# **MSB接入服务动态刷新方案**

|  |  |
| --- | --- |
|  | **文档版本：V0.1** |

**修订记录**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **日期** | **修订版本** | **CR号** | **修改章节** | **修改描述** | **作者** |
|  |  |  |  |  | 支芳龙 |
| 2018-9-14 | V0.2 |  | 1.2.2-5 | 修改了动态刷新时序图 |  |
| 2018-9-14 | V0.2 |  | 1.2.1-3.1 | 解释了心跳间隔30秒的含义 |  |
| 2018-9-14 | V0.2 |  | 1.3.3 | 新增算法描述和代码举例 |  |
| 2018-9-14 | V0.2 |  | 1.2.2-3.3 | 增加了刷新结果的通知规范，对数据不一致的情况做了描述和应该采取的措施 |  |

目录

[**MSB接入服务动态刷新方案** 1](#_Toc524709292)

[**1.1** 概述 2](#_Toc524709293)

[**1.2** 接入步骤描述 3](#_Toc524709294)

[1.2.1 接入EUREKA 注册中心 3](#_Toc524709295)

[1. 概述 3](#_Toc524709296)

[2. 第一步，提交注册信息 （service->MSB） 3](#_Toc524709297)

[2.1 非spring cloud 架构的服务，主动访问HTTP接口 3](#_Toc524709298)

[2.2 基于spring cloud的服务 5](#_Toc524709299)

[2.3 调用时机 6](#_Toc524709300)

[2.4 服务注册时序图 6](#_Toc524709301)

[3. 第二步，发送心跳续租（service->MSB） 7](#_Toc524709302)

[3.1 非spring cloud架构的服务 7](#_Toc524709303)

[3.2 spring cloud架构的服务，注册中心客户端已实现 8](#_Toc524709304)

[3.3 发送心跳数据时序图 8](#_Toc524709305)

[1.2.2 服务配置动态刷新接入 8](#_Toc524709306)

[1． 概述 8](#_Toc524709307)

[2. 第一步，提供数据刷新接口（MSB->service） 8](#_Toc524709308)

[2.1 非spring cloud架构的服务 8](#_Toc524709309)

[2.2 spring cloud架构的服务 9](#_Toc524709310)

[3. 第二步，配置查询接口（service->MSB） 9](#_Toc524709311)

[3.1 非spring cloud 架构的服务 9](#_Toc524709312)

[3.2 spring cloud 架构的服务 10](#_Toc524709313)

[3.3 关于刷新结果通知 10](#_Toc524709314)

[3.4 推荐方案 11](#_Toc524709315)

[4. 在服务中使用获取的动态属性 11](#_Toc524709316)

[5. 动态刷新时序图 13](#_Toc524709317)

[**1.3** 安全 13](#_Toc524709318)

[1.3.1 敏感数据加密 13](#_Toc524709319)

[1.3.2 服务配置属性获取权限控制 13](#_Toc524709320)

[1.3.3 加密算法及JAVA举例 14](#_Toc524709321)

[1. 配置中心加密算法描述 14](#_Toc524709322)

[2. 加密算法 JAVA代码举例 14](#_Toc524709323)

[3. 客户端解密数据算法描述 16](#_Toc524709324)

[4. 解密算法JAVA举例 17](#_Toc524709325)

## 概述

MSB平台实现了对接入服务的集中配置管理，接入服务必须实现相应的接口规范。所有操作以HTTP API 完成交互，数据格式为JSON，符合Restful 规范。接入服务请按照如下接入步骤，实现或调用相应的Rest接口即可。

## 接入步骤描述

### 接入EUREKA 注册中心

#### 概述

服务接入MSB平台的第一步是主动向EUREKA注册中心提供注册信息，注册自己。只有注册到EUREKA的服务才能被网关发现，从而对外提供服务，并被MSB平台集中管理。**以下信息中，appID 是服务的名称，instanceID是全局唯一的服务实例的标识符。**

#### 第一步，提交注册信息 （service->MSB）

访问服务注册接口，提交注册信息；

##### 2.1 非spring cloud 架构的服务，主动访问HTTP接口

测试环境服务注册接口地址：

[http://188.0.96.27:30113/eureka/apps/{appID}](http://188.0.96.27:30113/eureka/apps/%7bappID%7d)

提交方式: POST

输入参数：JSON数据,具体参数如下表所示

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **参数名称** | **父节点** | **参数类型** | **中文名称** | **限定** | **说明** |
| hostName | instance | String | 实例HOST | M | 服务实例的HOST名称 |
| app | instance | String | 服务名称 | M | 全局唯一的服务名称,与注册接口中的appID保持一致 |
| ipAddr | instance | String | 实例IP地址 | M | 服务实例运行主机的IP地址 |
| port | instance | Port | 实例端口 | M | 实例提供服务的端口 |
| homePageUrl | instance | String | 实例的主页URL | O | 实例主页的URL地址 |
| statusPageUrl | instance | String | 信息查询接口 | O | 用于在注册中心查询展示服务基本信息 |
| instanceId | instance | String | 实例ID | M | 实例的全局唯一的标识符 |
| dataCenterInfo | instance | DataCenterInfo | 数据中心 | M | 数据中心 |
| vipAddress | instance | String | 用户定义虚拟主机名称 | M | 需要与app的值保持一致,配置中心根据该值查询实例列表 |

Port:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **参数名称** | **父节点** | **参数类型** | **中文名称** | **限定** | **说明** |
| $ | port | Integer | 实例端口 | M | 实例端口 |
| @enabled | port | boolean | 是否开启 | M | 请设置为true |

DataCenterInfo:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **参数名称** | **父节点** | **参数类型** | **中文名称** | **限定** | **说明** |
| @class | dataCenterInfo | String | 数据中心处理类 | M | 固定值，请设置为com.netflix.appinfo.  InstanceInfo$DefaultDataCenterInfo |
| name | dataCenterInfo | String | 数据中心名称 | M | 如果服务部署在AWS上，该值为 Amazon,其他，该值为 MyOwn |

响应：

HTTP CODE：204 success

JSON报文举例：

{

"instance": {

"instanceId": "{hostName}:{app}:{port}",

"hostName": "{hostName}",

"app": "{app}",

"ipAddr": "{ipAddr}",

"vipAddress": "{app}",

"port": {

"$": "{port}",

"@enabled": "true"

},

"statusPageUrl": "",

"homePageUrl": "",

"dataCenterInfo": {

"@class": "com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo",

"name": "MyOwn"

}

}

}

##### 基于spring cloud的服务

服务会通过注册中心客户端自动注册自己，只需引入相应的JAR包，做必要的配置即可，步骤如下：

a.引入JAR包，mave坐标如下 :

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

b.启用注册中心客户端： 在启动类上添加 @EnableDiscoveryClient，举例如下：

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

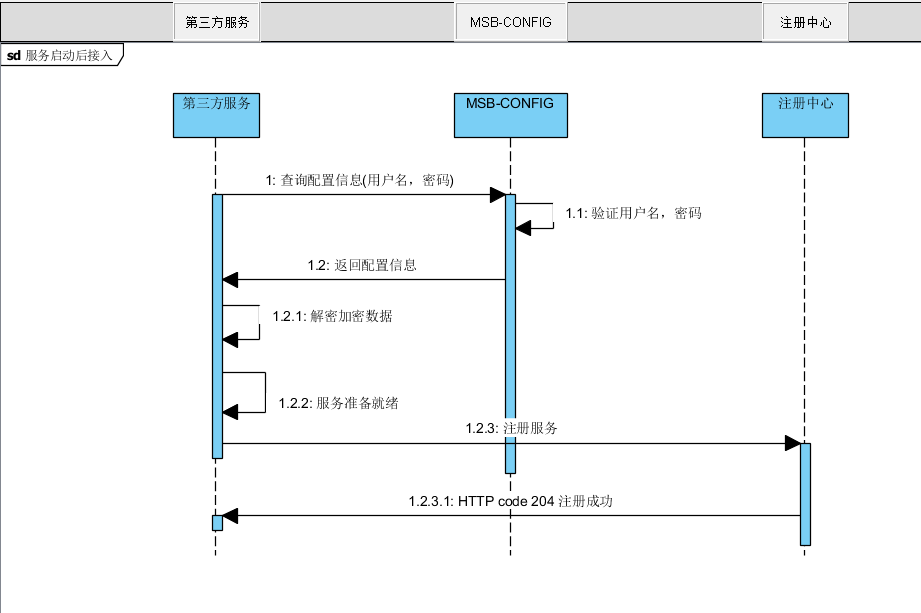
c. 在配置中心配置注册中心地址，属性名称为 eureka.client.serviceUrl.applicationZone，例如：

eureka.client.serviceUrl.applicationZone： <http://188.0.96.27:30113/eureka>

##### 调用时机

1. 基于spring cloud架构的的服务，无需手动调用，平台集成
2. 非spring cloud的服务，需要在以下两种情况下，调用注册接口，注册自己：
3. **当服务启动后，一切准备就绪，可以对外提供服务了，此时调用注册接口,上送注册信息**
4. **服务注册成功后，在后面的运行过程中，如果连续三次上送心跳信息失败，此时应该重新调用注册接口，上送注册信息**

##### 服务注册时序图



#### 第二步，发送心跳续租（service->MSB）

##### 3.1 非spring cloud架构的服务

**每隔30秒上送一条心跳信息到MSB，MSB系统据此判断服务是否健康**。关于30秒这个数字的解释：注册中心会检查服务的心跳信息，如果超过90秒没有收到心跳，会从注册表中删除该服务实例，所以心跳间隔必须小于90秒，30秒是推荐的配置

测试环境服务心跳上送接口：

[http://188.0.96.27:30113/eureka/apps/{appID}/{instanceID}](http://188.0.96.27:30113/eureka/apps/%7bappID%7d/%7binstanceID%7d)

提交方式：PUT

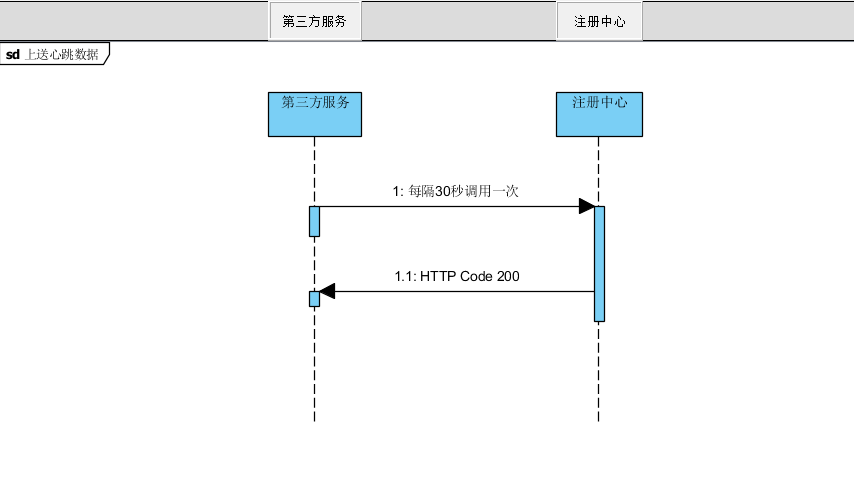
输入参数：无

响应：

HTTP CODE: 200 success; 404 instanceID 不存在

##### spring cloud架构的服务，注册中心客户端已实现

##### 发送心跳数据时序图



### 服务配置动态刷新接入

#### 概述

接入MSB平台的服务，需要实现刷新接口和调用数据获取接口，这样就可以融入MSB系统的配置集中化管理以及数据的动态刷新。

#### 2. 第一步，提供数据刷新接口（MSB->service）

##### 2.1 非spring cloud架构的服务

接入服务需要提供一个数据刷新接口，MSB系统的数据刷新操作会调用该接口，将数据发生变化这一事件通过给接入服务。**该接口由接入服务提供**，以下的描述中，myservice是接入服务的根地址

数据刷新接口URL：<http://myservice/refresh>

方法：POST

参数： 无

##### 2.2 spring cloud架构的服务

只需引入相应JAR包

1. 引入JAR包：

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-actuator</**artifactId**>  
</**dependency**>

#### 第二步，配置查询接口（service->MSB）

接入服务在收到数据变化通知后，主动调用该接口从MSB获取最新的配置信息。下面接口中的profile指的是服务想获取的变量的环境，比如想获取测试环境的变量，这个值就是test

##### 非spring cloud 架构的服务

数据查询接口URL: [http://188.0.96.27:30117/msbconfig/{appID}-{profile}.properties](http://188.0.96.27:30117/msbconfig/%7bappID%7d-%7bprofile%7d.properties)

方法：GET

参数： 无

响应：

Property格式，

响应Porperty报文举例：

测试URL：http://localhost:8888/msb-marvel-local.properties

eureka.client.serviceUrl.applicationZone: http://188.0.96.27:30113/eureka

eureka.instance.prefer-ip-address: true

management.security.enabled: false

spring.datasource.driver-class-name: com.mysql.jdbc.Driver

spring.datasource.password: 123456

spring.datasource.tomcat.max-active: 2

spring.datasource.url: jdbc:mysql://localhost:3306/msb?useUnicode=true&characterEncoding=utf8

spring.datasource.username: root

test.test.test: 9999

##### spring cloud 架构的服务

spring cloud config客户端会在收到刷新通知后自动获取

##### 关于刷新结果通知

通常一个服务有多个实例，刷新操作需要通知到每一个实例，MSB平台认为成功通知到所有实例才算刷新成功，页面才会得到 code 为0的成功通知；

如果有一个实例通知失败，将停止通知剩余的实例，并返回结果刷新失败，同时返回刷新成功的实例列表，失败的实例列表，未刷新的实例列表；

部分刷新成功会造成一个服务的多个实例之间的状态不一致行为，请在刷新失败时，重新进行刷新操作，直到刷新成功

##### 推荐方案

1. Spring cloud 架构服务已经实现了动态刷新功能，并且，自动刷新从更新通知到数据获取是一个同步操作，我们可以根据刷新接口的返回结果，明确断言是否刷新成功（HTTP code 200 表示成功）
2. 其他架构服务，我们推荐使用同步方式实现整个刷新过程，这样MSB管理平台就可以根据刷新接口返回的结果判断数据是否推送成功，例如：

@PostMapping(**"/refresh"**)  
**public** Map refresh(){

//1.同步调用配置数据获取接口

//2.根据配置数据获取接口的返回码判断此次刷新是否成功

//3.成功返回 HTTP code 200

}

#### 在服务中使用获取的动态属性

应用程序根据需求，自行处理

Spring cloud 服务动态刷新改造示例：

普通对象属性动态刷新，在类上添加 @RefreshScop注解，例如：

@Component  
@RefreshScope  
**public class** RedisConfig {  
 @Value(**"${spring.redis.host:localhost}"**)  
 **private** String **redisHost**;  
 @Value(**"${spring.redis.port:6379}"**)  
 **private** Integer **redisPort**;  
 @Value(**"${spring.redis.password:}"**)  
 **private** String **password**;  
  
 **public** String getRedisHost() {  
 **return redisHost**;  
 }  
  
 **public void** setRedisHost(String redisHost) {  
 **this**.**redisHost** = redisHost;  
 }  
  
 **public** Integer getRedisPort() {  
 **return redisPort**;  
 }  
  
 **public void** setRedisPort(Integer redisPort) {  
 **this**.**redisPort** = redisPort;  
 }  
  
 **public** String getPassword() {  
 **return password**;  
 }  
  
 **public void** setPassword(String password) {  
 **this**.**password** = password;  
 }  
}

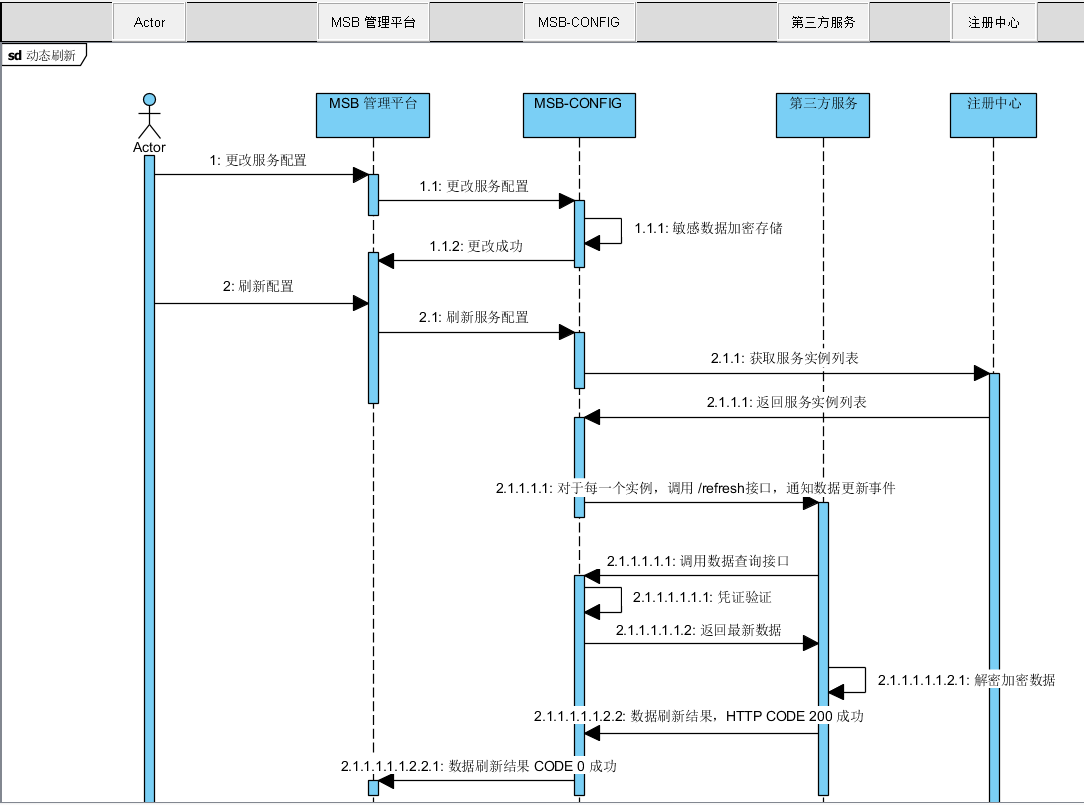
关系数据库数据源动态刷新：

@Bean  
@RefreshScope  
@ConfigurationProperties(prefix = **"spring.datasource"**)  
**public** DataSource dataSource() {  
 **return** DataSourceBuilder.*create*().build();  
}

Redis 链接动态刷新：

@Bean  
@RefreshScope  
@ConfigurationProperties(prefix = **"spring.redis"**)  
**public** RedisConnectionFactory dynamicConfiguration() {  
 JedisConnectionFactory jedisConnectionFactory = **new** JedisConnectionFactory();  
 jedisConnectionFactory.setPort(**redisConfig**.getRedisPort());  
 jedisConnectionFactory.setHostName(**redisConfig**.getRedisHost());  
 jedisConnectionFactory.setPassword(**redisConfig**.getPassword());  
 jedisConnectionFactory.afterPropertiesSet();  
 **return** jedisConnectionFactory;  
}

#### 动态刷新时序图



## 安全

### 敏感数据加密

配置中心启用加密模式，采用对称加密，禁用自动解密功能，由客户端解密。

### 服务配置属性获取权限控制

1. 为每个接入服务分配独立的密钥
2. 服务目录的每一次同步，都需要检查是否有密钥，没有则生成
3. 在配置中心添加数据获取权限验证，只有服务名和密钥凭证相匹配，才允许获取相应的数据

### 加密算法及JAVA举例

#### 配置中心加密算法描述

1.采用AES加密算法的CBC工作模式

2.对称密钥的产生：

配置服务器和各客户端服务共用一个口令，口令越长越安全

基于口令使用PBE算法生成对称密钥：a.设置 salt（必须为16进制字符串，为 deadbeef） b.将口令和salt杂糅，然后经过1024次信息摘要算法迭代，生成256位的密钥 c.采用的信息摘要算法是 PBKDF2WithHmacSHA1

3.加密过程：

1.随机产生一个16字节数组，生成一个IvParameterSpec向量对象

2.填充模式 AES/CBC/PKCS5Padding

3.加密。加密时会首先使用第一步生成的随机向量与明文异或运算，然后再加密，这样每次产生的密文都不一样，增加安全性

4.将16字节的随机向量与密文拼接一起返回，作为最终的密文结果

5.转换成16进制字符串

#### 加密算法 JAVA代码举例

public static String encode(String content) throws Exception {

//口令

String password="ABCDEFGHABCDEF";

//salt串

String salt="deadbeef";

//1.采用PEB算法生成密钥

PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(), hexStringToByte(salt), 1024, 256);

SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");

SecretKey secretKey =factory.generateSecret(keySpec);

//转换成AES专用密钥

SecretKeySpec secretKeySpec = new SecretKeySpec(secretKey.getEncoded(), "AES");

//2.使用PKCS5Padding填充模式创建加密器

Cipher cipher =Cipher.getInstance("AES/CBC/PKCS5Padding");

//3.产生一个16字节的随机字节数组，然后生成IV对象

SecureRandom random = new SecureRandom();

byte[] bytes = new byte[16];

random.nextBytes(bytes);

IvParameterSpec iv =new IvParameterSpec(bytes);

//4.初始化加密模式的密码器

cipher.init(Cipher.ENCRYPT\_MODE, secretKeySpec,iv);

//5.加密

byte[] encontent =cipher.doFinal(content.getBytes("utf-8"));

//6.将随机字节数组与密文拼接

byte[] result =concatenate(new byte[][]{bytes,encontent});

//7.转换为16进制字符串返回（使用小写字母）

return HexUtils.toHexString(result);

}

/\*\*

\* 将多维数组转换成一维数组

\* @param arrays

\* @return

\*/

public static byte[] concatenate(byte[]... arrays) {

int length = 0;

byte[][] var2 = arrays;

int destPos = arrays.length;

for(int var4 = 0; var4 < destPos; ++var4) {

byte[] array = var2[var4];

length += array.length;

}

byte[] newArray = new byte[length];

destPos = 0;

byte[][] var9 = arrays;

int var10 = arrays.length;

for(int var6 = 0; var6 < var10; ++var6) {

byte[] array = var9[var6];

System.arraycopy(array, 0, newArray, destPos, array.length);

destPos += array.length;

}

return newArray;

}

#### 客户端解密数据算法描述

1.采用AES加密算法的CBC工作模式

2.对称密钥的产生：

配置服务器和各客户端服务共用一个口令，口令越长越安全

基于口令使用PBE算法生成对称密钥：a.设置 salt（必须为16进制字符串，默认为 deadbeef） b.将口令和salt杂糅，然后经过1024次信息摘要算法迭代，生成256位的密钥 c.采用的信息摘要算法是 PBKDF2WithHmacSHA1

3.解密过程：

1.截取密文数组的前16个字节作为一个新的数组，用该新的数组生成一个IvParameterSpec向量对象

2.填充模式 AES/CBC/PKCS5Padding

3.解密。

4.将字节数组转换为字符串

#### 解密算法JAVA举例

public static String decode(String content) throws Exception {

//口令

String password="ABCDEFGHABCDEF";

//salt串

String salt="deadbeef";

//将密文转换成字节数组

byte[] encodingText = hexStringToByte(content);

//1.采用PEB算法生成密钥

PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(), hexStringToByte(salt), 1024, 256);

SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");

SecretKey secretKey =factory.generateSecret(keySpec);

//转换成AES专用密钥

SecretKeySpec secretKeySpec = new SecretKeySpec(secretKey.getEncoded(), "AES");

//2.使用PKCS5Padding填充模式创建加密器

Cipher cipher =Cipher.getInstance("AES/CBC/PKCS5Padding");

//3.截取密文数组的前16个字节作为一个新的数组,用于生成iv向量；真正的密文是除去前16个字节的剩余字节

byte[] ivarray = new byte[16];

byte[] encodingarray=new byte[encodingText.length-16];

System.arraycopy(encodingText, 0, ivarray, 0, 16);

System.arraycopy(encodingText, 16, encodingarray, 0, encodingText.length-16);

//4.生成IV向量

IvParameterSpec iv =new IvParameterSpec(ivarray);

//5.初始化解密模式的密码器

cipher.init(Cipher.DECRYPT\_MODE, secretKeySpec,iv);

//5.解密

byte[] decontent =cipher.doFinal(encodingarray);

//6.转换成字符串

return new String(decontent);

}

/\*\*

\* 把16进制字符串转换成字节数组

\* @param

\* @return byte[]

\*/

public static byte[] hexStringToByte(String hex) {

int len = (hex.length() / 2);

byte[] result = new byte[len];

char[] achar = hex.toCharArray();

for (int i = 0; i < len; i++) {

int pos = i \* 2;

result[i] = (byte) (toByte(achar[pos]) << 4 | toByte(achar[pos + 1]));

}

return result;

}

private static int toByte(char c) {

byte b = (byte) "0123456789abcdef".indexOf(c);

return b;

}