深入底层C源码讲透Redis核心设计原理

1.Redis K-V 底层设计原理

2.Redis 渐进式rehash及动态扩容机制

3.Redis核心编码结构精讲

4.亿级用户日活统计BitMap实战及源码分析

授课讲师:郭嘉

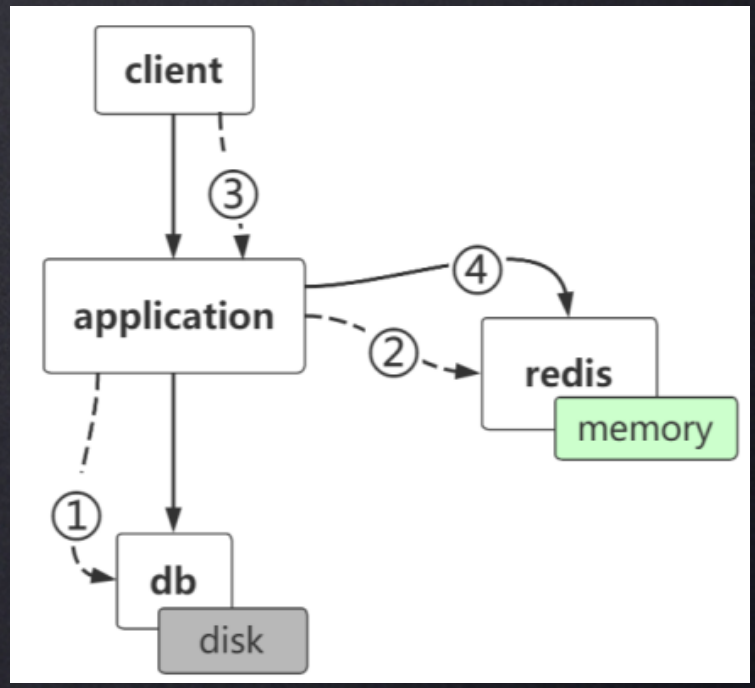
Redis 基本特性

1. 非关系型的键值对数据库，可以根据键以O(1) 的时间复杂度取出或插入关联值
2. Redis 的数据是存在内存中的
3. 键值对中键的类型可以是字符串，整型，浮点型等，且键是唯一的
4. 键值对中的值类型可以是string，hash，list，set，sorted set 等
5. Redis 内置了复制，磁盘持久化，LUA脚本，事务，SSL，ACLs，客户端缓存，客户端代理等功能
6. 通过Redis哨兵和Redis Cluster 模式提供高可用性

Redis应用场景

缓存

存储器	硬件介质	单位成本(美元/MB)	随机访问延时	说明
L1 Cache	SRAM	7	1ns	
L2 Cache	SRAM	7	4ns	访问延时15x L1 Cache
Memory	DRAM	0.015	100ns	访问延时15X SRAM，价格1/40 SRAM
Disk	SSD(NAND)	0.0004	150μs	访问延时 1500X DRAM，价格 1/40 DRAM
Disk	HDD	0.00004	10ms	访问延时 70X SSD，价格 1/10 SSD



1s =1000 ms
1ms=1000 us
1us =1000 ns

Redis应用场景

计数器
可以对 String 进行自增自减运算，从而实现计数器功能。Redis 这种内存型数据库的读写性能非常高，很适合存储频繁读写的计数量。

分布式ID生成
利用自增特性，一次请求一个大一点的步长如 incr 2000 ,缓存在本地使用，用完再请求。

海量数据统计
位图（bitmap）:存储是否参过某次活动，是否已读谋篇文章，用户是否为会员， 日活统计。

会话缓存
可以使用 Redis 来统一存储多台应用服务器的会话信息。当应用服务器不再存储用户的会话信息，也就不再具有状态，一个用户可以请求任意一个应用服务器，从而更容易实现高可用性以及可伸缩性。

分布式队列/阻塞队列
List 是一个双向链表，可以通过 lpush/rpush 和 rpop/lpop 写入和读取消息。可以通过使用 brpop/blpop 来实现阻塞队列。

Redis应用场景

分布式锁实现
在分布式场景下，无法使用基于进程的锁来对多个节点上的进程进行同步。可以使用 Redis 自带的 SETNX 命令实现分布式锁。

热点数据存储
最新评论，最新文章列表，使用list 存储,ltrim取出热点数据，删除老数据。

社交类需求
Set 可以实现交集，从而实现共同好友等功能，Set通过求差集，可以进行好友推荐，文章推荐。

排行榜
sorted_set可以实现有序性操作，从而实现排行榜等功能。

延迟队列
使用sorted_set，使用 【当前时间戳 + 需要延迟的时长】做score, 消息内容作为元素,调用zadd来生产消息，消费者使用zrangbyscore获取当前时间之前的数据做轮询处理。消费完再删除任务 rem key member

String

String 常用API

/> help @string

/> SET/GET
/> SETNX
/> GETRANGE/SETRANGE

/> INCR/INCRBY/DECR/DECRBY

/> GETBIT/SETBIT/BITOPS/BITCOUNT

/> MGET/MSET

String

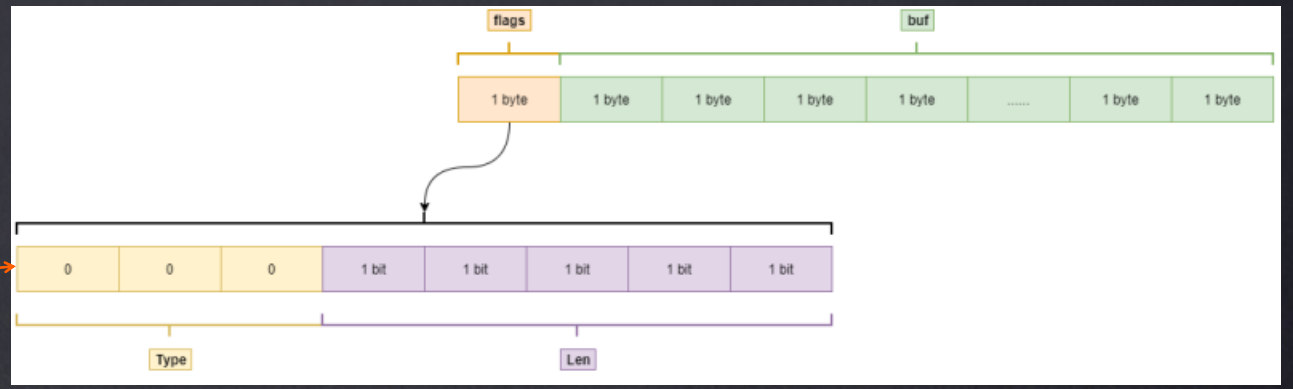
数据结构

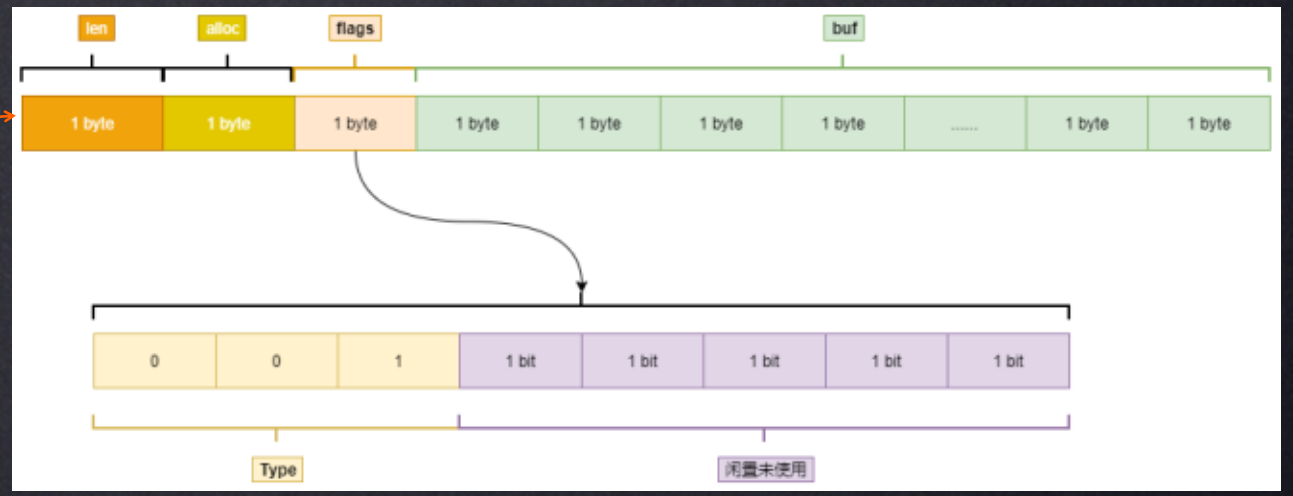
redis 3.2 以前

```
struct sdshdr {
    int len;
    int free;
    char buf[];
};
```

redis 3.2 后

```
typedef char *sds;

struct __attribute__ ((__packed__)) sdshdr5 {
    unsigned char flags; /* 3 lsb of type, and 5 msb of string
length */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr8 {
    uint8_t len; /* used */
    uint8_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
struct __attribute__ ((__packed__)) sdshdr16 {
    uint16_t len; /* used */
    uint16_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};
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[];
};
struct __attribute__ ((__packed__)) sdshdr64 {
.....
```






```
#define SDS_TYPE_5  0
#define SDS_TYPE_8  1
#define SDS_TYPE_16 2
#define SDS_TYPE_32 3
#define SDS_TYPE_64 4
```

```
static inline char sdsReqType(size_t string_size) {
    if (string_size < 32)
        return SDS_TYPE_5;
    if (string_size < 0xff) //2^8 -1
        return SDS_TYPE_8;
    if (string_size < 0xffff) // 2^16 -1
        return SDS_TYPE_16;
    if (string_size < 0xffffffff)  // 2^32 -1
        return SDS_TYPE_32;
    return SDS_TYPE_64;
}
```

RedisDb 数据结构

```
typedef struct redisDb {
    dict *dict;
    dict *expires;
    dict *blocking_keys;
    dict *ready_keys;
    dict *watched_keys;
    int id;
    long long avg_ttl;
    unsigned long expires_cursor;
    list *defrag_later;
} redisDb;
```

```
typedef struct dict {
    dictType *type;
    void *privdata;
    dictht ht[2];
    long rehashidx;
    unsigned long iterators;
    } dict;
```

```
typedef struct dictEntry {
    void *key;
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    } v;
    struct dictEntry *next;
} dictEntry;
```

```
typedef struct dictht {
    dictEntry **table;
    unsigned long size;
    unsigned long
sizemask;
    unsigned long used;
} dictht;
```

```
typedef struct
redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned
lru:LRU_BITS;
    int refcount;
    void *ptr;
} robj;
```

List

List常用API

/> help @list

LPUSH key element [element ...]
RPOP key
RPUSH key element [element ...]
LPOP key
BLPOP key [key ...] timeout
BRPOP key [key ...] timeout
BRPOPLPUSH source destination timeout
RPOPLPUSH source destination
LINDEX key index
LLEN key
LINSERT key BEFORE|AFTER pivot element
LRANGE key start stop
LREM key count element
LSET key index element
LTRIM key start stop

List

List是一个有序(按加入的时序排序)的数据结构，Redis采用quicklist（双端链表） 和 ziplist 作为List的底层实现。

可以通过设置每个ziplist的最大容量，quicklist的数据压缩范围，提升数据存取效率

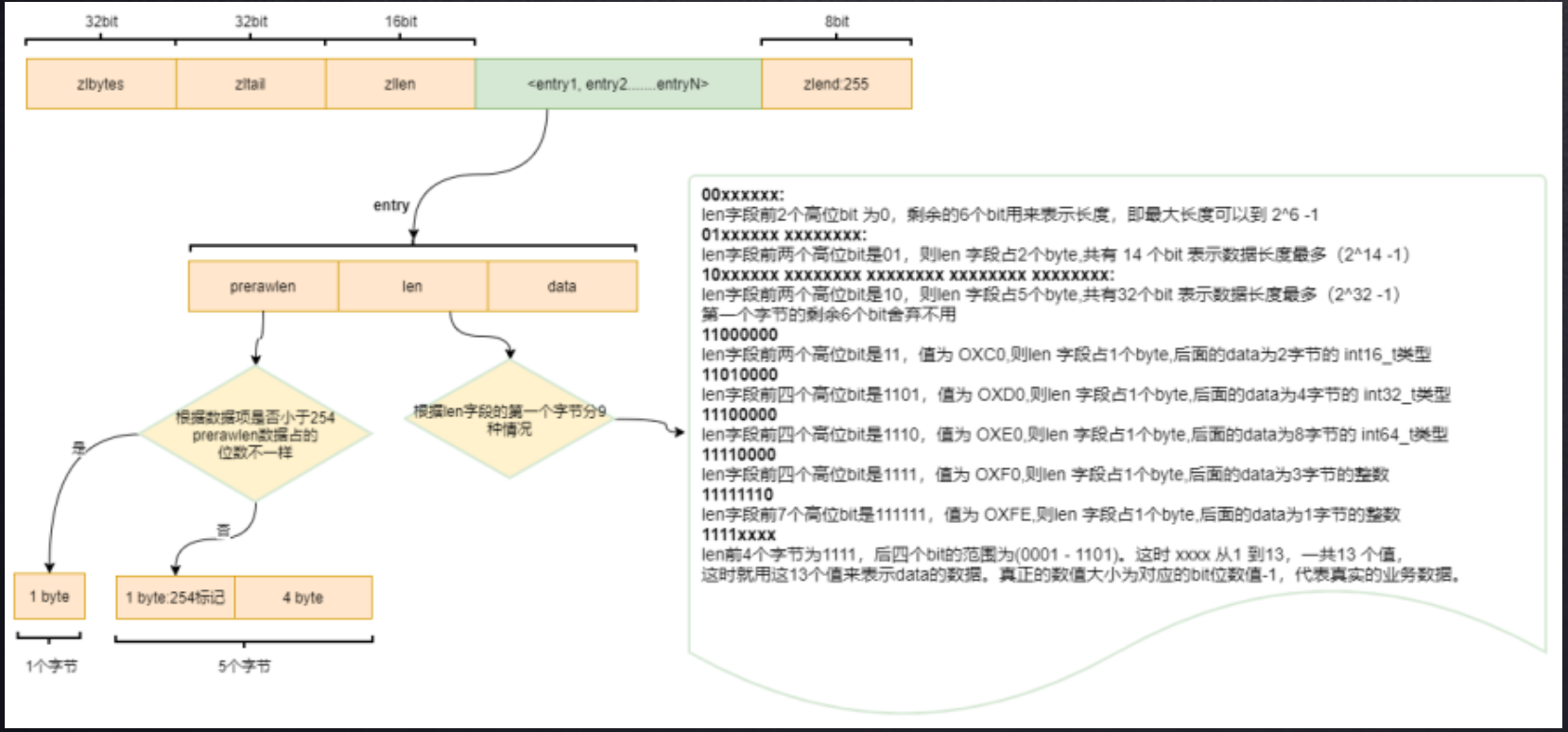
```
list-max-ziplist-size  -2        //  单个ziplist节点最大能存储  8kb  ,超过则进行分裂,将数据存储在新的ziplist节点中
list-compress-depth  1           //  0 代表所有节点，都不进行压缩，1， 代表从头节点往后走一个，尾节点往前走一个不用压缩，其他的全部压缩，2，3，4 ... 以此类推
```

ziplist

```
robj *createZiplistObject(void) {
    unsigned char *zl = ziplistNew();
    robj *o = createObject(OBJ_LIST,zl);
    o->encoding = OBJ_ENCODING_ZIPLIST;
    return o;
}
```

```
unsigned char *ziplistNew(void) {
    unsigned int bytes =
ZIPLIST_HEADER_SIZE+ZIPLIST_END_SIZE
;
    unsigned char *zl = zmalloc(bytes);
    ZIPLIST_BYTES(zl) = intrev32ifbe(bytes);
    ZIPLIST_TAIL_OFFSET(zl) =
intrev32ifbe(ZIPLIST_HEADER_SIZE);
    ZIPLIST_LENGTH(zl) = 0;
    zl[bytes-1] = ZIP_END;
    return zl;
}
```

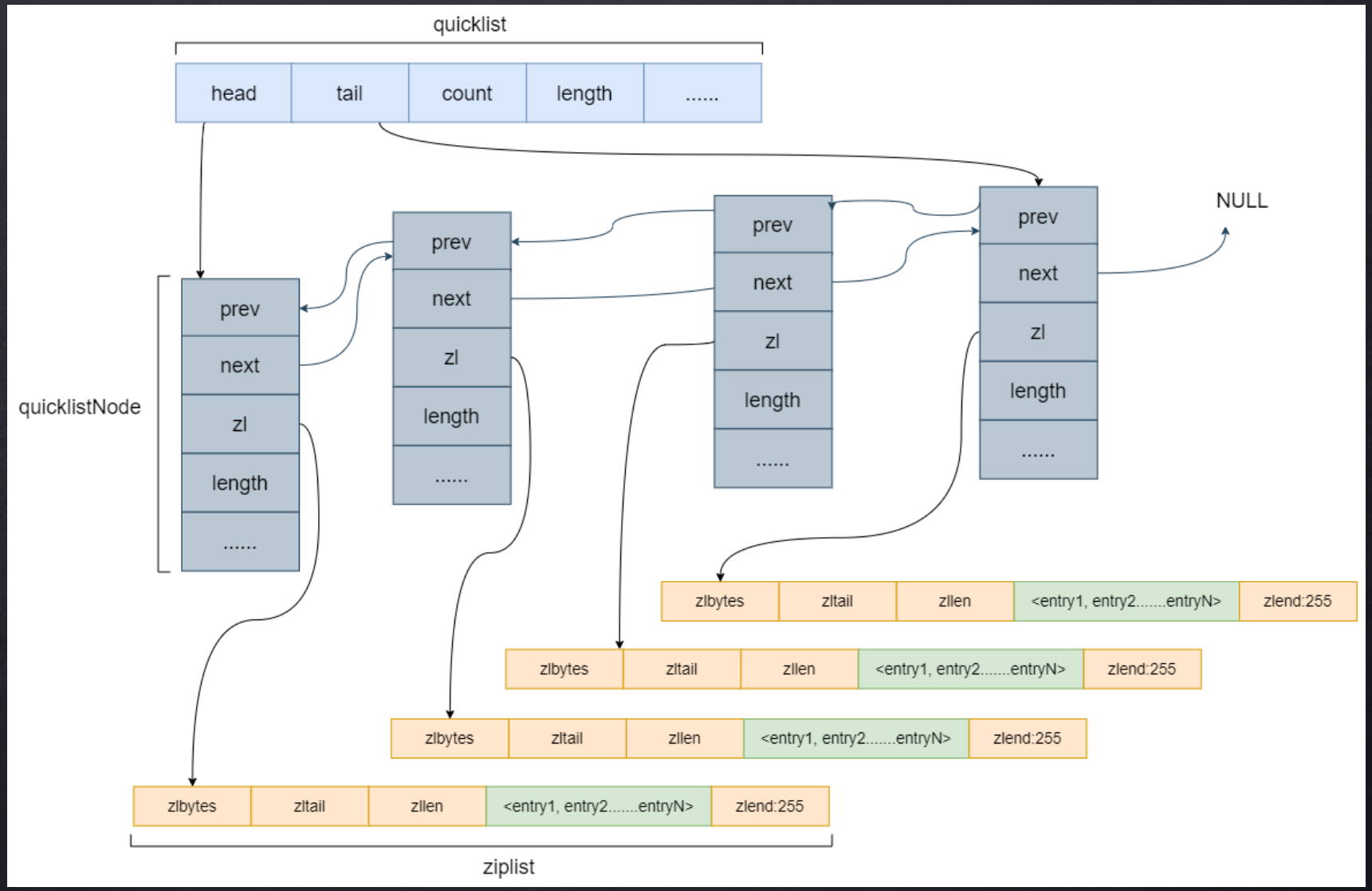
```
robj *createObject(int type, void *ptr) {
    robj *o = zmalloc(sizeof(*o));
    o->type = type;
    o->encoding = OBJ_ENCODING_RAW;
    o->ptr = ptr;
    o->refcount = 1;
    if (server.maxmemory_policy &
MAXMEMORY_FLAG_LFU) {
        o->lru = (LFUGetTimeInMinutes()<<8) |
LFU_INIT_VAL;
    } else {
        o->lru = LRU_CLOCK();   // 获取 24bit 当前时间秒数
    }
    return o;
}
```



zlbytes:32bit，表示ziplist占用的字节总数。
zltail: 32bit，表示ziplist表中最后一项（entry）在ziplist中的偏移字节数。通过zltail我们可以很方便地找到最后一项，从而可以在ziplist尾端快速地执行push或pop操作
zlen: 16bit， 表示ziplist中数据项（entry）的个数。
entry:表示真正存放数据的数据项，长度不定
zlend: ziplist最后1个字节，是一个结束标记，值固定等于255。
prerawlen: 前一个entry的数据长度。
len: entry中数据的长度
data: 真实数据存储

quicklist

```
robj *createQuicklistObject(void) {
    quicklist *l = quicklistCreate();
    robj *o = createObject(OBJ_LIST,l);
    o->encoding = OBJ_ENCODING_QUICKLIST;
    return o;
}

quicklist *quicklistCreate(void) {
    struct quicklist *quicklist;

    quicklist = zmalloc(sizeof(*quicklist));
    quicklist->head = quicklist->tail = NULL;
    quicklist->len = 0;
    quicklist->count = 0;
    quicklist->compress = 0;
    quicklist->fill = -2;
    quicklist->bookmark_count = 0;
    return quicklist;
}

typedef struct quicklist {
    quicklistNode *head;
    quicklistNode *tail;
    unsigned long count;
    unsigned long len;
    int fill : QL_FILL_BITS;
    unsigned int compress : QL_COMP_BITS;
    unsigned int bookmark_count: QL_BM_BITS;
    quicklistBookmark bookmarks[];
} quicklist;

typedef struct quicklistNode {
    struct quicklistNode *prev;
    struct quicklistNode *next;
    unsigned char *zl;
    unsigned int sz;
    unsigned int count : 16;
    unsigned int encoding : 2;
    unsigned int container : 2;
    unsigned int recompress : 1;
    unsigned int attempted_compress : 1;
    unsigned int extra : 10;
} quicklistNode;
```



Hash

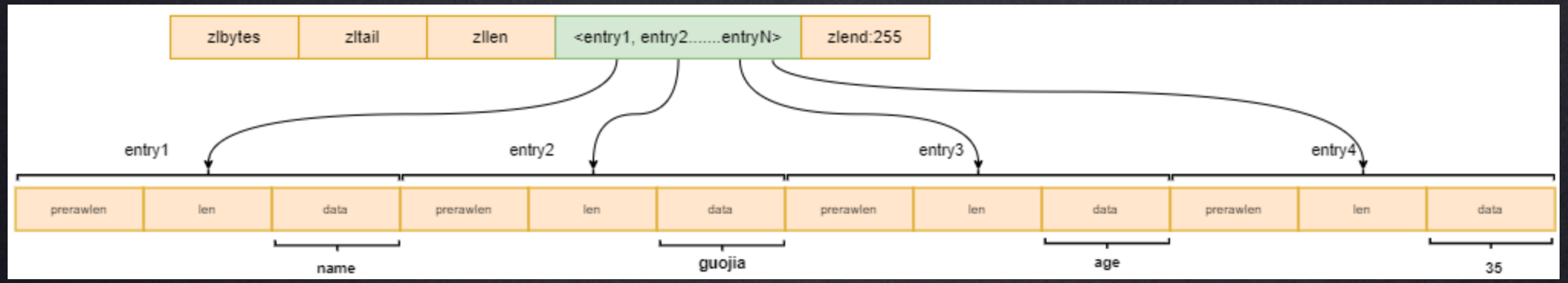
Hash常用API

/> help @hash

HSET key field value [field value ...]
HGET key field
HMGET key field [field ...]
HKEYS key
HGETALL key
HVALS key
HEXISTS key field
HDEL key field [field ...]
HINCRBY key field increment
HINCRBYFLOAT key field increment
HLEN key
HSCAN key cursor [MATCH pattern] [COUNT count]
HSETNX key field value
HSTRLEN key field

Hash

Hash 数据结构底层实现为一个字典(dict),也是RedisBb用来存储K-V的数据结构,当数据量比较小，或者单个元素比较小时，底层用ziplist存储，数据大小和元素数量阈值可以通过如下参数设置。




```
hash-max-ziplist-entries  512    //  ziplist 元素个数超过 512 ，将改为hashtable编码
hash-max-ziplist-value    64     //  单个元素大小超过 64 byte时，将改为hashtable编码
```

Set

Set 为无序的，自动去重的集合数据类型，Set 数据结构底层实现为一个value 为 null 的 字典 (dict),当数据可以用整形表示时，Set集合将被编码为intset数据结构。两个条件任意满足时Set将用hashtable存储数据。1， 元素个数大于 set-max-intset-entries，2 ， 元素无法用整形表示

set-max-intset-entries 512　　// intset 能存储的最大元素个数，超过则用hashtable编码

intset

```
typedef struct intset {
    uint32_t  encoding;
    uint32_t  length;
    int8_t    contents[];
} intset;

#define INTSET_ENC_INT16 (sizeof(int16_t))
#define INTSET_ENC_INT32 (sizeof(int32_t))
#define INTSET_ENC_INT64 (sizeof(int64_t))
```

整数集合是一个有序的，存储整型数据的结构。整型集合在Redis中可以保存int16_t,int32_t,int64_t类型的整型数据，并且可以保证集合中不会出现重复数据。

encoding: 编码类型
length: 元素个数
contents[]: 元素存储

Set

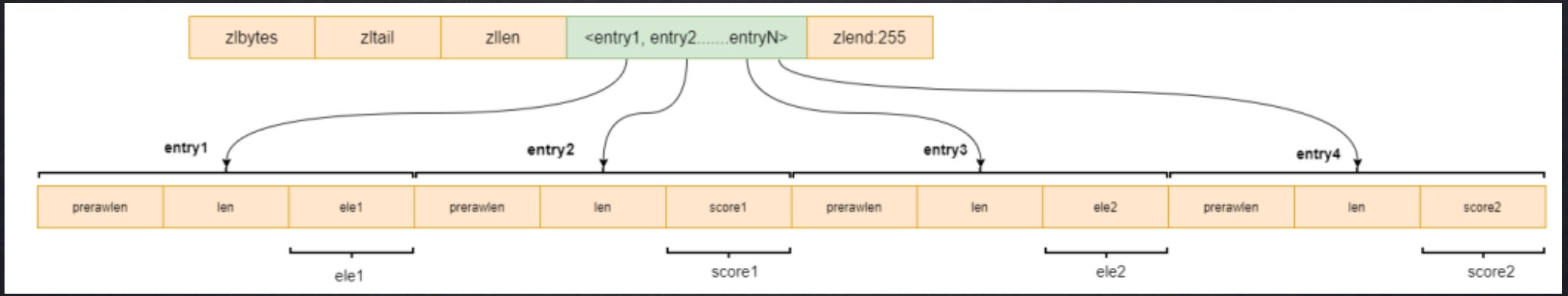
Set常用API

/> help @set

SADD key member [member ...]
SCARD key
SISMEMBER key member
SPOP key [count]
SDIFF key [key ...]
SINTER key [key ...]
SUNION key [key ...]
SMEMBERS key
SRANDMEMBER key [count]
SREM key member [member ...]
SMOVE source destination member
SUNIONSTORE destination key [key ...]
SDIFFSTORE destination key [key ...]
SINTERSTORE destination key [key ...]
SSCAN key cursor [MATCH pattern] [COUNT count]

ZSet

ZSet 为有序的，自动去重的集合数据类型，ZSet 数据结构底层实现为 字典(dict) + 跳表(skiplist),当数据比较少时，用ziplist编码结构存储。



zset-max-ziplist-entries 128 // 元素个数超过128， 将用skiplist编码
zset-max-ziplist-value 64 // 单个元素大小超过 64 byte, 将用 skiplist编码

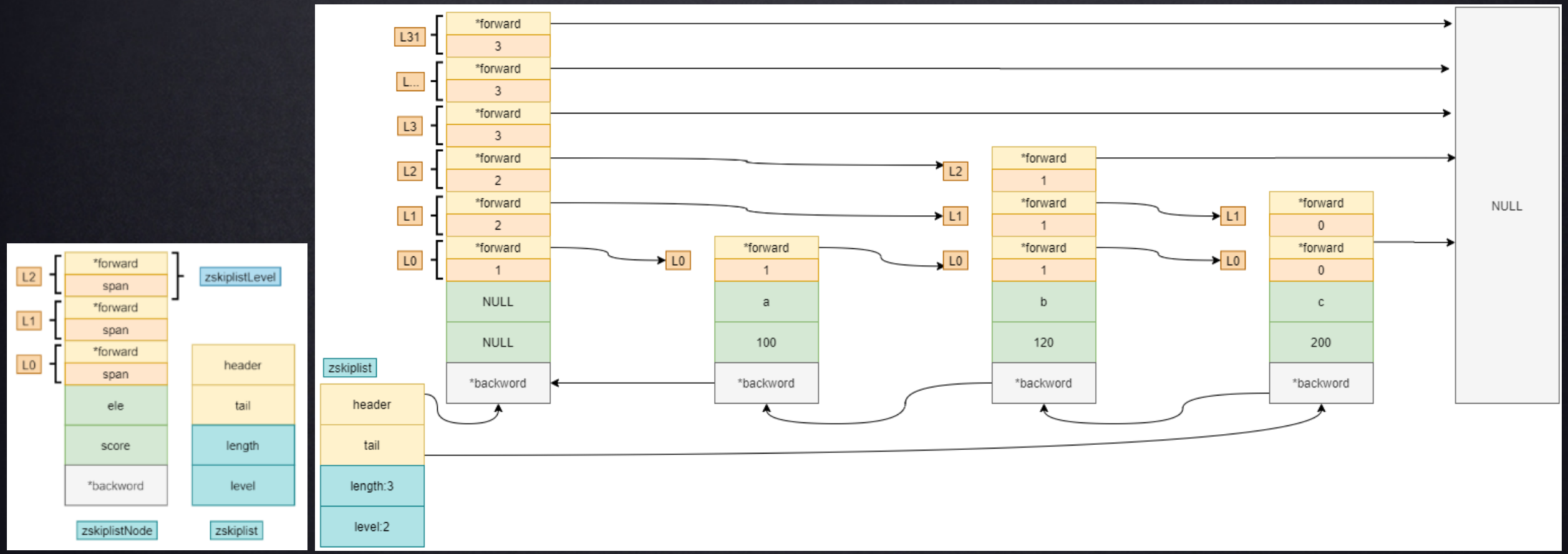
Zset

Zset 数据结构

```
 // 创建zset 数据结构: 字典 + 跳表
robj *createZsetObject(void) {
    zset *zs = zmalloc(sizeof(*zs));
    robj *o;
    // dict用来查询数据到分数的对应关系， 如 zscore 就可以直接根据 元素拿到分值
    zs->dict = dictCreate(&zsetDictType,NULL);

    // skiplist用来根据分数查询数据（可能是范围查找）
    zs->zsl = zslCreate();

    // 设置对象类型
    o = createObject(OBJ_ZSET,zs);

     // 设置编码类型
    o->encoding = OBJ_ENCODING_SKIPLIST;
    return o;
}
```

```
typedef struct zskiplistNode {
    sds ele;
    double score;
    struct zskiplistNode *backward;
    struct zskiplistLevel {
        struct zskiplistNode *forward
;
        unsigned long span;
    } level[];
} zskiplistNode;

typedef struct zskiplist {
    struct zskiplistNode *header, *t
ail;
    unsigned long length;
    int level;
} zskiplist;

typedef struct zset {
    dict *dict;
    zskiplist *zsl;
} zset;
```

skiplist


L2
*forward
span
L1
*forward
span
L0
*forward
span
ele
score
*backword
zskiplistLevel
zskiplistNode
header
tail
length
level
zskiplist
L31
L...
L3
L2
L1
L0
*forward
3
2
1
NULL
NULL
*backword
a
100
b
120
c
200
0
zskiplist
header
tail
length:3
level:2
NULL

ZSet

ZSet常用API

```
/> help  @sorted_set

  ZADD key [NX|XX] [CH] [INCR] score member [score member ...]
  ZCARD key
  ZCOUNT key min max
  ZINCRBY key increment member
  ZRANGE key start stop [WITHSCORES]
  ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]
  ZRANK key member
  ZREM key member [member ...]
  ZREMRANGEBYRANK key start stop
  ZREMRANGEBYSCORE key min max
  ZREVRANGE key start stop [WITHSCORES]
  ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]
  ZREVRANK key member
  ZSCAN key cursor [MATCH pattern] [COUNT count]
  ZSCORE key member
```

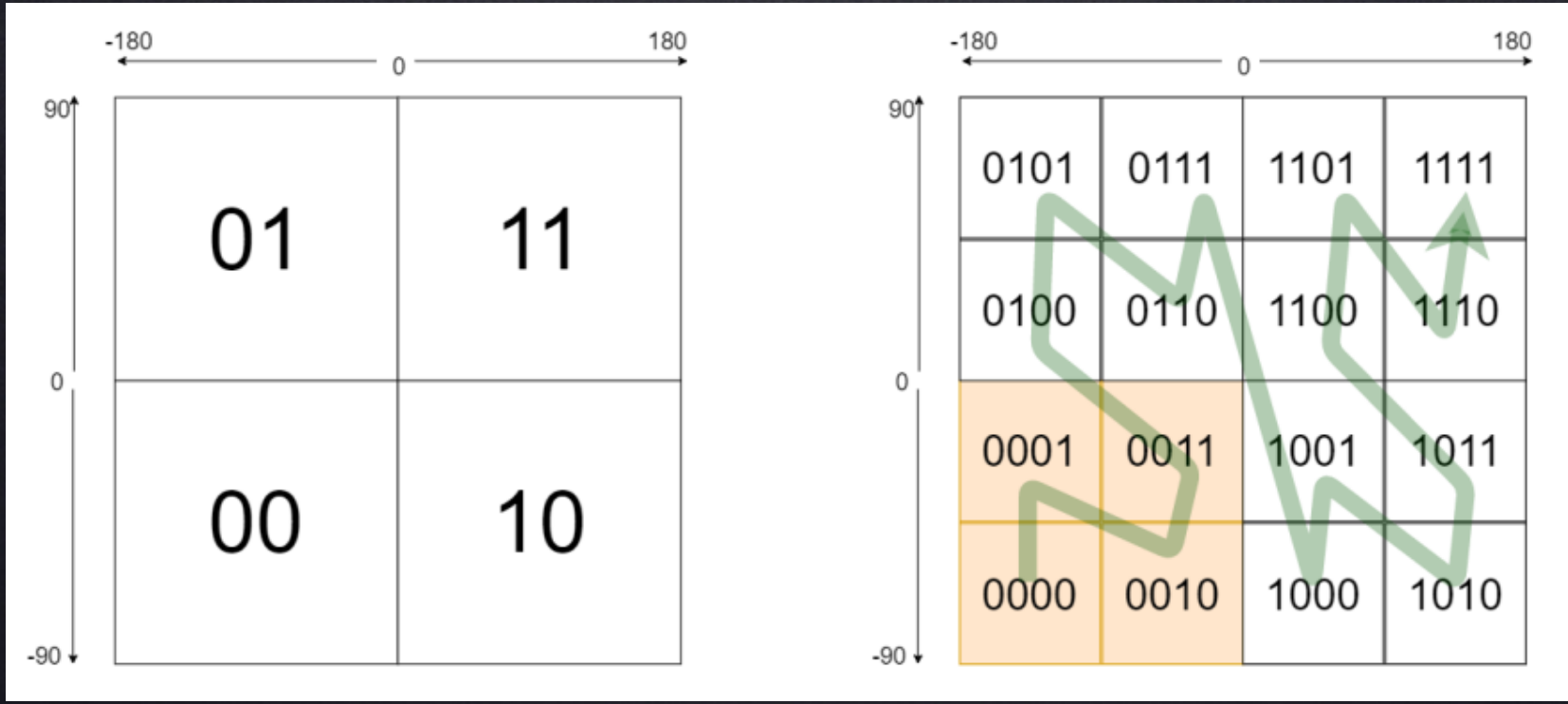
GeoHash算法

GeoHash是一种地理位置编码方法。 由Gustavo Niemeyer 和 G.M. Morton于2008年发明，它将地理位置编码为一串简短的字母和数字。它是一种分层的空间数据结构，将空间细分为网格形状的桶，这是所谓的z顺序曲线的众多应用之一，通常是空间填充曲线。

GeoHash经纬度编码

经度范围是东经180到西经180，纬度范围是南纬90到北纬90，我们设定西经为负，南纬为负，所以地球上的经度范围就是[-180， 180]，纬度范围就是[-90，90]。
如果以本初子午线、赤道为界，地球可以分成4个部分。

如果纬度范围[-90°, 0°)用二进制0代表，（0°, 90°]用二进制1代表，经度范围[-180°, 0°)用二进制0代表，（0°, 180°]用二进制1代表，那么地球可以分成如下(左图)4个部分



GeoHash经纬度编码

通过GeoHash算法，可以将经纬度的二维坐标变成一个可排序、可比较的的字符串编码。 在编码中的每个字符代表一个区域，并且前面的字符是后面字符的父区域。其算法的过程如下：

根据GeoHash 来计算 纬度的 二进制编码

地球纬度区间是[-90,90]， 如某纬度是39.92324，可以通过下面算法来进行维度编码:

1）区间[-90,90]进行二分为[-90,0),[0,90]，称为左右区间，可以确定39.92324属于右区间[0,90]，给标记为1

2）接着将区间[0,90]进行二分为 [0,45),[45,90]，可以确定39.92324属于左区间 [0,45)，给标记为0

3）递归上述过程39.92324总是属于某个区间[a,b]。随着每次迭代区间[a,b]总在缩小，并越来越逼近39.928167

4）如果给定的纬度（ 39.92324）属于左区间，则记录0，如果属于右区间则记录1，这样随着算法的进行会产生一个序列1011 1000 1100 0111 1001，序列的长度跟给定的区间划分次数有关。

GeoHash经纬度编码

纬度范围	划分区间0	划分区间1	39.92324所属区间
(-90, 90)	(-90, 0.0)	(0.0, 90)	1
(0.0, 90)	(0.0, 45.0)	(45.0, 90)	0
(0.0, 45.0)	(0.0, 22.5)	(22.5, 45.0)	1
(22.5, 45.0)	(22.5, 33.75)	(33.75, 45.0)	1
(33.75, 45.0)	(33.75, 39.375)	(39.375, 45.0)	1
(39.375, 45.0)	(39.375, 42.1875)	(42.1875, 45.0)	0
(39.375, 42.1875)	(39.375, 40.7812)	(40.7812, 42.1875)	0
(39.375, 40.7812)	(39.375, 40.0781)	(40.0781, 40.7812)	0
(39.375, 40.0781)	(39.375, 39.7265)	(39.7265, 40.0781)	1
(39.7265, 40.0781)	(39.7265, 39.9023)	(39.9023, 40.0781)	1
(39.9023, 40.0781)	(39.9023, 39.9902)	(39.9902, 40.0781)	0
(39.9023, 39.9902)	(39.9023, 39.9462)	(39.9462, 39.9902)	0
(39.9023, 39.9462)	(39.9023, 39.9243)	(39.9243, 39.9462)	0
(39.9023, 39.9243)	(39.9023, 39.9133)	(39.9133, 39.9243)	1
(39.9133, 39.9243)	(39.9133, 39.9188)	(39.9188, 39.9243)	1
(39.9188, 39.9243)	(39.9188, 39.9215)	(39.9215, 39.9243)	1

经度范围	划分区间0	划分区间1	116.3906所属区间
(-180, 180)	(-180, 0.0)	(0.0, 180)	1
(0.0, 180)	(0.0, 90.0)	(90.0, 180)	1
(90.0, 180)	(90.0, 135.0)	(135.0, 180)	0
(90.0, 135.0)	(90.0, 112.5)	(112.5, 135.0)	1
(112.5, 135.0)	(112.5, 123.75)	(123.75, 135.0)	0
(112.5, 123.75)	(112.5, 118.125)	(118.125, 123.75)	0
(112.5, 118.125)	(112.5, 115.312)	(115.312, 118.125)	1
(115.312, 118.125)	(115.312, 116.718)	(116.718, 118.125)	0
(115.312, 116.718)	(115.312, 116.015)	(116.015, 116.718)	1
(116.015, 116.718)	(116.015, 116.367)	(116.367, 116.718)	1
(116.367, 116.718)	(116.367, 116.542)	(116.542, 116.718)	0
(116.367, 116.542)	(116.367, 116.455)	(116.455, 116.542)	0
(116.367, 116.455)	(116.367, 116.411)	(116.411, 116.455)	0
(116.367, 116.411)	(116.367, 116.389)	(116.389, 116.411)	1
(116.389, 116.411)	(116.389, 116.400)	(116.400, 116.411)	0
(116.389, 116.400)	(116.389, 116.394)	(116.394, 116.400)	0

纬度产生的编码为1011 1000 1100 0111 1001，经度产生的编码为1101 0010 1100 0100 0100。偶数位放经度，奇数位放纬度，把2串编码组合生成新串：
11100 11101 00100 01111 00000 01101 01011 00001。

GeoHash经纬度编码

最后使用用0-9、b-z（去掉a, i, l, o）这32个字母进行base32编码，首先将11100 11101 00100 01111 00000 01101 01011 00001转成十进制 28，29，4，15，0，13，11，1，十进制对应的编码就是wx4g0ec1。同理，将编码转换成经纬度的解码算法与之相反

十进制	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
base32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
十进制	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
base32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

GeoHash优缺点

优点：

GeoHash利用Z阶曲线进行编码，Z阶曲线可以将二维所有点都转换成一阶曲线。地理位置坐标点通过编码转化成一维值，利用 有序数据结构如B树、SkipList等，均可进行范围搜索。因此利用GeoHash算法查找邻近点比较快

缺点：

Z 阶曲线有一个比较严重的问题，虽然有局部保序性，但是它也有突变性。在每个 Z 字母的拐角，都有可能出现顺序的突变。