|  |
| --- |
| 项目描述： |
| 项目职责：  参与需求评审，具体核心功能开发，编写开发文档，辅助功能接口设计，维护模块代码，根据需求进行二次开发及持续迭代  独自完成注册中心优化，节省了服务的上下线时间，降低了无效服务的调用率，提高了接口调用的成功率。   1. 完成注册中心eureka优化，节省了服务上下线时间，降低了无效服务调用率，提高了接口调用的成功率 2. Review代码的时候，提升了10倍效率（验证码问题），设定提升QPS的规范 |
| 面试项目核心：需求，架构图，流程图，关键业务的实现（不是核心业务的实现，而是核心业务用到的关键技术点，分布式锁，分布式事务，消息队列，缓存） |
| 扯皮技术点：缓存，FutureTask，编程式事务，跟面试官费时间，减少对面提问题的机会 |

**1,、公司项目的经历？**

逸品出行，凌动智行，在杭州，在航星园，4号楼。有个篮球场，篮球场那边有个大牌子，7月运行，12月底破产

车载系统：元心系统。

公司倒闭，两个创始人老矛盾

出差和领导一起，网约车申请牌照，到公安厅申请会客登记单，找网警总队，线下运营牌照还有线上牌照

公安找安全公司，扫你的代码，查看你的技术。申请牌照申请了半年多，之后还是找关系，找的公安厅专门做审核的，叫什么王工。

1. **为什么做这个项目，这个项目和其他出行产品有什么区别**

车是公司自己的，司机也是公司统一培训的包括基础急救，服务准则等，能有效规避各种复杂人员，造成违反犯罪事件。在安全方面做到有效把控

针对特殊群体，提供特殊服务，比如：残障人士，孕妇，带小孩的话，部分特殊车辆会配备轮椅，小孩座椅，孕妇的话找个开车稳的司机，就是有各种人士用的东西。有各种车，派对小巴士

**3、你们整个项目流程？**

启动阶段 ：Kick Off（意指开会，各方参与齐聚一堂，启动会动员大会，不认识的认识认识）

计划阶段：进度安排，资源计划，成本估计，质量保证计划，风险，实施

实施控制阶段：开发，测试，等

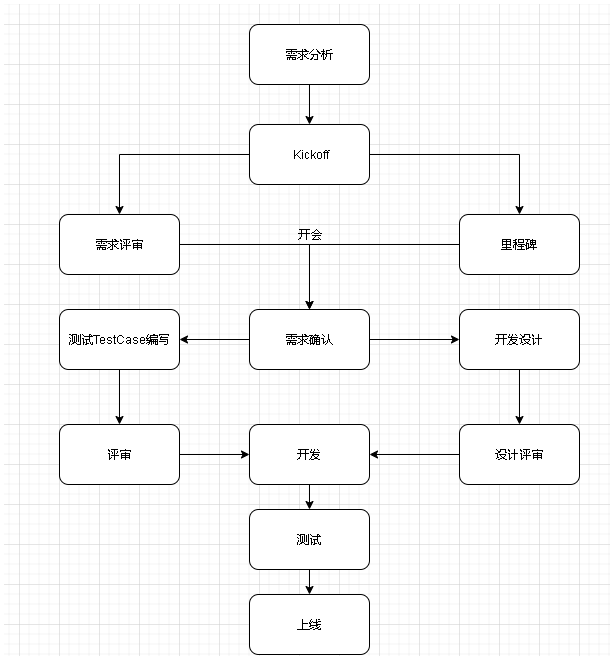
收尾：验收，公司的项目就产品验收，甲方的公司就甲方验收

你们真的按照以下的流程执行的吗？

不是，肯定跟团队沟通嘛，

若是沟通不畅怎么办？

请吃饭，交流



1. **项目做不完了怎么办？**

先加班，加人，功能排优先级，重要的先做，保证基本的功能可以用，后面再迭代

1. **你们团队有多少人，什么样的管理方式？你在项目中的职责？**

矩阵式管理方式，比如开发1组，2,组。产品1组，2组。从各个组抽两人。组成项目组，就开始做了。

**人员安排**：70人，3+1+2+8+10+8+14+4+3+3=56

项目管理：3（1高级项目经理，2助理） 技术总监：1 运维：2

能力层：10（我在能力层，开发server层） 业务层：12（开发api层）

产品：10，（乘客，司机，boss，h5） 安卓：4 ios：3 h5网页:5

测试：20（功能，自动化测试，接口测试，安全测试）

运营，市场，大客户关系：未知。

**自己的定位：**

组长：接口定义，工程结构设计，代码review，各方沟通（产品，测试），核心功能开发。

组员：具体开发实现。核心功能开发。

1. **产品功能基本描述？**

乘客端app：选择出发点和目的地，然后呼叫用车。服务完毕，乘客付款

司机端app：司机接单，然后去出发点接客，开车送往目的地。到达后，司机发起收款

两端同时在地图上显示汽车运行轨迹，这个叫司乘同现，买的高德的服务，挺贵的

不买高德的服务，司机的位置不会及时更新。买了这个服务

1. **电子围栏怎么做的**

用的第三方高德的地理围栏。Map服务下的FenceController。然后当时公司不想赔钱，车比较少，在杭州划了很多小圈。在西湖景区，阿里巴巴园区。用户与围栏位置发生变化的行为称为围栏触发行为，围栏触发行为也是用户进入围栏、退出围栏、在围栏内停留这三种行为的统称，行为的触发是通过Android 的广播进行发送的。在业务代码中有广播监听器，解析广播内容。第一布创建围栏，第二部开始定位（启动的定位机制：通过“远离围栏时的超低频定位策略”来降低电量消耗，“离近围栏会逐渐提高定位频率”，保证有足够的定位精度来完成围栏位置检测。），第三步接受围栏创建后的回调，第四布接受围栏触发行为广播

1. **购买的高德的服务有哪些？**

地理围栏（城市试点运营）高德鼓舞300万

乘客如何查看司机的行径位置：高德的猎鹰轨迹服务里面包含了轨迹管理和上传

定位纠偏：高德服务，

1. **如何拆分隔离的，微服务设计原则？**

拆分隔离的目的：隔离系统的变化点

具体原则：

高内聚（一个功能相似的业务组合成一个模块）

低耦合（各个模块各司其职）

高度自治（每个服务都可以自己开发，测试，构建，部署，运行，发布不会干扰其他服务）

以业务为中心，每个服务都有自己的业务逻辑

弹性设计（容错，隔离，降级）

自动化（持续集成，持续交付）

注意粒度把控：任何一个服务，不要因为自己的开发维护影响其他服务

1. **你们的项目是如何抗住那么高并发**

AKF拆分原则：x轴（水平复制，加机器做负载）y轴（业务拆分，单个服务业务简单化）

z轴（数据分片，不同地区数据分开）若想不同地区数据同步可以买服务：盘古公司

1. **产品架构，用到的SpringCloud技术栈有哪些**

Eureka：服务注册与发现，用于服务管理。

Feign： web调用客户端，能够简化HTTP接口的调用。

Ribbon：基于客户端的负载均衡。 Hystrix：熔断降级，防止服务雪崩。

Zuul：网关路由，提供路由转发、请求过滤、限流降级等功能。

Config：配置中心，分布式配置管理 Sleuth：服务链路追踪 Admin：健康管理

1. **你觉得微服务的优点缺点**

独立部署。不依赖其他服务，耦合性低，易于开发和维护：关注特定业务，所以业务清晰，代码复用。不需要重复写。按需伸缩：某个服务受限，可以按需增加机器等

分布式固有的复杂性：容错（某个服务宕机），网络延时，调用关系、分布式事务等。分布式事务的挑战：每个服务有自己的数据库，接口调整成本高，测试难度提升：一个接口改变，所有调用方都得测。自动化测试就变的重要了。API文档的管理也尤为重要。推荐：yapi。运维要求高：需要维护 几十 上百个服务

1. **在引用第三方技术的时候**

短信服务（腾讯，阿里短信，华信），语音服务：隐私号。（乘客和司机订单匹配后，A，B，X），录音。文件服务oss：阿里 地图：高德。100万 消息推送：极光。透传，通知。

支付：微信，支付宝。 航旅纵横：查航班。 发票：百望云。 前端：vue。android，ios。还有申请运营需要三级等保（信息产品安全等级资格认证）找的第三方中介公司做的，第三方跟网警有关系，然后容易就过了

青云，用来检测服务健康状态的，一根探针5000块钱。往app塞一个探针，往服务塞一个探针

1. **预测多少？能力层的QPS是多少？压测多少**

当时预估并发1500，压测的是2000，用的4核8G的服务器

用的压测工具是apache JMeter

1. **服务的安全如何保证？**

CIA: 保密性，完整性，可用性。

保密性：手机号，身份证号，脱敏。这也是三级等保要求的，就是 存库里也要脱敏

完整性：防篡改，就是说数据不丢失不被人篡改，即使篡改了，数据也会无效

可用性：防止资源被恶意占用。

在数据层面，要注意sql注入。${}：直接拼接。#{}：

可以用jsoup框架做接口参数过滤。

还要做数据权限控制，防止非法用户篡改数据

1. **csrf和xss的区别**

Xss：跨站脚本攻击）是向网页中注入恶意脚本在用户浏览网页时在用户浏览器中执行恶意脚本的攻击方式。问题出在：用户数据没有过滤，转义。可以用jsoup框架防止xss

Csrf：跨站请求伪造）是攻击者通过跨站请求，以合法的用户身份进行非法操作（如转账或发帖等）。CSRF的原理是利用浏览器的Cookie或服务器的Session，盗取用户身份。防御方式使用token验证，在表单中添加token

前者需要用户先登录网站，获取cookie，后者不需要。前者利用网站的漏洞请求api，后者是向网站注入js代码，篡改网站

1. **简单说一下eureka**

博文推荐：https://www.shared-code.com/type/8

Eureka是一个基于REST的服务，就是服务注册中心，服务提供者在启动时,会向EurekaServer发起一次请求,将自己注册到Eureka注册中心中去。并且通过心跳定期通过http方式向Eureka刷新自己的状态

消费者 : 向Eureka 订阅服务,Eureka会将对应服务的服务列表发送给消费者

***服务端***

Register：服务注册

当Eureka客户端向Eureka Server注册时，它提供自身的元数据，比如IP地址、端口，运行状况指示符URL，主页等。没有设置serviceID，就用application Name替代。具体实现类是DiscoveryClient类，它实现了EurekaClient接口并且是个单例，服务注册是该类下的register方法，将实例信息用过http请求来注册。同时 在DiscoveryClient初始化时创建的类InstanceInfoReplicator，在需要向服务注册时，开启定时任务InitScheduledTasks方法，向服务续约。在服务端有ApplicationResource类的addInstance方法提供注册接口，方法体中调用了PeerAwareInstanceRegistryImpl的register()方法（数据instanceInfo先保存在lease再保存在concurrentHashmap中，在这个过程中，为防止其他线程对注册表数据操作，会获取读锁，查询具体的实例后，判断租约存在，是则判断最后更新时间大小，大则更新。小则扔掉。不存在租约就创建新的。然后将租约放到resgitry中，接着一系列缓存操作及设置服务状态，设置租约上线时间。最后释放锁）

map<服务名，map<实例id，实例信息>>

ConcurrentHashMap<String, Map<String, Lease<InstanceInfo>>>

Renew：服务续约

Eureka客户会每隔30秒发送一次心跳来续约。 通过续约来告知Eureka Server该Eureka客户仍然存在，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。 建议不要更改续约间隔。在DiscoveryClient类下 有renew方法，服务端有InstanceResource类下有renewLease方法接受服务续约（查询租约【根据服务实例和appNanme和instanceInofd查询】，存在则判断状态是否是unknow。否则更新时间【更新参数lastUpdateTimestamp】,然后统计每分钟续约次数，用于自我保护，更新租约中有效时间）

Fetch Registries：获取注册列表信息

Eureka客户端从服务器获取注册表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期（每30秒钟）更新一次。每次返回注册列表信息可能与Eureka客户端的缓存信息不同， Eureka客户端自动处理。如果由于某种原因导致注册列表信息不能及时匹配，Eureka客户端则会重新获取整个注册表信息。 Eureka服务器缓存注册列表信息，整个注册表以及每个应用程序的信息进行了压缩，压缩内容和没有压缩的内容完全相同。Eureka客户端和Eureka 服务器可以使用JSON / XML格式进行通讯。在默认的情况下Eureka客户端使用压缩JSON格式来获取注册列表的信息。一般服务续约 有两个参数可以调试，clieant发送 心跳间隔的时间参数和服务端剔除实例的时间参数。但一般不建议修改

分为全量拉去 和增量拉去，（recentlyChangedQueue）增量拉去3分钟的注册列表

Eureka client第一次启动的时候，回去一次性全量拉取server端的注册表信息，缓存在本地，后续会通过一个定时调用的线程任务，每隔一定时间（默认30s），发动请求到server端，拉取增量的注册表。

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。该下线请求不会自动完成，它需要调用以下内容：DiscoveryManager.getInstance().shutdownComponent()；

服务端：也要获取读锁，查找lease，添加到最近下线的服务列表中，判断是否有租约，有则设置租约下线时间，将状态改为delete，从注册表中删除。然后添加下线记录【用于增量拉去注册表】。最后释放锁

Eviction 服务剔除

在默认的情况下，当Eureka客户端连续90秒没有向Eureka服务器发送服务续约，即心跳，Eureka服务器会将该服务实例从服务注册列表删除，即服务剔除。（源码中使用的是TimerTask方式执行的schedule任务，多个timerTask并发执行定时任务的情况下若有一个timertask异常没有捕获，其他任务也会终止，一般建议使用ScheduledExecutorService来代替timerTask）

Replicate 状态更新

集群情况下，server之间会发送Replicate，采用复制的方式进行注册信息的同步，但设计时先保证可用性，几个节点 挂掉也不会影响正常节点工作

基本流程如下：

InstanceResource类的statusUpdate()方法，是状态更新的入口，用来处理状态更新请求。

然后，调用PeerAwareInstanceRegistryImpl类的statusUpdate()方法，在该过程主要实现状态更新事件同步到集群中的其他节点

最后，调用AbstractInstanceRegistry类的statusUpdate()方法，该方法就是更新实例状态的真正方法。

自我保护模式

当一个server出现，会尝试向peer节点获取所有 的实例信息。获取成功则更新阈值。若服务端接受的续约在15分钟内低于百分之85则服务器开启自我保护，不在剔除注册列表信息。防止因网络原因而导致原本没有问题的服务被清除

流程：isSelfPreservationModeEnabled判断是否为true。接着就是判断大小，大【isLeaseExpirationEnabled为true】剔除，小isLeaseExpirationEnabled为false，不剔除放回

有三个参数，一个是注册中心期望每分钟收到客户端实例续约的阈值（实例总数\*阈值因子），一个是最后一分钟收到的客户端实例续约总数。当前实例续约总数达不到阈值时为开启条件即不清理

当服务节点发生注册，下线，过期，状态变更等变化时

1.在内存中更新注册表信息

2.同时过期掉readWriteCacheMap缓存，缓存清除只是会去清除readWriteCacheMap这个缓存， readOnlyCacheMap 只读 缓存并没有更新，也就说当客户端的信息发生变化之后， 只读缓存不是第一时间感知到的。只读缓存的更新只能依赖那个30秒的定时任务来更新。

3.一段时间后（默认30s），后台线程发现readWriteCacheMap缓存为空，于是也将readOnlyCacheMap中的缓存清空

4.当有服务消费者拉取注册表信息时，会调用ClassLoader的load方法，将内存中的注册表信息加载到各级缓存中，并返回注册表信息。

在Eureka Server 中会有两个线程，一个是定时同步两个缓存的数据，默认30s，一个是定时检测心跳故障，默认90s。 作者：hsx728 https://www.bilibili.com/read/cv8729685 出处：bilibili

实例自动过期

在Eureka-Server启动的时候，会启动一个定时任务，用来清理过期的客户端

initEurekaServerContext()方法下的openForTraffic开启定时清理过期客户端的定时器。清理方法是个同步方法

EvictionTask：是用来清理过期客户端的任务类 serverConfig.getEvictionIntervalTimerInMs() ： 默认为60秒 , 可配置。

Lease.isExpire()//判断是否过期，当前时间必须要大于实际最后更新时间180秒，才会认为他过期

《分批过期机制》 每次最多过期15%的机器，超过15%则不会自动过期，如果不想启用这种机制，那么可以关闭自我保护机制，同时设置registrySizeThreshold = 0； 就可以一次性过期。

***客户端***

工作流程：

EurekaClientAutoConfiguration

EurekaDiscoveryClient

DiscoveryClient

启动：读取与eurekaServer交互的配置信息封装成EurekaClientConfig

读取自身配雪信息封装EurekaInstanceConfig

从eureka服务端拉去注册信息，缓存到本地

服务注册

初始化三个定时器（1.发送心跳2.缓存刷新3.按需注册）

执行：定时发送心跳到eureka服务端，维持在注册表中的服务续约

定时从eureka服务服务端拉去注册表信息，更新本地缓存

Statuschangelisten监控自身变换，如果变了，重新发起注册

销毁：从eurekaserver销毁自身，一般情况下，应用服务在关闭的时候，eurekaclient会主动向eurekaserver注销自身在注册表中的信息

增量拉取和全量拉取

拉取注册表之全量拉取源码解读com.netflix.discovery.DiscoveryClient#getAndStoreFullRegistry

注册表更新会有个版本控制，如果版本不对，那么就不会更新，拉取到全量列表后要进行过滤掉离线服务

增量拉取com.netflix.discovery.DiscoveryClient#getAndUpdateDelta

判断增量列表是否为null，为null说明eureka server 不允许增量更新

比较本地注册表的reconcileHashCode 和 服务器的是否一致，不一致则拉取全量注册表进行替换

updateDelta增量更新（遍历增量注册列表，然后看每个服务实例是否在本地缓存中存在，如果存在则基于实例的getActionType来做出对应的行为，否则进行新增，其中新增和修改是进行服务实例删除再新增，删除是移除服务实例）

EurekaClient获取全量注册表信息实现方式：

这里是EurekaClient第一次全量获取注册表的实现原理，从注册中心拉取到注册表后，EurekaClient会将注册表信息保存在本地的list中。

这里也要提下EurekaServer中的两层缓存机制，我们每次从注册中心拉取注册表时都是直接走的缓存，缓存使用的是谷歌提供的GuavaCahe

EurekaClient获取增量注册表实现方式：

EurekaClient每隔30s去注册中心拉取注册表增量信息，拿回来后和本地缓存的注册信息进行比对，一顿增删改查操作后覆盖缓存中的注册信息数据。下面是增量获取注册表信息的代码示例，这里会从recentlyChangedQueue中获取存在变化的实例信息，最后还会设置一个appHashCode值

1. **你研究过eureka的源码吗？**

在创建一个eureka服务中，会在启动类上添加一个注解@EnableEurekaServer，该注解会创建一个Marker的空类并注入。作为标记。若去查依赖中的sprin.factories里得EurekaServiceAutoConfiguration就会发现条件注解中通过判断是否有marker这个类来决定是否生效。所以有了@EnableEurekaServer和pom依赖就可以开启一个eureka服务。

在eureka自动化配置类上还添加了@import注解，导入了EurekaServerInitializerConfiguration类，只要导入了该类，该类就会执行一个start的线程来帮助eureka初始化。因为导入的初始化配置类实现了一个SmartLifecycle（它继承有个lifecycle，两者都是可以在容器加载所有bean并初始化后，执行异步的start方法，里面做了初始化eureka环境，拉去其他节点的信息）

自我保护机制源码分析：第一步，我们引入Eureka Server 依赖。

第二步，我们找到eureka-core jar包下的路径为com.netflix.eureka下的registry包

第三步，进入AbstractInstanceRegistry 类，找到evict方法，这个是定期剔除任务的线程最终执行的方法

第四步，我们找到isLeaseExpirationEnabled()方法的实现，里面会判断是否开启了自动保护，还有每分钟最小续约阈值和最后一分钟实例续约次数，比较判断，若正常就会进行剔除操作。同时服务注册还有服务下线都会更新每分钟最小续约次数

1. **你是如何优化eureka的？**

不同数量服务的自我保护优化

|  |
| --- |
| 1配置参数：自我保护阈值85%，剔除间隔减小，续约间隔调小，阈值更新间隔，自我保护开关  2服务数量比较少，不建议开启自我保护，可以将剔除间隔时间设置短一点，及时剔除不可用服务，使服务 快速下线。防止消费者获取到一个无效的服务实例  3服务数比较大，建议开启自我保护。  4开发时可以部署两个server互相注册，实现高可用的同时保证server开启自我保护机制 |

快速下线

|  |
| --- |
| Eureka Server在启动时会创建一个定时任务，每隔一段时间(默认60秒)，从当前服务清单中把超时没有续约(默认90秒)的服务剔除。我们可以把定时任务间隔的时间设置得短一点，做到快速下线。防止拉取到不可用的服务。 |

三级缓存优化

|  |
| --- |
| 配置参数Use-Read-Only-response-Cache设置false禁用。让消费者拉取的更快一些  Response-cache-update-interval-ms:两个缓存同步的时间减小，默认30秒，可以设置成一秒  1原理流程：定时任务每 30s 将 readWriteCacheMap 同步至 readOnlyCacheMap，每 60s 清理超过 90s 未续约的节点，Eureka Client 每 30s 从 readOnlyCacheMap 拉取服务注册信息，而服务的注册则在 registry 更新信息 包含了三个缓存变量一级缓存：registry(嵌套concurrentHashMap)、readWriteCacheMap（本质是guava缓存）、readOnlyCacheMap 保存服务注册信息。写往registry上写再更新到rw上。读从ro读，读不到再从rw再从registry  2优点：保证了内存中注册表不会出现读写冲突问题，提升接受请求的性能。保证了CAP中的AP  生产环境的弱一致性问题。默认从readOnlyCacheMap中读取， readWriteCacheMap每隔30s才同步到readOnlyCacheMap，数据不是强一致性的，这也是没有 实现C的原因。  3优化：关闭从readOnly读注册表，直接去readWriteCacheMap中读。减少这两张表同步间隔时间实现快速剔除 （服务快速上线，快速下线） |

客户端优化

|  |
| --- |
| 1拉取注册表间隔时间（刷新注册表间隔）  2续约间隔时间都可以优化（心跳）  3饥饿加载，防止第一次请求超时  4缺少心跳的过期时间减小  Service-url 打乱配置，不要所有服务都写一样顺序的配置 |

1. **eureka客户端注册一个实例为什么这么慢**

在DafaultEurekaClientConfig类下有一个延迟向服务端注册 的方法，默认延迟时间为40秒。客户端本地缓存的注册表信息需要30秒才会刷新发现其他新注册的实例

**19、如果用户量大，区域分布广，如何保证服务可用**

为了保证服务 的可用，会在配置文件中使用region和zone来指定。可以调取其他区域的服务，一般用region划分，再用zone划分具体的机房

服务注册：要保证服务注册到同一个zone内的注册中心，因为如果注册到别zone的注册中心的话，网络延时比较大，心跳检测很可能出问题。 服务调用：要保证优先调用同一个zone内的服务，只有在同一个zone内的服务不可用时，才去调用别zone的服务。

**20、你在生产中遇到哪些问题**

下线重现问题

我在向注册中心发送delete请求下线再关闭微服务后，发现server中还是存在注册的实例，等于白下线。后来我debug发现内部运行机制：客户端每个30秒续约，如果注册列表中没有，会重新注册。所以没有及时停掉的话，服务又会再次回到服务列表中？

集群搭建问题

集群搭建完成后，发现服务节点均在unavailable-replicas下，即说明集群搭建失败，各节点之间不能互相通信，网上查找了各种资料，终于解决，现将问题处理的过程记录如下：

1检查各节点的spring.application.name必须一样

2eureka.instance.hostname``应该与host文件中的映射相同

3defaultZone中的url，不能包含自己，包含其他服务器节点，如：<http://hostname/>端口/eureka

4开启自注册和互相注册

EurekaServer集群重启问题

在生产环境重启Eureka Server集群的时候，发现订单客户端调用分布式Id生成服务出错

思考：Eureka Client缓存由一个定时线程去刷新，每30秒执行一次增量更新，通过后续调试，发现Eureka Client的本地缓存是空的。 由此引发了一个问题，当Eureka Server正在重启或者重启完成，Eureka Client来获取注册信息，然后更新到本地出了问题。发生问题的原因已经很明显了，就是在Eureka Server重启的时候，注册信息为空，刚好被Eureka Client获取到。也就是说Eureka Server刚刚启动的时候，他作为server端的注册信息是空的。 只能依赖后续集群续约同步的方式，慢慢补全自身的信息。

解决：将fetch-register修改为true， 这样在Eureka Server 刚刚启动的时候，就可以将注册信息全部注册到自己的节点上去。完整方案：添加eureka.instance.initial-status: STARTING 表示在Eureka Server 刚刚启动的时候，默认不主动去注册，等待服务同步数据完成之后。还可以自定义过滤器，对server状态进行判断，只有server启动并完成同步数据后才提供服务

1. **如何估算服务，你们eurekaServer的日均访问量是多少？**

20个服务，每个服务部署5个，一共有100个client

心跳间隔加起来一天28万次，拉去间隔也是28万次，一天也就50多万。而eureka服务器处理的速度一秒可以处理1000个。1个server完全可以承受几十上百个client。集群可以保证A高可用，P分区容错性

访问量几十万次，没有上百万。服务测算，意义就是针对不同的配置的机器，进行承受量估算

1. **eureka常见面试问题**

什么是否开启自我保护？

（每分钟最小续约数量，最后一分钟实例续约数量）

网络波动，集群同步问题

|  |
| --- |
| Eureka Server 允许同一时刻被任意Eureka Client做写入操作， 当发生客户端向Eureka Server 1发起注册时，明明在Eureka Server注册成功了，但是返回结果的时候，发生网络错误，误让Eureka Client 认为server 1不可用,正好这个时候Eureka Client的状态发生变更（会去修改laseDirtyTimestamp），进而像server 2发起注册， 这个时候问题就来了。有如下机制，可以确保集群状态下数据的最终一致性  1.当向server 2 注册之后， server 2 会同时向server 1 同步client端的注册信息，server 1由于之前被client注册过，同时也向server 2同步注册信息 ， 这就造成了，两个Eureka Server互相同步请求的问题， 这个时候就是 通过 lastDirtyTimestamp这个变量来判定以谁的为准， 谁的lastDirtyTimestamp 大，则以谁的为准。  2.单单通过lastDirtyTimestamp 这个机制，并没有办法保证集群之间的数据一致性，当网络出现异常的之后，集群同步请求一直不成功，同时任务的过期时间已经到了（集群同步是通过发布任务，让多线程批量去处理的，同时任务有过期时间），这个时候会造成集群之间的数据不一致的情况，这个时候，就是通过heatbeat机制， 当心跳发生如下三种情况的是，客户端会重新发起注册：https://www.shared-code.com/article/65 |

Eureka Server端实例过期，会同步到其他server服务器吗？

|  |
| --- |
| 实例过期不会同步到集群中其他服务器，每个Eureka Server 内部都有自己的过期定时任务，每个server  自己负责清理自己的，互不相干，会触发同步操作的是如下几个操作，  register 注册 renew 心跳续约 cancle 客户端主动下线  stateUpdate 添加覆盖状态 deleteStateovrride 删除覆盖状态 |

Eureka Client刚启动是否会立即注册？client端启动之后会立马像服务端发起注册，不是等待40秒

Eureka Server集群宕机后，客户端是否可用？

|  |
| --- |
| 情景一  Eureka Client 启动的时候，会主动去全量获取一次注册信息，如果这个时候Eureka Server集群  已经宕机，那么Eureka Client端是不可用的。  情景二  如果Eureka Client 启动时全量获取注册信息成功，在之后的运行过程当中，Eureka Server集群宕机了  那么这个时候，Eureka Client是不受影响的，Eureka Client会缓存注册信息在本地，如果后续增量获取  注册信息失败，也不会影响本地现有的注册信息。 唯一影响的是，由于Eureka Client所拥有的数据  得不到更新，这个时候，新增客户端，或者有客户端的信息发生变化，那么Eureka Client是感知不到的。 |

关键时间间隔有哪些

|  |
| --- |
| Eureka Client：  更新本地注册信息 默认：30秒  发送心跳 默认：30秒  检查应用信息，如果发生变化，则重新发起注册 默认：30秒  Eureka Server：  只读缓存过期时间 默认：30秒  读写缓存过期时间 默认：180秒  清理过期的任务 默认：60秒  租约过期时间 默认：90秒  启动时同步节点数据，为空等待时间 默认：30秒  清空每分钟续约次数 默认：60秒 |

Eureka Client 90秒没有心跳一定会过期吗？

|  |
| --- |
| 答案是，不一定会过期  情景一：自我保护模式的开启  情景二：Eureka Server中租约过期时间默认是90秒，服务端维护了一个lastUpdaeTimestap这个变量，  在续约的时候，更新最后修改时间，是用当前时间加了90秒上去的。判断过期，当前时间，一定要大于最后修改时间+90秒， 由于最后修改时间，在续约的时候已经加了90秒了。这里再加个90秒，就是180秒，要大于最后修改时间180秒才会被判定为过期。 |

1. **部分服务调用的流程**

登录：

（1）获取验证码，api-passenger服务，调用service-verification-code服务获取。该服务生成一个验证码，该码的有效期用redis设置过期时间，码放在redis，利用key的过期特性。

（2）发送验证码，api-passenger服务，调用Service-sms服务：发送验证码，还包括手机号，该服务利用第三方api发送（阿里）sms做成通用的：不仅能发验证码短信，还可以发别的

（3）校验验证码，api-passenger服务，调用service-verification-code服务，校验用户传进来的验证码

（4）用户，api-passenger服务调用service-passenger-user，用户入库，或查询，返回token给api-passenger

叫车计价：

1. api-passenger===》service-order，创建一个订单（高德服务需要订单号）
2. Service-order===》service-map，取起始点到终点的距离。还有时间
3. Service-order===》计价服务，取计价规则，然后service-order存计价规则，缺少派单规则就不能叫车
4. Service-order===》计价服务，去计价。返回计价信息。Service-order存计价信息
5. Service-order===》api-passenger，将计价信息保存在api-passenger。前端得到price字段。

下单：

1. Api-passenger===》order，改变订单状态，（校验：有未支付的不能继续，重新绑定价格）
2. Order===》计价系统。检查计价规则，有变就重新预估，将计价写到数据库，返回给order
3. Order===》派单系统。派单
4. Order===》api-passenger
5. **Api怎么设计的**

不仅支持单个功能的参数匹配，还要支持其他服务功能的参数匹配，这才是分割成单独服务的最大原因

Url：

发送验证码

发送短信

参数：

短信模板：${code]

1. **如何保证估价的准确性**

数据结构，Bigdecimal，int或long。用double的会出现问题，造成精度缺失，会出现0.9999999

用分去运算。计价业务可能会被问的很细。计价里用到了多线程，可以吹流批

用bigdecimal，不要用他自己，自己转换，写成String

New BigDecimal(“1.1”);尽量用参数为String的构造函数，遇到double，使用double.toString(double)，获取该结果，请使用valueOf(double)方法（bigdecimal参数.doubleValue(）将前者转换为double类型)

1.用String，float或者double变量构建BigDecimal对象;

2.通过调用BigDecimal的加，减，乘，除等相应的方法进行算术运算;

3.把BigDecimal对象转换成float，double，int等类型.

BigDecimal都是不可变的（immutable）的，在进行每一步运算时，都会产生一个新的对象，所以在做加减乘除运算时千万要保存操作后的值。

成就感比较高的事？

当时在一家公司实习。有一件事特别急，中午12的时候开会要上报信息。然后1点多就要上报。领导就拉了我还有个山东的硕士，开会的时候就讨论嘛，这些信息该怎么统计，里面有个字段是地理编码信息，那个山东的说递归，扯了一大堆。然后自己就去吃饭了。然后我就在研究这些编码信息的时候。突然看到这些编码信息其实规律的。比如一个北京11四个0，海淀区110100。然后通过前面的这些编码信息。来精确到每个大街的位置信息。将那10几万条的数据做了统计，然后上报成功。当时就特别开心。我想这种从信息中发现规律，在紧急状态下人的效率超高的事情是一件能获取成就感的事情

eureka拉取注册表：

