DSF分布式应用服务框架

使用手册

2014

肖海星arksea@gmail.com

百度91无线

2014/10/9

#### 目录

[1 DSF简介 3](#_Toc400801844)

[1.1 DSF的开发背景 3](#_Toc400801845)

[1.2 DSF的目标 3](#_Toc400801846)

[1.3 技术基础 3](#_Toc400801847)

[2 客户端编程入门 3](#_Toc400801848)

[2.1 查看服务接口定义 3](#_Toc400801849)

[2.2 获取服务接口代码 4](#_Toc400801850)

[2.3 获取DSF框架 5](#_Toc400801851)

[2.4 客户端访问服务的步骤 5](#_Toc400801852)

[2.5 客户端的例子代码 6](#_Toc400801853)

[3 客户端连接池的使用 6](#_Toc400801854)

[3.1 短链接ClientSource 6](#_Toc400801855)

[3.2 连接池PooledClientSource 6](#_Toc400801856)

[3.3 SpecifiedServerClientSource 9](#_Toc400801857)

[4 服务端开发入门 9](#_Toc400801858)

[4.1 接口定义 9](#_Toc400801859)

[4.2 应用服务实现 11](#_Toc400801860)

[4.3 应用服务管理 13](#_Toc400801861)

[4.4 AppManager配置 15](#_Toc400801862)

[4.5 应用服务管理类及其配置 15](#_Toc400801863)

[4.6 服务路由策略 15](#_Toc400801864)

[4.7 传输协议 15](#_Toc400801865)

[4.8 失败处理策略 15](#_Toc400801866)

[4.9 客户端流控 15](#_Toc400801867)

[5 DSF框架Util 15](#_Toc400801868)

[5.1 资源泄露监控（Java） 15](#_Toc400801869)

[5.2 简单的Actor框架 15](#_Toc400801870)

[附件A：应用服务管理规范 16](#_Toc400801871)

# DSF简介

DSF的全称是Distributed Service Framework，是一个分布式跨平台的应用服务框架，包括有Java和C#版本

## DSF的开发背景

91在线应用家族的业务和功能不断的增加，同时技术团队在不断的发展变化，网站变得越来越大、服务越来越多时，服务的URL地址信息呈现爆炸式增长，配置管理变得非常困难，LVS负载均衡器的单点压力也越来越大。

同时各个子系统、模块实现的技术五花八门，部署时各子系统的方式和要求不同，各个子系统之间的交互方式和方法不统一。这种现象使得服务间的依赖愈来愈错综复杂，即使系统架构师也很难完整描述应用服务的架构关系。

## DSF的目标

1. 打造一个跨平台、松耦合、基于服务的框架，简化服务的发布与订阅
2. 构建一个统一的编程模型，从而能让开发人员把精力集中于业务的接口与实现
3. 为服务提供水平伸缩能力，降低建立高性能、高可靠分布式集群的成本

## 技术基础

Thrift是一个RPC框架，DSF基于这个框架开发。Thrift是2007年Facebook提交给Apache的一个开源项目，Facebook开发Thrift是为了解决系统中各系统间大数据量的传输通信以及系统之间语言环境不同需要跨平台的特性。

# 客户端编程入门

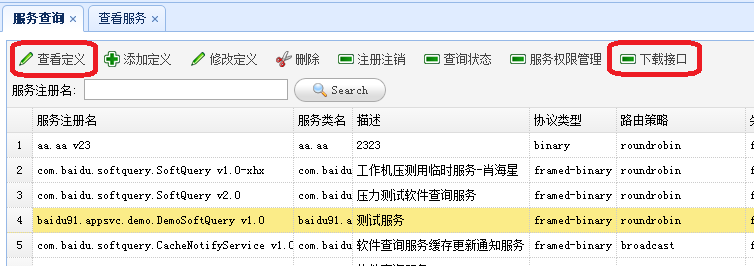
本章将讲解如何调用已经定义部署的DSF应用服务

## 查看服务接口定义

首先我们需要在“应用服务管理系统”找到要调用的应用服务。“应用服务管理系统”有内网的测试版本与IDC的线上版本，地址分别为

http://192.168.253.251:8181/webtoolplat

http://10.79.141.54:8080/webtoolplat



根据应用服务端提供的“注册名”比如“baidu91.appsvc.demo.DemoSoftQuery v1.0”，找到相应的服务，先来看看它的接口定义，点击“查看定义”可以看到接口的ThriftIDL定义：

namespace java baidu91.appsvc.demo

namespace csharp baidu91.appsvc.demo

struct SimpleSoft {

1: i64 softid,

2: string name,

3: string vender,

4: binary icon,

}

exception DemoException {

1:string message

}

service DemoSoftQuery {

void ping(),

list<SimpleSoft> getAll() throws (1:DemoException ex),

SimpleSoft getByID(1:i64 id) throws (1:DemoException ex),

list<SimpleSoft> getByVender(1:string vender) throws (1:DemoException ex),

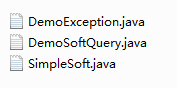
void put(1:i64 id, 2:SimpleSoft soft) throws(1:DemoException ex)

}

图表1 Demo服务定义

## 获取服务接口代码

选择并点击“下载接口”，解压下载的zip文件得到接口代码，比如java：



将这些文件添加到你的项目中

## 获取DSF框架

### gradle

在项目的build.gradle中添加以下配置，gradle会自动下载依赖的包

repositories {

maven {url "http://192.168.253.149:8081/nexus/content/groups/public/"}

}

dependencies {

compile "com.baidu:dsframework:1.2-SNAPSHOT"

}

### maven

<repositories>

<repository>

<id>public</id>

<url>http://192.168.253.149:8081/nexus/content/groups/public/</url>

</repository>

</repositories>

<dependency>

<groupId>com.baidu</groupId>

<artifactId>dsframework</artifactId>

<version>1.2-SNAPSHOT</version>

</dependency>

### 手动添加依赖jar包

因为DSF还在开发测试当中，不建议使用这种方式

## 客户端访问服务的步骤

## 客户端的例子代码

//配置应用注册服务器地址，启动DSF框架

InetSocketAddress[] addrs = new InetSocketAddress[2];

addrs[0] = new InetSocketAddress("192.168.253.251", 9002);

addrs[1] = new InetSocketAddress("192.168.253.149", 9002);

DSFramework.start(addrs, "my\_test\_client");

//用指定的服务接口与服务注册名创建Source

IClientSource<DemoSoftQuery.Iface> pool;

pool = new ClientSource<>("baidu91.appsvc.demo.DemoSoftQuery v1.0", 10000);

//从Source中获取接口

DemoSoftQuery.Iface client = pool.getClient();

try {

//调用接口，获取并输出查询结果

SimpleSoft s = client.getByID(1234);

logger.info(s.toString());

} catch (Exception ex) {

logger.error("test call failed", ex);

} finally {

//接口归还Source，有Source负责关闭连接，必须在finally中避免资源泄露，耗尽连接

pool.returnClient(client);

}

...

...  
//程序退出时，关闭DSF

DSFramework.stop();

# 客户端连接池的使用

## 短链接ClientSource

前一章例子代码使用的ClientSource是接口IClientSource的短连接实现，ClientSource类本身线程安全，但要注意取得的Client非线程安全；

短链接的ClientSource适合大部分应用，比如：远程方法执行时间远大于连接创建时间(看网络情况大约为数毫秒)的服务，其连接创建时间可以被忽略，此时短连接策略基本不会带来有影响的性能损失；另外，对于非频繁调用、对延迟时间不敏感的服务也适合使用短连接策略。 当调用者获取Client时，此实现将新建一个Client并返回。

当调用者调用IClientSource. returnClient时，此实现只是简单的直接关闭客户端连接。

## 连接池PooledClientSource

对于高并发或者高吞吐量的应用，网络连接的创建消耗是很大的，对于这种应用应该使用长连接策略的连接池实现PooledClientSource。使用起来也很简单，直接上代码再解释配置参数吧：

PooledClientSource<DemoSoftQuery.Iface> pool;

GenericObjectPoolConfig cfg = new GenericObjectPoolConfig();

cfg.setMinIdle(3);

cfg.setMaxTotal(10);

cfg.setTestOnBorrow(true);

cfg.setTimeBetweenEvictionRunsMillis(30000);

cfg.setTestWhileIdle(true);

pool = new PooledClientSource<>("baidu91.appsvc.demo.DemoSoftQuery v1.0", cfg, 10000);

...

...

//不再使用连接池时，需要关闭释放所有连接

pool.dispose();

### 几个常用的参数

MaxWaitMillis：最大等待时间，默认为-1，即无可用连接会抛出异常，设为0表示无穷大

MinIdle：最小空闲连接数

MaxIdle：最大空闲连接数

MaxTotal：连接池最大持有连接数

TestOnBorrow：从池中获取客户端接口时进行连接有效性测试

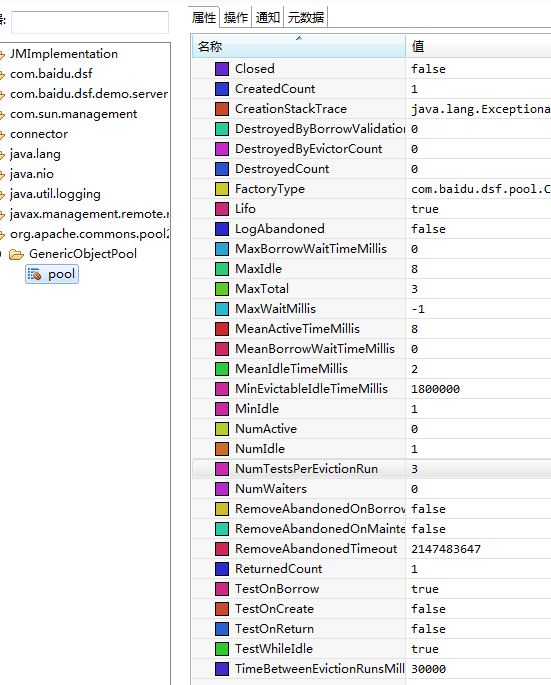
TestOnReturn：将客户端接口归还连接池时进行有效性测试

TestWhileIdle：当连接闲置时进行连接有效性测试，相当于增加了连接心跳检测的效果

TimeBetweenEvictionRunsMillis：空闲连接检测间隔，如果配置了TestWhileIdle为true则同  
 时进行连接有效性测试

PooledClientSource是基于ApacheCommonPool2对象池库实现的，具体的配置可以参考官方文档：[BaseObjectPoolConfig](http://commons.apache.org/proper/commons-pool2/apidocs/org/apache/commons/pool2/impl/BaseObjectPoolConfig.html)、[GenericObjectPoolConfig](http://commons.apache.org/proper/commons-pool2/apidocs/org/apache/commons/pool2/impl/GenericObjectPoolConfig.html)

JmxEnabled：如果你的程序配置了JMX服务，你可以通过这个参数向其注册连接池的管理，  
 我们可以从JMX管理工具中观察，连接池的使用情况。



### 连接有效性测试

连接有效性测试可以大大减少长连接失效造成的远程调用失败，对于那些对连接失效而造成的调用失败很敏感的服务，可以开启各种合适的连接有效性测试策略来保障所取得的客户端是连接正常的。

DSF应用服务可以配置连接有效性测试使用的方法，通常是在服务接口定义中增加一个ping方法，当然也可以复用那些消耗很小的无参数接口调用

...

service DemoSoftQuery {

void ping(),

list<SimpleSoft> getAll() throws (1:DemoException ex),

SimpleSoft getByID(1:i64 id) throws (1:DemoException ex),

list<SimpleSoft> getByVender(1:string vender) throws (1:DemoException ex),

void put(1:i64 id, 2:SimpleSoft soft) throws(1:DemoException ex)

}

并在“应用服务管理系统”中定义服务时设置服务接口附加属性validate = ping，如果服务定义了有效性测试方法，则可以在连接池中开启连接有效性测试属性，具体情况请参考服务定义章节。当然如果应用服务没有定义或配置连接有效性测试方法，则在连接池中开启这些配置参数也不会引起其他负面影响。



## SpecifiedServerClientSource

这是一个特殊的IClientSource实现，继承自ClientSource，是一个短链接实现，主要是用来做测试的。

ClientSource与PooledClientSource会根据应用服务定义的路由策略对部署的所有服务节点进行轮询，而当我们准备上线一个新的服务节点前，可以用SpecifiedServerClientSource指定这台服务器进行上线前的测试。测试通过后再调用管理接口向注册服务器注册。

下面是这个类的构造函数，参数host与port即是用来指定服务节点地址的

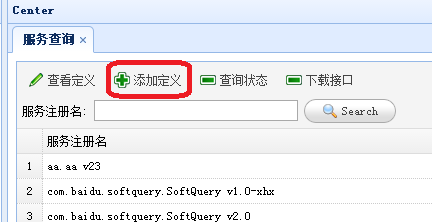
SpecifiedServerClientSource(String regname, String host, int port, int timeout)

# 服务端开发入门

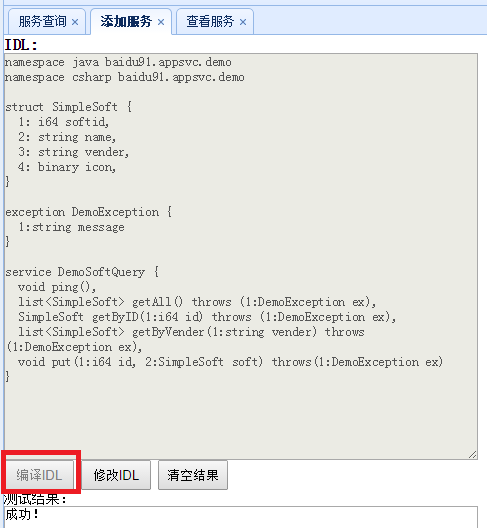
## 接口定义

DSF的RPC协议基于Apache Thrift项目，所以接口是使用Thrift IDL语言进行定义，IDL规范可以参考官方文档：[Thrift interface description language](http://thrift.apache.org/docs/idl)，不过事实上它很简单，就好比Json你只要看一个例子就基本可以掌握80%的内容了，比如客户端调用例子中的，与Java和C#的接口定义差不多，只是语法与关键字稍有区别。

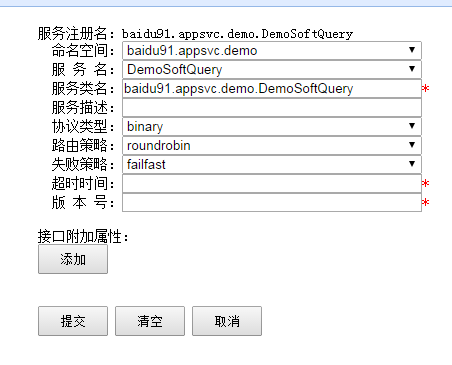
首先，你需要登录，如果没有账号，请联系张海（zhanghai@91.com）同学申请权限。



点击“添加定义”按钮，在添加服务页面中的IDL框中填入定义的服务接口，建议每个服务都定义用于测试服务连通性的ping方法，然后点击“编译IDL”按钮



如果服务定义语法正确，将会显示编译成功，右边的应用服务信息中会自动填入大部分必须的内容



我们还需要填写服务描述、超时时间（ms）、版本号，并选择我们需要的协议类型、路由策略与失败策略，点击提交即完成了应用服务的定义工作。

如果服务接口定义了ping方法，可以将其配置为连接有效性测试方法，当然其他无参数的低消耗方法也可以用来代替ping的功能，在提交服务定义前点击接口附加属性“添加按钮”，在属性名与属性值中分别填入validate与ping



## 应用服务实现

获取服务接口与DSF框架库请参考客户端开发的相应章节，方法和过程是一样的。

### 应用服务实现的步骤

### 实现服务接口

实现接口没啥好说的，根据业务写代码，直接上例子

public class DemoSoftQueryHandler implements DemoSoftQuery.Iface {

private static final Logger logger = LoggerFactory.getLogger(DemoSoftQueryHandler.class);

@Override

public void ping() throws TException {

logger.info("ping");

}

@Override

public List<SimpleSoft> getAll() throws DemoException, TException {

logger.info("getAll");

List<SimpleSoft> list = new LinkedList();

SimpleSoft s = new SimpleSoft();

s.softid = 1234567;

s.name = "hello world!";

s.vender = "baidu91";

list.add(s);

return list;

}

...

注意红色的部分，DemoSoftQuery.Iface就是我们定义的服务Thrift编译出来后的接口类，Thrift将接口放在类的内部，有点奇怪的做法，是为了减少生成文件的数量还是简化IDE编译器的实现？但是既然用了Thrift就忍了吧:-)

### Spring组装服务

<!-- 读取应用服务通用配置文件 -->

<context:property-placeholder properties-ref="props"/>

<util:properties id="props" location="file:./config/appconfig.properties"/>

<!-- 配置应用服务管理Bean，程序入口使用此Bean启动应用服务 -->

<bean id="appManager" class="com.baidu.dsf.server.AppManager">

<property name="registerServers" value="#{props['register\_server\_adds']}"/>

<property name="appName" value="dsf\_test\_server"/>

<property name="serviceList">

<list>

<ref bean = "demoSoftQueryManager"/>

</list>

</property>

</bean>

<bean id="demoSoftQueryManager"

class="com.baidu.dsf.demo.server.DemoSoftQueryManager">

<property name="serviceHandler" ref="demoSoftQueryHandler"/>

<property name="serviceName" value="baidu91.appsvc.demo.DemoSoftQuery"/>

<property name="version" value="v1.0"/>

<property name="dialedMethod" value="ping"/>

<property name="serviceBindHost" value="#{props['service\_bind\_host']}"/>

<property name="serviceBindPort" value="#{props['service\_bind\_port']}"/>

<property name="selectorThreads" value="#{props['selectorThreads']}"/>

<property name="workerThreads" value="#{props['workerThreads']}"/>

<property name="acceptQueueSizePerThread"

value="#{props['acceptQueueSizePerThread']}"/>

</bean>

<bean id="demoSoftQueryHandler"

class="com.baidu.dsf.demo.server.DemoSoftQueryHandler">

</bean>

</beans>

用Spring将前一步实现的接口注入到框架提供的应用管理类中，即完成了应用服务的装配，其他的配置都是应用服务的标准配置，按照需要修改即可。

对于.Net可以使用Spring.Net代替，如果不使用Spring装配服务，也可以用代码手工装配，只需要简单的new出这三个类，并设置相应的属性即可。

### 在Main中启动服务

public static void main(String[] args) {

applicationContext = new FileSystemXmlApplicationContext(

"./config/context\_app.xml");

AppManager app = applicationContext.getBean("appManager", AppManager.class);

app.startApplication(3000,0);

}

就只要少量的代码，你就已经将你实现的服务接口发布为一个可扩展的分布式应用服务了，是不是很简单！

服务启动后，AppManager会自动向应用注册服务器注册服务，我们可以通过“应用服务管理系统”查看服务的状态与订阅者信息。Demo服务没啥信息看，我们来看一个线上服务吧

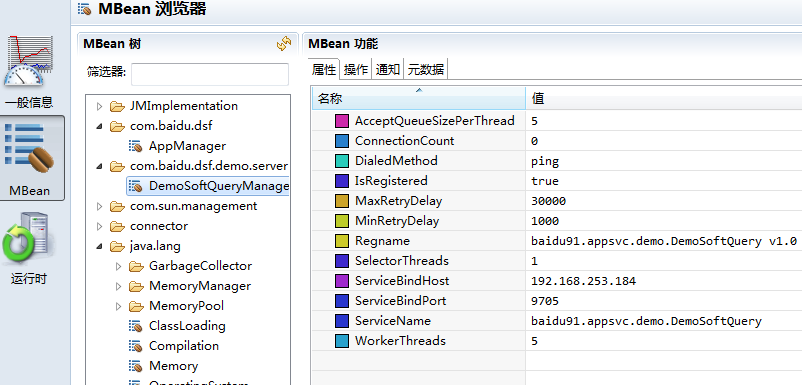


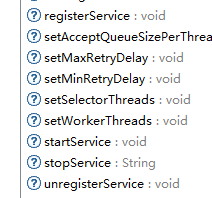
## 应用服务管理

前一章，我们用Spring装配服务，将服务接口实现DemoSoftQueryHandler装配到服务管理类DemoSoftQueryManager中，又用AppManager启动了整个服务，那么这两个管理类是做什么用的呢？正如其名字中的Manager所示，这两个类中预置了JMX的管理方法，我们可以用JDK的JMX工具来管理我们的应用。

首先我们打开JMX管理工具JMC.exe，这个程序是安装JDK时自带的。我们来看看这两个管理类提供的功能

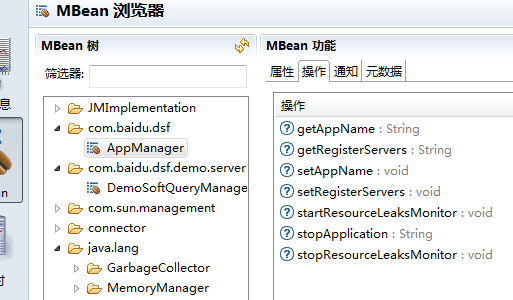
#### DemoSoftQueryManager





常用的有启、停服务，注册与注销服务，修改配置等

#### AppManager



可以通过stopApplication安全的停止应用程序，给应用清理资源，保存信息的机会，避免粗暴的Kill杀死进程所造成的一系列问题。

## AppManager配置

todo

## 应用服务管理类及其配置

todo

## 应用服务定义

在定义应用服务时可以配置服务的路由策略、传输协议与失败处理策略

#### 服务路由策略

roundrobin：轮询策略

broadcast：广播策略

weighting-roundrobin：带权重的轮询策略

#### 传输协议

binary 二进制协议，通常用于同步模型的服务，C#作为服务端只支持这种服务

framed-binary 分帧的二进制协议，用于异步模型的服务，Java版服务默认使用这种协议

zip-binary 带压缩的二进制协议，C#可以作为客户端可以支持这种协议

#### 失败处理策略

failfast：快速失败策略，远程调用失败后直接抛出异常

failover：重试策略，远程调用失败后会重新获取一个可用的连接进行重试，  
 再次失败才会抛出异常

failsafe：静默模式，对于可以容忍偶发性失败的通知、远程日志等服务可以使用这种方式

### 客户端流控

在定义应用服务时可以通过添加扩展属性client\_rate\_limit、client\_rate\_limit\_rt、rate\_upgrade来配置客户端流控，以下分别介绍这些参数的作用。

#### client\_rate\_limit

参数的配置格式:

client\_rate\_limit=rate\_limit,min

其中rate\_limit为正常状态限流量，min为超时错误状态下的最小降级限流量（次/秒）

这个参数用于保护服务器，限制每个客户端针对单台服务实例每秒调用的最大次数（QPS）。这个值的估算方法通常是这样：首先对**单台**服务实例进行压力测试，获得其最高负载MAX\_QPS，乘以一个裕量比如50%，再除以可能的客户端数。

客户端针对某台服务实例的QPS超过设定值的70%时，DSF框架将会记录警告级别的日志，超过80%时将记录错误级别的日志。（如果配置了alarm\_appender这些警告日志将会发往告警系统，在告警板中显示，相关维护人员将会收到告警短信）。如果发生了这种情况通常是需要对服务器进行扩容了。

没有配置这个参数，正常状态下将不进行客户端流控，“超时错误状态”下最小降级流量将为每秒1次

#### rate-upgrade与 超时错误限流

当客户端调用发生java.net.SocketTimeoutException或者java.concurrent.TimeoutException异常时，DSF将会启动超时错误限流功能。

1. 参数的配置格式：

rate-upgrade= coef,period

其中coef为流量升级与降级的系数，period为流量升级的周期，单位是毫秒

1. 流量降级：每5000毫秒，以当前QPS/coef为目标QPS进行流量控制
2. 流量升级：每period毫秒，以当前QPS\*coef为目标QPS进行流量控制

参数配置与调试的方法：首先确定coef控制流量降级速度，然后修改周期period让coef与period的组合满足流量升级的速度

#### client\_rate\_limit\_rt

如果配置了这个参数，客户端调用的RT时间超过其配置的值，将启动限流功能

# 附录A：应用服务管理规范

#### 应用服务必须提供管理接口

* 注册注销服务
* 启停服务，停止服务必须二次确认
* 停止进程，必须二次确认
* 修改重要配置
* 查询主要性能参数

#### 应用服务安装时必须安装随操作系统自启动的脚本

#### 应用服务必须提供手工启动、停止服务脚本

为保障数据安全，非必要不得用kill的方式停止服务进程

#### 应用服务在手工启动时不得自动注册应用服务，程序必须提供参数支持此能力

#### 此规范的目的在于服务注册上线前需要对服务的正确性进行测试检查

#### 应用服务下线操作流程

因为升级、修改配置、维护重启等原因，需要将一个应用服务节点下线，必须遵照以下标准流程进行操作， 此流程的目的是为了保障升级、维护过程中，线上服务的无缝切换

* 征得主管许可确认，做好相关应急预案
* 确认下线后，剩余的节点可以承载当前的业务
* 通知各相关业务负责人注意观察是否异常
* 通过管理接口向注册服务器注销服务节点
* 等待本节点负载降到零或者超过预期等待时间
* 向相关业务负责人确认是否有异常情况
* 确认各个剩余节点负载正常
* 如果上述两点中发现下线本节点会造成影响，重新注册服务，取消本次下线操作

#### 应用服务配置修改流程

* 参照3.5对应用服务进行下线操作
* 通过管理接口停止应用服务
* 通过管理接口临时修改应用配置
* 通过管理接口启动应用服务
* 用测试客户端测试应用服务保证其能正常提供服务
* 通过管理接口向注册服务器注册应用服务
* 观察业务负载是否正常切换到本节点
* 确认配置修改正确并符合需求，联系OP永久修改服务本地配置文件

#### 应用服务在线升级

* 在线升级必须保证向下兼容性
* 升级安装包必须经过QA或RD测试验证
* 尽量提供自动安装脚本，减少误操作的可能
* 参照3.5对应用服务进行下线操作
* 通过管理接口停止应用服务进程
* OP安装升级包启动应用实例
* 用测试客户端测试应用服务保证其能正常提供服务
* 通过管理接口向注册服务器注册应用服务
* 观察业务负载是否正常切换到本节点

#### 应用服务离线升级

对于无法进行在线升级的应用需遵照本流程进行离线升级，比如：接口更改等原因无法保证向下兼容性

* 在应用服务管理网站增加一个同名新版本的服务定义
* 新版本服务上线，与旧版本服务并行运行，互不影响
* 新版本服务各项测试通过后，各业务客户端逐步切换使用新版本服务
* 所有线上服务切换到新版本，并正常运行一周后，旧版本应用服务才能进行下线操作

#### 应用服务负责人必须登记告警接收手机号

新应用服务部署前必须向集中告警系统登记本应用的source、host规则及相关负责人手机号码，用于接收告警短信