# 一站式微服务架构Spring Cloud

主讲：Cat 老师

北京动力节点教育科技有限公司  
2019 • 北京

动力节点•版权所有•禁止传播

# 快速回顾

1. 分布式与微服务架构的理论梳理；
2. 什么是Spring Cloud？
3. Spring Cloud的整体架构 （与Dubbo比较）
4. 服务消费者Controller直连调用服务提供者Controller（http协议的restful）
5. Spring Cloud的注册中心Eureka
6. Spring Cloud Eureka与Zookeeper比较
7. Spring Cloud Eureka高可用集群
8. Spring Cloud Eureka自我保护机制
9. Spring Cloud Ribbon负载均衡
10. Spring Cloud Feign声明式服务调用（与Dubbo接口层比较）
11. Spring Cloud Hystrix 服务熔断降级、服务限流
12. Spring Cloud Hystrix DashBoard仪表盘监控
13. Spring Cloud Hystrix Turbine聚合监控
14. Spring Cloud Zuul网关（路由、过滤、异常、降级）
15. Spring Cloud Config配置中心（用途、使用、加解密）
16. Spring Cloud Config 配置中心（自动刷新、高可用、安全认证）
17. Spring Cloud Sleuth 分布式链路跟踪
18. Spring Cloud Sleuth 整合Zipkin 分布式链路跟踪
19. Spring Cloud Stream消息驱动框架

# 本次课主题大纲

1. Spring Cloud 服务安全机制
2. Spring Cloud + Apollo配置中心
3. Spring Cloud 项目案例

# Spring Cloud 服务安全机制

微服务的Rest服务都是http的，有可能暴露在公网上，那么任何人都可能调用，如果你的Rest服务有一些私密信息，这样会导致信息的泄露，所以我们的微服务需要增加一些安全机制。

如果想进行安全方面的处理，首先要在服务上进行如下处理；

1、配置依赖

*<!--spring-boot-starter-security-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-security</**artifactId**>  
</**dependency**>

2、application.properties文件

*#认证用户名***spring.security.user.name**=**cat***#认证密码***spring.security.user.password**=**123456**

3、测试访问

**<http://localhost:9100/service/goods>**

**<http://localhost:9200/service/goods>**

此时就需要输入账号和密码才能访问接口;

遇到的坑：

消费者集成了Sleuth+zipkin导致只能消费一次，可能内部有点冲突，把Sleuth+zipkin去掉，恢复正常；

## 需要安全认证的服务调用

当远程的一个服务已经使用了密码验证，那么这个时候服务的消费方如果想直接访问就不能访问了，此时需要进行相关的处理：

### 如果是restTemplate调用

1、首先配置一个HttpHeaders；

*/\*\*  
 \* 进行Http头信息配置  
 \*  
 \** ***@return*** *\*/*@Bean  
**public** HttpHeaders getHeaders() {  
 *//定义HTTP的头信息* HttpHeaders headers = **new** HttpHeaders();  
 *//认证的原始信息* String auth = **"cat:123456"**;  
 *//进行加密处理* **byte**[] encodedAuth = Base64.*getEncoder*().encode(auth.getBytes(Charset.*forName*(**"US-ASCII"**)));  
 String authHeader = **"Basic "** + **new** String(encodedAuth);  
 headers.set(**"Authorization"**, authHeader);  
 **return** headers;  
}

2、然后在调用的地方注入该HttpHeaders；

@Autowired  
**private** HttpHeaders **httpHeaders**;

3、调用的时候要传入这个http头信息；

@RequestMapping(**"/cloud/goods2"**)  
**public** ResultObject goods2() {  
 System.***out***.println(**"/cloud/goods-->8080"**);  
 *//调用远程的一个controller, restful的调用* ResponseEntity<ResultObject> responseEntity = **restTemplate**.exchange(***GOODS\_SERVICE\_URL\_02***, HttpMethod.***GET***, **new** HttpEntity<Object>(**httpHeaders**), ResultObject.**class**);  
 **return** responseEntity.getBody();  
}

### 如果是feign的方式调用

1、需要一个配置类；

@Configuration  
**public class** FeignConfiguration {  
  
 *//@RequestLine("GET /service/goods")  
  
 /\*\*  
 \* 一种契约，采用feign的契约方式，如果不配置该bean，会转成SpringMVC的方式  
 \*/  
 /\*@Bean  
 public Contract feignContract(){  
 return new Contract.Default();  
 }\*/* @Bean  
 **public** BasicAuthRequestInterceptor basicAuthRequestInterceptor(){  
 *//传用户名和密码* **return new** BasicAuthRequestInterceptor(**"cat"**,**"123456"**);  
 }  
}

1. 然后在feign的声明式接口注解上加入配置；

@FeignClient(value=**"${service.id}"**,  
 */\*fallback=GoodsRemoteClientFallBack.class\*/* fallbackFactory = GoodsRemoteClientFallBackFactory.**class**,  
 configuration = FeignConfiguration.**class**)  
**public interface** GoodsRemoteClient {

这样便可以实现调用；

## 使用独立的安全服务

在实际项目开发中，服务一般都会非常多，绝大多数情况下，这些服务都需要用到安全验证，而且密码也会一样，如果每个服务都单独配置安全认证，工作量比较大，所以应该单独建立一个安全验证的项目，其他微服务如果需要安全认证就引入该项目的依赖即可；

此时我们需要使用如下的配置类来完成：

@Configuration  
@EnableWebSecurity  
**public class** WebSecurityConfiguration **extends** WebSecurityConfigurerAdapter {  
 @Override  
 **public void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  
 auth.inMemoryAuthentication()  
 .passwordEncoder(**new** BCryptPasswordEncoder())  
 .withUser(**"cat"**)  
 .password(**new** BCryptPasswordEncoder().encode(**"123456"**))  
 .roles(**"USER"**)  
 .and()  
 .withUser(**"admin"**)  
 .password(**new** BCryptPasswordEncoder().encode(**"123456"**))  
 .roles(**"USER"**, **"ADMIN"**);  
 }  
 @Override  
 **protected void** configure(HttpSecurity http) **throws** Exception {  
 http.httpBasic().and().authorizeRequests().anyRequest().fullyAuthenticated();  
 http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);  
 }  
}

完整配置具体请参考项目代码；

# Apollo阿波罗运行环境

## 1、环境要求：

推荐虚拟机内存在2G或以上；

JDK 1.8环境；

MySQL版本要求：5.6.5+；

Apollo的表结构对timestamp使用了多个default声明，所以需要MySQL 5.6.5以上版本；

可通过该语句查看MySQL版本：SHOW VARIABLES WHERE Variable\_name = 'version';

## 2、项目下载打包

从Github下载项目自己进行编译打包也可以；

不过apollo本身提供了一个可以直接使用的jar包；

<https://github.com/nobodyiam/apollo-build-scripts>

## 3、创建数据库

Apollo服务端共需要两个数据库：ApolloPortalDB和ApolloConfigDB，把数据库、表的创建和样例数据sql文件导入数据库即可；

apolloportaldb.sql

导入成功后，可以通过执行如下sql语句来验证：

select `Id`, `AppId`, `Name` from ApolloPortalDB.App;

apolloconfigdb.sql

导入成功后，可以通过执行如下sql语句来验证：

select `NamespaceId`, `Key`, `Value`, `Comment` from ApolloConfigDB.Item;

Apollo服务端需要知道如何连接到你创建的数据库，所以需要修改startup.sh，修改ApolloPortalDB和ApolloConfigDB两个数据库的连接信息；

注意：配置的用户需要具有对ApolloPortalDB和ApolloConfigDB数据库的读写权限；

#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）

注意：不要修改demo.sh的其它部分；

## 4、启动Apollo配置中心

执行启动脚本

**./demo.sh start**  (里面会启动三个服务，启动会比较慢)

[root@rabbit128 apollo-config-center]# ./startup.sh start

==== starting service ====

Service logging file is ./service/apollo-service.log

Started [1498]

Waiting for config service startup.

......

Config service started. You may visit http://localhost:8080 for service status now!

Waiting for admin service startup.

Admin service started

==== starting portal ====

Portal logging file is ./portal/apollo-portal.log

Started [1719]

Waiting for portal startup......

Portal started. You can visit http://localhost:8070 now!

启动成功后，访问：<http://ip:8070> 登录账号默认是： apollo/admin；

Config-service 8080端口

Admin-service 8090端口

Portal管控台 8070端口

# Apollo阿波罗应用开发

对于我们的应用开发而言，Apollo阿波罗分布式配置中心属于客户端-服务端模式，我们的应用程序是客户端，Apollo阿波罗是服务端；

1. 在项目中添加apollo的依赖；

*<!-- apollo-client -->*<**dependency**>  
 <**groupId**>com.ctrip.framework.apollo</**groupId**>  
 <**artifactId**>apollo-client</**artifactId**>  
 <**version**>1.5.1</**version**>  
</**dependency**>

2、我们的应用程序也就是Apollo客户端依赖于AppId，Apollo Meta Server等环境信息来工作，需要做如下配置：

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；

1、Spring Boot application.properties

Apollo 支持通过Spring Boot的application.properties或bootstrap.properties文件配置：

app.id=YOUR-APP-ID

2、app.properties

在项目的classpath:/META-INF/app.properties文件存在，并且其中内容配置：

app.id=YOUR-APP-ID

Apollo Meta Server：元数据服务器；

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

1. 通过app.properties配置文件在classpath:/META-INF/app.properties指定apollo.meta=http://config-service-url

3、在main方法的类上加一个注解

@EnableApolloConfig *//开启apollo配置支持*

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；

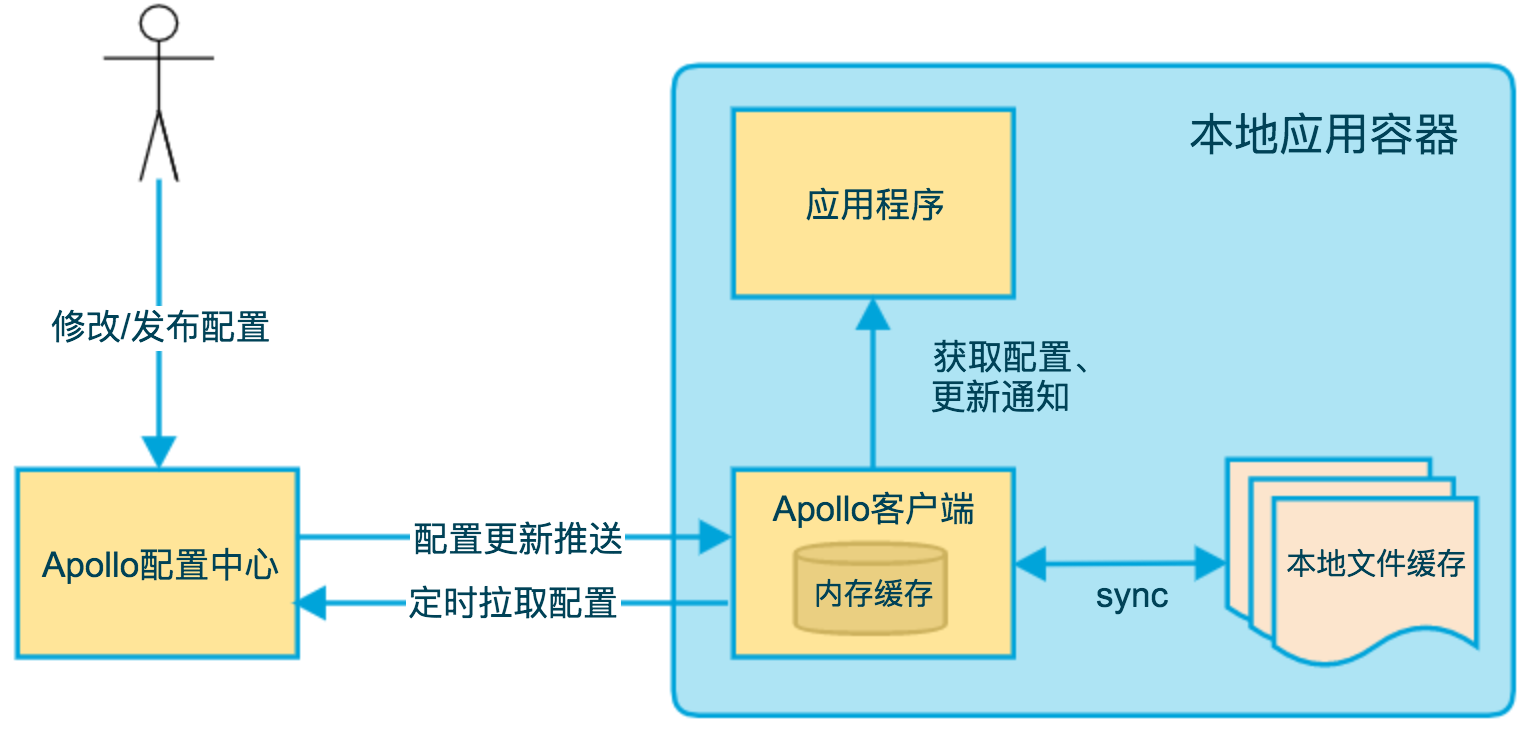
本地缓存路径默认位于以下路径，所以请确保/opt/data或C:\opt\data\目录存在，且应用有读写权限；

Mac/Linux: /opt/data/{appId}/config-cache

Windows: C:\opt\data\{appId}\config-cache

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

# 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；

3、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；

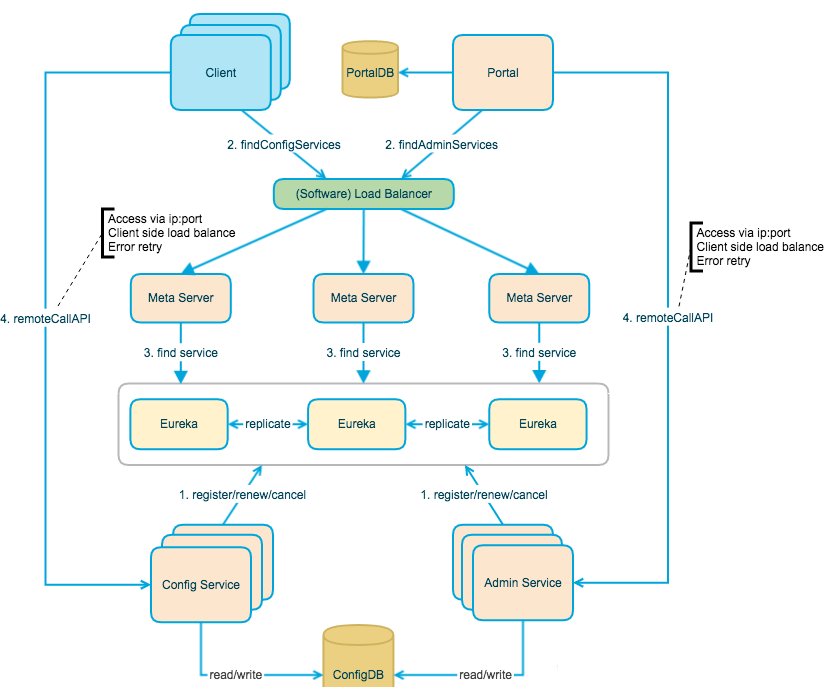
4、客户端从Apollo配置中心服务端获取到应用的最新配置后，会保存在内存中；

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；

# Apollo阿波罗配置中心设计



Apollo的总体设计，从下往上看分别是：

Config Service提供配置的读取、推送等功能，服务对象是Apollo客户端；

Admin Service提供配置的修改、发布等功能，服务对象是Apollo Portal（管理界面）；

Config Service和Admin Service都是多实例、无状态部署，所以需要将自己注册到Eureka中并保持心跳；

在Eureka之上我们架了一层Meta Server用于封装Eureka的服务发现接口；

Client通过域名访问Meta Server获取Config Service服务列表（IP+Port），而后直接通过IP+Port访问服务，同时在Client侧会做load balance、错误重试；

Portal通过域名访问Meta Server获取Admin Service服务列表（IP+Port），而后直接通过IP+Port访问服务，同时在Portal侧会做load balance、错误重试；

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；

**Apollo模块**

**1 Config Service**

提供配置获取接口

提供配置更新推送接口（基于Http long polling）

服务端使用Spring DeferredResult实现异步化，从而大大增加长连接数量

目前使用的tomcat embed默认配置是最多10000个连接（可以调整），使用了4C8G的虚拟机实测可以支撑10000个连接，所以满足需求（一个应用实例只会发起一个长连接）。

接口服务对象为Apollo客户端

**2 Admin Service**

提供配置管理接口

提供配置修改、发布等接口

接口服务对象为Portal

**3 Meta Server**

Portal通过域名访问Meta Server获取Admin Service服务列表（IP+Port）

Client通过域名访问Meta Server获取Config Service服务列表（IP+Port）

Meta Server从Eureka获取Config Service和Admin Service的服务信息，相当于是一个Eureka Client

增设一个Meta Server的角色主要是为了封装服务发现的细节，对Portal和Client而言，永远通过一个Http接口获取Admin Service和Config Service的服务信息，而不需要关心背后实际的服务注册和发现组件

Meta Server只是一个逻辑角色，在部署时和Config Service是在一个JVM进程中的，所以IP、端口和Config Service一致

**4 Eureka**

基于Eureka和Spring Cloud Netflix提供服务注册和发现

Config Service和Admin Service会向Eureka注册服务，并保持心跳

为了简单起见，目前Eureka在部署时和Config Service是在一个JVM进程中的（通过Spring Cloud Netflix）

**5 Portal**

提供Web界面供用户管理配置

通过Meta Server获取Admin Service服务列表（IP+Port），通过IP+Port访问服务

在Portal侧做load balance、错误重试

**6 Client**

Apollo提供的客户端程序，为应用提供配置获取、实时更新等功能

通过Meta Server获取Config Service服务列表（IP+Port），通过IP+Port访问服务

在Client侧做load balance、错误重试；