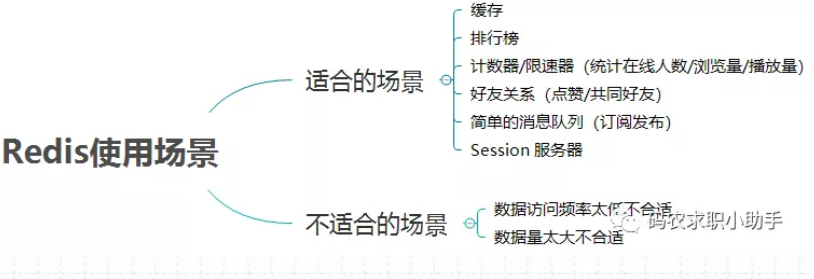
**Redis面试题总结**

**1、谈下你对 Redis 的了解？**

Redis（全称：Remote Dictionary Server 远程字典服务）是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。

**2、Redis 一般都有哪些使用场景？**



**Redis 适合的场景**

1. 缓存：减轻 MySQL 的查询压力，提升系统性能；2. 排行榜：利用 Redis 的 SortSet （有序集合）实现；  
3. 计算器/限速器：利用 Redis 中原子性的自增操作，我们可以统计类似用户点赞数、用 户访问数等。这类操作如果用 MySQL，频繁的读写会带来相当大的压力；限速器比较典 型的使用场景是限制某个用户访问某个 API 的频率，常用的有抢购时，防止用户疯狂 点击带来不必要的压力；  
4. 好友关系：利用集合的一些命令，比如求交集、并集、差集等。可以方便解决一些共同 好友、共同爱好之类的功能；5. 消息队列：除了 Redis 自身的发布/订阅模式，我们 也可以利用 List 来实现一个队列机制，比如：到货通知、邮件发送之类的需求，不需 要高可靠，但是会带来非常大的 DB 压力，完全可以用 List 来完成异步解耦；6. Session 共享：Session 是保存在服务器的文件中，如果是集群服务，同一个用户过来 可能落在不同机器上，这就会导致用户频繁登陆；采用 Redis 保存 Session 后，无论 用户落在那台机器上都能够获取到对应的 Session 信息。

**Redis 不适合的场景**

数据量太大、数据访问频率非常低的业务都不适合使用 Redis，数据太大会增加成本，访问频率太低，保存在内存中纯属浪费资源。

1. **Redis 有哪些常见的功能？**
2. 数据缓存功能

2. 分布式锁的功能

3. 支持数据持久化

4. 支持事务

5. 支持消息队列

**4、Redis 支持的数据类型有哪些？**

**1. string 字符串**

字符串类型是 Redis 最基础的数据结构，首先键是字符串类型，而且其他几种结构都是在字符串类型基础上构建的。字符串类型实际上可以是字符串：简单的字符串、XML、JSON；数字：整数、浮点数；二进制：图片、音频、视频。使用场景：缓存、计数器、共享 Session、限速。

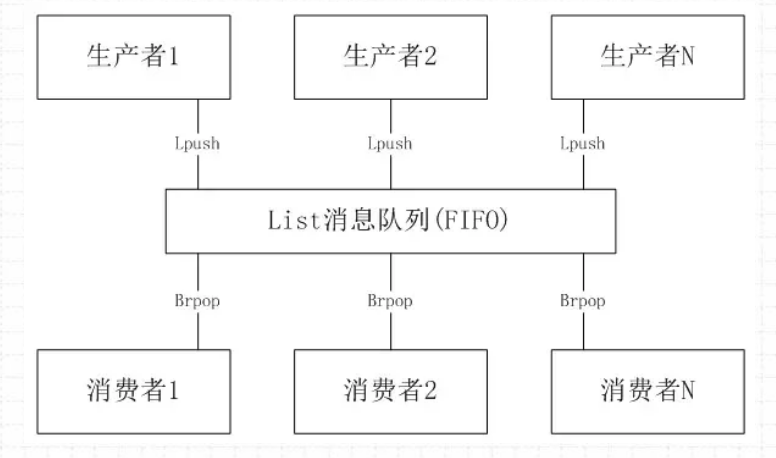
**2. Hash（哈希）**

在 Redis中哈希类型是指键本身是一种键值对结构，如 value={{field1,value1},......{fieldN,valueN}}

使用场景：哈希结构相对于字符串序列化缓存信息更加直观，并且在更新操作上更加便捷。所以常常用于用户信息等管理，但是哈希类型和关系型数据库有所不同，哈希类型是稀疏的，而关系型数据库是完全结构化的，关系型数据库可以做复杂的关系查询，而 Redis 去模拟关系型复杂查询开发困难且维护成本高。

**3. List（列表）**

列表类型是用来储存多个有序的字符串，列表中的每个字符串成为元素，一个列表最多可以储存 2 ^ 32 - 1 个元素，在 Redis 中，可以队列表两端插入和弹出，还可以获取指定范围的元素列表、获取指定索引下的元素等，列表是一种比较灵活的数据结构，它可以充当栈和队列的角色。使用场景：Redis 的 lpush + brpop 命令组合即可实现阻塞队列，生产者客户端是用 lpush 从列表左侧插入元素，多个消费者客户端使用 brpop 命令阻塞式的“抢”列表尾部的元素，多个客户端保证了消费的负载均衡和高可用性。



**4. Set（集合）**

集合类型也是用来保存多个字符串的元素，但和列表不同的是集合中不允许有重复的元素，并且集合中的元素是无序的，不能通过索引下标获取元素，Redis 除了支持集合内的增删改查，同时还支持多个集合取交集、并集、差集。合理的使用好集合类型，能在实际开发中解决很多实际问题。使用场景：如：一个用户对娱乐、体育比较感兴趣，另一个可能对新闻感兴趣，这些兴趣就是标签，有了这些数据就可以得到同一标签的人，以及用户的共同爱好的标签，这些数据对于用户体验以及曾强用户粘度比较重要。

**5. zset（sorted set：有序集合）**

有序集合和集合有着必然的联系，它保留了集合不能有重复成员的特性，但不同得是，有序集合中的元素是可以排序的，但是它和列表的使用索引下标作为排序依据不同的是：它给每个元素设置一个分数，作为排序的依据。使用场景：排行榜是有序集合经典的使用场景。例如：视频网站需要对用户上传的文件做排行榜，榜单维护可能是多方面：按照时间、按照播放量、按照获得的赞数等。

**5、Redis 为什么是单线程的？**

官方 FAQ 表示，因为 Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且 CPU 不会成为瓶颈，那就顺理成章地采用单线程的方案了，毕竟采用多线程会有很多麻烦。

**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；

1. 数据结构简单，对数据操作也简单；

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。

**7、什么是缓存穿透？怎么解决？**

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

**解决办法：**

1. 缓存空对象：如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

**缓存空对象带来的问题：**

1. 空值做了缓存，意味着缓存中存了更多的键，需要更多的内存空间，比较有效的方 法是针对这类数据设置一个较短的过期时间，让其自动剔除。

2. 缓存和存储的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如： 过期时间设置为 5分钟，如果此时存储添加了这个数据，那此段时间就会出现缓 存和存储数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空 对象。

2. 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。

**8、什么是缓存雪崩？该如何解决？**  
 如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

**解决办法：**

1. 加锁排队：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。 比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待；

2. 数据预热：可以通过缓存 reload 机制，预先去更新缓存，再即将发生大并发访问 前手动触发加载缓存不同的 key，设置不同的过期时间，让缓存失效的时间点尽量 均匀；

3. 做二级缓存，或者双缓存策略：Cache1 为原始缓存，Cache2 为拷贝缓存，Cache1 失 效时，可以访问 Cache2，Cache1 缓存失效时间设置为短期，Cache2 设置为长期。

4. 在缓存的时候给过期时间加上一个随机值，这样就会大幅度的减少缓存在同一时间 过期。

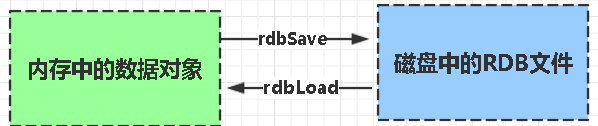
**9、 怎么保证缓存和数据库数据的一致性？**  
 1. 从理论上说，只要我们设置了合理的键的过期时间，我们就能保证缓存和数据库的 数据最终是一致的。因为只要缓存数据过期了，就会被删除。随后读的时候，因为 缓存里没有，就可以查数据库的数据，然后将数据库查出来的数据写入到缓存中。 除了设置过期时间，我们还需要做更多的措施来尽量避免数据库与缓存处于不一致 的情况发生。

2. 新增、更改、删除数据库操作时同步更新 Redis，可以使用事务机制来保证数据的 一致性。

**10、Redis 持久化有几种方式？**

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。Redis 提供了两种持久化方式：RDB（默认） 和 AOF。

**RDB：**

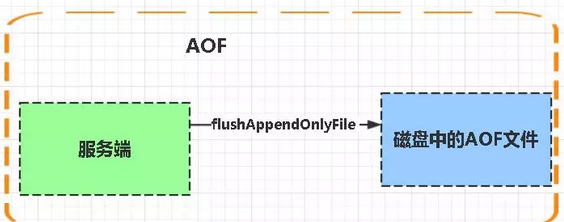
RDB 是 Redis DataBase 的缩写。按照一定的时间周期策略把内存的数据以快照的形式保存到硬盘的二进制文件。即 Snapshot 快照存储，对应产生的数据文件为 dump.rdb，通过配置文件中的 save 参数来定义快照的周期。核心函数：rdbSave（生成 RDB 文件）和 rdbLoad（从文件加载内存）两个函数。

**AOF：**

AOF 是 Append-only file 的缩写。Redis会将每一个收到的写命令都通过 Write 函数追加到文件最后，类似于 MySQL 的 binlog。当 Redis 重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。每当执行服务器（定时）任务或者函数时，flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作：

WRITE：根据条件，将 aof\_buf 中的缓存写入到 AOF 文件；

SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。



**RDB 和 AOF 的区别：**

1. AOF 文件比 RDB 更新频率高，优先使用 AOF 还原数据；

2. AOF比 RDB 更安全也更大；

3. RDB 性能比 AOF 好；

4. 如果两个都配了优先加载 AOF。

**11、Redis 怎么实现分布式锁？**

Redis 为单线程模式，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。一般使用 setnx(set if not exists) 指令，只允许被一个程序占有，使用完调用 del 释放锁。

**12、Redis 淘汰策略有哪些？**

1. volatile-lru：从已设置过期时间的数据集（server. db[i]. expires）中挑选最近最 少使用的数据淘汰；

2. volatile-random：从已设置过期时间的数据集（server. db[i]. expires）中任意选择 数据淘汰。

3. allkeys-lru：从数据集（server. db[i]. dict）中挑选最近最少使用的数据淘汰。

4. allkeys-random：从数据集（server. db[i]. dict）中任意选择数据淘汰。

5. volatile-ttl：从已设置过期时间的数据集（server. db[i]. expires）中挑选将要过 期的数据淘汰。

6、no-enviction（驱逐）：禁止驱逐数据。

**13、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，如 RDB 内存快照和 AOF 日志文件。如果数据比较 重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一次；

2. 为了主从复制的速度和连接的稳定性， Master 和 Slave 最好在同一个局域网内；

3. 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3…

**14、Redis的原子性如何保证？**

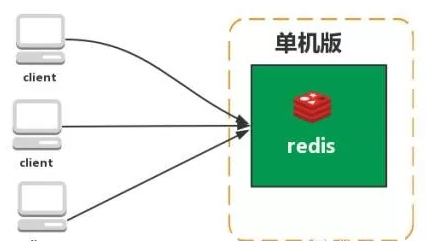
对于Redis而言，命令的原子性指的是：一个操作不可再分，操作要么执行要么不执行，Redis之所以是原子的，是因为Redis是单线程的。

Redis所有单个命令的执行都是原子的，Redis是实现事务的原理：

1. 批量操作在发送EXEC（执行）命令前被放入队列缓存；
2. 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令都不会被 执行；  
    3.  在事务执行过程，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。

**15.Redis 有哪些架构模式？讲讲各自的特点**

**单机版**

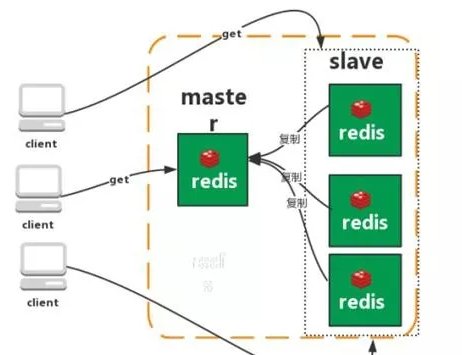


特点：简单

问题：

1、内存容量有限 2、处理能力有限 3、无法高可用。

**主从复制**

****

Redis 的复制（replication）功能允许用户根据一个 Redis 服务器来创建任意多个该服务器的复制品，其中被复制的服务器为主服务器（master），而通过复制创建出来的服务器复制品则为从服务器（slave）。 只要主从服务器之间的网络连接正常，主从服务器两者会具有相同的数据，主服务器就会一直将发生在自己身上的数据更新同步 给从服务器，从而一直保证主从服务器的数据相同。

特点：

1、master/slave 角色

2、master/slave 数据相同

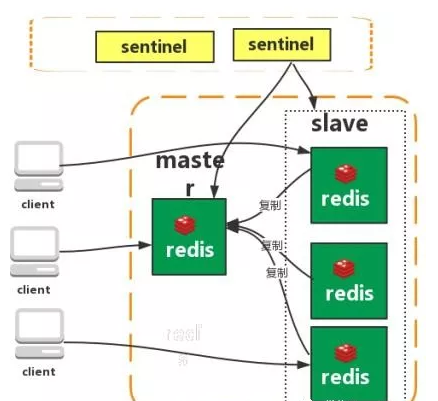
3、降低 master 读压力在转交从库

问题：

无法保证高可用

没有解决 master 写的压力

**哨兵**

****

Redis sentinel 是一个分布式系统中监控 redis 主从服务器，并在主服务器下线时自动进行故障转移。其中三个特性：

监控（Monitoring）：    Sentinel  会不断地检查你的主服务器和从服务器是否运作正常。

提醒（Notification）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。

自动故障迁移（Automatic failover）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作。

特点：

1、保证高可用

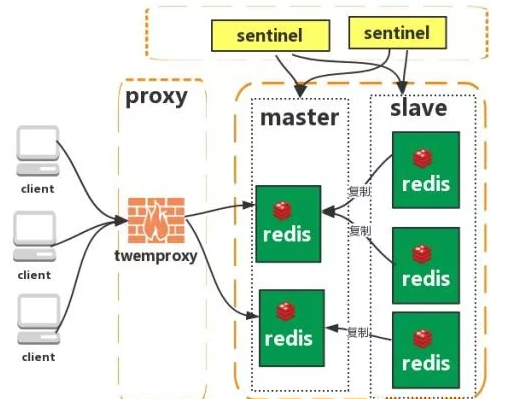
2、监控各个节点

3、自动故障迁移

缺点：主从模式，切换需要时间丢数据

没有解决 master 写的压力

**集群（proxy 型）：**

****

Twemproxy 是一个 Twitter 开源的一个 redis 和 memcache 快速/轻量级代理服务器； Twemproxy 是一个快速的单线程代理程序，支持 Memcached ASCII 协议和 redis 协议。

特点：1、多种 hash 算法：MD5、CRC16、CRC32、CRC32a、hsieh、murmur、Jenkins

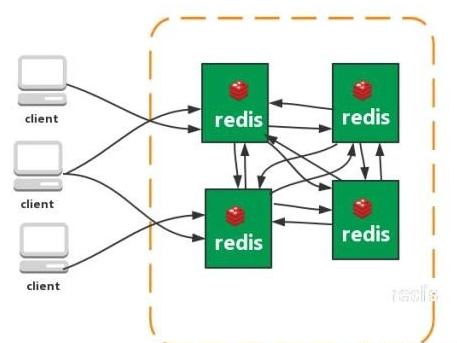
2、支持失败节点自动删除

3、后端 Sharding 分片逻辑对业务透明，业务方的读写方式和操作单个 Redis 一致

缺点：增加了新的 proxy，需要维护其高可用。

failover 逻辑需要自己实现，其本身不能支持故障的自动转移可扩展性差，进行扩缩容都需要手动干预

**集群（直连型）：**

****

从redis 3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。

特点：

1、无中心架构（不存在哪个节点影响性能瓶颈），少了 proxy 层。

2、数据按照 slot 存储分布在多个节点，节点间数据共享，可动态调整数据分布。

3、可扩展性，可线性扩展到 1000 个节点，节点可动态添加或删除。

4、高可用性，部分节点不可用时，集群仍可用。通过增加 Slave 做备份数据副本

5、实现故障自动 failover，节点之间通过 gossip 协议交换状态信息，用投票机制完成 Slave到 Master 的角色提升。

缺点：

1、资源隔离性较差，容易出现相互影响的情况。

2、数据通过异步复制,不保证数据的强一致性

**16.使用过Redis分布式锁么，它是怎么实现的？**

先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

**如果在setnx之后执行expire之前进程意外crash或者要重启维护了，那会怎么样？**

set指令有非常复杂的参数，这个应该是可以同时把setnx和expire合成一条指令来用的！

**17.使用过Redis做异步队列么，你是怎么用的？有什么缺点？**

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

缺点：

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

**18.能不能生产一次消费多次呢？**

使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

进程间的一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

graphic

**案例：**

先订阅后发布后才能收到消息，

1 可以一次性订阅多个，SUBSCRIBE c1 c2 c3

2 消息发布，PUBLISH c2 hello-redis

===============================================================================

3 订阅多个，通配符\*， PSUBSCRIBE new\*

4 收取消息， PUBLISH new1 redis2015

**19、Redis相比memcached有哪些优势？**  
(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

**20、Redis如何存储复杂的数据？**

# **复杂数据存储**

一般的数据存储，就是一个key对应一个简单字符串，可以要想向mysql一样，保存或者获取某一列的所有值呢，  
例如,用户表结构为

| **id** | **name** | **age** |
| --- | --- | --- |
| 1 | 张三 | 19 |
| 2 | 李四 | 23 |

这种情况一般可以利用json存储或者是hash

**json存储**

redis 127.0.0.1:6379> set charmtest:user:1 '{"name":"张三","age":"19"}'OK

redis 127.0.0.1:6379> set charmtest:user:2 '{"name":"李四","age":"23"}'OK

redis 127.0.0.1:6379> get charmtest:user:1"{"name":"张三","age":"19"}"

redis 127.0.0.1:6379> get charmtest:user:2"{"name":"李四","age":"23"}"

**hash存储**

redis 127.0.0.1:6379:0>hset 'charmhash:user:1' name '张三'

"1"

redis 127.0.0.1:6379:0>hset 'charmhash:user:1' age 19

"1"

##获取数据

redis 127.0.0.1:6379:0>hget 'charmhash:user:1' name

"张三"

redis 127.0.0.1:6379:0>hget 'charmhash:user:1' age

"19"

## **21、Redis的过期策略**

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。

**22、实现LRU算法：**

LRU全称是Least Recently Used，即最近最久未使用的意思。

# LRU算法的设计原则是：****如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小****。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

实现方案

使用LinkedHashMap实现

     LinkedHashMap底层就是用的HashMap加双链表实现的，而且本身已经实现了按照访问顺序的存储。此外，LinkedHashMap中本身就实现了一个方法removeEldestEntry用于判断是否需要移除最不常读取的数，方法默认是直接返回false，不会移除元素，所以需要重写该方法。即当缓存满后就移除最不常用的数。

|  |  |
| --- | --- |
|  | public class LRU<K,V> {  private static final float hashLoadFactory = 0.75f;  private LinkedHashMap<K,V> map;  private int cacheSize;    public LRU(int cacheSize) {  this.cacheSize = cacheSize;  int capacity = (int)Math.ceil(cacheSize / hashLoadFactory) + 1;  map = new LinkedHashMap<K,V>(capacity, hashLoadFactory, true){  private static final long serialVersionUID = 1;    @Override  protected boolean removeEldestEntry(Map.Entry eldest) {  return size() > LRU.this.cacheSize;  }  };  }    public synchronized V get(K key) {  return map.get(key);  }    public synchronized void put(K key, V value) {  map.put(key, value);  }    public synchronized void clear() {  map.clear();  }    public synchronized int usedSize() {  return map.size();  }    public void print() {  for (Map.Entry<K, V> entry : map.entrySet()) {  System.out.print(entry.getValue() + "--");  }  System.out.println();  }  } |

**Redis底层数据结构**：

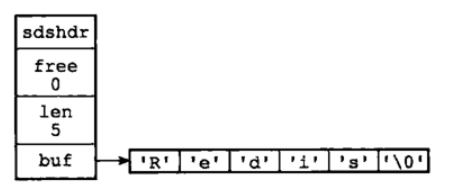
### **2、简单动态字符串**

　　第一篇文章我们就说过 Redis 是用 C 语言写的，但是对于Redis的字符串，却不是 C 语言中的字符串（即以空字符’\0’结尾的字符数组），它是自己构建了一种名为 简单动态字符串（simple dynamic string,SDS）的抽象类型，并将 SDS 作为 Redis的默认字符串表示。

**SDS 定义：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | struct sdshdr{       //记录buf数组中已使用字节的数量       //等于 SDS 保存字符串的长度  **int** len;       //记录 buf 数组中未使用字节的数量  **int** **free**;       //字节数组，用于保存字符串  **char** buf[];  } |

　　用SDS保存字符串 “Redis”具体图示如下：



　　　　　　　　　图片来源：《Redis设计与实现》

 　　我们看上面对于 SDS 数据类型的定义：

　　1、len 保存了SDS保存字符串的长度

　　2、buf[] 数组用来保存字符串的每个元素

　　3、free j记录了 buf 数组中未使用的字节数量

　　上面的定义相对于 C 语言对于字符串的定义，多出了 len 属性以及 free 属性。为什么不使用C语言字符串实现，而是使用 SDS呢？这样实现有什么好处？

**①、常数复杂度获取字符串长度**

　　由于 len 属性的存在，我们获取 SDS 字符串的长度只需要读取 len 属性，时间复杂度为 O(1)。而对于 C 语言，获取字符串的长度通常是经过遍历计数来实现的，时间复杂度为 O(n)。通过 strlen key 命令可以获取 key 的字符串长度。

**②、杜绝缓冲区溢出**

我们知道在 C 语言中使用 strcat  函数来进行两个字符串的拼接，一旦没有分配足够长度的内存空间，就会造成缓冲区溢出。而对于 SDS 数据类型，在进行字符修改的时候，会首先根据记录的 len 属性检查内存空间是否满足需求，如果不满足，会进行相应的空间扩展，然后在进行修改操作，所以不会出现缓冲区溢出。

**③、减少修改字符串的内存重新分配次数**

C语言由于不记录字符串的长度，所以如果要修改字符串，必须要重新分配内存（先释放再申请），因为如果没有重新分配，字符串长度增大时会造成内存缓冲区溢出，字符串长度减小时会造成内存泄露。

　　而对于SDS，由于len属性和free属性的存在，对于修改字符串SDS实现了空间预分配和惰性空间释放两种策略：

　　1、空间预分配：对字符串进行空间扩展的时候，扩展的内存比实际需要的多，这样可以减少连续执行字符串增长操作所需的内存重分配次数。

　　2、惰性空间释放：对字符串进行缩短操作时，程序不立即使用内存重新分配来回收缩短后多余的字节，而是使用 free 属性将这些字节的数量记录下来，等待后续使用。（当然SDS也提供了相应的API，当我们有需要时，也可以手动释放这些未使用的空间。）

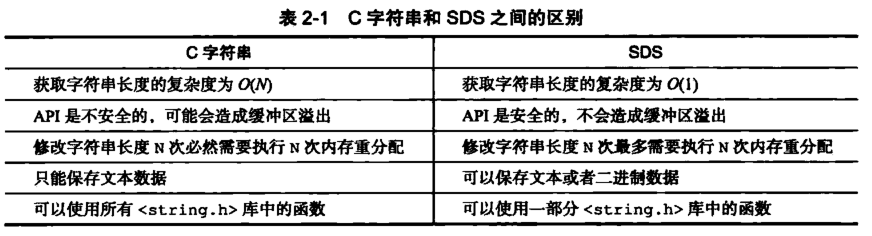
**④、二进制安全**

　　因为C字符串以空字符作为字符串结束的标识，而对于一些二进制文件（如图片等），内容可能包括空字符串，因此C字符串无法正确存取；而所有 SDS 的API 都是以处理二进制的方式来处理 buf 里面的元素，并且 SDS 不是以空字符串来判断是否结束，而是以 len 属性表示的长度来判断字符串是否结束。

**⑤、兼容部分 C 字符串函数**

　　虽然 SDS 是二进制安全的，但是一样遵从每个字符串都是以空字符串结尾的惯例，这样可以重用 C 语言库<string.h> 中的一部分函数。

**⑥、总结**



一般来说，SDS 除了保存数据库中的字符串值以外，SDS 还可以作为缓冲区（buffer）：包括 AOF 模块中的AOF缓冲区以及客户端状态中的输入缓冲区。后面在介绍Redis的持久化时会进行介绍。

### **3、链表**

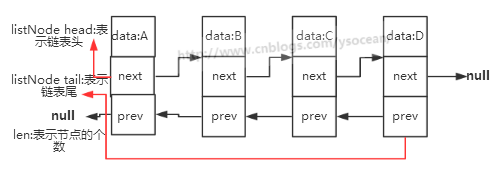
　　链表是一种常用的数据结构，C 语言内部是没有内置这种数据结构的实现，所以Redis自己构建了链表的实现。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | typedef  struct listNode{         //前置节点         struct listNode \*prev;         //后置节点         struct listNode \*next;         //节点的值         void \*value;  }listNode |

　　链表定义：

　　通过多个 listNode 结构就可以组成链表，这是一个双端链表，Redis还提供了操作链表的数据结构：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | typedef struct list{       //表头节点       listNode \*head;       //表尾节点       listNode \*tail;       //链表所包含的节点数量       unsigned **long** len;       //节点值复制函数       void (\***free**) (void \*ptr);       //节点值释放函数       void (\***free**) (void \*ptr);       //节点值对比函数  **int** (\*match) (void \*ptr,void \*key);  }list; |



　　Redis链表特性：

　　①、双端：链表具有前置节点和后置节点的引用，获取这两个节点时间复杂度都为O(1)。

　　②、无环：表头节点的 prev 指针和表尾节点的 next 指针都指向 NULL,对链表的访问都是以 NULL 结束。

　　③、带链表长度计数器：通过 len 属性获取链表长度的时间复杂度为 O(1)。

　　④、多态：链表节点使用 void\* 指针来保存节点值，可以保存各种不同类型的值。

### **4、字典**

　　字典又称为符号表或者关联数组、或映射（map），是一种用于保存键值对的抽象数据结构。字典中的每一个键 key 都是唯一的，通过 key 可以对值来进行查找或修改。C 语言中没有内置这种数据结构的实现，所以字典依然是 Redis自己构建的。

　　Redis 的字典使用哈希表作为底层实现，关于哈希表的详细讲解可以参考我[这篇博客](http://www.cnblogs.com/ysocean/p/8032656.html" \t "https://www.cnblogs.com/ysocean/p/_blank)。

　　哈希表结构定义：

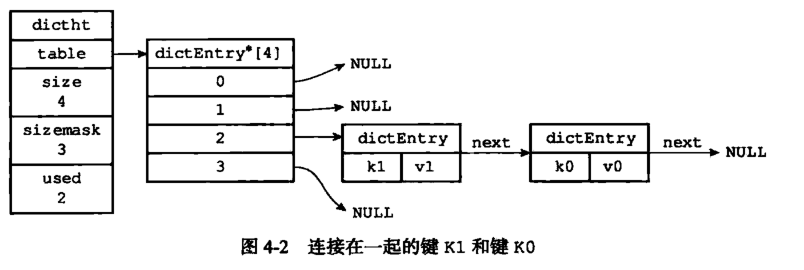
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | typedef struct dictht{       //哈希表数组       dictEntry \*\*table;       //哈希表大小       unsigned **long** size;       //哈希表大小掩码，用于计算索引值       //总是等于 size-1       unsigned **long** sizemask;       //该哈希表已有节点的数量       unsigned **long** used;    }dictht |

　　哈希表是由数组 table 组成，table 中每个元素都是指向 dict.h/dictEntry 结构，dictEntry 结构定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | typedef struct dictEntry{       //键       void \*key;       //值       union{            void \*val;            uint64\_tu64;            int64\_ts64;       }v;         //指向下一个哈希表节点，形成链表       struct dictEntry \*next;  }dictEntry |

　　key 用来保存键，val 属性用来保存值，值可以是一个指针，也可以是uint64\_t整数，也可以是int64\_t整数。

　　注意这里还有一个指向下一个哈希表节点的指针，我们知道哈希表最大的问题是存在哈希冲突，如何解决哈希冲突，有开放地址法和链地址法。这里采用的便是链地址法，通过next这个指针可以将多个哈希值相同的键值对连接在一起，用来解决**哈希冲突**。



**①、哈希算法：**Redis计算哈希值和索引值方法如下：

|  |  |
| --- | --- |
| 1  2  3  4 | #1、使用字典设置的哈希函数，计算键 key 的哈希值  hash = dict->type->hashFunction(key);  #2、使用哈希表的sizemask属性和第一步得到的哈希值，计算索引值  index = hash & dict->ht[x].sizemask; |

**②、解决哈希冲突：**这个问题上面我们介绍了，方法是链地址法。通过字典里面的 \*next 指针指向下一个具有相同索引值的哈希表节点。

**③、扩容和收缩：**当哈希表保存的键值对太多或者太少时，就要通过 rerehash(重新散列）来对哈希表进行相应的扩展或者收缩。具体步骤：

　　　　　　1、如果执行扩展操作，会基于原哈希表创建一个大小等于 ht[0].used\*2n 的哈希表（也就是每次扩展都是根据原哈希表已使用的空间扩大一倍创建另一个哈希表）。相反如果执行的是收缩操作，每次收缩是根据已使用空间缩小一倍创建一个新的哈希表。

　　　　　　2、重新利用上面的哈希算法，计算索引值，然后将键值对放到新的哈希表位置上。

　　　　　　3、所有键值对都迁徙完毕后，释放原哈希表的内存空间。

**④、触发扩容的条件：**

　　　　　　1、服务器目前没有执行 BGSAVE 命令或者 BGREWRITEAOF 命令，并且负载因子大于等于1。

　　　　　　2、服务器目前正在执行 BGSAVE 命令或者 BGREWRITEAOF 命令，并且负载因子大于等于5。

　　　　ps：负载因子 = 哈希表已保存节点数量 / 哈希表大小。

**⑤、渐近式 rehash**

　　　　什么叫渐进式 rehash？也就是说扩容和收缩操作不是一次性、集中式完成的，而是分多次、渐进式完成的。如果保存在Redis中的键值对只有几个几十个，那么 rehash 操作可以瞬间完成，但是如果键值对有几百万，几千万甚至几亿，那么要一次性的进行 rehash，势必会造成Redis一段时间内不能进行别的操作。所以Redis采用渐进式 rehash,这样在进行渐进式rehash期间，字典的删除查找更新等操作可能会在两个哈希表上进行，第一个哈希表没有找到，就会去第二个哈希表上进行查找。但是进行 增加操作，一定是在新的哈希表上进行的。

### **5、跳跃表**

　　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。具有如下性质：

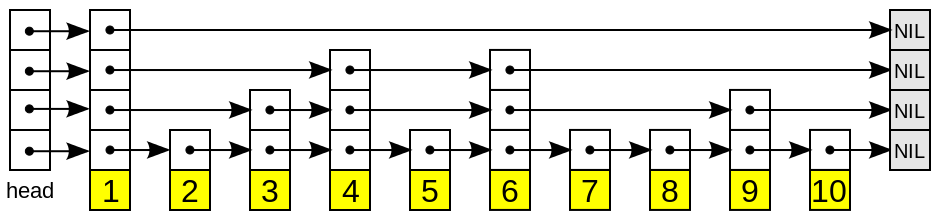
　　1、由很多层结构组成；

　　2、每一层都是一个有序的链表，排列顺序为由高层到底层，都至少包含两个链表节点，分别是前面的head节点和后面的nil节点；

　　3、最底层的链表包含了所有的元素；

　　4、如果一个元素出现在某一层的链表中，那么在该层之下的链表也全都会出现（上一层的元素是当前层的元素的子集）；

　　5、链表中的每个节点都包含两个指针，一个指向同一层的下一个链表节点，另一个指向下一层的同一个链表节点；

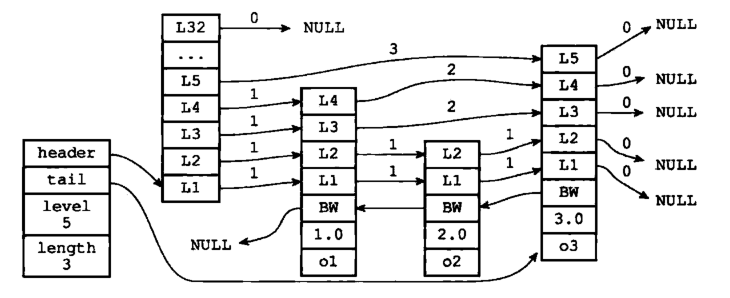


　　Redis中跳跃表节点定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | typedef struct zskiplistNode {       //层       struct zskiplistLevel{             //前进指针             struct zskiplistNode \*forward;             //跨度             unsigned **int** span;       }level[];         //后退指针       struct zskiplistNode \*backward;       //分值  **double** score;       //成员对象       robj \*obj;    } zskiplistNode |

　　多个跳跃表节点构成一个跳跃表：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | typedef struct zskiplist{       //表头节点和表尾节点       structz skiplistNode \*header, \*tail;       //表中节点的数量       unsigned **long** length;       //表中层数最大的节点的层数  **int** level;    }zskiplist; |



　　①、搜索：从最高层的链表节点开始，如果比当前节点要大和比当前层的下一个节点要小，那么则往下找，也就是和当前层的下一层的节点的下一个节点进行比较，以此类推，一直找到最底层的最后一个节点，如果找到则返回，反之则返回空。

　　②、插入：首先确定插入的层数，有一种方法是假设抛一枚硬币，如果是正面就累加，直到遇见反面为止，最后记录正面的次数作为插入的层数。当确定插入的层数k后，则需要将新元素插入到从底层到k层。

　　③、删除：在各个层中找到包含指定值的节点，然后将节点从链表中删除即可，如果删除以后只剩下头尾两个节点，则删除这一层。

### **6、整数集合**

　　整数集合（intset）是Redis用于保存整数值的集合抽象数据类型，它可以保存类型为int16\_t、int32\_t 或者int64\_t 的整数值，并且保证集合中不会出现重复元素。

　　定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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; |

　　整数集合的每个元素都是 contents 数组的一个数据项，它们按照从小到大的顺序排列，并且不包含任何重复项。

　　length 属性记录了 contents 数组的大小。

　　需要注意的是虽然 contents 数组声明为 int8\_t 类型，但是实际上contents 数组并不保存任何 int8\_t 类型的值，其真正类型有 encoding 来决定。

**①、升级**

　　当我们新增的元素类型比原集合元素类型的长度要大时，需要对整数集合进行升级，才能将新元素放入整数集合中。具体步骤：

　　1、根据新元素类型，扩展整数集合底层数组的大小，并为新元素分配空间。

　　2、将底层数组现有的所有元素都转成与新元素相同类型的元素，并将转换后的元素放到正确的位置，放置过程中，维持整个元素顺序都是有序的。

　　3、将新元素添加到整数集合中（保证有序）。

　　升级能极大地节省内存。

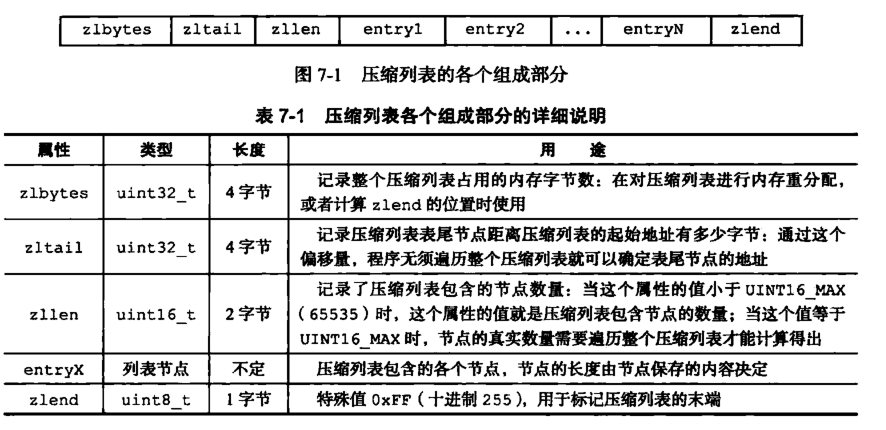
**②、降级**

　　整数集合不支持降级操作，一旦对数组进行了升级，编码就会一直保持升级后的状态。

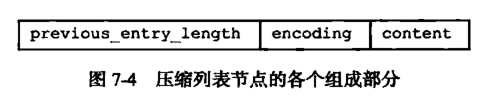
### **7、压缩列表**

　　压缩列表（ziplist）是Redis为了节省内存而开发的，是由一系列特殊编码的连续内存块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。

**压缩列表的原理：压缩列表并不是对数据利用某种算法进行压缩，而是将数据按照一定规则编码在一块连续的内存区域，目的是节省内存。**



　　压缩列表的每个节点构成如下：



　　①、previous\_entry\_ength：记录压缩列表前一个字节的长度。previous\_entry\_ength的长度可能是1个字节或者是5个字节，如果上一个节点的长度小于254，则该节点只需要一个字节就可以表示前一个节点的长度了，如果前一个节点的长度大于等于254，则previous length的第一个字节为254，后面用四个字节表示当前节点前一个节点的长度。利用此原理即当前节点位置减去上一个节点的长度即得到上一个节点的起始位置，压缩列表可以从尾部向头部遍历。这么做很有效地减少了内存的浪费。

　　②、encoding：节点的encoding保存的是节点的content的内容类型以及长度，encoding类型一共有两种，一种字节数组一种是整数，encoding区域长度为1字节、2字节或者5字节长。

　　③、content：content区域用于保存节点的内容，节点内容类型和长度由encoding决定。

### **8、总结**

　　大多数情况下，Redis使用简单字符串SDS作为字符串的表示，相对于C语言字符串，SDS具有常数复杂度获取字符串长度，杜绝了缓存区的溢出，减少了修改字符串长度时所需的内存重分配次数，以及二进制安全能存储各种类型的文件，并且还兼容部分C函数。

　　通过为链表设置不同类型的特定函数，Redis链表可以保存各种不同类型的值，除了用作列表键，还在发布与订阅、慢查询、监视器等方面发挥作用（后面会介绍）。

　　Redis的字典底层使用哈希表实现，每个字典通常有两个哈希表，一个平时使用，另一个用于rehash时使用，使用链地址法解决哈希冲突。

　　跳跃表通常是有序集合的底层实现之一，表中的节点按照分值大小进行排序。

　　整数集合是集合键的底层实现之一，底层由数组构成，升级特性能尽可能的节省内存。

　　压缩列表是Redis为节省内存而开发的顺序型数据结构，通常作为列表键和哈希键的底层实现之一。