

Java客户端使用指南

[Jump to bottom](#)

Jason Song edited this page on 3 Oct 2020 · 150 revisions

- [一、准备工作](#)
- [二、Maven Dependency](#)
- [三、客户端用法](#)
 - [3.1 API使用方式](#)
 - [3.1.1 获取默认namespace的配置](#)
 - [3.1.2 监听配置变化事件](#)
 - [3.1.3 获取公共Namespace的配置](#)
 - [3.1.4 获取非properties格式namespace的配置](#)
 - [3.2 Spring整合方式](#)
 - [3.2.1 配置](#)
 - [3.2.1.1 基于XML的配置](#)
 - [3.2.1.2 基于Java的配置（推荐）](#)
 - [3.2.1.3 Spring Boot集成方式（推荐）](#)
 - [3.2.2 Spring Placeholder的使用](#)
 - [3.2.3 Spring Annotation支持](#)
 - [3.2.4 已有配置迁移](#)
 - [3.3 Demo](#)
- [四、客户端设计](#)
- [五、本地开发模式](#)
- [六、测试模式](#)

注意：本文档适用对象是Apollo系统的使用者，如果你是公司内Apollo系统的开发者/维护人员，建议先参考[Apollo开发指南](#)。

一、准备工作

1.1 环境要求

- Java: 1.7+
- Guava: 15.0+
 - Apollo客户端默认会引用Guava 19，如果你的项目引用了其它版本，请确保版本号大于等于15.0

注：对于Apollo客户端，如果有需要的话，可以做少量代码修改来降级到Java 1.6，详细信息可以参考[Issue 483](#)

1.2 必选设置

Apollo客户端依赖于 AppId，Apollo Meta Server 等环境信息来工作，所以请确保阅读下面的说明并且做正确的配置：

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。

有以下几种方式设置，按照优先级从高到低分别为：

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
```

2. 操作系统的System Environment

Apollo 1.4.0+支持通过操作系统的System Environment APP_ID 来传入app.id信息，如

```
APP_ID=YOUR-APP-ID
```

3. 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
```

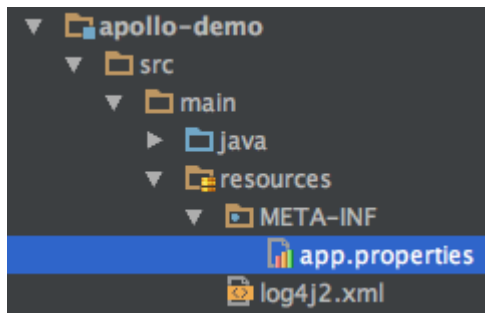
该配置方式不适用于多个war包部署在同一个tomcat的使用场景

4. 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
```

文件位置参考如下：



注：app.id是用来标识应用身份的唯一id，格式为string。

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、缩容等都可能導致IP列表的变化。

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

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");`

2. 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`

该配置方式不适用于多个war包部署在同一个tomcat的使用场景

3. 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

4. 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`

5. 通过 `app.properties` 配置文件

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

6. 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`
- 使用该配置方式, 那么就必须正确配置Environment, 详见[1.2.4.1 Environment](#)

7. 通过操作系统的System Environment `${ENV}_META` (1.2.0版本开始支持)

- 如果当前`env`是 `dev` , 那么用户可以配置操作系统的System Environment `DEV_META=http://config-service-url`
- 注意key为全大写
- 使用该配置方式, 那么就必须正确配置Environment, 详见[1.2.4.1 Environment](#)

8. 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
- 使用该配置方式, 那么就必须正确配置Environment, 详见[1.2.4.1 Environment](#)
- 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

1.2.2.1 自定义Apollo Meta Server地址定位逻辑

在1.0.0版本中, Apollo提供了MetaServerProvider SPI, 用户可以注入自己的MetaServerProvider来自定义Meta Server地址定位逻辑。

由于我们使用典型的Java Service Loader模式, 所以实现起来还是比较简单的。

有一点需要注意的是, apollo会在运行时按照顺序遍历所有的MetaServerProvider, 直到某一个MetaServerProvider提供了一个非空的Meta Server地址, 因此用户需要格外注意自定义MetaServerProvider的Order。规则是较小的Order具有较高的优先级, 因此Order=0的MetaServerProvider会排在Order=1的MetaServerProvider的前面。

如果你的公司有很多应用需要接入Apollo, 建议封装一个jar包, 然后提供自定义的Apollo Meta Server定位逻辑, 从而可以让接入Apollo的应用零配置使用。比如自己写一个 `xx-company-apollo-client` , 该jar包依赖 `apollo-client` , 在该jar包中通过spi方式定义自定义的MetaServerProvider实现, 然后应用直接依赖 `xx-company-apollo-client` 即可。

MetaServerProvider的实现可以参考[LegacyMetaServerProvider](#)和[DefaultMetaServerProvider](#)。

1.2.2.2 跳过Apollo Meta Server服务发现

适用于apollo-client 0.11.0及以上版本

一般情况下都建议使用Apollo的Meta Server机制来实现Config Service的服务发现，从而可以实现Config Service的高可用。不过apollo-client也支持跳过Meta Server服务发现，主要用于以下场景：

1. Config Service部署在公有云上，注册到Meta Server的是内网地址，本地开发环境无法直接连接
 - 如果通过公网 SLB 对外暴露 Config Service的话，记得要设置 IP 白名单，避免数据泄露
2. Config Service部署在docker环境中，注册到Meta Server的是docker内网地址，本地开发环境无法直接连接
3. Config Service部署在kubernetes中，希望使用kubernetes自带的服务发现能力（Service）

针对以上场景，可以通过直接指定Config Service地址的方式来跳过Meta Server服务发现，按照优先级从高到低分别为：

1. 通过Java System Property `apollo.configService`
 - 可以通过Java的System Property `apollo.configService` 来指定
 - 在Java程序启动脚本中，可以指定 `-Dapollo.configService=http://config-service-url:port`
 - 如果是运行jar文件，需要注意格式是 `java -Dapollo.configService=http://config-service-url:port -jar xxx.jar`
 - 也可以通过程序指定，如 `System.setProperty("apollo.configService", "http://config-service-url:port");`
2. 通过操作系统的System Environment `APOLLO_CONFIGSERVICE`
 - 可以通过操作系统的System Environment `APOLLO_CONFIGSERVICE` 来指定
 - 注意key为全大写，且中间是 `_` 分隔
3. 通过 `server.properties` 配置文件
 - 可以在 `server.properties` 配置文件中指定 `apollo.configService=http://config-service-url:port`
 - 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`

1.2.3 本地缓存路径

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`

- appId就是应用自己的appId，如100004458
- cluster就是应用使用的集群，一般在本地模式下没有做过配置的话，就是default
- namespace就是应用使用的配置namespace，一般是application

```
/opt/data/100004458/config-cache $ll
total 16
-rw-r--r--  1 jason  staff   69B Jul 26 10:33 100004458+default+FX.apollo.properties
-rw-r--r--  1 jason  staff   79B Jul 26 10:33 100004458+default+application.properties
```

文件内容以properties格式存储，比如如果有两个key，一个是request.timeout，另一个是batch，那么文件内容就是如下格式：

```
request.timeout=2000
batch=2000
```

1.2.3.1 自定义缓存路径

1.0.0版本开始支持以下方式自定义缓存路径，按照优先级从高到低分别为：

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");`
2. 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`
3. 通过操作系统的System Environment `APOLLO_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
4. 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`

注：本地缓存路径也可用于容灾目录，如果应用在所有config service都挂掉的情况下需要扩容，那么也可以先把配置从已有机器上的缓存路径复制到新机器上的相同缓存路径

1.2.4 可选设置

1.2.4.1 Environment

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

3. 通过配置文件

- 最后一个推荐的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

更多环境定义，可以参考[Env.java](#)

1.2.4.2 Cluster（集群）

Apollo支持配置按照集群划分，也就是说对于一个appId和一个环境，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，按照优先级从高到低分别为：

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");`

2. 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`

3. 通过Java System Property

- 可以通过Java的System Property `idc` 来指定环境
- 在Java程序启动脚本中，可以指定 `-Didc=xxx`
 - 如果是运行jar文件，需要注意格式是 `java -Didc=xxx -jar xxx.jar`
- 注意key为全小写

4. 通过操作系统的System Environment

- 还可以通过操作系统的System Environment `IDC` 来指定
- 注意key为全大写

5. 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：

- 我们会首先尝试从 `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`：

- 我们会首先尝试从 `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`：

- 我们会首先尝试从 `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：

- 我们会从默认的集群（`default`）加载配置

1.2.4.3 设置内存中的配置项是否保持和页面上的顺序一致

适用于1.6.0及以上版本

默认情况下，`apollo client`内存中的配置存放在`Properties`中（底下是`Hashtable`），不会刻意保持和页面上看到的顺序一致，对绝大部分的场景是没有影响的。不过有些场景会强依赖配置项的顺序（如`spring cloud zuul`的路由规则），针对这种情况，可以开启`OrderedProperties`特性来使得内存中的配置顺序和页面上看到的一致。

配置方式按照优先级从高到低分别为：

1. 通过Java System Property `apollo.property.order.enable`
 - 可以通过Java的System Property `apollo.property.order.enable` 来指定
 - 在Java程序启动脚本中，可以指定 `-Dapollo.property.order.enable=true`
 - 如果是运行jar文件，需要注意格式是 `java -Dapollo.property.order.enable=true -jar xxx.jar`
 - 也可以通过程序指定，如 `System.setProperty("apollo.property.order.enable", "true");`
2. 通过Spring Boot的配置文件
 - 可以在Spring Boot的 `application.properties` 或 `bootstrap.properties` 中指定 `apollo.property.order.enable=true`
3. 通过 `app.properties` 配置文件
 - 可以在 `classpath:/META-INF/app.properties` 指定 `apollo.property.order.enable=true`

1.2.4.4 配置访问密钥

适用于1.6.0及以上版本

Apollo从1.6.0版本开始增加访问密钥机制，从而只有经过身份验证的客户端才能访问敏感配置。如果应用开启了访问密钥，客户端需要配置密钥，否则无法获取配置。

配置方式按照优先级从高到低分别为：

1. 通过Java System Property `apollo.accesskey.secret`
 - 可以通过Java的System Property `apollo.accesskey.secret` 来指定
 - 在Java程序启动脚本中，可以指定 `-Dapollo.accesskey.secret=1cf998c4e2ad4704b45a98a509d15719`
 - 如果是运行jar文件，需要注意格式是 `java -Dapollo.accesskey.secret=1cf998c4e2ad4704b45a98a509d15719 -jar xxx.jar`
 - 也可以通过程序指定，如 `System.setProperty("apollo.accesskey.secret", "1cf998c4e2ad4704b45a98a509d15719");`

2. 通过Spring Boot的配置文件

- 可以在Spring Boot的 `application.properties` 或 `bootstrap.properties` 中指定 `apollo.accesskey.secret=1cf998c4e2ad4704b45a98a509d15719`

3. 通过操作系统的System Environment

- 还可以通过操作系统的System Environment `APOLLO_ACCESSKEY_SECRET` 来指定
- 注意key为全大写

4. 通过 `app.properties` 配置文件

- 可以在 `classpath:/META-INF/app.properties` 指定 `apollo.accesskey.secret=1cf998c4e2ad4704b45a98a509d15719`

二、Maven Dependency

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>
```

三、客户端用法

Apollo支持API方式和Spring整合方式，该怎么选择用哪一种方式？

- API方式灵活，功能完备，配置值实时更新（热发布），支持所有Java环境。
- Spring方式接入简单，结合Spring有N种酷炫的玩法，如
 - Placeholder方式：
 - 代码中直接使用，如： `@Value("${someKeyFromApollo:someDefaultValue}")`
 - 配置文件中替换placeholder，如： `spring.datasource.url:${someKeyFromApollo:someDefaultValue}`
 - 直接托管spring的配置，如在apollo中直接配置 `spring.datasource.url=jdbc:mysql://localhost:3306/somedb?characterEncoding=utf8`
 - Spring boot的[@ConfigurationProperties](#)方式
 - 从v0.10.0开始的版本支持placeholder在运行时自动更新，具体参见[PR #972](#)。（v0.10.0之前的版本在配置变化后不会重新注入，需要重启才会更新，如果需要配置值实时更新，可以参考后续[3.2.2 Spring Placeholder的使用](#)的说明）
- 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
```

- 更多有意思的实际使用场景和示例代码，请参考[apollo-use-cases](#)

3.1 API使用方式

API方式是最简单、高效使用Apollo配置的方式，不依赖Spring框架即可使用。

3.1.1 获取默认namespace的配置（application）

```
Config config = ConfigService.getAppConfig(); //config instance is singleton for e
String someKey = "someKeyFromDefaultNamespace";
String someDefaultValue = "someDefaultValueForTheKey";
String value = config.getProperty(someKey, someDefaultValue);
```

通过上述的`config.getProperty`可以获取到`someKey`对应的实时最新的配置值。

另外，配置值从内存中获取，所以不需要应用自己做缓存。

3.1.2 监听配置变化事件

监听配置变化事件只在应用真的关心配置变化，需要在配置变化时得到通知时使用，比如：数据库连接串变化后需要重建连接等。

如果只是希望每次都取到最新的配置的话，只需要按照上面的例子，调用`config.getProperty`即可。

```
Config config = ConfigService.getAppConfig(); //config instance is singleton for e
config.addChangeListener(new ConfigChangeListener() {
    @Override
    public void onChange(ConfigChangeEvent changeEvent) {
        System.out.println("Changes for namespace " + changeEvent.getNamespace());
        for (String key : changeEvent.changedKeys()) {
            ConfigChange change = changeEvent.getChange(key);
            System.out.println(String.format("Found change - key: %s, oldValue: %s",
                key, change.getOldValue()));
        }
    }
});
```

3.1.3 获取公共Namespace的配置

```
String somePublicNamespace = "CAT";
Config config = ConfigService.getConfig(somePublicNamespace); //config instance is
String someKey = "someKeyFromPublicNamespace";
String someDefaultValue = "someDefaultValueForTheKey";
String value = config.getProperty(someKey, someDefaultValue);
```

3.1.4 获取非properties格式namespace的配置

3.1.4.1 yaml/yml格式的namespace

apollo-client 1.3.0版本开始对yaml/yml做了更好的支持，使用起来和properties格式一致。

```
Config config = ConfigService.getConfig("application.yml");
String someKey = "someKeyFromYmlNamespace";
String someDefaultValue = "someDefaultValueForTheKey";
String value = config.getProperty(someKey, someDefaultValue);
```

3.1.4.2 非yaml/yml格式的namespace

获取时需要使用 `ConfigService.getConfigFile` 接口并指定Format，如 `ConfigFileFormat.XML` 。

```
String someNamespace = "test";
ConfigFile configFile = ConfigService.getConfigFile("test", ConfigFileFormat.XML);
String content = configFile.getContent();
```

3.2 Spring整合方式

3.2.1 配置

Apollo也支持和Spring整合（Spring 3.1.1+），只需要做一些简单的配置就可以了。

Apollo目前既支持比较传统的 基于XML 的配置，也支持目前比较流行的 基于Java（推荐） 的配置。

如果是Spring Boot环境，建议参照[3.2.1.3 Spring Boot集成方式（推荐）](#) 配置。

需要注意的是，如果之前有使用

`org.springframework.beans.factory.config.PropertyPlaceholderConfigurer` 的，请替换成 `org.springframework.context.support.PropertySourcesPlaceholderConfigurer` 。Spring 3.1以后就不建议使用`PropertyPlaceholderConfigurer`了，要改用 `PropertySourcesPlaceholderConfigurer`。

如果之前有使用 `<context:property-placeholder>`，请注意xml中引入的 `spring-context.xsd` 版本需要是3.1以上（一般只要没有指定版本会自动升级的），建议使用不带版本号的形式引入，如：`http://www.springframework.org/schema/context/spring-context.xsd`

注1: yaml/yml格式的namespace从1.3.0版本开始支持和Spring整合，注入时需要填写带后缀的完整名字，比如application.yml

注2: 非properties、非yaml/yml格式（如xml, json等）的namespace暂不支持和Spring整合。

3.2.1.1 基于XML的配置

注：需要把apollo相关的xml namespace加到配置文件头上，不然会报xml语法错误。

1.注入默认namespace的配置到Spring中

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:apollo="http://www.ctrip.com/schema/apollo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
       http://www.ctrip.com/schema/apollo http://www.ctrip.com/schema/apollo.xsd">
  <!-- 这个是最简单的配置形式，一般应用用这种形式就可以了，用来指示Apollo注入application na
  <apollo:config/>
  <bean class="com.ctrip.framework.apollo.spring.TestXmlBean">
    <property name="timeout" value="${timeout:100}"/>
    <property name="batch" value="${batch:200}"/>
  </bean>
</beans>
```

2.注入多个namespace的配置到Spring中

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:apollo="http://www.ctrip.com/schema/apollo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
       http://www.ctrip.com/schema/apollo http://www.ctrip.com/schema/apollo.xsd">
  <!-- 这个是最简单的配置形式，一般应用用这种形式就可以了，用来指示Apollo注入application na
  <apollo:config/>
  <!-- 这个是稍微复杂一些的配置形式，指示Apollo注入FX.apollo和application.yml namespace
  <apollo:config namespaces="FX.apollo,application.yml"/>
  <bean class="com.ctrip.framework.apollo.spring.TestXmlBean">
    <property name="timeout" value="${timeout:100}"/>
    <property name="batch" value="${batch:200}"/>
  </bean>
</beans>
```

3.注入多个namespace，并且指定顺序

Spring的配置是有顺序的，如果多个property source都有同一个key，那么最终是顺序在前的配置生效。

apollo:config如果不指定order，那么默认是最低优先级。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:apollo="http://www.ctrip.com/schema/apollo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
       http://www.ctrip.com/schema/apollo http://www.ctrip.com/schema/apollo.xsd">
  <apollo:config order="2"/>
  <!-- 这个是最复杂的配置形式，指示Apollo注入FX.apollo和application.yml namespace的配置 -->
  <apollo:config namespaces="FX.apollo,application.yml" order="1"/>
  <bean class="com.ctrip.framework.apollo.spring.TestXmlBean">
    <property name="timeout" value="${timeout:100}"/>
    <property name="batch" value="${batch:200}"/>
  </bean>
</beans>
```

3.2.1.2 基于Java的配置（推荐）

相对于基于XML的配置，基于Java的配置是目前比较流行的方式。

注意 @EnableApolloConfig 要和 @Configuration 一起使用，不然不会生效。

1.注入默认namespace的配置到Spring中

```
//这个是最简单的配置形式，一般应用用这种形式就可以了，用来指示Apollo注入application namespace
@Configuration
@EnableApolloConfig
public class AppConfig {
    @Bean
    public TestJavaConfigBean javaConfigBean() {
        return new TestJavaConfigBean();
    }
}
```

2.注入多个namespace的配置到Spring中

```
@Configuration
@EnableApolloConfig
public class SomeAppConfig {
    @Bean
    public TestJavaConfigBean javaConfigBean() {
        return new TestJavaConfigBean();
    }
}
```

```

    }
}

```

//这个是稍微复杂一些的配置形式，指示Apollo注入FX.apollo和application.yml namespace的配置到Spring

```

@Configuration
@EnableApolloConfig({"FX.apollo", "application.yml"})
public class AnotherAppConfig {}

```

3.注入多个namespace，并且指定顺序

//这个是最复杂的配置形式，指示Apollo注入FX.apollo和application.yml namespace的配置到Spring

```

@Configuration
@EnableApolloConfig(order = 2)
public class SomeAppConfig {
    @Bean
    public TestJavaConfigBean javaConfigBean() {
        return new TestJavaConfigBean();
    }
}
@Configuration
@EnableApolloConfig(value = {"FX.apollo", "application.yml"}, order = 1)
public class AnotherAppConfig {}

```

3.2.1.3 Spring Boot集成方式（推荐）

Spring Boot除了支持上述两种集成方式以外，还支持通过application.properties/bootstrap.properties来配置，该方式能使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[dubbo-spring-boot-project](#)），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. 注入默认 application namespace的配置示例

```

# will inject 'application' namespace in bootstrap phase
apollo.bootstrap.enabled = true

```

2. 注入非默认 application namespace或多个namespace的配置示例

```

apollo.bootstrap.enabled = true
# will inject 'application', 'FX.apollo' and 'application.yml' namespaces in
apollo.bootstrap.namespaces = application,FX.apollo,application.yml

```


3. 将Apollo配置加载提到初始化日志系统之前(1.2.0+)

从1.2.0版本开始，如果希望把日志相关的配置（如 `logging.level.root=info` 或 `logback-spring.xml` 中的参数）也放在Apollo管理，那么可以额外配置

`apollo.bootstrap.eagerLoad.enabled=true` 来使Apollo的加载顺序放到日志系统加载之前，不过这会导致Apollo的启动过程无法通过日志的方式输出(因为执行Apollo加载的时候，日志系统压根没有准备好呢！所以在Apollo代码中使用Slf4j的日志输出便没有任何内容)，更多信息可以参考 [PR 1614](#)。参考配置示例如下：

```
# will inject 'application' namespace in bootstrap phase
apollo.bootstrap.enabled = true
# put apollo initialization before logging system initialization
apollo.bootstrap.eagerLoad.enabled=true
```

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，具体参见[PR #972](#)。

如果需要关闭placeholder在运行时自动更新功能，可以通过以下两种方式关闭：

1. 通过设置System Property `apollo.autoUpdateInjectedSpringProperties`，如启动时传入 `-Dapollo.autoUpdateInjectedSpringProperties=false`
2. 通过设置META-INF/app.properties中的 `apollo.autoUpdateInjectedSpringProperties` 属性，如

```
app.id=SampleApp
apollo.autoUpdateInjectedSpringProperties=false
```

3.2.2.1 XML使用方式

假设我有一个TestXmlBean，它有两个配置项需要注入：

```
public class TestXmlBean {
    private int timeout;
    private int batch;

    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    public void setBatch(int batch) {
```

```

    this.batch = batch;
}

public int getTimeout() {
    return timeout;
}

public int getBatch() {
    return batch;
}
}

```

那么，我在XML中会使用如下方式来定义（假设应用默认的application namespace中有timeout和batch的配置项）：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:apollo="http://www.ctrip.com/schema/apollo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
       http://www.ctrip.com/schema/apollo http://www.ctrip.com/schema/apollo.xsd">
  <apollo:config/>
  <bean class="com.ctrip.framework.apollo.spring.TestXmlBean">
    <property name="timeout" value="${timeout:100}"/>
    <property name="batch" value="${batch:200}"/>
  </bean>
</beans>

```

3.2.2.2 Java Config使用方式

假设我有一个TestJavaConfigBean，通过Java Config的方式还可以使用@Value的方式注入：

```

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();
    }
}
```

3.2.2.3 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](#)或[RefreshScope](#)。相关代码实现，可以参考[apollo-use-cases](#)项目中的[ZuulPropertiesRefresher.java](#)和[apollo-demo](#)项目中的[SampleRedisConfig.java](#)以及[SpringBootApolloRefreshConfig.java](#)

3.2.3 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。

1. `@ApolloConfig`
 - 用来自动注入Config对象
2. `@ApolloConfigChangeListener`
 - 用来自动注册ConfigChangeListener
3. `@ApolloJsonValue`
 - 用来把配置的json字符串自动注入为对象

使用样例如下：

```
public class TestApolloAnnotationBean {
    @ApolloConfig
    private Config config; //inject config for namespace application
    @ApolloConfig("application")
    private Config anotherConfig; //inject config for namespace application
    @ApolloConfig("FX.apollo")
    private Config yetAnotherConfig; //inject config for namespace FX.apollo
    @ApolloConfig("application.yml")
    private Config yamlConfig; //inject config for namespace application.yml

    /**
     * ApolloJsonValue annotated on fields example, the default value is specified a
     * <br />
     * jsonBeanProperty=[{"someString":"hello","someInt":100},{"someString":"world!"}
     */
    @ApolloJsonValue("${jsonBeanProperty:[]}")
    private List<JsonBean> anotherJsonBeans;

    @Value("${batch:100}")
    private int batch;

    //config change listener for namespace application
    @ApolloConfigChangeListener
    private void someOnChange(ConfigChangeEvent changeEvent) {
        //update injected value of batch if it is changed in Apollo
        if (changeEvent.isChanged("batch")) {
            batch = config.getIntProperty("batch", 100);
        }
    }

    //config change listener for namespace application
    @ApolloConfigChangeListener("application")
```

```

private void anotherOnChange(ConfigChangeEvent changeEvent) {
    //do something
}

//config change listener for namespaces application, FX.apollo and application.y
@ApolloChangeListener({"application", "FX.apollo", "application.yml"})
private void yetAnotherOnChange(ConfigChangeEvent changeEvent) {
    //do something
}

//example of getting config from Apollo directly
//this will always return the latest value of timeout
public int getTimeout() {
    return config.getIntProperty("timeout", 200);
}

//example of getting config from injected value
//the program needs to update the injected value when batch is changed in Apollo
public int getBatch() {
    return this.batch;
}

private static class JsonBean{
    private String someString;
    private int someInt;
}
}

```

在Configuration类中按照下面的方式使用：

```

@Configuration
@EnableApolloConfig
public class AppConfig {
    @Bean
    public TestApolloAnnotationBean testApolloAnnotationBean() {
        return new TestApolloAnnotationBean();
    }
}

```

3.2.4 已有配置迁移

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/yml, application.properties/yml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

3. 建议把原先配置先转为properties格式，然后通过Apollo提供的文本编辑模式全部粘贴到应用的application namespace，发布配置
 - 如果原来格式是yml，可以使用`YamlPropertiesFactoryBean.getObject`转成properties格式
4. 如果原来是yml，想继续使用yml来编辑配置，那么可以创建私有的application.yml namespace，把原来的配置全部粘贴进去，发布配置
 - 需要apollo-client是1.3.0及以上版本
5. 把原先的配置文件如bootstrap.properties/yml, application.properties/yml从项目中删除
 - 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

如：

```
spring.application.name = reservation-service
server.port = 8080
```

```
logging.level = ERROR
```

```
eureka.client.serviceUrl.defaultZone = http://127.0.0.1:8761/eureka/
eureka.client.healthcheck.enabled = true
eureka.client.registerWithEureka = true
eureka.client.fetchRegistry = true
eureka.client.eurekaServiceUrlPollIntervalSeconds = 60
```

```
eureka.instance.preferIpAddress = true
```

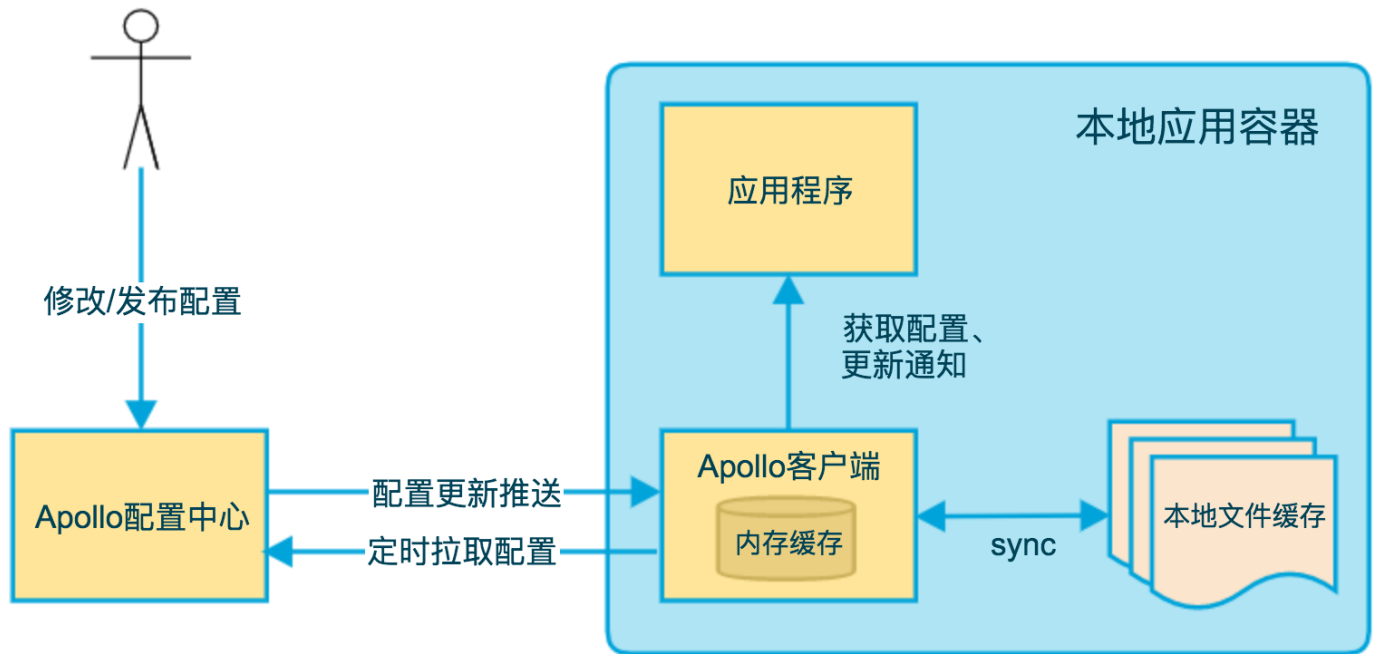


3.3 Demo

项目中有一个样例客户端的项目： `apollo-demo` ， 具体信息可以参考[Apollo开发指南](#)中的2.3 Java 样例客户端启动部分。

更多使用案例Demo可以参考[Apollo使用场景和示例代码](#)。

四、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

五、本地开发模式

Apollo客户端还支持本地开发模式，这个主要用于当开发环境无法连接Apollo服务器的时候，比如在邮轮、飞机上做相关功能开发。

在本地开发模式下，Apollo只会从本地文件读取配置信息，不会从Apollo服务器读取配置。

可以通过下面的步骤开启Apollo本地开发模式。

5.1 修改环境

修改/opt/settings/server.properties (Mac/Linux) 或C:\opt\settings\server.properties (Windows) 文件, 设置env为Local:

```
env=Local
```

更多配置环境的方式请参考[1.2.2 Environment](#)

5.2 准备本地配置文件

在本地开发模式下, Apollo客户端会从本地读取文件, 所以我们需要事先准备好配置文件。

5.2.1 本地配置目录

本地配置目录位于:

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

appId就是应用的appId, 如100004458。

请确保该目录存在, 且应用程序对该目录有读权限。

【小技巧】 推荐的方式是先和普通模式下使用Apollo, 这样Apollo会自动创建该目录并在目录下生成配置文件。

5.2.2 本地配置文件

本地配置文件需要按照一定的文件名格式放置于本地配置目录下, 文件名格式如下:

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

- appId就是应用自己的appId, 如100004458
- cluster就是应用使用的集群, 一般在本地模式下没有做过配置的话, 就是default
- namespace就是应用使用的配置namespace, 一般是application

```
▶ /opt/data/100004458/config-cache $ll
total 16
-rw-r--r--  1 jason  staff   69B Jul 26 10:33 100004458+default+FX.apollo.properties
-rw-r--r--  1 jason  staff   79B Jul 26 10:33 100004458+default+application.properties
```

文件内容以properties格式存储, 比如如果有两个key, 一个是request.timeout, 另一个是batch, 那么文件内容就是如下格式:

```
request.timeout=2000
batch=2000
```

5.3 修改配置

在本地开发模式下，Apollo不会实时监测文件内容是否有变化，所以如果修改了配置，需要重启应用生效。

六、测试模式

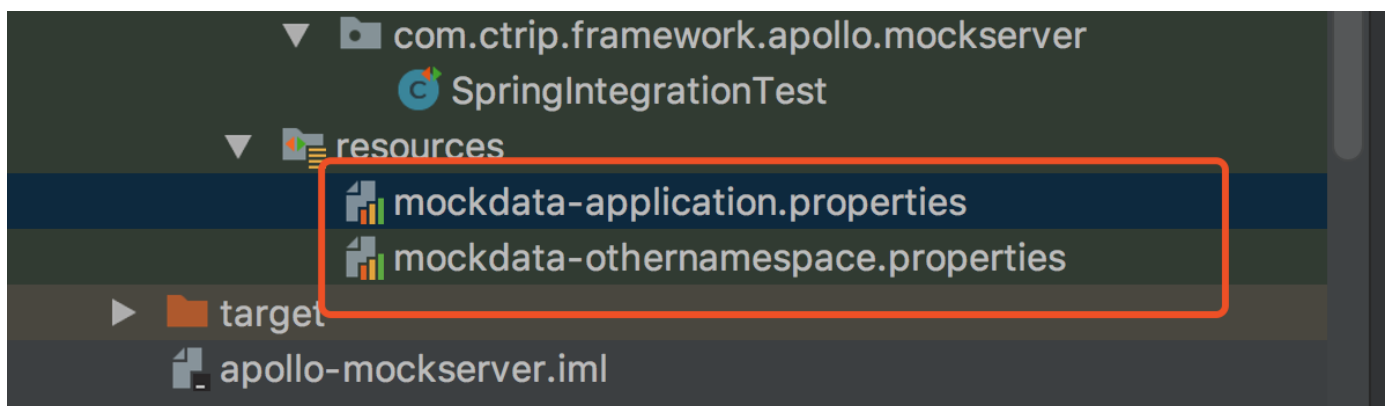
1.1.0版本开始增加了 `apollo-mockserver`，从而可以很好地支持单元测试时需要mock配置的场景，使用方法如下：

6.1 引入pom依赖

```
<dependency>
  <groupId>com.ctrip.framework.apollo</groupId>
  <artifactId>apollo-mockserver</artifactId>
  <version>1.1.0</version>
</dependency>
```

6.2 在test的resources下放置mock的数据

文件名格式约定为 `mockdata-{namespace}.properties`



6.3 写测试类

更多使用demo可以参考[ApolloMockServerApiTest.java](#)和[ApolloMockServerSpringIntegrationTest.java](#)。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = TestConfiguration.class)
public class SpringIntegrationTest {
    // 启动apollo的mockserver
    @ClassRule
    public static EmbeddedApollo embeddedApollo = new EmbeddedApollo();
```

```
@Test
@DirtiesContext // 这个注解很有必要，因为配置注入会弄脏应用上下文
public void testPropertyInject(){
    assertEquals("value1", testBean.key1);
    assertEquals("value2", testBean.key2);
}

@Test
@DirtiesContext
public void testListenerTriggeredByAdd() throws InterruptedException, ExecutionException {
    String otherNamespace = "othernamespace";
    embeddedApollo.addOrModifyProperty(otherNamespace, "someKey", "someValue");
    ConfigChangeEvent changeEvent = testBean.futureData.get(5000, TimeUnit.MILLISECONDS);
    assertEquals(otherNamespace, changeEvent.getNamespace());
    assertEquals("someValue", changeEvent.getChange("someKey").getNewValue());
}

@EnableApolloConfig("application")
@Configuration
static class TestConfiguration{
    @Bean
    public TestBean testBean(){
        return new TestBean();
    }
}

static class TestBean{
    @Value("${key1:default}")
    String key1;
    @Value("${key2:default}")
    String key2;

    SettableFuture<ConfigChangeEvent> futureData = SettableFuture.create();

    @ApolloConfigChangeListener("othernamespace")
    private void onChange(ConfigChangeEvent changeEvent) {
        futureData.set(changeEvent);
    }
}
}
```

► Pages 21

- 设计文档
 - [Apollo配置中心介绍](#)
 - [Apollo配置中心设计](#)
 - [Apollo核心概念之“Namespace”](#)
- 部署文档

- [Quick Start](#)
- [Docker方式部署Quick Start](#)
- [分布式部署指南](#)
- [Apollo源码解析（全）](#)
- 开发文档
 - [Apollo开发指南](#)
 - Code Styles
 - [Eclipse Code Style](#)
 - [IntelliJ Code Style](#)
 - [Portal实现用户登录功能](#)
 - [邮件模板样例](#)
- 系统使用文档
 - [Apollo使用指南](#)
 - [Java客户端使用指南](#)
 - [.Net客户端使用指南](#)
 - [Go、Python、NodeJS、PHP等客户端使用指南](#)
 - [其它语言客户端接入指南](#)
 - [Apollo开放平台接入指南](#)
 - [Apollo使用场景和示例代码](#)
 - [Apollo实践案例](#)
 - [Apollo安全相关最佳实践](#)
- FAQ
 - [常见问题回答](#)
 - [部署&开发遇到的常见问题](#)
- 其它
 - [版本历史](#)
 - [Apollo性能测试报告](#)

Clone this wiki locally

`https://github.com/apolloconfig/apollo.wiki.git`

