Redis服务端实现 & HA

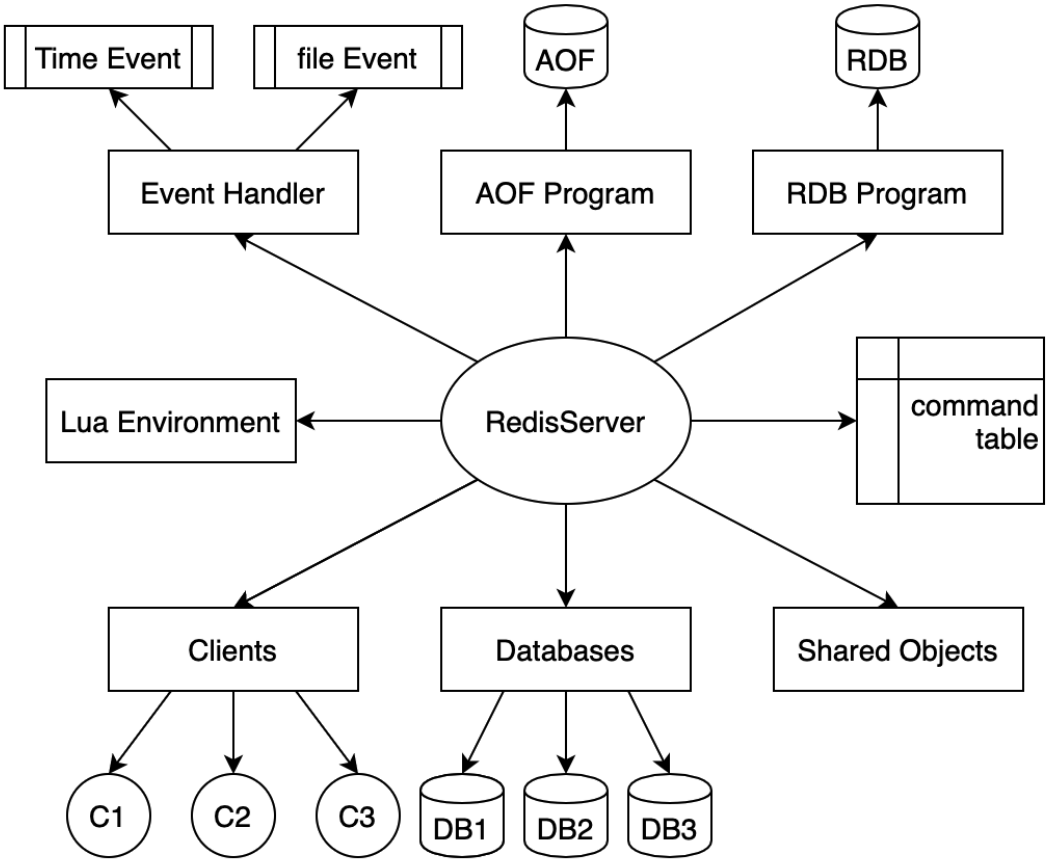
> 阅读本文，您将了解：
> 1. Redis 服务端的启动与执行流程
> 2. Redis 关键技术
> 3. 公司Redis相关架构

Redis服务端实现

Redis服务端启动流程

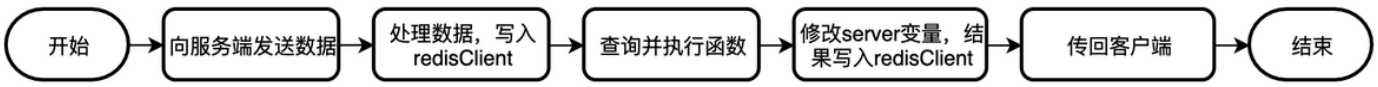
- 初始化服务器全局状态（数据库、命令表、网络链接信息、客户端信息……`redisServer`）
- 载入配置文件（载入用户配置文件，修改默认配置）
- 创建 `daemon` 进程（程序将创建 `daemon` 进程来运行 Redis）
- 初始化服务器功能模块（为 `redisServer` 变量的数据结构子属性分配内存，并初始化数据结构）
- 载入数据（将持久化的数据写入服务进程）
- 开始事件循环

初始化完成之后，服务器和各个模块之间的关系：



Redis服务的执行流程

一个简单的命令执行流程：



执行流程：

- 客户端把用户输入的命令转化为协议，发向服务器
- 客户端与服务器的Socket因为客户端写入变得可读，服务器读取协议，保存到输入缓冲区。
- 通过命令表找个函数，至此，函数、参数、参数个数集齐
- 相关检查（函数是否存在，函数参数是否一致，身份验证，内存检查……）
- 调用函数
- 修改全局状态的redisServer变量，返回值保存到客户端redisClient结构的回复缓冲区中，关联写事件。
- 当客户端的写事件就绪时，将回复缓存中的命令结果传回给客户端。

事件循环

整个redis服务在启动之后会陷入一个巨大的while循环，会调用 `aeMain` 函数陷入 `aeEventLoop` 循环中，等待外部事件的发生：

```
int main(int argc, char **argv) {
    ...
    aeMain(server.el);
}
```

`aeMain` 函数其实就是一个封装的 `while` 循环，循环中的代码会一直运行直到 `eventLoop` 的 `stop` 被设置为 `true`：

```
void aeMain(aeEventLoop *eventLoop) {
    eventLoop->stop = 0;
    while (!eventLoop->stop) {
        if (eventLoop->beforesleep != NULL)
            eventLoop->beforesleep(eventLoop);
        aeProcessEvents(eventLoop, AE_ALL_EVENTS);
    }
}
```

它会不停尝试调用 `aeProcessEvents` 对可能存在的多种事件进行处理，而 `aeProcessEvents` 就是实际用于处理事件的函数。
整个方法大体由两部分代码组成，一部分处理文件事件，另一部分处理时间事件。

整个事件处理器程序可以用以下伪代码描述：

```
def process_event():
    # 获取执行时间最接近现在的一个时间事件
    te = get_nearest_time_event(server.time_event_linked_list)
    # 检查该事件的执行时间和现在时间之差
    # 如果值 <= 0 ，那么说明至少有一个时间事件已到达
    # 如果值 > 0 ，那么说明目前没有任何时间事件到达
    nearest_te_remaind_ms = te.when - now_in_ms()
    if nearest_te_remaind_ms <= 0:
        # 如果有时间事件已经到达
        # 那么调用不阻塞的文件事件等待函数
        poll(timeout=None)
    else:
        # 如果时间事件还没到达
```

```
        # 那么阻塞的最大时间不超过 te 的到达时间
        poll(timeout=nearest_te_remaind_ms)
    # 处理已就绪文件事件
    process_file_events()
    # 处理已到达时间事件
    process_time_event()
```

文件事件

Redis基于Reactor模式开发了自己的网络事件处理器，也就是文件事件处理器。文件事件处理器使用IO多路复用技术，同时监听多个套接字，并为套接字关联不同的事件处理函数。当套接字的可读或者可写事件触发时，就会调用相应的事件处理函数。
Redis 将这类因为对套接字进行多路复用而产生的事件称为文件事件（`file event`），文件事件可以分为读事件和写事件两类。

在一般情况下，`aeProcessEvents` 都会先**计算最近的时间事件发生所需要等待的时间**，然后调用 `aeApiPoll` 方法在这段时间中等待事件的发生，在这段时间中如果发生了文件事件，就会优先处理文件事件，否则就会一直等待，直到最近的时间事件需要触发：

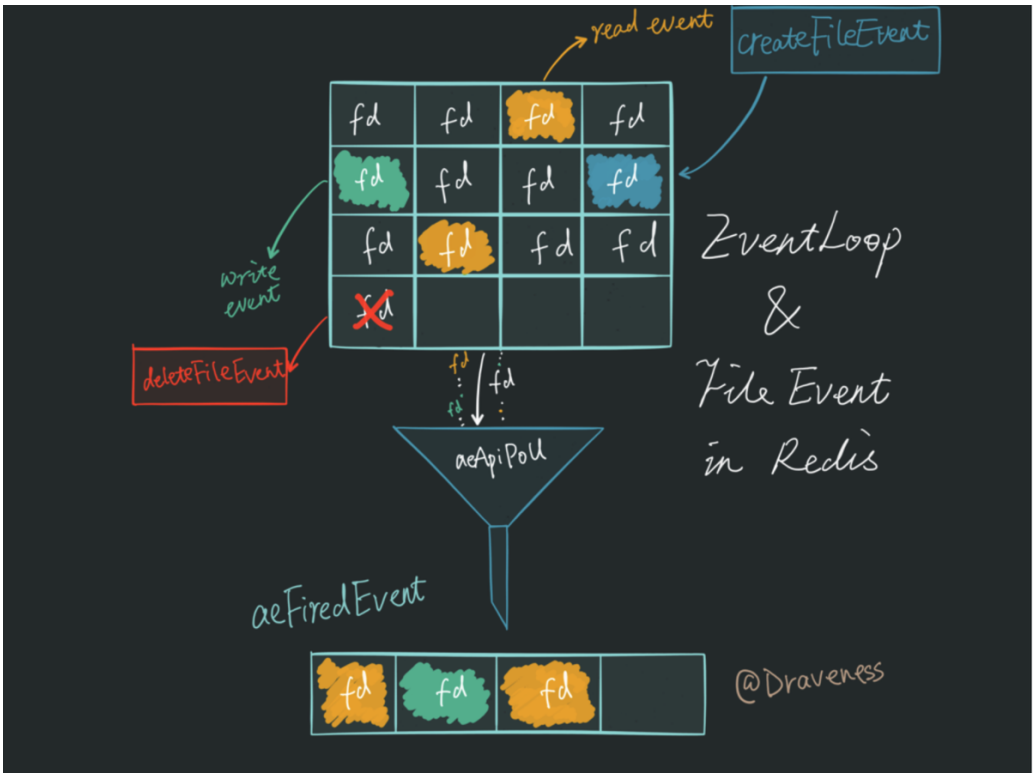
```
numevents = aeApiPoll(eventLoop, tvp);
for (j = 0; j < numevents; j++) {
    aeFileEvent *fe = &eventLoop->events[eventLoop->fired[j].fd];
    int mask = eventLoop->fired[j].mask;
    int fd = eventLoop->fired[j].fd;
    int rfired = 0;

    if (fe->mask & mask & AE_READABLE) {
        rfired = 1;
        fe->rfileProc(eventLoop,fd,fe->clientData,mask);
    }
    if (fe->mask & mask & AE_WRITABLE) {
        if (!rfired || fe->wfileProc != fe->rfileProc)
            fe->wfileProc(eventLoop,fd,fe->clientData,mask);
    }
    processed++;
}
```

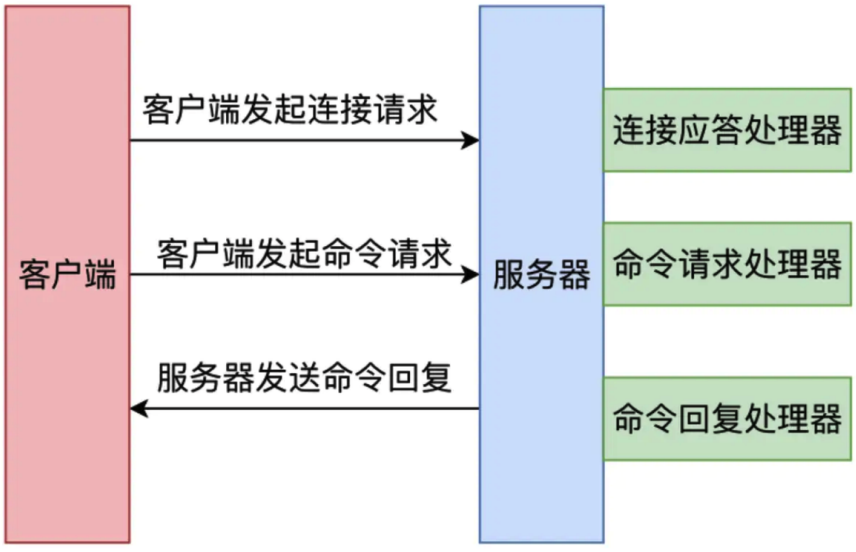
文件事件如果绑定了对应的读/写事件，就会执行对应的对应的代码。

文件事件的处理

整个 I/O 多路复用模块在事件循环看来就是一个输入事件、输出 `aeFiredEvent` 数组的一个黑箱：



一次 Redis 客户端与服务器进行连接并且发送命令的过程



时间事件

记录那些要在指定时间点运行的事件，多个时间事件以无序链表的形式保存在服务器状态中。

每个时间事件主要由三个属性组成：

- `when`：以毫秒格式的 UNIX 时间戳为单位，记录了应该在什么时间点执行事件处理函数。

- `timeProc`：事件处理函数。
- `next`：指向下一个时间事件，形成链表。

时间事件的处理在 `processTimeEvents` 中进行，我们会分三部分分析这个方法的实现：

调整系统时间的判断

```
static int processTimeEvents(aeEventLoop *eventLoop) {
    int processed = 0;
    aeTimeEvent *te, *prev;
    long long maxId;
    time_t now = time(NULL);
    //如果发现了系统时间被改变（小于上次 processTimeEvents 函数执行的开始时间），就会强制所有时间事件尽早执行
    if (now < eventLoop->lastTime) {
        te = eventLoop->timeEventHead;
        while(te) {
            te->when_sec = 0;
            te = te->next;
        }
    }
    eventLoop->lastTime = now;
```

删除标记时间事件

```
    prev = NULL;
    te = eventLoop->timeEventHead;
    maxId = eventLoop->timeEventNextId-1;
    while(te) {
        long now_sec, now_ms;
        long long id;
        //Redis 处理时间事件时，不会在当前循环中直接移除不再需要执行的事件，而是会在当前循环中将时间事件的 id 设置为 AE_DELETED_EVENT_ID，然后再下一个循环中删除，并执行绑定的 finalizerProc
        if (te->id == AE_DELETED_EVENT_ID) {
            aeTimeEvent *next = te->next;
            if (prev == NULL)
                eventLoop->timeEventHead = te->next;
            else
```

```
                prev->next = te->next;
            if (te->finalizerProc)
                te->finalizerProc(eventLoop, te->clientData);
            zfree(te);
            te = next;
            continue;
        }
```

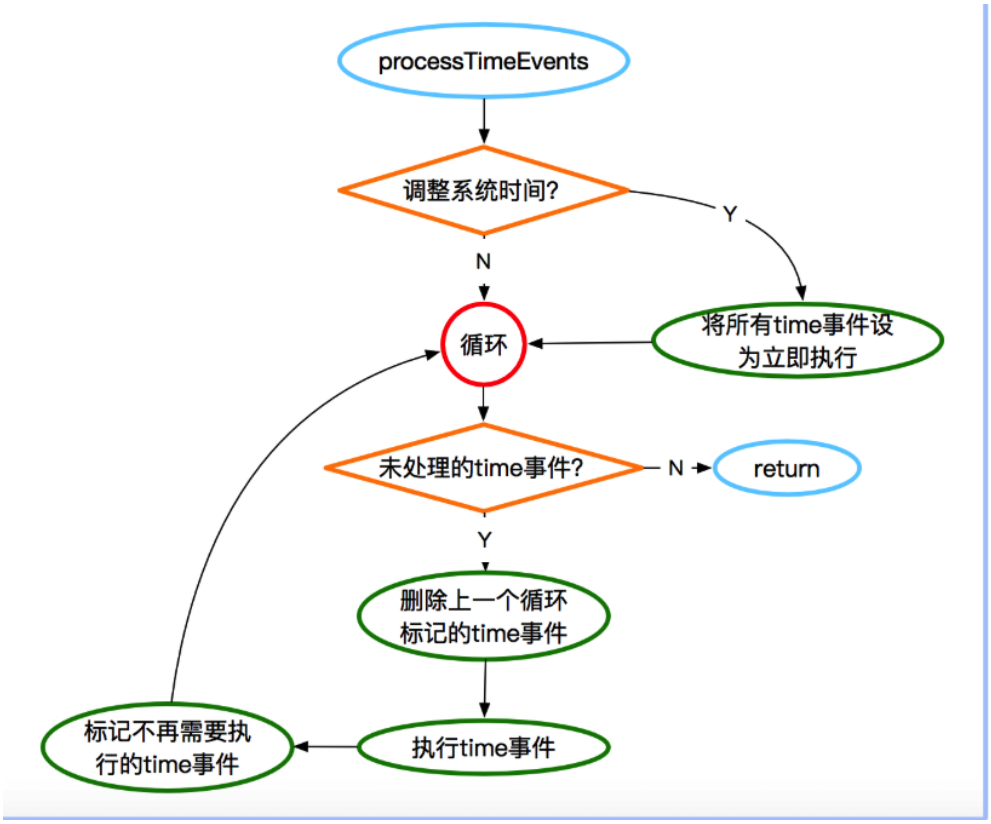
执行并标记

```
        aeGetTime(&now_sec, &now_ms);
        if (now_sec > te->when_sec ||
            (now_sec == te->when_sec && now_ms >= te->when_ms))
        {
            int retval;
            id = te->id;
            retval = te->timeProc(eventLoop, id, te->clientData);
            processed++;
            if (retval != AE_NOMORE) {
                aeAddMillisecondsToNow(retval,&te->when_sec,&te->when_ms);
            } else {
                te->id = AE_DELETED_EVENT_ID;
            }
        }
        prev = te;
        te = te->next;
    }
    return processed;
}
```

在移除不需要执行的时间事件之后，我们就开始通过比较时间来判断是否需要调用 `timeProc` 函数，`timeProc` 函数的返回值 `retval` 为时间事件执行的时间间隔：

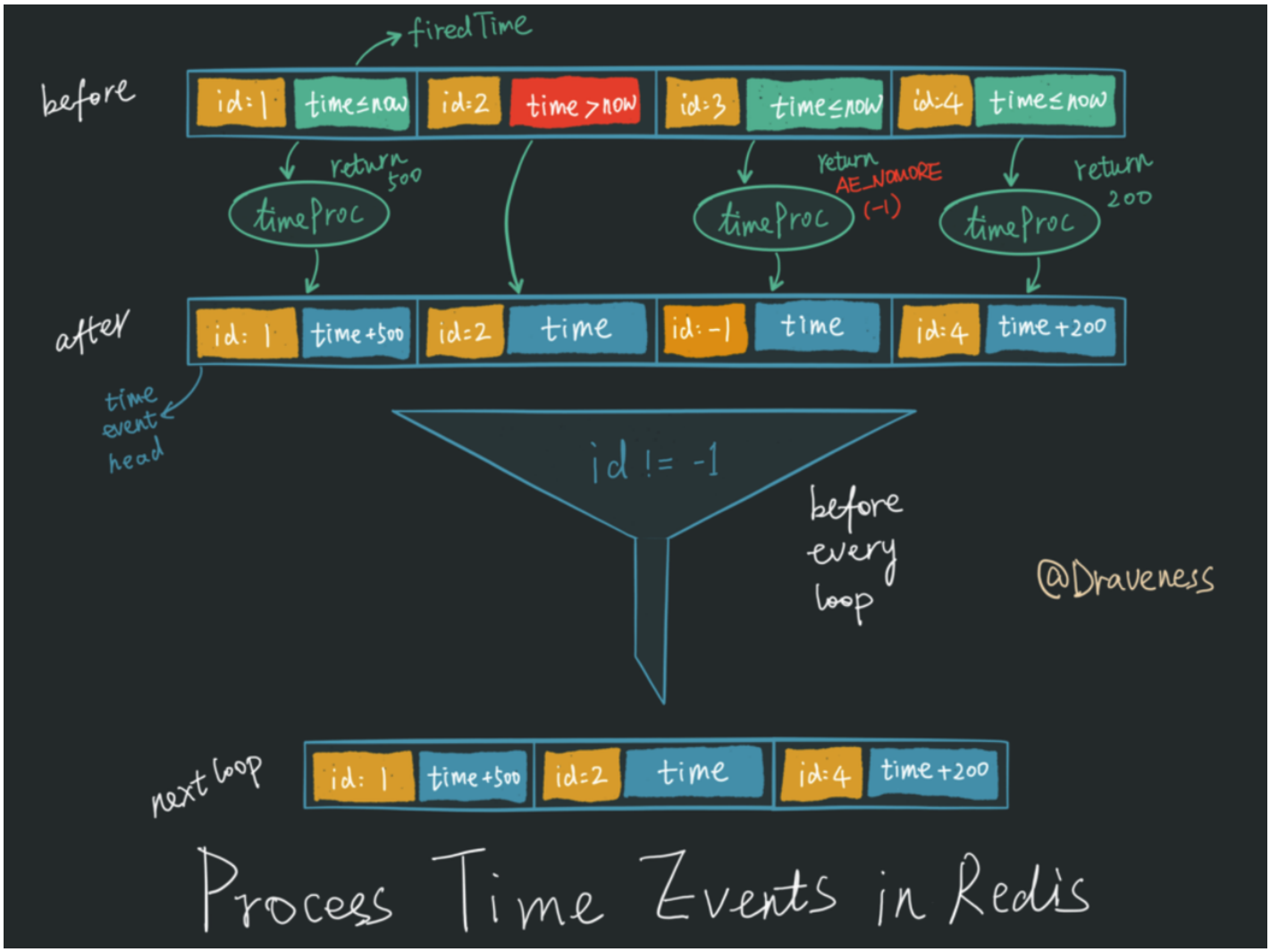
- `retval == AE_NOMORE`：将时间事件的 `id` 设置为 `AE_DELETED_EVENT_ID`，等待下次 `aeProcessEvents` 执行时将事件清除；
- `retval != AE_NOMORE`：修改当前时间事件的执行时间并重复利用当前的时间事件；

时间事件的执行流程



时间事件的处理

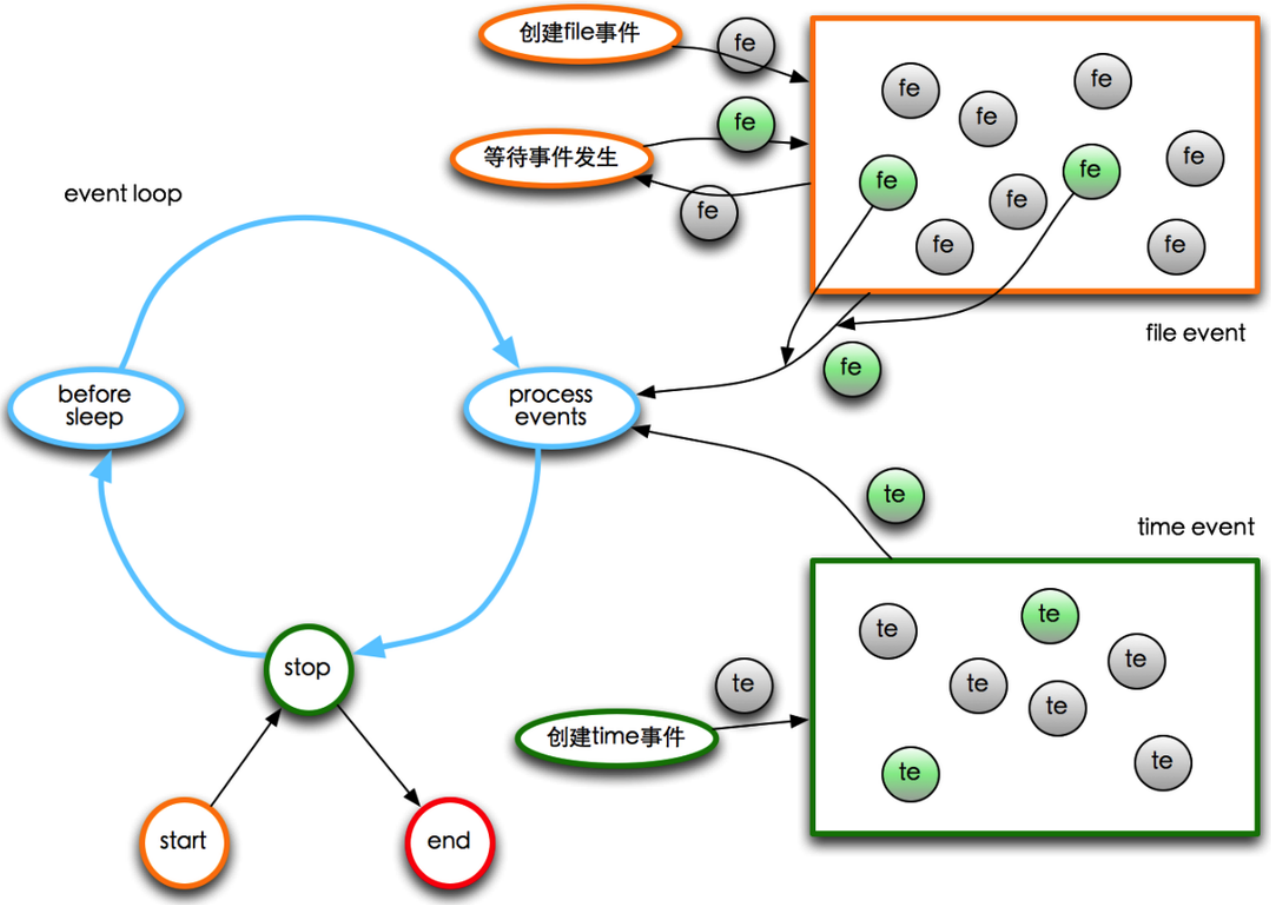
时间事件的处理相比文件事件就容易多了，每次 `processTimeEvents` 方法调用时都会对整个 `timeEventHead` 数组进行遍历：



遍历的过程中会将时间的触发时间与当前时间比较，然后执行时间对应的 `timeProc`，并根据 `timeProc` 的返回值修改当前事件的参数，并在下一个循环的遍历中移除不再执行的时间事件。

总结

事件的处理：



beforesleep - 进入event loop前执行：

- cluster集群状态检查，ok->fail、fail->ok
- 处理被block住的的client，如一些阻塞请求BLPOP等
- 将AOF buffer持久化到AOF文件

将这个事件处理函数置于一个循环中， 加上初始化和清理函数， 这就构成了 Redis 服务器的主函数调用：

```
def redis_main():
    # 初始化服务器
    init_server()
    # 一直处理事件，直到服务器关闭为止
    while server_is_not_shutdown():
        process_event()
    # 清理服务器
    clean_server()
```

Redis HA相关

逐出&过期

逐出

当执行write但内存达到上限时，强制将一些key删除
key集合

- allkeys - 所有key
- volatile - 设置了过期的key

缓存淘汰机制：

- LRU - 最久未被使用
- random - 随机
- ttl - 最快过期的
- Lfu - 最近最少使用

特点

- 不是精准算法，而是抽样比对
- 每次写入操作前判断
- 逐出是阻塞请求的

过期

当某个key到达了ttl时间，认为该key已经失效
两种方式

- 惰性删除 - 读、写操作前判断ttl，如过期则删除
- 定期删除 - 在redis定时事件中随机抽取部分key判断ttl

特点

- 并不一定是按设置时间准时地过期
- 定期删除的时候会判断过期比例，达到阈值才退出

建议

打撒key的过期时间，避免大量key在同一时间点过期。

持久化

RDB持久化

在指定的时间间隔对数据进行快照存储。

- 经过压缩的二进制格式
- fork子进程dump可能造成瞬间卡顿

AOF持久化

记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据。

- 保存所有修改数据库的命令
- 先写aof缓存，再同步到aof文件
- AOF重写，达到阈值时触发，减小文件大小

利用AOF文件灾备，可将数据恢复到最近3天任意小时粒度

混合持久化

这种持久化能够通过 AOF 重写操作创建出一个同时包含 RDB 数据和 AOF 数据的 AOF 文件， 其中 RDB 数据位于 AOF 文件的开头， 它们储存了服务器开始执行重写操作时的数据库状态： 至于那些在重写操作执行之后执行的 Redis 命令， 则会继续以 AOF 格式追加到 AOF 文件的末尾， 也即是 RDB 数据之后。

主从复制

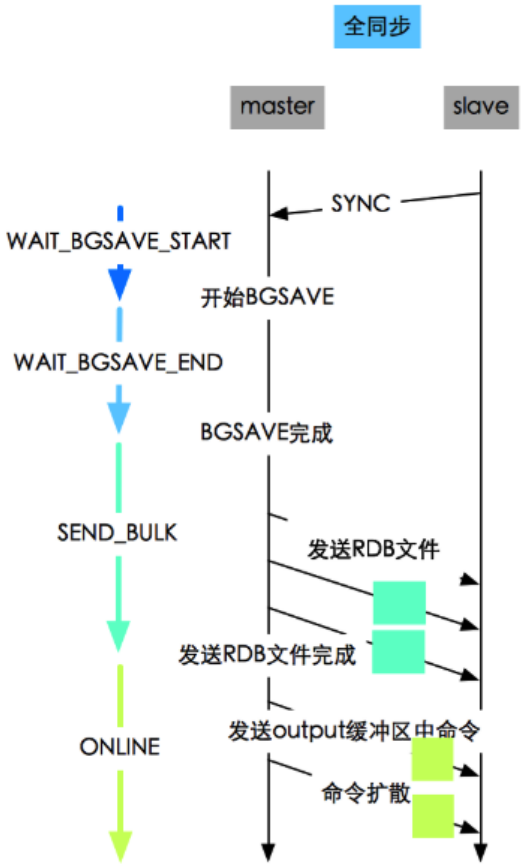
其中持久化侧重解决的是Redis数据的单机备份问题（从内存到硬盘的备份，硬盘到内存的恢复）；而主从复制则侧重解决数据的多机热备。此外，主从复制还可以实现负载均衡和故障恢复。
主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。

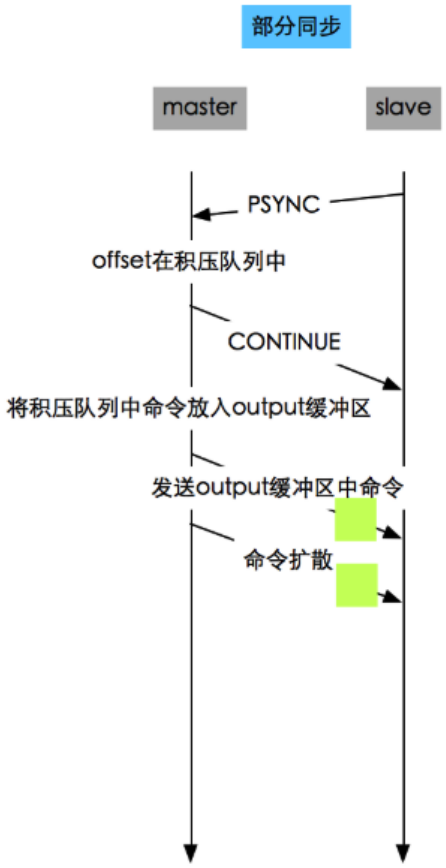
主要作用

- 数据冗余
- 故障恢复
- 负载均衡
- HA基石

同步方式

- 全量同步
  - 传递RDB文件&restore命令重建kv
  - 传递在RDB dump过程中的写入数据
- 部分同步
  - 根据offset传递积压缓存中的部分数据





Redis集群

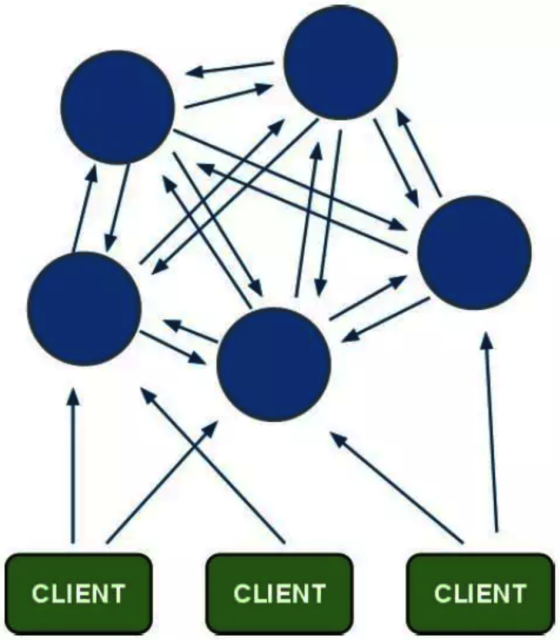
Redis Cluster，是Redis 3.0开始引入的分布式存储方案。

集群由多个节点(Node)组成，Redis的数据分布在这些节点中。集群中的节点分为主节点和从节点：只有主节点负责读写请求和集群信息的维护；从节点只进行主节点数据和状态信息的复制。

集群的作用

- 数据分区：数据分区(或称数据分片)是集群最核心的功能。
- 高可用：集群支持主从复制和主节点的自动故障转移；当任一节点发生故障时，集群仍然可以对外提供服务。

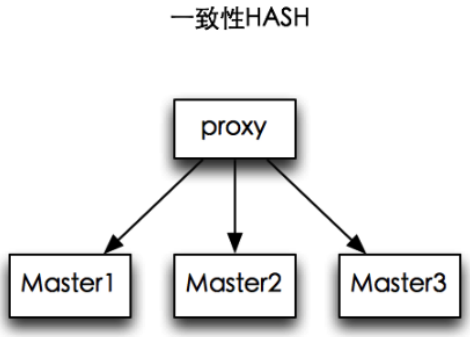
Redis集群模型图



节点之间采用Gossip协议进行通信，Gossip协议就是指节点彼此之间不断通信交换信息。当主从角色变化或新增节点，彼此通过ping/pong进行通信知道全部节点的最新状态并达到集群同步。

公司集群架构

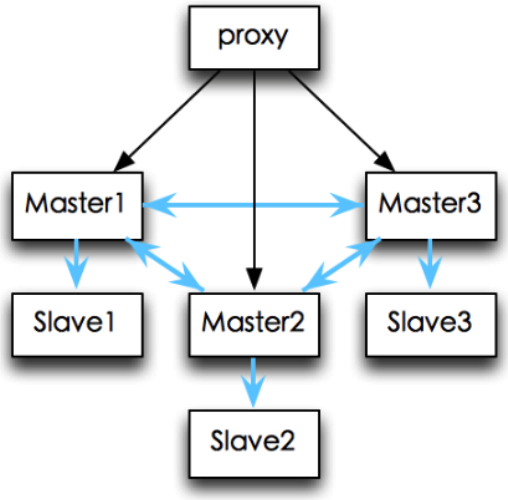
缓存集群架构



采用一致性hash的数据分布方式。proxy作为中心管理节点，路由分配相关命令。
实例宕机、加节点容易造成数据丢失。

存储集群架构

redis cluster（基于gossip协议）



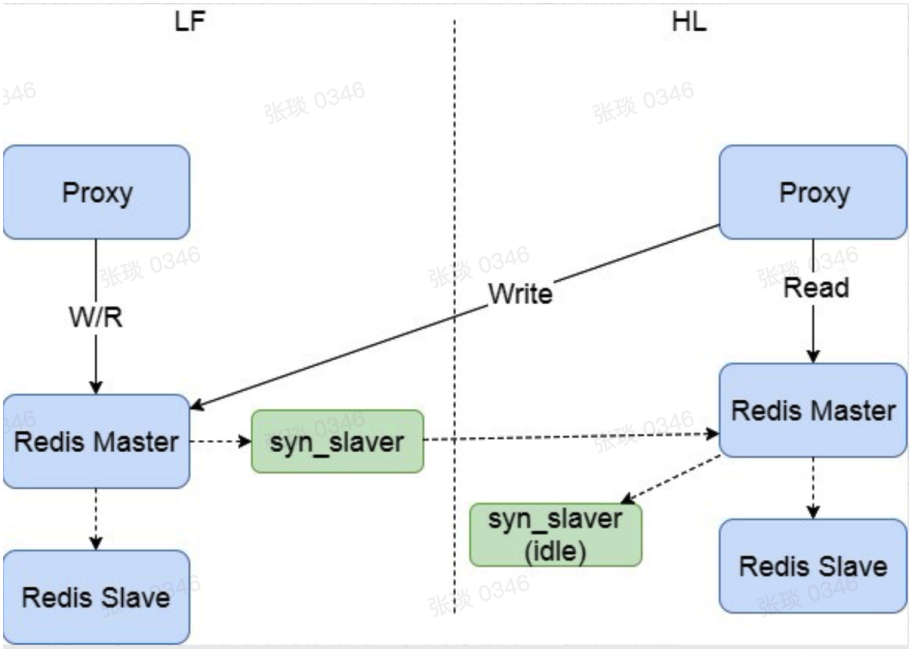
采用redis cluster原生的集群模式，每个主会挂一个从。两两通信维护拓扑信息。
Proxy 作用：

- 对外提供一个统一的client
- proxy层做一些策略（大key，热key，双机房删除）

有节点数量上限，达不到数据一致。

双机房架构

目前新申请的redis 集群都是rc(redis cluster)架构，基于同步组件支持双机房架构，所有写都到LF机房，基于同步组件将LF数据同步到HL；



- `syn_slaver`：数据同步组件，作为redis的伪从，将数据推到对端机房，遇到网络故障（无法写入时）优先写内存缓冲区，内存缓冲区写满后落磁盘；

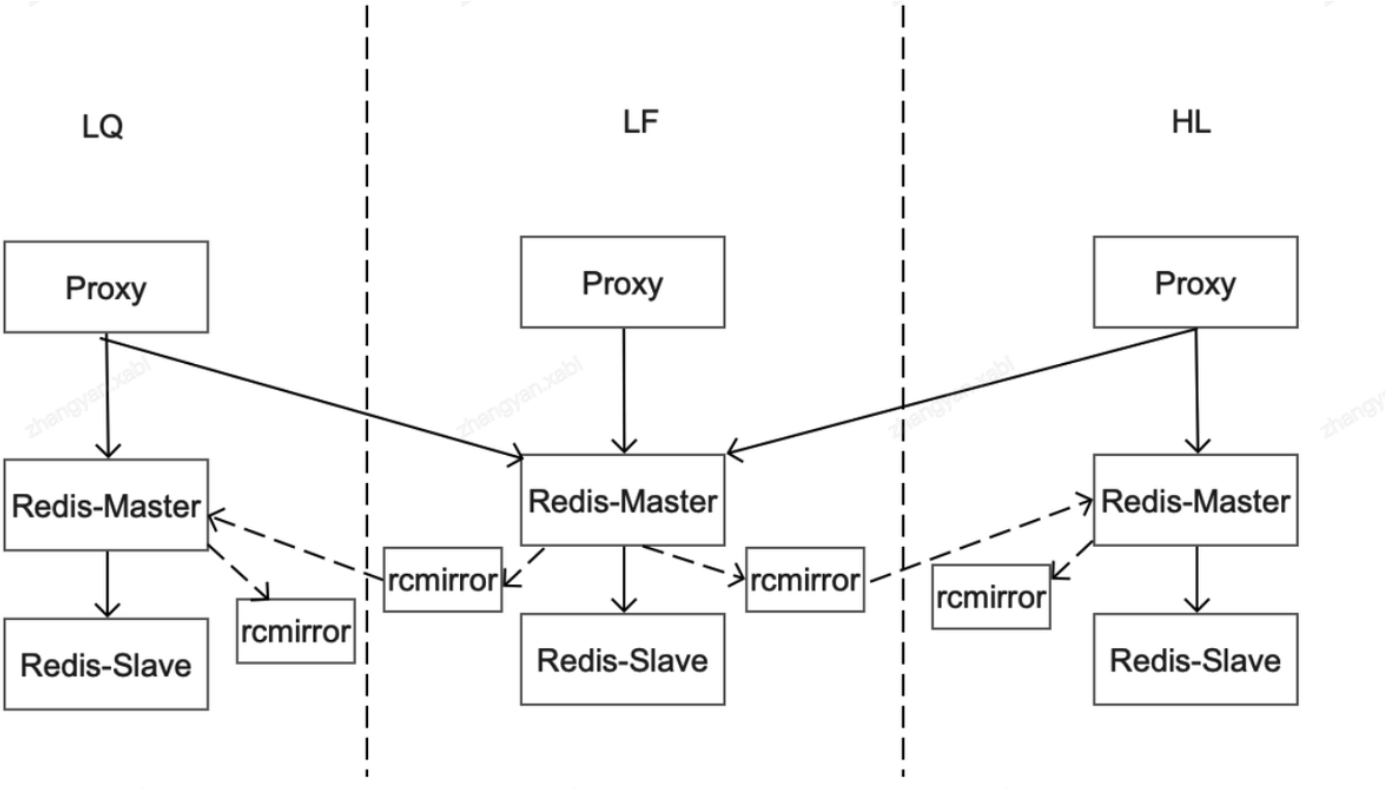
- `syn_slaver(idle)`：从机房的数据同步组件，平时不做数据同步；机房故障流量切从恢复后做数据同步；任意时刻只会有单个方向的同步组件work；

缺点：

- HL写请求有延迟
- 数据一致性问题

多机房架构

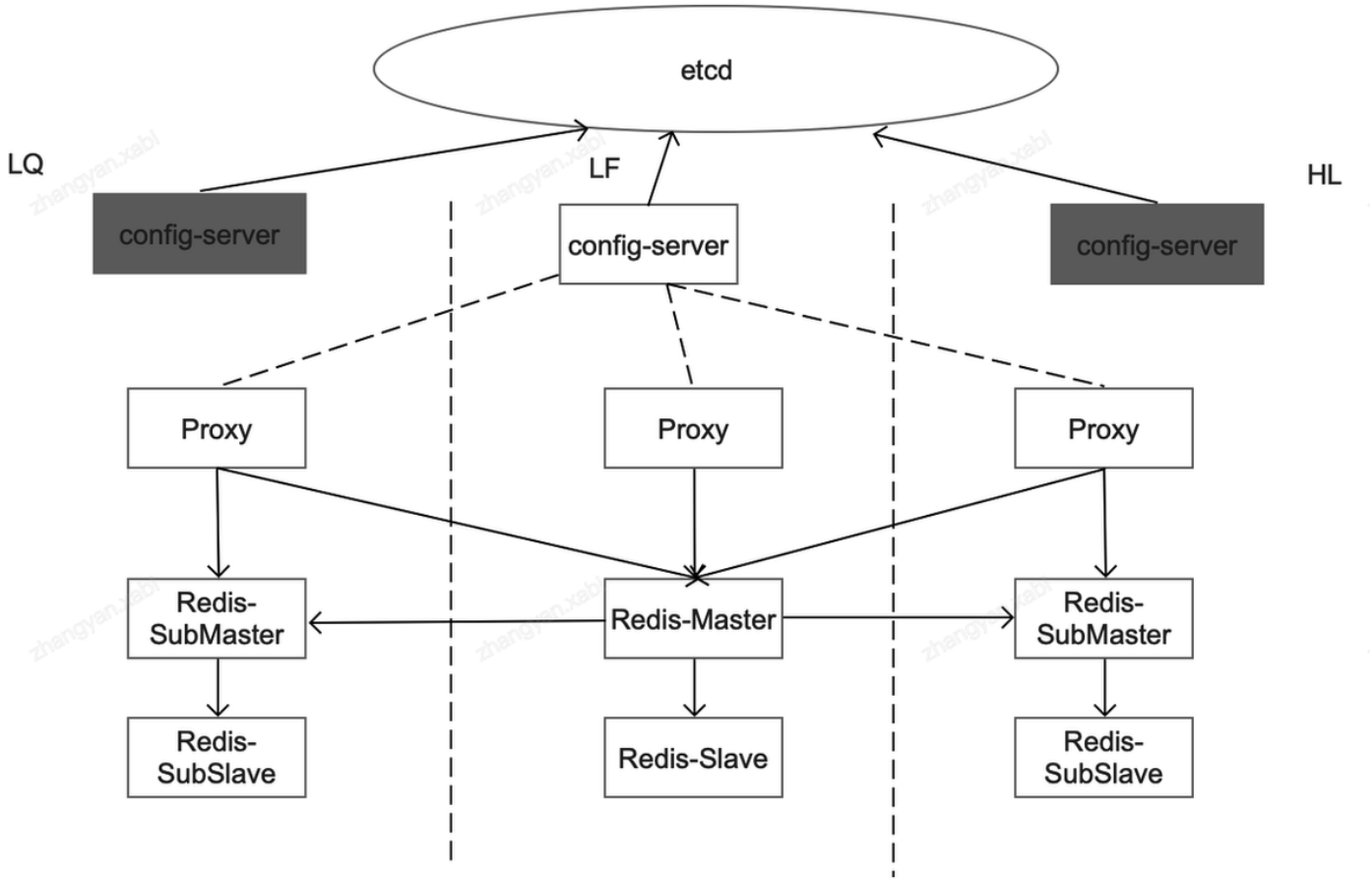
Redis Cluster



- Proxy为代理，业务请求的入口，默认LF为主机房，主机房读写本机房Redis，从机房写主机房Redis，读本机房Redis
- Redis使用去中心化的Redis Cluster架构，由集群中的Redis维护集群拓扑，每个slot一主一从
- rcmirror是自研的伪从，接收master的同步数据并发送到对端的master，三个机房都有部署，通过一个key判断主机房，当自身是主机房的rcmirror时进行同步

Alchemy

alchemy是天然多机房同步的模型，在申请服务时直接申请alchemy三机房即可完成三机房的创建与数据同步。



- alchemy架构中集群的拓扑信息由config-server维护，并存储在etcd中
- proxy为代理，业务请求的入口，通过config-server得知redis拓扑，默认LF为主机房，主机房读写本机房Redis，从机房写主机房Redis，读本机房Redis
- Redis是主从架构，从机房的主Redis是通过成为主机房主Redis的slave来完成数据同步

HA的实现方式

Sentinel

Sentinel是redis自带的一种实现HA的方式。Redis 的 Sentinel 系统用于管理多个 Redis 服务器（`instance`）， 该系统执行以下三个任务：

- 监控（`Monitoring`）： Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
- 提醒（`Notification`）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
- 自动故障迁移（`Automatic failover`）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作， 它会将失效主服务器的其中一个从服务器升级为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。

Keepalived

keepalived是集群管理中保证集群高可用的一个服务软件，用来防止单点故障。具体参阅：https://www.jianshu.com/p/e433978892b5

基本设计思路：

当 Master 与 Slave 均运作正常时, Master负责服务，Slave负责Standby；

当 Master 挂掉，Slave 正常时, Slave接管服务，有写权限，同时关闭主从复制功能；

当 Master 恢复正常，则从Slave同步数据，同步数据之后关闭主从复制功能，恢复Master身份，同时Slave等待Master同步数据完成之后，恢复Slave身份。

然后依次循环。

参考文档

事件循环: https://draveness.me/redis-eventloop
https://redisbook.readthedocs.io/en/latest/internal/ae.html#id2
https://www.iteye.com/blog/olylakers-1287211
https://juejin.im/post/5d4c3a5df265da03934bcbe8

redis持久化: https://juejin.im/post/5b70dfcf518825610f1f5c16
https://blog.huangz.me/2017/redis-rdb-aof-mixed-persistence.html

逐出&过期: https://segmentfault.com/a/1190000005103635

redis主从复制: https://www.cnblogs.com/kismetv/p/9236731.html

redis集群: https://juejin.im/post/5b8fc5536fb9a05d2d01fb11#heading-16

HA实现方式: https://blog.csdn.net/fuyuwei2015/article/details/71106918

双机房架构: Cache服务同城双机房部署方案 【Cache服务用户手册】- Cache 双机房集群部署 【Cache服务用户手册】- Cache 双机房集群部署选型.bak Cache平台方案设计

多机房架构：https://doc.bytedance.net/docs/2707/3269/internal_dc/

其他：

https://spldeolin.com/posts/redis-client-server/

https://docs.google.com/presentation/d/1dzUT_SprawZUVanDm_eliIA1uHx_z1ghwAYECfnVszw/edit#slide=id.p3

https://docs.google.com/presentation/d/1dzUT_SprawZUVanDm_eliIA1uHx_z1ghwAYECfnVszw/edit#slide=id.g432e45af74_0_7

https://study.bytedance.net/course_play/95

https://study.bytedance.net/course_play/41