Docker+K8S+DevOps 微服务架构师

学神 IT 教育：从零基础到实战，从入门到精通！

联系方式：

学神 IT 教育官方网站: http://www.xuegod.cn

学神 K8S 精英学习 11 群 QQ 群: 957231097

学习顾问：小语老师　　学习顾问：边边老师　　学神微信公众号

微信扫码添加学习顾问微信，同时扫码关注学神公众号了解最新行业动态，获取更多学习资料及答疑就业服务！

第 6 四章　在 k8s 平台部署 SpringCloud 电商项目-模拟京东在线购物

本节所讲内容：

6.1 大型电商平台微服务架构解读
6.2 单体架构 vs 微服务解读
6.3 哪些项目适合微服务部署?
6.4 SpringCloud 概述
6.5 SpringCloud 组件介绍
实战：在 k8s 平台部署 SpringCloud 框架的电商项目：部署 eureka 服务

实验环境：k8s 集群：　k8s 的控制节点
ip：192.168.1.63
主机名：xuegod63
配置：6vCPU/6Gi 内存
k8s 的工作节点：
ip：192.168.1.64
主机名：xuegod64
配置：12vCPU/8Gi 内存
harbor 机器：
ip：192.168.1.62
主机名：harbor
配置：4vCPU/6Gi 内存

Mysql 服务：mysql 部署在 xuegod63 上，暴露的端口是 3306

下面实验依赖的服务有 harbor、Ingress controller、mysql，上一节课已经安装成功了

互动：为什么要要将 SpringCloud 项目迁移到 K8S 平台?

SpringCloud 只能用在 SpringBoot 的 java 环境中，而 kubernetes 可以适用于任何开发语言，只要能被放进 docker 的应用，都可以在 kubernetes 上运行，而且更轻量，更简单。

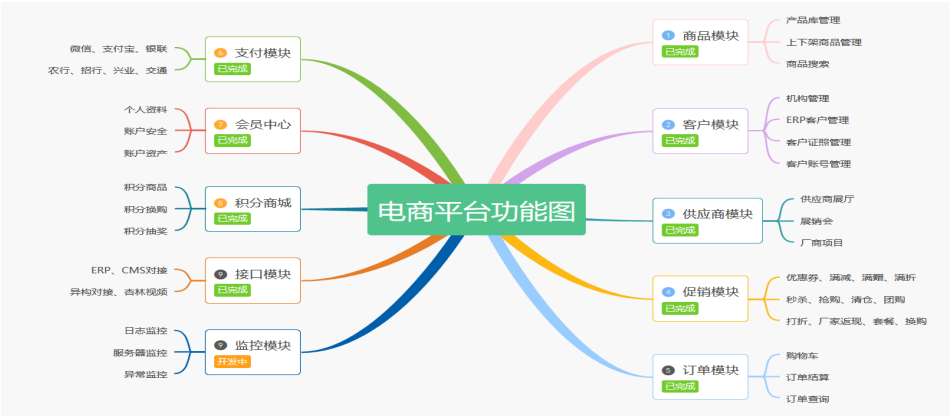
每个微服务可以部署多个，没有多少依赖，并且有负载均衡能力，比如一个服务部署一个副本或 5 个副本，通过 k8s 可以更好的去扩展我们的应用。

SpringCloud 很多功能都跟 kubernetes 重合，比如服务发现，负载均衡，配置管理，所以如果把 SpringCloud 部署到 k8s，那么很多功能可以直接使用 k8s 原生的，减少复杂度。

SpringCloud 容易上手，是对开发者比较友好的平台；Kubernetes 是可以实现 DevOps 流程的，SpringCloud 和 kubernetes 各有优点，只有结合起来，才能发挥更大的作用，达到最佳的效果。

6.1　大型电商平台微服务架构解读

6.1.1 基于 springcloud 的电商平台功能图

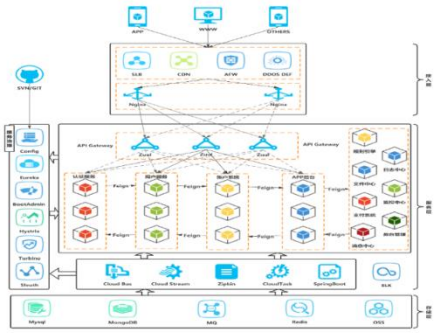


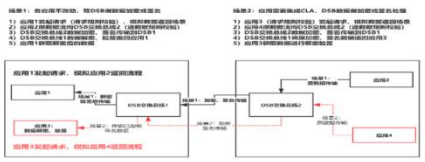
Xmind

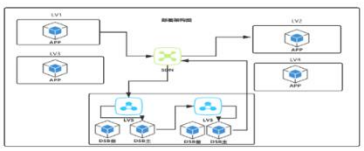
Visio

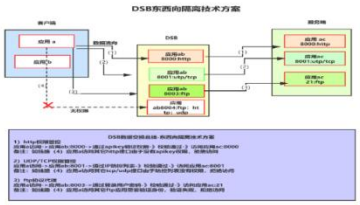
https://www.processon.com

6.1.2 基于 springcloud 的电商平台架构图









6.2 单体架构 vs 微服务解读

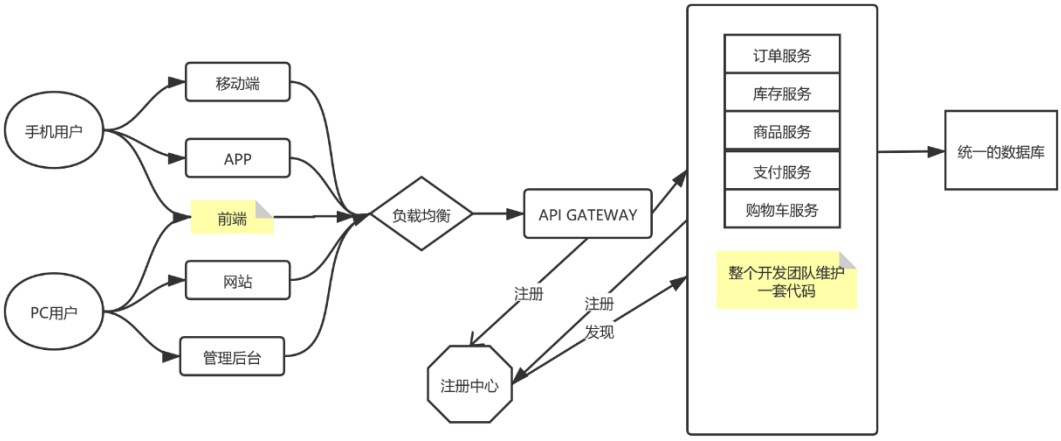
6.2.1 什么是微服务?

微服务是用于构建应用程序的架构风格，一个大的系统可由一个或者多个微服务组成，微服务架构可将应用拆分成多个核心功能，每个功能都被称为一项服务，可以单独构建和部署，这意味着各项服务在工作和出现故障的时候不会相互影响。

6.2.2 为什么要用微服务?

1、单体架构扩展性差、维护成本高、不可靠

互动：什么是单体架构?



在单体架构下修改代码，需要把整个代码重新编译，重新部署，这个时间周期会很长；

单体架构下的所有代码模块都耦合在一起，代码量大，维护困难，想要更新一个模块的代码，也可能会影响其他模块，不能很好的定制化代码。

所有模块都用同一个数据库，存储方式比较单一。

2、微服务中可以有 java 编写、有 Python 编写的，他们都是靠 restful 架构风格统一成一个系统的，所以微服务本身与具体技术无关、扩展性强

6.2.3 微服务特性

1）灵活部署、独立扩展

传统的单体架构是以整个系统为单位进行部署，而微服务则是以每一个独立组件（例如订单服务，商品服务）为单位进行部署。

2）资源的有效隔离

每一个微服务拥有独立的数据源，假如微服务 A 想要读写微服务 B 的数据库，只能调用微服务 B 对外暴露的接口来完成。这样有效避免了服务之间争用数据库和缓存资源所带来的问题。另外微服务各模块部署在 k8s 中，可以进行 CPU、内存等资源的限制和隔离。

3）高度可扩展性

随着某些服务模块的不断扩展，可以跨多个服务器和基础架构进行部署，充分满足业务需求。

4）易于部署

相对于传统的单体式应用，基于微服务的应用更加模块化且小巧，且易于部署。

5）服务组件化

在微服务架构中，需要我们对服务进行组件化分解，服务是一种进程外的组件，它通过 HTTP 等通信协议进行协作，而不是像传统组件那样镶入式的方式协同工作，每一个服务都独立开发、部署、可以有效避免一个服务的修改引起整个系统的重新部署。

6）去中心化治理

在整个微服务架构，通过采用轻量级的契约定义接口，使得我们对服务本身的具体技术平台不再那么敏感，这样整个微服务架构系统中的各个组件就能针对不同的业务特点选择不同的技术平台。

7）容错设计

在微服务架构中，快速检测出故障源并尽可能地自动恢复服务是必须被设计考虑的，通常我们都希望在每个服务中实现监控和日志记录。比如对服务状态、断路器状态、吞吐量、网络延迟等关键数据进行可视化展示。

8）技术栈不受限

在微服务架构中，可以结合项目业务及团队的特点，合理地选择技术栈。

9）局部修改容易部署

单体应用只要有修改，就得重新部署整个应用，微服务解决了这样的问题。

10）易于开发和维护

一个微服务只会关注一个特定的业务功能，所以它业务清晰，代码量较少。

6.3 哪些项目适合微服务部署?

在复杂度比较低的项目中，单体架构就可以满足需求，而且部署效率也会比较高，在复杂度比较高的项目中，单体架构就不能满足了，需要进行微服务化。

微服务可以按照业务功能本身的独立性来划分，如果系统提供的业务是非常底层的，如：操作系统内核、存储系统、网络系统、数据库系统等，这类系统都偏底层，功能和功能之间有着紧密的配合关系，如果强制拆分为较小的服务单元，会让集成工作量急剧上升，并且这种人为的切割无法带来业务上的真正的隔离，所以无法做到独立部署和运行，也就不适合做成微服务了。

那到底什么样的项目适合微服务呢?

1. 业务并发量大，项目复杂，访问流量高，为了将来更好的扩展，随时对代码更新维护，可以使用微服务

2. 代码依赖程度高，想要解耦合，交给多个开发团队维护

3. 业务初期，服务器数量少，可以使用微服务，能有效节省资源。

4. 从思想上： 对未来有清晰的认识，对技术更新要保持着一种自信，超前思维，知道这个东西在将来肯定会发展起来。

这就告诉了我们一个道理，在学习技术的时候，适合自己的才是最好的，比方说很多人说我们公司单体架构用的也挺好的啊，为什么还要用微服务，其实他们再用单体可能适合他们业务需求，但是我们公司可能业务规模大，项目复杂，我就想要用微服务，或者我们在未来上有更大的远见，那我也会选择用微服务，不要说看别人用，我也用，而是我用是符合我们实际需求的，一切脱离实际业务的微服务都是耍流氓。

6.3.1 使用微服务需要考虑的问题

1、统一的配置管理中心

服务拆分以后，服务的数量非常多，如果所有的配置都以配置文件的方式放在应用本地的话，非常难以管理，可以想象当有几百上千个进程中有一个配置出现了问题，是很难将它找出来的，因而需要有统一的配置中心，来管理所有的配置，进行统一的配置下发。

在微服务中，配置往往分为几类，一类是几乎不变的配置，这种配置可以直接打在容器镜像里面，第二类是启动时就会确定的配置，这种配置往往通过环境变量，在容器启动的时候传进去，第三类就是统一的配置，需要通过配置中心进行下发，例如在大促的情况下，有些功能需要降级，哪些功能可以降级，哪些功能不能降级，都可以在配置文件中统一配置。

2、全链路监控

1）系统和应用的监控

监控系统和服务的健康状态和性能瓶颈，当系统出现异常的时候，监控系统可以配合告警系统，及时地发现，通知，干预，从而保障系统的顺利运行。

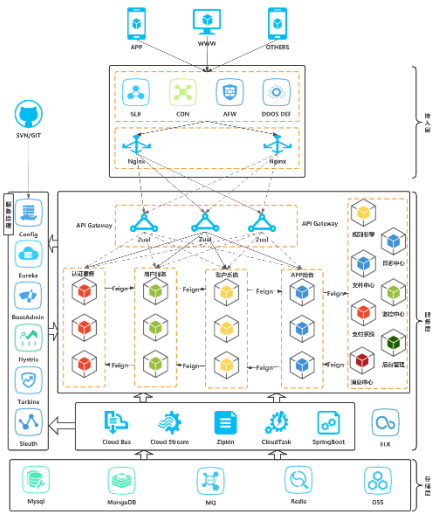
2）调用关系的监控

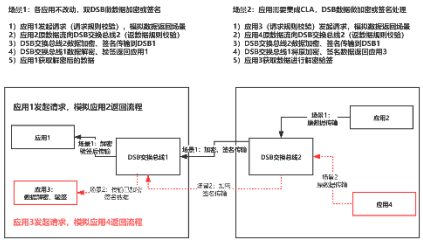
对代码调用关系进行监控

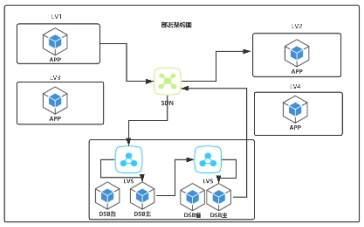
3、日志收集

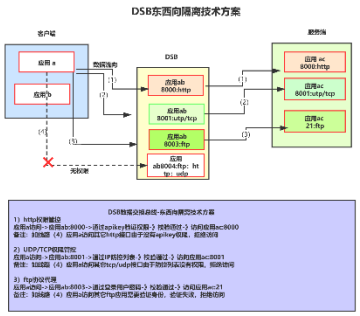
业务层面、代码层面、系统层面

6.4 SpringCloud 概述









6.4.1 SpringCloud 是什么?

官方解释：

官网： https://spring.io/projects/spring-cloud/

SpringCloud 是一系列框架的有序集合。它利用 SpringBoot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 SpringBoot 的开发风格做到一键启动和部署。SpringCloud 并没有重复制造轮子，它只是将各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过 SpringBoot 风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

6.4.2 SpringCloud 和 SpringBoot 什么关系?

SpringBoot 专注于快速方便的开发单个个体微服务。

SpringCloud 是关注全局的微服务协调整理治理框架，它将 SpringBoot 开发的一个个单体微服务整合并管理起来，SpringBoot 可以离开 SpringCloud 独立开发项目，但是 SpringCloud 离不开 SpringBoot，属于依赖关系。

6.4.3 SpringCloud 优缺点

1）SpringCloud 来源于 Spring，质量、稳定性、持续性都可以得到保证。

SpirngCloud 以 SpringBoot 为基础开发框架，可以给开发者大量的微服务开发经验，例如，只要极少量的标签，你就可以创建一个配置服务器，再加一些标签，你就可以得到一个客户端库来配置你的服务，更加便于业务落地。

2）SpringCloud 是 Java 领域最适合做微服务的框架，对 Java 开发者来说就很容易开发。

3）耦合度低，不影响其他模块

4）多个开发团队可以并行开发项目，提高开发效率

5）直接写自己的代码即可，然后暴露接口，通过组件进行服务通信。

缺点：

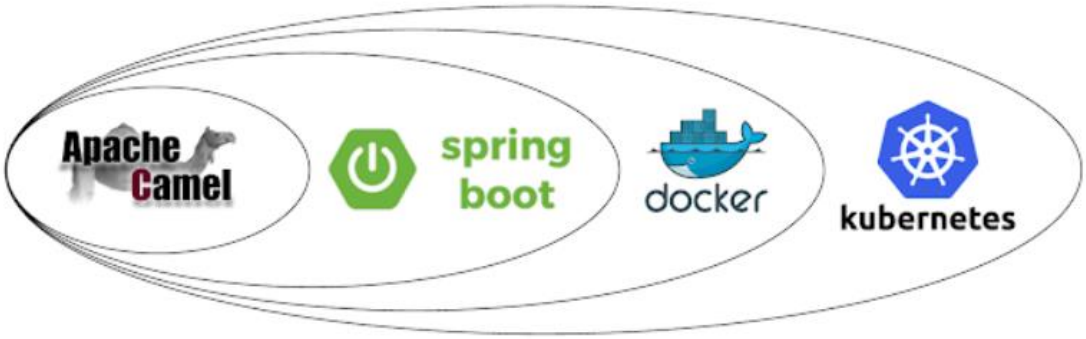
只能针对 Java 开发

部署麻烦、组件多

每个微服务都可以用一个数据库，导致数据管理复杂

一套完整的微服务包括自动化部署，调度，资源管理，进程隔离，自愈，构建流水线等功能，单靠 SpringCloud 是无法实现的，所以 SpringCloud+k8s 才是最好的方案

6.4.4 为何要将 SpringCloud 项目部署到 k8s 平台?



SpringCloud 只能用在 SpringBoot 的 java 环境中，而 kubernetes 可以适用于任何开发语言，只要能被放进 docker 的应用，都可以在 kubernetes 上运行，而且更轻量，更简单。

每个微服务可以部署多个，没有多少依赖，并且有负载均衡能力，比如一个服务部署一个副本或 5 个副本，通过 k8s 可以更好的去扩展我们的应用。

Spring 提供应用的打包，Docker 和 Kubernetes 提供部署和调度。Spring 通过 Hystrix 线程池提供应用内的隔离，而 Kubernetes 通过资源，进程和命名空间来提供隔离。Spring 为每个微服务提供健康终端，而 Kubernetes 执行健康检查，且把流量导到健康服务。Spring 外部化配置并更新它们，而 Kubernetes 分发配置到每个微服务。

SpringCloud 很多功能都跟 kubernetes 重合，比如服务发现，负载均衡，配置管理，所以如果把 SpringCloud 部署到 k8s，那么很多功能可以直接使用 k8s 原生的，减少复杂度。

SpringCloud 容易上手，是对开发者比较友好的平台；Kubernetes 是可以实现 DevOps 流程的，SpringCloud 和 kubernetes 各有优点，只有结合起来，才能发挥更大的作用，达到最佳的效果。

6.4.5 SpringCloud 项目部署到 k8s 的流程

制作镜像--->控制管理 pod--->暴露应用--->对外发布应用--->数据持久化--->日志/监控

1.制作镜像： 应用程序、运行环境、文件系统

2.控制器管理 pod：deployment 无状态部署、statefulset 有状态部署、Daemonset 守护进程部署、job & cronjob 批处理

3.暴露应用：服务发现、负载均衡

4.对外发布应用：service、Ingress HTTP/HTTPS 访问

5.pod 数据持久化：分布式存储-ceph 和 gluster

6.日志/监控：efk、prometheus、pinpoint 等

6.5 SpringCloud 组件介绍

6.5.1 服务发现与注册组件 Eureka

Eureka 是 Netflix 开发的服务发现框架， SpringCloud 将它集成在自己的子项目 spring-cloud-netflix 中，以实现 SpringCloud 中服务发现和注册功能。Eureka 包含两个组件：Eureka Server 和 Eureka Client。

互动 1：Netflix 是什么?

Netflix 在 SpringCloud 项目中占着重要的作用，Netflix 公司提供了包括 Eureka、Hystrix、Zuul、Archaius 等在内的很多组件，在微服务架构中至关重要。

互动 2： 举个例子服务发现与注册

我们在买车的时候，需要找中介，如果不找中介，我们自己去找厂商或者个人车主，这是很麻烦的，也很浪费时间，所以为了方便，我们一般去找中介公司，把我们的需求说出来，他们就会按需给我们推荐车型，我们相当于微服务架构中的消费者 Consumer，中介相当于微服务架构中的提供者 Provider，Consumer 需要调用 Provider 提供的一些服务，就像是我们要买的车一样。

#Eureka 服务组成

Eureka Server

Eureka Server 提供服务注册中心，各个节点启动后，会将自己的 IP 和端口等网络信息注册到 Eureka Server 中，这样 Eureka Server 服务注册表中将会存储所有可用服务节点的信息，在 Eureka 的图形化界面可以看到所有注册的节点信息。

Eureka Client

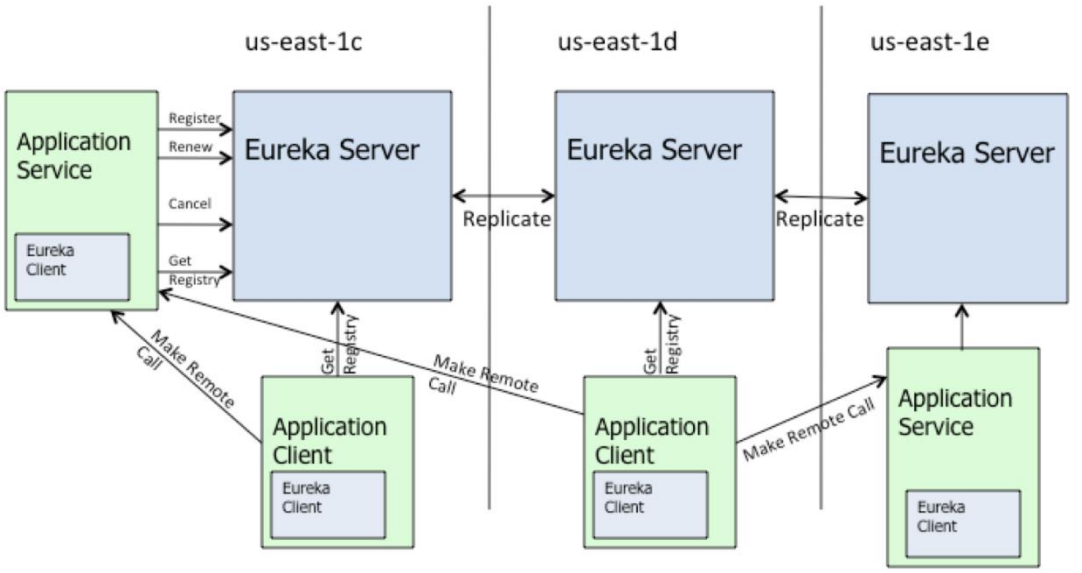
Eureka Client 是一个 java 客户端，在应用启动后，Eureka 客户端将会向 Eureka Server 端发送心跳，默认周期是 30s，如果 Eureka Server 在多个心跳周期内没有接收到某个节点的心跳，Eureka Server 将会从服务注册表中把这个服务节点移除(默认 90 秒)。

Eureka Client 分为两个角色，分别是 Application Service 和 Application Client

Application Service 是服务提供方，是注册到 Eureka Server 中的服务。

Application Client 是服务消费方，通过 Eureka Server 发现其他服务并消费。

#Eureka 架构原理



Register(服务注册)：当 Eureka 客户端向 Eureka Server 注册时，会把自己的 IP、端口、运行状况等信息注册给 Eureka Server。

Renew(服务续约)：Eureka 客户端会每隔 30s 发送一次心跳来续约，通过续约来告诉 Eureka Server 自己正常，没有出现问题。正常情况下，如果 Eureka Server 在 90 秒没有收到 Eureka 客户的续约，它会将实例从其注册表中删除。

Cancel(服务下线)：Eureka 客户端在程序关闭时向 Eureka 服务器发送取消请求。 发送请求后，该客户端实例信息将从服务器的实例注册表中删除，防止 consumer 调用到不存在的服务。该下线请求不会自动完成，它需要调用以下内容：DiscoveryManager.getInstance().shutdownComponent();

Get Registry(获取服务注册列表)：获取其他服务列表。

Replicate(集群中数据同步)：eureka 集群中的数据复制与同步。

Make Remote Call(远程调用)：完成服务的远程调用。

6.5.2 客户端负载均衡之 Ribbon

#Ribbon 简介

Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡器，主要提供客户侧的软件负载均衡算法，运行在消费者端。客户端负载均衡是当浏览器向后台发出请求的时候，客户端会向 Eureka Server 读取注册到服务器的可用服务信息列表，然后根据设定的负载均衡策略，抉择出向哪台服务器发送请求。在客户端就进行负载均衡算法分配。Ribbon 客户端组件提供一系列完善的配置选项，比如连接超时、重试、重试算法等。下面是用到的

一些负载均衡策略：

随机策略---随机选择 server

轮询策略---轮询选择， 轮询 index，选择 index 对应位置的 Server

重试策略--在一个配置时间段内当选择 Server 不成功，则一直尝试使用 subRule 的方式选择一个可用的 server

最低并发策略--逐个考察 server，如果 server 断路器打开，则忽略，再选择其中并发链接最低的 server

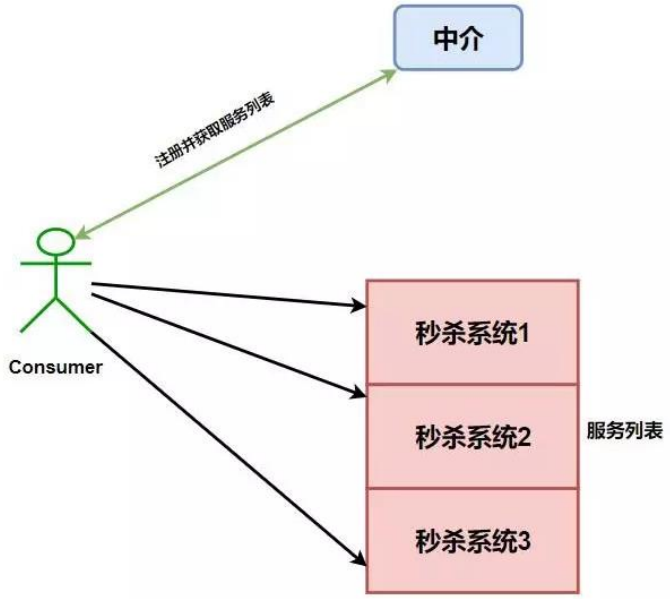
可用过滤策略--过滤掉一直失败并被标记为 circuit tripped 的 server，过滤掉那些高并发链接的 server（active connections 超过配置的阈值）或者使用一个 AvailabilityPredicate 来包含过滤 server 的逻辑，其实就就是检查 status 里记录的各个 Server 的运行状态；

响应时间加权重策略--根据 server 的响应时间分配权重，响应时间越长，权重越低，被选择到的概率也就越低。响应时间越短，权重越高，被选中的概率越高，这个策略很贴切，综合了各种因素，比如：网络，磁盘，io 等，都直接影响响应时间；

区域权重策略--综合判断 server 所在区域的性能，和 server 的可用性，轮询选择 server 并且判断一个 AWS Zone 的运行性能是否可用，剔除不可用的 Zone 中的所有 server。

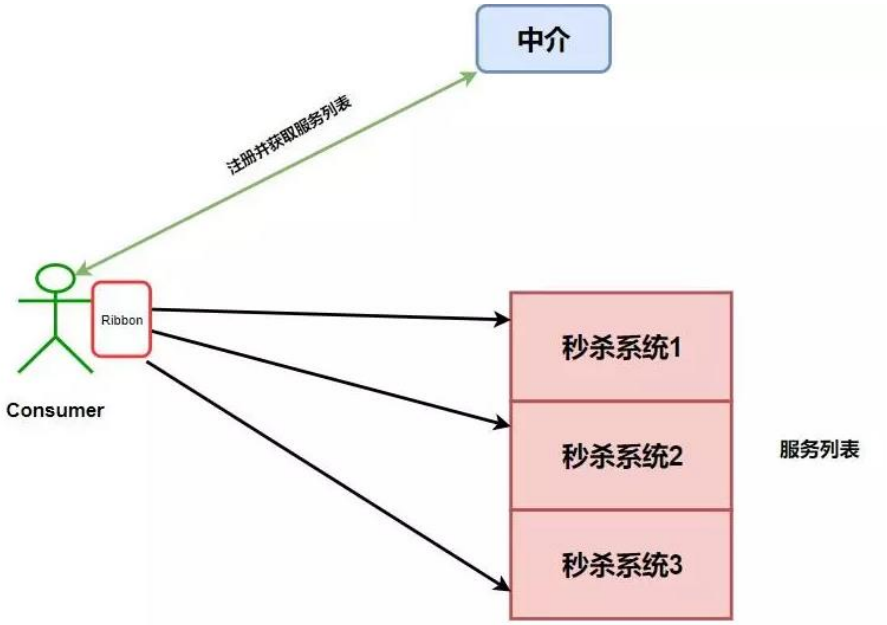
互动： 举个列子说明 ribbon

比如我们设计了一个秒杀系统，但是为了整个系统的高可用，我们需要将这个系统做一个集群，而这个时候我们消费者就可以拥有多个秒杀系统的调用途径了，如下图。



如果这个时候我们没有进行一些均衡操作，如果我们对秒杀系统 1 进行大量的调用，而另外两个基本不请求，就会导致秒杀系统 1 崩溃，而另外两个就变成了傀儡，那么我们为什么还要做集群，我们高可用体现的意义又在哪呢?

所以 Ribbon 出现了，注意我们上面加粗的几个字——运行在消费者端。指的是，Ribbon 是运行在消费者端的负载均衡器，如下图。



其工作原理就是 Consumer 端获取到了所有的服务列表之后，在其内部使用负载均衡算法，进行对多个系统的调用。

#Ribbon 的功能

易于与服务发现组件（比如 Eureka）集成

使用 Archaius 完成运行时配置

使用 JMX 暴露运维指标，使用 Servo 发布

多种可插拔的序列化选择

异步和批处理操作

自动 SLA 框架

系统管理/指标控制台

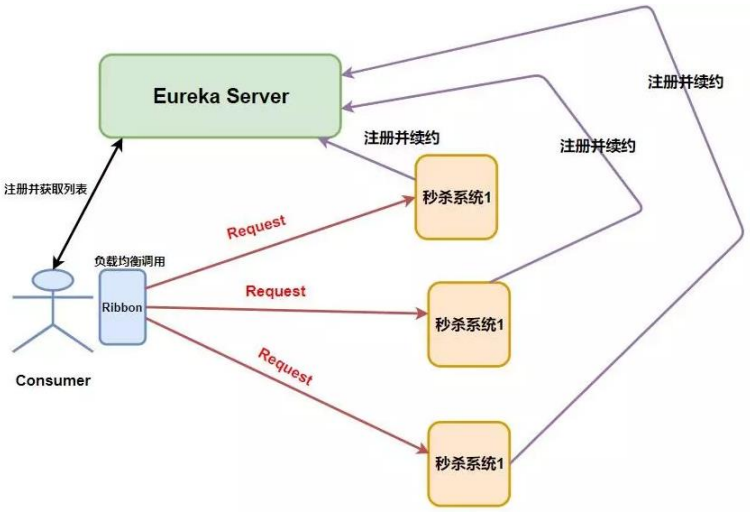
#Ribbon 和 nginx 对比分析

区别：

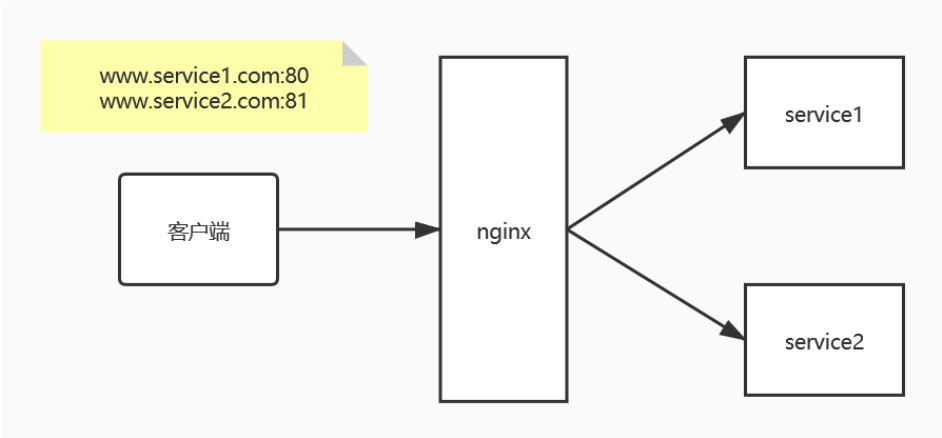
Ribbon 实现的是客户端负载均衡，它可以在客户端经过一系列算法来均衡调用服务。Ribbon 工作时分两步：

第一步：从 Eureka Server 中获取服务注册信息列表，它优先选择在同一个 Zone 且负载较少的 Server。

第二步：根据用户指定的策略，在从 Server 取到的服务注册列表中选择一个地址，其中 Ribbon 提供了多种策略，例如轮询、随机等。



Nginx 是服务器端负载均衡，所有请求统一交给 nginx，由 nginx 实现负载均衡请求转发，属于服务器端负载均衡。



6.5.3 服务网关 Zuul

Zuul 是 SpringCloud 中的微服务网关，首先是一个微服务。也是会在 Eureka 注册中心中进行服务的注册和发现。也是一个网关，请求应该通过 Zuul 来进行路由。Zuul 网关不是必要的，是推荐使用的。

互动：网关是什么?

是一个网络整体系统中的前置门户入口。请求首先通过网关，进行路径的路由，定位到具体的服务节点上。

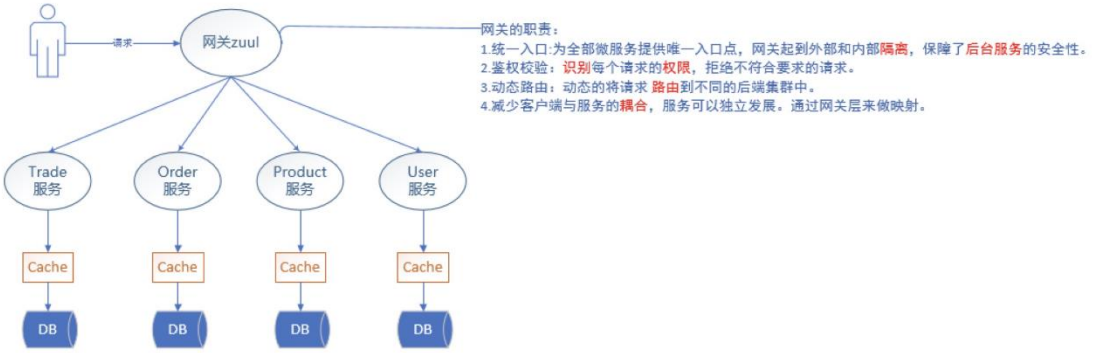
Zuul 网关的作用：

统一入口：为服务提供一个唯一的入口，网关起到外部和内部隔离的作用，保障了后台服务的安全性。

鉴权校验：识别每个请求的权限，拒绝不符合要求的请求。

动态路由：动态的将请求路由到不同的后端集群中。

减少客户端与服务端的耦合：服务可以独立发展，通过网关层来做映射。



6.5.4 熔断器 Hystrix

Hystrix 的中文名字是“豪猪”，豪猪是满身长满了刺，能够保护自己不受天敌的伤害，代表了一种防御机制，Hystrix 在 SpringCloud 中负责服务熔断和服务降级的作用。

什么是服务熔断？（熔断可以保护服务）：

在讲熔断之前先看个概念： 服务雪崩

假设有 A、B、C 三个服务，服务 A 调用服务 B 和 C，链路关系如下：

假设服务 C 因为请求量大，扛不住请求，变得不可用，这样就是积累大量的请求，服务 B 的请求也会阻塞，会逐渐耗尽线程资源，使得服务 B 变得不可用，那么服务 A 在调用服务 B 就会出现问题，导致服务 A 也不可用，那么整条链路的服务调用都失败了，我们称之为雪崩。

接下来看下服务熔断：

互动：举个生活中的例子

当电路发生故障或异常时，伴随着电流不断升高，并且升高的电流有可能能损坏电路中的某些重要器件，也有可能烧毁电路甚至造成火灾。若电路中正确地安置了保险丝，那么保险丝就会在电流异常升高到一定的高度和热度的时候，自身熔断切断电流，从而起到保护电路安全运行的作用。

在微服务架构中，在高并发情况下，如果请求数量达到一定极限（可以自己设置阈值），超出了设置的阈值，Hystrix 会自动开启服务保护功能，然后通过服务降级的方式返回一个友好的提示给客户端。假设当 10 个请求中，有 10%失败时，熔断器就会打开，此时再调用此服务，将会直接返回失败，不再调远程服务。直到 10s 钟之后，重新检测该触发条件，判断是否把熔断器关闭，或者继续打开。

服务降级（提高用户体验效果）：

在高并发的场景下，当服务器的压力剧增时，根据当前业务以及流量的情况，对一些服务和页面进行策略控制，对这些请求做简单的处理或者不处理，来释放服务器资源用以保证核心业务不受影响，确保业务可以正常对外提供服务，比如电商平台，在针对 618、双 11 的时候会有一些秒杀场景，秒杀的时候请求量大，可能会返回报错标志“当前请求人数多，请稍后重试”等，如果使用服务降级，无法提供服务的时候，消费者会调用降级的操作，返回服务不可用等信息，或者返回提前准备好的静态页面写好的信息。

6.5.5 API 网关 Springcloud Gateway

互动： 为什么学习了网关 Zuul，又要讲 Spring Cloud Gateway 呢?

原因很简单，就是 Spring Cloud 已经放弃 Zuul 了。现在 Spring Cloud 中引用的还是 Zuul 1.x 版本，而这个版本是基于过滤器的，是阻塞 IO，不支持长连接，spring 官网上也已经没有 zuul 的组件了，所以给大家讲下 SpringCloud 原生的网关产品 Gateway。

Spring Cloud Gateway 是 Spring Cloud 新推出的网关框架，之前是 Netflix Zuul，由 spring 官方基于 Spring5.0,Spring Boot2.0,Project Reactor 等技术开发的网关，该项目提供了一个构建在 Spring Ecosystem 之上的 API 网关，旨在提供一种简单而有效的途径来发送 API，并向他们提供交叉关注点，例如：安全性，监控/指标和弹性.

SpringCloud Gateway 特征:

SpringCloud 官方对 SpringCloud Gateway 特征介绍如下:

（1）集成 Hystrix 断路器

（2）集成 Spring Cloud DiscoveryClient

（3）Predicates 和 Filters 作用于特定路由，易于编写的 Predicates 和 Filters

（4）具备一些网关的高级功能：动态路由、限流、路径重写

从以上的特征来说，和 Zuul 的特征差别不大。SpringCloud Gateway 和 Zuul 主要的区别，还是在底层的通信框架上。

简单说明一下上文中的三个术语:

1）Filter（过滤器）：

和 Zuul 的过滤器在概念上类似，可以使用它拦截和修改请求，并且对上游的响应，进行二次处理。过滤器为 org.springframework.cloud.gateway.filter.GatewayFilter 类的实例。

2）Route（路由）：

网关配置的基本组成模块，和 Zuul 的路由配置模块类似。一个 Route 模块由一个 ID，一个目标 URI，一组断言和一组过滤器定义。如果断言为真，则路由匹配，目标 URI 会被访问。

3）Predicate（断言）：

这是一个 Java8 的 Predicate，可以使用它来匹配来自 HTTP 请求的任何内容，例如 headers 或参数。断言的输入类型是一个 ServerWebExchange。

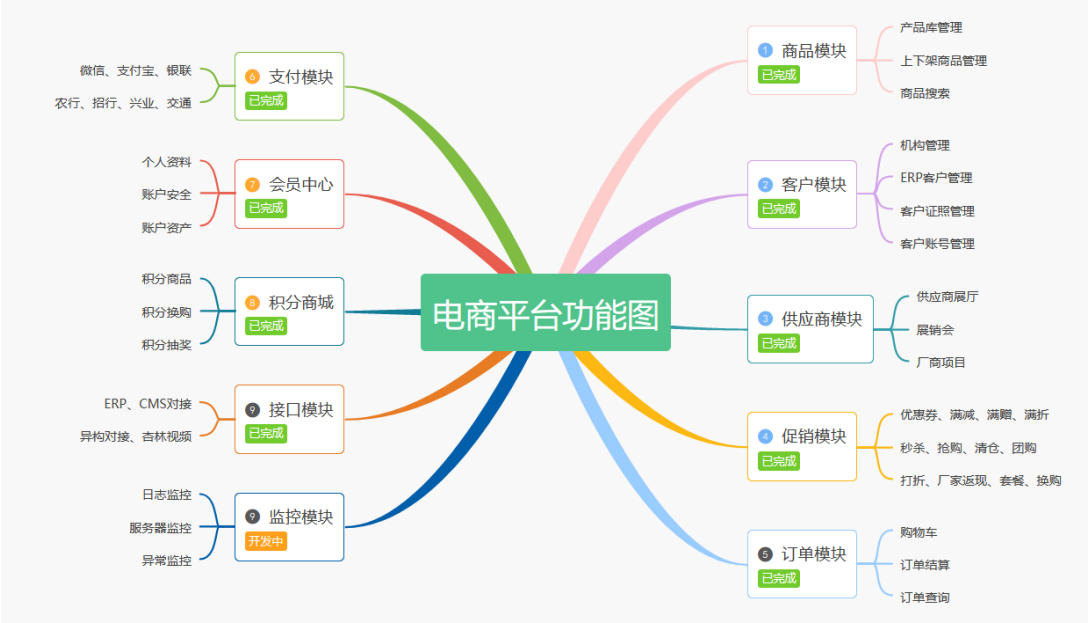
6.5.6 配置中心 SpringCloud Config

SpringCloud Config 是一个解决分布式系统的配置管理方案，它包含了 server 和 client 两个部分。server 用来获取远程的配置信息（默认为 Git 仓库），并且以接口的形式提供出去，client 根据 server 提供的接口读取配置文件，以便于初始化自己的应用。如果配置中心出现了问题，将导致灾难性的后果，因此在生产环境下配置中心都会做集群，来保证高可用。此处配置高可用实际就是把多个配置中心（指定同一个 Git 远程仓库）注册到注册中心。

实战：在 k8s 平台部署 SpringCloud 框架的电商项目：在线购物平台

以下步骤均在 k8s 的控制节点 xuegod63 上操作

1.SpringCloud 的微服务电商框架



2.安装 openjdk 和 maven

在 k8s 控制节点 xuegod63 上操作

```
[root@xuegod63 ~]# yum install java-1.8.0-openjdk   maven-3.0.5* -y
```

3.上传微服务源码包到 k8s 的控制节点 xuegod63 上

```
#解压
[root@xuegod63 ~]# unzip microservic-test.zip
[root@xuegod63 ~]# cd microservic-test
```

4.修改源代码，更改数据库连接地址

在 k8s 控制节点 xuegod63 上操作

1）修改库存服务 stock 的数据库连接地址

```
[root@xuegod63 microservic-test]# vim /root/microservic-test/stock-service/stock-service-biz/src/main/resources/application-fat.yml
```

改成如下：

```
jdbc:mysql://192.168.1.63:3306/tb_stock?characterEncoding=utf-8
#变成自己的数据库地址，192.168.1.63 是我安装数据库的地址
```

2）修改产品服务 product 的数据库连接地址

```
[root@xuegod63 microservic-test]# vim /root/microservic-test/product-service/product-service-biz/src/main/resources/application-fat.yml
```

改成如下：

jdbc:mysql://192.168.1.63:3306/tb_product?characterEncoding=utf-8
#变成自己的数据库地址

3）修改订单数据库
[root@xuegod63 microservic-test]# vim /root/microservic-test/order-service/order-service-biz/src/main/resources/application-fat.yml

改成如下：

url: jdbc:mysql://192.168.1.63:3306/tb_order?characterEncoding=utf-8
#变成自己的数据库地址

5.通过 Maven 编译、构建、打包源代码
在 k8s 控制节点 xuegod63 上操作
修改源代码数据库的地址之后回到/root/microservic-test 目录下执行如下命令：
[root@xuegod63 ~]# cd microservic-test
[root@xuegod63 microservic-test]# mvn clean package -D maven.test.skip=true

编译完成大概需要 15-20 分钟，看到如下说明编译打包已经成功了：
[INFO] simple-microservice ..... SUCCESS [52.385s]
[INFO] basic-common ..... SUCCESS [0.001s]
[INFO] basic-common-core ..... SUCCESS [6:11.156s]
[INFO] gateway-service ..... SUCCESS [3:33.707s]
[INFO] eureka-service ..... SUCCESS [12.075s]
[INFO] product-service ..... SUCCESS [0.001s]
[INFO] product-service-api ..... SUCCESS [0.271s]
[INFO] stock-service ..... SUCCESS [0.002s]
[INFO] stock-service-api ..... SUCCESS [0.233s]
[INFO] product-service-biz ..... SUCCESS [3.776s]
[INFO] stock-service-biz ..... SUCCESS [0.332s]
[INFO] order-service ..... SUCCESS [0.000s]
[INFO] order-service-api ..... SUCCESS [0.270s]
[INFO] order-service-biz ..... SUCCESS [0.364s]
[INFO] basic-common-bom ..... SUCCESS [0.000s]
[INFO] portal-service ..... SUCCESS [0.738s]
[INFO] --
[INFO] BUILD SUCCESS
[INFO] --
[INFO] Total time: 11:13.305s
[INFO] Finished at: Wed Jan 20 15:53:22 CST 2021
[INFO] Final Memory: 92M/710M

[INFO] --

6.在 k8s 中部署 Eureka 组件

修改 k8s 的控制节点 xuegod63 和工作节点的 xuegod64 上的 docker 的配置文件：

```
cat > /etc/docker/daemon.json <<EOF
{
"registry-mirrors":["https://rsbud4vc.mirror.aliyuncs.com","https://registry.docker-cn.com","https://docker.mirrors.ustc.edu.cn","https://dockerhub.azk8s.cn","http://hub-mirror.c.163.com","http://qtid6917.mirror.aliyuncs.com"],
"insecure-registries":["192.168.1.62","harbor"],
"exec-opts":["native.cgroupdriver=systemd"],
 "log-driver":"json-file",
 "log-opts": {
  "max-size": "100m"
  },
 "storage-driver":"overlay2",
 "storage-opts": [
  "overlay2.override_kernel_check=true"
  ]
}
EOF
```

注意：

daemon.json 中新增加了如下一行内容：

```
"insecure-registries":["192.168.1.62","harbor"],
```

这样才可以使用 harbor 私有镜像仓库

重启 docker 使配置生效

```
systemctl daemon-reload && systemctl restart docker && systemctl status docker
```

在 k8s 的控制节点 xuegod63 上操作

创建拉取私有镜像仓库需要的 secret

```
[root@xuegod63 ~]# kubectl create ns ms && kubectl create secret docker-registry registry-pull-secret --docker-server=192.168.1.62 --docker-username=admin --docker-password=Harbor12345   -n ms
```

在 harbor 上创建一个项目 microservice

1）构建镜像

[root@xuegod63 microservic-test]# cd /root/microservic-test/eureka-service

#把 jave-8.tar.gz 上传到 xuegod63 节点，手动解压

[root@xuegod63 microservic-test]# docker load -i jave-8.tar.gz

#构建镜像

[root@xuegod63 microservic-test]# docker build -t

192.168.1.62/microservice/eureka:v1 .

[root@xuegod63 eureka-service]# docker login 192.168.1.62

```
[root@xuegod63 eureka-service]# docker login 192.168.40.132
Username: admin
Password:
```

Username：admin

Password：Harbor12345

[root@xuegod63 microservic-test]# docker push 192.168.1.62/microservice/eureka:v1

2）部署服务

[root@xuegod63 ~]# cd ingress/

[root@xuegod63 ingress]# kubectl apply -f nginx-ingress-controller-rbac.yml

[root@xuegod63 ingress]# kubectl apply -f default-backend.yaml

[root@xuegod63 ingress]# kubectl apply -f nginx-ingress-controller.yaml

[root@xuegod63 ingress]# cd /root/microservic-test/k8s

修改 eureka.yaml 文件，把镜像变成 image: 192.168.1.62/microservice/eureka:v1

```
spec:
  imagePullSecrets:
  - name: registry-pull-secret
  containers:
  - name: eureka
    image: 192.168.40.132/microservice/eureka:v1
```

3）更新 yaml 文件

[root@xuegod63 k8s]# kubectl apply -f eureka.yaml

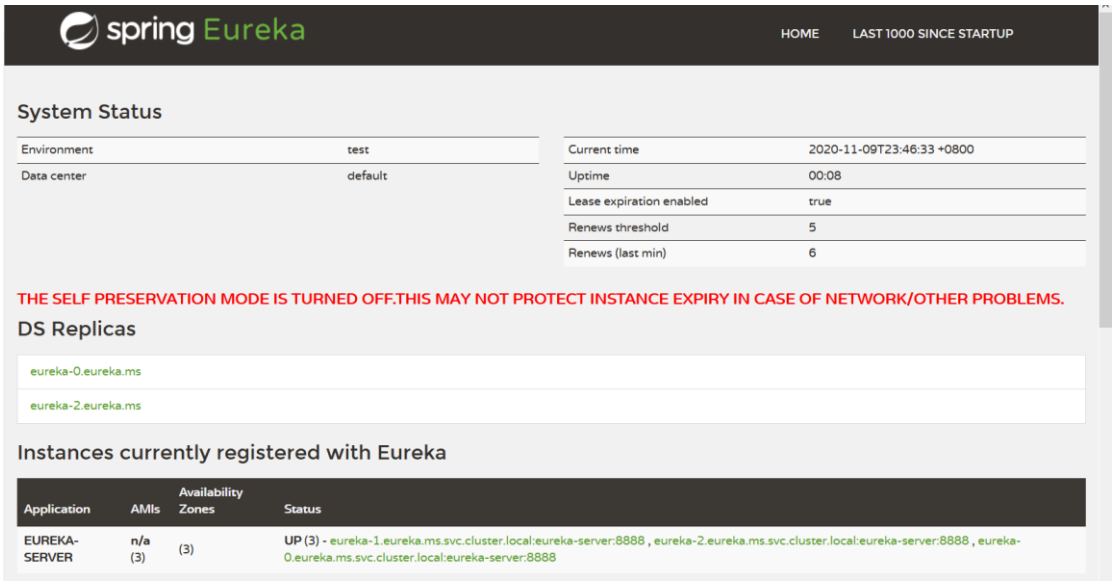
4）查看 pod 状态

```
[root@xuegod63 k8s]# kubectl get pods -n ms
NAME          READY   STATUS    RESTARTS   AGE
eureka-0      1/1     Running   0          3m12s
eureka-1      1/1     Running   0          116s
eureka-2      1/1     Running   0          49s
```

上面运行没问题之后，找到自己电脑的 hosts 文件，新加如下一行：

192.168.1.64　eureka.ctnrs.com

在浏览器访问 eureka.ctnrs.com 即可，可看到如下，说明 eureka 部署成功了：



注意：

要想访问域名 eureka.ctnrs.com，

这个需要在自己电脑的 hosts 文件新增加如下内容，192.168.1.64 是 default-http 和 ingress-nginx-controller 所在 node 节点的 ip 地址。

192.168.1.64　eureka.ctnrs.com

总结：

6.1 大型电商平台微服务架构解读

6.2 单体架构 vs 微服务解读

6.3 哪些项目适合微服务部署?

6.4 SpringCloud 概述

6.5 SpringCloud 组件介绍

实战：在 k8s 平台部署 SpringCloud 框架的电商项目：部署 eureka 服务