Spring Cloud Config

Spring Cloud Config为分布式系统中的外部化配置提供服务器和客户端支持。通过Config Server，您可以在所有环境中管理应用程序的外部属性。客户端和服务器上的概念与Spring Environment和PropertySource抽象一致，所以它们非常适合Spring应用程序，但可以与任何运行在任何语言中的应用程序一起使用。随着应用程序从开发到测试转移到部署流程中，您可以管理这些环境之间的配置，并确保应用程序具有迁移时所需的所有内容。服务器存储后端的默认实现使用git，因此它可以轻松支持配置环境的标签版本，并且可以用于管理内容的各种工具。使用Spring配置很容易添加替代实现并将其插入。

使用原因

在单体式应用中，我们通常的做法是将配置文件和代码放在一起，这没有什么不妥。当你的应用变得越来越大从而不得不进行服务化拆分的时候，会发现各种provider实例越来越多，修改某一项配置越来越麻烦，你常常不得不为修改某一项配置而重启某个服务所有的provider实例，甚至为了灰度上线需要更新部分provider的配置。这个时候有一套配置文件集中管理方案就变得十分重要，SpringCloudConfig和SpringCloudBus就是这种问题的解决方案之一，业界也有些知名的同类开源产品，比如百度的disconf。

相比较同类产品，SpringCloudConfig最大的优势是和Spring无缝集成，支持Spring里面`Environment`和`PropertySource`的接口，对于已有的Spring应用程序的迁移成本非常低，在配置获取的接口上是完全一致，结合SpringBoot可使你的项目有更加统一的标准（包括依赖版本和约束规范），避免了应为集成不同开软件源造成的依赖版本冲突。

ConfigClient 启动顺序

ConfigClient最好要在ConfigServer之后启动，Spring加载配置文件是有顺序的，靠前的配置文件会覆盖靠后的配置文件中相同键的值，如果ConfigServer先启动可以保证ConfigClient将远程的配置文件加载到最前面，如果使用中没有注意到这一点，有可能导致你本地的配置文件先于远程的加载，导致本地的配置覆盖远程配置。当然，你也可以让本地配置和远程配置完全不重复，这样也可以避免键/值覆盖的问题。

快速开始

启动服务器：

```
$ cd spring-cloud-config-server
$ ../mvnw spring-boot:run
```

服务器是Spring Boot应用程序，所以如果您愿意，您可以从IDE运行它（主要类是ConfigServerApplication）。 然后尝试一个客户端：

```
$ curl localhost:8888/foo/development
{"name":"foo","label":"master","propertySources":[
  {"name":"https://github.com/scratches/config-repo/foo-
development.properties","source":{"bar":"spam"}},
  {"name":"https://github.com/scratches/config-repo/foo.properties","source":
{"foo":"bar"}}
]}
```

定位资源的默认策略是克隆一个git仓库（在`spring.cloud.config.server.git.uri`），并使用它来初始化一个迷你`SpringApplication`。小应用程序的`Environment`用于枚举属性源并通过JSON端点发布。

HTTP服务具有以下形式的资源：

```
/{application}/{profile}[/{label}]
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

其中“应用程序”作为`SpringApplication`中的`spring.config.name`注入（即常规的Spring Boot应用程序中通常是“应用程序”），“配置文件”是活动配置文件（或逗号分隔列表的属性），“label”是可选的git标签（默认为“master”）。

Spring Cloud Config服务器从git存储库（必须提供）为远程客户端提供配置：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
```

客户端使用情况

要在应用程序中使用这些功能，只需将其构建为依赖于spring-cloud-config-client的Spring引导应用程序（例如，查看配置客户端或示例应用程序的测试用例）。添加依赖关系的最方便的方法是通过Spring Boot启动器`org.springframework.cloud:spring-cloud-starter-config`。还有一个Maven用户的父pom和BOM（`spring-cloud-starter-parent`）和用于Gradle和Spring CLI用户的Spring IO版本管理属性文件。示例Maven配置：

的pom.xml

```
 <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.5.10.RELEASE</version>
       <relativePath /> <!-- lookup parent from repository -->
   </parent>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Edgware.SR3</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
           <plugin>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-maven-plugin</artifactId>
           </plugin>
    </plugins>
</build>

   <!-- repositories also needed for snapshots and milestones -->
```

然后你可以创建一个标准的Spring Boot应用程序，就像这个简单的HTTP服务器一样

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

运行时，它将从端口8888上的默认本地配置服务器接收外部配置（如果它正在运行）。 要修改启动行为，可以使用bootstrap.properties（如application.properties，但是应用程序上下文的引导阶段）来更改配置服务器的位置，例如，

```
spring.cloud.config.uri: http://myconfigserver.com
```

bootstrap属性将作为高优先级属性源显示在/ env端点中，例如，

```
$ curl localhost:8080/env
{
  "profiles":[],
  "configService:https://github.com/spring-cloud-samples/config-repo/bar.properties":
{"foo":"bar"},
  "servletContextInitParams":{},
  "systemProperties":{...},
  ...
}
```

（名为“configService：<远程存储库的URL> / <文件名>”的属性源包含属性“foo”，其值为“bar”并且是最高优先级）。

Spring Cloud Config服务器

服务器为外部配置（名称 - 值对或同等YAML内容）提供了一个HTTP，基于资源的API。

pom.xml

```
    <!--添加相关依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
```

服务器可以使用@EnableConfigServer批注轻松嵌入到Spring Boot应用程序中。 所以这个程序是一个配置服务器：

ConfigServer.java.

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
  public static void main(String[] args) {
    SpringApplication.run(ConfigServer.class, args);
  }
}
```

与所有Spring Boot应用程序一样，它默认在端口8080上运行，但您可以通过各种方式将其切换到常规端口8888。最简单的方法是设置默认的配置库，方法是使用spring.config.name = configserver启动它（在Config Server jar中有一个configserver.yml）。 另一种方法是使用自己的application.properties，例如

application.properties.

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

其中`${user.home}/config-repo`是包含YAML和属性文件的git仓库。

> 在Windows中，如果文件URL为绝对驱动器前缀，例如`file:///${user.home}/config-repo`，则需要额外的“/”。

以下是上面示例中创建git仓库的方法：

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```

> 使用本地文件系统进行git存储库仅用于测试。使用服务器在生产环境中托管配置库。
>
> 如果您只保留文本文件，则配置库的初始克隆将会快速有效。如果您开始存储二进制文件，尤其是较大的文件，则可能会遇到服务器中第一个配置请求和/或内存不足错误的延迟。

环境库

您要在哪里存储配置服务器的配置数据？管理此行为的策略是`EnvironmentRepository`，服务于`Environment`对象。此`Environment`是Spring `Environment`（包括`propertySources`作为主要功能）的域的浅层副本。`Environment`资源由三个变量参数化：

- `{application}`映射到客户端的“spring.application.name”;
- `{profile}`映射到客户端上的“spring.profiles.active”（逗号分隔列表）; 和
- `{label}`这是一个服务器端功能，标记“版本”的配置文件集。

存储库实现通常表现得像一个Spring Boot应用程序从“spring.config.name”等于`{application}`参数加载配置文件，“spring.profiles.active”等于`{profiles}`参数。配置文件的优先级规则也与常规启动应用程序相同：活动配置文件优先于默认配置，如果有多个配置文件，则最后一个获胜（例如向`Map`添加条目）。

示例：客户端应用程序具有此引导配置：

bootstrap.yml

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

（通常使用Spring Boot应用程序，这些属性也可以设置为环境变量或命令行参数）。

如果存储库是基于文件的，则服务器将从`application.yml`创建`Environment`（在所有客户端之间共享），`foo.yml`（以`foo.yml`优先））。如果YAML文件中有文件指向Spring配置文件，那么应用的优先级更高（按照列出的配置文件的顺序），并且如果存在特定于配置文件的YAML（或属性）文件，那么这些文件也应用于优先级高于默认值。较高优先级转换为`Environment`之前列出的`PropertySource`。（这些规则与独立的Spring Boot应用程序相同。）

Git后端

`EnvironmentRepository`的默认实现使用Git后端，这对于管理升级和物理环境以及审核更改非常方便。要更改存储库的位置，可以在Config Server中设置“spring.cloud.config.server.git.uri”配置属性（例如`application.yml`）。如果您使用`file:`前缀进行设置，则应从本地存储库中工作，以便在没有服务器的情况下快速方便地启动，但在这种情况下，服务器将直接在本地存储库上进行操作，而不会克隆如果它不是裸机，因为配置服务器永远不会更改“远程”资源库）。要扩展Config Server并使其高度可用，您需要将服务器的所有实例指向同一个存储库，因此只有共享文件系统才能正常工作。即使在这种情况下，最好使用共享文件系统存储库的`ssh:`协议，以便服务器可以将其克隆并使用本地工作副本作为缓存。

该存储库实现将HTTP资源的`{label}`参数映射到git标签（提交ID，分支名称或标签）。如果git分支或标签名称包含斜杠（“/”），则应使用特殊字符串“`(_)`”*指定HTTP URL中的标签，以避免与其他URL路径模糊。例如，如果标签为`foo/bar`，则替换斜杠将导致标签看起来像`foo(_)bar`。包含特殊字符串“`(_)`”*也可以应用于{application}参数。如果您使用像curl这样的命令行客户端（例如使用引号将其从shell中转出来），请小心URL中的方括号。

Git URI中的通配符

如果您需要，Spring Cloud Config Server支持带有{application}和{profile}（和{label}）占位符的git存储库URL，但请记住该标签作为git标签应用。 因此，您可以轻松地使用（例如）支持“每个应用程序一个repo”政策：

> 一个微服务一个配置文件,不影响其他的项目

```
spring:
  cloud:
    config:
      server:
        git:
           uri: https://github.com/myorg/{application}
```

或使用类似模式但使用`{profile}`的“每个配置文件一个”策略。

此外，在{应用}参数中使用特殊字符串“（_）”可以支持多个组织（例如）：

模式匹配和多个存储库

还有对应用程序和配置文件名称进行模式匹配的更复杂要求的支持。 模式格式是带有通配符的{application} / {profile}名称的逗号分隔列表（其中可能需要引用以通配符开头的模式）。 例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            simple: https://github.com/simple/config-repo
            special:
              pattern: special*/dev*,*special*/dev*
              uri: https://github.com/special/config-repo
            local:
              pattern: local*
              uri: file:/home/configsvc/config-repo
```

如果{application} / {profile}与任何模式都不匹配，它将使用在“spring.cloud.config.server.git.uri”下定义的默认URI。 在上面的示例中，对于“simple”存储库，该模式simple/ *（即它只匹配所有配置文件中名为“简单”的一个应用程序）。 “local”存储库匹配所有配置文件中以“local”开头的所有应用程序名称（/ *后缀自动添加到任何没有配置文件匹配器的模式）。

> 上述“simple”示例中使用的“单行”快捷方式只能在要设置的唯一属性是URI的情况下使用。 如果您需要设置其他任何内容（凭据，模式等），则需要使用完整的表单。

repo中的`pattern`属性实际上是一个数组，因此您可以使用属性文件中的YAML数组（或`[0]`，`[1]`等后缀）绑定到多个模式。如果要运行具有多个配置文件的应用程序，则可能需要执行此操作。例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          repos:
            development:
              pattern:
                - '*/development'
                - '*/staging'
              uri: https://github.com/development/config-repo
            staging:
              pattern:
                - '*/qa'
                - '*/production'
              uri: https://github.com/staging/config-repo
```

> Spring Cloud会猜测，包含不以*结尾的配置文件的模式意味着您实际上想匹配以此模式开头的配置文件列表（因此* / staging是[“* / staging”，“*/分期，*”]）。 例如，您需要在本地“开发”配置文件中运行应用程序，而在远程运行“cloud”配置文件时，这也很常见。

每个存储库还可以选择将配置文件存储在子目录中，并且可以将搜索这些目录的模式指定为searchPaths。 例如在顶层：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: foo,bar*
```

在这个例子中，服务器搜索顶层和“foo /”子目录中的配置文件，以及名称以“bar”开头的任何子目录。

默认情况下，服务器在首次请求配置时克隆远程存储库。 可以将服务器配置为在启动时克隆存储库。 例如在顶层：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          repos:
            team-a:
                pattern: team-a-*
                cloneOnStart: true
                uri: http://git/team-a/config-repo.git
            team-b:
                pattern: team-b-*
                cloneOnStart: false
                uri: http://git/team-b/config-repo.git
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

在这个例子中，服务器在接受任何请求之前，在启动时克隆team-a的config-repo。 除非请求存储库中的配置，否则所有其他存储库都不会被克隆。

> 在配置服务器启动时设置要克隆的存储库可以帮助快速识别配置服务器启动时错误配置的配置源（例如，无效的存储库URI）。 使用cloneOnStart未启用配置源时，配置服务器可能会以错误配置或无效的配置源成功启动，并且在应用程序从该配置源请求配置之前检测不到错误。

认证

要在远程存储库上使用HTTP基本认证，单独添加“username”和“password”属性（不在URL中）

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          username: trolley
          password: strongpassword
```

如果您不使用HTTPS和用户凭据，当您将密钥存储在默认目录（`~/.ssh`）中，并且uri指向SSH位置时，SSH也应该开箱即用，例如“ git@github.com：配置/云配置”。必须在`~/.ssh/known_hosts`文件中存在Git服务器的条目，并且它是`ssh-rsa`格式。其他格式（如`ecdsa-sha2-nistp256`）不受支持。为了避免意外，您应该确保Git服务器的`known_hosts`文件中只有一个条目，并且与您提供给配置服务器的URL匹配。如果您在URL中使用了主机名，那么您希望在`known_hosts`文件中具有这一点，而不是IP。使用JGit访问存储库，因此您发现的任何文档都应适用。HTTPS代理设置可以`~/.git/config`设置，也可以通过系统属性（`-Dhttps.proxyHost`和`-Dhttps.proxyPort`）与任何其他JVM进程相同。

> 如果您不知道`~/.git`目录使用`git config--global`来处理设置的位置（例如`git config --globalhttp.sslVerify false`）。

使用AWS CodeCommit进行认证

AWS CodeCommit认证也可以完成。当从命令行使用Git时，AWS CodeCommit使用身份验证助手。该帮助器不与JGit库一起使用，因此如果Git URI与AWS CodeCommit模式匹配，则将创建用于AWS CodeCommit的JGit CredentialProvider。AWS CodeCommit URI始终看起来像 https：//git-codecommit.$AWS_REGION.amazonaws.com/{repopath}。

如果您使用AWS CodeCommit URI提供用户名和密码，那么这些URI必须是用于访问存储库的AWS accessKeyId和secretAccessKey。如果不指定用户名和密码，则将使用AWS默认凭据提供程序链检索accessKeyId和secretAccessKey 。

如果您的Git URI与CodeCommit URI模式（上述）匹配，则必须在用户名和密码或默认凭据提供程序链支持的某个位置中提供有效的AWS凭据。AWS EC2实例可以使用EC2实例的 IAM角色。

注意：aws-java-sdk-core jar是一个可选的依赖关系。如果aws-java-sdk-core jar不在您的类路径上，则无论git服务器URI如何，都将不会创建AWS代码提交凭据提供程序。

使用Git的SSH配置

默认情况下，Spring Cloud Config Server使用的JGit库使用SSH配置文件，例如~/。 ssh / known_hosts和/ etc / ssh / ssh_config，当使用SSH URI连接到Git存储库时。在Cloud Foundry等云环境中，本地文件系统可能是短暂的或不易访问的。对于这些情况，可以使用Java属性设置SSH配置。为了激活基于属性的SSH配置，属性为`spring.cloud.config.server.git.ignoreLocalSshSettings`。 必须设置为true。例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: git@gitserver.com:team/repo1.git
          ignoreLocalSshSettings: true
          hostKey: someHostKey
          hostKeyAlgorithm: ssh-rsa
          privateKey: |
                      -----BEGIN RSA PRIVATE KEY-----
                      MIIEpgIBAAKCAQEAx4UbaDzY5xjW6hc9jwN0mX33XpTDVW9WqHp5AKaRbtAC3DqX
                      IXFMPgw3K45jxRb93f8tv9vL3rD9CUG1Gv4FM+o7ds7FRES5RTjv2RT/JVNJCoqF
                      ol8+ngLqRZCyBtQN7zYByWMRirPGoDUqdPYrj2yq+ObBBNhg5N+hOwKjjpzdj2Ud
                      1l7R+wxIqmJo1IYyy16xS8WsjyQuyC0lL456qkd5BDZ0Ag8j2X9H9D5220Ln7s9i
                      oezTipXipS7p7Jekf3Ywx6abJwOmB0rX79dV4qiNcGgzATnG1PkXxqt76VhcGa0W
                      DDVHEEYGbSQ6hIGSh0I7BQun0aLRZojfE3gqHQIDAQABAoIBAQCZmGrk8BK6tXCd
                      fY6yTiKxFzwb38IQP0ojIUWNrq0+9Xt+NsypviLHkXfXXCKKU4zUHeIGVRq5MN9b
                      BO56/RrcQHHOoJdUWuOV2qMqJvPUtC0CpGkD+valhfD75MxoXU7s3FK7yjxy3rsG
                      EmfA6tHV8/4a5umo5TqSd2YTm5B19AhRqiuUVI1wTB41DjULUGiMYrnYrhzQlVvj
                      5MjnKTlYu3V8PoYDfv1GmxPPh6vlpafXEeEYN8VB97e5x3DGHjZ5UrurAmTLTdO8
                      +AahyoKsIY612TkkQthJlt7FJAwnCGMgY6podzzvzICLFmmTXYiZ/28I4BX/mOSe
                      pZVnfRixAoGBAO6Uiwt40/PKs53mCEWngslSCsh9oGAaLTf/XdvMns5VmuyyAyKG
                      ti8Ol5wqBMi4GIUzjbgUvSUt+IowIrG3f5tN85wpjQ1UGVcpTnl5Qo9xaS1PFScQ
                      xrtWZ9eNj2TsIAMp/svJsyGG3OibxfnuAIpSXNQiJPwRlW3irzpGgVx/AoGBANYW
                      dnhshUcEHMJi3aXwR12OTDnaLoanVGLwLnkqLSYUZA7ZegpKq90UAuBdcEfgdpyi
                      PhKpeaeIiAaNnFo8m9aoTKr+7I6/uMTlwrVnfrsVTZv3orxjwQV20YIBCVRKD1uX
                      VhE0ozPZxwwKSPAFocpyWpGHGreGF1AIYBE9UBtjAoGBAI8bfPgJpyFyMiGBjO6z
                      FwlJc/xlFqDusrcHL7abW5qq0L4v3R+FrJw3ZYufzLTVcKfdj6GelwJJO+8wBm+R
                      gTKYJItEhT48duLIfTDyIpHGVm9+I1MGhh5zKuCqIhxIYr9jHloBB7kRm0rPvYY4
                      VAykcNgyDvtAVODP+4m6JvhjAoGBALbtTqErKN47V0+JJpapLnF0KxGrqeGIjIRV
                      cYA6V4WYGr7NeIfesecfOC356PyhgPfpcVyEztwlvwTKb3RzIT1TZN8fH4YBr6Ee
                      KTbTjefRFhVUjQqnucAvfGi29f+9oE3Ei9f7wA+H35ocF6JvTYUsHNMIO/3gZ38N
                      CPjyCMa9AoGBAMhsITNe3QcbsXAbdUR00dDsIFVROzyFJ2m40i4KCRM35bC/BIBs
                      q0TY3we+ERB40U8Z2BvU61QuwaunJ2+uGadHo58VSVdggqAo0BSkH58innKKt96J
                      69pcVH/4rmLbXdcmNYGm6iu+MlPQk4BUZknHSmVHIFdJ0EPupVaQ8RHT
                      -----END RSA PRIVATE KEY-----
```

Table 5.1. SSH Configuration properties

Property Name	Remarks
ignoreLocalSshSettings	If true, use property based SSH config instead of file based. Must be set at as `spring.cloud.config.server.git.ignoreLocalSshSettings`, **not** inside a repository definition.
privateKey	Valid SSH private key. Must be set if `ignoreLocalSshSettings` is true and Git URI is SSH format
hostKey	Valid SSH host key. Must be set if `hostKeyAlgorithm` is also set
hostKeyAlgorithm	One of `ssh-dss, ssh-rsa, ecdsa-sha2-nistp256, ecdsa-sha2-nistp384 ,ecdsa-sha2-nistp521`. Must be set if `hostKey` is also set
strictHostKeyChecking	`true` or `false`. If false, ignore errors with host key
knownHostsFile	Location of custom .known_hosts file
preferredAuthentications	Override server authentication method order. This should allow evade login prompts if server has keyboard-interactive authentication before `publickey` method.

Git搜索路径中的通配符

Spring Cloud Config服务器还支持具有`{application}`和`{profile}`（以及`{label}`（如果需要））占位符的搜索路径。例：

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          searchPaths: '{application}'
```

在资源库中搜索与目录（以及顶级）相同名称的文件。通配符在具有占位符的搜索路径中也是有效的（搜索中包含任何匹配的目录）。

强制刷新Git存储库

如前所述，Spring Cloud Config Server对远程git存储库进行了克隆，并且如果本地副本以某种方式变脏（例如，按OS进程更改文件夹内容），Spring Cloud Config Server无法从远程存储库更新本地副本。

为了解决这个问题，如果本地副本很脏，会有一个强制拉取属性，使Spring Cloud Config Server从远程存储库强制拉取。例：

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/spring-cloud-samples/config-repo
          force-pull: true
```

如果您有多个存储库配置，则可以配置每个存储库的强制拉取属性。例：

```yaml
spring:
  cloud:
    config:
      server:
        git:
          uri: https://git/common/config-repo.git
          force-pull: true
          repos:
            team-a:
                pattern: team-a-*
                uri: http://git/team-a/config-repo.git
                force-pull: true
            team-b:
                pattern: team-b-*
                uri: http://git/team-b/config-repo.git
                force-pull: true
            team-c:
                pattern: team-c-*
                uri: http://git/team-a/config-repo.git
```

> force-pull属性的默认值为false。

删除Git存储库中未跟踪的分支---新

由于Spring Cloud Config Server在检出分支到本地回购（例如通过标签获取属性）后拥有远程git存储库的克隆，它将永久保留此分支或直到下一个服务器重新启动（这会创建新的本地回购）。所以可能会出现这样的情况：远程分支被删除，但它的本地副本仍然可用于获取。如果Spring Cloud Config Server客户端服务`spring.cloud.config.label=deletedRemoteBranch,master`，掌握它将从deletedRemoteBranch本地分支获取属性，但不从master获取属性。

```yaml
spring:
  cloud:
    config:
      server:
        git:
         uri: https://github.com/spring-cloud-samples/config-repo
         delete-untracked-branches: true
```

指定系统目录存储版本文件

> 使用基于VCS的后端（git，svn）文件被检出或克隆到本地文件系统。默认情况下，它们放在系统临时目录中，前缀为`config-repo-`。在linux上，例如可以是`/tmp/config-repo-`。一些操作系统会定期清除临时目录。这可能会导致意外的行为，例如缺少属性。为避免此问题，请通过将`spring.cloud.config.server.git.basedir`或`spring.cloud.config.server.svn.basedir`设置为不驻留在系统临时结构中的目录来更改Config Server使用的目录。

```
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/startSnow/config-repo
          basedir: /home/chixue/repo
```

文件系统后端

配置服务器中还有一个不使用Git的“本机”配置文件，只是从本地类路径或文件系统加载配置文件（您想要指向的任何静态URL“spring.cloud.config.server .native.searchLocations“）。要使用本机配置文件，只需使用“spring.profiles.active = native”启动Config Server。

> 请记住使用`file:`前缀的文件资源（缺省没有前缀通常是classpath）。与任何Spring Boot配置一样，您可以嵌入`${}`样式的环境占位符，但请记住，Windows中的绝对路径需要额外的“/”，例如`file:///${user.home}/config-repo`

> `searchLocations`的默认值与本地Spring Boot应用程序（所以`[classpath:/, classpath:/config,file:./, file:./config]`）相同。这不会将`application.properties`从服务器暴露给所有客户端，因为在发送到客户端之前，服务器中存在的任何属性源都将被删除。

> 文件系统后端对于快速入门和测试是非常好的。要在生产中使用它，您需要确保文件系统是可靠的，并在配置服务器的所有实例中共享。

搜索位置可以包含`{application}`，`{profile}`和`{label}`的占位符。以这种方式，您可以隔离路径中的目录，并选择一个有用的策略（例如每个应用程序的子目录或每个配置文件的子目录）。

如果您不在搜索位置使用占位符，则该存储库还将HTTP资源的`{label}`参数附加到搜索路径上的后缀，因此属性文件将从每个搜索位置加载**并**具有相同名称的子目录作为标签（标记的属性在Spring环境中优先）。因此，没有占位符的默认行为与添加以`/{label}/. For example`file:/tmp/config`结尾的搜索位置与`file:/tmp/config,file:/tmp/config/{label}`相同

Vault后端

Spring Cloud Config服务器还支持Vault作为后端。

> Vault是安全访问秘密的工具。一个秘密是你想要严格控制访问的任何东西，如API密钥，密码，证书等等。Vault为任何秘密提供统一的界面，同时提供严格的访问控制和记录详细的审核日志。

有关Vault的更多信息，请参阅Vault快速入门指南。

有关Vault的更多信息，请参阅Vault快速入门指南。

要使配置服务器使用Vault后端，您必须使用`vault`配置文件运行配置服务器。例如在配置服务器的`application.properties`中，您可以添加`spring.profiles.active=vault`。

默认情况下，配置服务器将假定您的Vault服务器正在运行于`http://127.0.0.1:8200`。它还将假定后端名称为`secret`，密钥为`application`。所有这些默认值都可以在配置服务器的`application.properties`中配置。以下是可配置Vault属性的表。所有属性前缀为`spring.cloud.config.server.vault`。

名称	默认值
host	127.0.0.1
port	8200
scheme	HTTP
backend	秘密
defaultKey	应用
profileSeparator	,

所有可配置的属性可以在`org.springframework.cloud.config.server.environment.VaultEnvironmentRepository`找到。

运行配置服务器后，可以向服务器发出HTTP请求，以从Vault后端检索值。为此，您需要为Vault服务器创建一个令牌。

首先放置一些数据给你Vault。例如

```
$ vault write secret/application foo=bar baz=bam
$ vault write secret/myapp foo=myappsbar
```

现在，将HTTP请求发送给您的配置服务器以检索值。

```
$ curl -X "GET" "http://localhost:8888/myapp/default" -H "X-Config-Token: yourtoken"
```

在提出上述要求后，您应该会看到类似的回复。

```
{
   "name":"myapp",
   "profiles":[
      "default"
   ],
   "label":null,
   "version":null,
   "state":null,
   "propertySources":[
      {
         "name":"vault:myapp",
         "source":{
            "foo":"myappsbar"
         }
      },
      {
         "name":"vault:application",
         "source":{
            "baz":"bam",
            "foo":"bar"
         }
      }
   ]
}
```

多个Properties来源

使用Vault时，您可以为应用程序提供多个属性源。例如，假设您已将数据写入Vault中的以下路径。

```
secret/myApp,dev
secret/myApp
secret/application,dev
secret/application
```

写入`secret/application`的Properties可 用于使用配置服务器的所有应用程序。名称为`myApp`的应用程序将具有写入`secret/myApp`和`secret/application`的任何属性。当`myApp`启用`dev`配置文件时，写入所有上述路径的属性将可用，列表中第一个路径中的属性优先于其他路径

与所有应用共享配置

基于文件的存储库

使用基于文件（即git，svn和native）的存储库，文件名为`application*`的资源在所有客户端应用程序（所以`application.properties`，`application.yml`，`application-*.properties`等）之间共享）。您可以使用这些文件名的资源来配置全局默认值，并根据需要将其覆盖应用程序特定的文件。

#_property_overrides [属性覆盖]功能也可用于设置全局默认值，并且允许占位符应用程序在本地覆盖它们。

提示	**使用“本机”配置文件（本地文件系统后端），建议您使用不属于服务器自身配置的显式搜索位置。否则，默认搜索位置中的`application*`资源将被删除，因为它们是服务器的一部分。**

Vault服务器

当使用Vault作为后端时，可以通过将配置放在`secret/application`中与所有应用程序共享配置。例如，如果您运行此Vault命令

```
$ vault write secret/application foo=bar baz=bam
```

使用配置服务器的所有应用程序都可以使用属性`foo`和`baz`。

JDBC Backend----

Spring Cloud Config Server支持JDBC（关系数据库）作为配置属性的后端。您可以通过将spring-jdbc添加到类路径中，并使用'jdbc'配置文件或添加JdbcEnvironmentRepository类型的bean来启用此功能。
如果您在类路径中包含正确的依赖关系，Spring Boot将配置数据源（有关更多详细信息，请参阅用户指南）。

数据库需要有一个名为'PROPERTIES'的表，其中包含'APPLICATION'，'PROFILE'，'LABEL'（具有通常的环境含义），以及属性样式中的键和值对的'KEY'和'VALUE'。所有字段都是Java中的String类型，因此您可以将它们设置为您需要的任何长度的VARCHAR。属性值的行为方式与它们来自名为{application} - {profile}的Spring Boot属性文件的行为相同。属性，包括所有的加密和解密，它们将作为后处理步骤（即不在存储库实现中直接）应用。

```
CREATE TABLE `properties` (
  `id` int(11) NOT NULL,
  `key` varchar(50) DEFAULT NULL,
  `value` varchar(500) DEFAULT NULL,
  `application` varchar(50) DEFAULT NULL,
  `profile` varchar(50) DEFAULT NULL,
  `lable` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

复合环境库

在某些情况下，您可能希望从多个环境存储库中提取配置数据。为此，只需在配置服务器的应用程序属性或YAML文件中启用多个配置文件即可。例如，如果您要从Git存储库以及SVN存储库中提取配置数据，那么您将为配置服务器设置以下属性。

```
spring:
  profiles:
    active: git, svn
  cloud:
    config:
      server:
        svn:
          uri: file:///path/to/svn/repo
          order: 2
        git:
          uri: file:///path/to/git/repo
          order: 1
```

除了指定URI的每个repo之外，还可以指定`order`属性。`order`属性允许您指定所有存储库的优先级顺序。`order`属性的数值越低，优先级越高。存储库的优先顺序将有助于解决包含相同属性的值的存储库之间的任何潜在冲突。

> 从环境仓库检索值时的任何类型的故障将导致整个复合环境的故障。

> 当使用复合环境时，重要的是所有repos都包含相同的标签。如果您有类似于上述的环境，并且使用标签`master`请求配置数据，但是SVN repo不包含称为`master`的分支，则整个请求将失败。

自定义复合环境库

除了使用来自Spring Cloud的环境存储库之外，还可以提供自己的`EnvironmentRepository` bean作为复合环境的一部分。要做到这一点，你的bean必须实现`EnvironmentRepository`接口。如果要在复合环境中控制自定义`EnvironmentRepository`的优先级，您还应该实现`Ordered`接口并覆盖`getOrdered`方法。如果您不实现`Ordered`接口，那么您的`EnvironmentRepository`将被赋予最低优先级。

属性覆盖

配置服务器具有“覆盖”功能，允许操作员为应用程序使用普通的Spring Boot钩子不会意外更改的所有应用程序提供配置属性。要声明覆盖，只需将名称/值对的地图添加到`spring.cloud.config.server.overrides`。例如

```
spring:
  cloud:
    config:
      server:
        overrides:
          foo: bar
```

将导致配置客户端的所有应用程序独立于自己的配置读取`foo=bar`。（当然，应用程序可以以任何方式使用Config Server中的数据，因此覆盖不可强制执行，但如果它们是Spring Cloud Config客户端，则它们确实提供有用的默认行为。）

> 通过使用反斜杠（“\”）来转义“$”或“{”，例如`\${app.foo:bar}`解析，可以转义正常的Spring具有“$ {}”的环境占位符到“bar”，除非应用程序提供自己的“app.foo”。请注意，在YAML中，您不需要转义反斜杠本身，而是在您执行的属性文件中配置服务器上的覆盖。

您可以通过在远程存储库中设置标志`spring.cloud.config.overrideNone=true`（默认为false），将客户端中所有覆盖的优先级更改为更为默认值，允许应用程序在环境变量或系统属性中提供自己的值。

健康指标

配置服务器附带运行状况指示器，检查配置的`EnvironmentRepository`是否正常工作。默认情况下，它要求`EnvironmentRepository`应用程序名称为`app`，`default`配置文件和`EnvironmentRepository`实现提供的默认标签。

您可以配置运行状况指示器以检查更多应用程序以及自定义配置文件和自定义标签，例如

```yaml
spring:
  cloud:
    config:
      server:
        health:
          repositories:
            myservice:
              label: mylabel
            myservice-dev:
              name: myservice
              profiles: development
```

您可以通过设置`spring.cloud.config.server.health.enabled=false`来禁用运行状况指示器。

安全

您可以以任何对您有意义的方式（从物理网络安全性到OAuth2承载令牌）保护您的Config Server，并且Spring Security和Spring Boot可以轻松做任何事情。

要使用默认的Spring Boot配置的HTTP Basic安全性，只需在类路径中包含Spring Security（例如通过`spring-boot-starter-security`）。默认值为“user”的用户名和随机生成的密码，这在实践中不会非常有用，因此建议您配置密码（通过`security.user.password`）并对其进行加密（请参阅下文的说明怎么做）>

加密和解密

> **先决条件：**要使用加密和解密功能，您需要在JVM中安装全面的JCE（默认情况下不存在）。您可以从Oracle下载“Java加密扩展（JCE）无限强度管理策略文件”，并按照安装说明（实际上将JRE lib / security目录中的2个策略文件替换为您下载的文件）。

如果远程属性源包含加密内容（以`{cipher}`开头的值），则在通过HTTP发送到客户端之前，它们将被解密。这种设置的主要优点是，当它们“静止”时，属性值不必是纯文本（例如在git仓库中）。如果值无法解密，则从属性源中删除该值，并添加具有相同键的附加属性，但以“无效”作为前缀。和“不适用”的值（通常为“”）。这主要是为了防止密码被用作密码并意外泄漏。

如果要为config客户端应用程序设置远程配置存储库，可能会包含一个`application.yml`，例如：

application.yml

```yaml
spring:
  datasource:
    username: dbuser
    password: '{cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ'
```

.properties文件中的加密值不能用引号括起来，否则不会解密该值：

application.properties

```
spring.datasource.username: dbuser
spring.datasource.password: {cipher}FKSAJDFGYOS8F7GLHAKERGFHLSAJ
```

您可以安全地将此纯文本推送到共享git存储库，并且保密密码。

服务器还暴露了`/encrypt`和`/decrypt`端点（假设这些端点将被保护，并且只能由授权代理访问）。如果您正在编辑远程配置文件，可以使用Config Server通过POST到`/encrypt`端点来加密值，例如

```
$ curl localhost:8888/encrypt -d mysecret
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
```

逆向操作也可通过`/decrypt`获得（如果服务器配置了对称密钥或全密钥对）：

注意 如果要加密的值具有需要进行URL编码的字符，则应使用`--data-urlencode`选项`curl`来确保它们已正确编码。

```
$ curl localhost:8888/decrypt -d
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

小费 如果您使用curl进行测试，则使用`--data-urlencode`（而不是`-d`）或设置显式`Content-Type: text/plain`，以确保在有特殊字符时正确地对数据进行编码（'+'特别是棘手）。

将加密的值添加到`{cipher}`前缀，然后再将其放入YAML或属性文件中，然后再提交并将其推送到远程可能不安全的存储区。

`/encrypt`和`/decrypt`端点也都接受`/*/{name}/{profiles}`形式的路径，当客户端调用到主环境资源时，可以用于每个应用程序（名称）和配置文件控制密码。

注意 为了以这种细微的方式控制密码，您还必须提供一种`TextEncryptorLocator`类型的`@Bean`，可以为每个名称和配置文件创建不同的加密器。默认提供的不会这样做（所有加密使用相同的密钥）。

`spring`命令行客户端（安装了Spring Cloud CLI扩展）也可以用于加密和解密，例如

```
$ spring encrypt mysecret --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
$ spring decrypt --key foo
682bc583f4641835fa2db009355293665d2647dade3375c0ee201de2a49f7bda
mysecret
```

要在文件中使用密钥（例如用于加密的RSA公钥），使用“@”键入键值，并提供文件路径，例如

```
$ spring encrypt mysecret --key @${HOME}/.ssh/id_rsa.pub
AQAjPgt3eFZQXwt8tsHAVv/QHiY5sI2dRcR+...
```

关键参数是强制性的（尽管有一个`--`前缀）

调试中遇到的问题

问题一 报错

Spring Cloud Config配置文件加解密坑爹的问题

> curl http://localhost:8888/encrypt -d 123
> {"description":"No key was installed for encryption service","status":"NO_KEY"}

遇到这个问题有三个原因

```
未配置JCE
未设置key/keystore
spring cloud bug
```

> 网上查找资料，有2个帖子都说是上面的问题，但找了好久还是不对，突然想到一个问题是不是配置文件加载顺序啊，把原来的application.yml->bootstarp.yml 重启运行果断好用。

```
##加密curl -X POST localhost:8888/encrypt -d  foo
##解密 curl -X POST localhost:8888/decrypt -d
2c3400b2bc95a50a1edbb2a97271f900f91b573632b4866672cd347dadf7730a
```

浏览器访问127.0.0.1:8888/test/master

问题二 特殊符号加密

当被加密内容包含一些诸如`=`、`+`这些特殊字符的时候，使用上篇文章中提到的类似这样的命令`curl localhost:7001/encrypt -d`去加密和解密的时候，会发现特殊字符丢失的情况。

比如下面这样的情况：

```
$ curl localhost:7001/encrypt -d eF34+5edo=
a34c76c4ddab706fbcae0848639a8e0ed9d612b0035030542c98997e084a7427
$ curl localhost:7001/decrypt -d
a34c76c4ddab706fbcae0848639a8e0ed9d612b0035030542c98997e084a7427
eF34 5edo
```

可以看到，经过加密解密之后，又一些特殊字符丢失了。由于之前在这里也小坑了一下，所以抽空写出来分享一下，给遇到同样问题的朋友，希望对您有帮助。

其实关于这个问题的原因在官方文档中是有具体说明的，只能怪自己太过粗心了，具体如下：

> If you are testing like this with curl, then use `--data-urlencode` (instead of -d) or set an explicit `Content-Type: text/plain` to make sure curl encodes the data correctly when there are special characters (‘+’ is particularly tricky).

所以，在使用`curl`的时候，正确的姿势应该是：

```
$ curl localhost:7001/encrypt -H 'Content-Type:text/plain' --data-urlencode "eF34+5edo="
335e618a02a0ff3dc1377321885f484fb2c19a499423ee7776755b875997b033

$ curl localhost:7001/decrypt -H 'Content-Type:text/plain' --data-urlencode
"335e618a02a0ff3dc1377321885f484fb2c19a499423ee7776755b875997b033"
eF34+5edo=
```

自定义加密算法

那么，如果我们自己写工具来加密解密的时候怎么玩呢？下面举个`OkHttp`的例子，以供参考：

```
private String encrypt(String value) {
    String url = "http://localhost:7001/encrypt";
    Request request = new Request.Builder()
            .url(url)
            .post(RequestBody.create(MediaType.parse("text/plain"), value.getBytes()))
            .build();

    Call call = okHttpClient.newCall(request);
    Response response = call.execute();
    ResponseBody responseBody = response.body();
    return responseBody.string();
}

private String decrypt(String value) {
    String url = "http://localhost:7001/decrypt";
    Request request = new Request.Builder()
            .url(url)
            .post(RequestBody.create(MediaType.parse("text/plain"), value.getBytes()))
            .build();

    Call call = okHttpClient.newCall(request);
    Response response = call.execute();
    ResponseBody responseBody = response.body();
    return responseBody.string();
}
```

密钥管理

Config Server可以使用对称（共享）密钥或非对称密钥（RSA密钥对）。非对称选择在安全性方面是优越的，但是使用对称密钥往往更方便，因为它只是配置的一个属性值。

要配置对称密钥，您只需要将`encrypt.key`设置为一个秘密字符串（或使用环境变量`ENCRYPT_KEY`将其从纯文本配置文件中删除）。

要配置非对称密钥，您可以将密钥设置为PEM编码的文本值（`encrypt.key`），也可以通过密钥库设置密钥（例如由JDK附带的`keytool`实用程序创建）。密钥库属性为`encrypt.keyStore.*`，`*`等于

- `location`（a `Resource`位置），
- `password`（解锁密钥库）和
- `alias`（以识别商店中使用的密钥）。

使用公钥进行加密，需要私钥进行解密。因此，原则上您只能在服务器中配置公钥，如果您只想进行加密（并准备使用私钥本地解密值）。实际上，您可能不想这样做，因为它围绕所有客户端传播密钥管理流程，而不是将其集中在服务器中。另一方面，如果您的配置服务器真的相对不安全，并且只有少数客户端需要加密的属性，这是一个有用的选项。

非对称加密创建用于测试的密钥库

要创建一个测试密钥库，可以这样做：执行shell 命令

```
keytool -genkeypair -alias mytestkey -keyalg RSA \
  -dname "CN=Web Server,OU=Unit,O=Organization,L=City,S=State,C=US" \
  -keypass changeme -keystore server.jks -storepass letmein
```

- 参数说明

~$ keytool -help
密钥和证书管理工具

命令:

-certreq 生成证书请求
-changealias 更改条目的别名
-delete 删除条目
-exportcert 导出证书
-genkeypair 生成密钥对
-genseckey 生成密钥
-gencert 根据证书请求生成证书
-importcert 导入证书或证书链
-importpass 导入口令
-importkeystore 从其他密钥库导入一个或所有条目
-keypasswd 更改条目的密钥口令
-list 列出密钥库中的条目
-printcert 打印证书内容
-printcertreq 打印证书请求的内容
-printcrl 打印 CRL 文件的内容
-storepasswd 更改密钥库的存储口令

将server.jks文件放在类路径中（例如），然后放到您的配置服务器的bootstrap.yml文件中：

```
encrypt:
  keyStore:
    location: classpath:/server.jks  ##src/main/resouces 目录下
    password: letmein
    alias: mytestkey
    secret: changeme
```

在命令行执行，可查看效果

```
curl -X POST localhost:8888/encrypt -d  foo
```

AQBMAnN3jA2mMU/MPY0Kl7aSWZHvTHT62+viP0QFj681kv13Ey2K+bdR6Uuh2wwdVwG4UMY+mfpojgg/ReB
FzN2yO9wIOzMaDOARHj6HOAIj3CnaiGr5MnbYuXMkj80xx7zX4Nfssb7p92oRbisbmphuFICAzMr3oImvkMt2hj
A+R06fQO9OV5ffzvsdGZAsRGdwwe2wMymdng5dM8jhgKp1eZBxjcsJhc+uOcRgbi3vkYjT3k7yA/M6N6/+1upfj4
EioJBMwN6Szz+hvBDCl9w8rUqH0v6EslZws0X+3d5RnydwC/W3UBL9w8ax9Ln5mDLcdiQLZMGxftOLWA8tB+9
A/ooDV9Xpdbtsx+zVfuB7MD56Oisf1J5jK7cem5CXdho

使用多个键和键回转

除了加密属性值中的`{cipher}`前缀之外，配置服务器在（Base64编码）密文开始前查找`{name:value}`前缀（零或多个）。密钥被传递给`TextEncryptorLocator`，它可以执行找到密码的`TextEncryptor`所需的任何逻辑。如果配置了密钥库（`encrypt.keystore.location`），默认定位器将使用“key”前缀提供的别名，即使用如下密码查找存储中的密钥：

```
foo:
  bar: `{cipher}{key:testkey}...`
```

定位器将寻找一个名为“testkey”的键。也可以通过前缀中的`{secret:…}`值提供一个秘密，但是如果不是默认值，则使用密钥库密码（这是您在构建密钥库时获得的，并且不指定密码）。如果你**这样做**提供一个秘密建议你也加密使用自定义`SecretLocator`的秘密。

如果密钥只用于加密几个字节的配置数据（即它们没有在其他地方使用），则密码转换几乎不是必需的，但是如果存在安全漏洞，有时您可能需要更改密钥实例。在这种情况下，所有客户端都需要更改其源配置文件（例如，以git格式），并在所有密码中使用新的`{key:…}`前缀，当然事先检查密钥别名在配置服务器密钥库中是否可用。

> 如果要让Config Server处理所有加密以及解密，也可以将`{name:value}`前缀添加到发布到`/encrypt`端点的明文中。

服务加密Properties

有时您希望客户端在本地解密配置，而不是在服务器中进行配置。在这种情况下，您仍然可以拥有/encrypt和/decrypt端点（如果您提供`encrypt.*`配置来定位密钥），但是您需要使用`spring.cloud.config.server.encrypt.enabled=false`明确地关闭传出属性的解密。如果您不关心端点，那么如果您既不配置密钥也不配置使能的标志，则应该起作用。

提供替代格式

来自环境端点的默认JSON格式对于Spring应用程序的消费是完美的，因为它直接映射到`Environment`抽象。如果您喜欢，可以通过向资源路径（“.yml”，“.yaml”或“.properties”）添加后缀来使用与YAML或Java属性相同的数据。这对于不关心JSON端点的结构的应用程序或其提供的额外的元数据的应用程序来说可能是有用的，例如，不使用Spring的应用程序可能会受益于此方法的简单性。

YAML和属性表示有一个额外的标志（作为一个布尔查询参数`resolvePlaceholders`提供）），以标示Spring `${…}`形式的源文档中的占位符，应在输出中解析可能在渲染之前。对于不了解Spring占位符惯例的消费者来说，这是一个有用的功能。

> 使用YAML或属性格式存在局限性，主要是与元数据的丢失有关。JSON被构造为属性源的有序列表，例如，名称与源相关联。即使源的起源具有多个源，并且原始源文件的名称丢失，YAML和属性表也合并成一个映射。YAML表示不一定是后台存储库中YAML源的忠实表示：它是由平面属性源的列表构建的，并且必须对键的形式进行假设。

提供纯文本

您的应用程序可能需要通用的纯文本配置文件，而不是使用`Environment`抽象（或YAML中的其他替代表示形式或属性格式）。配置服务器通过`/{name}/{profile}/{label}/{path}`附加的端点提供这些服务，其中“name”，“profile”和“label”的含义与常规环境端点相同，但“path”是文件名（例如`log.xml`）。此端点的源文件位于与环境端点相同的方式：与属性或YAML文件相同的搜索路径，而不是聚合所有匹配的资源，只返回匹配的

第一个。

找到资源后，使用正确格式（`${…}`）的占位符将使用有效的`Environment`解析为应用程序名称，配置文件和标签提供。以这种方式，资源端点与环境端点紧密集成。例如，如果您有一个GIT（或SVN）资源库的布局：

```
application.yml
nginx.conf
```

其中`nginx.conf`如下所示：

```
server {
    listen              80;
    server_name         ${nginx.server.name};
}
```

和`application.yml`这样：

```
nginx:
  server:
    name: example.com
---
spring:
  profiles: development
nginx:
  server:
    name: develop.com
```

那么`/foo/default/master/nginx.conf`资源如下所示：

```
server {
    listen              80;
    server_name         example.com;
}
```

和`/foo/development/master/nginx.conf`这样：

```
server {
    listen              80;
    server_name         develop.com;
}
```

> 注意 就像环境配置的源文件一样，“配置文件”用于解析文件名，因此，如果您想要一个特定于配置文件的文件，则`/*/development/*/logback.xml`将由一个名为`logback-development.xml`的文件解析（优先于`logback.xml`）。

嵌入配置服务器

配置服务器最好作为独立应用程序运行，但如果需要，可以将其嵌入到另一个应用程序中。只需使用`@EnableConfigServer`注释。在这种情况下可以使用的可选属性是`spring.cloud.config.server.bootstrap`，它是一个标志，表示服务器应该从其自己的远程存储库配置自身。该标志默认关闭，因为它可能会延迟启动，但是当嵌入在另一个应用程序中时，以与其他应用程序相同的方式初始化是有意义的。

> 应该是显而易见的，但请记住，如果您使用引导标志，配置服务器将需要在`bootstrap.yml`中配置其名称和存储库URI。

要更改服务器端点的位置，您可以（可选）设置`spring.cloud.config.server.prefix`，例如“/ config”，以提供前缀下的资源。前缀应该开始但不以“/”结尾。它被应用于配置服务器中的`@RequestMappings`（即Spring Boot前缀`server.servletPath`和`server.contextPath`）之下。

如果您想直接从后端存储库（而不是从配置服务器）读取应用程序的配置，这基本上是一个没有端点的嵌入式配置服务器。如果不使用`@EnableConfigServer`注释（仅设置`spring.cloud.config.server.bootstrap=true`），则可以完全关闭端点。

推送通知和Spring Cloud Bus

许多源代码存储库提供程序（例如Github，Gitlab或Bitbucket）将通过webhook通知您存储库中的更改。您可以通过提供商的用户界面将webhook配置为URL和一组感兴趣的事件。例如，Github将使用包含提交列表的JSON主体和“X-Github-Event”等于“push”的头文件发送到webhook。如果在`spring-cloud-config-monitor`库中添加依赖关系并激活配置服务器中的Spring Cloud Bus，则启用“/ monitor”端点。

当Webhook被激活时，配置服务器将发送一个`RefreshRemoteApplicationEvent`针对他认为可能已经改变的应用程序。变更检测可以进行策略化，但默认情况下，它只是查找与应用程序名称匹配的文件的更改（例如，“foo.properties”针对的是“foo”应用程序，“application.properties”针对所有应用程序）。如果要覆盖该行为的策略是`PropertyPathNotificationExtractor`，它接受请求标头和正文作为参数，并返回更改的文件路径列表。

默认配置与Github，Gitlab或Bitbucket配合使用。除了来自Github，Gitlab或Bitbucket的JSON通知之外，您还可以通过使用表单编码的身体参数`path={name}`通过POST为“/ monitor”来触发更改通知。这将广播到匹配“{name}”模式的应用程序（可以包含通配符）。

> 只有在配置服务器和客户端应用程序中激活`spring-cloud-bus`时才会传送`RefreshRemoteApplicationEvent`。

> 默认配置还检测本地git存储库中的文件系统更改（在这种情况下不使用webhook，但是一旦编辑配置文件，将会播放刷新）。

- 需要下载(RabbitMq)[www.rabbitmq.com/download.html]
  - 官网下载，安装步骤省略
- 启动mq
  - sudo service rabbitmq-server start
- 可安装管理界面图形化的地址 www.rabbitmq.com/management.html
- 默认访问地址127.0.0.1:15672 帐号 guest 密码 guest

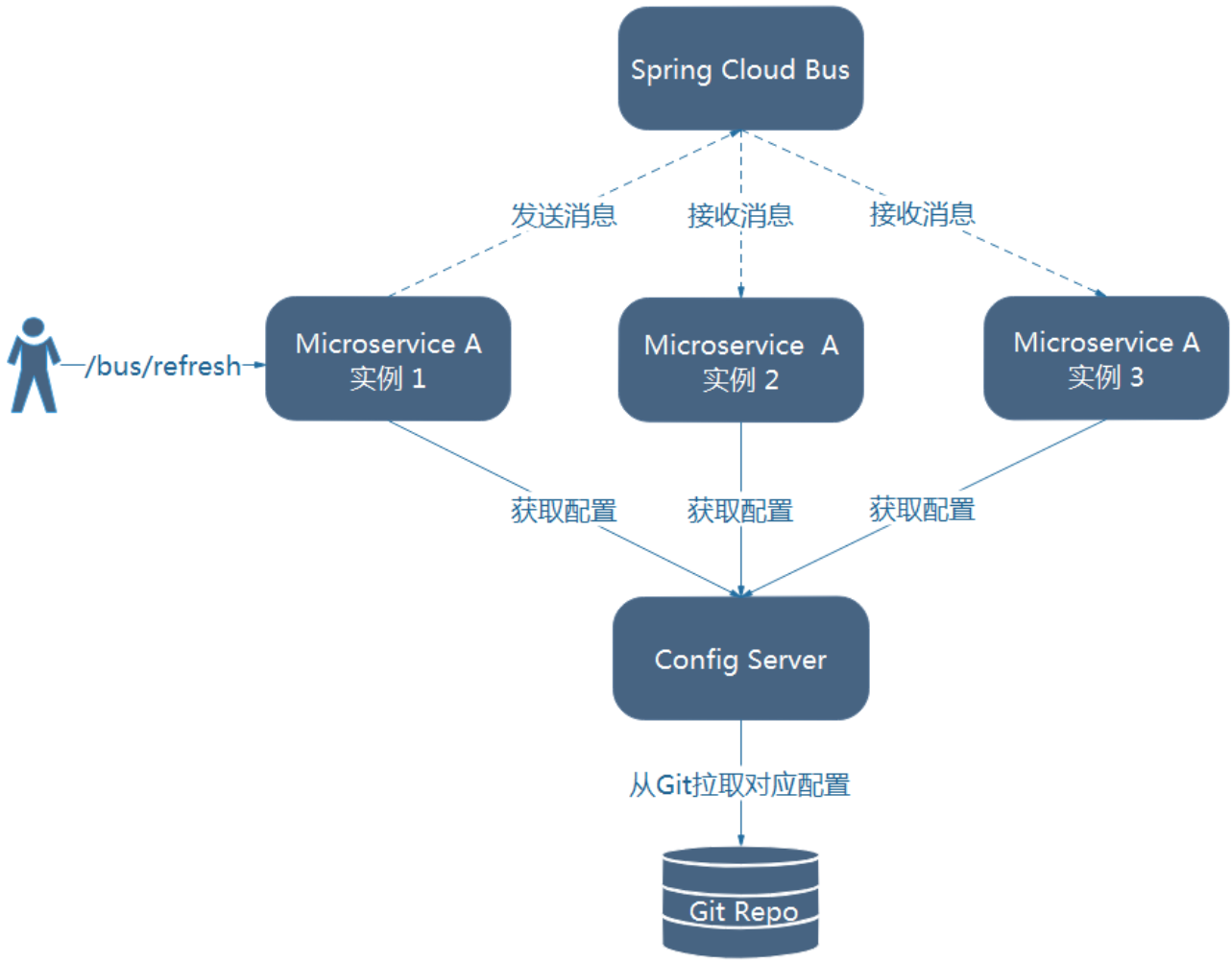
修改git 仓库中的配置信息，在CMD 中执行如下命令也可自动刷新

```
curl -X POST  http://127.0.0.1:9999//bus/refresh
```

自动刷新

使用`/refresh`端点手动刷新配置，但是如果所有微服务节点的配置都需要手动去刷新的话，那必然是一个繁琐的工作，并且随着系统的不断扩张，会变得越来越难以维护。因此，实现配置的自动刷新是很有必要的，本节我们讨论使用Spring Cloud Bus实现配置的自动刷新。

Spring Cloud Bus提供了批量刷新配置的机制，它使用轻量级的消息代理（例如RabbitMQ、Kafka等）连接分布式系统的节点，这样就可以通过Spring Cloud Bus广播配置的变化或者其他的管理指令。使用Spring Cloud Bus后的架构如图9-2所示。



使用Spring Cloud Bus的架构图

由图可知，微服务A的所有实例通过消息总线连接到了一起，每个实例都会订阅配置更新事件。当其中一个微服务节点的/bus/refresh端点被请求时，该实例就会向消息总线发送一个配置更新事件，其他实例获得该事件后也会更新配置。

下面我们以RabbitMQ为例，为大家讲解如何使用Spring Cloud Bus实现配置的自动刷新。

(1) 安装RabbitMQ。RabbitMQ的安装非常简单，本书不再赘述。

(2) 复制项目`microservice-config-client-refresh`，将ArtifactId修改为`microservice-config-client-refresh-cloud-bus`。

(3) 为项目添加`spring-cloud-starter-bus-amqp`的依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

(4) 在bootstrap.yml中添加以下内容：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样代码就改造完成了。

测试

(1) 启动microservice-config-server

(2) 启动microservice-config-client-refresh-cloud-bus，可发现此时控制台打印类似于以下的内容：

```
[           main] o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "
{[/bus/refresh],methods=[POST]}" onto public void
org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)
```

说明此时有一个`/bus/refresh`端点。

(3) 将microservice-config-client-refresh-cloud-bus的端口改成8082，再启动一个节点。

(4) 访问http://localhost:8081/profile ，可获得结果：dev-1.0。

(4) 将git仓库中的microservice-foo-dev.properties文件内容改为`profile=dev-1.0-bus`

(5) 发送POST请求到其中一个Config Client节点的的/bus/refresh端点，例如：

```
curl -X POST http://localhost:8081/bus/refresh
```

(6) 访问两个Config Client节点的/profile端点，会发现两个节点都会返回`dev-1.0-bus` ，说明配置内容已被刷新。

借助Git仓库的WebHook，我们就可轻松实现配置的自动刷新。如图9-3所示。

图9-3 Git WebHooks设置

局部刷新

某些场景下（例如灰度发布），我们可能只想刷新部分微服务的配置，此时可通过/bus/refresh端点的destination参数来定位要刷新的应用程序。

例如：`/bus/refresh?destination=customers:9000` ，这样消息总线上的微服务实例就会根据destination参数的值来判断是否需要要刷新。其中，`customers:9000` 指的是各个微服务的ApplicationContext ID。

destination参数也可以用来定位特定的微服务。例如：`/bus/refresh?destination=customers:**` ，这样就可以触发customers微服务所有实例的配置刷新。

扩展阅读：关于ApplicationContext ID

默认情况下，ApplicationContext ID是spring.application.name:server.port，详见`org.springframework.boot.context.ContextIdApplicationContextInitializer.getApplicationId(ConfigurableEnvironment)` 方法。

http://www.itmuch.com/spring-cloud-code-read/spring-cloud-code-read-spring-cloud-bus/

架构改进---刷新config-server

在前面的示例中，我们通过请求某个微服务的/bus/refresh端点的方式来实现配置刷新，但这种方式并不优雅。原因如下：

(1) 打破了微服务的职责单一性。微服务本身是业务模块，它本不应该承担配置刷新的职责。

(2) 破坏了微服务各节点的对等性。

(3) 有一定的局限性。例如，微服务在迁移时，它的网络地址常常会发生变化，此时如果想要做到自动刷新，那就不得不修改WebHook的配置。

我们不妨改进一下我们的架构。

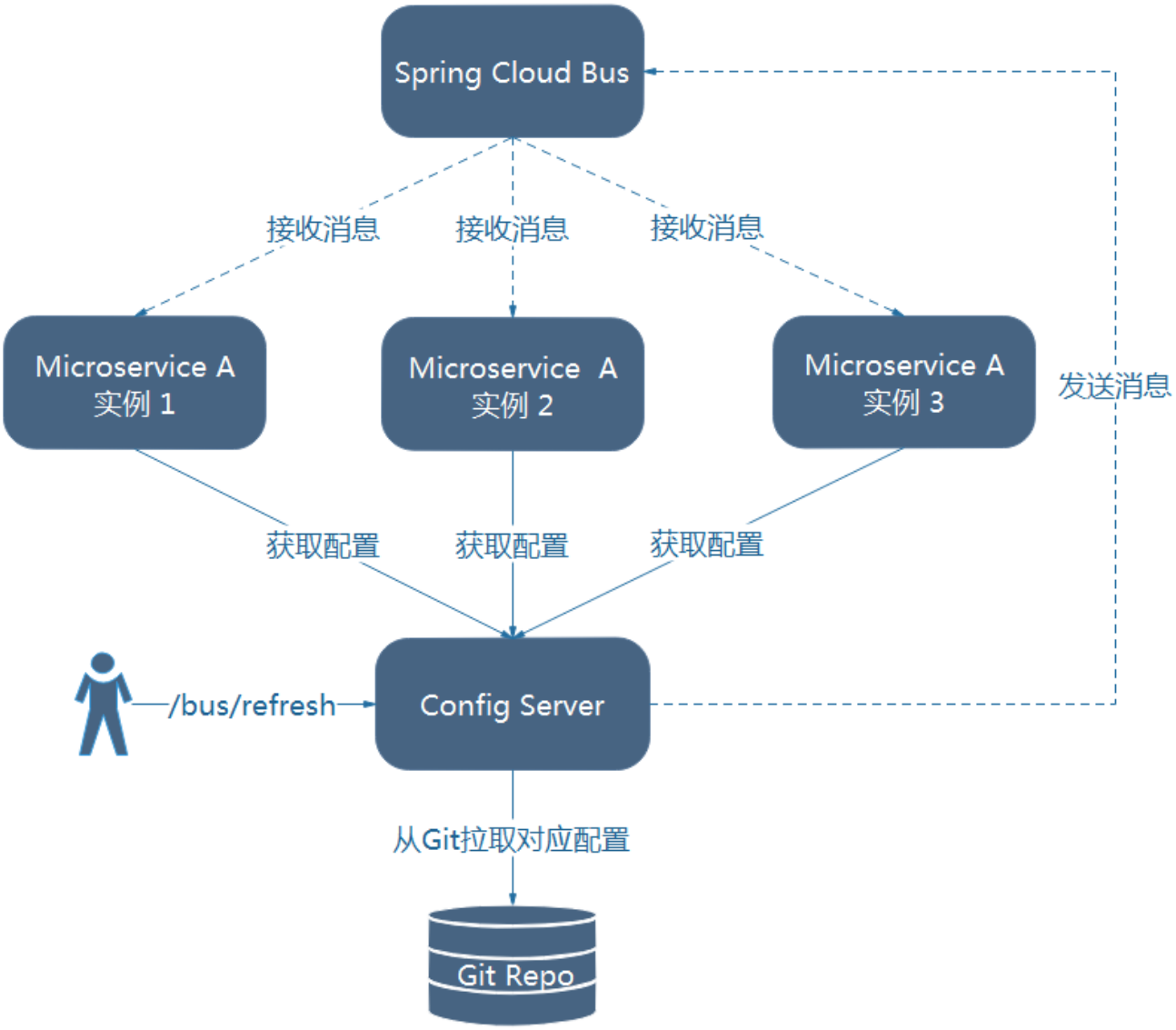


图9-4 使用Spring Cloud Bus的架构图

如图9-4，我们将Config Server也加入到消息总线中，并使用Config Server的/bus/refresh端点来实现配置的刷新。这样，各个微服务只需要关注自身的业务，而不再承担配置刷新的职责。代码详见`microservice-config-server-refresh-cloud-bus`。

跟踪总线事件

一些场景下，我们可能希望知道Spring Cloud Bus事件传播的细节。此时，我们可以跟踪总线事件（RemoteApplicationEvent的子类都是总线事件）。

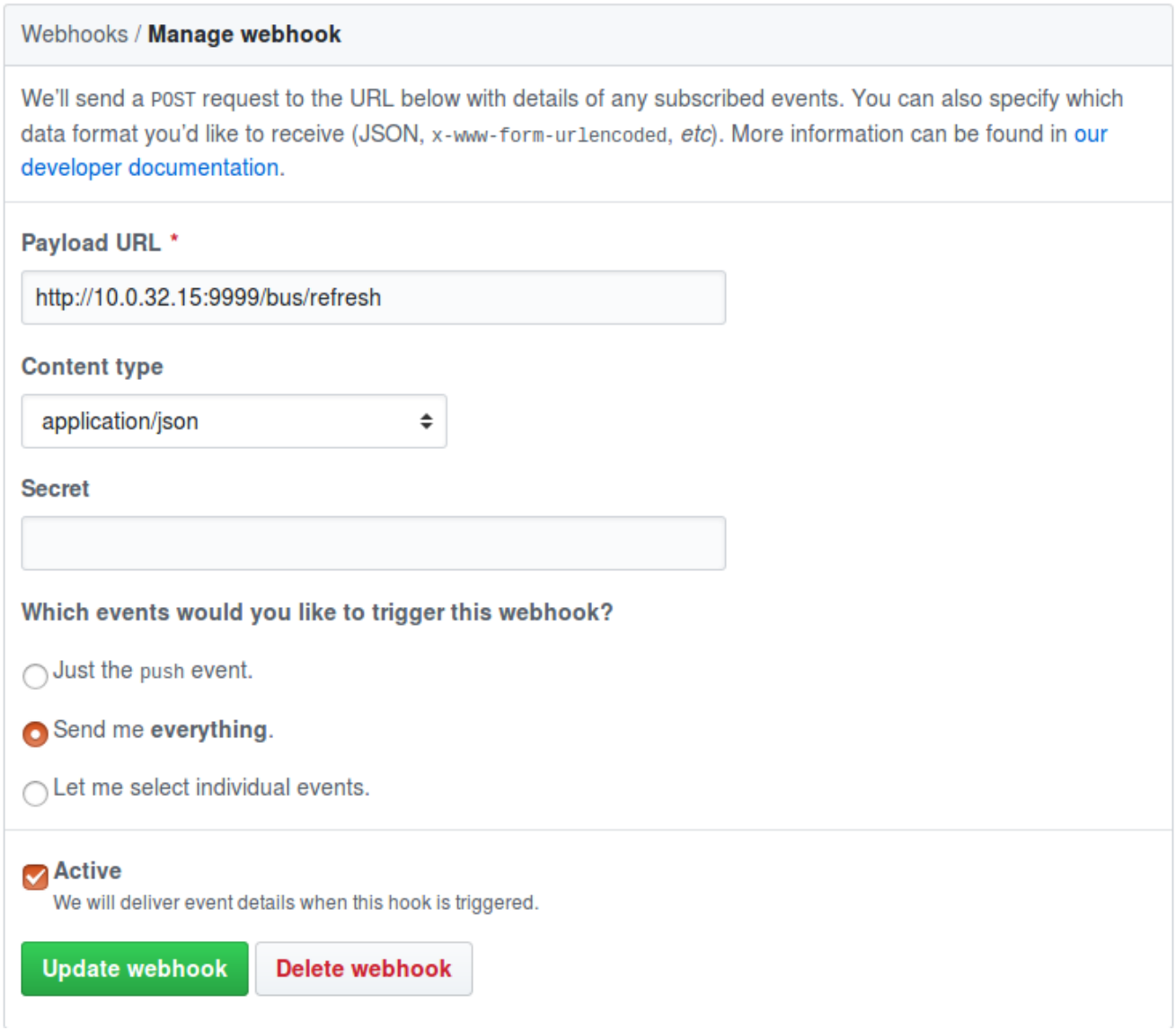
跟踪总线事件非常简单，只需设置`spring.cloud.bus.trace.enabled=true`，这样在/bus/refresh端点被请求后，访问/trace端点就可获得类似如下的结果：

```
{
  "timestamp": 1481098786017,
  "info": {
    "signal": "spring.cloud.bus.ack",
    "event": "RefreshRemoteApplicationEvent",
    "id": "66d172e0-e770-4349-baf7-0210af62ea8d",
    "origin": "microservice-foo:8081",
    "destination": "**"
  }
},{
  "timestamp": 1481098779073,
  "info": {
    "signal": "spring.cloud.bus.sent",
        "type": "RefreshRemoteApplicationEvent",
        "id": "66d172e0-e770-4349-baf7-0210af62ea8d",
        "origin": "microservice-config-server:8080",
        "destination": "**:**"
  }
}...
```


这样，我们就可清晰地知道事件的传播细节。

当服务太多，配置信息太多如何实现自动刷新呢。其实还是需要借助GIT上的Webhooks / 功能

在管理界面配置如下：原理很简单就是通过配置URL 执行定时任务而已



> 额外
>
> pivotal (EMC（惠普收购）+vmware两家公司)
>
> -redis
>
> -spring
>
> rabbitmq
>
> greenplum
>
> reactor
>
> PCF cloud foundry

FAQ

解决Spring Cloud Bus不刷新所有节点的问题及理解"Application Context ID must be unique"

如果同一微服务的多个实例使用的端口相同，当配置修改时，使用Spring Cloud Bus不会刷新全部实例的配置。此时需要配置各个实例的spring.application.index为不同的值。下面我们来分析一下原因。

在Spring Cloud Config上有这么一段：

- Application Context ID must be unique

The bus tries to eliminate processing an event twice, once from the original `ApplicationEvent` and once from the queue. To do this, it checks the sending application context id againts the current application context id. If multiple instances of a service have the same application context id, events will not be processed. Running on a local machine, each service will be on a different port and that will be part of the application context id. Cloud Foundry supplies an index to differentiate. To ensure that the application context id is the unique, set `spring.application.index` to something unique for each instance of a service. For example, in lattice, set `spring.application.index=${INSTANCE_INDEX}` in application.properties (or bootstrap.properties if using configserver).

这段话的意思，大致上是说如果相同微服务的多个实例，使用的是相同的端口时，需要配置`spring.application.index`属性，本文来分析一下为什么。

(1) 我们知道定位Spring Boot的问题，往往可以从配置开始。按照这个思路，先找到spring.application.index 所在的类ContextIdApplicationContextInitializer。至于怎么找到的，可以看这里：http://docs.spring.io/spring-boot/docs/1.4.2.RELEASE/reference/htmlsingle/#common-application-properties ，搜索spring.application.index即可。

(2) 在`org.springframework.boot.context.ContextIdApplicationContextInitializer`类的getApplicationId() 方法中，有类似以下的内容：

```
private String getApplicationId(ConfigurableEnvironment environment) {
  String name = environment.resolvePlaceholders(this.name);
  String index = environment.resolvePlaceholders(INDEX_PATTERN);
  String profiles = StringUtils
      .arrayToCommaDelimitedString(environment.getActiveProfiles());
  if (StringUtils.hasText(profiles)) {
    name = name + ":" + profiles;
  }
  if (!"null".equals(index)) {
    name = name + ":" + index;
  }
  return name;
}
```

其中，name的表达式如下：

`${spring.application.name:${vcap.application.name:${spring.config.name:application}}}` ，也就是配置的`spring.application.name` （以主流方式为例，当然也可能是spring.config.name ）。

而index的表达式是：

`${vcap.application.instance_index:${spring.application.index:${server.port:${PORT:null}}}}`

也就是如果什么都不配置，就取server.port。

综上，如果什么都不配置，那么getApplicationId返回的是`${spring.application.name}:${server.port}`

(3) 在Spring Cloud Bus中的`org.springframework.cloud.bus.ServiceMatcher`有以下代码：

```
public boolean isFromSelf(RemoteApplicationEvent event) {
  String originService = event.getOriginService();
  String serviceId = getServiceId();
  return this.matcher.match(originService, serviceId);
}

public boolean isForSelf(RemoteApplicationEvent event) {
  String destinationService = event.getDestinationService();
  return (destinationService == null || destinationService.trim().isEmpty() ||
this.matcher
      .match(destinationService, getServiceId()));
}

public String getServiceId() {
  return this.context.getId();
}
```

从代码可知，如果什么都不设置，并且相同微服务的多个实例使用的是相同的端口的话，那么isFromSelf将会返回true。

(4) 在org.springframework.cloud.bus.BusAutoConfiguration.acceptRemote(RemoteApplicationEvent)中的代码：

```
@StreamListener(SpringCloudBusClient.INPUT)
public void acceptRemote(RemoteApplicationEvent event) {
  if (event instanceof AckRemoteApplicationEvent) {
    if (this.bus.getTrace().isEnabled() && !this.serviceMatcher.isFromSelf(event)
        && this.applicationEventPublisher != null) {
      this.applicationEventPublisher.publishEvent(event);
    }
    // If it's an ACK we are finished processing at this point
    return;
  }
  if (this.serviceMatcher.isForSelf(event)
      && this.applicationEventPublisher != null) {
    if (!this.serviceMatcher.isFromSelf(event)) {
      this.applicationEventPublisher.publishEvent(event);
    }
    if (this.bus.getAck().isEnabled()) {
      AckRemoteApplicationEvent ack = new AckRemoteApplicationEvent(this,
          this.serviceMatcher.getServiceId(),
          this.bus.getAck().getDestinationService(),
          event.getDestinationService(), event.getId(), event.getClass());
      this.cloudBusOutboundChannel
          .send(MessageBuilder.withPayload(ack).build());
      this.applicationEventPublisher.publishEvent(ack);
    }
  }
  if (this.bus.getTrace().isEnabled() && this.applicationEventPublisher != null) {
    // We are set to register sent events so publish it for local consumption,
    // irrespective of the origin
    this.applicationEventPublisher.publishEvent(new SentApplicationEvent(this,
        event.getOriginService(), event.getDestinationService(),
        event.getId(), event.getClass()));
  }
}
```

看到这段代码，原因已经一目了然了。

Github上的相关issue：https://github.com/spring-cloud/spring-cloud-bus/issues/18 。

> blog.didispace.com/Spring-Cloud-Config-Server-ip-change-problem/