

Redis的数据结构

注意：截止2020/12/25，海外业务的线上redis集群，大部分版本是3.2.3(官方发布时间2016.8.12)

2016-2020年之间，有新增的4.0.0+，5.0.0+，6.0.0+版本。功能、底层结构都可能发生变化。

故，本文内容基于3.2.3的版本

https://blog.csdn.net/u011983531/article/details/79598671?utm_term=redis%E5%AD%A8%E6%95%B0%E6%8D%AE%E5%A4%A7%E5%B0%8F%E8%AE%A1%E7%AE%97&utm_medium=distribute.pc_aggpage_search_result.none-task-blog-2~all~sobaiduweb~default-2-79598671&spm=3001.4430

零、最小储存单位

排序	数据结构	解释	计算公式
1	指针	8字节	8
2	int	4字节	4
3	long	8字节	8

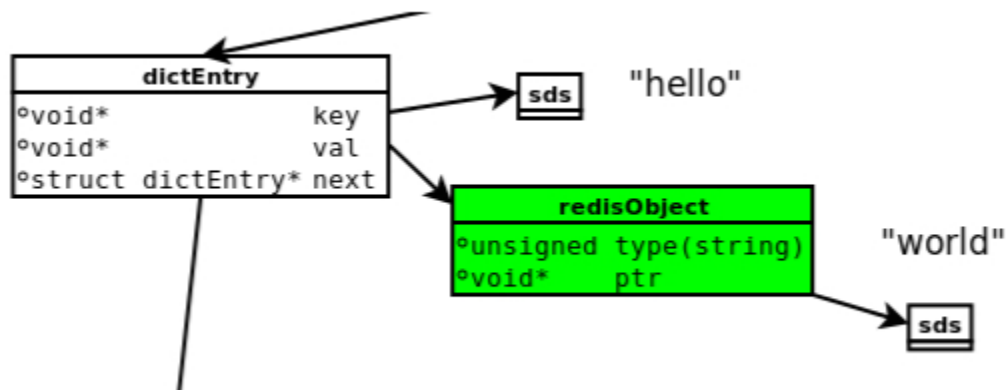
一、底层数据结构

排序	底层数据结构	最底层结构体	计算公式
1	SDS		$size = 4 + 4 + (\text{字符数} * 1 + 1)$
2	链表		
3	字典		
4	跳跃表		
		zskiplist	
		zskiplistNode	
5	压缩列表		
6	整数集合		

二、辅助数据结构

排序	辅助数据结构	解释	计算公式
1	DictEntry	k-v 结构	$size = 8 + 8 + 8 = 24B$
2	RedisObject		$size = 4 + 4 + 8 = 16B$

以执行命令：set hello word 为例



2. RedisObject

```

typedef struct redisObject {
    //
    unsigned type:4;

    //
    unsigned encoding:4;

    // , REDIS_LRU_BITS = 24 (in server.h)
    unsigned lru:REDIS_LRU_BITS; /* lru time (relative to server.lruclock) */

    // 0redis
    int refcount;

    //
    void *ptr;
}
  
```

4bit + 4bit + 24bit + 4 byte + 8byte

2.1 对象类型

表 8-1 对象的类型

类型常量	对象的名称
REDIS_STRING	字符串对象
REDIS_LIST	列表对象
REDIS_HASH	哈希对象
REDIS_SET	集合对象
REDIS_ZSET	有序集合对象

【小提示：type命令就是打印这些常量，即redisObject中的type字段】

```
172.17.160.222:30004> set hello oppo
OK
172.17.160.222:30004> type hello
string
172.17.160.222:30004> sadd myset mem1 mem2 mem3
(integer) 3
172.17.160.222:30004> type myset
set
```

2.2 encoding编码

encoding 表示 ptr 指向的具体数据结构，即这个对象使用了什么数据结构作为底层实现。
encoding 的取值范围如下（出自《Redis设计与实现第二版》第八章：对象）：

表 8-3 对象的编码

编码常量	编码所对应的底层数据结构
REDIS_ENCODING_INT	long 类型的整数
REDIS_ENCODING_EMBSTR	embstr 编码的简单动态字符串
REDIS_ENCODING_RAW	简单动态字符串
REDIS_ENCODING_HT	字典
REDIS_ENCODING_LINKEDLIST	双端链表
REDIS_ENCODING_ZIPLIST	压缩列表
REDIS_ENCODING_INTSET	整数集合
REDIS_ENCODING_SKIPLIST	跳跃表和字典

每种类型的对象都至少使用了两种不同的编码，对象和编码的对应关系如下（出自《Redis设计与实现第二版》第八章：对象）：

表 8-4 不同类型和编码的对象

类 型	编 码	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

【tips: object encoding命令就是打印这些常量，即redisObject中的encoding字段】

三、基本数据结构

从Redis3.2开始，sds就有了5种类型，5种类型分别存放不同大小的字符串。

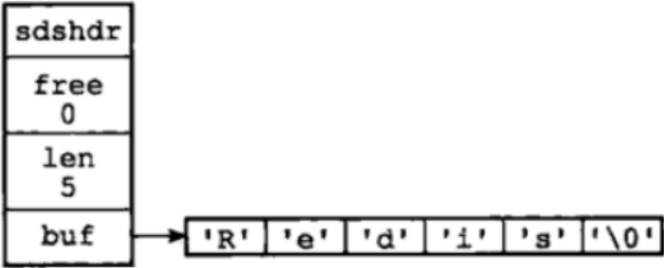
在创建字符串时，sds会根据字符串的长度选择不同的类型。最终由sdsnewlen函数创建字符串(1-16字节)

```
6 struct __attribute__((packed)) sdshdr5 {
7     unsigned char flags; /* 3 lsb of type, and 5 msb of string length */
8     char buf[];
9 };
10 struct __attribute__((packed)) sdshdr8 {
11     uint8_t len; /* used */
12     uint8_t alloc; /* excluding the header and null terminator */
13     unsigned char flags; /* 3 lsb of type, 5 unused bits */
14     char buf[];
15 };
16 struct __attribute__((packed)) sdshdr16 {
17     uint16_t len; /* used */
18     uint16_t alloc; /* excluding the header and null terminator */
19     unsigned char flags; /* 3 lsb of type, 5 unused bits */
20     char buf[];
21 };
22 struct __attribute__((packed)) sdshdr32 {
23     uint32_t len; /* used */
24     uint32_t alloc; /* excluding the header and null terminator */
25     unsigned char flags; /* 3 lsb of type, 5 unused bits */
26     char buf[];
27 };
28 struct __attribute__((packed)) sdshdr64 {
29     uint64_t len; /* used */
30     uint64_t alloc; /* excluding the header and null terminator */
31     unsigned char flags; /* 3 lsb of type, 5 unused bits */
32     char buf[];
33 };
```

```
struct sdshdr {
    //buf
    //SDS
    int len;

    //buf
    int free

    //
    char buf[];
};
```

Redis常用的数据结构	可能的底层结构	使用条件(关系全是&)	备注	示意图
String	int	1、值是整数 2、值可以用long来表示	浮点数使用字符串存的	
	raw (即sds)	1、值是字符串值 (ps: 使用strlen命令获取key对应值的长度) 2、长度>39字节	分别调用2次jemalloc()	

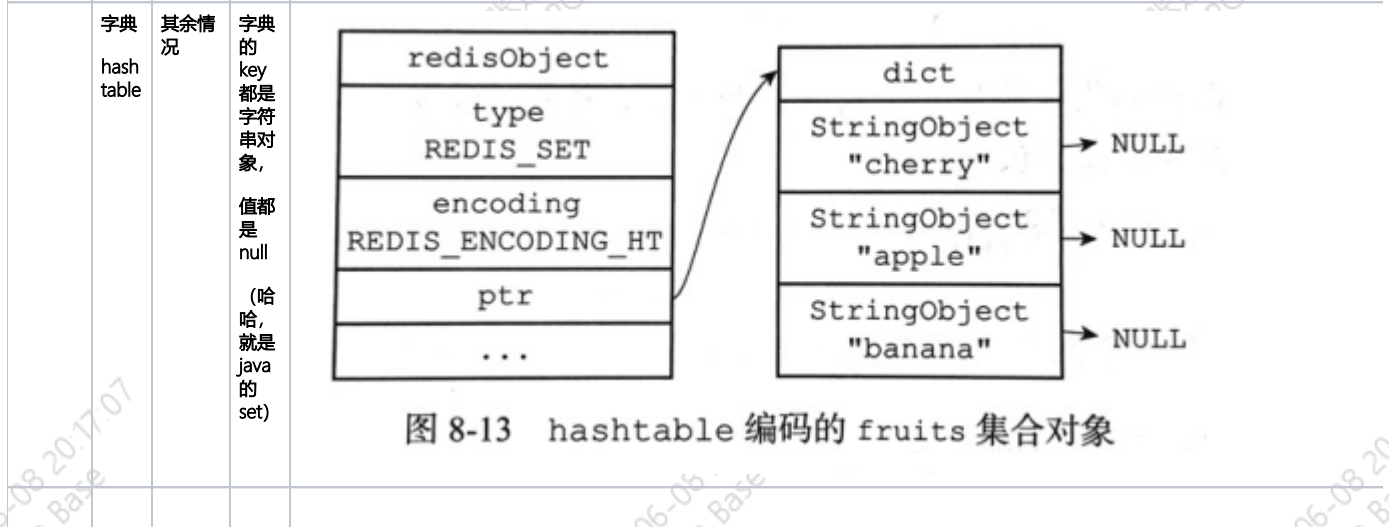
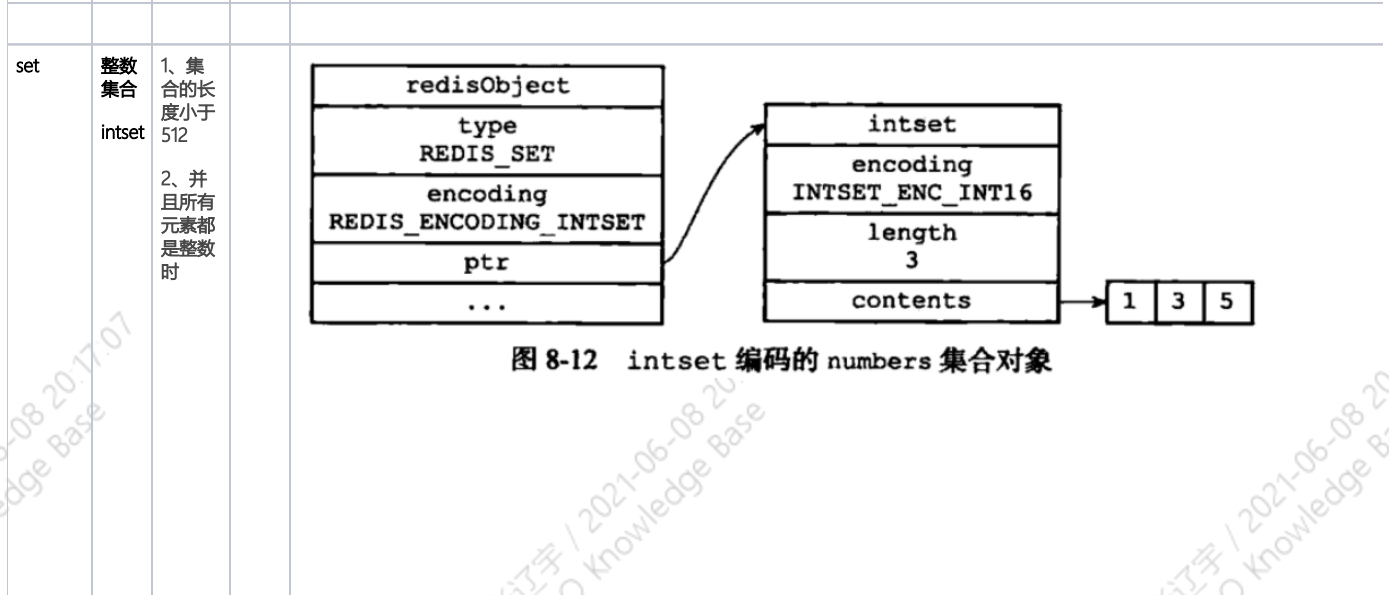
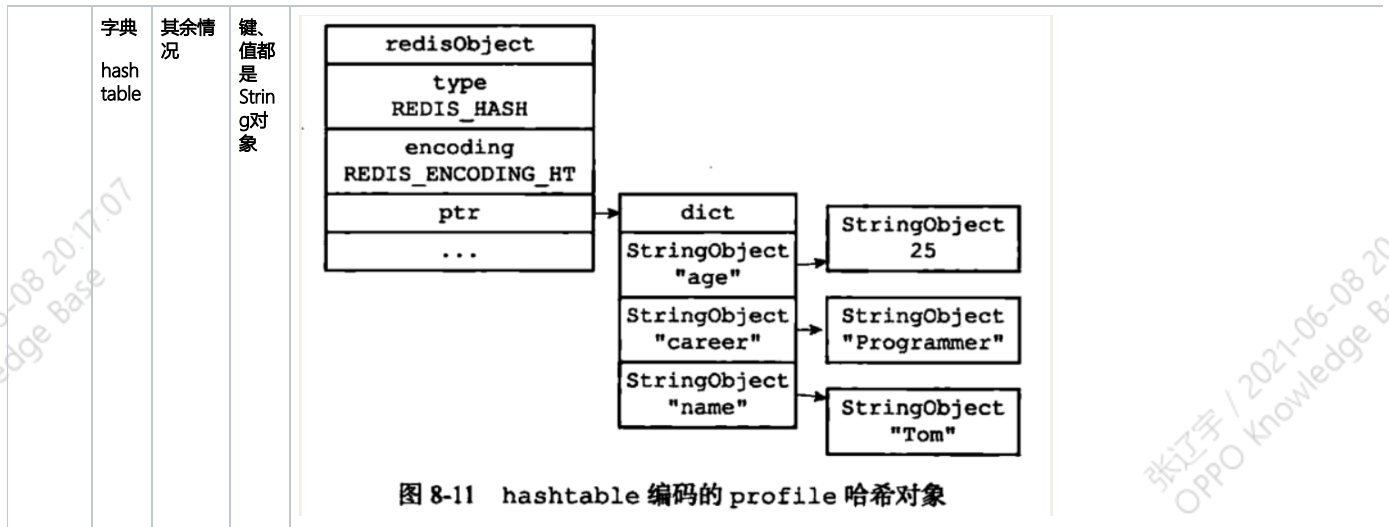
embstr	1、值是字符串 (ps: 使用 strlen 命令获取key 对应值的长度) 2、长度 <=39 字节	只调用一次 jemalloc 来分配一块连续的内存空间, 给 redis Object 和 sdshdr <
--------	---	---

	双端链表 linke dlist	其余情况	<pre>type def struc t listNode{ str uct listNode * prev; str uct listNode * prev; vo id * valu e; }</pre>	<p>redisObject</p> <p>type REDIS_LIST</p> <p>encoding REDIS_ENCODING_LINKEDLIST</p> <p>ptr</p> <p>...</p> <p>链表</p> <p>StringObject 1 → StringObject "three" → StringObject 5</p>
--	------------------------	------	---	---

图 8-6 linkedlist 编码的 numbers 列表对象

hash	压缩列表 ziplist	<p>1、键值对数量 < 512</p> <p>2、所有键值对长度 < 64字节</p>	<p>第一个添加的键值对 第二个添加的键值对 最新添加的键值对</p> <p>键 值</p> <p>zllen "name" "Tom" "age" 25 "career" "Programmer"</p>
------	-----------------	--	--

图 8-10 profile 哈希对象的压缩列表底层实现



zSet	压缩链表	<p>1、有序集合的长度小于128</p> <p>2、并且所有元素的长度都小于64字节时</p>	<p>分值较小的排在前面</p>	<p>图 8-14 ziplist 编码的有序集合对象</p> <p>图 8-15 有序集合元素在压缩列表中按分值从小到大排列</p>
	跳跃表&字典	<p>其余情况</p> <p>Q: 为什么同时要使用跳跃表和字典存储?</p> <p>A:</p> <p>1、字典的优势是 $O(1)$ 查找, 但是 $zrank$, $zrange$ 等命令就是 $O(N \log N)$</p> <p>2、跳跃表的查找操作为 $O(\log N)$</p>		<p>图 8-17 有序集合元素同时被保存在字典和跳跃表中</p>

四、扩展数据结构

Redis常用的数据结构	可能的底层结构				计算公式
HyperLogLog					size =
位图					size =
流					size =
地理坐标					size =

五、工具

http://www.redis.cn/redis_memory/

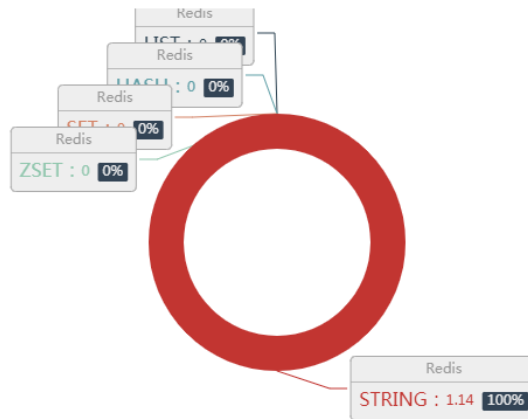
Redis容量预估

Key类型	Key个数	Key长度 单位 (字节)	元素/字段数	元素/字段长度 单位 (字节)	Value长度 单位 (字节)
string	10000	15			20
list	Key数量	Key长度	元素/字段数	元素/字段长度	
hash	Key数量	Key长度	元素/字段数	元素/字段长度	Value长度
set	Key数量	Key长度	元素/字段数	元素/字段长度	
zset	Key数量	Key长度	元素/字段数	元素/字段长度	Value长度

开始计算

所需总内存 : 1.14M

- STRING
- LIST
- HASH
- SET
- ZSET



六、最底层的数据结构

1、SDS

// 五种header类型, flags取值为0~4

```
#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
```

4. 跳跃表

4.1、

```
typedef struct zskiplistNode {  
    int key;  
    int value;  
    struct zskiplistLevel {  
        struct zskiplistNode *forward;  
    } level[1];  
} zskiplistNode;
```

4.2

```
typedef struct zskiplist {  
    struct zskiplistNode *header;  
    int level;  
} zskiplist;
```

了解redis的内存模型，对优化redis内存占用有很大帮助:

(1) 利用jemalloc特性进行优化:

如上个例子所述，一个key，8字节，所以SDS(key)需要8+9=17个字节，jemalloc会分配32字节的内存块;但是如果把key长度设置为7个字节，那么SDS(key)需要7+9=16个字节，jemalloc会分配16字节，整整缩小了一半;

(2) 使用整型/长整型:

int类型 (8字节) 存储来代替字符串，可以节省更多空间

(3) 共享对象:

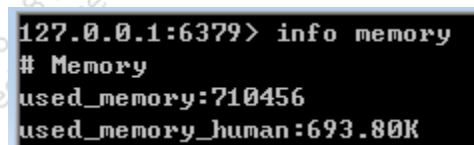
可以减少对象的创建 (同时减少了redisObject的创建)，节省内存空间。目前redis中的共享对象只包括10000个整数 (0-9999)；可以通过调整REDIS_SHARED_INTEGERS参数提高共享对象的个数；例如将REDIS_SHARED_INTEGERS调整到20000，则0-19999之间的对象都可以共享。

(4) 避免过度设计:

需要从实际的业务角度出发，从内存空间、设计复杂度来做权衡；如果数据量较小，通过节省内存来加大代码复杂度其实并不是很划算；比如上个例子中的100000个key，优化效果可能就能节省几M的效果，但是如果数量较大，上亿或者千万级别数据量，优化就很有必要了；

八、本地验证

1. set 命令前



```
127.0.0.1:6379> info memory  
# Memory  
used_memory:710456  
used_memory_human:693.80K
```

2. set命令后

```
127.0.0.1:6379> set a_b_c a_b_c
OK
```

```
127.0.0.1:6379> info memory
# Memory
used_memory:710528
used_memory_human:693.88K
```

3. 差距约 $693.88K - 693.80K = 0.08K = 80 \text{ Byte}$