Go 进阶训练营
第8课
分布式缓存 & 分布式事务

毛剑

目录

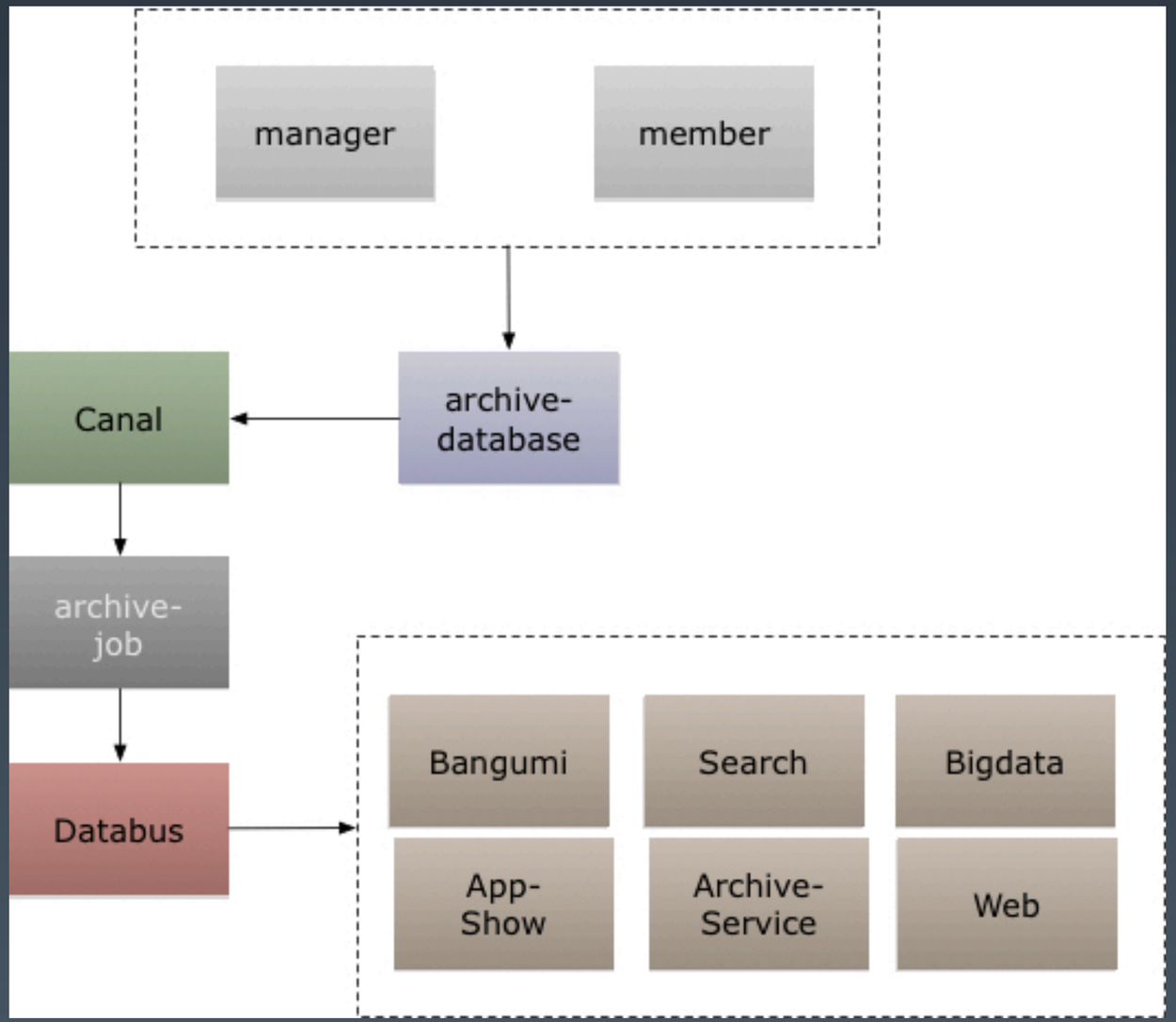
缓存模式 - 数据一致性

Storage 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：

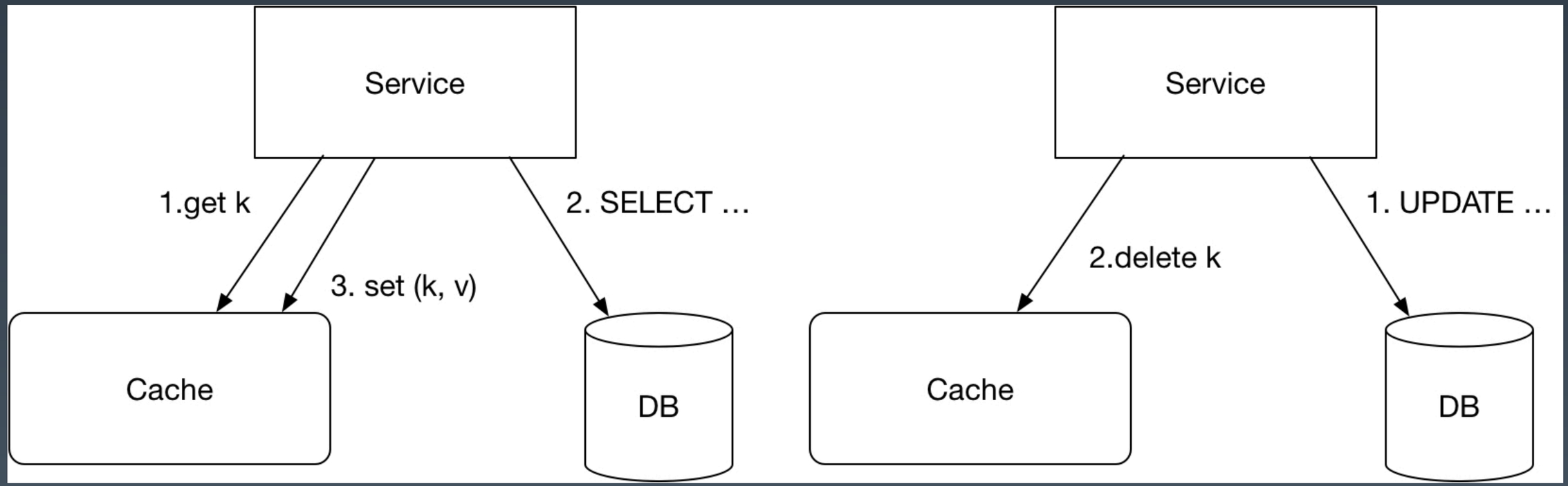
1. *同步操作 DB；*
2. *同步操作 Cache；*
3. *利用 Job 消费消息，重新补偿一次缓存操作*

保证时效性和一致性。



缓存模式 - 数据一致性

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 - 数据一致性

读/写同时操作：

读操作，读缓存，缓存 MISS

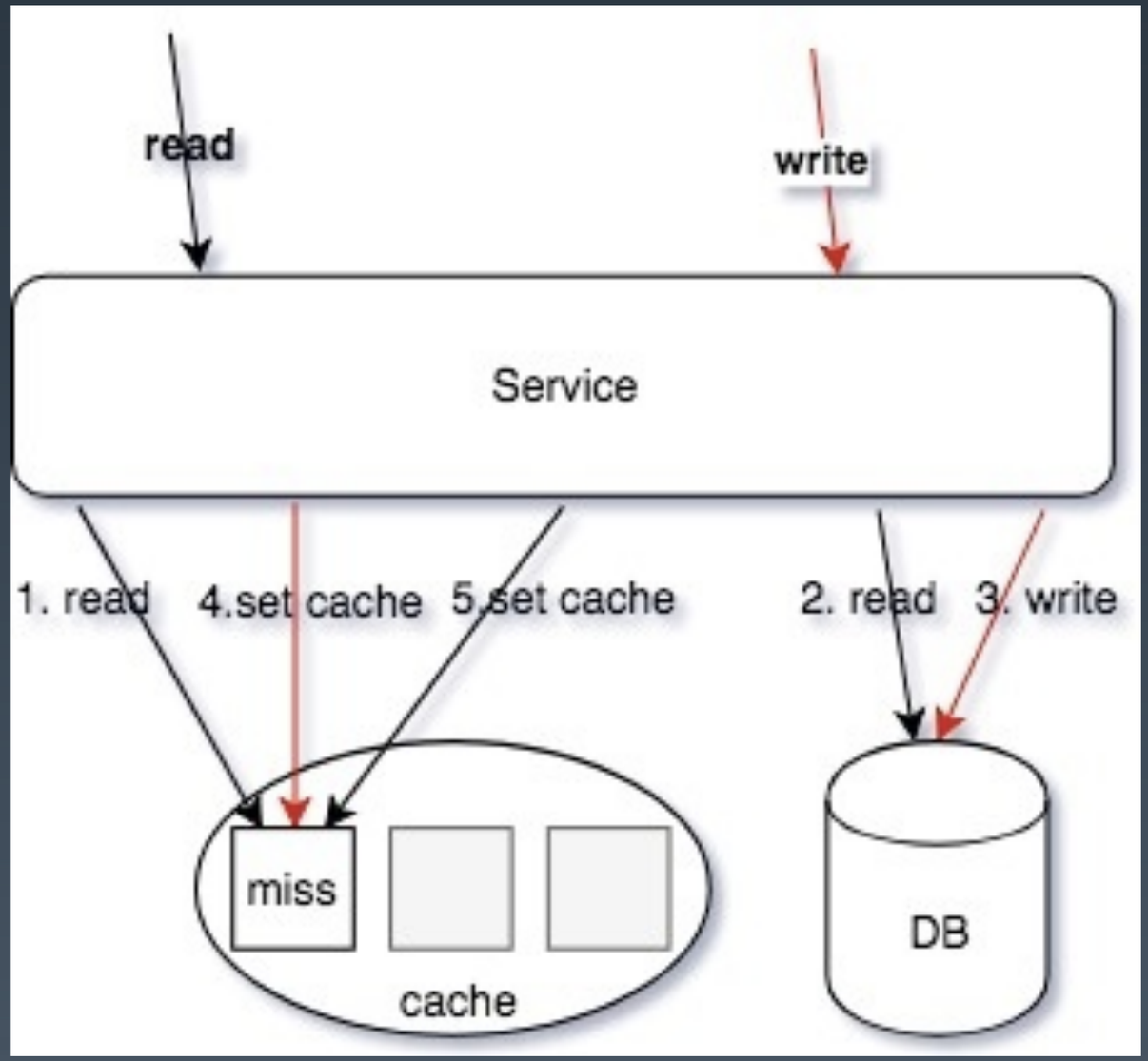
读操作，读 DB，读取到数据

写操作，更新 DB 数据

写操作 SET/DELETE Cache（可 Job 异步操作）

读操作，SET 操作数据回写缓存（可 Job 异步操作）

这种交互下，由于4和5操作步骤都是设置缓存，导致写入的值互相覆盖；并且操作的顺序性不确定，从而导致 cache 存在脏缓存的情况。



缓存模式 - 数据一致性

读/写同时操作：

读操作，读缓存，缓存 MISS

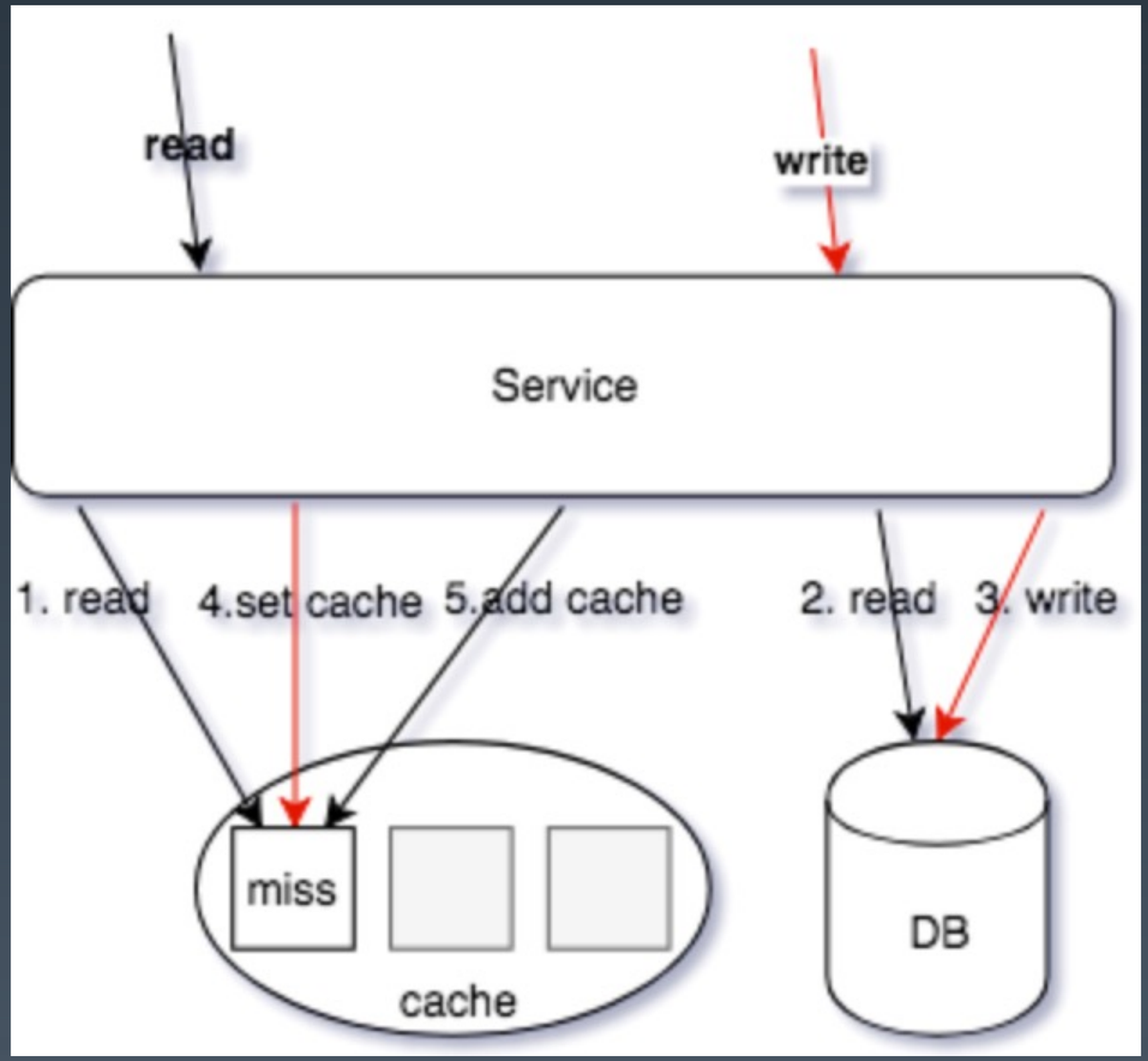
读操作，读 DB，读取到数据

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（可 Job 异步操作，Redis 可以使用 SETNX 操作）

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。



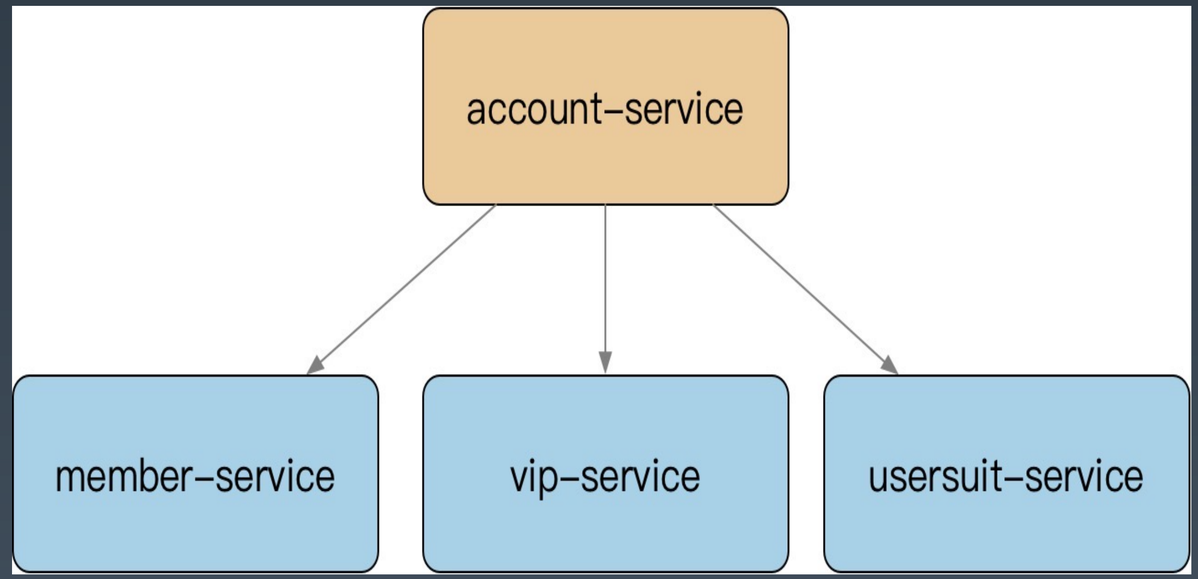
缓存模式 - 多级缓存

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；*

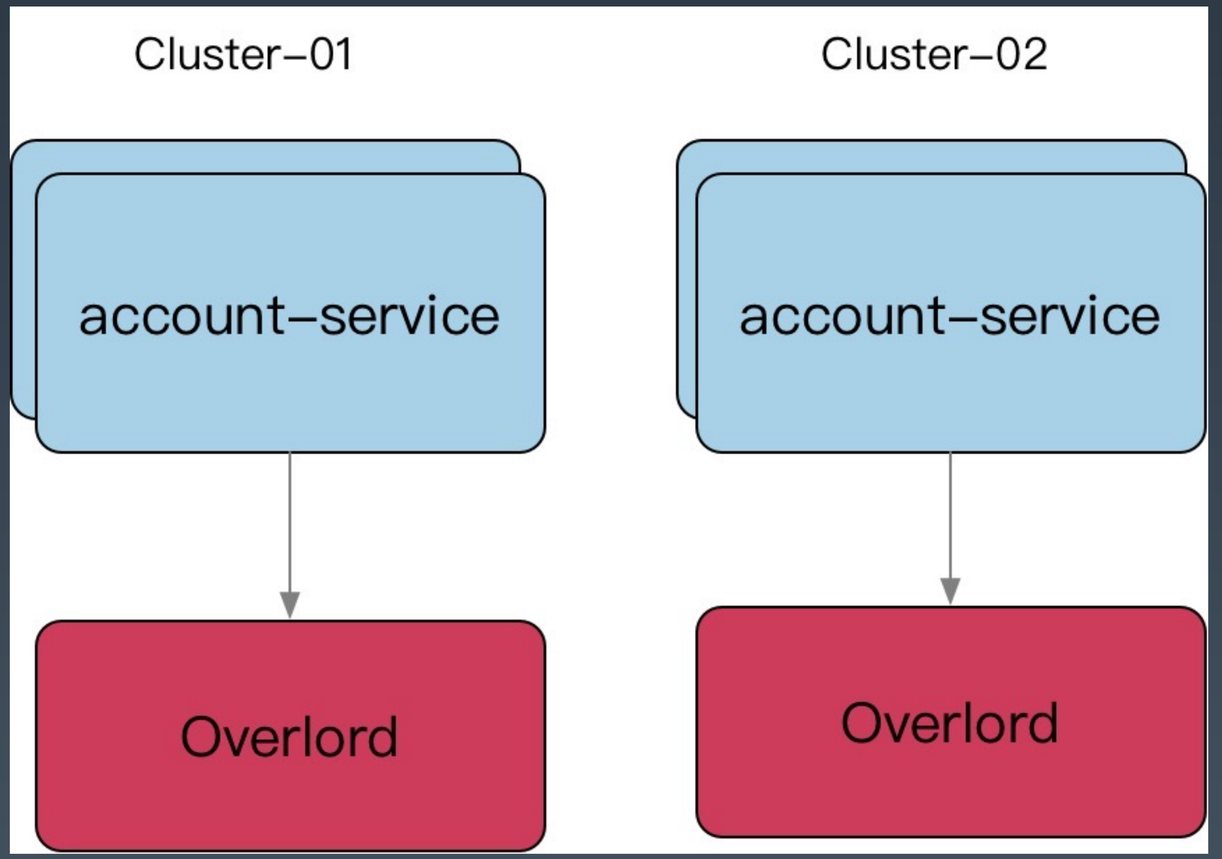
天下大势分久必合，适当的微服务合并也是不错的做法，再使用 DDD 思路以及我们介绍的目录结构组织方式，区分不同的 Usecase。



缓存模式 - 热点缓存

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；
  - 多 Key 设计: 使用多副本，减小节点热点的问题
    - 使用多副本 ms_1,ms_2,ms_3 每个节点保存一份数据，使得请求分散到多个节点，避免单点热点问题。



缓存模式 - 热点缓存

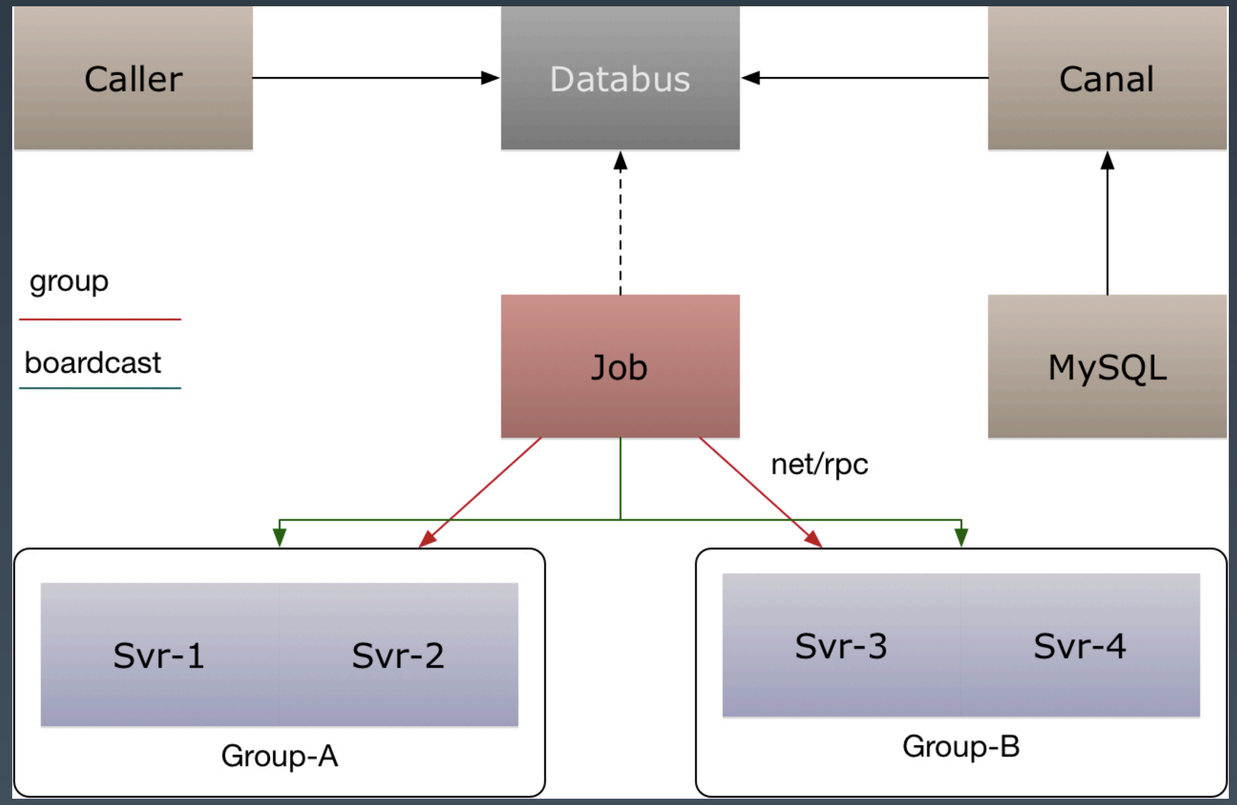
建立多个 Cluster ， 和微服务、存储等一起组成一个 Region。

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；

当业务频繁更新时候，cache 频繁过期，会导致命中率低: stale sets

如果应用程序层可以忍受稍微过期一点的数据，针对这点可以进一步降低系统负载。当一个 key 被删除的时候（delete 请求或者 cache 爆棚清空间了），它被放倒一个临时的数据结构里，会再续上比较短的一段时间。当有请求进来的时候会返回这个数据并标记为“Stale”。对于大部分应用场景而言，Stale Value 是可以忍受的。(需要改 memcache、Redis 源码，或者基础库支持）；



缓存模式 - 穿透缓存

- singlefly

对关键字进行一致性 hash，使其某一个维度的 key 一定命中某个节点，然后在节点内使用互斥锁，保证归并回源，但是对于批量查询无解；

- 分布式锁

设置一个 lock key，有且只有一个人成功，并且返回，交由这个人来执行回源操作，其他候选者轮训 cache 这个 lock key，如果不存在去读数据缓存，hit 就返回，miss 继续抢锁；

- 队列

如果 cache miss，交由队列聚合一个 key，来 load 数据回写缓存，对于 miss 当前请求可以使用 singlefly 保证回源，如评论架构实现。适合回源加载数据重的任务，比如评论 miss 只返回第一页，但是需要构建完成评论数据索引。

- lease

通过加入 lease 机制，可以很好避免这两个问题，lease 是 64-bit 的 token，与客户端请求的 key 绑定，对于过时设置，在写入时验证 lease，可以解决这个问题；对于 thundering herd，每个 key 10s 分配一次，当 client 在没有获取到 lease 时，可以稍微等一下再访问 cache，这时往往 cache 中已有数据。（基础库支持 & 修改 cache 源码）；

目录

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

- 易读性的前提下，key 设置尽可能小，减少资源的占用，redis value 可以用 int 就不要用 string，对于小于 N 的 value，redis 内部有 shared_object 缓存。
- 拆分 key。主要是用在 redis 使用 hashes 情况下。同一个 hashes key 会落到同一个 redis 节点，hashes 过大的情况下会导致内存及请求分布的不均匀。考虑对 hash 进行拆分为小的 hash，使得节点内存均匀及避免单节点请求热点。
- 空缓存设置。对于部分数据，可能数据库始终为空，这时应该设置空缓存，避免每次请求都缓存 miss 直接打到 DB。
- 空缓存保护策略。
- 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）。
- 序列化使用 protobuf，尽可能减少 size。
- 工具化浇水代码

```
//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)
}
```

缓存技巧 - memcache 小技巧

- flag 使用：标识 compress、encoding、large value 等；
- memcache 支持 gets，尽量读取，尽可能的 pipeline，减少网络往返；
- 使用二进制协议，支持 pipeline delete，UDP 读取、TCP 更新；

缓存技巧 - Redis 小技巧

- 增量更新一致性：EXPIRE、ZADD/HSET 等，保证索引结构体务必存在的情况下去操作新增数据；
- BITSET: 存储每日登陆用户，单个标记位置（boolean），为了避免单个 BITSET 过大或者热点，需要使用 region sharding，比如按照 mid求余 %和/ 10000，商为 KEY、余数作为 offset；
- List：抽奖的奖池、顶弹幕，用于类似 Stack PUSH/POP操作；
- Sortedset: 翻页、排序、有序的集合，杜绝 zrange 或者 zrevrange 返回的集合过大；
- Hashs: 过小的时候会使用压缩列表、过大的情况容易导致 rehash 内存浪费，也杜绝返回 hgetall，对于小结构体，建议直接使用 memcache KV；
- String: SET 的 EX/NX 等 KV 扩展指令，SETNX 可以用于分布式锁、SETEX 聚合了SET + EXPIRE；
- Sets: 类似 Hashs，无 Value，去重等；
- 尽可能的 PIPELINE 指令，但是避免集合过大；
- 避免超大 Value；