Apollo分布式配置中心部署及使用

# Apollo配置中心介绍

## 1.1 什么是Apollo

### 1.1.1 背景

随着程序功能的日益复杂，程序的配置日益增多：各种功能的开关、参数的配置、服务器的地址等.

对程序配置的期望值也越来越高：配置修改后实时生效，灰度发布，分环境、分集群管理配置，完善的权限、审核机制……

在这样的大环境下，传统的通过配置文件、数据库等方式已经越来越无法满足开发人员对配置管理的需求。

因此Apollo配置中心应运而生.

### 1.1.2 Apollo简介

Apollo（阿波罗）是携程框架部门研发的开源配置管理中心，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. Apollo支持4个维度管理Key-Value格式的配置：

1. application (应用)

2. environment (环境)

3. cluster (集群)

4. namespace (命名空间)

* application (应用)

指实际使用配置的应用，Apollo客户端在运行时需要知道当前应用是谁，从而可以去获取对应的配置

每个应用都需要有唯一的身份标识 -- appId，我们认为应用身份是跟着代码走的，所以需要在代码中配置。

* environment (环境)

配置对应的环境，Apollo客户端在运行时需要知道当前应用处于哪个环境，从而可以去获取应用的配置.

我们认为环境和代码无关，同一份代码部署在不同的环境就应该能够获取到不同环境的配置.

所以环境默认是通过读取机器上的配置（server.properties中的env属性）指定的，不过为了开发方便，我们也支持运行时通过System Property等指定。

* cluster (集群)

一个环境下不同实例的分组，比如典型的可以按照数据中心分，把上海机房的应用实例分为一个集群，把北京机房的应用实例分为另一个集群。

对不同的cluster，同一个配置可以有不一样的值，如zookeeper地址。

集群默认是通过读取机器上的配置（server.properties中的idc属性）指定的，不过也支持运行时通过System Property指定。

* namespace (命名空间)

一个应用下不同配置的分组，可以简单地把namespace类比为文件，不同类型的配置存放在不同的文件中，如数据库配置文件，RPC配置文件，应用自身的配置文件等

应用可以直接读取到公共组件的配置namespace，如DAL，RPC等.

应用也可以通过继承公共组件的配置namespace来对公共组件的配置做调整，如DAL的初始数据库连接数.

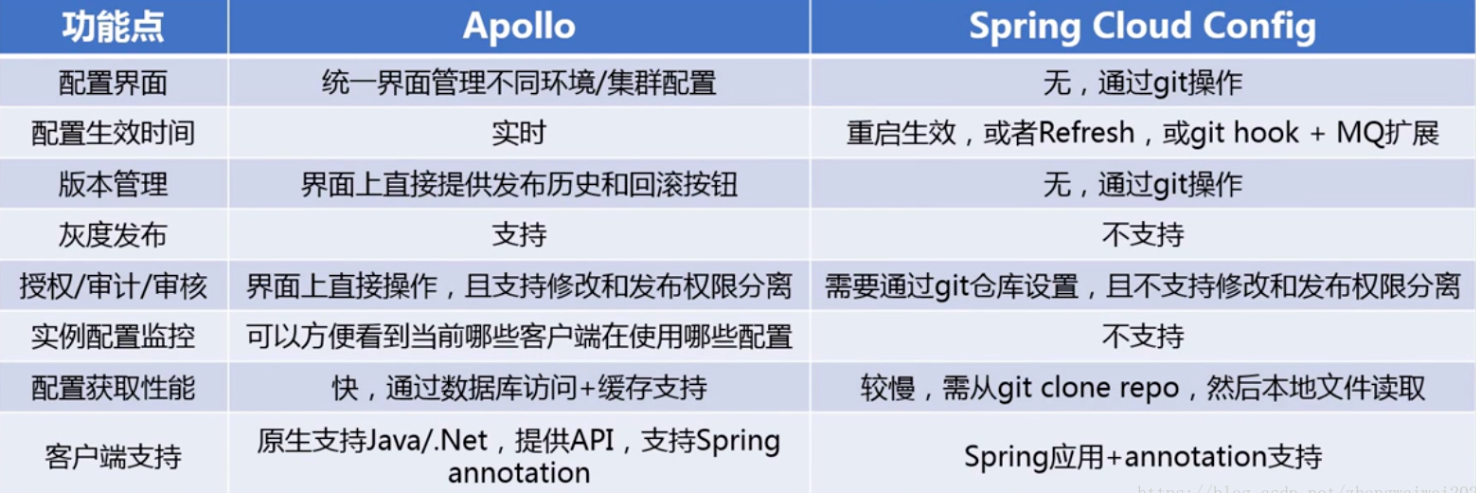
## 1.2 为什么选择Apollo

### 1.2.1 Apollo特性

Apollo从设计之初就立志于成为一个有治理能力的配置发布平台，目前提供了以下的特性：

* 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zookeeper的地址等
  + 通过命名空间（namespace）可以很方便地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，从而可以方便地支持配置的回滚
* 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，可以方便地追踪问题
* 客户端配置信息监控
  + 可以在界面上方便地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应用也可以方便地使用
* 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，不会对配置的修改做过多限制，只要符合基本的格式就能保存，不会针对不同的配置值进行针对性的校验，如数据库用户名、密码，Redis服务地址等
  + 对于这类应用配置，Apollo支持应用方通过开放平台API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

### 1.2.2 与Spring Cloud Config对比

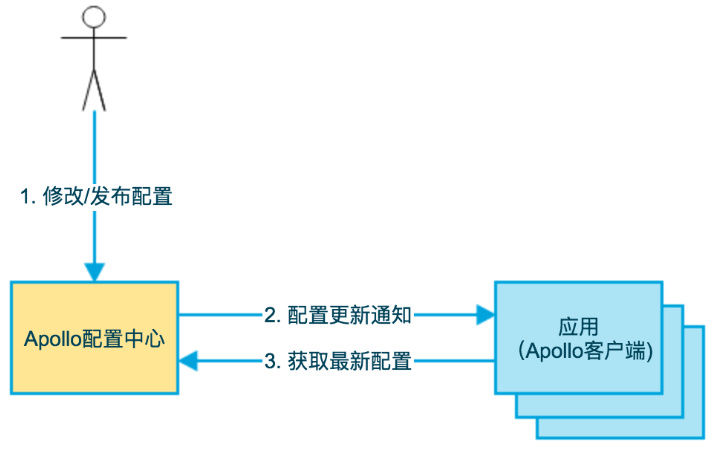


## 1.3 Apollo初体验

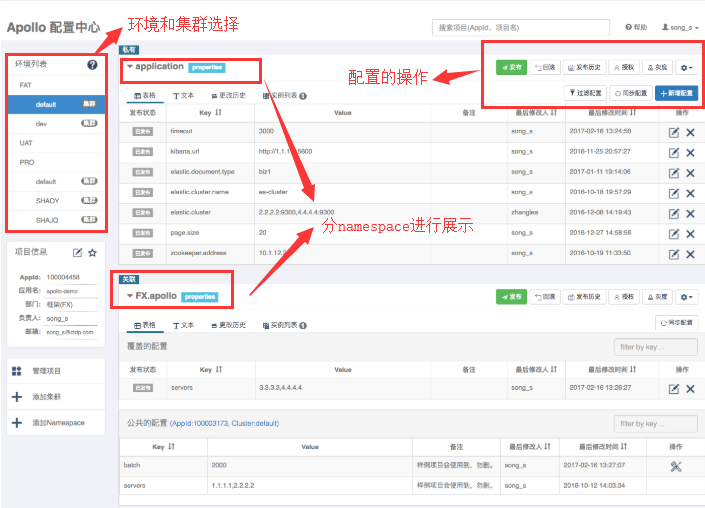
### 1.3.1 基础模型

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



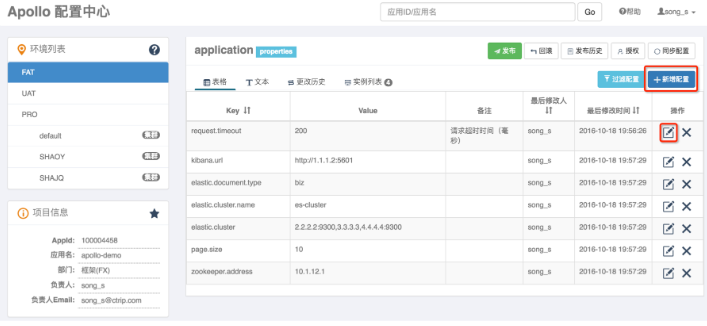
### 1.3.2 界面概览



* 在页面左上方的环境列表模块展示了所有的环境和集群，用户可以随时切换。
* 页面中央展示了两个namespace(application和FX.apollo)的配置信息，默认按照表格模式展示、编辑。用户也可以切换到文本模式，以文件形式查看、编辑。
* 页面上可以方便地进行发布、回滚、灰度、授权、查看更改历史和发布历史等操作

### 1.3.3 添加/修改配置项

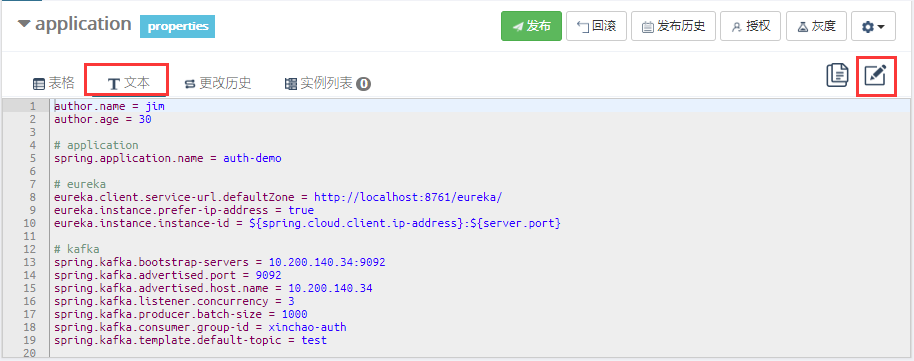
用户可以通过配置中心界面方便的添加/修改配置项



输入配置信息

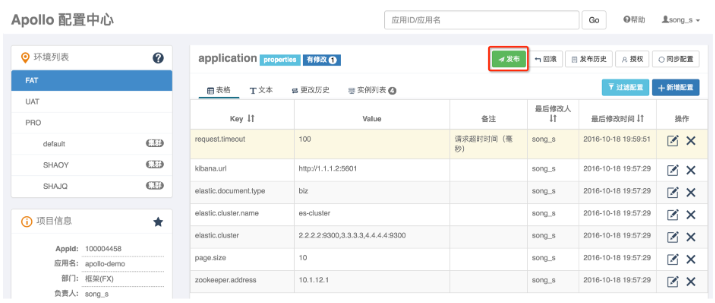


也可以通过文本的方式进行批量修改.



### 1.3.4 发布配置

修改后,配置并未生效.需要通过发布按钮,对修改的配置进行发布.



填写发布信息



# 2 Apollo服务端分布式部署

## 2.1 准备工作

### 2.1.1 JDK

要求JDK版本为1.8.

### 2.1.2 Mysql

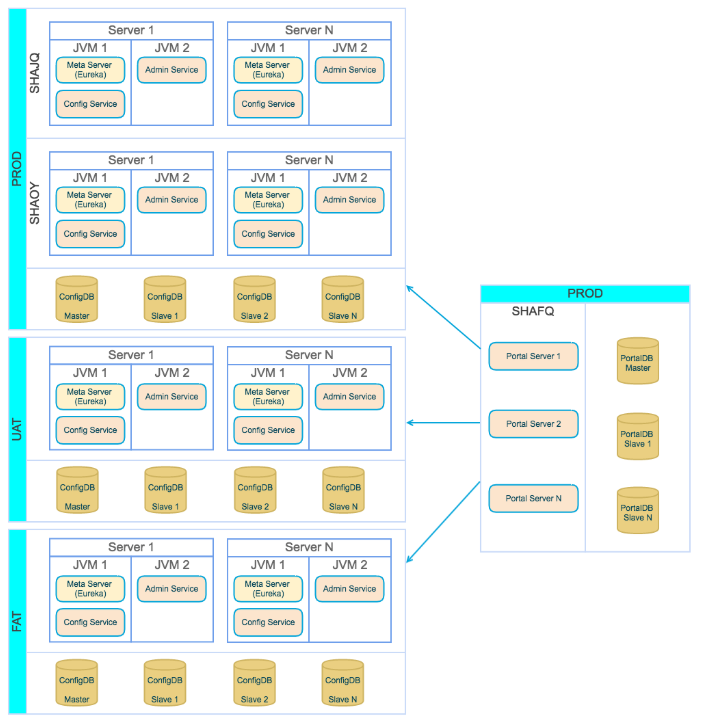
要求Mysql版本 5.6.5 +.

### 2.1.3 环境

分布式部署需要事先确定部署的环境以及部署方式。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

以携程为例,部署策略如下:



* Portal部署在生产环境的机房，通过它来直接管理FAT、UAT、PRO等环境的配置
* Meta Server、Config Service和Admin Service在每个环境都单独部署，使用独立的数据库
* Meta Server、Config Service和Admin Service在生产环境部署在两个机房，实现双活
* Meta Server和Config Service部署在同一个JVM进程内，Admin Service部署在同一台服务器的另一个JVM进程内

### 2.1.4 网络策略

#### 2.1.4.1 网络限制

分布式部署的时候，apollo-configservice和apollo-adminservice需要把自己的IP和端口注册到Meta Server（apollo-configservice本身）。

Apollo客户端和Portal会从Meta Server获取服务的地址（IP+端口），然后通过服务地址直接访问。

所以如果实际部署的机器有多块网卡（如docker），或者存在某些网卡的IP是Apollo客户端和Portal无法访问的（如网络安全限制），那么我们就需要在apollo-configservice和apollo-adminservice中做相关限制以避免Eureka将这些网卡的IP注册到Meta Server。

#### 2.1.4.2修改方式

##### 忽略特定网卡

如apollo-configservice,把docker0和veth.\*的网卡注册到Eureka时忽略掉.修改apollo-configservice的application.yml文件.

spring:

application:

name: apollo-configservice

profiles:

active: ${apollo\_profile}

cloud:

inetutils:

ignoredInterfaces:

- docker0

- veth.\*

##### 指定Ip

可以修改startup.sh，通过JVM System Property在运行时传入，如-Deureka.instance.ip-address=${指定的IP}，或者也可以修改apollo-adminservice或apollo-configservice 的bootstrap.yml文件，加入以下配置

eureka:

instance:

ip-address: ${指定的IP}

##### 指定IP+端口

可以修改startup.sh，通过JVM System Property在运行时传入，如-Deureka.instance.homePageUrl=http://${指定的IP}:${指定的Port}，或者也可以修改apollo-adminservice或apollo-configservice 的bootstrap.yml文件，加入以下配置

eureka:

instance:

homePageUrl: http://${指定的IP}:${指定的Port}

preferIpAddress: false

做完上述修改并重启后，可以查看Eureka页面（http://${config-service-url:port}）检查注册上来的IP信息是否正确。

如果Apollo部署在公有云上，本地开发环境无法连接，但又需要做开发测试的话，客户端可以升级到0.11.0版本及以上，然后通过-Dapollo.configService=http://config-service的公网IP:端口来跳过meta service的服务发现.

### 2.1.5 Docker部署

Apollo项目已经自带了Docker file，可以在获取安装包并配置好安装包后通过下面的文件来打Docker镜像：

1. [apollo-configservice](https://github.com/ctripcorp/apollo/blob/master/apollo-configservice/src/main/docker/Dockerfile)
2. [apollo-adminservice](https://github.com/ctripcorp/apollo/blob/master/apollo-adminservice/src/main/docker/Dockerfile)
3. [apollo-portal](https://github.com/ctripcorp/apollo/blob/master/apollo-portal/src/main/docker/Dockerfile)

### 2.1.6 K8S支持

使用说明参考[apollo-on-kubernetes](https://github.com/ctripcorp/apollo/blob/master/scripts/apollo-on-kubernetes/README.md)。

## 2.2 部署步骤

部署步骤共三步：

1. 创建数据库

Apollo服务端依赖于MySQL数据库，所以需要事先创建并完成初始化

1. 获取安装包

Apollo服务端安装包共有3个：apollo-configservice, apollo-adminservice, apollo-portal

可以直接下载我们事先打好的安装包，也可以自己通过源码构建

1. 部署Apollo服务端

获取安装包后就可以部署到公司的测试和生产环境了

### 2.2.1 创建数据库

Apollo服务端共需要两个数据库：ApolloPortalDB和ApolloConfigDB，我们把数据库、表的创建和样例数据都分别准备了sql文件，只需要导入数据库即可。

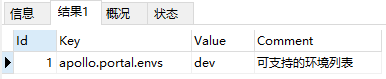
需要注意的是ApolloPortalDB只需要在生产环境部署一个即可，而ApolloConfigDB需要在每个环境部署一套，如fat、uat和pro分别部署3套ApolloConfigDB。

#### 2.2.1.1 创建ApolloPortalDB

导入apolloportaldb.sql文件即可。导入成功后，可以通过执行以下sql语句来验证：

**select** `Id`, `Key`, `Value`, `Comment` **from** `ApolloPortalDB`.`ServerConfig` limit 1;

结果如下：



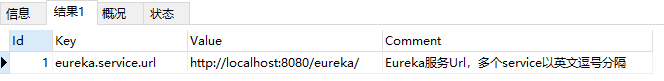
* 注意：ApolloPortalDB只需要在生产环境部署一个即可.

#### 2.2.1.2 创建ApolloConfigDB

导入apolloconfigdb.sql文件即可。导入成功后，可以通过执行以下sql语句来验证：

**select** `Id`, `Key`, `Value`, `Comment` **from** `ApolloConfigDB`.`ServerConfig` limit 1;

结果如下：



* 注意：ApolloConfigDB需要在每个环境部署一套，如fat、uat和pro分别部署3套ApolloConfigDB

#### 2.2.1.3 调整服务端配置

Apollo自身的一些配置是放在数据库里面的，所以需要针对实际情况做一些调整。

##### 2.2.1.3.1 调整ApolloPortalDB配置

配置项统一存储在ApolloPortalDB.ServerConfig表中，也可以通过界面中**管理员工具 - 系统参数**页面进行配置。



###### apollo.portal.envs - 可支持的环境列表

默认值是dev，如果portal需要管理多个环境，以逗号分隔即可（大小写不敏感），如：

DEV,FAT,UAT,PRO

注意：

* 1. 一套Portal可以管理多个环境，但是每个环境都需要独立部署一套Config Service、Admin Service和ApolloConfigDB
  2. 只在数据库添加环境是不起作用的，还需要为apollo-portal添加新增环境对应的meta server地址。后面详解

###### organizations - 部门列表

Portal中新建的App都需要选择部门，所以需要在这里配置可选的部门信息，样例如下：

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

###### superAdmin - Portal超级管理员

超级管理员拥有所有权限，需要谨慎设置。

如果没有接入自己公司的SSO系统的话，可以先暂时使用默认值apollo（默认用户）。等接入后，修改为实际使用的账号，多个账号以英文逗号分隔(,)。

###### consumer.token.salt – 自定义token的盐值

如果会使用开放平台API的话，可以设置一个token salt。如果不使用，可以忽略。

###### wiki.address – 帮助链接地址

portal上“帮助”链接的地址，默认是Apollo github的wiki首页，可自行设置。

###### admin.createPrivateNamespace.switch

是否允许项目管理员创建private namespace。设置为true允许创建，设置为false则项目管理员在页面上看不到创建private namespace的选项。

###### configView.memberOnly.envs

只对项目成员显示配置信息的环境列表，多个env以英文逗号分隔。

对设定了只对项目成员显示配置信息的环境，只有该项目的管理员或拥有该namespace的编辑或发布权限的用户才能看到该私有namespace的配置信息和发布历史。公共namespace始终对所有用户可见。

##### 2.2.1.3.2 调整ApolloConfigDB配置

配置项统一存储在ApolloConfigDB.ServerConfig表中，需要注意每个环境的ApolloConfigDB.ServerConfig都需要单独配置。

###### eureka.service.url - Eureka服务Url

不管是apollo-configservice还是apollo-adminservice都需要向eureka服务注册，所以需要配置eureka服务地址。 按照目前的实现，apollo-configservice本身就是一个eureka服务，所以只需要填入apollo-configservice的地址即可，如有多个，用逗号分隔（注意不要忘了/eureka/后缀）。

需要注意的是每个环境只填入自己环境的eureka服务地址，比如FAT的apollo-configservice是1.1.1.1:8080和2.2.2.2:8080，UAT的apollo-configservice是3.3.3.3:8080和4.4.4.4:8080，PRO的apollo-configservice是5.5.5.5:8080和6.6.6.6:8080，那么：

* 在FAT环境的ApolloConfigDB.ServerConfig表中设置eureka.service.url为：

http://1.1.1.1:8080/eureka/,http://2.2.2.2:8080/eureka/

* 在UAT环境的ApolloConfigDB.ServerConfig表中设置eureka.service.url为：

http://3.3.3.3:8080/eureka/,http://4.4.4.4:8080/eureka/

* 在PRO环境的ApolloConfigDB.ServerConfig表中设置eureka.service.url为：

http://5.5.5.5:8080/eureka/,http://6.6.6.6:8080/eureka/

###### namespace.lock.switch

一次发布只能有一个人修改开关，用于发布审核。这是一个功能开关，如果配置为true的话，那么一次配置发布只能是一个人修改，另一个发布。

###### config-service.cache.enabled - 是否开启配置缓存

这是一个功能开关，如果配置为true的话，config service会缓存加载过的配置信息，从而加快后续配置获取性能。默认为false，开启前请先评估总配置大小并调整config service内存配置。

###### item.key.length.limit - 配置项 key 最大长度限制

默认配置长度128.

###### item.value.length.limit - 配置项 value 最大长度限制

默认配置长度20000.

### 2.2.2 获取安装包

可以通过两种方式获取安装包：

1. 直接下载安装包
   * 从[GitHub Release](https://github.com/ctripcorp/apollo/releases)页面下载预先打好的安装包
   * 如果对Apollo的代码没有定制需求，建议使用这种方式，可以省去本地打包的过程
2. 通过源码构建
   * 从[GitHub Release](https://github.com/ctripcorp/apollo/releases)页面下载Source code包或直接clone[源码](https://github.com/ctripcorp/apollo)后在本地构建
   * 如果需要对Apollo的做定制开发，需要使用这种方式

#### 2.2.2.1 直接下载安装包

##### 2.2.2.1.1 获取configservice、adminservice、portal安装包

从[GitHub Release](https://github.com/ctripcorp/apollo/releases)页面下载最新版本的apollo-configservice-x.x.x-github.zip、apollo-adminservice-x.x.x-github.zip和apollo-portal-x.x.x-github.zip即可。

##### 2.2.2.1.2 配置数据库连接信息

Apollo服务端需要知道如何连接到你前面创建的数据库，数据库连接串信息位于上一步下载的压缩包中的config/application-github.properties中。

* apollo-configservice数据库连接配置

1. 解压apollo-configservice-x.x.x-github.zip
2. 打开config目录下的application-github.properties文件
3. 填写正确的ApolloConfigDB数据库连接串信息，注意用户名和密码后面不要有空格。
4. 修改完的效果如下：

# DataSource

spring.datasource.url = jdbc:mysql://localhost:3306/ApolloConfigDB?characterEncoding=utf8

spring.datasource.username = username

spring.datasource.password = password

注意：由于ApolloConfigDB在每个环境都有部署，所以对不同的环境config-service需要配置对应环境的数据库参数

* apollo-adminservice数据库连接配置

1. 解压apollo-adminservice-x.x.x-github.zip
2. 打开config目录下的application-github.properties文件
3. 填写正确的ApolloConfigDB数据库连接串信息，注意用户名和密码后面不要有空格。
4. 修改完的效果如下：

# DataSource

spring.datasource.url = jdbc:mysql://localhost:3306/ApolloConfigDB?characterEncoding=utf8

spring.datasource.username = username

spring.datasource.password = password

注意：由于ApolloConfigDB在每个环境都有部署，所以对不同的环境admin-service需要配置对应环境的数据库参数

* apollo-portal数据库连接配置

1. 解压apollo-portal-x.x.x-github.zip
2. 打开config目录下的application-github.properties文件
3. 填写正确的ApolloPortalDB数据库连接串信息，注意用户名和密码后面不要有空格。
4. 修改完的效果如下：

# DataSource

spring.datasource.url = jdbc:mysql://localhost:3306/ApolloPortalDB?characterEncoding=utf8

spring.datasource.username = username

spring.datasource.password = password

* 配置apollo-portal的meta service信息

Apollo Portal需要在不同的环境访问不同的meta service(apollo-configservice)地址，所以我们需要在配置中提供这些信息。默认情况下，meta service和config service是部署在同一个JVM进程，所以meta service的地址就是config service的地址。

打开apollo-portal-x.x.x-github.zip中config目录下的apollo-env.properties文件。

假设DEV的apollo-configservice未绑定域名，地址是1.1.1.1:8080，FAT的apollo-configservice绑定了域名apollo.fat.xxx.com，UAT的apollo-configservice绑定了域名apollo.uat.xxx.com，PRO的apollo-configservice绑定了域名apollo.xxx.com，那么可以如下修改各环境meta service服务地址，格式为${env}.meta=http://${config-service-url:port}，如果某个环境不需要，也可以直接删除对应的配置项（如lpt.meta）：

dev.meta=http://1.1.1.1:8080

fat.meta=http://apollo.fat.xxx.com

uat.meta=http://apollo.uat.xxx.com

pro.meta=http://apollo.xxx.com

备注：

1. 为了实现meta service的高可用，推荐通过SLB（Software Load Balancer）做动态负载均衡
2. meta service地址也可以填入IP，0.11.0版本之前只支持填入一个IP。从0.11.0版本开始支持填入以逗号分隔的多个地址，如http://1.1.1.1:8080,http://2.2.2.2:8080，不过生产环境还是建议使用域名（走slb），因为机器扩容、缩容等都可能导致IP列表的变化。

#### 2.2.2.2 通过源码构建

从[GitHub Release](https://github.com/ctripcorp/apollo/releases)页面下载Source code包或直接clone[源码](https://github.com/ctripcorp/apollo)后在本地构建。

##### 2.2.2.2.1 配置数据库连接信息

Apollo服务端需要知道如何连接到你前面创建的数据库，所以需要编辑[scripts/build.sh](https://github.com/ctripcorp/apollo/blob/master/scripts/build.sh)，修改ApolloPortalDB和ApolloConfigDB相关的数据库连接串信息。

#apollo config db info

apollo\_config\_db\_url=jdbc:mysql://localhost:3306/ApolloConfigDB?characterEncoding=utf8

apollo\_config\_db\_username=用户名

apollo\_config\_db\_password=密码（如果没有密码，留空即可）

# apollo portal db info

apollo\_portal\_db\_url=jdbc:mysql://localhost:3306/ApolloPortalDB?characterEncoding=utf8

apollo\_portal\_db\_username=用户名

apollo\_portal\_db\_password=密码（如果没有密码，留空即可）

备注：

1. 由于ApolloConfigDB在每个环境都有部署，所以对不同的环境config-service和admin-service需要使用不同的数据库参数打不同的包，portal只需要打一次包。
2. 每个环境都需要独立部署一套config-service、admin-service和ApolloConfigDB

##### 2.2.2.2.2 配置各环境meta service地址

Apollo Portal需要在不同的环境访问不同的meta service(apollo-configservice)地址，所以需要在打包时提供这些信息。

假设DEV的apollo-configservice未绑定域名，地址是1.1.1.1:8080，FAT的apollo-configservice绑定了域名apollo.fat.xxx.com，UAT的apollo-configservice绑定了域名apollo.uat.xxx.com，PRO的apollo-configservice绑定了域名apollo.xxx.com，那么编辑scripts/build.sh，如下修改各环境meta service服务地址，格式为${env}\_meta=http://${config-service-url:port}，如果某个环境不需要，也可以直接删除对应的配置项：

dev\_meta=http://1.1.1.1:8080

fat\_meta=http://apollo.fat.xxx.com

uat\_meta=http://apollo.uat.xxx.com

pro\_meta=http://apollo.xxx.com

META\_SERVERS\_OPTS="-Ddev\_meta=$dev\_meta -Dfat\_meta=$fat\_meta -Duat\_meta=$uat\_meta -Dpro\_meta=$pro\_meta"

备注：

1. 为了实现meta service的高可用，推荐通过SLB（Software Load Balancer）做动态负载均衡
2. meta service地址也可以填入IP，0.11.0版本之前只支持填入一个IP。从0.11.0版本开始支持填入以逗号分隔的多个地址，如http://1.1.1.1:8080,http://2.2.2.2:8080，不过生产环境还是建议使用域名（走slb），因为机器扩容、缩容等都可能导致IP列表的变化。

##### 2.2.2.2.3 编译打包

做完上述配置后，就可以执行编译和打包了。初次编译会从Maven中央仓库下载不少依赖，如果网络情况不佳时很容易出错，建议使用国内的Maven仓库源，比如[阿里云Maven镜像](http://www.cnblogs.com/geektown/p/5705405.html)，或者使用公司私服。执行：

./build.sh

该脚本会依次打包apollo-configservice, apollo-adminservice, apollo-portal。在各自的target目录下获取zip压缩包。

### 2.2.3 部署Apollo服务端

#### 2.2.3.1 部署apollo-configservice

将对应环境的apollo-configservice-x.x.x-github.zip上传到服务器上，解压后执行scripts/startup.sh即可。如需停止服务，执行scripts/shutdown.sh.

在scripts/startup.sh中按照实际的环境设置一个JVM内存，以下是我们的默认设置，供参考：

export JAVA\_OPTS="-server -Xms6144m -Xmx6144m -Xss256k -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=384m -XX:NewSize=4096m -XX:MaxNewSize=4096m -XX:SurvivorRatio=18"

参数说明：

* -Xms : 设置JVM初始内存。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。
* -Xmx：设置JVM最大可用内存。
* Xss：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成。
* -XX:MetaspaceSize：metaspace（JDK8存储类的元数据信息）初始空间大小，达到该值就会触发垃圾收集进行类型卸载，同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。
* -XX:MaxMetastapceSize：metasptace最大空间，默认是没有限制的。
* -XX:NewSize：新生代大小
* -XX:MaxNewSize：新生代最大大小
* -XX:SurvivorRatio：Eden区与Survivor区大小比值。如这里默认的18，即Eden区大小是幸存区To或幸存区From的18倍。所以Eden占新生代内存的18/20. 两个Survivor（幸存区）各占1/20.
* 特别说明

1. 如果需要修改JVM参数，可以修改scripts/startup.sh的JAVA\_OPTS部分。
2. 如要调整服务的日志输出路径，可以修改scripts/startup.sh和apollo-configservice.conf中的LOG\_DIR。
3. 如要调整服务的监听端口，可以修改scripts/startup.sh中的SERVER\_PORT。另外apollo-configservice同时承担meta server职责，如果要修改端口，注意要同时ApolloConfigDB.ServerConfig表中的eureka.service.url配置项以及apollo-portal和apollo-client中的使用到的meta server信息。
4. 如果ApolloConfigDB.ServerConfig的eureka.service.url只配了当前正在启动的机器的话，在启动apollo-configservice的过程中会在日志中输出eureka注册失败的信息，如com.sun.jersey.api.client.ClientHandlerException: java.net.ConnectException: Connection refused。需要注意的是，这个是预期的情况，因为apollo-configservice需要向Meta Server（它自己）注册服务，但是因为在启动过程中，自己还没起来，所以会报这个错。后面会进行重试的动作，所以等自己服务起来后就会注册正常了。

#### 2.2.3.2 部署apollo-adminservice

将对应环境的apollo-adminservice-x.x.x-github.zip上传到服务器上，解压后执行scripts/startup.sh即可。如需停止服务，执行scripts/shutdown.sh.

在scripts/startup.sh中按照实际的环境设置一个JVM内存，以下是我们的默认设置，供参考：

export JAVA\_OPTS="-server -Xms2560m -Xmx2560m -Xss256k -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=384m -XX:NewSize=1024m -XX:MaxNewSize=1024m -XX:SurvivorRatio=22"

* 特别说明

1. 如果需要修改JVM参数，可以修改scripts/startup.sh的JAVA\_OPTS部分。
2. 如要调整服务的日志输出路径，可以修改scripts/startup.sh和apollo-adminservice.conf中的LOG\_DIR。
3. 如要调整服务的监听端口，可以修改scripts/startup.sh中的SERVER\_PORT。

#### 2.2.3.3 部署apollo-portal

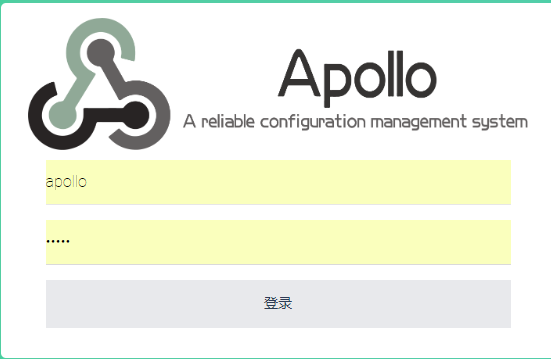
将apollo-portal-x.x.x-github.zip上传到服务器上，解压后执行scripts/startup.sh即可。如需停止服务，执行scripts/shutdown.sh.

在startup.sh中按照实际的环境设置一个JVM内存，以下是我们的默认设置，供参考：

export JAVA\_OPTS="-server -Xms4096m -Xmx4096m -Xss256k -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=384m -XX:NewSize=1536m -XX:MaxNewSize=1536m -XX:SurvivorRatio=22"

* 特别说明

1. 如果需要修改JVM参数，可以修改scripts/startup.sh的JAVA\_OPTS部分。
2. 如要调整服务的日志输出路径，可以修改scripts/startup.sh和apollo-portal.conf中的LOG\_DIR。
3. 如要调整服务的监听端口，可以修改scripts/startup.sh中的SERVER\_PORT。
4. Apollo服务端的启动请严格按照apollo-configservice -> apollo-adminservice -> apollo-portal的顺序。
5. portal服务启动后，就可以通过${portalservice.ip}:{port}访问apollo界面了。会自动跳转到/signin页面。使用默认超级管理员账号 apollo, 密码 admin进行登录。



# 3 Java客户端接入

## 3.1 准备工作

### 3.1.1 环境要求

JDK1.7以上，Guava15.0以上。（Apollo客户端默认会引用Guava19）.

### 3.1.2 必选设置

#### 3.1.2.1 AppId

AppId是应用的身份信息，是从服务端获取配置的一个重要信息。有以下3种方式设置，按照优先级从高到底分别为：

1. System Property

Apollo 0.7.0+支持通过System Property传入app.id信息，如

-Dapp.id=YOUR-APP-ID

1. Spring Boot application.properties

Apollo 1.0.0+支持通过Spring Boot的application.properties文件配置，如

app.id=YOUR-APP-ID

1. app.properties

确保classpath:/META-INF/app.properties文件存在，并且其中内容形如：

app.id=YOUR-APP-ID

#### 3.1.2.2 Apollo Meta Server

Apollo支持应用在不同的环境有不同的配置，所以需要在运行提供给Apollo客户端当前环境的Apollo Meta Server信息。默认情况下，meta server和config service是部署在同一个JVM进程，所以meta server的地址就是config service的地址。

为了实现meta server的高可用，推荐通过SLB（Software Load Balancer）做动态负载均衡。Meta server地址也可以填入IP，如http://1.1.1.1:8080,http://2.2.2.2:8080，不过生产环境还是建议使用域名（走slb），因为机器扩容、缩容等都可能导致IP列表的变化。

1.0.0版本开始支持以下方式配apollo meta server信息，按照优先级从高到底分别为：

1. 通过Java System Property apollo.meta

可以通过Java的System Property apollo.meta来指定。在Java程序启动脚本中，可以指定-Dapollo.meta=http://config-service-url。如果是运行jar文件，需要注意格式是

java -Dapollo.meta=http://config-service-url -jar xxx.jar

也可以通过程序指定，如System.setProperty("apollo.meta", "http://config-service-url");

1. 通过Spring Boot的配置文件

可以在Spring Boot的application.properties或bootstrap.properties中指定apollo.meta=http://config-service-url

1. 通过操作系统的System Environment APOLLO\_META

可以通过操作系统的System Environment APOLLO\_META来指定。注意key为全大写，且中间是\_分隔

1. 通过server.properties配置文件

可以在server.properties配置文件中指定apollo.meta=http://config-service-url，对于Mac/Linux，文件位置为/opt/settings/server.properties，对于Windows，文件位置为C:\opt\settings\server.properties

1. 通过app.properties配置文件

可以在classpath:/META-INF/app.properties指定apollo.meta=http://config-service-url

1. 通过Java system property ${env}\_meta

如果当前env是dev，那么用户可以配置-Ddev\_meta=http://config-service-url

1. 通过apollo-env.properties文件

用户也可以创建一个apollo-env.properties，放在程序的classpath下，或者放在spring boot应用的config目录下

文件内容形如：

dev.meta=http://1.1.1.1:8080

fat.meta=http://apollo.fat.xxx.com

uat.meta=http://apollo.uat.xxx.com

pro.meta=http://apollo.xxx.com

如果通过以上各种手段都无法获取到Meta Server地址，Apollo最终会fallback到http://apollo.meta作为Meta Server地址

#### 3.1.2.3 本地缓存路径

##### 3.1.2.3.1 默认路径

Apollo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，用于在遇到服务不可用，或网络不通的时候，依然能从本地恢复配置，不影响应用正常运行。

本地缓存路径默认位于以下路径，所以请确保/opt/data或C:\opt\data\目录存在，且应用有读写权限。

* Mac/Linux: /opt/data/{appId}/config-cache
* Windows: C:\opt\data\{appId}\config-cache

本地配置文件会以下面的文件名格式放置于本地缓存路径下：

{appId}+{cluster}+{namespace}.properties

##### 3.1.2.3.2 自定义路径

1.0.0版本开始支持以下方式自定义缓存路径，按照优先级从高到底分别为：

1. 通过Java System Property apollo.cacheDir

可以通过Java的System Property apollo.cacheDir来指定。在Java程序启动脚本中，可以指定-Dapollo.cacheDir=/opt/data/some-cache-dir。如果是运行jar文件，需要注意格式是：java -Dapollo.cacheDir=/opt/data/some-cache-dir -jar xxx.jar

也可以通过程序指定，如System.setProperty("apollo.cacheDir", "/opt/data/some-cache-dir");

1. 通过Spring Boot的配置文件

可以在Spring Boot的application.properties或bootstrap.properties中指定apollo.cacheDir=/opt/data/some-cache-dir

1. 通过操作系统的System EnvironmentAPOLLO\_CACHEDIR

可以通过操作系统的System Environment APOLLO\_CACHEDIR来指定。注意key为全大写，且中间是\_分隔

1. 通过server.properties配置文件

可以在server.properties配置文件中指定apollo.cacheDir=/opt/data/some-cache-dir

对于Mac/Linux，文件位置为/opt/settings/server.properties，对于Windows，文件位置为C:\opt\settings\server.properties。

### 3.1.3 可选设置

#### 3.1.3.1 Environment环境

Environment可以通过以下3种方式的任意一个配置：

1. 通过Java System Property
   * 可以通过Java的System Property env来指定环境
   * 在Java程序启动脚本中，可以指定-Denv=YOUR-ENVIRONMENT
   * 如果是运行jar文件，需要注意格式是java -Denv=YOUR-ENVIRONMENT -jar xxx.jar。注意key为全小写
2. 通过操作系统的System Environment

* 还可以通过操作系统的System Environment ENV来指定
* 注意key为全大写

1. 通过配置文件

最后一个方式是通过配置文件来指定env=YOUR-ENVIRONMENT

* 对于Mac/Linux，文件位置为/opt/settings/server.properties
* 对于Windows，文件位置为C:\opt\settings\server.properties

文件内容形如：

env=DEV

目前，env支持以下几个值（大小写不敏感）：

* DEV : Development environment
* FAT：Feature Acceptance Test environment
* UAT：User Acceptance Test environment
* PRO：Production environment

#### 3.2.3.2 Cluster集群

Apollo支持配置按照集群划分，也就是说对于同一个appId和同一个环境，对不同的集群可以有不同的配置。

1.0.0版本开始支持以下方式集群，按照优先级从高到底分别为：

1. 通过Java System Property apollo.cluster

* 可以通过Java的System Property apollo.cluster来指定
* 在Java程序启动脚本中，可以指定-Dapollo.cluster=SomeCluster
* 如果是运行jar文件，需要注意格式是java -Dapollo.cluster=SomeCluster -jar xxx.jar
* 也可以通过程序指定，如System.setProperty("apollo.cluster", "SomeCluster");

1. 通过Spring Boot的配置文件

* 可以在Spring Boot的application.properties或bootstrap.properties中指定apollo.cluster=SomeCluster

1. 通过server.properties配置文件

* 可以在server.properties配置文件中指定idc=xxx
* 对于Mac/Linux，文件位置为/opt/settings/server.properties
* 对于Windows，文件位置为C:\opt\settings\server.properties
* **Cluster Precedence（集群顺序）**

1. 如果apollo.cluster和idc同时指定：
   * 首先尝试从apollo.cluster指定的集群加载配置
   * 如果没找到任何配置，会尝试从idc指定的集群加载配置
   * 如果还是没找到，会从默认的集群（default）加载
2. 如果只指定了apollo.cluster：
   * 首先尝试从apollo.cluster指定的集群加载配置
   * 如果没找到，会从默认的集群（default）加载
3. 如果只指定了idc：
   * 首先尝试从idc指定的集群加载配置
   * 如果没找到，会从默认的集群（default）加载
4. 如果apollo.cluster和idc都没有指定：
   * 从默认的集群（default）加载配置

## 3.2 Maven依赖

Apollo的客户端jar包已经上传到中央仓库，应用在实际使用时只需要按照如下方式引入即可。

<dependency>

<groupId>com.ctrip.framework.apollo</groupId>

<artifactId>apollo-client</artifactId>

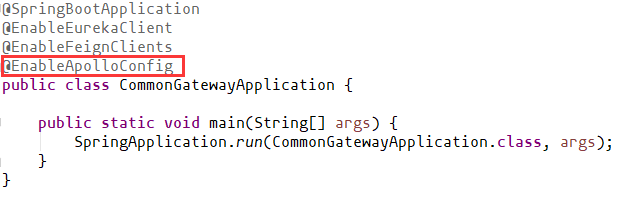
<version>1.1.0</version>

</dependency>

## 3.3 客户端用法

### 3.3.1 简易用法

apollo客户端最简单的用法，在springboot的启动类上添加注解@EnableApolloConfig



### 3.3.2 自定义用法

这里主要介绍和Spring整合方式。Apollo客户端使用Spring方式接入简单，结合Spring有多种酷炫玩法。如：

* Placeholder方式
  + 代码中使用，如：@Value("${key:default}")
  + application.properties中使用。如：spring.datasource.url= ${key:default}
* Spring boot的@ConfigurationProperties方式
* Spring方式也可以结合API方式使用，如注入Apollo的Config对象，就可以照常通过API方式获取配置了：

@ApolloConfig

private Config config; //inject config for namespace application

#### 3.3.2.1 SpringBoot推荐接入方式

Apollo会在Spring的postProcessBeanFactory阶段注入配置到Spring的Environment中，早于bean的初始化阶段，所以对于普通的bean注入配置场景已经能很好的满足。

不过Spring Boot有一些场景需要配置在更早的阶段注入，比如使用@ConditionalOnProperty的场景或者是有一些spring-boot-starter在启动阶段就需要读取配置做一些事情（如spring-boot-starter-dubbo)，所以对于Spring Boot环境建议通过以下方式来接入Apollo(需要0.10.0及以上版本）。

使用方式很简单，只需要在application.properties/bootstrap.properties中按照如下样例配置即可。

1. 在bootstrap阶段注入默认application namespace的配置示例

# will inject 'application' namespace in bootstrap phase

apollo.bootstrap.enabled = true

1. 在bootstrap阶段注入非默认application namespace或多个namespace的配置示例

apollo.bootstrap.enabled = true

# will inject 'application' and 'FX.apollo' namespaces in bootstrap phase

apollo.bootstrap.namespaces = application,FX.apollo

#### 3.3.2.2 Spring Placeholder的使用

Spring应用通常会使用Placeholder来注入配置，使用的格式形如${someKey:someDefaultValue}，如${timeout:100}。冒号前面的是key，冒号后面的是默认值。

建议在实际使用时尽量给出默认值，以免由于key没有定义导致运行时错误。从v0.10.0开始的版本支持placeholder在运行时自动更新。

##### 3.3.2.2.1 Java Config使用方式

例如有一个TestJavaConfigBean，通过Java Config的方式可以使用@Value的方式注入属性。

public class TestJavaConfigBean {

@Value("${timeout:100}")

private int timeout;

private int batch;

@Value("${batch:200}")

public void setBatch(int batch) {

this.batch = batch;

}

public int getTimeout() {

return timeout;

}

public int getBatch() {

return batch;

}

}

在Configuration类中按照下面的方式使用（假设应用默认的application namespace中有timeout和batch的配置项）：

@Configuration

@EnableApolloConfig

public class AppConfig {

@Bean

public TestJavaConfigBean javaConfigBean() {

return new TestJavaConfigBean();

}

}

这样，就能实时获取application中timeout和batch的值。

##### 3.3.2.2.2 ConfigurationProperties使用方式

Spring Boot提供了@ConfigurationProperties把配置注入到bean对象中。

Apollo也支持这种方式，下面的例子会把redis.cache.expireSeconds和redis.cache.commandTimeout分别注入到SampleRedisConfig的expireSeconds和commandTimeout字段中。

@ConfigurationProperties(prefix = "redis.cache")

public class SampleRedisConfig {

private int expireSeconds;

private int commandTimeout;

public void setExpireSeconds(int expireSeconds) {

this.expireSeconds = expireSeconds;

}

public void setCommandTimeout(int commandTimeout) {

this.commandTimeout = commandTimeout;

}

}

在Configuration类中按照下面的方式使用（假设应用默认的application namespace中有redis.cache.expireSeconds和redis.cache.commandTimeout的配置项）：

@Configuration

@EnableApolloConfig

public class AppConfig {

@Bean

public SampleRedisConfig sampleRedisConfig() {

return new SampleRedisConfig();

}

}

需要注意的是，@ConfigurationProperties如果需要在Apollo配置变化时自动更新注入的值，需要配合使用[EnvironmentChangeEvent](https://cloud.spring.io/spring-cloud-static/spring-cloud.html#_environment_changes)或[RefreshScope](https://cloud.spring.io/spring-cloud-static/spring-cloud.html#_refresh_scope)。

##### 3.3.2.2.3 Spring Annotation支持

Apollo同时还增加了几个新的Annotation来简化在Spring环境中的使用。

* @ApolloConfig
  + 用来自动注入Config对象
* @ApolloConfigChangeListener
  + 用来自动注册ConfigChangeListener
* @ApolloJsonValue
  + 用来把配置的json字符串自动注入为对象

public class TestApolloAnnotationBean {

@ApolloConfig

private Config config; //注入namespace为application的配置

@ApolloConfig("application")

private Config anotherConfig; //注入namespace为application的配置

@ApolloConfig("FX.apollo")

private Config yetAnotherConfig; //注入namespace为FX.apollo的配置

/\*\*

\* 在字段示例上注释的ApolloJsonValue，默认值被指定为空列表

\* <br />

\* jsonBeanProperty=[{"someString":"hello","someInt":100},{"someString":"world!","someInt":200}]

\*/

@ApolloJsonValue("${jsonBeanProperty:[]}")

private List<JsonBean> anotherJsonBeans;

@Value("${batch:100}")

private int batch;

//监听namespace为application的配置变化

@ApolloConfigChangeListener

private void someOnChange(ConfigChangeEvent changeEvent) {

//如果apollo发布了batch的最新值，就更新

if (changeEvent.isChanged("batch")) {

batch = config.getIntProperty("batch", 100);

}

}

//监听namespace为application的配置变化

@ApolloConfigChangeListener("application")

private void anotherOnChange(ConfigChangeEvent changeEvent) {

//可以发邮件，发短信通知等等

}

//监听namespace为application和FX.apollo的配置变化

@ApolloConfigChangeListener({"application", "FX.apollo"})

private void yetAnotherOnChange(ConfigChangeEvent changeEvent) {

//可以发邮件，发短信通知等等

}

// 直接从apollo获取值的示例

// 始终会返回timeout的最新值

public int getTimeout() {

return config.getIntProperty("timeout", 200);

}

//从注入值获取配置的示例

//当batch值在Apollo中更改时，程序需要使用上面显示的@ApolloConfigChangeListener更新注入值

public int getBatch() {

return this.batch;

}

private static class JsonBean{

private String someString;

private int someInt;

}

}

### 3.3.3 迁移已有配置

很多情况下，应用可能已经有不少配置了，比如Spring Boot的应用，就会有bootstrap.properties, application.properties等配置。

在应用接入Apollo之后，这些配置是可以非常方便的迁移到Apollo的，具体步骤如下：

1. 在Apollo为应用新建项目
2. 在应用中配置好META-INF/app.properties或在application.properties中配置好appId等。
3. 把原先配置（必须是properties格式）复制一下，然后通过Apollo提供的文本编辑模式全部粘帖到应用的application namespace，发布配置
4. 如果原来是其它格式，如yml，请先转成properties格式
5. 把原先的配置文件如bootstrap.properties, application.properties从项目中删除
6. 如果需要保留本地配置文件，需要注意部分配置如server.port必须确保本地文件已经删除该配置项

### Java代码（Spring boot）

<dependency>  
 <groupId>com.ctrip.framework.apollo</groupId>  
 <artifactId>apollo-client</artifactId>  
 <version>1.3.0</version>  
</dependency>

**app:  
 id:** java // AppID

**apollo:  
 meta:** <http://192.168.17.132:8080> // 配置地址

@SpringBootApplication  
@EnableFeignClients  
@EnableSwagger2  
@EnableEurekaClient  
@EnableApolloConfig(value=”namespace”) // namespace不填默认为application  
public class AggregationApplication {  
 public static void main(String[] args) {  
 ConfigurableApplicationContext app = SpringApplication.*run*(AggregationApplication.class, args);

SpringContextUtils.*getInstance*().setCfgContext(app);  
 }  
}

@RestController  
@RequestMapping(produces = MediaType.*APPLICATION\_JSON\_UTF8\_VALUE*)  
@RefreshScope // 实时刷新配置

public class AggregationController {  
  
 @Value("${test.name}") // 配置来自Apollo配置中心

private String name;  
  
 @ApiOperation(value = "简单测试")  
 @GetMapping  
 public String get(int a) {  
 return "ssss";  
 }  
}