# 微服务专题-一站式微服务架构SpringCloud

主讲：Cat 老师

**北京动力节点教育科技有限公司**  
**2019 • 北京**

**动力节点版权所有，禁止私自传播**

# 快速回顾

1. 分布式与微服务架构的理论梳理；
2. 什么是Spring Cloud？
3. Spring Cloud的整体架构 （与Dubbo比较）
4. 服务消费者Controller直连调用服务提供者Controller（http协议的restful）
5. Spring Cloud的注册中心Eureka
6. Spring Cloud Eureka与Zookeeper比较
7. Spring Cloud Eureka高可用集群
8. Spring Cloud Eureka自我保护机制
9. Spring Cloud Ribbon负载均衡
10. Spring Cloud Feign声明式服务调用（与Dubbo接口层比较）
11. Spring Cloud Hystrix 服务熔断降级、服务限流
12. Spring Cloud Hystrix DashBoard仪表盘监控
13. Spring Cloud Hystrix Turbine聚合监控
14. Spring Cloud Zuul网关（路由、过滤、异常、降级）

# Spring Cloud Config分布式配置

## 1、什么是配置中心

1. 传统配置方式：

配置信息分散到系统各个角落方式，配置文件或者在代码中；

1. 集中式配置中心：

将应用系统中对配置信息的管理作为一个新的应用功能模块，进行集中统一管理，并且提供额外功能；  
3、分布式配置中心：

## 在分布式、微服务架构中，独立的配置中心服务； 3. 为什么需要分布式配置中心在分布式微服务体系中，服务的数量以及配置信息日益增多，比如各种服务器参数配置、各种数据库访问参数配置、各种环境下配置信息的不同、配置信息修改之后实时生效等等，传统的配置文件方式或者将配置信息存放于数据库中的方式已无法满足开发人员对配置管理的要求，如：

安全性：配置跟随源代码保存在代码库中，容易造成配置泄漏

时效性：修改配置，需要重启服务才能生效；

局限性：无法支持动态调整：例如日志开关、功能开关；

## 2、常用分布式配置中心框架

Apollo（阿波罗）：携程框架部门研发的分布式配置中心，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性，适用于微服务配置管理场景；

diamond : 淘宝开源的持久配置中心，支持各种持久信息（比如各种规则，数据库配置等）的发布和订阅；（更新稍微落后一点）

XDiamond：全局配置中心，存储应用的配置项，解决配置混乱分散的问题，名字来源于淘宝的开源项目diamond，前面加上一个字母X以示区别。

Qconf： 奇虎360内部分布式配置管理工具，用来替代传统的配置文件，使得配置信息和程序代码分离，同时配置变化能够实时同步到客户端，而且保证用户高效读取配置，这使的工程师从琐碎的配置修改、代码提交、配置上线流程中解放出来，极大地简化了配置管理工作；

Disconf：百度的分布式配置管理平台，专注于各种分布式系统配置管理的通用组件和通用平台, 提供统一的配置管理服务；

Spring Cloud Config：Spring Cloud微服务开发的配置中心，提供服务端和客户端支持；

国外也有很多开源的配置中心 Apache 的 Apache Commons Configuration、owner、cfg4j 等等；

大型互联网公司自己内部都有自己独立分布式配置中心，用于解决服务配置管理问题；

## 什么是Spring Cloud Config？

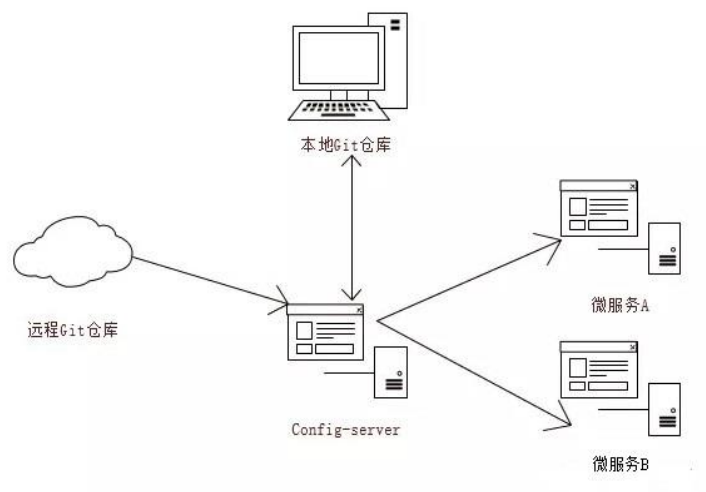
Spring Cloud Config 是一个解决分布式系统的配置管理方案。它包含 Client

和 Server 两个部分，Server 提供配置文件的存储、以接口的形式将配置文件的内容提供出去，Client 通过接口获取数据、并依据此数据初始化自己的应用。

Spring cloud config使用 git 或 svn 、也可以是本地存放配置文件，默认情况下使用 git。

## Springcloud config 的工作原理

Spring cloud Config Server 的工作过程如下图所示：



1、首先需要一个远程 Git 仓库，平时测试可以使用 GitHub，在实际生产环境

中，需要自己搭建一个 Git 服务器，远程 Git 仓库的主要作用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，都会克隆一份到本地，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、微服务 A、微服务 B 则是我们的具体应用，这些应用在启动的时会从 Config

Server 中获取相应的配置信息；

4.当微服务 A、微服务 B 尝试从 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，所以配置文件天然具有版本管理功能；

## 构建 Spring cloud config 配置中心仓库

前面我们介绍了spring cloud config的配置中心一般是采用git存储配置，（我们在这里采用国内的码云 gitee， 也可以使用国外的GitHub，由于使用Github网络比较慢，可能连接失败） ，

接下来我们需要在 gitee 上设置好配置中心，我们通过idea 把gitee上的springcloud项目clone到本地，然后再项目下创建一个文件夹config-center，然后在 config-center中创建四个配置文件，如下：

application.properties

application-dev.properties

application-test.properties

application-online.properties

在四个文件里面分别写上要测试的内容：

比如：

url=http://www.bjpowernode.com

url=http://dev.bjpowernode.com

url=http://test.bjpowernode.com

url=http://online.bjpowernode.com

然后通过idea将本地文件同步到 Gitee 仓库中：

## 构建 Spring cloud config 配置中心服务端

构建一个 spring cloud config 配置中心按照如下方式进行：

1、创建一个普通的 Spring Boot 项目

2、在 pom.xml 文件中添加如下依赖：

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-config-server</**artifactId**>  
</**dependency**>

3、在入口类，也就是 main 方法的类上添加注解 @EnableConfigServer

@EnableConfigServer *//开启spring cloud config配置中心支持*@SpringBootApplication  
**public class** Application {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(Application.**class**, args);  
 }  
}

4、在application.properties中配置一下git仓库信息，此处我们使用gitee （也可以使用码云GitHub，由于网络比较慢，可能连接失败） ，

首先我的gitee上有一个名为 springcloud的项目，我们的配置文件如下：

**server.port**=**8888  
  
spring.application.name**=**34-springcloud-service-config  
  
spring.cloud.config.server.git.uri**=**https://gitee.com/verycat/springcloud.git  
spring.cloud.config.server.git.search-paths**=**config-center  
spring.cloud.config.server.git.username**=**verycat  
spring.cloud.config.server.git.password**=**123456.**

其中：

1．uri 表示配置中心所在仓库的位置

2．search-paths 表示仓库下的子目录

3．username 表示你的 GitHub 用户名

4．password 表示你的 GitHub 密码

至此我们的配置中心服务端就创建好了。

此时启动我们的配置中心，通过/{application}/{profile}/{label}就能访问到我们

的配置文件了；

配置文件的映射规则：

/{application}/{profile}[/{label}]

<http://localhost:8888/application/dev/master>

/{application}-{profile}.properties

<http://localhost:8888/application-dev.properties>

/{label}/{application}-{profile}.properties

<http://localhost:8888/master/application-dev.properties>

/{application}-{profile}.yml

[http://localhost:8888/application-dev.yml](http://localhost:8888/master/application-dev.yml)

/{label}/{application}-{profile}.yml

<http://localhost:8888/master/application-dev.yml>

其中：

{application} 表示配置文件的名字，对应的配置文件即 application，

{profile} 表示环境，有 dev、test、online 及默认，

{label} 表示分支，默认我们放在 master 分支上，

通过浏览器上访问 http://localhost:8888/application/dev/master

返回的 JSON 格式的数据：

name 表示配置文件名 application 部分，

profiles 表示环境部分，

label 表示分支，

version 表示 GitHub 上提交时产生的版本号，

同时当我们访问成功后，在控制台会打印了相关的日志信息；

当访问成功后配置中心会通过 git clone 命令将远程配置文件在本地也保存一

份，以确保在 git 仓库故障时我们的应用还可以继续正常使用。

## 构建 Springcloud config 配置中心客户端

前面已经搭建好了配置中心的服务端，并且通过访问接口从config服务端读取配置信息， 不过实际开发中，更多的不是我们人为去获取配置信息，而是由微服务自己从config服务端上加载配置信息， 那么怎么来加载呢?

1、我们的各个微服务都是客户端，比如服务提供者、服务消费者等各个微服务都可以从配置中心获取配置信息，但是要添加如下依赖：

*<!-- spring-cloud-starter-config -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**artifactId**>  
</**dependency**>

2、创建 bootstrap.properties 文件，用于获取配置信息，文件内容如下：

（注意这些信息一定要放在 bootstrap.properties 文件中才有效） **spring.application.name=application****spring.cloud.config.profile**=**dev  
spring.cloud.config.label**=**master  
spring.cloud.config.uri**=**http://localhost:8888/**

其中：

name 对应配置文件中的 application 部分，

profile 对应了 profile 部分，

label 对应了 label 部分，

uri 表示配置中心的地址。

我们解释一下bootstrap.properties 文件；

Spring Cloud有一个“引导上下文"的概念，这是主应用程序的父上下文。引导上下文负责从配置服务器加载配置属性，以及解密外部配置文件中的属性。和主应用程序加载application.*(yml或 properties)中的属性不同，引导上下文加载(bootstrap.*)中的属性。配置在 bootstrap.\*中的属性有更高的优先级，因此默认情况下它们不能被本地配置覆盖；

3、创建一个 Controller 进行测试：

@RestController

@RefreshScope

**public class** ConfigController {

@Value(**"${url}"**)//读取远程配置中心配置的信息

**private** String **url**;

@Autowired

**private** Environment **env**;

@RequestMapping(**"/cloud/url"**)

**public** String url () {

**return this**.**url**;

}

@RequestMapping(**"/cloud/url2"**)

**public** String url2 () {

**return env**.getProperty(**"url"**);

}

}

我们可以直接使用@Value 注解注入配置的属性值，也可以通过 Environment

对象来获取配置的属性值。

## 配置信息的加解密安全处理

前面是在Git仓库中明文存储配置信息值，很多场景下，对于某些敏感的配置内容（例如数据库账号、密码等），应该加密存储，config server为我们考虑到了这一点，对配置内容提供了加密与解密支持；

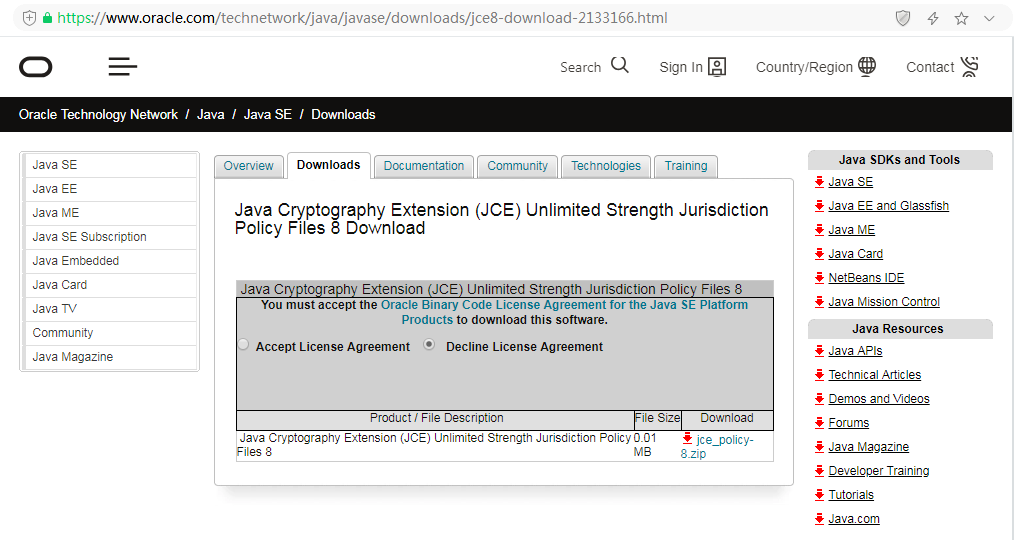
**安装JCE**

config server的加解密功能依赖Java Cryptography Extension（JCE）

Java 8 JCE下载地址：

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

下载得到JCE的zip压缩包并解压，将其中的jar包覆盖到JDK/jre/lib/security目录下；



**对称加密**

config server提供了加密与解密的接口，分别是

加密接口：<http://localhost:8888/encrypt>

解密接口：<http://localhost:8888/decrypt> （这个接口我们不需要用）

解密是config-server自动完成的；

在原来的34-springcloud-service-config项目中添加bootstrap.properties配置文件，里面配置密钥：

*#设置对称密钥，用这个密钥进行加密和解密***encrypt.key**=**cat**

运行项目，验证加解密：

加密：curl http://localhost:8888/encrypt -d root

得到：9455e7d7333e1cc9d1a8680244f6db3aae6de93825d4bc6f9f4258993f7a8cad  
解密：curl http://localhost:8888/decrypt -d 9455e7d7333e1cc9d1a8680244f6db3aae6de93825d4bc6f9f4258993f7a8cad

在git仓库增加修改配置文件application-dev.properties，增加如下内容：

**spring.datasource.username**=**{cipher}9455e7d7333e1cc9d1a8680244f6db3aae6de93825d4bc6f9f4258993f7a8cad  
spring.datasource.password**=**{cipher}5a6e3e417d416736d638cd4db7a6380a71a8709c3ec335a28e01cff05705eeaa**

访问地址：<http://localhost:8888/application-dev.properties> 得到密钥原文；

说明 config server能自动解密配置内容；