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 ANS。使用 ANS，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 ACM。使用 ACM，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		- RabbitMQ 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		
			ANS	ACM	SchedulerX
Greenwich	2.1.x	0.9.0.RELEASE	0.9.0.RELEASE	0.9.0.RELEASE	0.9.0.RELEASE
Finchley	2.0.x	0.2.2.RELEASE	0.2.2.RELEASE	0.2.2.RELEASE	0.2.2.RELEASE
Edgware	1.5.x	0.1.2.RELEASE	0.1.2.RELEASE	0.1.2.RELEASE	0.1.2.RELEASE

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

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-alicloud-ans，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-alicloud-acm，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

EDAS 将应用配置管理 ACM（Nacos 的商业化组件）集成到控制台中，提供对应用的**配置管理**功能。本文将以一个简单配置为例介绍如何将 Spring Cloud 应用在本地接入 Nacos，并部署到 EDAS 中，使用 ACM 实现配置管理功能。

基于技术背景，会分为以下 3 类用户，您可以根据您的实际情况，选择阅读对应的内容。

如果您完全不了解 Spring Cloud，只有简单的 Spring 和 Maven 基础。本文将向您完整的介绍如何在本地使用 Spring Cloud Alibaba Nacos Config 实现 Spring Cloud 应用的配置管理，如何将应用部署到 EDAS，以及如何使用 EDAS 集成的 ACM 进行配置管理。

如果您熟悉 Spring Cloud 的配置管理组件（如 Consul Config 和 Spring Cloud Config），但未使用过 Spring Cloud Alibaba 的配置管理组件 Nacos Config，那么您只需要将 Spring Cloud 的配置管理组件的依赖和配置替换成 Spring Cloud Alibaba Nacos Config。然后按照本文完成本地结果验证，将应用部署到 EDAS，使用 ACM 进行配置管理。

Spring Cloud Alibaba Nacos Config 同样实现了 Spring Cloud Config 的标准接口与规范，和您之前实现配置管理的方式基本一致。

如果您已经熟悉如何使用 Spring Cloud Alibaba Nacos Config 实现 Spring Cloud 应用的配置管理，可以自行完成应用开发。只需按照文档将应用部署到 EDAS，使用 ACM 进行配置管理。

ACM 与 Nacos、Eureka、ZooKeeper 和 Spring Cloud Config 相比，具有以下优势：

- 共享组件：节省了您部署、运维 Nacos、Consul、ZooKeeper 或 Spring Cloud Config Server 的成本。
- 实时推送：与 Spring Cloud Config 相比，商业版的 ACM 支持修改过的配置自动推送到监听的客户端。
- 配置推送跟踪：可查询所有客户端配置推送状态和轨迹。

本地开发

本地开发中主要描述实现配置管理的相关信息，如果您想了解完整的 Spring Cloud 程序，可下载 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>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-alibaba-nacos-config</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.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 ＋。

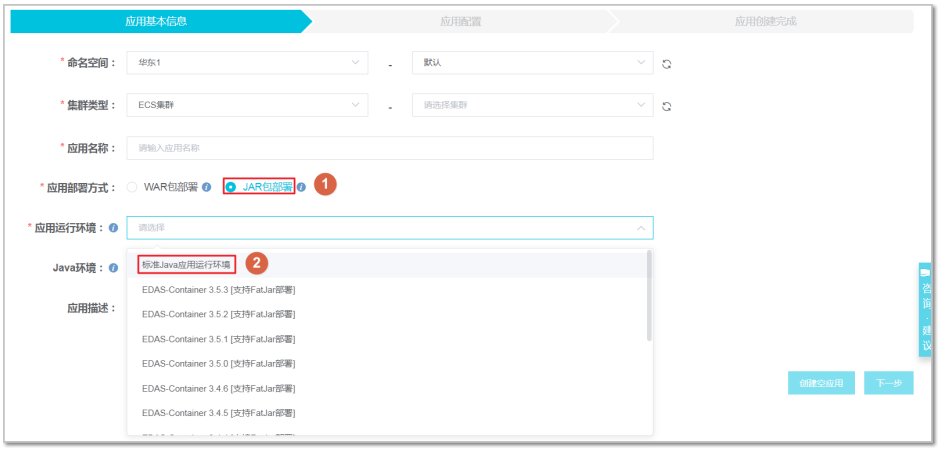
在**新建配置**页面设置 **Data ID**、**Group** 和 **配置内容**，然后单击**发布**。

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>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-alibaba-nacos-discovery</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>
```

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-gateway</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-alibaba-nacos-discovery</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>
```

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 0.9.0.RELEASE。Spring Cloud Finchley 对应的 Spring Cloud Alibaba 版本为 0.2.2.RELEASE，Spring Cloud Edgware 对应的 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。

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 = "hengyu-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-test2.json")
private Resource file;
@GetMapping("/upload")
public String upload() {
try {
ossClient.putObject(OssApplication.BUCKET_NAME, "oss-test2.json", this
```

```
.getClass().getClassLoader().getResourceAsStream("oss-test2.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-test2.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）*。

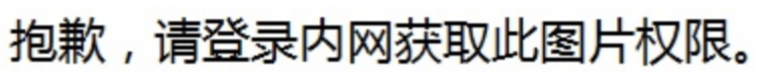
结果验证

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):

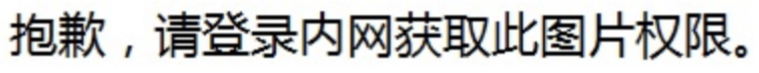


异常比例验证

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):



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*。

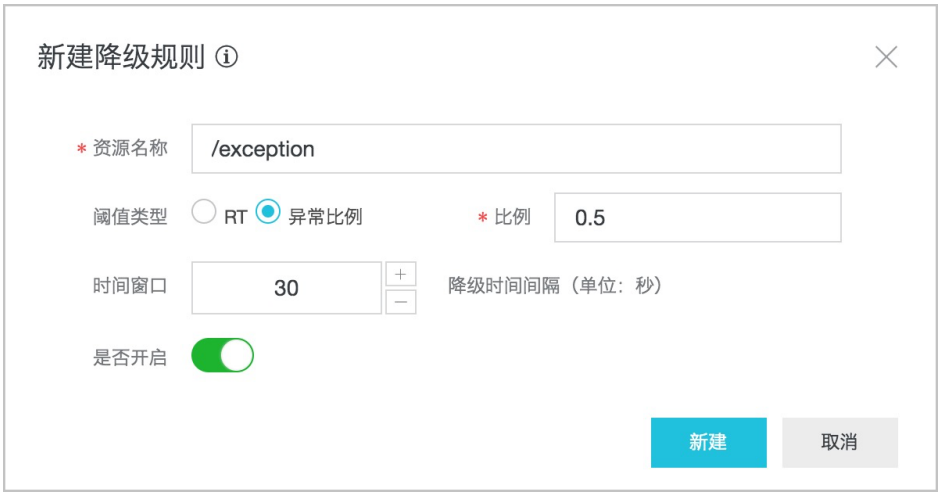
在应用详情页左侧导航栏选择**限流降级** > **降级规则**。

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

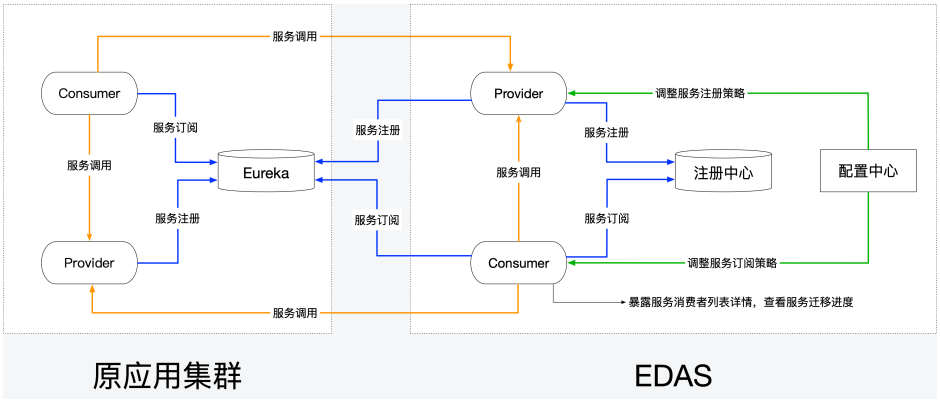
使用 Spring Cloud Alibaba 将原有的服务注册中心切换到 Nacos。开发一套新的应用部署到 EDAS，最后通过 SLB 和域名配置来进行切流。

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

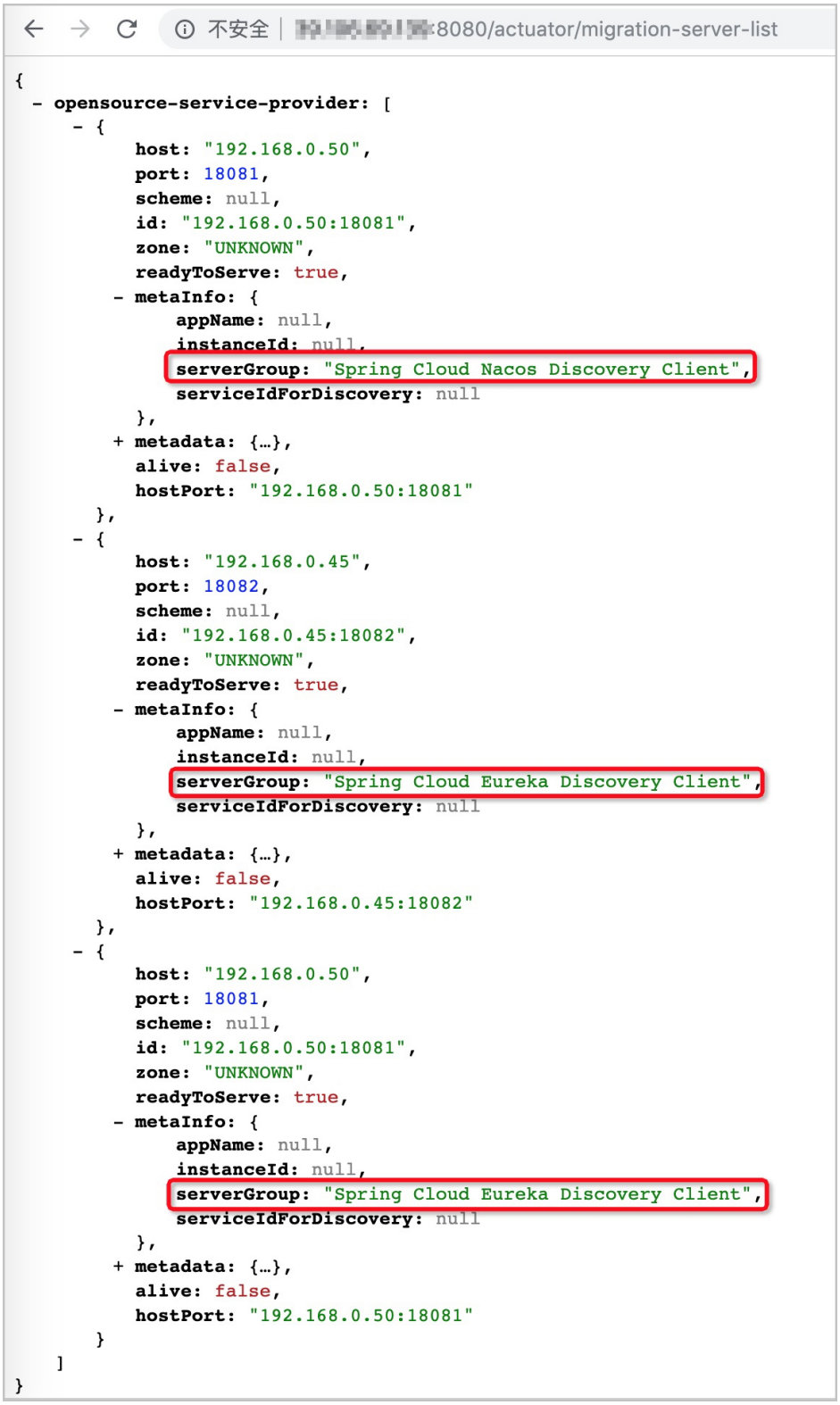
最重要的是观察业务本身是否正常。

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。

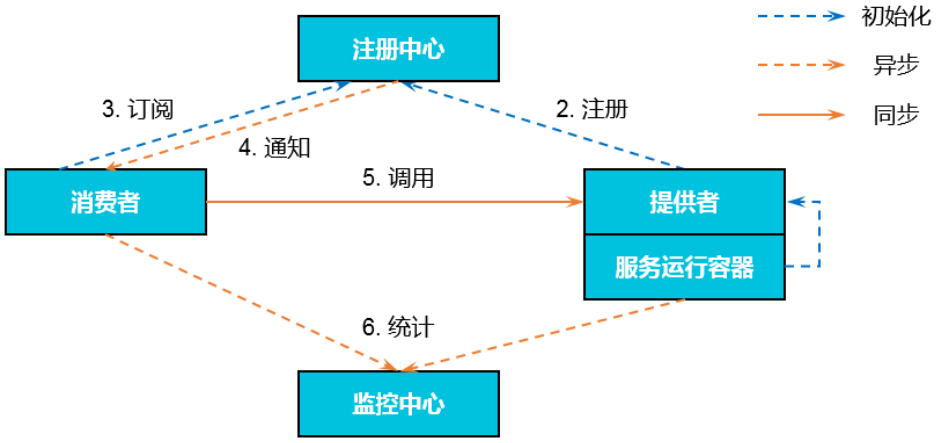
使用 Dubbo 开发应用

Dubbo 概述

EDAS 支持原生 Dubbo 微服务框架，您在这个框架下开发的微服务只需添加依赖和修改配置，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，实现代码零入侵。

Dubbo 架构

Dubbo 的架构如下图所示。



1. 服务运行容器负责启动，加载，运行提供者服务。
2. 提供者在启动时，向注册中心注册。
3. 消费者在启动时，向注册中心订阅所需的服务。
4. 注册中心返回提供者地址列表给消费者。如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 消费者从提供者地址列表中，基于软负载均衡算法，选一个提供者进行调用。如果调用失败，再调用另一个。
6. 消费者和提供者在内存中存储累计调用次数和调用时间，定时（每分钟）发送一次统计数据到监控中心。

使用 Spring Boot 开发 Dubbo 应用

如果您只有简单的 Java 基础和 Maven 经验，而不熟悉 Dubbo，您可以从零开始开发 Dubbo 服务，并使用 EDAS 服务注册中心实现服务注册与发现。

从零开始开发 Dubbo 服务有两种主要的方式：

- 使用 xml 开发

- 使用 Spring Boot 的注解方式开发

使用 xml 方式请参考 将 Dubbo 应用托管到 EDAS。文本档将向您介绍如何使用 Spring Boot 的注解方式开发 Dubbo 服务。Spring Boot 使用一些极简的配置，就能快速搭建一个基于 Spring 的 Dubbo 服务，相比 xml 的开发方式，开发效率更高。

为什么使用 EDAS 服务注册中心

EDAS 服务注册中心实现了 Dubbo 所提供的 SPI 标准的注册中心扩展，能够完整地支持 Dubbo 服务注册、路由规则、配置规则功能。

EDAS 服务注册中心能够完全代替 ZooKeeper 和 Redis，作为您 Dubbo 服务的注册中心。同时，与 ZooKeeper 和 Redis 相比，还具有以下优势：

- EDAS 服务注册中心为共享组件，节省了您运维、部署 ZooKeeper 等组件的机器成本。
- EDAS 服务注册中心在通信过程中增加了鉴权加密功能，为您的服务注册链路进行了安全加固。
- 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。

本地开发

准备工作

下载、启动及配置轻量级配置中心。

为了便于本地开发，EDAS 提供了一个包含了 EDAS 服务注册中心基本功能的轻量级配置中心。基于轻量级配置中心开发的应用无需修改任何代码和配置就可以部署到云端的 EDAS 中。

请您参考 配置轻量级配置中心 进行下载、启动及配置。推荐使用最新版本。

下载 Maven 并设置环境变量（本地已安装的可略过）。

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。

这里我们以 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>com.alibaba.boot</groupId>
<artifactId>dubbo-spring-boot-starter</artifactId>
<version>0.2.0</version>
</dependency>
<dependency>
<groupId>com.alibaba.edas</groupId>
<artifactId>edas-dubbo-extension</artifactId>
<version>1.0.6</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 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=edas://127.0.0.1:8080
```

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的值前缀必须是一个 **edas://** 开头，后面的 IP 地址和端口指的是轻量版配置中心

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);
}

}
```

登录轻量版配置中心控制台，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。

这里我们以 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>com.alibaba.boot</groupId>
<artifactId>dubbo-spring-boot-starter</artifactId>
<version>0.2.0</version>
</dependency>
<dependency>
<groupId>com.alibaba.edas</groupId>
<artifactId>edas-dubbo-extension</artifactId>
<version>1.0.6</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=edas://127.0.0.1:8080
```

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的值前缀必须是一个 **edas://** 开头，后面的 IP 地址和端口指的是轻量版配置中心。

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);
}

}
```

登录轻量版配置中心控制台，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

本地验证。

curl http://localhost:17080/sayHello/EDAS

Hello, EDAS (from Dubbo with Spring Boot)

在 EDAS 中验证。

curl http://localhost:8080/sayHello/EDAS

Hello, EDAS (from Dubbo with Spring Boot)

部署到 EDAS

edas-dubbo-extension 在设计之初就考虑到了从本地迁移到 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。

分别在DubboProvider和DubboConsumer的pom.xml文件中添加如下配置，然后执行 mvn clean package 将本地的程序打成可执行的 JAR 包。

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

根据您要部署的集群类型，参考对应的文档部署应用。

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

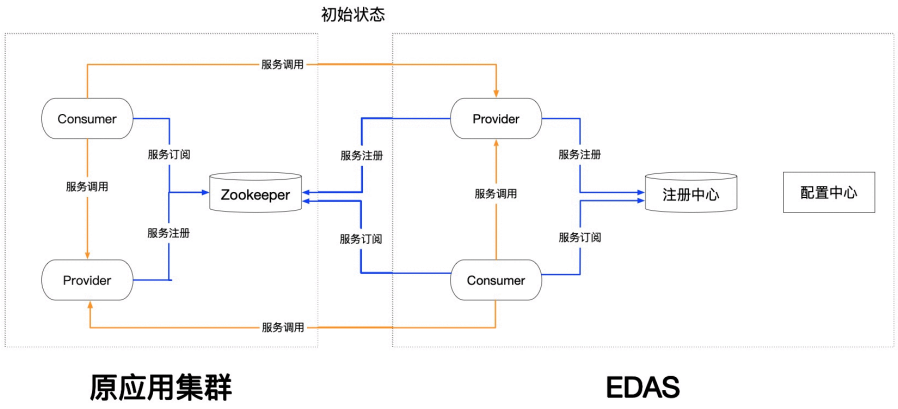
使用 Dubbo 将原有的服务注册中心切换到 EDAS ConfigServer，开发一套新的应用部署到 EDAS，最后通过 SLB 和域名配置来进行切流。

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。

使用 Cloud Toolkit 创建 Apache Dubbo 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，并部署到 EDAS 上。本文介绍如何使用 Cloud Toolkit 创建 Apache Dubbo 应用样例工程（包含一个 Provider 和 Consumer），并完成本地调用验证。

安装及配置 Cloud Toolkit

本文主要介绍如何通过 Cloud Toolkit 创建 Apache Dubbo 应用样例工程，所以首先需要安装及配置 Cloud Toolkit：

在开发工具（IntelliJ IDEA 或 Eclipse）中安装 Cloud Toolkit。

您可以通过 JetBrains 市场和离线包两种方式安装 Cloud Toolkit。推荐使用 JetBrains 插件市场安装，但如果遇到网络原因，搜索不到 Alibaba Cloud Toolkit，建议多尝试几次，或者使用离线包安装。

在 Cloud Toolkit 中配置 Accounts。

配置 Accounts 主要指配置阿里云账号的 Access Key ID 和 Access Key Secret。所以，在配置 Accounts 前需要获取 Access Key ID 和 Access Key Secret。

安装和配置 Cloud Toolkit 的详细操作步骤请参见在 IntelliJ IDEA 中安装和配置 Cloud Toolkit 和在 Eclipse 中安装和配置 Cloud Toolkit。

创建 Apache Dubbo 应用样例工程

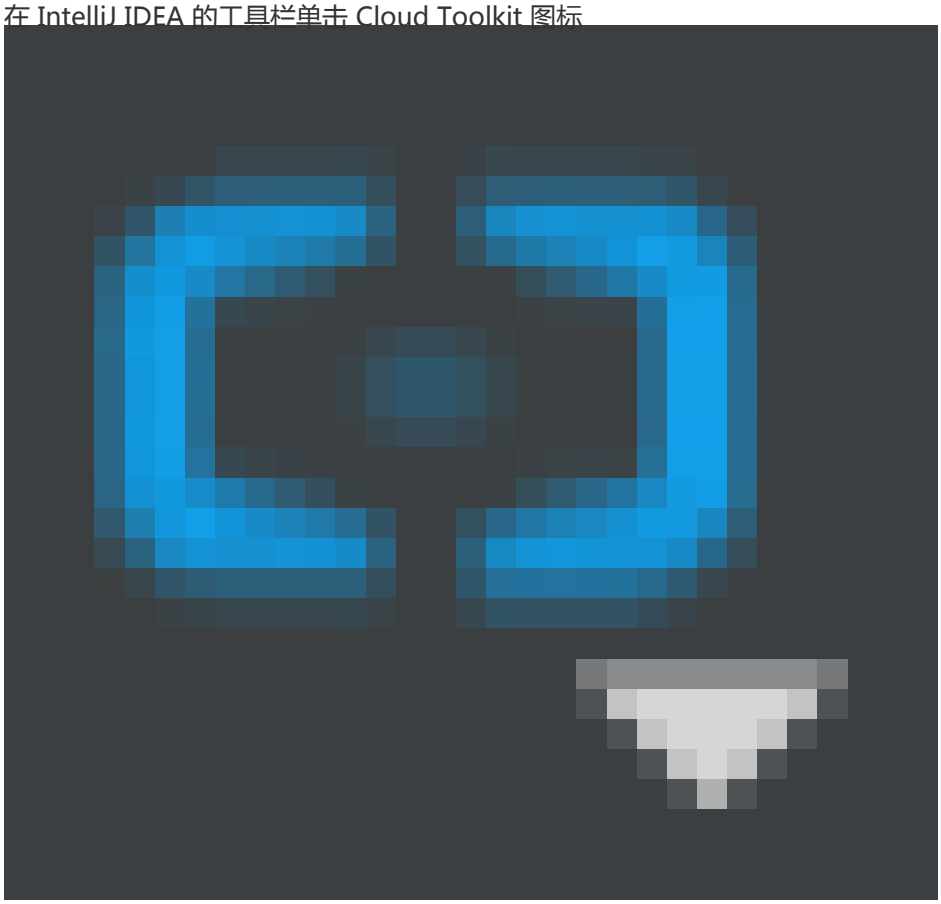
本文以 IntelliJ IDEA 为例介绍如何创建 Apache Dubbo 应用样例工程。在 Eclipse 中创建的过程基本一致，仅仅是界面稍有不同。

前提条件

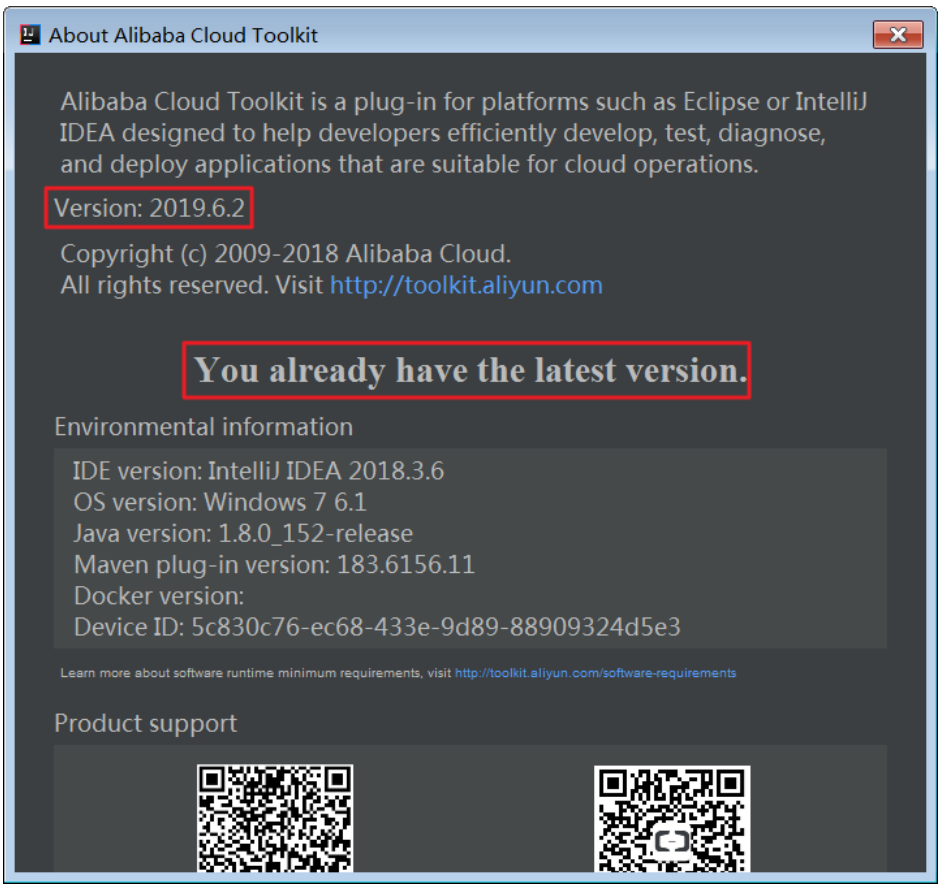
2019.6.2 及后续版本的 Cloud Toolkit 支持 Apache Dubbo 工程。

如果您是按照上面的步骤新安装的 Cloud Toolkit，版本符合要求，可以创建 Apache Dubbo 工程。

如果您此前已经安装了 Cloud Toolkit，需要先确认版本是否为 *2019.6.2* 及后续版本：

在 IntelliJ IDEA 的工具栏单击 Cloud Toolkit 图标  ，在下拉菜单中单击 **About**。

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

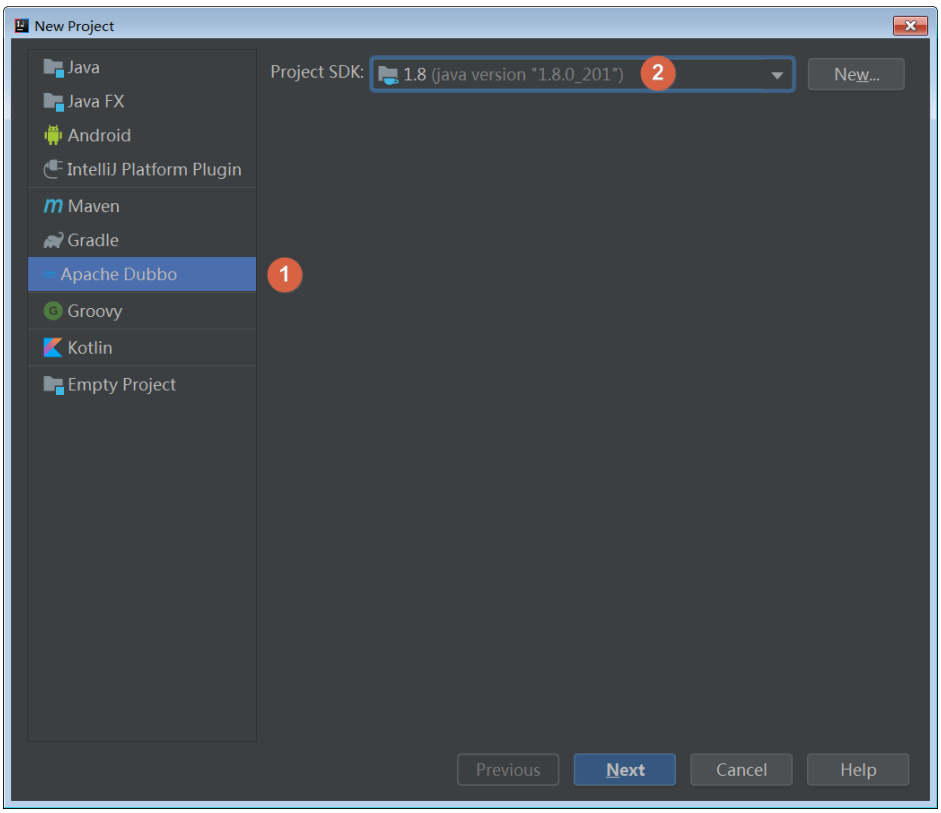


- 如果 Cloud Toolkit 的版本为 *2019.6.2* 或更新版本，则可以支持 Apache Dubbo 工程。
- 如果 Cloud Toolkit 的版本为 *2019.6.2* 之前的版本，则需要升级。

步骤一：新建工程

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

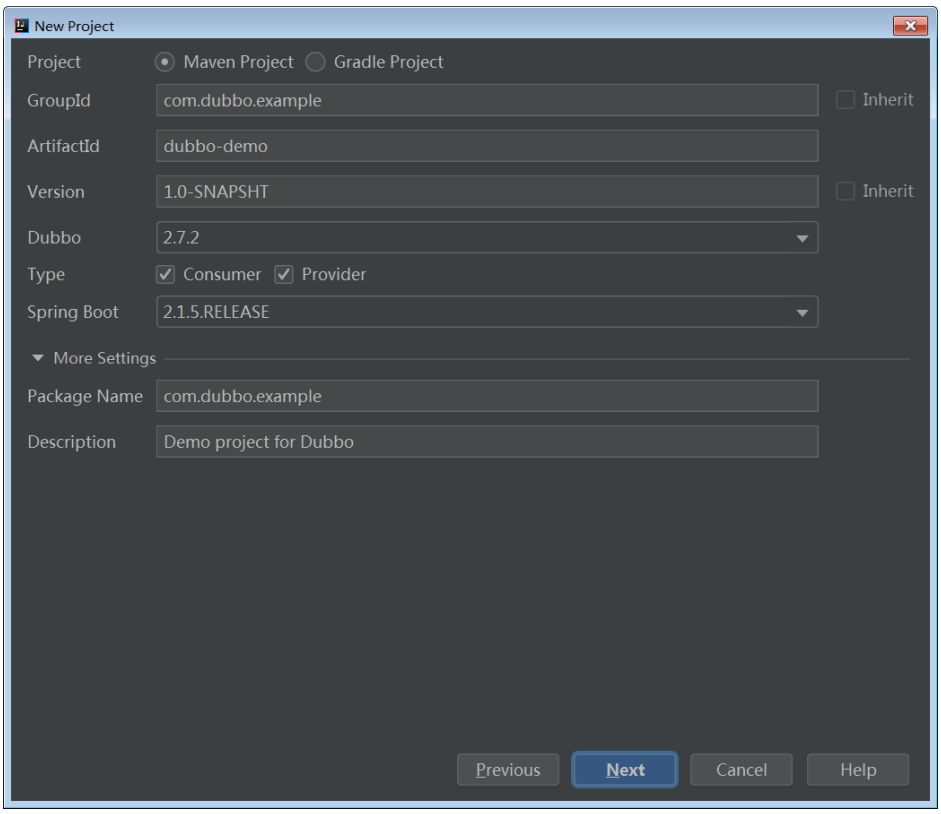
在 **New Project** 对话框左侧的导航栏中单击 **Apache Dubbo**，在右侧界面中选择 **JDK 版本**，然后单击 **Next**。



说明：JDK 在安装 Cloud Toolkit 时已经安装。

步骤二：设置工程基本配置

在界面中设置基本参数。



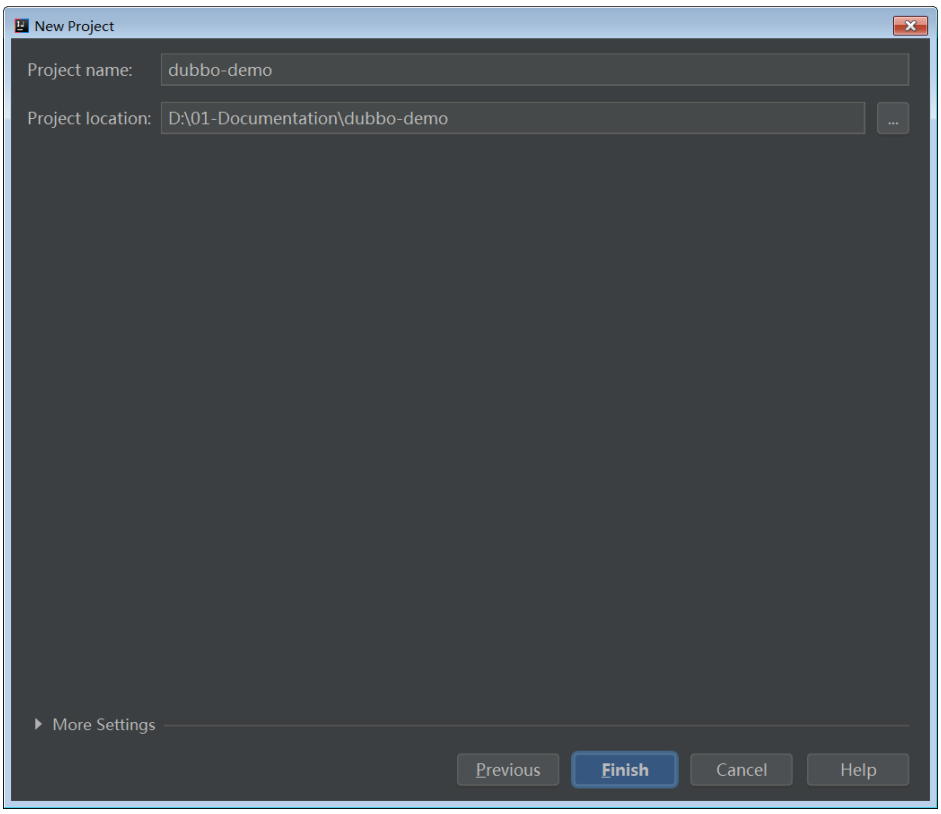
基本参数设置说明及示例：

- **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**：选择 Dubbo 的版本，如 *2.7.2*。
- **Type**：工程的类型，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*。

设置完成后，单击 **Next**。

步骤三：配置工程名和目录

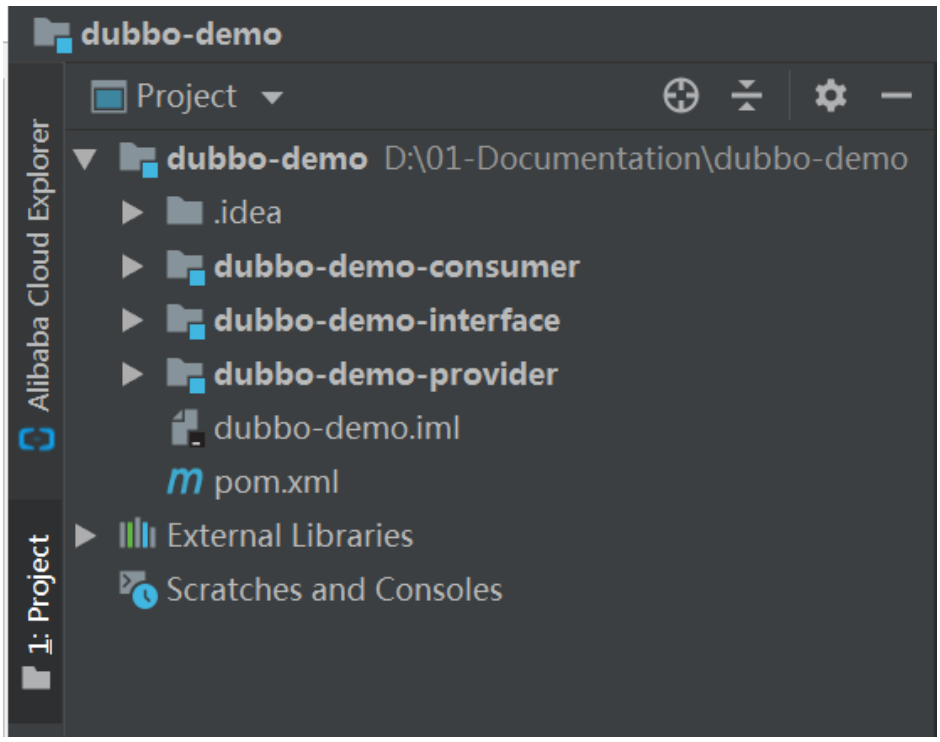
在界面中设置 **Project name** 和 **Project location**。



设置完成后，单击 **Finish**。

结果验证

Apache Dubbo 样例工程创建完成后，可以在 IntelliJ IDEA 中看到此工程。



验证 Apache Dubbo 应用样例工程

通过 Cloud Toolkit 创建的 Apache Dubbo 应用样例工程是一个 Spring boot + Dubbo 的工程。服务端（Provider）提供了一个服务com.dubbo.example.DemoService，通过 Dubbo 协议暴露在12345端口。

注意：由于默认配置为dubbo.registry.address=N/A，说明服务没有注册到任何注册中心，需要客户端（Consumer）通过直连的方式来发现服务。

步骤一：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，说明服务端（Provider）成功启动。

```
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...
```

步骤二：启动 Consumer 并验证调用

在客户端（Consumer）的com.dubbo.example.consumer.DubboConsumerBootstrap中有如下代码：

```
@Reference(version = "1.0.0", url = "dubbo://127.0.0.1:12345")
private DemoService demoService;
```

这表明客户端（Consumer）通过指定服务端（Provider）地址dubbo://127.0.0.1:12345的直连方式调用服务。

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：

```
Hello mercyblitz, request from consumer: /30.5.124.39:59553
```

观察客户端（Consumer）打印日志，出现以下字段，则说明调用成功。

```
Hello mercyblitz, response from provider: 30.5.124.39:12345
```

注意：

- 若要使用注册中心进行服务注册发现，请修改dubbo-demo/dubbo-demo-provider/src/main/resources/application.properties和dubbo-demo/dubbo-demo-consumer/src/main/resources/application.properties文件中的dubbo.registry.address为对应的注册中心的地址，如*zookeeper://localhost:2181*。
- 如果使用 ZooKeeper 为注册中心，还需要在 Provider 和 Consumer 各自的样例工程的pom.xml文件中添加 ZooKeeper 的依赖。

后续操作

在完成 Apache Dubbo 样例工程的创建和调用验证后，可以将该样例工程打包（JAR 包）并部署到 EDAS 的不同集群（主要为 ECS 集群和容器服务 Kubernetes 集群）中，部署可以通过控制台或工具的多种途径完成，详情请参见应用部署概述。

使用 HSF 开发应用

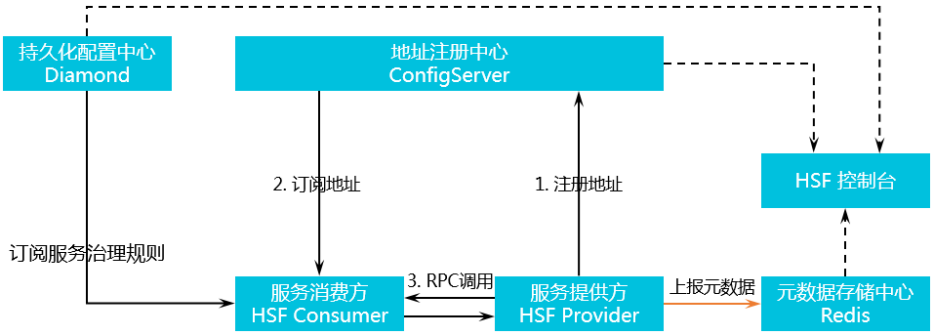
HSF 概述

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，从而帮助您方便、快速的开发分布式应用，以及提供或使用公共功能模块，并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。



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。地址注册中心的角色是由 ConfigServer 承担的。

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。持久化配置中心的角色是由 Diamond 承担的。

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。元数据存储中心的角色是由 Redis 承担的。

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。

配置轻量配置中心

轻量配置中心给开发者提供在开发、调试、测试的过程中的服务发现、注册和查询功能。在一个公司内部，通常只需要在一台机器上安装轻量配置中心服务，并在其他开发机器上绑定特定的 hosts 即可。轻量级配置中心的安装和使用步骤请参见下文。

轻量级配置中心不属于 EDAS 正式环境中的服务，如希望开发过程中使用正式的注册中心，可使用 EDAS 提供的 IDE 联调插件，将本地应用注册到云端注册中心。云端联调请参考 Eclipse 插件端云互联或 IntelliJ IDEA 插件端云互联进行安装使用。

步骤一：下载轻量配置中心

1. 确认环境是否达到要求。正确配置环境变量 JAVA_HOME，指向一个 1.6 或 1.6 以上版本的 JDK。确认 8080 和 9600 端口未被使用。由于启动 EDAS 配置中心将会占用此台机器的 8080 和 9600 端口，因此推荐您找一台**专门的**机器启动 EDAS 配置中心，比如某台测试机器。如果您是在同一台机器上进行测试，请将 Web 项目的端口修改为其它未被占用的端口。
2. 下载最新版本的 EDAS 配置中心安装包并解压。如有需要，可以下载历史版本：2018年10月版本 2018年01月版本2017年08月版本2017年07月版本2017年03月版本2016年12月版本

步骤二：启动轻量配置中心

进入解压目录（edas-config-center），启动配置中心。

- Windows 操作系统：请双击 startup.bat。
- Unix 操作系统：请在当前目录下执行 sh startup.sh 命令。

步骤三：配置 hosts

对于需要使用轻量配置中心的开发机器，请在本地 DNS（hosts 文件）中，将 jmenv.tbsite.net 域名指向启动了 EDAS 配置中心的机器 IP。

hosts 文件的路径如下：

- Windows 操作系统：C:\Windows\System32\drivers\etc\hosts

- Unix 操作系统：/etc/hosts

示例

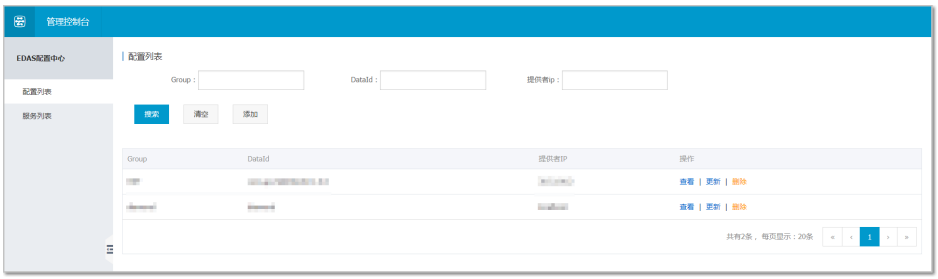
如果您在 IP 为 192.168.1.100 的机器上面启动了 EDAS 配置中心，则所有开发者只需要在机器的 hosts 文件里加入如下一行即可。

192.168.1.100 jmenv.tbsite.net

结果验证

绑定轻量配置中心的 host 之后，打开浏览器，在地址栏输入 jmenv.tbsite.net:8080，回车。

即可看到轻量配置中心首页：



- 如果可以正常显示，说明轻量配置中心配置成功。
- 如果不能正常显示，请根据之前的步骤一步步排查问题所在。

如果您在配置轻量配置中心过程中遇到问题，请参见轻量配置中心问题进行定位、解决。

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）

您可以使用 EDAS-SDK 开发 HSF 应用，实现服务注册发现，还可以实现隐式传参、异步调用、泛化调用、调用链路 Filter 扩展等高级 HSF 特性，以及 HSF 单元测试。

快速开始

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

本章中介绍的代码均可以通过官方 Demo 获取。

下载 Demo 工程。

解压下载的压缩包，可以看到carshop文件夹，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：生产者服务
- 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，消费者也是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**。

开发生产者服务

生产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 文件夹为生产者服务的示例代码。

实现服务接口

可以参考ItemServiceImpl.java文件中的示例：

```
package com.alibaba.edas.carshop.itemcenter;
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;
}
}
```

配置服务属性

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：

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>1.5.0</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>
```

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

生产者服务属性列表

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

服务创建及发布限制

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	aliware	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）。

生产者服务属性配置示例

```
<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>
```

开发消费者服务

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，详情请参见开发生产者服务的配置服务属性。

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>
```

配置服务调用

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" ) );
……
}
}
```

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

消费者服务属性列表

属性	描述
interface	必须配置，类型为 [String]，为需要调用的服务的接口。
version	可选配置，类型为 [String]，为需要调用的服务的版本，默认为1.0.0。
methodSpecials	可选配置，为方法单独配置超时时间(单位 ms)。这样接口中的方法可以采用不同的超时时间，该配置优先级高于服务端的超时配置。
target	主要用于单元测试环境和开发环境中，手动地指定服务提供端的地址。如果不想通过此方式，而是通

	过配置中心推送的目标服务地址信息来指定服务端地址，可以在消费者端指定 -Dhsf.run.mode=0。
connectionNum	可选配置，为支持设置连接到 server 连接数，默认为1。在小数据传输，要求低延迟的情况下设置多一些，会提升 TPS。
clientTimeout	客户端统一设置接口中所有方法的超时时间(单位ms)。超时时间设置优先级由高到低是：客户端methodSpecials，客户端接口级别，服务端methodSpecials，服务端接口级别 。
asyncallMethods	可选配置，类型为 [List]，设置调用此服务时需要采用**异步调用**的方法名列表以及异步调用的方式。默认为空集合，即所有方法都采用同步调用。
maxWaitTimeForCsAddress	配置该参数，目的是当服务进行订阅时，会在该参数指定时间内，阻塞线程等待地址推送，避免调用该服务时因为地址为空而出现地址找不到的情况。若超过该参数指定时间，地址还是没有推送，线程将不再等待，继续初始化后续内容。 **注意**，在应用初始化时，需要调用某个服务时才使用该参数。如果不需要调用其它服务，请勿使用该参数，会延长启动启动时间。

消费者服务属性配置示例

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

发布服务

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 开发环境和配置 IDEA 开发环境。

在开发环境配置时，有一些额外 JVM 启动参数来改变 HSF 的行为，具体如下：

属性	**描述**
-Dhsf.server.port	指定 HSF 的启动服务绑定端口，默认值为12200。
-Dhsf.serializer	指定 HSF 的序列化方式，默认值为 hessian。
-Dhsf.server.max.poolsize	指定 HSF 的服务端最大线程池大小，默认值为600。
-Dhsf.server.min.poolsize	指定 HSF 的服务端最小线程池大小。默认值为50。
-DHSF_SERVER_PUB_HOST	指定对外暴露的 IP，如果不配置，使用 -

	Dhsf.server.ip 的值。
-DHSF_SERVER_PUB_PORT	指定对外暴露的端口，该端口必须在本机被监听，并对外开放了访问授权，默认使用 -Dhsf.server.port 的配置，如果 -Dhsf.server.port 没有配置，默认使用12200。

开发环境查询 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。

开发高级特性

在您参照上面的快速开始开发完基本的应用之后，下面将向您介绍如何在上面的应用中实现隐式传参、异步调用、泛化调用和 Filter 链路扩展等 HSF 的高级特性。您可以直接下载 Demo。

隐式传参（目前仅支持字符串传输）

隐式传参一般用于传递一些简单 KV 数据，又不想通过接口方式传递，类似于 Cookie。

单个参数传递

服务消费者：

RpcContext.getContext().setAttachment("key", "args test");

服务提供者：

String keyVal=RpcContext.getContext().getAttachment("key");

多个参数传递

服务消费者：

```
Map<String,String> map=new HashMap<String,String>();
map.put("param1", "param1 test");
map.put("param2", "param2 test");
map.put("param3", "param3 test");
map.put("param4", "param4 test");
map.put("param5", "param5 test");
RpcContext rpcContext = RpcContext.getContext();
rpcContext.setAttachments(map);
```

服务提供者：

```
Map<String,String> map=rpcContext.getAttachments();
Set<String> set=map.keySet();
for (String key : set) {
System.out.println("map value:"+map.get(key));
}
```

说明：隐式传参只对单次调用有效，当消费端调用返回后，会自动擦除 RpcContext 中的信息。

异步调用

支持 callback 和 future 两种异步调用方式。

callback 调用方式

客户端配置为 callback 方式时，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。

注意：这个 HSFResponseCallback 接口的 listener 不能是内部类，否则 Pandora 的 classloader 在加载时就会报错。

XML 中的配置：

```
<hsf:consumer id="demoApi" interface="com.alibaba.demo.api.DemoApi"
version="1.1.2" >
<hsf:asyncallMethods>
<hsf:method name="ayncTest" type="callback"
listener="com.alibaba.ifree.hsf.consumer.AsynABTestCallbackHandler" />
</hsf:asyncallMethods>
</hsf:consumer>
```

其中 AsynABTestCallbackHandler 类实现了 HSFResponseCallback 接口。DemoApi 接口中有一个方法是 ayncTest 。

代码示例

```
public void onAppResponse(Object appResponse) {
//获取到异步调用后的值
String msg = (String)appResponse;
System.out.println("msg:"+msg);
}
```

注意：

- 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。
- 不支持在 call 里再发起 HSF 调用。这种做法可能导致 IO 线程挂起，无法恢复。

future 调用方式

客户端配置为 future 方式时，发起调用之后，通过 HSFResponseFuture 中的 public static Object getResponse(long timeout) 来获取返回结果。

XML 中的配置：

```
<hsf:consumer id="demoApi" interface="com.alibaba.demo.api.DemoApi" version="1.1.2" >
<hsf:asyncallMethods>
<hsf:method name="ayncTest" type="future" />
</hsf:asyncallMethods>
</hsf:consumer>
```

代码示例如下。

单个调用异步处理：

```
//发起调用
demoApi.ayncTest();
// 处理业务
...
//直接获得消息（若无需获得结果，可以不用操作该步骤）
String msg=(String) HSFResponseFuture.getResponse(3000);
```

多个调用需要并发处理：

若是多个业务需要并发处理，可以先获取 future，存储起来，等调用完毕后再使用。

```
//定义集合
List<HSFFuture> futures = new ArrayList<HSFFuture>();
```

方法内进行并行调用：

```
//发起调用
demoApi.ayncTest();
//第一步获取 future 对象
HSFFuture future=HSFResponseFuture.getFuture();
futures.add(future);
//继续调用其他业务(同样采取异步调用)
HSFFuture future=HSFResponseFuture.getFuture();
futures.add(future);

// 处理业务
...

//获得数据并做处理
for (HSFFuture hsfFuture : futures) {
String msg=(String) hsfFuture.getResponse(3000);
//处理相应数据
...
}
```

泛化调用

通过泛化调用可以组合接口、方法、参数进行 RPC 调用，无需依赖任何业务 API。

步骤一：在消费者 XML 配置中加入泛化属性

```
<hsf:consumer id="demoApi" interface="com.alibaba.demo.api.DemoApi" generic="true"/>
```

说明：generic 代表泛化参数，true 表示支持泛化，false 表示不支持，默认为 false。

DemoApi 接口方法：

```
public String dealMsg(String msg);
public GenericTestDO dealGenericTestDO(GenericTestDO testDO);
```

步骤二：获取 demoApi 进行强制转换为泛化服务

导入泛化服务接口

import com.alibaba.dubbo.rpc.service.GenericService

获取泛化对象

- XML 加载方式

```
//若 WEB 项目中，可通过 Spring bean 进行注入后强制转换，这里是单元测试，所以采用加载配置文件方式
ClassPathXmlApplicationContext consumerContext = new
ClassPathXmlApplicationContext("hsf-generic-consumer-beans.xml");
//强制转换接口为 GenericService
GenericService svc = (GenericService) consumerContext.getBean("demoApi");
```

代码订阅方式

```
HSFApiConsumerBean consumerBean = new HSFApiConsumerBean();
consumerBean.setInterfaceName("com.alibaba.demo.api.DemoApi");
consumerBean.setGeneric("true"); // 设置 generic 为 true
consumerBean.setVersion("1.0.0");
consumerBean.init();
// 强制转换接口为 GenericService
GenericService svc = (GenericService) consumerBean.getObject();
```

步骤三：泛化接口

```
Object $invoke(String methodName, String[] parameterTypes, Object[] args) throws GenericException;
```

接口参数说明：

methodName：需要调用的方法名称。

parameterTypes：需要调用方法参数的类型。

args：需要传输的参数值。

步骤四：泛化调用

String 类型参数

```
svc.$invoke("dealMsg", new String[] { "java.lang.String" }, new Object[] { "hello" })
```

对象参数，服务端和客户端需要保证相同的对象

```
// 第一步构造实体对象 GenericTestDO，该实体有 id、name 两个属性
GenericTestDO genericTestDO = new GenericTestDO();
genericTestDO.setId(1980l);
genericTestDO.setName("genericTestDO-tst");
// 使用 PojoUtils 生成二方包 pojo 的描述
Object comp = PojoUtils.generalize(genericTestDO);
// 服务泛化调用
svc.$invoke("dealGenericTestDO",new String[] { "com.alibaba.demo.generic.domain.GenericTestDO" },
new Object[] { comp });
```

调用链路 Filter 扩展

下载 Demo。

基础接口

```
public interface ServerFilter extends RPCFilter {
}

public interface ClientFilter extends RPCFilter {
}

public interface RPCFilter {

ListenableFuture<RPCResult> invoke(InvocationHandler invocationHandler, Invocation invocation) throws
Throwable;

void onResponse(Invocation invocation, RPCResult rpcResult);

}
```

实现步骤

1. 实现 ServerFilter 进行服务端拦截。
2. 实现 ClientFilter 进行客户端拦截。
3. 业务通过标准的 META-INF/services/com.taobao.hsf.invocation.filter.RPCFilter 文件来注册 Filter。

实现示例

```
    import com.taobao.hsf.invocation.Invocation;
import com.taobao.hsf.invocation.InvocationHandler;
import com.taobao.hsf.invocation.RPCResult;
import com.taobao.hsf.invocation.filter.ServerFilter;
import com.taobao.hsf.util.PojoUtils;
import com.taobao.hsf.util.concurrent.ListenableFuture;


public class HSFServerFilter implements ServerFilter {
public ListenableFuture<RPCResult> invoke(InvocationHandler invocationHandler, Invocation invocation) throws
Throwable {
//process args
String[] sigs = invocation.getMethodArgSigs();
Object [] args = invocation.getMethodArgs();

System.out.println("#### intercept request");
for(String sig : sigs) {
System.out.print(sig);
System.out.print(";");
}
System.out.println();

for(Object arg : args) {
System.out.println(PojoUtils.generalize(arg));
System.out.print(";");
}
System.out.println();


return invocationHandler.invoke(invocation);
}

public void onResponse(Invocation invocation, RPCResult rpcResult) {
System.out.println("#### intercept response");
Object resp = rpcResult.getHsfResponse().getAppResponse();
System.out.println(PojoUtils.generalize(resp));
}


}
```

配置 META-INF/services/com.taobao.hsf.invocation.filter.RPCFilter

```
com.alibaba.edas.carshop.itemcenter.filter.HSFServerFilter
```

运行效果

```
#### intercept request
long
1111
```

intercept response

```
{itemId=1, itemName=Mercedes Benz, class=com.alibaba.edas.carshop.itemcenter.Item}
```

可选的 Filter

在一些场景下，您定制了 filter，但是只希望在某些服务上使用，这时可以使用可选的 Filter。具体做法是在对应的 Filter上 增加 @Optional 注解，如下：

```
```
@Optional
@Name("HSFOptionalServerFilter")
public class HSFOptionalServerFilter implements ServerFilter {
public ListenableFuture<RPCResult> invoke(InvocationHandler invocationHandler,
Invocation invocation) throws Throwable {
System.out.println("#### HSFOptionalServerFilter intercept request");
return invocationHandler.invoke(invocation);
}

public void onResponse(Invocation invocation, RPCResult rpcResult) {
System.out.println("#### HSFOptionalServerFilter intercept response");
}
}
```
```

当指定服务需要使用该 Filter 时，只需要在配置的 Bean 上声明即可，配置如下：

```
    <bean class="com.taobao.hsf.app.spring.util.HSFSpringProviderBean">
<property name="serviceInterface" value="com.alibaba.middleware.hsf.guide.api.service.OrderService" />
<property name="version" value="1.0.0" />
<property name="group" value="HSF" />
<property name="includeFilters">
<list>
<value>HSFOptionalServerFilter</value>
<value>NoFilter</value>
</list>
</property>
<property name="target" ref="orderService" />
</bean>
```

上述配置的服务，将会使用所有的非 @Optional 修饰的 ServerFilter ，并且会包括 HSFOptionalServerFilter 和 NoFilter，而 HSFOptionalServerFilter 的名称是来自于对应的 Filter 配置上的 @Name 修饰。

```
如果无法找到该名称的 Filter ，只会提醒您，但是不会导致您无法启动或者运行。
```

单元测试

在测试环境中，有两种方式做单元测试。

方式一 通过 LightApi 代码发布和订阅服务

方式二 通过 XML 配置发布订阅服务

相关样例请下载 Demo。

方式一 通过 LightApi 代码发布和订阅服务

在 Maven 中添加 LightApi 依赖。

```
 <dependency>
<groupId>com.alibaba.hsf</groupId>
<artifactId>LightApi</artifactId>
<version>1.0.5</version>
</dependency>
```

注意: 请使用 *1.0.5* 或以上版本的 LightApi，否则可能遇到 hsf: can not load class {com.taobao.hsf.address.AddressService} after all phase的错误。

创建 ServiceFactory。

这里需要设置 Pandora 的地址，参数是 SAR 包所在目录。如果 SAR 包地址是 /Users/Jason/Work/AliSoft/PandoraSar/DevSar/taobao-hsf.sar，则参数如下：

```
private static final ServiceFactory factory =
ServiceFactory.getInstanceWithPath("/Users/Jason/Work/AliSoft/PandoraSar/DevSar");
```

通过代码进行发布和订阅服务。

```
 // 进行服务发布（若有发布者，无需再在这里写）
factory.provider("helloProvider")// 参数是一个标识，初始化后，下次只需调用 provider("helloProvider")即可提供对应服务
.service("com.alibaba.edas.unit.service.UnitTestService")// 接口全类名
.version("1.0.0")// 版本号
.impl(new UnitTestServiceImpl())// 对应的服务实现
.publish();// 发布服务，至少要调用 service()和 version()才可以发布服务

// 进行服务消费
factory.consumer("helloConsumer")// 参数是一个标识，初始化后，下次只需调用 consumer("helloConsumer")即可直接提供对应服务
.service("com.alibaba.edas.unit.service.UnitTestService")// 接口全类名
.version("1.0.0")// 版本号
.subscribe();
factory.consumer("helloConsumer").sync();// 同步等待地址推送，最多6秒。
UnitTestService log4jService = (UnitTestService) factory.consumer("helloConsumer").subscribe();// 用 ID
```

```
获取对应服务，subscribe()方法返回对应的接口
// 调用服务方法
System.out.println("bean -> msg rec success:-"+log4jService.print());
```

方式二 通过 XML 配置发布订阅服务

编写好 HSF 的 XML 配置。

通过代码方式加载配置文件。

```
//XML 方式加载服务提供者
new ClassPathXmlApplicationContext("hsf-provider-beans.xml");
//XML 方式加载服务消费者
ClassPathXmlApplicationContext consumerContext=new ClassPathXmlApplicationContext("hsf-
consumer-beans.xml");
//获取 Bean
UnitTestXMLConsumer unitTestXMLConsumer=(UnitTestXMLConsumer)
consumerContext.getBean("unitTestConsumer");
//服务调用
unitTestXMLConsumer.testUnitProvider();
```

将使用 Dubbo 开发的应用迁移到 HSF（不推荐）

您可以通过更改 Dubbo 应用配置、配置多注册中心和将 JAR 转换成 WAR 等步骤将使用 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，所以新用户不推荐使用这种开发方式。

原生 Dubbo 框架下的应用开发请参考使用 Spring Boot 开发 Dubbo 应用。

更改 Dubbo 应用配置

目前 Dubbo 应用（包括服务提供者和服务消费者）在 EDAS 中支持两种配置的方式：XML 文件配置和注解配置。本文提供这两种方式的配置示例。

XML 文件配置方式

以下是 Dubbo XML 配置示例，设置正确则不需要做修改即可直接放入 EDAS 中运行。

通过 XML 文件配置服务生产者

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
<dubbo:application name="edas-dubbo-demo-provider" ></dubbo:application>
<bean id="demoProvider" class="com.alibaba.edas.dubbo.demo.provider.DemoProvider" ></bean>
<dubbo:registry address="zookeeper://127.0.0.1:2181" ></dubbo:registry>

<dubbo:protocol name="dubbo" port="20880" threadpool="cached"
threads="100" ></dubbo:protocol>

<dubbo:service delay="-1" interface="com.alibaba.edas.dubbo.demo.api.DemoApi"
ref="demoProvider" version="1.0.0" group="dubbogroup" retries="3" timeout="3000"></dubbo:service>

</beans>
```

注意：

- 可选配置包括 threadpool、threads、delay、version、retries、timeout，其他均为必选配置。配置项可以任意调换位置。
- Dubbo 的 RPC 协议支持多种方式，如 RMI， hessian 等，但是目前 EDAS 的适配方案只支持了 Dubbo 协议，如：<dubbo:protocol name="dubbo" port=“20880” >，否则会引起类似于：” com.alibaba.dubbo.config.ServiceConfig service [xx.xx.xxx] contain xx protocal，HSF not supportted” 的错误发生。

通过 XML 文件配置服务消费者

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
<dubbo:application name="edas-dubbo-consumer" />
<dubbo:registry address="zookeeper://127.0.0.1:2181" />
<dubbo:reference id="demoProviderApi"
interface="com.alibaba.edas.dubbo.demo.api.DemoApi" version="1.0.0" group="dubbogroup" lazy="true"
loadbalance="random">
<!-- 指定某个方法不用等待返回值 -->
<dubbo:method name="sayMsg" async="true" return="false" />
</dubbo:reference>
<bean id="demoConsumer" class="com.alibaba.edas.dubbo.demo.consumer.DemoConsumer"
init-method="reviceMsg">
<property name="demoApi" ref="demoProviderApi"></property>
</bean>
```

```
</beans>
```

注意：

- 可选配置包括 version、group、lazy、loadbalance、async、return，其他选项为必须。配置项可以任意调换位置。
- 注册中心在 EDAS 中是不生效的，所有 Dubbo 的服务会自动注册到 EDAS 的配置中心，您无需关心。
- 由于 Dubbo 配置文件消费者可以指定多个分组，而 EDAS 目前只能通过 group 属性配置一个分组，无法指定多个分组。
- 当有业务需要在程序启动过程中加载服务，则需要设置 lazy=true，进行延迟加载

注解配置方式

从 EDAS-Container 3.0 版本开始，EDAS 已经支持 Dubbo 原生注解，您无需进行注解转换 XML 即可使用 EDAS 服务。

兼容说明：

- 服务发布注解： @Service
- 服务订阅注解： @Reference

支持属性： group、version、timeout

使用方式： 在创建容器的时候，选择最新 EDAS-Container 版本即可。

配置多注册中心

有时候需要的服务不在同一个 EDAS、ZooKeeper 注册中心上，则需要在 Dubbo 配置文件中配置多个注册中心。例如：有些服务来不及在北京部署，只在上海部署，而北京的其他应用需要引用此服务，就可以将服务同时注册到两个注册中心。

多订阅指 Dubbo/HSF 应用去消费一个服务时，可以同时订阅 EDAS、ZooKeeper 注册中心中的服务。

在应用中加入 1.5.1 及后续版本的 edas-sdk 依赖。

```
<dependency>
<groupId>com.alibaba.edas</groupId>
<artifactId>edas-sdk</artifactId>
<version>1.5.1</version>
</dependency>
```

指定 ZooKeeper 注册/订阅中心地址。

指定方式主要包含以下两种：

环境变量指定（支持 HSF 、Dubbo 应用）：

-Dhsf.registry.address=zookeeper://IP 地址:端口

XML 指定方式（只支持 HSF 应用）：

<hsf:registry address="zookeeper://IP 地址:端口" />

指定 ZooKeeper 地址后，Dubbo 应用默认会启动双注册和订阅。HSF 应用若需要启用双注册/订阅，还需要设置调用参数 invokeType 。

设置 HSF 应用多注册中心的调用参数。

- 只注册/订阅 ConfigServer 中的服务：invokeType="hsf"
- 只注册/订阅 ZooKeeper 中的服务: invokeType="dubbo"
- 双订阅/注册: invokeType="hsf，dubbo"

创建应用时，需要选择不低于 3.0 版本的 EDAS-Container，然后上传启动即可。

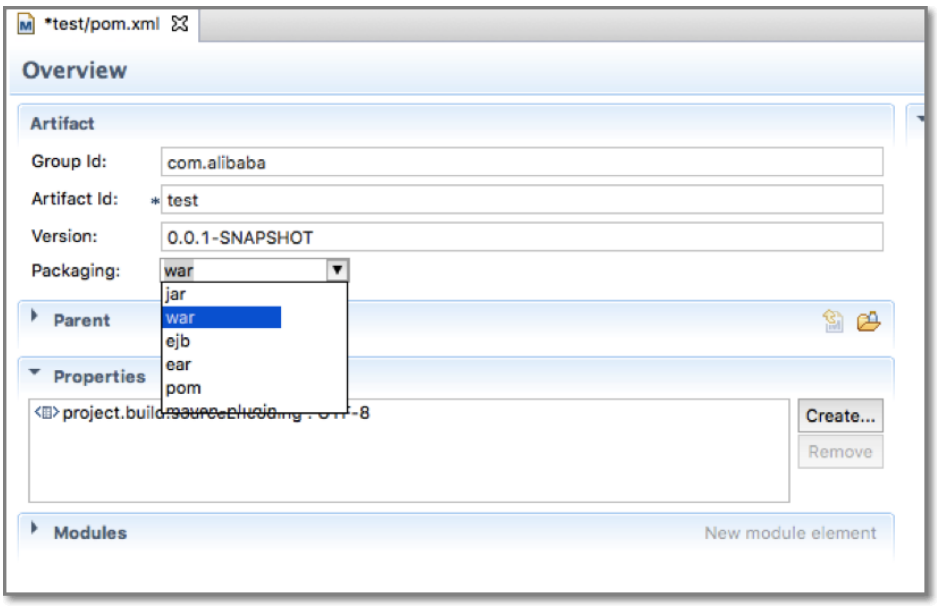
说明：

- **分组问题**：Dubbo 服务分组默认为空，EDAS 里服务分组默认是 HSF。因此务必修改 Dubbo 服务分组。
- **版本问题**：Dubbo 服务版本默认是0.0.0，EDAS 里面服务版本默认是1.0.0，因此务必修改 Dubbo 服务版本。
- **订阅成功，调用不成功**：EDAS 里面存在鉴权，故若要让 Dubbo 调用成功，则需关闭 EDAS 里面服务鉴权，参数为-DneedAuth=false，需要在 JVM 参数中设置，重启即可。

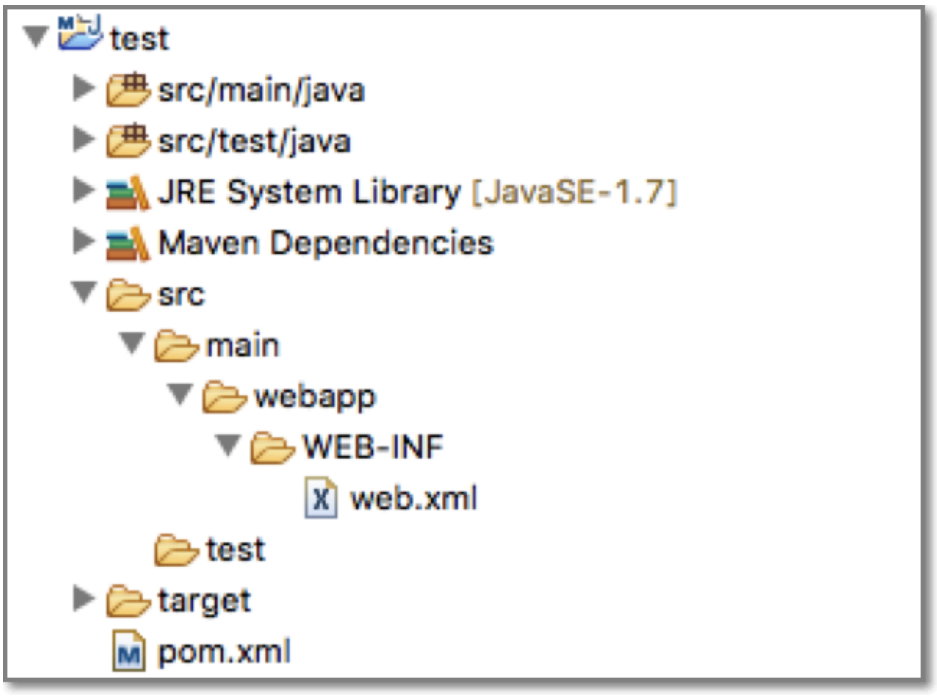
将应用程序从 JAR 转换为 WAR

HSF 只支持 WAR 形式的 Web 项目，所以如果你的应用程序是以 JAR 包发布的，需要先转换为 WAR。下面以 Maven 项目为例说明。

在pom.xml文件中将 packaging 由 JAR 改为 WAR。



如果没有web.xml，则需要增加一个web.xml文件配置。



在web.xml中配置加载配置文件。

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:hsf-provider-beans.xml</param-value>
</context-param>

<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
```

```
</listener>
<listener>
<listener-class>
org.springframework.web.context.request.RequestContextListener
</listener-class>
</listener>
```

检查 Dubbo 和 HSF 的配置兼容性

对照下表，检查 Dubbo 配置文件中的服务属性与 HSF 的兼容情况。检查完配置兼容性之后，即可按照前面文档介绍的内容进行应用的调试与发布。

功能特性	Dubbo 配置参数	兼容情况说明	错误提示	EDAS 是否支持
超时	timeout			支持
延迟暴露	delay			支持
线程模型	dispatcher= “all” threadpool= “fixed” threads= “100”			支持
回声测试				支持
延迟	lazy= “true”	默认开启		支持
连接				
本地调用	protocol= “injvm”			支持
隐式传参				支持
并发控制	actives= “10” executes= “10 ”	已经实现 EDAS 控制台可视化配置，限流降级-限流规则-限流粒度		支持
连接控制	accepts= “10” connections=” 2”	已经实现 EDAS 控制台可视化配置，限流降级-限流规则-限流粒度		支持
服务降级		已经实现 EDAS 控制台可视化配置，限流降级-降级规则		支持
集群容错	retries/cluster	支持 retries	无报错	部分支持
负载均衡	loadbalance	默认 random	无报错	部分支持
服务分组	group	不支持 * 配置	java.lang.IllegalStateException: hsf2 不支持同时	部分支持

			消费多个分组!	
多版本	version	不支持 * 配置	[HSF-Consumer] 未找到需要调用的服务的目标地址	部分支持
异步调用	async=“true” return=” false ”	return 参数无效	无报错	部分支持
启动时检查	check	EDAS 默认是启动不检查	无报错	默认支持启动不检查
多协议		只支持 Dubbo 协议	com.alibaba.dubbo.config.ServiceConfig 服务 [com.alibaba.demo.api.DemoApi] 配置了 RMI 协议， HSF2 不支持	部分支持
路由规则		已经实现 EDAS 控制台可视化配置，无需配置		支持
配置规则		已经实现 EDAS 控制台可视化配置，无需配置		支持
多注册中心				不支持
分组聚合	group= “aaa,bbb” merger= “true”	报错	java.lang.IllegalStateException: hsf2 不支持同时消费多个分组!	不支持
上下文信息		报错	Caused by: java.lang.UnsupportedOperationException: not support getInvocation method in hsf2	不支持

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。

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
- 配置轻量配置中心

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。

Demo 源码下载： sc-hsf-provider 、 sc-hsf-consumer。

创建服务提供者

创建一个 Maven 工程，命名为sc-hsf-provider。

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-hsf</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.boot</groupId>
<artifactId>spring-boot-starter-web</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>
```

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。

如果您的工程不想将parent设置为spring-boot-starter-parent，也可以通过如下方式添加dependencyManagement，设置scope=import，来达到依赖版本管理的效果。

```
<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>
```

定义服务接口，创建一个接口类com.aliware.edas.EchoService。

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 EchoService {
String echo(String string);
}
```

接口com.aliware.edas.EchoService提供了一个echo方法，也可以理解成服务com.aliware.edas.EchoService将提供一个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 = EchoService.class, serviceVersion = "1.0.0")
public class EchoServiceImpl implements EchoService {
@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.aliware.edas.EchoService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-provider
server.port=18081

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(ServerApplication.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();
}
}
```

服务提供者属性列表

属性	是否必配	描述	类型	默认值
serviceInterface	是	服务对外提供的接口	Class	java.lang.Object
serviceVersion	否	服务的版本号	String	1.0.0.DAILY
serviceGroup	否	服务的组名	String	HSF
clientTimeout	否	该配置对接口中的所有方法生效，但是如果客户端通过 methodSpecials 属性对某方法配置了超时时间，则该方法的超时时间以客户端配置为准。其他方法不受影响，还是以服务端配置为准（单位 ms）	int	-1
corePoolSize	否	单独针对这个服务设置最小活跃线程数，从公用线程池中划分出来	int	0
maxPoolSize	否	单独针对这个服务设置最大活跃线程数，从公用线程池中划分出来	int	0
delayedPublish	否	是否延迟发布	boolean	false
includeFilters	否	用户可选的自定义过滤器	String[]	空
enableTXC	否	是否开启分布式事务 GTS	boolean	false

serializeType	否	服务接口序列化类型，hessian 或者 java	String	hessian
supportAsynCall	否	是否支持异步调用	String	false

服务创建及发布限制

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	aliware	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	是，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）

创建服务消费者

本示例中，将创建一个服务消费者，通过HSFProvider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sc-hsf-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。

```
 <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-hsf</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.boot</groupId>
<artifactId>spring-boot-starter-web</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>
```

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.aliware.edas.EchoService。

```
public interface EchoService {
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 EchoService echoService;

@RequestMapping(value = "/hsf-echo/{str}", method = RequestMethod.GET)
public String echo(@PathVariable String str) {
return echoService.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-consumer
server.port=18082

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();
}
}
```

服务消费者属性列表

属性	是否必配	描述	类型	默认值
serviceGroup	否	服务的组名	String	HSF
serviceVersion	否	服务的版本号	String	1.0.0.DAILY
clientTimeout	否	客户端统一设置接口中所有方法的超时时间（单位 ms）	int	-1
generic	否	是否支持泛化调用	boolean	false
addressWaitTime	否	同步等待服务注册中心（ConfigServer）推送服务提供	int	3000

		者地址的时间（单位 ms）		
proxyStyle	否	代理方式（JDK 或 Javassist）	String	jdk
futureMethods	否	设置调用此服务时需要采用异步调用的方法名列表以及异步调用的方式，默认为空，即所有方法都采用同步调用	String[]	空
consistent	否	负载均衡是否使用一致性哈希	String	空
methodSpecials	否	配置方法级的超时时间、重试次数、方法名称	com.alibaba.boot.hsf.annotation.HSFConsumer.ConsumerMethodSpecial[]	空

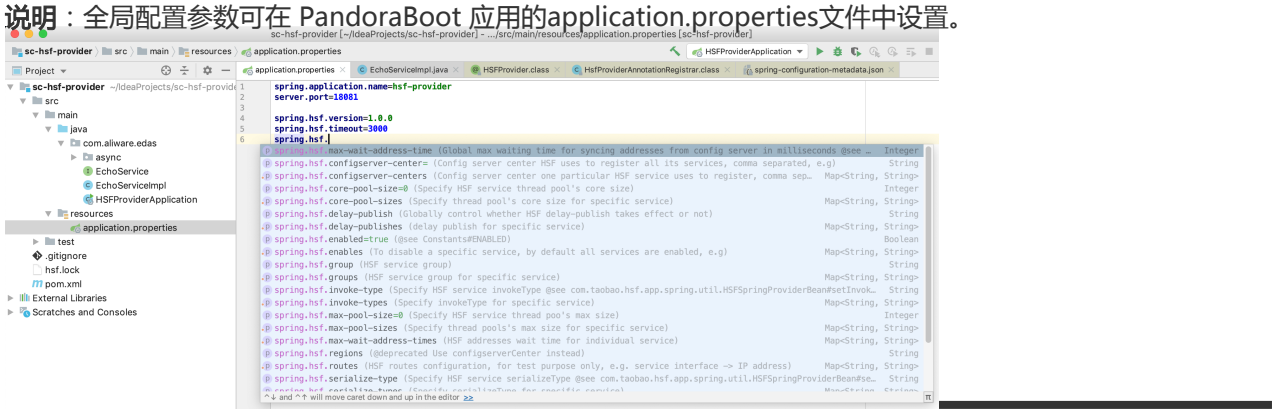
服务提供者和消费者全局配置参数列表

属性	是否必配	描述	类型	默认值
spring.hsf.version	否	服务的全局版本号，还可以使用spring.hsf.versions.<完整的服务接口名>=<单独为该服务接口设置的版本号，String类型>，例如*spring.hsf.versions.com.alibaware.edas.EchoService="1.0.0"*为具体某个服务设置版本号。	String	1.0.0.DAILY
spring.hsf.group	否	服务的全局组名，还可以使用spring.hsf.groups.<完整的服务接口名>=<单独为该服务接口设置的组名，String类型>为具体某个服务设置组名。	String	HSF
spring.hsf.timeout	否	服务的全局超时时间，还可以使用spring.hsf.time	Integer	无

		outs.<完整的服务接口名>=<超时时间，String类型>为具体某个服务设置超时时间。		
spring.hsf.max-wait-address-time	否	同步等待服务注册中心（ConfigServer）推送服务提供者地址的全局时间（单位 ms），还可以使用spring.hsf.max-wait-address-times.<完整的服务接口名>=<等待时间，String类型>为具体某个服务设置的等待服务注册中心（ConfigServer）推送服务提供者地址的时间。	Integer	3000
spring.hsf.delay-publish	否	服务延迟发布的全局开关，”true” or “false”，还可以使用spring.hsf.delay-publishes.<完整的服务接口名>=<是否延迟发布，String类型>为具体某个服务设置是否延迟。	String	无
spring.hsf.core-pool-size	否	服务的全局最小活跃线程数，还可以使用spring.hsf.core-pool-sizes.<完整的服务接口名>=<最小活跃线程数，String类型>单独为某服务设置最小活跃线程数。	int	无
spring.hsf.max-pool-size	否	服务的全局最大活跃线程数，还可以使用spring.hsf.max-pool-sizes.<完	int	无

		整的服务接口名>=<最大活跃线程数，String类型>单独为某服务设置最大活跃线程数。		
spring.hsf.serialize-type	否	服务的全局序列化类型，Hessian 或者 Java，还可以使用 spring.hsf.serialize-types.<完整的服务接口名>=<序列化类型>单独为某服务设置序列化类型。	String	无

说明：全局配置参数可在 PandoraBoot 应用的application.properties文件中设置。



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

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，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。

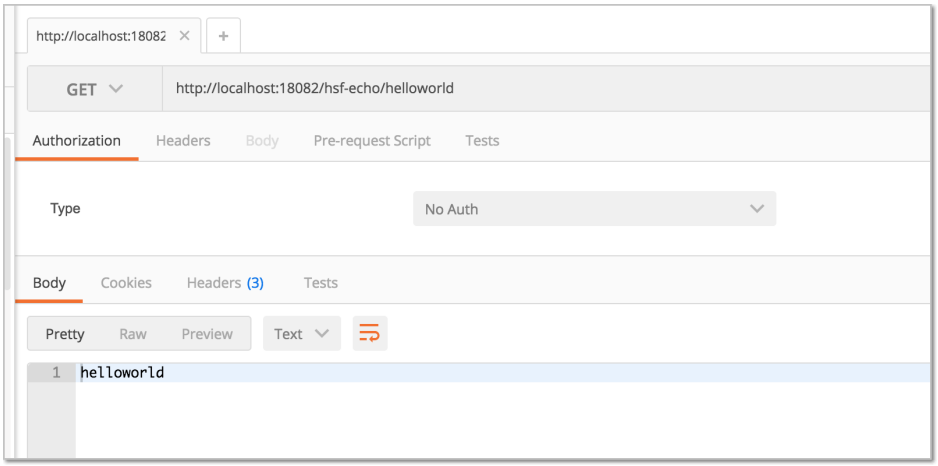
通过 Java 命令启动应用。

```
java -Djmenv.tbsite.net=127.0.0.1 -
Dpandora.location=/Users/{$username}/.m2/repository/com/taobao/pandora/taobao-
hsf.sar/dev-SNAPSHOT/taobao-hsf.sar-dev-SNAPSHOT.jar  -jar sc-hsf-provider-0.0.1-
SNAPSHOT.jar
```

说明：-Dpandora.location指定的路径必须是全路径，且必须放在sc-hsf-provider-0.0.1-SNAPSHOT.jar之前。

结果验证

启动服务，进行调用，可以看到调用成功。



异步调用

HSF 提供了两种类型的异步调用，Future 和 Callback。

在实现异步调用功能之前，先发布一个新的服务AsyncEchoService。

```
public interface AsyncEchoService {
String future(String string);
String callback(String string);
}
```

服务提供者实现AsyncEchoService，并通过注解发布。

```
@HSFProvider(serviceInterface = AsyncEchoService.class, serviceVersion = "1.0.0")
public class AsyncEchoServiceImpl implements AsyncEchoService {
@Override
public String future(String string) {
return string;
}

@Override
public String callback(String string) {
return string;
}
}
```

从这两点中可以看出，服务提供者与普通的发布没有任何区别。同样，之后的配置和应用启动流程也是一致的，详情请参见创建服务提供者中的内容。

说明：异步调用的逻辑修改都在消费者，提供者无需做任何修改。

Future

使用 Future 类型的异步调用的消费者，也是通过注解的方式将服务消费者的实例注入到 Spring 的 Context 中，并在@HSFConsumer注解的futureMethods属性中配置异步调用的方法名。

将AsyncEchoService的 Future 方法标记为 Future 类型的异步调用。

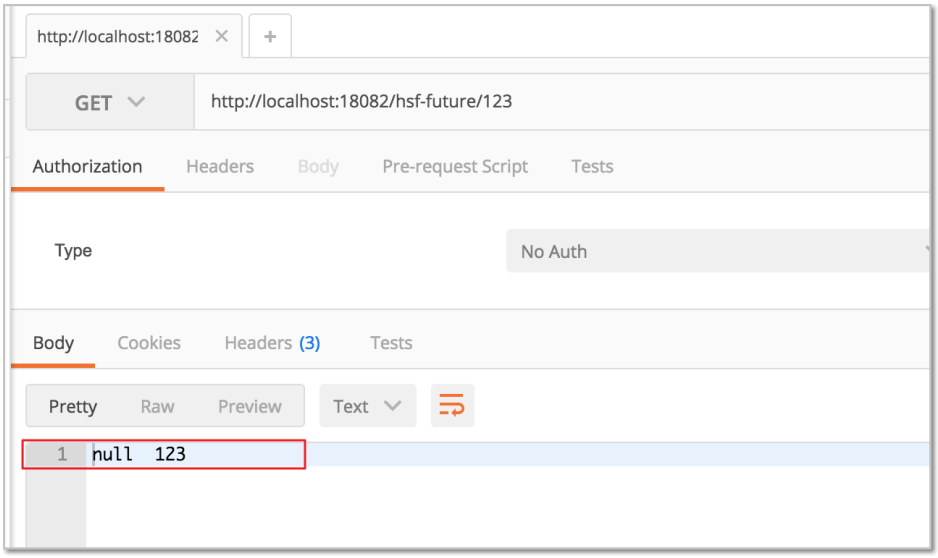
```
@Configuration
public class HsfConfig {
@HSFConsumer(serviceVersion = "1.0.0", futureMethods = "future")
private AsyncEchoService asyncEchoService;
}
```

方法在被标记成 Future 类型的异步调用后，同步执行时的方法返回值是 null，需要通过HSFResponseFuture来获取调用的结果。

通过TestAsyncController来进行演示，示例代码如下：

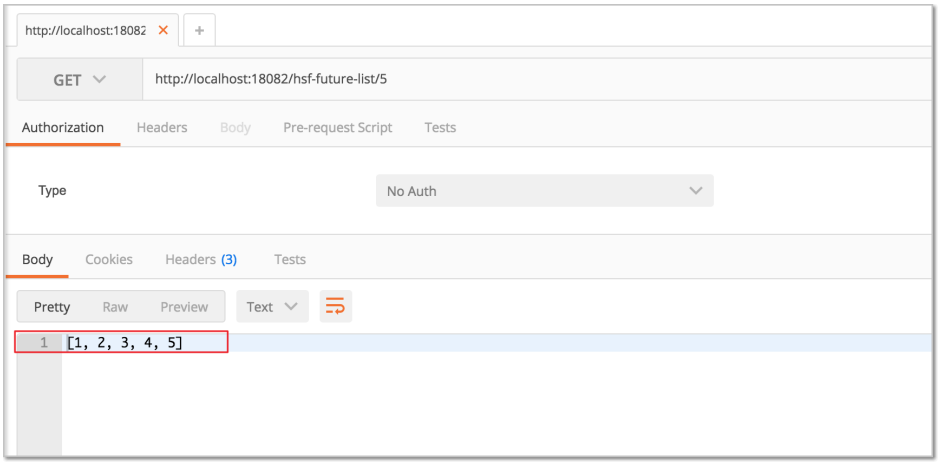
```
@RestController
public class TestAsyncController {

@Autowired
private AsyncEchoService asyncEchoService;

@RequestMapping(value = "/hsf-future/{str}", method = RequestMethod.GET)
public String testFuture(@PathVariable String str) {

String str1 = asyncEchoService.future(str);
String str2;
try {
HSFFuture hsfFuture = HSFResponseFuture.getFuture();
str2 = (String) hsfFuture.getResponse(3000);
} catch (Throwable t) {
t.printStackTrace();
str2 = "future-exception";
}
return str1 + " " + str2;
}
}
```

调用/hsf-future/123，可以看到str1的值为null，str2才是真实的调用返回值 123。



当服务中需要结合一批操作的返回值进行处理时，参考如下的调用方式。

```
@RequestMapping(value = "/hsf-future-list/{str}", method = RequestMethod.GET)
public String testFutureList(@PathVariable String str) {
try {

int num = Integer.parseInt(str);
List<String> params = new ArrayList<String>();
for (int i = 1; i <= num; i++) {
params.add(i + "");
}

List<HSFFuture> hsfFutures = new ArrayList<HSFFuture>();
for (String param : params) {
asyncEchoService.future(param);
hsfFutures.add(HSFResponseFuture.getFuture());
}

ArrayList<String> results = new ArrayList<String>();
for (HSFFuture hsfFuture : hsfFutures) {
results.add((String) hsfFuture.getResponse(3000));
}

return Arrays.toString(results.toArray());

} catch (Throwable t) {
return "exception";
}
}
```



Callback

使用 Callback 类型的异步调用的消费者，首先创建一个接口类HSFResponseCallback，并通过@Async注解进行配置。

```
 @AsyncOn(interfaceName = AsyncEchoService.class,methodName = "callback")
public class AsyncEchoResponseListener implements HSFResponseCallback{
@Override
public void onAppException(Throwable t) {
t.printStackTrace();
}

@Override
public void onAppResponse(Object appResponse) {
System.out.println(appResponse);
}

@Override
public void onHSFException(HSFException hsfEx) {
hsfEx.printStackTrace();
}
}
```

AsyncEchoResponseListener实现了HSFResponseCallback接口，并在@Async注解中分别配置interfaceName为AsyncEchoService.class、methodName为callback。

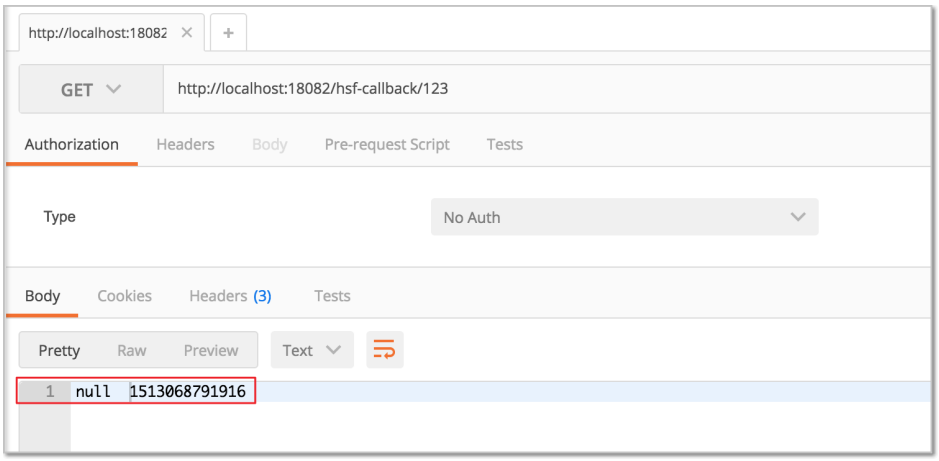
这样，就将AsyncEchoService的 callback 方法标记为 Callback 类型的异步调用。

通过TestAsyncController来进行演示，示例代码如下：

```
 @RequestMapping(value = "/hsf-callback/{str}", method = RequestMethod.GET)
public String testCallback(@PathVariable String str) {

String timestamp = System.currentTimeMillis() + "";
```

```
String str1 = asyncEchoService.callback(str);
return str1 + " " + timestamp;
}
```

执行调用，可以看到如下结果：



消费着将 callback 方法配置为 Callback 类型异步调用时，同步返回结果其实是 null。

结果返回之后，HSF 会调用 AsyncEchoResponseListener 中的方法，在 onAppResponse 方法中我们可以得到调用的真实返回值。

如果需要将调用时的上下文信息传递给 callback ，需要使用CallbackInvocationContext来实现。

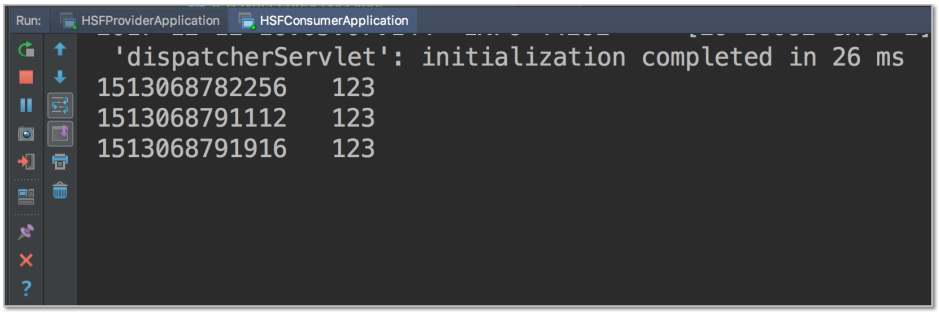
调用时的示例代码入下：

```
CallbackInvocationContext.setContext(timestamp);
String str1 = asyncEchoService.callback(str);
CallbackInvocationContext.setContext(null);
```

AsyncEchoResponseListener示例代码如下：

```
@Override
public void onAppResponse(Object appResponse) {
Object timestamp = CallbackInvocationContext.getContext();
System.out.println(timestamp + " " +appResponse);
}
```

我们可以在控制台中看到输出了 1513068791916 123，证明 AsyncEchoResponseListener 的 onAppResponse 方法通过 CallbackInvocationContext 拿到了调用前传递过来的 timestamp 的内容。



单元测试

spring-cloud-starter-hsf 的实现依赖于 Pandora Boot，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spring-boot-starter-test的依赖。

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
</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, EchoServiceTest.class })
@Component
public class EchoServiceTest {

/**
* 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)
EchoService echoService;

//普通的调用
@Test
public void testInvoke() {
TestCase.assertEquals("hello world", echoService.echo("hello world"));
}
//泛化调用
@Test
public void testGenericInvoke() {
GenericService service = (GenericService) echoService;
```

```
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() {
EchoService mock = Mockito.mock(EchoService.class, AdditionalAnswers.delegatesTo(echoService));
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。

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

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。

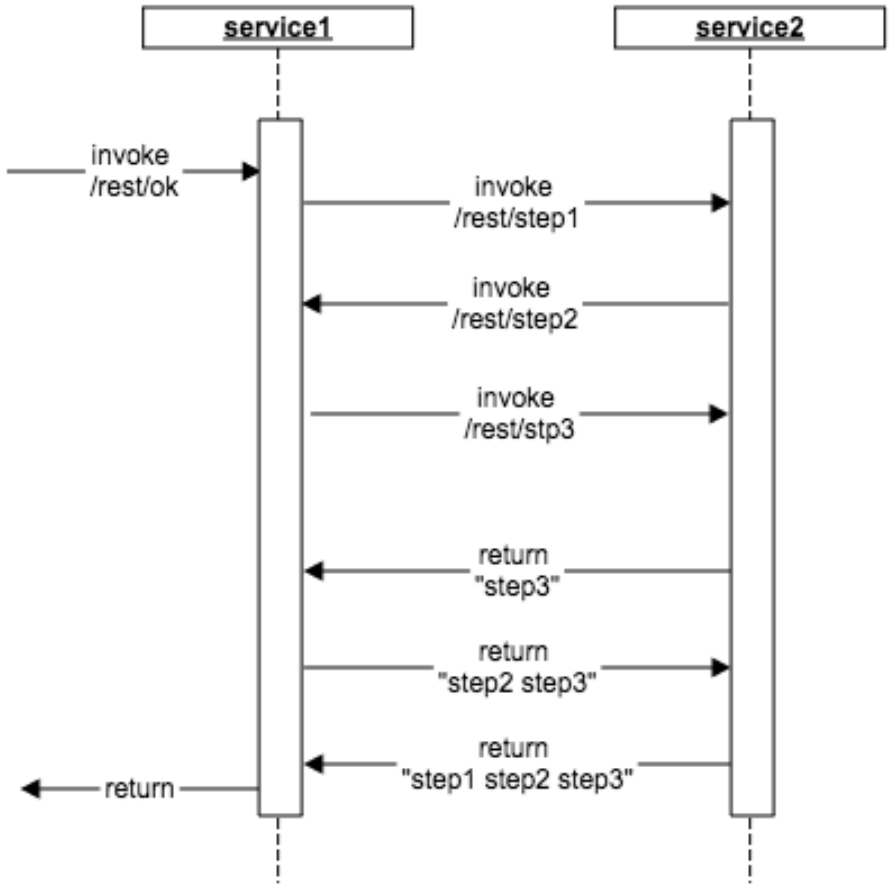
全链路跟踪示例

源码

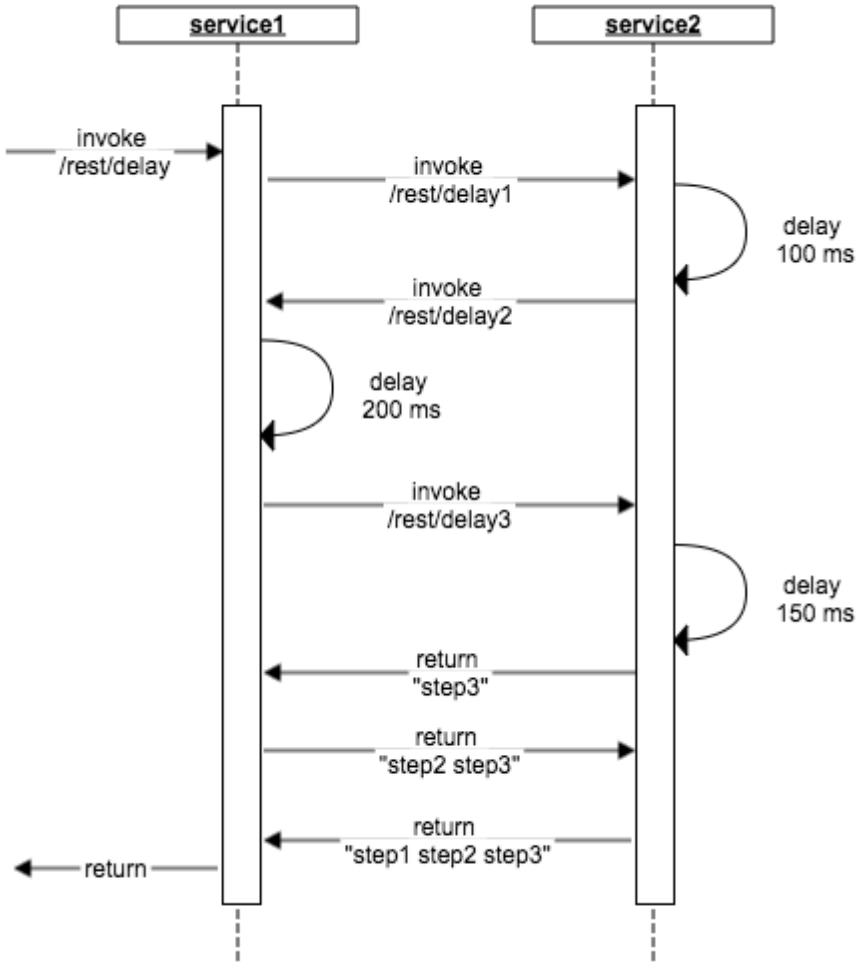
为了演示如何使用全链路跟踪功能 ，这里提供两个应用 Demo：service1 和 service2。

service1 用作入口的服务，提供了三个场景演示的入口：

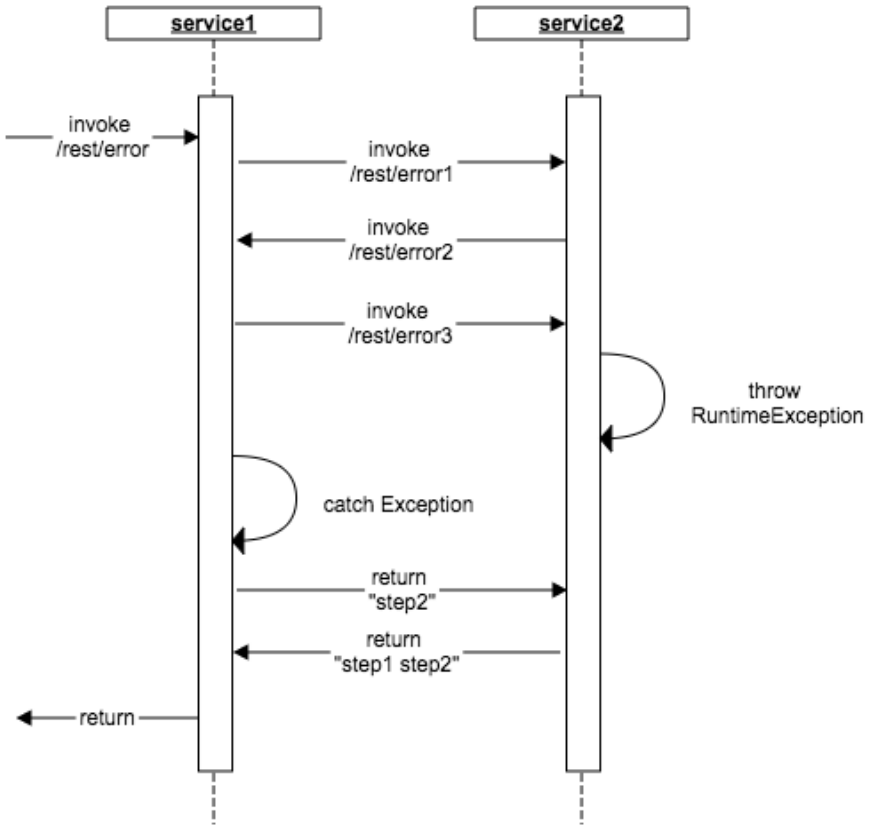
/rest/ok，对应正常的调用



/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**。

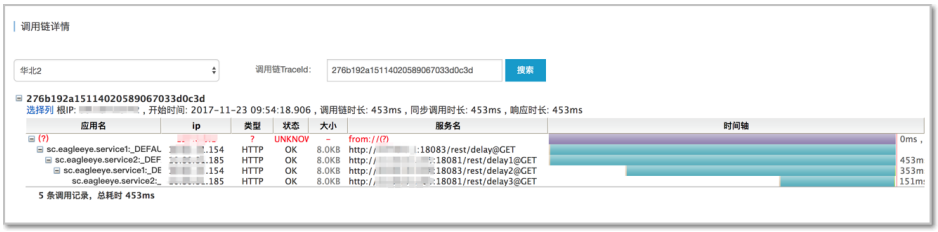
在服务监控页面单击**提供的 RPC 服务**页签，然后单击**查看调用链**。

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
3.5.3	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. 其他Pandora插件版本的更新
3.5.1	2018-11-28	52	3.5.0	1. 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	1. 升级ONS到1.7.8-Eagle Eye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	1. 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，timeout 值为 0 导致超时异常。 2. EDAS 在使用 Dubbo 时，RpcConte

				xt 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. Diamond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ConfigCenter 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. Eagle Eye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	1. 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. 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	1. 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。

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。

工具开发

IntelliJ IDEA 插件端云互联

对于开发者来说，存在本地应用与云端应用需要相互调用的需求，但搭建 VPN 打通本地与云端网络方式比较麻烦。现 EDAS 提供基于 IntelliJ IDEA 插件更加轻量级的端云互联解决方案，通过简单的配置即可进行本地与远端应用通信。

限制条件

在 IntelliJ IDEA 中使用 Cloud Toolkit 插件对 EDAS 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。

前提条件

安装 IntelliJ Idea，请选择 2018.3（含）以上的版本;

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。

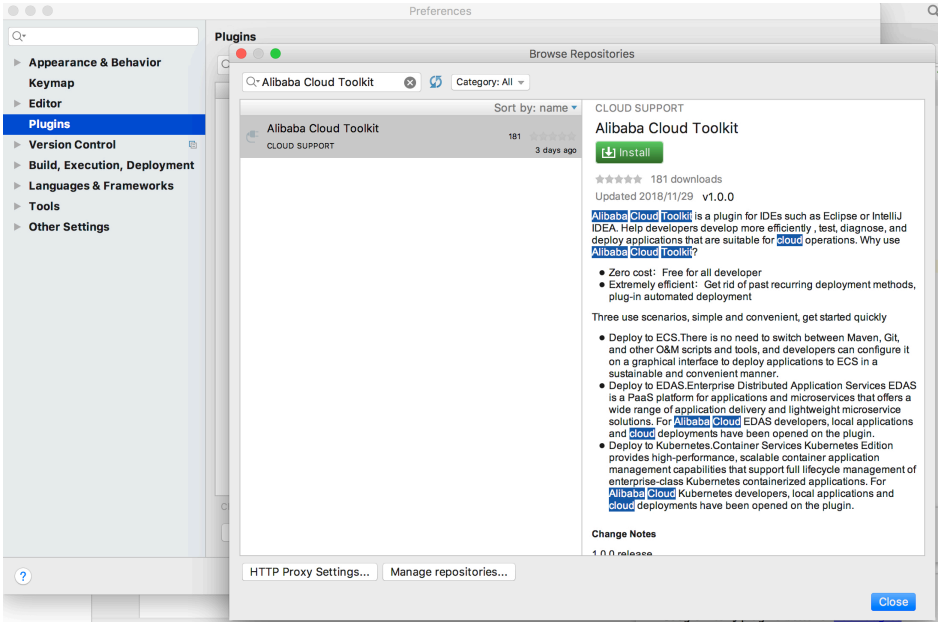
注意：SSH 通道需要使用密码方式登录，暂不支持使用密钥对登录。

步骤一：安装 Cloud Toolkit

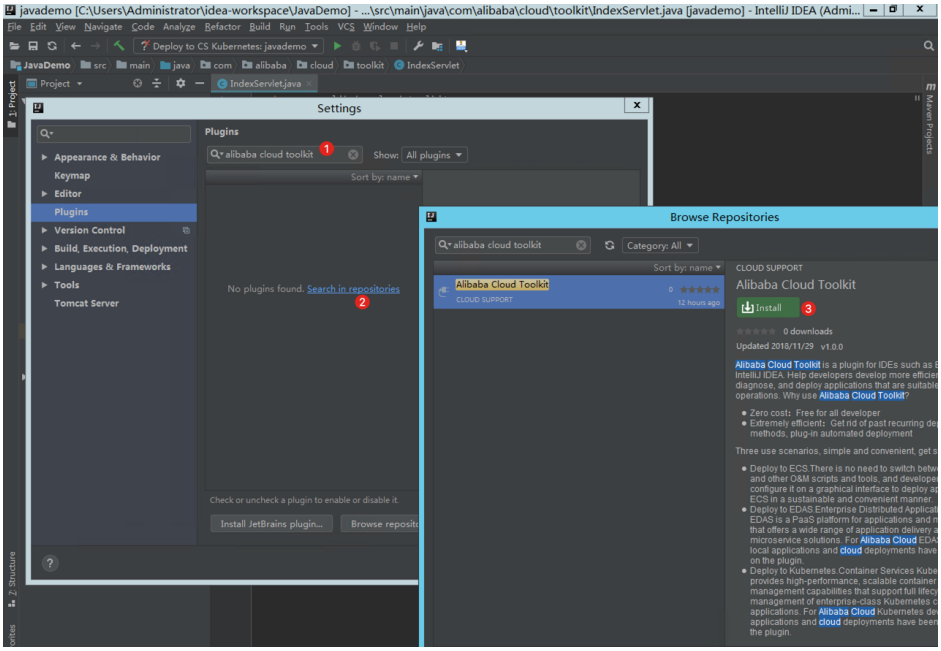
启动 IntelliJ IDEA。

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

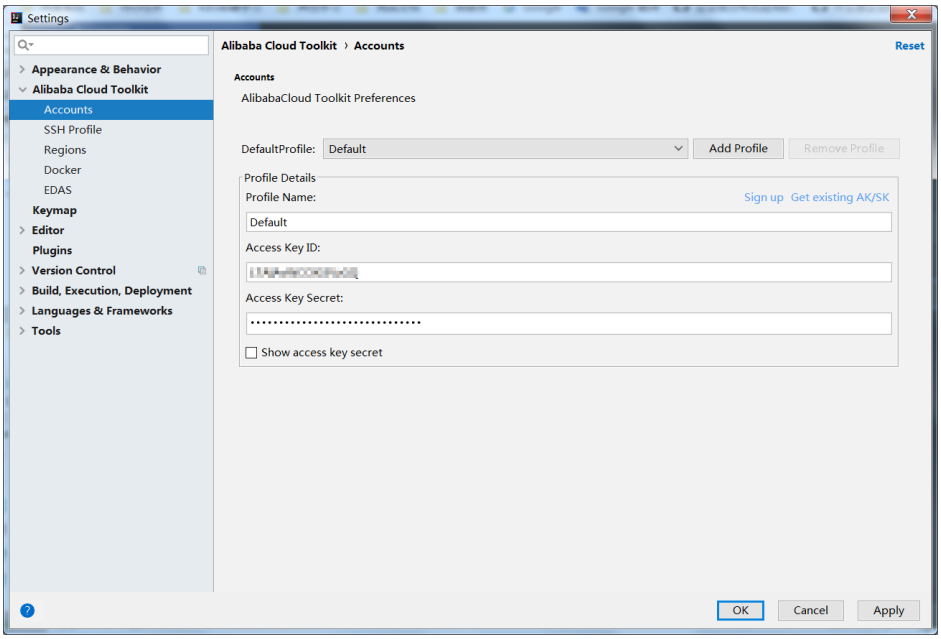
在安装完 Alibaba Cloud Toolkit 后，您需使用 Access Key ID 和 Access Key Secret 来配置 Cloud Toolkit 的账号。

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。



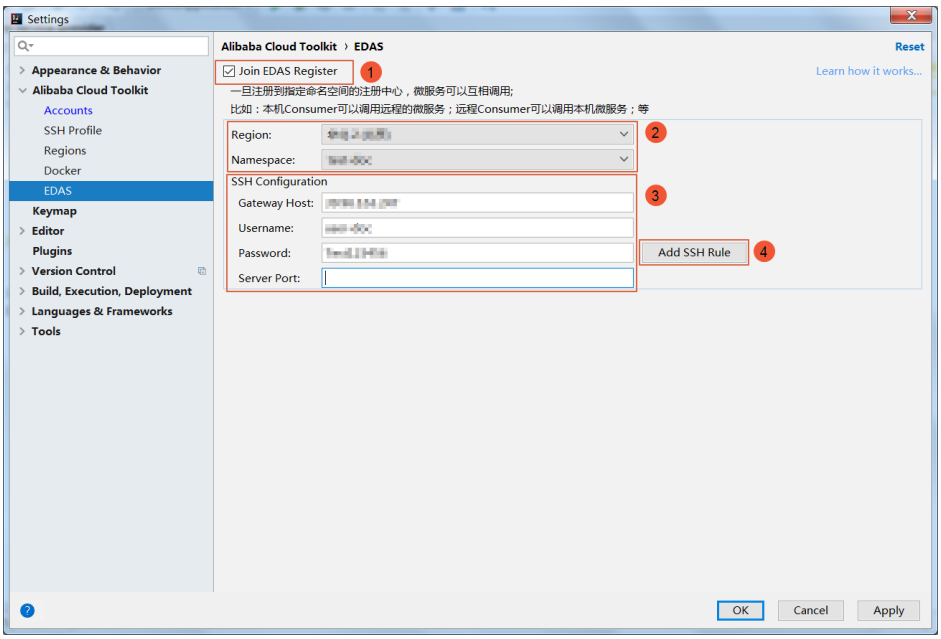
如果您已经注册过阿里云账号，在 **Accounts** 界面中单击 **Get existing AK/SK**，进入阿里云登录页面。用已有账号登录后，跳转至安全信息管理页面，获取 **Access Key ID** 和 **Access Key Secret**。

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。

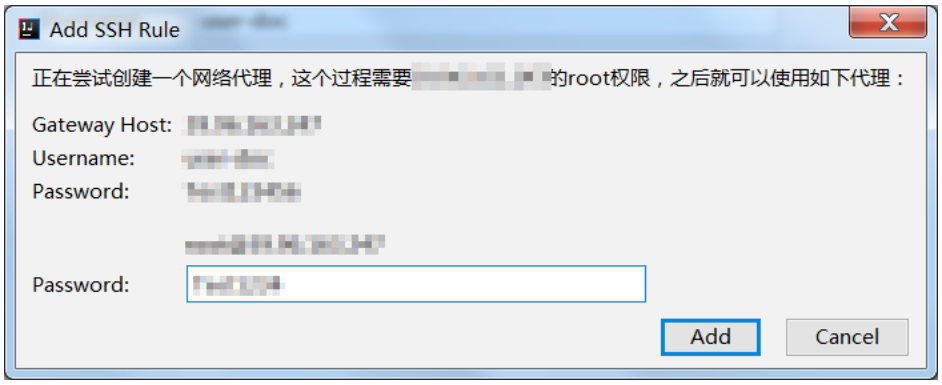
Server Port: Spring Boot 应用需添加该应用的服务端口，其他类型应用不需要填写。

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。

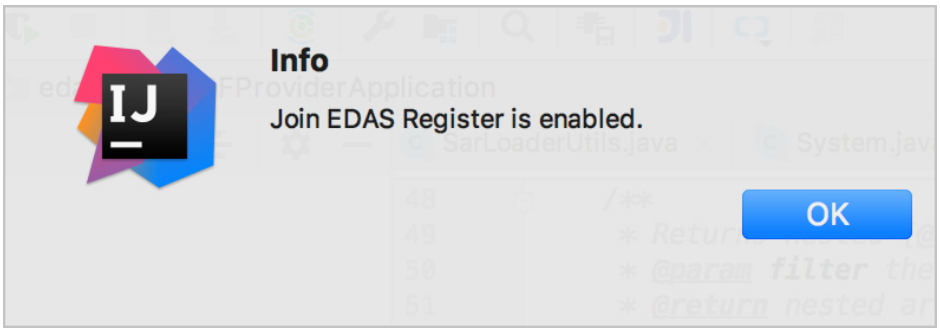
说明：如果使用 EDAS 专有云企业版，还需要按以下步骤在 Cloud Toolkit 中配置 Endpoint。Endpoint 请联系 EDAS 技术支持获取。

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

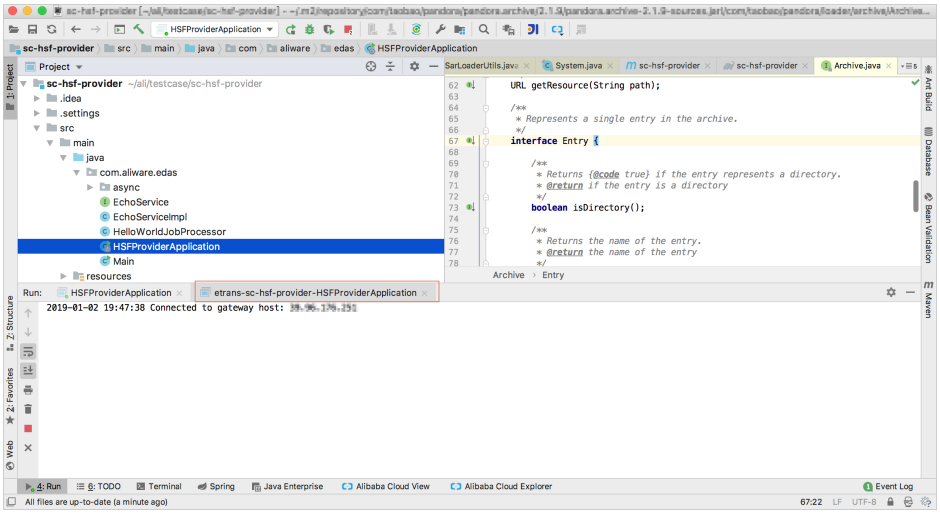
在 **Endpoint** 界面中设置 Endpoint，配置完成后，单击 **Apply and Close**。

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：



常见问题

本文档模块介绍了使用插件进行端云互联时的常见问题的总结。

SOCKS proxy:connect timed out

如果您之前使用轻量级配置中心，那么现在启用端云互联插件后，需要把原来轻量级配置中心的相关配置删除，否则会出现连接超时的问题。

Spring Cloud 应用：请去掉以下配置项：

```
spring.cloud.alicloud.acm.server-list=127.0.0.1
spring.cloud.alicloud.acm.server-port=8080
spring.cloud.alicloud.ans.server-list=127.0.0.1
spring.cloud.alicloud.ans.server-port=8080
```

HSF 应用：请去掉以下的 Hosts 绑定：

```
192.168.1.100  jmenv.tbsite.net
```

注册中心拒绝请求（code:403，msg: access denied）

注册中心会通过时间戳来保证服务注册的安全性，如果本地的系统时间不准确会导致云端注册中心拒绝请求，请校准系统时间并进行重试。

更多信息

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**。

问题反馈

如果您在使用 Cloud Toolkit 过程中有任何疑问，欢迎您扫描下面的二维码加入钉钉群进行反馈。

Eclipse 插件端云互联

对于开发者来说，存在本地应用与云端应用需要相互调用的需求，但搭建 VPN 打通本地与云端网络方式比较麻烦。现 EDAS 提供基于 Eclipse 插件更加轻量级的端云互联解决方案，通过简单的配置即可进行本地与远端应用通信。

限制条件

在 IntelliJ IDEA 中使用 Cloud Toolkit 插件对 EDAS 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，网关桥接的 ECS 实例要跟远端服务应用在同一集群内。

ECS 集群和 Swarm 集群中的应用进行端云互联时，网关桥接的 ECS 实例要跟远端服务应用在同一 VPC 内。

前提条件

- 下载并安装 Eclipse IDE 4.5.0 或更高版本；

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。

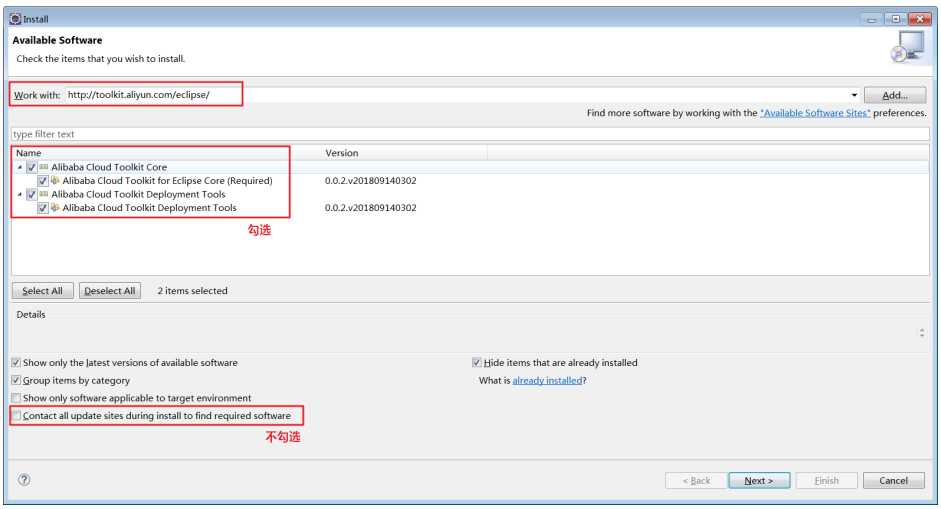
步骤一：安装 Cloud Toolkit

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/。

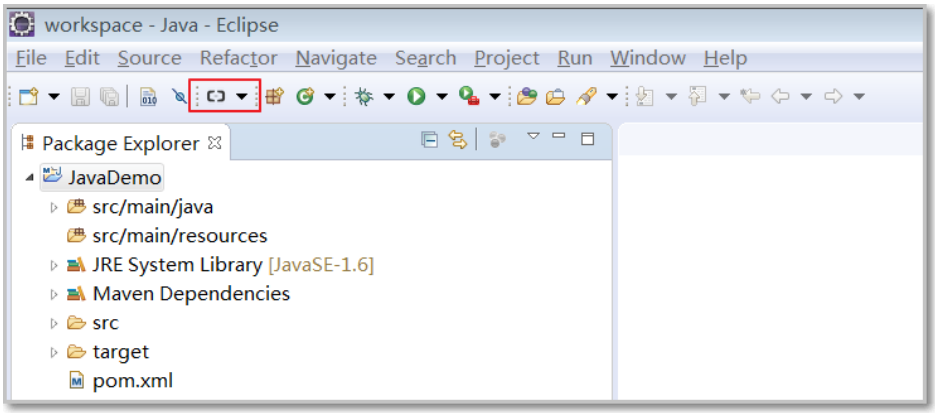
在下面的列表区域中勾选需要的组件 **Alibaba Cloud Toolkit Core** 和 **Alibaba Cloud Toolkit Deployment Tools**，并在下方 Details 区域中不勾选 **Connect all update sites during install to find required software**。完成组件选择之后，单击**Next**。



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。

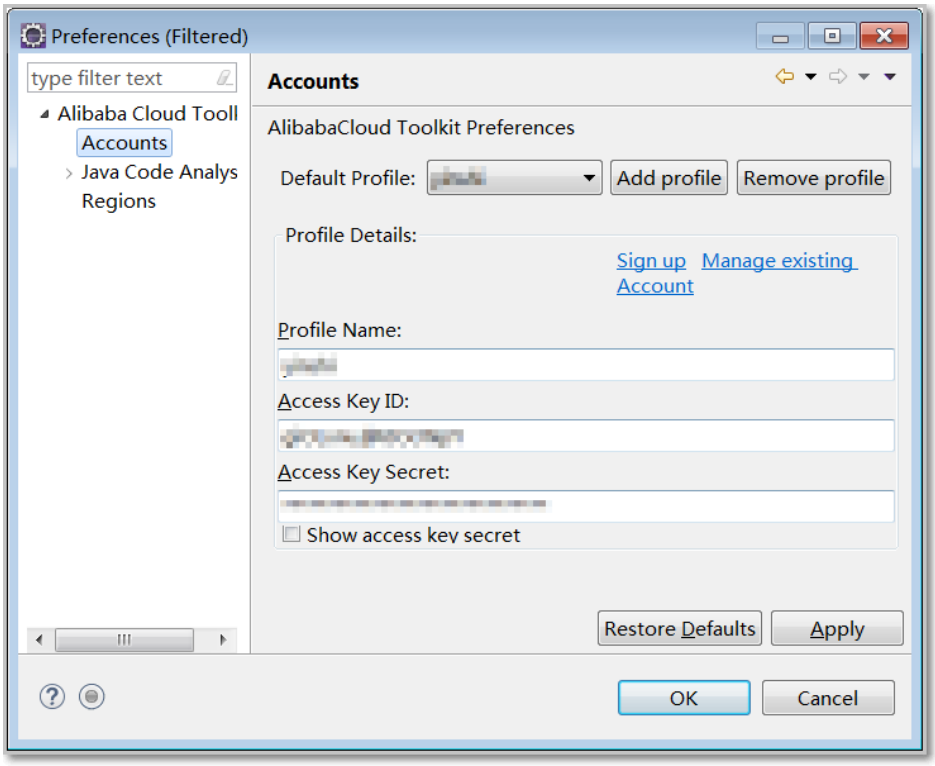
启动 Eclipse。

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。



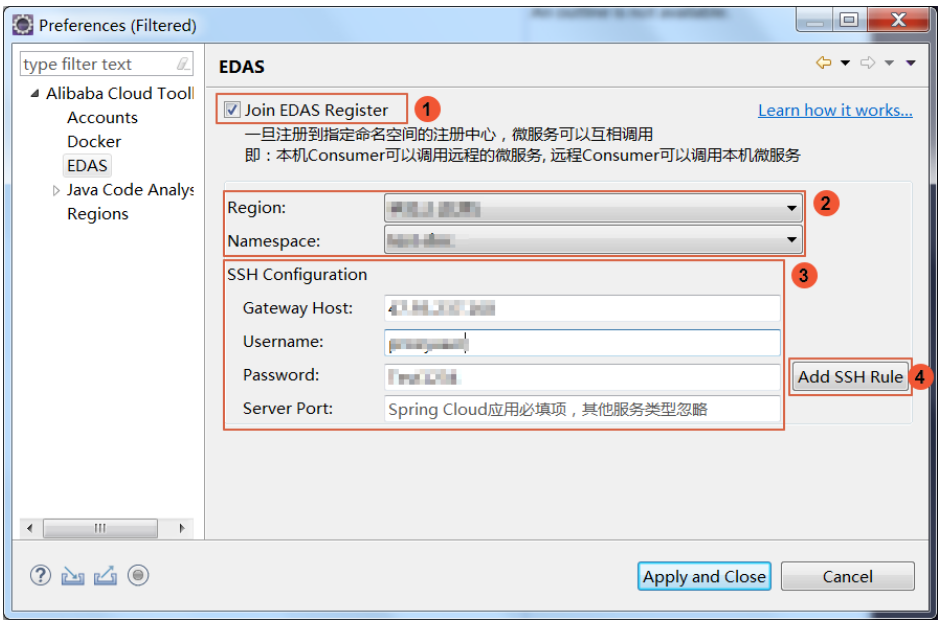
如果您已经注册过阿里云账号，在 **Accounts** 界面中单击 **Manage existing Acount**，进入阿里云登录页面。用已有账号登录后，跳转至安全信息管理页面，获取 **Access Key ID** 和 **Access Key Secret**。

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：端云互联配置

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:

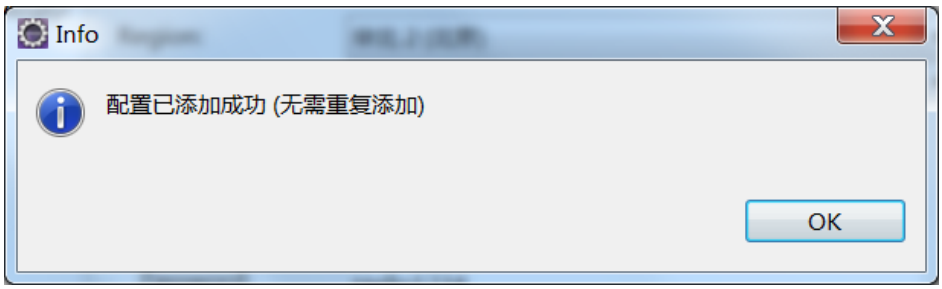
在 **Gateway Host** 输入框内输入您创建的 ECS 的**公网 IP**；

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。

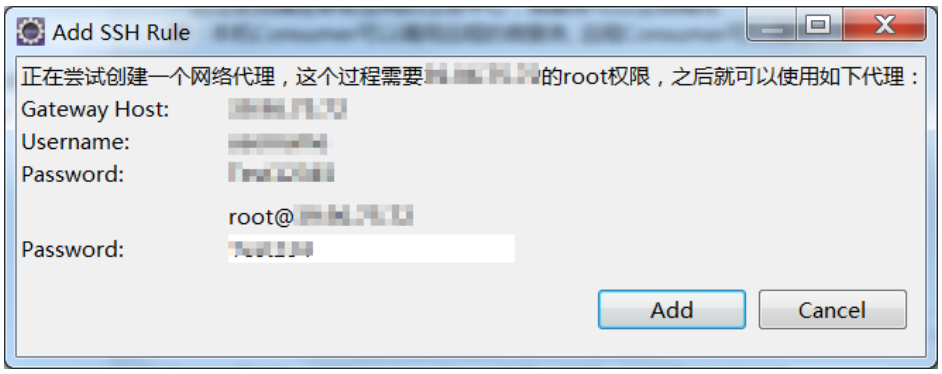
Server Port: Spring Boot 应用需添加该应用的服务端口，其他类型应用不需要填写。

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。

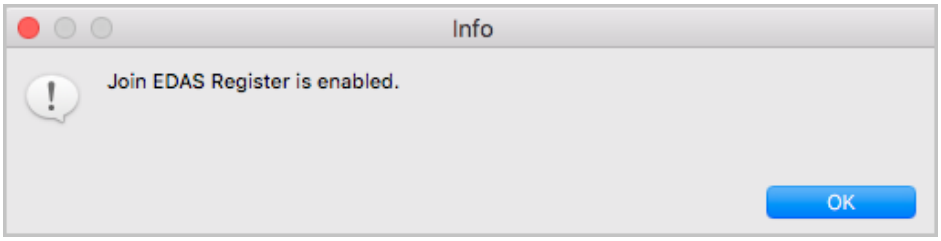
在 **Preference (Filtered)** 对话框的左侧导航栏中选择 **Appearance & BehaviorEndpoint**。

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

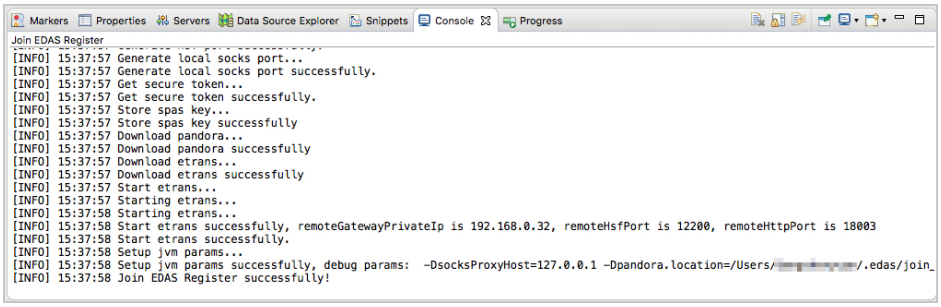
步骤四：启动本地应用进行端云互联

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。



常见问题

本文档模块介绍了使用插件进行端云互联时的常见问题的总结。

SOCKS proxy:connect timed out

如果您之前使用轻量级配置中心，那么现在启用端云互联插件后，需要把原来轻量级配置中心的相关配置删除，否则会出现连接超时的问题。

Spring Cloud 应用：请去掉以下配置项：

```
spring.cloud.alicloud.acm.server-list=127.0.0.1
spring.cloud.alicloud.acm.server-port=8080
spring.cloud.alicloud.ans.server-list=127.0.0.1
spring.cloud.alicloud.ans.server-port=8080
```

- **HSF 应用**：请去掉以下的 Hosts 绑定：

```
192.168.1.100  jmenv.tbsite.net
```

注册中心拒绝请求（code:403，msg: access denied）

注册中心会通过时间戳来保证服务注册的安全性，如果本地的系统时间不准确会导致云端注册中心拒绝请求

，请校准系统时间并进行重试。

更多信息

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。

问题反馈

如果您在使用 Cloud Toolkit 过程中有任何疑问，欢迎您扫描下面的二维码加入钉钉群进行反馈。