# SDS(simple dynamic string)

## 结构设计

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr32 {

uint32\_t len; /\* used \*/

uint32\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

## O(1)获得字符串长度

* C字符串本身不记录长度信息，每次获取长度信息都需要遍历整个字符串，复杂度为 O(n)；C字符串遍历时遇到'\0'时结束。
* SDS中len字段保存着字符串的长度，所以总能在常数时间内获取字符串长度，复杂度是 O(1)。

## 二进制安全

* 在Redis中不仅可以存储Strin 类型的数据，也可能存储一些二进制数据。
* 二进制数据并不是规则的字符串格式，其中会包含一些特殊的字符如'\0'，在C中遇到'\0'则表示字符串的结束，但在SDS中，标志字符串结束的是 len属性。

## 不会发生缓冲区溢出

* C语言的字符串标准库提供的字符串操作函数，大多数（比如strcat追加字符串函数）都是不安全的，可能发生异常。
* 所以，Redis 的 SDS 结构里引入了alloc（SDS分配给字符串的长度）和len（字符串本身的长度）成员变量，这样 SDS API通过 alloc - len 计算，可以算出剩余可用的空间大小，这样在对字符串做修改操作的时候，就可以由程序内部判断缓冲区大小是否足够用，不够用则重新分配alloc。

## 减少字符串修改时带来的内存重新分配的次数

在C中，当我们频繁的对一个字符串进行修改操作的时候，需要频繁的进行内存重新分配的操作，十分影响性能。Redis实现了空间预分配和惰性空间释放的策略。

#### 空间预分配

当检测到空间不够用时，不仅为字符串分配需要的空间，还会为其分配额外的空间：对sds修改后如果len<1M则分配长度为len的额外空间，如果len>1M则分配长度为1M的额外空间，以此减少内存重新分配次数。

#### 惰性空间释放

当sds进行缩短操作时，不会修改alloc，仅将len缩小。

# 哈希表

哈希表是一种保存键值对（key-value）的数据结构。哈希表优点在于，它能以 O(1)的复杂度快速查询数据。哈希表实际上是数组，哈希表将key通过Hash函数的计算，然后mod哈希表的长度，就能得到数据在表中的下标位置，所以可以通过索引值快速查询到数据。

## 哈希表结构设计

struct dict {

dictType \*type;

dictEntry \*\*ht\_table[2]; //两个哈希表

unsigned long ht\_used[2]; //两个哈希表内数据的数量

//第一个哈希表中的更新下标

long rehashidx; /\* rehashing not in progress if rehashidx == -1 \*/

/\* Keep small vars at end for optimal (minimal) struct padding \*/

int16\_t pauserehash; /\* If >0 rehashing is paused (<0 indicates coding error) \*/

signed char ht\_size\_exp[2]; /\* exponent of size. (size = 1<<exp) \*/

};

在dict结构中，有两个哈希表ht\_table，同时记录了两个哈希表内数据的数量ht\_table和rehash的第一个哈希表更新的下标位置rehashidx，=-1表示未更新，=0表示开始rehash。

## 链式哈希

哈希冲突是指hash(key) % size后落在同一个哈希桶中而产生的冲突，链式哈希实现的方式就是每个哈希表节点都有一个next指针，用于指向下一个哈希表节点，因此多个哈希表节点可以用next指针构成一个单项链表，被分配到同一个哈希桶上的多个节点可以用这个单项链表连接起来，这样就解决了哈希冲突。

## rehash

在dict结构中定义了2个哈希表。

rehash步骤：

* 给「哈希表2」分配空间，一般是「哈希表1」的2 倍；
* 将「哈希表1」的数据迁移到「哈希表2」中；
* 迁移完成后，「哈希表 1」的空间会被释放，并把「哈希表 2」设置为「哈希表1」，然后在「哈希表2」新创建一个空白的哈希表，为下次 rehash 做准备。

如果「哈希表1」的数据量非常大，那么在迁移至「哈希表2」的时候，因为会涉及大量的哈希计算和数据拷贝，会影响redis性能。

## 渐进式 rehash

为解决上述问题，采用渐进式rehash，分多次把数据从「哈希表1」迁移到「哈希表2」。

渐进式rehash步骤：

* 给「哈希表2」分配空间；
* 在 rehash 进行期间每次哈希表元素进行新增、删除操作时，Redis优先在「哈希表2」上操作。每次哈希表元素进行新增、删除、查找或者更新操作时，会触发一次迁移，每次迁移一个不为空的哈希桶内所有数据。

## rehash触发条件

rehash 的触发条件跟负载因子（load factor）有关系。

负载因子=哈希表已保存节点数/哈希表大小

触发条件：

* 当负载因子大于等于1，并且Redis没有在执行某些特殊命令，就会进行 rehash 操作。
* 当负载因子大于等于5时，此时说明哈希冲突非常严重了，不管有没有在执行某些特殊命令，都会强制进行rehash操作。

# 双向链表

## 优点

* listNode 链表节点的结构里带有 prev 和 next 指针，获取某个节点的前置节点或后置节点的时间复杂度只需O(1)；
* list 结构因为提供了表头指针 head 和表尾节点 tail，所以获取链表的表头节点和表尾节点的时间复杂度只需O(1)；
* list 结构因为提供了链表节点数量 len，所以获取链表中的节点数量的时间复杂度只需O(1)；
* listNode 链表节使用 void\* 指针保存节点值，并且可以通过 list 结构的 dup、free、match 函数指针为节点设置该节点类型特定的函数，因此链表节点可以保存各种不同类型的值；

## 缺点

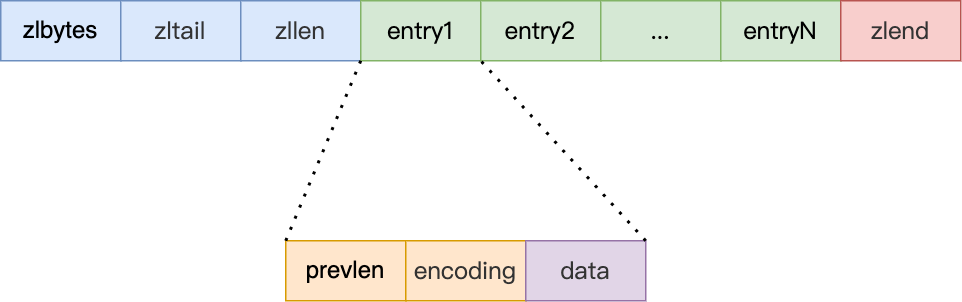
* 链表每个节点之间的内存都是不连续的，意味着无法很好利用 CPU 缓存。能很好利用 CPU 缓存的数据结构就是数组，因为数组的内存是连续的，这样就可以充分利用 CPU 缓存来加速访问。
* 保存一个链表节点的值都需要一个链表节点结构头的分配，内存开销较大。

# 压缩列表

Redis老版本中，对象（List 对象、Hash 对象、Zset 对象）包含的元素数量较少，或者元素值不大的情况才会使用压缩列表作为底层数据结构。

## 结构设计

压缩列表是 Redis 为了节约内存而开发的，它是由连续内存块组成的顺序型数据结构，有点类似于数组。



压缩列表在表头有三个字段：

* zlbytes，记录整个压缩列表占用对内存字节数；
* zltail，记录压缩列表「尾部」节点距离起始地址由多少字节，也就是列表尾的偏移量；
* zllen，记录压缩列表包含的节点数量；
* zlend，标记压缩列表的结束点，固定值 0xFF（十进制255）。

压缩列表节点包含三部分内容：

* prevlen，记录了「前一个节点」的长度；
* encoding，记录了当前节点实际数据的类型以及长度；
* data，记录了当前节点的实际数据；

当我们往压缩列表中插入数据时，压缩列表就会根据数据是字符串还是整数，以及数据的大小，会使用不同空间大小的 prevlen 和 encoding 这两个元素里保存的信息，这种根据数据大小和类型进行不同的空间大小分配的设计思想，正是 Redis 为了节省内存而采用的。

分别说下，prevlen 和 encoding 是如何根据数据的大小和类型来进行不同的空间大小分配。

压缩列表里的每个节点中的 prevlen 属性都记录了「前一个节点的长度」，而且 prevlen 属性的空间大小跟前一个节点长度值有关，比如：

* 如果前一个节点的长度小于 254 字节，那么 prevlen 属性需要用 1 字节的空间来保存这个长度值；
* 如果前一个节点的长度大于等于 254 字节，那么 prevlen 属性需要用 5 字节的空间来保存这个长度值；

encoding 属性的空间大小跟数据是字符串还是整数，以及字符串的长度有关：

* 如果当前节点的数据是整数，则 encoding 会使用 1 字节的空间进行编码。
* 如果当前节点的数据是字符串，根据字符串的长度大小，encoding 会使用 1 字节/2字节/5字节的空间进行编码。

## 为什么需要prevlen字段？

每个结点需要实现向前向后访问，prevlen字段指定了前一个字段的长度，根据压缩列表的连续性和prevlen字段可以直接访问前一个结点，相当于双向链表的prev指针。

## 连锁更新

在压缩列表的结点中，字段prevlen保存了前一个结点的长度，如果<254prevlen占1字节，>=254占5字节，当新增或修改某个元素时，结点长度由<254变成>=254则会影响后一个结点的prevlen，如果后一个结点长度也由<254变成>=254，则又会影响后一个结点的空间分配，以此产生的多米诺效应就是连锁更新。

## 优点

* 压缩列表被设计成一种内存紧凑型的数据结构，占用一块连续的内存空间，可以有效利用 CPU 缓存；
* 针对不同长度的数据，进行相应编码，这种方法可以有效地节省内存开销。

## 缺点

* 不能保存过多的元素，否则查询效率就会降低；
* 新增或修改某个元素时，压缩列表占用的内存空间需要重新分配，甚至可能引发连锁更新的问题。

# 整数集合

## 结构设计

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;

整数集合类似一个有序数组，插入整数时先用二分查找找到插入位置，然后把该位置后面的整数向后移动，最后把数字插入进去（删除类似）。查找使用二分查找。

## 升级操作

整数集合的编码方式有3中，16、32、64位整数。由最大的数字决定使用什么样的编码方式，如果目前使用16位编码，但此时插入一个20位的整数，就需要先将16位编码调整为32位编码再插入，调整的顺序是从后往前。

整数集合升级的好处是节省内存资源。不支持降级操作，一旦对数组进行了升级，就会一直保持升级后的状态。

# 比较压缩列表和整数集合的共同点和区别

## 共同点

采用类似数组的内存紧凑型的数据结构，有效提高cpu缓存，节省内存开销。

## 区别

* 压缩列表不一定是有序存储的，比如使用压缩列表的list，整数集合一定是有序存储的；
* 压缩列表可以存储各种类型的数据，如sds，k-v，整数集合只能存储整数。

# 跳表

Redis只有在Zset对象的底层实现用到了跳表，跳表的优势是能支持平均 O(logN)复杂度的节点查找。Zset是按score进行排序的。

## 结构设计

链表在查找元素的时候，因为需要逐一查找，所以查询效率非常低，时间复杂度是O(N)，于是就出现了跳表。跳表是在链表基础上改进过来的，实现了一种「多层」的有序链表，这样的好处是能快读定位数据。

typedef struct zskiplist {

struct zskiplistNode \*header, \*tail;

unsigned long length;

int level;

} zskiplist;

typedef struct zskiplistNode {

//Zset 对象的元素值

sds ele;

//元素权重值

double score;

//后向指针

struct zskiplistNode \*backward;

//节点的level数组，保存每层上的前向指针和跨度

struct zskiplistLevel {

struct zskiplistNode \*forward;

unsigned long span;

} level[];

} zskiplistNode;

## 查找

跳表是一种多层的有序链表，最底层是普通的双向链表，越往上层节点数量越少。

查找时从最上层开始查找，如果待查找分数等于结点分数，找到并返回；如果带查找分数大于结点分数则回退，然后进入下一层再向前查找；如果带查找分数小于结点分数，则向前查找。

## 实现多层的方法

当插入一个新结点时，有概率p会多一层，如果多出了一层，则又有概率p再多出一层，以此类推。p=0.25，最大层高64。则插入大量数据后相邻两层上层的节点数大约是下层的1/4。

# quicklist

quicklist is a doubly linked list of ziplists. quicklist是一个结点是压缩列表的双向链表。

## 为什么这样设计？

总的来说是一个空间和时间的折中：

* 双向链表易于修改操作，但双向链表的各个节点是单独的内存块，地址不连续，容易产生内存碎片。
* 压缩列表由于是一整块连续内存，所以存储效率很高。但是，它不利于修改操作，每次数据变动都会引发一次内存的重新分配。

压缩列表的大小可以通过list-max-ziplist-size设置，默认是8KB。

## 结构设计

typedef struct quicklist {

//quicklist的链表头

quicklistNode \*head; //quicklist的链表头

//quicklist的链表头

quicklistNode \*tail;

//所有压缩列表中的总元素个数

unsigned long count;

//quicklistNodes的个数

unsigned long len;

...

} quicklist;

typedef struct quicklistNode {

//前一个quicklistNode

struct quicklistNode \*prev; //前一个quicklistNode

//下一个quicklistNode

struct quicklistNode \*next; //后一个quicklistNode

//quicklistNode指向的压缩列表

unsigned char \*zl;

//压缩列表的的字节大小

unsigned int sz;

//压缩列表的元素个数

unsigned int count : 16; //ziplist中的元素个数

....

} quicklistNode;

## 插入

* 当插入位置所在的ziplist大小没有超过限制时，直接插入到ziplist中就好了；
* 当插入位置所在的ziplist大小超过了限制，但插入的位置位于ziplist两端，并且相邻的quicklist链表节点的ziplist大小没有超过限制，那么就转而插入到相邻的那个quicklist链表节点的ziplist中；
* 当插入位置所在的ziplist大小超过了限制，但插入的位置位于ziplist两端，并且相邻的quicklist链表节点的ziplist大小也超过限制，这时需要新创建一个quicklist链表节点插入。
* 对于插入位置所在的ziplist大小超过了限制的其它情况（主要对应于在ziplist中间插入数据的情况），则需要把当前ziplist分裂为两个节点，然后再其中一个节点上插入数据。

# listpack

在redis新版本中listpack替代了ziplist（压缩列表）成为了hash、zset的底层数据结构。ziplist在极小的概率下有可能发生级联更新，当连续规模较大的级联更新发生时，对Redis的性能有比较大的影响。

Listpack和ziplist思想基本一致，主要有3处改动：

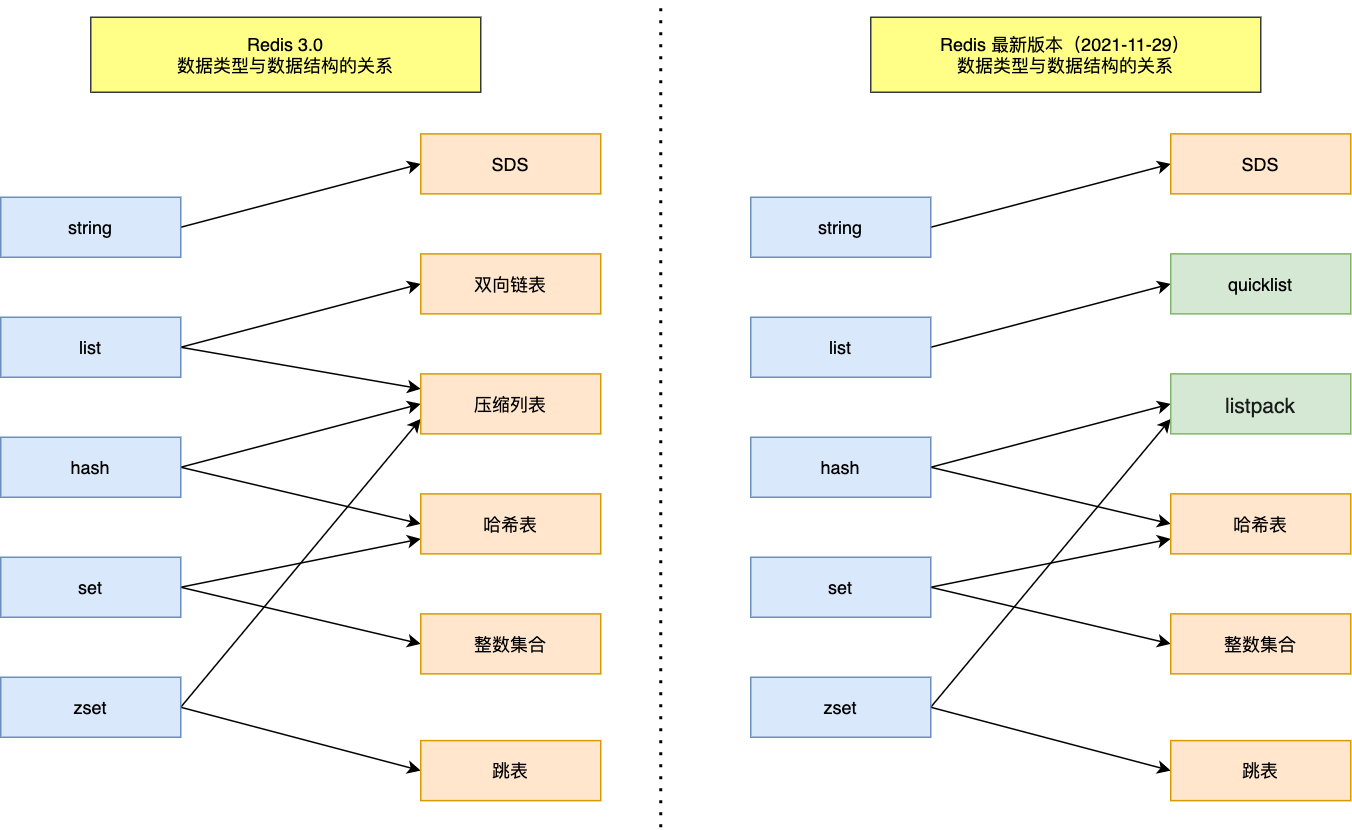
* 去掉了尾部偏移量
* 每个结点只保存自身的长度，不再保存前一个结点的长度
* 把长度字段放到了结点的尾部

细节：长度字段具体使用几个字节由当前字节的最高位决定，最高位为0表示到此为止，为1表示还需要前面一个字节。所以长度字段每个字节的有效位是7位。

## 如何做到向前遍历？

* 尾部偏移量是为了快速找到最后一个结点，在listpack中可以通过listpack的长度减去最后一个字节内最后一个结点的长度得到。
* 每个结点只保存自身的长度自然不会出现连锁更新的情况。
* 访问完一个结点可以访问前一个字节中存储的前一个结点的长度来实现向前遍历。

**以上是redis底层的数据结构，下面是面向用户的数据结构**



zset元素数量少时老版本使用压缩列表，新版本使用listpack；

zset元素数量多时使用字典+跳表。

字典中的键是member，值是score，跳表是按score排序的，score相同可能是按member排序的。

# Zset为什么使用跳表不使用红黑树？

* 在使用zrange、zrangebyscore等命令进行范围查询时，跳表具有更好的性能，跳表只需要查询到满足条件的初始结点，然后向后遍历即可，红黑树则需要进行局部的中序遍历。
* 跳表的插入删除修改操作实现比较简单，红黑树更新后需要保持平衡，跳表的开销比较小。

# AOF(Append Only File)日志

## 什么是AOF日志？

Redis每执行一条写操作命令，就把该命令以追加的方式写入到一个文件里，然后重启Redis的时候，先去读取这个文件里的命令，并且执行它。注意只会记录写操作命令，读操作命令是不会被记录的。

## 先执行写操作命令后，再记录到AOF日志里的优缺点

### 优点

* 避免额外的检查开销，如果先记录到AOF日志的命令有语法错误，需要检测开销，并去除读操作命令。
* 不会阻塞当前写操作命令的执行。

### 缺点

* 执行写操作和记录到AOF日志是两个过程，如果执行完写操作后没来得及记录到AOF日志服务器宕机，会造成数据丢失。
* 可能会阻塞下一个命令的执行。

## 三种写回策略

将AOF日志写入硬盘的过程是先将命令写入内核缓冲区，再由内核缓冲区写入硬盘，三种策略对应3个由内核缓冲区写入硬盘的时机。

### Always

这个单词的意思是「总是」，所以它的意思是每次写操作命令执行完后，同步将 AOF 日志数据写回硬盘；

### Everysec

这个单词的意思是「每秒」，所以它的意思是每次写操作命令执行完后，先将命令写入到 AOF 文件的内核缓冲区，然后每隔一秒将缓冲区里的内容写回到硬盘；

### No

意味着不由 Redis 控制写回硬盘的时机，转交给操作系统控制写回的时机，也就是每次写操作命令执行完后，先将命令写入到 AOF 文件的内核缓冲区，再由操作系统决定何时将缓冲区内容写回硬盘。

Always可靠性最高，性能最低；No性能最高，可靠性最低；Everysec是折中。

## 重写机制

AOF日志是一个文件，随着执行的写操作命令越来越多，文件的大小会越来越大。如果文件过大，整个恢复的过程就会很慢。当AOF文件的大小超过所设定的阈值后，Redis就会启用AOF重写机制，来压缩AOF文件。

### 如何重写？

简单来说就是把多余的写操作去掉，比如多次写入同一个数据则只需要记录最后一次即可。

## 后台重写

重写操作是比较耗时的，所以使用子进程来进行重写，这样可以避免阻塞主进程。

### 为什么不用多线程呢？

子进程和父进程读时共享，写时复制，不需要加锁，而多线程在重写时则需要加锁来保证数据安全，会降低性能。

### 如何解决AOF后台重写造成的数据不一致问题？

设置AOF重写缓冲区，当主进程修改数据时将命令同步写入重写缓冲区，当子进程完成后台重写后，再将重写缓冲区内的命令写入AOF文件，以此保证数据一致性。

# RDB（Redis Database）快照

RBD快照保存的内容是二进制数据，就像是瞬时拍的一张照片，记录着某个瞬间的内存数据。执行快照是一个比较重的操作，如果频率太频繁，可能会对 Redis性能产生影响。如果频率太低，服务器故障时，丢失的数据会更多。

通常可能设置至少5分钟才保存一次快照，这时如果 Redis 出现宕机等情况，则意味着最多可能丢失5分钟数据。

这就是RDB快照的缺点，在服务器发生故障时，丢失的数据会比AOF持久化的方式更多，因为RDB快照是全量快照的方式，因此执行的频率不能太频繁，否则会影响Redis性能，而AOF日志可以以秒级的方式记录操作命令，所以丢失的数据就相对更少。

RDB快照可以选择单线程和子进程的处理方式，单线程会阻塞主线程，子进程不会阻塞主进程。如果选择子进程方式，主进程可以继续修改数据，但是子进程产生的快照只保存修改前的内存数据。

## RDB快照频率

* 如果频率太低，两次快照间一旦服务器发生宕机，就可能会比较多的数据丢失；
* 如果频率太高，频繁写入磁盘和创建子进程会带来额外的性能开销。

## AOF日志和RDB快照的区别

* AOF日志保存的是操作命令，RDB快照保存的是二进制文件。
* AOF日志每次记录只需要添加操作命令，RDB快照每次记录需要记录内存中的全部数据。
* AOF日志的记录频率较高，RDB快照频率较低。
* AOF每次记录的开销小，RDB快照重启数据恢复的速度快。

# 混合使用AOF日志和内存快照

当内存数据量较大时，可以先进行RDB快照，保存到混合文件中，之后用AOF的添加操作命令写入混合文件。这样由于前半部分时RDB快照，数据恢复速度快，采用AOF的添加操作命令的方式花费更小的开销，可以提高保存频率，如果发生宕机丢失的数据也较少。

# 缓存数据库

用户的数据一般都是存储于数据库，数据库的数据是落在磁盘上的，磁盘的读写速度可以说是计算机里最慢的硬件了。

当用户的请求，都访问数据库的话，请求数量一上来，数据库很容易就奔溃的了，所以为了避免用户直接访问数据库，会用 Redis 作为缓存层。

因为 Redis 是内存数据库，我们可以将数据库的数据缓存在Redis里，相当于数据缓存在内存，内存的读写速度比硬盘快好几个数量级，这样大大提高了系统性能。

# 缓存雪崩

通常我们为了保证缓存中的数据与数据库中的数据一致性，会给 Redis 里的数据设置过期时间，当缓存数据过期后，用户访问的数据如果不在缓存里，业务系统需要重新生成缓存，因此就会访问数据库，并将数据更新到Redis里。

那么，当大量缓存数据在同一时间过期（失效）或者Redis故障宕机时，如果此时有大量的用户请求，都无法在 Redis 中处理，于是全部请求都直接访问数据库，从而导致数据库的压力骤增，严重的会造成数据库宕机，从而形成一系列连锁反应，造成整个系统崩溃，这就是缓存雪崩的问题。

## 解决方法

### 大量数据同时过期

#### 均匀设置过期时间

如果要给缓存数据设置过期时间，应该避免将大量的数据设置成同一个过期时间。我们可以在对缓存数据设置过期时间时，给这些数据的过期时间加上一个随机数，这样就保证数据不会在同一时间过期。

#### 互斥锁

当业务线程在处理用户请求时，如果发现访问的数据不在Redis里，就加个互斥锁，保证同一时间内只有一个请求从数据库读取数据，再将数据更新到Redis里。实现互斥锁的时候，最好设置超时时间，防止这个请求出现意外阻塞。

#### 双key策略

我们对缓存数据可以使用两个key，一个是主key，会设置过期时间，一个是备key，不会设置过期，它们只是key不一样，但是value值是一样的，相当于给缓存数据做了个副本。

当业务线程访问不到「主key」的缓存数据时，就直接返回「备key」的缓存数据，然后在更新缓存的时候，同时更新「主key」和「备key」的数据。

#### 后台更新缓存

更新缓存的任务交给后台线程，不再设置过期时间，当系统内存紧张时淘汰部分缓存数据，定期检查缓存是否有效，失效则更新缓存。

### Redis故障宕机

#### 服务熔断或请求限流机制

可以启动服务熔断机制，暂停业务应用对缓存服务的访问，直接返回错误，不用再继续访问数据库，从而降低对数据库的访问压力，保证数据库系统的正常运行，然后等到Redis恢复正常后，再允许业务应用访问缓存服务。但这会导致全部业务无法正常工作。

为了减少对业务的影响可以启用请求限流机制，只将少部分请求发送到数据库进行处理，再多的请求就在入口直接拒绝服务，等到 Redis恢复正常并把缓存预热完后，再解除请求限流的机制。

#### 构建Redis缓存高可靠集群

可以通过主从节点的方式构建 Redis 缓存高可靠集群。

如果Redis缓存的主节点故障宕机，从节点可以切换成为主节点，继续提供缓存服务，避免了由于Redis故障宕机而导致的缓存雪崩问题。

# 缓存击穿

业务通常会有几个数据会被频繁地访问，比如秒杀活动，这类被频地访问的数据被称为热点数据。

如果缓存中的某个热点数据过期了，此时大量的请求访问了该热点数据，就无法从缓存中读取，直接访问数据库，数据库很容易就被高并发的请求冲垮，这就是缓存击穿的问题。缓存击穿是缓存雪崩的一个子集。

## 解决方案

* 互斥锁方案，保证同一时间只有一个业务线程读取数据库更新缓存，未能获取互斥锁的请求，要么等待锁释放后重新读取缓存，要么就返回空值或者默认值。
* 不给热点数据设置过期时间，由后台异步更新缓存，或者在热点数据准备要过期前，提前通知后台线程更新缓存以及重新设置过期时间。

# 缓存穿透

用户访问的数据，既不在缓存中，也不在数据库中，导致请求在访问缓存时，发现缓存缺失，再去访问数据库时，发现数据库中也没有要访问的数据，没办法构建缓存数据，来服务后续的请求。那么当有大量这样的请求到来时，数据库的压力骤增，这就是缓存穿透的问题。

## 缓存穿透的发生的情况

* 业务误操作，缓存中的数据和数据库中的数据都被误删除了，所以导致缓存和数据库中都没有数据。
* 黑客恶意攻击，故意大量访问某些读取不存在数据的业务。

## 解决方案

### 限制非法请求

当有大量恶意请求访问不存在的数据的时候，也会发生缓存穿透，因此我们要判断求请求参数是否合理，请求参数是否含有非法值、请求字段是否存在，如果判断出是恶意请求就直接返回错误，避免进一步访问缓存和数据库。

### 缓存空值或者默认值

当遇到访问缓存和数据库均不存在的数据时，将空值或默认值存入缓存，那么其他请求此数据时会直接命中缓存中的空值或默认值。

### 使用boolean过滤器快速判断数据是否存在

我们可以在写入数据库数据时，使用布隆过滤器做个标记，然后在用户请求到来时，业务线程确认缓存失效后，可以通过查询布隆过滤器快速判断数据是否存在，如果不存在，就不用通过查询数据库来判断数据是否存在。

即使发生了缓存穿透，大量请求只会查询Redis和布隆过滤器，而不会查询数据库，保证了数据库能正常运行，Redis自身也是支持布隆过滤器的。

#### boolean过滤器流程

布隆过滤器由初始值都为0的数组和N个哈希函数两部分组成。当我们在写入数据库数据时，在布隆过滤器里做个标记，这样下次查询数据是否在数据库时，只需要查询布隆过滤器，如果查询到数据没有被标记，说明不在数据库中。

布隆过滤器会通过3个操作完成标记：

* 使用N个哈希函数分别对数据做哈希计算，得到N个哈希值。
* 将第一步得到的N个哈希值对位图数组的长度取模，得到每个哈希值在位图数组的对应位置。
* 将每个哈希值在位图数组的对应位置的值设置为1。

当应用要查询数据x是否数据库时，通过布隆过滤器只要查到位图数组的N个位置的值是否全为1，只要有一个为0，就认为数据x不在数据库中。

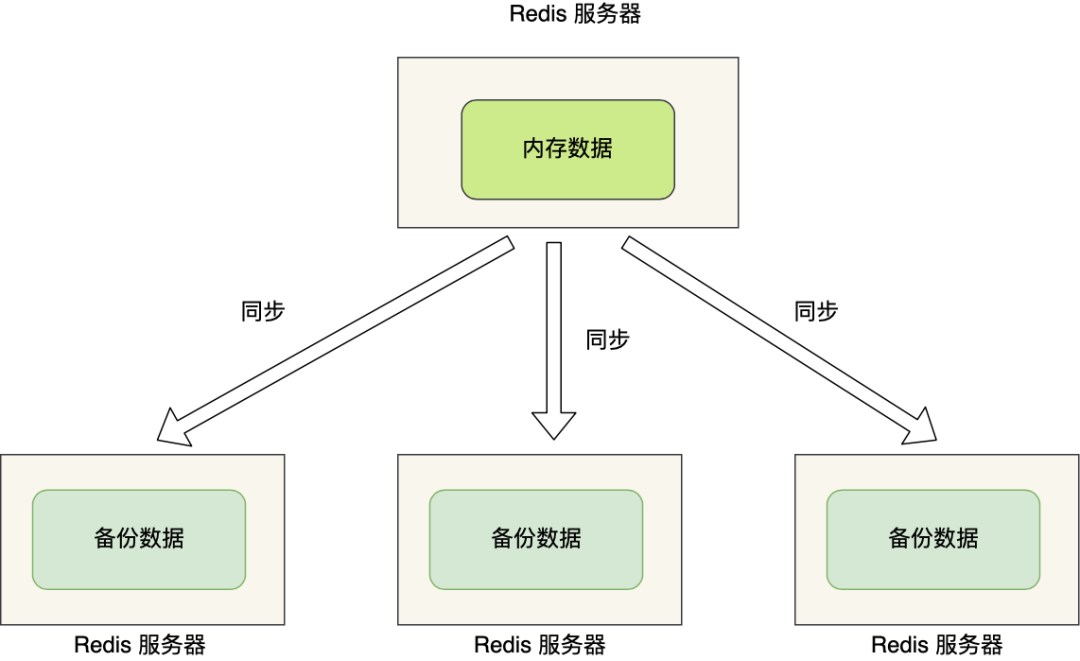
布隆过滤器由于是基于哈希函数实现查找的，高效查找的同时存在哈希冲突的可能性。所以，查询布隆过滤器说数据存在，并不一定证明数据库中存在这个数据，但是查询到数据不存在，数据库中一定就不存在这个数据。

# Redis缓存高可靠集群（主从复制）

由于数据都是存储在一台服务器上，如果出现意外如：

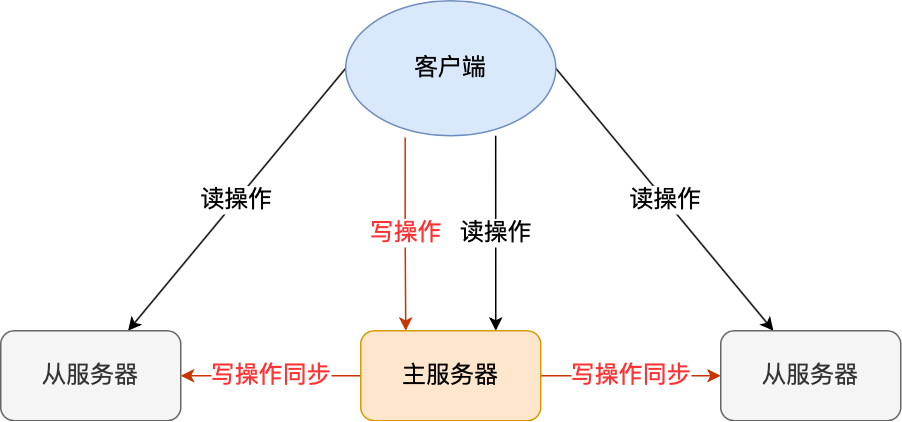
* 如果服务器发生了宕机，由于数据恢复是需要点时间，那么这个期间是无法服务新的请求的；
* 如果这台服务器的硬盘出现了故障，可能数据就都丢失了。

要避免这种单点故障，最好的办法是建立redis缓存高可靠集群，将数据备份到其他服务器上，让这些服务器也可以对外提供服务，这样即使有一台服务器出现了故障，其他服务器依然可以继续提供服务。



Redis提供了主从复制，读写分离的模式，来保证数据一致性。

主服务器可以进行读写操作，从服务器只能进行读操作，也就是说写操作只在主服务器上进行，然后将最新的数据同步给从服务器。



## 第一次同步

从服务器可以使用replicaof命令形成主从关系。

# 服务器 B 执行这条命令

replicaof <服务器 A 的 IP 地址> <服务器 A 的 Redis 端口号>

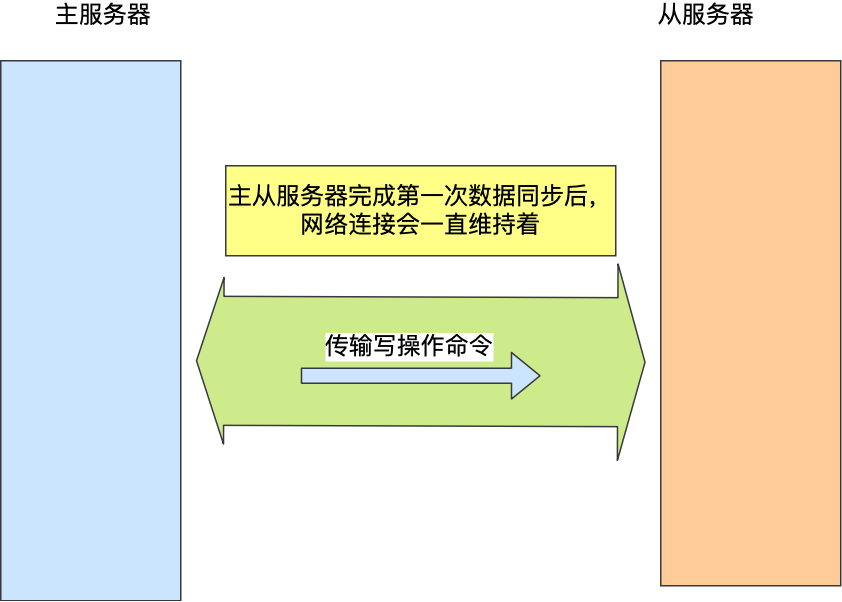
服务器B就成为服务器A的从服务器，接下来进行第一次同步。

### 第一次同步的过程可分为三个阶段

* 第一阶段是建立链接、协商同步（replicaof）；
* 第二阶段是主服务器同步数据给从服务器（传输RDB文件，全量复制）；
* 第三阶段是主服务器发送新写操作命令给从服务器（执行第二阶段时客户端进行的写操作）。

## 命令传播

主从服务器在完成第一次同步后，双方之间就会维护一个TCP长连接。



后续主服务器可以通过这个连接继续将写操作命令传播给从服务器，然后从服务器执行该命令，使得与主服务器的数据库状态相同。

而且这个连接是长连接的，目的是避免频繁的TCP连接和断开带来的性能开销。

上面的这个过程被称为基于长连接的命令传播，通过这种方式来保证第一次同步后的主从服务器的数据一致性。

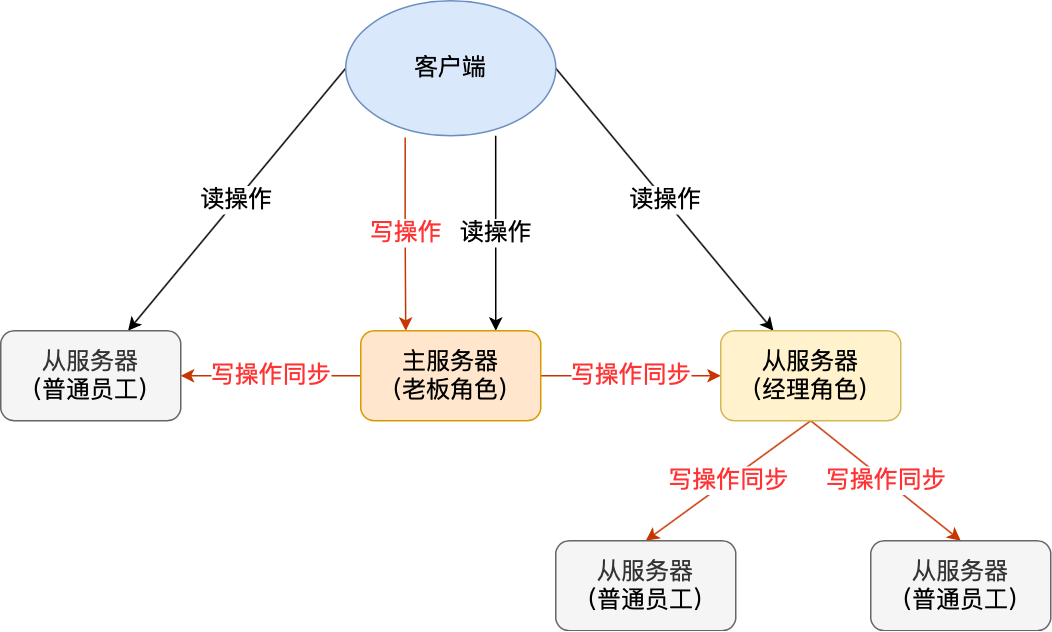
## 分摊主服务器的压力

主从服务器在第一次数据同步的过程中，主服务器会做两件耗时的操作：生成 RDB 文件和传输 RDB 文件。

主服务器是可以有多个从服务器的，如果从服务器数量非常多，而且都与主服务器进行全量同步的话，就会带来两个问题：

* 主服务器后台生成RDB文件，忙于使用fork()创建子进程，如果主服务器的内存数据非大，在执行fork()函数时是会阻塞主线程的，从而使得 Redis 无法正常处理请求；
* 传输 RDB 文件会占用主服务器的网络带宽，会对主服务器响应命令请求产生影响。

从服务器可以有自己的从服务器，我们可以把拥有从服务器的从服务器当作经理角色，它不仅可以接收主服务器的同步数据，自己也可以同时作为主服务器的形式将数据同步给从服务器，这样可以分摊主服务器的压力。



## 增量复制

主从服务器在完成第一次同步后，就会基于长连接进行命令传播。

可是，网络总是不按套路出牌的嘛，说延迟就延迟，说断开就断开。

如果主从服务器间的网络连接断开了，那么就无法进行命令传播了，这时从服务器的数据就没办法和主服务器保持一致了，客户端就可能从「从服务器」读到旧的数据。

如果此时断开的网络，又恢复正常了，要怎么继续保证主从服务器的数据一致性呢？

网络断开又恢复后，从主从服务器会采用增量复制的方式继续同步，也就是只会把网络断开期间主服务器接收到的写操作命令，同步给从服务器。

主服务器怎么知道要将哪些增量数据发送给从服务器呢？

使用一个环形的缓冲区和一个同步标记，在重新连接上后可以找到同步数据的差异，再把网络断开期间主服务器收到的写操作命令同步给从服务器。

# 哨兵（Sentinel）

## 为什么需要哨兵？

在Redis的主从架构中，由于主从模式是读写分离的，如果主节点挂了，那么将没有主节点来服务客户端的写操作请求，也没有主节点给从节点进行数据同步了。哨兵可以监测主节点是否存活，如果发现主节点挂了，它就会选举一个从节点切换为主节点，并且把新主节点的相关信息通知给从节点和客户端。

## 哨兵是如何工作的？

哨兵其实是一个运行在特殊模式下的Redis进程，所以它也是一个节点。从“哨兵”这个名字也可以看得出来，它相当于是“观察者节点”，观察的对象是主从节点。哨兵节点主要负责三件事情：监控、选主、通知。

## 哨兵如何判断主节点真的故障了？

哨兵会周期性地给所有主从节点发送PING命令，当主从节点收到PING命令后，会发送一个响应命令给哨兵，这样就可以判断它们是否在正常运行。如果主节点或者从节点没有在规定的时间内响应哨兵的PING命令，哨兵就会将它们标记为主观下线。主观下线的意思是这个哨兵认为该节点下线了。

相对应的客观下线只适用于主节点。

之所以针对「主节点」设计「主观下线」和「客观下线」两个状态，是因为有可能「主节点」其实并没有故障，可能只是因为主节点的系统压力比较大或者网络发送了拥塞，导致主节点没有在规定时间内响应哨兵的PING命令。

所以，为了减少误判的情况，哨兵在部署的时候不会只部署一个节点，而是用多个节点部署成哨兵集群（最少需要三台机器来部署哨兵集群），通过多个哨兵节点一起判断，就可以就可以避免单个哨兵因为自身网络状况不好，而误判主节点下线的情况。同时，多个哨兵的网络同时不稳定的概率较小，由它们一起做决策，误判率也能降低。

当一个哨兵判断主节点为「主观下线」后，就会向其他哨兵发起命令，其他哨兵收到这个命令后，就会根据自身和主节点的网络状况，做出赞成投票或者拒绝投票的响应当赞成票达到一定数值（quorum），则判断主节点客观下线。客观下线表示大多数哨兵都认为主节点下线。

## 如何选新主节点？

1. 过滤掉已经离线的从节点。
2. 过滤掉历史网络连接状态不好的从节点。
3. 依次进行3论考察。

* 优先级最高的从节点胜出，优先级可以用户设置，一般硬件配置高的服务器优先级更高。
* 复制进度最靠前的从节点胜出。
* ID号小的从节点胜出。

## 由哪个哨兵进行主从切换？

哪个哨兵节点判断主节点为「客观下线」，这个哨兵节点就是候选者，所谓的候选者就是想当Leader的哨兵，然后由各个哨兵进行投票，赞成票达到一定数值时成为Leader，负责主从切换。

## 如何通知客户端新主节点的信息？

通过Redis的发布者/订阅者机制来实现的。每个哨兵节点提供发布者/订阅者机制，客户端可以从哨兵订阅消息。

比如，客户端订阅了主从切换的事件，当哨兵把新主节点选择出来后，就会发布新主节点的 IP 地址和端口信息，这个时候客户端就可以收到这条信息，然后用这里面的新主节点的 IP 地址和端口进行通信了。

SUBSCRIBE 频道（订阅频道）

PUBLISH 频道 消息（在频道发布消息）

## 哨兵集群是如何组成的？

哨兵节点之间是通过 Redis 的发布者/订阅者机制来相互发现的。每个哨兵都订阅一个相同的频道，然后把自己的ip和端口发布在该频道上，其他哨兵就能与其建立TCP连接。

主节点知道所有「从节点」的信息，所以哨兵会向主节点发送INFO命令来获取所有「从节点」的信息。至此，哨兵建立了集群并获得了主从节点的信息。

# redis缓存和数据库一致性

## 全量数据刷入缓存

这是最简单直接的方案，启动一个定时任务，将数据库的数据定时全部刷入缓存且不设置过期时间，写请求不修改缓存。

### 优点

所有读请求都命中缓存，性能非常高。

### 缺点

* 缓存利用率低，不经常访问的数据也会留在缓存中。
* 数据不一致，因为是定时刷新，刷新的时间间隔数据库数据也可能被修改。

### 适用场景

* 业务体量小
* 对数据一致性要求不高

## 如何提高缓存利用率？

想要缓存利用率「最大化」，我们很容易想到的方案是，缓存中只保留最近访问的「热数据」。可以这样优化：

* 写请求依旧只写数据库。
* 读请求先读缓存，如果缓存不存在，则从数据库读取，并重建缓存。
* 同时，写入缓存中的数据，都设置失效时间。

这样一来，缓存中不经常访问的数据，随着时间的推移，都会逐渐「过期」淘汰掉，最终缓存中保留的，都是经常被访问的「热数据」，缓存利用率得以最大化。

## 如何保证数据一致性？

当数据发生更新时，我们不仅要操作数据库，还要一并操作缓存。有以下4中方案：

* 先更新数据库，再更新缓存。
* 先更新缓存，再更新数据库。
* 先更新数据库，再删除缓存。
* 先删除缓存，再更新数据库。

比较上述方案需要考虑2种情况，分别是：

* 第一步操作成功，第二步操作失败。
* 并发情况。

先更新数据库，再更新缓存和先更新缓存，再更新数据库从缓存利用率的角度出发都是不太推荐的，因为每次数据发生变更，都「无脑」更新缓存，但是缓存中的数据不一定会被「马上读取」，这就会导致缓存中可能存放了很多不常访问的数据，浪费缓存资源。因此接下来考虑删除缓存的情况。

### 先删除缓存再更新数据库

#### 删除缓存成功，更新数据库失败

用户查询缓存未命中，会到数据库读取数据，再将数据写入缓存，并不会出现数据不一致的情况。

#### 并发情况

数据库和缓存中X=1

线程A删除缓存X

线程B读取X，缓存未命中，读取数据库得到X=1

线程A更新数据库X=2

线程B将X=1写入缓存

此时缓存X=1是旧值，数据库X=2是新值，数据不一致。

##### 解决方案

延迟双删，在线程 A 删除缓存、更新完数据库之后，先「休眠一会」，再「删除」一次缓存。目的是等待别的线程在这段时间将旧值写入缓存后再次删除。

### 先更新数据库再删除缓存

#### 更新数据库成功 删除缓存失败

如果数据库更新成功了，但缓存删除失败，那么此时数据库中是最新值，缓存中是「旧值」。之后的读请求读到的都是旧数据。这时用户会发现，自己刚刚修改了数据，但却看不到变更，一段时间过后，数据才变更过来，对业务也会有影响。

##### **解决方案**

有两种解决方案，第一种是消息队列重试，第二种是订阅数据库变更日志再操作缓存。

1. 消息队列重试

我们可以引入消息队列，将第二个操作（删除缓存）要操作的数据加入到消息队列，由消费者来操作数据。

* 如果应用删除缓存失败，可以从消息队列中重新读取数据，然后再次删除缓存，这个就是重试机制。当然，如果重试超过的一定次数，还是没有成功，我们就需要向业务层发送报错信息了。
* 如果删除缓存成功，就要把数据从消息队列中移除，避免重复操作，否则就继续重试。

1. 订阅数据库变更日志再操作缓存

我们的业务应用在修改数据时，「只需」修改数据库，无需操作缓存。拿MySQL举例，当一条数据发生修改时，MySQL就会产生一条变更日志（Binlog），我们可以订阅这个日志，拿到具体操作的数据，然后再根据这条数据，去删除对应的缓存。

PS: 这两种方法有一个共同的特点，都是采用异步操作缓存。

#### 并发情况

数据库X=1缓存中没有X

1. 线程A读取X，缓存未命中，读取数据库X=1
2. 线程B更新数据库X=2
3. 线程B删除缓存
4. 线程A写缓存X=1

此时缓存X=1是旧值，数据库X=2是新值，数据不一致。

虽然理论上是可能发生的，但实际上更新数据库需要加锁，一般比都数据库要慢，更新数据库+删除缓存比写缓存快的可能性很低，所以实际上先更新数据库再删除缓存在并发情况下是可以保证数据一致的。

# Redis面试问题

## redis主从节点时长连接还是短链接？

长连接。

## 怎么判断redis某个节点是否正常工作？

Redis判断接点是否正常工作，基本都是通过互相的心跳检测机制，如果有一半以上的节点去ping一个节点的时候没有pong回应，集群就会认为这个节点挂掉了，会断开与这个节点的连接。

redis主从节点发送的心态间隔是不一样的，而且作用也有一点区别：

* Redis主节点ping从节点的时间间隔较长，判断从节点的存活性和连接状态。
* redis从节点ping主节点时间间隔较短，这样可以实时监测主从节点网络状态，上报自身复制偏移量，检查复制数据是否丢失，如果丢失再从主节点复制缓冲区以增量复制的形式拉取丢失数据。

## 主从复制架构中，过期key如何处理？

主节点处理了一个key或者通过淘汰算法淘汰了一个key，这个时间主节点模拟一条del命令发送给从节点，从节点收到该命令后，就进行删除key的操作。

## redis 是同步复制还是异步复制？

redis主节点每次收到写命令之后，先写到内部的缓冲区，然后异步发送给从节点。

## redis 主从切换如何减少数据丢失？

### 异步复制同步丢失

对于 redis 主节点与从节点之间的数据复制，是时异步复制的，当客户端发送写请求给主节点的时候，客户端会返回ok，接着主节点将写请求异步同步给各个从节点，但是如果此时主节点还没来得及同步给从节点时发生了断电，那么主节点内存中的数据会丢失。

可以有 2 种解决方案：

* 客户端将数据暂时写入本地缓存和磁盘中，在一段时间后将本地缓存或者磁盘的数据发送给主节点，来保证数据不丢失；
* 客户端将数据写入到消息队列中，发送一个延时消费消息，比如10分钟后再消费消息队列中的数据，然后再写到主节点。

### 集群产生脑裂数据丢失

#### 什么是脑裂？

由于网络问题，集群节点之间失去联系。主从数据不同步；哨兵重新选举，产生两个主服务。等网络恢复，旧主节点会降级为从节点，再与新主节点进行同步复制的时候，由于会从节点会清空自己的缓冲区，所以导致之前客户端写入的数据丢失。

#### 解决方案

当主节点发现从节点下线或者通信超时的总数量小于阈值时，那么禁止主节点进行写数据，直接把错误返回给客户端。

## redis 主从如何做到故障自动切换？

使用redis哨兵机制，哨兵在发现主节点出现故障时，由哨兵选举新的主节点。

## Redis是单线程还是多线程？

* Redis6.0之前的单线程指的是网络IO和键值对读写在服务端是由一个线程完成的。
* Redis6.0之后引入的多线程是指网络请求使用了多线程，但键值对的读写还是单线程，所以Redis还是并发安全的。
* Redis只有网络请求和数据操作是单线程的，其持久化、集群数据同步都是由后台进程执行的。

## Redis单线程为什么还能这么快？

* 命令执行基于内存操作。
* 命令执行基于单线程，没有线程切换的开销。
* 网络请求基于多路I/O复用（epoll）。
* 高效的数据存储结构：哈希表、跳表、整数集合、quicklist、listpack。

## Redis有可能会出现Key过期但内存还没有释放的情况吗？

有可能，Redis删除过期Key有两种策略：

* 惰性删除，当用户再次读写时检查Key是否过期，如果过期则删除，这种情况下如果一个过期的Key不再被读写就不会被删除。
* 定时删除，每隔一个固定的时间（100ms）删除一批过期的Key，注意不是全部的Key，因为如果一段时间过期的Key非常多，全部删除对服务器性能影响太大。

这两种策略都有可能出现Key过期但内存还没有释放。

## Redis可能会清理没设置过期时间的Key吗？

有可能，当Redis当前内存使用超出限制时，有3种清理策略，清理没设置过期时间的Key可能发生在第2种策略，清理所有的Key。

### 只清理过期Key

* 根据过期的时间先后顺序清理。
* 随机清理。
* LRU（Least Recently Used）清理，淘汰最久未被使用的过期的Key，以最后一次访问时间作为参考。
* LFU（Least Frequently Used）清理，淘汰使用频次最少的过期的Key，以最近一段时间访问次数作为参考。

Ps：大多数情况都可以使用LRU，在存在大量热点数据时使用LFU更好。

### 清理所有Key

* 随机清理。
* LRU清理。
* LFU清理。

### 不清理，拒绝写入

## 删除Key的命令会阻塞Redis吗？

Redis的命令执行是单线程的，所以是有可能阻塞Redis的，因为DEL命令的时间复杂度是O(M)，M是要删除的Key结构内元素的大小，比如hash结构。

## Redis主从、哨兵、集群架构优缺点比较

### 主从模式

Redis服务器由一个主服务器和多个从服务器组成，主服务器负责读写数据，从服务器负责读数据，每当主服务器写入数据时，都将数据同步给从服务器，这种架构在主服务器出现异常时，需要人工选取一个从服务器成为新的主服务器。

### 哨兵模式

哨兵模式就是在主从模式的基础上引入哨兵集群，由哨兵集群监听主从节点，如果哨兵判断主节点挂了，则选举一个新的从节点成为新的主节点。哨兵模式下的主节点内存不宜过大，否则会使持久化文件过大，影响数据恢复和同步效率。哨兵模式的主节点只有一个，并发性能有限。哨兵进行主从切换的时候，会出现瞬断（客户端访问不到请求数据）。

### 高可靠集群模式

Redis集群是由主从节点群组成的分布式服务器群，它具有复制、高可用和分片特性。分片的意思是总数据100G，集群中可以把100G分散为5个20G保存在5个主从节点中。集群模式中负责主从节点切换的是所有主节点，相当于哨兵。

集群模式的并发性能和可用性都高于哨兵模式。

## Redis集群数据hash分片算法

在集群中有多个主节点，如何判断key在哪个主节点中呢？Redis Cluster将所有数据划分为16384个槽位，每个节点负责一部分槽位，插入或查询Key时对Key作hash运算再mod16384得到对应的槽位，这样就能找到是哪个主节点负责这个Key。

## Redis死循环Bug

在Redis5.0之前，在从节点中执行RANDOMKEY操作时有可能出现死循环。

### 在主节点中执行RANDOMKEY

Redis采用惰性删除和定时删除的策略，内存中留有过期的Key，随机返回一个Key首先在内存中随机一个Key，然后检查它有没有过期，如果没有过期直接返回，如果过期了由于惰性删除策略将它删除，再随机一个Key直到找到一个没有过期的Key，这样理论上时不会死循环的。

### 在从节点中执行RANDOMKEY

在从节点中，是不能主动删除数据的，必须等主节点删除了过期的Key并发送给从节点一个DEL命令从节点才能删除。所以在执行RANDOMKEY时，即使发现Key过期也不能删除它，只能继续挑选Key，如果所有的Key都过期了，此时就会出现死循环。

在5.0后Redis修复了这个BUG，解决方法是设置随机选取次数的上限（100）。