redis-1.3.7

sds.h/c = simple dynamic String

adlist.h/c = adjacent list

dict.h/c = dictionary / hash table

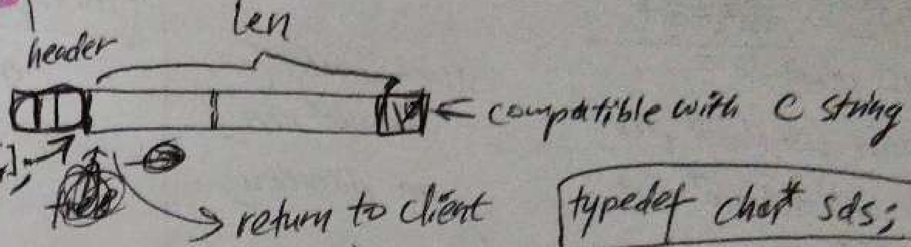
zipmap =

ae.h/c
- anet.h/c
- ae_epoll.h/c
- ae_select.h/c
- ae_kqueue.h/c

redis.h/c = server

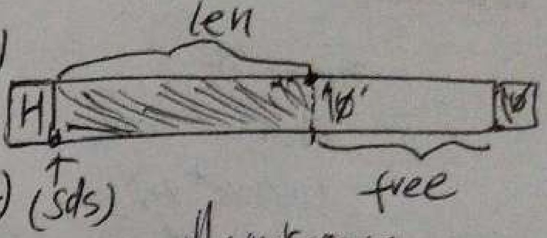
redis-cli.c = client

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



typedef char* sds;

(free: free space in bytes)
(len: not including header and '\0')



⇓ makeroom
(len+extra)*2
⇓
len

```
sdsprintf(sds s, const char* fmt, ...)
va_list ap;
char* buf, *t;
size_t buflen=16;
while(1) {
    buf= malloc(buflen);
    buf[buflen-2]='\0';
    va_start(ap, fmt);
    vsnprintf(buf, buflen, fmt, ap);
    va_end(ap);
    if (buf[buflen-2] != '\0') {
        free(buf);
        buflen *= 2;   // doubling
        continue;
    }
    break;
}
t = sdscat(s, buf);
free(buf);   return t;
```

判断是否溢出.
需要更大空间.

adlist 双线链表

```
struct listNode {
  struct listNode* prev;
  struct listNode* next;
  void* value;
};

struct listIter {
  listNode* next;
  int direction;
};

struct list {
  listNode* head;
  listNode* tail;
  unsigned int len;
};
```

dict.h/c Hash table

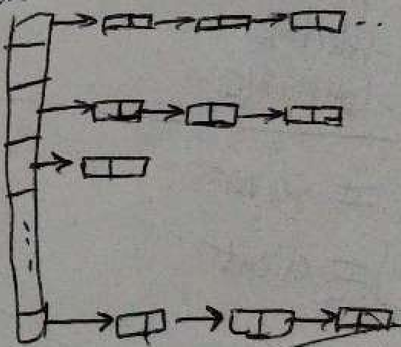
```
struct dictEntry {
    void* key;
    void* val;
    struct dictEntry* next;
};

struct dictType {
    uint32_t (*hashFuction)(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);
};

struct dict {
  dictEntry** table;
  dictType* type;
  unsigned long size;      表slot个数.
  unsigned long sizemask;
  unsigned long used;      结点数
  void* privdata;
};
```

ratio = $\frac{used}{size}$ = 1

⇒ expand.



先生成iter

再由iter获取下个iter.

```
struct dictIterator {
  dict* ht;
  int index;
  dictEntry* entry, *nextEntry;
};
```

这个变量是保存entry→next值.
在iterate过程中 client 有可能
会 delete Node

```
while (1) {
  if (iter->entry == NULL) {
    iter->index++;
    if (iter->index >= iter->ht->size)
       break;
    iter->entry = iter->ht->table
                  [iter->index]
  } else {
    iter->entry = iter->nextEntry
  }
  (if (iter->entry) {
     iter->nextEntry
     = iter->entry->next;
     return iter->entry; }
```

aeEventLoop

```
struct aeEventLoop {
    int maxfd;
    long long timeEventNextId;
    aeFileEvent events[AE_SETSIZE];
    aeFiredEvent fired[AE_SETSIZE];
    aeTimeEvent* timeEventHead;  // 链表
                                     慢
    int stop;
    void* apidata;
    aeBeforeSleepProc* beforesleep;
}
```

aeEventLoop* ← aeCreateEventLoop → aeApiCreate(eventloop) ⟹ dispatch到不同后端

aeApiFree

epoll

malloc

```
struct aeApiState {
    int epfd;    = epoll_create(..);
    struct epollevent events[AESETSIZE];
};
```

1024*10=10k (hardcoded)

```
aeCreateFileEvent (aeEventLoop* eventloop, int fd, int mask,
                   aeFileProc* proc, void* clientData)
```

proc → Event handler

↓

aeApiAddEvent (fd, mask)

mask → 事件类型

```
struct aeFileEvent {   // 事件类型 R/W.
    int mask;
    aeFileProc* rfileProc;
    aeFileProc* wfileProc;
    void* clientData;
};
```

SO_KEEPALIVE → SOL_SOCKET
TCP_NODELAY → IPPROTO_TCP
SO_SNDBUF → SOL_SOCKET

```
aeMain (aeEventLoop* eventloop)
{
    eventloop->stop=0;
    while (!eventloop->stop) {
        if (eventloop->beforesleep != NULL)
            eventloop->beforesleep(eventloop);
        aeProcessEvents(eventloop, AE_ALL_EVENTS);
    }
}
```

O(n) 线性 查找

→ 根据 timer event 来设置 timeout, 调用 aeApiPoll { select, poll, epoll } 都有timeout选项

最临近的 | 距离当前时间之差。没有timer event, 则要么 blocking wait, 要么 timeout=0.

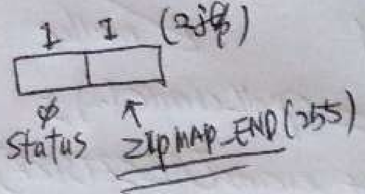
→ 处理 fired events 不同 backend 将就绪事件放入 fired events 中. 统一处理, 调用对应的 process 处理.

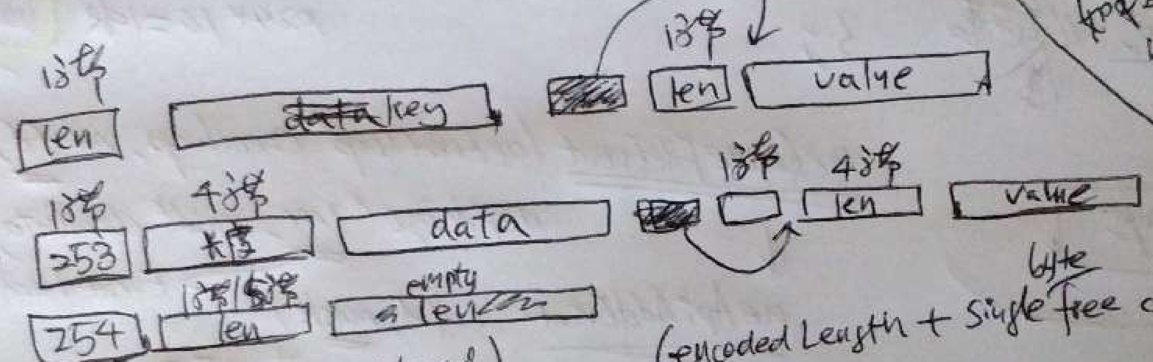
zipmap

String ⇒ String Map data structure optimized for size (O(n))

在key-value 比较少时生 zipmap，较多时，用 hashtable (dict)。

- zipmapNew ⇒





- 长度<253
- 长度>=253
- mark=254

(encoded Length + payload)

(1/5字节)

(encoded Length + single free byte count + payload)

在已有的 empty slot 上放入一个 key/value，(有长度)。如~~[illegible]~~剩余长度 >= 3 + ~~[illegible]~~ 1 + 1 = 5 (key 1字节 value 1字节) 则作为一个 another empty slot 来处理。否则 后面的 key/value，变为 free byte 多[value后面有多少字节]。

"foo" ⇒ "bar", "hello" ⇒ "world"

<status><len>"foo"<len><free>"bar"<len>"hello"<len><free>"world"

<free>是1字节，以 key/value 有被 update，value 长度变小，记录有多少 free bytes

可以方便下次~~再~~ update 成更长的 value.

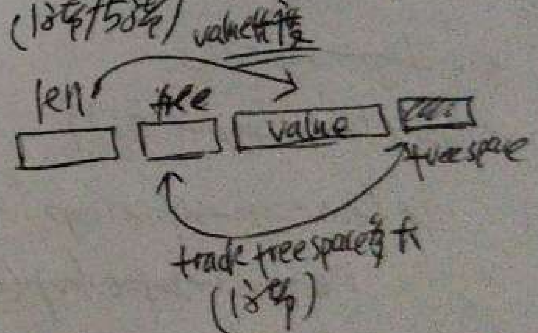
"\x00\x03 foo\x03\x00bar\x05hello\x05\x00 world\xff"

status

free为空

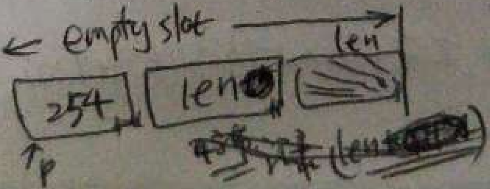
free为空

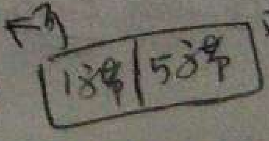
→ end



删除 key/value 之后，留个 empty slot.

254 代表 slot 是 empty 的.





对len的编码