对话张冬洪 | 全面解读NoSQL数据库Redis的核心技术与应用实践

互联网和Web的蓬勃发展正在改变着我们的世界，随着互联网的不断发展和壮大，企业数据规模越来越大，并发量越来越高，关系数据库无法应对新的负载压力，随着Hadoop，Cassandra，MongoDB，Redis等NoSQL数据库的兴起，因其良好的可扩展性，弱化数据库的设计范式，弱化一致性要求，在解决海量数据和高并发的问题上明显优于关系型数据库。因而很快广泛应用于互联网业务中。

Redis作为基于K-V的NoSQL数据库，具有高性能、丰富的数据结构、持久化、高可用、分布式、支持复制等特性。从09年至今，经历8年多的锤炼，已经非常稳定，并且得到业界的广泛认可和使用，同时社区非常活跃。

今天我们有幸邀请到了Redis中国用户组主席张冬洪老师，请他分享Redis的核心技术，应用现状及发展方向等。

我是来自新浪微博研发中心的高级DBA 张冬洪，目前在微博带一个小组，主要负责微博平台、手机微博、话题、红包飞、开放平台、私信、以及内容管控项目的数据库产品运维和服务保障工作。工作中涉及的数据库产品主要包括MySQL、Redis、Memcached、MCQ、Kafka、Pika、Postgresql等。

微博研发中心数据库部门主要负责全微博平台的后端资源的托管和运维，涉及的资源种类比较多，数据量比较大，业务线和资源实例数目也是非常之多，并发量巨大。而这些正是微博这种体量的公司应该具有的，微博作为当今中文社交媒体的第一品牌，拥有超过3.76亿的月活用户，也是当前社会热点事件传播的最主要平台，其中包括但不限制于大型活动（如：里约奥运会、朱日和沙场大点兵等），春晚，明星动态（如：王宝强离婚事件、女排夺冠、乔任梁去世、白百合出轨、TFBOYS生日、鹿晗关晓彤CP等）。

而热点事件往往具有不可预见性和突发性，并且伴随着极短时间内流量的数倍增长，甚至更多，有时持续时间较长。如何快速应对突发流量的冲击，确保线上服务的稳定性，是一个非常巨大的挑战和有意义的事情。为了达到这一目标，需要有一个完善的，稳定可靠的，健壮的数据库运维体系来提供支撑和管理，所以我们团队也是在领导的指导下，有目标、有计划的开展一些数据库自动化运维平台的建设工作。

在业余的时间我和其他几大互联网公司的朋友一起发起了Redis中国用户组（简称CRUG），也欢迎大家加入我们。

Redis在过去的版本演进中，比较重大的变化包括哪些呢？在最新4.0版本上，您认为最核心的变化是什么呢？您关注的新特性有哪些，可以简单介绍下吗？

我最早接触应该是在12年的时候，当时最新的版本应该是2.6.x。那个时候也没有在线上用，只是学习Linux的时候了解过。所以知道Redis的版本号命名规则借鉴了Linux的方式，版本号第二位如果是奇数，则为非稳定版本，如果为偶数，则为稳定版本。

这里我说下稳定版本的一些主要改进吧：

•Redis2.6

1）键的过期时间支持毫秒

2）从节点提供只读功能

3）服务端支持Lua脚本

4）放开客户端连接数的硬编码限制

5）去掉虚拟内存相关功能等

- **Redis2.8**

1）完善主从复制功能，实现增量复制

2）Redis设置明显的进程名，在系统中ps命令即可查看

3）发布/订阅添加pub/sub命令

4）Redis Sentinel第二版发布，较Redis 2.6更加完善，可以线上使用

5）可以通过config set命令设置maxclients等

- **Redis3.0**

1）推出Redis的分布式集群 Redis Cluster

2）全新的embedded string对象编码结果，优化小对象的内存访问，在特定的工作负载下能大幅度提升性能

3）LRU算法提升

4）config set 设置maxmemory的时候可以设置不用的单位

5）新的Client pause命令，在指定时间内停止处理客户端请求等

- **Redis3.2**

1）添加GEO功能

2）新的List编码类型quicklist

3）SDS在速度和节省空间上都做了优化

4）Lua脚本功能增强

5）新的RDB格式，仍兼容旧版RDB，同时加载速度上也有提升

6）Cluster nodes命令加速等

在Redis4.0版本上，我认为最核心的功能应该是支持了module，这极大的丰富的Redis的功能，使得许多Redis本身不具有的，第三方开发者拓展的功能也能加载到Redis中当一个功能进行使用，比如RediSearch、ReJSON、Redis-ML等。除此之外，还看到有很多新特性：

1）psync2.0，优化了之前版本主从节点切换必然引起全量复制的问题

2）提供全新的缓存剔除算法LFU，并对已有算法进行了优化

3）提供了非阻塞del和flushall和flushdb功能，有效解决了删除bigkey可能造成的Redis阻塞

4）提供了RDB-AOF混合持久化格式

5）提供memory命令，实现对内存的更为全面的监控统计

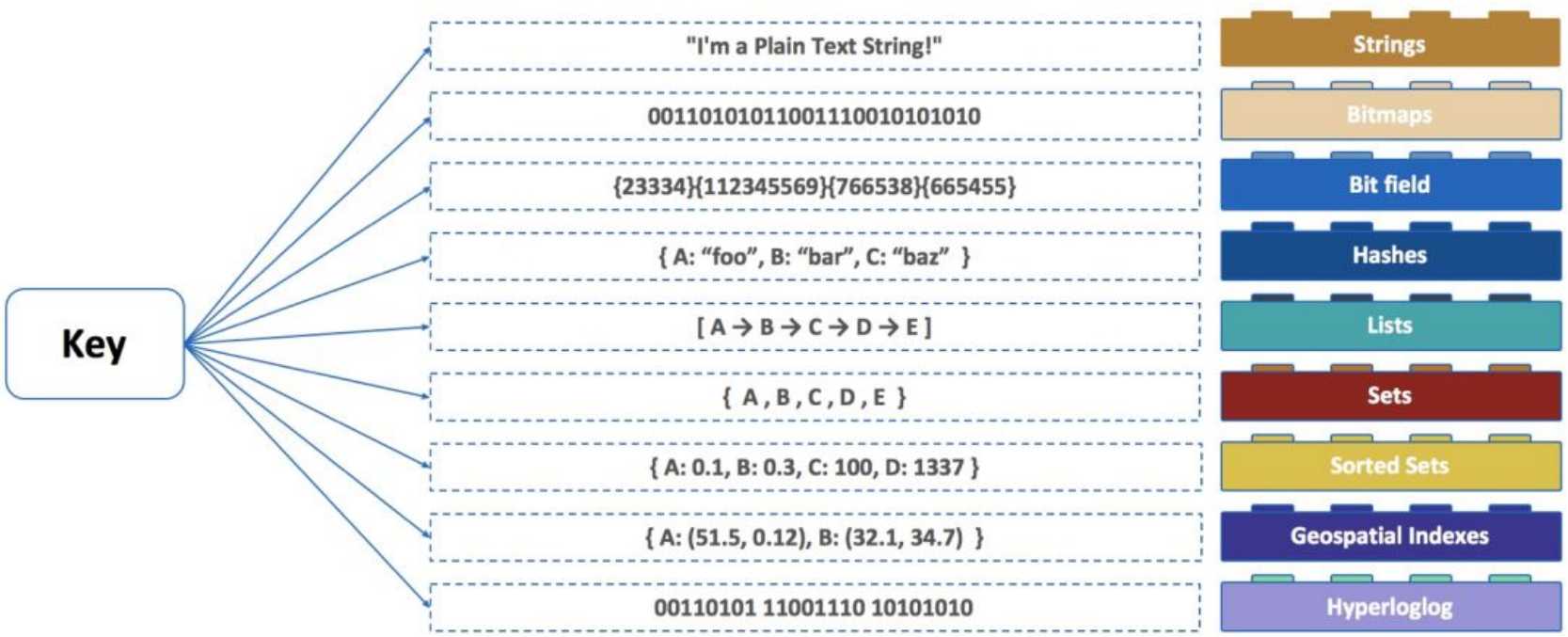
6）Redis Cluster 兼容NAT和Docker

7）引入Jemalloc库，优化内存访问等等

您能介绍一下Redis中的数据类型和它们的使用场景吗?

Redis之所以能够被广泛的应用于企业的架构中，而且是不可或缺的重要组成部分，也可以说是标配吧，**其中很重要的一点就是得益于它具有丰富的数据结构，这也是它逐渐替代Memcached，备受青睐的重要原因**。那么Redis都提供哪些数据类型呢?

相信对Redis有了解过的同学都知道，它的数据类型有：String、Hash、List、Set、Zset、Bitmaps、HyperLogLog、GEO等。



随着互联网的兴起和Redis技术的不断完善和发展，它已经被广泛应用于各行各业中，应用场景也是百花齐放。比如：会话缓存（Session cache）、全页缓存（FPC）、手机验证码、访问频率限制/黑白名单、消息队列、发布与订阅、消息通知、排名/排行榜/最新列表、计数器（比如微博的转评赞计

数、阅读数（浏览数，视频播放计数）、博文数（发帖数）、粉丝数、关注数（喜欢商品数）、未读数（动态数））、共同好友/喜好/标签、推送、下拉刷新、私信、商品库存管理（限时的优惠活动信息）、证券指标实时计算，发号器/UUID、以及随着LBS（基于位置服务）的发展，加入的GEO（地理信息定位）的功能和基于Lua自定义命令或功能等等。大家在使用过程中，需要结合自己的业务场景，选择正确的数据类型。

Redis数据库有哪些主要的特点和优势？使用时有什么需要关注的点，可能会带来哪些问题，可否通过实践案例详细描述一下？

Redis作为基于K-V的NoSQL数据库，具有高性能、丰富的数据结构、持久化、高可用、分布式、支持复制等特性。从09年至今，经历8年多的锤炼，已经非常稳定，并且得到业界的广泛认可和使用，同时社区非常活跃，开发者又很严谨，这使得Redis版本非常精简，bug fix非常高效。根据similarweb.com的统计，中国Redis用户占全球Redis用户的40.96%，所以我们在使用的过程中遇到的问题，大部分可能都有解决方案。

需要关注的点比较多，之前在CRUG深圳站活动的分享中也提到过一些，下面举一些例子吧：

1)安全问题：Redis用非root用户启动，并且运行在内网，尽可能不要暴露在外网，配置认证requirepass xxx，减少被攻击的风险；开启危险命令认证（keys-need-auth yes　rename-command KEYS MY_KEYS）

2)容量问题：合理评估；合理使用内存分配策略（no-enviction、allkeys-random、allkeys-lru、volatile-random、volatile-ttl、volatile-lru）；检查是否有内存碎片；选择合适的服务类型（Redis cluster或者pika）；水平拆分；性能满足的条件下选择诸如ziplist类的内部编码

3)big-key问题：可能会引起慢查或者带宽瓶颈，按照业务逻辑拆成小key或者业务解耦剥离big-key或直接改用其他存储方式

4)hot-key问题：Redis是单进程，节点实例很容易成为系统的短板，垂直扩容；增加local cache；如果只是简单的k-v结构，可以考虑使用Memcached

5)**使用姿势问题**：避免使用阻塞操作（如：flushall、flushdb、keys *等）；尽量使用Pipeline，减少syscall带来的网络IO，但要注意限制数据量大小；对多个元素操作时，像使用SORT、LREM、SUNION，计算复杂度为O(N), 避免线上乱用；尽可能使用最新的版本

6)**Key过期问题**：合理设置过期时间；如果存在许多该过期的key而没被及时删除，可以通过命令scan、hscan、sscan、zscan、keys *遍历一遍key的方式实现

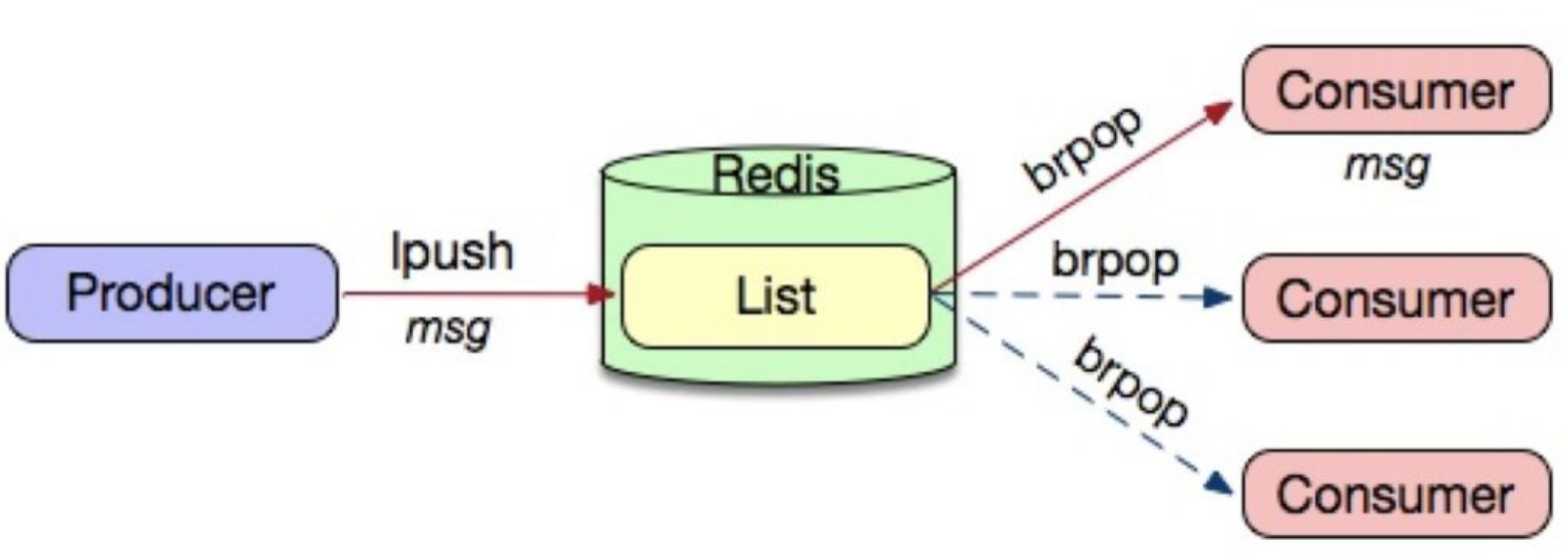
7)**配置上**：建议开启tcp-keepalive，tcp-backlog，从库设置readonly yes

8)**系统设置**：关闭NUMA；关闭transparent _hugepage; 关闭swap

等等，大家有问题的时候可以相互交流。

Redis做消息队列，有两种实现方式：

第一种：通过数据结构List来实现



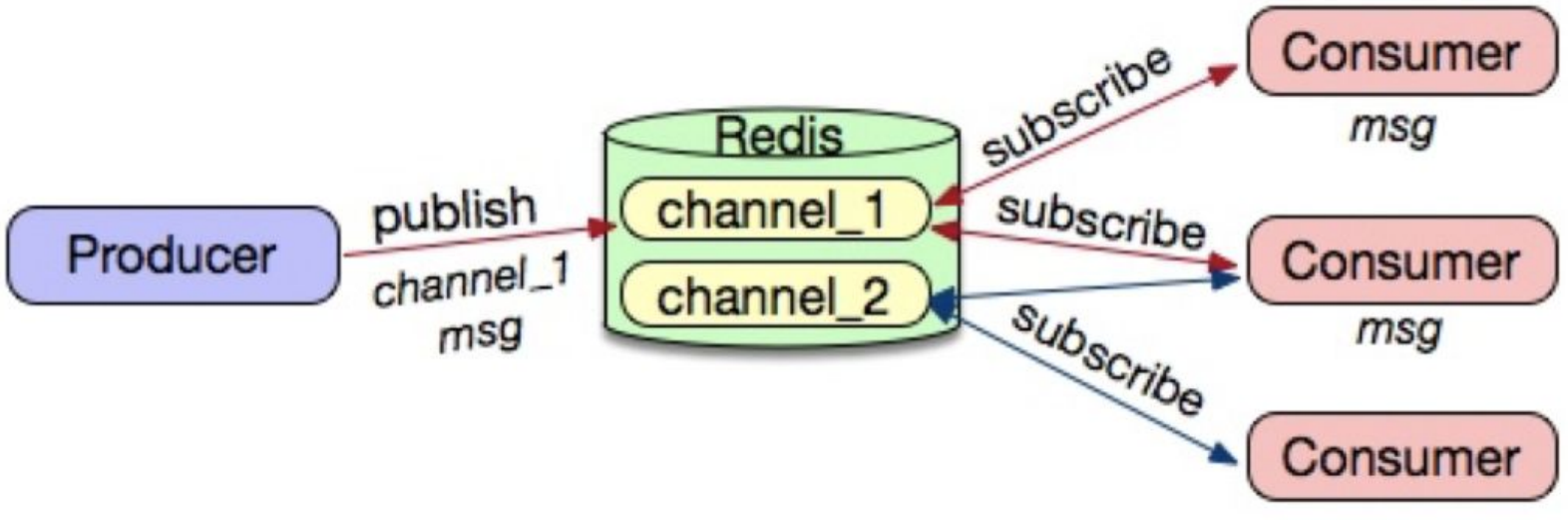
优点：能够实现持久化；支持集群；接口使用简单

缺点：

- 如上图所示，一条消息只会被一个消费者消费，所以不存在有多个消费者消费一条消息
- 生产者和消费者的高可用或崩溃后的处理机制需要自己实现

- 当生产者消息写入太快，消费者消费太慢，则有可能会导致内存溢出问题，导致进程crash

第二种：通过pub/sub来实现



优点：

一个生产者可以对应多个消费者，但是必须保证消息发布者和消息的订阅者同时在线，否则，否则一旦消息订阅者由于各种异常情况而被迫断开连接，在其重新连接后，其离线期间的消息是无法被重新通知的（即发即弃）。当然，生产者不需要关心有多少的订阅者，也不用关心订阅者的具体信息，而订阅者可以根据需要自由选择订阅哪些频道：支持集群；接口使用简单等。

缺点：

- 没有持久化机制，属于即发即弃模式，因此也不需要制定消息的备份和恢复机制
- Redis没有提供保证pub/sub消息性能的方案
- 当大量的消息到达Redis服务时，如果订阅者不能及时完成消费，则就会导致消息堆积，引发上面一样的内存问题

要了解Redis的集群功能，可以从数据分片、数据迁移、集群通讯、故障检

测以及故障转移等方面进行了解，Cluster相关的代码也不是很多，注释也很详细，可自行查看，地址是：
https://github.com/antirez/redis/blob/unstable/src/cluster.c。

这里由于篇幅的原因，主要从数据分片和数据迁移两方面进行详细介绍：

- 数据分片

Redis cluster在设计中没有使用一致性哈希（consistency hashing），而是使用数据分片（sharding），引入哈希槽（hash slot）来实现；一个 redis cluster包含16384（0~16383）个哈希槽，存储在redis cluster中的所有的键都会被映射到这些slot中，集群中的每个键都属于这16384个哈希槽的其中一个，集群使用公式slot=CRC16（key）/16384来计算key属于哪个槽，其中CRC16(key)语句用于计算key的CRC16 校验和。

集群中的每个主节点（Master）都负责处理16384个哈希槽中的一部分，当集群处于稳定状态时，每个哈希槽都只由一个主节点进行处理，每个主节点可以有一个到N个从节点（Slave），当主节点出现宕机或网络断线等不可用时，从节点能自动提升为主节点进行处理。

```
typedef struct clusterNode {
    mstime_t ctime; /* Node object creation time. */
    char name[CLUSTER_NAMELEN]; /* Node name, hex string, sha1-size */
    int flags;      /* CLUSTER_NODE_... */
    uint64_t configEpoch; /* Last configEpoch observed for this node */
    unsigned char slots[CLUSTER_SLOTS/8]; /* slots handled by this node */
    int numslots;   /* Number of slots handled by this node */
    int numslaves;  /* Number of slave nodes, if this is a master */
    struct clusterNode **slaves; /* pointers to slave nodes */
    struct clusterNode *slaveof; /* pointer to the master node. Note that it
                                    may be NULL even if the node is a slave
                                    if we don't have the master node in our
                                    tables. */
    mstime_t ping_sent;      /* Unix time we sent latest ping */
    mstime_t pong_received;  /* Unix time we received the pong */
    mstime_t fail_time;      /* Unix time when FAIL flag was set */
    mstime_t voted_time;     /* Last time we voted for a slave of this master */
    mstime_t repl_offset_time;  /* Unix time we received offset for this node */
    mstime_t orphaned_time;     /* Starting time of orphaned master condition */
    long long repl_offset;      /* Last known repl offset for this node. */
    char ip[NET_IP_STR_LEN];  /* Latest known IP address of this node */
    int port;                   /* Latest known clients port of this node */
    int cport;                  /* Latest known cluster port of this node. */
    clusterLink *link;          /* TCP/IP link with this node */
    list *fail_reports;         /* List of nodes signaling this as failing */
} clusterNode;
```

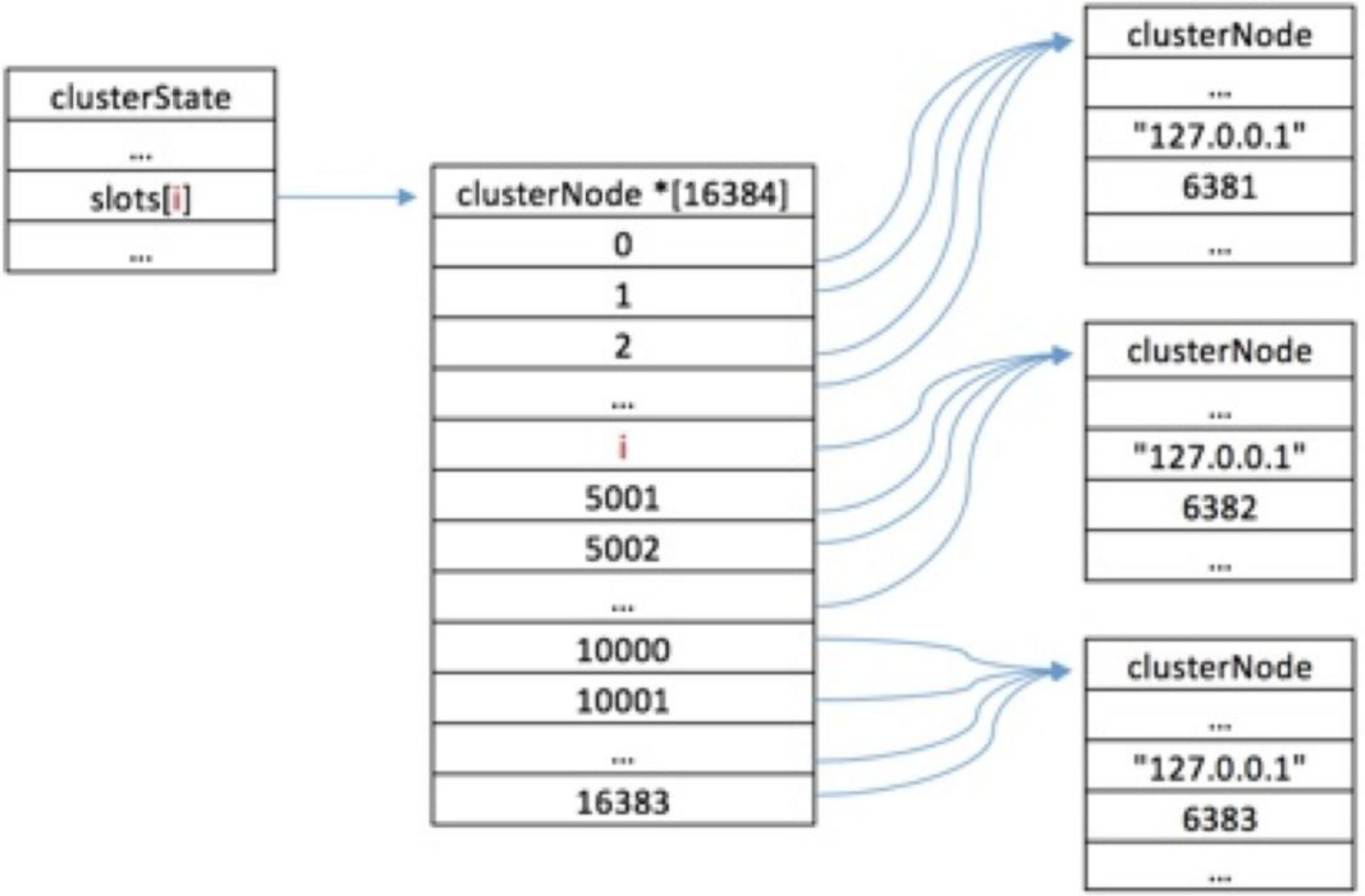
如上，clusterNode数据结构中的slots和numslots属性记录了节点负责处理哪些槽。其中，slot属性是一个二进制位数组(bitarray),其长度为16384/8=2048 Byte，共包含16384个二进制位。集群中的master节点用bit（0和1）来标识对于某个槽是否拥有。比如，对于编号为1的槽，master只要判断序列的第二位（索引从0开始）的值是不是1即可，时间复杂度为O(1)。

索引	0	1	2	…	16382	16383
值	1	1	0	0	1	0

集群中所有槽的分配信息都保存在clusterState数据结构的slots数组中，程序要检查槽i是否已经被分配或者找出处理槽i的节点，只需要访问clusterState.slots[i]的值即可，复杂度也为O(1)。clusterState数据结构如下所示：

```
typedef struct clusterState {
    clusterNode *myself;  /* This node */
    uint64_t currentEpoch;
    int state;            /* CLUSTER_OK, CLUSTER_FAIL, ... */
    int size;             /* Num of master nodes with at least one slot */
    dict *nodes;          /* Hash table of name -> clusterNode structures */
    dict *nodes_black_list; /* Nodes we don't re-add for a few seconds. */
    clusterNode *migrating_slots_to[CLUSTER_SLOTS];
    clusterNode *importing_slots_from[CLUSTER_SLOTS];
    clusterNode *slots[CLUSTER_SLOTS];
    zskiplist *slots_to_keys;
    /* The following fields are used to take the slave state on elections. */
    mstime_t failover_auth_time; /* Time of previous or next election. */
    int failover_auth_count;    /* Number of votes received so far. */
    int failover_auth_sent;     /* True if we already asked for votes. */
    int failover_auth_rank;     /* This slave rank for current auth request. */
    uint64_t failover_auth_epoch; /* Epoch of the current election. */
    int cant_failover_reason;   /* Why a slave is currently not able to
                                   failover. See the CANT_FAILOVER_* macros. */
    /* Manual failover state in common. */
    mstime_t mf_end;            /* Manual failover time limit (ms unixtime).
                                   It is zero if there is no MF in progress. */
    /* Manual failover state of master. */
    clusterNode *mf_slave;      /* Slave performing the manual failover. */
    /* Manual failover state of slave. */
    long long mf_master_offset; /* Master offset the slave needs to start MF
                                   or zero if stil not received. */
    int mf_can_start;           /* If non-zero signal that the manual failover
                                   can start requesting masters vote. */
    /* The followign fields are used by masters to take state on elections. */
    uint64_t lastVoteEpoch;     /* Epoch of the last vote granted. */
    int todo_before_sleep; /* Things to do in clusterBeforeSleep(). */
    long long stats_bus_messages_sent;  /* Num of msg sent via cluster bus. */
    long long stats_bus_messages_received; /* Num of msg rcvd via cluster bus.*/
} clusterState;
```

查找关系如下图：

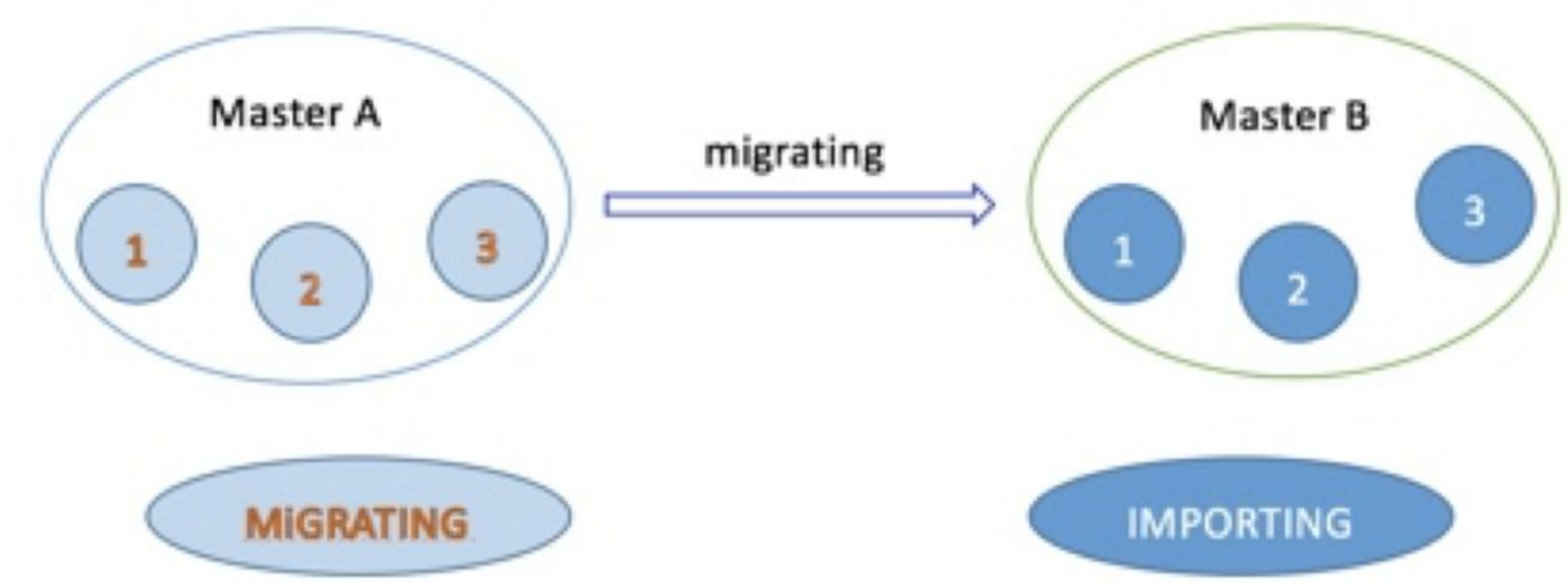


- 数据迁移

数据迁移可以理解为slot和key的迁移，这个功能很重要，极大的方便了集群做线性扩展，实现平滑的扩容或缩容。

那么它是一个怎样的实现过程呢?

下面举个例子：现在要将Master A节点中的编号为1、2、3的slot迁移到Master B节点中，在slot迁移的中间状态下，slot 1、2、3在Master A节点的状态表现为MIGRATING,在Master B节点的状态表现为IMPORTING。



MIGRATING状态

这个状态如上图所示是被迁移slot在当前所在Master A节点中出现的一种状态，预备迁移slot从Mater A到Master B的时候，被迁移slot的状态首先变为MIGRATING状态，当客户端请求的某个key所属的slot的状态处于MIGRATING状态的时候，可能会出现以下几种情况：

> 1）如果key存在则成功处理
>
> 2）如果key不存在，则返回客户端ASK，客户端根据ASK首先发送ASKING命令到目标节点，然后发送请求的命令到目标节点
>
> 3）当key包含多个命令时：
>
> - 如果都存在则成功处理
> - 如果都不存在，则返回客户端ASK
> - 如果一部分存在，则返回客户端TRYAGAIN，通知客户端稍后重试，这样当所有的key都　迁移完毕的时候客户端重试请求的时候回得到ASK，然后经过一次重定向就可以获取这批键

此时并不刷新客户端中node的映射关系

IMPORTING状态

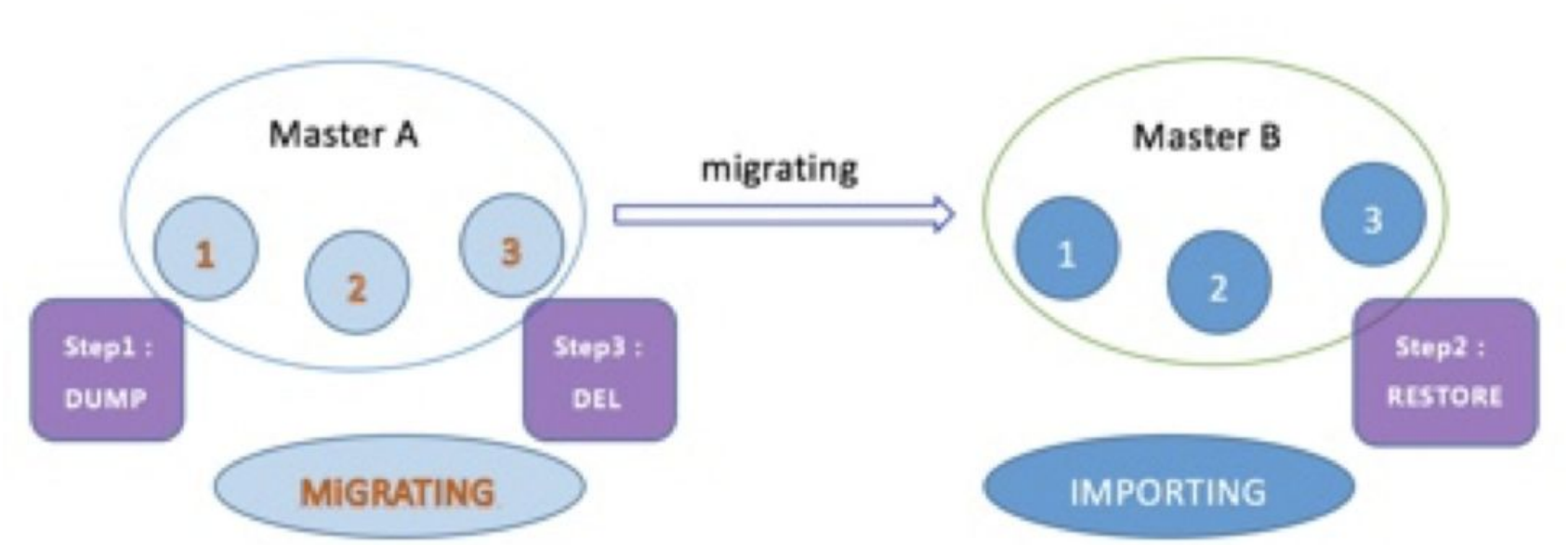
这个状态如上图所示是被迁移slot在目标Master B节点中出现的一种状态，

预备迁移slot从Mater A到Master B的时候，被迁移slot的状态首先变为IMPORTING状态。在这种状态下的slot对客户端的请求可能会有下面几种影响：

> 1）如果key不存在则新建
>
> 2）如果key不在该节点上，命令会被MOVED重定向，刷新客户端中node的映射关系
>
> 如果是ASKING命令则命令会被执行，从而key没在被迁移的节点，已经被迁移到目标节点的情况命令可以被顺利执行。

键空间迁移

这是完成数据迁移重要的一步，键空间迁移是指当满足了slot迁移前提的情况下，通过相关命令将slot 1、2、3中的键空间从Master A节点转移到Master B节点，这个过程由MIGRATE命令经过3步真正完成数据转移。步骤示意如下：



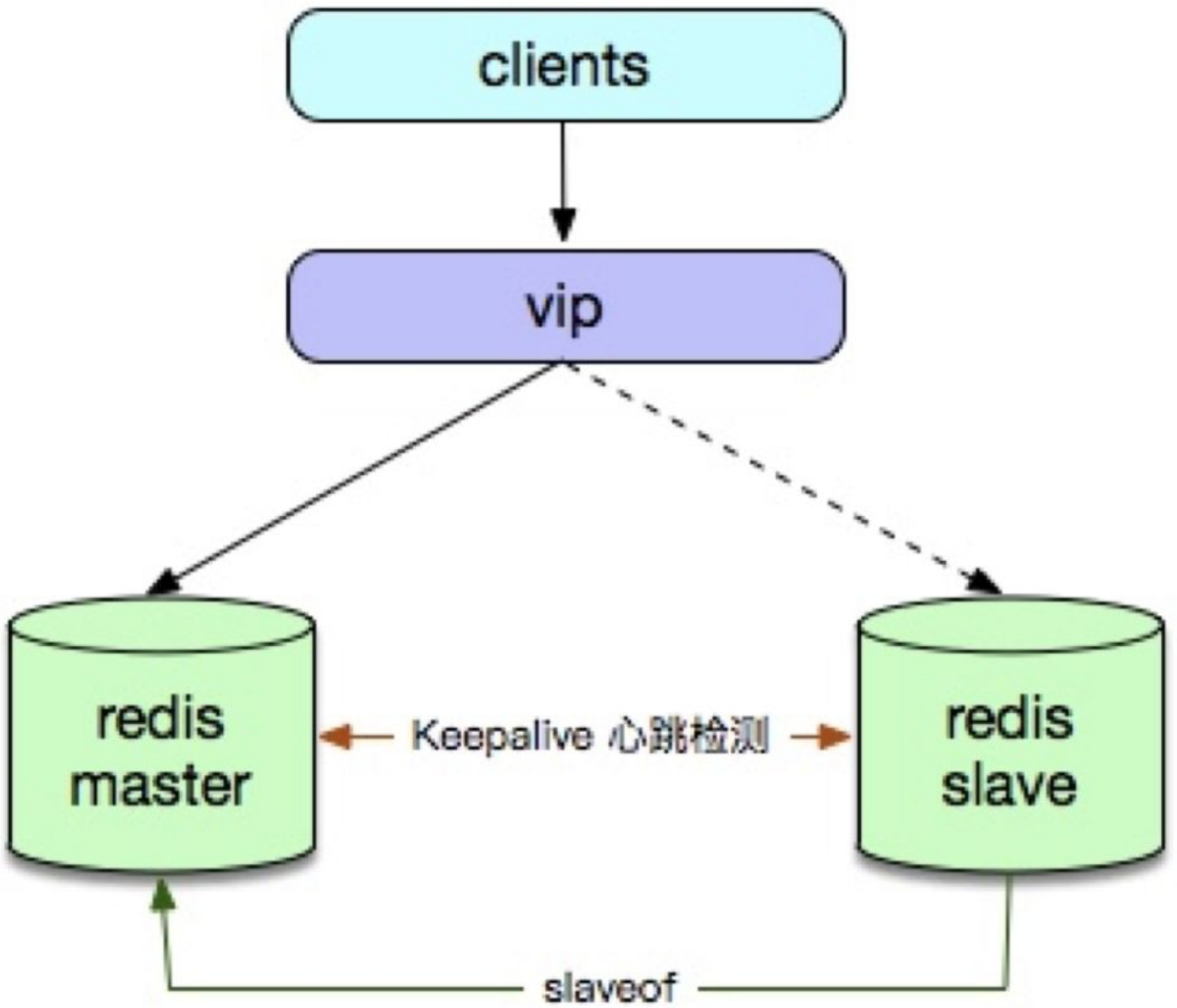
经过上面三步可以完成键空间数据迁移，然后再将处于MIGRATING和IMPORTING状态的槽变为常态即可，从而完成整个重新分片的过程。

请您简单介绍下Redis的高可用方案，如果可以，结合您的实践经验和案例进行分析

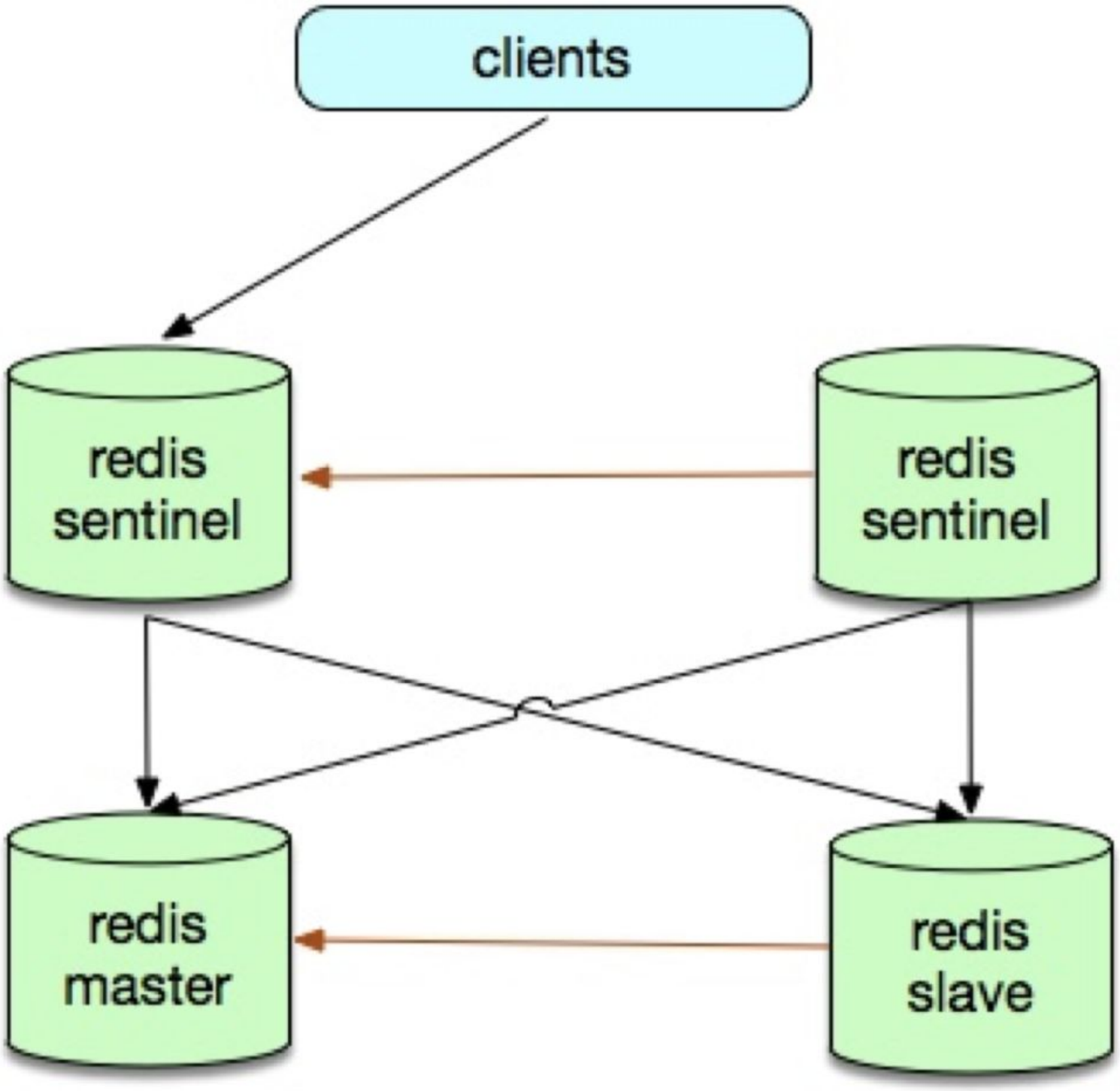
根据业务的规模以及Redis使用环境的不同，Redis的高可用方案也比较多。

这里举一些例子说明一下业界比较常用的一些方案吧。需要说明一下的是下面的这些方案不涉及到同城多活或异地多活的场景，但是部分方案能够做到跨数据中心的灾备。

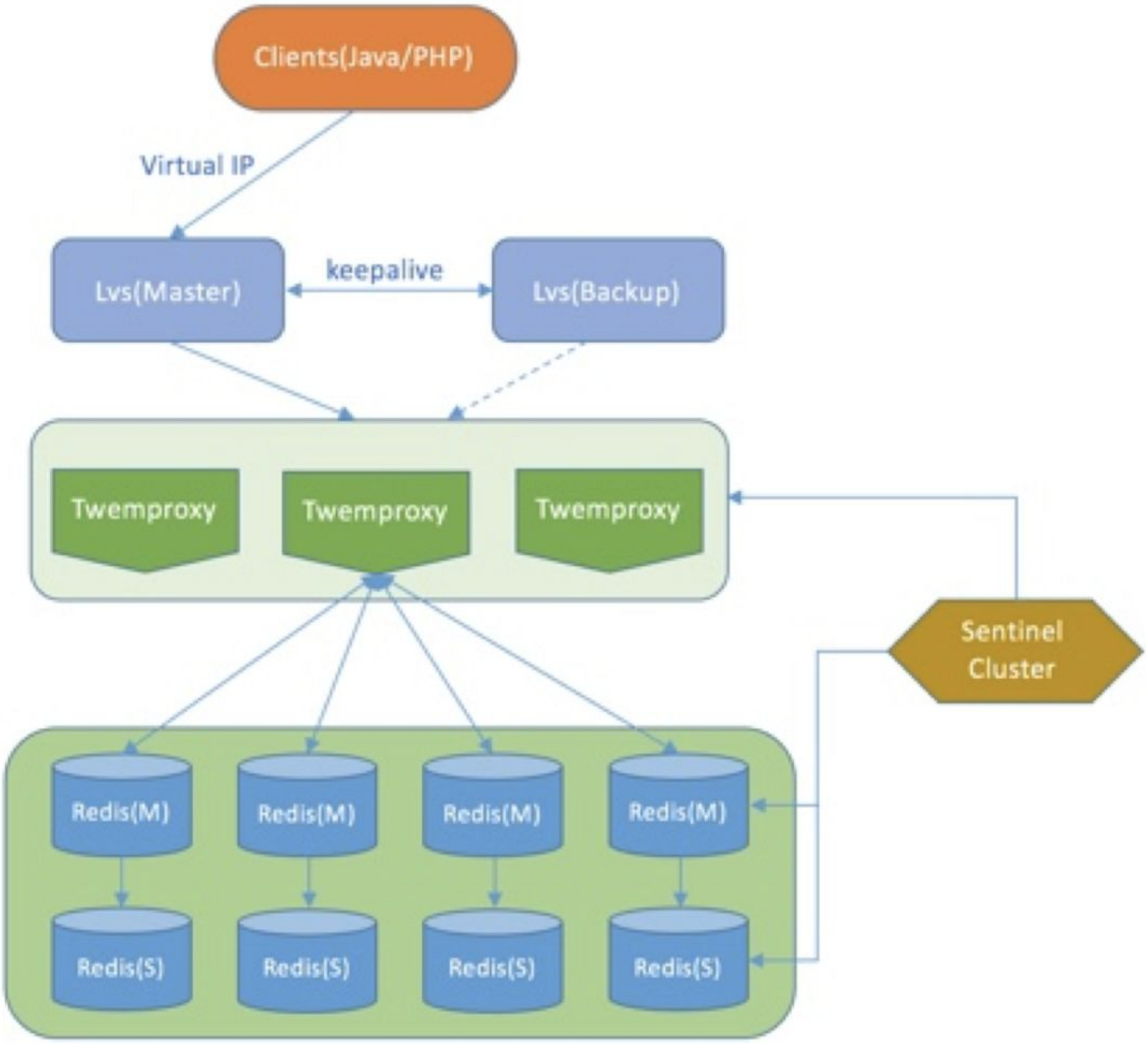
•Keepalive + Redis



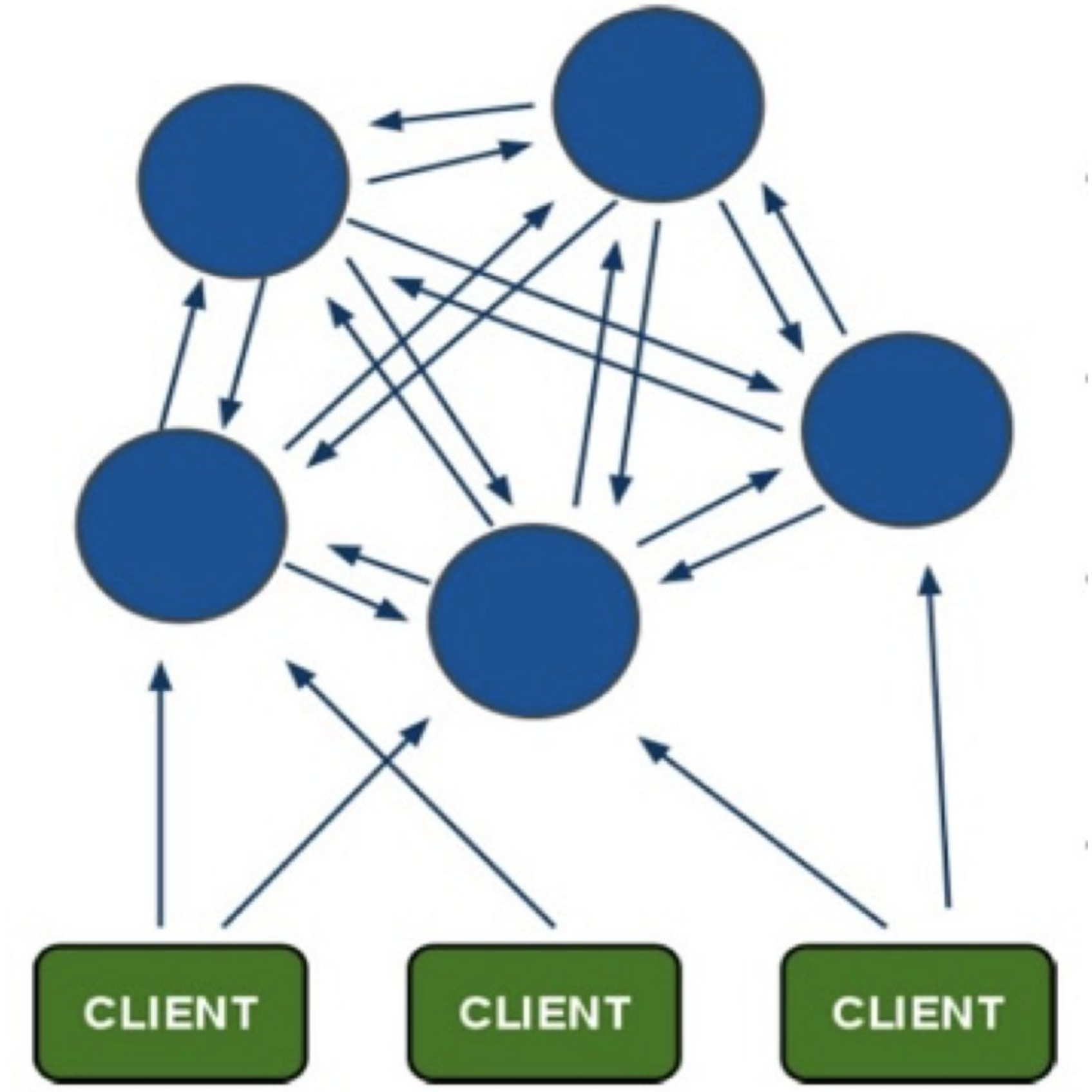
•Redis Sentinel



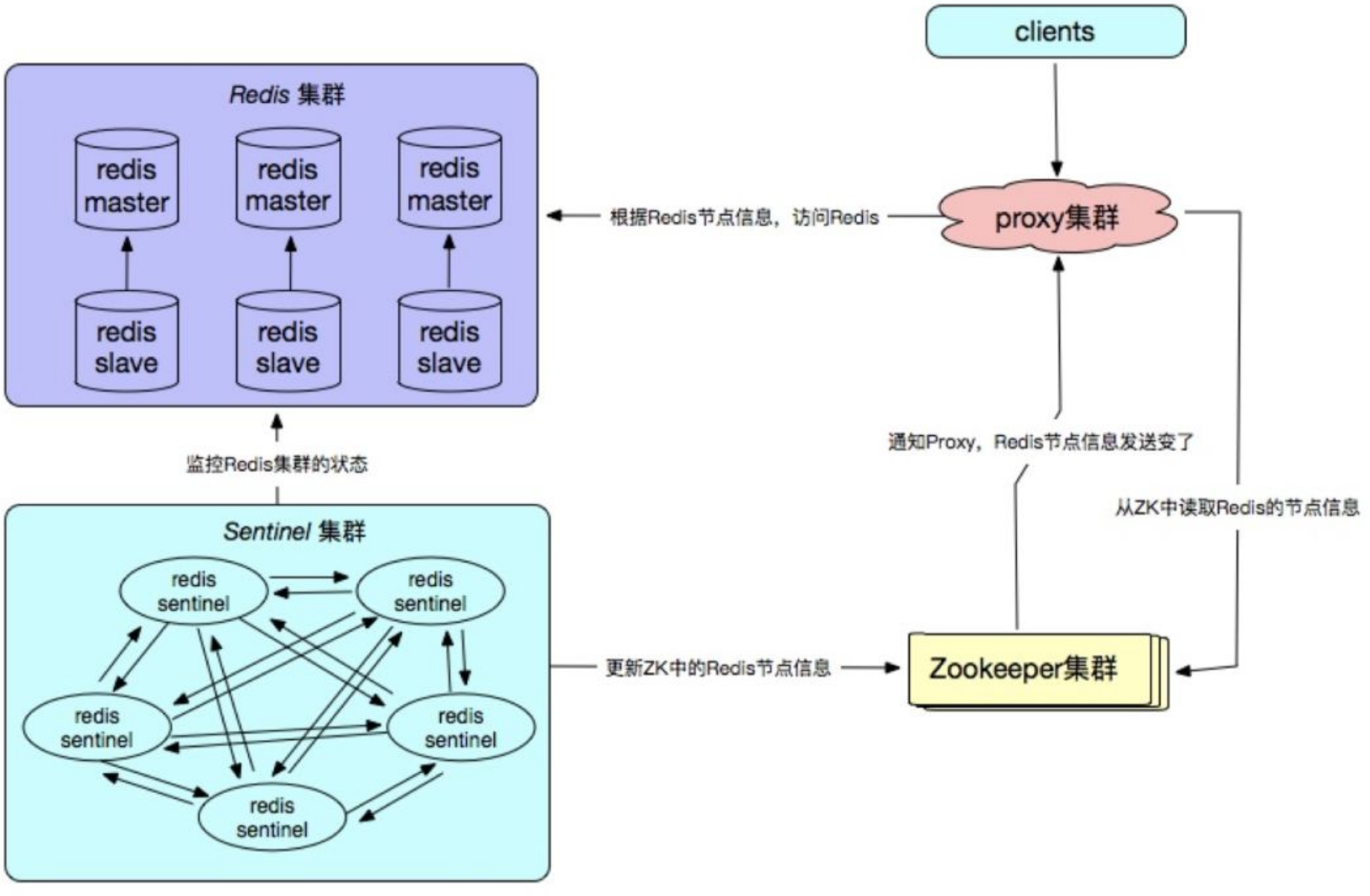
•Twemproxy + Redis Sentinel + Redis



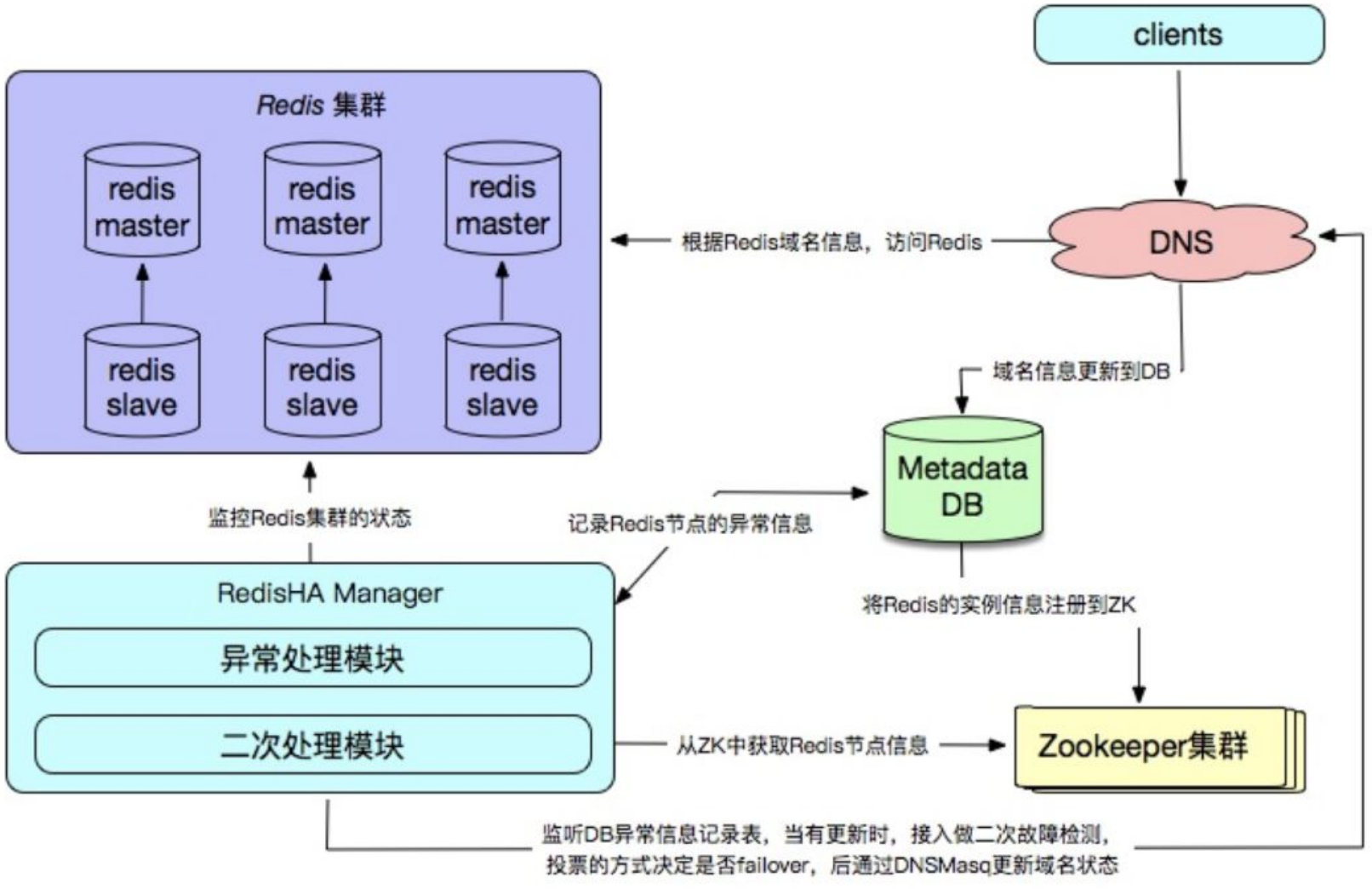
•Redis Cluster



•Redis Sentinel + Proxy + Zookeeper + Redis



•Zookeeper + MySQL + Redis + DNS



Redis数据库在向着自动化运维的方向发展的过程中，面临的最大的挑战是什么？如何克服？

如果不强调“最大的”话，我知道有不少的挑战，哈哈。

我想最大的挑战应该是“智能化”吧。现在业界都在追捧DevOps、AIOps，那么在Redis的自动化运维过程中，也需要学习行业的先进思想，结合部门自身的实际稳扎稳打，逐一突破。

首先在智能化实现之前，我们要尽力实现下面的一些需求：

1）数据库运维自动化平台的建设，为RD和DBA提供较全面的自助服务和数据库管理功能

2）工单事件关联任务系统，一键完成自动安装部署，添加产品线和报警

3）海量报警智能分类（比如按产品线或按DBA分类），实现部分报警故障自愈 （比如：从库readonly设置、内存使用率超过95%自动扩容）

4）基于队列分布式批量部署，可横向扩展，任务异步调度，满足大规模部署、扩容的需求

5）日志实时展示，监控数据实时采集，图形化多维度展示，满足可视化的需求

6）RedisHA支持秒级响应，实现故障无缝切换，满足高可用的需求

7）缓存弹性扩缩容（利用私有云和公有云，结合Docker容器化技术实现），满足对热点的快速应对

8）内部开发了各种通用的管理和运维脚本，化繁为简，提高工作效率

9）根据历史/晚高峰资源的性能指标设置水位线，和报警阈值，提供决策支持

10）在Redis集群、容灾、异地多活（跨数据中心数据同步）、微服务等等方面还需要花很大的力气去建设，目前依旧比较欠缺

智能化的实现还需要持续的投入和迭代。

近几年随着大数据时代的到来，NoSQL数据库在处理海量数据上表现出越来越多的优势，请问您如何看待数据库的未来，会朝着什么样的方向发展?

是的，这几年从运维的角度看，明显能感觉到数据体量上的膨胀，一个实例动不动就几十G，一个集群动不动几百G、几T，甚至更多。以Redis为代表的NoSQL数据库在处理这方面的表现还是令人非常满意的。它的高性能表现得淋漓尽致，从微博的业务来看，Redis实例每天承载着千亿级的写QPS、万亿级的读QPS，数据量TB级，确实没有Redis，且不说能不能用关系型数据库存储，单是硬件成本就几何倍增长了吧。

未来，随着硬件成本的降低，硬件性能优化的极致，比如PCIE、25GE网络的升级，一定是软硬结合，会出现更多的Redis的相关产品和服务。包括收费的Redis Enterprise，Cloud Redis等，也包括开源的Pika、TiDB等NewSQL。正如阿里云的同学所说，“数据库终极是九九归一 -- 量子数据库”，一起期待吧。

对于初学Redis的同学，您有什么好的学习方法和技巧给他们吗?

首先当然还是需要多看官方文档，多看看业界大牛的技术博客。其次可以买几本书对比着学习下，目前市面上的相关书籍也就4，5本的样子。然后主要的还是要多动手，实践出真知，在使用的过程中，还是要结合具体的业务场景，灵活使用。有能力的时候，看看源码，了解底层的实现原理。最后，多思考多总结，好记性不如烂笔头，每一次踩坑都是一次成长，遇到解决不了的问题的时候多向业界大牛们请教学习，平时多关注一些业界相关技术的最新动态。

您如何看待Redis的未来？从技术和非技术的多个维度，您如何看待Redis的发展方向

纵观Redis的发展，无独有偶的与互联网的发展浪潮紧紧相随，从web1.0到web2.0的转变，从博客、贴吧、论坛到社交媒体，从文本到图文再到短视频的兴起，从互联网到移动互联网，从3G到4G到即将普及的5G，从异军突起的新兴产业，如：团购、电商、外卖、旅游、游戏、互联网金融和证券、出行和直播等等，商业模式在不断的改变，不断的在刷新人们的生活方式。

但是在这些变化的背后，不变的是Redis作为基础服务为企业的高可用架构保驾护航，变化的是Redis的使用案例越来越丰富、服务体验越来越好。在Redis的发展过程中，由于企业需求的多样化，对Redis的要求越来越高，许多场景Redis的功能显得相形见绌，因此出现了诸如Codis、Pika、CounterService、ApsaraCache、CacheCloud、smartClient（Redisson）等产品，它们是对Redis的有益补充。

随着4.0版本module功能的推出，使得Redis拥有更多的想象和发展空间，在全文索引（RediSearch），AI（Redis-ML）、云计算、大数据、物联网、人工智能、BlockChain等领域，模块化的融合能力将极大的丰富Redis的功能，从而为企业构建更为多元化、立体化的数据库使用方式。

在使用Redis的过程中，还需要保持关注官方的动态，多看看github上的changelog和issue，因为随着Redis的用户群体的增多，使用场景的复杂化，Redis自身隐藏的问题或瓶颈也会暴露出来，这样有助于我们避免踩一些不必要的坑，及早规避一些风险。

不仅仅是Redis，学习其他的数据库也是一样的。另外，我们在关注Redis本身以外，还需要关注一些其他的Redis的替代产品（比如：SSDB、Pika、Codis、ApsaraCache、TiDB等），Redis的周边工具（如：redis-migrate-tool、redis-faina、redis-audit、redis-rdb-tools、RedisMyAdmin等）、中间件(如：twemproxy、corvus、redis-cerberus)等。如果你想要从事Redis的开发，

那么你可能需要关注的更多，比如各种锁的实现。

跨界与融合、机遇与挑战、个人与企业、现在与未来。让各行业、企业，以及每一个向未来而努力的人，听见时代最前沿的声音，见证成长！

推荐阅读：

遇见未来|DB舞台谁是王者之PostgreSQL专访

遇见未来 | 软件定义数据中心：人类文明运行在软件之上

遇见未来 | 超融合如何兼顾企业的“敏态”和“稳态”的业务需求

关注公众号：数据和云（OraNews）回复关键字获取

‘2017DTC’，2017DTC大会PPT

‘DBALIFE’，“DBA的一天”海报

‘DBA04’，DBA手记4经典篇章电子书

‘RACV1’, RAC系列课程视频及ppt

‘122ARCH’，Oracle 12.2体系结构图

‘2017OOW’，Oracle OpenWorld资料

‘PRELECTION’，大讲堂讲师课程资料