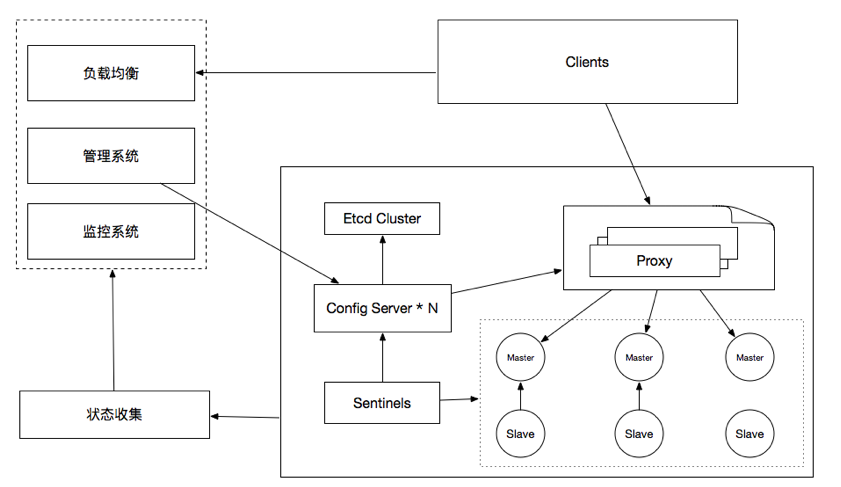
**Redis集群方案Alchemy设计**

**设计方案**

**整体架构**



**基本设计**

**基本的组件**

* redis-server
* config-server/master 依赖etcd
* proxy

**元信息的管理**

元信息存放在etcd中

slot信息感觉还是需要下方至redis-server， redis-server知道slot的信息

使用Prepare slots

proxy的元信息不同步问题，redis-server缓存一个迁移结果，基于推测，很少有在短时间内迁移多次。如果在迁移过程中proxy 发生获取元信息变更不准确，能够从redis那里感知到槽位置，类似redis cluster的模式

config-server 作为与etcd强通信的中心，本身也是无状态的，附带作为管理的apiserver，每次变更需要拿到分布式锁才能进行操作，持久化到etcd成功后才能发起操作，整体的plan算作一个batch，需要完全成功，失败后能够diff 重做

**Config-Server**

通信对象: etcd, proxy, redis-sentinel, redis-server，管理服务

etcd: 持久化存储，调度/Reblanace协调

proxy: 用来探活，配置读写分离，同步slots信息

redis-server: 下发slot信息

管理服务: 触发管理操作

常见的变更:

* 添加group: 即添加redis server
* rebalance

**Proxy**

proxy的通信对象 config-server， redis-server，client

从client接受请求转发给redis-server， 按照crc32取模/支持hashtag,

每个proxy connect watch一个config-server

proxy需要定期向config-server定期上报自己的状况 还有一个问题需不需要向config-server check自己的slot状况，这个在无状态的config-server中关系到etcd的数据结构设计 && 数据通信量的问题

proxy 向config-server watch key， 执行来自服务端的通知

读写分离的执行

**Redis Server**

需要做一些定制化处理:

* 暂时可能需要打async migrate patch
* 关于slots信息的存储，迁移状态机

**高可用方案**

* 依赖于Redis Sentinels机制
* Config-Server订阅Redis Sentinels/or 直接把sentinels在config-server中实现， 这个有etcd锁，谁抢到锁谁来订阅(后期可以考虑优化) 单config-server订阅，每次更新到etcd后才对proxy进行通知

**关于多机房的方案**

**牺牲一致性(主要是读)，读写分离**

* 写主(主机房的master)
* 读: 主机房master > 主机房master/slave > 异地DC master > 异地DC slave
* 机房之间依靠redis信息同步
* 两个机房之间的同步还是取决于redis-server: 官方的redis-server的话中间需要挂上syncer的角色或者使用psync2来做(时间过长还是有问题)，非官方的使用带有binlog的版本的比如阿里的可以支持秒级回复任意回放

**数据扩容和迁移方案**

使用codis 3.2之后的异步方案 Codis 异步迁移方案

异步迁移过程中是可以访问数据的，主要是一个slot可能迁移了一半，此处解决冲突

* 读-迁冲突
* Key Missing
* 写-迁冲突
* Key Missing + 读-迁冲突

**开发相关**

**Basic**

* C/CPP 开发proxy部分， 减少语言层面的开销，比如GC，内存，网络，runtime等
* config server 类似codis dashboard的角色， 此处可以用golang开发
* 基于predixy等方案做二次开发

**Smart Client相关的**

* evcache那套

**Etcd信息存储**

|  |
| --- |
| Shell #{service\_name}/redis/ #{service\_name}/proxy/ #{service\_name}/slot/ #{service\_name}/version |

**Config Server HA 方案**

为了减少每个config-server的压力，此处采取和Kubernetes apiserver类似的设计，通信基本靠Etcd的RPC和Lock来完成

一个机房可以有一组CnfigServer，每个机房有一组。

对于不同的Redis服务采取锚定的方式。每个ConfigServer去获取Service的Lock，判定是Service的Master机房的ConfigServer优先拿到Lock，因为对于Redis的Service本身而言，Master要比Slave服务重要，ConfigServer和Redis Master处于同一个机房能够更好的从网络上减少误判并提供更好的可用性。

在ConfigServer的RPC中有一个RedirectEndpoint的字段，这个是作为Alchemy Proxy或者平台访问的ConfigServer访问了不是锚定的ConfigServer，ConfigServer返回真正的对这个Service负责的ConfigServer的Endpoint.

此处我是更加倾向于增加一层VIP来做屏障，考虑公司暂时没有多少使用VIP的，使用Redirect来实现。这个就需要保证AlchemyProxy在启动的时候注入的ConfigServer Endpoints中至少要有一个可用。后面可以考虑ConfigServer的Meta中带有所有可用的Endpoints列表，类似Raft的Membership。

**多机房**

* Etcd按照三机房方案部署
* ConfigServer每个机房一组
* Alchemy Proxy每个机房使用不同的Write/Read Pool来针对不同的机房提供不同的读写服务
* RedisServer的部署，由于每个机房都会有多个Redis实例，需要考虑同机房和机房之间的问题，为了减少机房之间的通信和主的同步开销，对于备份机房采取伪主(发生对面机房down的时候，他优先切换成主，机房内部另外一个server也是从它这里同步)。

**Failover**

* Etcd，依赖已有的Etcd的Failover方案
* ConfigServer本身无状态，只要能够监控ConfigServer dead后，随意拉起一个即可。其他的ConfigServer会去竞争这个ConfigServer原先Serve的Service的Lock。
* Alchemy Proxy本身无状态，由Agent定期汇报是否Alive，Dead后拉起即可。
* RedisServer的Failover，这个见下

**Redis Server Failover**

处于正常状态，没有slot迁移

* Master down，这里就只需要解决主从切换就可以，使用Redis本身的策略，优先选择原先主机房的成为master，保证主基本能够在同一个机房内。
* Slave down， 随意拉起，由Redis本身的同步策略来保证

处于slot迁移状态

slot迁移本身需要持久化到文件，并且通知slave，slave持久化到文件

* Master down， 这个时候发生主从切换(同正常状态的主从切换)，从需要能够主动的去迁移slot
* Slave down， 随意拉起，从Master那里同步