# 基于微服务架构的服务治理

一、微服务与服务治理相关技术分析

1. 微服务是什么？

微服务(Microservices Architecture)是一种架构风格。一个大型复杂软件应用由一个或多个微服务组成。系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。这些服务通常有自己的堆栈，包括数据库和数据模型；通过REST API，事件流和消息代理的组合相互通信；按业务能力组织的，分隔服务的线通常称为有界上下文。在所有情况下，每个任务代表着一个小的业务能力。

**定义：**围绕业务领域组件来创建应用，这些应用可独立地进行开发、管理和迭代。在分散的组件中使用云架构和平台式部署、管理和服务功能，使产品交付变得更加简单。

**本质：**用一些功能比较明确、业务比较精练的服务去解决更大、更实际的问题。

**优点：**每个服务都比较简单，只关注于一个业务功能。微服务架构方式是松耦合的，可以提供更高的灵活性。可通过不同的编程语言与工具进行开发，能够做到有的放矢地解决针对性问题。每个微服务可由不同团队独立开发，互不影响，加快推出市场的速度。微服务架构是持续交付的巨大推动力，允许在频繁发布不同服务的同时保持系统其他部分的可用性和稳定性。

**案例：**

①SpringCloud。

②Apache ServiceComb是业界第一个Apache微服务顶级项目，是一个开源微服务解决方案，致力于帮助企业、用户和开发者将企业应用轻松微服务化上云，并实现对微服务应用的高效运维管理。其提供一站式开源微服务解决方案，融合SDK框架级、0侵入ServiceMesh场景并支持多语言。

③ZeroC IceGrid 是ZeroC公司的杰作，继承了CORBA的血统，是新一代的面向对象的分布式系统中间件。作为一种微服务架构，它基于RPC框架发展而来，具有良好的性能与分布式能力。

2. 为什么需要微服务？

**1）早期的单体架构带来的问题**

单体架构把所有的功能集成在一个项目工程中，项目架构简单，前期开发成本低，周期短。在规模比较小的情况下工作情况良好，但是随着系统规模的扩大，暴露出来的问题也越来越多，主要有以下几点：

①复杂性逐渐变高。大型项目有几十万行代码，各个模块之间区别比较模糊。代码多导致复杂性高，遇到问题后难以解决。

②技术债务逐渐上升。公司的人员流动，有的员工在离职之前，疏于代码质量的自我管束，留下来很多问题。而且很难被发觉，这就给新来的员工带来麻烦。就是所谓的技术债务越来越多。

③部署速度逐渐变慢。单体架构模块非常多，代码量非常庞大，导致部署项目所花费的时间越来越多。

④无法按需伸缩。由于所有的模块都在一个架构下，扩展某一模块的性能时不得不考虑其它模块的因素。大型项目不易开发、扩展和维护。

**2）微服务与单体架构区别**

①单体架构所有的模块全都耦合在一块，代码量大，维护困难。微服务每个模块就相当于一个单独的项目，代码量明显减少，遇到问题也相对来说较好解决。

②单体架构通常所有的模块都共用一个数据库，存储方式比较单一，微服务每个模块都可以使用不同的存储方式，数据库也是单个模块对应自己的数据库。

③单体架构所有的模块开发所使用的技术一样，微服务每个模块都可以使用不同的开发技术，开发模式更灵活。

**3）微服务架构与SOA架构的对比**

当应用越来越多时，应用之间交互不可避免。出现SOA（ Service Oriented Architecture ，面向服务的架构）。可以根据需求通过网络对松散耦合的粗粒度应用组件(服务)进行分布式部署、组合和使用。一个服务通常以独立的形式存在于操作系统进程中。把原先固有的业务功能转变为通用的业务服务，实现业务逻辑的快速复用。多个服务之间通过相互依赖最终提供一系列的功能。微服务架构其实和 SOA 架构类似，是在 SOA 上做的升华。微服务架构强调的一个重点是“业务需要彻底的组件化和服务化”，原有的单个业务系统会拆分为多个可以独立开发、设计、运行的小应用。这些小应用之间通过服务完成交互和集成。

3. 如何实现微服务？

一个微服务架构：

**图片包含 游戏机, 截图

描述已自动生成**

图1.1: 微服务架构

实际应用微服务，需要解决四个问题：

1）客户端如何访问这些服务

在后端N个服务和UI之间一般会有一个代理或者API Gateway，作用包括：提供统一服务入口，让微服务对前端透明；聚合后端的服务，节省流量，提升性能；提供安全，过滤，流控等API管理功能。API Gateway有很多广义的实现办法，可以是一个软硬一体的盒子，也可以是一个简单的MVC框架，甚至是一个Node.js的服务端。重要的作用是为前端（通常是移动应用）提供后端服务的聚合，提供一个统一的服务出口，解除他们之间的耦合，但API Gateway也有可能成为单点故障点或者性能的瓶颈。

2）每个服务之间如何通信

所有的微服务都是独立的Java进程跑在独立的虚拟机上，所以服务间的通信是IPC（inter process communication），已经有很多成熟的方案。现在基本通用的有两种方式：

同步调用：REST（JAX-RS，Spring Boot）、RPC（Thrift, Dubbo）

异步消息调用(Kafka, Notify, MetaQ)

同步和异步的区别：一般同步调用比较简单，一致性强，但是容易出问题，性能体验上也会差些，特别是调用层次多的时候。REST基于HTTP，更容易实现，更容易被接受，服务端实现技术也更灵活些，各个语言都能支持，同时能跨客户端，对客户端没有特殊的要求，只要封装了HTTP的SDK就能调用，所以相对使用广泛。RPC也有自己的优点，传输协议更高效，安全更可控。如果有统一的开发规范和统一的服务框架时，开发效率优势更明显些。而异步消息的方式在分布式系统中有广泛的应用，既能减低调用服务之间的耦合，又能成为调用之间的缓冲，确保消息积压不会冲垮被调用方，同时能保证调用方的服务体验，不被后台性能拖慢。但一致性减弱，数据能达到最终一致性。还有后台服务一般要实现幂等性，因为消息发送出于性能的考虑一般会有重复（保证消息的被收到且仅收到一次对性能是很大的考验）；最后必须引入一个独立的broker。

3）很多服务，如何实现

在微服务架构中，一般每一个服务都是有多个拷贝，来做负载均衡。一个服务随时可能下线，也可能应对临时访问压力增加新的服务节点。服务之间如何相互感知？服务如何管理？这就是服务发现的问题了。一般有两类做法，基本都是通过zookeeper等类似技术做服务注册信息的分布式管理。当服务上线时，服务提供者将自己的服务信息注册到ZK（或类似框架），并通过心跳维持长链接，实时更新链接信息。服务调用者通过ZK寻址，根据可定制算法，找到一个服务，还可以将服务信息缓存在本地以提高性能。当服务下线时，ZK会发通知给服务客户端。

客户端做：优点是架构简单，扩展灵活，只对服务注册器依赖。缺点是客户端要维护所有调用服务的地址，有技术难度，一般大公司都有成熟的内部框架支持，比如Dubbo。

服务端做：优点是简单，所有服务对于前台调用方透明，一般在小公司在云服务上部署的应用采用的比较多

4）服务挂了，如何解决

分布式最大的特性就是网络不可靠。通过微服务拆分能降低这个风险，但没有特别的保障是不行的。所以系统是由一系列的服务调用链组成时，必须确保任一环节出问题都不至于影响整体链路。相应的手段有很多：①重试机制②限流③熔断机制④负载均衡⑤降级等。

4. 微服务框架

现在比较成熟的服务框架有很多，比如Dubbo、Spring Cloud、gRPC、thrift等。本文档只介绍Spring Cloud。

图片包含 游戏机, 文字

描述已自动生成图1.2：Spring Cloud介绍

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性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Spring Boot的开发风格做到一键启动和部署。Spring Cloud将目前各家公司开发的比较成熟的服务框架组合起来，通过Spring Boot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。目前Spring Cloud 规范有 Spring官方，Spring Cloud Netflix，Spring Cloud Alibaba等实现。通过组件化的方式，Spring Cloud将这些实现整合到一起构成微服务技术栈。

1）组件介绍

Eureka，服务注册中心

Ribbon，客户端负载均衡

Feign，声明式服务调用

Hystrix，客户端容错保护

Zuul ，API服务网关

Config，分布式配置中心，支持本地仓库、SVN、Git、Jar包内配置等模式

Dashboard，Hystrix仪表盘，监控集群模式和单点模式，其中集群模式需要收集器Turbine配合

Bus，消息总线

2）组件主要功能

Eureka和Ribbon，一个注册服务，一个消费服务。

Hystrix，防止整个微服务架构因为某个服务节点的问题导致崩溃，起到保险丝的作用。

Dashboard，给Hystrix统计和展示用，而且监控服务节点的整体压力和健康情况。Turbine，集群收集器，服务于Dashboard。

Zuul，加在整个微服务最前沿的防火墙和代理器，隐藏微服务结点IP端口信息，加强安全保护。

Config，为了解决所有微服务各自维护各自的配置，设置一个统一的配置中心，方便修改配置。

Bus是因为config修改完配置后各个结点都要refresh才能生效实在太麻烦，所以交给bus来通知服务节点刷新配置的。

为了提供微服务之间的发现和注册，需要服务注册中心提供注册与发现功能，采用Eureka组件。消费者从注册中心拿到服务提供者的注册地址信息列表，为了从中选择一个提供方，避免流量集中在某台机，需要客户端负载均衡组件Ribbon。消费者得到具体提供方的地址信息后，发起远程调用，通过声明式调用组件Feign，将远程RestFul调用封装成接口调用。在微服务架构中，为了避免某个服务挂掉后引起服务雪崩，以及做到服务端失败后快速响应，提供了服务熔断以及降级机制Hystrix组件。为了对外屏蔽内部服务调用细节，在最前沿加入网关组件，提供流量控制、身份鉴别、负载均衡等安全防护功能，采用API网关Zuul组件。随着系统访问量的增加，为避免单点的注册中心成为系统性能瓶颈，注册中心采用Eureka集群方式部署。随着系统扩大，Eureka集群的部署以及服务模块的增多，每个模块维护各自的配置信息，复杂且容易出错，引入配置中心SpringCloud Config组件，提供统一的配置信息管理。为了避免每次配置信息的变更都需要重启服务才能生效，引入消息总线Bus组件，提供无需重启、实时刷新配置信息等功能。

5.服务治理

微服务架构的缺点中最主要的就是由于微服务数量众多导致维护成本巨大，服务治理为解决此问题而产生的。服务治理的作用是让维护人员从人工维护中解放出来，由服务自维护，微服务作为服务提供方主动向服务治理中心注册，服务的消费方通过服务治理中心查询需要的服务并进行调用。

图片包含 游戏机, 截图

描述已自动生成

图1.3：Spring Cloud服务流程

相关技术：

1）服务注册与发现

①服务注册

服务实例将自身服务信息注册到注册中心。这部分服务信息包括服务所在主机IP和提供服务的Port，以及暴露服务自身状态以及访问协议等信息。

②服务发现

服务实例请求注册中心获取所依赖服务信息。服务实例通过注册中心，获取到注册其中的服务实例的信息，通过这些信息去请求它们提供的服务。

图片包含 游戏机, 文字, 地图

描述已自动生成

图1.4：服务注册与发现

2）负载均衡

负载均衡是高可用网络基础架构的关键组件，通常用于将工作负载分布到多个服务器来提高网站、应用、数据库或其他服务的性能和可靠性。

图片包含 游戏机

描述已自动生成

图1.5：负载均衡

3）熔断

在互联网系统中，当下游服务因访问压力过大而响应变慢或失败，上游服务为了保护系统整体的可用性，可以暂时切断对下游服务的调用。这种牺牲局部，保全整体的措施就叫做熔断。

图片包含 游戏机, 钟表, 标志

描述已自动生成

图1.6：熔断

4）链路追踪

随着微服务架构的流行，服务按照不同的维度进行拆分，一次请求往往需要涉及到多个服务。互联网应用构建在不同的软件模块集上，这些软件模块，有可能是由不同的团队开发、可能使用不同的编程语言来实现、有可能布在了几千台服务器，横跨多个不同的数据中心。因此，就需要对一次请求涉及的多个服务链路进行日志记录，性能监控即链路追踪。

图片包含 游戏机

描述已自动生成

图1.7：链路追踪

5）API网关

随着微服务的不断增多，不同的微服务一般会有不同的网络地址，而外部客户端可能需要调用多个服务的接口才能完成一个业务需求，如果让客户端直接与各个微服务通信可能出现：客户端需要调用不同的url地址，增加难度；在一定的场景下，存在跨域请求的问题；每个微服务都需要进行单独的身份认证。针对这些问题，API网关顺势而生。API网关是将所有API调用统一接入到API网关层，由网关层统一接入和输出。一个网关的基本功能有：统一接入、安全防护、协议适配、流量管控、长短链接支持、容错能力。有了网关之后，各个API服务提供团队可以专注于自己的的业务逻辑处理，而API网关更专注于安全、流量、路由等问题。

图片包含 游戏机, 截图, 钟表

描述已自动生成

图1.8：API网关

二、基本功能实现

本次实验实现了商品微服务，模拟商家在销售中对自身中对自身库存进行查询、得到指定商品的库存数目以及罗列出目前仓库中所有商品的库存、进货后对商品的库存进行更改。

服务提供者通过查询本地的数据库获得指定商品的库存、列举指定商品的信息和所有商品的信息、进货修改本地库存的数量。在这种场景下，商品微服务是一个服务提供者，如下图所示：

图片包含 游戏机, 画, 桌子

描述已自动生成

图2.1：最简单的微服务架构实现

商品类的属性包含id（唯一的标识，不能相同），商品的名称，商品的描述信息以及当前库存的数量。实验中通过product\_common实现了商品的基本类，并实现了对商品库存量的查询、商品的进货后库存的修改、根据id查询指定商品的信息。如下图所示：

图片包含 游戏机, 截图

描述已自动生成 

图2.2：商品类的构造和实现

实验中定义的数据库如下：

图片包含 截图, 游戏机, 播放器, 男人

描述已自动生成图2.3：数据库的初始化

该仓库目前所拥有的的商品主要有电视机、笔记本电脑、平板、电水壶、电冰箱等，库存数量间number所在行。通过product\_provider实现了服务提供者对应的微服务，主要的操作如下：

a.查询指定商品（<http://localhost:8004/product/get/1>）：

图片包含 游戏机, 截图

描述已自动生成

图2.4：查询指定商品信息

b.查询目前仓库中所有的商品情况（<http://localhost:8004/product/list>）：

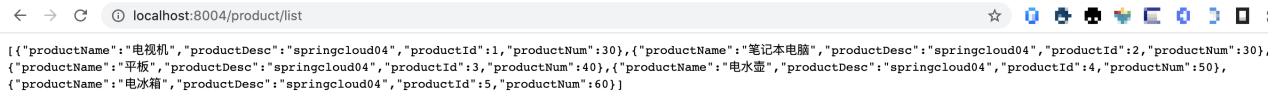


图2.5：查询仓库中所有商品的信息

c.查询指定商品的库存(<http://127.0.0.1:8004/product/getRepertory/1>)：

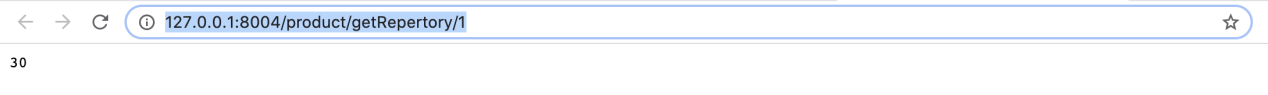
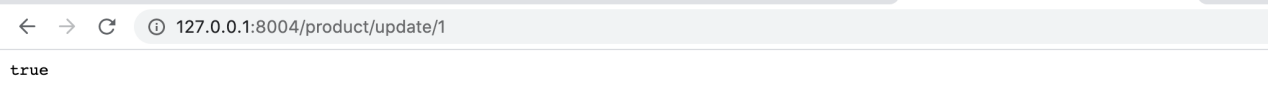


图2.6：查询指定商品库存

d.进货后更新商品库存(<http://127.0.0.1:8004/product/update/1>)：



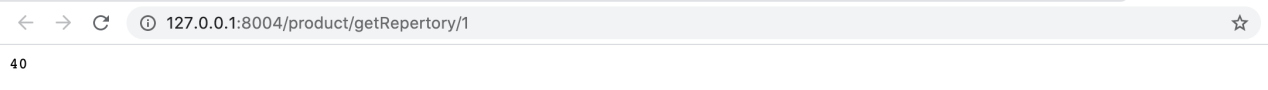


图2.7：进货后更新指定商品库存

备注，此时进货默认每件货进10件，商品的库存数量增加10，返回进货成功与否。

用户不能每次都到提供者所提供的端口和地址去访问，此时用户可以通过创建服务消费者的微服务，消费者和提供者通过网络连接起来，用户通过消费者调用商品微服务的相关操作。本次实验中，通过定义product\_consumer来调用提供者所提供的服务，此时的调用关系如下：



图2.8：加入服务消费者后的架构

product\_cosumer在80端口通过网络访问商品微服务在8004端口提供的微服务，进一步将服务解耦，保护后端程序的安全性和隐私性，用户只能拿到提供者想让他看的。功能实现如下：

a.查询指定商品(<http://127.0.0.1/consumer/product/get/1>)

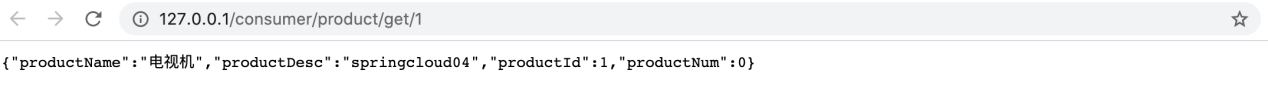


图2.9：查询指定商品信息

b.查询目前仓库中所有的商品情况(<http://127.0.0.1/consumer/product/list>)

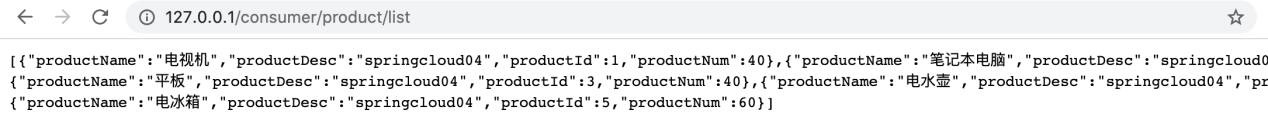


图2.10：查询仓库中所有商品的信息

c.查询指定商品的库存(<http://127.0.0.1/consumer/product/getRepertory/1>)

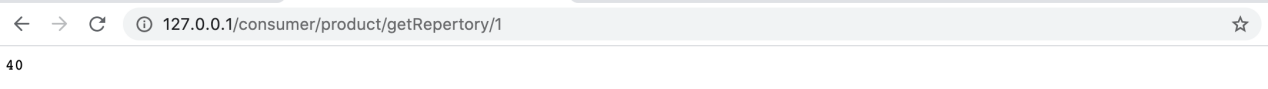


图2.11：查询指定商品库存

d.进货后更新商品库存

图片包含 游戏机, 截图, 鸟

描述已自动生成

图2.12：进货后更新失败的结果

对用户权限进行控制，不允许其通过服务消费者进货后修改库存。

三、微服务进阶功能

第二节已经实现了基本的服务和功能调用，但是随时用户访问数量的增多和服务的增多，为保证服务的健壮性和安全性，需要进一步将服务进行划分，主要实现了以下功能:

1. 服务注册（Eureka Server）

注册中心可以说是微服务架构中的”通讯录“，它记录了服务和服务地址的映射关系。在分布式架构中，服务会注册到这里，当服务需要调用其它服务时，就这里找到服务的地址，进行调用。

服务注册中心是微服务架构非常重要的一个组件，在微服务架构里主要起到了协调者的一个作用，可以实现服务注册和发现、服务配置和服务健康监测等功能。主要的服务注册中心有Zookeeper, Eureka, Consul, Nacos, 本次实验主要基于经典的Zookeeper进行实现。

图片包含 游戏机, 标志

描述已自动生成

图3.1：加入服注册中心后的架构

本次实验由四个同名的商品微服务分别在8001、8002、8003、8004端口提供微服务，他们将向服务注册中心（端口为7001）注册服务，用户可以根据服务名称在注册中心查询得到服相应的服务并调用服务。

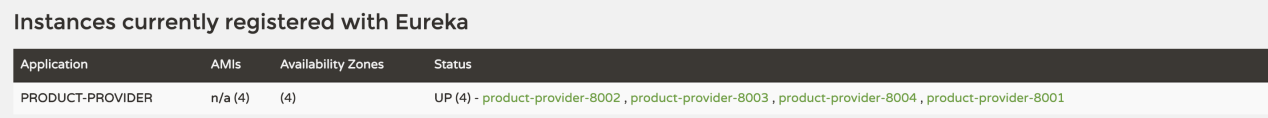
注册结果如下所示：  


图3.2：服务在注册中心中注册

如图所示，四个名称为product-provider的服务已经注册到7001的注册中心，后续可以直接通过该名称访问该服务。

2. 服务发现与使用（Eureka Client）

第二节客户端在80端口通过客户消费者访问服务提供者的所提供的商品微服务，但消费者到提供者是通过连接提供者的ip静态连接的。静态连接的方式使得服务方ip变动以后客户端必须相应作出修改，否则不能访问；另外，当服务方较多时需要多次绑定且容易出错。本实验基于3.1节的服务注册，通过服务提供者提供的商品名称，动态匹配服务提供者的路由，增加了灵活性和可用性。

实现过程的部分代码如下图：

图片包含 截图, 室内, 屏幕, 监控

描述已自动生成

图3.3：服务发现的部分代码

客户端通过消费者访问的结果如下：

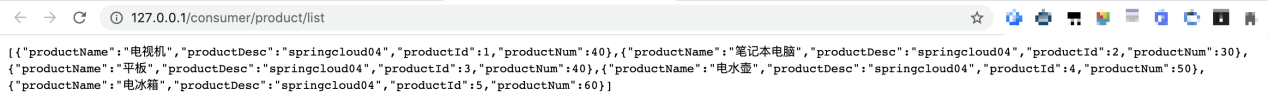
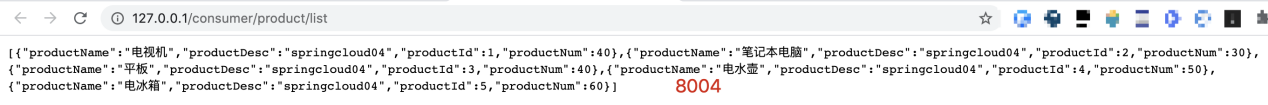


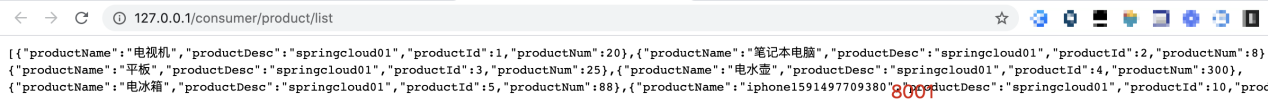
图3.4：通过服务发现请求服务的结果

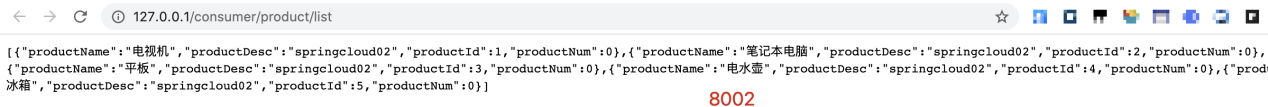
此时通过查询注册中心注册服务动态匹配访问的服务提供者地址，结果正确。

3. 负载均衡（Ribbon和Feign）

目前已经实现了服务的注册和服务发现。当启动某个服务的时候，可以通过HTTP的形式将服务注册到注册中心，并且可以通过SpringCloud提供的工具获取注册中心的服务列表。但是当访问请求逐渐增多时如何选择后端的微服务，这就是负载均衡需要解决的问题。本次实验实现了基于Ribbon和Feign的负载均衡方法，服务提供者有四个，采用“轮询”的负载均衡策略顺序分配客户端的调用请求，执行结果如下：







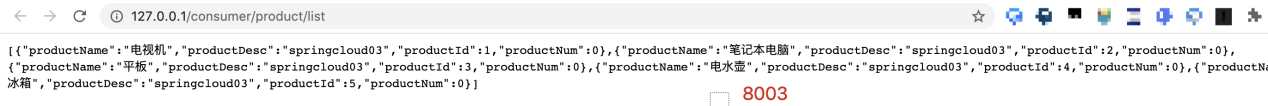
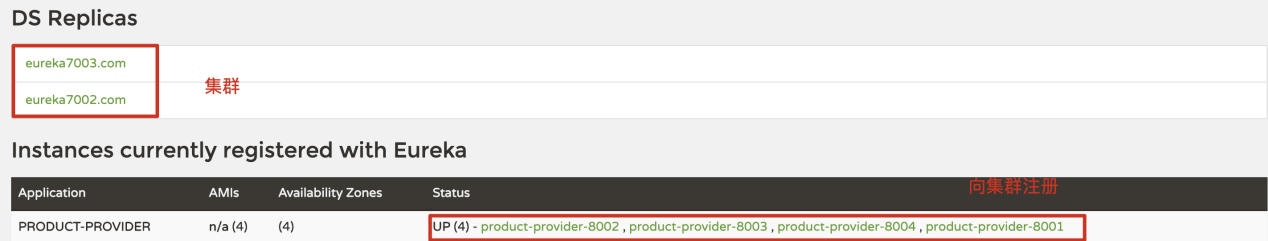
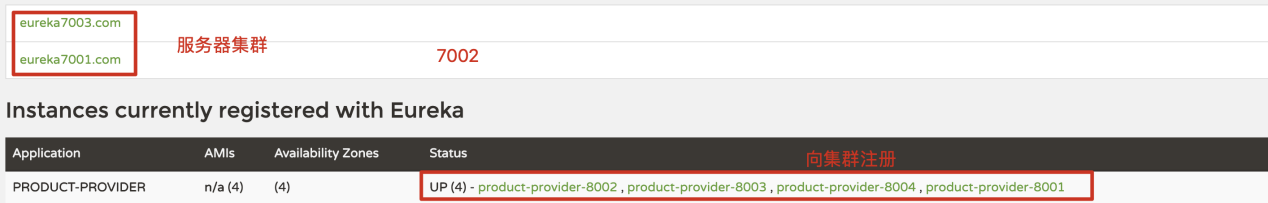


图3.5：负载均衡的结果

4.注册中心集群化

服务提供者需要将自己所提供的服务注册到注册中心，服务消费者需要访问服务注册中心得到所请求服务的地址。此时，Eureka服务注册中心一旦被破坏，整个系统就不能正常工作。为解决注册中心被破坏或者受到拒绝服务攻击使得系统瘫痪，本实验将注册中心由一个扩展成三个注册中心组成的注册中心集群，服务注册时向集群注册并且请求服务时向集群查询，任意注册中心被破坏后不影响系统的整体作用，并且注册查询的效率有所提升。结果如下所示：





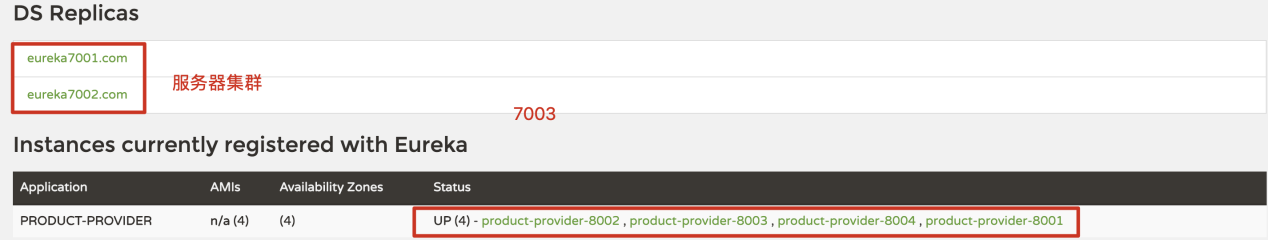


图3.6：注册中心集群化

如上图所示，服务注册中心由原来的7001端口的单个微服务，组合形成了三个注册中心组成的的集群（端口分别是7001，7002和7003）。整体的体系结构如下图所示：

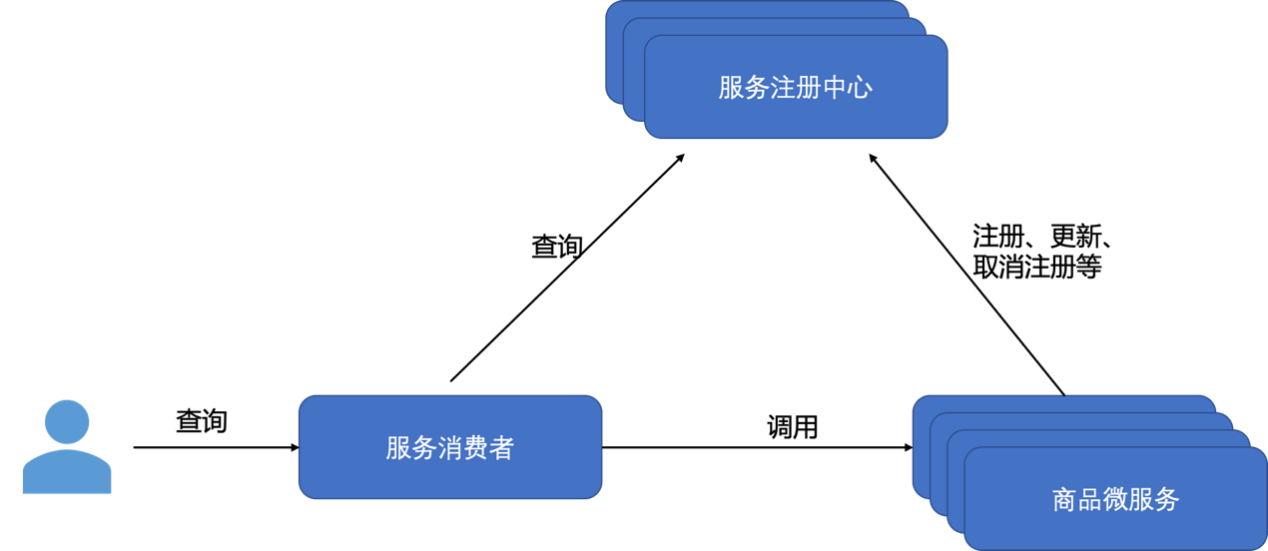


图3.7：注册中心集群化后的体系结构

5. 服务降级（Feign Circuit）

1）服务降级的概念、作用原理等

服务降级和服务熔断非常类似，都是为了系统的稳定性，防止因为个别微服务的不可用而拖死整个系统服务；在表现上都是让用户感知，该服务暂时不可用请稍后再试；粒度上，都是服务级别的粒度。但降级和熔断存在区别：

服务熔断一般是某个服务挂掉了引起的，一般是下游服务，而服务降级一般是从整体的负荷考虑，主动降级；

熔断其实是一个框架级的处理，每个微服务都需要，没有层次之分，而降级一般需要对业务有层级之分，一般是从最外围服务开始。

2）服务降级的实现（consumer\_feign\_80）

添加Feign支持

**<dependency>**

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

**<artifactId>**spring-cloud-starter-openfeign**</artifactId>**

**<version>**2.1.0.RELEASE**</version>**

**</dependency>**

增加Product\_feign\_consumer\_80类：

@SpringBootApplication

@EnableHystrixDashboard

@EnableFeignClients

@EnableCircuitBreaker

**public** **class** Product\_feign\_consumer\_80 {

**public** **static** **void** main(String[] args) {

        SpringApplication.run(Product\_feign\_consumer\_80.**class**, args);

    }

}

在ConsumerController类中添加降级方法

@RestController

@RequestMapping("/consumer")

**public** **class** ConsumerController {

    @Autowired

**private** ProductService productService;

    @RequestMapping("/product/get/{id}")

**public** Object getProduct(@PathVariable("id") **long** id) {

        Product product = productService.getProduct(id);

**return** product;

    }

    @RequestMapping("/product/list")

**public** Object listProduct() {

        List<Product> list = productService.listProduct();

**return** list;

    }

    @RequestMapping("/product/add")

**public** Object addPorduct(Product product) {

        Boolean result = productService.addProduct(product);

**return** result;

    }

}

申明需要调用的微服务

@FeignClient(name = "PRODUCT-PROVIDER", fallback = ProductFeignServiceCallBack.**class**)

**public** **interface** ProductService {

    /\*

     \*配置调用微服务的接口

     \*/

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

**public** Product getProduct(@PathVariable("id") **long** id);

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

**public** List<Product> listProduct();

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

**public** Boolean addProduct(Product product);

}

实现服务熔断的降级操作

@Component

**public** **class** ProductFeignServiceCallBack **implements** ProductService {

   ......

**public** List<Product> listProduct() {

        Product product = **new** Product();

        product.setProductId(-1L);

        product.setProductName("服务熔断的降级操作");

        product.setProductDesc("不好意思啊 现在用不了 先打一把游侠放松下待会再来哦");

        List<Product> list = **new** ArrayList<Product>();

        list.add(product);

**return** list;

    }

......

3）运行测试(已修改)

当product\_provider服务正常时，浏览器正常访问：

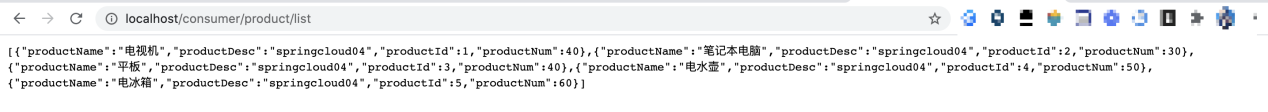


图3.8：正常访问

将服务提供者关闭，再次访问时发生服务熔断,再次访问结果如下：

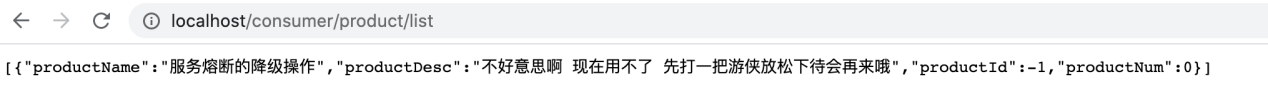


图3.9：服务降级的访问

6. 服务监控（Dashboard）

单点应用定位问题相对来说是比较好定位的，但是服务多了，微服务之间互相的接口交互，哪个节点出现了问题都需要快速定位到出现问题的点，若微服务运行在容器当中，我们同样要考虑容器的监控，以及app的日志监控，同样还有app的业务监控。

微服务架构下的监控有如下难点：监控对象动态可变，无法进行预先配置；监控范围非常繁杂，各类监控难以互相融合；微服务实例间的调用关系非常复杂，故障排查会很困难；微服务架构仍在快速发展，难以抽象出稳定的通用监控模型。

Hystrix提供了对于微服务调用状态的监控信息，但是需要结合spring-boot-actuator模块一起使用。Hystrix Dashboard是Hystrix的一个组件，Hystrix Dashboard提供一个断路器的监控面板，可以使我们更好的监控服务和集群的状态，仅仅使用Hystrix Dashboard只能监控到单个断路器的状态，实际开发中还需要结合Turbine使用。

Hystrix Dashboard主要用来实时监控Hystrix的各项指标信息。通过Hystrix Dashboard反馈的实时信息，可以帮助我们快速发现系统中存在的问题。  
 使用基于Hystrix的提供者访问数据库表数据，每访问一次都会记录是否成功以及最近10s错误百分比、超时数、熔断数、线程拒绝数、错误请求数、失败/异常数、服务请求频率等相关信息

2）导入依赖

<dependencies>

    <dependency>

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

        <artifactId>spring-cloud-starter-netflix-turbine</artifactId>

        <version>2.1.0.RELEASE</version>

    </dependency>

    <dependency>

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

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

        <version>1.4.6.RELEASE</version>

    </dependency>

    <dependency>

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

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

        <version>1.4.6.RELEASE</version>

    </dependency>

</dependencies>

3） 在启动类使用@EnableHystrixDashboard注解激活仪表盘项目

**public** **class** Hystrix\_trubine\_7010 {

**public** **static** **void** main(String[] args) {

        SpringApplication.run(Hystrix\_trubine\_7010.**class**, args);

    }

4） 访问测试

在7010端口实现了服务的监控，访问localhost:7010/hystrix得到监控的主界面如下：

图片包含 游戏机, 截图

描述已自动生成

输入监控的页面网址为：<http://localhost:7010/actuator/hystrix.stream>进行服务监控

7. 服务熔断（Hystrix）

1）服务熔断发生的情况、作用及原理等

在我们的微服务架构中，一个请求需要调用多个服务是很常见的。当一个服务处于阻塞状态时，若有大量的请求涌入，容器的线程资源会被消耗完毕，导致服务瘫痪。而服务之间的依赖性会导致故障传播，造成连锁反应，对整个微服务系统造成灾难性的严重后果，即“雪崩”效应。

从源头我们无法杜绝雪崩的发生，但是雪崩的根本原因来自于服务之间的强依赖，所以我们可以提前评估，做好熔断、降级、限流。这一部分我们介绍我们的商品微服务中熔断的实现。

2）熔断的实现(已修改)：

a). 配置依赖

图片包含 截图, 监控, 电话, 屏幕

描述已自动生成

b). 配置触发熔断的方法

图片包含 监控, 屏幕, 绿色, 桌子

描述已自动生成

requestVolumeThreshold：触发熔断的最小请求次数，默认2，即为每10秒钟访问20次，本次实验为方便观察设置为5。

errorThresholdPercentage：触发熔断的失败请求最小占比，默认50% sleepWindowInMilliseconds：熔断后多少秒后去尝试请求，默认为5s，实验为方便观察设置为10s。

c). 设置服务熔断的降级方法

图片包含 游戏机

描述已自动生成

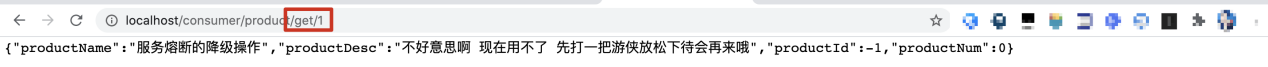
3）运行测试

在服务中改写get方法，当访问id为1时访问失败；多次请求get/1就会触发熔断；一切请求都会被降级处理。此时你访问id为2的请求，会发现返回的也是失败，而且失败时间很短，只有20毫秒左右：

图片包含 截图, 监控, 屏幕, 电视

描述已自动生成

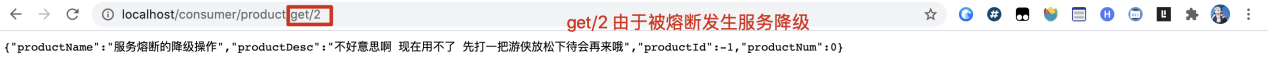
访问get/1必然报错，发生服务降级：



此时访问get/2,能够正常返回：



多次访问get/1（至少五次）,使得服务熔断；此时访问get/2,发现该服务也被降级处理：



等待10s后再次请求get/2,结果正常：



8. 网关服务（Zuul）

1) 网关服务作用

不同的微服务一般会有不同的网络地址，客户端在访问这些微服务时必须记住几十甚至几百个地址，这对于客户端方来说太复杂也难以维护。如果让客户端直接与各个微服务通讯，可能会有很多问题：

* 客户端会多次请求不同的微服务，增加了客户端的复杂性。
* 存在跨域请求，在一定场景下处理相对复杂。
* 认证复杂，每个服务都需要独立认证。
* 难以重构，随着项目的迭代，可能需要重新划分微服务。例如，可能将多个服务合并成一个或者将一个服务拆分成多个。如果客户端直接与微服务通讯，那么重构将会很难实施。
* 某些微服务可能使用了防火墙/浏览器不友好的协议，直接访问会有一定困难。

因此，我们需要一个微服务网关，介于客户端与服务器之间的中间层，所有的外部请求都会先经过微服务网关。客户端只需要与网关交互，只知道一个网关地址即可，这样简化了开发还有以下优点：

* 易于监控。可在微服务网关收集监控数据并将其推送到外部系统进行分析。
* 易于认证。可在微服务网关上进行认证。然后再将请求转发到后端的微服务，而无须在每个微服务中进行认证。
* 减少了客户端与各个微服务之间的交互次数。通常而言不同的客户端对于显示时对于数据的需求是不一致的，比如手机端或者Web端又或者在低延迟的网络环境或者高延迟的网络环境。因此为了优化客户端的使用体验，API Gateway可以对通用性的响应数据进行裁剪以适应不同客户端的使用需求。同时还可以将多个API调用逻辑进行聚合，从而减少客户端的请求数，优化客户端用户体验
* 简化客户端调用复杂度。在微服务架构模式下后端服务的实例数一般是动态的，对于客户端而言很难发现动态改变的服务实例的访问地址信息。因此在基于微服务的项目中为了简化前端的调用逻辑，通常会引入API Gateway作为轻量级网关，同时API Gateway中也会实现相关的认证逻辑从而简化内部服务之间相互调用的复杂度

2） Zuul网关实现

在pom.xml中添加相关依赖

<dependency>

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

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

</dependency>

<dependency>

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

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

    <version>2.1.0.RELEASE</version>

</dependency>

在application.yml中进行配置

server:

  port: 8080

spring:

  application:

    name: server-zuul-8080

eureka:

  client:

    register-with-eureka: **false**

    service-url:

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

创建启动类，在启动类上添加@EnableZuulProxy注解来启用Zuul的API网关功能

@SpringBootApplication

@EnableZuulProxy

@EnableDiscoveryClient

**public** **class** Zuul\_server\_8080 {

**public** **static** **void** main(String[] args) {

        SpringApplication.run(Zuul\_server\_8080.**class**, args);

    }

}

Zuul路由转发

根据url匹配规则，将不同的请求转发到不同的微服务处理

routes:

    - id: product-gateway-server-8080

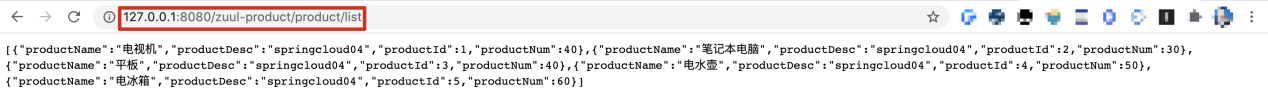
#   url: http://127.0.0.1:8004      #静态路由

    url: lb://product-provider       #动态路由，通过服务名来匹配服务

    predicates:

    - Path=/product/\*\*

配置好zuul路由后启动服务，在浏览器中输入<http://127.0.0.1:8080/zuul-product/product/list>,即可访问到微服务：



面向服务的路由

微服务一般是由几十、上百个服务组成，对于一个URL请求，最终会确认一个服务实例进行处理。如果对每个服务实例手动指定一个唯一访问地址，然后根据URL去手动实现请求匹配，这样做显然就不合理。我们将Zuul与Eureka整合开发，根据ServiceID自动的从注册中心中获取服务地址并转发请求，这样做不仅可以通过单个端点来访问应用的所有服务，而且在添加或移除服务实例的时候不用修改Zuul的路由配置。

添加Eureka客户端依赖

<dependency>

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

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

    <version>2.1.0.RELEASE</version>

</dependency>

开启Eureka客户端发现功能

@SpringBootApplication

**public** **class** Product\_gateway\_8080 {

**public** **static** **void** main(String[] args) {

        SpringApplication.run(Product\_gateway\_8080.**class**, args);

    }

}

添加Eureka配置，获得服务信息

eureka:

  client:

    register-with-eureka: **false**

    service-url:

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

  instance:

    prefer-ip-address: **true**

    ip-address: localhost

9. 服务过滤（Zuul Filter）

1） 过滤器的作用

我们已经实现请求的路由功能，所以我们的微服务应用提供的接口就可以通过统一的 API 网关入口被客户端访问到了。在现实应用中，每个客户端用户请求微服务应用提供的接口时，它们的访问权限往往都有一定的限制，系统并不会将所有的微服务接口都对它们开放。

为了实现对客户端请求的安全校验和权限控制，有两种方法，第一种是为每个微服务应用都实现一套用于校验签名和鉴别权限的过滤器或拦截器。不过，这样的做法会增加日后的系统维护难度，因为同一个系统中的各种校验逻辑很多情况下都是大致相同或类似的，这样的实现方式会使得相似的校验逻辑代码被分散到了各个微服务中去，会带来很多冗余代码。所以，另一种方法是将这些校验逻辑剥离出去，构建出一个独立的鉴权服务。

在完成了剥离之后，如果直接在微服务应用中通过调用鉴权服务来实现校验，但是这仅仅只是解决了鉴权逻辑的分离，并没有在本质上将这部分不属于业余的逻辑拆分出原有的微服务应用，冗余的拦截器或过滤器依然会存在。

更好的做法是通过前置的网关服务来完成这些非业务性质的校验。由于网关服务的加入，外部客户端访问我们的系统已经有了统一入口，由于这些校验与具体业务无关，我们可以在请求到达的时候就完成校验和过滤，而不是转发后再过滤而导致更长的请求延迟。同时，通过在网关中完成校验和过滤，微服务应用端就可以去除各种复杂的过滤器和拦截器了，这使得微服务应用的接口开发和测试复杂度也得到了相应的降低。

2） 认证过滤

当客户端第一次请求服务时，服务端对用户进行信息认证（登录）

认证通过，将用户信息进行加密形成token，返回给客户端，作为登录凭证

以后每次请求，客户端都携带认证的token

服务端对token进行解密，判断是否有效

**public** **class** AuthorizeFilter **implements** GlobalFilter, Ordered {

    @Override

**public** Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

        String token = exchange.getRequest().getQueryParams().getFirst("token");

        // 判断token不为空才能正常通过登录验证，否则失效

        // 判断token的值是否为admin，否则不能成功登录，登录失效

**if**(!(token.equals("admin"))){

            System.out.println("token error, please retry.");

            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);

**return** exchange.getResponse().setComplete();

        }

**return** chain.filter(exchange);

    }

    @Override

**public** **int** getOrder() {

**return** 0;

    }

}

3) 网关限流

基于请求路径的限流

**public** KeyResolver PathKeyResolver(){

**return** exchange -> Mono.just(

                exchange.getRequest().getPath().toString()

    );

基于请求ip地址的限流

**public** KeyResolver IpKeyResolver(){

**return** exchange -> Mono.just(

                exchange.getRequest().getHeaders().getFirst("X-Forwarded-For")

        );

基于请求用户的限流

**public** KeyResolver UserKeyResolver(){

**return** exchange -> Mono.just(

                exchange.getRequest().getQueryParams().getFirst("user")

）；

3)本次实验实现了登录验证，用户通过网关调用微服务时必须输入指定的token和密码才能得到相应的微服务提供的数据。

图片包含 游戏机, 截图

描述已自动生成

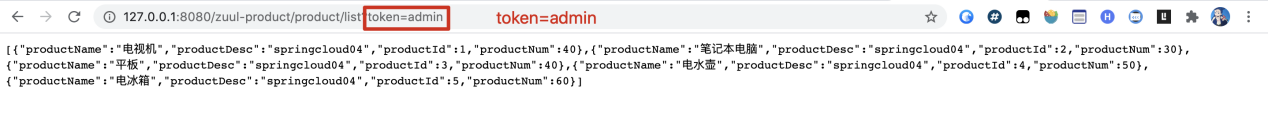
如上图所示，只有当token=admin时访问才会成功，否则都会报错，得不到正确的微服务。运行结果如下：

图片包含 游戏机, 截图

描述已自动生成

图片包含 游戏机, 截图

描述已自动生成



10. 安全（Security）

Eureka控制台页面默认没有安全控制，这样会带来以下三个问题：

可以看到各服务的状态和参数等信息，如果不加控制，只要知道注册中心的地址，就可以登录上去看到各服务信息；

只要知道注册中心地址，服务提供者就可以注册上来，对外提供服务；

只要知道注册中心地址，服务消费者就可以发现注册中心的服务，并调用服务；

恶意注册一个同名服务，但是实现不同；

因此我们加入了安全控制功能，将Eureka注册中心保护起来，用spring.security框架加入认证功能。

2）  spring cloud Eureka server 相关设置

pom.xml 加入secirity支持

 <dependency>

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

    <artifactId>spring-boot-starter-security</artifactId>

</dependency>

application.properties 设置改为

server.port=8888

eureka.instance.hostname=centos7

eureka.client.register-with-eureka=**false**

eureka.client.fetch-registry=**false**

eureka.client.serviceUrl.defaultZone=url

management.endpoints.web.exposure.include=\*

spring.security.user.name=admin

spring.security.user.password=123456

增加WebSecurityConfig 认证配置类

因为高版本的spring cloud中没有  security.basic.enabled=true这个设置项，为了能够接受客户端认证请求，在Eureka server需要增加这个类

@Configuration

@EnableWebSecurity

**public** **class** WebSecurityConfiguration **extends** WebSecurityConfigurerAdapter {

    @Override

**public** **void** configure(AuthenticationManagerBuilder auth)

**throws** Exception {

        auth.inMemoryAuthentication().passwordEncoder(**new** BCryptPasswordEncoder()).withUser("root").password(**new** BCryptPasswordEncoder().encode("123456")).roles("USER").

                and().withUser("admin").password(**new** BCryptPasswordEncoder().encode("123456")).roles("USER", "ADMIN");

    }

3）  Eureka client 客户端的配置

pom.xml 同样加入security支持

配置文件  application.properties

server.port=8080

eureka.client.serviceUrl.defaultZone=url

spring.application.name=SpringDemo

spring.security.user.name=admin

spring.security.user.password=123456

4） 运行测试，分别启动Eureka服务和客户端

需要登录进入Eureka界面

图片包含 游戏机, 截图

描述已自动生成

输入账号密码后，可见客户端注册成功

