# redis笔记

## 1. SDS

SDS除了用来保存数据库中的字符串值之外，SDS还被用作缓冲区：AOF模块中的AOF缓冲区，以及客户端状态中的输入缓冲区

### 1.1 sds优势：

- **更快速的获取字符串长度**

  我们都知道Java的字符串有提供length方法，列表有提供size方法，我们可以直接获取大小。但是C却不一样，更偏向底层实现，所以没有直接的方法使用。这样就带来一个问题，如果我们想要获取某个数组的长度，就只能从头开始遍历，当遇到第一个'\0'则表示该数组结束。这样的速度太慢了，不能每次因为要获取长度就变量数组。所以设计了SDS数据结构，在原来的字符数组外面增加总长度，和已用长度，这样每次直接获取已用长度即可。复杂度为O(1)。

- **数据安全，不会截断**

  如果传统字符串保存图片，视频等二进制文件，中间可能出现'\0'，如果按照原来的逻辑，会造成数据丢失。所以可以用已用长度来表示是否字符数组已结束。

- **杜绝缓冲区溢出**

- **减少修改字符串长度所需要的的内存重分配次数**

```
1.空间预分配：小于1M，空间每次翻倍，大于1M，空间每次加1M，最大大小512M
2.惰性空间释放
3.初始大小为本身大小，不会多分配空间
```

### 1.2 编码方式

- embstr：将redisobject和数据对象sdshdr连续存放，在分配时只分配一次空间，适用于小字符串
- raw：将redisobject和数据对象sdshdr不连续存放，在分配时只分配两次空间

## 2. 双端链表

除了链表建之外，发布与订阅、慢查询、监视器等功能也用到了链表，Redis服务器本身还用链表保存多个客户端的状态信息，以及用链表来构建客户端输出缓冲区。

redis的链表是自带头尾指针的双端链表

## 3. 字典

dict 是 Redis 服务器中出现最为频繁的复合型数据结构，除了 hash 结构的数据会用到字典外，整个 Redis 数据库的所有 key 和 value 也组成了一个全局字典，还有带过期时间的 key 集合也是一个字典。zset 集合中存储 value 和 score 值的映射关系也是通过 dict 结构实现的。

```
struct RedisDb {
     dict* dict; // all keys  key=>value
     dict* expires; // all expired keys key=>long(timestamp)
     ...
}

struct zset {
     dict *dict; // all values  value=>score，zscore是直接从dict中获取分数的
     zskiplist *zsl;
}
```

## 字典内部结构

```
//哈希节点结构
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next;
} dictEntry;

//哈希表结构 该部分是理解字典的关键
typedef struct dictht {
    dictEntry **table;
    unsigned long size;
    unsigned long sizemask;
    unsigned long used;
} dictht;

//字典结构
typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2]; //在rehash过程中，依次两ht[1]中的桶位迁移到ht[2]，完成迁移后替换两个ht
    long rehashidx; /* rehashing not in progress if rehashidx == -1 */
    unsigned long iterators; /* number of iterators currently running */
} dict;
```

## 扩容缩容

- 扩容条件

  当服务器目前没有执行BGSAVE命令或者BGREWRITEAOF命令时，并且哈希表的负载因子大于等于1
  当服务器目前正在执行BGSAVE命令或则BGREWRITEAOF命令，并且哈希表的负载因子大于等于5

  > 负载因子的计算方式为：负载因子 = 哈希表已保存结点数量/哈希表大小(used/size)

> 执行BGSAVE命令或者BGREWRITEAOF命令的过程中，Redis需要创建当前服务器进程的子进程，而大多数操作系统都采用 写时复制（copy-on-write）来优化子进程的使用效率，所以在子进程存在期间，服务器会提高负载因子的阈值，从而避免在子进程存在期间进行哈希表扩展操作，避免不必要的内存写入操作，最大限度地节约内存。
> 参考文献：https://blog.csdn.net/qq_34556414/article/details/108399543

- 缩容条件

  元素个数低于数组长度的10%，切无需考虑是否在进行bgsave

## 渐进式rehash

hash过程耗时很长，而redis又是单线程处理的，如果在线程内同步rehash操作，会阻塞线程，使相应变慢。所以采用渐进式hash方式。

- 对桶位中元素进行写操作时，对整个桶进进行惰性迁移。
- 定时任务对未完成的迁移的桶位进行主动迁移

# 4.整数集合

整数集合是集合键底层实现之一，当一个集合只包含整数元素，且数据不多的时候，redis就会使用整数集合来作为底层实现。

```
typedef struct intset {

    // 编码方式
    uint32_t encoding;

    // 集合包含的元素数量
    uint32_t length;

    // 保存元素的数组
    int8_t contents[];
} intset;
```

## 升级

- 整数集合的encoding保存了数据元素的编码，当数组元素中的最大值大于阈值时，编码会进行升级。
- 他可以保存int_16t,int_32t,int_64t这三类编码的数组。
- 没有降级的过程

# 5. 压缩列表

当hash、list、zset结构中元素较少，且列表项要么是小整数值，要么是长度比较短的字符串时候，redis会使用压缩列表来做列表建的底层实现。

```
typedef struct zlentry {
    unsigned int prevrawlensize; /* encode前一个entry的字节长度*/
    unsigned int prevrawlen;     /* 前一个entry长度*/
    unsigned int lensize;        /* encode当前字节长度*/
    unsigned int len;            /* 实际长度 */
    unsigned int headersize;     /* prevrawlensize + lensize. */
    unsigned char encoding;      /* Set to ZIP_STR_* or ZIP_INT_* depending on
                                    the entry encoding. However for 4 bits
```

```
                                    immediate integers this can assume a range
                                    of values and must be range-checked. */
    unsigned char *p;             /* Pointer to the very start of the entry, that
                                    is, this points to prev-entry-len field. */
} zlentry;
```

## previous属性

prevlen属性以字节为单位，记录了压缩列表中前一个节点的长度，其长度可以是 1 字节或者 5 字节：

1. 如果前一节点的长度小于254字节，那么prevlen属性的长度为1字节， 前一节点的长度就保存在这一个字节里面。
2. 如果前一节点的长度大于等于254字节，那么prevlen属性的长度为5字节,第一字节会被设置为**0xFE**，之后的四个字节则用于保存前一节点的长度。

## 通过previous双向遍历

假设我们有一个指向当前节点起始地址的指针c，那我们获取前一个节点的起始地址p只需要：

```
p = c - current_entry.previous_entry_length
```

## 连锁更新

再思考一个问题，为什么prevlen的长度要么是1字节要么是5字节呢？为啥没有2字节、3字节、4字节这些中间态的长度呢？要解答这个问题就引出了一个关键问题：连锁更新问题。

试想这样一种增加节点的场景：

> 如果在压缩列表的头部增加一个新节点，并且长度大于254字节，所以其后面节点的prevlen必须是5字节，然而在增加新节点之前其prevlen是1字节，必须进行扩展，极端情况下如果一直都需要扩展那么将产生连锁反应。

理解了连锁更新问题，再来看看为什么要么1字节要么5字节的问题吧，如果是2-4字节那么可能产生连锁反应的概率就更大了，相反直接给到最大5字节会大大降低连锁更新的概率，所以笔者也认为这种内存的小小浪费也是值得的。

# 6.跳跃表

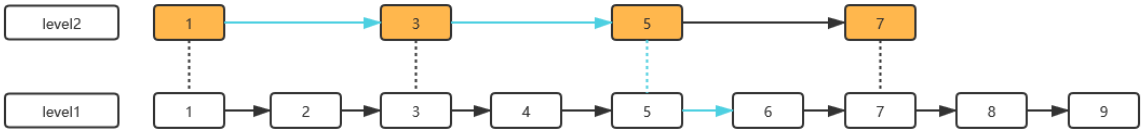
## 跳跃表出现的原因

假如我们要用某种数据结构来维护一组有序的int型数据的集合，并且希望这个数据结构在插入、删除、查找等操作上能够尽可能着快速，那么，你会用什么样的数据结构呢？

- 数组：查询可以使用二分法，通过数组下标直接定位，查询友好，但是写操作需要移动数组中的元素，复杂度O(N)
- 链表：查询只能遍历，复杂度O(N)，但是写操作友好，只需要改变指针指向即可。

跳跃表加快链表的遍历速度，使得写操作和查询操作都很快速。

## 跳跃表结构简介




如果没有跳跃表，在有序链表中要查找6这个元素，我们需要遍历的路径为:

```
1->2->3->4->5->6
```

而跳跃表，就是从有序链表中，抽取部分节点，作为它上一层的节点。在遍历的时候，从层往下层，从左往右遍历。在每一层中，找到最后一个比目标元素要小的节点，然后往下一层移动，继续找到比目标元素要小的最后一个节点，依次类推。如上图所示，在这样一个2层的跳跃表中，我们要查找6这个元素，我们需要遍历的路径为；

```
1->3->5->5->6
```

如果要在加快遍历速度，我们则需要再增加跳跃表的层数，也需要更多的存储空间，**这是一种典型的空间换时间的例子。**这也是为什么当zset元素比较少的时候，我们会认为跳跃表的空间增加带来的查询效率提升并不等价，会采用压缩列表作为zset数据结构的原因。

## redis中的跳跃表

```
typedef struct zskiplistNode {

    // member 对象
    robj *obj;

    // 分值
    double score;

    // 后退指针
    struct zskiplistNode *backward;

    // 层
    struct zskiplistLevel {

        // 前进指针
        struct zskiplistNode *forward;

        // 这个层跨越的节点数量
        unsigned int span;

    } level[];

} zskiplistNode;
```

### 插入操作

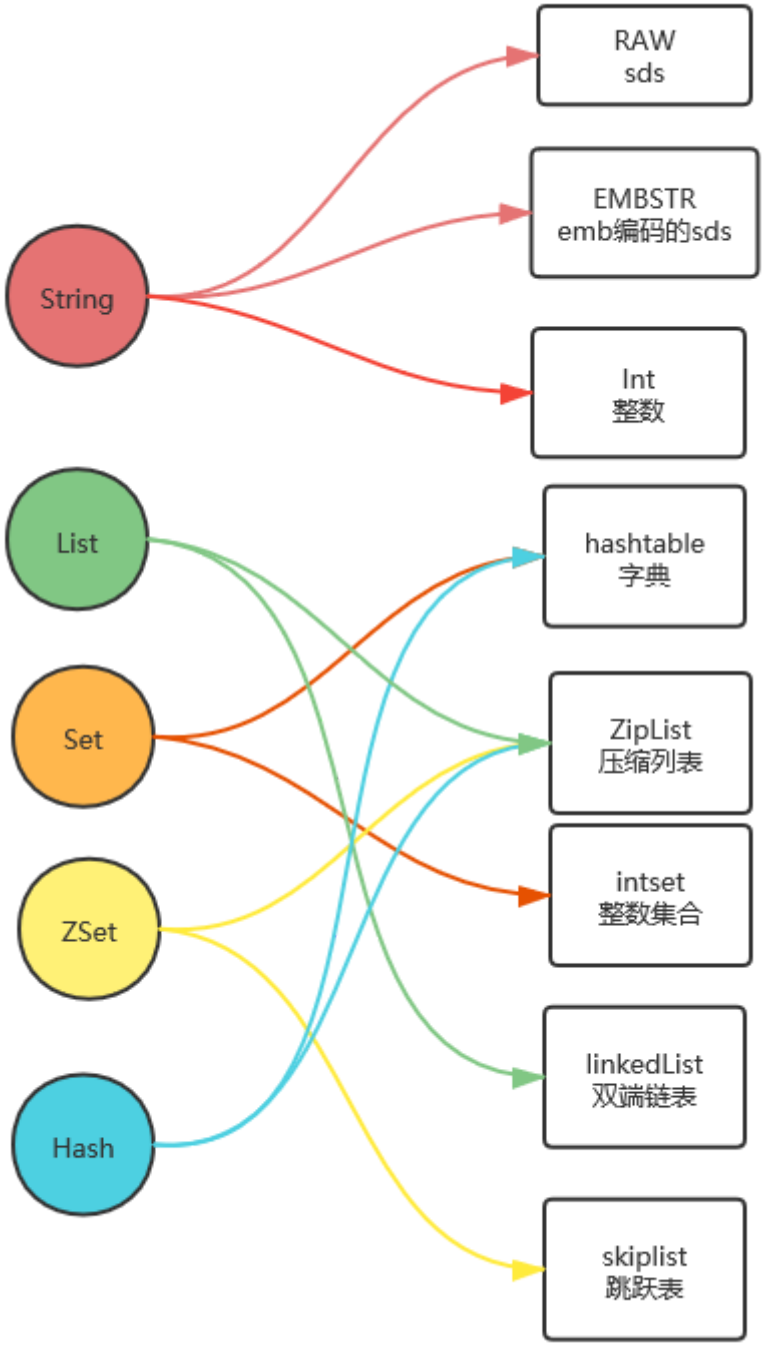
和查找操作类似，我们记录查找路径中，每一层最后一个比目标节点小的节点。然后再随机生成一个目标节点的层高，遍历层高，重新修改前进指针后退指针和跨度。

## 7. RedisObject

每次当我们在redis数据库新创建一个值键对时，至少会创建两个对象，一个用于作为建对象，一个用于作为值对象。redis每个对象都由一个redisObject的结构表示，该结构中与保存数据相关的三个属性分别是：type,encoding,ptr

```
/*
 * Redis 对象
 */
typedef struct redisObject {

    // 类型
    unsigned type:4;

    // 对齐位
    unsigned notused:2;

    // 编码方式
    unsigned encoding:4;

    // LRU 时间（相对于 server.lruclock，于server.lruclock比较得到对象空转时间）
    unsigned lru:22;

    // 引用计数
    int refcount;

    // 指向对象的值
    void *ptr;

}
```

而我们常用的Redis对象有：字符串对象，列表对象，集合对象，有序集合对象，hash对象这五种。这些对象底层的编码方式，也可以说是底层数据结构，可能有不同的实现方式。具体对应关系如下




# 8.键空间

## 7. 1 定义

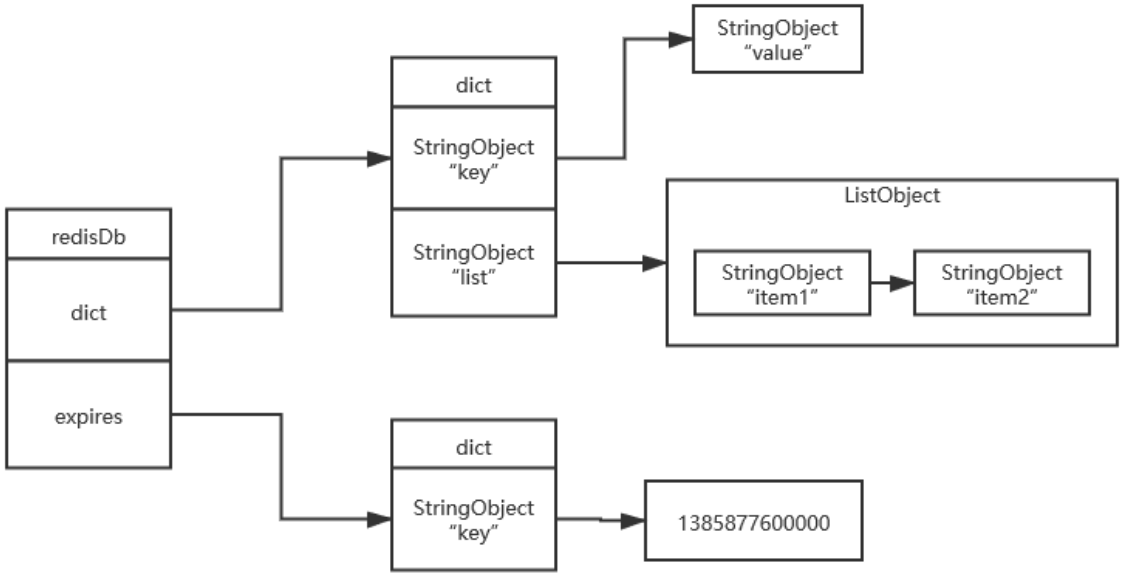
redis是一个键值对数据库服务器，每个数据库都有一个redisDb结构表示，其中redisDb结构的dict字典保存了数据库中的所有键值对，我们将这个字典称之为**键空间**。

- 键空间中的键也就是数据库的键，都是一个字符串对象。
- 键空间的芝也就是数据库的值，每一个值可以是上文所述的常用对象。

## 7.2 读写键空间时的维护操作

- 更新redisObject中的lru
- 发现该键过期则删除
- watch命令监视了某个键，那么服务器在对被监视的键进行修改之后，会将整个键标记为脏
- 每次修改一次键后，对脏键计数器值增加1，这个计数器会触发服务器的持久化以及复制操作

## 7.3 键空间结构示例




# 9. 过期键处理

所有的过期键维护在redisDb结构中的expires字典里，它的键和键空间dict中的键指向同一个对象，所以不会出现任何重复对象，也不会浪费任何空间。它的值是过期时间戳。

## 删除策略

惰性删除：
　　读写操作都会先判断键是否已经过期，如已经过期则删除

定期删除：
　　定时任务从expires中随机抽取一定数量的随机键进行检查，删除其中的过期键。

## 在持久化过程中对过期键的处理

### RDB

生成RDB文件不会把已经过期的键写入RDB文件中
主库加载RDB文件不会把已经过期的键写入RDB文件中，从库不区分，但是一般也会从主库复制的时候删除掉过期键。

### AOF

过期键已经过期，但是没有惰性删除和定期删除时，AOF文件无影响。只有在已经删除后，会在AOF文件中追加一条DEL命令。在AOF重写时，也不会把过期的键保存到新的AOF文件中。

## 复制过程中对过期键的处理

参考文献：https://blog.csdn.net/xiaochao_bos/article/details/103140678

在老版本的redis中，在读取过期键的时候，从库不会主动删除过期键（这种统一、中心化的过期键删除策略可以保持主从数据一致性，若从库可以删，那么主库延长过期时间，可能存在并发问题），而是直接返回原始值（但这里有问题），所以可能导致从库读取到脏数据的问题。

在在redis 3.2-rc1版本中之后版本的redis中，redis加入了一个新特性来解决主从不一致导致读取到过期数据的问题，如果key已过期，当前访问的是master则返回null；当前访问的是从库，且执行的是只读命令也返回null。

# 10. 内存淘汰机制

为了保证Redis的安全稳定运行，设置了一个max-memory的阈值，那么当内存用量到达阈值，新写入的键值对无法写入，此时就需要内存淘汰机制，在Redis的配置中有几种淘汰策略可以选择，详细如下：

**noeviction**: 当内存不足以容纳新写入数据时，新写入操作会报错；

**allkeys-lru**：当内存不足以容纳新写入数据时，在键空间中移除最近最少使用的 key；

**allkeys-random**：当内存不足以容纳新写入数据时，在键空间中随机移除某个 key；

**volatile-lru**：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 key；

**volatile-random**：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个 key；

**volatile-ttl**：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 key 优先移除；

## 近似lru实现

redis当内存超标后，每次抽取一定数量的key，选取其中最近最久没使用的key淘汰

## 删除key后，内存并不立即释放

官方说明：

```
https://redis.io/topics/memory-optimization
```

当键被删除时，Redis并不总是释放(返回)内存到操作系统。这并不是Redis的特别之处，但大多数malloc()实现都是这样工作的。例如一个实例有5gb的数据,然后删除相当于2gb的数据,used_memory_rss可能仍会5 gb左右。这是因为底层分配器不能轻松地释放内存。例如，通常将删除的大多数键分配到与其他仍然存在的键相同的页面中。然而分配器是聪明和能够重用空闲块的内存,所以在你释放2 gb的5 gb的数据集,当你开始再次增加键,您将看到used_memory_rss保持稳定而不会再增加很多。分配器基本上是尝试重用以前(逻辑上)释放的2GB内存。

# 11. 发布订阅

虽然redis实现了发布订阅（publish/subscribe）的功能，但是在通常的情况下是不推荐使用的，如果想使用消息队列这种功能，最好还是使用专业的各种MQ中间件，例如rabbitMQ，rockedMQ,activitedMQ等。概要说一下就是，PUBLISH和SUBSCRIBE的缺陷在于客户端必须一直在线才能接收到消息，断线可能会导致客户端丢失消息，除此之外，旧版的redis可能会由于订阅者消费不够快而变的不稳定导致崩溃，甚至被管理员杀掉

**第一个原因是和redis系统的稳定性有关。**对于旧版的redis来说，如果一个客户端订阅了某个或者某些频道，但是它读取消息的速度不够快，那么不断的积压的消息就会使得redis输出缓冲区的体积越来越大，这可能会导致redis的速度变慢，甚至直接崩溃。也可能会导致redis被操作系统强制杀死，甚至导致操作系统本身不可用。新版的redis不会出现这种问题，因为它会自动断开不符合client-output-

buffer-limit pubsub配置选项要求的订阅客户端

**第二个原因是和数据传输的可靠性有关。**任何网络系统在执行操作时都可能会遇到断网的情况。而断线产生的连接错误通常会使得网络连接两端中的一端进行重新连接。如果客户端在执行订阅操作的过程中断线，那么客户端将会丢失在断线期间的消息，这在很多业务场景下是不可忍受的。

# 12. RDB

## BGSAVE

> 那么RDB单机持久化时，过程中新写入的值是否会持久化了?
> 先说答案：不会
> 分析原因：RDB持久化的过程使用，为了节省内存，使用了copy on write 的策略，与父进程共享同一内存，此时若想当然认为新新写入的也会一同被子进程持久化，则错了。“写时复制“技术，在只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程，所以新
> 写入数据时，子进程会单独复制一份写之前的数据，此时此段子进程与父进程是各自独立维护的。当父进程对其中一个页面的数据进行修改时，会将被共享的页面复 制一份分离出来，然后 对这个复制的页面进行修改。这时 子进程 相应的页面是 没有变化的，还是进程产生时那一瞬间的数据。随着父进程修改操作的持续进行，越来越多的共享页面将会被分离出来，内存就会持续增长，但是也不会超过原有数据内存的两倍大小（Redis实例里的冷数据占的比例往往是比较高的，所以很少出现所有页面都被分离的情况）。

## 自动间隔性保存

**自动保存的判断依据**：在某一段时间内，redis的写操作数量有没有达到阈值。

**配置参数**

- 阈值参数：serverParams
- 写操作计数器：dirty
- 上一次保存完成的时间:lastsave

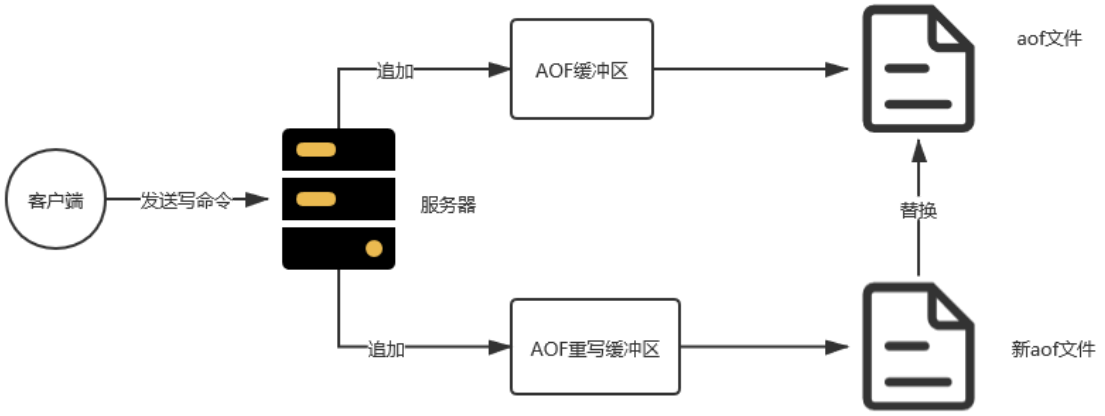
通过配置serverParams参数，调整自动保存的触发时间。当redis执行写命令时，会对dirty计数器进行累加操作。而redis的服务器周期性操作函数severCron会定时执行一次，该函数用于对正在进行的服务器进行位，其中一项重要工作就是遍历serverParams数组判断是否达到自动保存的条件。

# 13. AOF

## aof缓冲刷盘配置

- always：将aof缓冲区的内容同步到aof文件
- everysec：将aof缓冲区内容每秒同步一次
- no：由操作系统来决定何时刷盘

## aof后台重写




子进程在进行 AOF 重写期间， 主进程还需要继续处理命令， 而新的命令可能对现有的数据进行修改，会出现数据库的数据和重写后的 AOF 文件中的数据不一致。因此Redis 增加了一个 AOF 重写缓存， 这个缓存在 fork 出子进程之后开始启用， Redis 主进程在接到新的写命令之后， 除了会将这个写命令的协议内容追加到现有的 AOF 文件之外， 还会追加到这个缓存中。当子进程完成 AOF 重写之后向父进程发送一个完成信号， 父进程在接到完成信号之后会调用信号处理函数，完成以下工作：

1. 将 AOF 重写缓存中的内容全部写入到新 AOF 文件中
2. 对新的 AOF 文件进行改名，覆盖原有的 AOF 文件

## aof后台重写的触发时机

AOF 重写可以由用户通过调用 BGREWRITEAOF 手动触发。

服务器在 AOF 功能开启的情况下，会维持以下三个变量：

1. 当前 AOF 文件大小
2. 最后一次 重写之后， AOF 文件大小的变量
3. AOF文件大小增长百分比

每次当 serverCron 函数执行时， 它都会检查以下条件是否全部满足， 如果是的话， 就会触发自动的 AOF 重写：

1. 没有 BGSAVE 命令在进行 防止于RDB的冲突
2. 没有 BGREWRITEAOF 在进行 防止和手动AOF冲突
3. 当前 AOF 文件大小至少大于设定值 基本要求 太小没意义
4. 当前 AOF 文件大小和最后一次 AOF 重写后的大小之间的比率大于等于指定的增长百分比

# 14. 混合持久化

**旧版本redis恢复策略**：默认aof，如果没有配置aof则才是RDB

**Redis4.0混合持久化**：重启 Redis 时，如果使用 RDB 来恢复内存状态，会丢失大量数据。而如果只使用 AOF 日志重放，那么效率又太过于低下。Redis 4.0 提供了混合持久化方案，将 RDB 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是自 RDB 持久化开始到持久化结束这段时间发生的增量 AOF 日志，通常这部分日志很小。

# 15. 事件

## 事件分类

- 文件事件：读事件，写事件
- 时间时间：周期性时间，定时时间
    1. 删除数据库的key
    2. 触发RDB和AOF持久化
    3. 主从同步
    4. 集群化保活
    5. 关闭清理死客户端链接
    6. 统计更新服务器的内存、key数量等信息

## 时间事件的组织方式

以list结构组成一个链表


## 事件的执行

我们都知道redis是单线程模型，时间自然也是单线程执行的。而文件时间，采用的是reactor模式，用io多路复用，获取要执行的文件事件。而我们知道io多路复用，在select或者epoll.wait的时候，是会阻塞的，当然也可以传递超时时间，到达超时时间后不管有无读写请求，均可返回。而我们知道时间时间维护着一些很重要的功能，如果文件事件阻塞过长，会导致时间事件执行受影响。那么我们如何制定文件事件的阻塞事件呢?

> 遍历时间事件列表，获取最近到期的时间事件的时间，设置epoll的超时时间不超过此时间。这样，我们就能大致保证时间事件按时执行。由于时间事件在文件事件之后执行，并且事件之间不会出现抢占，所以时间事件的实际处理时间一般会比设定的时间稍晚一些。

# 16. 主从同步

主从同步是在时间事件中异步处理，并不是写命令执行完立马同步。

## 旧版本

实现

> RDB文件+写命令传播

缺陷

> 断线重连后，从服务器需要发送sync命令到主服务器重新生成RDB文件。
> 主从同步过程中，写命令传播时如果命令丢失了，主从是无法感知到的

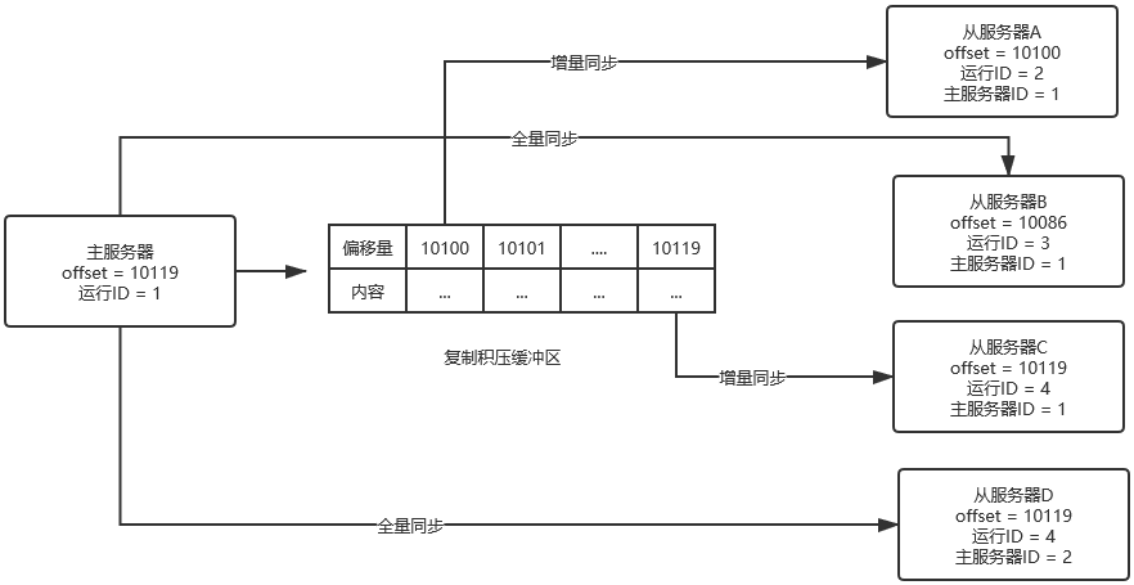
## 新版本

redis从2.8版本后，采用psync命令替代了sync命令，psync命令有两种模式

- 全量同步
- 增量同步

### 增量同步的实现

- 主服务器和从服务器的复制偏移量：表示数据偏移量
- 主服务器的复制积压缓冲区：FIFO的环形数组
- 服务器运行ID




服务器A：本地保存的主服务器ID与实际的主服务器ID相同，偏移量10100能在复制积压缓冲区找到，所以进行增量同步

服务器B：本地保存的主服务器ID与实际的主服务器ID相同，偏移量10086不能在复制积压缓冲区找到，所以进行全量同步

服务器C：本地保存的主服务器ID与实际的主服务器ID相同，偏移量10119能在复制积压缓冲区找到，所以进行增量同步

服务器D：本地保存的主服务器ID与实际的主服务器ID不相同，所以进行全量同步

# 17. Sentinel

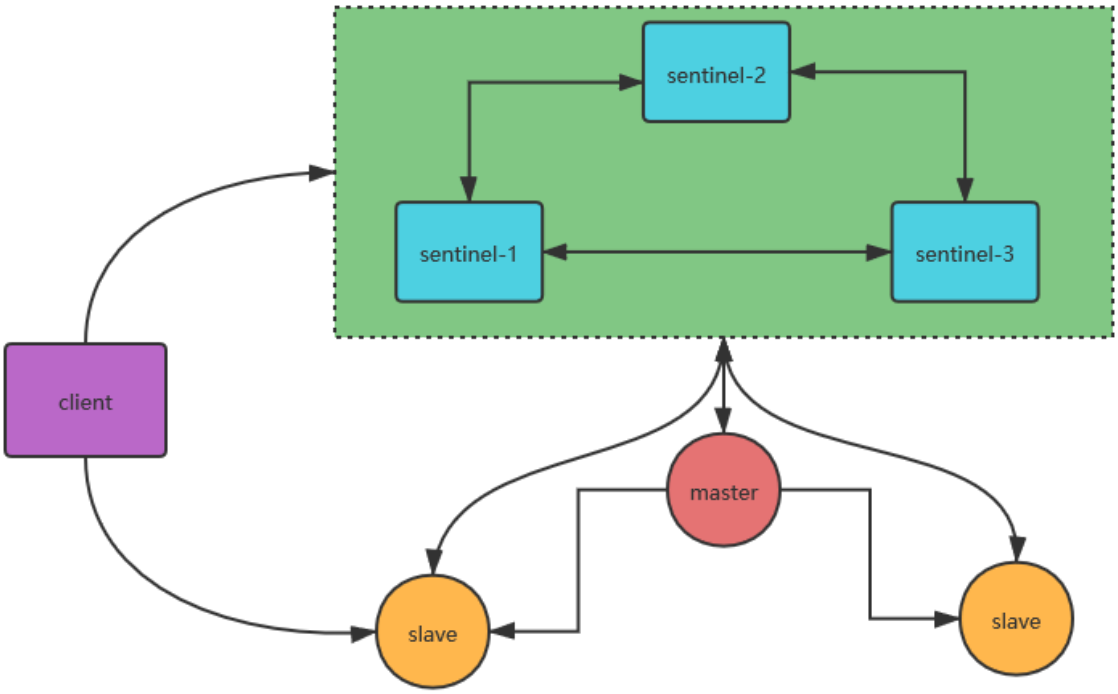
## sentinel产生的原因

主从同步虽然提高了系统的可用性，但是存在以下问题：

- 主节点挂了需要手动主从切换
- 判断主节点挂了的机制是否健全和标准

sentinel就是为了解决上述问题而存在的

## sentinel架构介绍




**两类节点**：

- sentinel节点：监控其他节点
- 数据节点：存储数据

两类节点都是redis节点，不过sentinel节点是用来做监控的，数据节点是用来存数据的。

**客户端首次来链接集群时**，会先链接sentinel，通过sentinel查询主节点信息，然后再与主节点交互，查询数据。

**当主节点故障时**，客户端会重新链接sentinel节点，通过sentinel查询主节点信息，然后再与主节点交互，查询数据。

## Sentinel的通信命令

`Sentinel` 节点连接一个 `Redis` 实例的时候，会创建 `cmd` 和 `pub/sub` 两个 **连接。** `Sentinel` 通过 `cmd` 连接给 `Redis` 发送命令，通过 `pub/sub` 连接到 `Redis` 实例上的其他 `Sentinel` 实例。

`Sentinel` 与 `Redis` **主节点** 和 **从节点** 交互的命令，主要包括：

| 命令 | 作 用 |
| --- | --- |
| PING | `Sentinel` 向 `Redis` 节点发送 `PING` 命令，检查节点的状态 |
| INFO | `Sentinel` 向 `Redis` 节点发送 `INFO` 命令，获取它的 **从节点信息** |
| PUBLISH | `Sentinel` 向其监控的 `Redis` 节点 `__sentinel__:hello` 这个 `channel` 发布 **自己的信息** 及 **主节点** 相关的配置 |
| SUBSCRIBE | `Sentinel` 通过订阅 `Redis` **主节点** 和 **从节点** 的 `__sentinel__:hello` 这个 `channnel`，获取正在监控相同服务的其他 `Sentinel` 节点 |

`Sentinel` 与 `Sentinel` 交互的命令，主要包括：

| 命令 | 作 用 |
| --- | --- |
| PING | `Sentinel` 向其他 `Sentinel` 节点发送 `PING` 命令，检查节点的状态 |
| SENTINEL:is-master-down-by-addr | 和其他 `Sentinel` 协商 **主节点** 的状态，如果 **主节点** 处于 `SDOWN` 状态，则投票自动选出新的 **主节点** |

## 三个定时任务

1. sentinel向master节点发送info命令，获取主从节点的信息
2. sentinel向数据节点的*sentinel*:hello频道发送自己对主节点的判断以及当前sentinel节点的信息
3. sentinel向所有其他节点发送ping命令用来检测心跳

## 主观下线和客观下线

默认情况下，**每个** `Sentinel` 节点会以 **每秒一次** 的频率对 `Redis` 节点和 **其它** 的 `Sentinel` 节点发送 `PING` 命令，并通过节点的 **回复** 来判断节点是否在线。

- **主观下线**

**主观下线** 适用于所有 **主节点** 和 **从节点。**如果在 `down-after-milliseconds` 毫秒内，`Sentinel` 没有收到 **目标节点** 的有效回复，则会判定 **该节点** 为 **主观下线。**

- **客观下线**

**客观下线** 只适用于 **主节点。**如果 **主节点** 出现故障，`Sentinel` 节点会通过 `sentinel is-master-down-by-addr` 命令，向其它 `Sentinel` 节点询问对该节点的 **状态判断。**如果超过 `<quorum>` 个数的节点判定 **主节点** 不可达，则该 `Sentinel` 节点会判断 **主节点** 为 **客观下线。**

## 故障迁移

当sentinel节点对主节点进行客观下线操作后，需要进行一下两个步骤进行故障迁移

1. 选举sentinel领导者
2. 由领导者选出最新的slave节点作为主节点

## sentinel领导者选举

raft算法选举领导者，这里就不展开讲解了，推荐一篇好文给大家了解

```
https://mp.weixin.qq.com/s?
__biz=MzIwMjU4MzU4MA==&mid=2247484698&idx=1&sn=ca53dcbb50957daab29beebbba2f0140&
chksm=96dd3e69a1aab77f1c27aa488d04ae52c0ce1dfecf2b909374934291596bf65eb8d3585ed3
f8&scene=178&cur_album_id=1343082386795626497#rd
```

# 18. Cluster

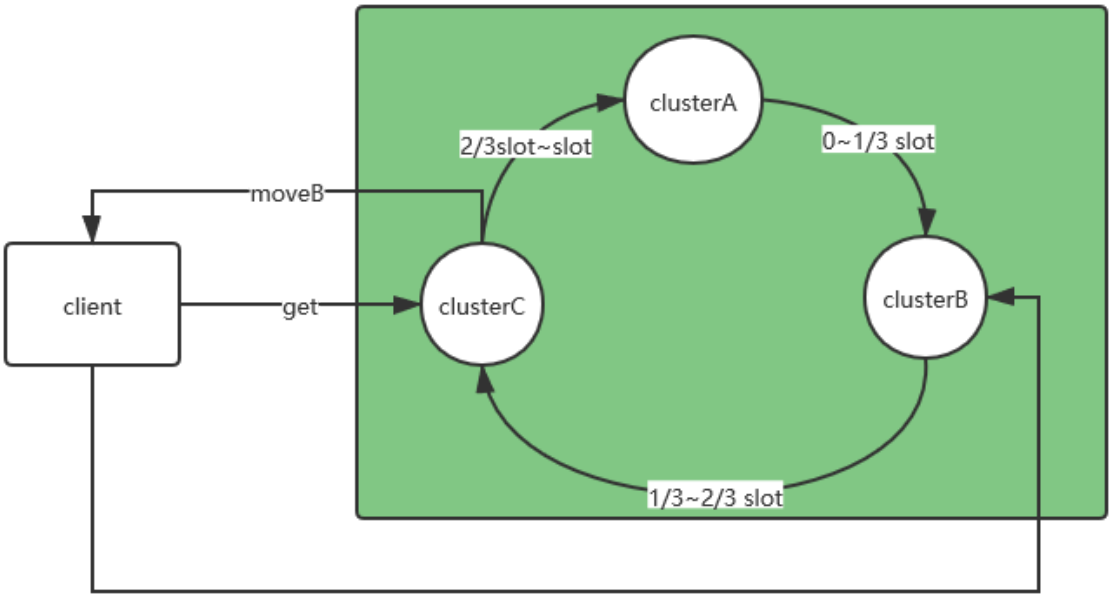
## cluster产生的原因

sentinel架构保证了高可用，但是仍然存在一个问题是，数据始终是存储在master节点上，如果要求海量数据存储，单一的master节点无法应对，必须要多个master节点存储数据，这个时候就有了codis架构。

```
codis架构：客户端不直接访问master节点，通过codis代理访问master节点，codis是无状态的，这样实际的redis数据节点就可以横向扩展。以存储海量数据
```

在codis架构之后，redis官方推出了它的集群版本cluster，来应对海量数据的存储。

# cluster架构介绍




### 数据存储

采用一致性hash算法，把数据分散到不同的cluster节点中存储

### 数据访问

如果数据在此cluster节点中，直接返回，如果不在，则返回**move错误**，告知客户端应该去那个redis节点

# 重新分片

reshard可以将已经分配给某个节点的任意数量的slot迁移给另一个节点，在Redis内部是由redis-trib负责执行的。你可以理解为Redis其实已经封装好了所有的命令，而redis-trib则负责向获取slot的节点和被转移slot的节点发送命令来最终实现reshard。

假设我们需要向集群中加入一个D节点，而此时集群内已经有A、B、C三个节点了。

此时redis-trib会向A、B、C三个节点发送迁移出槽位的请求，同时向D节点发送准备导入槽位的请求，做好准备之后A、B、C这三个源节点就开始执行迁移，将对应的slot所对应的键值对迁移至目标节点D。最后redis-trib会向集群中所有主节点发送槽位的变更信息。

### 重分片时候访问数据

当客户端向源节点发送一个访问命令时，如果该节点恰好正在迁移数据，则会返回一个**ASK错误**，告知客户端去目标节点查找数据。

# 节点通信

redis cluster采用gossip算法进行消息通信。

## gossip协议消息类型

Redis Cluster中，节点之间的消息类型有5种，分别是MEET、PING、PONG、FAIL和PUBLISH。这些消息分别传递了什么内容呢？我简单总结了一下。

| 消息类型 | 消息内容 |
| --- | --- |
| MEET | 给某个节点发送MEET消息，请求接收消息的节点加入到集群中 |
| PING | 每隔一秒钟，选择5个最久没有通信的节点，发送PING消息，检测对应的节点是否在线；同时还有一种策略是，如果某个节点的通信延迟大于了`cluster-node-time`的值的一半，就会立即给该节点发送PING消息，避免数据交换延迟过久 |
| PONG | 当节点接收到MEET或者PING消息之后，会回一个PONG消息给发送方，代表自己收到了MEET或者PING消息。同时，节点也可以主动的通过PONG消息向集群中广播自己的信息，让其他节点获取到自己最新的属性，就像完成了故障转移之后新的master向集群发送PONG消息一样 |
| FAIL | 用于广播自己的对某个节点的宕机判断，假设当前节点对A节点判断为宕机，就会立即向Redis Cluster广播自己对于A节点的判断，所有收到消息的节点就会对A节点做标记 |
| PUBLISH | 用于向指定的Channel发送消息，某个节点收到PUBLISH消息之后会直接在集群内广播，这样一来，客户端无论连接到任何节点都能够订阅这个Channel |

## 使用gossip的优劣

既然Redis Cluster选择了gossip，那肯定存在一些gossip的优点，我们接下来简单梳理一下。

| 优点 | 描述 |
| --- | --- |
| 扩展性 | 网络可以允许节点的任意增加和减少，新增加的节点的状态最终会与其他节点一致。 |
| 容错性 | 由于每个节点都持有一份完整元数据，所以任何节点宕机都不会影响gossip的运行 |
| 健壮性 | 与容错性类似，由于所有节点都持有数据，地位平台，是一个去中心化的设计，任何节点都不会影响到服务的运行 |
| 最终一致性 | 当有新的信息需要传递时，消息可以快速的发送到所有的节点，让所有的节点都拥有最新的数据 |

gossip可以在O(logN) 轮就可以将信息传播到所有的节点，为什么是O(logN)呢？因为每次ping，当前节点会带上自己的信息外加整个Cluster的1/10数量的节点信息，一起发送出去。你可以简单的把这个模型抽象为：

> 你转发了一个特别有意思的文章到朋友圈，然后你的朋友们都觉得还不错，于是就一传十、十传百这样的散播出去了，这就是朋友圈的**裂变传播**。

当然，gossip仍然存在一些缺点。例如消息可能最终会经过很多轮才能到达目标节点，而这可能会带来较大的延迟。同时由于节点会随机选出5个最久没有通信的节点，这可能会造成某一个节点同时收到n个重复的消息。

# 故障迁移

当一个从节点发现自己正在复制的主节点进入了已下线状态时，从节点将开始对下线主节点进行故障转移，以下是故障转移执行的步骤：

```
1.复制下线主节点的所有从节点里面，会有一个从节点被选中；
2.被选中的从节点会执行SLAVEOF no one命令，成为新的主节点；
3.新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己；
4.新的主节点向集群广播一条PONG消息，这条PONG消息可以让集群中的其他节点立即知道这个节点已经由从节点变成了主节点，并且这个主节点已经接管了原本由已下线节点负责处理的槽。
5.新的主节点开始接收和自己负责处理的槽有关的命令请求，故障转移完成。
```

选举新主节点的算法和sentinel领导者选举算法一致，也是使用raft算法