目录

[1、微服务概述 3](#_Toc535779894)

[1.1、是什么 3](#_Toc535779895)

[1.2、微服务与微服务架构 4](#_Toc535779896)

[1.3、微服务优缺点 4](#_Toc535779897)

[1.4、微服务技术栈 4](#_Toc535779898)

[1.5、为什么选择Spring Cloud作为微服务架构的实现 5](#_Toc535779899)

[2、Spring Cloud概述 5](#_Toc535779900)

[2.1、是什么 5](#_Toc535779901)

[2.2、Spring Cloud和Spring Boot的关系： 6](#_Toc535779902)

[2.3、Spring Cloud VS Dubbo 7](#_Toc535779903)

[2.4、能干什么 7](#_Toc535779904)

[2.5、去哪里学习 7](#_Toc535779905)

[3、工程实战 8](#_Toc535779906)

[3.1、父工程 8](#_Toc535779907)

[3.2、公共子模块Module 10](#_Toc535779908)

[3.3、部门微服务提供者Module 10](#_Toc535779909)

[3.4、部门微服务消费者Module（客户端） 12](#_Toc535779910)

[4、Eureka服务注册与发现 14](#_Toc535779911)

[4.1、是什么 14](#_Toc535779912)

[4.2、原理 14](#_Toc535779913)

[4.3、构建Eureka服务注册中心（示例） 16](#_Toc535779914)

[4.4、Eureka的自我保护机制： 17](#_Toc535779915)

[4.5、CAP和ACID 17](#_Toc535779916)

[4.6、作为服务注册中心，Eureka比Zookeeper好在哪里： 19](#_Toc535779917)

[5、Ribbon 客户端 负载均衡 20](#_Toc535779918)

[5.1、概述 20](#_Toc535779919)

[5.2、架构说明 20](#_Toc535779920)

[5.3、Ribbon核心组件IRule 21](#_Toc535779921)

[5.4、示例 22](#_Toc535779922)

[6、Feign 负载均衡 22](#_Toc535779923)

[6.1、概述 22](#_Toc535779924)

[6.2、示例 23](#_Toc535779925)

[8、Hystrix 断路器 24](#_Toc535779926)

[8.1、概述 24](#_Toc535779927)

[8.1.1、分布式系统面临的问题 25](#_Toc535779928)

[8.1.2、是什么 25](#_Toc535779929)

[8.1.3、能干嘛 26](#_Toc535779930)

[8.2、服务熔断 26](#_Toc535779931)

[8.3、服务降级 27](#_Toc535779932)

[8.4、服务监控hystrixDashboard 27](#_Toc535779933)

[8.5、示例 28](#_Toc535779934)

[9、Zuul路由网关 29](#_Toc535779935)

[9.1、概述 29](#_Toc535779936)

[9.2、路由基本配置 29](#_Toc535779937)

[9.3、路由访问映射规则 30](#_Toc535779938)

[10、SpringCloud Config 分布式配置中心 30](#_Toc535779939)

[10.1、概述 30](#_Toc535779940)

[10.2、分布式面临的问题 31](#_Toc535779941)

[10.3、是什么 31](#_Toc535779942)

[10.4、能干什么 31](#_Toc535779943)

[10.5、与GitHub整合 32](#_Toc535779944)

[10.6、SpringCloud Config 服务端与Github通信 32](#_Toc535779945)

[10.7、SpringCloud Config 客户端通过服务端获取Github上的配置 33](#_Toc535779946)

[10.8、示例 34](#_Toc535779947)

# 1、微服务概述

## 1.1、是什么

Martin Fowler

<https://martinfowler.cn/articles/microservices.html>

https://www.martinfowler.com/articles/microservices.html

The microservice archiitectural style is an approach to developing a single application

as a suite of small services,each running in its own process and communicating with

lightweight mechanisms often an HTTP resource API.These services are built around business capabilities

and independently deployable by fully automated deployment manchinery.There is a bare minimum of

centralized management of these services,which may be written in different programming languages and

use different data storage technologies.

微服务架构是一种架构模式。它提倡将单一应用程序划分成一组小的服务，服务之间互相协调，互相配合，为用户提供

最终的价值。每个服务运行在各自独立的进程中，服务与服务之间采用轻量级的通信机制互相协作

（通常是基于HTTP协议的RESTful API）。每个服务都围绕着具体的业务进行构建，并且能够被独立的部署到生产环境，

类生产环境等。另外，应当尽量避免统一的，集中式的服务管理机制，对具体的某一个服务而言，应根据业务上下文，选择

合适的语言，工具对其进行构建。

技术维度理解：

微服务化的核心就是将传统的一站式应用，根据业务拆分成一个个的服务，彻底的去耦合，每一个微服务提供单个业务功能的服务，一个服务做一件事，从技术角度看就是一种小而独立的处理过程，类似进程的概念，能够自行单独启动或销毁，拥有自己独立的数据库。

a single application --> small services -->

own process

## 1.2、微服务与微服务架构

微服务：

强调的是服务的大小，关注的是某一个点，是具体解决某一个问题/提供落地对应服务的一个服务应用，狭义的看，可以看做eclipse里面的一个个微服务工程/或者Module

## 1.3、微服务优缺点

优点：

* 每个服务足够内聚，聚焦一个指定的业务功能
* 开发简单，效率高，一个服务只干一件事
* 可以小团队开发，需要人少
* 服务之间松耦合，从开发到部署都是独立
* 可以使用不同语言
* 容易和第三方集成，部署方式灵活，可以使用持续集成工具：Jenkins\Hudson\bamboo
* 服务只关注业务逻辑代码，不会和HTML\CSS其他界面组件混合
* 有独立的存储能力，有自己的数据库实现

缺点：

* 要处理分布式系统的复杂性
* 运维难度，随着服务的增加而加大
* 系统部署依赖
* 服务间通信成本
* 数据一致性
* 系统集成测试
* 性能监控

## 1.4、微服务技术栈

|  |  |  |
| --- | --- | --- |
| **微服务技术** | **落地实现** | 备注 |
| 服务开发 | SpringBoot、Spring、SpringMVC |  |
| 服务配置与管理 | Netflix Archaius、Alibaba Diamond |  |
| 服务注册与发现 | Eureka、Consul、Zookeeper |  |
| 服务调用 | Rest、RPC、gRPC |  |
| 服务熔断器 | Hystrix、Envoy |  |
| 负载均衡 | Ribbon、Nginx |  |
| 服务接口调用（客户端调用服务的简化工具） | Feign |  |
| 消息队列 | Kafka、RabbitMQ、ActiveMQ |  |
| 服务配置中心管理 | SpringCloudConfig、Chef |  |
| 服务路由（API网关） | Zuul |  |
| 服务监控 | Zabbix、Nagios、Metrics、Spectator |  |

## 1.5、为什么选择Spring Cloud作为微服务架构的实现

* 框架成熟度
* 社区热度
* 可维护性
* 学习曲线

# 2、Spring Cloud概述

## 2.1、是什么

SpringCloud，基于SpringBoot提供了一套微服务解决方案，包括服务注册与发现，配置中心，全链路监控，

服务网关，负载均衡，熔断器等组件，除了基于Netflix的开源组件做高度抽象之外，还有一些选型中立的开源组件。

SpringCloud，利用SpingBoot的开发便利性巧妙地简化了分布式系统基础设施的开发，SpringCloud为开发人员提供

了快捷构建分布式系统的一些工具，包括配置管理，服务发现，断路器，路由，微代理，事件总线，全局锁，

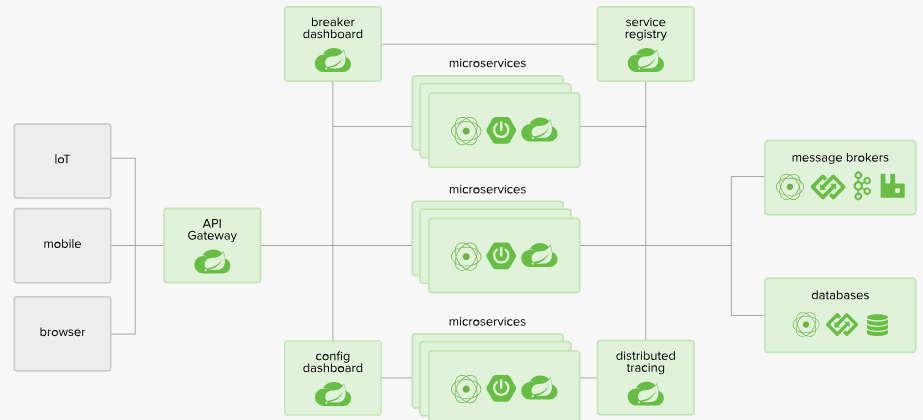
决策竞选，分布式会话等，他们可以用SB的开发风格做到一键启动和部署。

SC没有重复制作轮子，只是将当前开发的比较成熟，禁得起考验的服务框架组合起来，通过SB风格进行再封装，

屏蔽了复杂的配置和实现原理，最终给开发者一套简单易懂，易部署和易维护的分布式系统开发工具包。

官网：

<https://spring.io>



## 2.2、Spring Cloud和Spring Boot的关系：

SB专注于快速方便的开发单个服务。

SC关注全局的服务协调整理治理框架，它将SB开发的一个个服务整合并管理起来，为各个服务之间提供配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等集成服务。

SB可以独立于SC，但SC离不开SB

**目前成熟的互联网架构：分布式+服务治理（Dubbo）**



## 2.3、Spring Cloud VS Dubbo

<https://github.com/dubbo>

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

对比：

|  |  |  |
| --- | --- | --- |
|  | **Dubbo** | **Spring Cloud** |
| 服务注册中心 | Zookeeper | Spring Cloud Netflix Eureka |
| 服务调用方式 | RPC | REST API |
| 服务监控 | Dubbo-monitor | Spring Boot Admin |
| 断路器 | 不完善 | Netflix Hystrix |
| 服务网关 | 无 | Netflix Zuul |
| 分布式配置 | 无 | Spring Cloud Config |
| 服务跟踪 | 无 | Spring Cloud Sleuth |
| 消息总线 | 无 | Spring Cloud Bus |
| 数据流 | 无 | Spring Cloud Stream |
| 批量任务 | 无 | Spring Cloud Task |

Dubbo定位于一款RPC框架，SC是基于Http的REST方式，牺牲了服务调用的性能，但也避免了原生RPC带来的问题，

而且REST比RPC更为灵活，服务提供方和调用方的依赖只靠一纸契约，不存在代码级别的强依赖，这在强调快速

演化的微服务环境下，更为合适。

Dubbo的定位是一款RPC框架，而Spring Cloud目标是微服务架构下的一站式解决方案。

## 2.4、能干什么

Distributed/versioned configuration 分布式/版本控制配置

Service registration and discovery

Routing

Service-to-service calls（服务间的调用）

Load Banlancing

Circuit Breakers（断路器）

Distributed Messaging（分布式消息管理）

…

## 2.5、去哪里学习

Spring Cloud中文网站

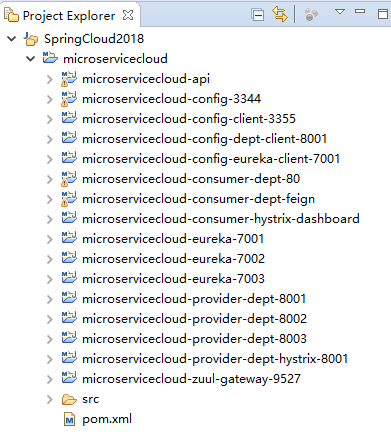
https://springcloud.cc/

Spring Cloud中文社区

http://www.springcloud.cn/

https://cloud.spring.io/spring-cloud-static/Dalston.SR5/single/spring-cloud.html

# 3、工程实战



## 3.1、父工程

New>maven project>packing:pom

**microservicecloud**

<groupId>com.leaf.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>microservicecloud-api</module>

<module>microservicecloud-provider-dept-8001</module>

<module>microservicecloud-consumer-dept-80</module>

<module>microservicecloud-eureka-7002</module>

<module>microservicecloud-eureka-7003</module>

<module>microservicecloud-provider-dept-8002</module>

<module>microservicecloud-provider-dept-8003</module>

<module>microservicecloud-consumer-dept-feign</module>

<module>microservicecloud-provider-dept-hystrix-8001</module>

<module>microservicecloud-consumer-hystrix-dashboard</module>

<module>microservicecloud-zuul-gateway-9527</module>

<module>microservicecloud-config-3344</module>

<module>microservicecloud-config-client-3355</module>

<module>microservicecloud-config-dept-client-8001</module>

<module>microservicecloud-config-eureka-client-7001</module>

</modules>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Camden.SR7</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>1.5.9.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<delimiters>

<delimit>$</delimit>

</delimiters>

</configuration>

</plugin>

info:

app.name: atguigu-microservicecloud

company.name: www.atguigu.com

build.artifactId: $project.artifactId$

build.version: $project.version$

## 3.2、公共子模块Module

**microservicecloud-api**

<parent><!-- 子类里面显示声明才能有明确的继承表现，无意外就是父类的默认版本否则自己定义 -->

<groupId>com.leaf.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-api</artifactId><!-- 当前Module我自己叫什么名字 -->

<dependencies><!-- 当前Module需要用到的jar包，按自己需求添加，如果父类已经包含了，可以不用写版本号 -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</dependency>

<!-- Feign -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

</dependencies>

## 3.3、部门微服务提供者Module

**microservicecloud-provider-dept-8001**

**@SpringBootApplication**

**@EnableEurekaClient //本服务启动后会自动注册进eureka服务中**

**@EnableDiscoveryClient //服务发现**

**public class DeptProvider8001\_App**

<parent>

<groupId>com.leaf.springcloud</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<!-- 引入自己定义的api通用包，可以使用Dept部门Entity -->

<dependency>

<groupId>com.leaf.springcloud</groupId>

<artifactId>microservicecloud-api</artifactId>

<version>${project.version}</version>

</dependency>

<!-- 将微服务provider侧注册进eureka -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

**<!-- actuator监控信息完善，保证被hystrix dashboard监控到 -->**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-actuator</artifactId>**

**</dependency>**

<!-- 修改后立即生效，热部署 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

eureka:

client: #客户端注册进eureka服务列表内

service-url:

#defaultZone: http://localhost:7001/eureka

defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

**instance:**

**instance-id: microservicecloud-dept8001 # 自定义服务名称**

**prefer-ip-address: true #访问路径可以显示IP地址**

info:

app.name: atguigu-microservicecloud

company.name: www.atguigu.com

build.artifactId: $project.artifactId$

build.version: $project.version$

## 3.4、部门微服务消费者Module（客户端）

**microservicecloud-consumer-dept-80**

80端口是为HTTP协议开放的，主要用于WWW(World Wide Web)传输信息的协议，浏览网页服务默认的端口都是80，

输网址的时候可以不输入。

RestTemplate提供了多种便捷访问远程Http服务的方法，是一种

简单便捷的访问restful服务模板类，是spring提供的用于访问Rest服务的

客户端模板工具类。

@SpringBootApplication

@EnableEurekaClient

//在启动该微服务的时候就能去加载我们的自定义Ribbon配置类，从而使配置生效

@RibbonClient(name="MICROSERVICECLOUD-DEPT",configuration=MySelfRule.class)

public class DeptConsumer80\_App

@Configuration

public class ConfigBean {

//boot -->spring

// applicationContext.xml --- @Configuration配置 ConfigBean = applicationContext.xml

@Bean

@LoadBalanced//Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端 负载均衡的工具。

public RestTemplate getRestTemplate()

{

return new RestTemplate();

}

import org.springframework.web.client.RestTemplate;

@RestController

public class DeptController\_Consumer

/\*\*

\* 使用 使用restTemplate访问restful接口非常的简单粗暴无脑。 (url, requestMap,

\* ResponseBean.class)这三个参数分别代表 REST请求地址、请求参数、HTTP响应转换被转换成的对象类型。

\*/

@Autowired

private RestTemplate restTemplate;

@RequestMapping(value = "/consumer/dept/add")

public boolean add(Dept dept) {

return restTemplate.postForObject(REST\_URL\_PREFIX + "/dept/add", dept, Boolean.class);

}

// private static final String REST\_URL\_PREFIX = "http://localhost:8001";

private static final String REST\_URL\_PREFIX = "http://MICROSERVICECLOUD-DEPT";

<!-- Ribbon相关 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

eureka:

client:

register-with-eureka: false

service-url:

defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

# 4、Eureka服务注册与发现

## 4.1、是什么

Netflix/Eureka

Github

Netflix设计Eureka的时候遵循AP原则。

Eureka:

是Netflixd的一个子模块,也是核心模块之一。

是一个基于REST的服务，用于定位服务，以实现云端中间层服务发现和

故障转移。服务注册与发现对于微服务架构来说非常重要，有了服务发现与

注册，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务

调用的配置文件了。

功能类似Dubbo的注册中心，比如Zookeeper。

## 4.2、原理

基本架构：

http://dict.cn/

Spring Cloud封装了Netflix公司研发的Eureka模块

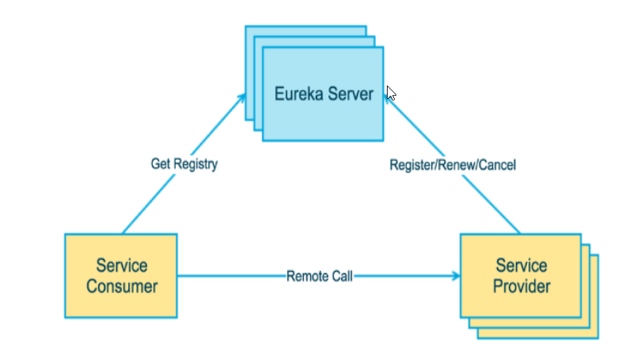
来实现服务注册和发现。

Eureka采用了C-S的设计架构。

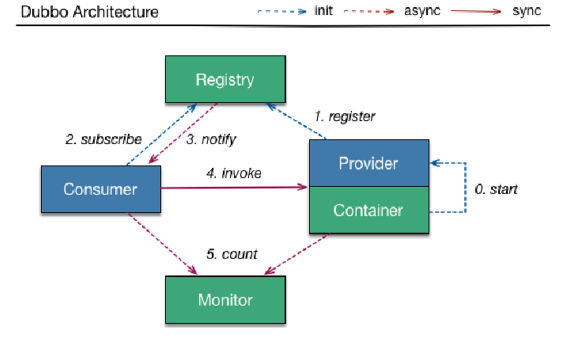
Eureka Server 作为服务注册功能的服务器，它是服务注册中心。

系统中的其他微服务，使用Eureka的客户端连接到Eureka Server并维持心跳连接。这样系统的维护人员可以通过Eureka Server来监控系统中各个服务是否正常运行。Spring Cloud的一些其他模块（比如Zuul）就可以通过Eureka Server来发现系统中的其他服务，并执行相关的逻辑。

Eureka结构图：



Dubbo架构图：



Eureka包含两个组件：

- Eureka Server

- Eureka Client

Eureka Server提供服务注册服务

各个节点启动后，会在Eureka Server中注册，这样Eureka Server的服务注册表中将会存储所有可用节点的信息，服务节点的信息可以在界面中直观的看到。

Eureka Client是一个java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的，使用轮询（round-robin）负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳（默认周期是30秒）。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server将会从服务注册表中把这个服务节点移除（默认90s）。

Eureka 3大角色：

- Eureka Server 提供服务注册与发现

- Server Provider 服务提供方将自身服务注册到Eureka，从而使服务消费方能够找到

- Server Consumer 服务消费方从Eureka获取注册服务列表，从而能够消费服务

## 4.3、构建Eureka服务注册中心（示例）

**microservicecloud-eureka-7001**

**@SpringBootApplication**

**@EnableEurekaServer // EurekaServer服务器端启动类,接受其它微服务注册进来**

<!--eureka-server服务端 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

eureka:

instance:

hostname: eureka7001.com #eureka服务端的实例名称

client:

register-with-eureka: false #false表示不向注册中心注册自己。

fetch-registry: false #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务

service-url:

#defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址（单机）。

defaultZone: http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/

server:

enable-self-preservation: true #启用自我保护模式

## 4.4、Eureka的自我保护机制：

某时刻某一个微服务不可用了，eureka不会立刻清理，依旧会对该微服务的信息进行保存。

默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务的心跳，EurekaServer将会注销该服务（默认是90s）。但是当网络分区故障时，服务与EurekaServer之间无法正常通信，以上行为会变得非常危险-因为服务本省其实是健康的，此时不应该注销这个服务。Eureka通过“自我保护模式”来解决这个问题，当EurekaServer节点在短时间内丢失了过多客户端，那么这个节点就会进入自我保护模式。一旦进入该模式，EurekaServer就会保护服务注册表的信息，不再删除服务注册表中的数据。当网络恢复后，EurekaServer自动退出自我保护模式。

**在自我保护模式中，ES会保护服务注册表中的信息，不在注销任务服务实例，当他收到的心跳树重新恢复到阈值以上时，该ES节点就会自动退出自我保护模式。它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。一句话：好死不如赖活着。**

综上，自我保护模式是一种应对网络异常的安全保护措施，它的架构哲学是宁可同时保留所有的服务（健康和不健康的），也不盲目注销任务可能健康的服务。使用自我保护模式，可以让Eureka集群更加健壮，稳定。

在SpringCloud中，可以使用eureka.server.enable-self-preservation= fasle禁用自我保护模式

**C:\Windows\System32\drivers\etc\hosts**

## 4.5、CAP和ACID

数据库遵循的原则：

RDBMS (mysql/oracle/sqlserver) ---> ACID

NOSQL (redis/mongdb) ---> CAP

A Atomicity 原子性

C Consistency 一致性

I Isolation 独立性

D Durability 持久性

C Consistency 强一致性

A Availability可用性

P Partition tolerance 分区容错性

**关系型数据库遵循ACID原则：**

事务在英文为Transaction，有如下4个特性：

1、A(Atomicity) 原子性

在事务里的所有操作要么全部做完，要不不做，事务成功的条件是事务里的所有操作都成功，只要有一个失败，整个事务就失败，需要回滚。

2、C(Consistency) 一致性

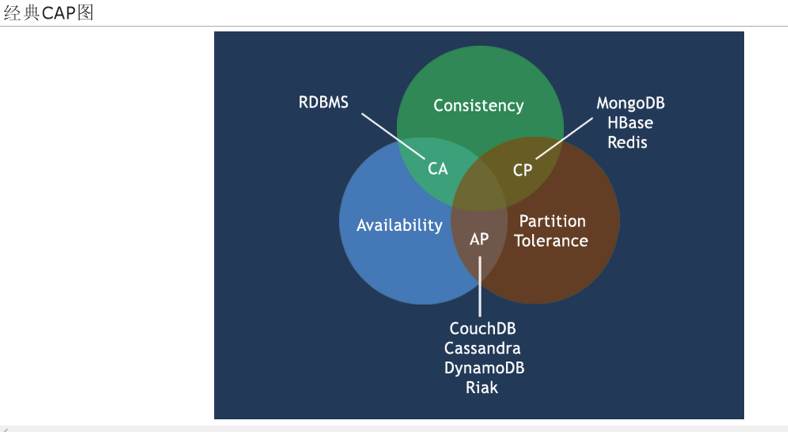
数据库一直处于一致的状态，事务的运行不会改变数据库原本的一致性约束。

3、I(Isolation) 独立性

指并发的事务之间不会互相影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务没提交，它访问的数据就不受未提交事务的影响。

4、D(Durability) 持久性

指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。



任何一个分布式系统只能同时满足CAP的其中2种：

CAP最多只能同时满足2个：

**CAP理论的核心是**：一个分布式的系统不可能同时很好的满足一致性，可用性，分区容错性这3点。

因此，根据CAP理论将NOSQL数据库分成了满足CA原则、满足CP原则和满足AP原则三大类：

CA --单点集群，满足一致性，可用性的系统，通常在可扩展上不太强

CP --满足一致性，分区容错的系统，通常性能不高

AP --满足可用性，分区容错性的系统，通常对一致性要求低一些

CAP理论就是说在分布式存储系统中，最多只能实现上面的2点。

由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容错性是我们必须需要实现的。

所以只能在一致性和可用性之间权衡，没有NOSQL系统能同时保证这2点。

CA 传统Oracle数据库

AP 大多数网站架构的选择

CP Redis,Mongodb

注意：分布式架构的时候必须做出取舍

## 4.6、作为服务注册中心，Eureka比Zookeeper好在哪里：

著名的CAP理论指出，一个分布式系统不可能同时满足C\A\P。由于分区容错性在分布式系统中是必须要保证的，因此只能在A\C之间进行权衡。

**Zookeeper保证CP**

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对于可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举时间太长，30-120s，且选举期间整个zk集群都是不可用的。这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

**Eureka保证AP**

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作。剩余节点依然可以提供注册和查询服务。而在Eureka的客户端向某个Eureka注册或者如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用（保证可用性），只不过查到的信息可能不是最新的（不保证强一致性）。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，会出现以下情况：

1、Eureka不在从注册列表中移除因为长时间没收到心跳而应该过期的服务

2、Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其他节点上（即保证当前节点依然可用）

3、当网络稳定后，当前实例新的注册信息就会被同步到其他节点中

因此，Eureka可以很好的应对网络故障导致部分节点失去联系的情况，而不会想zookeeper那样使整个注册服务瘫痪

# 5、Ribbon 客户端 负载均衡

## 5.1、概述

Netflix / ribbon

Spring Cloud Ribbon是基于Netflix Ribbon实现的一套**客户端负载均衡**的工具

简单的说，Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法，将Netflix的中间层连接在一起。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。简单说，就是在配置文件中列出Load Balance (LB) 后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。

LB: 负载均衡 Load Balance

在微服务或分布式集群中经常使用的一种应用。

简单的说就是将用户的请求平摊的分配到多个服务上，从而达到系统的HA（High Avaliability）。

常见的负载均衡软件：Nginx LVS 硬件F5等

相应的中间件，例如Dubbo和SC均提供了负载均衡，**SC的LB算法可以自定义。**

**LB 有2大方向：**

-- 集中式LB 偏硬件

在服务的消费方和提供方之间使用独立的LB设施（可以是硬件，如F5，也可以是软件，如Nginx），由该设施负责把访问请求通过某种策略转发至服务的提供方；

-- 进程内LB 偏软件

将LB逻辑集成到消费方，将消费方从服务注册中心获知有哪些地址可以用，然后自己再从这些地址中选择出一个合适的服务器。

Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

## 5.2、架构说明

Ribbon在工作的时候分2步：

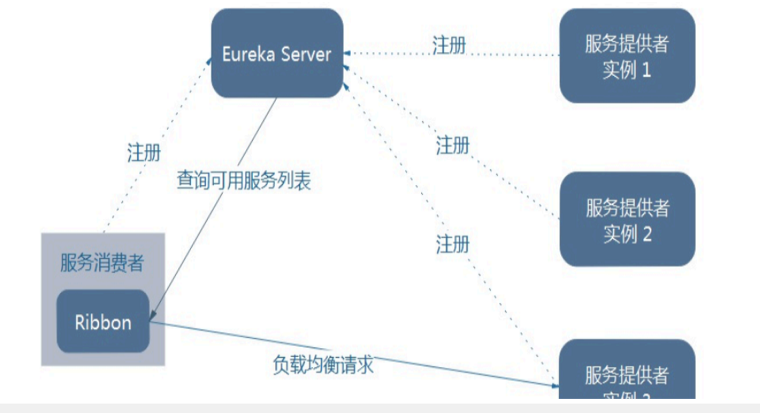
1.先选择Eureka Server，优先选择在同一区域内负载较少的server

2.再根据用户指定的策略，比如轮询，随机和根据响应时间加权，从server服务注册列表中选取一个服务地址访问

Ribbon其实就是一个软负载均衡的客户端组件，它可以和其他所需请求的客户端结合使用，和Eureka结合只是其中一个实例。

instance-id: microservicecloud-dept8002 #自定义服务名称信息 有空格和没空格

**默认均衡算法是轮询请求（顺序轮询）**



## 5.3、Ribbon核心组件IRule

IRule: 根据特定的算法从服务列表中选取一个要访问的服务：

RoundRobinRule 轮询 默认

RandomRule 随机

AvailabilityFilteringRule 会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，还有并发的连接数量超过阈值的服务，然后对剩余的服务列表按照轮询策略访问

WeightedResponseTimeRule

根据平均响应时间计算所有的服务权重，响应时间越快权重越大被选中的概率越大，刚启动时如果统计信息不足，则使用RoundRobinRule策略，等统计信息充足，会切换到该规则

RetryRule

先按照RoundRobinRule的侧率获取服务，如果获取失败在指定的时间内进行重试，获取可用的服务

BestAvailableRule

会先过滤由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务

ZoneAvoidanceRule

默认规则，复合判断server所在的区域的性能和server的可用性选择服务器

## 5.4、示例

@EnableEurekaClient

//在启动该微服务的时候就能去加载我们的自定义Ribbon配置类，从而使配置生效

@RibbonClient(name="MICROSERVICECLOUD-DEPT",configuration=MySelfRule.class)

@Bean

@LoadBalanced//Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端 负载均衡的工具。

public RestTemplate getRestTemplate()

{

return new RestTemplate();

}

@Bean

public IRule myRule() {

return new RoundRobinRule(); //默认轮询算法

// return new RandomRule();// 达到的目的，用我们重新选择的随机算法替代默认的轮询。

// return new RetryRule(); //

}

官方文档明确给出了警告：

这个自定义的配置类不能放在@ComponentScan所扫描的当前包下以及子包下，否则我们自定义的这个配置类就会被所有的Ribbon客户端所共享，也就是说我们达不到特殊化定制的目的。

# 6、Feign 负载均衡

## 6.1、概述

https://github.com/OpenFeign/feign

OpenFeign / feign

Feign：

是一个声明式WebService客户端。使用Feign能编写Web Service客户端更加简单，它的使用方法是定义一个接口，然后在上面添加注解，同时也支持JAX-RS标准的注解。Feign也支持可插拔式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持了Spring MVC标准注解和HttpMessageConverters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

Feign是一个声明式的Web服务客户端，使得编写Web服务客户端变得非常容易。

只需要创建一个接口，然后在上面添加注解即可。

Feign能干什么：

Feign旨在使编写Java Http客户端变得更容易

前面在使用Ribbon+RestTemplate时，利用RestTemplate对HTTP请求的封装处理，形成了一套模板化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务进行封装一些客户端类来包装这些依赖服务的调用。所以，Feign在此基础上做了进一步的封装，由他来帮助我们定义和实现依赖服务接口的定义。在Feign的实现下，我们只需要创建一个接口并使用注解的方式来配置它（以前是在Dao接口上标注Mapper注解，现在是一个微服务接口上标注一个Feign注解即可），即可完成对服务提供方的接口绑定，简化了使用Spring Cloud Ribbon时，自动封装服务调用客户端的开发量。

Feign集成了Ribbon

利用Ribbon维护了服务提供方的服务列表，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，通过Feign只需要定义服务绑定接口且以声明式的方法，优雅而简单的实现了服务调用。

Feign通过接口的方法调用Rest服务（之前是通过Ribbon+RestTemplate），该请求发送给Eureka服务器，通过Feign直接找到服务接口，由于在进行服务接口调用的时候融合了Ribbon技术，所以也支持负载均衡。

## 6.2、示例

**microservicecloud-consumer-dept-feign**

@EnableFeignClients(basePackages= {"com.leaf.springcloud"})

@ComponentScan("com.leaf.springcloud")

public class DeptConsumer80\_Feign\_App

@FeignClient(value = "MICROSERVICECLOUD-DEPT",fallbackFactory=DeptClientServiceFallbackFactory.class)

public interface DeptClientService {

@RequestMapping(value = "/dept/get/{id}", method = RequestMethod.GET)

public Dept get(@PathVariable("id") long id);

@RequestMapping(value = "/dept/list", method = RequestMethod.GET)

public List<Dept> list();

@RequestMapping(value = "/dept/add", method = RequestMethod.POST)

public boolean add(Dept dept);

}

<!--Feign -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<!-- Ribbon相关 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

**公共API mvn clean mvn install**

# 8、Hystrix 断路器

## 8.1、概述

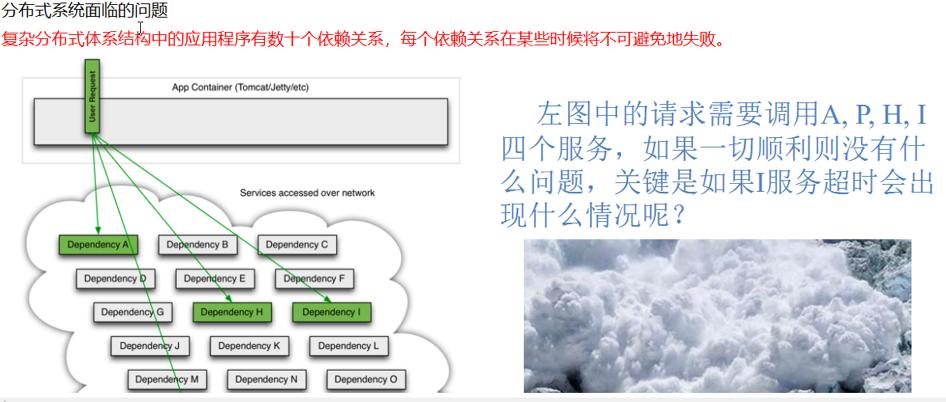
### 8.1.1、分布式系统面临的问题

复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免的失败。

---》**服务雪崩**：

多个微服务之间的调用的时候,假设服务A调用服务B和服务C，服务B和服务C又调用其他的微服务，这就是所谓的**“扇出”**。如果扇出的链路上某个服务的调用响应时间过长或者不可用，对服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的**“雪崩效应”**。

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几秒内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障。这些都表示需要对故障和延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或系统。



### 8.1.2、是什么

Hystrix是一个**处理分布式系统的延迟和容错**的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时，异常等，Hystrix能够保证在一个依赖出问题的情况下，**不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性**。

“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），**向调用方返回一个符合预期的，可处理的备选响应（Fallback），而不是长时间的等待或者抛出调用方无法处理的异常**，这样就保证了服务调用方的线程不会被长时间，不必要的占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

### 8.1.3、能干嘛

服务降级

服务熔断

服务限流

接近实时的监控

。。。

## 8.2、服务熔断

熔断机制是应对雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回“错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在Spring Cloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间的调用情况，当失败的调用到一定的阈值，缺省是5s内20次调用失败就会启动熔断机制。熔断机制的注解是

@HystrixCommand

[服务熔断]

一般是某个服务故障或者异常引起的，类似现实世界中的"保险丝"，当某个异常条件被处罚，直接熔断整个服务，而不是一直等待到超时。

<!-- hystrix -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

@EnableEurekaClient //本服务启动后会自动注册进eureka服务中

@EnableDiscoveryClient //服务发现

@EnableCircuitBreaker//对hystrixR熔断机制的支持

@RequestMapping(value = "/dept/get/{id}", method = RequestMethod.GET)

//一旦调用服务方法失败并抛出了错误信息后，会自动调用@HystrixCommand标注好的fallbackMethod调用类中的指定方法

@HystrixCommand(fallbackMethod = "processHystrix\_Get")

public Dept get(@PathVariable("id") Long id)

{

Dept dept = this.service.get(id);

if (null == dept) {

throw new RuntimeException("该ID：" + id + "没有没有对应的信息");

}

return dept;

}

public Dept processHystrix\_Get(@PathVariable("id") Long id)

{

return new Dept().setDeptno(id).setDname("该ID：" + id + "没有没有对应的信息,null--@HystrixCommand")

.setDb\_source("no this database in MySQL");

}

## 8.3、服务降级

[服务降级]

所谓降级，一般是从整体负荷考虑。就是当某个服务熔断之后，服务器将不再被调用，此时客户端可以自己准备一个本地的fallback回调，返回一个缺省值。这样做，虽然服务水平下降，但好歹可用，比直接挂掉要强。

服务降级：

整体资源快不够了，忍痛将某些服务先关掉，待度过难关，再开启回来。

服务的降级处理是在客户端实现完成的，与服务端没有关系

服务端有异常时，客户端不可用也会获得提示信息而不会挂起耗死服务器。

@FeignClient(value = "MICROSERVICECLOUD-DEPT",fallbackFactory=DeptClientServiceFallbackFactory.class)

public interface DeptClientService

@Component // 不要忘记添加，不要忘记添加

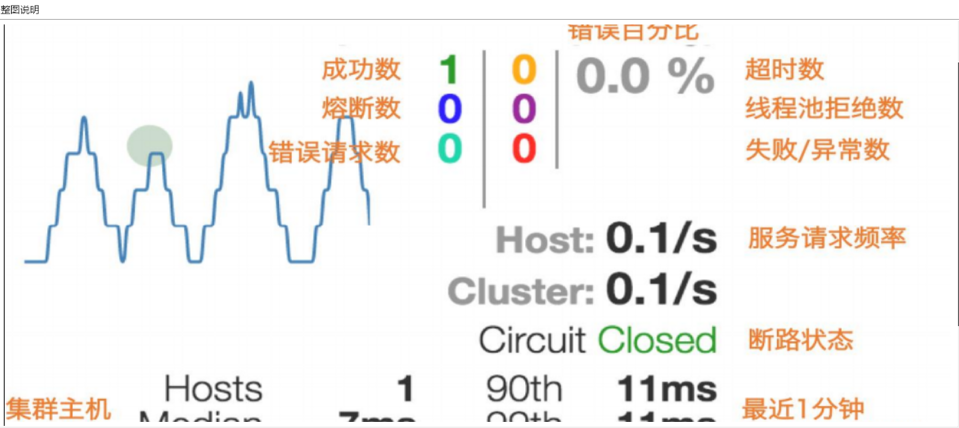
public class DeptClientServiceFallbackFactory implements FallbackFactory<DeptClientService>{

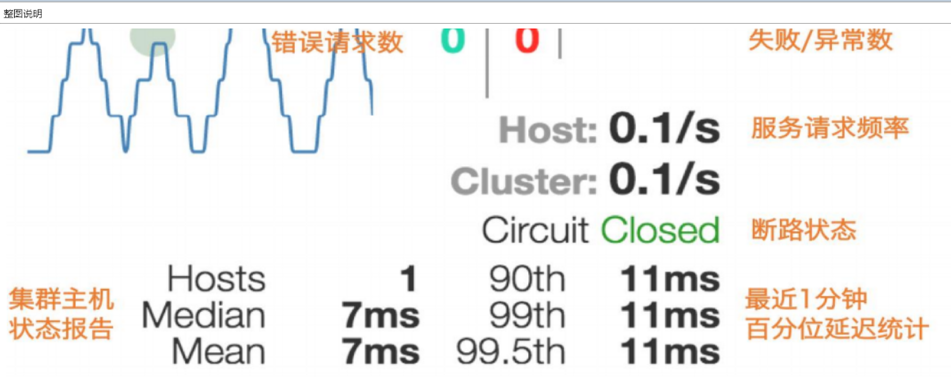
## 8.4、服务监控hystrixDashboard

除了隔离依赖服务的调用以外，Hystrix还提供了准实时的调用监控（Hystrix Dashboard）,Hystrix会持续记录所有通过Hystrix发起的请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求多少成功，失败等。Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。SpringCloud也提供了Hystrix Dashboard的整合，对监控内容转化为可视化界面。

localhost:9001/hystrix

localhost:8001/hystrix.stream





## 8.5、示例

**microservicecloud-consumer-hystrix-dashboard**

<!-- hystrix和 hystrix-dashboard相关 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>

</dependency>

@EnableHystrixDashboard

public class DeptConsumer\_DashBoard\_App

# 9、Zuul路由网关

## 9.1、概述

github.com Netflix / zuul

zuul 路由网关

**Zuul包含了对请求的路由和过滤2个最主要的功能**：

其中路由功能负责将外部请求转发到具体的微服务实例上，是显现外部访问统一入口的基础

而过滤器功能则负责对请求的处理过程进行干预，是实现请求效验，服务聚合等功能的基础。

Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的消息，也即也厚访问微服务都是通过Zuul跳转后获得。

注意：Zuul服务最终还是会注册进Eureka

**提供 = 代理 + 路由 + 过滤 3大功能**

## 9.2、路由基本配置

microservicecloud-zuul-gateway-9527

<!-- zuul路由网关 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

@EnableZuulProxy

public class Zuul\_9527\_StartSpringCloudApp

## 9.3、路由访问映射规则

zuul:

ignored-services: microservicecloud-dept

prefix: /lf

#ignored-services: "\*"

routes:

mydept.serviceId: microservicecloud-dept

mydept.path: /wl/\*\*

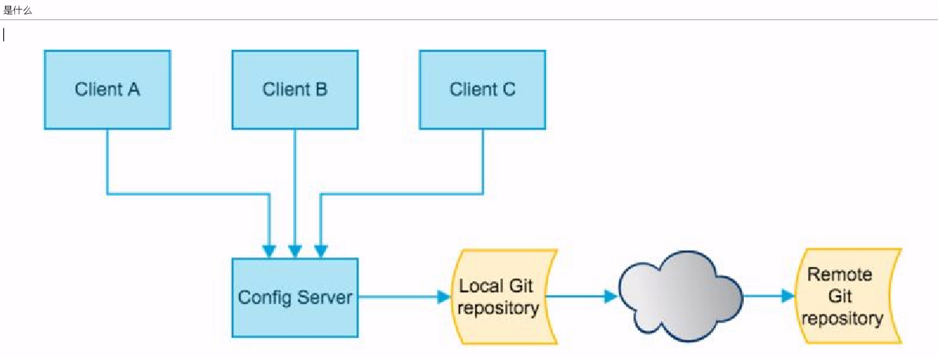
测试：

http://localhost:8001/dept/list

http://myzuul.com:9527/microservicecloud-dept/dept/list

# 10、SpringCloud Config 分布式配置中心

## 10.1、概述



## 10.2、分布式面临的问题

Spring Cloud Config 分布式配置中心

微服务意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务。由于每个服务都需要必要的配置信息才能运行，所以一套集中式的，动态的配置管理设施是必不可少的。SpringCloud提供了ConfigServer来解决这个问题，我们每一个微服务自己带着一个application.properties/yml。微服务多了对运维造成很大的困难。。。

## 10.3、是什么

**Spring Cloud Config为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。**

Spring Cloud Config **分为服务端和客户端2部分**。

服务端也称为**分布式配置中心**，他**是一个独立的微服务应用**，用来连接配置服务器并为客户端提供获取配置信息，加密、解密信息等访问接口。

客户端则是通过制定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。

配置服务器默认采用git来存储配置信息，这样有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容。

## 10.4、能干什么

* 集中管理配置文件
* 不同环境不同配置，动态化的配置更新，分环境部署、dev/test/prod/beta/release
* 运行期间动态调整配置，不在需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取自己的信息
* 当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置
* 将配置信息以REST接口的形式暴露

## 10.5、与GitHub整合

由于SpringCloud Config默认使用git来存储配置文件（也支持SVN和本地文件），但推荐Git，而且使用的是http、https访问的形式。

github新建repository

本地新建application.yml

git add .

git commit -m "init file"

git push origin master

**application.yml 是用户级的资源配置项**

**bootstrap.yml 是系统级的，优先级更高**

Spring Cloud会创建一个“Bootstrap Context”，作为Spring应用的“Application Context”的父上下文。初始化的时候，“Bootstrap Context”负责从外部源加载配置属性并解析配置。这2个上下文共享一个从外部获取的“Environment”。“Bootstrap”属性有高优先级，默认情况下，他们不会被本地配置覆盖。“Bootstrap Context”和“Application Context”有不同的约定。

新增一个“bootstrap.yml”文件，保证“Bootstrap Context”和“Application Context”配置的分离

Nginx： 反向代理 + 动静分离 + 负载均衡

## 10.6、SpringCloud Config 服务端与Github通信

**microservicecloud-config-3344**

<!-- springCloud Config -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<!-- 避免Config的Git插件报错：org/eclipse/jgit/api/TransportConfigCallback -->

<dependency>

<groupId>org.eclipse.jgit</groupId>

<artifactId>org.eclipse.jgit</artifactId>

<version>4.10.0.201712302008-r</version>

</dependency>

@EnableConfigServer

public class Config\_3344\_StartSpringCloudApp

spring:

application:

name: springcloud-config-demo

cloud:

config:

server:

git:

uri: git@github.com:xywanhh/springcloud-config-demo.git #GitHub上面的git仓库名字

## 10.7、SpringCloud Config 客户端通过服务端获取Github上的配置

HyperSnap

**microservicecloud-config-client-3355**

<!-- SpringCloud Config客户端 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

bootstrap.yml

spring:

cloud:

config:

name: microservicecloud-config-client #需要从github上读取的资源名称，注意没有yml后缀名

profile: test #本次访问的配置项

label: master

uri: http://config-3344.com:3344 #本微服务启动后先去找3344号服务，通过SpringCloudConfig获取GitHub的服务地址

spring:

cloud:

config:

name: microservicecloud-config-client #需要从github上读取的资源名称，注意没有yml后缀名

profile: test #本次访问的配置项

label: master

uri: http://config-3344.com:3344 #本微服务启动后先去找3344号服务，通过SpringCloudConfig获取GitHub的服务地址

## 10.8、示例

**microservicecloud-config-dept-client-8001**

<!-- SpringCloudConfig配置 -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

bootstrap.yml

spring:

cloud:

config:

name: microservicecloud-config-dept-client #需要从github上读取的资源名称，注意没有yml后缀名

#profile配置是什么就取什么配置dev or test

profile: dev

#profile: test

label: master

uri: http://config-3344.com:3344 #SpringCloudConfig获取的服务地址

@EnableEurekaClient //本服务启动后会自动注册进eureka服务中

@EnableDiscoveryClient //服务发现

public class DeptProvider8001\_Config\_App