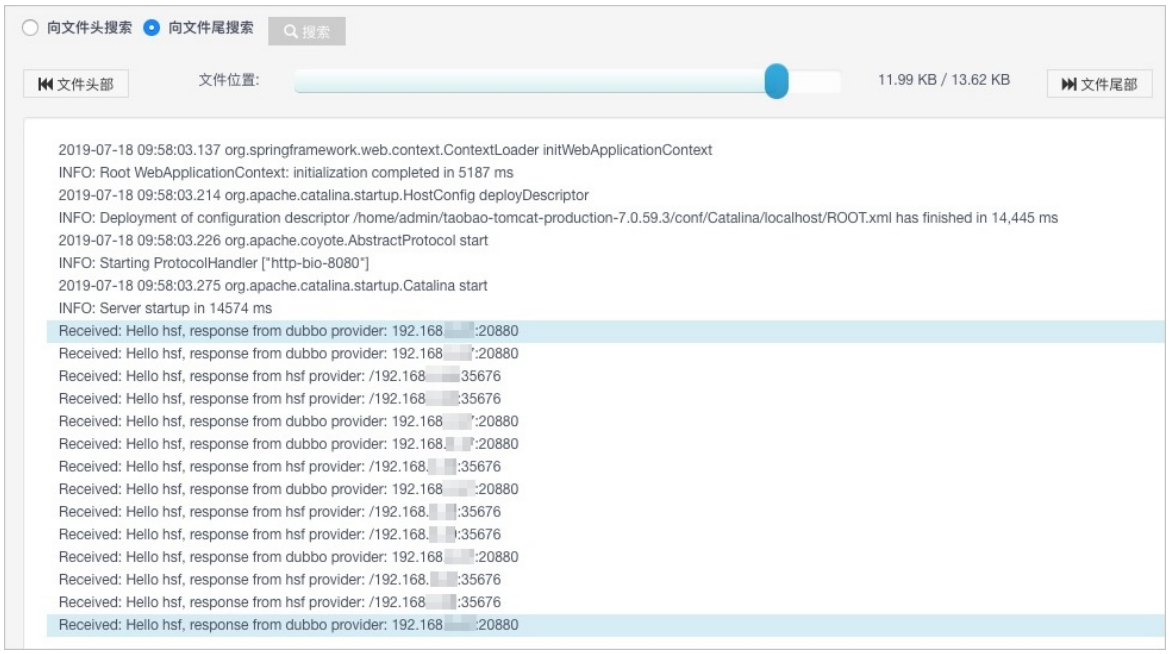
1. 分别将4个WAR包部署到4个应用中。详情请参见应用部署概述。
2. 执行以下命令，测试服务消费者edas-hsf-demo-consumer能够调用到HSF和Dubbo的服务提供者。

```
curl http://39.106.XX.XXX:8080/index.htm
```

3. 在edas-hsf-demo-consumer应用的Ali-Tomcat的标准输出中查看日志。如*/home/admin/taobao-to*

mcat-production-7.0.59.3/logs/catalina.out。

如果显示类似下图所示内容，那么表示HSF服务消费者消费到了HSF和Dubbo的服务。

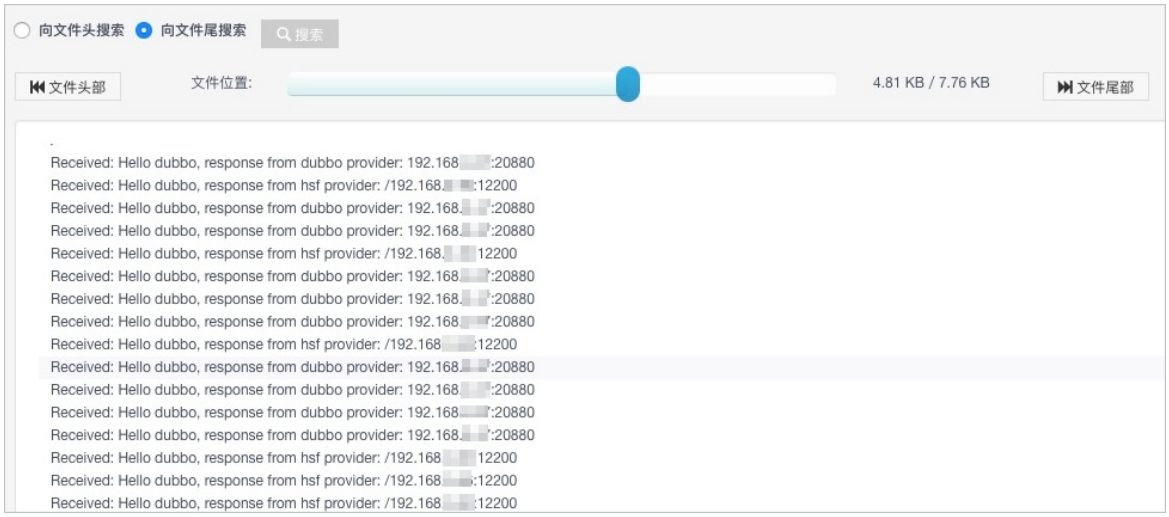


4. 执行以下命令，测试edas-dubbo-demo-consumer能否调用到HSF和Dubbo的Provider。

```
curl http://192.168.XX.XX:8080/index.htm
```

5. 在edas-dubbo-demo-consumer应用Apache Tomcat的标准输出中查看日志，例如*/home/admin/apache-tomcat-7.0.91/logs/catalina.out* 。

如果显示类似下图所示内容，那么表表示Dubbo服务消费者消费到了HSF和Dubbo的服务。



FAQ

- Dubbo服务消费者启动后，提示找不到服务提供者地址。

  问题现象

```
java.lang.IllegalStateException: Failed to check the status of the service com.xxxx.xxxxx
.service.xxxxxConfigService. No provider available for the service HSF/com.xxxxx.xxxxx.se
rvice.xxxxxxxxxxService:1.0.0 from the url edas://127.0.0.1:8080/org.apache.dubbo.registr
y.RegistryService?application=xxxx-flow-center-bj&dubbo=2.0.2&group=HSF&interface=com.xxx
x.xxxxxx.service.xxxxxxxxxxService&lazy=false&methods=queryConfigs,getConfig,saveConfig&p
id=11596&register.ip=xxx.xx.xx.xxx&release=2.7.3&revision=1.0.1-SNAPSHOT&side=consumer&st
icky=false&timeout=2000&timestamp=1564242421194&version=1.0.0 to the consumer xxx.xx.xx.x
xx use dubbo version 2.7.3
```

可能原因

注册中心的地址推送为异步推送，启动过程中Dubbo默认会检查服务提供者是否有可用地址。如果没有，则会抛出该异常。

解决方案

在Dubbo的 `<dubbo:reference>` 标签中增加 `check="false"` 配置：

```
<dubbo:reference id="demoService" interface="com.alibaba.edas.DemoService" group="HSF" ve
rsion="1.0.0" check="false"/>
```

该参数表示Dubbo启动过程中不会去检查提供者地址是否可用。但是，如果业务初始化逻辑里面有需要调用Dubbo服务的话，这种情况下业务可能会失败。

- HSF服务消费者调用Dubbo服务异常。

  问题现象

```
2019-07-28 23:07:38.005 [WARN ] [cf67433d1e7a44412a518bd190100d176-node401] [NettyServerW
orker-6-1] [o.a.d.r.exchange.codec.ExchangeCodec:91] |  [DUBBO] Fail to encode response:
Response [id=343493, version=HSF2.0, status=20, event=false, error=null, result=AppRespon
se [value=FlowControlDto(postWeightDtoHashMap={614215325=PostWeightDto(postId=614215325,
weight=1.0, postSourceType=null)}), exception=null]], send bad_response info instead, cau
se: For input string: "", dubbo version: 2.7.3, current host: xxx.xx.xx.xxx
java.lang.NumberFormatException: For input string: ""
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:592)
  at java.lang.Integer.parseInt(Integer.java:615)
  at org.apache.dubbo.common.Version.parseInt(Version.java:133)
  at org.apache.dubbo.common.Version.getIntVersion(Version.java:118)
  at org.apache.dubbo.common.Version.isSupportResponseAttachment(Version.java:102)
  at org.apache.dubbo.rpc.protocol.dubbo.DubboCodec.encodeResponseData(DubboCodec.java:19
5)
  at org.apache.dubbo.remoting.exchange.codec.ExchangeCodec.encodeResponse(ExchangeCodec.
java:283)
  at org.apache.dubbo.remoting.exchange.codec.ExchangeCodec.encode(ExchangeCodec.java:71)
  at org.apache.dubbo.rpc.protocol.dubbo.DubboCountCodec.encode(DubboCountCodec.java:40)
  at org.apache.dubbo.remoting.transport.netty4.NettyCodecAdapter$InternalEncoder.encode(
NettyCodecAdapter.java:70)
  ...
```

  可能原因

  Dubbo升级到2.7后，HSF对Dubbo的协议兼容性出现问题。

  解决方案

  升级EDAS-container到V3.5.5，该版本的HSF已经修复了该问题。

- Dubbo服务消费者调用HSF服务提供者失败。

  问题现象

```
java.lang.Exception: [HSF-Provider-192.168.0.46] Error log: [HSF-Provider] App [xxxxxxx-3b6f-42d3-xxxx-0ad2434xxxxx] failed to verify the caller signature [null] for [com.alibaba.edas.DemoService:1.0.0] [sayHello] from client [192.168.XX.XX]
        com.taobao.hsf.io.remoting.dubbo2.Dubbo2PacketFactory.serverCreate(Dubbo2PacketFactory.java:284)
        com.taobao.hsf.io.stream.AbstractServerStream.write(AbstractServerStream.java:25)
        com.taobao.hsf.io.RpcOutput.flush(RpcOutput.java:37)
        com.taobao.hsf.remoting.provider.ProviderProcessor$OutputCallback.operationComplete(ProviderProcessor.java:155)
        com.taobao.hsf.remoting.provider.ProviderProcessor$OutputCallback.operationComplete(ProviderProcessor.java:130)
        com.taobao.hsf.util.concurrent.AbstractListener.run(AbstractListener.java:18)
        com.taobao.hsf.invocation.AbstractContextAwareRPCCallback.access$001(AbstractContextAwareRPCCallback.java:12)
        com.taobao.hsf.invocation.AbstractContextAwareRPCCallback$1.run(AbstractContextAwareRPCCallback.java:27)
        com.taobao.hsf.util.concurrent.WrappedListener.run(WrappedListener.java:34)
        com.taobao.hsf.invocation.AbstractContextAwareRPCCallback.run(AbstractContextAwareRPCCallback.java:36)
        com.google.common.util.concurrent.MoreExecutors$DirectExecutor.execute(MoreExecutors.java:456)
        com.google.common.util.concurrent.AbstractFuture.executeListener(AbstractFuture.java:817)
        com.google.common.util.concurrent.AbstractFuture.addListener(AbstractFuture.java:595)
        com.taobao.hsf.util.concurrent.DefaultListenableFuture.addListener(DefaultListenableFuture.java:32)
        com.taobao.hsf.remoting.provider.ProviderProcessor.handleRequest(ProviderProcessor.java:55)
        com.taobao.hsf.io.remoting.dubbo2.message.Dubbo2ServerHandler$1.run(Dubbo2ServerHandler.java:65)
        java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        java.lang.Thread.run(Thread.java:748)
```

  可能原因

  HSF开启了调用鉴权，而Dubbo暂时不支持鉴权。

  解决方案

  在HSF服务端增加参数 `-DneedAuth=false` ，关闭调用鉴权。

- Dubbo服务消费者调用HSF服务提供者失败。

  问题现象

```
2019-08-02 17:17:15.187 [WARN ] [cf67433d1e7a44412a518bd190100d176-node401] [NettyClientW
orker-4-1] [o.a.d.r.p.dubbo.DecodeableRpcResult:91] |  [DUBBO] Decode rpc result failed:
null, dubbo version: 2.7.3, current host: xxx.xx.xx.xxx
java.lang.StackOverflowError: null
    at sun.reflect.UnsafeFieldAccessorImpl.ensureObj(UnsafeFieldAccessorImpl.java:57)
    at sun.reflect.UnsafeByteFieldAccessorImpl.setByte(UnsafeByteFieldAccessorImpl.java:9
8)
    at java.lang.reflect.Field.setByte(Field.java:838)
    at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer$ByteFieldDeserializer.deseriali
ze(JavaDeserializer.java:452)
    at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.jav
a:276)
    at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.jav
a:203)
    at com.alibaba.com.xxxxxx.hessian.io.SerializerFactory.readObject(SerializerFactory.j
ava:532)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObjectInstance(Hessian2Input.j
ava:2820)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2743
)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2278
)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2080
)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2074
)
    at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer$ObjectFieldDeserializer.deseria
lize(JavaDeserializer.java:406)
    at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.jav
a:276)
    at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.jav
a:203)
    at com.alibaba.com.xxxxxx.hessian.io.SerializerFactory.readObject(SerializerFactory.j
ava:532)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObjectInstance(Hessian2Input.j
ava:2820)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2743
)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2278
)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2080
)
    at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2074
)
    at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer$ObjectFieldDeserializer.deseria
lize(JavaDeserializer.java:406)
    ...
```

可能原因

HSF服务提供和依赖的hessian-lite版本较低，不支持JDK 8的LocalDateTime的序列化。

解决方案

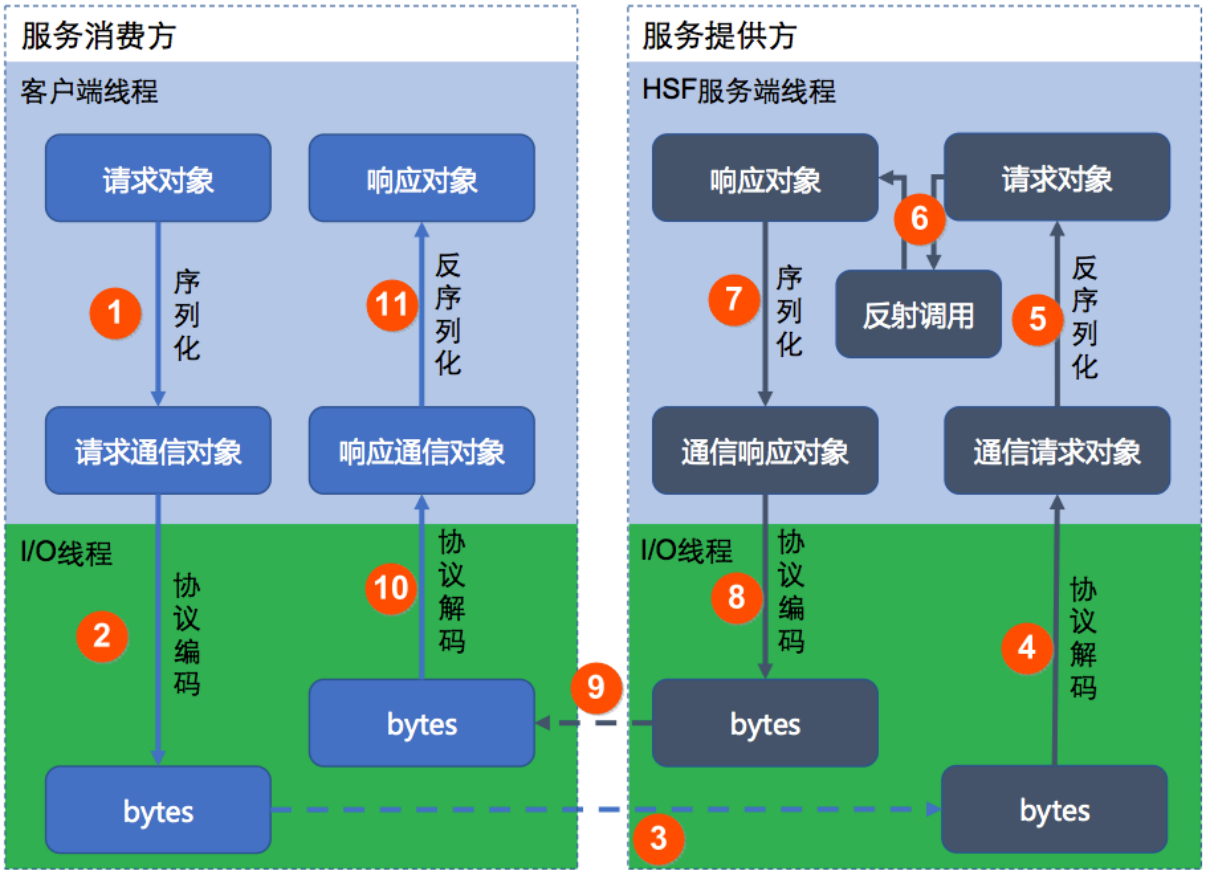
升级HSF服务端的EDAS-Container的版本到v3.5.5。

3.8. 一次调用过程

本文介绍HSF的一次调用过程。

HSF的一次调用过程是从服务消费方发起，经过网络抵达服务提供方，再将服务提供方的结果通过网络携带返回，最终返回给用户。该过程涉及多线程交互，同时也涉及HSF中的不同领域对象。

HSF的一次调用过程如下图所示：



过程	说明
1	在客户端线程中将用户的请求参数即请求对象进行序列化，并将序列后的内容存放在请求通信的对象中。 **说明** 请求通信对象对应的是HSF协议，包括了请求ID等多个与请求对象无关的内容。
2	系统将请求通信对象递交给I/O线程，并在I/O线程中完成编码。
3	编码完成后，将内容传递到服务提供方的I/O线程。客户端线程会等待结果返回。
4	服务提供方的I/O线程接收到二进制内容，解码后生成通信请求对象，并将其递交给HSF服务端线程。

过程	说明
5	在HSF服务端线程完成反序列化还原成请求对象。
6	发起反射调用，并得到结果，即响应对象。
7	响应对象会在HSF服务端线程中完成序列化，并存储在通信响应对象中。
8	HSF 服务端线程将通信响应对象递交给I/O线程，在I/O线程中完成编码。
9	服务提供方将I/O线程中完成编码，发送回服务消费方。
10	服务消费方收到二进制内容，在I/O线程中完成解码，生成响应通信对象，并唤醒客户端线程。
11	客户端线程根据响应通信对象中的内容进行反序列化，用户收到响应对象，一次远程调用结束。

3.9. 异步调用

本文介绍HSF如何进行异步调用。

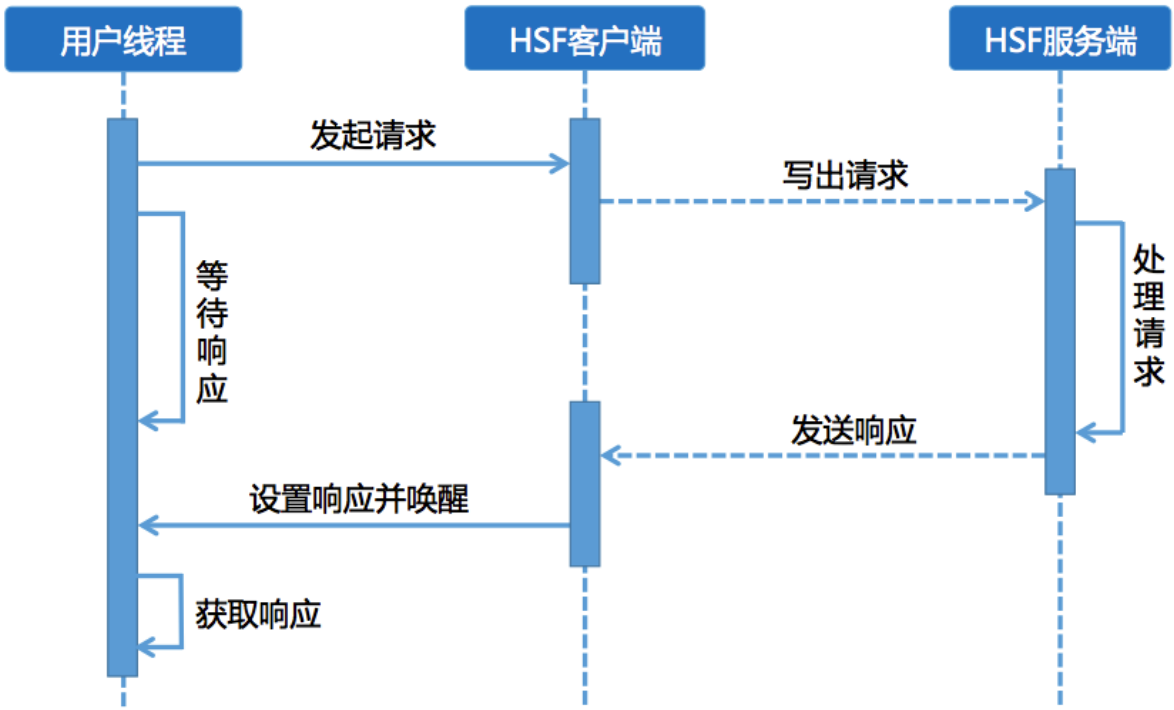
前提条件

在开发应用前，确保您已完成以下操作：

- 配置SAE的私服地址和轻量级配置及注册中心
- 启动轻量级配置及注册中心

同步调用

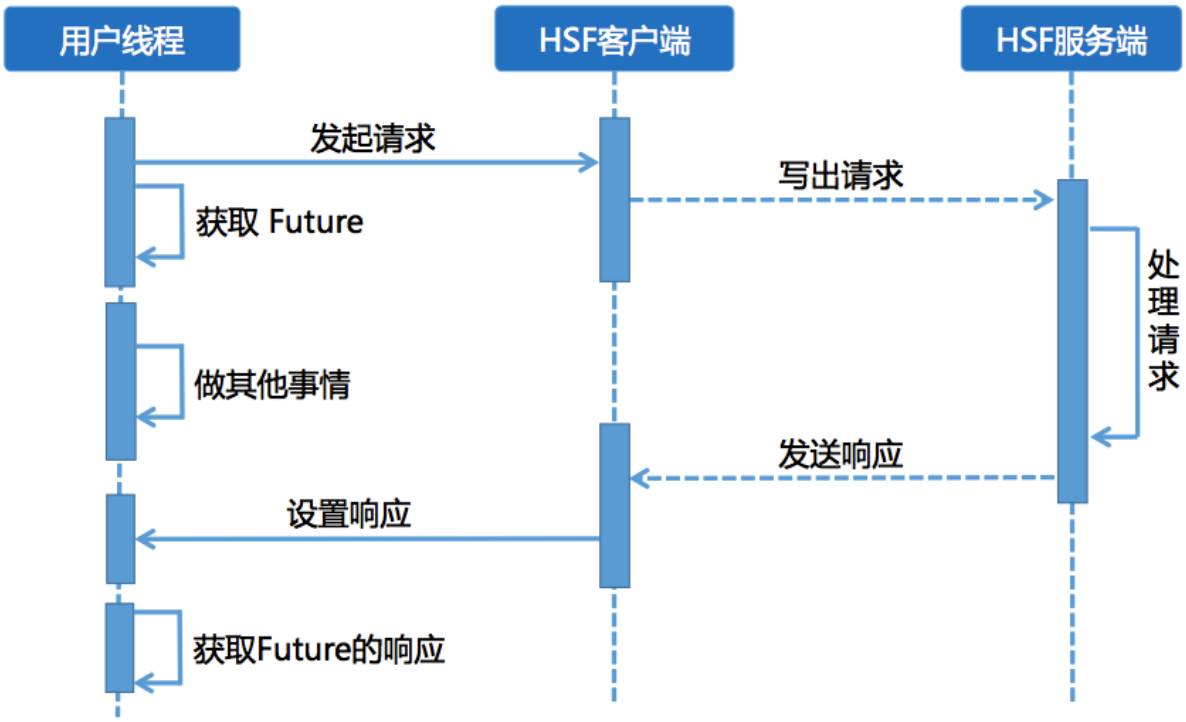
HSF的IO操作是异步操作，客户端同步调用的本质是执行 `future.get（timeout）` 操作，等待服务端的结果返回，这里的 `timeout` 就是客户端生效的超时时间 （默认3000ms）。同步调用时序图如下所示：



对于客户端来说，并不是所有的HSF服务都是需要同步等待服务端返回结果的，对于这些服务，HSF提供异步调用的形式，让客户端不必同步阻塞在HSF操作上。 异步调用在发起调用时，HSF服务的调用结果都是返回值的默认值，例如返回类型是int，则会返回0，返回类型是Object，则会返回null。而真正的结果，是在HSFResponseFuture或者回调函数（Callback）中获得的。

Future异步调用

HSF发起调用后，用户可以在上下文中获取跟返回结果关联的 `HSFFuture` 对象，获取对象后调用 `HSFFuture.getResponse（timeout）` 获取服务端的返回结果。Future异步调用的时序图如下所示：



- API形式配置HSF服务

  HSF提供了方法级别的异步调用配置，格式为 `name:${methodName};type:future` ，由于只用方法名字来标识方法，所以并不区分重载的方法。同名的方法都会被设置为同样的调用方式。

```
HSFApiConsumerBean hsfApiConsumerBean = new HSFApiConsumerBean();
hsfApiConsumerBean.setInterfaceName("com.alibaba.middleware.hsf.guide.api.service.OrderSe
rvice");
hsfApiConsumerBean.setVersion("1.0.0");
hsfApiConsumerBean.setGroup("HSF");
// [设置] 异步future调用List<String> asyncallMethods = new ArrayList<String>();
// 格式: name:{methodName};type:future
asyncallMethods.add("name:queryOrder;type:future");
hsfApiConsumerBean.setAsyncallMethods(asyncallMethods);
hsfApiConsumerBean.init(true);
// [代理] 获取HSF代理OrderService orderService = (OrderService) hsfApiConsumerBean.getObjec
t();
// ---------------------- 调用 ----------------------//
// [调用] 发起HSF异步调用，返回null
OrderModel orderModel = orderService.queryOrder(1L);
// 及时在当前调用上下文中，获取future对象；因为该对象是放在ThreadLocal中，同一线程中后续调用会覆盖f
uture对象，所以要及时取出HSFFuture hsfFuture = HSFResponseFuture.getFuture();
// do something else
// 这里才真正地获取结果，如果调用还未完成，将阻塞等待结果，5000ms是等待结果的最大时间try {
    System.out.println(hsfFuture.getResponse(5000));
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

  HSF默认的超时配置是3000ms，如果过了超时时间，业务对象未返回，这时调用HSFFuture.getResponse会抛出超时异常；HSFFuture.getResponse(timeout)，如果这里的timeout时间内，业务结果没有返回，也没有超时，可以调用多次执行getResponse去获取结果。

- Spring配置HSF服务

  Spring框架在应用中广泛使用，如果不想以API的形式配置HSF服务，您可以使用Spring XML的形式进行配置，上述例子中API配置等同于如下XML配置。

```
<bean id="orderService" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean">
    <property name="interfaceName" value="com.alibaba.middleware.hsf.guide.api.service.Or
derService"/>
    <property name="version" value="1.0.0"/>
    <property name="group" value="HSF"/>
    <!--[设置] 订阅服务的接口 -->
    <property name="asyncallMethods">
        <list>
            <value>name:queryOrder;type:future</value>
        </list>
     </property>
</bean>
```

- 注解配置HSF服务

  SpringBoot广泛使用的今天，使用注解装配SpringBean也成为一种选择，HSF也支持使用注解进行配置，用来订阅服务。

  在项目中增加依赖starter。

```
<dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>pandora-hsf-spring-boot-starter</artifactId>
</dependency>
```

  通常一个HSF Consumer会在多个地方使用，但并不需要在每次使用的地方都用 @HSFConsumer来标记。只需要写一个统一个Config类，然后在其它需要使用的地方，直接 @Autowired注入即可。上述例子中的API配置等同于如下注解配置。

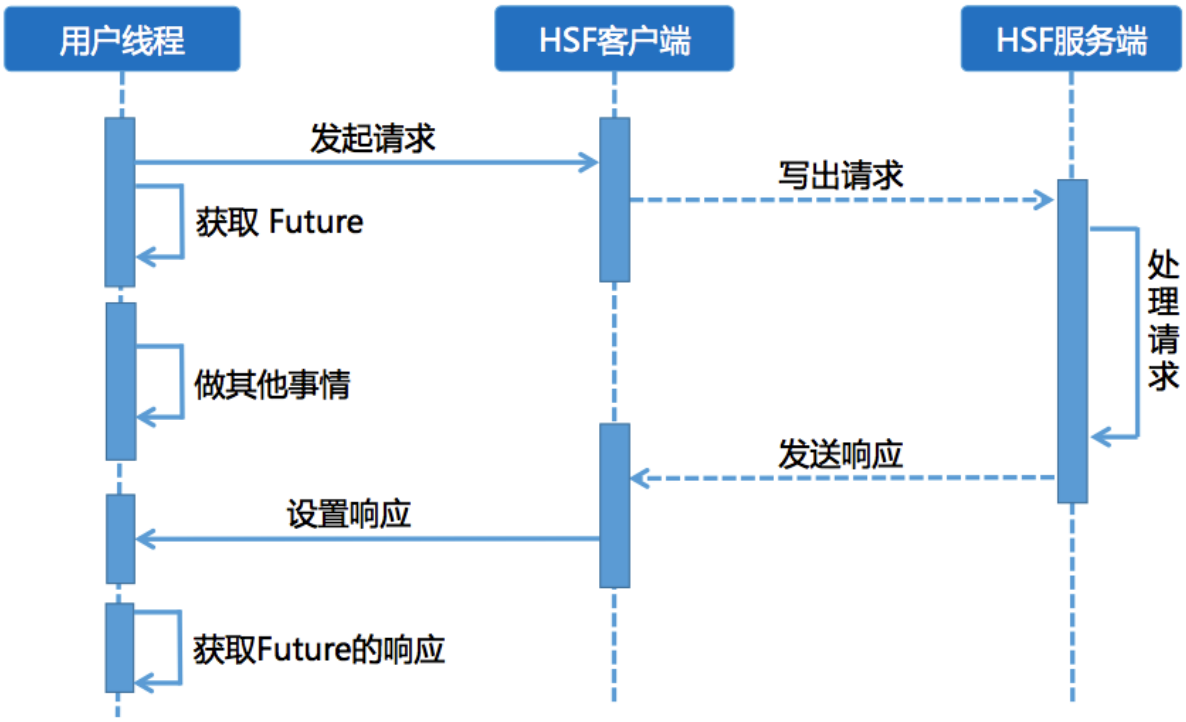
```
@Configuration
public class HsfConfig {
    @HSFConsumer(serviceVersion = "1.0.0", serviceGroup = "HSF", futureMethods = "sayHell
oInFuture")
    OrderService orderService;
}
```

  在使用时直接注入即可。

```
@Autowired
OrderService orderService;
```

Future异步调用

HSF发起调用后，用户可以在上下文中获取跟返回结果关联的 `HSFFuture` 对象，获取对象后调用 `HSFFuture.getResponse (timeout)` 获取服务端的返回结果。Future异步调用的时序图如下所示：



- API形式配置HSF服务

  HSF提供了方法级别的异步调用配置，格式为 `name:${methodName};type:future` ，由于只用方法名字来标识方法，所以并不区分重载的方法。同名的方法都会被设置为同样的调用方式。

```
HSFApiConsumerBean hsfApiConsumerBean = new HSFApiConsumerBean();
hsfApiConsumerBean.setInterfaceName("com.alibaba.middleware.hsf.guide.api.service.OrderSe
rvice");
hsfApiConsumerBean.setVersion("1.0.0");
hsfApiConsumerBean.setGroup("HSF");
// [设置] 异步future调用List<String> asyncallMethods = new ArrayList<String>();
// 格式: name:{methodName};type:future
asyncallMethods.add("name:queryOrder;type:future");
hsfApiConsumerBean.setAsyncallMethods(asyncallMethods);
hsfApiConsumerBean.init(true);
// [代理] 获取HSF代理OrderService orderService = (OrderService) hsfApiConsumerBean.getObjec
t();
// ---------------------- 调用 ----------------------//
// [调用] 发起HSF异步调用，返回null
OrderModel orderModel = orderService.queryOrder(1L);
// 及时在当前调用上下文中，获取future对象；因为该对象是放在ThreadLocal中，同一线程中后续调用会覆盖f
uture对象，所以要及时取出HSFFuture hsfFuture = HSFResponseFuture.getFuture();
// do something else
// 这里才真正地获取结果，如果调用还未完成，将阻塞等待结果，5000ms是等待结果的最大时间try {
    System.out.println(hsfFuture.getResponse(5000));
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

  HSF默认的超时配置是3000ms，如果过了超时时间，业务对象未返回，这时调用HSFFuture.getResponse会抛出超时异常；HSFFuture.getResponse(timeout)，如果这里的timeout时间内，业务结果没有返回，也没有超时，可以调用多次执行getResponse去获取结果。

- Spring配置HSF服务

  Spring框架在应用中广泛使用，如果不想以API的形式配置HSF服务，您可以使用Spring XML的形式进行配置，上述例子中API配置等同于如下XML配置。

```
<bean id="orderService" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean">
    <property name="interfaceName" value="com.alibaba.middleware.hsf.guide.api.service.OrderService"/>
    <property name="version" value="1.0.0"/>
    <property name="group" value="HSF"/>
    <!--[设置] 订阅服务的接口 -->
    <property name="asyncallMethods">
        <list>
            <value>name:queryOrder;type:future</value>
        </list>
     </property>
</bean>
```

- 注解配置HSF服务

  SpringBoot广泛使用的今天，使用注解装配SpringBean也成为一种选择，HSF也支持使用注解进行配置，用来订阅服务。

  在项目中增加依赖starter。

```
<dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>pandora-hsf-spring-boot-starter</artifactId>
</dependency>
```

  通常一个HSF Consumer会在多个地方使用，但并不需要在每次使用的地方都用 @HSFConsumer来标记。只需要写一个统一个Config类，然后在其它需要使用的地方，直接 @Autowired注入即可。上述例子中的API配置等同于如下注解配置。

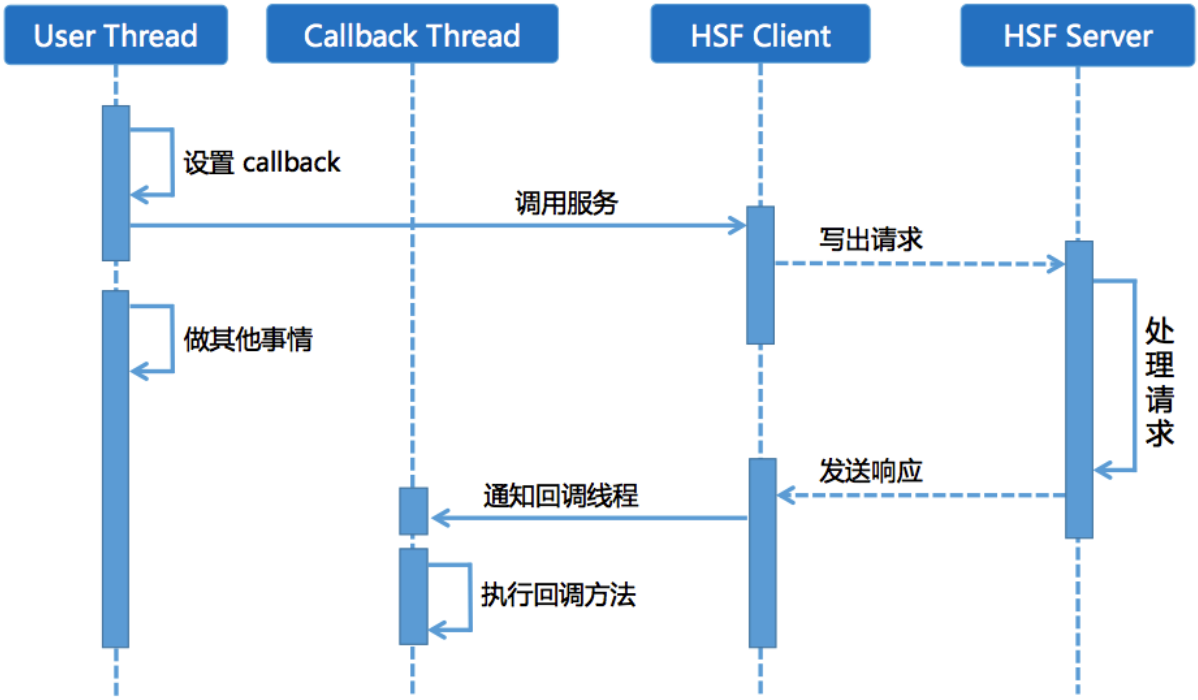
```
@Configuration
public class HsfConfig {
    @HSFConsumer(serviceVersion = "1.0.0", serviceGroup = "HSF", futureMethods = "sayHelloInFuture")
    OrderService orderService;
}
```

  在使用时直接注入即可。

```
@Autowired
OrderService orderService;
```

Callback异步调用

客户端配置为Callback方式调用时，需要配置一个实现了HSFResponseCallback接口的listener，结果返回之后，HSF会调用HSFResponseCallback中的方法。时序图如下所示：



- API形式配置HSF服务

  Callback的调用上下文

  用户在调用前还可以通过CallbackInvocationContext.setContext(Object obj)，来设置一个关于本次调用的上下文信息，该信息存放在threadlocal中。在listener的回调函数中，可以通过CallbackInvocationContext.getContext() 来获取该对象。

  回调函数示例

```
public class CallbackHandler implements HSFResponseCallback {
    // 业务异常时会触发
    @Override
    public void onAppException(Throwable t) {
        t.printStackTrace();
    }
    // 业务返回结果
    @Override
    public void onAppResponse(Object result) {
        // 取callback调用时设置的上下文Object context = CallbackInvocationContext.getContext
();
        System.out.println(result.toString() + context);
    }
    //HSF异常
    @Override
    public void onHSFException(HSFException e) {
        e.printStackTrace();
    }
}
```

  接口Callback方法配置

```
HSFApiConsumerBean hsfApiConsumerBean = new HSFApiConsumerBean();
hsfApiConsumerBean.setInterfaceName("com.alibaba.middleware.hsf.guide.api.service.OrderSe
rvice");
hsfApiConsumerBean.setVersion("1.0.0");
hsfApiConsumerBean.setGroup("HSF");
// [设置] 异步callback调用List<String> asyncallMethods = new ArrayList<String>();
asyncallMethods.add("name:queryOrder;type:callback;listener:com.alibaba.middleware.hsf.Ca
llbackHandler");
hsfApiConsumerBean.setAsyncallMethods(asyncallMethods);
hsfApiConsumerBean.init(true);
// [代理] 获取HSF代理OrderService orderService = (OrderService) hsfApiConsumerBean.getObjec
t();
 // 可选步骤，设置上下文。CallbackHandler中通过api可以获取到CallbackInvocationContext.setConte
xt("in callback");
// 发起调用orderService.queryOrder(1L); // 这里返回的其实是null
// 清理上下文CallbackInvocationContext.setContext(null);
// do something else
```

在调用线程中可以设置上下文，然后在listener中获取使用。相对于Future异步调用，callback会立即知晓结果的返回。

- Spring配置HSF服务

```
<bean id="CallHelloWorld" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean">
    <!--[设置] 订阅服务的接口 -->
    <property name="interfaceName" value="com.alibaba.middleware.hsf.guide.api.service.Or
derService"/>
    <!--[设置] 服务的版本 -->
    <property name="version" value="1.0.0"/>
    <!--[设置] 服务的归组 -->
    <property name="group" value="HSF"/>
    <property name="asyncallMethods">
        <list>
            <!--future的含义为通过Future的方式去获取请求执行的结果，例如先调用下远程的接口，接着在
同一线程继续做别的事情，然后再在同一线程中通过Future来获取结果 -->
            <!--name:methodName;type:future|callback-->
            <value>name:queryOrder;type:callback;listener:com.alibaba.middleware.hsf.Call
backHandler</value>
        </list>
     </property>
</bean>
```

- 注解配置HSF接口方法为callback调用

```
@AsyncOn(interfaceName = OrderService.class, methodName = "queryOrder")
public class CallbackHandler implements HSFResponseCallback {
    @Override
    public void onAppException(Throwable t) {
        t.printStackTrace();
    }
    @Override
    public void onAppResponse(Object result) {
        // 取callback调用时设置的上下文Object context = CallbackInvocationContext.getContext
();
        System.out.println(result.toString() + context);
    }
    @Override
    public void onHSFException(HSFException e) {
        e.printStackTrace();
    }
}
```

注意 回调函数是由单独的线程池（LinkedBlockingQueue无限队列）来调用的，不要做太费时间的操作，避免影响其他请求的onAppResponse回调。callback线程默认的corePoolSize、maxPoolSize是实例CPU数目。 下面的*-D*参数可以去自定义配置。

- CALLBACK线程池最小配置： `-Dhsf.callback.min.poolsize`
- CALLBACK线程池最大配置： `-Dhsf.callback.max.poolsize`

3.10. 泛化调用

相对于需要依赖业务客户端JAR包的正常调用，泛化调用不需要依赖二方包，使用其特定的GenericService接口，传入需要调用的方法名、方法签名和参数值进行调用服务。 泛化调用适用于一些网关应用（没办法依赖所有服务的二方包），其中hsfops服务测试也是依赖泛化调用功能。

前提条件

在开发应用前，确保您已完成以下操作：

- 配置SAE的私服地址和轻量级配置及注册中心
- 启动轻量级配置及注册中心

API形式配置HSF服务

将HSFConsumerBean配置 `generic` 为 `true` ，HSF客户端将忽略加载不到接口的异常。

```
HSFApiConsumerBean hsfApiConsumerBean = new HSFApiConsumerBean();
hsfApiConsumerBean.setInterfaceName("com.alibaba.middleware.hsf.guide.api.service.OrderServ
ice");
hsfApiConsumerBean.setVersion("1.0.0");
hsfApiConsumerBean.setGroup("HSF");
// [设置] 泛化配置hsfApiConsumerBean.setGeneric("true");
hsfApiConsumerBean.init(true);
// 使用泛化接口获取代理GenericService genericOrderService = (GenericService) hsfApiConsumerBea
n.getObject();
// ---------------------- 调用 ----------------------//
// [调用] 发起HSF泛化调用，返回map类型的result
Map orderModelMap = (Map) genericOrderService.$invoke("queryOrder",
                            // 方法入参类型数组（xxx.getClass().getName())
                            new String[] { Long.class.getName() },
                            // 参数，如果是pojo，则需要转成Map
                            new Object[] { 1L});
```

GenericService提供的$invoke方法包含了真实调用的方法名、入参类型和参数，以便服务端找到该方法。由于没有依赖服务端的API JAR包，传入的参数如果是自定义的DTO，需要转成客户端可以序列化的 `Map` 类型。

调用方法和参数说明

- 方法没有入参，可以只传： `methodName:service.$invoke("sayHello", null, null)` 。
- 方法类型有泛型的，列如 `List<String>` ，只需要传 `java.util.List` ，即List.class.getName()的值，不要传成 `java.util.List<String>` ，否则会出现方法找不到的错误。
- 调用方法在不确定格式的情况下，可以写个单元测试，测试时依赖需要泛化调用的二方包，使用HSF提供的工具类com.taobao.hsf.util.PojoUtils的generalize()方法来生成一个pojo Bean的Map描述格式。

```
Map pojoMap = (Map) PojoUtils.generalize(new OrderModel());
```

- 传递参数为pojo的demo。

```
 class User {
    private String name;
    private int age;
    // 需要是标准的pojo格式，这里省略getter setter
   }
   // 直接使用map去构造pojo对应的泛化参数Map param = new HashMap<String, Object>();
   param.put("age", 11);
   param.put("name","Miles");
   // 当传递的参数是声明参数类型的子类时，需要传入class字段，标明该pojo的真实类型（服务端需要有该类
型）
   param.put("class", "com.taobao.User");
```

Spring配置HSF服务

Spring框架是在应用中广泛使用的组件，如果不想通过API的形式配置HSF服务，可以使用Spring XML的形式进行配置，上述例子中的API配置等同于如下XML配置。

```
<bean id="CallHelloWorld" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean">
    <!--[设置] 订阅服务的接口 -->
    <property name="interfaceName" value="com.alibaba.middleware.hsf.guide.api.service.OrderService"/>
    <!--[设置] 服务的版本 -->
    <property name="version" value="1.0.0"/>
    <!--[设置] 服务的归组 -->
    <property name="group" value="HSF"/>
    <property name="generic" value="true"/>
</bean>
```

注意事项

- 泛化调用，如果客户端没有接口类，路由规则默认不生效。
- 泛化调用性能会比正常调用差。
- 配置 `-Dhsf.generic.throw.exception=true` （默认是false，把异常泛化成map返回） 抛出业务异常。

  本地存在异常类，如果不是RuntimeException类型或其子类，则会抛出UndeclaredThrowableException，这是由于com.taobao.hsf.remoting.service.GenericService上没有声明该异常，可以通过getCause获取真实异常。

  本地没有该异常类，则抛出com.taobao.hsf.util.GenericInvocationException。

3.11. 调用上下文

请求上下文包括一次调用相关的属性，例如调用的地址、调用方的应用名、超时时间等属性和用户在接口定义的参数之外传递自定义的数据。

设置和获取本次调用上下文

com.taobao.hsf.util.RequestCtxUtil提供设置和获取调用上下文的静态方法，基于ThreadLocal工作，getXXX操作会将XXX属性从当前ThreadLocal变量中remove，仅作用于当前线程的单次调用。具体属性的设置和获取如下：

- 客户端

方法	说明
setRequestTimeout()	设置单次调用的超时时间。
setUserId()	设置本次调用的单元化服务的User ID（泛化调用中需要通过此方法配置）。
getProviderIp()	获取【最近一次】调用的服务端的IP地址。
setTargetServerIp(String ip)	设置当前线程下一次调用的目标服务器IP地址（此IP地址必须包含在内存已提供服务的地址列表里）。
setDirectTargetServerIp(String targetIp)	设置当前线程下一次调用的目标服务器IP地址（绕过注册中心，忽略内存里的地址列表）。

- 服务端

方法	说明
getClientIp()	服务端获取调用方IP地址。
getAppNameOfClient()	服务端获取调用方的应用名。
isHttpRequest()	是否是HTTP调用。
getHttpHeader(String key)	获取HTTP请求的Header属性。

传递自定义请求上下文

RpcContext 提供一种不修改接口、向服务端额外传递数据的方式。参数可以是自定义的DO或者基本类型。要保证对端也有该对应的类型，并且可以能够被序列化。

客户端发起调用前，设置上下文。

```
//setup context before rpc call
RPCContext rpcContext = RPCContext.getClientContext();
rpcContext.putAttachment("tetantId", "123");
//rpc call, context也会传到远端orderService.queryOrder(1L);
```

服务端业务方法内，获取上下文。

```
//get context data
RPCContext rpcContext = RPCContext.getServerContext();
String myContext = (String)rpcContext.getAttachment("tetantId");
```

3.12. 序列化方式选择

序列化的过程是将Java对象转成byte数组在网络中传输，反序列化会将byte数组转成Java对象。

简介

序列化的选择需要考虑兼容性，性能等因素，HSF的序列化方式支持java、hessian2，默认是hessian2。序列化方式的对比和配置（只在服务端配置HSFApiProviderBean）如下表所示。

序列化方式	Maven依赖	配置	兼容性	性能
hessian2	<artifactId>hsf-io-serialize-hessian2</artifactId>	`setPreferSerializeType("hessian2")`	好	好
java	<artifactId>hsf-io-serialize-java</artifactId>	`setPreferSerializeType("java")`	佳	一般

前提条件

在开发应用前，您已经完成以下工作：

-

-

API形式配置HSF服务

```
HSFApiProviderBean hsfApiProviderBean = new HSFApiProviderBean();
hsfApiProviderBean.setPreferSerializeType("hessian2");
```

Spring配置HSF服务

Spring框架是在应用中广泛使用的组件，如果不想通过API的形式配置HSF服务，可以使用Spring XML的形式进行配置，上述例子中的API配置等同于如下XML配置。

```
<bean class="com.taobao.hsf.app.spring.util.HSFSpringProviderBean" init-method="init">
    <!--[设置] 发布服务的接口 -->
    <property name="serviceInterface" value="com.alibaba.middleware.hsf.guide.api.service.O
rderService"/>
    <!--[设置] 服务的实现对象target必须配置 [ref]，为需要发布为HSF服务的spring bean id-->
    <property name="target" ref="引用的BeanId"/>
    <!--[设置] 服务的版本 -->
    <property name="serviceVersion" value="1.0.0"/>
    <!--[设置] 服务的归组 -->
    <property name="serviceGroup" value="HSF"/>
    <!--[设置] 服务的响应时间 -->
    <property name="clientTimeout" value="3000"/>
    <!--[设置]  服务传输业务对象时的序列化类型 -->
    <property name="preferSerializeType" value="hessian2"/>
</bean>
```

3.13. 超时配置

本文介绍开发HSF应用过程中如何进行超时配置。

前提条件

在开发应用前，您已经完成以下工作：

- 配置EDASSAE的私服地址和轻量级配置及注册中心
- 启动轻量级配置及注册中心

背景信息

有关网络调用的请求，都需要配置超时，HSF的默认超时时间是3000ms。客户端和服务端都可以设置超时，默认优先采用客户端的配置，如果客户端没有配置，使用服务端的超时配置。 在服务端设置超时时，需要考虑到业务本身的执行耗时，加上序列化和网络通讯的时间。所以推荐服务端给每个服务都配置个默认的时间。当然客户端也可以根据自己的业务场景配置超时时间，例如一些前端应用，需要用户快速看到结果，可以把超时时间设置小一些。

配置的作用范围、作用域，按照优先级由高到低如下表所示。

优先级	API	范围	作用域

优先级	API	范围	作用域
0	com.taobao.hsf.util.RequestCtxUtil#setRequestTimeout	客户端	单次调用
1	HSFApiConsumerBean#setMethodSpecials	客户端	方法
2	HSFApiConsumerBean#setClientTimeout	客户端	接口
3	-DdefaultHsfClientTimeout	客户端	所有接口
4	HSFApiProviderBean#setMethodSpecials	服务端	方法
5	HSFApiProviderBean#setClientTimeout	服务端	接口

说明 客户端配置优先于服务端，方法优先于接口。

客户端超时配置

- API形式配置HSF服务。

  配置HSFApiConsumerBean的clientTimeout属性，单位是ms，我们把接口的超时配置为1000ms，方法queryOrder配置为100ms, 代码如下。

```
HSFApiConsumerBean consumerBean = new HSFApiConsumerBean();
// 接口级别超时配置consumerBean.setClientTimeout(1000);
//xxx
MethodSpecial methodSpecial = new MethodSpecial();
methodSpecial.setMethodName("queryOrder");
// 方法级别超时配置，优先于接口超时配置methodSpecial.setClientTimeout(100);
consumerBean.setMethodSpecials(new MethodSpecial[]{methodSpecial});
```

- Spring配置HSF服务。

  Spring框架是在应用中广泛使用的组件，如果不想通过API的形式配置HSF服务，可以使用Spring XML的形式进行配置，上述例子中的API配置等同于如下XML配置。

```
<bean id="CallHelloWorld" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean">
    ...
    <property name="clientTimeout" value="1000" />
    <property name="methodSpecials">
      <list>
        <bean class="com.taobao.hsf.model.metadata.MethodSpecial">
          <property name="methodName" value="queryOrder" />
          <property name="clientTimeout" value="100" />
        </bean>
      </list>
    </property>
    ...
</bean>
```

- 注解配置。

  SpringBoot广泛使用的今天，使用注解装配SpringBean也成为一种选择，HSF也支持使用注解进行配置，用来订阅服务。

  i. 在项目中增加依赖starter。

  ```
  <dependency>
      <groupId>com.alibaba.boot</groupId>
      <artifactId>pandora-hsf-spring-boot-starter</artifactId>
  </dependency>
  ```

  ii. 在代码中注解配置。

  通常一个HSF Consumer需要在多个地方使用，但并不需要在每次使用的地方都用 @HSFConsumer来标记。只需要写一个统一个Config类，然后在其它需要使用的地方，直接 @Autowired注入即可上述例子中的API配置等同于如下注解配置。

  ```
   @HSFConsumer(clientTimeout = 1000, methodSpecials = @HSFConsumer.ConsumerMethodSpeci
  al(methodName = "queryOrder", clientTimeout = "100"))
   private OderService orderService;
  ```

- 客户端全局接口超时配置。
  - 在启动参数中添加 `-DdefaultHsfClientTimeout=100`
  - 在代码中添加 `System.setProperty("defaultHsfClientTimeout", “100”)`

服务端方法超时配置

- API配置HSF服务。

  配置HSFApiProviderBean的clientTimeout属性，单位是ms，代码如下。

  ```
  HSFApiProviderBean providerBean = new HSFApiProviderBean();
  // 接口级别超时配置providerBean.setClientTimeout(1000);
  //xxx
  MethodSpecial methodSpecial = new MethodSpecial();
  methodSpecial.setMethodName("queryOrder");
  // 方法级别超时配置，优先于接口超时配置methodSpecial.setClientTimeout(100);
  providerBean.setMethodSpecials(new MethodSpecial[]{methodSpecial});
  ```

- Spring配置HSF服务。

Spring框架是在应用中广泛使用的组件，如果不想通过API的形式配置HSF服务，可以使用Spring XML的形式进行配置，上述例子中的API配置等同于如下XML配置。

```
<bean class="com.taobao.hsf.app.spring.util.HSFSpringProviderBean" init-method="init">
    ...
    <property name="clientTimeout" value="1000" />
    <property name="methodSpecials">
      <list>
        <bean class="com.taobao.hsf.model.metadata.MethodSpecial">
          <property name="methodName" value="queryOrder" />
          <property name="clientTimeout" value="2000" />
        </bean>
      </list>
    </property>
    ...
</bean>
```

- 注解配置HSF服务。

  注入即可上述例子中的API配置等同于如下注解配置。

```
@HSFProvider(serviceInterface = OrderService.class, clientTimeout = 3000)
public class OrderServiceImpl implements OrderService {
    @Autowired
    private OrderDAO orderDAO;
    @Override
    public OrderModel queryOrder(Long id) {
        return orderDAO.queryOrder(id);
    }
}
```

3.14. 服务端线程池配置

本文介绍开发HSF应用过程中如何进行服务端线程池配置。

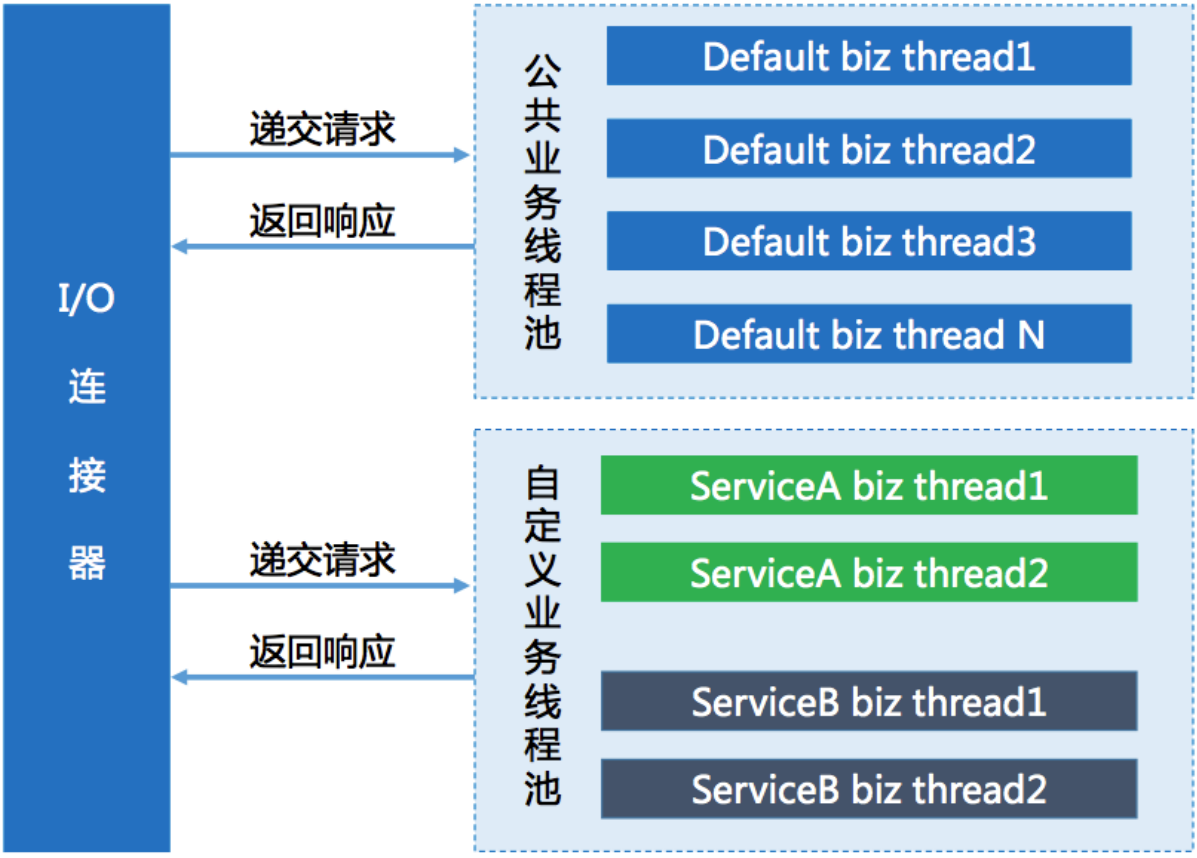
前提条件

在开发应用前，您已经完成以下工作：

-
-

服务线程池业务示意图

HSF服务端线程池主要分为IO线程和业务线程，其中IO线程模型就是netty reactor网络模型中使用的。我们主要讨论业务线程池的配置。业务线程池分为默认业务线程池和服务线程池，其中服务线程池是从默认线程池中分割出来的。



默认线程池配置

服务端线程池是用来执行业务逻辑的线程池，线程池默认的core size是50，max size是720, keepAliveTime 500s。队列使用的是SynchronousQueue，没有缓存队列，不会堆积用户请求。 当服务端线程池所有线程（720）都在处理请求时，对于新的请求，会立即拒绝，返回Thread pool is full异常。可以使用下面VM参数（-D参数）进行配置。

- 线程池最小配置：`-Dhsf.server.min.poolsize`
- 线程池最大配置：`-Dhsf.server.max.poolsize`
- 线程收敛的存活时间：`-Dhsf.server.thread.keepalive`

服务线程池配置

对于一些慢服务、并发高，可以为其单独配置线程池，以免占用过多的业务线程，影响应用的其他服务的调用。

- API形式配置HSF服务

```
HSFApiProviderBean hsfApiProviderBean = new HSFApiProviderBean();
//...
hsfApiProviderBean.setCorePoolSize("50");
hsfApiProviderBean.setMaxPoolSize("200");
```

- Spring配置HSF服务

  Spring框架是在应用中广泛使用的组件，如果不想通过API的形式配置HSF服务，可以使用Spring XML的形式进行配置，上述例子中的API配置等同于如下XML配置。

```
<bean class="com.taobao.hsf.app.spring.util.HSFSpringProviderBean" init-method="init">
    <!--[设置] 发布服务的接口 -->
    <property name="serviceInterface" value="com.alibaba.middleware.hsf.guide.api.service
.OrderService"/>
    <property name="corePoolSize" value="50" />
    <property name="maxPoolSize" value="200" />
</bean>
```

- 注解配置HSF服务

  SpringBoot广泛使用的今天，使用注解装配SpringBean也成为一种选择，HSF也支持使用注解进行配置，用来发布服务。

  i. 在项目中增加依赖starter。

  ```
  <dependency>
      <groupId>com.alibaba.boot</groupId>
      <artifactId>pandora-hsf-spring-boot-starter</artifactId>
  </dependency>
  ```

  ii. 将 @HSFProvider配置到实现的类型。

  上述例子中的API配置等同于如下注解配置。

  ```
  @HSFProvider(serviceInterface = OrderService.class, corePoolSize = 50, maxPoolSize =
  200)
  public class OrderServiceImpl implements OrderService {
      @Autowired
      private OrderDAO orderDAO;
      @Override
      public OrderModel queryOrder(Long id) {
          return orderDAO.queryOrder(id);
      }
  }
  ```

3.15. API手册

在HSF应用的API中，最关键的是创建ProviderBean和ConsumerBean相关的API。

背景信息

根据用户使用的场景不同，主要分为4个关键的类。

- com.taobao.hsf.app.api.util.HSFApiProviderBean: 通过API编程的方式创建Provider Bean
- com.taobao.hsf.app.api.util.HSFApiConsumerBean: 通过API编程的方式创建Consumer Bean
- com.taobao.hsf.app.spring.util.HSFSpringProviderBean: 通过Spirng配置的方式创建Provider Bean
- com.taobao.hsf.app.spring.util.HSFSpringConsumerBean: 通过Spirng配置的方式创建Consumer Bean

其中，HSFSpringXxxBean的配置属性与HSFApiXxxBean的setter方法相对应。接下来就分别以ProviderBean和ConsumerBean的视角介绍这4个类的API。

ProviderBean

- API编程方式 - HSFApiProviderBean

  通过配置并初始化com.taobao.hsf.app.api.util.HSFApiProviderBean即可完成HSF服务的发布。

对于一个服务，com.taobao.hsf.app.api.util.HSFApiProviderBean的配置及初始化过程只需配置一次。此外，考虑到HSFApiProviderBean对象比较复杂，建议缓存。

- 范例代码：

```
// 实例化并配置Provider Bean
HSFApiProviderBean hsfApiProviderBean = new HSFApiProviderBean();
hsfApiProviderBean.setServiceInterface("com.taobao.hsf.test.HelloWorldService");
hsfApiProviderBean.setTarget(target); // target为serviceInterface指定接口的实现对象hsfApi
ProviderBean.setServiceVersion("1.0.0");
hsfApiProviderBean.setServiceGroup("HSF");
// 初始化Provider Bean，发布服务hsfApiProviderBean.init();
```

- 可配置属性表：

  除上述范例代码中设置的属性，HSFApiProviderBean还包含许多其他可配置的属性，均可通过对应的setter方法进行设置。

属性名	类型	是否必选	默认值	含义
serviceInterface	String	是	无	设置HSF服务对外提供的业务接口。客户端通过此属性进行订阅。
target	Object	是	无	设置serviceInterface指定接口的服务实现对象。
serviceVersion	String	否	1.0.0	设置服务的版本号。 客户端通过此属性进行订阅。
serviceGroup	String	否	HSF	设置服务的组别。客户端通过此属性进行订阅。
serviceDesc	String	否	null	设置服务的描述，从而方便管理。
clientTimeout	int	否	3000	设置响应超时时间（单位：毫秒）。如果服务端在设置的时间内没有返回，则抛出HSFTimeOutException。

属性名	类型	是否必选	默认值	含义
methodSpecials	MethodSpecial[]	否	空	设置服务中某些方法的响应超时时间。通过设置MethodSpecial.methodName指定方法名，通过设置MethodSpecial.clientTimout指定当前方法的超时时间，优先级高于当前服务端的clientTimeout。
preferSerializeType	String	否	hessian2	针对HSF2，设置服务的请求参数和响应结果的序列化方式。可选值有java、hessian、hessian2、json和kryo。
corePoolSize	值为整型的String	否	0	配置服务单独的线程池，并指定最小活跃线程数量。若不设置该属性，则默认使用HSF服务端的公共线程池。
maxPoolSize	值为整型的String	否	0	配置服务单独的线程池，并指定最大活跃线程数量。若不设置该属性，则默认使用HSF服务端的公共线程池。

- Spring配置方式 - HSFSpringProviderBean

  通过在Spring配置文件中，配置一个class为 com.taobao.hsf.app.spring.util.HSFSpringProviderBean的bean，即可完成HSF服务的发布。

  示例代码

```
<bean id="helloWorldService" class="com.taobao.hsf.test.HelloWorldService" />
<bean class="com.taobao.hsf.app.spring.util.HSFSpringProviderBean" init-method="init">
    <!-- [必选] 设置HSF服务对外提供的业务接口 -->
    <property name="serviceInterface" value="com.taobao.hsf.test.HelloWorldService" />
    <!-- [必选] 设置serviceInterface指定接口的服务实现对象，即需要发布为HSF服务的spring bean id
-->
    <property name="target" ref="helloWorldService" />
    <!-- [可选] 设置服务的版本号，默认为1.0.0 -->
    <property name="serviceVersion" value="1.0.0" />
    <!-- [可选] 设置服务的组别，默认为HSF -->
    <property name="serviceGroup" value="HSF" />
    <!-- [可选] 设置服务的描述，从而方便管理，默认为null -->
    <property name="serviceDesc" value="HelloWorldService providered by HSF" />
    <!-- [可选] 设置响应超时时间（单位：毫秒）。如果服务端在设置的时间内没有返回，则抛出HSFTimeOutE
xception -->
    <!-- 默认为3000 ms -->
    <property name="clientTimeout" value="3000"/>
    <!-- [可选] 设置服务中某些方法的响应超时时间。优先级高于上面的clientTimeout -->
    <!-- 通过设置MethodSpecial.methodName指定方法名，MethodSpecial.clientTimout指定方法的超时
时间 -->
    <property name="methodSpecials">
        <list>
            <bean class="com.taobao.hsf.model.metadata.MethodSpecial">
                <property name="methodName" value="sum" />
                <property name="clientTimeout" value="2000" />
            </bean>
        </list>
    </property>
    <!-- [可选] 设置服务的请求参数和响应结果的序列化方式。可选值包含java、hessian、hessian2、json
和kryo -->
    <!-- 默认为hessian2 -->
    <property name="preferSerializeType" value="hessian2"/>
    <!-- [可选] 配置服务单独的线程池，并指定最小活跃线程数量。若不设置该属性，则默认使用HSF服务端的
公共线程池 -->
    <property name="corePoolSize" value="10"/>
    <!-- [可选] 配置服务单独的线程池，并指定最大活跃线程数量。若不设置该属性，则默认使用HSF服务端的
公共线程池 -->
    <property name="maxPoolSize" value="60"/>
</bean>
```

ConsumerBean

- API配置方式 - HSFApiConsumerBean

  通过配置并初始化com.taobao.hsf.app.api.util.HSFApiConsumerBean完成HSF服务的订阅。

  对于同一个服务，com.taobao.hsf.app.api.util.HSFApiConsumerBean的配置及初始化过程只需配置一次。

  由于HSFApiConsumerBean对象内容多，建议将该对象以及获取到的HSF代理进行缓存。

  > **说明** 在HSF内部HSFApiConsumerBean对服务的配置是缓存起来的。即如果对某个订阅的服务进行了多次配置，只有第一次的配置生效。

- 范例代码

```
// 实例化并配置Consumer Bean
HSFApiConsumerBean hsfApiConsumerBean = new HSFApiConsumerBean();
hsfApiConsumerBean.setInterfaceName("com.taobao.hsf.test.HelloWorldService");
hsfApiConsumerBean.setVersion("1.0.0");
hsfApiConsumerBean.setGroup("HSF");
// 初始化Consumer Bean，订阅服务
// true表示等待地址推送（超时时间为3000毫秒），默认为false（异步）
hsfApiConsumerBean.init(true);
// 获取HSF代理HelloWorldService helloWorldService = (HelloWorldService) hsfApiConsumerBean.getObject();
// 发起HSF调用String helloStr = helloWorldService.sayHello("Li Lei");
```

- 可配置属性表

  除上述范例代码中设置的属性，HSFApiConsumerBean还包含许多其他可配置的属性，均可通过对应的setter方法进行设置。

属性名	类型	是否必选	默认值	含义
interfaceName	String	是	无	设置需要订阅服务的接口名。 客户端通过此属性进行订阅。
version	String	是	无	设置需要订阅服务的版本号。 客户端通过此属性进行订阅。
group	String	是	无	设置需要订阅服务的组别。 客户端通过此属性进行订阅。
clientTimeout	int	否	无	设置请求超时时间（单位：毫秒）。如果客户端在设置的时间内没有收到服务端响应，则抛出HSFTimeOutException。 若客户端设置了clientTimeout，则优先级高于服务端设置的clientTimeout。否则，在服务的远程调用过程中，使用服务端设置的clientTimeout。

属性名	类型	是否必选	默认值	含义
methodSpecials	MethodSpecial[]	否	空	设置服务中某些方法的请求超时时间。通过设置MethodSpecial.methodName指定方法名，通过设置MethodSpecial.clientTimout指定当前方法的超时时间，优先级高于当前客户端的clientTimeout。
maxWaitTimeForCsAddress	int	否	无	设置同步等待ConfigServer推送地址的时间（单位：毫秒），从而避免因地址还未推送到就发起服务调用造成的HSFAddressNotFoundException。一般建议设置为5000毫秒，即可满足推送等待时间。

属性名	类型	是否必选	默认值	含义
asyncallMethods	List	否	null	设置需要异步调用的方法列表。List中的每一个字符串的格式为： ■ name：方法名 ■ type：异步调用类型 ■ listener：监听器 其中listener只对callback类型的异步调用生效。type的类型有： ■ future：通过Future的方式去获取请求执行的结果。 ■ callback：当远程服务的调用完成后，HSF会使用响应结果回调此处配置的listener，该listener需要实现HSFResponseCallback接口。
proxyStyle	String	否	jdk	设置服务的代理模式，一般不用配置。如果要拦截这个consumer bean，需要配置成javassist。

- Spring配置方式 - HSFSpringConsumerBean

  通过在Spring配置文件中，配置一个class为com.taobao.hsf.app.api.util.HSFSpringConsumerBean的bean，即可实现服务的订阅。

  示例代码

```
<bean id="helloWorldService" class="com.taobao.hsf.app.spring.util.HSFSpringConsumerBean"
>
    <!-- [必选] 设置需要订阅服务的接口名 -->
    <property name="interfaceName" value="com.taobao.hsf.test.HelloWorldService" />
    <!-- [必选] 设置需要订阅服务的版本号 -->
    <property name="version" value="1.0.0" />
    <!-- [必选] 设置需要订阅服务的组别 -->
    <property name="group" value="HSF" />
    <!-- [可选] 设置请求超时时间（单位：毫秒）。如果客户端在设置的时间内没有收到服务端响应，则抛出HS
FTimeOutException -->
    <!-- 若客户端设置了clientTimeout，则优先级高于服务端设置的clientTimeout。否则，在服务的远程调
用过程中，使用服务端设置的clientTimeout。-->
    <property name="clientTimeout" value="3000" />
    <!-- [可选] 设置服务中某些方法的请求超时时间，优先级高于当前客户端的clientTimeout -->
    <!-- 通过设置MethodSpecial.methodName指定方法名，通过设置MethodSpecial.clientTimout指定当
前方法的超时时间 -->
    <property name="methodSpecials">
        <list>
            <bean class="com.taobao.hsf.model.metadata.MethodSpecial">
                <property name="methodName" value="sum" />
                <property name="clientTimeout" value="2000" />
            </bean>
        </list>
    </property>
    <!-- [可选] 设置同步等待ConfigServer推送地址的时间（单位：毫秒）-->
    <!-- 从而避免因地址还未推送到就发起服务调用造成的HSFAddressNotFoundException -->
    <!-- 一般建议设置为5000毫秒，即可满足推送等待时间 -->
    <property name="maxWaitTimeForCsAddress"  value="5000"/>
    <!-- [可选] 设置需要异步调用的方法列表。List中的每一个字符串的格式为：-->
    <!-- name：方法名 -->
    <!-- type：异步调用类型 -->
    <!-- listener：监听器 -->
    <!-- 其中，listener只对callback类型的异步调用生效-->
    <!-- type的类型有： -->
    <!-- future：通过Future的方式去获取请求执行的结果 -->
    <!-- callback：当远程服务的调用完成后，HSF会使用响应结果回调此处配置的listener，该listener需
要实现HSFResponseCallback接口 -->
    <property name="asyncallMethods">
        <list>
            <value>name:sayHello;type:callback;listener:com.taobao.hsf.test.service.Hello
WorldServiceCallbackHandler</value>
        </list>
    </property>
    <!-- [可选] 设置服务的代理模式，一般不用配置。如果要拦截这个consumer bean，需要配置成javassis
t -->
    <property name="proxyStyle" value="jdk" />
</bean>
```

3.16. JVM -D启动配置参数

本文介绍HSF应用开发时JVM -D启动参数的配置信息。

-D `hsf.server.port`

指定HSF的启动服务绑定端口，默认为12200。如果在本地启动多个HSF Provider，则需要修改此端口。

-D `hsf.server.max.poolsize`

指定HSF的服务端最大线程池大小，默认值为 `720` 。

-D `hsf.server.min.poolsize`

指定HSF的服务端最小线程池大小，默认值为 `50` 。

-D `hsf.client.localcall`

打开或者关闭本地优先调用，默认值为 `true` 。

-D `pandora.qos.port`

指定Pandora监控端口，默认值为 `12201` 。如果在本地启动多个HSF Provider，则需要修改此端口。

-D `hsf.http.enable`

是否开启HTTP端口，默认值为 `true` 。

-D `hsf.http.port`

指定HSF暴露的HTTP接口，默认值为 `12220` 。如果在本地启动多个HSF Provider，则需要修改此端口。

-D `hsf.run.mode`

指定HSF客户端是否指定target进行调用，即绕开ConfigServer。值为 `1` ，表示不允许指定target调用；值为 `0` ，表示允许指定target调用。默认值为 `1` 时，不推荐指定为 `0` 。

-D `hsf.shuthook.wait`

HSF优雅关闭的等待时间，单位是ms，默认值是 `10000` 。

-D `hsf.publish.delayed`

是否所有的服务都需要延迟发布，默认是false，不需要延迟发布 。

-D `hsf.server.ip`

指定需要绑定的IP地址。在多网卡情况下默认绑定第一个网卡，通过该参数指定需要绑定的IP。

-D `HsfBindHost`

指定需要绑定的Host。在多网卡情况下默认绑定和上报给地址注册中心第一个网卡的IP地址，通过该参数可以指定需要绑定的Host，例如 `-DHsfBindHost=0.0.0.0` 将HSF Server端口绑定本机所有网卡。

-D `hsf.publish.interval=400`

指定发布服务之间的时间间隔。HSF服务发布时会瞬间暴露出去，在应用启动时如果承受不住压力，可以配置该参数。默认值是400，单位ms。

-D `hsf.client.low.water.mark=32` -D `hsf.client.high.water.mark=64` -D `hsf.server.low.water.mark=32` -D `hsf.server.high.water.mark=64`

指定客户端或者服务端的每个channel写缓冲的限制。

- 客户端每个channel的写缓冲的限制，单位为KB，一旦超过高水位，channel禁写，新的请求放弃写出，直接报错。禁写之后，等到缓冲区低于低水位才能恢复。
- 服务端每个channel的写缓冲的限制，单位为KB，超过高水位时，新的响应放弃写出，客户端收不到响应会超时。缓冲区低于低水位时才能恢复写。
- 高低水位需成对设置，并且需要高水位大于低水位。

-D `hsf.generic.remove.class=true`

获取泛化调用的结果，但不输出 `class` 字段信息。

-D `defaultHsfClientTimeout`

全局的客户端超时配置。

-D `hsf.invocation.timeout.sensitive`

`hsf.invocation.timeout.sensitive` 默认值设置为false，决定HSF调用时间是否包含创建连接、选址等耗时逻辑。

4.应用迁移

4.1. 应用迁移概述

如果您的应用已经部署到生产环境并处于正常运行状态，为了保持业务不中断运行，并且不发生数据丢失，您可以采用平滑迁移的方式将应用迁移至SAE。

迁移到SAE的价值

- SAE为应用部署提供了灵活的启动参数配置、可视化的部署流程、优雅上下线服务和分批发布等功能，让您的应用发布可配、可查、可控。
- SAE提供了服务发现与配置管理功能，您无需运维Eureka、ZooKeeper、Consul等中间件组件，可以直接使用SAE提供的商业版服务发现与配置管理。
- SAE控制台提供了统一的服务治理，目前支持查询发布和消费的服务详情。
- SAE提供了动态扩、缩容功能，可以根据流量高峰和低谷实时地为您的应用扩容和缩容。
- SAE提供了高级监控功能，除了支持基本的实例信息查询外，还支持微服务调用链查询、系统调用拓扑图、慢 SQL 查询等高级监控功能。
- 对于Spring Cloud框架应用SAE提供限流降级功能，保证您的应用高可用。
- 对于Spring Cloud框架应用SAE提供了全链路灰度功能，满足您的应用在迭代、更新时通过灰度进行小规模验证的需求。

什么是平滑迁移

如果您的Spring Cloud集群及应用已经部署在生产环境并处于正常运行状态中，现在需要将集群迁移到SAE并享用完整的SAE功能，在迁移过程中，业务需要平稳运行而不中断，而保证应用平台运行不中断迁移到SAE即为平滑迁移。

> **说明** 如果您的集群尚未在生产环境中运行，或者您可以接受停机迁移，则无须参考本文进行平滑迁移，可直接将应用在本地开发完再部署到SAE，详情请参见如下章节内容。
>
> - Spring Cloud应用：将Spring Cloud应用托管到SAE
> - Dubbo应用：将Dubbo应用托管到SAE

迁移方案

- 切流迁移方案

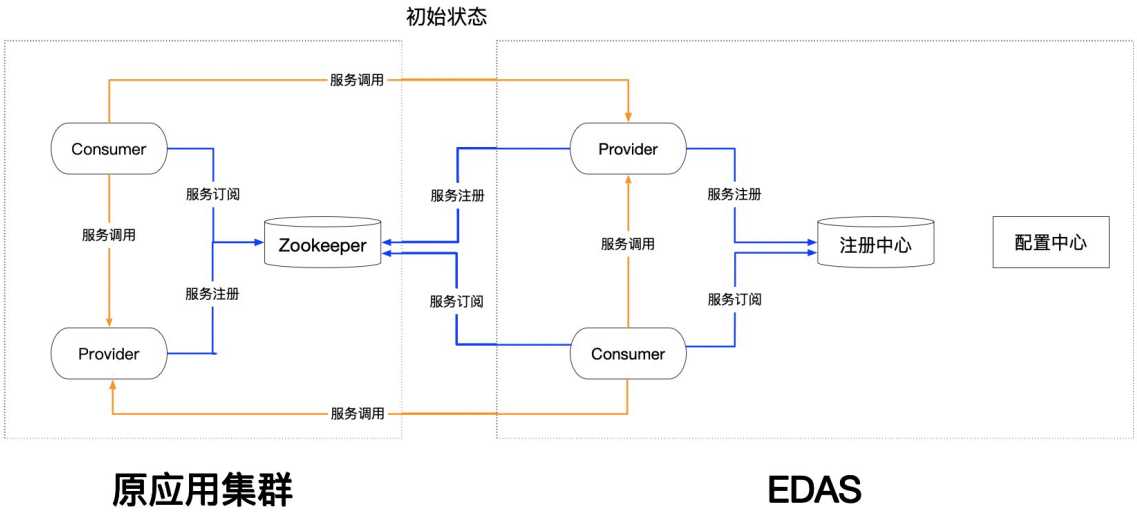
  使用Dubbo将原有的服务注册中心切换到SAE ConfigServer，开发新的应用部署到SAE，最后通过SLB和域名配置来进行切流。

  如果选择此方案，请参考微服务场景指引 开发应用。

- 双注册和双订阅迁移方案

  双注册和双订阅迁移方案是指在应用迁移时同时接入两个注册中心（原有注册中心和 SAE 注册中心）以保证已迁移的应用和未迁移的应用之间的相互调用。

本方案实现架构图如下：



- 已迁移的应用和未迁移的应用可以互相发现，从而实现互相调用，保证了业务的连续性。
- 使用方式简单，仅需添加依赖，并修改极少的代码，可以实现双注册和双订阅。
- 支持查看消费者服务调用列表的详情，实时地查看到迁移的进度。
- 支持在不需要重启应用的情况下，动态地变更服务注册的策略和服务订阅的策略，只需要重启一次应用就可以完成迁移。

4.2. 将Spring Cloud框架应用平滑迁移至SAE

如果您的Spring Cloud集群（包含多个应用）已经部署在阿里云上，您可以将应用迁移至SAE。本文说明了如何将应用平滑迁移到SAE中，以及实现基本的服务注册与发现。如果您的Spring Cloud集群尚未部署至阿里云，请提交工单或联系SAE技术支持人员获取完整的上云及迁移方案。

迁移流程

1. 迁移应用

   迁移的应用通常是无状态的，需要先进行应用迁移。

2. （可选）迁移SLB或修改域名配置

   在应用迁移完成后，您还需要迁移SLB或修改域名配置。

   - SLB
     - 如果您的应用在迁移之前已经使用SLB，应用迁移后可以复用该SLB。您可以根据您的实际需求选择绑定SLB的策略，具体操作，请参见为应用绑定SLB。
     - 如果您的应用在迁移之前没有使用SLB，建议在迁移完入口应用（如流程图中所示的API Gateway）后，为该应用创建并绑定一个新SLB。
     - 迁移方案中，推荐使用双注册和双订阅方案，以节约ECS成本。如果由于某种原因（例如原ECS端口被占用）不能复用原ECS，那么需要采用切流迁移方案，添加新的ECS用于应用迁移。在应用迁移完成后，依据迁移前应用是否使用SLB，选择复用SLB或创建SLB并绑定到迁移后应用。
   - 域名
     - 如果迁移后的应用可以复用SLB，则域名配置无需修改。
     - 如果迁移后的应用需要创建新的SLB并绑定，则需要在域名中添加新的SLB配置，并删除原来不再使用的SLB。具体操作，请参见域名DNS修改。

3. （可选）迁移存储和消息队列

   - 如果应用迁移前已经部署在阿里云上，同时存储和消息队列同样使用了阿里云相关产品（如RDS、MQ等），那么应用迁移完成后，迁移前的存储和消息队列无需迁移。
   - 如果应用迁移前没有部署在阿里云上，请提交工单或联系SAE技术支持人员为您提供完整的上云及迁移到方案。

本文以Demo应用演示平滑迁移。Demo下载：Demo。

迁移方案

迁移应用有两种方案，切流迁移、双注册和双订阅迁移方案。两种方案均可保证应用正常运行不中断情况下完成平滑迁移。

> **说明** 本文将主要介绍双注册和双订阅方案。

- 切流迁移方案

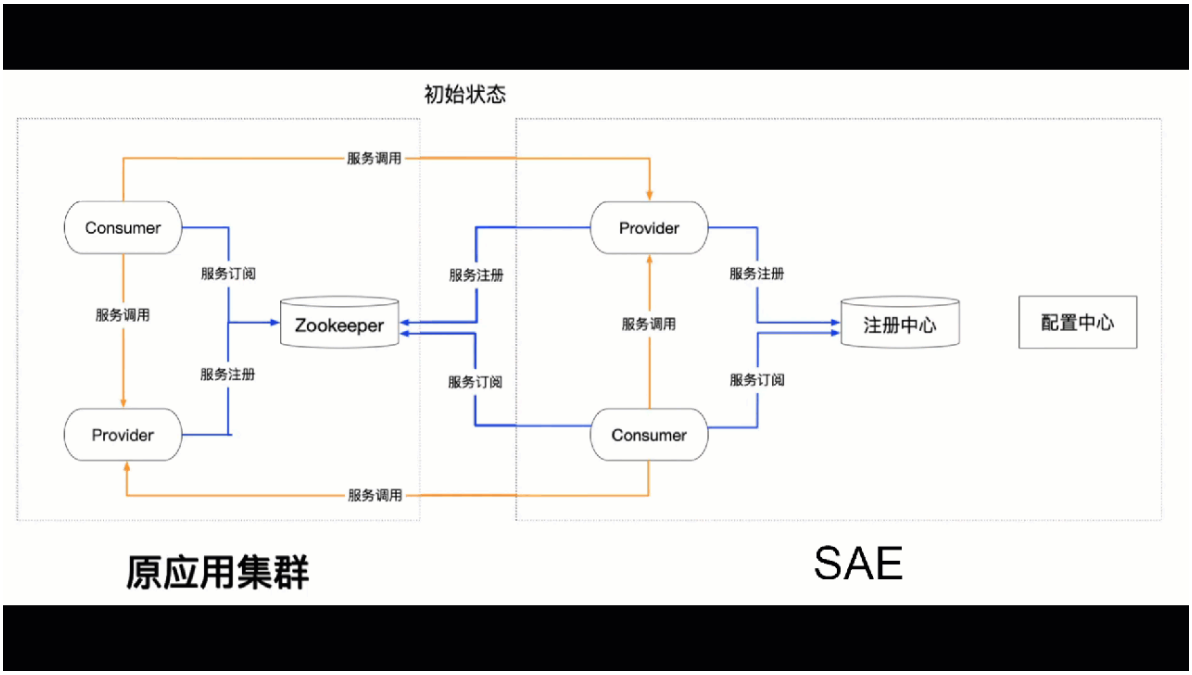
  使用Spring Cloud Alibaba将原有的服务注册中心切换到Nacos。开发一套新的应用部署到SAE，最后通过SLB和域名配置来进行切流。

  如果选择此方案，请参见将Spring Cloud应用托管到SAE。

- 双注册和双订阅迁移方案

  双注册和双订阅迁移方案指在应用迁移时同时接入两个注册中心（原有注册中心和SAE注册中心），以保证已迁移的应用和未迁移的应用之间可相互调用。

  双注册和双订阅平滑迁移方案架构图如下：



  - 已迁移的应用和未迁移的应用之间可以互相发现，从而实现互相调用，保证了业务的连续性。
  - 使用方式简单，仅需要添加依赖，并修改极少代码，实现双注册和双订阅。
  - 支持查看消费者服务调用列表的详情，实时地查看到迁移的进度。
  - 支持在不重启应用的情况下，动态地变更服务注册的策略和服务订阅的策略，只需要重启一次应用就可以完成迁移。

迁移第一个应用

1. 制定应用迁移优先级。

   选择迁移需求优先级高的应用，建议从最下层Provider开始迁移。如果调用链路太复杂难分析，可以任意选一应用进行迁移。

2. 在应用程序中添加依赖并修改配置。

   i. 在 `pom.xml` 文件中添加 `spring-cloud-starter-alibaba-nacos-discovery` 依赖。

   ```
   <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        <version>{相应的版本}</version>
    </dependency>
   ```

   ii. 在 `application.properties` 中添加nacos-server的IP地址。

   ```
   spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
   ```

   iii. 添加多注册中心依赖 `edas-sc-migration-starter` 。

   Spring Cloud默认依赖中只能引入一个注册中心，存在多个注册中心时，启动会异常。如果需要至支持多注册，需要添加依赖 `edas-sc-migration-starter` 。

   ```
   <dependency>
        <groupId>com.alibaba.edas</groupId>
        <artifactId>edas-sc-migration-starter</artifactId>
        <version>1.0.2</version>
    </dependency>
   ```

3. 修改RibbonClients默认配置。

   Ribbon是实现负载均衡组件，应用从多个注册中心订阅服务，需要修改Ribbon配置。在应用启动的主类中，将RibbonClients默认配置修改为 `MigrationRibbonConfiguration` 。

   假设原有的应用主类启动代码如下：

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

   修改后应用主类启动代码如下：

   ```
   @SpringBootApplication
    @RibbonClients(defaultConfiguration = MigrationRibbonConfiguration.class)
    public class ConsumerApplication {
        public static void main(String[] args) {
            SpringApplication.run(ConsumerApplication.class, args);
        }
    }
   ```

> **说明**
>
> 在本地修改应用或者应用部署到SAE后，如果对应用有其他控制需求，例如将应用注册到某些注册中心或从某些注册中心订阅，则可以通过Spring Cloud Config或Nacos Config动态地调整配置，而无需重启应用。调整配置的方法，请参见动态调整服务注册和订阅方式。
>
> 要通过Spring Cloud Config或Nacos Config动态地调整配置，需要在应用中添加配置管理依赖和修改配置。如果使用Spring Cloud Config，请参见相应的开源文档。如果使用Nacos Config，请参见实现配置管理。

4. 将应用部署到SAE。

   根据实际需求将应用部署到SAE。具体操作，请参见部署应用概述。

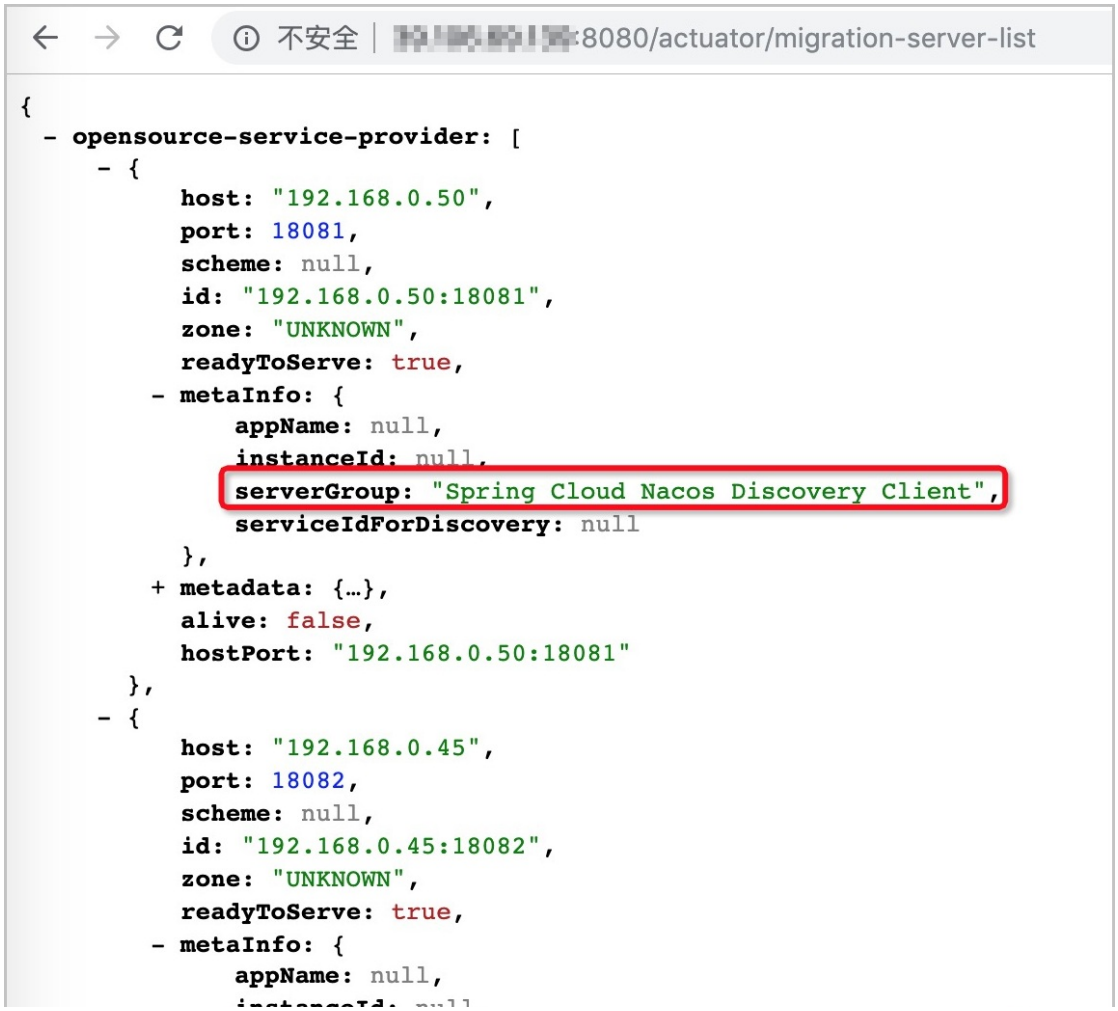
5. 结果验证。

   i. 观察业务运行是否正常。

   ii. 查看服务订阅监控。

   - Spring Boot 1.x版本：http://ip:port/dubboRegistry
   - Spring Boot 2.x版本：http://ip:port/actuator/dubboRegistry

   如果应用开启了Spring Boot Actuator监控功能，请访问Actuator查看此应用订阅的各服务的RibbonServerList信息。Actuator地址如下： metaInfo中serverGroup字段表示此节点的服务注册中心。



迁移其他所有应用

按照迁移第一个应用的步骤依次将所有应用迁移到SAE。

清理迁移配置

迁移完成后，删除原有的注册中心配置和迁移过程专用的依赖 `edas-sc-migration-starter` 。

`edas-sc-migration-starter` 迁移专用的starter，长期使用对业务的稳定性没有影响，对于Ribbon负载均衡实现有一定的局限性，建议在迁移完毕后删除，并在业务量较小的时间段内进行分批重启应用。

- 动态调整服务注册和订阅方式

  应用迁移过程中，可以通过SAE配置管理功能动态变更服务注册和订阅方式。

- 动态调整服务订阅

  系统默认订阅策略是从所有注册中心订阅，并对数据进行聚合。

  您可以通过SAE的配置管理修改 `spring.cloud.edas.migration.subscribes` 属性，选择具体的注册中心订阅数据。

  ```
  spring.cloud.edas.migration.subscribes=nacos,eureka # 同时从Eureka和Nacos订阅服务。
  spring.cloud.edas.migration.subscribes=nacos        # 只从Nacos订阅服务。
  ```

- 动态变更服务注册

  系统默认订阅策略是从所有注册中心订阅。

  您可以通过SAE的配置管理来调整服务注册中心。

  通过修改 `spring.cloud.edas.migration.registry.excludes` 属性关闭指定的注册中心。

```
spring.cloud.edas.migration.registry.excludes=    #默认值为空，注册到所有的服务注册中心。
spring.cloud.edas.migration.registry.excludes=eureka    #关闭Eureka的注册。
spring.cloud.edas.migration.registry.excludes=nacos,eureka    #关闭Nacos和Eureka的注册。
```

应用运行时如需要动态修改服务注册策略，可使用Spring Cloud配置管理功能在运行时修改此属性。

更多信息

- 您在SAE部署完应用后，可以对应用进行更新、扩缩容、启停、删除应用等生命周期管理操作。具体操作，请参见管理应用生命周期。
- 您在SAE部署完应用后，可以对应用进行自动弹性伸缩、SLB绑定和批量启停等提升应用性能的操作。具体操作，请参见以下文档：
  - 绑定SLB
  - 配置弹性伸缩策略
  - 一键启停应用
  - 配置管理
  - 变更实例规格
- 您在SAE部署完应用后，还可以对应用进行日志管理、监控管理、应用事件查看和变更记录查看等聚焦应用运行状态的操作。具体操作，请参见以下文档：
  - 日志管理
  - 监控管理
  - 应用事件查看
  - 变更记录查看
  - 使用Webshell诊断应用

4.3. 将Dubbo应用平滑迁移至SAE

如果您的Dubbo应用已经部署在阿里云上，您可以将应用迁移至Serverless应用引擎SAE（Serverless App Engine）。本文介绍如何将应用平滑迁移到SAE中。如果您的Dubbo应用尚未部署至阿里云，请提交工单或联系SAE技术支持人员获取完整的上云及迁移方案。

背景信息

应用迁移概述

步骤一：迁移第一个应用

本文以Demo应用演示平滑迁移。关于Demo应用的下载地址，请参见Demo。

1. 制定应用迁移优先级。

   选择迁移需求优先级高的应用，建议从最下层Provider开始迁移。如果调用链路太复杂难分析，可以任意选一应用进行迁移。

2. 在应用程序中添加依赖并修改配置。

   > **说明** 本文介绍的迁移方案为双注册双订阅方案。

i. 在*pom.xml*文件中添加 `edas-dubbo-migration-bom` 依赖。

```
<dependency>
    <groupId>com.alibaba.edas</groupId>
    <artifactId>edas-dubbo-migration-bom</artifactId>
    <version>2.6.5.1</version>
    <type>pom</type>
</dependency>
```

ii. 在*application.properties*中添加SAE注册中心的IP地址。

```
dubbo.registry.address = edas-migration://192.168.124.15:9999?service-registry=edas
://127.X.X.X:8080,zookeeper://192.168.20.219:2181&reference-registry=zookeeper://19
2.168.20.219:2181&config-address=127.X.X.X:8848
```

> **注意** 如果是非Spring Boot应用，您需要在dubbo.properties或者对应的Spring配置文件中进行设置。

- edas-migration://192.168.124.15:9999：多注册中心的头部可以不做修改，启动的时候，因为Dubbo会对IP和端口进行校验，如果您的日志级别为WARN及以下，您可能会收到WARN的日志，请忽略该日志。
- service-registry：服务的注册中心地址，支持服务的多注册中心，可以注入多个注册中心地址。每个注册中心均采用标准的Dubbo注册中心格式；多个用 `,` 分隔。示例中 `192.168.20.219` 为ZooKeeper地址，现场配置时请使用真实地址和端口。
- reference-registry：服务订阅的注册中心地址，支持多注册或者注册到未迁移前的注册中心。
- config-address：动态推送的地址。

iii. 其他修改。

对于非Spring Boot的Spring应用，需要将 `com.alibaba.edas.dubbo.migration.controller.EdasDubboRegistryRest` 添加到您的扫描路径中。

3. 本地验证。

此处以动态配置的方式为例。

i. 验证准备。验证前，请确保您已完成以下操作：

- 下载ZooKeeper
- 启动轻量级配置及注册中心
- 下载并启动Nacos

ii. 检查服务是否成功注册。

- 登录轻量配置中心，在服务提供者列表中查看对应的服务。
- 登录ZooKeeper，查看服务注册和消费信息。

iii. （可选）登录Nacos，配置服务注册信息。

> **说明** 如果不需要动态配置的话，无需执行此步骤。

参数	配置及操作
DataId	设置为*dubbo.registry.config*。
Group	设置对应Dubbo应用的名称applicationName，如*dubbo-migration-demo-server*。配置信息是应用维度，所以应用名不能重复。

配置内容	■ 应用级别
	`dubbo.reference.registry=edas://127.X.X.X:8080 ##`**注册服务的注册中心地址。**
	`dubbo.service.registry=edas://127.X.X.X:8080,zookeeper:127.X.X.X:2181 ##`**订阅服务的注册中心地址。**
	■ 实例IP级别
	`192.168.15.86.dubbo.reference.registry=edas://127.X.X.X:8080,zookeeper:127.X.X.X:2181`
	`192.168.15.86.dubbo.service.registry=edas://127.X.X.X:8080`
	集群验证时建议先验证实例IP级别，再验证整个应用验证。

iv. 查看迁移后应用调用是否正常，查看注册中心的注册订阅关系。

- Spring Boot 1.x版本：http://ip:port/dubboRegistry
- Spring Boot 2.x版本：http://ip:port/actuator/dubboRegistry

4. 将应用部署到SAE。

   根据实际需求将应用部署到SAE。具体操作，请参见部署应用概述。

5. 结果验证。

   i. 观察业务运行是否正常。

   ii. 查看服务订阅监控。

   如果应用开启了Spring Boot Actuator监控功能，请访问Actuator查看此应用订阅的各服务的RibbonServerList信息。Actuator地址如下：

   - Spring Boot 1.x版本：http://ip:port/dubboRegistry
   - Spring Boot 2.x版本：http://ip:port/actuator/dubboRegistry

   - `dubbo.orig.**` 表示应用中配置的注册中心信息。
   - `dubbo.orig.**` 表示生效的注册中心信息。

步骤二：迁移其他所有应用

依次将所有应用迁移到SAE。具体步骤，请参见步骤一：迁移第一个应用。

步骤三：清理迁移配置

迁移完成后，删除原有的注册中心配置和迁移过程专用的依赖 `edas-dubbo-migration-bom` 。

修改对应的注册中心地址（即删除ZooKeeper的配置），保证Consumer、Provider仅从SAE订阅。

- 方式一：动态配置。具体步骤，请参见本地验证。
- 方式二：手动修改。

  所有的应用修改完成后，修改应用的注册中心地址，将订阅的地址改为*SAE ConfigServer*。

  ```
  dubbo.registry.address = edas-migration://192.168.124.15:9999?service-registry=edas://127.X.X.X:8080,zookeeper://192.168.20.219:2181&reference-registry=edas://127.X.X.X:8080&config-address=127.X.X.X:8848
  ```

  `reference-registry` 的 `zookeeper://192.168.20.219:2181` 改为 `edas://127.X.X.X:8080` 。修改完成之后，即可部署应用。

  ```
  dubbo.registry.address = edas://127.X.X.X:8080
  ```

  > **说明** 当应用迁移完成之后，如果不再使用 `ZooKeeper` ，需要从注册中心配置中删除 `zookeeper://192.168.20.219:2181`

请在业务量较小的时间段分批重启应用。