**【Spring Cloud Bus】**

**【主要内容】**

1. 消息总线简介
2. Config集成bus
3. 消息总线局部刷新配置
4. Webhook全自动刷新配置
5. 配置文件中心集群

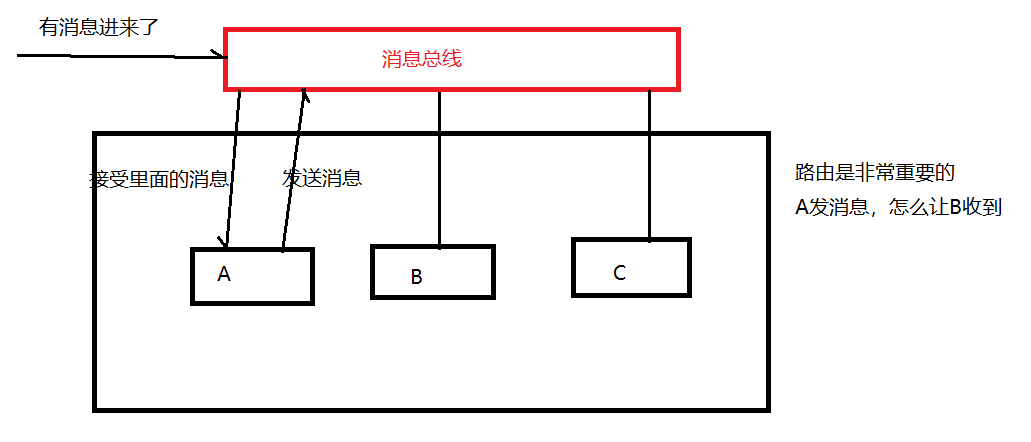
**【学习目标】**

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 消息总线简介 | 了解 |
| Config集成bus | 掌握 |
| 消息总线局部刷新配置 | 掌握 |
| Webhook全自动刷新配置 | 掌握 |
| 配置文件中心集群 | 掌握 |

# 什么是Spring Cloud Bus

官网：<https://spring.io/projects/spring-cloud-bus>

Spring Cloud Bus**将轻量级消息代理程序链接到分布式系统的节点**。然后可以将其用于**广播状态更改（例如，配置更改）或其他管理指令**



# Spring Cloud Bus能用来做什么？

**Spring Cloud Bus配合Spring Cloud Config使用可以实现配置的动态刷新**

Spring Cloud Bus是用来把分布式系统的节点与轻量级消息系统连接起来的框架

**它整合了java的事件处理机制和消息中间件的功能**

它就像一个分布式执行器，可用于广播状态更改，事件推送等，也可以当作微服务间的通信通道。目前支持RabbitMQ和Kafka

微服务之间的通讯属于进程间通讯 通讯方式 跨进程间通讯 线程间怎么通讯 生产者消费者模型

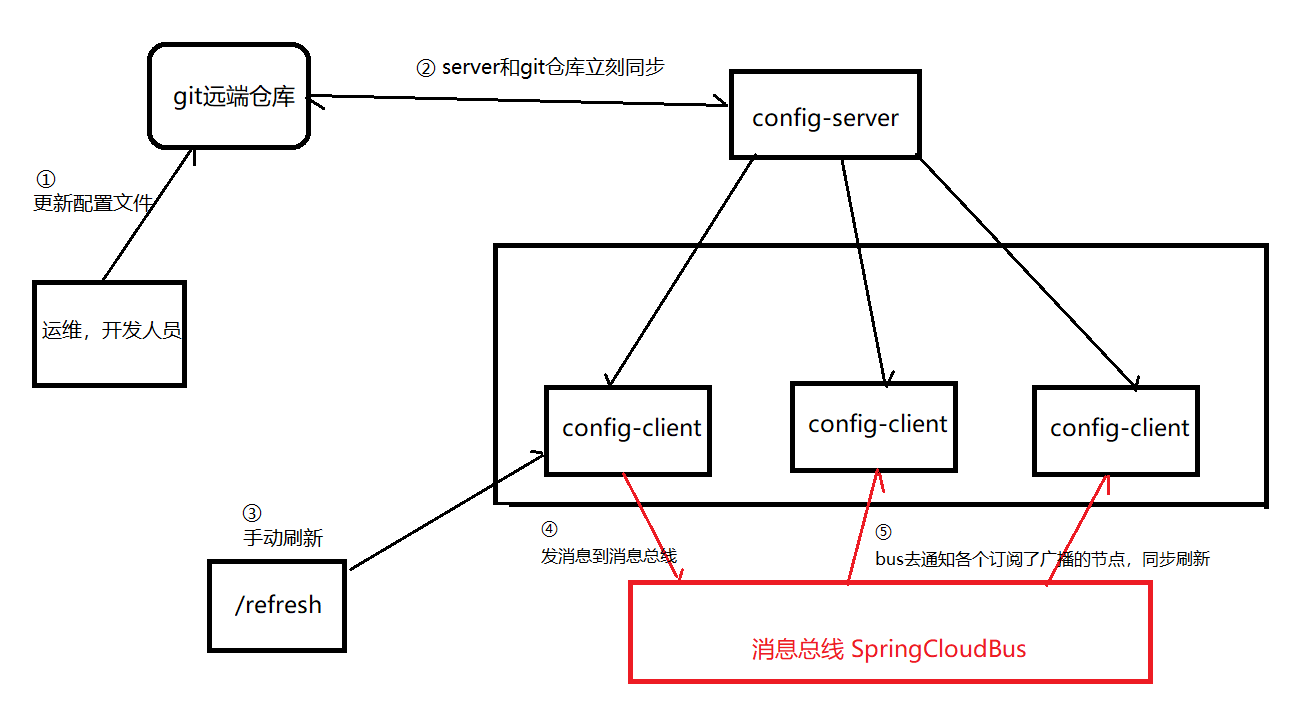
1. RPC:(耦合度较高，面向连接的比较稳定 适合大数据量的传输)  
   2. MQ(性能没有RPC强，耦合度比较低，适合数据量小)

# 使用RabbitMq实现消息总线（全刷新）

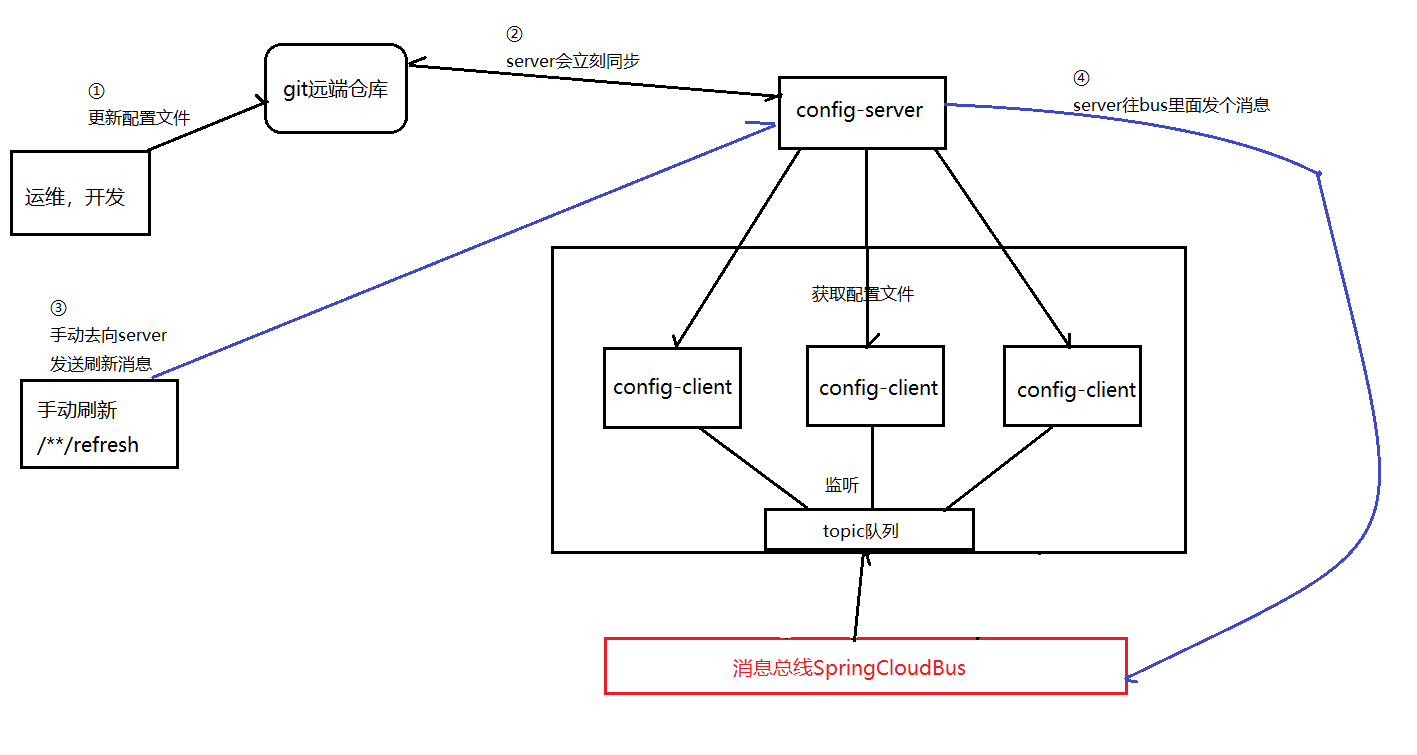
### 架构图

**第一种，手动刷新一个客户端，然后通过bus传播到其他客户端**

**但是这样做流违背了设计原则， 客户端只做客户端的事**



**第二种，我们去刷新config-server，然后config-server去往bus里面发一个消息，让订阅了这个topic的客户端全部刷新。**



### 目的

就是为了刷新配置文件

## 具体实现（第二种方案，使用config-server去做）

### 修改config-server

#### 添加依赖

|  |
| --- |
| <!-- 添加bus和mq的集成依赖注意创建项目选的bus依赖不对--> <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId> </dependency> |

#### 修改配置文件

|  |
| --- |
| server:  port: 8888 spring:  cloud:  config:  server:  git:  uri: https://gitee.com/smiledouble/springcloud-config  username: smiledouble  password: \*\*\*\*\*\*  search-paths: /config  **rabbitmq:  host: 192.168.226.128  username: admin  password: admin  port: 5672** **management:  endpoints:  web:  exposure:  include: '\*'** |

### 再创建一个config-client-b，除了端口不一样，其他都一致

下面的配置两个都要改

#### 添加依赖

|  |
| --- |
| <!-- 添加bus和mq的集成依赖注意创建项目选的bus依赖不对--> <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId> </dependency> |

#### 修改配置文件

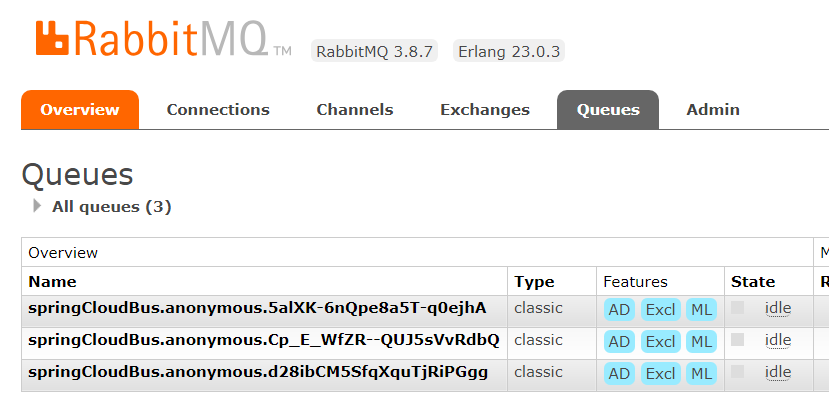
|  |
| --- |
| **server:  port: 8181 #注意client-b的端口改成8182** spring:  application:  name: config-client  cloud:  config:  uri: http://localhost:8888/ #找到配置文件中心的地址，  profile: dev #选择哪一个配置文  name: config #对应application  label: master #对应分支  **rabbitmq:  host: 192.168.226.128  username: admin  password: admin  port: 5672 management:  endpoints:  web:  exposure:  include: '\*' #暴露监听的端口 也可以写成 'bus-refresh'** |

### 测试

**启动config-server**

**启动client**

**启动client-b**



**修改git上的配置内容**

**执行POST请求 <http://localhost:8888/actuator/bus-refresh> 没有返回值，即成功**

**查看两个客户端内容是否刷新**

# 消息总线局部刷新

**在微服务架构中，有些配置文件修改 只想通知给部分客户端，那么怎么办呢**

**只想定点通知只通知8181不通知8182**

## 公式说明

公式：**http://ip:配置中心的端口号/actuator/bus-refresh/{destination}**

/bus/refresh请求不再发送到具体的服务实例上

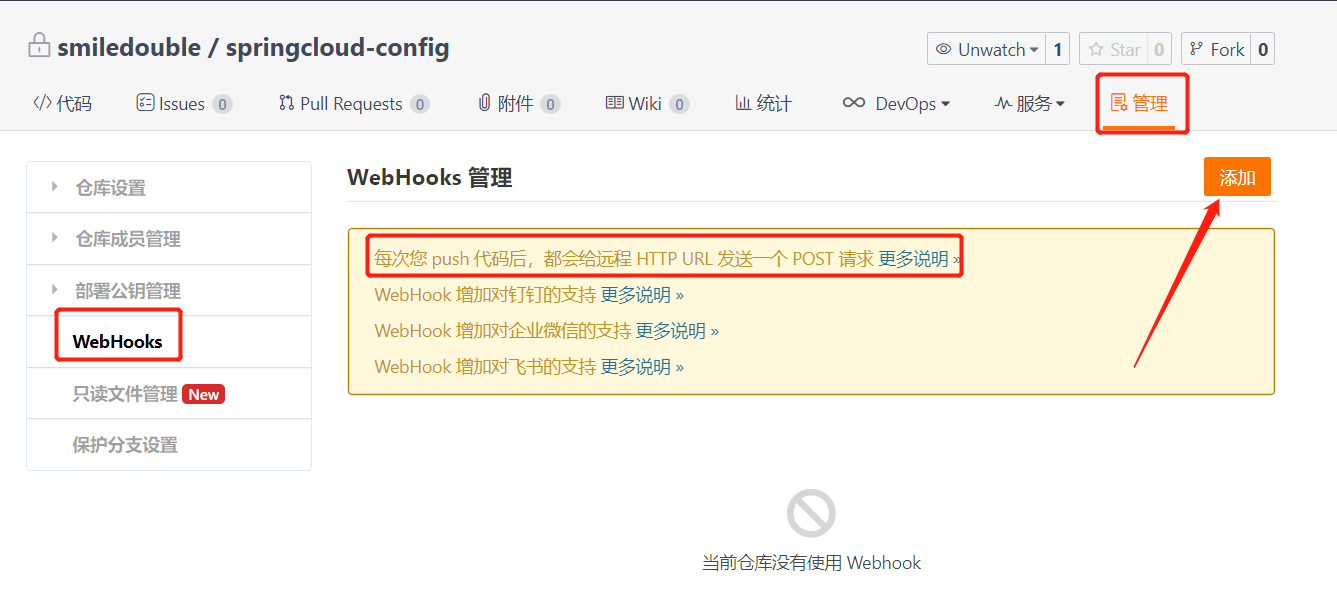
而是发给config-server并通过destination参数类指定需要更新配置的服务或实例

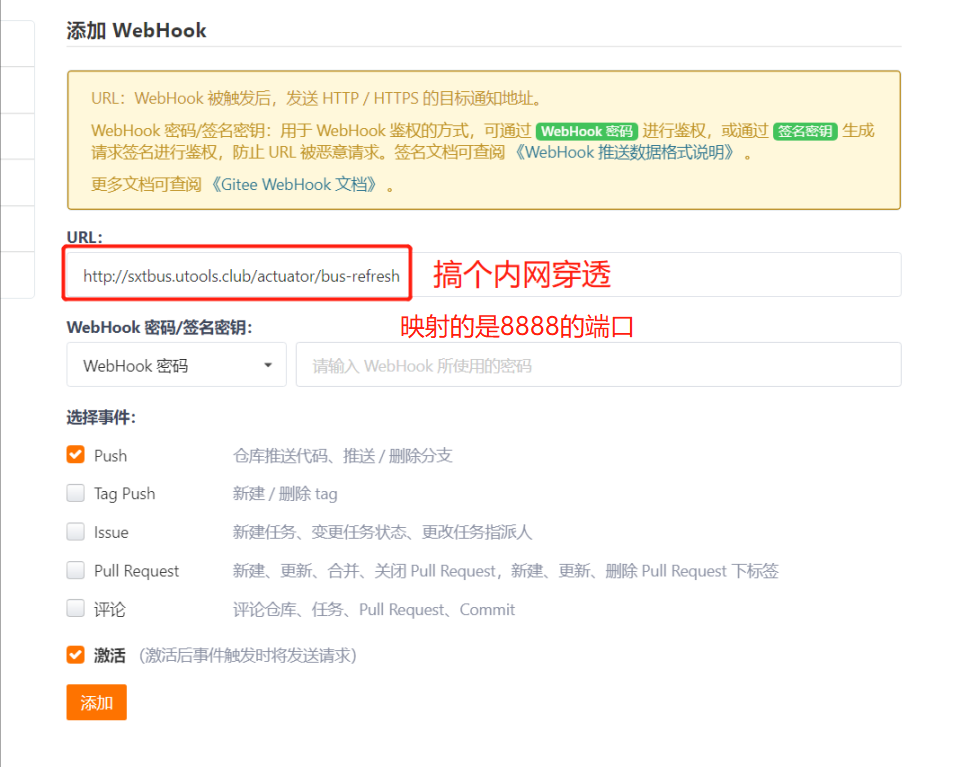
## 修改git上的配置后使用公式通知8081客户端

[http://localhost:8888/actuator/bus-refresh/](http://localhost:8888/actuator/bus-refresh/config-client:8181)**[config-client:8181](http://localhost:8888/actuator/bus-refresh/config-client:8181)** 执行完毕即可

# 使用钩子实现真正的全自动刷新

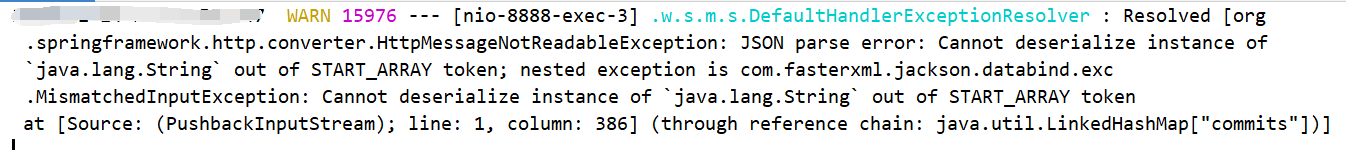
## 配置gitee上的WebHooks





## 修改配置内容后

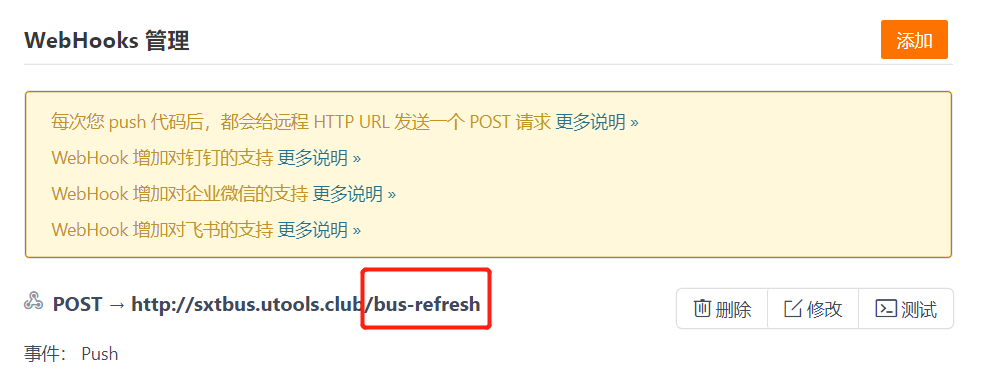
在config-server中报错了，说明我们的钩子确实成功了，但是gitee发送的请求有很多自带的东西，我们不需要



## 我们给config-server写一个接口

|  |
| --- |
| @RestController public class RefreshController {    @Autowired  private RestTemplate restTemplate;   @Value("${server.port}")  private Integer port;   /\*\*  \* gitee -> myRefresh -> http://localhost:8888/actuator/bus-refresh/  \* 自定义的刷新接口  \* //自己发请求 http://localhost:8888/actuator/bus-refresh/  \* //restTempalte POST http://localhost:8888/actuator/bus-refresh/  \*  \* @return  \*/  @PostMapping("myRefresh")  public ResponseEntity<String> refresh() {  //url地址  String url = "http://localhost:" + port + "/actuator/bus-refresh/";  //设置请求头参数  HttpHeaders httpHeaders = new HttpHeaders();  httpHeaders.add("content-type", "application/json");  HttpEntity<String> httpEntity = new HttpEntity<>(httpHeaders);  //这里我们真正的发刷新配置的请求  ResponseEntity<String> responseEntity = restTemplate.postForEntity(url, httpEntity, String.class);  //拿到响应体  String body = responseEntity.getBody();  System.*out*.println(body);  //拿到响应码  int value = responseEntity.getStatusCode().value();  System.*out*.println(value);  if (value == HttpStatus.*NO\_CONTENT*.value()) {  //如果是204的话就成功了  return ResponseEntity.*ok*("刷新成功");  }  //返回刷新失败  return ResponseEntity.*badRequest*().body("刷新失败");  }  } |

## 修改WebHooks访问地址



## 再次修改git内容提交后即可全部刷新了

# 配置文件中心集群，高可用

**这里我们需要用到eureka了，将config配置文件中心注册到eureka里面**

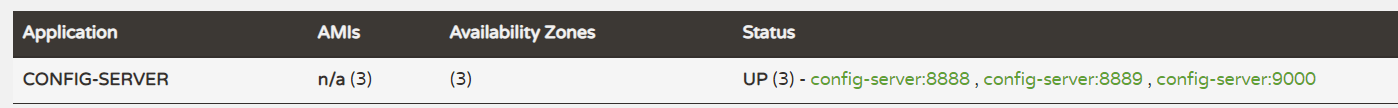
## 添加eureka-client的依赖

|  |
| --- |
| <!-- 集群搭建需要eureka--> <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId> </dependency> |

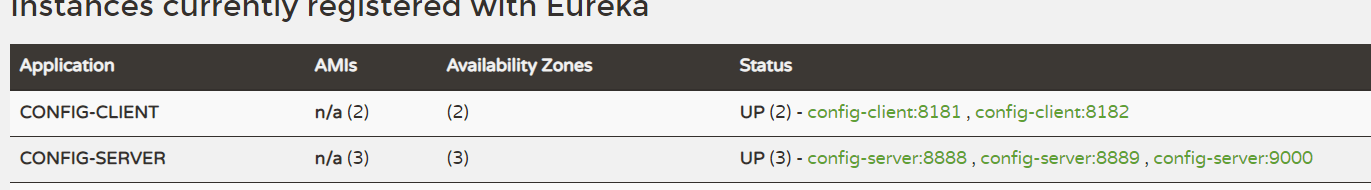
## 修改配置文件

|  |
| --- |
| server:  port: 8888 spring:  application:  name: config-server  cloud:  config:  server:  git:  uri: https://gitee.com/smiledouble/springcloud-config  username: smiledouble  password: \*\*\*\*\*\*  search-paths: /config  rabbitmq:  host: 192.168.226.128  username: admin  password: admin  port: 5672 management:  endpoints:  web:  exposure:  include: '\*' eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka  instance:  instance-id: ${spring.application.name}:${server.port}  prefer-ip-address: true |

## 修改端口启动三台config-server



## 启动两台config-client



**然后gitee修改内容提交后，都刷新了**

**但是还是往8888端口发送的请求，如果8888挂了呢？**

## 我们可以使用网关，让网关来发请求选择一个server执行刷新请求

**启动网关（之前案例）**

**修改gitee上的WebHooks为网关的地址，这里内网穿透到网关（80的端口）动态路由**

**<http://sxtbus.utools.club/config-server/bus-refresh>**

**测试提交推送后成功**

# 总结

## 客户端的bootstrap.yml放什么？

配置文件中心的配置信息必须放

Eureka 也要放

应用的名称

要让bootstrap 成为一个最小化的配置文件信息，能不放在bootstrap 里面就不放，我们尽量放在配置文件中心服务器里面！

## Config-server的application.yml放什么？

理论上说，congfig-server、 可以放一个应用的所有配置信息

但是config-server 里面可以实现配置文件的动态刷新

可以抄袭复制别的服务的配置文件