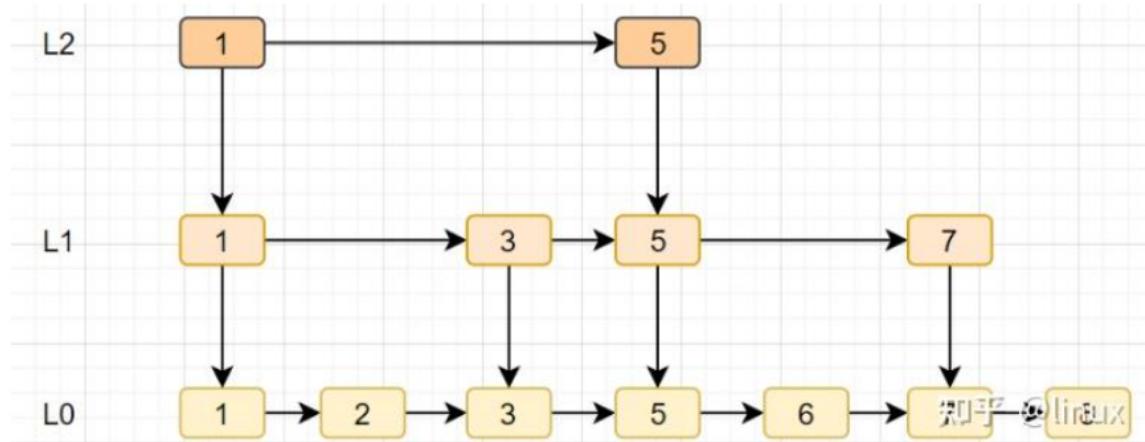


### 1. Redis特点:

- 键值型
- 单线程,每个命令具备原子性
- 低延迟,速度快(基于内存、IO多路复用、良好的编码)
- 支持数据持久化
- 支持主从集群、分片集群
- 支持多语言客户端(java,C++等)

### 2. QuickList的每一个节点都指向一个ZipList,QuickList=LinkedList+ZipList

### 3. SkipList和LevelDB的跳表一样



跳跃表

- 跳跃表是有序集合zset的底层实现之一
- 跳跃表支持平均 $O(\log N)$ ,最坏 $O(N)$ 复杂度的节点查找,还可以通过顺序性操作批量处理节点。
- 跳跃表实现由zskiplist和zskiplistNode两个结构组成,其中zskiplist用于保存跳跃表信息(如表头节点、表尾节点、长度),而zskiplistNode则用于表示跳跃表节点。
- 跳跃表就是在链表的基础上,增加多级索引提升查找效率。

### 4. Redis的五种基本数据结构:

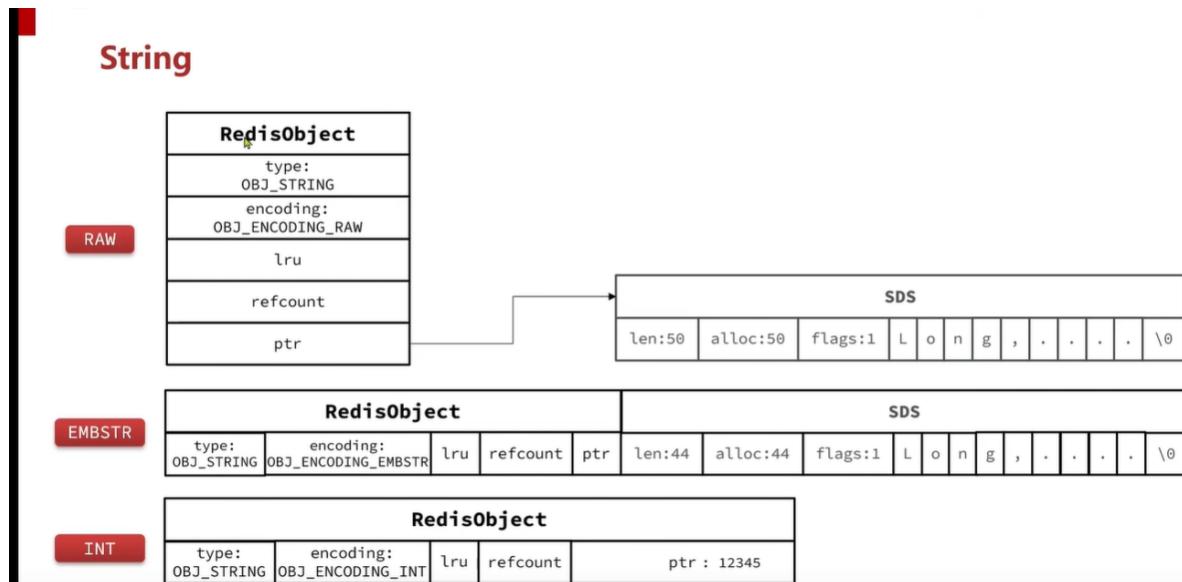
#### ◦ 字符串

String的常见命令有:

- SET: 添加或者修改已经存在的一个String类型的键值对
- GET: 根据key获取String类型的value
- MSET: 批量添加多个String类型的键值对
- MGET: 根据多个key获取多个String类型的value
- INCR: 让一个整型的key自增1
- INCRBY: 让一个整型的key自增并指定步长,例如: incrby num 2 让num值自增2
- INCRBYFLOAT: 让一个浮点类型的数字自增并指定步长
- SETNX: 添加一个String类型的键值对,前提是这个key不存在,否则不执行
- SETEX: 添加一个String类型的键值对,并且指定有效期

String是Redis中最常见的数据存储类型：

- ◆ 其基本编码方式是**RAW**，基于简单动态字符串（SDS）实现，存储上限为512mb。
- ◆ 如果存储的SDS长度小于44字节，则会采用**EMBSTR**编码，此时object head与SDS是一段连续空间。申请内存时只需要调用一次内存分配函数，效率更高。
- ◆ 如果存储的字符串是整数值，并且大小在LONG\_MAX范围内，则会采用**INT**编码：直接将数据保存在RedisObject的ptr指针位置（刚好8字节），不再需要SDS了。



## ○ 哈希表

- HSET key field value: 添加或者修改hash类型key的field的值
- HGET key field: 获取一个hash类型key的field的值
- HMSET: 批量添加多个hash类型key的field的值
- HMGET: 批量获取多个hash类型key的field的值
- HGETALL: 获取一个hash类型的key中的所有的field和value
- HKEYS: 获取一个hash类型的key中的所有的field
- HVALS: 获取一个hash类型的key中的所有的value
- HINCRBY: 让一个hash类型key的字段值自增并指定步长
- HSETNX: 添加一个hash类型的key的field值，前提是这个field不存在，否则不执行

因此，Hash底层采用的编码与Zset也基本一致，只需要把排序有关的SkipList去掉即可：

- ◆ Hash结构默认采用ZipList编码，用以节省内存。ZipList中相邻的两个entry分别保存field和value
- ◆ 当数据量较大时，Hash结构会转为HT编码，也就是Dict，触发条件有两个：
  - ① ZipList中的元素数量超过了hash-max-ziplist-entries（默认512）
  - ② ZipList中的任意entry大小超过了hash-max-ziplist-value（默认64字节）

- 列表:可以看作是一个双向链表结构,既可以支持正向检索也可以支持反向检索

- LPUSH key element ... : 向列表左侧插入一个或多个元素
- LPOP key: 移除并返回列表左侧的第一个元素, 没有则返回nil
- RPUSH key element ... : 向列表右侧插入一个或多个元素
- RPOP key: 移除并返回列表右侧的第一个元素
- LRANGE key star end: 返回一段角标范围内的所有元素
- BLPOP和BRPOP: 与LPOP和RPOP类似, 只不过在没有元素时等待指定时间, 而不是直接返回nil

Redis的List结构类似一个双端链表, 可以从首、尾操作列表中的元素:

- ◆ 在3.2版本之前, Redis采用ZipList和LinkedList来实现List, 当元素数量小于512并且元素大小小于64字节时采用ZipList编码, 超过则采用LinkedList编码。
- ◆ 在3.2版本之后, Redis统一采用QuickList来实现List:

- 集合

String的常见命令有:

- SADD key member ... : 向set中添加一个或多个元素
- SREM key member ... : 移除set中的指定元素
- SCARD key: 返回set中元素的个数
- SISMEMBER key member: 判断一个元素是否存在于set中
- SMEMBERS: 获取set中的所有元素
- SINTER key1 key2 ... : 求key1与key2的交集
- SDIFF key1 key2 ... : 求key1与key2的差集
- SUNION key1 key2 ...: 求key1和key2的并集

s

Set是Redis中的集合, 不一定确保元素有序, 可以满足元素唯一、查询效率要求极高。

- ◆ 为了查询效率和唯一性, set采用HT编码(Dict)。Dict中的key用来存储元素, value统一为null。
- ◆ 当存储的所有数据都是整数, 并且元素数量不超过set-max-intset-entries时, Set会采用IntSet编码, 以节省内存。

- 有序集合(SortedSet=ZSet):每个元素都带有一个score属性,可以基于score属性对元素排序,底层的实现是一个跳表+哈希表

SortedSet的常见命令有：

- ZADD key score member: 添加一个或多个元素到sorted set，如果已经存在则更新其score值
- ZREM key member: 删除sorted set中的一个指定元素
- ZSCORE key member : 获取sorted set中的指定元素的score值
- ZRANK key member: 获取sorted set 中的指定元素的排名
- ZCARD key: 获取sorted set中的元素个数
- ZCOUNT key min max: 统计score值在给定范围内的所有元素的个数
- ZINCRBY key increment member: 让sorted set中的指定元素自增，步长为指定的increment值
- ZRANGE key min max: 按照score排序后，获取指定排名范围内的元素
- ZRANGEBYSCORE key min max: 按照score排序后，获取指定score范围内的元素
- ZDIFF、ZINTER、ZUNION: 求差集、交集、并集

因此，zset底层数据结构必须满足键值存储、键必须唯一、可排序这几个需求。之前学习的哪种编码结构可以满足？

- ◆ SkipList: 可以排序，并且可以同时存储score和ele值 (member)
- ◆ HT (Dict) : 可以键值存储，并且可以根据key找value

当元素数量不多时，HT和SkipList的优势不明显，而且更耗内存。因此zset还会采用ZipList结构来节省内存，不过需要同时满足两个条件：

- ① 元素数量小于zset\_max\_ziplist\_entries，默认值128
- ② 每个元素都小于zset\_max\_ziplist\_value字节，默认值64

ziplist本身没有排序功能，而且没有键值对的概念，因此需要有zset通过编码实现：

- ZipList是连续内存，因此score和element是紧挨在一起的两个entry， element在前， score在后
- score越小越接近队首， score越大越接近队尾，按照score值升序排列

5. SDS、InSet、Dict、ZipList、QuickList、SkipList是底层的数据结构,它们会封装成RedisObject对象,也就是Redis对象,其实就是String、Hash、List、Set、ZSet类型对象

6. Redis的命令行使用

7. Redis的key允许有多个单词形成层级结构,多个单词之间用'：'隔开

8. HSET命令返回integer 0是因为name这个字段在heima:user:1这个哈希表中已经存在了(SET等命令返回的情况和这个类似)

9. Redis没有直接使用C语言中的字符串

10. 动态字符串SDS

```
struct __attribute__((__packed__)) sdshdr8 {
    uint8_t len;// 已保存的字符串字节数,不包含结束标识
    uint8_t alloc;// 申请的总的字节数,不包含结束标识
    unsigned char flags;// 不同SDS的头类型(有5种,如 #define SDS_TYPE_8 1),用来控制SDS的
    // 头大小
    char buf[];
}
// 上述这个SDS结构体保存的最大范围为255,还有其它范围更大的SDS结构体(有5种)
```



### 11. SDS具备动态扩容的能力,如:

SDS之所以叫做动态字符串，是因为它具备动态扩容的能力，例如一个内容为“hi”的SDS：



假如我们要给SDS追加一段字符串“,Amy”，这里首先会申请新内存空间：

- ◆ 如果新字符串小于1M，则新空间为扩展后字符串长度的两倍+1；
- ◆ 如果新字符串大于1M，则新空间为扩展后字符串长度+1M+1。称为**内存预分配**。



### 12. SDS优点: \* 获取字符串长度的时间复杂度为O(1) \* 支持动态扩容 \* 减少内存分配次数 \* 二进制安全 13.

**IntSet**是Redis中集合的一种实现方式,基于整数数组实现,并且具备元素唯一、长度可变、有序(底层采用二分查找来保证有序性)等特征

```
C typedef struct intset { uint32_t encoding;// 编码方式,支持16位、32位、64位的整数 uint32_t length;// 元素个数 int8_t contents[];// 整数数组,保存集合数据 }
```

intset; 14. **intset**默认是升序将整数保存在**contents**中



现在，数组中每个数字都在**int16\_t**的范围内，因此采用的编码方式是**INTSET\_ENC\_INT16**，每部分占用的字节大小为：

- encoding: 4字节
- length: 4字节
- contents: 2字节 \* 3 = 6字节

### 15. **intset**的编码方式可以自动升级,即将编码方式升级到合适的大小

现在，假设有一个intset，元素为{5,10, 20}，采用的编码是INTSET\_ENC\_INT16，则每个整数占2字节：



我们向该其中添加一个数字：50000，这个数字超出了int16\_t的范围，intset会自动升级编码方式到合适的大小。

以当前案例来说流程如下：

- ① 升级编码为INTSET\_ENC\_INT32，每个整数占4字节，并按照新的编码方式及元素个数扩容数组
- ② 倒序依次将数组中的元素拷贝到扩容后的正确位置
- ③ 将待添加的元素放入数组末尾
- ④ 最后，将intset的encoding属性改为INTSET\_ENC\_INT32，将length属性改为4



## 16. Dict由三部分组成,分别是:哈希表、哈希节点(键值对)、字典

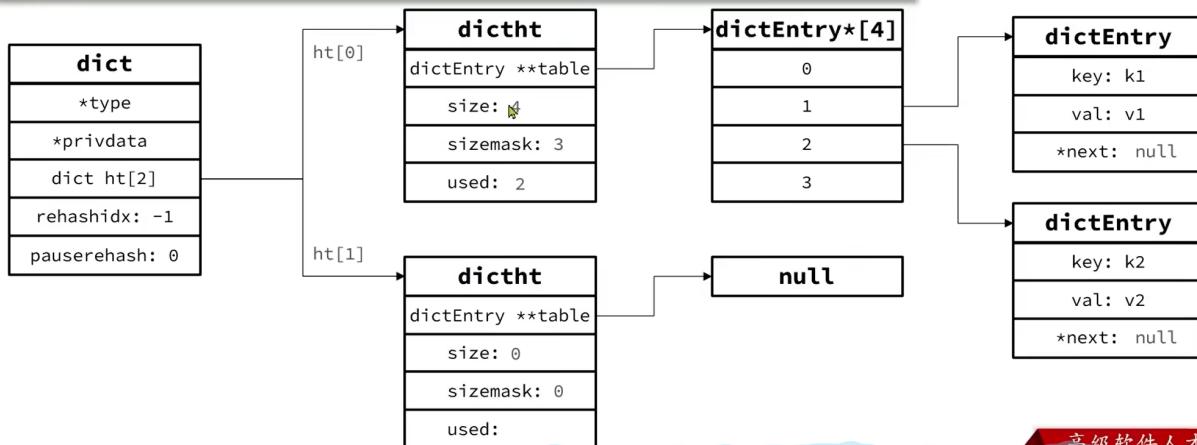
```
typedef struct dictht {
    // entry数组
    // 数组中保存的是指向entry的指针
    dictEntry **table;
    // 哈希表大小
    unsigned long size;
    // 哈希表大小的掩码, 总等于size - 1
    unsigned long sizemask;
    // entry个数
    unsigned long used;
} dictht;

typedef struct dictEntry {
    void *key; // 键
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v; // 值
    // 下一个Entry的指针
    struct dictEntry *next;
} dictEntry;
```

当我们向dict添加键值对时,Redis首先根据key计算出hash值,然后利用h&sizemask来计算元素应该存储到数组中的哪个索引位置

### Dict

```
typedef struct dict {
    dictType *type; // dict类型, 内置不同的hash函数
    void *privdata; // 私有数据, 在做特殊hash运算时用
    dictht ht[2]; // 一个Dict包含两个哈希表, 其中一个是当前数据, 另一个一般是空, rehash时使用
    long rehashidx; // rehash的进度, -1表示未进行
    int16_t pauserehash; // rehash是否暂停, 1则暂停, 0则继续
} dict;
```



高级软件人才培训

17. Dict在每次新增键值对时都会检查负载因子( $\text{LoadFactor} = \text{used}/\text{size}$ ),满足以下两种情况时会触发哈希表扩容:  
\* 哈希表的 $\text{LoadFactor} \geq 1$ ,并且服务器没有执行BGSAVE或者BGREWRITEAOF等后台进程  
\* 哈希表的 $\text{LoadFactor} > 5$

18. Dict每次删除元素时,也会对负载因子做检查,当 $\text{LoadFactor} < 0.1$ 时,会做哈希表收缩.不管是扩容还是收缩,必定会创建新的哈希表,导致哈希表的size和sizemask变化,而key的查询与sizemask有关.因此必须对哈希表中的每一个key重新计算索引,插入新的哈希表,这个过程称为rehash,过程如下:

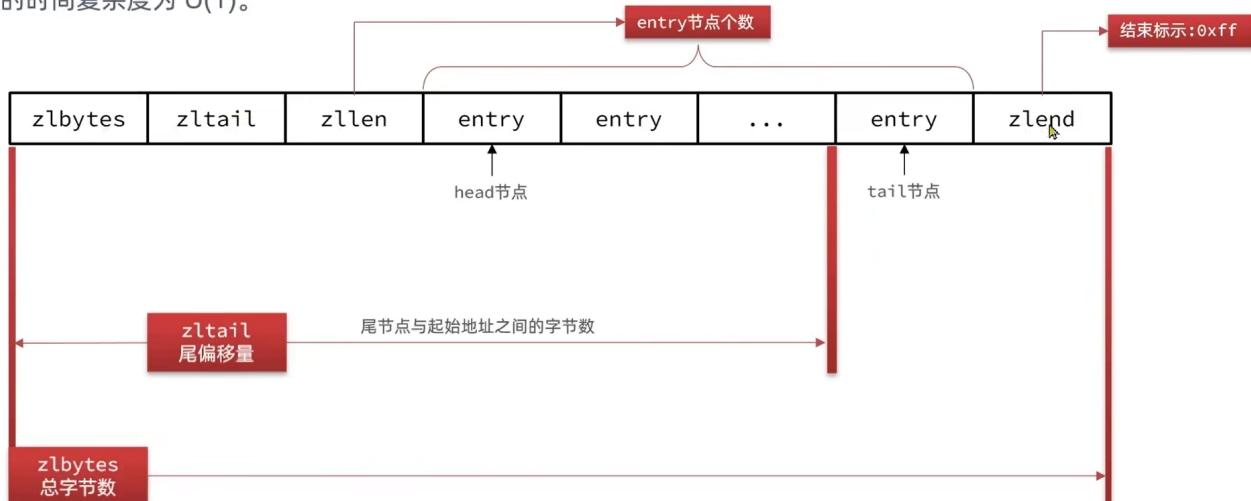
- ① 计算新hash表的realeSize，值取决于当前要做的是扩容还是收缩：
  - ◆ 如果是扩容，则newSize为第一个大于等于dict.ht[0].used + 1的 $2^n$
  - ◆ 如果是收缩，则newSize为第一个大于等于dict.ht[0].used的 $2^n$ （不得小于4）
- ② 按照新的realeSize申请内存空间，创建dictht，并赋值给dict.ht[1]
- ③ 设置dict.rehashidx = 0，标示开始rehash
- ④ 将dict.ht[0]中的每一个dictEntry都rehash到dict.ht[1]
- ⑤ 将dict.ht[1]赋值给dict.ht[0]，给dict.ht[1]初始化为空哈希表，释放原来的dict.ht[0]的内存

Dict的rehash并不是一次性完成的。如果dict中包含数百万的entry，要在一次rehash完成，极有可能导致主线程阻塞。因此dict的rehash是分多次、渐进式的完成，因此称为渐进式rehash（每次访问dict时（增删改查）执行一次rehash），流程为：

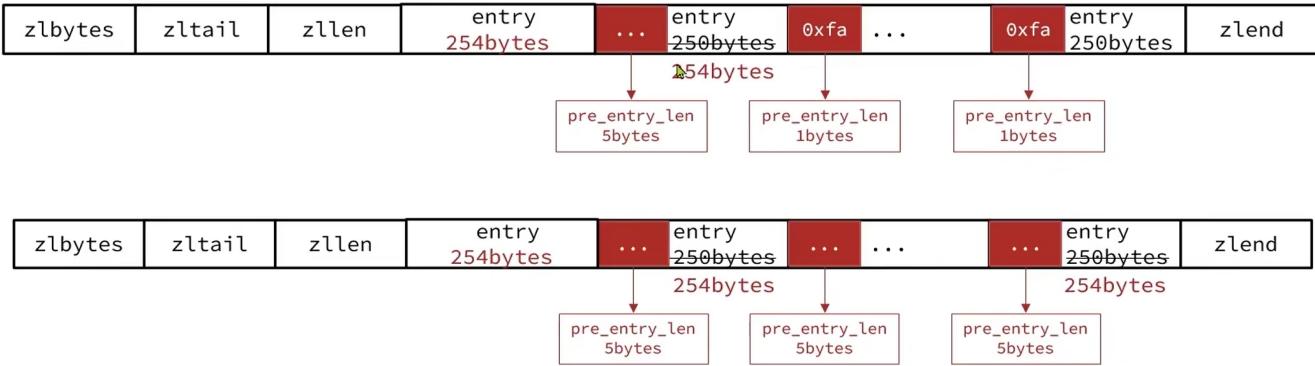
- ① 计算新hash表的size，值取决于当前要做的是扩容还是收缩：
  - ◆ 如果是扩容，则newSize为第一个大于等于dict.ht[0].used + 1的 $2^n$
  - ◆ 如果是收缩，则newSize为第一个大于等于dict.ht[0].used的 $2^n$ （不得小于4）
- ② 按照新的size申请内存空间，创建dictht，并赋值给dict.ht[1]
- ③ 设置dict.rehashidx = 0，标示开始rehash
- ④ 将dict.ht[0]中的每一个dictEntry都rehash到dict.ht[1]
- ⑤ 每次执行新增、查询、修改、删除操作时，都检查一下dict.rehashidx是否大于-1，如果是则将dict.ht[0].table[rehashidx]的entry链表rehash到dict.ht[1]，并且将rehashidx++。直至dict.ht[0]的所有数据都rehash到dict.ht[1]
- ⑥ 将dict.ht[1]赋值给dict.ht[0]，给dict.ht[1]初始化为空哈希表，释放原来的dict.ht[0]的内存
- ⑦ 将rehashidx赋值为-1，代表rehash结束
- ⑧ 在rehash过程中，新增操作，则直接写入ht[1]，查询、修改和删除则会在dict.ht[0]和dict.ht[1]依次查找并执行。这样可以确保ht[0]的数据只减不增，随着rehash最终为空

19. dict底层是数组+链表来解决哈希冲突，dict包含两个哈希表ht[0]平常用，ht[1]用来rehash。20. ziplist是一种特殊的“双端链表”，由一系列特殊编码的连续内存块组成。可以在任意一端进行压入/弹出操作（类似双端队列deque），并且该操作的时间复杂度为O(1)

该操作的时间复杂度为 O(1)。



21. ziplist的entry结构： 22. ziplist的连锁更新问题：在特殊情况下产生的连续多次空间扩展操作称之为连锁更新（概率降低）



、 23. Redis的五种数据结构:String、 list、 hash、 set、 sorted set 24. Redis和LevelDB一样,也有部分数据在硬盘上 25. Redis是单线程的 26. Redis与Memcached的区别

#### 1、存储方式

- Memcache把数据全部存在内存之中，断电后会挂掉，没有持久化功能，数据不能超过内存大小。
- Redis有部份存在硬盘上，这样能保证数据的持久性。

#### 2、数据支持类型

- Memcache对数据类型支持相对简单,只有String这一种类型
- Redis有复杂的数据类型。Redis不仅仅支持简单的k/v类型的数据，同时还提供 list, set, zset, hash等数据结构的存储。

#### 3、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。

4、集群模式：Memcached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；但是 Redis 目前 是原生支持 cluster 模式的。

5、Memcached是多线程，非阻塞IO复用的网络模型；Redis使用单线程的多路 IO 复用模型。

6、Value 值大小不同：Redis 最大可以达到 512MB；Memcached 只有 1MB。

27. Redis为什么这么快? \* 全部操作是纯内存的操作 \* 采用单线程,有效避免了频繁的上下文切换 \* 采用非阻塞I/O多路复用 28. Redis设置过期时间的方案: \* 定期删除:默认每隔100ms就随机抽取一些设置了过期时间的key,检查是否过期,如果过期就删除 \* 惰性删除:指某个键值过期后,此键值不会马上被删除,而是等到下层被使用的时候,才会被检查到过期,此时才能得到删除 29. 定期删除和惰性删除不能保证一定删除数据,Redis采用内存淘汰机制来确保数据一定被删除 30. Redis是通过I/O多路复用来处理多个客户端请求 31. 缓存雪崩:指的是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 32. 缓存穿透:指查询一个一定不存在的数据,由于缓存不命中,接着查询数据库也无法查询出结果,因此也不会写入到缓存中,这会导致每个查询都会去请求数据库,造成缓存穿透(布隆过滤器解决) 33. 缓存击穿:指一个key非常热点,大并发集中对这一个点访问,当这个key失效瞬间,持续的大并发就穿破缓存,直接请求数据库 34. Redis可以作为中间缓存使用 35. Redis的持久化机制: \* 快照持久化:Redis可以通过创建快照来获得存储在内存里面的数据在某个时间点的副本.创建快照后,可以对快照进行备份,可以将快照复制到其它服务器从而创建具有相同数据的服务器副本,还快也将快照留在原地以便重启服务器的时候使用 \* AOF持久化 36. Redis为什么是单线程的? Redis是纯内存操作,执行速度非常快,因为性能瓶颈不在CPU上,因此不需要用多线程提升性能.并且多线程模型会带来并发安全问题和页面切换的系统开销

## 分布式缓存

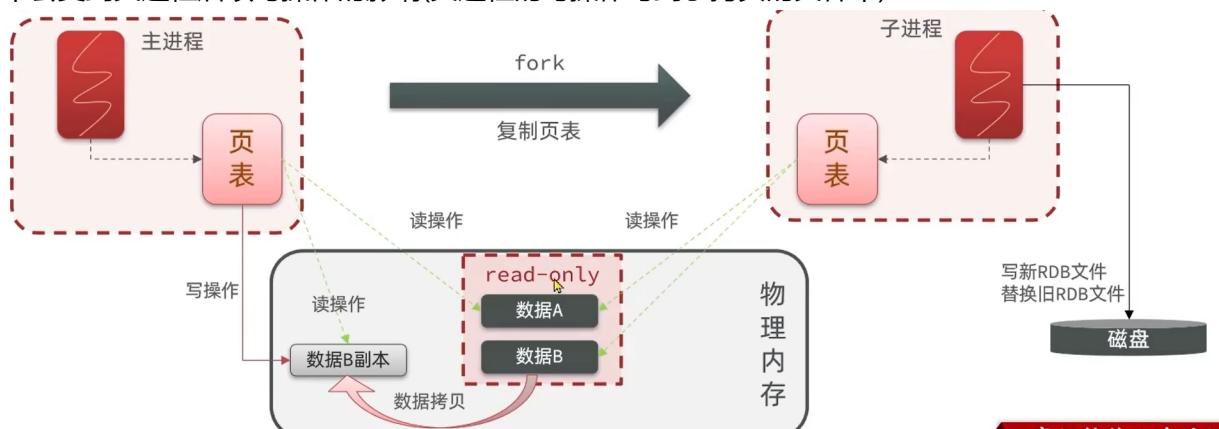
1. 单点部署Redis服务会发生一旦宕机,就不可用了.因此为了实现高可用,会将数据库复制到多个副本以部署在不同的服务器上,其中一台挂了也可以继续提供服务,Redis实现高可用的三种部署模式:主从模式、哨兵模式、集群模式

# RDB持久化

## 1. save和bgsave



2. bgsave中的异步RDB持久化时,操作系统的copy-on-write技术确保了子进程中的数据是执行fork()时的状态,不会受到父进程后续写操作的影响(父进程的写操作写到了拷贝的文件中)



# AOF持久化

1. AOF持久化,采用日志的形式来记录每个写操作,追加到文件中,重启时再重新执行AOF文件中的命令来恢复数据.它主要解决数据持久化的实时性问题.默认是不开启的
2. AOF持久化记录的是命令,RDB记录的是具体的值
3. AOF数据更完整,RDB一般可能是60秒记录一次(因为bgsave里面会fork一个子进程,然后读写会有很多磁盘IO操作,因此较为耗时),因此RDB不够完整,而AOF一般每秒记录一次

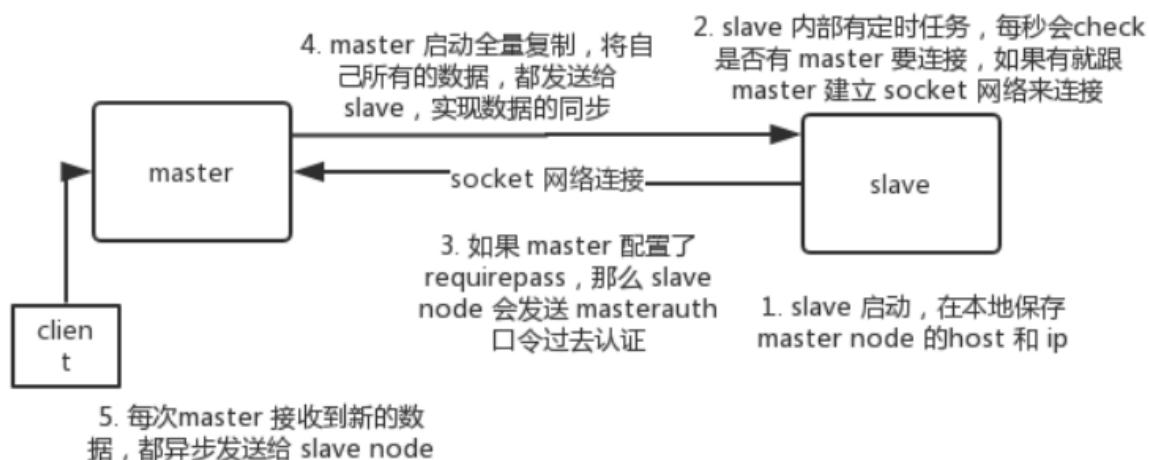
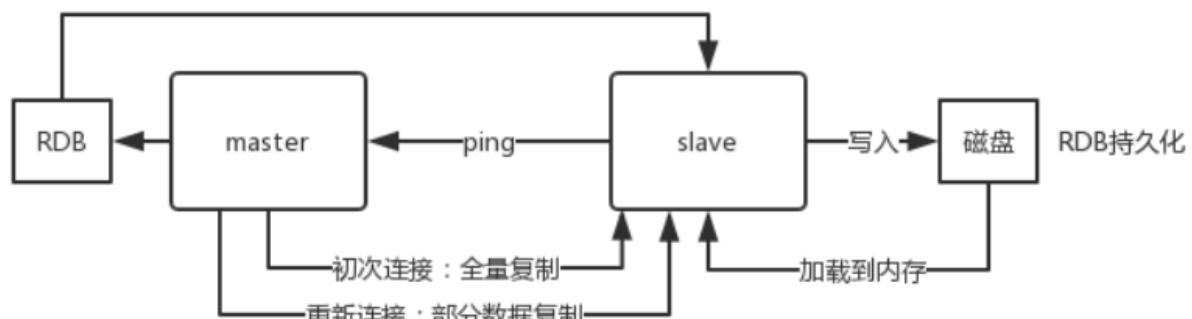
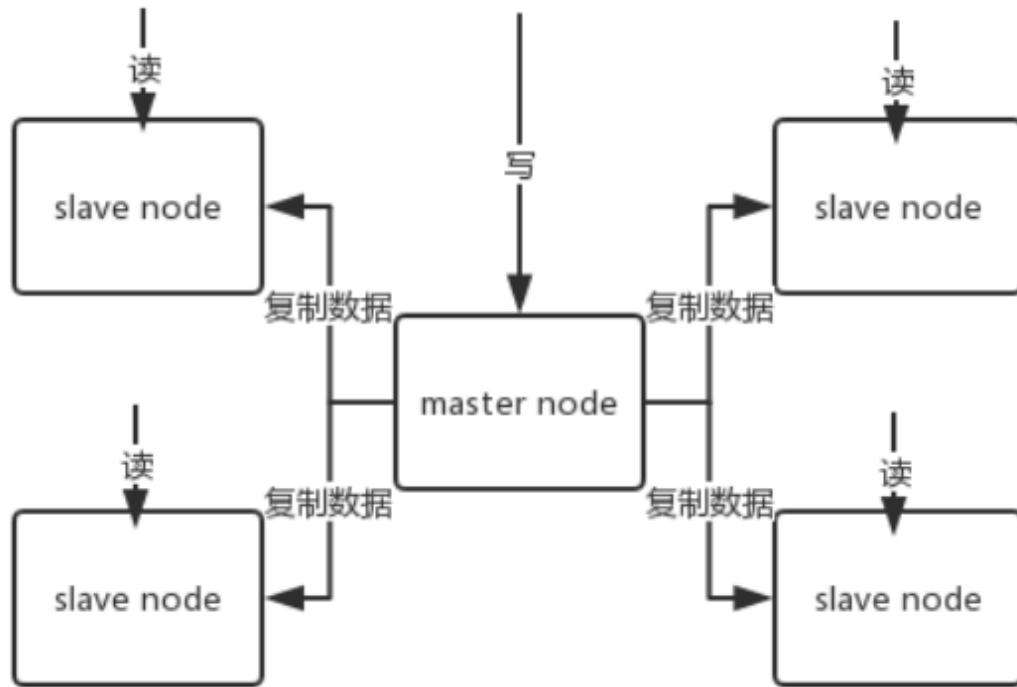
RDB和AOF各有自己的优缺点,如果对数据安全性要求较高,在实际开发中往往会结合两者来使用。

	RDB	AOF
持久化方式	定时对整个内存做快照	记录每一次执行的命令
数据完整性	不完整,两次备份之间会丢失	相对完整,取决于刷盘策略
文件大小	会有压缩,文件体积小	记录命令,文件体积很大
宕机恢复速度	很快	慢
数据恢复优先级	低,因为数据完整性不如AOF	高,因为数据完整性更高
系统资源占用	高,大量CPU和内存消耗	低,主要是磁盘IO资源 但AOF重写时会占用大量CPU和内存资源
使用场景	可以容忍数分钟的数据丢失,追求更快的启动速度	对数据安全性要求较高常见

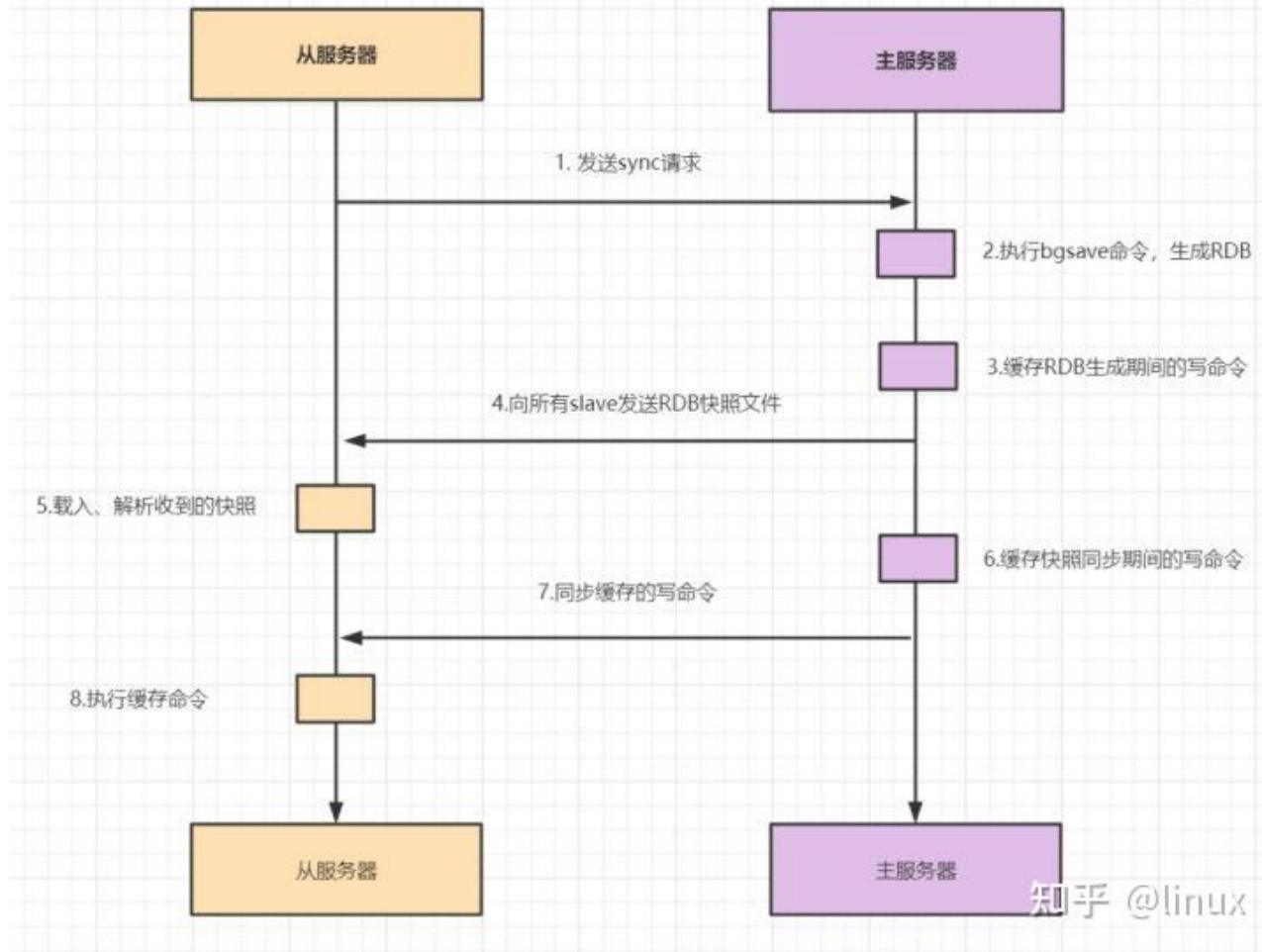


# 主从模式

1. 主从模式中,Redis部署了多台机器,有主节点,负责读写操作(主要是写),有从节点,只负责读操作.从节点的数据来自主节点,实现原理就是主从复制机制



2. 主从复制包括全量复制、增量复制两种.一般当 slave 第一次启动连接 master,或者认为是第一次连接,就采用全量复制,其流程为:



### 3. 增量复制:

#### 增量复制

- 如果全量复制过程中，master-slave 网络连接断掉，那么 slave 重新连接 master 时，会触发增量复制。
- master 直接从自己的 backlog 中获取部分丢失的数据，发送给 slave node，默认 backlog 就是 1MB。
- master 就是根据 slave 发送的 psync 中的 offset 来从 backlog 中获取数据的。

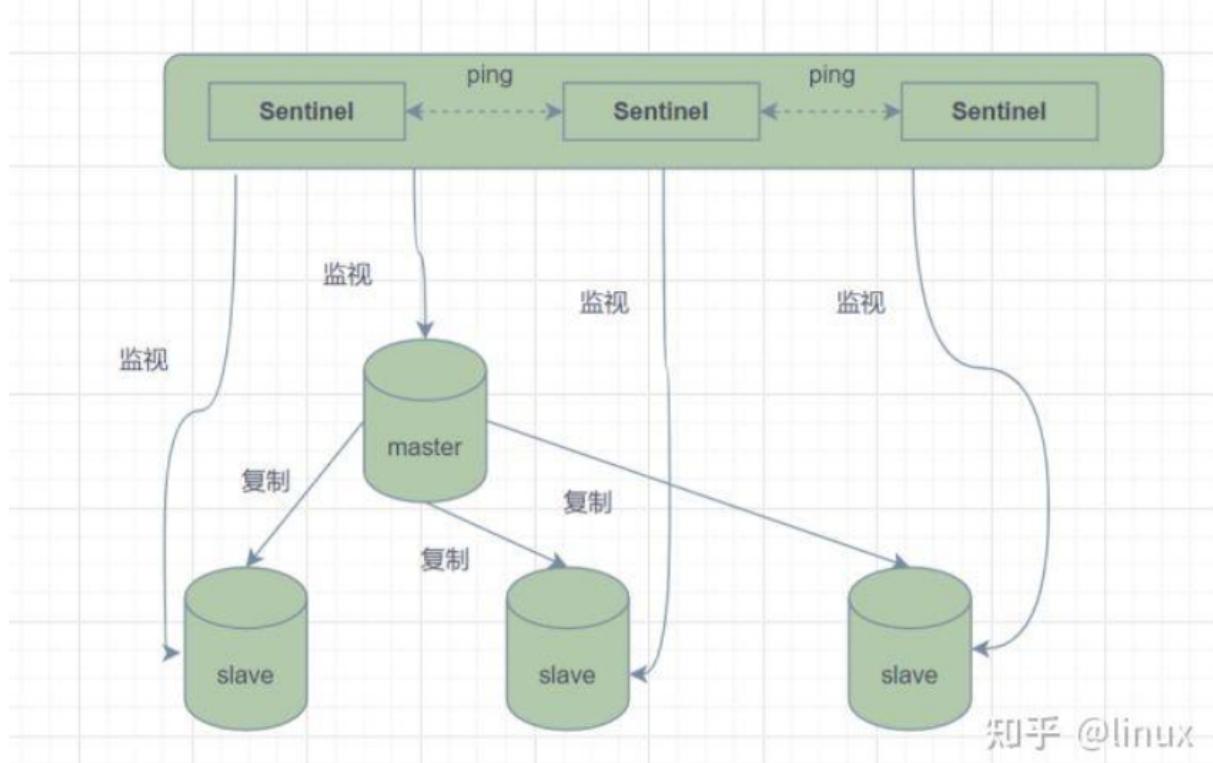
4. 主从复制的断点续传:如果主从复制过程中,网络连接断掉了,那么可以接着上次复制的地方,继续复制下去,而不是从头开始复制一份.master node会在内存中维护一个backlog, master和slave都会保存一个replica offset还有一个master runid, offset就是保存在backlog中.如果master和slave网络连接断掉了,slave会让master从上次replica offset开始继续复制,如果没有找到对应的offset,那么就会执行一次resynchronization

5. Redis的高可用架构,叫做故障转移,也可以叫做主备切换

## 哨兵模式

1. 主从模式中，一旦主节点由于故障不能提供服务，需要人工将从节点晋升为主节点，同时还要通知应用方更新主节点地址。显然，多数业务场景都不能接受这种故障处理方式。Redis从2.8开始正式提供了Redis Sentinel（哨兵）架构来解决这个问题
2. 哨兵模式，由一个或多个Sentinel实例组成的Sentinel系统，它可以监视所有的Redis主节点和从节点，并在被监视的主节点进入下线状态时，自动将下线主服务器属下的某个从节点升级为新的主节点。但是呢，一个哨兵进程对Redis节点进行监控，就可能会出现问题（单点问题），因此，可以使用多个哨兵来

进行监控Redis节点，并且各个哨兵之间还会进行监控

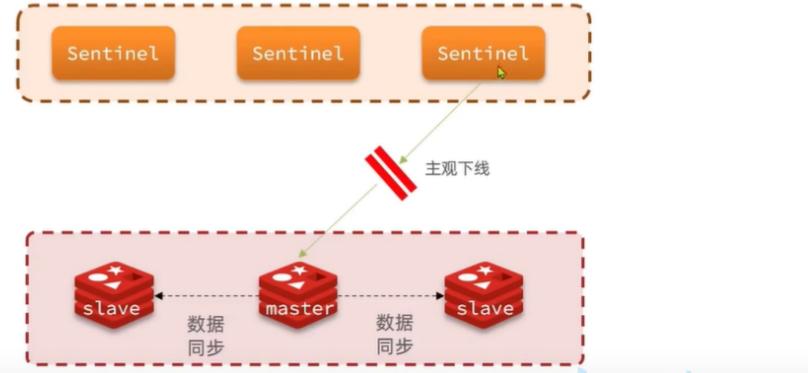


### 3. 哨兵的三个作用

- 监控(master、 slave)

Sentinel基于心跳机制监测服务状态，每隔1秒向集群的每个实例发送ping命令：

- 主观下线：如果某sentinel节点发现某实例未在规定时间响应，则认为该实例**主观下线**。
- 客观下线：若超过指定数量（quorum）的sentinel都认为该实例主观下线，则该实例**客观下线**。quorum值最好超过Sentinel实例数量的一半。



- 故障转移(切换从节点为主节点)
- 通知(通过新的master地址)

4. 哨兵+Redis的主从部署架构,不能保证数据零丢失,只能保证Redis集群的高可用性

5. 哨兵至少需要3个实例来保证自己的健壮性

6. 哨兵主从切换的数据丢失问题:

- 异步复制导致的数据丢失:master->slave的复制是异步的,所以可能有部分数据还没复制到 slave, master就宕机了,此时这部分数据就丢失了
- 脑裂导致的数据丢失:脑裂:某个master所在机器突然脱离了正常的网络,跟其他slave机器不能连接,但是实际上master还允许着.此时哨兵可能就会认为master宕机了,然后开启选举,将其它slave切换成master.这个时候,集群里就会有两个master,也就是所谓的脑裂.此时虽然某个slave被切换成了master,但是可能客户端还没来得及切换到新的master,还继续向旧master写数据.因此旧master恢复时,就会被作为一个slave挂到新的master上去,自己的数据会清空,重新从新的master复制数据.而新的master并没有后来客户端写入的数据,因此,这部分数据就丢失了

## ◦ 解决办法:

```
1 min-slaves-to-write 1
2 min-slaves-max-lag 10Copy to clipboardErrorCopied
```

sh

表示，要求至少有 1 个 slave，数据复制和同步的延迟不能超过 10 秒。

如果说一旦所有的 slave，数据复制和同步的延迟都超过了 10 秒钟，那么这个时候，master 就不会再接收任何请求了。

- 减少异步复制数据的丢失

有了 `min-slaves-max-lag` 这个配置，就可以确保说，一旦 slave 复制数据和 ack 延时太长，就认为可能 master 宕机后损失的数据太多了，那么就拒绝写请求，这样可以把 master 宕机时由于部分数据未同步到 slave 导致的数据丢失降低的可控范围内。

- 减少脑裂的数据丢失

如果一个 master 出现了脑裂，跟其他 slave 丢了连接，那么上面两个配置可以确保说，如果不能继续给指定数量的 slave 发送数据，而且 slave 超过 10 秒没有给自己 ack 消息，那么就直接拒绝客户端的写请求。因此在脑裂场景下，最多就丢失 10 秒的数据。

## 7. `sdown`: 主管下线; `odown`: 客观下线

## 8. 利用哨兵时的主从切换选举算法:

如果一个 master 被认为 `odown` 了，而且 majority 数量的哨兵都允许主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个 slave 来，会考虑 slave 的一些信息：

- 跟 master 断开连接的时长
- slave 优先级
- 复制 offset
- run id

如果一个 slave 跟 master 断开连接的时间已经超过了 `down-after-milliseconds` 的 10 倍，外加 master 宕机的时长，那么 slave 就被认为不适合选举为 master。

```
1 (down-after-milliseconds * 10) + milliseconds_since_master_is_in_SDOWN_stateCopy to clipboardErrorCopied
```

接下来会对 slave 进行排序：

- 按照 slave 优先级进行排序，slave priority 越低，优先级就越高。
- 如果 slave priority 相同，那么看 replica offset，哪个 slave 复制了更多的数据，offset 越靠后，优先级就越高。
- 如果上面两个条件都相同，那么选择一个 run id 比较小的那个 slave。

## 9. 哨兵完成切换后，会向其它哨兵进行 `configuration` 传播:

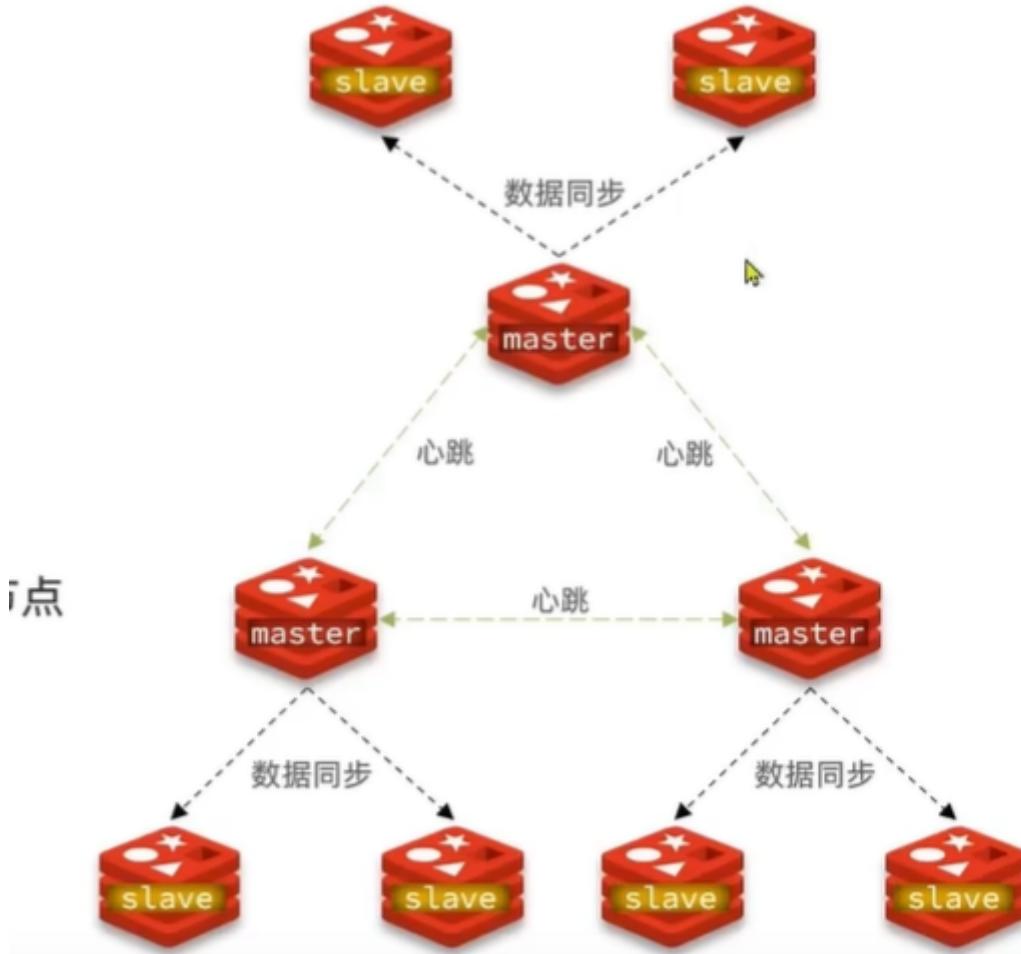
哨兵完成切换之后，会在自己本地更新生成最新的 master 配置，然后同步给其他的哨兵，就是通过之前说的 `pub/sub` 消息机制。

这里之前的 version 号就很重要了，因为各种消息都是通过一个 channel 去发布和监听的，所以一个哨兵完成一次新的切换之后，新的 master 配置是跟着新的 version 号的。其他的哨兵都是根据版本号的大小来更新自己的 master 配置的。

# Cluster 集群模式

1. 主从机制和哨兵机制可以解决高可用、高并发读的问题，但是没有解决海量数据存储(每个节点存储的数据是一样的，主从同步)。分片集群模式可以解决，`Cluster` 集群中有多个 `master`，每个 `master` 保存不同数据(利用哈希槽来访问每个 `master` 中的数据)，每个 `master` 也可以基于主从机制，即有多个 `slave` 节点；`master`

之间通过ping监测彼此状态(充当哨兵)



2. Cluster的分布式寻址算法:Redis会把每一个master节点映射到0~16383共16384个插槽(hash slot)上。插槽算法把整个数据库被分为16384个槽,每个进入Redis的键值对,根据key进行散列,分配到这16384插槽中的一个.使用的哈希映射也比较简单,用CRC16算法计算出一个16位的值,再对16384取模.数据库中的每个键都属于这16384个槽的其中一个,集群中的每个节点都可以处理这16384个槽

集群中的每个节点负责一部分的hash槽, 比如当前集群有A、B、C个节点, 每个节点上的哈希槽数 = $16384/3$ , 那么就有:

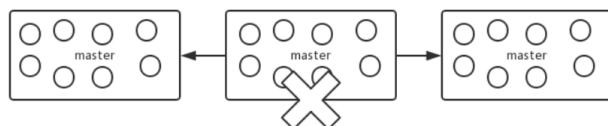
- 节点A负责0~5460号哈希槽
- 节点B负责5461~10922号哈希槽
- 节点C负责10923~16383号哈希槽

#### # Redis cluster 的 hash slot 算法

Redis cluster 有固定的 16384 个 hash slot, 对每个 key 计算 CRC16 值, 然后对 16384 取模, 可以获取 key 对应的 hash slot。

Redis cluster 中每个 master 都会持有部分 slot, 比如有 3 个 master, 那么可能每个 master 持有 5000 多个 hash slot。hash slot 让 node 的增加和移除很简单, 增加一个 master, 就将其 master 的 hash slot 移动部分过去, 减少一个 master, 就将它的 hash slot 移动到其他 master 上去。移动 hash slot 的成本是非常低的。客户端的 api, 可以对指定的数据, 让他们走同一个 hash slot, 通过 hash tag 来实现。

任何一台机器宕机, 另外两个节点, 不影响的。因为 key 找的是 hash slot, 不是机器。



3. 数据key不是与节点绑定,而是与插槽绑定.Redis会先根据key的有效部分计算插槽值(有效部分利用CRC16算法得到哈希值,然后再对16384取模得到的结果就是这个key对应的插槽值)

## Hash Slot插槽算法

既然是分布式存储，Cluster集群使用的分布式算法是一致性Hash嘛？并不是，而是Hash Slot插槽算法。

插槽算法把整个数据库被分为16384个slot（槽），每个进入Redis的键值对，根据key进行散列，分配到这16384插槽中的一个。使用的哈希映射也比较简单，用CRC16算法计算出一个16位的值，再对16384取模。数据库中的每个键都属于这16384个槽的其中一个，集群中的每个节点都可以处理这16384个槽。

集群中的每个节点负责一部分的hash槽，比如当前集群有A、B、C三个节点，每个节点上的哈希槽数 =  $16384/3$ ，那么就有：

- 节点A负责0~5460号哈希槽
- 节点B负责5461~10922号哈希槽
- 节点C负责10923~16383号哈希槽

4. 集群模式的故障转移的时候主从切换不需要哨兵，因为它们master节点会相互通过心跳机制监控，即它们自己起到了哨兵的作用

5. Redis Cluster的高可用和主备切换原理

### Redis cluster 的高可用与主备切换原理

Redis cluster 的高可用的原理，几乎跟哨兵是类似的。

#### # 判断节点宕机

如果一个节点认为另外一个节点宕机，那么就是 `pfail`，主观宕机。如果多个节点都认为另外一个节点宕机了，那么就是 `fail`，客观宕机，跟哨兵的原理几乎一样，`sdown`, `odown`。在 `cluster-node-timeout` 内，某个节点一直没有返回 `pong`，那么就被认为 `pfail`。

如果一个节点认为某个节点 `pfail` 了，那么会在 `gossip ping` 消息中，`ping` 给其他节点，如果超过半数的节点都认为 `pfail` 了，那么就会变成 `fail`。

#### 从节点过滤

对宕机的 master node，从其所有的 slave node 中，选择一个切换成 master node。

检查每个 slave node 与 master node 断开连接的时间，如果超过了 `cluster-node-timeout * cluster-slave-validity-factor`，那么就没有资格切换成 master。

#### 从节点选举

每个从节点，都根据自己对 master 复制数据的 offset，来设置一个选举时间，offset 越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举。

所有的 master node 开始 slave 选举投票，给要进行选举的 slave 进行投票，如果大部分 master node ( $N/2 + 1$ ) 都投票给了某个从节点，那么选举通过，那个从节点可以切换成 master。

从节点执行主备切换，从节点切换为主节点。

#### 与哨兵比较

整个流程跟哨兵相比，非常类似，所以说，Redis cluster 功能强大，直接集成了 replication 和 sentinel 的功能。

6. 为了保证高可用性，Cluster集群也用了主从模式

## 7. 集群的完整性问题:

### 集群完整性问题

在Redis的默认配置中，如果发现任意一个插槽不可用，则整个集群都会停止对外服务：

```
# By default Redis Cluster nodes stop accepting queries if they detect there
# is at least a hash slot uncovered (no available node is serving it).
# This way if the cluster is partially down (for example a range of hash slots
# are no longer covered) all the cluster becomes, eventually, unavailable.
# It automatically returns available as soon as all the slots are covered again.
#
# However sometimes you want the subset of the cluster which is working,
# to continue to accept queries for the part of the key space that is still
# covered. In order to do so, just set the cluster-require-full-coverage
# option to no.
#
# cluster-require-full-coverage yes
```

为了保证高可用特性，这里建议将 cluster-require-full-coverage 配置为 false

## 8. Cluster 节点间采用 Gossip 协议通信

## 9. Gossip 协议包含多种消息, ping、pong、meet、fail 等

# Redis分布式锁

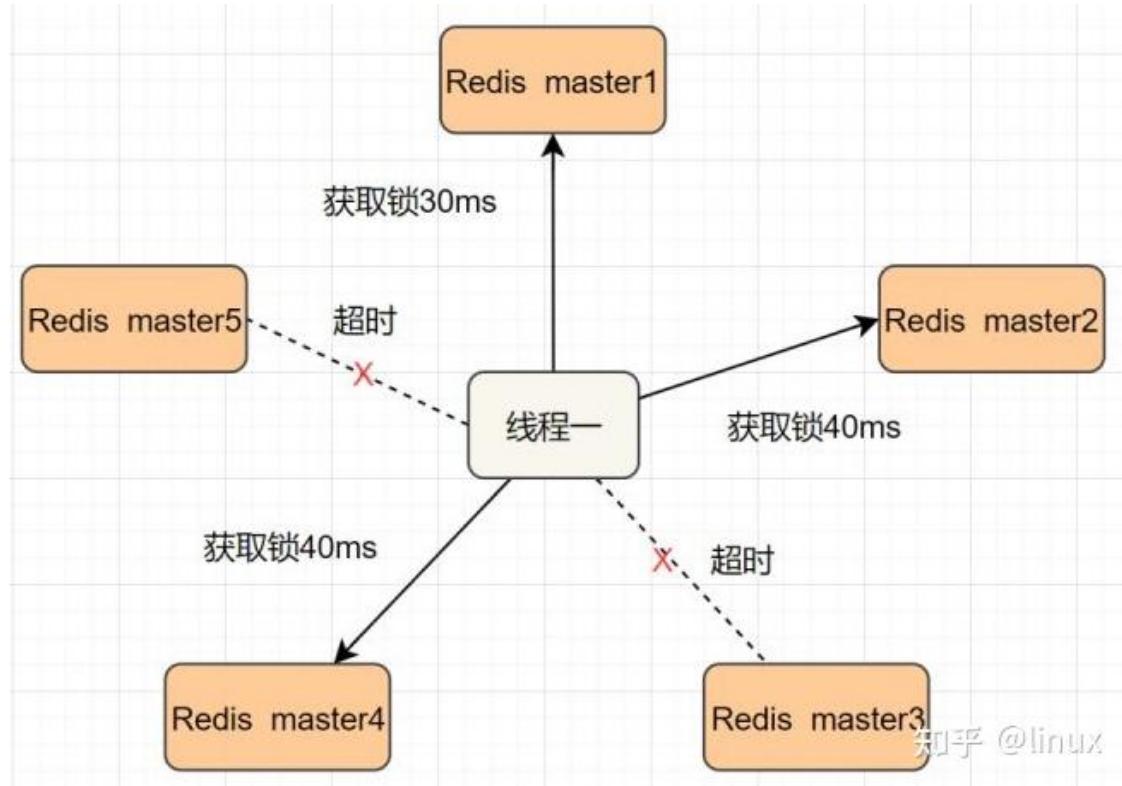
---

## 1. 分布式锁:是控制分布式系统不同进程共同访问共享资源的一种锁的实现

### 2. 一共有五种方式:

- **setnx + expire** 分开写:如果执行完 **setnx** 加锁,正要执行 **expire** 设置过期时间时,进程 **crash** 掉或者要重启维护了,那这个锁就“长生不老”了,别的线程永远获取不到锁啦,所以分布式锁不能这么实现
- **setnx ex px nx**:存在锁过期释放了,业务还没执行完的问题和锁被别的线程误删
- **setnx ex px nx + 校验唯一随机值**:存在锁过期释放了,业务还没执行完的问题
- **Redisson**:只要线程一加锁成功,就会启动一个 **watch dog** 看门狗,它是一个后台线程,会每隔 10 秒检查一下,如果线程 1 还持有锁,那么就会不断的延长锁 **key** 的生存时间.因此,Redisson 就是使用 Redisson 解决了锁过期释放,业务没执行完问题
- **Redlock**:如果线程一在 Redis 的 **master** 节点上拿到了锁,但是加锁的 **key** 还没同步到 **slave** 节点.恰好这时, **master** 节点发生故障,一个 **slave** 节点就会升级为 **master** 节点.线程二就可以获取同个 **key** 的锁啦(因为之前没有同步成功,可以从之前那个 **slave** 节点获取锁),但线程一也已经拿到锁了,锁的安全性就没了.因此,使用了 Redlock 算法:用多个 **master** 部署,保证它们不会同时宕机掉,并且这些 **master** 节点是完全相互独立的,相互之间不存在数据同步
  - 按顺序向 5 个 **master** 节点请求加锁
  - 根据设置的超时时间来判断, 是不是要跳过该 **master** 节点
  - 如果大于等于三个节点加锁成功, 并且使用的时间小于锁的有效期, 即可认定加锁成功啦

- 如果获取锁失败,解锁



3. 分布式锁的特性:互斥性、安全性、死锁、容错

## Redis过期策略

1. 定时过期、惰性过期、定期过期(定时抽样检查)

### 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对key进行清除。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。

2. Redis使用了惰性过期和定期过期

# Redis内存淘汰策略

---

1. **volatile-lru**:当内存不足以容纳新写入数据时，从设置了过期时间的key中使用LRU（最近最少使用）算法进行淘汰
2. **allkeys-lru**:当内存不足以容纳新写入数据时，从所有key中使用LRU（最近最少使用）算法进行淘汰
3. **volatile-lfu**:当内存不足以容纳新写入数据时，从设置了过期时间的key中使用LFU（最近最少使用）算法进行淘汰
4. **allkeys-lfu**:当内存不足以容纳新写入数据时，从所有key中使用LFU（最近最少使用）算法进行淘汰
5. **volatile-random**:当内存不足以容纳新写入数据时，从设置了过期时间的key中随机淘汰数据
6. **allkeys-random**:当内存不足以容纳新写入数据时，从所有key中随机淘汰数据
7. **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key中，根据过期时间进行淘汰，越早过期的优先被淘汰
8. **noeviction**:默认策略，当内存不足以容纳新写入数据时，新写入操作会报错

# Redis通信协议

---

1. Redis是一个CS架构的软件。Redis采用RESP协议通信
2. RESP通过首字节的字符来区分不同数据类型，常用的数据类型有五种：单行字符串、错误、数值、多行字符串、数组
3. RESP主要有实现简单、解析速度块、可读性好等优点

# Redis网络模型

---

1. Redis到底是单线程还是多线程？仅考虑Redis的核心业务部分（命令处理），那么它就是单线程；如果聊整个Redis，那么就是多线程
2. Redis 6.0在核心网络模型中引入了多线程，进一步提高对于多核CPU的利用率

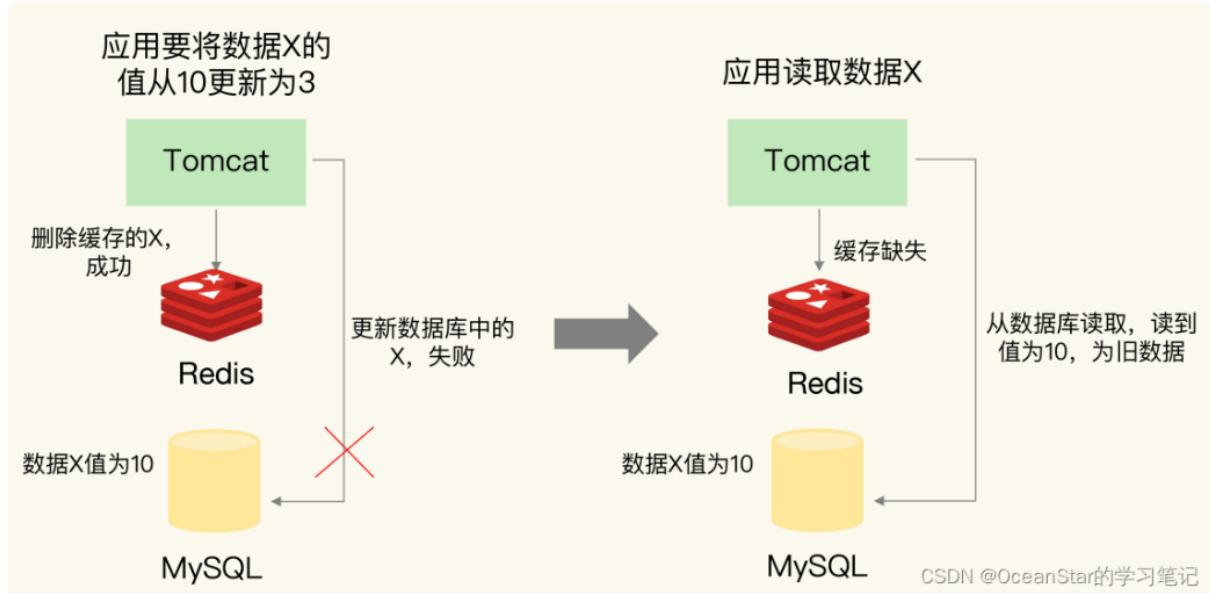
# Redis和MySQL如何保持一致性

---

1. 涉及两部分：
  - 对于缓存数据是采用修改还是删除
  - 先删缓存再更新数据库，还是先更新数据库再删缓存

## 2. Redis缓存和数据库出现的一致性问题,比如

- 我们假设应用先删除缓存,再更新数据库,如果缓存删除成功,但是数据库更新失败,那么,应用再访问数据时,缓存中没有数据,就会发生缓存缺失。然后,应用再访问数据库,但是数据库中的值为旧值,应用就访问到旧值了。



在更新数据库和删除缓存值的过程中,无论是先更新缓存再更新数据库还是先更新数据库再更新缓存,只要有一个操作失败了,就会导致客户端读取到旧值,即出现不一致问题

### 3. 修改一条数据用删除的逻辑是最好的,因为删除操作很简单,而修改的付出的成本很高.为什么是删除缓存,而不是更新缓存?

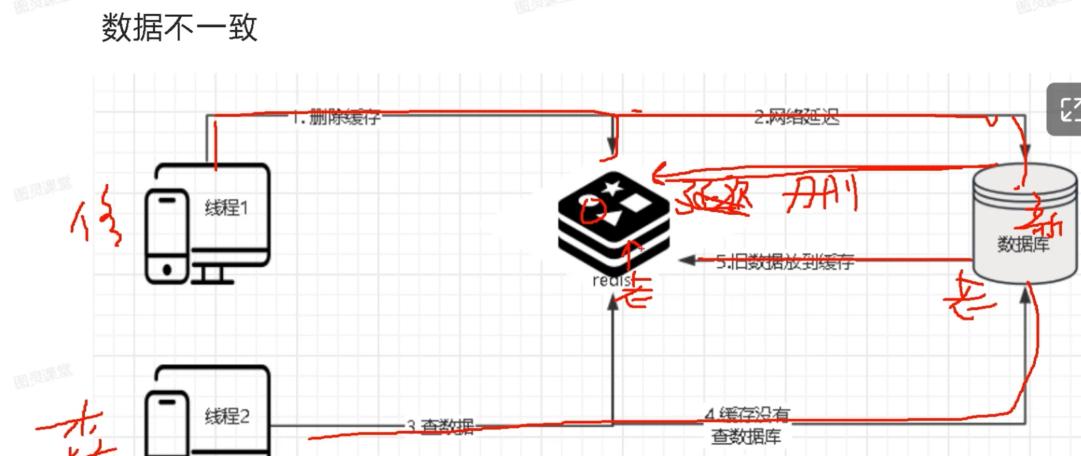
另外更新缓存的代价有时候是很高的。是不是说,每次修改数据库的时候,都一定要将其对应的缓存更新一份?也许有的场景是这样,但是对于**比较复杂的缓存数据计算的场景**,就不是这样了。如果你频繁修改一个缓存涉及的多个表,缓存也频繁更新。但是问题在于,这个缓存到底会不会被频繁访问到?

举个栗子,一个缓存涉及的表的字段,在1分钟内就修改了20次,或者是100次,那么缓存更新20次、100次;但是这个缓存在1分钟内只被读取了1次,有**大量的冷数据**。实际上,如果你只是删除缓存的话,那么在1分钟内,这个缓存不过就重新计算一次而已,开销大幅度降低。**用到缓存才去算缓存**。

其实删除缓存,而不是更新缓存,就是一个lazy计算的思想,不要每次都重新做复杂的计算,不管它会不会用到,而是让它到需要被使用的时候再重新计算。像mybatis, hibernate, 都有懒加载思想。查询一个部门,部门带了一个员工的list,没有必要说每次查询部门,都把里面的1000个员工的数据也同时查出来啊。80%的情况,查这个部门,就只是要访问这个部门的信息就可以了。先查部门,同时要访问里面的员工,那么这个时候只有在你要访问里面的员工的时候,才会去数据库里面查询1000个员工。

### 4. 延迟双删(先删除缓存再更新数据库)

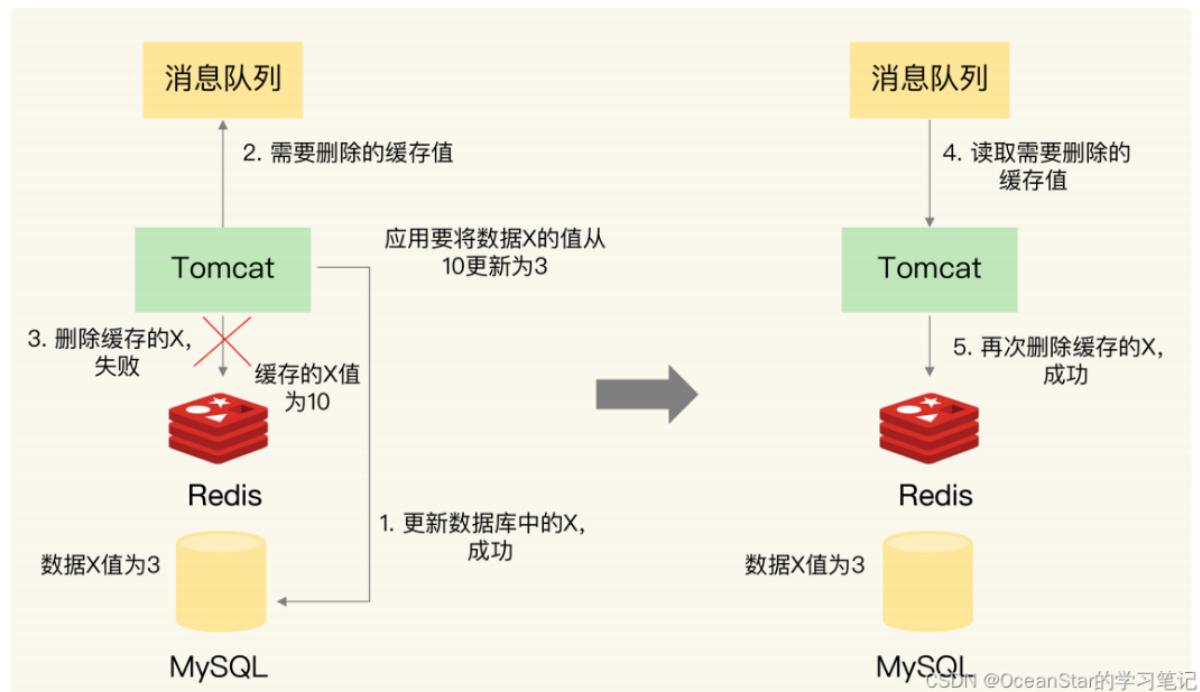
- 先删除缓存
- 再更新数据库
- 休眠一会,再次删除缓存



### 5. 删除缓存重试(先更新数据库再删除缓存):延迟双删里面的第二次删除可能失败,删除重试就是为了解决这个问题:

- 可以把要删除的缓存值或者要更新的数据库值暂存到消息队列中。当应用没有能够成功的删除缓存值或者是更新数据库值时,可以从消息队列中重新读取这些值,然后再次进行删除或者更新

- 如果能够成功删除或者更新，我们就要把这些值从消息队列中去除，以免重复操作，此时，我们也可以保证数据库和缓存一致了。否则的话，我们还需要再次进行重试。如果重试超过一定次数，还是没有成功，我们就需要向应用层发送报错信息了



(先操作数据库,再操作缓存)

- Binlog同步策略(如:把删除redis的操作变更的数据记录到cannal客户端):加了删除重试,会引入消息队列这些业务代码侵入,因此提出了Binlog同步,即利用数据库的二进制日志来实现数据的同步
- 建议优先更新数据库再删除缓存:先删除缓存值在更新数据库,有可能导致请求因为缓存缺失而访问数据库,给数据库带来压力;如果业务应用中读取数据库和写缓存的时间不好估算,那么,延时双删中的等待时间就不好设置

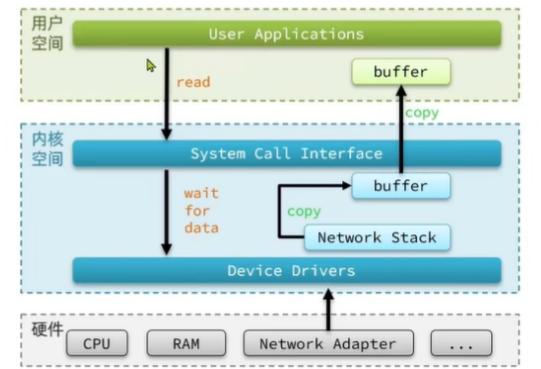
## Redis的事务机制

- Redis事务就是顺序性、一次性、排他性的执行一个队列里的一系列命令
- Redis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(顺序性),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(排他性)
- Redis事务里面没有隔离级别概念
- Redis事务不保证原子性(一个事务所有操作要么全部成功, 要么全部失败)
- Redis事务没有回滚概念
- Redis事务本身不提供持久性保证, 数据的持久性依赖于Redis的持久化机制
- Redis事务的三个阶段
  - 开始事务(MULTI)
  - 命令入队
  - 执行事务(EXEC)、撤销事务(DISCARD)
- 撤销事务会把入队的命令给不执行,即还是没有开启事务前的状态
- 如果事务入队中有错误命令,那么根据不同错误有不同的处理
  - 语法错误(入队过程的错误):Redis会拒绝执行整个事务
  - 执行错误(EXEC阶段):Redis会继续执行事务中的其他命令

## Redis网络模型

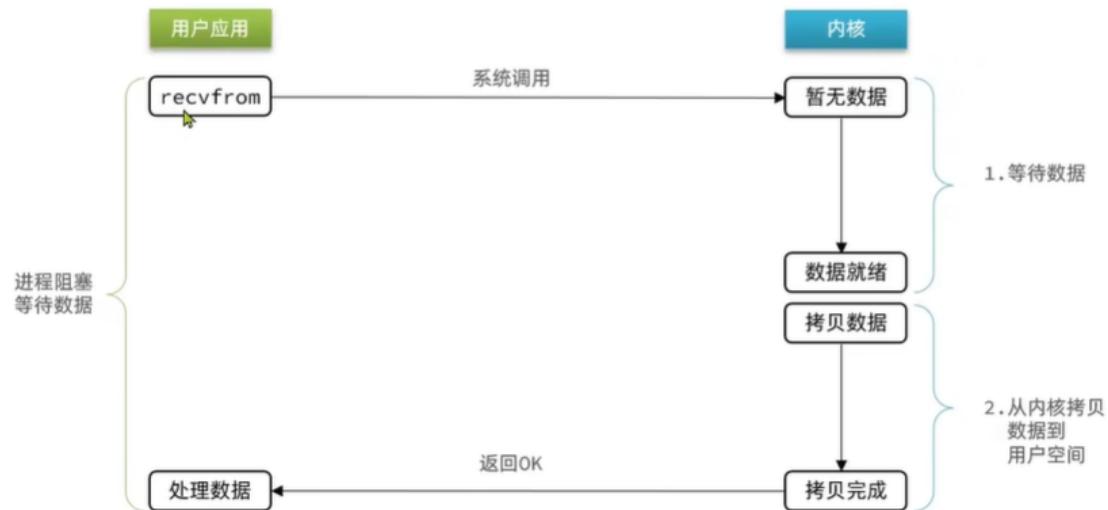
# 用户空间和内核空间

1. 为了避免用户应用导致冲突甚至内核崩溃, 用户应用和内核应该是分离的
2. 进程的寻址空间(进程只能访问虚拟内存空间, 然后虚拟内存会对虚拟地址进行映射寻址到真实的物理内存中去)分为内核空间(32位系统为例, 高地址的1GB)、用户空间(32位系统为例, 低地址的3GB)
3. 用户空间只能执行受限的命令, 而且不能直接调用系统资源, 必须通过内核提供的接口(如socket)来访问
4. 内核空间可以执行特权命令, 调用一切系统资源
5. Linux系统为了提高IO效率,会在用户空间和内核空间都加入缓冲区;
  - 写数据时,要把用户缓冲数据拷贝到内核缓冲区,然后写入底层设备(如磁盘)
  - 读数据时,要从设备读取数据到内核缓冲区,然后拷贝到用户缓冲区

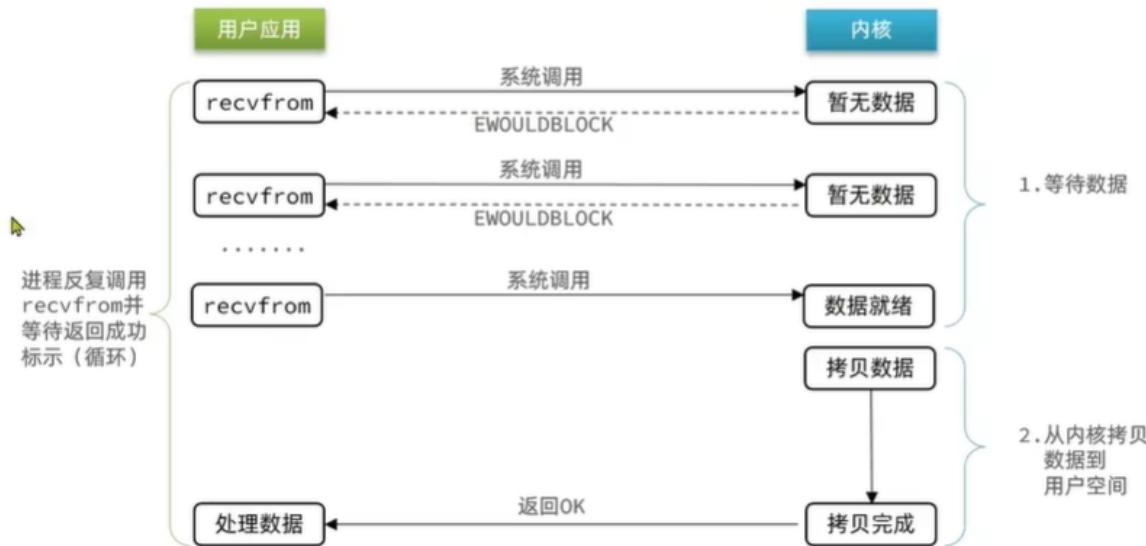


## Linux五种不同IO模型

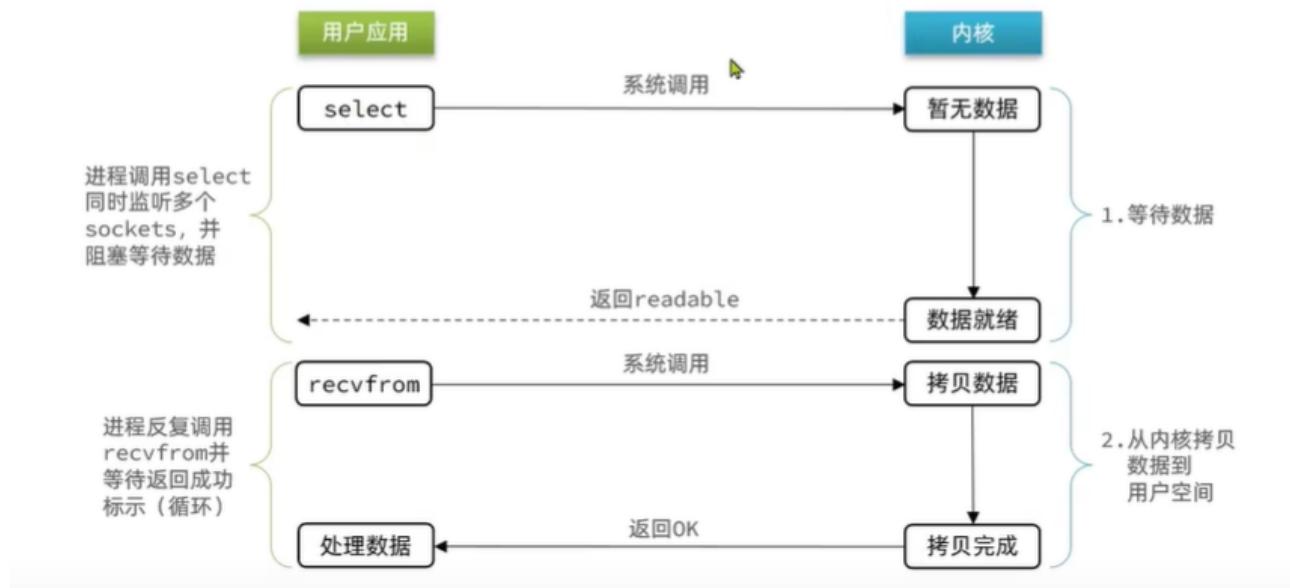
1. 阻塞IO:阻塞等待数据就绪



## 2. 非阻塞IO:通过fcntl来设置,默认是阻塞的

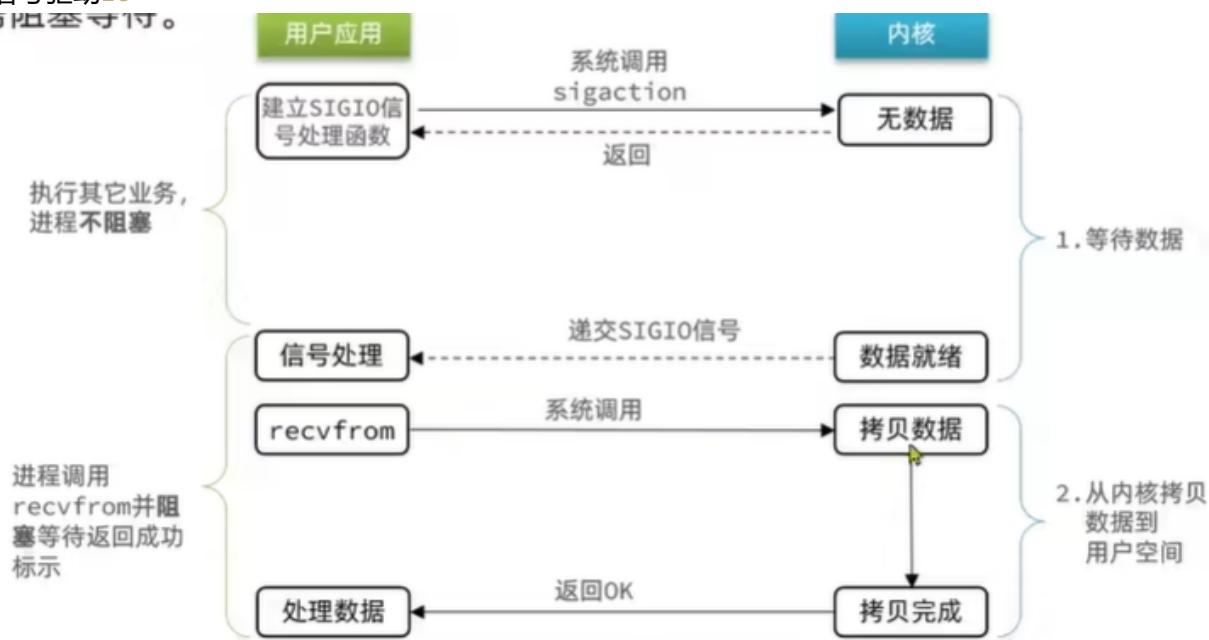


3. **IO多路复用**: Redis使用的是epoll.(Linux中一切皆文件,每个文件都用一个文件描述符fd来访问). IO多路复用是利用单个线程来同时监听多个fd,并在某个fd可读、可写时得到通知,从而避免无效的等待,充分利用CPU资源



#### 4. 信号驱动IO

可阻塞可不阻塞。



#### 5. 异步IO



## Redis的乐观锁、悲观锁

1. 乐观锁是一种并发控制策略，它假设在大多数情况下数据不会被其他线程修改，因此在读取数据时不进行加锁，而是在更新数据时才检查数据是否被其他线程修改过
2. 乐观锁的实现方式:在Redis中,乐观锁可以通过WATCH命令和事务(MULTI/EXEC)来实现
  - 使用WATCH命令监视一个或多个键，记录下它们的当前值
  - 开始一个事务，使用MULTI命令
  - 在事务中执行一系列操作，这些操作基于被监视键的当前值
  - 使用EXEC命令提交事务.如果在事务提交前，被监视的键的值被其他客户端修改过，则事务会被回滚，不执行任何操作.因此只有被监视的键没有被其它客户端(线程)修改才能更新成功
3. 悲观锁是假设数据在大多数情况下都会被其它线程修改,因此在读取数据时就进行加锁,以确保数据的一致性

#### 4. 悲观锁可以利用Redis的SETNX实现:(分布式环境中,悲观锁可以通过分布式锁来实现)

- 尝试获取锁, 使用SETNX命令设置一个键值对, 如果键不存在则设置成功并获取锁, 否则获取锁失败
- 如果获取锁成功, 设置锁的过期时间, 防止死锁
- 执行相关操作
- 操作完成后释放锁, 删除键值对

## LUA脚本

---

1. Redis是单线程执行LUA脚本的,因此保证了lua脚本的原子性

2. Redis使用lua脚本的优点:

- 原子性
- 减少网络开销:将多个命令打包成一个脚本,减少了客户端和服务器之间的往返次数
- 灵活性:lua是一种编程语言,可以在脚本中实现复杂的逻辑