redis

数据类型

string

Redis 的字符串是动态字符串，是可以修改的字符串，内部结构实现上类似于 Java 的ArrayList，采用预分配冗余空间的方式来减少内存的频繁分配，内部为当前字 符串实际分配的空间 capacity 一般要高于实际字符串长度len。当字符串长度小于1M时，扩容都是加倍现有的空间，如果超过1M，为了避免加倍后的冗余空间过大而导致浪费，所以扩容时一次只会多扩1M的空间。需要注意的是 字符串最大长度为 512M。

list

Redis的列表相当于 Java 语言里面的 LinkedList，注意它是链表而不是数组。这意味着 list 的插入和删除操作非常快，时间复杂度为 O(1)，但是索引定位很慢，时间复杂度为 O(n) 。当列表弹出了最后一个元素之后，该数据结构自动被删除，内存被回收。

Redis 的列表结构常用来做异步队列使用。将需要延后处理的任务结构体序列化成字符串塞进Redis 的列表，另一个线程从这个列表中轮询数据进行处理。

Redis底层存储的还不是一个简单的linkedlist，而是称之为快速链表quicklist的一个结构。首先在列表元素较少的情况下会使用一块连续的内存存储，这个结构是ziplist，也即是 压缩列表。它将所有的元素紧挨着一起存储，分配的是一块连续的内存。当数据量比较多的时候才会改成quicklist。因为普通的链表需要的附加指针空间太大，会比较浪费空间，而且 会加重内存的碎片化。比如这个列表里存的只是int类型的数据，结构上还需要两个额外的指针prev和next。所以Redis将链表和ziplist结合起来组成了quicklist。也就是将多个 ziplist 使用双向指针串起来使用。这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。

hash

Redis 的字典相当于 Java 语言里面的 HashMap，它是无序字典。内部实现结构上同 Java 的 HashMap 也是一致的，同样的数组 + 链表二维结构。第一维 hash 的数组位置碰撞 时，就会将碰撞的元素使用链表串接起来。不同的是，Redis 的字典的值只能是字符串，另外它们 rehash 的方式不一样，因为 Java 的 HashMap 在字典很大时，rehash 是个耗时的操作，需要一次性全部 rehash。Redis 为了高性能，不能堵塞服务，所以采用了渐进式 rehash 策略。

渐进式 rehash 会在 rehash 的同时，保留新旧两个 hash 结构，查询时会同时查询两个 hash 结构，然后在后续的定时任务中以及 hash 的子指令中，循序渐进地将旧 hash 的内容 一点点迁移到新的 hash 结构中。当 hash 移除了最后一个元素之后，该数据结构自动被删除，内存被回收。

set

Redis 的集合相当于 Java 语言里面的 HashSet，它内部的键值对是无序的唯一的。它的 内部实现相当于一个特殊的字典，字典中所有的 value 都是一个值 NULL。

当集合中最后一个元素移除之后，数据结构自动删除，内存被回收。 set 结构可以用来 存储活动中奖的用户 ID，因为有去重功能，可以保证同一个用户不会中奖两次。

zset

Redis的有序集合它类似于 Java 的 SortedSet 和 HashMap 的结合体，一方面它是一个 set，保证了内部 value 的唯一性，另一方面它可以给每个 value 赋予一个 score，代表这个 value 的排序权 重。它的内部实现用的是一种叫着「跳跃列表」的数据结构。zset 中最后一个 value 被移除后，数据结构自动删除，内存被回收。

通用规则

list/set/hash/zset 这四种数据结构是容器型数据结构，它们共享下面两条通用规则：

1、create if not exists

2、drop if no elements

数据结构

sds

Redis 的字符串叫 sds，也就是 Simple Dynamic String。它的结构是一个带长度信息的字节数组。

```
struct SDS<T> {
    T capacity; // 数组容量
    T len; // 数组长度
    byte flags; // 特殊标识位，不理睬它
    byte[] content; // 数组内容
}
```

content里面存储了真正的字符串的内容，capacity是分配的数组长度，len是字符串实际的长度

Redis 的字符串有两种存储方式，在长度特别短时，使用 emb 形式存储 (embeded)，当 长度超过 44 时，使用 raw 形式存储。embstr 存储形式是这样一种存储形式，它将 RedisObject 对象头和 SDS对象连续存在一起，使用malloc方法一次分配。而raw存储形式不一样，它需要两次malloc，两个对象头在内存地址上一般是不连续的。而内存分配器 jemalloc/tcmalloc 等分配内存大小的单位都是 2、4、8、16、32、64 等 等，为了能容纳一个完整的 embstr 对象，jemalloc 最少会分配 32 字节的空间，如果字符 串再稍微长一点，那就是 64 字节的空间。如果总体超出了 64 字节，Redis 认为它是一个 大字符串，不再使用 emdstr 形式存储，而该用 raw 形式。

44=64-16-3-1

64字节是规定embeded最大的占用内存，16是redis对象头的长度，3是sds头的长度，1是最后的'\0'。

dict

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

dict 结构内部包含两个 hashtable，通常情况下只有一个 hashtable 是有值的。但是在dict 扩容缩容时，需要分配新的 hashtable，然后进行渐进式搬迁，这时候两个 hashtable 存储的分别是旧的 hashtable 和新的 hashtable。待搬迁结束后，旧的 hashtable 被删除，新的hashtable 取而代之。

hashtable 的结构和 Java 的HashMap 几乎是一样的，都是通过分桶的方式解决 hash 冲突。第一维是数组，第二维是链表。数组中存储的是第二维链表的第一个元素的指针。

扩容

正常情况下，当 hash 表中元素的个数等于第一维数组的长度时，就会开始扩容，扩容 的新数组是原数组大小的 2 倍。但是当 Redis 正在做 bgsave，为了减少内存页的过多分离 (Copy On Write)，Redis 尽量不去扩容 (dict_can_resize)，但是如果 hash 表已经非常满 了，元素的个数已经达到了一维数组长度的 5 倍 (dict_force_resize_ratio)，说明 hash 表 已经过于拥挤了，这个时候就会强制扩容。

缩容

当 hash 表因为元素的逐渐删除变得越来越稀疏时，Redis 会对 hash 表进行缩容来减少。hash 表的第一维数组空间占用。缩容的条件是元素个数低于数组长度的 10%。缩容不会考 虑 Redis 是否正在做 bgsave。

ziplist

Redis 为了节约内存空间使用，zset 和 hash 容器对象在元素个数较少的时候，采用压缩列表进行存储。压缩列表是一块连续的内存空间，元素之间紧挨着存储，没有任何冗余空隙。

因为 ziplist 都是紧凑存储，没有冗余空间 。意味着插 入一个新的元素就需要调用 realloc 扩展内存。取决于内存分配器算法和当前的 ziplist 内存 大小，realloc 可能会重新分配新的内存空间，并将之前的内容一次性拷贝到新的地址，也可 能在原有的地址上进行扩展，这时就不需要进行旧内容的内存拷贝。

如果 ziplist 占据内存太大，重新分配内存和拷贝内存就会有很大的消耗。所以 ziplist 不适合存储大型字符串，存储的元素也不宜过多。

quicklist

在Redis早期的时候存储 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，也就是元素少时用 ziplist，元素多时用 linkedlist。但是链表的附加成本相对比较高，prev与next指针就需要16字节，而且每个节点的内存都是单独分配，会让内存产生碎片化，从而影响内存管理的效率，在后面就加入了quicklist。

它相当于ziplist与linkedlist的结合体，将linkedlist按段划分，然后每一段用ziplist紧凑存储，多个ziplist采用双向指针连接

而且ziplist是可以深度压缩

quicklist 默认的压缩深度是 0，也就是不压缩。压缩的实际深度由配置参数 list- compress-depth决定。为了支持快速的 push/pop 操作，quicklist 的首尾两个 ziplist 不压 缩，此时深度就是 1。如果深度为 2，就表示 quicklist 的首尾第一个 ziplist 以及首尾第二 个 ziplist 都不压缩。

skiplist

Redis 的 zset 是一个复合结构，一方面它需要一个 hash 结构来存储 value 和 score 的 对应关系，另一方面需要提供按照 score 来排序的功能，还需要能够指定 score 的范围来获 取 value 列表的功能，这就需要另外一个结构跳跃列表。

Redis 的跳跃表共有 64 层，意味着最多可以容纳 2^64 次方个元素。每一个 kv 块对应的结构如下面的代码中的 zslnode 结构，kvheader 也是这个结构，只不过 value 字段是 null 值—无效的，score 是Double.MIN_VALUE，用来垫底的。kv 之间使用指针串起来形成了双向链表结构，它们是有序 排列的，从小到大。不同的 kv 层高可能不一样，层数越高的 kv 越少。同一层的 kv会使用指针串起来。每一个层元素的遍历都是从 kv header 出发。

```
struct zslnode {
    string value;
    double score;
    zslnode*[] forwards; //多层连接指针
    zslnode* backward; //回溯指针
}
    struct zsl {
    zslnode* header; //跳跃列表头指针
    int maxLevel; //跳跃列表当前的最高层
    map<string, zslnode*> ht; // hash 结构的所有键值对
}
```

查找：我们要定位到那个某个 kv，需要从 header 的最高层开始遍历找到第一个 节点 (最后一个比「我」小的元素)，然后从这个节点开始降一层再遍历找到第二个节点 (最 后一个比「我」小的元素)，然后一直降到最底层进行遍历就找到了期望的节点 (最底层的最 后一个比我「小」的元素)。

我们将中间经过的一系列节点称之为「搜索路径」，它是从最高层一直到最底层的每一层最后一个比「我」小的元素节点列表。有了这个搜索路径，我们就可以插入这个新节点了。不过这个插入过程也不是特别简单。因为新插入的节点到底有多少层，得有个算法来分配一下，跳跃列表使用的是随机算法。

插入：对于每一个新插入的节点，都需要调用一个随机算法给它分配一个合理的层数。直观上期望的目标是 50% 的 Level1，25% 的 Level2，12.5% 的 Level3，一直到最顶层 2^-63，因为这里每一层的晋升概率是 50%。

首先我们在搜索合适插入点的过程中将「搜索路径」摸出来了，然后就可以开始创建新节点了，创建的时候需要给这个节点随机分配一个层数，再将搜索路径上的节点和这个新节点通过前向后向指针串起来。如果分配的新节点的高度高于当前跳跃列表的最大高度，就需要更新一下跳跃列表的最大高度。

删除：删除过程和插入过程类似，都需先把这个「搜索路径」找出来。然后对于每个层的相关 节点都重排一下前向后向指针就可以了。同时还要注意更新一下最高层数 maxLevel。

更新：当我们调用 zadd 方法时，如果对应的 value 不存在，那就是插入过程。如果这个value 已经存在了，只是调整一下 score的值，那就需要走一个更新的流程。假设这个新的score值不会带来排序位置上的改变，那么就不需要调整位置，直接修改元素的score值就可以了。但是如果排序位置改变了，那就要调整位置。一个简单的策略就是先删除这个元素，再插入这个元素，需要经过两次路径搜索。Redis就是这么干的。 不过 Redis 遇到 score 值改变了就直接删除再插入，不会去判断位置是否需要调整

如果 score 值都一样呢?

在一个极端的情况下，zset 中所有的 score 值都是一样的，zset 的查找性能会退化为 O(n) 么?Redis 作者自然考虑到了这一点，所以 zset 的排序元素不只看 score 值，如果 score 值相同还需要再比较 value 值 (字符串比较)。

Rax

Rax 是 Redis 内部比较特殊的一个数据结构，它是一个有序字典树 (基数树 Radix Tree)，按照 key 的字典序排列，支持快速地定位、插入和删除操作。Redis 五大基础数据结 构里面，能作为字典使用的有 hash 和 zset。hash 不具备排序功能，zset 则是按照 score 进 行排序的。rax 跟 zset 的不同在于它是按照 key 进行排序的。

你也可以将公安局的人员档案信息看成一棵 radix tree，它的 key 是每个人的身份证 号，value 是这个人的履历。因为身份证号的编码的前缀是按照地区进行一级一级划分的， 这点和单词非常类似。有了这棵树，你就可以快速地定位出人员档案，还可以快速查询出某个小片区都有哪些人。

Rax 被用在 Redis Stream 结构里面用于存储消息队列，在 Stream 里面消息 ID 的前缀 是时间戳 + 序号，这样的消息可以理解为时间序列消息。使用Rax结构进行存储就可以快 速地根据消息 ID 定位到具体的消息，然后继续遍历指定消息之后的所有消息。

Rax 是一棵比较特殊的 radix tree，它在结构上不是标准的 radix tree。如果一个中间节点有多个子节点，那么路由键就只是一个字符。如果只有一个子节点，那么路由键就是一个字 符串。后者就是所谓的「压缩」形式，多个字符压在一起的字符串。

位图

概念

redis的数据结构之一，是一个byte数组，会自动扩容，当偏移量超过现有的内容范围，就会自动将数组进行零扩充()。如果要存hello的话，就需要将这个字符串的ASCII码找到，然后再不停地去存。位数组的存储于字符的的位顺序是相反的。setbit s 1 1 setbit s 2 1 setbit s 5 1 后面的1就是为1，前面的数字是第几位，如果需要获取的话就只需要 get s 就可以了

特性

零存整取、零存零取、整存零取。零存：就是用setbit挨个设置，整存就是使用字符串去填充所有的位数组，零取就是获取这个对象的第n位是0/1,整取就是获取这个对象的整体。

节省空间：因为是位图，每次存储0或者1，在一些特定情况下可以减少很多存储空间

为布隆过滤器打基础。

HyperLogLog

HyperLogLog 提供不精确的去重计数方案，虽然不精确但是也不是非常不精确，标准误差是 0.81%，这样的精确度已经可以满足上面的UV统计需求了。提供了两个指令 pfadd 和 pfcount，根据字面意义很好理解，一个是增加 计数，一个是获取计数。pfadd 用法和 set 集合的 sadd 是一样的，来一个用户 ID，就将用 户 ID 塞进去就是。pfcount 和 scard 用法是一样的，直接获取计数值。HyperLogLog 除了前面说的 pfadd 和 pfcount 之外，还提供了第三个指令 pfmerge，用于 将多个 pf 计数值累加在一起形成一个新的 pf 值。HyperLogLog 这个数据结构不是免费的，不是说使用这个数据结构要花钱，它需要占据 一定 12k 的存储空间，所以它不适合存储单个用户的信息，适合存储上千万上亿的用户信息。

它的存储空间采用稀疏矩阵存储，空间占用很小，仅仅在计数慢慢变大，稀疏矩阵占 用空间渐渐超过了阈值时才会一次性转变成稠密矩阵，才会占用 12k 的空间。

布隆过滤器：

在Redis4之后出现的插件，为RedisServer服务。

如果需要自定义参数的布隆过滤器，就得要在bf.add之前显示调用bf.reserve，它有三个参数key，error_rate 和 initial_size。错误率越低，需要的空间越大。initial_size 参数表示预计放 入的元素数量，当实际数量超出这个数值时，误判率会上升。

所以需要提前设置一个较大的数值避免超出导致误判率升高。如果不使用 bf.reserve，默 认的 error_rate 是 0.01，默认的 initial_size 是 100。

布隆过滤器的 initial_size 估计的过大，会浪费存储空间，估计的过小，就会影响准确 率，用户在使用之前一定要尽可能地精确估计好元素数量，还需要加上一定的冗余空间以避 免实际元素可能会意外高出估计值很多。

布隆过滤器的 error_rate 越小，需要的存储空间就越大，对于不需要过于精确的场合， error_rate 设置稍大一点也无伤大雅。比如在新闻去重上而言，误判率高一点只会让小部分文 章不能让合适的人看到，文章的整体阅读量不会因为这点误判率就带来巨大的改变。

当我们在往里面插入key的时候，尽量不要让实际元素数>初始化的数，如果超过了初始化的数，就得要对这个布隆过滤器进行重载，然后分配一个更大的内存。

原理：

其实就是多个hash函数，当我们向布隆过滤器添加一个元素的时候，会用多个hash函数对key算出一个整数的索引，然后再将相应的位置变为1，每一个hash函数会算出不同的位置。所以当我们在判断一个key在不在的时候，就会通过对这个key多个hash时的索引，看这几个索引在的位置是不是都为1，如果都为1就说明存在，如果没有1说明不存在。所以对布隆过滤器而言是有一定的误差的，但是当我们在判断某个元素不存在的时候，这个key肯定不存在，但是在判断存在的时候，就有可能出现问题，因为可能会有其他key的hash函数将这个位置变为1，虽然概率比较小，但是还是存在这种情况下的。所以在判断存在的时候就只是极可能存在而不是一定存在

错误率计算公式：

```
k=0.7*(l/n) # 约等于
f=0.6185^(l/n)
```

特点：

1、位数组相对越长 (l/n)，错误率 f 越低，这个和直观上理解是一致的
2、位数组相对越长 (l/n)，hash 函数需要的最佳数量也越多，影响计算效率
3、当一个元素平均需要 1 个字节 (8bit) 的指纹空间时 (l/n=8)，错误率大约为 2%
4、错误率为 10%，一个元素需要的平均指纹空间为 4.792 个 bit，大约为 5bit
5、错误率为 1%，一个元素需要的平均指纹空间为 9.585 个 bit，大约为 10bit
6、错误率为 0.1%，一个元素需要的平均指纹空间为 14.377 个 bit，大约为 15bit

用处：

布隆过滤器可以显著降低数据库的 IO 请求数量。当用户来查询某个 row 时，可以先通过内存中的布隆过滤器过滤掉大量不存在的 row 请求，然后再去磁盘进行查询。

在爬虫的时候，我们就得需要对URL进行去重，已经爬过的页面那就不需要再爬了，如果用set去装的话就会非常浪费空间，如果使用布隆过滤器的话可以大幅降低去重存储消耗，只不过也会使得爬虫系统 错过少量的页面。

邮箱系统的垃圾邮件过滤功能也普遍用到了布隆过滤器，因为用了这个过滤器，所以平时也会遇到某些正常的邮件被放进了垃圾邮件目录中，这个就是误判所致，概率很低。

持久化

rdb

RDB其实也就是我们常说的快照，备份的是全量数据，以二进制文件的形式存放在内存中，存储的非常紧凑。因为redis是一个单线程的程序，这个线程同时负责多个客户端的并发读写操作和内存结构的逻辑读写，在快照过程中，Redis必须进行文件IO操作，但是文件IO操作不能用多路复用，所以就会导致文件IO操作会大幅度的拖累服务器的性能。所以，redis采用的是COW机制来作为快照持久化。在持久化的过程中，并不是这个主线程去进行持久化操作而是先fork出一个子进程，这个子进程才是处理快照操作的进程，主进程还是继续接收客户端的请求。这个时候子进程在做持久化操作，所以不会修改内存结构，但是主进程还在接收客户端的请求，这时候肯定会去修改内存结构。所以就有了COW的用武之处，COW会将某一个瞬间的数据快照，生成某个时间点的数据快照，然后去操作这个快照。

aof

连续的增量备份，与MySQL的binlog比较类似。AOF记录的是数据修改的指令记录文本，随着时间的增长，这个文本会变得越来越大。为此，redis有了AOF重写的功能，将一些没有用的命令给删掉，只保留最后一次有效修改的命令，bgrewriteaof命令。

重启 Redis 时，我们很少使用 rdb 来恢复内存状态，因为会丢失大量数据。我们通常 使用 AOF 日志重放，但是重放 AOF 日志性能相对 rdb 来说要慢很多，这样在 Redis 实 例很大的情况下，启动需要花费很长的时间。Redis 4.0 为了解决这个问题，带来了一个新的持久化选项—混合持久化。将 rdb 文 件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是自 持久化开始到持久化结束的这段时间发生的增量 AOF 日志，通常这部分 AOF 日志很小。于是在 Redis 重启的时候，可以先加载 rdb 的内容，然后再重放增量 AOF 日志就可 以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到 升。

setnx

我们可以用redis的setnx去实现一个分布式锁，如果仅仅用setnx的话是有很大的问题的。如果一个线程获取了这个锁之后，在执行过程中还没来得及del的时候，程序出现了异常，就会陷入到一种死锁状态。为了防止这种情况的出现，可以加一个expire，这样有了过期时间后，就会减少一部分问题。但是还会有一些问题，就比如说setnx之后expir之前的这个时间段出现异常也会导致死锁状态。所以为了防止这种情况出现，可以把nx与ex同时作为set的参数，这样就可以达到一种原子性的状态。

不支持可重入锁，所以如果分布式要使用可重入的话就得在逻辑层上修改，redis并不支持。所以，我们应该避免用redis的setnx做可重入锁的逻辑。

scan

当我们需要从海量的key中获取其中具有某些特性的key时，很多人第一个想法就是keys,举个栗子，我想要获取前缀是common的key时，很多人就会选择用这条命令去寻找符合自己预期的key。 'keys common*' 但是，在我们的key非常多的时候，或者符合条件的key太多的时候，你就知道这个命令会造成多大的破坏。因为keys命令是遍历所有的key，所以说它的时间复杂度是O(n),如果我们的数据集非常大，就会导致redis服务卡顿，所有redis的读写命令都会被延后甚至超时，因为redis是单线程的程序，它需要顺序的去执行所有指令，当keys命令这里不停地运行，就会导致其他命令发生阻塞，必须等待keys命令执行完才可以执行。
为了解决这种问题，redis引入了scan命令。
scan命令拥有很多优点：
1、通过游标分步进行，不会阻塞线程
2、提供limit参数，可以控制返回条数，不像keys会返回所有的条数
当然也会有比较多的缺点：
1、返回的结果可能有重复，需要客户端去重
2、遍历的过程中如果修改了数据，不能保证这个数据会被便利到
3、单次返回的结果是null，并不意味着遍历结束了，应该看游标的返回值。因为limit并不是说返回的结果条数，而是一次遍历的槽值，

缓存穿透

概念：
用户的请求后，在数据库里没有，那么自然redis里也没有。这样就会直接穿过redis，到达DB，这就是缓存穿透。

解决方案：
缓存空值
如果一个数据查询时空的时候，就先将这个空值去缓存到redis中，然后第二次的时候就会有值了。而不会继续访问数据库
布隆过滤器
和缓存空值比较类似，不过它是将空值放到布隆过滤器中，第二次查询的时候，先去布隆过滤器查，如果在布隆过滤器里面有的话就会直接返回空，如果没有再去查。
优点：底层是bitmap,占据的空间比较少 。
性能比较高

缓存雪崩

概念

如果缓存在一段时间内大范围的过期，而新的缓存还没有更新出来，就会导致发生大范围的缓存穿透。所有的查询压力全部放在了数据库上，对数据库的CPU和内存造成了很大压力，甚至宕机这就是缓存雪崩

解决方案

1）加锁排队

用setnx去加锁，如果成功，则把这条数据放在数据中，再load进缓存，如果失败则进行一次get操作

2）双重缓存

设置两个缓存，C1与C2，C1的过期时间可以设置的相对比较短，C2的过期时间可以设置的相对比较长。我们遇到请求可以先去查询C1，如果C1查询不到可以去查询C2。如果C2也查询不到再去数据库查询，这样减少了DB的压力，也可以预防缓存雪崩

3）数据预热

系统上线之前，先将相关的数据加载到缓存系统中。通过缓存reload机制，预先去更新缓存，在发生大的并发访问之前手动出发加载不同的key。

4）定时更新缓存

时效性要求不高的缓存，可以在容器启动时进行初始化加载，采用定时任务来更新或者删除缓存。

5）随机时间

给缓存的失效时间加一个随机值，尽量让缓存的失效时间均匀一些。