企业级分布式应用服务 EDAS

开发指南

开发指南

开发工具准备

安装 Ali-Tomcat 和 Pandora

Ali-Tomcat 和 Pandora 为 EDAS 中的服务运行时所依赖的容器，主要集成了服务的发布、订阅、调用链追踪等一系列的核心功能，无论是开发环境还是运行时，均必须将应用程序发布在该容器中。

Ali-Tomcat 和 Pandora 的安装步骤如下：

注意：请使用 JDK 1.7及以上版本。

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

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

查看 Pandora 容器的目录结构。

Linux 系统中，在相应路径下执行 **tree** -L 2 deploy/ 命令查看目录结构。

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

Windows 中，直接进入相应路径进行查看。

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

配置 Eclipse 开发环境

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

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

压缩包内容如下：

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

在 Install 对话框中 Work with 区域右侧单击 **Add**，然后在弹出的 Add Repository 对话框中单击 **Local**。在弹出的对话框中选中已下载并解压的 Tomcat4E 插件的目录（d:\work\tomcat4e\）>，单击 **OK**。

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

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

重启 Eclipse。

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

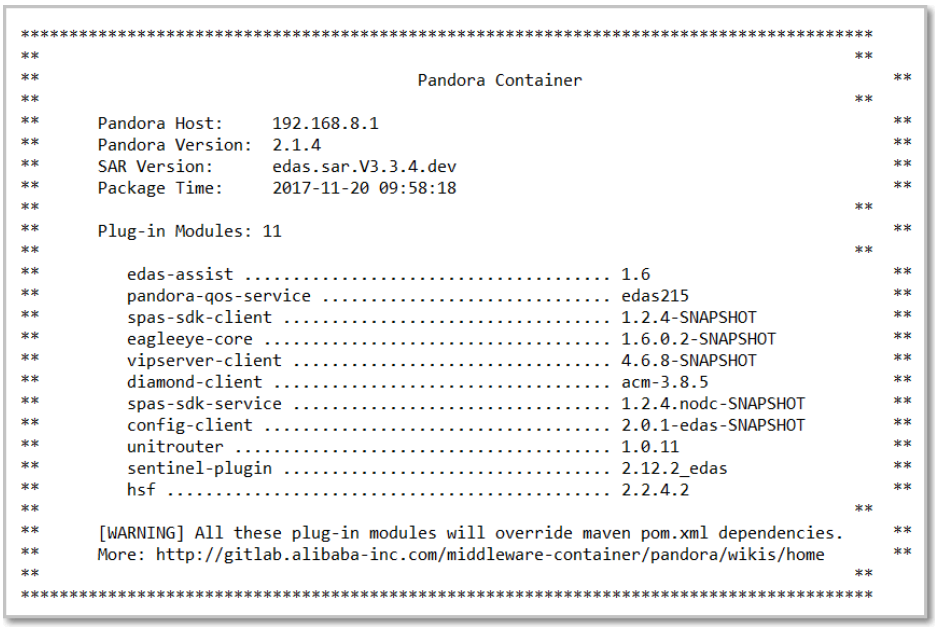
选择左侧导航选项中的 **AliTomcat Webapp**，单击上方的 **New launch configuration** 图标。

在弹出的界面中，选择 **AliTomcat** 页签，在 **taobao-hsf.sar Location** 区域单击 **Browse**，选择本地的 Pandora 路径，如：d:\work\tomcat\deploy\taobao-hsf.sar。

单击 **Apply** 或 **Run**，完成设置。

一个工程只需配置一次，下次可直接启动。

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



配置 IDEA 开发环境

前提条件

- 在配置 IDEA 开发环境前，请确保相关插件（Ali-Tomcat、Pandora）已下载并解压完成。否则，请参考下载 Ali-Tomcat 和 Pandora 容器，进行下载及解压。
- 目前仅支持 IDEA 商业版，社区版暂不支持。所以，请确保本地安装了商业版 IDEA。

配置步骤

运行 IntelliJ IDEA。

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

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

配置 AliTomcat。

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

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

将 Tomcat Home 的路径设置为本地解压后的 Ali-Tomcat 路径，Tomcat base directory 可以自动使用该路径，无需再设置。

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

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

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

单击 **Apply** 或 **OK** 完成配置。

配置轻量配置中心

轻量配置中心给开发者提供在开发、调试、测试的过程中的服务发现、注册和查询功能。此模块不属于 EDAS 正式环境中的服务，使用时请下载安装包进行安装。

在一个公司内部，通常只需要在一台机器上安装轻量配置中心服务，并在其他开发机器上绑定特定的 host 即可。具体安装和使用的步骤请参见下文。

1. 下载轻量配置中心

确认环境是否达到要求。

正确配置环境变量 JAVA_HOME，指向一个 1.6 或 1.6 以上版本的 JDK。

确认 8080 和 9600 端口未被使用。

由于启动 EDAS 配置中心将会占用此台机器的 8080 和 9600 端口，因此推荐您找一台**专门的**机器启动 EDAS 配置中心，比如某台测试机器。如果您是在同一台机器上进行测试，请将 Web 项目的端口修改为其它未被占用的端口。

下载 EDAS 配置中心安装包并解压。

如有需要，可以下载历史版本：

- 2018年01月版本
- 2017年08月版本
- 2017年07月版本
- 2017年03月版本
- 2016年12月版本

2. 启动轻量配置中心

进入解压目录（edas-config-center），启动配置中心。

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

3. 配置 hosts

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

hosts 文件的路径如下：

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

Unix 操作系统：/etc/hosts

示例

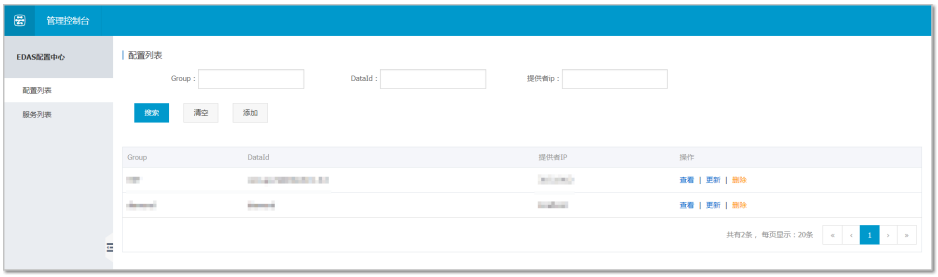
如果您在 IP 为 192.168.1.100 的机器上面启动了 EDAS 配置中心，则所有开发者只需要在机器的 hosts 文件里加入如下一行即可。

192.168.1.100 jmenv.tbsite.net

结果验证

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

即可看到轻量配置中心首页：



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

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

服务开发

下载 Demo 工程

本章中介绍的代码均可以通过官方 Demo 获取。

下载 Demo 工程。

将下载下来的压缩包解压，可以看到 carshop 文件夹，里面包含 itemcenter-api，itemcenter 和 detail 三个 Maven 工程文件夹。

- itemcenter-api：提供接口定义
- itemcenter：生产者应用
- detail：消费者应用

注意：请使用 JDK 1.7 及以上版本。

定义服务接口

HSF 的服务基于接口实现，当接口定义好之后，生产者将通过该接口以实现具体的服务，消费者也是基于此接口作为服务去订阅。

在 Demo 的 itemcenter-api 工程中，定义了一个服务接口 com.alibaba.edas.carshop.itemcenter.ItemService，内容如下：

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

此接口将提供两个方法，getItemById 与 getItemByName，也可以理解成为该服务 com.alibaba.edas.carshop.itemcenter.ItemService 将提供两个方法。

开发生产者服务

生产者将实现服务接口以提供具体服务。同时，由于使用了 Spring 框架，还需要在 .xml 文件中配置服务属性。

说明：Demo 工程中的 itemcenter 文件夹为生产者服务的示例代码。

实现服务接口

可以参考 ItemServiceImpl.java 文件中的示例：

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

```
public class ItemServiceImpl implements ItemService {

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

配置服务属性

上述例子主要实现了 com.alibaba.edas.carshop.itemcenter.ItemService，并在两个方法中返回了一个 Item 对象。代码开发完成之后，除了在web.xml 中进行必要的常规配置，您还需要增加相应的 Maven 依赖，同时在 Spring 配置文件使用 <hsf /> 标签注册并发布该服务。具体内容如下：

在 pom.xml 中添加如下 Maven 依赖的内容：

```
<dependencies>
<!-- 添加 servlet 的依赖 -->
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>servlet-api</artifactId>
<version>2.5</version>
<scope>provided</scope>
</dependency>
<!-- 添加 Spring 的依赖 -->
<dependency>
<groupId>com.alibaba.edas.carshop</groupId>
<artifactId>itemcenter-api</artifactId>
<version>1.0.0-SNAPSHOT</version>
</dependency>
<!-- 添加服务接口的依赖 -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>2.5.6(及其以上版本)</version>
</dependency>
<!-- 添加 edas-sdk 的依赖 -->
<dependency>
<groupId>com.alibaba.edas</groupId>
<artifactId>edas-sdk</artifactId>
<version>1.5.0</version>
</dependency>
```

```
</dependencies>
```

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

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

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

生产者服务属性列表

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

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

生产者服务属性配置示例

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

发布服务

完成代码、接口开发和服务配置后，在 Eclipse 或 IDEA 中，可直接以 Ali-Tomcat 运行该服务（具体请参照文档配置 Eclipse 开发环境和配置 IDEA 开发环境。

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

属性	描述
-Dhsf.server.port	指定 HSF 的启动服务绑定端口，默认值为 12200。
-Dhsf.serializer	指定 HSF 的序列化方式，默认值为 hessian。
-Dhsf.server.max.poolsize	指定 HSF 的服务端最大线程池大小，默认值为 600。
-Dhsf.server.min.poolsize	指定 HSF 的服务端最小线程池大小。默认值为 50。
-DHSF_SERVER_PUB_HOST	指定对外暴露的 IP，如果不配置，使用 -Dhsf.server.ip 的值。
-DHSF_SERVER_PUB_PORT	指定对外暴露的端口，该端口必须在本机被监听，并对外开放了访问授权，默认使用 -Dhsf.server.port 的配置，如果 -Dhsf.server.port 没有配置，默认使用12200。

运行成功后，可在轻量配置中心查询到所发布的服务，具体请参考服务查询。

开发消费者服务

消费者订阅服务从代码编写的角度分为两个部分：首先 Spring 的配置文件使用标签<hsf:consumer/>定义好一个 Bean；然后在使用的时候从 Spring 的 context 中将 Bean 取出来即可。

说明：Demo 工程中的 detail 文件夹为消费者服务的示例代码。

配置服务属性

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

在 pom.xml 文件中添加 Maven 依赖。

Maven 依赖配置与生产者相同，详情请参见**生产者实现服务的配置服务属性**内容。

在 hsf-consumer-beans.xml 文件中添加 Spring 关于 HSF 服务的配置。

增加消费者的定义，HSF 框架将根据该配置文件去服务中心订阅所需的服务。

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

配置服务调用

可以参考 StartListener.java 文件中的示例：

```
public class StartListener implements ServletContextListener{

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

```
// 根据 Spring 配置中的 Bean ID “item” 获取订阅到的服务
final ItemService itemService = ( ItemService ) ctx.getBean( "item" );
……
// 调用服务 ItemService 的 getItemById 方法
System.out.println( itemService.getItemById( 1111 ) );
// 调用服务 ItemService 的 getItemByName 方法
System.out.println( itemService.getItemByName( "myname is le" ) );
……
}
}
```

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

消费者服务属性列表

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

消费者服务属性配置示例

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

发布服务

完成代码、接口开发和服务配置后，在 Eclipse 或 IDEA 中，可直接以 Ali-Tomcat 运行该服务（具体请参照文档配置 Eclipse 开发环境和配置 IDEA 开发环境。

运行成功后，可在轻量配置中心查询到所发布的服务，具体请参考服务查询。

查询服务

目前 EDAS 支持 Dubbo 和 HSF 的服务注册，此文档仅说明 HSF 的服务查询方式。如果您的 Dubbo 服务还是发布到原有的注册中心（如：ZooKeeper），目前在 EDAS 控制台无法进行查询。

开发环境查询 HSF 服务

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

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

进入 http://192.168.1.100:8080/ 。

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

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

常见查询案例

提供者列表页

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

在搜索条件里输入服务名或服务分组，即可查询哪些 IP 地址提供了这个服务。

调用者列表页

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

在搜索条件里输入服务名或服务分组，即可查询哪些 IP 地址调用了这个服务。

线上环境查询 HSF 服务

开发好的服务打包并部署到 EDAS 之后，确认应用可以正常启动的情况下，您可在 EDAS 控制台查询相应的服务列表信息。具体步骤如下：

登录 EDAS 控制台，在左侧菜单栏单击**应用管理** 。

在应用列表页，单击部署的应用进入应用详情页。

在左侧菜单栏选择**服务列表**选项，可看到**发布的服务**与**消费的服务**两个 TAB 标签。**发布的服务**即应用配置的 Provider， **消费的服务**为应用配置的 Consumer。

注意：如您是通过子账号登录，请确认有无查看**服务列表**页的权限。您可以在控制台左侧菜单栏选择**账号管理** > **所有权限**，在权限管理页查看**应用管理**下有无服务列表的查看权限。

如果在相应的服务列表中没有相应的服务，可按照以下步骤排除可能存在的问题：

- 请确认服务配置在代码中是否配置正确。
- 请确认服务的 Tomcat 进程正常启动，且在 log 中没有报错 (查看 TOMCAT_HOME/logs/catalina.out 和 $TOMCAT_HOME/logs/localhost.log.$DATE_FORMAT)。
- 请确认是否是最新的软件版本(具体在对应服务信息界面左侧菜单栏的**软件版本**处查看), 如果不是最新版本，需确认对应的 HSF 分组是否创建。
- 请确认对应机器的 Host 是否有特殊的网络绑定。正常情况下线上机器无须绑定任何 Host。
- 请确认机器的网络与 ECS 安全组配置是否有明显的限制。

HSF 特性使用

本文介绍 HSF 一些特性的使用方法及注意事项。

所有特性的实例 Demo 请在这里下载：Demo 下载

前提条件

使用 HSF 相关特性，请在 pom.xml 文件中加入以下 edas-sdk 依赖。

```
<dependency>
<groupId>com.alibaba.edas</groupId>
<artifactId>edas-sdk</artifactId>
<version>1.8.1</version>
</dependency>
```

隐式传参（目前仅支持字符串传输）

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

单个参数传递

服务消费者：

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

服务提供者：

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

多个参数传递

服务消费者：

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

服务提供者：

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

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

异步调用

支持 callback 和 future 两种异步调用方式。

callback 调用方式

客户端配置为 callback 方式时，需要配置一个实现了 HSFResponseCallback 接口的 listener。结果返回之后， HSF 会调用 HSFResponseCallback 中的方法。

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

XML 中的配置：

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

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

代码示例

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

注意：

- 由于只用方法名字来标识方法，所以并不区分重载的方法。同名的方法都会被设置为同样的调用方式。
- 不支持在 call 里再发起 HSF 调用。这种做法可能导致 IO 线程挂起，无法恢复。

future 调用方式

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

XML 中的配置：

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

代码示例如下。

单个调用异步处理：

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

多个调用需要并发处理：

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

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

方法内进行并行调用：

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

// 处理业务
...

//获得数据并做处理
for (HSFFuture hsfFuture : futures) {
```

```
String msg=(String) hsfFuture.getResponse(3000);
//处理相应数据
...
}
```

泛化调用

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

1. 在消费者 XML 配置中加入泛化属性

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

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

DemoApi 接口方法：

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

2. 获取 demoApi 进行强制转换为泛化服务

导入泛化服务接口

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

获取泛化对象

- XML 加载方式

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

代码订阅方式

```
HSFApiConsumerBean consumerBean = new HSFApiConsumerBean();
consumerBean.setInterfaceName("com.alibaba.demo.api.DemoApi");
consumerBean.setGeneric("true"); // 设置 generic 为 true
```

```
consumerBean.setVersion("1.0.0");
consumerBean.init();
// 强制转换接口为 GenericService
GenericService svc = (GenericService) consumerBean.getObject();
```

3. 泛化接口

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

接口参数说明：

methodName：需要调用的方法名称。

parameterTypes：需要调用方法参数的类型。

args：需要传输的参数值。

4. 泛化调用

String 类型参数

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

对象参数，服务端和客户端需要保证相同的对象

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

HSF 单元测试

在测试环境中，有两种方式做单元测试。

Demo 下载

方式一 、通过 LightApi 代码发布和订阅服务

在 Maven 中添加 LightApi 依赖。

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

创建 ServiceFactory。

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

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

通过代码进行发布和订阅服务。

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

// 进行服务消费
factory.consumer("helloConsumer")// 参数是一个标识，初始化后，下次只需调用 consumer("helloConsumer")即可直接提供对应服务
.service("com.alibaba.edas.unit.service.UnitTestService")// 接口全类名
.version("1.0.0")// 版本号
.subscribe();
factory.consumer("helloConsumer").sync();// 同步等待地址推送，最多6秒。
UnitTestService log4jService = (UnitTestService) factory.consumer("helloConsumer").subscribe();// 用 ID 获取对应服务，subscribe()方法返回对应的接口
// 调用服务方法
System.out.println("bean -> msg rec success:-"+log4jService.print());
```

方式二、通过 XML 配置发布订阅服务。

编写好 HSF 的 XML 配置。

通过代码方式加载配置文件。

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

HSF 调用链路 Filter 扩展

Demo下载

添加 SDK 依赖

```
<dependency>
<groupId>com.alibaba.edas</groupId>
<artifactId>edas-sdk</artifactId>
<version>1.7.0</version>
</dependency>
```

基础接口

```
public interface ServerFilter extends RPCFilter {
}

public interface ClientFilter extends RPCFilter {
}

public interface RPCFilter {
```

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

void onResponse(Invocation invocation, RPCResult rpcResult);

}
```

实现步骤

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

实现示例

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


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

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

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


return invocationHandler.invoke(invocation);
}

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

```
System.out.println(PojoUtils.generalize(resp));
}


}
```

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

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

运行效果

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

intercept response

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

可选的 Filter

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

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

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

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

```
    <bean class="com.taobao.hsf.app.spring.util.HSFSpringProviderBean">
<property name="serviceInterface" value="com.alibaba.middleware.hsf.guide.api.service.OrderService" />
```

```
<property name="version" value="1.0.0" />
<property name="group" value="HSF" />
<property name="includeFilters">
<list>
<value>HSFOptionalServerFilter</value>
<value>NoFilter</value>
</list>
</property>
<property name="target" ref="orderService" />
</bean>
```

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

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

服务发布

服务打包

以当前 Demo 为例，打包流程如下：

使用命令提示符(Win 环境)或 SHELL 终端（ *nix 环境 ）进入到示例工程目录。

打包 itemcenter-api：cd itemcenter-api && mvn clean install && cd ../

打包 detail 工程：cd detail && mvn clean package && cd ../

打包 itemcenter 工程：cd itemcenter && mvn clean package && cd ../

服务上线

完成了生产者和消费者开发并通过测试之后，先进行服务打包，然后需要将服务发布到线上。下面以当前 Demo 为例，简述线上发布的流程。

注意： EDAS RPC 服务需要使用12200端口，因此请确保服务所在机器该端口可以被服务消费者访问。

发布服务

由于有两个应用（detail.war 和 itemcenter.war）要发布，所以至少需准备两台机器，且两台机器上均需要安装 EDAS Agent，假设两台机器的名字分别为 (edas-detail 和 edas-itemcenter)，部署应用的步骤为：

登录 EDAS 控制台，在左侧导航栏中选择**应用管理**，在页面右上角单击**创建应用**。在弹出的界面中选择应用所在区域并填入正确的应用名后，单击**下一步**。

在机器列表中选择相应的机器；在这里我们创建两个应用， app-detail 和 app-itemcenter，对应部署的机器分别为 edas-detail 和 edas-itemcenter。

创建好应用后，进入**应用管理**，分别进入应用 app-detail 和 app-itemcenter。单击右上角的**部署应用**，在弹出的对话框中，选择在**服务打包**中打好的 WAR 包上传并部署。

部署完毕后，单击页面右上角的**启动应用**。

应用在机器上启动完毕之后，在应用基本信息页面的**实例信息**区域，可以看到对应的机器状态，当机器的实时状态为**正常**且任务状态为**运行中**时，说明应用在机器上已经启动成功。

查看发布的服务

1. 两个应用均启动成功后，在**应用管理**界面选择应用 app-temcenter，进入应用。

   在应用界面的左侧菜单栏中，选择**服务列表**，然后再选择**发布的服务**选项卡，可以看到 Spring 配置文件中所定义的发布的服务：com.alibaba.edas.carshop.itemcenter.ItemService

   同上，选择 app-detail 进入应用，可以看到 Spring 配置文件中所定义的消费的服务：com.alibaba.edas.carshop.itemcenter.ItemService

   进入机器 edas-detail，用 admin 身份进入到 AliTomcat 的 logs 目录下（假设路径为：/home/admin/taobao-tomcat/logs/）时，可以在 catalina.out 的输出中看到有如下的 log 信息：Item[id: 1, nam: Mercedes Benz] ，这正是从服务提供者 (app-itemcenter) 处返回的 Item 对象。

Dubbo 开发

概述

Dubbo 是阿里巴巴集团开源的一个分布式 RPC 框架，具体的介绍和开发指南请参阅官方文档。本节主要介绍 Dubbo 在 EDAS 中的使用流程及相关说明。

主要包括以下内容：

JAR 转换 WAR。

配置 Dubbo。

（可选）多注册中心兼容

使用 Dubbo 发布应用。

检查和 HSF 的兼容性。

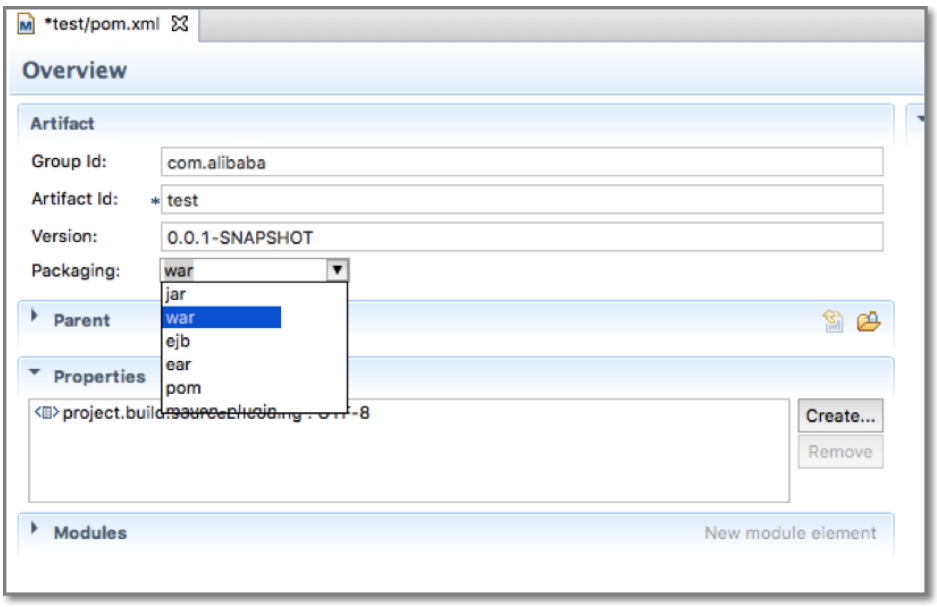
开始在 EDAS 中开发 Dubbo 服务之前，请确保您已经了解并掌握了 Dubbo 项目的开发，且对 Dubbo 相关的参数和属性有一定的了解。

JAR 转换 WAR

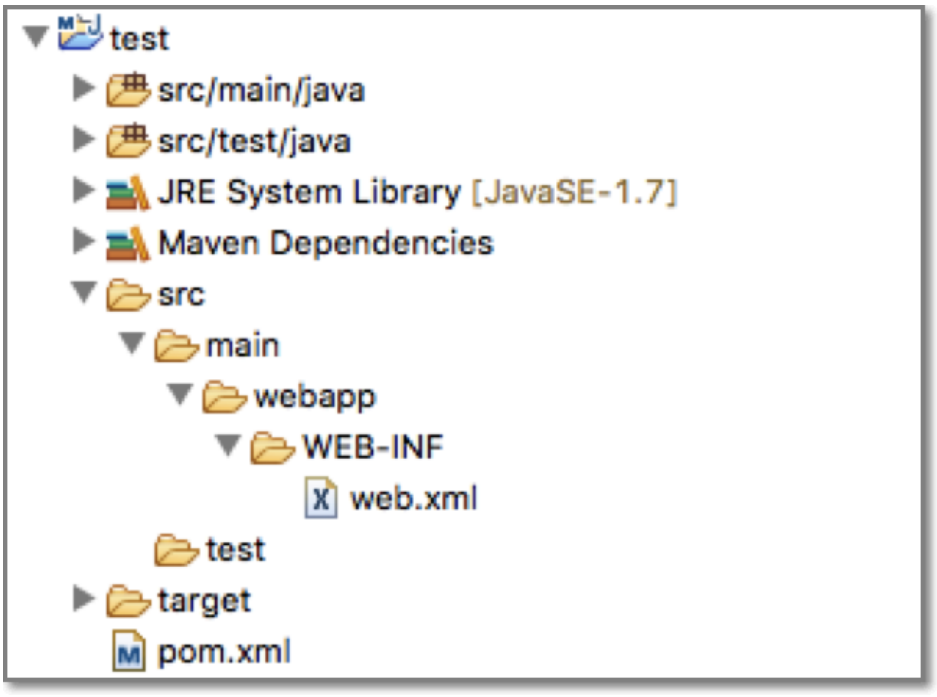
目前 EDAS 产品只支持 WAR 形式的 Web 项目，所以如果你的项目是 JAR 方式发布的，需要先进行转换。本文主要基于 Maven 项目来做示例。

步骤如下：

修改 POM 文件 JAR 为 WAR。



如果没有 web.xml，则需要增加一个 web.xml 文件配置。



配置 web.xml 加载配置文件。

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

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

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

配置 Dubbo

目前 Dubbo 在 EDAS 中运行支持两种配置服务提供者和服务消费者的方式：XML 配置、注解配置。本文档提供这两种方式的配置示例。

XML 文件配置方式

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

服务生产者 XML 配置

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

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

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

</beans>
```

注意：

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

supportted” 的错误发生。

服务消费者 XML 配置

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

注意：

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

注解配置方式

从 EDAS 容器 V3.0 版本开始，已经支持 Dubbo 原生注解，您无需进行注解转换 XML 即可使用 EDAS 服务。

兼容说明：

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

支持属性： group、version、timeout

使用方式： 在创建容器的时候，选择最新版本容器 V3.0 即可。

多注册中心兼容

多注册指 Dubbo/HSF 应用可以同时注册服务到 EDAS、ZooKeeper 注册中心，为其他消费者提供服务。

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

使用方式

在当前应用中加入不低于1.5.1的 edas-sdk 依赖。

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

指定 ZooKeeper 注册/订阅中心地址。

指定方式主要包含以下两种：

环境变量指定（支持 HSF 、Dubbo 应用）：

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

XML 指定方式（只支持 HSF 应用）：

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

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

设置 HSF 应用多注册中心的调用参数。

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

创建应用时，需要选择不低于3.0版本的容器，然后上传启动即可。

注意：

- **分组问题** Dubbo 服务分组默认为空，EDAS 里服务分组默认是 HSF。因此务必修改 Dubbo 服务分

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

使用 Dubbo 发布 Web 项目

使用 Dubbo 发布 Web 项目有两种方式。

通过右键直接启动 Tomcat4E 插件来启动 Web 项目。

这种方式常用于测试环境中，直接在 IDE 中运行项目，比较简单，无需过多配置，如果是多个项目只需要保证 Tomcat 的端口不要重复即可。Tomcat4E 插件配置请参考文档 Ali-Tomcat 安装介绍。

通过 EDAS 控制台来发布 Web 项目（WAR 包）。

检查和 HSF 的兼容性

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

功能特性	Dubbo 配置参数	兼容情况说明	错误提示	EDAS 是否支持
超时	timeout			支持
延迟暴露	delay			支持
线程模型	dispatcher= "all" threadpool= "fixed" threads= "100"			支持
回声测试				支持
延迟	lazy= "true"	默认开启		支持
连接				
本地调用	protocol= "injvm"			支持

隐式传参				支持
并发控制	actives=“10” executes=“10”	已经实现 EDAS 控制台可视化配置，无需配置		支持
连接控制	accepts=“10” connections="2"	已经实现 EDAS 控制台可视化配置，无需配置		支持
服务降级		已经实现 EDAS 控制台可视化配置，无需配置		支持
集群容错	retries/cluster	支持 retries	无报错	部分支持
负载均衡	loadbalance	默认 random	无报错	部分支持
服务分组	group	不支持 * 配置	java.lang.IllegalStateException: hsf2 不支持同时消费多个分组!	部分支持
多版本	version	不支持 * 配置	[HSF-Consumer] 未找到需要调用的服务的目标地址	部分支持
异步调用	async=“true" return="false"	return 参数无效	无报错	部分支持
启动时检查	check	EDAS 默认是启动不检查	无报错	默认支持启动不检查
多协议		只支持 Dubbo 协议	com.alibaba.dubbo.config.ServiceConfig 服务 [com.alibaba.demo.api.DemoApi] 配置了 RMI 协议， HSF2 不支持	部分支持
路由规则		已经实现 EDAS 控制台可视化配置，无需配置		支持
配置规则		已经实现 EDAS 控制台可视化配置，无需配置		支持
多注册中心				不支持
分组聚合	group= "aaa,bbb" merger= "true"	报错	java.lang.IllegalStateException: hsf2 不支持同时消费多个分组!	不支持
上下文信息		报错	Caused by: java.lang.UnsupportedOperationException: not	不支持

			support getInvocation method in hsf2	

Spring Cloud 开发

名词解释

介绍在基于Spring Cloud进行应用开发过程中，可能遇到的名词及其含义，帮助您了解相关概念。

Pandora

Pandora 是阿里巴巴内部的一个轻量级隔离容器，Aliware 通过 Pandora 来实现类的隔离以及加载。Pandora 是使用 Spring Cloud for Aliware 的必选依赖。

VIPServer

VIPServer 是 EDAS 中 RESTful 开发方式中涉及的注册发现中心的服务端，对应的客户端为 VIPClient。

轻量级配置中心

一个可以在本地运行的 EDAS 轻量级配置中心，包含了服务发现和配置管理的功能。

FatJar

FatJar (也称作可执行 jars) 是包含编译后的类及代码运行所需依赖 JAR 的存档，可以使用 java -jar 命令运行该应用程序。

EagleEye

EagleEye 是阿里巴巴的分布式追踪系统，基于 Google 的分布式调用跟踪系统 Dapper 实现。EagleEye 日均处理万亿级别的分布式调用链数据，通过收集和分析在不同的网络调用中进行日志埋点，可以得到同一次请求上的各个系统的调用链关系，有助于梳理应用的请求入口与服务的调用来源、依赖关系，同时，也对分析系统调用瓶颈、估算链路容量、快速定位异常有很大帮助。

HSF

HSF 是阿里巴巴内部的一个分布式服务框架，HSF 从分布式应用层面以及统一的发布/调用方式层面为开发人员提供支持，从而可以很容易的开发分布式的应用以及提供或使用公用功能模块，而不用考虑分布式领域中的各种细节技术，例如远程通讯、性能损耗、调用的透明化、同步/异步调用方式的实现等等问题。

服务注册与发现

目前 EDAS 已经完全支持 Spring Cloud 应用了，您可以将 Spring Cloud 应用直接部署到 EDAS 中。

同时，为了更好地将阿里中间件的功能以云服务的方式提供给大家，EDAS 还对 Spring Cloud 中的一些组件进行了替换，增强了安全性和易用性。

本文档将详细介绍如何接入 EDAS 服务注册发现。

Demo 源码下载： sc-server-server 、 sc-vip-client。

准备工作

Maven 私服配置

目前 Spring Cloud for Aliware 的第三方包只发布在 EDAS 的私服中，所以需要配置私服地址，有关 Maven 私服的配置的说明，参见如何在 Maven 中配置 EDAS 的私服地址。

注意： Maven 版本要求 3.x 及以上，请在你的 Maven 配置文件 settings.xml 中，加入 EDAS 私服地址。样例文件下载。

轻量级配置中心

本地开发调试时，需要启动轻量级配置中心。轻量级配置中心包含了 EDAS 服务发现和配置管理功能的轻量版，详细文档及下载请参见轻量级配置中心。

实现服务注册和发现

通过一个简单的示例说明如何实践服务的注册和发现。

创建服务提供者

此服务提供者提供了一个简单的 echo 服务，并将自身注册到服务发现中心。

创建一个 Spring Cloud 工程，命名为 sc-vip-server。

在 pom.xml 中引入需要的依赖内容：

```
 <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>1.1</version>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-pandora</artifactId>
<version>1.2</version>
</dependency>
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>Dalston.SR4</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

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

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

添加服务提供端的代码，其中 @EnableDiscoveryClient 注解表明此应用需开启服务注册与发现功能。

```
@SpringBootApplication
@EnableDiscoveryClient
public class ServerApplication {

public static void main(String[] args) {
PandoraBootstrap.run(args);
```

```
SpringApplication.run(ServerApplication.class, args);
PandoraBootstrap.markStartupAndWait();
}
}
```

创建一个 EchoController，提供简单的 echo 服务。

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

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

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

创建服务消费者

这个例子中，我们将创建一个服务消费者，消费者通过 RestTemplate、AsyncRestTemplate、FeignClient 这三个客户端去调用服务提供者。

创建一个 Spring Cloud 工程，命名为 sc-vip-client。

在 pom.xml 中引入需要的依赖内容：

```
<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>1.1</version>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-pandora</artifactId>
<version>1.2</version>
</dependency>
```

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-feign</artifactId>
</dependency>
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>Dalston.SR4</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

因为在这里要演示 FeignClient 的使用，所以与 service-provider 相比，pom.xml 文件中的依赖增加了一个 spring-cloud-starter-feign。

与 Server 端相比，除了开启服务与注册外。还需要添加两项配置才能使用 RestTemplate、AsyncRestTemplate、FeignClient 这三个客户端：

- 添加 @LoadBalanced 注解将 RestTemplate 与 AsyncRestTemplate 与服务发现结合。

  使用 @EnableFeignClients 注解激活 FeignClients。

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

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

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

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

}
```

在使用 EchoService 的 FeignClient 之前，还需要完善它的配置。配置服务名以及方法对应的 HTTP 请求，服务名 为 sc-vip-server 工程中配置的服务名 service-provider ，代码如下：

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

创建一个 Controller 供我们调用测试。

- /echo-rest/* 验证通过 RestTemplate 去调用服务提供者。
- /echo-async-rest/* 验证通过 AsyncRestTemplate 去调用服务提供者。

/echo-feign/* 验证通过 FeignClient 去调用服务提供者。

```
@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);
}

}
```

配置应用名以及监听端口号。

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

本地开发调试

启动轻量级配置中心

本地开发调试时，需要使用轻量级配置中心，轻量级配置中心包含了 EDAS 服务注册发现服务端的轻量版，详细文档请参见轻量级配置中心。

启动应用

本地启动应用可以通过两种方式。

IDE 中启动

在 IDE 中启动，通过 VM options 配置启动参数 -Dvipserver.server.port=8080，通过 main 方法直接启动。

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

FatJar 启动

添加 FatJar 打包插件。

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

```
 <build>
<plugin>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.7.8</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
</build>
```

添加完插件后，在工程的主目录下，使用 maven 命令 mvn clean package 进行打包，即可在 target 目录下找到打包好的 FatJar 文件。

通过 Java 命令启动。

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

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

演示

启动服务，分别进行调用，可以看到调用都成功了。

EDAS 中部署

1. 在创建应用时，选择最新版本的容器。
2. 添加 FatJar 打包插件，在工程所在的目录下执行 mvn clean package 命令，打包成 FatJar。
3. 部署应用 ，部署时上传 target 目录下的 FatJar 应用即可将应用部署到 EDAS，无需关心配置。

从 Eureka 迁移

已经接入 Eureka 服务注册与发现的应用，只需要进行两步操作，就可以将服务接入 EDAS 服务注册发现中心。

修改源码。

只需要在 main 函数中添加两行，修改之前的 main 函数内容如下：

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

修改之后的 main 函数如下：

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

修改 pom.xml 依赖。

将 spring-cloud-starter-eureka 替换成 spring-cloud-starter-vipclient

替换前

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

替换后

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-vipclient</artifactId>
<version>1.1</version>
</dependency>
```

常见问题

AsyncRestTemplate无法接入服务发现。

AsyncRestTemplate 接入服务发现的时间比较晚，需要在 Dalston 之后的版本才能使用，具体详情参见此 pull request。

FatJar打包插件冲突

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

打包时可不可以不排除taobao-hsf.sar？

可以，但是不建议这么做。

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

```
<plugin>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.7.8</version>
<configuration>
<excludeSar>false</excludeSar>
</configuration>
```

```
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
```

这样打包后可以在不配置 Pandora 地址的情况下启动。

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

请在将应用部署到 EDAS 前恢复到默认排除 sar 包的配置。

全链路跟踪

目前 EDAS 已经完全支持 Spring Cloud 应用了，您可以将 Spring Cloud 应用直接部署到 EDAS 中。

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

本文档将详细介绍如何接入 EDAS 的全链路跟踪。

Demo 源码下载：service1、service2

接入 EagleEye

在 Maven 中配置 EDAS 的私服地址

目前 Spring Cloud for Aliware 的第三方包只发布在 EDAS 的私服中，所以需要配置私服地址，有关 Maven 私服的配置的说明，参见如何在 Maven 中配置 EDAS 的私服地址。

注意： Maven 版本要求 3.x 及以上，请在你的 Maven 配置文件 settings.xml 中，加入 EDAS 私服地址。样例文件下载。

修改代码

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

只需要在 pom.xml 文件中加入如下公共配置。

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-eagleeye</artifactId>
<version>1.1</version>
</dependency>

<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-pandora</artifactId>
<version>1.2</version>
</dependency>
```

在 main 函数中添加两行，假设修改之前的 main 函数内容如下：

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

修改之后的 main 函数如下：

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

添加 FatJar 打包插件。

使用 maven 将 pandora-boot 工程打包成 FatJar， 需要在 pom.xml 中添加如下插件。

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

```
<build>
<plugins>
<plugin>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.7.8</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
```

```
</plugin>
</plugins>
</build>
```

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

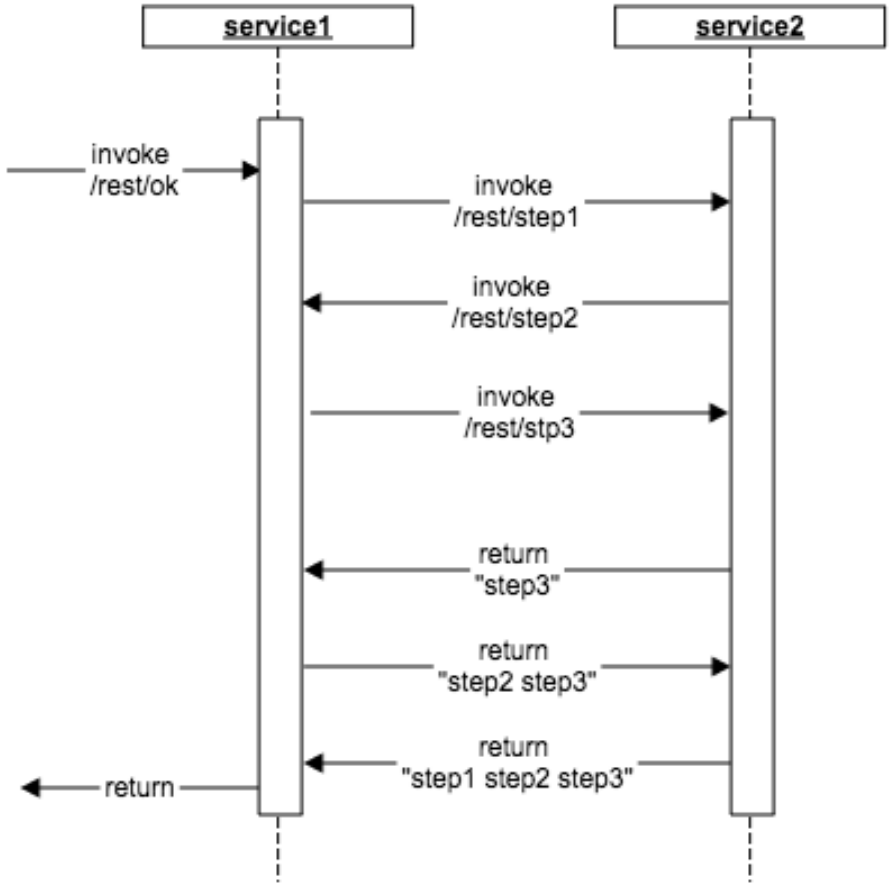
全链路跟踪示例

源码

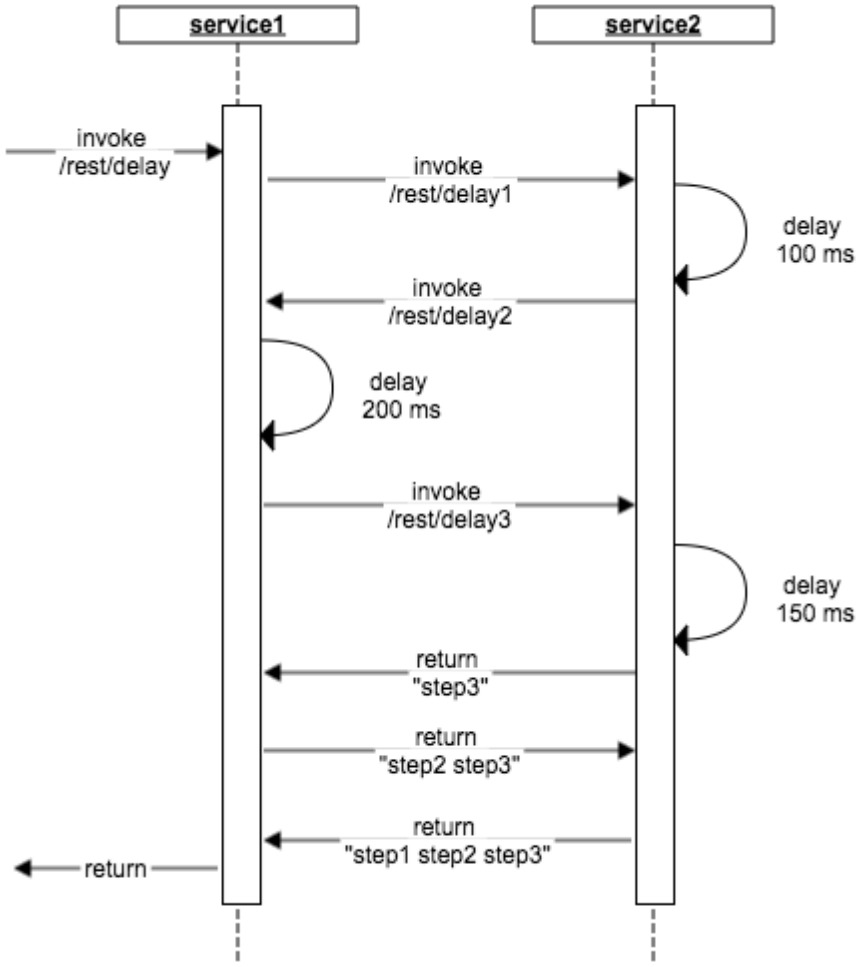
为了演示如何使用全链路跟踪功能 ，这里提供两个应用 Demo：service1 和 service2。

service1 用作入口的服务，提供了三个场景演示的入口：

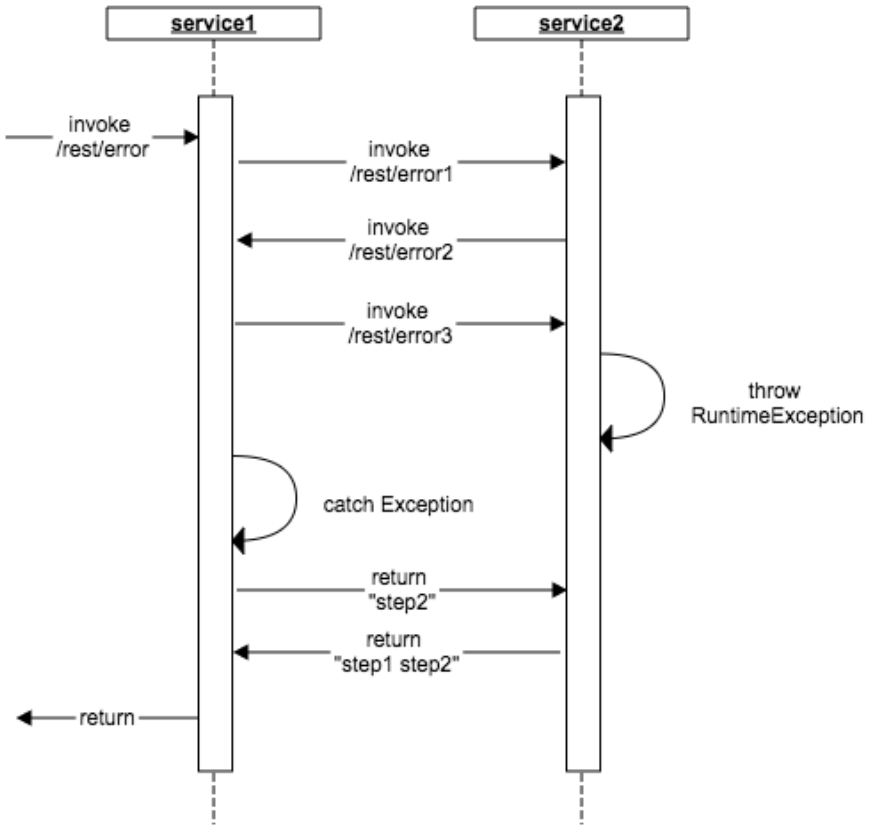
/rest/ok，对应正常的调用



/rest/delay，对应延迟较大的调用



/rest/error，对应异常出错的调用



部署应用

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

1. 在**创建应用**时，选择最新版本的容器。
2. 添加 FatJar 打包插件，在工程所在的目录下执行 mvn clean package 命令，打包成 FatJar。
3. **部署应用**，部署时上传 target 目录下的 FatJar 应用即可将应用部署到 EDAS。

部署完毕之后，为了能够查看调用链的信息，我们还需要调用 service1 三个场景演示的入口对应的方法。

您可以通过执行 curl http://{$ip:$port}/rest/ok 这种简单的方式来调用。也可以使用 postman 等工具或者直接在浏览器中调用。

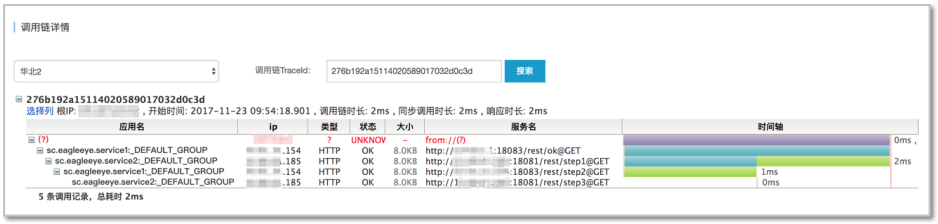
为了便于观察，建议使用脚本等方式多调用几次。

调用链查看

1. 登录 EDAS **控制台**，进入刚刚部署的应用中。
2. 在应用详情页面左侧的导航栏中单击**应用监控** > **服务监控**。
3. 在服务监控页面单击**提供的 RPC 服务**页签，然后单击**查看调用链**。

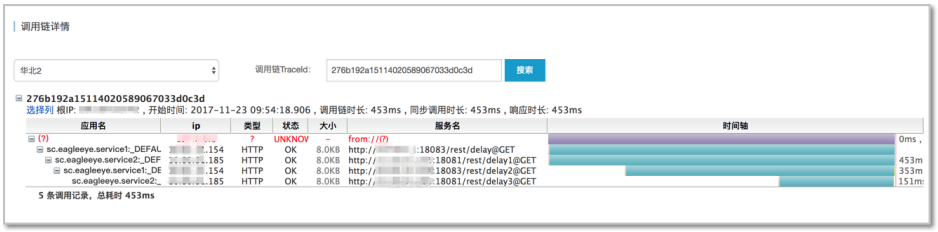
更详细的使用信息，请参考服务监控

正常的调用链详情



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

延迟较大的调用链详情



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

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

异常出错的调用链详情

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

其他客户端的演示

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

常见问题

埋点支持

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

AsyncRestTemplate

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

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

FatJar 打包插件

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

扩展

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

HSF 开发

目前 EDAS 已经完全支持 Spring Cloud 应用了，您可以将 Spring Cloud 应用直接部署到 EDAS 中。

- Spring Boot 提出的概念是 Build Anything，解决繁琐的 xml 配置的问题。
- Spring Cloud 提出的概念是 Coordinate Anything，通过提供大量的、方便组件接入的 spring-cloud-starter 的支持，简化了分布式微服务的开发。

EDAS 也实现了自己的 Spring Cloud Starter HSF，您同样可以通过 Spring Cloud 来开发 HSF 应用。

本文档将介绍如何使用 Spring Cloud 来开发 HSF 应用。

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

准备工作

Maven 私服配置

目前 Spring Cloud for Aliware 的第三方包只发布在 EDAS 的私服中，所以需要配置私服地址，有关 Maven 私服的配置的说明，参见如何在 Maven 中配置 EDAS 的私服地址。

注意： Maven 版本要求 3.x 及以上，请在你的 Maven 配置文件 settings.xml 中，加入 EDAS 私服地址。样例文件下载。

轻量级配置中心

本地开发调试时，需要启动轻量级配置中心。轻量级配置中心包含了 EDAS 服务发现和配置管理功能的轻量版，详细文档及下载请参见轻量级配置中心。

示例：一个简单的 HSF 应用

创建服务提供者

创建一个 Spring Cloud 工程，命名为 sc-hsf-provider。

在 pom.xml 中引入需要的依赖内容：

```
 <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>1.1</version>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-pandora</artifactId>
<version>1.2</version>
</dependency>
```

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

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>Dalston.SR4</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

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

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

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

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

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

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

接口 com.aliware.edas.EchoService 提供了一个 echo 方法，也可以理解成服务 com.aliware.edas.EchoService 将提供一个 echo 方法。

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

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

除了 接口名 serviceInterface 之外， HSF 还需要 serviceVersion(服务版本) 才能唯一确定一个服务，这里将注解 HSFProvider 里的 serviceVersion 属性设置为 “1.0.0” 。于是我们发布的服务就可以通过 接口名 com.aliware.edas.EchoService 和 服务版本 1.0.0 这两者结合来确定了。

HSFProvider 注解中的配置拥有最高的优先级，如果在 HSFProvider 注解中没有配置，服务发布时会优先在 resources/application.properties 文件中查找这些属性的全局配置。如果前两项都没有配置，则会使用 HSFProvider 注解中的默认值。

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

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

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

最佳实践: 建议统一将**服务版本**、**服务超时**都统一配置在 application.properties 中。

添加服务启动的 main 函数入口。

```
@SpringBootApplication
public class HSFProviderApplication {

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

创建服务消费者

这个例子中，我们将创建一个服务消费者，消费者通过 HSFProvider 所提供的 API 接口去调用服务提供者。

创建一个 Spring Cloud 工程，命名为 sc-hsf-consumer。

在 pom.xml 中引入需要的依赖内容：

HSFConsumer 和 HSFProvider 的 Maven 依赖是完全一样的。

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.5.8.RELEASE</version>
<relativePath/>
</parent>

<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-hsf</artifactId>
<version>1.1</version>
</dependency>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-pandora</artifactId>
<version>1.2</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>Dalston.SR4</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

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

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

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

```
@Configuration
public class HsfConfig {
```

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

最佳实践：在 Config 类里配置一次 @HSFConsumer ，然后在多处通过 @Autowired 注入使用。通常一个 HSF Consumer 需要在多个地方使用，但并不需要在每次使用的地方都用 @HSFConsumer 来标记。只需要写一个统一的 Config 类，然后在其它需要使用的地方，直接通过 @Autowired 注入即可。

为了便于测试，通过一个 SimpleController 来暴露一个 /hsf-echo/* 的 http 接口，/hsf-echo/* 接口内部实现调用了 HSF 服务提供者。

```
@RestController
public class SimpleController {
@Autowired
private EchoService echoService;

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

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

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

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

最佳实践: 建议统一将**服务版本**、**服务超时**都统一配置在 application.properties 中。

添加服务启动的 main 函数入口。

```
@SpringBootApplication
public class HSFConsumerApplication {

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

本地开发调试

启动轻量级配置中心

本地开发调试时，需要使用轻量级配置中心，轻量级配置中心包含了 EDAS 服务注册发现服务端的轻量版，详细文档请参见轻量级配置中心。

启动应用

本地启动应用可以通过两种方式。

在 IDE 中启动

通过 VM options 配置启动参数 -Djmenv.tbsite.net={$IP}，通过 main 方法直接启动。其中 {$IP} 为 启动轻量级配置中心服务的那台机器的地址。比如本机启动轻量级配置中心，则 {$IP} 为 127.0.0.1。

您也可以不配置 JVM 的参数，而是直接通过修改 hosts 文件将 jmenv.tbsite.net 绑定到启动轻量级配置中心服务的那台机器的 IP。详情见轻量级配置中心。

通过 FatJar 启动

添加 FatJar 打包插件。

使用 Maven 将 pandora-boot 工程打包成 FatJar， 需要在 pom.xml 中添加如下插件。

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

```
 <build>
<plugin>
<groupId>com.taobao.pandora</groupId>
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.7.8</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
</build>
```

添加完插件后，在工程的主目录下，使用 maven 命令 mvn clean package 进行打包，即可在 target 目录下找到打包好的 FatJar 文件。

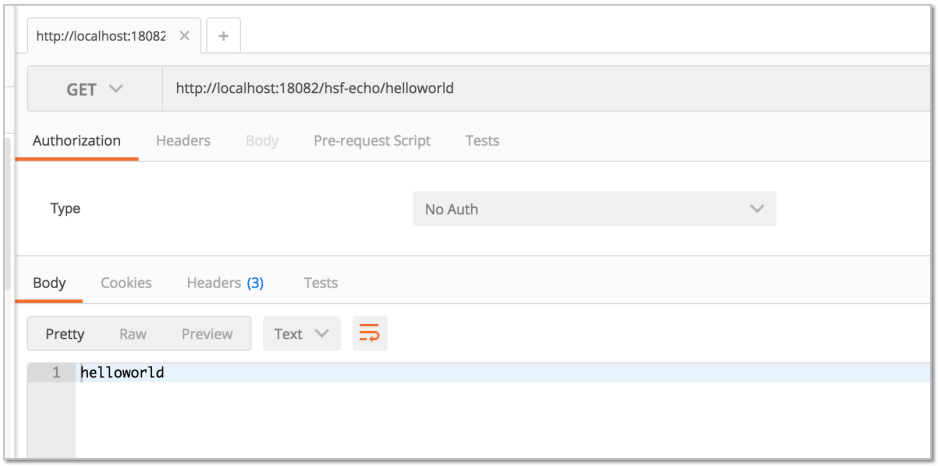
通过 Java 命令启动。

```
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
```

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

演示

启动服务，进行调用，可以看到调用成功。



EDAS 中部署

1. 在创建应用时，选择最新版本的容器。
2. 添加 FatJar 打包插件，在工程所在的目录下执行 mvn clean package 命令，打包成 FatJar。
3. 部署应用 ，部署时上传 target 目录下的 FatJar 应用即可将应用部署到 EDAS。

HSF 高级特性

在之前的文档中，HSF 开发已经介绍了如何使用 Spring Cloud 来开发 HSF 应用。

本文将介绍一下 HSF 的一些高级特性在 Spring Cloud 开发方式下的使用方式。目前内容包含 单元测试 和 异步调用 两部分，后续会有更多的介绍。

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

单元测试

spring-cloud-starter-hsf 的实现依赖于 Pandora Boot，Pandora Boot 的单元测试可以通过 PandoraBootRunner 启动，并与 SpringJUnit4ClassRunner 无缝集成。

我们将演示一下，如何在服务提供者中进行单元测试，供大家参考。

在 Maven 中添加 spring-boot-starter-test 的依赖。

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

编写测试类的代码。

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

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

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

异步调用

HSF 提供了两种类型的异步调用，Future 和 Callback。

在演示异步调用之前，我们先发布一个新的服务： com.aliware.edas.async.AsyncEchoService。

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

服务提供者实现 AsyncEchoService，并通过注解发布。

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

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

从这两点中可以看出，服务提供端与普通的发布没有任何区别，同样，之后的配置和应用启动流程也是一致的，详情请参考 HSF 开发中创建服务提供者部分的内容。

注意：异步调用的逻辑修改都在消费端，服务端无需做任何修改。

Future

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

这里我们将 com.aliware.edas.async.AsyncEchoService 的 Future 方法标记为 Future 类型的异步调用。

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

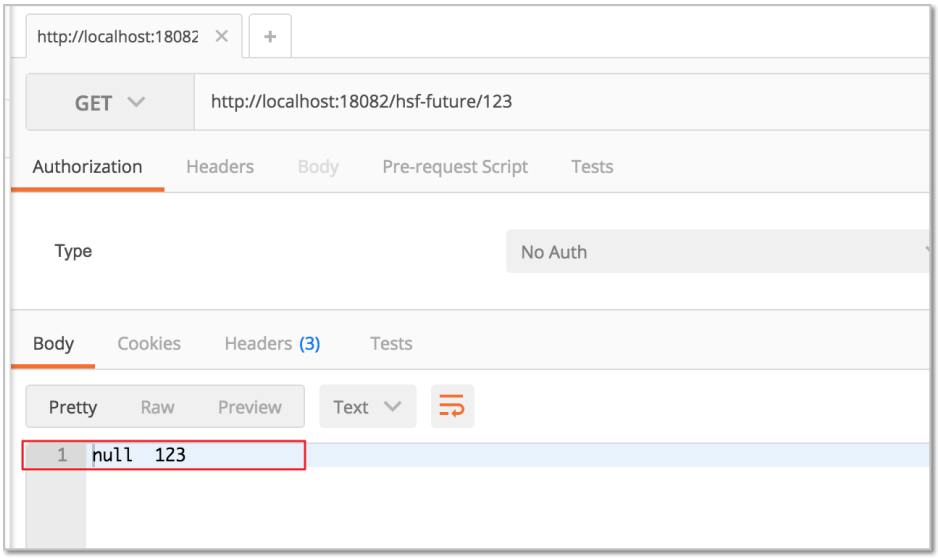
```
}
```

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

我们在这里通过 TestAsyncController 来进行演示，示例代码如下：

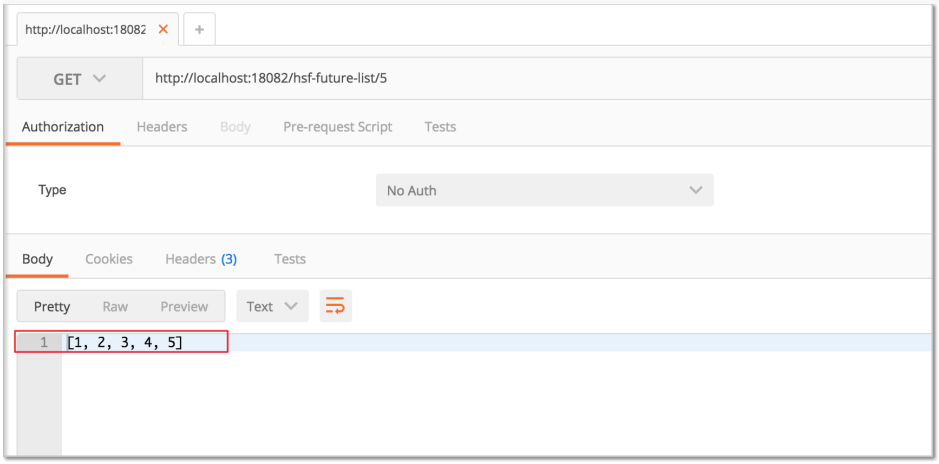
```
@RestController
public class TestAsyncController {

@Autowired
private AsyncEchoService asyncEchoService;

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

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

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



当服务中需要结合一批操作的返回值进行处理时，参考如下的调用方式。

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

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

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

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

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

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



Callback

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

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

```
t.printStackTrace();
}

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

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

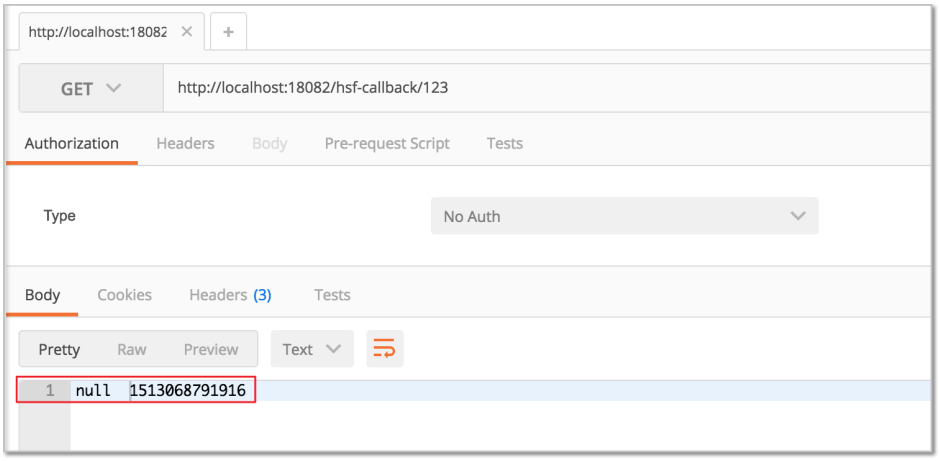
AsyncEchoResponseListener 实现了 HSFResponseCallback 接口，并在 @Async 注解中分别配置 interfaceName 为 AsyncEchoService.class、methodName 为 callback。

这样，就将 com.aliware.edas.async.AsyncEchoService 的 callback 方法标记为 Callback 类型的异步调用。

同样，通过 TestAsyncController 来进行演示，示例代码如下：

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

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

执行调用，可以看到如下结果：



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

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

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

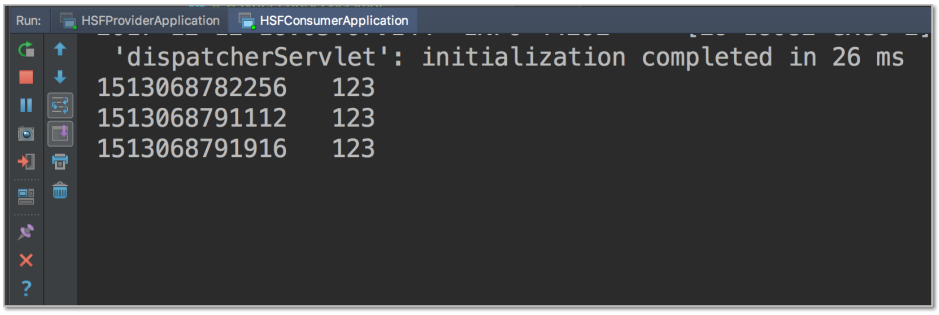
调用时的示例代码入下：

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

AsyncEchoResponseListener 示例代码如下：

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

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



Dubbo 开发

目前 EDAS 已经完全支持 Spring Cloud 应用了，您可以将 Spring Cloud 应用直接部署到 EDAS 中。

- Spring Boot 提出的概念是 Build Anything，解决繁琐的 xml 配置的问题。
- Spring Cloud 提出的概念是 Coordinate Anything，通过提供大量的、方便组件接入的 spring-cloud-starter 的支持，简化了分布式微服务的开发。

Dubbo 本身是开源的，官方目前也已经提供了 dubbo-spring-boot-starter。Dubbo 的代码及如何使用 Dubbo 进行应用开发请参考 GitHub。

您使用 Dubbo 开发的应用如果想使用全链路跟踪、服务分析、限流降级等高级功能，则需要将 Dubbo 升级成商业版。

本文档主要介绍如何通过简单的代码修改将 Spring Boot 编程模型的 Dubbo 升级成商业版，以及如何将本地开发的应用部署到 EDAS 中。应用开发过程不再详细描述。

升级成商业版的 Dubbo 的代码 Demo，下载地址：edas-dubbo-spring-boot-demo。

准备工作

Maven 私服配置

目前 Spring Cloud for Aliware 的第三方包（包括 Dubbo 商业版）只发布在 EDAS 的私服中，所以需要配置私服地址。Maven 私服的配置说明，请参见**如何在 Maven 中配置 EDAS 的私服地址**。

注意： Maven 版本要求 3.x 及以上，请在您的 Maven 配置文件 Settings.xml 中，加入 EDAS 私服地址。样例文件**下载**。

轻量级配置中心

Dubbo 升级到商业版后，原有的服务注册功能仍旧可以正常运行，同时还会将服务注册到 EDAS 的服务注册中心。

注意：在本地开发调试时，如果需要依赖 Dubbo 的商业版所提供的功能，则需要启动轻量级配置中心。

轻量级配置中心包含了 EDAS 服务发现和配置管理功能的轻量版，详细文档及下载请参见**轻量级配置中心**。

如何升级成商业版

增加 Maven 依赖

在项目的 pom.xml 中，增加 spring-cloud-starter-pandora 的依赖。

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-pandora</artifactId>
<version>1.2</version>
</dependency>
```

增加或修改 Maven 打包插件

在项目的 pom.xml 中，增加或修改 Maven 的打包插件。**为避免与其他打包插件发生冲突，请勿在 build 的 plugin 中添加其他 FatJar 插件。**

```
<build>
<plugins>
<plugin>
<groupId>com.taobao.pandora</groupId>
```

```
<artifactId>pandora-boot-maven-plugin</artifactId>
<version>2.1.7.8</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

修改代码

在 SpringBoot 的启动类中，增加两行加载 Pandora 的代码：

```
import com.taobao.pandora.boot.PandoraBootstrap;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ServerApplication {

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

将本地应用部署到 EDAS

1. 在**创建应用**时，选择最新版本的容器。
2. 在工程所在的目录下执行 mvn clean package 命令，打包成 FatJar。
3. **部署应用**，部署时上传 target 目录下的 FatJar 文件即可将应用部署到 EDAS。

使用 Jenkins 实现 EDAS 持续集成

概述

使用 Jenkins 可以构建 EDAS 应用的持续集成方案。该方案涉及下面的计算机语言或开发工具，阅读本文需要对下述的语言或工具有一定的理解。

工具	说明
Maven	Maven 是一个项目管理和构建的自动化工具。
Jenkins	Jenkins 是一个可扩展的持续集成引擎。
GitLab	GitLab 是一个利用 Ruby on Rails 开发的开源应用程序，实现一个自托管的 Git 项目仓库，可通过 Web 界面进行访问公开的或者私人项目。 它拥有与 GitHub 类似的功能，能够浏览源代码，管理缺陷和注释。

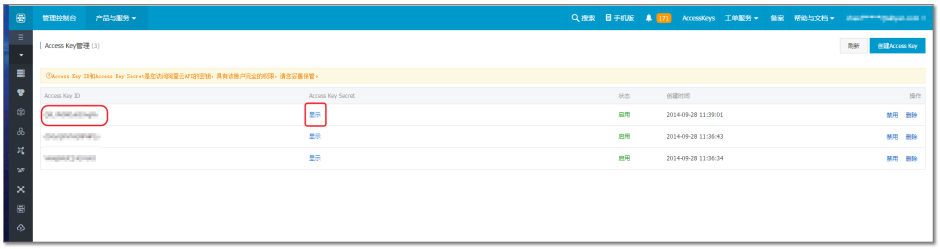
准备工作

在开始持续集成之前，需要完成下述的准备工作。

获取阿里云的 Access Key ID 和 Access Key Secret。

使用已经开通了 EDAS 服务的**主账号**登录阿里云官网。

进入 Access Key 控制台，创建 Access Key ID 和 Access Key Secret。



在 EDAS 控制台创建应用。

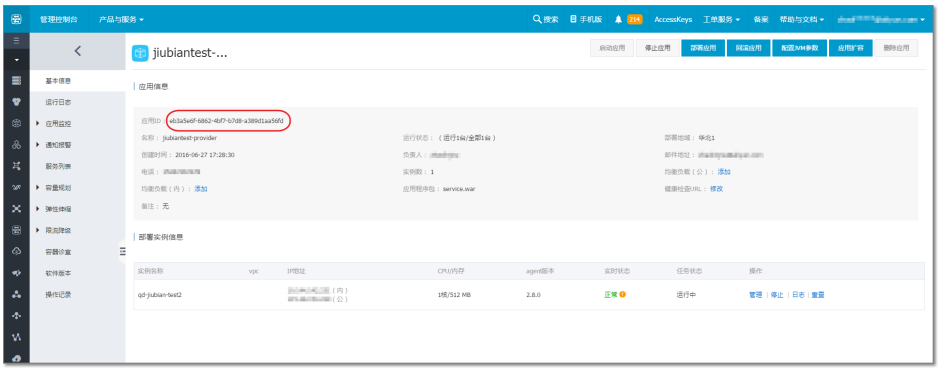
在使用 Jenkins 自动部署应用之前，需要先在 EDAS 控制台中创建一个可以部署的应用。

登录 EDAS 控制台。

参考发布应用 ，创建应用。

如果已经创建了应用，请忽略此步。

在左侧导航栏中单击**应用管理**。找到您在上一步中创建的应用并单击进入详情页面，获取应用 ID 的字段内容。



使用 GitLab 托管您的代码。

可以自行搭建 Gitlab 或者使用阿里云 Code。

本文使用通过自行搭建的 GitLab 做演示，关于 Gitlab 的更多信息请参考 GitLab。

了解并使用 Jenkins。

关于 Jenkins 的更多详细信息请参考 Jenkins 官网。

目前，阿里云还没有合适的产品替代 Jenkins，不过即将推出基于 Jenkins 的 DevOps 平台，请持续关注。

创建持续集成

创建持续集成主要包含以下三个步骤：

安装和配置 Jenkins

安装 Jenkins。

安装 Jenkins，请参考 Jenkins。 如已安装则请忽略此步。

在 Jenkins 服务器安装 Python 运行环境（仅支持2.7及以上版本，不支持 Python3）。

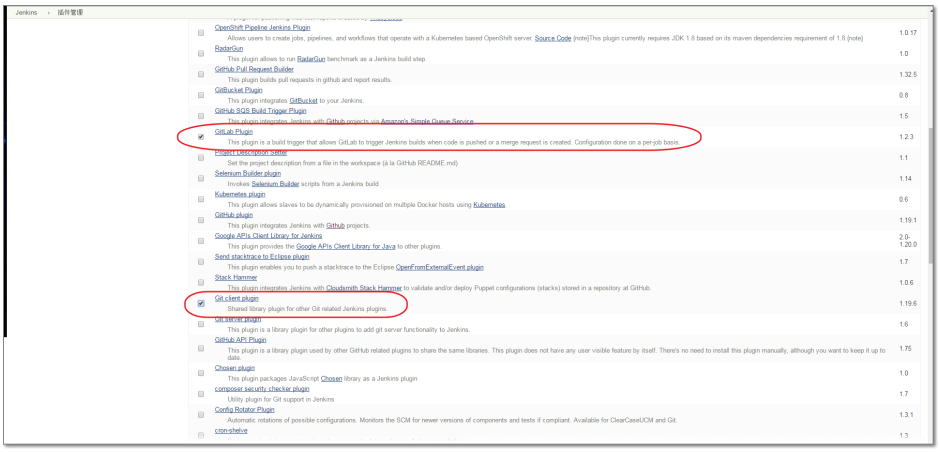
安装 Python，请参考 Python。 如已安装请忽略此步。

在 Jenkins 中安装 Git 和 GitLab 插件。

在 Jenkins 控制台的菜单栏中选择**系统管理** > **插件管理**，安装插件。

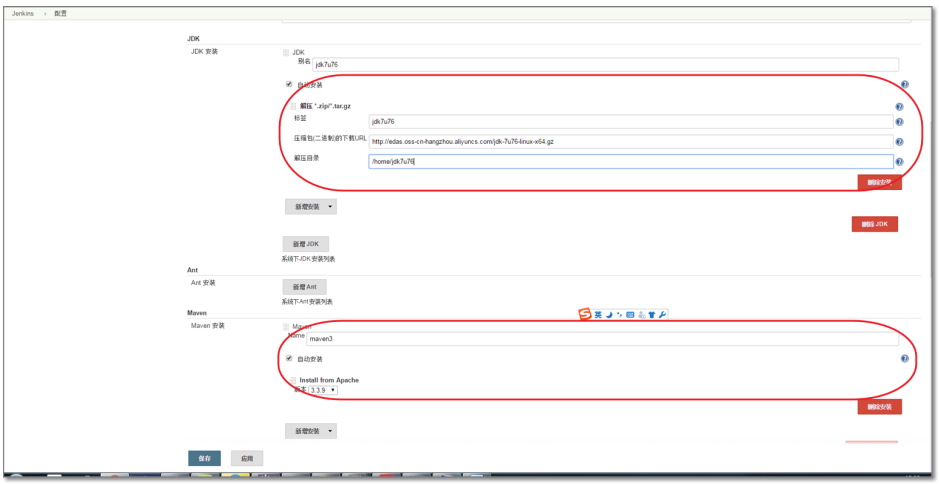
安装 GIT Client Plugin 和 GIT Plugin 插件可以帮助 Jenkins 拉取 Git 仓库中的代码。

安装 Gitlab Hook Plugin 插件可以帮助 Jenkins 在收到 Gitlab 发来的 Hook 后触发一次构建。



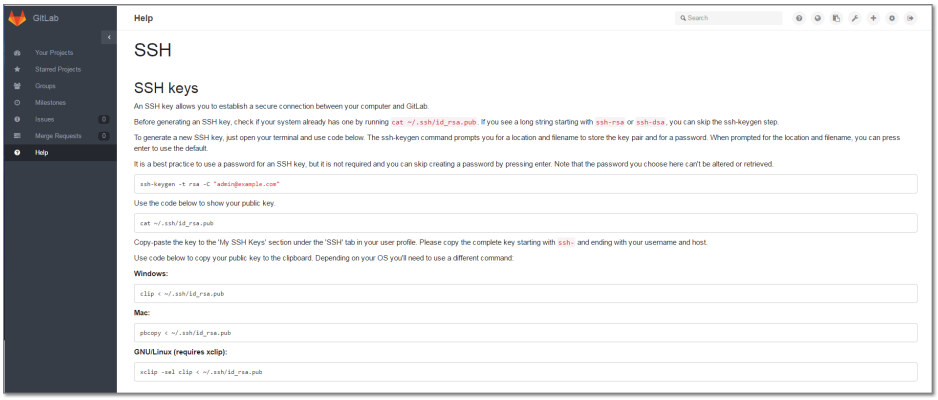
安装 JDK 和 Maven。

在 Jenkins 控制台的菜单栏中选择**系统管理** > **系统设置**，参考下图中的标示为 Jenkins 安装 JDK 和 Maven。

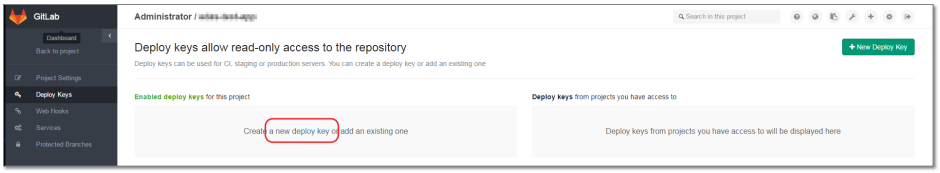


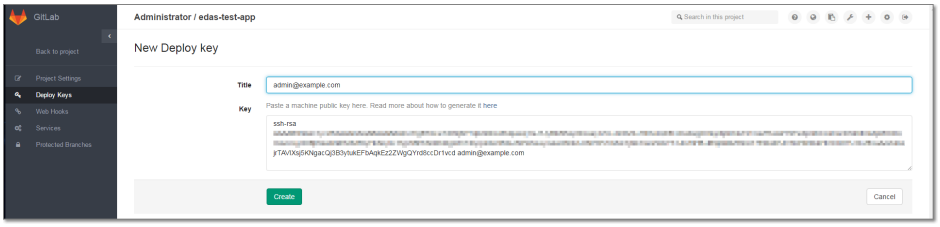
生成 RSA 密钥对，导入 GitLab 和 Jenkins。 实现 Jenkins 拉取 GitLab 代码时的认证。

参考 GitLab 文档，创建 RSA 密钥对。



进入您的项目的 GitLab 首页，在菜单栏选择 **Settings** > **Deploy Keys**。然后单击 **new deploy key** 添加 key，导入密钥。



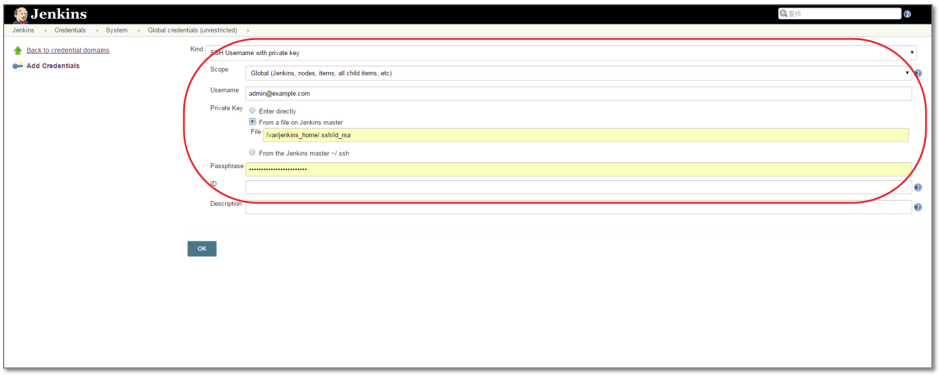


通过 RSA 私钥添加 Jenkins 认证。

在 Jenkin s 首页单击 **Credentials** 菜单。 单击 **Add credentials**，在下图页面输入相关信息，单击 **OK**。

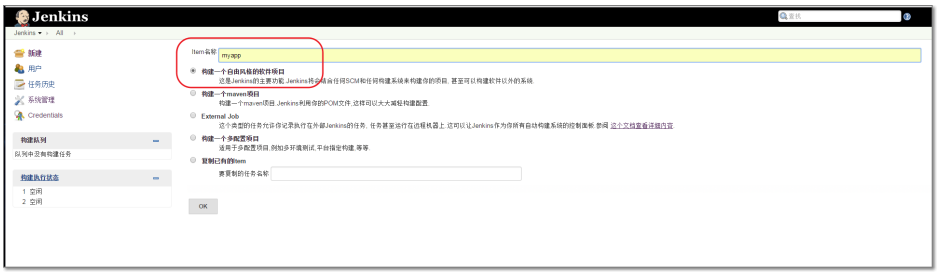
选择 **SSH Username with private key** 的认证方式。

按照图例填写 **Scope**，**UserName**， **Private Key** 等配置， 将第一步生成 RSA 密钥对中生成的私钥文件拷贝到”/var/jenkins_home/.ssh/id_rsa”文件。 图例中的 Scope、UserName、Private Key 可以根据您的需要来填写。

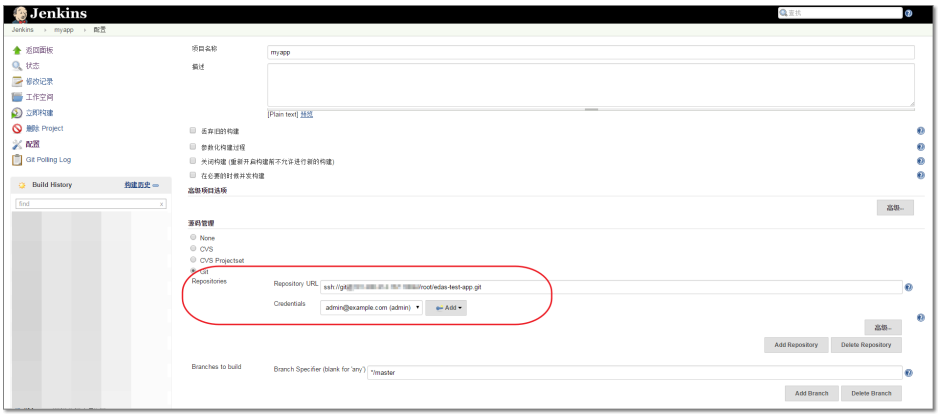


创建 Jenkins 项目。

在 Jenkins 首页左侧单击**新建**，创建 Jenkins 项目。



配置 Git 项目地址时，勾选上一步中通过 RSA 创建的认证方式。正确配置后如下图所示。“Poll SCM” 必须勾选。



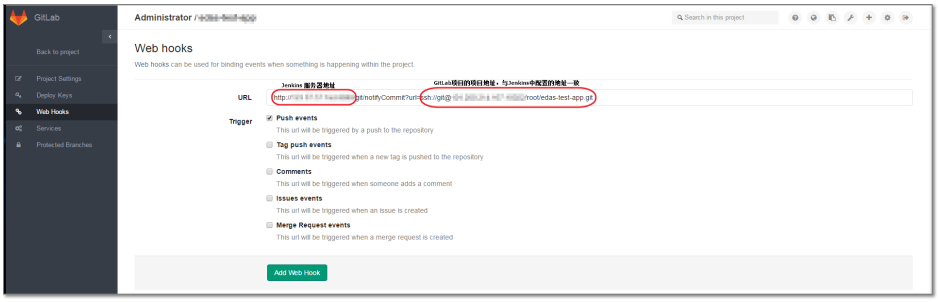
配置 Maven 构建和自定义构建动作 **Execute shell**（本文示例通过调用 Shell 命令完成构建后的自动部署，如果您的 Jenkins 是在 Windows 服务器上搭建的，则需要选择 **Execute Windows batch command**）。

上图中配置的 *“python $WORKSPACE/build/build.py”* 是一个 EDAS 提供的示例脚本，该脚本主要完成 WAR 包的上传和应用的部署工作。具体如何使用，在接下来的第三步、第四步中有详细说明。

注：如果部署的是 Docker 应用，脚本” python $WORKSPACE/build/build.py” 请使用” python $WORKSPACE/build/dockerbuild.py”

配置 Gitlab 的 Web Hook，实现自动构建

单击 GitLab 工程进入配置（” Settings” ）页面，参考下图进行配置。 图中表示的 Jenkins 服务器地址为您的 Jenkins 服务器的 Web 访问地址如 http://localhost:8080/。



配置完成后可以单击页面中的 “Test Hook” 进行测试。

调用 EDAS Open API 进行部署

下载示例工程 demo.zip。

拷贝示例中的 build 目录到您的 Git 工程中。

打开 build 目录中的.osscredentials 文件，配置为您在准备工作中获取的 Access Key ID 和 Access Key Secret。

```
[OSSCredentials]
accessid = 您的Access Key ID
accesskey = 您的Access Key Securet

```

由于部署分为普通应用和 Docker 应用，由于 Open API 不相同，配置方式不同，分别介绍：

普通应用

打开 build 目录中的 config.json 文件，配置 WAR 包地址，应用 ID 等属性。 配置文件格式满足 JSON 格式。

```
{
    "edas": {
        "host": "edas.console.aliyun.com",
        "port": 80
    },
    "apps": [
        {
            "appName": "jenkins-consumer",
            "appId": "b8ce22a3-4554-4385-bb89-f43848eedea1",
            "userId": [redacted],
            "target": "edas-demo-portal/target/portal.war",
            "deployGroupId":"",
            "batch":1
        },
        {
            "appName": "jenkins-provider",
            "appId": "faf4ffb1-91a7-42bf-9ea1-4a764106695d",
            "userId": [redacted],
            "target": "edas-demo-service/target/service.war",
            "deployGroupId":"",
            "batch":1
        }
    ]
}

```

“host”配置项代表了您访问的 EDAS 的控制台的域名或者 IP，如果您使用的是公有云的 EDAS 服务，则无需修改”edas”的”host”和”port”属性。

“apps”配置项中可以配置多个应用，也可以只配置一个。 各配置项的含义及获取方式如下：

- appName：应用名称，准备工作中创建，通过 EDAS 控制台可以取到。
- appId：应用 ID，准备工作中创建，通过 EDAS 控制台可以取到。
- userId；您登录阿里云的用户 ID。

- target：Maven 编译后打出来的 WAR 的本地路径，以 Git 项目为根目录的相对目录。
- deployGroupId：应用分组 ID。
- batch：分批发布。

Docker 应用

Docker 应用支持 WAR 部署和镜像部署（其中镜像部署只需要在 Jenkins 项目中配置 Maven 构建和自定义构建动作 **Execute shell**，内容为”python $WORKSPACE/build/dockerbuild.py”。）

打开 build 目录中的 dockerconfig.json 文件，配置 WAR 包地址，应用 ID 等属性。 配置文件格式满足 JSON 格式。

```
{
    "edas": {
        "host": "edas.console.aliyun.com",
        "port": 80
    },
    "apps": [
       {
            "appName": "doc-jenkins-consumer",
            "appId": "6e941be8-8c9e-4cb9-9ac8-6b3263e99173",
            "userId": [redacted],
            "target": "edas-demo-portal/target/portal.war",
            "type": "upload",
            "deployToStr": "all",
            "packageVersion":"1.3",
            "description": "",
            "imageUrl":"",
            "regionId":"cn-hangzhou"
        },
        {
            "appName": "doc-jenkins-provider",
            "appId": "9b670366-9ba0-4336-88da-1d8a9bad5773",
            "userId": [redacted],
            "target": "",
            "type": "image",
            "deployToStr": "all",
            "packageVersion":"1.3",
            "description": "",
            "imageUrl":"registry.cn-hangzhou.aliyuncs.com/edas_test1/onenight:1",
            "regionId":"cn-hangzhou"
        }
    ]
}
```

“apps”配置项中可以配置多个应用，上图配置了两个应用，第一个为 Docker 应用 WAR 包部署方式，第二个为 Docker 应用镜像部署方式，各配置项的含义及获取方式如下：

- appName：应用名称，准备工作中创建，通过 EDAS 控制台可以取到。
- appId：应用 ID，准备工作中创建，通过 EDAS 控制台可以取到。
- userId；您登录阿里云的用户 ID。
- type：部署方式类型。upload 为 WAR 包部署，image 为镜像部署。
- target：Maven 编译后打出来的 WAR 的本地路径，WAR 部署不能为空。
- imageUrl：镜像地址。image 部署时，不能为空。
- packageVersion：部署包的版本号。
- description：描述信息。

- deployToStr：应用分组 ID。 为” all”时，代表该应用所有应用实例。
- regionId：区域 ID，应用所在的区域 ID。

配置正确后，提交变更到 GitLab。

如果上述步骤配置正确，这次提交会触发一次 GitLab Hook。 Jenkins 在接受到这个 Hook 后会构建您的 Maven 项目，并在构建结束时调用 Open API 触发部署。

说明：如为 Docker 镜像部署方式，配置完成后，可手动触发 jenkins 项目。

开发问题排查

开发中遇到的很多问题都可以通过查看相关日志进行定位、解决。下表是 EDAS 相关日志路径汇总。

文件名	说明
/home/admin/taobao-tomcat-production-xxxx/logs/catalina.out	最重要的日志，可以看到应用和 Tomcat 应用服务器的异常，这是开发最应该关注的日志。
/home/admin/taobao-tomcat-production-xxxx/logs/localhost.log.xxx	如果通过 catalina.out 看到的错误很模糊或者没有错误，可以结合 localhost 来一起看，保证应用正常启动，再看下面的日志继续排查问题。
/home/admin/configclient/logs/config.client.log	可以通过此日记查看服务发布订阅是否成功，“Register-ok”、“Publish-ok” 等关键字。
/home/admin/logs/hsf/hsf.log	HSF 服务的日志，有 HSF 服务调用过程的一些详细信息。如果 HSF 调用中出现异常，可以看看这个日志。

作为应用的容器，简单的排查问题的思路就是：

查看 /home/admin/taobao-tomcat-production-xxxx/logs/catalina.out

查看看 /home/admin/taobao-tomcat-production-xxxx/logs/localhost.log.xxx

找到 Tomcat 相关的第一个错误，解决第一个错误后重启观察, 看是否还有其它错误。