Enterprise Distributed Application Service (EDAS)

Application Development

MORE THAN JUST CLOUD | Alibaba Cloud

Application Development

Develop applications in Spring Cloud

Spring Cloud overview

EDAS supports the native Spring Cloud microservice framework. For the microservices developed under this framework, you only need to add dependencies and modify their configurations to obtain EDAS microservice hosting, microservice management, monitoring and alarm, application diagnosis, and other capabilities. This ensures zero code intrusion.

Spring Cloud provides a series of standards and specifications to simplify application development,covering service discovery, load balancing, fusion, configuration management, message event triggering, and message bus. In addition, Spring Cloud provides implementation components for service gateways, distributed tracing, security, distributed task scheduling, and distributed task coordination.

Currently, the most popular Spring Cloud implementation components in the industry include Spring Cloud Netflix, Spring Cloud Consul, Spring Cloud Gateway, and Spring Cloud Sleuth. Spring Cloud Alibaba, an open source middleware recently developed by Alibaba, is also a very popular implementation component in the industry.

For the applications developed by using Spring Cloud components such as Spring Cloud Netflix and Spring Cloud Consul, you can directly deploy them to EDAS and enable the application hosting capability. In addition, you can directly use the advanced monitoring functions provided by EDAS without modifying any code, enabling monitoring functions such as distributed tracing, monitoring and alarm, and application diagnosis.

To use more service management functions in EDAS to manage your Spring Cloud applications, replace your Spring Cloud components with those in Spring Cloud Alibaba or add the Spring Cloud Alibaba component.

Compatibility

Currently, EDAS supports Spring Cloud Greenwich, Spring Cloud Finchley and Spring Cloud Edgware. The details for version mapping relationship of Spring Cloud, Spring Boot and Spring Cloud Alibaba, see Version mapping notes.

The following table shows the compatibility between Spring Cloud functions, other implementation components, and EDAS.

Function		Component	Compatibility with EDAS	Description
Common functions	Service registration and discovery	- Netflix Eureka - Consul Discovery	Compatible, with an equivalent component	EDAS provides ANS for replacement.In addition to the service registration and discovery function of Spring Cloud, ANS also provides many other service management functions.
	Load balancing	Netflix Ribbon	Compatible	You can directly use Spring Cloud with the service registration and discovery component of EDAS.
	Service call	- Feign - RestTemplate	Compatible	You can directly use the service discovery and distributed tracing functions of EDAS.
Configuration management		- Config Server - Consul Config	Compatible, with an equivalent component	EDAS provides ACM for replacement.In addition to the service registration and discovery function of Spring Cloud, ACM also allows configuration management in the EDAS console and

			provides real-time refreshing and push track viewing.
Service gateway	- Spring Cloud Gateway - Netflix Zuul	Compatible	You can directly use the service discovery, configuration management, and distributed tracing functions of EDAS.
Distributed tracing	Spring Cloud Sleuth	Compatible, with an equivalent component	EDAS provides ARMS for replacement.To use ARMS, you only need to enable advanced monitoring in the EDAS console, without modifying any code or dependencies.In addition to the distributed tracing function, ARMS also provides functions such as complete troubleshooting and thread analysis.
Message-driven: Spring Cloud Stream	- RabbitMQ binder - Kafka binder	Compatible, with an equivalent component	EDAS provides RocketMQ Binder for replacement. It can be used with other components.
Message bus: Spring Cloud Bus	- RabbitMQ - Kafka	Compatible, with an equivalent component	EDAS provides RocketMQ Bus for replacement. It can be used with other components.
Security	Spring Cloud Security	Compatible	-

Distributed task scheduling	Spring Cloud Task	Compatible	-
Distributed coordination	Spring Cloud Cluster	Compatible	-

Version mapping notes

The mapping relation among Spring Cloud, Spring Boot, Sring Cloud Alibaba and EDAS commercial components is shown in the following table.

Spring Cloud	**Spring Boot**	**Spring Cloud Alibaba**	**EDAS commercial components**		
			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

Note: Spring Cloud Alibaba Nacos Discovery and Spring Cloud Alibaba Nacos Config are the corresponding open-source components of ANC and ACM.

Documentation

To replace the service discovery component (such as Spring Cloud Eureka or Spring Cloud Consul) with the spring-cloud-alicloud-ans component provided by EDAS, you only need to modify the dependencies and configurations, and not the code.For more information, see Service discovery.

To replace the configuration management component (such as Spring Cloud Config or Spring Cloud Consul) with the spring-cloud-alicloud-acm component provided by EDAS, you only need to modify the dependencies and configurations, and not the code. For more information, see Configuration management.

If you are using a service gateway but want to use the service registration and discovery, configuration management, and rate limiting and degradation functions provided by EDAS, you only need to introduce the starter dependencies and modify the configurations. For more information, see Service gateways.

Quick start

You can simply add basic dependencies and configurations to your Spring Cloud applications to deploy them to EDAS, and use the EDAS service registry for service discovery.For detailed steps, see Deploy Spring Cloud Applications to EDAS.

Implement load balancing

Spring Cloud uses the Ribbon component for load balancing. Ribbon mainly provides client-side software load balancing algorithms.In Spring Cloud, load balancing is achieved for the underlying RestTemplate and Feign clients through Ribbon.

Spring Cloud Alibaba ANS integrates the functions of Ribbon and AnsServerList implements the com.netflix.loadbalancer.ServerList API provided by Ribbon.

This API is generic and other similar service discovery components, such as Nacos, Eureka, Consul, and ZooKeeper, implement ServerList APIs such as NacosServerList, DomainExtractingServerList, ConsulServerList, and ZookeeperServerList.

Implementing the com.netflix.loadbalancer.ServerList API is equivalent to accessing the load balancing specifications of Spring Cloud. These specifications are generic.This means that no code modification is required to change the service discovery solution from Eureka, Consul, or ZooKeeper to Spring Cloud Alibaba, including RestTemplate, FeignClient, and the outdated AsyncRestTemplate.

The following describes how to implement load balancing of RestTemplate and Feign in your application.

You can download service-provider and service-consumer for complete demos.

RestTemplate

RestTemplate is a client provided by Spring Cloud to access RESTful services. It provides multiple ways to conveniently access remote HTTP services, greatly improving the writing efficiency of client-side code.

To use the load balancing feature of RestTemplate, you need to modify the code in your application based on the following example.

```
public class MyApp {
```

```
// Inject the RestTemplate you built with the @LoadBalanced annotation
// This annotation adds the LoadBalancerInterceptor to RestTemplate
// Internally, LoadBalancerInterceptor uses the implementation class RibbonLoadBalancerClient of the
LoadBalancerClient API for load balancing
@Autowired
private RestTemplate restTemplate;

@LoadBalanced //Modify the built RestTemplate with this annotation to enable its load balancing function.
@Bean
public RestTemplate restTemplate() {
return new RestTemplate();
}

// RestTemplate calls services in load balancing mode internally
public void doSomething() {
Foo foo = restTemplate.getForObject("http://service-provider/query", Foo.class);
doWithFoo(foo);
}

...
}
```

Feign

Feign is an HTTP client written in Java to simplify RESTful calls.

Use @EnableFeignClients and @FeignClient to initiate a load balancing request.

Enable the functions of Feign with the @EnableFeignClients annotation.

```
@SpringBootApplication
@EnableFeignClients // Enable the functions of Feign
public class MyApplication {
...
}
```

Build a FeignClient with the @FeignClient annotation.

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

Inject EchoService and call the echo method.

Calling the echo method is equivalent to initiating an HTTP request.

```
public class MyService {
@Autowired // Inject the EchoService you built with the @FeignClient annotation
private EchoService echoService;

public void doSomething() {
// This is equivalent to initiating an "http://service-provider/echo/test" request
echoService.echo("test");
}
...
}
```

Implement configuration management

This topic describes how to connect your Spring Cloud applications to ACM and use ACM to manage their configurations.

Why is ACM used?

Application Configuration Management (ACM) is a configuration management product of Alibaba Cloud, which is a commercial version of open source Nacos configuration management.

Compared with other similar products, ACM offers certain advantages.For more information, see ACM product comparison.

Local development

Spring Cloud Alicloud ACM implements the integration of ACM with Spring Cloud framework and supports the injection of native Spring configurations.

Preparation

Download, start, and configure the lightweight configuration center.

To facilitate local development, EDAS provides a lightweight configuration center that has the basic features of the EDAS service registry.You can deploy the applications developed based on the lightweight configuration center to EDAS without making any modifications to code or configurations.

For more information about how to download, start, and configure the lightweight configuration center, see Configure a lightweight configuration center.The latest version is recommended.

Log on to the console of the lightweight configuration center. In the left-side navigation pane, click **Configuration List**. On the **Configuration List** page, click **Add**. On the **Create Configuration** page, enter the following information.

- Group: DEFAULT_GROUP
- DataId: acm-example.properties
- Content: user.id=amctest

Use ACM for configuration management

Create a Maven project named acm-example.

The following takes Spring Boot 2.0.6 RELEASE and Spring Cloud Finchley.SR1 as an example. Add the following dependencies to the pom.xml file.

```
 <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-acm</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>
```

If you need to use Spring Boot 1.x, use Spring Boot 1.5.x, Spring Cloud Edgware, and Spring

Cloud Alibaba 0.1.1. RELEASE.

Note: Spring Boot 1.x will expire in August 2019, so we recommend that you use a later version to develop applications.

Develop the startup class AcmExampleApplication of acm-example.

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

Create a simple controller and specify a UserId from the key in the configuration file named user.id.

```
@RestController
public class EchoController {

@Value("${user.id}")
private String userId;

@RequestMapping(value = "/")
public String echo() {
return userId;
}
}
```

Add the following configuration to the bootstrap.properties file and specify the EDAS lightweight configuration center as the registry.

where, *127.0.0.1* is the address of the lightweight configuration center, which must be changed to the corresponding IP address if your lightweight configuration center is deployed on another instance.The lightweight configuration center does not support port modification, so port **8080** must be used.

```
spring.application.name=acm-example
server.port=18081
spring.cloud.alicloud.acm.server-list=127.0.0.1
spring.cloud.alicloud.acm.server-port=8080
```

Execute the main function in the startup class AcmExampleApplication to enable the service.

Result verification

In your browser, enter the address http://127.0.0.1:18081/. The value acmtest is returned, which indicates the value of user.id you configured in the lightweight configuration center.

Deploy applications to EDAS

ACM is designed for migrating applications from the development environment to EDAS. It allows you to directly deploy applications to EDAS without any code or configuration modifications.

Add the following configuration to the pom.xml file of acm-example. Then, run mvn clean package to package native programs into executable JAR packages.

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

Deploy applications based on the appropriate documents for the corresponding cluster types.

Reference configuration items

Configuration item	Key	Default value	Description
Extension	spring.cloud.alicloud.acm.file-extension	properties	Indicates the configuration file extension, typically being properties or yaml
Timeout	spring.cloud.alicloud.acm.timeout	3000	Indicates the timeout period for configuration retrieval
Refresh or not	spring.cloud.alicloud.acm.refresh-enabled	true	Indicates whether the Spring context is to be refreshed when the configuration changes
Endpoint	spring.cloud.alicloud.acm.endpoint	None	For more information, see

			ACM-SDK documentation.
Namespace	spring.cloud.alicloud.acm.namespace	None	For more information, see ACM-SDK documentation.
RAM role	spring.cloud.alicloud.acm.ram-role-name	None	For more information, see ACM-SDK documentation.

FAQ

What is the relationship between ACM and EDAS?

A: ACM is an independent Alibaba Cloud product and EDAS can provide a runtime environment for applications using ACM.

Deploy service gateways

This topic describes how to, based on Spring Cloud Gateway and Netflix Zuul, deploy service gateways for applications from scratch by using ANS.

[TOC]

Why do service gateways use ANS as the registry?

Application Naming Service (ANS) is a service discovery component provided by EDAS, which is a commercial version of open source Nacos.

org.springframework.cloud:spring-cloud-starter-alicloud-ans implements the standard APIs and specifications of Spring Cloud Registry. ANS can completely replace the service discovery function provided by Spring Cloud Eureka or Spring Cloud Consul.

In addition, ANS offers the following advantages over Spring Cloud Eureka and Spring Cloud Consul:

- ANS is a shared component that saves you the cost of deploying, operating, or maintaining Spring Cloud Eureka or Spring Cloud Consul.
- ANS provides link encryption for both service registration and discovery calls, protecting your service from being detected by others.

- ANS is fully integrated with other EDAS components to provide you with a complete set of microservice solutions.

Preparation

Download, start, and configure the lightweight configuration center

To facilitate local development, EDAS provides a lightweight configuration center that has the basic features of the EDAS service registry.You can deploy the applications developed based on the lightweight configuration center to EDAS without making any modifications to code or configurations.

For more information about how to download, start, and configure the lightweight configuration center, see Configure the lightweight configuration center.The latest version is recommended.

Download Maven and set environment variables (skip this step if Maven is installed locally).

Deploy service gateways based on Spring Cloud Gateway

The following describes how to use ANS to deploy service gateways from scratch based on Spring Cloud Gateway.

Create a service gateway

Create a Maven project named spring-cloud-example-ans-gateway.

Add Spring Boot and Spring Cloud Finchley dependencies to the pom.xml file.

The following takes Spring Boot 2.0.6 RELEASE and Spring Cloud Finchley.SR1 as an example.

Note: Spring Cloud Gateway is a component developed based on Spring Boot 2.0. If you are using Spring Cloud Gateway as the service gateway, select Spring Boot 2.0 or later.If you are using Spring Boot 1.x, we recommend that you upgrade it to Spring Cloud 2.0.

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

```
<properties>
   <spring-cloud.version>Finchley.SR1</spring-cloud.version>
   <spring-cloud-alibaba-cloud.version>0.2.1.RELEASE</spring-cloud-alibaba-cloud.version>
</properties>

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

<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-alicloud-ans</artifactId>
      <exclusions>
         <! - Spring Cloud Gateway uses Netty as its HTTP server, so you need to **exclude** dependencies
on spring-boot-starter-web to prevent startup failures. -->
         <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-alicloud-context</artifactId>
   </dependency>
</dependencies>
```

Develop the service gateway startup class AnsGatewayApplication.

```
@SpringBootApplication
```

```
@EnableDiscoveryClient
<! - Service registration and discovery functions must be enabled for the application -->
public class AnsGatewayApplication {
    public static void main(String[] args) {

        SpringApplication.run(AnsGatewayApplication.class, args);
    }
}
```

Add the following configuration to the application.yaml file and specify the EDAS lightweight configuration center as the registry.

where, *127.0.0.1* is the address of the lightweight configuration center, which must be changed to the corresponding IP address if your lightweight configuration center is deployed on another instance.The lightweight configuration center does not support port modification, so port **8080** must be used.

```
server:
  port: 15012
spring:
  application:
    name: spring-gateway-example
  cloud:
    gateway: # config the routes for gateway
      routes:
      - id: lb_service-provider
        uri: lb://service-provider
        predicates:
        - Path=/**
    alicloud:
      ans:
        server-list: 127.0.0.1
        server-port: 8080
```

Execute the main function in the startup class AnsGatewayApplication to enable the service.

Log on to the console of the lightweight configuration center (http://127.0.0.1:8080). In the left-side navigation pane, click Services to view the list of service providers.spring-gateway-example exists in the list of service providers.

Create a service provider

Create an application that serves as the service provider.For more information, see Quick start.

Sample service provider:

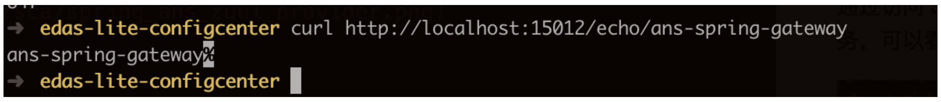
```
@SpringBootApplication
@EnableDiscoveryClient
```

```
public class AnsProviderApplication {

    public static void main(String[] args) {

        SpringApplication.run(AnsProviderApplication.class, args);
    }

    @RestController
    public class EchoController {
        @RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
        public String echo(@PathVariable String string) {
            return string;
        }
    }
}
```

Result verification

Locally verify the result.

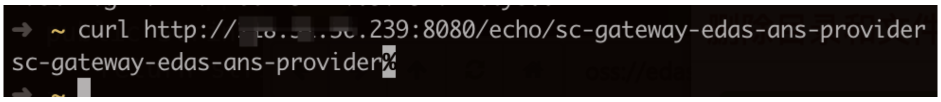
Locally start the developed service gateway and service provider and access Spring Cloud Gateway to forward the request to the backend service. The result indicating a successful call is returned.



```
➜ edas-lite-configcenter curl http://localhost:15012/echo/ans-spring-gateway
ans-spring-gateway%
➜ edas-lite-configcenter
```

Verify the result in EDAS.

Deploy the developed service gateway and service provider to EDAS and access Spring Cloud Gateway to forward the request to the backend service. The result indicating a successful call is returned.



```
➜ ~ curl http://[redacted].239:8080/echo/sc-gateway-edas-ans-provider
sc-gateway-edas-ans-provider%
➜ ~
```

Use Spring Boot 2.x to deploy service gateways based on Zuul

The following describes how to use ANS to deploy service gateways from scratch in Spring Boot 2.x based on Netflix Zuul.

Create a service gateway

Create a Maven project named spring-cloud-example-ans-zuul.

Add Spring Boot and Spring Cloud Finchley dependencies to the pom.xml file.

The following takes Spring Boot 2.0.6 RELEASE and Spring Cloud Finchley.SR1 as an example.

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

  <properties>
   <spring-cloud.version>Finchley.SR1</spring-cloud.version>
   <spring-cloud-alibaba-cloud.version>0.2.1.RELEASE</spring-cloud-alibaba-cloud.version>
</properties>

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

<dependencies>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-alicloud-ans</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-alicloud-context</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
```

```
    </dependency>
  </dependencies>
```

If you need to use Spring Boot 1.x, use Spring Boot 1.5.x, Spring Cloud Edgware, and **org.springframework.cloud:spring-cloud-starter-alicloud-ans** version 0.1.1. RELEASE.

Note: Spring Boot 1.x will expire in August 2019, so we recommend that you use a later version to develop applications.

Develop the service gateway startup class AnsZuulTwoXApplication.

```
@SpringBootApplication
@EnableDiscoveryClient
<! - Enable the service registration and discovery functions -->
@EnableZuulProxy
  <! - Enable the Zuul Server agent -->
public class AnsZuulTwoXApplication {

    public static void main(String[] args) {

        SpringApplication.run(AnsZuulTwoXApplication.class, args);
    }
}
```

Add the following configuration to the application.properties file and specify the EDAS lightweight configuration center as the registry.

where, *127.0.0.1* is the address of the lightweight configuration center, which must be changed to the corresponding IP address if your lightweight configuration center is deployed on another instance.The lightweight configuration center does not support port modification, so port **8080** must be used.

```
spring.application.name=spring-cloud-ans-gateway
server.port=13012
spring.cloud.alicloud.ans.server-list=127.0.0.1
spring.cloud.alicloud.ans.server-port=8080
# config zuul
zuul.routes.service-provider.path=/**
```

Execute the main function AnsZuulTwoXApplication in spring-cloud-example-ans-zuul to enable the service.

Log on to the console of the lightweight configuration center *http://127.0.0.1:8080*. In the left-side navigation pane, click **Services** to view the list of service providers.spring-cloud-ans-gateway exists in the list of service providers.

Create a service provider

For more information about how to quickly create a service provider, see Quick start.

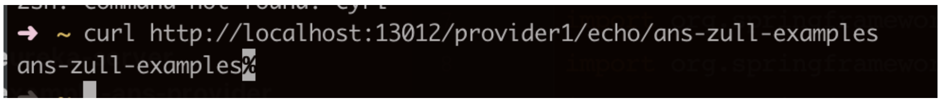
Sample service provider startup class:

```
@SpringBootApplication
@EnableDiscoveryClient
public class AnsProviderApplication {

    public static void main(String[] args) {

        SpringApplication.run(AnsProviderApplication.class, args);
    }

    @RestController
    public class EchoController {
        @RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
        public String echo(@PathVariable String string) {
            return string;
        }
    }
}
```

Result verification

Locally verify the result.

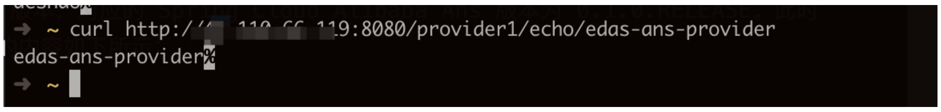
Access the API provided by the backend service through Netflix Zuul. The result indicating a successful call is returned.



```
➜ ~ curl http://localhost:13012/provider1/echo/ans-zull-examples
ans-zull-examples%
```

Verify the result in EDAS.

Deploy the developed service gateway and service provider to EDAS and access the backend service through the service gateway. The result indicating a successful call is returned.



```
➜ ~ curl http://[redacted]119:8080/provider1/echo/edas-ans-provider
edas-ans-provider%
➜ ~
```

Use Spring Boot 1.x to deploy service gateways based on

Netflix Zuul

The following describes how to use ANS to deploy service gateways from scratch in Spring Boot 1.x based on Netflix Zuul.

If you need to use Spring Boot 1.x, use Spring Boot 1.5.x, Spring Cloud Edgware, and **org.springframework.cloud:spring-cloud-starter-alicloud-ans** version 0.1.1. RELEASE.The content in the pom.xml file is as follows:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.13.RELEASE</version>
  <relativePath/>
</parent>

<properties>
  <spring-cloud.version>Edgware.SR4</spring-cloud.version>
  <spring-cloud-alibaba-cloud.version>0.1.1.RELEASE</spring-cloud-alibaba-cloud.version>
</properties>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>${spring-cloud-alibaba-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alicloud-ans</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-alicloud-context</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
  </dependency>
  <dependency>
```

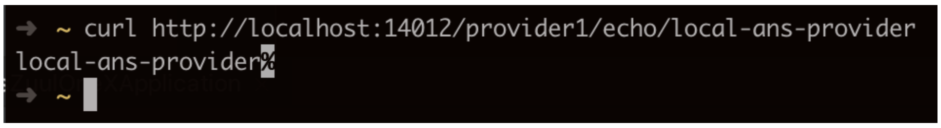
```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Note: In contrast to using Spring Boot 2.x to integrate ANS based on Zuul, this method uses different versions of Spring Boot, Spring Cloud, and **org.springframework.cloud:spring-cloud-starter-alicloud-ans**, but follows the same main method and the same application.properties file.

Result verification

Locally verify the result.

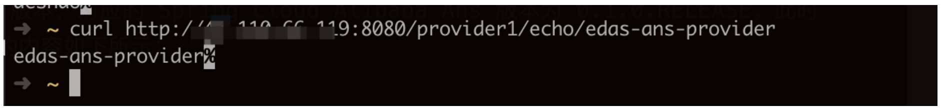
Access the API provided by the backend service through Netflix Zuul. The result indicating a successful call is returned.



```
➜ ~ curl http://localhost:14012/provider1/echo/local-ans-provider
local-ans-provider%
➜ ~
```

Verify the result in EDAS.

Deploy the developed service gateway and service provider to EDAS and access the backend service through the service gateway. The result indicating a successful call is returned.



```
➜ ~ curl http://[redacted]19:8080/provider1/echo/edas-ans-provider
edas-ans-provider%
➜ ~
```

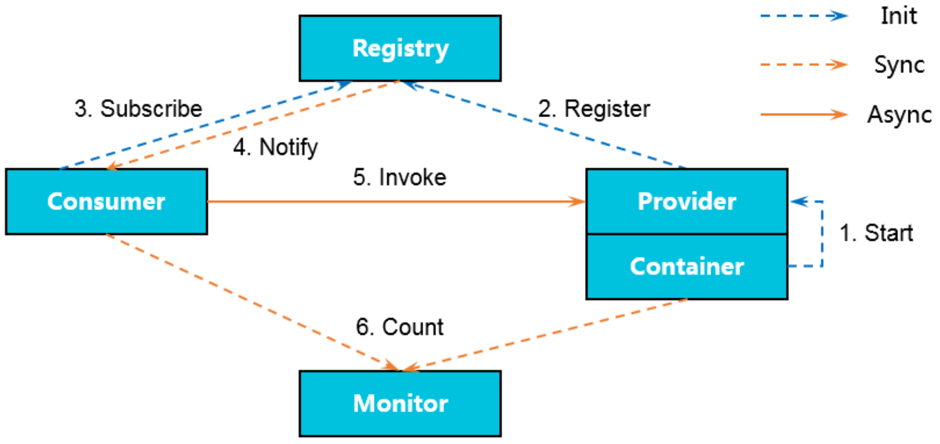
Develop applications in Dubbo

Dubbo overview

EDAS supports native Dubbo microservice framework. You need only add dependencies and modify few configurations for the microservices developed in the Dubbo framework and deploy the microservices onto EDAS, then the microservices can be managed, governed, monitored and diagnosed on EDAS console.

Dubbo architecture

The following figure shows the Dubbo architecture.



1. The service running container starts, load, and runs provider services.
2. During startup, the provider registers with the registry.
3. During startup, the consumer registers with the registry.
4. The registry returns a list of provider addresses to the consumer.When changes occur, the registry pushes changed data to the consumer based on persistent connection.
5. The consumer selects a provider from the list of provider addresses based on the software load balancing algorithm.If the call fails, the consumer can call another provider.
6. The consumer and provider store the accumulated call count and call time in the memory, and send statistical data to the monitoring center periodically (every minute).

Use Spring Boot to develop Dubbo applications

Spring Boot simplifies the configuration and deployment of microservice applications. Meanwhile, Nacos also provides the service registration and discovery as well as configuration management functions. Together, both functions can help you improve development efficiency. This topic describes how to use Spring Boot annotations to develop a Dubbo microservice sample application based on Nacos.

Preparations

Before using Spring Boot to develop a Dubbo microservice application, complete the following tasks:

Download Maven and set environment variables.

Download the latest version of Nacos Server.

To start Nacos Server, follow these steps.

i. Decompress the downloaded Nacos Server package.
ii. Go to the nacos/bin directory and start Nacos Server.
  - For Linux/UNIX/Mac: Run the sh startup.sh -m standalone command.
  - For Windows: Double-click the startup.cmd file to run it.

Sample project

You can follow the steps described in this topic to build the project. Alternatively, you can directly download the sample project used in this topic, or use Git to clone the project by running this command: git clone https://github.com/aliyun/alibabacloud-microservice-demo.git.

This project contains many sample projects. The sample project used in this topic can be found in alibabacloud-microservice-demo/microservice-doc-demo/dubbo-samples-spring-boot.

Create a service provider

Create a Maven project named spring-boot-dubbo-provider.

Add required dependencies to the pom.xml file.

The following uses Spring Boot 2.0.6.RELEASE as an example.

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.0.6.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
```

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

Develop a Dubbo service provider.

All services in Dubbo are provided as interfaces.

In src/main/java, create a package named com.alibaba.edas.boot.

In com.alibaba.edas.boot, create an interface named IHelloService that contains a SayHello method.

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

Create a class named IHelloServiceImpl in com.alibaba.edas.boot to implement the interface.

```
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)";
}
}
```

Note: In Dubbo, the service annotation is **com.alibaba.dubbo.config.annotation.Service**.

Configure the Dubbo service.

In src/main/resources, create a file named application.properties or application.yaml and open it.

In application.properties or application.yaml, add the following configuration items.

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

Note:

You must specify values for the preceding three configuration items because they have no defaults.

i. The value of dubbo.scan.basePackages is the name of the package with code containing the annotations com.alibaba.dubbo.config.annotation.Service and com.alibaba.dubbo.config.annotation.Reference. Separate multiple packages with commas (,).
ii. The prefix of the value of dubbo.registry.address must start with **nacos://** . The IP address and port that follow refer to the address and port of Nacos Server, respectively. The IP address in the code example is a local address. If you deploy Nacos Server on another machine, change it to the actual IP address.

Develop and start the Spring Boot handler class 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);
}

}
```

Log on to the Nacos console at http://127.0.0.1:8848. In the left-side navigation pane, click **Services** to view the list of providers.

You can see com.alibaba.edas.boot.IHelloService in the list of service providers and you can query the group and provider IP address of the service.

Create a service consumer

Create a Maven project named spring-boot-dubbo-consumer.

Add required dependencies to the pom.xml file.

The following uses Spring Boot 2.0.6.RELEASE as an example.

```
 <dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.0.6.RELEASE</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-actuator</artifactId>
</dependency>
<dependency>
<groupId>org.apache.dubbo</groupId>
<artifactId>dubbo-spring-boot-starter</artifactId>
<version>2.7.3</version>
</dependency>
<dependency>
<groupId>com.alibaba.nacos</groupId>
<artifactId>nacos-client</artifactId>
<version>1.1.1</version>
</dependency>

</dependencies>
```

If you want to use Spring Boot 1.x, select Spring Boot 1.5.x. The corresponding com.alibaba.boot:dubbo-spring-boot-starter version is 0.1.0.

Note: Spring Boot 1.x will reach end-of-life in August 2019. We recommend that you use a later version to develop applications.

Develop a Dubbo consumer

In src/main/java, create a package named com.alibaba.edas.boot.

In com.alibaba.edas.boot, create an interface named IHelloService that contains a SayHello method.

```
package com.alibaba.edas.boot;

public interface IHelloService {
String sayHello(String str);
}
```

Develop a Dubbo service call.

For example, you need to call a remote Dubbo service once in Controller. The code is as follows.

```
 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);
}
}
```

Note: The Reference annotation is com.alibaba.dubbo.config.annotation.Reference.

Add the following configuration items in the application.properties/application.yaml configuration file:

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

Note:

- The preceding two configuration items have no defaults and must be specified.
- The prefix of the value of dubbo.registry.address must start with **nacos://**. The IP address and port that follow refer to the address and port of Nacos Server, respectively. The IP address in the code example is a local address. If you deploy Nacos Server on another machine, change it to the actual IP address.

Develop and start the Spring Boot handler class 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);
}
}
```

Log on to the Nacos console at http://127.0.0.1:8848. Then, in the left-side navigation pane, choose **Services**. On the **Services** page that appears, select **Callers** to view the list of callers.

You can see com.alibaba.edas.boot.IHelloService in the list. Also, you can view the group and caller IP address of the service.

Verify the result

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

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

Deploy the application to EDAS

You can directly deploy the application that uses local Nacos as the registry to EDAS without making any changes. This registry will be automatically replaced with the registry in EDAS.

Based on your actual needs, you can choose the type of cluster to deploy the application to, which is mainly ECS cluster or Container Service Kubernetes cluster, as well as the deployment method, which can be in the console or with tools. For more information, see Application deployment overview.

If you use the console for deployment, complete the following steps in your local application before deploying it:

Add the following packaging plug-in configuration items to the pom.xml file.

Provider

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

Consumer

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

Execute mvn clean package to package your local program into a JAR file.

More information

In addition to Spring Boot, you can also develop Dubbo microservice applications by using XML. For more information, see Host Dubbo applications in EDAS.

If you are using edas-dubbo-extension, see Host Dubbo applications in EDAS with EDAS-Dubbo-extension. With edas-dubbo-extension, you are unable to use related capabilities provided by EDAS, such as Dubbo service governance. Therefore, we recommend that you migrate to Nacos instead.

Develop applications in HSF

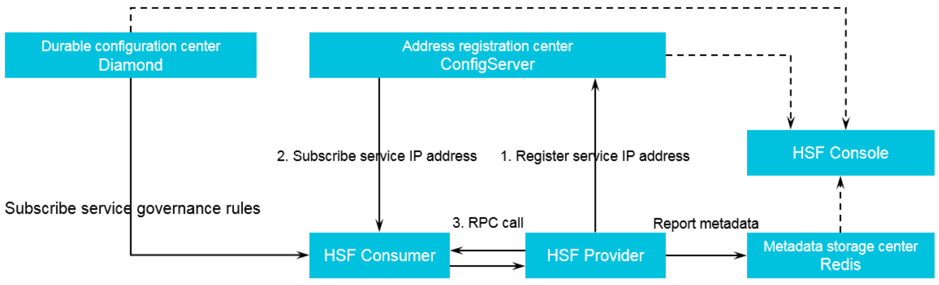
HSF overview

High-speed Service Framework (HSF) is a distributed RPC service framework widely used within the Alibaba Group.

HSF connects different business systems, decoupling the implementation of the systems from each other.HSF unifies service publishing/call methods from the perspective of distributed applications, helping you conveniently and quickly develop distributed applications. It provides public function modules, which avoid complex technical details in distributed systems, such as remote communication, serialization implementation, performance loss, and synchronous/asynchronous call method implementation.

HSF architecture

As a client-side RPC framework, HSF itself has no server cluster. All HSF service calls are point-to-point between the consumer and the provider.However, HSF must work with the following external systems to implement the complete distributed service system.


Durable configuration center
Diamond
Address registration center
ConfigServer
HSF Console
2. Subscribe service IP address
1. Register service IP address
Subscribe service governance rules
3. RPC call
Report metadata
HSF Consumer
HSF Provider
Metadata storage center
Redis

Address registration center

HSF relies on the address registration center for service discovery. Without the address registration center, HSF can only make simple point-to-point calls.The service provider cannot advertise its service information to others. The service consumer may know which services to call, but cannot obtain information about the instances providing these services.In this case, the address registration center serves as a medium for the discovery of service information.The role of address registration center is played by ConfigServer.

Persistent configuration center

The persistent configuration center is used to store various governance rules of HSF services. At startup, the HSF client subscribes necessary service governance rules, such as routing rules, grouping rules, and weight rules, from the persistent configuration center to intervene in the address selection logic of the calling procedure based on the rules.The role of persistence configuration center is played by Diamond.

Metadata storage center

Metadata refers to the list of methods, parameter structure, and other information related to HSF services. Metadata does not affect the calling procedure of HSF. Therefore, the metadata storage center is not required.However, to ensure convenient service maintenance, at startup the HSF client submits the metadata to the metadata storage center for further maintenance.The role of metadata storage center is played by Redis.

Functions

As a distributed RPC framework, HSF supports multiple service calling methods:

Synchronous calls

By default, the HSF client consumes services by synchronous calls, and the client’ s codes must synchronously wait for the returned results of calls.

Asynchronous calls

For the client that calls HSF services, it is not always necessary to synchronously wait for the returned results of calls.For such services, HSF supports asynchronous calls, releasing clients from being congested synchronously in HSF calling operations.There are two kinds of asynchronous HSF calls:

*Future call: The client obtains the returned results of calls by HSFResponseFuture.getResponse(int timeout) when needed.

- Callback call: The callback call utilizes the callback mechanism provided by HSF. When specified HSF services are consumed and the results are returned, the HSF framework calls the HSFResponseCallback API used by the consumer to obtain the call results. The client obtains the results by using the callback notification.

Generic calls

For a typical HSF call, the HSF client has to perform a programming call with the API of the second party library of the service to obtain the returned results.In contrast, generic call means to initiate HSF call and obtain returned results, independent of the second party library of the service.For some platform-based products, generic calls can effectively reduce their dependence on second party libraries and support lightweight system running.

HTTP calls

HSF can advertise service information over HTTP, so that non-Java clients can call HSF services over HTTP.

Trace Filter extension

HSF, designed with a built-in call filter, can actively find and integrate the user’ s call filter extension point into HSF trace, enhancing the convenience of HSF request extension.

Application development methods

Under HSF, you can use Ali-Tomcat and Pandora Boot to develop applications.

Ali-Tomcat: Relying on Ali-Tomcat and Pandora, this method provides complete HSF functions, including service registration and discovery, implicit parameter passing, asynchronous call, generic call, trace Filter extension, rate limiting and degradation, and distributed tracing.In this method, applications must be deployed with WAR packages.

Pandora Boot: Relying on Pandora, this method provides complete HSF functions, including service registration and discovery, implicit parameter passing, asynchronous call, generic call, trace Filter extension, rate limiting and degradation, and distributed tracing.Applications can be packaged and deployed as JAR packages that run independently.

Configure the lightweight configuration

center

The lightweight configuration center allows developers to discover, register, and query services during development, debugging, and testing.This module does not belong to the official environment of EDAS. You must download the installation package and complete installation before using it.

Within a company, you generally only need to install the lightweight configuration center on one ECS instance and bind specific hosts on other development computers.The following specifies the steps for installation and use.

1. Download the lightweight configuration center

Check that the environment requirements are met.

Check that the environment variable JAVA_HOME points to JDK 1.6 or later.

Check that port 8080 and port 9600 are unused.

Because port 8080 and port 9600 are used for starting the EDAS configuration center, we recommend that you use a **dedicated** ECS instance (for example, a test ECS instance) to start the EDAS configuration center.If the whole test is conducted on the same ECS instance, change the port of the Web project to another unused port.

Download the EDAS configuration center installation package and decompress it.

If required, you can download a previous version:

- January 2018 version
- August 2017 version
- July 2017 version
- March 2017 version
- December 2016 version

2. Start the lightweight configuration center

Go to the decompressed edas-config-center directory to start the configuration center.

- Windows operating system: Double-click startup.bat.

- UNIX operating system: Run the sh startup.sh command in the current directory.

3. Configure hosts

To use the lightweight configuration center on development ECS instances, point the jmenv.tbsite.net domain on the local DNS (hosts file) to the IP address of the ECS instance that starts the EDAS configuration center.

The path of the hosts file is as follows:

Windows operating system: C:\Windows\System32\drivers\etc\hosts

UNIX operating system: /etc/hosts

Example

If you start the EDAS configuration center on the ECS instance whose IP address is 192.168.1.100, add the following line to the hosts files on all development ECS instances:

192.168.1.100 jmenv.tbsite.net

Result verification

After binding a host to the lightweight configuration center, open the browser, enter jmenv.tbsite.net:8080 in the address bar, and press Enter.

Then, the homepage of the lightweight configuration center is displayed.

- If the homepage is displayed normally, you have configured the lightweight configuration center successfully.
- If the homepage is not displayed, troubleshoot the problem by going backwards step by step.

If you encounter a problem when configuring the lightweight configuration center, see Lightweight configuration center problems for troubleshooting.

Ali-Tomcat Developer Guide

Ali-Tomcat overview

Ali-Tomcat is a container that EDAS relies on to run services. It integrates service publishing, subscription, service call tracing, and other core functions.You can deploy applications in this container in both development and runtime environments.

Pandora is a lightweight isolation container, namely taobao-hsf.sar.This container is used to isolate dependencies between applications and middleware products and between middleware products.EDAS Pandora integrates plugins that implement service discovery, configuration pushing, service call tracing, and other middleware products.By using these plugins, you can monitor, process, trace, analyze, maintain, and manage EDAS applications.

If you have not used HSF, do not use Ali-Tomcat to develop EDAS applications.

Note: On EDAS, Ali-Tomcat is only available for web applications in WAR format.

Prepare development tools

Install Ali-Tomcat and Pandora

Ali-Tomcat and Pandora are containers that EDAS relies on to run services. They integrate service publishing, subscription, service call tracing, and other core functions. Applications must be published on such containers in both development and runtime environments.

Note: Use JDK 1.7 or later.

Download Ali-Tomcat and decompress the downloaded archive to the appropriate directory (for example, d:\work\tomcat).

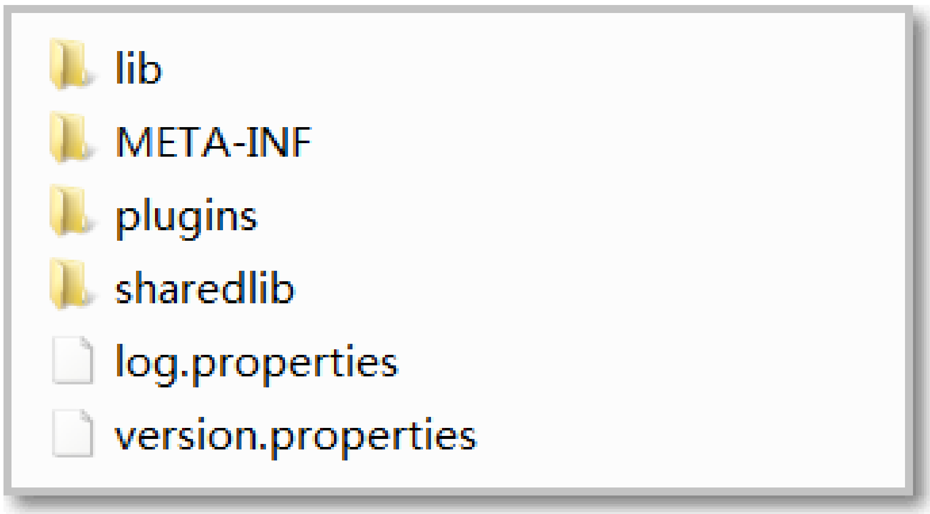
Download Pandora Container and decompress the downloaded archive to the Deploy directory where Ali-Tomcat is saved (which is d:\work\tomcat\deploy\ in this example).

View the Pandora Container directory structure.

In Linux systems, run the **tree** -L 2 deploy/ command in the relevant path to view the directory structure.

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

In a Windows system, directly navigate to the appropriate path to view the directory structure.



lib
META-INF
plugins
sharedlib
log.properties
version.properties

If you encounter a problem when installing or using Ali-Tomcat or Pandora, see Ali-Tomcat or Pandora to troubleshoot and resolve the problem.

Configure the development environment

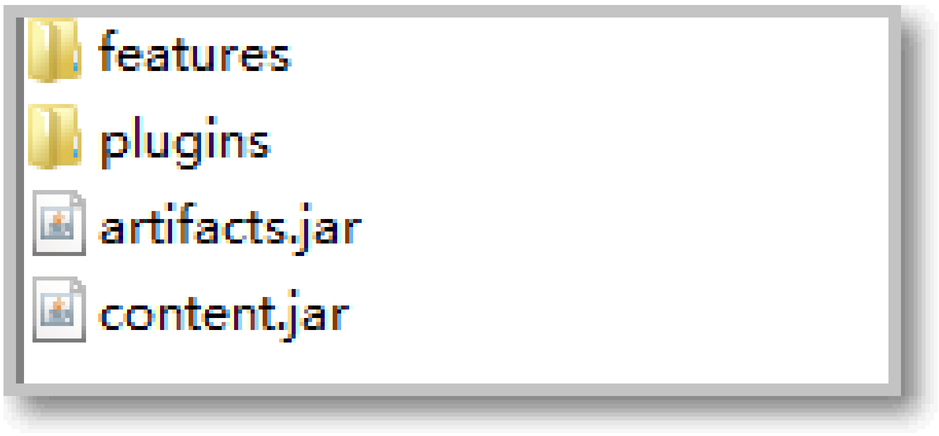
To develop an application locally, use either Eclipse or IntelliJ IDEA.This topic describes how to configure Eclipse and IntelliJ IDEA development environments.

Configure the Eclipse environment

To configure Eclipse, you must download the Tomcat4E plug-in and save it to the directory where Pandora Container is stored, which was created when you installed Ali-Tomcat. After configuring Eclipse, you can directly publish and debug local code in this environment.The operations are as follows:

Download the Tomcat4E plug-in and decompress it to a local directory (such as d:\work\tomcat4e).

The package contains the following items:



features
plugins
artifacts.jar
content.jar

Open Eclipse and choose **Help** > **Install New Software** from the menu bar.

In the Install dialog box, click **Add** to the right of the "Work with" area. In the Add Repository dialog box that appears, click **Local**.In the dialog box that appears, select the directory where the decompressed Tomcat4E plug-in resides (d:\work\tomcat4e\ in this example) and click **OK**.

Return to the Install dialog box, and click **Select All** and then **Next**.

Follow the prompts on the interface to perform the subsequent steps.After installing the Tomcat4E plug-in, you must restart Eclipse to make the plug-in take effect.

Restart Eclipse.

After the restart, choose **Run As** > **Run Configurations** from the Eclipse menu.

Select **AliTomcat Webapp** in the left-side navigation pane, and click the **New launch configuration** icon at the top.

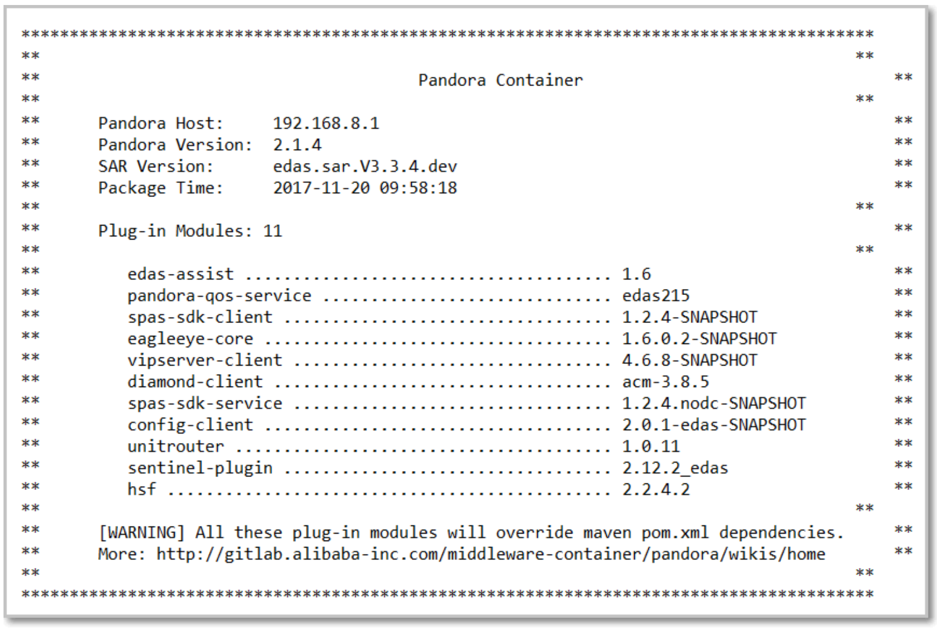
On the page that appears, click the **AliTomcat** tab. Then, in the **taobao-hsf.sar Location** area, click **Browse** and select the local path for Pandora, such as d:\work\tomcat\deploy\taobao-hsf.sar.

Click **Apply** or **Run** to complete the configuration.

Next time, you can start this project directly, without the need to configure it again.

View the output information concerning project operation. If the following Pandora

Container information is displayed, you have configured the Eclipse development environment successfully.



```
*************************************************************************************
**                                                                                 **
**                                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       **
**                                                                                 **
*************************************************************************************
```

Configure the IntelliJ IDEA environment

Note: At present, the IntelliJ IDEA commercial edition, but not the community edition, is supported.For this reason, ensure that the IntelliJ IDEA commercial edition is installed locally.

Run IntelliJ IDEA.

In the menu bar, choose **Run** > **Edit Configuration**.

On the **Run/Debug Configuration** page, choose **Defaults** > **Tomcat Server** > **Local** in the left-side navigation pane.

Configure AliTomcat.

On the right of the page, click the **Server** tab. In the **Application Server** area, click **Configure**.

On the **Application Server** page, click **+** in the upper-right corner. In the **Tomcat Server** dialog box, set **Tomcat Home** and **Tomcat base directory**, then click **OK**.

Set **Tomcat Home** to the local directory where Ali-Tomcat was decompressed. Then, **Tomcat base directory** is automatically set to the same path.

From the drop-down list in the **Application Server** area, select the configured Ali-Tomcat instance.

In the **VM Options** area, set the JVM startup parameter to the Pandora path, such as -Dpandora.location=d:\work\tomcat\deploy\taobao-hsf.sar.

Note: Replace *d:\work\tomcat\deploy\taobao-hsf.sar* with the actual local path where Pandora was installed.

Click **Apply** or **OK** to complete the configuration.

Develop applications with EDAS SDK

Quick start

This topic describes how to use EDAS SDK to quickly develop applications in HSF.

Download demo projects

Code described in this topic can be obtained from the official demo.

Download Demo projects.

Decompress the downloaded package and locate the carshop folder, where you can see the following Maven project subfolders: itemcenter-api, itemcenter, and detail.

- itemcenter-api: provides the API definition.
- itemcenter: specifies the service provider application.
- detail: specifies the service consumer application.

Note: Use JDK 1.7 or later.

Define service APIs

HSF services are implemented based on APIs. After an API is defined, the provider can implement a specific service over this API. The consumer also subscribes to the service over this API.

In the itemcenter-api project in demo, the service API com.alibaba.edas.carshop.itemcenter.ItemService is defined and has the following content:

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

The service API provides two methods: **getItemById** and **getItemByName**.

Develop provider services

The provider implements service APIs to provide specific services.Because the Spring framework is used for development, you also need to configure service attributes in the .xml file.

Note: The itemcenter folder in the demo project contains the sample code of a provider service.

Implement a service API

Implement a service API by referring to the example in the ItemServiceImpl.java file:

```
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;
}
}
```

Configure service attributes

The preceding example implements the service API com.alibaba.edas.carshop.itemcenter.ItemService and returns an Item object to both methods.After developing code, configure general items and add Maven dependencies in the web.xml file. Then, use the <hsf /> tag in the Spring configuration file to register and publish the service.This procedure is as follows:

Add the following Maven dependency to the pom.xml file:

```
<dependencies>
<! -- Add Servlet dependency -->
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>servlet-api</artifactId>
<version>2.5</version>
<scope>provided</scope>
</dependency>
<! -- Add Spring dependency -->
<dependency>
<groupId>com.alibaba.edas.carshop</groupId>
<artifactId>itemcenter-api</artifactId>
<version>1.0.0-SNAPSHOT</version>
</dependency>
<! -- Add service API dependency -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>2.5.6 (or later)</version>
</dependency>
<! -- Add edas-sdk dependency -->
<dependency>
<groupId>com.alibaba.edas</groupId>
<artifactId>edas-sdk</artifactId>
<version>1.5.0</version>
</dependency>
</dependencies>
```

Add the HSF-specific Spring configuration items to the hsf-provider-beans.xml file.

```
<?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">
<! -- Define the implementation of the service -->
<bean id="itemService" class="com.alibaba.edas.carshop.itemcenter.ItemServiceImpl" />
<! -- Use the hsf:provider tag to provide a service provider. -->
<hsf:provider id="itemServiceProvider"
<! -- Use the interface attribute to indicate that the service is an implementation of the class. -->
interface="com.alibaba.edas.carshop.itemcenter.ItemService"
<! -- Spring object implemented by the service -->
ref="itemService"
<! -- Version of the published service, which is user-defined and is 1.0.0 by default. -->
version="1.0.0"
</hsf:provider>
</beans>
```

The preceding example uses basic configuration items for demonstration purposes. To add other configuration items, refer to the following list of provider service attributes.

List of provider service attributes

Attribute	Description
interface	This string-format attribute is required and indicates the service API provided for external applications.
version	This string-format attribute is optional and indicates the version of the service, which is 1.0.0 by default.
clientTimeout	This attribute applies to all methods in the API. However, if the client sets a timeout period for a method using the MethodSpecials attribute, the timeout period configured on the client prevails over that defined for the method.Other methods are not affected by this attribute and still use the timeout period configured on the server.
serializeType	This attribute is optional and indicates the serialization type. Its value is in string format and can be hessian (default) or java.
corePoolSize	This attribute is used to repurpose part of the public thread pool as the core thread pool dedicated to this service.
maxPoolSize	This attribute is used to repurpose part of the public thread pool as the maximum thread pool dedicated to this service.
enableTXC	This attribute enables distributed transaction GTS.
ref	This ref-format attribute is required and indicates the ID of the Spring bean to be published as an HSF service.
methodSpecials	This attribute is optional and used to configure a timeout period (in ms) for each method. With this attribute, different timeout periods can be applied to different methods in an API.This timeout attribute takes precedence over clientTimeout but defers to MethodSpecials on the client.

Example of configuration of provider service attributes

```
<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>
```

Develop consumer services

Service subscription for consumers is coded in two steps:

1. Define a bean using the <hsf:consumer/> tag in the Spring configuration file;
2. Retrieve the bean from the Spring context to locate the service.

Note: The detail folder in the demo project provides the sample code of a consumer service.

Configure service attributes

Similar to that for providers, service attribute configuration for consumers consists of the Maven dependency configuration and Spring configuration.

Add the Maven dependency to the pom.xml file.

The Maven dependency configuration of a consumer is the same as that of a provider. For more information, see Configure service attributes in "Develop provider services."

Add the HSF-specific Spring configuration items to the hsf-consumer-beans.xml.

Add the consumer definition to the Spring configuration file. The HSF framework subscribes to services in the service center according to the configuration file.

```
<?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">
<! -- Example of service consumption-->
<hsf:consumer
<! -- Indicates the Bean ID for retrieving the consumer object by code injection -->
id="item"
<! -- Indicates the service name that corresponds to the service name of the service provider. HSF
queries and subscribes to services according to the combined criteria of interface and version. -->
interface="com.alibaba.edas.carshop.itemcenter.ItemService"
<! -- Indicates the version that corresponds to the version of the service provider. HSF queries and
subscribes to services according to the combined criteria of interface and version. -->
version="1.0.0"
</hsf:consumer>
</beans>
```

Configure service calls

You can configure a service call by referring to the example in the StartListener.java file:

```
public class StartListener implements ServletContextListener{

@Override
public void contextInitialized( ServletContextEvent sce ) {
ApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext( sce.getServletContext() );
// Retrieve the subscribed services according to the bean ID "item" in Spring configurations.
final ItemService itemService = ( ItemService ) ctx.getBean( "item" );

// Call the getItemById method of ItemService.
System.out.println( itemService.getItemById( 1111 ) );
// Call the getItemByName method of ItemService.
System.out.println( itemService.getItemByName( "myname is le" ) );

}
}
```

The preceding example uses basic configuration items for demonstration purposes. To add other configuration items, refer to the following list of service attributes.

List of consumer service attributes

Attribute	Description
interface	This string-format attribute is required and indicates the API of the service to be called.
version	This string-format attribute is optional and indicates the version of the service to be called, which is 1.0.0 by default.
methodSpecials	This attribute is optional and used to configure a timeout period (in ms) for each method.With this attribute, you can apply different timeout periods to different methods in an API. This timeout attribute takes precedence over the timeout setting of the provider.
target	This attribute is primarily used in unit testing and development environments to manually specify the address of the service provider.To specify the address of the provider based on the target service address information pushed from the configuration center, you can specify -Dhsf.run.mode=0 on the consumer.
connectionNum	This attribute is optional and set to 1 by default. It indicates the maximum number of connections to the server.To improve TPS, set this attribute to a larger value to minimize the delay of transferring small amounts of data.

clientTimeout	This attribute indicates the same timeout period (in ms) set for all methods in an API by the consumer.Timeout settings are sorted in descending order of priority as follows: consumer MethodSpecials, consumer API level, provider MethodSpecials, and provider API level.
asyncallMethods	This list-format attribute is optional and indicates that the **asynchronously called** method name list and asynchronous calls are required for calling the service.This attribute is an empty set by default, which indicates that all methods are called synchronously.
maxWaitTimeForCsAddress	This attribute indicates the time during which the thread is blocked to wait for address push when a service is subscribed. Otherwise, the address may not be found due to empty address when the service is called.If the address is not pushed before the blocking time expires, the thread no longer waits and continues initializing subsequent content. **Note**: This parameter is only used to call a service during application initialization.As this parameter extends the startup time, we recommend that you do not use this parameter when no other services need to be called.

Configuration example of consumer service attributes

```
<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>
```

Publish services

After coding, API development, and service configuration, you can directly run the service using Ali-Tomcat in Eclipse or IntelliJ IDEA (for more information, see Configure the Eclipse development environment and Configure the IDEA development environment in Prepare development tools.

The following table lists additional JVM startup parameters that you can configure in the development environment to change HSF behaviors:

Attribute	Description
-Dhsf.server.port	Indicates the port bound to the HSF startup service, which is port 12200 by default.

-Dhsf.serializer	Indicates the serialization type of HSF, which is hessian by default.
-Dhsf.server.max.poolsize	Indicates the maximum size of the thread pool of the HSF provider, which is 600 by default.
-Dhsf.server.min.poolsize	Indicates the minimum size of the thread pool of the HSF provider.The default value is 50.
-DHSF_SERVER_PUB_HOST	Indicates the exposed IP address, which uses the value of -Dhsf.server.ip if not configured.
-DHSF_SERVER_PUB_PORT	Indicates the exposed port that must be monitored locally and authorized for access. It uses the value of -Dhsf.server.port by default or port 12200 if -Dhsf.server.port is not configured.

Query HSF services in the development environment

During development and debugging, if your service is registered and discovered in the lightweight configuration center, you can query the services provided or called by an application in the EDAS console.

The following assumes that you start the EDAS configuration center on an ECS instance whose IP address is 192.168.1.100.

Go to http://192.168.1.100:8080/.

In the left-side navigation pane, click **Service List**. On the page that is displayed, enter the service name, service group name, or IP address to search for the service, and view the service provider and caller.

Note: After the configuration center is started, the address of the first NIC is the service discovery address by default. If the ECS instance of the developer has multiple INCs, set the SERVER_IP variable in the startup script to explicitly bind an address.

Common query cases

Provider list page

Enter the IP address in the search condition box and click **Search** to query the services provided by the instance with the entered IP address.

Enter the service name or service group in the search condition box and click **Search** to query which IP addresses provide the service.

Caller list page

Enter the IP address in the search condition box and click **Search** to query the services called by the instance with the entered IP address.

Enter the service name or service group in the search condition box and click **Search** to query what IP addresses call the service.

Develop advanced features

After following the steps outlined in Quick start, you have developed the basic features of an HSF application. The following section describes how to develop the advanced features of an HSF application based on these basic features.

Download Demos.

Implicit parameter passing (currently, only string-based parameter passing is supported)

Implicit parameter passing is generally used to replace API-based passing for passing simple KV data and it is similar to cookie.

Pass a single parameter

Service consumer:

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

Service provider:

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

Pass multiple parameters

Service consumer:

```
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);
```

Service provider:

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

Note: Implicit parameter passing is only valid for a single call. When the consumer call returns the result, the information in RpcContext is cleared automatically.

Asynchronous calls

Asynchronous calls can be made using either the callback or the future method.

Callback method

If the callback method is configured for the consumer, you must configure a listener that implements the HSFResponseCallback API.After the result is returned, HSF calls the method in HSFResponseCallback.

Note: The listener of the HSFResponseCallback API must not be an internal class. Otherwise, the Pandora classloader returns an error during loading.

XML configuration:

```
<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>
```

The AsynABTestCallbackHandler class implements the HSFResponseCallback API.The DemoApi API has the ayncTest method.

Sample code

```
public void onAppResponse(Object appResponse) {
//Retrieve the value after the asynchronous call.
String msg = (String)appResponse;
System.out.println("msg:"+msg);
}
```

Note:

- Method names are used to identify methods. Therefore, overloaded methods are not differentiated.Methods sharing the same name are set using the same call method.
- HSF calls cannot be initiated in a call.Otherwise, the I/O thread is suspended and cannot be recovered.

Future method

If the future method is configured for the consumer, after you make a call, the returned result is retrieved through public static Object getResponse(long timeout) in HSFResponseFuture.

XML configuration:

```
<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>
```

The sample code is as follows:

Asynchronous processing of a single call:

```
//Initiate a call.
demoApi.ayncTest();
// Process the service.
...
//Directly obtain the message. (If the result is not required, you can skip this step.)
String msg=(String) HSFResponseFuture.getResponse(3000);
```

Concurrent processing of multiple calls:

To process multiple tasks concurrently, retrieve and store the future object and then reuse it after the call.

```
//Define a set.
List<HSFFuture> futures = new ArrayList<HSFFuture>();
```

Concurrent call within a method:

```
//Initiate a call.
demoApi.ayncTest();
```

```
//Retrieve the future object.
HSFFuture future=HSFResponseFuture.getFuture();
futures.add(future);
//Continue calling other services (asynchronous calling is used).
HSFFuture future=HSFResponseFuture.getFuture();
futures.add(future);

// Process the service.
...

//Retrieve and process the data.
for (HSFFuture hsfFuture : futures) {
String msg=(String) hsfFuture.getResponse(3000);
//Process corresponding data.
...
}
```

Generic calls

Generic calls can combine APIs, methods, and parameters for remote procedure calls (RPCs) without relying on any service APIs.

Step 1: Add the generic attribute to the consumer’s XML configuration.

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

Note: “generic” indicates generic parameters, “true” indicates that generic parameters are supported, and “false” (default) indicates that generic parameters are not supported.

DemoApi API method:

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

Step 2: Obtain demoApi to enforce conversion to a generic service.

Import the generic service API

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

Retrieve generic objects

- XML loading method

```
//In a Web project, you can force service conversion after injection using a Spring bean. This
```

```
example is a unit test and therefore must load the configuration file.
ClassPathXmlApplicationContext consumerContext = new
ClassPathXmlApplicationContext("hsf-generic-consumer-beans.xml");
//The forced conversion API is GenericService.
GenericService svc = (GenericService) consumerContext.getBean("demoApi");
```

Code subscription method

```
HSFApiConsumerBean consumerBean = new HSFApiConsumerBean();
consumerBean.setInterfaceName("com.alibaba.demo.api.DemoApi");
consumerBean.setGeneric("true"); // Set generic to true.
consumerBean.setVersion("1.0.0");
consumerBean.init();
// The forced conversion API is GenericService.
GenericService svc = (GenericService) consumerBean.getObject();
```

Step 3: Implement generic APIs.

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

Description of API parameters:

methodName: indicates the name of the method to be called.

parameterTypes: indicates the type of the parameters of the method to be called.

args: indicates the parameter value to be passed.

Step 4: Initiate generic calls.

String-type parameters

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

Object parameters, which must be the same on the provider and consumer

```
// Construct the entity object GenericTestDO, which has the ID and name attributes.
GenericTestDO genericTestDO = new GenericTestDO();
genericTestDO.setId(1980l);
genericTestDO.setName("genericTestDO-tst");
// Use PojoUtils to generate the POJO description of the second party library.
Object comp = PojoUtils.generalize(genericTestDO);
// Call the service in generic mode.
```

```
svc.$invoke("dealGenericTestDO",new String[] { "com.alibaba.demo.generic.domain.GenericTestDO" },
new Object[] { comp });
```

Trace Filter extension

Download Demos.

Basic APIs

```
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);

}
```

Implementation procedure

1. Implement ServerFilter to enable interception on the provider.
2. Implement ClientFilter to enable interception on the consumer.
3. Use the standard META-INF/services/com.taobao.hsf.invocation.filter.RPCFilter file to register Filter.

Implementation example

```
    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));
}



}
```

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

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

Running effect

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

intercept response

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

Optional Filter

In some scenarios, if you have customized the Filter but want to apply it only to certain services, use an optional Filter.To do this, annotate Filter with @Optional, as shown below:

```
```
@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");
}
}
```
```

When a specific service needs to use this Filter, simply advertise the service on the configured bean, as shown below:

```
    <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>
```

In the preceding configuration, all the ServerFilters without the @Optional modifier, including HSFOptionalServerFilter and NoFilter, are used. The name of HSFOptionalServerFilter comes from the @Name modifier configured in the corresponding Filter.

> If the Filter with this name cannot be found, the system displays a prompt, but you still can start or run this Filter.

Perform the unit test

There are two unit test methods in the test environment.

Method 1: Publish and subscribe to services using the LightApi code

Method 2: Publish and subscribe to services using XML configuration

For demos, click Demo download.

Method 1: Publish and subscribe to services using the LightApi code

Add LightApi dependency to Maven.
```

```
 <dependency>
<groupId>com.alibaba.hsf</groupId>
<artifactId>LightApi</artifactId>
<version>1.0.0</version>
</dependency>
```

Note: Please use *1.0.5* or later version of LightApi, or hsf: can not load class {com.taobao.hsf.address.AddressService} after all phase error may appear.

Create ServiceFactory.

The Pandora address must be set, and the parameter indicates the directory where the SAR package is located.If the SAR package address is /Users/Jason/Work/AliSoft/PandoraSar/DevSar/taobao-hsf.sar the parameter is as follows:

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

Use codes to publish and subscribe to the service.

```
 // Publish the service (you can skip this step if the publisher already exists.)
factory.provider("helloProvider")// This parameter is an identifier. After initialization, you can call
provider("helloProvider") to retrieve the corresponding service.
.service("com.alibaba.edas.unit.service.UnitTestService")// Fully-Qualified Class Name (FQCN) of the API
.version("1.0.0")// Version
.impl(new UnitTestServiceImpl())// Corresponding service implementation
.publish();// Publish the service by calling at least service() and version().

// Consume the service.
factory.consumer("helloConsumer")// This parameter is an identifier. After initialization, you can call
consumer("helloConsumer") to retrieve the corresponding service.
.service("com.alibaba.edas.unit.service.UnitTestService")// FQCN of the API
.version("1.0.0")// Version
.subscribe();
factory.consumer("helloConsumer").sync();// Synchronously wait for address push for up to 6 seconds.
UnitTestService log4jService = (UnitTestService) factory.consumer("helloConsumer").subscribe();//
Retrieve the corresponding service using the ID. The subscribe() method returns the corresponding API.
// Call the service method.
System.out.println("bean -> msg rec success:-"+log4jService.print());
```

Method 2: Publish and subscribe to services using XML configuration

Complete the XML configuration of HSF.

Load the configuration file using code.

```
//Load the service provider using XML.
new ClassPathXmlApplicationContext("hsf-provider-beans.xml");
//Load the service consumer using XML.
ClassPathXmlApplicationContext consumerContext=new ClassPathXmlApplicationContext("hsf-
consumer-beans.xml");
//Retrieve the bean.
UnitTestXMLConsumer unitTestXMLConsumer=(UnitTestXMLConsumer)
consumerContext.getBean("unitTestConsumer");
//Call the service.
unitTestXMLConsumer.testUnitProvider();
```

Pandora Boot Developer Guide

Pandora Boot overview

Derived from Pandora, Pandora Boot is more lightweight.

Pandora Boot can directly start a Pandora environment in IDE based on Pandora and FatJar, greatly improving the development and debugging efficiency.

Pandora Boot deeply integrates Spring Boot AutoConfigure to provide you with the convenience of the Spring Boot framework.

Spring Boot users who need to use HSF and users who already use Pandora Boot can use Pandora Boot to develop EDAS applications.

Preparation

To use Pandora Boot for application development, you must configure the following development environment:

Configure the EDAS private server address in Maven: Currently, third-party Spring Cloud packages for Aliware are only available on EDAS private servers. Therefore, you must add a private server address into the Maven configuration file.

Configure the lightweight configuration center: You need to enable the lightweight configuration center for local development and commissioning.The lightweight configuration center provides a lightweight version with EDAS service discovery and configuration management features.

Configure the EDAS private server address in Maven

Note: Maven 3.x or later is required. Add the EDAS private server address in the Maven configuration file settings.xml.

Add EDAS private server settings

Add the following settings in the Maven configuration file settings.xml, whose path is generally ~/.m2/settings.xml:

```
<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>
```

Download the setting.xml sample file.

Check whether the settings have been successfully added

Run the mvn help:effective-settings command on the CLI.

1. If no error is returned, the setting.xml file is correctly formatted.
2. If edas.oss.repo is included in profiles, the private server settings have been added to profiles.
3. If edas.oss.repo is included in activeProfiles, the edas.oss.repo private server has been activated.

Note: If no error is returned when you run the Maven packaging command on the CLI, but IDE still cannot download the dependency, close IDE and start it again or search for a solution in the documentation for configuring Maven in IDE.

Develop applications

Service registration and discovery

The following describes how to use Pandora Boot to develop applications and implement service registration and discovery.

Download the demo source code sc-hsf-provider and sc-hsf-consumer.

Create a service provider

Create a Maven project named sc-hsf-provider.

Introduce the necessary dependencies to the pom.xml file:

```
 <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>
```

Although the HSF service framework is independent of the web environment, web-related features are required when EDAS is used to manage the lifecycle of applications. Therefore, you must add a dependency for spring-boot-starter-web.

If you do not want to configure the parent of the project as spring-boot-starter-parent, you can add dependencyManagement and set scope=import as follows to manage dependency versions.

```
 <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>
```

Define a service API, and create an API class of com.aliware.edas.EchoService.

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

The HSF service framework enables service communication based on APIs. When an API is defined, producers use this API to implement and release specific services, and consumers also use this API to subscribe to and consume services.

The API com.aliware.edas.EchoService provides an echo method, which also means that the service com.aliware.edas.EchoService provides an echo method.

Add the implementation class EchoServiceImpl of the service provider, and publish the service using annotations.

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

In addition to the API name serviceInterface, HSF also requires the serviceVersion (service version) to uniquely identify a service. In this case, the serviceVersion attribute in the HSFProvider annotation is set to “1.0.0”.Then, the service to be published can be identified by the serviceInterface com.aliware.edas.EchoService and serviceVersion 1.0.0 combination.

The configuration in the HSFProvider annotation has the highest priority. If it is not configured in the HSFProvider annotation, the global configuration of these properties is checked in the file resources/application.properties when the service is published.If neither is configured, the default values in the HSFProvider annotation are used.

Configure the application name and the listener port number in the application.properties file in resources.

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

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

Best practices: We recommend that you configure both the **service version** and **service timeout** in the application.properties file.

Add main function entrance for starting the service.

```
 @SpringBootApplication
public class HSFProviderApplication {

public static void main(String[] args) {
// Start Pandora Boot to load the Pandora container
PandoraBootstrap.run(args);
SpringApplication.run(ServerApplication.class, args);
// This indicates that the service has been started, and a thread waiting time is set.This prevents the
container from exiting due to users who exit after running the service code.
PandoraBootstrap.markStartupAndWait();
}
}
```

Create a service consumer

In this example, we create a service consumer that calls the service provider using the API provided by HSFProvider.

Create a Maven project named sc-hsf-consumer.

Introduce the necessary dependencies to the pom.xml file:

The Maven dependencies for HSFConsumer and HSFProvider are exactly the same.

```
<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>
```

Copy the service API (including the package name) com.aliware.edas.EchoService published by the service provider to a local machine.

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

Use an annotation, to inject the **service consumer** instance into the Context of Spring.

```
 @Configuration
public class HsfConfig {

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

Best practices: Configure @HSFConsumer once in the Config class, and inject and use it in multiple places through @Autowired.Usually, an HSF consumer is used in multiple places, but you do not have to mark each place where it is used with @HSFConsumer.You can write a unified Config class and inject it directly wherever it is needed by using @Autowired.

To facilitate the test, an HTTP API of /hsf-echo/* is exposed through the SimpleController. Calls to the HSF service provider are internally implemented in the API /hsf-echo/*.

```
 @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);
}
}
```

Configure the application name and the listener port in the application.properties file in resources.

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

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

Best practices: We recommend that you configure both the **service version** and **service timeout** in the application.properties file.

Add main function entrance for starting the service.

```
 @SpringBootApplication
public class HSFConsumerApplication {

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

Local development and debugging

Start the lightweight configuration center

The lightweight configuration center, which includes a lightweight version of the EDAS service registration and discovery server, must be started for local development and debugging. For more information, see the lightweight configuration center.

Start the application

The application can be locally started in two ways.

Start in IDE

Configure the startup parameter -Djmenv.tbsite.net={$IP} in VM options, and start the application directly by using the main method.Here, {$IP} indicates the address of the computer on which the lightweight configuration center is started.For example, if the center is started on the current computer, $IP is 127.0.0.1.

Rather than configuring JVM parameters, you can also directly modify the hosts file to bind jmenv.tbsite.net to the IP address of the instance on which the lightweight configuration center is started.For more information, see the lightweight configuration center.

Start with FatJar

Add the FatJar packaging plugin.

To package the pandora-boot project into FatJar with Maven, add the following plugin in the pom.xml file.

To prevent conflicts with other packaging plugins, do not add any other FatJar plugins to the build plugin.

```
<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>
```

After adding the plugins, run the Maven command mvn clean package in the home directory of the project to create a package. The created FatJar file is in the target directory.

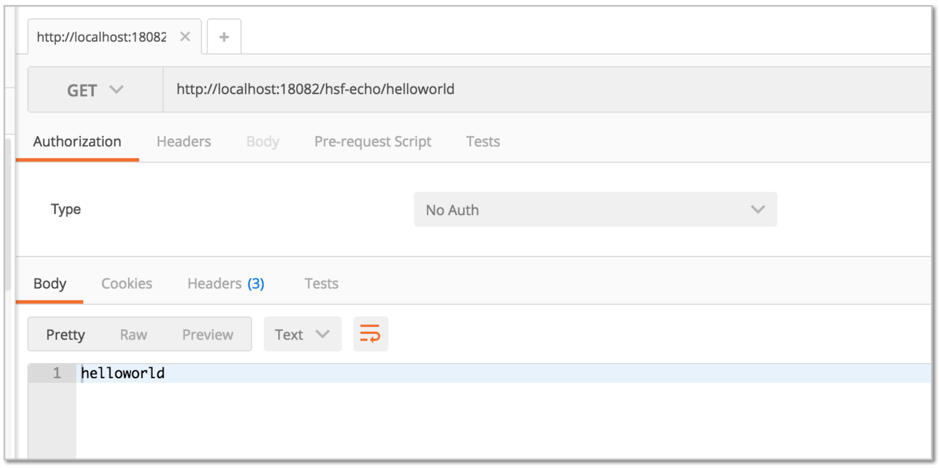
Run the Java command to start the application.

```
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
```

Note: The path specified by -Dpandora.location must be a full path followed by sc-hsf-provider-0.0.1-SNAPSHOT.jar.

Demonstration

Enable the service and check whether it can be called.



GET http://localhost:18082/hsf-echo/helloworld

Type: No Auth

```
helloworld
```

Asynchronous calls

HSF enables two types of asynchronous calling, Future and Callback.

To demonstrate asynchronous calls, we have published a new service: com.aliware.edas.async.AsyncEchoService.

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

The service provider implements AsyncEchoService and uses annotations to publish it.

```
 @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;
}
}
```

Likewise, the subsequent configuration steps and application startup processes are the same.

Note: The logic of asynchronous calls is modified on the consumer rather than the server.

Future

To enable Future-type asynchronous calls for the consumer end, use annotations to inject instances of the service consumer into Context of Spring, and configure the method name of asynchronous calls in futureMethods properties of @HSFConsumer annotations.

This Future method of com.aliware.edas.async.AsyncEchoService is marked as Future-type asynchronous calls.

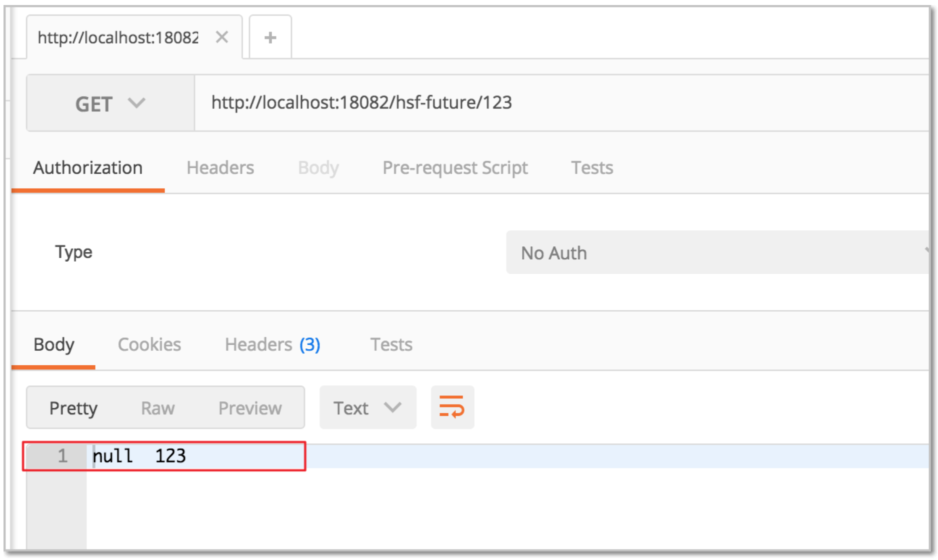
```
 @Configuration
public class HsfConfig {
@HSFConsumer(serviceVersion = "1.0.0", futureMethods = "future")
private AsyncEchoService asyncEchoService;
}
```

After the method is marked as Future-type asynchronous calls, the actual return value of the method during synchronous execution is null, and the call result must be obtained through HSFResponseFuture.

TestAsyncController is used for demonstration. The sample code is as follows:

```
@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;
}
}
```

Call /hsf-future/123; the str1 value is null, and str2 value is 123, which is the actual return value.

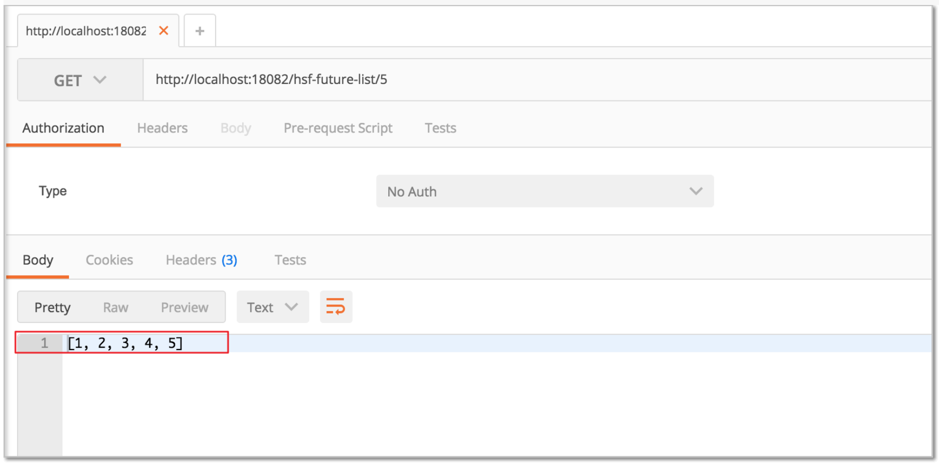


http://localhost:18082/hsf-future/123

```
null  123
```

If a series of operation return values are needed for service processing, refer to the following call method.

```
@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";
}
}
```


GET http://localhost:18082/hsf-future-list/5

Authorization — Type: No Auth

Body — Pretty, Text

```
[1, 2, 3, 4, 5]
```

Callback

To enable Callback-type asynchronous calls for the consumer end, create a class to implement the HSFResponseCallback API and use @Async annotations for configuration.

```
 @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();
}
}
```

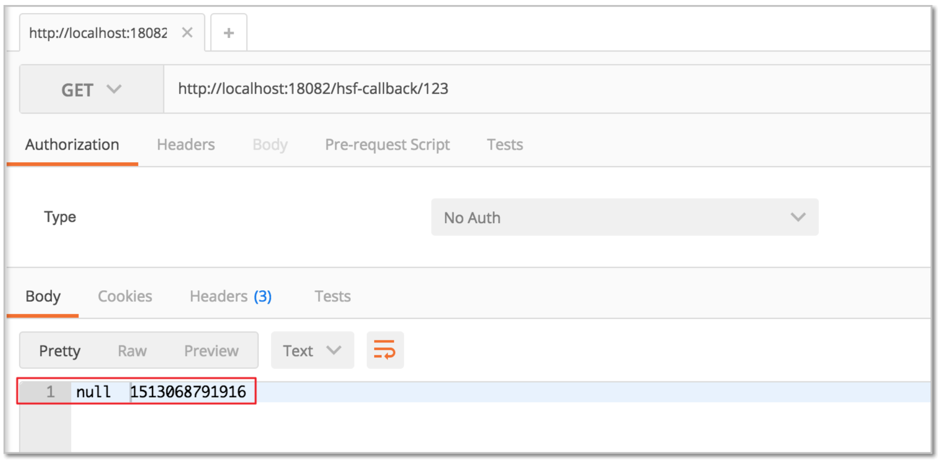
AsyncEchoResponseListener implements the HSFResponseCallback API, and sets the interfaceName to AsyncEchoService.class and the methodName to callback in the @Async annotation.

The Callback method of com.aliware.edas.async.AsyncEchoService is marked as Callback-type asynchronous calls.

Similarly, TestAsyncController is used for demonstration. The sample code is as follows:

```

 @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;
}
```

After the feature is called, the following result is returned:



http://localhost:18082/hsf-callback/123

Type No Auth

```
null  1513068791916
```

After the consumer end configures the callback method to Callback-type asynchronous calling, the synchronous return value is actually null.

After the result is returned, HSF calls the method in AsyncEchoResponseListener, and the actual return value of calling can be obtained using the onAppResponse method.

Use CallbackInvocationContext to transmit the contextual information of the calling to

Callback.

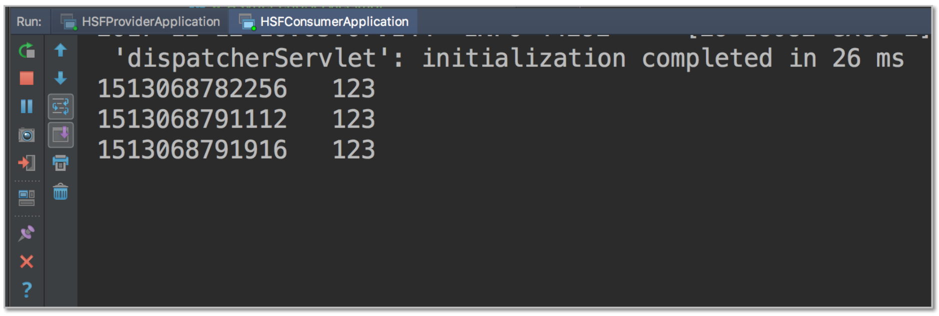
The sample code for calling is as follows:

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

The sample code of AsyncEchoResponseListener is as follows:

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

The output on the console is 1513068791916 123, which means that the onAppResponse method of AsyncEchoResponseListener has used the CallbackInvocationContext to receive the timestamp transferred before the call.



```
 'dispatcherServlet': initialization completed in 26 ms
1513068782256   123
1513068791112   123
1513068791916   123
```

Perform the unit test

The implementation of spring-cloud-starter-hsf depends on Pandora Boot, and unit testing of Pandora Boot is enabled through PandoraBootRunner and seamlessly integrated with SpringJUnit4ClassRunner.

The procedure of unit testing in service providers is demonstrated as follows for your reference.

Add dependency for spring-boot-starter-test in Maven.

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

Develop the test code.

```
 @RunWith(PandoraBootRunner.class)
@DelegateTo(SpringJUnit4ClassRunner.class)
// Add the test class. In this case, both the startup class of Spring Boot and this test class are required.
@SpringBootTest(classes = {HSFProviderApplication.class, EchoServiceTest.class })
@Component
public class EchoServiceTest {

/**
* If you are using @HSFConsumer, you must add this class to the @SpringBootTest classes and use it to inject objects to prevent abnormal class conversion during generalization.
*/
@HSFConsumer(generic = true)
EchoService echoService;

//Common calls
@Test
public void testInvoke() {
TestCase.assertEquals("hello world", echoService.echo("hello world"));
}
//Generic calls
@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);
}
//Return the value 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(""));
}
}
```

Develop RESTful applications (not recommended)

Implement service discovery

This section describes how to enable the service discovery function of EDAS for your RESTful applications.

Download the Demo source code sc-vip-server and sc-vip-client.

Create a service provider

The service provider in this example provides a simple echo service and registers itself with the service discovery center.

Create a Spring Cloud project named sc-vip-server.

Introduce necessary dependencies in pom.xml:

```
 <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>
```

If you do not want to configure the parent of the project as spring-boot-starter-parent, you can add dependencyManagement and set scope=import as follows to manage dependencies.

```
 <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>
```

Add the code of the service provider, in which the annotation of @EnableDiscoveryClient indicates that the service registration and discovery features must be enabled for the service.

```
 @SpringBootApplication
@EnableDiscoveryClient
public class ServerApplication {

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

Create an EchoController to provide simple echo service.

```
 @RestController
public class EchoController {
@RequestMapping(value = "/echo/{string}", method = RequestMethod.GET)
public String echo(@PathVariable String string) {
return string;
}
}
```

Configure the service name and the listener port number in the application.properties file in resources.

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

Create a service consumer

A service consumer is created in this example. The service consumer calls the service provider through **RestTemplate**, **AsyncRestTemplate**, and **FeignClient**.

Create a Spring Cloud project named sc-vip-client.

Introduce necessary dependencies in pom.xml:

```
 <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>
```

To demonstrate the use of FeignClient, an additional dependency of spring-cloud-starter-feign is added to the pom.xml file compared to that of the service provider.

Different from the server, the clients support other functions in addition to service enabling and registration. In addition to service enabling and registration, two more configurations must be added to use the three clients of **RestTemplate**, **AsyncRestTemplate**, and **FeignClient**:

- Add the annotation of @LoadBalanced to combine RestTemplate, AsyncRestTemplate, and service discovery.

Activate FeignClients by using the annotation of @EnableFeignClients.

```
@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();
}

}
```

Complete the configuration of FeignClient of EchoService before using it. Configure the service name and the HTTP request corresponding to the method. The service name is service-provider configured in the sc-vip-server project. The code is as follows:

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

Create a Controller for call test.

/echo-rest/ verifies the call of the service provider through RestTemplate.

/echo-async-rest/ verifies the call of the service provider through AsyncRestTemplate.

/echo-feign/ verifies the call of the service provider through FeignClient.

```
 @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);
}

}
```

Configure the application name and the listener port number.

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

Test the RESTful services

Start the Lightweight Configuration Center

The Lightweight Configuration Center must be started for local development and commissioning, which includes a lightweight version of the server of EDAS service registration and discovery. See Lightweight Configuration Center for details.

Start the services

The services can be locally started in two ways.

Start in IDE

To start the service in IDE, configure the startup parameter -Dvipserver.server.port=8080 in VM options and start the service directly using the main method.

If your Lightweight Configuration Center and the service are deployed on different computers, hosts binding is required. See Lightweight Configuration Center for details.

Start by using FatJar

Add FatJar packaging plugins.

To package pandora-boot project into FatJar by using Maven, the following plugins should be added in pom.xml.To avoid conflicts with other packaging plugins, do not add other FatJar plugins in build plugin.

```
<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>
```

After adding the plugins, run the maven command mvn clean package under the home directory of the project to create a package. The created FatJar file is located under the target directory.

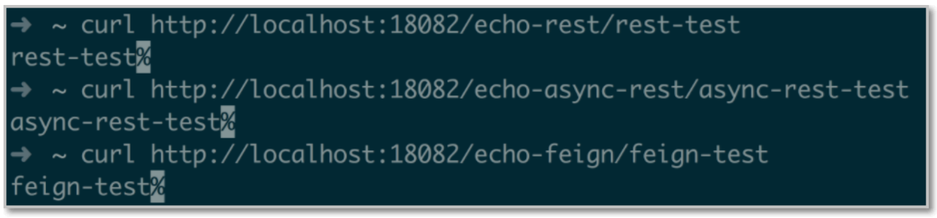
Start the service using Java command.

```
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
```

NOTE: The path specified by -Dpandora.location must be a full path placed before sc-vip-server-0.0.1-SNAPSHOT.jar.

Demonstration

Start the service and call the service provider through the clients. Each call is successful.



```
➜ ~ 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%
```

FAQ

Failed to enable service discovery for AsyncRestTemplate.

AsyncRestTemplate is not enabled with service discovery until recently and versions after Dalston are required. See the pull request for details.

FatJar packaging plugin conflict

To avoid conflicts with other packaging plugins, do not add other FatJar plugins in build plugin.

Can taobao-hsf.sar be included during packaging?

Yes, but this is not recommended.

Modify the pandora-boot-maven-plugin plugin and set excludeSar as false to automatically include taobao-hsf.sar during packaging.

```
  <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>
```

In this way, the package can be started without a configured Pandora address.

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

Restore the configuration to excluding the SAR package before deploying an application in EDAS.

Implement distributed tracing

To reduce development cost and increase development efficiency, EDAS provides EagleEye, a component for service call tracing.Once EagleEye tracking is configured in the code, you can directly use the tracing function of EDAS without considering other processes including log collection, analysis, or storage.

This document introduces how to enable the distributed tracing function for your services.

Download the Demo source code service1 and service2.

How to use EagleEye

Configure the EDAS private server address in Maven

Currently, packages of Pandora Boot Starter are only published on the private servers of EDAS. You need to add the private server address in the Maven configuration file. See **Configure the EDAS private server address in Maven** in Prepare development tools for details.

NOTE: Maven 3.x or later is required. Add the EDAS private server address in the Maven configuration file settings.xml. Click to download the sample file.

Modify the code

You can connect Spring Cloud to EDAS EagleEye according to the following three steps.

Add the following public configurations to pom.xml.

```
 <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>
```

Add two lines in the main function. The original content of the main function is as follows:

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

The modified content of the main function is as follows:

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

Add FatJar packaging plugins.

To package the pandora-boot project into FatJar by using Maven, you need to add the following plugins in pom.xml.

To avoid conflicts with other packaging plugins, do not add other FatJar plugins in build plugin.

```
 <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>
```

After completing the preceding steps, you can directly use the distributed tracing function of EDAS without setting up any collection or analysis system.

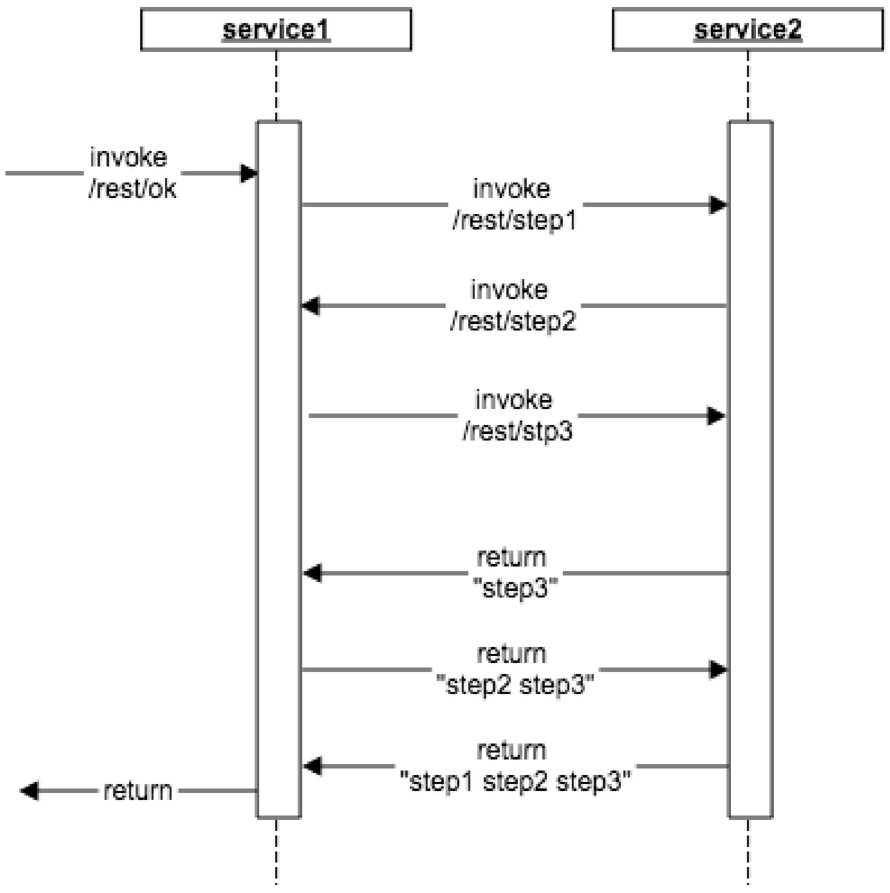
Distributed tracing example

Source code

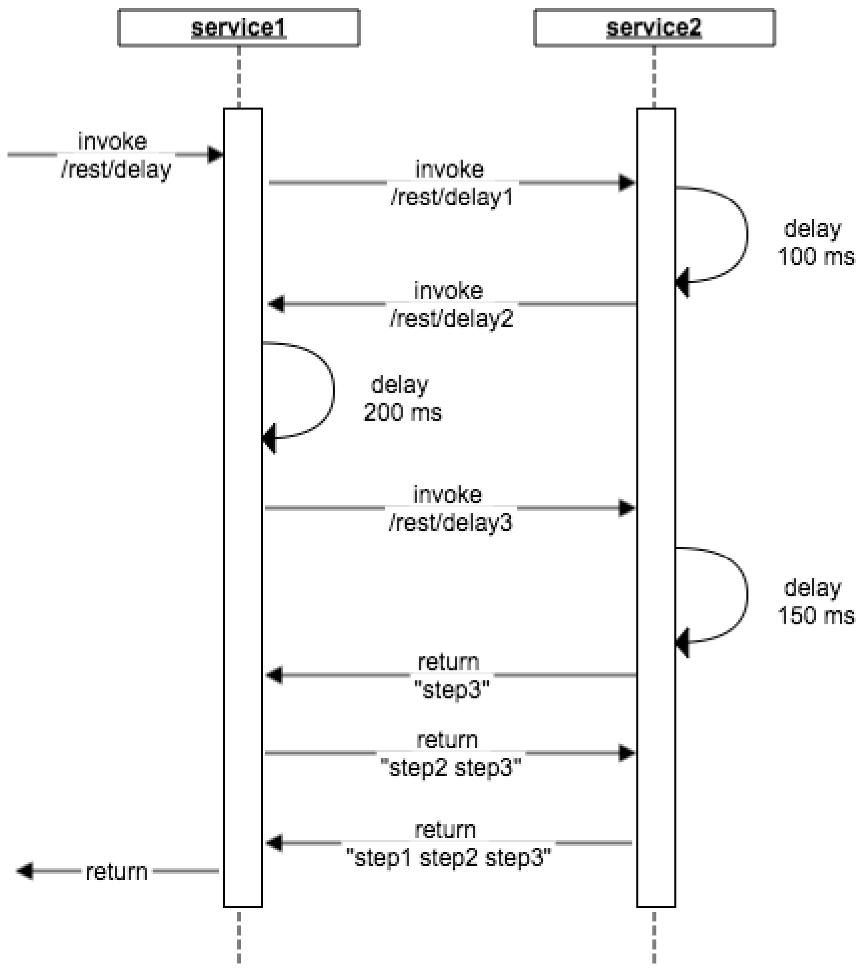
To demonstrate how to use the distributed tracing function, two code demos service1 and service2 are used in this example.

service1 provides the entrances for three demonstrating scenes.

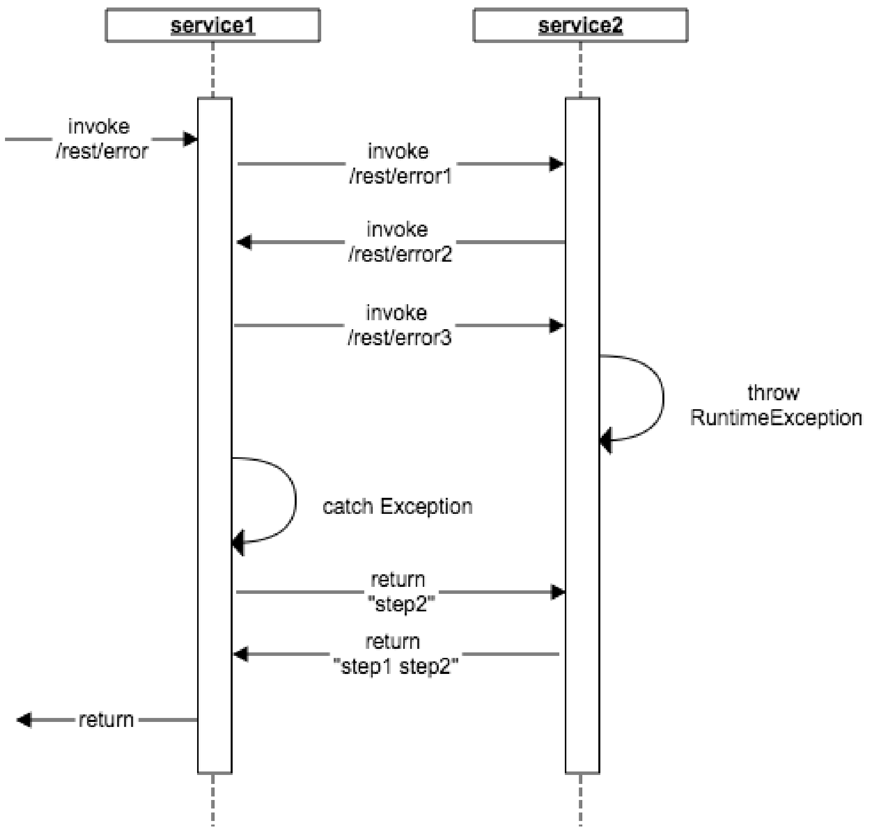
/rest/ok for normal calls



/rest/delay for calls with large delay



/rest/error for calls with exceptions or errors



Deploy the services

The collection and analysis functions of EagleEye are both set up in EDAS. To demonstrate the trace view function, we must deploy the two applications service1 and service2 in EDAS.

When creating an application, choose the latest version of the container.

Add FatJar packaging plugins, and run the mvn clean package command under the directory of the project to create a FatJar package.

Deploy applications on EDAS by uploading the FatJar services in the target directory without configuration.

To view the trace information after the deployment, calling methods corresponding to the entrances of three demonstrating scenes of service1 are required.

You can run the curl http://{$ip:$port}/rest/ok command.Or you can use tools such as postman, or directly call the methods in browsers.

To observe the response, it is recommended that you call the methods in script mode for multiple times.

View the call trace

Log on to the EDAS console and enter the deployed services.

In the left-side navigation pane on the application details page, choose **Application Monitoring** > **Service Monitoring**.

On the service monitoring page, click **RPC Service Provided**, and then **View Trace**.

Demonstration of other clients

At the same time, the automatic tracking of EagleEye for RestTemplate, AsyncRestTemplate, and FeignClient is demonstrated separately in the three URIs /echo-rest/str, /echo-async-rest/str, and /ech-feign/str of service1.

FAQ

Tracking support

Now, EagleEye of EDAS supports the automatic tracking for requests called by RestTemplate, AsyncRestTemplate, and FeignClient. **We will provide the automatic tracking for more components in the future**.

AsyncRestTemplate

As AsyncRestTemplate requires to perform the modification of tracking support during class instantiation, injection of object eagleEyeAsyncRestTemplate, which supports service discovery by default, is required to enable tracing.

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

FatJar packaging plugin

To package the pandora-boot project into FatJar by using Maven, you need to add the pandora-boot-maven-plugin in pom.xml.To avoid conflicts with other packaging plugins, do not add other FatJar plugins in build plugin.

More information

See Enable the EDAS distributed tracing function for Spring Cloud for more information about the distributed tracing function and EagleEye.

Migrate Dubbo to HSF (not recommended)

Dubbo is an open source RPC framework, while HSF is another RPC framework supported by EDAS.Before the implementation of Dubbo for EDAS, this document provided a solution to convert Dubbo to HSF, enabling rate limiting and degradation, distributed tracing, service analysis, and other functions.Now, EDAS supports Dubbo and offers service governance, distributed tracing, and more functions. For more information, see Quick start.**For this reason, we recommend that new users do not use this method**.

This topic describes how to convert Dubbo for the Spring Boot programming model to HSF by modifying the code.The process of application development is not described in detail here.

Download the Demos for converting Dubbo to HSF.

Add a Maven dependency

Add spring-cloud-starter-pandora dependencies in the project file pom.xml.

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

Add or modify the packaging plug-in for Maven

Add or modify the Maven packaging plugin in the project file pom.xml.**To prevent conflicts with other packaging plugins, do not add any other FatJar plugins to the build plugin**.

```
<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>
```

Modify the code

In the Spring Boot startup class, add these two lines for loading Pandora:

```
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();
}
}
```

Container versions

Version	Release date	Basic edition	Change
3.5.0	2018-9-10	3.4.7	1. Upgraded eagleeye-core to V1.7.4.8. Fixed the problem of returning garbled Chinese parameter values for an URL request of the Web

			application. 2. Upgraded HSF to V2.2.6.7-edas. Fixed the problem of failing to obtain an HSF service list by running the Pandora QoS command. 3. Removed the ons-client plugin because of the possible conflict between the JAR Package for the ons-client plugin and that for the application.
3.4.7	2018-8-1	3.4.6	Upgraded ONS to V1.7.8-EagleEye. Eliminated the class conflicts resulted from the MQ Trace function.
3.4.6	2018-7-5	3.4.5	<Ol> - Upgraded

			HSF to V2.2.6.1. - <Ol> - Support for the CSB function. - Fixed serialization errors in some scenarios. - Fixed the problem of strong VIP dependency. - Support for health check on Dubbo in the Spring Boot runtime environment. - Upgraded config-client to V1.9.6. Support for dynamic adjustment of the maximum number of registrations. - Upgraded Sentinel to V2.12.12-edas. Support for

			Spring Boot 2. - </Ol>
3.4.5	2018-6-14	3.4.4	Upgraded ACM to V3.8.10. Fixed the problem of ineffective multi-tenant monitoring over a native interface.
3.4.4	2018-5-18	3.4.3	1. The time-out value is 0 during asynchronous processing on HSF provider or a local call, resulting in a time-out error. 2. Some peer IP attributes are missing in RpcContext for using Dubbo on EDAS. 3. Support for use of the service tags on Dubbo in the AOP scenario. 4. Unsupported if the Bool value is included in Map

			during a generalized call of HSF.
3.4.3	2018-4-24	3.4.1	1. Upgraded the Diamond to V3.8.8. 2. Fixed the problem of failing to locate the print certificate and improved the security level. 3. Upgraded EDAS-Assist to V2.0. 4. Optimized the port availability detection logic and upgraded fastjson to V1.2.48.
3.4.1	2018-3-15	3.4.0	1. Upgraded hsf-plugin. Support for dubboX. 2. Upgraded diamond-client and configcent

			er-client. 3. Upgraded edas-assit and canceled the port check after the specified port value is displayed.
3.4.0	2018-3-7	3.3.9	1. Upgraded edas-assist and improved the speed to check the available ports. Support for dynamic setting of a CSP interface. 2. Provided the ConfigCenter version for tenants. 3. Upgraded configclient to ensure unified internal and external consumers

			. Support for CS2.0 and CS3.0 providers.
3.3.9	2018-1-17	3.3.8	1. Upgraded HSF and solved the problem of ZooKeeper blocking. 2. Upgraded Sentinel to provide additional system protection (effective only after you introduce the console push rules). 3. Modified the links of default errors for internation alization.
3.3.8	2018-1-3	3.3.6	1. Upgraded Sentinel to provide an interface that dynamicall y generates metadata. 2. Upgraded

			HSF to provide the tid transmission feature. 3. Upgraded ons and fixed the problem of ineffective link tracing.
3.3.6	2017-12-20	3.3.4	1. When the same exception is thrown repeatedly, the header information increases constantly, resulting in a ultra-long prompt message. 2. EagleEye parses an exception into UNKnown, affecting call trace parsing and topological display of an application.

3.3.4	2017-11-30	3.3.3	1. Upgraded Diamond to the latest version to be compatible with ACM. 2. Fixed a series of problems, including HSF generalization, Unit dependency, resolution exception of multiple ZK addresses, and InetAddress serialization. Support for setting whitelist rules. 3. Fixed the problem that the settings on the custom throttling and downgrade page have to wait for about 30 seconds to

			take effect. 4. EagleEye supports health check, with alimetric and tomcat monitor features. 5. Upgraded ons-client to configure message cache size in MQ on the consumer.
3.3.3	2017-10-18	3.3.2	1. Support for automatic registration of applications (disabled by default). 2. Fixed the problem of HSF file handle occupation. 3. Sentinel supports HSF2.2,4. Enhanced the Pandora QoS

			command.
3.3.2	2017-10-18	3.3.1	1. Fixed the problem of hsf.lock handle occupation on HSF. 2. Added Redis tracing. 3. Upgraded tddl-driver for online comprehensive stress testing. 4. Enhanced the Pandora QoS command.
3.3.1	2017-07-13	3.3.0	Upgraded tddl-driver separately for online comprehensive stress testing.

Microservice Management

Overview

Data-driven operations is an important set of functions within EDAS. The most important data-driven operations function is distributed link analysis.

Distributed link analysis analyzes every service distributed system call, all sent and received troubleshooting messages of interest, and all database access to help you precisely identify system bottlenecks and risks.

Data-driven operations provide these functions:

Trace query

By setting query conditions, you can accurately find businesses with poor performance or exceptions.

Trace details

Based on the trace query results, you can view detailed information for slow or erroneous businesses and reorganize their dependencies. This information allows you to identify frequent failures, performance bottlenecks, strong dependencies, and other problems. You can also evaluate business capacities based on link call ratios and peak QPS.

Service topology

The topology intuitively presents the calling between services and relevant performance data.

Service query

You can view the HSF, Spring Cloud and Service Mesh services in the specific region and namespace.

Service statistics

Service statistics shows the Total Calls in the Last 24 Hours, Average RT(ms) in the Last 24 Hours and Total Errors in the Last 24 Hours of current services in the specific region.

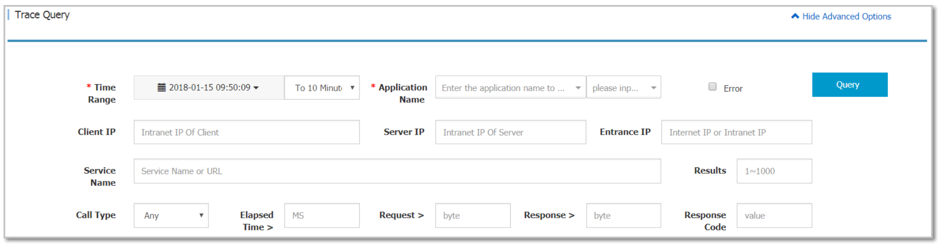
Trace query

By using the trace query function, you can view the status of the invocation trace in the system, especially for tasks that are slow or have encountered an error.

Log on to the EDAS console and Select **Microservice Management** > **Trace Query** in the left-side navigation pane.

Click **Show Advanced Options** in the upper-right corner of the **Trace Query** page to display more query conditions.

Specify the query conditions and click **Query**.



* Time Range	2018-01-15 09:50:09 / To 10 Minut	* Application Name	Enter the application name to ... / please inp...	Error	
Client IP	Intranet IP Of Client	Server IP	Intranet IP Of Server	Entrance IP	Internet IP or Intranet IP
Service Name	Service Name or URL			Results	1~1000
Call Type	Any	Elapsed Time >	MS	Request >	byte
Response >	byte	Response Code	value		

The descriptions for the parameters of the invocation traces (advanced query conditions) are as follows:

Time range: Click the time selector, set the query start time, and then select the end time. The options for the end time are "This Second", "To 1 Minutes Later", and "To 10 Minutes Later". Therefore, the latest time periods are: last second, last one minute, and last ten minutes.

Application name: Select an application from the drop-down list. You can also enter a keyword to search for an application. Manual input of an application name is not supported.

Call type: Select the call type to query from the drop-down list. Options are HTTP, HSF provider, HSF consumer, MySQL, Redis cache, message sending, and message receiving.

Set the threshold values for time elapsed, request, or response for querying slow tasks in the system.

Select the **Error** check box in the upper-right corner to query the error cases only.

Specify other parameters as needed.

In the query result, click on a slow or erroneous task to view trace details.

For the procedure to view the trace details, see Call trace details.

Trace details

The trace details function enables you to query by TraceId the details of a specific service invocation trace in a selected region.

The trace details page displays the trace of the RPC service calls, not including local method calls.

The trace details function is used mainly for tracking the consumed time and occurred exceptions at each point of the distributed service calls. Local methods are not the core content of the calls, so it is recommended that you use logs to track the consumed time and occurred exceptions for local methods. For example, the trace details page will not display the local trace of methodA() calling localMethodB() and localMethodC(). Therefore, it could happen that the elapsed time on a parent node is longer than the total elapsed time on all subnodes.

You can log on to the EDAS console and Select **Microservice Management** > **Trace Details** in the left-side navigation pane to view the details of a service invocation trace. However, a more typical scenario is to view the trace details of the slow or erroneous services. The following example demontrates how to view the trace details entering from **Trace Query** on the left-side menu bar.

In the trace query result, find the HSF method, DB request, or other RPC service call that consumes the longest time.

For DB, Redis, MQ, or other simple calls, find out the reason why accesses to these nodes are slow and check whether they are caused by slow SQL or network congestion.

For HSF methods, further analyze the reason why the method consumes so much time.

Confirm the time consumed by a local method.

Hover the cursor over the time bar on the method row, and in the displayed page, view the elapsed time for the client to send the request, the elapsed time for the server to process the request, and elapsed time for the client to receive the response.

If it takes a long time for the server to process the request, analyze the tasks. Otherwise, conduct the analysis using the method that is used for analyzing call timeout.

Check whether the total time consumed on subnodes is close to that consumed on the method.

If the time difference is small, it indicates that most of the time is consumed on network calls. In this case, reduce network calls as many as possible to shorten the time consumed on each method.

The preceding figure shows that the same method is cyclically called. Instead, it could be just called once in batch.

If the time difference is large, for example, the time consumed on the parent node is 607 ms while the total time consumed on the subnodes does not reach 100 ms. Then it indicates most of the time is consumed on the task logic of the server itself, rather than the RPC service call.

Locate the time-consuming call.

By looking at the time bars to first locate the call before which much time is consumed. The time is purely consumed by the local logic, for which further troubleshooting is required.

After locating the time-consuming logic, review the codes or add logs to the codes to locate the errors.

If it is found that the codes do not consume so much time, perform the following step.

Check whether GC occurred at that time. Therefore, the gc.log file is important.

Locate the timeout error.

An timeout error occurs. Perform the following steps to evaluate the time.

The time is divided into three parts:

Client sends request (0 ms): indicates the time duration from the client sends the request to the server recieves it. This process includes serialization, network transmission, and deserialization. If this process takes a long time, consider if a consumer GC should be triggered. It will take a long time if the object for serialization or deserialization is large, the network is under great transmission pressure, or the provider GC occurs.

Requests processed on server (10,077 ms): indicates the time duration from the server recieves the request to the server returns the response to the client. The time is taken only by the server to process the request, not including other operations.

Client receives response (3,002 ms): indicates the time duration from the server sends the reponse to the client receives it. As the timeout time of 3s is set, the server directly returns timeout after 3s, but the server is still processing the request. If this process consumes much time, perform troubleshooting using the same method that is used for the client.

Service topology

The service topology function is used to view the real-time (last second) call topology between applications in the system.

Log on to the EDAS console, and Select **Microservice Management** > **Service Topology** in the left-side navigation pane.

View the service topology.

The Service topology shows the real-time (last second) call topology between all applications under the current account.

Hover your cursor over an application to view the call topology for this application.

Click on an application to view its call topology and traffic data.

Traffic data refers to the current application’ s QPS, including:

Source traffic: The QPS for calls from other applications to this application.

Call traffic: The QPS for calls from this application to other applications.

Redis tracing

Function overview

After Redis tracing support is added, whenever applications access and perform operations on Redis, the process is recorded in EagleEye trace logs and EDAS collects, analyzes the statistics of the logs. Then information about Redis calls is displayed on the tracing and call analysis page of the EDAS platform.

Supported scope

Due to the wide range of Redis database variants and the usability of Spring Data, Redis trace support is only available for Spring Data Redis of 1.7.4.RELEASE. If you use any other database (for example, Jedis) than Spring Data Redis, you cannot view relevant information on the EagleEye trace interface (which is accessible from **Digital Operations** > **Trace Details** on the left-side menu bar of the EDAS console).

Note: If you use Spring Data Redis later than 1.7.4.RELEASE and the version does not support the provided functions, open a ticket to consult with us.

Usage instructions

For applications on the EDAS platform, Redis trace support replaces Spring Data Redis and is used in the same way as Spring Data Redis. For usage instructions on Spring Data Redis, see the user guide. At the code level, EDAS is compatible with Spring Data Redis 1.7.4-RELEASE. To enable Redis tracing support, follow these steps:

Open the {user.home}/.m2/settings.xml file to configure the local Maven repository.

```
 <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>
```

Activate the corresponding profile:

```
 <activeProfiles>
<activeProfile>edas.oss.repo</activeProfile>
</activeProfiles>
```

Add dependency to the pom.xml file in the Maven project.

```
<dependency>
<groupId>com.alibaba.middleware</groupId>
<artifactId>spring-data-redis</artifactId>
<version>1.7.4.RELEASE</version>
</dependency>
```

Redis command support

The following tables list the Redis commands supported by Spring Data Redis and the support for EagleEye trace logs.

Key-type operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks
Key	DEL	RedisOperations.delete	Y	
	DUMP	RedisOperations.dump	Y	
	EXISTS	RedisOperations.hasKey	Y	
	EXPIRE	RedisOperations.expire	Y	
	EXPIREAT	RedisOperations.expireAt	Y	
	KEYS	RedisOperations.keys	Y	

	MIGRATE	N		
	MOVE	RedisOperations.move	Y	
	OBJECT	N		
	PERSIST	RedisOperations.persist	Y	
	PEXPIRE	RedisOperations.expire	Y	
	PEXPIREAT	RedisOperations.expireAt	Y	
	PTTL	RedisOperations.getExpire	Y	
	RANDOMKEY	RedisOperations.randomKey	Y	
	RENAME	RedisOperations.rename	Y	key: oldKey : ${oldKey};newKey:${newKey}
	RENAMENX	RedisOperations.renameIfAbsent	Y	
	RESTORE	RedisOperations.restore	Y	
	SORT	RedisKeyCommands.sort	Y	key: query:${SortQuery}
	TTL	RedisOperations.getExpire	Y	
	TYPE	RedisOperations.type	Y	
	SCAN	RedisKeyCommands.scan	**N**	

String-type operations

Data structure/Object	**Operation**	**Spring Data Redis method**	**EDAS support for EagleEye tracing (Y/N)**	**Remarks**
String	APPEND	ValueOperations.append	Y	
	BITCOUNT	N		
	BITOP	N		

	BITFIELD	N		
	DECR	ValueOperations.increment	Y	
	DECRBY	ValueOperations.increment	Y	
	GET	ValueOperations.get	Y	
	GETBIT	ValueOperations.getBit	Y	
	GETRANGE	ValueOperations.get	Y	
	GETSET	ValueOperations.getAndSet	Y	
	INCR	ValueOperations.increment	Y	
	INCRBY	ValueOperations.increment	Y	
	INCRBYFLOAT	ValueOperations.increment	Y	
	MGET	ValueOperations.multiGet	Y	
	MSET	ValueOperations.multiSet	Y	
	MSETNX	ValueOperations.multiSetIfAbsent	Y	
	PSETEX	ValueOperations.set	Y	
	SET	ValueOperations.set	Y	
	SETBIT	ValueOperations.setBit	Y	
	SETEX	ValueOperations.set	Y	
	SETNX	ValueOperations.setIfAbsent	Y	
	SETRANGE	ValueOperations.set	Y	
	STRLEN	ValueOperations.size	Y	

Hash-type operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks
Hash	HDEL	HashOperations.delete	Y	
	HEXISTS	HashOperations.hasKey	Y	
	HGET	HashOperations.get	Y	
	HGETALL	HashOperations.entries	Y	
	HINCRBY	HashOperations.increment	Y	
	HINCRBYFLOAT	HashOperations.increment	Y	
	HKEYS	HashOperations.keys	Y	
	HLEN	HashOperations.size	Y	
	HMGET	HashOperations.multiGet	Y	
	HMSET	HashOperations.putAll	Y	
	HSET	HashOperations.put	Y	
	HSETNX	HashOperations.putIfAbsent	Y	
	HVALS	HashOperations.values	Y	
	HSCAN	HashOperations.san	Y	
	HSTRLEN	N		

List-type operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks
List	BLPOP	ListOperations.leftPop	Y	

	BRPOP	ListOperations.rightPop	Y	
	BRPOPLPUSH	ListOperations.rightPopAndLeftPush	Y	key: sourceKey:${sourceKey};destKey:${destKey}
	LINDEX	ListOperations.index	Y	
	LINSERT	ListOperations.leftPush	Y	
	LLEN	ListOperations.size	Y	
	LPOP	ListOperations.leftPop	Y	
	LPUSH	ListOperations.leftPush	Y	
	LPUSHX	ListOperations.leftPushIfPresent	Y	
	LRANGE	ListOperations.range	Y	
	LREM	ListOperations.remove	Y	
	LSET	ListOperations.set	Y	
	LTRIM	ListOperations.trim	Y	
	RPOP	ListOperations.rightPop	Y	
	RPOPLPUSH	ListOperations.rightPopAndLeftPush	Y	key: sourceKey:${sourceKey};destKey:${destKey}
	RPUSH	ListOperations.rightPush	Y	
	RPUSHX	ListOperations.rightPushIfPresent	Y	

Set-type operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks

Set	SADD	SetOpertions.add	Y	
	SCARD	SetOpertions.size	Y	
	SDIFF	SetOpertions.difference	Y	
	SDIFFSTORE	SetOpertions.differenceAndStore	Y	
	SINTER	SetOpertions.intersect	Y	
	SINTERSTORE	SetOpertions.intersectAndStore	Y	
	SISMEMBER	SetOpertions.isMember	Y	
	SMEMBERS	SetOpertions.members	Y	
	SMOVE	SetOpertions.move	Y	
	SPOP	SetOpertions.pop	Y	
	SRANDMEMBER	SetOpertions.randomMember randomMembers distinctRandomMembers	Y	
	SREM	SetOpertions.remove	Y	
	SUNION	SetOpertions.union	Y	
	SUNIONSTORE	SetOpertions.unionAndStore	Y	
	SSCAN	SetOpertions.scan	Y	

SortedSet-type operations

Data structure/Object	**Operation**	**Spring Data Redis method**	**EDAS support for EagleEye tracing (Y/N)**	**Remarks**
SortedSet	ZADD	ZSetOperations.add	Y	

	ZCARD	ZSetOperations.size/zCard	Y	
	ZCOUNT	ZSetOperations.count	Y	
	ZINCRBY	ZSetOperations.incrementScore	Y	
	ZRANGE	ZSetOperYations.range rangeWithScores	Y	
	ZRANGEBYSCORE	ZSetOperations.rangeByScore rangeByScoreWithScores	Y	
	ZRANK	ZSetOperations.rank	Y	
	ZREM	ZSetOperations.remove	Y	
	ZREMRANGEBYRANK	ZSetOperations.removeRange	Y	
	ZREMRANGEBYSCORE	ZSetOperations.removeRangeByScore	Y	
	ZREVRANGE	ZSetOperations.reverseRange reverseRangeWithScores	Y	
	ZREVRANGEBYSCORE	ZSetOperations.reverseRangeByScore reverseRangeByScoreWithScores	Y	
	ZREVRANK	ZSetOperations.reverseRank	Y	
	ZSCORE	ZSetOperations.score	Y	
	ZUNIONSTORE	ZSetOperations.unionAndStore	Y	
	ZINTERSTORE	ZSetOperations.intersectAndStore	Y	
	ZSCAN	ZSetOperations.scan	Y	
	ZRANGEBYLEX	ZSetOperations	Y	

		.rangeByLex		
	ZLEXCOUNT	N		
	ZREMRANGEBYLEX	N		

HyperLogLog operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks
HyperLogLog	PFADD	HyperLogLogOperations.add	Y	
	PFCOUNT	HyperLogLogOperations.size	Y	
	PFMERGE	HyperLogLogOperations.union	Y	key: dest:${destination}

Pub/Sub (publish/subscribe) operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks
Pub/Sub	PSUBSCRIBE	N		
	PUBLISH	RedisOperations.convertAndSend	Y	key: msg:${msg}
	PUBSUB	RedisMessageListenerContainer .setMessageListeners .addMessageListener	**N**	
	PUNSUBSCRIBE	N		
	UNSUBSCRIBE	N		

Transaction operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks
Transaction	DISCARD	RedisOperations.discard	Y	

	EXEC	RedisOperations.exec	Y	key: execRaw
	MULTI	RedisOperations.multi	Y	
	UNWATCH	RedisOperations.unwatch	Y	
	WATCH	RedisOperations.watch	Y	

Script operations

Data structure/Object	Operation	Spring Data Redis method	EDAS support for EagleEye tracing (Y/N)	Remarks
Script	EVAL	ScriptExecutor.execute	Y	key: Null
	EVALSHA	ScriptExecutor.execute	Y	key: Null
	SCRIPT EXISTS	RedisScriptingCommands.scriptExists	N	
	SCRIPT FLUSH	RedisScriptingCommands.scriptFlush	N	
	SCRIPT KILL	RedisScriptingCommands.scriptKill	N	
	SCRIPT LOAD	RedisScriptingCommands.scriptLoad	N	