# Spring Cloud Config：分布式配置中心

- 分布式配置中心
- 用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持， 它分为服务端与客户端两个部分
    - 其中服务端也称为分布式配置中心， 它是一个独立的微服务应用， 用来连接配置仓库并为客户端提供获取配置信息、 加密/解密信息等访问接口
    - 客户端则是微服务架构中的各个微服务应用或基础设施，它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息

## 服务端

- `config-server,` 引入`config-server`依赖
- 主类添加 @EnableConfigServer 注解， 开启 Spring Cloud Config 的服务端功能

```
# application.properties
spring.application.name=config-server
server.port=7001

# master分支 http://localhost:7001/woody/prod
# config-label-test 分支 http://localhost:7001/woody/prod/config-label-test
spring.cloud.config.server.git.uri=https://github.com/xiaojieWoody/SpringCloudConfigServer/
# 也支持使用{application}、{profile}和{label}占位符
spring.cloud.config.server.git.search-paths=springcloud_in_test/config-repo
spring.cloud.config.server.git.username=xiaojieWoody
spring.cloud.config.server.git.password=Iam0108dyj

## 本地 配置, 客户端访问有问题
# spring.profiles.active=native
# spring.cloud.config.server.native.search-locations=/Users/dingyuanjie/Documents/code/myself/local-config

eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/
```

- 配置规则

```
# git上 master分支
woody.properties        # from=git-default-1.0
woody-dev.properties    # from=git-dev-1.0
woody-test.properties   # from=git—test-1.0
woody-prod.properties   # frorn=git-prod-1.0
```

```
# config-label-test 分支
woody.properties         # from=git-default-2.0
woody-dev.properties     # from=git-dev-2.0
woody-test.properties    # from=git—test-2.0
woody-prod.properties    # frorn=git-prod-2.0

# 访问配置信息的URL与配置文件的映射关系
/{application}/{profile} [/{label}]
/{application}—{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
#  {label}对应Git上不同的分支,默认为 master
#  比如，要访问 config-label-test 分支， woody 应用的 prod 环境
#       http://localhost:7001/woody/prod/config-label-test
```

    - 配置服务器在从 Git 中获取配置信息后，会存储一份在config-server 的文件系统中，实质上config-server是通过 git clone 命令将配置内容复制了一份在本地存储，然后读取这些内容并返回给微服务应用进行加载
    - 通过 Git 在本地仓库暂存，可以有效防止当 Git 仓库出现故障而引起无法加载配置信息的情况
- 占位符配置URI
    - {application}、{profile}、{label}这些占位符除了用于标识配置文件的规则之外， 还可以用于ConfigServer 中对 Git 仓库地址的URI配置

      ```
      spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}
      spring.cloud.config.server.git.username=username
      spring.cloud.config.server.git.password=password
      ```

    - {application}代表了应用名， 所以当客户端应用向ConfigServer发起获取配置的请求时，ConfigServer会根据客户端的spring.application.name 信息来填充{application}占位符以定位配置资源的存储位置， 从而实现根据微服务应用的属性动态获取不同位置的配置
    - 如果Git的分支和标签名包含"/", 那么{label}参数在HTTP的URL中应该使用 "()" 来替代， 以避免改变了URI含义， 指向到其他的URI资源
    - 当使用 Git 作为配置中心来存储各个微服务应用配置文件的时候， 该功能会变得非常有用，通过在URI中使用占位符可以帮助规划和实现通用的仓库配置
        - 代码库：使用服务名作为Git仓库名称， 比如会员服务的代码库 http://git.oschina.net/didispace/mernber-service
        - 配置库：使用服务名加上-config后缀作为Gti 仓库名称， 比如上面会员服务对应的配置库地址位置http://git.oschina .net/didispace/member-service-config
        - 就可以使用 spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}-config 配置， 来同时匹配多个不 同服务的配置仓库
- 配置多个仓库
- 本地仓库

    - 在使用了Git或SYN仓库之后，文件都会在ConfigServer的本地文件系统中存储一份，这些文件默认会被存 储于以config-repo 为前缀的临时目录中，比如名为/tmp/config-repo-<随机数>的目录
    - 最好的办法就是指定一个固定的位置来存储这些重要信息。只需要通过 spring.cloud.config.server.git.basedir 或 spring.cloud.config.server.svn.basedir来配置一个准备好的目录即可
- 本地文件系统
    - 只需要设置属性 spring.profiles.active= native, ConfigServer会默认从应用的 src/main/resource 目录下搜索配置文件。
    - 如果需要指定搜索配置文件的路径， 可以通过 spring.cloud.config.server.native.searchLocations 属性来指定具体的配 置文件位置
- 属性覆盖
    - 为所有的应用提供配置属性，只需要通过 spring.cloud.config.server.overrides 属性来设置键值对 的参数， 这些参数会以 Map 的方式加载到客户端的配置中

      ```
      spring.cloud.config.server.overrides.name=didi
      spring.cloud.config.server.overrides.from=shanghai
      ```

    - 通过该属性配置的参数，不会被 Spring Cloud 的客户端修改，并且 Spring Cloud 客户端从 Config Server 中获取配置信息时，都会取得这些配置信息。 利用该特性可以方便地为Spring Cloud 应用配置一些共同属性或是默认属性
    - 当然，这些属性并非强制的，可以通过改变客户端中更高优先级的配置方式(比如，配置环境变量或是系统属性)，来选择是 否使用 Config Server 提供的默认值。
- 安全保护
    - 加入`spring-boot-starter-security`

      ```
      # 服务端和客户端 都配置
      security.user.name=user
      security.user.password=37cc5635-559b-4e6f-b633-7e932b813f73
      ```

- 加密解密

  ```
  spring.datasource.username=didi
  spring.datasource.password=
  {cipher}dba6505baa8ld78bd08799d8d4429de499bd4c2053c05f029e7cfbf143695f5b
  ```

    - 在Spring Cloud Config中通过在属性值前使用{cipher}前缀来标注该内容是一个加密值， 当微服务客户端加载配置时，配置中心会自动为带有 {cipher} 前缀的值进行解密
    - 前提：需要在配置中心的运行环境中安装不限长度的 JCE 版本
- 高可用配置
    - 传统模式
        - 将所有的 Config Server 都指向同一个Git 仓库，这样所有的配置内容就通过统一的共享文件系统来维护。而客户端在指定 Config Server 位置时，只需要配置 Config Server上层的负载均衡设备地址即可

- 服务模式
  - 将 Config Server 作为一个普通的微服务应用，纳入 Eureka 的服务治理体系中，微服务应用就可 以通过配置中心的服务名来获取配置信息
  - 因为对于服务端的负载均衡配置和客户端的配置中心指定都通过服务治理机制一并解决了， 既实现了高可用， 也实现了自维护。 由于这部分的实现需要客户端的配合

# 客户端映射

- `config-client`，引入`web`，`config`

```
# bootstrap.properties
## 对应 ${application}
spring.application.name=woody
## 对应 ${profile}
spring.cloud.config.profile=dev
## 对应 ${label}
spring.cloud.config.label=master
## 对应 config-server 配置中心地址
# 只有当我们配置spring.cloud.config.uri 的时候， 客户端应用才会尝试连接
SpringCloudConfig的服务端来获取远程配置信息并初始化Spring环境配置
spring.cloud.config.uri=http://localhost:7001/

server.port=7002
management.security.enabled=false
```

- 上面这些属性必须配置在 bootstrap.properties 中， 这样 config-server 中的配置信息才能被正确加载。 Spring Boot 对配置文件的加载顺序， 对于本应用 jar 包之外的配置文件加载会优先于应用 jar 包内的配 置内容， 而通过 bootstrap.properties 对 config-server 的配置， 使得该应用会 从 config-server 中获取一些外部配置信息， 这些信息的优先级比本地的内容要高， 从 而实现了外部化配置

```
@RefreshScope
@RestController
public class TestController {

    @Value("${from}")
    private String from;

    @Autowired
    private Environment env;

    //访问  http://localhost:7002/from,
    @RequestMapping("/from")
    public String from() {
        return this.from + "1111111111111" + env.getProperty("from",
"undefined") +
                env.getProperty("test", "222222");
```

```
    }
}
```

- 客户端应用从配置管理中获取配置信息遵从下面的执行流程
  - 应用启动时，根据bootstrap.properties中配置的应用名{application}、环境名{profile}、 分支名{label}, 向ConfigServer请求获取配置信息
  - ConfigServer根据自己维护的Git仓库信息和客户端传递过来的配置定位信息去查找配置信息
  - 通过g江 clone命令将找到的配置信息下载到ConfigServer的文件系统中
  - ConfigServer创建Spring的ApplicationContext实例，并从Git本地仓库中加载配置文件， 最后将这些配置内容读取出来返回给客户端应用
  - 客户端应用在获得外部配置文件后加载到客户端的ApplicationContext实例，该配置内容的优先级高于客户端Jar包内部的配置内容， 所以在Jar包中重复的内容将不再被加载

# 服务化配置中心

- 在SpringCloud中， 也可以把ConfigServer视为微服务 架构中与其他业务服务一样的一个基本单元
- 服务端
  - `config-server`，增加`eureka`
  - application.properties中添加指定服务注册中心的位置

    `eureka.client.serviceOrl.defaultZone=http://localhost:1111/eureka/`

    ```
    spring.application.name=config-server
    server.port=7001

    # master分支 http://localhost:7001/woody/prod
    # config-label-test 分支 http://localhost:7001/woody/prod/config-label-test
    spring.cloud.config.server.git.uri=https://github.com/xiaojieWoody/SpringCloudConfigServer/
    spring.cloud.config.server.git.search-paths=springcloud_in_test/config-repo
    spring.cloud.config.server.git.username=xiaojieWoody
    spring.cloud.config.server.git.password=Iam0108dyj

    eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/
    ```

  - 主类新增@EnableDiscoveryClient注解，用来将config-server注册到上面配置的服务注册中心上去
- 客户端
  - `config-client`，新增`eureka`
  - bootstrap.properties中

    ```
    # 用来定位Git中的资源
    ```

```
spring.application.name=woody
server.port=7002
management.security.enabled=false

# 服务化配置中心
# 用于服务的注册与发现
eureka.client.serviceUrl.defaultZone=http://localhost:1110/eureka/
# 开启通过服务来访问 Config Server 的功能
spring.cloud.config.discovery.enabled=true
# 指定 Config Server 注册的服务名
spring.cloud.config.discovery.serviceId=config-server
# 用来定位Git中的资源
spring.cloud.config.profile=test
```

    - 主类中增加@EnableDiscoveryClient 注解，用来发现config-server服务， 利用其来加载应用配置
    - 沿用之前创建的Controller来加载Git中的配置信息
    - 访问客户端应用提供的服务`http://localhost:7002/from`

- 失败快速响应与重试
    - 希望可以快速知道当前应用是否能顺利地 从Cofin gServer获取到配置信息， 这对在初期构建调试环境时， 可以减少很多等待启动的时间。 要实现客户端优先判断ConfigServer获取是否正常，并快速响应失败内容， 只需在 bootstrap.properties中配置参数 spring.cloud.config.failFast= true即可
    - 自动重试的功能， 在开启重试功能前， 先确保已经配置了 spring.cloud.config.failFast=true, 再进行下面的操作
        - 客户端，增加 spring-retry 和 spring-boot-starter-aop依赖

```
# 初始重试间隔时间(毫秒)，默认为1000毫秒
spring.cloud.config.retry.multiplier=1000
# 下一间隔的乘数，默认为1.1，所以当最初间隔是1000毫秒时，下一次失败后的间隔为1100毫秒
spring.cloud.config.retry.initial-interval=1.1
#最大间隔时间， 默认为 2000 毫秒
spring.cloud.config.retry.max-interval=2000
# 最大重试次数，默认为 6 次
spring.cloud.config.retry.max-attempts=6
```

- 获取远程配置
    - 通过向 Config Server 发送 GET 请求以直接的方式获取
        - 不带{label}分支信息，默认访问 master 分支
            - /{application}-{profile}.yml
            - /{application}-{profile}.properties
        - 带{label}分支信息
            - /{label}/{application}-{profile}.yml

        - /{label}/{application}-{profile}.properties
        - /{application}/{profile} [/{label}]
    - 通过客户端 配置方式加载的内容
        - spring.application.name: 对应配置文件中的{application}内容
        - spring.cloud.config.profile: 对应配置文件中{profile} 内容
        - spring.cloud.config.label: 对应分支内容，如不配置，默认为master
- 动态刷新配置
    - 在config-client端做一些改造以实现配置信息的动态刷新
        - 添加`actuator`模块，其中包含了/refresh端点的实现，该端点将用于实现客户端应用配置信 息的重新获取与刷新
        - 修改远程git仓库配置文件后，通过POST请求发送到`http://localhost:7002/refresh`