[**SpringCloud之Config配置中心+BUS消息总线原理及其配置**](https://www.cnblogs.com/jing99/p/11720834.html)

**一、配置中心作用**

　　在常规的开发中，每个微服务都包含代码和配置。其配置包含服务配置、各类开关和业务配置。如果系统结构中的微服务节点较少，那么常规的代码+配置的开发方式足以解决问题。当系统逐步迭代，其微服务会越来越复杂，慢慢演化成网状依赖结构，这个时候常规的代码+配置的开发方式就并不合适了，因为还要考虑整体系统的扩展性、伸缩性和耦合性等。这些问题中，配置的管理也是非常麻烦的。

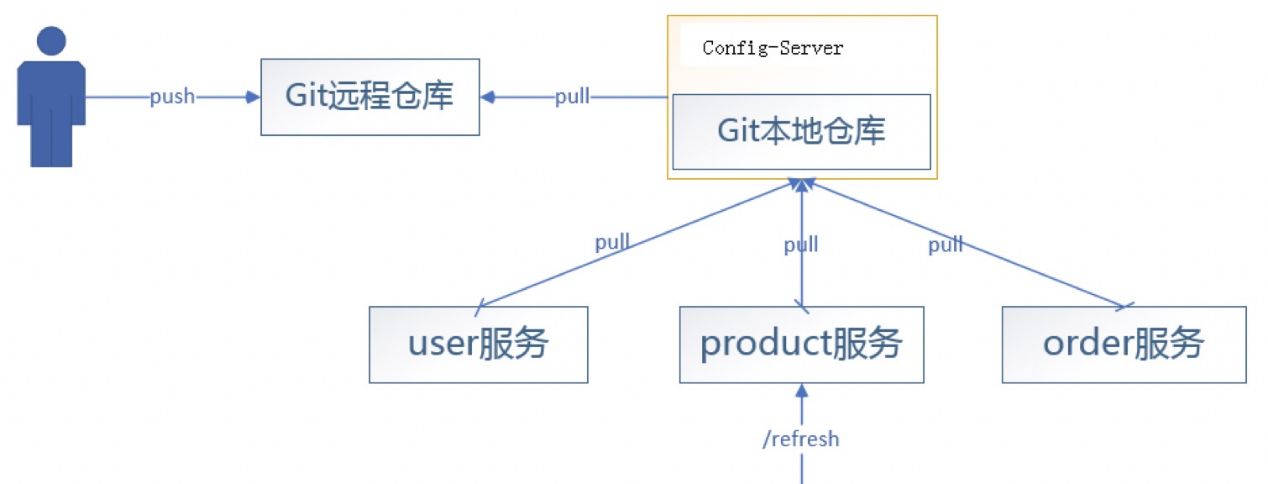
　　如果还是以常规开发形式管理配置，则要承担反复修改编译代码、重启系统、重新打包等风险。所以，一个可以集中管理，带有版本控制的配置中心应运而生。

**spring cloud config就是一个配置中心**。其采用集中式管理每个微服务的配置信息，并**使用GIT等版本仓库统一存储配置内容**，实现版本化管理控制。微服务与配置中心使用rest方式交互来实现可扩展的配置服务。

　　spring cloud config配置中心解决了微服务系统的配置中心化、配置版本控制、平台独立、语言独立等问题，其特性如下：

* 提供服务端和客户端支持(spring cloud config server和spring cloud config client)；
* 集中式管理分布式环境中的配置信息；
* 基于spring环境提供配置管理，与spring系列框架无缝结合；
* 可用于任何语言开发环境；
* 默认基于GIT仓库实现版本控制；

**二、配置中心原理**



**三、配置中心应用**

**3.1 搭建config server**

　　使用spring cloud config配置中心，首先要搭建一个配置中心服务，配置中心服务本身也是一个spring cloud微服务。需要在Eureka注册中心中发布订阅。需要依赖下述资源：

[复制代码](javascript:void(0);)

<dependency>

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

<artifactId>**spring-boot-starter-web**</artifactId>

</dependency>

<!-- spring cloud Eureka Client 启动器 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>**spring-cloud-starter-eureka**</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>**spring-cloud-config-server**</artifactId>

</dependency>

[复制代码](javascript:void(0);)

　　修改全局配置文件：

[复制代码](javascript:void(0);)

spring.application.name=spring-cloud-config-server

server.port=10801

# Git repository configure

# 远程仓库是用于集中管理各微服务配置文件的位置。配置文件需要通过GIT管理工具上传到GIT仓库中。配置文件命名有特有的规则，常见如下：

# applicationName-profile.properties

# applicationName-profile.yml

# applicationName是被管理配置文件的微服务服务命名，即spring.application.name

# profile是被管理配置文件的微服务环境，如开发环境的dev，测试环境的test等。如果未定义profile代表默认环境default。

# 配置Git远程仓库地址

spring.cloud.config.server.git.uri=https://github.com/kosamino/configs

# 如果远程仓库是公开的，不需要提供用户名和密码，如果是私有的，需要提供。

# spring.cloud.config.server.git.username=

# spring.cloud.config.server.git.password=

[复制代码](javascript:void(0);)

　　提供启动类：

[复制代码](javascript:void(0);)

/\*\*

\* @EnableConfigServer - **开启配置中心，代表当前应用是一个配置中心服务端**。

\* 应用会根据全局配置文件访问GIT远程仓库，并将远程仓库中的配置内容下载到本地。

\*/

@SpringBootApplication

**@EnableEurekaClient**

**@EnableConfigServer**

public class ConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

[复制代码](javascript:void(0);)

　　浏览器测试配置文件内容获取，从config server中获取配置文件有固定的访问方式。其格式为：

**http://config-server-ip:port/applicationName/profile[/label]**

* config-server-ip - 配置中心服务端IP
* applicationName - 服务应用名，代表的是配置中心客户端服务命名，就是真正的服务应用中的spring.application.name值。也是配置文件的核心命名。
* profile - 环境命名，代表的是配置中心客户端使用的具体环境，和之前课程中的多环境内容一致。也是配置文件的核心命名。根据applicationName和profile的含义，决定了GIT仓库中保存的配置文件命名是有要求的，**配置文件命名要求是：applicationName[-profile].properties | applicationName[-profile].yml**。提供固定的命名规则是用于配置中心客户端和服务端友好交互定义的。
* label - 是GIT中的分支命名，默认为master，即主干。如果不是主干分支，必须提供分支命名。

　　如：分支名称是abc，applicationName是testApp，环境是dev，那么请求路径是：http://config-server-ip:port/testApp/dev/abc

　　从浏览器获取的配置文件内容结果是JSON格式的字符串。内容大致如下：

[复制代码](javascript:void(0);)

{

"name":"spring-cloud-config-client", // 服务名称

"profiles":["dev"], // 环境名称

"label":null, // 分支名称，默认为master

"version":"a600c404267f5341933ce02b3f62672ce5791b16", // 版本，由spring-cloud-config-server生成管理

"state":null,

// 配置具体内容，从spring-cloud-config-server中获取的配置一定包含default配置内容。  
 // 如:获取环境配置文件路径为http://config-server-ip:port/spring-cloud-config-client/dev，  
 // 那么不仅会返回master分支的spring-cloud-config-client-dev.properties配置文件内容，还会返回master分支的spring-cloud-config-client.properties配置内容。

"propertySources":[

{

"name":"https://gitee.com/kerwin\_kim/test/s**pring-cloud-config-client-dev.properties**", // 配置文件名，也是配置文件的远程GIT访问URL

"source": // 配置文件内容

{

"test.config.db.url":"jdbc:oracle:thin@localhost:1521:xe",

"test.config.db.password":"oracle-password",

"test.config.db.driver-class-name":"oracle.jdbc.driver.OracleDriver",

"test.config.db.username":"oracle"

}

},

{　　//此处即为默认配置文件spring-cloud-config-client.properties

"name":"https://gitee.com/kerwin\_kim/test/**spring-cloud-config-client.properties**",

"source":

{

"test.config.db.url":"jdbc:mysql://localhost:3306/test",

"test.config.db.password":"mysql-password",

"test.config.db.driver-class-name":"com.mysql.jdbc.Driver",

"test.config.db.username":"mysql"

}

}

]

}

[复制代码](javascript:void(0);)

**3.2 搭建config client**

　　被管理配置文件的微服务称为spring-cloud-config-client，其本身就是一个普通的微服务，只是配置文件由spring-cloud-config-server集中管理。这个微服务应用必须依赖spring-cloud-starter-config启动器资源，否则无法访问spring-cloud-config-server来获取远程配置内容。

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>**spring-cloud-starter-config**</artifactId>

</dependency>

　　spring-cloud-config-client应用必须提供当前服务的基础配置内容，如服务名称、服务端口、Eureka注册中心等，同时还必须配置访问哪一个spring-cloud-config-server。而**当前应用的配置由被集中管理，所以全局配置文件命名修改为bootstrap.properties**。**bootstrap配置文件也是spring-cloud的全局配置文件，其加载优先级高于application系列全局配置文件**，**spring-cloud-config-client应用启动的时候先根据bootstrap配置来设置本地应用最基础的环境，再根据bootstrap中配置的远程spring-cloud-config-server信息来远程加载被集中管理的配置文件**。

　　bootstrap.properties配置内容如下：

[复制代码](javascript:void(0);)

spring.application.name=spring-cloud-config-client

server.port=10803

# 配置config-client相关内容

# 开启配置中心搜索

spring.cloud.config.discovery.enabled=true

# 配置config-server配置中心的服务名称

spring.cloud.config.discovery.serviceId=spring-cloud-config-server

# 配置要从配置中心获取的配置文件具体环境，默认为default

spring.cloud.config.profile=dev

# 配置GIT中的分支名称，默认为master分支

spring.cloud.config.label=develop

[复制代码](javascript:void(0);)

**四、热刷新client配置内容**

　　Config-client客户端服务应用，在启动的时候，会根据bootstrap配置内容远程访问config-server，获取当时的最新的配置内容。如果config-client运行过程中，GIT仓库中的配置内容发生变更，**config-client不会自动的加载刷新配置内容**。需要人为干预。

* 重启服务，一定会刷新配置。重启服务代价太高。
* 热刷新，服务不重启，不间断对外提供服务，而是重新加载配置内容，初始化应用环境。

　　在工作中，不能因为配置的变更频繁重启各微服务，这样会严重影响系统的服务能力。那么在配置变化后，如何让微服务热刷新配置内容是一个必须解决的问题。对spring-cloud-config-server来说，GIT远程仓库中的配置内容是否发生变化都是无所谓的，因为当spring-cloud-config-client访问spring-cloud-config-server获取配置内容时，spring-cloud-config-server都会尝试访问GIT，获取最新的配置内容。

　　而spring-cloud-config-client应用在启动后，需要提供不重启服务重新加载远程配置逻辑。这个**热属性配置内容的功能需要依赖spring-boot-starter-actuator启动器来实现**。

<dependency>

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

<artifactId>**spring-boot-starter-actuator**</artifactId>

</dependency>

　　在actuator组件中，提供了refresh功能，这个功能就是刷新应用环境。但是**spring-cloud默认关闭了这个功能的访问权限，需要在bootstrap配置文件中开启refresh访问权限**。

management.security.enabled=false

　　actuator组件提供的refresh功能要求**必须使用http协议访问，且必须是POST请求**。方式任意(如postmam或者thymeleaf发送http请求)。如下通过thymeleaf刷新客户端示例：

　　引入依赖：

<!-- thymeleaf启动器 -->

<dependency>

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

<artifactId>**spring-boot-starter-thymeleaf**</artifactId>

</dependency>

　　新增Controller：

[复制代码](javascript:void(0);)

@Controller

public class DispatcherController {

**@GetMapping("/")**

public String test(){

return "index";

}

}

[复制代码](javascript:void(0);)

　　新增页面：

[复制代码](javascript:void(0);)

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>热刷新应用配置</title>

<script type="text/javascript" src="jquery-1.9.1.js"></script>

<script type="text/javascript">

/\*

\* 注意： AJAX默认是不能做跨域访问的。当前的代码是为当前应用提供服务的。

\* 如果需要跨域访问，建议使用HTTPClient技术实现。或其他技术实现。

\*/

function refresh(){

var url = $("#urlValue").val();

$.ajax({

'url' : url,

'type': 'post',

'success' : function(data){

alert(data);

}

});

}

</script>

</head>

<body>

<center>

刷新工程：<input type="text" name="urlValue" id="urlValue"/>

<input type="button" value="刷新" onclick="refresh();">

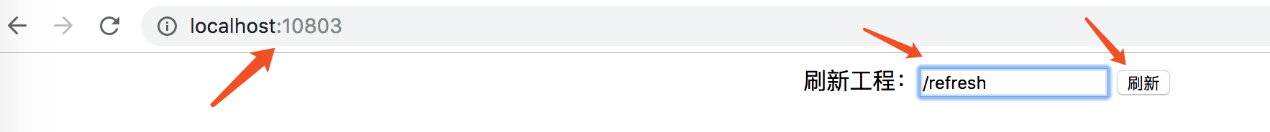
</center>

</body>

</html>

[复制代码](javascript:void(0);)

　　浏览器输入链接访问：http://ip:port/；再界面中输入/refresh点击“刷新”按钮即可，如图。



**五、配置中心加密**

　　配置文件统一存储在GIT中，虽然增强了管理和版本控制，但是文件内容的安全性也成了问题。如果外部任意知道了GIT地址或spring-cloud-config-server的地址，那么配置文件的全部内容相当于对外完全开放。这个时候就需要提供配置内容加解密逻辑。将加密后的配置内容保存在GIT仓库中，在spring-cloud-config-client访问spring-cloud-config-server的时候，通过spring-cloud-config-server实现加密数据的解密，这样就可以保证配置内容的安全了。

　　在spring-cloud架构中，Dalston.SR1版本可以正常提供加解密功能，而其他版本有BUG，无法提供加解密功能。使用加解密逻辑，需要在JDK/JRE环境中增加policy支持。需要在oracle官网上下载policy相关资源，地址为：

　　http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html

　　将下载的两个policy相关jar包复制到$JAVA\_HOME/lib/和$JAVA\_HOME/jre/lib/目录中即可。当环境准备完毕后，可以通过下述地址测试spring-cloud-config-server的加解密环境是否生效：

　　http://config-server-ip:port/encrypt/status

**5.1 对称加密**

　　使用相同的密钥（key）实现配置数据的加密和解密。在spring-cloud-config-server的全局配置文件中增加密钥配置：

# 对称加密：配置密钥内容，加密和解密都使用这个密钥

encrypt.key=test

　　spring-cloud-config-server提供了对数据加密和解密的处理逻辑，要求必须使用**POST请求方式**，在请求体中直接传递要加解密的内容（json请求参数提交方式），spring-cloud-config-server会返回加解密后的结果。

* **加密请求路径：http://config-server-ip:port/encrypt**
* **解密请求路径：http://config-server-ip:port/decrypt**

　　在GIT仓库中保存的密文数据使用前缀{cipher}表示数据为密文，如：

test.config.db.username=**{cipher}**bb605c7e4bac3ae59ed83860a164b58b48fa1b08012ccaf49eb0b67825fefc33

test.config.db.password=**{cipher}**7e97d20fe476b5268da4fea3dec63636f96e0dde4357acda4e614d33fcda1cfd

　　这样spring-cloud-config-client通过spring-cloud-config-server获取配置数据的时候，spring-cloud-config-server会自动实现解密过程，并将解密后的结果返回给spring-cloud-config-client。

　　同样采用thymeleaf，引入依赖**spring-boot-starter-thymeleaf，**然后添加Controller，

[复制代码](javascript:void(0);)

@Controller

public class DispatcherController {

@GetMapping("/")

public String test(){

return "index";

}

}

[复制代码](javascript:void(0);)

　　添加页面：

[复制代码](javascript:void(0);)

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>加解密测试工具</title>

<script type="text/javascript" src="jquery-1.9.1.js"></script>

<script type="text/javascript">

function encrypt(){

var source = $("#sourceEncrypt").val();

$.ajax({

'url' : '**http://localhost:10801/encrypt**',

'type': '**post**',

'data': source,

'success' : function(data){

$("#singEncrypt").val(data);

}

});

}

function decrypt(){

var sing = $("#singDecrypt").val();

$.ajax({

'url' : '**http://localhost:10801/decrypt**',

'type': '**post**',

'data': sing,

'success' : function(data){

$("#sourceDecrypt").val(data);

}

});

}

</script>

</head>

<body>

<div style="padding-left: 100px; padding-right: 100px">

明文：<input type="text" name="urlValue" id="sourceEncrypt" style="width: 300px"/>

<br>

密文：<input type="text" name="urlValue" id="singEncrypt" style="width: 300px"/>

<input type="button" value="加密" onclick="encrypt();">

<hr>

密文：<input type="text" name="urlValue" id="singDecrypt" style="width: 300px"/>

<br>

明文：<input type="text" name="urlValue" id="sourceDecrypt" style="width: 300px"/>

<input type="button" value="解密" onclick="decrypt();">

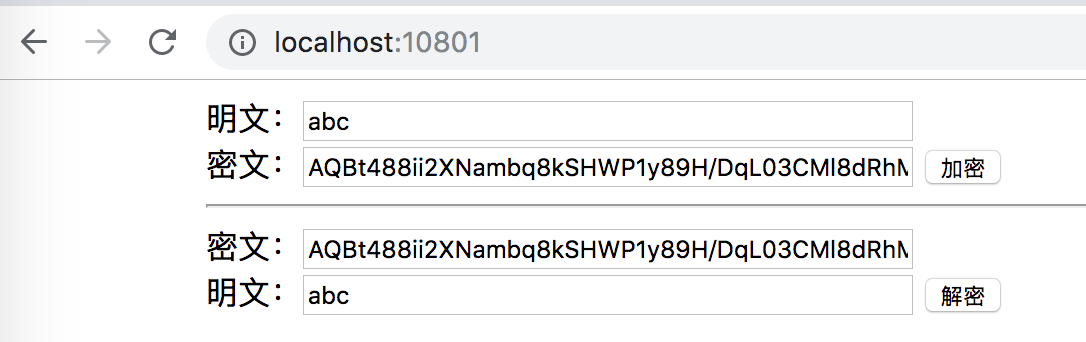
</div>

</body>

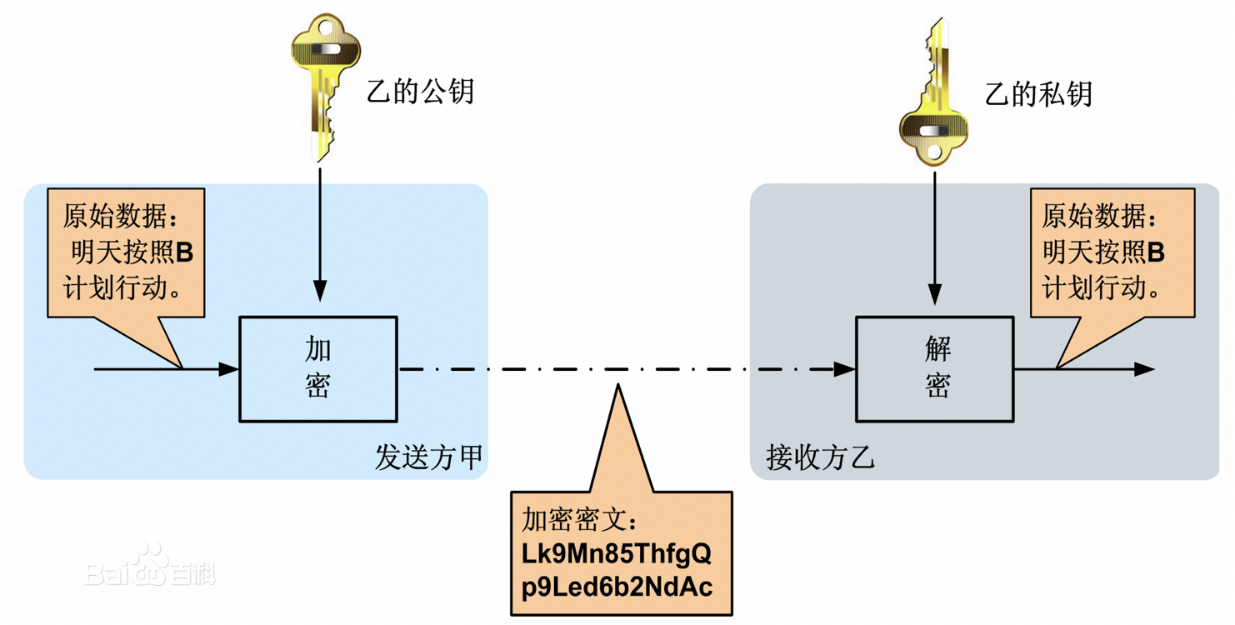
</html>

[复制代码](javascript:void(0);)

　　示例如图：



**5.2 非对称加密**

　　使用不同的密钥（key）实现配置数据的加密和解密。**使用公钥加密数据，使用私钥解密数据**。  
　　　　　　　　　　　　

　　JDK提供了java-keytool工具来生成密钥，生成的密钥信息是"keystore"文件。keytool工具使用方式如下：

[复制代码](javascript:void(0);)

keytool -genkeypair -alias "test" -keyalg "RSA" -keystore "test.keystore"

输入密钥库口令: 123456

再次输入新口令: 123456

您的名字与姓氏是什么?

[Unknown]: test

您的组织单位名称是什么?

[Unknown]: test

您的组织名称是什么?

[Unknown]: test

您所在的城市或区域名称是什么?

[Unknown]: beijing

您所在的省/市/自治区名称是什么?

[Unknown]: beijing

该单位的双字母国家/地区代码是什么?

[Unknown]: cn

CN=test, OU=test, O=test, L=beijing, ST=beijing, C=cn是否正确?

[否]: y

输入 <test> 的密钥口令test123

(如果和密钥库口令相同, 按回车):

再次输入新口令:test123

[复制代码](javascript:void(0);)

　　将生成的密钥文件test.keystore保存到spring-cloud-config-server的classpath下（src/main/resources），并在全局配置文件中增加下述内容：

[复制代码](javascript:void(0);)

# 密钥文件保存路径

encrypt.key-store.location=classpath:test.keystore

# 密钥别名，此别名是公开的

encrypt.key-store.alias=test

# 密钥库口令

encrypt.key-store.password=123456

# 密钥口令

encrypt.key-store.secret=test123

[复制代码](javascript:void(0);)

　　实现配置数据的加解密方式和对称加密一致。

**六、配置中心的安全认证**

　　现在配置内容通过加解密的方式实现了安全保障，但是配置中心spring-cloud-config-server还是完全对外公开的。如果通过http协议地址：

**http://config-server-ip:port/applicationName/profile/label**

　　还是可以获取到配置文件内容，这对配置信息的安全保护还是不足。需要给spring-cloud-config-server提供用户安全认证，在这里**使用spring-boot提供的security组件来使用HTTP Bastic用户安全认证**。

　　在spring-cloud-config-server中增加新的依赖security：

<dependency>

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

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

</dependency>

　　在spring-cloud-config-server全局配置文件中增加用户信息：

security.basic.enabled=true

security.user.name=test

security.user.password=123456

　　这样就为spring-cloud-config-server提供了用户安全认证。

　　在客户的bootstrap配置文件中增加下述内容，来实现spring-cloud-config-client访问spring-cloud-config-server的用户登录：

spring.cloud.config.username=test

spring.cloud.config.password=123456

**七、多配置文件加载**

　　如果要想加载多个指定配置文件，而且此配置文件不是以spring.application.name命名规则命名的配置文件，则可以在config-client端配置文件：

# 加载多个配置文件，指定配置文件命名，不再使用spring.application.name查找配置文件，而是使用配置内容找指定的配置文件。常用。是配置在client端

spring.cloud.config.name=fileName1,fileName2

　　如果要加载多目录下的配置文件，则需要在config-server端添加配置：

# 检索多个目录中的同名配置文件，常用于分目录管理多环境配置文件的情况下，如：dir1中保存default配置，dir2中保存dev配置，dir3中保存test配置。  
# 配置文件命名规则不变，默认还是applicationName[-profile].properties。是配置在server端的。

spring.cloud.config.server.git.search-paths=dir1,dir2,dir3

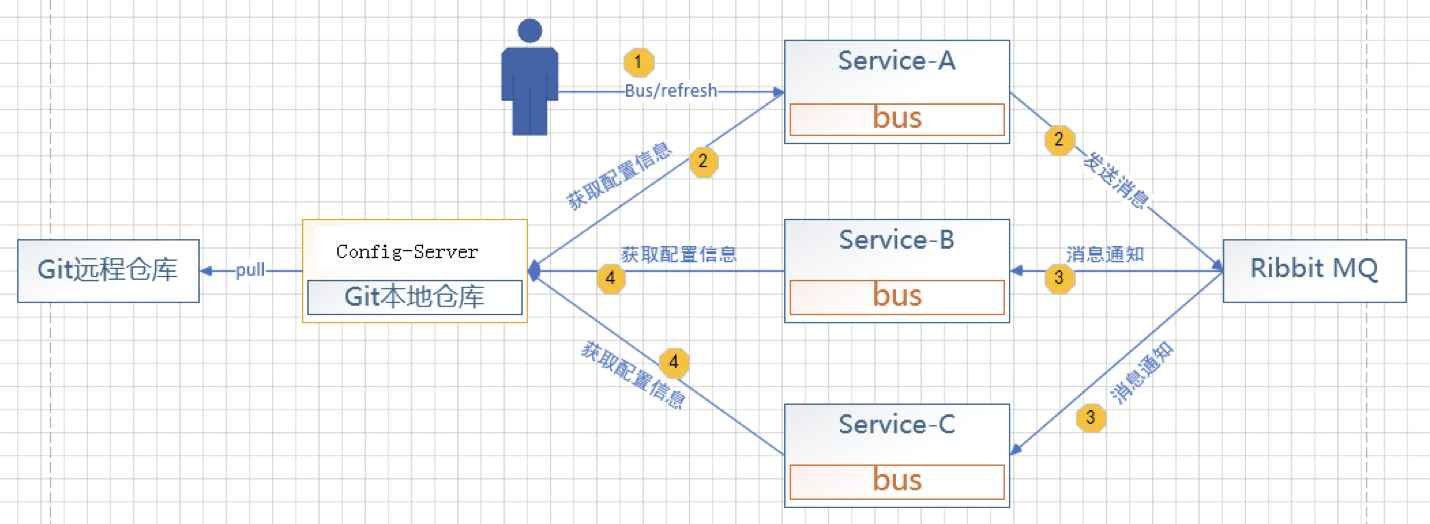
**八、 消息总线-BUS**

**1、什么是Spring Cloud Bus**

　　Spring Cloud Bus集成了市面上常见的RabbitMQ和Kafka等消息代理。其会连接微服务系统中所有拥有Bus总线机制的节点，当有数据变更的时候，会通过消息中间件使用消息广播的方式通知所有的微服务节点同步更新数据。（如：微服务配置更新等）

**2 使用消息总线实现配置热刷新**

**2.1 基于config client实现全局刷新**



　　这种实现方案在设计上并不是非常合适，比较刷新功能和某一个config client微服务耦合到了一起。可作为了解。

　　实现Bus刷新功能需要在所有的config client端应用中增加spring-cloud-starter-bus-amqp依赖，这个依赖是消息总线集成的RabbitMQ消息同步组件。

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>**spring-cloud-starter-bus-amqp**</artifactId>

</dependency>

　　在config client端的bootstrap配置文件中增加RabbitMQ相关配置：

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=test

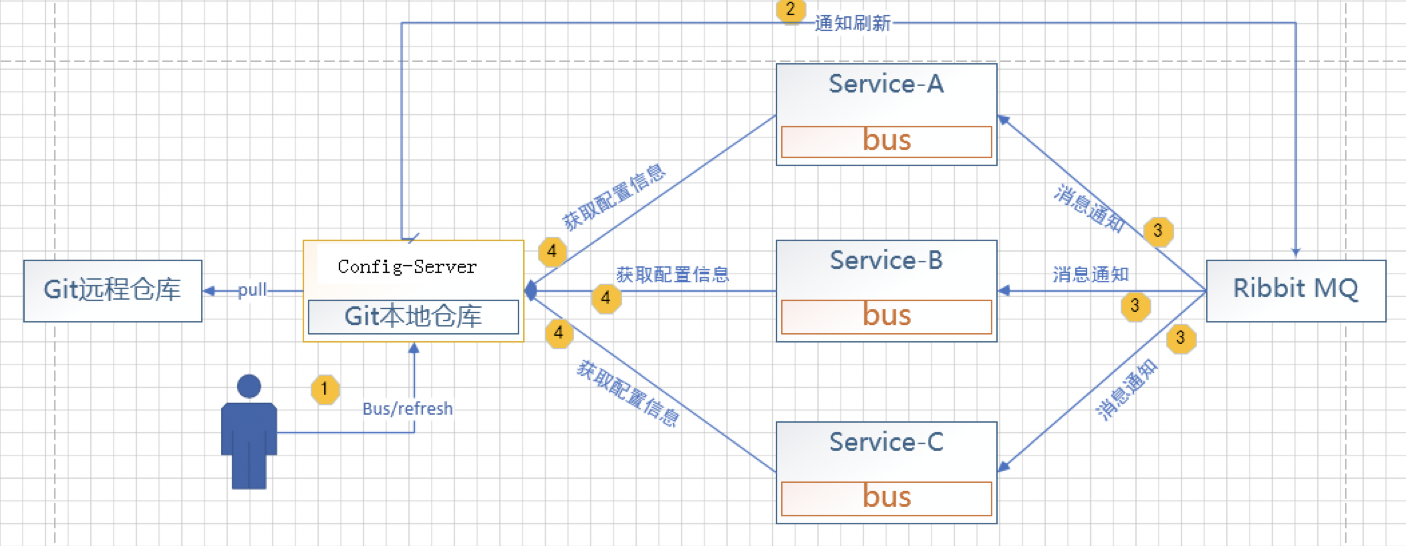
spring.rabbitmq.password=123456

　　Bus消息总线会提供一个**/bus/refresh**服务来实现应用的热刷新。不再使用actuator来提供人刷新逻辑。**/bus/refresh服务要求请求必须是post请求，可用任意技术实现**。

　　这里要注意，**虽然刷新服务的逻辑由Bus消息总线来提供，但是/bus/refresh服务默认还是关闭的**，需要开启，在config client端的bootstrap配置文件中必须配置下述内容：

management.security.enabled=false

**2.2 基于config server实现全局刷新**



　　这种实现方案在设计上更加完美，将热刷新逻辑和具体的服务应用解耦。推荐使用。实现过程和2.1一致。需要config server和config client端都引入Bus消息总线组件。

**2.3 局部刷新服务**

　　使用Bus总线也可以实现部分服务热刷新。前文讲到的两种刷新实现方式都是全局热刷新，所有带有Bus消息总线组件且与同一个RabbitMQ连接的微服务应用都会执行热刷新逻辑。有些时候，热刷新只需要在一个微服务应用或一个微服务应用集群中执行，这个时候就需要使用局部刷新功能。

**局部热刷新的实现过程和全局热刷新一致，只是请求地址/bus/refresh需要提供一个GET请求参数，请求参数命名为destination**，参数值是要刷新的微服务名称:端口[\*\*]。如：

　　微服务名： config-client

　　微服务端口： 10802、10803、10804

　　如果只热刷新微服务config-client:10802时，请求的路径为/bus/refresh?**destination=config-client:10802**。

　　如果需要刷新config-client微服务集群，则请求路径为/bus/refresh?**destination=config-client:\*\***