### 走一遍

systemctl stop firewalld.service #停止firewall

systemctl disable firewalld.service #禁止firewall开机启动

firewall-cmd --state #查看默认防火墙状态（关闭后显示notrunning，开启后显示running）

# eurekaserver的注解:

@SpringBootApplication

@EnableEurekaServer

# eurekaserver的配置文件:

eureka:

client:

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

service-url:

defaultZone: http://localhost:8080/eureka/

server:

enable-self-preservation: false #默认为true,我们改为false,关闭服务端和客户端之间的keepalived的检测,如果安全则会发出红色文字的安全警告(在开发环境可以这么设置,在生产环境一定不要这么设置!)

# eurekaclient的注解:

@SpringBootApplication

@EnableDiscoveryClient // 客户端服务发现

# eurekaclient的配置文件:

eureka:

client:

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

service-url:

defaultZone: http://localhost:8080/eureka/

instance:

hostname: clientName #自定义链接(在注册的客户端,点击跳转后的url栏显示的服务名称,一般用不到)

spring:

application:

name: client #注册中心的服务的名称,不写则为unknow

# eureka高可用之eurekaserver互相注册

jvm设置区分端口号:

-Dserver.port=8761

-Dserver.port=8762

怎么做呢?

将注册服务的地址互相写对方的地址:

8761的服务这么写:

eureka:

client:

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

service-url:

defaultZone: http://localhost:8762/eureka/

8762的服务这么写:

eureka:

client:

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

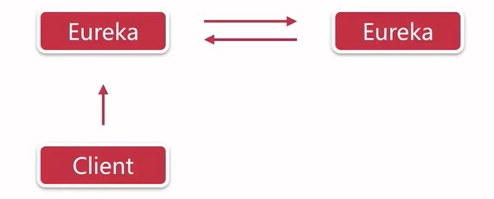
service-url:

defaultZone: http://localhost:8761/eureka/

此时,如果**启动一个client端,且注册地址为8761**,

则你会发现,在两个注册中心都会发现这个客户端client;

什么原因呢?



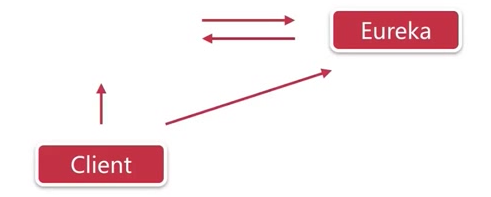
如果此时eurekaserver8761挂掉了呢?

我们会发现,在eureka8762的服务上依然可以看到client,但是不能正常使用!

如果我们重启client和eurekaserver8762(注意此时eurekaserver8761已经挂掉!),我们会发现eurekaserver8762上已经没有client了

这里的服务并不是高可用的架构!

那如何做到一个高可用的架构呢?



就是将eureka的地址都写在client中就可以了!

**client 端like this:客户端都把服务写进去**

eureka:

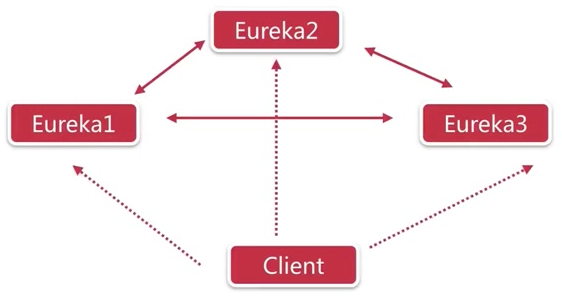
client: #客户端注册进eureka服务列表内

service-url:

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

**server端不互相注册自己,但是都把其他服务写进去**

**然后逐渐演化:**



# eureka总结

@enableEurekaServer @EnableEurekaClient

心跳检测,健康检查,负载均衡等功能

Eureka的高可用,生产上建议至少两台以上

分布式系统中,服务注册中心是最重要的基础部分

# 分布式系统中为什么需要服务发现?





# 服务发现的两种方式:

客户端发现(需要自己实现一套逻辑)(Eureka是客户端发现)

服务端发现(需要代理)(Nginx,Dubbo(zookeeper),kubernates)

# 微服务的特点:

-不同语言

-不同类型的数据库

-REST(springcloud)

# 微服务拆分?如何拆分

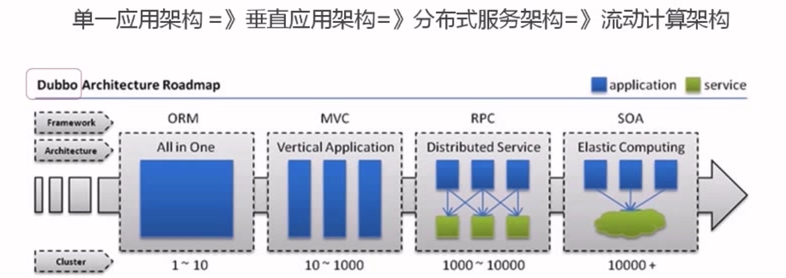
起点:

-既有架构的形态

终点:

-好的架构不是设计出来的,而是进化而来的

-一直在演进ing



微服务与SOA之间只差一个ESB(企业服务总线)

# 适合上微服务吗?

业务形态不适合的

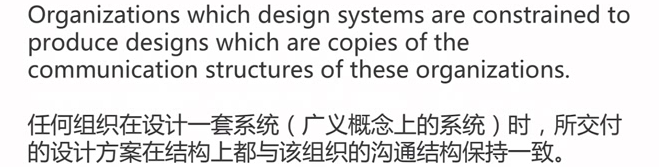
系统中包含很多很多强事务场景的

业务相对稳定,迭代周期长

访问压力不大,可用性要求不高

...

康威定律



# 应用间通信之restTemplate

springcloud 服务间两种restful调用方式

restTemplate(http客户端)

1:第一种方式:restTemplate

RestTemplate restTemplate = new RestTemplate();

String response = restTemplate.getForObject("http://localhost:8080/msg", String.class);

缺点:

url写死

多个服务地址时是个考验

2:第二种方式:LoadBalancerClient通过应用名获取url,然后再使用restTemplate

ServiceInstance serviceInstance = loadBalancerClient.choose("服务名");

String url = String.format("http://%s:%s", serviceInstance.getHost(), serviceInstance.getPort(), "/msg");

String response = restTemplate.getForObject(url, String.class);

3:第三种方式: @LoadBalanced利用注解可在restTemplate使用应用名字

String response = restTemplate.getForObject("http://product/msg", String.class);

# 客户端负载均衡器:Ribbon

RestTemplate

Fegin

Zuul

Ribbon实现软负载均衡的核心有三点:

服务发现

服务选择规则

服务监听

Ribbon主要组件:

ServerList

Irule

ServerListFilter

Ribbon默认的是RoundRabin轮询

也可以自定义Ribbon的规则:

在配置文件中:

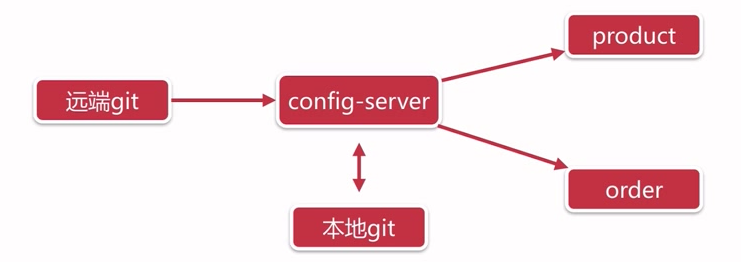
B.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule

# 应用间通信之Fegin

<https://www.jianshu.com/p/a961bce65448>

@FeignClient(name = "product")

# 统一配置中心



# zipkin

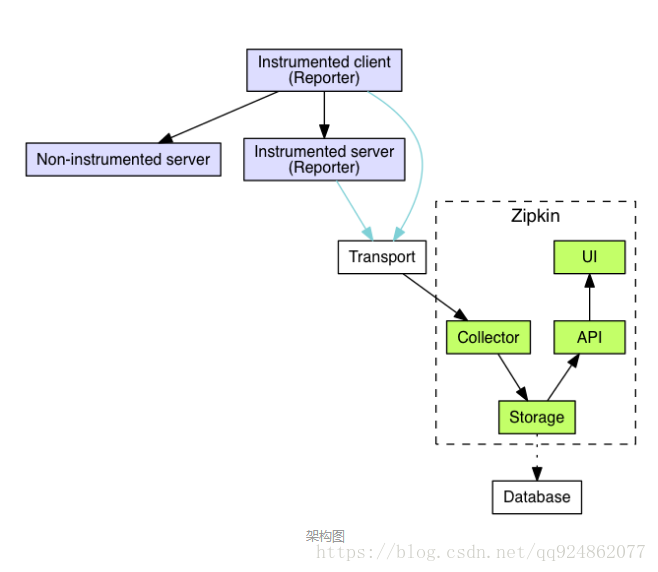
Zipkin是根据Google Dapper的论文设计的全链路监控系统，由Twitter公司开发。

集成步骤:

引入依赖

启动ZipKin Server

配置参数



# 分布式追踪系统

核心步骤:

数据采集

数据存储

查询展示

opentracing

优势:

来自大名鼎鼎的CNCF

Dapper(Google) : 各 tracer 的基础

StackDriver Trace (Google)

Zipkin(twitter)

Appdash(golang)

鹰眼(taobao)

谛听(盘古，阿里云云产品使用的Trace系统)

云图(蚂蚁Trace系统)

sTrace(神马)

X-ray(aws)

# 服务容器化的步骤

1:写服务程序

2:写dockerfile

3:build镜像

4:run容器

如果想让别人也运行自己的容器,可以将它传到云上

# Rancher

Rancher是一个开源的全栈化企业级容器管理平台，通过极简的操作体验和强大完善的功能