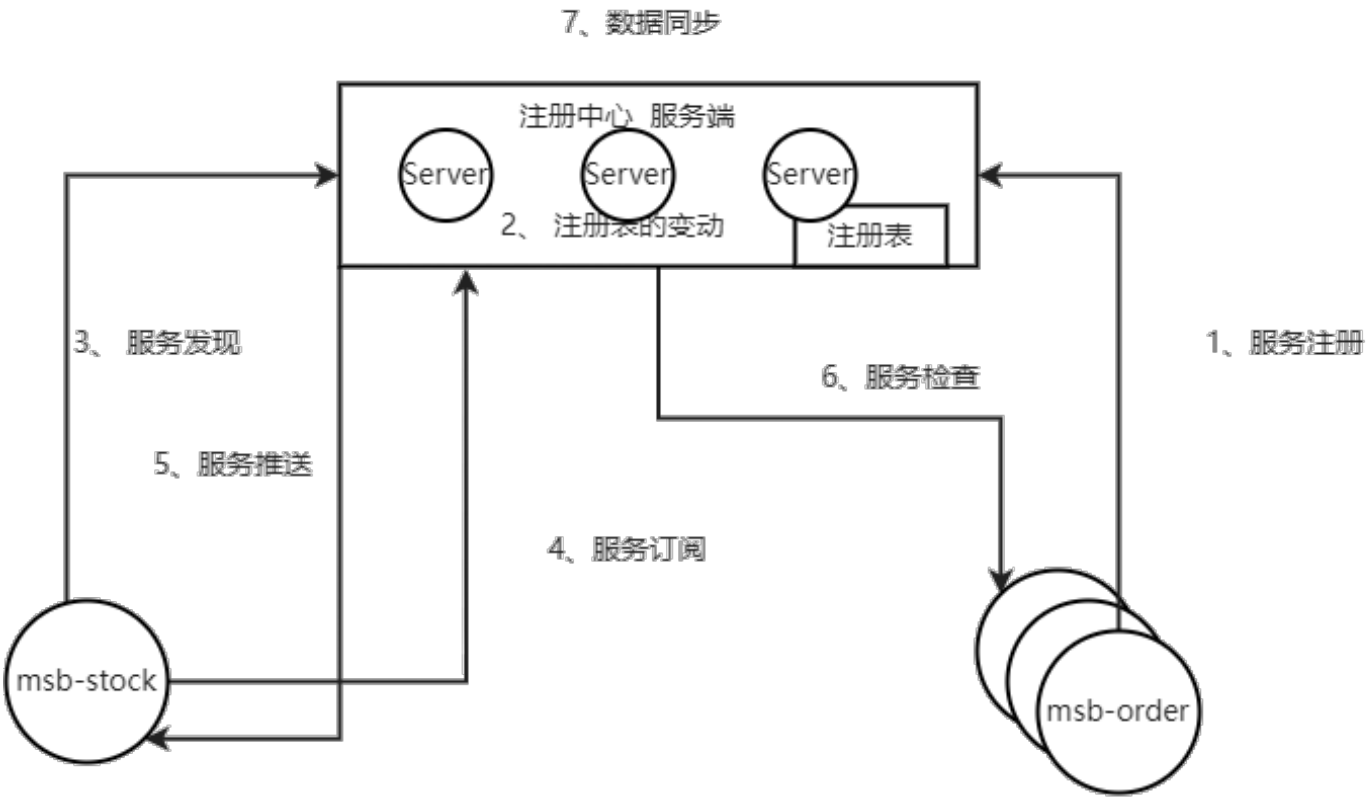
1、如何设计一个注册中心？

高可用： 集群
高并发： 响应时间、吞吐量、并发用户数等等 ：
增加服务性能、 扩展服务实例
高性能： 程序处理速度

数据存储结构、 通信机制、 集群同步



服务 、实例

- 服务注册
- 注册表结构设计

- 服务发现
- 服务订阅
- 服务推送
- 健康检查
- 集群同步：设计到数据同步，数据同步我们有哪些协议 raft 、distro、ZAB

2、Nacos1.x作为注册中心的原理？

1、使用Http发送注册

2、查询服务提供方列表

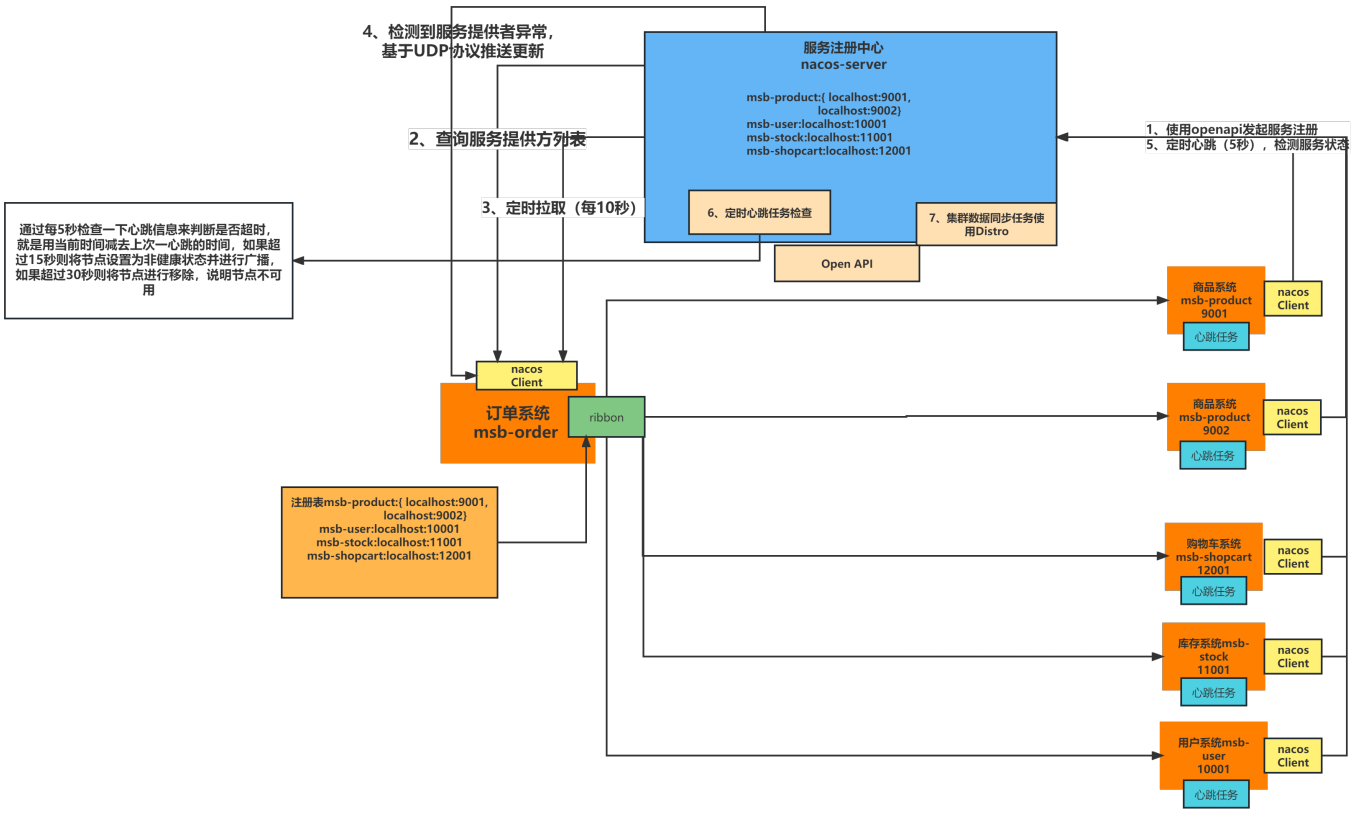
3、定时拉取（每10秒）

4、检测到服务提供者异常，基于UDP协议推送更新

5、定时心跳（5秒），检测服务状态

6、定时心跳任务检查

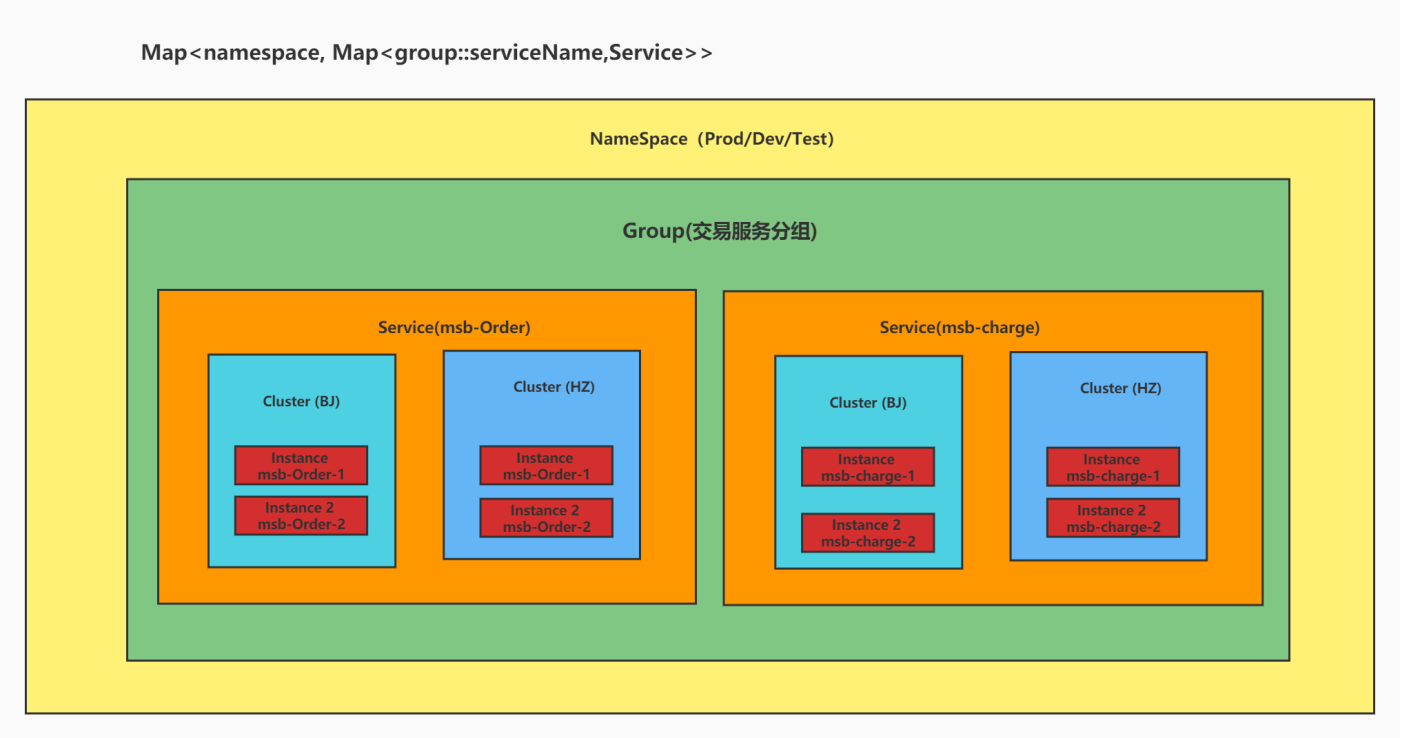
7、集群数据同步任务使用Distro



3、Nacos服务领域模型有哪些？

模型名称	解释
Namespace	实现环境隔离，默认值public
Group	不同的service可以组成一个Group，默认值Default-Group
Service	服务名称
Cluster	对指定的微服务虚拟划分，默认值Default
Instance	某个服务的具体实例

Nacos服务注册中心于发现的领域模型的最佳实践。

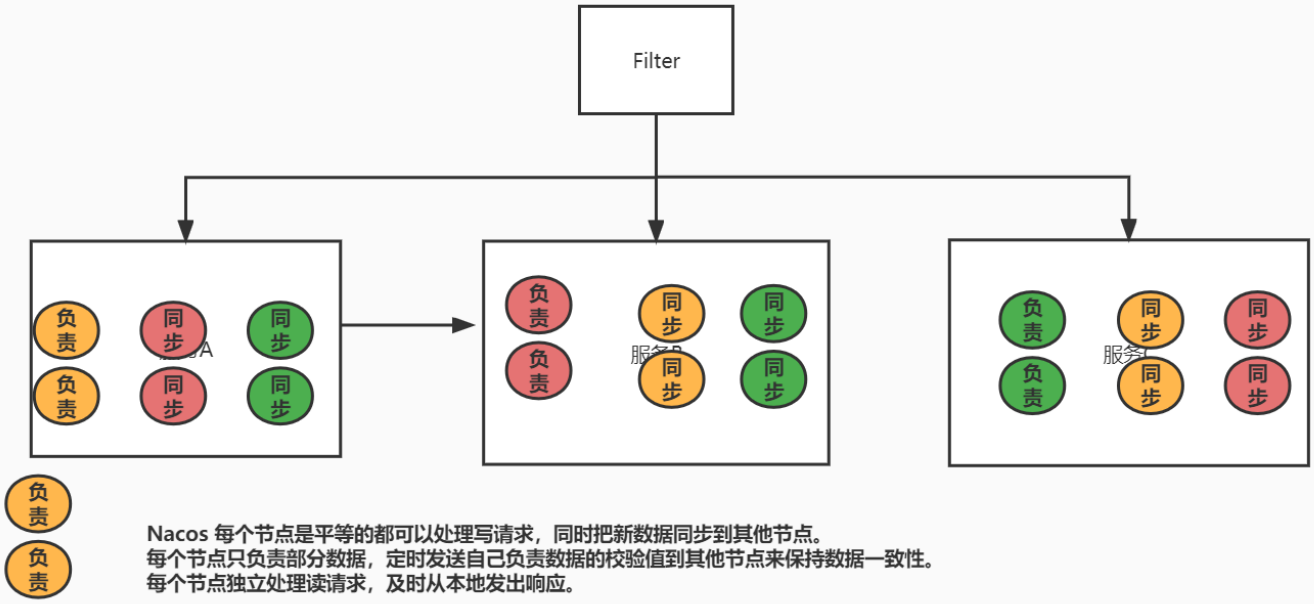


4、Nacos中的Distro协议

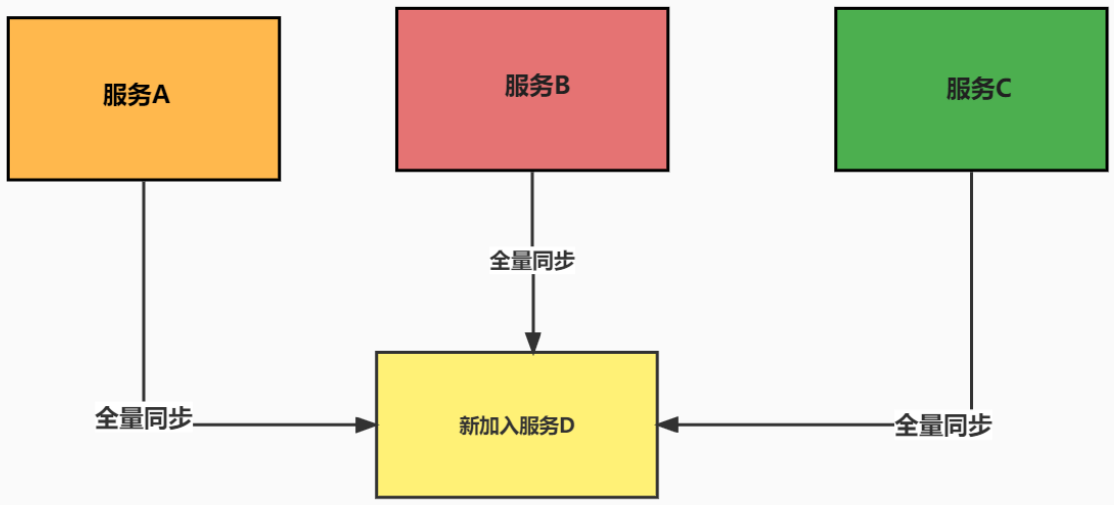
- Nacos 每个节点自己负责部分的写请求。
- 每个节点会把自己负责的新增数据同步给其他节点。
- 每个节点定时发送自己负责数据的校验值到其他节点来保持数据一致性。
- 每个节点独立处理读请求，及时从本地发出响应。
- 新加入的 Distro 节点会进行全量数据拉取。

（具体操作是轮询所有的 Distro节点，通过向其他的机器发送请求拉取全量数据。）

Nacos 每个节点自己负责部分的写请求。
每个节点会把自己负责的新增数据同步给其他节点。
每个节点定时发送自己负责数据的校验值到其他节点来保持数据一致性。
每个节点独立处理读请求，及时从本地发出响应。
新加入的 Distro 节点会进行全量数据拉取。
（具体操作是轮询所有的 Distro节点，通过向其他的机器发送请求拉取全量数据。）



新加入的 Distro 节点会进行全量数据拉取。具体操作是轮询所有的 Distro 节点，通过向其他的机器发送请求拉取全量数据。



5、配置中心的技术选型

功能点	Spring Cloud Config	Apollo	Nacos
版本管理	支持(Git)	支持	支持
配置实时推送	支持(Spring Cloud Bus)	支持(HTTP长轮询1s内)	支持(HTTP长轮询
配置回滚	支持(Git)	支持	支持
灰度发布	支持（调用机器接口）	支持	**不支持**
权限管理	支持(依赖Git)	支持	支持
配置生效时间	**重启**	实时	实时
审计	支持(依赖Git)	支持	不支持
多集群	支持	支持	支持
多环境	支持	支持	支持
client本地缓存	**不支持**	支持	支持
监听查询	支持	支持	支持
运维成本	高	中等	较低
多语言	仅Java	主流语言，提供了Open API	主流语言，提供了
配置项维护	基于git	统一界面	统一界面
社区活跃度	1.8k Star	27.4k Star	24.2k Star

1、社区活跃度

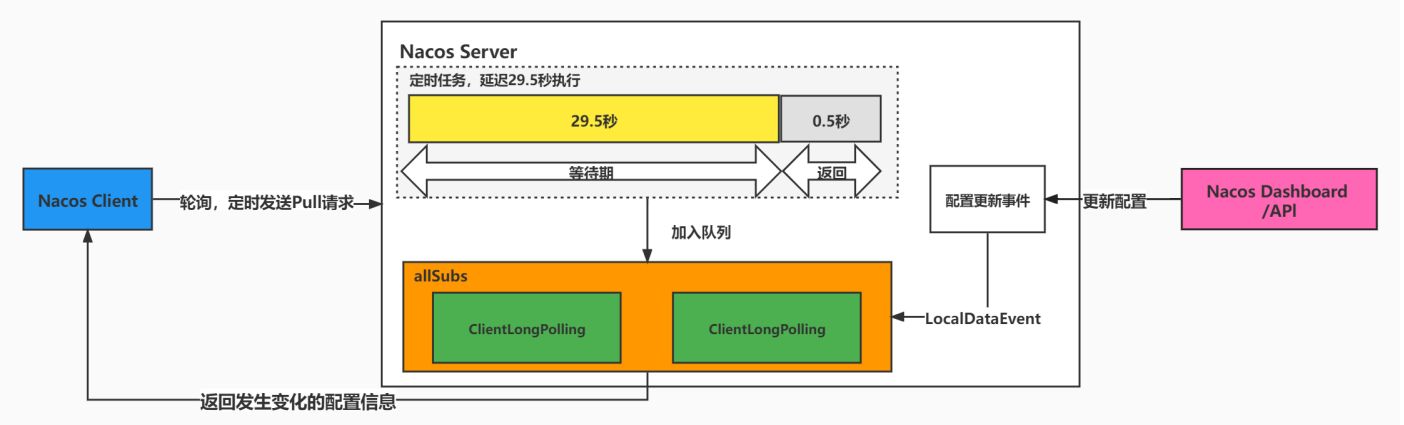
2、自己的技术栈 我们以前 rocketmq 和kafka rocketmq :不支持指定时间的延时消息、不支持事务消息

Java kafka scala ,我们技术栈是java ,所以我们选择rocketmq

3、产品功能 rocketmq或者kafka技术栈是否hold住

6、Nacos1.x配置中心长轮询机制？

客户端会轮询向服务端发出一个长连接请求，这个长连接最多30s就会超时，服务端收到客户端的请求会先判断当前是否有配置更新，有则立即返回，如果没有服务端会将这个请求拿住“hold”29.5s加入队列，最后0.5s再检测配置文件无论有没有更新都进行正常返回，但等待的29.5s期间有配置更新可以提前结束并返回。



7、Nacos配置中心配置优先级？

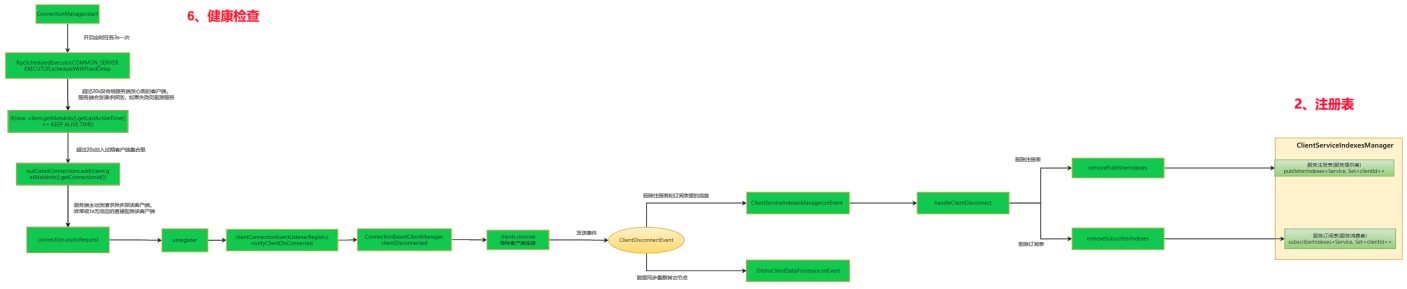
```
#作用：顺序
#${application.name}-${profile}.${file- extension}  nacos-config-prod.yaml
#${application.name}.${file-extension}  nacos-config.yaml
#${application.name}  nacos-config
#extensionConfigs  扩展配置文件
#sharedConfigs  多个微服务公共配置 redis
```

```
spring.application.name=nacos-config

server.port=8081
#nacos地址
spring.cloud.nacos.config.server-addr=localhost:8848
#1、只有上面的配置的时候他默认加载文件为：${application.name}  nacos-config
#2、指定文件后缀名称
#加载文件为：${application.name}.${file-extension}
#nacos-config.yaml
#spring.cloud.nacos.config.file-extension=yaml
##3、profile：  指定环境  文件名：${application.name}-${profile}.${file-extension}
##nacos-config-prod.yaml
#spring.profiles.active=prod
```

```
#4、nacos自己提供的环境隔离 ，这里是开发环境下的
#spring.cloud.nacos.config.namespace=ff02931a-6fdb-4681-ac37-2f6d9a0596f8

#5、 自定义 group 配置，这里也可以设置为数据库配置组，中间件配置组，但是一般不用，
# 配置中心淡化了组的概念，使用默认值DEFAULT_GROUP
#spring.cloud.nacos.config.group=DEFAULT_GROUP
#
#6、自定义Data Id的配置 共享配置（sharedConfigs）
#spring.cloud.nacos.config.shared-configs[0].data-id= common.yaml
##可以不配置，使用默认
#spring.cloud.nacos.config.shared-configs[0].group=DEFAULT_GROUP
## 这里需要设置为true，动态可以刷新，默认为false
#spring.cloud.nacos.config.shared-configs[0].refresh=true

# 7、扩展配置(extensionConfigs)
# 支持一个应用有多个DataId配置，mybatis.yaml datasource.yaml
#spring.cloud.nacos.config.extension-configs[0].data-id=datasource.yaml
#spring.cloud.nacos.config.extension-configs[0].group=DEFAULT_GROUP
#spring.cloud.nacos.config.extension-configs[0].refresh=true


#作用：顺序
#${application.name}-${profile}.${file- extension}  msb-edu-prod.yaml
#${application.name}.${file-extension}  nacos-config.yaml
#${application.name}  nacos-config
#extensionConfigs  扩展配置文件
#sharedConfigs  多个微服务公共配置 redis
```

8、Nacos2.x客户端探活机制？

Nacos服务端会启动一个定时任务，每3秒执行一次，查看所有连接是否超过20s没有通信，如果超过
20秒没有通信，服务端就会给客户端发送一个请求，进行探活，如果能正常返回就表示这个服务为正常服务，如果不能正常返回就将其连接删除。



9、Ribbon底层怎样实现不同服务的不同配置

为不同服务创建不同的Spring上下文

10、为什么Feign第一次调用耗时很长？

ribbon默认是懒加载的，只有第一次调用的时候才会生成ribbon对应的组件，这就会导致首次调用的会很慢的问题。

```
ribbon:
  eager-load:
    enabled: true
      clients: msb-stock
```

11、Ribbon的属性配置和类配置优先级

类的优先级更高

12、Feign的性能优化？

Feign底层默认是 JDK自带的HttpURLConnection，它是单线程发送HTTP请求的，不能配置线程池，我们使用Okhttp或者HttpClient来发送http请求，并且它们两个都支持线程池。

常见HTTP客户端

- **HttpClient**
  HttpClient 是 Apache Jakarta Common 下的子项目，用来提供高效的、最新的、功能丰富的支持 Http 协 议的客户端编程工具包，并且它支持 HTTP 协议最新版本和建议。HttpClient 相比传统 JDK 自带的 URLConnection，提升了易用性和灵活性，使客户端发送 HTTP 请求变得容易，提高了开发的效率。
- **Okhttp**
  一个处理网络请求的开源项目，是安卓端最火的轻量级框架，由 Square 公司贡献，用于替代 HttpUrlConnection 和 Apache HttpClient。OkHttp 拥有简洁的 API、高效的性能，并支持多种协议（HTTP/2 和 SPDY）。
- **HttpURLConnection**
  HttpURLConnection 是 Java 的标准类，它继承自 URLConnection，可用于向指定网站发送 GET 请求、 POST 请求。HttpURLConnection 使用比较复杂，不像 HttpClient 那样容易使用。

- **RestTemplate**
  RestTemplate 是 Spring 提供的用于访问 Rest 服务的客户端，RestTemplate 提供了多种便捷访问远程 HTTP 服务的方法，能够大大提高客户端的编写效率。

13、Feign怎样实现认证的传递？

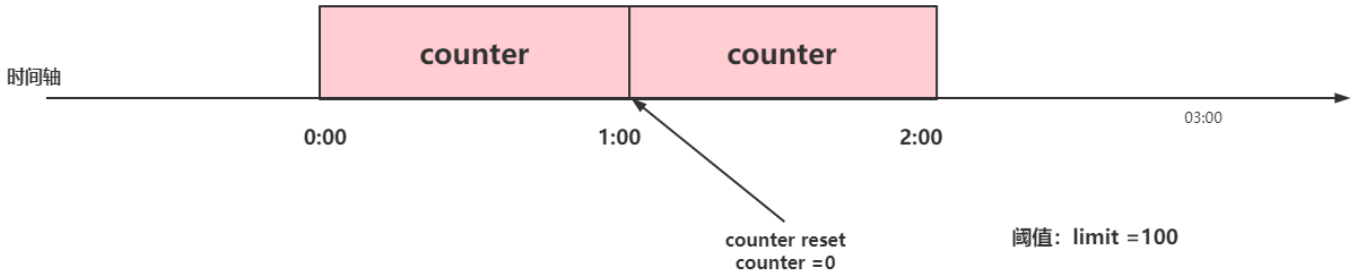
实现接口RequestInterceptor，通过header实现认证传递

```
package feign;

public interface RequestInterceptor {
  void apply(RequestTemplate template);
}
```

14、谈谈Sentienl中使用的限流算法

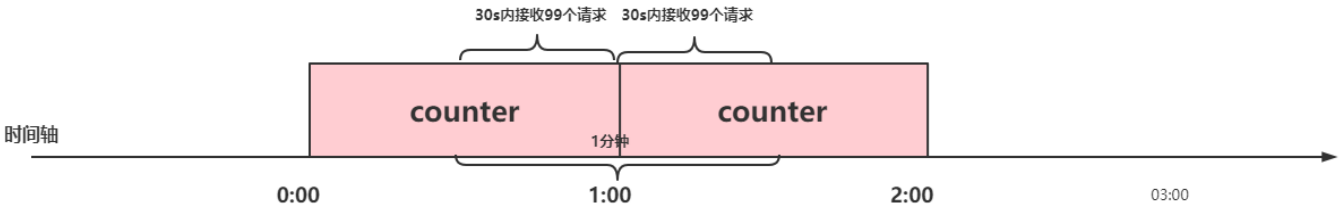
1、计数器固定窗口算法

计数器固定窗口算法是最简单的限流算法，实现方式也比较简单。就是通过维护一个单位时间内的计数值，每当一个请求通过时，就将计数值加1，当计数值超过预先设定的阈值时，就拒绝单位时间内的其他请求。如果单位时间已经结束，则将计数器清零，开启下一轮的计数。



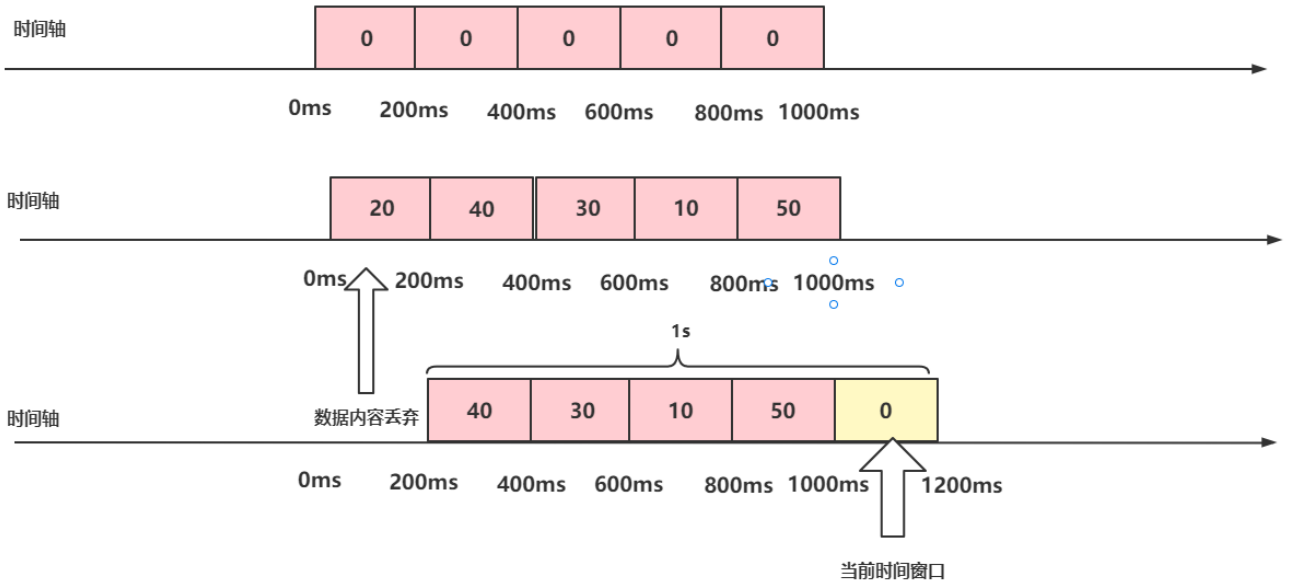
但是这种实现会有一个问题，举个例子:

> 假设我们设定1秒内允许通过的请求阈值是99，如果有用户在时间窗口的最后几毫秒发送了99个请求，紧接着又在下一个时间窗口开始时发送了99个请求，那么这个用户其实在一秒内成功请求了198次，显然超过了阈值但并不会被限流。其实这就是临界值问题，那么临界值问题要怎么解决呢？



2、计数器滑动窗口算法

计数器滑动窗口法就是为了解决上述固定窗口计数存在的问题而诞生。前面说了固定窗口存在临界值问题，要解决这种临界值问题，显然只用一个窗口是解决不了问题的。假设我们仍然设定1秒内允许通过的请求是200个，但是在这里我们需要把1秒的时间分成多格，假设分成5格（格数越多，流量过渡越平滑），每格窗口的时间大小是200毫秒，每过200毫秒，就将窗口向前移动一格。为了便于理解，可以看下图



图中将窗口划为5份，每个小窗口中的数字表示在这个窗口中请求数，所以通过观察上图，可知在当前时间快（200毫秒）允许通过的请求数应该是70而不是200（只要超过70就会被限流），因为我们最终统计请求数时是需要把当前窗口的值进行累加，进而得到当前请求数来判断是不是需要进行限流。

那么滑动窗口限流法是完美的吗？

细心观察的我们应该能马上发现问题，*滑动窗口限流法其实就是计数器固定窗口算法的一个变种*。流量的过渡是否平滑依赖于我们设置的窗口格数也就是统计时间间隔，格数越多，统计越精确，但是具体要分多少格我们也说不上来呀...

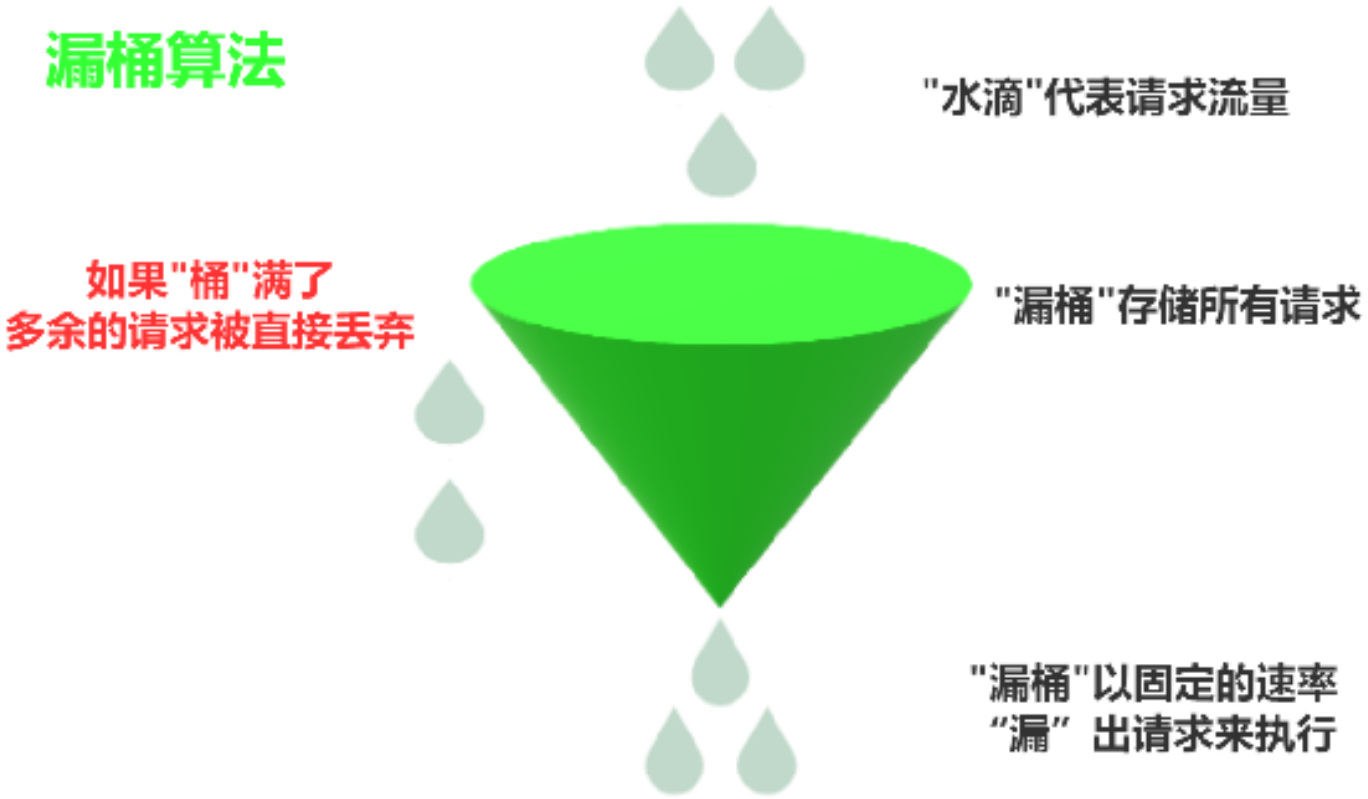
3、漏桶算法

上面所介绍的两种算法都不能非常平滑的过渡，下面就是漏桶算法登场了

什么是漏桶算法？

漏桶算法以一个常量限制了出口流量速率，因此漏桶算法可以平滑突发的流量。其中漏桶作为流量容器我们可以看做一个FIFO的队列，当入口流量速率大于出口流量速率时，因为流量容器是有限的，当超出流量容器大小时，超出的流量会被丢弃。

下图比较形象的说明了漏桶算法的原理，其中水龙头是入口流量，漏桶是流量容器，匀速流出的水是出口流量。



漏桶算法的特点

- 漏桶具有固定容量，出口流量速率是固定常量（流出请求）
- 入口流量可以以任意速率流入到漏桶中（流入请求）
- 如果入口流量超出了桶的容量，则流入流量会溢出（新请求被拒绝）

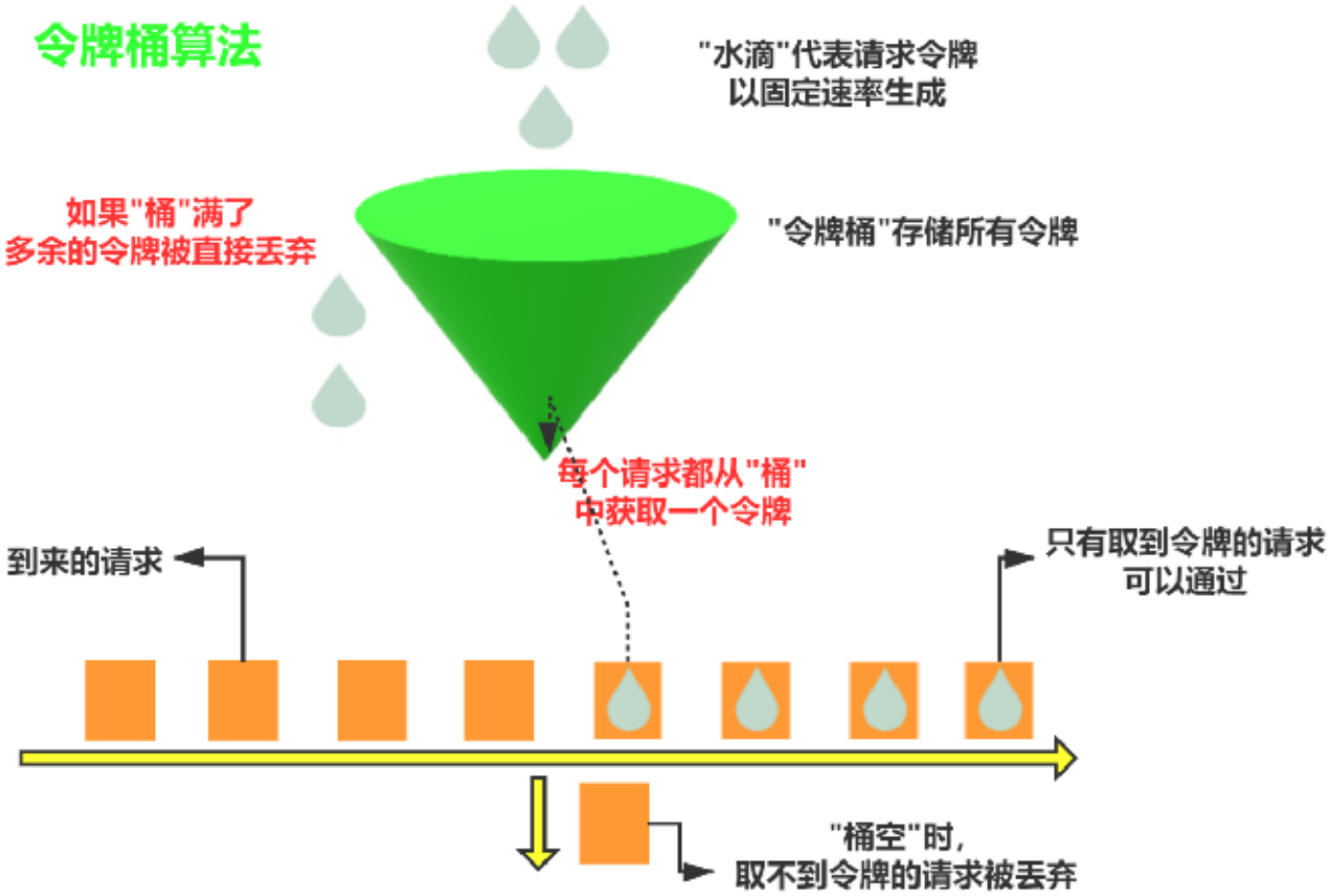
不过因为漏桶算法限制了流出速率是一个固定常量值，所以漏桶算法不支持出现突发流出流量。但是在实际情况下，流量往往是突发的。

4、令牌桶算法

令牌桶算法是漏桶算法的改进版，可以支持突发流量。不过与漏桶算法不同的是，令牌桶算法的漏桶中存放的是令牌而不是流量。

那么令牌桶算法是怎么突发流量的呢？

最开始，令牌桶是空的，我们以恒定速率往令牌桶里加入令牌，令牌桶被装满时，多余的令牌会被丢弃。当请求到来时，会先尝试从令牌桶获取令牌（相当于从令牌桶移除一个令牌），获取成功则请求被放行，获取失败则阻塞活拒绝请求。



令牌桶算法的特点

- 最多可以存发b个令牌。如果令牌到达时令牌桶已经满了，那么这个令牌会被丢弃
- 每当一个请求过来时，就会尝试从桶里移除一个令牌，如果没有令牌的话，请求无法通过。

令牌桶算法限制的是平均流量，因此其允许突发流量（只要令牌桶中有令牌，就不会被限流）

15、谈谈Sentienl服务熔断过程

服务熔断一般有三种状态：

- 熔断关闭状态（Closed）

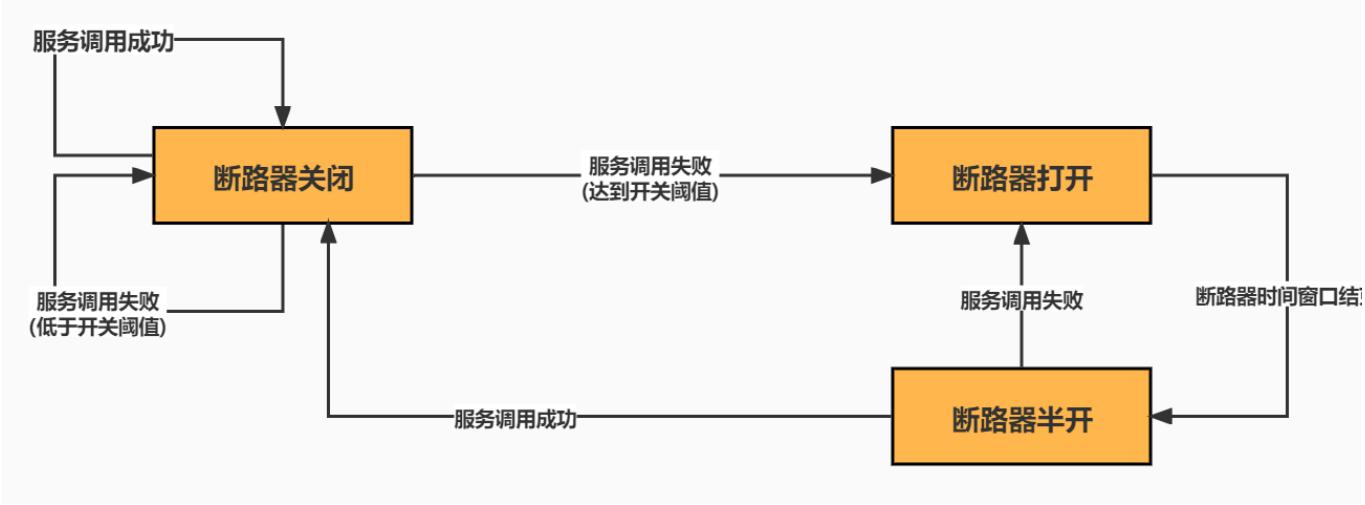
  服务没有故障时，熔断器所处的状态，对调用方的调用不做任何限制。

- 熔断开启状态（Open）

  后续对该服务接口的调用不再经过网络，直接执行本地的fallback方法。

- 半熔断状态（Half-Open）

  尝试恢复服务调用，允许有限的流量调用该服务，并监控调用成功率。如果成功率达到预期，则说明服务已恢复，进入熔断关闭状态；如果成功率仍旧很低，则重新进入熔断关闭状态。



16、在Gateway中怎样实现服务平滑迁移？

使用Weight路由的断言工厂进行服务权重的配置，并将配置放到Nacos配置中心进行动态迁移

Weight路由断言工厂

规则：

该断言工厂中包含两个参数，分别是用于表示组 group，与权重 weight。对于同一组中的多个 uri 地址，路由器会根

配置：

```
spring:
  cloud:
    gateway:
      routes:
      - id: weight_high
        uri: https://weighthigh.org
        predicates:
        - Weight=group1, 8
      - id: weight_low
        uri: https://weightlow.org
```

```
        predicates:
        - Weight=group1, 2
```

group 组，权重根据组来计算
weight 权重值，是一个 Int 的值

17、Seata支持那些事务模式？

概念：Seata 是一款开源的分布式事务解决方案，致力于提供高性能和简单易用的分布式事务服务。Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案。

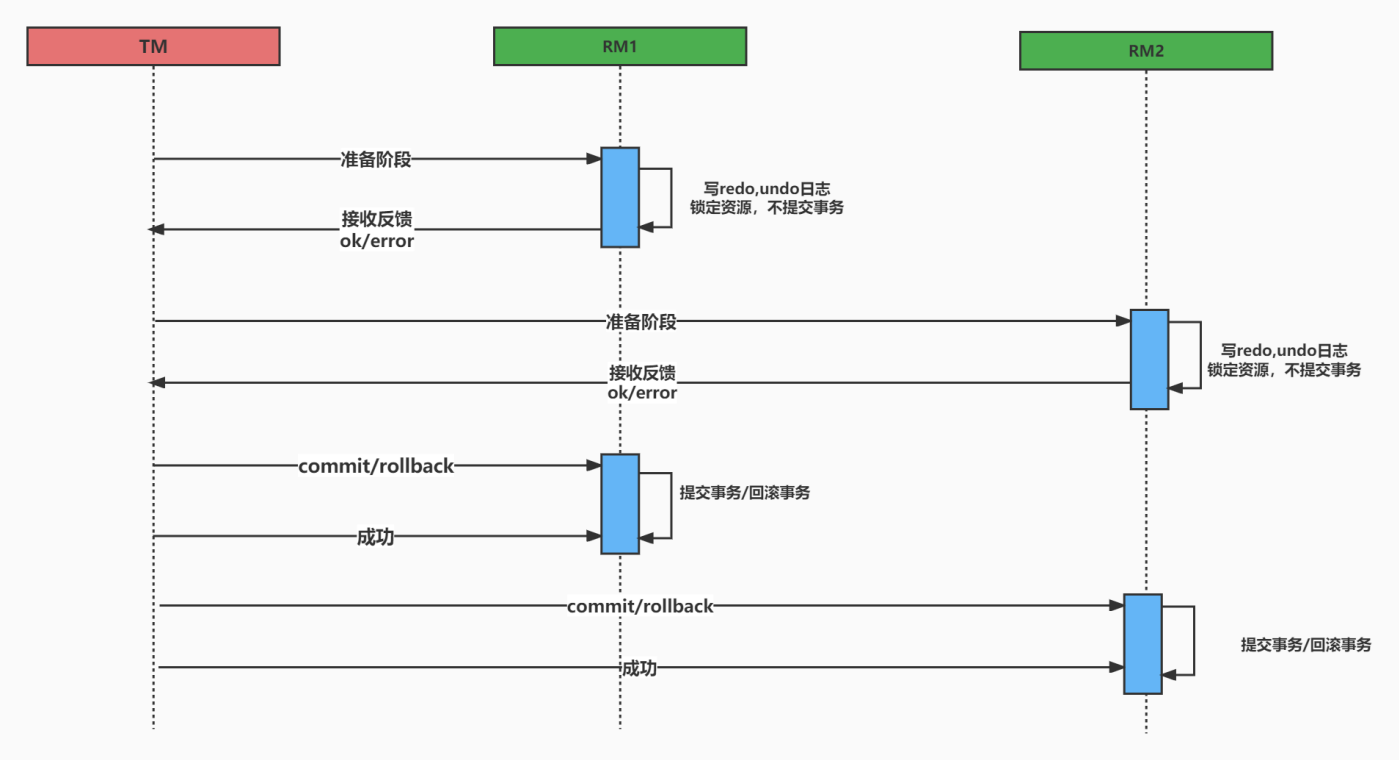
这四种模式2PC（两阶段提交）

- **AT模式**:提供无侵入自动补偿的事务模式 【这里是基于本地能支持事务的关系型数据库，然后java代码可以通过JDBC访问数据库， 这里的无侵入：我们只需要加上对应的注解就可以开启全局事务】
- **XA模式**:支持已实现XA接口的数据库的XA模式【这里一般是需要数据库实现对应的XA模式的接口，一般像 mysql oracle 都实现了XA】
- **TCC模式**:TCC则可以理解为在应用层面的 2PC，是需要我们编写业务逻辑来实现。
- **SAGA模式**:为长事务提供有效的解决方案

18、请简述2PC流程以及优缺点

1、2PC

1.1 流程



1.2 优缺点

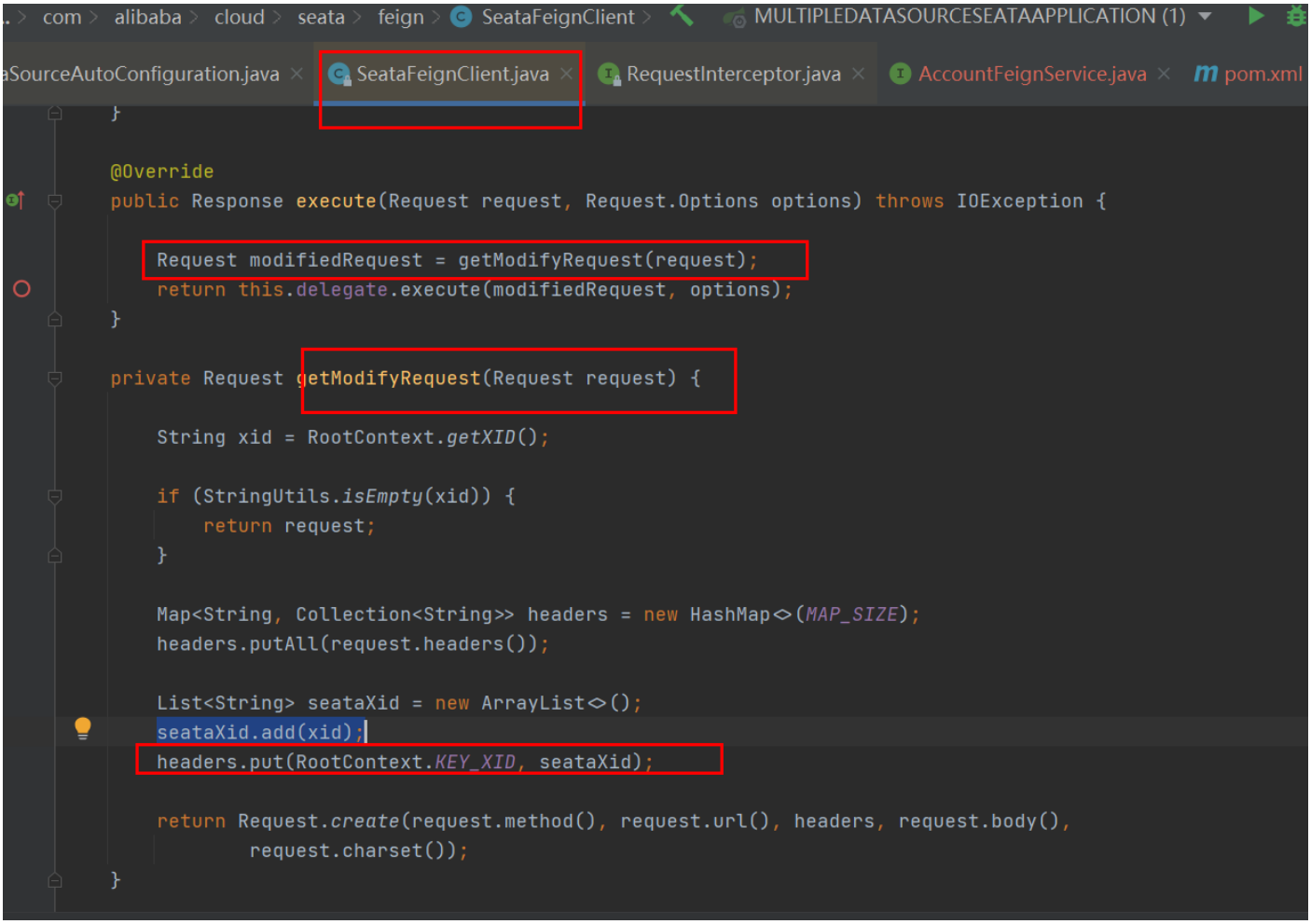
优点： 尽量保证了数据的强一致，实现成本较低，在各大主流数据库都有自己实现，对于MySQL是从5.5开始支持(XA)。

缺点：

- **单点问题**:事务管理器在整个流程中扮演的角色很关键，如果其宕机，比如在第一阶段已经完成，在第二阶段正准备提交的时候事务管理器宕机，资源管理器就会一直阻塞，导致数据库无法使用。
- **同步阻塞**:在准备就绪之后，资源管理器中的资源一直处于阻塞，直到提交完成，释放资源。
- **数据不一致**:两阶段提交协议虽然为分布式数据强一致性所设计，但仍然存在数据不一致性的可能，比如在第二阶段中，假设协调者发出了事务commit的通知，但是因为网络问题该通知仅被一部分参与者所收到并执行了commit操作，其余的参与者则因为没有收到通知一直处于阻塞状态，这时候就产生了数据的不一致性。

19、Seata中xid怎样通过Feign进行全局传递

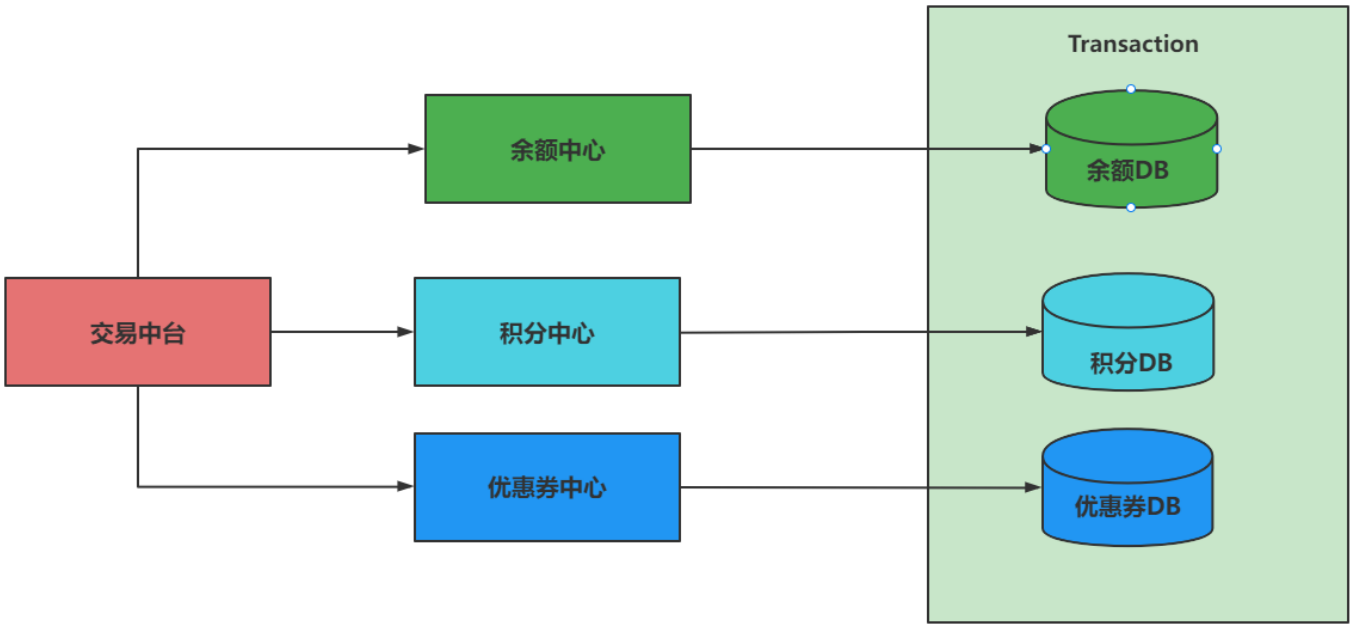
全局事务id通过Feign的header进行传递。



20、分布式事务应用的典型场景

1、多服务

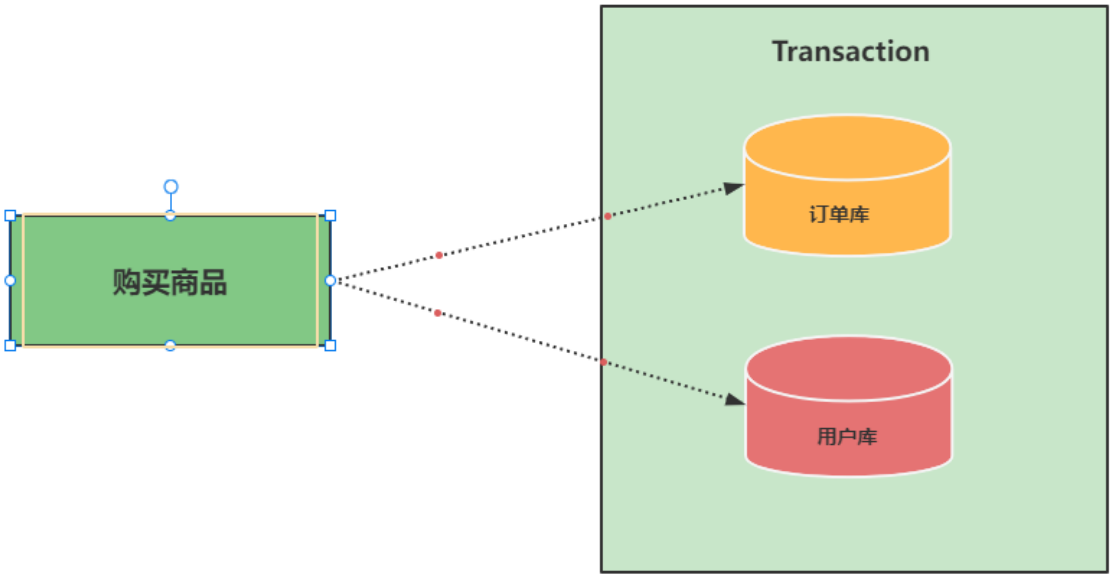
随着互联网快速发展，微服务，SOA等服务架构模式正在被大规模的使用，会出现很多分布式事务问题如下：



2、多数据源

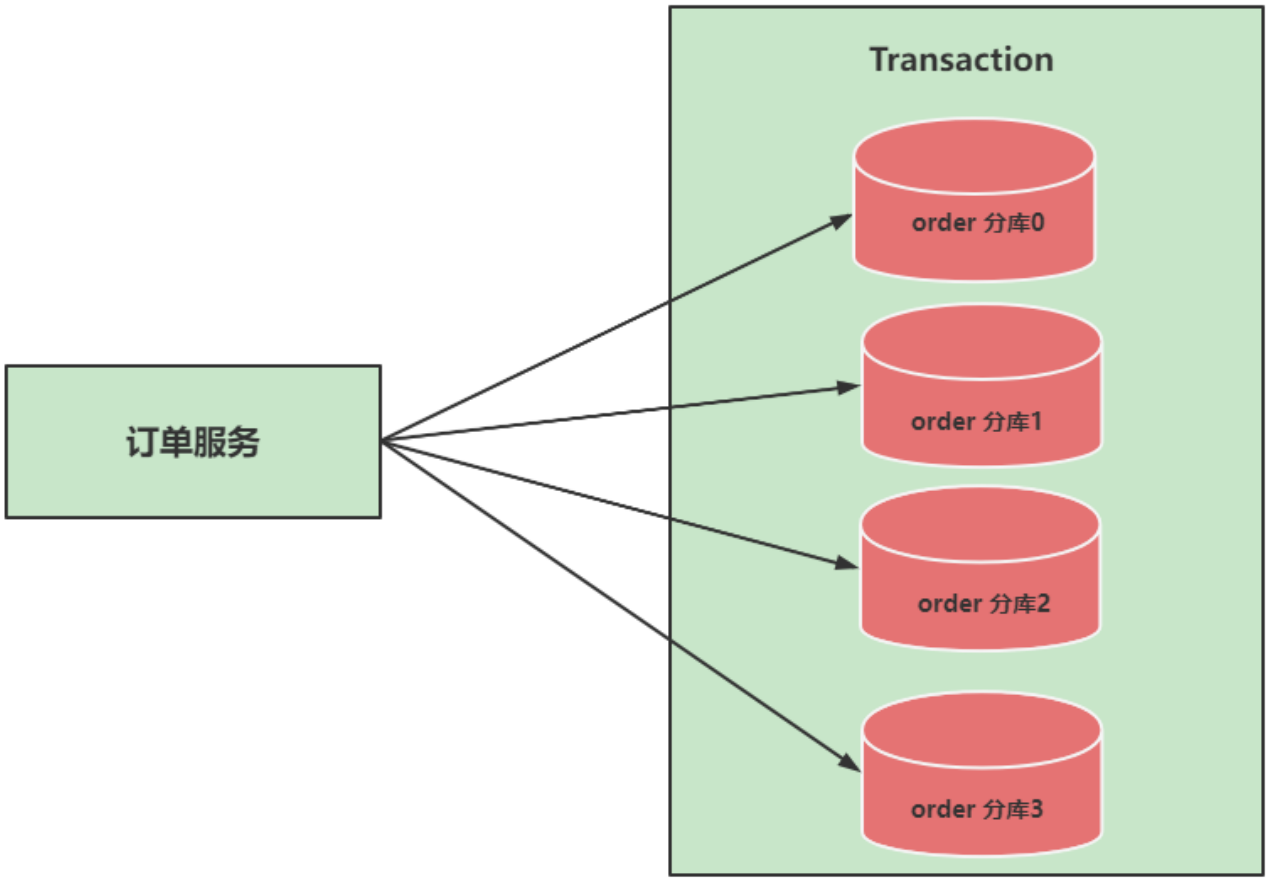
2.1 跨库

跨库事务指的是，一个应用某个功能需要操作多个库，如下图：



2.2 分库分表

通常一个库数据量比较大或者预期未来的数据量比较大，都会进行水平拆分，也就是分库分表。如下图，将订单数据库拆分成了4个库：



21、请说一下CAP和BASE理论

CAP和BASE是分布式必备理论基础

1、CAP理论

这个定理的内容是指的是在一个分布式系统中、Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。

- 一致性（C）

  一致性意思就是写操作之后进行读操作无论在哪个节点都需要返回写操作的值
- 可用性（A）

  非故障的节点在合理的时间内返回合理的响应
- 分区容错性（P）

  当出现网络分区后，系统能够继续工作。打个比方，这里个集群有多台机器，有台机器网络出现了问题，但是这个集群仍然可以正常工作。

在分布式系统中，网络无法100%可靠，分区其实是一个必然现象，如果我们选择了CA而放弃了P，那么当发生分区现象时，为了保证一致性，这个时候必须拒绝请求，但是A又不允许，所以分布式系统理论上不可能选择CA架构，只能选择CP或者AP架构。

对于CP来说，放弃可用性，追求一致性和分区容错性，我们的zookeeper其实就是追求的强一致。

对于AP来说，放弃一致性(这里说的一致性是强一致性)，追求分区容错性和可用性，Nacos就是AP模式，这是很多分布式系统设计时的选择，后面的BASE也是根据AP来扩展。

2、BASE理论

BASE是Basically Available（基本可用）、Soft state（软状态）和 Eventually consistent（最终一致性）三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是基于CAP定理逐步演化而来的。BASE理论的核心思想是：**即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。**

- 基本可用

  基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性—-注意，这绝不等价于系统不可用。比如：

  （1）**响应时间上的损失。**正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒

  （2）**系统功能上的损失**：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面
- 软状态

  软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时
- 最终一致性

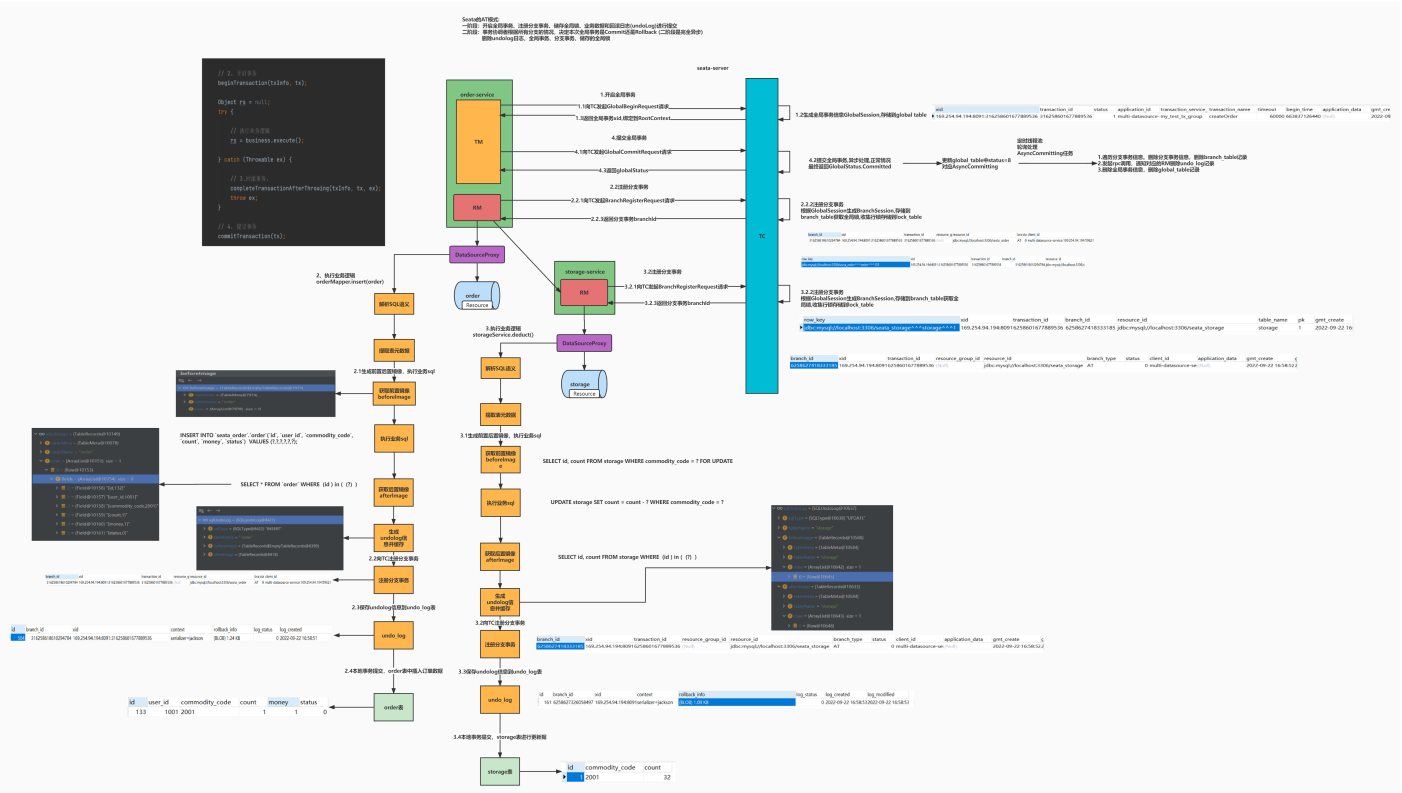
  最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

22、简述Seata的AT模式两阶段过程

Seata的AT模式

一阶段：开启全局事务、注册分支事务、储存全局锁、业务数据和回滚日志(undoLog)进行提交

二阶段：事务协调者根据所有分支的情况，决定本次全局事务是Commit还是Rollback (二阶段是完全异步删除undolog日志，全局事务、分支事务、储存的全局锁）



23、简述Eureka自我保护机制

Eureka服务端会检查最近15分钟内所有Eureka 实例正常心跳占比，如果低于85%就会触发自我保护机制。触发了保护机制，Eureka将暂时把这些失效的服务保护起来，不让其过期，但这些服务也并不是永远不会过期。Eureka在启动完成后，每隔60秒会检查一次服务健康状态，如果这些被保护起来失效的服务过一段时间后（默认90秒）还是没有恢复，就会把这些服务剔除。如果在此期间服务恢复了并且实例心跳占比高于85%时，就会自动关闭自我保护机制。

```
protected void updateRenewsPerMinThreshold() {
    this.numberOfRenewsPerMinThreshold = (int) (this.expectedNumberOfClientsSendingRenews
            * (60.0 / serverConfig.getExpectedClientRenewalIntervalSeconds())
            * serverConfig.getRenewalPercentThreshold());
}
```

阈值的计算方式：

每分钟 能接受最少的续租次数 = 微服务实例总数 * （ 60 s / 实例的续约时间间隔：30s ） * 有效的心跳比率 默认 85%

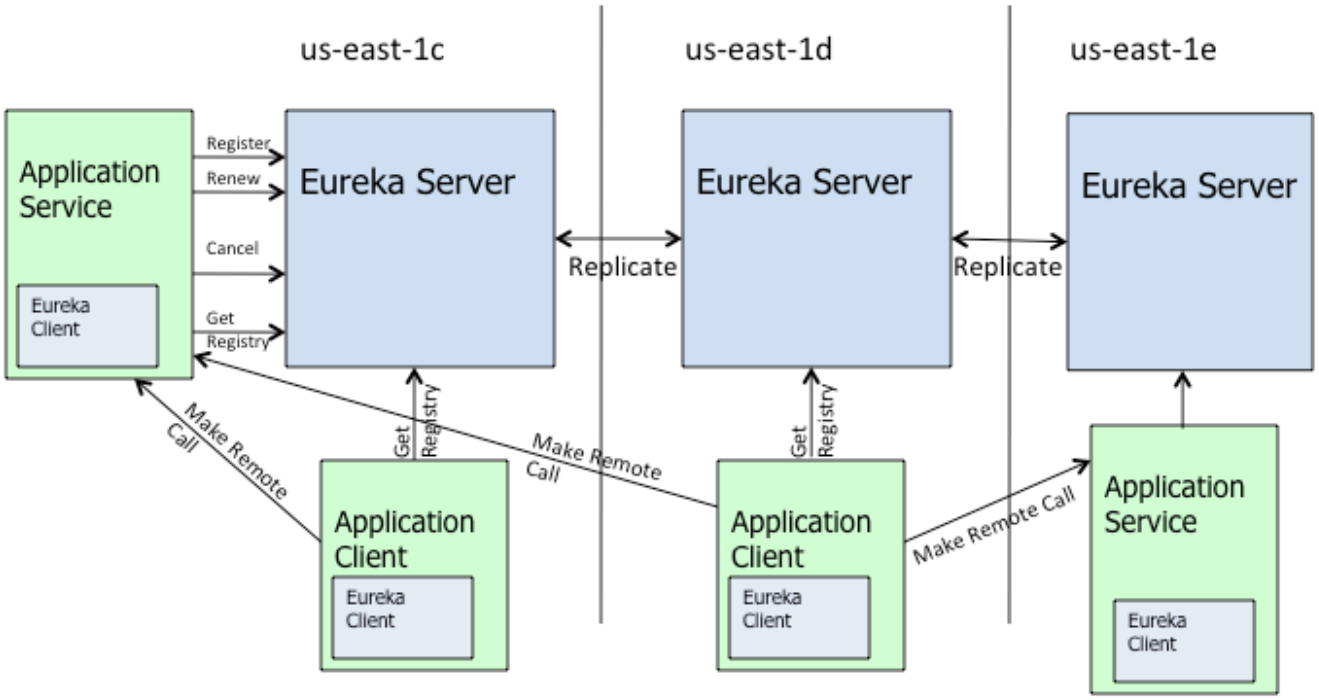
100 乘以（ 60 /30 ） 乘以 85% = 170

自我保护机制更改条件： 注册、服务下架、 服务初始化、15分钟更新一次

自我保护机制的触发位置： 服务剔除

24、简述Eureka集群架构

参考官网：https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance



- Register(服务注册)：把自己的 IP 和端口注册给 Eureka。
- Renew(服务续约)：发送心跳包，每 30 秒发送一次。告诉 Eureka 自己还活着。
- Cancel(服务下线)：当 provider 关闭时会向 Eureka 发送消息，把自己从服务列表中删除。防止 consumer 调用到不存在的服务。
- Get Registry(获取服务注册列表)：获取其他服务列表。
- Make Remote Call(远程调用)：完成服务的远程调用。
- Replicate(集群中数据同步)：eureka 集群中的数据复制与同步。

25、从Eureka迁移到Nacos的解决方案

版本依赖关系：

https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E

2.2.x 分支

适配 Spring Boot 为 2.4，Spring Cloud Hoxton 版本及以下的 Spring Cloud Alibaba 版本按从新到旧排列如下表（最新版本用*标记）：

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2.2.9.RELEASE*	Spring Cloud Hoxton.SR12	2.3.12.RELEASE
2.2.8.RELEASE	Spring Cloud Hoxton.SR12	2.3.12.RELEASE
2.2.7.RELEASE	Spring Cloud Hoxton.SR12	2.3.12.RELEASE
2.2.6.RELEASE	Spring Cloud Hoxton.SR9	2.3.2.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
2.1.4.RELEASE	Spring Cloud Greenwich.SR6	2.1.13.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

- 换依赖

```
 <spring-cloud.alibaba.version>2.2.9.RELEASE</spring-cloud.alibaba.version>
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-dependencies</artifactId>
    <version>${spring-cloud.alibaba.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

```
<dependencies>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
</dependencies>
```

- 配置换一下

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
        service: msb-order
```

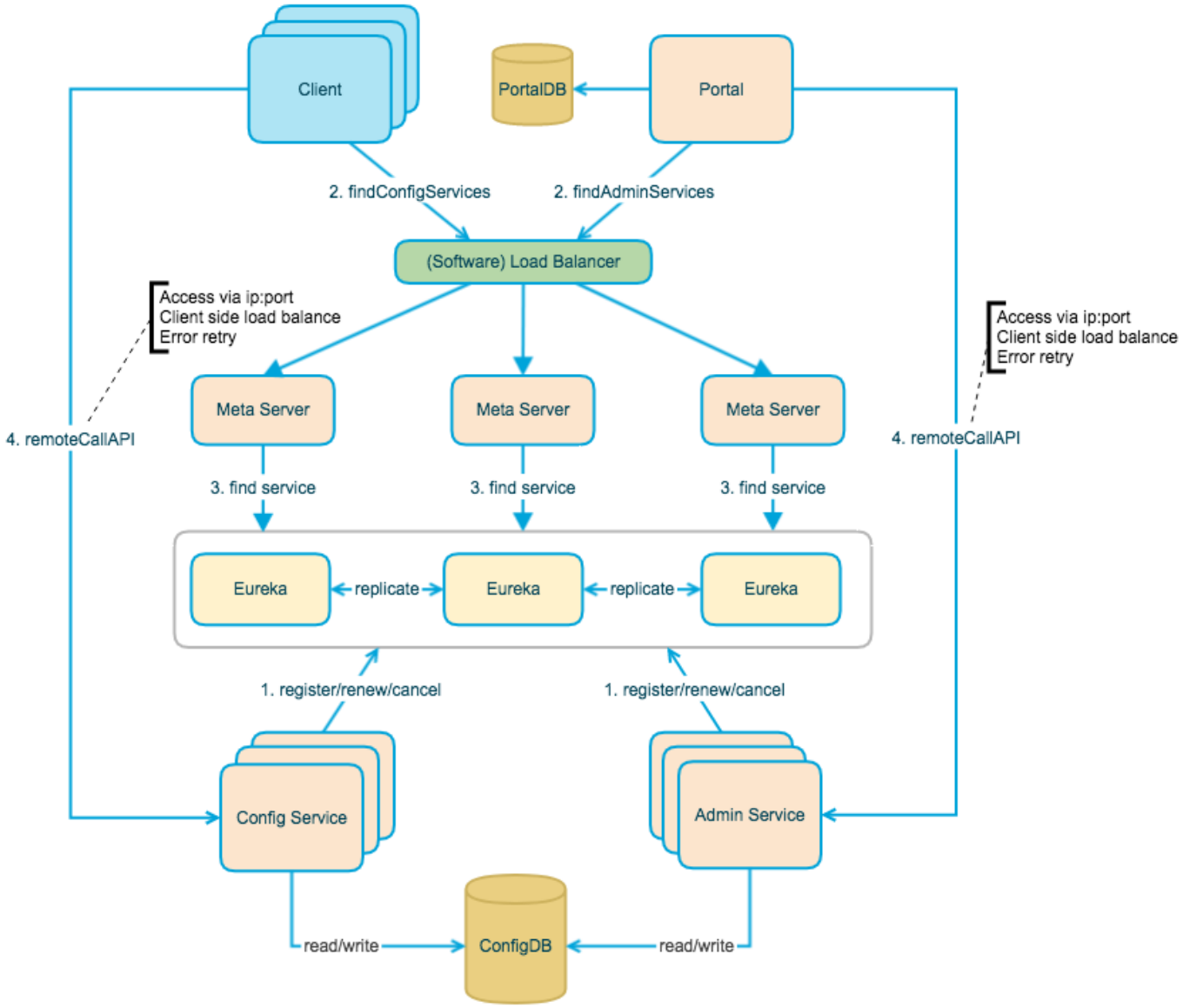
- 使用@EnableDiscoveryClient 开启注

```
@EnableDiscoveryClient // 这是官方提供的 ,我们以后可能切换其他的注册中心比如说nacos，那我们就直接切
//@EnableEurekaClient  // 是netflix提供的，如果用这个注解就只能服务于eureka
```

- 梳理项目关系

  只提供调用的基础服务先进行更改，进行迁移

26、Apollo的整体架构



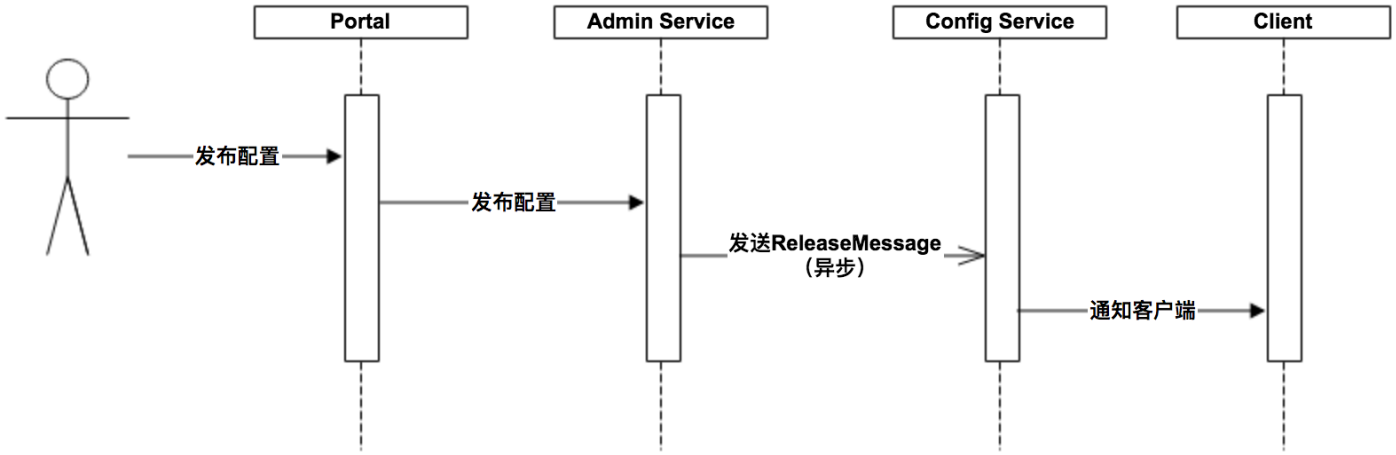
- Config Service提供配置的读取、推送等功能，服务对象是Apollo客户端。
- Admin Service提供配置的修改、发布等功能，服务对象是Apollo Portal（管理界面）。
- Config Service和Admin Service都是多实例、无状态部署，所以需要将自己注册到Eureka中并保持心跳。
- 在Eureka之上我们架了一层Meta Server用于封装Eureka的服务发现接口。
- Client通过域名访问Meta Server获取Config Service服务列表（IP+Port），而后直接通过IP+Port访问服务，同时在Client侧会做load balance、错误重试。
- Portal通过域名访问Meta Server获取Admin Service服务列表（IP+Port），而后直接通过IP+Port访问服务，同时在Portal侧会做load balance、错误重试。
- 为了简化部署，我们实际上会把Config Service、Eureka和Meta Server三个逻辑角色部署在同一个JVM进程中。

27、Apollo的整体架构可靠性分析

场景	影响	降级
某台Config Service下线	无影响	
所有Config Service下线	客户端无法读取最新配置，Portal无影响	客户端重启时，可以读取本
某台Admin Service下线	无影响	
所有Admin Service下线	客户端无影响，Portal无法更新配置	
某台Portal下线	无影响	
全部Portal下线	客户端无影响，Portal无法更新配置	
某个数据中心下线	无影响	
数据库宕机	客户端无影响，Portal无法更新配置	Config Service开启配置缓

28、Apollo配置发布后的实时推送设计

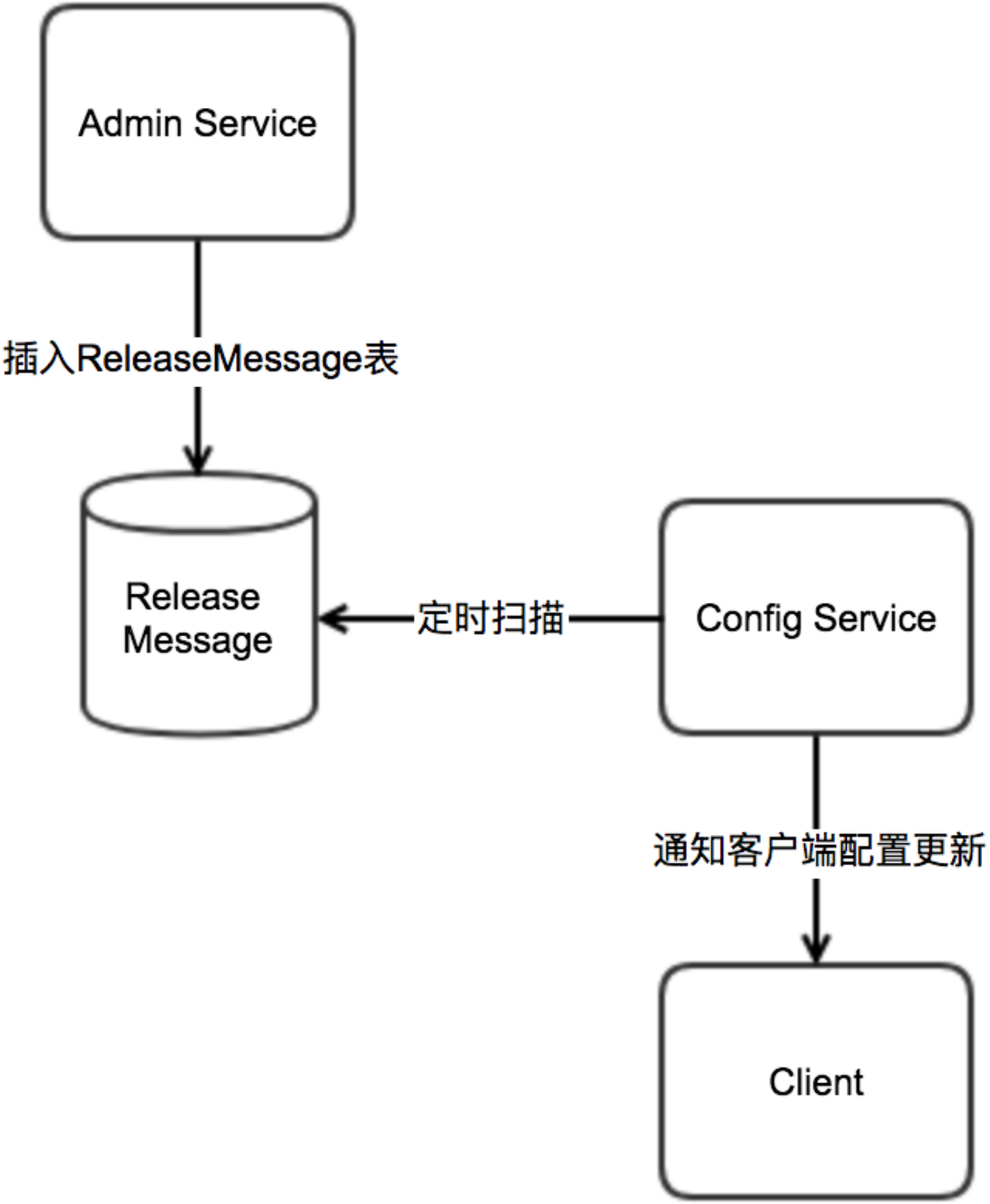
在配置中心中，一个重要的功能就是配置发布后实时推送到客户端。下面我们简要看一下这块是怎么设计实现的。



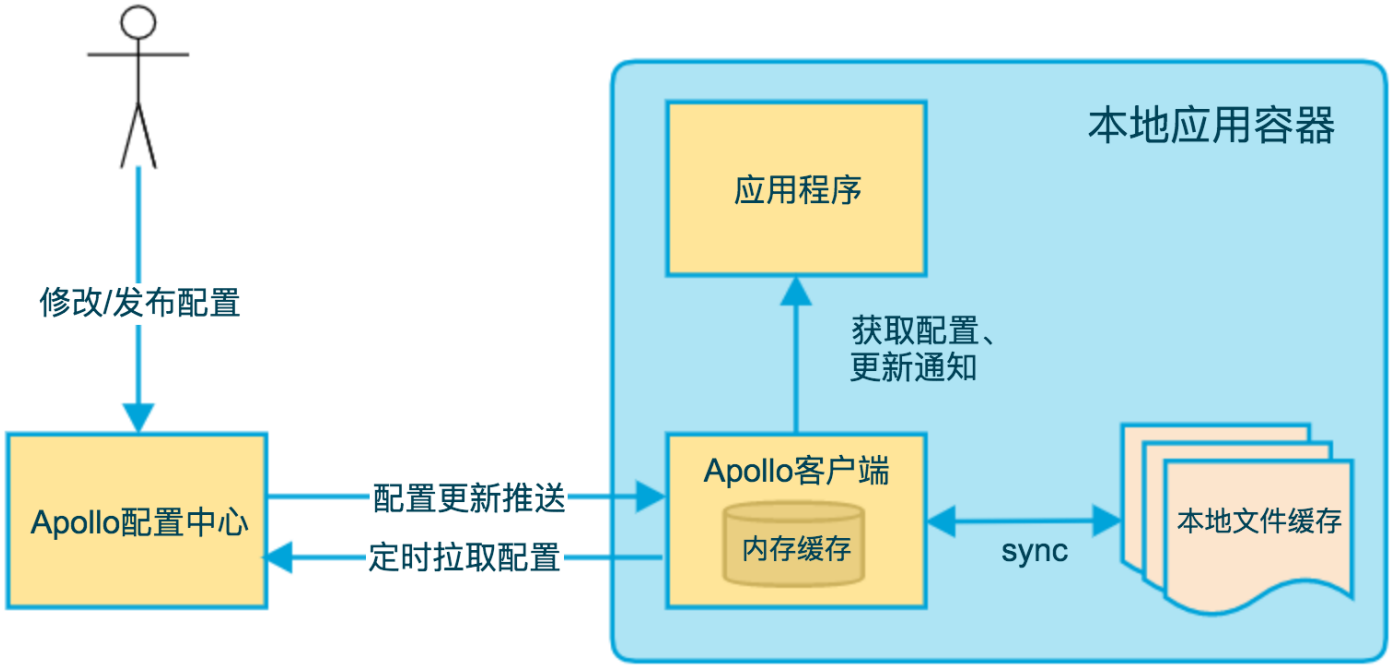
上图简要描述了配置发布的大致过程：

1. 用户在Portal操作配置发布

2. Portal调用Admin Service的接口操作发布
3. Admin Service发布配置后，发送ReleaseMessage给各个Config Service
4. Config Service收到ReleaseMessage后，通知对应的客户端



29、Apollo客户端设计



上图简要描述了Apollo客户端的实现原理：

1. 客户端和服务端保持了一个长连接，从而能第一时间获得配置更新的推送。（通过Http Long Polling实现）
2. 客户端还会定时从Apollo配置中心服务端拉取应用的最新配置。
3. 客户端从Apollo配置中心服务端获取到应用的最新配置后，会保存在内存中
4. 客户端会把从服务端获取到的配置在本地文件系统缓存一份
   - 在遇到服务不可用，或网络不通的时候，依然能从本地恢复配置
5. 应用程序可以从Apollo客户端获取最新的配置、订阅配置更新通知

30、Zuul有几种过滤器类型?分别是?

zuul 有四种过滤器类型，分别是：

1、Pre：过滤器在请求被路由之前调用。我们可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等；

2、Routing：过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用Apache HttpClient或Netfilx feign请求微服；

3、Post：过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端；

4、Error：在其他阶段发生错误时执行该过滤器。除了默认的过滤器类型微服务；