

ch03-服务配置

公众号：锋哥聊编程

主讲：小锋



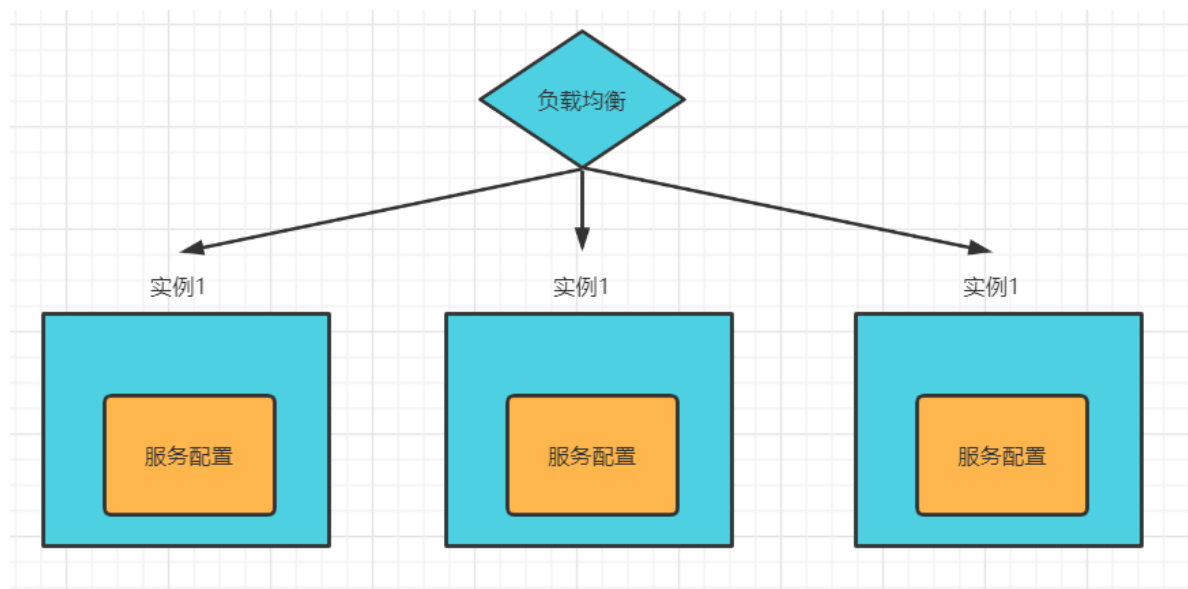
1、为什么要统一服务配置管理？

为了避免参数变化引起的频繁的程序改动，通常我们在应用程序中将常用的一些系统参数、启动参数、数据库参数等等写到配置文件或其他的存储介质里面。

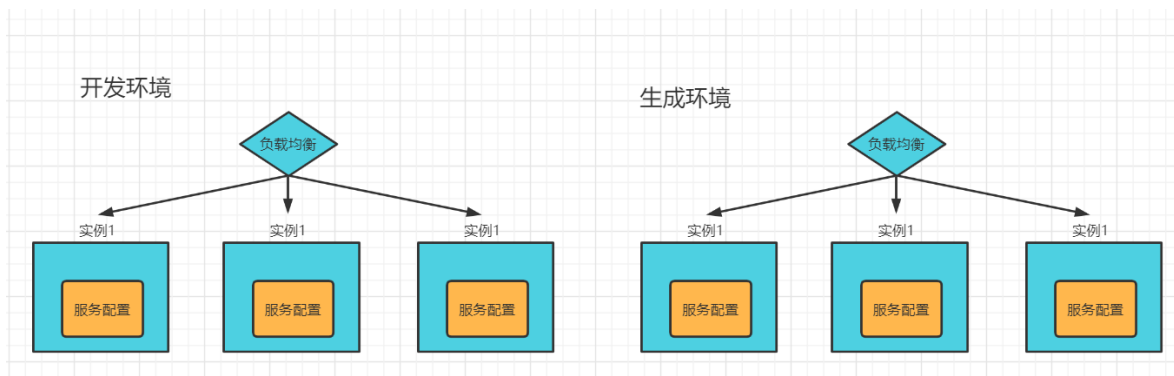
- 配置常见的存储方式：配置文件、数据库等
- 配置对于应用程序是只读的，程序通过读取配置来影响程序的运行行为
- 配置是区分环境的同一份程序部署到生产、测试、开发、演示环境下，需要做不同的配置

传统应用程序的配置分散，导致了在进行部署、运维方面，需要极大的成本。那么传统的应用程序配置面临哪些问题？

问题一：应用程序多实例集群部署，每个微小的配置的修改将导致每个实例都需要重新打包部署



问题二：每一套环境的配置不同，难于维护，增加了人工犯错的几率



问题三：没有严格的配置管理权限控制，导致公司的核心数据泄露

不知道大家有没有看过一条报道，国外某著名的公司，在开源代码的数据库连接配置中，携带了其"生产环境"的数据库配置信息，导致其核心的用户数据泄露。

除了上面的三点，还有很多传统的配置管理方式面临的问题，所以我们要进行集中的统一的配置管理。这点在微服务应用中体现的更为明显。

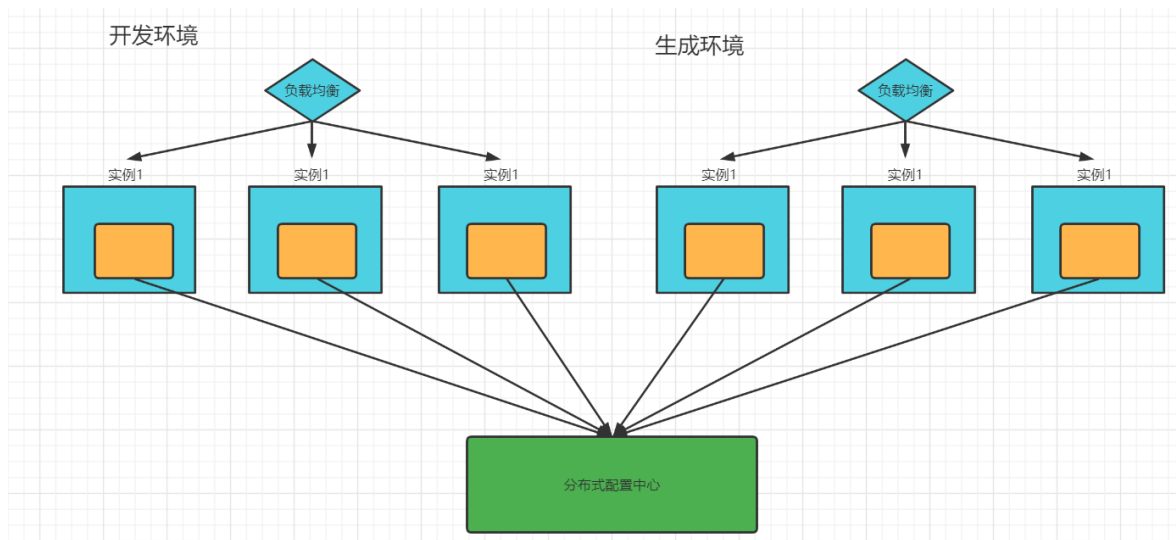
2、分布式配置管理中心

理想的配置管理中心应该是：

- 支持多应用配置管理
- 支持多环境(生产、测试等)配置管理
- 支持配置权限管理
- 支持配置版本化管理、配置回滚
- 支持配置的动态发布、灰度发布

配置中心将配置从应用中剥离出来，统一管理，优雅的解决了配置的动态变更、持久化、运维成本等问题。应用自身既不需要去添加管理配置接口，也不需要自己去实现配置的持久化，更不需要引入“定时任务”以便降低运维成本。**总的来说，配置中心就是一种统一管理各种应用配置的基础服务组件。**

在系统架构中，配置中心是整个微服务基础架构体系中的一个组件，如下图，它的功能看上去并不起眼，无非就是配置的管理和存取，但它是整个微服务架构中不可或缺的一环。



3、主流的分布式配置管理

目前市面上用的比较多的分布式配置中心有：

Spring Cloud Config

2014年9月开源，Spring Cloud 生态组件，可以和Spring Cloud体系无缝整合。

<https://github.com/spring-cloud/spring-cloud-config>

Apollo

2016年5月，携程开源的配置管理中心，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。

<https://github.com/ctripcorp/apollo>

Nacos

2018年6月，阿里开源的配置中心，也可以做DNS和RPC的服务发现。（之前已经学过它的服务注册与发现功能）

<https://github.com/alibaba/nacos>

三个产品对比

对比	Spring CloudConfig	Apollo	Nacos
开源时间	2014.9	2016.5	2018.6
配置实时推送	弱支持 (Spring Cloud Bus)	支持 (HTTP长轮询1s内)	支持 (HTTP长轮询1s内)
版本管理	支持 (Git)	自动管理	自动管理
配置回滚	弱支持 (Git+Bus)	支持	支持
配置的灰度发布	理念上支持, 可操作性不强	支持	1.1.0开始支持
权限管理	不支持 (没有区分用户、角色、权限的概念)	支持	1.2.0开始支持
多集群多环境	对集群概念支持较弱	支持	支持
多语言	只支持Java	Go,C++,Python,Java,.net,OpenAPI	Python,Java,Nodejs,OpenAPI
分布式高可用最小集群数量	Config-Server 2+Git+MQ	Config2 +Admin3+Portal*2+Mysql=8	Nacos *3+MySql=4
配置格式校验	不支持	支持	支持
通信协议	HTTP和AMQP	HTTP	HTTP
数据一致性	Git保证数据一致性, Config-Server 从Git读取数据	数据库模拟消息队列, Apollo 定时读消息	HTTP异步通知

项目中用哪个配置中心好?

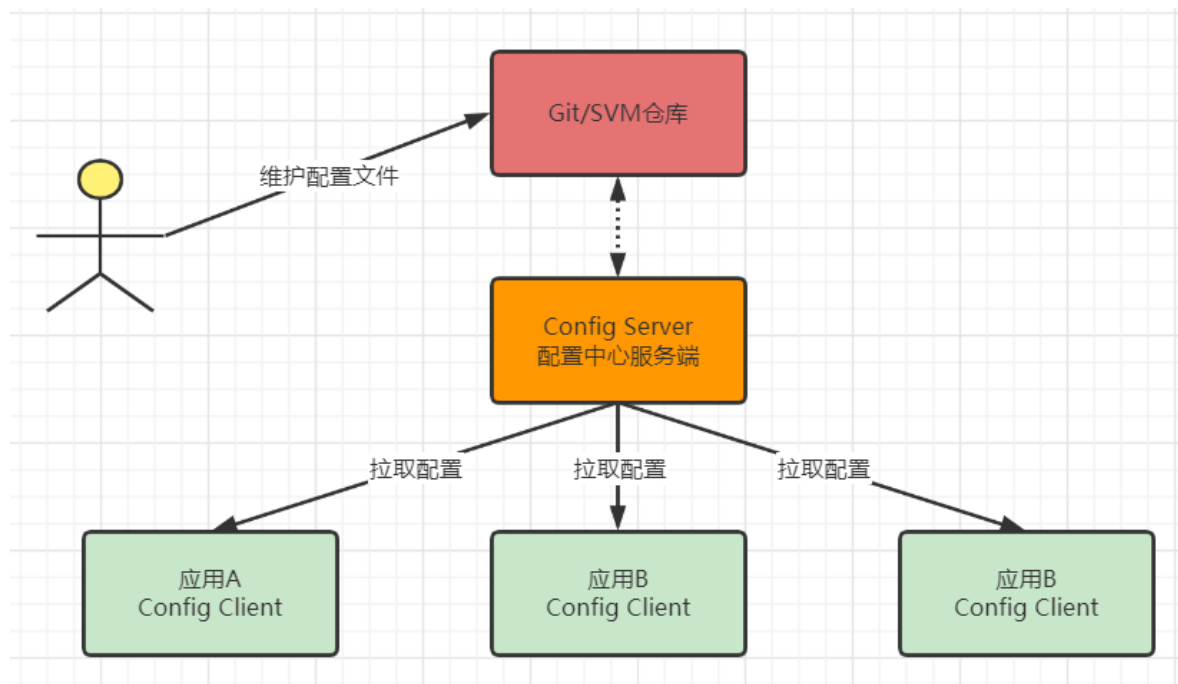
- 如果你希望完成单纯的分布式配置集中管理, 其实三者都能满足你的需求。
- 如果你的项目到已经用Nacos实现了服务注册中心, 不想单独搞出来一个配置管理中心, 合二为一的话, Nacoos可能是你的最佳选择
- 携程的Apollo与Nacos很多相似之处, 有颇多的亮点。目前Apollo从文档细节到方便度要好于Nacos。但是Nacos毕竟开源时间较短, 依托强大的Alibaba的支持, 有很大的潜力和发展空间。
- Spring Cloud Config对比其他两者, 在功能以及友好度方面都逊色。唯一的优点可能是它比较轻量级。

4、分布式配置管理（一）：Spring Cloud Config

Spring Cloud Config介绍

Spring Cloud Config提供了可水平扩展的集中式配置服务。它使用可插拔的存储库层作为数据存储，该存储层目前支持本地存储，Git和Subversion。其核心功能：

- 通过将版本控制系统用作配置存储，开发人员可以轻松地对配置更改进行版本控制和审核。
- 实现集中的配置管理，不同的环境、不同应用的配置通过文件名称进行区分。
- 支持运行时动态配置更新，即：配置的热更新
- 提供配置访问的REST接口



- 首先，我们需要一个远程的Git仓库（在实际生产环境中，一般需要自己搭建一个Git服务器。方便起见，建议使用了github仓库）
- 其次，我们需要搭建Config Sever配置中心服务端，微服务应用（如上图应用A，应用B，应用C）在启动的时候会从Config Server中来拉取载相应的配置信息。前提是Spring Cloud微服务集成了Spring Cloud Config的客户端程序。
- 当微服务应用从Config Server拉取配置信息的时候，Config Server会先通过 `git clone` 命令从远程Git Repository仓库克隆一份配置文件保存到本地。这样当Git Repository远程仓库无法连接时，就直接使用Config Server本地存储的配置信息。

搭建单机Spring Cloud Config

步骤（一）：Git配置文件仓库

在gitee创建名为 `mafeng-shop-cloud` 的仓库，然后在里面创建 `shop-config` 目录

把mafeng-user和mafeng-order项目的application.yml重名为：

- `mafeng-user-dev.yml`
- `mafeng-order-dev.yml`

把文件上传到shop-config目录下

配置文件命名规则为：{application}-{profile}.yml

- application表示项目的名称，即：spring.application.name 的配置值
- profile代表基础环境，通常是指：pro(生产)、dev（开发）、test（测试）等等。

步骤（二）：搭建Config配置中心

1) 创建config-server项目

2) 导入config-server依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

3) 启动类添加注解

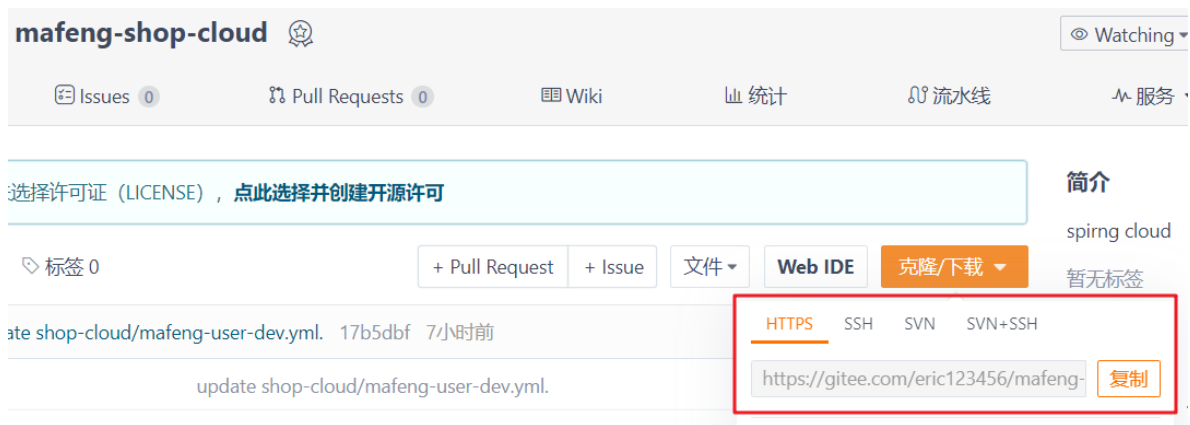
```
@EnableConfigServer    //开启Config Server
```

4) 配置application.yml

```
server:
  port: 12000    #端口自定义

spring:
  application:
    name: config-server    #config server项目名称
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/eric123456/mafeng-shop-cloud.git
          searchPaths: shop-cloud
```

- spring.cloud.config.server.git.uri: 配置git仓库位置的http访问地址
- spring.cloud.config.server.git.searchPaths: 配置仓库路径下的相对搜索位置，可以配置多个。因为我们把配置文件放在了shop-config目录下，所以配置该目录。
- spring.cloud.config.server.git.username: git仓库的用户名
- spring.cloud.config.server.git.password: git仓库的用户密码



5) Config Server访问测试

<http://localhost:12000/mafeng-user-dev.yml>

<http://localhost:12000/mafeng-order-dev.yml>

步骤（三）：Config客户端获取配置

在mafeng-order和mafeng-user服务中做以下操作：

1) 引入config client依赖

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

2) 添加bootstrap.yml文件，内容如下：

```
spring:
  application:
    name: mafeng-user
  cloud:
    config:
      uri: http://localhost:12000
      label: master
      profile: dev
```

- spring.cloud.config.profile: 对应前配置文件中的{profile}部分
- spring.cloud.config.label: 对应前配置文件的git分支
- spring.cloud.config.uri: Config Server配置中心的地址

注意：上面这些属性必须配置在 bootstrap.yml 或properties文件中，而不是 application.yml 中，config配置内容才能被正确加载。因为 bootstrap.yml 加载优先级高于 application.yml，保证在应用启动时就去加载配置。

3) 启动微服务，验证结果

Config Server认证访问

在使用Config Server的时候，我们发现可以通过URL没有任何限制就访问到项目的配置文件，这样很不安全。为了解决这个问题，我们可以使用Spring Security进行简单的Basic安全认证。

Server端引入SpringSecurity依赖

```
<dependency>
  <!-- spring security 安全认证 -->
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Server端的yml添加配置

```
spring:
  security:
    basic:
      enabled: true  #启用基本认证(默认)
    user:
      name: mafeng   #自定义登录用户名
      password: mafeng123  #自定义登录密码
```

启动测试

启动服务，测试一下。请求 <http://localhost:12000/mafeng-user-dev.yml>，会发现弹出了SpringSecurity的Basic登录验证，输入账户和密码就可以访问配置。

Client端授权访问

当Config Server增加了登录认证之后，微服务客户端想要正确的获取配置信息，在发送请求的时候也要携带用户名密码。修改配置文件bootstrap.yml，增加username和password配置：

```
spring:
  application:
    name: mafeng-user
  cloud:
    config:
      uri: http://localhost:12000
      label: master
      profile: dev
      username: mafeng
      password: mafeng123
```

Config客户端配置实时刷新

我们先在mafeng-user的Controller上面添加一个@Value参数，如下：


```
@RestController
@RequestMapping("user")
public class UserController {
    @Autowired
    private UserService userService;

    @Value("${mafeng.password}")
    private String password; // 用password这个参数演示参数刷新
```

同时，在gitee的mafeng-user-dev.yml上面添加该参数

```
mafeng:
  password: 111111
```

启动mafeng-user，访问 <http://localhost:9001/user/3>，此时password值为111111。

接着，我们直接修改gitee上面的mafeng.password变量为333333，重新访问 <http://localhost:9001/user/3>，看到password值依然为111111。

只有重启mafeng-user项目才可以看到更新后的值333333。

如何能做到gitee上的配置修改后，不用重启微服务也能获取到更新后的值呢？这时可以为微服务引入 `actuator` 机制实现配置实时刷新的能力！

导入actuator依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

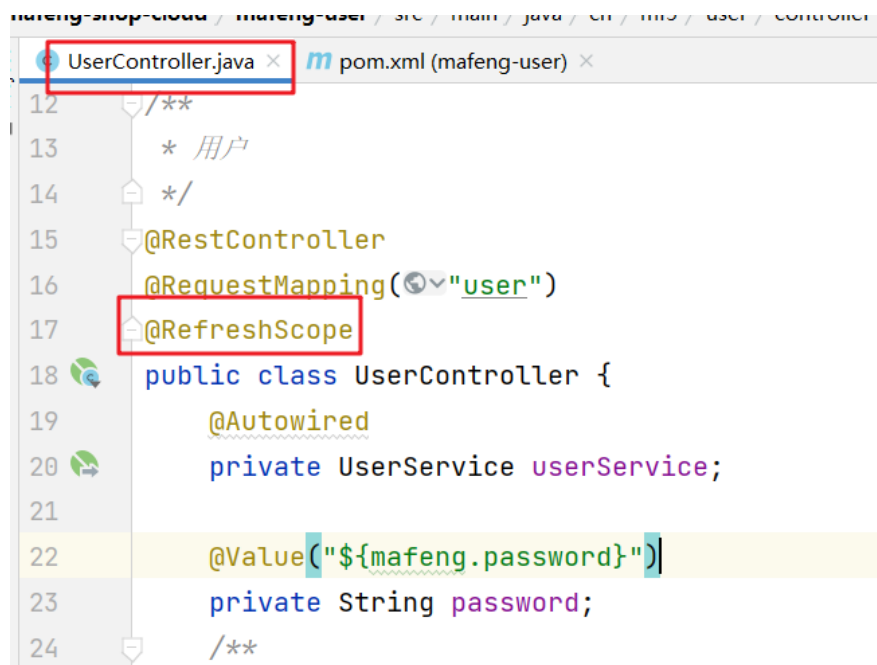
actuator可以为微服务提供配置刷新接口，刷新接口路径为：`/actuator/refresh`

配置mafeng-user-dev.yml

```
management:
  endpoints:
    web:
      exposure:
        include: refresh,health
```

- management.endpoints.web.exposure.include=refresh,health，表示我们只开放配置刷新接口和健康检查接口

添加@RefreshScope 注解



```

12  /**
13   * 用户
14   */
15  @RestController
16  @RequestMapping("user")
17  @RefreshScope
18  public class UserController {
19      @Autowired
20      private UserService userService;
21
22      @Value("${mafeng.password}")
23      private String password;
24  }

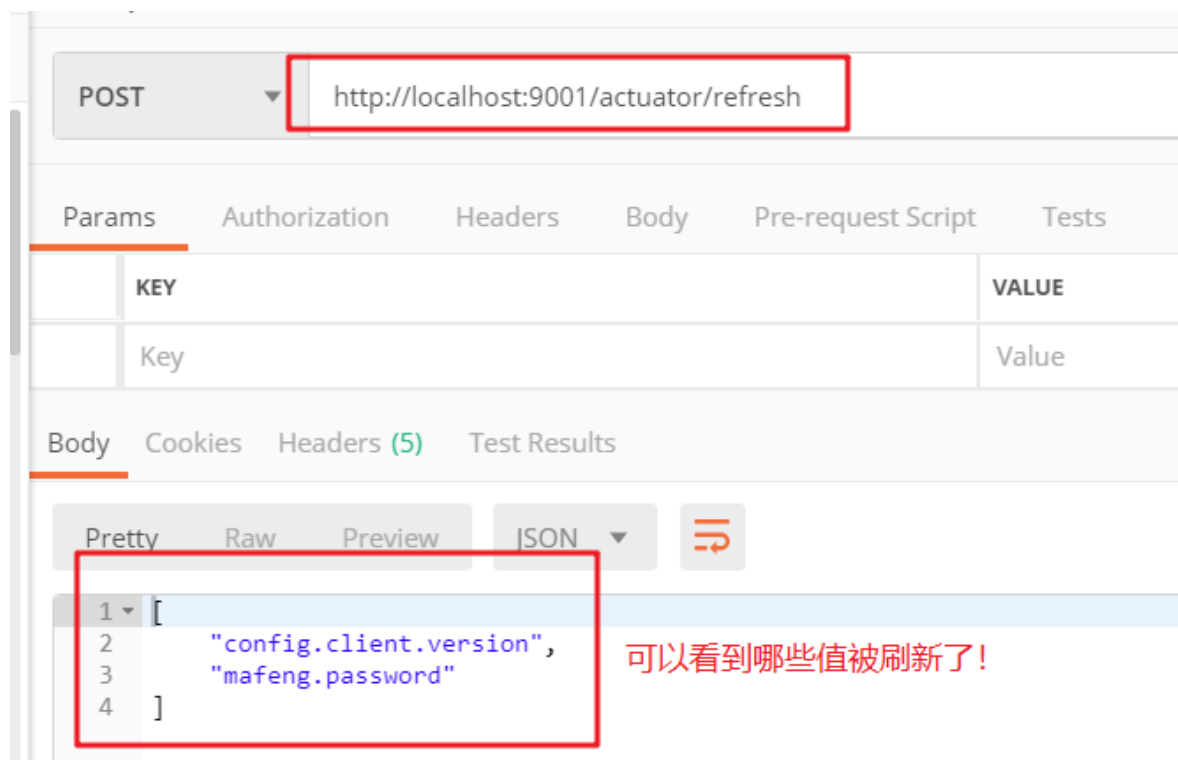
```

在需要进行配置刷新的类上使用 `@RefreshScope` 就初步具备刷新的能力。

测试触发刷新配置

修改gitee配置文件

postman请求: <http://localhost:9001/actuator/refresh> , 注意必须为post方式

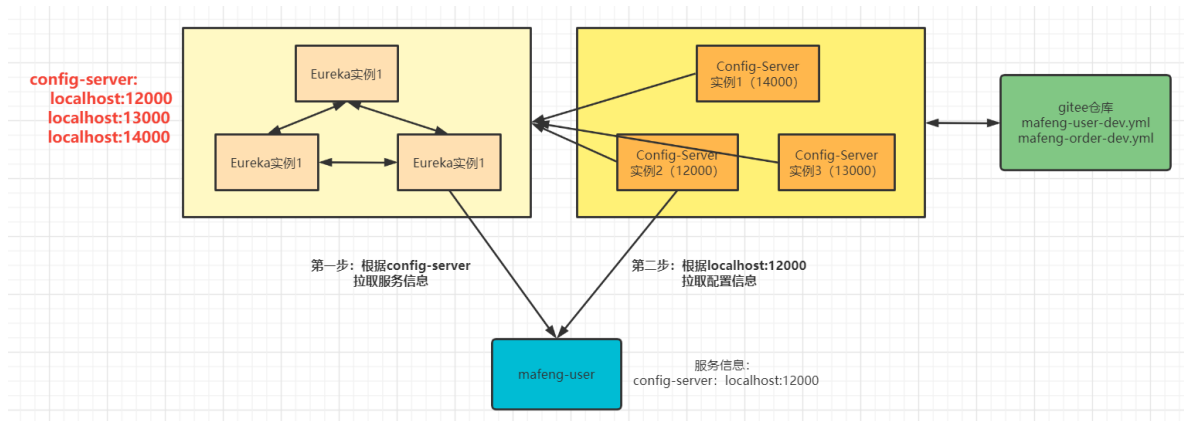


不用重启微服务, 直接查看password是否被更新? 答案是: 已经更新啦!

但以上这种方式刷新配置, 只适合手动刷新一个微服务配置, 如果比较多微服务配置都发生变更, 需要逐个手动刷新配置, 比较麻烦! 如果需要实现一键更新所有微服务配置, 这时需要用到Spring Cloud Bus (后面讲)。

Config配置中心高可用

Config Server实现高可用的原理比较简单：因为Config Server 是用Spring Boot构建的，所以我们可以把它当做微服务注册到eureka，启动多个实例向eureka注册，并对外提供服务即可。



- 业务微服务服务（如mafeng-user和Config Server都向Eureka-Server注册，所以mafeng-user可以通过Eureka获取ConfigServer服务列表。
- mafeng-user通过负载均衡策略，在多个Config Server实例中选择一个作为获取配置请求的请求目标。
- 到选择到Config-Server实例请求拉取配置信息。

Config-Server端

1) 添加eureka-client依赖

```
<!-- eureka client 端包 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

2) yml文件添加eureka注册配置

```
eureka:
  client:
    serviceUrl:
      register-with-eureka: true
      fetch-registry: true
      defaultZone:
http://mafeng:mafeng123@localhost:8761/eureka/eureka/,http://mafeng:mafeng123@localhost:8762/eureka/eureka/,http://mafeng:mafeng123@localhost:8763/eureka/eureka/
```

3) 添加注解

```
@EnabledDiscoveryClient
```

4) 启动所有Config-Servier，就可以看到三台Config-Server已经成功注册。

Config-Client端

配置如下：

```
spring:
  application:
    name: mafeng-user
  cloud:
    config:
      #uri: http://localhost:12000
      label: master
      profile: dev
      username: mafeng
      password: mafeng123
      discovery:
        enabled: true
        service-id: config-server # 这里由原来直接写路径改成serviceName

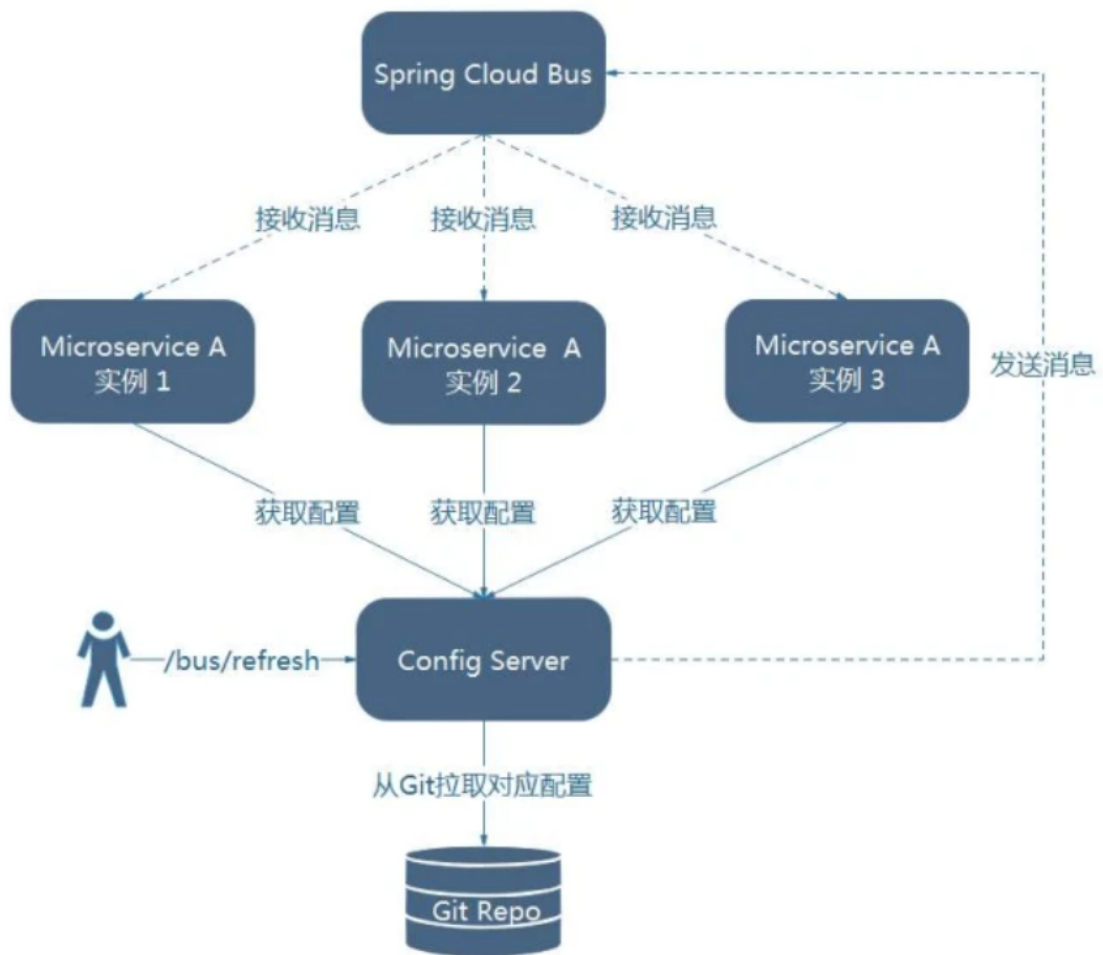
eureka:
  client:
    serviceUrl:
      register-with-eureka: true
      fetch-registry: true
      defaultZone:
http://mafeng:mafeng123@localhost:8761/eureka/eureka/,http://mafeng:mafeng123@localhost:8762/eureka/eureka/,http://mafeng:mafeng123@localhost:8763/eureka/eureka/
```

Spring Cloud Bus介绍

之前的课程，我们说过actuator只适合手动刷新一个微服务配置，如果比较多微服务配置都发生变更，需要逐个手动刷新配置，比较麻烦！如果需要实现一键更新所有微服务配置，这时需要用到Spring Cloud Bus（消息总线）！

Spring Cloud Bus将分布式系统的微服务通过轻量级的消息中间件连接起来，通过消息中间件（MQ）广播状态更改（例如配置更改）或其他管理指令。总线就像是横向扩展的Spring Boot应用程序的分布式的actuator，它也可以用作微服务之间的通信渠道。

消息总线的核心工作逻辑：将收到的消息（事件）批量分发到所有或部分的微服务节点应用，微服务应用在接到消息之后做出相应的业务处理相应。



安装RabbitMQ服务

拉取rabbitmq镜像

```
docker pull rabbitmq:management
```

使用rabbitMQ镜像运行一个实例，暴露5672和15672端口。

```
docker run -d -p 5672:5672 -p 15672:15672 --name rabbitmq rabbitmq:management
```

访问管理界面的地址就是 <http://192.168.66.133:15672>，可以使用默认的账户登录，用户名和密码都guest。

Config Server集成Bus

导入依赖

```

<!--添加基于RabbitMQ的Spring Cloud Bus的支持 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
<!--actuator提供刷新请求接口: "/actuator/bus-refresh" -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

```

配置yml

加上如下配置：指定RabbitMQ消息中间件的服务地址。因为我们的Spring Cloud Bus作为消息发布者向rabbitMQ中放入“配置刷新”消息，所以需要rabbit的地址及认证信息。

```

spring:
  rabbitmq:
    host: 192.168.66.133
    port: 5672
    virtual-host: /
    username: guest
    password: guest

management:
  endpoints:
    web:
      exposure:
        include: bus-refresh

```

注意事项：

如果你的Config Server也引入spring-boot-starter-security，把下面的代码配置加入。否则通过“/actuator/bus-refresh”访问将会报错：401，没有权限访问。

```

package cn.mf5.config.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;

@Configuration
@EnableWebSecurity
public class ConfigServerWebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

```

```
// 都是无状态请求，不需要session，节省资源

http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.NEVER);
//关闭csrf跨站请求防御
http.csrf().disable();
// 所有的请求必须登录认证后才能访问
http.authorizeRequests().antMatchers("/bus-refresh", "/actuator/bus-
refresh").permitAll().anyRequest().authenticated().and().httpBasic();
    }
}
```

Config Client集成Bus

导入依赖

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

yml配置

加上RabbitMQ消息队列的配置（加到gitee仓库对应微服务的配置文件中）

```
spring:
  rabbitmq:
    host: 192.168.66.133
    port: 5672
    username: guest
    password: guest
```

测试Spring Cloud Bus

gitee配置修改后，使用以下地址请求Config-Server，就可以实现配置刷新啦！

<http://localhost:12000/actuator/bus-refresh>

5、分布式配置管理（二）：Apollo

Apollo介绍



Apollo - A reliable configuration management system

Apollo（阿波罗）是一款可靠的分布式配置管理中心，诞生于携程框架研发部，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。

服务端基于Spring Boot和Spring Cloud开发，打包后可以直接运行，不需要额外安装Tomcat等应用容器。

Java客户端不依赖任何框架，能够运行于所有Java运行时环境，同时对Spring/Spring Boot环境也有较好的支持。

.Net客户端不依赖任何框架，能够运行于所有.Net运行时环境。

Apollo的核心特征

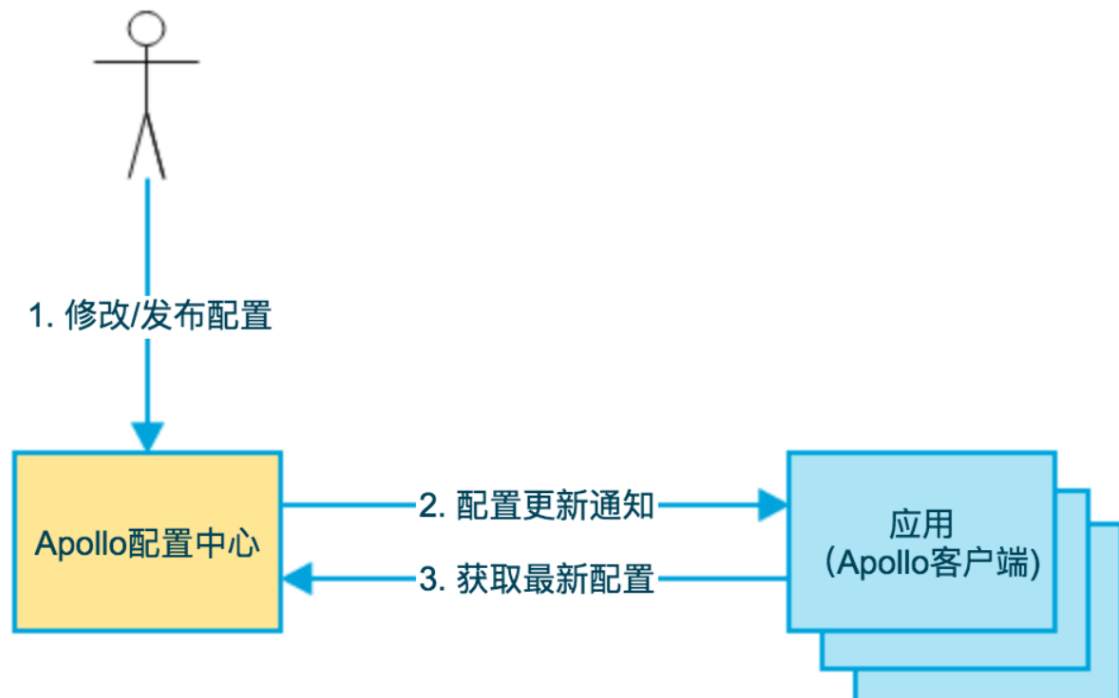
- **统一管理不同环境、不同集群的配置**
 - Apollo提供了一个统一界面集中式管理不同环境（environment）、不同集群（cluster）、不同命名空间（namespace）的配置。
 - 同一份代码部署在不同的集群，可以有不同的配置，比如zk的地址等
 - 通过命名空间（namespace）可以很方便的支持多个不同应用共享同一份配置，同时还允许应用对共享的配置进行覆盖
 - 配置界面支持多语言（中文，English）
- **配置修改实时生效（热发布）**
 - 用户在Apollo修改完配置并发布后，客户端能实时（1秒）接收到最新的配置，并通知到应用程序。
- **版本发布管理**
 - 所有的配置发布都有版本概念，从而可以方便的支持配置的回滚。
- **灰度发布**
 - 支持配置的灰度发布，比如点了发布后，只对部分应用实例生效，等观察一段时间没问题后再推给所有应用实例。
- **权限管理、发布审核、操作审计**
 - 应用和配置的管理都有完善的权限管理机制，对配置的管理还分为了编辑和发布两个环节，从而减少人为的错误。
 - 所有的操作都有审计日志，可以方便的追踪问题。
- **客户端配置信息监控**
 - 可以方便的看到配置在被哪些实例使用
- **提供Java和.Net原生客户端**
 - 提供了Java和.Net的原生客户端，方便应用集成
 - 支持Spring Placeholder，Annotation和Spring Boot的ConfigurationProperties，方便应用使用（需要Spring 3.1.1+）

- 同时提供了Http接口，非Java和.Net应用也可以方便的使用
- **提供开放平台API**
 - Apollo自身提供了比较完善的统一配置管理界面，支持多环境、多数据中心配置管理、权限、流程治理等特性。
 - 不过Apollo出于通用性考虑，对配置的修改不会做过多限制，只要符合基本的格式就能够保存。
 - 在我们的调研中发现，对于有些使用方，它们的配置可能会有比较复杂的格式，如xml, json，需要对格式做校验。
 - 还有一些使用方如DAL，不仅有特定的格式，而且对输入的值也需要进行校验后方可保存，如检查数据库、用户名和密码是否匹配。
 - 对于这类应用，Apollo支持应用方通过开放接口在Apollo进行配置的修改和发布，并且具备完善的授权和权限控制
- **部署简单**
 - 配置中心作为基础服务，可用性要求非常高，这就要求Apollo对外部依赖尽可能地少
 - 目前唯一的外部依赖是MySQL，所以部署非常简单，只要安装好Java和MySQL就可以让Apollo跑起来
 - Apollo还提供了打包脚本，一键就可以生成所有需要的安装包，并且支持自定义运行时参数

Apollo基本工作流程

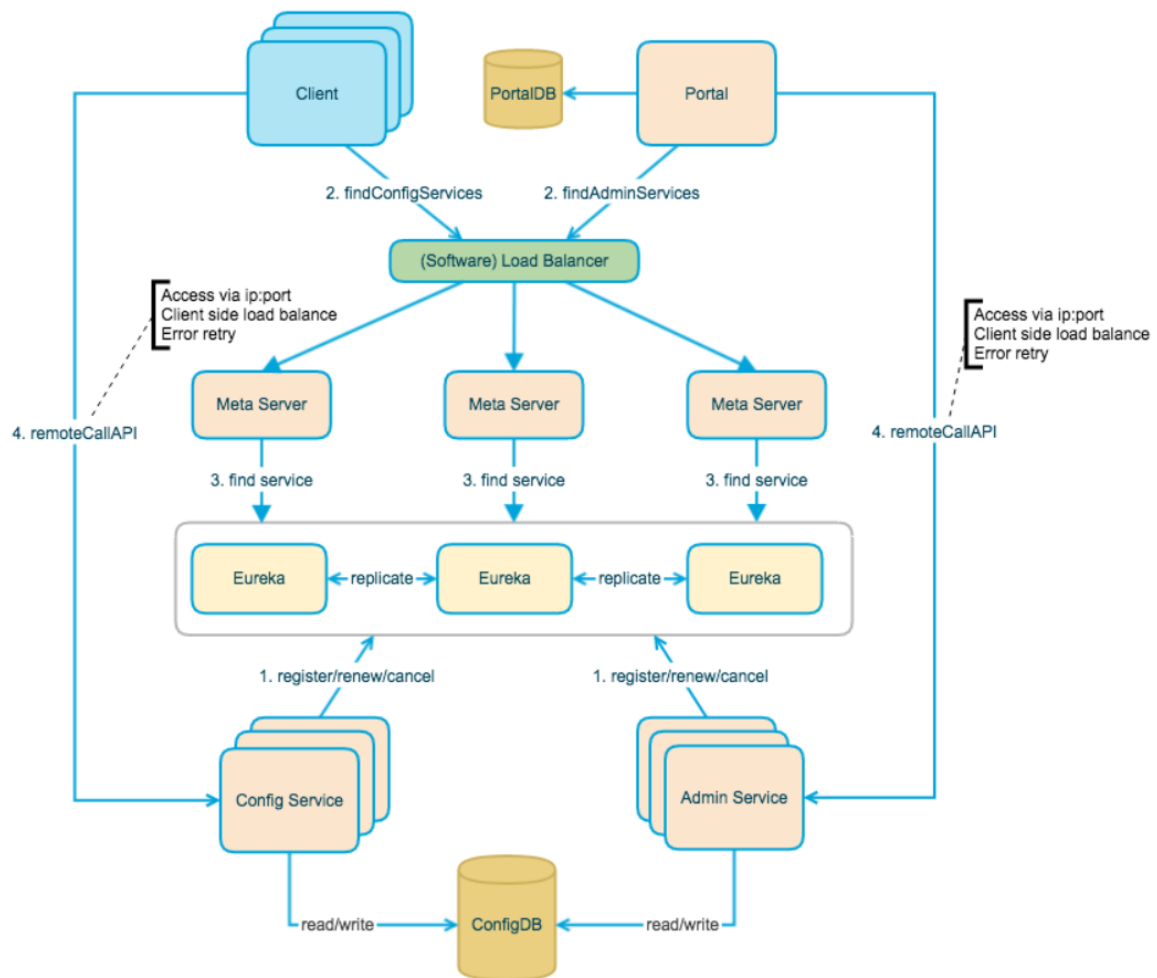
如下即是Apollo的基础模型：

1. 用户在配置中心对配置进行修改并发布
2. 配置中心通知Apollo客户端有配置更新
3. Apollo客户端从配置中心拉取最新的配置、更新本地配置并通知到应用



Apollo核心架构图

下图是Apollo架构模块的概览



下面是Apollo的七个模块，其中四个模块是和功能相关的核心模块，另外三个模块是辅助服务发现的模块：

四个核心模块及其主要功能

1. ConfigService

2.
 - 提供配置获取接口
 - 提供配置推送接口
 - 服务于Apollo客户端

3. AdminService

4.
 - 提供配置管理接口
 - 提供配置修改发布接口
 - 服务于管理界面Portal

5. Client

6.
 - 为应用获取配置，支持实时更新
 - 通过MetaServer获取ConfigService的服务列表
 - 使用客户端软负载SLB方式调用ConfigService

7. Portal

8.
 - 配置管理界面
 - 通过MetaServer获取AdminService的服务列表
 - 使用客户端软负载SLB方式调用AdminService

三个辅助服务发现模块

1. Eureka

2.
 - 用于服务发现和注册
 - Config/AdminService注册实例并定期报心跳

- 和ConfigService住在一起部署

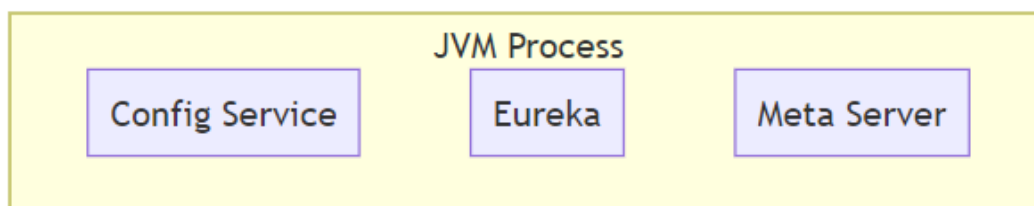
3. MetaServer

- 4. ◦ Portal通过域名访问MetaServer获取AdminService的地址列表
- Client通过域名访问MetaServer获取ConfigService的地址列表
- 相当于一个Eureka Proxy
- 逻辑角色，和ConfigService住在一起部署

5. NginxLB

- 6. ◦ 和域名系统配合，协助Portal访问MetaServer获取AdminService地址列表
- 和域名系统配合，协助Client访问MetaServer获取ConfigService地址列表
- 和域名系统配合，协助用户访问Portal进行配置管理

注意：为了简化部署，我们实际上会把Config Service、Eureka和Meta Server三个逻辑角色部署在同一个JVM进程中



搭建单环境的Apollo服务

Apollo目前支持以下环境：

- DEV
 - 开发环境
- FAT
 - 测试环境，相当于alpha环境(功能测试)
- UAT
 - 集成环境，相当于beta环境（回归测试）
- PRO
 - 生产环境

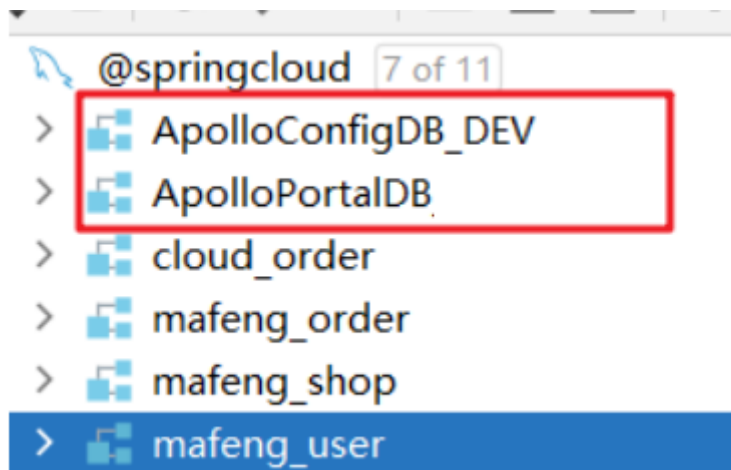
接下来以搭建DEV开发环境为例，演示单环境Apollo服务的搭建过程（Docker方式搭建）。

Apollo官网：<https://github.com/apolloconfig/apollo>

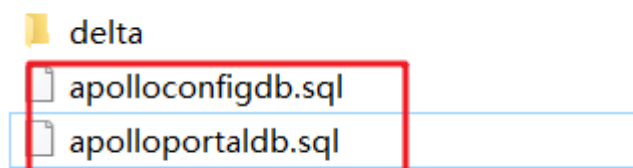
DockerHub镜像地址：<https://hub.docker.com/u/apolloconfig>

导入Config和Portal的SQL

在MySQL8的数据中创建两个数据库，如下：



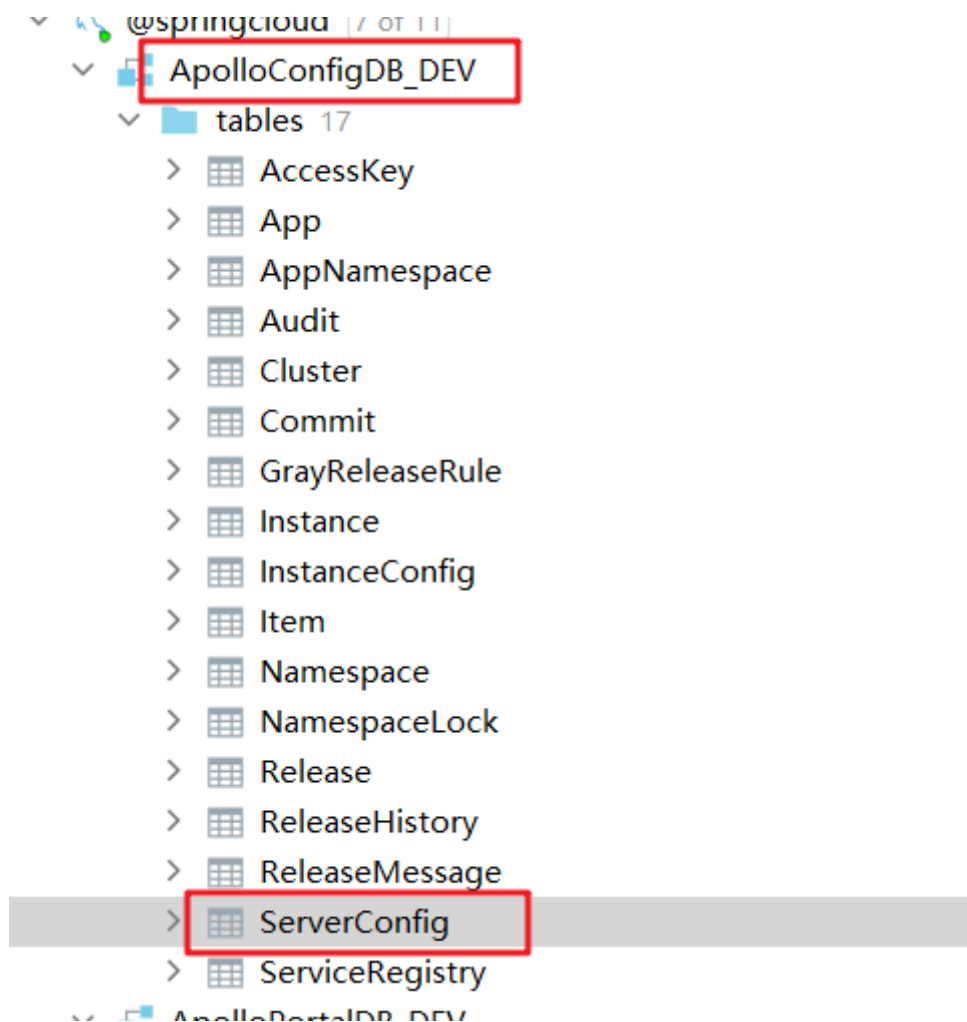
在Apolloa源码的scripts/sql目录中，找到sql脚本并导入以上Database：
名称



创建Config、Admin、Portal容器

Docker部署文档: <https://www.apolloconfig.com/#/zh/deployment/distributed-deployment-guide?id=23-docker%E9%83%A8%E7%BD%B2>

修改ApolloConfigDB_DEV库的ServerConfig表



修改Eureka注册地址:

ServerConfig					
WHERE					
ORDER BY					
	Id	Key	Cluster	Value	Comment
1	1	eureka.service.url	default	http://192.168.66.133:8082/eureka/	Eureka注册地址
2	2	namespace.lock.switch	default	false	一次只允许一个客户端修改namespace
3	3	item.key.length.limit	default	128	item key 最大长度
4	4	item.value.length.limit	default	20000	item value 最大长度
5	5	config-service.cache.enabled	default	false	ConfigService缓存是否启用

8082端口是Eureka对外暴露的服务注册端口。

下载Apollo Config Service镜像

```
docker pull apolloconfig/apollo-configservice:2.0.1
```

创建Apollo Config Service容器

```
docker run -p 8082:8082 \
-e
SPRING_DATASOURCE_URL="jdbc:mysql://192.168.66.133:3306/ApolloConfigDB_DEV?
characterEncoding=utf8" \
-e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root \
-e EUREKA_INSTANCE_IP_ADDRESS=192.168.66.133 \
-e SERVER_PORT=8082 \
-d -v /tmp/logs:/opt/logs --name apollo-configservice-dev
apolloconfig/apollo-configservice:2.0.1
```

参数说明:

- SPRING_DATASOURCE_URL: 对应环境ApolloConfigDB的地址
- SPRING_DATASOURCE_USERNAME: 对应环境ApolloConfigDB的用户名
- SPRING_DATASOURCE_PASSWORD: 对应环境ApolloConfigDB的密码

Apolloa Config Service启动成功后, 访问: <http://192.168.66.133:8082/>

可以看到服务已经成功注册到Eureka

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
APOLLO-ADMINSERVICE	n/a (1)	(1)	UP (1) - d47eedef76572:apollo-adminservice:8092
APOLLO-CONFIGSERVICE	n/a (1)	(1)	UP (1) - c0fe80783440:apollo-configservice:8082
General Info			

下载Apollo Admin Service镜像

```
docker pull apolloconfig/apollo-adminservice:2.0.1
```

创建Apollo Admin Service容器

```
docker run -p 8092:8092 \
-e
SPRING_DATASOURCE_URL="jdbc:mysql://192.168.66.133:3306/ApolloConfigDB_DEV?
characterEncoding=utf8" \
-e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root \
-e EUREKA_INSTANCE_IP_ADDRESS=192.168.66.133 \
-e SERVER_PORT=8092 \
-d -v /tmp/logs:/opt/logs --name apollo-adminservice-dev
apolloconfig/apollo-adminservice:2.0.1
```

参数说明:

- SPRING_DATASOURCE_URL: 对应环境ApolloConfigDB的地址
- SPRING_DATASOURCE_USERNAME: 对应环境ApolloConfigDB的用户名
- SPRING_DATASOURCE_PASSWORD: 对应环境ApolloConfigDB的密码

Apolloa Admin Service启动成功后, 访问: <http://192.168.66.133:8082/>

可以看到服务已经成功注册到Eureka

Application	AMIs	Availability Zones	Status
APOLLO-ADMINSERVICE	n/a (1)	(1)	UP (1) - d47eedef76572:apollo-adminservice:8092
APOLLO-CONFIGSERVICE	n/a (1)	(1)	UP (1) - c0fe80783440:apollo-configservice:8082

下载Apollo Portal镜像

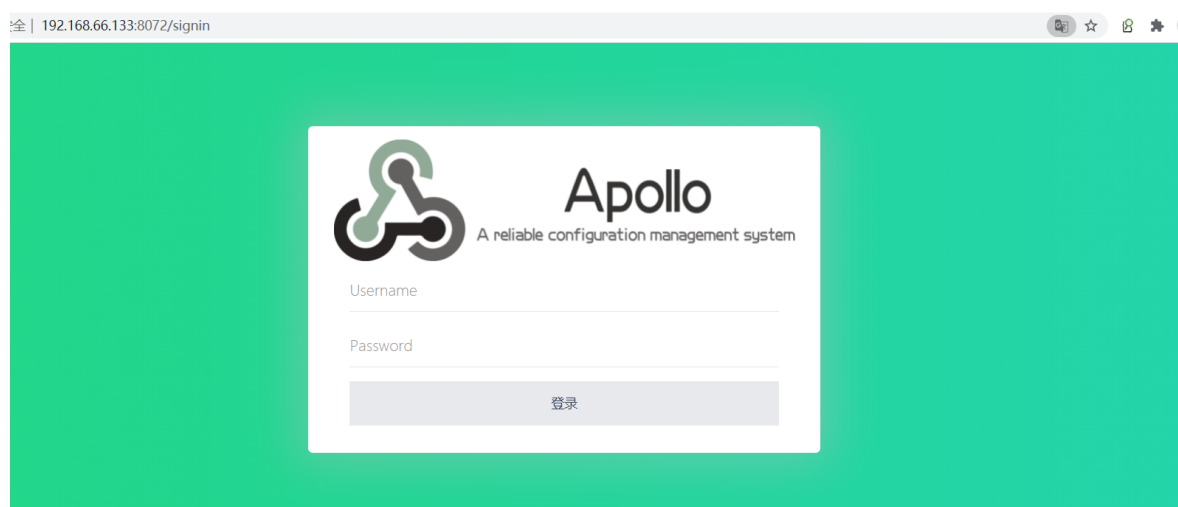
```
docker pull apolloconfig/apollo-portal:2.0.1
```

创建Apollo Portal容器

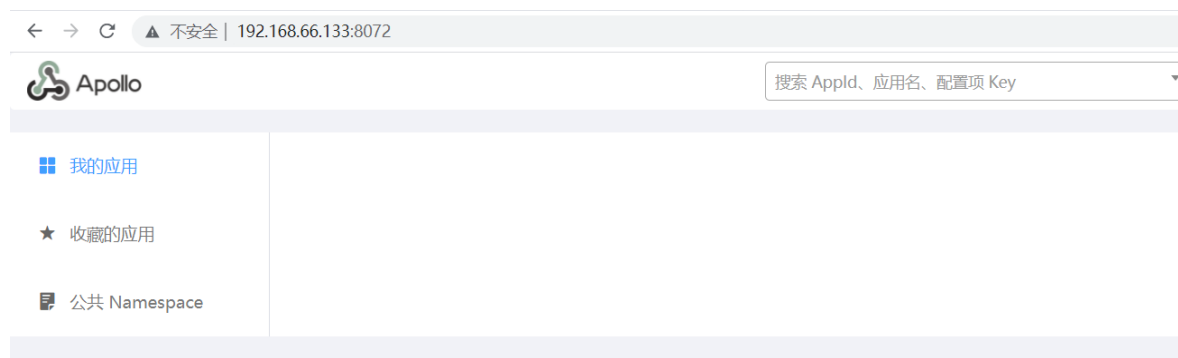
```
docker run -p 8072:8070 \
  -e SPRING_DATASOURCE_URL="jdbc:mysql://192.168.66.133:3306/ApolloPortalDB?characterEncoding=utf8" \
  -e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root \
  -e APOLLO_PORTAL_ENVS=dev \
  -e DEV_META=http://192.168.66.133:8082 \
  -d -v /tmp/logs:/opt/logs --name apollo-portal apolloconfig/apollo-portal:2.0.1
```

访问Portal

访问: <http://192.168.66.133:8072/>

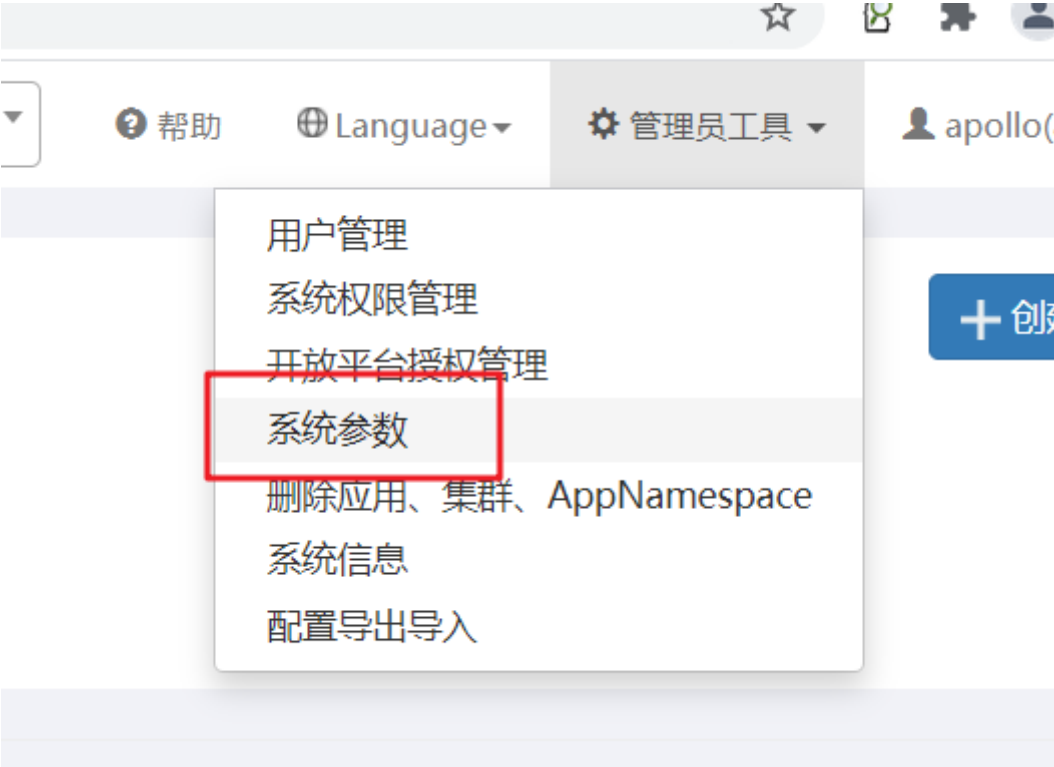


账户/密码为: apollo/admin, 登录后进入主页



Apollo项目管理

管理/添加部门



key输入organizations，点击查询

* Key

查询

(修改配置前请先查询该配置信息)

* Value

```
[{"orgId":"TEST1","orgName":"样例部门1"}, {"orgId":"TEST2","orgName":"样例部门2"}]
```

Comment

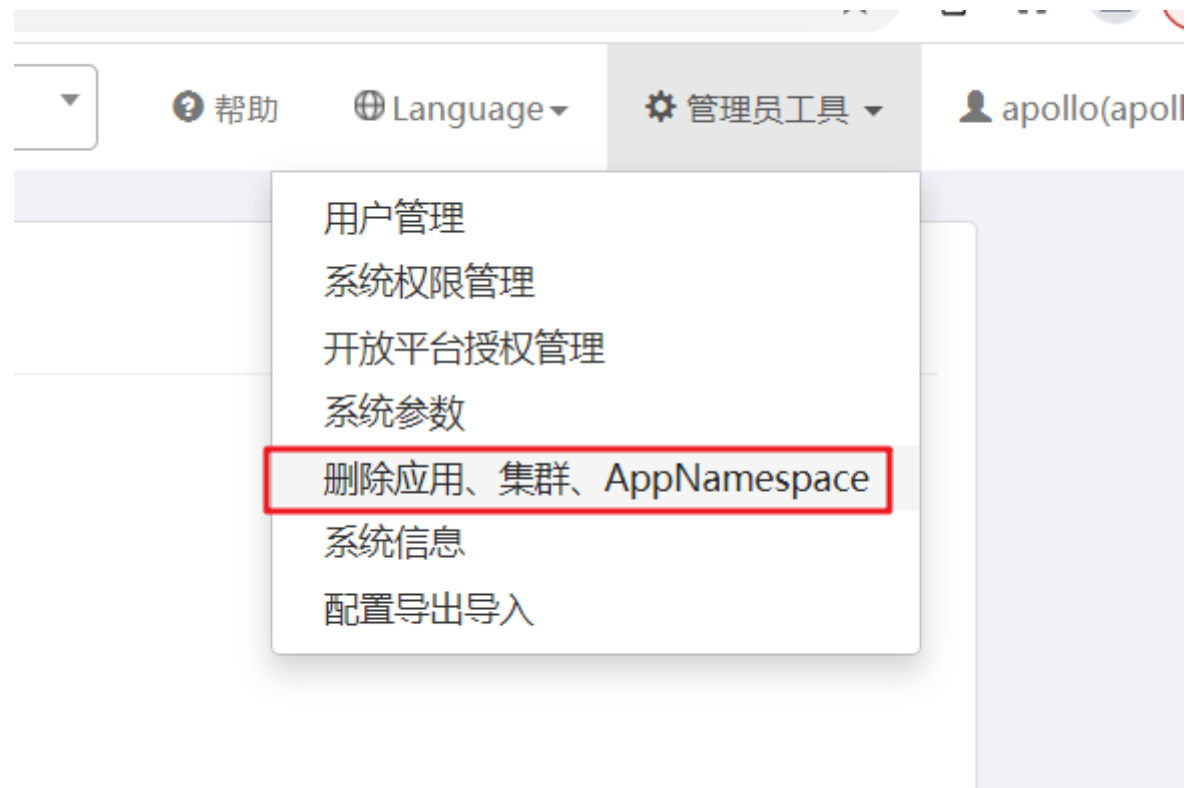
保存

按照json格式添加新部门，添加保存

点击应用



删除应用



Apollo发布配置

在默认namespace添加配置

添加配置项 (温馨提示: 可以通过文本模式批量添加配置)

* Key mafeng.password

Value 111111

注意: 隐藏字符 (空格、换行符、制表符Tab) 容易导致配置出错, 如果需要检测 \

Comment

* 选择集群

☐ 环境

集群

☒ DEV

default

配置默认没有发布, 需要手动发布

私有 properties

application 有修改 1

表格 文本 更改历史 实例列表 0

发布 回滚 发布历史

点击发布

过滤配置 同步配置 撤销配置

Tips: 此 Namespace 从来没有发布过, Apollo 客户端将获取不到配置并记录 404 日志信息, 请及时发布。

发布状态 ↑↓	Key ↑↓	Value	备注	最后修改人 ↑↓	最后修
未发布	mafeng.password 新	111111		apollo(apollo)	2022-10-25 2

私有 properties

application 发布 回滚

表格 文本 更改历史 实例列表 0

过滤配置 同步配置

发布状态 ↑↓	Key ↑↓	Value	备注	最后修改人 ↑↓
已发布	mafeng.password	111111		apollo(apollo)

Apollo-Java客户端拉取配置

导入apollo客户端依赖

```
<dependency>
    <groupId>com.ctrip.framework.apollo</groupId>
    <artifactId>apollo-client</artifactId>
    <version>2.0.1</version>
</dependency>
```

编写测试代码

```
package cn.mf5.test;

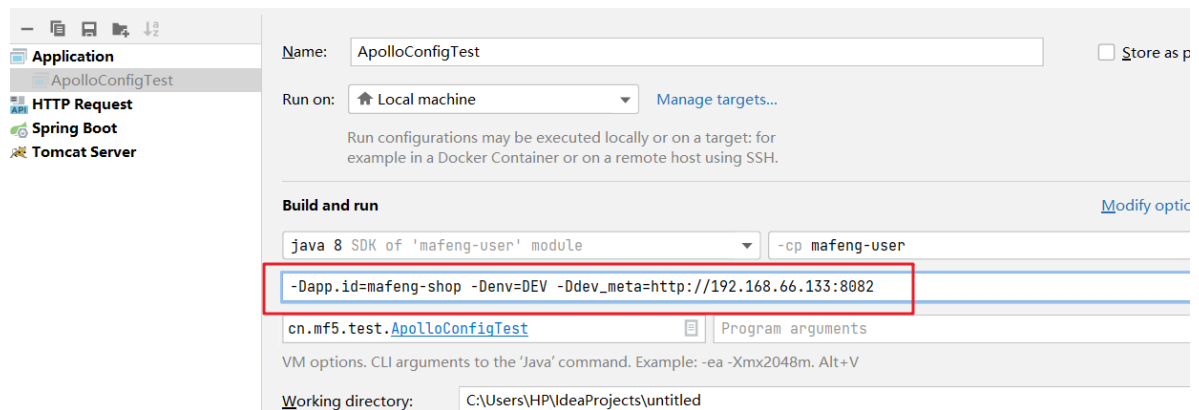
import com.ctrip.framework.apollo.Config;
import com.ctrip.framework.apollo.ConfigService;

public class ApolloConfigTest {

    public static void main(String[] args) {
        //获取配置对象
        Config appConfig = ConfigService.getAppConfig();
        //通过key取value
        String password = appConfig.getProperty("mafeng.password", "0000000");
        System.out.println(password);
    }
}
```

添加VM运行参数

```
-Dapp.id=mafeng-shop -Denv=DEV -Ddev_meta=http://192.168.66.133:8082
```



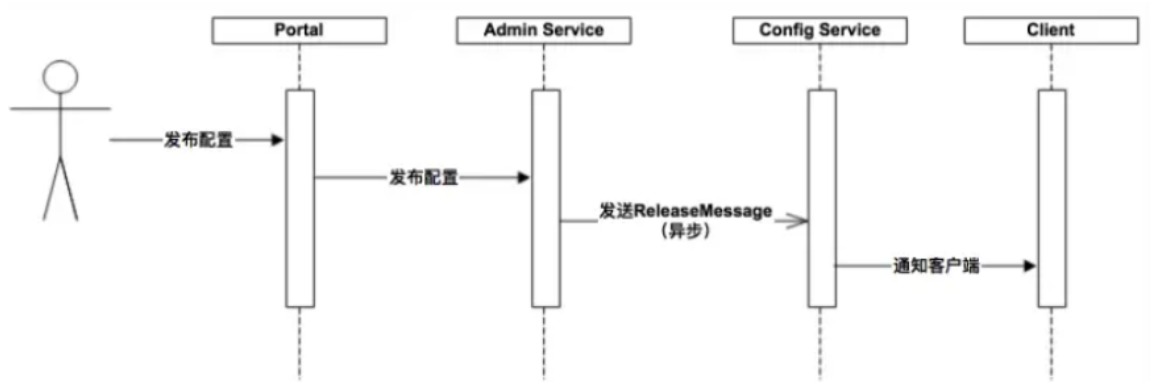
执行效果如下：

```
in: ApolloConfigTest x
23:23:53.887 [main] WARN com.ctrip.framework.apollo.internals.Default
23:23:53.891 [main] WARN com.ctrip.framework.apollo.core.utils.Resou
23:23:53.891 [main] INFO com.ctrip.framework.apollo.core.MetaDomainC
23:23:54.544 [main] DEBUG com.ctrip.framework.apollo.internals.Remot
23:23:54.838 [main] DEBUG com.ctrip.framework.apollo.internals.Remot
23:23:54.840 [main] DEBUG com.ctrip.framework.apollo.internals.Remot
23:23:54.840 [main] DEBUG com.ctrip.framework.apollo.internals.Remot
23:23:54.841 [Apollo-RemoteConfigLongPollService-1] DEBUG com.ctrip.
23:23:54.847 [main] DEBUG com.ctrip.framework.apollo.spi.DefaultConf
111111
```

Apollo配置更新原理分析

服务端设计

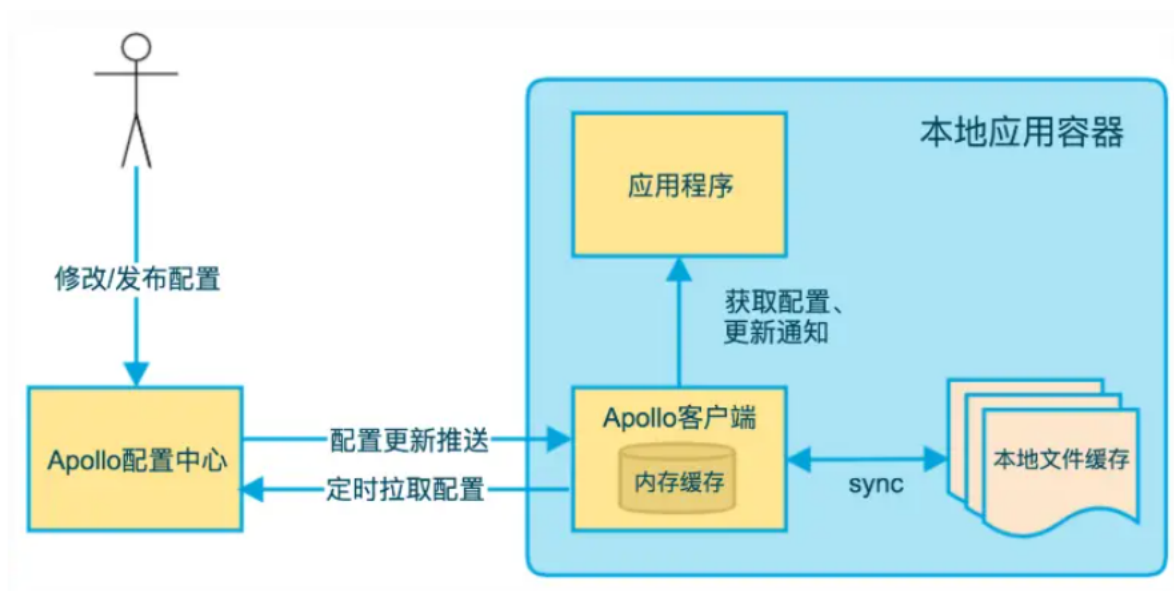
在配置中心中，一个重要的功能就是配置发布后实时推送到客户端。下面我们简要看一下这块是怎么设计实现的。



上图简要描述了配置发布的大致过程：

1. 用户在Portal操作配置发布
2. Portal调用Admin Service的接口操作发布
3. Admin Service发布配置后，发送ReleaseMessage给各个Config Service
4. Config Service收到ReleaseMessage后，通知对应的客户端

客户端设计



上图简要描述了Apollo客户端的实现原理：

1. 客户端和服务端保持了一个长连接，从而能第一时间获得配置更新的推送。（通过 `Http Long Polling` 实现）
2. 客户端还会定时从Apollo配置中心服务端拉取应用的最新配置。
 - 这是一个fallback机制，为了防止推送机制失效导致配置不更新
 - 客户端定时拉取会上报本地版本，所以一般情况下，对于定时拉取的操作，服务端都会返回 `304 - Not Modified`
 - 定时频率默认为每5分钟拉取一次，客户端也可以通过在运行时指定System Property: `apollo.refreshInterval` 来覆盖，单位为分钟。
3. 客户端从Apollo配置中心服务端获取到应用的最新配置后，会保存在内存中
4. 客户端会把从服务端获取到的配置在本地文件系统缓存一份
 - 在遇到服务不可用，或网络不通的时候，依然能从本地恢复配置
5. 应用程序可以从Apollo客户端获取最新的配置、订阅配置更新通知

通过namespace管理项目配置

什么是namespace

- namespace是配置项的集合，类似于一个配置文件的概念。
- application的namespace: Apollo在创建项目的时候，都会默认创建一个“application”的namespace。顾名思义，“application”是给应用自身使用的。

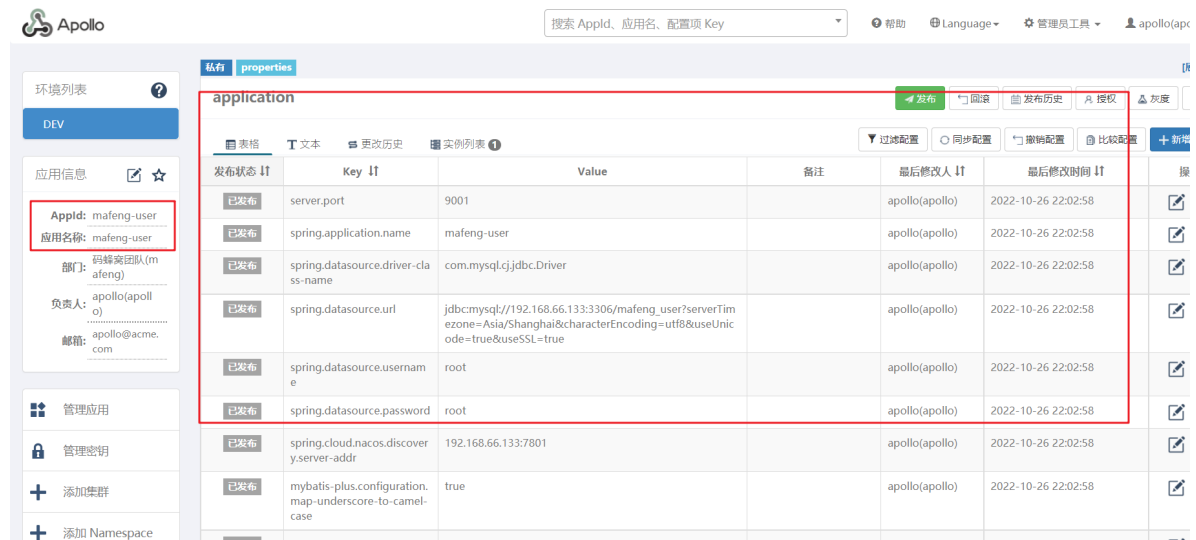
namespace的类别

- 私有类型：具有private权限。
- 公共类型：具有public权限，公共类型的namespace通过名称标识，因此名称必须全局唯一。具体使用场景如下：
 - 部门级别共享的配置
 - 小组级别共享的配置
 - 几个项目之间共享的配置
 - 中间件客户端的配置
- 关联类型（继承类型）

通过namespace发布配置

yaml和properties互转: <https://www.toyaml.com/index.html>

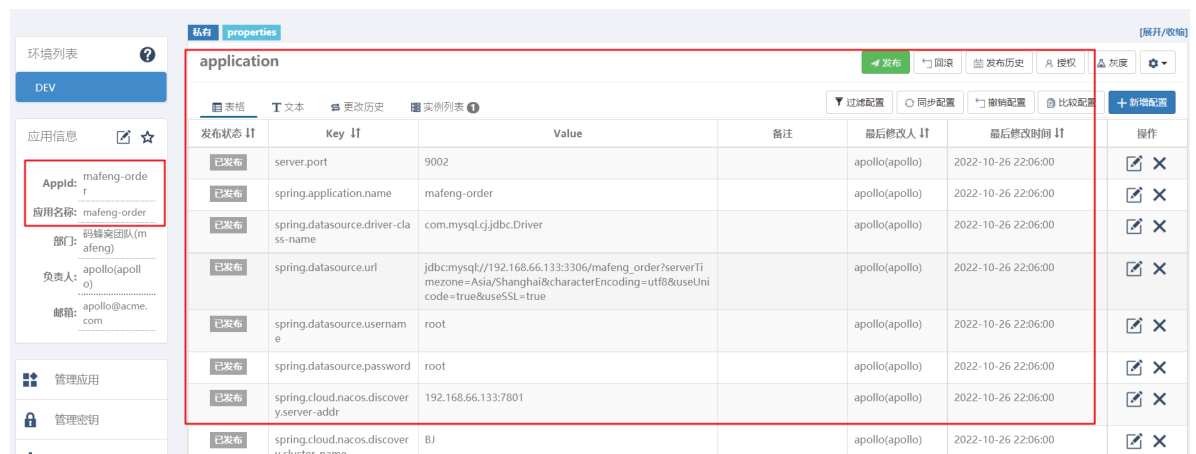
创建新的应用mafeng-user, 把yaml配置先转为properties格式, 再填入application默认namespace中



The screenshot shows the Apollo console interface for the 'mafeng-user' application. The application is in the 'DEV' environment. The configuration table shows various properties like server.port, spring.application.name, and spring.datasource.driver-class-name, all with values and a status of '已发布' (Released).

发布状态	Key	Value	备注	最后修改人	最后修改时间	操作
已发布	server.port	9001		apollo(apollo)	2022-10-26 22:02:58	✎
已发布	spring.application.name	mafeng-user		apollo(apollo)	2022-10-26 22:02:58	✎
已发布	spring.datasource.driver-class-name	com.mysql.cj.jdbc.Driver		apollo(apollo)	2022-10-26 22:02:58	✎
已发布	spring.datasource.url	jdbc:mysql://192.168.66.133:3306/mafeng_user?serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=true		apollo(apollo)	2022-10-26 22:02:58	✎
已发布	spring.datasource.username	root		apollo(apollo)	2022-10-26 22:02:58	✎
已发布	spring.datasource.password	root		apollo(apollo)	2022-10-26 22:02:58	✎
已发布	spring.cloud.nacos.discovery.server-addr	192.168.66.133:7801		apollo(apollo)	2022-10-26 22:02:58	✎
已发布	mybatis-plus.configuration.map-underscore-to-camel-case	true		apollo(apollo)	2022-10-26 22:02:58	✎

创建新的应用mafeng-order, 把yaml配置先转为properties格式, 再填入application默认namespace中



The screenshot shows the Apollo console interface for the 'mafeng-order' application. The application is in the 'DEV' environment. The configuration table shows various properties like server.port, spring.application.name, and spring.datasource.driver-class-name, all with values and a status of '已发布' (Released).

发布状态	Key	Value	备注	最后修改人	最后修改时间	操作
已发布	server.port	9002		apollo(apollo)	2022-10-26 22:06:00	✎ ✕
已发布	spring.application.name	mafeng-order		apollo(apollo)	2022-10-26 22:06:00	✎ ✕
已发布	spring.datasource.driver-class-name	com.mysql.cj.jdbc.Driver		apollo(apollo)	2022-10-26 22:06:00	✎ ✕
已发布	spring.datasource.url	jdbc:mysql://192.168.66.133:3306/mafeng_order?serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=true		apollo(apollo)	2022-10-26 22:06:00	✎ ✕
已发布	spring.datasource.username	root		apollo(apollo)	2022-10-26 22:06:00	✎ ✕
已发布	spring.datasource.password	root		apollo(apollo)	2022-10-26 22:06:00	✎ ✕
已发布	spring.cloud.nacos.discovery.server-addr	192.168.66.133:7801		apollo(apollo)	2022-10-26 22:06:00	✎ ✕
已发布	spring.cloud.nacos.discovery.cluster-name	BJ		apollo(apollo)	2022-10-26 22:06:00	✎ ✕

记得要发布配置!!!

在mafeng-user或mafeng-order创建public类型的namespace, 把nacos配置抽取在这个namespace中

创建 Namespace

关联公共 Namespace

AppId mafeng-user

* 类型 ☐ private ☒ public

自动添加部门前缀 ☒ mafeng.
(公共 Namespace 的名称需要全局唯一，添加部门前缀有助于保证全局唯一性)

* 名称

mafeng.

properties ▼

 mafeng.nacos

备注

nacos配置

提交

公共 properties

mafeng.nacos

发布 回滚 发布历史 授权 灰度

表格 文本 更改历史 实例列表 2

过滤配置 同步配置 撤销配置 比较配置 新增配置

发布状态	Key	Value	备注	最后修改人	最后修改时间	操作
已发布	spring.cloud.nacos.discovery.server-addr	192.168.66.133:7801		apollo(apollo)	2022-10-26 22:19:49	

Apollo与SpringCloud项目整合

mafeng-user和mafeng-order微服务使用以下步骤整合Apollo

导入Apollo-Java客户端依赖

```
<dependency>
    <groupId>com.ctrip.framework.apollo</groupId>
    <artifactId>apollo-client</artifactId>
    <version>2.0.1</version>
</dependency>
```

在resources目录下创建 apollo-env.properties 文件，指定各种环境的meta-server地址

```
dev.meta=http://192.168.66.133:8082
# pro.meta=http://192.168.66.133:8083    后续加上PRO环境后再使用
```

配置application.yml

mafeng-user微服务的application.yml

```
app:
  id: mafeng-user
apollo:
  bootstrap:
    enabled: true
  namespaces: application,mafeng.nacos
```


mafeng-order微服务的application.yml

```
app:
  id: mafeng-order
apollo:
  bootstrap:
    enabled: true
  namespaces: application,mafeng.nacos
```

启动类添加 @EnableApolloConfig 注解

```
import ...
/**
 * 用户模块
 */
@SpringBootApplication
@MapperScan("cn.mf5.user.mapper")
//@EnableEurekaClient
@EnableDiscoveryClient //开启服务注册
@EnableApolloConfig
public class UserApplication {
    public static void main(String[] args) { SpringApplica
    }
```

项目启动时，添加运行参数

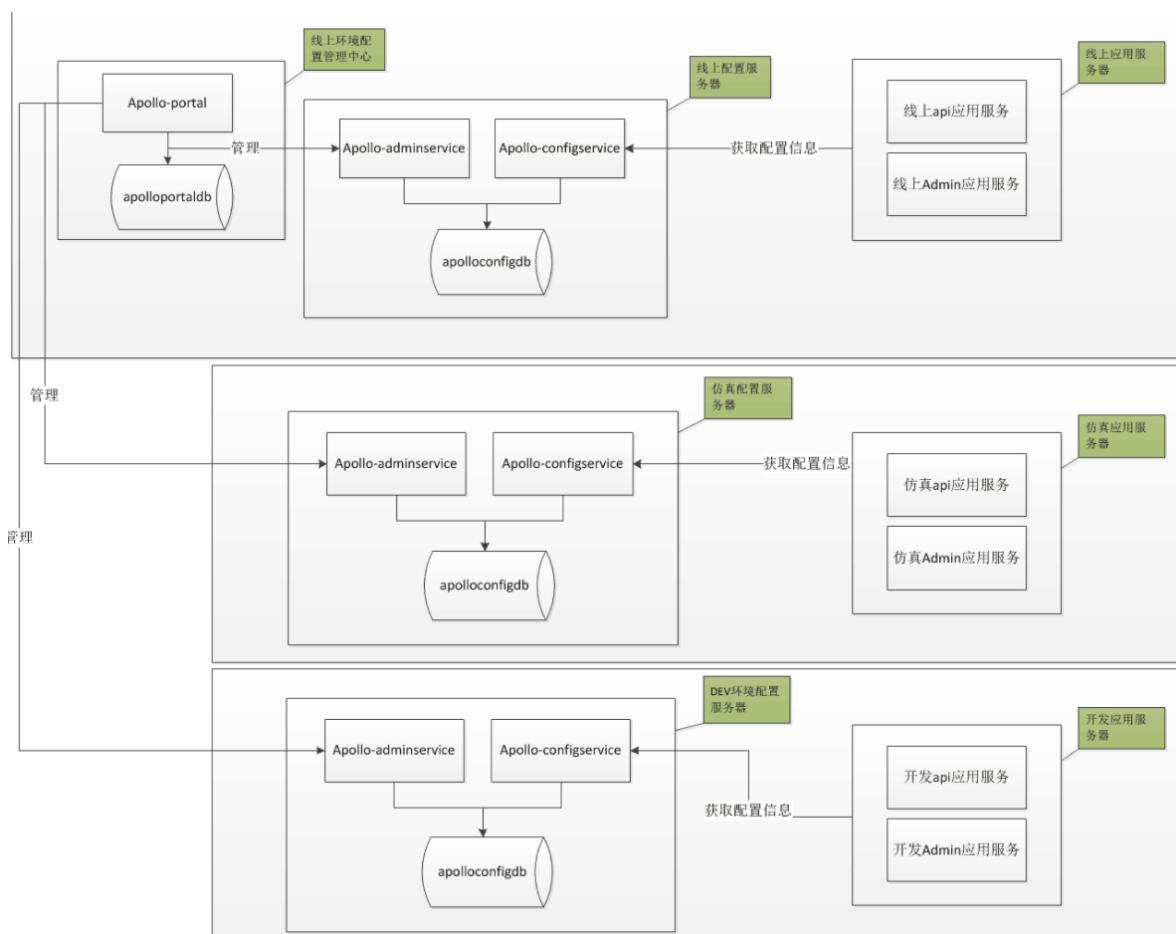
Duration	Code Coverage	Logs
Class:	cn.mf5.user.UserApplication	
Environment		
Options:	<div>-Denv=DEV</div>	
Program arguments:		

Apollo配置实时刷新的作用域

配置中心	@Value	@ConfigurationProperties
Nacos	需要结合RefreshScope才能生效	需要结合RefreshScope才能生效
Apollo	不需要结合RefreshScope就能生效 (例外)	需要结合RefreshScope才能生效

配置中心 Apollo-portal Config	@Value 需要结合RefreshScope才能生效	@ConfigurationProperties 需要结合RefreshScope才能生效
---------------------------------	--------------------------------	--

搭建多环境的Apollo服务

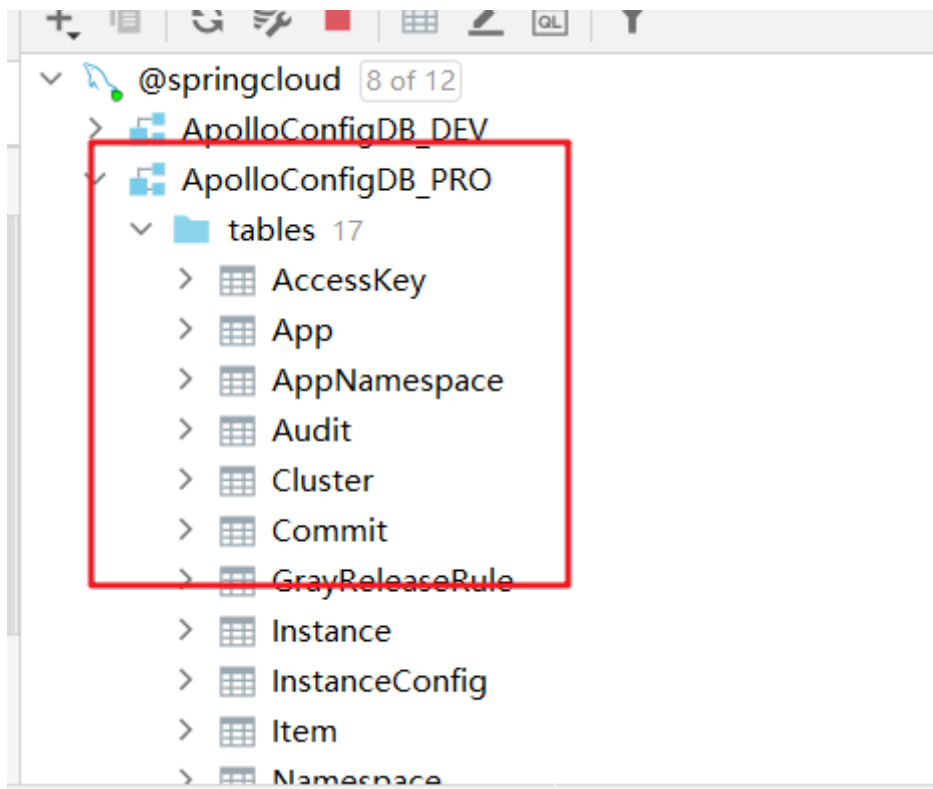


Config-service和Admin-Service的部署：每个环境（DEV，PRO等）需要分别部署一套Config-Service和Admin-Service实例

Apollo-Portail的部署：Portal界面应用本身不需要进行集群高可用部署，因为它就是一个web页面，供给团队内部进行配置管理使用的。即使挂掉了也没有很大关系，也不影响线上服务的正常运行。

导入PRO环境的SQL

在MySQL8的数据中为PRO环境创建一个数据库，如下：



修改ServerConfig表的Eureka注册地址

ServerConfig					
WHERE					
ORDER BY					
	Id	Key	Cluster	Value	
1	1	eureka.service.url	default	http://192.168.66.133:8083/eureka/	Eur
2	2	namespace.lock.switch	default	false	一
3	3	item.key.length.limit	default	128	ite
4	4	item.value.length.limit	default	20000	ite
5	5	config-service.cache.e...	default	false	Cor

创建Config、Admin容器

创建Apollo Config Service容器

```
docker run -p 8083:8083 \
  -e
  SPRING_DATASOURCE_URL="jdbc:mysql://192.168.66.133:3306/ApolloConfigDB_PRO?
  characterEncoding=utf8" \
  -e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root \
  -e EUREKA_INSTANCE_IP_ADDRESS=192.168.66.133 \
  -e SERVER_PORT=8083 \
  -d -v /tmp/logs:/opt/logs --name apollo-configservice-pro
  apolloconfig/apollo-configservice:2.0.1
```

创建Apollo Admin Service容器

```
docker run -p 8093:8093 \
-e
SPRING_DATASOURCE_URL="jdbc:mysql://192.168.66.133:3306/ApolloConfigDB_PRO?
characterEncoding=utf8" \
-e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root \
-e EUREKA_INSTANCE_IP_ADDRESS=192.168.66.133 \
-e SERVER_PORT=8093 \
-d -v /tmp/logs:/opt/logs --name apollo-adminservice-pro
apolloconfig/apollo-adminservice:2.0.1
```

重新创建Portal容器

删除之前的Portal容器，重新创建新容器（加上PRO环境）

创建Apollo Portal容器

```
docker run -p 8072:8070 \
-e SPRING_DATASOURCE_URL="jdbc:mysql://192.168.66.133:3306/ApolloPortalDB?
characterEncoding=utf8" \
-e SPRING_DATASOURCE_USERNAME=root -e SPRING_DATASOURCE_PASSWORD=root \
-e APOLLO_PORTAL_ENVS=dev,pro \
-e DEV_META=http://192.168.66.133:8082 \
-e PRO_META=http://192.168.66.133:8083 \
-d -v /tmp/logs:/opt/logs --name apollo-portal apolloconfig/apollo-
portal:2.0.1
```

补缺环境

重新登录Portal，补全PRO环境





同步配置到各环境

创建完PRO环境后，配置是空的，这时可以使用同步配置功能，把DEV的全部配置同步到PRO环境，然后PRO环境再修改个性化修改配置。



同步的 Namespaceapplication

同步到哪个集群

☒ 环境

集群

☐ DEV

default

☒ PRO

default

需要同步的配置

全选配置

按最后更新时间过滤 开始时间: 年 / 月 / 日 结束时间: 年 / 月 / 日 过滤 重置

<input checked="" type="checkbox"/>	Key ↑↓	Value	Create Time ↑↓	Update Time ↑↓
<input checked="" type="checkbox"/>	mybatis-plus.configuration.map-underscore-to-camel-case	true	2022-10-26 22:02:58	2022-10-26 22:19:41
<input checked="" type="checkbox"/>	mybatis-plus.type-aliases-package	cn.mf5.shop.pojo	2022-10-26 22:02:58	2022-10-26 22:19:41
<input checked="" type="checkbox"/>	logging.level.cn.mf5	debug	2022-10-26 22:02:58	2022-10-26 22:19:41
<input checked="" type="checkbox"/>	mafeng.password	111111	2022-10-26 22:04:01	2022-10-26 22:19:41
<input checked="" type="checkbox"/>	server.port	9001	2022-10-26 22:02:58	2022-10-26 22:02:58
<input checked="" type="checkbox"/>	spring.application.name	mafeng-user	2022-10-26 22:02:58	2022-10-26 22:02:58
<input checked="" type="checkbox"/>	spring.datasource.driver-class-name	com.mysql.cj.jdbc.Driver	2022-10-26 22:02:58	2022-10-26 22:02:58
<input checked="" type="checkbox"/>	spring.datasource.url	jdbc:mysql://192.168.66.133:3306/mafeng_user?serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=true	2022-10-26 22:02:58	2022-10-26 22:02:58

上一步

同步

返回到应用首页

环境:PRO 集群:default Namespace:application

同步后	Comment	操作
-----	---------	----

注意：但同步后的配置是没有发布的，需要手动发布！！

切换微服务添加PRO环境

apollo-env.properties添加PRO环境

```
dev.meta=http://192.168.66.133:8082
pro.meta=http://192.168.66.133:8083
```

运行环境参数，可以改为PRO启动试试

Configuration Code Coverage Logs

Main class:

cn.mf5.user.UserApplication

Environment

VM options:

-Denv=PRO

Program arguments:

Working directory:

```

ApplicationProvider : App ID is set to mafeng-user by app.id property from System Property
ApplicationProvider : app.label is not available from System Property and /META-INF/app.properties. It is set t
ServerProvider : Environment is set to [PRO] by JVM system property 'env'.
MetaServerProvider : Located meta services from apollo.meta configuration: http://192.168.66.133:8082,http://1
MetaDomainConsts : Located meta server address http://192.168.66.133:8082,http://192.168.66.133:8083 for env
Configuration : No active profile set, falling back to default profiles: default
Scope.GenericScope : BeanFactory id=7e15b194-0de7-390c-8867-1a8cc6afc4ff
Start TomcatWebServer : Tomcat initialized with port(s): 8081 (http)

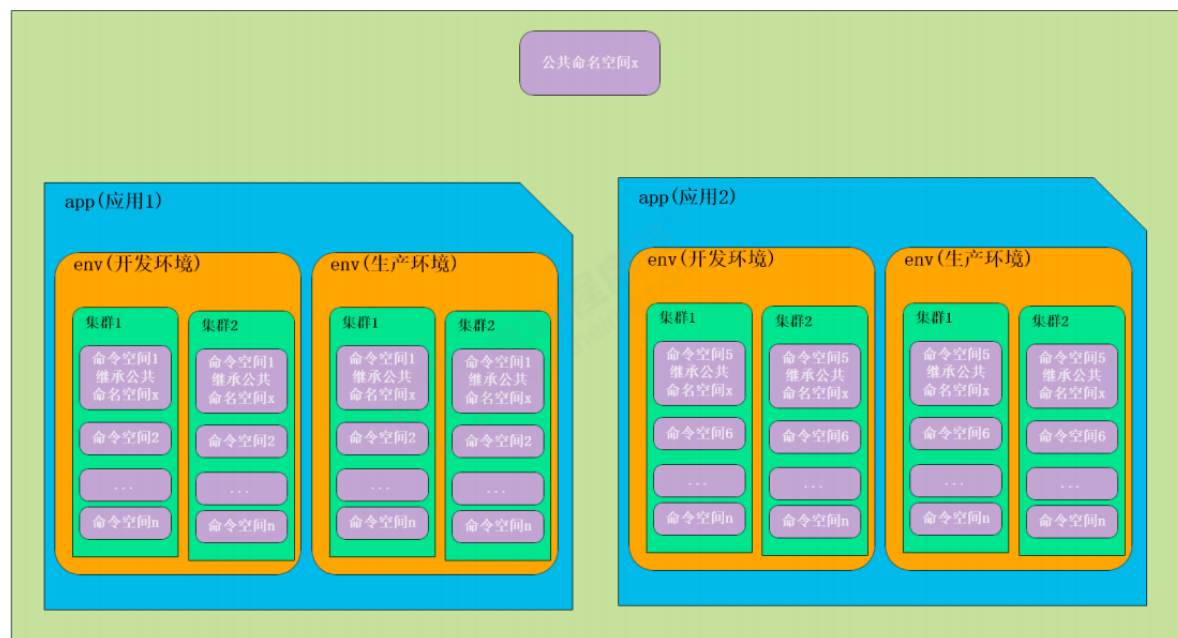
```

如果微服务能正常启动，代表PRO环境是OK的！

同一个环境下配置不同集群

“集群”就起到一个为应用按部署集群（机房）不同进行配置管理分类的作用。某些大型企业，为了满足异地容灾的需要，通常将应用部署在不同的机房。项目A在B机房有一套配置（IP等环境不同），在C机房有一套配置。为了能区分配置的不同，Apollo衍生出了集群的概念。

所以，Apollo配置中心要为项目增加集群，一定是项目在不同集群(机房)部署的时候配置不同。如果应用在不同的集群（机房）可以使用相同的配置，就没必要为项目添加集群。



添加集群

应用信息

AppId: mafeng-user

应用名称: mafeng-user

部门: 码蜂窝团队(mafeng)

负责人: apollo(apollo)

邮箱: apollo@acme.com

管理应用

管理密钥

添加集群

添加 Namespace

已发布	server.p
已发布	spring.a
已发布	spring.d ss-name
已发布	spring.d
已发布	spring.d e
已发布	spring.d
已发布	mybatis map-un case
已发布	mybatis ackage
已发布	logging
已发布	mafeng

指定集群信息

* AppId

mafeng-user

* 集群名称

SH-DATA1

(部署集群如: SHAJQ,SHAOY 或自定义集群如: SHAJQ-xx,SHAJQ-yy)

* 选择环境

☐ DEV

☒ PRO

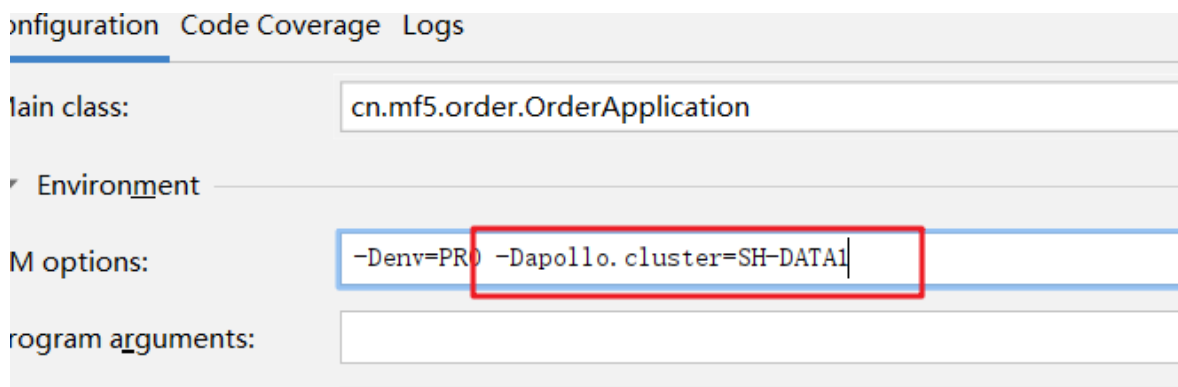
提交

集群创建好了



和之前一样，可以使用同步配置，把配置通过到新集群中。

最后修改微服务项目，切换到新集群



Apollo的灰度发布

什么是灰度发布

灰度发布是指在黑与白之间，能够平滑过渡的一种发布方式。AB test就是一种灰度发布方式，让一部分用户继续用A，一部分用户开始用B，如果用户对B没有什么反对意见，那么逐步扩大范围，把所有用户都迁移到B上面来。灰度发布可以保证整体系统的稳定，在初始灰度的时候就可以发现、调整问题。

Apollo与灰度发布

- 对于一些对程序有较大影响的配置，可以先在一个或者多个实例生效，观察一段时间没问题后再全量发布配置。
- 对于一些需要调优的配置参数，可以通过灰度发布功能来实现A/B测试。可以在不同的机器上应用不同的配置，不断调整、测评一段时间后找出较优的配置再全量发布配置。

启动两个mafeng-user服务

分别在本地和Linux服务上运行mafeng-user项目（mafeng-user项目需要打成jar包发布到Linux上）

假设两个实例的IP为：

192.168.66.133 (Linux)

192.168.79.1 (本地)

对mafeng-user打包需要导入Spring Boot打包插件

```
<build>
  <plugins>
    <!-- 用于对SpringBoot项目打包 -->
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

创建灰度配置

[版本/灰度]				
<div>发布 回滚 发布历史 授权 灰度 设置</div>				
<div>过滤配置 同步配置 撤销配置 比较配置 + 新增配置</div>				
	备注	按 Key 过滤配置 最后修改人 ↑↓	最后修改时间 ↑↓	操作
		apollo(apollo)	2022-10-26 23:46:39	 

新增灰度配置

<div>+ 新增灰度配置</div>	
最后修改时间 ↑↓	操作

添加好的灰度配置

主版本灰度版本

application有修改1

灰度发布全量发布放弃灰度

配置灰度规则灰度实例列表0更改历史

灰度的配置+新增灰度配置

发布状态	Key	主版本的值	灰度的值	备注	最后修改人	最后修改时间	操作
未发布	mafeng.password	333333	333333		apollo(apollo)	2022-10-27 21:57:14	

新增灰度规则

配置灰度规则灰度实例列表0更改历史

灰度的 AppId

新增规则

选择灰度配置生效的机器（没有选择的机器灰度配置不会生效）

编辑灰度规则

灰度的 IP

从实例列表中选择

192.168.79.1

192.168.66.133

灰度的标签

输入标签列表，英文逗号隔开，输入完后点击添加按钮

添加

取消确定

注意：记住灰度配置要发布才生效！！！！

灰度发布全量发布放弃灰度

点击灰度发布，配置生效

灰度的规则列表操作

测试如果灰度配置没有问题，可以点击 全量发布，如果配置有问题，点击 放弃灰度

介绍Apollo高可用部署架构方式

<https://www.apolloconfig.com/#/zh/deployment/deployment-architecture>

6、分布式配置管理（三）：Nacos

Nacos配置中心介绍

Nacos作为配置管理中心，实现的核心功能就是配置的统一管理。Nacos配置中心提供系统配置的集中管理（编辑、存储、分发）、动态更新不重启、回滚配置（变更管理、历史版本管理、变更审计）等所有与配置相关的活动。

抽取微服务配置到Nacos

在mafeng-user和mafeng-order进行以下操作：

导入nacos-config依赖

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

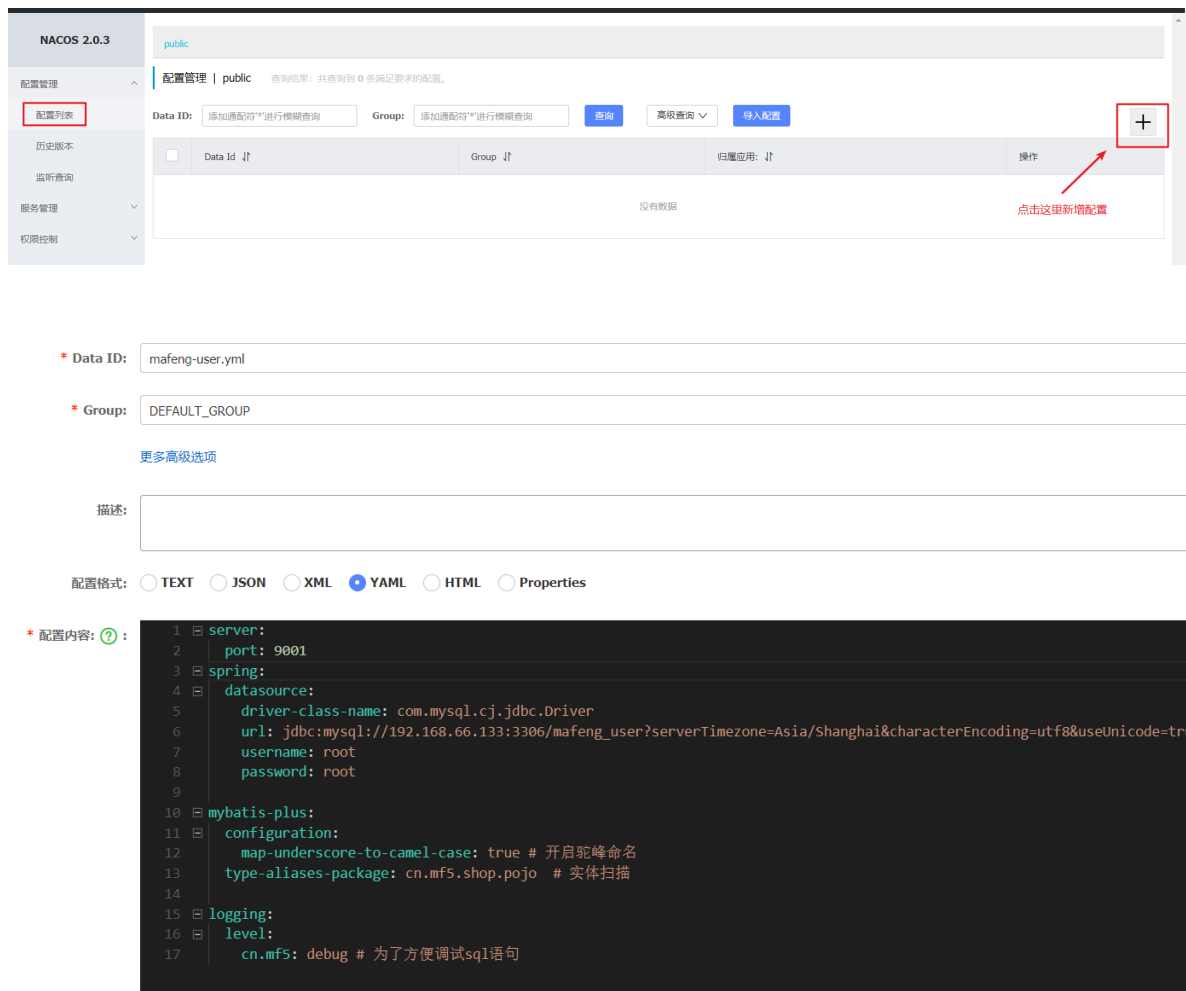
创建bootstrap.yml

在项目resources目录下建立bootstrap.yml，保留以下内容

```
spring:
  application:
    name: mafeng-user
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.66.133:8848
      config:
        server-addr: ${spring.cloud.nacos.discovery.server-addr}
        file-extension: yaml
```

注意：上面这些属性必须配置在 bootstrap.yml 或properties文件中，而不是application.yml中，config配置内容才能被正确加载。因为 bootstrap.yml 加载优先级高于 application.yml，保证在应用一起动时就去加载配置，对于Spring 中一些自动装载类来说这很重要。

Nacos发布服务配置



- Data Id: 该配置文件在Nacos系统内的唯一标识, 在 Nacos Spring Cloud 中, dataId的完整格式如:

```
${prefix}-${spring.profile.active}.${file-extension}
```

- `prefix` 默认为 `spring.application.name` 的值, 也可以通过配置项 `spring.cloud.nacos.config.prefix` 来自定义配置。
- `spring.profile.active` 即为当前环境对应的 profile, 如: `mafeng-user-dev.yaml` 中的 `dev` 就是指开发环境。**注意: 当 `spring.profile.active` 为空时, 对应的环境定义字符将不存在, 如: `mafeng-user.yaml`**
- `file-extension` 为配置内容的数据格式, 可以通过配置项 `spring.cloud.nacos.config.file-extension` 来配置。注意我们使用的是 `yaml` 类型, 不是 `yml`。虽然二者是一个意思, 但是 `nacos` 只认 `yaml`。
- Group: 同 `spring.cloud.nacos.config.group` 配置, 界面填写的内容与项目中的配置二者一定要统一, 否则无法正确读取配置, Group起到配置“隔离”的作用。

mafeng-user.yaml

```
server:
  port: 9002

spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
```

```

url: jdbc:mysql://192.168.66.133:3306/mafeng_order?
serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=true
username: root
password: root

mybatis-plus:
  configuration:
    map-underscore-to-camel-case: true # 开启驼峰命名
    type-aliases-package: cn.mf5.shop.pojo # 实体扫描

logging:
  level:
    cn.mf5: debug # 为了方便调试sql语句

mafeng-user:
  ribbon:
    NFLoadBalancerRuleClassName: com.alibaba.cloud.nacos.ribbon.NacosRule

ribbon:
  eager-load:
    enabled: true # 开启饥饿加载
    clients:
      - mafeng-user # clients是一个List集合。如果要配置多个服务名字，则换一行。用-做为前缀，每行写一个

```

mafeng-order.yaml

```

server:
  port: 9002

spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://192.168.66.133:3306/mafeng_order?
serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=true
    username: root
    password: root

mybatis-plus:
  configuration:
    map-underscore-to-camel-case: true # 开启驼峰命名
    type-aliases-package: cn.mf5.shop.pojo # 实体扫描

logging:
  level:
    cn.mf5: debug # 为了方便调试sql语句

mafeng-user:
  ribbon:
    NFLoadBalancerRuleClassName: com.alibaba.cloud.nacos.ribbon.NacosRule

ribbon:
  eager-load:
    enabled: true # 开启饥饿加载
    clients:

```

- mafeng-user # clients是一个List集合。如果要配置多个服务名字，则换一行。用-做为前缀，每行写一个

Nacos配置实时刷新

配置管理平台	Value	ConfigurationProperties
Nacos	需要结合RefreshScope才能生效	需要结合RefreshScope才能生效
Apollo	不需要结合RefreshScope就能生效（例外）	需要结合RefreshScope才能生效
Spring cloud Config	需要结合RefreshScope才能生效	需要结合RefreshScope才能生效

Nacos抽取公共配置

我们在多个应用之间往往存在一些配置信息是公共的，例如 DataSource数据源配置、服务注册配置、MQ中间件配置、Redis配置等等。我们可以将这公共的配置抽取出来，被多个应用共享。

抽取数据源配置

单独创建 common-datasource.yml，把数据源信息加进去

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://192.168.66.133:3306/${app.database.name}?
serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=true
    username: root
    password: root
```

注意：这里定义 app.database.name 变量，由具体应用配置传入。

微服务应用配置

Nacos上面的mafeng-user.yml:

```
server:
  port: 9001

mybatis-plus:
  configuration:
    map-underscore-to-camel-case: true # 开启驼峰命名
    type-aliases-package: cn.mf5.shop.pojo # 实体扫描
```

```
logging:
  level:
    cn.mf5: debug # 为了方便调试sql语句

mafeng:
  password: 666666

app:
  database:
    name: mafeng_user # 应用数据库名
```

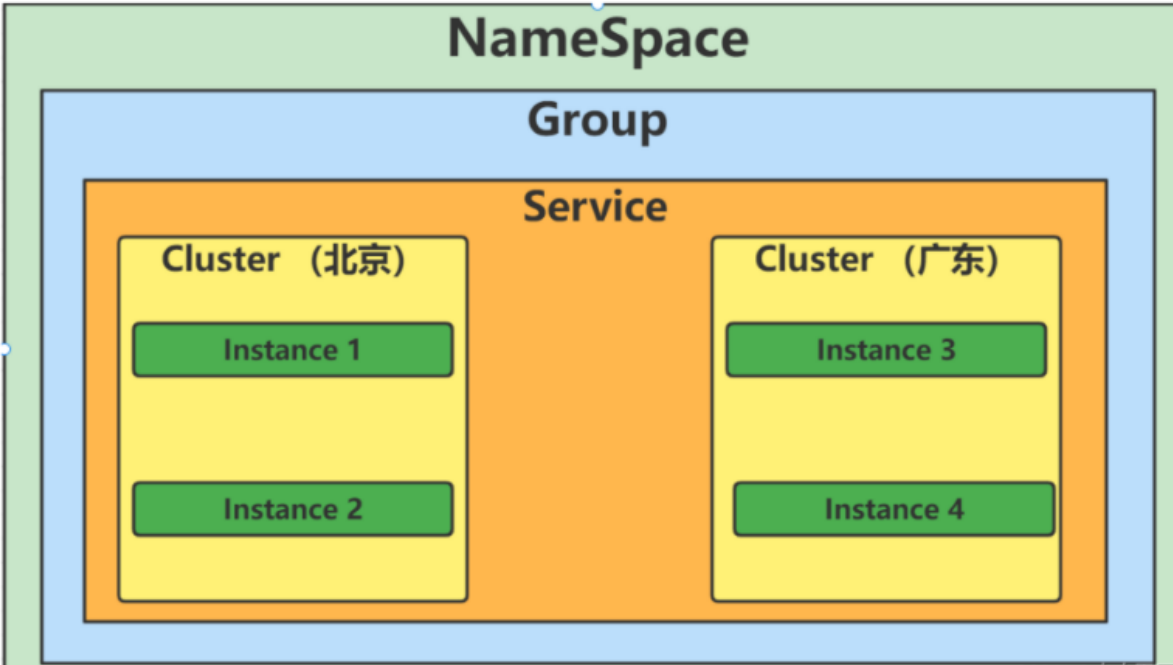
应用的bootstrap.yml配置:

```
spring:
  application:
    name: mafeng-user

cloud:
  nacos:
    discovery:
      server-addr: 192.168.66.133:7801
    config:
      server-addr: ${spring.cloud.nacos.discovery.server-addr}
      extension-configs:
        - data-id: mafeng-user.yml
          refresh: true
        - data-id: common-datasource.yml
          group: BASE_GROUP
          refresh: true
```

- 通过 `spring.cloud.nacos.config.extension-configs[n].data-id` 的配置方式来支持多个 Data Id 的配置。
- 通过 `spring.cloud.nacos.config.extension-configs[n].group` 的配置方式自定义 Data Id 所在的组, 不明确配置的话, 默认是 `DEFAULT_GROUP`。
- 通过 `spring.cloud.nacos.config.extension-configs[n].refresh` 的配置方式来控制该 Data Id 在配置变更时, 是否支持应用中可动态刷新, 感知到最新的配置值。默认是不支持的。

Nacos配置隔离



下面为大家举一些Nacos环境隔离例子：

隔离级别	举例一	举例二
namespace（一级）	一套部署环境一个namespace，如：DEV开发环境；PRO生产环境	一个公司的一个部门建立一个namespace。如：开发一部，开发二部
group（二级）	一个微服务综合项目一个组	一个微服务综合项目一个组
dataid/service（三级）	微服务配置文件xxxx.yaml、yyyy.yaml	微服务配置文件xxxx-dev.yaml、xxxx-pro.yaml来区分部署环境

注意：group的分组模式：通常是一个微服务综合项目作为一个group，因为微服务模块之间需要互相调用，放在不同的组会产生隔离，无法彼此远程调用。

Nacos配置的灰度发布（Beta）

之前在Apollo章节中，我们已经解释了什么叫灰度发布，Nacos作为配置中心也支持灰度发布功能（Beta发布）。

<input type="checkbox"/>	Data Id ⚡	Group ⚡	归属应用: ⚡	操作
<input type="checkbox"/>	mafeng-user.yml	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	common-datasource.yml	BASE_GROUP		详情 示例代码 编辑 删除 更多

[删除](#) [克隆](#) [导出](#) [▼](#)

每页显示: 10 [▼](#) [< 上一页](#)

勾选Beta发布，输入需要灰度发布的实例IP地址

* Data ID: mafeng-user.yml

* Group: DEFAULT_GROUP

[更多高级选项](#)

描述:

Beta发布: ☒ 默认不要勾选。 192.168.66.133 × ▾ 自行输入希望灰度发布的IP

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

最后，点击Beta发布

编辑配置

正式

BETA

[查看正式版和Beta版](#)

* Data ID: mafeng-user.yml

* Group: DEFAULT_GROUP

[更多高级选项](#)

描述:

Beta发布: 192.168.66.133 × ▾

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

配置内容[?]:

```
1 server:
2   port: 9001
```