Springcloud注册中心：拉取或注册服务信息

Springcloud配置中心：拉去配置信息

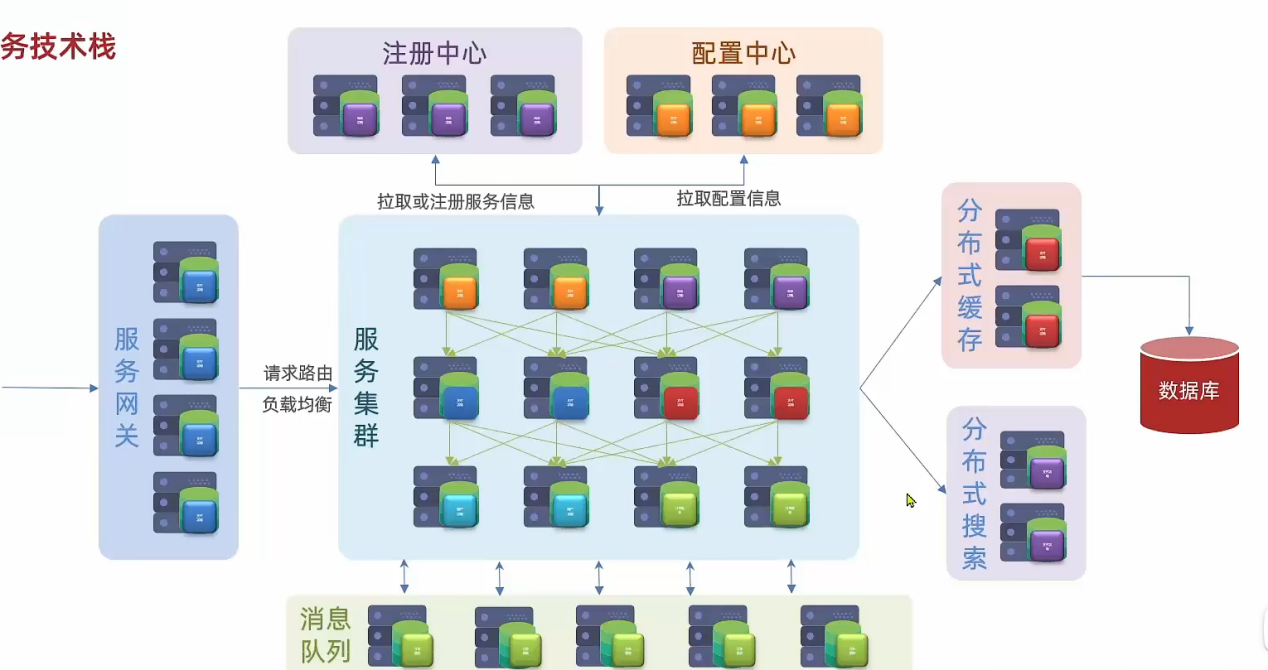
Springcloud微服务网关：请求路由，负载均衡

分布式缓存：集群缓存

分布式搜索：缓存搜索

数据库：集群数据库

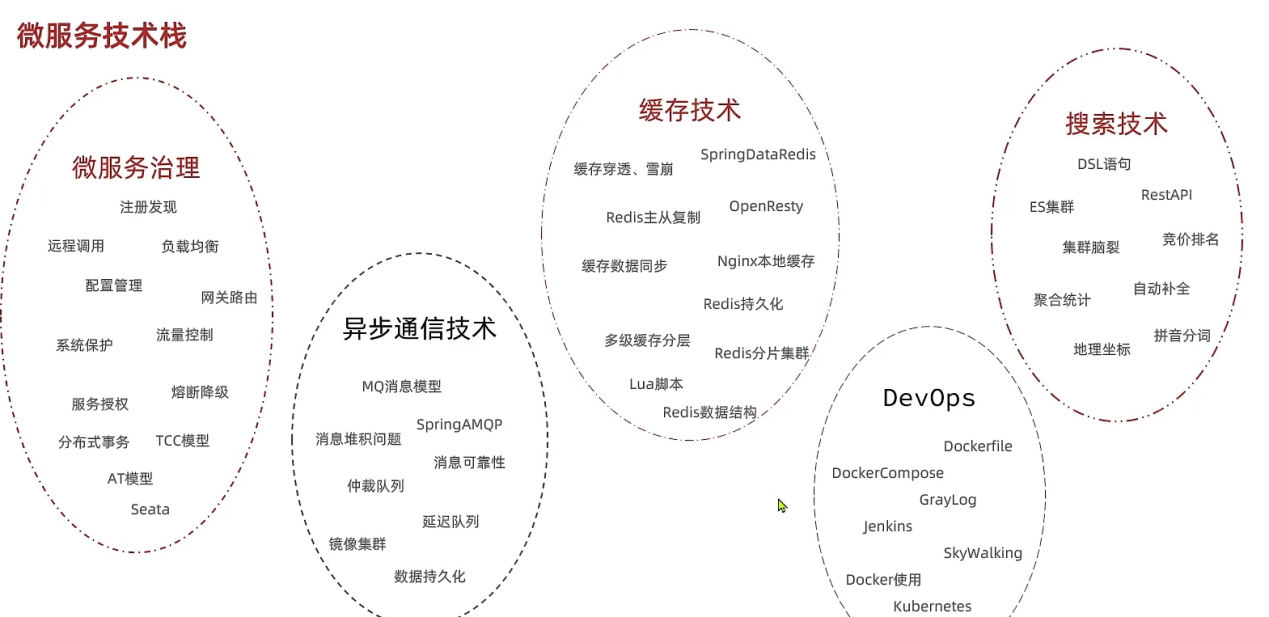
消息队列：异步通讯



分布式日志服务：可以记录日志统一存储分析

系统监控链路追踪：监控整个集群中每一个服务中的运行状态

自动化部署：Jenkins



**认识微服务**

之前：

**单体架构=>**将业务的所有功能集中在一个项目开发，打成一个包部署

优点：架构简单，部署成本低，

缺点：耦合度高

SpringCloud：

**分布式架构=>**根据业务功能对系统进行拆分，每个业务模块作为独立项目开发，称为一个服务

优点：

降低服务耦合

有利于服务升级拓展

缺点：部署成本高，

**微服务=>**微服务是一种经过良好架构设计的分布式架构方案，

**微服务架构特征**：

单一职责：微服务拆分粒度更小，每一个服务都对应唯一的业务能力，做到单一职责，避免业务的重复开发。

面向服务：微服务对外暴露业务接口

自治：团队独立，技术独立，数据独立，部署独立

隔离性强：服务调用做好隔离，容错，降级，避免出现级联失败

**微服务技术对比：**

**Dubbo:**

注册中心=>{zookeeper=>集群管理,Redis=>缓存}并不具备完善的注册中心作用

服务远程调用=>Dubbo协议（核心）。

配置中心：无

服务网关：无

服务监控和保护：dubbo-admin，功能弱

**Springcloud：**

注册中心=>Eureka,Consul

服务远程调用=>Feign（http协议）

配置中心：SpringCloudConfig

服务网关：SpringCloudGateWay,zuul

服务监控和保护：Hystrix(核心是保护)

**SpringCloudAlibaba:**

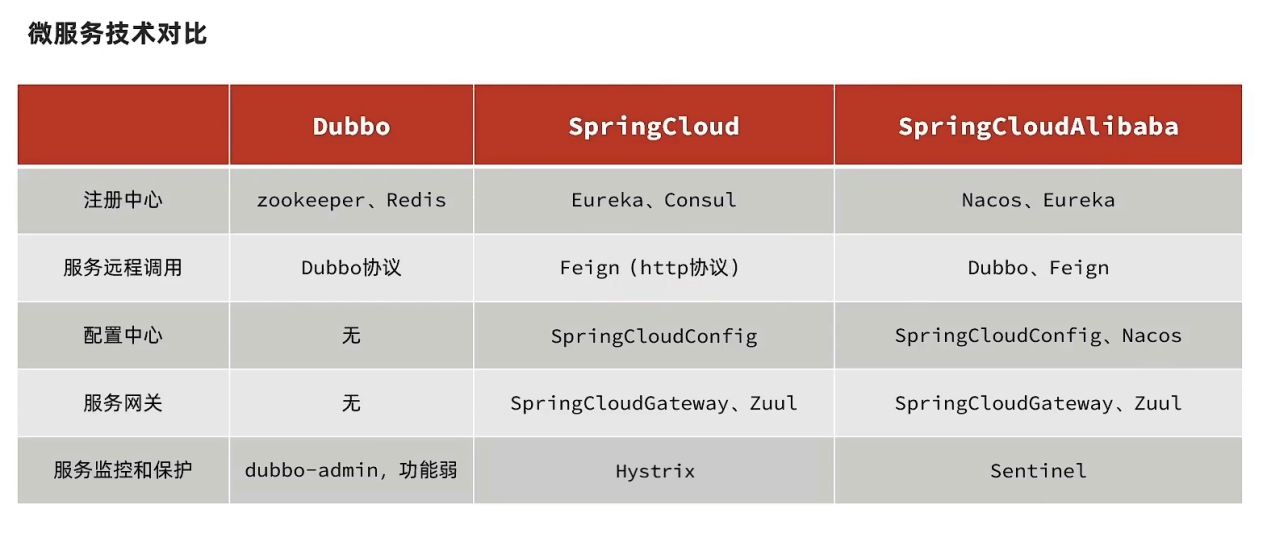
注册中心：Nacos,Eureka

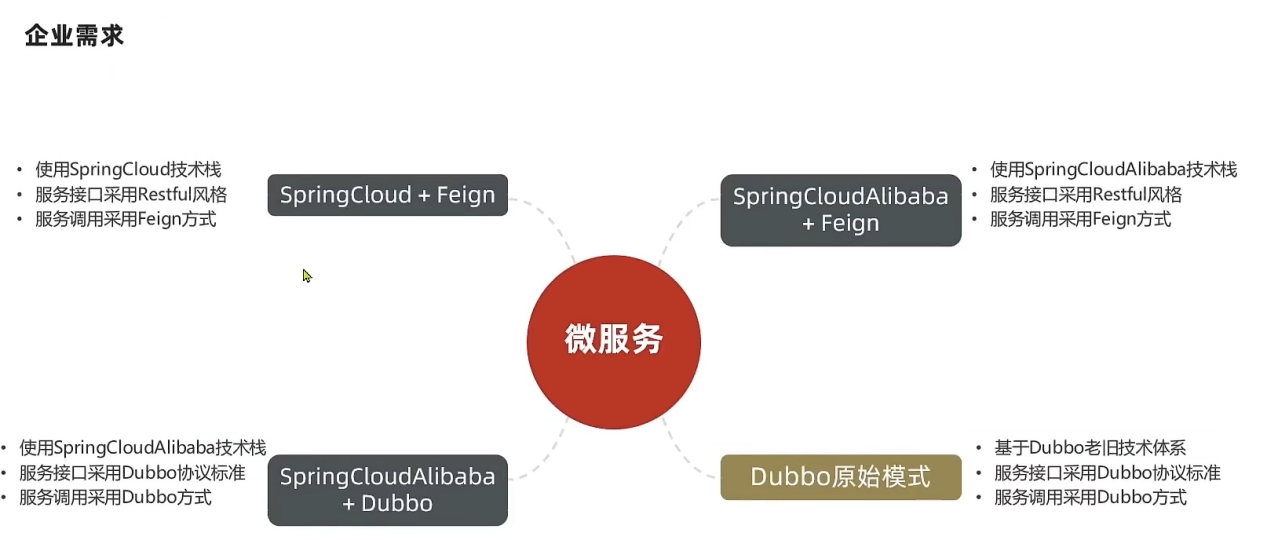
服务远程调用：Dubbo，Feign

配置中心：SpringCloudConfig，Nacos

服务网关：SpringCloudGateway,Zuul

服务监控和保护：Sentinel





**SpringCloud=>https://spring.io/projects/spring-cloud**

**springCloud集成了各种微服务的功能，并且基于springboot实现了组建的自动装配，从而提供了良好的开箱即用体验。**



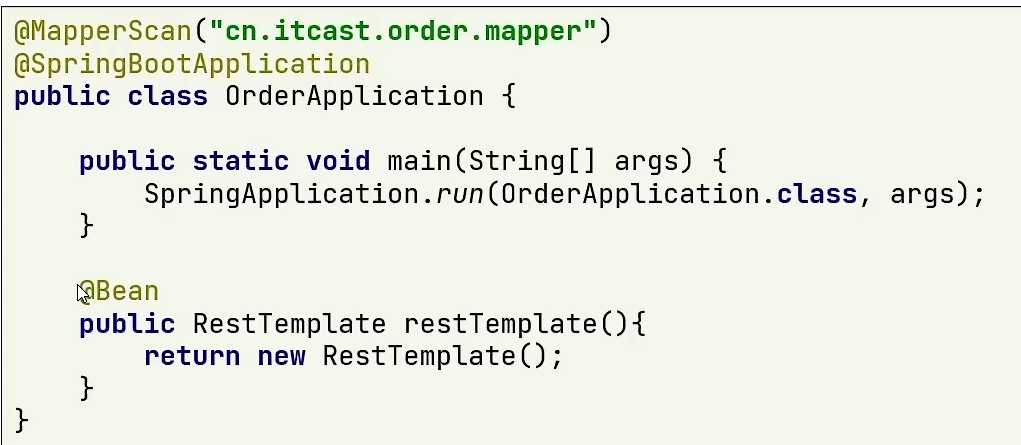


**服务拆分注意事项：**

1. 不同微服务，不要重复开发相同业务
2. 微服务数据独立，不要访问其他微服务的数据库
3. 微服务可以将自己的业务暴露成接口，供其他微服务引用

**一、微服务远程调用：**

1. 注册RestTemplate



1. 利用RestTemplate发起http请求

2.1

String url = 路径+get传参

2.2

restTemplate.getForObjct(url,返回值类型);

1. 封装返回对象
2. 返回

**二、提供者与消费者：**

服务提供者=>一次业务中，被其他微服务调用的服务。（提供接口给其他微服务）。

服务消费者=>一次业务中，调用其他微服务的服务（调用其他微服务提供的接口）。

A调B,B调C，B是什么角色？

一个服务既可以是提供者，也可以是消费者

1. **Eureka注册中心**

**Eureka的作用**

**服务消费者该如何获取服务消费者的地址信息？**

·服务提供者启动时像eureka注册自己的信息

·eureka保存这些信息

·消费者根据服务名称像eureka拉去提供者信息

**如果有多个服务提供者，消费者该如何选择？**

·服务消费者利用负载均衡算法，从服务列表中挑选一个

**消费者如何得知服务提供者的健康状态？**

·服务提供者会每隔30秒像EurekaServer发送心跳请求，报告健康状态

·eureka会更新记录服务列表信息，心跳不正常会被剔除

·消费者就可以拉去到最新的信息

**在Eureka架构中，微服务角色有两类：**

**EurekaServe:服务端，注册中心**

·记录服务信息

·心跳监控

**EurekaClient:客户端**

·provider：服务提供者

■注册自己的信息到EurekaServer

■每隔30秒像EurekaServer发送心跳

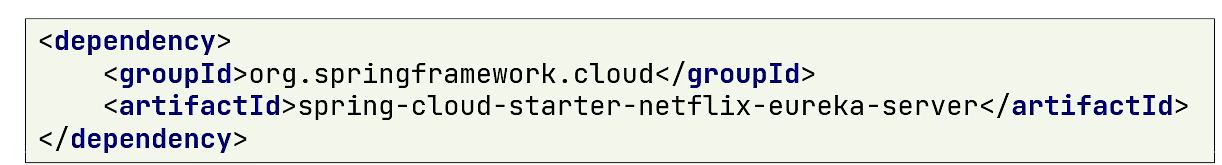
·consumer：服务消费者

■根据服务名称从EurekaServer拉取服务列表

■基于服务列表做负载均衡，选中一个微服务后发起远程调用

**搭建EurekaServer服务步骤：**

1. **创建项目，引入spring-cloud-starter-netflix-eureka-server**



1. 编写启动类，添加@EnableEurekaServer
2. 添加application.yml



一个服务每部署一次就是一个实例

**服务注册**

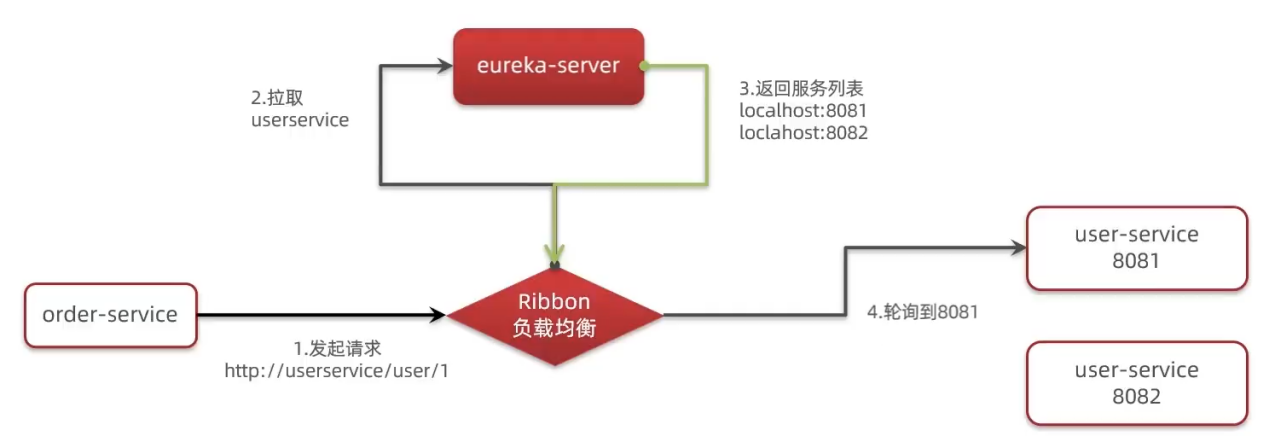
引入Eureka-client依赖

在yml中配置Eureka地址

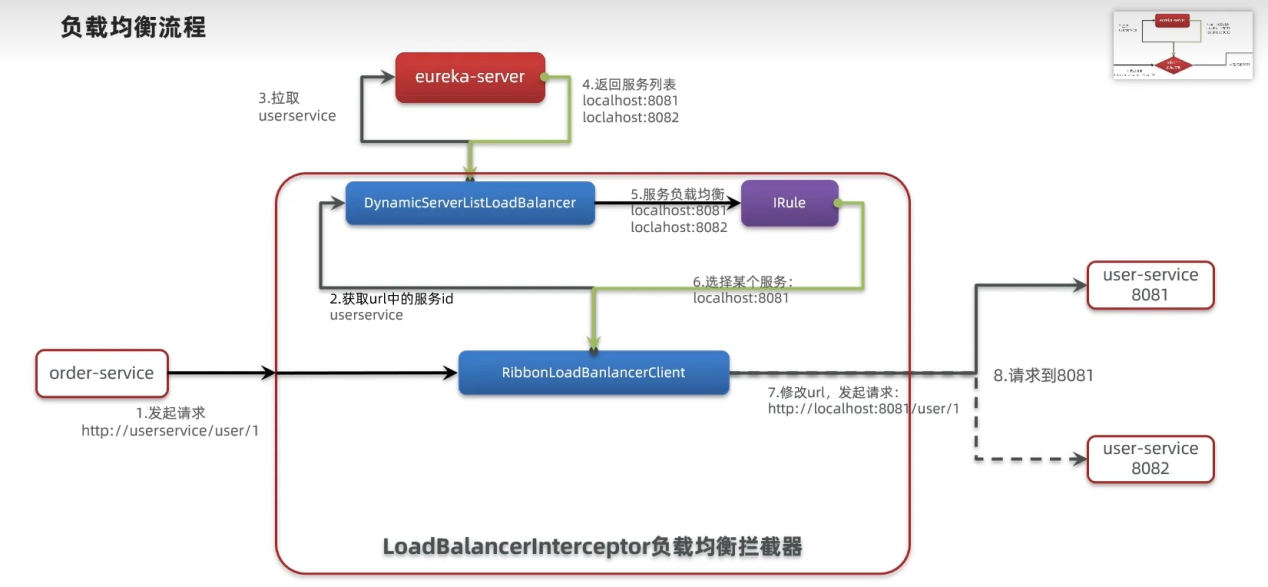
※无论是消费者还是提供者，引入eureka依赖，知道eureka地址后都会被当作服务被注册

**负载均衡**

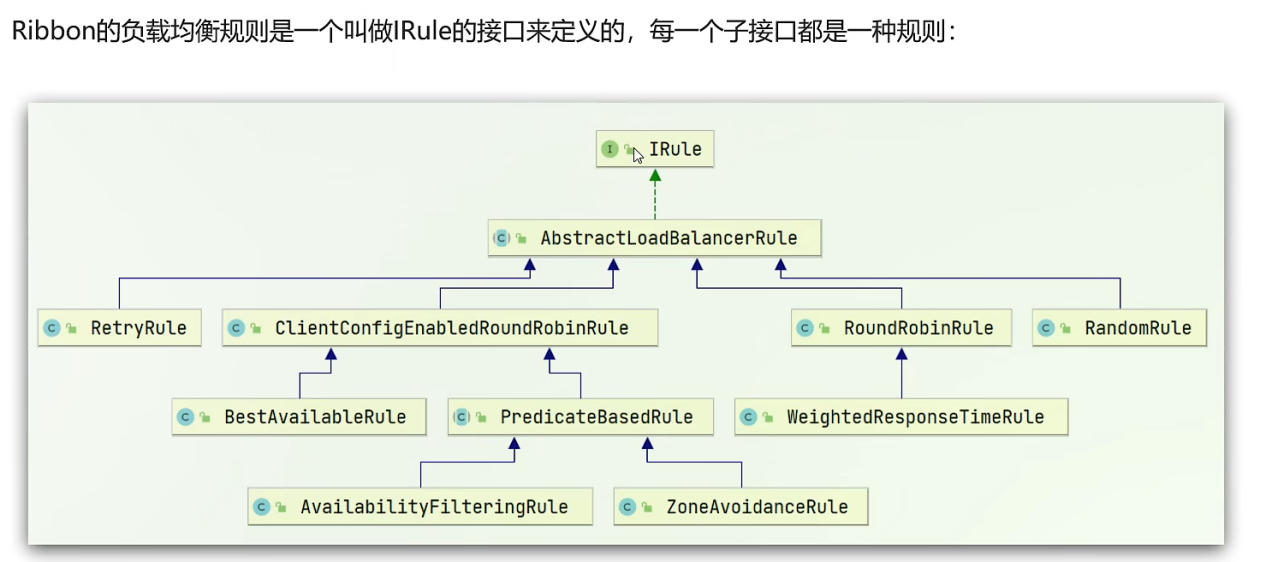
基于Ribbon组件实现



当请求进入到Ribbon，请求会被LoadBalancerInterceptor负载均衡拦截器拦截，会得到请求中的请求名称，将拦截下来的请求名称交给RibbonLoadBalancerClient,然后将url中的服务id交给DynamicServerListLoadBalancer去eureka中拉取服务，得到多个服务信息，然后DynamicServerListLoadBalancer会找到Irule，Irule会根据规则找到负载均衡的服务，再将服务交给RibbonLoadBalancerClient,然后会根据返回的端口替换服务名称，得到真实的请求名称，请求服务成功



IRule接口的实现



默认是ZoneAvoidanceRule

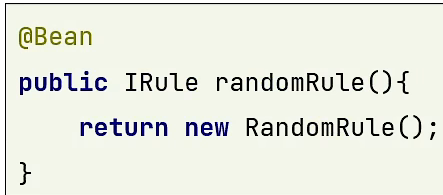
**常见的IRule实现**



**调整负载均衡的两种方式**

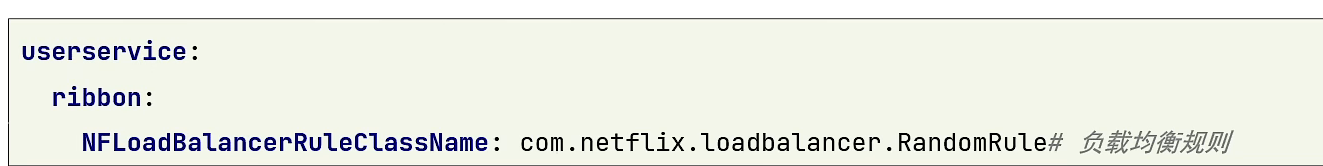
**一、**

代码方式=>在orderService的启动类中，加入一个新的IRule，发布需要重新打包



二、

配置方式=>在oderService的yml文件中添加新配置修改规则，发布不需要重新打包



**Ribbon的饥饿加载**

**Ribbon默认采取懒加载，**即第一次访问时才会创建LoadBalanceClint，请求时间会长，而饥饿加载会在项目启动时创建，降低第一次访问的速度，



Nacos注册中心

Nacos是alibaba的产品，现在是springcloud中的一个组件，相比eureka功能更加丰富，

**Nacos服务分级存储模型**

**服务=>集群=>实例**

服务调用尽可能选择本地集群的服务，跨集群调用延迟较高

本地集群不可访问时，再去访问其他集群服务

**服务集群属性**

修改application.yml文件，添加集群属性



**NacosRule负载均衡**

①优先选择同集群的服务

②本地找不到服务提供者才会去其他集群寻找服务实例，并且会报警告

③确定可用实例后，会采用随机负载均衡挑选实例

**根据权重负载均衡**

**实际部署中会出现这样的场景：**

服务器设备性能有差异，部分实例所在机器性能比较好。另一些较差，我们希望性能好的机器能多承担一些用户请求=>Nacos提供了权重配置来控制访问频率，权重越大则访问频率越高，=>设置权重大小

Nacos控制台可以控制实例的权重值，通常在0~1之间

同集群内的实例权重值越高被访问的频率就越高

权重设置为0就不会被访问

**Nacos环境隔离 -namespace**

Nacos不仅是一个注册中心还是一个数据中心

Nacos中服务存储和数据存储的最外层都是一个名为namespace的东西，用来做最外层隔离

Namespace配置：在yml文件中配置namespace：Id

不同的namespace的服务是不可见的

Nacos和Eureka区别

Nacos=> 对于临时实例来说，每隔一段时间会对Nacos注册中心发送一次心跳，挂掉会被直 接被剔除

对于非临时实例来说，每隔一段时间nacos注册中心会询问一次，并且不会从列表 中提出，但是会标记服务实例不健康

Nacos会主动推送服务的状态更新，pull+push

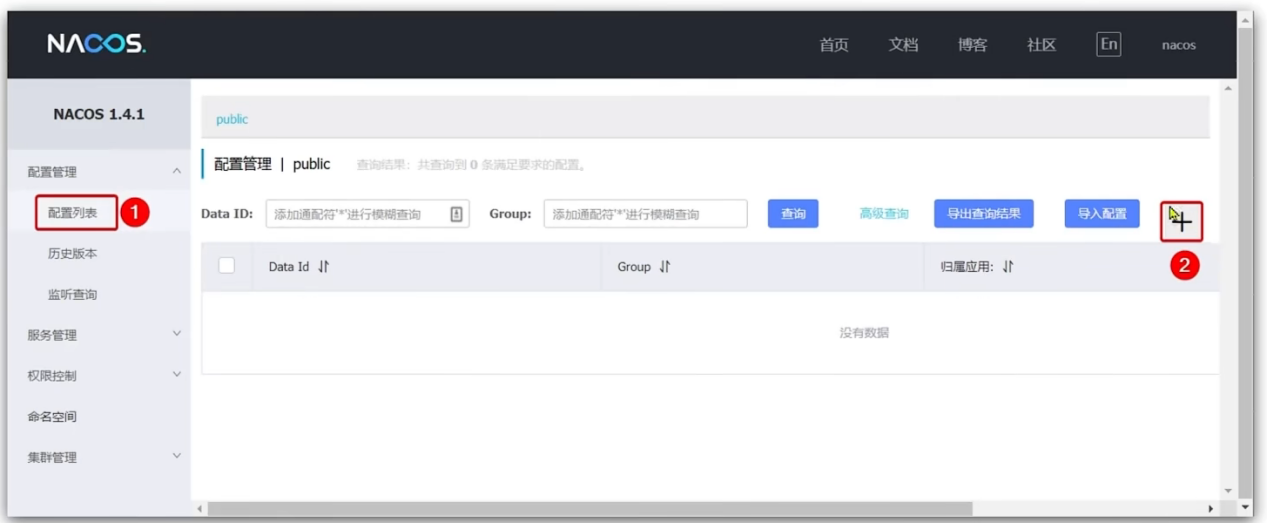
Eureka=>每隔三十秒像eureka注册中心拉去服务，在30秒内服务挂掉，感应不及时，eureka只能pull

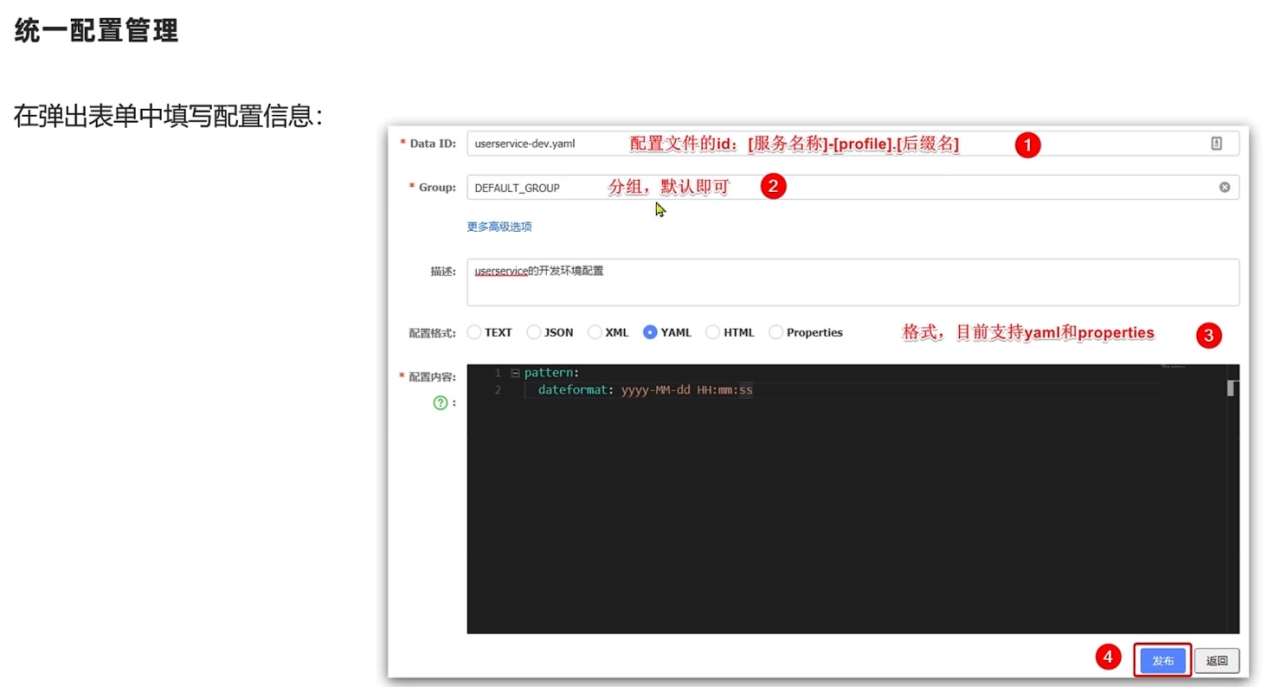
在配置文件中可以设置当前实例是否是临时实例：ephemeral：false（默认是true）临时实例

**统一配置管理**

**·配置更改热更新**

在nacos中添加配置信息





**将配置交给nacos管理**

1. 在nacos中添加配置

2.引入nacos的配置客户端管理依赖

3.在resource中写bootstrap.yml文件，优先级高于application.yml文件=>提前获取nacos地址，配置服务名称，环境，后缀名

**Nacos中的配置热更新**

**方法一、**

在@value类上加@RefreshScope

**方法二、**

通过@ConfigurationProperties注入，自动刷新

**注意事项**

**不是所有的配置都适合放在配置中心，维护起来比较麻烦**

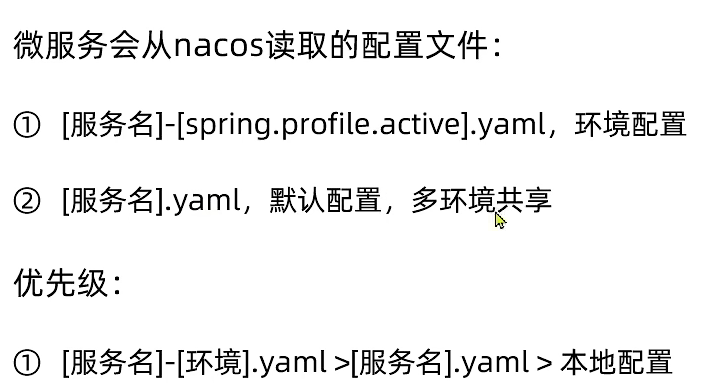
**建议将一些关键参数，需要运行时调整的参数放到nacos配置中心，一般都是自定义配置**

**Nacos多环境配置共享**

**微服务在启动时会从nacos读取多个配置文件：**

环境配置+服务配置

环境配置>服务配置>本地配置



**搭建nacos集群**

1. 搭建nacos集群并初始化数据库表
2. 修改nacos集群配置，数据库集群配置
3. 分别启动多个nacos节点
4. Nginx反向代理

**RestTemplate方法调用存在问题**

**·代码可读性差，编程体验不统一**

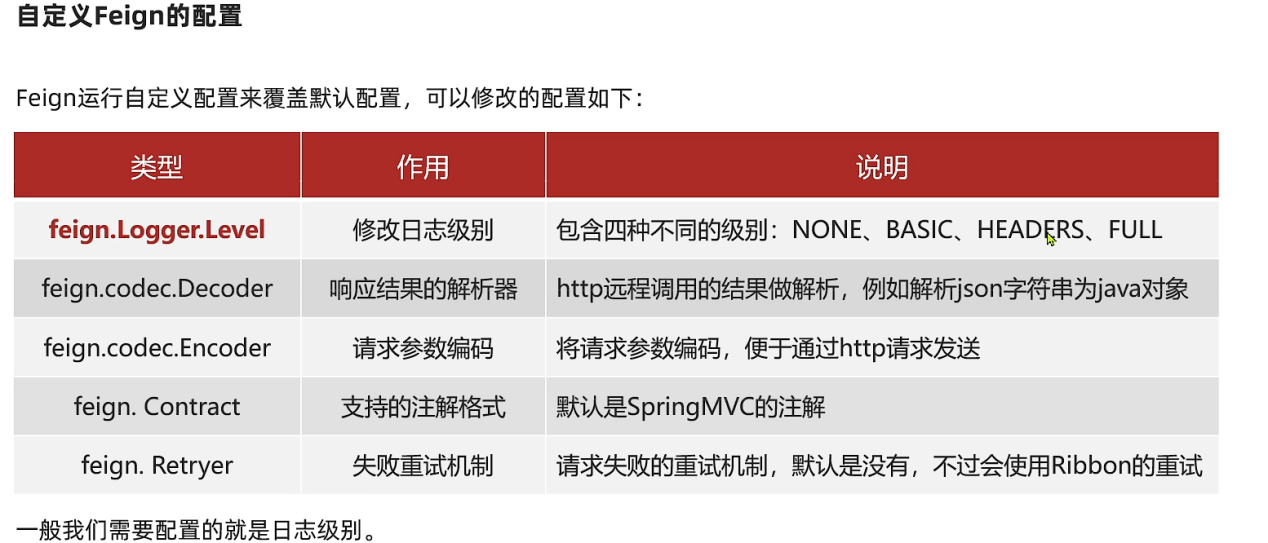
**·参数复杂的url难以维护**

**Feign是一个声明式的http客户端**

**作用=>帮我们优雅的发送http请求**

**步骤：**

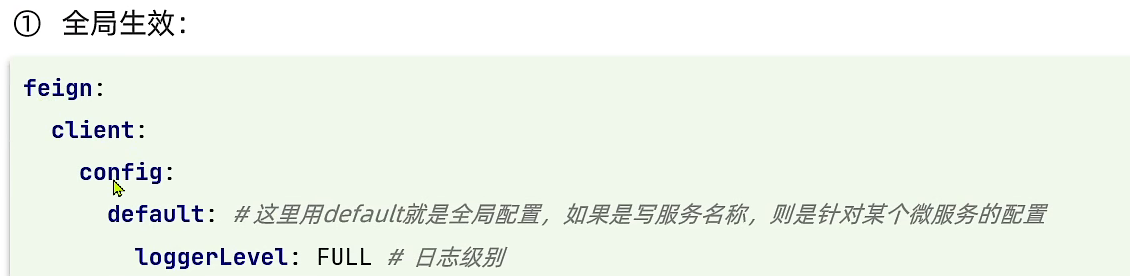
1. 加入feign的依赖
2. 在消费者启动类中添加注解开启feign的功能@EnableFeignClients
3. 编写feignclient接口=>类上加@FeignClient(“服务名称”)，建议一个服务写一个feignclient接口，方法上按照springmvc-restful风格配置方法和参数
4. 调用接口中封装的方法
5. Feign中集成了ribbon实现了负载均衡

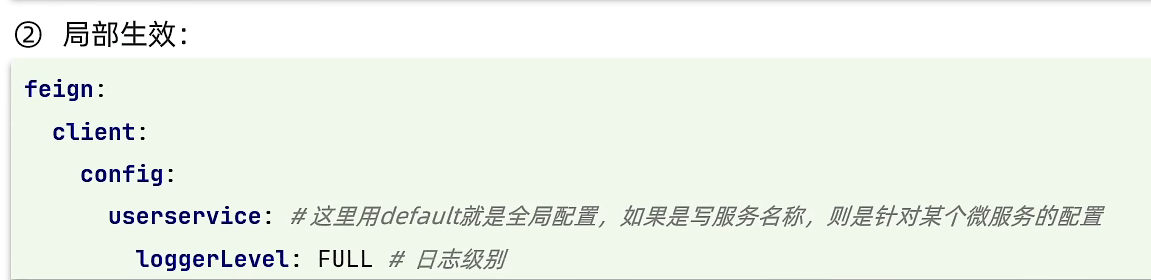


**自定义feign的配置**

配置feign日志有两种方式;

1. 方式一=>配置文件





1. 方式二=>java代码



Feign的性能优化

Feign的底层的客户端实现：

URLConnection:默认实现，不支持连接池

Apache HttpClient：支持连接池

OKHttp：支持连接池

优化feign的性能主要包括：

①使用连接池代替默认的URLConnection

②日志级别，最好用basic或none

**Feign的最佳实践**

优化实现：

1.0引入httpclient的依赖

2.0配置连接池



**Feign的最佳实践**

**方法一（继承）：给消费者的FeignClient和提供者的controller定义统一的父接口作为标准**

服务紧耦合，父接口参数列表中的映射不会被继承

方法二（抽取）：将feignclient抽取为独立模块，并且把接口有关的pojo，默认的feign配置都放在这个模块中，提供给所有的消费者

将服务提供者所提供的api pojo feign的配置打包，服务消费者引入包的依赖



弊端=>所有的代码都会被引入

当定义的feignclient不在springbootApplication的扫描包范围时，这些FeignClient无法使用，

方法一：指定feignclient所在包

@EnableFeignClients(basePackages=””)

方法二：指定feignclient的字节码

@EnableFeignClients(clients={接口.class})

**网关功能：**

·身份认证和权限校验

·服务路由、负载均衡

·请求限流

**Gateway**

**Zuul**

**区别：**

Zuul是基于servlet的实现，属于阻塞式编程。而SpringCloudGateway则是基于Spring5中提供的webflux，属于响应式编程的实现，具备更好的性能

搭建网关服务的步骤：

1. 创建新的module引入SpringCloudGateway的依赖，和nacos服务发现依赖=>网关也是一种服务也需要注册到nacos也需要从nacos上拉去服务
2. 编写路由配置以及nacos地址

Gateway：

routes：

-id：唯一ID，任意

uri：<http://127.0.0.1:8080> =>目标地址 不建议这么写http 固定的地址

uri：lb://userservice =>目标服务 lb=>loadbalance负载均衡

Predicates: 路由断言，也就是判断请求是否符合路由规则的 条件

-path=/user/\*\* 这个是按照路径匹配，只要是以/user/开头 就符合要求

filters：

- addrequestheader=truth,Itcast is freakig awesome

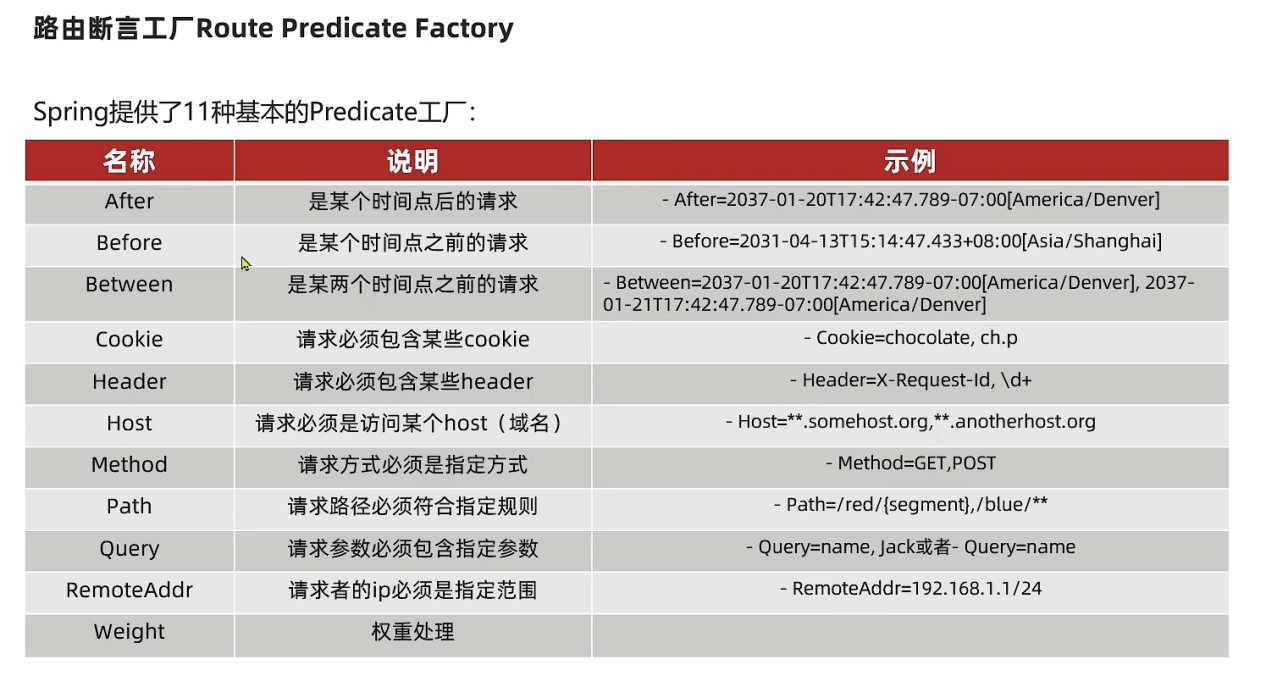
Default-filters: #会对所有请求都生效 需要和route同级，因为是全局过滤器

路由过滤器，处理请求或响应

**路由断言工厂**

·我们在配置文件中写的断言规则只是字符串，这些字符串需要被断言工厂读取并处理，转变为路由判断的条件

·例如Path=/user/\*\*是按照路径匹配，这个规则是由pathrouteperdicateFactory类处理的



**网关过滤器**

GatewayFilter是co中提供的一种过滤器，可以对进入网关的请求和微服务返回的相应做处理

全局过滤器GlobalFilter

全局过滤器的作用也是处理一切进入网关的请求和微服务相应，与Gatewayfilters的作用一样

区别在于GatewayFilter通过配置定义，处理逻辑是固定的。而GlobalFilter的逻辑需要自己写

定义方式=>实现GlobalFilter接口

重写filter方法

用ServerWebExchange.getRequest.getQueryparams()获取请求参数.getFirst()获取value

允许访问return chain.filter(exchange);

禁止访问 => exchange.getResponse().setStatusCode();

最后结束处理 return exchange.getResponse().setComplete();

添加@order给过滤器排序

全局过滤器的作用：对所有路由都生效的过滤器，并且可以自定义处理逻辑

过滤器的执行顺序：请求路由后，会将当前路由过滤器和DefaultFilter、GlobalFilter合并到一个过滤器链中，排序后依次执行每个过滤器

·每个过滤器都必须指定一个int类型的order值，order值越小优先级越高，执行顺序越靠前

·GlobalFilter通过实现Ordered接口，或者添加@Order注解来指定order值，由我们自己指定，

·路由过滤器和defaultFilter的order由spring指定，默认是按照生命顺序从1递增

·当过滤器的order值一样时，会按照defaultFilter>路由过滤器>GlobalFilter的顺序执行

网关的跨域问题处理

跨域：域名不一致就是跨域，主要包括：

①域名不同

②域名相同，端口不桐：localhost：8080和localhost:8081

跨域问题：浏览器禁止请求的发起者与服务端发生ajax请求，请求被浏览器拦截的问题

Docker

大型项目组件较多，运行环境较为复杂，部署时会碰到一些问题：

·依赖关系复杂，容易出现兼容性问题

·开发、测试、生产环境有差异

Docker如何解决依赖的兼容问题？

·将应用的libs(函数库)、deps(依赖)、配置与应用一起打包

·将每个应用放到一个隔离的容器去运行，使用沙箱机制，避免互相干扰

Linux系统

内核与硬件交互，提供操作硬件的指令

系统应用封装内核指令为函数，便于程序员调用

用户程序基于系统函数库实现功能

Docker如何解决不同系统环境的问题？

·Docker将用户程序与所需要调用的系统(比如Ubuntu)函数库一起打包

·Docker运行到不同操作系统时，直接基于打包的库函数，借助于操作系统的Linux内核来运行

Docker是一个快速交付应用、运行应用的技术：

1. 可以将程序及其依赖，运行环境，函数库打包成一个可移植的镜像，可以迁移至Linux任意的操作系统上
2. 运行时利用沙箱机制形成隔离容器，各个应用互不干扰
3. 启动和移除都可以通过一行指令完成，方便快捷

Docker与虚拟机

虚拟机是指在操作系统中模拟硬件设备，然后运行另一个操作系统，比如在windows系统上运行Ubuntu系统，这样就可以运行任意的Ubuntu应用了

Docker是一个系统进程；虚拟机是在操作系统中操作系统

Docker体积小，启动快，性能好。虚拟机体积大，启动慢，性能一般

镜像和容器

镜像：Docker将应用程序及其所需的依赖、函数库、环境、配置等文件打包在一起，称为镜像

容器：镜像中的应用程序运行后形成的进程就是容器，只是dockers会对容器进行隔离，对外不可见

Docker和DockerHub

DockerHub：DockerHub是一个Docker镜像的托管平台，这样的平台成为DockerRegistry

国内也有类似于DockerHub的公开服务，比如网易云镜像服务、阿里云镜像库等

Docker是一个CS架构的程序，由两部分组成：

※服务端(server)：Docker守护进程，负责处理Docker指令，管理镜像、容器等

※客户端(client)：通过命令或RestAPI向Docker服务端发送指令。可以在本地或者远程像服务端发送指令

**镜像相关命令**

·镜像名称一般分两部分组成：仓库：版本

不写版本默认最新版本

Docker images 查看镜像

Docker rmi 删除镜像

Docker push 推送镜像到服务

Docker pull 从服务上拉镜像

Docker save 把docker镜像压成压缩包

Docker load 把docker压缩包解压成镜像

**Docker容器命令**

Docker run 创建容器，并且运行

Docker pause 容器暂停

Docker unpause 容器从暂停到运行

Docker stop 容器停止

Dockers start 容器从停止到开始

Docker ps 查看容器运行状态

Docker logs 查看容器运行log

Docker exec 进入容器执行命令

Docker rm 删除容器

创建运行一个Nginx容器

Docker run --name containerName -p 80:80 -d nginx

Docker run：创建运行一个容器

--name ：给容器起一个名字，比如containerName

-p:将宿主机端口和容器端口映射，冒号左边是宿主机端口，右边是容器端口

-d：后台运行容器

Nginx：镜像名称

Docker数据卷

容器和数据耦合的问题

1. 不便于修改=>需要进入容器内部修改，很不方便
2. 数据不可复用=>在容器内的修改对外是不可见的。所有修改对新创建的容器都是不可复用的，
3. 升级维护困难=>数据在容器内，如果要升级容器就必须删除旧容器，所有数据就跟着删除掉

数据卷=>是一个虚拟目录，指向宿主机文件系统中的某个目录

数据卷的操作基本语法

Docker volume[COMMAND]

Docker volume命令是数据卷操作，根据命令后跟随的command来确定下一步的操作

Create 创建一个volume

Inspect 显示一个或多个volume的信息

Ls 列出所有的volume

Prune 删除未使用的volume

Rm 删除一个或多个指定的volume

数据卷的作用：

将容器与数据分离，解耦合，方便操作容器内的数据，保证数据安全

挂载数据卷

我们在创建容器时，可以通过-v参数来挂在一个数据卷到某个容器目录

Docker run\--name mn\-v html:/root/html\-p 8080:80 nginx\

Docker run 中-v参数挂载文件或目录到容器中；

1. v volume名称：容器内目录
2. V 宿主机文件：容器内文件
3. v宿主机目录：容器内目录

数据卷挂载和目录直接挂载得区别

①数据卷挂载耦合性更低，由docker管理目录，但是目录较深不好找

②目录挂载耦合性较高，由我们自己管理目录，目录容易查找

镜像结构

·镜像是将应用程序机器需要的系统函数库、环境、配置、依赖打包而成

·基础镜像层=>应用依赖得系统函数库、环境、配置、文件等

·入口=>镜像运行入口，一般是程序启动的脚本和参数

·层(layer)=>在baseimage基础上添加安装包、依赖、配置等，每次操作都形成新的一层

**什么是Dockerfile**

Dockerfile就是一个文本文件，其中包含一个个的指令，用指令来声明要进行什么操作来构建镜像。每个指令都会形成一层layer

·FROM=>指定基础镜像

·EVN=>设置环境变量，可在后面指令使用

·COPY=>拷贝本地文件到镜像得指定目录

·RUN=>执行Linux得shell命令，一般是安装过程得命令

·EXPOSE=>指定容器运行时监听的端口，是给镜像使用者看的

·ENTRYPOINT=>镜像中应用的启动命令，容器运行时调用

Dockerfile的本质是一个文件，通过指令描述镜像的构建过程

Dockerfile第一行必须是from，从一个基础镜像开始构建

基础镜像可以是操作系统，如Ubuntu，也可以是其他人制作好的镜像，java:8-alpine.

DockerCompose

·Docker Compose可以基于compose文件帮我们快速的部署分布式应用，而无需一个个手动创建和运行容器

·compose 文件是一个文本文件，通过指令定义集群中的每隔容器如何运行

服务异步通讯

**同步通讯的优缺点：**

微服务基于feign的调用就属于同步

缺点：1.耦合度高，每次加入新的需求，都要修改原来的代码

2.性能下降=>调用者需要等待服务提供者相应，如果调用链过长则乡音时间等于每次调用的时间之和

1. 浪费资源=>调用链中的每个服务在等待响应过程中，不能释放请求占用的资源，高并发的场景下会极度浪费系统资源
2. 级联失败=>如果服务提供者出现问题，那么所有调用方都会出现问题，如同多米诺骨牌一样，迅速导致整个微服务群故障

优点：时效性较强，可以立即得到结果

**异步通讯的优缺点：**

异步调用常见实现就是事件驱动模式

优势：

1. 服务解耦
2. 性能提升，吞吐量提升
3. 服务没有强依赖，不担心级联失败
4. 流量消峰

缺点：

依赖于broker的可靠性、安全性、吞吐能力

架构复杂，业务没有明显的流程线，不好追踪管理

**MQ和MQ的常见框架**

MQ(messageQueue)，中文是消息队列，字面来看就是存放消息的队列，也就是事件驱动架构中的broker。

**RabbitMQ：**

社区=>Rabbit

开发语言=>Erlang

协议支持=>AMQP.XMPP.SMTP.STOMP

可用性=>高

单机吞吐量=>一般

消息延迟=>微秒级

消息可靠性=>高

**ActiveMQ：**

社区=>Apache

开发语言=>java

协议支持=>OpenWire,STOMP,REST,XMPP,AMQP

可用性=>一般

单机吞吐量=>差

消息延迟=>毫秒级

消息可靠性=>一般

**RocketMQ：**

社区=>阿里巴巴

开发语言=>java

协议支持=>自定义协议

可用性=>高

单机吞吐量=>高

消息延迟=>毫秒级

消息可靠性=>高

**Kafka：**

社区=>Apache

开发语言=>java(调用API)&Scala(基于java)

协议支持=>自定义协议

可用性=>高

单机吞吐量=>非常高

消息延迟=>毫秒以内

消息可靠性=>一般

**RabbitMQ概述=>**RabbitMQ是基于Erlong语言开发的开源消息中间件

服务提供成会把服务发送到exchange(交换机)，交换机负责路由，再把消息传递给队列，队列负责暂存消息，消费者负责去队列中获取消息，处理消息

Channel：操作mq的工具

Exchange：路由消息队列中

Queue：缓存消息

Virtual host：虚拟主机，是对exchange和queue等资源的逻辑分组

常见消息模型

·基本消息队列

·工作消息队列

·发布订阅，又根据交换机类型不同分为三种

☆ Fanout Exchange：广播

☆ Direct Exchange：路由

☆ Topic Exchange：主题

基本消息队列只包括三个角色：消息发送者，队列，服务消费者

Publisher：服务发布者，将消息发送到队列queue

Queue：消息队列，服务接受并缓存消息

Consumer：订阅队列，处理队列中的消息

基本消息队列的消息发送流程：

1. 建立connection连接
2. 创建channel
3. 利用channel声明队列
4. 利用channel向队列发消息

基本消息队列的消息接受流程

1.建立connection连接

2.创建channel

3.利用channel声明队列

4.定义consume的消费行为handleDelivery()

5.利用channel将消费者和队列绑定在一起

**SpringAMQP**

AMQP=>高级消息队列协议，是用于在应用程序或之间传递业务消息的开放标准。该协议与语言和平台无关，更符合微服务种独立性的要求

SpringAMQP是基于AMQP协议定义的一套API规范，提供了模板来发送和接受消息。其中包含两部分->spring-amqp是基础抽象，spring-rabbit是默认实现

**BasisQueue**

**使用AMQP发送消息**

①引入spring-starter-amqp的依赖

②配置amqp地址

③注入RabbitTemplate

④用convertAndSend像队列发送消息

**使用AMQP消费消息**

①引入amqp依赖

②配置amqp地址

③建一个类交给spring容器管理

④用@RabbitListener(queues=”队列名”)来读取指定的队列

⑤throws InterruptedException

workQueue工作队列

WorkQueue 工作队列，可以提高消息处理速度，避免队列消息堆积

消息预取限制

修改application.yml文件，设置preFetch这个值可以控制预取消息的上限

Work模型的使用：

·多个消费者绑定到一个队列，同一个消息只会被一个消费者处理

·通过设置prefetch来控制消费者预取的消息数量

**发布、订阅**

发布订阅模式就是允许将统一消息发送给多个消费者。实现方式就是加入exchange(交换机)

交换机类型

Fanout:广播

Direct：路由

Topic：话题

※交换机负责消息路由，而不是存储，路由失败则消息丢失

Fanoutexchange会将收到的消息路由到每一个跟其绑定的队列上

交换机的作用=>接受publisher发送的消息、将消息按照规则路由到与其绑定的队列上、不能缓存数据，路由失败，消息丢失

声明队列、交换机、绑定关系的bean？

Queue Exchange Binding

**Direct Exchange**

·会将消息根据规则路由到指定的Queue

·每一个Queue都会与DirectExchange设置一个BindingKey

·发布者发布消息时，指定消息的RoutingKey给路由

·exchange将消息路由到BindingKey与消息RoutingKey一致的队列

TopicExchange

·topicExchange与DirectExchange类似，区别在于RoutingKey必须是多个单词的列表，并且.分割

·Queue与Exchange绑定BindingKey时可以使用通配符

#：代指0个或多个单词

\*：代指一个单词

**消息转换器：**

Spring对消息对象的处理是由messageConvert来处理的，而默认实现市SimpleMessageConvert，基于ObjectOutPutStream完成序列化。如果要修改只需要定义一个MessageConvert得bean就可以，推荐用josn序列化

※发送方和接收方必须用得同样的序列化配置

**分布式搜索**

**Elasticsearch=>**是一款非常强大的开源搜索引擎，可以帮助我们从海量数据中快速找到需要的内容

Elasticsearch结合kibana、logstash、beats，也就是elastic stack（ELK）。被广泛运用于日志数据分析、实时监控等领域

Elasticsearch是elastic stack得核心，负责存储、搜索、分析数据

Kibana=>数据可视化

Logstash、beat=>数据抓取

Es底层实现是=>lucene

lucene是java语言得搜索引擎类库，

优势=>易扩展、高性能

缺点=>只限于java开发

正向索引和倒排索引

传统数据库采用正向索引

Es采用倒排索引

·文档（document）：每条数据都是一个文档

·词条（term）：文档按照语义分成的词语

Es是面向文档存储的：

·可以是数据库得一条商品数据，订单数据文档数据会被序列化为json格式后存储在es里

·索引=>相同类型得文档得集合

·映射=>索引中文档的字段约束信息，类似于表的结构约束

Mysql：擅长事务类型操作，可以确保数据的安全性和一致性

Es：擅长海量数据得搜索、分析、计算

**IK分词器**

作用=>

·创建倒排索引的时候对文档进行分词

·用户搜索的时候对输入的内容分词

**IK分词器有几种模式**

·ik\_smart：智能切分，粗粒度

·ik\_max\_word:最细切分，细粒度

IK分词器如何拓展词条？停用词条？

·利用config目录下的IKAnalyzer.cfg.xml文件添加拓展词典和停用词典

·在词典中添加拓展词或停用词

Mapping是对索引库中文档的约束，常见的mapping属性包括：

Type:字段数据类型，常见的类型有：

字符串：text(可分词的文本)、keyword（精确值，例如：品牌、国 家、Ip地址）

数值：long、integer、short、byte、double、float、

布尔：boolean

日期：date

对象：object

Index：是否创建倒排索引，默认为true

Analyzer：使用哪种分词器

Properties：该字段的子字段

创建索引库=>PUT /索引库名

查询索引库=>GET /索引库名

删除索引库=>DELETE /索引库名

添加字段=>PUT /索引库名/\_mapping

新增文档=>POST /索引库名/\_doc/文档id

查询文档=>GET /索引库名/\_doc/文档id

删除文档=>DELETE /索引库名/\_doc/文档id

全量修改文档=>PUT /索引库名/\_doc/文档id 既能修改也能新增

局部修改文档=>POST /索引库名/\_update/文档id

什么是RestClient

ES官方提供了各种不同语言的客户端，用来操作ES。

**聚合的分类**

**聚合**可以实现对文档数据的统计、分析、运算。聚合常见的有三类：

·桶（Bucket）聚合：用来对文档做分组

TermAggregation：按照文档字段值分组

Date Histogram：按照日期阶梯分组，例如一周为一组或者一月为一组

·度量（Metric）聚合：用以计算一些值，比如：最大值、最小值、平均值

Avg：求平均值

Max ：求最大值

Min：求最小值

Stats：同时求max avg min sum等

·管道（pipeline）聚合：其他聚合的结果为基础做聚合

Bucket聚合：

Aggs代表聚合，与query同级，此时query的作用是限定聚合的文档范围

聚合必须的三要素：

聚合名称

聚合类型

聚合字段

聚合可配置属性

Size

Order

Field

Metrics聚合

Elasticsearch中的分词器（analyzer）的组成包括三部分

Character filters：再tokenizer之前对文本进行处理。如删除字符、替换字符

Tokenizer：将文本按照一定的规则切割成词条（term）。如keyword，就是部分此；还有ik\_smart

Tokenizer filter:将tokenizer输出的词条做进一步处理。例如大小写转换、同义词处理、拼音处理

数据同步问题分析

Elasticsearch中得酒店数据来自于mysql数据库，因此mysql数据发生改变时，elasticsearch也必须跟着改变，这个就是elasticsearch与mysql之间的数据同步

方式一：同步调用：实现简单，业务耦合高

方式二：异步通知MQ：实现复杂，依赖mq的可靠性

方式三：监听binlog完全解除服务耦合，开启binlog增加数据库负担

Elasticsearch集群

搭建es集群

集群脑裂问题

集群故障转移

集群分布式存储

集群分布式查询

ES集群结构

单机的elasticsearch做数据存储，必然面临两个问题：海量数据存储问题、单点故障问题

海量数据存储问题：将索引库从逻辑上拆分为N个分片，存储到多个节点

单点故障问题：将分片数据再不同节点备份

Master eligible ：备选主节点：主节点可以管理和记录集群状态，决定分片在那个节点，处理创建和删除索引库的请求

Data ：数据节点：存储数据、搜索、聚合、CRUD

Ingest：数据存储之前的预处理

Coordinating：路由请求到其他节点，合并其他节点处理的结果，返回给用户

微服务保护->sentinel

**雪崩问题及其解决方案：**

微服务调用链路中的某个服务故障，引起整个链路中的所有微服务都不可用，这就是雪崩

解决雪崩问题的常见方式有四种：

·超时处理（缓解雪崩问题，并不会解决）：设定超时时间，请求超过一定时间没有相应就返回错误信息，不会无休止等待

·舱壁模式（容易造成资源浪费）：限定每个业务能使用的线程数，避免耗尽整个tomcat资源，因此，也叫线程隔离

·熔断降级：由断路器统计业务执行的异常比例，如果超出阈值则会熔断该业务，拦截访问该业务的所有请求。

·流量控制（避免服务宕机）：限制业务访问的QPS,避免服务因流量的突增而故障。

**服务保护技术对比：**

|  |  |  |
| --- | --- | --- |
|  | Sentinel | Hystrix |
| 熔断策略 | 信号量隔离 | 线程池隔离（默认）\信号量隔离 |
| 熔断降级策略 | 基于慢调用比例或异常比例 | 基于失败比例 |
| 实时指标实现 | 滑动窗口 | 滑动窗口（基于Rxjava） |
| 规则配置 | 支持多种数据源 | 支持多种数据源 |
| 扩展性 | 多个拓展点 | 插件的形式 |
| 基于注解的支持 | 支持 | 支持 |
| 限流 | 基于QPS，支持基于调用关系的限流 | 有限的限流 |
| 流量整形 | 支持慢启动，匀速排队模式 | 不支持 |
| 系统自适应保护 | 支持 | 不支持 |
| 控制台 | 开箱即用，可配置规则、查看秒级监控、机器发现等 | 只能查看，不完善 |
| 常见框架的适配 | Servlet、spring Cloud、Dubbo、gRPC等 | Servlet、Spring Cloud、Netflix |

Sentinel介绍和安装：

Sentinel是阿里巴巴开源的一款微服务流量控制组件。

Sentinel具有以下特征：

·丰富的应用场景：sentinel承接阿里巴巴近十年的双十一促销流量的核心场景，例如秒杀（即突发流量控制在系统容量可以承受的范围）、消息消峰填谷、集群流量控制、实时熔断下游不可用应用等

·完备的实时监控：sentinel同时提供实时的监控功能，可以在控制台看到接入应用的单台机器秒级数据，甚至500台以下规模的集群的汇总运行情况。

·广泛的开源生态：sentinel提供开箱即用的与其他开源框架/库整合模块。例如与spring Cloud、Dubbo、gRPC的整合。只需要引入相应的依赖，并进行简单的配置基克快速接入sentinel

·完善的SPI扩展点：sentinel提供简单易用、完善的SPI扩展接口。你可以通过实现扩展接口来快速定制逻辑。例如定制规则管理、适配动态数据等。

Sentinel限流规则：

簇点链路：就是项目内的调用链路，链路中被监控的每个接口就是一个资源。默认情况下sentinel会监控SpringMVC的每一个端点，因此springMVC的每一个端点就是调用链路中的一个资源

在添加限流规则时，点击高级选项，可以选择三种流控模式：

·直接：统计当前资源的请求，出发阈值时对当前资源直接限流，也是默认的模式

·关联：统计与当前资源相关的另一个资源，触发阈值时，对当前资源限流

满足下面条件可以使用关联模式：

两个有竞争关系的资源

一个优先级较高，一个优先级较低

·链路：统计从指定链路访问到本资源的请求，触发阈值时，对指定链路限流

**流控效果：**

·快速失败：QPS超过阈值时，拒绝新的请求

·Warm up：也叫预热模式，是应对服务冷启动的一种方案。请求阈值初始值是

threshold÷coldFactor,持续指定时长后，逐渐提高到threshold。而clodFactor的默认值是3

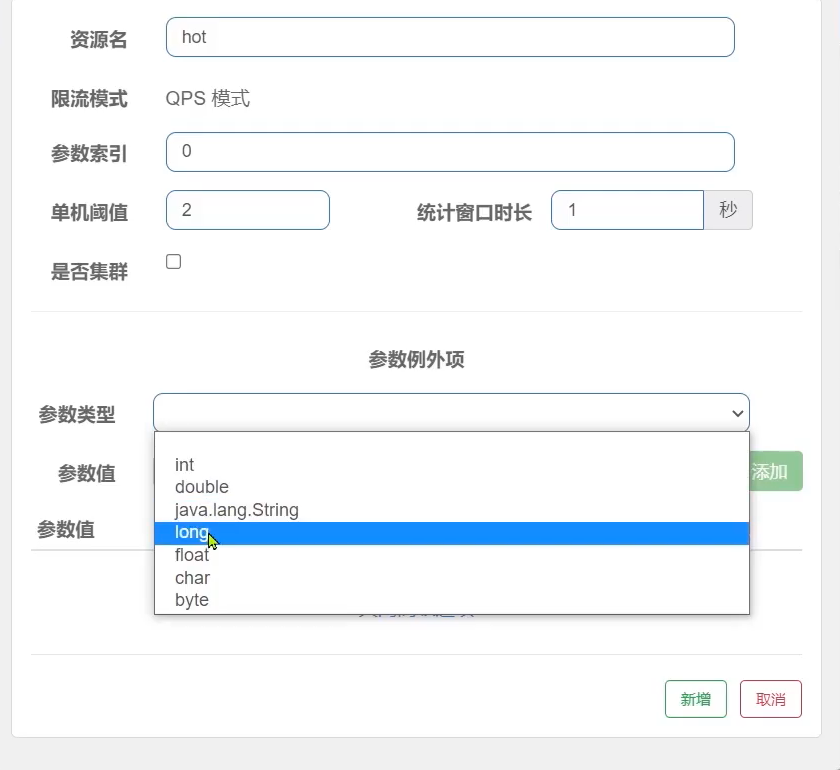
·排队等待：当请求超过QPS阈值时，快速失败和warm up会拒绝新的请求并抛出异常。而排队等待则是让所有请求进入一个队列中，然后按照阈值允许的时间间隔依次执行。后来的请求必须等待前面执行完毕，如果请求预期的等待时长超过最大时长，则会被拒绝

热点参数限流：

之前的限流时统计访问某个资源的所有请求，判断是否超过QPS阈值，而热点参数限流是分别统计参数值相同的请求，判断是否超过QPS阈值



**！！！！热点参数限流对默认的SpringMVC资源是无效的只有被@SentinelResource注解声明的资源才可以配置热点参数限流，配置热点参数限流时，参数类型只能是基本数据类型和String**



**隔离和降级**

虽然限流可以避免高并发而引起的服务故障，但服务还会因为其他原因而故障。而要将这些故障控制在一定范围，避免雪崩，就要靠线程隔离（舱壁模式）和熔断降级手段了

无论时线程隔离还是熔断降级，都是对客户端的保护

·FeignClient整合Sentinel

SpringCloud中，微服务调用都是通过Feign来实现的，因此做客户端保护必须整个Feign和Sentinel

1. 修改yml文件，开启Feign的Sentinel功能
2. 给FeignClient编写失败后的降级逻辑
   1. FallbackClass，无法对远程调用的异常做处理
   2. FallbackFactory，可以对远程调用的异常做处理

步骤一：在feign-api项目中定义类，实现FallbackFactory<要调用的客户端>，并编写降级逻辑

步骤二：在feign-api项目中的DefaultFeignConfiuration类中将步骤一的类注册成bean

步骤三：在feign-api项目中的客户端，用注解指定fallbackFactory

·线程隔离（舱壁模式）

1. 线程池隔离：

支持主动超时和异步调用，线程的额外开销较大，低扇出

1. 信号量隔离

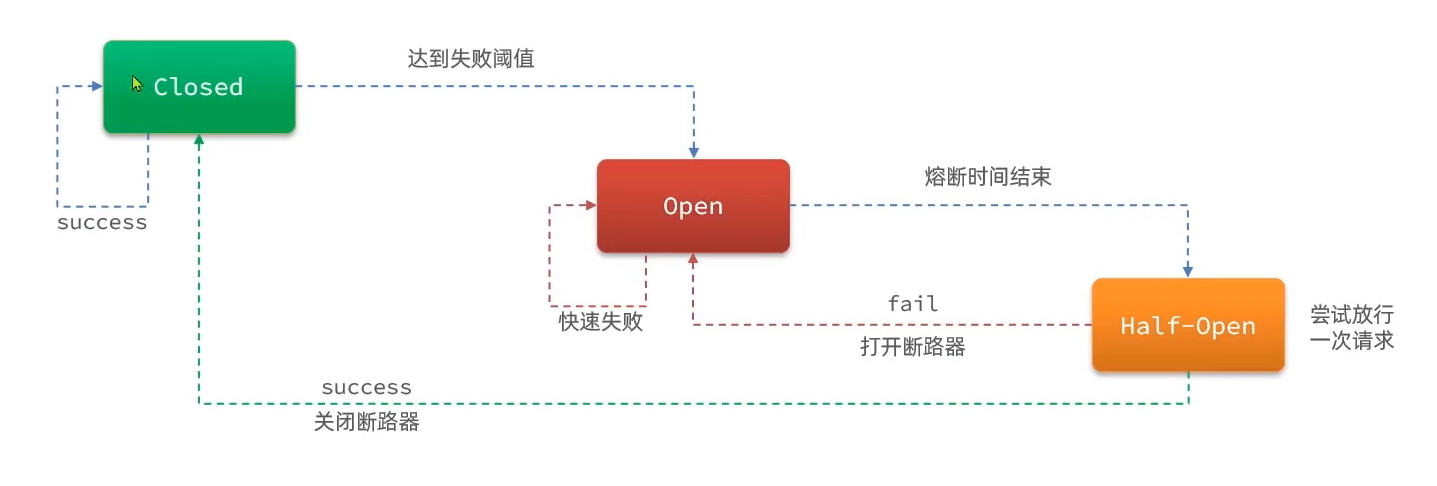
轻量级，无额外开销，不支持主动超时，不支持异步调用

·熔断降级

熔断降级是解决雪崩问题的重要手段。其思路就是由断路器统计服务调用的异常比例、慢请求比例，如果超出阈值则会熔断该服务。即拦截访问该服务的一切请求；而当服务恢复时，断路器会放行访问该服务的请求

状态机的三种状态：Closed、Open、Half-Open

当状态机是Closed时，则会放行一切访问该服务的请求，当异常比例达到失败阈值时，则会将状态变为Open，拦截一切访问该服务的请求，当熔断时间结束后，状态会变成Half-Open，状态机会尝试放行一次请求，根据该请求的状态来判断后面的状态。



断路器熔断策略有三种：慢调用、异常比例、异常数

·慢调用：业务的响应时长（RT）大于指定时长的请求认定为慢调用请求。在指定时间内，如果请求数量超过设定的最小数量，慢调用比例大于设定的阈值，则触发熔断；

·异常比例或者异常数：统计指定时间内的调用，如果调用次数超过指定请求数，并且出现异常的请求数超过设定的比例阈值，则触发熔断

**授权规则：**

授权规则可以对调用方的来源做控制，有白名单和黑名单两种方式。

·白名单：来源在白名单内的调用者允许访问

·黑名单：来源在黑名单内的调用者不允许访问

**规则管理模式**

原始模式：sentinel的默认模式，将规则保存在内存中，重启服务会丢失

Pull模式：可以实现规则持久化。保存在本地的文件或数据库中，定时去读取

Push模式：可以实现规则持久化， 保存在nacos，监听变更实时更新

微服务整合sentinel

**分布式事务：**

**理论基础**

·CAP理论

Consistency（一致性） ：用户访问分布式系统中的任意节点，得到的数据必须一致

Availability（可用性） ：用户访问集群中的任意健康节点，必须得到相应，而不是超时或者拒绝

Partition tolerance（分区容错性）：

分区：因为网络故障或者其他原因导致分布式系统中的部分节点与其他节点失去连接，形成独立分区

容错：在集群出现分区时，整个系统也要持续对外提供服务

·BASE理论

Basically Available（基本可用）：分布式系统在出现故障时，允许损失部分可用，即保证核心可用。

Soft State（软状态）：在一定时间内，允许出现中间状态，比如临时的不一致状态

Eventually Consistent（最终一致性）：虽然无法保证强一致性，但是在软状态结束后，最终达到数据一致

而分布式事务最大的问题就是各个子事务的一致性问题，因此可以借鉴SAP定理和BASE理论

·AP模式：各子事务分别执行和提交，允许出现结果不一致，然后采取弥补措施恢复数据，实现最终一致

·CP模式：各子事务执行后互相等待，同时提交，同时回滚，达成强一致，但是在事务等待中，处于弱可用状态

解决分布式事务，各个子系统之间必须等感知到彼此的事务状态，才能保证状态一致，因此需要一个事务协调者来协调每一个事务的参与者（子系统事务）

这里的子系统事务，称为分支事务，有关联的各个分支事务在一起称为全局事务。

Seata

·seata的架构

Seata事务管理中有三个重要的角色：

TC（transaction coordinator）-事务协调者：维护全局和分支事务的状态，协调全局事务提交或回滚

TM（transaction manager）-事务管理器：定义全局事务的范围、开始全局业务、提交或回滚全局事务

RM（resource manager）-资源管理器：管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚

·部署TC服务

·微服务集成seata

**Seata的XA模式（强一致，好实现），占用数据库锁，资源浪费，性能低。依赖数据库底层实现**

RM第一阶段的工作：

1. 注册分支事务到TC
2. 执行分支业务sql但是不提交
3. 报告执行状态到TC

TC第二阶段的工作

如果都成功就通知所有的RM提交

如果失败就通知所有的RM回滚

RM第二阶段的工作

接受TC指令，全部提交或者是回滚事务

**Seata的AT模式**

RM第一阶段工作：

注册分支事务

记录undo-lo（数据快照）

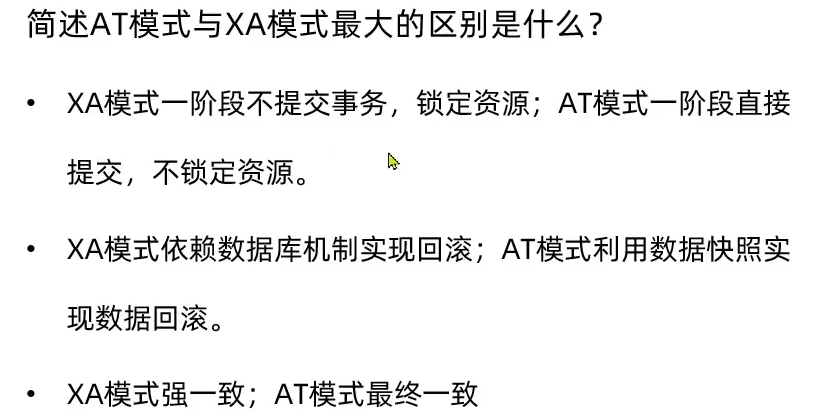
执行业务sql并提交

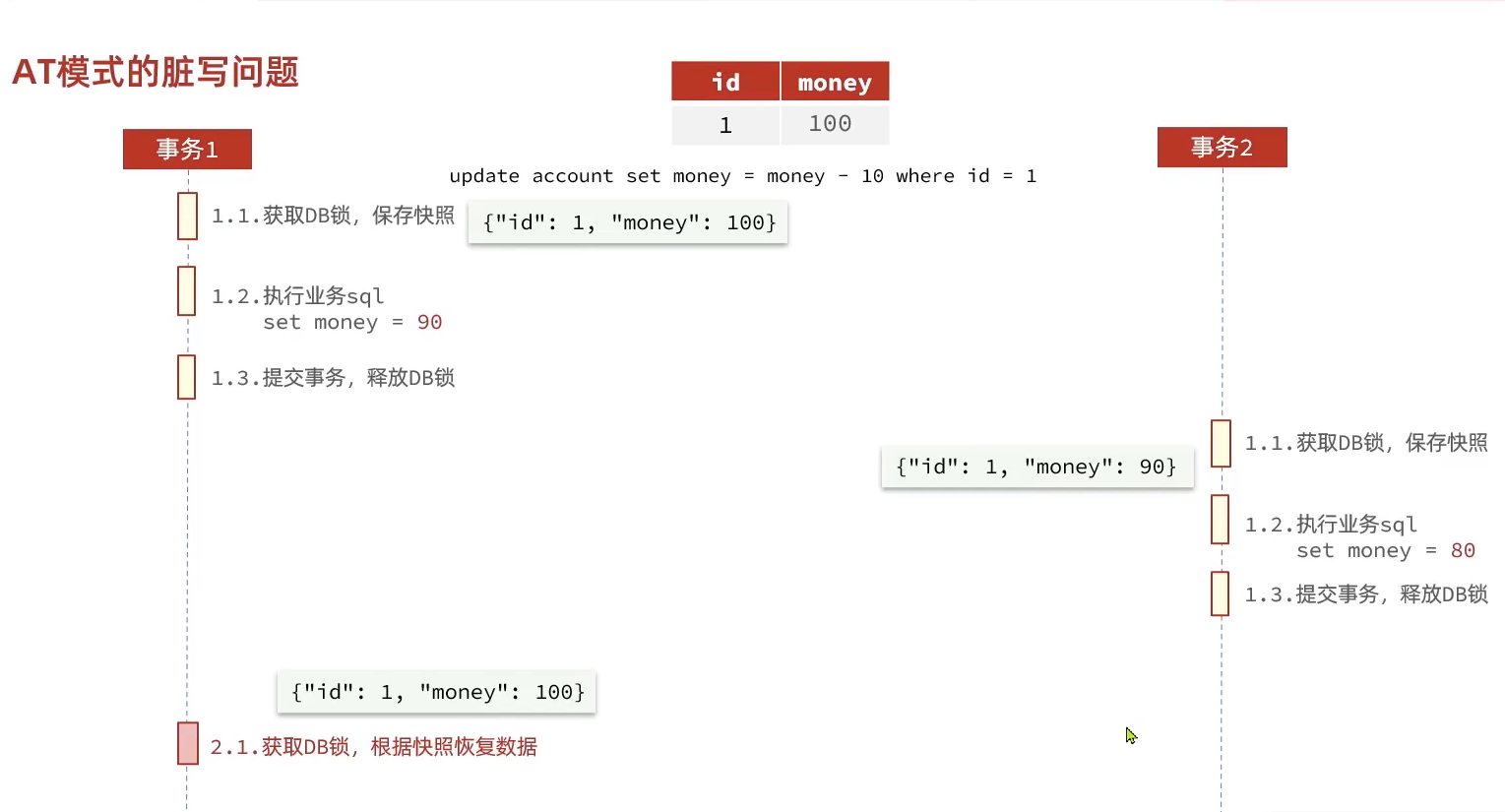
报告事务状态

RM第二阶段的工作：

所有分支事务提交成功：删除undo-log

某个分支业务挂掉：根据undo-lg恢复数据到更新前，然后在删除undo-log





如何解决AT模式的脏写问题

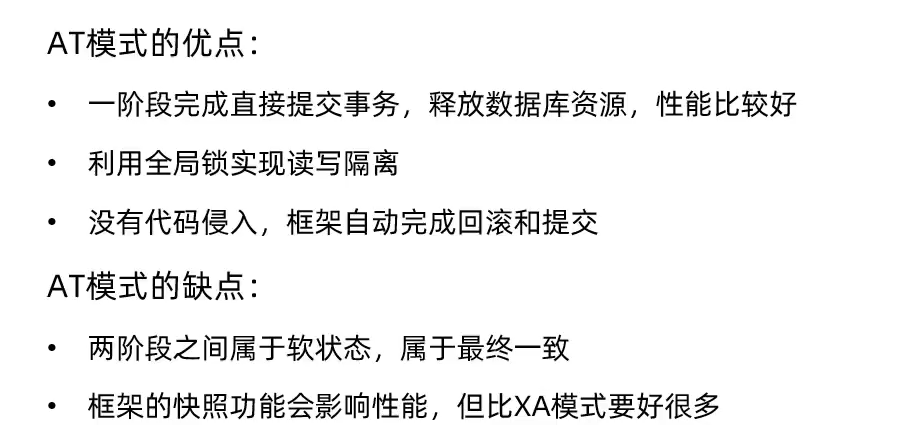
全局锁：由TC记录当前正在执行操作某行数据的事务，该事务持有全局锁，具备执行权。



事务一和事务二导致死锁

Seata内部默认获取全局锁30次，每次间隔10毫秒，解决死锁问题

所以事务二先释放DB锁，事务一获取DB锁，根据快照恢复数据



**Seata的TCC模式原理**

TCC模式和AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复，需要实现三个方法。

·Try：资源的检测和预留

·confirm：完成资源操作业务；要求Try成功confirm一定要能成功

·cancel：预留资源释放，可以理解为try的反向操作

**TCC的优点是什么**

一阶段完成直接提交事务，释放数据库锁，性能好

相比AT模型，无需生产快照，无需使用快照，性能最强

不依赖数据库事务，而是依赖补偿操作，可以用于非事务性数据库

**TCC的缺点**

代码侵入，需要人为编写TCC接口

软状态，事务最终一直

需要考虑confirm和cancel失败的情况，做好幂等处理

TCC的空回滚和业务悬挂

当某分支事务的try阶段阻塞时，可能导致全局事务超时而触发二阶段的cancel操作。在未执行try操作时先执行了cancel操作，这时cancel不能做回滚，就是空回滚

对于已经空回滚的业务，如果以后继续执行try，就永远不可能confirm或cancel，这就是业务悬挂。应当组织执行后空回滚后的try操作，避免悬挂

Saga模式

Saga模式是seata提供的长事务解决方案

一阶段：直接提交本地事务

二阶段：成功则什么都不做，失败则通过编写补偿业务来回滚

Saga模式优点：

事务参与者可以基于事件驱动实现异步调用，吞吐高

一阶段直接提交事务。无锁，性能好

不用编写TCC中的三个阶段，实现简单

缺点：

软状态持续时间不确定，时效性差

没有锁，没有事务隔离，会有脏写



**分布式缓存**

Redis集群

数据丢失问题：Redis是内存存储，服务重启可能会导致数据丢失问题

并发能力问题：单节点Redis并发能力虽然不错，但也无法满足超级高并发场景

故障恢复：如果Redis宕机，则服务不可用，需要一种自动的故障恢复手段

存储能力问题：Redis基于内存，单节点能存储的数据量难以满足海量数据需求

数据丢失问题->实现Redis数据持久化

并发能力问题->搭建主从集群，实现读写分离

故障恢复问题->利用Redis哨兵，实现健康检测和自动恢复

存储能力问题->搭建分片集群，利用插槽机制实现动态扩容

RDB全程Redis Database Backup file（Redis数据备份文件），也被叫做Redis数据快照，简单来说就是把内存中的数据记录到磁盘中。当Redis实例故障重启后，从磁盘读取快照文件，恢复数据。

快照文件称为RDB文件，默认是保存在当前运行目录

Save命令：由redis主进程执行RDB，会阻塞所有命令

Bgsave：开启子进程执行RDB，避免主进程受到影响。

Redis内部有触发RDB的机制，可以在redis.conf文件中找到。

Bgsave开始时会fork主进程得到子进程，子进程共享主进程的内存数据。完成fork后读取内存数据并写入RDB文件。

Fork采用的是copy-on-write技术：

当主进程执行读操作时，访问共享内存，

当主进程执行写操作时，则会拷贝一份数据，执行写操作。

AOF持久化：

AOF全称为Append Only File(追加文件)。Redis处理的每一个写命令都会记录在AOF文件，可以看作是命令日志文件。



数据同步



**Master如何判断slave是不是第一次来同步数据？**

Replication id：简称replid，是数据集的标记，id一致则表示是一个数据集，每一个master都有唯一的replid，slave则会集成master节点的replid

Offset：偏移量，随着记录在repl\_baklog中的数据增多而逐渐增大，slave完成同步时也会记录当前同步的offset。如果slave的offset小于master的offset，说明slave数据落后于master，需要更新

因为slave做数据同步，必须向master声明自己的replication id和offset，master才可以判断到底需要同步哪些数据

简述全量同步的流程？

Slave节点请求增量同步

Master节点判断replid，发现不一致，拒绝增量同步

Master将完整内存数据生成RDB，发送RDB到slave

Slave清空本地数据，加载master的RDB

Master将RDB期间的命令记录在repl\_baklog，并持续将log中的命令发送给slave

Slave执行接收到的命令，保持与master之间的同步

**增量同步**

当offset被覆盖执行全量同步

通过offset执行增量同步

Redis主从集群优化：

在master中配置repl-diskless-sync yes启用无磁盘复制，避免全量同步时磁盘IO不卸写在磁盘，写在网络中。

Redis单节点上的内存占用不要太大，减少RDB导致的过多磁盘IO

适当提高repl\_baklog的打小，发现slave宕机时尽快实现故障恢复，尽可能避免全量同步

Slave节点宕机恢复后可以找master节点同步数据，那么master节点宕机怎么办？

Redis哨兵

哨兵的作用和工作原理

Redis提供了哨兵机制来实现主从集群的自动故障恢复。

**监控：**sentinel会不断检查master和slave是否按预期工作

**自动故障恢复：**如果master故障，sentinel会将一个slave提升为master。当故障实例恢复后也以新的master为主

搭建哨兵集群

**通知**：sentinel充当Redis客户端的服务发现来源，当集群发生故障转移时，会将最新信息推送给Redis客户端

Sentinel基于心跳机制监测服务状态，每隔一秒向集群的每个实例发送ping命令

主管下线：如果某sentinel节点发现某实例未在规定时间响应，则认为该实例主管下线。

客观下线：若超过指定数量的sentinel都认为该实例主管下线免责该实例客观下线，quorum值最好超过sentinel实例数量的一半

**选举新的master：**

一旦发现master故障，sentinel需要在salve中选择一个作为新的master，选择依据：

首先会判断slave节点与master节点断开时间长短，如果超过指定值，则会排除该slave节点

然后判断slave节点的slave-priority值，越小优先级越高，如果是0则用不参与选举

如果slave-priority一样，则判断slave节点的offset值，越大说梦数据越新，优先级越高

最后判断slave节点的运行id大小，越小优先级越高

**实现故障转移：**

Sentinel给备选的master节点发送 slaveof no one命令，让该节点称为master

Sentinel给所有slave发送slaveof 新的主ip 端口命令，让这些slave称为新的master的从节点，开始从新的master上同步数据

最后sentinel将故障节点标记为salve，当故障节点恢复后会自动成为新的master的slave节点

※

主从和哨兵可以解决高可用高并发读的问题。但还是有两个问题没有解决

海量数据存储问题

高并发写问题

使用分片集群可以解决上述问题，分片集群特征：

集群中有多个master，每个master存储不同数据

每个master都可以有多个slave节点

Master之间通过ping检测彼此健康状态

客户端请求可以访问集群任意节点，最终都会被转发到正确节点

**Redis的分片集群**

搭建分片集群

散列插槽

Redis如何判断某个key应该在那个实例

·将16384个插槽分配到不同的实例

·根据key的有效部分计算哈希值，对16834取余

·余数作为插槽，寻找插槽所在实例即可

如何将同一类数据固定的保存在同一Redis实例

Key有一个共通的{}

集群伸缩

故障转移

当集群中有一个master宕机：

1. 首先是该实例与其他实例失去连接
2. 然后是意思宕机
3. 最后是确认下线，自动提升一个slave为一个新的master

利用cluster failover命令可以手动让集群中的某个master宕机，切换到执行cluster failover命令的这个slave节点，实现无感知的数据迁移。

手动的failover支持三种不同模式：

·缺省：默认的流程

·force：省略了对offset的一致性校验

·takeover：忽略数据一致性，忽略master状态和其他master的意见

**Redis多级缓存：**

传统的缓存策略一般是请求到达tomcat后，先查询redis，如果未命中则查询数据库，存在下面的问题：

·请求要经过tomcat处理，tomcat的性能称为整个系统的瓶颈

·Redis缓存失效时会对数据库产生冲击

多级缓存方案：多级缓存就是充分利用处理的每个环节，分别添加缓存，减轻tomcat压力，提升服务性能

JVM进程缓存

Lua语法

多级缓存

缓存同步策略

JVM进程缓存

Caffeine ->JVM进程缓存

缓存在日常开发中起到至关重要的作用，由于是存储在内存中，数据的读取速度是非常快的，能大量减少对数据库的访问，减少数据库的压力。我们把缓存分为两种：

·分布式缓存，例如redis：

优点：存储容量大，可靠性好，可以在集群中共享

缺点：访问缓存有网络开销

场景：缓存数据量较大，可靠性要求较高，需要在集群间共享

·进程本地缓存，如HashMap、GuavaCache

优点：读取本地内存、没有网络开销，速度更快

缺点：存储容量有限，可靠性低。无法共享

场景：性能要求高，缓存数据量小

Caffeine是基于JAVA8开发的，提供了近乎最佳命中率的高性能本地缓存库。目前Spring内部的缓存使用的就是Caffeine

Lua语法

|  |  |
| --- | --- |
| 数据类型 | 描述 |
| nil | 这个最简单，只有值nil属于该类，表示一个无效值，在条件表达式中相当于false |
| boolean | 包含两个值false、true |
| number | 表示双精度的浮点类型小数 |
| string | 字符串，单引或者双引都行 |
| function | 由C或者Lua编写的函数 |
| table | Lua中的表其实是一个关联数组，数组的索引可以是数字，字符串，或者表类型，在Lua中，table的创建是通过构造表达式来创建的，最简单的构造表达式是{}，用来创建一个空表 |

可以通过type函数测试给定变量或者值得类型

OpenResty是一个基于Nginx的高性能web平台，用于方便的搭建能够处理超高并发、扩展性极高的动态web应用、web服务和动态网关。

·具备Nginx的完整功能

·基于Lua语言进行扩展，继承了大量的Lua库、第三方模块

·允许使用Lua自定义业务逻辑、自定义库

OpenResty提供了各种API用来获取不同类型的请求参数：

|  |  |  |
| --- | --- | --- |
| 参数格式 | 参数示例 | 参数解析代码示例 |
| 路径占位符 | /item/1001 | 1.正则表达式匹配：  Location ~ /item/(\d+){  Content\_by\_lua\_file lua/item.lua  }  2.匹配到的参数都会存到ngx.var[]数组中，可以通过角标来获取参数 |
| 请求头 | Id:1001 | 获取请求头，返回值是table类型  ngx.req.get\_headers() |
| Get请求参数 | ?id=1001 | 获取get请求参数，返回值是table类型  Ngx.req.get\_url\_argss() |
| Post表单参数 | Id=1001 | 读取请求体  Ngx.req.read\_body()  获取post表单参数，返回值是table类型  Ngx.req.get\_post\_args() |
| JSON参数 | {“id” = 1001} | 获取请求体  Ngx.req.read\_body()  Nx.req.get\_body\_data() |

缓存同步问题

数据同步策略

设置有效期：给缓存设置有效期，到期后自动删除。再次查询时更新

优势：简单、方便

缺点：时效性差，缓存过期之前可能不一致

场景：更新频率较低，时效性要求低的业务

同步双写：在修改数据库的同时，直接修改缓存

优点：时效性长，缓存和数据库强一致

缺点：有代码侵入，耦合度高

场景：对一致性，时效性要求较高的缓存数据

异步通知：修改数据库时发送时间通知，相关服务监听到通知后修改缓存

优点：低耦合，可以同时通知多个缓存

缺点：时效性一般，可能存在中间不一致状态

场景：时效性要求一般，由多个服务需要同步

安装canal

监听canal

基于canal的异步通知：

监听mysql的binlog

·mysql master将数据变更写入二进制日志（binary log），其中记录的数据叫做binary log events

·Mysql slave将master的binary log events拷贝到他的中继日志（relay log）

·Mysql slave重放relay log中的事件，将数据变更反映到自己的数据

Canal就是把自己伪装成Mysql的一个slave节点，从而监听master的binary log 变换，再把得到的变化信息通知给canal的客户端，比如mq，es

MQ的常见问题

消息可靠性问题：如何保证发送的消息至少被消费一次

延迟消息问题：如何实现消息的延迟投递

消息堆积问题：如何解决百万消息堆积，无法处理消费的问题

高可用：如何避免单点的MQ故障而导致的不可用问题

消息可靠性：

·发送时丢失

生产者发送的消息未送达exchange

消息到达exchange后未到达queue

·MQ宕机，queue将消息丢失

·consumer接收到消息后未消费就宕机

生产者消息确认->解决发送失效时

RabbitMQ提供了publisher confirm机制避免消息发送到MQ的过程中丢失，消息发送到MQ后，会返回一个结果给发送者，表示消息是否发送成功。结果有两种请求

Publisher-confirm,发送者确认

消息成功投递到交换机返回ack

消息未投递到交换机返回nack

Publisher-return，发送者回执

消息投递到交换机了，但是没有路由到队列。返回ACK，及路由失败原因

消费持久化->解决MQ宕机导致的消息丢失

配置声明：

·publish-confirm-type:开启publisher-confim，支持两种类型

Simple：同步等待confirm结果，直到超时

Correlated：异步回调，定义confirmcallback，MQ返回结果时会回调这个confirmcallback

·publish-returns：开启publish-return功能，同样是基于callback机制，不过是定义Returncallback

Template.mandatory：定义消息路由失败时的策略。True，则调用ReturnCallBack；false则直接丢弃消息

将消息持久化

默认创建交换机队列和发送消息时，都是持久化的

消费者确认：

RabbitMQ支持消费者确认机制，消费者处理完消息后可以向mq发送ack回执，MQ收到ack回执后才会删除消息。而SpringAMQP支持三种确认模式

Manual：手动发送回执，代码侵入性高，调用api发送ack

Auto：由Spring监听listener代码是否异常

None：消息被消费者拿走，就从mq中删除

失败重试机制：

当消费者出现异常时，消息会不断地requeue，再发送给消费者，如果一直异常，会给mq造成巨大压力

可以利用Spring的retry机制，在消费者出现异常时利用本地重试，而不是一直requeue到MQ

在开启重试模式后，重试次数耗尽，如果消息依然失败，则需要由messageRecoverer接口来处理，他包含三种不同的实现：

RejectAndDontRequeueRecoverer：重试耗尽后，直接reject，丢弃消息。默认方式

ImmediateRequeueMessageRecoverer：重试耗尽后，返回nack，消息重新入列

RepublishMessageRecoverer:重试耗尽后，将失败的消息投递到指定的交换机

死信交换机->延迟消息

当一个队列的消息满足下列情况之一时，可以成为死信

消费者使用basic.reject或basic.nack声明消费失败，并且消息的requeue参数设置为false

消息是一个过期消息，超时无人消费

要投递的队列消息堆满了，最早的消息可能成为死信。

如果该队列配置了dead-letter-exchange属性，指定一个交换机，那么队列中的死信就会投递到这个交换机上，而这个交换机被称为死信交换机（Dead Letter Exchange）简称DLE

TTL，也就是Time-To-Live.如果一个队列中的消息TTL结束仍未消费，则会变为死信，ttl超时分为两种情况

消息所在的队列设置了存活时间

消息本身设置了存活时间

延迟队列：

利用TTL结合死信交换机，我们实现了消息发出后，消费者延迟收到消息的效果，这种消息模式就称为延迟队列（Delay Queue）模式

惰性队列->解决消息堆积问题

消息堆积问题：当生产者发送消息的速度超过了消费者处理消息的速度，就会导致队列中的消息堆积，知道队列存储消息达到上限，最早接收到的消息，可能成为死信，会被丢掉，这就是消息堆积问题

解决消息堆积有三种思路：

增加更多消费者，提高消费速度

在消费者内开启线程池加快消息处理速度

扩大队列内容，提高堆积上线

惰性队列的特征：

接收到消息后直接存入磁盘而非内存

消费者要消费消息是才会从磁盘中读取并加载到内存

支持数百万条的消息存储

MQ集群：

RabbitMQ是基于Erlang语言编写的，Erlang是面向并发的编程语言，天然支持集群

集群分类：

普通集群：是一种分布式集群，将队列分布在集群各个节点上，从而提高整个集群的并发能力，（当某个节点的MQ宕机，则队列丢失）

镜像集群：是一种主从集群，普通集群的基础上，增加了主从备份功能，提高集群的数据可用性

镜像集群虽然支持主从，但主从同步并不是强一致的，某些情况下可能有数据丢失因此在3.8版本后推出新功能，仲裁队列来代替镜像集群，底层采用Raft协议确保主从一致。

普通集群：

·会在集群的各个节点间共享部分数据，包括：交换机，队列元信息。不包含消息本身

·当访问集群某节点是，如果队列不在该节点，会从数据所在节点传递到当前节点并返回

·队列所在节点宕机，队列中的消息就会丢失

镜像集群：本质是主从模式，具备下面的特征：

·交换机、队列、队列中的消息会在各个mq的镜像节点之间同步备份

·创建队列的节点被称为该节点的主节点，备份到其他的节点叫做该队列的镜像节点

·一个队列的主节点可能是另一个队列的镜像节点

·所有操作都是主节点完成，然后同步给镜像节点

·主宕机后，镜像节点会代替新的主

仲裁队列：仲裁队列是3.8版本以后才有的新功能，用来代替镜像队列，具备下列特征：

·与镜像队列一样，都是主从模式，支持主从数据同步

·使用非常简单，没有复杂配置

·主从同步基于Raft协议，强一致

服务治理：

分布式架构要考虑的问题：

**服务拆分粒度如何？**

**服务集群地址如何维护？**

**服务之间如何实现远程调用？**

**服务健康状态如何感知？**

SpringCloud常见组件？



Nacos的服务注册表结构是怎样的？

要了解Nacos的服务注册表结构，需要从两方面入手：

·Nacos的分级存储模型

Namespace:环境隔离

Group:服务分组

服务->集群->实例

·Nacos的服务端源码

Nacos如何支撑阿里内部数十万服务注册压力？

Nacos如何避免并发读写冲突问题？

Nacos与Eureka的区别有哪些？

Sentinel的线程隔离和Hystix的线程隔离有什么区别？

Sentinel的限流与Gateway的限流有什么区别？