

# Redis

## 基础

1. 什么是Redis
2. Redis有哪些优缺点
3. 为什么要用 Redis /为什么要用缓存

高性能：

高并发：

4. 为什么要用 Redis 而不用 map/guava 做缓存?
5. Redis与Memcached的区别
6. Redis为什么这么快

## 数据类型

1. Redis有哪些数据类型
2. Redis的应用场景
  - 2.1 分布式计数器
  - 2.2 缓存
  - 2.3 会话缓存
  - 2.4 消息队列(发布/订阅功能)
  - 2.5 分布式锁实现
  - 2.6 其它

## Redis 的数据类型的底层实现

0. 数据类型的底层实现
  - 0.1. String 的底层实现原理
    - int
    - embstr
    - raw
  - 0.2. 列表的底层实现原理
  - 0.3. 字典的底层实现原理
  - 0.4. 集合的底层实现
  - 0.5. 有序集合的底层实现

## 1. 说一说 redis 的底层数据结构

1.1 字符串

1.2 链表linkedlist

1.3 哈希表 hashtable

1.4 跳跃表skiplist

1.5 整数集合intset

1.6 压缩列表ziplist：压缩列表是为节约内存而开发的顺序性数据结构，它可以包含任意多个节点，每个...

1.7 quicklist (3.2)

1.8 listpack (5.0)

## 2. Redis 的 SDS 和 C 中字符串相比有什么优势？

## 3. 字典是如何实现的？Rehash 了解吗？

## 4. 跳跃表是如何实现的？原理？

什么是跳跃表

查找流程：

为什么使用跳跃表？

跳跃表是怎么实现的？

## 5. 压缩列表了解吗？

## 6. 快速列表 quicklist 了解吗？

## 7. 数据类型的实现

## 8. 什么是空间预分配以及惰性空间释放，SDS 是怎么实现的

## 9. 为什么说 SDS 是二进制安全的呢

## 10. 说说 redis 里的对象

## 11. 使用 RedisObject 的好处

## 12. RedisObject 的具体结构是什么

## 持久化

## 1. 什么是Redis持久化？

## 2. Redis 的持久化机制是什么？各自的优缺点？

2.1 RDB (Redis DataBase) , 快照

2.2 AOF (Append Only File) , 日志

## 3. 优缺点是什么？

## 4. 如何选择合适的持久化方式

## 5. Redis持久化数据和缓存怎么做扩容？

6. Redis 怎么确保 Aof 不丢失
7. Aof 日志是写前还是写后日志
8. AOF为什么要先执行命令再记日志
9. Redis Aof 的风险
10. Redis 的写回策略
11. 如何选择回写策略
12. Redis 回写有什么性能问题
13. Aof 日志文件太大怎么办
14. redis 的重写机制了解吗
15. 为什么重写会使日志文件变小
16. AOF 重写会堵塞主线程吗
17. 重写的过程

18. AOF日志重写的时候，是由bgrewriteaof子进程来完成的，不用主线程参与，我们今天说的非阻塞也是...  
19. AOF重写也有一个重写日志，为什么它不共享使用AOF本身的日志呢

## 垃圾回收

1. Redis的过期键的删除策略
  - 定时过期：
  - 惰性过期：
  - 定期过期：
2. Redis key的过期时间和永久有效分别怎么设置？
3. 我们知道通过 expire 来设置key 的过期时间，那么对过期的数据怎么处理呢？
4. MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据
5. Redis的内存淘汰策略有哪些
6. Redis主要消耗什么物理资源？
7. Redis的内存用完了会发生什么？
8. Redis如何做内存优化？
9. Redis回收进程如何工作的？
10. Redis回收使用的是什么算法？

## 线程模型

1. Redis线程模型
2. 单线程为什么快

## 事务

1. 什么是事务?
2. Redis事务的概念
3. Redis事务的三个阶段
4. Redis事务相关命令
5. 事务管理（ACID）概述

    原子性（Atomicity）

    一致性（Consistency）

    隔离性（Isolation）

    持久性（Durability）

6. Redis事务支持隔离性吗

7. Redis事务保证原子性吗，支持回滚吗

8. Redis事务其他实现

9. redis 事务、流水线与 lua 脚本有什么区别

    Pipeline 流水线

    事务

    Lua 脚本

10. 了解 WATCH 命令吗

11. redis 事务的安全性如何保证

LUA 脚本

1. redis 为什么要引入 lua 脚本
2. redis 使用 lua 有什么特性
3. lua 脚本执行一半，机器断电了会怎么样
4. lua 脚本跟事务有什么差别

分布式问题

1. Redis实现分布式锁
2. 如何解决 Redis 的并发竞争 Key 问题
3. 分布式Redis是前期做还是后期规模上来了再做好？为什么？
4. 什么是 RedLock

缓存异常

1. 缓存雪崩
2. 缓存穿透
3. 缓存击穿

4. 缓存预热
5. 缓存降级
6. 热点数据和冷数据
7. 缓存热点key

## 其他问题

1. 如何保证缓存与数据库双写时的数据一致性?
2. Redis常见性能问题和解决方案?
3. 一个字符串类型的值能存储最大容量是多少?
4. Redis如何做大量数据插入?
5. 假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出...
6. 使用Redis做过异步队列吗，是如何实现的
7. Redis如何实现延时队列

## 大Key

1. 什么是 Redis 大 key 问题

2. 大key会造成什么问题呢?

3. 如何找到大key?

4. 如何处理大key?

    4.1 删除大key

    4.2 压缩和拆分key

## 集群方案 (校招基本不问)

哨兵模式

官方Redis Cluster 方案(服务端路由查询)

基于客户端分配

基于代理服务器分片

Redis 主从架构

Redis集群的主从复制模型是怎样的?

生产环境中的 redis 是怎么部署的?

说说Redis哈希槽的概念?

Redis集群会有写操作丢失吗? 为什么?

Redis集群之间是如何复制的?

Redis集群最大节点个数是多少?

Redis集群如何选择数据库?

分区（校招基本不会问）

Redis是单线程的，如何提高多核CPU的利用率？

为什么要做Redis分区？

你知道有哪些Redis分区实现方案？

Redis分区有什么缺点？

## 基础

### 1. 什么是Redis

Redis(Remote Dictionary Server) 是一个使用 C 语言编写的，开源的（BSD许可）高性能非关系型（NoSQL）的键值对数据库。

Redis 可以存储键和五种不同类型的值之间的映射。键的类型只能为字符串，值支持五种数据类型：字符串、列表、集合、散列表、有序集合。

与传统数据库不同的是 Redis 的数据是存在内存中的，所以读写速度非常快，因此 redis 被广泛应用于缓存方向，每秒可以处理超过 10万次读写操作，是已知性能最快的Key–Value DB。另外，Redis 也经常用来看做分布式锁。除此之外，Redis 支持事务、持久化、LUA脚本、LRU驱动事件、多种集群方案。

### 2. Redis有哪些优缺点

优点

- 读写性能优异。
- 支持数据持久化，支持AOF和RDB两种持久化方式。
- 支持事务，Redis的所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行。
- 数据结构丰富，除了支持string类型的value外还支持hash、set、zset、list等数据结构。
- 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

缺点

- 数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较少数据量的高性能操作和运算上。
- Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。
- 主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性。

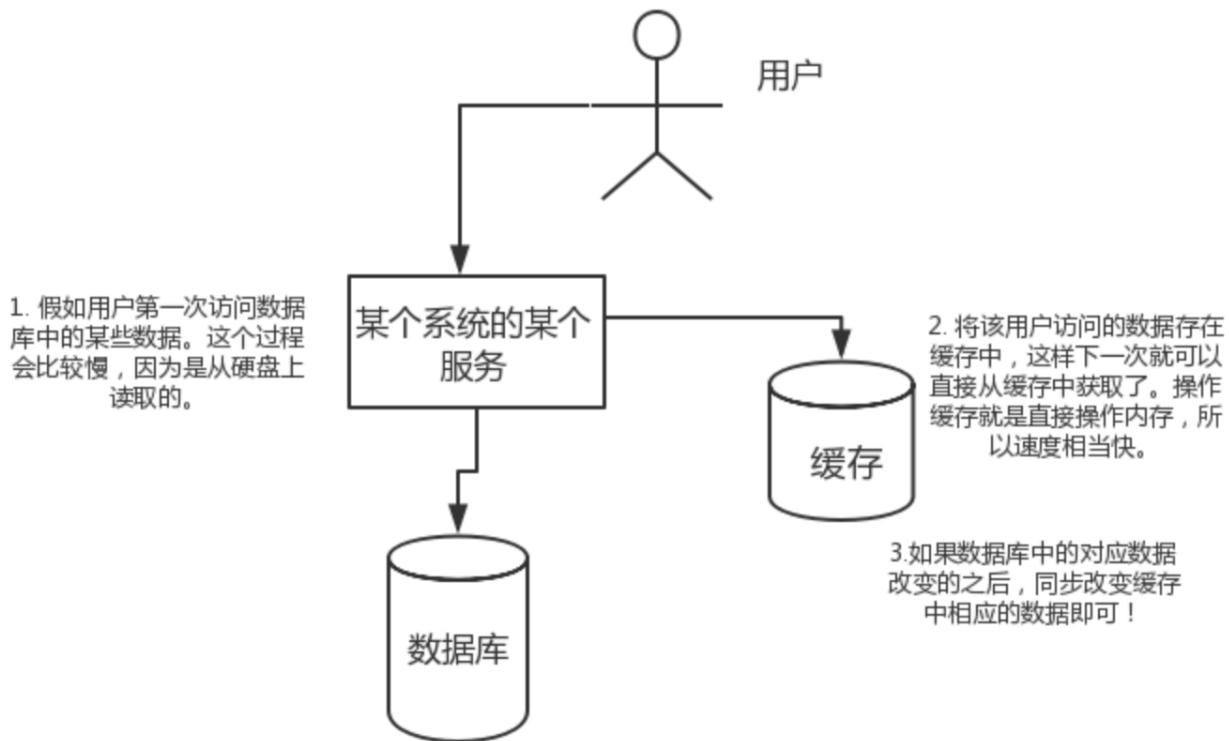
- Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

### 3. 为什么要用 Redis /为什么要用缓存

主要从“高性能”和“高并发”这两点来看待这个问题。

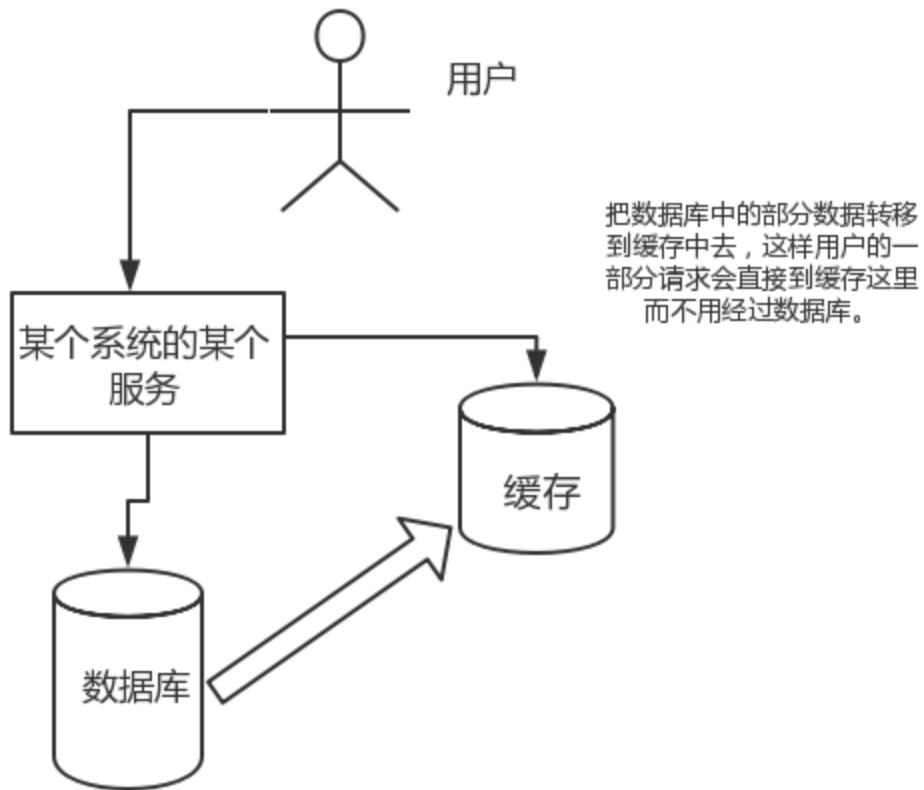
#### 高性能：

假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在数缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可！



#### 高并发：

直接操作缓存能够承受的请求数量是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。



## 4. 为什么要用 Redis 而不用 map/guava 做缓存？

缓存分为本地缓存和分布式缓存。比如说我们可以使用全局 map 实现本地缓存。本地缓存多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached 服务的高可用，整个程序架构上较为复杂。

## 5. Redis与Memcached的区别

两者都是非关系型内存键值数据库，现在公司一般都是用 Redis 来实现缓存，而且 Redis 自身也越来越强大了！Redis 与 Memcached 主要有以下不同：

对比参数	Redis	Memcached
类型	1. 支持内存 2. 非关系型数据库	1. 支持内存 2. 键值对形式 3. 缓存形式
数据存储类型	1. String 2. List 3. Set 4. Hash 5. Sort Set 【俗称ZSet】	1. 文本型 2. 二进制类型
查询【操作】类型	1. 批量操作 2. 事务支持 3. 每个类型不同的CRUD	1. 常用的CRUD 2. 少量的其他命令
附加功能	1. 发布/订阅模式 2. 主从分区 3. 序列化支持 4. 脚本支持【Lua脚本】	1. 多线程服务支持
网络IO模型	1. 单线程的多路 IO 复用模型	1. 多线程，非阻塞IO模式
事件库	自封转简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1. RDB 2. AOF	不支持
集群模式	原生支持 cluster 模式，可以实现主从复制，读写分离	没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据
内存管理机制	在 Redis 中，并不是所有数据都一直存储在内存中，可以将一些很久没用的 value 交换到磁盘	Memcached 的数据则会一直在内存中，Memcached 将内存分割成特定长度的块来存储数据，以完全解决内存碎片的问题。但是这种方式会使得内存的利用率不高，例如块的大小为 128 bytes，只存储 100 bytes 的数据，那么剩下的 28 bytes 就浪费掉了。
适用场景	复杂数据结构，有持久化，高可用需求，value存储内容较大	纯key-value，数据量非常大，并发量非常大的业务

- (1) memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型
- (2) redis的速度比memcached快很多
- (3) redis可以持久化其数据

## 6. Redis为什么这么快

- 1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是O(1)；
- 2、数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；
- 3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- 4、使用多路 I/O 复用模型，非阻塞 IO；

5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

## 数据类型

### 1. Redis有哪些数据类型

Redis主要有5种数据类型，包括 String, List, Set, Zset, Hash，满足大部分的使用要求

数据类型	可以存储的值	操作	应用场景
STRING	字符串、整数或者浮点数	对整个字符串或者字符串的其中一部分执行操作 对整数和浮点数执行自增或者自减操作	做简单的键值对缓存
LIST	列表	从两端压入或者弹出元素 对单个或者多个元素进行修剪，只保留一个范围内的元素	存储一些列表型的数据结构，类似粉丝列表、文章的评论列表之类的数据
SET	无序集合	添加、获取、移除单个元素 检查一个元素是否存在于集合中 计算交集、并集、差集 从集合里面随机获取元素	交集、并集、差集的操作，比如交集，可以把两个人的粉丝列表整合一个交集
HASH	包含键值对的无序散列表	添加、获取、移除单个键值对 获取所有键值对 检查某个键是否存在	结构化的数据，比如一个对象
ZSET	有序集合	添加、获取、删除元素 根据分值范围或者成员来获取元素 计算一个键的排名	去重但可以排序，如获取排名前几名的用户

### 2. Redis的应用场景

#### 2.1 分布式计数器

可以对 String 进行自增自减运算，从而实现计数器功能。Redis 这种内存型数据库的读写性能非常高，很适合存储频繁读写的计数量。

#### 2.2 缓存

将热点数据放到内存中，设置内存的最大使用量以及淘汰策略来保证缓存的命中率。

#### 2.3 会话缓存

可以使用 Redis 来统一存储多台应用服务器的会话信息。当应用服务器不再存储用户的会话信息，也就不再具有状态，一个用户可以请求任意一个应用服务器，从而更容易实现高可用性以及可伸缩性。

## 查找表

例如 DNS 记录就很适合使用 Redis 进行存储。查找表和缓存类似，也是利用了 Redis 快速的查找特性。但是查找表的内容不能失效，而缓存的内容可以失效，因为缓存不作为可靠的数据来源。

## 2.4 消息队列(发布/订阅功能)

List 是一个双向链表，可以通过 lpush 和 rpop 写入和读取消息。**不过最好使用 Kafka、RabbitMQ 等消息中间件。**

## 2.5 分布式锁实现

在分布式场景下，无法使用单机环境下的锁来对多个节点上的进程进行同步。可以使用 Redis 自带的 **SETNX** 命令实现分布式锁，除此之外，还可以使用官方提供的 RedLock 分布式锁实现。

## 2.6 其它

Set 可以实现交集、并集等操作，从而实现共同好友等功能。ZSet 可以实现有序性操作，从而实现排行榜等功能。

# Redis 的数据类型的底层实现

## 0. 数据类型的底层实现

### 0.1. String 的底层实现原理

String 主要有三种编码方式：`int`、`embstr`、`raw`，当字符串是一个可以用长整型（64位有符号整数）表示的时候，采用 `int` 编码；当字符串长度小于45字节时(`redis5`)，采用 `embstr` 编码，其余情况使用 `raw` 编码。

#### int

Redis 启动时会预先建立 10000 个分别存储 0~9999 的 `redisObject` 变量作为共享对象，这就意味着如果 set 字符串的键值在 0~10000 之间的话，则可以 **直接指向共享对象** 而不需要再建立新对象，此时键值不占空间！

```

/*
 * Check if we can represent this string as a long integer.
 * Note that we are sure that a string larger than 20 chars is not
 * representable as a 32 nor 64 bit integer. */
len = sdslen(s);
if (len <= 20 && string2l(s,len,&value)) {
    /* This object is encodable as a long. Try to use a shared object.
     * Note that we avoid using shared integers when maxmemory is used
     * because every object needs to have a private LRU field for the LRU
     * algorithm to work well. */
    if ((server.maxmemory == 0 ||
        !(server.maxmemory_policy & MAXMEMORY_FLAG_NO_SHARED_INTEGERS)) &&
        value >= 0 &&
        value < OBJ_SHARED_INTEGERS)
    {
        decrRefCount(o);
        incrRefCount(shared.integers[value]);
        return shared.integers[value];
    } else {
        if (o->encoding == OBJ_ENCODING_RAW) sdsfree(o->ptr);
        o->encoding = OBJ_ENCODING_INT;
        o->ptr = (void*) value;
        return o;
    }
}

```

当键值可用一个64位有符号整形表示时（即长度<20），Redis将用LONG类型来存储，编码即为OBJ\_ENCODING\_INT

当键值为0~10000之间时，优先使用共享字符串对象的策略

## embstr

顾名思义即： embedded string，表示嵌入式的String。从内存结构上看是字符串 sds 结构体与其对应的 redisObject 对象分配在 同一块连续的内存空间，这就仿佛字符串 sds 嵌入在 redisObject 对象之中一样，这一切从下面的代码即可清楚地看到：

```

/*
 * Create a string object with EMBSTR encoding if it is smaller than
 * OBJ_ENCODING_EMBSTR_SIZE_LIMIT, otherwise the RAW encoding is
 * used.
 *
 * The current limit of 39 is chosen so that the biggest string object
 * we allocate as EMBSTR will still fit into the 64 byte arena of jemalloc. */
#define OBJ_ENCODING_EMBSTR_SIZE_LIMIT 44
robj *createStringObject(const char *ptr, size_t len) {
    if (len <= OBJ_ENCODING_EMBSTR_SIZE_LIMIT)      当字符串长度<=44时用
        return createEmbeddedStringObject(ptr,len); EMBSTR 编码方式；超过时
    else                                              就用 RAW 编码方式
        return createRawStringObject(ptr,len);
}

```

## raw

与上面的 `OBJ_ENCODING_EMBSTR` 编码方式的不同之处在于 此时动态字符串 sds 的内存与其依赖的 `redisObject` 的 内存不再连续，需要申请两次内存。

## 0.2. 列表的底层实现原理

列表对象的编码有两种，分别是：`ziplist`、`linkedlist`。

当列表的长度小于 512，并且所有元素的长度都小于 64 字节时，使用压缩列表存储；否则使用 `linkedlist` 存储。

## 0.3. 字典的底层实现原理

哈希对象的编码有两种，分别是：`ziplist`、`hashtable`。

当哈希对象保存的键值对数量小于 512，并且所有键值对的长度都小于 64 字节时，使用压缩列表存储；否则使用 `hashtable` 存储。

## 0.4. 集合的底层实现

集合对象的编码有两种，分别是：`intset`、`hashtable`。

当集合的长度小于某个特定值，并且所有元素都是整数时，使用整数集合存储；否则使用 `hashtable` 存储。

## 0.5. 有序集合的底层实现

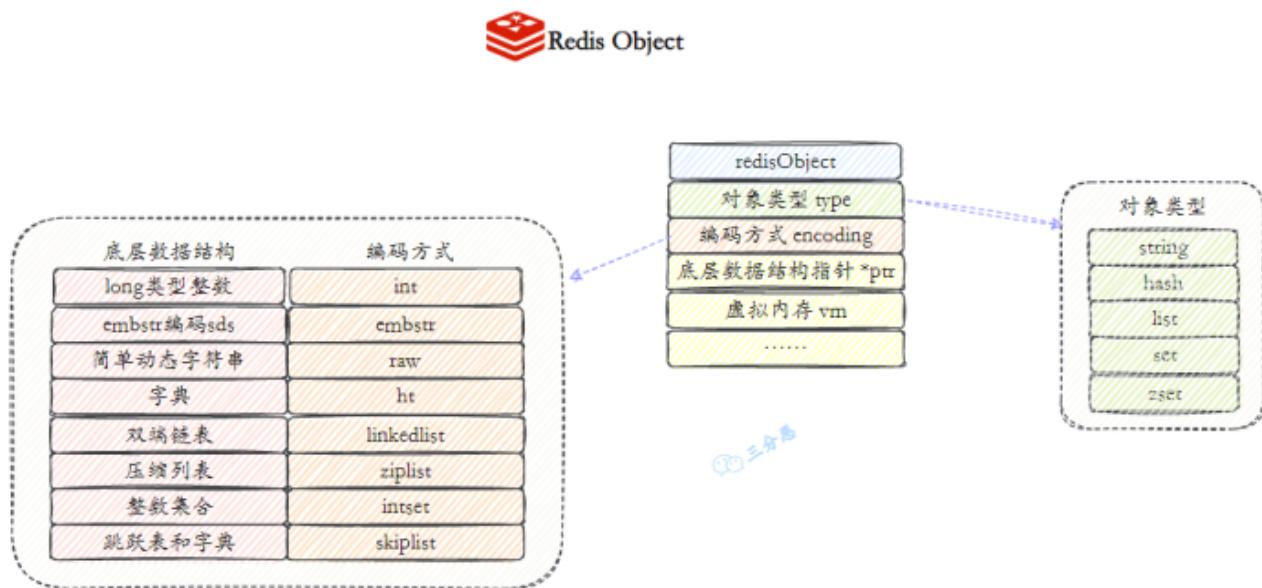
有序集合对象的编码有两种，分别是：`ziplist`、`skiplist`。

当有序集合的长度小于某个特定值，并且所有元素的长度都小于 64 字节时，使用压缩列表存储；否则使用 `skiplist` 存储。

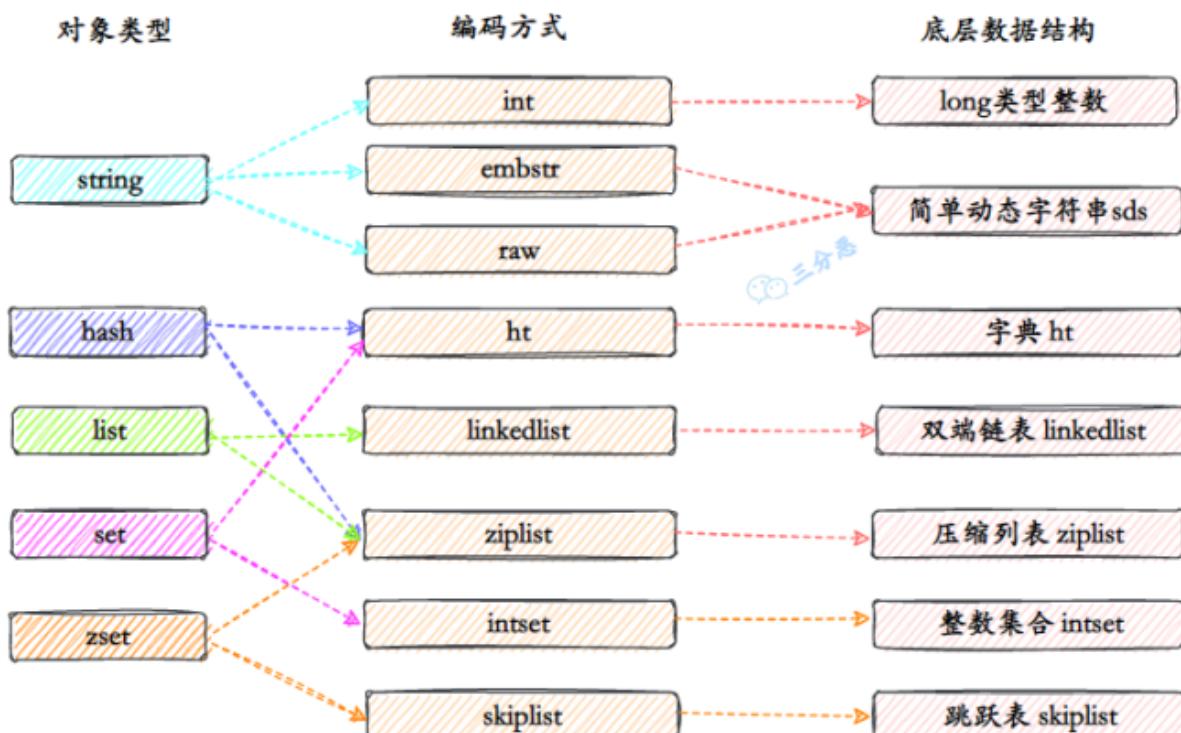
# 1. 说一说 redis 的底层数据结构

Redis有动态字符串(sds)、链表(list)、字典(ht)、跳跃表(skiplist)、整数集合(intset)、压缩列表(ziplist)等底层数据结构。

Redis并没有使用这些数据结构来直接实现键值对数据库，而是基于这些数据结构创建了一个对象系统，来表示所有的key-value。

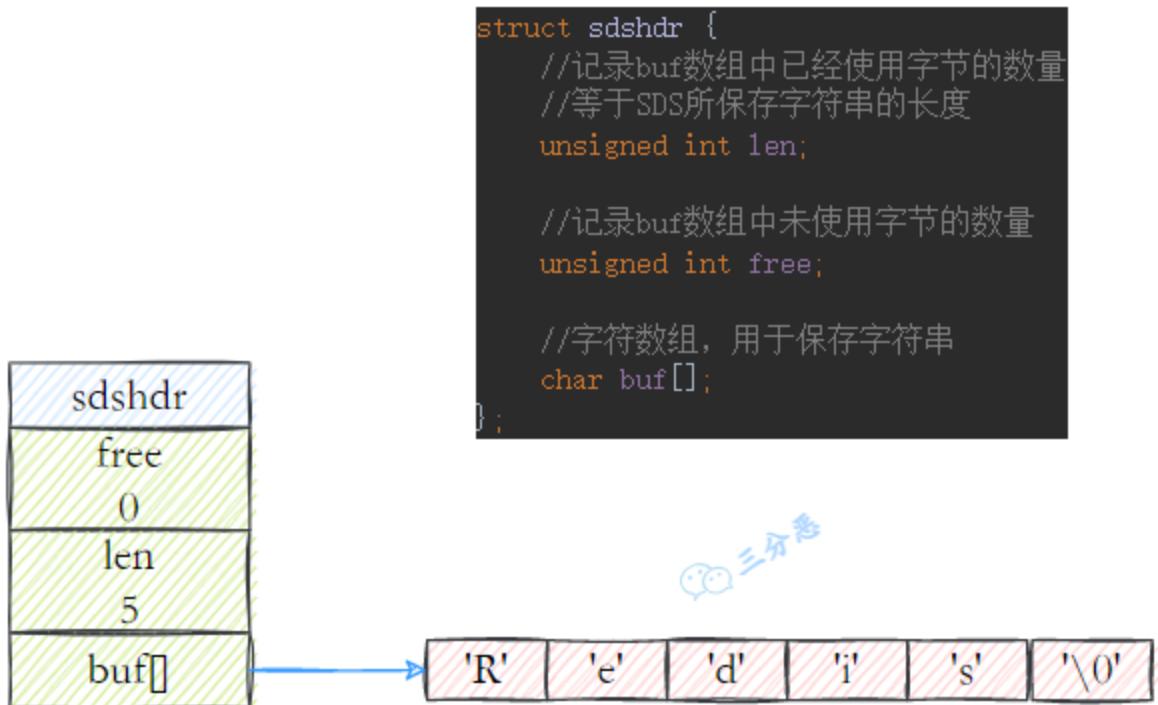


我们常用的数据类型和编码对应的映射关系：



## 1.1 字符串

redis 没有直接使用 C 语言传统的字符串表示，而是自己实现的叫做**简单动态字符串SDS**的抽象类型。C 语言的字符串不记录自身的长度信息，而 SDS 则保存了长度信息，这样将获取字符串长度的时间由  $O(N)$  降低到了  $O(1)$ ，同时可以避免缓冲区溢出和减少修改字符串长度时所需的内存重分配次数。



## 1.2 链表linkedlist

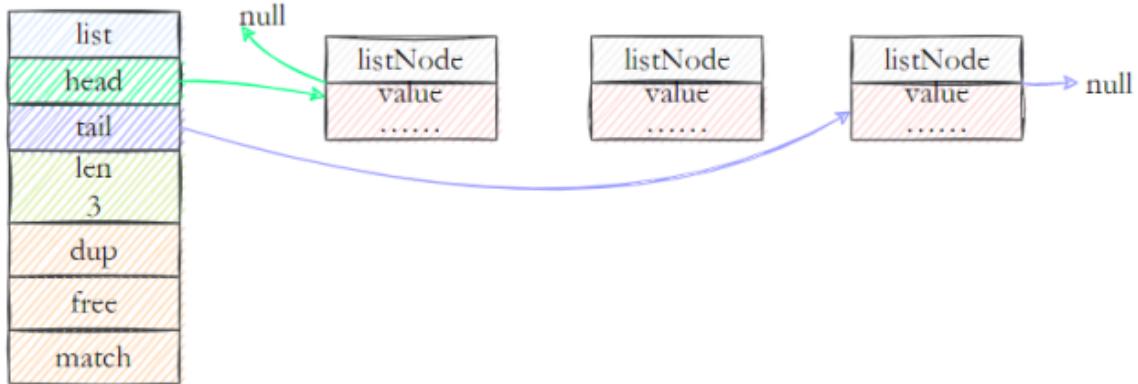
redis 链表是一个双向无环链表结构，每个链表的节点由一个 `listNode` 结构来表示，**每个节点都有指向前置节点和后置节点的指针**，同时表头节点的前置和表尾的后置节点都指向NULL。

### 链表节点

```
typedef struct listNode {  
    //前驱节点  
    struct listNode *prev;  
    //后继节点  
    struct listNode *next;  
    //节点的值  
    void *value;  
} listNode;
```

### 链表类型

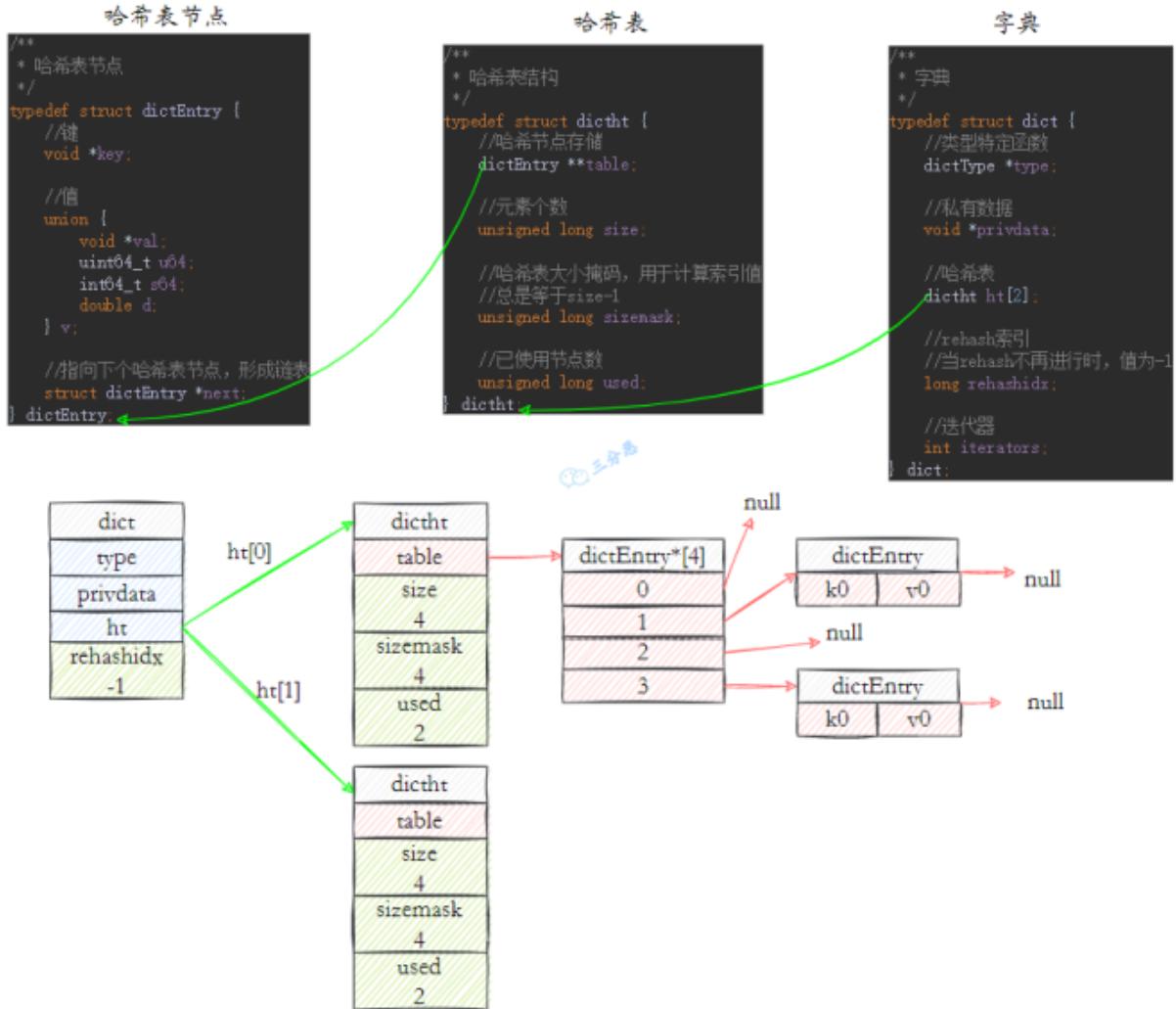
```
typedef struct list {  
    //链表头节点  
    listNode *head;  
    //链表尾节点  
    listNode *tail;  
    //节点值复制函数  
    void *(*dup)(void *ptr);  
    //节点值释放函数  
    void (*free)(void *ptr);  
    //节点值比对函数  
    int (*match)(void *ptr, void *key);  
    //链表包含的节点数量  
    unsigned long len;  
} list;
```



### 1.3 哈希表 hashtable

一个哈希表里可以有多个哈希表节点，而每个哈希表节点就保存了字典里的一个键值对。每个字典带有两个hash表，供平时使用和 rehash 时使用，**hash表使用链地址法来解决键冲突，被分配到同一个索引位置的多个键值对会形成一个单向链表，在对hash表进行扩容或者缩容的时候，**

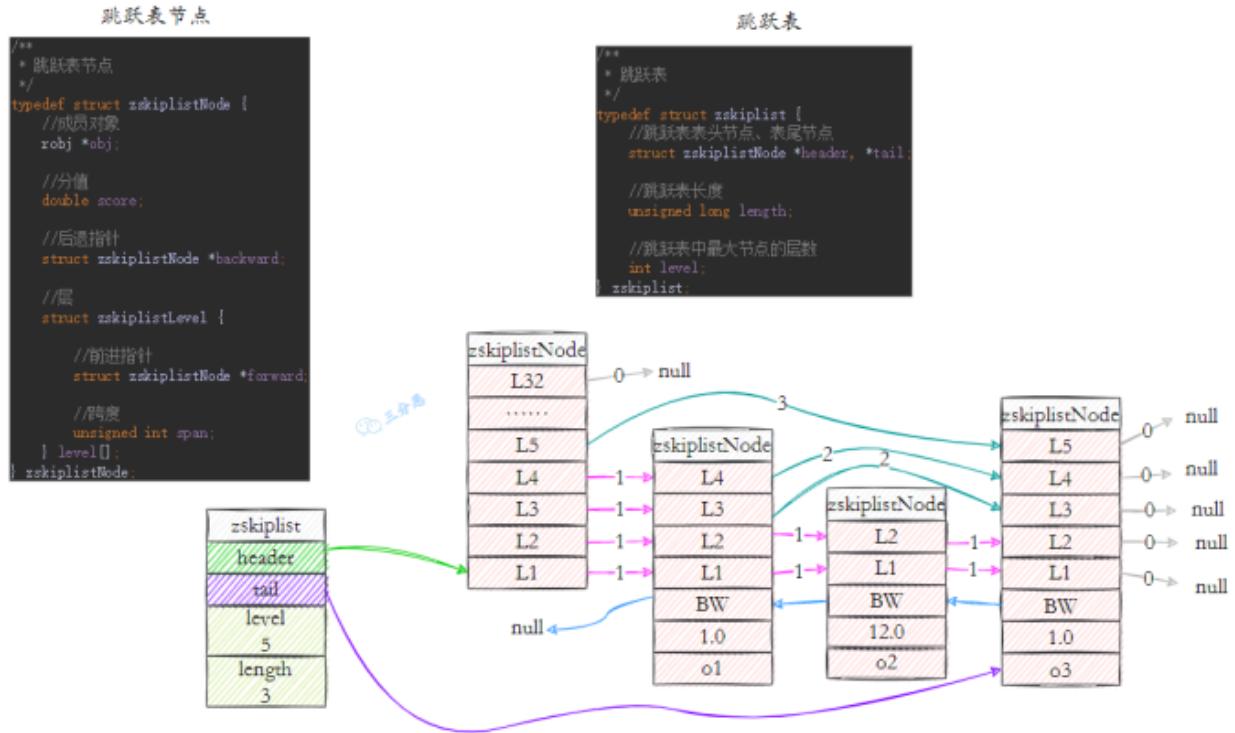
为了服务的可用性，rehash的过程不是一次性完成的，而是渐进式的。



## 1.4 跳跃表skiplist

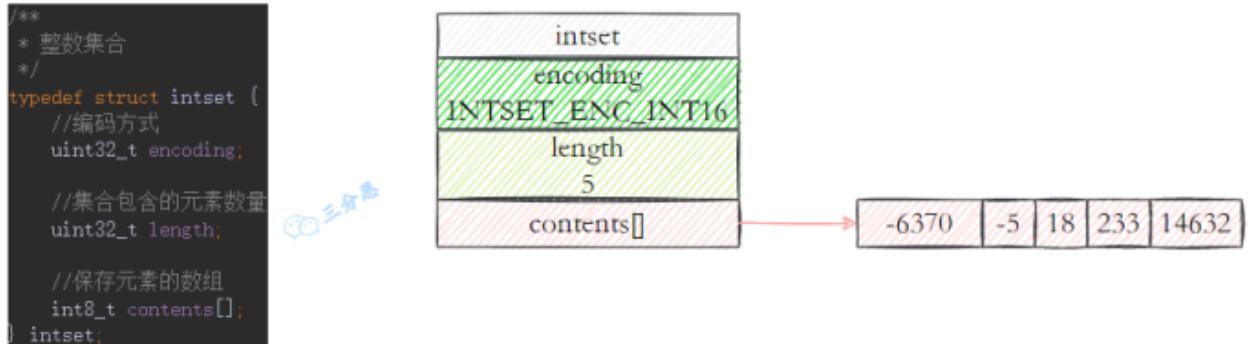
Redis跳跃表由 zskiplist 和 zskiplistNode 组成，zskiplist 用于保存跳跃表信息（表头、表尾节点、长度等），zskiplistNode 用于表示表跳跃节点，**每个跳跃表节点的层高都是 1-32 的随机数**，在同一个跳跃表中，多个节点可以包含相同的分值，但是每个节点的成员对象必须是唯一的，

节点按照分值大小排序，如果分值相同，则按照成员对象的大小排序。



## 1.5 整数集合intset

用于保存整数值的集合抽象数据结构，不会出现重复元素，底层实现为数组。



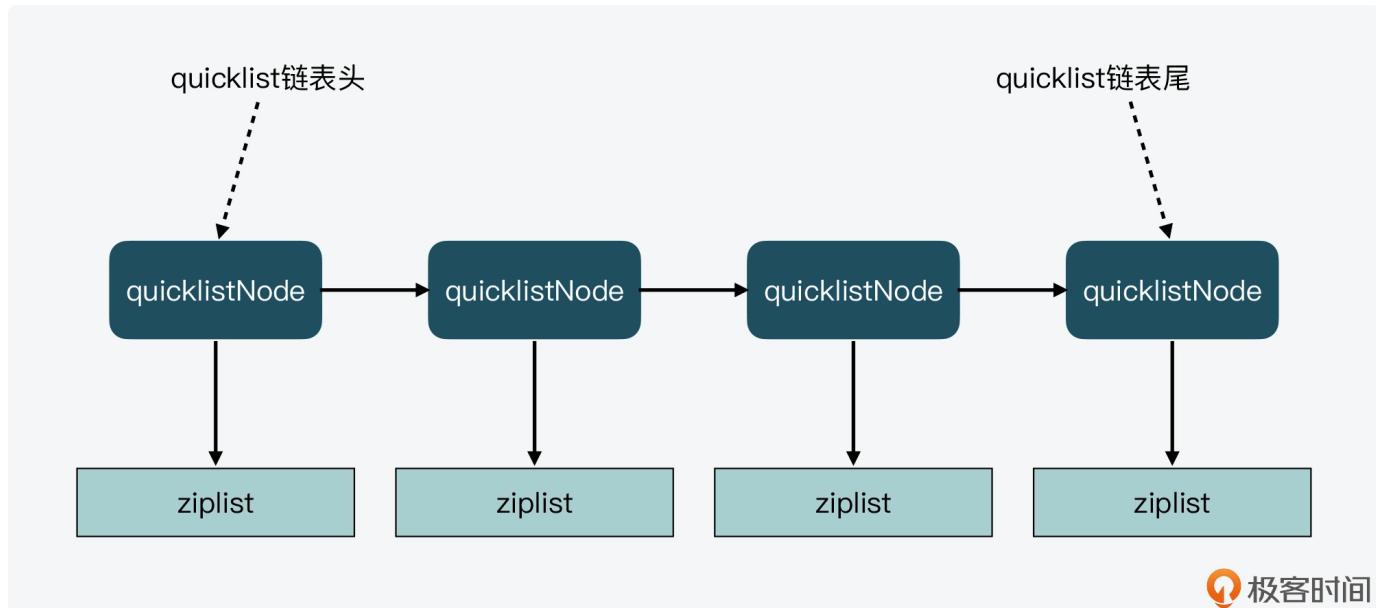
**1.6 压缩列表ziplist:** 压缩列表是为节约内存而开发的顺序性数据结构，它可以包含任意多个节点，每个节点可以保存一个字节数组或者整数值。

zbytes	zltail	zllen	entry1	entry2	.....	entryN	zlen
--------	--------	-------	--------	--------	-------	--------	------

## 1.7 quicklist (3.2)

<https://github.com/redis/redis/blob/unstable/src/quicklist.h#L46>

quicklist 的设计，其实是结合了链表和 ziplist 各自的优势。简单来说，一个 quicklist 就是一个链表，而链表中的每个元素又是一个 ziplist。





复制代码

```
1  typedef struct quicklist {
2      quicklistNode *head;
3      quicklistNode *tail;
4      unsigned long count;           /* total count of all entries in all zipli-
5      sts */
6      unsigned long len;            /* number of quicklistNodes */
7      int fill : 16;              /* fill factor for individual nodes */
8      unsigned int compress : 16; /* depth of end nodes not to compress; 0=of-
9      f */
10 } quicklist;
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31 
```

- quicklist 是一个双向链表， head、 tail分别指向头尾节点
- quicklistNode 是双向链表的节点， prev、 next分别指向前驱、后继结点
- quicklistNode.zl 指向一个ziplist（或者quicklistLZF结构）
- quicklistEntry 包裹着list的每一个值，作为ziplist的一个节点

可以想象得到，当一个空的quicklist加入一个值value时，会有以下操作（不一定以这个顺序）：

1. 使用Entry包裹value

2. 创建一个ziplist，把Entry加入到ziplist中
3. 创建一个Node，Node.zl指向ziplist
4. 创建quicklist，将Node加入quicklist中

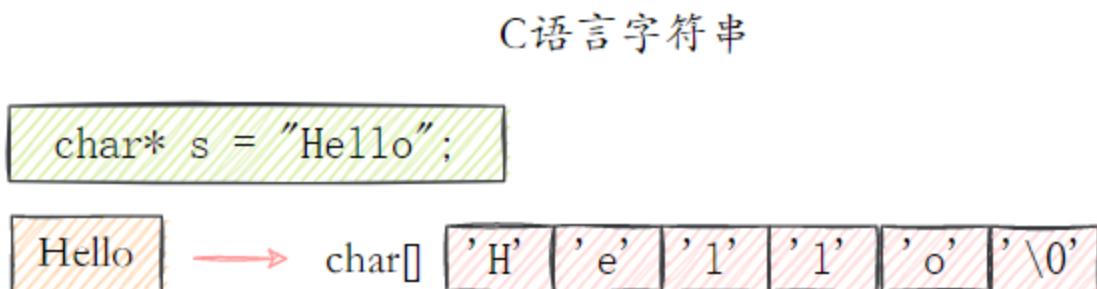
## 1.8 listpack (5.0)

而 Redis 除了设计了 quicklist 结构来应对 ziplist 的问题以外，还在 5.0 版本中新增了 listpack 数据结构，用来彻底避免连锁更新。

listpack 也叫紧凑列表，它的特点就是用一块连续的内存空间来紧凑地保存数据，同时为了节省内存空间，listpack 列表项使用了多种编码方式，来表示不同长度的数据，这些数据包括整数和字符串。

## 2. Redis 的 SDS 和 C 中字符串相比有什么优势？

C 语言使用了一个长度为  $N+1$  的字符数组来表示长度为  $N$  的字符串，并且字符数组最后一个元素总是 `\0`，这种简单的字符串表示方式不符合 Redis 对字符串在安全性、效率以及功能方面的要求。



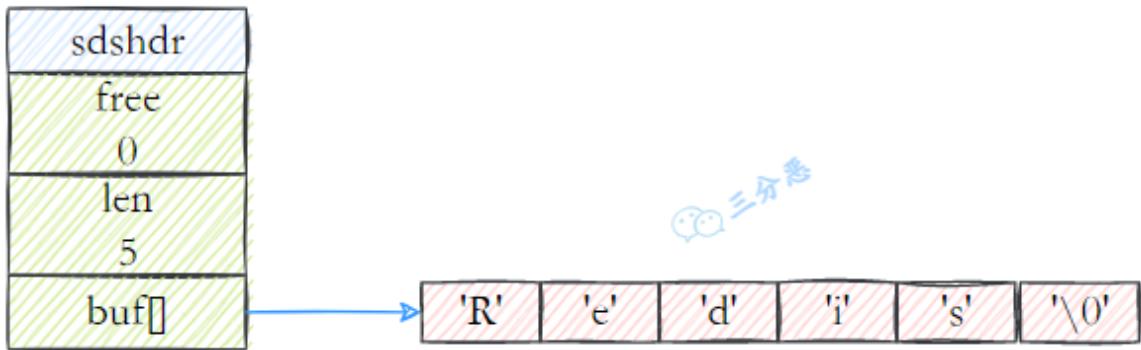
C语言的字符串可能有什么问题？

这样简单得数据结构可能会造成以下一些问题：

- **获取字符串长度复杂度高**：因为 C 不保存数组的长度，每次都需要遍历一遍整个数组，时间复杂度为  $O(n)$ ；
- **不能杜绝 缓冲区溢出/内存泄漏** 的问题：C字符串不记录自身长度带来的另外一个问题是容易造成缓存区溢出（buffer overflow），例如在字符串拼接的时候，新的
- **C 字符串 只能保存文本数据** → 因为 C 语言中的字符串必须符合某种编码（比如 ASCII），

例如中间出现的 '\0' 可能会被判定为提前结束的字符串而识别不了；

Redis如何解决？优势？



简单来说一下 Redis 如何解决的：

1. 多增加 len 表示当前字符串的长度：这样就可以直接获取长度了，复杂度 O(1)；
2. 自动扩展空间：当 SDS 需要对字符串进行修改时，首先借助于 len 和 alloc 检查空间是否满足修改所需的要求，如果空间不够的话，SDS 会自动扩展空间，避免了像 C 字符串操作中的溢出情况；
3. 有效降低内存分配次数：C 字符串在涉及增加或者清除操作时会改变底层数组的大小造成重新分配，SDS 使用了 空间预分配 和 惰性空间释放 机制，简单理解就是每次在扩展时是成倍的多分配的，在缩容时也是先留着并不正式归还给 OS；
4. 二进制安全：C 语言字符串只能保存 ascii 码，对于图片、音频等信息无法保存，SDS 是二进制安全的，写入什么读取就是什么，不做任何过滤和限制；

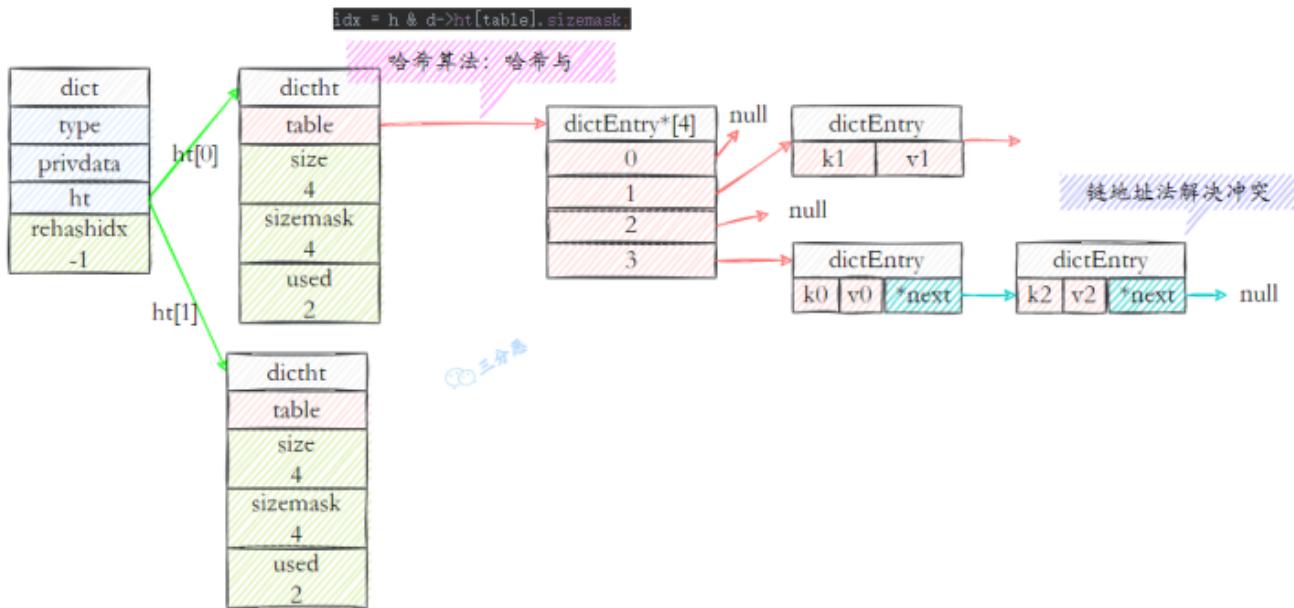
### 3. 字典是如何实现的？Rehash 了解吗？

字典是 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 也是一个字典。（存储在 RedisDb 数据结构中）

字典结构是什么样的呢？

Redis 中的字典相当于 HashMap，内部实现也差不多类似，采用哈希与运算计算下标位置；通过 "数组 + 链表" 的链地址法 来解决哈希冲突，同时这样的结构也吸收了两种不同数据结构的优

点



字典是怎么扩容的?

字典结构内部包含 **两个 hashtable**，通常情况下只有一个哈希表 `ht[0]` 有值，在扩容的时候，把 `ht[0]` 里的值 `rehash` 到 `ht[1]`，然后进行 **渐进式rehash**——所谓渐进式 `rehash`，指的是这个 `rehash` 的动作并不是一次性、集中式地完成的，而是分多次、渐进式地完成的。

待搬迁结束后，`ht[1]` 就取代 `ht[0]` 存储字典的元素。

- 1) 为 `ht[1]` 分配空间，让字典同时持有 `ht[0]` 和 `ht[1]` 两个哈希表。
- 2) 在字典中维持一个索引计数器变量 `rhashidx`，并将它的值设置为 0，表示 `rehash` 工作正式开始。
- 3) 在 `rehash` 进行期间，每次对字典执行添加、删除、查找或者更新操作时，程序除了执行指定的操作以外，还会顺带将 `ht[0]` 哈希表在 `rhashidx` 索引上的所有键值对 `rehash` 到 `ht[1]`，当 `rehash` 工作完成之后，程序将 `rhashidx` 属性的值增一。
- 4) 随着字典操作的不断执行，最终在某个时间点上，`ht[0]` 的所有键值对都会被 `rehash` 至 `ht[1]`，这时程序将 `rhashidx` 属性的值设为 -1，表示 `rehash` 操作已完成。

## 4. 跳跃表是如何实现的? 原理?

PS:跳跃表是比较常问的一种结构。

什么是跳跃表

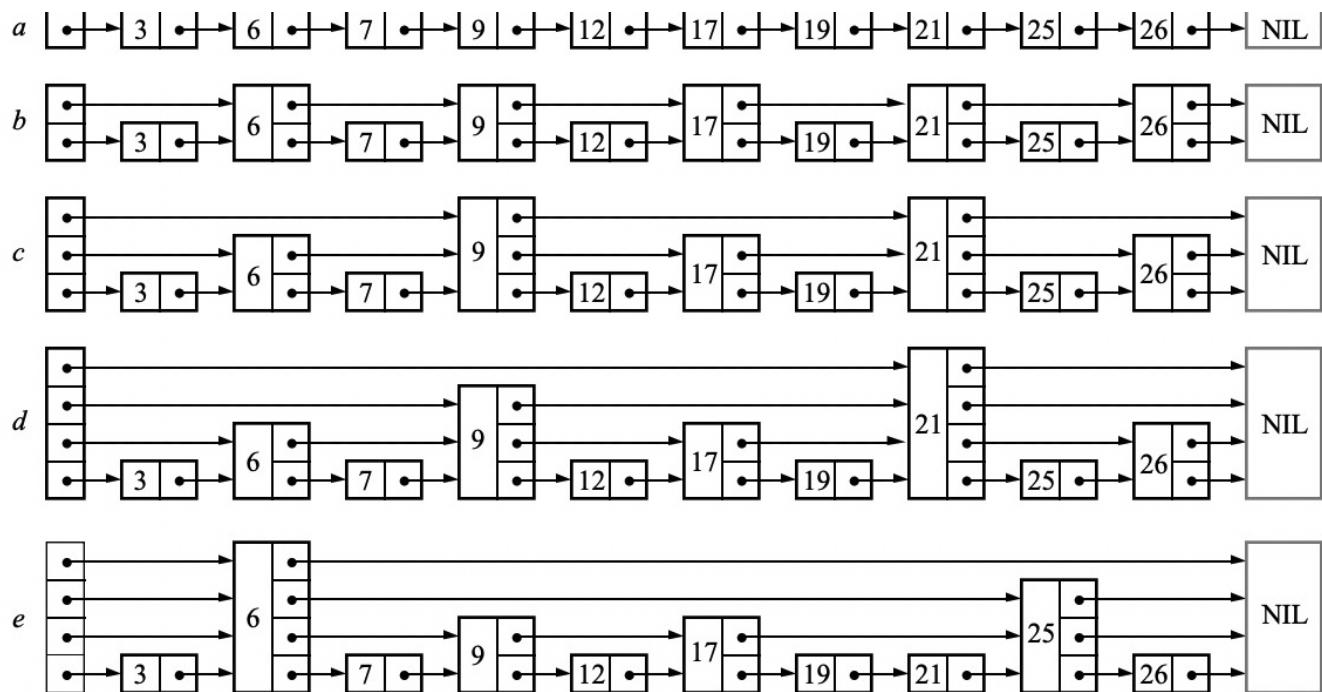


-

- 跳跃表是一种可以用来代替平衡树的数据结构。
- 跳跃表使用概率平衡，而不是严格强制的平衡，
- 跳跃表中插入和删除的算法比平衡树的等效算法简单得多，速度也快得多

| 实质上就是一种可以进行二分查找的有序链表。而二分查找的基础就是分层索引。

eg: 论文第二页给的图



**核心解决高效 ( $O(N) \rightarrow O(\log_2 N)$ ) 查找问题**

## Initialization

An element NIL is allocated and given a key greater than any legal key. All levels of all skip lists are terminated with NIL.

A new list is initialized so that the the *level* of the list is equal to 1 and all forward pointers of the list's header point to NIL.

## Search Algorithm

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for (Figure 2). When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

查找流程：

```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
            -- x→key < searchKey ≤ x→forward[1]→key
        x := x→forward[1]
        if x→key = searchKey then return x→value
        else return failure
```

FIGURE 2 - Skip list search algorithm

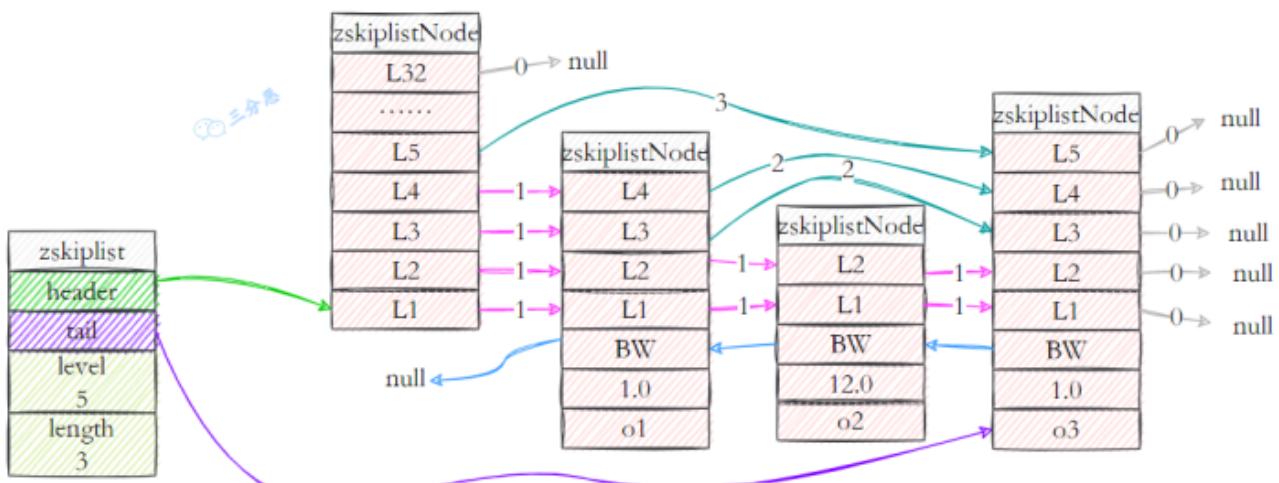
在头领节点的最高楼沿着跨度向前寻找，如果刚好找到所需元素，则直接返回，否则继续往前寻找，知道遇到比寻找的元素大的元素，然后返回一个跨度，并下一层楼寻找。如果在一楼也找不到说明元素不存在。

每个楼层至少有两个元素，跨度和前向指针。

eg: 插入，删除

```
Insert(list, searchKey, newValue)
  local update[1..MaxLevel]
  x := list→header
  for i := list→level downto 1 do
    while x→forward[i]→key < searchKey do
      x := x→forward[i]
      -- x→key < searchKey ≤ x→forward[i]→key
      update[i] := x
  x := x→forward[1]
  if x→key = searchKey then x→value := newValue
  else
    lvl := randomLevel()
    if lvl > list→level then
      for i := list→level + 1 to lvl do
        update[i] := list→header
      list→level := lvl
  x := makeNode(lvl, searchKey, value)
  for i := 1 to level do
    x→forward[i] := update[i]→forward[i]
    update[i]→forward[i] := x
```

跳跃表 (skiplist) 是一种有序数据结构，它通过在每个节点中维持多个指向其它节点的指针，从而达到快速访问节点的目的。



## 为什么使用跳跃表？

首先，因为 zset 要支持随机的插入和删除，所以它 不宜使用数组来实现，关于排序问题，我们也很容易就想到 红黑树/ 平衡树 这样的树形结构，为什么 Redis 不使用这样一些结构呢？

1. **性能考虑：** 在高并发的情况下，树形结构需要执行一些类似于 rebalance 这样的可能涉及整棵树的操作，相对来说跳跃表的变化只涉及局部；

2. **实现考虑：** 在复杂度与红黑树相同的情况下，跳跃表实现起来更简单，看起来也更加直观；

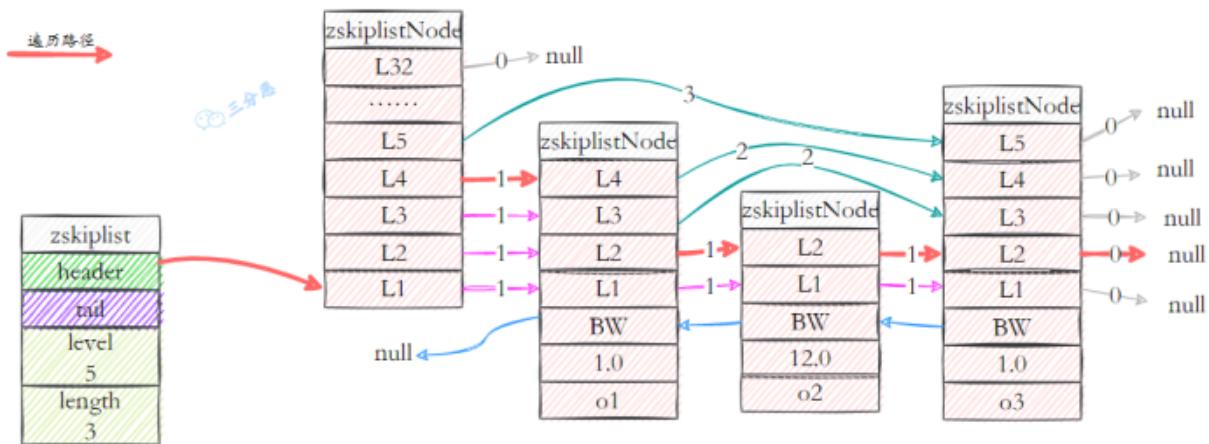
基于以上的一些考虑，Redis 基于 William Pugh 的论文做出一些改进后采用了 跳跃表 这样的结构。

本质是解决查找问题。

## 跳跃表是怎么实现的？

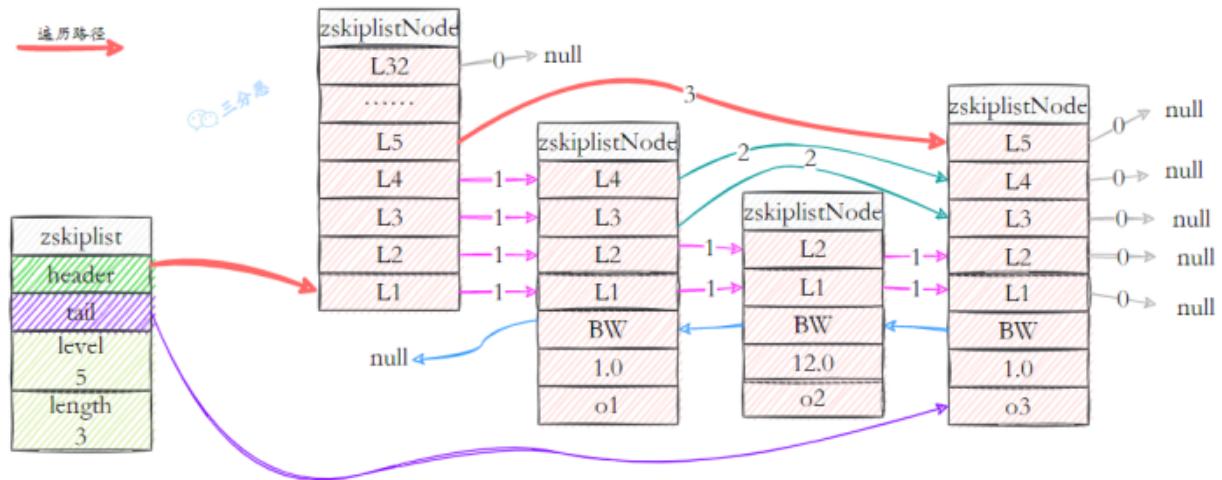
跳跃表的节点里有这些元素：

- **层** 跳跃表节点的level数组可以包含多个元素，每个元素都包含一个指向其它节点的指针，程序可以通过这些层来加快访问其它节点的速度，一般来说，层的数量越多，访问其它节点的速度就越快。每次创建一个新的跳跃表节点的时候，程序都根据幂次定律，随机生成一个介于1和32之间的值作为level数组的大小，这个大小就是层的“高度”
- **前进指针** 每个层都有一个指向表尾的前进指针（level[i].forward属性），用于从表头向表尾方向访问节点。我们看一下跳跃表从表头到表尾，遍历所有节点的路径：



- **跨度** 层的跨度用于记录两个节点之间的距离。跨度是用来计算排位 (rank) 的：在查找某个节点的过程中，将沿途访问过的所有层的跨度累计起来，得到的结果就是目标节点在跳跃表中的排位。例如查找，分值为3.0、成员对象为o3的节点时，沿途经历的层：查找的过程只经过

了一个层，并且层的跨度为3，所以目标节点在跳跃表中的排位为3。



- 分值和成员 节点的分值（score属性）是一个double类型的浮点数，跳跃表中所有的节点都按分值从小到大来排序。节点的成员对象（obj属性）是一个指针，它指向一个字符串对象，而字符串对象则保存这一个SDS值。

## 5. 压缩列表了解吗？

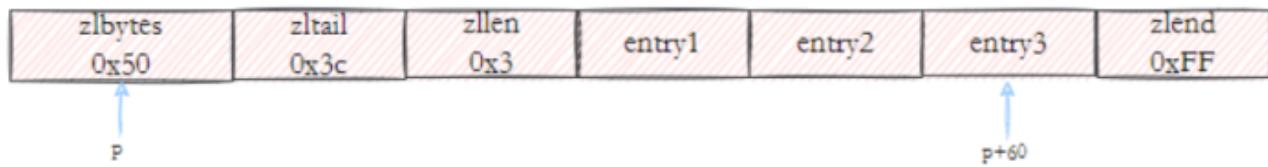
压缩列表是 Redis 为了节约内存 而使用的一种数据结构，是由一系列特殊编码的连续内存快组成的顺序型数据结构。

一个压缩列表可以包含任意多个节点（entry），**每个节点可以保存一个字节数组或者一个整数值**。



压缩列表由这么几部分组成：

- zbytes**:记录整个压缩列表占用的内存字节数
- ztail**:记录压缩列表表尾节点距离压缩列表的起始地址有多少字节
- zlen**:记录压缩列表包含的节点数量
- entryX**:列表节点
- zlenend**:用于标记压缩列表的末端



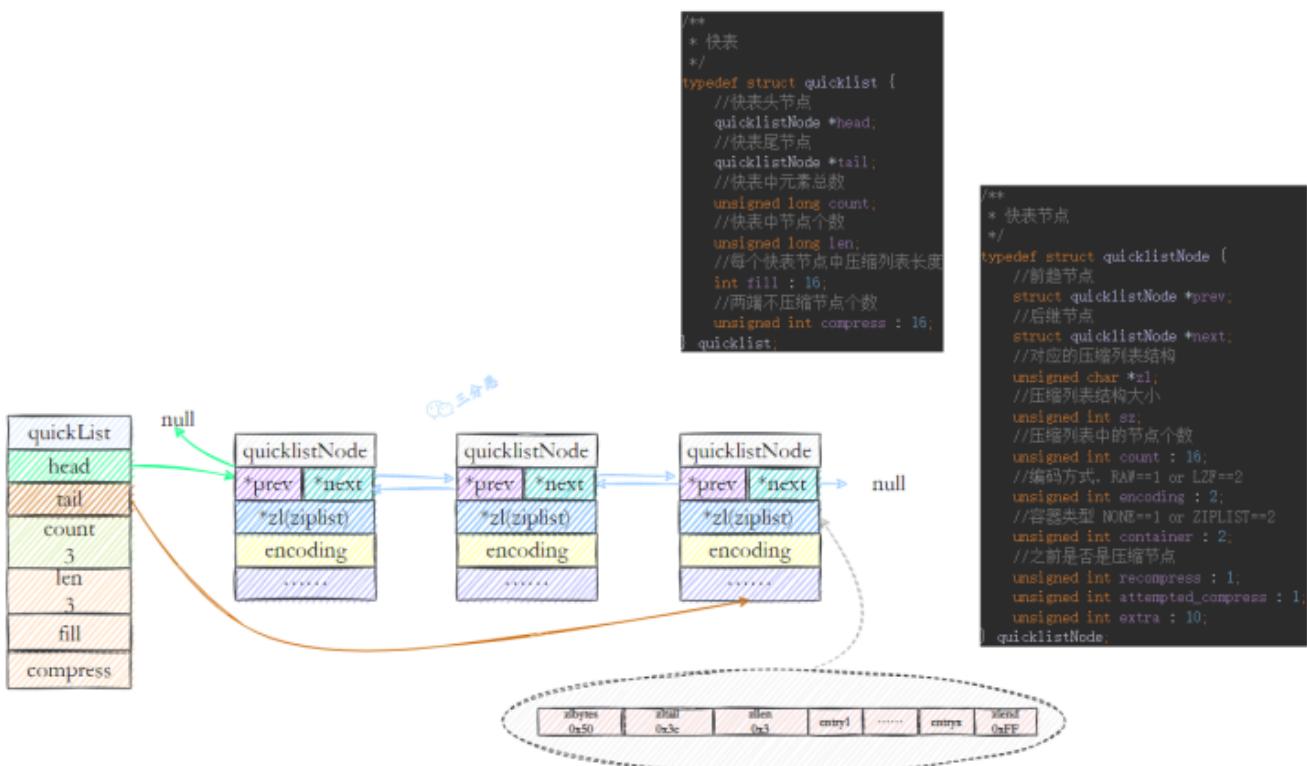
## 6. 快速列表 quicklist 了解吗？

Redis 早期版本存储 list 列表数据结构使用的是压缩列表 ziplist 和普通的双向链表 linkedlist，也就是说当元素少时使用 ziplist，当元素多时用 linkedlist。

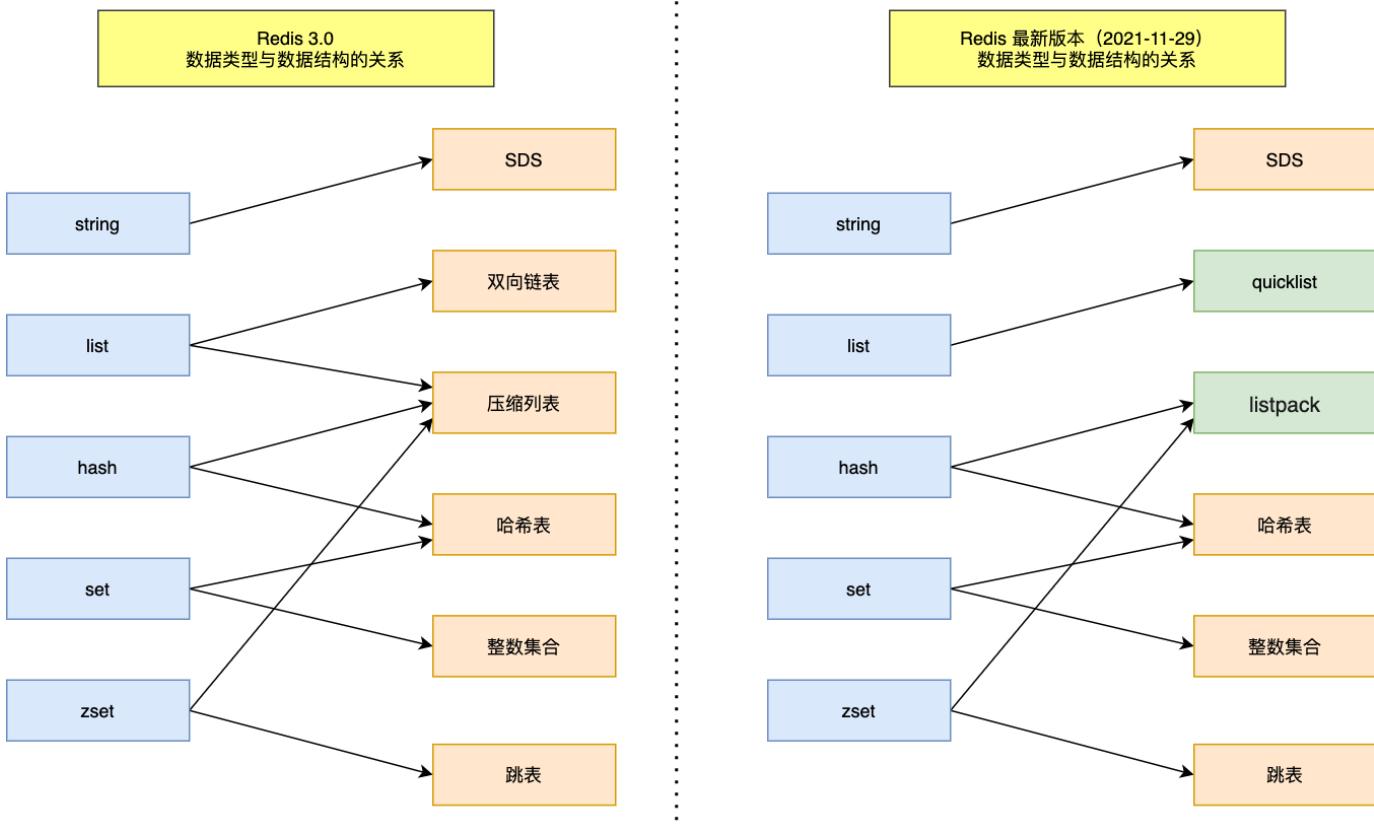
但考虑到链表的附加空间相对较高，prev 和 next 指针就要占去 16 个字节（64 位操作系统占用 8 个字节），另外每个节点的内存都是单独分配，会家具内存的碎片化，影响内存管理效率。

后来 Redis 新版本（3.2）对列表数据结构进行了改造，使用 quicklist 代替了 ziplist 和 linkedlist，quicklist 是综合考虑了时间效率与空间效率引入的新型数据结构。

quicklist 由 list 和 ziplist 结合而成，它是一个由 ziplist 充当节点的双向链表。



## 7. 数据类型的实现



## 8. 什么是空间预分配以及惰性空间释放，SDS 是怎么实现的

首先，SDS 主要通过 未使用空间 来实现的。

- 空间预分配，在 SDS 中的表现是，如果对 SDS 的 **长度** 进行修改，程序会多分配一部分的未使用空间，至于分配多少取决于 `len` 属性，当 `len < 1MB` 时，会分配 `len` 的未使用空间；否则分配 `1MB`。

如果未使用空间，足以放下修改后的字符，`len` 是不会发生变化的。也就是说，空间的预分配把字符的重分配次数从 N 次，降为 最多 N 次。

- 惰性空间释放，在 SDS 中的表现是，当字符串缩短时，并不立即释放空间，而是保留为未使用空间，留待后续使用。（当然也可以手动释放）。

## 9. 为什么说 SDS 是二进制安全的呢

因为 SDS 主要利用 `buffer + len + free`，表示字符串，由底层的字符数组 `buf` 存储字符，`len` 表示有效的字符长度，程序不会对 `buffer` 中数据进行任何的限制、过滤或者假设等操作。

## 10. 说说 redis 里的对象

对象也就是 redis 在底层数据结构之上包装的一层 RedisObject（对象），常见的 RedisObject 有五种：字符串对象、列表对象、哈希对象、集合对象和有序集合对象。

## 11. 使用 RedisObject 的好处

使用 RedisObject 的优点主要有两个，分别是：

1. 通过不同类型的对象，Redis 可以在执行命令之前，根据对象的类型来判断一个对象是否可以执行给定的命令。
2. 我们可以针对不同的使用场景，为对象设置不同的实现，从而优化内存或查询速度。

## 12. RedisObject 的具体结构是什么

```
1 typedef struct redisObject {  
2     // 类型  
3     unsigned type:4;  
4     // 编码  
5     unsigned encoding:4;  
6     // 指向实际值的指针  
7     void *ptr;  
8     ...  
9 } robj;  
11
```



复制代码

## 持久化

# 1. 什么是Redis持久化？

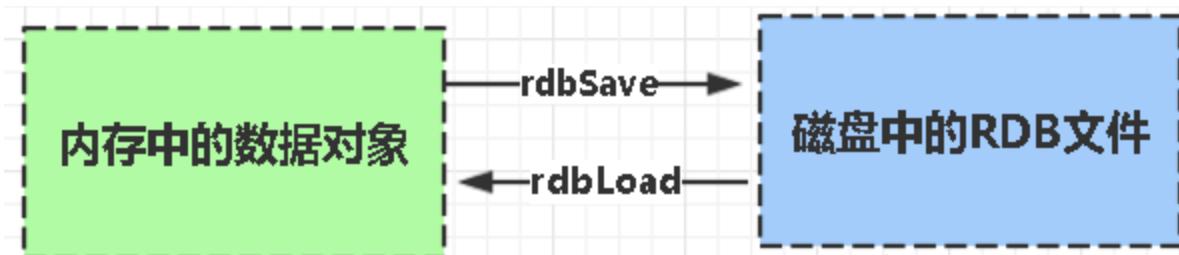
持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

## 2. Redis 的持久化机制是什么？各自的优缺点？

Redis 提供两种持久化机制 RDB（默认） 和 AOF 机制。

### 2.1 RDB (Redis DataBase) , 快照

RDB是Redis默认的持久化方式。按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为 `dump.rdb`。通过配置文件中的`save`参数来定义快照的周期。



优点：

- 1. 只有一个文件 `dump.rdb`, 方便持久化。
- 2. 容灾性好，一个文件可以保存到安全的磁盘。
- 3. 性能最大化，`fork` 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能
- 4. 相对于数据集大时，比 AOF 的启动效率更高。

缺点：

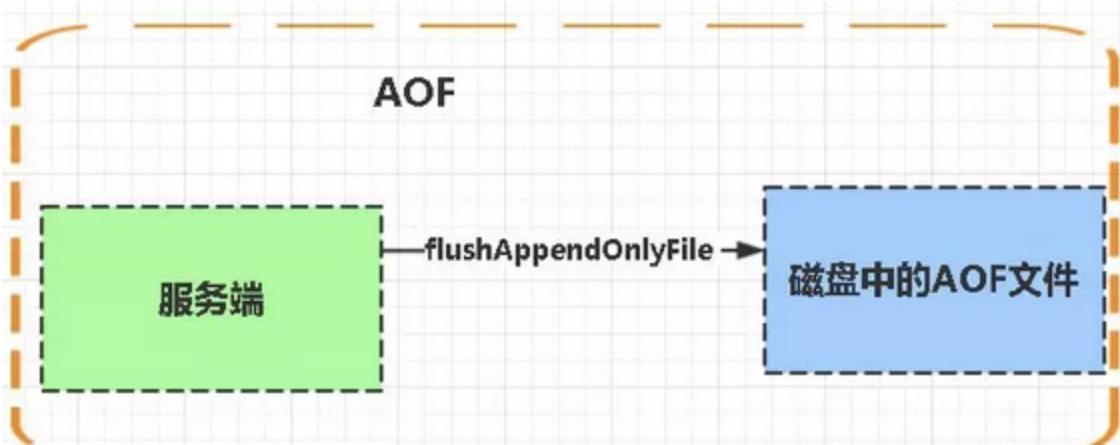
- 1. 数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)

### 2.2 AOF (Append Only File) , 日志

AOF持久化(即Append Only File持久化)，则是将Redis执行的每次写命令记录到单独的日志文件中，当重启Redis会重新将持久化的日志中文件恢复数据。

所有的命令行记录以 redis 命令请求协议的格式完全持久化存储)保存为 `aof` 文件。

当两种方式同时开启时，数据恢复Redis会优先选择AOF恢复。



优点：

- 1、数据安全，aof 持久化可以配置 `appendfsync` 属性，有 always，每进行一次 命令操作就记录到 aof 文件中一次。
- 2、通过 append 模式写文件，即使中途服务器宕机，可以通过 `redis-check-aof` 工具解决数据一致性问题。
- 3、AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（文件过大时会对命令 进行合并重写），可以删除其中的某些命令（比如误操作的 flushall））

缺点：

- 1、AOF 文件比 RDB 文件大，且恢复速度慢。
- 2、数据集大的时候，比 rdb 启动效率低。

### 3. 优缺点是什么？

- AOF文件比RDB更新频率高，优先使用AOF还原数据。
- AOF比RDB更安全也更大
- RDB性能比AOF好
- 如果两个都配了优先加载AOF

### 4. 如何选择合适的持久化方式

- 如果极度关心数据安全性，你应该同时使用两种持久化功能。在这种情况下，当 Redis 重启的时候会优先载入AOF 文件来恢复原始的数据，因为在通常情况下AOF文件保存的数据集要比RDB文件保

存的数据集要完整。

- 如果非常关心你的数据，但仍然可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。
- 如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。
- 有很多用户都只使用 AOF 持久化，但并不推荐这种方式，因为定时生成 RDB 快照 (snapshot) 非常便于进行数据库备份，并且 RDB 恢复数据集的速度也要比AOF恢复的速度要快，除此之外，使用 RDB 还可以避免 AOF 程序的 bug 。

## 5. Redis持久化数据和缓存怎么做扩容？

- 如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。
- 如果Redis被当做一个持久化存储使用，必须使用固定的 keys-to-nodes 映射关系，节点的数量一旦确定不能变化。否则的话( 即Redis节点需要动态变化的情况 )，必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。

## 6. Redis 怎么确保 Aof 不丢失

首先，要 AOF 完全不丢失，是不可能的，因为 aof 的写回策略只有三种， always, everySec, no ，就算启用的是 always ，由于 redis 是写后日志，当数据写入成功的瞬间，redis 实例宕机了，依然会导致 aof 丢失，但是此时， redis 没有响应，所以可以认为 aof 不曾丢失（因为业务没有成功响应）。

## 7. Aof 日志是写前还是写后日志

说到日志，我们比较熟悉的是数据库的写前日志 (Write Ahead Log, WAL) ，也就是说，在实际写数据前，先把修改的数据记到日志文件中，以便故障时进行恢复。不过，AOF日志正好相反，它是写后日志，“写后”的意思是Redis是先执行命令，把数据写入内存，然后才记录日志。

## 8. AOF为什么要先执行命令再记日志

要回答这个问题，我们要先知道AOF里记录了什么内容。

传统数据库的日志，例如redo log（重做日志），记录的是修改后的数据，而AOF里记录的是Redis收到的每一条命令，这些命令是以文本形式保存的。

我们以Redis收到“`set testkey testvalue`”命令后记录的日志为例，看看 AOF 日志的内容。其中，“`*3`”表示当前命令有三个部分，每部分都是由“`$+数字`”开头，后面紧跟着具体的命令、键或值。这里，“数字”表示这部分中的命令、键或值一共有多少字节。例如，“`$3 set`”表示这部分有3个字节，也就是“`set`”命令。

最重要的原因是 Redis 作为一个内存数据库追求极致的性能。

为了避免额外的检查开销，Redis在向AOF里面记录日志的时候，并不会先去对这些命令进行语法检查。所以，如果先记日志再执行命令的话，日志中就有可能记录了错误的命令，Redis在使用日志恢复数据时，就可能会出错。

而写后日志这种方式，就是先让系统执行命令，只有命令能执行成功，才会被记录到日志中，否则，系统就会直接向客户端报错。所以，Redis使用写后日志这一方式的一大好处是，可以避免出现记录错误命令的情况。

除此之外，AOF还有一个好处：它是在命令执行后才记录日志，所以不会阻塞当前的写操作。

## 9. Redis Aof 的风险

AOF也有两个潜在的风险。

首先，如果刚执行完一个命令，还没有来得及记日志就宕机了，那么这个命令和相应的数据就有丢失的风险。如果此时Redis是用作缓存，还可以从后端数据库重新读入数据进行恢复，但是，如果Redis是直接用作数据库的话，此时，因为命令没有记入日志，所以就无法用日志进行恢复了。

其次，AOF虽然避免了对当前命令的阻塞，但可能会给下一个操作带来阻塞风险。这是因为，AOF日志也是在主线程中执行的，如果在把日志文件写入磁盘时，磁盘写压力大，就会导致写盘很慢，进而导致后续的操作也无法执行了。

仔细分析的话，你就会发现，这两个风险都是和AOF写回磁盘的时机相关的。这也就意味着，如果我们能够控制一个写命令执行完后AOF日志写回磁盘的时机，这两个风险就解除了。

## 10. Redis 的写回策略

其实，对于上面的问题，AOF机制给我们提供了三个选择，也就是AOF配置项 `appendfsync` 的三个可选值。

- **Always**，同步写回：每个写命令执行完，立马同步地将日志写回磁盘；

- **Everysec**, 每秒写回: 每个写命令执行完, 只是先把日志写到AOF文件的内存缓冲区, 每隔一秒把缓冲区中的内容写入磁盘;
- **No**, 操作系统控制的写回: 每个写命令执行完, 只是先把日志写到AOF文件的内存缓冲区, **由操作系统决定何时将缓冲区内容写回磁盘**。

配置项	写回时机	优点	缺点
Always	同步写回	可靠性高, 数据基本不丢失	每个写命令都要落盘, 性能影响较大
Everysec	每秒写回	性能适中	宕机时丢失1秒内的数据
No	操作系统控制的写回	性能好	宕机时丢失数据较多

针对避免主线程阻塞和减少数据丢失问题, 这三种写回策略都无法做到两全其美。我们来分析下其中的原因。

- “同步写回”可以做到基本不丢数据, 但是它在每一个写命令后都有一个慢速的落盘操作, 不可避免地会影响主线程性能;
- 虽然“操作系统控制的写回”在写完缓冲区后, 就可以继续执行后续的命令, 但是落盘的时机已经不在Redis手中了, 只要AOF记录没有写回磁盘, 一旦宕机对应的数据就丢失了;
- “每秒写回”采用一秒写回一次的频率, 避免了“同步写回”的性能开销, 虽然减少了对系统性能的影响, 但是如果发生宕机, 上一秒内未落盘的命令操作仍然会丢失。所以, 这只能算是, 在避免影响主线程性能和避免数据丢失两者间取了个折中。

## 11. 如何选择回写策略

其实这是一个高性能以及高可靠的权衡问题。

想要获得高性能, 就选择 No 策略;

如果想要得到高可靠性保证, 就选择 Always 策略;

如果允许数据有一点丢失, 又希望性能别受太大影响的话, 那么就选择 Everysec 策略。

## 12. Redis 回写有什么性能问题

主要在于以下三个方面:

1. 文件系统本身对文件大小有限制，无法保存过大的文件；
2. 如果文件太大，之后再往里面追加命令记录的话，效率也会变低；
3. 如果发生宕机，AOF中记录的命令要一个个被重新执行，用于故障恢复，如果日志文件太大，整个恢复过程就会非常缓慢，这会影响到Redis的正常使用。

## 13. Aof 日志文件太大怎么办

可以重写Aof

## 14. redis 的重写机制了解吗

简单来说，AOF重写机制就是在重写时，Redis根据数据库的现状创建一个新的AOF文件，也就是说，读取数据库中的所有键值对，然后对每一个键值对用一条命令记录它的写入。比如说，当读取了键值对“testkey”: “testvalue”之后，重写机制会记录 set testkey testvalue 这条命令。这样，当需要恢复时，可以重新执行该命令，实现 “testkey”: “testvalue”的写入。

## 15. 为什么重写会使日志文件变小

主要是因为重写的过程中自动把过期数据筛除了，达到 多变一 的目的。

我们知道，AOF文件是以追加的方式，逐一记录接收到的写命令的。**当一个键值对被多条写命令反复修改时，AOF文件会记录相应的多条命令**。但是，在重写的时候，是根据这个键值对当前的最新状态，为它生成对应的写入命令。这样一来，一个键值对在重写日志中只用一条命令就行了，而且，在日志恢复时，只用执行这条命令，就可以直接完成这个键值对的写入了。

## 16. AOF 重写会堵塞主线程吗

和AOF日志由主线程写回不同，重写过程是由后台线程 `bgrewriteaof` 来完成的，这也是为了避免阻塞主线程，导致数据库性能下降。

## 17. 重写的过程

我把重写的过程总结为“一个拷贝，两处日志”。

在fork出子进程时的拷贝，以及在重写时，如果有新数据写入，主线程就会将命令记录到两个aof日志内存缓冲区中。

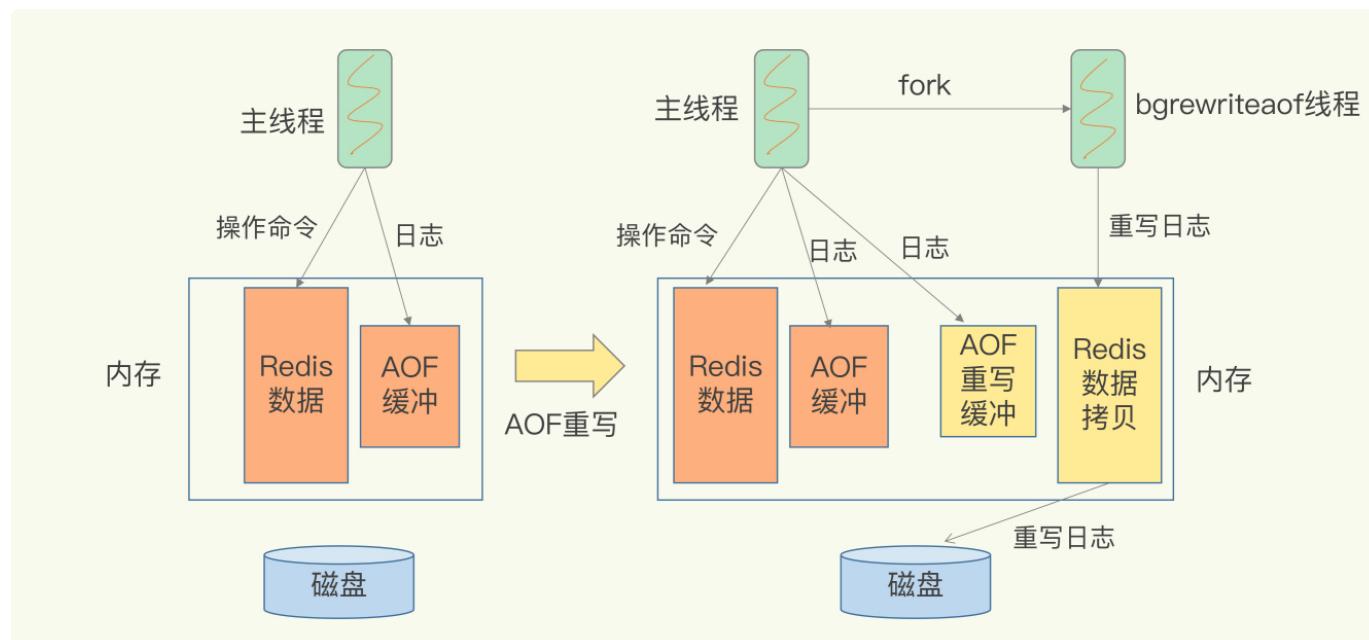
如果AOF写回策略配置的是always，则直接将命令写回旧的日志文件，并且保存一份命令至AOF重写缓冲区，这些操作对新的日志文件是不存在影响的。（**旧的日志文件：主线程使用的日志文件，新的日志文件：bgrewriteaof进程使用的日志文件**）

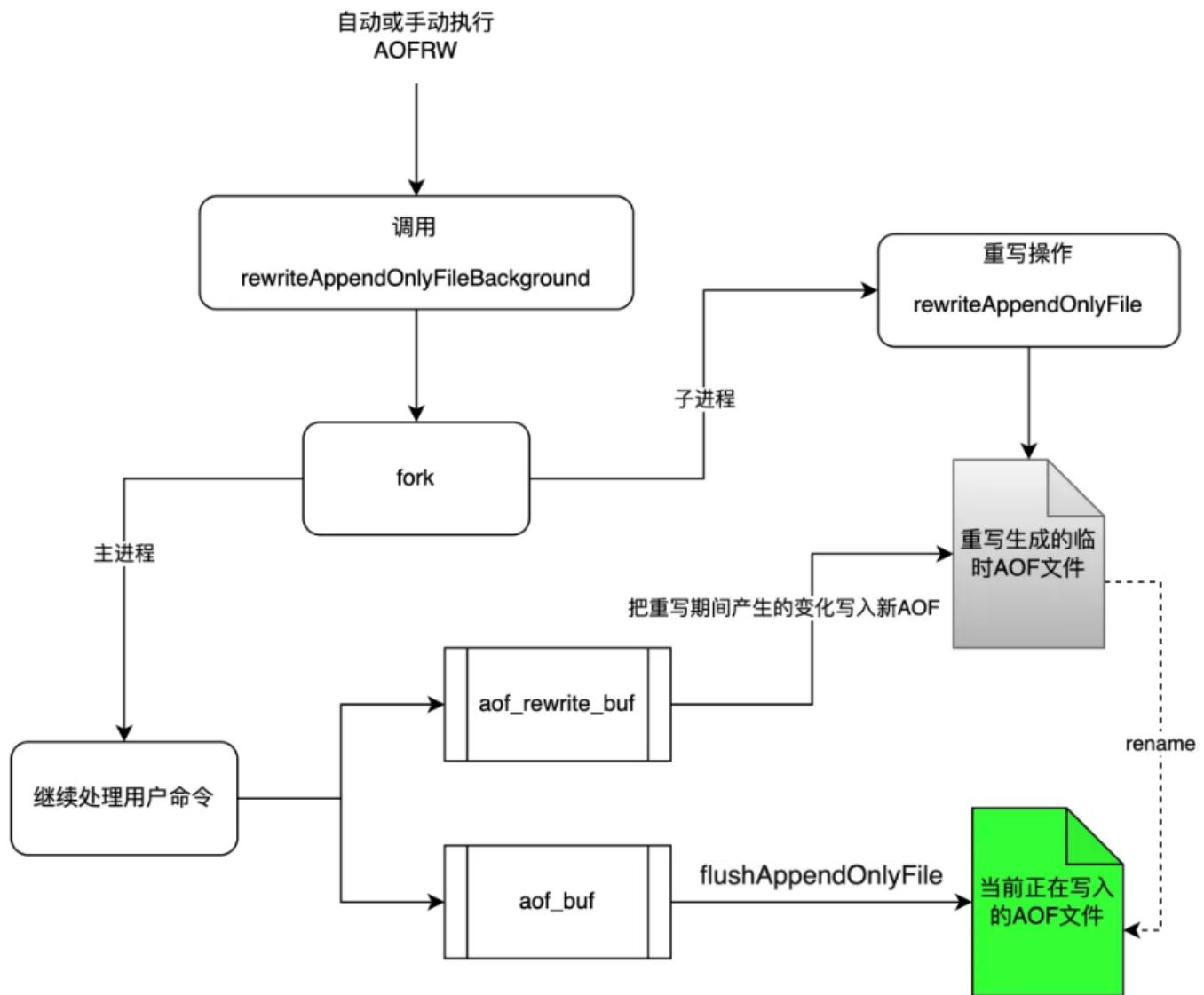
而在bgrewriteaof子进程完成会日志文件的重写操作后，会提示主线程已经完成重写操作，主线程会将AOF重写缓冲中的命令追加到新的日志文件后面。这时候在高并发的情况下，AOF重写缓冲区积累可能会很大，这样就会造成阻塞，Redis后来通过Linux管道技术让aof重写期间就能同时进行回放，这样aof重写结束后只需回放少量剩余的数据即可。

最后通过修改文件名的方式，保证文件切换的原子性。

在AOF重写日志期间发生宕机的话，因为日志文件还没切换，所以恢复数据时，用的还是旧的日志文件。

著作版权归pdai所有 原文链接：<https://pdai.tech/md/interview/x-interview.html>





18. AOF日志重写的时候，是由bgrewriteao子进程来完成的，不用主线程参与，我们今天说的非阻塞也是指子进程的执行不阻塞主线程。但是，你觉得，这个重写过程有没有其他潜在的阻塞风险呢？如果说有的话，会在哪里阻塞？

a. fork子进程，fork这个瞬间一定是会阻塞主线程的（注意，fork时并不会一次性拷贝所有内存数据给子进程，老师文章写的是拷贝所有内存数据给子进程，我个人认为是有歧义的），fork采用操作系统提供的写实复制(Copy On Write)机制，就是为了避免一次性拷贝大量内存数据给子进程造成的时间阻塞问题，但fork子进程需要拷贝进程必要的数据结构，其中有一项就是拷贝内存页表（虚拟内存和物理内存的映射索引表），这个拷贝过程会消耗大量CPU资源，拷贝完成之前整个进程是会阻塞的，阻塞时间取决于整个实例的内存大小，实例越大，内存页表越大，fork阻塞时间越久。拷贝内存页表完成后，子进程与父进程指向相同的内存地址空间，也就是说此时虽然产生了子进程，但是并没有申请与父进程相同的内存大小。那什么时候父子进程才会真正内存分离

呢？“写实复制”顾名思义，就是在写发生时，才真正拷贝内存真正的数据，这个过程中，父进程也可能会产生阻塞的风险，就是下面介绍的场景。

b. fork出的子进程指向与父进程相同的内存地址空间，此时子进程就可以执行AOF重写，把内存中的所有数据写入到AOF文件中。但是此时父进程依旧是会有流量写入的，如果父进程操作的是一个已经存在的key，那么这个时候父进程就会真正拷贝这个key对应的内存数据，申请新的内存空间，这样逐渐地，父子进程内存数据开始分离，父子进程逐渐拥有各自独立的内存空间。**因为内存分配是以页为单位进行分配的，默认4k**，如果父进程此时操作的是一个bigkey，重新申请大块内存耗时会变长，可能会产阻塞风险。另外，如果操作系统开启了内存大页机制(Huge Page，页面大小2M)，那么父进程申请内存时阻塞的概率将会大大提高，所以在Redis机器上需要关闭Huge Page机制。Redis每次fork生成RDB或AOF重写完成后，都可以在Redis log中看到父进程重新申请了多大的内存空间。

## 19. AOF重写也有一个重写日志，为什么它不共享使用AOF本身的日志呢

AOF重写不复用AOF本身的日志，一个原因是父子进程写同一个文件必然会产生竞争问题，控制竞争就意味着会影响父进程的性能。

二是如果AOF重写过程中失败了，那么原本的AOF文件相当于被污染了，无法做恢复使用。所以Redis AOF重写一个新文件，重写失败的话，直接删除这个文件就好了，不会对原先的AOF文件产生影响。等重写完成之后，直接替换旧文件即可

# 垃圾回收

## 1. Redis的过期键的删除策略

我们都知道，Redis是key-value数据库，我们可以设置Redis中缓存的key的过期时间。**Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。**

过期策略通常有以下三种：

### 定时过期：

每个设置过期时间的key都需要**创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据**，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。

## 惰性过期：

只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。

## 定期过期：

每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

( `expires` 字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)

Redis中同时使用了惰性过期和定期过期两种过期策略。

## 2. Redis key的过期时间和永久有效分别怎么设置？

EXPIRE 和 PERSIST 命令。

## 3. 我们知道通过 expire 来设置key 的过期时间，那么对过期的数据怎么处理呢？

定时过期、惰性过期、定期过期。

## 4. MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据

redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

## 5. Redis的内存淘汰策略有哪些

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

全局的键空间选择性移除

- 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。（这个是最常用的）
- 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。

### 设置过期时间的键空间选择性移除

- 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。

### 总结

Redis的内存淘汰策略的选取并不会影响过期的key的处理。内存淘汰策略用于处理内存不足时的需要申请额外空间的数据；过期策略用于处理过期的缓存数据。

## 6. Redis主要消耗什么物理资源？

内存。

## 7. Redis的内存用完了会发生什么？

如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以配置内存淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

## 8. Redis如何做内存优化？

可以好好利用 `Hash, list, sorted set, set` 等集合类型数据，因为通常情况下很多小的 Key-Value 可以用更紧凑的方式存放一起。

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key，而是应该把这个用户的所有信息存储到一张散列表里面。

## 9. Redis回收进程如何工作的？

1. 一个客户端运行了新的命令，添加了新的数据。
2. Redis检查内存使用情况，如果大于maxmemory的限制，则根据设定好的内存淘汰策略进行回收。
3. 一个新的命令被执行，等等。
4. 所以我们不断地穿越内存限制的边界，通过不断达到边界然后不断地收回回到边界以下。

如果一个命令的结果导致大量内存被使用（例如很大的集合的交集保存到一个新的键），不用多久内存限制就会被这个内存使用量超越。

## 10. Redis回收使用的是什么算法？

LRU算法。

LRU是Least Recently Used的缩写，即最近最少使用页面置换算法，是为虚拟页式存储管理服务的，是根据页面调入内存后的使用情况进行决策。由于无法预测各页面将来的使用情况，只能利用“最近的过去”作为“最近的将来”的近似，因此，LRU算法就是将最近最久未使用的页面予以淘汰。

# 线程模型

## 1. Redis线程模型

Redis基于Reactor模式开发了网络事件处理器，这个处理器被称为文件事件处理器（file event handler）。它的组成结构为4部分：多个套接字、IO多路复用程序、文件事件分派器、事件处理器。因为只有一个工作者线程，所以Redis才叫单线程模型。

- 文件事件处理器使用I/O多路复用（multiplexing）程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。
- 当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时，与操作相对应的文件事件就会产生，这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

虽然worker是单线程方式运行，但通过使用I/O多路复用程序来监听多个套接字，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与redis服务器中其他同样以单线程方式运行的模块进行对接，这保持了Redis内部单线程设计的简单性。

## 2. 单线程为什么快

redis 利用 epoll 进行多路复用管理客户端连接，通过监听 epoll 事件，对客户的请求操作放到工作者线程去处理。由于工作者线程是单线程，所以他并不存在数据冲突问题，也就是说它不需要开启事务或者说加锁来解决并发所带来的线程安全问题。

所以我认为 redis 单线程快，就是因为它的逻辑处理上避免了线程安全问题，减少了事务或者锁的开销，从而加快数据的计算速度。

# 事务

## 1. 什么是事务？

在 redis 中，事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

## 2. Redis事务的概念

Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。

总结说：redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。

## 3. Redis事务的三个阶段

1. 事务开始 MULTI
2. 命令入队
3. 事务执行 EXEC

事务执行过程中，如果服务端收到有EXEC、DISCARD、WATCH、MULTI之外的请求，将会把请求放入队列中排队。

## 4. Redis事务相关命令

Redis事务功能是通过 MULTI、EXEC、DISCARD 和 WATCH 四个原语实现的

Redis会将一个事务中的所有命令序列化，然后按顺序执行。

1. redis 不支持回滚，“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”，所以 Redis 的内部可以保持简单且快速。
2. 如果在一个事务中的命令出现错误，那么所有的命令都不会执行；
3. 如果在一个事务中出现运行错误，那么正确的命令会被执行。

- WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set (CAS) 行为。可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。
- MULTI 命令用于开启一个事务，它总是返回OK。MULTI 执行之后，客户端可以继续向服务器发送任意多条命令，**这些命令不会立即被执行，而是被放到一个队列中**，当 EXEC 命令被调用时，所有队列中的命令才会被执行。
- EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 nil。
- 通过调用 DISCARD，客户端可以清空事务队列，并放弃执行事务，并且客户端会从事务状态中退出。
- UNWATCH 命令可以取消watch对所有key的监控。

## 5. 事务管理 (ACID) 概述

### 原子性 (Atomicity)

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

### 一致性 (Consistency)

事务前后数据的完整性必须保持一致。

### 隔离性 (Isolation)

多个事务并发执行时，一个事务的执行不应影响其他事务的执行

### 持久性 (Durability)

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

Redis的事务总是具有ACID中的一致性和隔离性，其他特性是不支持的。当服务器运行在AOF持久化模式下，并且appendfsync选项的值为always时，事务也具有耐久性。

## 6. Redis事务支持隔离性吗

Redis只有一个工作者线程，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis的事务是总是带有隔离性的。

## 7. Redis事务保证原子性吗，支持回滚吗

Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

## 8. Redis事务其他实现

- 基于Lua脚本，Redis可以保证脚本内的命令一次性、按顺序地执行，其同时也不提供事务运行错误的回滚，执行过程中如果部分命令运行错误，剩下的命令还是会继续运行完。
- 基于中间标记变量，通过另外的标记变量来标识事务是否执行完成，读取数据时先读取该标记变量判断是否事务执行完成。但这样会需要额外写代码实现，比较繁琐。

## 9. redis 事务、流水线与 lua 脚本有什么区别

### Pipeline 流水线

将指令在客户端的缓冲区里打包，最后一次性发送，重点在于节省多次发送指令的网络开销。和事务&Lua有以下几个区别：

1. 这是纯客户端行为，服务端无感知，也没有进行对应的特殊处理。
2. 不阻塞服务端执行其他客户端的指令，即没有串行化
3. 无法在一个PPL内，一个指令读取上一个指令的结果

### 事务

将每条命令缓存在服务端的一个队列中，提交时再一次性执行。

和Pipeline的区别如下：

1. 服务端感知，每次客户端发送单指令都和服务端交互。
2. 严格串行化，会阻塞事务之外其他客户端命令的执行。
3. 如果遇到指令的语法错误，会在提交前即入队列时暴露出来。
4. 拥有 watch 指令功能，相当于一种乐观锁。

## Lua 脚本

Lua 相当于 Pipeline 和 事务优势的结合，并提供了更强大的能力。一般在开发中代替原生提供的事务。主要的区别在于：

1. 可自定义复杂逻辑，如if-else，高级运算符等。
2. 具有编译缓存能力，可以使用 sha 调用缓存在服务器的同一脚本，进一步提高性能。

## 10. 了解 WATCH 命令吗

Watch 命令是Exec命令的执行条件，如果Watch的Key没有被修改则Redis执行事务，否则事务不会被执行。

Watch 命令可以被调用多次，一个 Watch 命令可以监控多个 key 。Watch 命令调用即启动监控功能，从Watch 命令开始点到执行 EXEC 命令终止。一旦 EXEC 被调用，所有的键都将不被监视，无论所讨论的事务是否被中止。关闭客户端连接也会触发所有的键被取消监视。

Redis Watch 命令给事务提供check-and-set (CAS) 机制。被 Watch 的 Key 被持续监控，如果 key 在 Exec 命令执行前有改变，那么整个事务被取消，Exec返回null表示事务没有成功。

请看下面示例，我们需要给key的值加一，使用下面命令：



Shell

复制代码

```
1 num = GET sampleKey
2 num = num + 1
3 SET sampleKey $num
```

上面命令在单用户执行环境下没有任何问题。如果多个用户尝试同时正键的值会产生问题，假设 sampleKey 原来值为13，两个客户端尝试同时增加值。两者都将键设为14，最终结果为14而不是15。

使用watch命令可以解决该问题：

```
1 WATCH sampleKey
2 num = GET sampleKey
3 num = num + 1
4 MULTI
5 SET sampleKey $num
6 EXEC
```

使用上面代码，如果竞争条件发生，因为键被监控，Exec会执行失败。只有当没有竞争条件时才能正确执行事务。这种锁处理也称为“**乐观锁**”，它提供有效昂是保障竞争条件。大多数情况下不同客户端访问不同的键，因此冲突的概率相当小，事务不太可能需要重复执行。

## 11. redis 事务的安全性如何保证

redis 事务的安全性，可以利用 Watch 提供的监视机制，启用 cas 乐观锁标志，在多极坏境下，客户端 a 开启监视后启用事务过程中，如果客户端b修改了监视的键，那么 `REDIS_DIRTY_CAS` 会打开，当客户端 a 执行 `EXEC` 的时候，服务器会拒绝执行它提交的事务，以此来保证事务的数据安全。

## LUA 脚本

### 1. redis 为什么要引入 lua 脚本

因为 Redis 虽然提供了非常丰富的指令集，官网上提供了200多个命令。但是某些特定领域，需要扩充若干指令原子性执行时，仅使用原生命令便无法完成。

Redis 为这样的用户场景提供了 lua 脚本支持，用户可以向服务器发送 lua 脚本来执行自定义动作，获取脚本的响应数据。

### 2. redis 使用 lua 有什么特性

1. 减少网络开销。可以将多个请求通过脚本的形式一次发送，减少网络时延。
2. 原子操作。Redis会将整个脚本作为一个整体执行，中间不会被其他请求插入。因此在脚本运行过程中无需担心会出现竞态条件，无需使用事务。
3. 复用。客户端发送的脚本可以使用 load 加载然后缓存起来获得该脚本的 sha 值，这样其他客户端可以根据 sha 复用这一脚本，而不需要使用代码完成相同的逻辑。

### 3. lua 脚本执行一半，机器断电了会怎么样

Redis是基于内存的数据库，所以当发生宕机或者停止后重新启动时，Redis会使用磁盘上的持久化文件来恢复数据，**所以是否能恢复数据，能恢复多少数据，取决于使用哪一种持久化策略。**

- 如果是使用 rdb，很显然恢复开机之后，只会恢复我们备份时的数据。
- 如果是使用AOF的话，如果在Redis事务执行期间宕机，那么这次事务还是相当于"没有执行"，由于命令还没来及写入AOF，在服务恢复后更不可能恢复数据。**对于Redis客户端而言，会收到服务端的异常响应。**写入AOF的过程也是会被打断的，Redis 文档中提到，如果Redis服务器突然崩溃，导致出现了"半写状态"的AOF文件时，服务器重新启动时，会检测到这种情况，并且退出提示用户使用 redis-check-aof 修复 AOF 文件。"半写状态"的事务或者命令会被删除，服务器可以重新启动。

### 4. lua 脚本跟事务有什么差别

redis 事务执行中，每一条指令之间是相互独立的，我们无法让后面的操作依赖前面命名的结果，这就让整个**事务仅仅成为了一个命令集合**，在命令之间我们完全无法做任何事。

但是，Lua 作为一个脚本语言，可以拥有分支、循环等语法结构，可以进行业务逻辑的编写

## 分布式问题

### 1. Redis实现分布式锁

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系 Redis 中可以使用 SETNX 命令实现分布式锁。

当且仅当 key 不存在，将 key 的值设为 value。若给定的 key 已经存在，则 SETNX 不做任何动作

```
127.0.0.1:6379> setnx lock-key value1
<integer> 1
127.0.0.1:6379> setnx lock-key value2
<integer> 0
127.0.0.1:6379> get lock-key
"value1"
```

使用 SETNX 完成同步锁的流程及事项如下：

使用 SETNX 命令获取锁，若返回0（key已存在，锁已存在）则获取失败，反之获取成功

为了防止获取锁后程序出现异常，导致其他线程/进程调用SETNX命令总是返回0而进入死锁状态，需要为该key设置一个“合理”的过期时间

释放锁，使用DEL命令将锁数据删除

## 2. 如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）

基于zookeeper临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推Zookeeper。

参考：<https://www.jianshu.com/p/8bdd381de06>

## 3. 分布式Redis是前期做还是后期规模上来了再做好？为什么？

既然Redis是如此的轻量（单实例只使用1M内存），为防止以后的扩容，最好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让Redis以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。

一开始就多设置几个Redis实例，例如32或者64个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。

这样的话，当你的数据不断增长，需要更多的Redis服务器时，你需要做的就是仅仅将Redis实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的Redis实例从第一台机器迁移到第二台机器。

## 4. 什么是 RedLock

Redis 官方站提出了一种权威的基于 Redis 实现分布式锁的方式名叫 *Redlock*，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

## 缓存异常

### 1. 缓存雪崩

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

#### 解决方案

1. 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
2. 一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。
3. 给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。

### 2. 缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

#### 解决方案

1. 接口层增加校验，如用户鉴权校验，id 做基础校验， $id \leq 0$  的直接拦截；
2. 从缓存取不到的数据，在数据库中也没有取到，这时也可以将 key-value 对写为 key-null，缓存有效时间可以设置短点，如 30 秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个 id 暴力攻击
3. 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力

#### 附加

对于空间的利用到达了一种极致，那就是 Bitmap 和布隆过滤器(Bloom Filter)。

Bitmap：典型的就是哈希表

缺点是，Bitmap 对于每个元素只能记录 1bit 信息，如果还想完成额外的功能，恐怕只能靠牺牲更多的空间、时间来完成了。

## 布隆过滤器（推荐）

就是引入了  $k(k>1)$  个相互独立的哈希函数，保证在给定的空间、误判率下，完成元素判断的过程。

它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

Bloom–Filter算法的核心思想就是利用多个不同的Hash函数来解决“冲突”。

Hash存在一个冲突（碰撞）的问题，用同一个Hash得到的两个URL的值有可能相同。为了减少冲突，我们可以多引入几个Hash，如果通过其中的一个Hash值我们得出某元素不在集合中，那么该元素肯定不在集合中。只有在所有的Hash函数告诉我们该元素在集合中时，才能确定该元素存在于集合中。这便是Bloom–Filter的基本思想。

Bloom–Filter一般用于在大数据量的集合中判定某元素是否存在。

## 3. 缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

### 解决方案

1. 设置热点数据永远不过期。
2. 加互斥锁，互斥锁

## 4. 缓存预热

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

### 解决方案

1. 直接写个缓存刷新页面，上线时手工操作一下；
2. 数据量不大，可以在项目启动的时候自动进行加载；
3. 定时刷新缓存；

## 5. 缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

**缓存降级**的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

1. 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
2. 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；
3. 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；
4. 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

## 6. 热点数据和冷数据

热点数据，缓存才有价值

对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。频繁修改的数据，看情况考虑使用缓存

对于热点数据，比如我们的某IM产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次。再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次。

数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。

那存不存在，修改频率很高，但是又不得不考虑缓存的场景呢？有！比如，这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到Redis缓存，减少数据库压力。

## 7. 缓存热点key

缓存中的一个Key(比如一个促销商品)，在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回写到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

### 解决方案

对缓存查询加锁，如果KEY不存在，就加锁，然后查DB入缓存，然后解锁；其他进程如果发现有锁就等待，然后等解锁后返回数据或者进入DB查询

## 其他问题

### 1. 如何保证缓存与数据库双写时的数据一致性？

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的  
问题，那么你如何解决一致性问题？

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶尔  
有不一致的情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可以  
保证一定不会出现不一致的情况

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个  
请求。

还有一种方式就是可能会暂时产生不一致的情况，但是发生的几率特别小，就是先更新数据库，然后再  
删除缓存。

问题场景	描述	解决
先写缓存，再写数据 库，缓存写成功，数据 库写失败	缓存写成功，但写数据库失败或者响应延迟，则下次读取 (并发读) 缓存时，就出现脏读	这个写缓存的方式，本身就是错误的，需要改为先写数据库，把旧 缓存置为失效；读取数据的时候，如果缓存不存在，则读取数据库 再写缓存
先写数据库，再写缓 存，数据库写成功，缓 存写失败	写数据库成功，但写缓存失败，则下次读取 (并发读) 缓 存时，则读不到数据	缓存使用时，假如读缓存失败，先读数据库，再回写缓存的方式实 现
需要缓存异步刷新	指数据库操作和写缓存不在一个操作步骤中，比如在分布 式场景下，无法做到同时写缓存或需要异步刷新 (补救措 施) 时候	确定哪些数据适合此类场景，根据经验值确定合理的数据不一致时 间，用户数据刷新的时间间隔

### 2. Redis常见性能问题和解决方案？

1. Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化。
2. 如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。
3. 为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内。
4. 尽量避免在压力较大的主库上增加从库
5. Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。

6. 为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：  
Master<—Slave1<—Slave2<—Slave3...，这样的结构也方便解决单点故障问题，实现Slave对  
Master的替换，也即，如果Master挂了，可以立马启用Slave1做Master，其他不变。

### 3. 一个字符串类型的值能存储最大容量是多少？

512M

### 4. Redis如何做大量数据插入？

Redis2.6开始redis-cli支持一种新的被称之为pipe mode的新模式用于执行大量数据插入工作。

### 5. 假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？

使用keys指令可以扫出指定模式的key列表。

对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？

这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

### 6. 使用Redis做过异步队列吗，是如何实现的

使用list类型保存数据信息，**rpush**生产消息，**lpop**消费消息，当**lpop**没有消息时，可以**sleep**一段时间，  
然后再检查有没有信息，如果不**sleep**的话，可以使用**blpop**，在没有信息的时候，会一直阻塞，直到信息的到来。  
redis可以通过pub/sub主题订阅模式实现一个生产者，多个消费者，当然也存在一定的缺点，当消费者下线时，生产的消息会丢失。

### 7. Redis如何实现延时队列

使用sortedset，使用时间戳做score，消息内容作为key，调用zadd来生产消息，消费者使用  
**zrangebyscore** 获取n秒之前的数据做轮询处理。

# 大Key

## 1. 什么是 Redis 大 key 问题

根据数据类型来分，主要分两大类 字符型和其他。

- 单个简单的key存储的value很大，size超过10KB
- hash, set, zset, list 中存储过多的元素（以万为单位）

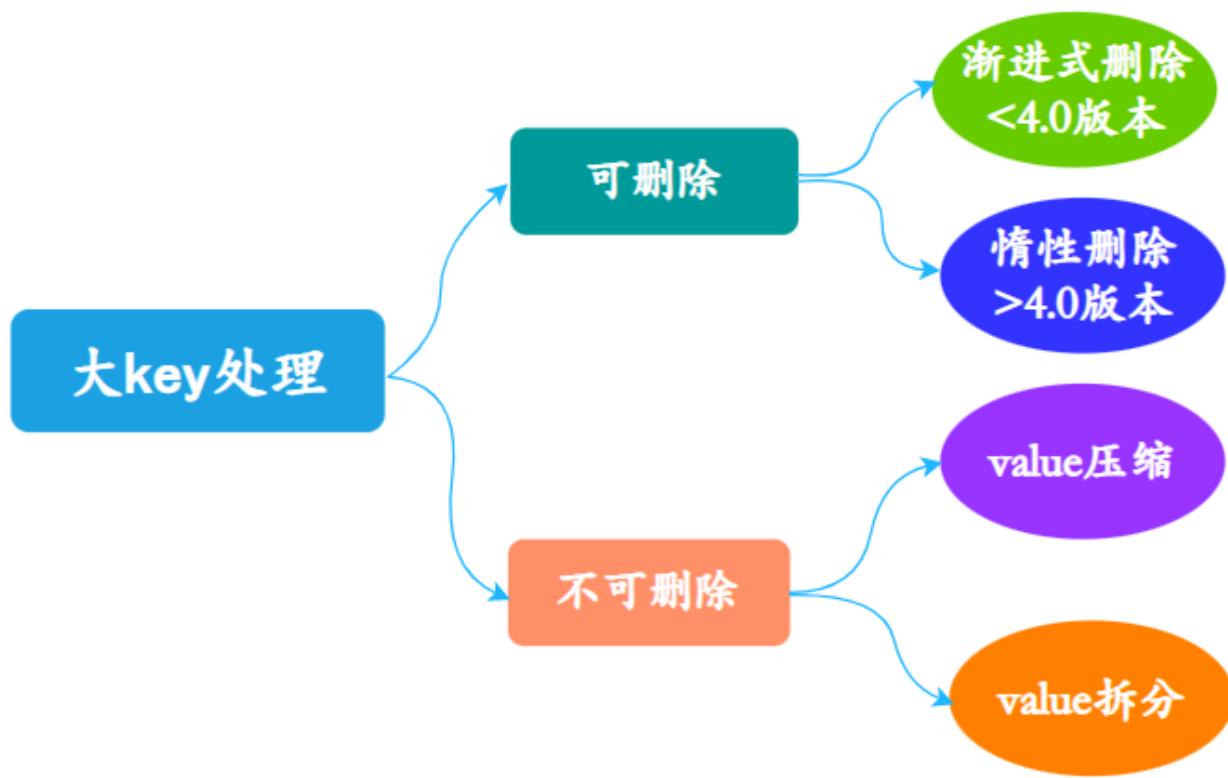
## 2. 大key会造成什么问题呢？

- 客户端耗时增加，甚至超时
- 对大key进行IO操作时，会严重占用带宽和CPU
- 造成Redis集群中数据倾斜
- 主动删除、被动删等，可能会导致阻塞

## 3. 如何找到大key？

- bigkeys命令：使用 `bigkeys` 命令以遍历的方式分析Redis实例中的所有Key，并返回整体统计信息与每个数据类型中Top1的大Key
- redis-rdb-tools：redis-rdb-tools是由Python写的用来分析Redis的rdb快照文件用的工具，它可以把rdb快照文件生成json文件或者生成报表用来分析Redis的使用详情。

## 4. 如何处理大key？



## 4.1 删大key

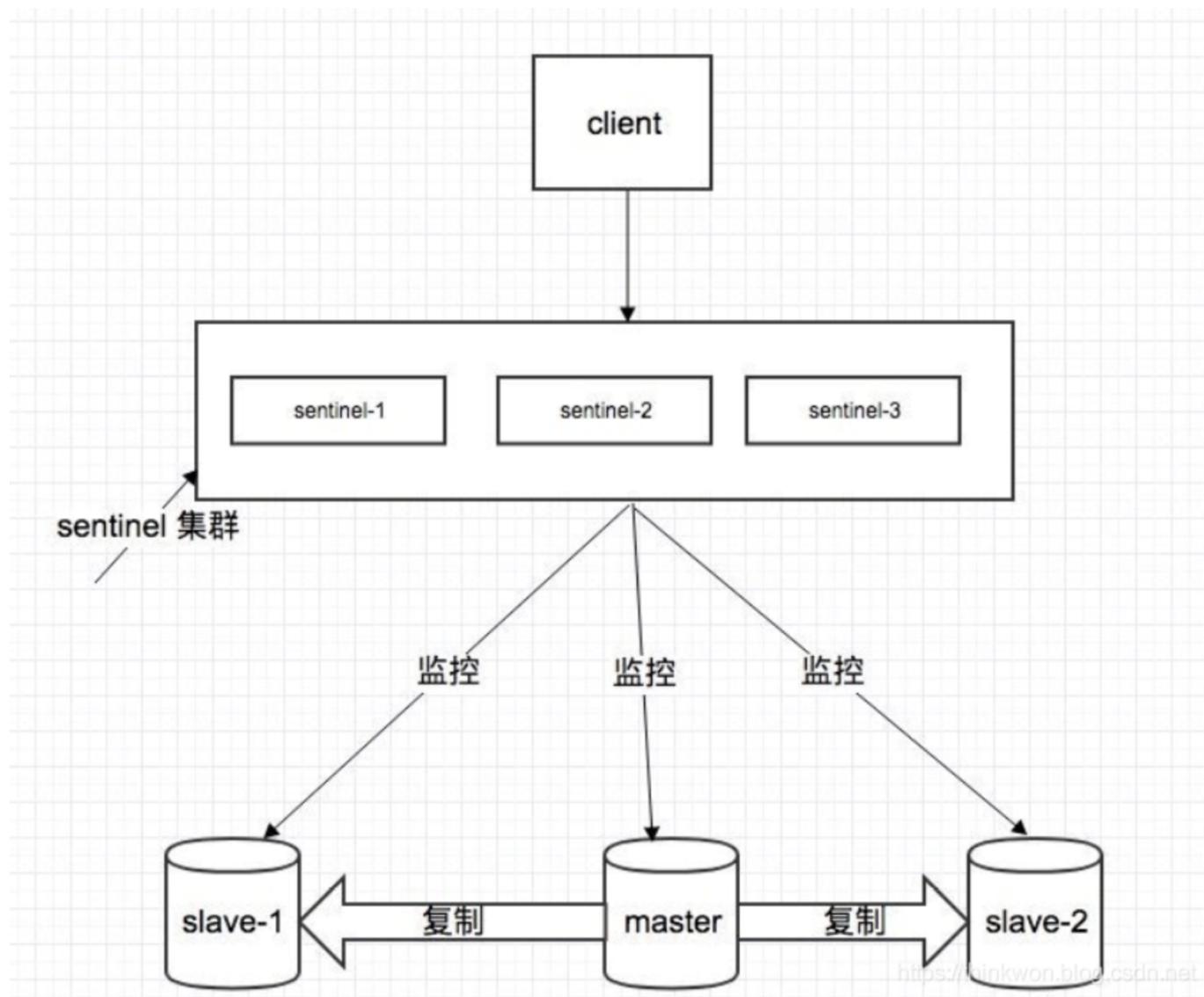
- 当Redis版本大于4.0时，可使用UNLINK命令安全地删除大Key，该命令能够以非阻塞的方式，逐步地清理传入的Key。
- 当Redis版本小于4.0时，避免使用阻塞式命令KEYS，而是建议通过SCAN命令执行增量迭代扫描key，然后判断进行删除。

## 4.2 压缩和拆分key

- 当value是string时，比较难拆分，则使用序列化、压缩算法将key的大小控制在合理范围内，但是序列化和反序列化都会带来更多时间上的消耗。
- 当value是string，压缩之后仍然是大key，则需要进行拆分，一个大key分为不同的部分，记录每个部分的key，使用multiget等操作实现事务读取。
- 当value是list/set等集合类型时，根据预估的数据规模来进行分片，不同的元素计算后分到不同的片。

# 集群方案 (校招基本不问)

## 哨兵模式



## 哨兵的介绍

sentinel，中文名是哨兵。哨兵是 redis 集群机构中非常重要的一个组件，主要有以下功能：

- 集群监控：负责监控 redis master 和 slave 进程是否正常工作。
- 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
- 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

- 故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式

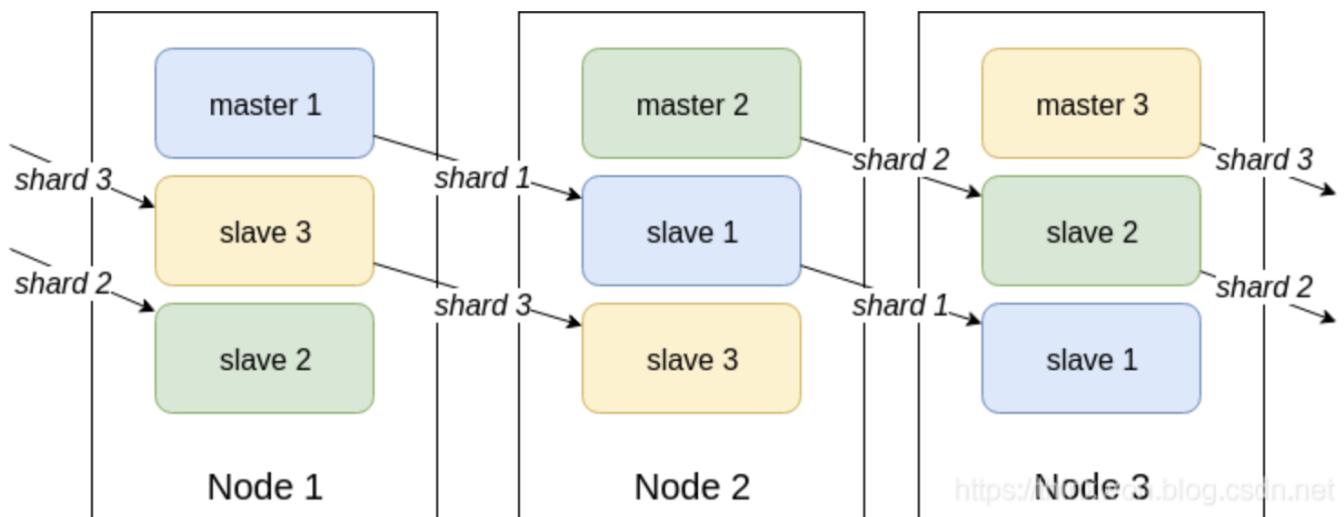
选举的问题。

- 即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了。

## 哨兵的核心知识

- 哨兵至少需要 3 个实例，来保证自己的健壮性。
- 哨兵 + redis 主从的部署架构，是不保证数据零丢失的，只能保证 redis 集群的高可用性。
- 对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

## 官方Redis Cluster 方案(服务端路由查询)



redis 集群模式的工作原理能说一下么？在集群模式下，redis 的 key 是如何寻址的？分布式寻址都有哪些算法？了解一致性 hash 算法吗？

## 简介

Redis Cluster是一种服务端Sharding技术，3.0版本开始正式提供。Redis Cluster并没有使用一致性hash，而是采用slot(槽)的概念，一共分成16384个槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行

## 方案说明

- 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了 16384 个槽位
- 每份数据分片会存储在多个互为主从的多节点上
- 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)

4. 同一分片多个节点间的数据不保持一致性
5. 读取数据时，当客户端操作的key没有分配在该节点上时，redis会返回转向指令，指向正确的节点
6. 扩容时时需要需要把旧节点的数据迁移一部分到新节点

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是 加1w 的端口号，比如 16379。

16379 端口号是用来进行节点间通信的，也就是 cluster bus 的东西，cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。cluster bus 用了另外一种二进制的协议， gossip 协议，用于节点间进行高效的数据交换，占用更少的网络带宽和处理时间。

## 节点间的内部通信机制

基本通信原理

集群元数据的维护有两种方式：集中式、Gossip 协议。redis cluster 节点间采用 gossip 协议进行通信。

## 分布式寻址算法

- hash 算法（大量缓存重建）
- 一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡）
- redis cluster 的 hash slot 算法

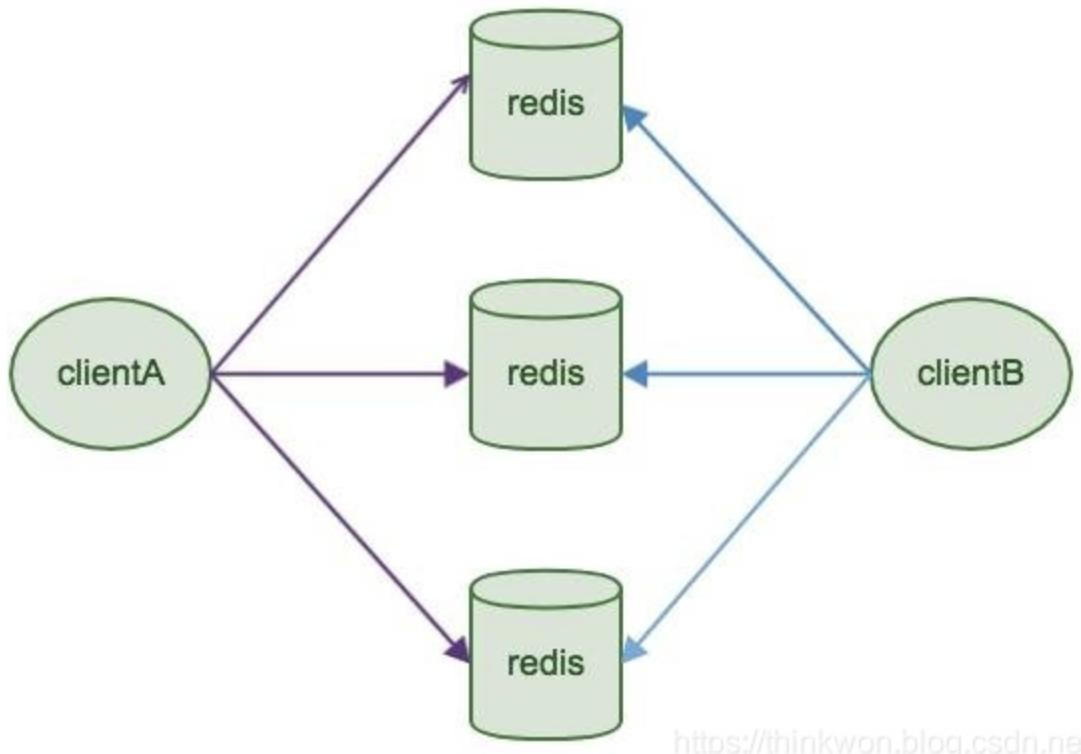
## 优点

- 无中心架构，支持动态扩容，对业务透明
- 具备Sentinel的监控和自动Failover(故障转移)能力
- 客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可
- 高性能，客户端直连redis服务，免去了proxy代理的损耗

## 缺点

- 运维也很复杂，数据迁移需要人工干预
- 只能使用0号数据库
- 不支持批量操作(pipeline管道操作)
- 分布式逻辑和存储模块耦合等

## 基于客户端分配



<https://thinkwon.blog.csdn.net>

## 简介

Redis Sharding是Redis Cluster出来之前，业界普遍使用的多Redis实例集群方法。其主要思想是采用哈希算法将Redis数据的key进行散列，通过hash函数，特定的key会映射到特定的Redis节点上。Java redis客户端驱动jedis，支持Redis Sharding功能，即ShardedJedis以及结合缓存池的ShardedJedisPool

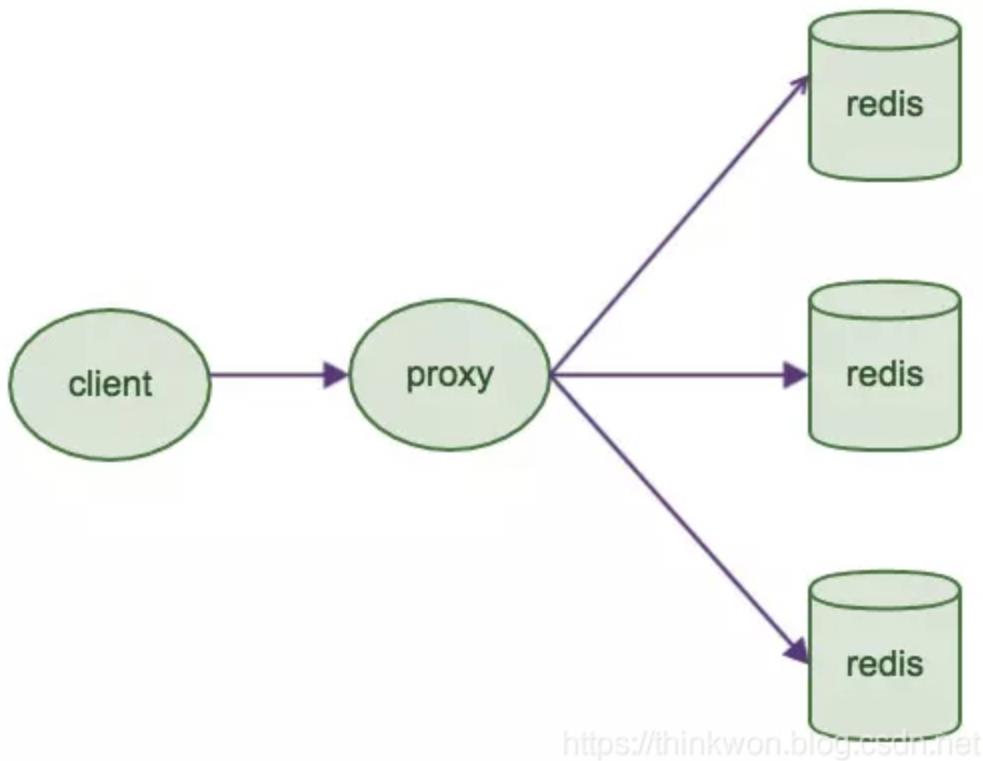
## 优点

优势在于非常简单，服务端的Redis实例彼此独立，相互无关联，每个Redis实例像单服务器一样运行，非常容易线性扩展，系统的灵活性很强

## 缺点

- 由于sharding处理放到客户端，规模进一步扩大时给运维带来挑战。
- 客户端sharding不支持动态增删节点。服务端Redis实例群拓扑结构有变化时，每个客户端都需要更新调整。连接不能共享，当应用规模增大时，资源浪费制约优化

## 基于代理服务器分片



## 简介

客户端发送请求到一个代理组件，代理解析客户端的数据，并将请求转发至正确的节点，最后将结果回复给客户端

## 特征

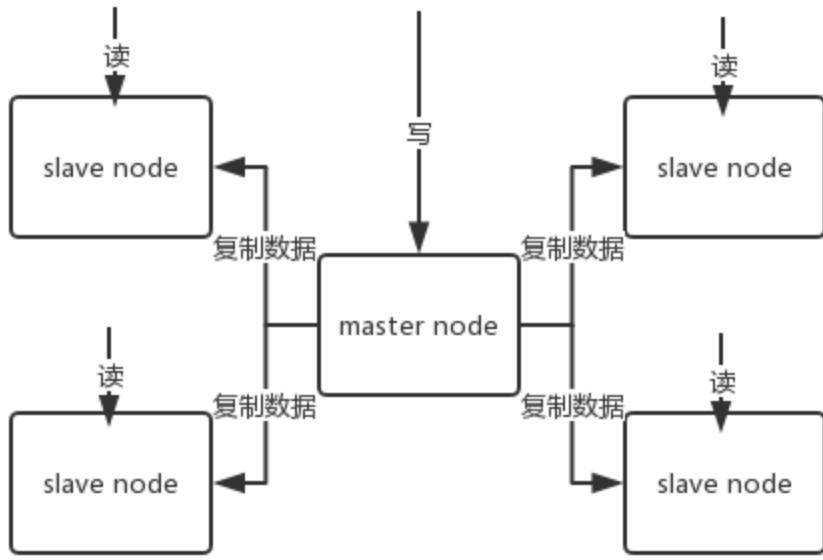
- 透明接入，业务程序不用关心后端Redis实例，切换成本低
- Proxy 的逻辑和存储的逻辑是隔离的
- 代理层多了一次转发，性能有所损耗

## 业界开源方案

- Twitter开源的Twemproxy
- 豌豆荚开源的Codis

## Redis 主从架构

单机的 redis，能够承载的 QPS 大概就在上万到几万不等。对于缓存来说，一般都是用来支撑读高并发的。因此架构做成主从(master-slave)架构，一主多从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的读请求全部走从节点。这样也可以很轻松实现水平扩容，支撑读高并发。



redis replication -> 主从架构 -> 读写分离 -> 水平扩容支撑读高并发

### redis replication 的核心机制

- redis 采用**异步方式**复制数据到 slave 节点，不过 redis2.8 开始，slave node 会周期性地确认自己每次复制的数据量；
- 一个 master node 是可以配置多个 slave node 的；
- slave node 也可以连接其他的 slave node；
- slave node 做复制的时候，不会 block master node 的正常工作；
- slave node 在做复制的时候，也不会 block 对自己的查询操作，它会用旧的数据集来提供服务；但是复制完成的时候，需要删除旧数据集，加载新数据集，这个时候就会暂停对外服务了；
- slave node 主要用来进行横向扩容，做读写分离，扩容的 slave node 可以提高读的吞吐量。

注意，如果采用了主从架构，那么建议必须**开启** master node 的持久化，不建议用 slave node 作为 master node 的数据热备，因为那样的话，如果你关掉 master 的持久化，可能在 master 宕机重启的时候数据是空的，然后可能一经过复制， slave node 的数据也丢了。

另外，master 的各种备份方案，也需要做。万一本地的所有文件丢失了，从备份中挑选一份 rdb 去恢复 master，这样才能**确保启动的时候，是有数据的**，即使采用了后续讲解的高可用机制，slave node 可以自动接管 master node，但也可能 sentinel 还没检测到 master failure，master node 就自动重启了，还是可能导致上面所有的 slave node 数据被清空。

### redis 主从复制的核心原理

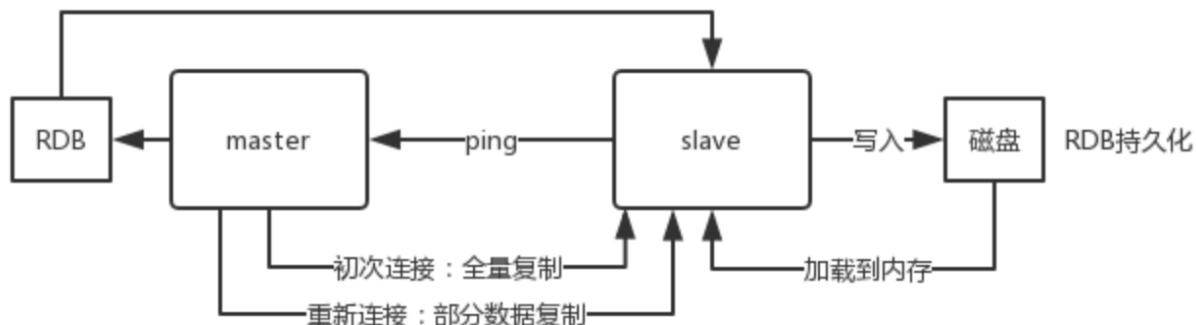
当启动一个 slave node 的时候，它会发送一个 `PSYNC` 命令给 master node。

如果这是 slave node 初次连接到 master node，那么会触发一次 `full resynchronization` 全量复制。此时 master 会启动一个后台线程，开始生成一份 `RDB` 快照文件，

同时还会将从客户端 client 新收到的所有写命令缓存在内存中。`RDB` 文件生成完毕后，master 会将这个 `RDB` 发送给 slave，slave 会先写入本地磁盘，然后再从本地磁盘加载到内存中，

接着 master 会将内存中缓存的写命令发送到 slave，slave 也会同步这些数据。

slave node 如果跟 master node 有网络故障，断开了连接，会自动重连，连接之后 master node 仅会复制给 slave 部分缺少的数据。



## 过程原理

1. 当从库和主库建立MS关系后，会向主数据库发送SYNC命令
2. 主库接收到SYNC命令后会开始在后台保存快照(RDB持久化过程)，并将期间接收到的写命令缓存起来
3. 当快照完成后，主Redis会将快照文件和所有缓存的写命令发送给从Redis
4. 从Redis接收到后，会载入快照文件并且执行收到的缓存的命令
5. 之后，主Redis每当接收到写命令时就会将命令发送从Redis，从而保证数据的一致

## 缺点

所有的slave节点数据的复制和同步都由master节点来处理，会照成master节点压力太大，使用主从从结构来解决

## Redis集群的主从复制模型是怎样的？

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型，每个节点都会有N-1个复制品

## 生产环境中的 redis 是怎么部署的？

redis cluster, 10 台机器，5 台机器部署了 redis 主实例，另外 5 台机器部署了 redis 的从实例，每个主实例挂了一个从实例，5 个节点对外提供读写服务，每个节点的读写高峰qps可能可以达到每秒 5 万，5 台机器最多是 25 万读写请求/s。

机器是什么配置？32G 内存+ 8 核 CPU + 1T 磁盘，但是分配给 redis 进程的是10g内存，一般线上生产环境，redis 的内存尽量不要超过 10g，超过 10g 可能会有问题。

5 台机器对外提供读写，一共有 50g 内存。

因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis 从实例会自动变成主实例继续提供读写服务。

你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是 10kb。100 条数据是 1mb，10 万条数据是 1g。常驻内存的是 200 万条商品数据，占用内存是 20g，仅仅不到总内存的 50%。目前高峰期每秒就是 3500 左右的请求量。

其实大型的公司，会有基础架构的 team 负责缓存集群的运维。

## 说说Redis哈希槽的概念？

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

## Redis集群会有写操作丢失吗？为什么？

Redis并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

## Redis集群之间是如何复制的？

异步复制

## Redis集群最大节点个数是多少？

16384个

## Redis集群如何选择数据库？

Redis集群目前无法做数据库选择， 默认在0数据库。

## 分区（校招基本不会问）

### Redis是单线程的，如何提高多核CPU的利用率？

可以在同一个服务器部署多个Redis的实例，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的，所以，如果你想使用多个CPU，你可以考虑一下分片（shard）。

### 为什么要做Redis分区？

分区可以让Redis管理更大的内存，Redis将可以使用所有机器的内存。如果没有分区，你最多只能使用一台机器的内存。分区使Redis的计算能力通过简单地增加计算机得到成倍提升，Redis的网络带宽也会随着计算机和网卡的增加而成倍增长。

### 你知道有哪些Redis分区实现方案？

- 客户端分区就是在客户端就已经决定数据会被存储到哪个redis节点或者从哪个redis节点读取。大多数客户端已经实现了客户端分区。
- 代理分区意味着客户端将请求发送给代理，然后代理决定去哪个节点写数据或者读数据。代理根据分区规则决定请求哪些Redis实例，然后根据Redis的响应结果返回给客户端。redis和memcached的一种代理实现就是Twemproxy
- 查询路由(Query routing)的意思是客户端随机地请求任意一个redis实例，然后由Redis将请求转发给正确的Redis节点。Redis Cluster实现了一种混合形式的查询路由，但并不是直接将请求从一个redis节点转发到另一个redis节点，而是在客户端的帮助下直接redirected到正确的redis节点。

### Redis分区有什么缺点？

- 涉及多个key的操作通常不会被支持。例如你不能对两个集合求交集，因为他们可能被存储到不同的Redis实例（实际上这种情况也有办法，但是不能直接使用交集指令）。
- 同时操作多个key，则不能使用Redis事务。
- 分区使用的粒度是key，不能使用一个非常长的排序key存储一个数据集（The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set）

- 当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的Redis实例和主机同时收集RDB / AOF文件。
- 分区时动态扩容或缩容可能非常复杂。Redis集群在运行时增加或者删除Redis节点，能做到最大程度对用户透明地数据再平衡，但其他一些客户端分区或者代理分区方法则不支持这种特性。然而，有一种预分片的技术也可以较好的解决这个问题。