看看string

1. 简单动态字符串

Redis没有使用传统的C语言模式的字符串表示（以空字符结尾的字符数组），而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用作Redis的默认字符串表示。

在Redis里，C字符串只会作为字符串字面量（string literal）用在一些无须对字符串值进行修改的地方，比如打印日志。

当Redis需要的不仅仅是一个字符串字面变量，而是一个可以被修改的字符串值，Redis会使用SDS来表示字符串的值，比如在Redis的数据库里面，包含字符串值的键值对在底层都是由SDS实现的。，

查看Redis原码可知：

```
typedef char *sds;
```

看到这个定义，有的人可能会说，sds不就是char*嘛，有什么了不起的。设计者为了与C语言保持兼容性，因此他们的定义都是一样的。在有些情况下，需要传入一个C语言的字符串的地方，也的确可以传入一个sds。但是sds是Binary Safe（只关心二进制化的字符串,不关心具体格式.只会严格的按照二进制的数据存取。不会妄图已某种特殊格式解析数据。）的，它可以群出任意二进制数据，不像C语言字符串那样以'\0'表示字符串的结束，那么这个sds结构它必然就会有一个长度字段。查看redis原码，我们可以发现 sds里面有个sds-header结构，里面存储这字符串的相关信息，如下图所示：

```


/* Note: sdshdr5 is never used, we just access the flags byte directly.
 * However is here to document the layout of type 5 SDS strings. */
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 {
    uint64_t len; /* used */
    uint64_t alloc; /* excluding the header and null terminator */
    unsigned char flags; /* 3 lsb of type, 5 unused bits */
    char buf[];
};

```

```
 __attribute__ ((packed)) 的作用就是告诉编译器取消结构在编译过程中的优化对齐,按照实际占用字节数进行对齐，是GCC特有的语法。
```

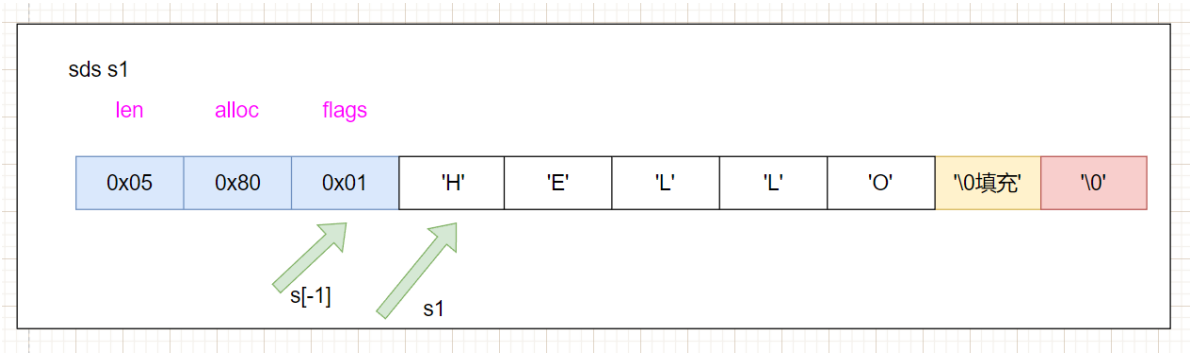
通过查看原码我们发现sds一共有5种类型的header，sdshdr5比较特殊我们暂不考虑它，redis设计者设计这么多类型的原因是为了更好利用内存、尽量少占用内存空间。

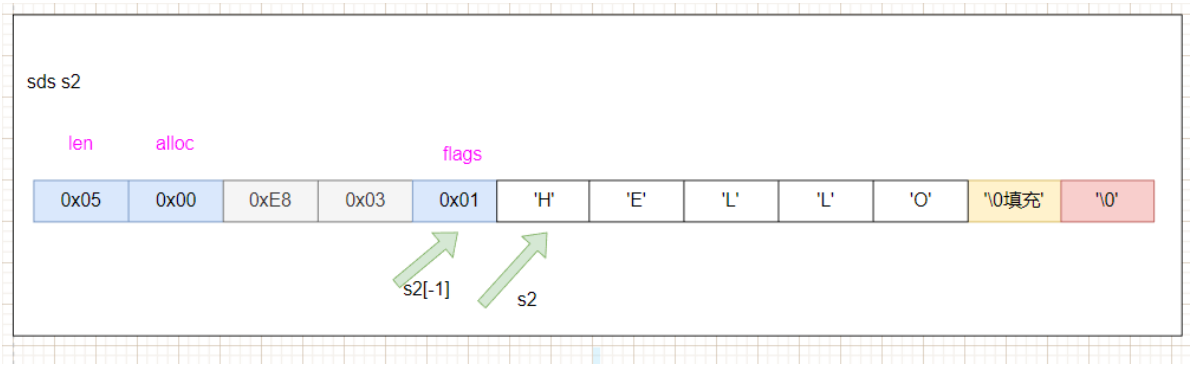
一个sds字符串的完整结构，由在内存地址前后相邻的两部分组成：

- 一个sds-header。该部分通常包括以下三个字段：
    1. len：表示字符串的真正长度（不包括NULL结束符在内）
    2. alloc：表示字符串的最大容量（不包括最后多余的字符）
    3. flags：一个字节。最低三位的bit用来表示header的类型。header的类型由宏变量定义

```
#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
```

- 一个字符数组。这个字符串数组的长度等于最大容量+1。真正优先哦的字符串数据，其长度通常小于最大容量。在真正的字符串数据之后，其空余未用的字节（一般是以字节0填充），允许在不重新分配内存的前提下让字符串数据向后做有效的扩展。在真正的字符串数据后，还有个NULL结束符，即ASCII码为0的'\n'字符，此目的是为了和传统C字符兼容。





上面两张图片是sds的内部结构的例子。一个为sds-8，下面的那个为sds-16，结合原码我们来进行分析分析

sds字符指针，在本图中为s1和s2就是指向真实数据（字符数组）开始的位置，而header是位于内存中地址较低的地方。

```
#define SDS_TYPE_MASK 7
#define SDS_TYPE_BITS 3
#define SDS_HDR_VAR(T,s) struct sdshdr##T *sh = (void*)((s)-(sizeof(struct sdshdr##T)));
#define SDS_HDR(T,s) ((struct sdshdr##T *)((s)-(sizeof(struct sdshdr##T))))
#define SDS_TYPE_5_LEN(f) ((f)>>SDS_TYPE_BITS)
```

SDS_HDR(T,s) ((struct sdshdr##T *)((s)-(sizeof(struct sdshdr##T))))该函数从sds字符串获得header起始位置的指针，SDS_HDR(8,s1)代表s1的header的指针，SDS_HDR(16,s2)代表的是s2的header指针。

我们要使用SDS_HDR之前，首先得要判断sds是什么类型的，flages内刚好就保存了sds-header信息，s[-1]取出来，并与SDS_TYPE_MASK做&运算得出其sds对应的header类型，具体步骤如下:

```
 unsigned char flags = s[-1];
 flags&SDS_TYPE_MASK;
// s1
//0000 0001
//0000 0111
//0000 0001 == define SDS_TYPE_8  1
//0000 0010
//0000 0111
//0000 0010 == define SDS_TYPE_16 2
```

有了header指针，我们就可以轻松的获取到里面的值了

- s1的header中，len的值为0x06，表示字符串数据长度为6；alloc的长度为0x80，表示字符串数组最大的容量为128
- s2的header中，len的值为0x0006，表示字符串数据长度为6；alloc的长度为0x03E8，表示字符串数组最大的容量为1000（小端存储）

注意细节

1. 在各个header的定义中最后有一个char buf[]。我们注意到这是一个没有指明长度的字符数组，这是C语言中定义字符数组的一种特殊写法，称为柔性数组（flexible array member），**只能定义在一个结构体的最后一个字段上。**它在这里只是起到一个标记的作用，表示在flags字段后面就是一个字符数组，或者说，它指明了紧跟在flags字段后面的这个字符数组在结构体中的偏移位置。而程序在为header分配的内存的时候，它并不占用内存空间。如果计算sizeof(struct sdshdr16)的值，那么结果是5个字节，其中没有buf字段。
2. sdshdr5与其它几个header结构不同，它不包含alloc字段，而长度使用flags的高5位来存储。因此，它不能为字符串分配空余空间。如果字符串需要动态增长，那么它就必然要重新分配内存才行。所以说，这种类型的sds字符串更适合存储静态的短字符串（长度小于32）。

至此，我们非常清楚地看到了：sds字符串的header，其实隐藏在真正的字符串数据的前面（低地址方向）。这样的一个定义，有如下几个好处：

1. header和数据相邻，而不用分成两块内存空间来单独分配。这有利于减少内存碎片，提高存储效率（memory efficiency）。

2. 虽然header有多个类型，但sds可以用统一的char *来表达。且它与传统的C语言字符串保持类型兼容。如果一个sds里面存储的是可打印字符串，那么我们可以直接把它传给C函数，比如使用strcmp比较字符串大小，或者使用printf进行打印。

1.1. sds的相关函数

```
sdslen(const sds s): 获取sds字符串长度。
sdssetlen(sds s, size_t newlen): 设置sds字符串长度。
sdsinclen(sds s, size_t inc): 增加sds字符串长度。
sdsalloc(const sds s): 获取sds字符串容量。
sdssetalloc(sds s, size_t newlen): 设置sds字符串容量。
sdsavail(const sds s): 获取sds字符串空余空间（即alloc - len）。
sdsHdrSize(char type): 根据header类型得到header大小。
sdsReqType(size_t string_size): 根据字符串数据长度计算所需要的header类型。
```

```
static inline size_t sdslen(const sds s) {
    unsigned char flags = s[-1];
    switch(flags&SDS_TYPE_MASK) {
        case SDS_TYPE_5:
            return SDS_TYPE_5_LEN(flags);
        case SDS_TYPE_8:
            return SDS_HDR(8,s)->len;
        case SDS_TYPE_16:
            return SDS_HDR(16,s)->len;
        case SDS_TYPE_32:
            return SDS_HDR(32,s)->len;
        case SDS_TYPE_64:
            return SDS_HDR(64,s)->len;
    }
    return 0;
}
static inline char sdsReqType(size_t string_size) {
    if (string_size < 1<<5)
        return SDS_TYPE_5;
    if (string_size < 1<<8)
        return SDS_TYPE_8;
    if (string_size < 1<<16)
        return SDS_TYPE_16;
#if (LONG_MAX == LLONG_MAX)
    if (string_size < 1ll<<32)
        return SDS_TYPE_32;
    return SDS_TYPE_64;
```

```
#else
    return SDS_TYPE_32;
#endif
}
```

跟前面的分析类似，sdslen先用s[-1]向低地址方向偏移1个字节，得到flags；然后与SDS_TYPE_MASK进行按位与，得到header类型；然后根据不同的header类型，调用SDS_HDR得到header起始指针，进而获得len字段。

通过sdsReqType的代码，很容易看到：

1. 长度在0和2^5-1之间，选用SDS_TYPE_5类型的header。
2. 长度在2^5和2^8-1之间，选用SDS_TYPE_8类型的header。
3. 长度在2^8和2^{16}-1之间，选用SDS_TYPE_16类型的header。
4. 长度在2^{16}和2^{32}-1之间，选用SDS_TYPE_32类型的header。
5. 长度大于2^{32}的，选用SDS_TYPE_64类型的header。能表示的最大长度为2^{64}-1。

sds的创建与销毁

```
sds _sdsnewlen(const void *init, size_t initlen, int trymalloc) {
    void *sh;
    sds s;
    char type = sdsReqType(initlen);
    /* Empty strings are usually created in order to append. Use type 8
     * since type 5 is not good at this. */
    if (type == SDS_TYPE_5 && initlen == 0) type = SDS_TYPE_8;
    int hdrlen = sdsHdrSize(type);
    unsigned char *fp; /* flags pointer. */
    size_t usable;

    assert(initlen + hdrlen + 1 > initlen); /* Catch size_t overflow */
    sh = trymalloc?
        s_trymalloc_usable(hdrlen+initlen+1, &usable) :
        s_malloc_usable(hdrlen+initlen+1, &usable);
    if (sh == NULL) return NULL;
    if (init==SDS_NOINIT)
        init = NULL;
    else if (!init)
        memset(sh, 0, hdrlen+initlen+1);
    s = (char*)sh+hdrlen;
    fp = ((unsigned char*)s)-1;
```

```
    usable = usable-hdrlen-1;
    if (usable > sdsTypeMaxSize(type))
        usable = sdsTypeMaxSize(type);
    switch(type) {
        case SDS_TYPE_5: {
            *fp = type | (initlen << SDS_TYPE_BITS);
            break;
        }
        case SDS_TYPE_8: {
            SDS_HDR_VAR(8,s);
            sh->len = initlen;
            sh->alloc = usable;
            *fp = type;
            break;
        }
        case SDS_TYPE_16: {
            SDS_HDR_VAR(16,s);
            sh->len = initlen;
            sh->alloc = usable;
            *fp = type;
            break;
        }
        case SDS_TYPE_32: {
            SDS_HDR_VAR(32,s);
            sh->len = initlen;
            sh->alloc = usable;
            *fp = type;
            break;
        }
        case SDS_TYPE_64: {
            SDS_HDR_VAR(64,s);
            sh->len = initlen;
            sh->alloc = usable;
            *fp = type;
            break;
        }
    }
    if (initlen && init)
        memcpy(s, init, initlen);
    s[initlen] = '\0';
    return s;
}
sds sdsempty(void) {
    return sdsnewlen("",0);
}
```

```
68
69  /* Create a new sds string starting from a null terminated C
    string. */
70  sds sdsnew(const char *init) {
71      size_t initlen = (init == NULL) ? 0 : strlen(init);
72      return sdsnewlen(init, initlen);
73  }
74  /* Free an sds string. No operation is performed if 's' is
    NULL. */
75  void sdsfree(sds s) {
76      if (s == NULL) return;
77      s_free((char*)s-sdsHdrSize(s[-1]));
78  }
```

sdsnewlen创建一个长度为initlen的sds字符串，并使用init指向的字符数组（任意二进制数据）来初始化数据。如果init为NULL，那么使用全0来初始化数据。它的实现中，我们需要注意的是：

- 如果要创建一个长度为0的空字符串，那么不使用SDS_TYPE_5类型的header，而是转而使用SDS_TYPE_8类型的header。这是因为创建的空字符串一般接下来的操作很可能是追加数据，但SDS_TYPE_5类型的sds字符串不适合追加数据（会引发内存重新分配）。
- 需要的内存空间一次性进行分配，其中包含三部分：header、数据、最后的多余字节（hdrlen+initlen+1）。
- 初始化的sds字符串数据最后会追加一个NULL结束符（s[initlen] = '\0'）。

关于sdsfree，需要注意的是：内存要整体释放，所以要先计算出header起始指针，把它传给s_free函数。这个指针也正是在sdsnewlen中调用s_malloc返回的那个地址。

```
1   /* Append the specified binary-safe string pointed by 't' of
    'len' bytes to the
2    * end of the specified sds string 's'.
3    *
4    * After the call, the passed sds string is no longer valid
    and all the
5    * references must be substituted with the new pointer
    returned by the call.
6    */
7   sds sdscatlen(sds s, const void *t, size_t len) {
8       size_t curlen = sdslen(s);
9
10      s = sdsMakeRoomFor(s,len);
11      if (s == NULL) return NULL;
12      memcpy(s+curlen, t, len);
```

```
13      sdssetlen(s, curlen+len);
14      s[curlen+len] = '\0';
15      return s;
16  }
17
18  /* Append the specified null terminated C string to the sds
    string 's'.
19   *
20   * After the call, the passed sds string is no longer valid
    and all the
21   * references must be substituted with the new pointer
    returned by the call. */
22  sds sdscat(sds s, const char *t) {
23      return sdscatlen(s, t, strlen(t));
24  }
25
26  /* Append the specified sds 't' to the existing sds 's'.
27   *
28   * After the call, the modified sds string is no longer valid
    and all the
29   * references must be substituted with the new pointer
    returned by the call. */
30  sds sdscatsds(sds s, const sds t) {
31      return sdscatlen(s, t, sdslen(t));
32  }
33  /* Enlarge the free space at the end of the sds string so that
    the caller
34   * is sure that after calling this function can overwrite up
    to addlen
35   * bytes after the end of the string, plus one more byte for
    nul term.
36   * If there's already sufficient free space, this function
    returns without any
37   * action, if there isn't sufficient free space, it'll
    allocate what's missing,
38   * and possibly more:
39   * When greedy is 1, enlarge more than needed, to avoid need
    for future reallocs
40   * on incremental growth.
41   * When greedy is 0, enlarge just enough so that there's free
    space for 'addlen'.
42   *
43   * Note: this does not change the *length* of the sds string
    as returned
44   * by sdslen(), but only the free buffer space we have.
```

```
 */

sds _sdsMakeRoomFor(sds s, size_t addlen, int greedy) {
    void *sh, *newsh;
    size_t avail = sdsavail(s);
    size_t len, newlen, reqlen;
    char type, oldtype = s[-1] & SDS_TYPE_MASK;
    int hdrlen;
    size_t usable;

    /* Return ASAP if there is enough space left. */
    if (avail >= addlen) return s;

    len = sdslen(s);
    sh = (char*)s-sdsHdrSize(oldtype);
    reqlen = newlen = (len+addlen);
    assert(newlen > len);   /* Catch size_t overflow */
    if (greedy == 1) {
        if (newlen < SDS_MAX_PREALLOC)
            newlen *= 2;
        else
            newlen += SDS_MAX_PREALLOC;
    }

    type = sdsReqType(newlen);

    /* Don't use type 5: the user is appending to the string and type 5 is
     * not able to remember empty space, so sdsMakeRoomFor() must be called
     * at every appending operation. */
    if (type == SDS_TYPE_5) type = SDS_TYPE_8;

    hdrlen = sdsHdrSize(type);
    assert(hdrlen + newlen + 1 > reqlen);  /* Catch size_t overflow */
    if (oldtype==type) {
        newsh = s_realloc_usable(sh, hdrlen+newlen+1, &usable);
        if (newsh == NULL) return NULL;
        s = (char*)newsh+hdrlen;
    } else {
        /* Since the header size changes, need to move the string forward,
         * and can't use realloc */
```

```
85          newsh = s_malloc_usable(hdrlen+newlen+1, &usable);
86          if (newsh == NULL) return NULL;
87          memcpy((char*)newsh+hdrlen, s, len+1);
88          s_free(sh);
89          s = (char*)newsh+hdrlen;
90          s[-1] = type;
91          sdssetlen(s, len);
92      }
93      usable = usable-hdrlen-1;
94      if (usable > sdsTypeMaxSize(type))
95          usable = sdsTypeMaxSize(type);
96      sdssetalloc(s, usable);
97      return s;
98  }
99
```

sdscatlen将t指向的长度为len的任意二进制数据追加到sds字符串s的后面。本文开头演示的string的append命令，内部就是调用sdscatlen来实现的。

在sdscatlen的实现中，先调用sdsMakeRoomFor来保证字符串s有足够的空间来追加长度为len的数据。sdsMakeRoomFor可能会分配新的内存，也可能不会。

sdsMakeRoomFor是sds实现中很重要的一个函数。关于它的实现代码，我们需要注意的是：

- 如果原来字符串中的空余空间够用（avail >= addlen），那么它什么也不做，直接返回。
- 如果需要分配空间，它会比实际请求的要多分配一些，以防备接下来继续追加。它在字符串已经比较长的情况下要至少多分配SDS_MAX_PREALLOC个字节，这个常量在sds.h中定义为(1024*1024)=1MB。
- 按分配后的空间大小，可能需要更换header类型（原来header的alloc字段太短，表达不了增加后的容量）。
- 如果需要更换header，那么整个字符串空间（包括header）都需要重新分配（s_malloc），并拷贝原来的数据到新的位置。
- 如果不需要更换header（原来的header够用），那么调用一个比较特殊的s_realloc，试图在原来的地址上重新分配空间。s_realloc的具体实现得看Redis编译的时候选用了哪个allocator（在Linux上默认使用jemalloc）。但不管是哪个realloc的实现，它所表达的含义基本是相同的：它尽量在原来分配好的地址位置重新分配，如果原来的地址位置有足够的空余空间完成重新分配，那么它返回的新地址与传入的旧地址相同；否则，它分配新的地址块，并进行数据搬迁。参见http://man.cx/realloc。
- 从sdscatlen的函数接口，我们可以看到一种使用模式：调用它的时候，传入一个旧的sds变量，然后它返回一个新的sds变量。由于它的内部实现可能会造成地址变化，因此调用者在调用完之后，原来旧的变量就失效了，而都

应该用新返回的变量来替换。不仅仅是sdscatlen函数，sds中的其它函数（比如sdscpy、sdstrim、sdsjoin等），还有Redis中其它一些能自动扩展内存的数据结构（如ziplist），也都是同样的使用模式。

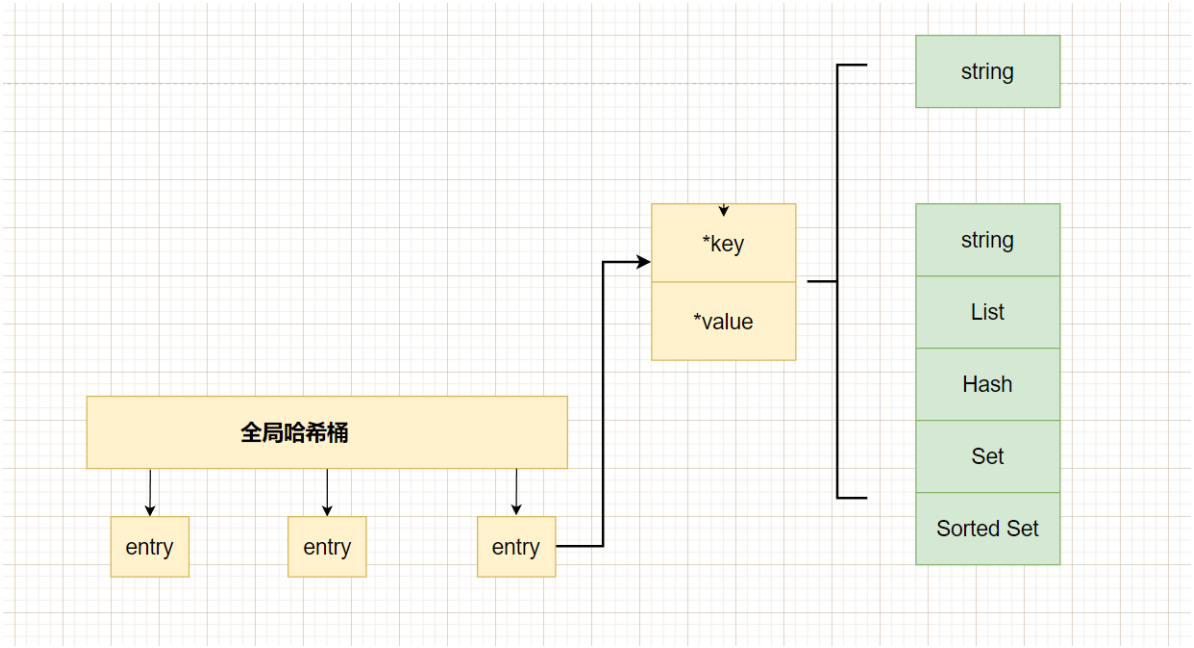
2. 字符串对象

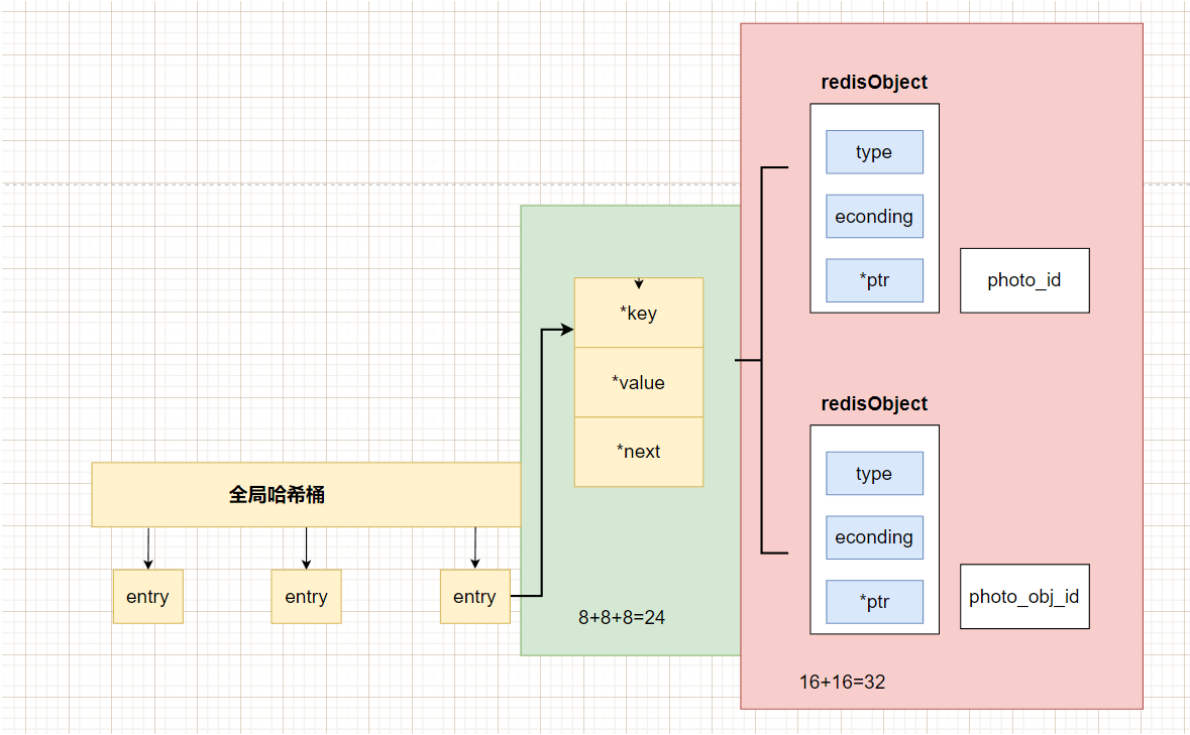
在刚才的案例中，我们保存了 1 亿张图片的信息，用了约 6.4GB 的内存，一个图片 ID 和图片存储对象 ID 的记录平均用了 64 字节。

但问题是，一组图片 ID 及其存储对象 ID 的记录，实际只需要 16 字节就可以了。我们来分析一下。图片 ID 和图片存储对象 ID 都是 10 位数，我们可以用两个 8 字节的Long 类型表示这两个 ID。因为 8 字节的 Long 类型最大可以表示 2 的 64 次方的数值，所以肯定可以表示 10 位数。但是，为什么 String 类型却用了 64 字节呢?

> 其实，除了记录实际数据，String 类型还需要额外的内存空间记录数据长度、空间使用等信息，这些信息也叫作元数据。当实际保存的数据较小时，元数据的空间开销就显得比较大了，有点“喧宾夺主”的意思。

我么大致画一下该对象在Redis实例中存储的示意图：





我们发现开销很大。原本只需要16b现在需要64b可怕。

下面，我将为你简单的介绍介绍string对象的存储方式。

2.1. RedisObject对象

在介绍string对象存储方式时，我们先来看一看Redis所有对象的保存方式吧！

```
#define LRU_BITS 24
struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                            * LFU data (least significant 8 bits frequency
                            * and most significant 16 bits access time). */
    int refcount;
    void *ptr;
};
```

type： 占4个比特位，表示对象的类型，有五种类型。当我们执行type命令时，便是通过type字段获取对象的类型。

```
* The actual Redis Object */
#define OBJ_STRING 0    /* String object. */
#define OBJ_LIST 1      /* List object. */
#define OBJ_SET 2       /* Set object. */
#define OBJ_ZSET 3      /* Sorted set object. */
#define OBJ_HASH 4      /* Hash object. */ 100
```

那我们先使用一下type吧！！！

```
127.0.0.1:6379[1]> set name ty
OK
127.0.0.1:6379[1]> type name
string
127.0.0.1:6379[1]> LPUSH list 1 2 3 4
(integer) 4
127.0.0.1:6379[1]> type list
list
127.0.0.1:6379[1]> SADD members a b c d
(integer) 4
127.0.0.1:6379[1]> type members
set
127.0.0.1:6379[1]> ZADD score 99 tom 100 liming
(integer) 2
127.0.0.1:6379[1]> type score
zset
127.0.0.1:6379[1]> hset hashtable name wyf
(integer) 1
127.0.0.1:6379[1]> hset hashtable age  18
(integer) 1
127.0.0.1:6379[1]> type hashtable
hash
```

encoding： 占4个比特位，表示对象使用哪种编码，redis会根据不同的场景使用不同的编码，大大提高了redis的灵活性和效率。

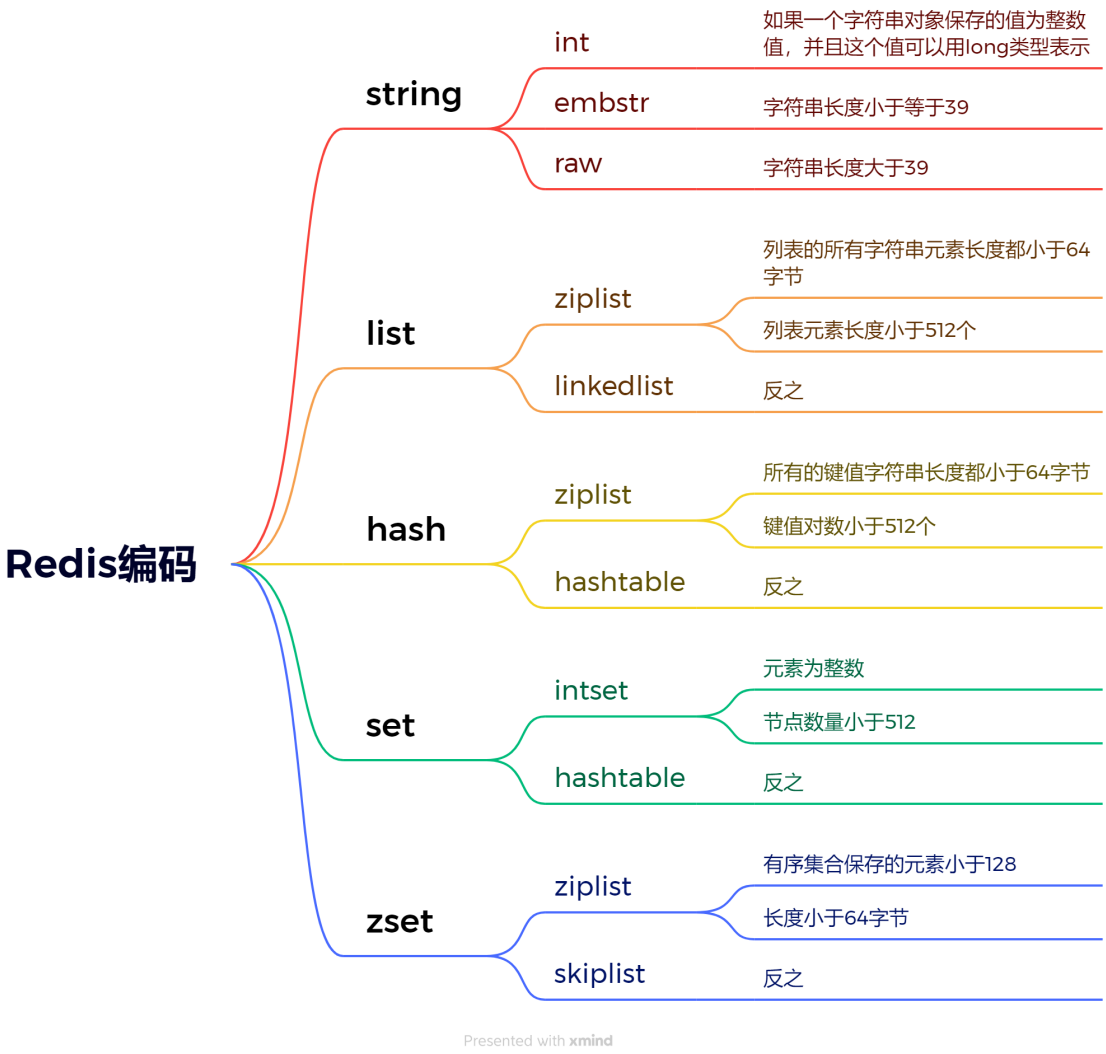
```
/* Objects encoding. Some kind of objects like Strings and Hashes can be
 * internally represented in multiple ways. The 'encoding' field of the object
 * is set to one of this fields for this object. */
#define OBJ_ENCODING_RAW 0     /* Raw representation */
#define OBJ_ENCODING_INT 1     /* Encoded as integer */
#define OBJ_ENCODING_HT 2      /* Encoded as hash table */
```

```
#define OBJ_ENCODING_ZIPMAP 3  /* No longer used: old hash encoding. */
#define OBJ_ENCODING_LINKEDLIST 4 /* No longer used: old list encoding. */
#define OBJ_ENCODING_ZIPLIST 5 /* No longer used: old list/hash/zset encoding. */
#define OBJ_ENCODING_INTSET 6  /* Encoded as intset */
#define OBJ_ENCODING_SKIPLIST 7  /* Encoded as skiplist */
#define OBJ_ENCODING_EMBSTR 8  /* Embedded sds string encoding */
#define OBJ_ENCODING_QUICKLIST 9 /* Encoded as linked list of listpacks */
#define OBJ_ENCODING_STREAM 10 /* Encoded as a radix tree of listpacks */
#define OBJ_ENCODING_LISTPACK 11 /* Encoded as a listpack */
```

1. lru： 占 24 个比特位，记录该对象最后一次被访问的时间。
2. refcount： 对象的引用计数，类似于shared_ptr 智能指针的引用计数，当refcount为0时，释放该对象。
3. ptr: 指向对象具体的底层实现的数据结构。

2.2 Redis不同对象编码规则

下面我将大致的介绍介绍一下Redis不同对象的编码规则，在本篇文章只会详细的介绍string类型的详细实现，其他的类型会在后续的章节中慢慢的介绍给大家。



注意限制可能与redis的版本有关，不同的版本可能会有所差异。

2.3 string

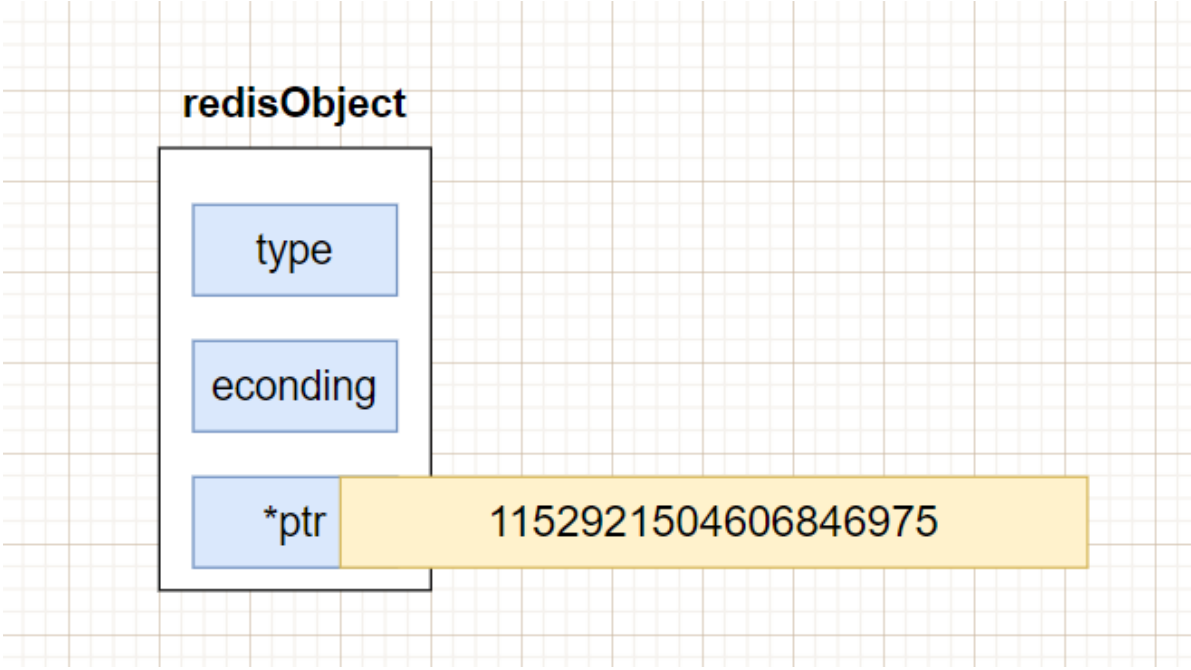
下面我将给你介绍三种字符串在redis对象中的存储方法以及注意细节。

```
set long1 1152921504606846975（未超出）
set long2 11529215046068469751(超出long类型所能存储的最大值)
set long3 1152921504606846975111111111111111111111111111(44)
set long4 11529215046068469751111111111111111111111111111(45)
127.0.0.1:6379[1]> type long1
string
127.0.0.1:6379[1]> type long2
string
127.0.0.1:6379[1]> type long3
string
127.0.0.1:6379[1]> type long4
string
127.0.0.1:6379[1]> OBJECT ENCODING long1
"int"
127.0.0.1:6379[1]> OBJECT ENCODING long2
```

```
"embstr"
127.0.0.1:6379[1]> OBJECT ENCODING long3
"embstr"
127.0.0.1:6379[1]> OBJECT ENCODING long4
"raw"
```

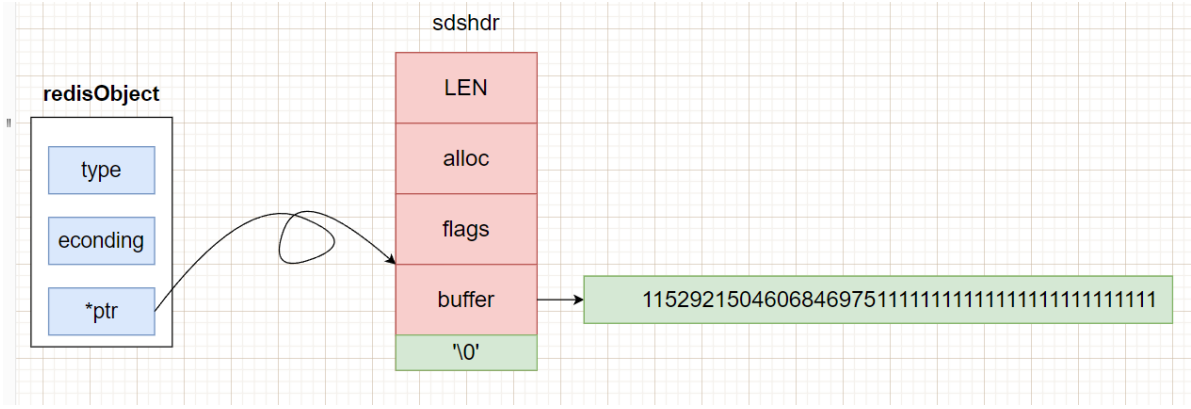
我们可以发现 在7.0.9这个版本中 embstr和raw的界限为44字节，下来我会依次的介绍这三种字符串对象在内存中的结构。

1. int



当存储的整数范围在long类型以内时，那么字符串对象会将整数的值保存在字符串对象结构的ptr属性中(将void*转换为long 并将encoding设置为int)

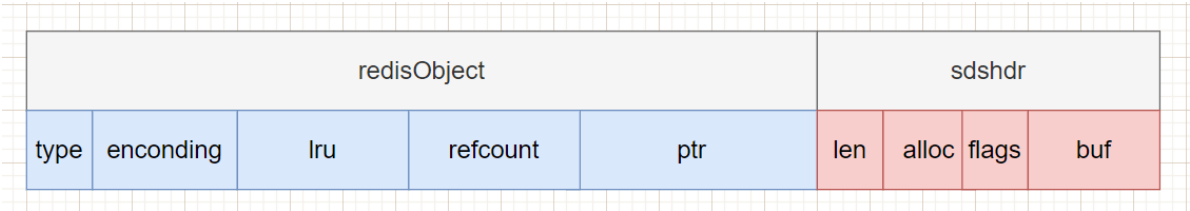
2. raw



当字符串的长度大于44字节时，redis对象会采用该方式存储字符

3. embstr

embstr编码是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编码一样，都是以redisObject结构和sdshdr结构来表示字符串对象，我们先介绍embstr存储方式，稍后我会说明这两种存储方式的不同。



2.4 小结

好了，到现在我们已经介绍完了字符串在redis中存储的方式，接下来我会详细的分析分析这三种结构

上面提到了embstr和raw格式都是采用sdshdr的方式存储，那么这两种结构有什么区别呢，相信大家能很清晰的从上述两幅图看出区别：

- embstr的redisObject和sdshdr两个结构采取顺序存储的方式，也就是说embstr编码通过调用一次内存分配函数来分配一块连续的空间，空间中一次包含redisObject和sdshdr两个结构
- raw则会调用两次内存分配函数来分别创建redisObject和sdshdr两个结构

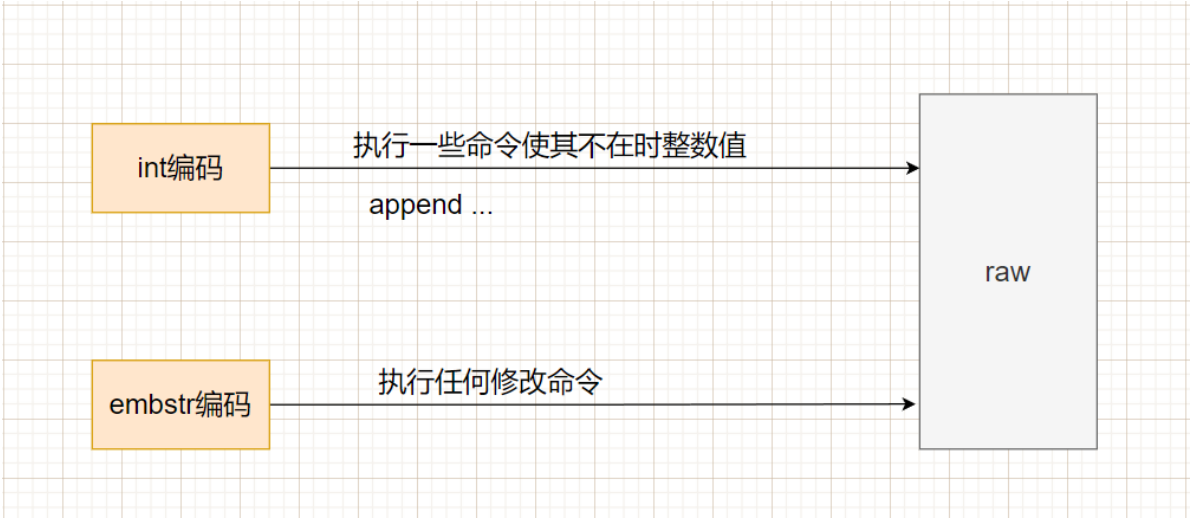
采用embstr编码的字符串对象来保存短字符串值的由以下的好处：

1. embstr编码将创建字符串所需的内存分配次数从raw编码的两次降低为一次
2. 释放embstr编码的字符串对象只需要调用一次内粗释放函数，而释放raw则需要两次
3. 因为embstr编码的字符串对象的所有数据都保存在一块连续的内存里面，所以这种编码的字符串对象比起raw编码的字符串对象能更好的利用缓存带来的优势。

2.5. 编码的转换

int编码的字符串对象和embstr编码的字符串对象在条件满足的情况下，会被转化为raw编码的字符串对象。

以下一张图会向你展示：



embstr编码的字符串比较特殊，redis没有为他提供相应的API（只有int编码和raw编码的字符串有），所以embstr字符串只是可读的，当我们对embstr编码的字符串执行任何修改命令时，程序会先将对象编码从embstr转换为raw，然后在执行。