企业级分布式应用服务 EDAS

应用开发

应用开发

使用 Spring Cloud 开发应用

Spring Cloud 概述

EDAS 支持原生 Spring Cloud 微服务框架，您在这个框架下开发的应用只需添加依赖和修改配置，即可获取 EDAS 企业级的应用托管、应用治理、监控报警和应用诊断等能力，实现代码零入侵。

Spring Cloud 提供了简化应用开发的一系列标准和规范。这些标准和规范包含了服务发现、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 还在这些规范的基础上，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的实现。

目前业界比较流行的 Spring Cloud 具体实现有 Spring Cloud Netflix、Spring Cloud Consul、Spring Cloud Gateway、Spring Cloud Sleuth 等，最近由阿里巴巴中间件开源的 Spring Cloud Alibaba 也是业界中受关注度很高的另一种实现。

如果您已经使用 Spring Cloud Netflix、Spring Cloud Consul 等 Spring Cloud 组件开发的应用，可以直接部署到 EDAS 正常运行并获得应用托管能力，同时还可以不修改任何一行代码直接使用 EDAS 所提供的高级监控功能，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 应用想使用 EDAS 中更多的服务治理相关的功能，那么您需要将您的 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

EDAS 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 EDAS 兼容性如下表所示：

Spring Cloud 功能		开源实现	EDAS 兼容性	说明
通用功能	服务注册与发现	- Netflix Eureka - Consul	兼容且提供替换组件	提供替换组件 Nacos。使用 Nacos，除了 Spring Cloud 服

		Discovery		务注册发现的标准功能外，还可以获得更多服务治理的功能。
	负载均衡	Netflix Ribbon	兼容	可以直接与 EDAS 的服务注册发现组件配合使用。
	服务调用	- Feign - RestTemplate	兼容	可以直接使用 EDAS 的服务发现、链路跟踪功能。
配置管理		- Config Server - Consul Config	兼容且提供替换组件	提供替换 Nacos。使用 Nacos，除了 Spring Cloud 服务注册发现的标准功能外，还可以从 EDAS 控制台管理配置，并获得实时动态刷新、推送轨迹查看等功能。
服务网关		- Spring Cloud Gateway - Netflix Zuul	兼容	可以直接使用 EDAS 的服务发现、配置管理、全链路跟踪功能。
链路跟踪		Spring Cloud Sleuth	兼容且提供替换组件	提供替换组件 ARMS。只需在 EDAS 控制台开启高级监控，无需修改任何代码和依赖，即可使用 ARMS。除全链路跟踪功能外，还可获得全息排查、线程剖析等功能。
消息驱动 Spring Cloud Stream		- Rabbit MQ binder - Kafka binder	兼容且提供替换组件	提供替换组件 RocketMQ binder，可以与其它实现同时使用
消息总线 Spring Cloud Bus		- Rabbit	兼容且提供替换组件	提供替换组件 RocketMQ

	MQ - Kafka		bus，可以与其它实现同时使用
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 EDAS 提供的商业化组件的版本配套关系如下表所示。

Spring Cloud	Spring Boot	Spring Cloud Alibaba	EDAS 商业化组件		
			Nacos Discovery	Nacos Config	SchedulerX
Greenwich	2.1.x	2.1.0.RELEASE	2.1.0.RELEASE	2.1.0.RELEASE	2.1.0.RELEASE
Finchley	2.0.x	2.0.0.RELEASE	2.0.0.RELEASE	2.0.0.RELEASE	2.0.0.RELEASE
Edgware	1.5.x	1.5.0.RELEASE	1.5.0.RELEASE	1.5.0.RELEASE	1.5.0.RELEASE

相关文档

如果您想将服务发现组件，如 Eureka、Consul 替换成 EDAS 所提供的 spring-cloud-alibaba-nacos-discovery，只需修改依赖和配置即可，无需修改任何代码。详情参考服务发现。

如何你想将配置管理组件，如 Spring Cloud Config 、Consul 替换成 EDAS 所提供的 spring-cloud-alibaba-nacos-config，只需修改依赖和配置即可，无需修改任何代码，详情参考配置管理。

如果你已经使用了服务网关，想使用 EDAS 提供的服务注册发现，配置管理，限流降级的功能，只需要引入相应的starter依赖并修改配置即可，详情参考服务网关。

快速开始

您可以在您的 Spring Cloud 应用中添加基本的依赖及配置，即可部署到 EDAS 中，并使用 EDAS 服务注册中心实现服务发现。详细步骤请参考 Spring Cloud 服务接入 EDAS。

实现负载均衡

Spring Cloud 的负载均衡是通过 Ribbon 组件完成的。 Ribbon 主要提供客户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都是通过 Ribbon 实现的。

Spring Cloud AliCloud Ans 集成了 Ribbon 的功能，AnsServerList 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比如 Nacos、Eureka、Consul、ZooKeeper 也都实现了对应的 ServerList 接口，比如 NacosServerList、 DomainExtractingServerList、ConsulServerList、ZookeeperServerList 等。

实现该接口相当于接入了 Spring Cloud 负载均衡规范，这个规范是共用的。这也意味着，从 Eureka、Consul、ZooKeeper 等服务发现方案切换到 Spring Cloud Alibaba 方案，在负载均衡这个层面，无需修改任何代码，RestTemplate、FeignClient，包括已过时的 AsyncRestTemplate ，都是如此。

下面介绍如何在您的应用中实现 RestTemplate 和 Feign 的负载均衡用法。

本地开发中主要描述开发中涉及的关键信息，如果您想了解完整的 Spring Cloud 程序，可下载 service-provider和 service-consumer。

操作步骤

RestTemplate 和 Feign 的实现方式有所不同，下面将分别介绍。

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。

要想在 Feign 上使用负载均衡，需要添加 Ribbon 的依赖。

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
<version>{version}</version>
</dependency>
```

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

使用 @EnableFeignClients 开启 Feign 功能。

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

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);
}
```

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。

实现配置管理

在开发 Spring Cloud 应用时，您可以使用 Nacos 在本地实现应用的配置管理。同时，由于 Nacos 是应用配置管理 ACM的开源版本，在将应用部署到 EDAS 后，即可通过 EDAS 集成的 ACM 对应用进行配置的管理和推送。

您可以按照文档从头开发该应用示例，使用 Spring Cloud Alibaba Nacos Config 实现配置管理。也可以直接下载该应用示例的 Demo nacos-config-example。

说明：Spring Cloud Alibaba Nacos Config 完成了 Nacos 与 Spring Cloud 框架的整合，支持 Spring Cloud 的配置注入规范。

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。

启动 Nacos Server。

解压下载的 Nacos Server 压缩包

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。

在本地 Nacos Server 控制台新建配置。

登录本地 Nacos Server 控制台（用户名和密码默认同为 nacos）。

在左侧导航栏中单击**配置列表**，在**配置列表**页面右上角单击新建配置图标 + 。

在**新建配置**页面填入以下信息，然后单击**发布**。

Data ID: nacos-config-example.properties

Group: DEFAULT_GROUP

配置内容: test.name=nacos-config-test

使用 Nacos Config 实现配置管理

创建一个 Maven 工程，命名为 nacos-config-example。

在pom.xml文件中添加依赖。

以 Spring Boot 2.1.4.RELEASE 和 Spring Cloud Greenwich.SR1 为例，依赖如下：

```
 <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>2.1.0.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 0.9.0.RELEASE。

- 如果使用 Spring Cloud Finchley 版本，对应 Spring Cloud Alibaba 版本为 0.2.2.RELEASE。

  如果使用 Spring Cloud Edgware 版本，对应 Spring Cloud Alibaba 版本为 0.1.2.RELEASE。

  说明：Spring Cloud Edgware 版本将在 2019 年 8 月结束生命周期，不推荐使用这个版本开发应用。

在src\main\java下创建 Packagecom.aliware.edas。

在 Packagecom.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);
}
}
```

在 Packagecom.aliware.edas中创建一个简单的 ControllerEchoController，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;
}
}
```

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
```

执行NacosConfigExampleApplication中的 main 函数，启动应用。

本地结果验证

在浏览器访问 http://127.0.0.1:18081 ，可以看到返回值为 nacos-config-test ，该值即为在本地 Nacos Server 中新建配置中的配置内容，即test.name的值。

部署到 EDAS

当在本地完成应用的开发和测试后，便可将应用程序打包并部署到 EDAS。您可以根据您的实际情况选择将 Spring Cloud 应用部署到 ECS 集群、容器服务 Kubernetes集群或 EDAS Serverless。部署应用的详细步骤请参见部署应用概述。

EDAS 集成的 ACM 即 Nacos 的商业化版本。当您将应用部署到 EDAS 的时候，EDAS 会通过优先级更高的方式去设置 Nacos 服务端地址和服务端口，以及 namespace、access-key、secret-key、context-path 信息。

在部署应用前，需要在 EDAS 控制台的配置管理中新建和本地 Nacos Server 中相同的配置，具体操作步骤如下：

登录 EDAS 控制台。

在左侧导航栏中选择**应用管理** > **配置管理**。

在**配置管理**页面选择**地域**和**命名空间**，然后在页面右侧单击新建配置图标 ＋。

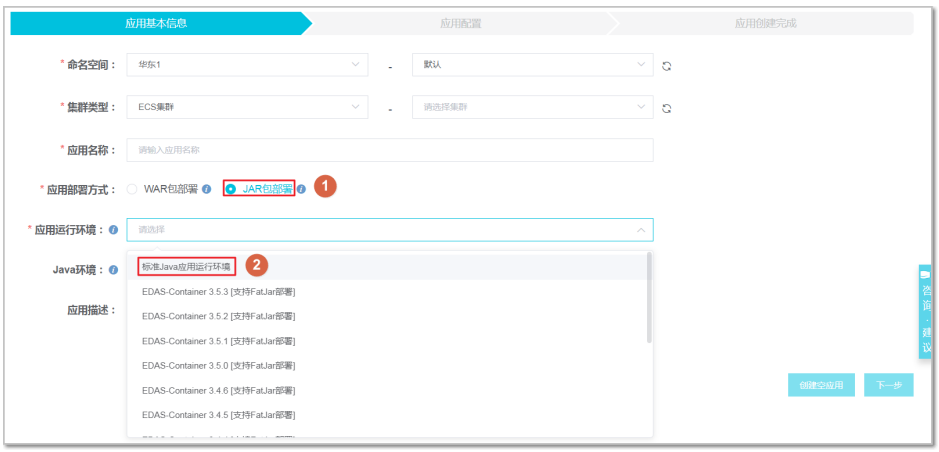
在**新建配置**页面设置 **Data ID**、**Group** 和 **配置内容**，然后单击**发布**。

Data ID: nacos-config-example.properties

Group: DEFAULT_GROUP

配置内容: test.name=nacos-config-test

第一次部署建议是通过控制台部署，且如果使用 JAR 包部署，在创建应用时**应用运行环境** 务必选择**标准 Java 应用运行环境**。



结果验证

部署完成后，可以通过查看日志确认应用是否启动成功。

执行命令curl http://<应用实例 IP>:<服务端口>，如curl http://192.168.0.34:8080 查看是否返回配置内容 *nacos-config-test*。

在 EDAS 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>，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
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	
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，支持 yaml 和 yml。
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拿到服务端地址。
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。

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 从零搭建应用的服务网关。

- 服务网关为什么使用 EDAS 注册中心
- 本地开发 准备工作基于 Spring Cloud Gateway 搭建服务网关 创建服务网关创建服务提供者结果验证基于 Zuul 搭建服务网关 创建服务网关创建服务提供者结果验证
- FAQ

服务网关为什么使用 EDAS 注册中心

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

商业版的 EDAS 服务注册中心，与开源版本的 Nacos、Eureka 和 Consul 相比，还具有以下优势：

共享组件，节省了你部署运维 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。

- EDAS 服务注册中心 与 EDAS 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、平滑上下线、灰度发布等。

本地开发

本地开发中主要描述开发中涉及的关键信息，如果您想了解完整的 Spring Cloud 程序，可下载 spring-cloud-gateway-nacos、spring-cloud-zuul-nacos 和 nacos-service-provider。

准备工作

下载 Maven 并设置环境变量。(已经操作的可略过)

请您通过 **下载地址** 下载最新版本的 Nacos Server。(已经操作的可以略过)

启动 Nacos Server

- 解压下载的 Nacos Server 压缩包，并切换到 nacos/bin 目录。
- Linux/Unix/Mac 类系统执行如下命令 sh startup.sh -m standalone，Windows 系统则 cmd startup.cmd或者 双击 startup.cmd 运行文件。

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。

创建服务网关

创建一个 Maven 工程，命名为spring-cloud-gateway-nacos。

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>2.1.0.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>
```

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);
}
}
```

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: 15012

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
```

执行启动类GatewayApplication中的 main 函数，启动 Gateway。

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。

创建服务提供者

创建一个服务提供者的应用。详情请参考快速开始。

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;
}
}
}
```

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://127.0.0.1:18012/provider1/echo/123456
123456%
```

在 EDAS 中验证。

您可以参考快速开始中的将应用部署到 EDAS 部分，将您的应用部署到 EDAS，并验证。

EDAS 服务注册中心提供了商业化版本 Nacos Server。当您将应用部署到 EDAS 的时候，EDAS 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、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。

创建服务网关

创建一个 Maven 工程，命名为spring-cloud-zuul-nacos。

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>2.1.0.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>
```

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);
}
}
```

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
```

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

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。

创建服务提供者

如何快速创建一个服务提供者可参考快速开始。

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;
}
}
}
```

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 EDAS 中验证。

您可以参考快速开始中的将应用部署到 EDAS 部分，将您的应用部署到 EDAS，并验证。

EDAS 服务注册中心提供了商业化版本 Nacos Server。当您将应用部署到 EDAS 的时候，EDAS 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、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。

FAQ

使用其他版本

示例中使用的 Spring Cloud 版本为 Greenwich，对应的 Spring Cloud Alibaba 版本为 2.1.0.RELEASE。Spring Cloud Finchley 对应的 Spring Cloud Alibaba 版本为 2.0.0.RELEASE，Spring Cloud Edgware 对应的 Spring Cloud Alibaba 版本为 1.5.0.RELEASE。

说明：Spring Cloud Edgware 版本的生命周期即将在 2019 年 8 月结束，不推荐使用这个版本开发应用。

从 ANS 迁移

EDAS 注册中心在服务端对 ANS 和 Nacos 的数据结构做了兼容，在同一个命名空间下，且 Nacos 未设置 group 时，Nacos 和 ANS 客户端可以互相发现 对方注册的服务。

实现对象存储

本文档通过一个示例向您介绍如何在本地 Spring Cloud 应用中实现对象存储，并将该应用托管到 EDAS 中。

为什么使用 OSS

OSS 是阿里云提供的海量、安全、低成本、高可靠的云存储服务。具有与平台无关的 RESTful API 接口，您可以在 Spring Cloud 开发的应用中存储和访问任意类型的数据。

准备工作

在应用中实现对象存储功能前，您需要先使用您的阿里云账号在 OSS 创建存储空间。

开通 OSS 服务。

创建存储空间。

在本地实现对象存储

创建一个 Maven 工程，命名为oss-example。

以 *Spring Boot 2.0.6.RELEASE* 和 *Spring Cloud Finchley.SR1* 为例，在pom.xml文件中添加如下依赖。

```
 <parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.0.6.RELEASE</version>
<relativePath/>
</parent>

<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-alicloud-oss</artifactId>
<version>0.2.1.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>Finchley.SR1</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

说明：

- 如果您需要选择使用 *Spring Boot 1.x* 的版本，请使用 *Spring Boot 1.5.x* 和 *Spring Cloud Edgware* 版本，对应的 **Spring Cloud Alibaba** 版本为 *0.1.1.RELEASE*。
- *Spring Boot 1.x* 版本的生命周期即将在 **2019 年 8 月** 结束，推荐使用 Spring Boot 新版本开发您的应用。

在src/main/java下创建一个 package，如spring.cloud.alicloud.oss。

在 packagespring.cloud.alicloud.oss下创建oss-example的启动类OssApplication。

```
package spring.cloud.alicloud.oss;

import com.aliyun.oss.OSS;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import java.net.URISyntaxException;

@SpringBootApplication
public class OssApplication {
public static final String BUCKET_NAME = "test-bucket";
public static void main(String[] args) throws URISyntaxException {
SpringApplication.run(OssApplication.class, args);
}
@Bean
public AppRunner appRunner() {
return new AppRunner();
}
class AppRunner implements ApplicationRunner {
@Autowired
private OSS ossClient;
@Override
public void run(ApplicationArguments args) throws Exception {
try {
if (!ossClient.doesBucketExist(BUCKET_NAME)) {
ossClient.createBucket(BUCKET_NAME);
}
} catch (Exception e) {
System.err.println("oss handle bucket error: " + e.getMessage());
System.exit(-1);
}
}
}
}
```

在src/main/resources路径下再添加一个用于上传的示例文件oss-test.json。

```
{
"name": "oss-test"
}
```

在 packagespring.cloud.alicloud.oss下创建类OssController并添加配置，包含上传、下载，以及使用 Spring 的 Resouce 规范获取文件的功能。

```
 package spring.cloud.alicloud.oss;

import com.aliyun.oss.OSS;
import com.aliyun.oss.common.utils.IOUtils;
import com.aliyun.oss.model.OSSObject;
import org.apache.commons.codec.CharEncoding;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.io.Resource;
import org.springframework.util.StreamUtils;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.nio.charset.Charset;

@RestController
public class OssController {
@Autowired
private OSS ossClient;
@Value("oss://" + OssApplication.BUCKET_NAME + "/oss-test.json")
private Resource file;
@GetMapping("/upload")
public String upload() {
try {
ossClient.putObject(OssApplication.BUCKET_NAME, "oss-test.json", this
.getClass().getClassLoader().getResourceAsStream("oss-test.json"));
} catch (Exception e) {
e.printStackTrace();
return "upload fail: " + e.getMessage();
}
return "upload success";
}
@GetMapping("/file-resource")
public String fileResource() {
try {
return "get file resource success. content: " + StreamUtils.copyToString(
file.getInputStream(), Charset.forName(CharEncoding.UTF_8));
} catch (Exception e) {
e.printStackTrace();
return "get resource fail: " + e.getMessage();
}
}
@GetMapping("/download")
public String download() {
try {
OSSObject ossObject = ossClient.getObject(OssApplication.BUCKET_NAME, "oss-test.json");
return "download success, content: " + IOUtils
.readStreamAsString(ossObject.getObjectContent(), CharEncoding.UTF_8);
} catch (Exception e) {
e.printStackTrace();
return "download fail: " + e.getMessage();
}
}
}
```

获取 Access Key ID、Access Key Secret 和 Endpoint，并在本地添加配置。

登录**安全信息管理**页面，获取 Access Key ID 和 Access Key Secret。

参考**访问域名和数据中心**，按创建存储空间的地域获取 Endpoint。

在src/main/resources路径下创建application.properties文件，并添加 Access Key ID、Access Key Secret 和 Endpoint 配置。

```
spring.application.name=oss-example
server.port=18084
# 填写 Access Key ID
spring.cloud.alicloud.access-key=xxxxx
# 填写 Access Key Secret
spring.cloud.alicloud.secret-key=xxxxx
# 填写 Endpoint
spring.cloud.alicloud.oss.endpoint=xxx.aliyuncs.com
management.endpoints.web.exposure.include=*
```

执行OssApplication中的 main 函数，启动服务。

结果验证

在浏览器中访问 http://127.0.0.1:18084/upload。

如果提示upload success，则说明示例文件oss-test.json上传成功。否则，请检查本地代码，排查问题，然后再次执行OssApplication中的 main 函数，启动服务。

登录**OSS控制台**，进入您创建并上传文件的 Bucket，然后在顶部单击**文件管理**，查看是否有示例文件。

如果看到oss-test.json，则说明上传成功。否则，请检查本地代码，排查问题，然后再次执行OssApplication中的 main 函数，启动服务。

在浏览器访问 http://127.0.0.1:18084/download 即可下载文件，会得到 oss-test.json 的文件内容。

```
{
"name": "oss-test"
}
```

在浏览器访问 http://127.0.0.1:18084/file-resource 即可获得示例文件oss-test.json的内容。

```
{
"name": "oss-test"
}
```

部署应用到 EDAS

Spring Cloud AliCloud OSS 在设计之初就考虑到了从开发环境迁移到 EDAS 的场景，您可以直接将应用部署到 EDAS 中，无需修改任何代码和配置。部署方式和详细步骤请参考应用部署概述。

实现任务调度

EDAS 将分布式任务调度 SchedulerX 作为组件集成到控制台中，实现任务调度。本文将介绍如何在您的 Spring Cloud 应用中使用 SchedulerX 实现任务调度，并部署到 EDAS 中，实现一个*简单 Job 单机版* 的任务调度功能。

为什么使用 SchedulerX

SchedulerX 是阿里巴巴的一款分布式任务调度产品。它为您提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务，同时提供分布式的任务执行模型，如网格任务。

在本地实现任务调度

创建一个 Maven 工程，命名为scx-example。

以 *Spring Boot 2.0.6.RELEASE* 和 *Spring Cloud Finchley.SR1* 为例，在pom.xml文件中添加如下依赖。

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.0.6.RELEASE</version>
<relativePath/>
</parent>

<dependencies>
<dependency>
```

```
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-alicloud-schedulerx</artifactId>
<version>0.2.1.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>Finchley.SR1</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

说明：

- 如果您需要选择使用 *Spring Boot 1.x* 的版本，请使用 *Spring Boot 1.5.x* 和 *Spring Cloud Edgware* 版本，对应的 **Spring Cloud Alibaba** 版本为 *0.1.1.RELEASE*。
- *Spring Boot 1.x* 版本的生命周期即将在 **2019 年 8 月** 结束，推荐使用 Spring Boot 新版本开发您的应用。

创建scx-example的启动类ScxApplication。

```
 import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

创建一个简单的类TestService，通过 Spring 向测试任务中的 IOC 进行注入。

```
 import org.springframework.stereotype.Service;

@Service
public class TestService {

public void test() {
System.out.println("---------IOC Success--------");
```

```
}
}
```

创建一个简单的SimpleTask作为测试任务类，并在其中注入TestService。

```
 import com.alibaba.edas.schedulerx.ProcessResult;
import com.alibaba.edas.schedulerx.ScxSimpleJobContext;
import com.alibaba.edas.schedulerx.ScxSimpleJobProcessor;
import org.springframework.beans.factory.annotation.Autowired;

public class SimpleTask implements ScxSimpleJobProcessor {

@Autowired
private TestService testService;

@Override
public ProcessResult process(ScxSimpleJobContext context) {
System.out.println("-----------Hello world----------");
testService.test();
ProcessResult processResult = new ProcessResult(true);
return processResult;
}

}
```

创建调度任务并添加配置。

登录 EDAS 控制台，在**测试**地域中创建调度任务分组并记录分组 ID。

在创建的任务分组中按如下参数创建调度任务。

i. **Job 分组**：选择在*测试*地域下刚创建的任务分组的*分组 ID*。
ii. **Job 处理接口**：Job 处理接口实现类的全类名，本文档中为 *SimpleTask*，和应用中的测试任务类保持一致。
iii. **类型**：*简单 Job 单机版*
iv. **定时表达式**：*默认选项*、*0 * * * * ?*。表示任务每分钟会被执行一次。
v. **Job 描述**：无
vi. **自定义参数**：无

在本地 Maven 工程的src/main/resources路径下创建文件application.properties，在中添加如下配置。

```
server.port=18033
# 配置任务的地域（测试地域对应的 **regionName** 为 *cn-test*）和分组ID（group-id）
spring.cloud.alicloud.scx.group-id=***
spring.cloud.alicloud.edas.namespace=cn-test
```

执行ScxApplication中的 main 函数，启动服务。

结果验证

在 IDEA 的 Console 中观察标准输出，可以看到会定时的打印出如下的测试信息。

```
----------Hello world----------
---------IOC Success--------
```

部署到 EDAS

Spring Cloud AliCloud SchedulerX 在设计之初就考虑到了从开发环境迁移到 EDAS 的场景，您可以直接将应用部署到 EDAS 中，无需修改任何代码和配置。部署方式和详细步骤请参考应用部署概述。

在部署完成之后，即可使用 EDAS 控制台进行任务调度。

在非测试地域使用调度任务的附加步骤

本文档以在测试地域中使用为例，测试环境为公网环境，您在本地或云端都可以进行验证，且没有权限的限制。如果您要部署到其它地域（如*杭州*）中，还需要在创建调度任务并调度的步骤中完成以下操作：

登录 EDAS 控制台，在*杭州*地域创建任务分组和调度任务。

登录安全信息管理页面，获取 Access Key ID 和 Access Key Secret。

在application.properties文件中配置调度任务。

除了*简单 Job 单机版*，您还可以配置其它类型的调度任务。详情请参考分布式任务调度 SchedulerX 简介。

在application.properties文件中添加您阿里云账号的 Access Key ID 和 Access Key Secret 的配置。

```
spring.cloud.alicloud.access-key=xxxxx
spring.cloud.alicloud.secret-key=xxxxx
```

后续操作

您的应用部署到 EDAS 之后，就可以使用 SchedulerX 组件实现更多任务调度的能力。详情请参考分布式任务调度 SchedulerX 简介。

实现限流降级

目前原生 Spring Cloud 应用已经支持新版本的限流降级功能。限流降级所使用的基础框架是开源项目 Sentinel。

Sentinel 目前支持 WebServlet，RestTemplate, Feign。Zuul 和 Spring Cloud Gateway 需要 Spring Cloud Alibaba 发布新版本后才支持，或者您可以参考 网关限流 自行进行一些 Bean 的设置。

注意：Sentinel 1.6 版本才开始支持网关限流。

本地开发中主要描述开发中涉及的关键信息，如果您想了解完整的 Spring Cloud 程序，可下载 sentinel-flow-example 以及 sentinel-degrade-example。

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 Sentinel Dashboard。

- yourPort为您本地 Sentinel Dashboard 设置的端口，如 *8081*。
- sentinel-dashboard-version.jar为您本地下载的 Sentinel Dashboard 的 JAR 包，如 *sentinel-dashboard-1.6.0.jar*。

在本地执行以下命令，启动 Sentinel Dashboard。

java -jar -Dserver.port=yourPort sentinel-dashboard-version.jar

- 登录本地 Sentinel Dashboard 控制台（用户名和密码同为 sentinel），新建限流规则。

本地开发

使用 Sentinel 实现限流

操作步骤

创建一个 Maven 工程，命名为 sentinel-flow-example。

在pom.xml文件中添加依赖。

以 Spring Boot 2.1.4.RELEASE 和 Spring Cloud Greenwich.SR1 为例，依赖如下：

```
 <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>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
<version>0.9.0.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 0.9.0.RELEASE。

如果使用 Spring Cloud Finchley 版本，对应 Spring Cloud Alibaba 版本为 0.2.2.RELEASE。

如果使用 Spring Cloud Edgware 版本，对应 Spring Cloud Alibaba 版本为 0.1.2.RELEASE。

说明：Spring Cloud Edgware 版本的生命周期即将在 2019 年 8 月结束，不推荐使用这个版本开发应用。

在 src\main\java 下创建 Package com.aliware.edas。

在 Package com.aliware.edas中创建 sentinel-flow-example 的启动类 SentinelFlowExampleApplication。

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

在 Packagecom.aliware.edas中创建一个简单的 ControllerSentinelFlowController，并暴露一些方法。

```
@RestController
public class SentinelFlowController {

@GetMapping("/flow")
public String flow() {
return "success";
}

@GetMapping("/save")
@SentinelResource(value = "test", blockHandler = "block")
public String save() {
return "test";
}

public String block(BlockException e) {
return "block";
}

}
```

说明：@SentinelResource 用于定义资源，并提供可选的异常处理和 fallback 配置项。详情请参见**注解支持**。

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 Sentinel Dashboard 的地址。

其中 **127.0.0.1:8081** 为 Sentinel Dashboard 的地址，如果您的 Sentinel Dashboard 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，可以参考**配置项参考**在application.properties文件中增加配置。

```
spring.application.name=sentinel-flow-example
server.port=18888

spring.cloud.sentinel.eager=true
spring.cloud.sentinel.transport.dashboard=127.0.0.1:8081
```

执行SentinelFlowExampleApplication中的 main 函数，启动应用。

新建流控规则

在本地 Sentinel Dashboard 中对 sentinel-flow-example 应用新建流控规则。

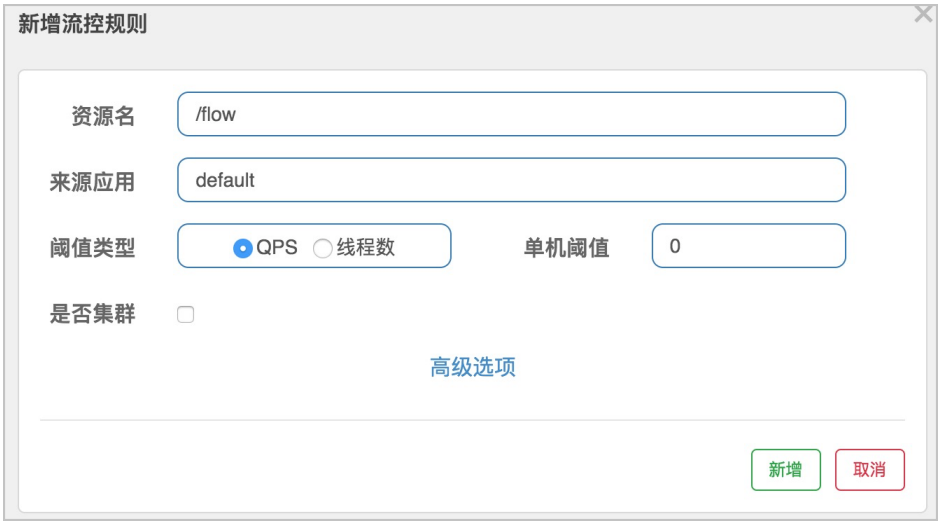
访问 http://127.0.0.1:8081, 进入 Sentinel 控制台。

会看到运行的 sentinel-flow-example 应用。

在左侧导航栏单击该应用右侧的下拉箭头，然后在菜单中单击**流控规则**。

在**流控规则**页面添加两个流控规则。

资源名为/flow，来源应用会 “default” ， 阈值类型是 QPS，单机阈值为 0。



资源名为test，来源应用会 “default” ， 阈值类型是 QPS，单机阈值为 0。

结果验证

在浏览器访问 http://localhost:18888/flow ，返回 Blocked by Sentinel (flow limiting)，则表明该应用被成功限流。

在浏览器访问 http://localhost:18888/save ，返回 block ，则表明该应用被成功限流。

使用 Sentinel 实现降级

操作步骤

创建一个 Maven 工程，命名为sentinel-degrade-example。

在pom.xml文件中添加依赖。

以 Spring Boot 2.1.4.RELEASE 和 Spring Cloud Greenwich.SR1 为例，依赖如下：

```
 <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>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
<version>0.9.0.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 0.9.0.RELEASE。

如果使用 Spring Cloud Finchley 版本，对应 Spring Cloud Alibaba 版本为 0.2.2.RELEASE。

如果使用 Spring Cloud Edgware 版本，对应 Spring Cloud Alibaba 版本为 0.1.2.RELEASE。

说明：Spring Cloud Edgware 版本的生命周期即将在 2019 年 8 月结束，不推荐使用这个版本开发应用。

在 src\main\java 下创建 Package com.aliware.edas。

在 Package com.aliware.edas中创建 sentinel-degrade-example 的启动类 SentinelDegradeExampleApplication。

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

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 Sentinel Dashboard 的地址。

其中 **127.0.0.1:8081** 为 Sentinel Dashboard 的地址，如果您的 Sentinel Dashboard 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，可以参考**配置项参考**在application.properties文件中增加配置。

```
spring.application.name=sentinel-degrade-example
server.port=19999

spring.cloud.sentinel.eager=true
spring.cloud.sentinel.transport.dashboard=127.0.0.1:8081
```

执行SentinelDrgradeExampleApplication中的 main 函数，启动应用。

新建降级规则

在本地 Sentinel Dashboard 中对sentinel-degrade-example应用新建降级规则。

访问http://127.0.0.1:8081，进入 Sentinel 控制台。

会看到运行的sentinel-degrade-example应用。

在左侧导航栏单击该应用右侧的下拉箭头，然后在菜单中单击**降级规则**。

在**降级规则**页面添加两个流控规则。

资源名为/rt，降级策略是 *RT*，，RT 阈值为 *200(毫秒)*，时间窗口为 *30（秒）*。

资源名为/exception，降级策略是 *异常比例*， 异常比列阈值*0.5*，阈值为 *0.5*，时间窗口为 *30（秒）*。

结果验证

RT 验证

执行脚本 rt.sh

```
#!/usr/bin/env bash
n=1
while [ $n -le 10 ]
do
echo `curl -s http://localhost:19999/rt`
let n++
done
```

执行 5 次后，接口被降级(降级至少要发生 5 次):

```
$ sh rt.sh
rt success
rt success
rt success
rt success
rt success
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
```

异常比例验证

执行脚本 exception.sh

```
#!/usr/bin/env bash
n=1
while [ $n -le 10 ]
do
echo `curl -s http://localhost:19999/exception`
let n++
done
```

执行 5 次后，接口被降级(降级至少要发生 5 次):

```
$ sh exception.sh
{"timestamp":"2019-05-17T06:21:22.585+0000","status":500,"error":"Internal Server Error","message":"custom exception","path":"/exception"}
{"timestamp":"2019-05-17T06:21:22.712+0000","status":500,"error":"Internal Server Error","message":"custom exception","path":"/exception"}
{"timestamp":"2019-05-17T06:21:22.744+0000","status":500,"error":"Internal Server Error","message":"custom exception","path":"/exception"}
{"timestamp":"2019-05-17T06:21:22.766+0000","status":500,"error":"Internal Server Error","message":"custom exception","path":"/exception"}
{"timestamp":"2019-05-17T06:21:22.803+0000","status":500,"error":"Internal Server Error","message":"custom exception","path":"/exception"}
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
Blocked by Sentinel (flow limiting)
```

关于 RestTemplate 和 Feign 的限流降级

Spring Cloud Alibaba Sentinel Starter 提供了对 RestTemplate 的支持(构造 RestTemplate 的时候需要加上 @SentinelRestTemplate注解)：

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

具体内容请参见 Sentinel 支持 RestTemplate。

Feign 的内容请参见 Sentinel 支持 Feign。

将应用部署到 EDAS

当在本地完成应用的开发和测试后，便可将应用程序打包并部署到 EDAS。您可以根据您的实际情况选择将 Spring Cloud 应用可以部署到 ECS 集群、容器服务 Kubernetes集群或 EDAS Serverless。部署应用的详细步骤请参见部署应用概述。

EDAS 配置管理中心提供了商业化版本 Sentinel Dashboard Server。您只需添加如下依赖即可：

```
<dependency>
<groupId>com.alibaba.csp</groupId>
<artifactId>spring-boot-starter-ahas-sentinel-client</artifactId>
<version>1.2.1</version>
</dependency>
```

注意：如果要部署到 EDAS 上spring-boot-starter-ahas-sentinel-client 依赖必须要引入，否则在 EDAS 控制台上对规则的操作无法真正生效。

为应用 *sentinel-flow-example* 配置限流规则

登录 EDAS 控制台。

在左侧导航栏中选择**应用管理** > **应用列表**。

在**应用列表**页面选择部署应用的地域和命名空间，然后单击部署的应用 *sentinel-flow-example*。

在应用详情页左侧导航栏选择**限流降级** > **流控规则**。

在**流控规则**页面右上角单击**新建流控规则**。

在新建流控规则页面设置流控规则参数，然后单击**新建**。

/flow 流控规则

test 规则

为应用 *sentinel-degrade-example* 配置降级规则

登录 EDAS 控制台。

在左侧导航栏中选择**应用管理** > **应用列表**。

在**应用列表**页面选择部署应用的地域和命名空间，然后单击部署的应用 *sentinel-flow-example*。

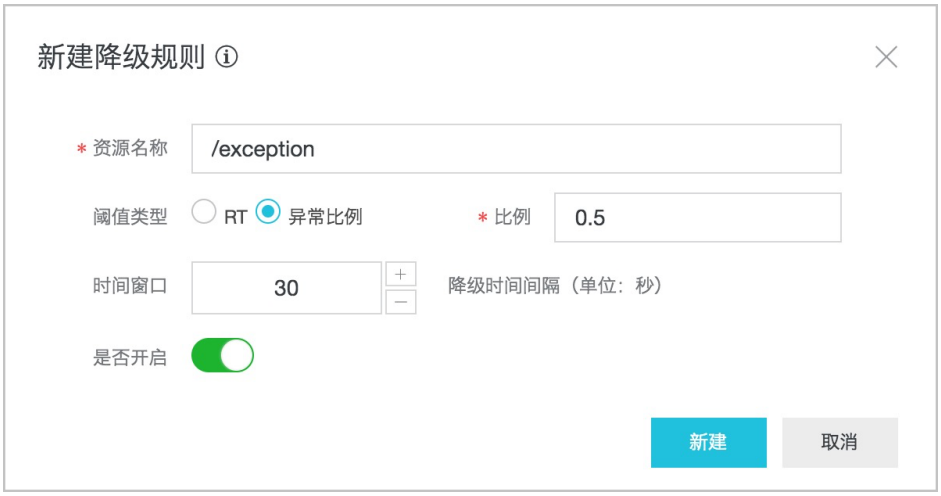
在应用详情页左侧导航栏选择**限流降级** > **降级规则**。

在**降级规则**页面右上角单击**新建降级规则**。

在**新建降级规则**页面设置降级规则参数，然后单击**新建**。

rt 降级规则

/exception 规则



结果验证

部署完成后，查看日志，确认应用是否启动成功。详情请参见查看实例日志。

查看 *sentinel-flow-example* 流控结果。

执行命令curl http://<应用实例 IP>:<服务端口>/flow，如curl http://192.168.0.34:9999/flow 查看是否返回Blocked by Sentinel (flow limiting)

执行命令curl http://<应用实例 IP>:<服务端口>/save，如curl http://192.168.0.34:9999/save查看是否返回block

说明：实际的服务端口可以在应用详情页的**基本信息**页面查询。

查看 *sentinel-degrade-example* 降级结果。

执行脚本rt.sh。

假设 IP 是 192.168.0.34， 服务端口是 9999。

说明：实际的服务端口可以在应用详情页的**基本信息**页面查询。

```
#!/usr/bin/env bash
n=1
while [ $n -le 10 ]
do
echo `curl -s http://192.168.0.34:9999/rt`
```

```
let n++
done
```

看执行 5 次后，接口是否被降级（降级至少要发生 5 次）。

执行脚本exception.sh。

假设 ip 是 192.168.0.34， 服务端口是 9999。

说明：实际的服务端口可以在应用详情页的**基本信息**页面查询。

```
#!/usr/bin/env bash
n=1
while [ $n -le 10 ]
do
echo `curl -s http://192.168.0.34:9999/exception`
let n++
done
```

看执行 5 次后，接口是否被降级（降级至少要发生 5 次）。

将 Spring Cloud 集群（多应用）平滑迁移到 EDAS

如果您的 Spring Cloud 集群（包含多个应用）已经部署在阿里云上，那么本文档将向您介绍如何将集群及集群中的所有应用平滑迁移到 EDAS 中，并实现基本的服务注册与发现。如果您的 Spring Cloud 集群还未部署到阿里云，请提交工单或联系 EDAS 技术支持人员为您提供完整的上云及迁移到 EDAS 方案。

迁移到 EDAS 的价值

EDAS 为应用部署提供了启动参数灵活配置、流程可视化、服务优雅上下线和分批发布等功能，让您的应用发布可配、可查、可控。

EDAS 提供了服务发现与配置管理功能，您无需再自行运维 Eureka、ZooKeeper、Consul 等中间件组件，可以直接使用 EDAS 提供的商业版服务发现与配置管理。

EDAS 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。

EDAS 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。

EDAS 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。

EDAS 提供限流降级功能，保证您的应用高可用。

EDAS 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到生产环境并处于正常运行状态中，此时想将集群迁移到 EDAS 享受完整的 EDAS 功能，那么在迁移过程中，保证业务的平稳运行不中断是第一要务，而保证应用平台运行不中断迁移到 EDAS 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，则没必要按照本文进行平滑迁移，可直接将应用在本地开发完再部署到 EDAS，详情请参见 将 Spring Cloud 应用托管到 EDAS 。

迁移流程

下图是一个比较典型的应用架构，根据迁移的先后顺序需要将迁移流程分为三步。

（**必选**）迁移应用

迁移的应用一般都是无状态的，所以这一步是可以最先进行的。同时，本文将重点介绍如何迁移应用

。

（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，在应用迁移后，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，详情请参见SLB 绑定概述。

如果您的应用在迁移之前没有使用 SLB，建议您在迁移完入口应用（如上图所示的 API Gateway）后，为该应用创建并绑定一个新的 SLB。

迁移应用的方案中，我们推荐您使用双注册和双订阅方案，以节约您的 ECS 成本。但如果由于某种原因（如原 ECS 端口被占用）不能复用之前的 ECS，则需要采用切流迁移方案，您需要添加新的 ECS用于迁移应用。在应用迁移完成后，参考上面描述的 SLB 的状态，选择复用 SLB 或创建 SLB 并绑定到应用。

域名

如果迁移后的应用可以复用 SLB，域名配置也无需修改。

如果迁移后的应用需要创建新的 SLB 并绑定到应用，则需要在域名中添加新的 SLB 配置，详情请参见域名DNS修改，并删除原来不再使用的 SLB。

（可选）迁移存储和消息队列

- 如果你之前的应用已经部署在阿里云上，则存储和消息队列也使用了阿里云相关产品（如 RDS、MQ 等），则应用迁移完成后，之前的存储和消息队列无需迁移。
- 如果您之前的应用不在阿里云上，请提交工单或联系 EDAS 技术支持人员为您提供完整的上云及迁移到 EDAS 方案。

本文将主要介绍如何迁移应用。如果您想通过一个 Demo 快速体验平滑迁移的过程，可下载 Demo，参考 Readme 运行一个迁移的样例。

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。这两种方案都可以保证您的应用正常运行不中断的完成迁移。

说明：本文将主要介绍如果使用双注册和双订阅方案迁移应用。

切流迁移方案

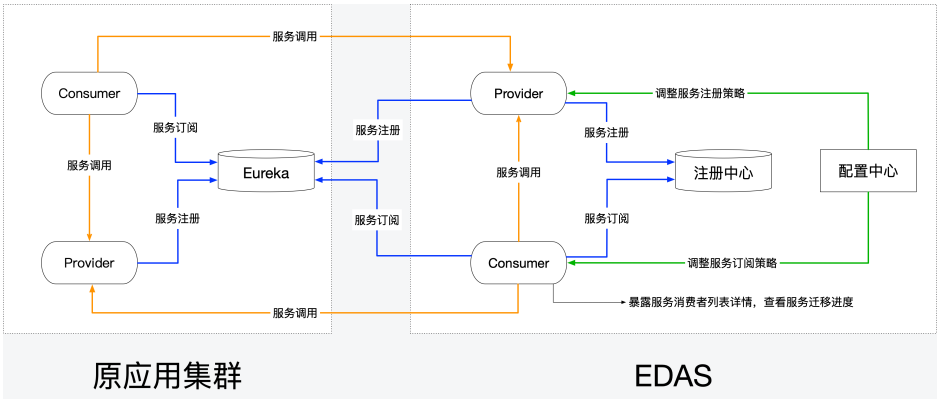
使用 Spring Cloud Alibaba 将原有的服务注册中心切换到 Nacos。开发一套新的应用部署到 EDAS，最后通过 SLB 和域名配置来进行切流。

如果您选择此方案，那您可以参考将 Spring Cloud 应用托管到 EDAS开发应用，不需要再阅读迁移应用的后续内容，只需要关注本文末尾的迁移风险点和技术支持。

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 EDAS 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。

通过双注册和双订阅平滑迁移应用的架构图如下：



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，只需要添加依赖，并修改一行代码，就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

步骤一：选择最先迁移的应用

建议是从最下层 Provider 开始迁移。但如果调用链路太复杂，比较难分析，也可以任意选一个应用进行迁移。选择完成后，即可参考下面的迁移步骤迁移第一个应用。

步骤二：在应用程序中添加依赖并修改配置

为了能将您原来的应用托管到 EDAS 中，您需要在您的应用程序中添加相关依赖并修改配置。

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>
```

在 application.properties 中添加 nacos-server 的地址。

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

默认情况下 Spring Cloud 只支持在依赖中引入一个注册中心，当存在多个注册中心时，启动会报错。所以这里需要添加一个依赖 edas-sc-migration-starter，使 Spring Cloud 应用支持多注册。

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

Ribbon 是实现负载均衡的组件，为了使您的应用可以支持从多个注册中心订阅服务，你需要修改 Ribbon 配置。在应用启动的主类中，将 RibbonClients 默认配置为 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);
}
}
```

那么修改后的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：您在本地修改应用或者应用部署到 EDAS 后，如果对应用有其它控制需求（如注册到哪些注册中心或从哪些注册中心订阅），可以通过 Spring Cloud Config 或 Nacos Config 进行动态的配置调整，无需重启应用，调整配置请参考动态调整服务注册和订阅方式。

不过，这两种方式都需要在应用中添加配置管理依赖和修改配置的操作，使用 Spring Cloud Config 请参考开源文档，使用 Nacos Config 请参考实现配置管理。

步骤三：将修改后的应用部署到 EDAS 中

您可以根据您的实际需求将应用部署到 ECS 集群或容器服务 Kubernetes 集群中，在部署时也可以选择通过控制台、工具等方式进行部属。详情请参见部署应用概述。

为了帮助您节约成本，建议您继续使用之前 ECS，但需要将 ECS 导入到 EDAS 中，详情请参见导入 ECS 。**在导入 ECS 的时候如果提示需要转化后导入，请对重要的数据做好备份。**

如果需要创建新的 ECS、集群等资源，请确保在原有 VPC 内创建，以保证迁移前后的应用网络互通，顺利完成迁移。详情请参见创建资源。

在数据库、缓存、消息队列等产品中为新 ECS 配置 IP 白名单等，确保应用依赖的这些第三方组件可以正常访问。

结果验证

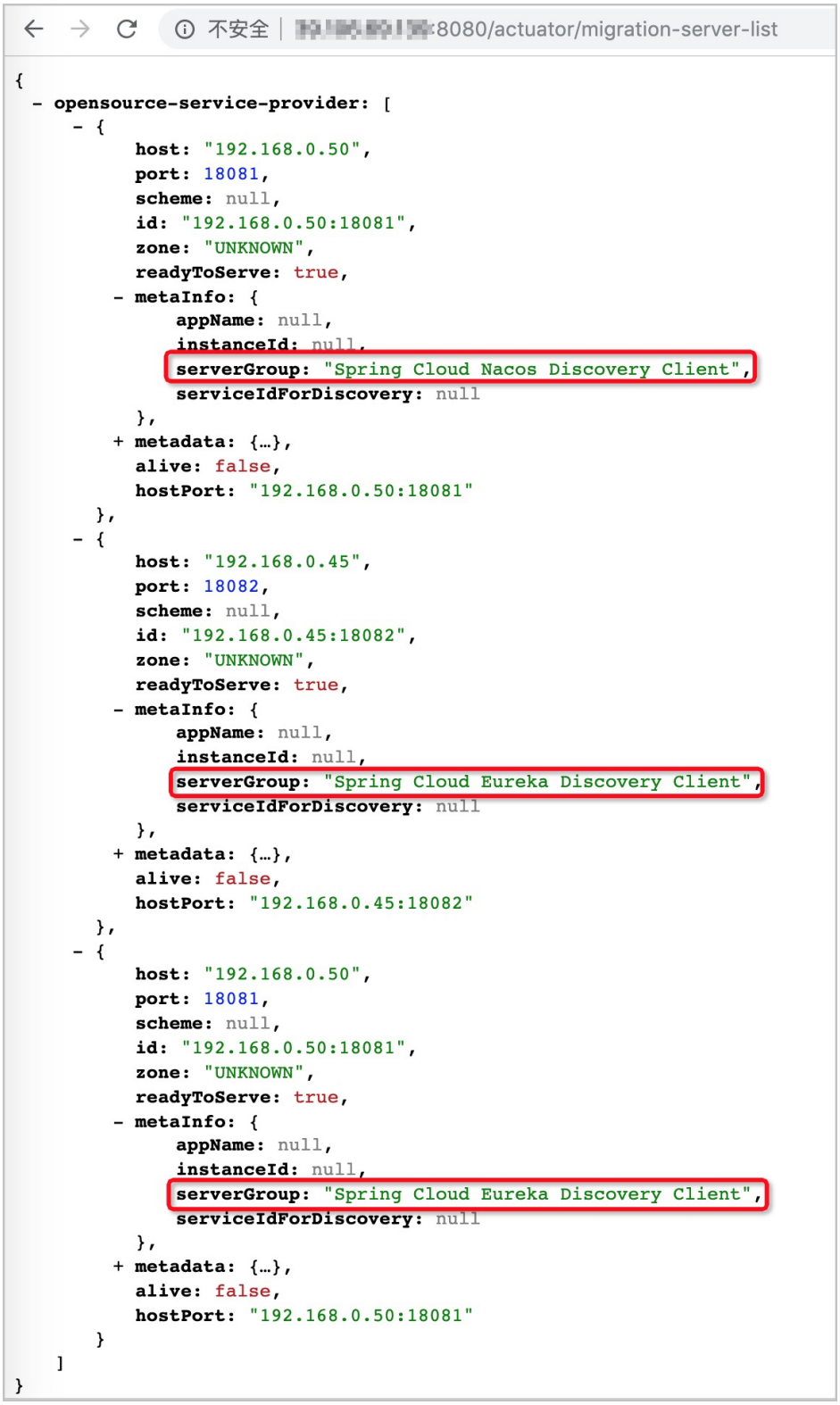
最重要的是观察业务本身是否正常。

查看服务订阅监控。

如果您的应用开启了 Spring Boot Actuator 监控，那么可以访问 Actuator 来查看此应用订阅的各服务的 RibbonServerList 的信息。Actuator 地址如下：

- Spring Boot 1.x 版本：http://ip:port/migration_server_list
- Spring Boot 2.x 版本：http://ip:port/actuator/migration-server-list

metaInfo 中的 serverGroup 字段代表了此节点来源于哪个服务注册中心。



迁移其它所有应用

依照迁移第一个应用的迁移步骤，依次将所有应用迁移到 EDAS。

清理迁移配置

迁移完成后，删除原有的注册中心的配置和迁移过程专用的依赖edas-sc-migration-starter，在业务量较小的时间分批重启应用。

edas-sc-migration-starter 是一个迁移专用的 starter，虽然长期使用对您业务的稳定性没有影响，但在 Ribbon 负载均衡实现方面有一定的局限性，推荐您在迁移完毕后清理掉，然后在业务量较小的时间分批重启应用。

动态调整服务注册和订阅方式

在完成迁移过程中，您可以通过 EDAS 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

默认的订阅策略是从所有注册中心订阅，并对数据做一些简单的聚合。

您可以通过 EDAS 的配置管理来修改spring.cloud.edas.migration.subscribes属性以便选择从哪几个注册中心订阅数据。

```
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

默认的注册策略是注册到所有注册中心。

您可以通过 EDAS 的配置管理来调整服务注册中。

spring.cloud.edas.migration.registry.excludes属性来选择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
```

如果您想在应用运行时动态修改从注册到哪些注册中心，直接使用 Spring Cloud 配置管理功能在运行时修改此属性即可。

迁移问题咨询

如果在迁移过程中遇到异常情况，请加入钉钉群进行咨询。

说明：为了方便为您提供服务，请在申请加入钉钉群时备注公司名和阿里云账号。

Spring Cloud 应用优雅下线

对于任何一个线上应用，如何在服务更新或停止过程中保证消费者无感知？优雅下线机制可以做到。您既可以通过修改容器或者框架的相关配置来实现优雅下线，也可以直接将应用部署至 EDAS 上进行生命周期管理来实

现。本文将以 Spring Cloud 应用作为示例来讲述优雅下线。

背景信息

如何保证从应用停止到恢复服务期间不影响正常运行的消费者的业务请求？理想条件下，在整个服务没有请求的时候再进行更新是最安全可靠的。但实际情况下，无法保证在服务下线的同时完全没有任何调用请求。传统的解决方式是通过将应用更新流程划分为手工摘流量、停应用、更新重启三个步骤，由人工操作实现客户端无对更新感知。

如果在容器/框架级别提供某种自动化机制，来自动进行摘流量并确保处理完以到达的请求，不仅能保证业务不受更新影响，还可以极大地提升更新应用时的运维效率。这个机制就是优雅下线。

EDAS 将优雅下线的流程整合在发布流程中，对 EDAS 的 ECS 集群中的应用进行停止、部署、回滚、缩容、重置等操作时，优雅下线会自动执行。

开源 Spring Cloud 服务优雅下线方案

本章节将会介绍开源 Spring Cloud 应用中跟优雅下线相关的配置方案说明。

第一步：服务提供者注销服务

Spring Cloud Commons 中的 ServiceRegistryEndpoint 提供了服务注销的功能，可以发送一个请求使用 Endpoint 来控制服务的下线。

```
curl -X "POST" "http://{ip}:{port}/actuator/service-registry?status=DOWN" -H "Content-Type:
application/vnd.spring-boot.actuator.v2+json;charset=UTF-8"
```

在使用 Endpoint 控制服务下线的方法中，会调用接口 ServiceRegistry 的实现类的 setStatus(R registration, String status)方法，将服务提供者注册的服务从服务注册中心注销。

第二步：服务消费者更新服务提供者列表

当服务提供者从注册中心注销服务后，服务消费者会在下一次请求的服务提供者列表中摘除已经注销的服务提供者。开源 Spring Cloud 框架应用的负载均衡一般由 Ribbon 实现，故本步骤以修改 Ribbon 的刷新时间作为示例。

Ribbon 默认的刷新时间为 30s，这意味着在最差的情况下，服务提供者在下线 30 秒之后，服务消费者的请求服务列表中才能摘除掉已经注销了的服务提供者，因此您可以修改刷新时间的配置来缩短感应时间。例如修改下面的配置可将刷新时间缩短为 2000ms。

```
ribbon.ServerListRefreshInterval=2000
```

第三步：停止服务提供者应用

当服务提供者不再收到新请求后，而且在服务注销之前收到的请求也已经处理完毕后，就可以在不影响业务的情况下停止应用。

EDAS 中 Spring Cloud 应用的优雅下线

上述的开源 Spring Cloud 优雅平滑下线的方案中， ServiceRegistryEndpoint 的调用前提是引入 Spring Boot Actuator 依赖，需要在应用中添加一些额外的安全配置才使 Actuator 能被正常访问。Spring Boot 1 和 Spring Boot 2 两个版本还存在 Actuator 地址不一致的情况。

将开源 Spring Cloud 应用部署至 EDAS，需修改配置依赖为 Nacos Sever。在 EDAS 中执行应用停止相关的操作时，EDAS 会直接在服务注册中心将服务提供者的摘除，这套方案的优点是对部署的应用没有额外的配置需求，也无需强依赖 Spring Boot Actuator。您只需将服务列表的刷新时间调整到一个小于 3 秒的值即可。下文展示了在 EDAS 中执行优雅下线的操作流程。

登录 EDAS 控制台，在页面左上角选择地域。

在左侧导航栏选择**应用管理** > **应用列表**，在应用列表中单击 ECS 集群中 Spring Cloud 应用的应用名。

在应用详情页面进行以下操作，便会触发 EDAS 上的 Spring Cloud 应用的优雅下线流程。

- 在页面右上角单击**停止应用**、**部署应用**或**回滚应用**。
- 在执行完停止应用后，在页面右侧单击**批量操作实例**，在**批量操作实例**页面勾选要缩容的实例，单击**批量缩容**。
- 在**实例部署信息**页签的应用实例的操作列，单击**重置**。

在应用详情页左上角会提示**应用有变更流程正在执行，处于执行中状态**，单击该提示右侧的**查看详情**。

在**变更详情**页面可以查看优雅下线的变更记录详情。

在应用变更单中的**应用平滑下线**阶段，会将服务从 Nacos 注册中心注销。

在服务注销之后，默认在 3 秒之后，应用将被停止。

使用 Dubbo 开发应用

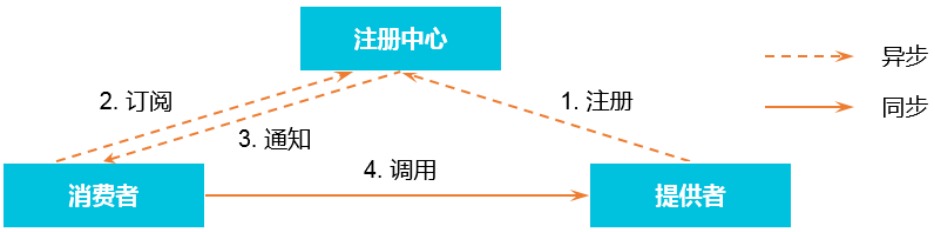
Dubbo 概述

EDAS 支持原生 Dubbo 微服务框架，您在 Dubbo 框架下开发的微服务只需添加依赖和修改配置（代码零入侵），并部署到 EDAS 之后，即可获取 EDAS 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。

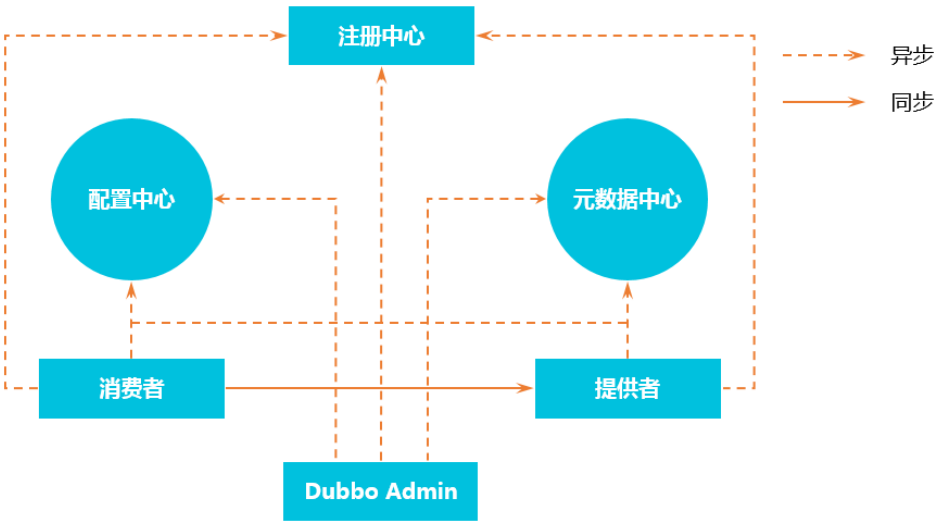
Dubbo 架构

开源 Dubbo 目前包含两个主流版本：2.6.x 和 2.7.x。两个版本的架构如下图所示。

Dubbo 2.6.x



Dubbo 2.7.x



Dubbo 服务框架的工作流程如下：

1. 提供者在启动时，向注册中心注册。
2. 消费者在启动时，向注册中心订阅所需的服务。
3. 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

。

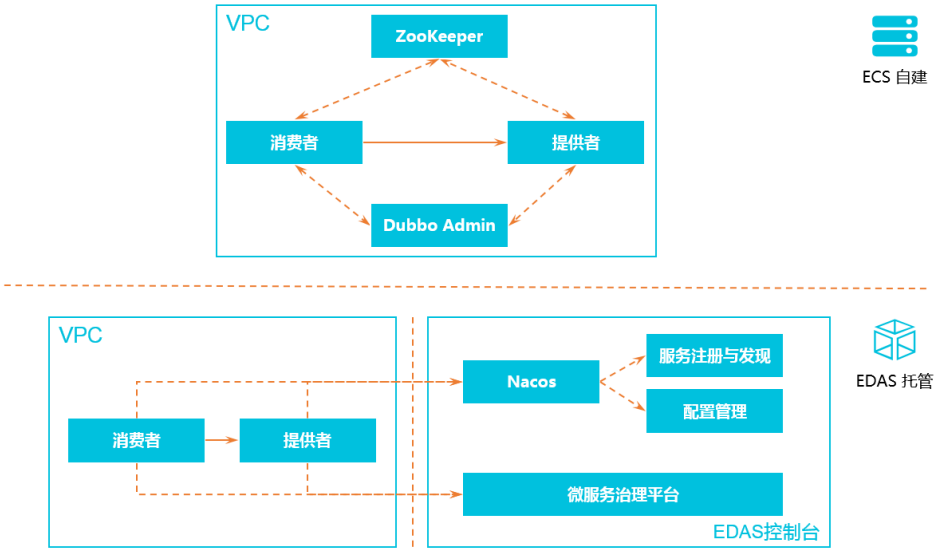
4. 消费者从提供者地址列表中，基于软负载均衡算法，选一个提供者进行调用。

将 Dubbo 应用托管到 EDAS 的价值

本节介绍将 Dubbo 应用托管到 EDAS 的含义和价值。

托管的含义

Dubbo 应用托管到 EDAS 的核心即三个中心的托管，注册中心、配置中心和元数据中心。



- ECS 自建的部署结构中，您需要自行搭建 ZooKeeper（注册中心）和 Dubbo Admin（包括部署元数据中心和配置中心）。
- 托管到 EDAS 后，Nacos（包含注册中心、配置中心和元数据中心）和 Dubbo 服务治理平台均由 EDAS 提供，您不需要关注这些组件的可用性，并且还可以体验到比自建 Dubbo Admin 更强大的微服务治理平台。

中心类型	开源组件	EDAS 组件	托管说明
注册中心	- Nacos（推荐） - ZooKeeper（推荐） - etcd - Consul - Eureka	- Nacos（推荐） - EDAS 注册中心	Nacos 为推荐的注册中心，您只需在应用中添加开源版本的dubbo-nacos-registry依赖。
配置中心	- Nacos（推荐） - ZooKeeper（推荐）	Nacos（推荐）	在应用中添加dubbo-configcenter-nacos依赖。

	- Apollo		
元数据中心	- Nacos（推荐） - Redis（推荐） - ZooKeeper	Nacos（推荐）	在应用中添加dubbo-metadata-report-nacos依赖。

托管的价值

将 Dubbo 应用托管到 EDAS，您只需要关注 Dubbo 应用自身的逻辑，无需再关注注册中心、配置中心和元数据中心的搭建和维护，托管后还可以使用 EDAS 提供的弹性伸缩、限流降级、监控及微服务治理能力，而且整个托管过程对您来说是完全透明的，不会增加理解和开发成本。托管的具体价值如下：

- 成本：无需再自行运维 Eureka、ZooKeeper、Consul 等中间件组件，可以直接使用 EDAS 提供的服务发现与配置管理能力。
- 部署：EDAS 提供了启动参数灵活配置、流程可视化、服务优雅上下线和分批发布等功能，让您的应用部署可配、可查、可控。
- 服务治理：EDAS 提供了服务查询、条件路由、黑白名单、标签路由、动态配置、负载均衡配置、权重配置和统一配置管理，您可以对应用进行全面的服务治理。
- 弹性伸缩：EDAS 提供了弹性伸缩功能，您可以根据流量高峰和低谷实时地为应用扩容和缩容。
- 限流降级：EDAS 提供了限流降级功能，保证您的应用高可用。
- 监控：EDAS 提供了高级监控功能，除了基本的实例信息查询外，您还可以查询微服务调用链、服务调用拓扑和慢 SQL。

相关操作

EDAS 支持多种 Dubbo 微服务应用的开发方式：

如果想使用 Spring Boot 开发 Dubbo 微服务应用，请参见使用 Spring Boot 开发 Dubbo 微服务应用。

如果想使用 XML 开发 Dubbo 微服务应用，请参见将 Dubbo 应用托管到 EDAS。

如果仅想体验如何托管到 EDAS，可以使用 Alibaba Cloud Toolkit 创建一个 Demo 示例，然后部署到 EDAS，详情请参见使用 Cloud Toolkit 开发 Dubbo 微服务应用样例工程。

使用 Spring Boot 开发 Dubbo 微服务应用

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，这两者结合的方式能够帮助您提升开发效率。本文介绍如何使用 Spring Boot 注解的方式基于 Nacos 开发一个 Dubbo 微服务示例应用。

前提条件

在使用 Spring Boot 开发 Dubbo 微服务应用前，请先完成以下工作：

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。

示例工程

您可以按照本文的步骤一步步搭建出最终的工程，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

创建一个 Maven 工程，命名为spring-boot-dubbo-provider。

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>
```

开发 Dubbo 服务提供者。

Dubbo 中服务都是以接口的形式提供的。

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

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);
}
```

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** 。

配置 Dubbo 服务。

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

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：

i. 以上三个配置没有默认值，必须要给出具体的配置。
ii. 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。
iii. 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。

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);
}

}
```

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

创建一个 Maven 工程，命名为spring-boot-dubbo-consumer。

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 8 月 结束，推荐使用新版本开发您的应用。

开发 Dubbo 消费者

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

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);
}
```

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 。

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。

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);
}

}
```

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**，查看调用者列表。

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/EDAS`

`Hello, EDAS (from Dubbo with Spring Boot)`
```

部署到 EDAS

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

您可以根据实际需求选择部署的集群类型（主要为 ECS 集群或容器服务 Kubernetes 集群）和部署途径（控制台或工具），详情请参见部署应用概述。

如果您使用控制台部署，在部署前，需要在本地应用程序中完成以下操作：

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>
```

执行 mvn clean package 将本地的程序打成 JAR 包。

更多信息

除 Spring Boot 外，还可以通过 XML 的方式开发 Dubbo 微服务应用，详情请参见请参见将 Dubbo 应用托管到 EDAS。

如果您使用了edas-dubbo-extension，请参见通过 edas-dubbo-extension 将 Dubbo 应用托管到 EDAS。由于edas-dubbo-extension的方式不能使用 EDAS 提供的相关功能，如 Dubbo 服务治理，所以不推荐使用，建议您迁移到 Nacos。

使用 Cloud Toolkit 开发 Dubbo 微服务应用样例工程

Alibaba Cloud Toolkit（简称 Cloud Toolkit）是为开发者提供的一款 IDE 插件，您可以使用 Cloud Toolkit 快速创建 Apache Dubbo 应用样例工程，验证后，再使用 Cloud Toolkit 部署到 EDAS 上。本文以 IntelliJ IDEA 为例介绍如何创建一个以 Nacos 作为注册中心的 Apache Dubbo 应用样例工程。

准备工作

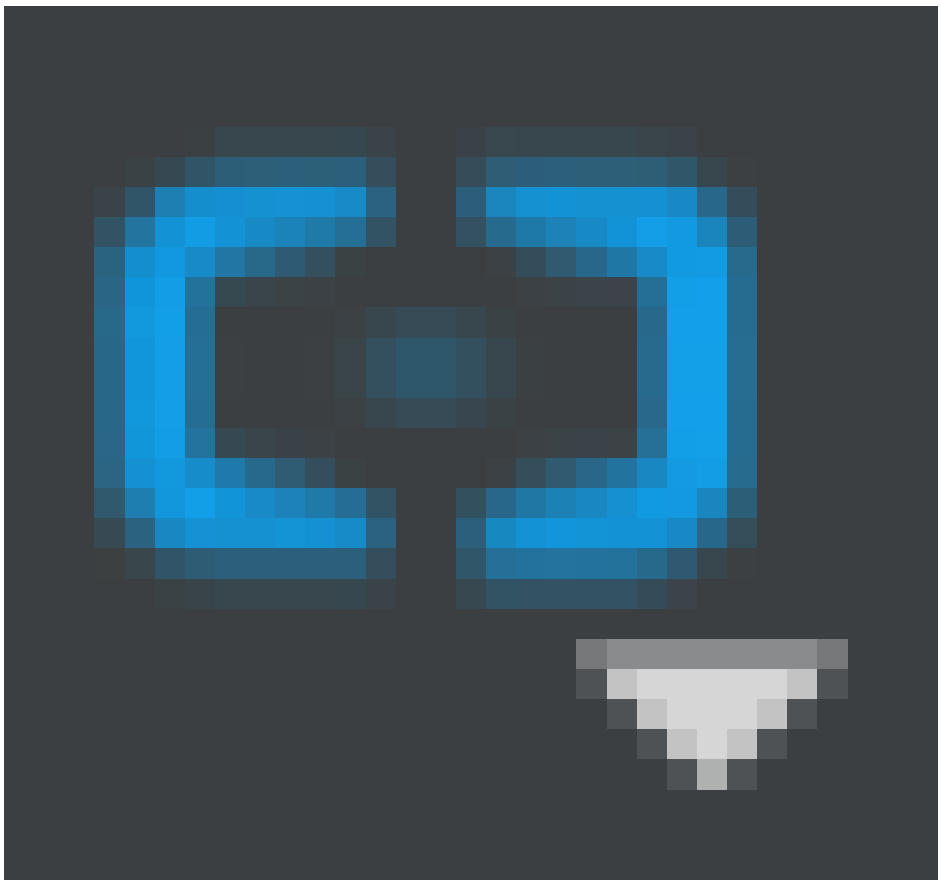
在使用 Cloud Toolkit 创建 Apache Dubbo 应用样例工程前，请完成以下工作：

在 IntelliJ IDEA 中安装和配置 Cloud Toolkit

如果您使用 Eclipse，请参见在 Eclipse 中安装和配置 Cloud Toolkit。

如果您此前已经安装了 Cloud Toolkit，请检查 Cloud Toolkit 是否为 2019.6.2 及以上版本。如果低于 2019.6.2，请升级。

在 IntelliJ IDEA 的工具栏单击 Cloud Toolkit 图标

 ，在下拉菜单中选择 **About**。

在 **About Alibaba Cloud Toolkit** 对话框中查看版本信息。

在本地启动 Nacos Server。

本文的样例工程的注册中心为 Nacos，在创建工程前，还需要现在本地启动 Nacos Server。操作步骤如下：

下载最新版本的 Nacos Server 。

解压下载的 Nacos Server 压缩包。

进入nacos/bin目录，启动 Nacos Server。

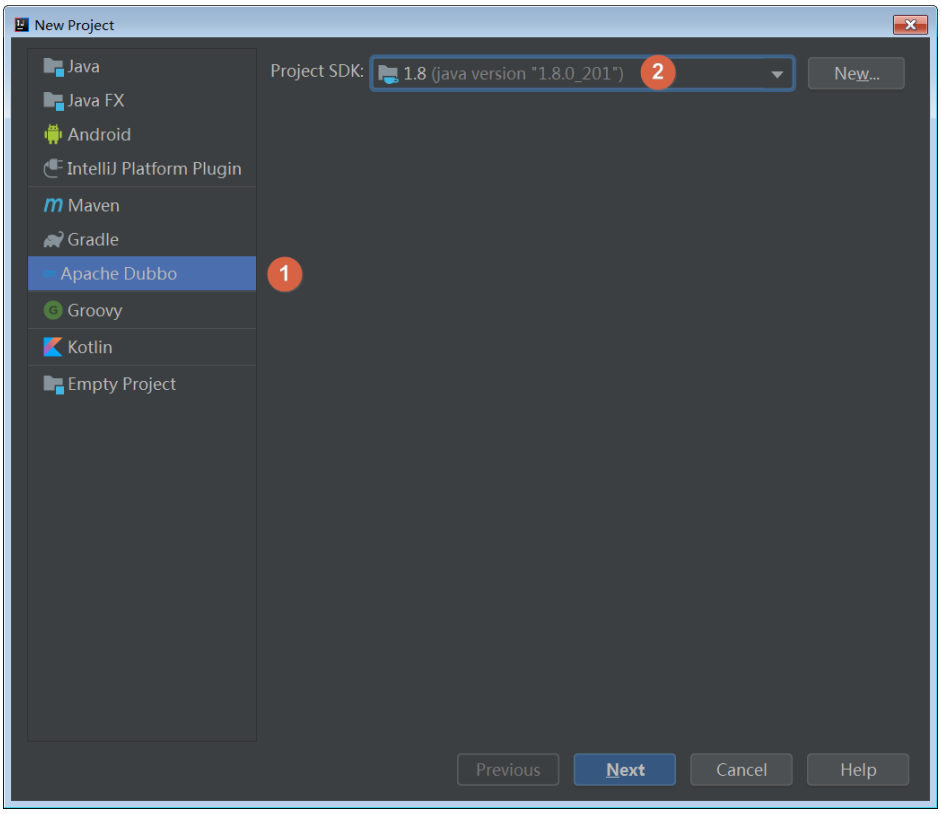
- Linux/Unix/Mac 系统：执行命令sh startup.sh -m standalone。
- Windows 系统：双击执行startup.cmd文件。

创建 Apache Dubbo 应用样例工程

创建 Apache Dubbo 应用样例工程的步骤如下：

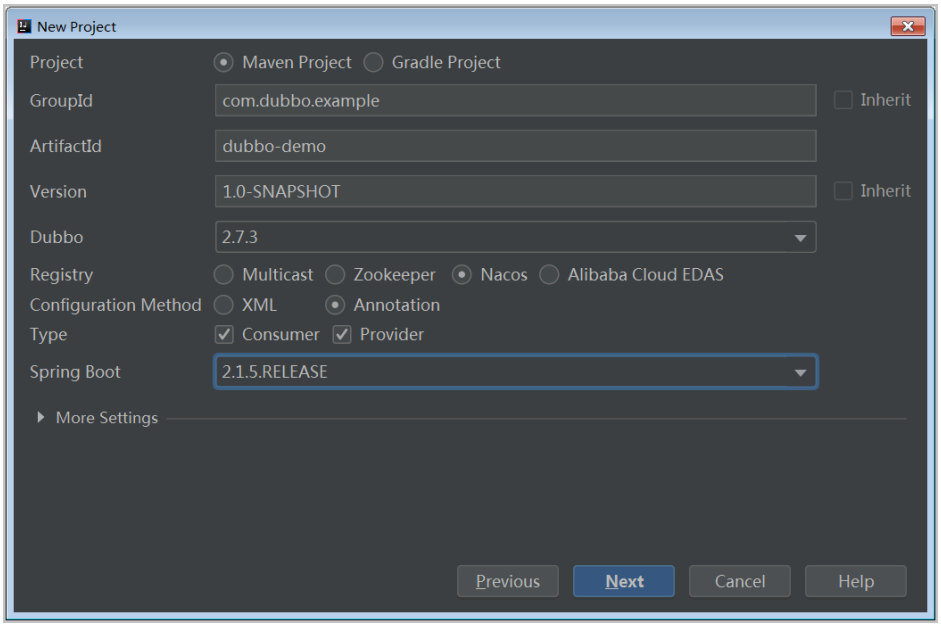
启动 IntelliJ IDEA，在菜单栏选择 **File** > **New** > **Project**。

在 **New Project** 对话框左侧的导航栏中单击 **Apache Dubbo**，在右侧界面中选择本地安装的 **JDK**，然后单击 **Next**。



设置样例工程的参数，单击 **Next**。

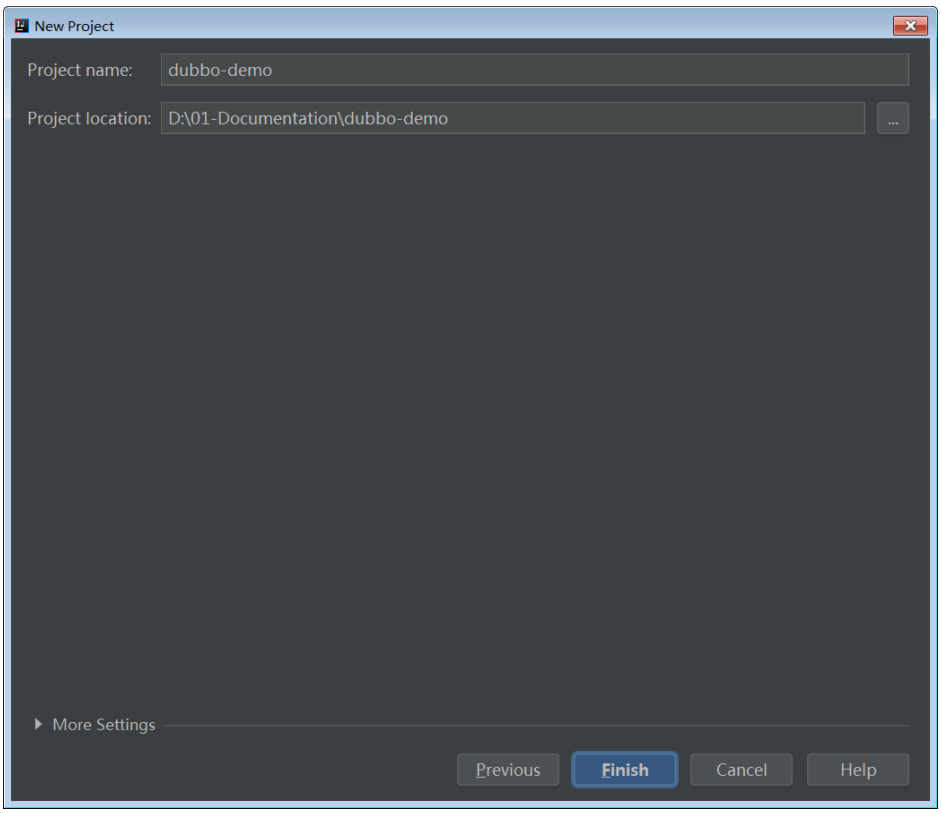
工程参数主要包括 Dubbo 的版本、注册中心、配置方式和 Spring Boot 的版本。



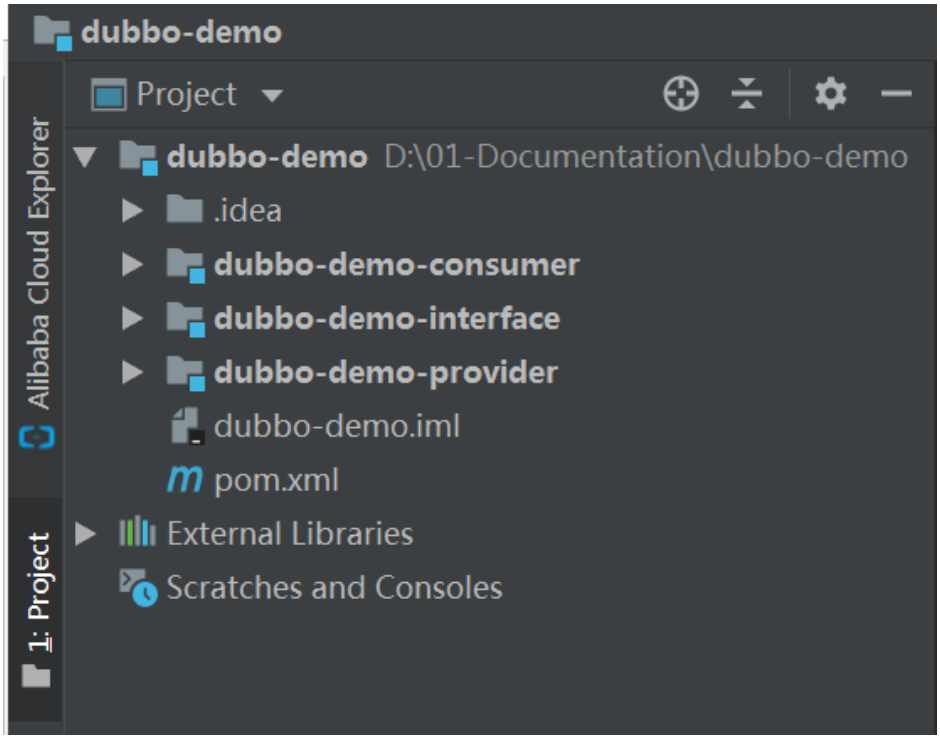
参数说明及示例：

- **Project**： 选择 **Maven Project**。
- **GroupId**： 输入相应的 Group ID，如 **com.dubbo.example**。
- **ArtifactId**：输入 **dubbo-demo**。
- **Version**: 应用工程的版本，如 **1.0-SNAPSHOT**。
- **Dubbo**：在下拉菜单中选择 Dubbo 的版本，如 **2.7.3**。
- **Registry**：选择样例工程的注册中心，推荐选择 **Nacos**。
- **Configuration Method**：样例工程的开发方式。推荐使用 **Annotation**。
- **Type**：工程的服务类型，勾选 **Consumer** 和 **Provider**，则会创建服务提供者和服务消费者的工程 Demo。
- **Spring Boot**：Spring Boot 的版本，如 **2.1.5.RELEASE**。

设置 **Project name** 和 **Project location**，然后单击 **Finish**。



创建完成后，可以在 IntelliJ IDEA 中可以看到 Apache Dubbo 样例工程。此样例工程中包含 Provider、Consumer 和调用的接口。



验证 Apache Dubbo 应用样例工程

本文的样例工程使用 Nacos 为注册中心，且使用了注解的开发方式，所以需要验证 Demo 中的代码和配置。同时，由于同时创建了 Provider 和 Consumer，所以还需要验证服务调用是否成功。

该样例工程是一个 Spring Boot + Dubbo 的工程，检查pom.xml中包含 Apache Dubbo 的依赖。

```
<dependency>
<groupId>org.apache.dubbo</groupId>
<artifactId>dubbo</artifactId>
<version>${dubbo.version}</version>
</dependency>
```

该样例工程注册中心选择了 Nacos，检查pom.xml中包含 Nacos Registry 的依赖。

```
<dependency>
<groupId>org.apache.dubbo</groupId>
<artifactId>dubbo-registry-nacos</artifactId>
<version>${dubbo.version}</version>
</dependency>
```

同时在application.properties中检查本地 Nacos Server 的配置。

```
## Dubbo Registry
dubbo.registry.address=nacos://localhost:8848
```

该样例工程包含 Provider 和 Consumer。

Provider 提供了一个服务com.dubbo.example.DemoService，通过 Dubbo 协议暴露在12345端口，配置在application.properties中。

```
# Dubbo Protocol
dubbo.protocol.name=dubbo
dubbo.protocol.port=12345
```

Consumer 通过 Nacos Server 调用 Provider 提供的服务。

```
@Reference(version = "1.0.0")
private DemoService demoService;
```

验证服务调用

启动 Provider。

在 IntelliJ IDEA 中运行（run）com.dubbo.example.provider.DubboProviderBootstrap的main函数。

观察标准输出。

出现以下字段，说明 Provider 启动成功。

```
2019-07-03 16:05:50.585  INFO 19246 --- [           main] c.d.e.provider.DubboProviderBootstrap    : Started DubboProviderBootstrap in 36.512 seconds (JVM running for 42.004)
2019-07-03 16:05:50.587 INFO 19246 --- [pool-1-thread-1] .b.c.e.AwaitingNonWebApplicationListener : [Dubbo] Current Spring Boot Application is await...
```

启动 Consumer 并验证调用。

在 IntelliJ IDEA 中运行（Run）com.dubbo.example.consumer.DubboConsumerBootstrap的main函数。

观察服务端（Provider）的打印日志，出现以下字段，则说明调用成功。

```
Hello Provider, response from provider: 30.5.125.39:12345
```

部署 Apache Dubbo 应用样例工程

在完成 Apache Dubbo 样例工程的创建和验证后，可以使用 Cloud Toolkit 将该样例工程打包（JAR 包）并根据需要部署到 EDAS 的不同集群中。详情请参见：

- 使用 Cloud Toolkit 部署应用到 ECS 集群
- 使用 Cloud Toolkit 部署应用到容器服务 Kubernetes 集群

Dubbo 应用部署到 EDAS 之后，可以进行服务治理，详情请参见 Dubbo 服务治理。

将 Dubbo 应用平滑迁移到 EDAS

如果您的 Dubbo 应用已经部署在阿里云上，那么本文档将向您介绍如何将应用平滑迁移到 EDAS 中，并实现基本的服务注册与发现。如果您的 Dubbo 应用还未部署到阿里云，请提交工单或联系 EDAS 技术支持人员为

您提供完整的上云及迁移到 EDAS 方案。

迁移到 EDAS 的价值

EDAS 为应用部署提供了启动参数灵活配置、流程可视化、服务优雅上下线和分批发布等功能，让您的应用发布可配、可查、可控。

EDAS 提供了服务发现与配置管理功能，您无需再自行运维 Eureka、ZooKeeper、Consul 等中间件组件，可以直接使用 EDAS 提供的商业版服务发现与配置管理。

EDAS 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。

EDAS 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。

EDAS 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。

EDAS 提供限流降级功能，保证您的应用高可用。

EDAS 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 Dubbo 应用已经部署到生产环境并处于正常运行状态中，此时想将应用迁移到 EDAS 享受完整的 EDAS 功能，那么在迁移过程中，保证业务的平稳运行不中断是第一要务，而保证应用平台运行不中断迁移到 EDAS 即为平滑迁移。

说明：如果您的应用尚未在生产环境中运行，或者您可以接受停机迁移，则没必要按照本文进行平滑迁移，可直接将应用在本地开发完再部署到 EDAS，详情请参见将 Dubbo 应用托管到 EDAS 。

迁移流程

下图是一个比较典型的应用架构，根据迁移的先后顺序需要将迁移流程分为三步。

（**必选**）迁移应用

迁移的应用一般都是无状态的，所以这一步是可以最先进行的。同时，本文将重点介绍如何迁移应用。

（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，在应用迁移后，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，详情请参见 SLB 绑定概述。

如果您的应用在迁移之前没有使用 SLB，建议您在迁移完入口应用（如上图所示的 API Gateway）后，为该应用创建并绑定一个新的 SLB。

迁移应用的方案中，我们推荐您使用双注册和双订阅方案，以节约您的 ECS 成本。但如果由于某种原因（如原 ECS 端口被占用）不能复用之前的 ECS，则需要采用切流迁移方案，您需要添加新的 ECS用于迁移应用。在应用迁移完成后，参考上面描述的 SLB 的状态，选择复用 SLB 或创建 SLB 并绑定到应用。

域名

如果迁移后的应用可以复用 SLB，域名配置也无需修改。

如果迁移后的应用需要创建新的 SLB 并绑定到应用，则需要在域名中添加新的 SLB 配置，详情请参见域名 DNS 修改，并删除原来不再使用的 SLB。

（可选）迁移存储和消息队列

- 如果你之前的应用已经部署在阿里云上，则存储和消息队列也使用了阿里云相关产品（如 RDS、MQ 等），则应用迁移完成后，之前的存储和消息队列无需迁移。
- 如果您之前的应用不在阿里云上，请提交工单或联系 EDAS 技术支持人员为您提供完整的上云及迁移到 EDAS 方案。

本文将主要介绍如何迁移应用。如果您想通过一个 Demo 快速体验平滑迁移的过程，可下载 Provider Demo，Consumer Demo，参考 Readme 运行一个迁移的样例。

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。这两种方案都可以保证您的应用正常运行不中断的完成迁移。

说明：本文将主要介绍如果使用双注册和双订阅方案迁移应用。

切流迁移方案

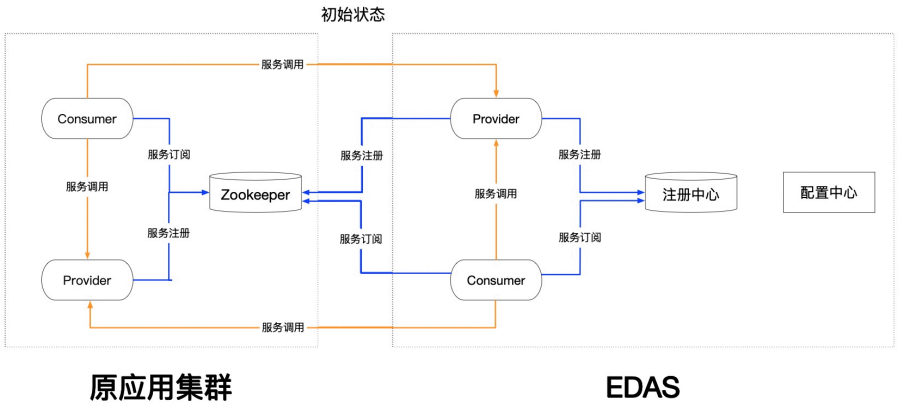
使用 Dubbo 将原有的服务注册中心切换到 EDAS ConfigServer，开发一套新的应用部署到 EDAS，最后通过 SLB 和域名配置来进行切流。

如果您选择此方案，那您可以参考将 Dubbo 应用托管到 EDAS 开发应用，不需要再阅读迁移应用的后续内容。

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 EDAS 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。

通过双注册和双订阅平滑迁移应用的架构图如下：



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，只需要添加依赖，并修改一行代码，就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

步骤一：选择最先迁移的应用

建议是从最下层 Provider 开始迁移。但如果调用链路太复杂，比较难分析，也可以任意选一个应用进行迁移。选择完成后，即可参考下面的迁移步骤迁移第一个应用。

步骤二：在应用程序中添加依赖并修改配置 (双注册、双订阅)

为了能将您原来的应用托管到 EDAS 中，您需要在您的应用程序中添加相关依赖并修改配置。

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>
```

在 application.properties 中添加注册中心的地址。

```
dubbo.registry.address = edas-migration://30.5.124.15:9999?service-
registry=edas://127.0.0.1:8080,zookeeper://172.31.20.219:2181&reference-
registry=zookeeper://172.31.20.219:2181&config-address=127.0.0.1:8848
```

说明：如果是非 Spring Boot 应用，在 dubbo.properties 或者对应的 Spring 配置文件中配置。

edas-migration://30.5.124.15:9999

多注册中心的头部可以不做更改，启动的时候，如果日志级别是 WARN 及以下，可能会抛一个 WARN 的日志，因为 Dubbo 会对 IP 和端口做校验，可以忽略。

service-registry是服务注册的注册中心地址，默认会进行多注册，写入多个注册中心地址。每个注册中心都是标准的 Dubbo 注册中心格式；多个用,分隔。 其中 ZooKeeper 的地址 *172.31.20.219* 为实例，请使用真实的 ZooKeeper 地址和端口。`

reference-registry是服务订阅的注册中心地址，可以进行多注册或者先注册到老的注册中心都可以。每个注册中心都是标准的 Dubbo 注册中心格式；多个用,分隔。

config-address是动态推送的地址，如果本地想进行尝试，需要下载 Nacos。 EDAS 会对这个地址进行转换。

其他修改。

对于非 Spring Boot 的 Spring 应用，将 com.alibaba.edas.dubbo.migration.controller.EdasDubboRegistryRest加入到你的扫描路径里。

步骤三：本地验证

如果只想进行一次修改，则可以使用动态配置的方式。

准备工作

下载 ZooKeeper

配置轻量配置中心

下载并启动 Nacos

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。

（可选）登录 Nacos，配置对应的服务注册信息。

说明：如果不需要动态配置的话，可以跳过此步骤。

DataId：dubbo.registry.config

Group： 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：包含两种，一种是应用级别，一种是针对实例的 IP 级别（多张网卡可能出现问题）。

应用级别

```
dubbo.reference.registry=edas://127.0.0.1:8080  ##注册服务的注册中心
dubbo.service.registry=edas://127.0.0.1:8080,zookeeper:127.0.0.1:2181 ##订阅服务的注册中心
```

实例 IP 级别

```
169.254.15.86.dubbo.reference.registry=edas://127.0.0.1:8080,zookeeper:127.0.0.1:2181
169.254.15.86.dubbo.service.registry=edas://127.0.0.1:8080
```

当对集群验证的时候，可以先从实例 IP 级别，再从整个应用验证这些配置是否生效依赖于应用的修改。

查看应用调用是否正常，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


```
{
  "dubbo.effective.reference.registry": [
    "edas://127.0.0.1:8080"
  ],
  "dubbo.orig.reference.registry": [
    "zookeeper://127.0.0.1:2181"
  ],
  "dubbo.orig.service.registry": [
    "edas://127.0.0.1:8080",
    "zookeeper://127.0.0.1:2181"
  ],
  "dubbo.effective.service.registry": [
    "edas://127.0.0.1:8080",
    "zookeeper://127.0.0.1:2181"
  ]
}
```

步骤四：将修改后的应用部署到 EDAS 中

您可以根据您的实际需求将应用部署到 ECS 集群或容器服务 Kubernetes 集群中，在部署时也可以选择通过控制台、工具等方式进行部属。详情请参见部署应用概述。

为了帮助您节约成本，建议您继续使用之前 ECS，但需要将 ECS 导入到 EDAS 中，详情请参见导入 ECS 。**在导入 ECS 的时候如果提示需要转化后导入，请对重要的数据做好备份。**

如果需要创建新的 ECS、集群等资源，请确保在原有 VPC 内创建，以保证迁移前后的应用网络互通，顺利完成迁移。详情请参见创建资源。

在数据库、缓存、消息队列等产品中为新 ECS 配置 IP 白名单等，确保应用依赖的这些第三方组件可以正常访问。

结果验证

最重要的是观察业务本身是否正常。

查看服务订阅监控。

如果您的应用开启了 Spring Boot Actuator 监控，那么可以访问 Actuator 来查看此应用订阅的各服务的 RibbonServerList 的信息。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.effective.**表示生效的注册中心信息。

迁移其它所有应用

依照迁移第一个应用的迁移步骤，依次将所有应用迁移到 EDAS。

清理迁移配置

迁移完成后，删除原有的注册中心的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，即将 ZooKeeper 的配置删除，保证 Consumer 只从 EDAS 订阅，Provider只在 EDAS 订阅。有以下两种方式：

方式一：动态配置

可以直接参考**步骤四：本地验证**里的配置修改。

方式二：手动修改

当所有的应用都修改完成之后，修改应用的注册中心地址，将订阅的地址改为 EDAS ConfigServer。

```
dubbo.registry.address = edas-migration://30.5.124.15:9999?service-
registry=edas://127.0.0.1:8080,zookeeper://172.31.20.219:2181&reference-
registry=edas://127.0.0.1:8080&config-address=127.0.0.1:8848
```

reference-registry的值从zookeeper://172.31.20.219:2181改为edas://127.0.0.1:8080。修改完成之后，即可部署应用。

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://172.31.20.219:2181，最后就变成了如下地址。

```
dubbo.registry.address = edas://127.0.0.1:8080
```

虽然长期使用对您业务的稳定性没有影响，但增加了 Dubbo 使用注册中心的复杂性和出错率，推荐您在迁移完毕后清理掉，然后在业务量较小的时间分批重启应用。

Dubbo 应用优雅下线

对于任何一个线上应用，如何在服务更新部署过程中保证客户端无感知是开发者必须要解决的问题，即从应用停止到重启恢复服务这个阶段不能影响正常的业务请求。在应用执行部署、停止、回滚、缩容、重置时，需要通过优雅下线的配置来保证应用正常关闭。

背景信息

优雅下线是指在执行部署应用、停止应用、回滚应用、应用缩容、重置应用时，执行的一系列保证应用正常关闭的操作，优雅下线可以避免非正常关闭应用导致数据异常或丢失、应用调用异常等问题。

如何保证从应用停止到恢复服务期间不影响正常运行的消费者的业务请求？理想条件下，在整个服务没有请求的时候再进行更新是最安全可靠的。但实际情况下，无法保证在服务下线的同时完全没有任何调用请求。传统的解决方式是通过将应用更新流程划分为手工摘流量、停应用、更新重启三个步骤，由人工操作实现客户端无对更新感知。

如果在容器/框架级别提供某种自动化机制，来自动进行摘流量并确保处理完以到达的请求，不仅能保证业务不受更新影响，还可以极大地提升更新应用时的运维效率。这个机制就是优雅下线。

EDAS 将优雅下线的流程整合在发布流程中，对 EDAS 的 ECS 集群中的应用进行停止、部署、回滚、缩容、重置等操作时，优雅下线会自动执行。

Dubbo 服务优雅下线配置说明

本章节将会介绍开源 Dubbo 中跟优雅停机相关的配置说明。

服务的优雅停机

在 Dubbo 中，优雅停机是默认开启的，默认停机等待时间为 10000 毫秒。您可以通过配置 dubbo.service.shutdown.wait 来修改等待时间。例如将等待时间设置为 20 秒可通过增加以下配置实现：

```
dubbo.service.shutdown.wait=20000
```

优雅下线不保证会等待所有已发送/到达请求结束，为了保证处理中的请求不被中断，Dubbo 提供了该参数作为一个关闭应用前的缓冲时间，请根据业务方法的处理时间合理设置关闭应用时的等待时间。

容器的优雅停机

当使用 org.apache.dubbo.container.Main这种容器方式来使用 Dubbo 时，也可以通过配置 dubbo.shutdown.hook 为 true 来开启优雅停机。

通过 QoS 优雅下线

基于 ShutdownHook 方式的优雅停机无法确保所有关闭流程一定执行完，所以 Dubbo 推出了多段关闭的方式来保证服务完全无损。

多段关闭即将停止应用分为多个步骤，通过运维自动化脚本或手工操作的方式来保证脚本每一阶段都能执行完毕。

在关闭应用前，首先通过 QoS 的 offline 指令下线所有服务，由于服务已经在注册中心下线，故当前应用不会有新的请求，然后等待一定时间确保到达的请求已经全部处理完毕。这时再执行真正的关闭(SIGTERM 或 SIGINT)流程，就能保证服务无损。

QoS 可通过 telnet 或 HTTP 方式使用，具体说明请参见 Dubbo-QOS命令使用说明。

EDAS Dubbo 优雅下线

Dubbo 框架本身提供的优雅下线能力在系统运行期间，需要运维系统去搭配框架使用，才能使该功能生效。Dubbo 应用在 EDAS 上部署后，可以在 EDAS 上进行 Dubbo 应用的应用生命周期管理，在 EDAS 内执行 Dubbo 框架应用的停机操作时，均可以在应用变更单中查看优雅下线的记录。

登录 EDAS 控制台，在页面左上角选择地域。

在左侧导航栏选择**应用管理** > **应用列表**，在应用列表中单击 ECS 集群中 Dubbo 应用的应用名。

在应用详情页面进行以下操作，均会触发 EDAS 上的 Spring Cloud 应用的优雅下线流程。

- 在页面右上角单击**停止应用**、**部署应用**或**回滚应用**。
- 在执行完停止应用后，在页面右侧单击**批量操作实例**，在**批量操作实例**页面勾选要缩容的实例，单击**批量缩容**。

- 在**实例部署信息**页签的应用实例的操作列，单击**重置**。

在应用详情页左上角会提示**应用有变更流程正在执行，处于执行中状态**，单击该提示右侧的**查看详情**。

在**变更详情**页面可以查看优雅下线的变更记录详情。

i. 在应用变更单中的**应用平滑下线**阶段，会将服务从 Nacos 注册中心注销。

在**执行应用停止**流程中，会检测 Dubbo 应用是否开启了 QoS 端口，如已开启，则调用offline指令下线服务，停止应用的进程接收到指令后执行下线操作。

```
2019-08-19 15:01:46.629  INFO 31887 --- [ qos-worker-3-1] o.apache.dubbo.qos.command.impl.Offline
:  [DUBBO] receive offline command, dubbo version: 2.7.3, current host: 30.5.121.2
2019-08-19 15:01:46.630 INFO 31887 --- [ qos-worker-3-1] o.a.dubbo.registry.nacos.NacosRegistry :
[DUBBO] Unregister:
dubbo://30.5.121.2:20883/com.alibaba.middleware.sp.dubbo.samples.DemoApi?anyhost=true&applicati
on=dubbo-nacos-provider-from-
d&bean.name=demo1&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&group=DUBB
O&interface=com.alibaba.middleware.sp.dubbo.samples.DemoApi&methods=hi,echo,test1,plus&pid=3
1887&register=true&release=2.7.3&revision=1.0
```

注意：

使用 kill -9 pid 关闭应用不会执行优雅停机，建议使用 EDAS 控制台提供的停止应用功能；

配置的优雅停机等待时间 timeout 不是所有步骤等待时间的总和，而是每一个 destroy 执行的最大时间。例如配置等待时间为 5 秒，则关闭 Server、关闭 Client 等步骤会分别等待 5 秒。

- EDAS 会尝试访问机器的 22222 端口，调用 qos 指令，用户需要配置开启 qos 指令并且暴露默认端口号：22222。

使用 HSF 开发应用

HSF 概述

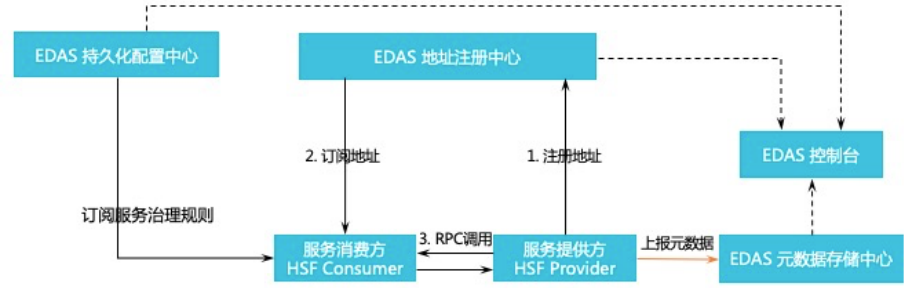
HSF 概述

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

HSF 联通不同的业务系统，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/调用方式，从而帮助用户可以方便、快速的开发分布式应用，以及提供或使用公共功能模块。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/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，本身是没有服务端集群的，所有的 HSF 服务调用都是服务消费方（Consumer）与服务提供方（Provider）点对点进行的。然而，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

服务提供方会绑定一个端口（一般是12200），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

消费服务提供方提供的服务，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，地址注册中心作为旁路不参与调用。

EDAS 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。因为作为服务提供端，没有办法将自己的服务信息对外发布，让外界知晓；作为服务消费端，可能已经知道需要调用的服务，但是无法获取能够提供这些服务的机器。而注册中心就是服务信息的中介，提供服务发现的能力。

EDAS 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会向持久化配置中心订阅各种服务治理规则，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。

EDAS 元数据存储中心

元数据是指 HSF 服务对应的方法列表以及参数结构等信息，元数据不会对 HSF 的调用过程产生影响，因此元数据存储中心也并不是必须的。但考虑到服务运维的便捷性，HSF客户端在启动时会将元数据上报到元数据存储中心，以便提供给服务运维使用。

EDAS 控制台

EDAS 控制台通过打通地址注册中心、持久化配置中心、元数据存储中心，为用户提供了一些列服务运维功能，包括服务查询、服务治理规则管理等，旨在提高 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。

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来说，并不是所有的 HSF 服务都需要同步等待返回结果的。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 调用操作上。HSF 的异步调用，有 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，当指定的 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。而泛化调用是指不需要依赖服务的二方包，从而发起 HSF 调用、获取返回结果的方式。在一些平台型的产品中，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。

HTTP 调用

HSF 支持将服务以 HTTP 的形式暴露出来，从而支持非 Java 语言的客户端以 HTTP 协议进行服务调用。

调用链路 Filter 扩展

HSF 内部设计了调用过滤器，并且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，使扩展方能够方便的对 HSF 请求进行扩展处理。

应用开发方式

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

- **Ali-Tomcat**： 依赖 Ali-Tomcat 和 Pandora，可以提供完整的 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。应用程序需要以 WAR 包方式部署。
- **Pandora Boot**：依赖 Pandora，可以提供比较完整的 HSF 功能，包括服务注册与发现和异步调用。应用程序可以打包成独立运行的 JAR 包并部署。

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 EDAS 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。

升级说明

原轻量级配置中心已升级为轻量级配置及注册中心。轻量级配置及注册中心兼容轻量级配置中心的使用场景，同时，增加了对 Nacos 的支持。

升级前后的功能对比如下：

功能	轻量级配置中心	轻量级配置及注册中心
使用 ACM 实现配置管理	支持	支持
使用 ANS 实现服务注册发现	支持	支持
HSF 应用的服务注册发现和配置管理	支持	支持
使用 Nacos 实现服务注册发现和配置管理	不支持	支持

说明：原来使用轻量级配置中心的用户，可以直接使用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：

下载轻量级配置及注册中心压缩包。

在本地解压压缩包。

Unix：

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。

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

进入目录edas-lightweight\bin。

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。

```
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.201.b09-2.el7_6.x86_64/bin/java  -server -Xms1g -Xmx1g
-Xmn512m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320 m -XX:-
OmitStackTraceInFastThrow -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=/root/edas-lightweight/logs/java_heapdump.hprof -XX:-UseLargePages  -
Djava.ext.dirs=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.201.b09-
2.el7_6.x86_64/jre/lib/ext:/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.201.b09-2.el7_6.x86_64
/lib/ext:/root/edas-lightweight/plugins/cmdb:/root/edas-lightweight/plugins/mysql -
Xloggc:/root/edas-lightweight/logs/nacos_gc.log -verbose:gc -XX:+P rintGCDetails -
XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation -
XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M -Dnacos.home=/ root/edas-
lightweight -Dnacos.standalone=true -jar /root/edas-lightweight/target/edas-lightweight.jar
--spring.config.location=classpath:/,classpath :/config/,file:./,file:./config/,file:/root/edas-
lightweight/conf/ --logging.config=/root/edas-lightweight/conf/nacos-logback.xml --
server.max-http-h eader-size=524288
edas lightweight is starting with standalone
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。

（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/Dubbo 的服务注册及订阅。
- 8080: 支持 ANS 的服务注册以及 ACM 的配置管理。

Linux & 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

   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：

- 轻量级配置及注册中心可用性。
- 功能可用性，包括配置管理、服务注册和命名空间（仅适用于之前使用 Nacos 的用户）。

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

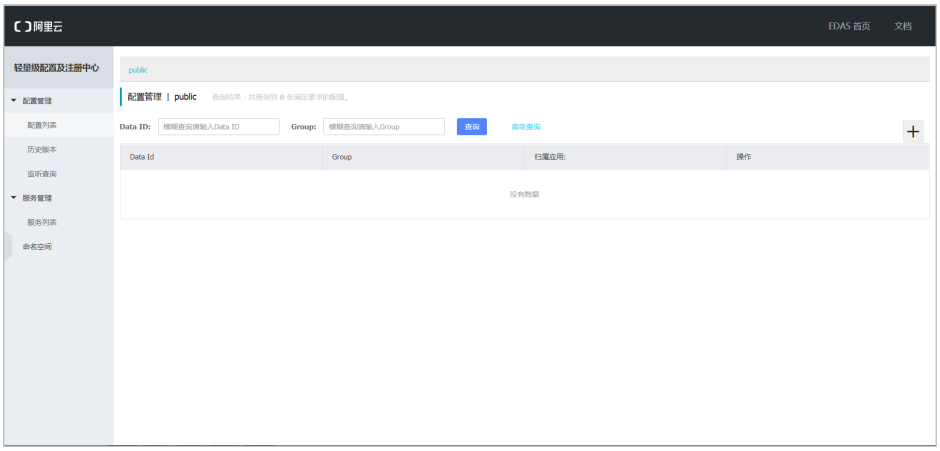
在浏览器中输入轻量级配置及注册中心地址*http://127.0.0.1:8080*并回车。

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（仅适用于原有 Nacos 用户）功能。

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。

相关文档

在您使用轻量级配置及注册中心的过程中，如果遇到问题，可以参考轻量级配置及注册中心处理。

使用 Ali-Tomcat 开发应用

Ali-Tomcat 概述

Ali-Tomcat 是 EDAS 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。EDAS 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。您可以利用这些插件对 EDAS 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 EDAS 中，只有 WAR 包格式的 HSF 应用才需要使用 Ali-Tomcat。

安装 Ali-Tomcat 和 Pandora 并配置开发环境

安装 Ali-Tomcat 和 Pandora

Ali-Tomcat 和 Pandora 为 EDAS 中的服务运行时所依赖的容器，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，均必须将应用程序发布在该容器中。

注意：请使用 JDK 1.7及以上版本。

下载 Ali-Tomcat，保存后解压至相应的目录（如：d:\work\tomcat\）。

下载 Pandora 容器，保存后将内容解压至上述保存的 Ali-Tomcat 的 deploy 目录(d:\work\tomcat\deploy)下。

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 问题进行定位、解决。

配置开发环境

您在本地开发应用时，需要使用 Eclipse 或 IntelliJ IDEA。本节将分别介绍如何配置 Eclipse 或 IntelliJ IDEA 开发环境。

配置 Eclipse 环境

配置 Eclipse 需要下载 Tomcat4E 插件，并存放在安装 Ali-Tomcat 时 Pandora 容器的保存路径中，配置之后开发者可以直接在 Eclipse 中发布、调试本地代码。具体步骤如下：

下载 Tomcat4E 插件，并解压至本地（如：d:\work\tomcat4e\）。

压缩包内容如下：

打开 Eclipse，在菜单栏中选择 **Help** > **Install New Software**。

在 Install 对话框中 Work with 区域右侧单击 **Add**，然后在弹出的 Add Repository 对话框中单击 **Local**。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**。

返回 Install 对话框，单击 **Select All**，然后单击 **Next**。

后续还有几个步骤，按界面提示操作即可。安装完成后，Eclipse 需要重启，以使 Tomcant4E 插件生效。

重启 Eclipse。

重启后，在 Eclipse 菜单中选择 **Run As** > **Run Configurations**。

选择左侧导航选项中的 **AliTomcat Webapp**，单击上方的 **New launch configuration** 图标。

在弹出的界面中，选择 **AliTomcat** 页签，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。

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。

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:     192.168.8.1                                         **
**      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 环境

注意：目前仅支持 IDEA 商业版，社区版暂不支持。所以，请确保本地安装了商业版 IDEA。

运行 IntelliJ IDEA。

从菜单栏中选择 **Run** > **Edit Configuration**。

在 **Run/Debug Configuration** 页面左侧的导航栏中选择 **Defaults** > **Tomcat Server** > **Local**。

配置 AliTomcat。

在右侧页面单击 **Server** 页签，然后在 **Application Server** 区域单击 **Configure**。

在 **Application Server** 页面右上角单击 **+**，然后在 **Tomcat Server** 对话框中设置 **Tomcat Home** 和 **Tomcat base directory** 路径，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。

在 **Application Server** 区域的下拉菜单中，选择刚刚配置好的 Ali-Tomcat。

在 **VM Options** 区域的文本框中，设置 JVM 启动参数指向 Pandora 的路径，如：-Dpandora.location=d:\work\tomcat\deploy\taobao-hsf.sar

说明：*d:\work\tomcat\deploy\taobao-hsf.sar* 需要替换为在本地安装 Pandora 的实际路径。

单击 **Apply** 或 **OK** 完成配置。

开发 HSF 应用（EDAS SDK）

介绍如何使用 EDAS-SDK 快速开发 HSF 应用，完成服务注册与发现。

下载 Demo 工程

您可以按照本文的步骤一步步搭建出最终的工程，也可以选择直接下载本文对应的示例工程，或者使用 Git 下载: 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/hsf-ali-tomcat，里面包含 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，内容如下：

```
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。

实现服务接口

可以参考ItemServiceImpl.java文件中的示例：

```
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;
}
}
```

服务提供者配置

上述示例主要实现了 com.alibaba.edas.carshop.itemcenter.ItemService，并在两个方法中返回了一个 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。具体内容如下：

1. 在pom.xml中添加如下 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 Spring 的依赖 -->
<dependency>
<groupId>com.alibaba.edas.carshop</groupId>
<artifactId>itemcenter-api</artifactId>
<version>1.0.0-SNAPSHOT</version>
</dependency>
<!-- 添加服务接口的依赖 -->
<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>
```

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.ItemServiceImpl" />
<!-- 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>
```

服务创建及发布存在一定的限制：

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
一个 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**，在输入框中添加 -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。

1. Spring 的配置文件使用标签<hsf:consumer/>定义好一个 Bean。

2. 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。

配置服务属性

与生产者一样，消费者的服务属性配置分为 Maven 依赖配置与 Spring 的配置。

在pom.xml文件中添加 Maven 依赖。

Maven 依赖配置与生产者相同，详情请参见开发服务提供者的配置服务属性。

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>
```

服务消费者配置

可以参考StartListener.java文件中的示例：

```
public class StartListener implements ServletContextListener{

@Override
public void contextInitialized( ServletContextEvent sce ) {
ApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext( sce.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.getItemById( 1111 ) );
```

```
// 调用服务 ItemService 的 getItemByName 方法
System.out.println( itemService.getItemByName( "myname is le" ) );
……
}
}
```

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（具体请参照文档**开发工具准备**中的配置 Eclipse 开发环境或者 IDEA 开发环境。

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

在搜索条件里输入 IP 地址，单击**搜索**即可查询该 IP 地址的机器提供了哪些服务。

在搜索条件里输入服务名或服务分组，即可查询哪些 IP 地址提供了这个服务。

调用者列表页

在搜索条件里输入 IP 地址，单击**搜索**即可查询该 IP 地址的机器调用了哪些服务。

在搜索条件里输入服务名或服务分组，即可查询哪些 IP 地址调用了这个服务。

部署到 EDAS

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

集群类型可以在 ECS 集群和 Kubernetes 集群中任意选择，应用运行时环境需要选择 EDAS-Container。

部署途径可以选择 WEB 控制台或使用工具部署，详情请参见部署应用概述。

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

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>
```

执行 mvn clean package 将本地的程序打成 WAR 包。

使用 Pandora Boot 开发应用

Pandora Boot 概述

Pandora Boot 是在 Pandora 的基础之上，发展出的更轻量使用 Pandora 的方式。

> 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 EDAS 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。

配置 EDAS 的私服地址和轻量配置中心

使用 Pandora Boot 开发 HSF 应用前，需要先配置 EDAS 的私服地址和轻量配置中心。

> 目前 Spring Cloud for Aliware 的第三方包只发布在 EDAS 的私服中，所以需要在 Maven 中配置 EDAS 的私服地址。
>
> 本地开发调试时，需要启动轻量级配置中心。轻量级配置中心包含了 EDAS 服务发现和配置管理功能的轻量版。

在 Maven 中配置 EDAS 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 EDAS 私服地址。

在 Maven 所使用的配置文件（一般为~/.m2/settings.xml）中添加 EDAS 的私服配置。配置示例如下：

```
<profiles>
<profile>
<id>nexus</id>
<repositories>
<repository>
<id>central</id>
<url>http://repo1.maven.org/maven2</url>
<releases>
<enabled>true</enabled>
</releases>
<snapshots>
<enabled>true</enabled>
</snapshots>
</repository>
</repositories>
<pluginRepositories>
<pluginRepository>
<id>central</id>
<url>http://repo1.maven.org/maven2</url>
<releases>
<enabled>true</enabled>
</releases>
<snapshots>
<enabled>true</enabled>
</snapshots>
</pluginRepository>
</pluginRepositories>
</profile>
<profile>
<id>edas.oss.repo</id>
<repositories>
<repository>
<id>edas-oss-central</id>
<name>taobao mirror central</name>
<url>http://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>http://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>
```

在命令行执行命令mvn help:effective-settings，验证配置是否成功。

验证时，关注如下信息：

- 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。

配置轻量配置中心

配置轻量配置中心的步骤请参见配置轻量配置中心。

开发 HSF 应用（Pandora Boot）

您可以使用 Pandora Boot 开发 HSF 应用，实现服务注册发现、异步调用，并完成单元测试。相比使用 ali-tomcat 部署 HSF 的 war 包，Pandora Boot 部署的是 jar 包。直接将 HSF 应用打包成 FatJar，这更加符合微服务的风格，不需要依赖外置的 ali-tomcat 也使得应用的部署更加灵活。Pandora Boot 可以认为是 Spring Boot 的增强。

前提条件

在开发应用前，您已经完成以下工作：

- 在 Maven 中配置 EDAS 私服地址

- 启动轻量级配置及注册中心

服务注册与发现

介绍如何使用 Pandora Boot 开发应用（包括服务提供者和服务消费者）并实现服务注册与发现。

注意：严禁在应用启动时调用HSF远程服务，否则会导致启动失败。

Demo 源码下载：https://github.com/aliyun/alibabacloud-microservice-demo/tree/master/microservice-doc-demo/hsf-pandora-boot

使用 git 克隆整个项目，并在 microservice-doc-demo/hsf-pandora-boot 文件夹内可以找到本文使用的示例工程。

创建服务提供者

创建一个 Maven 工程，命名为hsf-pandora-boot-provider。

在 pom.xml 中引入需要的依赖。

```
<properties>
<java.version>1.8</java.version>
<spring-boot.version>2.1.6.RELEASE</spring-boot.version>
<pandora-boot.version>2019-06-stable</pandora-boot.version>
</properties>

<dependencies>
<dependency>
<groupId>com.alibaba.boot</groupId>
<artifactId>pandora-hsf-spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>${spring-boot.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
<dependency>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-starter-bom</artifactId>
<version>${pandora-boot.version}</version>
<type>pom</type>
```

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

<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.7.0</version>
<configuration>
<source>1.8</source>
<target>1.8</target>
</configuration>
</plugin>
<plugin>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.11.8</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

虽然 HSF 服务框架并不依赖于 Web 环境，但是 EDAS 管理应用的生命周期过程中需要使用到 Web 相关的特性，所以需要添加 spring-boot-starter-web 的依赖。

另外引入了 pandora-hsf-spring-boot-starter ，它会帮我们实现 HSF 配置的自动装配。

pandora-boot-maven-plugin 是 Pandora Boot 提供的 maven 打包插件，它可以将 Pandora Boot HSF 工程打包成可执行的 FatJar，并在 EDAS Container 中部署运行。

dependencyManagement 中包含了 spring-boot-dependencies 和 pandora-boot-starter-bom 两个依赖，分别负责 Spring Boot 和 Pandora Boot 相关依赖的版本管理，设置之后，您的工程不再需要设置 parent 为 spring-boot-starter-parent。

定义服务接口，创建一个接口类 com.alibaba.edas.HelloService。

HSF 服务框架基于接口进行服务通信，当接口定义好之后，生产者将通过该接口实现具体的服务并发布，消费者也是基于此接口去订阅和消费服务。

```
public interface HelloService {
String echo(String string);
```

```
}
```

接口com.alibaba.edas.HelloService提供了一个echo方法

添加服务提供者的具体实现类EchoServiceImpl，并通过注解方式发布服务。

```
@HSFProvider(serviceInterface = HelloService.class, serviceVersion = "1.0.0")
public class HelloServiceImpl implements HelloService {
@Override
public String echo(String string) {
return string;
}
}
```

在 HSF 应用中，接口名和服务版本才能唯一确定一个服务，所以在注解HSFProvider中的需要添加接口名com.alibaba.edas.HelloService和服务版本1.0.0。

说明：

- 注解中的配置拥有最高优先级。
- 如果在注解中没有配置，服务发布时会优先在resources/application.properties文件中查找这些属性的全局配置。
- 如果注解和resources/application.properties文件中都没有配置，则会使用注解中的默认值。

在resources目录下的application.properties文件中配置应用名和监听端口号。

```
spring.application.name=hsf-pandora-boot-provider
server.port=8081

spring.hsf.version=1.0.0
spring.hsf.timeout=3000
```

最佳实践: 建议将**服务版本**和**服务超时**都统一配置在application.properties中。

添加服务启动的 main 函数入口。

```
@SpringBootApplication
public class HSFProviderApplication {

public static void main(String[] args) {
// 启动 Pandora Boot 用于加载 Pandora 容器
PandoraBootstrap.run(args);
SpringApplication.run(HSFProviderApplication.class, args);
// 标记服务启动完成,并设置线程 wait。防止业务代码运行完毕退出后，导致容器退出。
PandoraBootstrap.markStartupAndWait();
}
}
```

创建服务消费者

本示例中，将创建一个服务消费者，通过HSFConsumer所提供的 API 接口去调用服务提供者。

创建一个 Maven 工程，命名为hsf-pandora-boot-consumer。

在pom.xml中引入需要的依赖内容：

说明：消费者和提供者的 Maven 依赖是完全一样的。

```
<properties>
<java.version>1.8</java.version>
<spring-boot.version>2.1.6.RELEASE</spring-boot.version>
<pandora-boot.version>2019-06-stable</pandora-boot.version>
</properties>

<dependencies>
<dependency>
<groupId>com.alibaba.boot</groupId>
<artifactId>pandora-hsf-spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>

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

<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
```

```
<version>3.7.0</version>
<configuration>
<source>1.8</source>
<target>1.8</target>
</configuration>
</plugin>
<plugin>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.11.8</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

将服务提供者所发布的 API 服务接口（包括包名）拷贝到本地，如 com.alibaba.edas.HelloService。

```
public interface HelloService {
String echo(String string);
}
```

通过注解的方式将服务消费者的实例注入到 Spring 的 Context 中。

```
@Configuration
public class HsfConfig {

@HSFConsumer(clientTimeout = 3000, serviceVersion = "1.0.0")
private EchoService echoService;

}
```

最佳实践：在HsfConfig类里配置一次@HSFConsumer，然后在多处通过@Autowired注入使用。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。只需要写一个统一的HsfConfig类，然后在其它需要使用的地方，直接通过@Autowired注入即可。

为了便于测试，使用SimpleController来暴露一个/hsf-echo/*的 HTTP 接口，/hsf-echo/*接口内部实现调用了 HSF 服务提供者。

```
@RestController
```

```
public class SimpleController {

@Autowired
private HelloService helloService;

@RequestMapping(value = "/hsf-echo/{str}", method = RequestMethod.GET)
public String echo(@PathVariable String str) {
return helloService.echo(str);
}
}
```

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

```
spring.application.name=hsf-pandora-boot-consumer
server.port=8080

spring.hsf.version=1.0.0
spring.hsf.timeout=1000
```

最佳实践: 建议将**服务版本**和**服务超时**都统一配置在application.properties中。

添加服务启动的 main 函数入口。

```
@SpringBootApplication
public class HSFConsumerApplication {

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

本地开发调试

配置轻量配置中心

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 EDAS 服务注册发现服务端的轻量版，详情请参见配置轻量配置中心。

启动应用

应用可以通过以下两种方式启动。

在 IDE 中启动

通过 VM options 配置启动参数 -Djmenv.tbsite.net={$IP}，通过 main 方法直接启动。其中 {$IP}

为轻量配置中心的 IP 地址。比如本机启动轻量配置中心，则{$IP}为127.0.0.1。

您也可以不配置 JVM 的参数，而是直接通过修改 hosts 文件将 jmenv.tbsite.net 绑定为轻量配置中心的 IP。详情请参见配置轻量级配置中心。

通过 FatJar 启动

增加 taobao-hsf.sar 依赖，这样会下载到我们需要的依赖：/.m2/com/taobao/pandora/taobao-hsf.sar/2019-06-stable/taobao-hsf.sar-2019-06-stable.jar，在后面的启动参数中依赖它。

```
<dependency>
<groupId>com.taobao.pandora</groupId>
<artifactId>taobao-hsf.sar</artifactId>
<version>2019-06-stable</version>
</dependency>
```

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。

```
<plugin>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.11.8</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
```

添加完插件后，在工程的主目录下，执行 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。

通过 Java 命令启动应用。

```
java -Djmenv.tbsite.net=127.0.0.1 -
Dpandora.location=${M2_HOME}/.m2/repository/com/taobao/pandora/taobao-hsf.sar/2019-06-
stable/taobao-hsf.sar-2019-06-stable.jar -jar hsf-pandora-boot-provider-1.0.jar
```

说明：-Dpandora.location 指定的路径必须是全路径，使用命令行启动时，必须显示指定 taobao-

hsf.sar 的位置。

结果验证

访问 consumer 所在机器的地址，可以触发 consumer 远程调用 provider

```
curl localhost:8080/hsf-echo/helloworld
helloworld
```

单元测试

Pandora Boot 的单元测试可以通过 PandoraBootRunner 启动，并与 SpringJUnit4ClassRunner 无缝集成。

我们将演示一下如何在服务提供者中进行单元测试，供大家参考。

在 Maven 中添加 Pandora Boot 和 Spring Boot 测试必要的依赖。

```
<dependency>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-test</artifactId>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
```

编写测试类的代码。

```
@RunWith(PandoraBootRunner.class)
@DelegateTo(SpringJUnit4ClassRunner.class)
// 加载测试需要的类，一定要加入 Spring Boot 的启动类，其次需要加入本类。
@SpringBootTest(classes = {HSFProviderApplication.class, HelloServiceTest.class })
@Component
public class HelloServiceTest {

/**
* 当使用 @HSFConsumer 时，一定要在 @SpringBootTest 类加载中，加载本类，通过本类来注入对象，否则
当做泛化时，会出现类转换异常。
*/
@HSFConsumer(generic = true)
HelloService helloService;

//普通的调用
@Test
public void testInvoke() {
```

```
TestCase.assertEquals("hello world", helloService.echo("hello world"));
}
//泛化调用
@Test
public void testGenericInvoke() {
GenericService service = (GenericService) helloService;
Object result = service.$invoke("echo", new String[] {"java.lang.String"}, new Object[] {"hello world"});
TestCase.assertEquals("hello world", result);
}
//返回值 Mock
@Test
public void testMock() {
HelloService mock = Mockito.mock(HelloService.class, AdditionalAnswers.delegatesTo(helloService));
Mockito.when(mock.echo("")).thenReturn("beta");
TestCase.assertEquals("beta", mock.echo(""));
}
}
```

开发 RESTful 应用（不推荐）

你可以在 HSF 框架中开发 RESTful 应用，并实现服务注册与发现、全链路追踪。不过，EDAS 已经支持原生 Spring Cloud 框架的应用，新用户不推荐使用这种开发方式。

原生 Spring Cloud 框架下的服务开发请参考 快速开始。

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。

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。

创建一个 RESTful 应用工程，命名为sc-vip-server。

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>
```

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();
}
}
```

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;
}
}
```

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
```

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。

创建一个 RESTful 应用工程，命名为sc-vip-client。

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>
```

因为要演示 FeignClient 的使用，所以与sc-vip-server（服务提供者）相比，pom.xml文件中的依赖增加了一个spring-cloud-starter-feign。

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 FeignClients。

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class ConsumerApplication {

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

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

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

}
```

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);
}
```

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, String.class);
}
@RequestMapping(value = "/echo-async-rest/{str}", method = RequestMethod.GET)
public String asyncRest(@PathVariable String str) throws Exception{
ListenableFuture<ResponseEntity<String>> future = asyncRestTemplate.
getForEntity("http://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。

配置应用名以及监听端口号。

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

本地开发调试

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 EDAS 服务注册发现服务端的轻量版，详情请参见配置轻量配置中心。

启动应用

本地应用可以通过两种方式启动。

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 EDAS 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。

如果你的轻量级配置中心与应用部署在不同的机器上，还需进行 hosts 绑定，详情请参见轻量级配置中心。

FatJar 启动

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>
```

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。

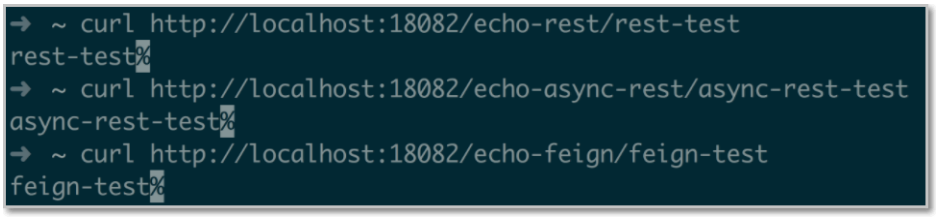
通过 Java 命令启动应用。

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

注意 ： -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。

演示

启动服务，分别进行调用，可以看到调用都成功了。



常见问题

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，具体详情参见此 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 EDAS 前恢复到默认排除 SAR 包的配置。

全链路跟踪

您可以在本地对您的 RESTful 应用经过简单的修改接入到 EDAS 的 EagleEye，从而实现全链路跟踪。

为了节约您的开发成本和提升您的开发效率，EDAS 提供了全链路跟踪的组件 EagleEye。您只需在代码中配置 EagleEye 埋点，即可直接使用 EDAS 的全链路跟踪功能，无需关心日志采集、分析、存储等过程。

Demo 源码下载：service1、service2

接入 EagleEye

在 Maven 中配置 EDAS 的私服地址

目前 Pandora Boot Starter 相关的包只发布在 EDAS 的私服中，所以需要在 Maven（要求 3.x 及后续版本）配置文件settings.xml中配置 EDAS 的私服地址，详情请参见- 在 Maven 中配置 EDAS 私服地址。

修改代码

RESTful 应用接入 EDAS 的 EagleEye 很简单，只需要完成以下三步。

在pom.xml文件中加入如下公共配置。

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

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

在 main 函数中添加修改配置。

假设修改之前的 main 函数内容如下：

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

则修改为：

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

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 中添加其它 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>
```

完成上述三处修改后，您无需搭建任何采集分析系统，即可直接使用 EDAS 的全链路跟踪功能。

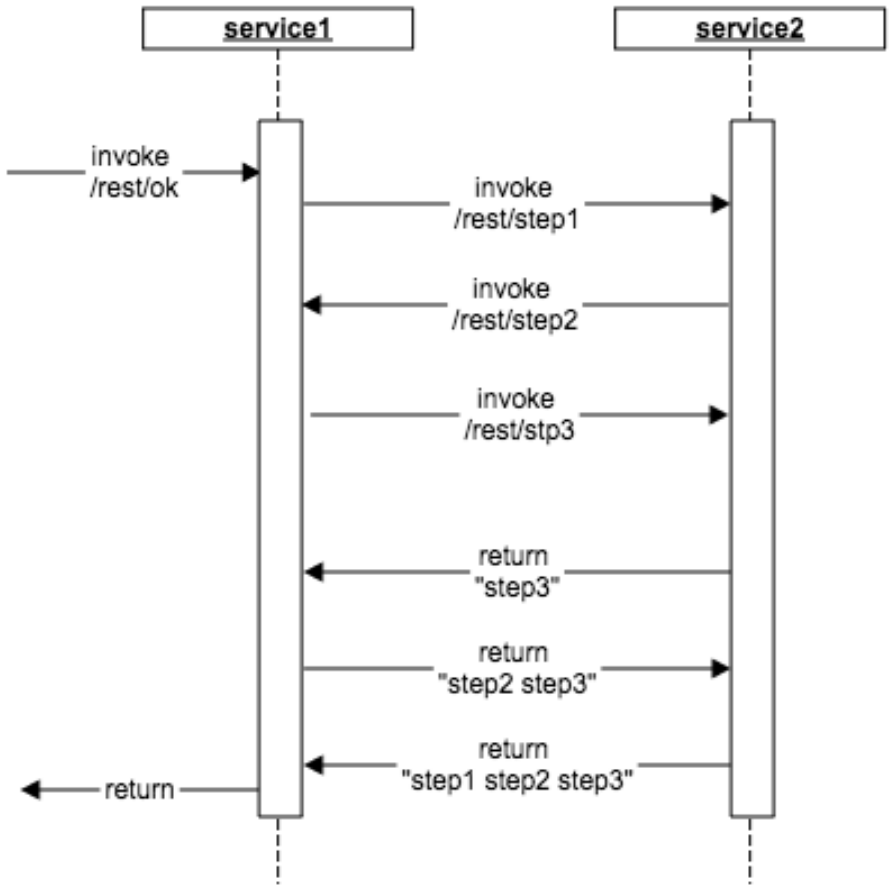
全链路跟踪示例

源码

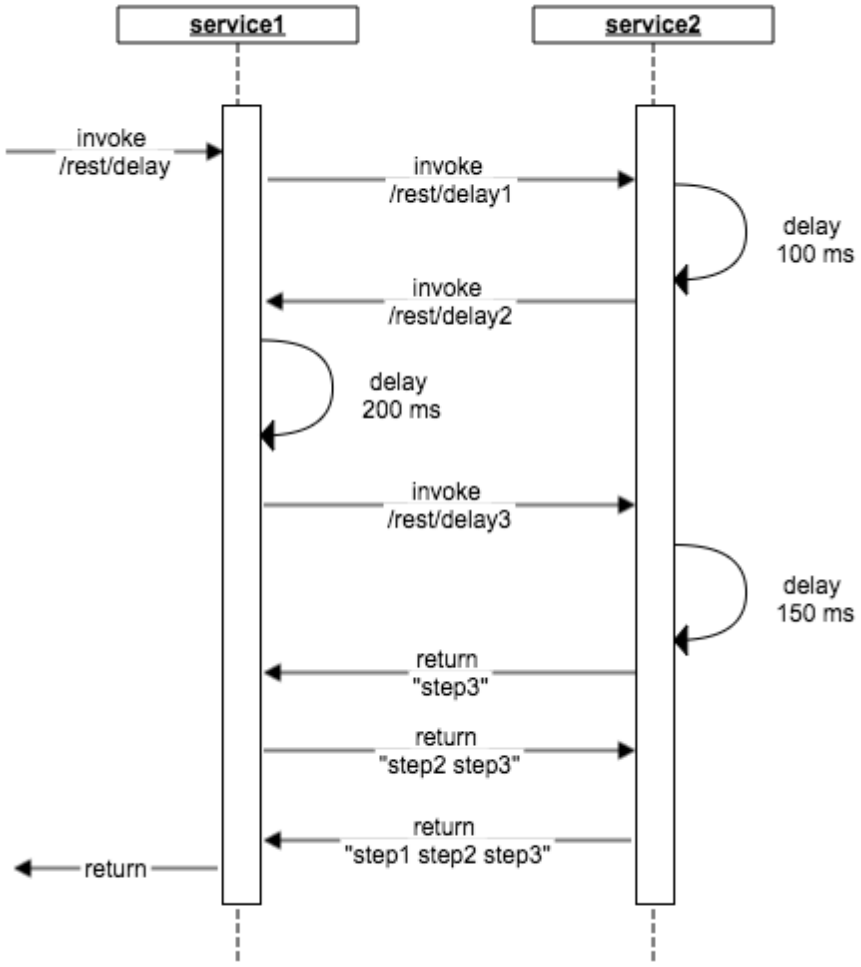
为了演示如何使用全链路跟踪功能 ，这里提供两个应用 Demo：service1 和 service2。

service1 用作入口的服务，提供了三个场景演示的入口：

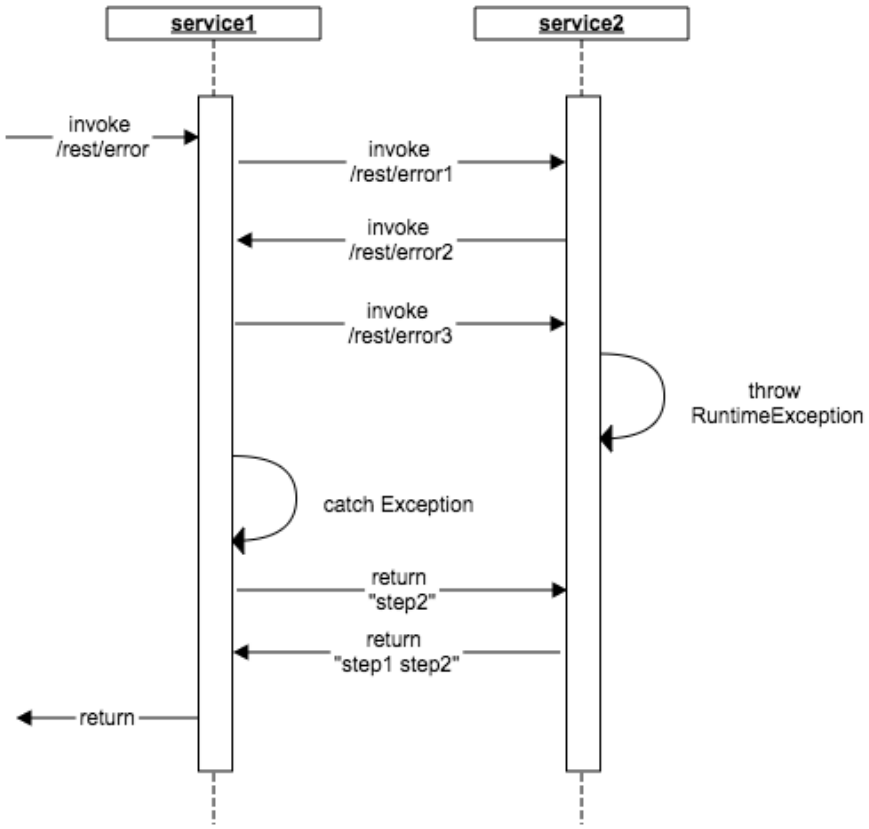
/rest/ok，对应正常的调用



/rest/delay，对应延迟较大的调用



/rest/error，对应异常出错的调用



部署应用

EagleEye 的采集和分析功能都搭建在 EDAS 上，为了演示调用链查看功能，我们首先将 service1 和 service2 这两个应用部署到 EDAS 中。详情请参见应用部署概述。

查看调用链

部署完毕之后，为了能够查看调用链的信息，我们还需要调用 service1 三个场景演示的入口对应的方法。您可以通过执行 curl http://{$ip:$port}/rest/ok 这种简单的方式来调用。也可以使用 postman 等工具或者直接在浏览器中调用。

为了便于观察，建议使用脚本等方式多调用几次。然后按以下步骤查看调用链。

登录 EDAS 控制台，进入刚刚部署的应用中。

在应用详情页面左侧的导航栏中选择**应用监控** > **服务监控**。

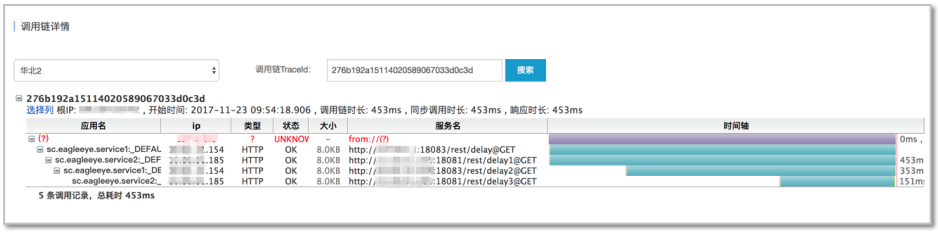
在服务监控页面单击**提供的 RPC 服务**页签，然后单击**查看调用链**。

更详细的使用信息，请参考服务监控。

正常的调用链详情

从图中可以看出服务经过了哪几次调用，并且可以看到 step1、step2、step3 的耗时分别是 2ms、1ms、0ms。

延迟较大的调用链详情



从图中可以看到 delay1、delay2、delay3 的耗时分别是 453ms、353ms、151ms，

将鼠标停留在 delay3 这个调用段，还可以看到更多详细的调用链的信息。其中服务端处理请求花费了 150ms，客户端在服务端处理完请求后的 1ms 收到了响应。

异常出错的调用链详情

从图中我们可以很清晰地看到，出错的请求为 /rest/error3 ，极大地方便了我们对问题进行定位。

其他客户端的演示

同时，在 service1 的 /echo-rest/{str} 、 /echo-async-rest/{str} 、 /echo-feign/{str} 这三个 URI 中，分别演示了 EagleEye 对 RestTemplate、 AsyncRestTemplate 、 FeignClient 的自动埋点支持，您可以在调用后，通过同样的方式查看着这三者的调用链信息。

常见问题

埋点支持

目前 EDAS 的 EagleEye 已经支持自动对 RestTemplate、AsyncRestTemplate、FeignClient 调用的请求自动进行跟踪。**后续我们将接入更多的组件的自动埋点。**

AsyncRestTemplate

由于 AsyncRestTemplate 需要在类实例化的阶段进行埋点支持的修改，所以如果需要使用全链路跟踪功能，需要按名称注入对象，eagleEyeAsyncRestTemplate，此对象默认添加了服务发现的支持。

```
@Autowired
private AsyncRestTemplate eagleEyeAsyncRestTemplate;
```

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 pandora-boot-maven-plugin 的打包插件。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。

扩展

更多全链路跟踪以及 EagleEye 的信息，请参考 Spring Cloud 接入 EDAS 之全链路跟踪。

将使用 Dubbo 开发的应用迁移到 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 EDAS 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。

原生 Dubbo 框架下的应用开发请参见使用 Spring Boot 开发 Dubbo 应用。

说明：本文主要介绍如何修改配置，应用开发过程不再详细描述。

如果需要，可以下载 Dubbo 转换为 HSF 的 Demo。

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。

注意：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();
}
}
```

容器版本说明

版本号	发布时间	构建包序号	Pandora 版本	修改内容
3.5.6	2019-9-12	57	3.5.6	1. 更新config-client插件，修复多租户场景未读缓存的问题。 2. 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。 3. 升级所有用到fastjson 的插件到sec06安全版本。
3.5.5	2019-8-15	56	3.5.5	升级 HSF 插件中 Dubbo 依赖的 hessian-lite。
3.5.4	2019-7-18	55	3.5.4	1. 更新HSF插件,修复RPCContext 不能清除的问题。 2. 更新tddl-driver插件,对于prepareCall操作遗漏了

				TXC_XID HINT 信息的拼装处理。 3. 更新 metrics 插件，修复 BinAppender 对象内存使用较多的问题。 4. 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	2019-3-13	54	3.5.3	1. 升级了 HSF 与

				Eagle Eye 插件版本，支持全链路灰度与 HSF 灰度流量控制。 2. 升级了 ONS-CLIENT 插件版本到 1.8.0-Eagle Eye。
3.5.2	2019-1-26	53	3.5.2	1. 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。 2. 升级 Ali-

				Tomcat 版本到 7.0.92。 3. 重新添加 ONS-CLIENT 插件并升级版本至 1.7.9-Eagle Eye。 4. 更新其它 Pandora 插件版本。
3.5.1	2018-11-28	52	3.5.0	Docker 镜像 的 JDK 升级到 JDK 1.8.0_191。
3.5.0	2018-9-10	51	3.5.0	1. 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。 2. 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。 3. 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	2018-8-1	50	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。
3.4.6	2018-7-5	49	3.4.6	1. 升级 HSF 到 2.2.6.1 版本。 2. 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。

				3. 升级 config-client 到 1.9.6 版本，支持动态调整最大注册数。 4. 升级 Sentinel 到 2.12.12-edas 版本，支持 Spring Boot 2。
3.4.5	2018-6-14	48	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	2018-5-18	47	3.4.4	1. HSF 服务端异步处理时+本地调用时，tim

				eout 值为 0 导致超时异常。 2. EDAS 在使用 Dubbo 时，RpcContext 缺少对端等IP属性。 3. 支持 Dubbo 的 service 标签在 AOP 场景使用。 4. HSF 泛化调用时，如果 Bool 值在 Map 里面，则不支持。
3.4.3	2018-4-24	46	3.4.3	1. Diam

				ond 升级到 3.8.8。 2. 修复不断打印证书找不到问题，增加安全能力。 3. EDAS-Assist 升级到 2.0 4. 优化端口可用性检测逻辑，Fastjson 版本升级成 1.2.48。
3.4.1	2018-3-15	44	3.4.1	1. 升级 hsf-plugin，支持 dubboX。 2. 升级

				diamond-client和configcenter-client。 3. 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	2018-3-7	43	3.4.0	1. 升级edas-assist，动态设置CSP端口及解决检查可用端口慢问题。 2. 新增ConfigCent

				er 租户版本。 3. 升级 configclient，实现内外客户端统一，支持 CS2.0、CS3.0 服务端。
3.3.9	2018-1-17	42	3.3.9	1. 升级 HSF 版本，解决了 ZooKeeper 阻塞的问题。 2. 升级 Sentinel 新增系统保护（引入控制台推送规则才能生效

				。 3. 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	1. 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。 2. 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	1. 升级 Diamond 到最新版本，兼容 ACM 。 2. 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。 3. 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。 4. Eagle Eye 支持健康检查，附带 alimetric、tomcat monitor。 5. 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	1. 新增自动注册应用的功

				能，默认关闭。 2. 解决HSF文件句柄占用问题 3. Sentinel 支持HSF2.2,4，增强Pandora QoS命令。
3.3.2	2017-10-18	36	3.3.2	1. 修复HSF持有hsf.lock 句柄问题。 2. 增加Redis埋点。 3. 升级tddl-driver，线上做全链路压测使用。 4. 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。

备注：

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。点击该列对应的版本 Pandora 版本号数值可下载该版本的 Pandora 归档包。

EDAS SDK 版本说明

SDK 版本	EDAS Container 最低版本要求	描述
1.8.2	3.5.0	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，比如 filter 内部特殊业务异常逻辑处理的场景。

将应用从 HSF 架构迁移到 Dubbo（Ali-Tomcat）

本文介绍如何将使用 Ali-Tomcat 开发的应用的框架从 HSF 迁移到 Dubbo。

迁移方案

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

两步迁移

i. 将 HSF+EDAS 注册中心迁移到 Dubbo+EDAS 注册中心。
ii. 将 EDAS 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。

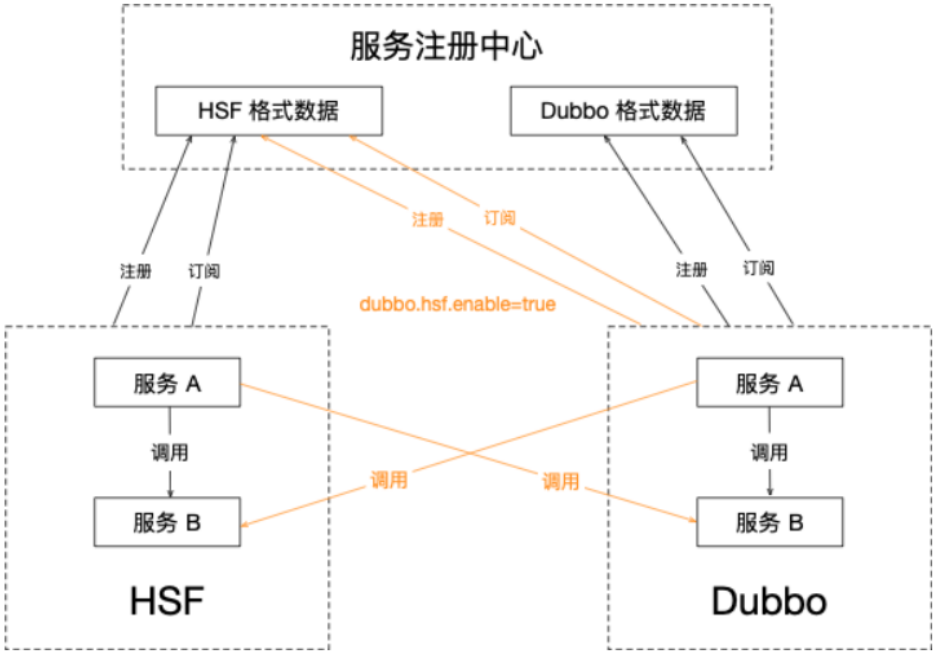
直接迁移

将 HSF+EDAS 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：

- 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
- 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。

迁移主要是修改 3 个 XML 文件。

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
```

步骤一：在 pom.xml 中增加 Dubbo 相关依赖

老版本的 HSF 采用依赖的是比较老的 Spring 版本 2.5.6，建议直接升级到 4.x 版本。

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>
```

增加 Dubbo 相关依赖。

edas-dubbo-extension：主要解决的是将 Dubbo 服务注册到 EDAS 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>
```

步骤二：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.springframework.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 consumer 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="HSF" 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 EDAS 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，已保证 HSF 客户端可以发现该服务服务。
- 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。

步骤三：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>
```

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。

验证服务注册

请按照以下步骤验证服务提供者是否能正常注册到 EDAS 轻量级配置及注册中心。

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
```

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

执行mvn clean package命令将，将edas-hsf-demo-provider-war打成 WAR 包。

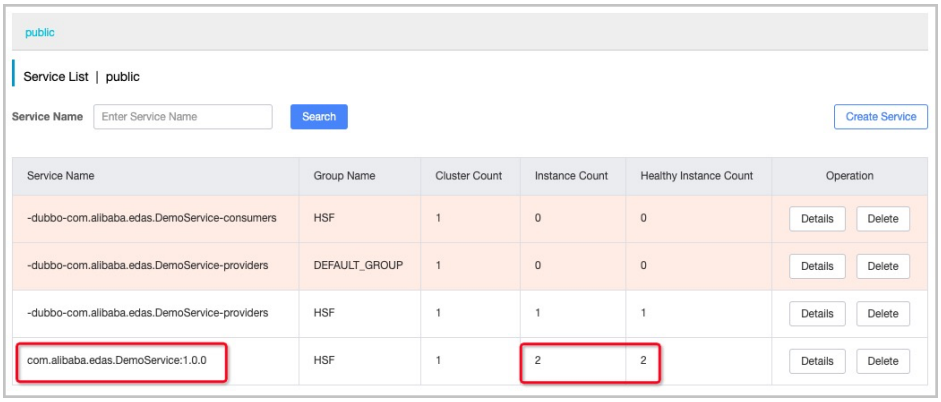
成功之后会在edas-hsf-demo-provider-war和edas-dubbo-demo-provider-war这两个工程的target 目录下，找到两个 WAR 包，分别对应 HSF 和 Dubbo 的服务。

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

注意：启动两个 Tomcat 的端口可能会冲突，假设使用命令行方式部署，在 Tomcat 的conf/server.xml文件中搜索8005和8080端口，修改为不冲突的两个端口。

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。



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。

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.getRequiredWebApplicationContext(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);
}
}
```

执行mvn clean package命令，将edas-hsf-demo-consumer-war打包为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。

登录轻量级配置及注册中心控制台，发现 HSF 服务消费者并没有数据在控制台里面展现，这是正常现象。

启动 Ali-Tomcat 后，访问如下的 URL：

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

观察edas-hsf-demo-consumer.war所对应的 Ali-Tomcat 的标准输出，发现以下内容：

```
Received: Hello hsf, response from hsf provider: /30.5.124.128:62385
Received: Hello hsf, response from dubbo provider: 30.5.124.128:20880
Received: Hello hsf, response from hsf provider: /30.5.124.128:62385
Received: Hello hsf, response from hsf provider: /30.5.124.128:62385
Received: Hello hsf, response from dubbo provider: 30.5.124.128:20880
Received: Hello hsf, response from hsf provider: /30.5.124.128:62385
Received: Hello hsf, response from dubbo provider: 30.5.124.128:20880
Received: Hello hsf, response from dubbo provider: 30.5.124.128:20880
Received: Hello hsf, response from hsf provider: /30.5.124.128:62385
Received: Hello hsf, response from hsf provider: /30.5.124.128:62385
Received: Hello hsf, response from hsf provider: /30.5.124.128:62385
```

这说明 HSF 客户端同时调用到了 Dubbo 和 HSF 提供的服务。

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。

步骤一：在 pom.xml 中增加 Dubbo 相关依赖

和服务提供者迁移的操作一致，主要添加dubbo、dubbo-edas-extension依赖，删除edas-sdk依赖。详情请参见在 pom.xml 中增加 Dubbo 相关依赖(服务端)。

步骤二：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>
```

需要修改为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.springframework.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="HSF" version="1.0.0"
check="false"/>
</beans>
```

说明：

1. Dubbo 的注册中心地址需要以为edas://开头
2. 配置<dubbo:service>标签的时候，需要指定group和version，且group和version务必要和服务提供者保持一致。默认的情况下，group是 **HSF**，version是 **1.0.0**。
3. 需要添加check="false"配置。这个参数表示服务消费者在启动的时候如果没有服务端地址不会立刻失败，因为服务提供者地址推送是异步的，地址可能会比启动流程晚一些再推送到服务消费者。
4. 需要增加<dubbo:parameter key="hsf.enable" value="true"/>配置，表明服务消费者订阅服务提供者的数据。

步骤三：在 web.xml 文件中将 hsf-consumer-beans.xml 替换为 dubbo-comsumer-beans.xml

只需要把hsf-consumer-beans.xml替换为dubbo-consumer-beans.xml即可。

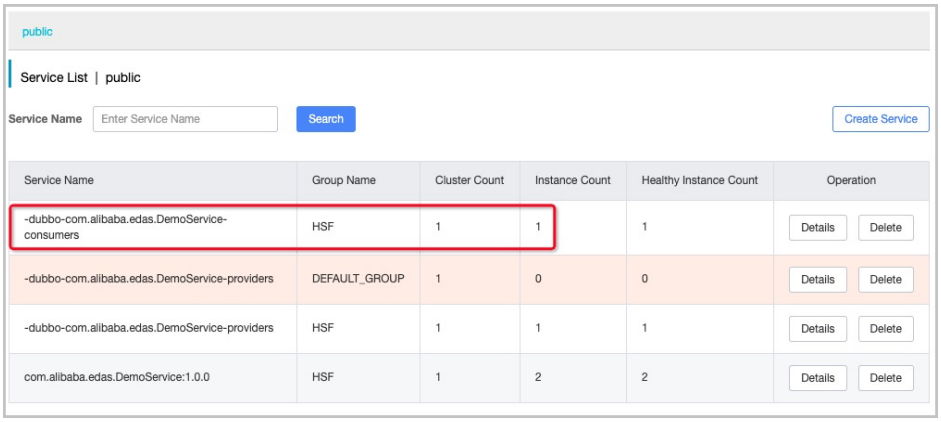
本地验证

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。操作步骤如下：

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

注意避免端口冲突。

观察轻量级配置及注册中心的控制台，发现有一个 Dubbo 服务消费者注册上来。



使用curl命令访问如下链接：

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

观察客户端 Apache Tomcat 的标准输出，发现如下内容：

```
Received: Hello dubbo, response from dubbo provider: 30.5.124.128:20880
Received: Hello dubbo, response from dubbo provider: 30.5.124.128:20880
Received: Hello dubbo, response from hsf provider: /30.5.124.128:12202
Received: Hello dubbo, response from hsf provider: /30.5.124.128:12202
Received: Hello dubbo, response from dubbo provider: 30.5.124.128:20880
Received: Hello dubbo, response from hsf provider: /30.5.124.128:12202
Received: Hello dubbo, response from hsf provider: /30.5.124.128:12202
Received: Hello dubbo, response from dubbo provider: 30.5.124.128:20880
Received: Hello dubbo, response from dubbo provider: 30.5.124.128:20880
Received: Hello dubbo, response from hsf provider: /30.5.124.128:12202
Received: Hello dubbo, response from hsf provider: /30.5.124.128:12202
Received: Hello dubbo, response from dubbo provider: 30.5.124.128:20880
Received: Hello dubbo, response from dubbo provider: 30.5.124.128:20880
```

这说明 Dubbo 服务消费者消费了 HSF 和 Dubbo 的服务。

将应用部署到 EDAS 并验证

本文采用 ECS 集群方式部署，在 EDAS 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)

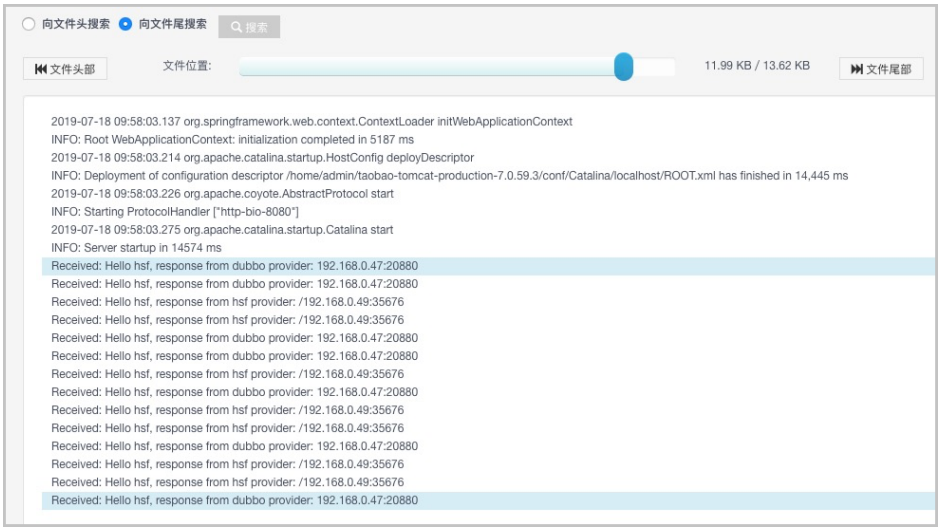
分别将 4 个 WAR 包部署到 4 个应用中。详情请参见应用部署概述。

为了确保能在本地通过 HTTP 访问到 EDAS 的服务消费者，请确保服务消费者的8080端口能够被公网访问。

执行若干次以下命令，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://39.106.76.128:8080/index.htm
```

在edas-hsf-demo-consumer应用的 Ali-Tomcat 的标准输出中，如/home/admin/taobao-tomcat-production-7.0.59.3/logs/catalina.out，观察到以下日志：

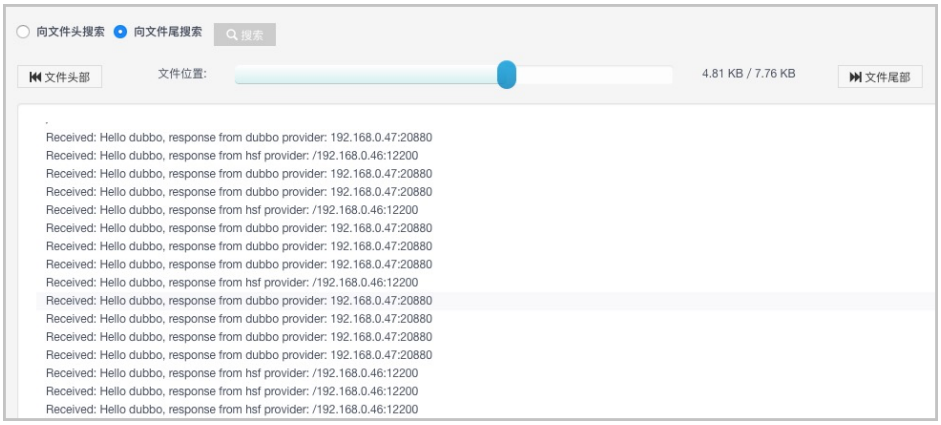


这说明 HSF 服务消费者消费到了 HSF 和 Dubbo 的服务。

执行若干次以下命令，测试edas-dubbo-demo-consumer能否调用到 HSF 和 Dubbo 的 Provider。

```
curl http://39.106.74.16:8080/index.htm
```

在edas-dubbo-demo-consumer应用 Apache Tomcat 的标准输出中，例如/home/admin/apache-tomcat-7.0.91/logs/catalina.out，观察到以下日志。



这说明 Dubbo 服务消费者消费到了 HSF 和 Dubbo 的服务。

FAQ

1. Dubbo 服务消费者启动后，提示找不到服务提供者地址。

异常信息

```
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.service.xxxxxxxxxxService:1.0.0 from the url edas://127.0.0.1:8080/org.apache.dubbo.registry.RegistryService?application=xxxx-flow-center-bj&dubbo=2.0.2&group=HSF&interface=com.xxxx.xxxxxx.service.xxxxxxxxxxService&lazy=false&methods=queryConfigs,getConfig,saveConfig&pid=11596&register.ip=xxx.xx.xx.xxx&release=2.7.3&revision=1.0.1-SNAPSHOT&side=consumer&sticky=false&timeout=2000&timestamp=15642424211194&version=1.0.0 to the consumer xxx.xx.xx.xxx use dubbo version 2.7.3
```

可能的原因

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。

解决办法

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" version="1.0.0" check="false"/>
```

这个参数表示 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。

2. HSF 服务消费者调用 Dubbo 服务异常

异常信息

```
2019-07-28 23:07:38.005 [WARN ] [cf67433d1e7a44412a518bd190100d176-node401] [NettyServerWorker-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=AppResponse
[value=FlowControlDto(postWeightDtoHashMap={614215325=PostWeightDto(postId=614215325, weight=1.0,
postSourceType=null)}), exception=null]], send bad_response info instead, cause: 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:195)
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)
...
```

可能的原因

Dubbo 升级到 2.7 后，HSF 对 Dubbo 的协议兼容性出现问题。

解决办法

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

3. Dubbo 服务消费者调用 HSF 服务提供者失败

异常信息

```
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 [192.168.0.48]
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$OutputCallback.operationComplete(ProviderProcessor.java:15
5)
com.taobao.hsf.remoting.provider.ProviderProcessor$OutputCallback.operationComplete(ProviderProcessor.java:13
0)
com.taobao.hsf.util.concurrent.AbstractListener.run(AbstractListener.java:18)
com.taobao.hsf.invocation.AbstractContextAwareRPCCallback.access$001(AbstractContextAwareRPCCallback.java:1
```

```
2)
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)
```

可能的原因

HSF 开启了调用鉴权，而 Dubbo 暂时不支持鉴权。

解决办法

在 HSF 服务端增加参数-DneedAuth=false，关闭调用鉴权。

4. Dubbo 服务消费者调用 HSF 服务提供者失败

异常信息

```
2019-08-02 17:17:15.187 [WARN ] [cf67433d1e7a44412a518bd190100d176-node401] [NettyClientWorker-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:98)
at java.lang.reflect.Field.setByte(Field.java:838)
at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer$ByteFieldDeserializer.deserialize(JavaDeserializer.java:452)
at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.java:276)
at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.java:203)
at com.alibaba.com.xxxxxx.hessian.io.SerializerFactory.readObject(SerializerFactory.java:532)
at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObjectInstance(Hessian2Input.java:2820)
at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2743)
at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2278)
at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2080)
at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObject(Hessian2Input.java:2074)
at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer$ObjectFieldDeserializer.deserialize(JavaDeserializer.java:406)
at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.java:276)
at com.alibaba.com.xxxxxx.hessian.io.JavaDeserializer.readObject(JavaDeserializer.java:203)
at com.alibaba.com.xxxxxx.hessian.io.SerializerFactory.readObject(SerializerFactory.java:532)
at com.alibaba.com.xxxxxx.hessian.io.Hessian2Input.readObjectInstance(Hessian2Input.java: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.deserialize(JavaDeserializer.java:406)
```

```
...
```

可能的原因

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。

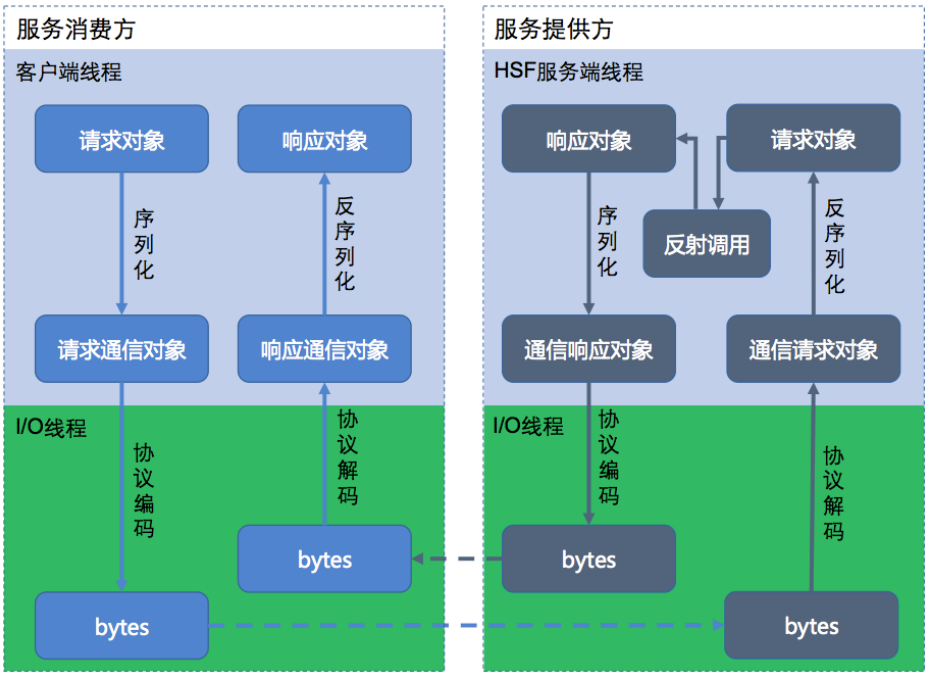
解决办法

升级 HSF 服务端的 EDAS-Container 的版本到 v3.5.5。

一次调用过程

HSF 一次调用过程会从服务消费方发起，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回来，最终返回给用户。这个过程会涉及到多个线程交互，也会涉及到 HSF 中的不同领域对象。

HSF 一次调用过程如下图所示：



作为服务消费方，在客户端线程（比如：tomcat 线程）中首先会将用户的参数也就是请求对象进行序列化，将序列化之后的内容放置到请求通信对象中，请求通信对象对应的是 HSF 协议，它包含诸如请求 id 等多个与请求对象无关的内容。请求通信对象会提交给 I/O 线程，在 I/O 线程中完成编码，最终发送到服务提供方，此时客户端线程会等待结果返回，处于等待状态。

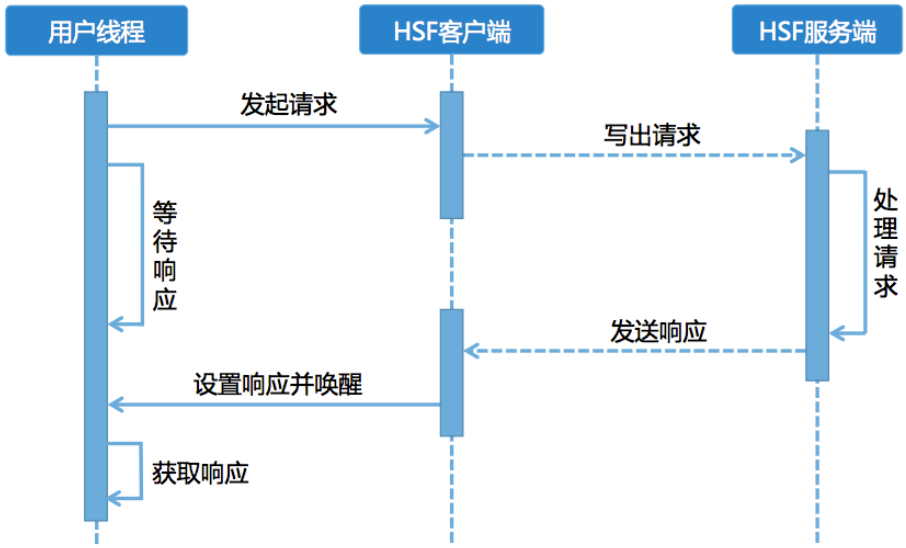
服务提供方的 I/O 线程接收到二进制内容，解码后生成通信请求对象并将其递交给 HSF 服务端线程，在 HSF 服务端线程完成反序列化还原成请求对象，然后发起反射调用，得到结果，也就是响应对象。响应对象会在 HSF服务端线程中完成序列化，并放置到通信响应对象中。HSF 服务端线程会将通信响应对象提交给 I/O 线程

，在 I/O 线程中完成编码，最终发送回服务消费方。

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，客户端线程会根据响应通信对象中的内容完成反序列化，最终拿到响应对象，一次远程调用结束。

异步调用

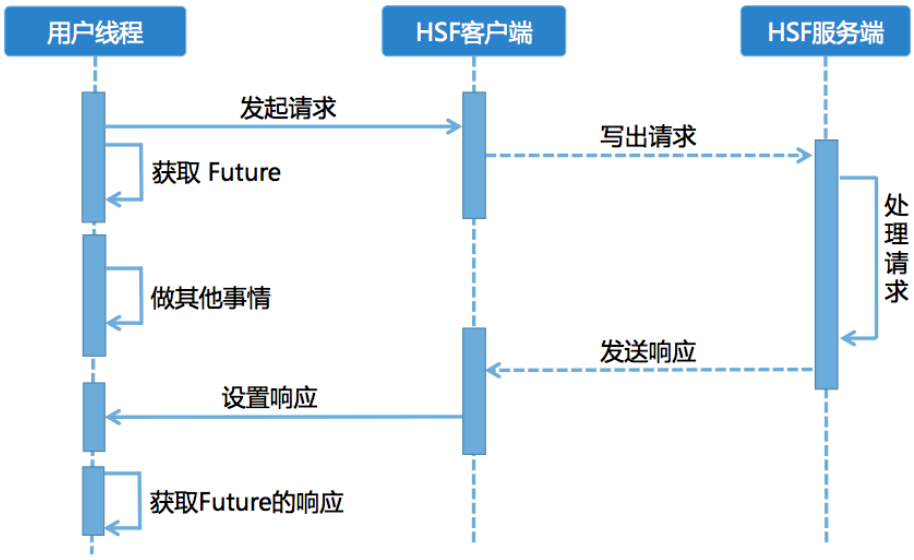
HSF 的 IO 操作都是异步的，客户端同步调用的本质是做 future.get(timeout) 操作，等待服务端的结果返回，这里的 timeout 就是客户端生效的超时时间 (默认 3000ms)。默认的同步调用时序图如下所示：



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，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，然后用户可以在任意时刻调用 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。具体配置如下：

```
HSFApiConsumerBean hsfApiConsumerBean = new HSFApiConsumerBean();
hsfApiConsumerBean.setInterfaceName("com.alibaba.middleware.hsf.guide.api.service.OrderService");
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.getObject();
// ---------------------- 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 future 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();
}
```

这里提下超时的概念，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="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。

首先是在项目中增加依赖 starter。

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

通常一个 HSF Consumer 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：

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

}
```

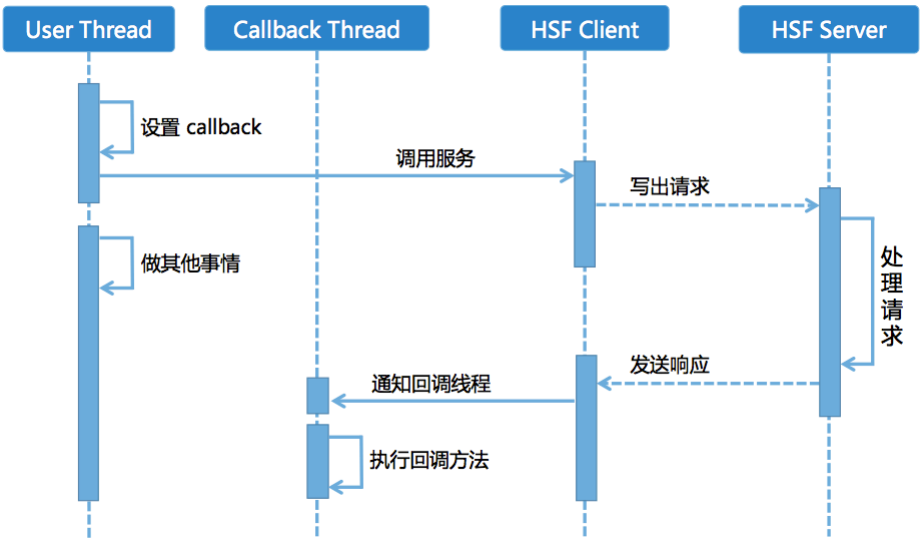
在使用时直接注入即可：

```
@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.OrderService");
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.CallbackHandler");
hsfApiConsumerBean.setAsyncallMethods(asyncallMethods);
hsfApiConsumerBean.init(true);
// [代理] 获取 HSF 代理
OrderService orderService = (OrderService) hsfApiConsumerBean.getObject();
// 可选步骤，设置上下文。CallbackHandler 中通过 api 可以获取到
CallbackInvocationContext.setContext("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.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="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.CallbackHandler</value>
</list>
</property>
</bean>
```

注解配置 HSF 接口方法为 callback 调用

将 @AsyncOn 注解配置到 callback 类上，指明作用到的接口类和方法。

```
@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 是机器 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 线程池最大的配置 -Dhsf.callback.max.poolsize

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，不要不依赖二方包，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

API 形式配置 HSF 服务

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

```
HSFApiConsumerBean hsfApiConsumerBean = new HSFApiConsumerBean();
hsfApiConsumerBean.setInterfaceName("com.alibaba.middleware.hsf.guide.api.service.OrderService");
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) hsfApiConsumerBean.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});
```

可以看到，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，以便服务端找到改方法。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。

调用传方法签名和参数说明

方法没有入参，可以只传 methodName: service.$invoke("sayHello", null, null)

方法类型有泛型的，比如 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

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

配置抛出业务异常

-Dhsf.generic.throw.exception=true（默认是 false, 把异常泛化成 map 返回）

- 本地存在异常类，由于 com.taobao.hsf.remoting.service.GenericService 上没有声明该异常，如果不是 RuntimeException 类型或其子类，则会抛出 UndeclaredThrowableException，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

调用上下文

请求上下文包括一次调用相关的属性，比如调用的地址，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 掉，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 userId（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
setTargetServerIp(String ip)	设置当前线程下一次调用的目标服务器 Ip（此 ip 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(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
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。

范例

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");
```

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。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”)	最好	一般

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.OrderService"/>
```

```
<!--[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>
```

超时配置

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，比如一些前端应用，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:

客户端配置优先于服务端，方法优先于接口

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

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。

首先是在项目中增加依赖 starter。

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

通常一个 HSF Consumer 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.ConsumerMethodSpecial(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);
}
}
```

服务端线程池配置

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，如下图所示：

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
- 线程池最大的配置: -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。

首先是在项目中增加依赖 starter。

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

然后将 @HSFProvider 配置到实现的类型上，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);
}
}
```

API 手册

在 HSF 面向用户的 API 中，最为关键的就是与创建 ProviderBean 和 ConsumerBean 相关的 API 了。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，该过程只需要做一次。此外，考虑到 HSFApiProviderBean 对象比较重，建议缓存起来。

- 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
hsfApiProviderBean.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。
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，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。

- 范例配置：

```
<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 HSFTimeOutException -->
<!-- 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。可选值有：java，hessian，hessian2，json，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 即可完成 HSF 服务的订阅。

对于一个服务，该过程只需要做一次。此外，考虑到 HSFApiConsumerBean 对象比较重，建议将该对象以及获取到的 HSF 代理都缓存起来。

> 其实在 HSF 内部 HSFApiConsumerBean 对服务的配置也是缓存起来的。也就是说，如果针对一个订阅的服务有多次配置，只有第一次的配置会生效。

- 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。
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。
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 编程方式 - 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。

- 范例配置：

```
<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 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。-->
<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.HelloWorldServiceCallbackHandler</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 javassist -->
<property name="proxyStyle" value="jdk" />
</bean>
```

JVM -D 启动配置参数

-Dhsf.server.port

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

-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 60

-Dhsf.client.localcall

打开/关闭本地优先调用。默认值是 true

-Dpandora.qos.port

指定 pandora 的监控端口，默认值为 12201。如果想要在本地启动多个 HSF Provider，需要修改此端口。

-Dhsf.http.enable

是否开启 http 端口，默认是 true。

-Dhsf.http.port

指定 HSF 暴露的http接口。默认值是 12220。如果想要在本地启动多个 HSF Provider，需要修改此端口。

-Dhsf.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。默认为 1，线上不推荐指定为 0。

-Dhsf.shuthook.wait

HSF 优雅关闭的等待时间，单位是 ms，默认是 10000。

-Dhsf.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。

-Dhsf.server.ip

多网卡情况下默认绑定第一个网卡，通过这个参数指定需要绑定的 IP

-DHsfBindHost

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，通过这个参数可以指定需要绑定的 Host，比如-DHsfBindHost=0.0.0.0将 HSF Server 端口绑定本机所有网卡；

-Dhsf.publish.interval=400

发布服务之间的时间间隔，HSF 服务会一瞬间暴露出去，如果应用启动时承受不住压力，可以配置这个参数。默认值是 400，单位 ms

-Dhsf.client.low.water.mark=32 -Dhsf.client.high.water.mark=64 -Dhsf.server.low.water.mark=32 -Dhsf.server.high.water.mark=64

- 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 > 低水位

-Dhsf.generic.remove.class=true

泛化调用拿到的结果，不输出class字段信息

-DdefaultHsfClientTimeout

全局的客户端超时配置

-Dhsf.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

使用 Cloud Toolkit 实现服务的端云互联

端云互联简介

在开发应用时，可以使用 Alibaba Cloud Toolkit 插件实现本地应用和部署在 EDAS 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。

目前集成开发环境 IDE（Integrated Development Environment）主要包括 IntelliJ IDEA 和 Eclipse。您可以在这两种环境中分别配置 Alibaba Cloud Toolkit 的端云互联功能，实现本地应用和云上应用的相互调用。详情请参见：

使用 Cloud Toolkit 实现端云互联（IntelliJ IDEA）

使用 Cloud Toolkit 事项端云互联（Eclipse）

使用限制

在 IDE 中使用 Cloud Toolkit 插件对应用进行端云互联时，开发框架和集群类型均存在以下限制。

开发框架限制

- 基于 Dubbo 框架开发的应用需满足下面的版本条件才可支持端云互联：
    - Dubbo 2.7.2 + edas-dubbo-extension 2.0.2 及以上版本。
    - Dubbo 2.7.2 + dubbo-nacos-registry 2.7.2 及以上版本。
- 基于 Spring Cloud 框架开发的应用使用 Nacos 进行配置管理时，Spring Cloud Alibaba 需满足下面的版本才可支持端云互联：
    - Spring Cloud Alibaba 0.9.0
    - Spring Cloud Alibaba 0.2.2
    - Spring Cloud Alibaba 0.1.2。

集群类型限制

容器服务 Kubernetes 集群和自建 Kubernetes 集群中的应用进行端云互联时，网关桥接 ECS 实例要选择应用所在集群内的机器。

ECS 集群和 Swarm 集群中的应用进行端云互联时，网关桥接 ECS 实例要选择应用所在 VPC 内的机器。

使用 Cloud Toolkit 实现端云互联（IntelliJ IDEA）

您可以在 IntelliJ IDEA 中使用 Cloud Toolkit 的端云互联功能实现本地和云上应用的相互调用，提升开发效率。

前提条件

在使用 Cloud Toolkit 实现端云互联前，请完成以下工作：

开发框架和集群类型对使用端云互联都有些限制，请了解使用限制。

安装 IntelliJ IDEA，请选择 2018.3 及以上版本。

说明：因 JetBrains 插件市场官方服务器在海外，如遇访问缓慢无法下载安装的，请加入文末交流群，向 Cloud Toolkit 产品运营获取离线包安装。

登录云服务器 ECS 控制台 创建一台可使用 SSH 登录的 ECS，用于建立端云互联通道。

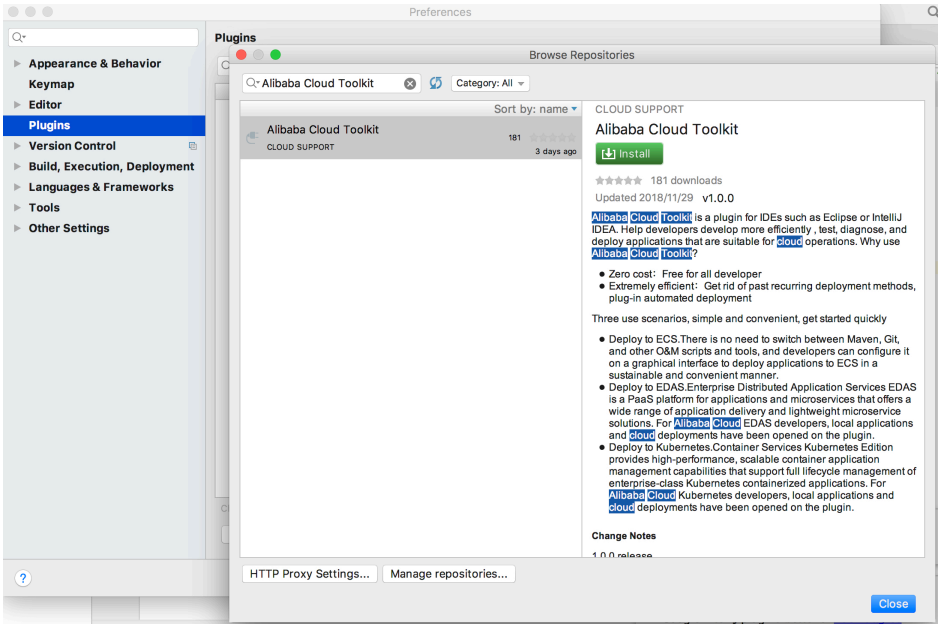
注意：SSH 通道需要使用密码方式登录，暂不支持使用密钥对登录。

步骤一：安装 Cloud Toolkit

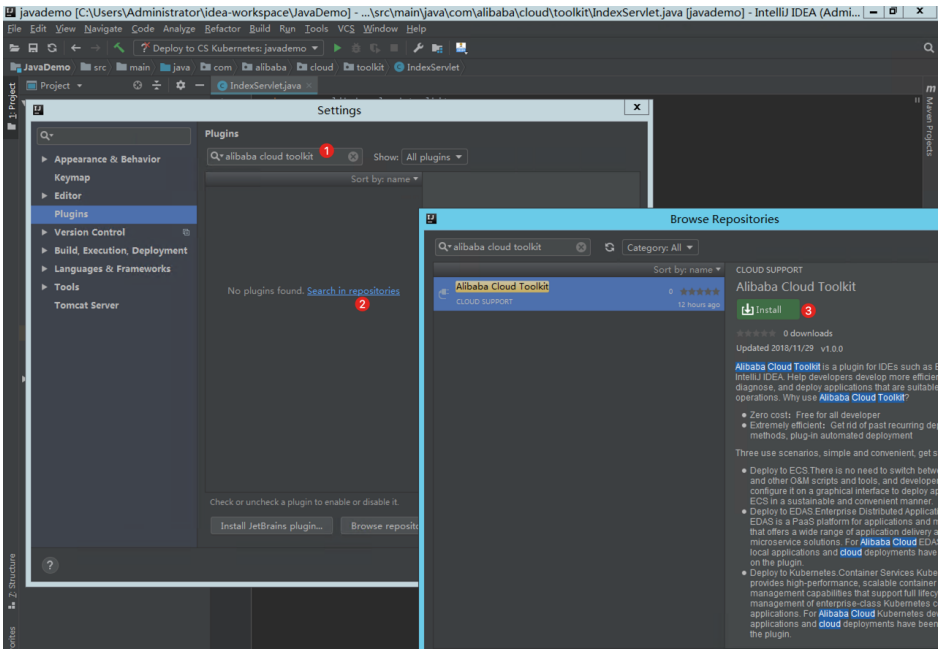
启动 IntelliJ IDEA。

在 IntelliJ IDEA 中安装插件。

Mac 系统： 进入 **Preference** 配置页面，选择左边的 **Plugins**，在右边的搜索框里输入 **Alibaba Cloud Toolkit** ，并单击 **Install** 安装。



Windows 系统：进入 **Plugins** 选项，搜索 **Alibaba Cloud Toolkit**，并单击 **Install** 安装。



在 IntelliJ IDEA 中插件安装成功后，重启 IntelliJ IDEA，您可以在工具栏看到 Alibaba Cloud Toolkit 的图标（ ）。

步骤二：配置 Cloud Toolkit 账号

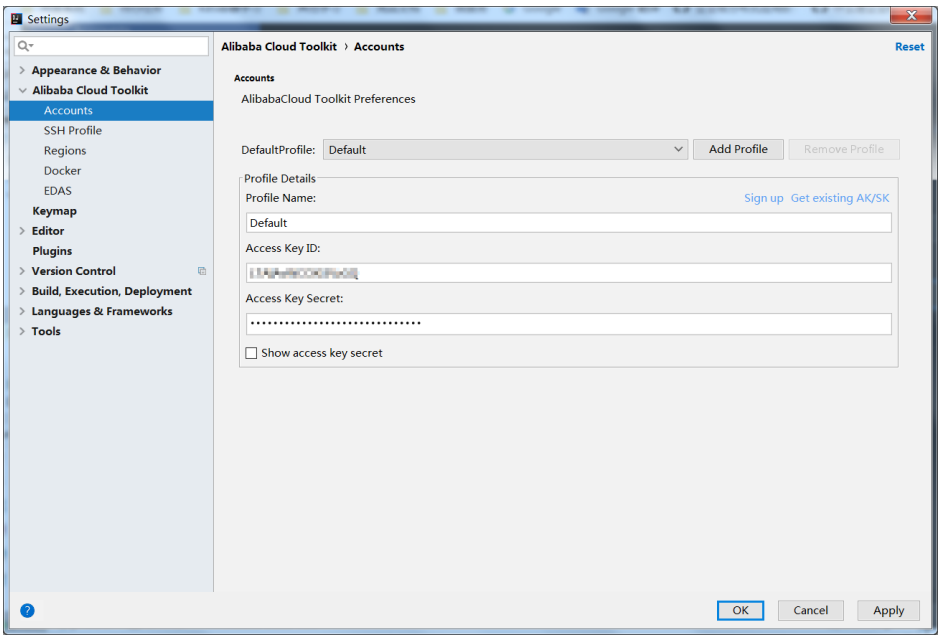
在安装完 Alibaba Cloud Toolkit 后，您需使用 Access Key ID 和 Access Key Secret 来配置 Cloud Toolkit 的账号。

启动 IntelliJ IDEA。

单击 Alibaba Cloud Toolkit 的图标（ ），在下拉列表中单击 **Preference...**，进入设置页面，在左侧导航栏单击 **Alibaba Cloud Toolkit** > **Accounts**。

在 **Accounts** 界面中设置 **Access Key ID** 和 **Access Key Secret**，然后单击 **OK**。

注意：如果您使用子账号的Access Key ID和Access Key Secret，请确认该子账号至少拥有**部署应用**的权限，具体操作方式请参见RAM 账号授权。



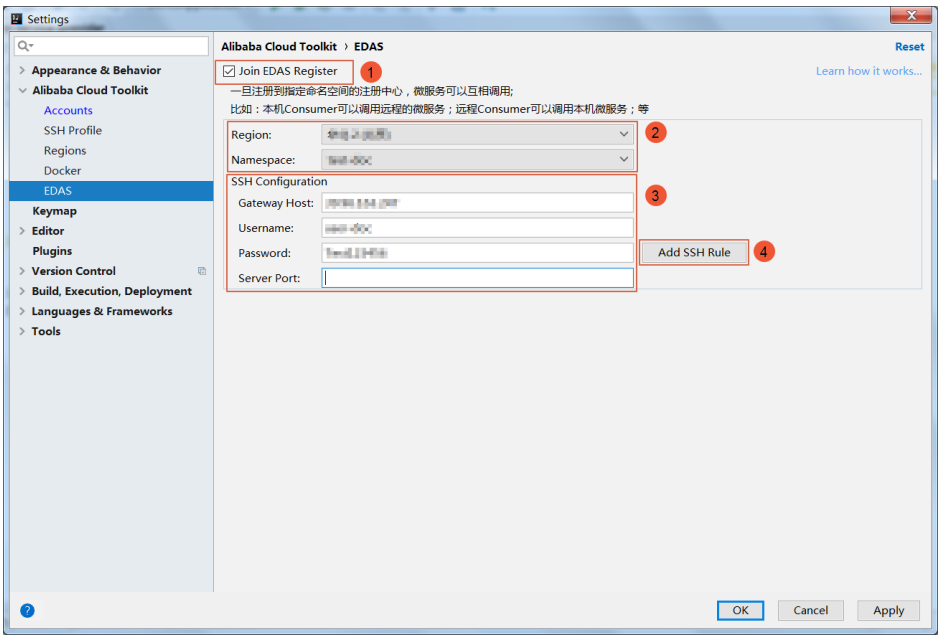
如果您已经注册过阿里云账号，在 **Accounts** 界面中单击 **Get existing AK/SK**，进入阿里云登录页面。用已有账号登录后，跳转至安全信息管理页面，获取 **Access Key ID** 和 **Access Key Secret**。

如果您还没有阿里云账号，在 **Accounts** 界面中单击单击 **Sign up**，进入阿里云账号注册页面，注册账号。注册完成后按照上述方式获取 **Access Key ID** 和 **Access Key Secret**。

步骤三：端云互联配置

在 IntelliJ IDEA 上单击工具栏 Alibaba Cloud Toolkit 的图标（ ），在下拉列表中单击 **Preference...**。

进入设置页面，在左侧导航栏单击 **Alibaba Cloud Toolkit** > **EDAS**，在页面右侧设置区域进行端云互联配置。



勾选 **Join EDAS Register** 开启端云互联功能。

设置 **Region** 和 **Namespace** 为端云互联应用所在的区域和命名空间。

注意：除了默认命名空间外，其他命名空间需手动开启**允许远程调试**选项：

a. 登录EDAS 控制台。
b. 选择**地域**，进入**应用管理** > **命名空间**。
c. 在命名空间列表中单击你要选择的命名空间**操作**列的**编辑**按钮。
d. 在**编辑命名空间**对话框中开启**允许远程调试**按钮。

在 **SSH Configuration** 区域:

在 **Gateway Host** 输入框内输入您创建的 ECS 的**公网 IP**；

在 **Username** 和 **Password** 输入框内输入用于建立 SSH 端云互联通道的用户名和密码：您可以直接输入您用于建立 SSH 端云互联通道的 ECS 的用户名和密码，也可以在这里填入新的用户名和密码，然后通过下面的**Add SSH Rule**来增加此新用户及密码。

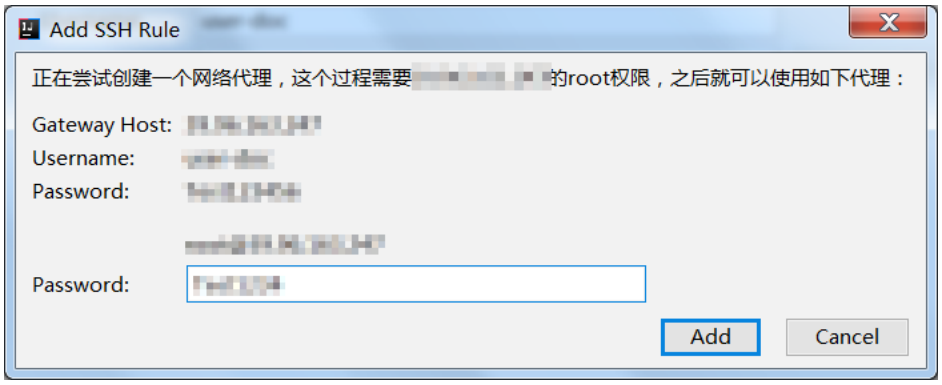
Server Port: Spring Boot 应用需添加该应用的服务端口，其他类型应用不需要填写。

然后单击 **Add SSH Rule** 完成配置。

如果您输入的是 ECS 的 root 用户名和密码，则会使用此 root 账号进行配置

，如果成功则会出现**配置已添加成功**的提示弹窗。

如果使用新账号或其他非root账号进行互联，那么需要root权限来对此账号进行代理配置，如果成功则会出现**配置已添加成功**的提示弹窗。



注意：

i. 此处使用ECS机器的密码只是用来创建一个网络代理，不会将ECS的用户名和密码用于其他用途。
ii. 推荐使用新账号或其他非root账号进行互联，后续可将此新账号或非root账号直接共享给其他需要端云互联的团队成员使用，避免泄漏root信息。

单击 **OK** 或 **Apply** 使配置生效。

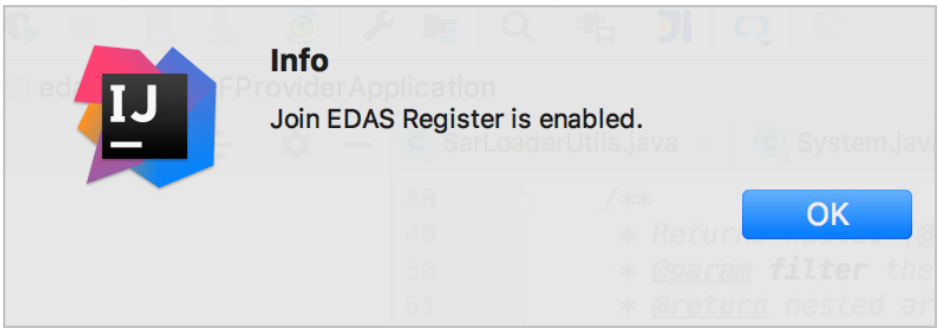
说明：如果使用 EDAS 专有云企业版，还需要按以下步骤在 Cloud Toolkit 中配置 Endpoint。Endpoint 请联系 EDAS 技术支持获取。

在 **Preference (Filtered)** 对话框的左侧导航栏中选择 **Appearance & BehaviorEndpoint**。

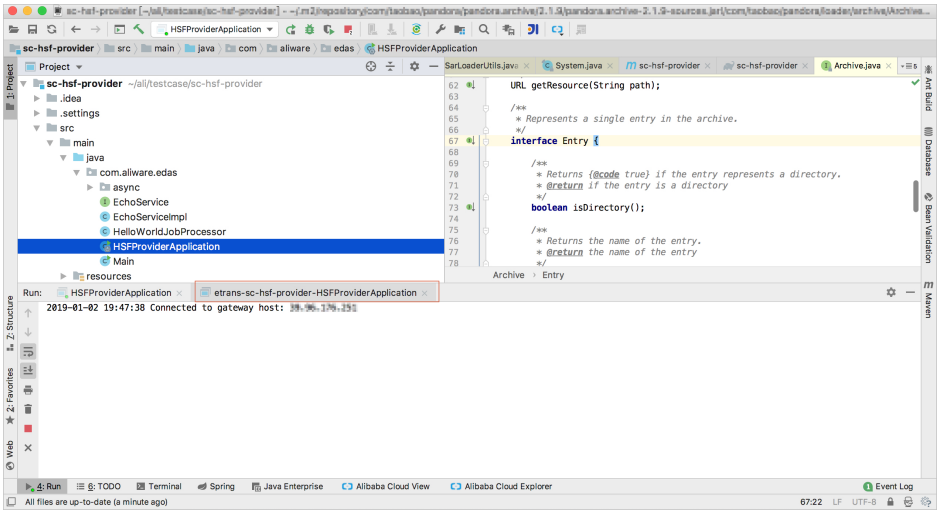
在 **Endpoint** 界面中设置 Endpoint，配置完成后，单击 **Apply and Close**。

步骤四：启动本地应用进行端云互联

启动本地应用，如果当前状态处于端云互联状态，那么会有如下提示：



并且，在启动应用之外会启动一个 etrans 的进程：



更多信息

在使用 Cloud Toolkit 实现端云互联时，如果遇到相关问题，请参考端云互联常见问题进行解决。

您可以在 EDAS 上代理购买 ECS ，详情参考创建 ECS 实例。

如果您想使用 IntelliJ IDEA 插件快速在 EDAS 上部署应用。详情参考使用 IntelliJ IDEA 插件快速部署应用。

使用 Cloud Toolkit 实现端云互联（Eclipse）

您可以在 Eclipse 中使用 Cloud Toolkit 的端云互联功能实现本地和云上应用的相互调用，提升开发效率。

前提条件

开发框架和集群类型对使用端云互联都有些限制，请了解**使用限制**。

下载并安装 Eclipse IDE 4.5.0 及以上版本。

登录**云服务器 ECS 控制台** 创建一台可使用 SSH 登录的 ECS，用于建立端云互联通道。

注意：SSH 通道需要使用密码方式登录，暂不支持使用密钥对登录。

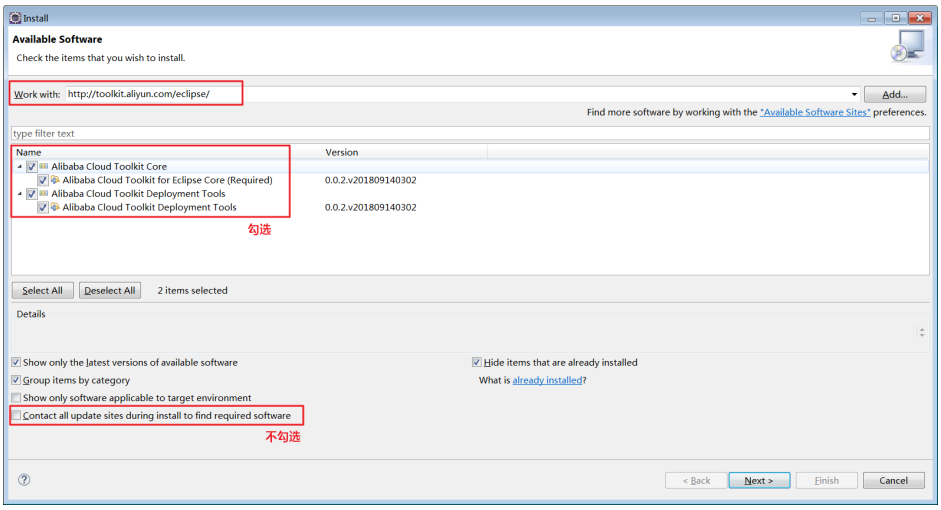
步骤一：安装 Cloud Toolkit

启动 Eclipse。

在菜单栏中选择 **Help** > **Install New Software**。

在 **Available Software** 对话框的 **Work with** 文本框中输入 Cloud Toolkit for Eclipse 的 URL http://toolkit.aliyun.com/eclipse/。

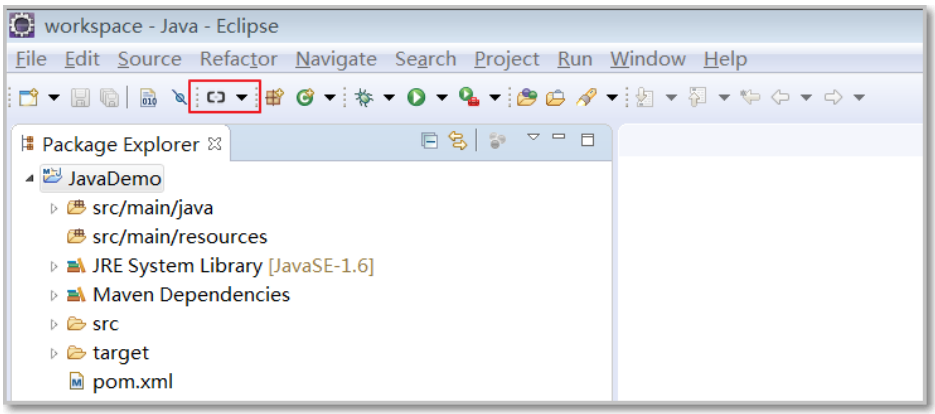
在下面的列表区域中勾选需要的组件 **Alibaba Cloud Toolkit Core** 和 **Alibaba Cloud Toolkit Deployment Tools**，并在下方 Details 区域中不勾选 **Connect all update sites during install to find required software**。完成组件选择之后，单击**Next**。



按照 Eclipse 安装页面的提示，完成后续安装步骤。

注意：安装过程中可能会出现没有数字签名对话框，选择**Install anyway**即可。

Cloud Toolkit 插件安装完成后，重启 Eclipse，您可以在工具栏看到 Alibaba Cloud Toolkit 的图标。



步骤二：配置 Cloud Toolkit 账号

您需使用 Access Key ID 和 Access Key Secret 来配置 Cloud Toolkit 的账号。

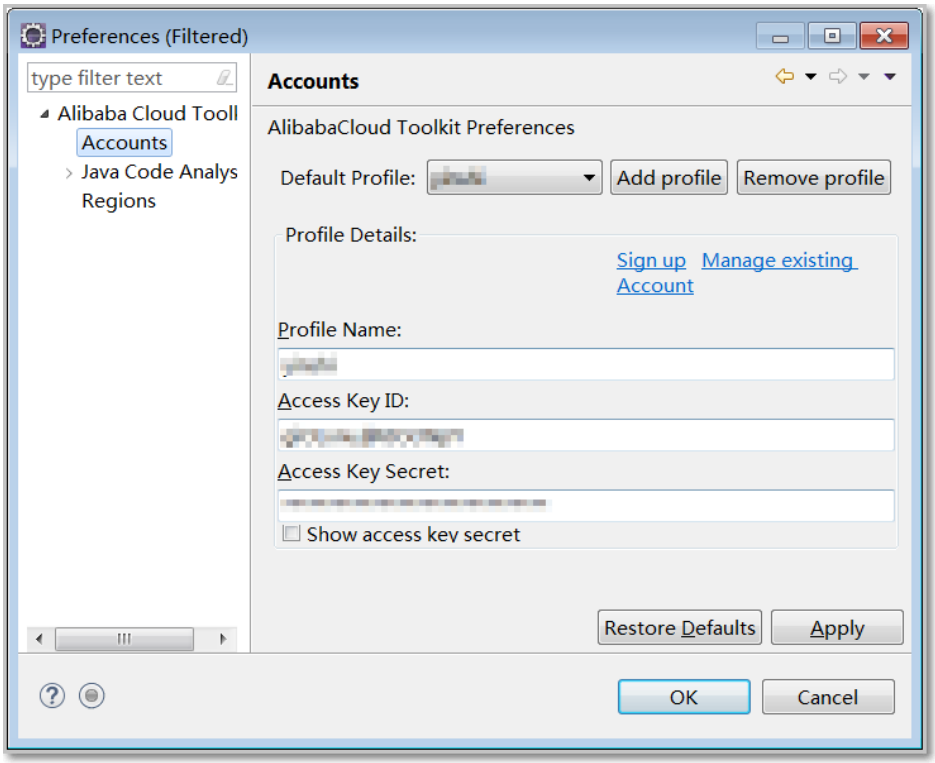
启动 Eclipse。

在工具栏单击 Alibaba Cloud Toolkit 图标右侧的下拉按钮，在下拉菜单中单击 **Preference...**。

在 **Preference (Filtered)** 对话框的左侧导航栏中单击 **Accounts**。

在 **Accounts** 界面中设置 **Access Key ID** 和 **Access Key Secret**，然后单击 **Apply**。

注意：如果您使用子账号的Access Key ID和Access Key Secret，请确认该子账号至少拥有**部署应用**的权限，具体操作方式请参见**RAM 账号授权**。



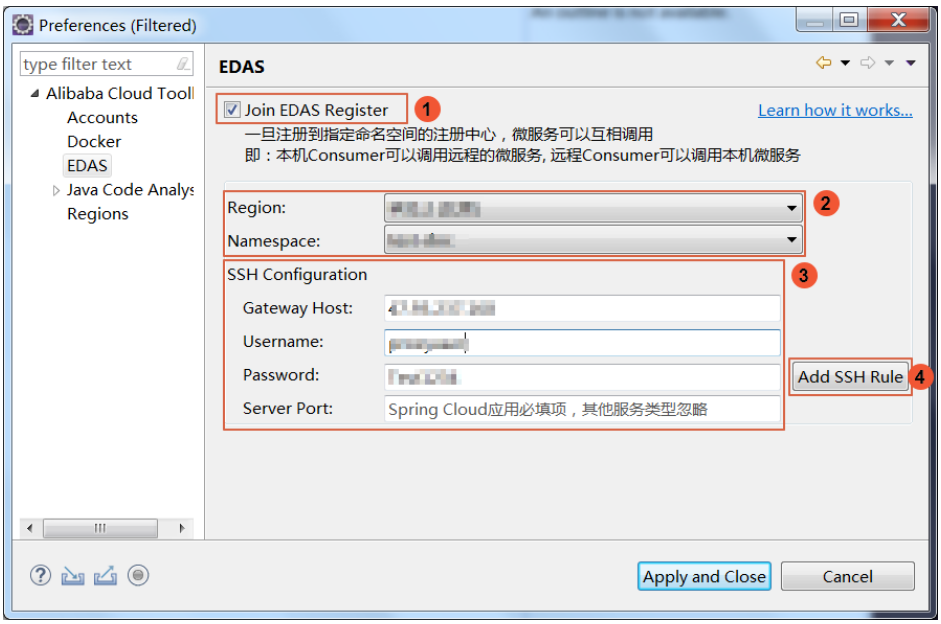
如果您已经注册过阿里云账号，在 **Accounts** 界面中单击 **Manage existing Acount**，进入阿里云登录页面。用已有账号登录后，跳转至安全信息管理页面，获取 **Access Key ID** 和 **Access Key Secret**。

如果您还没有阿里云账号，在 **Accounts** 界面中单击单击 **Sign up**，进入阿里云账号注册页面，注册账号。注册完成后按照上述方式获取 **Access Key ID** 和 **Access Key Secret**。

步骤三：配置端云互联

在 Eclipse 上单击工具栏 Alibaba Cloud Toolkit 的图标（ ），在下拉列表中单击 **Preference...**。

在 **Preference (Filtered)** 对话框的左侧导航栏单击 **Alibaba Cloud Toolkit** > **EDAS**，在页面右侧设置区域进行端云互联配置。



勾选 **Join EDAS Register** 开启端云互联功能。

设置 **Region** 和 **Namespace** 为端云互联应用所在的区域和命名空间。

注意：除了默认命名空间外，其他命名空间需手动开启**允许远程调试**选项：

a. 登录EDAS 控制台。
b. 选择**地域**，进入**应用管理** > **命名空间**。
c. 在命名空间列表中单击你要选择的命名空间**操作**列的**编辑**按钮。
d. 在**编辑命名空间**对话框中开启**允许远程调试**按钮。

在 **SSH Configuration** 区域:

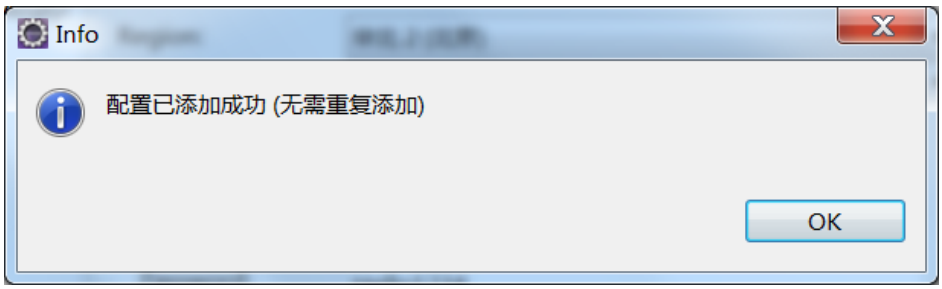
在 **Gateway Host** 输入框内输入您创建的 ECS 的**公网 IP**；

在 **Username** 和 **Password** 输入框内输入用于建立 SSH 端云互联通道的用户名和密码：您可以直接输入您用于建立 SSH 端云互联通道的 ECS 的用户名和密码，也可以在这里填入新的用户名和密码，然后通过下面的**Add SSH Rule**来增加此新用户及密码。

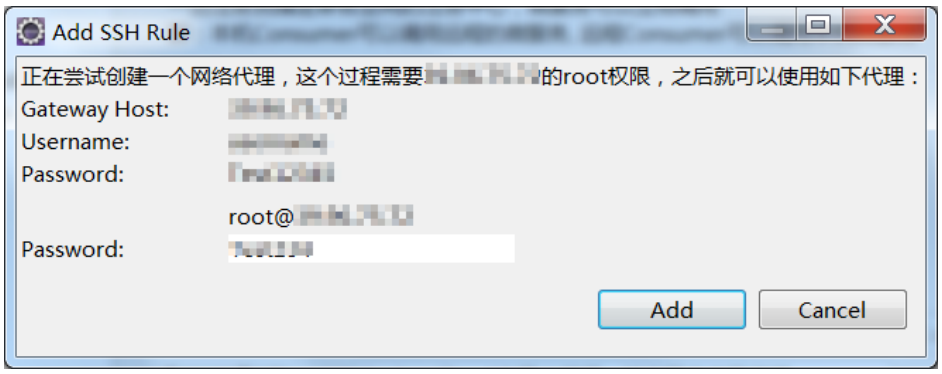
Server Port: Spring Boot 应用需添加该应用的服务端口，其他类型应用不需要填写。

然后单击 **Add SSH Rule** 完成配置。

如果您输入的是 ECS 的 root 用户名和密码，则会使用此 root 账号进行配置，如果成功则会出现**配置已添加成功**的提示弹窗。



如果使用新账号或其他非root账号进行互联，那么需要root权限来对此账号进行代理配置，如果成功则会出现**配置已添加成功**的提示弹窗。



注意：

i. 此处使用ECS机器的密码只是用来创建一个网络代理，不会将ECS的用户名和密码用于其他用途。
ii. 推荐使用新账号或其他非root账号进行互联，后续可将此新账号或非root账号直接共享给其他需要端云互联的团队成员使用，避免泄漏root信息。

单击**Apply and Close**使配置生效。

说明：如果使用 EDAS 专有云企业版，还需要按以下步骤在 Cloud Toolkit 中配置 Endpoint。Endpoint 请联系 EDAS 技术支持获取。

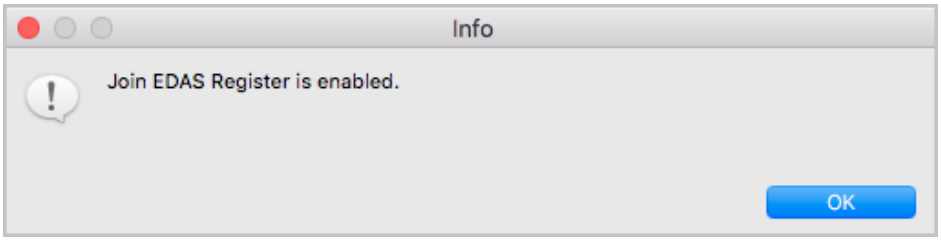
在 **Preference (Filtered)** 对话框的左侧导航栏中选择 **Appearance & BehaviorEndpoint**。

在 **Endpoint** 界面中设置 Endpoint，配置完成后，单击 **Apply and Close**。

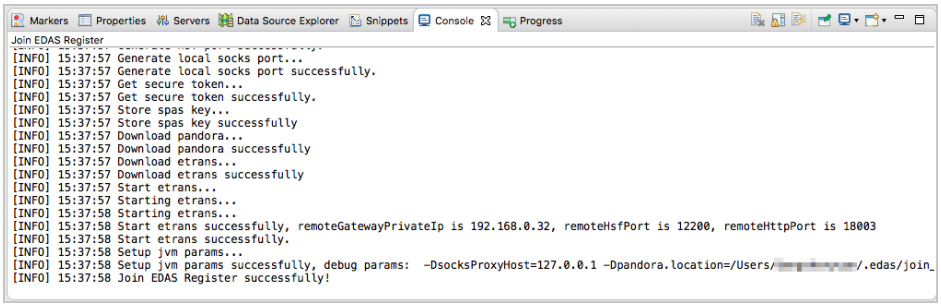
步骤四：启动本地应用进行端云互联

在项目列表中选中工程项目的根目录，然后启动应用。如果当前状态处于端云互联状态，那么会有如下提示：

端云互联可用的提示框。



在 Console 面板中会有一个标题为 Join EDAS Register 的控制台打印初始化端云互联环境的日志。



更多信息

在使用 Cloud Toolkit 实现端云互联时，如果遇到相关问题，请参考端云互联常见问题进行解决。

您可以在 EDAS 上代理购买 ECS ，详情参考创建 ECS 实例。

如果您想使用 Eclipse 插件快速在 EDAS 上部署应用。详情参考使用 Eclipse 插件快速部署应用。