Redis集群实现方案选择参考

# Redis概述

1.1 Redis是什么

Redis是一个开源的基于键值对的NoSQL数据库，官网表述为，Redis是一个基于内存的数据结构存储，可用作数据库、缓存和消息中间件。

1.2 Redis特性

* 速度快
* 基于键值对的数据结构服务器
* 功能丰富。支持String、List、Hash、Set、Sorted Set、Bitmaps、HyperLogLog、 GEO等多种数据结构。
* 单线程，避免了线程切换和锁的性能消耗
* 原子操作
* 可持久化（RDB与AOF）
* 发布/订阅
* 支持Lua脚本
* 分布式锁
* 事务
* 主从复制与高可用（Redis Sentinel）
* 集群（3.0版本以上）

1.3 Redis适用场景

1.3.1 缓存

缓存机制几乎在所有的大型网站都有使用，合理地使用缓存不仅可以加

快数据的访问速度，而且能够有效地降低后端数据源的压力。Redis提供了

键值过期时间设置，并且也提供了灵活控制最大内存和内存溢出后的淘汰策

略。可以这么说，一个合理的缓存设计能够为一个网站的稳定保驾护航。

1.3.2 排行榜系统

排行榜系统几乎存在于所有的网站，例如按照热度排名的排行榜，按照

发布时间的排行榜，按照各种复杂维度计算出的排行榜，Redis提供了列表

和有序集合数据结构，合理地使用这些数据结构可以很方便地构建各种排行

榜系统。

1.3.3 计数器应用

计数器在网站中的作用至关重要，例如视频网站有播放数、电商网站有

浏览数，为了保证数据的实时性，每一次播放和浏览都要做加1的操作，如

果并发量很大对于传统关系型数据的性能是一种挑战。Redis天然支持计数

功能而且计数的性能也非常好，可以说是计数器系统的重要选择。

1.3.4 社交网络

赞/踩、粉丝、共同好友/喜好、推送、下拉刷新等是社交网站的必备功能，由于社交网站访问量通常比较大，而且传统的关系型数据不太适合保存这种类型的数据，Redis提供的数据结构可以相对比较容易地实现这些功能。

1.3.5 消息队列系统

消息队列系统可以说是一个大型网站的必备基础组件，因为其具有业务解耦、非实时业务削峰等特性。Redis提供了发布订阅功能和阻塞队列的功能，虽然和专业的消息队列比还不够足够强大，但是对于一般的消息队列功能基本可以满足

# Redis集群实现方案概览

2.1 为什么采用Redis集群

通常，为了提高网站响应速度，总是把热点数据保存在内存中而不是直接从后端数据库中读取。Redis是一个很好的Cache工具。大型网站应用，热点数据量往往巨大，几十G上百G是很正常的事在这种情况下，无论是使用自己的物理主机，还是使用云服务主机，内存资源往往是有限制的，scale up不是一个好办法，我们需要scale out横向可伸缩扩展，这需要由多台主机协同提供服务即分布式多个Redis实例协同运行。其次，目前硬件资源成本降低，多核CPU，几十G内存的主机很普遍，对于主进程是单线程工作的Redis，只运行一个实例就显得有些浪费。同时，管理一个巨大内存不如管理相对较小的内存高效，因此，实际使用中，通常会根据业务场景，采用合适的Redis集群解决方案。

2.2 Redis集群常见思路

2.2.1 高可用主要场景和对应思路

适用于redis非重度用户，内存占用不大，总体内存大小的增长趋势可预估，有一定停机时间的系统——纵向扩容即可满足，可以对全库进行主从复制即满足需求而不需要做分片，一般针对单个小型项目的cache 等场景。一般采用一主多从的sentinel方案进行部署。

2.2.2 分片主要场景和对应思路

分片是为了应对业务增长带来的数据增长，需要对动态横向扩容有一定要求时采用。对于一般的分片采用一致性哈希，它极大的优化机器增删时带来的哈希目标漂移问题。同时对于Hash目标漂移时产生的严重的数据倾斜，可以利用虚拟节点来优化。基本上，物理节点有了一定规模后，只要不是同时挂多个节点，或者同时扩容多个节点，数据分片不会有太大的扰动。穿透过Cache的请求后端存储可以抗住即可。

稍微复杂的方案是可以使用“预分片（Pre-Sharding)”的方案，也称为按“桶”进行数据划分，即分配一个相当大的集合，对Key哈希的结果落在这个集合中，集合的每个元素又与具体的物理节点存在多对一的路由映射关系，这张路由表由一个配置中心进行维护。其实，一致性哈希中的虚拟节点，实际上也可以归类到Pre-Sharding方案中。换句话说，只要是key经过两次哈希，第一次Hash到虚拟节点，第二次Hash到物理节点，都可以算作Pre-Sharding。只不过区别在于，一致性哈希的第二次Hash其路由表是按照算法固定的，而分桶的第二次Hash其路由表是第三方可配的。

# Redis常见集群技术

长期以来，Redis本身仅支持单实例，内存一般最多10~20GB。这无法支撑大型线上业务系统的需求。而且也造成资源的利用率过低——毕竟现在服务器内存动辄100~200GB。

为解决单机承载能力不足的问题，各大互联网企业纷纷出手，“自助式”地实现了集群机制。在这些非官方集群解决方案中，物理上把数据“分片”（sharding）存储在多个Redis实例，一般情况下，每一“片”是一个Redis实例。

包括Redis官方推出的Redis Cluster，Redis集群有三种实现机制，分别介绍如下，希望对大家选型有所帮助。

3.1 主从高可用

该方案就是单实例形式，只是为了保证数据的安全，对于用户数据少，业务的前期可以采用。

Redis官方提供的Redis Sentinel是Redis的高可用实现方案，在实际的生产环境中，对提高整个系统的高可用性是非常有帮助的。

Redis Sentinel是包含若干个Sentinel节点和Redis数据节点，每个Sentinel节点会对数据节点和其余Sentinel节点进行监控，当它发现节点不可达时，会对节点做下线标识。如果被标识的是主节点，它还会和其他Sentinel节点进行“协商”，当大多数Sentinel节点都认为主节点不可达时，它们会选举出一个Sentinel节点来完成自动故障转移的工作，同时会将这个变化实时通知给Redis应用方。整个过程完全是自动的，不需要人工来介入，所以这套方案很有效地解决了Redis的高可用问题。

3.2 客户端分片方案

这种方案将分片工作放在业务程序端，程序代码根据预先设置的路由规则，直接对多个Redis实例进行分布式访问。这样的好处是，不依赖于第三方分布式中间件，实现方法和代码都自己掌控，可随时调整，不用担心踩到坑。

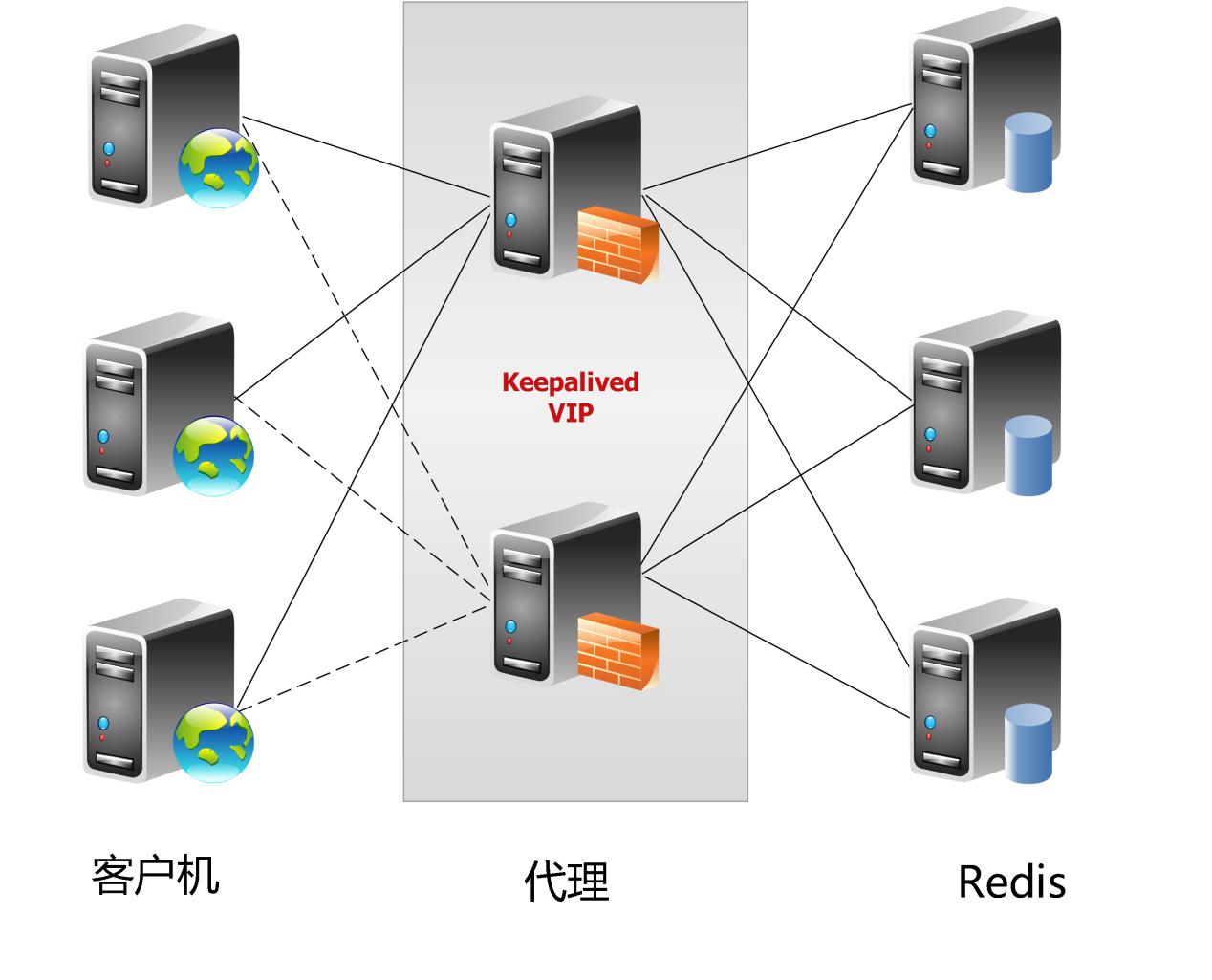
这实际上是一种静态分片技术。Redis实例的增减，都得手工调整分片程序。基于此分片机制的开源产品，现在仍不多见。

这种分片机制的性能比代理式更好（少了一个中间分发环节）。但缺点是升级麻烦，对研发人员的个人依赖性强——需要有较强的程序开发能力做后盾。如果主力程序员离职，可能新的负责人，会选择重写一遍。

所以，这种方式下，可运维性较差。出现故障，定位和解决都得研发和运维配合着解决，故障时间变长。这种方案，难以进行标准化运维，不太适合中小公司（除非有足够的DevOPS）。

3.3 代理中间件方案

代理中间件方案，将分片工作交给专门的代理程序来做。代理程序接收到来自业务程序的数据请求，根据路由规则，将这些请求分发给正确的Redis实例并返回给业务程序。



这种机制下，一般会选用第三方代理程序（而不是自己研发），因为后端有多个Redis实例，所以这类程序又称为分布式中间件。这样的好处是，业务程序不用关心后端Redis实例，运维起来也方便。虽然会因此带来些性能损耗，但对于Redis这种内存读写型应用，相对而言是能容忍的。基于该机制的开源产品Codis、Twemproxy，便是其中代表之一，应用非常广泛。

3.4 Redis Cluster(Offical)

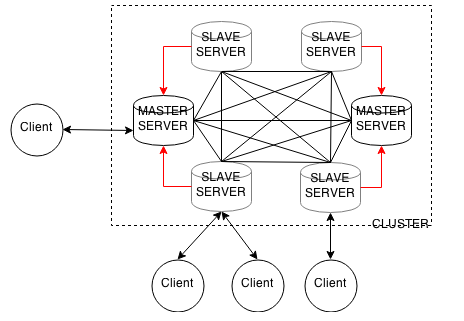
Redis Cluster将所有Key映射到16384个Slot中，集群中每个Redis实例负责一部分，业务程序通过集成的Redis Cluster客户端进行操作。客户端可以向任一实例发出请求，如果所需数据不在该实例中，则该实例引导客户端自动去对应实例读写数据。Redis Cluster的成员管理（节点名称、IP、端口、状态、角色）等，都通过节点之间两两通讯，定期交换并更新。

# Redis集群实现方案详述

4.1 Redis Cluster（Offical）

Redis Cluster是一种服务器Sharding技术，3.0版本开始正式提供。

Redis 4.0版本的集群是去中心化的结构，集群元数据信息分布在每个节点上，主备切换依赖于多个节点协商选主。redis 提供了redis-trib 工具做部署集群及运维等操作。客户端访问散列的db节点需依赖smart client，也就是客户端需要对redis返回的节点信息做判断选择路由等操作。

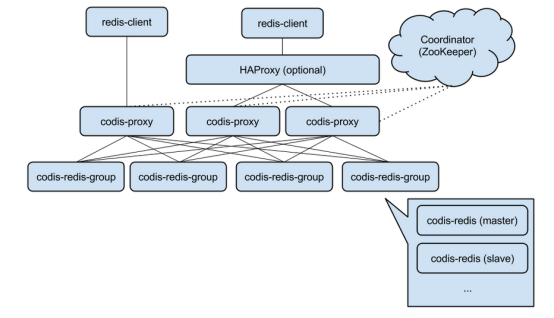


4.2 Codis

4.2.1 Codis是什么？

Codis 是 Wandoujia Infrastructure Team 开源的一个分布式 Redis 服务, 最新版本为Codis 3.x，基于Redis-3.3.8分支开发，目前处于稳定阶段。用户可以看成是一个无限内存的Redis 服务, 有动态扩/缩容的能力.。对偏存储型的业务更实用, 如果你需要 SUBPUB 之类的指令, Codis 是不支持的.。时刻记住 Codis 是一个分布式存储的项目，并不太适合 key 少，但是 value 特别大的应用， 而且你的 key 越少， value 越大，最后就会退化成单个 redis 的模型 （性能还不如 raw redis）。对于海量的 key, value不太大( <= 1M ), 随着业务扩展缓存也要随之扩展的业务场景有特效。

4.2.2 Codis组件构成



* Codis Server：基于 redis-3.2.8 分支开发。增加了额外的数据结构，以支持 slot 有关的操作以及数据迁移指令。具体的修改可以参考文档 redis 的修改。
* Codis Proxy：客户端连接的 Redis 代理服务, 实现了 Redis 协议。 除部分命令不支持以外(不支持的命令列表)，表现的和原生的 Redis 没有区别（就像 Twemproxy）。
  + 对于同一个业务集群而言，可以同时部署多个 codis-proxy 实例；
  + 不同 codis-proxy 之间由 codis-dashboard 保证状态同步。
* Codis Dashboard：集群管理工具，支持 codis-proxy、codis-server 的添加、删除，以及据迁移等操作。在集群状态发生改变时，codis-dashboard 维护集群下所有 codis-proxy 的状态的一致性。
  + 对于同一个业务集群而言，同一个时刻 codis-dashboard 只能有 0个或者1个；
  + 所有对集群的修改都必须通过 codis-dashboard 完成。
* Codis Admin：集群管理的命令行工具。

可用于控制 codis-proxy、codis-dashboard 状态以及访问外部存储。

* Codis FE：集群管理界面。
  + 多个集群实例共享可以共享同一个前端展示页面；
  + 通过配置文件管理后端 codis-dashboard 列表，配置文件可自动更新。
* Storage：为集群状态提供外部存储。
  + 提供 Namespace 概念，不同集群的会按照不同 product name 进行组织；
  + 目前仅提供了 Zookeeper、Etcd、Fs 三种实现，但是提供了抽象的 interface 可自行扩展。

4.2.4 Codis有什么好处？

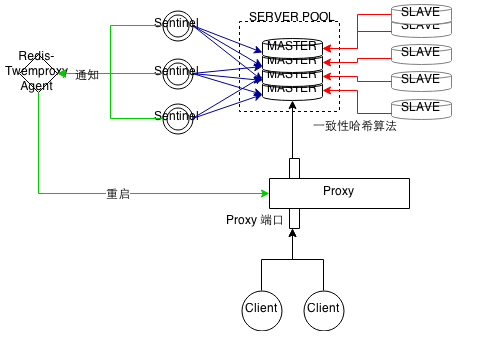
Codis使Redis获得动态扩容/缩容的能力，增减redis实例对client完全透明、不需要重启服务，不需要业务方担心 Redis 内存爆掉的问题，也不用担心申请太大, 造成浪费，业务方也不需要自己维护 Redis。

Codis支持水平扩容/缩容，扩容可以直接界面的 "Auto Rebalance" 按钮，缩容只需要将要下线的实例拥有的slot迁移到其它实例，然后在界面上删除下线的group即可。

## 4.3 Twemproxy(nutcracker)

4.3.1 Twemproxy是什么？

[Twemproxy](http://blog.nosqlfan.com/tags/twemproxy) ，又叫nutcracker，是Twitter开源的Redis和Memcached快速、轻量级的代理服务器，通过引入一个代理层，可以将其后端的多台 Redis 或 Memcached 实例进行统一管理与分配，使应用程序只需要在 Twemproxy 上进行操作，而不用关心后面具体有多少个真实的 Redis 或 Memcached 存储。



4.3.2 Twemproxy特性

* 单线程工作，用C语言开发
* 直接支持大部分的redis指令，redis客户端可以像访问正常redis实例一样访问TwemProxy。
* 支持失败节点自动删除
  + 可以设置重新连接该节点的时间
  + 可以设置连接多少次之后删除该节点
* 支持设置HashTag
  + 通过HashTag可以自己设定将两个key哈希到同一个实例上去
* 减少与redis的直接连接数
  + 保持与redis的长连接
  + 减少了客户端直接与服务器连接的链接数量
* 自动分片到后端多个redis实例上
* 多种hash算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins
* 多种分片算法：ketama(一致性hash算法的一种实现)、modular、random
* 可以设置后端实例的权重
* 避免单节点问题
  + 可以平行部署多个代理层，通过HAProxy做负载均衡，将redis的读写分散到多个twemproxy上。
* 支持状态监控
  + 可设置状态监控IP和端口，访问IP和端口可以得到一个json格式的状态信息串
  + 可设置监控信息刷新间隔时间
* 使用pipelining处理请求和响应
  + 连接复用，内存服用
  + 将多个连接请求，组成redis pipelining统一redis请求
* 并不是支持所有redis命令
  + 不支持redis的事务操作
  + 不支持针对多个值的操作，比如取sets的子交并补等
  + 使用SIDFF,SDIFFSTORE,SINTER,SINTERSTORE,SMOVE,SUNION and SUNIONSTORE 命令需要保证key都在同一个分片上。

最主要功能：用户不再直接操作真正的Redis，而且支持高性能的数据访问，而且支持分片处理，可以操作Redis集群。

# Redis集群实现方案详细对比

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Twemproxy** | **Codis** | | | **Redis Cluster** |
| 架构  设计 | 分布式CAP,牺牲P性能，而仅仅只是数据分片 | 分布式CAP,牺牲P性能，设计初衷为数据一致性 | | | 分布式CAP,牺牲C数据强一致性原则，追求redis最大的特点性能 |
| 设计  模型 | Proxy-based | Proxy-based | | | Gossip/P2P |
| 设计  思路 | 分布式逻辑和存储引擎分开，逻辑层proxy代理，存储用的是原子redis，当每个层面需要添加删除节点必须重启服务生效（要重新利用散列函数生成KEY分片更新） | 分布式逻辑和存储引擎分开，逻辑层codis-proxy，存储用的是修改过的codis-server,这种好处是proxy层可以自动扩展和收缩，存储层也同样可以，每个层面都可以热插拨 | | | 分布式的逻辑和存储引擎不分开，即又负责读写操作，又负责集群交互，升级困难，如果代码有bug，集群无法工作 |
| 架构  特点 | Proxy无状态，redis数据层有状态的，客户端可以请求任一proxy代理上面，再由其转发至正确的redis节点，该KEY分片算法至某个节点都是预先已经算好的，在proxy配置文件保存着，但是如果更新或者删除节点，又要根据一致性hash重新计算分片，并且重启服务 | Proxy无状态，codis-server分为组间，每个组存在一个主节点（必须有并且只能有一个）和多个从节点。客户端请求都是和proxy链接，链接哪个proxy都一样，然后由它根据zookeeper路由信息转发至正确节点，直接可以定位到正确节点上 | | | 这个结构为无中心的组织，不好把控集群当前的存活状态，客户端可以向任一节点发送请求，再有其重定向正确的节点上。如果在第一次请求和重定向期间cluster拓扑结构改变，则需要再一次或者多次重定向至正确的节点，但是这方面性能可以忽悠不计 |
| Codis  独特  之处 | 不支持 | 1、 有中心节点，逻辑问题交由proxy处理，codis还有个特点下层存储可以根据数据的冷热程度把冷数据暂时保存至磁盘，待其为热数据的时候又可以上线  2、 提供数据在线迁移的工具  比如需求要从redis或者twemproxy迁移数据至codis或者以后redis数据库中，又不想直接预热，可以借助其提供的redis-port命令行工具 | | | 不支持 |
| 开发  语言 | C语言 | Go语言、C语言 | | | C语言 |
| 服务  启动  方式 | 单进程 | 多进程 | | | 单进程 |
| 性能  问题 | 1、 单点的话比起原子redis降低20%左右；  2、 增加PIPELINE管道提高性能  只要是代理的设计性能瓶颈肯定在其，因redis实在太快 | 1、 相当于单redis实例40%性能丢失(从最开始的版本比Twemproxy慢20%至目前比其快100%)；  2、 弥补：增加proxy数量和支持多核，比起Twemproxy还是好一点，因Twemproxy最好的情况跑满一个CPU；  3、 弥补：增加PIPELINE管道提高性能  只要是代理的设计性能瓶颈肯定在其，因redis实在太快 | | | 没什么损失，追求的就是这个  1000个节点内拥有线性的伸缩性，和操作redis实例性能差不多 |
| 分片  算法 | Redis一致hash，当初设计好如后续变更修改（增减节点）需要配置proxy通知新的算法，重启服务 | 通过presharding采用solt槽位的形式，整个集群分为1024个哈希槽，分片算法位SlotId = crc32(key) % 1024，增减节点不需要重启服务 | | | 采用solt槽位的形式，整个集群分为16384个哈希槽，分片算法位SlotId = crc16(key) % 16384，增减节点不需要重启服务 |
| 所需  组件 | Redis、twemproxy(nutcracker) | Codis、zookeeper | | | Redis |
| 数据  一致性 | 不能保证强一致性  1、 如redis cluster第一种情况没有，设计不一样  2、 网络分裂，这种情况其有监控工具可以通知管理员某个主节点宕机，这时如果管理员切换HA（但是不提供自动提升从节点为主节点，因从节点变为主节点必须更新分片算法，重启服务），数据就会丢失，因主节点的写命令会丢失，除非再次AOF同步最后一条写命令，二者如国管理员可以判断其为网络分裂，等待网络恢复是主节点会向从节点同步写命令数据 | 强一致性  1、 数据迁移过程中数据强一致性性，因迁移都是一个个KEY原子迁移，每次操作都是给ZK发送信息，通知proxy，同时所有操作都是上一把锁，假如该期间有客户端访问，则提升访问该KEY的操作为优先操作，快速迁移至新节点，访问转移至新节点，不会访问老的KEY，如期间也可以中断正在同步的数据，也是不影响，因为redis没有回滚机制，要么成功要么失败  2、 网络分裂：因为它的设计初衷就是不考虑HA自动切换（后面添加该功能），等到网络恢复Zookeeper保证数据一致性，写命令不会丢失，所有操作都要在zookeeper上面注册 | | | 不能保证强一致性  比如官网给出的两种有可能丢失写命令的情况如下  1、 客户端向主节点A发送一条写命令，主节点是首先立马向客户端返回命令回复，然后再把刚刚的执行写命令同步至从节点，追求性能所致该设计  2、 网络分裂(network partition)，如果时间很长导致A节点的从节点转换为主节点，然这中间可能存在客户端正在和A节点通信也就被孤立了，这样写的命令将丢失，如当网络恢复A又重新加入集群 |
| 数据  的  特殊  安全 | 1、 比如某段时间业务数据一下爆表（内存写满），数据来不及迁移，这样的话redis会根据LRU策略，会淘汰一部分老的key，这个没办法，所以运维中当内存使用80%时应该扩容  2、 主从切换过程中有一部分数据丢失（主挂了到从切换上来丢失的这部分数据） | | | | |
| 磁盘  IO | 基于redis本身的持久化（RDB和AOF），肯定会存在数据丢失的情况 | | | | |
| 数据  的  迁移 | 不可在线迁移，并且迁移完之后需要修改配置文件的分片算法，再重新启动Twemproxy服务，重新识别分片算法 | | 采用sharding在线迁移数据，安全透明，可以自动平衡数据到其他节点，相当于可热插拨，不会造成响应时延（因迁移时会另外有个进程来做，数据迁移还是原子的数据迁移指令，这样迁移的话就会相当慢一些） | 在线迁移数据，动态迁移KEY值会造成响应时延（迁移数据是拷贝RDB数据文件，而且因为redis是单进程的），另外新节点solt又不自动，依赖ruby(redis cluster集群管理和分配脚本)脚本来平衡数据，无中心设计如此 | |
| 水平  扩容  缩容 | Redis存储层操作，不提供自动的解决方案，并且要自己写脚本支持数据的搬迁活动，然后更改proxy哈希算法，重启服务 | | Redis存储层，都是在线操作（扩容数据迁移，缩容的话先搬迁数据至别的节点，然后下线该节点） | 没有代理和存储之分，可以在线操作（扩容数据迁移，缩容的话先搬迁数据至别的节点，然后下线该节点） | |
| 主从  是否  必须 | 1、 没有数据复制不影响可用节点代替故障节点  2、 如果没有做备份，故障节点key全部丢失 | | 1、 故障节点如果没有备份，则丢失掉该组的所有数据，但是不影响其他组的运行，不会导致整个集群坏掉  2、 如有备份节点，则把其升为主节点，集群没有影响，正常运转（需要管理员手动变更从节点为主节点，最新版本添加HA功能） | 没有主从复制的节点一旦故障，将导致整个集群不可用，无法写入或者读入任何key，无法进行数据重新分片  官网建议：至少3主3从 | |
| 主从  特点 | 基于redis本身的主从方案（利用发布/订阅机制，采用广播形式），  采用异步复制（asynchronous replication）的方案，无论是master端还是slave端都不会引起阻塞，但是肯定是存在数据丢失的情况 | | | | |
| 集群  设计 | 1、 proxy部署高可用（多proxy结合keepalived+haporxy）  2、 redis层设计多主多从部署 | 1、 proxy部署（多proxy+zookeeper集群方案，并且结合keepalived+haporxy）  2、 codis-server部署多组，每组部署一主多从架构 | | 利用redis本身部署集群：至少3主3从 | |
| HA  方案 | Proxy层已经有了，由上面的设计，redis层基于自带的HA解决方案（sentinel），这里不阐述sentinel机制 | Proxy层已经有了，存储层本来设计就没有考虑自动HA切换，后面根据用户强烈的要求，目前添加codis-ha解决 | | 自主监控自动切换(把sentinel功能搬迁过来) | |
| 故障  监控 | 自带监控并告警 | 自带监控并告警 | | 目前还没有提供 | |
| 功能  限制 | 1、 不支持多key操作  2、 不支持MULTI/EXEC  3、 不支持EVAL | 比较多，可以查看官方文档 | | 1、 复杂的多KEY(set求并求交集)不能跨节点操作  2、 不支持MULTI/EXEC  3、 写丢失比较频繁 | |
| 提供  支持 | Twitter不再维护 | 目前活跃状态 | | 官网主打 | |
| 客户端  driver  工具 | 保持不变（redis支持的都支持） | 保持不变（redis支持的都支持） | | 只能支持cluster协议的客户端工具（目前官网上面说的是针对JAVA的是Jides，针对PHP的是Predis，而且不成熟 | |
| **概括** | 1、 轻量级  2、 在proxy层实现一致性哈希  3、 快速的故障转移  4、 可借助sentinel实现底层HA  5、 每次变更必须重启生效 | 1、 基于zookeeper的proxy高可用，zookeeper会记录整个集群的生存状态，则需要维护好zookeeper  2、 优势为动态水平扩容，平衡数据，在迁移的时候不影响业务访问和响应时间，这点很炫，也是它主打的方向  3、 Dashboard操作降低人失误率，图形直观查看信息  4、 强一致数据（也是设计的重点） | | 1、 性能好（也是设计的原则）  2、 无中心组织结构，有利有弊  3、 故障转移响应时间长  4、 有写丢失，比较频繁 | |