

## 23 Spring Cloud Config 中配置文件的加密与解密

更新时间：2019-07-10 14:04:40



每个人都是自己命运的主宰。

——斯蒂尔斯

上篇文章和大家聊了 Spring Cloud Config 分布式配置中心的基本用法，相信大家对 Spring Cloud Config 已经有了一个基本的认识。可能有读者也发现问题了，原本在非分布式环境下，一些由运维工程师掌握的敏感信息现在不得不写在配置文件中了，这样网传的程序员删库跑路的段子可能就成真了！但是在微服务中，我们又不大可能让运维工程师手动去维护这些信息，因为工作量太大了，那么一个好的办法，就是对这些配置信息进行加密，这也是我们本文要说的重点。

### 常见加密方案

说到加密，需要先和大家来捋一捋一些常见的加密策略，首先，从整体上来说，加密分为两大类：

- 不可逆加密
- 可逆加密

不可逆加密就是大家熟知的在 Spring Security 或者 Shiro 这一类安全管理框架中我们对密码加密经常采取的方案。这种加密算法的特点就是不可逆，即理论上无法使用加密后的密文推算出明文，常见的算法如 MD5 消息摘要算法以及 SHA 安全散列算法，SHA 又分为不同版本，这种不可逆加密相信大家在密码加密中经常见到，就不需要松哥多说了。

可逆算法看名字就知道，这种算法是可以根据密文推断出明文的，可逆算法又分为两大类：

- 对称加密
- 非对称加密

对称加密是指加密的密钥和解密的密钥一致，例如 A 和 B 之间要通信，为了防止别人偷听，两个人提前约定好一个密钥。每次发消息时，A 使用这个密钥对要发送的消息进行加密，B 收到消息后则使用相同的密钥对消息进行解密。这是对称加密，常见的算法有 DES、3DES、AES 等。

对称加密在一些场景下并不适用，特别是在一些一对多的通信场景下，于是就有了非对称加密，非对称加密就是加密的密钥和解密的密钥不是同一个，加密的密钥叫做公钥，这个可以公开告诉任何人，解密的密钥叫做私钥，只有自己知道。非对称加密不仅可以用来做加密，也可以用来做签名，使用场景还是非常多的，常见的加密算法是 **RSA**。

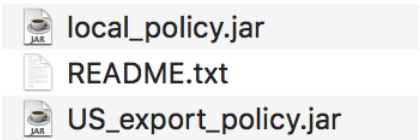
配置文件加密肯定是可逆加密，不然给我一个加密后的字符串，我拿着也没用，还是没法使用。可逆算法中的对称加密和非对称加密在 **Spring Cloud Config** 中都得到支持，下面我们就分别来看。

## 对称加密

Java 中提供了一套用于实现加密、密钥生成等功能的包 **JCE**(Java Cryptography Extension)，这些包提供了对称、非对称、块和流密码的加密支持，但是默认的 **JCE** 是一个有限长度的 **JCE**，我们需要到 **Oracle** 官网去下载一个不限长度的 **JCE**：

[不限长度JCE下载地址](#)

下载完成后，将下载文件解压，解压后的文件包含如下三个文件：



将 **local\_policy.jar** 和 **US\_export\_policy.jar** 两个文件拷贝到 **JDK** 的安装目录下，具体位置是 **%JAVA\_HOME%\jre\lib\security**，如果该目录下有同名文件，则直接覆盖即可。

这是我们的一点准备工作。

接下来步骤和上文一样，我们创建一个 **CloudConfig** 的父工程，在这个工程中创建 **config\_server** 和 **config\_client**，同时继续使用上文创建的仓库 **configRepo**，这一系列操作和上文一模一样，读者也可以不用创建新项目，直接在上文的基础上进行修改，这里我就不赘述了。

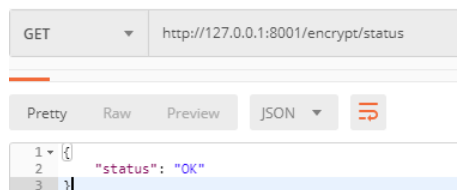
当 **config\_server** 和 **config\_client** 都准备好之后，在 **config\_server** 的 **bootstrap.properties** 文件中，添加如下一行配置：

```
encrypt.key=123
```

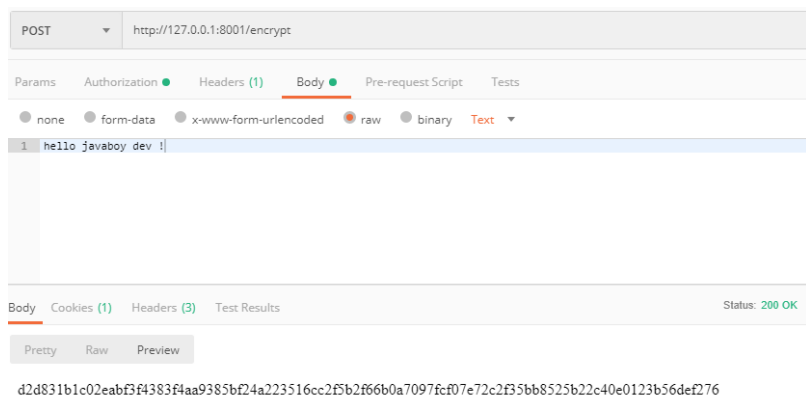
这就是我们配置的加密密钥了，配置完成后启动 **config\_server**。如果你使用的是 **IntelliJ IDEA**，**config\_server** 启动成功之后，从控制台的 **Mappings** 中就能看到这里帮我们自动加入了好几个接口：

Path	
{GET /actuator/info, produces [application/vnd.spring-boot.actuator.v2+json]}	Actuator web endpoint 'info'
{GET /encrypt/status}	EncryptionController#status
{GET /key/{name}/{profiles}}	EncryptionController#getPublicKey
{GET /key}	EncryptionController#getPublicKey
{GET /{label}/{name}-{profiles}.json}	EnvironmentController#labelledJsonProperties
{GET /{label}/{name}-{profiles}.properties}	EnvironmentController#labelledProperties
{GET /{name}-{profiles}.json}	EnvironmentController#jsonProperties
{GET /{name}-{profiles}.properties}	EnvironmentController#properties
{GET /{name}/{profiles}.{[-]*}}	EnvironmentController#defaultLabel
{GET /{name}/{profiles}/{label}.{[-]*}}	EnvironmentController#labelled
{GET /{name}/{profile}/{label}/{label}.properties}	ResourceController#retrieve
{GET /{name}/{profile}/{label}/{label}.properties}	ResourceController#retrieve
{GET /{label}/{name}-{profiles}.yaml, /{label}/{name}-{profiles}.yml}	EnvironmentController#labelledYaml
{GET /{name}-{profiles}.yaml, /{name}-{profiles}.yml}	EnvironmentController#yaml
{POST /decrypt/{name}/{profiles}}	EncryptionController#decrypt
{POST /decrypt}	EncryptionController#decrypt
{POST /encrypt/{name}/{profiles}}	EncryptionController#encrypt
{POST /encrypt}	EncryptionController#encrypt

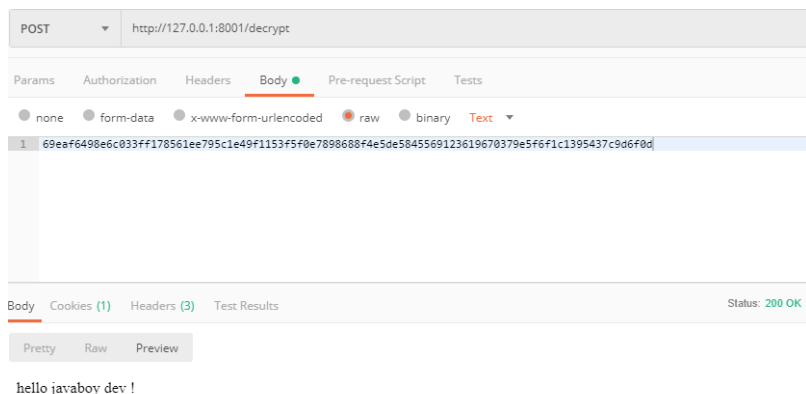
这些接口中就有加解密的接口，也有查看加解密接口状态的接口，首先我们来查看接口状态，访问路径是：<http://127.0.0.1:8001/encrypt/status>，查看其加密模块是否正常运行：



看到 **status** 的值为 **ok** 表示这个模块正常运行，接下来调用 <http://127.0.0.1:8001/encrypt> 接口发送一个 **POST** 请求，来给一段文本进行加密：

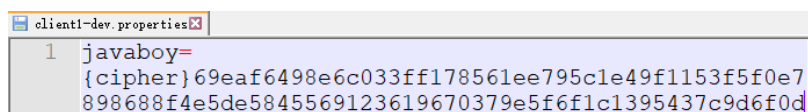


注意，请求参数就是要加密的字符串，请求的响应结果则是加密之后的文本。这个是加密接口，也有解密接口，解密接口则是 **/decrypt**，例如对刚刚加密的这个字符串进行解密：

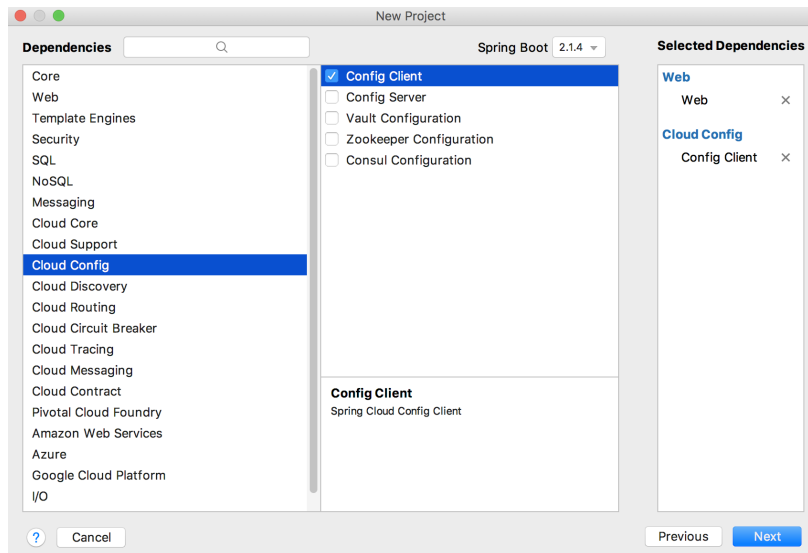


同样是 **POST** 请求，请求参数是加密之后的文本，响应结果则是解密之后的明文文本。

在确保了这两个接口没问题的情况下，接下来修改本地仓库中的 **client1-dev.properties** 文件，将加密字符串拷贝进来，如下：



注意加密字符串需要添加一个前缀 **{cipher}**，有了这个前缀，当 **config\_server** 加载到该文本时，就会对这个文本进行解密，再返回给 **config\_client**。修改完 **client1-dev.properties** 文件后，将之提交到 **GitHub** 上，然后重启 **config\_server**，也启动 **config\_client**，访问 **config\_client** 的 **/hello** 接口，如下：



如果 `config_client` 加载的是其它配置文件的话，其它文件因为没有 `{cipher}` 前缀，所以就不会对相应的文本进行解密。

好了，这个是使用对称加密的方式来加密配置文件。

## 非对称加密

当然我们也可以使用非对称加密的方式来对配置文件进行加密，非对称加密要求我们先有一个密钥，密钥的生成我们可以使用 JDK 中自带的 `keytool`。`keytool` 是一个 Java 自带的数字证书管理工具，`keytool` 将密钥（`key`）和证书（`certificates`）存在一个称为 `keystore` 的文件中。具体操作步骤如下：

首先打开命令行窗口，输入如下命令：

```
keytool -genkeypair -alias config-server -keyalg RSA -keystore D:\config-server.keystore
```

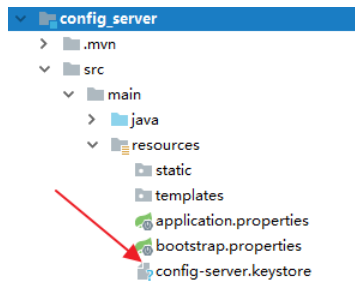
参数解释：

- `-genkeypair` 表示生成密钥对
- `-alias` 表示 `keystore` 关联的别名
- `-keyalg` 表示指定密钥生成的算法
- `-keystore` 指定密钥库的位置和名称

以上命令在执行过程中，还有如下一些参数需要大家设置，如图：

```
C:\Program Files\Java\jdk1.8.0_171\bin> keytool -genkeypair -alias config-server -keyalg RSA -keystore D:\config-server.keystore
输入密钥库口令:
再次输入新口令:
您的名字与姓氏是什么?
[Unknown]:
您的组织单位名称是什么?
[Unknown]:
您的组织名称是什么?
[Unknown]:
您所在的城市或区域名称是什么?
[Unknown]:
您所在的省/市/自治区名称是什么?
[Unknown]:
该单位的双字母国家/地区代码是什么?
[Unknown]:
CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown是否正确?
[否]: 是
输入 <config-server> 的密钥口令:
(如果和密钥库口令相同, 按回车):
```

执行过程中，密钥库口令需要牢记，这个我们在后面还会用到。其它的信息可以输入也可以直接回车表示 `Unknown`，自己做练习无所谓，实际开发中还是建议如实填写。好了，这个命令执行完成后，在 `D` 盘下就会生成一个名为 `config-server.keystore` 的文件，将这个文件直接拷贝到 `config_server` 项目的 `classpath` 下，如下：



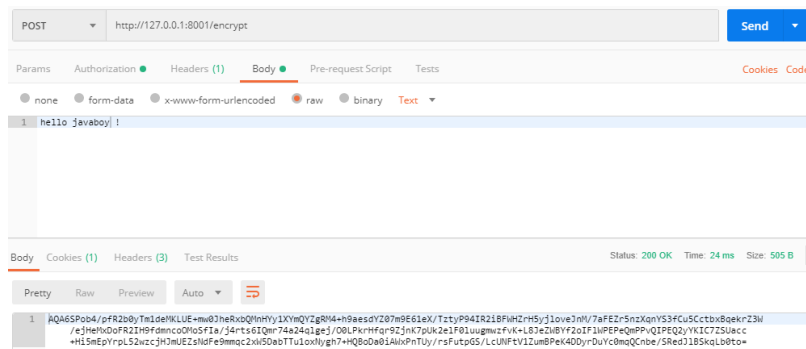
然后在 `config_server` 的 `bootstrap.properties` 文件中，添加如下配置（注意注释掉对称加密时的那一行配置）：

```
encrypt.key-store.location=config-server.keystore
encrypt.key-store.alias=config-server
encrypt.key-store.password=123456
encrypt.key-store.secret=123456
```

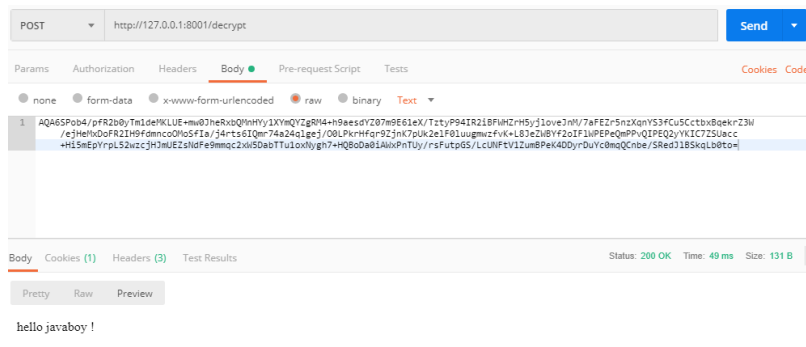
这四行配置根据生成过程的参数来配置即可。

配置完成后，重新启动 `config_server`。启动成功后，加密解密的链接地址和对称加密都是一样的，因此，我们可以继续使用 `http://127.0.0.1:8001/encrypt` 对文本进行加密，使用 `http://127.0.0.1:8001/decrypt` 对文本进行解密，如下图：

加密请求：



解密请求：



两个请求接口都没问题，接下来，我们依然是修改 `client1-dev.properties` 文件，将加密字符串放进去，如下：

```
client1-dev.properties
1 javaboy=
  {cipher}AQA6SPob4/pfR2b0yTmldeMKLUE+mw0JheRxbQMnHYy1
  XYmQYZgRM4+h9aesdYZ07m9E6leX/TztyP94IR2iBFWHzrH5yjl0
  veJnM/7aFEZr5nzXqnYS3fCu5CctbxBqekrZ3W/ejHeMxDoFR2IH
  9fdmncoOMoSfIa/j4rts6IQmr74a24qlgej/O0LPkrHfqr9ZjnK7
  pUk2elF0luugmwzfvK+L8JeZWBYf2oIFlWPEPeQmPPvQIPEQ2yYK
  IC7ZSUacc+Hi5mEpYrpL52wzcjHJmUEZsNdFe9mmqc2xW5DabTTu
  1oxNygh7+HQBoDa0iAWxPnTUy/rsFutpGS/LcUNFtVlZumBPek4D
  DyrDuYc0mqQCnbe/SRedJlBSkqLb0to=
```

然后将本地仓库中的数据提交到远程仓库中。提交成功后，重启 `config_client`，然后访问相关接口，我们发现数据已经发生了变化了。

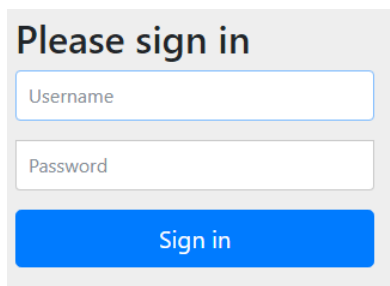
好了，这是和大伙介绍的两种配置加密方式。

## 安全管理

目前的 `config_server` 存在很大的安全隐患，因为所有的数据都可以不经过 `config_client` 直接访问。出于数据安全考虑，我们要给 `config_server` 中的接口加密。在 `Spring Boot` 项目中，项目加密方案当然首选 `Spring Security`，使用 `Spring Security` 也很简单，只需要在 `config_server` 项目中添加如下依赖即可：

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

添加完成之后，重启 `config_server` 项目，然后浏览器中输入 `http://localhost:8001/client1/dev/master`，访问结果如下：



可以看到，此时接口已经被保护起来了，必须要登录之后才能访问，默认的登录用户名是 `user`，登录密码在 `config_server` 的启动控制台上，如下：

```
2019-05-08 18:44:59.719 INFO 18104 --- [main] .s.s.UserDetailsServiceAutoConfiguration :
Using generated security password: 35478445-aaa9-4d14-8bdc-0d495f5d8e7c
```

这是默认的登录密码，这个登录密码是项目启动时随机生成的，每次启动都不一样，如果想要使用固定的用户名密码，则可以直接在 `config_server` 的 `bootstrap.properties` 配置文件中添加如下配置：

```
spring.security.user.name=javaboy
spring.security.user.password=123
```

配置完成后，再次启动 `config_server`，此时，控制台就不会有默认的随机密码输出了，用户需要使用 `javaboy/123` 来登录系统，登录之后，就可以访问 `config_server` 中的接口了。

当 `config_server` 中添加了接口之后，此时如果 `config_client` 不做任何额外的配置，直接启动，就会抛出如下错误：

```
main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://localhost:8001/
main] c.c.c.ConfigServicePropertySourceLocator : Could not locate PropertySource: 401 null
main] c.j.c.ConfigClientApplication : No active profile set, falling back to default profiles: default
```

解决办法也很简单，直接在 `config_client` 的 `bootstrap.properties` 文件中添加如下配置：

```
spring.cloud.config.username=javaboy
spring.cloud.config.password=123
```

配置完成后，`config_client` 就可以像之前一样访问 `config_server` 了。

## 小结

本文主要和读者聊了两个话题，文件加解密和 `config_server` 的安全管理。虽然是两个话题，其实是为了解决一个问题，就是配置文件的安全问题，这两个技能点在分布式配置中心 `Spring Cloud Config` 中也算是刚需了，基本上都会用到，大家一定要掌握。

## 精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示



目前暂无任何讨论