**redis详细说明文档**

|  |  |
| --- | --- |
|  | |
| **版本** | **V-1.0** |
| **最近更新人** | **夏灵杰** |
| **最近更新时间** | **2013/4/1** |
| **文档说明** | **redis功能使用说明** |

|  |  |  |  |
| --- | --- | --- | --- |
| **更新一览** | | | |
| **修改内容** | **修改时间** | **修改人** | **备注** |
|  |  |  |  |

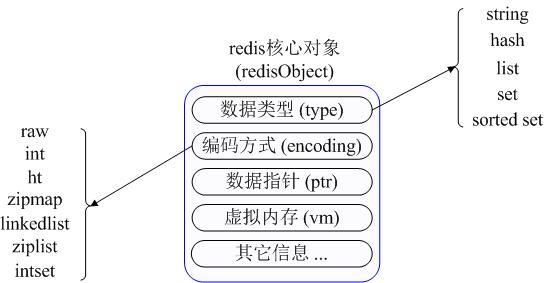
## Redis 简介

**说明：redis官网[http://redis.cn/](http://redis.cn/commands.html)**

### Redis 功能简介

redis支持复制的数据类型，可持久化，也支持多点备份

#### Redis常用的数据类型



**首先Redis内部使用一个redisObject对象来表示所有的key和value,redisObject最主要的信息如上图所示：type代表一个value对象具体是何种数据类型，encoding是不同数据类型在redis内部的存储方式，比如：type=string代表value存储的是一个普通字符串，那么对应的encoding可以是raw或者是int,如果是int则代表实际redis内部是按数值型类存储和表示这个字符串的，当然前提是这个字符串本身可以用数值表示，比如:"123" "456"这样的字符串。**

**这里需要特殊说明一下vm字段，只有打开了Redis的虚拟内存功能，此字段才会真正的分配内存，该功能默认是关闭状态的，该功能会在后面具体描述。通过上图我们可以发现Redis使用redisObject来表示所有的key/value数据是比较浪费内存的，当然这些内存管理成本的付出主要也是为了给Redis不同数据类型提供一个统一的管理接口，实际作者也提供了多种方法帮助我们尽量节省内存使用。**

##### String

|  |
| --- |
| **常用命令：**  set,get,decr,incr,mget 等。  **应用场景：**  String是最常用的一种数据类型，普通的key/value存储都可以归为此类，这里就不所做解释了。  **实现方式：**  String在redis内部存储默认就是一个字符串，被redisObject所引用，当遇到incr,decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。 |

##### Hash

|  |
| --- |
| **常用命令：**  hget,hset,hgetall 等。  **应用场景：**  一个用户信息的存储，有3中方式：  用户id+对象序列化  用户id+姓名 用户id+年龄 用户id+……  用户id+hash<姓名，”jie”>的hashMap  也就是说，Key仍然是用户ID, value是一个Map，这个Map的key是成员的属性名，value是属性值，这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field), 也就是通过 key(用户ID) + field(属性标签) 就可以操作对应属性数据了，既不需要重复存储数据，也不会带来序列化和并发修改控制的问题。很好的解决了问题。  这里同时需要注意，Redis提供了接口(hgetall)可以直接取到全部的属性数据,但是如果内部Map的成员很多，那么涉及到遍历整个内部Map的操作，由于Redis单线程模型的缘故，这个遍历操作可能会比较耗时，而另其它客户端的请求完全不响应，这点需要格外注意。  **实现方式：**  上面已经说到Redis Hash对应Value内部实际就是一个HashMap，实际这里会有2种不同实现，这个Hash的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，对应的value redisObject的encoding为zipmap,当成员数量增大时会自动转成真正的HashMap,此时encoding为ht。 |

##### List

|  |
| --- |
| **常用命令：**  lpush,rpush,lpop,rpop,lrange等。  **应用场景：**  Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现，比较好理解，这里不再重复。  **实现方式：**  Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。 |

##### Set

|  |
| --- |
| **常用命令：**  sadd,spop,smembers,sunion 等。  **应用场景：**  Redis set对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的。  **实现方式：**  set 的内部实现是一个 value永远为null的HashMap，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。 |

##### Sorted set

|  |
| --- |
| **常用命令：**  zadd,zrange,zrem,zcard等  **使用场景：**  Redis sorted set的使用场景与set类似，区别是set不是自动有序的，而sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择sorted set数据结构，比如twitter 的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。  **实现方式：**  Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。 |

#### Redis持久化

|  |
| --- |
| Redis由于支持非常丰富的内存数据结构类型，如何把这些复杂的内存组织方式持久化到磁盘上是一个难题，所以Redis的持久化方式与传统数据库的方式有比较多的差别，Redis一共支持四种持久化方式，分别是：   * 定时快照方式(snapshot) * 基于语句追加文件的方式(aof) * 虚拟内存(vm) * Diskstore方式   在设计思路上，前两种是基于全部数据都在内存中，即小数据量下提供磁盘落地功能，而后两种方式则是作者在尝试存储数据超过物理内存时，即大数据量的数据存储，截止到本文，后两种持久化方式仍然是在实验阶段，并且vm方式基本已经被作者放弃，所以实际能在生产环境用的只有前两种，换句话说Redis目前还只能作为小数据量存储（全部数据能够加载在内存中），海量数据存储方面并不是Redis所擅长的领域。下面分别介绍下这几种持久化方式：  **定时快照方式(snapshot)：**  该持久化方式实际是在Redis内部一个定时器事件，每隔固定时间去检查当前数据发生的改变次数与时间是否满足配置的持久化触发的条件，如果满足则通过操作系统fork调用来创建出一个子进程，这个子进程默认会与父进程共享相同的地址空间，这时就可以通过子进程来遍历整个内存来进行存储操作，而主进程则仍然可以提供服务，当有写入时由操作系统按照内存页(page)为单位来进行copy-on-write保证父子进程之间不会互相影响。  该持久化的主要缺点是定时快照只是代表一段时间内的内存映像，所以系统重启会丢失上次快照与重启之间所有的数据。  **基于语句追加方式(aof)：**  aof方式实际类似mysql的基于语句的binlog方式，即每条会使Redis内存数据发生改变的命令都会追加到一个log文件中，也就是说这个log文件就是Redis的持久化数据。  aof的方式的主要缺点是追加log文件可能导致体积过大，当系统重启恢复数据时如果是aof的方式则加载数据会非常慢，几十G的数据可能需要几小时才能加载完，当然这个耗时并不是因为磁盘文件读取速度慢，而是由于读取的所有命令都要在内存中执行一遍。另外由于每条命令都要写log,所以使用aof的方式，Redis的读写性能也会有所下降。  **虚拟内存方式：**  虚拟内存方式是Redis来进行用户空间的数据换入换出的一个策略，此种方式在实现的效果上比较差，主要问题是代码复杂，重启慢，复制慢等等，目前已经被作者放弃。  **diskstore方式：**  diskstore方式是作者放弃了虚拟内存方式后选择的一种新的实现方式，也就是传统的B-tree的方式，目前仍在实验阶段，后续是否可用我们可以拭目以待。 |

### Redis 应用实例

## Redis 详解

### 应用

#### redis下载安装

##### 下载安装server

|  |
| --- |
| @user:wget http://redis.googlecode.com/files/redis-2.6.11.tar.gz (去官网查看最新版本)  @user:tar -zxf redis-2.6.11.tar.gz  @user:cd redis-2.6.11  @user:make  @user:sudo make install  这时Redis 的可执行文件被放到了/usr/local/bin |

##### 下载配置文件和init启动脚本

|  |
| --- |
| @user:wget https://github.com/ijonas/dotfiles/raw/master/etc/init.d/redis-server  @user:wget https://github.com/ijonas/dotfiles/raw/master/etc/redis.conf  @user:sudo mv redis-server /etc/init.d/redis-server  @user:sudo chmod +x /etc/init.d/redis-server  @user:sudo mv redis.conf /etc/redis.conf |

##### 初始化用户和日志路径

第一次启动Redis前，建议为Redis单独建立一个用户，并新建data和日志文件夹

|  |
| --- |
| @user:sudo useradd redis  @user:sudo mkdir -p /var/lib/redis  @user:sudo mkdir -p /var/log/redis  @user:sudo chown redis.redis /var/lib/redis  @user:sudo chown redis.redis /var/log/redis |

##### 开机自动启动，关机自动关闭

|  |
| --- |
| @user: sudo update-rc.d redis-server defaults |

##### 启动测试redis

|  |
| --- |
| @user: sudo /etc/init.d/redis-server start  @user: redis-cli  redis> set test redis  OK  redis> get test  "redis" |

##### 重启redis

重新读取指定redis.conf文件来启动redis

|  |
| --- |
| @user: sudo /etc/init.d/redis-server /etc/redis.conf |

#### redis server

##### redis 命令

参考详细：<http://redis.cn/commands.html>

|  |
| --- |
| **Redis-cli连接server**  [APPEND](http://redis.cn/commands/append.html) key value追加一个值到key上  [AUTH](http://redis.cn/commands/auth.html) password验证服务器  [BGREWRITEAOF](http://redis.cn/commands/bgrewriteaof.html) 异步重写追加文件  [BGSAVE](http://redis.cn/commands/bgsave.html) 异步保存数据集到磁盘上  [BITCOUNT](http://redis.cn/commands/bitcount.html) key [start] [end]统计字符串指定起始位置的字节数  [BITOP](http://redis.cn/commands/bitop.html) operation destkey key [key ...]Perform bitwise operations between strings  [BLPOP](http://redis.cn/commands/blpop.html) key [key ...] timeout删除，并获得该列表中的第一元素，或阻塞，直到有一个可用  [BRPOP](http://redis.cn/commands/brpop.html) key [key ...] timeout删除，并获得该列表中的最后一个元素，或阻塞，直到有一个可用  [BRPOPLPUSH](http://redis.cn/commands/brpoplpush.html) source destination timeout弹出一个列表的值，将它推到另一个列表，并返回它;或阻塞，直到有一个可用  [CLIENT KILL](http://redis.cn/commands/client-kill.html) ip:port关闭客户端连接  [CLIENT LIST](http://redis.cn/commands/client-list.html) 获得客户端连接列表  [CLIENT GETNAME](http://redis.cn/commands/client-getname.html) 获得当前连接名称  [CLIENT SETNAME](http://redis.cn/commands/client-setname.html) connection-name设置当前连接的名字  [CONFIG GET](http://redis.cn/commands/config-get.html) parameter获取配置参数的值  [CONFIG SET](http://redis.cn/commands/config-set.html) parameter value获取配置参数的值  [CONFIG RESETSTAT](http://redis.cn/commands/config-resetstat.html) 复位再分配使用info命令报告的统计  [DBSIZE](http://redis.cn/commands/dbsize.html) 返回当前数据库里面的keys数量  [DEBUG OBJECT](http://redis.cn/commands/debug-object.html) key获取一个key的debug信息  [DEBUG SEGFAULT](http://redis.cn/commands/debug-segfault.html) 使服务器崩溃  [DECR](http://redis.cn/commands/decr.html) key整数原子减1  [DECRBY](http://redis.cn/commands/decrby.html) key decrement原子减指定的整数  [DEL](http://redis.cn/commands/del.html) key [key ...]删除一个key  [DISCARD](http://redis.cn/commands/discard.html) 丢弃所有 MULTI 之后发的命令  [DUMP](http://redis.cn/commands/dump.html) key导出key的值  [ECHO](http://redis.cn/commands/echo.html) message回显输入的字符串  [EVAL](http://redis.cn/commands/eval.html) script numkeys key [key ...] arg [arg ...]在服务器端执行 LUA 脚本  [EVALSHA](http://redis.cn/commands/evalsha.html) sha1 numkeys key [key ...] arg [arg ...]在服务器端执行 LUA 脚本  [EXEC](http://redis.cn/commands/exec.html) 执行所有 MULTI 之后发的命令  [EXISTS](http://redis.cn/commands/exists.html) key查询一个key是否存在  [EXPIRE](http://redis.cn/commands/expire.html) key seconds设置一个key的过期的秒数  [EXPIREAT](http://redis.cn/commands/expireat.html) key timestamp设置一个UNIX时间戳的过期时间  [FLUSHALL](http://redis.cn/commands/flushall.html) 清空所有数据库  [FLUSHDB](http://redis.cn/commands/flushdb.html) 清空当前的数据库  [GET](http://redis.cn/commands/get.html) key获取key的值  [GETBIT](http://redis.cn/commands/getbit.html) key offset返回位的值存储在关键的字符串值的偏移量。  [GETRANGE](http://redis.cn/commands/getrange.html) key start end获取存储在key上的值的一个子字符串  [GETSET](http://redis.cn/commands/getset.html) key value设置一个key的value，并获取设置前的值  [HDEL](http://redis.cn/commands/hdel.html) key field [field ...]删除一个或多个哈希域  [HEXISTS](http://redis.cn/commands/hexists.html) key field判断给定域是否存在于哈希集中  [HGET](http://redis.cn/commands/hget.html) key field读取哈希域的的值  [HGETALL](http://redis.cn/commands/hgetall.html) key从哈希集中读取全部的域和值  [HINCRBY](http://redis.cn/commands/hincrby.html) key field increment将哈希集中指定域的值增加给定的数字  [HINCRBYFLOAT](http://redis.cn/commands/hincrbyfloat.html) key field increment将哈希集中指定域的值增加给定的浮点数  [HKEYS](http://redis.cn/commands/hkeys.html) key获取hash的所有字段  [HLEN](http://redis.cn/commands/hlen.html) key获取hash里所有字段的数量  [HMGET](http://redis.cn/commands/hmget.html) key field [field ...]获取hash里面指定字段的值  [HMSET](http://redis.cn/commands/hmset.html) key field value [field value ...]设置hash字段值  [HSET](http://redis.cn/commands/hset.html) key field value设置hash里面一个字段的值  [HSETNX](http://redis.cn/commands/hsetnx.html) key field value设置hash的一个字段，只有当这个字段不存在时有效  [HVALS](http://redis.cn/commands/hvals.html) key获得hash的所有值  [INCR](http://redis.cn/commands/incr.html) key执行原子加1操作  [INCRBY](http://redis.cn/commands/incrby.html) key increment执行原子增加一个整数  [INCRBYFLOAT](http://redis.cn/commands/incrbyfloat.html) key increment执行原子增加一个浮点数  [INFO](http://redis.cn/commands/info.html) [section]获得服务器的详细信息  [KEYS](http://redis.cn/commands/keys.html) pattern查找所有匹配给定的模式的键  [LASTSAVE](http://redis.cn/commands/lastsave.html) 获得最后一次同步磁盘的时间  [LINDEX](http://redis.cn/commands/lindex.html) key index获取一个元素，通过其索引列表  [LINSERT](http://redis.cn/commands/linsert.html) key BEFORE|AFTER pivot value在列表中的另一个元素之前或之后插入一个元素  [LLEN](http://redis.cn/commands/llen.html) key获得队列(List)的长度  [LPOP](http://redis.cn/commands/lpop.html) key从队列的左边出队一个元素  [LPUSH](http://redis.cn/commands/lpush.html) key value [value ...]从队到左边入队一个元素  [LPUSHX](http://redis.cn/commands/lpushx.html) key value当队列存在时，从队到左边入队一个元素  [LRANGE](http://redis.cn/commands/lrange.html) key start stop从列表中获取指定返回的元素  [LREM](http://redis.cn/commands/lrem.html) key count value从列表中删除元素  [LSET](http://redis.cn/commands/lset.html) key index value设置队列里面一个元素的值  [LTRIM](http://redis.cn/commands/ltrim.html) key start stop修剪到指定范围内的清单  [MGET](http://redis.cn/commands/mget.html) key [key ...]获得所有key的值  [MIGRATE](http://redis.cn/commands/migrate.html) host port key destination-db timeout原子性的将key从redis的一个实例移到另一个实例  [MONITOR](http://redis.cn/commands/monitor.html) 实时监控服务器  [MOVE](http://redis.cn/commands/move.html) key db移动一个key到另一个数据库  [MSET](http://redis.cn/commands/mset.html) key value [key value ...]设置多个key value  [MSETNX](http://redis.cn/commands/msetnx.html) key value [key value ...]设置多个key value,仅当key存在时  [MULTI](http://redis.cn/commands/multi.html) 标记一个事务块开始  [OBJECT](http://redis.cn/commands/object.html) subcommand [arguments [arguments ...]]检查内部的再分配对象  [PERSIST](http://redis.cn/commands/persist.html) key移除key的过期时间  [PEXPIRE](http://redis.cn/commands/pexpire.html) key milliseconds设置一个key的过期的毫秒数  [PEXPIREAT](http://redis.cn/commands/pexpireat.html) key milliseconds-timestamp设置一个带毫秒的UNIX时间戳的过期时间  [PING](http://redis.cn/commands/ping.html) Ping 服务器  [PSETEX](http://redis.cn/commands/psetex.html) key milliseconds valueSet the value and expiration in milliseconds of a key  [PSUBSCRIBE](http://redis.cn/commands/psubscribe.html) pattern [pattern ...]听出版匹配给定模式的渠道的消息  [PTTL](http://redis.cn/commands/pttl.html) key获取key的有效毫秒数  [PUBLISH](http://redis.cn/commands/publish.html) channel message发布一条消息到频道  [PUNSUBSCRIBE](http://redis.cn/commands/punsubscribe.html) [pattern [pattern ...]]停止发布到匹配给定模式的渠道的消息听  [QUIT](http://redis.cn/commands/quit.html) 关闭连接，退出  [RANDOMKEY](http://redis.cn/commands/randomkey.html) 返回一个随机的key  [RENAME](http://redis.cn/commands/rename.html) key newkey将一个key重命名  [RENAMENX](http://redis.cn/commands/renamenx.html) key newkey重命名一个key,新的key必须是不存在的key  [RESTORE](http://redis.cn/commands/restore.html) key ttl serialized-valueCreate a key using the provided serialized value, previously obtained using DUMP.  [RPOP](http://redis.cn/commands/rpop.html) key从队列的右边出队一个元素  [RPOPLPUSH](http://redis.cn/commands/rpoplpush.html) source destination删除列表中的最后一个元素，将其追加到另一个列表  [RPUSH](http://redis.cn/commands/rpush.html) key value [value ...]从队列的右边入队一个元素  [RPUSHX](http://redis.cn/commands/rpushx.html) key value从队列的右边入队一个元素，仅队列存在时有效  [SADD](http://redis.cn/commands/sadd.html) key member [member ...]添加一个或者多个元素到集合(set)里  [SAVE](http://redis.cn/commands/save.html) 同步数据到磁盘上  [SCARD](http://redis.cn/commands/scard.html) key获取集合里面的元素数量  [SCRIPT EXISTS](http://redis.cn/commands/script-exists.html) script [script ...]Check existence of scripts in the script cache.  [SCRIPT FLUSH](http://redis.cn/commands/script-flush.html) 删除服务器缓存中所有Lua脚本。  [SCRIPT KILL](http://redis.cn/commands/script-kill.html) 杀死当前正在运行的 Lua 脚本。  [SCRIPT LOAD](http://redis.cn/commands/script-load.html) script从服务器缓存中装载一个Lua脚本。  [SDIFF](http://redis.cn/commands/sdiff.html) key [key ...]获得队列不存在的元素  [SDIFFSTORE](http://redis.cn/commands/sdiffstore.html) destination key [key ...]获得队列不存在的元素，并存储在一个关键的结果集  [SELECT](http://redis.cn/commands/select.html) index选择数据库  [SET](http://redis.cn/commands/set.html) key value设置一个key的value值  [SETBIT](http://redis.cn/commands/setbit.html) key offset valueSets or clears the bit at offset in the string value stored at key  [SETEX](http://redis.cn/commands/setex.html) key seconds value设置key-value并设置过期时间（单位：秒）  [SETNX](http://redis.cn/commands/setnx.html) key value设置的一个关键的价值，只有当该键不存在  [SETRANGE](http://redis.cn/commands/setrange.html) key offset valueOverwrite part of a string at key starting at the specified offset  [SHUTDOWN](http://redis.cn/commands/shutdown.html) [NOSAVE] [SAVE]关闭服务  [SINTER](http://redis.cn/commands/sinter.html) key [key ...]获得两个集合的交集  [SINTERSTORE](http://redis.cn/commands/sinterstore.html) destination key [key ...]获得两个集合的交集，并存储在一个关键的结果集  [SISMEMBER](http://redis.cn/commands/sismember.html) key member确定一个给定的值是一个集合的成员  [SLAVEOF](http://redis.cn/commands/slaveof.html) host port指定当前服务器的主服务器  [SLOWLOG](http://redis.cn/commands/slowlog.html) subcommand [argument]管理再分配的慢查询日志  [SMEMBERS](http://redis.cn/commands/smembers.html) key获取集合里面的所有key  [SMOVE](http://redis.cn/commands/smove.html) source destination member移动集合里面的一个key到另一个集合  [SORT](http://redis.cn/commands/sort.html) key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA] [STORE destination]对队列、集合、有序集合排序  [SPOP](http://redis.cn/commands/spop.html) key删除并获取一个集合里面的元素  [SRANDMEMBER](http://redis.cn/commands/srandmember.html) key [count]从集合里面随机获取一个key  [SREM](http://redis.cn/commands/srem.html) key member [member ...]从集合里删除一个或多个key  [STRLEN](http://redis.cn/commands/strlen.html) key获取指定key值的长度  [SUBSCRIBE](http://redis.cn/commands/subscribe.html) channel [channel ...]聆听发布途径的消息  [SUNION](http://redis.cn/commands/sunion.html) key [key ...]添加多个set元素  [SUNIONSTORE](http://redis.cn/commands/sunionstore.html) destination key [key ...]合并set元素，并将结果存入新的set里面  [SYNC](http://redis.cn/commands/sync.html) 用于复制的内部命令  [TIME](http://redis.cn/commands/time.html) 返回当前服务器时间  [TTL](http://redis.cn/commands/ttl.html) key获取key的有效时间（单位：秒）  [TYPE](http://redis.cn/commands/type.html) key获取key的存储类型  [UNSUBSCRIBE](http://redis.cn/commands/unsubscribe.html) [channel [channel ...]]停止发布途径的消息听  [UNWATCH](http://redis.cn/commands/unwatch.html) 取消事务  [WATCH](http://redis.cn/commands/watch.html) key [key ...]锁定key直到执行了 MULTI/EXEC 命令  [ZADD](http://redis.cn/commands/zadd.html) key score member [score member ...]添加到有序set的一个或多个成员，或更新的分数，如果它已经存在  [ZCARD](http://redis.cn/commands/zcard.html) key获取一个排序的集合中的成员数量  [ZCOUNT](http://redis.cn/commands/zcount.html) key min max给定值范围内的成员数与分数排序  [ZINCRBY](http://redis.cn/commands/zincrby.html) key increment member增量的一名成员在排序设置的评分  [ZINTERSTORE](http://redis.cn/commands/zinterstore.html) destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]相交多个排序集，导致排序的设置存储在一个新的关键  [ZRANGE](http://redis.cn/commands/zrange.html) key start stop [WITHSCORES]返回的成员在排序设置的范围，由指数  [ZRANGEBYSCORE](http://redis.cn/commands/zrangebyscore.html) key min max [WITHSCORES] [LIMIT offset count]返回的成员在排序设置的范围，由得分  [ZRANK](http://redis.cn/commands/zrank.html) key member确定在排序集合成员的索引  [ZREM](http://redis.cn/commands/zrem.html) key member [member ...]从排序的集合中删除一个或多个成员  [ZREMRANGEBYRANK](http://redis.cn/commands/zremrangebyrank.html) key start stop在排序设置的所有成员在给定的索引中删除  [ZREMRANGEBYSCORE](http://redis.cn/commands/zremrangebyscore.html) key min max删除一个排序的设置在给定的分数所有成员  [ZREVRANGE](http://redis.cn/commands/zrevrange.html) key start stop [WITHSCORES]在排序的设置返回的成员范围，通过索引，下令从分数高到低  [ZREVRANGEBYSCORE](http://redis.cn/commands/zrevrangebyscore.html) key max min [WITHSCORES] [LIMIT offset count]返回的成员在排序设置的范围，由得分，下令从分数高到低  [ZREVRANK](http://redis.cn/commands/zrevrank.html) key member确定指数在排序集的成员，下令从分数高到低  [ZSCORE](http://redis.cn/commands/zscore.html) key member获取成员在排序设置相关的比分  [ZUNIONSTORE](http://redis.cn/commands/zunionstore.html) destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]添加多个排序集和导致排序的设置存储在一 |

##### redis.conf配置

|  |
| --- |
| #是否以后台进程运行，默认为no，如果需要以后台进程运行则改为yes  daemonize no      #如果以后台进程运行的话，就需要指定pid，你可以在此自定义redis.pid文件的位置。  pidfile /var/run/redis.pid      #接受连接的端口号，如果端口是0则redis将不会监听TCP socket连接  port 6379    # If you want you can bind a single interface, if the bind option is not  # specified all the interfaces will listen for incoming connections.  #  # bind 127.0.0.1    # Specify the path for the unix socket that will be used to listen for  # incoming connections. There is no default, so Redis will not listen  # on a unix socket when not specified.  #  # unixsocket /tmp/redis.sock  # unixsocketperm 755      #连接超时时间，单位秒。(0 to disable)？  timeout 300000000      #日志级别，默认是verbose（详细），各种日志级别：  #debug:很详细的信息，适合开发和测试  #verbose:包含许多不太有用的信息，但比debug要清爽一些（many rarely useful info, but not a mess like #the debug level）  #notice:比较适合生产环境  #warning:警告信息  loglevel verbose      #指定log文件的名字，默认是stdout。stdout会让redis把日志输出到标准输出。但是如果使用stdout而又以后台进#程的方式运行redis，则日志会输出到/dev/null  logfile stdout      #'syslog-enabled'设置为yes会把日志输出到系统日志，默认是no  # syslog-enabled no      #指定syslog的标示符，如果'syslog-enabled'是no，则这个选项无效。  # syslog-ident redis      #指定syslog 设备（facility), 必须是USER或者LOCAL0到LOCAL7.  # syslog-facility local0      #设置数据库数目。默认的数据库是DB 0。可以通过SELECT <dbid>来选择一个数据库，dbid是[0,'databases'-1]的数字  databases 16    ################## 快照#################################  #  # 硬盘上保存数据:  #  #   save <seconds> <changes>  #  #   <seconds>和<changes>都满足时就会触发数据保存动作。  #  #  #   以下面的例子来说明：  #   过了900秒并且有1个key发生了改变 就会触发save动作  #   过了300秒并且有10个key发生了改变 就会触发save动作  #   过了60秒并且至少有10000个key发生了改变 也会触发save动作  #  #   注意：如果你不想让redis自动保存数据，那就把下面的配置注释掉！    save 900 1  save 300 10  save 60 10000      #存储数据时是否压缩数据。默认是yes。  rdbcompression yes    # 保存dump数据的文件名  dbfilename dump.rdb    # 工作目录.  #  # 数据会被持久化到这个目录下的‘dbfilename’指定的文件中。  #  #  # 注意，这里指定的必须是目录而不能是文件。  dir ./    ######## REPLICATION（复制，冗余）#################################    # Master-Slave replication. 使用slaveof把一个 Redis 实例设置成为另一个Redis server的从库（热备）. 注意： #配置只对当前slave有效。  # 因此可以把某个slave配置成使用不同的时间间隔来保存数据或者监听其他端口等等。  #命令格式：  # slaveof <masterip> <masterport>      #如果master有密码保护，则在slave与master进行数据同步之前需要进行密码校验，否则master会拒绝slave的请#求。  #  # masterauth <master-password>    #当slave丢失与master的连接时，或者slave仍然在于master进行数据同步时（还没有与master保持一致），#slave可以有两种方式来响应客户端请求：  #  # 1) 如果 slave-serve-stale-data 设置成 'yes' (the default) slave会仍然响应客户端请求,此时可能会有问题。  #  # 2) 如果 slave-serve-stale data设置成  'no'  slave会返回"SYNC with master in progress"这样的错误信息。 但 INFO 和SLAVEOF命令除外。  #  slave-serve-stale-data yes    ############### 安全 ###################################    # 需要客户端在执行任何命令之前指定 AUTH <PASSWORD>  #  # requirepass foobared    # 命令重命名.  #  #  # 例如:  #  # rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52  #  # 同样可以通过把一个命令重命名为空串来彻底kill掉这个命令，比如：  #  # rename-command CONFIG ""    #################### 限制 ####################################    # 设置最大连接数. 默认没有限制,  '0' 意味着不限制.  #  # maxclients 128      #最大可使用内存。如果超过，Redis会试图删除EXPIRE集合中的keys，具体做法是：Redis会试图释放即将过期的#keys，而保护还有很长生命周期的keys。  #  #如果这样还不行，Redis就会报错，但像GET之类的查询请求还是会得到响应。  #  #警告：如果你想把Redis视为一个真正的DB的话，那不要设置<maxmemory>,只有你只想把Redis作为cache或者  #有状态的server（'state' server)时才需要设置。  #  # maxmemory <bytes>    #内存清理策略：如果达到了maxmemory，你可以采取如下动作：  #  # volatile-lru -> 使用LRU算法来删除过期的set  # allkeys-lru -> 删除任何遵循LRU算法的key  # volatile-random ->随机地删除过期set中的key  # allkeys->random -> 随机地删除一个key  # volatile-ttl -> 删除最近即将过期的key（the nearest expire time (minor TTL)）  # noeviction -> 根本不过期，写操作直接报错  #  #  # 默认策略:  #  # maxmemory-policy volatile-lru    # 对于处理redis内存来说，LRU和minor TTL算法不是精确的，而是近似的（估计的）算法。所以我们会检查某些样本#来达到内存检查的目的。默认的样本数是3，你可以修改它。  #  # maxmemory-samples 3    ################# APPEND ONLY MODE ###############################    #默认情况下，Redis会异步的把数据保存到硬盘。如果你的应用场景允许因为系统崩溃等极端情况而导致最新数据丢失#的话，那这种做法已经很ok了。否则你应该打开‘append only’模式，开启这种模式后，Redis会在#appendonly.aof文件中添加每一个写操作，这个文件会在Redis启动时被读取来在内存中重新构建数据集。  #  #注意：如果你需要，你可以同时开启‘append only’模式和异步dumps模式（你需要注释掉上面的‘save’表达式来禁#止dumps），这种情况下，Redis重建数据集时会优先使用appendonly.aof而忽略dump.rdb  #  appendonly no    #  append only 文件名 (默认: "appendonly.aof")  # appendfilename appendonly.aof    # 调用fsync()函数通知操作系统立刻向硬盘写数据  #  # Redis支持3中模式:  #  # no:不fsync, 只是通知OS可以flush数据了，具体是否flush取决于OS.性能更好.  # always: 每次写入append only 日志文件后都会fsync . 性能差，但很安全.  # everysec: 没间隔1秒进行一次fsync. 折中.  #  # 默认是 "everysec"  # appendfsync always  appendfsync everysec  # appendfsync no    # 当AOF fsync策略被设置为always或者everysec并且后台保存进程（saving process)正在执行大量I/O操作时  # Redis可能会在fsync()调用上阻塞过长时间  #  no-appendfsync-on-rewrite no    # append only 文件的自动重写  # 当AOF 日志文件即将增长到指定百分比时，Redis可以通过调用BGREWRITEAOF 来自动重写append only文件。  #  # 它是这么干的：Redis会记住最近一次重写后的AOF 文件size。然后它会把这个size与当前size进行比较，如果当前# size比指定的百分比大，就会触发重写。同样，你需要指定AOF文件被重写的最小size，这对避免虽然百分比达到了# 但是实际上文件size还是很小（这种情况没有必要重写）却导致AOF文件重写的情况很有用。  #  #  # auto-aof-rewrite-percentage 设置为 0 可以关闭AOF重写功能    auto-aof-rewrite-percentage 100  auto-aof-rewrite-min-size 64mb    ################## SLOW LOG ###################################    # Redis slow log用来记录超过指定执行时间的查询。  #  # 你可以指定两个参数：一个是慢查询的阀值，单位是毫秒；另外一个是slow log的长度，相当于一个队列。    # 负数则关闭slow log，0则会导致每个命令都被记录  slowlog-log-slower-than 10000    # 不设置会消耗过多内存，所以还是要设置一下。可以使用SLOWLOG RESET命令来回收slow log使用的内存  slowlog-max-len 1024    ################ 虚拟内存 ###############################  #使用redis 就别用虚拟内存了，绝对不是一个好主意，加个机器吧，所以这里不翻译啦！！    ### WARNING! Virtual Memory is deprecated in**Redis 2.4**  ### The use of Virtual Memory is **strongly discouraged**.    # Virtual Memory allows Redis to work with datasets bigger than the actual  # amount of RAM needed to hold the whole dataset in memory.  # In order to do so very used keys are taken in memory while the other keys  # are swapped into a swap file, similarly to what operating systems do  # with memory pages.  #  # To enable VM just set 'vm-enabled' to yes, and set the following three  # VM parameters accordingly to your needs.    vm-enabled no  # vm-enabled yes    # This is the path of the Redis swap file. As you can guess, swap files  # can't be shared by different Redis instances, so make sure to use a swap  # file for every redis process you are running. Redis will complain if the  # swap file is already in use.  #  # The best kind of storage for the Redis swap file (that's accessed at random)  # is a Solid State Disk (SSD).  #  # \*\*\* WARNING \*\*\* if you are using a shared hosting the default of putting  # the swap file under /tmp is not secure. Create a dir with access granted  # only to Redis user and configure Redis to create the swap file there.  vm-swap-file /tmp/redis.swap    # vm-max-memory configures the VM to use at max the specified amount of  # RAM. Everything that deos not fit will be swapped on disk \*if\* possible, that  # is, if there is still enough contiguous space in the swap file.  #  # With vm-max-memory 0 the system will swap everything it can. Not a good  # default, just specify the max amount of RAM you can in bytes, but it's  # better to leave some margin. For instance specify an amount of RAM  # that's more or less between 60 and 80% of your free RAM.  vm-max-memory 0    # Redis swap files is split into pages. An object can be saved using multiple  # contiguous pages, but pages can't be shared between different objects.  # So if your page is too big, small objects swapped out on disk will waste  # a lot of space. If you page is too small, there is less space in the swap  # file (assuming you configured the same number of total swap file pages).  #  # If you use a lot of small objects, use a page size of 64 or 32 bytes.  # If you use a lot of big objects, use a bigger page size.  # If unsure, use the default :)  vm-page-size 32    # Number of total memory pages in the swap file.  # Given that the page table (a bitmap of free/used pages) is taken in memory,  # every 8 pages on disk will consume 1 byte of RAM.  #  # The total swap size is vm-page-size \* vm-pages  #  # With the default of 32-bytes memory pages and 134217728 pages Redis will  # use a 4 GB swap file, that will use 16 MB of RAM for the page table.  #  # It's better to use the smallest acceptable value for your application,  # but the default is large in order to work in most conditions.  vm-pages 134217728    # Max number of VM I/O threads running at the same time.  # This threads are used to read/write data from/to swap file, since they  # also encode and decode objects from disk to memory or the reverse, a bigger  # number of threads can help with big objects even if they can't help with  # I/O itself as the physical device may not be able to couple with many  # reads/writes operations at the same time.  #  # The special value of 0 turn off threaded I/O and enables the blocking  # Virtual Memory implementation.  vm-max-threads 4    ################高级配置###############################    # Hashes are encoded in a special way (much more memory efficient) when they  # have at max a given numer of elements, and the biggest element does not  # exceed a given threshold. You can configure this limits with the following  # configuration directives.  hash-max-zipmap-entries 512  hash-max-zipmap-value 64    # Similarly to hashes, small lists are also encoded in a special way in order  # to save a lot of space. The special representation is only used when  # you are under the following limits:  list-max-ziplist-entries 512  list-max-ziplist-value 64    # Sets have a special encoding in just one case: when a set is composed  # of just strings that happens to be integers in radix 10 in the range  # of 64 bit signed integers.  # The following configuration setting sets the limit in the size of the  # set in order to use this special memory saving encoding.  set-max-intset-entries 512    # Similarly to hashes and lists, sorted sets are also specially encoded in  # order to save a lot of space. This encoding is only used when the length and  # elements of a sorted set are below the following limits:  zset-max-ziplist-entries 128  zset-max-ziplist-value 64    # Active rehashing uses 1 millisecond every 100 milliseconds of CPU time in  # order to help rehashing the main Redis hash table (the one mapping top-level  # keys to values). The hash table implementation redis uses (see dict.c)  # performs a lazy rehashing: the more operation you run into an hash table  # that is rhashing, the more rehashing "steps" are performed, so if the  # server is idle the rehashing is never complete and some more memory is used  # by the hash table.  #  # The default is to use this millisecond 10 times every second in order to  # active rehashing the main dictionaries, freeing memory when possible.  #  # If unsure:  # use "activerehashing no" if you have hard latency requirements and it is  # not a good thing in your environment that Redis can reply form time to time  # to queries with 2 milliseconds delay.  #  # use "activerehashing yes" if you don't have such hard requirements but  # want to free memory asap when possible.  activerehashing yes    ################## INCLUDES ###################################    # Include one or more other config files here.  This is useful if you  # have a standard template that goes to all redis server but also need  # to customize a few per-server settings.  Include files can include  # other files, so use this wisely.  #  # include /path/to/local.conf  # include /path/to/other.conf |

#### redis client

##### jedis

**说明：支持java开发访问redis的客户端**

官网：<https://github.com/xetorthio/jedis>

Jar下载地址：<http://github.com/xetorthio/jedis/downloads>

Api参见：

<http://www.jarvana.com/jarvana/view/redis/clients/jedis/2.0.0/jedis-2.0.0-javadoc.jar!/index.html>

|  |
| --- |
| **连接server** |
| String serverIp = "192.168.102.128";  **int** serverPort = 6379;  Jedis jedis = **new** Jedis(serverIp, serverPort); |

|  |
| --- |
| 操作数据: |
| /\*\*  \* 操作String  \*/  **public** **static** **void** testStr() {  String key = "listName";  String value = "value";  // 存储一个key 对应的value字符串  *jedis*.set(key, value);  // 获取key对应值  *jedis*.get(key);  }  /\*\*  \* 操作队列  \*/  **public** **static** **void** testList() {  String listName = "listName";  String value = "value";  **int** updateIndex = 0;  // 从队列头开始添加一个元素  *jedis*.lpush(listName, value);  // 从队列头开始移除一个元素  *jedis*.lpop(listName);  // 从队列尾开始添加一个元素  *jedis*.rpush(listName, value);  // 从队列尾开始移除一个元素  *jedis*.rpop(listName);  // 修改队列从头开始指定位置的值  *jedis*.lset(listName, updateIndex, value);  // 获取队列指定位置值  *jedis*.lindex(listName, updateIndex);  // 获取队列长度  *jedis*.llen(listName);  }  /\*\*  \* 操作hash  \*/  **public** **static** **void** testHash() {  String hashName = "hashName";  String key = "key";  String value = "value";  Map<String, String> map = **new** HashMap<String, String>();  // 添加hash 并添加键值对  *jedis*.hset(hashName, key, value);  // 添加hash 并添加一个map  *jedis*.hmset(key, map);  // 获取hash 指定key对应值  *jedis*.hget(hashName, key);  // 获取hash key的数量  *jedis*.hlen(hashName);  // 获取hash 所有key  *jedis*.hkeys(hashName);  } |

### Redis深究

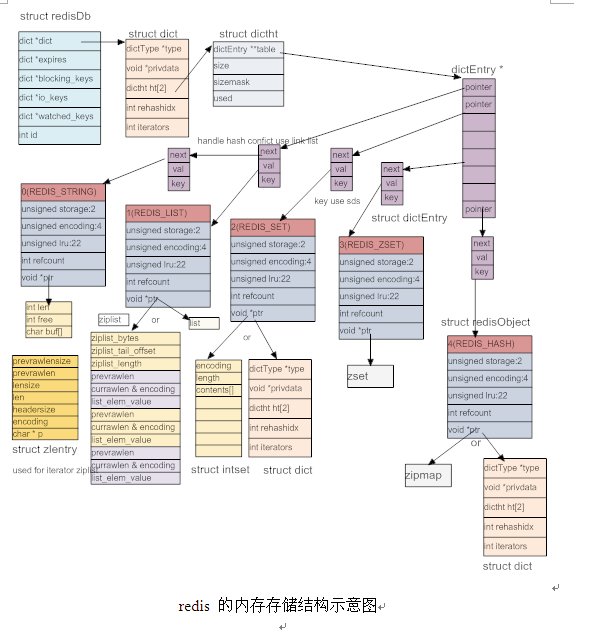
#### 数据结构

##### 1 Redis 内存存储结构

本文是基于 Redis-v2.2.4 版本进行分析.

###### 1.1 Redis 内存存储总体结构

Redis 是支持多key-value数据库(表)的,并用 RedisDb 来表示一个key-value数据库(表). redisServer 中有一个 redisDb \*db; 成员变量, RedisServer 在初始化时,会根据配置文件的 db 数量来创建一个 redisDb 数组. 客户端在连接后,通过 SELECT 指令来选择一个 reidsDb,如果不指定,则缺省是redisDb数组的第1个(即下标是 0 ) redisDb. 一个客户端在选择 redisDb 后,其后续操作都是在此 redisDb 上进行的. 下面会详细介绍一下 redisDb 的内存结构.

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011710.jpg)

redis 的内存存储结构示意图

**redisDb 的定义:**

|  |  |  |
| --- | --- | --- |
|  | |  | | --- | | typedef struct redisDb    {    dict \*dict; /\* The keyspace for this DB \*/    dict \*expires; /\* Timeout of keys with a timeout set \*/    dict \*blocking\_keys; /\* Keys with clients waiting for data (BLPOP) \*/    dict \*io\_keys; /\* Keys with clients waiting for VM I/O \*/    dict \*watched\_keys; /\* WATCHED keys for MULTI/EXEC CAS \*/    int id;    } redisDb;    struct | |

**redisDb 中 ,dict 成员是与实际存储数据相关的. dict 的定义如下:**

|  |
| --- |
| t typedef struct dictEntry    {    void \*key;    void \*val;    struct dictEntry \*next;    } dictEntry;    typedef struct dictType    {    unsigned int (\*hashFunction)(const void \*key);    void \*(\*keyDup)(void \*privdata, const void \*key);    void \*(\*valDup)(void \*privdata, const void \*obj);    int (\*keyCompare)(void \*privdata, const void \*key1, const void \*key2);    void (\*keyDestructor)(void \*privdata, void \*key);    void (\*valDestructor)(void \*privdata, void \*obj);    } dictType;    /\* This is our hash table structure. Every dictionary has two of this as we    \* implement incremental rehashing, for the old to the new table. \*/    typedef struct dictht    {    dictEntry \*\*table;    unsigned long size;    unsigned long sizemask;    unsigned long used;    } dictht;    typedef struct dict    {    dictType \*type;    void \*privdata;    dictht ht[2];    int rehashidx; /\* rehashing not in progress if rehashidx == -1 \*/    int iterators; /\* number of iterators currently running \*/    } dict; |

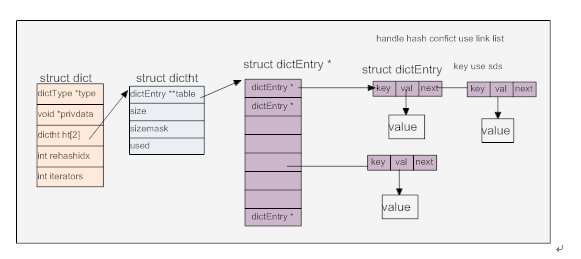
**dict 是主要是由 struct dictht 的 哈唏表构成的, 之所以定义成长度为2的( dictht ht[2] ) 哈唏表数组,是因为 redis 采用渐进的 rehash,即当需要 rehash 时,每次像 hset,hget 等操作前,先执行N 步 rehash. 这样就把原来一次性的 rehash过程拆散到进行, 防止一次性 rehash 期间 redis 服务能力大幅下降. 这种渐进的 rehash 需要一个额外的 struct dictht 结构来保存.**

**struct dictht 主要是由一个 struct dictEntry 指针数组组成的, hash 表的冲突是通过链表法来解决的.**

**struct dictEntry 中的 key 指针指向用 sds 类型表示的 key 字符串, val 指针指向一个 struct redisObject 结构体, 其定义如下:**

|  |
| --- |
| typedef struct dictEntry    {    void \*key;    void \*val;    struct dictEntry \*next;    } dictEntry;    typedef struct dictType    {    unsigned int (\*hashFunction)(const void \*key);    void \*(\*keyDup)(void \*privdata, const void \*key);    void \*(\*valDup)(void \*privdata, const void \*obj);    int (\*keyCompare)(void \*privdata, const void \*key1, const void \*key2);    void (\*keyDestructor)(void \*privdata, void \*key);    void (\*valDestructor)(void \*privdata, void \*obj);    } dictType;    /\* This is our hash table structure. Every dictionary has two of this as we    \* implement incremental rehashing, for the old to the new table. \*/    typedef struct dictht    {    dictEntry \*\*table;    unsigned long size;    unsigned long sizemask;    unsigned long used;    } dictht;    typedef struct dict    {    dictType \*type;    void \*privdata;    dictht ht[2];    int rehashidx; /\* rehashing not in progress if rehashidx == -1 \*/    int iterators; /\* number of iterators currently running \*/    } dict; |

###### 1.2 Dict 结构

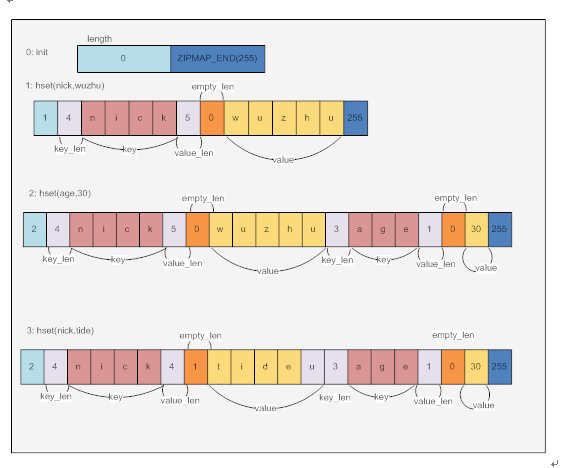
[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011720.jpg)

Dict 结构在<1.1Redis 内存存储结构>; 已经描述过了,这里不再赘述.

###### 1.3 zipmap 结构

如果redisObject的type 成员值是 REDIS\_HASH 类型的,则当该hash 的 entry 小于配置值: hash-max-zipmap-entries 或者value字符串的长度小于 hash-max-zipmap-value, 则可以编码成 REDIS\_ENCODING\_ZIPMAP 类型存储,以节约内存. 否则采用 Dict 来存储.

zipmap 其实质是用一个字符串数组来依次保存key和value,查询时是依次遍列每个 key-value 对,直到查到为止. 其结构示意图如下:

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011730.jpg)

为了节约内存,这里使用了一些小技巧来保存 key 和 value 的长度. 如果 key 或 value 的长度小于ZIPMAP\_BIGLEN(254),则用一个字节来表示,如果大于ZIPMAP\_BIGLEN(254),则用5个字节保存,第一个字节为保存ZIPMAP\_BIGLEN(254),后面4个字节保存 key或value 的长度.

1. 初始化时只有2个字节,第1个字节表示 zipmap 保存的 key-value 对的个数(如果key-value 对的个数超过 254,则一直用254来表示, zipmap 中实际保存的 key-value 对个数可以通过 zipmapLen() 函数计算得到).
2. hset(nick,wuzhu) 后,
   * 第1个字节保存key-value 对(即 zipmap 的entry 数量)的数量1
   * 第2个字节保存key\_len 值 4
   * 第3~6 保存 key “nick”
   * 第 7 字节保存 value\_len 值 5
   * 第 8 字节保存空闭的字节数 0 (当 该 key 的值被重置时,其新值的长度与旧值的长度不一定相等,如果新值长度比旧值的长度大,则 realloc 扩大内存; 如果新值长度比旧值的长度小,且相差大于 4 bytes ,则 realloc 缩小内存,如果相差小于 4,则将值往前移,并用 empty\_len 保存空闲的byte 数)
   * 第 9~13字节保存 value 值 “wuzhu”
3. hset(age,30)

插入 key-value 对 (“age”,30)

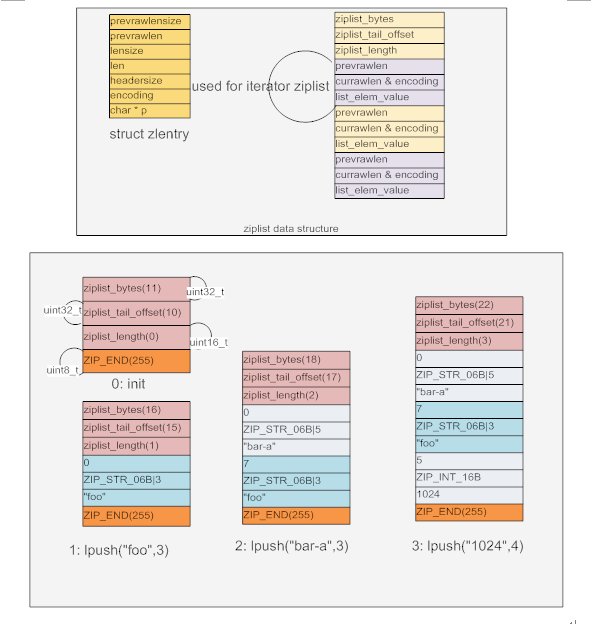
1. hset(nick,tide)

插入 key-value 对 (“nick”,”tide”), 后可以看到 empty\_len 为1 ,

###### 1.4 ziplist 结构

如果redisObject的type 成员值是 REDIS\_LIST 类型的,则当该list 的 elem数小于配置值: hash-max-ziplist-entries 或者elem\_value字符串的长度小于 hash-max-ziplist-value, 则可以编码成 REDIS\_ENCODING\_ZIPLIST 类型存储,以节约内存. 否则采用 list 来存储.

ziplist 其实质是用一个字符串数组形式的双向链表. 其结构示意图如下:

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011740.jpg)

1. ziplist header由3个字段组成:
   * ziplist\_bytes: 用一个uint32\_t 来保存, 构成 ziplist 的字符串数组的总长度,包括 ziplist header,
   * ziplist\_tail\_offset: 用一个uint32\_t 来保存,记录 ziplist 的尾部偏移位置.
   * ziplist\_length: 用一个 uint16\_t 来保存,记录 ziplist 中 elem 的个数
2. ziplist node 也由 3 部分组成:
   * prevrawlen: 保存上一个 ziplist node 的占用的字节数,包括: 保存prevarwlen,currawlen 的字节数和elem value 的字节数.
   * currawlen&encoding: 当前elem value 的raw 形式存款所需的字节数及在ziplist 中保存时的编码方式(例如,值可以转换成整数,如示意图中的”1024”, raw\_len 是 4 字节,但在 ziplist 保存时转换成 uint16\_t 来保存,占2 个字节).
   * (编码后的)value

可以通过 prevrawlen 和 currawlen&encoding 来遍列 ziplist.

ziplist 还能到一些小技巧来节约内存.

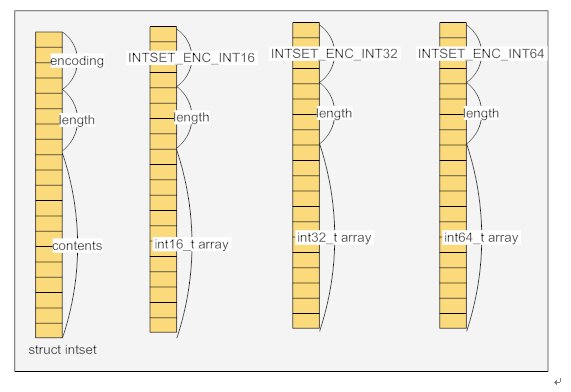
* len 的存储: 如果 len 小于 ZIP\_BIGLEN(254),则用一个字节来保存; 否则需要 5 个字节来保存,第 1 个字节存 ZIP\_BIGLEN,作为标识符.
* value 的存储: 如果 value 是数字类型的,则根据其值的范围转换成 ZIP\_INT\_16B, ZIP\_INT\_32B或ZIP\_INT\_64B 来保存,否则用 raw 形式保存.

###### 1.5 adlist 结构

|  |
| --- |
| typedef struct listNode  {    struct listNode \*prev;    struct listNode \*next;    void \*value;    } listNode;    typedef struct listIter    {    listNode \*next;    int direction;    } listIter;    typedef struct list    {    listNode \*head;    listNode \*tail;    void \*(\*dup)(void \*ptr);    void (\*free)(void \*ptr);    int (\*match)(void \*ptr, void \*key);    unsigned int len;    } list; |

常见的双向链表,不作分析.

###### 1.6 intset 结构

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011750.jpg)

intset 是用一个有序的整数数组来实现集合(set). struct intset 的定义如下:

|  |
| --- |
| typedef struct intset    {    uint32\_t encoding;    uint32\_t length;    int8\_t contents[];    } intset; |

* encoding: 来标识数组是 int16\_t 类型, int32\_t 类型还是 int64\_t 类型的数组. 至于怎么先择是那种类型的数组,是根据其保存的值的取值范围来决定的,初始化时是 int16\_t, 根据 set 中的最大值在 [INT16\_MIN, INT16\_MAX] , [INT32\_MIN, INT32\_MAX], [INT64\_MIN, INT64\_MAX]的那个取值范围来动态确定整个数组的类型. 例如set一开始是 int16\_t 类型,当一个取值范围在 [INT32\_MIN, INT32\_MAX]的值加入到 set 时,则将保存 set 的数组升级成 int32\_t 的数组.
* length: 表示 set 中值的个数
* contents: 指向整数数组的指针

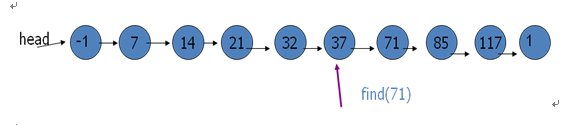
###### 1.7 zset 结构

首先，介绍一下 skip list 的概念，然后再分析 zset 的实现.

1.7.1 Skip List 介绍

**1.7.1.1 有序链表**

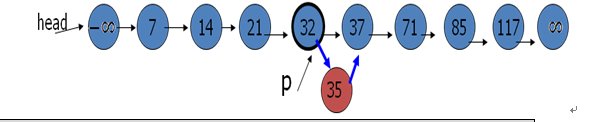
1) Searching a key in a Sorted linked list

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011760.jpg)

|  |
| --- |
| //Searching an element <em>x</em>    cell \*p =head ;    while (p->next->key < x )  p=p->next ;    return p ; |

Note: we return the element proceeding either the element containing x, or the smallest element with a key larger than x(if x does not exists)

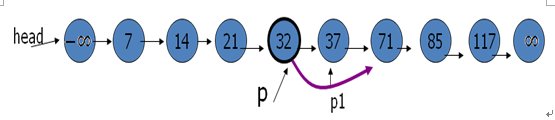
2) inserting a key into a Sorted linked list

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011770.jpg)

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
| //To insert 35 -    p=find(35);    CELL \*p1 = (CELL \*) malloc(sizeof(CELL));    p1->key=35;    p1->next = p->next ;    p->next  = p1 ; |

3) deleteing a key from a sorted list

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011780.jpg)

|  |
| --- |
| //To delete 37 -    p=find(37);    CELL \*p1 =p->next;    p->next = p1->next ;    free(p1); |

**1.7.1.2 SkipList(跳跃表)定义**

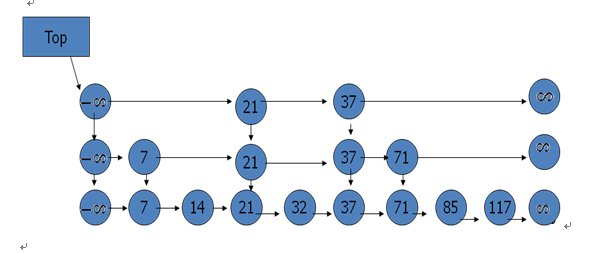
SKIP LIST : A data structure for maintaing a set of keys in a sorted order.

Consists of several **levels.**

All keys appear in level 1

Each level is a sorted list.

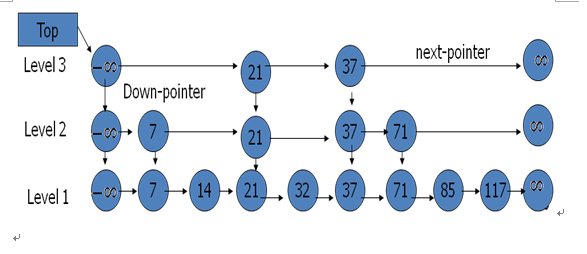
If key x appears in level i, then it also appears in all levels below i

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011790.jpg)

An element in levelipoints (via down pointer) to the element with the same key in the level below.

In each level the keys and appear. (In our implementation, INT\_MIN and INT\_MAX

Top points to the smallest element in the highest level.

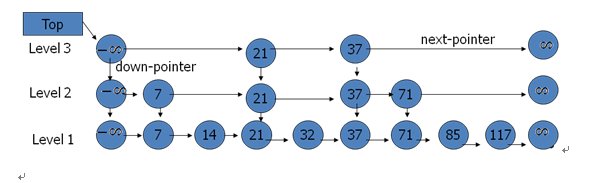
[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011800.jpg)

**1.7.1.3 SkipList(跳跃表)操作**

1) An empty SkipList

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011810.jpg)

2) Finding an element with key x

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011820.jpg)

|  |
| --- |
| p=top    While(1)    {    while (p->next->key < x ) p=p->next;    If (p->down == NULL ) return p->next    p=p->down ;    } |

Observe that we return x, if exists, or succ(x) if x is not in the SkipList

3) Inserting new element X

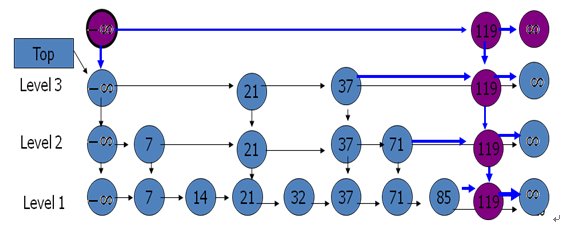
Determine **k** the number of levels in which x participates (explained later)

Do find(x), and insert x to the appropriate places in the lowest **k** levels. (after the elements at which the search path turns down or terminates)

Example – inserting 119. **k**=2

If **k**is larger than the current number of levels, add new levels (and update top)

Example – inser(119) when k=4

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011840.jpg)

Determining k

k – the number of levels at which an element x participate.

Use a random function OurRnd() — returns 1 or 0 (True/False) with equal probability.

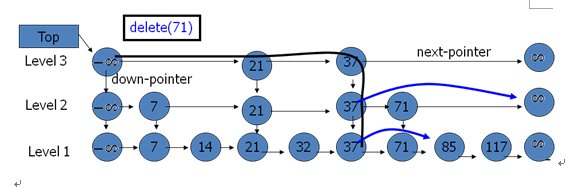
k=1 ;

While( OurRnd() ) k++ ;

Deleteing a key x

Find x in all the levels it participates, and delete it using the standard ‘delete from a linked list’ method.

If one or more of the upper levels are empty, remove them (and update top)

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011850.jpg)

Facts about SkipList

The expected number of levels is O( log n )

(here n is the numer of elements)

The expected time for insert/delete/find is O( log n )

The expected size (number of cells) is O(n)

1.7.2 redis SkipList 实现

/\* ZSETs use a specialized version of Skiplists \*/

|  |
| --- |
| typedef struct zskiplistNode    {    robj \*obj;    double score;    struct zskiplistNode \*backward;    struct zskiplistLevel    {    struct zskiplistNode \*forward;    unsigned int span;    } level[];    } zskiplistNode;    typedef struct zskiplist    {    struct zskiplistNode \*header, \*tail;    unsigned long length;    int level;    } zskiplist;    typedef struct zset    {    dict \*dict;    zskiplist \*zsl;    } zset; |

zset 的实现用到了2个数据结构: hash\_table 和 skip list (跳跃表),其中 hash table 是使用 redis 的 dict 来实现的,主要是为了保证查询效率为 O(1),而 skip list (跳跃表) 是用来保证元素有序并能够保证 INSERT 和 REMOVE 操作是 O(logn)的复杂度。

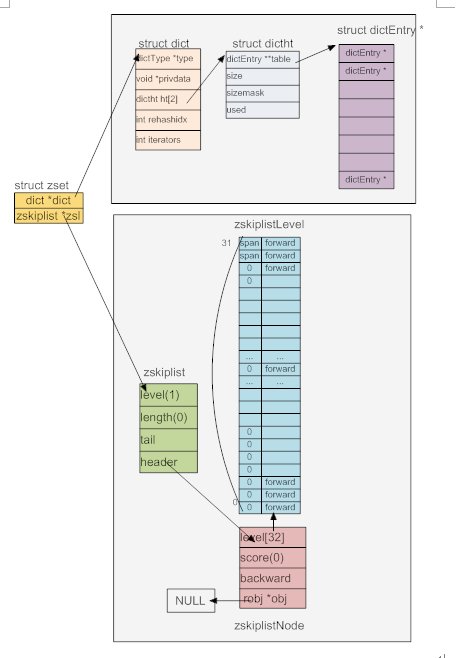
1) zset初始化状态

createZsetObject函数来创建并初始化一个 zset

|  |
| --- |
| robj \*createZsetObject(void)    {    zset \*zs = zmalloc(sizeof(\*zs));    robj \*o;    zs->dict = dictCreate(&zsetDictType,NULL);    zs->zsl = zslCreate();    o = createObject(REDIS\_ZSET,zs);    o->encoding = REDIS\_ENCODING\_SKIPLIST;    return o;    } |

zslCreate()函数用来创建并初如化一个 skiplist。 其中，skiplist 的 level 最大值为 ZSKIPLIST\_MAXLEVEL=32 层。

|  |
| --- |
| zskiplist \*zslCreate(void)    {    int j;    zskiplist \*zsl;    zsl = zmalloc(sizeof(\*zsl));    zsl->level = 1;    zsl->length = 0;    zsl->header = zslCreateNode(ZSKIPLIST\_MAXLEVEL,0,NULL);    for (j = 0; j < ZSKIPLIST\_MAXLEVEL; j++) {    zsl->header->level[j].forward = NULL;    zsl->header->level[j].span = 0;    }    zsl->header->backward = NULL;    zsl->tail = NULL;    return zsl;    } |

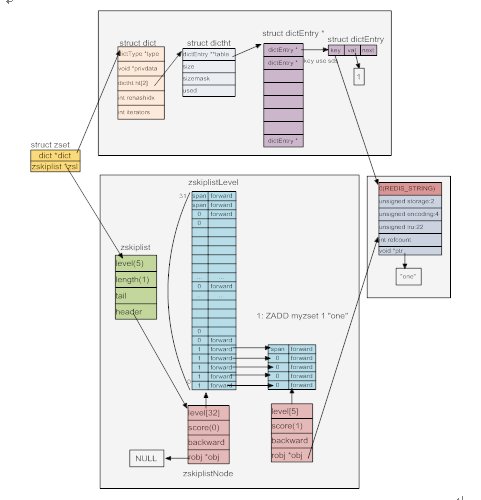
[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011860.jpg)

2) ZADD myzset 1 “one”

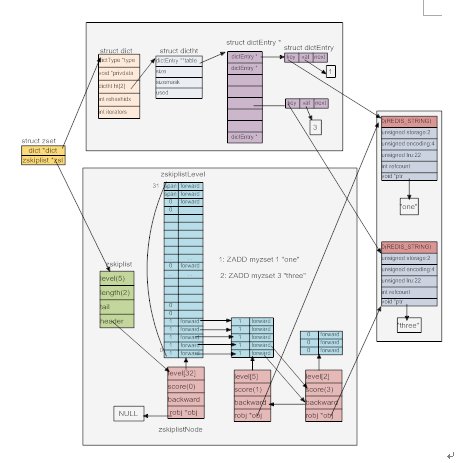
ZADD 命令格式：

ZADD key score member

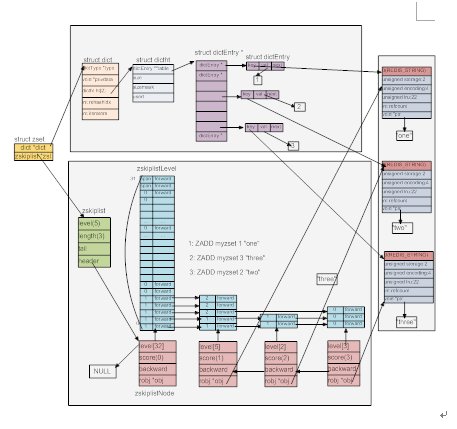
1. 根据 key 从 redisDb 进行查询，返回 zset 对象。
2. 以 member 作为 key,score 作为 value ，向 zset的 dict 进行中插入;
3. 如果返回成功，表明 member 没有在 dict 中出现过，直接向 skiplist 进行插入。
4. 如果步骤返回失败，表明以 member 已经在 dict中出现过，则需要先从 skiplist 中删除，然后以现在的 score 值重新插入。

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011870.jpg)

3) ZADD myzset 3 “three”

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011880.jpg)

4) ZADD myzset 2 “two”

[](http://www.searchtb.com/wp-content/uploads/2011/04/Image011890.jpg)

### 其它

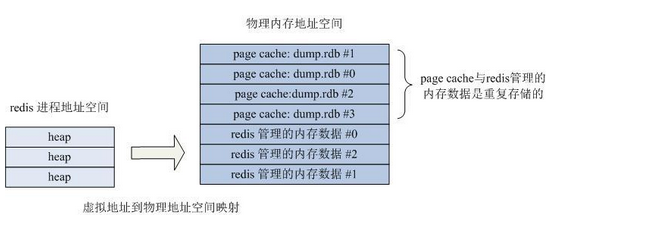
#### 问题

##### Redis持久化磁盘IO方式及其带来的问题

有Redis线上运维经验的人会发现Redis在物理内存使用比较多，但还没有超过实际物理内存总容量时就会发生不稳定甚至崩溃的问题，有人认为是基于快照方式持久化的fork系统调用造成内存占用加倍而导致的，这种观点是不准确的，因为fork 调用的copy-on-write机制是基于操作系统页这个单位的，也就是只有有写入的脏页会被复制，但是一般你的系统不会在短时间内所有的页都发生了写入而导致复制，那么是什么原因导致Redis崩溃的呢？

答案是Redis的持久化使用了Buffer IO造成的，所谓Buffer IO是指Redis对持久化文件的写入和读取操作都会使用物理内存的Page Cache,而大多数数据库系统会使用Direct IO来绕过这层Page Cache并自行维护一个数据的Cache，而当Redis的持久化文件过大(尤其是快照文件)，并对其进行读写时，磁盘文件中的数据都会被加载到物理内存中作为操作系统对该文件的一层Cache,而这层Cache的数据与Redis内存中管理的数据实际是重复存储的，虽然内核在物理内存紧张时会做Page Cache的剔除工作，但内核很可能认为某块Page Cache更重要，而让你的进程开始Swap ,这时你的系统就会开始出现不稳定或者崩溃了。我们的经验是当你的Redis物理内存使用超过内存总容量的3/5时就会开始比较危险了。

下图是Redis在读取或者写入快照文件dump.rdb后的内存数据图：



#### 总结

1. 根据业务需要选择合适的数据类型，并为不同的应用场景设置相应的紧凑存储参数。
2. 当业务场景不需要数据持久化时，关闭所有的持久化方式可以获得最佳的性能以及最大的内存使用量。
3. 如果需要使用持久化，根据是否可以容忍重启丢失部分数据在快照方式与语句追加方式之间选择其一，不要使用虚拟内存以及diskstore方式。
4. 不要让你的Redis所在机器物理内存使用超过实际内存总量的3/5。