

1.首先我们需要一个远程的Git仓库，自己学习可以直接用GitHub，在在实际生产环境中，需要自己搭建一个Git服务器，远程Git仓库的作用主要是用来保存我们的配置文件  
2.除了远程Git仓库之外，我们还需要一个本地Git仓库，每当Config Server访问远程Git仓库时，都会保存一份到本地，这样当远程仓库无法连接时，就直接使用本地存储的配置信息  
3.至于微服务A、微服务B则是我们具体的应用，这些应用在启动的时候会从Config Server中来加载相应的配置信息  
4.当微服务A/B尝试去从Config Server中加载配置信息的时候，Config Server会先通过git clone命令克隆一份配置文件保存到本地  
5.由于配置文件是存储在Git仓库中，所以配置文件天然的具备版本管理功能，Git中的Hook功能可以实时监控配置文件的修改

**Git URI中的占位符**

灵活的使用URI占位符，可以有效的减少我们的工作量。考虑这样一个问题，我有ServerA、ServerB两个服务，两个服务对应的配置文件的存储地址分别位于https://github.com/lenve/scConfig/sa和https://github.com/lenve/scConfig/sb，但是我的Config Server只有一个，那么当我的ServerA和ServerB连接上Config Server时，Config Server怎么知道去哪个地址下拿配置文件?这个时候就涉及到占位符的使用。  
在上篇文章中我们已经了解了Spring Cloud Config中的三种占位符，分别是{application}、{profile}和{label}，这些占位符除了用来标识配置文件的规则，还可以用在Config Server中对Git仓库的URI配置，用在URI配置中时，这三个占位符的含义分别如下所示：

1.{application}映射到客户端的 spring.application.name  
2.{profile}映射到客户端上的 spring.profiles.active  
3.{label}这是一个服务器端功能，标记”版本”的配置文件集

此时，假设我不同环境下的配置文件分别放在下面这些目录下：

https://github.com/lenve/scConfig/app/dev  
https://github.com/lenve/scConfig/app/prod  
https://github.com/lenve/scConfig/app/test

那么我的客户端文件这样配置：

spring.application.name=app

# dev根据具体情况来修改

spring.cloud.config.profile=dev

spring.cloud.config.label=master

spring.cloud.config.uri=http://localhost:2007/

server.port=2008

然后Config Server按下面这种方式配置即可：

spring.cloud.config.server.git.uri=https://github.com/lenve/scConfig.git

spring.cloud.config.server.git.search-paths={application}/{profile}

当然这种存储规划不一定最佳，这里只是给小伙伴们演示占位符的用法。

默认情况下，Config Server 克隆下来的文件保存在C:\Users\<当前用户>\AppData\Local\Temp目录下，我们可以通过如下配置来修改：

spring.cloud.config.server.git.basedir=E:\\111\\

## 健康监测

默认情况下Spring Cloud Config会为配置中心服务端创建一个健康监测器，该检测器默认情况下是访问的仓库文件是{application}为app的配置文件，如果仓库中不存在这个文件，健康显示器就会显示仓库无法连接，此时我们有两种解决方案：1.仓库中添加相应的配置文件；2.重新指定检测的配置，重新指定方式如下：spring.cloud.config.server.health.repositories.check.name=app

spring.cloud.config.server.health.repositories.check.label=master

spring.cloud.config.server.health.repositories.check.profiles=dev

此时，系统回去访问http://localhost:2007/app/dev/master地址，如果能够访问到，则显示仓库已连接

## 安全保护

开发环境中我们的配置中心肯定是不能随随便便被人访问的，我们可以加上适当的保护机制，由于微服务是构建在Spring Boot之上，所以整合Spring Security是最方便的方式。

首先添加依赖：

<dependency>

   <groupId>org.springframework.boot</groupId>

   <artifactId>spring-boot-starter-security</artifactId>

</dependency>

然后在application.properties中配置用户名密码：

security.user.name=sang

security.user.password=123

最后在客户端上**bootstrap.properties**配置用户名和密码即可，如下：

spring.cloud.config.username=sang

spring.cloud.config.password=123

## 加密解密

## 默认情况下我们的JRE中自带了JCE(Java Cryptography Extension)，但是默认是一个有限长度的版本，我们这里需要一个不限长度的JCE，这个JCE我们可以直接百度然后在Oracle官网下载，下载之后解压，我们可以看到如下三个文件：

## 

## 我们需要将这里的两个jar包拷贝到我们的jdk安装目录下，我的是%JAVA\_HOME%\jre\lib\security，覆盖该目录下原有的文件。

## 对称加解密

## 对称加解密比较简单，直接配置密钥就可以了，在我们前文创建出来的config-server中配置密钥，但是注意这个密钥需要配置在bootstrap.properties中，**另外这里还有非常重要一点：Spring Cloud的Dalston.SR3和Dalston.SR2版本在这个问题上是有BUG的，如果用这两个版本在这里测试会没有效果，应该避开使用这两个版本，我这里使用的是Dalston.SR4版本**，配置如下：

## encrypt.key=sang

## 配置完成之后，启动我们的config-server工程，然后访问如下地址<http://localhost:2007/encrypt/status>

## 此时我们就可以通过第三方工具如POSTMAN、RestClient等来访问/encrypt和/decrypt接口，比如说我要给dev这个字符加密，方式如下(我这里以POSTMAN为例，注意是POST请求)：

## 

## 解密方式如下：

## OK，拿到加密的字符串之后，我们就可以在配置文件中使用了，还是我们昨天的配置文件，这次我这样来写：

## 小伙伴们注意，配置文件的值如果是以{cipher}开头，表示该值是一个加密字符，配置中心config-server在获取到这个值之后会先对值进行解密，解密之后才会返回给客户端使用。

## 非对称加解密

## 上文我们使用了对称加解密的方式来确保配置文件的安全性，如果使用非对称加解密的方式，我们的安全性将会得到进一步的提高。使用非对称加密的话需要我们先生成密钥对，生成密钥对可以直接使用jdk中自带的keytool工具，方式如下：

## keytool -genkeypair -alias config-server -keyalg RSA -keystore config-server.keystore

## 执行效果如图：

## 执行成功之后，会在命令执行目录下生成一个名为config-server.keystore的文件，将该文件拷贝到config-server的src\main\resources目录下，然后做如下配置：

encrypt.key-store.location=config-server.keystore

encrypt.key-store.alias=config-server

encrypt.key-store.password=111111

## encrypt.key-store.secret=111111

## OK，如此之后我们的非对称加密就配置好了，测试方式和对称加密的测试方式一致