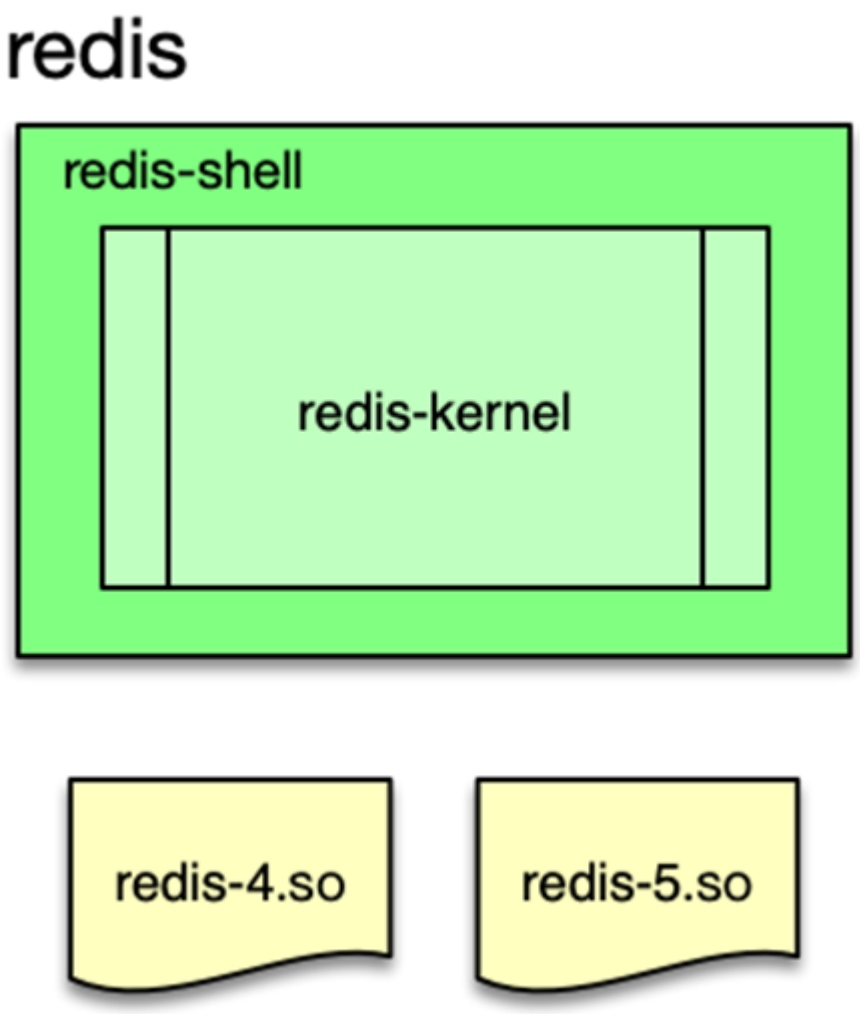
29 从容应对亿级QPS访问，Redis还缺少什么?

众所周知，Redis 在线上实际运行时，面对海量数据、高并发访问，会遇到不少问题，需要进行针对性扩展及优化。本课时，我会结合微博在使用 Redis 中遇到的问题，来分析如何在生产环境下对 Redis 进行扩展改造，以应对百万级 QPS。

功能扩展

对于线上较大流量的业务，单个 Redis 实例的内存占用很容易达到数 G 的容量，对应的 aof 会占用数十 G 的空间。即便每天流量低峰时间，对 Redis 进行 rewriteaof，减少数据冗余，但由于业务数据多，写操作多，aof 文件仍然会达到 10G 以上。

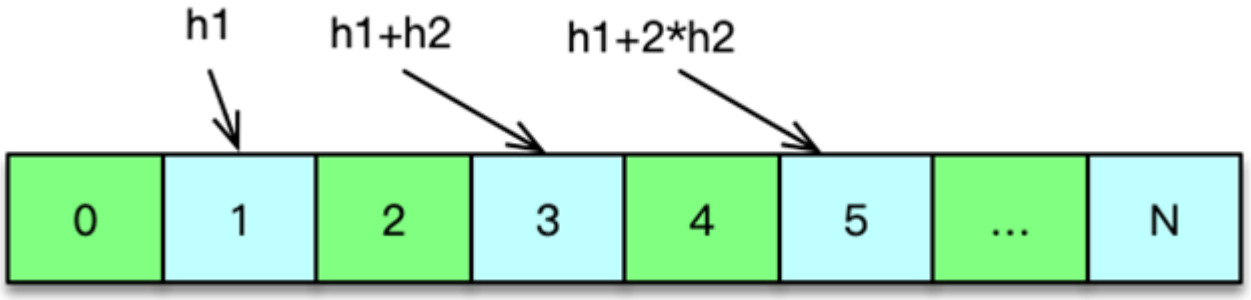
此时，在 Redis 需要升级版本或修复 bug 时，如果直接重启变更，由于需要数据恢复，这个过程需要近 10 分钟的时间，时间过长，会严重影响系统的可用性。面对这种问题，可以对 Redis 扩展热升级功能，从而在毫秒级完成升级操作，完全不影响业务访问。



热升级方案如下，首先构建一个 Redis 壳程序，将 redisServer 的所有属性（包括redisDb、client等）保存为全局变量。然后将 Redis 的处理逻辑代码全部封装到动态连接库 so 文件中。Redis 第一次启动，从磁盘加载恢复数据，在后续升级时，通过指令，壳程序重新加载 Redis 新的 so 文件，即可完成功能升级，毫秒级完成 Redis 的版本升级。而且整个过程中，所有 Client 连接仍然保留，在升级成功后，原有 Client 可以继续进行读写操作，整个过程对业务完全透明。

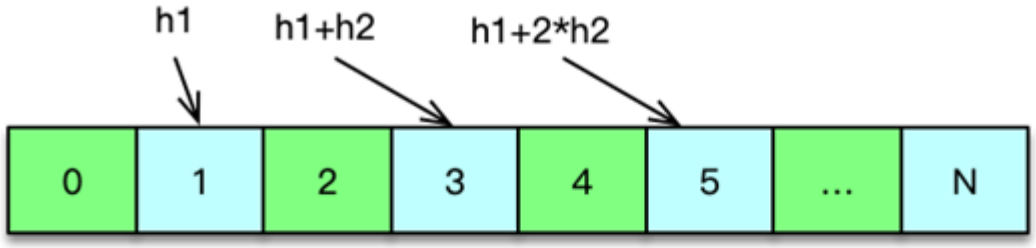
在 Redis 使用中，也经常会遇到一些特殊业务场景，是当前 Redis 的数据结构无法很好满足的。此时可以对 Redis 进行定制化扩展。可以根据业务数据特点，扩展新的数据结构，甚至扩展新的 Redis 存储模型，来提升 Redis 的内存效率和处

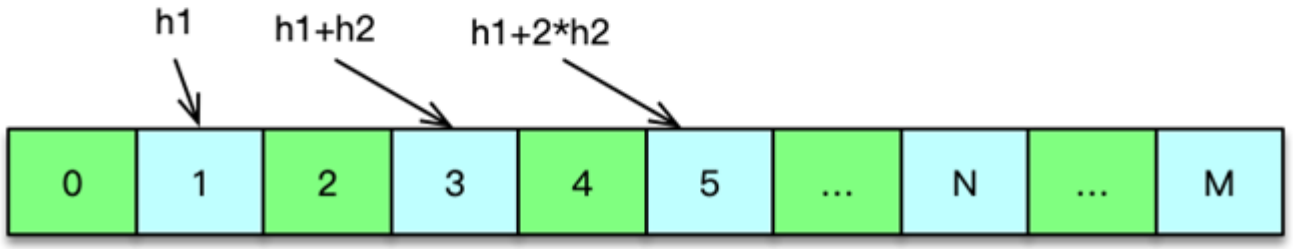
理性能。



在微博中，有个业务类型是关注列表。关注列表存储的是一个用户所有关注的用户 uid。关注列表可以用来验证关注关系，也可以用关注列表，进一步获取所有关注人的微博列表等。由于用户数量过于庞大，存储关注列表的 Redis 是作为一个缓存使用的，即不活跃的关注列表会很快被踢出 Redis。在再次需要这个用户的关注列表时，重新从 DB 加载，并写回 Redis。关注列表的元素全部 long，最初使用 set 存储，回种 set 时，使用 sadd 进行批量添加。线上发现，对于关注数比较多的关注列表，比如关注数有数千上万个用户，需要 sadd 上成千上万个 uid，即便分几次进行批量添加，每次也会消耗较多时间，数据回种效率较低，而且会导致 Redis 卡顿。另外，用 set 存关注列表，内存效率也比较低。

于是，我们对 Redis 扩展了 longset 数据结构。longset 本质上是一个 long 型的一维开放数组。可以采用 double-hash 进行寻址。

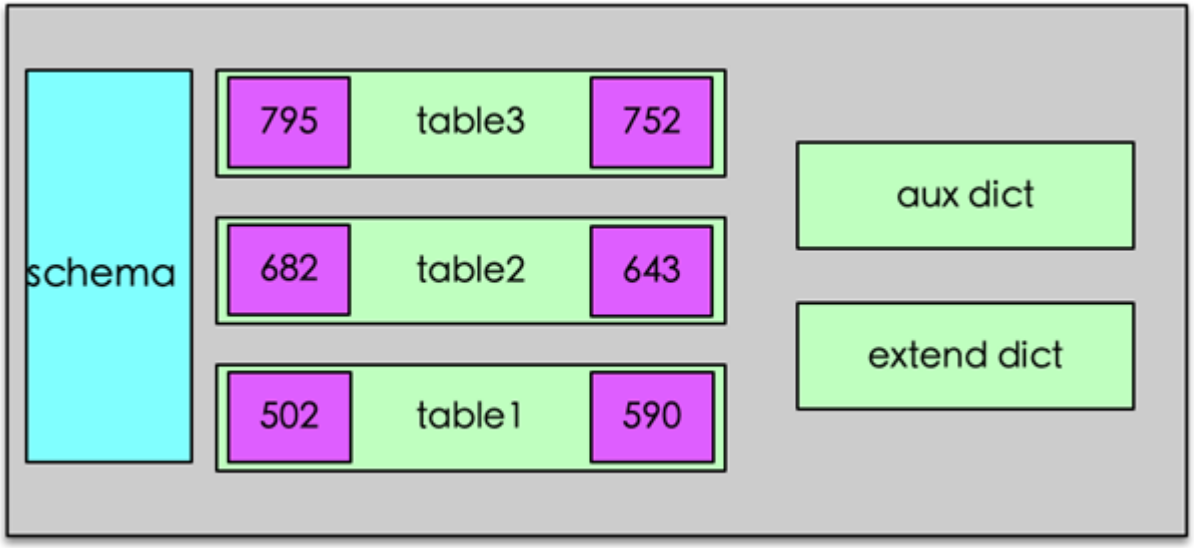




从 DB 加载到用户的关注列表，准备写入 Redis 前。Client 首先根据关注的 uid 列表，构建成 long 数组的二进制格式，然后通过扩展的 lsset 指令写入 Redis。Redis 接收到指令后，直接将 Client 发来的二进制格式的 long 数组作为 value 值进行存储。

longset 中的 long 数组，采用 double-hash 进行寻址，即对每个 long 值采用 2 个哈希函数计算，然后按 (h1 + n*h2)% 数组长度 的方式，确定 long 值的位置。n 从 0 开始计算，如果出现哈希冲突，即计算的哈希位置，已经有其他元素，则 n 加 1，继续向前推进计算，最大计算次数是数组的长度。

在向 longset 数据结构不断增加 long 值元素的过程中，当数组的填充率超过阀值，Redis 则返回 longset 过满的异常。此时 Client 会根据最新全量数据，构建一个容量加倍的一维 long 数组，再次 lsset 回 Redis 中。

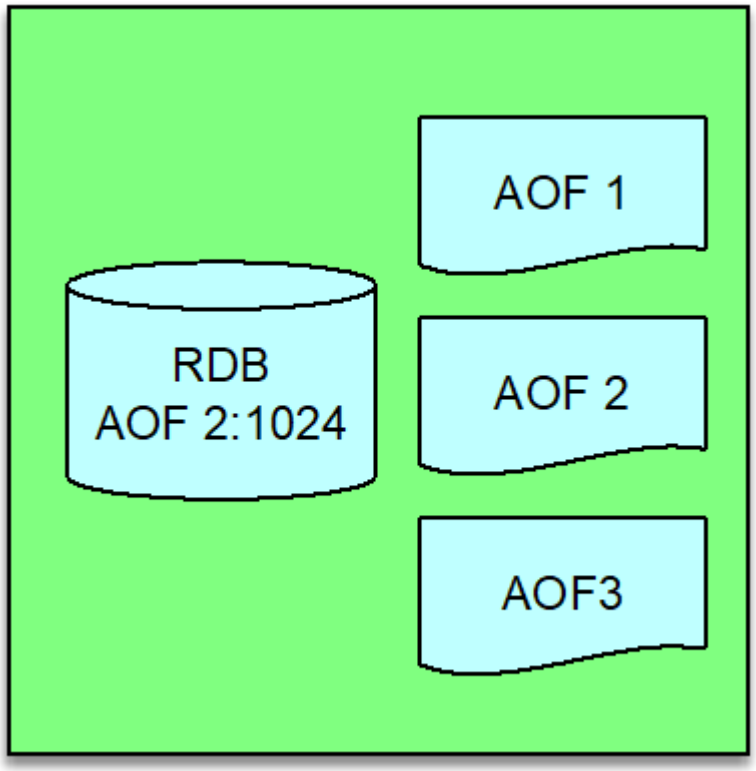


在移动社交平台中，庞大的用户群体，相互之间会关注、订阅，用户自己会持续分享各种状态，另外这些状体数据会被其他用户阅读、评论、扩散及点赞。这样，在用户维度，就有关注数、粉丝数、各种状态行为数，然后用户每发表的一条 feed、状态，还有阅读数、评论数、转发数、表态数等。一方面会有海量 key 需要进行计数，另外一方面，一个 key 会有 N 个计数。在日常访问中，一次查询，不仅需要查询大量的 key，而且对每个 key 需要查询多个计数。

以微博为例，历史计数高达千亿级，而且随着每日新增数亿条 feed 记录，每条记录会产生 4~8 种计数，如果采用 Redis 的计数，仅仅单副本存储，历史数据需要占用 5~6T 以上的内存，每日新增 50G 以上，如果再考虑多 IDC、每个 IDC 部署 1 主多从，占用内存还要再提升一个数量级。由于微博计数，所有的 key 都是随时间递增的 long 型值，于是我们改造了 Redis 的存储结构。

首先采用 cdb 分段存储计数器，通过预先分配的内存数组 Table 存储计数，并且采用 double hash 解决冲突，避免 Redis 实现中的大量指针开销。 然后，通过 Schema 策略支持多列，一个 key id 对应的多个计数可以作为一条计数记录，还支持动态增减计数列，每列的计数内存使用精简到 bit。而且，由于 feed 计数冷热区分明显，我们进行冷热数据分离存储方案，根据时间维度，近期的热数据放在内存，之前的冷数据放在磁盘， 降低机器成本。

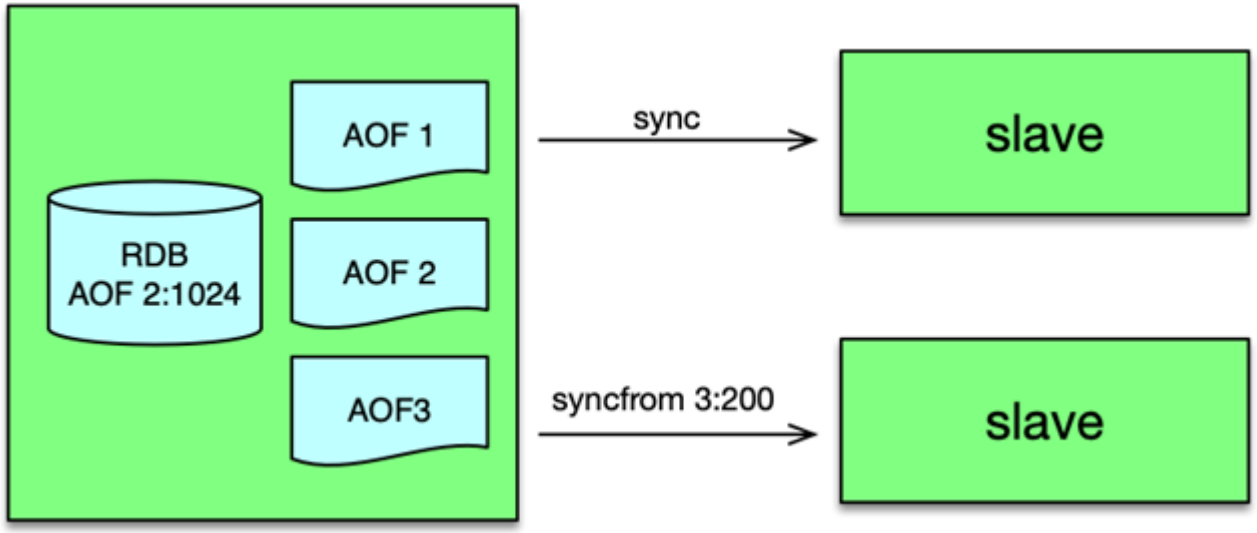
关于计数器服务的扩展，后面的案例分析课时，我会进一步深入介绍改造方案。



线上 Redis 使用，不管是最初的 sync 机制，还是后来的 psync 和 psync2，主从复制都会受限于复制积压缓冲。如果 slave 断开复制连接的时间较长，或者 master 某段时间写入量过大，而 slave 的复制延迟较大，slave 的复制偏移量落在 master 的复制积压缓冲之外，则会导致全量复制。

完全增量复制

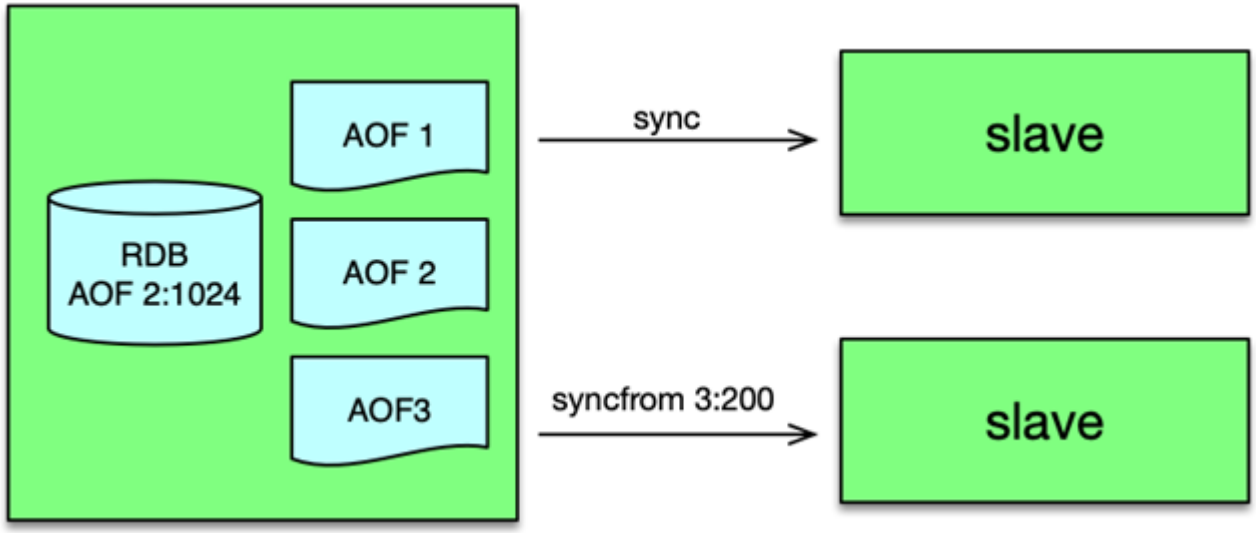
于是，微博整合 Redis 的 rdb 和 aof 策略，构建了完全增量复制方案。



在完全增量方案中，aof 文件不再只有一个，而是按后缀 id 进行递增，如 aof.00001、aof.00002，当 aof 文件超过阀值，则创建下一个 id 加 1 的文件，从而滚动存储最新的写指令。在 bgsave 构建 rdb 时，rdb 文件除了记录当前的内存数据快照，还会记录 rdb 构建时间，对应 aof 文件的 id 及位置。这样 rdb 文件和其记录 aof 文件位置之后的写指令，就构成一份完整的最新数据记录。

主从复制时，master 通过独立的复制线程向 slave 同步数据。每个 slave 会创建一个复制线程。第一次复制是全量复制，之后的复制，不管 slave 断开复制连接有多久，只要 aof 文件没有被删除，都是增量复制。

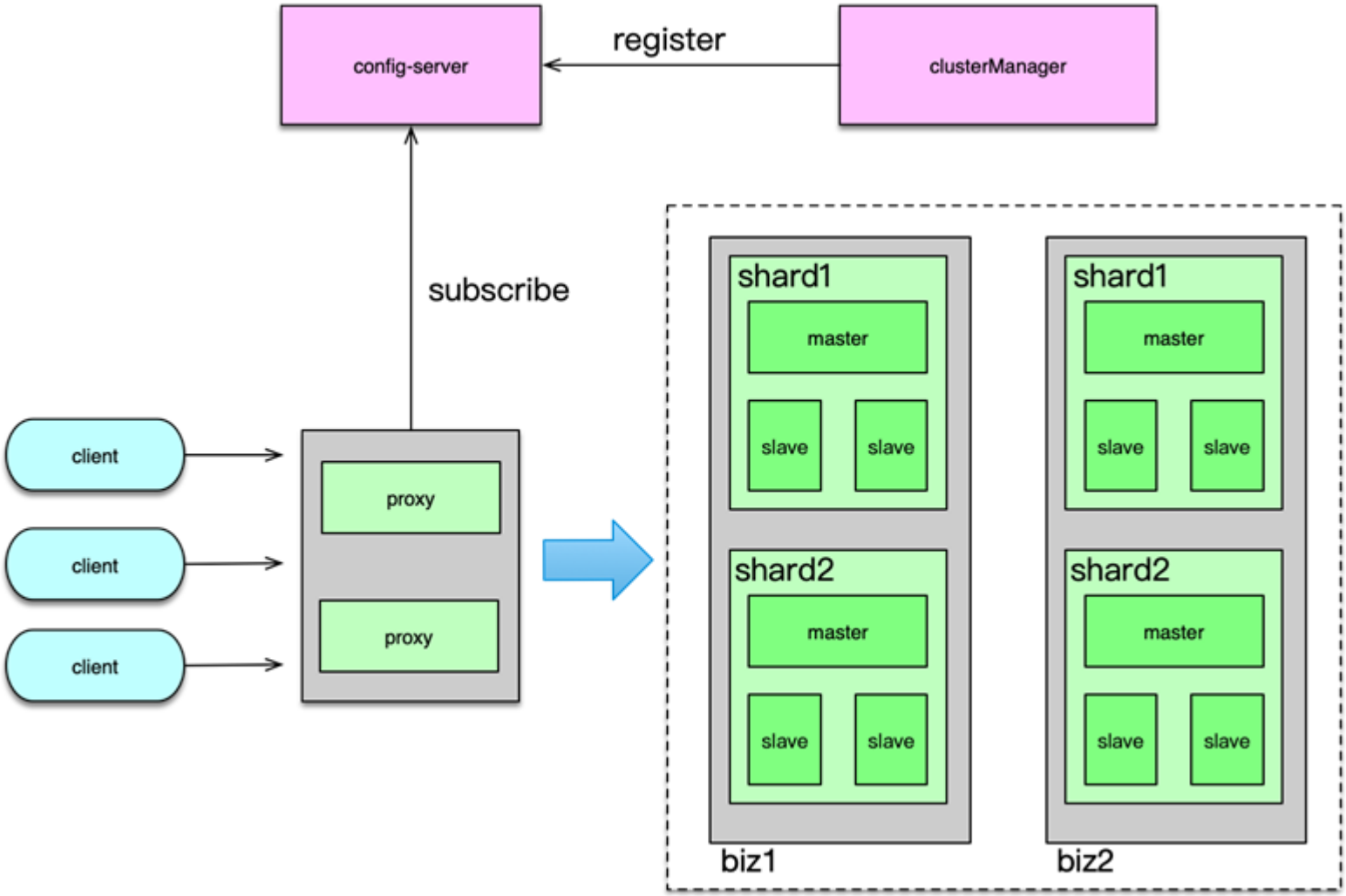
第一次全量复制时，复制线程首先将 rdb 发给 slave，然后再将 rdb 记录的 aof 文件位置之后的所有数据，也发送给 slave，即可完成。整个过程不用重新构建 rdb。



后续同步时，slave 首先传递之前复制的 aof 文件的 id 及位置。master 的复制线程根据这个信息，读取对应 aof 文件位置之后的所有内容，发送给 slave，即可完成数据同步。

由于整个复制过程，master 在独立复制线程中进行，所以复制过程不影响用户的正常请求。为了减轻 master 的复制压力，全增量复制方案仍然支持 slave 嵌套，即可以在 slave 后继续挂载多个 slave，从而把复制压力分散到多个不同的 Redis 实例。

集群管理



前面讲到，Redis-Cluster 的数据存储和集群逻辑耦合，代码逻辑复杂易错，存储 slot 和 key 的映射需要额外占用较多内存，对小 value 业务影响特别明显，而且迁移效率低，迁移大 value 容易导致阻塞，另外，Cluster 复制只支持 slave 挂在 master 下，无法支持 需要较多slave、读 TPS 特别大的业务场景。除此之外，Redis 当前还只是个存储组件，线上运行中，集群管理、日常维护、状态监控报警等这些功能，要么没有支持，要么支持不便。

因此我们也基于 Redis 构建了集群存储体系。首先将 Redis 的集群功能剥离到独立系统，Redis 只关注存储，不再维护 slot 等相关的信息。通过新构建的 clusterManager 组件，负责 slot 维护，数据迁移，服务状态管理。

Redis 集群访问可以由 proxy 或 smart client 进行。对性能特别敏感的业务，可以通过 smart client 访问，避免访问多一跳。而一般业务，可以通过 Proxy 访问 Redis。

业务资源的部署、Proxy 的访问，都通过配置中心进行获取及协调。clusterManager 向配置中心注册业务资源部署，并持续探测服务状态，根据服务状态进行故障转移，切主、上下线 slave 等。proxy 和 smart client 从配置中心获取配置信息，并持续订阅服务状态的变化。

上一页 下一页