

# springcloudAlibaba

---

## 1.服务架构演进

1.1 架构演进图:

1.2 架构演进怎么说

## 2.alibaba的落地组件，搞定什么问题

2.1 微服务架构遇到的问题

2.2 全套解决方案

## 3.分布式项目变成微服务项目

3.1 项目搭建迷惑点

3.2 微服务组件依赖关系

## 4.nacos

4.1 nacos原理图

4.2 nacos客户端注册服务步骤

4.3 nacos服务端如何处理请求

4.4 nacos服务端分析总结

## 5. sentinel限流

4.1 sentinel后台架构模型

4.2 滑动时间窗算法

4.3 漏桶算法和令牌桶算法

## 6.seata分布式事务

6.1 什么是事务，什么是分布式事务

6.2 seata的基本原理

## 7.Ribbon负载均衡器

7.1 技术背景

7.2 负载均衡算法

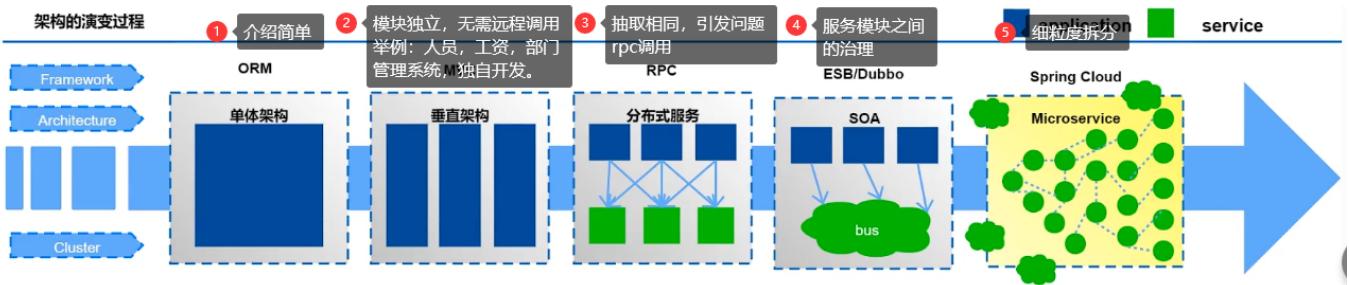
## 8.openfegin面向接口调用

8.1 openfegin解决的痛点

8.2 openfegin的原理

# 1.服务架构演进

## 1.1 架构演进图：



## 1.2 架构演进怎么说

- 架构的演进特点
- 架构的优缺点,也就是引发的问题

# 2.alibaba的落地组件，搞定什么问题

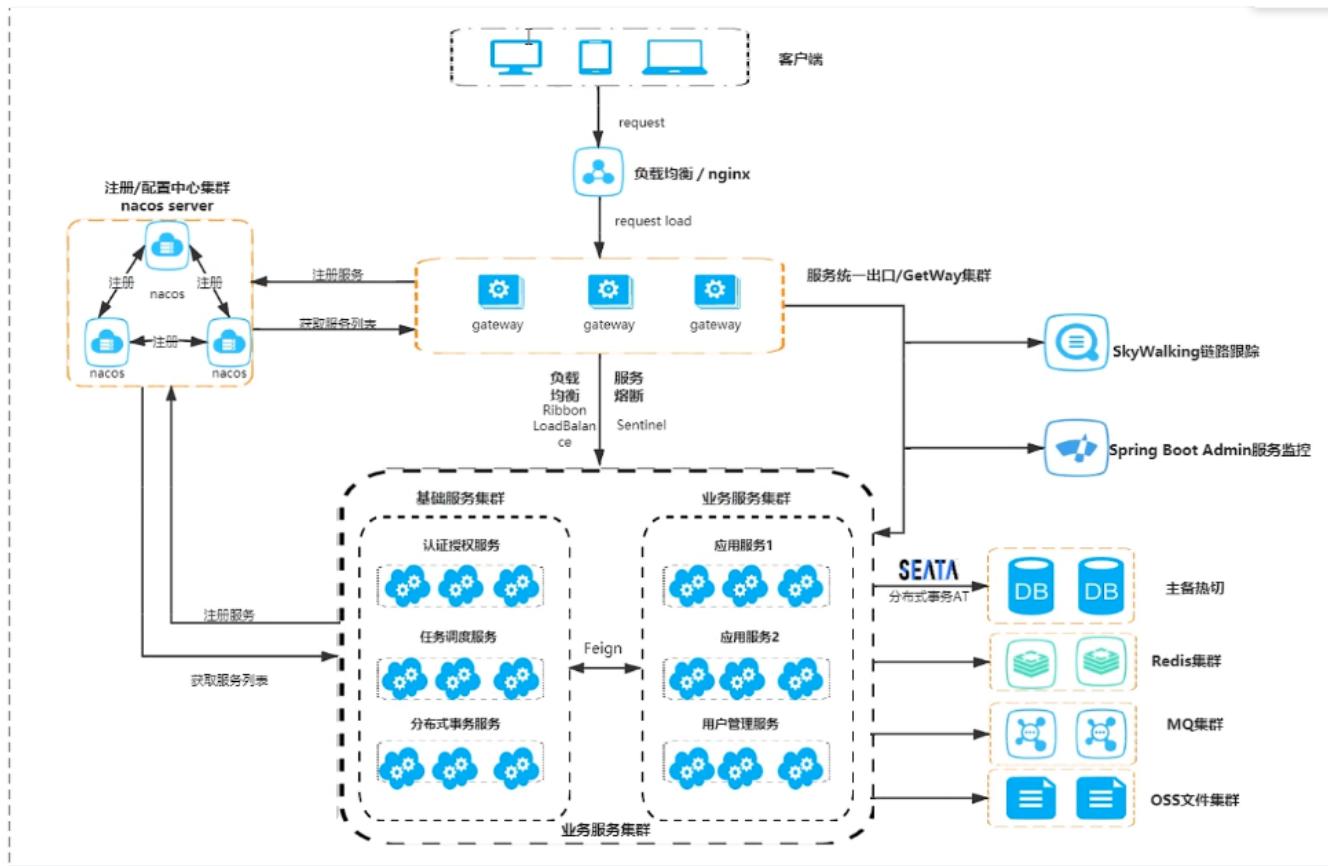
## 2.1 微服务架构遇到的问题

### 1.2.1 微服务架构的常见问题

一旦采用微服务系统架构，就势必会遇到这样几个问题：

- 这么多小服务，如何管理他们？(服务治理 注册中心[服务注册 发现 剔除]) nacos
- 这么多小服务，他们之间如何通讯？(restful rpc dubbo feign) httpclient("url",参数), springBoot restTemplate("url",参数) , feign
- 这么多小服务，客户端怎么访问他们？(网关) gateway
- 这么多小服务，一旦出现问题了，应该如何自处理？(容错) sentinel
- 这么多小服务，一旦出现问题了，应该如何排错？(链路追踪) |

## 2.2 全套解决方案



### 3. 分布式项目变成微服务项目

#### 3.1 项目搭建迷惑点

##### 1. maven

- 我们在使用springboot时候一半选择优先创建springboot初始化向导，父工程容易有boot配置，无需我们手动配置
- maven继承 ---子工程maven什么都没有自然会继承父工程，---用boot做子工程导致idea也不知道继承不.
- 总结：非boot项目 maven--maven boot 项目 boot-maven 最佳方案**

##### 2. 图 注意 boot 可以选啥

**Project Metadata**

Group:	com.tulingxueyuan.springcloud
Artifact:	springcloudalibaba
Type:	Maven POM (Generate a Maven pom.xml.)
Language:	Java
Packaging:	Jar
Java Version:	8
Version:	0.0.1-SNAPSHOT
Name:	springcloudalibaba
Description:	Spring Cloud Alibaba
Package:	com.tulingxueyuan.springcloud.springcloudalibaba

## 3.2 微服务组件依赖关系

- 项目连接

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

### 毕业版本依赖关系(推荐使用)

下表为按时间顺序发布的 Spring Cloud Alibaba 以及对应的适配 Spring Cloud 和 Spring Boot 版本关系 (由于 Spring Cloud 版本命名有调整，所以对应的 Spring Cloud Alibaba 版本号也做了对应变化)

#### 组件版本关系

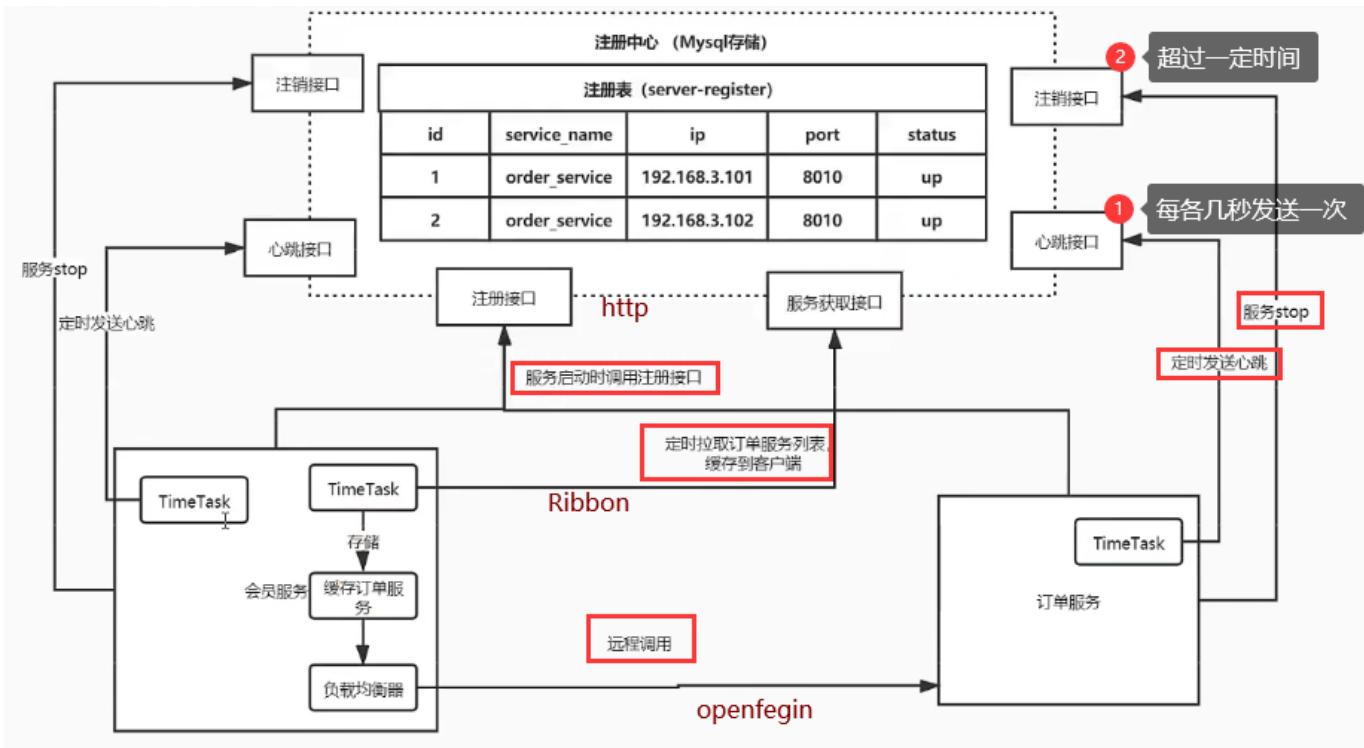
Spring Cloud Alibaba Version	Sentinel Version	Nacos Version	RocketMQ Version	Dubbo Version	Seata Version
2.2.4.RELEASE	1.8.0	1.4.1	4.4.0	2.7.8	1.3.0
2.2.3.RELEASE or 2.1.3.RELEASE or 2.0.3.RELEASE	1.8.0	1.3.3	4.4.0	2.7.8	1.3.0
2.2.1.RELEASE or 2.1.2.RELEASE or 2.0.2.RELEASE	1.7.1	1.2.1	4.4.0	2.7.6	1.2.0
2.2.0.RELEASE	1.7.1	1.1.4	4.4.0	2.7.4.1	1.0.0
2.1.1.RELEASE or 2.0.1.RELEASE or 1.5.1.RELEASE	1.7.0	1.1.4	4.4.0	2.7.3	0.9.0
2.1.0.RELEASE or 2.0.0.RELEASE or 1.5.0.RELEASE	1.6.3	1.1.1	4.4.0	2.7.3	0.7.1

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2021.0.1.0	Spring Cloud 2021.0.1	2.6.3
2.2.7.RELEASE	Spring Cloud Hoxton.SR12	2.3.12.RELEASE
2021.1	Spring Cloud 2020.0.1	2.4.2
2.2.6.RELEASE	Spring Cloud Hoxton.SR9	2.3.2.RELEASE
2.1.4.RELEASE	Spring Cloud Greenwich.SR6	2.1.13.RELEASE
2.2.1.RELEASE	Spring Cloud Hoxton.SR3	2.2.5.RELEASE
2.2.0.RELEASE	Spring Cloud Hoxton.RELEASE	2.2.X.RELEASE
2.1.2.RELEASE	Spring Cloud Greenwich	2.1.X.RELEASE
2.0.4.RELEASE(停止维护, 建议升级)	Spring Cloud Finchley	2.0.X.RELEASE
1.5.1.RELEASE(停止维护, 建议升级)	Spring Cloud Edgware	1.5.X.RELEASE

## 4.nacos

### 4.1 nacos原理图

概念：三大职责：注册中心，配置中心，web管理平台



- 注意nacos默认是有负载均衡的，它继承了ribbon openfeign是没有的，但没有说没有openfeign就不能远程调用。
- 包括我们的ribbon在拉去注册中心中的列表的时候也是通过定时任务，向服务获取接口干一个请求，得到数据

## 4.2 nacos客户端注册服务步骤

- 说明，我们的nacos要使用的时候只需要向客户端引入一个nacos的包,是基于springboot 的一个starter依赖场景启动器.
- 看spring启动原理的配置文件spring.~~~~ers:找与报名相似的类

```

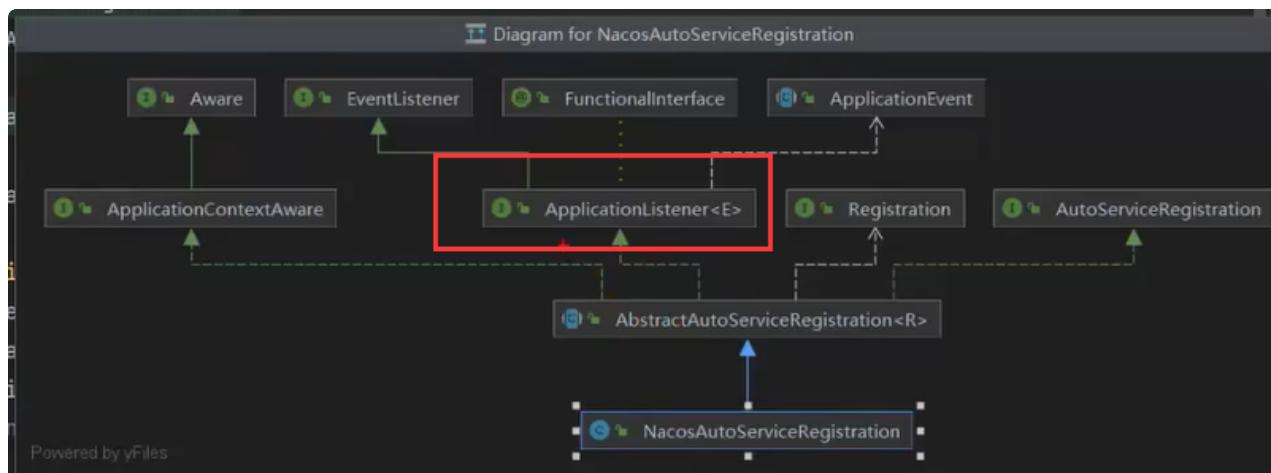
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
    com.alibaba.cloud.nacos.NacosDiscoveryAutoConfiguration,\ 
    com.alibaba.cloud.nacos.ribbon.RibbonNacosAutoConfiguration,\ 
    com.alibaba.cloud.nacos.endpoint.NacosDiscoveryEndpointAutoConfiguration,\ 
    com.alibaba.cloud.nacos.discovery.NacosDiscoveryClientAutoConfiguration,\ 
    com.alibaba.cloud.nacos.discovery.configclient.NacosConfigServerAutoConfiguration
org.springframework.cloud.bootstrap.BootstrapConfiguration=\
    com.alibaba.cloud.nacos.discovery.configclient.NacosDiscoveryClientConfigServiceBootstrapConfigurati

```

```

@Bean
@ConditionalOnBean(AutoServiceRegistrationProperties.class)
public NacosAutoServiceRegistration nacosAutoServiceRegistration(
    NacosServiceRegistry registry,
    AutoServiceRegistrationProperties autoServiceRegistrationProperties,
    NacosRegistration registration) {
    return new NacosAutoServiceRegistration(registry,
        autoServiceRegistrationProperties, registration);
}

```



```

@FunctionalInterface
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {

    /**
     * Handle an application event.
     * @param event the event to respond to
     */
    void onApplicationEvent(E event);    I
}

```

① spring容器启动的时候需要回调的事件监听器

- 上面的这个是spring提供的类，我们spring容器启动的时候就会去回调这个接口，执行它，所以我们实现了这个接口就可以做一些事情

```

protected ApplicationContext getContext() { return this.context; }

public void onApplicationEvent(WebServerInitializedEvent event) {
    this.bind(event);
}

```

```
/** @deprecated */
@Deprecated
public void bind(WebServerInitializedEvent event) {
    ApplicationContext context = event.getApplicationContext();
    if (!(context instanceof ConfigurableWebServerApplicationContext) || !"management".equals(((ConfigurableWebServerApplicationContext) context).getBeanName())) {
        this.port.compareAndSet(expect: 0, event.getWebServer().getPort());
        this.start();
    }
}
```

```
} else {
    if (!this.running.get()) {
        this.context.publishEvent(new InstancePreRegisteredEvent( source: this));
        this.register();
        if (this.shouldRegisterManagement()) {
            this.registerManagement();
        }

        this.context.publishEvent(new InstanceRegisteredEvent( source: this));
        this.running.compareAndSet( expect: false, update: true);
    }
}
```

```
public void register(Registration registration) {
    if (StringUtils.isEmpty(registration.getServiceId())) {
        log.warn("No service to register for nacos client...");
    } else {
        NamingService namingService = this.namingService();
        String serviceId = registration.getServiceId();
        String group = this.nacosDiscoveryProperties.getGroup();
        Instance instance = this.getNacosInstanceFromRegistration(registration);

        try {
            namingService.registerInstance(serviceId, group, instance);
            log.info("nacos registry, {}:{}:{} register finished", new Object[]{group, serviceId, instance});
        } catch (Exception var7) {
            log.error("nacos registry, {} register failed...{}", new Object[]{serviceId, registration});
            ReflectionUtils.rethrowRuntimeException(var7);
        }
    }
}
```

```

        namespaceId, serviceName, instance);

    final Map<String, String> params = new HashMap<?>( initialCapacity: 9);
    params.put(CommonParams.NAMESPACE_ID, namespaceId);
    params.put(CommonParams.SERVICE_NAME, serviceName);
    params.put(CommonParams.GROUP_NAME, groupName);
    params.put(CommonParams.CLUSTER_NAME, instance.getClusterName());
    params.put("ip", instance.getIp());
    params.put("port", String.valueOf(instance.getPort()));
    params.put("weight", String.valueOf(instance.getWeight()));
    params.put("enable", String.valueOf(instance.isEnabled()));
    params.put("healthy", String.valueOf(instance.isHealthy()));
    params.put("ephemeral", String.valueOf(instance.isEphemeral()));
    params.put("metadata", JSON.toJSONString(instance.getMetadata()));

    reqAPI(UtilAndComs.NACOS_URL_INSTANCE, params, HttpMethod.POST);
}

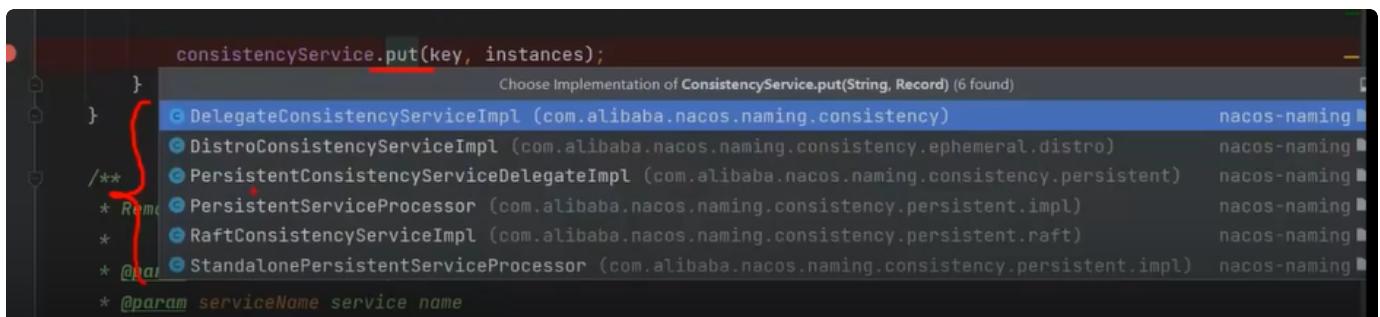
```

- 实际上就是向nacos这个web应用暴露的接口中的注册接口发送了一个注册请求，将注册信息封装到一个对象中，发送给nacos服务端。

### 4.3 nacos服务端如何处理请求

#### 1. 服务端源码查看技巧

- 静态看源码出现这种问题，**我们需要点击看前面的consistencyService是那个类生成的即可**



The screenshot shows an IDE interface with Java code. The code is as follows:

```

        consistencyService.put(key, instances);
    }

    /**
     * Remove
     */
    * @param service
     * @param name
     * @param ephemeral whether instance is ephemeral
    */
    Instances.setInstanceList(instanceList);

    consistencyService.put(key, instances);
}

```

A red arrow points from the line 'consistencyService.put(key, instances);' to the code completion dropdown. The dropdown lists six implementations of the `ConsistencyService` interface:

- DelegateConsistencyServiceImpl (com.alibaba.nacos.naming.consistency)
- DistroConsistencyServiceImpl (com.alibaba.nacos.naming.consistency.ephemeral.distro)
- PersistentConsistencyServiceDelegateImpl (com.alibaba.nacos.naming.consistency.persistent)
- PersistentServiceProcessor (com.alibaba.nacos.naming.consistency.persistent.impl)
- RaftConsistencyServiceImpl (com.alibaba.nacos.naming.consistency.persistent.raft)
- StandalonePersistentServiceProcessor (com.alibaba.nacos.naming.consistency.persistent.impl)

The first item, `DelegateConsistencyServiceImpl`, is highlighted in blue.

Below the code completion dropdown, another red arrow points to the line 'consistencyService.put(key, instances);' with the annotation '看这个属性的初始化的地方' (Look at where this attribute is initialized).

At the bottom of the screen, the code continues with:

```

91
92     private final Lock lock = new ReentrantLock();
93
94     @Resource(name = "consistencyDelegate")
95     private ConsistencyService consistencyService;
96
97     private final SwitchDomain switchDomain;
98

```

- 当某一个类实现了线程创建的接口，盲猜可能是要启动一个线程异步去执行任务，一定要看run方法

```

public class Notifier implements Runnable {
    ...
    private BlockingQueue<Pair<String, DataOperation>> tasks = new ArrayBlockingQueue<>(capacity: 1024 * 1024);
    ...
    public void addTask(String datumKey, DataOperation action) {
        ...
        if (services.containsKey(datumKey) && action == DataOperation.CHANGE) {
            return;
        }
        if (action == DataOperation.CHANGE) {
            services.put(datumKey, StringUtils.EMPTY);
        }
        tasks.offer(Pair.with(datumKey, action));
    }
}

@Override
public void run() {
    Loggers.DISTRO.info("distro notifier started");

    for (;;) {
        try {
            Pair<String, DataOperation> pair = tasks.take();
            handle(pair);
        } catch (Throwable e) {
            Loggers.DISTRO.error("[NACOS-DISTRO] Error while handling notifying task", e);
        }
    }
}

```

① 一般是开启一个线程去异步执行，所以我们优先看run方法

② 搞一个死循环去消费这个队列 实现异步，高性能

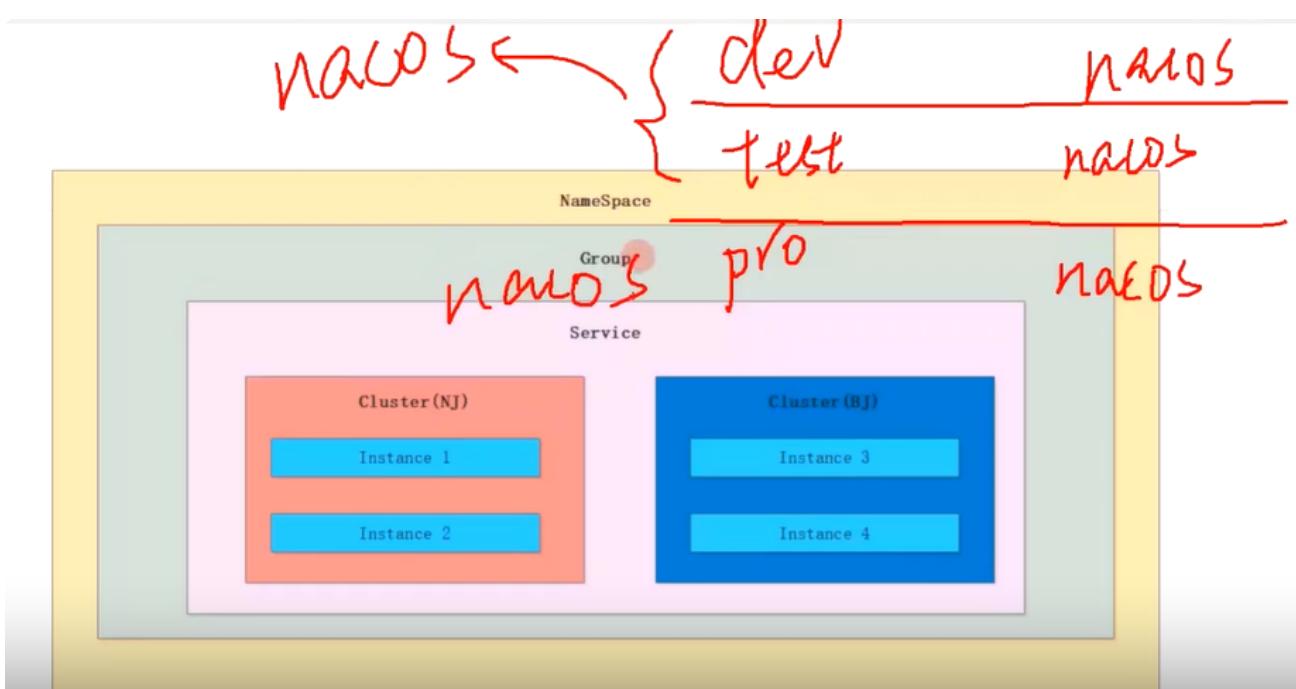
- 服务端的注册表结构:双层map的方式, 外层的namespace用于区分开发环境, 所以注册表可以支持不同环境下的配置, group分组也支持

```

@Component
public class ServiceManager implements RecordListener<Service> {

    /**
     * Map(namespace, Map(group::serviceName, Service)).
     */
    private final Map<String, Map<String, Service>> serviceMap = new ConcurrentHashMap<>();
}

```



## 4.4 nacos服务端分析总结

### 1. nacos服务工作机制

当客户端启动的时候会发送一个请求到nacos服务端，nacos接收到请求，得到客户端的服务信息，如ip,port,健康状态，等，封装成一个一个的任务会存入到一个阻塞队列中，然后启动一个线程去消费队列，可以实现异步的操作，支持了接收请求的高可用，这个消费线程是一个死循环的消费队列，会不会cpu100%，不会，因为，阻塞队列，队列没有任务就会阻塞住，任务满了就会阻塞生产的线程，这样就完成了注册；我们的ribbon在后台会开启一个定时任务，不停的从nacos注册中心的获取服务的接口拉去nacos注册信息，一个读一个写会出现并发问题？nacos通过copyonwrite的思想解决了这个问题，复制一份出来，写去操作，原来的读操作最后同步一下即可。

### 2. nacos中的心跳机制

心跳机制呢，其实是客户端启动完成之后，注册成功，会来一个定时任务，每隔一段事件就去发送请求，请求nacos服务端的接口，服务端处理请求，交给一个异步的线程去消费任务，这个线程获取到一个一个的服务健康状态，当前系统时间-上一次心跳时间>检查时间，就会将健康状态改为false，如果大于deleteTimeout ok删除服务即可，从注册列表中移除，也是请求注销接口

### 3. nacos注册列表介绍

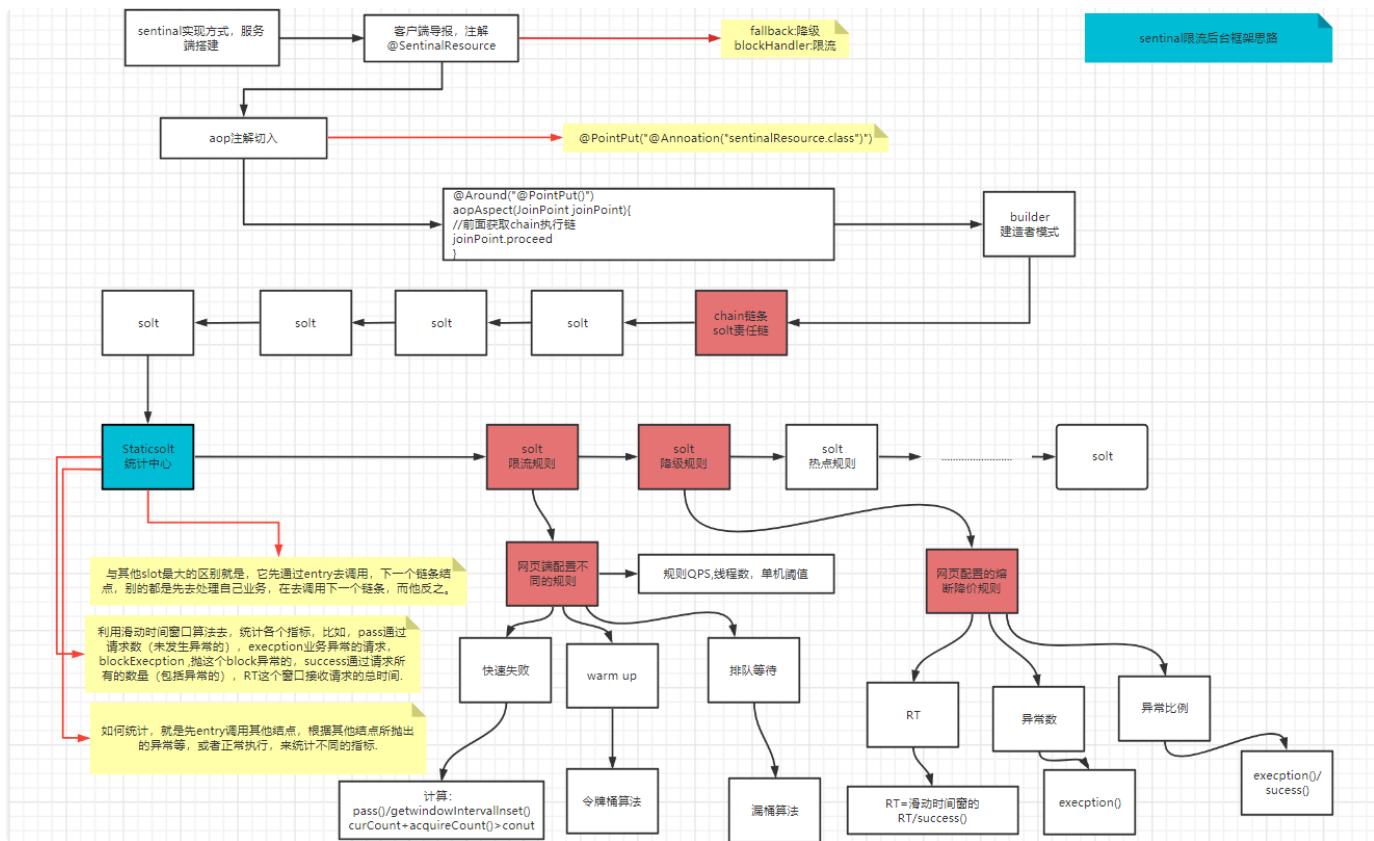
nacos中的注册列表实际上是一个双层map==>`map<namespace, map<group::servicename, service>>`为什么要搞这么复杂，在实际开发过程中，商用版的nacos一般会很贵的，没有那么多，但我们需要有dev,test,生产分支，namespace就能隔离空间分支，group还可以对业务进行分组注册，service它是一个复杂的对象里面有更多的信息。

### 4. nacos技术选型，考虑点

springcloud停更的原因，是因为它是一个免费开源的生态圈，没有盈利，最大关键，而springcloudalibaba这一套是盈利的，如果该公司并发量比较小，用开源的alibaba即可，但如果并发量比较足够，还是建议你用商用版的，因为开源的有很多bug，当你并发量上去，就会踩很多坑，商用版，会有阿里专门维护alibaba这个技术栈的来维护这个东西，当然如果我们技术够深，够厉害，遇到问题可以解决，那可以采用开源，减少开支。

## 5. sentinel限流

### 4.1 sentinel后台架构模型



## 4.2 滑动时间窗算法

### 1. sentinel的底层实现

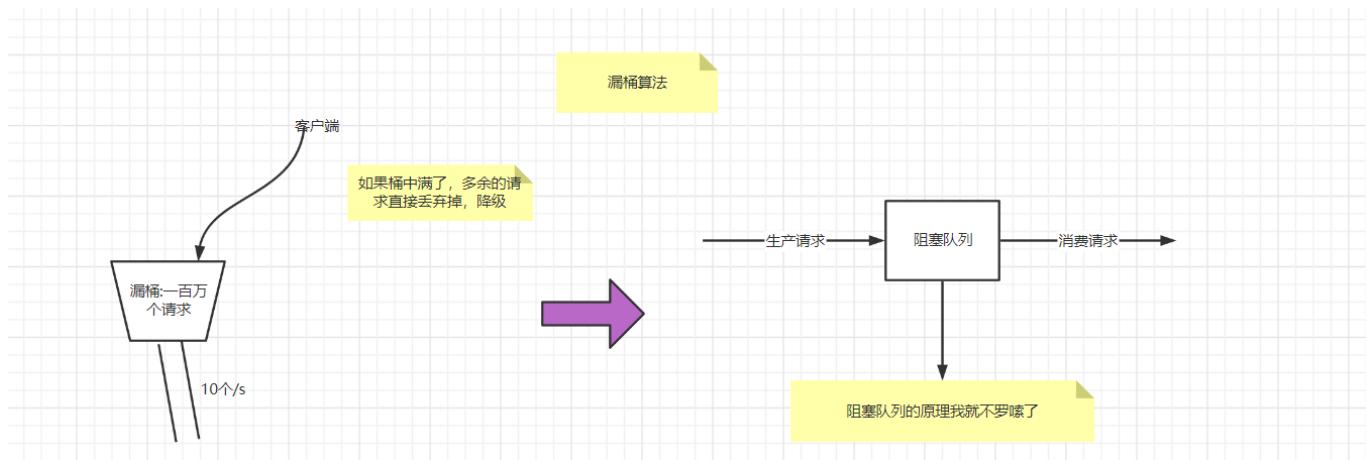
- 搞个数组，长度为2（默认值）；这个总共窗口大小为0~1000ms 那么窗口一分为2，一个窗口大小为500ms
- 当前时间/500ms=有多少个大小的500的窗口
- timeld/array.length=窗口下标 要么0窗口，要么1窗口
- 当前时间 - 当前时间%500ms = 每个窗口的起始位置
- 起始位置>窗口的最终位置====清除+更新起始窗口位置

### 2. 自己实现方式

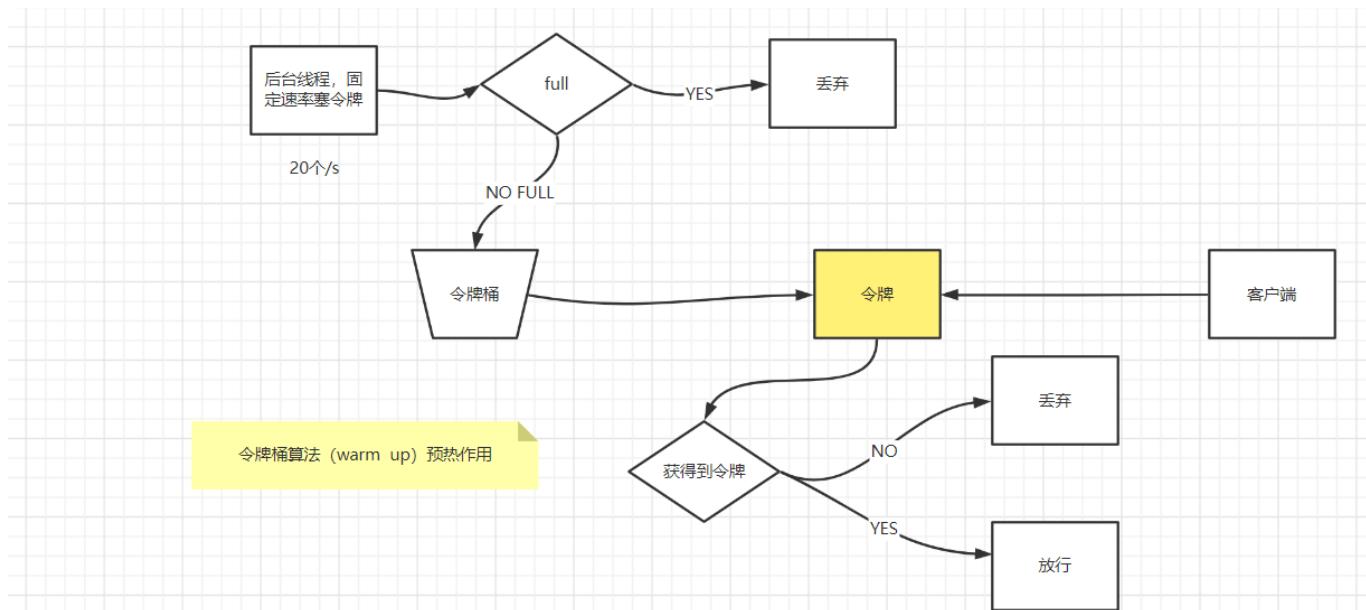
- 搞一个linkedList 队列 0~1000ms分了10个窗口
- 搞一个统计指标count统计请求数
- 0,10,10,30,40.....110
- if (peekLast()-peekFirst()>count(限流指标)) { 限流 } else { 放行 } **限流**
- if(size()>10){ 移除队头 } **保证实时性**

## 4.3 漏桶算法和令牌桶算法

### 1. 漏桶算法



### 2. 令牌桶算法



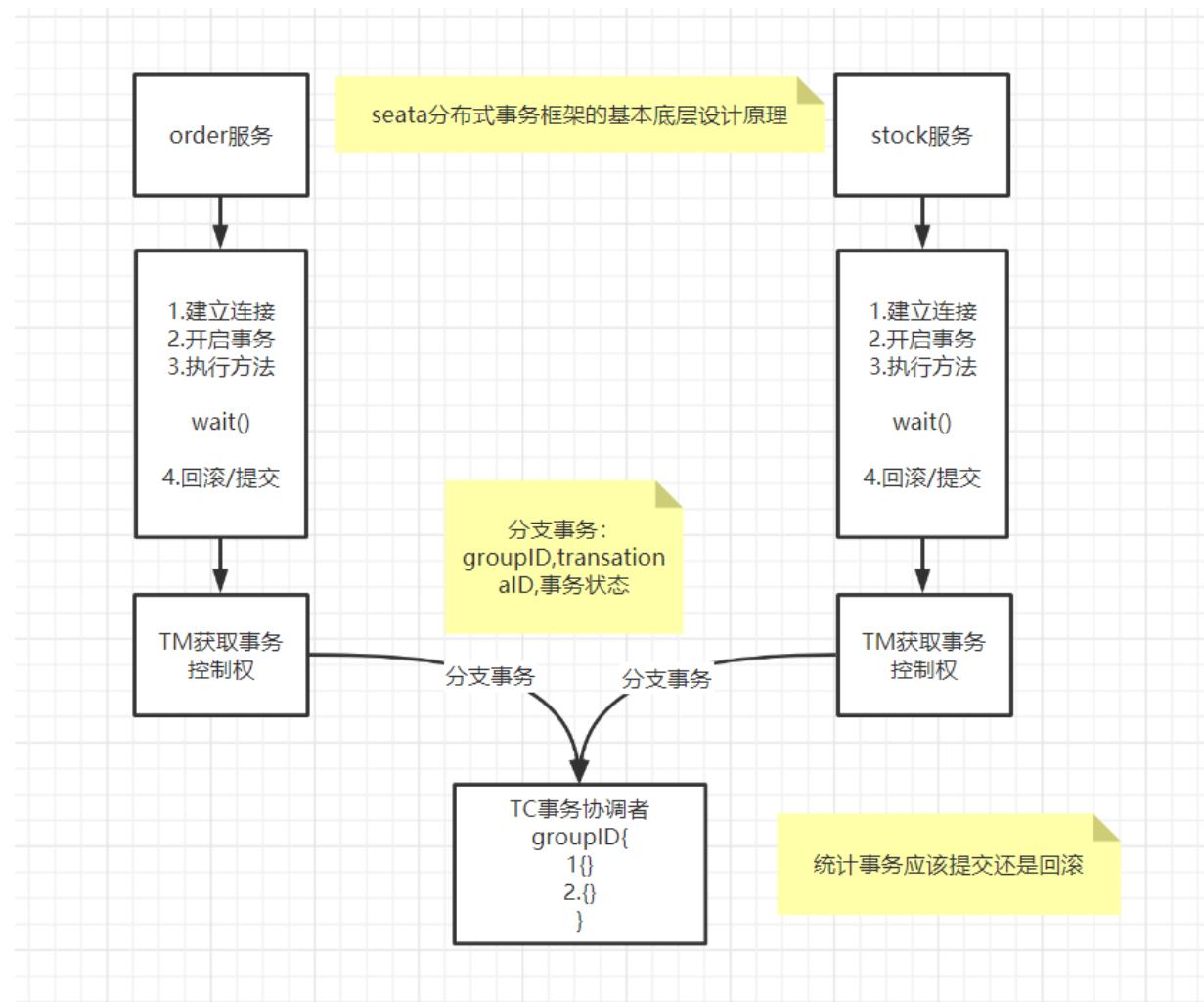
## 6.seata分布式事务

### 6.1 什么是事务，什么是分布式事务

- 事务是原子的，是面向连接的，也就是说，我们的事务用一个数据库连接，才能保证事务的原子性
- 遇到的问题，当我们有两个服务模块，order模块和stock模块，order模块需要调用stock模块的方法，加上spring提供的Transactional事务注解，不能保证两个都回滚，用的不是一个数据库连接。
- 分布式事务：上面的事务问题就是一个分布式事务问题。

### 6.2 seata的基本原理

原理架构图：



seata的原理是通过几种角色来完成的：

1. TC: 事务协调者

- 统计事务结果，响应给TM事务管理者

2. TM: 事务管理者

- **aop切入**: 我们在使用seata事务框架的时候会在控制事务的方法上加一个globalTransational全局事务控制主机：实际上我们可以通过切面的方式去切这个注解，这样我们就可以在这个方法进行功能的增强

- **创建分支事务，事务组**: 设置分支事务状态

- **发命令**: 向TC协调者注册分支事务

- **提交还是回滚**

3. RM: 代理对象，要获取事务的控制权

- 事务控制权获取: 事务是面向连接的，原生的事务连接是通过DataSource.getConnection()获取的，而spring 在实现的时候对这个getConnection有自己的实现，我们可以定义一个自己的LBConnection事务连接，**通过aop切入这个方法**，原本它返回的是connection,我们可以返回一个自己的LBconnection并且还可以复用connection，我们只需要做提交的**wait()**逻辑即可；

- **代理模式的方式**: 我们通过代理的方式找一个中介做事务的控制权，去执行这些复杂的逻辑，实际上是一个原理。

#### ▼ 事务控制权获取

Java | 复制代码

```
1 ▼ public class GUIDataSourceAspect{
2     @Around("execution(* javax.sql.DataSource.getConnection())");
3     public Connection around(ProceedJoinPoint point){
4         Connection connection=(Connection) point.proceed(); //spring本身的
        实现类
5         return new GuiConnection(connection);
6     }
7 }
```

4. XID: groupID 分布式事务组

- 我们的TC可能有多个事务分组，用于区分事务分组，并且可以精准注册那个事务组的分支事务。

## 7.Ribbon负载均衡器

## 7.1 技术背景

1. 集中式负载均衡：硬件和软件
2. 硬件：我们计算机网络中通过路由表转发，或者交换机实现负载均衡
3. 软件：
  - nginx: 服务端的负载均衡，就是客户端并不知道服务端的具体服务在那，将请求发送到代理服务，代理服务做负载均衡转发。
  - ribbon：软件负载均衡，客户端知道服务端的地址，在自己的客户端通过不同的算法，选择性的发送到某一台服务器，而ribbon呢是有一个后台线程，不停的从注册中心拉去服务列表，比如nacos，它就对nacos的获取列表接口定时发送请求，放在自己缓存中，然后通过拦截器拦截请求，根据负载均衡算法发送到某一台服务器。
  - LoadBlance

## 7.2 负载均衡算法

- 随机：略
- 轮询：略
- 权重：略
- 重试：如果一个请求访问第一台服务器访问失败，就会重试去访问令一台服务器；这也是dubbo集群戎策的一种策略。
- 一致性hash：搞一个hash环，将hash环分为16384个槽位，我们将我们的提供服务方的ip通过hash运算%16384，分别在hash环的某个位置，当然hash运算的散列性，不会平均分配在hash环上，所以我们使用了虚拟主机映射技术，让hash环上多出多个虚拟主机hash槽，然后用户请求ip通过hash运算发送到不同的主机，如果访问的虚拟主机，就去找对应的真实主机，当然，用户hash运算结束是hash的某一个位置，就会去找对应的最近的主机去处理请求。
- 最少活跃数：通过一个变量action计数，发送请求action+1，这个服务器响应回来action-1，过一段时间，action值越小，它的响应速度越高，我们就给他多发请求。
- 区域（默认策略，多台服务器在不同的区域）：服务器部署在不同的区域中，请求通过就近原则的方式。

## 8.openfegin面向接口调用

### 8.1 openfegin解决的痛点

我们在服务调用的时候一般用两种方式

- httpClient: 这种是最原生的方式，我们需要自己写请求地址信息，请求需要序列化json,回来再反序列化，维护不方便。
- jdbcTempleaf: 做了升级，发送请求的时候，会指定参数类型，从而我们不需要自己序列化，但是不方便维护的特点还是不能解决。
- **如何能向dubbo那样，向调用本地接口一样呢，这也是解决的痛点**

## 8.2 openfeign的原理

实际上就是代理模式：

让我们这些，发送请求，做负载均衡，或者，接收请求，转换数据等操作搞一个中介去帮忙解决这些麻烦事。

dubbo也是这样的原理，如果有兴趣去了解一下它们的底层设计。