redis内部数据结构深入浅出

最大感受，无论从设计还是源码，Redis都尽量做到简单，其中运用到的原理也通俗易懂。特别是源码，简洁易读，真正做到clean and clear， 这篇文章以unstable分支的源码为基准，先从大体上整理Redis的对象类型以及底层编码。 当我们在本文中提到Redis的"数据结构"，可能是在两个不同的层面来讨论它。

- 第一个层面，是从使用者的角度，string，list，hash，set，sorted set
- 第二个层面，是从内部实现的角度，属于更底层的实现，ht(dict),raw,embstr,intset,sds,ziplist,quicklist,skiplist

在讨论任何一个系统的内部实现的时候，我们都要先明确它的设计原则，这样我们才能更深刻地理解它为什么会进行如此设计的真正意图。

- 存储效率（memory efficiency）。Redis是专用于存储数据的，它对于计算机资源的主要消耗就在于内存，因此节省内存是它非常非常重要的一个方面。这意味着Redis一定是非常精细地考虑了压缩数据、减少内存碎片等问题。
- 快速响应时间（fast response time）。与快速响应时间相对的，是高吞吐量（high throughput）。Redis是用于提供在线访问的，对于单个请求的响应时间要求很高，因此，快速响应时间是比高吞吐量更重要的目标。有时候，这两个目标是矛盾的。
- 单线程（single-threaded）。Redis的性能瓶颈不在于CPU资源，而在于内存访问和网络IO。而采用单线程的设计带来的好处是，极大简化了数据结构和算法的实现。相反，Redis通过异步IO和pipelining等机制来实现高速的并发访问。显然，单线程的设计，对于单个请求的快速响应时间也提出了更高的要求。

比如：Redis一个重要的基础数据结构：dict。

- dict是一个用于维护key和value映射关系的数据结构，与很多语言中的Map或dictionary类似。Redis的一个database中所有key到value的映射，就是使用一个dict来维护的。不过，这只是它在Redis中的一个用途而已，它在Redis中被使用的地方还有很多。比如，一个Redis hash结构，当它的field较多时，便会采用dict来存储。再比如，Redis配合使用dict和skiplist来共同维护一个sorted set
- dict本质上是为了解决算法中的查找问题（Searching），一般查找问题的解法分为两个大类：一个是基于各种平衡树，一个是基于哈希表。我们平常使用的各种Map或dictionary，大都是基于哈希表实现的。在不要求数据有序存储，且能保持较低的哈希值冲突概率的前提下，基于哈希表的查找性能能做到非常高效，接近O(1)，而且实现简单。

- dict也是一个基于哈希表的算法。和传统的哈希算法类似，它采用某个哈希函数从key计算得到在哈希表中的位置，采用拉链法解决冲突，并在装载因子（load factor）超过预定值时自动扩展内存，引发重哈希（rehashing）。Redis的dict实现最显著的一个特点，就在于它的重哈希。它采用了一种称为增量式重哈希（incremental rehashing）的方法，在需要扩展内存时避免一次性对所有key进行重哈希，而是将重哈希操作分散到对于dict的各个增删改查的操作中去。这种方法能做到每次只对一小部分key进行重哈希，而每次重哈希之间不影响dict的操作。dict之所以这样设计，是为了避免重哈希期间单个请求的响应时间剧烈增加，这与前面提到的"快速响应时间"的设计原则是相符的。

一、对象类型

redis 是 key-value 存储系统，其中 key 类型一般为字符串，而 value 类型则为 redis 对象（redis object），可以绑定各种类型的数据，譬如 string、list 和set，redis.h 中定义了 struct redisObject，它是一个简单优秀的数据结构

```
#define LRU_BITS 24
#define LRU_CLOCK_MAX ((1<<LRU_BITS)-1) /* Max value of obj->lru */
#define LRU_CLOCK_RESOLUTION 1000 /* LRU clock resolution in ms */

typedef struct redisObject {
    //对象的数据类型，占4bits，共5种类型
    unsigned type:4;
    //对象的编码类型，占4bits，共10种类型
    unsigned encoding:4;

    //least recently used
    //实用LRU算法计算相对server.lruclock的LRU时间
    unsigned lru:LRU_BITS; /* lru time (relative to server.lruclock) */

    //引用计数
    int refcount;

    //指向底层数据实现的指针
    void *ptr;
} robj;

//type的占5种类型：
/* Object types */
#define OBJ_STRING 0    //字符串对象
#define OBJ_LIST 1      //列表对象
#define OBJ_SET 2       //集合对象
#define OBJ_ZSET 3      //有序集合对象
#define OBJ_HASH 4      //哈希对象

/* Objects encoding. Some kind of objects like Strings and Hashes can be
 * internally represented in multiple ways. The 'encoding' field of the object
 * is set to one of this fields for this object. */
// encoding 的10种类型
#define OBJ_ENCODING_RAW 0     /* Raw representation */     //原始表示方式，字符串对象是简单动态字
#define OBJ_ENCODING_INT 1     /* Encoded as integer */         //long类型的整数
#define OBJ_ENCODING_HT 2      /* Encoded as hash table */      //字典
#define OBJ_ENCODING_ZIPMAP 3  /* Encoded as zipmap */          //不在使用
#define OBJ_ENCODING_LINKEDLIST 4 /* Encoded as regular linked list */  //双端链表,不在使用
#define OBJ_ENCODING_ZIPLIST 5 /* Encoded as ziplist */         //压缩列表
```

```
#define OBJ_ENCODING_INTSET 6  /* Encoded as intset */          //整数集合
#define OBJ_ENCODING_SKIPLIST 7  /* Encoded as skiplist */      //跳跃表和字典
#define OBJ_ENCODING_EMBSTR 8  /* Embedded sds string encoding */   //embstr编码的简单动态字符
#define OBJ_ENCODING_QUICKLIST 9 /* Encoded as linked list of ziplists */   //由压缩列表组成的
```

其中，void *ptr 已经给了我们无限的遐想空间了（把最后一个指针留给了真正的数据）
每种类型的对象至少都有两种或以上的encoding方式，不同编码可以在不同的使用场景上优化对象的使用场景，用TYPE命令可查看某个键值对的类型

二、对象编码

不同类型和编码的对象

```
REDIS_STRING    REDIS_ENCODING_INT  使用整数值实现的字符串对象。
REDIS_STRING    REDIS_ENCODING_EMBSTR   使用 embstr 编码的简单动态字符串实现的字符串对象。
REDIS_STRING    REDIS_ENCODING_RAW  使用简单动态字符串实现的字符串对象。
REDIS_LIST  REDIS_ENCODING_ZIPLIST  使用压缩列表实现的列表对象。
REDIS_LIST  REDIS_ENCODING_LINKEDLIST   使用双端链表实现的列表对象。
REDIS_HASH  REDIS_ENCODING_ZIPLIST  使用压缩列表实现的哈希对象。
REDIS_HASH  REDIS_ENCODING_HT   使用字典实现的哈希对象。
REDIS_SET   REDIS_ENCODING_INTSET   使用整数集合实现的集合对象。
REDIS_SET   REDIS_ENCODING_HT   使用字典实现的集合对象。
REDIS_ZSET  REDIS_ENCODING_ZIPLIST  使用压缩列表实现的有序集合对象。
REDIS_ZSET  REDIS_ENCODING_SKIPLIST 使用跳跃表和字典实现的有序集合对象。
```

OBJECT ENCODING 对不同编码的输出

```
整数  REDIS_ENCODING_INT  "int"
embstr 编码的简单动态字符串（SDS）  REDIS_ENCODING_EMBSTR   "embstr"
简单动态字符串 REDIS_ENCODING_RAW  "raw"
字典  REDIS_ENCODING_HT   "hashtable"
双端链表    REDIS_ENCODING_LINKEDLIST   "linkedlist"
压缩列表    REDIS_ENCODING_ZIPLIST  "ziplist"
整数集合    REDIS_ENCODING_INTSET   "intset"
跳跃表和字典  REDIS_ENCODING_SKIPLIST "skiplist"
```

本质上，Redis就是基于这些数据结构而构造出一个对象存储系统。
关于redisObject

- ptr指针，指向对象的底层实现数据结构
- encoding属性记录对象所使用的编码
- 淘汰时钟，Redis 对数据集占用内存的大小有「实时」的计算，当超出限额时，会淘汰超时的数据
- 引用计数，一个 Redis 对象可能被多个指针引用。当需要增加或者减少引用的时候，必须调用相应的函数，程序员必须遵守这一准则

```
// 增加 Redis 对象引用
void incrRefCount(robj *o) {
    o->refcount++;
}
```

```
// 减少 Redis 对象引用。特别的，引用为零的时候会销毁对象
void decrRefCount(robj *o) {
    if (o->refcount <= 0) redisPanic("decrRefCount against refcount <= 0");
    // 如果取消的是最后一个引用，则释放资源
    if (o->refcount == 1) {
    // 不同数据类型，销毁操作不同
    switch(o->type) {
        case REDIS_STRING: freeStringObject(o); break;
        case REDIS_LIST: freeListObject(o); break;
        case REDIS_SET: freeSetObject(o); break;
        case REDIS_ZSET: freeZsetObject(o); break;
        case REDIS_HASH: freeHashObject(o); break;
        default: redisPanic("Unknown object type"); break;
    }
    zfree(o);
    } else {
        o->refcount--;
    }
}
```

得益于 Redis 是单进程单线程工作的，所以增加/减少引用的操作不必保证原子性，这在 memcache 中是做不到的（ memcached 是多线程的工作模式，需要做到互斥）

1、Keys

redis是一个key-value db，首先key也是字符串类型，但是key中不能包括边界字符，由于key不是binary safe的字符串，所以像”my key”和”mykey\n”这样包含空格和换行的key是不允许的 ，顺便说一下在redis内部并不限制使用binary字符，这是redis协议限制的，”\r\n”在协议格式中会作为特殊字符。

redis 1.2以后的协议中部分命令已经开始使用新的协议格式了(比如MSET)，总之目前还是把包含边界字符当成非法的key，另外关于key的一个格式约定介绍下，object-type:id:field。比如user:1000:password，blog:xxidxx:title

2、string

string是redis最基本的类型，而且string类型是二进制安全的。意思是redis的string可以包含任何数据，比如jpg图片或者序列化的对象。从内部实现来看其实string可以看作byte数组，最大上限是1G字节。

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

buf是个char数组用于存贮实际的字符串内容。其实char和c#中的byte是等价的，都是一个字节 ，len是buf数组的长度，free是数组中剩余可用字节数。 由此可以理解为什么string类型是二进制安全的了。因为它本质上就是个byte数组。当然可以包含任何数据了。 另外string类型可以被部分命令按int处理，比如incr等命令，redis的其他类型像list,set,sorted set ,hash它们包含的元素与都只能是string类型。

编码

字符串对象的编码可以是 INT、RAW 或 EMBSTR。如果保存的是整数值并且可以用long表示，那么编码会设置为INT。当字符串值得长度大于44字节使用RAW，小于等于44字节使用EMBSTR。

Redis在3.0引入EMBSTR编码，这是一种专门用于保存短字符串的一种优化编码方式，这种编码和RAW编码都是用sdshdr简单动态字符串结构来表示。RAW编码会调用两次内存分配函数来分别创建redisObject和sdshdr结构，而EMBSTR只调用一次内存分配函数来分配一块连续的空间保存数据，比起RAW编码的字符串更能节省内存，以及能提升获取数据的速度。

不过要注意！EMBSTR是不可修改的，当对EMBSTR编码的字符串执行任何修改命令，总会先将其转换成RAW编码再进行修改；而INT编码在条件满足的情况下也会被转换成RAW编码。

两种字符串对象编码方式的区别

```
/* Create a string object with EMBSTR encoding if it is smaller than
 * REIDS_ENCODING_EMBSTR_SIZE_LIMIT, otherwise the RAW encoding is
 * used.
 *
 * The current limit of 39 is chosen so that the biggest string object
 * we allocate as EMBSTR will still fit into the 64 byte arena of jemalloc. */

//sdshdr8的大小为3个字节，加上1个结束符共4个字节
//redisObject的大小为16个字节
//redis使用jemalloc内存分配器，且jemalloc会分配8，16，32，64等字节的内存
//一个embstr固定的大小为16+3+1 = 20个字节，因此一个最大的embstr字符串为64-20 = 44字节
#define OBJ_ENCODING_EMBSTR_SIZE_LIMIT 44

// 创建字符串对象，根据长度使用不同的编码类型
// createRawStringObject和createEmbeddedStringObject的区别是：
// createRawStringObject是当字符串长度大于44字节时，robj结构和sdshdr结构在内存上是分开的
// createEmbeddedStringObject是当字符串长度小于等于44字节时，robj结构和sdshdr结构在内存上是连续的
robj *createStringObject(const char *ptr, size_t len) {
    if (len <= OBJ_ENCODING_EMBSTR_SIZE_LIMIT)
        return createEmbeddedStringObject(ptr,len);
    else
        return createRawStringObject(ptr,len);
}
```

字符串对象编码的优化

```
/* Try to encode a string object in order to save space */
//尝试优化字符串对象的编码方式以节约空间
robj *tryObjectEncoding(robj *o) {
    long value;
    sds s = o->ptr;
    size_t len;

    /* Make sure this is a string object, the only type we encode
     * in this function. Other types use encoded memory efficient
     * representations but are handled by the commands implementing
     * the type. */
    serverAssertWithInfo(NULL,o,o->type == OBJ_STRING);

    /* We try some specialized encoding only for objects that are
     * RAW or EMBSTR encoded, in other words objects that are still
     * in represented by an actually array of chars. */
    //如果字符串对象的编码类型为RAW或EMBSTR时，才对其重新编码
    if (!sdsEncodedObject(o)) return o;
```

```
    /* It's not safe to encode shared objects: shared objects can be shared
     * everywhere in the "object space" of Redis and may end in places where
     * they are not handled. We handle them only as values in the keyspace. */
    //如果refcount大于1，则说明对象的ptr指向的值是共享的，不对共享对象进行编码
     if (o->refcount > 1) return o;

    /* Check if we can represent this string as a long integer.
     * Note that we are sure that a string larger than 20 chars is not
     * representable as a 32 nor 64 bit integer. */
    len = sdslen(s);            //获得字符串s的长度

    //如果len小于等于20，表示符合long long可以表示的范围，且可以转换为long类型的字符串进行编码
    if (len <= 20 && string2l(s,len,&value)) {
        /* This object is encodable as a long. Try to use a shared object.
         * Note that we avoid using shared integers when maxmemory is used
         * because every object needs to have a private LRU field for the LRU
         * algorithm to work well. */
        if ((server.maxmemory == 0 ||
             (server.maxmemory_policy != MAXMEMORY_VOLATILE_LRU &&
              server.maxmemory_policy != MAXMEMORY_ALLKEYS_LRU)) &&
            value >= 0 &&
            value < OBJ_SHARED_INTEGERS)     //如果value处于共享整数的范围内
        {
            decrRefCount(o);                 //原对象的引用计数减1，释放对象
            incrRefCount(shared.integers[value]); //增加共享对象的引用计数
            return shared.integers[value];       //返回一个编码为整数的字符串对象
        } else {         //如果不处于共享整数的范围
            if (o->encoding == OBJ_ENCODING_RAW) sdsfree(o->ptr);    //释放编码为OBJ_ENCODING_
            o->encoding = OBJ_ENCODING_INT;      //转换为OBJ_ENCODING_INT编码
            o->ptr = (void*) value;              //指针ptr指向value对象
            return o;
        }
    }

    /* If the string is small and is still RAW encoded,
     * try the EMBSTR encoding which is more efficient.
     * In this representation the object and the SDS string are allocated
     * in the same chunk of memory to save space and cache misses. */
    //如果len小于44，44是最大的编码为EMBSTR类型的字符串对象长度
    if (len <= OBJ_ENCODING_EMBSTR_SIZE_LIMIT) {
        robj *emb;

        if (o->encoding == OBJ_ENCODING_EMBSTR) return o;    //将RAW对象转换为OBJ_ENCODING_EMBS
        emb = createEmbeddedStringObject(s,sdslen(s)); //创建一个编码类型为OBJ_ENCODING_EMBSTR
        decrRefCount(o);     //释放之前的对象
        return emb;
    }

    /* We can't encode the object...
     *
     * Do the last try, and at least optimize the SDS string inside
     * the string object to require little space, in case there
     * is more than 10% of free space at the end of the SDS string.
     *
     * We do that only for relatively large strings as this branch
     * is only entered if the length of the string is greater than
     * OBJ_ENCODING_EMBSTR_SIZE_LIMIT. */
    //无法进行编码，但是如果s的未使用的空间大于使用空间的10分之1
    if (o->encoding == OBJ_ENCODING_RAW &&
        sdsavail(s) > len/10)
    {
        o->ptr = sdsRemoveFreeSpace(o->ptr);     //释放所有的未使用空间
    }
```

```
    /* Return the original object. */
    return o;
}
```

3、list

list类型其实就是一个每个子元素都是string类型的双向链表。所以[lr]push和[lr]pop命令的算法时间复杂度都是O(n)，另外list会记录链表的长度。所以llen操作也是O(n).链表的最大长度是(2的32次方-1)。

我们可以通过push,pop操作从链表的头部或者尾部添加删除元素。这使得list既可以用作栈，也可以用作队列。 有意思的是list的pop操作还有阻塞版本的。当我们[lr]pop一个list对象，如果list是空，或者不存在，会立即返回nil。但是阻塞版本的b[lr]pop可以则可以阻塞， 当然可以加超时时间，超时后也会返回nil。为什么要阻塞版本的pop呢，主要是为了避免轮询。 如果我们用list来实现一个工作队列。执行任务的thread可以调用阻塞版本的pop去 ，获取任务这样就可以避免轮询去检查是否有任务存在。当任务来时候工作线程可以立即返回，也可以避免轮询带来的延迟。

编码

Redis3.0之前的列表对象的编码可以是ziplist或者linkedlist。当列表对象保存的字符串元素的长度都小于64字节并且保存的元素数量小于512个，使用ziplist编码，可以通过修改配置list-max-ziplist-value和list-max-ziplist-entries来修改这两个条件的上限值、两个条件任意一个不满足时，ziplist会变为linkedlist。

从3.2开始Redis只使用quicklist作为列表的编码，quicklist是ziplist和双向链表的结合体，quicklist的每个节点都是一个ziplist。可以通过修改list-max-ziplist-size来设置一个quicklist节点上的ziplist的长度，取正值表示通过元素数量来限定ziplist的长度；负数表示按照占用字节数来限定，并且Redis规定只能取-1到-5这五个负值

```
-5: 每个quicklist节点上的ziplist大小不能超过64 Kb。（注：1kb => 1024 bytes）
-4: 每个quicklist节点上的ziplist大小不能超过32 Kb。
-3: 每个quicklist节点上的ziplist大小不能超过16 Kb。
-2: 每个quicklist节点上的ziplist大小不能超过8 Kb。（默认值）
-1: 每个quicklist节点上的ziplist大小不能超过4 Kb。
```

另外配置参数list-compress-depth表示一个quicklist两端不被压缩的节点个数

```
0: 表示都不压缩。默认值。
1: 表示quicklist两端各有1个节点不压缩，中间的节点压缩。
2: 表示quicklist两端各有2个节点不压缩，中间的节点压缩。
3: 表示quicklist两端各有3个节点不压缩，中间的节点压缩。
依此类推…
```

这里采用的是一种叫LZF的无损压缩算法

4、hash

哈希对象的编码可以是ziplist或者hashtable。使用ziplist 编码时，保存同一键值对的两个节点总是紧挨在一起，键节点在前，值节点在后，同时满足以下两个条件将使用ziplist编

码：

- 所有键和值的字符串长度小于64字节
- 键值对的数量小于512个

不能满足这两个条件的都需要使用hashtable编码。以上两个上限值可以通过hash-max-ziplist-value和hash-max-ziplist-entries来修改
hash是一个string类型的field和value的映射表，它的添加，删除操作都是O(1)，hash特别适合用于存储对象。 相较于将对象的每个字段存成单个string类型，将一个对象存储在hash类型中会占用更少的内存，并且可以更方便的存取整个对象。
省内存的原因是新建一个hash对象时开始是用zipmap（又称为small hash）来存储的。 这个zipmap其实并不是hash table，但是zipmap相比正常的hash实现可以节省不少hash本身需要的一些元数据存储开销。 尽管zipmap的添加，删除，查找都是O(n)，但是由于一般对象的field数量都不太多。 所以使用zipmap也是很快的,也就是说添加删除平均还是O(1)。 如果field或者value的大小超出一定限制后，redis会在内部自动将zipmap替换成正常的hash实现，这个限制可以在配置文件中指定

```
hash-max-zipmap-entries 64 #配置字段最多64个
hash-max-zipmap-value 512 #配置value最大为512字节
```

5、set
集合对象的编码可以是intset或者hashtable。当满足以下两个条件时使用intset编码：

- 所有元素都是整数值
- 元素数量不超过512个

可以修改set-max-intset-entries设置元素数量的上限。使用hashtable编码时，字典的每一个键都是字符串对象，每个字符串对象包含一个集合元素，字典的值全部设置为null。
redis的set是string类型的无序集合。set元素最大可以包含(2的32次方-1)个元素。 set的是通过hash table实现的，所以添加，删除，查找的复杂度都是O(1)。hash table会随着添加或者删除自动的调整大小。 需要注意的是调整hash table大小时候需要同步（获取写锁）会阻塞其他读写操作。 可能不久后就会改用跳表（skip list）来实现跳表已经在sorted set中使用了 关于set集合类型除了基本的添加删除操作，其他有用的操作还包含集合的取并集(union)，交集(intersection)， 差集(difference)。
6、sorted set
有序集合对象的编码可以是ziplist或者skiplist。同时满足以下条件时使用ziplist编码：

- 元素数量小于128个
- 所有member的长度都小于64字节

以上两个条件的上限值可通过zset-max-ziplist-entries和zset-max-ziplist-value来修改。

ziplist编码的有序集合使用紧挨在一起的压缩列表节点来保存，第一个节点保存member，第二个保存score。ziplist内的集合元素按score从小到大排序，score较小的排在表头位置。
skiplist编码的有序集合底层是一个命名为zset的结构体，而一个zset结构同时包含一个字典和一个跳跃表。跳跃表按score从小到大保存所有集合元素。而字典则保存着从member到score的映射，这样就可以用O(1)的复杂度来查找member对应的score值。虽然同时使用两种结构，但它们会通过指针来共享相同元素的member和score，因此不会浪费额外的内存。
和set一样sorted set也是string类型元素的集合，不同的是每个元素都会关联一个double类型的score。sorted set的实现是skip list和hash table的混合体，当元素被添加到集合中时，一个元素到score的映射被添加到hash table中，所以给定一个元素获取score的开销是O(1)，另一个score到元素的映射被添加到skip list并按照score排序，所以就可以有序的获取集合中的元素。 添加，删除操作开销都是O(1)和skip list的开销一致，redis的skip list实现用的是双向链表，这样就可以逆序从尾部取元素。 sorted set最经常的使用方式应该是作为索引来使用，我们可以把要排序的字段作为score存储，对象的id当元素存储。