缓存架构设计

毛剑

Agenda

- 选型和演进
- 缓存模式
- 缓存技巧
- 未来思考

选型和演进

早期使用twemproxy作为缓存代理，但是在使用上有如下一些痛点：

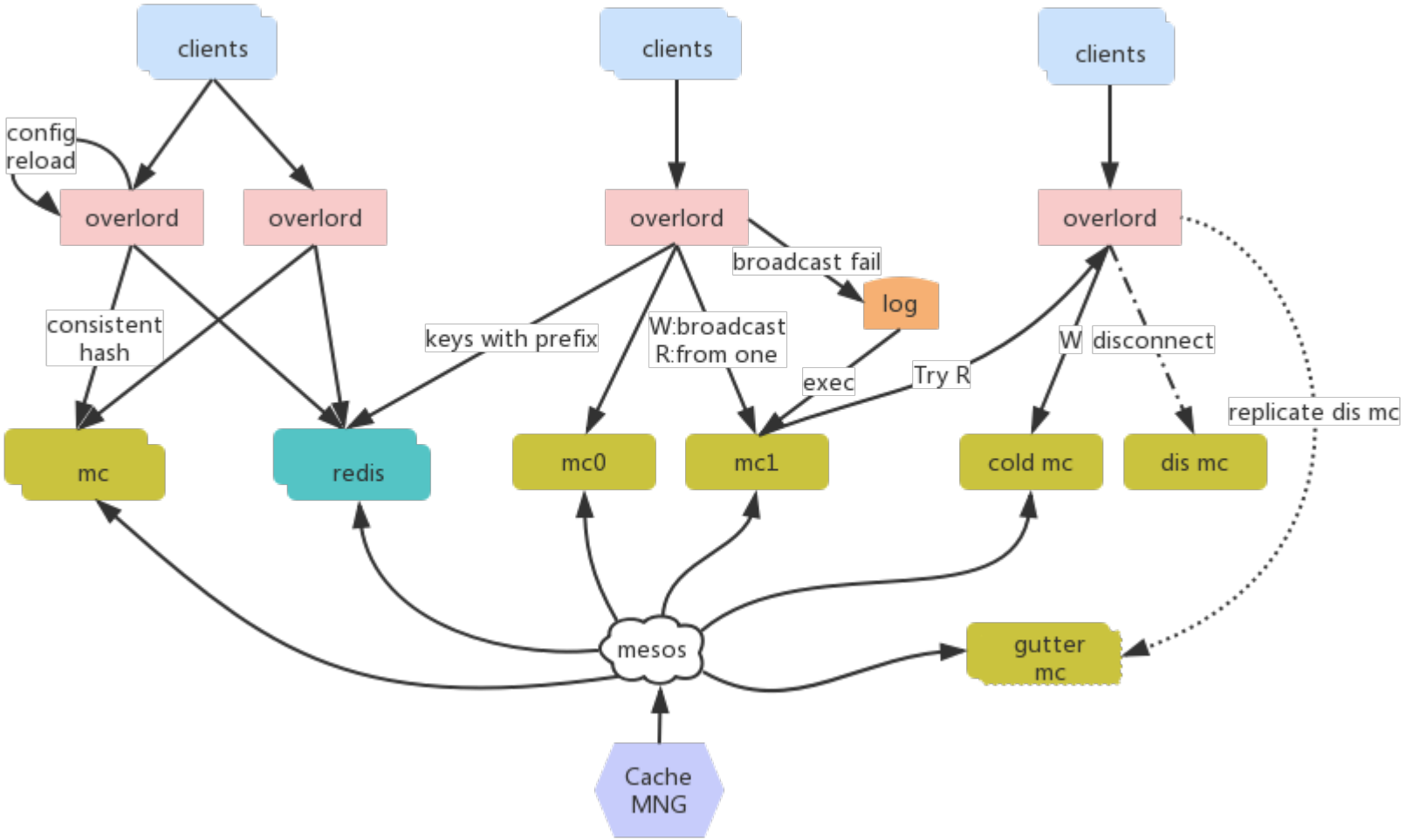
- 单进程单线程模型和redis类似，在处理一些大key的时候可能出现io瓶颈；
- 二次开发成本难度高，难以于公司运维平台进行深度集成；
- 不支持自动伸缩，不支持autorebalance增删节点需要重启才能生效；
- 运维不友好，没有控制面板；

业界开源的的其他代理工具：

codis ： 只支持redis协议，且需要使用patch版本的redis

mcrouter： 只支持memcache协议，C开发，与运维集成开发难度高

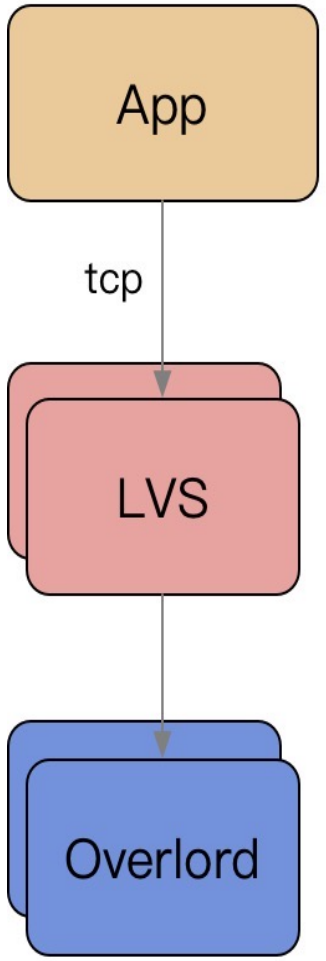
选型和演进

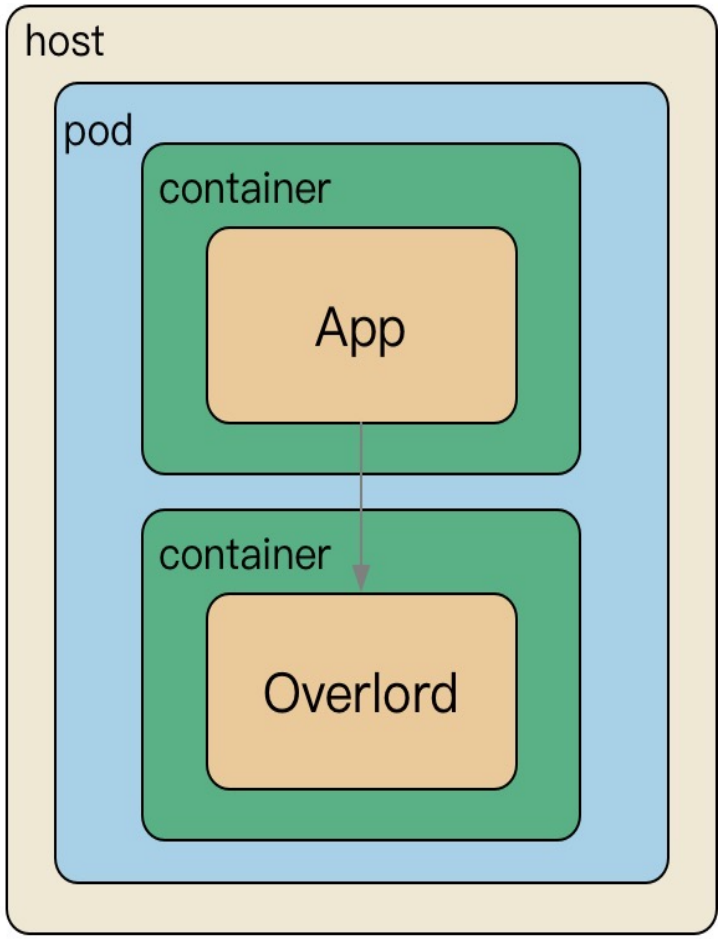

clients
clients
clients
config
reload
overlord
overlord
overlord
overlord
broadcast fail
log
consistent
hash
keys with prefix
W:broadcast
R:from one
exec
Try R
W
disconnect
replicate dis mc
mc
redis
mc0
mc1
cold mc
dis mc
mesos
gutter
mc
Cache
MNG

选型和演进

从集中式访问缓存到Sidecar访问缓存：

- 微服务强调去中心化；
- LVS运维困难，容易流量热点，随下游扩容而扩容，连接不均衡等问题；
- Sidecar伴生容器随App容器启动而启动，配置简化；





Agenda

- 选型和演进
- 缓存模式
- 缓存技巧
- 未来思考

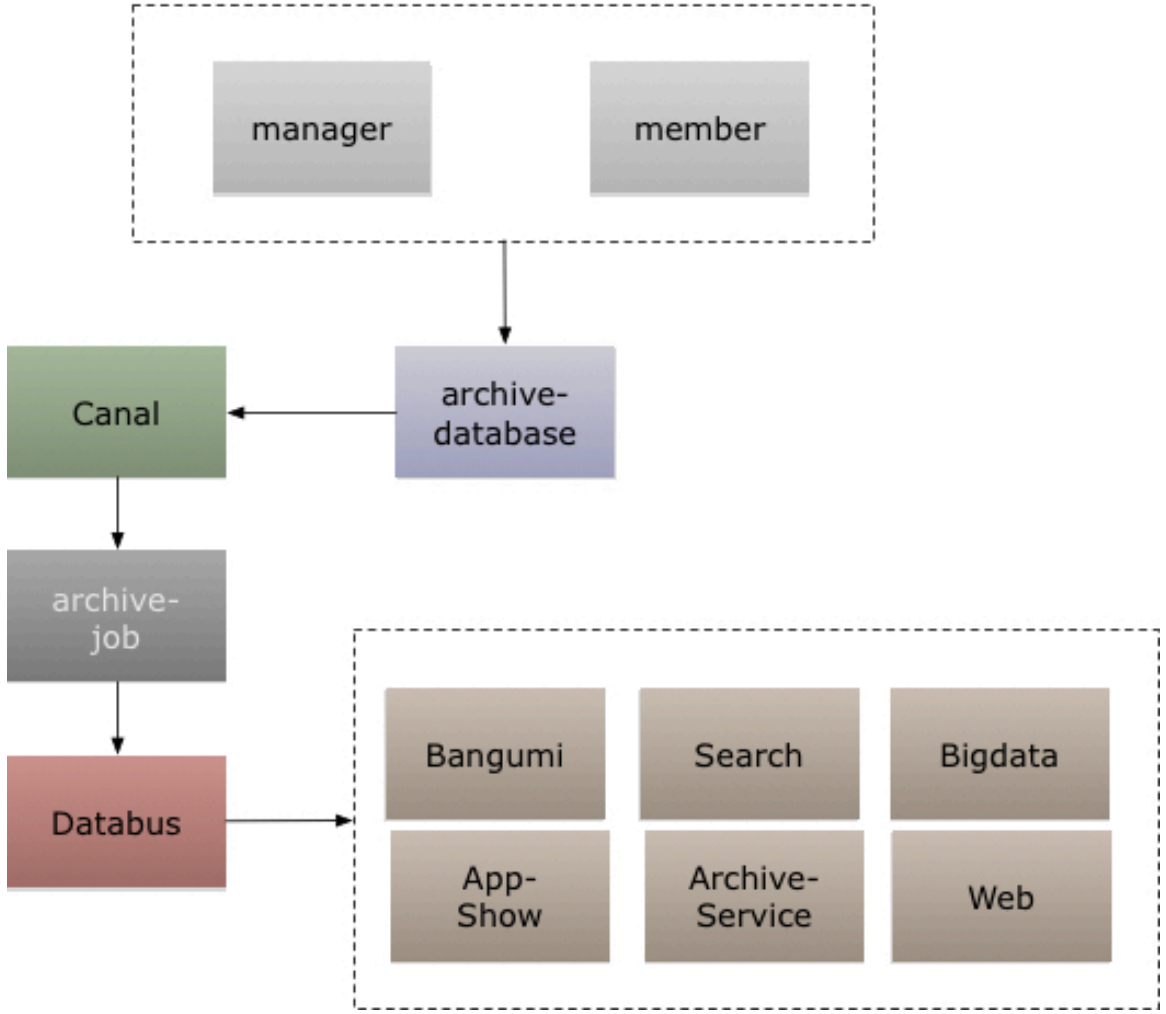
缓存模式：一致性

DB和Cache同步更新容易出现数据不一致。

模拟MySQL Slave做数据复制，再把消息投递到Kafka，保证至少一次消费：

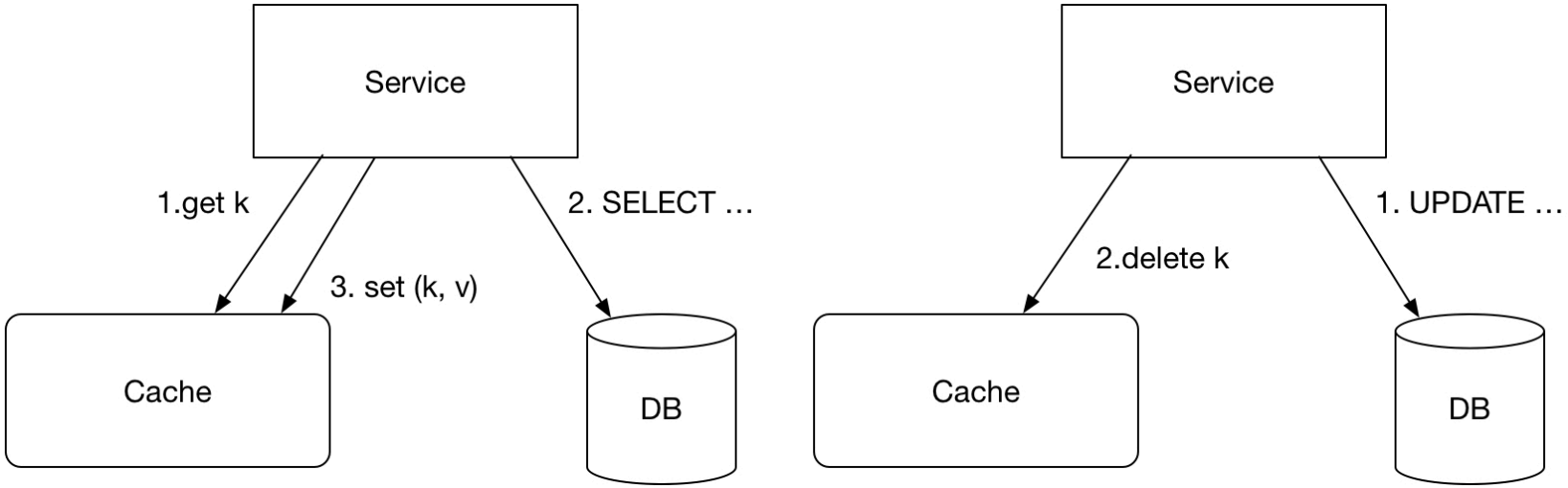
1. 同步操作DB；
2. 同步操作Cache；
3. 利用Job消费消息，重新补偿一次缓存操作

保证时效性和一致性。



缓存模式：一致性

Cache Aside模型中，读缓存Miss的回填操作，和修改数据同步更新缓存，包括消息队列的异步补偿缓存，都无法满足“Happens Before”，会存在相互覆盖的情况。



缓存模式：一致性

读操作：读缓存，缓存MISS

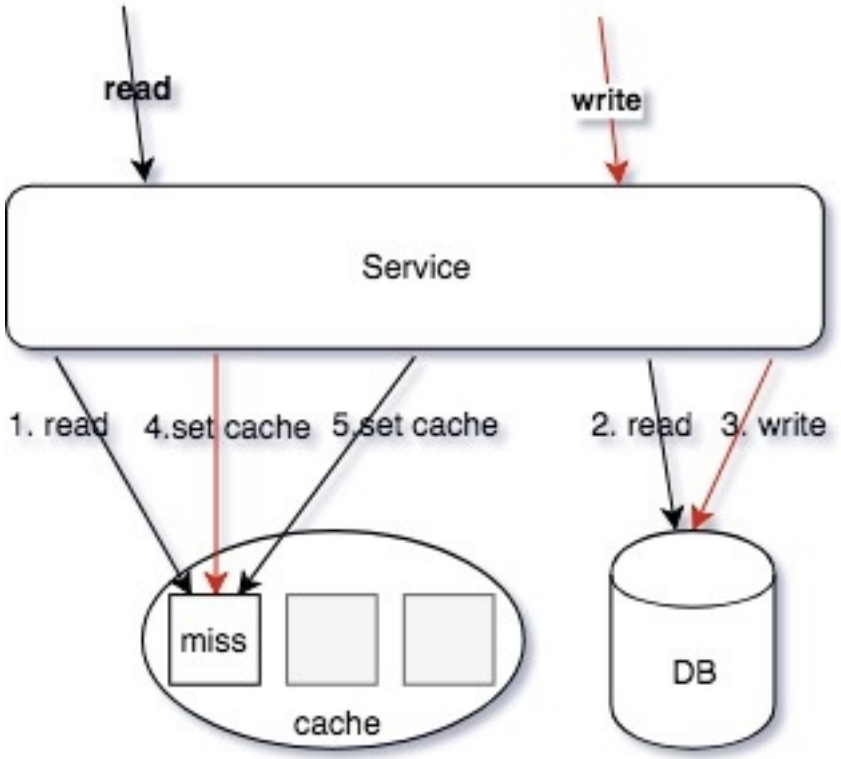
读操作：读DB，读取到数据

写操作：更新DB数据

写操作：SET Cache（可异步job操作，Redis可以使用SETEX操作）

读操作：ADD操作数据回写缓存

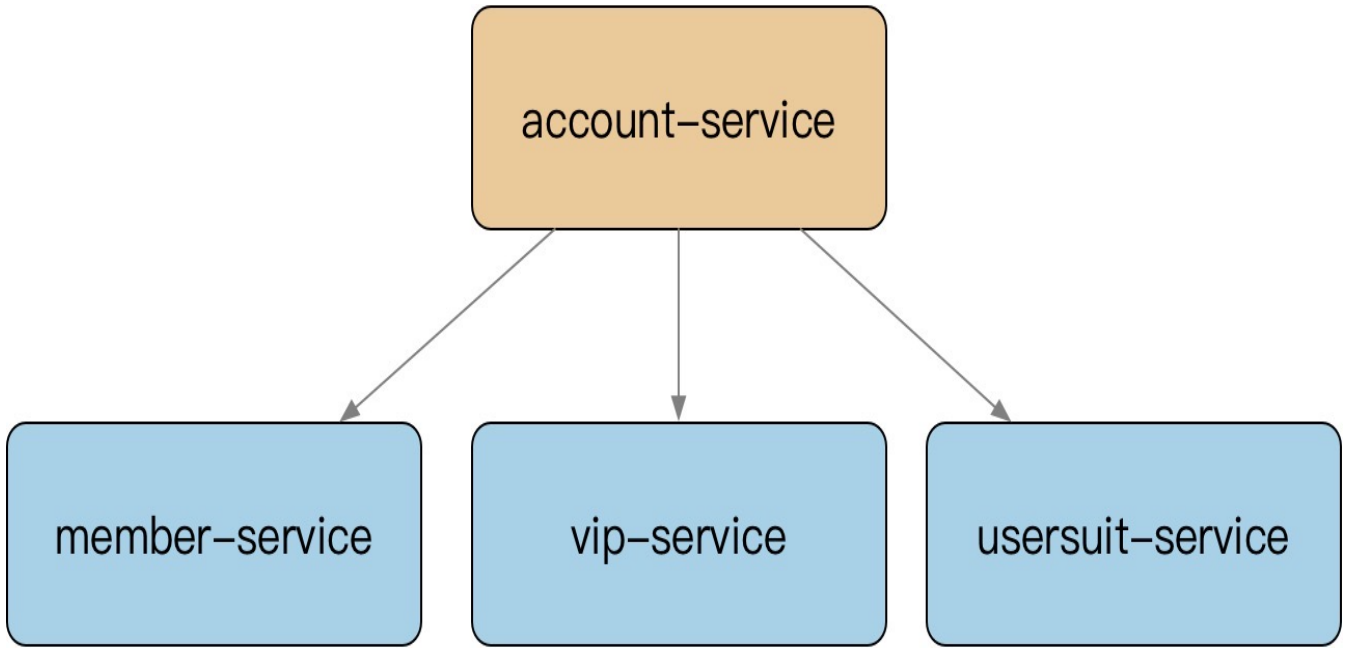
写操作：使用SET操作命令，覆盖写缓存；读操作，使用ADD操作回写MISS数据，从而保证写操作的最新数据不会被读操作的回写数据覆盖。



缓存模式：多级缓存

微服务拆分细粒度原子业务下的整合服务（聚合服务），用于提供粗粒度的接口，以及二级缓存加速，减少扇出的rpc网络请求，减少延迟。最重要是保证多级缓存的一致性：

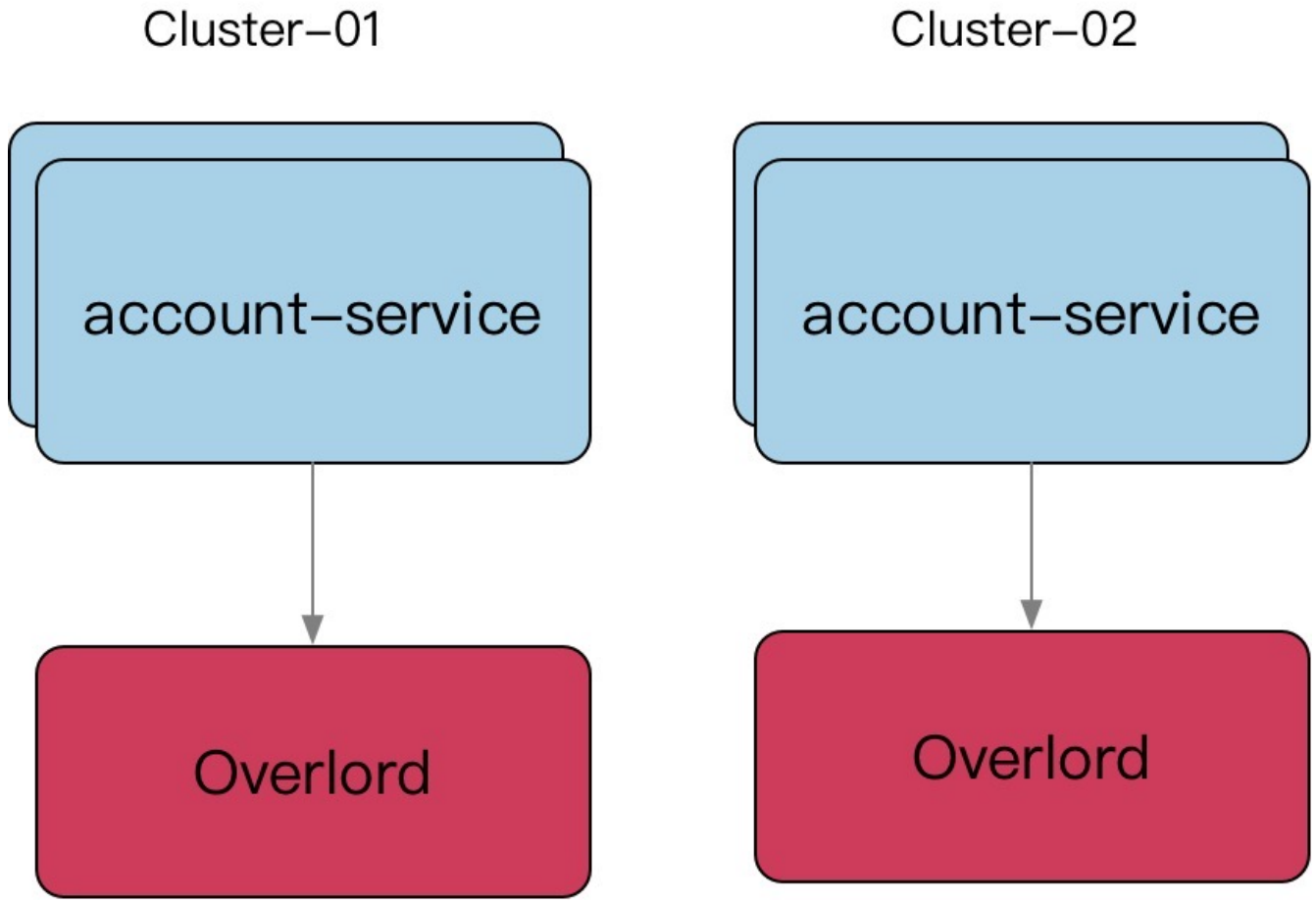
- 清理的优先级是有要求的，先优先清理下游再上游；
- 下游的缓存expire要大于上游，里面穿透回源；



缓存模式：热点问题

对于热点缓存Key，按照如下思路解决：

- 小表广播，从RemoteCache提升为LocalCache，App定时更新，甚至可以让运营平台支持广播刷新LocalCache；
- 主动监控防御预热，比如直播房间页高在线情况下直接外挂服务主动防御；
- 基础库框架支持热点发现，自动短时的short-live cache；
- 多Cluster支持；

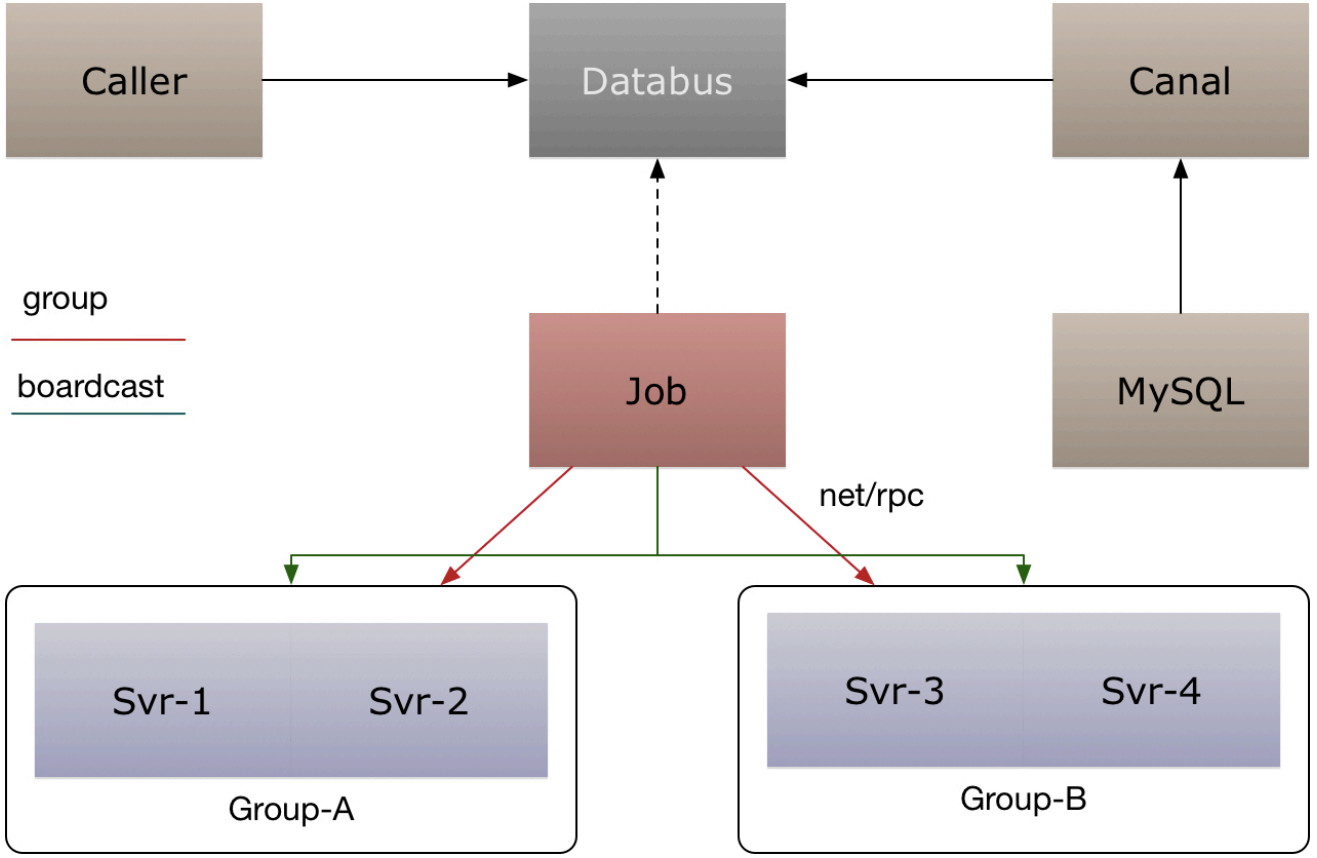


缓存模式：热点问题

建立多个Cluster ，和微服务、存储等一起组成一个Region。

这样相当于是用空间换时间：同一个key在每一个Frontend Cluster都可能有一个Copy，这样会带来consistency的问题，但是这样能够降低latency和提高availability。

- 利用MySQL Binlog 消息Anycast到不同集群的某个节点清理或者更新缓存；



Agenda

- 选型和演进
- 缓存模式
- 缓存技巧
- 未来思考

缓存技巧：Singleflight

对于热Key的回源问题，我们倾向是更新缓存而非删除缓存，回源的机制仍然需要考虑做保护，我们使用in-process 级别的归并回源，尽可能减少回源请求（少尝试复杂的分布式锁之类的策略）：

- app级别的单飞策略；
- lease机制；
- 回写大缓存异步化（通过Job）；

```go
// Do executes and returns the results of the given function, making
// sure that only one execution is in-flight for a given key at a
// time. If a duplicate comes in, the duplicate caller waits for the
// original to complete and receives the same results.
// The return value shared indicates whether v was given to multiple callers.
func (g *Group) Do(key string, fn func() (interface{}, error)) (v interface{}, err error, shared bool) {
    g.mu.Lock()
    if g.m == nil {
        g.m = make(map[string]*call)
    }
    if c, ok := g.m[key]; ok {
        c.dups++
        g.mu.Unlock()
        c.wg.Wait()
        return c.val, c.err, true
    }
    c := new(call)
    c.wg.Add(1)
    g.m[key] = c
    g.mu.Unlock()

    g.doCall(c, key, fn)
    return c.val, c.err, c.dups > 0
}
```

缓存技巧：Incast Congestion

如果在网路中的包太多，就会发生Incast Congestion的问题（可以理解为，network有很多switch，router啥的，一旦一次性发一堆包，这些包同时到达switch，这些switch就会忙不过来）。应对这个问题就是不要让大量包在同一时间发送出去，在客户端限制每次发出去的包的数量（具体实现就是客户端弄个队列）。每次发送的包的数量称为“Window size”。这个值太小的话，发送太慢，自然延迟会变高；这个值太大，发送的包太多把network switch搞崩溃了，就可能发生比如丢包之类的情况，可能被当作cache miss，这样延迟也会变高。所以这个值需要调，一般会在Proxy层面实现。

缓存技巧：小技巧

- 空缓存保护；
- 尽可能Pipeline发送请求，但是也要控制批次大小，可以并行+多批次；
- 读失败后的写缓存策略（降级后一般读失败不触发回写缓存）；
- Region Sharding策略；
- 序列化使用Protobuf，尽可能减少size；
- 工具化

```
//go:generate kratos tool genbts
type _bts interface {
        // bts: -batch=2 -max_group=20 -batch_err=break -nullcache=&Demo{ID:-1} -check_null_code=$.ID==-1
        Demos(c context.Context, keys []int64) (map[int64]*Demo, error)
        // bts: -batch=2 -max_group=20 -batch_err=continue -nullcache=&Demo{ID:-1} -check_null_code=$.ID==-1
        Demos1(c context.Context, keys []int64) (map[int64]*Demo, error)
        // bts: -sync=true -nullcache=&Demo{ID:-1} -check_null_code=$.ID==-1
        Demo(c context.Context, key int64) (*Demo, error)
        // bts: -paging=true
        Demo1(c context.Context, key int64, pn int, ps int) (*Demo, error)
        // bts: -nullcache=&Demo{ID:-1} -check_null_code=$.ID==-1
        None(c context.Context) (*Demo, error)
}
```

Agenda

- 选型和演进
- 缓存模式
- 缓存技巧
- 未来思考

未来思考

- Memcache、Redis的协议设计存在缺陷，不能单连接多路复用（参考HTTP 2.0），基于Request-Response的请求模式，需要Client测维护复杂的连接池（未来Sidecar ServiceMesh使用gRPC收敛所有中间件）；
- DAG Pipeline级别的优化，可以进一步减少网络请求导致的 Context Switch 和 Syscall；

Thanks

QA